Arm® Architecture Reference Manual

for A-profile architecture
## Release Information

The following releases of this document have been made.

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
<th>Confidentiality</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 April 2013</td>
<td>A.a-1</td>
<td>Confidential-Beta Draft</td>
<td>Beta draft of first issue, limited circulation</td>
</tr>
<tr>
<td>12 June 2013</td>
<td>A.a-2</td>
<td>Confidential-Beta Draft</td>
<td>Second beta draft of first issue, limited circulation</td>
</tr>
<tr>
<td>04 September 2013</td>
<td>A.a</td>
<td>Non-Confidential Beta</td>
<td>Beta release</td>
</tr>
<tr>
<td>24 December 2013</td>
<td>A.b</td>
<td>Non-Confidential Beta</td>
<td>Second beta release</td>
</tr>
<tr>
<td>18 July 2014</td>
<td>A.c</td>
<td>Non-Confidential Beta</td>
<td>Third beta release</td>
</tr>
<tr>
<td>09 October 2014</td>
<td>A.d</td>
<td>Non-Confidential Beta</td>
<td>Fourth beta release</td>
</tr>
<tr>
<td>17 December 2014</td>
<td>A.e</td>
<td>Non-Confidential Beta</td>
<td>Fifth beta release</td>
</tr>
<tr>
<td>25 March 2015</td>
<td>A.f</td>
<td>Non-Confidential Beta</td>
<td>Sixth beta release</td>
</tr>
<tr>
<td>10 July 2015</td>
<td>A.g</td>
<td>Non-Confidential Beta</td>
<td>Seventh beta release</td>
</tr>
<tr>
<td>30 September 2015</td>
<td>A.h</td>
<td>Non-Confidential Beta</td>
<td>Eighth beta release</td>
</tr>
<tr>
<td>28 January 2016</td>
<td>A.i</td>
<td>Non-Confidential Beta</td>
<td>Ninth beta release</td>
</tr>
<tr>
<td>03 June 2016</td>
<td>A.j</td>
<td>Non-Confidential EAC</td>
<td>EAC release</td>
</tr>
<tr>
<td>30 September 2016</td>
<td>A.k</td>
<td>Non-Confidential Armv8.0 EAC</td>
<td>Updated EAC release</td>
</tr>
<tr>
<td>31 March 2017</td>
<td>B.a</td>
<td>Non-Confidential Armv8.1 EAC, v8.2 Beta</td>
<td>Initial release incorporating Armv8.1 and Armv8.2</td>
</tr>
<tr>
<td>26 September 2017</td>
<td>B.b</td>
<td>Non-Confidential Armv8.2 EAC</td>
<td>Initial Armv8.2 EAC release, incorporating SPE</td>
</tr>
<tr>
<td>20 December 2017</td>
<td>C.a</td>
<td>Non-Confidential Armv8.3 EAC</td>
<td>Initial Armv8.3 EAC release</td>
</tr>
<tr>
<td>31 October 2018</td>
<td>D.a</td>
<td>Non-Confidential Armv8.4 EAC</td>
<td>Initial Armv8.4 EAC release</td>
</tr>
<tr>
<td>29 April 2019</td>
<td>D.b</td>
<td>Non-Confidential Armv8.4 EAC</td>
<td>Updated Armv8.4 EAC release incorporating accessibility changes</td>
</tr>
<tr>
<td>05 July 2019</td>
<td>E.a</td>
<td>Non-Confidential Armv8.5 EAC</td>
<td>Initial Armv8.5 EAC release</td>
</tr>
<tr>
<td>20 February 2020</td>
<td>F.a</td>
<td>Non-Confidential Armv8.6 Beta</td>
<td>Initial Armv8.6 Beta release</td>
</tr>
<tr>
<td>31 March 2020</td>
<td>F.b</td>
<td>Non-Confidential Armv8.5 EAC, v8.6 Beta</td>
<td>Armv8.5 EAC release, initial Armv8.6 Beta release</td>
</tr>
<tr>
<td>17 July 2020</td>
<td>F.c</td>
<td>Non-Confidential Armv8.6 EAC</td>
<td>Initial Armv8.6 EAC release</td>
</tr>
<tr>
<td>22 January 2021</td>
<td>G.a</td>
<td>Non-Confidential Armv8.7 EAC</td>
<td>Initial Armv8.7 EAC release</td>
</tr>
<tr>
<td>22 July 2021</td>
<td>G.b</td>
<td>Non-Confidential Armv8.7 EAC</td>
<td>Updated Armv8.7 EAC release</td>
</tr>
<tr>
<td>04 February 2022</td>
<td>H.a</td>
<td>Non-Confidential Armv9 EAC</td>
<td>Initial Armv9 EAC release</td>
</tr>
<tr>
<td>19 August 2022</td>
<td>I.a</td>
<td>Non-Confidential Armv9 EAC release</td>
<td>Updated Armv9 EAC release incorporating BRBE, ETE, and TRBE</td>
</tr>
</tbody>
</table>
Limitations of this issue

This issue of the Arm Architecture Reference Manual contains many improvements and corrections. Validation of this document has identified the following issues that Arm will address in future issues:

- Some diagrams in the register descriptions chapter have long field names that split over several lines. These are not wrong, and the descriptions list the field name correctly.

- The Performance Monitors external register descriptions on page I5-10753 do not include updates to registers or fields that change as a result of the optional 64-bit external access feature.

- Appendix K16 Arm Pseudocode Definition requires further review and update. Since this appendix is informative, rather than being part of the architecture specification, this does not affect the quality status of this release.

- For a list of the known issues in this Manual, please refer to the Known Issues document on https://developer.arm.com/documentation/102105/latest.

Contents

Arm Architecture Reference Manual for A-profile architecture

Preface
About this Manual .......................................................... xviii
Using this Manual ........................................................................................................ xx
Conventions xxvii
Additional reading xxx
Feedback xxxii

Part A Arm Architecture Introduction and Overview

Chapter A1 Introduction to the Arm Architecture
A1.1 About the Arm architecture .......................................................... A1-36
A1.2 Architecture profiles .......................................................... A1-38
A1.3 Arm architectural concepts .......................................................... A1-39
A1.4 Supported data types .......................................................... A1-42
A1.5 Floating-point support .......................................................... A1-55
A1.6 The Arm memory model .......................................................... A1-69

Chapter A2 Armv8-A Architecture Extensions
A2.1 Armv8.0 architecture extensions .......................................................... A2-72
A2.2 Architectural features within Armv8.0 architecture .......................................................... A2-76
A2.3 The Armv8 Cryptographic Extension .......................................................... A2-80
A2.4 The Armv8.1 architecture extension .......................................................... A2-82
A2.5 The Armv8.2 architecture extension .......................................................... A2-86
A2.6 The Armv8.3 architecture extension .......................................................... A2-96
A2.7 The Armv8.4 architecture extension .......................................................... A2-101
A2.8 The Armv8.5 architecture extension .......................................................... A2-107
Chapter C5  The A64 System Instruction Class
C5.1  The System instruction class encoding space ...........................................  C5-654
C5.2  Special-purpose registers ...........................................................................  C5-671
C5.3  A64 System instructions for cache maintenance .........................................  C5-776
C5.4  A64 System instructions for address translation .........................................  C5-841
C5.5  A64 System instructions for TLB maintenance ...........................................  C5-866
C5.6  A64 System instructions for prediction restriction .........................................  C5-1146
C5.7  A64 System instructions for the Branch Record Buffer Extension ................  C5-1159

Chapter C6  A64 Base Instruction Descriptions
C6.1  About the A64 base instructions ...............................................................  C6-1166
C6.2  Alphabetical list of A64 base instructions ..................................................  C6-1169

Chapter C7  A64 Advanced SIMD and Floating-point Instruction Descriptions
C7.1  About the A64 Advanced SIMD and floating-point instructions ...............  C7-2040
C7.2  Alphabetical list of A64 Advanced SIMD and floating-point instructions ......  C7-2042

Chapter C8  SVE Instruction Descriptions
C8.1  About the SVE instructions .......................................................................  C8-2974
C8.2  Alphabetical list of SVE instructions ..........................................................  C8-2975

Chapter C9  SME Instruction Descriptions
C9.1  About the SME instructions .......................................................................  C9-4542
C9.2  Alphabetical list of SME instructions ..........................................................  C9-4543

Part D  The AArch64 System Level Architecture

Chapter D1  The AArch64 System Level Programmers’ Model
D1.1  Exception levels ..........................................................................................  D1-4632
D1.2  Registers for instruction processing and exception handling ....................  D1-4636
D1.3  Exceptions ..................................................................................................  D1-4638
D1.4  Process state, PSTATE ...............................................................................  D1-4670
D1.5  Resets and power domains ........................................................................  D1-4673
D1.6  Mechanisms for entering a low-power state ..............................................  D1-4676
D1.7  Self-hosted debug ......................................................................................  D1-4681
D1.8  Event monitors ...........................................................................................  D1-4683
D1.9  Interprocessing ...........................................................................................  D1-4684

Chapter D2  AArch64 Self-hosted Debug
D2.1  About self-hosted debug ............................................................................  D2-4692
D2.2  The debug exception enable controls ......................................................  D2-4696
D2.3  Routing debug exceptions ........................................................................  D2-4697
D2.4  Enabling debug exceptions from the current Exception level ....................  D2-4699
D2.5  The effect of powerdown on debug exceptions ........................................  D2-4701
D2.6  Summary of the routing and enabling of debug exceptions .......................  D2-4702
D2.7  Pseudocode description of debug exceptions ...........................................  D2-4704
D2.8  Breakpoint Instruction exceptions .............................................................  D2-4705
D2.9  Breakpoint exceptions ...............................................................................  D2-4707
D2.10  Watchpoint exceptions ............................................................................  D2-4726
D2.11  Vector Catch exceptions ..........................................................................  D2-4741
D2.12  Software Step exceptions ........................................................................  D2-4742
D2.13  Synchronization and debug exceptions ..................................................  D2-4755

Chapter D3  AArch64 Self-hosted Trace
D3.1  About self-hosted trace ............................................................................  D3-4758
D3.2  Prohibited regions in self-hosted trace .....................................................  D3-4759
Chapter D4 The Embedded Trace Extension
D4.1 About the Embedded Trace Extension ................................................................. D4-4764
D4.2 Programmers’ model ......................................................................................... D4-4771
D4.3 Trace elements ................................................................................................. D4-4775
D4.4 Instruction and exception classification ............................................................ D4-4791
D4.5 About the ETE trace unit .................................................................................. D4-4800
D4.6 Resource operation ......................................................................................... D4-4843

Chapter D5 ETE Protocol Descriptions
D5.1 About the ETE protocol ..................................................................................... D5-4874
D5.2 Summary list of ETE packets ............................................................................. D5-4877
D5.3 Alphabetical list of ETE packets ....................................................................... D5-4880

Chapter D6 The Trace Buffer Extension
D6.1 About the Trace Buffer Extension ................................................................. D6-5004
D6.2 The trace buffer ............................................................................................... D6-5006
D6.3 Trace buffer management ............................................................................... D6-5018
D6.4 Synchronization and the Trace Buffer Unit ..................................................... D6-5026
D6.5 Trace synchronization and memory barriers ................................................. D6-5031
D6.6 Trace of Speculative execution ....................................................................... D6-5032
D6.7 Trace in Debug state ...................................................................................... D6-5033
D6.8 Detailed synchronization litmus tests ............................................................. D6-5034
D6.9 UNPREDICTABLE behavior .......................................................................... D6-5038

Chapter D7 The AArch64 System Level Memory Model
D7.1 About the memory system architecture ......................................................... D7-5040
D7.2 Address space ................................................................................................. D7-5041
D7.3 Mixed-endian support .................................................................................... D7-5042
D7.4 Cache support ................................................................................................. D7-5043
D7.5 External aborts ............................................................................................... D7-5072
D7.6 Memory barrier instructions ......................................................................... D7-5074
D7.7 Pseudocode description of general memory System instructions ................. D7-5075

Chapter D8 The AArch64 Virtual Memory System Architecture
D8.1 Address translation ......................................................................................... D8-5080
D8.2 Translation process ....................................................................................... D8-5093
D8.3 Translation table descriptor formats ............................................................. D8-5123
D8.4 Memory access control .................................................................................. D8-5132
D8.5 Memory region attributes ............................................................................ D8-5151
D8.6 Other descriptor fields .................................................................................. D8-5158
D8.7 Address tagging ............................................................................................. D8-5162
D8.8 Pointer authentication .................................................................................... D8-5164
D8.9 Virtualization Host Extensions ..................................................................... D8-5168
D8.10 Nested virtualization .................................................................................... D8-5173
D8.11 Memory aborts ............................................................................................ D8-5180
D8.12 Translation Lookaside Buffers .................................................................... D8-5193
D8.13 TLB maintenance ....................................................................................... D8-5197
D8.14 Caches .......................................................................................................... D8-5215
D8.15 Pseudocode description of VMSAv8-64 address translation ......................... D8-5217

Chapter D9 The Memory Tagging Extension
D9.1 Introduction .................................................................................................... D9-5220
D9.2 Allocation Tags ............................................................................................... D9-5221
D9.3 Tag checking .................................................................................................. D9-5223
D9.4 Tagged and Untagged Addresses ................................................................... D9-5224
Chapter D17 AArch64 System Register Descriptions

D17.1 About the AArch64 System registers ............................................................. D17-5546
D17.2 General system control registers ................................................................. D17-5555
D17.3 Debug registers ......................................................................................... D17-6393
D17.4 Trace registers ......................................................................................... D17-6519
D17.5 Performance Monitors registers ............................................................... D17-6740
D17.6 Activity Monitors registers ...................................................................... D17-6816
D17.7 Statistical Profiling Extension registers .................................................... D17-6858
D17.8 Branch Record Buffer Extension registers .............................................. D17-6912
D17.9 RAS registers ......................................................................................... D17-6956
D17.10 Realm Management Extension registers ................................................. D17-7004
D17.11 Generic Timer registers ......................................................................... D17-7014

Part E The AArch32 Application Level Architecture

Chapter E1 The AArch32 Application Level Programmers’ Model

E1.1 About the Application level programmers’ model ...................................... E1-7128
E1.2 The Application level programmers’ model in AArch32 state ................. E1-7129
E1.3 Advanced SIMD and floating-point instructions ...................................... E1-7140
E1.4 About the AArch32 System register interface .......................................... E1-7158
E1.5 Exceptions .................................................................................................. E1-7159

Chapter E2 The AArch32 Application Level Memory Model

E2.1 About the Arm memory model ................................................................. E2-7162
E2.2 Atomicity in the Arm architecture ............................................................ E2-7164
E2.3 Definition of the memory model ............................................................... E2-7168
E2.4 Ordering of translation table walks ......................................................... E2-7187
E2.5 Caches and memory hierarchy ............................................................... E2-7188
E2.6 Alignment support .................................................................................. E2-7193
E2.7 Endian support ......................................................................................... E2-7195
E2.8 Memory types and attributes ................................................................. E2-7198
E2.9 Mismatched memory attributes .............................................................. E2-7208
E2.10 Synchronization and semaphores ......................................................... E2-7211

Part F The AArch32 Instruction Sets

Chapter F1 About the T32 and A32 Instruction Descriptions

F1.1 Format of instruction descriptions ......................................................... F1-7224
F1.2 Standard assembler syntax fields ........................................................... F1-7228
F1.3 Conditional execution ............................................................................ F1-7229
F1.4 Shifts applied to a register ...................................................................... F1-7231
F1.5 Memory accesses .................................................................................. F1-7233
F1.6 Encoding of lists of general-purpose registers and the PC .................... F1-7234
F1.7 General information about the T32 and A32 instruction descriptions ...... F1-7235
F1.8 Additional pseudocode support for instruction descriptions ................. F1-7248
F1.9 Additional information about Advanced SIMD and floating-point instructions .. F1-7249

Chapter F2 The AArch32 Instruction Sets Overview

F2.1 Support for instructions in different versions of the Arm architecture .... F2-7256
F2.2 Unified Assembler Language ................................................................. F2-7257
F2.3 Branch instructions .............................................................................. F2-7259
F2.4 Data-processing instructions .................................................................. F2-7260
F2.5 PSTATE and banked register access instructions .................................. F2-7268
F2.6 Load/store instructions .......................................................................... F2-7269
F2.7 Load/store multiple instructions ............................................................... F2-7272
F2.8 Miscellaneous instructions .................................................................. F2-7273
Chapter F3  T32 Instruction Set Encoding
F3.1  T32 instruction set encoding ................................................................. F3-7296
F3.2  About the T32 Advanced SIMD and floating-point instructions and their encoding  F3-7371

Chapter F4  A32 Instruction Set Encoding
F4.1  A32 instruction set encoding ................................................................. F4-7374
F4.2  About the A32 Advanced SIMD and floating-point instructions and their encoding  F4-7440

Chapter F5  T32 and A32 Base Instruction Set Instruction Descriptions
F5.1  Alphabetical list of T32 and A32 base instruction set instructions ............... F5-7442
F5.2  Encoding and use of banked register transfer instructions ....................... F5-8160

Chapter F6  T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions
F6.1  Alphabetical list of Advanced SIMD and floating-point instructions .......... F6-8166

Part G  The AArch32 System Level Architecture

Chapter G1  The AArch32 System Level Programmers’ Model
G1.1  About the AArch32 System level programmers’ model ......................... G1-8898
G1.2  Exception levels .......................................................... G1-8899
G1.3  Exception terminology .......................................................... G1-8900
G1.4  Execution state .......................................................... G1-8902
G1.5  Instruction Set state .......................................................... G1-8904
G1.6  Security state .......................................................... G1-8905
G1.7  Security state, Exception levels, and AArch32 execution privilege .......... G1-8908
G1.8  Virtualization .......................................................... G1-8910
G1.9  AArch32 state PE modes, and general-purpose and Special-purpose registers G1-8912
G1.10  Process state, PSTATE .......................................................... G1-8921
G1.11  Instruction set states .......................................................... G1-8927
G1.12  Handling exceptions that are taken to an Exception level using AArch32 ... G1-8929
G1.13  Routing of aborts taken to AArch32 state ........................................ G1-8948
G1.14  Exception return to an Exception level using AArch32 ......................... G1-8951
G1.15  Asynchronous exception behavior for exceptions taken from AArch32 state .. G1-8956
G1.16  AArch32 state exception descriptions ........................................... G1-8964
G1.17  Reset into AArch32 state .......................................................... G1-8986
G1.18  Mechanisms for entering a low-power state ....................................... G1-8988
G1.19  The AArch32 System register interface ............................................ G1-8993
G1.20  Advanced SIMD and floating-point support ....................................... G1-8996
G1.21  Configurable instruction enables and disables, and trap controls .......... G1-9001

Chapter G2  AArch32 Self-hosted Debug
G2.1  About self-hosted debug .......................................................... G2-9038
G2.2  The debug exception enable controls ........................................... G2-9042
G2.3  Routing debug exceptions .......................................................... G2-9043
G2.4  Enabling debug exceptions from the current Privilege level and Security state G2-9045
G2.5  The effect of powerdown on debug exceptions ................................... G2-9047
G2.6  Summary of permitted routing and enabling of debug exceptions .......... G2-9048
G2.7  Pseudocode description of debug exceptions ..................................... G2-9050
G2.8  Breakpoint Instruction exceptions ........................................... G2-9051
G2.9  Breakpoint exceptions .......................................................... G2-9054
# External Debug

## Part H

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Sections</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H1</strong></td>
<td>About External Debug</td>
<td>H1.1 Introduction to external debug</td>
<td>H1-10228</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H1.2 External debug</td>
<td>H1-10229</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H1.3 Required debug authentication</td>
<td>H1-10230</td>
</tr>
<tr>
<td><strong>H2</strong></td>
<td>Debug State</td>
<td>H2.1 About Debug state</td>
<td>H2-10232</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H2.2 Halting the PE on debug events</td>
<td>H2-10233</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H2.3 Entering Debug state</td>
<td>H2-10239</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H2.4 Behavior in Debug state</td>
<td>H2-10242</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H2.5 Exiting Debug state</td>
<td>H2-10271</td>
</tr>
<tr>
<td><strong>H3</strong></td>
<td>Halting Debug Events</td>
<td>H3.1 Introduction to Halting debug events</td>
<td>H3-10274</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H3.2 Halting Step debug events</td>
<td>H3-10276</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H3.3 Halt Instruction debug event</td>
<td>H3-10286</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H3.4 Exception Catch debug event</td>
<td>H3-10287</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H3.5 External Debug Request debug event</td>
<td>H3-10291</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H3.6 OS Unlock Catch debug event</td>
<td>H3-10292</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H3.7 Reset Catch debug events</td>
<td>H3-10293</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H3.8 Software Access debug event</td>
<td>H3-10294</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H3.9 Synchronization and Halting debug events</td>
<td>H3-10295</td>
</tr>
<tr>
<td><strong>H4</strong></td>
<td>The Debug Communication Channel and Instruction Transfer Register</td>
<td>H4.1 Introduction</td>
<td>H4-10298</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H4.2 DCC and ITR registers</td>
<td>H4-10299</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H4.3 DCC and ITR access modes</td>
<td>H4-10302</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H4.4 Flow control of the DCC and ITR registers</td>
<td>H4-10306</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H4.5 Synchronization of DCC and ITR accesses</td>
<td>H4-10309</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H4.6 Interrupt-driven use of the DCC</td>
<td>H4-10314</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H4.7 Pseudocode description of the operation of the DCC and ITR registers</td>
<td>H4-10315</td>
</tr>
<tr>
<td><strong>H5</strong></td>
<td>The Embedded Cross-Trigger Interface</td>
<td>H5.1 About the Embedded Cross-Trigger</td>
<td>H5-10318</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H5.2 Basic operation on the ECT</td>
<td>H5-10320</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H5.3 Cross-triggers on a PE in an Arm A-profile implementation</td>
<td>H5-10324</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H5.4 Description and allocation of CTI triggers</td>
<td>H5-10325</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H5.5 CTI registers programmers’ model</td>
<td>H5-10329</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H5.6 Examples</td>
<td>H5-10330</td>
</tr>
<tr>
<td><strong>H6</strong></td>
<td>Debug Reset and Powerdown Support</td>
<td>H6.1 About Debug over powerdown</td>
<td>H6-10334</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H6.2 Power domains and debug</td>
<td>H6-10335</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H6.3 Core power domain power states</td>
<td>H6-10336</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H6.4 Powerup request mechanism</td>
<td>H6-10338</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H6.5 Emulating low-power states</td>
<td>H6-10340</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H6.6 Debug OS Save and Restore sequences</td>
<td>H6-10342</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H6.7 Reset and debug</td>
<td>H6-10348</td>
</tr>
<tr>
<td><strong>H7</strong></td>
<td>The PC Sample-based Profiling Extension</td>
<td>H7.1 About the PC Sample-based Profiling Extension</td>
<td>H7-10352</td>
</tr>
<tr>
<td><strong>H8</strong></td>
<td>About the External Debug Registers</td>
<td>H8.1 Relationship between external debug and System registers</td>
<td>H8-10356</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H8.2 Endianness and supported access sizes</td>
<td>H8-10357</td>
</tr>
</tbody>
</table>
Appendix K2  Recommendations for Reporting Memory Attributes on an Interconnect
   K2.1  Arm recommendations for reporting memory attributes on an interconnect  .  K2-11604

Appendix K3  ETE Recommended Configurations
   K3.1  Configurations ................................................................. K3-11606

Appendix K4  ETE and TRBE Software Usage Examples
   K4.1  Trace analyzer ............................................................. K4-11608
   K4.2  ETE programming ....................................................... K4-11654
   K4.3  Trace examples ............................................................ K4-11660
   K4.4  Differences between ETM and ETE ............................... K4-11671
   K4.5  Context switching ....................................................... K4-11673
   K4.6  Controlling generation of trace buffer management events  .  K4-11675

Appendix K5  Stages of execution
   K5.1  Stages of execution without the Transactional Memory Extension (TME)  .  K5-11678
   K5.2  Stages of execution with the TME .................................... K5-11679

Appendix K6  Recommended External Debug Interface
   K6.1  About the recommended external debug interface .................. K6-11682
   K6.2  PMUEVENT bus ............................................................ K6-11686
   K6.3  Recommended authentication interface ............................ K6-11687
   K6.4  Management registers and CoreSight compliance ................ K6-11688

Appendix K7  Additional Information for Implementations of the Generic Timer
   K7.1  Providing a complete set of features in a system level implementation .... K7-11702
   K7.2  Gray-count scheme for timer distribution scheme ................ K7-11704

Appendix K8  Legacy Instruction Syntax for AArch32 Instruction Sets
   K8.1  Legacy Instruction Syntax ............................................... K8-11706

Appendix K9  Address Translation Examples
   K9.1  AArch64 Address translation examples .............................. K9-11714
   K9.2  AArch32 Address translation examples .............................. K9-11726

Appendix K10  Example OS Save and Restore Sequences
   K10.1  Save Debug registers .................................................. K10-11736
   K10.2  Restore Debug registers ............................................... K10-11738

Appendix K11  Recommended Upload and Download Processes for External Debug
   K11.1  Using memory access mode in AArch64 state ...................... K11-11742

Appendix K12  Software Usage Examples
   K12.1  Use of the Advanced SIMD complex number instructions .......... K12-11746
   K12.2  Use of the Armv8.2 extensions to the Cryptographic Extension .... K12-11748

Appendix K13  Barrier Litmus Tests
   K13.1  Introduction ............................................................. K13-11756
   K13.2  Load-Acquire, Store-Release and barriers ........................ K13-11759
   K13.3  Load-Acquire Exclusive, Store-Release Exclusive and barriers .... K13-11763
   K13.4  Using a mailbox to send an interrupt ............................. K13-11768
   K13.5  Cache and TLB maintenance instructions and barriers ............ K13-11769
   K13.6  Armv7 compatible approaches for ordering, using DMB and DSB barriers K13-11781

Appendix K14  Random Number Generation
   K14.1  Properties of the generated random number ....................... K14-11796
Preface

This preface introduces the Arm Architecture Reference Manual, for A-profile architecture. It contains the following sections:

- *About this Manual* on page xviii.
- *Using this Manual* on page xx.
- *Conventions* on page xxvii.
- *Additional reading* on page xxx.
- *Feedback* on page xxxii.
About this Manual

This Manual describes the Arm® architecture v8, Armv8, and the Arm® architecture v9, Armv9. The architecture describes the operation of an Armv8-A and an Armv9-A Processing element (PE), and this Manual includes descriptions of:

- The two Execution states, AArch64 and AArch32.
- The instruction sets:
  - In AArch32 state, the A32 and T32 instruction sets, which are compatible with earlier versions of the Arm architecture.
  - In AArch64 state, the A64 instruction set.
- The states that determine how a PE operates, including the current Exception level and Security state, and in AArch32 state the PE mode.
- The Exception model.
- The interprocessing model, that supports transitioning between AArch64 state and AArch32 state.
- The memory model, that defines memory ordering and memory management. This Manual covers the Arm A architecture profile, both Armv8-A and Armv9-A, that defines a Virtual Memory System Architecture (VMSA). This Manual gives the assembler syntax for the instructions it describes, meaning that it describes instructions in textual form. However, this Manual is not a tutorial for Arm assembler language, nor does it describe Arm assembler language, except at a very basic level. To make effective use of Arm assembler language, read the documentation supplied with the assembler being used.

This Manual is organized into parts:

**Part A**
Provides an introduction to the Arm architecture, and an overview of the AArch64 and AArch32 Execution states.

**Part B**
Describes the application level view of the AArch64 Execution state, meaning the view from EL0. It describes the application level view of the programmers’ model and the memory model.

**Part C**
Describes the A64 instruction set, which is available in the AArch64 Execution state. The descriptions for each instruction also include the precise effects of each instruction when executed at EL0, described as unprivileged execution, including any restrictions on its use, and how the effects of the instruction differ at higher Exception levels. This information is of primary importance to authors and users of compilers, assemblers, and other programs that generate Arm machine code.

**Part D**
Describes the system level view of the AArch64 Execution state. It includes details of the System registers, most of which are not accessible from EL0, and the system level view of the programmers’ model and the memory model. This part includes the description of self-hosted debug.
Part E  Describes the application level view of the AArch32 Execution state, meaning the view from the EL0. It describes the application level view of the programmers’ model and the memory model.

——— Note ————
In AArch32 state, execution at EL0 is execution in User mode.

Part F  Describes the T32 and A32 instruction sets, which are available in the AArch32 Execution state. These instruction sets are backwards-compatible with earlier versions of the Arm architecture. This part describes the precise effects of each instruction when executed in User mode, described as *unprivileged* execution or execution at EL0, including any restrictions on its use, and how the effects of the instruction differ at higher Exception levels. This information is of primary importance to authors and users of compilers, assemblers, and other programs that generate Arm machine code.

——— Note ————
User mode is the only mode where software execution is unprivileged.

Part G  Describes the system level view of the AArch32 Execution state, which is generally compatible with earlier versions of the Arm architecture. This part includes details of the System registers, most of which are not accessible from EL0, and the instruction interface to those registers. It also describes the system level view of the programmers’ model and the memory model.

Part H  Describes the Debug architecture for external debug. This provides configuration, breakpoint and watchpoint support, and a *Debug Communications Channel* (DCC) to a debug host.

Part I  Describes additional features of the architecture that are not closely coupled to a *processing element* (PE), and therefore are accessed through memory-mapped interfaces. Some of these features are optional.

Part J  Provides pseudocode that describes various features of the Arm architecture.

Part K, Appendixes
Provide additional information. Some appendixes give information that is not part of the A-profile architectural requirements. The cover page of each appendix indicates its status.

Glossary  Defines terms used in this Manual that have a specialized meaning.

——— Note ————
Terms that are generally well understood in the microelectronics industry are not included in the Glossary.
Using this Manual

The information in this Manual is organized into parts, as described in this section.

Part A, Introduction and Architecture Overview

Part A gives an overview of the Armv8-A and Armv9-A architecture profiles, including its relationship to the other Arm PE architectures. It introduces the terminology used to describe the architecture, and gives an overview of the Executions states, AArch64 and AArch32. It contains the following chapters:

Chapter A1 Introduction to the Arm Architecture
Read this for an introduction to the Arm architecture.

Chapter A2 Armv8-A Architecture Extensions
Read this for an introduction to the Armv8 architecture extensions.

Chapter A3 Armv9 Architecture Extensions
Read this for an introduction to the Armv9 architecture extensions.

Part B, The AArch64 Application Level Architecture

Part B describes the AArch64 state application level view of the architecture. It contains the following chapters:

Chapter B1 The AArch64 Application Level Programmers' Model
Read this for an application level description of the programmers' model for software executing in AArch64 state. It describes execution at EL0 when EL0 is using AArch64 state.

Chapter B2 The AArch64 Application Level Memory Model
Read this for an application level description of the memory model for software executing in AArch64 state. It describes the memory model for execution in EL0 when EL0 is using AArch64 state. It includes information about Arm memory types, attributes, and memory access controls.

Part C, The A64 Instruction Set

Part C describes the A64 instruction set, which is used in AArch64 state. It contains the following chapters:

Chapter C1 The A64 Instruction Set
Read this for a description of the A64 instruction set and common instruction operation details.

Chapter C2 About the A64 Instruction Descriptions
Read this to understand the format of the A64 instruction descriptions.

Chapter C3 A64 Instruction Set Overview
Read this for an overview of the A64 instructions.

Chapter C4 A64 Instruction Set Encoding
Read this for a description of the A64 instruction set encoding.

Chapter C5 The A64 System Instruction Class
Read this for a description of the AArch64 System instructions and register descriptions, and the System instruction class encoding space.

Chapter C6 A64 Base Instruction Descriptions
Read this for information on key aspects of the A64 base instructions and for descriptions of the individual instructions, which are listed in alphabetical order.
Chapter C7 A64 Advanced SIMD and Floating-point Instruction Descriptions
Read this for information on key aspects of the A64 Advanced SIMD and floating-point instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

Chapter C8 SVE Instruction Descriptions
Read this for information on key aspects of the SVE instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

Chapter C9 SME Instruction Descriptions
Read this for information on key aspects of the SME instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

Part D, The AArch64 System Level Architecture
Part D describes the AArch64 state system level view of the architecture. It contains the following chapters:

Chapter D1 The AArch64 System Level Programmers’ Model
Read this for a description of the AArch64 state system level view of the programmers’ model.

Chapter D2 AArch64 Self-hosted Debug
Read this for an introduction to, and a description of, self-hosted debug in AArch64 state.

Chapter D3 AArch64 Self-hosted Trace
Read this for an introduction to, and a description of, self-hosted trace in AArch64 state.

Chapter D4 The Embedded Trace Extension
Read this for a description of the Embedded Trace Extension.

Chapter D5 ETE Protocol Descriptions
Read this for a description of the ETE packets.

Chapter D6 The Trace Buffer Extension
Read this for a description of the Trace Buffer Extension.

Chapter D7 The AArch64 System Level Memory Model
Read this for a description of the AArch64 state system level view of the general features of the memory system.

Chapter D8 The AArch64 Virtual Memory System Architecture
Read this for a system level view of the AArch64 Virtual Memory System Architecture (VMSA), the memory system architecture of an Armv8 or Armv9 implementation executing in AArch64 state.

Chapter D9 The Memory Tagging Extension
Read this for a description of the Memory Tagging Extension.

Chapter D10 The Generic Timer in AArch64 state
Read this for a description of the AArch64 view of the Arm Generic Timer.

Chapter D11 The Performance Monitors Extension
Read this for a description of the Arm Performance Monitors, an optional non-invasive debug component.

Chapter D12 The Activity Monitors Extension
Read this for a description of the Arm Activity Monitors.
Chapter D13 The Statistical Profiling Extension
Read this for a description of the Statistical Profiling Extension, an optional AArch64 state
non-invasive debug component.

Chapter D14 Statistical Profiling Extension Sample Record Specification
Read this for a description of the sample records generated by the Statistical Profiling Extension.

Chapter D15 The Branch Record Buffer Extension
Read this for a description of the Branch Record Buffer Extension.

Chapter D16 AArch64 System Register Encoding
Read this for a description of the encoding of the AArch64 System registers, and the other uses of
the AArch64 System registers encoding space.

Chapter D17 AArch64 System Register Descriptions
Read this for an introduction to, and description of, each of the AArch64 System registers.

Part E, The AArch32 Application Level Architecture
Part E describes the AArch32 state application level view of the architecture. It contains the following chapters:

Chapter E1 The AArch32 Application Level Programmers’ Model
Read this for an application level description of the programmers’ model for software executing in
AArch32 state. It describes execution at EL0 when EL0 is using AArch32 state.

Chapter E2 The AArch32 Application Level Memory Model
Read this for an application level description of the memory model for software executing in
AArch32 state. It describes the memory model for execution in EL0 when EL0 is using AArch32
state. It includes information about Arm memory types, attributes, and memory access controls.

Part F, The AArch32 Instruction Sets
Part F describes the T32 and A32 instruction sets, which are used in AArch32 state. It contains the following
chapters:

Chapter F1 About the T32 and A32 Instruction Descriptions
Read this to understand the format of the T32 and A32 instruction descriptions.

Chapter F2 The AArch32 Instruction Sets Overview
Read this for an overview of the T32 and A32 instruction sets.

Chapter F3 T32 Instruction Set Encoding
Read this for a description of the T32 instruction set encoding. This includes the T32 encoding of
the Advanced SIMD and floating-point instructions.

Chapter F4 A32 Instruction Set Encoding
Read this for a description of the A32 instruction set encoding. This includes the A32 encoding of
the Advanced SIMD and floating-point instructions.

Chapter F5 T32 and A32 Base Instruction Set Instruction Descriptions
Read this for a description of each of the T32 and A32 base instructions.

Chapter F6 T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions
Read this for a description of each of the T32 and A32 Advanced SIMD and floating-point
instructions.
Part G, The AArch32 System Level Architecture

Part G describes the AArch32 state system level view of the architecture. It contains the following chapters:

Chapter G1 The AArch32 System Level Programmers’ Model
Read this for a description of the AArch32 state system level view of the programmers’ model for execution in an Exception level that is using AArch32.

Chapter G2 AArch32 Self-hosted Debug
Read this for an introduction to, and a description of, self-hosted debug in AArch32 state.

Chapter G3 AArch32 Self-hosted Trace
Read this for an introduction to, and a description of, self-hosted trace in AArch32 state.

Chapter G4 The AArch32 System Level Memory Model
Read this for a system level view of the general features of the memory system.

Chapter G5 The AArch32 Virtual Memory System Architecture
Read this for a description of the AArch32 Virtual Memory System Architecture (VMSA).

Chapter G6 The Generic Timer in AArch32 state
Read this for a description of the AArch32 view of an implementation of the Arm Generic Timer.

Chapter G7 AArch32 System Register Encoding
Read this for a description of the encoding of the AArch32 System registers, including the System instructions that are part of the AArch32 System registers encoding space.

Chapter G8 AArch32 System Register Descriptions
Read this for a description of each of the AArch32 System registers.

Part H, External Debug

Part H describes the architecture for external debug. It contains the following chapters:

Chapter H1 About External Debug
Read this for an introduction to external debug, and a definition of the scope of this part of the Manual.

Chapter H2 Debug State
Read this for a description of Debug state, which the PE might enter as the result of a Halting debug event.

Chapter H3 Halting Debug Events
Read this for a description of the external debug events referred to as Halting debug events.

Chapter H4 The Debug Communication Channel and Instruction Transfer Register
Read this for a description of the communication between a debugger and the PE debug logic using the Debug Communications Channel and the Instruction Transfer register.

Chapter H5 The Embedded Cross-Trigger Interface
Read this for a description of the embedded cross-trigger interface.

Chapter H6 Debug Reset and Powerdown Support
Read this for a description of reset and powerdown support in the Debug architecture.

Chapter H7 The PC Sample-based Profiling Extension
Read this for a description of the PC Sample-based Profiling Extension that is an OPTIONAL extension to an Armv8 or Armv9 implementation.
Part I, Memory-mapped Components of the Arm architecture

Part I describes the memory-mapped components in the architecture. It contains the following chapters:

Chapter 11 Requirements for Memory-mapped Components
Read this for descriptions of some general requirements for memory-mapped components within a system that complies with the Arm architecture.

Chapter 12 System Level Implementation of the Generic Timer
Read this for a definition of a system level implementation of the Generic Timer.

Chapter 13 Recommended External Interface to the Performance Monitors
Read this for a description of the recommended memory-mapped and external debug interfaces to the Performance Monitors.

Chapter 14 Recommended External Interface to the Activity Monitors
Read this for a description of the recommended memory-mapped interface to the Activity Monitors.

Chapter 15 External System Control Register Descriptions
Read this for a description of each memory-mapped system control register.

Part J, Architectural Pseudocode

Part J contains pseudocode that describes various features of the Arm architecture. It contains the following chapter:

Chapter J1 Armv8 Pseudocode
Read this for the pseudocode definitions that describe various features of the Arm architecture, for operation in AArch64 state and in AArch32 state.

Part K, Appendixes

Note
Some of the descriptions in the following appendixes are not part of the Arm architecture specification. They are included here as supplementary information, for the convenience of developers and users who might require this information.

This Manual contains the following appendixes:

Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors
Read this for a description of the architecturally-required constraints on UNPREDICTABLE behaviors in the Arm architecture, including AArch32 behaviors that were UNPREDICTABLE in previous versions of the architecture.

Appendix K2 Recommendations for Reporting Memory Attributes on an Interconnect
Read this for the Arm recommendations about how the architectural memory attributes are reported on an interconnect.

Appendix K3 ETE Recommended Configurations
Read this for a description of the ETE recommended configurations.
Appendix K4 ETE and TRBE Software Usage Examples
Read this for software examples that help understanding of ETE and TRBE.

Appendix K5 Stages of execution
Read this for a description of the stages of execution.

Appendix K6 Recommended External Debug Interface
Read this for a description of the recommended external debug interface.

Appendix K7 Additional Information for Implementations of the Generic Timer
Read this for additional information about implementations of the Arm Generic Timer. This information does not form part of the architectural definition of the Generic Timer.

Appendix K8 Legacy Instruction Syntax for AArch32 Instruction Sets
Read this for information about the pre-UAL syntax of the AArch32 instruction sets, which can still be valid for the A32 instruction set.

Appendix K9 Address Translation Examples
Read this for examples of translation table lookups using the translation regimes described in Chapter D8 The AArch64 Virtual Memory System Architecture and Chapter G5 The AArch32 Virtual Memory System Architecture.

Appendix K10 Example OS Save and Restore Sequences
Read this for software examples that perform the OS Save and Restore sequences for an Armv8 or Armv9 debug implementation.

——— Note ————
Chapter H6 Debug Reset and Powerdown Support describes the OS Save and Restore mechanism.

Appendix K11 Recommended Upload and Download Processes for External Debug
Read this for information about implementing and using the Arm architecture.

Appendix K12 Software Usage Examples
Read this for software examples that help understanding of some aspects of the Arm architecture.

Appendix K13 Barrier Litmus Tests
Read this for examples of the use of barrier instructions provided by the Arm architecture.

Appendix K14 Random Number Generation
Read this for information on the generation of random numbers using FEAT_RNG.

Appendix K15 Legacy Feature Naming Convention
Read this for an understanding of how the current feature names map to the legacy naming convention.

Appendix K16 Arm Pseudocode Definition
Read this for definitions of the Arm pseudocode.

Appendix K17 Registers Index
Read this for an alphabetic and functional index of AArch32 and AArch64 registers, and memory-mapped registers.
Glossary

Defines terms used in this Manual that have a specialized meaning.

——— Note ————

Terms that are generally well understood in the microelectronics industry are not included in the Glossary.
Conventions

The following sections describe conventions that this book can use:

- **Typographic conventions.**
- **Rules-based writing on page xxviii.**
- **Signals on page xxix.**
- **Numbers on page xxix.**
- **Pseudocode descriptions on page xxix.**
- **Assembler syntax descriptions on page xxix.**

**Typographic conventions**

The typographical conventions are:

**italic** Introduces special terminology, and denotes citations.

**bold** Denotes signal names, and is used for terms in descriptive lists, where appropriate.

**monospace** Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

**SMALL CAPITALS**

Used in body text for a few terms that have specific technical meanings, and are defined in the *Glossary*.

**Colored text** Indicates a link. This can be:

- A URL, for example https://developer.arm.com.
- A cross-reference that includes the page number of the referenced information if it is not on the current page, for example, Assembler syntax descriptions on page xxix.
- A link, to a chapter or appendix, or to a glossary entry, or to the section of the Manual that defines the colored term, for example Simple sequential execution or SCTLR.

**{ and }** Braces, { and }, have two distinct uses:

**Optional items**

In syntax descriptions braces enclose optional items. In the following example they indicate that the <shift> parameter is optional:

```
ADD <Wd>|<Wsp>, <Wn>|<Wsp>, #<imm>{, <shift>}
```

Similarly they can be used in generalized field descriptions, for example TCR_ELx.{I}PS refers to a field in the TCR_ELx registers that is called either IPS or PS.

**Sets of items**

Braces can be used to enclose sets. For example, HCR_EL2.{E2H, TGE} refers to a set of two register fields, HCR_EL2.E2H and HCR_EL2.TGE.

**Notes**

Notes are formatted as:

```markdown
--- Note ---

This is a Note.
```

In this Manual, Notes are used only to provide additional information, usually to help understanding of the text. While a Note may repeat architectural information given elsewhere in the Manual, a Note never provides any part of the definition of the architecture.
Rules-based writing

Some sections of this Manual use rules-based writing. Rules-based writing consists of a set of individual content items. A content item is classified as one of the following:

- **Rule.**
- **Information.**

Rules are normative statements. An implementation that is compliant with this specification must conform to all Rules in this Manual that apply to that implementation.

Rules must not be read in isolation. Where a particular feature is specified by multiple Rules, these are generally grouped into sections and subsections that provide context. Where appropriate, these sections begin with a short introduction.

Arm strongly recommends that implementers read all chapters and sections of this Manual to ensure that an implementation is compliant.

Content items other than Rules are informative statements. These are provided as an aid to understanding this Manual.

Content item identifiers

A content item may have an associated identifier which is unique among content items in this Manual. After content reaches beta status, a given content item has the same identifier across subsequent versions of this Manual.

Content item rendering

In this Manual, a content item is rendered with a token of the following format in the left margin: \texttt{Liiii}. \texttt{L} is a label that indicates the content class of the content item.

\texttt{iii} is the identifier of the content item.

Content item classes

Each of the content item classes has a different function in this Manual.

**Rule**

A Rule is a statement that describes the behavior of a compliant implementation.

A Rule explains what happens in a particular situation.

A Rule does not define concepts or terminology.

A Rule is rendered with the label \texttt{R}.

**Information**

An Information statement provides information and guidance as an aid to understanding the Manual.

An Information statement is rendered with the label \texttt{I}.

**Software usage**

A Software usage statement provides guidance on how software can make use of the features defined by the specification.

A Software usage statement is rendered with the label \texttt{S}. 
Signals

In general this specification does not define hardware signals, but it does include some signal examples and recommendations. The signal conventions are:

**Signal level**

The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- **HIGH** for active-HIGH signals.
- **LOW** for active-LOW signals.

**Lowercase n**

At the start or end of a signal name denotes an active-LOW signal.

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a monospace font, for example `0xFFFF0000`. To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

Pseudocode descriptions

This Manual uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in monospace font, and is described in Appendix K16 Arm Pseudocode Definition.

Assembler syntax descriptions

This Manual contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a monospace font, and use the conventions described in Structure of the A64 assembler language on page C1-228, and Appendix K16 Arm Pseudocode Definition.
Additional reading

This section lists relevant publications from Arm and third parties.

See Arm Developer, https://developer.arm.com, for access to Arm documentation.

Arm publications

- Arm® Reliability, Availability, and Serviceability (RAS) Specification, for A-profile architecture (ARM DDI 0587).
- Arm® Architecture Reference Manual Supplement, Armv8, for R-profile AArch64 architecture (ARM DDI 600).
- ARM® Debug Interface Architecture Specification, ADIv6.0 (ARM IHI 0074).
- ARM® Debug Interface Architecture Specification, ADIv5.0 to ADIv5.2 (ARM IHI 0031).
- ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0 (ARM IHI 0069).
- ARM® CoreSight™ Architecture Specification (ARM IHI 0029).
- ARM® Procedure Call Standard for the ARM 64-bit Architecture (ARM IHI 0055).

Other publications

The following publications are referred to in this Manual, or provide more information:


• *SM3 Cryptographic Hash Algorithm*, China Internet Network Information Center (CNNIC).

• *SM4 Block Cipher Algorithm*, China Internet Network Information Center (CNNIC).

Feedback

Arm welcomes feedback on its documentation.

Feedback on this Manual

If you have comments on the content of this Manual, send email to errata@arm.com. Give:

• The number, ARM DDI 0487I.a.
• The section name to which your comments refer.
• The page numbers to which your comments refer.
• A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader.

Progressive Terminology Commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

Previous issues of this document included terms that can be offensive. We have replaced these terms. If you find offensive terms in this document, please contact terms@arm.com.
Part A

Arm Architecture Introduction and Overview
Chapter A1
Introduction to the Arm Architecture

This chapter introduces the Arm architecture. It contains the following sections:

• About the Arm architecture on page A1-36.
• Supported data types on page A1-42.
• Floating-point support on page A1-55.
• The Arm memory model on page A1-69.
A1.1 About the Arm architecture

The Arm architecture described in this Architecture Reference Manual defines the behavior of an abstract machine, referred to as a processing element, often abbreviated to PE. Implementations compliant with the Arm architecture must conform to the described behavior of the processing element. It is not intended to describe how to build an implementation of the PE, nor to limit the scope of such implementations beyond the defined behaviors.

Except where the architecture specifies differently, the programmer-visible behavior of an implementation that is compliant with the Arm architecture must be the same as a simple sequential execution of the program on the processing element. This programmer-visible behavior does not include the execution time of the program.

The Arm Architecture Reference Manual also describes rules for software to use the processing element.

The Arm architecture includes definitions of:

- An associated debug architecture, see:
  - Chapter D2 AArch64 Self-hosted Debug.
  - Chapter G2 AArch32 Self-hosted Debug.

- Associated trace architectures that define trace units that implementers can implement with the associated processor hardware. For more information, see:
  - The Embedded Trace Macrocell Architecture Specification.
  - Chapter D3 AArch64 Self-hosted Trace.
  - Chapter D4 The Embedded Trace Extension.
  - Chapter D6 The Trace Buffer Extension.
  - Chapter G3 AArch32 Self-hosted Trace.

The Arm architecture is a Reduced Instruction Set Computer (RISC) architecture with the following RISC architecture features:

- A large uniform register file.
- A load/store architecture, where data-processing operations only operate on register contents, not directly on memory contents.
- Simple addressing modes, with all load/store addresses determined from register contents and instruction fields only.

The architecture defines the interaction of the PE with memory, including caches, and includes a memory translation system. It also describes how multiple PEs interact with each other and with other observers in a system.


The Arm architecture supports implementations across a wide range of performance points. Implementation size, performance, and very low power consumption are key attributes of the Arm architecture.

An important feature of the Arm architecture is backwards compatibility, combined with the freedom for optimal implementation in a wide range of standard and more specialized use cases. The Arm architecture supports:

- A 64-bit Execution state, AArch64.
- A 32-bit Execution state, AArch32, that is compatible with previous versions of the Arm architecture.

Features that are optional are explicitly defined as such in this Manual.

**Note**

The presence of an ID register field for a feature does not imply that the feature is optional.

Both Execution states support floating-point instructions:

- AArch32 state provides:
  - SIMD instructions in the base instruction sets that operate on the 32-bit general-purpose registers.
— Advanced SIMD instructions that operate on registers in the SIMD and floating-point register (SIMD&FP register) file.
— Scalar floating-point instructions that operate on registers in the SIMD&FP register file.

• AArch64 state provides:
  — Advanced SIMD instructions that operate on registers in the SIMD&FP register file.
  — Scalar floating-point instructions that operate on registers in the SIMD&FP register file.

  **Note**
  The A64 instruction set does not include SIMD instructions that operate on the general-purpose registers, therefore, some AArch64 instructions descriptions use SIMD as a synonym for Advanced SIMD.

  **Note**
  SVE instructions that operate on registers in the SVE register file.

See *Conventions on page xxvii* for information about conventions used in this Manual, including the use of small capitals for particular terms that have Arm-specific meanings that are defined in the *Glossary*.
A1.2 Architecture profiles

The Arm architecture has evolved significantly since its introduction, and Arm continues to develop it. Nine major versions of the architecture have been defined to date, denoted by the version numbers 1 to 9. Of these, the first three versions are now obsolete.

The generic names AArch64 and AArch32 describe the 64-bit and 32-bit Execution states:

**AArch64** Is the 64-bit Execution state, meaning addresses are held in 64-bit registers, and instructions in the base instruction set can use 64-bit registers for their processing. AArch64 state supports the A64 instruction set.

**AArch32** Is the 32-bit Execution state, meaning addresses are held in 32-bit registers, and instructions in the base instruction sets use 32-bit registers for their processing. AArch32 state supports the T32 and A32 instruction sets.

--- Note ---
The Base instruction set comprises the supported instructions other than the floating-point instructions.

---


Arm defines three architecture profiles:

**A** Application profile, described in this Manual:

- Supports a Virtual Memory System Architecture (VMSA) based on a Memory Management Unit (MMU).
  
  --- Note ---
  An Armv8-A implementation can be called an AArchv8-A implementation and an Armv9-A implementation can be called an AArchv9-A implementation.

- Supports the A64, A32, and T32 instruction sets.

**R** Real-time profile:

- Supports a Protected Memory System Architecture (PMSA) based on a Memory Protection Unit (MPU).
  - Supports an optional VMSA based on an MMU.
  - Supports the A64, A32, and T32 instruction sets.

**M** Microcontroller profile:

- Implements a programmers' model designed for low-latency interrupt processing, with hardware stacking of registers and support for writing interrupt handlers in high-level languages.
- Implements a variant of the R-profile PMSA.
- Supports a variant of the T32 instruction set.

This Manual describes only Armv8-A and Armv9-A. For information about the R and M architecture profiles, and earlier Arm architecture versions, see:

- The *Arm®v8-M Architecture Reference Manual*.
- The *Arm®v7-M Architecture Reference Manual*.
- The *Arm®v6-M Architecture Reference Manual*. 
A1.3 Arm architectural concepts

This section introduces architectural concepts and the associated terminology.

The following subsections describe key architectural concepts. Each section introduces the corresponding terms that are used to describe the architecture:
- **Execution state.**
- **The instruction sets** on page A1-40.
- **System registers** on page A1-40.
- **Arm Debug** on page A1-41.

### A1.3.1 Execution state

The Execution state defines the PE execution environment, including:
- The supported register widths.
- The supported instruction sets.
- Significant aspects of:
  - The Exception model.
  - The **Virtual Memory System Architecture (VMSA)**.
  - The programmers' model.

The Execution states are:

**AArch64**

The 64-bit Execution state. This Execution state:
- Provides 31 64-bit general-purpose registers, of which X30 is used as the procedure link register.
- Provides a 64-bit Program Counter (PC), stack pointers (SPs), and Exception Link Registers (ELRs).
- Provides 32 128-bit registers for Advanced SIMD vector and scalar floating-point support.
- Provides a single instruction set, A64. For more information, see **The instruction sets** on page A1-40.
- Defines the Armv8 Exception model, with up to four Exception levels, EL0 - EL3, that provide an execution privilege hierarchy, see **Exception levels** on page D1-4632.
- Provides support for 64-bit virtual addressing. For more information, including the limits on address ranges, see Chapter D8 The **AArch64 Virtual Memory System Architecture**.
- Defines a number of Process state (PSTATE) elements that hold PE state. The A64 instruction set includes instructions that operate directly on various PSTATE elements.
- Names each System register using a suffix that indicates the lowest Exception level at which the register can be accessed.

**AArch32**

The 32-bit Execution state. This Execution state:
- Provides 13 32-bit general-purpose registers, and a 32-bit PC, SP, and Link Register (LR). The LR is used as both an ELR and a procedure link register. Some of these registers have multiple banked instances for use in different PE modes.
- Provides a single ELR, for exception returns from Hyp mode.
- Provides 32 64-bit registers for Advanced SIMD vector and scalar floating-point support.
- Provides two instruction sets, A32 and T32. For more information, see **The instruction sets** on page A1-40.
- Supports the Armv7-A Exception model, based on PE modes, and maps this onto the Armv8 Exception model, that is based on the Exception levels.
- Provides support for 32-bit virtual addressing.
Introduction to the Arm Architecture
A1.3 Arm architectural concepts

- Defines a number of Process state (PSTATE) elements that hold PE state. The A32 and T32 instruction sets include instructions that operate directly on various PSTATE elements, and instructions that access PSTATE by using the Application Program Status Register (APSR) or the Current Program Status Register (CPSR).

Later subsections give more information about the different properties of the Execution states.

Transferring control between the AArch64 and AArch32 Execution states is known as interprocessing. The PE can move between Execution states only on a change of Exception level, and subject to the rules given in Interprocessing on page D1-4684. This means different software layers, such as an application, an operating system kernel, and a hypervisor, executing at different Exception levels, can execute in different Execution states.

A1.3.2 The instruction sets

The possible instruction sets depend on the Execution state:

AArch64
AArch64 state supports only a single instruction set, called A64. This is a fixed-length instruction set that uses 32-bit instruction encodings.
For information on the A64 instruction set, see Chapter C3 A64 Instruction Set Overview.
If FEAT_SVE is implemented, the instruction set also supports scalable vector instructions. See About the SVE instructions on page C8-2974.

AArch32
AArch32 state supports the following instruction sets:
- A32 This is a fixed-length instruction set that uses 32-bit instruction encodings.
- T32 This is a variable-length instruction set that uses both 16-bit and 32-bit instruction encodings.

In previous documentation, these instruction sets were called the ARM and Thumb instruction sets. Armv8 and Armv9 extend each of these instruction sets. In AArch32 state, the Instruction set state determines the instruction set that the PE executes.
For information on the A32 and T32 instruction sets, see Chapter F2 The AArch32 Instruction Sets Overview.

The instruction sets support SIMD and scalar floating-point instructions. See Floating-point support on page A1-55.

A1.3.3 System registers

System registers provide control and status information of architected features.

The System registers use a standard naming format: <register_name>.<bit_field_name> to identify specific registers as well as control and status bits within a register.

Bits can also be described by their numerical position in the form <register_name>[x:y] or the generic form bits[x:y].

In addition, in AArch64 state, most register names include the lowest Exception level that can access the register as a suffix to the register name:
- <register_name>_ELx, where x is 0, 1, 2, or 3.

For information about Exception levels, see Exception levels on page D1-4632.

The System registers comprise:
- The following registers that are described in this Manual:
  - General system control registers.
  - Debug registers.
  - Generic Timer registers.
  - Optionally, Performance Monitor registers.
  - Optionally, the Activity Monitors registers.
— Optionally, the Scalable Vector Extension registers.
— Optionally, in Armv9, Trace System registers.

- Optionally, one or more of the following groups of registers that are defined in other Arm architecture specifications:
  - Trace System registers, as defined in the Embedded Trace Macrocell Architecture Specification, ETMv4.
  - Generic Interrupt Controller (GIC) System registers, see The Arm Generic Interrupt Controller System registers.

- RAS Extension System registers, as defined in the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

For information about the AArch64 System registers, see Chapter D17 AArch64 System Register Descriptions.
For information about the AArch32 System registers, see Chapter G8 AArch32 System Register Descriptions.

### The Arm Generic Interrupt Controller System registers

From version 3 of the Arm Generic Interrupt Controller architecture, GICv3, the GIC architecture specification defines a System register interface to some of its functionality. The System register summaries in this Manual include these registers, see:

- About the GIC System registers on page D16-5542, for more information about the AArch64 GIC System registers.
- About the GIC System registers on page G7-9319, for more information about the AArch32 GIC System registers.

These sections give only short overviews of the GIC System registers. For more information, including descriptions of the registers, see the ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0 (ARM IHI 0069).

--- Note

The programmers’ model for earlier versions of the GIC architecture is wholly memory-mapped.

### A1.3.4 Arm Debug

Armv8 and later architectures support the following:

#### Self-hosted debug

In this model, the PE generates debug exceptions. Debug exceptions are part of the Armv8 Exception model.

#### External debug

In this model, debug events cause the PE to enter Debug state. In Debug state, the PE is controlled by an external debugger.

All Armv8 and later implementations support both models. The model chosen by a particular user depends on the debug requirements during different stages of the design and development life cycle of the product. For example, external debug might be used during debugging of the hardware implementation and OS bring-up, and self-hosted debug might be used during application development.

For more information about self-hosted debug:
- In AArch64 state, see Chapter D2 AArch64 Self-hosted Debug.
- In AArch32 state, see Chapter G2 AArch32 Self-hosted Debug.

For more information about external debug, see Part H External Debug on page Part H-10225.
A1.4 **Supported data types**

The Arm architecture supports the following integer data types:

- **Byte**
  - 8 bits.
- **Halfword**
  - 16 bits.
- **Word**
  - 32 bits.
- **Doubleword**
  - 64 bits.
- **Quadword**
  - 128 bits.

The architecture also supports the following floating-point data types:

- Double-precision, see *Double-precision floating-point format* on page A1-51 for details.
- BFloat16, see *BFloat16 floating-point format* on page A1-52 for details.

It also supports:

- Vectors, where a register holds multiple elements, each of the same data type. See *Advanced SIMD vector formats* on page A1-43 for details.

The architecture provides the following register files:

- A general-purpose register file.
- A SIMD&FP register file.
- If FEAT_SVE is implemented, an SVE scalable vector register file.

In each of these, the possible register widths depend on the Execution state.

In AArch64 state:

- A general-purpose register file contains 64-bit registers:
  - Many instructions can access these registers as 64-bit registers or as 32-bit registers, using only the bottom 32 bits.
- A SIMD&FP register file contains 128-bit registers:
  - While the AArch64 vector registers support 128-bit vectors, the effective vector length can be 64-bits or 128-bits depending on the A64 instruction encoding used, see *Instruction Mnemonics* on page C1-229.
- An SVE scalable vector register file contains registers of an *IMPLEMENTATION DEFINED* width:
  - An SVE scalable vector register has an *IMPLEMENTATION DEFINED* width that is a multiple of 128 bits, up to a maximum of 2048 bits.
  - All SVE scalable vector registers in an implementation are the same width.
- An SVE predicate register file contains registers of an *IMPLEMENTATION DEFINED* width:
  - An SVE predicate register has an *IMPLEMENTATION DEFINED* width that is a multiple of 16 bits, up to a maximum of 256 bits.

For more information on the register files in AArch64 state, see *Registers in AArch64 Execution state* on page B1-137.

In AArch32 state:

- A general-purpose register file contains 32-bit registers:
  - Two 32-bit registers can support a doubleword.
  - Vector formatting is supported, see *Figure A1-4* on page A1-46.
- A SIMD&FP register file contains 64-bit registers:
  - AArch32 state does not support quadword integer or floating-point data types.
A1.4 Supported data types

--- Note ---
Two consecutive 64-bit registers can be used as a 128-bit register.

For more information on the register files in AArch32 state, see The general-purpose registers, and the PC, in AArch32 state on page E1-7131.

A1.4.1 Advanced SIMD vector formats

In an implementation that includes the Advanced SIMD instructions that operate on the SIMD&FP register file, a register can hold one or more packed elements, all of the same size and type. In AArch32 state, the combination of a register and a data type describes a vector of elements, where the number of elements in the vector is implied by the size of the data type and the size of the register. In AArch64 state, the explicit combination of a register, number of elements, and element size describes a vector of elements. The vector is considered to be a one-dimensional array of elements of the data type specified in the instruction.

Vector indices are in the range 0 to (number of elements – 1). An index of 0 refers to the least significant bits of the vector.

For more information on the Advanced SIMD and floating-point registers in AArch32 state, see The SIMD and floating-point register file on page E1-7140.

Advanced SIMD vector formats in AArch64 state

In AArch64 state, the SIMD&FP registers can be referred to as Vn, where n is a value from 0 to 31.

The SIMD&FP registers support three data formats for loads, stores, and data-processing operations:

- A single, scalar, element in the least significant bits of the register.
- A 64-bit vector of byte, halfword, word, or doubleword elements.
- A 128-bit vector of byte, halfword, word, or doubleword elements.

The element sizes are defined in Table A1-1 on page A1-43 with the vector format described as:

- For a 128-bit vector: Vn{.2D, .4S, .8H, .16B}.
- For a 64-bit vector: Vn{.1D, .2S, .4H, .8B}.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>8 bits</td>
</tr>
<tr>
<td>H</td>
<td>16 bits</td>
</tr>
<tr>
<td>S</td>
<td>32 bits</td>
</tr>
<tr>
<td>D</td>
<td>64 bits</td>
</tr>
</tbody>
</table>

Table A1-1: SIMD elements in AArch64 state

Figure A1-1 on page A1-44 shows the SIMD vectors in AArch64 state.
A1.4 Supported data types

Table A1-2 shows the available formats. Each instruction description specifies the data types that the instruction supports.

<table>
<thead>
<tr>
<th>Data type specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.&lt;size&gt;</td>
<td>Any element of &lt;size&gt; bits</td>
</tr>
<tr>
<td>.F&lt;size&gt;</td>
<td>Floating-point number of &lt;size&gt; bits</td>
</tr>
<tr>
<td>.I&lt;size&gt;</td>
<td>Signed or unsigned integer of &lt;size&gt; bits</td>
</tr>
<tr>
<td>.P&lt;size&gt;</td>
<td>Polynomial over {0, 1} of degree less than &lt;size&gt;</td>
</tr>
<tr>
<td>.S&lt;size&gt;</td>
<td>Signed integer of &lt;size&gt; bits</td>
</tr>
<tr>
<td>.U&lt;size&gt;</td>
<td>Unsigned integer of &lt;size&gt; bits</td>
</tr>
</tbody>
</table>

Polynomial arithmetic over \{0, 1\} on page A1-54 describes the polynomial data type.

The .F16 data type is the half-precision data type selected by the FPSCR.AHP bit, see Half-precision floating-point formats on page A1-48.
The `.F32` data type is the Arm standard single-precision floating-point data type, see *Single-precision floating-point format* on page A1-50.

The instruction definitions use a data type specifier to define the data types appropriate to the operation. Figure A1-2 shows the hierarchy of the Advanced SIMD data types.

![Figure A1-2 Advanced SIMD data type hierarchy in AArch32 state](image)

For example, a multiply instruction must distinguish between integer and floating-point data types.

An integer multiply instruction that generates a double-width (long) result must specify the input data types as signed or unsigned. However, some integer multiply instructions use modulo arithmetic, and therefore do not have to distinguish between signed and unsigned inputs.

Figure A1-3 on page A1-46 shows the Advanced SIMD vectors in AArch32 state.

---

*Note*

In AArch32 state, a pair of even and following odd numbered doubleword registers can be concatenated and treated as a single quadword register.
Figure A1-3 Advanced SIMD vectors in AArch32 state

The AArch32 general-purpose registers support vectors formats for use by the SIMD instructions in the Base instruction set. Figure A1-4 shows these formats, that means that a general-purpose register can be treated as either 2 halfwords or 4 bytes.

Figure A1-4 Vector formatting in AArch32 state

A1.4.2 SVE vector format

In an implementation that includes the AArch64 SVE instructions, an SVE register can hold one or more packed or unpacked elements, all of the same size and type. The combination of a register and an element size describes a vector of elements. The vector is considered to be a one-dimensional array of elements of the data type specified in the instruction. The number of elements in the vector is implied by the size of the data elements and the Effective SVE vector length of the register.

Vector indexes are in the range 0 to (number of elements – 1). An index of 0 refers to the least significant bits of the vector.
Scalable vector format in AArch64 state

In AArch64 state, the SVE registers can be referred to as $Z_n$, where $n$ is a value from 0 to 31. For a full description of the SVE scalable vector registers, see *SVE vector registers on page B1-140*.

The element sizes are defined in Table A1-3 with the vector format described as:

### Table A1-3 SVE elements in AArch64 state

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Element size</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>8 bits</td>
</tr>
<tr>
<td>H</td>
<td>16 bits</td>
</tr>
<tr>
<td>S</td>
<td>32 bits</td>
</tr>
<tr>
<td>D</td>
<td>64 bits</td>
</tr>
<tr>
<td>Q</td>
<td>128 bits</td>
</tr>
</tbody>
</table>

SVE configurable vector length

Privileged Exception levels can use the ZCR_ELx.LEN System register fields to constrain the vector length at that Exception level and at less privileged Exception levels.

An implementation allows the vector length to be constrained to any power of two that is less than the maximum implemented vector length.

An implementation is permitted to allow the vector length to be constrained to multiples of 128 that are not a power of two. It is IMPLEMENTATION DEFINED which of the permitted multiples of 128 are supported.

The following table shows the SVE configurable vector lengths:

<table>
<thead>
<tr>
<th>Maximum</th>
<th>Required</th>
<th>Permitted</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>128</td>
<td>-</td>
</tr>
<tr>
<td>256</td>
<td>128, 256</td>
<td>-</td>
</tr>
<tr>
<td>384</td>
<td>128, 256</td>
<td>384</td>
</tr>
<tr>
<td>512</td>
<td>128, 256</td>
<td>384</td>
</tr>
<tr>
<td>640</td>
<td>128, 256, 512</td>
<td>384</td>
</tr>
<tr>
<td>768</td>
<td>128, 256, 512</td>
<td>384, 640</td>
</tr>
<tr>
<td>896</td>
<td>128, 256, 512</td>
<td>384, 640, 768</td>
</tr>
<tr>
<td>1024</td>
<td>128, 256, 512</td>
<td>384, 640, 768, 896</td>
</tr>
<tr>
<td>1152</td>
<td>128, 256, 512, 1024</td>
<td>384, 640, 768, 896</td>
</tr>
<tr>
<td>1280</td>
<td>128, 256, 512, 1024</td>
<td>384, 640, 768, 896, 1152</td>
</tr>
<tr>
<td>1408</td>
<td>128, 256, 512, 1024</td>
<td>384, 640, 768, 896, 1152, 1280</td>
</tr>
<tr>
<td>1536</td>
<td>128, 256, 512, 1024</td>
<td>384, 640, 768, 896, 1152, 1280, 1408</td>
</tr>
<tr>
<td>1664</td>
<td>128, 256, 512, 1024</td>
<td>384, 640, 768, 896, 1152, 1280, 1408, 1536</td>
</tr>
</tbody>
</table>
A1.4 Supported data types

### A1.4.3 Half-precision floating-point formats

The Arm architecture supports two half-precision floating-point formats:

- IEEE half-precision, as described in the IEEE 754-2008 standard.
- Arm alternative half-precision format.

--- **Note** ---

BFloat16 is not a half-precision floating-point format, see *BFloat16 floating-point format on page A1-52.*

Both formats can be used for conversions to and from other floating-point formats. FPCR.AHP controls the format in AArch64 state and FPSCR.AHP controls the format in AArch32 state. FEAT_FP16 adds half-precision data-processing instructions, which always use the IEEE format. These instructions ignore the value of the relevant AHP field, and behave as if it has an *Effective value* of 0. The FEAT_SVE half-precision data-processing instructions ignore the value of FPCR.AHP, and behave as if it has an *Effective value* of 0.

The description of IEEE half-precision includes Arm-specific details that are left open by the standard, and is only an introduction to the formats and to the values they can contain. For more information, especially on the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.
For both half-precision floating-point formats, the layout of the 16-bit format is the same. The format is:

```
  15 14  10  9           0
 S | exponent | fraction
```

The interpretation of the format depends on the value of the exponent field, bits[14:10] and on which half-precision format is being used.

**0 < exponent < 0x1F**

The value is a normalized number and is equal to:

\[ (-1)^S \times 2^{(\text{exponent}-15)} \times (1.fraction) \]

The minimum positive normalized number is \(2^{-14}\), or approximately \(6.104 \times 10^{-5}\).

The maximum positive normalized number is \((2 - 2^{-10}) \times 2^{15}\), or 65504.

Larger normalized numbers can be expressed using the alternative format when the exponent == 0x1F.

**exponent == 0**

The value is either a zero or a denormalized number, depending on the fraction bits:

- **fraction == 0**
  
  The value is a zero. There are two distinct zeros:
  
  +0 when S==0.
  
  -0 when S==1.

- **fraction != 0**
  
  The value is a denormalized number and is equal to:
  
  \[ (-1)^S \times 2^{-14} \times (0.fraction) \]
  
  The minimum positive denormalized number is \(2^{-24}\), or approximately \(5.960 \times 10^{-8}\).

Half-precision denormalized numbers are not flushed to zero by default. When FEAT_FP16 is implemented, the FPCR.FZ16 bit controls whether flushing denormalized numbers to zero is enabled for half-precision data-processing instructions. For details, see [Flushing denormalized numbers to zero](#) on page A1-58.

**exponent == 0x1F**

The value depends on which half-precision format is being used:

- **IEEE half-precision**
  
  The value is either an infinity or a Not a Number (NaN), depending on the fraction bits:
  
  - **fraction == 0**
    
    The value is an infinity. There are two distinct infinities:
    
    +infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.
    
    -infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.
  
  - **fraction != 0**
    
    The value is a NaN, and is either a quiet NaN or a signaling NaN.
    
    The two types of NaN are distinguished by their most significant fraction bit, bit[9]:
    
    bit[9] == 0 The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.
    
    bit[9] == 1 The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.
Alternative half-precision
The value is a normalized number and is equal to:
\(-1^S \times 2^{16} \times (1.\text{fraction})\)
The maximum positive normalized number is \((2-2^{-10}) \times 2^{16}\) or 131008.

A1.4.4 Single-precision floating-point format
The single-precision floating-point format is as defined by the IEEE 754 standard.
This description includes Arm-specific details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.

A single-precision value is a 32-bit word with the format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>23</th>
<th>22</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>exponent</td>
<td></td>
<td></td>
<td>fraction</td>
</tr>
</tbody>
</table>

The interpretation of the format depends on the value of the exponent field, bits[30:23]:

0 < exponent < 0xFF

The value is a normalized number and is equal to:
\((-1)^S \times 2^{(\text{exponent} - 127)} \times (1.\text{fraction})\)
The minimum positive normalized number is \(2^{-126}\), or approximately \(1.175 \times 10^{-38}\).
The maximum positive normalized number is \((2 - 2^{-23}) \times 2^{127}\), or approximately \(3.403 \times 10^{38}\).

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:
+0 When S==0.
–0 When S==1.
These usually behave identically. In particular, the result is equal if +0 and –0 are compared as floating-point numbers. However, they yield different results in some circumstances. For example, the sign of the infinity produced as the result of dividing by zero depends on the sign of the zero. The two zeros can be distinguished from each other by performing an integer comparison of the two words.

fraction != 0

The value is a denormalized number and is equal to:
\((-1)^S \times 2^{-126} \times (0.\text{fraction})\)
The minimum positive denormalized number is \(2^{-149}\), or approximately \(1.401 \times 10^{-45}\).

Denormalized numbers are always flushed to zero in Advanced SIMD processing in AArch32 state. They are optionally flushed to zero in floating-point processing and in Advanced SIMD processing in AArch64 state. For details, see Flushing denormalized numbers to zero on page A1-58.

exponent == 0xFF

The value is either an infinity or a Not a Number (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:
+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.
-infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.
fraction != 0

The value is a NaN, and is either a quiet NaN or a signaling NaN. The two types of NaN are distinguished by their most significant fraction bit, bit[22]:

bit[22] == 0

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[22] == 1

The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

For details of the default NaN, see The Default NaN on page A1-60.

--- Note ---

NaNs with different sign or fraction bits are distinct NaNs, but this does not mean software can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as unordered with everything, including itself.

A1.4.5 Double-precision floating-point format

The double-precision floating-point format is as defined by the IEEE 754 standard. Double-precision floating-point is supported by both SIMD and floating-point instructions in AArch64 state, and only by floating-point instructions in AArch32 state.

This description includes implementation-specific details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.

A double-precision value is a 64-bit doubleword, with the format:

```
S  exponent  fraction
```

Double-precision values represent numbers, infinities, and NaNs in a similar way to single-precision values, with the interpretation of the format depending on the value of the exponent:

0 < exponent < 0x7FF

The value is a normalized number and is equal to:

\((-1)^S \times 2^{(exponent-1023)} \times (1.fraction)\)

The minimum positive normalized number is \(2^{-1022}\), or approximately \(2.225 \times 10^{-308}\).

The maximum positive normalized number is \((2 - 2^{-52}) \times 2^{1023}\), or approximately \(1.798 \times 10^{308}\).

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros that behave in the same way as the two single-precision zeros:

+0 when S==0.

–0 when S==1.

fraction != 0

The value is a denormalized number and is equal to:

\((-1)^S \times 2^{-1022} \times (0.fraction)\)

The minimum positive denormalized number is \(2^{-1074}\), or approximately \(4.941 \times 10^{-324}\).
Optionally, denormalized numbers are flushed to zero in floating-point calculations. For details, see Flushing denormalized numbers to zero on page A1-58.

\[
\begin{align*}
\text{exponent} &= \text{0xFF} \\
\text{fraction} &= \text{0} \\
\text{fraction} &
\end{align*}
\]

The value is either an infinity or a NaN, depending on the fraction bits:

\[
\begin{align*}
\text{fraction} &= \text{0} \\
\text{fraction} &
\end{align*}
\]

The value is an infinity. As for single-precision, there are two infinities:

\[
\begin{align*}
+\text{infinity} &\quad \text{When S==0.} \\
-\text{infinity} &\quad \text{When S==1.}
\end{align*}
\]

The value is a NaN, and is either a quiet NaN or a signaling NaN.

The two types of NaN are distinguished by their most significant fraction bit, bit[51] of the doubleword:

\[
\begin{align*}
\text{bit}[51] &= \text{0} \\
\text{bit}[51] &
\end{align*}
\]

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

\[
\begin{align*}
\text{bit}[51] &= \text{1} \\
\text{bit}[51] &
\end{align*}
\]

The NaN is a quiet NaN. The sign bit and the remaining fraction bits can take any value.

For details of the default NaN, see The Default NaN on page A1-60.

---

**Note**

NaNs with different sign or fraction bits are distinct NaNs, but this does not mean software can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as unordered with everything, including itself.

---

### A1.4.6 BFloat16 floating-point format

BFloat16, or BF16, is a 16-bit floating-point storage format. The BF16 format inherits many of its properties and behaviors from the single-precision format defined by the IEEE 754 standard, as described in Single-precision floating-point format on page A1-50.

For the BFloat16 floating-point format, the layout is:

\[
\begin{array}{c|c|c|c}
15 & 14 & 7 & 6 & 0 \\
\hline
S & \text{exponent} & \text{fraction} &
\end{array}
\]

\[
0 < \text{exponent} < \text{0xFF}
\]

The value is a normalized number and is equal to:

\[
(-1)^S \times 2^{(\text{exponent}-127)} \times (\text{1.fraction})
\]

The minimum positive normalized number is $2^{-126}$, or approximately $1.175 \times 10^{-38}$.

The maximum positive normalized number is $(2 - 2^{-7}) \times 2^{127}$, or approximately $3.390 \times 10^{38}$.

\[
\text{exponent} == \text{0}
\]

The value is either a zero or a denormalized number, depending on the fraction bits:

\[
\begin{align*}
\text{fraction} &= \text{0} \\
\text{fraction} &
\end{align*}
\]

The value is a zero. There are two distinct zeros:

\[
\begin{align*}
+0 &\quad \text{when S==0.} \\
-0 &\quad \text{when S==1.}
\end{align*}
\]
These usually behave identically. However, they yield different results in some circumstances. For example, the sign of the result produced as the result of multiplying by zero depends on the sign of the zero. The two zeros can be distinguished from each other by performing an integer bitwise comparison of the two halfwords.

\[
\text{fraction} \neq 0
\]

The value is a denormalized number and is equal to:

\[
(-1)^S \times 2^{-126} \times (0.\text{fraction})
\]

The minimum positive denormalized number is \(2^{-133}\), or approximately \(9.184 \times 10^{-41}\).

Denormalized numbers are always flushed to zero in Advanced SIMD processing in AArch32 state. They are optionally flushed to zero in floating-point processing and in Advanced SIMD processing in AArch64 state. For details, see Flushing denormalized numbers to zero on page A1-58.

\[
exponent == 0xFF
\]

The value is either an infinity or a Not a Number (NaN), depending on the fraction bits:

\[
\text{fraction} == 0
\]

The value is an infinity. There are two distinct infinities:

**+infinity** When \(S==0\). This represents all positive numbers that are too big to be represented accurately as a normalized number.

**-infinity** When \(S==1\). This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

\[
\text{fraction} \neq 0
\]

The value is a NaN, and is either a quiet NaN or a signaling NaN.

The two types of NaN are distinguished by their most significant fraction bit, bit[6]:

\[
\text{bit}[6] == 0 \quad \text{The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.}
\]

\[
\text{bit}[6] == 1 \quad \text{The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.}
\]

In the arithmetic instructions that accept BF16 inputs, there is no distinction between quiet and signaling input NaNs, since these instructions cannot signal a floating-point exception, and any type of input NaN generates the same Default NaN result.

BF16 values are 16-bit halfwords that software can convert to single-precision format, by appending 16 zero bits, so that single-precision arithmetic instructions can be used. A single-precision value can be converted to BF16 format if required, either by:

- Truncating, by removing the least significant 16 bits.
- Using the BFloat16 conversion instructions, see Floating-point single-precision to BFloat16 conversion instruction on page C3-313.

### A1.4.7 Fixed-point format

Fixed-point formats are used only for conversions between floating-point and fixed-point values. They apply to general-purpose registers.

Fixed-point values can be signed or unsigned, and can be 16-bit or 32-bit. Conversion instructions take an argument that specifies the number of fraction bits in the fixed-point number. That is, it specifies the position of the binary point.

### A1.4.8 Conversion between floating-point and fixed-point values

The Arm architecture supports the conversion of a scalar floating-point to or from a signed or unsigned fixed-point value in a general-purpose register.

The instruction argument \#fbits indicates that the general-purpose register holds a fixed-point number with fbits bits after the binary point, where fbits is in the range 1 to 64 for a 64-bit general-purpose register, or 1 to 32 for a 32-bit general-purpose register.
More specifically:
- For a 64-bit register \( X_d \):
  - The integer part is \( X_d[63:#fbits] \).
  - The fractional part is \( X_d[(#fbits-1):0] \).
- For a 32-bit register \( W_d \) or \( R_d \):
  - The integer part is \( W_d[31:#fbits] \) or \( R_d[31:#fbits] \).
  - The fractional part is \( W_d[(#fbits-1):0] \) or \( R_d[(#fbits-1):0] \).

These instructions can cause the following floating-point exceptions:

- **Invalid Operation**
  - When the floating-point input is NaN or Infinity or when a numerical value cannot be represented within the destination register.
- **Inexact**
  - When the numeric result differs from the input value.
- **Input Denormal**
  - When flushing denormalized numbers to zero is enabled and the denormal input is replaced by a zero, see Flushing denormalized numbers to zero on page A1-58 and Input Denormal exceptions on page A1-65.

---

Note
An out of range fixed-point result is saturated to the destination size.

For more information, see Floating-point exceptions and exception traps on page A1-64.

### A1.4.9 Polynomial arithmetic over \{0, 1\}

Some SIMD instructions that operate on SIMD&FP registers can operate on polynomials over \{0, 1\}, see Supported data types on page A1-42. The polynomial data type represents a polynomial in \( x \) of the form \( b_{n-1}x^{n-1} + \ldots + b_1x + b_0 \) where \( b_k \) is bit\([k]\) of the value.

The coefficients 0 and 1 are manipulated using the rules of Boolean arithmetic:
- \( 0 + 0 = 1 + 1 = 0 \).
- \( 0 + 1 = 1 + 0 = 1 \).
- \( 0 \times 0 = 0 \times 1 = 1 \times 0 = 0 \).
- \( 1 \times 1 = 1 \).

That is:
- Adding two polynomials over \{0, 1\} is the same as a bitwise exclusive OR.
- Multiplying two polynomials over \{0, 1\} is the same as integer multiplication except that partial products are exclusive-ORed instead of being added.

A64, A32, and T32 provide instructions for performing polynomial multiplication of 8-bit values.
- For AArch32, see VMUL (integer and polynomial) on page F6-8575 and VMULL (integer and polynomial) on page F6-8581.
- For AArch64, see PMUL on page C7-2535 and PMULL, PMULL2 on page C7-2537.

The Cryptographic Extension adds the ability to perform long polynomial multiplies of 64-bit values. See PMULL, PMULL2 on page C7-2537.

#### Pseudocode description of polynomial multiplication

In pseudocode, polynomial addition is described by the EOR operation on bitstrings.

Polynomial multiplication is described by the PolynomialMult() function defined in Chapter J1 Armv8 Pseudocode.
A1.5 Floating-point support

Note

The architecture includes the following types of floating-point instructions:

- Scalar floating-point instructions that operate on the lowest numbered element of the SIMD&FP registers.
- Advanced SIMD floating-point instructions that operate on multiple elements of the SIMD&FP registers.
- If FEAT_SVE is implemented, AArch64 SVE instructions that operate on multiple elements of the scalable vector registers, in which the SIMD&FP registers occupy the least significant 128 bits.

The architecture can support the following levels of support for Advanced SIMD and floating-point instructions:

- Full Advanced SIMD and floating-point support without floating-point exception trapping.
- Full Advanced SIMD and floating-point support with floating-point exception trapping.
- No Advanced SIMD or floating-point support. This option is licensed only for implementations targeting specialized markets.

Note

All Armv8-A systems that support standard operating systems with rich application environments provide hardware support for Advanced SIMD and floating-point instructions. All Armv9-A systems that support standard operating systems with rich application environments also provide hardware support for SVE2 instructions. It is a requirement of the ARM Procedure Call Standard for AArch64, see Procedure Call Standard for the Arm 64-bit Architecture.

The Arm architecture supports single-precision (32-bit) and double-precision (64-bit) floating-point data types and arithmetic as defined by the IEEE 754 floating-point standard. It also supports the half-precision (16-bit) floating-point data type for data storage, by supporting conversions between single-precision and half-precision data types and double-precision and half-precision data types. When FEAT_FP16 is implemented, it also supports the half-precision floating-point data type for data-processing operations.

The SIMD instructions provide packed Single Instruction Multiple Data (SIMD) and single-element scalar operations, and support:

- Single-precision and double-precision arithmetic in AArch64 state.
- Single-precision arithmetic only in AArch32 state.
- When FEAT_FP16 is implemented, half-precision arithmetic is supported in AArch64 and AArch32 states.

Floating-point support in AArch64 state SIMD is IEEE 754-2008 compliant with:

- Configurable rounding modes.
- Configurable Default NaN behavior.
- Configurable flushing to zero of denormalized numbers.

Floating-point computation using AArch32 Advanced SIMD instructions remains unchanged from Armv7. A32 and T32 Advanced SIMD floating-point always uses Arm standard floating-point arithmetic and performs IEEE 754 floating-point arithmetic with the following restrictions:

- Denormalized numbers are flushed to zero, see Flushing denormalized numbers to zero on page A1-58.
- Only default NaNs are supported, see The Default NaN on page A1-60.
- The Round to Nearest rounding mode is used.
- Untrapped floating-point exception handling is used for all floating-point exceptions.

If floating-point exception trapping is supported, floating-point exceptions, such as Overflow or Divide by Zero, can be handled without trapping. This applies to both SIMD and floating-point operations. When handled in this way, a floating-point exception causes a cumulative status register bit to be set to 1 and a default result to be produced by the operation. For more information about floating-point exceptions, see Floating-point exceptions and exception traps on page A1-64.
In AArch64 state, the following registers control floating-point operation and return floating-point status information:

- The Floating-Point Control Register, FPCR, controls:
  - The half-precision format where applicable, FPCR.AHP bit.
  - Default NaN behavior, FPCR.DN bit.
  - Flushing of denormalized numbers to zero, FPCR.{FZ, FZ16} bits. If FEAT_FP16 is not implemented, FPCR.FZ16 is RES0.
  - Rounding mode support, FPCR.Rmode field.
  - Len and Stride fields associated with execution in AArch32 state, and only supported for a context save and restore from AArch64 state. These fields are obsolete in Armv8 and can be implemented as RAZ/WI. If they are implemented as RW and are programmed to a nonzero value, they make some AArch32 floating-point instructions UNDEFINED.
  - Floating-point exception trap controls, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits, see Floating-point exceptions and exception traps on page A1-64.

- The Floating-Point Status Register, FPSR, provides:
  - Cumulative floating-point exceptions flags, FPSR.{IDC, IXC, UFC, OFC, DZC, IOC and QC}.
  - The AArch32 floating-point comparison flags {N,Z,C,V}. These bits are RES0 if AArch32 floating-point is not implemented.

**Note**

In AArch64 state, the process state flags, PSTATE.{N,Z,C,V} are used for all data-processing compares and any associated conditional execution.

If FEAT_FlagM2 is implemented, the instructions AXFLAG and XFLAG convert between the Arm condition flag format and an alternative format shown in Relationship between ARM format and alternative format PSTATE condition flags on page C6-1168.

AArch32 state provides a single Floating-Point Status and Control Register, FPSCR, combining the FPCR and FPSR fields.

For system level information about the SIMD and floating-point support, see Advanced SIMD and floating-point support on page G1-8996.

### A1.5.1 Instruction support

The floating-point instructions support:

- Load and store for single elements and vectors of multiple elements.

**Note**

Single elements are also referred to as scalar elements.

- Data processing on single and multiple elements for both integer and floating-point data types.
- When FEAT_FCMA is implemented, complex number arithmetic.
- Floating-point conversion between different levels of precision.
- Conversion between floating-point, fixed-point integer, and integer data types.
- Floating-point rounding.

For more information on floating-point instructions in AArch64 state, see Chapter C3 A64 Instruction Set Overview.

For more information on floating-point instructions in AArch32 state, see Chapter F2 The AArch32 Instruction Sets Overview.
A1.5.2 Floating-point standards, and terminology

The Arm architecture includes support for all the required features of ANSI/IEEE Std 754-2008, *IEEE Standard for Binary Floating-Point Arithmetic*, referred to as IEEE 754-2008. However, some terms in this Manual are based on the 1985 version of this standard, referred to as IEEE 754-1985:

- Arm floating-point terminology generally uses the IEEE 754-1985 terms. This section summarizes how IEEE 754-2008 changes these terms.
- References to IEEE 754 that do not include the issue year apply to either issue of the standard.

Table A1-4 on page A1-57 shows how the terminology in this Manual differs from that used in IEEE 754-2008.

<table>
<thead>
<tr>
<th>This manual</th>
<th>IEEE 754-2008</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized *</td>
<td>Normal</td>
</tr>
<tr>
<td>Denormal, or denormalized</td>
<td>Subnormal</td>
</tr>
<tr>
<td>Round towards Minus Infinity (RM)</td>
<td>roundTowardsNegative</td>
</tr>
<tr>
<td>Round towards Plus Infinity (RP)</td>
<td>roundTowardsPositive</td>
</tr>
<tr>
<td>Round towards Zero (RZ)</td>
<td>roundTowardZero</td>
</tr>
<tr>
<td>Round to Nearest (RN)</td>
<td>roundTiesToEven</td>
</tr>
<tr>
<td>Round to Nearest with Ties to Away</td>
<td>roundTiesToAway</td>
</tr>
<tr>
<td>Rounding mode</td>
<td>Rounding-direction attribute</td>
</tr>
</tbody>
</table>

* Normalized number is used in preference to normal number, because of the other specific uses of normal in this Manual.

A1.5.3 Arm standard floating-point input and output values

The Arm architecture provides full IEEE 754 floating-point arithmetic support. In AArch32 state, floating-point operations performed using Advanced SIMD instructions are limited to *Arm standard floating-point operation*, regardless of the selected rounding mode in the FPSCR. Unlike AArch32, AArch64 SIMD floating point arithmetic is performed using the rounding mode selected by the FPCR.

Arm standard floating-point arithmetic supports the following input formats defined by the IEEE 754 floating-point standard:

- Zeros.
- Normalized numbers.
- Denormalized numbers are flushed to 0 before floating-point operations, see *Flushing denormalized numbers to zero on page A1-58*.
- NaNs.
- Infinities.

Arm standard floating-point arithmetic supports the Round to Nearest (roundTiesToEven) rounding mode defined by the IEEE 754 standard.

Arm standard floating-point arithmetic supports the following output result formats defined by the IEEE 754 standard:

- Zeros.
- Normalized numbers.
- Results that are less than the minimum normalized number are flushed to zero, see *Flushing denormalized numbers to zero on page A1-58*.
- NaNs produced in floating-point operations are always the default NaN, see *The Default NaN on page A1-60*.
- Infinities.
A1.5.4 Flushing denormalized numbers to zero

For this section if FEAT_AFP is not implemented, the behavior is the same as if FPCR.AH == 0, FPCR.FZ == 0 and FPCR.NEP == 0.

Calculations involving denormalized numbers and Underflow exceptions can reduce the performance of floating-point processing. For many algorithms, replacing the denormalized operands and Intermediate results with zeros can recover this performance, without significantly affecting the accuracy of the final result. Arm floating-point implementations allow denormalized numbers to be flushed to zero to permit this optimization.

If a number value satisfies the condition 0 < Abs(value) < MinNorm, it is treated as a denormalized number.

MinNorm is defined as follows:
- For half-precision numbers, MinNorm is $2^{-14}$.
- For single-precision and BFloat16 numbers, MinNorm is $2^{-126}$.
- For double-precision numbers, MinNorm is $2^{-1022}$.

Flushing denormals to zero is incompatible with the IEEE 754 standard, and must not be used when IEEE 754 compatibility is a requirement. Enabling flushing of denormals to zero must be done with care. Although it can improve performance on some algorithms, there are significant limitations on its use. These are application-dependent:
- On many algorithms, it has no noticeable effect, because the algorithm does not usually process denormalized numbers.
- On other algorithms, it can cause exceptions to occur and can seriously reduce the accuracy of the results of the algorithm.

Flushing denormalized inputs to zero

If flushing denormalized inputs to zero is enabled for an instruction and a data type, and an input to that instruction is a denormalized number of that data type, the input operand is flushed to zero, and its sign bit is not changed.

If a floating-point operation has an input denormalized number that is flushed to zero, for all purposes within the instruction other than calculating Input Denormal floating-point exceptions, all inputs that are denormalized numbers are treated as though they were zero with the same sign as the input.

For floating-point instructions, if the instruction processes half-precision inputs, flushing denormalized inputs to zero can be controlled as follows:
- If FPCR.FZ16 == 0, denormalized half-precision inputs are not flushed to zero.
- If FPCR.FZ16 == 1, for half-precision data-processing instructions, flushing of input denormalized numbers to zero occurs as follows:
  - If an instruction does not convert a half-precision input to a higher precision output, all input denormalized numbers are flushed to zero.
  - If an instruction converts a half-precision input to a higher precision output, input denormalized numbers are not flushed to zero.

If FPCR.FIZ == 1, or both FPCR.AH == 0 and FPCR.FZ == 1, for Advanced SIMD and floating-point instructions other than FABS and FNEG, all single-precision, double-precision and BF16 input operands that are denormalized numbers are flushed to zero. Half-precision input operands are not flushed to zero.

If FPCR.FZ == 0, for Advanced SIMD, floating-point and BF16 instructions, the FPCR.FZ setting does not cause denormalized inputs to be flushed to zero, although other factors might cause denormalized numbers to be flushed to zero.

If FPCR.AH == 1, regardless of the value of FPCR.FIZ, all of the following instructions flush all input denormal numbers to zero:
- BFloat instructions: BFCVT, BFCVTN, BFCVTN2, BFMLALB, BFMLALT (by element), BFMLALB, BFMLALT (vector), and BFCVTNT.
• Single-precision and double-precision instructions: FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.

**Flushing to zero of denormalized numbers as Intermediate results of some BF16 instructions**

BF16 arithmetic instructions BFDOT (by element), BFDOT (vector), BFMMMLA in AArch64, and VDOT (by element), VDOT (vector), VMMMLA in AArch32 when working with BF16 inputs, convert BF16 input values to IEEE single-precision format, and calculate N-way dot-products, accumulating the products in single-precision accumulators.

If a BF16 arithmetic instruction processes an intermediate result that is a single-precision denormalized number, the Intermediate result is unconditionally flushed to zero.

**Flushing denormalized outputs to zero**

If a denormalized output is flushed to zero, the output is returned as zero with the same sign bit as the denormalized output value.

If FPCR.AH == 0, for half-precision, single-precision and double-precision numbers, the test for a denormalized number for the purpose of flushing the output to zero occurs before rounding.

If FPCR.AH == 1, and if output flushing is caused by FPCR.FZ == 1 or FPCR.FZ16 == 1, for half-precision, single-precision and double-precision numbers, the test for a denormalized number for the purpose of flushing the output to zero occurs after rounding using an unbounded exponent.

If FPCR.AH == 1, and if FPCR.FZ == 0, Advanced SIMD, floating-point and BF16 instructions, for single-precision, double-precision and BF16 outputs, the FPCR.FZ setting does not cause denormalized outputs to be flushed to zero, although other factors might cause denormalized outputs to be flushed to zero.

BFDOT (by element), BFDOT (vector), and BFMMMLA instructions unconditionally flush denormalized output numbers to zero.

If FPCR.AH == 0, for Advanced SIMD, floating-point, and BF16 instructions, for single-precision, double-precision and BF16 outputs, flushing denormalized numbers to zero can be controlled as follows:

- If FPCR.FZ == 0, the FPCR.FZ setting does not cause denormalized output numbers to be flushed to zero, although other factors might cause denormalized output numbers to be flushed to zero.
- If FPCR.FZ == 1, for all Advanced SIMD, floating-point and BF16 instructions other than FABS and FNEG, all single-precision, double-precision, and BF16 outputs that are denormalized numbers are flushed to zero.

If FPCR.FZ16 == 0 denormalized half-precision output numbers are not flushed to zero.

If FPCR.FZ16 == 1, for floating-point instructions other than FABS, FNEG, FMAX*, and FMIN*, if the instruction processes half-precision numbers, flushing denormalized output numbers to zero can be controlled as follows:

- Instructions that convert between half-precision and single-precision numbers do not flush denormalized half-precision output numbers to zero.
- Instructions that convert between half-precision and double-precision numbers do not flush denormalized half-precision output numbers to zero.
- All other half-precision data-processing instructions flush all denormalized half-precision output numbers to zero.

If FPCR.AH == 1 and FPCR.FZ == 1, for Advanced SIMD, floating-point and BF16 instructions, all of the following apply:

- For all floating-point operations other than FABS, FNEG, FMAX*, and FMIN*, all single-precision and double-precision denormalized output operands are flushed to zero.
- For FABS, FNEG, FMAX*, and FMIN*, denormalized output operands are not flushed to zero.
If \( \text{FPCR.AH} = 1 \), regardless of the value of \( \text{FPCR.FZ} \) bit, for both Advanced SIMD and SVE, all of the following instructions flush all output denormal numbers to zero:

- BFloat instructions: \( \text{BFCVT} \), \( \text{BFCVTN} \), \( \text{BFCVTN2} \), \( \text{BFMLALB} \), \( \text{BFMLALT} \) (by element), \( \text{BFMLALB} \), \( \text{BFMLALT} \) (vector), and \( \text{BFCVTNT} \).
- Single-precision and double-precision instructions: \( \text{FRECPE} \), \( \text{FRECPS} \), \( \text{FRECPX} \), \( \text{FRSQRTS} \), and \( \text{FRSQRT} \).

### A1.5.5 NaN handling and the Default NaN

The IEEE 754 standard defines a NaN as a number with all exponent bits set to 1 and a nonzero number in the mantissa. The Arm architecture additionally defines a Default NaN which does not follow this format.

The IEEE 754 standard specifies that the sign bit of a NaN has no significance.

For a quiet NaN output derived from a signaling NaN operand, the most significant fraction bit is set to 1.

A PE is forbidden to generate a NaN whose value is strongly correlated to the values of non-NaN inputs as a speculative result of a floating-point calculation not involving NaN inputs.

#### The Default NaN

The Default NaN is encoded as described in Table A1-5 on page A1-60.

<table>
<thead>
<tr>
<th>Table A1-5 Default NaN encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign bit</td>
</tr>
<tr>
<td>( \text{If FPCR.AH} = 0 )</td>
</tr>
<tr>
<td>Sign bit</td>
</tr>
<tr>
<td>( \text{If FPCR.AH} = 1 )</td>
</tr>
<tr>
<td>Exponent</td>
</tr>
<tr>
<td>Fraction</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

IF \( \text{FPCR.DN} = 1 \), for floating-point instructions other than \( \text{FABS} \), \( \text{FMAX} \), \( \text{FMIN} \), and \( \text{FNEG} \), if any input to a floating-point operation performed by the instruction is a NaN, the output of the floating-point operation is the Default NaN.

For \( \text{FABS} \), \( \text{FNEG} \), \( \text{FMAX} \), \( \text{FMIN} \), Default NaN behavior is explained in the instruction description.

If \( \text{FPCR.DN} = 0 \), for floating-point processing the Default NaN is not used for NaN propagation.

If a floating-point instruction performs a floating-point operation, and that instruction generates an untrapped Invalid Operation floating-point exception for a reason other than one of the inputs being a signaling NaN, the output is the Default NaN.

#### NaN handling

The IEEE 754 standard does not specify which input NaN is used as the output NaN. Therefore, where the Arm architecture specifies which input NaN to use, this is an addition to the requirements in the IEEE 754 standard.

Depending on the operation, the exact value of a derived quiet NaN output might have both a different sign and a different number of fraction bits from its source. See instruction descriptions for details.
**NaN propagation**

If an output NaN is derived from one of the operands, how the input NaN propagates to the output depends on the instruction and the number of operands.

If an output NaN is derived from an input NaN and if the size of the output format is the same as the input format, then all of the following apply:

- If the input NaN is a quiet NaN, the output NaN is the same as the input NaN.
- If the input NaN is a signaling NaN, the output NaN is derived as follows:
  - If the handling of a signaling NaN by the instruction generates an Invalid Operation exception, the output NaN is the quieted version of the input NaN.
  - If the handling of a signaling NaN by the instruction does not generate an Invalid Operation exception, the output NaN is the same as the input NaN. This case applies for FABS, FNEG, and FTSSEL instructions.

If an output NaN is derived from an input NaN and if the size of the output format is larger than the input format, all of the following apply:

- If the input NaN is a quiet NaN, the output NaN is the same as the input NaN except that the mantissa is zero-extended in the low-order bit to fit the output format, and the exponent field is set to all ones.
- If the input NaN is a signaling NaN, the output NaN is the quieted version of the input NaN, except that the mantissa is zero-extended in the low-order bits and the exponent field is set to all ones.

If an output NaN is derived from an input NaN and if the size of the output format is smaller than the input format, all of the following apply:

- If the input NaN is a quiet NaN, the output NaN is the same as the input NaN except that the mantissa is truncated in the lower-order bits to fit the output format, and the exponent field is set to all ones.
- If the input NaN is a signaling NaN, the output NaN is the quieted version of the input NaN except that the mantissa is truncated in the lower-order bits to fit the output format, and the exponent field is set to all ones.

For the following descriptions, the term “first operand” and “second operand” relate to the left-to-right ordering of the arguments of the pseudocode function that describes the operation.

If $\text{FPCR.DN} = 0$, for Advanced SIMD, floating-point, or BF16 instructions that perform a floating-point operation, other than $\text{FABS}$, $\text{FNEG}$, $\text{FMAX}$, and $\text{FMIN}$, NaN outputs that derive from NaN inputs are derived as follows:

- If all of the following apply, an instruction outputs a quiet NaN derived from the first signaling NaN operand:
  - $\text{FPCR.AH} = 0$.
  - At least one operand is a signaling NaN.
  - The instruction is not trapped.
- If all of the following apply, an instruction outputs a quiet NaN derived from the first NaN operand:
  - $\text{FPCR.AH} = 0$.
  - At least one operand is a NaN, but none of the operands is a signaling NaN.
  - The instruction is not trapped.
- If all of the following apply, the output is a quiet NaN derived from the NaN operand:
  - $\text{FPCR.AH} = 1$.
  - The operation has two floating-point inputs.
  - The operation has only one NaN operand.
- If all of the following apply, the output is a NaN derived from the $\langle Vn \rangle$, $\langle Hn \rangle$, $\langle Sn \rangle$, or $\langle Dn \rangle$ register:
  - $\text{FPCR.AH} = 1$.
  - The operation has two floating-point inputs.
  - The operation has two NaN operands.
• If all of the following apply, the output is a NaN derived from the NaN held in the <Vn>, <Hn>, <Sn>, or <Dn> register:
  — FPCR.AH == 1
  — The instruction is one of: BFMLALB, BFMLALT (by element), BFMLALB, BFMLALT (vector),
    FCMLA, FMADD, FMLA (by element), FMLA (vector), FMLAL, FMLAL2 (by element), FMLAL,
    FMLAL2 (vector), FMLS (by element), FMLS (vector), FMLSL, FMLSL2 (by element), FMLSL,
    FMLSL2 (vector), FMSUB, FNADD, and FNMSUB.
  — One of the following applies:
    — The operation has three NaN operands.
    — The operation has two NaN operands and the <Vn>, <Hn>, <Sn> or <Dn> register holds a NaN.

• If all of the following apply, the output is a NaN derived from the NaN held in the <Vm>, <Hm>, <Sm>, or <Dm> register:
  — FPCR.AH == 1
  — The instruction is one of: BFMLALB, BFMLALT (by element), BFMLALB, BFMLALT (vector),
    FCMLA, FMADD, FMLA (by element), FMLA (vector), FMLAL, FMLAL2 (by element), FMLAL,
    FMLAL2 (vector), FMLS (by element), FMLS (vector), FMLSL, FMLSL2 (by element), FMLSL,
    FMLSL2 (vector), FMSUB, FNADD, and FNMSUB.
  — The operation has two NaN operands and the <Vn>, <Hn>, <Sn> or <Dn> register does not hold a NaN.

If FPCR.AH == 0, and an output NaN is derived from an input NaN, the pseudocode functions FPAbs(), FPNeg(),
FPTrigMAdd(), and FPTrigSSel() can change the sign of the NaN.

If FPCR.AH == 1, and an output NaN is derived from an input NaN, for all cases, the sign bit of the NaN is unchanged.

For FMAX* and FMIN*, the NaN handling is described in the instruction.

A1.5.6 Rounding

The rounding mode specifies how the exact result of a floating-point operation is rounded to a value in the destination format.

The rounding mode is either determined by the rounding mode control field FPCR.RMode or by the instruction.

If FPCR.AH == 1, for any value of FPCR.RMode, the following instructions use Round to Nearest on outputs:

• BF16 instructions BFCVT, BFCVTN, BFCVTN2, BFMLALB, BFMLALT (by element), BFMLALB,
  BFMLALT (vector), and the SVE instruction BFCVNT.

• Single-precision and double-precision instructions FRECPE, FRECPS, FRECPX, FRSQRTE, and
  FRSQRTS.

• Half-precision instructions FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS.

The rounding mode control field FPCR.RMode can select the following rounding modes:

• Round to Nearest (RN) mode.
• Round towards Plus Infinity (RP) mode.
• Round towards Minus Infinity (RM) mode.
• Round towards Zero (RZ) mode.

The following two additional rounding modes are not selected by FPCR.RMode, but are used by some instructions:

• Round to Odd mode.
• Round to Nearest with ties to away mode.
Round to Nearest mode

Round to Nearest rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format as follows:

- If the value before rounding has an absolute value that is too large to represent in the output format, the rounded value is an Infinity. The sign of the rounded value is the same as the sign of the value before rounding.
- If the value before rounding has an absolute value that is not too large to represent in the output format, the result is calculated as follows:
  - If the two nearest floating-point numbers bracketing the value before rounding are equally near, the result is the number with an even least significant digit.
  - If the two nearest floating-point numbers bracketing the value before rounding are not equally near, the result is the floating-point number nearest to the value before rounding.

Round towards Plus Infinity mode

Round towards Plus Infinity rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the floating-point number in the output format that is closest to and not less than the value before rounding. The result can be plus infinity.

Round towards Minus Infinity mode

Round towards Minus Infinity rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the number in the output format that is closest to and not greater than the value before rounding. The result can be minus infinity.

Round towards Zero mode

Round towards Zero rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the floating-point number in the output format that is closest to and not greater in absolute value than the value before rounding.

Round to Nearest with Ties to Away

Round to Nearest with Ties to Away rounding mode is used by the FCVTAS (scalar), FCVTAS (vector), FCVTAU (scalar), FCVTAU (vector), FRINTA (scalar), and FRINTA (vector) instructions.

Round to Nearest with Ties to Away rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format as follows:

- If the value before rounding has an absolute value that is too large to represent in the output format, the rounded value is an Infinity, the sign of the rounded value is the same as the sign of the value before rounding.
- If the value before rounding has an absolute value that is not too large to represent in the output format, the result is calculated as follows:
  - If the two nearest floating-point numbers bracketing the value before rounding are equally near, the result is the larger number.
  - If the two nearest floating-point numbers bracketing the value before rounding are not equally near, the result is the floating-point number nearest to the value before rounding.

Round to Odd mode

Round to Odd mode is not defined by IEEE 754, and differs between the FCVTXN, FCVTXN2 instructions, and the BFDO (by element), BFDO (vector), and BFMMMLA instructions.

The FCVTXN, FCVTXN2 instructions use Round to Odd rounding mode. If the result of the rounding is inexact, the least significant bit of the mantissa is forced to 1.
Round to Odd rounding mode can avoid double rounding errors when a floating-point value is converted to a lower precision destination format through an intermediate precision format.

**Example A1-1 Converting 64-bit floating-point format to 16-bit floating-point format**

A 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value using the following steps:

1. Use an FCVTXN instruction to produce a 32-bit value.
2. Use another instruction with the required rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

For BFDOT (by element), BFDOT (vector), and BFMM MLA instructions, if the intermediate format has at least two more bits of precision than the result format, Round to Odd mode is used and operates as follows:

- If the rounded value is inexact, the least significant bit of the fraction is set to 1.
- If the value is too large to represent in the single-precision format, the rounded value is a single-precision Infinity, the sign of the rounded value is the same as the sign of the value before rounding.

### A1.5.7 Floating-point exceptions and exception traps

Execution of a floating-point instruction, or execution of an Advanced SIMD instruction that performs floating-point operations, can generate an exceptional condition, called a *floating-point exception*. Predicated SVE floating-point instructions only generate floating-point exceptions in response to floating-point operations performed on *Active elements*.

**Note**

In AArch64 state, an Advanced SIMD or SVE instruction that operates on floating-point values can perform multiple floating-point operations. Therefore, this section describes the handling of a floating-point exception on an operation, rather than on an instruction.

The architecture does not support asynchronous reporting of floating-point exceptions.

For each of the following floating-point exceptions, it is IMPLEMENTATION DEFINED whether an implementation includes synchronous exception generation:

- Input Denormal.
- Inexact.
- Underflow.
- Overflow.
- Divide by Zero.
- Invalid Operation.

If an implementation does not support synchronous exception generation from a floating-point exception, then that synchronous exception is never generated and all statements about synchronous exception generation from that floating-point exception do not apply to the implementation.

Synchronous exception generation by floating-point exceptions is enabled using the FPCR as follows:

- For each floating-point exception that supports synchronous exception generation, the relevant control bit chosen from FPCR.\{IDE, IXE, UFE, OFE, DZE, IOE\} is used to enable synchronous exception generation.
- For each floating-point exception that does not support synchronous exception generation, the relevant bit chosen from FPCR.\{IDE, IXE, UFE, OFE, DZE, IOE\} is RAZ/WI.
**Input Denormal exceptions**

The cumulative floating-point exception bit FPSR.IDC, and the trap enable bit FPCR.IDE both relate to Input Denormal exceptions.

If an input denormalized number is flushed to zero, the occurrence of the Input Denormal exception is determined using the value before flushing.

If an input denormalized number is flushed to zero, and FPCR.AH is 0, the occurrence of all floating-point exceptions, except Input Denormal, is determined treating the input value that is flushed to zero as zero.

If an input denormalized number is flushed to zero, and FPCR.AH is 1, the occurrence of all floating-point exceptions is determined treating the input value that is flushed to zero as zero.

If FPCR.AH is 0, when a single-precision or double-precision floating-point input is flushed to zero, an Input Denormal exception is generated.

If FPCR.AH is 1, and FPCR.FIZ is 0, if and only if none of the following apply, any operation that unpacks a denormalized floating-point input, other than unpacking a BFloat or half-precision value, generates an Input Denormal exception:

- One of the other operands of the instruction is a NaN.
- The operation generates an Invalid Operation floating-point exception.
- The operation generates a Divide-by-Zero floating-point exception.
- The instruction that generated the operation was one of: BFCVTN, BFCVTN2, BFCVT, and BFCVTNT.
- The denormalized floating-point input is flushed to zero.

When a half-precision floating-point value is flushed to zero, an Input Denormal exception is not generated.

If FPCR.AH is 1, or FPCR.FZ is 0, when FPCR.FIZ causes flushing of a denormalized number, an Input Denormal Exception is not generated.

**Inexact exceptions**

The cumulative floating-point exception bit FPSR.IXC and the trap enable bit FPCR.IXE both relate to Inexact exceptions.

If a denormalized output is flushed to zero, all of the following apply:

- If FPCR.AH is 1, an Inexact exception is generated.
- If FPCR.AH is 0, an Inexact exception is not generated.

If a result is not flushed to zero, and the result does not equal the result computed with unbounded exponent range and unbounded precision, then an Inexact exception is generated.

**Underflow exceptions**

The cumulative floating-point exception bit FPSR.UFC, and the trap enable bit FPCR.UFE both relate to Underflow exceptions.

If FPCR.AH is 1, for all floating points other than BFu1H() and BFu1AdH() which are used by BF00T and BFMLA, for the purpose of underflow floating-point exception generation, a denormalized number is detected after rounding with an unbounded exponent.

If FPCR.AH is 0, for the purpose of underflow floating-point exception generation, a denormalized number is detected before rounding is applied.

If the result of a floating-point operation is a denormalized number that is not flushed to zero, then:

- If FPCR.UFE is 0, and the result is inexact, then the underflow floating-point exception is generated.
- If FPCR.UFE is 1, then the underflow floating-point exception is generated.

If the result of a floating-point operation is a denormalized number that is flushed to zero, then the Underflow floating-point exception is not generated.
Overflow exceptions

The cumulative floating-point exception bit FPSR.OFC, and the trap enable bit FPCR.OFE both relate to Overflow exceptions.

If the output of an instruction rounded with an unbounded exponent is greater than the maximum normalized number for the output precision, an overflow exception is generated.

If an untrapped Overflow exception is generated, the result is determined by the rounding mode and the sign of the result before rounding as follows:

- Round to Nearest carries all overflows to infinity with the sign of the result before rounding.
- Round towards Plus Infinity carries negative overflows to the most negative finite number of the output precision, and carries positive overflows to plus infinity.
- Round towards Minus Infinity carries positive overflows to the largest finite number of the output precision, and carries negative overflows to minus infinity.
- Round towards Zero carries all overflows to the output precision’s largest finite number with the sign of the result before rounding.

Divide by Zero exceptions

The cumulative floating-point exception bit FPSR.DZC, and the trap enable bit FPCR.DZE both relate to Divide by Zero exceptions.

If a floating-point operation divides a finite nonzero number by zero, a Divide by Zero exception is generated.

If a floating-point operation divides a finite nonzero number by zero, and the Divide by Zero exception is untrapped, the result is a correctly signed infinity.

Invalid Operation exceptions

The cumulative floating-point exception bit FPSR.IOC, and the trap enable bit FPCR.IOE both relate to Invalid Operation exceptions.

For any floating-point instruction that performs a floating-point operation, if any of the following apply, the instruction generates an Invalid Operation exception:

- At least one operand is a signaling NaN.
- Magnitude subtraction of infinities.
- Multiplying a zero by an infinity.
- Dividing a zero by a zero.
- Dividing an infinity by an infinity.
- Square root of an operand that is less than zero.

If the input is one of: a quiet NaN, an infinity, or a number that overflows the values that can be represented in the output format, and if another exception is not generated to signal the condition, then a conversion from floating-point to either integer or fixed-point format, generates an Invalid Operation floating-point exception.

For the signaling compare instructions FCMPE and FCCMPE, if either of the source operands is any type of NaN, the instruction generates an Invalid Operation floating-point exception.

If FPCR.AH is 1, for FMAX (vector), FMAX (scalar), FMAXP (scalar), FMAXP (vector), FMAXV, FMIN (vector), FMIN (scalar), FMINP (scalar), FMINP (vector), and FMINV, if either input is any type of NaN, then an Invalid Operation floating-point exception is generated.

Operations that do not generate floating point exceptions

BFDOT (by element), BFDOT (vector), and BFMMMLA do not generate floating-point exceptions.
If FPCR.AH is 1, all of the following instructions do not generate any floating-point exceptions regardless of their input values:

- BF16 instructions BFMLALB, BFMLALT (by element), BFMLALB, BFMLALT (vector), BFCVT, BFCVTN, BFCVTN2, and BFCVTNT.
- Single-precision, double-precision and half-precision instructions FRECPE, FRECPS, FRECPX, FRSQRTN, and FRSQRTE.
- Floating-point to integer and integer to floating-point rounding instructions: FCVTMS (scalar), FCVTMS (vector), FCVTMU (scalar), FCVTMU (vector), FCVTNS (scalar), FCVTNS (vector), FCVTNU (scalar), FCVTNU (vector), FCVTPS (scalar), FCVTPS (vector), FCVPUS (scalar), FCVPUS (vector), FCVTZS (scalar, fixed-point), FCVTZS (scalar), integer), FCVTZS (vector, fixed-point), FCVTZS (vector, integer), FCVTZU (scalar, fixed-point), FCVTZU (scalar, integer), FCVTZU (vector, fixed-point), FCVTAS (scalar), FCVTAS (vector), FCVTZU (vector, fixed-point), FCVTZS (vector, scalar), FCVTZS (vector, integer), FRINTA (scalar), FRINTA (vector), FRINTZ (scalar), FRINTZ (vector), FRINTM (scalar), FRINTM (vector), FRINTP (scalar), FRINTP (vector), FRINTN (scalar), FRINTN (vector), FRINTX (scalar), FRINTX (vector), FRINTI (scalar), FRINTI (vector), FRINT32X (scalar), FRINT32X (vector), FRINT32Z (scalar), FRINT32Z (vector), FRINT64X (scalar), FRINT64X (vector), FRINT64Z (scalar), and FRINT64Z (vector).

FPAbs() and FPNeg() are not classified as floating-point operations and all of the following apply to them:

- They cannot generate floating-point exceptions.
- The floating-point behavior described in the Flushing denormalized numbers to zero on page A1-58 does not apply to them.
- The floating-point behavior described in the section NaN handling and the Default NaN on page A1-60 does not apply to them.

Handling floating-point exceptions

If an implementation supports synchronous exception generation for floating-point exceptions, the synchronous exceptions generated by the floating-point exception traps are taken to the lowest Exception level that can handle such an exception and that is not at a lower Exception level than where the exception was generated.

If an implementation supports synchronous exception generation for floating-point exceptions in AArch64 state, all of the following apply:

- The registers that are presented to the exception handler are consistent with the state of the PE immediately before the instruction that caused the exception, except that an implementation is permitted to not restore the cumulative floating-point exception bits in the event of such an exception.
- When the execution of separate operations in separate SIMD elements causes multiple floating-point exceptions, the ESR_ELx reports one exception associated with one element that the instruction uses. The architecture does not specify which element is reported.

The AArch64.FPTrappedException() and FPProcessException() pseudocode functions describe the handling of trapped floating-point exceptions generated in AArch64 state.

Combinations of floating-point exceptions

More than one floating-point exception can occur on the same operation. The only combinations of floating-point exceptions that can occur are:

- Overflow with Inexact.
- Underflow with Inexact.
- If FPCR.AH is 0, Input Denormal with any other floating-point exceptions.
- If FPCR.AH is 1, Input Denormal with Inexact, Underflow, or Overflow.

If two floating-point exceptions occur on the same operation, the Input Denormal exception is treated as highest priority and the Inexact exception is treated as lowest priority.
Some floating-point instructions specify more than one floating-point operation, this is indicated by the pseudocode descriptions of the instruction. In these cases, it is possible for one instruction to generate multiple exceptions. Multiple exceptions from one instruction are prioritized as follows:

- If an exception generating operation outputs a result that is used by a second exception generating operation, the exception generated by the operation that outputs the result is treated as higher priority than the exception generated by the second operation that uses the result.

- If exception generating operations do not use the outputs of other exception generating operations, it is constrained unpredictable which floating-point exception is treated as higher priority. The exception prioritized might differ between different instances of the same two floating-point exceptions being generated on the same operation during execution of the instruction.

- A trapped Underflow exception has priority over a trapped inexact exception.

If none of the floating-point exceptions caused by an operation is trapped, any floating-point exception that occurs causes the associated cumulative exception flag bit in the FPSR to be set to 1.

When a floating-point exception is trapped, all of the following apply:

- When the trapped floating-point exception is taken, it is implementation defined whether the FPSR is restored to the value of the FPSR immediately before the instruction that generated the trapped floating-point exception.

When the trapped floating-point exception is taken, if the FPSR is not restored, it is constrained unpredictable which untrapped floating-point exceptions, if any, are indicated by the corresponding FPSR cumulative exception flag bits having the value 1.

- In the ESR_ELx to which the trapped exception is taken all of the following apply:
  - The highest priority trapped floating-point exception has a floating-point exception trapped bit set to 1.
  - If any other untrapped floating-point exceptions are generated by the same operation, each untrapped exception has a floating-point exception trapped bit set to 0. This applies to both higher priority and lower priority untrapped floating-point exceptions.
  - If any lower priority trapped floating-point exceptions are generated by the same operation, for each exception, it is constrained unpredictable whether the floating-point exception trapped bit is set to 1.

The architectural requirements for floating-point exception prioritization apply only to multiple floating-point exceptions generated on the same element of an Advanced SIMD or SVE operation. For trapped floating-point exceptions from Advanced SIMD or SVE instructions, the architecture does not define the floating-point exception prioritization between different elements of the instruction.
A1.6 The Arm memory model

The Arm memory model supports:

- Generating an exception on an unaligned memory access.
- Restricting access by applications to specified areas of memory.
- Translating virtual addresses (VAs) provided by executing instructions to physical addresses (PAs).
- Altering the interpretation of multi-byte data between big-endian and little-endian.
- Controlling the order of accesses to memory.
- Controlling caches and address translation structures.
- Synchronizing access to shared memory by multiple PEs.
- Barriers that control and prevent speculative access to memory.

VA support depends on the Execution state, as follows:

**AArch64 state**

Supports 64-bit virtual addressing, with the Translation Control Register determining the supported VA range. Execution at EL1 and EL0 supports two independent VA ranges, each with its own translation controls.

**AArch32 state**

Supports 32-bit virtual addressing, with the Translation Control Register determining the supported VA range. For execution at EL1 and EL0, system software can split the VA range into two subranges, each with its own translation controls.

The supported PA space is IMPLEMENTATION DEFINED, and can be discovered by system software.

Regardless of the Execution state, the Virtual Memory System Architecture (VMSA) can translate VAs to blocks or pages of memory anywhere within the supported PA space.

For more information, see:

**For execution in AArch64 state**

- Chapter B2 The AArch64 Application Level Memory Model.
- Chapter D7 The AArch64 System Level Memory Model.
- Chapter D8 The AArch64 Virtual Memory System Architecture.

**For execution in AArch32 state**

- Chapter E2 The AArch32 Application Level Memory Model.
- Chapter G4 The AArch32 System Level Memory Model.
- Chapter G5 The AArch32 Virtual Memory System Architecture.
Chapter A2
Armv8-A Architecture Extensions

This chapter introduces the Arm architecture versions and extensions. It contains the following sections:

- Armv8.0 architecture extensions on page A2-72.
- Architectural features within Armv8.0 architecture on page A2-76.
- The Armv8 Cryptographic Extension on page A2-80.
- The Armv8.1 architecture extension on page A2-82.
- The Armv8.2 architecture extension on page A2-86.
- The Armv8.3 architecture extension on page A2-96.
- The Armv8.4 architecture extension on page A2-101.
- The Armv8.5 architecture extension on page A2-107.
- The Armv8.6 architecture extension on page A2-111.
- The Armv8.7 architecture extension on page A2-113.
- The Armv8.8 architecture extension on page A2-117.
- The Performance Monitors Extension on page A2-121.
- The Reliability, Availability, and Serviceability Extension on page A2-122.
- The Statistical Profiling Extension (SPE) on page A2-123.
- The Scalable Vector Extension (SVE) on page A2-124.
- The Activity Monitors Extension (AMU) on page A2-125.
- The Memory Partitioning and Monitoring (MPAM) Extension on page A2-126.
A2.1 Armv8.0 architecture extensions

The original Armv8-A architecture is called Armv8.0. The following sections of this manual describe or summarize permitted extensions to Armv8.0:

- The Armv8 Cryptographic Extension on page A2-80.
- The Reliability, Availability, and Serviceability Extension on page A2-122.
- Event monitors on page D1-4683.
- The IVIPT Extension described Instruction caches on page D8-5215.
- Chapter H7 The PC Sample-based Profiling Extension.

Note

The naming convention of features in the Arm architecture has been redefined. For more information on how these names map to the legacy convention, see Appendix K15 Legacy Feature Naming Convention.

In addition to describing Armv8.0, this manual describes the following architectural extensions:

Features added to Armv8.0 in later releases

Architectural features and architectural requirements have been added to the original Armv8-A architecture. For more information, see:

- Additional functionality added to Armv8.0 in later releases on page A2-76.
- Architectural requirements within Armv8.0 architecture on page A2-79.

For more information, see Architectural features within Armv8.0 architecture on page A2-76.

The Armv8.1 architectural extension

The Armv8.1 architecture extension adds both:

- Architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.
- Architectural requirements. These are mandatory.

An implementation is Armv8.1 compliant if all of the following apply:

- It includes all of the Armv8.1 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.1 compliant implementation includes the optional architecture component or extension. See Architectural features added by Armv8.1 on page A2-82 for all of the Armv8.1 architectural features.
- It includes all of the Armv8.1 architectural requirements. Additional requirements of Armv8.1 on page A2-84 lists these requirements.

For more information, see The Armv8.1 architecture extension on page A2-82.

The Armv8.2 architectural extension

The Armv8.2 architecture extension is an extension to Armv8.1. It adds both:

- Architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.
- Architectural requirements. These are mandatory.

An implementation is Armv8.2 compliant if all of the following apply:

- It is Armv8.1 compliant.
- It includes all of the Armv8.2 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.2 compliant implementation includes the optional architecture component or extension. The features are listed at: Architectural features added by Armv8.2 on page A2-86, which lists the original Armv8.2 architectural features.
The Armv8.2 architecture extension

The Armv8.2 architecture extension is an extension to Armv8.1. It adds both:

- Architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.
- Architectural requirements. These are mandatory.

An implementation is Armv8.2 compliant if all of the following apply:

- It includes all of the Armv8.2 architectural requirements. Additional requirements of Armv8.2 on page A2-93 lists these requirements.

For more information, see The Armv8.2 architecture extension on page A2-86.

The Armv8.3 architectural extension

The Armv8.3 architecture extension is an extension to Armv8.2. It adds both:

- Architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.
- Architectural requirements. These are mandatory.

An implementation is Armv8.3 compliant if all of the following apply:

- It is Armv8.2 compliant.
- It includes all of the Armv8.3 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.3 compliant implementation includes the optional architecture component or extension. The features are listed at:
  - Architectural features added by Armv8.3 on page A2-96, which lists the original Armv8.3 architectural features.
  - Features added to the Armv8.3 extension in later releases on page A2-98, which lists additional Armv8.3 architectural features.
- It includes all of the Armv8.3 architectural requirements. Additional requirements of Armv8.3 on page A2-98 lists these requirements.

For more information, see The Armv8.3 architecture extension on page A2-96.

The Armv8.4 architectural extension

The Armv8.4 architecture extension is an extension to Armv8.3. It adds architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.

An implementation is Armv8.4 compliant if all of the following apply:

- It is Armv8.3 compliant.
- It includes all of the Armv8.4 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.4 compliant implementation includes the optional architecture component or extension. See Architectural features added by Armv8.4 on page A2-101 for all of the Armv8.4 architectural features.

For more information, see The Armv8.4 architecture extension on page A2-101.

The Armv8.5 architectural extension

The Armv8.5 architecture extension is an extension to Armv8.4. It adds architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.

An implementation is Armv8.5 compliant if all of the following apply:

- It is Armv8.4 compliant.
- It includes all of the Armv8.5 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.5 compliant implementation includes the optional architecture component or extension. See Architectural features added by Armv8.5 on page A2-107 for all of the Armv8.5 architectural features.
- It includes all of the Armv8.5 architectural requirements. Additional requirements of Armv8.5 on page A2-109 lists these requirements.

For more information, see The Armv8.5 architecture extension on page A2-107.
The Armv8.6 architectural extension

The Armv8.6 architecture extension is an extension to Armv8.5. It adds architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.

An implementation is Armv8.6 compliant if all of the following apply:

- It is Armv8.5 compliant.
- It includes all of the Armv8.6 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.6 compliant implementation includes the optional architecture component or extension. See "Architectural features added by Armv8.6" on page A2-111 for all of the Armv8.6 architectural features.
- It includes all of the Armv8.6 architectural requirements. "Additional requirements of Armv8.6" on page A2-112 lists these requirements.

For more information, see "The Armv8.6 architecture extension" on page A2-111.

The Armv8.7 architectural extension

The Armv8.7 architecture extension is an extension to Armv8.6. It adds architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.

An implementation is Armv8.7 compliant if all of the following apply:

- It is Armv8.6 compliant.
- It includes all of the Armv8.7 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.7 compliant implementation includes the optional architecture component or extension. See "Architectural features added by Armv8.7" on page A2-113 for all of the Armv8.7 architectural features.
- It includes all of the Armv8.7 architectural requirements. "Additional requirements of Armv8.7" on page A2-116 lists these requirements.

For more information, see "The Armv8.7 architecture extension" on page A2-113.

The Armv8.8 architectural extension

The Armv8.8 architecture extension is an extension to Armv8.7. It adds architectural features. Some of these are mandatory, others are optional. Some features must be implemented together.

An implementation is Armv8.8 compliant if all of the following apply:

- It is Armv8.7 compliant.
- It includes all of the Armv8.8 architectural features that are mandatory. This includes all architectural features of an optional architecture component or extension that are defined as mandatory, if the Armv8.8 compliant implementation includes the optional architecture component or extension. See "Architectural features added by Armv8.8" on page A2-117 for all of the Armv8.8 architectural features.
- It includes all of the Armv8.8 architectural requirements. "Additional requirements of Armv8.8" on page A2-120 lists these requirements.

For more information, see "The Armv8.8 architecture extension" on page A2-117.

The Statistical Profiling Extension (SPE)

SPE is an optional extension to Armv8.2. That is, SPE requires the implementation of Armv8.2.

For more information, see "The Statistical Profiling Extension (SPE)" on page A2-123.

The Scalable Vector Extension (SVE)

SVE is an optional extension to Armv8.2. That is, SVE requires the implementation of Armv8.2.

For more information, see "The Scalable Vector Extension (SVE)" on page A2-124.

The Activity Monitors Extension (AMU)

AMU is an optional extension to Armv8.4. That is, AMU requires the implementation of Armv8.4.
For more information, see *The Activity Monitors Extension (AMU)* on page A2-125.

**The Memory Partitioning and Monitoring Extension (MPAM)**

MPAM is an optional extension to Armv8.2. That is, MPAM requires the implementation of Armv8.2.

For more information, see *The Memory Partitioning and Monitoring (MPAM) Extension* on page A2-126.

See also *Permitted implementation of subsets of Armv8.x and Armv8.(x+1) architectural features*.

---

**A2.1.1 Permitted implementation of subsets of Armv8.x and Armv8.(x+1) architectural features**

An Armv8.x compliant implementation can include any arbitrary subset of the architectural features of Armv8.(x+1), subject only to those constraints that require that certain features be implemented together.

Unless this manual permits otherwise, an Armv8.x compliant implementation does not include any features of Armv8.(x+2) or later.

---

**Note**

The addition of Armv8.(x+1) features to an Armv8.x compliant implementation is permitted only if the implementer has a license to Armv8.(x+1) in addition to the license to Armv8.x.
A2.2 Architectural features within Armv8.0 architecture

This includes architectural features and architectural requirements that have been added to the Armv8.0 architecture since the initial release, that were not part of the original Armv8-A architecture, see:

- Additional functionality added to Armv8.0 in later releases.
- Architectural requirements within Armv8.0 architecture on page A2-79.

A2.2.1 Additional functionality added to Armv8.0 in later releases

An implementation of Armv8.0 can include any or all of the features that this section describes.

The Armv8.0 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

**FEAT_SB, Speculation Barrier**

FEAT_SB introduces a barrier to control speculation.

This instruction is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.0 implementations and mandatory in Armv8.5 implementations.

The following fields identify the presence of FEAT_SB:

- ID_AA64ISAR1_EL1.SB.
- ID_ISAR6_EL1.SB.
- ID_ISAR6.SB.

For more information, see:

- Speculation Barrier (SB) on page B2-176.
- Barriers and CLREX instructions on page C3-251.
- Speculation Barrier (SB) on page E2-7183.
- Miscellaneous instructions on page F2-7273.

**FEAT_SSBS, FEAT_SBSS2, Speculative Store Bypass Safe**

FEAT_SSBS allows software to indicate whether hardware is permitted to load or store speculatively in a manner that could give rise to a cache timing side channel, which in turn could be used to derive an address from values loaded to a register from memory.

FEAT_SSBS2 provides controls for the MSR and MRS instructions to read and write the PSTATE.SSBS field.

FEAT_SSBS is supported in both AArch64 and AArch32 states. FEAT_SBSS2 is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.0 implementations.

The following fields identify the presence of FEAT_SSBS and FEAT_SSBS2:

- ID_AA64PFR1_EL1.SSBS.
- ID_PFR2_EL1.SSBS.
- ID_PFR2.SSBS.

For more information, see:

- Speculative Store Bypass Safe (SSBS) on page B2-172.
- Speculative Store Bypass Safe (SSBS) on page E2-7179.

**FEAT_CSV2, FEAT_CSV2_2, and FEAT_CSV2_3, Cache Speculation Variant 2**

FEAT_CSV2 adds a mechanism to identify if hardware cannot disclose information about whether branch targets trained in one hardware described context can control speculative execution in a different hardware described context.

FEAT_CSV2_2 adds the SCXTNUM_ELx registers, which provide a number that can be used to separate out different context numbers within their respective Exception levels for the purpose of protecting against side-channels using branch prediction and similar resources.
FEAT_CSV2_3 adds a mechanism to identify if hardware cannot disclose information about whether branch targets and branch history trained in one hardware described context can control speculative execution in a different hardware described context.

FEAT_CSV2 is supported in both AArch64 and AArch32 states.

FEAT_CSV2_2 and FEAT_CSV2_3 are supported in AArch64 state only.

FEAT_CSV2 is OPTIONAL in Armv8.0 implementations and mandatory in Armv8.5 implementations.

FEAT_CSV2_2 and FEAT_CSV2_3 are OPTIONAL in Armv8.0 implementations.

The following fields identify the presence of FEAT_CSV2:
- ID_AA64PFR0_EL1.CSV2.
- ID_PFR0_EL1.CSV2.
- ID_PFR0.CSV2.

The ID_AA64PFR0_EL1.CSV2 field identifies the presence of FEAT_CSV2_2 and FEAT_CSV2_3.

For more information, see:
- Restrictions on the effects of speculation on page E2-7179.

**FEAT_CSV2_1p1 and FEAT_CSV2_1p2, Cache Speculation Variant 2**

For each of these features, within a hardware-described context, branch targets trained for branches situated at one address can control speculative execution of branches situated at different addresses only in a hard-to-determine way.

FEAT_CSV2_1p1 does not support the SCXTNUM_ELx registers, and the contexts do not include the SCXTNUM_ELx register contexts.

FEAT_CSV2_1p2 adds the SCXTNUM_ELx registers, but the contexts do not include the SCXTNUM_ELx register contexts.

FEAT_CSV2_1p1 is supported in both AArch64 and AArch32 states.

FEAT_CSV2_1p2 is supported in AArch64 state only.

These features are OPTIONAL in Armv8.0 implementations.

The following fields identify the presence of FEAT_CSV2_1p1:
- ID_AA64PFR1_EL1.CSV2_frac.
- ID_PFR0_EL1.CSV2.
- ID_PFR0.CSV2.

The ID_AA64PFR1_EL1.CSV2_frac field identifies the presence of FEAT_CSV2_1p2.

For more information, see:
- Restrictions on the effects of speculation on page E2-7179.

**FEAT_CSV3, Cache Speculation Variant 3**

FEAT_CSV3 adds a mechanism to identify if hardware cannot disclose information about whether data loaded under speculation with a permission or domain fault can be used to form an address, generate condition codes, or generate SVE predicate values, to be used by instructions newer than the load in the speculative sequence.

This feature is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.0 implementations and mandatory in Armv8.5 implementations.

This feature is mandatory when FEAT_E0PD is implemented.

The following fields identify the presence of FEAT_CSV3:
- ID_AA64PFR0_EL1.CSV3.
- ID_PFR2_EL1.CSV3.
- ID_PFR2.CSV3.
FEAT_SPECRES, Speculation restriction instructions
FEAT_SPECRES adds the CFP RCTX, CPP RCTX, DVP RCTX, CFPRCTX, CPPRCTX, and DVP RCTX System instructions. These instructions prevent predictions based on information gathered from earlier execution within a particular execution context from affecting the later speculative execution within that context, to the extent that the speculative execution is observable through side channels.

This feature is supported in both AArch64 and AArch32 states.
This feature is optional in Armv8.0 implementations and mandatory in Armv8.5 implementations.

The following fields identify the presence of FEAT_SPECRES:
- ID_AA64ISAR1_EL1.SPECRES.
- ID_ISAR6_EL1.SPECRES.
- ID_ISAR6.SPECRES.

For more information, see:
- Prediction restriction instructions on page C5-661.
- Execution, data prediction and prefetching restriction System instructions on page D7-5070.
- Execution and data prediction restriction System instructions on page G4-9134.

FEAT_CP15SDISABLE2, CP15SDISABLE2
FEAT_CP15SDISABLE2 provides an implementation-defined mechanism, the CP15SDISABLE2 signal, which when asserted HIGH prevents writes to a set of Secure CP15 registers. This signal is analogous to the existing CP15SDISABLE signal.

This feature is supported only when EL3 is executing in AArch32 state.
This feature is optional in Armv8.0 implementations.

For more information, see The CP15SDISABLE and CP15SDISABLE2 input signals on page G5-9285.

FEAT_DoubleLock, Double Lock
FEAT_DoubleLock is the mnemonic used for the OS Double Lock.

If FEAT_DoPD is not implemented and FEAT_Debugv8p2 is implemented, this feature is optional.
If FEAT_DoPD is not implemented and FEAT_Debugv8p2 is not implemented, this feature is mandatory.
If FEAT_DoPD is implemented, this feature is not implemented.
The ID_AA64DFR0_EL1.DoubleLock field identifies that the OS Double Lock has been implemented.

FEAT_DGH, Data Gathering Hint
FEAT_DGH adds the Data Gathering Hint instruction to the hint space.

This instruction is added to the A64 instruction set only.
This feature is optional in Armv8.0 implementations.

The ID_AA64ISAR1_EL1.DGH field identifies the presence of FEAT_DGH.

For more information, see Hint instructions on page C3-251.

FEAT_ETS, Enhanced Translation Synchronization
FEAT_ETS adds support for enhanced memory access ordering requirements for translation table walks.

This feature is supported in both AArch64 and AArch32 states.
This feature is optional in Armv8.0 implementations and mandatory in Armv8.7 implementations.
The following fields identify the presence of FEAT_ETS:
- ID_AA64MMFR1_EL1.ETS.
- ID_MMFR5_EL1.ETS.
A2.2 Architectural features within Armv8.0 architecture

- ID_MMFR5.ETS.
  For more information, see:
  - Ordering of memory accesses from translation table walks on page D8-5099.
  - Ordering of translation table walks on page E2-7187.

**FEAT_nTLBPA, Intermediate caching of translation table walks**

FEAT_nTLBPA adds a mechanism to identify if the intermediate caching of translation table walks does not include non-coherent caches of previous valid translation table entries since the last completed TLBI applicable to the PE.

This feature is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.0 implementations.

The following fields identify the presence of FEAT_nTLBPA:
- ID_AA64MMFR1_EL1.nTLBPA.
- ID_MMFR5_EL1.nTLBPA.
- ID_MMFR5.nTLBPA.

For more information, see:
- TLB maintenance on page D8-5197.
- General TLB maintenance requirements on page G5-9220.

**FEAT_PCSRv8, PC Sample-based Profiling Extension**

FEAT_PCSRv8 adds support for PC Sample-based Profiling Extension that provides coarse-grained, non-invasive profiling by an external debugger.

This feature is OPTIONAL in Armv8.0 implementations.

The following fields identify the presence of FEAT_PCSRv8:
- EDDEVID.PCSample.
- DBGDEVID.PCSample.
- EDDEVID1.PCSROffset.
- DBGDEVID1.PCSROffset.
- PMDEVID.PCSample.

For more information, see About the PC Sample-based Profiling Extension on page H7-10352.

**A2.2.2 Architectural requirements within Armv8.0 architecture**

The Armv8.0 architecture includes some mandatory changes, that have been added to the architecture at a later date, that are not associated with a feature. These are:

**Prefetch speculation protection**

When substituting a direct branch with another direct branch, or a NOP with a direct branch, by the modified PE, at around the time that the executing PE is executing the software being modified, prefetch speculation protection prevents the old instructions from accidentally being fetched to the executing PE. For further information on implementation of these requirements, see:

- Ordering of instruction fetches on page B2-170.
- Ordering of instruction fetches on page E2-7178.

An implementation of the Armv8.0 architecture must comply with all of the additional requirements. When combined with the mandatory architectural features that have been added to the Armv8.0 architecture, such an implementation is also called an implementation of the Armv8.0 architecture.
A2.3 The Armv8 Cryptographic Extension

The Armv8.0 Cryptographic Extension provides instructions for the acceleration of encryption and decryption, and includes the following features:

- **FEAT_AES**, which includes the **AESD** and **AESE** instructions.
- **FEAT_PMULL**, which includes the **PMULL**, **PMULL2** instructions.
- **FEAT_SHA1**, which includes the **SHA1+** instructions.
- **FEAT_SHA256**, which includes the **SHA256+** instructions.

From Armv8.2, an implementation of the Armv8.0 Cryptographic Extension can include either or both of:

- The AES functionality, including support for multiplication of 64-bit polynomials. The **ID_AA64ISAR0_EL1.AES** field indicates whether this functionality is supported.
- The SHA1 and SHA2-256 functionality. The **ID_AA64ISAR0_EL1.{SHA2, SHA1}** fields indicate whether this functionality is supported.

The presence of the Cryptographic Extension in an implementation is subject to export license controls. The Cryptographic Extension is an extension of the SIMD support and operates on the vector register file.

The Cryptographic Extension also provides multiply instructions that operate on long polynomials.

The Cryptographic Extension provides this functionality in AArch64 state and AArch32 state, and an implementation that supports both AArch64 state and AArch32 state provides the same Cryptographic Extension functionality in both states.

For more information, see *The Cryptographic Extension on page C3-334 or The Cryptographic Extension in AArch32 state on page F2-7290.*

A2.3.1 Armv8.2 extensions to the Cryptographic Extension

Armv8.2 adds optional extensions to the Armv8 Cryptographic Extension, that provide cryptographic functionality in AArch64 state only. These optional features are:

**FEAT_SHA512, Advanced SIMD SHA512 instructions**

FEAT_SHA512 adds Advanced SIMD instructions that support SHA2-512 functionality.

These instructions are added to the A64 instruction set only.

Implementation of FEAT_SHA512 requires implementation of the Armv8.0 Cryptographic Extension FEAT_SHA1 and FEAT_SHA256 functionality.

The **ID_AA64ISAR0_EL1.SHA2** field identifies the presence of FEAT_SHA512.

For more information, see *FEAT_SHA512, SHA2-512 functionality on page C3-335.*

**FEAT_SHA3, Advanced SIMD SHA3 instructions**

FEAT_SHA3 adds Advanced SIMD instructions that support SHA3 functionality.

These instructions are added to the A64 instruction set only.

Implementation of FEAT_SHA3 requires implementation of the Armv8.0 Cryptographic Extension FEAT_SHA1 and FEAT_SHA256 functionality.

The **ID_AA64ISAR0_EL1.SHA3** field identifies the presence of FEAT_SHA3.

For more information, see *FEAT_SHA3, SHA3 functionality on page C3-336.*

**FEAT_SM3, Advanced SIMD SM3 instructions**

FEAT_SM3 adds Advanced SIMD instructions that support the Chinese cryptography algorithm SM3.

These instructions are added to the A64 instruction set only.

Implementation of FEAT_SM3 is independent of the implementation of any SHA functionality.

The **ID_AA64ISAR0_EL1.SM3** field identifies the presence of FEAT_SM3.
For more information, see *FEAT_SM3, SM3 functionality on page C3-337.*

**FEAT_SM4, Advanced SIMD SM4 instructions**

FEAT_SM4 adds Advanced SIMD instructions that support the Chinese cryptography algorithm SM4.

Implementation of FEAT_SM4 is independent of the implementation of any SHA functionality. These instructions are added to the A64 instruction set only.

The ID_AA64ISAR0_EL1.SM4 field identifies the presence of FEAT_SM4.

For more information, see *FEAT_SM4, SM4 functionality on page C3-337.*
A2.4 The Armv8.1 architecture extension

The Armv8.1 architecture extension adds both architectural features and architectural requirements, see:

- Architectural features added by Armv8.1.
- Additional requirements of Armv8.1 on page A2-84.
- Features added to the Armv8.1 extension in later releases on page A2-85.
- Features made optional in Armv8.1 implementations on page A2-85.

A2.4.1 Architectural features added by Armv8.1

An implementation of the Armv8.1 extension must include all of the features that this section describes as mandatory. Such an implementation, when combined with the additional requirements of Armv8.1, is also called an implementation of the Armv8.1 architecture.

The Armv8.1 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

**FEAT_LSE, Large System Extensions**

FEAT_LSE introduces a set of atomic instructions:

- Compare and Swap instructions, CAS and CASP.
- Atomic memory operation instructions, LD<OP> and ST<OP>, where <OP> is one of ADD, CLR, EOR, SET, SMAX, SMIN, UMAX, and UMIN.
- Swap instruction, SWP.

These instructions are added only to the A64 instruction set.

This feature is mandatory in Armv8.1 implementations.

Implementations of FEAT_VHE require the implementation of FEAT_LSE.

The ID_AA64ISAR0_EL1.Atomic field identifies the presence of FEAT_LSE.

For more information, see:

- Atomic memory operations on page C3-268.
- Swap on page C3-272.
- Compare and Swap on page C3-272.

**FEAT_RDM, Advanced SIMD rounding double multiply accumulate instructions**

FEAT_RDM introduces Rounding Double Multiply Add/Subtract Advanced SIMD instructions.

For more information, see:

For the A64 instruction set

- SQRDMLAH (by element) on page C7-2698.
- SQRDMLAH (vector) on page C7-2701.
- SQRDMLSH (by element) on page C7-2704.
- SQRDMLSH (vector) on page C7-2707.

For the T32 and A32 instruction sets

- VQRDMLAH on page F6-8658.
- VQRDMLSH on page F6-8662.

This feature is mandatory in Armv8.1 implementations.

The following fields identify the presence of FEAT_RDM:

- ID_AA64ISAR0_EL1.RDM.
- ID_ISAR5_EL1.RDM.
- ID_ISAR5.RDM.
FEAT_LOR, Limited ordering regions
Limited ordering regions allow large systems to perform special Load-Acquire and Store-Release instructions that provide order between the memory accesses to a region of the PA map as observed by a limited set of observers.
This feature is supported in AArch64 state only.
This feature is mandatory in Armv8.1 implementations.
The ID_AA64MMFR1_EL1.LO field identifies the presence of FEAT_LOR.
For more information, see:
• Limited ordering regions on page B2-182.

FEAT_HPDS, Hierarchical permission disable
FEAT_HPDS introduces the facility to disable the hierarchical attributes, APTable, PXNTable, and UXNTable, in the translation tables. This disable has no effect on the NSTable bit.
This feature is mandatory in Armv8.1 implementations.
This feature is added only to the VMSAv8-64 translation regimes. Armv8.2 extends this to the AArch32 translation regimes, see FEAT_AA32HPD.
The ID_AA64MMFR1_EL1.HPDS field identifies the presence of FEAT_HPDS.

FEAT_HAFDBS, Hardware management of the Access flag and dirty state
In Armv8.0, all updates to the translation tables are performed by software. From Armv8.1, for the VMSAv8-64 translation regimes only, hardware can perform updates to the translation tables in two contexts:
• Hardware management of the Access flag.
• Hardware management of dirty state, with updates to a dirty state in the translation tables.
The dirty state is introduced in Armv8.1.
Hardware management of dirty state can be enabled only when hardware management of the Access flag is also enabled.
This feature is OPTIONAL in Armv8.1 implementations. It is IMPLEMENTATION DEFINED whether this is implemented.
The ID_AA64MMFR1_EL1.HAFDBS field identifies the presence of FEAT_HAFDBS.
For more information, see:
• Hardware management of the Access flag on page D8-5144.
• Hardware management of the dirty state on page D8-5145.

FEAT_PAN, Privileged access never
FEAT_PAN adds a bit to PSTATE. When the value of this PAN state bit is 1, any privileged data access from EL1, or EL2 when HCR_EL2.E2H is 1, to a virtual memory address that is accessible to data accesses at EL0, generates a Permission fault.
This feature is mandatory in Armv8.1 implementations.
This feature is supported in both AArch64 and AArch32 states.
The following fields identify the presence of FEAT_PAN:
• ID_AA64MMFR1_EL1.PAN.
• ID_MMFR3_EL1.PAN.
• ID_MMFR3.PAN.
For more information, see:
• PSTATE.PAN on page D8-5132.
• About the PAN bit on page G5-9195.

FEAT_VMID16, 16-bit VMID
In an Armv8.1 implementation, when EL2 is using AArch64, the virtual machine identifier (VMID) size is an IMPLEMENTATION DEFINED choice of 8 bits or 16 bits.
This feature is OPTIONAL in Armv8.1 implementations. When implemented, this feature is supported only when EL2 is using AArch64. The ID_AA64MMFR1_EL1.VMIDBits field identifies the supported VMID size. For more information, see:

- VMID size on page D8-5196.

**FEAT_VHE, Virtualization Host Extensions**

Armv8.1 introduces the Virtualization Host Extensions (VHE) that provide enhanced support for Type 2 hypervisors in Non-secure state. This feature is mandatory in Armv8.1 implementations. An implementation that includes FEAT_VHE requires FEAT_LSE to be implemented. The ID_AA64MMFR1_EL1.VH field identifies the presence of FEAT_VHE. The following fields indicate the presence of the Virtualization Host Extensions for debug, including the changes for the PC Sample-based Profiling Extension and the Performance Monitors Extension:

- ID_AA64DFR0_EL1.DebugVer.
- ID_DFR0_EL1.{CopSDbg, CopDbg}.

For more information, see:

- Virtualization Host Extensions on page D8-5168.

**FEAT_PMUv3p1, PMU Extensions v3.1**

Armv8.1 makes the following enhancements to the Performance Monitors Extension:

- The event number space is extended to 16 bits to allow additional IMPLEMENTATION DEFINED event types, and the reserved space for future additions to the architecturally-defined event types is extended.
- The HPMD bit is added to MDCR_EL2. This bit disables event counting at EL2.
- The STALL_FRONTEND and STALL_BACKEND events are required to be implemented. For more information, see Required events on page D11-5425.

The Performance Monitors Extension is an optional feature, but if it is implemented, an Armv8.1 implementation must include FEAT_PMUv3p1. The following fields identify the presence of FEAT_PMUv3p1:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.

**A2.4.2 Additional requirements of Armv8.1**

The Armv8.1 architecture includes some mandatory changes that are not associated with a feature. These are:

**FEAT_CRC32, Changes to CRC32 instructions**

All implementations of the Armv8.1 architecture are required to implement the CRC32* instructions. These are OPTIONAL in Armv8.0. The following fields identify the presence of the CRC32* instructions:

- ID_AA64ISAR0_EL1.CRC32.
- ID_ISAR5_EL1.CRC32.
- ID_ISAR5.CRC32.

An implementation of the Armv8.1 extension must comply with all of the additional requirements. Such an implementation, when combined with the mandatory architectural features of Armv8.1, is also called an implementation of the Armv8.1 architecture.
A2.4.3 Features added to the Armv8.1 extension in later releases

FEAT_PAN3, Support for SCTLR_ELx.EPAN

FEAT_PAN3 adds a bit to SCTLR_EL1 and SCTLR_EL2, EPAN, to support using Privileged Access Never with instruction accesses for stage 1 translation regimes. This feature is supported in AArch64 state only. This feature is OPTIONAL in Armv8.1 implementations and mandatory in Armv8.7 implementations. The ID_AA64MMFR1_EL1.PAN field identifies the presence of FEAT_PAN3. For more information, see PSTATE.PAN on page D8-5132.

A2.4.4 Features made OPTIONAL in Armv8.1 implementations

The feature that has been made OPTIONAL in Armv8.1 implementations is FEAT_PAN2 on page A2-86.
A2.5 The Armv8.2 architecture extension

The Armv8.2 architecture extension adds both architectural features and architectural requirements, see:

- Architectural features added by Armv8.2.
- Additional requirements of Armv8.2 on page A2-93.
- Features added to the Armv8.2 extension in later releases on page A2-93.
- Features made optional in Armv8.2 implementations on page A2-95.

The Armv8.2 architecture extension also adds functionality to the Cryptographic Extension, see Armv8.2 extensions to the Cryptographic Extension on page A2-80.

A2.5.1 Architectural features added by Armv8.2

An implementation of the Armv8.2 extension must include all of the features that this section describes as mandatory. Such an implementation, when combined with the additional requirements of Armv8.2, is also called an implementation of the Armv8.2 architecture.

The Armv8.2 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

**FEAT_ASMv8p2, Armv8.2 changes to the A64 ISA**

FEAT_ASMv8p2 adds the BFC instruction to the A64 instruction set as an alias of BFM. It also requires that the BFC instruction and the A64 pseudo-instruction REV64 are implemented by assemblers.

--- Note ---

- In Armv8.0 and Armv8.1, the A64 pseudo-instruction REV64 is OPTIONAL.
- Because this feature relates to support for an instruction alias and for a pseudo-instruction, there are no corresponding feature ID register fields.

This change to the instruction set and assembler requirements is mandatory in an Armv8.2 implementation.

For more information, see:

- **BFC** on page C6-1217.
- **REV64** on page C6-1762.

**FEAT_PAN2, AT S1E1R and AT S1E1W instruction variants affected by PSTATE.PAN**

FEAT_PAN2 adds variants of the AArch64 AT S1E1R and AT S1E1W instructions and the AArch32 ATS1CPWP instructions. These instructions factor in the PSTATE.PAN bit when determining whether or not the location will generate a Permission fault for a privileged access, as is reported in the PAR. For more information, see:

For the AArch64 System instructions

- **AT S1E1RP, Address Translate Stage 1 EL1 Read PAN** on page C5-856.
- **AT S1E1WP, Address Translate Stage 1 EL1 Write PAN** on page C5-860.

For the AArch32 System instructions

- **ATS1CPRP, Address Translate Stage 1 Current state PL1 Read PAN** on page G8-9360.
- **ATS1CPWP, Address Translate Stage 1 Current state PL1 Write PAN** on page G8-9362.

This feature is OPTIONAL in Armv8.1 implementations and mandatory in Armv8.2 implementations. These instructions are added to the A64 and A32/T32 instruction sets.

The following fields identify the presence of FEAT_PAN2:

- **ID_AA64MMFR1_EL1.PAN**.
- **ID_MMFR3_EL1.PAN**.
- **ID_MMFR3.PAN**.
Feat_fp16, Half-precision floating-point data processing

Feat_fp16 supports:
- Half-precision data-processing instructions for Advanced SIMD and floating-point in both AArch64 and AArch32 states.
- The FPSCR.FZ16 and FPSCR.FZ16 bits, which enables flushing of denormalized numbers to zero for half-precision data-processing instructions.

This feature is optional in Armv8.2 implementations, unless one of the following is implemented:
- The Scalable Vector Extension (SVE).
- Feat_fhm.

If SVE or Feat_fhm is implemented, Feat_fp16 is implemented. From Armv8.4, if Feat_fhm is not implemented, Feat_fp16 is not implemented.

When this feature is implemented it is implemented in both Advanced SIMD and floating-point, and in AArch64 and AArch32 states.

The following fields identify the presence of Feat_fp16:
- ID_AA64PFR0_EL1.{FP, AdvSIMD}.
- MVFR1_EL1.{FHPH, SIMDHP}.
- MVFR1.{FHPH, SIMDHP}.

For more information, see:
- Flushing denormalized numbers to zero on page A1-58.
- Modified immediate constants in A64 instructions on page C2-244.

Feat_dotprod, Advanced SIMD dot product instructions

Feat_DOTProd provides instructions to perform the dot product of two 32-bit vectors, accumulating the result in a third 32-bit vector. This can be performed using signed or unsigned arithmetic.

This feature is optional in Armv8.2 implementations and mandatory in Armv8.4 implementations. These instructions are added to the A64 and A32/T32 instruction sets.

The following fields identify the presence of Feat_DOTProd:
- ID_AA64ISAR0_EL1.DP.
- ID_ISAR6_EL1.DP.
- ID_ISAR6.DP.

For more information, see:
- SIMD integer dot product on page C3-331.
- Advanced SIMD dot product instructions on page F2-7287.

Feat_fhm, Floating-point half-precision multiplication instructions

Feat_fhm adds floating-point multiplication instructions.

These instructions are added to the A64 and A32/T32 instruction sets.

This feature is optional in Armv8.2 implementations, and can be implemented only when Feat_fp16 is implemented. This feature is mandatory in Armv8.4 implementations when Feat_fp16 is implemented. This feature is not implemented in Armv8.4 implementations when Feat_fp16 is not implemented.

The following fields identify the presence of Feat_fhm:
- ID_AA64ISAR0_EL1.FHM.
- ID_ISAR6_EL1.FHM.
• ID_ISAR6.FHM.
For more information, see:
• SIMD arithmetic on page C3-318.
• SIMD by element arithmetic on page C3-325.
• Advanced SIMD multiply instructions on page F2-7286.

FEAT_LSMAOC, AArch32 Load/Store Multiple instruction atomicity and ordering controls
FEAT_LSMAOC adds controls that disable legacy behavior of AArch32 load multiple and store multiple instructions, and provide a trap of one aspect of this legacy behavior.
Implementation of FEAT_LSMAOC is OPTIONAL. When implemented it provides:
• LSMAOE fields in the SCTLR_EL1, SCTLR_EL2, HSCTLR, and SCTLR registers. These fields can have the following effects on the behavior of AArch32 load multiple and store multiple instructions:
  — An interrupt can be taken between two memory accesses made by a single load multiple or store multiple instruction.
  — The memory accesses made by a single load multiple or store multiple instruction to Device memory with the non-Reordering attribute can be reordered.
• nTLSMD fields in the SCTLR_EL1, SCTLR_EL2, HSCTLR, and SCTLR registers. These fields can cause an access to Device-nGRE, Device-nGnRE, or Device-nGnRnE memory by an AArch32 load multiple and store multiple instruction to generate an Alignment fault.

Note
Armv8.2 deprecates software dependence on the legacy behavior of AArch32 load multiple and store multiple instructions, and these fields disable this behavior.

The following fields identify the presence of FEAT_LSMAOC:
• ID_AA64MMFR2_EL1.LSM.
• ID_MMFR4_EL1.LSM.
• ID_MMFR4.LSM.
For more information, see the register field descriptions and:
• Generation of Alignment faults by load/store multiple accesses to Device memory on page E2-7194.
• Multi-register loads and stores that access Device memory on page E2-7206.
• Taking an interrupt or other exception during a multiple-register load or store on page G1-8963.

FEAT_UAO, Unprivileged Access Override control
Armv8.2 adds a bit to PSTATE. When the value of PSTATE.UAO is 1, and when executed at EL1 or at EL2 with HCR_EL2.{E2H, TGE} == {1, 1}, the memory accesses made by the load/store unprivileged instructions behave as if they were made by the load/store register instructions. See Load/store unprivileged on page C3-260 and Load/store register on page C3-256.
This feature is mandatory in Armv8.2 implementations.
This feature is supported in AArch64 state only.
The ID_AA64MMFR2_EL1.UAO field identifies the presence of FEAT_UAO.
For more information, see PSTATE_UAO on page D8-5133.

FEAT_DPB, DC CVAP instruction
FEAT_DPB introduces a mechanism to identify and manage persistent memory locations in a shared memory hierarchy, including adding the DC CVAP instruction.
This feature is mandatory in Armv8.2 implementations.
This feature is supported in AArch64 state only.
The ID_AA64ISAR1_EL1.DPB field identifies the presence of FEAT_DPB.
For more information about FEAT_DPB, see Memory hierarchy on page B2-184.

**FEAT_VPIPT, VMID-aware PIPT instruction cache**

FEAT_VPIPT supports a instruction cache type, described as the VMID-aware PIPT (VPIPT) instruction cache.

--- Note ---

Armv8.2 adds VPIPT to the set of supported cache types, meaning an Armv8.2 implementation is permitted to implement VPIPT caches, but is not required to do so.

---

This feature is supported in both AArch64 and AArch32 states.

The CTR_EL0.L1Ip and CTR.L1Ip fields identify the presence of FEAT_VPIPT.

For more information, see:
- VMID-aware PIPT instruction caches on page D8-5216.
- VPIPT (VMID-aware PIPT) instruction caches on page G5-9236.

**FEAT_AA32HPD, AArch32 hierarchical permission disables**

FEAT_HPDS introduced the ability to disable the hierarchical attributes, APTable, PXNTable, and UXNTable, in the VMSA8v8-64 translation regimes. FEAT_AA32HPD extends this functionality to the VMSA8v8-32 translation regimes when those regimes are using the Long descriptor Translation Table format.

This feature is OPTIONAL in Armv8.2 implementations. It is IMPLEMENTATION DEFINED whether this is implemented.

The ID_MMFR4_EL1.HPDS and ID_MMFR4.HPDS fields identify the presence of FEAT_AA32HPD.

For more information, see Attribute fields in VMSA8-32 Long-descriptor translation table format descriptors on page G5-9176.

**FEAT_HPDS2, Translation table page-based hardware attributes**

Armv8.2 provides a mechanism to allow operating systems or hypervisors to make up to four bits of Translation Table final-level descriptors available for IMPLEMENTATION DEFINED hardware use.

This functionality is available for all translation regimes in AArch64 state and for stages of translation in AArch32 state that use the Long descriptor Translation Table format.

FEAT_HPDS2 is OPTIONAL in Armv8.2 implementations, but implementation of FEAT_HPDS2 requires implementation of both:
- FEAT_HPDS.
- FEAT_AA32HPD, if any Exception level higher than EL0 can use AArch32.

--- Note ---

For stage 1 translations, page-based hardware attributes can be used only for a stage of translation for which the Hierarchical permission disables field has a value of 1.

---

The following fields identify the presence of FEAT_HPDS2:
- ID_AA64MMFR1_EL1.HPDS.
- ID_MMFR4_EL1.HPDS.
- ID_MMFR4.HPDS.

For more information, see:
- Page Based Hardware attributes on page D8-5159.
- Attribute fields in VMSA8-32 Long-descriptor translation table format descriptors on page G5-9176.
FEAT_LPA, Large PA and IPA support

FEAT_LPA:

- Allows a larger intermediate physical address (IPA) and PA space of up to 52 bits when using the 64KB translation granule.
- Allows a level 1 block size where the block covers a 4TB address range for the 64KB translation granule if the implementation support 52 bits of PA.

This is an OPTIONAL feature in Armv8.2 implementations. It is IMPLEMENTATION DEFINED whether it is implemented.

This feature is supported in AArch64 state only.

The ID_AA64MMFR0_EL1.PARange field identifies the presence of FEAT_LPA.

For more information about FEAT_LPA, see:
- Implemented physical address size on page D8-5088.
- Output address size configuration on page D8-5088.
- Intermediate physical address size configuration on page D8-5091.
- Translation using the 64KB granule on page D8-5112.
- Table descriptor format on page D8-5123.
- Block descriptor and Page descriptor formats on page D8-5126.

FEAT_LVA, Large VA support

FEAT_LVA supports a larger VA space for each translation table base register of up to 52 bits when using the 64KB translation granule.

This feature is supported in AArch64 state only.

This is an OPTIONAL feature in Armv8.2 implementations. It is IMPLEMENTATION DEFINED whether it is implemented.

If FEAT_LVA is implemented, then any implemented trace macrocell must be at least ETMv4.2 for Armv8-A, and FEAT_ETE for Armv9-A.

The ID_AA64MMFR2_EL1.VARange field identifies the presence of FEAT_LVA.

For more information about FEAT_LVA, see:
- Supported virtual address ranges on page D8-5089.
- Input address size configuration on page D8-5090.
- Translation using the 64KB granule on page D8-5112.
- Table descriptor format on page D8-5123.
- Block descriptor and Page descriptor formats on page D8-5126.

FEAT_TTCNP, Translation table Common not private translations

FEAT_TTCNP permits multiple PEs in the same Inner Shareable domain to use the same translation tables for a given stage of address translation.

This feature is mandatory in Armv8.2 implementations.

This facility is available for all VMSAv8-64 translation regimes and for VMSAv8-32 translation stages that use the Long descriptor Translation Table format.

The following fields identify the presence of FEAT_TTCNP:

- ID_AA64MMFR2_EL1.CnP.
- ID_MIMFR4_EL1.CnP.
- ID_MIMFR4.CnP.

For more information, see:
- Common not private translations on page D8-5196.
- Common not private translations in VMSAv8-32 on page G5-9225.
FEAT_XNX, Translation table stage 2 Unprivileged Execute-never

FEAT_XNX extends the stage 2 translation table access permissions to provide control of whether memory is executable at EL0 independent of whether it is executable at EL1.

This feature is mandatory in Armv8.2 implementations that implement EL2.

This facility is available for stage 2 translation stages in VMSAv8-64 and VMSAv8-32.

The following fields identify the presence of FEAT_XNX:

- ID_AA64MMFR1_EL1.XNX.
- ID_MMFR4_EL1.XNX.
- ID_MMFR4.XNX.

For more information, see:

- Instruction execution permissions on page D8-5138.
- Access permissions for instruction execution on page G5-9196.

FEAT_Debugv8p2, Debug v8.2

FEAT_Debugv8p2 covers a selection of mandatory changes, including:

- If the Core power domain is powered up and DoubleLockStatus() == TRUE, EDPRSR.{DLK,SPD,PU} is only permitted to read {UNKNOWN, 0, 0}.
- The definition of Exception Catch debug events is extended to include reset entry.
- All CONSTRAINED UNPREDICTABLE cases that generate Exception Catch debug events are removed.
- Controls are added to EDECCR to control Exception Catch debug event generation on exception return.
- All IMPLEMENTATION DEFINED control of external debug accesses to OSLAR_EL1 is removed.
- ExternalSecureNoninvasiveDebugEnabled() cannot override software controls of counting attributable events in Secure state.

If FEAT_Debugv8p2 is implemented, FEAT_DoubleLock is OPTIONAL.

The fields that identify the presence of FEAT_Debugv8p2 are:

- ID_AA64DFR0_EL1.DebugVer and DBGDIDR.Version.
- ID_DFR0_EL1.{CopSDbg, CopDbg} and ID_DFR0.{CopSDbg, CopDbg}.
- EDDEVARCH.ARCHID.

For more information, see:

- Exception Catch debug event on page H3-10287.
- EDPRSR.{DLK, SPD, PU} and the Core power domain on page H6-10342.
- Interaction with EL3 on page D11-5245.
- External access disabled on page H8-10364.

FEAT_PCSRv8p2, PC Sample-based profiling

In Armv8.2, the control and implementation of the OPTIONAL PC Sample-based Profiling extension is moved from ED*SR Debug registers to PM*SR registers in the Performance Monitors address space. See Chapter H7 The PC Sample-based Profiling Extension.

The PC Sample-based Profiling Extension is an OPTIONAL feature. If it is implemented, an Arm8.2 implementation must also include FEAT_PCSRv8p2.

If Secure EL2 and PC Sample-based Profiling are both implemented, FEAT_PCSRv8p2 is mandatory.

The following fields identify the presence of FEAT_PCSRv8p2:

- EDDEVID.PCSample.
- DBGDEVID.PCSample.
- EDDEVID1.PCSROffset.
- DBGDEVID1.PCSROffset.
FEAT_IESB, Implicit Error Synchronization event

FEAT_IESB adds an implicit error synchronization event at exception entry and return, controlled by the added SCTLR_ELx.IESB fields. An IESB field is added to the ESR_ELx syndrome registers. The implicit error synchronization events affect the same synchronizable asynchronous events that are synchronized by the ESB instruction, see The Reliability, Availability, and Serviceability Extension on page A2-122.

This feature is OPTIONAL in Armv8.2 implementations.

This feature is supported in AArch64 state only.

The ID_AA64MMFR2_EL1.IESB field identifies the presence of FEAT_IESB.

For more information, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile.

FEAT_F32MM, Single-precision Matrix Multiplication

FEAT_F32MM adds support for the SVE FP32 single-precision floating-point matrix multiplication variant of the FMMLA instruction.

This feature is OPTIONAL in Armv8.2 implementations.

This feature is supported in AArch64 state only.

FEAT_F32MM requires FEAT_SVE.

The ID_AA64ZFR0_EL1.F32MM field identifies the presence of FEAT_F32MM.

FEAT_F64MM, Single-precision Matrix Multiplication

FEAT_F64MM adds support for the following SVE instructions:

- FMMLA (FP64 double-precision variant).
- LD1ROB (scalar plus immediate).
- LD1ROB (scalar plus scalar).
- LD1ROD (scalar plus immediate).
- LD1ROD (scalar plus scalar).
- LD1ROH (scalar plus immediate).
- LD1ROH (scalar plus scalar).
- LD1ROW (scalar plus immediate).
- LD1ROW (scalar plus scalar).
- TRN1, TRN2 (vectors) (128-bit variant).
- UZP1, UZP2 (vectors) (128-bit variant).
- ZIP1, ZIP2 (vectors) (128-bit variant).

This feature is OPTIONAL in Armv8.2 implementations.

This feature is supported in AArch64 state only.

FEAT_F64MM requires FEAT_SVE.

The ID_AA64ZFR0_EL1.F64MM field identifies the presence of FEAT_F64MM.

Extensions to the Arm Cryptographic Extensions

See the description of the FEAT_SHA512 and FEAT_SM3 features in Armv8.2 extensions to the Cryptographic Extension on page A2-80.
A2.5.2 Additional requirements of Armv8.2

The Armv8.2 architecture includes some mandatory changes that are not associated with a feature. These are:

**Change to ACTLR2 and HCTLR2 registers**

In AArch32 state, the ACTLR2 and HACTLR2 registers become mandatory.

**Implementation of RAS Extension**

The RAS Extension must be implemented, see *The Reliability, Availability, and Serviceability Extension* on page A2-122.

An implementation of the Armv8.2 extension must comply with all of the additional requirements. Such an implementation, when combined with the mandatory architectural features of Armv8.2, is also called an implementation of the Armv8.2 architecture.

If FEAT_PMUv3 is implemented, the feature FEAT_PMUv3p4 is optional in Armv8.2 implementations.

A2.5.3 Features added to the Armv8.2 extension in later releases

**FEAT_EVT, Enhanced Virtualization Traps**

FEAT_EVT introduces additional traps for EL1 and EL0 Cache controls. These traps are independent of existing controls.

This feature is supported in both AArch64 and AArch32 states.

This feature is optional in Armv8.2 implementations and is mandatory in Armv8.5.

ID_AA64MMFR2_EL1.EVT identifies the presence of the AArch64 traps controls.

ID_MMFR4_EL1.EVT and ID_MMFR4.EVT identify the presence of the AArch32 traps.

For more information, see:

• HCR_EL2.{TTLBIS, TTLBOS, TICAB, TOCU, TID4}.
• HCR2.{TTLBIS, TICAB, TOCU, TID4}.

**FEAT_DPB2, DC CVADP instruction**

FEAT_DPB2 allows two levels of cache clean to the Point of Persistence by:

• Redefining Point of Persistence, which changes the scope of DC CVAP.
• Defining a Point of Deep Persistence.
• Adding the DC CVADP System instruction.

This feature is supported in AArch64 state only.

This feature is optional in Armv8.2 implementations and is mandatory in Armv8.5 implementations.

The ID_AA64ISAR1_EL1.DPB field identifies the presence of FEAT_DPB2.

For further information, see *Terminology for Clean, Invalidate, and Clean and Invalidate instructions* on page D7-5051.

**FEAT_BF16, AArch64 BFloat16 instructions**

FEAT_BF16 supports the BFloat16, or BF16, 16-bit floating-point storage format in AArch64 state. This format supports:

• The BFloat16 floating-point data type.
• Arithmetic instructions to accelerate dot products and matrix multiplications of BF16 values.
• Instructions to convert single-precision floating-point values to BF16 format.
• Arithmetic instructions to multiply BFloat16 values and accumulate into single-precision results.

This feature is supported in AArch64 state only.

This feature is optional in Armv8.2 implementations and mandatory in Armv8.6 implementations.

The ID_AA64ISAR1_EL1.BF16 field identifies the presence of FEAT_BF16.
When both Advanced SIMD and SVE are implemented, the ID_AA64ISAR1_EL1.BF16 and ID_AA64ZFR0_EL1.BF16 fields must return the same value.

For further information, see:
- Floating-point single-precision to BFloat16 conversion instruction on page C3-313.
- SIMD BFloat16 on page C3-332.
- SVE BFloat16 floating-point multiply-add on page C3-353.
- SVE BFloat16 floating-point dot product on page C3-353.
- SVE BFloat16 floating-point matrix multiply on page C3-354.
- SVE BFloat16 floating-point convert on page C3-355.

FEAT_AA32BF16, AArch32 BFloat16 instructions
FEAT_AA32BF16 supports the BFloat16, or BF16, 16-bit floating-point storage format in AArch32 state. This format supports:
- The BFloat16 floating-point data type.
- Arithmetic instructions to accelerate dot products and matrix multiplications of BF16 values.
- Instructions to convert single-precision floating-point values to BF16 format.

This feature is supported in AArch32 state only.

This feature is OPTIONAL in Armv8.2 implementations.

The ID_ISAR6_EL1.BF16 and ID_ISAR6.BF16 fields identify the presence of FEAT_AA32BF16.

For further information, see:
- Advanced SIMD BFloat16 instructions on page F2-7288.
- Floating-point data-processing on page F3-7329.

FEAT_I8MM, AArch64 Int8 matrix multiplication instructions
FEAT_I8MM introduces integer matrix multiply-accumulate instructions and mixed sign dot product instructions.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.2 implementations and mandatory in Armv8.6 implementations.

The ID_AA64ISAR1_EL1.I8MM field identifies the presence of FEAT_I8MM.

When both Advanced SIMD and SVE are implemented, the ID_AA64ISAR1_EL1.I8MM and the ID_AA64ZFR0_EL1.I8MM fields must return the same value.

For further information, see:
- SIMD integer dot product on page C3-331.
- SIMD integer matrix multiply-accumulate on page C3-334.
- SVE Integer dot product on page C3-342.
- SVE Integer matrix multiply operations on page C3-342.

FEAT_AA32I8MM, AArch32 Int8 matrix multiplication instructions
FEAT_AA32I8MM introduces integer matrix multiply-accumulate instructions and mixed sign dot product instructions.

This feature is supported in AArch32 state only.

This feature is OPTIONAL in Armv8.2 implementations.

The ID_ISAR6_EL1.I8MM and ID_ISAR6.I8MM fields identify the presence of FEAT_AA32I8MM.

For further information, see:
- Advanced SIMD dot product instructions on page F2-7287.
- Advanced SIMD matrix multiply instructions on page F2-7288.
A2.5.4 Features made OPTIONAL in Armv8.2 implementations

The features that have been made OPTIONAL in Armv8.2 implementations are:

- \textit{FEAT\_FlagM} on page A2-101.
- \textit{FEAT\_LSE2} on page A2-101.
- \textit{FEAT\_LRCPC2} on page A2-101.
A2.6 The Armv8.3 architecture extension

The Armv8.3 architecture extension adds both architectural features and additional requirements, see:

- Architectural features added by Armv8.3.
- Additional requirements of Armv8.3 on page A2-98.
- Features added to the Armv8.3 extension in later releases on page A2-98.

A2.6.1 Architectural features added by Armv8.3

An implementation of the Armv8.3 extension must include all of the features that this section describes as mandatory. Such an implementation is also called an implementation of the Armv8.3 architecture.

The Armv8.3 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

FEAT_FCMA, Floating-point complex number instructions

FEAT_FCMA introduces instructions for floating-point multiplication and addition of complex numbers.

These instructions are added to the A64 and A32/T32 instruction sets.

This feature is mandatory in Armv8.3 implementations.

The half-precision versions of these instructions are implemented only if FEAT_FP16 is implemented. Otherwise they are UNDEFINED.

The fields that identify the presence of FEAT_FCMA are:

- ID_AA64ISAR1_EL1.FCMA.
- ID_ISAR5_EL1.VCMA.
- ID_ISAR5.VCMA.

For more information, see:

- SIMD complex number arithmetic on page C3-332.
- Advanced SIMD complex number arithmetic instructions on page F2-7287.

FEAT_JSCVT, JavaScript conversion instructions

FEAT_JSCVT introduces instructions that perform a conversion from a double-precision floating point value to a signed 32-bit integer, with rounding to zero. For more information, see:

For the A64 instruction set

- FJCVTZS on page C7-2269.

For the A32/T32 instruction set

- VJCVT on page F6-8419.

These instructions are added to the A64 and A32/T32 instruction sets.

This feature is mandatory in Armv8.3 implementations.

The fields that identify the presence of FEAT_JSCVT are:

- ID_AA64ISAR1_EL1.JSCVT.
- ID_ISAR6_EL1.JSCVT.
- ID_ISAR6.JSCVT.

For more information, see:

- Floating-point conversion on page C3-313.
- About the A64 Advanced SIMD and floating-point instructions on page C7-2040.
- Advanced SIMD and floating-point instructions on page E1-7140.
- Floating-point data-processing instructions on page F2-7292.
FEAT_LRCPC, Load-Acquire RCpc instructions

FEAT_LRCPC introduces three instructions to support the weaker *Release Consistency processor consistent* (RCpc) model that enables the reordering of a Store-Release followed by a Load-Acquire to a different address:

- `LDAPR` on page C6-1520.
- `LDAPRB` on page C6-1522.
- `LDAPRH` on page C6-1524.

These instructions are added to the A64 instruction set.

The feature is mandatory in Armv8.3 implementations.

The ID_AA64ISAR1_EL1.LRCPC field identifies the presence of FEAT_LRCPC.

For more information, see:

FEAT_NV, Nested virtualization support

FEAT_NV provides support for a Guest Hypervisor to run in Non-secure EL1 and ensures that the Guest Hypervisor is unaware that it is running at that Exception level. A Guest Hypervisor is supported regardless of the value of HCR_EL2.E2H.

This feature is supported in AArch64 state only.

The feature is *optional* in Armv8.3 implementations. This feature must be implemented if FEAT_NV2 is implemented.

The ID_AA64MMFR2_EL1.NV field identifies the presence of FEAT_NV.

For more information, see *Nested virtualization* on page D8-5173.

FEAT_CCIDX, Extended cache index

FEAT_CCIDX introduces the following registers to allow caches to be described with greater numbers of sets and greater associativity:

- A 64-bit format of CCSIDR_EL1.
- CCSIDR2_EL1.
- CCSIDR2.

This feature is supported in both AArch64 and AArch32 states.

This feature is *optional* in Armv8.3 implementations.

The following fields identify the presence of FEAT_CCIDX:

- ID_AA64MMFR2_EL1.CCIDX.
- ID_MMFR4_EL1.CCIDX.
- ID_MMFR4.CCIDX.

For more information, see:
- *Possible formats of the Cache Size Identification Register, CCSIDR_EL1* on page D7-5045.
- *Possible formats of the Cache Size Identification Registers, CCSIDR and CCSIDR2* on page G4-9115.

FEAT_PAuth and FEAT_EPAC, Pointer authentication and Enhanced Pointer authentication

FEAT_PAuth adds functionality that supports address authentication of the contents of a register before that register is used as the target of an indirect branch, or as a load.

FEAT_EPAC adds functionality that permits setting the *Pointer Authentication Code* (PAC) field to 0 on performing a PAC operation on a non-canonical address.

These features are supported in AArch64 state only.

FEAT_PAuth is mandatory in Armv8.3 implementations.

FEAT_EPAC is *optional* in Armv8.3 implementations.
When FEAT_PAuth is implemented, one of the following must be true:

- Exactly one of the PAC algorithms is implemented.
- If the PACGA instruction and other Pointer authentication instructions use different PAC algorithms, exactly two PAC algorithms are implemented.

The PAC algorithm features are:

- FEAT_PACQARMA5.
- FEAT_PACIMP.
- FEAT_PACQARMA3.

The following fields identify the presence of FEAT_PAuth:

- ID_AA64ISAR1_EL1.{GPI, GPA, API, APA}.
- ID_AA64ISAR2_EL1.{GPA3, APA3}.

The following fields identify the presence of FEAT_EPAC:

- ID_AA64ISAR1_EL1.{API, APA}.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Pointer authentication on page D8-5164.

**FEAT_PACQARMA5, Pointer authentication - QARMA5 algorithm**

FEAT_PACQARMA5 adds the QARMA5 cryptographic algorithm for PAC calculation.

This feature is supported in AArch64 state only.

This feature is optional in Armv8.3 implementations, and can be implemented only if FEAT_PAuth is implemented.

The ID_AA64ISAR1_EL1.{GPA, APA} fields identify the support for FEAT_PACQARMA5.

For more information, see Pointer authentication on page D8-5164.

**FEAT_PACIMP, Pointer authentication - IMPLEMENTATION DEFINED algorithm**

FEAT_PACIMP permits an IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

This feature is supported in AArch64 state only.

This feature is optional in Armv8.3 implementations, and can be implemented only if FEAT_PAuth is implemented.

The ID_AA64ISAR1_EL1.{GPI, API} fields identify the support for FEAT_PACIMP.

For more information, see Pointer authentication on page D8-5164.

### A2.6.2 Additional requirements of Armv8.3

If FEAT_PMUv3 is implemented, FEAT_PMUv3p4 is optional in Armv8.3 implementations.

### A2.6.3 Features added to the Armv8.3 extension in later releases

**FEAT_SPEv1p1, Armv8.3 Statistical Profiling Extensions**

FEAT_SPEv1p1 adds an Alignment Flag in the Events packet and filtering on this event using PMSEVFR_EL1, together with support for the profiling of Scalable Vector Extension operations.

This feature is supported in AArch64 state only.

This feature is optional in Armv8.3 implementations. An Armv8.5 implementation that includes the Statistical Profiling Extension must include FEAT_SPEv1p1.

The fields in ID_AA64DFR0_EL1.PMSVer identify the presence of FEAT_SPEv1p1.

For more information, see Chapter D13 The Statistical Profiling Extension and Chapter D14 Statistical Profiling Extension Sample Record Specification.
FEAT_DoPD, Debug over Powerdown

FEAT_DoPD provides a debug programmers’ model where all debug and PMU registers are in the Core power domain, all CTI registers are in the Debug power domain. Power control is provided by a CoreSight Granular Power Requester (GPR) component.

When the OPTIONAL powerup mechanism is implemented and this feature is implemented, the debugger makes power control requests for the Core power domain using a CoreSight Class 0x9 ROM Table block, instead of using EDRCR.COREPURQ. EDRCR.COREPURQ is not implemented. Refer to the ARM® CoreSight Architecture Specification for more information.

This feature is OPTIONAL in Armv8.3 implementations.

When FEAT_DoPD is implemented:

- FEAT_DoubleLock is not implemented.
- FEAT_Debugv8p2 must be implemented.
- If PC Sample-based profiling is implemented, FEAT_PCSRv8p2 must be implemented.
- The optional Software Lock is not implemented by the architecturally defined debug components in the PE Core power domain.
- If an ETMv4 trace unit is implemented, the ETM must implement:
  - ETMv4.2 or later.
  - The Unified Power Domain Model.
- If FEAT_ETE is implemented, the trace unit always implements a single power domain.

The fields that identify the presence of FEAT_DoPD are:

- EDDEVID.DebugPower.
- CTIDEVARCH.REVISION.

For more information, see Chapter H6 Debug Reset and Powerdown Support.

FEAT_PAuth2, Enhancements to pointer authentication

FEAT_PAuth2 adds enhanced pointer authentication functionality that changes the mechanism by which a PAC is added to the pointer.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.3 implementations and mandatory in Armv8.6 implementations.

The fields that identify the support for FEAT_PAuth2 are:

- ID_AA64ISAR1_EL1.{API, APA}.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Pointer authentication on page D8-5164.

FEAT_FPAC and FEAT_FPACCOMBINE, Faulting on AUT* instructions and combined pointer authentication instructions

FEAT_FPAC introduces faulting on an AUT* instruction.

FEAT_FPACCOMBINE introduces faulting on the combined instructions that perform pointer authentication.

FEAT_FPAC is added as a further extension to FEAT_PAuth2.

FEAT_FPACCOMBINE is added as a further extension to FEAT_FPAC.

These features are supported in AArch64 state only.

FEAT_FPAC is OPTIONAL in Armv8.3 implementations, and can be implemented only if FEAT_PAuth2 is implemented.

FEAT_FPACCOMBINE is OPTIONAL in Armv8.3 implementations, and can be implemented only if FEAT_FPAC is implemented.

The fields that identify the support for FEAT_FPAC and FEAT_FPACCOMBINE are:

- ID_AA64ISAR1_EL1.{API, APA}.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Faulting on pointer authentication on page D8-5167.
FEAT_PACQARMA3, Pointer authentication - QARMA3 algorithm

FEAT_PACQARMA3 adds the QARMA3 cryptographic algorithm for PAC calculation.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.3 implementations, and can be implemented only if FEAT_PAuth is implemented.

The ID_AA64ISAR2_EL1.{GPA3, APA3} fields identify the support for FEAT_PACQARMA3.

For more information, see Pointer authentication on page D8-5164.

FEAT_CONSTPACFIELD, PAC algorithm enhancement

FEAT_CONSTPACFIELD introduces functionality that permits an implementation with pointer authentication to use the value of bit[55] in the virtual address to determine the size of the PAC field, even when the top byte is not being ignored.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in Armv8.3 implementations, and can be implemented only if FEAT_PAuth2 is implemented.

The ID_AA64ISAR2_EL1.PAC_frac field identifies the support for FEAT_CONSTPACFIELD.

For more information, see PAC field on page D8-5164.
A2.7 The Armv8.4 architecture extension

The Armv8.4 architecture extension adds architectural features, see Architectural features added by Armv8.4. It also adds features to earlier architecture extensions, see Features added to earlier extensions on page A2-106.

A2.7.1 Architectural features added by Armv8.4

An implementation of the Armv8.4 extension must include all of the features that this section describes as mandatory. Such an implementation is also called an implementation of the Armv8.4 architecture.

The Armv8.4 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

FEAT_DIT, Data Independent Timing instructions
FEAT_DIT provides independent timing for data processing instructions with the addition of the PSTATE.DIT and CPSR.DIT fields.
This feature is supported in both AArch64 and AArch32 states.
This feature is mandatory in Armv8.4 implementations.
The following fields identify the presence of FEAT_DIT:
• ID_AA64PFR0_EL1.DIT.
• ID_PFR0_EL1.DIT.
• ID_PFR0.DIT.
For more information, see:
• About PSTATE.DIT on page B1-147.
• About the DIT bit on page E1-7139.

FEAT_FlagM, Flag manipulation instructions v2
FEAT_FlagM provides instructions which manipulate the PSTATE.{N,Z,C,V} flags.
These instructions are added to the A64 instruction set only.
This feature is OPTIONAL in Armv8.2 implementations.
This feature is mandatory in Armv8.4 implementations.
The ID_AA64ISAR0_EL1.TS field identifies the presence of FEAT_FlagM.
For more information, see Flag manipulation instructions on page C3-305.

FEAT_LRCPC2, Load-Acquire RCpc instructions v2
FEAT_LRCPC2 provides versions of LDAPR and STLlR with a 9-bit unscaled signed immediate offset.
These instructions are added to the A64 instruction set only.
This feature is OPTIONAL in Armv8.2 implementations.
This feature is mandatory in Armv8.4 implementations.
The ID_AA64ISAR1_EL1.LRCPC field identifies the presence of FEAT_LRCPC2.
For more information, see:
• Changes to single-copy atomicity in Armv8.4 on page B2-155.
• Non-exclusive Load-Acquire and Store-Release instructions on page C3-262.
• A64 instructions that are changed in Debug state on page H2-10243.

FEAT_LSE2, Large System Extensions v2
FEAT_LSE2 introduces changes to single-copy atomicity requirements for loads and stores, and changes to alignment requirements for loads and stores.
This feature is supported in AArch64 state only.
This feature is OPTIONAL in Armv8.2 implementations.
This feature is mandatory in Armv8.4 implementations.
The ID_AA64MMFR2_EL1.AT field identifies the presence of FEAT_LSE2.
For more information, see:
  • Requirements for single-copy atomicity on page B2-154.
  • Alignment of data accesses on page B2-189.

FEAT_TLBIOs, TLB invalidate instructions in Outer Shareable domain
FEAT_TLBIOs provides TLBI maintenance instructions that extend to the Outer Shareable domain.
This feature is supported in AArch64 state only.
This feature is mandatory in Armv8.4 implementations.
The field ID_AA64ISAR0_EL1.TLB identifies the presence of FEAT_TLBIOs.
For more information, see:
  • TLB maintenance instructions on page D8-5201.

FEAT_TLBIRANGE, TLB invalidate range instructions
FEAT_TLBIRANGE provides TLBI maintenance instructions that apply to a range of input addresses. FEAT_TLBIRANGE being implemented implies that FEAT_TLBIOs is implemented.
This feature is supported in AArch64 state only.
This feature is mandatory in Armv8.4 implementations.
The field ID_AA64ISAR0_EL1.TLB identifies the presence of FEAT_TLBIRANGE.
For more information, see:
  • TLB maintenance instructions on page D8-5201.
  • TLB maintenance instructions that apply to a range of addresses on page D8-5203.

FEAT_TTL, Translation Table Level
FEAT_TTL provides the TTL field to indicate the level of translation table walk holding the leaf entry for the address that is being invalidated. This field is provided in all TLB maintenance instructions that take a VA or an IPA argument.
This feature is supported in AArch64 state only.
This feature is mandatory in Armv8.4 implementations.
The field ID_AA64MMFR2_EL1.TTL identifies the presence of FEAT_TTL.
For more information, see:
  • TLB maintenance instructions on page D8-5201.
  • TLB maintenance instructions that apply to a range of addresses on page D8-5203.

FEAT_S2FWB, Stage 2 forced Write-Back
FEAT_S2FWB reduces the requirement of additional cache maintenance instructions in systems where the data Cacheability attributes used by the Guest operating system are different from those expected by the Hypervisor. If this feature is implemented, there is no meaningful distinction between the Inner and Outer Shareability domains for accesses to Normal Cacheable memory.
This feature is supported in AArch64 state.
This feature is mandatory in Armv8.4 implementations that implement EL2.
The ID_AA64MMFR2_EL1.FWB field identifies the presence of FEAT_S2FWB.
For more information, see:
  • Block descriptor and Page descriptor formats on page D8-5126.
  • Stage 2 memory type and Cacheability attributes when FEAT_S2FWB is enabled on page D8-5155.
FEAT_TTST, Small translation tables

FEAT_TTST relaxes the lower limit on the size of translation tables, by increasing the maximum permitted value of the T1SZ and T0SZ fields in TCR_EL1, TCR_EL2, TCR_EL3, VTCR_EL2 and VSTCR_EL2.

This feature is supported in AArch64 state only.

This feature is mandatory if FEAT_SEL2 is implemented.

This feature is optional if FEAT_SEL2 is not implemented.

The ID_AA64MMFR2_EL1.ST field identifies the presence of FEAT_TTST.

For more information, see:

• Input address size configuration on page D8-5090.
• Translation using the 4KB granule on page D8-5100.
• Translation using the 16KB granule on page D8-5106.
• Translation using the 64KB granule on page D8-5112.

FEAT_BBM, Translation table break-before-make levels

FEAT_BBM provides support to identify the requirements of hardware to have break-before-make sequences when changing between block size for a translation.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.4 implementations.

The ID_AA64MMFR2_EL1.BBM field identifies the presence of FEAT_BBM.

For more information, see:

• Block descriptor and Page descriptor formats on page D8-5126.
• Block translation entry on page D8-5160.
• Support levels for changing block size on page D8-5199.

FEAT_SEL2, Secure EL2

FEAT_SEL2 permits EL2 to be implemented in Secure state. When Secure EL2 is enabled, a translation regime is introduced that follows the same format as the other Secure translation regimes.

This feature is not supported if EL2 is using AArch32.

This feature is mandatory in Armv8.4 implementations that implement both EL2 and Secure state.

The ID_AA64PFR0_EL1.SEL2 field identifies the presence of FEAT_SEL2.

For more information, see:

• Translation regimes on page D8-5082.

FEAT_NV2, Enhanced nested virtualization support

FEAT_NV2 supports nested virtualization by redirecting register accesses that would be trapped to EL1 and EL2 to access memory instead. The address of the memory access depends on information held in introduced register, VNCR_EL2.

This feature is supported in AArch64 state only.

This feature is optional in Armv8.4 implementations.

The ID_AA64MMFR2_EL1.NV field identifies the presence of FEAT_NV2.

For more information, see Enhanced support for nested virtualization on page D8-5175.

FEAT_IDST, ID space trap handling

FEAT_IDST causes all AArch64 read accesses to the feature ID space when exceptions are generated to be reported in ESR_ELx using the EC code 0x18.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.4 implementations.
The ID_AA64MMFR2_EL1.IDS field identifies the presence of FEAT_IDST.

**FEAT_CNTSC, Generic Counter Scaling**

FEAT_CNTSC adds a scaling register to the memory-mapped counter module that allows the frequency of the counter that is generated to be scaled from the basic frequency reported in the counter ID mechanisms.

This feature is supported in both AArch64 and AArch32 states.

This feature is **OPTIONAL** in Armv8.4 implementations.

The CNTID.CNTSC field identifies the presence of FEAT_CNTSC.

For more information, see:

- *CNTCR, Counter Control Register* on page 15-10878.

**FEAT_Debugv8p4, Debug v8.4**

FEAT_Debugv8p4 covers a selection of mandatory changes:

- The fields MDCR_EL3.{EPMAD, EDAD} control Non-secure access to the debug and PMU registers. The bus Requester is responsible for other debug authentication.
- The Software Lock is obsolete.
- Non-invasive Debug controls are relaxed.
- Secure and Non-secure views of the debug registers are enabled.

This feature is mandatory if FEAT_SEL2 is implemented.

The fields that identify the presence of FEAT_Debugv8p4 are:

- ID_AA64DFR0_EL1.DebugVer.
- DBGDIDR.Version.
- ID_DFR0_EL1.{CopSDbg, CopDbg}.
- ID_DFR0.{CopSDbg, CopDbg}.
- EDDEV ARCH.ARCHID.

For more information, see:

- *Definition and constraints of a debugger in the context of external debug* on page H1-10228
- *External debug interface register access permissions* on page H8-10364

**FEAT_TRF, Self-hosted Trace Extensions**

FEAT_TRF adds controls of trace in a self-hosted system through System registers.

The feature provides:

- Control of Exception levels and Security states where trace generation is prohibited.
- Control of whether an offset is used for the timestamp recorded with trace information.
- A barrier instruction `TSB CSYNC` which can be used to prevent reordering of trace operation accesses with respect to other accesses of the same System registers.

If Armv8.4 and an ETM Architecture trace unit is implemented, this feature is mandatory. If this feature is implemented, the trace unit must include System register access to its control registers.

The reset state of the PE has prohibited regions controlled by the feature and not the external authentication signals. An external trace controller must override the internal controls before enabling trace, including trace from reset. This is a change from previous trace architectures and is not backwards-compatible.

The fields that identify the presence of FEAT_TRF are:

- ID_AA64DFR0_EL1.TraceFilt.
- ID_DFR0_EL1.TraceFilt.
- ID_DFR0.TraceFilt.
- EDDFR.TraceVer.
- ID_AA64DFR0_EL1.TraceVer.
For more information, see:

- Chapter D3 AArch64 Self-hosted Trace.
- Chapter G3 AArch32 Self-hosted Trace.

**FEAT_PMUv3p4, PMU Extensions v3.4**

FEAT_PMUv3p4 introduces the PMMIR_EL1 and PMMIR registers.

This feature is supported in both AArch64 and AArch32 states.

The Performance Monitors Extension is an OPTIONAL feature, but if it is implemented, an Armv8.4 implementation must include FEAT_PMUv3p4.

The fields that identify the presence of FEAT_PMUv3p4 are:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see *PMU events and event numbers* on page D11-5266.

**FEAT_RASv1p1, RAS Extension v1.1**

FEAT_RASv1p1 implements RAS System Architecture v1.1 and adds support for:

- Simplifications to ERR<n>STATUS.
- Additional ERR<n>MISC<m> registers.
- The OPTIONAL RAS Common Fault Injection Model Extension.

This feature is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.2 implementations and mandatory in Armv8.4 implementations.

The following fields identify the complete or partial presence of FEAT_RASv1p1:

- ID_AA64PFR0_EL1.RAS.
- ID_AA64PFR1_EL1.RAS_frac.
- ID_PFR0_EL1.RAS.
- ID_PFR2_EL1.RAS_frac.
- ID_PFR0.RAS.
- ID_PFR2.RAS_frac.

For more information, see:

- *Arm® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.

**FEAT_DoubleFault, Double Fault Extension**

FEAT_DoubleFault provides two controls:

- SCR_EL3.EASE.
- SCR_EL3.NMEA.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.4 implementations if EL3 is implemented and EL3 uses AArch64. Otherwise, it is not implemented.

This feature is implemented if ID_AA64PFR0_EL1.RAS >= 0b0010 and the implementation includes EL3 using AArch64.

For more information, see:

- *Arm® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile*.
A2.7.2 Features added to earlier extensions

The existing functionality of OS Double Lock is added as a feature mnemonic in Armv8.0, see `FEAT_DoubleLock` on page A2-78.
A2.8  The Armv8.5 architecture extension

The Armv8.5 architecture extension adds architectural features and additional requirements, see:

• Architectural features added by Armv8.5.
• Additional requirements of Armv8.5 on page A2-109.
• Features added to earlier extensions on page A2-110.
• Architectural requirements added to earlier extensions on page A2-110.
• Features added to the Armv8.5 extension in later releases on page A2-110.

A2.8.1  Architectural features added by Armv8.5

An implementation of the Armv8.5 extension must include all of the features that this section describes as mandatory. Such an implementation is also called an implementation of the Armv8.5 architecture.

The Armv8.5 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

**FEAT_FlagM2, Enhancements to flag manipulation instructions**

FEAT_FlagM2 provides instructions that convert between the PSTATE condition flag format used by the FCMP instruction and an alternative format described in *Relationship between ARM format and alternative format PSTATE condition flags* on page C6-1168.

These instructions are added to the A64 instruction set only.

This feature is mandatory in Armv8.5 implementations.

The ID_AA64ISAR0_EL1.TS field identifies the presence of FEAT_FlagM2.

For more information, see:

• Flag manipulation instructions on page C3-305.
• Relationship between ARM format and alternative format PSTATE condition flags on page C6-1168.

**FEAT_FRINTTS, Floating-point to integer instructions**

FEAT_FRINTTS provides instructions that round a floating-point number to an integral valued floating-point number that fits in a 32-bit or 64-bit integer number range.

These instructions are added to the A64 instruction set only.

This feature requires SIMD&FP, and is mandatory in Armv8.5 implementations when SIMD&FP is implemented.

The ID_AA64ISAR1_EL1.FRINTTS identifies the presence of FEAT_FRINTTS.

For more information, see *Floating-point round to integral value* on page C3-314.

**FEAT_ExS, Context synchronization and exception handling**

FEAT_ExS provides a mechanism to control whether exception entry and exception return are context synchronization events. Fields in the SCTLR_ELx registers enable and disable context synchronization at exception entry and return at an Exception level.

This feature is supported in AArch64 state only.

This feature is optional in Armv8.5 implementations.

The ID_AA64MMFR0_EL1.ExS identifies the presence of FEAT_ExS.

For more information, see:

• SCTLR_EL1, System Control Register (EL1) on page D17-6169, SCTLR_EL2 and SCTLR_EL3.
• Context synchronization event on page Glossary-11933

**FEAT_GTG, Guest translation granule size**

FEAT_GTG allows a hypervisor to support different granule sizes for stage 2 and stage 1 translation, and allows a nested hypervisor to determine what stage 2 granule sizes are available.
This feature is supported in AArch64 state only.
This feature is mandatory in Armv8.5 implementations.

The ID_AA64MMFR0_EL1.{TGran16_2, TGran64_2, TGran4_2} fields identify whether each of
the granule sizes is supported for stage 2 translation. The ID_AA64MMFR0_EL1.{TGran16,
TGran64, TGran4} fields identify whether each of the granule sizes is supported for stage 1
translations.

For more information, see Translation granules on page D8-5081.

FEAT_BTI, Branch Target Identification

FEAT_BTI allows memory pages to be guarded against the execution of instructions that are not the intended target of a branch. To do this, it introduces:

• The GP field, which denotes the blocks and pages in stage 1 translation tables that are guarded pages.
• The PSTATE.BTYPE field, which is used to determine whether an access to a guarded memory region will generate a Branch Target exception.
• The BTI instruction, which is used to guard against the execution of instructions that are not the intended target of a branch.

This feature is supported in AArch64 state only.
This feature is mandatory in Armv8.5 implementations.
The ID_AA64PFR1_EL1.BT field identifies the presence of FEAT_BTI.
For more information, see:
• Exception entry on page D1-4641.
• Synchronous exception types on page D1-4646.
• Table descriptor format on page D8-5123.
• PSTATE.BTYPE on page D8-5133.
• Effect of entering Debug state on PSTATE on page H2-10240.

FEAT_E0PD, Preventing EL0 access to halves of address maps

FEAT_E0PD prevents access at EL0 to half of the addresses in the memory map.
This feature is supported in AArch64 state only. When EL1 is using AArch64 state, this feature affects access to EL0, in either Execution state.
This feature is mandatory in Armv8.5 implementations.
Implementations that support FEAT_E0PD must also support FEAT_CSV3.
The ID_AA64MMFR2_EL1.E0PD field identifies presence of FEAT_E0PD.
For more information, see:
• Preventing EL0 access to halves of the address map on page D8-5097.
• TCR_EL1.{E0PD0, E0PD1}.
• TCR_EL2.{E0PD0, E0PD1}.

FEAT_RNG, Random number generator

FEAT_RNG introduces the RNDR and RNDRRS registers. Reads to these registers return a 64-bit random number. A read to RNDRRS will cause a reseeding of the random number before the generation of the random number that is returned.
This feature is supported in AArch64 state only.
This feature is OPTIONAL in Armv8.5 implementations.
The ID_AA64ISRAR0_EL1.RNDR field identifies presence of FEAT_RNG.
• Effect of random number generation instructions on Condition flags on page C6-1168.
• Appendix K14 Random Number Generation.
FEAT_MTE and FEAT_MTE2, Memory Tagging Extension

FEAT_MTE and FEAT_MTE2 provide architectural support for runtime, always-on detection of various classes of memory error to aid with software debugging to eliminate vulnerabilities arising from memory-unsafe languages.

These features are supported in AArch64 state only.

These features are optional in Armv8.5 implementations.

The ID_AA64FPFR1_EL1.MTE field identifies the presence of FEAT_MTE and FEAT_MTE2.

For more information, see:
• Chapter D9 The Memory Tagging Extension.
• Chapter B2 The AArch64 Application Level Memory Model.
• PMU events and event numbers on page D11-5266.
• Chapter D13 The Statistical Profiling Extension.
• Chapter H2 Debug State.

FEAT_PMUv3p5, PMU Extensions v3.5

FEAT_PMUv3p5 extends event counters to 64-bit event counters, and adds mechanisms to disable the cycle counter in Secure state and in EL2.

FEAT_PMUv3p5 relaxes the behavior of PMCR.{IMP, IDCODE}, and deprecates use of these fields.

This feature is supported in both AArch64 and AArch32 states.

The Performance Monitors Extension is an optional feature, but if it is implemented, an Armv8.5 implementation must include FEAT_PMUv3p5.

The fields that identify the presence of FEAT_PMUv3p5 are:
• ID_AA64DFR0_EL1.PMUVer.
• ID_DFR0_EL1.PerfMon.
• ID_DFR0.PerfMon.
• EDDFR.PMUVer.

For more information, see:
• Behavior on overflow on page D11-5250
• Controlling the PMU counters on page D11-5254.
• PMU events and event numbers on page D11-5266.

A2.8.2 Additional requirements of Armv8.5

The Armv8.5 architecture includes some mandatory changes that are not associated with a feature. These are:

Restrictions on effects of speculation

Further restrictions are placed on execution for:
• Execution prediction instructions that predict addresses or register values.
• Data loaded under speculation with a permission or domain fault.
• Any System register read under speculation to a register that is not architecturally accessible from the current Exception level.

For more information, see:
• Restrictions on the effects of speculation on page B2-171.
• Restrictions on the effects of speculation on page E2-7179.

Changes to CTIDEVAR, CTIDEVAFF0, and CTIDEVAFF1

CTIDEVAR, CTIDEVAFF0, and CTIDEVAFF1 must be implemented.

Changes to the input channel gate function
If the Cross Trigger Matrix (CTM) is implemented, the input channel gate function must be implemented.

Deprecation of EDPRCR.CWRR

EDPRCR.CWRR is deprecated.

Mandatory changes are also made to earlier architectural extensions, see Architectural requirements added to earlier extensions.

A2.8.3 Features added to earlier extensions

The features that have been added to earlier architectural extensions are:

- FEAT_SB on page A2-76.
- FEAT_SSBS on page A2-76.
- FEAT_CSV2 on page A2-76.
- FEAT_CSV3 on page A2-77.
- FEAT_SPECRES on page A2-78.
- FEAT_CP15SDISABLE2 on page A2-78.
- FEAT_EVT on page A2-93.
- FEAT_DPB2 on page A2-93.
- FEAT_SPEv1p1 on page A2-98.

A2.8.4 Architectural requirements added to earlier extensions

The additional architectural requirement that has been added to earlier extensions is Prefetch speculation protection on page A2-79.

A2.8.5 Features added to the Armv8.5 extension in later releases

FEAT_MTE3, MTE Asymmetric Fault Handling

FEAT_MTE3 introduces support for asymmetric Tag Check Fault handling.
This feature is optional in Armv8.5 implementations.
This feature is mandatory from Armv8.7 when FEAT_MTE2 is implemented.
This feature is supported in AArch64 state.
The ID_AA64PFR1_EL1.MTE field identifies the presence of FEAT_MTE3.
For more information, see Chapter D9 The Memory Tagging Extension.

FEAT_RNG_TRAP, Trapping support for RNDR/RNDRRS

FEAT_RNG_TRAP introduces support for EL3 trapping of reads of the RNDR and RNDRRS registers.
This feature is supported in AArch64 state.
This feature is optional in Armv8.5 implementations.
The ID_AA64PFR1_EL1.RNDR_trap field identifies the presence of FEAT_RNG_TRAP.
A2.9 The Armv8.6 architecture extension

The Armv8.6 architecture extension adds architectural features and additional requirements, see:

- Architectural features added by Armv8.6.
- Additional requirements of Armv8.6 on page A2-112.

Features are also added to earlier architecture extensions, see Features added to earlier extensions on page A2-112.

A2.9.1 Architectural features added by Armv8.6

An implementation of the Armv8.6 extension must include all of the features that this section describes as mandatory. Such an implementation is also called an implementation of the Armv8.6 architecture.

The Armv8.6 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

FEAT_ECV, Enhanced Counter Virtualization

FEAT_ECV enhances the Generic Timer architecture.

When executing in AArch64 state or AArch32 state, FEAT_ECV provides:

- Self-synchronizing views of the virtual and physical timers in AArch64 and AArch32 state.
- The ability to scale the generation of the event stream.

When EL2 is using AArch64 state, FEAT_ECV provides:

- An optional offset between the EL1 or EL0 view of physical time, and the EL2 or EL3 view of physical time.
- Traps configurable in CNTHCTL_EL2 that trap EL0 and EL1 access to the virtual counter or timer registers, and accesses to the physical timer registers when they are accessed using an EL02 descriptor.

The optional offset to views of physical time, and the configurable traps in CNTHCTL_EL2, both apply to EL1 and EL0 whether EL1 and EL0 are in AArch64 state or AArch32 state.

This feature is mandatory in Armv8.6 implementations.

The ID_AA64MMFR0_EL1.ECV field identifies the presence of FEAT_ECV. The ID_PFR1_EL1.GenTimer and ID_PFR1.GenTimer fields identify support for self-synchronized counter views in AArch32 state.

For more information, see:

- Self-hosted trace timestamps on page D3-4761.
- The profiling data on page D13-5446.
- The AArch64 view of the Generic Timer on page D10-5236.
- The AArch32 view of the Generic Timer on page G6-9292.

FEAT_FGT, Fine Grain Traps

FEAT_FGT introduces additional traps to EL2 of EL1 and EL0 access to individual or small groups of System registers and instructions, and traps to EL3 and EL2 of the Debug Communications Channel registers. The traps are independent of existing controls.

This feature is supported in AArch64, and when EL1 is using AArch64, EL0 accesses using AArch32 are also trapped.

This feature is mandatory in Armv8.6 implementations.

The ID_AA64MMFR0_EL1.FGT field identifies the presence of FEAT_FGT.

For more information, see Configurable instruction controls on page D1-4665.

FEAT_TWED, Delayed Trapping of WFE

FEAT_TWED introduces support for configurable delayed trapping of the WFE instruction.

This feature is supported in both AArch64 and AArch32 states.

This feature is OPTIONAL in Armv8.6 implementations.
The ID_AA64MMFR1_EL1.TWED field identifies the presence of FEAT_TWED.
For more information, see the Wait for Event and Wait for Event with Timeout instructions on page D1-4677.

FEAT_AMUv1p1, AMU Extensions v1.1
FEAT_AMUv1p1 introduces support for virtualization of Activity Monitors event counters, and introduces controls to disable access to auxiliary event counters below the highest Exception level.
This feature is supported in AArch32 state and AArch64 state, if the hypervisor is using AArch64.
This feature is optional in Armv8.6 implementations if the optional FEAT_AMUv1 is implemented.
The fields ID_AA64PFR0_EL1.AMU, ID_PFR0_EL1.AMU, and ID_PFR0.AMU identify the presence of FEAT_AMUv1p1.
For more information, see Chapter D12 The Activity Monitors Extension.

FEAT_MTPMU, Multi-threaded PMU Extensions
FEAT_MTPMU introduces controls to disable PMEVTYPE<n>_EL0.MT.
This feature requires at least one of EL2 and EL3. If neither is implemented, this feature is not implemented.
If EL2 or EL3 is implemented, the feature is optional if FEAT_PMUv3 is implemented.
Multithreaded Armv8.6 implementations with FEAT_PMUv3 implemented must implement FEAT_MTPMU to enable any multithreaded event counting.
This feature is supported in both AArch64 and AArch32 states.
The fields ID_AA64DFR0_EL1.MTPMU and ID_DFR1.MTPMU identify the presence of FEAT_MTPMU.
For more information, see:
- Multithreaded implementations on page D11-5258.
- MDACR_EL3.MTPME, SDCR.MTPME, MDACR_EL2.MTPME, and HDACR.MTPME.
- Common event numbers on page D11-5306.

A2.9.2 Additional requirements of Armv8.6
The Armv8.6 architecture includes some mandatory changes that are not associated with a feature. These are:

Changes to the frequency of the physical counter
The frequency of CNTFRQ_EL0 is standardized to a frequency of 1GHz. This means that the system counter must be implemented at 64 bits. For more information, see:
- The system counter on page D10-5234.
- The system counter on page G6-9290.

A2.9.3 Features added to earlier extensions
The features that have been added to earlier architectural extensions are:
- FEAT_DGH on page A2-78.
- FEAT_ETS on page A2-78.
- FEAT_BF16 on page A2-93.
- FEAT_AA32BF16 on page A2-94.
- FEAT_I8MM on page A2-94.
- FEAT_AA32I8MM on page A2-94.
- FEAT FPAC on page A2-99.
A2.10 The Armv8.7 architecture extension

The Armv8.7 architecture extension adds architectural features and additional requirements, see:

- Architectural features added by Armv8.7.
- Additional requirements of Armv8.7 on page A2-116.

Features are also added to earlier architecture extensions, see Features added to earlier extensions on page A2-116.

A2.10.1 Architectural features added by Armv8.7

An implementation of the Armv8.7 extension must include all of the features that this section describes as mandatory. Such an implementation is also called an implementation of the Armv8.7 architecture.

The Armv8.7 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

- **FEAT_AFP, Alternate floating-point behavior**
  
  FEAT_AFP allows alternate behavior for specified floating-point instructions including:
  - Flushing of denormalized numbers to zero can be controlled separately on inputs and outputs.
  - Alternate NaN propagation rules can apply.
  - Output elements for specified scalar Advanced SIMD instructions can be determined using alternate rules.
  - Changes to floating-point exception generation.
  
  This feature is supported in AArch64 state only.
  
  This feature is mandatory in Armv8.7 implementations that implement floating-point support.
  
  The ID_AA64MMFR1_EL1.AFP field identifies the presence of FEAT_AFP.
  
  For more information, see:
  - Flushing denormalized numbers to zero on page A1-58.
  - NaN handling and the Default NaN on page A1-60.
  - Floating-point exceptions and exception traps on page A1-64.

- **FEAT_RPRES, Increased precision of Reciprocal Estimate and Reciprocal Square Root Estimate**
  
  FEAT_RPRES allows an increase in the precision of the Reciprocal Estimate and Reciprocal Square Root Estimate from an 8-bit mantissa to a 12-bit mantissa.
  
  This feature is supported in AArch64 state only.
  
  This feature is OPTIONAL in Armv8.7 implementations. This feature requires implementation of FEAT_AFP.
  
  The ID_AA64ISAR2_EL1.RPRES field identifies the presence of FEAT_RPRES.
  
  For more information, see RecipEstimate() and RecipSqrtEstimate().

- **FEAT_LS64, FEAT_LS64_V, FEAT_LS64_ACCDATA, Support for 64 byte loads/stores**
  
  FEAT_LS64 introduces support for atomic single-copy 64-byte loads and stores without return. For more information, see:
  - LD64B on page C6-1512.
  - ST64B on page C6-1837.
  
  FEAT_LS64_V introduces support for atomic single-copy 64-byte stores with return. For more information, see:
  - ST64BV on page C6-1838.
  
  FEAT_LS64_ACCDATA introduces support for atomic single-copy 64-byte EL0 stores with return. For more information, see:
  - ST64BV0 on page C6-1840.
  - ACCDATA_EL1, Accelerator Data on page D17-5556.
These features are supported in AArch64 state only.
FEAT_LS64 is OPTIONAL in Armv8.7 implementations.
FEAT_LS64_V is OPTIONAL in Armv8.7 implementations, and can be implemented only if 
FEAT_LS64 is implemented.
FEAT_LS64_ACCDATA is OPTIONAL in Armv8.7 implementations, and can be implemented only 
if FEAT_LS64_V is implemented.

--- Note ---

The meaning of any values being returned by the ST64BV and ST64BV0 instructions are 
retrospectively relaxed such that they are defined by the peripheral providing the response.

The ID_AA64ISAR1_EL1.LS64 field identifies the presence of FEAT_LS64, FEAT_LS64_V, and 
FEAT_LS64_ACCDATA.

For more information, see Single-copy atomic 64-byte load/store on page C3-271.

FEAT_WFxT, WFE and WFI instructions with timeout

FEAT_WFxT introduces WFET and WFIT. These instructions support the generation of a local timeout 
event to act as a wake-up event for the PE when the virtual count in CNTVCT_EL0 equals or 
exceeds the value supplied by the instruction for the first time. The register number that holds the 
timeout value for trapped WFET and WFIT instructions is reported in ESR_ELx.

These instructions are added to the A64 instruction set only.

FEAT_WFxT is mandatory in Armv8.7 implementations.
The ID_AA64ISAR2_EL1.WFxT field identifies the presence of FEAT_WFxT.

For more information, see:
• Instructions with register argument on page C3-250.
• Wait for Event on page D1-4676.
• Wait for Interrupt mechanism on page D1-4678.

FEAT_HCX, Support for the HCRX_EL2 register

FEAT_HCX introduces the Extended Hypervisor Configuration Register, HCRX_EL2, that 
provides configuration controls for virtualization in addition to those provided by HCR_EL2, 
including defining whether various operations are trapped to EL2.

This feature is supported in AArch64 state only.
This feature is mandatory in Armv8.7 implementations.
The ID_AA64MMFR1_EL1.HCX field identifies the presence of FEAT_HCX.

For more information, see Configurable instruction controls on page D1-4665.

FEAT_LPA2, Larger physical address for 4KB and 16KB translation granules

FEAT_LPA2:
• Allows a larger VA space for each translation table base register of up to 52 bits when using 
  the 4KB or 16KB translation granules.
• Allows a larger intermediate physical address (IPA) and PA space of up to 52 bits when using 
  the 4KB or 16KB translation granules.
• Allows a level 0 block size where the block covers a 512GB address range for the 4KB 
  translation granule if the implementation supports 52 bits of PA.
• Allows a level 1 block size where the block covers a 64GB address range for the 16KB 
  translation granule if the implementation supports 52 bits of PA.

This feature is supported in AArch64 state only.
This feature is OPTIONAL in Armv8.7 implementations. This feature requires implementation of 
FEAT_LPA and FEAT_LVA.
The ID_AA64MMFR0_EL1.{TGRAN4_2, TGRAN16_2, TGRAN4, TGRAN16} fields identify the presence of FEAT_LPA2.

For more information, see:
- Implemented physical address size on page D8-5088.
- Output address size configuration on page D8-5088.
- Supported virtual address ranges on page D8-5089.
- Input address size configuration on page D8-5090.
- Intermediate physical address size configuration on page D8-5091.
- Translation using the 4KB granule on page D8-5100.
- Translation using the 16KB granule on page D8-5106.
- Table descriptor format on page D8-5123.
- Block descriptor and Page descriptor formats on page D8-5126.

**FEAT_XS, XS attribute**

FEAT_XS introduces the XS attribute for memory to indicate that an access could take a long time to complete. This feature provides variants of DSB instructions and TLB maintenance instructions, the completion of which does not depend on the completion of memory accesses with the XS attribute.

FEAT_XS adds:
- A mechanism to define the XS attribute for memory.
- An optional nXS variant to the AArch64 DSB instruction and optional nXS qualifier to each AArch64 TLBI instruction to handle memory accesses with the XS attribute.
- The FGTnXS bit to HCRX_EL2 to determine the behavior of fine-grained traps in HFGITR_EL2 for TLB maintenance instructions with the nXS qualifier.
- The FnXS bit to HCRX_EL2 to determine the behavior of pre-existing TLB maintenance instructions in relation to the XS attribute.

This feature is supported in AArch64 state only, but the XS attribute also impacts AArch32 state execution.

This feature is mandatory in Armv8.7 implementations.

The ID_AA64ISAR1_EL1.XS field identifies the presence of FEAT_XS.

For more information, see:
- Data Synchronization Barrier (DSB) on page B2-178.
- Behavior when stage 1 address translation is disabled on page D8-5118.
- Block descriptor and Page descriptor formats on page D8-5126.
- Stage 1 memory type and Cacheability attributes on page D8-5151.
- XS attribute modifier on page D8-5160.
- Data Synchronization Barrier (DSB) on page E2-7182.
- Overview of memory region attributes for stage 1 translations on page G5-9203.
- Ordering and completion of TLB maintenance instructions on page G5-9223.

**FEAT_PMUv3p7, Armv8.7 PMU extensions**

FEAT_PMUv3p7 adds the following features to the Performance Monitors Extension:
- PMU counters can be frozen when an event counter has an unsigned overflow.
- Event counters can be prohibited from counting events at EL3 without affecting the rest of Secure state.
- The cycle counter can be prohibited from counting cycles at EL3 without affecting the rest of Secure state.

This feature is supported in both AArch64 and AArch32 states.
The Performance Monitors Extension is an **optional** feature, but if it is implemented, an Armv8.7 implementation must include **FEAT_PMUv3p7**.

The fields that identify the presence of **FEAT_PMUv3p7** are:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see:
- *Controlling the PMU counters* on page D11-5254.
- *Freezing event counters* on page D11-5255.
- *Common microarchitectural events* on page D11-5317.
- *PMMIR_EL1, Performance Monitors Machine Identification Register* on page D17-6788.

**FEAT_SPEv1p2, Armv8.7 SPE features**

**FEAT_SPEv1p2** adds the following features to the Statistical Profiling Extension, **FEAT_SPE**:

- Adds an inverse event filter control.
- Adds controls to freeze the PMU event counters after an SPE buffer management event occurs.
- Adds a discard mode that allows all SPE data to be discarded rather than written to memory.

This feature is mandatory from Armv8.7 when **FEAT_SPE** is implemented.

This feature is supported in AArch64 state.

**FEAT_SPEv1p2** optionally enables support for a packet for each taken branch that provides the target address for the previous taken branch.

**ID_AA64DFR0_EL1.PMSVer** identifies the presence of **FEAT_SPEv1p2**.

If **FEAT_SPEv1p2** is implemented, **PMSIDR_EL1.PBT** indicates support for the previous branch target packet.

For more information, see:
- *Freezing event counters* on page D11-5255.
- *Common event numbers* on page D11-5306.
- *Filtering sample records* on page D13-5444.
- *Previous branch target* on page D13-5447.
- *About the Statistical Profiling Extension sample records* on page D14-5472.
- *Address packet* on page D14-5476.

### A2.10.2 Additional requirements of Armv8.7

The Armv8.7 architecture includes some mandatory changes that are not associated with a feature. These are:

**FEAT_ETS, Enhanced Translation Synchronization**

All implementations of the Armv8.7 architecture are required to implement **FEAT_ETS**.

For more information, see **FEAT_ETS on page A2-78**.

### A2.10.3 Features added to earlier extensions

The features that have been added to earlier architectural extensions are:

- **FEAT_PAN3** on page A2-85.
- **FEAT_MTE3** on page A2-110
A2.11 The Armv8.8 architecture extension

The Armv8.8 architecture extension adds architectural features and additional requirements, see:

- Architectural features added by Armv8.8
- Additional requirements of Armv8.8 on page A2-120.

Features are also added to earlier architecture extensions, see Features added to earlier extensions on page A2-120.

A2.11.1 Architectural features added by Armv8.8

An implementation of the Armv8.8 extension must include all of the features that this section describes as mandatory. Such an implementation is also called an implementation of the Armv8.8 architecture.

The Armv8.8 architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

FEAT_MOPS, Standardization of memory operations

FEAT_MOPS provides instructions that perform a memory copy or memory set, and adds Memory Copy and Memory Set exceptions.

FEAT_MOPS also adds the HCRX_EL2.{MSCEn, MCE2}, SCTLR_EL1.MSCEn, and SCTLR_EL2.MSCEn control bits.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.8 implementations.

The ID_AA64ISAR2_EL1.MOPS field identifies the presence of FEAT_MOPS.

For more information, see:
- Memory Copy and Memory Set exceptions on page D1-4654.

FEAT_HBC, Hinted conditional branches

FEAT_HBC provides the BC.cond instruction to give a conditional branch with a hint to branch prediction logic that this branch will consistently and is highly unlikely to change direction.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.8 implementations.

The ID_AA64ISAR2_EL1.BC field identifies the presence of FEAT_HBC.

For more information, see:
- Conditional branch on page C3-248.

FEAT_NMI, Non-maskable Interrupts

FEAT_NMI provides a mechanism to support non-maskable interrupts (NMI) and less-masked interrupts (LMI). In addition to legacy behavior, the feature includes the following:

- A mode for supporting an LMI interrupt mask that is distinct from PSTATE.{I, F}.
- A mode for supporting a limited NMI, where the value when PSTATE.SP is 1 is taken as an interrupt mask for all interrupts targeting that Exception level, and where the LMI interrupt mask can also be used.

FEAT_NMI adds:
- The AllIntMask variable.
- An optional Superpriority attribute to denote virtual and physical IRQ and FIQ interrupts as non-maskable.
- The SCTLR_EL{x}.{NMI, SPINTMASK} control bits.
- The PSTATE.ALLINT bit and associated instructions.
- The HCRX_EL2.TALLINT bit to enable trapping of ALLINT instructions at EL1.

This feature is supported in AArch64 state only.

This feature is mandatory in Armv8.8 implementations.

The ID_AA64PFR1_EL1.NMI field identifies the presence of FEAT_NMI.
FEAT_TIDCP1, EL0 use of IMPLEMENTATION DEFINED functionality

FEAT_TIDCP1 adds a control at EL1 and EL2 to enable trapping of EL0 accesses to registers that might control IMPLEMENTATION DEFINED functions.

This feature adds controls only in AArch64 state, and controls IMPLEMENTATION DEFINED execution at EL0 in both AArch32 and AArch64 states.

This feature is mandatory in Armv8.8 implementations.

The ID_AA64MMFR1_EL1.TIDCP1 field identifies the presence of FEAT_TIDCP1.

For more information, see Prioritization of Synchronous exceptions taken to AArch64 state on page D1-4650.

FEAT_CMOW, Control for cache maintenance permission

FEAT_CMOW introduces support for cache maintenance instructions that controls whether:

- Cache maintenance instructions executed at EL0 require stage 1 read and write permission to prevent the instructions from generating a Permission fault.
- Cache maintenance instructions executed at EL1 or EL0 require stage 2 read and write permission to prevent the instructions from generating a Permission fault.

This feature is supported in AArch64 state only, but also impacts AArch32 instructions.

This feature is mandatory in Armv8.8 implementations.

The ID_AA64MMFR1_EL1.CMOW field identifies the presence of FEAT_CMOW.

For more information, see:
- A64 Cache maintenance instructions on page D7-5054.
- Permission fault on page D8-5183.

FEAT_PMUv3p8, Armv8.8 PMU extensions

FEAT_PMUv3p8 adds the following features to the Performance Monitors Extension:

- The Common event number space is extended to include the ranges 0x0040-0x00BF and 0x4040-0x40BF.
- For an event counter n, if any reserved or unimplemented PMU event number is written to PMEVTYPE<\textit{n}\.evtCount, the event counter \textit{n} does not count, and a read of PMEVTYPE<\textit{n}\.evtCount returns the value written.

This feature is supported in both AArch64 and AArch32 states.

The Performance Monitors Extension is an OPTIONAL feature, but if it is implemented, an Armv8.8 implementation must include FEAT_PMUv3p8.

The fields that identify the presence of FEAT_PMUv3p8 are:

- ID_AA64DFR0_EL1.PMUVer.
- ID_DFR0_EL1.PerfMon.
- ID_DFR0.PerfMon.
- EDDFR.PMUVer.

For more information, see:
- The PMU event number space and common events on page D11-5275.
- Common event numbers on page D11-5306.
- PMEVTYPE<\textit{n}>_EL0, Performance Monitors Event Type Registers, \textit{n} = 0 - 30 on page D17-6773.
- PMEVTYPE<\textit{n}>, Performance Monitors Event Type Registers, \textit{n} = 0 - 30 on page G8-10006.
FEAT_HPMN0, Setting of MDCR_EL2.HPMN to zero

FEAT_HPMN0 permits a hypervisor to provide zero PMU event counters for a guest operating system by setting MDCR_EL2.HPMN to zero.

This feature is supported in both AArch64 and AArch32 states.

This feature is mandatory in Armv8.8 implementations that include FEAT_PMUv3 and EL2. Otherwise, this feature is optional in implementations that include FEAT_PMUv3, FEAT_FGT, and EL2.

The fields that identify the presence of FEAT_HPMN0 are:
- ID_AA64DFR0_EL1.HPMN0.
- ID_DFR1_EL1.HPMN0.
- ID_DFR1.HPMN0.

For more information, see:
- Interaction with EL2 on page D11-5245.
- Controlling the PMU counters on page D11-5254.
- The Performance Monitors Extension on page K1-11569.
- The Performance Monitors Extension on page K1-11586.

FEAT_PMUv3_TH, Event counting threshold

FEAT_PMUv3_TH adds threshold condition controls to each PMEVTYPER<n>_EL0 register. This feature permits the counter to count only when PMEVTYPER<n>_{MT, evtCount} describes an event whose count meets a specified threshold condition.

This feature is supported in both AArch64 and AArch32 states. The threshold condition controls are only accessible in AArch64 state. However, threshold conditions still apply in AArch32 state.

This feature is optional in Armv8.8 implementations.

This feature requires FEAT_PMUv3 to be implemented, and AArch64 state to be supported.

The fields that identify the presence of FEAT_PMUv3_TH are:
- PMMIR_EL1.THWIDTH.
- PMMIR.THWIDTH.
- If the external debug interface to the PMU registers is implemented, PMMIR.THWIDTH.

For more information, see Event counting threshold on page D11-5262.

FEAT_SPEv1p3, Armv8.8 Statistical Profiling Extensions

FEAT_SPEv1p3 adds the following features to the Statistical Profiling Extension:
- Support for sampling Tag operations.
- Support for sampling Memory Copy and Set operations.

This feature is supported in both AArch64 and AArch32 states.

This feature is mandatory from Armv8.8 when FEAT_SPE is implemented.

The ID_AA64DFR0_EL1.PMSVer field identifies the presence of FEAT_SPEv1p3.

For more information, see:
- Additional information for each profiled memory access operation on page D13-5448.
- About the Statistical Profiling Extension sample records on page D14-5472.
- Address packet on page D14-5476.

FEAT_Debugv8p8, Debug v8.8

FEAT_Debugv8p8 adds support to allow an asynchronous exception to be taken after an exception generates an Exception Catch debug event, but before the PE halts.

This feature is supported in both AArch64 and AArch32 states.

This feature is mandatory in Armv8.8 implementations.

The fields that identify the presence of FEAT_Debugv8p8 are:
- ID_AA64DFR0_EL1.DebugVer,
A2.11.2 Additional requirements of Armv8.8

The Armv8.8 architecture includes the following mandatory changes that are associated with FEAT_SPE:

**Access Flag and dirty state management by SPE**

Removal of the option for the Access Flag and dirty state to behave as if always disabled for accesses made by the SPU. For more information, see *Hardware management of dirty state and the Access flag by the Statistical Profiling Extension* on page D13-5465.

**External abort handling by SPE**

External aborts on writes made by the SPU must generate SError interrupt exceptions. For more information, see *External aborts* on page D13-5466.

The Armv8.8 architecture includes the following optional change that is associated with FEAT_PMUv3:

**64-bit external PMU programmers’ model extension**

The 64-bit external PMU programmers’ model extension extends the Performance Monitors registers to 64 bits, except the 32-bit CoreSight management registers.

This extension is supported in both AArch64 and AArch32 states.

This extension is optional when the external debug interface to the Performance Monitors is implemented.

This extension requires the external debug interface to the Performance Monitors to be implemented, and AArch64 state to be supported.

The PMDEVARCH.ARCHPART field indicates the presence of the 64-bit external PMU programmers’ model extension.

For more information, see *Chapter I3 Recommended External Interface to the Performance Monitors*.

A2.11.3 Features added to earlier extensions

The features that have been added to earlier architectural extensions are:

- **FEAT_PACQARMA3** on page A2-100.
- **FEAT_CONSTPACFIELD** on page A2-100.
- **FEAT_RNG_TRAP** on page A2-110.
The Performance Monitors Extension

The Performance Monitors Extension, FEAT_PMUv3, is an OPTIONAL extension but Arm strongly recommends that Armv8-A implementations include version 3 of the Performance Monitors Extension.

ID_AA64DFR0_EL1.PMUVer indicates whether the Performance Monitors Extension is implemented.

For more information, see Chapter D11 *The Performance Monitors Extension*.

Armv8.1 introduces the following architectural feature to the Performance Monitors Extension:
- FEAT_PMUv3p1.

Armv8.4 introduces the following architectural feature to the Performance Monitors Extension:
- FEAT_PMUv3p4.

Armv8.5 introduces the following architectural feature to the Performance Monitors Extension:
- FEAT_PMUv3p5.

Armv8.6 introduces the following architectural feature to the Performance Monitors Extension:
- FEAT_MTPMU.

Armv8.7 introduces the following architectural feature to the Performance Monitors Extension:
- FEAT_PMUv3p7.

Armv8.8 introduces the following architectural features to the Performance Monitors Extension:
- FEAT_PMUv3p8.
- FEAT_HPMN0.
- FEAT_PMUv3_TH.
A2.13 The Reliability, Availability, and Serviceability Extension

The RAS Extension, FEAT_RAS, is a mandatory extension to the Armv8.2 architecture, and an OPTIONAL extension to the Armv8.0 and the Armv8.1 architectures.

The RAS Extension improves the dependability of a system by providing:

- Reliability, that is, the continuity of correct service.
- Availability, that is, the readiness for correct service.
- Serviceability, that is, the ability to undergo modifications and repairs.

ID_AA64PFR0_EL1.RAS in AArch64 state, and ID_PFR0.RAS in AArch32 state, indicate whether the RAS Extension is implemented.

The RAS Extension introduces a barrier instruction, the Error Synchronization Barrier (ESB), to the A32, T32, and A64 instruction sets.

System registers introduced by the RAS Extension are described in:

- For AArch64, RAS registers on page D17-6956.
- For AArch32, RAS registers on page G8-10083.

In addition, the RAS Extension introduces a number of memory-mapped registers. These are described in the Arm® Architecture Reference Manual Supplement: Reliability, Availability, and Serviceability (RAS), for Armv8-A.

Armv8.2 introduces the following architectural features to the RAS Extension:

- FEAT_IESB.

Armv8.4 introduces the following architectural features to the RAS Extension:

- FEAT_RASv1p1.
- FEAT_DoubleFault.
A2.14 The Statistical Profiling Extension (SPE)

The Statistical Profiling Extension, FEAT_SPE, is an OPTIONAL extension introduced by the Armv8.2 architecture. Implementation of the Statistical Profiling Extension requires implementation of at least Armv8.1 of the Armv8-A architecture profile. The Statistical Profiling Extension is supported only in AArch64 state.

The Statistical Profiling Extension provides a non-invasive method of sampling software and hardware using randomized sampling of either architectural instructions, as defined by the instruction set architecture, or by microarchitectural operations.

ID_AA64DFR0_EL1.PMSVer indicates whether the Statistical Profiling Extension is implemented.

For more information, see Chapter D13 The Statistical Profiling Extension.

Armv8.3 introduces the following architectural feature to the SPE:
• FEAT_SPEv1p1.

Armv8.7 introduces the following architectural feature to the SPE:
• FEAT_SPEv1p2.

Armv8.8 introduces the following architectural feature to the SPE:
• FEAT_SPEv1p3.
A2.15 The Scalable Vector Extension (SVE)

The Scalable Vector Extension, FEAT_SVE, is an OPTIONAL extension introduced by the Armv8.2 architecture. SVE is supported in AArch64 state only.

The Scalable Vector Extension includes the following functionality:

- Configurable vector length with scalable vector lengths from 128 bits up to 2048 bits.
- Predication using scalable predicate registers from 16 bits up to 256 bits.
- Instructions that operate on scalable size vectors and predicates.
- Gather-load and scatter-store.
- Software-managed speculative vectorization.
- System registers and fields to configure the Effective SVE vector length and traps.

ID_AAA64PFR0_EL1.SVE indicates whether the Scalable Vector Extension is implemented.

The Scalable Vector Extension complements the AArch64 Advanced SIMD and floating-point functionality. SVE does not replace the AArch64 Advanced SIMD and floating-point functionality.

Implementation of FEAT_SVE requires implementation of FEAT_FCMA and FEAT_FP16.
A2.16 The Activity Monitors Extension (AMU)

The Activity Monitors Extension is an OPTIONAL extension introduced by the Armv8.4 architecture. AMU is supported in AArch64 and AArch32 states.

The Activity Monitors Extension implements version 1 of the Activity Monitors architecture, FEAT_AMUv1, which provides a function similar to a subset of the existing Performance Monitors Extension functionality, intended for system management use rather than debugging and profiling.

The Activity Monitors Extension implements a System register interface to the Activity Monitors registers, and supports an optional external memory-mapped interface.

The fields that identify the presence of the Activity Monitors Extension are:

- ID_AA64PFR0_EL1.AMU.
- ID_PFR0_EL1.AMU.
- ID_PFR0.AMU.
- EDPFR.AMU.

For more information, see Chapter D12 The Activity Monitors Extension.

Armv8.6 introduces the following architectural feature to the Activity Monitors Extension:

- FEAT_AMUv1p1.
A2.17 The Memory Partitioning and Monitoring (MPAM) Extension

The MPAM Extension, FEAT_MPAM, is an optional extension introduced by the Armv8.4 architecture and requires implementation of at least Armv8.2 of the Armv8-A architecture profile. MPAM is supported in AArch64 state only.

The MPAM Extension provides a framework for memory-system component controls that partition one or more of the performance resources of the component.

The fields that identify the presence of the MPAM Extension are:

- ID_AA64PFR0_EL1.MPAM.
- EDPFR.MPAM.

For more information, see Arm® Architecture Reference Manual Supplement, Memory System Resource Partitioning and Monitoring (MPAM), for A-profile architecture.
Chapter A3

Armv9 Architecture Extensions

This chapter introduces the Armv9 architecture versions and extensions. It contains the following sections:

• *Armv9-A architecture extensions* on page A3-128.
A3.1 Armv9-A architecture extensions

An implementation of the Armv9-A architecture must include all of the extensions that this section describes as mandatory. Such an implementation is also called an implementation of the Armv9-A architecture.

The AArch32 state might optionally be implemented at EL0. The AArch32 state is not implemented at EL1, EL2, and EL3.

The implementation of FEAT_DoubleLock in an Armv9 implementation is prohibited.

An implementation of the Armv9-A architecture cannot include an ETM.

The Armv9 architecture adds the following features, which are identified by the architectural feature name and a short description of the feature:

**FEAT_SVE2, Scalable Vector Extension version 2**

The Scalable Vector Extension version 2 (SVE2) is a superset of SVE that incorporates functionality similar to Advanced SIMD, and other enhancements. In this Manual, unless stated otherwise, when SVE is used, the behavior also applies to SVE2.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in an Armv9.0 implementation, standard Armv9-A software platforms support FEAT_SVE2.

This feature requires FEAT_SVE.

The following fields indicate the presence of FEAT_SVE2:

- ID_AA64PFR0_EL1.SVE.
- ID_AA64ZFR0_EL1.SVEver.

For more information, see:

- *The Scalable Vector Extension (SVE)* on page A2-124.
- *Data processing - SVE2* on page C3-368.

**FEAT_SVE_AES, Scalable Vector AES instructions**

FEAT_SVE_AES provides the following SVE AES cryptographic instructions:

- AESD.
- AESE.
- AESIMC.
- AESMC.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in an Armv9.0 implementation.

FEAT_SVE_AES requires FEAT_SVE2.

The ID_AA64ZFR0_EL1.AES field identifies support for FEAT_SVE_AES.

**FEAT_SVE_BitPerm, Scalable Vector Bit Permutes instructions**

FEAT_SVE_BitPerm provides the following SVE bit permute instructions:

- BEXT.
- BDEP.
- BGRP.

This feature is supported in AArch64 state only.

This feature is OPTIONAL in an Armv9.0 implementation.

FEAT_SVE_BitPerm requires FEAT_SVE2.

The ID_AA64ZFR0_EL1.BitPerm field identifies support for FEAT_SVE_BitPerm.
FEAT_SVE_PMULL128, Scalable Vector PMULL instructions
FEAT_SVE_PMULL128 provides the following SVE 128-bit polynomial multiply instructions:
- PMULLB.
- PMULLT.
This feature is supported in AArch64 state only.
This feature is optional in an Armv9.0 implementation.
FEAT_SVE_PMULL128 requires FEAT_SVE2.
The ID_AA64ZFR0_EL1.AES field identifies support for FEAT_SVE_PMULL128.

FEAT_SVE_SHA3, Scalable Vector SHA3 instructions
FEAT_SVE_SHA3 provides the following SVE SHA3 cryptographic instruction:
- RAX1.
This feature is supported in AArch64 state only.
This feature is optional in an Armv9.0 implementation.
FEAT_SVE_SHA3 requires FEAT_SVE2.
The ID_AA64ZFR0_EL1.SHA3 field identifies support for FEAT_SVE_SHA3.

FEAT_SVE_SM4, Scalable Vector SM4 instructions
FEAT_SVE_SM4 provides the following SVE SM4 cryptographic instructions:
- SM4E.
- SM4EKEY.
This feature is supported in AArch64 state only.
This feature is optional in an Armv9.0 implementation.
FEAT_SVE_SM4 requires FEAT_SVE2.
The ID_AA64ZFR0_EL1.SM4 field identifies support for FEAT_SVE_SM4.

FEAT_ETE, Embedded Trace Extension
FEAT_ETE provides a trace unit that records details about software control flow running on a PE, which can be used to aid debugging or optimizing. The trace unit provides filtering functionality to allow the targeting of the information to specific code regions or periods of operation.
This feature is supported in AArch64 state, and performs trace in both AArch64 and AArch32 states.
This feature is optional in an Armv9.0 implementation.
This feature requires FEAT_TRBE and FEAT_TRF.
The ID_AA64DFR0_EL1.TraceVer field identifies the presence of FEAT_ETE.
For more information, see Chapter D4 The Embedded Trace Extension.

FEAT_TME, Transactional Memory Extension
FEAT_TME introduces a set of instructions to support hardware transaction memory, which means a group of instructions can appear to be collectively executed as a single atomic operation. For more information on these instructions, see:
- TCANCEL.
- TCOMMIT.
- TSTART.
- TTEST.
This feature is supported in AArch64 state only.
This feature is optional in an Armv9.0 implementation.
The ID_AA64ISAR0_EL1.TME field identifies the presence of FEAT_TME.
For more information on FEAT_TME, see Arm® Architecture Reference Manual Supplement, Transactional Memory Extension (TME), for A-profile architecture (ARM DDI 0617).

**FEAT_TRBE, Trace Buffer Extension**

FEAT_TRBE enables support for a Trace Buffer Unit within a PE. When the Trace Buffer Unit is enabled, program-flow trace generated by a trace unit is written directly to memory by the Trace Buffer Unit, rather than routing trace data to a trace sink.

This feature is supported in AArch64 state, and performs trace in both AArch64 and AArch32 states.

This feature is **OPTIONAL** in an Armv9.0 implementation.

FEAT_TRBE requires FEAT_ETE and FEAT_TRF.

The ID_AA64DFR0_EL1.TraceBuffer field identifies support for FEAT_TRBE.

For more information, see Chapter D6 The Trace Buffer Extension.

**FEAT_ETEv1p1, Embedded Trace Extension**

FEAT_ETEv1p1 extends FEAT_ETE to support FEAT_FGT.

This feature is supported in AArch64 state, and performs trace in both AArch64 and AArch32 states.

This feature is **OPTIONAL** in an Armv9.1 implementation.

This feature requires FEAT_ETE.

The TRCDEVARCH.REVISION field identifies the presence of FEAT_ETEv1p1.

For more information, see Chapter D4 The Embedded Trace Extension.

**FEAT_BRBE, Branch Record Buffer Extension**

FEAT_BRBE provides a Branch record buffer for capturing control path history.

This feature is supported in AArch64 state only.

This feature is **OPTIONAL** in an Armv9.2 implementation.

The ID_AA64DFR0_EL1.BRBE field identifies the presence of FEAT_BRBE.

For more information, see Chapter D15 The Branch Record Buffer Extension.

**FEAT_ETEv1p2, Embedded Trace Extension**

FEAT_ETEv1p2 extends FEAT_ETE to support FEAT_RME.

This feature is supported in AArch64 state, and performs trace in both AArch64 and AArch32 states.

This feature is **OPTIONAL** in an Armv9.2 implementation.

This feature requires FEAT_ETEv1p1.

This feature is required if FEAT_RME is implemented and FEAT_ETE is implemented.

The TRCDEVARCH.REVISION field identifies the presence of FEAT_ETEv1p2.

For more information, see Arm® Architecture Reference Manual Supplement, The Realm Management Extension (RME), for Armv9-A (ARM DDI 0615).

**FEAT_BRBEv1p1, Branch Record Buffer Extension**

FEAT_BRBEv1p1 enables branch recording at EL3.

This feature is supported in AArch64 state only.

This feature requires FEAT_BRBE.

If FEAT_BRBE is implemented, an Armv9.3-compliant implementation includes the FEAT_BRBEv1p1 feature.

The ID_AA64DFR0_EL1.BRBE field identifies the presence of FEAT_BRBEv1p1.

For more information, see Chapter D15 The Branch Record Buffer Extension.
FEAT_RME, Realm Management Extension

FEAT_RME is one component of the Arm Confidential Compute Architecture (Arm CCA). Together with the other components of the Arm CCA, RME enables support for dynamic, attestable, and trusted execution environments (Realms) to be run on an Arm PE.

This feature is supported in AArch64 state only.
This feature is OPTIONAL in an Armv9.2 implementation.
FEAT_RME requires FEAT_RNG or FEAT_RNG_TRAP.
If the Trace Architecture is implemented, a PE that implements FEAT_RME also implements FEAT_ETEv1p2.
For more information, see Arm® Architecture Reference Manual Supplement, The Realm Management Extension (RME), for Armv9-A (ARM DDI 0615).

FEAT_SME, Scalable Matrix Extension

FEAT_SME introduces two AArch64 execution modes that can be enabled and disabled by application software:

- In ZA mode, scalable, two-dimensional, architectural ZA tile storage becomes available and instructions are defined to load, store, extract, insert, and clear rows and columns of the ZA tiles.
- In Streaming SVE mode, the Effective SVE vector length changes to match the Effective ZA tile width, support for a substantial subset of the SVE2 instruction set is available, and, when ZA mode is also enabled, instructions are defined that accumulate the matrix outer product of two SVE vectors into a ZA tile.

This feature is supported in AArch64 state only.
This feature is OPTIONAL from Armv9.2.
The ID_AA64PFR1_EL1.SME field identifies the presence of FEAT_SME.
For more information, see Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A (ARM DDI 0616).

FEAT_SME_FA64, Full Streaming SVE mode instructions

FEAT_SME_FA64 supports the full A64 instruction set in Streaming SVE mode.
This feature is supported in AArch64 state only.
This feature is OPTIONAL in an Armv9.2 implementation.
FEAT_SME_FA64 requires FEAT_SME.
The ID_AA64SMFR0_EL1.FA64 field identifies support for FEAT_SME_FA64.
For more information, see Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A (ARM DDI 0616).

FEAT_EBF16, AArch64 Extended BFloat16 instructions

FEAT_EBF16 supports the Extended BFloat16 mode.
This feature is supported in AArch64 state only.
This feature is OPTIONAL in an Armv9.2 implementation.
FEAT_EBF16 requires FEAT_SME.
The ID_AA64ISAR1_EL1.BF16 and ID_AA64ZFR0_EL1.BF16 fields identify the presence of FEAT_EBF16, and must return the same value.
For more information, see Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A (ARM DDI 0616).

FEAT_SME_F64F64, Double-precision floating-point outer product instructions

FEAT_SME_F64F64 indicates SME support for instructions that accumulate into FP64 double-precision floating-point elements in the ZA array.
This feature is supported in AArch64 state only.
This feature is OPTIONAL in an Armv9.2 implementation. FEAT_SME_F64F64 requires FEAT_SME.
The ID_AA64SMFR0_EL1.F64F64 field identifies support for FEAT_SME_F64F64. For more information, see Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A (ARM DDI 0616).

**FEAT_SME_I16I64, 16-bit to 64-bit integer widening outer product instructions**

FEAT_SME_I16I64 indicates SME support for instructions that accumulate into 64-bit integer elements in the ZA array. This feature is supported in AArch64 state only. This feature is OPTIONAL in an Armv9.2 implementation. FEAT_SME_I16I64 requires FEAT_SME.
The ID_AA64SMFR0_EL1.I16I64 field identifies support for FEAT_SME_I16I64. For more information, see Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A (ARM DDI 0616).

### A3.1.1 Architectural requirements within Armv9-A architecture

An Armv9 compliant implementation must also be Armv8.5 compliant.

An Armv9.1 compliant implementation must also be Armv8.6 and Armv9.0 compliant.

An Armv9.2 compliant implementation must also be Armv8.7 and Armv9.1 compliant.

An Armv9.3 compliant implementation must also be Armv8.8 and Armv9.2 compliant.
Part B
The AArch64 Application Level Architecture
Chapter B1
The AArch64 Application Level Programmers’ Model

This chapter describes the AArch64 application level programmers’ model. It contains the following sections:

- About the Application level programmers’ model on page B1-136.
- Registers in AArch64 Execution state on page B1-137.
- Software control features and EL0 on page B1-146.
B1.1 About the Application level programmers’ model

This chapter contains the programmers’ model information required for application development.

The information in this chapter is distinct from the system information required to service and support application execution under an operating system, or higher level of system software. However, some knowledge of the system information is needed to put the Application level programmers’ model into context.

Depending on the implementation choices, the architecture supports multiple levels of execution privilege, indicated by different Exception levels that number upwards from EL0 to EL3. EL0 corresponds to the lowest privilege level and is often described as unprivileged. The Application level programmers’ model is the programmers’ model for software executing at EL0. For more information, see Exception levels on page D1-4632.

System software determines the Exception level, and therefore the level of privilege, at which software runs. When an operating system supports execution at both EL1 and EL0, an application usually runs unprivileged at EL0. This:

- Permits the operating system to allocate system resources to an application in a unique or shared manner.
- Provides a degree of protection from other processes, and so helps protect the operating system from malfunctioning software.

This chapter indicates where some system level understanding is necessary, and where relevant it gives a reference to the system level description.

Execution at any Exception level above EL0 is often referred to as privileged execution.

For more information on the system level view of the architecture refer to Chapter D1 The AArch64 System Level Programmers’ Model.
B1.2 Registers in AArch64 Execution state

This section describes the registers and process state visible at EL0 when executing in the AArch64 state. It includes the following:

- Registers in AArch64 state
- Process state, PSTATE on page B1-142
- System registers on page B1-144

B1.2.1 Registers in AArch64 state

In the AArch64 application level view, an Arm processing element has:

**R0-R30**

31 general-purpose registers, R0 to R30. Each register can be accessed as:
- A 64-bit general-purpose register named X0 to X30.
- A 32-bit general-purpose register named W0 to W30.

See the register name mapping in Figure B1-1.

**Note**

In instruction encodings, the value 0b11111 (31) is used to indicate the ZR (zero register). This indicates that the argument takes the value zero, but does not indicate that the ZR is implemented as a physical register.

**SP**

A 64-bit dedicated Stack Pointer register. The least significant 32 bits of the stack pointer can be accessed using the register name WSP.

The use of SP as an operand in an instruction, indicates the use of the current stack pointer.

**Note**

Stack pointer alignment to a 16-byte boundary is configurable at EL1. For more information, see the Procedure Call Standard for the Arm 64-bit Architecture.

**PC**

A 64-bit Program Counter holding the address of the current instruction.

Software cannot write directly to the PC. It can be updated only on a branch, exception entry or exception return.

**Note**

Attempting to execute an A64 instruction that is not word-aligned generates a PC alignment fault, see PC alignment checking on page D1-4668.

**V0-V31**

32 SIMD&FP registers, V0 to V31. Each register can be accessed as:
- A 128-bit register named Q0 to Q31.
- A 64-bit register named D0 to D31.
- A 32-bit register named S0 to S31.
- A 16-bit register named H0 to H31.
- An 8-bit register named B0 to B31.
- A 128-bit vector of elements.
A 64-bit vector of elements.

Where the number of bits described by a register name does not occupy an entire SIMD&FP register, it refers to the least significant bits. See Figure B1-2.

**FPCR, FPSR** Two SIMD and floating-point control and status registers, FPCR and FPSR.

**Z0-Z31** 32 scalable vector registers, Z0 to Z31. Each register can be accessed as:

- A vector of elements with an IMPLEMENTATIONDEFINED maximum length of 128 to 2048 bits.
- A fixed-length 128-bit vector of elements named V0 to V31.
- A 128-bit register named Q0 to Q31.
- A 64-bit register named D0 to D31.
- A 32-bit register named S0 to S31.
- A 16-bit register named H0 to H31.
- An 8-bit register named B0 to B31.

**P0-P15** 16 SVE scalable predicate registers, P0 to P15. See SVE predicate registers on page B1-141.

**FFR** The dedicated SVE First Fault Register that has the same size and format as the predicate registers, P0-P15. See FFR, First Fault Register on page B1-142.

For more information on SVE registers, see SVE vector registers on page B1-140 and SVE writes to scalar registers on page B1-141.

For more information about data types and vector formats, see Supported data types on page A1-42.
Pseudocode description of registers in AArch64 state

In the pseudocode functions that access registers:

• The assignment form is used for register writes.
• The non-assignment for register reads.

The uses of the X[] function are:

• Reading or writing X0-X30, using n to index the required register.
• Reading the zero register ZR, accessed as X[31].

Note

The pseudocode use of X[31] to represent the zero register does not indicate that hardware must implement this register.

The AArch64 SP[] function is used to read or write the current SP.

The AArch64 PC[] function is used to read the PC.

The AArch64 V[] function is used to read or write the Advanced SIMD and floating-point registers V0-V31, using a parameter n to index the required register.

The AArch64 Vpart[] function is used to read or write a part of one of V0-V31, using a parameter n to index the required register, and a parameter part to indicate the required part of the register, see the function description for more information.

The AArch64 Z[] function is used to read or write the SVE scalable vector registers Z0-Z31, using a parameter n to index the required register.

The Z[], V[] and Vpart[] functions access the same underlying vector register file.

The X[], SP[], PC[], V[], Vpart[], and Z[] functions are defined in Chapter J1 Armv8 Pseudocode.
### SVE vector registers

SVE has 32 scalable vector registers named Z0-Z31.

All SVE scalable vector registers are the same size.

The size of an SVE scalable vector register is an IMPLEMENTATION DEFINED multiple of 128 bits.

The maximum size of an SVE scalable vector register is 2048 bits.

The minimum size of an SVE scalable vector register is 128 bits.

Unless stated otherwise in an instruction description, SVE instructions treat an SVE scalable vector register as containing one or more vector elements that are equal in size.

Unless stated otherwise in an instruction description, vector elements can be processed in parallel by SVE instructions.

When an SVE scalable vector register is divided into vector elements by an instruction, the size of the vector elements is encoded in the opcode of the instruction. The size of the vector elements is 8, 16, 32, 64, or 128 bits.

When the order of operations performed by an SVE instruction on vector or predicate elements has observable significance, elements are processed in increasing element number order.

The layouts of an SVE 256-bit vector register and a SIMD&FP vector in AArch64 state are:

![Figure B1-4 SVE vectors in AArch64 state](image-url)
Bits[127:0] of each of the SVE scalable vector registers, Z0-Z31, hold the correspondingly numbered AArch64 SIMD&FP register, V0-V31.

When the accessible SVE vector length at the current Exception level is greater than 128 bits, any AArch64 instruction that writes to V0-V31 sets all the accessible bits above bit [127] of the corresponding SVE scalable vector register to zero.

**SVE writes to scalar registers**

Certain SVE instructions generate a scalar result that is written to an AArch64 general-purpose register or to element[0] of a vector register.

When an SVE instruction generates a scalar result of width N bits, the instruction places the result in bits [N-1:0] of the destination register.

When an instruction generates a scalar result of width N bits, and N is less than the maximum accessible destination register width RW, the instruction sets bits [RW-1:N] of the destination register to zero.

**B1.2.3 SVE predicate registers**

SVE has 16 scalable predicate registers named P0-P15.

Each SVE predicate register holds one bit for each byte of a vector register.

The size of an SVE predicate register is an IMPLEMENTATION DEFINED multiple of 16 bits.

The maximum size of an SVE predicate register is 256 bits.

The minimum size of an SVE predicate register is 16 bits.

Unless stated otherwise in the instruction description, SVE instructions treat an SVE predicate register as containing one or more predicate elements of equal size.

Each predicate register can be subdivided into a number of 1-bit, 2-bit, 4-bit, or 8-bit elements.

Each predicate element in a predicate register corresponds to a vector element.

When a predicate register is divided into predicate elements by an instruction, the size of the predicate elements is encoded in the opcode of the instruction.

If the lowest-numbered bit of a predicate element is 1, the value of the predicate element is TRUE.

If the lowest-numbered bit of a predicate element is 0, the value of the predicate element is FALSE.

For all SVE instructions, if all of the following are true, all bits except the lowest-numbered bit of each predicate element are ignored on reads:

- The instructions are not used to move and permute predicate elements.
- The instructions are not predicate logical operations.

For all SVE instructions, if all of the following are true, all bits except the lowest-numbered bit of each predicate element are set to zero on writes:

- The instructions are not used to move and permute predicate elements.
- The instructions are not predicate logical operations.
B1.2.4 FFR, First Fault Register

SVE has a dedicated First Fault Register named FFR.

The FFR captures the cumulative fault status of a sequence of SVE First-fault and Non-fault vector load instructions.

The FFR and the predicate registers have the same size and format.

The FFR is a Special-purpose register.

All bits in the FFR that are accessible at the current Exception level are initialized to 1 by using the `SETFFR` instruction.

Bits in the FFR are indirectly set to 0 as a result of a suppressed access or fault generated in response to an Active element of an SVE First-fault or Non-fault vector load.

Bits in the FFR are never set to 1 as a result of a vector load instruction.

After a sequence of one or more SVE First-fault or Non-fault loads that follow a `SETFFR` instruction, the FFR contains a sequence of zero or more TRUE elements, followed by zero or more FALSE elements.

The TRUE elements in the FFR indicate the shortest sequence of consecutive elements that could contain valid data loaded from memory.

The only instructions that directly read the FFR are:
- `RDFFR` (predicated).
- `RDFFRS`.

The only instructions that directly write the FFR are:
- `WRFFR`.
- `SETFFR`.

All direct and indirect reads and writes to the FFR occur in program order relative to other instructions, without explicit synchronization.

B1.2.5 Process state, PSTATE

Process state or PSTATE is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

The following PSTATE information is accessible at EL0:

The Condition flags

Flag-setting instructions set these. They are:

N  Negative Condition flag. If the result of the instruction is regarded as a two's complement signed integer, the PE sets this to:
   • 1 if the result is negative.
   • 0 if the result is positive or zero.

Z  Zero Condition flag. Set to:
   • 1 if the result of the instruction is zero.
   • 0 otherwise.
   A result of zero often indicates an equal result from a comparison.

C  Carry Condition flag. Set to:
   • 1 if the instruction results in a carry condition, for example an unsigned overflow that is the result of an addition.
   • 0 otherwise.
V  Overflow Condition flag. Set to:
  • 1 if the instruction results in an overflow condition, for example a signed
    overflow that is the result of an addition.
  • 0 otherwise.

Conditional instructions test the N, Z, C and V Condition flags, combining them with the Condition
code for the instruction to determine whether the instruction must be executed. In this way,
execution of the instruction is conditional on the result of a previous operation. For more
information about conditional execution, see Condition flags and related instructions on
page C6-1167.

The exception masking bits

D  Debug exception mask bit. When EL0 is enabled to modify the mask bits, this bit is
    visible and can be modified. However, this bit is architecturally ignored at EL0.

A  SError interrupt mask bit.

I  IRQ interrupt mask bit.

F  FIQ interrupt mask bit.

For each bit, the values are:
0  Exception not masked.
1  Exception masked.

Access at EL0 using AArch64 state depends on SCTLR_EL1.UMA.

See Process state, PSTATE on page D1-4670 for the system level view of PSTATE.

Accessing PSTATE fields at EL0

At EL0 using AArch64 state, PSTATE fields can be accessed using Special-purpose registers that can be directly
read using the MRS instruction and directly written using the MSR (register) instructions. Table B1-1 shows the
Special-purpose registers that access the PSTATE fields that hold AArch64 state when the PE is at EL0 using
AArch64. All other PSTATE fields do not have direct read and write access at EL0.

Software can also use the MSR (immediate) instruction to directly write to PSTATE. {D, A, I, F}. Table B1-2 shows
the MSR (immediate) operands that can directly write to PSTATE. {D, A, I, F} when the PE is at EL0 using AArch64
state.

However, access to the PSTATE. {D, A, I, F} fields at EL0 using AArch64 state depends on SCTLR_EL1.UMA.

Writers to the PSTATE fields have side-effects on various aspects of the PE operation. All of these side-effects, are
guaranteed:
• Not to be visible to earlier instructions in the execution stream.
• To be visible to later instructions in the execution stream.
SVE use of PSTATE N, Z, C, and V Condition flags

This section describes the SVE-specific use of PSTATE.

PSTATE N, Z, C and V condition flags can be updated by any of the following:

- An SVE instruction that generates a predicate result and updates the PSTATE N, Z, C and V Condition flags based on the value of the result.
- An SVE instruction that updates the PSTATE N, Z, C and V Condition flags based on the value in its predicate source register or FFR:
  - PTEST.
  - RDFFR (predicated).
- An SVE instruction that updates the PSTATE N, Z, C and V Condition flags based on the values in its general-purpose source registers:
  - CTERMEQ, CTERMNE.

When setting the PSTATE N, Z, C and V Condition flags for SVE predicated flag-setting instructions, the instruction's Governing predicate determines which predicate elements are considered Active.

When setting the PSTATE N, Z, C and V Condition flags for SVE unpredicated flag-setting instructions, all predicate elements are considered Active.

Unless otherwise specified in an instruction description, the SVE flag-setting instructions update the PSTATE N, Z, C and V Condition flags as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>SVE Name</th>
<th>SVE interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>First</td>
<td>Set to 1 if the First active element was TRUE, otherwise cleared to 0.</td>
</tr>
<tr>
<td>Z</td>
<td>None</td>
<td>Cleared to 0 if any Active element was TRUE, otherwise set to 1.</td>
</tr>
<tr>
<td>C</td>
<td>Not last</td>
<td>Cleared to 0 if the Last active element was TRUE, otherwise set to 1.</td>
</tr>
<tr>
<td>V</td>
<td>-</td>
<td>Cleared to 0.</td>
</tr>
</tbody>
</table>

B1.2.6 System registers

System registers provide support for execution control, status and general system configuration. The majority of the System registers are not accessible at EL0.

However, some System registers can be configured to allow access from software executing at EL0. Any access from EL0 to a System register with the access right disabled causes the instruction to behave as UNDEFINED. The registers that can be accessed from EL0 are:

- **Cache ID registers**: The CTR_EL0 and DCZID_EL0 registers provide implementation parameters for EL0 cache management support.
- **Debug registers**: A Debug Communications Channel is supported by the MDCCSR_EL0, DBGDTR_EL0, DBGDTRRX_EL0 and DBGDTRTX_EL0 registers.
- **Performance Monitors registers**: The Performance Monitors Extension provides counters and configuration registers. Software executing at EL1 or a higher Exception level can configure some of these registers to be accessible at EL0. For more details, see Chapter D11 *The Performance Monitors Extension*.
- **Activity Monitors registers**: The Activity Monitors Extension provides counters and configuration registers. Software executing at EL1 or a higher Exception level can configure these registers to be accessible at EL0.
For more details, see Chapter D12 *The Activity Monitors Extension*.

**Thread ID registers**
The TPIDR_EL0 and TPIDRRO_EL0 registers are two thread ID registers with different access rights.

**Timer registers**
The following operations are performed by these registers:

- Read access to the system counter clock frequency using CNTFRQ_EL0.
- Physical and virtual timer count registers, CNTPCT_EL0 and CNTVCT_EL0.
- Physical up-count comparison, down-count value and timer control registers, CNTP_CVAL_EL0, CNTP_TVAL_EL0, and CNTP_CTL_EL0.
- Virtual up-count comparison, down-count value and timer control registers, CNTV_CVAL_EL0, CNTV_TVAL_EL0, and CNTV_CTL_EL0.
B1.3 Software control features and EL0

The following sections describe the EL0 view of the software control features:

- Exception handling
- Wait for Interrupt and Wait for Event
- The YIELD instruction
- Application level cache management on page B1-147
- Instructions relating to Debug on page B1-147
- About PSTATE.DIT on page B1-123

B1.3.1 Exception handling

In the Arm architecture, an exception causes a change of program flow. Execution of an exception handler starts, at an Exception level higher than EL0, from a defined vector that relates to the exception taken.

Exceptions include:
- Interrupts.
- Memory system aborts.
- Exceptions generated by attempting to execute an instruction that is UNDEFINED.
- System calls.
- Secure monitor or Hypervisor traps.
- Debug exceptions.

Most details of exception handling are not visible to application level software, and are described in Chapter D1 The AArch64 System Level Programmers' Model.

The SVC instruction causes a Supervisor Call exception. This provides a mechanism for unprivileged software to make a system call to an operating system.

The BRK instruction generates a Breakpoint Instruction exception. This provides a mechanism for debugging software using debugger executing on the same PE, see Breakpoint Instruction exceptions on page D2-4705.

--- Note ---

The BRK instruction is supported only in the A64 instruction set. The equivalent instruction in the T32 and A32 instruction sets is BKPT.

B1.3.2 Wait for Interrupt and Wait for Event

Issuing a WFI instruction indicates that no further execution is required until a WFI wake-up event occurs, see Wait for Interrupt mechanism on page D1-4678. This permits entry to a low-power state.

Issuing a WFE instruction indicates that no further execution is required until a WFE wake-up event occurs, see Wait for Event on page D1-4676. This permits entry to a low-power state.

B1.3.3 The YIELD instruction

The YIELD instruction provides a hint that the task performed by a thread is of low importance so that it could yield, see YIELD on page C6-2037. This mechanism can be used to improve overall performance in a Symmetric Multithreading (SMT) or Symmetric Multiprocessing (SMP) system.

Examples of when the YIELD instruction might be used include a thread that is sitting in a spin-lock, or where the arbitration priority of the snoop bit in an SMP system is modified. The YIELD instruction permits binary compatibility between SMT and SMP systems.

The YIELD instruction is a NOP hint instruction.
The YIELD instruction has no effect in a single-threaded system, but developers of such systems can use the instruction to flag its intended use for future migration to a multiprocessor or multithreading system. Operating systems can use YIELD in places where a yield hint is wanted, knowing that it will be treated as a NOP if there is no implementation benefit.

### B1.3.4 Application level cache management

A small number of cache management instructions can be enabled at EL0 from higher levels of privilege using the SCTLR_EL1 System register. Any access from EL0 to an operation with the access right disabled causes the instruction to behave as UNDEFINED.

About the available operations, see Application level access to functionality related to caches on page B2-185.

### B1.3.5 Instructions relating to Debug

Exception handling on page B1-146 refers to the BRK instruction, which generates a Breakpoint Instruction exception. In addition, in both AArch64 state and AArch32 state, the HLT instruction causes the PE to halt execution and enter Debug state. This provides a mechanism for debugging software using a debugger that is external to the PE, see Chapter H1 About External Debug.

---

**Note**

In AArch32 state, previous versions of the architecture defined the DBG instruction, which could provide a hint to the debug system. This instruction executes as a NOP. Arm deprecates the use of the DBG instruction.

### B1.3.6 About PSTATE.DIT

When the value of PSTATE.DIT is 1:

- The instructions listed in DIT are required to have:
  - Timing which is independent of the values of the data supplied in any of its registers, and the values of the NZCV flags.
  - Responses to asynchronous exceptions which do not vary based on the values supplied in any of their registers, or the values of the NZCV flags.

- All loads and stores must have their timing insensitive to the value of the data being loaded or stored.

---

**Note**

- The use of value prediction for load data values when PSTATE.DIT is set, is not compatible with the requirement that the timing is insensitive to the data value being loaded.

- Arm recommends that the FEAT_PAuth instructions do not have their timing dependent on the key value used in the pointer authentication, regardless of the PSTATE.DIT bit.

- When the value of PSTATE.DIT is 0, the architecture makes no statement about the timing properties of any instructions. However, it is likely that these instructions have timing that is invariant of the data in many situations.

- If SVE2 is not implemented, the data independent timing control introduced by FEAT_DIT does not affect the timing properties of SVE instructions.

- The Operational information section of an SVE or an SVE2 instruction description indicates whether or not that instruction honors the PSTATE.DIT control. If the Operational information section of an SVE instruction description does not mention PSTATE.DIT or if the section does not exist, then the instruction timing is not affected by PSTATE.DIT.

- For SVE and SVE2 predicated instructions, it is the programmer’s responsibility to use a Governing predicate that does not reflect the values of the data being operated on.

---

A corresponding DIT bit is added to PSTATE in AArch64 state, and to CPSR in AArch32 state.
On an exception that is taken from AArch64 state to AArch64 state, PSTATE.DIT is copied to SPSR_ELx.DIT.

On an exception that is taken from AArch32 state to AArch64 state, CPSR.DIT is copied to SPSR_ELx.DIT.

On an exception return from AArch64 state:
- SPSR_ELx.DIT is copied to PSTATE.DIT, when the target Exception level is in AArch64 state.
- SPSR_ELx.DIT is copied to CPSR.DIT, when the target Exception level is in AArch32 state.

PSTATE.DIT can be written and read at all Exception levels.

--- Note ---
- PSTATE.DIT is unchanged on entry into Debug state.
- PSTATE.DIT is not guaranteed to have any effect in Debug state.
B1.4 SVE predicated instructions

IVKHDR If an instruction supports predication, it is known as a predicated instruction.

IFNFRN The predicate operand that is used to determine the Active elements of a predicated instruction is known as the Governing predicate.

ISVYXB An instruction that does not have a Governing predicate operand and implicitly treats all other vector and predicate elements as Active is known as an unpredicated instruction.

IKNKBN Many predicated instructions can only use P0-P7 as the Governing predicate.

ILZVFFJ When a Governing predicate element is TRUE, the corresponding element in other vector or predicate operands is an Active element.

ICNFLG When a Governing predicate element is FALSE, the corresponding element in other vector or predicate operands is an Inactive element.

ICBYYJH Predicated instructions process Active elements.

ILDXSF Predicated instructions do not process Inactive elements.

IGILPZ Unpredicated instructions process all elements in their vector or predicate operands.

ILWLQBD When a predicated instruction writes to a vector destination register or a predicate destination register, one of the following happens:

- The Inactive elements in the destination register are set to zero.
- The Inactive elements in the destination register retain their previous value.

IQBHRN Zeroing predication is performed when the Inactive elements in the destination register are set to zero.

IFYRF Merging predication is performed when Inactive elements in the destination register retain their previous value.
B1.4 SVE predicated instructions
Chapter B2  
The AArch64 Application Level Memory Model

This chapter gives an application level view of the memory model. It contains the following sections:

- About the Arm memory model on page B2-152.
- Definition of the Arm memory model on page B2-159.
- Caches and memory hierarchy on page B2-184.
- Alignment support on page B2-189.
- Endian support on page B2-192.
- Memory types and attributes on page B2-196.
- Synchronization and semaphores on page B2-211.

--- Note ---

In this chapter, System register names usually link to the description of the register in Chapter D17 AArch64 System Register Descriptions, for example. SCTLR_EL1.
B2.1 About the Arm memory model

The Arm architecture is a weakly ordered memory architecture that permits the observation and completion of memory accesses in a different order from the program order. The following sections of this chapter provide the complete definition of the memory model, this introduction is not intended to contradict the definition found in those sections. In general, the basic principles of the memory model are:

- To provide a memory model that has similar weaknesses to those found in the memory models used by high-level programming languages such as C or Java. For example, by permitting independent memory accesses to be reordered as seen by other observers.
- To avoid the requirement for multi-copy atomicity in the majority of memory types.
- The provision of instructions and memory barriers to compensate for the lack of multi-copy atomicity in the cases where it would be needed.
- The use of address, data, and control dependencies in the creation of order so as to avoid having excessive numbers of barriers or other explicit instructions in common situations where some order is required by the programmer or the compiler.
- If FEAT_MTE2 is implemented, the definitions of the memory model which apply to data accesses and data apply to Allocation Tag accesses and Allocation tags.

This section contains:
- Address space.
- Memory type overview.

B2.1.1 Address space

Address calculations are performed using 64-bit registers. However, supervisory software can configure the top eight address bits for use as a tag, as described in Address tagging on page D8-5162. If this is done, address bits[63:56]:

- Are not considered when determining whether the address is valid.
- Are never propagated to the program counter.

Supervisory software determines the valid address range. Attempting to access an address that is not valid generates an MMU fault.

Simple sequential execution of instructions might overflow the valid address range. For more information, see Virtual address space overflow on page D7-5041.

Memory accesses use the Mem[] function. This function makes an access of the required type. If supervisory software configures the top eight address bits for use as a tag, the top eight address bits are ignored.

The AccType{} enumeration defines the different access types.

Note

- Chapter D7 The AArch64 System Level Memory Model and Chapter D8 The AArch64 Virtual Memory System Architecture include descriptions of memory system features that are transparent to the application, including memory access, address translation, memory maintenance instructions, and alignment checking and the associated fault handling. These chapters also include pseudocode descriptions of these operations.
- For information on the pseudocode that relates to memory accesses, see Basic memory access on page D7-5075, Unaligned memory access on page D7-5075, and Aligned memory access on page D7-5075.

B2.1.2 Memory type overview

The Arm architecture provides the following mutually-exclusive memory types:

Normal This is generally used for bulk memory operations, both read/write and read-only operations.
The Arm architecture forbids *Speculative* reads of any type of Device memory. This means Device memory types are suitable attributes for read-sensitive Locations.

Locations of the memory map that are assigned to peripherals are usually assigned the Device memory attribute.

Device memory has additional attributes that have the following effects:

- They prevent aggregation of reads and writes, maintaining the number and size of the specified memory accesses. See *Gathering* on page B2-203.
- They preserve the access order and synchronization requirements for accesses to a single peripheral. See *Reordering* on page B2-204.
- They indicate whether a write can be acknowledged other than at the end point. See *Early Write Acknowledgement* on page B2-205.

For more information on Normal memory and Device memory, see *Memory types and attributes* on page B2-196.

---

**Note**

Earlier versions of the Arm architecture defined a single Device memory type and a Strongly-ordered memory type. A Note in *Device memory* on page B2-200 describes how these memory types map onto the Armv8 memory types.

---

**B2.1.3 SVE memory model**

SVE predicated memory operations have a vector element size and a memory element access size. The vector element size specifies the data that is read from and written to the vector. The memory element access size specifies the amount of data that is read from and written to the memory.

The vector element size and the memory element access size do not need to have the same value.

For each memory element, there is an associated element address.

SVE also affects behavior in the following areas:

- *SVE memory ordering relaxations* on page B2-170.
- *Load or Store of Single or Multiple registers* on page B2-189.
- *SVE loads and stores that access Device memory* on page B2-206.
B2.2 Atomicity in the Arm architecture

Atomicity is a feature of memory accesses, described as atomic accesses. The Arm architecture description refers to two types of atomicity, single-copy atomicity and multi-copy atomicity. In the Arm architecture, the atomicity requirements for memory accesses depend on the memory type, and whether the access is explicit or implicit. For more information, see:

- Requirements for single-copy atomicity.
- Properties of single-copy atomic accesses on page B2-156.
- Multi-copy atomicity on page B2-156.
- Requirements for multi-copy atomicity on page B2-156.
- Concurrent modification and execution of instructions on page B2-156.

For more information about the memory types, see Memory type overview on page B2-152.

B2.2.1 Requirements for single-copy atomicity

For explicit memory effects generated from an Exception level the following rules apply:

- A read that is generated by a load instruction that loads a single general-purpose register and is aligned to the size of the read in the instruction is single-copy atomic.
- A write that is generated by a store instruction that stores a single general-purpose register and is aligned to the size of the write in the instruction is single-copy atomic.
- Reads that are generated by a Load Pair instruction that loads two general-purpose registers and are aligned to the size of the load to each register are treated as two single-copy atomic reads, one for each register being loaded.
- Writes that are generated by a Store pair instruction that stores two general-purpose registers and are aligned to the size of the store of each register are treated as two single-copy atomic writes, one for each register being stored.
- Load-Exclusive Pair instructions of two 32-bit quantities and Store-Exclusive Pair instructions of 32-bit quantities are single-copy atomic.
- When the Store-Exclusive of a Load-Exclusive/Store-Exclusive pair instruction using two 64-bit quantities succeeds, it causes a single-copy atomic update of the entire memory location being updated.

Note

To atomically load two 64-bit quantities, perform a Load-Exclusive pair/Store-Exclusive pair sequence of reading and writing the same value for which the Store-Exclusive pair succeeds, and use the read values from the Load-Exclusive pair.

- Where translation table walks generate a read of a translation table entry, this read is single-copy atomic.
- For the atomicity of instruction fetches, see Concurrent modification and execution of instructions on page B2-156.
- Reads to SIMD and floating-point registers of a single 64-bit or smaller quantity that is aligned to the size of the quantity being loaded are treated as single-copy atomic reads.
- Writes from SIMD and floating-point registers of a single 64-bit or smaller quantity that is aligned to the size of the quantity being stored are treated as single-copy atomic writes.
- Element or Structure Reads to SIMD and floating-point registers of 64-bit or smaller elements, where each element is aligned to the size of the element being loaded, have each element treated as a single-copy atomic read.
- Element or Structure Writes from SIMD and floating-point registers of 64-bit or smaller elements, where each element is aligned to the size of the element being stored, have each element treated as a single-copy atomic store.
- Reads to SIMD and floating-point registers of a 128-bit value that is 64-bit aligned in memory are treated as a pair of single-copy atomic 64-bit reads.
- Writes from SIMD and floating-point registers of a 128-bit value that is 64-bit aligned in memory are treated as a pair of single-copy atomic 64-bit writes.
- Atomicity rules for SIMD load and store instructions also apply to SVE load and store instructions.
- SVE predicated load and store instructions are performed as a sequence of memory element accesses.
• If an SVE predicated load or store instruction uses an element address that is aligned to the specified memory element access size, the related element memory access is performed as a single-copy atomic access.
• SVE unpredicated load and store instructions are performed as a sequence of byte accesses.
• SVE unpredicated load and store instructions do not guarantee that any access larger than a byte will be performed as a single-copy atomic access.
• When FEAT_LS64 is implemented, a single-copy atomic load of a 64-byte value that is 64-byte aligned in memory is treated as an atomic 64-byte read from the target address.
• When FEAT_LS64 is implemented, a single-copy atomic store of a 64-byte value that is 64-byte aligned in memory is treated as an atomic 64-byte write to the target address.
• For unaligned memory accesses, the single-copy atomicity is described in Alignment of data accesses on page B2-189.
• The reads and writes of the two words or two double-words accessed by CASP instructions are single-copy atomic at the size of the two words or double-words.

All other memory accesses are regarded as streams of accesses to bytes, and no atomicity between accesses to different bytes is ensured by the architecture.

All accesses to any byte are single-copy atomic.

--- Note ---
In AArch64 state, no memory accesses from a DC ZVA have single-copy atomicity of any quantity greater than individual bytes.

---

If, according to these rules, an instruction is executed as a sequence of accesses, exceptions, including interrupts, can be taken during that sequence, regardless of the memory type being accessed. If any of these exceptions are returned from using their preferred return address, the instruction that generated the sequence of accesses is re-executed, and so any access performed before the exception was taken is repeated. See also Taking an interrupt during a multi-access load or store on page D1-4664.

--- Note ---
The exception behavior for these multiple access instructions means that they are not suitable for use for writes to memory for the purpose of software synchronization.

---

Changes to single-copy atomicity in Armv8.4

In addition to the single-copy atomicity requirements listed above:

Instructions that are introduced in FEAT_LRCPC are single-copy atomic when all of the following conditions are true:
• All bytes being accessed are within the same 16-byte quantity aligned to 16 bytes.
• Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

If FEAT_LSE2 is implemented, all loads and stores are single-copy atomic when all of the following conditions are true:
• Accesses are unaligned to their data size but all bytes being accessed are within a 16-byte quantity that is aligned to 16 bytes.
• Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

If FEAT_LSE2 is implemented, LDP, LDNP, and STP instructions that load or store two 64-bit registers are single-copy atomic when all of the following conditions are true:
• The overall memory access is aligned to 16 bytes.
• Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.

If FEAT_LSE2 is implemented, LDP, LDNP, and STP instructions that access fewer than 16 bytes are single-copy atomic when all of the following conditions are true:
• All bytes being accessed are within a 16-byte quantity aligned to 16 bytes.
• Accesses are to Inner Write-Back, Outer Write-Back Normal cacheable memory.
B2.2.2 Properties of single-copy atomic accesses

A memory access instruction that is single-copy atomic has the following properties:

1. For a pair of overlapping single-copy atomic store instructions, all of the overlapping writes generated by one of the stores are Coherence-after the corresponding overlapping writes generated by the other store.

2. For a single-copy atomic load instruction $L_1$ that overlaps a single-copy atomic store instruction $S_2$, if one of the overlapping reads generated by $L_1$ Reads-from one of the overlapping writes generated by $S_2$, then none of the overlapping writes generated by $S_2$ are Coherence-after the corresponding overlapping reads generated by $L_1$.

For more information, see Definition of the Arm memory model on page B2-159.

B2.2.3 Multi-copy atomicity

In a multiprocessing system, writes to a memory location are multi-copy atomic if the following conditions are both true:

- All writes to the same location are serialized, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes.
- A read of a location does not return the value of a write until all observers observe that write.

Note
Writes that are not coherent are not multi-copy atomic.

B2.2.4 Requirements for multi-copy atomicity

For Normal memory, writes are not required to be multi-copy atomic.

For Device memory, writes are not required to be multi-copy atomic.

The Arm memory model is Other-multi-copy atomic. For more information, see External ordering constraints on page B2-166.

B2.2.5 Concurrent modification and execution of instructions

The Arm architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization.

Concurrent modification and execution of instructions can lead to the resulting instruction performing any behavior that can be achieved by executing any sequence of instructions that can be executed from the same Exception level, except where each of the instruction before modification and the instruction after modification is one of a B, B.cond, BL, BRK, CBNZ, CBZ, HVIC, ISB, NOP, SMC, SVC, TBNZ or TBZ instruction.

For the B, B.cond, BL, BRK, CBNZ, CBZ, HVIC, ISB, NOP, SMC, SVC, TBNZ and TBZ instructions, the architecture guarantees that after modification of the instruction, behavior is consistent with execution of either:

- The instruction originally fetched.
- A fetch of the modified instruction.

For all other instructions, to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior, instruction modifications must be explicitly synchronized before they are executed. The required synchronization is as follows:

1. No PE must be executing an instruction when another PE is modifying that instruction.

2. To ensure that the modified instructions are observable, a PE that is writing the instructions must issue the following sequence of instructions and operations:

   ; Coherency example for data and instruction accesses within the same Inner Shareable domain.
   ; enter this code with <Wt> containing a new 32-bit instruction,
   ; to be held in Cacheable space at a location pointed to by Xn.
   STR Wt, [Xn]
The AArch64 Application Level Memory Model
B2.2 Atomicity in the Arm architecture

DC CVAU, Xn ; Clean data cache by VA to point of unification (PoU)
DSB ISH ; Ensure visibility of the data cleaned from cache
IC IVAU, Xn ; Invalidate instruction cache by VA to PoU
DSB ISH

Note
• The DC CVAU operation is not required if the area of memory is either Non-cacheable or Write-Through Cacheable.
• If the contents of physical memory differ between the mappings, changing the mapping of VAs to PAs can cause the instructions to be concurrently modified by one PE and executed by another PE. If the modifications affect instructions other than those listed as being acceptable for modification, synchronization must be used to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.
• In a multiprocessor system, the DC CVAU and IC IVAU are broadcast to all PEs within the Inner Shareable domain of the PE running this sequence.

3. When the modified instructions are observable, each PE that is executing the modified instructions must execute an ISB or perform a context synchronizing event to ensure execution of the modified instructions:
   ISB ; Synchronize fetched instruction stream

For more information about the required synchronization operation, see Synchronization and coherency issues between data and instruction accesses on page B2-187.

For information about memory accesses caused by instruction fetches, see Ordering relations on page B2-163.

B2.2.6 Possible implementation restrictions on using atomic instructions

In some implementations, and for some memory types, the properties of atomicity can be met only by functionality outside the PE. Some system implementations might not support atomic instructions for all regions of the memory. In particular, this can apply to:
• Any type of memory in the system that does not support hardware cache coherency.
• Device, Non-cacheable memory, or memory that is treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such implementations, it is defined by the system:
• Whether the atomic instructions are atomic in regard to other agents that access memory.
• If the atomic instructions are atomic in regard to other agents that access memory, which address ranges or memory types this applies to.

An implementation can choose which memory type is treated as Non-cacheable.

The memory types for which it is architecturally guaranteed that the atomic instructions will be atomic are:
• Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.
• Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.

The architecture only requires that Conventional memory that is mapped in this way supports this functionality.

If the atomic instructions are not atomic in regard to other agents that access memory, then performing an atomic instruction to such a location can have one or more of the following effects:
• The instruction generates a synchronous External abort.
• The instruction generates a System Error interrupt.
• The instruction generates an IMPLEMENTATION DEFINED MMU fault reported using the Data Abort Fault status code of ESR_ELx.DFSC = 110101.
For the EL1&0 translation regime, if the atomic instruction is not supported because of the memory type that is defined in the first stage of translation, or the second stage of translation is not enabled, then this exception is a first stage abort and is taken to EL1. Otherwise, the exception is a second stage abort and is taken to EL2.

- The instruction is treated as a `nop`.
- The instructions are performed, but there is no guarantee that the memory accesses were performed atomically in regard to other agents that access memory. In this case, the instruction might also generate a System Error interrupt.
B2.3 Definition of the Arm memory model

This section describes observation and ordering in the Arm memory model. It contains the following subsections:

- **Basic definitions**
- **Dependency definitions** on page B2-162.
- **Ordering relations** on page B2-163.
- **Ordering constraints** on page B2-165.
- **Internal visibility requirement** on page B2-166.
- **External ordering constraints** on page B2-166.
- **Completion and endpoint ordering** on page B2-168.
- **Ordering of instruction fetches** on page B2-170.
- **Memory barriers** on page B2-174.
- **Limited ordering regions** on page B2-182.

For more information about endpoint ordering of memory accesses, see *Reordering* on page B2-204.

In the Arm memory model, the Shareability memory attribute indicates the degree to which hardware must ensure memory coherency between a set of observers, see *Memory types and attributes* on page B2-196.

The Arm architecture defines additional memory attributes and associated behaviors, which are defined in the system level section of this manual. See:
- **Chapter D7 The AArch64 System Level Memory Model**.
- **Chapter D8 The AArch64 Virtual Memory System Architecture**.

See also *Mismatched memory attributes* on page B2-208.

### B2.3.1 Basic definitions

The Arm memory model provides a set of definitions that are used to construct conditions on the permitted sequences of accesses to memory.

**Observer**

An *Observer* refers to a processing element or mechanism in the system, such as a peripheral device, that can generate reads from, or writes to, memory.

**Common Shareability Domain**

For the purpose of this section, all *Observers* are assumed to belong to a Common Shareability Domain. All read and write effects access only Normal memory locations in a Common Shareability Domain, and excludes the situations described in *Mismatched memory attributes* on page B2-208.

**Location**

A *Location* is a byte that is associated with an address in the physical address space.

--- **Note** ---

It is expected that an operating system will present the illusion to the application programmer that is consistent with a location also being considered as a byte that is associated with an address in the virtual address space.

---

**Effects**

The *Effects* of an instruction can be:

- Register effects.
- Memory effects.
- Barrier effects.
- Tag effects.
- Branching effects.
The effects of an instruction $I_1$ are said to appear in program order before the effects of an instruction $I_2$ if and only if $I_1$ occurs before $I_2$ in the order specified by the program. Each effect generated by an instruction has a unique identifier, which characterizes it amongst the events generated by the same instruction.

**Register effect**

The *Register effects* of an instruction are register reads or register writes of that instruction. For an instruction that accesses registers, a register read effect is generated for each register read by the instruction and a register write effect is generated for each register written by the instruction. An instruction may generate both read and write Register effects.

**Memory effect**

The *Memory effects* of an instruction are the memory reads or writes generated by that instruction. For an instruction that accesses memory, a memory read effect is generated for each Location read by the instruction and a memory write effect is generated for each Location written by the instruction. An instruction may generate both read and write Memory effects.

**Tag effect**

The *Tag effects* of a Memory Tagging instruction are the memory read or write effects of that instruction that affect tag locations.

**Tag-read**

A *Tag-read* is a read of a tag location generated by an $LDG$ instruction.

**Tag-write**

A *Tag-write* is a write of a tag location generated by an $STG$ instruction.

**Tag-Check-read**

A *Tag-Check-read* is a read of a tag location that is generated by a checked memory access. All other reads and writes are considered Data accesses.

**Branching effect**

The *Branching effects* of an instruction are effects which correspond to a branching decision being taken.

Note

Conditional and compare-and-swap instructions do not create Branching effects.

**Intrinsic order**

There is a per-instruction *Intrinsic order* relation that provides a partial order over the effects of that instruction, according to the operation of that instruction.

The operation of an instruction is defined by the pseudocode in Chapter C6 *A64 Base Instruction Descriptions*.

**Reads-from-register**

The *Reads-from-register* relation couples register read and write effects to the same register such that each register read effect is paired with exactly one register write effect in the execution of a program. A register read effect $R_2$ Reads-from-register a register write effect $W_1$ to the same register if and only if $R_2$ takes its data from $W_1$. By construction $W_1$ must be in program order before $R_2$ and there must be no intervening write to the same register in program order between $W_1$ and $R_2$.

**Reads-from**

The *Reads-from* relation couples memory read and write effects to the same Location such that each memory read effect is paired with exactly one memory write effect in the execution of a program. A memory read effect $R_2$ from a Location Reads-from a memory write effect $W_1$ to the same Location if and only if $R_2$ takes its data from $W_1$.
Coherence order
There is a per-location Coherence order relation that provides a total order over all memory write effects from all coherent Observers to that Location, starting with a notional memory write effect of the initial value. The Coherence order of a Location represents the order in which memory write effects to the Location arrive at memory.

Local read successor
A memory read effect R2 of a Location is the Local read successor of a memory write effect W1 from the same Observer to the same Location if and only if W1 appears in program order before R2 and there is not a memory write effect W3 from the same Observer to the same Location appearing in program order between W1 and R2.

Local write successor
A memory write effect W2 of a Location is a Local write successor of a memory read or write effect RW1 from the same Observer to the same Location if and only if RW1 appears in program order before W2.

Coherence-after
A memory write effect W2 to a Location is Coherence-after another memory write effect W1 to the same Location if and only if W2 is sequenced after W1 in the Coherence order of the Location.
A memory write effect W2 to a Location is Coherence-after a memory read effect R1 of the same location if and only if R1 Reads-from a memory write effect W3 to the same Location and W2 is Coherence-after W3.

Observed-by
A memory read or write effect RW1 from an Observer is Observed-by a memory write effect W2 from a different Observer if and only if W2 is coherence-after RW1.
A memory write effect W1 from an Observer is Observed-by a memory read effect R2 from a different Observer if and only if R2 Reads-from W1.

Note
The Observed-by relation relates only Memory effects generated by different Observers.

Overlapping accesses
Two Memory effects overlap if and only if they access the same Location. Two instructions overlap if and only if one or more of their generated Memory effects overlap.

Single-copy-atomic-ordered-before
A memory read effect R1 is Single-copy-atomic-ordered-before another memory read effect R2 if and only if all of the following statements are true:
• R1 and R2 are memory read effects generated by the same instruction.
• R1 is not a Local read successor of a memory write effect.
• R2 is a Local read successor of a memory write effect.

DMB FULL
A DMB FULL is a DMB with neither the LD or the ST qualifier.
Where this section refers to DMB without any qualification, then it is referring to all types of DMB. Unless a specific shareability domain is defined, a DMB applies to the Common Shareability Domain.
All properties that apply to DMB also apply to the corresponding DSB.

Context synchronization instruction
A Context synchronization instruction is one of the following:
• An ISB instruction.
• An instruction that generates a synchronous exception.
B2.3 Definition of the Arm memory model

B2.3.2 Dependency definitions

Dependency through registers

A Dependency through registers from a first effect E1 to a second effect E2 exists within a PE if and only if at least one of the following applies:

- E1 is a register write effect W1 which has not been generated by a Store Exclusive, E2 is a register read effect R2 and R2 Reads-from-register W1.
- E1 and E2 have been generated by the same instruction and E1 is before E2 in the Intrinsic order of that instruction.
- There is a Dependency through registers from E1 to a third effect E3, and there is a Dependency through registers from E3 to E2.

Address dependency

An Address dependency from a memory read effect R1 to a Memory effect RW2 exists if and only if there is a Dependency through registers from R1 to a Register effect E3 generated by RW2, and E3 affects the address part of RW2, and either:

- RW2 is a memory write effect W2.
- RW2 is a memory read effect R2 and there is no Branching effect D4 such that there is a Dependency through registers from R1 to D4 and from D4 to R2.

Note

An Address dependency exists from a memory read effect R1 to a Tag-Check-read R2 if and only if there is a Dependency through registers from R1 to the address part of R2.

Data dependency

A Data dependency from a memory read effect R1 to a memory write effect W2 exists if and only if there is a Dependency through registers from R1 to a Register effect E3 generated by W2, and E3 affects the data part of W2.

Control dependency

A Control dependency from a memory read effect R1 to a Memory effect RW2 exists if and only if either:

- There is a Dependency through registers from R1 to a Branching effect B3 and B3 is in program order before RW2.
- There is a Dependency through registers from R1 to the determination of a synchronous exception on an instruction generating an effect RW3, and RW2 appears in program order after RW3.

Note

This notion is under review. Arm’s intent is that a branch instruction between a read and a write, where the branch condition is dependent on the read, will provide order, regardless of whether the branch is taken. This applies only to branch instructions and not to conditional selection or other conditional data processing instructions. A formal definition of this change will be issued soon as an erratum to the Armv8 Architecture Reference Manual.

Pick Basic dependency

A Pick Basic dependency from a read Register effect or read Memory effect R1 to a Register effect or Memory effect E2 exists if and only if one of the following applies:

- There is a Dependency through registers from R1 to E2.
• There is an Intrinsic Control dependency from \( R_1 \) to \( E_2 \).
• There is a Pick Basic dependency from \( R_1 \) to an Effect \( E_3 \) and there is a Pick Basic dependency from \( E_3 \) to \( E_2 \).

### Pick Address dependency

A Pick Address dependency from a read Register effect or read Memory effect \( R_1 \) to a read or write Memory effect \( RW_2 \) exists if and only if all of the following apply:

• There is a Pick Basic dependency from \( R_1 \) to a Register effect \( E_3 \).
• There is an Intrinsic data dependency from the Register effect \( E_3 \) to \( RW_2 \).
• The Register effect \( E_3 \) affects the address of the location accessed by \( RW_2 \).

### Pick Data dependency

A Pick Data dependency from a read Register effect or read Memory effect \( R_1 \) to a write Memory effect \( W_2 \) exists if and only if all of the following apply:

• There is a Pick Basic dependency from \( R_1 \) to a Register effect \( E_3 \).
• There is an Intrinsic data dependency from the Register effect \( E_3 \) to \( RW_2 \).
• The Register effect \( E_3 \) affects the value written by \( W_2 \).

### Pick Control dependency

A Pick Control dependency from a read Register effect or read Memory effect \( R_1 \) to an effect \( E_2 \) exists if and only if all of the following apply:

• There is a Pick Basic dependency from \( R_1 \) to a Branching effect \( BR_3 \).
• The Branching effect \( BR_3 \) is in program order before \( E_2 \).

### Pick dependency

A Pick dependency from a read Register effect or read Memory effect \( R_1 \) to an effect \( E_2 \) exists if and only if one of the following applies:

• There is a Pick Basic dependency from \( R_1 \) to \( E_2 \).
• There is a Pick Address dependency from \( R_1 \) to \( E_2 \).
• There is a Pick Data dependency from \( R_1 \) to \( E_2 \).
• There is a Pick Control dependency from \( R_1 \) to \( E_2 \).

### Ordering relations

Dependency-ordered-before

A dependency creates externally-visible order between a memory read effect and another Memory effect generated by the same Observer. A memory read effect \( R_1 \) is Dependency-ordered-before a memory read or write effect \( RW_2 \) from the same Observer if and only if \( R_1 \) appears in program order before \( RW_2 \) and any of the following cases apply:

• There is an Address dependency or a Data dependency from \( R_1 \) to \( RW_2 \).
• \( RW_2 \) is a memory write effect \( W_2 \) and there is a Control dependency from \( R_1 \) to \( W_2 \).
• \( RW_2 \) is a memory read effect \( R_2 \) generated by an instruction appearing in program order after an instruction that generates a Context synchronization event \( E_3 \), and there is a Dependency through registers from \( R_1 \) to \( E_3 \).
• \( RW_2 \) is a memory write effect \( W_2 \) appearing in program order after a memory read or write effect \( RW_3 \) and there is an Address dependency on a Memory effect \( W_3 \) from \( R_1 \) to \( W_3 \).
• \( RW_2 \) is a Local read successor \( R_2 \) of a memory write effect \( W_3 \) and there is an Address dependency or a Data dependency from \( R_1 \) to \( W_3 \).
Pick-ordered-before

A read Register effect or read Memory effect $R_1$ is **Pick-ordered-before** a read or write Memory effect $RW_2$ from the same Observer if and only if $R_1$ appears in program order before $RW_2$ and any of the following cases apply:

- $RW_2$ is a write Memory effect $W_2$ and there is a Pick dependency from $R_1$ to $W_2$.
- $RW_2$ is a read Memory effect $R_2$ generated by an instruction appearing in program order after an instruction that generates a Context synchronization event $E_3$, and there is a Pick Control dependency from $R_1$ to $E_3$.
- $RW_2$ is a Memory effect generated by an instruction appearing in program order after an instruction that generates a Context synchronization event $E_3$, there is a Pick Address dependency from $R_1$ to an effect $E_4$ and $E_4$ is in program order before $E_3$.
- $RW_2$ is a write Memory effect $W_2$, there is a Pick Address dependency from $R_1$ to a read or write Memory effect $RW_3$ and $W_2$ is program-order-after $RW_3$.

Atomic-ordered-before

Load-Exclusive and Store-Exclusive instructions provide some ordering guarantees, even in the absence of dependencies. A read or a write Memory effect $RW_1$ is **Atomic-ordered-before** a read or write Memory effect $RW_2$ from the same Observer if and only if $RW_1$ appears in program order before $RW_2$ and either of the following cases apply:

- $RW_1$ is a read Memory effect $R_1$ and $RW_2$ is a write Memory effect $W_2$ such that $R_1$ and $W_2$ are generated by an atomic instruction or a successful Load-Exclusive/Store-Exclusive instruction pair to the same Location.
- $RW_1$ is a read Memory effect $R_1$ generated by an atomic instruction (resp. a Load-Exclusive instruction) and $RW_2$ is a read Memory effect $R_2$ generated by an instruction with Acquire or AcquirePC semantics such that $R_2$ is a Local read successor of the write Memory effect $W_3$ generated by the same atomic instruction as $R_1$ (resp. the successful Store-Exclusive instruction paired with the Load-Exclusive instruction that generated $R_1$).

For more information, see [Synchronization and semaphores](#) on page B2-211.

Barrier-ordered-before

Barrier instructions order prior Memory effects before subsequent Memory effects generated by the same Observer. A memory read or write effect $RW_1$ is **Barrier-ordered-before** a memory read or write effect $RW_2$ from the same Observer if and only if $RW_1$ appears in program order before $RW_2$ and any of the following cases apply:

- $RW_1$ appears in program order before a **DMB FULL** that appears in program order before $RW_2$.
- $RW_1$ is a memory write effect $W_1$ and is generated by an atomic instruction with both Acquire and Release semantics.
- $RW_1$ is a memory write effect $W_1$ generated by an instruction with Release semantics and $RW_2$ is a memory read effect $R_2$, except a Tag-Check-read, generated by an instruction with Acquire semantics.
- $RW_1$ is a memory read effect $R_1$ and appears in program order before a **DMB LD** that appears in program order before $RW_2$.
- $RW_1$ is a memory read effect $R_1$, except a Tag-Check-read, and is generated by an instruction with Acquire or AcquirePC semantics.
- $RW_1$ is a memory write effect $W_1$ and $RW_2$ is a memory write effect $W_2$ appearing in program order before a **DMB ST** that appears in program order before $W_2$.
- $RW_2$ is a memory write effect $W_2$ and is generated by an instruction with Release semantics.
Tag-ordered-before

If FEAT_MTE2 is implemented, a Tag read \( R_1 \) is Tag-ordered-before a memory read or write effect Checked data access \( RW_2 \) generated by the same instruction if and only if all of the following apply:

- \( R_1 \) is in the Intrinsic order of that instruction before \( RW_2 \).
- \( R_1 \) reads the Allocation Tag at a tag physical address and compares it with the physical address Tag of the instruction. If the result of the comparison can cause a precise exception and the result is negative, then \( RW_2 \) does not architecturally occur.

Tag-Location-Ordered

Tag-Check-reads \( R_1 \) and \( R_2 \) are Tag-Location-Ordered if and only if all the following apply:

- \( R_1 \) is Tag-ordered-before a Checked data access \( RW_3 \).
- \( R_2 \) is Tag-ordered-before a Checked data access \( RW_4 \).
- \( RW_3 \) and \( RW_4 \) are to the same Location.

Locally-ordered-before

Dependencies, Local write successor, load/store-exclusive, atomic and barrier instructions can be composed within an Observer to create externally-visible order. A memory read or write effect \( RW_1 \) is Locally-ordered-before a memory read or write effect \( RW_2 \) from the same Observer if and only if any of the following cases apply:

- \( RW_1 \) is a memory write effect \( W_1 \) and \( RW_2 \) is a memory write effect \( W_2 \) that is equal to or generated by the same instruction as a Local write successor of \( RW_1 \).
- \( RW_1 \) is Pick-ordered-before \( RW_2 \).
- \( RW_1 \) is Dependency-ordered-before \( RW_2 \).
- \( RW_1 \) is Atomic-ordered-before \( RW_2 \).
- \( RW_1 \) is Barrier-ordered-before \( RW_2 \).
- \( RW_1 \) is Tag-ordered-before \( RW_2 \).
- \( RW_1 \) is Locally-ordered-before a memory read or write effect that is Locally-ordered-before \( RW_2 \).

Pick-locally-ordered-before

A read or a write Memory effect \( RW_1 \) is Pick-locally-ordered-before a read or a write Memory effect \( RW_2 \) from the same Observer if and only if all of the following apply:

- There is a Pick Basic dependency from \( RW_1 \) to an effect \( E_3 \).
- \( RW_2 \) is a write Memory effect \( W_2 \).
- \( E_3 \) is Locally-ordered-before \( W_2 \).

B2.3.4 Ordering constraints

The Arm memory model is described as being Other-multi-copy atomic. The definition of Other-multi-copy atomic is as follows:

Other-multi-copy atomic

In an Other-multi-copy atomic system, it is required that a memory write effect from an Observer, if observed by a different Observer, is then observed by all other Observers that access the Location coherently. It is, however, permitted for an Observer to observe its own writes prior to making them visible to other observers in the system.
The Other-multi-copy atomic property of the Arm memory model is enforced by placing constraints on the possible executions of a program. Those executions that meet the constraints given by the ordering model are said to be Architecturally well-formed. An implementation that is executing a program is only permitted to exhibit behavior consistent with an Architecturally well-formed execution.

Architecturally well-formed

An Architecturally well-formed execution must satisfy both the Internal visibility requirement and any of the three alternative External ordering constraints.

B2.3.5 Internal visibility requirement

For a memory read or write effect RW₁ that appears in program order before a memory read or write effect RW₂ to the same Location:

- Where one or more of the following statements is true:
  - RW₁ is not a Tag-Check-read.
  - RW₂ is not a Tag-Check-read.
  - RW₁ and RW₂ are both Tag-Check-reads R₁ and R₂ that are Tag-Location-Ordered.

- The Internal visibility requirement requires that exactly one of the following statements is true:
  - RW₂ is a memory write effect W₂ that is Coherence-after RW₁.
  - RW₁ is a memory write effect W₁, RW₂ is a memory read effect R₂ and either:
    - R₂ Reads-from W₁.
    - R₂ Reads-from a memory write effect that is Coherence-after W₁.
  - RW₁ and RW₂ are both reads R₁, R₂, R₁ Reads-from a memory write effect W₃ and either:
    - R₂ Reads-from W₃.
    - R₂ Reads-from a memory write effect that is Coherence-after W₃.

Informally, if a Memory effect M₁ from an Observer appears in program order before a Memory effect M₂ from the same Observer, then M₁ will be seen to occur before M₂ by that Observer.

B2.3.6 External ordering constraints

The Arm memory model offers the following three alternative representations of the External ordering constraint:

- External visibility requirement.
- External completion requirement.
- External global completion requirement.

An Architecturally well-formed execution must satisfy both the Internal visibility requirement and one of the three alternative representations in the External ordering constraints.

External visibility requirement

Ordered-before

An arbitrary pair of Memory effects is ordered if it can be linked by a chain of ordered accesses consistent with external observation. A memory read or write effect RW₁ is Ordered-before a memory read or write effect RW₂ if and only if any of the following cases apply:

- RW₁ is Observed-by a memory read or write effect RW₃ which is generated by the same instruction as RW₂.
- RW₁ is Locally-ordered-before RW₂.
- RW₁ is Pick-locally-ordered-before RW₂.
- RW₁ is Ordered-before a memory read or write effect that is Ordered-before RW₂.
For a memory read or write effect \( \text{RW}_1 \) from an Observer that is Ordered-before a memory read or write effect \( \text{RW}_2 \) from a different Observer, the External visibility requirement requires that \( \text{RW}_2 \) is not Observed-by \( \text{RW}_1 \). This means that an Architecturally well-formed execution must not exhibit a cycle in the Ordered-before relation.

Informally, if a Memory effect \( \text{M}_1 \) from an Observer appears in program order before a Memory effect \( \text{M}_2 \) from the same Observer, then \( \text{M}_1 \) will be seen to occur before \( \text{M}_2 \) by all Observers in the system.

**Completes-before order**

The Completes-before order is a total order that corresponds to the order in which Memory effects complete within the system. The following effects constitute a single entry in the Completes-before order:

- Writes from the same instruction.
- Reads from the same instruction which read from external writes.
- Reads from the same instruction which read from the same internal write.

All other reads constitute distinct entries in the Completes-before order.

**Completes-before**

A memory read or write effect \( \text{RW}_1 \) Completes-before a memory read or write effect \( \text{RW}_2 \) if and only if \( \text{RW}_1 \) appears in the Completes-before order before \( \text{RW}_2 \).

**Deriving Reads-from and Coherence order from the Completes-before order**

The Completes-before order can be used to resolve the Reads-from and Coherence order relations for every memory access in the system as follows:

- For a memory read effect \( \text{R}_1 \) of a memory location by an Observer, then:
  - If there is a memory write effect \( \text{W}_2 \) to the same Location from the same Observer and all of the following are true:
    - \( \text{W}_2 \) appears in program order before \( \text{R}_1 \).
    - \( \text{R}_1 \) Completes-before \( \text{W}_2 \).
    - There are no writes to the Location appearing in program order between \( \text{W}_2 \) and \( \text{R}_1 \) then \( \text{R}_1 \) Reads-from \( \text{W}_2 \).
  - Otherwise, \( \text{R}_1 \) Reads-from its closest preceding write in the Completes-before order to the same Location. If no such write exists, then \( \text{R}_1 \) Reads-from the initial value of the memory location.

- The Coherence order of writes to a memory location is the order in which those writes appear in the Completes-before order. The final value of each memory location is therefore determined by the final write to each Location in the Completes-before order. If no such write exists for a given Location, the final value is the initial value of that Location.

**External completion requirement**

A memory read or write effect \( \text{RW}_1 \) Completes-before a memory read or write effect \( \text{RW}_2 \) if and only if any of the following statements are true:

- \( \text{RW}_1 \) is Locally-ordered-before \( \text{RW}_2 \).
- \( \text{RW}_1 \) is Pick-locally-ordered-before \( \text{RW}_2 \).
- \( \text{RW}_1 \) is a memory read effect \( \text{R}_1 \) and \( \text{RW}_2 \) is a memory read effect \( \text{R}_2 \) and \( \text{R}_1 \) is Single-copy-atomic-ordered-before \( \text{R}_2 \).

**Globally-completes-before order**

The Globally-completes-before order is a total order that corresponds to the order in which Memory effects globally-complete within the system. The following effects constitute a single entry in the Globally-completes-before order:

- Writes from the same instruction.
- Reads from the same instruction which read from external writes.
• Reads from the same instruction which read from the same internal write. All other reads constitute distinct entries in the Globally-completes-before order.

**Globally-completes-before**

A memory read or write effect RW<sub>1</sub> *Globally-completes-before* a memory read or write effect RW<sub>2</sub> if and only if RW<sub>1</sub> appears in the Globally-completes-before order before RW<sub>2</sub>.

**Deriving Reads-from and Coherence order from the Globally-completes-before order**

The Globally-completes-before order can be used to resolve the Reads-from and Coherence order relations for every memory access in the system as follows:

• A memory read effect R<sub>1</sub> of a memory location by an Observer Reads-from its closest preceding write in the Globally-completes-before order to the same Location. If no such write exists, then R<sub>1</sub> Reads-from the initial value of the memory location.

• The Coherence order of writes to a memory location is the order in which those writes appear in the Globally-completes-before order. The final value of each memory location is therefore determined by the final write to each Location in the Globally-completes-before order. If no such write exists for a given Location, the final value is the initial value of that Location.

**External global completion requirement**

The External global completion requirement requires that a memory read or write effect RW<sub>1</sub> *Globally-completes-before* a memory read or write effect RW<sub>2</sub> if and only if any of the following statements are true:

• RW<sub>1</sub> is Locally-ordered-before RW<sub>2</sub> and either:
  — RW<sub>1</sub> is a memory write effect.
  — RW<sub>1</sub> is a memory read effect R<sub>1</sub> and either:
    — R<sub>1</sub> is not a Local read successor of a memory write effect.
    — R<sub>1</sub> is a Local read successor of a memory write effect that is Locally-ordered-before RW<sub>2</sub>.
    — R<sub>1</sub> is a Local read successor of a memory write effect that is Pick-locally-ordered-before RW<sub>2</sub>.
  — RW<sub>1</sub> is a memory read effect R<sub>1</sub> and RW<sub>2</sub> is a memory read effect R<sub>2</sub> and R<sub>1</sub> is Single-copy-atomic-ordered-before R<sub>2</sub>.

**B2.3.7 Completion and endpoint ordering**

Interaction between Observers in a system is not restricted to communication via shared variables in coherent memory. For example, an Observer could configure an interrupt controller to raise an interrupt on another Observer as a form of message passing. These interactions typically involve an additional agent, which defines the instruction sequence that is required to establish communication links between different Observers. When these forms of interaction are used in conjunction with shared variables, a DSB instruction can be used to enforce ordering between them.

For all memory, the completion rules are defined as:

• A memory read effect R<sub>1</sub> to a Location is complete for a shareability domain when all of the following are true:
  — Any write to the same Location by an Observer within the shareability domain will be Coherence-after R<sub>1</sub>.
  — Any translation table walks associated with R<sub>1</sub> are complete for that shareability domain.

• A memory write effect W<sub>1</sub> to a Location is complete for a shareability domain when all of the following are true:
  — Any write to the same Location by an Observer within the shareability domain will be Coherence-after W<sub>1</sub>.
— Any read to the same Location by an Observer within the shareability domain will either Reads-from \( W_1 \) or Reads-from a memory write effect that is Coherence-after \( W_1 \).
— Any translation table walks associated with the write are complete for that shareability domain.

- A translation table walk is complete for a shareability domain when the memory accesses, including the updates to translation table entries, associated with the translation table walk are complete for that shareability domain, and the TLB is updated.
- A cache maintenance instruction is complete for a shareability domain when the memory effects of the instruction are complete for that shareability domain, and any translation table walks that arise from the instruction are complete for that shareability domain.
- A TLB invalidate instruction is complete when all memory accesses using the TLB entries that have been invalidated are complete.

The completion of any cache or TLB maintenance instruction includes its completion on all PEs that are affected by both the instruction and the DSB operation that is required to guarantee visibility of the maintenance instruction.

--- Note ---

These completion rules mean that, for example, a cache maintenance instruction that operates by VA to the PoC completes only after memory at the PoC has been updated.

---

Additionally, for Device-nGnRnE memory, a read or write of a Location in a Memory-mapped peripheral that exhibits side-effects is complete only when the read or write both:
- Can begin to affect the state of the Memory-mapped peripheral.
- Can trigger all associated side-effects, whether they affect other peripheral devices, PEs, or memory.

--- Note ---

This requirement for Device-nGnRnE memory is consistent with the memory access having reached the peripheral endpoint.

---

Peripherals

This section defines a Memory-mapped peripheral and the total order of reads and writes to a peripheral which is defined as the Peripheral coherence order:

Memory-mapped peripheral

A Memory-mapped peripheral occupies a memory region of IMPLEMENTATION DEFINED size and can be accessed using load and store instructions. Memory effects to a Memory-mapped peripheral can have side-effects, such as causing the peripheral to perform an action. Values that are read from addresses within a Memory-mapped peripheral might not correspond to the last data value written to those addresses. As such, Memory effects to a Memory-mapped peripheral might not appear in the Reads-from or Coherence order relations.

Peripheral coherence order

The Peripheral coherence order of a Memory-mapped peripheral is a total order on all reads and writes to that peripheral.

--- Note ---

The Peripheral coherence order for a Memory-mapped peripheral signifies the order in which accesses arrive at the endpoint.

---

For a memory read or write effect \( RW_1 \) and a memory read or write effect \( RW_2 \) to the same peripheral, then \( RW_1 \) will appear in the Peripheral coherence order for the peripheral before \( RW_2 \) if either of the following cases apply:
- \( RW_1 \) and \( RW_2 \) are accesses using Non-cacheable or Device attributes and \( RW_1 \) is Ordered-before \( RW_2 \).
• RW₁ and RW₂ are accesses using Device-nGnRE or Device-nGnRnE attributes, with the same XS attribute value, and RW₁ appears in program order before RW₂.

Note
When FEAT_XS is implemented, if accesses marked with the Device-nGnRE or Device-nGnRnE attributes are within the same Memory-mapped peripheral, but the XS attribute is not the same on those accesses, the order of arrival at the endpoint is not defined by the architecture.

Out-of-band-ordered-before
A memory read or write effect RW₁ is Out-of-band-ordered-before a memory read or write effect RW₂ if and only if either of the following cases apply:

• RW₁ appears in program order before a DSB instruction that begins an IMPLEMENTATION DEFINED instruction sequence indirectly leading to the generation of RW₂.
• RW₁ is Ordered-before a memory read or write effect RW₃ and RW₃ is Out-of-band-ordered-before RW₂.

If a Memory effect M₁ is Out-of-band-ordered-before a memory read or write effect M₂, then M₁ is seen to occur before M₂ by all Observers.

Note
Arm expects that, in most systems with early acknowledgments, those acknowledgments will come from a point at or after the point that establishes global visibility. This is expected in such systems to enable the acknowledgments to be used as part of the mechanisms to implement the ordering requirements of the Arm memory model.

B2.3.8 SVE memory ordering relaxations

ICTNGV
The Arm memory model is relaxed for reads and writes generated by SVE load and store instructions.

QLIPC
When two reads generated by SVE vector load instructions have an address dependency, the dependency does not contribute to the dependency-ordered-before relation.

YMBMZ
When a pair of reads access the same location, and at least one of the reads is generated by an SVE load instruction, for a given observer, the pair of reads is not required to satisfy the internal visibility requirement.

CHWV
When a single SVE vector store instruction generates multiple writes to the same location, the instruction ensures that these writes appear in the coherence order for that location, in order of increasing vector element number. No other ordering restrictions apply to memory effects generated by the same SVE store instruction.

LVXTJ
If a single SVE load instruction generates multiple reads, the order in which the reads for different elements and registers appear is not architecturally defined.

VMDYZ
If an address dependency exists between two memory reads and an SVE non-temporal vector load instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, the memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

CCWGN
For any SVE load or store instruction that generates multiple single-copy atomic accesses to Normal or Device memory, there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic memory element access sizes.

B2.3.9 Ordering of instruction fetches
For two memory locations A and B, if A has been written to with an updated value and been made coherent with the instruction fetches of the shareability domain before B has been written to with an updated value by an observer in the same shareability domain, then where, for an observer in the shareability domain, an instruction read from B appears in program order before an instruction fetched from A, if the instruction read from B contains the updated value of B then the instruction read from A appearing later in program order will contain the updated value of A.
A write has been made coherent with an instruction fetch of a shareability domain when:

**CTR_EL0.{DIC, IDC} == \{0, 0\}**

The location written to has been cleaned to the Point of unification (PoU) from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

**CTR_EL0.{DIC, IDC} == \{1, 0\}**

The location written to has been cleaned to the Point of unification (PoU) from the data cache, and that clean is complete for the shareability domain.

**CTR_EL0.{DIC, IDC} == \{0, 1\}**

The write is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

**CTR_EL0.{DIC, IDC} == \{1, 1\}**

The write is complete for the shareability domain.

--- Note

Microarchitecturally, this means that these situations cannot both be true in an implementation:

- After delays in fetching from memory, the instruction queue can have entries written into it out of order.
- For an implementation:
  - When **CTR_EL0.DIC == 0**, if there is an outstanding entry in the instruction queue, then later entries in the instruction queue are not impacted by the **IC IVAU** instructions of a different core.
  - When **CTR_EL0.DIC == 1**, if there is a write to the location that is held in the queue when there is an outstanding entry in the instruction queue for an older entry, then the instruction queue does not have entries invalidated from it.

### B2.3.10 Restrictions on the effects of speculation

The Arm architecture places certain restrictions on the effects of speculation. These are:

- Each load from a location using a particular VA after an exception return that is a Context synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception exit.
- Each load from a location using a particular VA after an exception entry that is a Context synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception entry.
- Any load from a location using a particular VA before an exception entry that is a Context synchronization event will not speculatively read data from a store to the same location using the same VA after the exception entry.
- Any load from a location using a particular VA before an exception return that is a Context synchronization event will not speculatively read data from a store to the same location using the same VA after the exception exit.
- When data is loaded under speculation with a Translation fault, it cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence.
- When data is loaded under speculation from a location without a translation for the translation regime being speculated in, the data cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence.
• When data is loaded as a result of speculative accesses made after TLBI + DSB + ERET using a translation that was invalidated by the TLBI, the data cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence. The execution timing of any other instructions in the speculative sequence is not a function of the data loaded.

• Changes to System registers must not occur speculatively in a way that can affect a speculative memory access that can cause a change to the micro-architectural state.

• Changes to Special-purpose registers can occur speculatively.

• Execute-never controls apply to speculative instruction fetching. See Instruction execution permissions on page D8-5138.

• When writing new instructions to memory, there is no requirement for an SB instruction to prevent speculative execution of the old code. See Instruction cache maintenance instructions on page K13-11773.

Note
The prohibition of using data loaded under speculation with faults to form addresses, condition codes or SVE predicate values does not prohibit the use of value predicted data from such locations for such purposes, so long as the training of the data value prediction was from the hardware defined context that is using the prediction. A consequence of this is that training of value prediction cannot be based on data loaded under speculation with a translation or Permission fault.

Speculative Store Bypass Safe (SSBS)
When FEAT_SSBS is implemented, PSTATE.SSBS is a control that can be set by software to indicate whether hardware is permitted to use, in a manner that is potentially speculatively exploitable, a speculative value in a register that has been loaded from memory using a load instruction that speculatively read the location being loaded from, where the entry that is speculatively read is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

A speculative value in a register is used in a potentially speculatively exploitable manner if it is used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence or if the execution timing of any other instructions in the speculative sequence is a function of the data loaded under speculation.

When the value of PSTATE.SSBS is 0, hardware is not permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

When the value of PSTATE.SSBS is 1, hardware is permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

Note
• If speculation is permitted, then cache timing side channels can lead to addresses being derived using reads of address values that have been speculatively loaded from memory to a register.

• Software written for architectures from Armv8.0 to Armv8.4 will set SPSR_ELx.SSBS to 0. This means that PSTATE.SSBS will not set, so hardware will not be permitted to use speculative loads with outstanding memory disambiguation issues for any subsequent speculative memory accesses if there is any possibility of those subsequent memory accesses creating a cache timing side channel.
Restrictions on exploitative control of speculative execution

The execution of some code (code1) can exploitatively control speculative execution of some other code (code2) if and only if all of the following apply:

• The actions of code1 can influence, in a manner that is not hard-to-determine, the speculative execution of code2 to cause an irreversible change to the microarchitectural state of the PE that is indicative of some architectural state accessible to the execution context of code2.

• Code1 has control in determining the choice of the architecture state that the irreversible change to the microarchitectural state is indicative of.

• The irreversible changes to the microarchitectural state of the PE can be measured by code executing in an execution context other than that of code2 to allow the retrieval of the architectural state in a manner that is not hard-to-determine.

Note

Mechanisms to prevent the influence and the state recovery being “not hard-to-determine” are left open to implementations. Examples could include the complete separation of prediction resources, or the isolation of the predictions using a cryptographic or pseudo-random mechanism to separate each context.

Restrictions on predictive leakage

The execution of some code (code1) can predictively leak to some other code (code2) if and only if all of the following apply:

• The execution of code1 influences, in a manner that is not hard-to-determine, the predictive micro-architectural structures of the implementation to behave in a way that is indicative of some architectural state accessible to the execution context of code1.

• The predictive micro-architectural structures of the implementation impact the timing of the speculative execution of code2 in a way that enables code2 to recover the architecture state in a manner that is not hard-to-determine.

Note

Mechanisms to prevent the influence and the state recovery being “not hard-to-determine” are left open to implementations. Examples could include the complete separation of prediction resources, or the isolation of the predictions using a cryptographic or pseudo-random mechanism to separate each context.

Restrictions on the effects of speculation from Armv8.5

Note

If SCR_EL3.EEL2 is changed, in order to remove all VMID tagging from Secure EL1 and Secure EL0 entries, each prediction resource should be invalidated by software for:

• Secure EL0 for all ASID and VMID values.
• Secure EL1 for all VMID values.

Further restrictions on speculation are introduced by some additional architectural features as described here.

FEAT_CSV3 introduces these restrictions:

• Data loaded under speculation with a Permission or Domain fault cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence.

• Any System register read under speculation to a register that is not architecturally accessible from the current Exception level cannot be used to form an address, to generate condition codes, or to generate SVE predicate values to be used by other instructions in the speculative sequence.
As the effects of speculation are not architecturally visible, this restriction requires that the effect of any speculation cannot give rise to side channels that will leak the values of memory locations, System registers, or Special-purpose registers to a level of privilege that would otherwise not be able to determine those values.

- Changes to System registers must not occur speculatively in a way that can affect a speculative memory access that can cause a change to the micro-architectural state.

- Changes to Special-purpose registers can occur speculatively.

**Note**

Changes to Special-purpose registers can occur speculatively.

**FEAT_CSV2, FEAT_CSV2_1p1, FEAT_CSV2_1p2, FEAT_CSV2_2 and FEAT_CSV2_3** introduce a range of additional restrictions.

If FEAT_CSV2 is implemented:

- Code running in one hardware-defined context (context1) cannot either exploitatively control, or predictively leak to, the speculative execution of code in a different hardware-defined context (context2), as a result of the behavior of any of the following resources:
  - Branch target prediction based on the branch targets used in context1.
    - This applies to both direct and indirect branches, but excludes the prediction of the direction of a conditional branch.
  - Data Value predictions based on data value from execution in context1.
  - Virtual address-based cache prefetch predictions generated as a result of execution in context1.
  - Any other prediction mechanisms, other than Branch, Data Value, or Cache Prefetch predictions.

In this definition, the hardware-defined context is determined by:

- The Exception level.
- The Security state.
- When executing at EL1, if EL2 is implemented and enabled in the current Security state, the VMID.
- When executing at EL0, whether the EL1&0 or the EL2&0 translation regime is in use.
- When executing at EL0 and using the EL1&0 translation regime, the address space identifier (ASID) and, if EL2 is implemented and enabled in the current Security state, the VMID.
- When executing at EL0 and using the EL2&0 translation regime, the ASID.

If FEAT_CSV2_2 is implemented, then SCXTNUM_ELx is also part of the hardware-defined context.

If either FEAT_CSV2_1p1 or FEAT_CSV2_3 is implemented, code running in one hardware-defined context (context1) cannot either exploitatively control, or predictively leak to, the speculative execution of code in a different hardware-defined context (context2) as a result of the behavior of branch target prediction based on the branch history used in context1.

If FEAT_CSV2_1p1 is implemented, branch or data values trained from one instruction address can exploitatively control, or predictively leak to, the speculative execution of code from a different address only in a hard-to-determine way.

If FEAT_CSV2_1p2 is implemented, the SCXTNUM_ELx register is implemented, but is not part of the hardware-defined context.

**B2.3.11 Memory barriers**

Memory barrier is the general term applied to an instruction, or sequence of instructions, that forces synchronization events by a PE with respect to retiring load/store instructions. The memory barriers defined by the Arm architecture provide a range of functionality, including:

- Ordering of load/store instructions.
- Completion of load/store instructions.
The following subsections describe the Arm memory barrier instructions:

- **Instruction Synchronization Barrier (ISB)**
- **Data Memory Barrier (DMB)**
- **Data Synchronization Barrier (DSB)** on page B2-178.
- **Speculation Barrier (SB)** on page B2-176.
- **Consumption of Speculative Data Barrier (CSDB)** on page B2-176.
- **Speculative Store Bypass Barrier (SSBB)** on page B2-177.
- **Profiling Synchronization Barrier (PSB CSYNC)** on page B2-177.
- **Physical Speculative Store Bypass Barrier (PSSBB)** on page B2-177.
- **Trace Synchronization Barrier (TSB CSYNC)** on page B2-177.
- **Shareability and access limitations on the data barrier operations** on page B2-179.
- **LoadLOAcquire, StoreLORelease** on page B2-181.

--- Note ---

Depending on the required synchronization, a program might use memory barriers on their own, or it might use them in conjunction with cache maintenance and memory management instructions that in general are available only when software execution is at EL1 or higher.

DMB and DSB instructions affect reads and writes to the memory system generated by load/store instructions and data or unified cache maintenance instructions being executed by the PE.

### Instruction Synchronization Barrier (ISB)

An ISB instruction ensures that all instructions that come after the ISB instruction in program order are fetched from the cache or memory after the ISB instruction has completed. Using an ISB ensures that the effects of context-changing operations executed before the ISB are visible to the instructions fetched after the ISB instruction.

Examples of context-changing operations that require the insertion of an ISB instruction to ensure the effects of the operation are visible to instructions fetched after the ISB instruction are:

- Completed cache and TLB maintenance instructions.
- Changes to System registers.

Any context-changing operations appearing in program order after the ISB instruction take effect only after the ISB has been executed.

The pseudocode function for the operation of an ISB is `InstructionSynchronizationBarrier()`.

See also **Memory barriers** on page D7-5077.

### Data Memory Barrier (DMB)

The DMB instruction is a memory barrier instruction that ensures the relative order of memory accesses before the barrier with memory accesses after the barrier. The DMB instruction does not ensure the completion of any of the memory accesses for which it ensures relative order.

The full definition of the DMB instruction is covered formally in the **Definition of the Arm memory model** on page B2-159 and this introduction to the DMB instruction is not intended to contradict that section.

The basic principle of a DMB instruction is to introduce order between memory accesses that are specified to be affected by the DMB options supplied as arguments to the DMB instruction. The DMB instruction ensures that all affected memory accesses by the PE executing the DMB instruction that appear in program order before the DMB instruction and those which originate from a different PE, to the extent required by the DMB options, which have been Observed-by the PE before the DMB instruction is executed, are Observed-by each PE, to the extent required by the DMB options, before any affected memory accesses that appear in program order after the DMB instruction are Observed-by that PE.
The use of a DMB instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The DMB instruction affects only memory accesses and the operation of data cache and unified cache maintenance instructions, see A64 Cache maintenance instructions on page D7-5054. It has no effect on the ordering of any other instructions executing on the PE.

The pseudocode function for the operation of a DMB instruction is `DataMemoryBarrier()`.

### Speculation Barrier (SB)

An SB instruction is a memory barrier that prevents speculative execution of instructions until after the barrier has completed when those instructions could be observed through side-channels.

Until the barrier completes, the speculative execution of any instruction appearing later in the program order than the barrier:

- Cannot be performed to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be performed when predicting that a instruction that could generate an exception does not generate an exception.

Speculative execution of an SB instruction:

- Cannot be as a result of control flow speculation.
- Cannot be as a result of data value speculation.
- Can be as a result of predicting that an instruction that could generate an exception does not generate an exception.

An SB instruction can complete when:

- It is known that it is not speculative.
- All the predicted data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

**Note**

The SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched, so long as the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after the SB instruction.

### Consumption of Speculative Data Barrier (CSDB)

The CSDB instruction is a memory barrier instruction that controls speculative execution and data value prediction. This includes:

- Data value predictions of any instructions.
- \( \text{PSTATE.}(N,Z,C,V) \) predictions of any instructions other than conditional branch instructions appearing in program order before the CSDB that have not been architecturally resolved.
- Predictions of SVE predication state for any SVE instructions.

For purposes of the definition of CSDB, \( \text{PSTATE.}(N,Z,C,V) \) is not considered a data value. This definition permits:

- Control flow speculation before and after the CSDB instruction.
- Speculative execution of conditional data processing instructions after the CSDB instruction, unless they use the results of data value or \( \text{PSTATE.}(N,Z,C,V) \) predictions of instructions appearing in program order before the CSDB instruction that have not been architecturally resolved.
Speculative Store Bypass Barrier (SSBB)

The SSBB instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB instruction, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the SSBB instruction.
- When a load to a location appears in program order before the SSBB instruction, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order after the SSBB instruction.

Profiling Synchronization Barrier (PSB CSYNC)

The PSB CSYNC instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following DSB instruction completes when the writes to the profiling buffer have completed.

If the Statistical Profiling Extension is not implemented, this instruction executes as a NOP.

Physical Speculative Store Bypass Barrier (PSSBB)

The PSSBB instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same physical address under certain conditions.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the PSSBB instruction, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the PSSBB instruction.
- When a load to a location appears in program order before the PSSBB instruction, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the PSSBB instruction.

Note: The effect of this barrier applies to accesses to the same location even if they are accessed with different virtual addresses and from different Exception levels.

Trace Synchronization Barrier (TSB CSYNC)

The TSB CSYNC instruction is a barrier instruction that preserves the relative order of accesses to System registers due to trace operations and other accesses to the same registers.

A trace operation is an operation of the trace unit generating trace for an instruction when FEAT_TRF is implemented and enabled.
A TSB CSYNC instruction is not required to execute in program order with respect to other instructions. This includes being reordered with respect to other trace instructions. One or more Context synchronization events are required to ensure that TSB CSYNC instruction is executed in the necessary order.

If trace is generated between a Context synchronization event and a TSB CSYNC operation, these trace operations may be reordered with respect to the TSB CSYNC operation, and therefore may not be synchronized.

The following situations are synchronized using a TSB CSYNC operation:

- A direct write B to a System register is ordered after an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all of the following are true:
  - A is executed in program order before a Context synchronization event C.
  - C is in program order before a TSB CSYNC operation T.
  - B is executed in program order after T.

- A direct read B of a System register is ordered after an indirect write to the same register by a trace operation of a traced instruction A if all the following are true:
  - A is executed in program order before a Context synchronization event C1.
  - C1 is in program order before TSB CSYNC operation T.
  - T is executed in program order before a second Context synchronization event C2.
  - B is executed in program order after C2.

A TSB CSYNC operation is not needed to ensure a direct write B to a System register is ordered before an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all the following are true:

- A is executed in program order after a Context synchronization event C.
- B is executed in program order before C.

If FEAT_TRBE is implemented, the requirements in this section are extended. See Synchronization and the Trace Buffer Unit on page D6-5026.

The pseudocode function for the operation of a TSB CSYNC instruction is TraceSynchronizationBarrier().

Data Synchronization Barrier (DSB)

A DSB instruction is a memory barrier that ensures that memory accesses that occur before the DSB instruction have completed before the completion of the DSB instruction. In doing this, it acts as a stronger barrier than a DMB and all ordering that is created by a DMB with specific options is also generated by a DSB with the same options.

Execution of a DSB instruction:

- At EL2 ensures that any memory accesses caused by Speculative translation table walks from the EL1&0 translation regime have been observed.
- At EL3 ensures that any memory accesses caused by speculative translation table walks from the EL2, EL1&0 or EL2&0 translation regimes have been observed.

For more information, see Out-of-context translation regimes on page D8-5087.

A DSB instruction executed by a PE, PEe, completes when all of the following apply:

- All explicit memory effects of the required access types appearing in program order before the DSB are complete for the set of observers in the required shareability domain.
- If the required access types of the DSB is reads and writes, the following instructions issued by PEe before the DSB are complete for the required shareability domain:
  - All cache maintenance instructions.
  - All TLB maintenance instructions.
  - All PSB CYNC instructions.
When FEAT_XS is implemented, if the required access types of the DSB is reads and writes, completion of the DSB instruction with the nXS qualifier executed by a PE, PEe, ensures that:

- All previous TLBInXS maintenance operations generated by AArch64 TLB maintenance instructions with the nXS qualifier executed by PEe are finished for all PEs in the shareability domain of the DSB instruction.
- All previous TLBInXS maintenance operations generated by AArch32 or AArch64 TLB maintenance instructions executed at EL1 by PEe when HCRX_EL2.FnXS is 1 are finished for all PEs in the shareability domain of the DSB instruction.

Completion of the DSB instruction with the nXS qualifier executed by a PE, PEe, does not ensure that:

- All previous TLB maintenance operations generated by AArch32 or AArch64 TLB maintenance instructions executed at EL1 by PEe when HCRX_EL2.FnXS is 0 are finished for all PEs in the shareability domain of the DSB instruction.
- All previous TLB maintenance operations generated by AArch32 or AArch64 TLB maintenance instructions executed at EL2 or EL3 by PEe are finished for all PEs in the shareability domain of the DSB instruction.

In addition, no instruction that appears in program order after the DSB instruction can alter any state of the system or perform any part of its functionality until the DSB completes other than:

- Being fetched from memory and decoded.
- Reading the general-purpose, SIMD and floating-point, SVE vector or predicate, Special-purpose, or System registers that are directly or indirectly read without causing side-effects.
- If FEAT_ETS is not implemented, having any virtual addresses of loads and stores translated.

If FEAT_MTE2 is implemented, on completion of a DSB instruction operating over the Non-shareable domain, all updates to TFSR_ELx.TFx or TFSRE0_EL1.TFx due to Tag Check fails caused by accesses for which the DSB operates will be complete. For more information on FEAT_MTE2, see Chapter D9 The Memory Tagging Extension.

When FEAT_XS is implemented and HCRX_EL2.FnXS is 1, an AArch64 DSB instruction executed at EL1 or EL0 behaves in the same way as the corresponding DSB instruction with the nXS qualifier executed at EL1 or EL0.

If FEAT_TRBE is implemented, the requirements in this section are extended. See Trace synchronization and memory barriers on page D6-5031.

The pseudocode function for the operation of a DSB is DataSynchronizationBarrier().

See also:
- Memory barriers on page D7-5077.
- Ordering and completion of TLB maintenance instructions on page D8-5213.

Shareability and access limitations on the data barrier operations

The DMB and DSB instructions take an argument that specifies:

- The shareability domain over which the instruction must operate. This is one of:
  - Full system.
  - Outer Shareable.
  - Inner Shareable.
  - Non-shareable.

  Full system applies to all the observers in the system and, as such, encompasses the Inner and Outer Shareable domains of the processor.

  __________ Note __________

  The distinction between Full system and Outer Shareable is applicable only for Normal Non-cacheable memory accesses and Device memory accesses.

  __________

- The accesses for which the instruction operates. This is one of:
  - Read and write accesses, both before and after the barrier instruction.
— Write accesses only, before and after the barrier instruction.
— Read accesses before the barrier instruction, and read and write accesses after the barrier instruction.

--- Note ---
This form of a DMB or DSB instruction can be described as a load-load/store barrier.

For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) on page B2-175 or Data Synchronization Barrier (DSB) on page B2-178.

Table B2-1 on page B2-180 shows how these options are encoded in the <option> field of the instruction:

<table>
<thead>
<tr>
<th>Accesses</th>
<th>Shareability domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before the barrier</td>
<td>After the barrier</td>
</tr>
<tr>
<td>Reads and writes</td>
<td>Reads and writes</td>
</tr>
<tr>
<td>Writes</td>
<td>Writes</td>
</tr>
<tr>
<td>Reads</td>
<td>Reads and writes</td>
</tr>
</tbody>
</table>

See the instruction descriptions for more information:
• DMB on page C6-1485.
• DSB on page C6-1488.

--- Note ---
ISB also supports an optional limitation argument that can contain only one value that corresponds to full system operation, see ISB on page C6-1511.

**Load-Acquire, Load-AcquirePC, and Store-Release**

Arm provides a set of instructions with Acquire semantics for loads, and Release semantics for stores. These instructions support the Release Consistency sequentially consistent (RCsc) model. In addition, FEAT_LRCPC provides Load-AcquirePC instructions. The combination of Load-AcquirePC and Store-Release can be used to support the weaker Release Consistency processor consistent (RCpc) model.

The full definitions of the Load-Acquire and Load-AcquirePC instructions are covered formally in the Definition of the Arm memory model on page B2-159. This introduction to the Load-Acquire and Load-AcquirePC instructions is not intended to contradict that section.

The basic principle of both Load-Acquire and Load-AcquirePC instructions is to introduce order between:
• The memory access generated by the Load-Acquire or Load-AcquirePC instruction.
• The memory accesses appearing in program order after the Load-Acquire or Load-AcquirePC instruction, such that the memory access generated by the Load-Acquire or Load-AcquirePC instruction is Observed-by each PE to the extent that the PE is required to observe the access coherently, before any of the memory accesses appearing in program order after the Load-Acquire or Load-AcquirePC instruction are Observed-by that PE to the extent that the PE is required to observe the accesses coherently.

The use of a Load-Acquire or Load-AcquirePC instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The full definition of the Store-Release instruction is covered formally in the Definition of the Arm memory model on page B2-159 and this introduction to the Store-Release instruction is not intended to contradict that section.
The basic principle of a Store-Release instruction is to introduce order between the following:

- A set of memory accesses, RWx, that are generated by the PE executing the Store-Release instruction and that appear in program order before the Store-Release instruction, together with those that originate from a different PE to the extent that the PE is required to observe them coherently, Observed-by the PE before executing the Store-release.

- The memory access generated by the Store-Release (Wrel), such that all of the memory accesses, RWx, are Observed-by each PE to the extent that the PE is required to observe those accesses coherently, before Wrel is Observed-by that PE to the extent that the PE is required to observe that access coherently.

The use of a Store-Release instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

Where a Load-Acquire appears in program order after a Store-Release, the memory access generated by the Store-Release instruction is Observed-by each PE to the extent that PE is required to observe the access coherently, before the memory access generated by the Load-Acquire instruction is Observed-by that PE, to the extent that the PE is required to observe the access coherently. In addition, the use of a Load-Acquire, Load-AcquirePC or a Store-Release instruction on accesses to a Memory-mapped peripheral introduces order between the Memory effects of the instructions that access that peripheral, as described in the definition of Peripheral coherence order.

Load-Acquire, Load-AcquirePC and Store-Release, other than Load-Acquire Exclusive Pair and Store-Release-Exclusive Pair, access only a single data element. This access is single-copy atomic. The address of the data object must be aligned to the size of the data element being accessed, otherwise the access generates an Alignment fault.

Load-Acquire Exclusive Pair and Store-Release Exclusive Pair access two data elements. The address supplied to the instructions must be aligned to twice the size of the element being loaded, otherwise the access generates an Alignment fault.

A Store-Release Exclusive instruction has the release semantics only if the store is successful.

--- Note ---

- Each Load-Acquire Exclusive and Store-Release Exclusive instruction is essentially a variant of the equivalent Load-Exclusive or Store-Exclusive instruction. All usage restrictions and single-copy atomicity properties:
  - That apply to the Load-Exclusive instructions also apply to the Load-Acquire Exclusive instructions.
  - That apply to the Store-Exclusive instructions also apply to the Store-Release Exclusive instructions.

- The Load-Acquire, Load-AcquirePC, and Store-Release instructions can remove the requirement to use the explicit DMB instruction.

**LoadLOAcquire, StoreLORelease**

For each PE, the Non-secure physical memory map is divided into a set of LORegions using a table that is held within the PE. Any PA in the Non-secure memory map can be a member of one LORegion. If a PA is assigned to more than one LORegion, then an implementation might treat it as if it has been assigned to fewer LORegions than that have been specified. A PA in the Secure physical memory map cannot be a member of any LORegion. For more information, see Limited ordering regions on page B2-182.

**FEAT_LOR** provides a set of instructions with Acquire semantics for loads, and Release semantics for stores that apply in relation to the defined LORegions. The new variants of the Load-Acquire and Store-Release instructions are LoadLOAcquire and StoreLORelease. See LoadLOAcquire/StoreLORelease on page C3-263.

For all memory types, these instructions have the following ordering requirements:

- LoadLOAcquire has the same semantics as Load-Acquire except that the memory accesses affected lie within the same LORegion as the address of the memory access generated by the LoadLOAcquire instruction. See Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.
StoreLORelease has the same semantics as Store-Release except that the memory accesses affected lie within the same LORegion as the address of the memory access generated by the StoreLORelease instruction. See Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

In addition, for accesses to Memory-mapped peripherals:

- LoadLOAcquire has the same semantics as Load-Acquire except that the affected Memory effects of instructions that access the peripheral lie within the same LORegion as the address of the memory access generated by the LoadLOAcquire instruction. See Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.
- StoreLORelease has the same semantics as Store-Release except that the affected Memory effects of instructions that access the peripheral lie within the same LORegion as the address of the memory access generated by the StoreLORelease instruction. See Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

Note

The LoadLOAcquire/StoreLORelease instructions can remove the requirement to use the explicit DMB instruction.

B2.3.12 Limited ordering regions

FEAT_LOR introduces limited ordering regions (LORegions), which allow large systems to perform special load-acquire and store-release instructions that provide order between the memory accesses to a region of the PA map as observed by a set of observers.

This feature is supported in AArch64 state only.

Specification of the LORegions

The LORegions are defined in the Non-secure physical memory map using a set of LORegion descriptors. The number of LORegion descriptors is IMPLEMENTATION DEFINED, and can be discovered by reading the LORID_EL1 register.

Each LORegion descriptor consists of:

- A tuple of the following values:
  - A Start Address.
  - An End Address.
  - An LORegion Number.
- Valid bit which indicates whether that LORegion descriptor is valid.

A memory location lies within the LORegion identified by the LORegion Number if the PA lies between the Start Address and the End Address, inclusive. The Start Address must be defined to be aligned to 64KB and the End Address must be defined as the top byte of a 64KB block of memory.

The LORegion descriptors are programmed using the LORSA_EL1, LOREA_EL1, LORN_EL1, and LORC_EL1 registers in the System register space. These registers describe only memory addresses in the Non-secure memory map. These registers are UNDEFINED if accessed when SCR_EL3.NS == 0.

If a LoadLOAcquire or a StoreLORelease does not match with any LORegion, then:

- The LoadLOAcquire will behave as a Load-Acquire, and will be ordered in the same way with respect to all accesses, independent of their LORegions.
- The StoreLORelease will behave as a Store-Release, and will be ordered in the same way with respect to all accesses, independent of their LORegions.

Note

If no LORegions are implemented, then the LoadLOAcquire and StoreLORelease will therefore behave as a Load-Acquire and Store-Release.
A new access type `AccType_LIMITEDORDERED` has been added for these limited ordering instructions to be identified.
B2.4 Caches and memory hierarchy

The implementation of a memory system depends heavily on the microarchitecture and therefore many details of the memory system are IMPLEMENTATION DEFINED. The Arm architecture defines the application level interface to the memory system, including a hierarchical memory system with multiple levels of cache. This section describes an application level view of this system. It contains the subsections:

- Introduction to caches.
- Memory hierarchy.
- Application level access to functionality related to caches on page B2-185
- Implication of caches for the application programmer on page B2-186.
- Preloading caches on page B2-188.

B2.4.1 Introduction to caches

A cache is a block of high-speed memory that contains a number of entries, each consisting of:

- Main memory address information, commonly known as a tag.
- The associated data.

Caches increase the average speed of a memory access. Caching takes account of two principles of locality:

Spatial locality

An access to one Location is likely to be followed by accesses to adjacent Locations. Examples of this principle are:

- Sequential instruction execution.
- Accessing a data structure.

Temporal locality

An access to an area of memory is likely to be repeated in a short time period. An example of this principle is the execution of a software loop.

To minimize the quantity of control information stored, the spatial locality property groups several locations together under the same tag. This logical block is commonly known as a cache line. When data is loaded into a cache, access times for subsequent loads and stores are reduced, resulting in overall performance benefits. An access to information already in a cache is known as a cache hit, and other accesses are called cache misses.

Normally, caches are self-managing, with the updates occurring automatically. Whenever the PE accesses a cacheable memory location, the cache is checked. If the access is a cache hit, the access occurs in the cache. Otherwise, the access is made to memory. The Arm architecture permits different cache topologies and access policies, provided they comply with the memory coherency model described in this manual.

Caches introduce a number of potential problems, mainly because:

- Memory accesses can occur at times other than when the programmer would expect them.
- A data item can be held in multiple physical locations.

B2.4.2 Memory hierarchy

Typically memory close to a PE has very low latency, but is limited in size and expensive to implement. Further from the PE it is common to implement larger blocks of memory but these have increased latency. To optimize overall performance, an Armv8 memory system can include multiple levels of cache in a hierarchical memory system that exploits this trade-off between size and latency. Figure B2-1 on page B2-185 shows an example of such a system in an Armv8-A system that supports virtual addressing.
The AArch64 Application Level Memory Model

B2.4 Caches and memory hierarchy

Figure B2-1  Multiple levels of cache in a memory hierarchy

--- Note ---

In this manual, in a hierarchical memory system, Level 1 refers to the level closest to the processing element, as shown in Figure B2-1.

Instructions and data can be held in separate caches or in a unified cache. A cache hierarchy can have one or more levels of separate instruction and data caches, with one or more unified caches that are located at the levels closest to the main memory. Memory coherency for cache topologies can be defined using the conceptual points Point of Unification (PoU), Point of Coherency (PoC), Point of Persistence (PoP), and Point of Deep Persistence (PoDP).

For more information, including the definitions of PoU, PoC, PoP, and PoDP, see About cache maintenance in AArch64 state on page D7-5050.

If FEAT_MTE2 is implemented, the behavior of cache maintenance instructions is modified. For more information, see Allocation Tags on page D9-5221.

The cacheability and shareability memory attributes

Cacheability and shareability are two attributes that describe the memory hierarchy in a multiprocessing system:

**Cacheability**

This attribute defines whether memory locations are allowed to be allocated into a cache or not. Cacheability is defined independently for Inner and Outer Cacheability locations.

**Shareability**

This attribute defines whether memory locations are shareable between different agents in a system. Marking a memory location as shareable for a particular domain requires hardware to ensure that the location is coherent for all agents in that domain. Shareability is defined independently for Inner and Outer Shareability domains.

For more information about Cacheability and Shareability, see Memory types and attributes on page B2-196.

B2.4.3   Application level access to functionality related to caches

As indicated in About the Application level programmers' model on page B1-136, the application level corresponds to execution at EL0. The architecture defines a set of cache maintenance instructions that software can use to manage cache coherency. Software executing at a higher Exception level can enable use of some of this functionality from EL0, as follows:

**When the value of SCTLR_EL1.UCI is 1**

Software executing at EL0 can access:

- The data cache maintenance instructions, DC CVAU, DC CVAC, DC CVAP, DC CVADP, and DC CIVAC. See The data cache maintenance instruction (DC) on page D7-5056.
• The instruction cache maintenance instruction IC IVAU. See The instruction cache maintenance instruction (IC) on page D7-5056.

Attempted execution of these instructions might generate a Permission fault as described in Permission fault on page D8-5183.

When the value of SCTLR_EL1.UCT is 1

Software executing at EL0 can access the cache type register. See CTR_EL0.

When the value of SCTLR_EL1.DZE is 1

Software executing at EL0 can access the data cache zero instruction DC ZVA. See Data cache zero instruction on page D7-5068.

The SCTLR_EL1.{UCI, UCT, DZE} control fields are accessible only by software executing at EL1 or higher. When HCR_EL2.{E2H, TGE} == {1,1}, the controls {UCI, UCT and DZE} are found in SCTLR_EL2.

This functionality is UNDEFINED at EL0 when the value of the corresponding SCTLR_EL1 control field is 0.

B2.4.4 Implication of caches for the application programmer

In normal operation, the caches are largely invisible to the application programmer. However they can become visible when there is a breakdown in the coherency of the caches. Such a breakdown can occur:

• When memory locations are updated by other agents in the system that do not use hardware management of coherency.
• When memory updates made from the application software must be made visible to other agents in the system, without the use of hardware management of coherency.

For example:

• In the absence of hardware management of coherency of DMA accesses, in a system with a DMA controller that reads memory locations that are held in the data cache of a PE, a breakdown of coherency occurs when the PE has written new data in the data cache, but the DMA controller reads the old data held in memory.
• In a Harvard cache implementation, where there are separate instruction and data caches, a breakdown of coherency occurs when new instruction data has been written into the data cache, but the instruction cache still contains the old instruction data.

Data coherency issues

Software can ensure the data coherency of caches in the following ways:

• By not using the caches in situations where coherency issues can arise. This can be achieved by:
  — Using Non-cacheable or, in some cases, Write-Through Cacheable memory.
  — Not enabling caches in the system.

• By using cache maintenance instructions to manage the coherency issues in software. See Application level access to functionality related to caches on page B2-185.

• By using hardware coherency mechanisms to ensure the coherency of data accesses to memory for cacheable locations by observers within the different shareability domains, see Non-shareable Normal memory on page B2-198 and Shareable, Inner Shareable, and Outer Shareable Normal memory on page B2-197.

——— Note ————

The performance of these hardware coherency mechanisms is highly implementation-specific. In some implementations, the mechanism suppresses the ability to cache shareable locations. In other implementations, cache coherency hardware can hold data in caches while managing coherency between observers within the shareability domains.
Not all these mechanisms are directly available to software operating at EL0 and might involve interaction with software operating at a higher Exception level.

### Synchronization and coherency issues between data and instruction accesses

How far ahead of the current point of execution instructions are fetched from is IMPLEMENTATION DEFINED. Such prefetching can be either a fixed or a dynamically varying number of instructions, and can follow any or all possible future execution paths. For all types of memory:

- The PE might have fetched the instructions from memory at any time since the last Context synchronization event on that PE.
- Any instructions fetched in this way might be executed multiple times, if this is required by the execution of the program, without being refetched from memory. In the absence of a Context synchronization event, there is no limit on the number of times such an instruction might be executed without being refetched from memory.

The Arm architecture requires the hardware to ensure coherency between instruction caches and memory, even for locations of shared memory. A write has been made coherent with an instruction fetch of a shareability domain when:

- **CTR_EL0.{DIC, IDC} == {0, 0}**
  - The location written to has been cleaned to the Point of unification (PoU) from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

- **CTR_EL0.{DIC, IDC} == {1, 0}**
  - The location written to has been cleaned to the Point of unification (PoU) from the data cache, and that clean is complete for the shareability domain.

- **CTR_EL0.{DIC, IDC} == {0, 1}**
  - The write is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

- **CTR_EL0.{DIC, IDC} == {1, 1}**
  - The write is complete for the shareability domain.

If software requires coherency between instruction execution and memory, it must manage this coherency using Context synchronization events and cache maintenance instructions. The following code sequence can be used to allow a PE to execute code that the same PE has written.

```assembly
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <wt> containing a new 32-bit instruction, to be held in Cacheable space at a location pointed to by Xn.

STR Wt, [Xn]  ; Coherency example for data and instruction accesses within the same Inner Shareable domain.
DC CVAU, Xn    ; Clean data cache by VA to point of unification (PoU)
DSB ISH        ; Ensure visibility of the data cleaned from cache
IC IVAU, Xn    ; Invalidate instruction cache by VA to PoU
DSB ISH        ; Ensure completion of the invalidations
ISB             ; Synchronize the fetched instruction stream

Note

- If this sequence is not executed between writing data to a location and executing the instruction at that location, the lack of coherency between instruction caches and memory means that the instructions that are executed might be the old instruction or the updated instruction, and which is used can arbitrarily vary during execution. It must not be assumed by software, before the synchronization sequence is executed, that when the updated instruction has been seen, the old instruction will not be seen again.
• For Non-cacheable or Write-Through accesses, the clean data cache by VA instruction is not required. However, the invalidate instruction cache instruction is required because the Armv8-A AArch64 architecture allows Non-cacheable accesses to be held in an instruction cache. See Non-cacheable accesses and instruction caches on page D7-5049.

• This code can be used when the thread of execution modifying the code is the same thread of execution that is executing the code. The Arm architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization. See Concurrent modification and execution of instructions on page B2-156.

• The system software controls whether these cache maintenance instructions are available to the application level by setting SCTLR_EL1.UCI.

### B2.4.5 Preloading caches

The Arm architecture provides memory system hints PRFM, LDNP, and STNP that software can use to communicate the expected use of memory locations to the hardware. The memory system can respond by taking actions that are expected to speed up the memory accesses if they occur. The effect of these memory system hints is IMPLEMENTATION DEFINED. Typically, implementations use this information to bring the data or instruction locations into caches.

The Preload instructions are hints, and so implementations can treat them as NOPs without affecting the functional behavior of the device. The instructions cannot generate synchronous Data Abort exceptions, but the resulting memory system operations might, under exceptional circumstances, generate an asynchronous External abort, which is taken using an SError interrupt exception. For more information, see ISS encoding for an exception from a Data Abort on page D17-5685.

`PrefetchHint()` defines the prefetch hint types.

The `Hint_Prefetch()` function signals to the memory system that memory accesses of the type hint to or from the specified address are likely to occur in the near future. The memory system might take some action to speed up the memory accesses when they do occur, such as preloading the specified address into one or more caches as indicated by the innermost cache level target and non-temporal hint stream.

For more information on PRFM and load/store instructions that provide hints to the memory system, see Prefetch memory on page C3-267 and Load/store SIMD and floating-point non-temporal pair on page C3-265. For more information on SVE PRF* instructions that provide hints to the memory system, see Predicated non-contiguous element accesses on page C3-293.
B2.5 Alignment support

This section describes alignment support. It contains the following subsections:

- Instruction alignment.
- Alignment of data accesses.

B2.5.1 Instruction alignment

A64 instructions must be word-aligned.

Attempting to fetch an instruction from a misaligned location results in a PC alignment fault. See PC alignment checking on page D1-4668.

B2.5.2 Alignment of data accesses

An unaligned access to any type of Device memory causes an Alignment fault.

Unaligned accesses to Normal memory

The behavior of unaligned accesses to Normal memory is dependent on all of the following:

- The instruction causing the memory access.
- The memory attributes of the accessed memory.
- The value of SCTLR_ELx.{A, nAA}.
- Whether or not FEAT_LSE2 is implemented.

Load or Store of Single or Multiple registers

For all instructions that load or store single or multiple registers, but not Load-Exclusive, Store-Exclusive, Load-Acquire/Store-Release, Atomic, and SETG* Memory Copy and Memory Set instructions, if the address that is accessed is not aligned to the size of the data element being accessed, then:

When the value of SCTLR_ELx.A applicable to the current Exception level is 1, an Alignment fault is generated.

When the value of SCTLR_ELx.A applicable to the current Exception level is 0:

- An unaligned access is performed.
- If FEAT_LSE2 is not implemented, the access is not guaranteed to be single-copy atomic except at the byte access level.
- If FEAT_LSE2 is implemented:
  - If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable memory, the memory access is single-copy atomic. For LDNP, LDP, or STP instructions, the entire memory access will be single-copy atomic.
  - If all the bytes of the memory accessed do not lie within a 16-byte quantity aligned to 16 bytes, or the access is not to Normal Inner Write-Back, Outer Write-Back Cacheable memory, the access is not guaranteed to be single-copy atomic except at the byte access level.

For these instructions, the definition of an unaligned access is based on the size of the accessed elements, not the overall size of the memory access. This affects SIMD and SVE element and structure loads and stores, and also load/store pair instructions.

For predicated SVE vector element and structure load or store instructions, alignment checks are based on the memory element access size, not on the vector element size.

For predicated SVE vector element and structure load or store instructions, Inactive elements cannot cause an Alignment fault.

For unpredicated SVE vector register load or store instructions, the base address is checked for 16-byte alignment.

For unpredicated SVE predicate register load or store instructions, the base address is checked for 2-byte alignment.
Load-Exclusive/Store-Exclusive and Atomic instructions

For Load-Exclusive/Store-Exclusive, and Atomic instructions including those with acquire or acquire-release semantics:

When the value of SCTLR_ELx.A applicable to the current Exception level is 1, an Alignment fault is generated.

When the value of SCTLR_ELx.A applicable to the current Exception level is 0:

If FEAT_LSE2 is not implemented, these instructions generate an Alignment fault if the address being accessed is not aligned to the size of the data structure being accessed.

If FEAT_LSE2 is implemented, then:

- If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable memory, an unaligned access is performed.
- If all the bytes of the memory access do not lie within a 16-byte quantity aligned to 16 bytes, or the memory access is not to Normal Inner Write-Back, Outer Write-Back Cacheable memory, then it is a CONSTRAINED UNPREDICTABLE choice of either of the following:
  - An unaligned access is performed meeting all of the semantics of the instruction.
  - An Alignment fault is generated.

Where memory access is performed, then it is single-copy atomic.

For these instructions, the definition of an unaligned access is based on the overall access size.

If FEAT_LS64 is implemented, when a single-copy atomic 64-byte instruction accesses a memory location that is not aligned to 64 bytes, an Alignment fault always occurs, regardless of the value of SCTLR_ELx.A.

Non-atomic Load-Acquire/Store-Release instructions

For Load-Acquire/Store-Release instructions that do not have exclusive or atomic behaviors:

When the value of SCTLR_ELx.A applicable to the current Exception level is 1, an Alignment fault is generated.

When the value of SCTLR_ELx.A applicable to the current Exception level is 0:

If FEAT_LSE2 is not implemented, then these instructions generate an Alignment fault if the address being accessed is not aligned to the size of the data structure being accessed.

If FEAT_LSE2 is implemented, then:

- If the memory access is not to Normal Inner Write-Back or Outer Write-Back Cacheable memory, then it is a CONSTRAINED UNPREDICTABLE choice of either of the following:
  - An unaligned access is performed meeting all of the semantics of the instruction.
  - An Alignment fault is generated.
- If all of the bytes of the memory access do not lie within a 16-byte quantity aligned to 16 bytes then the following applies:
  - If SCTLR_ELx.nAA applicable to the current Exception level is 0, an Alignment fault is generated.
  - If SCTLR_ELx.nAA applicable to the current Exception level is 1, then an unaligned access is performed which is not guaranteed to be single-copy atomic except at the byte access level.

In this case, the architecture does no define the order of the different transactions of the access defined by the single instructions relative to each other.

- If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable memory, an unaligned access meeting all the semantics of the instruction is performed.

Note

- Unaligned accesses typically take additional cycles to complete compared to a naturally-aligned access.
- An operation that is not single-copy atomic above the byte level can abort on any memory access that it makes and can abort on more than one access. This means that an unaligned access that occurs across a page boundary can generate an abort on either side of the page boundary.
Memory Copy and Memory Set instructions

For SETG* instructions:
• There is an alignment check regardless of the value of SCTLR_ELx.A.
• If Xn is not a multiple of 16, an Alignment fault is generated.
• If Xd is not aligned to a multiple of 16, an Alignment fault is generated.

For more information, see the individual SETG* instruction descriptions.
B2.6 Endian support

General description of endianness in the Arm architecture describes the relationship between endianness and memory addressing in the Arm architecture.

The following subsections then describe the endianness schemes supported by the architecture:
• Instruction endianness on page B2-193.
• Data endianness on page B2-193.
• Endianness of memory-mapped peripherals on page B2-195.

B2.6.1 General description of endianness in the Arm architecture

This section describes only memory addressing and the effects of endianness for data elements up to quadwords of 128 bits. However, this description can be extended to apply to larger data elements.

For an address A, Figure B2-2 shows, for big-endian and little-endian memory systems, the relationship between:
• The quadword at address A.
• The doubleword at address A and A+8.
• The words at addresses A, A+4, A+8, and A+12.
• The halfwords at addresses A, A+2, A+4, A+6, A+8, A+10, A+12, and A+14.
• The bytes at addresses A, A+1, A+2, A+3, A+4, A+5, A+6, A+7, A+8, A+9, A+10, A+11, A+12, A+13, A+14, and A+15.

The terms in Figure B2-2 have the following definitions:
B_A Byte at address A.
HW_A Halfword at address A.
MSByte Most significant byte.
LSByte Least significant byte.

Figure B2-2 Endianness relationships
The big-endian and little-endian mapping schemes determine the order in which the bytes of a quadword, doubleword, word, or halfword are interpreted. For example, a load of a word from address 0x1000 always results in an access to the bytes at memory locations 0x1000, 0x1001, 0x1002, and 0x1003. The endianness mapping scheme determines the significance of these 4 bytes.

B2.6.2 Instruction endianness

A64 instructions have a fixed length of 32 bits and are always little-endian.

B2.6.3 Data endianness

SCTLR_EL1.E0E, configurable at EL1 or higher, determines the data endianness for execution at EL0.

The data size used for endianness conversions:

- Is the size of the data value that is loaded or stored for SIMD and floating-point register and general-purpose register loads and stores.
- Is the size of the data element that is loaded or stored for SIMD element and data structure loads and stores.

For more information, see Endianness in SIMD operations on page B2-194.

Note

This means the Armv8 architecture introduces a requirement for 128-bit endian conversions.

Instructions to reverse bytes in a general-purpose register, a SIMD and floating-point register, or an SVE register

An application or device driver might have to interface to memory-mapped peripheral registers or shared memory structures that are not the same endianness as the internal data structures. Similarly, the endianness of the operating system might not match that of the peripheral registers or shared memory. In these cases, the PE requires an efficient method to transform explicitly the endianness of the data.

Table B2-2 on page B2-193 shows the instructions that provide this functionality:

<table>
<thead>
<tr>
<th>Function</th>
<th>Instructions</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reverse bytes in 32-bit word or words&lt;sup&gt;a&lt;/sup&gt;</td>
<td>REV32</td>
<td>For use with general-purpose registers</td>
</tr>
<tr>
<td>Reverse bytes in whole register</td>
<td>REV</td>
<td>For use with general-purpose registers</td>
</tr>
<tr>
<td>Reverse bytes in 16-bit halfwords</td>
<td>REV16</td>
<td>For use with general-purpose registers</td>
</tr>
<tr>
<td>Reverse elements in doublewords, vector</td>
<td>REV64</td>
<td>For use with SIMD and floating-point registers</td>
</tr>
<tr>
<td>Reverse elements in words, vector</td>
<td>REV32</td>
<td>For use with SIMD and floating-point registers</td>
</tr>
<tr>
<td>Reverse elements in halfwords, vector</td>
<td>REV16</td>
<td>For use with SIMD and floating-point registers</td>
</tr>
<tr>
<td>Reverse bytes/halfwords/words within elements, predicated</td>
<td>REVB, REVH, REVW</td>
<td>For use with SVE registers</td>
</tr>
</tbody>
</table>

<sup>a</sup> Can operate on multiple words.
Endianness in SIMD operations

SIMD element load/store instructions transfer vectors of elements between memory and the SIMD and floating-point register file. An instruction specifies both the length of the transfer and the size of the data elements being transferred. This information is used to load and store data correctly in both big-endian and little-endian systems.

For example:

```
LD1 {V0.4H}, [X1]
```

This loads a 64-bit register with four 16-bit values. The four elements appear in the register in array order, with the lowest indexed element fetched from the lowest address. The order of bytes in the elements depends on the endianness configuration, as shown in Figure B2-3. Therefore, the order of the elements in the registers is the same regardless of the endianness configuration.

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>C[7:0]</td>
<td>5</td>
<td>C[15:8]</td>
<td>6</td>
<td>D[7:0]</td>
<td>7</td>
<td>D[15:8]</td>
</tr>
</tbody>
</table>

Memory system with little-endian addressing (LE)

Memory system with big-endian addressing (BE)
```

Figure B2-3 SIMD byte order example

The `BigEndian()` pseudocode function determines the current endianness of the data.

The `BigEndianReverse()` pseudocode function reverses the endianness of a bitstring.

The `BigEndian()` and `BigEndianReverse()` functions are defined in Chapter J1 Armv8 Pseudocode.

Endianness in SVE operations

Rules on byte and element order of SIMD load and store instructions apply to SVE load and store instructions.

Additional rules apply to the data endianness of memory accesses performed by SVE load and store instructions.

For predicated SVE vector element and structure load and store instructions, an endianness conversion is performed using the memory element access size. The size of the vector element is not used in endianness conversion.

For unpredicated SVE vector register load and store instructions, the vector byte elements are transferred in increasing element number order without any endianness conversion.

For unpredicated SVE predicate register load and store instructions, each 8 bits from the predicate are transferred as a byte in increasing element number order without any endianness conversion.

When an SVE load instruction is executed, endianness conversion occurs before any sign-extension or zero-extension into a vector element.

When an SVE store instruction is executed, endianness conversion occurs after any truncation from the vector element to the memory element access size.
B2.6.4 Endianness of memory-mapped peripherals

All memory-mapped peripherals defined in the Arm architecture must be little-endian.

Peripherals to which this requirement applies include:

- Memory-mapped register interfaces to a debugger, or to a Cross Trigger Interface, see Chapter H8 About the External Debug Registers.
- The memory-mapped register interface to the system level implementation of the Generic Timer, see Chapter I2 System Level Implementation of the Generic Timer.
- A memory-mapped register interface to the Performance Monitors, see Chapter I3 Recommended External Interface to the Performance Monitors.
- A memory-mapped register interface to the Activity Monitors, see Chapter I4 Recommended External Interface to the Activity Monitors.
- Memory-mapped register interfaces to an Arm Generic Interface Controller, see the ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0.
- The memory-mapped register interface to an Arm trace component. See, for example, the ARM® Embedded Trace Macrocell Architecture Specification, ETMv4.
B2.7 Memory types and attributes

The ordering of accesses for addresses in memory, referred to as the memory order model, is defined by the memory attributes. The following sections describe this model:

- Normal memory.
- Device memory on page B2-200.
- Memory access restrictions on page B2-207.

B2.7.1 Normal memory

The Normal memory type attribute applies to most memory in a system. It indicates that the hardware is permitted by the architecture to perform Speculative data read accesses to these locations, regardless of the access permissions for these locations.

The Normal memory type has the following properties:

- A write to a memory location with the Normal attribute completes in finite time.
- Writes to a memory location with the Normal memory type that is either Non-cacheable or Write-Through cacheable for both the Inner and Outer cacheability must reach the endpoint for that location in the memory system in finite time. Two writes to the same location, where at least one is using the Normal memory type, might be merged before they reach the endpoint unless there is an ordered-before relationship between the two writes.
- Unaligned memory accesses can access Normal memory if the system is configured to generate such accesses.
- There is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register load/store instructions. See Multi-register loads and stores that access Normal memory on page B2-200.

**Note**

- The Normal memory attribute is appropriate for locations of memory that are idempotent, meaning that they exhibit all of the following properties:
  - Read accesses can be repeated with no side-effects.
  - Repeated read accesses return the last value written to the resource being read.
  - Read accesses can fetch additional memory locations with no side-effects.
  - Write accesses can be repeated with no side-effects if the contents of the location accessed are unchanged between the repeated writes or as the result of an exception, as described in this section.
  - Unaligned accesses can be supported.
  - Accesses can be merged before accessing the target memory system.

- Normal memory allows speculative reads and may be affected by intermediate buffering and forwarding of data. If non-idempotent memory locations are mapped as Normal memory, the following may occur:
  - Memory accesses return UNKNOWN values.
  - UNPREDICTABLE effects on memory-mapped peripherals.

- An instruction that generates a sequence of accesses as described in Atomicity in the Arm architecture on page B2-154 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore, one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

For accesses to Normal memory, a DMB instruction is required to ensure the required ordering.

The following sections describe the other attributes for Normal memory:

- Shareable Normal memory on page B2-197.
• Non-shareable Normal memory on page B2-198.
• Cacheability attributes for Normal memory on page B2-198.

See also:
• Multi-register loads and stores that access Normal memory on page B2-200.
• Atomicity in the Arm architecture on page B2-154.
• Memory barriers on page B2-174.
• Concurrent modification and execution of instructions on page B2-156.

Shareable Normal memory

A Normal memory location has a Shareability attribute that is one of:

• Inner Shareable, meaning it applies across the Inner Shareable shareability domain.
• Outer Shareable, meaning it applies across both the Inner Shareable and the Outer Shareable shareability domains.
• Non-shareable.

The shareability attributes define the data coherency requirements of the location, which hardware must enforce. They do not affect the coherency requirements of instruction fetches, see Synchronization and coherency issues between data and instruction accesses on page B2-187.

Note

• System designers can use the shareability attribute to specify the locations in Normal memory for which coherency must be maintained. However, software developers must not assume that specifying a memory location as Non-shareable permits software to make assumptions about the incoherency of the location between different PEs in a shared memory system. Such assumptions are not portable between different multiprocessing implementations that might use the shareability attribute. Any multiprocessing implementation might implement caches that are shared, inherently, between different processing elements.
• This architecture assumes that all PEs that use the same operating system or hypervisor are in the same Inner Shareable shareability domain.

Shareable, Inner Shareable, and Outer Shareable Normal memory

The Arm architecture abstracts the system as a series of Inner and Outer Shareability domains.

Each Inner Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Inner Shareable attribute made by any member of that set.

Each Outer Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Outer Shareable attribute made by any member of that set.

The following properties also hold:

• Each observer is a member of only a single Inner Shareability domain.
• Each observer is a member of only a single Outer Shareability domain.
• All observers in an Inner Shareability domain are always members of the same Outer Shareability domain. This means that an Inner Shareability domain is a subset of an Outer Shareability domain, although it is not required to be a proper subset.

Note

• Because all data accesses to Non-cacheable locations are data coherent to all observers, Non-cacheable locations are always treated as Outer Shareable.
• The Inner Shareable domain is expected to be the set of PEs controlled by a single hypervisor or operating system.

The details of the use of the shareability attributes are system-specific. Example B2-1 on page B2-198 shows how they might be used.
Example B2-1 Use of shareability attributes

In an implementation, a particular subsystem with two clusters of PEs has the requirement that:

- In each cluster, the data caches or unified caches of the PEs in the cluster are transparent for all data accesses to memory locations with the Inner Shareable attribute.
- However, between the two clusters, the caches:
  - Are not required to be coherent for data accesses that have only the Inner Shareable attribute.
  - Are coherent for data accesses that have the Outer Shareable attribute.

In this system, each cluster is in a different shareability domain for the Inner Shareable attribute, but all components of the subsystem are in the same shareability domain for the Outer Shareable attribute.

A system might implement two such subsystems. If the data caches or unified caches of one subsystem are not transparent to the accesses from the other subsystem, this system has two Outer Shareable shareability domains.

Having two levels of shareability means system designers can reduce the performance and power overhead for shared memory locations that do not need to be part of the Outer Shareable shareability domain.

For shareable Normal memory, the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer in the same Shareability domain.

**Non-shareable Normal memory**

For Normal memory locations, the Non-shareable attribute identifies Normal memory that is likely to be accessed only by a single PE.

A location in Normal memory with the Non-shareable attribute does not require the hardware to make data accesses by different observers coherent, unless the memory is Non-cacheable. For a Non-shareable location, if other observers share the memory system, software must use cache maintenance instructions, if the presence of caches might lead to coherency issues when communicating between the observers. This cache maintenance requirement is in addition to the barrier operations that are required to ensure memory ordering.

For Non-shareable Normal memory, it is IMPLEMENTATION DEFINED whether the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer.

**Cacheability attributes for Normal memory**

In addition to being Outer Shareable, Inner Shareable or Non-shareable, each region of Normal memory is assigned a Cacheability attribute that is one of:

- Write-Through Cacheable.
- Write-Back Cacheable.
- Non-cacheable.

Also, for Write-Through Cacheable and Write-Back Cacheable Normal memory regions:

- A region might be assigned cache allocation hints for read and write accesses.
- It is IMPLEMENTATION DEFINED whether the cache allocation hints can have an additional attribute of Transient or Non-transient.

For more information, see *Cacheability, cache allocation hints, and cache transient hints on page D7-5046.*

A memory location can be marked as having different cacheability attributes, for example when using aliases in a VA to PA mapping:

- If the attributes differ only in the cache allocation hint, this does not affect the behavior of accesses to that location.
- For other cases, see *Mismatched memory attributes on page B2-208.*
The cacheability attributes provide a mechanism of coherency control with observers that lie outside the shareability domain of a region of memory. In some cases, the use of Write-Through Cacheable or Non-cacheable regions of memory might provide a better mechanism for controlling coherency than the use of hardware coherency mechanisms or the use of cache maintenance routines. To this end, the architecture requires the following properties for Non-cacheable or Write-Through Cacheable memory:

• A completed write to a memory location that is Non-cacheable or Write-Through Cacheable for a level of cache made by an observer accessing the memory system inside the level of cache is visible to all observers accessing the memory system outside the level of cache without the need of explicit cache maintenance.

• A completed write to a memory location that is Non-cacheable for a level of cache made by an observer accessing the memory system outside the level of cache is visible to all observers accessing the memory system inside the level of cache without the need of explicit cache maintenance.

• For accesses to Normal memory that is Non-cacheable, a DM B instruction introduces a Barrier-ordered-before relation on all accesses to a single peripheral or block of memory that is of IMPLEMENTATION DEFINED size. For more information, see Ordering relations on page B2-163.

Note
Implementations can use the cache allocation hints to indicate a probable performance benefit of caching. For example, a programmer might know that a piece of memory is not going to be accessed again and would be better treated as Non-cacheable. The distinction between memory regions with attributes that differ only in the cache allocation hints exists only as a hint for performance.

For Normal memory, the Arm architecture provides cacheability attributes that are defined independently for each of two conceptual levels of cache, the inner and the outer cache. The relationship between these conceptual levels of cache and the implemented physical levels of cache is IMPLEMENTATION DEFINED, and can differ from the boundaries between the Inner and Outer Shareability domains. However:

• Inner refers to the innermost caches, meaning the caches that are closest to the PE, and always includes the lowest level of cache.

• No cache that is controlled by the Inner cacheability attributes can lie outside a cache that is controlled by the Outer cacheability attributes.

• An implementation might not have any outer cache.

Example B2-2, Example B2-3 on page B2-200, and Example B2-4 on page B2-200 describe the possible ways of implementing a system with three levels of cache, level 1 (L1) to level 3 (L3).

Note

• L1 cache is the level closest to the PE, see Memory hierarchy on page B2-184.

• When managing coherency, system designs must consider both the inner and outer cacheability attributes, as well as the shareability attributes. This is because hardware might have to manage the coherency of caches at one conceptual level, even when another conceptual level has the Non-cacheable attribute.

Example B2-2 Implementation with two inner and one outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

• The Inner cacheability attribute applied to L1 and L2 cache.

• The Outer cacheability attribute applied to L3 cache.
Example B2-3 Implementation with three inner and no outer cache levels

Implement the three levels of cache in the system, L1 to L3, with the Inner cacheability attribute applied to L1, L2, and L3 cache. Do not use the Outer cacheability attribute.

Example B2-4 Implementation with one inner and two outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:
- The Inner cacheability attribute applied to L1 cache.
- The Outer cacheability attribute applied to L2 and L3 cache.

Multi-register loads and stores that access Normal memory

For all instructions that load or store more than one general-purpose register from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register from an Exception level the order in which the registers are accessed is not defined by the architecture.

For all instructions that load or store one or more SVE or SIMD&FP registers from an Exception level, there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load or store instructions.

B2.7.2 Device memory

The Device memory type attributes define memory locations where an access to the location can cause side-effects, or where the value returned for a load can vary depending on the number of loads performed. Typically, the Device memory attributes are used for memory-mapped peripherals and similar locations.

The attributes for Armv8 Device memory are:

Gathering      Identified as G or nG, see Gathering on page B2-203.
Reordering     Identified as R or nR, see Reordering on page B2-204.
Early Write Acknowledgement 
Identified as E or nE, see Early Write Acknowledgement on page B2-205.

The Armv8 Device memory types are:

Device-nGnRnE  Device non-Gathering, non-Reordering, No Early Write Acknowledgement. Equivalent to the Strongly-ordered memory type in earlier versions of the architecture.
Device-nGnRE   Device non-Gathering, non-Reordering, Early Write Acknowledgement. Equivalent to the Device memory type in earlier versions of the architecture.
Device-nGRE    Device non-Gathering, Reordering, Early Write Acknowledgement. Armv8 adds this memory type to the translation table formats found in earlier versions of the architecture. The use of barriers is required to order accesses to Device-nGRE memory.
Device-GRE     Device Gathering, Reordering, Early Write Acknowledgement. Armv8 adds this memory type to the translation table formats found in earlier versions of the architecture. Device-GRE memory has the fewest constraints. It behaves similar to Normal memory, with the restriction that Speculative accesses to Device-GRE memory is forbidden.
Collectively these are referred to as any Device memory type. Going down the list, the memory types are described as getting weaker; conversely the going up the list the memory types are described as getting stronger.

--- Note ---

- As the list of types shows, these additional attributes are hierarchical. For example, a memory location that permits Gathering must also permit Reordering and Early Write Acknowledgement.
- The architecture does not require an implementation to distinguish between each of these memory types and Arm recognizes that not all implementations will do so. The subsection that describes each of the attributes, describes the implementation rules for the attribute.

All of these memory types have the following properties:

- Speculative data accesses are not permitted to any memory location with any Device memory attribute. This means that each memory access to any Device memory type must be one that would be generated by a simple sequential execution of the program.

The following exceptions to this apply:

- Reads generated by the SIMD and floating-point instructions can access bytes that are not explicitly accessed by the instruction if the bytes accessed are in a 16-byte window, aligned to 16-bytes, that contains at least one byte that is explicitly accessed by the instruction.
- For reads, including hardware speculation, that are performed by an SVE unpredicate load instruction, all of the following are true:
  - For any 64-byte window aligned to 64 bytes containing at least 1 byte that is explicitly accessed by the instruction, any byte in the window can be accessed by the instruction.
  - All bytes accessed by the instruction will be in a 64-byte window aligned to 64 bytes containing at least 1 byte that is explicitly accessed by the instruction.
- For reads, including hardware speculation, that are performed by an SVE predicated load instruction that is not a non-temporal load, all of the following are true:
  - For any 64-byte window aligned to 64 bytes containing at least 1 byte that is explicitly accessed by an Active element of the instruction, any byte in the window can be accessed by the instruction.
  - All bytes accessed by the instruction will be in a 64-byte window aligned to 64 bytes that contains at least 1 byte that is explicitly accessed by an Active element of the instruction.
- For reads, including hardware speculation, that are performed by an SVE predicated load instruction from memory locations with the Gathering attributes, all of the following are true:
  - For any 128-byte window aligned to 128 bytes containing at least 1 byte that is explicitly accessed by an Active element of the instruction, any byte in the window can be accessed by the instruction.
  - All bytes accessed by the instruction are in a 128-byte window aligned to 128 bytes that contains at least 1 byte that is explicitly accessed by an Active element of the instruction.
- The architecture permits a Memory Copy and Memory Set CPY* instruction to perform speculative reads of any memory location, even those marked as Device, within a 64-byte quantity, aligned to 64 bytes, of a location that is within the range [Xs] to [Xs+Xn-1].
- For Device memory with the Gathering attribute, reads generated by the LDNP instructions are permitted to access bytes that are not explicitly accessed by the instruction, provided that the bytes accessed are in a 128-byte window, aligned to 128-bytes, that contains at least one byte that is explicitly accessed by the instruction.
- Where a load or store instruction performs a sequence of memory accesses, as opposed to one single-copy atomic access as defined in the rules for single-copy atomicity, these accesses might occur multiple times as a result of executing the load or store instruction. See Properties of single-copy atomic accesses on page B2-156.
Note — An instruction that generates a sequence of accesses as described in Atomicity in the Arm architecture on page B2-154 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception, the instruction is restarted, and therefore, one or more of the memory locations might be accessed multiple times. This can result in repeated accesses to a location where the program defines only a single access. For this reason, Arm strongly recommends that no accesses to Device memory are performed from a single instruction that spans the boundary of a translation granule or which in some other way could lead to some of the accesses being aborted.

— Write speculation that is visible to other observers is prohibited for all memory types.

- A write to a memory location with any Device memory type completes in finite time.
- If a value that would be returned from a read of a memory location with the Device memory type changes without an explicit memory write effect by an observer, this change must also be globally observed for all observers in the system in finite time. Such a change might occur in a peripheral location that holds status information.
- Data accesses to memory locations are coherent for all observers in the system, and correspondingly are treated as being Outer Shareable.
- A memory location with any Device memory attribute cannot be allocated into a cache.
- Writes to a memory location with any Device memory attribute must reach the endpoint for that address in the memory system in finite time. Two writes of Device memory type to the same location might be merged before they reach the endpoint, unless both writes have the non-Gathering attribute or there is an ordered-before relationship between the two writes.
- For accesses to any Device memory type, a DMB instruction introduces a Barrier-ordered-before relation on all accesses to a single peripheral or block of memory that is of implementation defined size. For more information, see Ordering relations on page B2-163.
- If a memory location is not capable of supporting unaligned memory accesses, then an unaligned access to that memory location generates an Alignment fault at the first stage of translation that defined the location as being Device.
- If a memory location is capable of supporting unaligned memory accesses, and such a memory location is marked as Device, then it is IMPLEMENTATION DEFINED whether an unaligned access to that memory location generates an Alignment fault at the first stage of translation that defined the location as being Device.
- Hardware does not prevent speculative instruction fetches from a memory location with any of the Device memory attributes unless the memory location is also marked as execute-never for all Exception levels.

Note — This means that to prevent speculative instruction fetches from memory locations with Device memory attributes, any location that is assigned any Device memory type must also be marked as execute-never for all Exception levels. Failure to mark a memory location with any Device memory attribute as execute-never for all Exception levels is a programming error.

Note — In the EL1&0 translation regime in systems where HCR_EL2.TGE==1 and HCR_EL2.DC==0, any Alignment fault that results from the fact that all locations are treated as Device is a fault at the first stage of translation. This causes ESR_EL2.ISS[24] to be 0.

See also Memory access restrictions on page B2-207.

The memory types for translation table walks cannot be defined as any Device memory type within the TCR_ELx. For the EL1&0 translation regime, the memory accesses made during a stage 1 translation table walk are subject to a stage 2 translation, and as a result of this second stage of translation, the accesses from the first stage translation
table walk might be made to memory locations with any Device memory type. These accesses might be made speculatively. When the value of the HCR_EL2.PTW bit is 1, a stage 2 Permission fault is generated if a first stage translation table walk is made to any Device memory type.

--- Note ---
In general, making a translation table walk to any Device memory type is the result of a programming error.

For an instruction fetch from a memory location with the Device attribute that is not marked as execute-never for the current Exception level, an implementation can either:

- Treat the instruction fetch as if it were to a memory location with the Normal Non-cacheable attribute.
- Take a Permission fault.

Gathering

In the Device memory attribute:

G Indicates that the location has the Gathering attribute.

nG Indicates that the location does not have the Gathering attribute, meaning it is non-Gathering.

The Gathering attribute determines whether it is permissible for either:

- Multiple memory accesses of the same type, read or write, to the same memory location to be merged into a single transaction.
- Multiple memory accesses of the same type, read or write, to different memory locations to be merged into a single memory transaction on an interconnect.

--- Note ---
This also applies to writebacks from the cache, whether caused by a Natural eviction or as a result of a cache maintenance instruction.

For memory types with the Gathering attribute, either of these behaviors is permitted, provided that the ordering and coherency rules of the memory location are followed.

For memory types with the non-Gathering attribute, neither of these behaviors is permitted. As a result:

- The number of memory accesses that are made corresponds to the number that would be generated by a simple sequential execution of the program.
- All accesses occur at their single-copy atomic sizes, except that there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic sizes accessed by multi-register load/store instructions that generate more than one single-copy atomic access. See Multi-register loads and stores that access Device memory on page B2-206.

Gathering between memory accesses separated by a memory barrier that affects those memory accesses is not permitted.

Gathering between two memory accesses generated by a Load-Acquire/Store-Release is not permitted.

Gathering between memory accesses generated by Memory Copy and Memory Set instructions is always permitted.

A read from a memory location with the non-Gathering attribute cannot come from a cache or a buffer, but must come from the endpoint for that address in the memory system. Typically this is a peripheral or physical memory.

--- Note ---
- A read from a memory location with the Gathering attribute can come from intermediate buffering of a previous write, provided that:
  - The accesses are not separated by a DMB or DSB barrier that affects both of the accesses.
  - The accesses are not separated by other ordering constructions that require that the accesses are in order. Such a construction might be a combination of Load-Acquire and Store-Release.
— The accesses are not generated by a Store-Release instruction.

• The Arm architecture defines only programmer visible behavior. Therefore, gathering can be performed if a programmer cannot tell whether gathering has occurred.

An implementation is permitted to perform an access with the Gathering attribute in a manner consistent with the requirements specified by the non-Gathering attribute.

An implementation is not permitted to perform an access with the non-Gathering attribute in a manner consistent with the relaxations allowed by the Gathering attribute.

Reordering

In the Device memory attribute:

R          Indicates that the location has the Reordering attribute. Accesses to the location can be reordered within the same rules that apply to accesses to Normal Non-cacheable memory. All memory types with the Reordering attribute have the same ordering rules as accesses to Normal Non-cacheable memory, see Ordering relations on page B2-163.

nR         Indicates that the location does not have the Reordering attribute, meaning it is non-Reordering.

Note

Some interconnect fabrics, such as PCIe, perform very limited reordering, which is not important for the software usage. It is outside the scope of the Arm architecture to prohibit the use of a non-Reordering memory type with these interconnects.

For all memory types with the non-Reordering attribute, the order of memory accesses arriving at a single peripheral of IMPLEMENTATION DEFINED size, as defined by the peripheral, must be the same order that occurs in a simple sequential execution of the program. That is, the accesses appear in program order. This ordering applies to all accesses using any of the memory types with the non-Reordering attribute. As a result, if there is a mixture of Device-nGnRE and Device-nGnRnE accesses to the same peripheral, these occur in program order. If the memory accesses are not to a peripheral, then this attribute imposes no restrictions.

Note

• The IMPLEMENTATION DEFINED size of the single peripheral is the same as applies for the ordering guarantee provided by the DMB instruction.

• The Arm architecture defines only programmer visible behavior. Therefore, reordering can be performed if a programmer cannot tell whether reordering has occurred.

• The non-Reordering property is only required by the architecture to apply the order of arrival of accesses to a single memory-mapped peripheral of an IMPLEMENTATION DEFINED size, and is not required to have an impact on the order of observation of memory accesses to SDRAM. For this reason, there is no effect of the non-Reordering attribute on the ordering relations between accesses to different locations described in Ordering relations on page B2-163 as part of the formal definition of the memory model.

• If the same memory location is mapped with different aliases, and different attribute values, these are a type of mismatched attribute. The different attributes could be:
  — A different Reordering attribute value.
  — A different Device memory attribute value.
  — When FEAT_XS is implemented, a different XS attribute value.

For information about the effects of accessing memory with mismatched attributes, see Mismatched memory attributes on page B2-208.
An implementation:

- Is permitted to perform an access with the Reordering attribute in a manner consistent with the requirements specified by the non-Reordering attribute.
- Is not permitted to perform an access with the non-Reordering attribute in a manner consistent with the relaxations allowed by the Reordering attribute.

The non-Reordering attribute does not require any additional ordering, other than that which applies to Normal memory, between:

- Accesses to one physical address with the non-Reordering attribute and accesses to a different physical address with the Reordering attribute.
- Access to one physical address with the non-Reordering attribute and access to a different physical address to Normal memory.
- Accesses with the non-Reordering attribute and accesses to different peripherals of IMPLEMENTATION DEFINED size.

The non-Reordering attribute has no effect on the ordering of cache maintenance instructions, even if the memory location specified in the instruction has the non-Reordering attribute.

The non-Reordering attribute has no effect on the memory accesses caused by Memory Copy and Memory Set instructions.

**Early Write Acknowledgement**

In the Device memory attribute:

| E  | Indicates that the location has the Early Write Acknowledgement attribute. |
| nE | Indicates that the location has the No Early Write Acknowledgement attribute. |

If the No Early Write Acknowledgement attribute is assigned for a Device memory location:

- For memory system endpoints where the system architecture in which the PE is operating requires that acknowledgement of a write comes from the endpoint, it is guaranteed that:
  - Only the endpoint of the write access returns a write acknowledgement of the access.
  - No earlier point in the memory system returns a write acknowledgement.

- For memory system endpoints where the system architecture in which the PE is operating does not require that acknowledgement of a write comes from the endpoint, the acknowledgement of a write is not required to come from the endpoint.

**Note**

A write with the No Early Write Acknowledgement attribute assigned for a Device memory location is not expected to generate an abort in any situation where the equivalent write to the same location without the No Early Write Acknowledgement attribute assigned does not generate an abort.

This means that a DSB barrier instruction, executed by the PE that performed the write to the No Early Write Acknowledgement Location, completes only after the write has reached its endpoint in the memory system if that is required by the system architecture.

Peripherals are an example of system endpoints that require that the acknowledgement of a write comes from the endpoint.

**Note**

- The Early Write Acknowledgement attribute only affects where the endpoint acknowledgement is returned from, and does not affect the ordering of arrival at the endpoint between accesses, which is determined by either the Device Reordering attribute, or the use of barriers to create order.
• The areas of the physical memory map for which write acknowledgement from the endpoint is required is outside the scope of the Arm Architecture definition and must be defined as part of the system architecture in which the PE is operating. In particular, regions of memory handled as PCIe configuration writes are expected to support write acknowledgement from the endpoint.

• Arm recognizes that not all areas of a physical memory map will be capable of supporting write acknowledgement from the endpoint. In particular, Arm expects that regions of memory handled as posted writes under PCIe will not support write acknowledgement from the endpoint.

• For maximum software compatibility, Arm strongly recommends that all peripherals for which standard software drivers expect that the use of a DSB instruction will determine that a write has reached its endpoint are placed in areas of the physical memory map that support write acknowledgement from the endpoint.

---

**Multi-register loads and stores that access Device memory**

For all instructions that load or store more than one general-purpose register and generate more than one single-copy atomic access for that load or store, there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic sizes accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register, the order in which the registers are accessed is not defined by the architecture. This applies even to accesses to any type of Device memory.

For all instructions that load or store one or more SIMD&FP or SVE registers, and generate more than one single-copy atomic access for that load or store, there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic sizes accessed by these load or store instructions, even for access to any type of Device memory.

The architecture permits that the non-speculative execution of an instruction that loads or stores more than one general-purpose or SIMD and floating-point register might result in repeated accesses to the same address.

**SVE loads and stores that access Device memory**

All rules applying to Device memory accesses by Advanced SIMD and floating-point load and store instructions apply to Device memory access by SVE load and store instructions.

Additional rules apply to Device memory access by SVE load and store instructions.

When an SVE vector prefetch instruction is executed, any resulting memory read is guaranteed not to access Device memory.

When an SVE Non-fault vector load is executed, or when any element from a First-fault load except the **First active element** attempts to access Device memory, the resulting memory reads will not access Device memory.

When an SVE Non-fault vector load instruction is executed, an attempt by any **Active element** to access Device memory is suppressed and reported in the FFR.

When an SVE First-fault vector load instruction is executed, any memory read performed for the **First active element** can access Device memory.

When an SVE First-fault vector load instruction is executed, an attempt by any **Active element** other than the **First active element** to access Device memory is suppressed and is reported in the FFR.

Any access to Device memory performed by an SVE load or store instruction is relaxed such that it might behave as if:

• The Gathering attribute is set, regardless of the configured value of the nG attribute.
• The Reordering attribute is set, regardless of the configured value of the nR attribute.
• The Early Acknowledgement attribute is set, regardless of the configured value of the nE attribute.

Whether or not attributes are classified as mismatched is determined strictly by the memory attributes derived from the page-table entry.
B2.7.3 Memory access restrictions

The following restrictions apply to memory accesses:

- For two explicit memory read effects to any two adjacent bytes in memory, \( p \) and \( p+1 \), generated by the same instruction, and for two explicit memory write effects to any two adjacent bytes in memory, \( p \) and \( p+1 \), that are generated by the same instruction:
  - The bytes \( p \) and \( p+1 \) must have the same memory type and Shareability attributes, otherwise the results are CONSTRAINED UNPREDICTABLE. For example, an LD1, ST1, or an unaligned load or store that spans the boundary between Normal memory and Device memory is CONSTRAINED UNPREDICTABLE.
  - Except for possible differences in the cache allocation hints, Arm deprecates having different cacheability attributes for bytes \( p \) and \( p+1 \).

For the permitted CONSTRAINED UNPREDICTABLE behavior, see Crossing a page boundary with different memory types or Shareability attributes on page K1-11589.

- If the accesses of an instruction that causes multiple accesses to any type of Device memory cross an address boundary that corresponds to the smallest implemented translation granule, then behavior is CONSTRAINED UNPREDICTABLE, and Crossing a peripheral boundary with a Device access on page K1-11590 describes the permitted behaviors. For this reason, it is important that an access to a volatile memory device is not made using a single instruction that crosses an address boundary of the size of the smallest implemented translation granule.

Note

- The boundary referred to is between two Device memory regions that are both of the size of the smallest implemented translation granule and aligned to the size of the smallest implemented translation granule.
- This restriction means it is important that an access to a volatile memory device is not made using a single instruction that crosses an address boundary of the size of the smallest implemented translation granule.
- Arm expects this restriction to constrain the placing of volatile memory devices in the system memory map, rather than expecting a compiler to be aware of the alignment of memory accesses.
B2.8 Mismatched memory attributes

Memory attributes are controlled by privileged software. For more information, see Chapter D8 The AArch64 Virtual Memory System Architecture.

Physical memory locations are accessed with mismatched attributes if all accesses to the location do not use a common definition of all of the following attributes of that location:

- Memory type: Device-nGnRnE, Device-nGnRE, Device-nGRE, Device-GRE or Normal.
- Shareability.
- Cacheability, for the same level of the inner or outer cache, but excluding any cache allocation hints.
- When FEAT_XS is implemented, XS attribute.

Collectively these are referred to as memory attributes.

If FEAT_MTE2 is implemented, accesses to a location which use a common definition of the memory attributes but the Tagged attribute of that location differs do not cause a mismatched access to occur.

--- Note ---

In this document, the terms location and memory location refer to any byte within the current coherency granule and are used interchangeably.

---

When a memory Location is accessed with mismatched attributes, the only software visible effects are one or more of the following:

- Uniprocessor semantics for reads and writes to that memory Location might be lost. This means:
  - A read of the memory Location by one agent might not return the value most recently written to that memory Location by the same agent.
  - Multiple writes to the memory Location by one agent with different memory attributes might not be ordered in program order.
- There might be a loss of coherency when multiple agents attempt to access a memory Location.
- There might be a loss of properties derived from the memory type, as described in later bullets in this section.
- If all Load-Exclusive/Store-Exclusive instructions executed across all threads to access a given memory Location do not use consistent memory attributes, the Exclusives monitor state becomes UNKNOWN.
- Bytes written without the Write-Back cacheable attribute within the same Write-Back granule as bytes written with the Write-Back cacheable attribute might have their values reverted to the old values as a result of cache Write-Back.

The loss of properties associated with mismatched memory type attributes refers only to the following properties of Device memory that are additional to the properties of Normal memory:

- Prohibition of Speculative read accesses.
- Prohibition on Gathering.
- Prohibition on reordering.

For the following situations, when a physical memory Location is accessed with mismatched attributes, a more restrictive set of behaviors applies. The description of each situation also describes the behaviors that apply:

1. Any agent that reads that memory Location using the same common definition of the Memory type, Shareability and Cacheability attributes is guaranteed to access it coherently, to the extent required by that common definition of the memory attributes, only if all the following conditions are met:

   - All writes are performed to an alias of the memory Location that uses the same definition of the Memory type, Shareability and Cacheability attributes.
   - Either:
     - In the EL1&0 translation regime, HCR_EL2.MIOCNCE has a value of 0.
     - All aliases with write permission have the Inner Cacheability attribute the same as the Outer Cacheability attribute.
• Either:
  — All writes are performed to an alias of the memory Location that has Inner Cacheability and Outer Cacheability attributes both as Non-cacheable.
  — All aliases to a memory Location use a definition of the Shareability attributes that encompasses all the agents with permission to access the Location.

2. The possible software-visible effects caused by mismatched attributes for a memory Location are defined more precisely if all of the mismatched attributes define the memory Location as one of:
• Any Device memory type.
• Inner Non-cacheable, Outer Non-cacheable Normal memory.

In these cases, the only permitted software-visible effects of the mismatched attributes are one or more of the following:
• Possible loss of properties derived from the memory type when multiple agents attempt to access the memory Location.
• Possible reordering of memory transactions to the same memory Location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Any possible loss of coherency or uniprocessor semantics can be avoided by inserting DMB barrier instructions between accesses to the same memory Location that might use different attributes.

Where there is a loss of the uniprocessor semantics, ordering, or coherency, the following approaches can be used:

1. If the mismatched attributes for a memory location all assign the same shareability attribute to a Location that has a cacheable attribute, any loss of uniprocessor semantics, ordering, or coherency within a shareability domain can be avoided by use of software cache management. To do so, software must use the techniques that are required for the software management of the ordering or coherency of cacheable Locations between agents in different shareability domains. This means:
• Before writing to a cacheable Location not using the Write-Back attribute, software must invalidate, or clean, a Location from the caches if any agent might have written to the Location with the Write-Back attribute. This avoids the possibility of overwriting the Location with stale data.
• After writing to a cacheable Location with the Write-Back attribute, software must clean the Location from the caches, to make the write visible to external memory.
• Before reading the Location with a cacheable attribute, software must invalidate, or clean and invalidate, the Location from the caches, to ensure that any value held in the caches reflects the last value made visible in external memory.
• Executing a DMB barrier instruction, with scope that applies to the common shareability of the accesses, between any accesses to the same cacheable Location that use different attributes.

In all cases:
• Location refers to any byte within the current coherency granule.
• A clean and invalidate instruction can be used instead of a clean instruction, or instead of an invalidate instruction.
• In the sequences outlined in this section, all cache maintenance instructions and memory transactions must be completed, or ordered by the use of barrier operations, if they are not naturally ordered by the use of a common address, see Ordering and completion of data and instruction cache instructions on page D7-5063.

--- Note ---

With software management of coherency, race conditions can cause loss of data. A race condition occurs when different agents write simultaneously to bytes that are in the same Location, and the invalidate, write, clean sequence of one agent overlaps with the equivalent sequence of another agent. A race condition also occurs if the first operation of either sequence is a clean, rather than an invalidate.
2. If the mismatched attributes for a Location mean that multiple cacheable accesses to the Location might be made with different shareability attributes, then uniprocessor semantics, ordering, and coherency are guaranteed only if:

- Software running on a PE cleans and invalidates a Location from cache before and after each read or write to that Location by that PE.
- A DMB barrier with scope that covers the full shareability of the accesses is placed between any accesses to the same memory Location that use different attributes.

--- Note ---

The Note in rule 1 of this list, about possible race conditions, also applies to this rule.

In addition, if multiple agents attempt to use Load-Exclusive or Store-Exclusive instructions to access a Location, and the accesses from the different agents have different memory attributes associated with the Location, the Exclusives monitor state becomes UNKNOWN.

Arm strongly recommends that software does not use mismatched attributes for aliases of the same Location. An implementation might not optimize the performance of a system that uses mismatched aliases.

--- Note ---

As described in Non-cacheable accesses and instruction caches on page D7-5049, a non-cacheable access is permitted to be cached in an instruction cache, despite the fact that a non-cacheable access is not permitted to be cached in a unified cache. Despite this, when cacheable and non-cacheable aliases exist for memory which is executable, these must be treated as mismatched aliases to avoid coherency issues from the data or unified caches that might hold entries that will be brought into the instruction caches.
B2.9  Synchronization and semaphores

Armv8 provides non-blocking synchronization of shared memory, using synchronization primitives. The information in this section about memory accesses by synchronization primitives applies to accesses to both Normal memory and to any type of Device memory.

Note

Use of the Armv8 synchronization primitives scales for multiprocessing system designs.

Table B2-3 on page B2-211 shows the synchronization primitives and the associated CLREX instruction.

<table>
<thead>
<tr>
<th>Transaction size</th>
<th>Additional semantics</th>
<th>Load-Exclusive&lt;sup&gt;a&lt;/sup&gt;</th>
<th>Store-Exclusive&lt;sup&gt;a&lt;/sup&gt;</th>
<th>Other&lt;sup&gt;a&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>-</td>
<td>LDXRB</td>
<td>STXRB</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Load-Acquire/Store-Release</td>
<td>LDAXR</td>
<td>STLXRB</td>
<td>-</td>
</tr>
<tr>
<td>Halfword</td>
<td>-</td>
<td>LDXH</td>
<td>STXH</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Load-Acquire/Store-Release</td>
<td>LDAXR</td>
<td>STLXH</td>
<td>-</td>
</tr>
<tr>
<td>Register&lt;sup&gt;b&lt;/sup&gt;</td>
<td>-</td>
<td>LDXR</td>
<td>STXR</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Load-Acquire/Store-Release</td>
<td>LDAXR</td>
<td>STLXR</td>
<td>-</td>
</tr>
<tr>
<td>Pair&lt;sup&gt;b&lt;/sup&gt;</td>
<td>-</td>
<td>LDXP</td>
<td>STXP</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Load-Acquire/Store-Release</td>
<td>LDAXP</td>
<td>STLXP</td>
<td>-</td>
</tr>
<tr>
<td>None</td>
<td>Clear-Exclusive</td>
<td>-</td>
<td>-</td>
<td>CLREX</td>
</tr>
</tbody>
</table>

<sup>a</sup> Instruction in the A64 instruction set.
<sup>b</sup> A register instruction operates on a doubleword if accessing an X register, or on a word if accessing a W register. A pair instruction operates on two doublewords if accessing X registers, or on two words if accessing W registers.

Except for the row showing the CLREX instruction, the two instructions in a single row are a Load-Exclusive/Store-Exclusive instruction pair. The model for the use of a Load-Exclusive/Store-Exclusive instruction pair accessing a non-aborting memory address \( x \) is:

- The Load-Exclusive instruction reads a value from memory address \( x \).
- The corresponding Store-Exclusive instruction succeeds in writing back to memory address \( x \) only if no other observer, process, or thread has performed a more recent store to address \( x \). The Store-Exclusive instruction returns a status bit that indicates whether the memory write succeeded.

A Load-Exclusive instruction marks a small block of memory for exclusive access. The size of the marked block is IMPLEMENTATION DEFINED, see Marking and the size of the marked memory block on page B2-217. A Store-Exclusive instruction to any address in the marked block clears the marking.

Note

In this section, the term PE includes any observer that can generate a Load-Exclusive or a Store-Exclusive instruction.

The following sections give more information:

- Exclusive access instructions and Non-shareable memory locations on page B2-212.
- Exclusive access instructions and Shareable memory locations on page B2-213.
- Marking and the size of the marked memory block on page B2-217.
- Context switch support on page B2-217.
Use of WFE and SEV instructions by spin-locks on page B2-221.

B2.9 Synchronization and semaphores

B2.9.1 Exclusive access instructions and Non-shareable memory locations

For memory locations for which the shareability attribute is Non-shareable, the exclusive access instructions rely on a local Exclusives monitor, or local monitor, that marks any address from which the PE executes a Load-Exclusive instruction. Any non-aborted attempt by the same PE to use a Store-Exclusive instruction to modify any address is guaranteed to clear the marking.

A Load-Exclusive instruction performs a load from memory, and:
- The executing PE marks the physical memory address for exclusive access.
- The local monitor of the executing PE transitions to the Exclusive Access state.

A Store-Exclusive instruction performs a conditional store to memory that depends on the state of the local monitor:

If the local monitor is in the Exclusive Access state

- If the address of the Store-Exclusive instruction is the same as the address that has been marked in the monitor by an earlier Load-Exclusive instruction, then the store occurs. Otherwise, it is IMPLEMENTATION DEFINED whether the store occurs.
- A status value is returned to a register:
  - If the store took place, the status value is 0.
  - Otherwise, the status value is 1.
- The local monitor of the executing PE transitions to the Open Access state.

When an Exclusives monitor is in the Exclusive Access state, the monitor is set.

If the local monitor is in the Open Access state

- No store takes place.
- A status value of 1 is returned to a register.
- The local monitor remains in the Open Access state.

When an Exclusives monitor is in the Open Access state, the monitor is clear.

The Store-Exclusive instruction defines the register to which the status value is returned.

When a PE writes using any instruction other than a Store-Exclusive instruction:
- If the write is to a PA that is not marked as Exclusive Access by its local monitor and that local monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.
- If the write is to a PA that is marked as Exclusive Access by its local monitor, it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

It is IMPLEMENTATION DEFINED whether a store to a marked PA causes a mark in the local monitor to be cleared if that store is by an observer other than the one that caused the PA to be marked.

Figure B2-4 on page B2-213 shows the state machine for the local monitor and the effect of each of the operations shown in the figure.
The AArch64 Application Level Memory Model
B2.9 Synchronization and semaphores

Figure B2-4  Local monitor state machine diagram

For more information about marking, see Marking and the size of the marked memory block on page B2-217.

Note

For the local monitor state machine, as shown in Figure B2-4:

• The IMPLEMENTATION DEFINED options for the local monitor are consistent with the local monitor being constructed so that it does not hold any PA, but instead treats any access as matching the address of the previous Load-Exclusive instruction.

• A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive instructions from other PEs.

• The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the local monitor.

• It is IMPLEMENTATION DEFINED whether the transition from Exclusive Access to Open Access state occurs when the Store or StoreExc1 is from another observer.

Changes to the local monitor state resulting from speculative execution

The architecture permits a local monitor to transition to the Open Access state as a result of speculation, or from some other cause. This is in addition to the transitions to Open Access state caused by the architectural execution of an operation shown in Figure B2-4.

An implementation must ensure that:

• The local monitor cannot be seen to transition to the Exclusive Access state except as a result of the architectural execution of one of the operations shown in Figure B2-4.

• Any transition of the local monitor to the Open Access state not caused by the architectural execution of an operation shown in Figure B2-4 must not indefinitely delay forward progress of execution.

B2.9.2  Exclusive access instructions and Shareable memory locations

In the context of this section, a shareable memory location is a memory location that has, or is treated as if it has, a Shareability attribute of Inner Shareable or Outer Shareable.
For shareable memory locations, exclusive access instructions rely on:
• A local monitor for each PE in the system, which marks any address from which the PE executes a Load-Exclusive. The local monitor operates as described in Exclusive access instructions and Non-shareable memory locations on page B2-212, except that for shareable memory any Store-Exclusive is then subject to checking by the global monitor if it is described in that section as doing at least one of the following:
  — Updating memory.
  — Returning a status value of 0.

The local monitor can ignore accesses from other PEs in the system.
• A global monitor that marks a PA as exclusive access for a particular PE. This marking is used later to determine whether a Store-Exclusive to that address that has not been failed by the local monitor can occur. Any successful write to the marked block by any other observer in the shareability domain of the memory location is guaranteed to clear the marking. For each PE in the system, the global monitor:
  — Can hold at least one marked block.
  — Maintains a state machine for each marked block it can hold.

For each PE, the architecture only requires global monitor support for a single marked address. Any situation that might benefit from the use of multiple marked addresses on a single PE is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE, see Load-Exclusive and Store-Exclusive instruction usage restrictions on page B2-218.

The global monitor can either reside within the PE, or exist as a secondary monitor at the memory interfaces. The IMPLEMENTATION DEFINED aspects of the monitors mean that the global monitor and local monitor can be combined into a single unit, provided that the unit performs the global monitor and local monitor functions defined in this manual.

For shareable memory locations, in some implementations and for some memory types, the properties of the global monitor require functionality outside the PE. Some system implementations might not implement this functionality for all locations of memory. In particular, this can apply to:
• Any type of memory in the system implementation that does not support hardware cache coherency.
• Non-cacheable memory, or memory treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such a system, it is defined by the system:
• Whether the global monitor is implemented.
• If the global monitor is implemented, which address ranges or memory types it monitors.

If FEAT_MTE2 is implemented, it is IMPLEMENTATION DEFINED whether a global monitor monitors access to the Tag PA space. For more information, see Chapter D9 The Memory Tagging Extension.

To support the use of the Load-Exclusive/Store-Exclusive mechanism when address translation is disabled, a system might define at least one location of memory, of at least the size of the translation granule, in the system memory map to support the global monitor for all Arm PEs within a common Inner Shareable domain. However, this is not an architectural requirement. Therefore, architecturally-compliant software that requires mutual exclusion must not rely on using the Load-Exclusive/Store-Exclusive mechanism, and must instead use a software algorithm such as Lamport’s Bakery algorithm to achieve mutual exclusion.

Because implementations can choose which memory types are treated as Non-cacheable, the only memory types for which it is architecturally guaranteed that a global Exclusives monitor is implemented are:
• Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.
• Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.

The architecture only requires that Conventional memory mapped in this way supports this functionality.

If the global monitor is not implemented for an address range or memory type, then performing a Load-Exclusive or a Store-Exclusive instruction to such a location has one or more of the following effects:

• The instruction generates an External abort.
• The instruction generates an IMPLEMENTATION DEFINED MMU fault. This is reported using the Data Abort Fault status code of ESR_ELx.DFSC = 110101.

If the IMPLEMENTATION DEFINED MMU fault is generated for the EL1&0 translation regime then:

— If the fault is generated because of the memory type defined in the first stage of translation, or if the second stage of translation is disabled, then this is a first stage fault and the exception is taken to EL1.
— Otherwise, the fault is a second stage fault and the exception is taken to EL2.

The priority of this fault is IMPLEMENTATION DEFINED.
• The instruction is treated as a NOP.
• The Load-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
• The Store-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN. In this case, if the store exclusive instruction is a store exclusive pair of 64-bit quantities, then the two quantities being stored might not be stored atomically.
• The value held in the result register of the Store-Exclusive instruction becomes UNKNOWN.

In addition, for write transactions generated by non-PE observers that do not implement exclusive accesses or other atomic access mechanisms, the effect that writes have on the global and local monitors used by Arm PEs is IMPLEMENTATION DEFINED. The writes might not clear the global monitors of other PEs for:

• Some address ranges.
• Some memory types.

**Operation of the global Exclusives monitor**

A Load-Exclusive instruction from shareable memory performs a load from memory, and causes the PA of the access to be marked as exclusive access for the requesting PE. This access can also cause the exclusive access mark to be removed from any other PA that has been marked by the requesting PE.

---

**Note**

The global monitor supports only a single outstanding exclusive access to shareable memory per PE.

---

A Load-Exclusive instruction by one PE has no effect on the global monitor state for any other PE.

A Store-Exclusive instruction performs a conditional store to memory:

• The store is guaranteed to succeed only if the PA accessed is marked as exclusive access for the requesting PE and both the local monitor and the global monitor state machines for the requesting PE are in the Exclusive Access state. In this case:
  — A status value of 0 is returned to a register to acknowledge the successful store.
  — The final state of the global monitor state machine for the requesting PE is IMPLEMENTATION DEFINED.
  — If the address accessed is marked for exclusive access in the global monitor state machine for any other PE, then that state machine transitions to Open Access state.

• If no address is marked as exclusive access for the requesting PE, the store does not succeed:
  — A status value of 1 is returned to a register to indicate that the store failed.
  — The global monitor is not affected and remains in Open Access state for the requesting PE.
• If a different PA is marked as exclusive access for the requesting PE, it is IMPLEMENTATION DEFINED whether the store succeeds or not:
  — If the store succeeds a status value of 0 is returned to a register, otherwise a value of 1 is returned.
  — If the global monitor state machine for the PE was in the Exclusive Access state before the Store-Exclusive instruction it is IMPLEMENTATION DEFINED whether that state machine transitions to the Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

In a shared memory system, the global monitor implements a separate state machine for each PE in the system. The state machine for accesses to shareable memory by PE(n) can respond to all the shareable memory accesses visible to it. This means that it responds to:
• Accesses generated by PE(n).
• Accesses generated by the other observers in the shareability domain of the memory location. These accesses are identified as (n).

In a shared memory system, the global monitor implements a separate state machine for each observer that can generate a Load-Exclusive or a Store-Exclusive instruction in the system.

A global monitor:
• In the Exclusive Access state is set.
• In the Open Access state is clear.

Clear global monitor event

Whenever the global monitor state for a PE changes from Exclusive access to Open access, an event is generated and held in the Event register for that PE. This register is used by the Wait for Event mechanism, see Mechanisms for entering a low-power state on page D1-4676.

Figure B2-5 shows the state machine for PE(n) in a global monitor.

Figure B2-5 Global monitor state machine diagram for PE(n) in a multiprocessor system

For more information about marking, see Marking and the size of the marked memory block on page B2-217.
Note

For the global monitor state machine, as shown in Figure B2-5 on page B2-216:

- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the global monitor.

- Whether a Store-Exclusive instruction successfully updates memory or not depends on whether the address accessed matches the marked shareable memory address for the PE issuing the Store-Exclusive instruction, and whether the local and global monitors are in the exclusive state. For this reason, Figure B2-5 on page B2-216 shows only how the operations by (!n) cause state transitions of the state machine for PE(n).

- A Load-Exclusive instruction can only update the marked shareable memory address for the PE issuing the Load-Exclusive instruction.

- When the global monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether a CLREX instruction causes the global monitor to transition from Exclusive Access to Open Access state.

- It is IMPLEMENTATION DEFINED:
  - Whether a modification to a Non-shareable memory location can cause a global monitor to transition from Exclusive Access to Open Access state.
  - Whether a Load-Exclusive instruction to a Non-shareable memory location can cause a global monitor to transition from Open Access to Exclusive Access state.

B2.9.3 Marking and the size of the marked memory block

When a Load-Exclusive instruction is executed, the resulting marked block ignores the least significant bits of the 64-bit memory address.

When a Load-Exclusive instruction is executed, a marked block of size $2^a$ bytes is created by ignoring the least significant bits of the memory address. A marked address is any address within this marked block. The size of the marked memory block is called the Exclusives reservation granule. The Exclusives reservation granule is IMPLEMENTATION DEFINED in the range 4-512 words.

Note

This definition means that the Exclusives reservation granule is:

- 4 words in an implementation where $a$ is 4.
- 512 words in an implementation where $a$ is 11.

For example, in an implementation where $a$ is 4, a successful LDXB of address 0x341B4 defines a marked block using bits[47:4] of the address. This means that the four words of memory from 0x341B0 to 0x341BF are marked for exclusive access.

In some implementations the CTR identifies the Exclusives reservation granule, see CTR_EL0. Otherwise, software must assume that the maximum Exclusives reservation granule, 512 words, is implemented.

B2.9.4 Context switch support

An exception return clears the local monitor. As a result, performing a CLREX instruction as part of a context switch is not required in most situations.

Note

Context switching is not an application level operation. However, this information is included here to complete the description of the exclusive operations.
B2.9.5 Load-Exclusive and Store-Exclusive instruction usage restrictions

The Load-Exclusive and Store-Exclusive instructions are intended to work together as a pair, for example a LDXP/STXP pair or a LDXR/STXR pair. To support different implementations of these functions, software must follow the notes and restrictions given here.

The following notes describe the use of a LoadExcl/StoreExcl instruction pair, to indicate the use of any of the Load-Exclusive/Store-Exclusive instruction pairs shown in Table B2-3 on page B2-211. In this context, a LoadExcl/StoreExcl pair comprises two instructions in the same thread of execution:

- The exclusives support a single outstanding exclusive access for each PE thread that is executed. The architecture makes use of this by not requiring an address or size check as part of the IsExclusiveLocal() function. If the target VA of a StoreExcl is different from the VA of the preceding LoadExcl instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The StoreExcl either passes or fails, the status value returned by the StoreExcl is UNKNOWN, and the states of the local and global monitors for that PE are UNKNOWN.

Note

This means the StoreExcl might pass for some instances of a LoadExcl/StoreExcl pair with mismatched addresses, and fail for other instances of a LoadExcl/StoreExcl pair with mismatched addresses.

- The data at the address accessed by the LoadExcl, and at the address accessed by the StoreExcl, is UNKNOWN.

This means software can rely on a LoadExcl/StoreExcl pair to eventually succeed only if the LoadExcl and the StoreExcl are executed with the same VA.

- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the transaction size of a StoreExcl instruction is the same as the transaction size of the preceding LoadExcl instruction executed in that thread. If the transaction size of a StoreExcl instruction is different from the preceding LoadExcl instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The StoreExcl either passes or fails, and the status value returned by the StoreExcl is UNKNOWN.

Note

This means the StoreExcl might pass for some instances of a LoadExcl/StoreExcl pair with mismatched transaction sizes, and fail for other instances of a LoadExcl/StoreExcl pair with mismatched transaction sizes.

- The block of data of the size of the larger of the transaction sizes used by the LoadExcl/StoreExcl pair at the address accessed by the LoadExcl/StoreExcl pair, is UNKNOWN.

This means software can rely on a LoadExcl/StoreExcl pair to eventually succeed only if they access the same number of registers. For more information, see CONSTRAINED UNPREDICTABLE behavior when Load-Exclusive/Store-Exclusive access a different number of registers on page B2-220.

- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the StoreExcl instruction accesses the same number of registers as the preceding LoadExcl instruction executed in that thread. If the StoreExcl instruction accesses a different number of registers than the preceding LoadExcl instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE. As a result, software can rely on an LoadExcl/StoreExcl pair to eventually succeed only if they access the same number of registers. For more information, see CONSTRAINED UNPREDICTABLE behavior when Load-Exclusive/Store-Exclusive access a different number of registers on page B2-220.

- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the Tag Checked property of a memory access due to a StoreExcl instruction is the same as the Tag Checked property of a memory access by the preceding LoadExcl instruction executed in that thread. If the Tag Checked property of memory accesses due to a LoadExcl/StoreExcl pair in the same thread of execution differ, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The StoreExcl either passes or fails, and the status value returned by the StoreExcl is UNKNOWN.

Note

This means the StoreExcl might pass for some instances of such a LoadExcl/StoreExcl pair, and fail for other instances of such a LoadExcl/StoreExcl pair.

- The data at the address accessed by the LoadExcl/StoreExcl pair is UNKNOWN.
This means software can rely on a LoadExcl/StoreExcl pair to eventually succeed only if the memory is accessed with the same Tag Checked property.

- **LoadExcl/StoreExcl** loops are guaranteed to make forward progress only if, for any LoadExcl/StoreExcl loop within a single thread of execution, the software meets all of the following conditions:
  1. Between the Load-Exclusive and the Store-Exclusive, there are no explicit memory effects, preloads, direct or indirect System register writes, address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, ISB barriers, or indirect branches.
  2. Between the Store-Exclusive returning a failing result and the retry of the corresponding Load-Exclusive:
     - There are no stores or PRFM instructions to any address within the Exclusives reservation granule accessed by the Store-Exclusive.
     - There are no loads or preloads to any address within the Exclusives reservation granule accessed by the Store-Exclusive that use a different VA alias to that address.
     - There are no direct or indirect System register writes, address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, or indirect branches.
     - All loads and stores are to a block of contiguous virtual memory of not more than 512 bytes in size.

The Exclusives monitor can be cleared at any time without an application-related cause, provided that such clearing is not systematically repeated so as to prevent the forward progress in finite time of at least one of the threads that is accessing the Exclusives monitor. However, it is permissible for the LoadExcl/StoreExcl loop not to make forward progress if a different thread is repeatedly doing any of the following in a tight loop:

- Performing stores to a PA covered by the Exclusives monitor.
- Prefetching with intent to write to a PA covered by the Exclusives monitor.
- Executing data cache clean, data cache invalidate, or data cache clean and invalidate instructions to a PA covered by the Exclusives monitor.
- Executing instruction cache invalidate all instructions.
- Executing instruction cache invalidate by VA instructions to a PA covered by the Exclusives monitor.
- Executing TLB maintenance to a PA covered by the Exclusives monitor.

- Implementations can benefit from keeping the **LoadExcl** and **StoreExcl** operations close together in a single thread of execution. This minimizes the likelihood of the Exclusives monitor state being cleared between the **LoadExcl** instruction and the **StoreExcl** instruction. Therefore, for best performance, Arm strongly recommends a limit of 128 bytes between **LoadExcl** and **StoreExcl** instructions in a single thread of execution.

- The architecture sets an upper limit of 2048 bytes on the Exclusives reservation granule that can be marked as exclusive. For performance reasons, Arm recommends that objects that are accessed by exclusive accesses are separated by the size of the Exclusives reservation granule. This is a performance guideline rather than a functional requirement.

- After taking a Data Abort exception, the state of the Exclusives monitors is UNKNOWN.

- For the memory location accessed by a **LoadExcl/StoreExcl** pair, if the memory attributes for a **StoreExcl** instruction are different from the memory attributes for the preceding **LoadExcl** instruction in the same thread of execution, behavior is **CONSTRAINED UNPREDICTABLE**. Where this occurs because the translation of the accessed address changes between the **LoadExcl** instruction and the **StoreExcl** instruction, the **CONSTRAINED UNPREDICTABLE** behavior is as follows:

  - The **StoreExcl** either passes or fails, and the status value returned by the **StoreExcl** is UNKNOWN.

  ____________

  **Note**

  This means the **StoreExcl** might pass for some instances of a **LoadExcl/StoreExcl** pair with changed memory attributes, and fail for other instances of a **LoadExcl/StoreExcl** pair with changed memory attributes.

  ____________

  - The data at the address accessed by the **StoreExcl** is UNKNOWN.

  ____________

  **Note**

  Another bullet point in this list covers the case where the memory attributes of a **LoadExcl/StoreExcl** pair differ as a result of using different VAs with different attributes that point to the same PA.
• The effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global
Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE, and the instruction
might clear the monitor, or it might leave it in the Exclusive Access state. For address-based maintenance
instructions, this also applies to the monitors of other PEs in the same shareability domain as the PE executing
the cache maintenance instruction, as determined by the shareability domain of the address being maintained.

—— Note ———
Arm strongly recommends that implementations ensure that the use of such maintenance instructions by a
PE in the Non-secure state cannot cause a denial of service on a PE in the Secure state.

—— Note ———
If the mapping of the VA to PA is changed between the LoadExcl instruction and the STREX instruction, and
the change is performed using a break-before-make sequence as described in Using break-before-make when
updating translation table entries on page D8-5198, if the StoreExcl is performed after another write to the
same PA as the StoreExcl, and that other write was performed after the old translation was properly
invalidated and that invalidation was properly synchronized, then the StoreExcl will not pass its monitor
check.

—— Note ———
— The TLB invalidation will clear either the local or global monitor.
— The PA will be checked between the LoadExcl and StoreExcl.

• The Exclusive Access state for an address accessed by a PE can be lost as a result of a PFRM PST* instruction
to the same PA executed by another PE. This means that a very high rate of repeated PFRM PST* accesses to a
memory location might impede the forward progress of another PE.
• If FEAT_MTE2 is implemented, and if a Tag Unchecked store exclusive instruction would not perform the
store and return a status value of 1, it is CONSTRAINED UNPREDICTABLE whether:
— The instruction is a Tag Checked access,
— The instruction is a Tag Unchecked access.
For more information, see Chapter D9 The Memory Tagging Extension.

—— Note ———
In the event of repeatedly-contending LoadExcl/StoreExcl instruction sequences from multiple PEs, an
implementation must ensure that forward progress is made by at least one PE.

CONSTRANGED UNPREDICTABLE behavior when Load-Exclusive/Store-Exclusive
access a different number of registers
As stated in this section, an implementation can require that the instructions of a Load-Exclusive/Store-Exclusive
pair access the same number of registers. In such an implementation, this means behavior is CONSTRAINED
UNPREDICTABLE if, in a single thread of execution, either:
• An LDXP instruction of two 32-bit quantities is followed by an STXR instruction of one 64-bit quantity at the
same address.
• An LDXR instruction of one 64-bit quantity is followed by an STXP instruction of two 32-bit quantities at the
same address.
In these cases, the CONSTRAINED UNPREDICTABLE behavior must be one of:
• The STXP or STXR instruction generates an external Data Abort.
• The STXP or STXR instruction generates an IMPLEMENTATION DEFINED MMU fault reported using the Data
Abort Fault status code of ESR_ELx.DFSC = 0b110101.
• The STXP or STXR instruction always fails, returning a status of 1.
• The STXP or STXR instruction always passes, returning a status of 0.
• This STXP or STXR instruction has the same pass or fail behavior that it would have had if the instruction had
used the same size and number of registers as the preceding LDXR or LDXP instruction.
B2.9.6 Use of WFE and SEV instructions by spin-locks

Armv8 provides Wait For Event, Send Event, and Send Event Local instructions, WFE, SEV, and SEVL, that can assist with reducing power consumption and bus contention caused by PEs repeatedly attempting to obtain a spin-lock. These instructions can be used at the application level, but a complete understanding of what they do depends on a system level understanding of exceptions. They are described in *Wait for Event* on page D1-4676. However, in Armv8, when the global monitor for a PE changes from Exclusive Access state to Open Access state, an event is generated.

--- Note

This is equivalent to issuing an SEVL instruction on the PE for which the monitor state has changed. It removes the need for spinlock code to include an SEV instruction after clearing a spinlock.
The AArch64 Application Level Memory Model

B2.9 Synchronization and semaphores
Part C
The AArch64 Instruction Set
Chapter C1
The A64 Instruction Set

This chapter describes the A64 instruction set. It contains the following sections:

- *About the A64 instruction set* on page C1-226.
- *Structure of the A64 assembler language* on page C1-228.
- *Address generation* on page C1-234.
- *Instruction aliases* on page C1-237.
C1.1 About the A64 instruction set

The A64 instruction set is the instruction set supported in the AArch64 Execution state.

All A64 instructions have a width of 32 bits. The A64 encoding structure breaks down into the following functional groups:

- A miscellaneous group of branch instructions, exception generating instructions, and System instructions.
- Data-processing instructions associated with general-purpose registers. These instructions are supported by two functional groups, depending on whether the operands:
  - Are all held in registers.
  - Include an operand with a constant immediate value.
- Load and store instructions associated with the general-purpose register file, the SIMD and floating-point register file, and the SVE register file.
- SIMD and scalar floating-point data-processing instructions that operate on the SIMD and floating-point registers.
- SVE data-processing instructions that operate on the SVE registers.

The encoding hierarchy within a functional group breaks down as follows:

- A functional group consists of a set of related instruction classes. A64 instruction set encoding on page C4-388 provides an overview of the instruction encodings in the form of a list of instruction classes within their functional groups.
- An instruction class consists of a set of related instruction forms. Instruction forms are documented in one of three alphabetic lists:
  - The load, store, and data-processing instructions associated with the general-purpose registers, together with those in the other instruction classes. See Chapter C6 A64 Base Instruction Descriptions.
  - The load, store, and data-processing instructions associated with the SIMD and floating-point support. See Chapter C7 A64 Advanced SIMD and Floating-point Instruction Descriptions.
  - The load, store, and data-processing instructions associated with SVE support. See Chapter C8 SVE Instruction Descriptions.
- An instruction form might support a single instruction syntax. Where an instruction supports more than one syntax, each syntax is an instruction variant. Instruction variants can occur because of differences in:
  - The size or format of the operands.
  - The register file used for the operands.
  - The addressing mode used for load/load/store memory operands.
Instruction variants might also arise as the result of other factors. Instruction variants are described in the instruction description for the individual instructions.

A64 instructions have a regular bit encoding structure:

- 5-bit register operand fields at fixed positions within the instruction. For general-purpose register operands, the values 0-30 select one of 31 registers. The value 31 is used as a special case that can:
  - Indicate use of the current stack pointer, when identifying a load/store base register or in a limited set of data-processing instructions. See The stack pointer registers on page D1-4636.
  - Indicate the value zero when used as a source register operand.
  - Indicate discarding the result when used as a destination register operand.
For SIMD and floating-point register access and SVE register access, the value used selects one of 32 registers.
- Immediate bits that provide constant data-processing values or address offsets are placed in contiguous bitfields. Some computed values in instruction variants use one or more immediate bitfields together with the secondary encoding bitfields.
All encodings that are not fully defined are described as unallocated. An attempt to execute an unallocated instruction is UNDEFINED, unless the behavior is otherwise defined in this Manual.
C1.2 Structure of the A64 assembler language

The following sections describe the A64 assembler syntax:

- General requirements.
- Common syntax terms.
- Instruction Mnemonics on page C1-229.
- Condition code on page C1-229.
- Register names on page C1-230.

C1.2.1 General requirements

The letter W denotes a general-purpose register holding a 32-bit word, and X denotes a general-purpose register holding a 64-bit doubleword.

An A64 assembler recognizes both uppercase and lowercase variants of the instruction mnemonics and register names, but not mixed case variants. An A64 disassembler can output either uppercase or lowercase mnemonics and register names. Program and data labels are case-sensitive.

The A64 assembly language does not require the # character to introduce constant immediate operands, but an assembler must allow immediate values introduced with or without the # character.

In Example C1-1 on page C1-229, the sequence // is used as a comment leader and A64 assemblers are encouraged to accept this syntax.

C1.2.2 Common syntax terms

The following syntax terms are used frequently throughout the A64 instruction set description.

- UPPER: Text in upper-case letters is fixed. Text in lower-case letters is variable. This means that register name Xn indicates that the X is required, followed by a variable register number, for example X29.
- < >: Any text enclosed by angle braces, < >, is a value that the user supplies. Subsequent text might supply additional information.
- {}: Any item enclosed by curly brackets, { }, is optional. A description of the item and how its presence or absence affects the instruction is normally supplied by subsequent text. In some cases curly braces are actual symbols in the syntax, for example when they surround a register list. These cases are called out in the surrounding text.
- []: Any items enclosed by square brackets, [ ], constitute a list of alternative characters. A single one of the characters can be used in that position and the subsequent text describes the meaning of the alternatives. In some case the square brackets are part of the syntax itself, such as addressing modes or vector elements. These cases are called out in the surrounding text.
- a|b: Alternative words are separated by a vertical bar, |, and can be surrounded by parentheses to delimit them. For example, U(ADD|SUB)W represents UADDW or USUBW.
- ±: This indicates an optional + or - sign. If neither is used then + is assumed.
- uimm: An n-bit unsigned, positive, immediate value.
- simm: An n-bit two’s complement, signed immediate value, where n includes the sign bit.
- SP: See Register names on page C1-230.
- Wn: See Register names on page C1-230.
- WSP: See Register names on page C1-230.
- WZR: See Register names on page C1-230.
- Xn: See Register names on page C1-230.
- XZR: See Register names on page C1-230.
C1.2.3 Instruction Mnemonics

The A64 assembly language overloads instruction mnemonics and distinguishes between the different forms of an instruction based on the operand types. For example, the following ADD instructions all have different opcodes. However, the programmer must remember only one mnemonic, as the assembler automatically chooses the correct opcode based on the operands. The disassembler follows the same procedure in reverse.

Example C1-1  ADD instructions with different opcodes

```
ADD W0, W1, W2  // add 32-bit register
ADD X0, X1, X2  // add 64-bit register
ADD X0, X1, W2, SXTW // add 64-bit extended register
ADD X0, X1, #42  // add 64-bit immediate
```

C1.2.4 Condition code

The A64 ISA has some instructions that set Condition flags or test Condition codes or both. For information about instructions that set the Condition flags or use the condition mnemonics, see Condition flags and related instructions on page C6-1167.

Table C1-1 on page C1-229 shows the available Condition codes.

Table C1-1 Condition codes

<table>
<thead>
<tr>
<th>cond</th>
<th>Mnemonic</th>
<th>Meaning (integer)</th>
<th>Meaning (floating-point)a</th>
<th>Condition flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>Equal</td>
<td>Equal</td>
<td>Z == 1</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>Not equal</td>
<td>Not equal or unordered</td>
<td>Z == 0</td>
</tr>
<tr>
<td>0010</td>
<td>CS or HS</td>
<td>Carry set</td>
<td>Greater than, equal, or unordered</td>
<td>C == 1</td>
</tr>
<tr>
<td>0011</td>
<td>CC or LO</td>
<td>Carry clear</td>
<td>Less than</td>
<td>C == 0</td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
<td>Minus, negative</td>
<td>Less than</td>
<td>N == 1</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td>Plus, positive or zero</td>
<td>Greater than, equal, or unordered</td>
<td>N == 0</td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
<td>Overflow</td>
<td>Unordered</td>
<td>V == 1</td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
<td>No overflow</td>
<td>Ordered</td>
<td>V == 0</td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
<td>Unsigned higher</td>
<td>Greater than, or unordered</td>
<td>C == 1 &amp;&amp; Z == 0</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td>Unsigned lower or same</td>
<td>Less than or equal</td>
<td>!(C == 1 &amp;&amp; Z == 0)</td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
<td>Signed greater than or equal</td>
<td>Greater than or equal</td>
<td>N == V</td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td>Signed less than</td>
<td>Less than, or unordered</td>
<td>N! = V</td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
<td>Signed greater than</td>
<td>Greater than</td>
<td>Z == 0 &amp;&amp; N == V</td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
<td>Signed less than or equal</td>
<td>Less than, or unordered</td>
<td>!(Z == 0 &amp;&amp; N == V)</td>
</tr>
<tr>
<td>1110</td>
<td>AL</td>
<td>Always</td>
<td>Always</td>
<td>Any</td>
</tr>
<tr>
<td>1111</td>
<td>NV</td>
<td>Always</td>
<td>Always</td>
<td>Any</td>
</tr>
</tbody>
</table>

a. Unordered means at least one NaN operand.
b. The Condition code NV exists only to provide a valid disassembly of the 0b1111 encoding, otherwise its behavior is identical to AL.
C1.2.5 SVE Condition code aliases

The SVE assembler syntax defines an alternative set of SVE condition code aliases for use with AArch64 conditional instructions, as follows:

Table C1-2 on page C1-230 shows the available SVE Condition code aliases.

<table>
<thead>
<tr>
<th>cond</th>
<th>Mnemonic</th>
<th>SVE alias</th>
<th>Meaning</th>
<th>Condition flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>NONE</td>
<td>All Active elements were FALSE or there were no Active elements.</td>
<td>Z == 1</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>ANY</td>
<td>An Active element was TRUE.</td>
<td>Z == 0</td>
</tr>
<tr>
<td>0010</td>
<td>CS or HS</td>
<td>NLAST</td>
<td>The Last active element was FALSE or there were no Active elements.</td>
<td>C == 1</td>
</tr>
<tr>
<td>0011</td>
<td>CC or L0</td>
<td>LAST</td>
<td>The Last active element was TRUE.</td>
<td>C == 0</td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
<td>FIRST</td>
<td>The First active element was TRUE.</td>
<td>N == 1</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td>NRFST</td>
<td>The First active element was FALSE or there were no Active elements.</td>
<td>N == 0</td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
<td>-</td>
<td>CTERM comparison failed, but end of partition reached.</td>
<td>V == 1</td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
<td>-</td>
<td>CTERM comparison succeeded, or end of partition not reached.</td>
<td>V == 0</td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
<td></td>
<td>An Active element was TRUE, but the Last active element was FALSE.</td>
<td>C == 1 &amp;&amp; Z == 0</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td>PLAST</td>
<td>The Last active element was TRUE, or all Active elements were FALSE, or there were no Active elements.</td>
<td>C == 1</td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
<td>TCONT</td>
<td>CTERM termination condition not detected.</td>
<td>N == V</td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td>TSTOP</td>
<td>CTERM termination condition detected.</td>
<td>N != V</td>
</tr>
</tbody>
</table>

C1.2.6 Register names

This section describes the AArch64 registers. It contains the following subsections:

- General-purpose register file and zero register and stack pointer.
- SIMD and floating-point register file on page C1-231.
- SIMD and floating-point scalar register names on page C1-232.
- SIMD vector register names on page C1-232.
- SIMD vector element names on page C1-232.

General-purpose register file and zero register and stack pointer

The 31 general-purpose registers in the general-purpose register file are named R0-R30 and encoded in the instruction register fields with values 0-30. In a general-purpose register field the value 31 represents either the current stack pointer or the zero register, depending on the instruction and the operand position.

When the registers are used in a specific instruction variant, they must be qualified to indicate the operand data size, 32 bits or 64 bits, and the data size of the instruction.

When the data size is 32 bits, the lower 32 bits of the register are used and the upper 32 bits are ignored on a read and cleared to zero on a write.
Table C1-3 shows the qualified names for registers, where \( n \) is a register number 0-30.

### Table C1-3 Naming of general-purpose registers, the zero register, and the stack pointer

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Encoding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wn</td>
<td>32 bits</td>
<td>0-30</td>
<td>General-purpose register 0-30</td>
</tr>
<tr>
<td>Xn</td>
<td>64 bits</td>
<td>0-30</td>
<td>General-purpose register 0-30</td>
</tr>
<tr>
<td>WZR</td>
<td>32 bits</td>
<td>31</td>
<td>Zero register</td>
</tr>
<tr>
<td>XZR</td>
<td>64 bits</td>
<td>31</td>
<td>Zero register</td>
</tr>
<tr>
<td>WSP</td>
<td>32 bits</td>
<td>31</td>
<td>Current stack pointer</td>
</tr>
<tr>
<td>SP</td>
<td>64 bits</td>
<td>31</td>
<td>Current stack pointer</td>
</tr>
</tbody>
</table>

This list gives more information about the instruction arguments shown in Table C1-3:

- The names Xn and Wn both refer to the same general-purpose register, Rn.
- There is no register named W31 or X31.
- The name SP represents the stack pointer for 64-bit operands where an encoding of the value 31 in the corresponding register field is interpreted as a read or write of the current stack pointer. When instructions do not interpret this operand encoding as the stack pointer, use of the name SP is an error.
- The name WSP represents the current stack pointer in a 32-bit context.
- The name XZR represents the zero register for 64-bit operands where an encoding of the value 31 in the corresponding register field is interpreted as returning zero when read or discarding the result when written. When instructions do not interpret this operand encoding as the zero register, use of the name XZR is an error.
- The name WZR represents the zero register in a 32-bit context.
- The architecture does not define a specific name for general-purpose register R30 to reflect its role as the link register on procedure calls. However, an A64 assembler must always use W30 and X30 for this purpose, and additional software names might be defined as part of the Procedure Call Standard, see Procedure Call Standard for the Arm 64-bit Architecture.

### SIMD and floating-point register file

The 32 registers in the SIMD and floating-point register file, V0-V31, hold floating-point operands for the scalar floating-point instructions, and both scalar and vector operands for the SIMD instructions. When they are used in a specific instruction form, the names must be further qualified to indicate the data shape, that is the data element size and the number of elements or lanes within the register. A similar requirement is placed on the general-purpose registers. See General-purpose register file and zero register and stack pointer on page C1-230.

--- Note ---

The data type is described by the instruction mnemonics that operate on the data. The data type is not described by the register name. The data type is the interpretation of bits within each register or vector element, whether these are integers, floating-point values, polynomials, or cryptographic hashes.
SIMD and floating-point scalar register names

SIMD and floating-point instructions that operate on scalar data only access the lower bits of a SIMD and floating-point register. The unused high bits are ignored on a read and cleared to 0 on a write.

Table C1-4 shows the qualified names for accessing scalar SIMD and floating-point registers. The letter $n$ denotes a register number between 0 and 31.

Table C1-4 SIMD and floating-point scalar register names

<table>
<thead>
<tr>
<th>Size</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>$8n$</td>
</tr>
<tr>
<td>16 bits</td>
<td>$16n$</td>
</tr>
<tr>
<td>32 bits</td>
<td>$32n$</td>
</tr>
<tr>
<td>64 bits</td>
<td>$64n$</td>
</tr>
<tr>
<td>128 bits</td>
<td>$128n$</td>
</tr>
</tbody>
</table>

SIMD vector register names

If a register holds multiple data elements on which arithmetic is performed in a parallel, SIMD, manner, then a qualifier describes the vector shape. The vector shape is the element size and the number of elements or lanes. If the element size in bits multiplied by the number of lanes does not equal 128, then the upper 64 bits of the register are ignored on a read and cleared to zero on a write.

Table C1-5 shows the SIMD vector register names. The letter $n$ denotes a register number between 0 and 31.

Table C1-5 SIMD vector register names

<table>
<thead>
<tr>
<th>Shape</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits × 8 lanes</td>
<td>$Vn.8B$</td>
</tr>
<tr>
<td>8 bits × 16 lanes</td>
<td>$Vn.16B$</td>
</tr>
<tr>
<td>16 bits × 4 lanes</td>
<td>$Vn.4H$</td>
</tr>
<tr>
<td>16 bits × 8 lanes</td>
<td>$Vn.8H$</td>
</tr>
<tr>
<td>32 bits × 2 lanes</td>
<td>$Vn.2S$</td>
</tr>
<tr>
<td>32 bits × 4 lanes</td>
<td>$Vn.4S$</td>
</tr>
<tr>
<td>64 bits × 1 lane</td>
<td>$Vn.1D$</td>
</tr>
<tr>
<td>64 bits × 2 lanes</td>
<td>$Vn.2D$</td>
</tr>
</tbody>
</table>

SIMD vector element names

Appending a constant, zero-based element index to the register name inside square brackets indicates that a single element from a SIMD and floating-point register is used as a scalar operand. The number of lanes is not represented, as it is not encoded in the instruction and can only be inferred from the index value.
Table C1-6 shows the vector register names and the element index. The letter $i$ denotes the element index.

<table>
<thead>
<tr>
<th>Size</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>Vn.8[i]</td>
</tr>
<tr>
<td>16 bits</td>
<td>Vn.16[i]</td>
</tr>
<tr>
<td>32 bits</td>
<td>Vn.32[i]</td>
</tr>
<tr>
<td>64 bits</td>
<td>Vn.64[i]</td>
</tr>
</tbody>
</table>

An assembler must accept a fully qualified SIMD register name if the number of lanes is greater than the index value. See **SIMD vector register names** on page C1-232. For example, an assembler must accept all of the following forms as the name for the 32-bit element in bits [63:32] of the SIMD and floating-point register V9:

- V9.2S[1] //optional number of lanes
- V9.4S[1] //optional number of lanes

**Note**

The SIMD and floating-point register element name Vn.5[0] is not equivalent to the scalar SIMD and floating-point register name Sn. Although they represent the same bits in the register, they select different instruction encoding forms, either the vector element or the scalar form.

**SIMD vector register list**

Where an instruction operates on multiple SIMD and floating-point registers, for example vector load/store structure and table lookup operations, the registers are specified as a list enclosed by curly braces. This list consists of either a sequence of registers separated by commas, or a register range separated by a hyphen. The registers must be numbered in increasing order, modulo 32, in increments of one. The hyphenated form is preferred for disassembly if there are more than two registers in the list and the register number are increasing. The following examples are equivalent representations of a set of four registers V4 to V7, each holding four lanes of 32-bit elements:

```
{ V4.4S - V7.4S } //standard disassembly
{ V4.4S, V5.4S, V6.4S, V7.4S } //alternative representation
```

**SIMD vector element list**

Registers in a list can also have a vector element form. For example, the LD4 instruction can load one element into each of four registers, and in this case the index is appended to the list as follows:

```
{ V4.4S, V5.4S, V6.4S, V7.4S }[3] //alternative with optional number of lanes
```
C1.3 Address generation

The A64 instruction set supports 64-bit virtual addresses (VAs). The valid VA range is determined by the following factors:

- The size of the implemented virtual address space.
- Memory Management Unit (MMU) configuration settings.

Limits on the VA size mean that the most significant bits of the virtual address do not hold valid address bits. These unused bits can hold:

- A tag, see Address tagging on page D8-5162.
- If FEAT_PAuth is implemented, a Pointer authentication code (PAC), see Pointer authentication on page D8-5164.

For more information on memory management and address translation, see Chapter D8 The AArch64 Virtual Memory System Architecture.

C1.3.1 Register indexed addressing

The A64 instruction set allows a 64-bit index register to be added to the 64-bit base register, with optional scaling of the index by the access size. Additionally it allows for sign-extension or zero-extension of a 32-bit value within an index register, followed by optional scaling.

C1.3.2 PC-relative addressing

The A64 instruction set has support for position-independent code and data addressing:

- PC-relative literal loads have an offset range of ± 1MB.
- Process state flag and compare based conditional branches have a range of ± 1MB. Test bit conditional branches have a restricted range of ± 32KB.
- Unconditional branches, including branch and link, have a range of ± 128MB.

PC-relative load/store operations, and address generation with a range of ± 4GB can be performed using two instructions.

C1.3.3 Load/store addressing modes

Load/store addressing modes in the A64 instruction set require a 64-bit base address from a general-purpose register X0-X30 or the current stack pointer, SP, with an optional immediate or register offset. Table C1-7 on page C1-234 shows the assembler syntax for the complete set of load/store addressing modes.

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Immediate</th>
<th>Register</th>
<th>Extended Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base register only (no offset)</td>
<td>[base{, #0}]</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Base plus offset</td>
<td>[base{, #imm}]</td>
<td>[base, Xm{, LSL #imm}]</td>
<td>[base, Wm, (S</td>
</tr>
<tr>
<td>Pre-indexed</td>
<td>[base, #imm]!</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Post-indexed</td>
<td>[base], #imm</td>
<td>[base], Xm</td>
<td>-</td>
</tr>
<tr>
<td>Literal (PC-relative)</td>
<td>label</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

a. The post-indexed by register offset mode can be used with the SIMD load/store structure instructions described in Load/store Advanced SIMD on page C3-265. Otherwise the post-indexed by register offset mode is not available.
Some types of load/store instruction support only a subset of the load/store addressing modes listed in Table C1-7 on page C1-234. Details of the supported modes are as follows:

- Base plus offset addressing means that the address is the value in the 64-bit base register plus an offset.
- Pre-indexed addressing means that the address is the sum of the value in the 64-bit base register and an offset, and the address is then written back to the base register.
- Post-indexed addressing means that the address is the value in the 64-bit base register, and the sum of the address and the offset is then written back to the base register.
- Literal addressing means that the address is the value of the 64-bit program counter for this instruction plus a 19-bit signed word offset. This means that it is a 4 byte aligned address within ±1MB of the address of this instruction with no offset. Literal addressing can be used only for loads of at least 32 bits and for prefetch instructions. The PC cannot be referenced using any other addressing modes. The syntax for labels is specific to individual toolchains.
- An immediate offset can be unsigned or signed, and scaled or unscaled, depending on the type of load/store instruction. When the immediate offset is scaled it is encoded as a multiple of the transfer size, although the assembly language always uses a byte offset, and the assembler or disassembler performs the necessary conversion. The usable byte offsets therefore depend on the type of load/store instruction and the transfer size.

Table C1-8 shows the offset and the type of load/store instruction.

<table>
<thead>
<tr>
<th>Offset bits</th>
<th>Sign</th>
<th>Scaling</th>
<th>Write-Back</th>
<th>Load/store type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Exclusive/acquire/release</td>
</tr>
<tr>
<td>7</td>
<td>Signed</td>
<td>Scaled</td>
<td>Optional</td>
<td>Register pair</td>
</tr>
<tr>
<td>9</td>
<td>Signed</td>
<td>Unscaled</td>
<td>Optional</td>
<td>Single register</td>
</tr>
<tr>
<td>12</td>
<td>Unsigned</td>
<td>Scaled</td>
<td>No</td>
<td>Single register</td>
</tr>
</tbody>
</table>

- A register offset means that the offset is the 64 bits from a general-purpose register, Xm, optionally scaled by the transfer size, in bytes, if LSL #imm is present and where imm must be equal to log2(transfer_size). The SXTX extend/shift option is functionally equivalent to LSL, but the LSL option is preferred in source code.
- An extended register offset means that offset is the bottom 32 bits from a general-purpose register Wm, sign-extended or zero-extended to 64 bits, and then scaled by the transfer size if so indicated by #imm, where imm must be equal to log2(transfer_size). An assembler must accept Wm or Xm as an extended register offset, but Wm is preferred for disassembly.
- Generating an address lower than the value in the base register requires a negative signed immediate offset or a register offset holding a negative value.
- When stack alignment checking is enabled by system software and the base register is the SP, the current stack pointer must be initially quadword aligned, that is aligned to 16 bytes. Misalignment generates a Stack Alignment fault. The offset does not have to be a multiple of 16 bytes unless the specific load/store instruction requires this. SP cannot be used as a register offset.
Address calculation

General-purpose arithmetic instructions can calculate the result of most addressing modes and write the address to a general-purpose register or, in most cases, to the current stack pointer.

Table C1-9 on page C1-236 shows the arithmetic instructions that can compute addressing modes.

### Table C1-9 Arithmetic instructions to compute addressing modes

<table>
<thead>
<tr>
<th>Addressing Form</th>
<th>Immediate</th>
<th>Register</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base register (no offset)</td>
<td>MOV Xd</td>
<td>SP, base</td>
<td>-</td>
</tr>
<tr>
<td>Base plus offset</td>
<td>ADD Xd</td>
<td>SP, base, #imm</td>
<td>ADD &lt;Xd</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>SUB Xd</td>
<td>SP, base, #imm</td>
</tr>
<tr>
<td>Pre-indexed</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Post-indexed</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Literal (PC-relative)</td>
<td>ADR Xd, label</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

#### Note

- For the 64-bit base plus register offset form, the UTX mnemonic is an alias for the LSL shift option, but LSL is preferred for disassembly. Similarly the SXTX extend/shift option is functionally equivalent to the LSL option, but the LSL option is preferred in source code.

- To calculate a base plus immediate offset the ADD instructions defined in Arithmetic (immediate) on page C3-298 accept an unsigned 12-bit immediate offset, with an optional left shift by 12. This means that a single ADD instruction cannot support the full range of byte offsets available to a single register load/store with a scaled 12-bit immediate offset. For example, a quadword LDR effectively has a 16-bit byte offset. To calculate an address with a byte offset that requires more than 12 bits it is necessary to use two ADD instructions. The following example shows this:

```c
ADD Xd, base, #((imm & 0xFFF)
ADD Xd, Xd, #((imm>>12), LSL #12
```

- To calculate a base plus extended register offset, the ADD instructions defined in Arithmetic (extended register) on page C3-304 provide a superset of the addressing mode that also supports sign-extension or zero-extension of a byte or halfword value with any shift amount between 0 and 4, for example:

```c
ADD Xd, base, Wm, SXTW #3 // Xd = base + (SignExtend(Wm) LSL 3)
ADD Xd, base, Wm, UXTH #4 // Xd = base + (ZeroExtend(Wm<15:0>) LSL 4)
```

- If the same extended register offset is used by more than one load/store instruction, then, depending on the implementation, it might be more efficient to calculate the extended and scaled intermediate result just once, and then reuse it as a simple register offset. The extend and scale calculation can be performed using the SBFIZ and UBFIZ bitfield instructions defined in Bitfield move on page C3-300, for example:

```c
SBFIZ Xd, Xm, #3, #32 //Xd = "Wm, SXTW #3"
UBFIZ Xd, Xm, #4, #16 //Xd = "Wm, UXTH #4"
```
C1.4 Instruction aliases

Some instructions have an associated *architecture alias* that is used for disassembly of the encoding when the associated conditions are met. Architecture alias instructions are included in the alphabetic lists of instruction types and clearly presented as an alias form in descriptions for the individual instructions.
Chapter C2
About the A64 Instruction Descriptions

This chapter describes the instruction descriptions contained in Chapter C6 A64 Base Instruction Descriptions, Chapter C7 A64 Advanced SIMD and Floating-point Instruction Descriptions, and Chapter C8 SVE Instruction Descriptions.

It contains the following sections:

• Understanding the A64 instruction descriptions on page C2-240.
• General information about the A64 instruction descriptions on page C2-243.
C2.1 Understanding the A64 instruction descriptions

Each instruction description in Chapter C6 and Chapter C7 has the following content:
1. A title.
2. An introduction to the instruction.
3. The instruction encoding or encodings.
4. Any alias conditions.
5. A list of the assembler symbols for the instruction.
6. Pseudocode describing how the instruction operates.
7. Notes, if applicable.

The following sections describe each of these.

C2.1.1 The title

The title of an instruction description includes the base mnemonic for the instruction. If different forms of an instruction use the same base mnemonic, each form has its own description. In this case, the title is the mnemonic followed by a short description of the instruction form in parentheses. This is most often used when an operand is an immediate value in one instruction form, but is a register in another form.

For example, in Chapter C6 there are the following titles for different forms of the `ADD` instruction:
- `ADD (extended register)` on page C6-1174.
- `ADD (immediate)` on page C6-1177.
- `ADD (shifted register)` on page C6-1179.

C2.1.2 An introduction to the instruction

This briefly describes the function of the instruction. The introduction is not a complete description of the instruction, and it is not definitive. If there is any conflict between it and the more detailed information that follows it, the more detailed information takes priority.

C2.1.3 The instruction encoding or encodings

This shows the instruction encoding diagram, or if the instruction has more than one encoding, shows all of the encoding diagrams. Each diagram has a subheading.

For example, for load and store instructions, the subheadings might be:
- Post-index.
- Pre-index.
- Unsigned offset.

Each diagram numbers the bits from 31 to 0. The diagram for an instruction at address $A$ shows, from left to right, the bytes at addresses $A+3$, $A+2$, $A+1$, and $A$.

There might be variants of an encoding, if the assembler syntax prototype differs depending on the value in one or more of the encoding fields. In this case, each variant has a subheading that describes the variant and shows the distinguishing field value or values in parentheses. For example, in Chapter C6 there are the following subheadings for variants of the `ADC` instruction encoding:
- 32-bit variant ($sf = 0$).
- 64-bit variant ($sf = 1$).

The assembler syntax prototype for an encoding or variant of an encoding shows how to form a complete assembler source code instruction that assembles to the encoding. Unless otherwise stated, the prototype is also the preferred syntax for a disassembler to disassemble the encoding to. Disassemblers are permitted to omit optional symbols that represent the default value of a field or set of fields, to produce more readable disassembled code, provided that the output re-assembles to the same encoding.
Each encoding diagram, and its associated assembler syntax prototypes, is followed by encoding-specific pseudocode that translates the fields of that encoding into inputs for the encoding-independent pseudocode that describes the operation of the instruction. See Pseudocode describing how the instruction operates on page C2-242.

C2.1.4 Any alias conditions, if applicable

This is an optional part of an instruction description. If included, it describes the set of conditions for which an alternative mnemonic and its associated assembler syntax prototypes are preferred for disassembly by a disassembler. It includes a link to the alias instruction description that defines the alternative syntax. The alias syntax and the original syntax can be used interchangeably in the assembler source code.

Arm recommends that if a disassembler outputs the alias syntax, it consistently outputs the alias syntax.

C2.1.5 A list of the assembler symbols for the instruction

The Assembler symbols subsection of the instruction description contains a list of the symbols that the assembler syntax prototype or prototypes use, if any.

In assembler syntax prototypes, the following conventions are used:

- < > Angle brackets. Any symbol enclosed by these is a name or a value that the user supplies. For each symbol, there is a description of what the symbol represents. The description usually also specifies which encoding field or fields encodes the symbol.

- { } Brace brackets. Any symbols enclosed by these are optional. For each optional symbol, there is a description of what the symbol represents and how its presence or absence is encoded.

In some assembler syntax prototypes, some brace brackets are mandatory, for example if they surround a register list. When the use of brace brackets is mandatory, they are separated from other syntax items by one or more spaces.

- # This usually precedes a numeric constant. All uses of # are optional in A64 assembler source code. Arm recommends that disassemblers output the # where the assembler syntax prototype includes it.

- +/- This indicates an optional + or - sign. If neither is coded, + is assumed.

Single spaces are used for clarity, to separate syntax items. Where a space is mandatory, the assembler syntax prototype shows two or more consecutive spaces.

Any characters not shown in this conventions list must be coded exactly as shown in the assembler syntax prototype. Apart from brace brackets, the characters shown are used as part of a meta-language to define the architectural assembler syntax for an instruction encoding or alias, but have no architecturally defined significance in the input to an assembler or in the output from a disassembler.

The following symbol conventions are used:

- <Xn> The 64-bit name of a general-purpose register (X0-X30) or the zero register (XZR).

- <Wn> The 32-bit name of a general-purpose register (W0-W30) or the zero register (WZR).

- <Xn|SP> The 64-bit name of a general-purpose register (X0-X30) or the current stack pointer (SP).

- <Wn|WSP> The 32-bit name of a general-purpose register (W0-W30) or the current stack pointer (WSP).

- <Bn>, <Hn>, <Sn>, <Dn>, <Qn> The 8, 16, 32, 64, or 128-bit name of a SIMD and floating-point register in a scalar context, as described in Register names on page C1-230.

- <Vn> The name of a SIMD and floating-point register in a vector context, as described in Register names on page C1-230.

- <Zn> The name of an SVE vector register, as described in SVE vector registers on page B1-140.

- <Pn> The name of an SVE predicate register, as described in SVE predicate registers on page B1-141.
If the description of a symbol specifies that the symbol is a register, the description might also specify that the range of permitted registers is extended or restricted. It also specifies any differences from the default rules for such fields.

--- Note ---

*Register names on page C1-230* provides the A64 register names.

### C2.1.6 Pseudocode describing how the instruction operates

The *Operation* subsection of the instruction description contains this pseudocode.

It is encoding-independent pseudocode that provides a precise description of what the instruction does.

--- Note ---

For a description of Arm pseudocode, see *Appendix K16 Arm Pseudocode Definition*. This appendix also describes the execution model for an instruction.

### C2.1.7 Notes, if applicable

If applicable, other notes about the instruction appear under additional subheadings.
C2.2 General information about the A64 instruction descriptions

This section provides general information about the A64 instruction descriptions. Some of this information also applies to System register descriptions, for example the terms defined in Fixed values in AArch64 instruction and System register descriptions apply to the AArch64 descriptions throughout this manual. The following subsections provide this information:

- Execution of instructions in debug state.
- Fixed values in AArch64 instruction and System register descriptions.
- Modified immediate constants in A64 instructions on page C2-244.

C2.2.1 Execution of instructions in debug state

In general, except for the instructions described in Debug state on page C3-250, the A64 instruction descriptions do not indicate any differences in the behavior of the instruction if it is executed in Debug state. For this information, see Executing instructions in Debug state on page H2-10243.

Note

For many instructions, execution is unchanged in Debug state. Executing instructions in Debug state on page H2-10243 identifies these instructions.

C2.2.2 Fixed values in AArch64 instruction and System register descriptions

This section summarizes the terms used to describe fixed values in AArch64 register and instruction descriptions. The Glossary gives full descriptions of these terms, and each entry in this section includes a link to the corresponding Glossary entry.

Note

In register descriptions, the meaning of some bits depends on the PE state. This affects the definitions of RES0 and RES1, as shown in the Glossary.

The following terms are used to describe bits or fields with fixed values:

- **RAZ** Read-As-Zero. See Read-As-Zero (RAZ).
  - In diagrams, a RAZ bit can be shown as 0.

- **(0), RES0** Reserved, Should-Be-Zero (SBZ) or RES0.
  - In instruction encoding diagrams, and sometimes in other descriptions, (0) indicates an SBZ bit. If the bit is set to 1, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:
    - The instruction is UNDEFINED.
    - The instruction is treated as a NOP.
    - The instruction executes as if the value of the bit was 0.
    - Any destination registers of the instruction become UNKNOWN.
  - This notation can be expanded for fields, so a three-bit field can be shown as either (0)(0)(0) or as (000).
  - In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES0. See the Glossary definition of RES0 for more information.

Note

Some of the System instruction descriptions in this chapter are based on the field description of the input value for the instruction. These are register descriptions and therefore can include RES0 fields, the (0) and RES0 descriptions can be applied to bits or bit fields that are read-only, or are write-only. The Glossary definitions cover these cases.

- **RAO** Read-As-One. See Read-As-One (RAO).
  - In diagrams, a RAO bit can be shown as 1.

- **(1), RES1** Reserved, Should-Be-One (SBO) or RES1.
In instruction encoding diagrams, and sometimes in other descriptions, (1) indicates an SBO bit. If the bit is set to 0, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if the value of the bit was 1.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either (1)(1)(1) or as (111).

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES1. See the Glossary definition of RES1 for more information.

**Note**

Some of the System instruction descriptions in this chapter are based on the field description of the input value for the instruction. These are register descriptions and therefore can include RES1 fields, and RES1 descriptions can be applied to bits or bit fields that are read-only, or are write-only. The Glossary definitions cover these cases.

### C2.2.3 Modified immediate constants in A64 instructions

It contains the following subsections:

- Modified immediate constants in A64 floating-point instructions.

#### Modified immediate constants in A64 floating-point instructions

Table C2-1 on page C2-244 shows the immediate constants available in FMOV (scalar, immediate) and FMOV (vector, immediate) floating-point instructions.

<table>
<thead>
<tr>
<th>Data type</th>
<th>immediate</th>
<th>Constant a</th>
</tr>
</thead>
<tbody>
<tr>
<td>F16</td>
<td>abcdefgh</td>
<td>aBbcdefgh0000000</td>
</tr>
<tr>
<td>F32</td>
<td>abcdefgh</td>
<td>aBbbbbbc defgh0000000000000000</td>
</tr>
<tr>
<td>F64</td>
<td>abcdefgh</td>
<td>aBbbbbbb bcdefgh00000000000000000000000000</td>
</tr>
</tbody>
</table>

a. In this column, B = NOT(b). The bit pattern represents the floating-point number \((-1)^s \times 2^{exp} \times \text{mantissa}\), where

\[ S = \text{UInt}(a), \quad \text{exp} = \text{UInt}(\text{NOT}(b)\text{::c:d}) - 1 \quad \text{and} \quad \text{mantissa} = (16+\text{UInt}(e:f:g:h))/16. \]

The immediate value shown in the table is either:

- The value of the imm8 field for an FMOV (scalar, immediate) instruction, see FMOV (scalar, immediate) on page C7-2340.
- The value obtained by concatenating the a:b:c:d:e:f:g:h fields for an FMOV (vector, immediate) instruction, see FMOV (vector, immediate) on page C7-2332.
Table C2-2 shows the floating-point constant values encoded in the $b:c:d:e:f:g:h$ fields of the FMOV (vector, immediate) instruction.

<table>
<thead>
<tr>
<th>$efgh$</th>
<th>$bcd$</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>2.0</td>
<td>4.0</td>
<td>8.0</td>
<td>16.0</td>
<td>0.125</td>
<td>0.25</td>
<td>0.5</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>2.125</td>
<td>4.25</td>
<td>8.5</td>
<td>17.0</td>
<td>0.1328125</td>
<td>0.265625</td>
<td>0.53125</td>
<td>1.0625</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>2.25</td>
<td>4.5</td>
<td>9.0</td>
<td>18.0</td>
<td>0.140625</td>
<td>0.28125</td>
<td>0.5625</td>
<td>1.125</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>2.375</td>
<td>4.75</td>
<td>9.5</td>
<td>19.0</td>
<td>0.1484375</td>
<td>0.296875</td>
<td>0.59375</td>
<td>1.1875</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>2.5</td>
<td>5.0</td>
<td>10.0</td>
<td>20.0</td>
<td>0.15625</td>
<td>0.3125</td>
<td>0.625</td>
<td>1.25</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>2.625</td>
<td>5.25</td>
<td>10.5</td>
<td>21.0</td>
<td>0.1640625</td>
<td>0.328125</td>
<td>0.65625</td>
<td>1.3125</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>2.75</td>
<td>5.5</td>
<td>11.0</td>
<td>22.0</td>
<td>0.171875</td>
<td>0.34375</td>
<td>0.6875</td>
<td>1.375</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>2.875</td>
<td>5.75</td>
<td>11.5</td>
<td>23.0</td>
<td>0.1796875</td>
<td>0.359375</td>
<td>0.71875</td>
<td>1.4375</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>3.0</td>
<td>6.0</td>
<td>12.0</td>
<td>24.0</td>
<td>0.1875</td>
<td>0.375</td>
<td>0.75</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>3.125</td>
<td>6.25</td>
<td>12.5</td>
<td>25.0</td>
<td>0.1953125</td>
<td>0.390625</td>
<td>0.78125</td>
<td>1.5625</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>3.25</td>
<td>6.5</td>
<td>13.0</td>
<td>26.0</td>
<td>0.203125</td>
<td>0.40625</td>
<td>0.8125</td>
<td>1.625</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>3.375</td>
<td>6.75</td>
<td>13.5</td>
<td>27.0</td>
<td>0.2109375</td>
<td>0.421875</td>
<td>0.84375</td>
<td>1.6875</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>3.5</td>
<td>7.0</td>
<td>14.0</td>
<td>28.0</td>
<td>0.21875</td>
<td>0.4375</td>
<td>0.875</td>
<td>1.75</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>3.625</td>
<td>7.25</td>
<td>14.5</td>
<td>29.0</td>
<td>0.2265625</td>
<td>0.453125</td>
<td>0.90625</td>
<td>1.8125</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>3.75</td>
<td>7.5</td>
<td>15.0</td>
<td>30.0</td>
<td>0.234375</td>
<td>0.46875</td>
<td>0.9375</td>
<td>1.875</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>3.875</td>
<td>7.75</td>
<td>15.5</td>
<td>31.0</td>
<td>0.2421875</td>
<td>0.484375</td>
<td>0.96875</td>
<td>1.9375</td>
<td></td>
</tr>
</tbody>
</table>

**Operation of modified immediate constants, floating-point instructions**

For an A64 floating-point instruction that uses a modified immediate constant, the operation described by the \textit{VFPExpandImm()} pseudocode function returns the value of the immediate constant.
About the A64 Instruction Descriptions
C2.2 General information about the A64 instruction descriptions
Chapter C3
A64 Instruction Set Overview

This chapter provides an overview of the A64 instruction set. It contains the following sections:

• Branches, Exception generating, and System instructions on page C3-248.
• Loads and stores on page C3-256.
• Loads and stores - SVE on page C3-287.
• Data processing - immediate on page C3-298.
• Data processing - register on page C3-303.
• Data processing - SIMD and floating-point on page C3-311.
• Data processing - SVE on page C3-339.
• Data processing - SVE2 on page C3-368.

For a structured breakdown of instruction groups by encoding, see Chapter C4 A64 Instruction Set Encoding.
C3.1 Branches, Exception generating, and System instructions

This section describes the branch, exception generating, and System instructions. It contains the following subsections:

- Conditional branch.
- Unconditional branch (immediate) on page C3-249.
- Unconditional branch (register) on page C3-249.
- Exception generation and return on page C3-249.
- System register instructions on page C3-250.
- System instructions on page C3-250.
- Hint instructions on page C3-251.
- Barriers and CLREX instructions on page C3-251.
- Pointer authentication instructions on page C3-252.

For information about the encoding structure of the instructions in this instruction group, see Branches, Exception Generating and System instructions on page C4-534.

——— Note ————

Software must:

- Use only BLR or BL to perform a nested subroutine call when that subroutine is expected to return to the immediately following instruction, that is, the instruction with the address of the BLR or BL instruction incremented by four.
- Use only RET to perform a subroutine return, when that subroutine is expected to have been entered by a BL or BLR instruction.
- Use only B, BR, or the instructions listed in Table C3-1 on page C3-248 to perform a control transfer that is not a subroutine call or subroutine return described in this Note.

C3.1.1 Conditional branch

Conditional branches change the flow of execution depending on the current state of the Condition flags or the value in a general-purpose register. See Table C1-1 on page C1-229 for a list of the Condition codes that can be used for cond.

Table C3-1 on page C3-248 shows the Conditional branch instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Branch offset range from the PC</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>B, cond</td>
<td>Branch conditionally</td>
<td>±1MB</td>
<td>B.cond on page C6-1215</td>
</tr>
<tr>
<td>BC, cond</td>
<td>Branch Consistent conditionally</td>
<td>±1MB</td>
<td>BC.cond on page C6-1216</td>
</tr>
<tr>
<td>CBNZ</td>
<td>Compare and branch if nonzero</td>
<td>±1MB</td>
<td>CBNZ on page C6-1250</td>
</tr>
<tr>
<td>CBZ</td>
<td>Compare and branch if zero</td>
<td>±1MB</td>
<td>CBZ on page C6-1251</td>
</tr>
<tr>
<td>TBNZ</td>
<td>Test bit and branch if nonzero</td>
<td>±32KB</td>
<td>TBNZ on page C6-1996</td>
</tr>
<tr>
<td>TBZ</td>
<td>Test bit and branch if zero</td>
<td>±32KB</td>
<td>TBZ on page C6-1997</td>
</tr>
</tbody>
</table>
C3.1.2 Unconditional branch (immediate)

Unconditional branch (immediate) instructions change the flow of execution unconditionally by adding an immediate offset with a range of ±128MB to the value of the program counter that fetched the instruction. The BL instruction also writes the address of the sequentially following instruction to general-purpose register, X30.

Table C3-2 on page C3-249 shows the Unconditional branch instructions with an immediate branch offset.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Immediate branch offset range from the PC</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Branch unconditionally</td>
<td>±128MB</td>
<td>B on page C6-1214</td>
</tr>
<tr>
<td>BL</td>
<td>Branch with link</td>
<td>±128MB</td>
<td>BL on page C6-1229</td>
</tr>
</tbody>
</table>

C3.1.3 Unconditional branch (register)

Unconditional branch (register) instructions change the flow of execution unconditionally by setting the program counter to the value in a general-purpose register. The BLR instruction also writes the address of the sequentially following instruction to general-purpose register X30. The RET instruction behaves identically to BR, but provides an additional hint to the PE that this is a return from a subroutine. Table C3-3 on page C3-249 shows Unconditional branch instructions that jump directly to an address held in a general-purpose register.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLR</td>
<td>Branch with link to register</td>
<td>BLR on page C6-1230</td>
</tr>
<tr>
<td>BR</td>
<td>Branch to register</td>
<td>BR on page C6-1233</td>
</tr>
<tr>
<td>RET</td>
<td>Return from subroutine</td>
<td>RET on page C6-1754</td>
</tr>
</tbody>
</table>

C3.1.4 Exception generation and return

This section describes the following exceptions:

- Exception generating.
- Exception return on page C3-250.
- Debug state on page C3-250.

Exception generating

Table C3-4 on page C3-249 shows the Exception generating instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRK</td>
<td>Breakpoint Instruction</td>
<td>BRK on page C6-1237</td>
</tr>
<tr>
<td>HLT</td>
<td>Halt Instruction</td>
<td>HLT on page C6-1506</td>
</tr>
<tr>
<td>HVC</td>
<td>Generate exception targeting Exception level 2</td>
<td>HVC on page C6-1507</td>
</tr>
<tr>
<td>SMC</td>
<td>Generate exception targeting Exception level 3</td>
<td>SMC on page C6-1824</td>
</tr>
<tr>
<td>SVC</td>
<td>Generate exception targeting Exception level 1</td>
<td>SVC on page C6-1981</td>
</tr>
</tbody>
</table>
### Exception return

Table C3-5 shows the Exception return instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERET</td>
<td>Exception return using current ELR and SPSR</td>
<td>ERET on page C6-1498</td>
</tr>
</tbody>
</table>

### Debug state

Table C3-6 shows the Debug state instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCP51</td>
<td>Debug switch to Exception level 1</td>
<td>DCP51 on page C6-1481</td>
</tr>
<tr>
<td>DCP52</td>
<td>Debug switch to Exception level 2</td>
<td>DCP52 on page C6-1482</td>
</tr>
<tr>
<td>DCP53</td>
<td>Debug switch to Exception level 3</td>
<td>DCP53 on page C6-1483</td>
</tr>
<tr>
<td>DRPS</td>
<td>Debug restore PE state</td>
<td>DRPS on page C6-1487</td>
</tr>
</tbody>
</table>

### System register instructions

For detailed information about the System register instructions, see Chapter C5 The A64 System Instruction Class. Table C3-7 on page C3-250 shows the System register instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRS</td>
<td>Move System register to general-purpose register</td>
<td>MRS on page C6-1707</td>
</tr>
<tr>
<td>MSR</td>
<td>Move general-purpose register to System register</td>
<td>MSR (register) on page C6-1712</td>
</tr>
<tr>
<td></td>
<td>Move immediate to PE state field</td>
<td>MSR (immediate) on page C6-1708</td>
</tr>
</tbody>
</table>

### Instructions with register argument

For detailed information about instructions with register argument, see Chapter C6 A64 Base Instruction Descriptions. Table C3-8 on page C3-250 shows the instructions with register argument.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>WFET</td>
<td>Wait for event with Timeout</td>
<td>WFET on page C6-2031</td>
</tr>
<tr>
<td>WFIT</td>
<td>Wait for interrupt with Timeout</td>
<td>WFIT on page C6-2033</td>
</tr>
</tbody>
</table>

### System instructions

For detailed information about the System instructions, see Chapter C5 The A64 System Instruction Class.
Table C3-9 shows the System instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS</td>
<td>System instruction</td>
<td>SYS on page C6-1993</td>
</tr>
<tr>
<td>SYSL</td>
<td>System instruction with result</td>
<td>SYSL on page C6-1995</td>
</tr>
<tr>
<td>IC</td>
<td>Instruction cache maintenance</td>
<td>IC on page C6-1508 and Table C5-1 on page C5-660</td>
</tr>
<tr>
<td>DC</td>
<td>Data cache maintenance</td>
<td>DC on page C6-1479 and Table C5-1 on page C5-660</td>
</tr>
<tr>
<td>AT</td>
<td>Address translation</td>
<td>AT on page C6-1205 and Table C5-3 on page C5-661</td>
</tr>
<tr>
<td>TLBI</td>
<td>TLB Invalidate</td>
<td>TLBI on page C6-2000 and Table C5-4 on page C5-662</td>
</tr>
</tbody>
</table>

Table C3-10 on page C3-251 shows the Hint instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>No operation</td>
<td>NOP on page C6-1726</td>
</tr>
<tr>
<td>YIELD</td>
<td>Yield hint</td>
<td>YIELD on page C6-2037</td>
</tr>
<tr>
<td>WFE</td>
<td>Wait for event</td>
<td>WFE on page C6-2030</td>
</tr>
<tr>
<td>WFI</td>
<td>Wait for interrupt</td>
<td>WFI on page C6-2032</td>
</tr>
<tr>
<td>SEV</td>
<td>Send event</td>
<td>SEV on page C6-1820</td>
</tr>
<tr>
<td>SEVL</td>
<td>Send event local</td>
<td>SEVL on page C6-1821</td>
</tr>
<tr>
<td>HINT</td>
<td>Unallocated hint</td>
<td>HINT on page C6-1504</td>
</tr>
<tr>
<td>DGH</td>
<td>Data Gathering Hint</td>
<td>DGH on page C6-1484</td>
</tr>
</tbody>
</table>

Table C3-11 on page C3-251 shows the barrier and CLREX instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLREX</td>
<td>Clear Exclusives monitor</td>
<td>CLREX on page C6-1266</td>
</tr>
<tr>
<td>DMB</td>
<td>Data memory barrier</td>
<td>DMB on page C6-1485</td>
</tr>
<tr>
<td>DSB</td>
<td>Data synchronization barrier</td>
<td>DSB on page C6-1488</td>
</tr>
<tr>
<td>ISB</td>
<td>Instruction synchronization barrier</td>
<td>ISB on page C6-1511</td>
</tr>
</tbody>
</table>

For more information about DSB, DMB, and ISB, see Memory barriers on page B2-174.
Table C3-12 shows the speculation and synchronization barriers. If these instructions are not implemented, then these instructions execute as a NOP.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSDB</td>
<td>Consumption of Speculative Data Barrier</td>
<td>CSDB on page C6-1466</td>
</tr>
<tr>
<td>ESB</td>
<td>Error synchronization barrier</td>
<td>ESB on page C6-1500</td>
</tr>
<tr>
<td>PSB CSYNC</td>
<td>Profiling synchronization barrier</td>
<td>PSB CSYNC on page C6-1750</td>
</tr>
<tr>
<td>PSSBB</td>
<td>Physical Speculative Store Bypass Barrier</td>
<td>PSSBB on page C6-1751</td>
</tr>
<tr>
<td>SB</td>
<td>Speculation Barrier</td>
<td>SB on page C6-1770</td>
</tr>
<tr>
<td>SSBB</td>
<td>Speculative Store Bypass Barrier</td>
<td>SSBB on page C6-1834</td>
</tr>
<tr>
<td>TSB CSYNC</td>
<td>Trace Synchronization Barrier</td>
<td>TSB CSYNC on page C6-2008</td>
</tr>
</tbody>
</table>

For more information about:
- CSDB, PSSBB, SB, SSBB, TSB CSYNC, see Memory barriers on page B2-174.
- ESB, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile.
- PSB CSYNC, see Chapter D13 The Statistical Profiling Extension.

C3.1.10 Pointer authentication instructions

FEAT_PAuth adds support for pointer authentication, see Pointer authentication on page D8-5164. This functionality includes the A64 instructions described in this section. These instructions fall into two groups, see:
- Basic pointer authentication instructions.
- Combined instructions that include pointer authentication on page C3-254.

Basic pointer authentication instructions

Each of these instructions only performs an operation that supports pointer authentication.

Table C3-13 on page C3-252 shows the instructions that add a Pointer Authentication Code (PAC) to the address in a register:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>PACIAZ</td>
<td>Add PAC to instruction address using APIAKey_EL1 and zero</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA on page C6-1736</td>
</tr>
<tr>
<td>PACIASP</td>
<td>Add PAC to instruction address using APIAKey_EL1 and SP</td>
<td></td>
</tr>
<tr>
<td>PACIA1716</td>
<td>Add PAC to instruction address X17 using APIAKey_EL1 and X16</td>
<td></td>
</tr>
</tbody>
</table>
Table C3-13 Instructions that add a PAC (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>PACIBSP</td>
<td>Add PAC to instruction address using APIBKey_EL1 and SP</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB on page C6-1739</td>
</tr>
<tr>
<td>PACIBZ</td>
<td>Add PAC to instruction address using APIBKey_EL1 and zero</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB on page C6-1739</td>
</tr>
<tr>
<td>PACIB1716</td>
<td>Add PAC to instruction address X17 using APIBKey_EL1 and X16</td>
<td></td>
</tr>
<tr>
<td>PACIA</td>
<td>Add PAC to instruction address using APIAKey_EL1, registers</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA on page C6-1736</td>
</tr>
<tr>
<td>PACDA</td>
<td>Add PAC to data address using APDAKey_EL1, registers</td>
<td>PACDA, PACDZA on page C6-1733</td>
</tr>
<tr>
<td>PACIB</td>
<td>Add PAC to instruction address using APIBKey_EL1, registers</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB on page C6-1739</td>
</tr>
<tr>
<td>PACDB</td>
<td>Add PAC to data address using APDBKey_EL1, registers</td>
<td>PACDB, PACDZB on page C6-1734</td>
</tr>
<tr>
<td>PACIZA</td>
<td>Add PAC to instruction address using APIAKey_EL1, register and zero</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA on page C6-1736</td>
</tr>
<tr>
<td>PACDZA</td>
<td>Add PAC to data address using APDAKey_EL1, register and zero</td>
<td>PACDA, PACDZA on page C6-1733</td>
</tr>
<tr>
<td>PACIZB</td>
<td>Add PAC to instruction address using APIBKey_EL1, register and zero</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB on page C6-1739</td>
</tr>
<tr>
<td>PACDZB</td>
<td>Add PAC to data address using APDBKey_EL1, register and zero</td>
<td>PACDB, PACDZB on page C6-1734</td>
</tr>
<tr>
<td>PACGA</td>
<td>Add generic PAC using APGAKey_EL1, registers</td>
<td>PACGA on page C6-1735</td>
</tr>
</tbody>
</table>

Table C3-14 on page C3-253 shows the instructions that authenticate a PAC in a register:

Table C3-14 Instructions that authenticate a PAC

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTIASP</td>
<td>Authenticate PAC for instruction address using APIAKey_EL1 and SP</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA on page C6-1209</td>
</tr>
<tr>
<td>AUTIAZ</td>
<td>Authenticate PAC for instruction address using APIAKey_EL1 and zero</td>
<td></td>
</tr>
<tr>
<td>AUTIA1716</td>
<td>Authenticate PAC for instruction address X17 using APIAKey_EL1 and X16</td>
<td></td>
</tr>
<tr>
<td>AUTIBSP</td>
<td>Authenticate PAC for instruction address using APIBKey_EL1 and SP</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB on page C6-1211</td>
</tr>
<tr>
<td>AUTIBZ</td>
<td>Authenticate PAC for instruction address using APIBKey_EL1 and zero</td>
<td></td>
</tr>
<tr>
<td>AUTIB1716</td>
<td>Authenticate PAC for instruction address X17 using APIBKey_EL1 and X16</td>
<td></td>
</tr>
</tbody>
</table>
Table C3-14 Instructions that authenticate a PAC (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTIA</td>
<td>Authenticate PAC for instruction address using APIAKey_EL1, registers</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA on page C6-1209</td>
</tr>
<tr>
<td>AUTDA</td>
<td>Authenticate PAC for data address using APDAKey_EL1, registers</td>
<td>AUTDA, AUTDZA on page C6-1207</td>
</tr>
<tr>
<td>AUTIB</td>
<td>Authenticate PAC for instruction address using APIBKey_EL1, registers</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB on page C6-1211</td>
</tr>
<tr>
<td>AUTDB</td>
<td>Authenticate PAC for data address using APDBKey_EL1, registers</td>
<td>AUTDB, AUTDZB on page C6-1208</td>
</tr>
<tr>
<td>AUTIZA</td>
<td>Authenticate PAC for instruction address using APIAKey_EL1, register and zero</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA on page C6-1209</td>
</tr>
<tr>
<td>AUTDZA</td>
<td>Authenticate PAC for data address using APDAKey_EL1, register and zero</td>
<td>AUTDA, AUTDZA on page C6-1207</td>
</tr>
<tr>
<td>AUTIZB</td>
<td>Authenticate PAC for instruction address using APIBKey_EL1, register and zero</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB on page C6-1211</td>
</tr>
<tr>
<td>AUTDZB</td>
<td>Authenticate PAC for data address using APDBKey_EL1, register and zero</td>
<td>AUTDB, AUTDZB on page C6-1208</td>
</tr>
</tbody>
</table>

Table C3-15 on page C3-254 shows the instructions that strip a PAC from a register, without performing any authentication:

Table C3-15 Instructions that strip a PAC

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPACLRI</td>
<td>Strip instruction address PAC from LR</td>
<td>XPACD, XPACI, XPACLRI on page C6-2035</td>
</tr>
<tr>
<td>XPACI</td>
<td>Strip instruction address PAC, register</td>
<td></td>
</tr>
<tr>
<td>XPACD</td>
<td>Strip data address PAC, register</td>
<td></td>
</tr>
</tbody>
</table>

Combined instructions that include pointer authentication

Each of these instructions combines a pointer authentication with another operation that uses the authenticated pointer. Table C3-16 on page C3-254 shows these instructions:

Table C3-16 Combined pointer authentication instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETAA</td>
<td>Authenticate PAC for LR using APIAKey_EL1 and SP, and return</td>
<td>RETAA, RETAB on page C6-1755</td>
</tr>
<tr>
<td>RETAB</td>
<td>Authenticate PAC for LR using APIBKey_EL1 and SP, and return</td>
<td></td>
</tr>
<tr>
<td>BRAAA</td>
<td>Authenticate PAC using APIAKey_EL1 (registers), and branch</td>
<td>BRAA, BRAAZ, BRAB, BRABZ on page C6-1234</td>
</tr>
<tr>
<td>BRAB</td>
<td>Authenticate PAC using APIBKey_EL1 (registers), and branch</td>
<td></td>
</tr>
</tbody>
</table>
### Table C3-16 Combined pointer authentication instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLRAA</td>
<td>Authenticate PAC using APIAKey_EL1 (registers), and branch with link</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ on page C6-1231</td>
</tr>
<tr>
<td>BLRAB</td>
<td>Authenticate PAC using APIBKey_EL1 (registers), and branch with link</td>
<td>BRAA, BRAAZ, BRAB, BRABZ on page C6-1234</td>
</tr>
<tr>
<td>BRAAZ</td>
<td>Authenticate PAC using APIAKey_EL1 (register and zero), and branch</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ on page C6-1231</td>
</tr>
<tr>
<td>BRABZ</td>
<td>Authenticate PAC using APIBKey_EL1 (register and zero), and branch</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ on page C6-1231</td>
</tr>
<tr>
<td>BLRAAZ</td>
<td>Authenticate PAC using APIAKey_EL1 (register and zero), and branch with link</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ on page C6-1231</td>
</tr>
<tr>
<td>BLRABZ</td>
<td>Authenticate PAC using APIBKey_EL1 (register and zero), and branch with link</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ on page C6-1231</td>
</tr>
<tr>
<td>ERETAZ</td>
<td>Authenticate PAC for ELR using APIAKey_EL1 and SP, and exception return</td>
<td>ERETAZ on page C6-1499</td>
</tr>
<tr>
<td>ERETAB</td>
<td>Authenticate PAC for ELR using APIBKey_EL1 and SP, and exception return</td>
<td>ERETAZ on page C6-1499</td>
</tr>
<tr>
<td>LDRAA</td>
<td>Authenticate PAC for data address using APDAKey_EL1 (register and zero) and Load</td>
<td>LDRAA, LDRAZ on page C6-1584</td>
</tr>
<tr>
<td>LDRAZ</td>
<td>Authenticate PAC for data address using APDBKey_EL1 (register and zero) and Load</td>
<td>LDRAA, LDRAZ on page C6-1584</td>
</tr>
</tbody>
</table>
C3.2 Loads and stores

This section describes the load/store instructions. It contains the following subsections:

- Load/store register.
- Load/store register (unscaled offset) on page C3-257.
- Load/store pair on page C3-258.
- Load/store non-temporal pair on page C3-259.
- Load/store unprivileged on page C3-260.
- Load-Exclusive/Store-Exclusive on page C3-260.
- LoadLOAcquire/StoreLORelease on page C3-263.
- Load/store scalar SIMD and floating-point on page C3-263.
- Load/store Advanced SIMD on page C3-265.
- Prefetch memory on page C3-267.
- Atomic instructions on page C3-268.
- Memory Tagging instructions on page C3-273.
- Memory Copy and Memory Set instructions on page C3-274.

The requirements for the alignment of data memory accesses are strict. For more information, see Alignment of data accesses on page B2-189.

The additional control bits SCTLR_ELx.SA and SCTLR_EL1.SA0 control whether the stack pointer must be quadword aligned when used as a base register. See SP alignment checking on page D1-4668. Using a misaligned stack pointer generates an SP alignment fault exception.

For information about the encoding structure of the instructions in this instruction group, see Loads and Stores on page C4-547.

--- Note ---

In some cases, load/store instructions can lead to CONSTRAINED UNPREDICTABLE behavior. See AArch64 CONSTRAINED UNPREDICTABLE behaviors on page K1-11584.

C3.2.1 Load/store register

The load/store register instructions support the following addressing modes:

- Base plus a scaled 12-bit unsigned immediate offset or base plus an unscaled 9-bit signed immediate offset.
- Base plus a 64-bit register offset, optionally scaled.
- Base plus a 32-bit extended register offset, optionally scaled.
- Pre-indexed by an unscaled 9-bit signed immediate offset.
- Post-indexed by an unscaled 9-bit signed immediate offset.
- PC-relative literal for loads of 32 bits or more.

See also Load/store addressing modes on page C1-234.

If a Load instruction specifies writeback and the register being loaded is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs the load using the specified addressing mode and the base register becomes UNKNOWN. In addition, if an exception occurs during the execution of such an instruction, the base address might be corrupted so that the instruction cannot be repeated.
If a Store instruction performs a writeback and the register that is stored is also the base register, then behavior is **CONSTRAINED UNPREDICTABLE** and one of the following behaviors must occur:

- The instruction is treated as **UNDEFINED**.
- The instruction is treated as a **NOP**.
- The instruction performs the store to the designated register using the specified addressing mode, but the value stored is **UNKNOWN**.

Table C3-17 shows the load/store register instructions.

**Table C3-17 Load/store register instructions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>Load register (register offset)</td>
<td>LDR (register) on page C6-1582</td>
</tr>
<tr>
<td></td>
<td>Load register (immediate offset)</td>
<td>LDR (immediate) on page C6-1577</td>
</tr>
<tr>
<td></td>
<td>Load register (PC-relative literal)</td>
<td>LDR (literal) on page C6-1580</td>
</tr>
<tr>
<td>LDRB</td>
<td>Load byte (register offset)</td>
<td>LDRB (register) on page C6-1589</td>
</tr>
<tr>
<td></td>
<td>Load byte (immediate offset)</td>
<td>LDRB (immediate) on page C6-1586</td>
</tr>
<tr>
<td>LDRSB</td>
<td>Load signed byte (register offset)</td>
<td>LDRSB (register) on page C6-1600</td>
</tr>
<tr>
<td></td>
<td>Load signed byte (immediate offset)</td>
<td>LDRSB (immediate) on page C6-1596</td>
</tr>
<tr>
<td>LDRH</td>
<td>Load halfword (register offset)</td>
<td>LDRH (register) on page C6-1594</td>
</tr>
<tr>
<td></td>
<td>Load halfword (immediate offset)</td>
<td>LDRH (immediate) on page C6-1591</td>
</tr>
<tr>
<td>LDRSH</td>
<td>Load signed halfword (register offset)</td>
<td>LDRSH (register) on page C6-1606</td>
</tr>
<tr>
<td></td>
<td>Load signed halfword (immediate offset)</td>
<td>LDRSH (immediate) on page C6-1602</td>
</tr>
<tr>
<td>LDRSW</td>
<td>Load signed word (register offset)</td>
<td>LDRSW (register) on page C6-1612</td>
</tr>
<tr>
<td></td>
<td>Load signed word (immediate offset)</td>
<td>LDRSW (immediate) on page C6-1608</td>
</tr>
<tr>
<td></td>
<td>Load signed word (PC-relative literal)</td>
<td>LDRSW (literal) on page C6-1611</td>
</tr>
<tr>
<td>STR</td>
<td>Store register (register offset)</td>
<td>STR (register) on page C6-1898</td>
</tr>
<tr>
<td></td>
<td>Store register (immediate offset)</td>
<td>STR (immediate) on page C6-1895</td>
</tr>
<tr>
<td>STRB</td>
<td>Store byte (register offset)</td>
<td>STRB (register) on page C6-1903</td>
</tr>
<tr>
<td></td>
<td>Store byte (immediate offset)</td>
<td>STRB (immediate) on page C6-1900</td>
</tr>
<tr>
<td>STRH</td>
<td>Store halfword (register offset)</td>
<td>STRH (register) on page C6-1908</td>
</tr>
<tr>
<td></td>
<td>Store halfword (immediate offset)</td>
<td>STRH (immediate) on page C6-1905</td>
</tr>
</tbody>
</table>

### C3.2.2 Load/store register (unscaled offset)

The load/store register instructions with an unscaled offset support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See *Load/store addressing modes* on page C1-234.
The load/store register (unscaled offset) instructions are required to disambiguate this instruction class from the load/store register instruction forms that support an addressing mode of base plus a scaled, unsigned 12-bit immediate offset, because that can represent some offset values in the same range.

The ambiguous immediate offsets are byte offsets that are both:
- In the range 0-255, inclusive.
- Naturally aligned to the access size.

Other byte offsets in the range -256 to 255 inclusive are unambiguous. An assembler program translating a load/store instruction, for example LDR, is required to encode an unambiguous offset using the unscaled 9-bit offset form, and to encode an ambiguous offset using the scaled 12-bit offset form. A programmer might force the generation of the unscaled 9-bit form by using one of the mnemonics in Table C3-18. Arm recommends that a disassembler outputs all unscaled 9-bit offset forms using one of these mnemonics, but unambiguous offsets can be output using a load/store single register mnemonic, for example, LDR.

Table C3-18 shows the load/store register instructions with an unscaled offset.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDUR</td>
<td>Load register (unscaled offset)</td>
<td>LDUR on page C6-1661</td>
</tr>
<tr>
<td>LDURB</td>
<td>Load byte (unscaled offset)</td>
<td>LDURB on page C6-1663</td>
</tr>
<tr>
<td>LDURSB</td>
<td>Load signed byte (unscaled offset)</td>
<td>LDURSB on page C6-1667</td>
</tr>
<tr>
<td>LDURH</td>
<td>Load halfword (unscaled offset)</td>
<td>LDURH on page C6-1665</td>
</tr>
<tr>
<td>LDURSH</td>
<td>Load signed halfword (unscaled offset)</td>
<td>LDURSH on page C6-1667</td>
</tr>
<tr>
<td>LDURSW</td>
<td>Load signed word (unscaled offset)</td>
<td>LDURSW on page C6-1669</td>
</tr>
<tr>
<td>STUR</td>
<td>Store register (unscaled offset)</td>
<td>STUR on page C6-1946</td>
</tr>
<tr>
<td>STURB</td>
<td>Store byte (unscaled offset)</td>
<td>STURB on page C6-1948</td>
</tr>
<tr>
<td>STURH</td>
<td>Store halfword (unscaled offset)</td>
<td>STURH on page C6-1948</td>
</tr>
</tbody>
</table>

C3.2.3 Load/store pair

The load/store pair instructions support the following addressing modes:
- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See also Load/store addressing modes on page C1-234.

If a Load Pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:
- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

If a Load Pair instruction specifies writeback and one of the registers being loaded is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:
- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
The instruction performs all of the loads using the specified addressing mode, and the base register becomes UNKNOWN. In addition, if an exception occurs during the instruction, the base address might be corrupted so that the instruction cannot be repeated.

If a Store Pair instruction performs a writeback and one of the registers being stored is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the stores of the registers indicated by the specified addressing mode, but the value stored for the base register is UNKNOWN.

Table C3-19 shows the load/store pair instructions.

### Table C3-19 Load/store pair instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDP</td>
<td>Load Pair</td>
<td>LDP on page C6-1570</td>
</tr>
<tr>
<td>LDPSW</td>
<td>Load Pair signed words</td>
<td>LDPSW on page C6-1574</td>
</tr>
<tr>
<td>STP</td>
<td>Store Pair</td>
<td>STP on page C6-1892</td>
</tr>
</tbody>
</table>

### C3.2.4 Load/store non-temporal pair

The load/store non-temporal pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

See Load/store addressing modes on page C1-234.

The load/store non-temporal pair instructions provide a hint to the memory system that an access is non-temporal or streaming, and unlikely to be repeated in the near future. This means that data caching is not required. However, depending on the memory type, the instructions might permit memory reads to be preloaded and memory writes to be gathered to accelerate bulk memory transfers.

In addition, there is an exception to the usual memory ordering rules. If an address dependency exists between two memory reads, and a Load Non-temporal Pair instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, the memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

If a Load Non-Temporal Pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

Table C3-20 on page C3-259 shows the load/store non-temporal pair instructions.

### Table C3-20 Load/store non-temporal pair instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDNP</td>
<td>Load Non-temporal Pair</td>
<td>LDNP on page C6-1568</td>
</tr>
<tr>
<td>STNP</td>
<td>Store Non-temporal Pair</td>
<td>STNP on page C6-1890</td>
</tr>
</tbody>
</table>
C3.2.5  **Load/store unprivileged**

The load/store unprivileged instructions support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See *Load/store addressing modes* on page C1-234.

The access permissions that apply to accesses made at EL0 apply to the memory accesses made by a load/store unprivileged instruction that is executed either:

- At EL1 when the *Effective value* of PSTATE.UAO is 0.
- At EL2 when both the *Effective value* of HCR_EL2.{E2H, TGE} is \{1, 1\} and the *Effective value* of PSTATE.UAO is 0.

Otherwise, memory accesses made by a load/store unprivileged instruction are subject to the access permissions that apply to the Exception level at which the instruction is executed. These are the permissions that apply to the corresponding load/store register instruction, see *Load/store register* on page C3-256.

--- Note ---

This means that when the value of PSTATE.UAO is 1 the access permissions for a load/store unprivileged instruction are always the same as those for the corresponding load/store register instruction.

---

Table C3-21 on page C3-260 shows the load/store unprivileged instructions.

**Table C3-21 Load-Store unprivileged instructions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDTR</td>
<td>Load unprivileged register</td>
<td><em>LDTR</em> on page C6-1635</td>
</tr>
<tr>
<td>LDTRB</td>
<td>Load unprivileged byte</td>
<td><em>LDTRB</em> on page C6-1637</td>
</tr>
<tr>
<td>LDTRSB</td>
<td>Load unprivileged signed byte</td>
<td><em>LDTRSB</em> on page C6-1641</td>
</tr>
<tr>
<td>LDTRH</td>
<td>Load unprivileged halfword</td>
<td><em>LDTRH</em> on page C6-1639</td>
</tr>
<tr>
<td>LDTRSH</td>
<td>Load unprivileged signed halfword</td>
<td><em>LDTRSH</em> on page C6-1643</td>
</tr>
<tr>
<td>LDTRSW</td>
<td>Load unprivileged signed word</td>
<td><em>LDTRSW</em> on page C6-1645</td>
</tr>
<tr>
<td>STTR</td>
<td>Store unprivileged register</td>
<td><em>STTR</em> on page C6-1928</td>
</tr>
<tr>
<td>STTRB</td>
<td>Store unprivileged byte</td>
<td><em>STTRB</em> on page C6-1930</td>
</tr>
<tr>
<td>STTRH</td>
<td>Store unprivileged halfword</td>
<td><em>STTRH</em> on page C6-1932</td>
</tr>
</tbody>
</table>

C3.2.6  **Load-Exclusive/Store-Exclusive**

The Load-Exclusive/Store-Exclusive instructions support only one addressing mode:

- Base register with no offset.

See *Load/store addressing modes* on page C1-234.

The Load-Exclusive instructions mark the physical address being accessed as an exclusive access. This exclusive access mark is checked by the Store-Exclusive instruction, permitting the construction of atomic read-modify-write operations on shared memory variables, semaphores, mutexes, and spinlocks. See *Synchronization and semaphores* on page B2-211.
If FEAT_LSE2 is not implemented then:

- The Load-Exclusive/Store-Exclusive instructions other than Load-Exclusive pair and Store-Exclusive pair require natural alignment, and an unaligned address generates an Alignment fault.
- Memory accesses generated by Load-Exclusive pair or Store-Exclusive pair instructions must be aligned to the size of the pair, otherwise the access generates an Alignment fault.

For more information on alignment requirements and behaviors, see *Load-Exclusive/ Store-Exclusive and Atomic instructions* on page B2-190.

When a Store-Exclusive pair succeeds, it causes a single-copy atomic update of the entire memory location being stored to.

Table C3-22 shows the Load-Exclusive/Store-Exclusive instructions.

### Table C3-22 Load-Exclusive/Store-Exclusive instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDXR</td>
<td>Load Exclusive register</td>
<td>LDXR on page C6-1672</td>
</tr>
<tr>
<td>LDXRB</td>
<td>Load Exclusive byte</td>
<td>LDXRB on page C6-1674</td>
</tr>
<tr>
<td>LDXRH</td>
<td>Load Exclusive halfword</td>
<td>LDXRH on page C6-1675</td>
</tr>
<tr>
<td>LDXP</td>
<td>Load Exclusive pair</td>
<td>LDXP on page C6-1670</td>
</tr>
<tr>
<td>STXR</td>
<td>Store Exclusive register</td>
<td>STXR on page C6-1953</td>
</tr>
<tr>
<td>STXRB</td>
<td>Store Exclusive byte</td>
<td>STXRB on page C6-1955</td>
</tr>
<tr>
<td>STXRH</td>
<td>Store Exclusive halfword</td>
<td>STXRH on page C6-1957</td>
</tr>
<tr>
<td>STXP</td>
<td>Store Exclusive pair</td>
<td>STXP on page C6-1950</td>
</tr>
</tbody>
</table>

#### C3.2.7 Load-Acquire/Store-Release

The Load-Acquire, Load-AcquirePC, and Store-Release instructions support only one addressing mode:

- Base register with no offset.

See *Load/store addressing modes* on page C1-234.


The Load-Acquire, Load-AcquirePC, and Store-Release instructions other than Load-Acquire pair and Store-Release pair require natural alignment, and an unaligned address generates an Alignment fault. Memory accesses generated by Load-Acquire pair or Store-Release pair instructions must be aligned to the size of the pair, otherwise the access generates an Alignment fault.

A Store-Release Exclusive instruction has the Release semantics only if the store is successful.

Armv8.1 adds more instructions with load-acquire and store-release mechanisms, see *LoadLOAcquire/StoreLORelease* on page C3-263.

FEAT_LRCPC2 introduces changes to the alignment requirements of Load-Acquire/Store-Release instructions.
Table C3-23 on page C3-262 shows the Non-exclusive Load-Acquire/Store-Release instructions.

### Table C3-23 Non-exclusive Load-Acquire and Store-Release instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAPR</td>
<td>Load-Acquire RCpc Register</td>
<td>LDAPR on page C6-1520</td>
</tr>
<tr>
<td>LDAPRB</td>
<td>Load-Acquire RCpc Register Byte</td>
<td>LDAPRB on page C6-1522</td>
</tr>
<tr>
<td>LDAPRH</td>
<td>Load-Acquire RCpc Register Halfword</td>
<td>LDAPRH on page C6-1524</td>
</tr>
<tr>
<td>LDAPUR</td>
<td>Load-Acquire RCpc Register (unscaled)</td>
<td>LDAPUR on page C6-1526</td>
</tr>
<tr>
<td>LDAPURB</td>
<td>Load-Acquire RCpc Register Byte (unscaled)</td>
<td>LDAPURB on page C6-1528</td>
</tr>
<tr>
<td>LDAPURH</td>
<td>Load-Acquire RCpc Register Halfword (unscaled)</td>
<td>LDAPURH on page C6-1530</td>
</tr>
<tr>
<td>LDAPURSB</td>
<td>Load-Acquire RCpc Register Signed Byte (unscaled) 32-bit</td>
<td>LDAPURSB on page C6-1532</td>
</tr>
<tr>
<td>LDAPURSB</td>
<td>Load-Acquire RCpc Register Signed Byte (unscaled) 64-bit</td>
<td>LDAPURSB on page C6-1532</td>
</tr>
<tr>
<td>LDAPURSH</td>
<td>Load-Acquire RCpc Register Signed Halfword (unscaled) 32-bit</td>
<td>LDAPURSH on page C6-1534</td>
</tr>
<tr>
<td>LDAPURSH</td>
<td>Load-Acquire RCpc Register Signed Halfword (unscaled) 64-bit</td>
<td>LDAPURSH on page C6-1534</td>
</tr>
<tr>
<td>LDAPURSW</td>
<td>Load-Acquire RCpc Register Signed Word (unscaled)</td>
<td>LDAPURSW on page C6-1536</td>
</tr>
<tr>
<td>LDAR</td>
<td>Load-Acquire Register</td>
<td>LDAR on page C6-1538</td>
</tr>
<tr>
<td>LDARB</td>
<td>Load-Acquire Byte</td>
<td>LDARB on page C6-1540</td>
</tr>
<tr>
<td>LDARH</td>
<td>Load-Acquire Halfword</td>
<td>LDARH on page C6-1541</td>
</tr>
<tr>
<td>STLRR</td>
<td>Store-Release Register</td>
<td>STLRR on page C6-1870</td>
</tr>
<tr>
<td>STLRB</td>
<td>Store-Release Byte</td>
<td>STLRB on page C6-1872</td>
</tr>
<tr>
<td>STLRH</td>
<td>Store-Release Halfword</td>
<td>STLRH on page C6-1873</td>
</tr>
<tr>
<td>STLUR</td>
<td>Store-Release Register (unscaled)</td>
<td>STLUR on page C6-1874</td>
</tr>
<tr>
<td>STLURB</td>
<td>Store-Release Register Byte (unscaled)</td>
<td>STLURB on page C6-1876</td>
</tr>
<tr>
<td>STLURH</td>
<td>Store-Release Register Halfword (unscaled)</td>
<td>STLURH on page C6-1878</td>
</tr>
</tbody>
</table>

Table C3-24 on page C3-262 shows the Exclusive Load-Acquire/Store-Release instructions.

### Table C3-24 Exclusive Load-Acquire and Store-Release instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAPXR</td>
<td>Load-Acquire Exclusive register</td>
<td>LDAPXR on page C6-1544</td>
</tr>
<tr>
<td>LDAPXRB</td>
<td>Load-Acquire Exclusive byte</td>
<td>LDAPXRB on page C6-1546</td>
</tr>
<tr>
<td>LDAPXRH</td>
<td>Load-Acquire Exclusive halfword</td>
<td>LDAPXRH on page C6-1547</td>
</tr>
<tr>
<td>LDAXP</td>
<td>Load-Acquire Exclusive pair</td>
<td>LDAXP on page C6-1542</td>
</tr>
<tr>
<td>STLXR</td>
<td>Store-Release Exclusive register</td>
<td>STLXR on page C6-1883</td>
</tr>
</tbody>
</table>
C3.2.8 LoadLOAcquire/StoreLORelease

The LoadLOAcquire/StoreLORelease instructions support only one addressing mode:

- Base register with no offset.

See *Load/store addressing modes* on page C1-234.

The LoadLOAcquire/StoreLORelease instructions can remove the requirement to use the explicit DMB memory barrier instruction. For more information about the ordering of LoadLOAcquire/StoreLORelease, see *LoadLOAcquire, StoreLORelease* on page B2-181.

The LoadLOAcquire/StoreLORelease instructions require natural alignment, and an unaligned address generates an Alignment fault.

Table C3-25 on page C3-263 shows the LoadLOAcquire/StoreLORelease instructions.

### Table C3-25 LoadLOAcquire and StoreLORelease instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDLARB</td>
<td>LoadLOAcquire byte</td>
<td>LDLARB on page C6-1564</td>
</tr>
<tr>
<td>LDLARH</td>
<td>LoadLOAcquire halfword</td>
<td>LDLARH on page C6-1565</td>
</tr>
<tr>
<td>LDLAR</td>
<td>LoadLOAcquire register</td>
<td>LDLAR on page C6-1566</td>
</tr>
<tr>
<td>STLLRB</td>
<td>StoreLORelease byte</td>
<td>STLLRB on page C6-1866</td>
</tr>
<tr>
<td>STLLRH</td>
<td>StoreLORelease halfword</td>
<td>STLLRH on page C6-1867</td>
</tr>
<tr>
<td>STLLR</td>
<td>StoreLORelease register</td>
<td>STLLR on page C6-1868</td>
</tr>
</tbody>
</table>

C3.2.9 Load/store scalar SIMD and floating-point

The load/store scalar SIMD and floating-point instructions operate on scalar values in the SIMD and floating-point register file as described in *SIMD and floating-point scalar register names* on page C1-232. The memory addressing modes available, described in *Load/store addressing modes* on page C1-234, are identical to the general-purpose register load/store instructions, and like those instructions permit arbitrary address alignment unless strict alignment checking is enabled. However, unlike the load/store instructions that transfer general-purpose registers, load/store scalar SIMD and floating-point instructions make no guarantee of atomicity, even when the address is naturally aligned to the size of the data.

**Load/store scalar SIMD and floating-point register**

The load/store scalar SIMD and floating-point register instructions support the following addressing modes:

- Base plus a scaled 12-bit unsigned immediate offset or base plus unscaled 9-bit signed immediate offset.
- Base plus 64-bit register offset, optionally scaled.
- Base plus 32-bit extended register offset, optionally scaled.
- Pre-indexed by an unscaled 9-bit signed immediate offset.
- Post-indexed by an unscaled 9-bit signed immediate offset.
PC-relative literal for loads of 32 bits or more.

For more information on the addressing modes, see Load/store addressing modes on page C1-234.

**Note**

The unscaled 9-bit signed immediate offset address mode requires its own instruction form, see Load/store scalar SIMD and floating-point register (unscaled offset).

Table C3-26 on page C3-264 shows the load/store instructions for a single SIMD and floating-point register.

### Table C3-26 Load/store single SIMD and floating-point register instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>Load scalar SIMD&amp;FP register (register offset)</td>
<td>LDR (register, SIMD&amp;FP) on page C7-2492</td>
</tr>
<tr>
<td></td>
<td>Load scalar SIMD&amp;FP register (immediate offset)</td>
<td>LDR (immediate, SIMD&amp;FP) on page C7-2486</td>
</tr>
<tr>
<td></td>
<td>Load scalar SIMD&amp;FP register (PC-relative literal)</td>
<td>LDR (literal, SIMD&amp;FP) on page C7-2490</td>
</tr>
<tr>
<td>STR</td>
<td>Store scalar SIMD&amp;FP register (register offset)</td>
<td>STR (register, SIMD&amp;FP) on page C7-2814</td>
</tr>
<tr>
<td></td>
<td>Store scalar SIMD&amp;FP register (immediate offset)</td>
<td>STR (immediate, SIMD&amp;FP) on page C7-2810</td>
</tr>
</tbody>
</table>

### Load/store scalar SIMD and floating-point register (unscaled offset)

The load/store scalar SIMD and floating-point register instructions support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See also Load/store addressing modes on page C1-234.

The load/store scalar SIMD and floating-point register (unscaled offset) instructions are required to disambiguate this instruction class from the load/store single SIMD and floating-point instruction forms that support an addressing mode of base plus a scaled, unsigned 12-bit immediate offset. This is similar to the load/store register (unscaled offset) instructions, that disambiguate this instruction class from the load/store register instruction, see Load/store register (unscaled offset) on page C3-257.

Table C3-27 on page C3-264 shows the load/store SIMD and floating-point register instructions with an unscaled offset.

### Table C3-27 Load/store SIMD and floating-point register instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDUR</td>
<td>Load scalar SIMD&amp;FP register (unscaled offset)</td>
<td>LDUR (SIMD&amp;FP) on page C7-2495</td>
</tr>
<tr>
<td>STUR</td>
<td>Store scalar SIMD&amp;FP register (unscaled offset)</td>
<td>STUR (SIMD&amp;FP) on page C7-2817</td>
</tr>
</tbody>
</table>

### Load/store SIMD and floating-point register pair

The load/store SIMD and floating-point register pair instructions support the following addressing modes:

- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See also Load/store addressing modes on page C1-234.
If a Load pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all of the loads using the specified addressing mode and the register being loaded takes an UNKNOWN value.

Table C3-28 shows the load/store SIMD and floating-point register pair instructions.

### Table C3-28 Load/store SIMD and floating-point register pair instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDP</td>
<td>Load pair of scalar SIMD&amp;FP registers</td>
<td>LDP (SIMD&amp;FP) on page C7-2482</td>
</tr>
<tr>
<td>STP</td>
<td>Store pair of scalar SIMD&amp;FP registers</td>
<td>STP (SIMD&amp;FP) on page C7-2807</td>
</tr>
</tbody>
</table>

**Load/store SIMD and floating-point non-temporal pair**

The load/store SIMD and floating-point non-temporal pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

See also *Load/store addressing modes* on page C1-234.

The load/store non-temporal pair instructions provide a hint to the memory system that an access is non-temporal or streaming, and unlikely to be repeated in the near future. This means that data caching is not required. However, depending on the memory type, the instructions might permit memory reads to be preloaded and memory writes to be gathered to accelerate bulk memory transfers.

In addition, there is an exception to the usual memory ordering rules. If an address dependency exists between two memory reads, and a load non-temporal pair instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, those memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

If a load non-temporal pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all of the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

Table C3-29 shows the load/store SIMD and floating-point Non-temporal pair instructions.

### Table C3-29 Load/store SIMD and floating-point non-temporal pair instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDNP</td>
<td>Load pair of scalar SIMD&amp;FP registers</td>
<td>LDNP (SIMD&amp;FP) on page C7-2480</td>
</tr>
<tr>
<td>STNP</td>
<td>Store pair of scalar SIMD&amp;FP registers</td>
<td>STNP (SIMD&amp;FP) on page C7-2805</td>
</tr>
</tbody>
</table>

**C3.2.10 Load/store Advanced SIMD**

The Advanced SIMD load/store structure instructions support the following addressing modes:

- Base register only.
- Post-indexed by a 64-bit register.
• Post-indexed by an immediate, equal to the number of bytes transferred.

Load/store vector instructions, like other load/store instructions, allow any address alignment, unless strict alignment checking is enabled. If strict alignment checking is enabled, then alignment checking to the size of the element is performed. However, unlike the load/store instructions that transfer general-purpose registers, the load/store vector instructions do not guarantee atomicity, even when the address is naturally aligned to the size of the element.

**Load/store structures**

Table C3-30 on page C3-266 shows the load/store structure instructions. A post-increment immediate offset, if present, must be 8, 16, 24, 32, 48, or 64, depending on the number of elements transferred.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD1</td>
<td>Load single 1-element structure to one lane of one register</td>
<td>LD1 (single structure) on page C7-2443 on page C7-2439</td>
</tr>
<tr>
<td></td>
<td>Load multiple 1-element structures to one register or to two, three, or four consecutive registers</td>
<td>LD1 (multiple structures) on page C7-2439</td>
</tr>
<tr>
<td>LD2</td>
<td>Load single 2-element structure to one lane of two consecutive registers</td>
<td>LD2 (single structure) on page C7-2453</td>
</tr>
<tr>
<td></td>
<td>Load multiple 2-element structures to two consecutive registers</td>
<td>LD2 (multiple structures) on page C7-2450</td>
</tr>
<tr>
<td>LD3</td>
<td>Load single 3-element structure to one lane of three consecutive registers</td>
<td>LD3 (single structure) on page C7-2463</td>
</tr>
<tr>
<td></td>
<td>Load multiple 3-element structures to three consecutive registers</td>
<td>LD3 (multiple structures) on page C7-2460</td>
</tr>
<tr>
<td>LD4</td>
<td>Load single 4-element structure to one lane of four consecutive registers</td>
<td>LD4 (single structure) on page C7-2473</td>
</tr>
<tr>
<td></td>
<td>Load multiple 4-element structures to four consecutive registers</td>
<td>LD4 (multiple structures) on page C7-2470</td>
</tr>
<tr>
<td>ST1</td>
<td>Store single 1-element structure from one lane of one register</td>
<td>ST1 (single structure) on page C7-2780</td>
</tr>
<tr>
<td></td>
<td>Store multiple 1-element structures from one register, or from two, three, or four consecutive registers</td>
<td>ST1 (multiple structures) on page C7-2776</td>
</tr>
<tr>
<td>ST2</td>
<td>Store single 2-element structure from one lane of two consecutive registers</td>
<td>ST2 (single structure) on page C7-2787</td>
</tr>
<tr>
<td></td>
<td>Store multiple 2-element structures from two consecutive registers</td>
<td>ST2 (multiple structures) on page C7-2784</td>
</tr>
</tbody>
</table>
Load single structure and replicate

Table C3-31 on page C3-267 shows the Load single structure and replicate instructions. A post-increment immediate offset, if present, must be 1, 2, 3, 4, 6, 8, 12, 16, 24, or 32, depending on the number of elements transferred.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD1R</td>
<td>Load single 1-element structure and replicate to all lanes of one register</td>
<td>LD1R on page C7-2447</td>
</tr>
<tr>
<td>LD2R</td>
<td>Load single 2-element structure and replicate to all lanes of two registers</td>
<td>LD2R on page C7-2457</td>
</tr>
<tr>
<td>LD3R</td>
<td>Load single 3-element structure and replicate to all lanes of three registers</td>
<td>LD3R on page C7-2467</td>
</tr>
<tr>
<td>LD4R</td>
<td>Load single 4-element structure and replicate to all lanes of four registers</td>
<td>LD4R on page C7-2477</td>
</tr>
</tbody>
</table>

C3.2.11 Prefetch memory

The Prefetch memory instructions support the following addressing modes:

- Base plus a scaled 12-bit unsigned immediate offset or base plus an unscaled 9-bit signed immediate offset.
- Base plus a 64-bit register offset. This can be optionally scaled by 8-bits, for example LSL#3.
- Base plus a 32-bit extended register offset. This can be optionally scaled by 8-bits.
- PC-relative literal.

The prefetch memory instructions signal to the memory system that memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory access when they do occur, such as preloading the specified address into one or more caches. Because these signals are only hints, it is valid for the PE to treat any or all prefetch instructions as a NOP.

Because they are hints to the memory system, the operation of a PRFM instruction cannot cause a synchronous exception. However, a memory operation performed as a result of one of these memory system hints might in exceptional cases trigger an asynchronous event, and thereby influence the execution of the PE. An example of an asynchronous event that might be triggered is an SError interrupt.

A PRFM instruction can only have an effect on software visible structures, such as caches and translation lookaside buffers associated with memory locations that can be accessed by reads, writes, or execution as defined in the translation regime of the current Exception level.

A PRFM instruction is guaranteed not to access Device memory.

A PRFM instruction using a PLI hint must not result in any access that could not be performed by the PE speculatively fetching an instruction. Therefore, if all associated MMUs are disabled, a PLI hint cannot access any memory location that cannot be accessed by instruction fetches.
The PRFM instructions require an additional <prfop> operand to be specified, which must be one of the following:

- PLDL1KEEP, PLDL1STRM, PLDL2KEEP, PLDL2STRM, PLDL3KEEP, PLDL3STRM
- PSTL1KEEP, PSTL1STRM, PSTL2KEEP, PSTL2STRM, PSTL3KEEP, PSTL3STRM
- PLIL1KEEP, PLIL1STRM, PLIL2KEEP, PLIL2STRM, PLIL3KEEP, PLIL3STRM

<prfop> is defined as <type><target><policy>.

Here:
- <type> is one of:
  - PLD: Prefetch for load.
  - PST: Prefetch for store.
  - PLI: Preload instructions.
- <target> is one of:
  - L1: Level 1 cache.
  - L2: Level 2 cache.
  - L3: Level 3 cache.
- <policy> is one of:
  - KEEP: Retained or temporal prefetch, allocated in the cache normally.
  - STRM: Streaming or non-temporal prefetch, for data that is used only once.

PRFUM explicitly uses the unscaled 9-bit signed immediate offset addressing mode, as described in Load/store register (unscaled offset) on page C3-257.

Table C3-32 shows the Prefetch memory instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRFM</td>
<td>Prefetch memory (register offset)</td>
<td>PRFM (register) on page C6-1746</td>
</tr>
<tr>
<td></td>
<td>Prefetch memory (immediate offset)</td>
<td>PRFM (immediate) on page C6-1742</td>
</tr>
<tr>
<td></td>
<td>Prefetch memory (PC-relative offset)</td>
<td>PRFM (literal) on page C6-1744</td>
</tr>
<tr>
<td>PRFUM</td>
<td>Prefetch memory (unscaled offset)</td>
<td>PRFUM on page C6-1748</td>
</tr>
</tbody>
</table>

### C3.2.12 Atomic instructions

The atomic instructions perform atomic read and write operations on a memory location such that the architecture guarantees that no modification of that memory location by another observer can occur between the read and the write defined by that instruction.

This section describes the following operations:

- Atomic memory operations,
- Single-copy atomic 64-byte load/store on page C3-271.
- Swap on page C3-272.
- Compare and Swap on page C3-272.

#### Atomic memory operations

The atomic memory operation instructions support only one addressing mode:

- Base register only.

See also Load/store addressing modes on page C1-234.
For the purpose of permission checking, and for watchpoints, all of the Atomic memory operation instructions are treated as performing both a load and a store.

If FEAT_LSE2 is not implemented then the LD<OP> and ST<OP> instructions require natural alignment, and an unaligned address generates an Alignment fault. For more information on alignment requirements and behaviors, see Load-Exclusive/Store-Exclusive and Atomic instructions on page B2-190.

The instructions are provided with ordering options, which map to the acquire and release definitions used in the architecture. The atomic instructions with release semantics have the same rules as Store-Release instructions regarding multi-copy atomicity. These operations map to the acquire and release definitions, and are counted as Load-Acquire and Store-Release operations respectively.

For the LD<OP> instructions, where the source and destination registers are the same, if the instruction generates a synchronous Data Abort, then the source register is restored to the value it held before the instruction was executed.

The ST<OP> instructions, and LD<OP> instructions where the destination register is WZR or XZR, are not regarded as doing a read for the purpose of a DMB LD barrier.

Table C3-33 Atomic memory operation instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDADD</td>
<td>Atomic add</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL on page C6-1517</td>
</tr>
<tr>
<td>LDADD8</td>
<td>Atomic add on byte</td>
<td>LDADD8, LDADDA8, LDADDAL8, LDADDL8 on page C6-1513</td>
</tr>
<tr>
<td>LDADDH</td>
<td>Atomic add on halfword</td>
<td>LDADDH, LDADDAH, LDADDALH, LDADDLH on page C6-1515</td>
</tr>
<tr>
<td>LDCLR</td>
<td>Atomic bit clear</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL on page C6-1552</td>
</tr>
<tr>
<td>LDCLR8</td>
<td>Atomic bit clear on byte</td>
<td>LDCLR8, LDCLRA8, LDCLRAL8, LDCLRL8 on page C6-1548</td>
</tr>
<tr>
<td>LDCLRH</td>
<td>Atomic bit clear on halfword</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH on page C6-1550</td>
</tr>
<tr>
<td>LDEOR</td>
<td>Atomic exclusive OR</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL on page C6-1559</td>
</tr>
<tr>
<td>LDEOR8</td>
<td>Atomic exclusive OR on byte</td>
<td>LDEOR8, LDEORA8, LDEORAL8, LDEORL8 on page C6-1555</td>
</tr>
<tr>
<td>LDEORH</td>
<td>Atomic exclusive OR on halfword</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH on page C6-1557</td>
</tr>
<tr>
<td>LDSET</td>
<td>Atomic bit set</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL on page C6-1618</td>
</tr>
<tr>
<td>LDSETB</td>
<td>Atomic bit set on byte</td>
<td>LDSETB, LDSETAB, LDSETALB, LDSETLB on page C6-1614</td>
</tr>
<tr>
<td>LDSETH</td>
<td>Atomic bit set on halfword</td>
<td>LDSETH, LDSETAH, LDSETALH, LDSETLH on page C6-1616</td>
</tr>
<tr>
<td>LDMAX</td>
<td>Atomic signed maximum</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL on page C6-1625</td>
</tr>
<tr>
<td>LDMAXB</td>
<td>Atomic signed maximum on byte</td>
<td>LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB on page C6-1621</td>
</tr>
<tr>
<td>LDMAXH</td>
<td>Atomic signed maximum on halfword</td>
<td>LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH on page C6-1623</td>
</tr>
<tr>
<td>LDMIN</td>
<td>Atomic signed minimum</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL on page C6-1632</td>
</tr>
<tr>
<td>LDMINB</td>
<td>Atomic signed minimum on byte</td>
<td>LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB on page C6-1628</td>
</tr>
<tr>
<td>LDMINH</td>
<td>Atomic signed minimum on halfword</td>
<td>LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH on page C6-1630</td>
</tr>
<tr>
<td>LDUMAX</td>
<td>Atomic unsigned maximum</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL on page C6-1651</td>
</tr>
</tbody>
</table>
### Table C3-33 Atomic memory operation instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDUMAXB</td>
<td>Atomic unsigned maximum on byte</td>
<td>LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB on page C6-1647</td>
</tr>
<tr>
<td>LDUMAXH</td>
<td>Atomic unsigned maximum on halfword</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH on page C6-1649</td>
</tr>
<tr>
<td>LDUMIN</td>
<td>Atomic unsigned minimum</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL on page C6-1658</td>
</tr>
<tr>
<td>LDUMINB</td>
<td>Atomic unsigned minimum on byte</td>
<td>LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB on page C6-1654</td>
</tr>
<tr>
<td>LDUMINH</td>
<td>Atomic unsigned minimum on halfword</td>
<td>LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH on page C6-1656</td>
</tr>
<tr>
<td>STADD</td>
<td>Atomic add, without return</td>
<td>STADD, STADDL on page C6-1846</td>
</tr>
<tr>
<td>STADDB</td>
<td>Atomic add on byte, without return</td>
<td>STADDB, STADDLB on page C6-1842</td>
</tr>
<tr>
<td>STADDH</td>
<td>Atomic add on halfword, without return</td>
<td>STADDH, STADDLH on page C6-1844</td>
</tr>
<tr>
<td>STCLR</td>
<td>Atomic bit clear, without return</td>
<td>STCLR, STCLRL on page C6-1852</td>
</tr>
<tr>
<td>STCLRB</td>
<td>Atomic bit clear on byte, without return</td>
<td>STCLRB, STCLRLB on page C6-1848</td>
</tr>
<tr>
<td>STCLRH</td>
<td>Atomic bit clear on halfword, without return</td>
<td>STCLRH, STCLRLH on page C6-1850</td>
</tr>
<tr>
<td>STEOR</td>
<td>Atomic exclusive OR, without return</td>
<td>STEOR, STEORL on page C6-1858</td>
</tr>
<tr>
<td>STEORB</td>
<td>Atomic exclusive OR on byte, without return</td>
<td>STEORB, STEORLB on page C6-1854</td>
</tr>
<tr>
<td>STEORH</td>
<td>Atomic exclusive OR on halfword, without return</td>
<td>STEORH, STEORLH on page C6-1856</td>
</tr>
<tr>
<td>STSET</td>
<td>Atomic bit set, without return</td>
<td>STSET, STSETL on page C6-1914</td>
</tr>
<tr>
<td>STSETB</td>
<td>Atomic bit set on byte, without return</td>
<td>STSETB, STSETLB on page C6-1910</td>
</tr>
<tr>
<td>STSETH</td>
<td>Atomic bit set on halfword, without return</td>
<td>STSETH, STSETLH on page C6-1912</td>
</tr>
<tr>
<td>STMAX</td>
<td>Atomic signed maximum, without return</td>
<td>STSMAX, STSMAXL on page C6-1920</td>
</tr>
<tr>
<td>STMAXB</td>
<td>Atomic signed maximum on byte, without return</td>
<td>STSMAXB, STSMAXLB on page C6-1916</td>
</tr>
<tr>
<td>STMAXH</td>
<td>Atomic signed maximum on halfword, without return</td>
<td>STSMAXH, STSMAXLH on page C6-1918</td>
</tr>
<tr>
<td>STMIN</td>
<td>Atomic signed minimum, without return</td>
<td>STSMIN, STSMINL on page C6-1926</td>
</tr>
<tr>
<td>STMINB</td>
<td>Atomic signed minimum on byte, without return</td>
<td>STSMINB, STSMINLB on page C6-1922</td>
</tr>
<tr>
<td>STMINH</td>
<td>Atomic signed minimum on halfword, without return</td>
<td>STSMINH, STSMINLH on page C6-1924</td>
</tr>
<tr>
<td>STUMAX</td>
<td>Atomic unsigned maximum, without return</td>
<td>STUMAX, STUMAXL on page C6-1938</td>
</tr>
<tr>
<td>STUMAXB</td>
<td>Atomic unsigned maximum on byte, without return</td>
<td>STUMAXB, STUMAXLB on page C6-1934</td>
</tr>
<tr>
<td>STUMAXH</td>
<td>Atomic unsigned maximum on halfword, without return</td>
<td>STUMAXH, STUMAXLH on page C6-1936</td>
</tr>
</tbody>
</table>
Single-copy atomic 64-byte load/store

If FEAT_LS64 is implemented, the following instructions are implemented:

- LD64B.
- ST64B.

If FEAT_LS64_V is implemented, the following instructions are implemented:

- LD64B.
- ST64B.
- ST64BV.

If FEAT_LS64_ACCDATA is implemented, the following instructions are implemented:

- LD64B.
- ST64B.
- ST64BV.
- ST64BV0.

The single-copy atomic 64-byte load/store instructions support one addressing mode:
- Base register only.

See also Load/store addressing modes on page C1-234.

The memory location accessed by the instructions is required to be aligned on a 64-byte boundary, otherwise an Alignment fault occurs.

When the instructions access a memory type that is not one of the following, a data abort for unsupported Exclusive or atomic access is generated for the stage of translation that provided the memory type:

- Normal Inner Non-cacheable, Outer Non-cacheable.
- Device-GRE.
- Device-nGRE.
- Device-nGnRE.
- Device-nGnRnE.

Regardless of the memory type:
- The memory access generated by an ST64BV or ST64BV0 instruction is not merged with any accesses.
- The memory access generated by an ST64B instruction is not merged with any accesses generated by store instructions appearing in program order after the instruction.

Table C3-34 shows the single-copy atomic 64-byte load/store instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD64B</td>
<td>Single-copy atomic 64-byte load</td>
<td>LD64B on page C6-1512</td>
</tr>
<tr>
<td>ST64B</td>
<td>Single-copy atomic 64-byte store without return</td>
<td>ST64B on page C6-1837</td>
</tr>
</tbody>
</table>
Table C3-35 shows the single-copy atomic 64-byte store with return.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST64BV</td>
<td>Single-copy atomic 64-byte store with return</td>
<td>ST64BV on page C6-1838</td>
</tr>
</tbody>
</table>

Table C3-36 shows the single-copy atomic 64-byte EL0 store with return.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST64BV0</td>
<td>Single-copy atomic 64-byte EL0 store with return</td>
<td>ST64BV0 on page C6-1840</td>
</tr>
</tbody>
</table>

**Swap**

The swap instructions support only one addressing mode:
- Base register only.

See also *Load/store addressing modes on page C1-234.*

For the purpose of permission checking, and for watchpoints, all of the Swap instructions are treated as performing both a load and a store.

If FEAT_LSE2 is not implemented, then the SWP instructions require natural alignment, and an unaligned address generates an Alignment fault. For more information on alignment requirements and behaviors, see *Load-Exclusive/Store-Exclusive and Atomic instructions on page B2-190.*

The instructions are provided with ordering options, which map to the acquire and release definitions used in the architecture. The atomic instructions with release semantics have the same rules as Store-Release instructions regarding multi-copy atomicity.

For the SWP instructions, where the source and destination registers are the same, if the instruction generates a synchronous Data Abort, then the source register is restored to the value it held before the instruction was executed.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWP</td>
<td>Swap</td>
<td>SWP, SWPA, SWPAL, SWPL on page C6-1986</td>
</tr>
<tr>
<td>SWPB</td>
<td>Swap byte</td>
<td>SWPB, SWPAB, SWPALB, SWPLB on page C6-1982</td>
</tr>
<tr>
<td>SWPH</td>
<td>Swap halfword</td>
<td>SWPH, SWPAH, SWPALH, SWPLH on page C6-1984</td>
</tr>
</tbody>
</table>

**Compare and Swap**

The Compare and Swap instructions support only one addressing mode:
- Base register only.

See also *Load/store addressing modes on page C1-234.*

For the purpose of permission checking, and for watchpoints, all of the Compare and Swap instructions are treated as performing both a load and a store.

If FEAT_LSE2 is not implemented then:
- The CAS instructions require natural alignment.
- The CASP instructions require alignment to the total size of the memory being accessed.
For more information on alignment requirements and behaviors, see *Load-Exclusive/ Store-Exclusive and Atomic instructions* on page B2-190.

The instructions are provided with ordering options, which map to the acquire and release definitions used in the architecture. If a compare and swap instruction does not perform a store, then the instruction does not have release semantics, regardless of the instruction ordering options.

The atomic instructions with release semantics have the same rules as Store-Release instructions regarding multi-copy atomicity.

For the CAS and CASP instructions, the architecture permits that a data read clears any Exclusives monitors associated with that location, even if the compare subsequently fails. If these instructions generate a synchronous Data Abort, the registers which are compared and loaded are restored to the values held in the registers before the instruction was executed.

**Table C3-38 Compare and swap instructions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAS</td>
<td>Compare and swap</td>
<td><em>CAS, CASA, CASAL, CASL</em> on page C6-1247</td>
</tr>
<tr>
<td>CASB</td>
<td>Compare and swap byte</td>
<td><em>CASB, CASAB, CASALB, CASLB</em> on page C6-1240</td>
</tr>
<tr>
<td>CASH</td>
<td>Compare and swap halfword</td>
<td><em>CASH, CASAH, CASALH, CASLH</em> on page C6-1242</td>
</tr>
<tr>
<td>CASP</td>
<td>Compare and swap pair</td>
<td><em>CASP, CASPA, CASPAL, CASPL</em> on page C6-1244</td>
</tr>
</tbody>
</table>

**C3.2.13 Memory Tagging instructions**

If FEAT_MTE is implemented, the following instructions are implemented.

Table C3-39 on page C3-273 shows the Memory Tagging Extension Tag generation instructions.

**Table C3-39 Tag generation instructions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDG</td>
<td>Add immediate value to Logical Address Tag</td>
<td><em>ADDG</em> on page C6-1181</td>
</tr>
<tr>
<td>GMI</td>
<td>Tag Mask Insert</td>
<td><em>GMI</em> on page C6-1503</td>
</tr>
<tr>
<td>IRG</td>
<td>Random Logical Address Tag generation</td>
<td><em>IRG</em> on page C6-1509</td>
</tr>
<tr>
<td>SUBG</td>
<td>Subtract immediate value to Logical Address Tag</td>
<td><em>SUBG</em> on page C6-1971</td>
</tr>
</tbody>
</table>

Table C3-40 on page C3-273 shows the Memory Tagging Extension Pointer Arithmetic instructions.

**Table C3-40 Pointer Arithmetic**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBP(S)</td>
<td>Subtract address and set flags</td>
<td><em>SUBPS</em> on page C6-1973</td>
</tr>
</tbody>
</table>
Table C3-41 shows the Memory Tagging Extension Tag setting instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>STG</td>
<td>Store Allocation Tag to granule</td>
<td>STG on page C6-1860</td>
</tr>
<tr>
<td>STZG</td>
<td>Store Allocation Tag to granule Zeroing</td>
<td>STZG on page C6-1961</td>
</tr>
<tr>
<td>ST2G</td>
<td>Store Allocation Tag to two granules</td>
<td>ST2G on page C6-1835</td>
</tr>
<tr>
<td>STZ2G</td>
<td>Store Allocation Tag to two granules Zeroing</td>
<td>STZ2G on page C6-1959</td>
</tr>
<tr>
<td>STGP</td>
<td>Store Allocation Tag to memory</td>
<td>STGP on page C6-1863</td>
</tr>
</tbody>
</table>

Table C3-42 shows the Memory Tagging Extension Tag getting instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDG</td>
<td>Load Allocation Tag</td>
<td>LDG on page C6-1562</td>
</tr>
</tbody>
</table>

If FEAT_MTE2 is implemented, all of the FEAT_MTE instructions are implemented, plus the following instructions.

Table C3-43 shows the Memory Tagging Extension Bulk Allocation Tag access instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDGM</td>
<td>Load an IMPLEMENTATION DEFINED number of Allocation Tags</td>
<td>LDGM on page C6-1563</td>
</tr>
<tr>
<td>STGM</td>
<td>Store an IMPLEMENTATION DEFINED number of Allocation Tags</td>
<td>STGM on page C6-1862</td>
</tr>
<tr>
<td>STZGM</td>
<td>Store Allocation Tag to granule Zeroing Multiple</td>
<td>STZGM on page C6-1963</td>
</tr>
</tbody>
</table>

C3.2.14 Memory Copy and Memory Set instructions

If FEAT_MOPS is implemented, the CPY*, SETE*, SETM*, and SETP* instructions described in this section are implemented. If FEAT_MOPS and FEAT_MTE are implemented, the SETG* instructions described in this section are also implemented. Collectively, the instructions described in this section are referred to as Memory Copy and Memory Set instructions.

To perform a memory copy or memory set, three instructions - a prologue, then a main, and then an epilogue - are expected to be run in succession, with no instruction appearing in the code between them:
- To perform a memory copy, forward-only: CPYFP*, then CPYFM*, and then CPYFE*.
- To perform a memory copy, forward or backward: CPYP*, then CPYMP*, and then CPYEP*.
- To perform a memory set with tag setting: SETGP*, then SETGM*, and then SETGE*.
- To perform a memory set without tag setting: SETP*, then SETM*, and then SETE*.

The variant of each instruction is also expected to be the same throughout one of these sequences. For example, CPYFPWTWN, CPYFPWTWN, CPYFEWTWN.

Fetching of a Memory Copy and Memory Set instruction multiple times during its execution is permissible.

The handling of synchronous exceptions generated by Memory Copy and Memory Set instructions is described in ESR_ELx, FAR_ELx, Definition of a precise exception and imprecise exception on page D1-4638, and Synchronous exception entry on page D1-4642.
The handling of asynchronous exceptions generated by Memory Copy and Memory Set instructions is described in *Definition of a precise exception and imprecise exception* on page D1-4638 and *Asynchronous exception entry* on page D1-4644.

Memory Copy and Memory Set exceptions can be generated by execution of Memory Copy and Memory Set instructions restarting on a physical hardware PE implementation that is different from the physical hardware PE implementation that an exception was taken from. For more information, see *Memory Copy and Memory Set exceptions* on page D1-4654.

If an exception is taken during the execution of a Memory Copy and Memory Set instruction, that instruction has not been executed for the purposes of instruction counting, instruction tracing, statistical profiling, or single stepping.

For the purposes of single stepping and performance monitoring, each Memory Copy and Memory Set instruction is regarded as a single instruction that performs a store and, in the case of $\text{CPY}^*$ instructions, also performs a load.

For Memory Copy and Memory Set instructions, the following are not architecturally defined:

- The size of the memory transactions they create.
- The order between accesses to different addresses.

The $\text{CPY}^*$ instructions are guaranteed to make forward progress if none of the following four leaf level translation table entries fault:

- The source leaf level translation table entry held in $X_s$.
- The next leaf level translation table entry, as determined by the copy direction, adjacent to the source leaf level translation table entry held in $X_s$.
- The destination leaf level translation table entry held in $X_d$.
- The next leaf level translation table entry, as determined by the copy direction, adjacent to the destination leaf level translation table entry held in $X_d$.

The $\text{SET}^*$ instructions are guaranteed to make forward progress if neither of the following two leaf level translation table entries fault:

- The destination leaf level translation table entry held in $X_d$.
- The next leaf level translation table entry adjacent to the destination leaf level translation table entry held in $X_d$.

--- **Note** ---

The forward progress described in this section can be achieved by ensuring that, when a memory management fault is encountered, all bytes leading up to the fault have been operated on.

Table C3-44 on page C3-275 shows the memory copy, forward-only instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPYFE</td>
<td>Memory Copy Forward-only Epilogue</td>
<td>CPYFP, CPYFM, CPYFE Epilogue variant on page C6-1287</td>
</tr>
<tr>
<td>CPYFEN</td>
<td>Memory Copy Forward-only Epilogue, reads and writes non-temporal</td>
<td>CPYFPN, CPYFMN, CPYFEN Epilogue variant on page C6-1292</td>
</tr>
<tr>
<td>CPYFERN</td>
<td>Memory Copy Forward-only Epilogue, reads non-temporal</td>
<td>CPYFPRN, CPYFMRN, CPYFERN Epilogue variant on page C6-1297</td>
</tr>
<tr>
<td>CPYFERT</td>
<td>Memory Copy Forward-only Epilogue, reads unprivileged</td>
<td>CPYFPRT, CPYFMRT, CPYFERT Epilogue variant on page C6-1302</td>
</tr>
</tbody>
</table>
### Table C3-44 Memory copy, forward-only instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPYFERTN</td>
<td>Memory Copy Forward-only Epilogue, reads unprivileged, reads and writes non-temporal</td>
<td>CPYFPRTN, CPYFMRTN, CPYFERTN Epilogue variant on page C6-1307</td>
</tr>
<tr>
<td>CPYFERTRN</td>
<td>Memory Copy Forward-only Epilogue, reads unprivileged and non-temporal</td>
<td>CPYFPRTRN, CPYFMRTRN, CPYFERTRN Epilogue variant on page C6-1312</td>
</tr>
<tr>
<td>CPYFERTWN</td>
<td>Memory Copy Forward-only Epilogue, reads unprivileged, writes non-temporal</td>
<td>CPYFPRTWN, CPYFMRTWN, CPYFERTWN Epilogue variant on page C6-1317</td>
</tr>
<tr>
<td>CPYFET</td>
<td>Memory Copy Forward-only Epilogue, reads and writes unprivileged</td>
<td>CPYFPFT, CPYFMT, CPYFET Epilogue variant on page C6-1322</td>
</tr>
<tr>
<td>CPYFETN</td>
<td>Memory Copy Forward-only Epilogue, reads and writes unprivileged and non-temporal</td>
<td>CPYFPFTN, CPYFMTN, CPYFETN Epilogue variant on page C6-1327</td>
</tr>
<tr>
<td>CPYFETRN</td>
<td>Memory Copy Forward-only Epilogue, reads and writes unprivileged, reads non-temporal</td>
<td>CPYFPTRN, CPYFMTRN, CPYFETRN Epilogue variant on page C6-1332</td>
</tr>
<tr>
<td>CPYFETWN</td>
<td>Memory Copy Forward-only Epilogue, reads and writes unprivileged, writes non-temporal</td>
<td>CPYFPTRWN, CPYFMTRWN, CPYFETWN Epilogue variant on page C6-1337</td>
</tr>
<tr>
<td>CPYFEW</td>
<td>Memory Copy Forward-only Epilogue, writes non-temporal</td>
<td>CPYFPWN, CPYFMWN, CPYFEWN Epilogue variant on page C6-1342</td>
</tr>
<tr>
<td>CPYFEWT</td>
<td>Memory Copy Forward-only Epilogue, writes unprivileged</td>
<td>CPYFPWT, CPYFMWT, CPYFEWT Epilogue variant on page C6-1347</td>
</tr>
<tr>
<td>CPYFEWTN</td>
<td>Memory Copy Forward-only Epilogue, writes unprivileged, reads and writes non-temporal</td>
<td>CPYFPWTN, CPYFMWTN, CPYFEWTN Epilogue variant on page C6-1352</td>
</tr>
<tr>
<td>CPYFEWTN</td>
<td>Memory Copy Forward-only Epilogue, writes unprivileged, reads non-temporal</td>
<td>CPYFPWTRN, CPYFMWTRN, CPYFEWTN Epilogue variant on page C6-1357</td>
</tr>
<tr>
<td>CPYFMTWN</td>
<td>Memory Copy Forward-only Epilogue, writes unprivileged and non-temporal</td>
<td>CPYFPWTWN, CPYFMWTWN, CPYFEWTWN Epilogue variant on page C6-1362</td>
</tr>
</tbody>
</table>
### Table C3-44 Memory copy, forward-only instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPYFM</td>
<td>Memory Copy Forward-only Main</td>
<td>CPYFP, CPYFM, CPYFE Main variant on page C6-1287</td>
</tr>
<tr>
<td>CPYFMN</td>
<td>Memory Copy Forward-only Main, reads and writes non-temporal</td>
<td>CPYFPN, CPYFMN, CPYFEN Main variant on page C6-1292</td>
</tr>
<tr>
<td>CPYFMRN</td>
<td>Memory Copy Forward-only Main, reads non-temporal</td>
<td>CPYFPRN, CPYFMRN, CPYFERN Main variant on page C6-1297</td>
</tr>
<tr>
<td>CPYFMRT</td>
<td>Memory Copy Forward-only Main, reads unprivileged</td>
<td>CPYFPRT, CPYFMRT, CPYFERT Main variant on page C6-1302</td>
</tr>
<tr>
<td>CPYFMRTN</td>
<td>Memory Copy Forward-only Main, reads unprivileged, reads and writes non-temporal</td>
<td>CPYFPRTN, CPYFMRTN, CPYFERTN Main variant on page C6-1307</td>
</tr>
<tr>
<td>CPYFMRTN</td>
<td>Memory Copy Forward-only Main, reads unprivileged and non-temporal</td>
<td>CPYFPRTN, CPYFMRTN, CPYFERTN Main variant on page C6-1317</td>
</tr>
<tr>
<td>CPYFMRTWN</td>
<td>Memory Copy Forward-only Main, reads unprivileged, writes non-temporal</td>
<td>CPYFPRTWN, CPYFMRTWN, CPYFERTWN Main variant on page C6-1317</td>
</tr>
<tr>
<td>CPYFMT</td>
<td>Memory Copy Forward-only Main, reads and writes unprivileged</td>
<td>CPYFPT, CPYFMT, CPYFET Main variant on page C6-1322</td>
</tr>
<tr>
<td>CPYFMTN</td>
<td>Memory Copy Forward-only Main, reads and writes unprivileged and non-temporal</td>
<td>CPYFPTN, CPYFMTN, CPYFETN Main variant on page C6-1327</td>
</tr>
<tr>
<td>CPYFMTN</td>
<td>Memory Copy Forward-only Main, reads and writes unprivileged, reads non-temporal</td>
<td>CPYFPTN, CPYFMTN, CPYFETN Main variant on page C6-1327</td>
</tr>
<tr>
<td>CPYFMTWN</td>
<td>Memory Copy Forward-only Main, reads and writes unprivileged, writes non-temporal</td>
<td>CPYFPTWN, CPYFMTWN, CPYFETWN Main variant on page C6-1337</td>
</tr>
<tr>
<td>CPYFWMN</td>
<td>Memory Copy Forward-only Main, writes non-temporal</td>
<td>CPYFPWN, CPYFMWN, CPYFEWN Main variant on page C6-1342</td>
</tr>
<tr>
<td>CPYFMT</td>
<td>Memory Copy Forward-only Main, writes unprivileged</td>
<td>CPYFPWT, CPYFMWT, CPYFEWT Main variant on page C6-1347</td>
</tr>
</tbody>
</table>
### Table C3-44 Memory copy, forward-only instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPYFMWTN</td>
<td>Memory Copy Forward-only Main, writes unprivileged, reads and writes non-temporal</td>
<td>CPYFPWTN, CPYFMWTN, CPYFEWTN Main variant on page C6-1352</td>
</tr>
<tr>
<td>CPYFMWTRN</td>
<td>Memory Copy Forward-only Main, writes unprivileged, reads non-temporal</td>
<td>CPYFPWTRN, CPYFMWTRN, CPYFEWTRN Main variant on page C6-1357</td>
</tr>
<tr>
<td>CPYFMWTWN</td>
<td>Memory Copy Forward-only Main, writes unprivileged and non-temporal</td>
<td>CPYFPWWTWN, CPYFMWTWN, CPYFEWWTWN Main variant on page C6-1362</td>
</tr>
<tr>
<td>CPYFP</td>
<td>Memory Copy Forward-only Prologue</td>
<td>CPYFP, CPYFM, CPYFE Prologue variant on page C6-1287</td>
</tr>
<tr>
<td>CPYFPN</td>
<td>Memory Copy Forward-only Prologue, reads and writes non-temporal</td>
<td>CPYFPN, CPYFMN, CPYFEN Prologue variant on page C6-1292</td>
</tr>
<tr>
<td>CPYFPRN</td>
<td>Memory Copy Forward-only Prologue, reads non-temporal</td>
<td>CPYFPRN, CPYFMRN, CPYFERN Prologue variant on page C6-1297</td>
</tr>
<tr>
<td>CPYFPRT</td>
<td>Memory Copy Forward-only Prologue, reads unprivileged</td>
<td>CPYFPRT, CPYFMRT, CPYFERT Prologue variant on page C6-1302</td>
</tr>
<tr>
<td>CPYFPRTN</td>
<td>Memory Copy Forward-only Prologue, reads unprivileged, reads and writes non-temporal</td>
<td>CPYFPRTN, CPYFMRTN, CPYFERTN Prologue variant on page C6-1307</td>
</tr>
<tr>
<td>CPYFPRTRN</td>
<td>Memory Copy Forward-only Prologue, reads unprivileged and non-temporal</td>
<td>CPYFPRTRN, CPYFMRTRN, CPYFERTRN Prologue variant on page C6-1312</td>
</tr>
<tr>
<td>CPYFPRTWN</td>
<td>Memory Copy Forward-only Prologue, reads unprivileged, writes non-temporal</td>
<td>CPYFPRTWN, CPYFMRTWN, CPYFERTWN Prologue variant on page C6-1317</td>
</tr>
<tr>
<td>CPYFPT</td>
<td>Memory Copy Forward-only Prologue, reads and writes unprivileged</td>
<td>CPYFPT, CPYFMT, CPYFET Prologue variant on page C6-1322</td>
</tr>
<tr>
<td>CPYFPTN</td>
<td>Memory Copy Forward-only Prologue, reads and writes unprivileged and non-temporal</td>
<td>CPYFPTN, CPYFMTN, CPYFETN Prologue variant on page C6-1327</td>
</tr>
<tr>
<td>CPYFPTRN</td>
<td>Memory Copy Forward-only Prologue, reads and writes unprivileged, reads non-temporal</td>
<td>CPYFPTRN, CPYFMRN, CPYFERN Prologue variant on page C6-1332</td>
</tr>
</tbody>
</table>
Table C3-44 Memory copy, forward-only instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPYFPTWN</td>
<td>Memory Copy Forward-only Prologue, reads and writes unprivileged, writes non-temporal</td>
<td>CPYFPTWN, CPYFMTWN, CPYFETWN Prologue variant on page C6-1337</td>
</tr>
<tr>
<td>CPYFPwN</td>
<td>Memory Copy Forward-only Prologue, writes non-temporal</td>
<td>CPYFPWN, CPYFMWN, CPYFEWN Prologue variant on page C6-1342</td>
</tr>
<tr>
<td>CPYFPwT</td>
<td>Memory Copy Forward-only Prologue, writes unprivileged</td>
<td>CPYFPWT, CPYFMWT, CPYFEWT Prologue variant on page C6-1347</td>
</tr>
<tr>
<td>CPYFPwTN</td>
<td>Memory Copy Forward-only Prologue, writes unprivileged, reads and writes non-temporal</td>
<td>CPYFPWTN, CPYFMWTN, CPYFEWWTN Prologue variant on page C6-1352</td>
</tr>
<tr>
<td>CPYFPwTRN</td>
<td>Memory Copy Forward-only Prologue, writes unprivileged, reads non-temporal</td>
<td>CPYFPWTRN, CPYFMWTRN, CPYFEWTRN Prologue variant on page C6-1357</td>
</tr>
<tr>
<td>CPYFPwTWN</td>
<td>Memory Copy Forward-only Prologue, writes unprivileged and non-temporal</td>
<td>CPYFPWTWN, CPYFMWTWN, CPYFEWTWN Prologue variant on page C6-1362</td>
</tr>
</tbody>
</table>

Table C3-45 on page C3-279 shows the memory copy, forward or backward instructions.

Table C3-45 Memory copy, forward or backward instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPYE</td>
<td>Memory Copy Epilogue</td>
<td>CPYP, CPYM, CPYE Epilogue variant on page C6-1368</td>
</tr>
<tr>
<td>CPYEN</td>
<td>Memory Copy Epilogue, reads and writes non-temporal</td>
<td>CPYPN, CPYMN, CPYEN Epilogue variant on page C6-1374</td>
</tr>
<tr>
<td>CPYERN</td>
<td>Memory Copy Epilogue, reads non-temporal</td>
<td>CPYPRN, CPYMRN, CPYERN Epilogue variant on page C6-1380</td>
</tr>
<tr>
<td>CPYERT</td>
<td>Memory Copy Epilogue, reads unprivileged</td>
<td>CPYPRT, CPYMRT, CPYERT Epilogue variant on page C6-1386</td>
</tr>
</tbody>
</table>
### Table C3-45 Memory copy, forward or backward instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPYERTN</td>
<td>Memory Copy Epilogue, reads unprivileged, reads and writes non-temporal</td>
<td>CPYPRTN, CPYMRTN, CPYERTN Epilogue variant on page C6-1392</td>
</tr>
<tr>
<td>CPYERTRN</td>
<td>Memory Copy Epilogue, reads unprivileged and non-temporal</td>
<td>CPYPRTN, CPYMRTN, CPYERTRN Epilogue variant on page C6-1398</td>
</tr>
<tr>
<td>CPYERTWN</td>
<td>Memory Copy Epilogue, reads unprivileged, writes non-temporal</td>
<td>CPYPRTWN, CPYMRTWN, CPYERTWN Epilogue variant on page C6-1404</td>
</tr>
<tr>
<td>CPYET</td>
<td>Memory Copy Epilogue, reads and writes unprivileged</td>
<td>CPYPTE, CPYMT, CPYET Epilogue variant on page C6-1410</td>
</tr>
<tr>
<td>CPYETN</td>
<td>Memory Copy Epilogue, reads and writes unprivileged and non-temporal</td>
<td>CPYPTE, CPYMT, CPYETN Epilogue variant on page C6-1416</td>
</tr>
<tr>
<td>CPYETRN</td>
<td>Memory Copy Epilogue, reads and writes unprivileged, reads non-temporal</td>
<td>CPYPTRN, CPYMTRN, CPYETRN Epilogue variant on page C6-1422</td>
</tr>
<tr>
<td>CPYETW</td>
<td>Memory Copy Epilogue, reads and writes unprivileged, writes non-temporal</td>
<td>CPYPTW, CPYMWT, CPYETW Epilogue variant on page C6-1428</td>
</tr>
<tr>
<td>CPYEWN</td>
<td>Memory Copy Epilogue, writes non-temporal</td>
<td>CPYPWN, CPYMWN, CPYEWN Epilogue variant on page C6-1434</td>
</tr>
<tr>
<td>CPYEWT</td>
<td>Memory Copy Epilogue, writes unprivileged</td>
<td>CPYPWT, CPYMWT, CPYEWT Epilogue variant on page C6-1440</td>
</tr>
<tr>
<td>CPYEWTN</td>
<td>Memory Copy Epilogue, writes unprivileged, reads and writes non-temporal</td>
<td>CPYPWTN, CPYMWTN, CPYEWTN Epilogue variant on page C6-1446</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Instruction</td>
<td>See</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>CPYEWTRN</td>
<td>Memory Copy Epilogue, writes unprivileged, reads non-temporal</td>
<td>CPYPWTRN, CPYMWTRN, CPYEWTRN Epilogue variant on page C6-1452</td>
</tr>
<tr>
<td>CPYMWTWN</td>
<td>Memory Copy Epilogue, writes unprivileged and non-temporal</td>
<td>CPYPWTWN, CPYMWTWN, CPYEWTWN Epilogue variant on page C6-1458</td>
</tr>
<tr>
<td>CPYM</td>
<td>Memory Copy Main</td>
<td>CPYP, CPYM, CPYE Main variant on page C6-1368</td>
</tr>
<tr>
<td>CPYMN</td>
<td>Memory Copy Main, reads and writes non-temporal</td>
<td>CPYPMN, CPYMN, CPYEN Main variant on page C6-1374</td>
</tr>
<tr>
<td>CPYMRN</td>
<td>Memory Copy Main, reads non-temporal</td>
<td>CPYPRPN, CPYMRN, CPYERN Main variant on page C6-1380</td>
</tr>
<tr>
<td>CPYMRT</td>
<td>Memory Copy Main, reads unprivileged</td>
<td>CPYPRT, CPYMRT, CPYERT Main variant on page C6-1386</td>
</tr>
<tr>
<td>CPYMRTN</td>
<td>Memory Copy Main, reads unprivileged, reads and writes non-temporal</td>
<td>CPYPRTN, CPYMRTN, CPYERTN Main variant on page C6-1392</td>
</tr>
<tr>
<td>CPYMRTRN</td>
<td>Memory Copy Main, reads unprivileged and non-temporal</td>
<td>CPYPRTRN, CPYMRTN, CPYERTRN Main variant on page C6-1398</td>
</tr>
<tr>
<td>CPYMRTWN</td>
<td>Memory Copy Main, reads unprivileged, writes non-temporal</td>
<td>CPYPRTWN, CPYMRTWN, CPYERTWN Main variant on page C6-1404</td>
</tr>
<tr>
<td>CPYMT</td>
<td>Memory Copy Main, reads and writes unprivileged</td>
<td>CPYP, CPYMT, CPYET Main variant on page C6-1410</td>
</tr>
<tr>
<td>CPYMTN</td>
<td>Memory Copy Main, reads and writes unprivileged and non-temporal</td>
<td>CPYPTN, CPYMTN, CPYETN Main variant on page C6-1416</td>
</tr>
<tr>
<td>CPYMTRN</td>
<td>Memory Copy Main, reads and writes unprivileged, reads non-temporal</td>
<td>CPYPTRN, CPYMTRN, CPYETRN Main variant on page C6-1422</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Instruction</td>
<td>See</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>CPYMTWN</td>
<td>Memory Copy Main, reads and writes unprivileged, writes non-temporal</td>
<td>CPYPTWN, CPYMTWN, CPYETWN Main variant on page C6-1428</td>
</tr>
<tr>
<td>CPYMNW</td>
<td>Memory Copy Main, writes non-temporal</td>
<td>CPYPWN, CPYMNW, CPYEWN Main variant on page C6-1434</td>
</tr>
<tr>
<td>CPYMNWT</td>
<td>Memory Copy Main, writes unprivileged</td>
<td>CPYPWT, CPYMWT, CPYEWT Main variant on page C6-1440</td>
</tr>
<tr>
<td>CPYMNWN</td>
<td>Memory Copy Main, writes unprivileged, reads and writes non-temporal</td>
<td>CPYPWTN, CPYMWTN, CPYEWTN Main variant on page C6-1446</td>
</tr>
<tr>
<td>CPYMNTRN</td>
<td>Memory Copy Main, writes unprivileged, reads non-temporal</td>
<td>CPYPWTRN, CPYMWTRN, CPYEWTN Main variant on page C6-1452</td>
</tr>
<tr>
<td>CPYMNWN</td>
<td>Memory Copy Main, writes unprivileged and non-temporal</td>
<td>CPYPWTWN, CPYMWTWN, CPYEWTWN Main variant on page C6-1458</td>
</tr>
<tr>
<td>CPYP</td>
<td>Memory Copy Prologue</td>
<td>CPYP, CPYM, CPYE Prologue variant on page C6-1368</td>
</tr>
<tr>
<td>CPYPN</td>
<td>Memory Copy Prologue, reads and writes non-temporal</td>
<td>CPYPN, CPYMN, CPYEN Prologue variant on page C6-1374</td>
</tr>
<tr>
<td>CPYPRN</td>
<td>Memory Copy Prologue, reads non-temporal</td>
<td>CPYPRN, CPYMRN, CPYERN Prologue variant on page C6-1380</td>
</tr>
<tr>
<td>CPYPRT</td>
<td>Memory Copy Prologue, reads unprivileged</td>
<td>CPYPRT, CPYMRT, CPYERT Prologue variant on page C6-1386</td>
</tr>
<tr>
<td>CPYPRTN</td>
<td>Memory Copy Prologue, reads unprivileged, reads and writes non-temporal</td>
<td>CPYPRTN, CPYMRTN, CPYERTN Prologue variant on page C6-1392</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Instruction</td>
<td>See</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>CPYPRTRN</td>
<td>Memory Copy Prologue, reads unprivileged and non-temporal</td>
<td>CPYPRTRN, CPYMTRN, CPYETRN Prologue variant on page C6-1398</td>
</tr>
<tr>
<td>CPYPRTWN</td>
<td>Memory Copy Prologue, reads unprivileged, writes non-temporal</td>
<td>CPYPRTWN, CPYMRTWN, CPYERTWN Prologue variant on page C6-1404</td>
</tr>
<tr>
<td>CPYPT</td>
<td>Memory Copy Prologue, reads and writes unprivileged</td>
<td>CPYPT, CPYMT, CPYET Prologue variant on page C6-1410</td>
</tr>
<tr>
<td>CPYPTN</td>
<td>Memory Copy Prologue, reads and writes unprivileged and non-temporal</td>
<td>CPYPTN, CPYMTN, CPYETN Prologue variant on page C6-1416</td>
</tr>
<tr>
<td>CPYPRTRN</td>
<td>Memory Copy Prologue, reads and writes unprivileged, reads non-temporal</td>
<td>CPYPRTRN, CPYMTRN, CPYETRN Prologue variant on page C6-1422</td>
</tr>
<tr>
<td>CPYPTWN</td>
<td>Memory Copy Prologue, reads and writes unprivileged, writes non-temporal</td>
<td>CPYPTWN, CPYMTRN, CPYETRN Prologue variant on page C6-1428</td>
</tr>
<tr>
<td>CPYPWN</td>
<td>Memory Copy Prologue, writes non-temporal</td>
<td>CPYPWN, CPYMWN, CPYEWN Prologue variant on page C6-1434</td>
</tr>
<tr>
<td>CPYPWT</td>
<td>Memory Copy Prologue, writes unprivileged</td>
<td>CPYPWT, CPYMWT, CPYEWT Prologue variant on page C6-1440</td>
</tr>
</tbody>
</table>
Table C3-45 Memory copy, forward or backward instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPYPWTN</td>
<td>Memory Copy Prologue, writes unprivileged, reads and writes non-temporal</td>
<td>CPYPWTN, CPYMWTN, CPYEWTN Prologue variant on page C6-1446</td>
</tr>
<tr>
<td>CPYPWTRN</td>
<td>Memory Copy Prologue, writes unprivileged, reads non-temporal</td>
<td>CPYPWTRN, CPYMWTN, CPYEWTRN Prologue variant on page C6-1452</td>
</tr>
<tr>
<td>CPYPWTWN</td>
<td>Memory Copy Prologue, writes unprivileged and non-temporal</td>
<td>CPYPWTWN, CPYMWTWN, CPYEWWTWN Prologue variant on page C6-1458</td>
</tr>
</tbody>
</table>

Table C3-46 on page C3-284 shows the memory set with tag setting instructions.

Table C3-46 Memory set with tag setting instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETGE</td>
<td>Memory Set with tag setting Epilogue</td>
<td>SETGP, SETGM, SETGE Epilogue variant on page C6-1785</td>
</tr>
<tr>
<td>SETGEN</td>
<td>Memory Set with tag setting Epilogue, non-temporal</td>
<td>SETGPN, SETGMN, SETGEN Epilogue variant on page C6-1790</td>
</tr>
<tr>
<td>SETGET</td>
<td>Memory Set with tag setting Epilogue, unprivileged</td>
<td>SETGPT, SETGMT, SETGET Epilogue variant on page C6-1795</td>
</tr>
<tr>
<td>SETGETN</td>
<td>Memory Set with tag setting Epilogue, unprivileged and non-temporal</td>
<td>SETGPTN, SETGTMN, SETGETN Epilogue variant on page C6-1800</td>
</tr>
<tr>
<td>SETGM</td>
<td>Memory Set with tag setting Main</td>
<td>SETGP, SETGM, SETGE Main variant on page C6-1785</td>
</tr>
<tr>
<td>SETGMN</td>
<td>Memory Set with tag setting Main, non-temporal</td>
<td>SETGPN, SETGMN, SETGEN Main variant on page C6-1790</td>
</tr>
<tr>
<td>SETGMT</td>
<td>Memory Set with tag setting Main, unprivileged</td>
<td>SETGPT, SETGMT, SETGET Main variant on page C6-1795</td>
</tr>
<tr>
<td>SETGMTN</td>
<td>Memory Set with tag setting Main, unprivileged and non-temporal</td>
<td>SETGPTN, SETGTMN, SETGETN Main variant on page C6-1800</td>
</tr>
<tr>
<td>SETGP</td>
<td>Memory Set with tag setting Prologue</td>
<td>SETGP, SETGM, SETGE Prologue variant on page C6-1785</td>
</tr>
</tbody>
</table>
### Table C3-46 Memory set with tag setting instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETGPN</td>
<td>Memory Set with tag setting Prologue, non-temporal</td>
<td>SETGPN, SETGMN, SETGEN Prologue variant on page C6-1790</td>
</tr>
<tr>
<td>SETGPT</td>
<td>Memory Set with tag setting Prologue, unprivileged</td>
<td>SETGPT, SETGMT, SETGET Prologue variant on page C6-1795</td>
</tr>
<tr>
<td>SETGPTN</td>
<td>Memory Set with tag setting Prologue, unprivileged and non-temporal</td>
<td>SETGPTN, SETGMTN, SETGETN Prologue variant on page C6-1800</td>
</tr>
</tbody>
</table>

Table C3-47 on page C3-285 shows the memory set without tag setting instructions.

### Table C3-47 Memory set without tag setting instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETE</td>
<td>Memory Set Epilogue</td>
<td>SETP, SETM, SETE Epilogue variant on page C6-1805</td>
</tr>
<tr>
<td>SETEN</td>
<td>Memory Set Epilogue, non-temporal</td>
<td>SETPN, SETMN, SETEN Epilogue variant on page C6-1809</td>
</tr>
<tr>
<td>SETET</td>
<td>Memory Set Epilogue, unprivileged</td>
<td>SETPT, SETMT, SETET Epilogue variant on page C6-1813</td>
</tr>
<tr>
<td>SETETN</td>
<td>Memory Set Epilogue, unprivileged and non-temporal</td>
<td>SETPTN, SETMTN, SETETN Epilogue variant on page C6-1817</td>
</tr>
<tr>
<td>SETM</td>
<td>Memory Set Main</td>
<td>SETP, SETM, SETE Main variant on page C6-1805</td>
</tr>
<tr>
<td>SETMN</td>
<td>Memory Set Main, non-temporal</td>
<td>SETPN, SETMN, SETEN Main variant on page C6-1809</td>
</tr>
<tr>
<td>SETMT</td>
<td>Memory Set Main, unprivileged</td>
<td>SETPT, SETMT, SETET Main variant on page C6-1813</td>
</tr>
<tr>
<td>SETMTN</td>
<td>Memory Set Main, unprivileged and non-temporal</td>
<td>SETPTN, SETMTN, SETETN Main variant on page C6-1817</td>
</tr>
<tr>
<td>SETP</td>
<td>Memory Set Prologue</td>
<td>SETP, SETM, SETE Prologue variant on page C6-1805</td>
</tr>
</tbody>
</table>
Table C3-47 Memory set without tag setting instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETPN</td>
<td>Memory Set Prologue, non-temporal</td>
<td>SETPN, SETMN, SETEN Prologue variant on page C6-1809</td>
</tr>
<tr>
<td>SETPT</td>
<td>Memory Set Prologue, unprivileged</td>
<td>SETPT, SETMT, SETET Prologue variant on page C6-1813</td>
</tr>
<tr>
<td>SETPTN</td>
<td>Memory Set Prologue, unprivileged and non-temporal</td>
<td>SETPTN, SETMTN, SETETN Prologue variant on page C6-1817</td>
</tr>
</tbody>
</table>
C3.3 Loads and stores - SVE

SVE vector load and store instructions transfer data in memory to or from elements of one or more vector or predicate transfer registers. SVE also includes vector prefetch instructions that provide read and write hints to the memory system. For SVE predicated load, store, and prefetch instructions, the memory element access size and type that is associated with each vector element is specified by a suffix to the instruction mnemonic, independently of the element size of the transfer registers. For example, LD1SH. The following table shows the supported instruction suffixes for SVE load, store, and prefetch instructions:

<table>
<thead>
<tr>
<th>Instruction suffix</th>
<th>Memory element access size and type</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Unsigned byte</td>
</tr>
<tr>
<td>H</td>
<td>Unsigned halfword or half-precision floating-point</td>
</tr>
<tr>
<td>W</td>
<td>Unsigned word or single-precision floating-point</td>
</tr>
<tr>
<td>D</td>
<td>Unsigned doubleword or double-precision floating-point</td>
</tr>
<tr>
<td>SB</td>
<td>Signed byte</td>
</tr>
<tr>
<td>SH</td>
<td>Signed halfword</td>
</tr>
<tr>
<td>SW</td>
<td>Signed word</td>
</tr>
</tbody>
</table>

The element size of the transfer registers is always greater than or equal to the memory element access size. When the element size of the transfer registers is strictly greater than the memory element access size, then these are referred to as unpacked data accesses. In the case of unpacked data accesses:

- For load instructions, each element access is sign-extended or zero-extended to fill the vector element, according to its memory element access size and type.
- For store instructions, each vector element is truncated to the memory element access size.

Where the vector element size and the memory element access size are the same, then these are referred to as packed data accesses. Signed access types are not supported for packed data accesses. Packed and unpacked access sizes and types relate to the vector element size of the transfer registers, as defined in the following table:

<table>
<thead>
<tr>
<th>Vector element</th>
<th>Packed access suffix</th>
<th>Unpacked access suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>.B</td>
<td>B</td>
<td>-</td>
</tr>
<tr>
<td>.H</td>
<td>H</td>
<td>B, SB</td>
</tr>
<tr>
<td>.S</td>
<td>W</td>
<td>H, SH, B, SB</td>
</tr>
<tr>
<td>.D</td>
<td>D</td>
<td>W, SW, H, SH, B, SB</td>
</tr>
</tbody>
</table>

For gather-load and scatter-store instructions, the vector element size can only be .S or .D. This means that any non-contiguous memory element access of less than a word is unpacked. Non-contiguous memory element accesses of a word can be either packed or unpacked, depending on the vector element size.

Load, store, and prefetch instructions consist of the following:

- Predicated single vector contiguous element accesses on page C3-288.
- Predicated multiple vector contiguous structure load/store on page C3-291.
- Predicated non-contiguous element accesses on page C3-293.
- Predicated replicating element loads on page C3-295.
- Unpredicated vector register load/store on page C3-296.
- Unpredicated predicate register load/store on page C3-296.
All predicated load instructions zero the *Inactive elements* of the destination vector, except for Non-fault loads and First-fault loads when the corresponding FFR element is FALSE.

Prefetch instructions provide hints to hardware and do not change architectural state. Therefore, a *Governing predicate* for a prefetch instruction provides an additional hint which indicates the memory locations to be prefetched. Prefetch instructions require a prefetch operation specifier. SVE prefetch instructions support all of the prefetch operations except for the PLI prefetch operand types.

Load, store, and prefetch instructions that multiply a scalar index register or an index vector element by the memory element access size specify a shift type, followed by a shift amount in bits. The shift type can be one of *LSL*, *SXTW*, or *UXTW*. The shift amount is always Log2 of the memory element access size, in bytes. The shift amount defaults to zero when the memory element access size is a byte, and the shift size can be omitted. The shift type of *LSL* must be omitted if the shift amount is omitted.

When included as part of the assembler syntax for an instruction, $\text{MUL } VL$ indicates that the specified immediate index value is multiplied by the size of the addressed vector or predicate in memory, measured in bytes, irrespective of predication. For a detailed description of the meaning of this assembler syntax for each instruction, see the appropriate subsection below. When used in pseudocode, the symbol $\text{vl}$ represents the vector length, measured in bits.

SVE load, store, and prefetch instructions do not support pre-indexed or post-indexed addressing.

### C3.3.1 Predicated single vector contiguous element accesses

Predicated contiguous load and store instructions access memory locations starting from an address that is defined by a scalar base register plus either:

- A scalar index register.
- An immediate index value that is in the range -8 to 7, inclusive. This defaults to zero if omitted.

The predicated contiguous load and store instructions have two supporting addressing modes:

- Scalar base plus immediate index.
- Scalar base plus scalar index.

Predicated contiguous prefetch instructions address memory locations in a similar manner, with the index being either:

- A scalar index register.
- An immediate index value that is in the range of -32 to 31, inclusive. This defaults to zero if omitted.

For predicated contiguous load and store SVE instructions:

- The immediate index value is a vector index, not an element index. The immediate index value is multiplied by the number of vector elements, irrespective of predication, and then multiplied by the memory element access size in bytes. The resulting offset is incremented following each element access by the memory element access size.
- The scalar index register value is multiplied by the memory element access size in bytes. The index value is incremented by one after each element access, but the scalar index register is not updated by the instruction.
When alignment checking is enabled for loads and stores, the value of the base address register must be aligned to the memory element access size.

**Table C3-48 Predicated single vector contiguous element accesses**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
</table>
| LD1      | Contiguous load unsigned bytes to vector (scalar plus immediate, scalar plus scalar) | LD1B (scalar plus immediate) on page C8-3409  
LD1B (scalar plus scalar) on page C8-3412 |
|          | Contiguous load doublewords to vector (scalar plus immediate, scalar plus scalar) | LD1D (scalar plus immediate) on page C8-3420  
LD1D (scalar plus scalar) on page C8-3422 |
|          | Contiguous load unsigned halfwords to vector (scalar plus immediate, scalar plus scalar) | LD1H (scalar plus immediate) on page C8-3429  
LD1H (scalar plus scalar) on page C8-3432 |
|          | Contiguous load and replicate 32-bytes (scalar plus immediate, scalar plus scalar) | LD1ROB (scalar plus immediate) on page C8-3449  
LD1ROB (scalar plus scalar) on page C8-3451 |
|          | Contiguous load and replicate four doublewords (scalar plus immediate, scalar plus scalar) | LD1ROD (scalar plus immediate) on page C8-3453  
LD1ROD (scalar plus scalar) on page C8-3455 |
|          | Contiguous load and replicate 16 halfwords (scalar plus immediate, scalar plus scalar) | LD1ROH (scalar plus immediate) on page C8-3457  
LD1ROH (scalar plus scalar) on page C8-3459 |
|          | Contiguous load and replicate eight words (scalar plus immediate, scalar plus scalar) | LD1ROW (scalar plus immediate) on page C8-3461  
LD1ROW (scalar plus scalar) on page C8-3463 |
|          | Contiguous load and replicate 16 bytes (scalar plus immediate, scalar plus scalar) | LD1RQB (scalar plus immediate) on page C8-3465  
LD1RQB (scalar plus scalar) on page C8-3467 |
|          | Contiguous load and replicate two doublewords (scalar plus immediate, scalar plus scalar) | LD1RQD (scalar plus immediate) on page C8-3469  
LD1RQD (scalar plus scalar) on page C8-3471 |
|          | Contiguous load and replicate eight halfwords (scalar plus immediate, scalar plus scalar) | LD1RQH (scalar plus immediate) on page C8-3473  
LD1RQH (scalar plus scalar) on page C8-3475 |
|          | Contiguous load and replicate four words (scalar plus immediate, scalar plus scalar) | LD1RW (scalar plus immediate) on page C8-3477  
LD1RW (scalar plus scalar) on page C8-3479 |
|          | Contiguous load signed bytes to vector (scalar plus immediate) | ST1B (scalar plus immediate) on page C8-4163  
ST1B (scalar plus scalar) on page C8-4165 |
|          | Contiguous load signed words to vector (scalar plus immediate) | ST1D (scalar plus immediate) on page C8-4172  
ST1D (scalar plus scalar) on page C8-4174 |
|          | Contiguous load unsigned words to vector (scalar plus immediate) | ST1H (scalar plus immediate) on page C8-4181  
ST1H (scalar plus scalar) on page C8-4183 |
|          | Contiguous store bytes from vector (scalar plus immediate, scalar plus scalar) | ST1B (scalar plus immediate) on page C8-4191  
ST1B (scalar plus scalar) on page C8-4193 |
|          | Contiguous store doublewords from vector (scalar plus immediate, scalar plus scalar) | ST1D (scalar plus immediate) on page C8-4191  
ST1D (scalar plus scalar) on page C8-4193 |
|          | Contiguous store halfwords from vector (scalar plus immediate, scalar plus scalar) | ST1H (scalar plus immediate) on page C8-4191  
ST1H (scalar plus scalar) on page C8-4193 |
|          | Contiguous store words from vector (scalar plus immediate, scalar plus scalar) | ST1W (scalar plus immediate) on page C8-4191  
ST1W (scalar plus scalar) on page C8-4193 |
### Table C3-48 Predicated single vector contiguous element accesses (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDFF1</td>
<td>Contiguous load first-fault unsigned bytes to vector (scalar plus scalar)</td>
<td>LDFF1B (scalar plus scalar) on page C8-3578</td>
</tr>
<tr>
<td></td>
<td>Contiguous load first-fault doublewords to vector (scalar plus scalar)</td>
<td>LDFF1D (scalar plus scalar) on page C8-3586</td>
</tr>
<tr>
<td></td>
<td>Contiguous load first-fault unsigned halfwords to vector (scalar plus scalar)</td>
<td>LDFF1H (scalar plus scalar) on page C8-3593</td>
</tr>
<tr>
<td></td>
<td>Contiguous load first-fault signed bytes to vector (scalar plus scalar)</td>
<td>LDFF1SB (scalar plus scalar) on page C8-3602</td>
</tr>
<tr>
<td></td>
<td>Contiguous load first-fault signed halfwords to vector (scalar plus scalar)</td>
<td>LDFF1SH (scalar plus scalar) on page C8-3610</td>
</tr>
<tr>
<td></td>
<td>Contiguous load first-fault signed words to vector (scalar plus scalar)</td>
<td>LDFF1SW (scalar plus scalar) on page C8-3619</td>
</tr>
<tr>
<td></td>
<td>Contiguous load first-fault unsigned words to vector (scalar plus scalar)</td>
<td>LDFF1W (scalar plus scalar) on page C8-3626</td>
</tr>
<tr>
<td>LDNF1</td>
<td>Contiguous load non-fault unsigned bytes to vector</td>
<td>LDNF1B on page C8-3635</td>
</tr>
<tr>
<td></td>
<td>Contiguous load non-fault doublewords to vector</td>
<td>LDNF1D on page C8-3638</td>
</tr>
<tr>
<td></td>
<td>Contiguous load non-fault unsigned halfwords to vector</td>
<td>LDNF1H on page C8-3640</td>
</tr>
<tr>
<td></td>
<td>Contiguous load non-fault signed bytes to vector</td>
<td>LDNF1SB on page C8-3643</td>
</tr>
<tr>
<td></td>
<td>Contiguous load non-fault signed halfwords to vector</td>
<td>LDNF1SH on page C8-3646</td>
</tr>
<tr>
<td></td>
<td>Contiguous load non-fault signed words to vector</td>
<td>LDNF1SW on page C8-3648</td>
</tr>
<tr>
<td></td>
<td>Contiguous load non-fault unsigned words to vector</td>
<td>LDNF1W on page C8-3650</td>
</tr>
<tr>
<td>LDNT1</td>
<td>Contiguous load non-temporal bytes to vector (scalar plus immediate, scalar plus scalar)</td>
<td>LDNT1B (scalar plus immediate) on page C8-3652</td>
</tr>
<tr>
<td></td>
<td>Contiguous load non-temporal doublewords to vector (scalar plus immediate, scalar plus scalar)</td>
<td>LDNT1D (scalar plus immediate) on page C8-3658</td>
</tr>
<tr>
<td></td>
<td>Contiguous load non-temporal halfwords to vector (scalar plus immediate, scalar plus scalar)</td>
<td>LDNT1H (scalar plus immediate) on page C8-3664</td>
</tr>
<tr>
<td></td>
<td>Contiguous load non-temporal words to vector (scalar plus immediate, scalar plus scalar)</td>
<td>LDNT1W (scalar plus immediate) on page C8-3676</td>
</tr>
<tr>
<td>STNT1</td>
<td>Contiguous store non-temporal bytes from vector (scalar plus immediate, scalar plus scalar)</td>
<td>STNT1B (scalar plus immediate) on page C8-4249</td>
</tr>
<tr>
<td></td>
<td>Contiguous store non-temporal doublewords from vector (scalar plus immediate, scalar plus scalar)</td>
<td>STNT1D (scalar plus immediate) on page C8-4255</td>
</tr>
<tr>
<td></td>
<td>Contiguous store non-temporal halfwords from vector (scalar plus immediate, scalar plus scalar)</td>
<td>STNT1H (scalar plus immediate) on page C8-4261</td>
</tr>
<tr>
<td></td>
<td>Contiguous store non-temporal words from vector (scalar plus immediate, scalar plus scalar)</td>
<td>STNT1W (scalar plus immediate) on page C8-4267</td>
</tr>
</tbody>
</table>
C3.3 Loads and stores - SVE

C3.3.2 Predicated multiple vector contiguous structure load/store

Predicated multiple vector contiguous structure load/store instructions are defined by a scalar base register plus either:

• A scalar index register.
• An immediate index that is a multiple of N, in the range -8N to 7N, inclusive. This defaults to zero if omitted.

The predicated contiguous structure load and store instructions have two supporting addressing modes:

• Scalar base plus immediate.
• Scalar base plus scalar index.

For the predicated multiple vector contiguous structure load/store SVE instructions:

• The immediate index value is a vector index, not an element index. The immediate index value is multiplied by the number of vector elements, irrespective of predication, and then multiplied by the memory element access size in bytes. The resulting offset is incremented following each element access by the memory element access size.

• The scalar index register value is multiplied by the memory element access size in bytes. Following each element access, the index value is incremented by one but the instruction does not update the scalar index register.

• Each predicate element applies to a single structure in memory, or equivalently to the same element number within each of the two, three, or four transferred vector registers.

• These instructions support packed data accesses only.
When alignment checking is enabled for loads and stores, the base address must be aligned to the element access size.

### Table C3-49 Predicated multiple vector contiguous structure load/store

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
</table>
| **LD2** | Contiguous load 2-byte structures to two vectors (scalar plus immediate, scalar plus scalar) | LD2B (scalar plus immediate) on page C8-3530  
LD2B (scalar plus scalar) on page C8-3532  
LD2D (scalar plus immediate, scalar plus scalar) on page C8-3534  
LD2D (scalar plus scalar) on page C8-3536  
LD2H (scalar plus immediate, scalar plus scalar) on page C8-3538  
LD2H (scalar plus scalar) on page C8-3540  
LD2W (scalar plus immediate, scalar plus scalar) on page C8-3542  
LD2W (scalar plus scalar) on page C8-3544 |
| **LD3** | Contiguous load 3-byte structures to three vectors (scalar plus immediate, scalar plus scalar) | LD3B (scalar plus immediate) on page C8-3546  
LD3B (scalar plus scalar) on page C8-3548  
LD3D (scalar plus immediate, scalar plus scalar) on page C8-3550  
LD3D (scalar plus scalar) on page C8-3552  
LD3H (scalar plus immediate, scalar plus scalar) on page C8-3554  
LD3H (scalar plus scalar) on page C8-3556  
LD3W (scalar plus immediate, scalar plus scalar) on page C8-3558  
LD3W (scalar plus scalar) on page C8-3560 |
| **LD4** | Contiguous load 4-byte structures to four vectors (scalar plus immediate, scalar plus scalar) | LD4B (scalar plus immediate) on page C8-3562  
LD4B (scalar plus scalar) on page C8-3564  
LD4D (scalar plus immediate, scalar plus scalar) on page C8-3566  
LD4D (scalar plus scalar) on page C8-3568  
LD4H (scalar plus immediate, scalar plus scalar) on page C8-3570  
LD4H (scalar plus scalar) on page C8-3572  
LD4W (scalar plus immediate, scalar plus scalar) on page C8-3574  
LD4W (scalar plus scalar) on page C8-3576 |
| **ST2** | Contiguous store 2-byte structures from two vectors (scalar plus immediate, scalar plus scalar) | ST2B (scalar plus immediate) on page C8-4201  
ST2B (scalar plus scalar) on page C8-4203  
ST2D (scalar plus immediate, scalar plus scalar) on page C8-4205  
ST2D (scalar plus scalar) on page C8-4207  
ST2H (scalar plus immediate, scalar plus scalar) on page C8-4209  
ST2H (scalar plus scalar) on page C8-4211  
ST2W (scalar plus immediate, scalar plus scalar) on page C8-4213  
ST2W (scalar plus scalar) on page C8-4215 |
### C3.3 Loads and stores - SVE

#### C3.3.3 Predicated non-contiguous element accesses

Predicated non-contiguous element accesses address non-contiguous memory locations that are specified by either:

- A scalar base register plus a vector of indices or offsets.
- A vector of base addresses plus an immediate byte offset. The immediate byte offset is a multiple of the memory element access size, in the range 0 to 31 times the memory element access size, inclusive, and defaults to zero if omitted.

The predicated non-contiguous element accesses have two supporting addressing modes:

- Scalar base plus 64-bit vector index.
- Scalar base plus 32-bit vector index.
- Vector base plus immediate offset.

For this group of SVE instructions:

- Vector registers used as part of the address must specify a vector element size of 32 bits or 64 bits, .S or .D. For load and store instructions, the transfer register must specify the same vector element size.
- If the index vector register contains 32-bit index values then the lowest 32 bits of each index vector element can either be zero-extended or sign-extended to 64 bits.
- For load and store instructions, the index vector elements are then optionally multiplied by the memory element access size, in bytes, if a shift amount is specified. For prefetch instructions, the index vector elements are always multiplied by the memory element access size, in bytes.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST3</td>
<td>Contiguous store 3-byte structures from three vectors (scalar plus immediate, scalar plus scalar)</td>
<td>ST3B (scalar plus immediate) on page C8-4217&lt;br&gt;ST3B (scalar plus scalar) on page C8-4219</td>
</tr>
<tr>
<td></td>
<td>Contiguous store three-doubleword structures from three vectors (scalar plus immediate, scalar plus scalar)</td>
<td>ST3D (scalar plus immediate) on page C8-4221&lt;br&gt;ST3D (scalar plus scalar) on page C8-4223</td>
</tr>
<tr>
<td></td>
<td>Contiguous store three-halfword structures from three vectors (scalar plus immediate, scalar plus scalar)</td>
<td>ST3H (scalar plus immediate) on page C8-4225&lt;br&gt;ST3H (scalar plus scalar) on page C8-4227</td>
</tr>
<tr>
<td></td>
<td>Contiguous store three-word structures from three vectors (scalar plus immediate, scalar plus scalar)</td>
<td>ST3W (scalar plus immediate) on page C8-4229&lt;br&gt;ST3W (scalar plus scalar) on page C8-4231</td>
</tr>
<tr>
<td>ST4</td>
<td>Contiguous store 4-byte structures from four vectors (scalar plus immediate, scalar plus scalar)</td>
<td>ST4B (scalar plus immediate) on page C8-4233&lt;br&gt;ST4B (scalar plus scalar) on page C8-4235</td>
</tr>
<tr>
<td></td>
<td>Contiguous store four-doubleword structures from four vectors (scalar plus immediate, scalar plus scalar)</td>
<td>ST4D (scalar plus immediate) on page C8-4237&lt;br&gt;ST4D (scalar plus scalar) on page C8-4239</td>
</tr>
<tr>
<td></td>
<td>Contiguous store four-halfword structures from four vectors (scalar plus immediate, scalar plus scalar)</td>
<td>ST4H (scalar plus immediate) on page C8-4241&lt;br&gt;ST4H (scalar plus scalar) on page C8-4243</td>
</tr>
<tr>
<td></td>
<td>Contiguous store four-word structures from four vectors (scalar plus immediate, scalar plus scalar)</td>
<td>ST4W (scalar plus immediate) on page C8-4245&lt;br&gt;ST4W (scalar plus scalar) on page C8-4247</td>
</tr>
</tbody>
</table>
When alignment checking is enabled for loads and stores, the computed virtual address of each element must be aligned to the memory element access size.

### Table C3-50 Predicated non-contiguous element accesses

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LD1</strong></td>
<td>Gather load unsigned bytes to vector (scalar plus vector, vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td><strong>LD1B</strong> (scalar plus vector) on page C8-3415</td>
</tr>
<tr>
<td></td>
<td><strong>LD1B</strong> (vector plus immediate) on page C8-3418</td>
</tr>
<tr>
<td></td>
<td>Gather load doublewords to vector (scalar plus vector, vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td><strong>LD1D</strong> (scalar plus vector) on page C8-3424</td>
</tr>
<tr>
<td></td>
<td><strong>LD1D</strong> (vector plus immediate) on page C8-3427</td>
</tr>
<tr>
<td></td>
<td>Gather load unsigned halfwords (scalar plus vector, vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td><strong>LD1H</strong> (scalar plus vector) on page C8-3435</td>
</tr>
<tr>
<td></td>
<td><strong>LD1H</strong> (vector plus immediate) on page C8-3439</td>
</tr>
<tr>
<td></td>
<td>Gather load signed bytes to vector (scalar plus vector, vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td><strong>LD1SB</strong> (scalar plus vector) on page C8-3496</td>
</tr>
<tr>
<td></td>
<td><strong>LD1SB</strong> (vector plus immediate) on page C8-3499</td>
</tr>
<tr>
<td></td>
<td>Gather load signed halfwords to vector (scalar plus vector, vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td><strong>LD1SH</strong> (scalar plus vector) on page C8-3505</td>
</tr>
<tr>
<td></td>
<td><strong>LD1SH</strong> (vector plus immediate) on page C8-3509</td>
</tr>
<tr>
<td></td>
<td>Gather load unsigned words to vector (scalar plus vector, vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td><strong>LD1SW</strong> (scalar plus vector) on page C8-3515</td>
</tr>
<tr>
<td></td>
<td><strong>LD1SW</strong> (vector plus immediate) on page C8-3518</td>
</tr>
<tr>
<td><strong>ST1</strong></td>
<td>Scatter store bytes from a vector (scalar plus vector, vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td><strong>ST1B</strong> (scalar plus vector) on page C8-4167</td>
</tr>
<tr>
<td></td>
<td><strong>ST1B</strong> (vector plus immediate) on page C8-4170</td>
</tr>
<tr>
<td></td>
<td>Scatter store doublewords from a vector (scalar plus vector, vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td><strong>ST1D</strong> (scalar plus vector) on page C8-4176</td>
</tr>
<tr>
<td></td>
<td><strong>ST1D</strong> (vector plus immediate) on page C8-4179</td>
</tr>
<tr>
<td></td>
<td>Scatter store halfwords from a vector (scalar plus vector, vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td><strong>ST1H</strong> (scalar plus vector) on page C8-4185</td>
</tr>
<tr>
<td></td>
<td><strong>ST1H</strong> (vector plus immediate) on page C8-4189</td>
</tr>
<tr>
<td></td>
<td>Scatter store words from a vector (scalar plus vector, vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td><strong>ST1W</strong> (scalar plus vector) on page C8-4195</td>
</tr>
<tr>
<td></td>
<td><strong>ST1W</strong> (vector plus immediate) on page C8-4199</td>
</tr>
<tr>
<td><strong>LDFF1</strong></td>
<td>Gather load first-fault unsigned bytes to vector (scalar plus vector, vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td><strong>LDFF1B</strong> (scalar plus vector) on page C8-3581</td>
</tr>
<tr>
<td></td>
<td><strong>LDFF1B</strong> (vector plus immediate) on page C8-3584</td>
</tr>
<tr>
<td></td>
<td>Gather load first-fault doublewords to vector (scalar plus vector, vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td><strong>LDFF1D</strong> (scalar plus vector) on page C8-3588</td>
</tr>
<tr>
<td></td>
<td><strong>LDFF1D</strong> (vector plus immediate) on page C8-3591</td>
</tr>
<tr>
<td></td>
<td>Gather load first-fault unsigned halfwords to vector (scalar plus vector, vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td><strong>LDFF1H</strong> (scalar plus vector) on page C8-3596</td>
</tr>
<tr>
<td></td>
<td><strong>LDFF1H</strong> (vector plus immediate) on page C8-3617</td>
</tr>
<tr>
<td></td>
<td>Gather load first-fault signed bytes to vector (scalar plus vector, vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td><strong>LDFF1SB</strong> (scalar plus vector) on page C8-3605</td>
</tr>
<tr>
<td></td>
<td><strong>LDFF1SB</strong> (vector plus immediate) on page C8-3608</td>
</tr>
<tr>
<td></td>
<td>Gather load first-fault signed halfwords to vector (scalar plus vector, vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td><strong>LDFF1SH</strong> (scalar plus vector) on page C8-3613</td>
</tr>
<tr>
<td></td>
<td><strong>LDFF1SH</strong> (vector plus immediate) on page C8-3617</td>
</tr>
<tr>
<td></td>
<td>Gather load first-fault signed words to vector (scalar plus vector, vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td><strong>LDFF1SW</strong> (scalar plus vector) on page C8-3621</td>
</tr>
<tr>
<td></td>
<td><strong>LDFF1SW</strong> (vector plus immediate) on page C8-3624</td>
</tr>
</tbody>
</table>
The load and replicate instructions read one or more contiguous memory locations starting from an address that is defined by a scalar base register plus either:

- A scalar index register.
- An immediate byte offset.

The predicated contiguous load and store instructions have two supporting addressing modes:

- Scalar base plus immediate offset.
- Scalar base plus scalar index.

This defaults to zero if omitted. For predicated replicating element load SVE instructions:

- When alignment checking is enabled, the base address must be aligned to the memory element access size.

### Table C3-50 Predicated non-contiguous element accesses (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instructions</th>
<th>See</th>
</tr>
</thead>
</table>
| LDFF1    | Gather load first-fault unsigned words to vector (scalar plus vector, vector plus immediate) | LDFF1W (scalar plus vector) on page C8-3629  
LDFF1W (vector plus immediate) on page C8-3633 |
| PRF      | Gather prefetch bytes (scalar plus vector, vector plus immediate) | PRFB (scalar plus vector) on page C8-3809  
PRFB (vector plus immediate) on page C8-3812 |
|          | Gather prefetch doublewords (scalar plus vector, vector plus immediate) | PRFD (scalar plus vector) on page C8-3818  
PRFD (vector plus immediate) on page C8-3821 |
|          | Gather prefetch halfwords (scalar plus vector, vector plus immediate) | PRFH (scalar plus vector) on page C8-3827  
PRFH (vector plus immediate) on page C8-3830 |
|          | Gather prefetch words (scalar plus vector, vector plus immediate) | PRFW (scalar plus vector) on page C8-3836  
PRFW (vector plus immediate) on page C8-3839 |

### C3.3.4 Predicated replicating element loads

The load and replicate instructions read one or more contiguous memory locations starting from an address that is defined by a scalar base register plus either:

- A scalar index register.
- An immediate byte offset.

The predicated contiguous load and store instructions have two supporting addressing modes:

- Scalar base plus immediate offset.
- Scalar base plus scalar index.

This defaults to zero if omitted. For predicated replicating element load SVE instructions:

- When alignment checking is enabled, the base address must be aligned to the memory element access size.

### Table C3-51 Predicated replicating element loads

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instructions</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD1RB</td>
<td>Load and broadcast unsigned byte to vector</td>
<td>LD1RB on page C8-3441</td>
</tr>
<tr>
<td>LD1RD</td>
<td>Load and broadcast doubleword to vector</td>
<td>LD1RD on page C8-3444</td>
</tr>
<tr>
<td>LD1RH</td>
<td>Load and broadcast unsigned halfword to vector</td>
<td>LD1RH on page C8-3446</td>
</tr>
<tr>
<td>LD1RW</td>
<td>Load and broadcast unsigned word to vector</td>
<td>LD1RW on page C8-3488</td>
</tr>
</tbody>
</table>
| LD1RQB   | Contiguous load and replicate 16 bytes (scalar plus immediate, scalar plus scalar) | LD1RQB (scalar plus immediate) on page C8-3465  
LD1RQB (scalar plus scalar) on page C8-3467 |
| LD1RQD   | Contiguous load and replicate two doublewords (scalar plus immediate, scalar plus scalar) | LD1RQD (scalar plus immediate) on page C8-3469  
LD1RQD (scalar plus scalar) on page C8-3471 |
| LD1RQH   | Contiguous load and replicate eight halfwords (scalar plus immediate, scalar plus scalar) | LD1RQH (scalar plus immediate) on page C8-3473  
LD1RQH (scalar plus scalar) on page C8-3475 |
| LD1RQW   | Contiguous load and replicate four words (scalar plus immediate, scalar plus scalar) | LD1RQW (scalar plus immediate) on page C8-3477  
LD1RQW (scalar plus scalar) on page C8-3479 |
C3.3.5 Unpredicated vector register load/store

The unpredicated vector register load, LDR, and store, STR, instructions transfer a single vector register from or to memory locations that are specified by a scalar base register plus an immediate index value that is in the range -256 to 255, inclusive. The immediate index value defaults to zero if omitted.

The unpredicated vector register load/store instructions have one supporting addressing mode:

- Scalar base plus immediate index.

For the unpredicated vector register load/store SVE instructions:

- The immediate index value is a vector index, not an element index. The immediate index value is multiplied by the current vector register length in bytes.
- The data transfer is performed as a contiguous stream of byte accesses in ascending element order, without endianness conversion.
- When alignment checking is enabled for loads and stores, the base address must be 16-byte aligned.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>Load vector register (vector)</td>
<td>LDR (vector) on page C8-3683</td>
</tr>
<tr>
<td>STR</td>
<td>Store vector register (vector)</td>
<td>STR (vector) on page C8-4274</td>
</tr>
</tbody>
</table>

C3.3.6 Unpredicated predicate register load/store

The unpredicated predicate register load, LDR, and store, STR, instructions transfer a single predicate register from or to memory locations that are specified by a scalar base register plus an immediate index value that is in the range -256 to 255, inclusive. The immediate index value defaults to zero if omitted.

The unpredicated predicate register load/store instructions have one supporting addressing mode:

- Scalar base plus immediate index.

For unpredicated predicate register load/store SVE instructions:

- The immediate index value is a predicate index, not an element index. The immediate index value is multiplied by the current predicate register length, in bytes.
- The data transfer is performed as a contiguous stream of byte accesses, each byte containing 8 consecutive predicate bits, in ascending bit and element order, without endianness conversion.
- When alignment checking is enabled for loads and stores, the base address must be 2-byte aligned.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>Load predicate register (predicate)</td>
<td>LDR (predicate) on page C8-3682</td>
</tr>
<tr>
<td>STR</td>
<td>Store predicate register (predicate)</td>
<td>STR (predicate) on page C8-4273</td>
</tr>
</tbody>
</table>
C3.4 Data processing - immediate

This section describes the instruction groups for data processing with immediate operands. It contains the following subsections:

- Arithmetic (immediate).
- Logical (immediate).
- Move (wide immediate) on page C3-299.
- Move (immediate) on page C3-299.
- PC-relative address calculation on page C3-300.
- Bitfield move on page C3-300.
- Bitfield insert and extract on page C3-301
- Extract register on page C3-301.
- Shift (immediate) on page C3-301.
- Sign-extend and Zero-extend on page C3-302.

For information about the encoding structure of the instructions in this instruction group, see Data Processing -- Immediate on page C4-530.

C3.4.1 Arithmetic (immediate)

The Arithmetic (immediate) instructions accept a 12-bit unsigned immediate value, optionally shifted left by 12 bits. The Arithmetic (immediate) instructions that do not set Condition flags can read from and write to the current stack pointer. The flag setting instructions can read from the stack pointer, but they cannot write to it.

Table C3-54 on page C3-298 shows the Arithmetic instructions with an immediate offset.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add</td>
<td>ADD (immediate) on page C6-1177</td>
</tr>
<tr>
<td>ADDS</td>
<td>Add and set flags</td>
<td>ADDS (immediate) on page C6-1185</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>SUB (immediate) on page C6-1967</td>
</tr>
<tr>
<td>SUBS</td>
<td>Subtract and set flags</td>
<td>SUBS (immediate) on page C6-1977</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>CMP (immediate) on page C6-1278</td>
</tr>
<tr>
<td>CMN</td>
<td>Compare negative</td>
<td>CMN (immediate) on page C6-1272</td>
</tr>
</tbody>
</table>

C3.4.2 Logical (immediate)

The Logical (immediate) instructions accept a bitmask immediate value that is a 32-bit pattern or a 64-bit pattern viewed as a vector of identical elements of size \( e = 2, 4, 8, 16, 32 \) or, 64 bits. Each element contains the same sub-pattern, that is a single run of 1 to \( (e - 1) \) nonzero bits from bit 0 followed by zero bits, then rotated by 0 to \( (e - 1) \) bits. This mechanism can generate 5334 unique 64-bit patterns as 2667 pairs of pattern and their bitwise inverse.

Note: Values that consist of only zeros or only ones cannot be described in this way.

The Logical (immediate) instructions that do not set the Condition flags can write to the current stack pointer, for example to align the stack pointer in a function prologue.
#### Note

Apart from `ANDS` and its `TST` alias, Logical (immediate) instructions do not set the Condition flags. However, the final results of a bitwise operation can be tested by a `CBZ`, `CBNZ`, `TBZ`, or `TBNZ` conditional branch.

Table C3-55 shows the Logical immediate instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Bitwise AND</td>
<td><code>AND (immediate)</code> on page C6-1191</td>
</tr>
<tr>
<td>ANDS</td>
<td>Bitwise AND and set flags</td>
<td><code>ANDS (immediate)</code> on page C6-1195</td>
</tr>
<tr>
<td>EOR</td>
<td>Bitwise exclusive OR</td>
<td><code>EOR (immediate)</code> on page C6-1494</td>
</tr>
<tr>
<td>ORR</td>
<td>Bitwise inclusive OR</td>
<td><code>ORR (immediate)</code> on page C6-1729</td>
</tr>
<tr>
<td>TST</td>
<td>Test bits</td>
<td><code>TST (immediate)</code> on page C6-2009</td>
</tr>
</tbody>
</table>

### C3.4.3 Move (wide immediate)

The Move (wide immediate) instructions insert a 16-bit immediate, or inverted immediate, into a 16-bit aligned position in the destination register. The value of the other bits in the destination register depends on the variant used. The optional shift amount can be any multiple of 16 that is smaller than the register size.

Table C3-56 on page C3-299 shows the Move (wide immediate) instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVZ</td>
<td>Move wide with zero</td>
<td><code>MOVZ</code> on page C6-1705</td>
</tr>
<tr>
<td>MOVN</td>
<td>Move wide with NOT</td>
<td><code>MOVN</code> on page C6-1703</td>
</tr>
<tr>
<td>MOVK</td>
<td>Move wide with keep</td>
<td><code>MOVK</code> on page C6-1701</td>
</tr>
</tbody>
</table>

### C3.4.4 Move (immediate)

The Move (immediate) instructions are aliases for a single `MOVZ`, `MOVN`, or `ORR` (immediate with zero register), instruction to load an immediate value into the destination register. An assembler must permit a signed or unsigned immediate, as long as its binary representation can be generated using one of these instructions, and an assembler error results if the immediate cannot be generated in this way. On disassembly, it is unspecified whether the immediate is output as a signed or an unsigned value.

If there is a choice between the `MOVZ`, `MOVN`, and `ORR` instruction to encode the immediate, then an assembler must prefer `MOVZ` to `MOVN`, and `MOVZ` or `MOVN` to `ORR`, to ensure reversability. A disassembler must output `ORR` (immediate with zero register) `MOVZ`, and `MOVN`, as a `MOV` mnemonic except that the underlying instruction must be used when:

- `ORR` has an immediate that can be generated by a `MOVZ` or `MOVN` instruction.
- A `MOVN` instruction has an immediate that can be encoded by `MOVZ`.
- `MOVZ` #0 or `MOVN` #0 have a shift amount other than `LSL` #0.
Table C3-57 shows the Move (immediate) instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>Move (inverted wide immediate)</td>
<td>MOV (inverted wide immediate) on page C6-1693</td>
</tr>
<tr>
<td>MOV</td>
<td>Move (wide immediate)</td>
<td>MOV (wide immediate) on page C6-1695</td>
</tr>
<tr>
<td>MOV</td>
<td>Move (bitmask immediate)</td>
<td>MOV (bitmask immediate) on page C6-1697</td>
</tr>
</tbody>
</table>

**C3.4.5 PC-relative address calculation**

The `ADR` instruction adds a signed, 21-bit immediate to the value of the program counter that fetched this instruction, and then writes the result to a general-purpose register. This permits the calculation of any byte address within ±1MB of the current PC.

The `ADRP` instruction shifts a signed, 21-bit immediate left by 12 bits, adds it to the value of the program counter with the bottom 12 bits cleared to zero, and then writes the result to a general-purpose register. This permits the calculation of the address at a 4KB aligned memory region. In conjunction with an `A00` (immediate) instruction, or a load/store instruction with a 12-bit immediate offset, this allows for the calculation of, or access to, any address within ±4GB of the current PC.

---

**Note**

The term *page* used in the `ADRP` description is short-hand for the 4KB memory region, and is not related to the virtual memory translation granule size.

Table C3-58 on page C3-300 shows the instructions used for PC-relative address calculations are as follows:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADRP</td>
<td>Compute address of 4KB page at a PC-relative offset</td>
<td>ADRP on page C6-1190</td>
</tr>
<tr>
<td>ADR</td>
<td>Compute address of label at a PC-relative offset</td>
<td>ADR on page C6-1189</td>
</tr>
</tbody>
</table>

**C3.4.6 Bitfield move**

The Bitfield move instructions copy a field of constant width from bit 0 in the source register to a constant bit position in the destination register, or from a constant bit position in the source register to bit 0 in the destination register. The remaining bits in the destination register are set as follows:

- For `BFM`, the remaining bits are unchanged.
- For `UBFM`, the lower bits, if any, and upper bits, if any, are set to zero.
- For `SBFM`, the lower bits, if any, are set to zero, and the upper bits, if any, are set to a copy of the most-significant bit in the copied field.
Table C3-59 shows the Bitfield move instructions.

### Table C3-59 Bitfield move instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFM</td>
<td>Bitfield move</td>
<td>BFM on page C6-1221</td>
</tr>
<tr>
<td>SBFM</td>
<td>Signed bitfield move</td>
<td>SBFM on page C6-1777</td>
</tr>
<tr>
<td>UBFM</td>
<td>Unsigned bitfield move (32-bit)</td>
<td>UBFM on page C6-2014</td>
</tr>
</tbody>
</table>

#### C3.4.7 Bitfield insert and extract

The Bitfield insert and extract instructions are implemented as aliases of the Bitfield move instructions. Table C3-60 on page C3-301 shows the Bitfield insert and extract aliases.

### Table C3-60 Bitfield insert and extract instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFC</td>
<td>Bitfield insert clear</td>
<td>BFC on page C6-1217</td>
</tr>
<tr>
<td>BFI</td>
<td>Bitfield insert</td>
<td>BFI on page C6-1219</td>
</tr>
<tr>
<td>BFXIL</td>
<td>Bitfield extract and insert low</td>
<td>BFXIL on page C6-1223</td>
</tr>
<tr>
<td>SBFIZ</td>
<td>Signed bitfield insert in zero</td>
<td>SBFIZ on page C6-1775</td>
</tr>
<tr>
<td>SBFX</td>
<td>Signed bitfield extract</td>
<td>SBFX on page C6-1780</td>
</tr>
<tr>
<td>UBFIZ</td>
<td>Unsigned bitfield insert in zero</td>
<td>UBFIZ on page C6-2012</td>
</tr>
<tr>
<td>UBFX</td>
<td>Unsigned bitfield extract</td>
<td>UBFX on page C6-2017</td>
</tr>
</tbody>
</table>

#### C3.4.8 Extract register

Depending on the register width of the operands, the Extract register instruction copies a 32-bit or 64-bit field from a constant bit position within a double-width value formed by the concatenation of a pair of source registers to a destination register.

Table C3-61 on page C3-301 shows the Extract (immediate) instructions.

### Table C3-61 Extract register instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTR</td>
<td>Extract register from pair</td>
<td>EXTR on page C6-1501</td>
</tr>
</tbody>
</table>

#### C3.4.9 Shift (Immediate)

Shifts and rotates by a constant amount are implemented as aliases of the Bitfield move or Extract register instructions. The shift or rotate amount must be in the range 0 to one less than the register width of the instruction, inclusive.
Table C3-62 shows the aliases that can be used as immediate shift and rotate instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR</td>
<td>Arithmetic shift right</td>
<td>ASR (immediate) on page C6-1201</td>
</tr>
<tr>
<td>LSL</td>
<td>Logical shift left</td>
<td>LSL (immediate) on page C6-1678</td>
</tr>
<tr>
<td>LSR</td>
<td>Logical shift right</td>
<td>LSR (immediate) on page C6-1684</td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate right</td>
<td>ROR (immediate) on page C6-1764</td>
</tr>
</tbody>
</table>

C3.4.10 Sign-extend and Zero-extend

The Sign-extend and Zero-extend instructions are implemented as aliases of the Bitfield move instructions.

Table C3-63 on page C3-302 shows the aliases that can be used as zero-extend and sign-extend instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SXTB</td>
<td>Sign-extend byte</td>
<td>SXTB on page C6-1988</td>
</tr>
<tr>
<td>SXTH</td>
<td>Sign-extend halfword</td>
<td>SXTH on page C6-1990</td>
</tr>
<tr>
<td>SXTW</td>
<td>Sign-extend word</td>
<td>SXTW on page C6-1992</td>
</tr>
<tr>
<td>UXTB</td>
<td>Unsigned extend byte</td>
<td>UXTB on page C6-2028</td>
</tr>
<tr>
<td>UXTH</td>
<td>Unsigned extend halfword</td>
<td>UXTH on page C6-2029</td>
</tr>
</tbody>
</table>
C3.5 Data processing - register

This section describes the instruction groups for data processing with all register operands. It contains the following subsections:

- Arithmetic (shifted register).
- Arithmetic (extended register) on page C3-304.
- Arithmetic with carry on page C3-305.
- Flag manipulation instructions on page C3-305.
- Logical (shifted register) on page C3-305.
- Move (register) on page C3-306.
- Shift (register) on page C3-306.
- Multiply and divide on page C3-307.
- CRC32 on page C3-308.
- Bit operation on page C3-309.
- Conditional select on page C3-309.
- Conditional comparison on page C3-310.

For information about the encoding structure of the instructions in this instruction group, see Data Processing -- Register on page C4-590.

C3.5.1 Arithmetic (shifted register)

The Arithmetic (shifted register) instructions apply an optional shift operator to the second source register value before performing the arithmetic operation. The register width of the instruction controls whether the new bits are fed into the intermediate result on a right shift or rotate at bit[63] or bit[31].

The shift operators LSL, ASR, and LSR accept an immediate shift amount in the range 0 to one less than the register width of the instruction, inclusive.

Omitting the shift operator implies LSL #0, which means that there is no shift. A disassembler must not output LSL #0. However, a disassembler must output all other shifts by zero.

The current stack pointer, SP or WSP, cannot be used with this class of instructions. See Arithmetic (extended register) on page C3-304 for arithmetic instructions that can operate on the current stack pointer.

Table C3-64 on page C3-303 shows the Arithmetic (shifted register) instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add</td>
<td>ADD (shifted register) on page C6-1179</td>
</tr>
<tr>
<td>ADDS</td>
<td>Add and set flags</td>
<td>ADDS (shifted register) on page C6-1187</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>SUB (shifted register) on page C6-1969</td>
</tr>
<tr>
<td>SUBS</td>
<td>Subtract and set flags</td>
<td>SUBS (shifted register) on page C6-1979</td>
</tr>
<tr>
<td>CMN</td>
<td>Compare</td>
<td>CMN (shifted register) on page C6-1274</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>CMP (shifted register) on page C6-1280</td>
</tr>
<tr>
<td>NEG</td>
<td>Negate</td>
<td>NEG (shifted register) on page C6-1718</td>
</tr>
<tr>
<td>NEGS</td>
<td>Negate and set flags</td>
<td>NEGS on page C6-1720</td>
</tr>
</tbody>
</table>
C3.5.2 Arithmetic (extended register)

The extended register instructions provide an optional sign-extension or zero-extension of a portion of the second source register value, followed by an optional left shift by a constant amount of 1-4, inclusive.

The extended shift is described by the mandatory extend operator SXTB, SXTW, UXTB, UXTH, or UXTW. This is followed by an optional left shift amount. If the shift amount is not specified, the default shift amount is zero. A disassembler must not output a shift amount of zero.

For 64-bit instruction forms, the additional operators UXTX and SXTX use all 64 bits of the second source register with an optional shift. In that case, Arm recommends UXTX as the operator. If and only if at least one register is SP, Arm recommends use of the LSL operator name, rather than UXTX, and when the shift amount is also zero then both the operator and the shift amount can be omitted. UXTW and SXTW both use all 32 bits of the second source register with an optional shift. In that case Arm recommends UXTW as the operator. If and only if at least one register is WSP, Arm recommends use of the LSL operator name, rather than UXTW, and when the shift amount is also zero then both the operator and the shift amount can be omitted.

For 32-bit instruction forms, the operators UXTW and SXTW both use all 32 bits of the second source register with an optional shift. In that case, Arm recommends UXTW as the operator. If and only if at least one register is WSP, Arm recommends use of the LSL operator name, rather than UXTW, and when the shift amount is also zero then both the operator and the shift amount can be omitted.

The non-flag setting variants of the extended register instruction permit the use of the current stack pointer as either the destination register and the first source register. The flag setting variants only permit the stack pointer to be used as the first source register.

In the 64-bit form of these instructions, the final register operand is written as \( m \) for all except the UXTX/LSL and SXTX extend operators. For example:

```
CMP X4, W5, SXTW
ADD X1, X2, W3, UXTB #2
SUB SP, SP, X1 // SUB SP, SP, X1, UXTX #0
```

Table C3-65 on page C3-304 shows the Arithmetic (extended register) instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add</td>
<td>ADD (extended register) on page C6-1174</td>
</tr>
<tr>
<td>ADDS</td>
<td>Add and set flags</td>
<td>ADDS (extended register) on page C6-1182</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>SUB (extended register) on page C6-1964</td>
</tr>
<tr>
<td>SUBS</td>
<td>Subtract and set flags</td>
<td>SUBS (extended register) on page C6-1974</td>
</tr>
<tr>
<td>CMN</td>
<td>Compare negative</td>
<td>CMN (extended register) on page C6-1270</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>CMP (extended register) on page C6-1276</td>
</tr>
</tbody>
</table>
C3.5.3 Arithmetic with carry

The Arithmetic with carry instructions accept two source registers, with the carry flag as an additional input to the calculation. They do not support shifting of the second source register.

Table C3-66 on page C3-305 shows the Arithmetic with carry instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Add with carry</td>
<td>ADC on page C6-1170</td>
</tr>
<tr>
<td>ADCS</td>
<td>Add with carry and set flags</td>
<td>ADCS on page C6-1172</td>
</tr>
<tr>
<td>SBC</td>
<td>Subtract with carry</td>
<td>SBC on page C6-1771</td>
</tr>
<tr>
<td>SBCS</td>
<td>Subtract with carry and set flags</td>
<td>SBCS on page C6-1773</td>
</tr>
<tr>
<td>NGC</td>
<td>Negate with carry</td>
<td>NGC on page C6-1722</td>
</tr>
<tr>
<td>NGCS</td>
<td>Negate with carry and set flags</td>
<td>NGCS on page C6-1724</td>
</tr>
</tbody>
</table>

C3.5.4 Flag manipulation instructions

The Flag manipulation instructions set the value of the NZCV condition flags directly.

The instructions SETF8 and SETF16 accept one source register and set the NZV condition flags based on the value of the input register. The instruction RMIF accepts one source register and two immediate values, rotating the first source register using the first immediate value and setting the NZCV condition flags masked by the second immediate value.

The instructions XAFLAG and AXFLAG convert PSTATE condition flags between the FCMP instruction format and an alternative format. See Table C6-1 on page C6-1168 for more information.

Table C3-67 on page C3-305 shows the Flag manipulation instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXFLAG</td>
<td>Convert from FCMP comparison format to the alternative format</td>
<td>AXFLAG on page C6-1213</td>
</tr>
<tr>
<td>CFINV</td>
<td>Invert value of the PSTATE.C bit</td>
<td>CFINV on page C6-1260</td>
</tr>
<tr>
<td>RMIF</td>
<td>Rotate, mask insert flags</td>
<td>RMIF on page C6-1763</td>
</tr>
<tr>
<td>SETF8</td>
<td>Evaluation of 8-bit flags</td>
<td>SETF8, SETF16 on page C6-1783</td>
</tr>
<tr>
<td>SETF16</td>
<td>Evaluation of 16-bit flags</td>
<td>SETF8, SETF16 on page C6-1783</td>
</tr>
<tr>
<td>XAFLAG</td>
<td>Convert from alternative format to FCMP comparison format</td>
<td>XAFLAG on page C6-2034</td>
</tr>
</tbody>
</table>

C3.5.5 Logical (shifted register)

The Logical (shifted register) instructions apply an optional shift operator to the second source register value before performing the main operation. The register width of the instruction controls whether the new bits are fed into the intermediate result on a right shift or rotate at bit[63] or bit[31].

The shift operators LSL, ASR, LSR, and ROR accept a constant immediate shift amount in the range 0 to one less than the register width of the instruction, inclusive.

Omitting the shift operator and amount implies LSL #0, which means that there is no shift. A disassembler must not output LSL #0. However, a disassembler must output all other shifts by zero.
Note
Apart from ANDS, TST, and BICS, the logical instructions do not set the Condition flags, but the final result of a bit operation can usually directly control a CBZ, CBNZ, TBZ, or TBNZ conditional branch.

Table C3-68 shows the Logical (shifted register) instructions.

### Table C3-68 Logical (shifted register) instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Bitwise AND</td>
<td>AND (shifted register) on page C6-1193</td>
</tr>
<tr>
<td>ANDS</td>
<td>Bitwise AND and set flags</td>
<td>ANDS (shifted register) on page C6-1197</td>
</tr>
<tr>
<td>BIC</td>
<td>Bitwise bit clear</td>
<td>BIC (shifted register) on page C6-1225</td>
</tr>
<tr>
<td>BICS</td>
<td>Bitwise bit clear and set flags</td>
<td>BICS (shifted register) on page C6-1227</td>
</tr>
<tr>
<td>EON</td>
<td>Bitwise exclusive OR NOT</td>
<td>EON (shifted register) on page C6-1492</td>
</tr>
<tr>
<td>EOR</td>
<td>Bitwise exclusive OR</td>
<td>EOR (shifted register) on page C6-1496</td>
</tr>
<tr>
<td>ORR</td>
<td>Bitwise inclusive OR</td>
<td>ORR (shifted register) on page C6-1731</td>
</tr>
<tr>
<td>MVN</td>
<td>Bitwise NOT</td>
<td>MVN on page C6-1716</td>
</tr>
<tr>
<td>ORN</td>
<td>Bitwise inclusive OR NOT</td>
<td>ORN (shifted register) on page C6-1727</td>
</tr>
<tr>
<td>TST</td>
<td>Test bits</td>
<td>TST (shifted register) on page C6-2010</td>
</tr>
</tbody>
</table>

### C3.5.6 Move (register)

The Move (register) instructions are aliases for other data processing instructions. They copy a value from a general-purpose register to another general-purpose register or the current stack pointer, or from the current stack pointer to a general-purpose register.

### Table C3-69 MOV register instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>Move register</td>
<td>MOV (register) on page C6-1699</td>
</tr>
<tr>
<td></td>
<td>Move register to SP or move SP to register</td>
<td>MOV (to/from SP) on page C6-1692</td>
</tr>
</tbody>
</table>

### C3.5.7 Shift (register)

In the Shift (register) instructions, the shift amount is the positive value in the second source register modulo the register size. The register width of the instruction controls whether the new bits are fed into the result on a right shift or rotate at bit[63] or bit[31].
C3.5 Data processing - register

Table C3-70 shows the Shift (register) instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASRV</td>
<td>Arithmetic shift right variable</td>
<td>ASRV on page C6-1203</td>
</tr>
<tr>
<td>LSLV</td>
<td>Logical shift left variable</td>
<td>LSLV on page C6-1680</td>
</tr>
<tr>
<td>LSRV</td>
<td>Logical shift right variable</td>
<td>LSRV on page C6-1686</td>
</tr>
<tr>
<td>RORV</td>
<td>Rotate right variable</td>
<td>RORV on page C6-1768</td>
</tr>
</tbody>
</table>

However, the Shift (register) instructions have a preferred set of aliases that match the shift immediate aliases described in *[Shift (immediate)]* on page C3-301.

Table C3-71 shows the aliases for Shift (register) instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR</td>
<td>Arithmetic shift right</td>
<td>ASR (register) on page C6-1199</td>
</tr>
<tr>
<td>LSL</td>
<td>Logical shift left</td>
<td>LSL (register) on page C6-1676</td>
</tr>
<tr>
<td>LSR</td>
<td>Logical shift right</td>
<td>LSR (register) on page C6-1682</td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate right</td>
<td>ROR (register) on page C6-1766</td>
</tr>
</tbody>
</table>

C3.5.8 Multiply and divide

This section describes the instructions used for integer multiplication and division. It contains the following subsections:

- *Multiply*.
- *Divide* on page C3-308.

### Multiply

The Multiply instructions write to a single 32-bit or 64-bit destination register, and are built around the fundamental four operand multiply-add and multiply-subtract operation, together with 32-bit to 64-bit widening variants. A 64-bit to 128-bit widening multiple can be constructed with two instructions, using SMULH or UMULH to generate the upper 64 bits. Table C3-72 on page C3-307 shows the Multiply instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>MADD</td>
<td>Multiply-add</td>
<td>MADD on page C6-1688</td>
</tr>
<tr>
<td>MSUB</td>
<td>Multiply-subtract</td>
<td>MSUB on page C6-1713</td>
</tr>
<tr>
<td>MNEG</td>
<td>Multiply-negate</td>
<td>MNEG on page C6-1690</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td>MUL on page C6-1715</td>
</tr>
<tr>
<td>SMADDL</td>
<td>Signed multiply-add long</td>
<td>SMADDL on page C6-1822</td>
</tr>
<tr>
<td>SMSUBL</td>
<td>Signed multiply-subtract long</td>
<td>SMSUBL on page C6-1830</td>
</tr>
</tbody>
</table>
Divide

The Divide instructions compute the quotient of a division, rounded towards zero. The remainder can then be computed as (numerator - (quotient × denominator)), using the MSUB instruction.

If a signed integer division (INT_MIN / -1) is performed where INT_MIN is the most negative integer value representable in the selected register size, then the result overflows the signed integer range. No indication of this overflow is produced and the result that is written to the destination register is INT_MIN.

A division by zero results in a zero being written to the destination register, without any indication that the division by zero occurred.

Table C3-73 shows the Divide instructions.

Table C3-73  Divide instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDIV</td>
<td>Signed divide</td>
<td>SDIV on page C6-1782</td>
</tr>
<tr>
<td>UDIV</td>
<td>Unsigned divide</td>
<td>UDIV on page C6-2020</td>
</tr>
</tbody>
</table>

C3.5.9  CRC32

The CRC32 instructions operate on the general-purpose register file to update a 32-bit CRC value from an input value comprising 1, 2, 4, or 8 bytes. There are two different classes of CRC instructions, CRC32, and CRC32C, that support two commonly used 32-bit polynomials, known as CRC-32 and CRC-32C.

To fit with common usage, the bit order of the values is reversed as part of the operation.

When bits[19:16] of ID_AA64ISAR0_EL1 are set to 0b0001, the CRC instructions are implemented.

These instructions are optional in an Armv8.0 implementation.

All implementations of Armv8.1 architecture and later are required to implement the CRC32 instructions.
Table C3-74 on page C3-309 shows the CRC instructions.

### Table C3-74  CRC32 instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC32B</td>
<td>CRC-32 sum from byte</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[CRC32B, CRC32H, CRC32W, CRC32X on page C6-1462]</td>
</tr>
<tr>
<td>CRC32H</td>
<td>CRC-32 sum from halfword</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[CRC32B, CRC32H, CRC32W, CRC32X on page C6-1462]</td>
</tr>
<tr>
<td>CRC32W</td>
<td>CRC-32 sum from word</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[CRC32B, CRC32H, CRC32W, CRC32X on page C6-1462]</td>
</tr>
<tr>
<td>CRC32X</td>
<td>CRC-32 sum from doubleword</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[CRC32B, CRC32H, CRC32W, CRC32X on page C6-1462]</td>
</tr>
<tr>
<td>CRC32CB</td>
<td>CRC-32C sum from byte</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[CRC32CB, CRC32CH, CRC32CW, CRC32CX on page C6-1464]</td>
</tr>
<tr>
<td>CRC32CH</td>
<td>CRC-32C sum from halfword</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[CRC32CB, CRC32CH, CRC32CW, CRC32CX on page C6-1464]</td>
</tr>
<tr>
<td>CRC32CW</td>
<td>CRC-32C sum from word</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[CRC32CB, CRC32CH, CRC32CW, CRC32CX on page C6-1464]</td>
</tr>
<tr>
<td>CRC32CX</td>
<td>CRC-32C sum from doubleword</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[CRC32CB, CRC32CH, CRC32CW, CRC32CX on page C6-1464]</td>
</tr>
</tbody>
</table>

### C3.5.10  Bit operation

Table C3-75 on page C3-309 shows the Bit operation instructions.

### Table C3-75 Bit operation instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLS</td>
<td>Count leading sign bits</td>
<td>CLS on page C6-1267</td>
</tr>
<tr>
<td>CLZ</td>
<td>Count leading zero bits</td>
<td>CLZ on page C6-1269</td>
</tr>
<tr>
<td>RBIT</td>
<td>Reverse bit order</td>
<td>RBIT on page C6-1752</td>
</tr>
<tr>
<td>REV</td>
<td>Reverse bytes in register</td>
<td>REV on page C6-1756</td>
</tr>
<tr>
<td>REV16</td>
<td>Reverse bytes in halfwords</td>
<td>REV16 on page C6-1758</td>
</tr>
<tr>
<td>REV32</td>
<td>Reverse bytes in words</td>
<td>REV32 on page C6-1760</td>
</tr>
<tr>
<td>REV64</td>
<td>Reverse bytes in register</td>
<td>REV64 on page C6-1762</td>
</tr>
</tbody>
</table>

### C3.5.11  Conditional select

The Conditional select instructions select between the first or second source register, depending on the current state of the Condition flags. When the named condition is true, the first source register is selected and its value is copied without modification to the destination register. When the condition is false the second source register is selected and its value might be optionally inverted, negated, or incremented by one, before writing to the destination register.

Other useful conditional set and conditional unary operations are implemented as aliases of the four Conditional select instructions.
Table C3-76 shows the Conditional select instructions.

**Table C3-76**  
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSEL</td>
<td>Conditional select</td>
<td>CSEL on page C6-1467</td>
</tr>
<tr>
<td>CSINC</td>
<td>Conditional select</td>
<td>CSINC on page C6-1473</td>
</tr>
<tr>
<td>CINV</td>
<td>Conditional select</td>
<td>CINV on page C6-1475</td>
</tr>
<tr>
<td>CSNEG</td>
<td>Conditional select</td>
<td>CSNEG on page C6-1477</td>
</tr>
<tr>
<td>CSET</td>
<td>Conditional set</td>
<td>CSET on page C6-1469</td>
</tr>
<tr>
<td>CSETM</td>
<td>Conditional set mask</td>
<td>CSETM on page C6-1471</td>
</tr>
<tr>
<td>CINC</td>
<td>Conditional increment</td>
<td>CINC on page C6-1262</td>
</tr>
<tr>
<td>CINV</td>
<td>Conditional invert</td>
<td>CINV on page C6-1264</td>
</tr>
<tr>
<td>CNEG</td>
<td>Conditional negate</td>
<td>CNEG on page C6-1283</td>
</tr>
</tbody>
</table>

**C3.5.12  Conditional comparison**

The Conditional comparison instructions provide a conditional select for the NZCV Condition flags, setting the flags to the result of an arithmetic comparison of its two source register values if the named input condition is true, or to an immediate value if the input condition is false. There are register and immediate forms. The immediate form compares the source register to a small 5-bit unsigned value.

Table C3-77 on page C3-310 shows the Conditional comparison instructions.

**Table C3-77**  
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCMN</td>
<td>Conditional compare negative (register)</td>
<td>CCMN (register) on page C6-1254</td>
</tr>
<tr>
<td>CCMN</td>
<td>Conditional compare negative (immediate)</td>
<td>CCMN (immediate) on page C6-1252</td>
</tr>
<tr>
<td>CCMP</td>
<td>Conditional compare (register)</td>
<td>CCMP (register) on page C6-1258</td>
</tr>
<tr>
<td>CCMP</td>
<td>Conditional compare (immediate)</td>
<td>CCMP (immediate) on page C6-1256</td>
</tr>
</tbody>
</table>
C3.6 Data processing - SIMD and floating-point

This section describes the instruction groups for data processing with SIMD and floating-point register operands.

Common features of SIMD instructions gives general information about SIMD instructions.

The following subsections describe the scalar floating-point data processing instructions:
- Floating-point move (register) on page C3-312.
- Floating-point move (immediate) on page C3-312.
- Floating-point conversion on page C3-313.
- Floating-point round to integral value on page C3-314.
- Floating-point multiply-add on page C3-316.
- Floating-point arithmetic (one source) on page C3-316.
- Floating-point arithmetic (two sources) on page C3-316.
- Floating-point minimum and maximum on page C3-317.
- Floating-point comparison on page C3-317.
- Floating-point conditional select on page C3-318.

The following subsections describe the SIMD data processing instructions:
- SIMD move on page C3-318
- SIMD arithmetic on page C3-318.
- SIMD compare on page C3-322.
- SIMD widening and narrowing arithmetic on page C3-322.
- SIMD table lookup on page C3-331.
- SIMD by element arithmetic on page C3-325.
- SIMD permute on page C3-327.
- SIMD immediate on page C3-327.
- SIMD shift (immediate) on page C3-327.
- SIMD floating-point and integer conversion on page C3-328.
- SIMD reduce (across vector lanes) on page C3-329.
- SIMD pairwise arithmetic on page C3-330.
- SIMD integer dot product on page C3-331.
- SIMD table lookup on page C3-331.
- SIMD complex number arithmetic on page C3-332.
- SIMD BFloat16 floating-point multiply-add on page C3-332.
- SIMD BFloat16 on page C3-332.
- SIMD integer matrix multiply-accumulate on page C3-334.
- The Cryptographic Extension on page C3-334.

For information about the encoding structure of the instructions in this instruction group, see Data Processing -- Scalar Floating-Point and Advanced SIMD on page C4-601.

For information about the floating-point exceptions, see Floating-point exceptions and exception traps on page A1-64.

C3.6.1 Common features of SIMD instructions

A number of SIMD instructions come in three forms:

**Wide**
Indicated by the suffix W. The element width of the destination register and the first source operand is double that of the second source operand.

**Long**
Indicated by the suffix L. The element width of the destination register is double that of both source operands.

**Narrow**
Indicated by the suffix N. The element width of the destination register is half that of both source operands.
In addition, each vector form of the instruction is part of a pair, with a second and upper half suffix of 2, to identify the variant of the instruction:

- Where a SIMD operation widens or lengthens a 64-bit vector to a 128-bit vector, the instruction provides a second part operation that can extract the source from the upper 64 bits of the source registers.
- Where a SIMD operation narrows a 128-bit vector to a 64-bit vector, the instruction provides a second-part operation that can pack the result of a second operation into the upper part of the same destination register.

Note

This is referred to as a lane set specifier.

### C3.6.2 Floating-point move (register)

The Floating-point move (register) instructions copy a scalar floating-point value from one register to another register without performing any conversion.

Some of the Floating-point move (register) instructions overlap with the functionality provided by the Advanced SIMD instructions DUP, INS, and UMOV. However, Arm recommends using the FMOV instructions when operating on scalar floating-point data to avoid the creation of scalar floating-point code that depends on the availability of the Advanced SIMD instruction set.

Table C3-78 on page C3-312 shows the Floating-point move (register) instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMOV</td>
<td>Floating-point move register without conversion</td>
<td>FMOV (register) on page C7-2334</td>
</tr>
<tr>
<td></td>
<td>Floating-point move to or from general-purpose register without conversion</td>
<td>FMOV (general) on page C7-2336</td>
</tr>
</tbody>
</table>

### C3.6.3 Floating-point move (immediate)

The Floating-point move (immediate) instructions convert a small constant immediate floating-point value into a half-precision, single-precision, or double-precision scalar floating-point value in a SIMD and floating-point register.

The floating-point constant can be specified either in decimal notation, such as 12.0 or -1.2e1, or as a string beginning with 0x followed by a hexadecimal representation of the IEEE 754 half-precision, single-precision, or double-precision encoding. Arm recommends that a disassembler uses the decimal notation, provided that this displays the value precisely.

Note

When FEAT_FP16 is not implemented, the only half-precision instructions that are supported are floating-point conversions between half-precision, single-precision, and double-precision.

The floating-point value must be expressible as \((\pm n/16 \times 2^r)\), where \(n\) is an integer in the range \(16 \leq n \leq 31\) and \(r\) is an integer in the range of \(-3 \leq r \leq 4\), that is a normalized binary floating-point encoding with one sign bit, four bits of fraction, and a 3-bit exponent.

Table C3-79 on page C3-312 shows the Floating-point move (immediate) instruction:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMOV</td>
<td>Floating-point immediate</td>
<td>FMOV (scalar, immediate) on page C7-2340</td>
</tr>
</tbody>
</table>
C3.6.4 Floating-point conversion

The following subsections describe the conversion of floating-point values:

- Convert floating-point precision.
- Convert between floating-point and integer or fixed-point.

Convert floating-point precision

These instructions convert a floating-point scalar with one precision to a floating-point scalar with a different precision, using the current rounding mode as specified by FPCR.RMode.

Table C3-80 on page C3-313 shows the Floating-point precision conversion instruction.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCVT</td>
<td>Floating-point convert precision (scalar)</td>
<td>FCVT on page C7-2196</td>
</tr>
</tbody>
</table>

Convert floating-point single-precision to BFloat16

The BFCVT instruction is provided by FEAT_BF16. This instruction converts a single-precision floating-point input to BFloat16 format, honoring the FPCR rounding mode controls to give a more accurate conversion than simply removing the bottom 16 bits of the input.

Table C3-81 on page C3-313 shows this instruction.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFCVT</td>
<td>BFloat16 floating-point convert from single-precision to BFloat16 format (scalar)</td>
<td>BFCVT on page C7-2061</td>
</tr>
</tbody>
</table>

Convert between floating-point and integer or fixed-point

These instructions convert a floating-point scalar in a SIMD and floating-point register to or from a signed or unsigned integer or fixed-point value in a general-purpose register. For a fixed-point value, the final immediate operand indicates that the general-purpose register holds a fixed-point number and fbits indicates the number of bits after the binary point. fbits is in the range 1-32 inclusive for a 32-bit general-purpose register name, and 1-64 inclusive for a 64-bit general-purpose register name.

These instructions can cause the following floating-point exceptions:

Invalid Operation

Occurs if the floating-point input is a NaN, infinity, or a numerical value that cannot be represented in the destination register. An out of range integer or fixed-point result is saturated to the size of the destination register.

Inexact

Occurs if the numeric result that differs from the input value.

Input Denormal

Can occur when zero replaces a double-precision or single-precision denormal input, see Flushing denormalized numbers to zero on page A1-58 and Input Denormal exceptions on page A1-65.
Table C3-82 on page C3-314 shows the Floating-point and fixed-point conversion instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCVTAS</td>
<td>Floating-point scalar convert to signed integer, rounding to nearest with ties to away (scalar form)</td>
<td>FCVTAS (scalar) on page C7-2201</td>
</tr>
<tr>
<td>FCVTAU</td>
<td>Floating-point scalar convert to unsigned integer, rounding to nearest with ties to away (scalar form)</td>
<td>FCVTAU (scalar) on page C7-2206</td>
</tr>
<tr>
<td>FCVTMS</td>
<td>Floating-point scalar convert to signed integer, rounding toward minus infinity (scalar form)</td>
<td>FCVTMS (scalar) on page C7-2213</td>
</tr>
<tr>
<td>FCVTMU</td>
<td>Floating-point scalar convert to unsigned integer, rounding toward minus infinity (scalar form)</td>
<td>FCVTMU (scalar) on page C7-2218</td>
</tr>
<tr>
<td>FCVTNS</td>
<td>Floating-point scalar convert to signed integer, rounding to nearest with ties to even (scalar form)</td>
<td>FCVTNS (scalar) on page C7-2225</td>
</tr>
<tr>
<td>FCVTNU</td>
<td>Floating-point scalar convert to unsigned integer, rounding to nearest with ties to even (scalar form)</td>
<td>FCVTNU (scalar) on page C7-2230</td>
</tr>
<tr>
<td>FCVTPS</td>
<td>Floating-point scalar convert to signed integer, rounding toward positive infinity (scalar form)</td>
<td>FCVTPS (scalar) on page C7-2235</td>
</tr>
<tr>
<td>FCVTPU</td>
<td>Floating-point scalar convert to unsigned integer, rounding toward positive infinity (scalar form)</td>
<td>FCVTPU (scalar) on page C7-2240</td>
</tr>
<tr>
<td>FCVTZS</td>
<td>Floating-point scalar convert to signed integer, rounding toward zero (scalar form)</td>
<td>FCVTZS (scalar; integer) on page C7-2253</td>
</tr>
<tr>
<td></td>
<td>Floating-point convert to signed fixed-point, rounding toward zero (scalar form)</td>
<td>FCVTZS (scalar; fixed-point) on page C7-2251</td>
</tr>
<tr>
<td>FCVTZU</td>
<td>Floating-point scalar convert to unsigned integer, rounding toward zero (scalar form)</td>
<td>FCVTZU (scalar; integer) on page C7-2263</td>
</tr>
<tr>
<td></td>
<td>Floating-point scalar convert to unsigned fixed-point, rounding toward zero (scalar form)</td>
<td>FCVTZU (scalar; fixed-point) on page C7-2261</td>
</tr>
<tr>
<td>FJCVTZS</td>
<td>Floating-point Javascript convert to signed fixed-point, rounding toward zero</td>
<td>FJCVTZS on page C7-2269</td>
</tr>
<tr>
<td>SCVTF</td>
<td>Signed integer scalar convert to floating-point, using the current rounding mode (scalar form)</td>
<td>SCVTF (scalar; integer) on page C7-2580</td>
</tr>
<tr>
<td></td>
<td>Signed fixed-point convert to floating-point, using the current rounding mode (scalar form)</td>
<td>SCVTF (scalar; fixed-point) on page C7-2578</td>
</tr>
<tr>
<td>UCVTF</td>
<td>Unsigned integer scalar convert to floating-point, using the current rounding mode (scalar form)</td>
<td>UCVTF (scalar; integer) on page C7-2863</td>
</tr>
<tr>
<td></td>
<td>Unsigned fixed-point convert to floating-point, using the current rounding mode (scalar form)</td>
<td>UCVTF (scalar; fixed-point) on page C7-2861</td>
</tr>
</tbody>
</table>

### C3.6.5 Floating-point round to integral value

The following subsections describe instructions which round a floating-point number to an integral valued floating-point number in the same format:

- *Floating-point round to an integer of the same size as the register* on page C3-315
- *Floating-point round to 32-bit or 64-bit integer* on page C3-315
Floating-point round to an integer of the same size as the register

The following instructions round a floating-point value to an integer floating-point value of the same size.

For these instructions:
- A zero input gives a zero result with the same sign.
- An infinite input gives an infinite result with the same sign.
- A NaN is propagated as in normal floating-point arithmetic.

These instructions can cause the following floating-point exceptions:

**Invalid Operation**
- Occurs in response to a floating-point input of a signaling NaN.

**Inexact, FRINTX instruction only**
- Occurs if the result is numeric and does not have the same numerical value as the input.

**Input Denormal**
- Can occur when zero replaces a double-precision or single-precision denormal input, see *Flushing denormalized numbers to zero* on page A1-58 and *Input Denormal exceptions* on page A1-65.

Table C3-83 on page C3-315 shows the Floating-point round to integer instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRINTA</td>
<td>Floating-point round to integer, to nearest with ties to away</td>
<td>FRINTA (scalar) on page C7-2395</td>
</tr>
<tr>
<td>FRINTI</td>
<td>Floating-point round to integer, using current rounding mode</td>
<td>FRINTI (scalar) on page C7-2399</td>
</tr>
<tr>
<td>FRINTM</td>
<td>Floating-point round to integer, toward minus infinity</td>
<td>FRINTM (scalar) on page C7-2403</td>
</tr>
<tr>
<td>FRINTN</td>
<td>Floating-point round to integer, to nearest with ties to even</td>
<td>FRINTN (scalar) on page C7-2407</td>
</tr>
<tr>
<td>FRINTP</td>
<td>Floating-point round to integer, toward positive infinity</td>
<td>FRINTP (scalar) on page C7-2411</td>
</tr>
<tr>
<td>FRINTX</td>
<td>Floating-point round to integer exact, using current rounding mode</td>
<td>FRINTX (scalar) on page C7-2415</td>
</tr>
<tr>
<td>FRINTZ</td>
<td>Floating-point round to integer, toward zero</td>
<td>FRINTZ (scalar) on page C7-2419</td>
</tr>
</tbody>
</table>

Floating-point round to 32-bit or 64-bit integer

The following instructions are present if FEAT_FRINTTS is implemented. The instructions round to a value that fits in a 32-bit integer or a 64-bit integer size, and use either round towards zero or the ambient rounding model.

**Invalid Operation**
- Forced to be the most negative integer representable in the target size, and occurs in response to a floating-point input of a signaling NaN, an infinite input, or an out of range input.

**Inexact**
- Occurs if the result is numeric and does not have the same numerical value as the input.

**Input Denormal**
- Can occur when zero replaces a double-precision or single-precision denormal input, see *Flushing denormalized numbers to zero* on page A1-58 and *Input Denormal exceptions* on page A1-65.
Table C3-84 on page C3-316 shows the Floating-point round to 32-bit or 64-bit integer instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRINT32X</td>
<td>Floating-point round to 32-bit integer, using current rounding model</td>
<td>FRINT32X (scalar) on page C7-2379</td>
</tr>
<tr>
<td>FRINT32Z</td>
<td>Floating-point round to 32-bit integer, toward zero</td>
<td>FRINT32Z (scalar) on page C7-2383</td>
</tr>
<tr>
<td>FRINT64X</td>
<td>Floating point round to 64-bit integer using current rounding model</td>
<td>FRINT64X (scalar) on page C7-2387</td>
</tr>
<tr>
<td>FRINT64Z</td>
<td>Floating point round to 64-bit integer, toward zero</td>
<td>FRINT64Z (scalar) on page C7-2391</td>
</tr>
</tbody>
</table>

C3.6.6 Floating-point multiply-add

Table C3-85 on page C3-316 shows the Floating-point multiply-add instructions that require three source register operands.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMADD</td>
<td>Floating-point scalar fused multiply-add</td>
<td>FMADD on page C7-2270</td>
</tr>
<tr>
<td>FMSUB</td>
<td>Floating-point scalar fused multiply-subtract</td>
<td>FMSUB on page C7-2342</td>
</tr>
<tr>
<td>FNMADD</td>
<td>Floating-point scalar negated fused multiply-add</td>
<td>FNMADD on page C7-2363</td>
</tr>
<tr>
<td>FNMSUB</td>
<td>Floating-point scalar negated fused multiply-subtract</td>
<td>FNMSUB on page C7-2365</td>
</tr>
</tbody>
</table>

C3.6.7 Floating-point arithmetic (one source)

Table C3-86 on page C3-316 shows the Floating-point arithmetic instructions that require a single source register operand.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instructions</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FABS</td>
<td>Floating-point scalar absolute value</td>
<td>FABS (scalar) on page C7-2133</td>
</tr>
<tr>
<td>FNEG</td>
<td>Floating-point scalar negate</td>
<td>FNEG (scalar) on page C7-2361</td>
</tr>
<tr>
<td>FSQRT</td>
<td>Floating-point scalar square root</td>
<td>FSQRT (scalar) on page C7-2429</td>
</tr>
</tbody>
</table>

C3.6.8 Floating-point arithmetic (two sources)

Table C3-87 on page C3-316 shows the Floating-point arithmetic instructions that require two source register operands.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD</td>
<td>Floating-point scalar add</td>
<td>FADD (scalar) on page C7-2145</td>
</tr>
<tr>
<td>FDIV</td>
<td>Floating-point scalar divide</td>
<td>FDIV (scalar) on page C7-2267</td>
</tr>
</tbody>
</table>
C3.6.9 Floating-point minimum and maximum

The $\min(x,y)$ and $\max(x,y)$ operations return a quiet NaN when either $x$ or $y$ is NaN.

As described in Flushing denormalized numbers to zero on page A1-58, if flushing denormalized inputs to zero is enabled, denormal operands are flushed to zero before comparison, and if the result of the comparison is the flushed value, then a zero value is returned. Where both $x$ and $y$ are zero, or denormal values flushed to zero, with different signs, then +0.0 is returned by $\max()$ and -0.0 by $\min()$.

The $\minNum(x,y)$ and $\maxNum(x,y)$ operations follow the IEEE 754-2008 standard and return the numerical operand when one operand is numerical and the other a quiet NaN. Apart from this additional handling of a single quiet NaN, the result is then identical to $\min(x,y)$ and $\max(x,y)$.

Table C3-88 on page C3-317 shows the Floating-point instructions that can perform floating-point minimum and maximum operations.

Table C3-87 Floating-point arithmetic instructions with two source registers (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMUL</td>
<td>Floating-point scalar multiply</td>
<td>FMUL (scalar) on page C7-2350</td>
</tr>
<tr>
<td>FNMUL</td>
<td>Floating-point scalar multiply-negate</td>
<td>FNMUL (scalar) on page C7-2367</td>
</tr>
<tr>
<td>FSUB</td>
<td>Floating-point scalar subtract</td>
<td>FSUB (scalar) on page C7-2433</td>
</tr>
</tbody>
</table>

C3.6.10 Floating-point comparison

These instructions set the NZCV Condition flags in PSTATE, based on the result of a comparison of two operands. If the floating-point comparisons are unordered, where one or both operands are a form of NaN, the C and V bits are set to 1 and the N and Z bits are cleared to 0.

--- Note ---

The NZCV flags in the FPSR are associated with AArch32 state. The A64 floating-point comparison instructions do not change the Condition flags in the FPSR.

For the conditional Floating-point comparison instructions, if the condition is TRUE, the flags are updated to the result of the comparison, otherwise the flags are updated to the immediate value that is defined in the instruction encoding.

The quiet compare instructions generate an Invalid Operation floating-point exception if either of the source operands is a signaling NaN. The signaling compare instructions generate an Invalid Operation floating-point exception if either of the source operands is any type of NaN.

--- Note ---

If FEAT_FlagM2 is implemented, instructions AXFLAG and XAFLAG convert between the PSTATE condition flag format used by the FCMP instruction and an alternative format. See FEAT_FlagM on page A2-101 for more information.
Table C3-89 shows the Floating-point comparison instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCMP</td>
<td>Floating-point quiet compare</td>
<td>FCMP on page C7-2190</td>
</tr>
<tr>
<td>FCMPE</td>
<td>Floating-point signaling compare</td>
<td>FCMPE on page C7-2192</td>
</tr>
<tr>
<td>FCCMP</td>
<td>Floating-point conditional quiet compare</td>
<td>FCCMP on page C7-2153</td>
</tr>
<tr>
<td>FCCMPE</td>
<td>Floating-point conditional signaling compare</td>
<td>FCCMPE on page C7-2155</td>
</tr>
</tbody>
</table>

C3.6.11 Floating-point conditional select

Table C3-90 on page C3-318 shows the Floating-point conditional select instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCSEL</td>
<td>Floating-point scalar conditional select</td>
<td>FCSEL on page C7-2194</td>
</tr>
</tbody>
</table>

C3.6.12 SIMD move

The functionality of some data movement instructions overlaps with that provided by the scalar floating-point FMOV instructions described in Floating-point move (register) on page C3-312.

Table C3-91 on page C3-318 shows the SIMD move instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUP</td>
<td>Duplicate vector element to vector or scalar</td>
<td>DUP (element) on page C7-2118</td>
</tr>
<tr>
<td>DUG</td>
<td>Duplicate general-purpose register to vector</td>
<td>DUG (general) on page C7-2121</td>
</tr>
<tr>
<td>INS a</td>
<td>Insert vector element from another vector element</td>
<td>INS (element) on page C7-2435</td>
</tr>
<tr>
<td>INS a</td>
<td>Insert vector element from general-purpose register</td>
<td>INS (general) on page C7-2437</td>
</tr>
<tr>
<td>MOV</td>
<td>Move vector element to vector element</td>
<td>MOV (element) on page C7-2507</td>
</tr>
<tr>
<td>MOV</td>
<td>Move general-purpose register to vector element</td>
<td>MOV (from general) on page C7-2509</td>
</tr>
<tr>
<td>MOV</td>
<td>Move vector element to scalar</td>
<td>MOV (scalar) on page C7-2505</td>
</tr>
<tr>
<td>MOV</td>
<td>Move vector element to general-purpose register</td>
<td>MOV (to general) on page C7-2512</td>
</tr>
<tr>
<td>UMOV</td>
<td>Unsigned move vector element to general-purpose register</td>
<td>UMOV on page C7-2896</td>
</tr>
<tr>
<td>SMOV</td>
<td>Signed move vector element to general-purpose register</td>
<td>SMOV on page C7-2659</td>
</tr>
</tbody>
</table>

a. Disassembles as MOV.

C3.6.13 SIMD arithmetic

Table C3-92 on page C3-319 shows the SIMD arithmetic instructions.
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add (vector and scalar form)</td>
<td><em>ADD (vector) on page C7-2045</em></td>
</tr>
<tr>
<td>AND</td>
<td>Bitwise AND (vector form)</td>
<td><em>AND (vector) on page C7-2059</em></td>
</tr>
<tr>
<td>BIC</td>
<td>Bitwise bit clear (register) (vector form)</td>
<td><em>BIC (vector, register) on page C7-2074</em></td>
</tr>
<tr>
<td>BIF</td>
<td>Bitwise insert if false (vector form)</td>
<td><em>BIF on page C7-2076</em></td>
</tr>
<tr>
<td>BIT</td>
<td>Bitwise insert if true (vector form)</td>
<td><em>BIT on page C7-2078</em></td>
</tr>
<tr>
<td>BSL</td>
<td>Bitwise select (vector form)</td>
<td><em>BSL on page C7-2080</em></td>
</tr>
<tr>
<td>EOR</td>
<td>Bitwise exclusive OR (vector form)</td>
<td><em>EOR (vector) on page C7-2123</em></td>
</tr>
<tr>
<td>FABD</td>
<td>Floating-point absolute difference (vector and scalar form)</td>
<td><em>FABD on page C7-2128</em></td>
</tr>
<tr>
<td>FADD</td>
<td>Floating-point add (vector form)</td>
<td><em>FADD (vector) on page C7-2143</em></td>
</tr>
<tr>
<td>FDIV</td>
<td>Floating-point divide (vector form)</td>
<td><em>FDIV (vector) on page C7-2265</em></td>
</tr>
<tr>
<td>FMAX</td>
<td>Floating-point maximum (vector form)</td>
<td><em>FMAXP (vector) on page C7-2288</em></td>
</tr>
<tr>
<td>FMAXNM</td>
<td>Floating-point maximum number (vector form)</td>
<td><em>FMAXNM (vector) on page C7-2276</em></td>
</tr>
<tr>
<td>FMIN</td>
<td>Floating-point minimum (vector form)</td>
<td><em>FMIN (vector) on page C7-2292</em></td>
</tr>
<tr>
<td>FMINNM</td>
<td>Floating-point minimum number (vector form)</td>
<td><em>FMINNM (vector) on page C7-2296</em></td>
</tr>
<tr>
<td>FMLA</td>
<td>Floating-point fused multiply-add (vector form)</td>
<td><em>FMLA (vector) on page C7-2316</em></td>
</tr>
<tr>
<td>FMLAL, FMLAL2</td>
<td>Floating-point fused multiply-add long (vector form)</td>
<td><em>FMLAL, FMLAL2 (vector) on page C7-2320</em></td>
</tr>
<tr>
<td>FMLS</td>
<td>Floating-point fused multiply-subtract (vector form)</td>
<td><em>FMLS (vector) on page C7-2326</em></td>
</tr>
<tr>
<td>FMLSL, FMLSL2</td>
<td>Floating-point fused multiply-subtract long (vector form)</td>
<td><em>FMLSL, FMLSL2 (vector) on page C7-2330</em></td>
</tr>
<tr>
<td>FMUL</td>
<td>Floating-point multiply (vector form)</td>
<td><em>FMUL (vector) on page C7-2348</em></td>
</tr>
<tr>
<td>FMULX</td>
<td>Floating-point multiply extended (vector and scalar form)</td>
<td><em>FMULX on page C7-2356</em></td>
</tr>
<tr>
<td>FRECPS</td>
<td>Floating-point reciprocal step (vector and scalar form)</td>
<td><em>FRECPS on page C7-2372</em></td>
</tr>
<tr>
<td>FRSQRTS</td>
<td>Floating-point reciprocal square root step (vector and scalar form)</td>
<td><em>FRSQRTS on page C7-2424</em></td>
</tr>
<tr>
<td>FSUB</td>
<td>Floating-point subtract (vector form)</td>
<td><em>FSUB (vector) on page C7-2431</em></td>
</tr>
<tr>
<td>MLA</td>
<td>Multiply-add (vector form)</td>
<td><em>MLA (vector) on page C7-2499</em></td>
</tr>
<tr>
<td>MLS</td>
<td>Multiply-subtract (vector form)</td>
<td><em>MLS (vector) on page C7-2503</em></td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply (vector form)</td>
<td><em>MUL (vector) on page C7-2519</em></td>
</tr>
<tr>
<td>MOV</td>
<td>Move vector register (vector form)</td>
<td><em>MOV (vector) on page C7-2511</em></td>
</tr>
<tr>
<td>ORN</td>
<td>Bitwise inclusive OR NOT (vector form)</td>
<td><em>ORN (vector) on page C7-2529</em></td>
</tr>
<tr>
<td>ORR</td>
<td>Bitwise inclusive OR (register) (vector form)</td>
<td><em>ORR (vector, register) on page C7-2533</em></td>
</tr>
<tr>
<td>PMUL</td>
<td>Polynomial multiply (vector form)</td>
<td><em>PMUL on page C7-2535</em></td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Instruction</td>
<td>See</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-----</td>
</tr>
<tr>
<td>SABA</td>
<td>Signed absolute difference and accumulate (vector form)</td>
<td>SABA on page C7-2554</td>
</tr>
<tr>
<td>SABD</td>
<td>Signed absolute difference (vector form)</td>
<td>SABD on page C7-2558</td>
</tr>
<tr>
<td>SHADD</td>
<td>Signed halving add (vector form)</td>
<td>SHADD on page C7-2604</td>
</tr>
<tr>
<td>SHSUB</td>
<td>Signed halving subtract (vector form)</td>
<td>SHSUB on page C7-2613</td>
</tr>
<tr>
<td>SMAX</td>
<td>Signed maximum (vector form)</td>
<td>SMAX on page C7-2636</td>
</tr>
<tr>
<td>SMIN</td>
<td>Signed minimum (vector form)</td>
<td>SMIN on page C7-2642</td>
</tr>
<tr>
<td>SQADD</td>
<td>Signed saturating add (vector and scalar form)</td>
<td>SQADD on page C7-2668</td>
</tr>
<tr>
<td>SQDMULH</td>
<td>Signed saturating doubling multiply returning high half (vector and scalar form)</td>
<td>SQDMULH (vector) on page C7-2687</td>
</tr>
<tr>
<td>SQRSHL</td>
<td>Signed saturating rounding shift left (register) (vector and scalar form)</td>
<td>SQRSHL on page C7-2716</td>
</tr>
<tr>
<td>SQRDMLAH</td>
<td>Signed saturating rounding doubling multiply accumulate returning high half</td>
<td>SQRDMLAH (vector) on page C7-2701</td>
</tr>
<tr>
<td>SQRDMLSH</td>
<td>Signed saturating rounding doubling multiply subtract returning high half</td>
<td>SQRDMLSH (vector) on page C7-2707</td>
</tr>
<tr>
<td>SQDMULH</td>
<td>Signed saturating rounding doubling multiply returning high half (vector and scalar form)</td>
<td>SQDMULH (vector) on page C7-2713</td>
</tr>
<tr>
<td>SQSHL</td>
<td>Signed saturating shift left (register) (vector and scalar form)</td>
<td>SQSHL (register) on page C7-2728</td>
</tr>
<tr>
<td>SQSUB</td>
<td>Signed saturating subtract (vector and scalar form)</td>
<td>SQSUB on page C7-2740</td>
</tr>
<tr>
<td>SRHADD</td>
<td>Signed rounding halving add (vector form)</td>
<td>SRHADD on page C7-2748</td>
</tr>
<tr>
<td>SRSHL</td>
<td>Signed rounding shift left (register) (vector and scalar form)</td>
<td>SRSHL on page C7-2753</td>
</tr>
<tr>
<td>SSHL</td>
<td>Signed shift left (register) (vector and scalar form)</td>
<td>SSHL on page C7-2761</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract (vector and scalar form)</td>
<td>SUB (vector) on page C7-2819</td>
</tr>
<tr>
<td>UABA</td>
<td>Unsigned absolute difference and accumulate (vector form)</td>
<td>UABA on page C7-2837</td>
</tr>
<tr>
<td>UABD</td>
<td>Unsigned absolute difference (vector form)</td>
<td>UABD on page C7-2841</td>
</tr>
<tr>
<td>UHADD</td>
<td>Unsigned halving add (vector form)</td>
<td>UHADD on page C7-2869</td>
</tr>
<tr>
<td>UHSUB</td>
<td>Unsigned halving subtract (vector form)</td>
<td>UHSUB on page C7-2871</td>
</tr>
<tr>
<td>UMAX</td>
<td>Unsigned maximum (vector form)</td>
<td>UMAX on page C7-2873</td>
</tr>
<tr>
<td>UMIN</td>
<td>Unsigned minimum (vector form)</td>
<td>UMIN on page C7-2879</td>
</tr>
<tr>
<td>UQADD</td>
<td>Unsigned saturating add (vector and scalar form)</td>
<td>UQADD on page C7-2903</td>
</tr>
<tr>
<td>UQRSRL</td>
<td>Unsigned saturating rounding shift left (register) (vector and scalar form)</td>
<td>UQRSRL on page C7-2905</td>
</tr>
<tr>
<td>UQSHL</td>
<td>Unsigned saturating shift left (register) (vector and scalar form)</td>
<td>UQSHL (register) on page C7-2914</td>
</tr>
<tr>
<td>UQSUB</td>
<td>Unsigned saturating subtract (vector and scalar form)</td>
<td>UQSUB on page C7-2920</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Instruction</td>
<td>See</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>URHADD</td>
<td>Unsigned rounding halving add (vector form)</td>
<td><em>URHADD on page C7-2926</em></td>
</tr>
<tr>
<td>URSHL</td>
<td>Unsigned rounding shift left (register) (vector and scalar form)</td>
<td><em>URSHL on page C7-2928</em></td>
</tr>
<tr>
<td>USHL</td>
<td>Unsigned shift left (register) (vector and scalar form)</td>
<td><em>USHL on page C7-2941</em></td>
</tr>
</tbody>
</table>
C3.6.14  SIMD compare

The SIMD compare instructions compare vector or scalar elements according to the specified condition and set the destination vector element to all ones if the condition holds, or to zero if the condition does not hold.

--- Note ---

Some of the comparisons, such as LS, LE, LO, and LT, can be made by reversing the operands and using the opposite comparison, HS, GE, HI, or GT.

Table C3-93 on page C3-322 shows that SIMD compare instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMEQ</td>
<td>Compare bitwise equal (vector and scalar form)</td>
<td>CMEQ (register) on page C7-2086</td>
</tr>
<tr>
<td></td>
<td>Compare bitwise equal to zero (vector and scalar form)</td>
<td>CMEQ (zero) on page C7-2088</td>
</tr>
<tr>
<td>CMHS</td>
<td>Compare unsigned higher or same (vector and scalar form)</td>
<td>CMHS (register) on page C7-2106</td>
</tr>
<tr>
<td>CMGE</td>
<td>Compare signed greater than or equal (vector and scalar form)</td>
<td>CMGE (register) on page C7-2091</td>
</tr>
<tr>
<td></td>
<td>Compare signed greater than or equal to zero (vector and scalar form)</td>
<td>CMGE (zero) on page C7-2094</td>
</tr>
<tr>
<td>CMHI</td>
<td>Compare unsigned higher (vector and scalar form)</td>
<td>CMHI (register) on page C7-2103</td>
</tr>
<tr>
<td>CMGT</td>
<td>Compare signed greater than (vector and scalar form)</td>
<td>CMGT (register) on page C7-2097</td>
</tr>
<tr>
<td></td>
<td>Compare signed greater than zero (vector and scalar form)</td>
<td>CMGT (zero) on page C7-2100</td>
</tr>
<tr>
<td>CMLE</td>
<td>Compare signed less than or equal to zero (vector and scalar form)</td>
<td>CMLE (zero) on page C7-2109</td>
</tr>
<tr>
<td>CMLT</td>
<td>Compare signed less than zero (vector and scalar form)</td>
<td>CMLT (zero) on page C7-2112</td>
</tr>
<tr>
<td>CMTST</td>
<td>Compare bitwise test bits nonzero (vector and scalar form)</td>
<td>CMTST on page C7-2114</td>
</tr>
<tr>
<td>FCMEQ</td>
<td>Floating-point compare equal (vector and scalar form)</td>
<td>FCMEQ (register) on page C7-2157</td>
</tr>
<tr>
<td></td>
<td>Floating-point compare equal to zero (vector and scalar form)</td>
<td>FCMEQ (zero) on page C7-2161</td>
</tr>
<tr>
<td>FCMEG</td>
<td>Floating-point compare greater than or equal (vector and scalar form)</td>
<td>FCMEG (register) on page C7-2164</td>
</tr>
<tr>
<td></td>
<td>Floating-point compare greater than or equal to zero (vector and scalar form)</td>
<td>FCMEG (zero) on page C7-2168</td>
</tr>
<tr>
<td>FCMT</td>
<td>Floating-point compare greater than (vector and scalar form)</td>
<td>FCMT (register) on page C7-2171</td>
</tr>
<tr>
<td></td>
<td>Floating-point compare greater than zero (vector and scalar form)</td>
<td>FCMT (zero) on page C7-2175</td>
</tr>
<tr>
<td>FCMLE</td>
<td>Floating-point compare less than or equal to zero (vector and scalar form)</td>
<td>FCMLE (zero) on page C7-2184</td>
</tr>
<tr>
<td>FCMLT</td>
<td>Floating-point compare less than zero (vector and scalar form)</td>
<td>FCMLT (zero) on page C7-2187</td>
</tr>
<tr>
<td>FACGE</td>
<td>Floating-point absolute compare greater than or equal (vector and scalar form)</td>
<td>FACGE on page C7-2135</td>
</tr>
<tr>
<td>FACGT</td>
<td>Floating-point absolute compare greater than (vector and scalar form)</td>
<td>FACGT on page C7-2139</td>
</tr>
</tbody>
</table>

C3.6.15  SIMD widening and narrowing arithmetic

For information about the variants of these instructions, see Common features of SIMD instructions on page C3-311.
Table C3-94 on page C3-323 shows the SIMD widening and narrowing arithmetic instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDHN, ADDHN2</td>
<td>Add returning high, narrow (vector form)</td>
<td>ADDHN, ADDHN2 on page C7-2047</td>
</tr>
<tr>
<td>PMULL, PMULL2</td>
<td>Polynomial multiply long (vector form)</td>
<td>PMULL, PMULL2 on page C7-2537</td>
</tr>
<tr>
<td>See also The Cryptographic Extension on page C3-334</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RADDHN, RADDHN2</td>
<td>Rounding add returning high, narrow (vector form)</td>
<td>RADDHN, RADDHN2 on page C7-2539</td>
</tr>
<tr>
<td>RSUBHN, RSUBHN2</td>
<td>Rounding subtract returning high, narrow (vector form)</td>
<td>RSUBHN, RSUBHN2 on page C7-2552</td>
</tr>
<tr>
<td>SABAL, SABAL2</td>
<td>Signed absolute difference and accumulate long (vector form)</td>
<td>SABAL, SABAL2 on page C7-2556</td>
</tr>
<tr>
<td>SABDL, SABDL2</td>
<td>Signed absolute difference long (vector form)</td>
<td>SABDL, SABDL2 on page C7-2560</td>
</tr>
<tr>
<td>SADDL, SADDL2</td>
<td>Signed add long (vector form)</td>
<td>SADDL, SADDL2 on page C7-2564</td>
</tr>
<tr>
<td>SADDL, SADDL2</td>
<td>Signed add wide (vector form)</td>
<td>SADDL, SADDL2 on page C7-2570</td>
</tr>
<tr>
<td>SMLAL, SMLAL2</td>
<td>Signed multiply-add long (vector form)</td>
<td>SMLAL, SMLAL2 (vector) on page C7-2651</td>
</tr>
<tr>
<td>SMLSL, SMLSL2</td>
<td>Signed multiply-subtract long (vector form)</td>
<td>SMLSL, SMLSL2 (vector) on page C7-2656</td>
</tr>
<tr>
<td>SMULL, SMULL2</td>
<td>Signed multiply long (vector form)</td>
<td>SMULL, SMULL2 (vector) on page C7-2664</td>
</tr>
<tr>
<td>SQDMLAL, SQDMLAL2</td>
<td>Signed saturating doubling multiply-add long (vector and scalar form)</td>
<td>SQDMLAL, SQDMLAL2 (vector) on page C7-2674</td>
</tr>
<tr>
<td>SQDMLSL, SQDMLSL2</td>
<td>Signed saturating doubling multiply-subtract long (vector and scalar form)</td>
<td>SQDMLSL, SQDMLSL2 (vector) on page C7-2681</td>
</tr>
<tr>
<td>SQDMULL, SQDMULL2</td>
<td>Signed saturating doubling multiply long (vector and scalar form)</td>
<td>SQDMULL, SQDMULL2 (vector) on page C7-2693</td>
</tr>
<tr>
<td>SSUBL, SSUBL2</td>
<td>Signed subtract long (vector form)</td>
<td>SSUBL, SSUBL2 on page C7-2772</td>
</tr>
<tr>
<td>SSUBW, SSUBW2</td>
<td>Signed subtract wide (vector form)</td>
<td>SSUBW, SSUBW2 on page C7-2774</td>
</tr>
<tr>
<td>SUBHN, SUBHN2</td>
<td>Subtract returning high, narrow (vector form)</td>
<td>SUBHN, SUBHN2 on page C7-2821</td>
</tr>
<tr>
<td>UABAL, UABAL2</td>
<td>Unsigned absolute difference and accumulate long (vector form)</td>
<td>UABAL, UABAL2 on page C7-2839</td>
</tr>
<tr>
<td>UABDL, UABDL2</td>
<td>Unsigned absolute difference long (vector form)</td>
<td>UABDL, UABDL2 on page C7-2843</td>
</tr>
<tr>
<td>UADDL, UADDL2</td>
<td>Unsigned add long (vector form)</td>
<td>UADDL, UADDL2 on page C7-2847</td>
</tr>
<tr>
<td>UADDL, UADDL2</td>
<td>Unsigned add wide (vector form)</td>
<td>UADDL, UADDL2 on page C7-2853</td>
</tr>
<tr>
<td>UMLAL, UMLAL2</td>
<td>Unsigned multiply-add long (vector form)</td>
<td>UMLAL, UMLAL2 (vector) on page C7-2888</td>
</tr>
<tr>
<td>UMLSL, UMLSL2</td>
<td>Unsigned multiply-subtract long (vector form)</td>
<td>UMLSL, UMLSL2 (vector) on page C7-2893</td>
</tr>
<tr>
<td>UMLULL, UMLULL2</td>
<td>Unsigned multiply long (vector form)</td>
<td>UMLULL, UMLULL2 (vector) on page C7-2901</td>
</tr>
<tr>
<td>USUBL, USUBL2</td>
<td>Unsigned subtract long (vector form)</td>
<td>USUBL, USUBL2 on page C7-2955</td>
</tr>
<tr>
<td>USUBW, USUBW2</td>
<td>Unsigned subtract wide (vector form)</td>
<td>USUBW, USUBW2 on page C7-2957</td>
</tr>
</tbody>
</table>
C3.6.16 SIMD unary arithmetic

For information about the variants of these instructions, see Common features of SIMD instructions on page C3-311. Table C3-95 on page C3-324 shows the SIMD unary arithmetic instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Absolute value (vector and scalar form)</td>
<td>ABS on page C7-2043</td>
</tr>
<tr>
<td>CLS</td>
<td>Count leading sign bits (vector form)</td>
<td>CLS (vector) on page C7-2082</td>
</tr>
<tr>
<td>CLZ</td>
<td>Count leading zero bits (vector form)</td>
<td>CLZ (vector) on page C7-2084</td>
</tr>
<tr>
<td>CNT</td>
<td>Population count per byte (vector form)</td>
<td>CNT on page C7-2116</td>
</tr>
<tr>
<td>FABS</td>
<td>Floating-point absolute (vector form)</td>
<td>FABS (vector) on page C7-2131</td>
</tr>
<tr>
<td>FCVTL, FCVTL2</td>
<td>Floating-point convert to higher precision long (vector form)</td>
<td>FCVTL, FCVTL2 on page C7-2208</td>
</tr>
<tr>
<td>FCVTN, FCVTN2</td>
<td>Floating-point convert to lower precision narrow (vector form)</td>
<td>FCVTN, FCVTN2 on page C7-2220</td>
</tr>
<tr>
<td>FCVTXN, FCVTXN2</td>
<td>Floating-point convert to lower precision narrow, rounding to odd (vector and scalar form)</td>
<td>FCVTXN, FCVTXN2 on page C7-2242</td>
</tr>
<tr>
<td>FNEG</td>
<td>Floating-point negate (vector form)</td>
<td>FNEG (vector) on page C7-2359</td>
</tr>
<tr>
<td>FRECPE</td>
<td>Floating-point reciprocal estimate (vector and scalar form)</td>
<td>FRECPE on page C7-2369</td>
</tr>
<tr>
<td>FRECPIX</td>
<td>Floating-point reciprocal exponent (scalar form)</td>
<td>FRECPIX on page C7-2375</td>
</tr>
<tr>
<td>FRINT32X</td>
<td>Floating-point round to 32-bit integer, using current rounding mode (vector form)</td>
<td>FRINT32X (vector) on page C7-2377</td>
</tr>
<tr>
<td>FRINT32Z</td>
<td>Floating-point round to 32-bit integer, toward zero (vector form)</td>
<td>FRINT32Z (vector) on page C7-2381</td>
</tr>
<tr>
<td>FRINT64X</td>
<td>Floating-point round to 64-bit integer, using current rounding mode (vector form)</td>
<td>FRINT64X (vector) on page C7-2385</td>
</tr>
<tr>
<td>FRINT64Z</td>
<td>Floating-point round to 64-bit integer, toward zero (vector form)</td>
<td>FRINT64Z (vector) on page C7-2389</td>
</tr>
<tr>
<td>FRINTA</td>
<td>Floating-point round to integer, to nearest with ties to away (vector form)</td>
<td>FRINTA (vector) on page C7-2393</td>
</tr>
<tr>
<td>FRINTI</td>
<td>Floating-point round to integer, using current rounding mode (vector form)</td>
<td>FRINTI (vector) on page C7-2397</td>
</tr>
<tr>
<td>FRINTM</td>
<td>Floating-point round to integer, toward minus infinity (vector form)</td>
<td>FRINTM (vector) on page C7-2401</td>
</tr>
<tr>
<td>FRINTN</td>
<td>Floating-point round to integer, to nearest with ties to even (vector form)</td>
<td>FRINTN (vector) on page C7-2405</td>
</tr>
<tr>
<td>FRINTP</td>
<td>Floating-point round to integer, toward positive infinity (vector form)</td>
<td>FRINTP (vector) on page C7-2409</td>
</tr>
<tr>
<td>FRINTX</td>
<td>Floating-point round to integer exact, using current rounding mode (vector form)</td>
<td>FRINTX (vector) on page C7-2413</td>
</tr>
<tr>
<td>FRINTZ</td>
<td>Floating-point round to integer, toward zero (vector form)</td>
<td>FRINTZ (vector) on page C7-2417</td>
</tr>
<tr>
<td>FRSQRTE</td>
<td>Floating-point reciprocal square root estimate (vector and scalar form)</td>
<td>FRSQRTE on page C7-2421</td>
</tr>
</tbody>
</table>
Table C3-95  SIMD unary arithmetic instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSQRT</td>
<td>Floating-point square root (vector form)</td>
<td>FSQRT (vector) on page C7-2427</td>
</tr>
<tr>
<td>MVN</td>
<td>Bitwise NOT (vector form)</td>
<td>MVN on page C7-2521</td>
</tr>
<tr>
<td>NEG</td>
<td>Negate (vector and scalar form)</td>
<td>NEG (vector) on page C7-2525</td>
</tr>
<tr>
<td>NOT</td>
<td>Bitwise NOT (vector form)</td>
<td>NOT on page C7-2527</td>
</tr>
<tr>
<td>RBIT</td>
<td>Bitwise reverse (vector form)</td>
<td>RBIT (vector) on page C7-2542</td>
</tr>
<tr>
<td>REV16</td>
<td>Reverse elements in 16-bit halfwords (vector form)</td>
<td>REV16 (vector) on page C7-2544</td>
</tr>
<tr>
<td>REV32</td>
<td>Reverse elements in 32-bit words (vector form)</td>
<td>REV32 (vector) on page C7-2546</td>
</tr>
<tr>
<td>REV64</td>
<td>Reverse elements in 64-bit doublewords (vector form)</td>
<td>REV64 on page C7-2548</td>
</tr>
<tr>
<td>SADALP</td>
<td>Signed add and accumulate long pairwise (vector form)</td>
<td>SADALP on page C7-2562</td>
</tr>
<tr>
<td>SADDLP</td>
<td>Signed add long pairwise (vector form)</td>
<td>SADDLP on page C7-2566</td>
</tr>
<tr>
<td>SQABS</td>
<td>Signed saturating absolute value (vector and scalar form)</td>
<td>SQABS on page C7-2666</td>
</tr>
<tr>
<td>SQNEG</td>
<td>Signed saturating negate (vector and scalar form)</td>
<td>SQNEG on page C7-2696</td>
</tr>
<tr>
<td>SQXTN, SQXTN2</td>
<td>Signed saturating extract narrow (vector form)</td>
<td>SQXTN, SQXTN2 on page C7-2742</td>
</tr>
<tr>
<td>SQXTUN, SQXTUN2</td>
<td>Signed saturating extract unsigned narrow (vector form)</td>
<td>SQXTUN, SQXTUN2 on page C7-2745</td>
</tr>
<tr>
<td>SUQADD</td>
<td>Signed saturating accumulate of unsigned value (vector and scalar form)</td>
<td>SUQADD on page C7-2825</td>
</tr>
<tr>
<td>SXTL, SXTL2</td>
<td>Signed extend long</td>
<td>SXTL, SXTL2 on page C7-2827</td>
</tr>
<tr>
<td>UADALP</td>
<td>Unsigned add and accumulate long pairwise (vector form)</td>
<td>UADALP on page C7-2845</td>
</tr>
<tr>
<td>UADDLP</td>
<td>Unsigned add long pairwise (vector form)</td>
<td>UADDLP on page C7-2849</td>
</tr>
<tr>
<td>UQXTN, UQXTN2</td>
<td>Unsigned saturating extract narrow (vector form)</td>
<td>UQXTN, UQXTN2 on page C7-2922</td>
</tr>
<tr>
<td>URECPE</td>
<td>Unsigned reciprocal estimate (vector form)</td>
<td>URECPE on page C7-2925</td>
</tr>
<tr>
<td>URSQRT</td>
<td>Unsigned reciprocal square root estimate (vector form)</td>
<td>URSQRT on page C7-2933</td>
</tr>
<tr>
<td>USQADD</td>
<td>Unsigned saturating accumulate of signed value (vector and scalar form)</td>
<td>USQADD on page C7-2950</td>
</tr>
<tr>
<td>UXTL, UXTL2</td>
<td>Unsigned extend long</td>
<td>UXTL, UXTL2 on page C7-2959</td>
</tr>
<tr>
<td>XTN, XTN2</td>
<td>Extract narrow (vector form)</td>
<td>XTN, XTN2 on page C7-2966</td>
</tr>
</tbody>
</table>

C3.6.17 SIMD by element arithmetic

For information about the variants of these instructions, see Common features of SIMD instructions on page C3-311.
Table C3-96 on page C3-326 shows the SIMD by element arithmetic instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMLA</td>
<td>Floating-point fused multiply-add (vector and scalar form)</td>
<td>FMLA (by element) on page C7-2312</td>
</tr>
<tr>
<td>FMLAL, FMLAL2</td>
<td>Floating-point fused multiply-add long (vector form)</td>
<td>FMLAL, FMLAL2 (by element) on page C7-2318</td>
</tr>
<tr>
<td>FMLS</td>
<td>Floating-point fused multiply-subtract (vector and scalar form)</td>
<td>FMLS (by element) on page C7-2322.</td>
</tr>
<tr>
<td>FMLSL, FMLSL2</td>
<td>Floating-point fused multiply-subtract long (vector form)</td>
<td>FMLSL, FMLSL2 (by element) on page C7-2328</td>
</tr>
<tr>
<td>FMUL</td>
<td>Floating-point multiply (vector and scalar form)</td>
<td>FMUL (by element) on page C7-2344</td>
</tr>
<tr>
<td>FMULX</td>
<td>Floating-point multiply extended (vector and scalar form)</td>
<td>FMULX (by element) on page C7-2352</td>
</tr>
<tr>
<td>MLA</td>
<td>Multiply-add (vector form)</td>
<td>MLA (by element) on page C7-2497</td>
</tr>
<tr>
<td>MLS</td>
<td>Multiply-subtract (vector form)</td>
<td>MLS (by element) on page C7-2501</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply (vector form)</td>
<td>MUL (by element) on page C7-2517</td>
</tr>
<tr>
<td>SMLAL, SMLAL2</td>
<td>Signed multiply-add long (vector form)</td>
<td>SMLAL, SMLAL2 (by element) on page C7-2648</td>
</tr>
<tr>
<td>SMLSL, SMLSL2</td>
<td>Signed multiply-subtract long (vector form)</td>
<td>SMLSL, SMLSL2 (by element) on page C7-2653</td>
</tr>
<tr>
<td>SMULL, SMULL2</td>
<td>Signed multiply long (vector form)</td>
<td>SMULL, SMULL2 (by element) on page C7-2661</td>
</tr>
<tr>
<td>SQDMLAL, SQDMLAL2</td>
<td>Signed saturating doubling multiply-add long (vector and scalar form)</td>
<td>SQDMLAL, SQDMLAL2 (by element) on page C7-2670</td>
</tr>
<tr>
<td>SQDMLSL, SQDMLSL2</td>
<td>Signed saturating doubling multiply-subtract long (vector form)</td>
<td>SQDMLSL, SQDMLSL2 (by element) on page C7-2677</td>
</tr>
<tr>
<td>SQDMULH</td>
<td>Signed saturating doubling multiply returning high half (vector and scalar form)</td>
<td>SQDMULH (by element) on page C7-2684</td>
</tr>
<tr>
<td>SQDMULL, SQDMULL2</td>
<td>Signed saturating doubling multiply long (vector and scalar form)</td>
<td>SQDMULL, SQDMULL2 (by element) on page C7-2690</td>
</tr>
<tr>
<td>SQRDMLAH</td>
<td>Signed saturating rounding doubling multiply accumulate returning high half</td>
<td>SQRDMLAH (by element) on page C7-2704</td>
</tr>
<tr>
<td>SQRDMLSH</td>
<td>Signed saturating rounding doubling multiply subtract returning high half</td>
<td>SQRDMLSH (by element) on page C7-2707</td>
</tr>
<tr>
<td>SQRDMLULH</td>
<td>Signed saturating rounding doubling multiply returning high half (vector and scalar form)</td>
<td>SQRDMLULH (by element) on page C7-2710</td>
</tr>
<tr>
<td>UMLAL, UMLAL2</td>
<td>Unsigned multiply-add long (vector form)</td>
<td>UMLAL, UMLAL2 (by element) on page C7-2885</td>
</tr>
<tr>
<td>UMLSL, UMLSL2</td>
<td>Unsigned multiply-subtract long (vector form)</td>
<td>UMLSL, UMLSL2 (by element) on page C7-2890</td>
</tr>
<tr>
<td>UMULL, UMULL2</td>
<td>Unsigned multiply long (vector form)</td>
<td>UMULL, UMULL2 (by element) on page C7-2898</td>
</tr>
</tbody>
</table>
**C3.6.18 SIMD permute**

Table C3-97 on page C3-327 shows the SIMD permute instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXT</td>
<td>Extract vector from a pair of vectors</td>
<td>EXT on page C7-2126</td>
</tr>
<tr>
<td>TRN1</td>
<td>Transpose vectors (primary)</td>
<td>TRN1 on page C7-2833</td>
</tr>
<tr>
<td>TRN2</td>
<td>Transpose vectors (secondary)</td>
<td>TRN2 on page C7-2835</td>
</tr>
<tr>
<td>UZP1</td>
<td>Unzip vectors (primary)</td>
<td>UZP1 on page C7-2961</td>
</tr>
<tr>
<td>UZP2</td>
<td>Unzip vectors (secondary)</td>
<td>UZP2 on page C7-2963</td>
</tr>
<tr>
<td>ZIP1</td>
<td>Zip vectors (primary)</td>
<td>ZIP1 on page C7-2968</td>
</tr>
<tr>
<td>ZIP2</td>
<td>Zip vectors (secondary)</td>
<td>ZIP2 on page C7-2970</td>
</tr>
</tbody>
</table>

**C3.6.19 SIMD immediate**

Table C3-98 on page C3-327 shows the SIMD immediate instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIC</td>
<td>Bitwise bit clear immediate</td>
<td>BIC (vector, immediate) on page C7-2072</td>
</tr>
<tr>
<td>FMOV</td>
<td>Floating-point move immediate</td>
<td>FMOV (vector, immediate) on page C7-2332</td>
</tr>
<tr>
<td>MOVI</td>
<td>Move immediate</td>
<td>MOVI on page C7-2514</td>
</tr>
<tr>
<td>MVNI</td>
<td>Move inverted immediate</td>
<td>MVNI on page C7-2522</td>
</tr>
<tr>
<td>ORR</td>
<td>Bitwise inclusive OR immediate</td>
<td>ORR (vector, immediate) on page C7-2531</td>
</tr>
</tbody>
</table>

**C3.6.20 SIMD shift (immediate)**

For information about the variants of these instructions, see *Common features of SIMD instructions* on page C3-311.

Table C3-99 on page C3-327 shows the SIMD shift immediate instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSHRN, RSHRN2</td>
<td>Rounding shift right narrow immediate (vector form)</td>
<td>RSHRN, RSHRN2 on page C7-2550</td>
</tr>
<tr>
<td>SHL</td>
<td>Shift left immediate (vector and scalar form)</td>
<td>SHL on page C7-2606</td>
</tr>
<tr>
<td>SHLL, SHLL2</td>
<td>Shift left long (by element size) (vector form)</td>
<td>SHLL, SHLL2 on page C7-2609</td>
</tr>
<tr>
<td>SHRN, SHRN2</td>
<td>Shift right narrow immediate (vector form)</td>
<td>SHRN, SHRN2 on page C7-2611</td>
</tr>
<tr>
<td>SLI</td>
<td>Shift left and insert immediate (vector and scalar form)</td>
<td>SLI on page C7-2615</td>
</tr>
<tr>
<td>SQRSHRN, SQRSHRN2</td>
<td>Signed saturating rounded shift right narrow immediate (vector and scalar form)</td>
<td>SQRSHRN, SQRSHRN2 on page C7-2719</td>
</tr>
</tbody>
</table>
## SIMD shift (immediate) instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQRSHRUN, SQRSHRUN2</td>
<td>Signed saturating shift right unsigned narrow immediate (vector and scalar form)</td>
<td>SQRSHRUN, SQRSHRUN2 on page C7-2722</td>
</tr>
<tr>
<td>SQSHL</td>
<td>Signed saturating shift left immediate (vector and scalar form)</td>
<td>SQSHL (immediate) on page C7-2725</td>
</tr>
<tr>
<td>SQSHLU</td>
<td>Signed saturating shift left unsigned immediate (vector and scalar form)</td>
<td>SQSHLU on page C7-2731</td>
</tr>
<tr>
<td>SQSHRN, SQSHRN2</td>
<td>Signed saturating shift right narrow immediate (vector and scalar form)</td>
<td>SQSHRN, SQSHRN2 on page C7-2734</td>
</tr>
<tr>
<td>SQSHRUN, SQSHRUN2</td>
<td>Signed saturating shift right unsigned narrow immediate (vector and scalar form)</td>
<td>SQSHRUN, SQSHRUN2 on page C7-2737</td>
</tr>
<tr>
<td>SRI</td>
<td>Shift right and insert immediate (vector and scalar form)</td>
<td>SRI on page C7-2750</td>
</tr>
<tr>
<td>SRSHR</td>
<td>Signed rounding shift right immediate (vector and scalar form)</td>
<td>SRSHR on page C7-2755</td>
</tr>
<tr>
<td>SRSRA</td>
<td>Signed rounding shift right and accumulate immediate (vector and scalar form)</td>
<td>SRSRA on page C7-2758.</td>
</tr>
<tr>
<td>SSHLL, SSHLL2</td>
<td>Signed shift left long immediate (vector form)</td>
<td>SSHLL, SSHLL2 on page C7-2764</td>
</tr>
<tr>
<td>SSHR</td>
<td>Signed shift right immediate (vector and scalar form)</td>
<td>SSHR on page C7-2766</td>
</tr>
<tr>
<td>SSRA</td>
<td>Signed integer shift right and accumulate immediate (vector and scalar form)</td>
<td>SSRA on page C7-2769</td>
</tr>
<tr>
<td>SXTL, SXTL2</td>
<td>Signed integer extend (vector only)</td>
<td>SXTL, SXTL2 on page C7-2827</td>
</tr>
<tr>
<td>UQRSHRN, UQRSHRN2</td>
<td>Unsigned saturating rounded shift right narrow immediate (vector and scalar form)</td>
<td>UQRSHRN, UQRSHRN2 on page C7-2908</td>
</tr>
<tr>
<td>UQSHL</td>
<td>Unsigned saturating shift left immediate (vector and scalar form)</td>
<td>UQSHL (immediate) on page C7-2911</td>
</tr>
<tr>
<td>UQSHRN, UQSHRN2</td>
<td>Unsigned saturating shift right narrow immediate (vector and scalar form)</td>
<td>UQSHRN, UQSHRN2 on page C7-2917</td>
</tr>
<tr>
<td>URSHR</td>
<td>Unsigned rounding shift right immediate (vector and scalar form)</td>
<td>URSHR on page C7-2930</td>
</tr>
<tr>
<td>URSRA</td>
<td>Unsigned integer rounding shift right and accumulate immediate (vector and scalar form)</td>
<td>URSRA on page C7-2934</td>
</tr>
<tr>
<td>USHLL, USHLL2</td>
<td>Unsigned shift left long immediate (vector form)</td>
<td>USHLL, USHLL2 on page C7-2944</td>
</tr>
<tr>
<td>USHR</td>
<td>Unsigned shift right immediate (vector and scalar form)</td>
<td>USHR on page C7-2946</td>
</tr>
<tr>
<td>USRA</td>
<td>Unsigned shift right and accumulate immediate (vector and scalar form)</td>
<td>USRA on page C7-2952</td>
</tr>
<tr>
<td>UXTL, UXTL2</td>
<td>Unsigned integer extend (vector only)</td>
<td>UXTL, UXTL2 on page C7-2959</td>
</tr>
</tbody>
</table>

### C3.6.21 SIMD floating-point and integer conversion

The SIMD floating-point and integer conversion instructions generate the Invalid Operation floating-point exception in response to a floating-point input of NaN, infinity, or a numerical value that cannot be represented within the destination register. An out of range integer or a fixed-point result is saturated to the size of the destination register. A numeric result that differs from the input raises the Inexact floating-point exception.
Table C3-100 on page C3-329 shows the SIMD floating-point and integer conversion instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCVTAS</td>
<td>Floating-point convert to signed integer, rounding to nearest with ties to away (vector and scalar form)</td>
<td>FCVTAS (vector) on page C7-2198</td>
</tr>
<tr>
<td>FCVTAU</td>
<td>Floating-point convert to unsigned integer, rounding to nearest with ties to away (vector and scalar form)</td>
<td>FCVTAU (vector) on page C7-2203</td>
</tr>
<tr>
<td>FCVTS</td>
<td>Floating-point convert to signed integer, rounding toward minus infinity (vector and scalar form)</td>
<td>FCVTS (vector) on page C7-2210</td>
</tr>
<tr>
<td>FCVTU</td>
<td>Floating-point convert to unsigned integer, rounding toward minus infinity (vector and scalar form)</td>
<td>FCVTU (vector) on page C7-2215</td>
</tr>
<tr>
<td>FCVTNS</td>
<td>Floating-point convert to signed integer, rounding to nearest with ties to even (vector and scalar form)</td>
<td>FCVTNS (vector) on page C7-2222</td>
</tr>
<tr>
<td>FCVTNU</td>
<td>Floating-point convert to unsigned integer, rounding to nearest with ties to even (vector and scalar form)</td>
<td>FCVTNU (vector) on page C7-2227</td>
</tr>
<tr>
<td>FCVTPS</td>
<td>Floating-point convert to signed integer, rounding toward positive infinity (vector and scalar form)</td>
<td>FCVTPS (vector) on page C7-2232</td>
</tr>
<tr>
<td>FCVTPU</td>
<td>Floating-point convert to unsigned integer, rounding toward positive infinity (vector and scalar form)</td>
<td>FCVTPU (vector) on page C7-2237</td>
</tr>
<tr>
<td>FCVTZS</td>
<td>Floating-point convert to signed integer, rounding toward zero (vector and scalar form)</td>
<td>FCVTZS (vector, integer) on page C7-2248</td>
</tr>
<tr>
<td></td>
<td>Floating-point convert to signed fixed-point, rounding toward zero (vector and scalar form)</td>
<td>FCVTZS (vector, fixed-point) on page C7-2245</td>
</tr>
<tr>
<td>FCVTZU</td>
<td>Floating-point convert to unsigned integer, rounding toward zero (vector and scalar form)</td>
<td>FCVTZU (vector, integer) on page C7-2258</td>
</tr>
<tr>
<td></td>
<td>Floating-point convert to unsigned fixed-point, rounding toward zero, (vector and scalar form)</td>
<td>FCVTZU (vector, fixed-point) on page C7-2255</td>
</tr>
<tr>
<td>SCVTF</td>
<td>Signed integer convert to floating-point (vector and scalar form)</td>
<td>SCVTF (vector, integer) on page C7-2575</td>
</tr>
<tr>
<td></td>
<td>Signed fixed-point convert to floating-point (vector and scalar form)</td>
<td>SCVTF (vector, fixed-point) on page C7-2572</td>
</tr>
<tr>
<td>UCVTF</td>
<td>Unsigned integer convert to floating-point (vector and scalar form)</td>
<td>UCVTF (vector, integer) on page C7-2858</td>
</tr>
<tr>
<td></td>
<td>Unsigned fixed-point convert to floating-point (vector and scalar form)</td>
<td>UCVTF (vector, fixed-point) on page C7-2855</td>
</tr>
</tbody>
</table>

**C3.6.22 SIMD reduce (across vector lanes)**

The SIMD reduce (across vector lanes) instructions perform arithmetic operations horizontally, that is across all lanes of the input vector. They deliver a single scalar result.
Table C3-101 shows the SIMD reduce (across vector lanes) instructions.

### Table C3-101 SIMD reduce (across vector lanes) instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDV</td>
<td>Add (across vector)</td>
<td>ADDV on page C7-2053</td>
</tr>
<tr>
<td>FMAXNMV</td>
<td>Floating-point maximum number (across vector)</td>
<td>FMAXNMV on page C7-2284</td>
</tr>
<tr>
<td>FMAXV</td>
<td>Floating-point maximum (across vector)</td>
<td>FMAXV on page C7-2290</td>
</tr>
<tr>
<td>FMINNMV</td>
<td>Floating-point minimum number (across vector)</td>
<td>FMINNMV on page C7-2304</td>
</tr>
<tr>
<td>FMINV</td>
<td>Floating-point minimum (across vector)</td>
<td>FMINV on page C7-2310</td>
</tr>
<tr>
<td>SADDLV</td>
<td>Signed add long (across vector)</td>
<td>SADDLV on page C7-2568</td>
</tr>
<tr>
<td>SMAXV</td>
<td>Signed maximum (across vector)</td>
<td>SMAXV on page C7-2640</td>
</tr>
<tr>
<td>SMINV</td>
<td>Signed minimum (across vector)</td>
<td>SMINV on page C7-2646</td>
</tr>
<tr>
<td>UADDLV</td>
<td>Unsigned add long (across vector)</td>
<td>UADDLV on page C7-2851</td>
</tr>
<tr>
<td>UMAXV</td>
<td>Unsigned maximum (across vector)</td>
<td>UMAXV on page C7-2877</td>
</tr>
<tr>
<td>UMINV</td>
<td>Unsigned minimum (across vector)</td>
<td>UMINV on page C7-2883</td>
</tr>
</tbody>
</table>

**C3.6.23 SIMD pairwise arithmetic**

The SIMD pairwise arithmetic instructions perform operations on pairs of adjacent elements and deliver a vector result.

Table C3-102 on page C3-330 shows the SIMD pairwise arithmetic instructions.

### Table C3-102 SIMD pairwise arithmetic instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDP</td>
<td>Add pairwise (vector and scalar form)</td>
<td>ADDP (vector) on page C7-2051</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADDP (scalar) on page C7-2049</td>
</tr>
<tr>
<td>FADDP</td>
<td>Floating-point add pairwise (vector and scalar form)</td>
<td>FADDP (vector) on page C7-2149</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FADDP (scalar) on page C7-2147</td>
</tr>
<tr>
<td>FMAXNMP</td>
<td>Floating-point maximum number pairwise (vector and scalar form)</td>
<td>FMAXNMP (vector) on page C7-2282</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FMAXNMP (scalar) on page C7-2280</td>
</tr>
<tr>
<td>FMAXP</td>
<td>Floating-point maximum pairwise (vector and scalar form)</td>
<td>FMAXP (vector) on page C7-2288</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FMAXP (scalar) on page C7-2286</td>
</tr>
<tr>
<td>FMINNMP</td>
<td>Floating-point minimum number pairwise (vector and scalar form)</td>
<td>FMINNMP (vector) on page C7-2302</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FMINNMP (scalar) on page C7-2300</td>
</tr>
<tr>
<td>FMINP</td>
<td>Floating-point minimum pairwise (vector and scalar form)</td>
<td>FMINP (vector) on page C7-2308</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FMINP (scalar) on page C7-2306</td>
</tr>
<tr>
<td>SMAXP</td>
<td>Signed maximum pairwise</td>
<td>SMAXP on page C7-2638</td>
</tr>
</tbody>
</table>
C3.6.24 SIMD integer dot product

FEAT_DotProd provides SIMD instructions that perform the dot product of the four 8-bit subelements of the 32-bit elements of one vector with the four 8-bit subelements of a second vector. It provides two forms of the instructions, each with signed and unsigned versions:

Vector form The dot product is calculated for each element of the first vector with the corresponding element of the second vector.

Indexed form The dot product is calculated for each element of the first vector with the element of the second vector that is indicated by the index argument to the instruction.

Note That is, a single element from the second vector is used, and the dot product is calculated between each element of the first vector and this single element from the second vector.

Table C3-103 on page C3-331 shows the SIMD integer dot product instructions.

C3.6.25 SIMD table lookup

Table C3-104 on page C3-331 shows the SIMD table lookup instructions.
C3.6.26   SIMD complex number arithmetic

FEAT_FCMA provides SIMD instructions that perform arithmetic on complex numbers held in element pairs in vector registers, where the less significant element of the pair contains the real component and the more significant element contains the imaginary component.

These instructions provide double-precision and single-precision versions. If FEAT_FP16 is implemented they also provide half-precision versions, otherwise the half-precision encodings are UNDEFINED.

Table C3-105 on page C3-332 shows the FEAT_FCMA SIMD instructions.

Table C3-105 SIMD complex number arithmetic instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCADD</td>
<td>Floating-point complex add</td>
<td>FCADD on page C7-2151</td>
</tr>
<tr>
<td>FCMLA</td>
<td>Floating-point complex multiply accumulate (vector form)</td>
<td>FCMLA on page C7-2181</td>
</tr>
<tr>
<td>FCMLA (by element)</td>
<td>Floating-point complex multiply accumulate (indexed form)</td>
<td>FCMLA (by element) on page C7-2178</td>
</tr>
</tbody>
</table>

A pair of FCMLA instructions can be used to perform a complex number multiplication. This is demonstrated in Complex multiplication on page K12-11746.

C3.6.27   SIMD BFloat16

If FEAT_BF16 is implemented, the instructions in this section are available.

SIMD BFloat16 floating-point multiply-add

The BFloat16 floating-point multiply-add instructions perform an implicit conversion of the bottom (even-numbered) or top (odd-numbered) BFloat16 source elements to IEEE 754 single-precision floating-point format before performing a fused multiply-add without intermediate rounding to the overlapping single-precision destination element. These instructions follow the normal floating-point behaviors that apply to single-precision arithmetic, controlled by the Effective value of the FPCR, and captured in the FPSR cumulative exception bits.

Table C3-106 SIMD BFloat16 floating-point multiply-add instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFMLALB</td>
<td>BFloat16 floating-point widening multiply-add long bottom (vector and indexed forms)</td>
<td>BFMLALB, BFMLALT (vector) on page C7-2069</td>
</tr>
<tr>
<td>BFMLALT</td>
<td>BFloat16 floating-point widening multiply-add long top (vector and index forms)</td>
<td>BFMLALB, BFMLALT (vector) on page C7-2069</td>
</tr>
</tbody>
</table>

SIMD BFloat16 floating-point dot product

The BFloat16 floating-point dot product instruction, BFDP, performs an implicit conversion of vectors of BFloat16 input values to IEEE 754 single-precision floating-point format.

The BFloat16 dot product instructions delimit their source vectors into pairs of BFloat16 elements.
If FEAT_EBF16 is not implemented or FPCR.EBF is 0, irrespective of the other control bits in the FPCR, a $\texttt{BFDOT}$ instruction:

- Performs an unfused, widening sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the corresponding or indexed pair in the second source vector. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Does not modify the FPSR cumulative exception bits.
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits are all zero.
- Flashes denormalized inputs and results to zero, as if FPCR.FIZ and FPCR.FIZ are 0.
- Generates only the default NaN, as if FPCR.DN is 1.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, a $\texttt{BFDOT}$ instruction:

- Performs a fused, widening sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the corresponding or indexed pair in the second source vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Generates only the default NaN, as if FPCR.DN is 1.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as controlled by the Effective value of the FPCR in the current execution mode, and captured in the FPSR cumulative exception bits.

### Table C3-107 SIMD BFloat16 floating-point dot product instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\texttt{BFDOT}$</td>
<td>BFloat16 floating-point dot product (vector and indexed forms)</td>
<td>$\texttt{BFDOT}$ (vector) on page C7-2065&lt;br&gt; $\texttt{BFDOT}$ (by element) on page C7-2063</td>
</tr>
</tbody>
</table>

### SIMD BFloat16 floating-point matrix multiply

The BFloat16 floating-point matrix multiply instruction, $\texttt{BFMMLA}$, treats each source and destination vector as a single matrix:

- The first source vector 2x4 BF16 matrix is organized in row-by-row order.
- The second source vector 4x2 BF16 matrix is organized in a column-by-column order.
- The destination vector 2x2 single-precision matrix is organized in row-by-row order.

One matrix multiplication is performed per source vector and accumulated into the corresponding destination vector. This corresponds to accumulating two 2-way BFloat16 widening dot products into each single-precision destination element, following the numeric behaviors described for $\texttt{BFDOT}$ instruction in SIMD BFloat16 floating-point dot product on page C3-332.

### Table C3-108 SIMD BFloat16 floating-point matrix multiply instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\texttt{BFMMLA}$</td>
<td>BFloat16 floating-point matrix multiply-accumulate into 2x2 matrix</td>
<td>$\texttt{BFMMLA}$ on page C7-2070</td>
</tr>
</tbody>
</table>
SIMD BFloat16 floating-point convert

The BFloat16 floating-point convert instructions perform accurately rounded down-conversion of IEEE 754 single-precision source vector elements to BFloat16 format.

The `BFCVTN` instruction places its half-width BFloat16 results in the bottom or even-numbered 16-bit elements of the destination vector, and sets the top or odd-numbered elements to zero.

The `BFCVTN2` instruction places its half-width BFloat16 results in the top or odd-numbered 16-bit elements of the destination vector, leaving the bottom or even-numbered elements unchanged.

These instructions follow the normal floating-point behaviors that apply to single-precision arithmetic, controlled by the Effective value of the FPCR, and captured in the FPSR cumulative exception bits.

### Table C3-109 SIMD BFloat16 floating-point convert instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFCVTN, BFCVTN2</td>
<td>Floating-point convert from single-precision to BFloat16 format (vector form)</td>
<td><code>BFCVTN, BFCVTN2 on page C7-2062</code></td>
</tr>
</tbody>
</table>

#### C3.6.28 SIMD integer matrix multiply-accumulate

The integer matrix multiply-accumulate instructions are provided by `FEAT_I8MM`.

The integer matrix multiply-accumulate instructions treat each source and destination vector as a single matrix:

- The first source vector 2x8 matrix is organized in row-by-row order.
- The second source vector 8x2 matrix is organized in column-by-column order.
- The destination vector 2x2 matrix is organized in row-by-row order.

One matrix multiplication is performed per segment.

Table C3-110 on page C3-334 shows these instructions.

### Table C3-110 Matrix multiply SIMD instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMLA</td>
<td>Widening 8-bit signed integer matrix multiply-accumulate into 2x2 matrix</td>
<td><code>SMLA (vector) on page C7-2658</code></td>
</tr>
<tr>
<td>UMLA</td>
<td>Widening 8-bit unsigned integer matrix multiply-accumulate into 2x2 matrix</td>
<td><code>UMLA (vector) on page C7-2895</code></td>
</tr>
<tr>
<td>USMMLA</td>
<td>Widening 8-bit mixed sign integer matrix multiply-accumulate into 2x2 matrix</td>
<td><code>USMMLA (vector) on page C7-2949</code></td>
</tr>
</tbody>
</table>

#### C3.6.29 The Cryptographic Extension

The instructions provided by the OPTIONAL Armv8.0 Cryptographic Extension use the SIMD and floating-point register file. For more information about the functions they provide, see:

- Announcing the Advanced Encryption Standard.
- *The Galois/Counter Mode of Operation*.
- Announcing the Secure Hash Standard.
Table C3-111 shows the Armv8.0 Cryptographic Extension instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Description</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>AESD</td>
<td>AES single round decryption</td>
<td>(AESD) on page C7-2055</td>
</tr>
<tr>
<td>AESE</td>
<td>AES single round encryption</td>
<td>(AESE) on page C7-2056</td>
</tr>
<tr>
<td>AESIMC</td>
<td>AES inverse mix columns</td>
<td>(AESIMC) on page C7-2057</td>
</tr>
<tr>
<td>AESMC</td>
<td>AES mix columns</td>
<td>(AESMC) on page C7-2058</td>
</tr>
<tr>
<td>PMULL</td>
<td>Polynomial multiply long</td>
<td>(PMULL, PMULL2) on page C7-2537^a</td>
</tr>
<tr>
<td>SHA1C</td>
<td>SHA1 hash update (choose)</td>
<td>(SHA1C) on page C7-2586</td>
</tr>
<tr>
<td>SHA1H</td>
<td>SHA1 fixed rotate</td>
<td>(SHA1H) on page C7-2587</td>
</tr>
<tr>
<td>SHA1M</td>
<td>SHA1 hash update (majority)</td>
<td>(SHA1M) on page C7-2588</td>
</tr>
<tr>
<td>SHA1P</td>
<td>SHA1 hash update (parity)</td>
<td>(SHA1P) on page C7-2589</td>
</tr>
<tr>
<td>SHA1SU0</td>
<td>SHA1 schedule update 0</td>
<td>(SHA1SU0) on page C7-2590</td>
</tr>
<tr>
<td>SHA1SU1</td>
<td>SHA1 schedule update 1</td>
<td>(SHA1SU1) on page C7-2591</td>
</tr>
<tr>
<td>SHA256H</td>
<td>SHA256 hash update, part 1</td>
<td>(SHA256H) on page C7-2593</td>
</tr>
<tr>
<td>SHA256H2</td>
<td>SHA256 hash update, part 2</td>
<td>(SHA256H2) on page C7-2592</td>
</tr>
<tr>
<td>SHA256SU0</td>
<td>SHA256 schedule update 0</td>
<td>(SHA256SU0) on page C7-2594</td>
</tr>
<tr>
<td>SHA256SU1</td>
<td>SHA256 schedule update 1</td>
<td>(SHA256SU1) on page C7-2595</td>
</tr>
</tbody>
</table>

^a. The Cryptographic Extension adds the variant of the instruction that operates on two 64-bit polynomials.

See *The Armv8 Cryptographic Extension* on page A2-80 for information about the permitted implementation options for the Cryptographic Extension.

**Armv8.2 extensions to the Cryptographic Extension**

Armv8.2 supports the following **OPTIONAL** extensions to the Cryptographic Extension:

- \(FEAT\_SHA512, SHA2-512\) functionality.
- \(FEAT\_SHA3, SHA3\) functionality on page C3-336.
- \(FEAT\_SM3, SM3\) functionality on page C3-337.
- \(FEAT\_SM4, SM4\) functionality on page C3-337.

**\(FEAT\_SHA512, SHA2-512\) functionality**

\(FEAT\_SHA512\) provides instructions to accelerate the SHA-2 hash algorithm using a digest that is larger than 256 bits. The relevant standards are SHA-384, SHA-512, SHA-512\|224 and SHA-512\|256. These are all based on the SHA-512 computation, and therefore this set of instructions is described as the \(SHA512\) instructions.

Implementation of \(FEAT\_SHA512\) requires the implementation of the SHA1 and SHA2-256 instructions from the Armv8.0 Cryptographic Extension.

--- **Note** ---

Implementation of \(FEAT\_SHA512\) does not require the implementation of the AES instructions, and the 64-bit polynomial variants of the PMULL instructions, from the Armv8.0 Cryptographic Extension.
When `FEAT_SHA512` is implemented, the value of `ID_AA64ISAR0_EL1.SHA2` is 0b00010, indicating support for the SHA512 instructions.

Table C3-112 shows the `FEAT_SHA512` instructions:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA512H</td>
<td>SHA512 Hash update part 1</td>
<td>SHA512H on page C7-2597</td>
</tr>
<tr>
<td>SHA512H2</td>
<td>SHA512 Hash update part 2</td>
<td>SHA512H2 on page C7-2599</td>
</tr>
<tr>
<td>SHA512SU0</td>
<td>SHA512 Schedule Update 0</td>
<td>SHA512SU0 on page C7-2601</td>
</tr>
<tr>
<td>SHA512SU1</td>
<td>SHA512 Schedule Update 1</td>
<td>SHA512SU1 on page C7-2602</td>
</tr>
</tbody>
</table>

*Use of the SHA512 instructions* on page K12-11748 shows an example of the use of these instructions to calculate a SHA512 hash iteration. This example code is not part of the architectural definition of these instructions.

**FEAT_SHA3, SHA3 functionality**

`FEAT_SHA3` provides instructions to accelerate the SHA-3 hash algorithm. This set of instructions is described as the SHA3 instructions.

--- Note ---

Implementation of `FEAT_SHA3` does not require the implementation of the AES instructions, and the 64-bit polynomial variants of the `PMULL` instructions, from the Armv8.0 Cryptographic Extension.

When `FEAT_SHA3` is implemented, the value of `ID_AA64ISAR0_EL1.SHA3` is 0b00001, indicating support for the SHA3 instructions.

Table C3-113 shows the `FEAT_SHA3` instructions. The SHA-3 hash algorithm is based on a running digest of 1600 bytes, arranged as a five by five array of 64-bit registers. The Arm acceleration of these instructions is based on mapping the 25 64-bit values into 25 vector registers, with each 64-bit value occupying the same 64-bit element in each vector. A series of transformations is performed on these registers as part of a round of the SHA-3 hash calculation.

The SIMD nature of the vector registers means the acceleration can compute two parallel SHA3 hash calculations, where one calculation is performed using the zeroth 64-bit element of each vector, and the other calculation is performed using the first 64-bit element of each vector.

To provide acceleration where the SIMD calculation is not required, the instructions provide variants that operate only on the zeroth 64-bit elements. These are provided as a power optimization.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCAX</td>
<td>Bit Clear and Exclusive OR</td>
<td>BCAX on page C7-2060</td>
</tr>
<tr>
<td>EOR3</td>
<td>Three-way Exclusive OR</td>
<td>EOR3 on page C7-2125</td>
</tr>
<tr>
<td>RAXI</td>
<td>Rotate and Exclusive OR</td>
<td>RAXI on page C7-2541</td>
</tr>
<tr>
<td>XAR</td>
<td>Exclusive OR and Rotate</td>
<td>XAR on page C7-2965</td>
</tr>
</tbody>
</table>

*Use of the SHA3 instructions* on page K12-11749 shows an example of the use of these instructions to calculate the combined theta, phi, rho and chi operations of a SHA3 iteration. This example code is not part of the architectural definition of these instructions.
**FEAT_SM3, SM3 functionality**

FEAT_SM3 provides instructions to accelerate the SM3 hash algorithm, the standard Chinese hash algorithm. These are described as the SM3 instructions.

FEAT_SM3 can be implemented independently of any part of the Armv8.0 Cryptographic Extension, and independently of FEAT_SHA512.

--- Note ---

This means that Armv8.2 permits an implementation of the Cryptographic Extension that provides only the FEAT_SM3 functionality.

When FEAT_SM3 is implemented, the value of ID_AA64ISAR0_EL1.SM3 is 0b0001, indicating support for the SM3 instructions.

Table C3-114 shows the FEAT_SM3 instructions. The SM3 algorithm computes a digest of 256 bits, which can be held in two vector registers. The SM3 instructions include instructions to accelerate the computation of the hash and the schedule update.

--- Note ---

The SM3 instruction names refer to intermediate variables defined as part of the SM3 Cryptographic Hash Algorithm specification.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM3SS1</td>
<td>SM3 SS1 calculation</td>
<td>SM3SS1 on page C7-2622</td>
</tr>
<tr>
<td>SM3TT1A</td>
<td>SM3 TT1 calculation, part A</td>
<td>SM3TT1A on page C7-2624</td>
</tr>
<tr>
<td>SM3TT1B</td>
<td>SM3 TT1 calculation, part B</td>
<td>SM3TT1B on page C7-2626</td>
</tr>
<tr>
<td>SM3TT2A</td>
<td>SM3 TT2 calculation, part A</td>
<td>SM3TT2A on page C7-2628</td>
</tr>
<tr>
<td>SM3TT2B</td>
<td>SM3 TT2 calculation, part B</td>
<td>SM3TT2B on page C7-2630</td>
</tr>
<tr>
<td>SM3PARTW1</td>
<td>SM3 PARTW calculation, part 1</td>
<td>SM3PARTW1 on page C7-2618</td>
</tr>
<tr>
<td>SM3PARTW2</td>
<td>SM3 PARTW calculation, part 1</td>
<td>SM3PARTW2 on page C7-2620</td>
</tr>
</tbody>
</table>

Use of the SM3 instructions on page K12-11750 shows an example of the use of these instructions to generate an SM3 hash. This example code is not part of the architectural definition of these instructions.

**FEAT_SM4, SM4 functionality**

FEAT_SM4 provides instruction to accelerate the SM4 encryption algorithm, the standard Chinese encryption algorithm. This set of instructions is described as the SM4 instructions.

FEAT_SM4 can be implemented independently of any part of the Armv8.0 Cryptographic Extension, and independently of FEAT_SHA3.

--- Note ---

This means that Armv8.2 permits an implementation of the Cryptographic Extension that provides only the FEAT_SM4 functionality.

When FEAT_SM4 is implemented, the value of ID_AA64ISAR0_EL1.SM4 is 0b00001, indicating support for the SM4 instructions.
Table C3-115 shows the FEAT_SM4 instructions. The SM4 algorithm is 128-bit wide block cipher. The SM4E instruction accelerates a single round of encryption or decryption, and the SM4EKEY instruction accelerates a single round of key generation:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM4E</td>
<td>SM4 Encrypt</td>
<td>SM4E on page C7-2632</td>
</tr>
<tr>
<td>SM4EKEY</td>
<td>SM4 Key</td>
<td>SM4EKEY on page C7-2634</td>
</tr>
</tbody>
</table>

Use of the SM4 instructions on page K12-11752 shows an example of the use of these instructions to perform SM4 encryption and decryption. This example code is not part of the architectural definition of these instructions.
C3.7 Data processing - SVE

The following subsections describe the SVE processing instructions:

- Vector move operations.
- Move operations on page C3-362.
- Reduction operations on page C3-366.

C3.7.1 Vector move operations

Element move and broadcast

The Element move and broadcast instructions copy data from scalar registers, immediate values, and other vectors to selected vector elements. The copied data might be in an integer or floating-point format.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
</table>
| CPY      | Copy signed integer immediate to vector elements | CPY (immediate, merging) on page C8-3150  
|          |            | CPY (immediate, zeroing) on page C8-3152 |
|          | Copy general-purpose register to vector elements | CPY (scalar) on page C8-3154 |
|          | Copy SIMD&FP scalar register to vector elements | CPY (SIMD&FP scalar) on page C8-3156 |
| DUP      | Broadcast signed immediate to vector elements | DUP (immediate) on page C8-3169 |
|          | Broadcast general-purpose register to vector elements | DUP (scalar) on page C8-3173 |
| FCPY     | Copy 8-bit floating-point immediate to vector elements | FCPY on page C8-3233 |
| FDUP     | Broadcast 8-bit floating-point immediate to vector elements | FDUP on page C8-3257 |
| FMOV     | Move floating-point +0.0 to vector elements (unpredicated) | FMOV (zero, unpredicated) on page C8-3324 |
|          | Move floating-point +0.0 to vector elements (predicated) | FMOV (zero, predicated) on page C8-3322 |
|          | Move 8-bit floating-point immediate to vector elements (unpredicated) | FMOV (immediate, unpredicated) on page C8-3321 |
|          | Move 8-bit floating-point immediate to vector element (predicated) | FMOV (immediate, predicated) on page C8-3320 |
| MOV      | Move signed integer immediate to vector elements (unpredicated) | MOV (immediate, unpredicated) on page C8-3729 |
|          | Move signed integer immediate to vector elements (predicated) | MOV (immediate, predicated, merging) on page C8-3725  
|          | Move signed integer immediate to vector elements (predicated) | MOV (immediate, predicated, zeroing) on page C8-3727 |
|          | Move general-purpose register to vector elements (unpredicated) | MOV (scalar, unpredicated) on page C8-3735 |
|          | Move general-purpose register to vector elements (predicated) | MOV (scalar, predicated) on page C8-3733 |
|          | Move SIMD&FP scalar register to vector elements (unpredicated) | MOV (SIMD&FP scalar, unpredicated) on page C8-3739 |
Table C3-116 Element move and broadcast instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>Move SIMD&amp;FP scalar register to vector elements (predicated)</td>
<td>MOV (SIMD&amp;FP scalar, predicated) on page C8-3737</td>
</tr>
<tr>
<td></td>
<td>Move vector register (unpredicated)</td>
<td>MOV (vector, unpredicated) on page C8-3742</td>
</tr>
<tr>
<td></td>
<td>Move vector register (predicated)</td>
<td>MOV (vector, predicated) on page C8-3741</td>
</tr>
<tr>
<td>SEL</td>
<td>Select vector elements from two vectors</td>
<td>SEL (predicates) on page C8-3921</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SEL (vectors) on page C8-3923</td>
</tr>
</tbody>
</table>

**Integer operations**

The following instructions operate on signed or unsigned integer data within a vector.

**SVE Integer arithmetic**

For binary operations, the Integer arithmetic instructions perform arithmetic operations on a source vector containing integer element values, and a second source vector of either integer element values or an immediate value.

For ternary operations, these instructions perform arithmetic operations on a source vector containing integer element values, a second source vector of either integer element values or an immediate value, and a third source vector containing integer element values.

Table C3-117 Integer arithmetic instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Absolute value</td>
<td>ABS on page C8-2976</td>
</tr>
<tr>
<td>ADD</td>
<td>Add vectors (predicated)</td>
<td>ADD (vectors, predicated) on page C8-2984</td>
</tr>
<tr>
<td></td>
<td>Add vectors (unpredicated)</td>
<td>ADD (vectors, unpredicated) on page C8-2986</td>
</tr>
<tr>
<td></td>
<td>Add immediate</td>
<td>ADD (immediate) on page C8-2982</td>
</tr>
<tr>
<td>CNOT</td>
<td>Logically invert Boolean condition</td>
<td>CNOT on page C8-3140</td>
</tr>
<tr>
<td>MAD</td>
<td>Multiply-add, writing to the multiplicand register</td>
<td>MAD on page C8-3708</td>
</tr>
<tr>
<td>MLA</td>
<td>Multiply-add, writing to the addend register</td>
<td>MLA (indexed) on page C8-3712</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MLA (vectors) on page C8-3715</td>
</tr>
<tr>
<td>MLS</td>
<td>Multiply-subtract, writing to the addend register</td>
<td>MLS (indexed) on page C8-3717</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MLS (vectors) on page C8-3720</td>
</tr>
<tr>
<td>MSB</td>
<td>Multiply-subtract, writing to the multiplicand register</td>
<td>MSB on page C8-3748</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply by immediate</td>
<td>MUL (immediate) on page C8-3750</td>
</tr>
<tr>
<td></td>
<td>Multiply vectors</td>
<td>MUL (vectors, predicated) on page C8-3755</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MUL (vectors, unpredicated) on page C8-3757</td>
</tr>
<tr>
<td>NEG</td>
<td>Negate</td>
<td>NEG on page C8-3765</td>
</tr>
<tr>
<td>SABD</td>
<td>Signed absolute difference</td>
<td>SABD on page C8-3883</td>
</tr>
<tr>
<td>SDIV</td>
<td>Signed divide</td>
<td>SDIV on page C8-3913</td>
</tr>
</tbody>
</table>
### Table C3-117 Integer arithmetic instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDIVR</td>
<td>Signed reversed divide</td>
<td>SDIVR on page C8-3915</td>
</tr>
<tr>
<td>SMAX</td>
<td>Signed maximum with immediate</td>
<td>SMAX (immediate) on page C8-3942</td>
</tr>
<tr>
<td></td>
<td>Signed maximum vectors</td>
<td>SMAX (vectors) on page C8-3944</td>
</tr>
<tr>
<td>SMIN</td>
<td>Signed minimum with immediate</td>
<td>SMIN (immediate) on page C8-3950</td>
</tr>
<tr>
<td></td>
<td>Signed minimum vectors</td>
<td>SMIN (vectors) on page C8-3952</td>
</tr>
<tr>
<td>SMULH</td>
<td>Signed multiply returning high half</td>
<td>SMULH (predicated) on page C8-3976</td>
</tr>
<tr>
<td></td>
<td>SMULH (unpredicated) on page C8-3978</td>
<td></td>
</tr>
<tr>
<td>SQADD</td>
<td>Signed saturating add immediate</td>
<td>SQADD (immediate) on page C8-3992</td>
</tr>
<tr>
<td></td>
<td>Signed saturating add vectors</td>
<td>SQADD (vectors, predicated) on page C8-3994</td>
</tr>
<tr>
<td></td>
<td>SQADD (vectors, unpredicated) on page C8-3996</td>
<td></td>
</tr>
<tr>
<td>SQSUB</td>
<td>Signed saturating subtract immediate</td>
<td>SQSUB (immediate) on page C8-4118</td>
</tr>
<tr>
<td></td>
<td>Signed saturating subtract vectors</td>
<td>SQSUB (vectors, predicated) on page C8-4120</td>
</tr>
<tr>
<td></td>
<td>SQSUB (vectors, unpredicated) on page C8-4122</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract immediate</td>
<td>SUB (immediate) on page C8-4275</td>
</tr>
<tr>
<td></td>
<td>Subtract vectors (predicated)</td>
<td>SUB (vectors, predicated) on page C8-4277</td>
</tr>
<tr>
<td></td>
<td>Subtract vectors (unpredicated)</td>
<td>SUB (vectors, unpredicated) on page C8-4279</td>
</tr>
<tr>
<td>SUBR</td>
<td>Reversed subtract from immediate</td>
<td>SUBR (immediate) on page C8-4285</td>
</tr>
<tr>
<td></td>
<td>Reversed subtract vectors</td>
<td>SUBR (vectors) on page C8-4287</td>
</tr>
<tr>
<td>SXTB</td>
<td>Signed byte extend</td>
<td>SXTB, SXTH, SXTW on page C8-4295</td>
</tr>
<tr>
<td>SXTH</td>
<td>Signed halfword extend</td>
<td>SXTB, SXTH, SXTW on page C8-4295</td>
</tr>
<tr>
<td>SXTW</td>
<td>Signed word extend</td>
<td>SXTB, SXTH, SXTW on page C8-4295</td>
</tr>
<tr>
<td>UABD</td>
<td>Unsigned absolute difference</td>
<td>UABD on page C8-4313</td>
</tr>
<tr>
<td>UDIV</td>
<td>Unsigned divide</td>
<td>UDIV on page C8-4337</td>
</tr>
<tr>
<td>UDIVR</td>
<td>Unsigned reversed divide</td>
<td>UDIVR on page C8-4339</td>
</tr>
<tr>
<td>UMAX</td>
<td>Unsigned maximum with immediate</td>
<td>UMAX (immediate) on page C8-4351</td>
</tr>
<tr>
<td></td>
<td>Unsigned maximum vectors</td>
<td>UMAX (vectors) on page C8-4353</td>
</tr>
<tr>
<td>UMIN</td>
<td>Unsigned minimum with immediate</td>
<td>UMIN (immediate) on page C8-4359</td>
</tr>
<tr>
<td></td>
<td>Unsigned minimum vectors</td>
<td>UMIN (vectors) on page C8-4361</td>
</tr>
<tr>
<td>UMULH</td>
<td>Unsigned multiply returning high half</td>
<td>UMULH (predicated) on page C8-4385</td>
</tr>
<tr>
<td></td>
<td>UMULH (unpredicated) on page C8-4387</td>
<td></td>
</tr>
<tr>
<td>UQADD</td>
<td>Unsigned saturating add immediate</td>
<td>UQADD (immediate) on page C8-4397</td>
</tr>
<tr>
<td></td>
<td>Unsigned saturating add vectors</td>
<td>UQADD (vectors, predicated) on page C8-4399</td>
</tr>
<tr>
<td></td>
<td>UQADD (vectors, unpredicated) on page C8-4401</td>
<td></td>
</tr>
</tbody>
</table>
SVE Integer dot product

The Integer dot product instructions delimit the source vectors into groups of four 8-bit or 16-bit integer elements. Within each group of four elements, the elements in the first source vector are multiplied by the corresponding elements in the second source vector. The resulting widened products are summed and added to the 32-bit or 64-bit element of the accumulator and destination vector that aligns with the group of four elements in the first source vector.

The indexed forms of these instructions specify a single, numbered, group of four elements within each 128-bit segment of the second source vector as the multiplier for all the groups of four elements within the corresponding 128-bit segment of the first source vector.

Table C3-118 Integer dot product instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDOT</td>
<td>Signed dot product by vector</td>
<td>SDOT (vectors) on page C8-3919</td>
</tr>
<tr>
<td></td>
<td>Signed dot product by indexed elements</td>
<td>SDOT (indexed) on page C8-3917</td>
</tr>
<tr>
<td>SUDOT</td>
<td>Signed by unsigned integer dot product by indexed elements</td>
<td>SUDOT on page C8-4289</td>
</tr>
<tr>
<td>UDOT</td>
<td>Unsigned dot product by vector</td>
<td>UDOT (vectors) on page C8-4343</td>
</tr>
<tr>
<td></td>
<td>Unsigned dot product by indexed elements</td>
<td>UDOT (indexed) on page C8-4341</td>
</tr>
<tr>
<td>USDOT</td>
<td>Unsigned by signed integer dot product</td>
<td>USDOT (vectors) on page C8-4483</td>
</tr>
<tr>
<td></td>
<td>Unsigned by signed integer dot product by indexed elements</td>
<td>USDOT (indexed) on page C8-4481</td>
</tr>
</tbody>
</table>

SVE Integer matrix multiply operations

The Integer matrix multiply instructions facilitate matrix multiplication and include integer matrix multiply-accumulate instructions.

The matrix multiply-accumulate instructions delimit source and destination vectors into segments. Within each segment:

- The first source vector 2x8 8-bit matrix is organized in row-by-row order.
- The second source vector 8x2 8-bit matrix is organized in a column-by-column order.
- The destination vector 2x2 32-bit matrix is organized in row-by-row order.

One matrix multiplication is performed per vector segment of 128 bits and accumulated into the destination vector segment.
SVE Integer comparison

The Integer comparison instructions compare Active elements in the first source vector with the corresponding elements in a second vector or with an immediate value. The Boolean result of each comparison is placed in the corresponding element of the destination predicate. Inactive elements in the destination predicate register are set to FALSE. All integer comparisons set the N, Z, and C condition flags based on the predicate result, and set the V flag to zero.

The wide element variants of the compare instructions allow a packed vector of narrower elements to be compared with wider 64-bit elements. These instructions treat the second source vector as having a fixed 64-bit doubleword element size and compare each narrow element of the first source vector with the corresponding vertically-aligned wide element of the second source vector. For example, if the first source vector contained 8-bit byte elements, then 8-bit element[0] to element[7] of the first source vector are compared with 64-bit element[0] of the second source vector, 8-bit element[8] to element[15] with 64-bit element[1], and so on. All 64 bits of the wide elements are significant for the comparison, with the narrow elements being sign-extended or zero-extended to 64 bits as appropriate for the type of comparison.

### Table C3-119 Integer matrix multiply operations

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMMLA</td>
<td>Widening signed 8-bit integer matrix multiply-accumulate into 2x2 matrix</td>
<td>SMMLA on page C8-3974</td>
</tr>
<tr>
<td>UMMLA</td>
<td>Widening unsigned 8-bit integer matrix multiply-accumulate into 2x2 matrix</td>
<td>UMMLA on page C8-4383</td>
</tr>
<tr>
<td>USMMLA</td>
<td>Widening mixed sign 8-bit integer matrix multiply-accumulate into 2x2 matrix</td>
<td>USMMLA on page C8-4489</td>
</tr>
</tbody>
</table>

### Table C3-120 Integer comparison instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPEQ</td>
<td>Compare signed equal to immediate</td>
<td>CMP&lt;cc&gt; (immediate) Equal on page C8-3116</td>
</tr>
<tr>
<td></td>
<td>Compare signed equal to wide elements</td>
<td>CMP&lt;cc&gt; (wide elements) Equal on page C8-3126</td>
</tr>
<tr>
<td></td>
<td>Compare signed equal to vector</td>
<td>CMP&lt;cc&gt; (vectors) Equal on page C8-3122</td>
</tr>
<tr>
<td>CMPGE</td>
<td>Compare signed greater than or equal to immediate</td>
<td>CMP&lt;cc&gt; (immediate) Greater than or equal on page C8-3117</td>
</tr>
<tr>
<td></td>
<td>Compare signed greater than or equal to wide elements</td>
<td>CMP&lt;cc&gt; (wide elements) Greater than or equal on page C8-3127</td>
</tr>
<tr>
<td></td>
<td>Compare signed greater than or equal to vector</td>
<td>CMP&lt;cc&gt; (vectors) Greater than or equal on page C8-3123</td>
</tr>
<tr>
<td>CMPGT</td>
<td>Compare signed greater than immediate</td>
<td>CMP&lt;cc&gt; (immediate) Greater than on page C8-3116</td>
</tr>
<tr>
<td></td>
<td>Compare signed greater than wide elements</td>
<td>CMP&lt;cc&gt; (wide elements) Greater than on page C8-3126</td>
</tr>
<tr>
<td></td>
<td>Compare signed greater than vector</td>
<td>CMP&lt;cc&gt; (vectors) Greater than on page C8-3122</td>
</tr>
<tr>
<td>CMPI</td>
<td>Compare unsigned higher than immediate</td>
<td>CMP&lt;cc&gt; (immediate) Higher on page C8-3117</td>
</tr>
<tr>
<td></td>
<td>Compare unsigned higher than wide elements</td>
<td>CMP&lt;cc&gt; (wide elements) Higher on page C8-3127</td>
</tr>
<tr>
<td></td>
<td>Compare unsigned higher than vector</td>
<td>CMP&lt;cc&gt; (vectors) Higher on page C8-3123</td>
</tr>
</tbody>
</table>
SVE Vector address calculation

The Vector address calculation instructions compute vectors of addresses and addresses of vectors. This includes instructions to add a multiple of the current vector length or predicate register length, in bytes, to a general-purpose register.

The **ADR** instruction is an integer arithmetic operation that is used to calculate a vector of 64-bit or 32-bit addresses.

The **ADR** destination vector elements are computed by the addition of the corresponding elements in the source vectors, with an optional sign or zero extension and optional bitwise left shift of 1-3 bits applied to the final operands. This can be considered as the addition of a vector base and a scaled vector index.

The **ADR** instruction computes a vector of 32-bit addresses by the addition of a 32-bit base and a scaled 32-bit unsigned index. The **ADR** instruction computes a vector of 64-bit addresses by one of:

- Addition of a 64-bit base and a scaled 64-bit unsigned index.
- Addition of a 64-bit base and a scaled, zero-extended 32-bit index.
• Addition of a 64-bit base and a scaled, sign-extended 32-bit index.

### Table C3-121 Vector address calculation instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPL</td>
<td>Add multiple of predicate register length, in bytes, to scalar register</td>
<td>ADDPL on page C8-2994</td>
</tr>
<tr>
<td>ADDVL</td>
<td>Add multiple of vector length, in bytes, to scalar register</td>
<td>ADDVL on page C8-2995</td>
</tr>
<tr>
<td>ADR</td>
<td>Compute vector of addresses</td>
<td>ADR on page C8-2996</td>
</tr>
<tr>
<td>RDVL</td>
<td>Read multiple of vector register length, in bytes, to scalar register</td>
<td>RDVL on page C8-3861</td>
</tr>
</tbody>
</table>

### SVE Bitwise logical operations

The Bitwise logical operations instructions perform bitwise logical operations on vectors. Where operations are unpredicated, the operations are independent of the element size.

### Table C3-122 Bitwise logical operations

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Bitwise AND vectors (predicated)</td>
<td>AND (vectors, predicated) on page C8-3011</td>
</tr>
<tr>
<td></td>
<td>Bitwise AND vectors (unpredicated)</td>
<td>AND (vectors, unpredicated) on page C8-3013</td>
</tr>
<tr>
<td></td>
<td>Bitwise AND with immediate</td>
<td>AND (immediate) on page C8-3007</td>
</tr>
<tr>
<td>BIC</td>
<td>Bitwise clear with vector (predicated)</td>
<td>BIC (vectors, predicated) on page C8-3061</td>
</tr>
<tr>
<td></td>
<td>Bitwise clear with vector (unpredicated)</td>
<td>BIC (vectors, unpredicated) on page C8-3063</td>
</tr>
<tr>
<td></td>
<td>Bitwise clear using immediate</td>
<td>BIC (immediate) on page C8-3057</td>
</tr>
<tr>
<td>DUPM</td>
<td>Broadcast bitmask immediate to vector (unpredicated)</td>
<td>DUPM on page C8-3175</td>
</tr>
<tr>
<td>EON</td>
<td>Bitwise exclusive OR with inverted immediate</td>
<td>EON on page C8-3177</td>
</tr>
<tr>
<td>EOR</td>
<td>Bitwise exclusive OR vectors (predicated)</td>
<td>EOR (vectors, predicated) on page C8-3183</td>
</tr>
<tr>
<td></td>
<td>Bitwise exclusive OR vectors (unpredicated)</td>
<td>EOR (vectors, unpredicated) on page C8-3185</td>
</tr>
<tr>
<td></td>
<td>Bitwise exclusive OR with immediate</td>
<td>EOR (immediate) on page C8-3179</td>
</tr>
<tr>
<td>MOV</td>
<td>Move bitmask immediate to vector</td>
<td>MOV on page C8-3722</td>
</tr>
<tr>
<td></td>
<td>Move vector register</td>
<td>MOV (vector, unpredicated) on page C8-3742</td>
</tr>
<tr>
<td>NOT</td>
<td>Bitwise invert vector</td>
<td>NOT (vector) on page C8-3774</td>
</tr>
<tr>
<td>ORN</td>
<td>Bitwise OR with inverted immediate</td>
<td>ORN (immediate) on page C8-3777</td>
</tr>
<tr>
<td>ORR</td>
<td>Bitwise OR vectors (predicated)</td>
<td>ORR (vectors, predicated) on page C8-3787</td>
</tr>
<tr>
<td></td>
<td>Bitwise OR vectors (unpredicated)</td>
<td>ORR (vectors, unpredicated) on page C8-3789</td>
</tr>
<tr>
<td></td>
<td>Bitwise OR with immediate</td>
<td>ORR (immediate) on page C8-3783</td>
</tr>
</tbody>
</table>

### SVE Bitwise shift, reverse, and count

Bitwise shifts, reversals, and counts within vector elements.

Shift counts saturate at the number of bits per element, rather than being used modulo the element size. If modulo behavior is required, then the modulus must be computed separately.
The wide element variants of the bitwise shift instructions allow a packed vector of narrower elements to be shifted by wider 64-bit shift amounts. These instructions treat the second source vector as having a fixed 64-bit doubleword element size and shift each narrow element of the first source vector by the corresponding vertically-aligned wide element of the second source vector. For example, if the first source vector contained 8-bit byte elements, then 8-bit element[0] to element[7] of the first vector are shifted by 64-bit element[0] of the second source vector, 8-bit element [8] to element[15] by 64-bit element[1], and so on. All 64 bits of the wide shift amount are significant.

Table C3-123 Bitwise shift, reverse, and count instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR</td>
<td>Arithmetic shift right by immediate (predicated)</td>
<td>ASR (immediate, predicated) on page C8-3018</td>
</tr>
<tr>
<td></td>
<td>Arithmetic shift right by immediate (unpredicated)</td>
<td>ASR (immediate, unpredicated) on page C8-3020</td>
</tr>
<tr>
<td></td>
<td>Arithmetic shift right by wide elements (predicated)</td>
<td>ASR (wide elements, predicated) on page C8-3024</td>
</tr>
<tr>
<td></td>
<td>Arithmetic shift right by wide elements (unpredicated)</td>
<td>ASR (wide elements, unpredicated) on page C8-3026</td>
</tr>
<tr>
<td>ASRD</td>
<td>Arithmetic shift right for divide by immediate</td>
<td>ASRD on page C8-3028</td>
</tr>
<tr>
<td>ASRR</td>
<td>Reversed arithmetic shift right by vector</td>
<td>ASRR on page C8-3030</td>
</tr>
<tr>
<td>CLS</td>
<td>Count leading sign bits</td>
<td>CLS on page C8-3107</td>
</tr>
<tr>
<td>CLZ</td>
<td>Count leading zero bits</td>
<td>CLZ on page C8-3109</td>
</tr>
<tr>
<td>ONT</td>
<td>Count nonzero bits</td>
<td>ONT on page C8-3142</td>
</tr>
<tr>
<td>LSL</td>
<td>Logical shift left by immediate (predicated)</td>
<td>LSL (immediate, predicated) on page C8-3684</td>
</tr>
<tr>
<td></td>
<td>Logical shift left by immediate (unpredicated)</td>
<td>LSL (immediate, unpredicated) on page C8-3686</td>
</tr>
<tr>
<td></td>
<td>Logical shift left by wide elements (predicated)</td>
<td>LSL (wide elements, predicated) on page C8-3690</td>
</tr>
<tr>
<td></td>
<td>Logical shift left by wide elements (unpredicated)</td>
<td>LSL (wide elements, unpredicated) on page C8-3692</td>
</tr>
<tr>
<td></td>
<td>Logical shift left by vector</td>
<td>LSL (vectors) on page C8-3688</td>
</tr>
<tr>
<td>LSLR</td>
<td>Reversed logical shift left by vector</td>
<td>LSLR on page C8-3694</td>
</tr>
<tr>
<td>LSR</td>
<td>Logical shift right by immediate (predicated)</td>
<td>LSR (immediate, predicated) on page C8-3696</td>
</tr>
<tr>
<td></td>
<td>Logical shift right by immediate (unpredicated)</td>
<td>LSR (immediate, unpredicated) on page C8-3698</td>
</tr>
<tr>
<td></td>
<td>Logical shift right by wide elements (predicated)</td>
<td>LSR (wide elements, predicated) on page C8-3702</td>
</tr>
<tr>
<td></td>
<td>Logical shift right by wide elements (unpredicated)</td>
<td>LSR (wide elements, unpredicated) on page C8-3704</td>
</tr>
<tr>
<td></td>
<td>Logical shift right by vector</td>
<td>LSR (vectors) on page C8-3700</td>
</tr>
<tr>
<td>LSRR</td>
<td>Reversed logical shift right by vector</td>
<td>LSRR on page C8-3706</td>
</tr>
<tr>
<td>RBIT</td>
<td>Reverse bits</td>
<td>RBIT on page C8-3856</td>
</tr>
</tbody>
</table>
Floating-point operations
The following instructions operate on floating-point data within a vector.

SVE Floating-point arithmetic
The Floating-point arithmetic instructions perform arithmetic operations on vectors containing floating-point element values.

Table C3-124 Floating-point arithmetic instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FABD</td>
<td>Floating-point absolute difference</td>
<td>FABD on page C8-3198</td>
</tr>
<tr>
<td>FABS</td>
<td>Floating-point absolute value</td>
<td>FABS on page C8-3200</td>
</tr>
<tr>
<td>FADD</td>
<td>Floating-point add (immediate)</td>
<td>FADD (immediate) on page C8-3206</td>
</tr>
<tr>
<td></td>
<td>Floating-point add (predicated)</td>
<td>FADD (vectors, predicated) on page C8-3208</td>
</tr>
<tr>
<td></td>
<td>Floating-point add (unpredicated)</td>
<td>FADD (vectors, unpredicated) on page C8-3210</td>
</tr>
<tr>
<td>FDIV</td>
<td>Floating-point divide</td>
<td>FDIV on page C8-3253</td>
</tr>
<tr>
<td>FDIVR</td>
<td>Floating-point reversed divide</td>
<td>FDIVR on page C8-3255</td>
</tr>
<tr>
<td>FMAX</td>
<td>Floating-point maximum with immediate</td>
<td>FMAX (immediate) on page C8-3264</td>
</tr>
<tr>
<td></td>
<td>Floating-point maximum vectors</td>
<td>FMAX (vectors) on page C8-3266</td>
</tr>
<tr>
<td>FMAXNM</td>
<td>Floating-point maximum number with immediate</td>
<td>FMAXNM (immediate) on page C8-3268</td>
</tr>
<tr>
<td></td>
<td>Floating-point maximum number vectors</td>
<td>FMAXNM (vectors) on page C8-3270</td>
</tr>
<tr>
<td>FMIN</td>
<td>Floating-point minimum with immediate</td>
<td>FMIN (immediate) on page C8-3278</td>
</tr>
<tr>
<td></td>
<td>Floating-point minimum vectors</td>
<td>FMIN (vectors) on page C8-3280</td>
</tr>
<tr>
<td>FMINNM</td>
<td>Floating-point minimum number with immediate</td>
<td>FMINNM (immediate) on page C8-3282</td>
</tr>
<tr>
<td></td>
<td>Floating-point minimum number vectors</td>
<td>FMINNM (vectors) on page C8-3284</td>
</tr>
<tr>
<td>FMUL</td>
<td>Floating-point multiply by immediate</td>
<td>FMUL (immediate) on page C8-3327</td>
</tr>
<tr>
<td></td>
<td>Floating-point multiply vectors (predicated)</td>
<td>FMUL (vectors, predicated) on page C8-3331</td>
</tr>
<tr>
<td></td>
<td>Floating-point multiply vectors (unpredicated)</td>
<td>FMUL (vectors, unpredicated) on page C8-3333</td>
</tr>
<tr>
<td>FMULX</td>
<td>Floating-point multiply-extended</td>
<td>FMULX on page C8-3334</td>
</tr>
<tr>
<td>FNEG</td>
<td>Floating-point negate</td>
<td>FNEG on page C8-3336</td>
</tr>
<tr>
<td>FRECPE</td>
<td>Floating-point reciprocal estimate</td>
<td>FRECPE on page C8-3346</td>
</tr>
<tr>
<td>FRECPS</td>
<td>Floating-point reciprocal step</td>
<td>FRECPS on page C8-3347</td>
</tr>
<tr>
<td>FRECPX</td>
<td>Floating-point reciprocal exponent</td>
<td>FRECPX on page C8-3348</td>
</tr>
<tr>
<td>FRSQRT</td>
<td>Floating-point reciprocal square root estimate</td>
<td>FRSQRT on page C8-3354</td>
</tr>
<tr>
<td>FRSQRTS</td>
<td>Floating-point reciprocal square root step</td>
<td>FRSQRTS on page C8-3355</td>
</tr>
<tr>
<td>FSQRT</td>
<td>Floating-point square root</td>
<td>FSQRT on page C8-3358</td>
</tr>
</tbody>
</table>
SVE Floating-point multiply accumulate

The Floating-point multiply accumulate instructions perform floating-point fused multiply-add or multiply-subtract operations and their negated forms. There are two groups of these instructions, as follows:

- Instructions where the result of the operation is written to the addend register.
  - Supported instructions are: FMLA, FMLS, FNMLA, FNMLS.
- Instructions where the result of the operation is written to the multiplicand register.
  - Supported instructions are: FMAD, FMSB, FN MAD, FNMSB.

### Table C3-125 Floating-point multiply accumulate instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
</table>
| FMLA     | Floating-point fused multiply-add vectors, writing to the addend | FMLA (indexed) on page C8-3292
|          |             | FMLA (vectors) on page C8-3295 |
| FMLS     | Floating-point fused multiply-subtract vectors, writing to the addend | FMLS (indexed) on page C8-3305 |
|          |             | FMLS (vectors) on page C8-3308 |
| FNMLA    | Floating-point negated fused multiply-add vectors, writing to the addend | FNMLA on page C8-3340 |
| FNMLS    | Floating-point negated fused multiply-subtract vectors, writing to the addend | FNMLS on page C8-3342 |
| FMAD     | Floating-point fused multiply-add vectors, writing to the multiplicand | FMAD on page C8-3262 |
| FMSB     | Floating-point fused multiply-subtract vectors, writing to the multiplicand | FMSL LB (indexed) on page C8-3310 |
|          |             | FMSL LB (vectors) on page C8-3312 |
| FN MAD   | Floating-point negated fused multiply-add vectors, writing to the multiplicand | FN MAD on page C8-3338 |
| FNMSB    | Floating-point negated fused multiply-subtract vectors, writing to the multiplicand | FNMSB on page C8-3344 |

SVE Floating-point complex arithmetic

The Floating-point complex arithmetic instructions perform arithmetic on vectors containing floating-point complex numbers as interleaved pairs of elements, where the even-numbered elements contain the real components and the odd-numbered elements contain the imaginary components.

The FCADD instructions rotate the complex numbers in the second source vector by 90 degrees or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, before adding active pairs of elements to the corresponding elements of the first source vector in a destructive manner.
The FCMLA instructions perform a transformation of the operands to allow the creation of multiply-add or multiply-subtract operations on complex numbers by combining two of the instructions. The transformations performed are as follows:

- The complex numbers in the second source vector, considered in polar form, are rotated by 0 degrees or 180 degrees before multiplying by the duplicated real components of the first source vector.
- The complex numbers in the second source vector, considered in polar form, are rotated by 90 degrees or 270 degrees before multiplying by the duplicated imaginary components of the first source vector.

The resulting products are then added to the corresponding components of the destination and addend vector, without intermediate rounding.

Two FCMLA instructions can be used as follows:

```
FCMLA Zda.S, Pg/M, Zn.S, Zm.S, #A
...
FCMLA Zda.S, Pg/M, Zn.S, Zm.S, #B
```

For example, some meaningful combinations of A and B are:

- A=0, B=90. In this case, the two vectors of complex numbers in Zn and Zm are multiplied and the products are added to the complex numbers in Zda.
- A=0, B=270. In this case, the conjugates of the complex numbers in Zn are multiplied by the complex numbers in Zm and the products are added to the complex numbers in Zda.
- A=180, B=270. In this case, the two vectors of complex numbers in Zn and Zm are multiplied and the products are subtracted from the complex numbers in Zda.
- A=180, B=90. In this case, the conjugates of the complex numbers in Zn are multiplied by the complex numbers in Zm and the products are subtracted from the complex numbers in Zda.

Note

The lack of intermediate rounding can give unexpected results in certain cases relative to a traditional sequence of independent multiply, add, and subtract instructions.

In addition, when using these instructions, the behavior of calculations such as \((\infty+i)\) multiplied by \((0+i)\) is \((\text{NaN}+\text{NaN}i)\), rather than the result expected by ISO C, which is complex. The expectation is that these instructions are only used in situations where the effect of differences in the rounding and handling of infinities are not material to the calculation.

### Table C3-126 Floating-point complex arithmetic instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCADD</td>
<td>Floating-point complex add with rotate</td>
<td>FCADD on page C8-3216</td>
</tr>
<tr>
<td>FCMLA</td>
<td>Floating-point complex multiply-add with rotate</td>
<td>FCMLA (vectors) on page C8-3229</td>
</tr>
</tbody>
</table>
SVE Floating-point rounding and conversion

The Floating-point rounding and conversion instructions change floating-point size and precision, round floating-point to integral floating-point with explicit rounding mode, and convert floating-point to or from integer format.

Table C3-127 Floating-point rounding and conversion instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFCVT</td>
<td>Floating-point down convert to BFloat16 format</td>
<td>BFCVT on page C8-3038</td>
</tr>
<tr>
<td>BFCVTNT</td>
<td>Floating-point down convert and narrow to BFloat16 format (top, predicated)</td>
<td>BFCVTNT on page C8-3040</td>
</tr>
<tr>
<td>FCVT</td>
<td>Floating-point convert precision</td>
<td>FCVT on page C8-3235</td>
</tr>
<tr>
<td>FCVTZS</td>
<td>Floating-point convert to signed integer, rounding toward zero</td>
<td>FCVTZS on page C8-3245</td>
</tr>
<tr>
<td>FCVTZU</td>
<td>Floating-point convert to unsigned integer, rounding toward zero</td>
<td>FCVTZU on page C8-3249</td>
</tr>
<tr>
<td>FRINTA</td>
<td>Floating-point round to integral value, to nearest with ties away from zero</td>
<td>FRINT&lt;r&gt; Nearest with ties away from zero on page C8-3351</td>
</tr>
<tr>
<td>FRINTI</td>
<td>Floating-point round to integral value, using the current rounding mode</td>
<td>FRINT&lt;r&gt; Current mode on page C8-3350</td>
</tr>
<tr>
<td>FRINTM</td>
<td>Floating-point round to integral value, toward minus infinity</td>
<td>FRINT&lt;r&gt; Toward minus infinity on page C8-3352</td>
</tr>
<tr>
<td>FRINTN</td>
<td>Floating-point round to integral value, to nearest with ties to even</td>
<td>FRINT&lt;r&gt; Nearest with ties to even on page C8-3351</td>
</tr>
<tr>
<td>FRINTP</td>
<td>Floating-point round to integral value, toward plus infinity</td>
<td>FRINT&lt;r&gt; Toward plus infinity on page C8-3352</td>
</tr>
<tr>
<td>FRINTX</td>
<td>Floating-point round to integral value exact, using the current rounding mode</td>
<td>FRINT&lt;r&gt; Current mode signalling inexact on page C8-3350</td>
</tr>
<tr>
<td>FRINTZ</td>
<td>Floating-point round to integral value, toward zero</td>
<td>FRINT&lt;r&gt; Toward zero on page C8-3352</td>
</tr>
<tr>
<td>SCVTF</td>
<td>Signed integer convert to floating-point</td>
<td>SCVTF on page C8-3909</td>
</tr>
<tr>
<td>UCVTF</td>
<td>Unsigned integer convert to floating-point</td>
<td>UCVTF on page C8-4333</td>
</tr>
</tbody>
</table>

SVE Floating-point comparisons

The Floating-point comparison instructions compare active floating-point element values in the first source vector with corresponding elements in the second vector or with the immediate value +0.0. The Boolean result of each comparison is placed in the corresponding element of the destination predicate. Inactive elements in the destination predicate register are set to FALSE. Floating-point vector comparisons do not set the condition flags.

Table C3-128 Floating-point comparison instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FACGE</td>
<td>Floating-point absolute compare greater than or equal</td>
<td>FAC&lt;cc&gt; - Greater than or equal on page C8-3202</td>
</tr>
<tr>
<td>FACGT</td>
<td>Floating-point absolute compare greater than</td>
<td>FAC&lt;cc&gt; - Greater than on page C8-3202</td>
</tr>
<tr>
<td>FACLE</td>
<td>Floating-point absolute compare less than or equal</td>
<td>FACLE on page C8-3204</td>
</tr>
</tbody>
</table>
SVE Floating-point transcendental acceleration

The Floating-point transcendental instructions accelerate calculations of sine, cosine, and exponential functions for vectors containing floating-point element values.

The trigonometric instructions accelerate the calculation of a polynomial series approximation for the sine and cosine functions. The exponential instruction accelerates the polynomial series calculation of the exponential function.

Table C3-129 Floating-point transcendental instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTMAD</td>
<td>Floating-point trigonometric multiply-add coefficient</td>
<td><em>FTMAD on page C8-3369</em></td>
</tr>
<tr>
<td>FTSMUL</td>
<td>Floating-point trigonometric starting value</td>
<td><em>FTSMUL on page C8-3373</em></td>
</tr>
<tr>
<td>FTSSEL</td>
<td>Floating-point trigonometric select coefficient</td>
<td><em>FTSSEL on page C8-3375</em></td>
</tr>
<tr>
<td>FEXPA</td>
<td>Floating-point exponential accelerator</td>
<td><em>FEXPA on page C8-3258</em></td>
</tr>
</tbody>
</table>
SVE Floating-point indexed multiples

The Floating-point indexed multiples instructions multiply all floating-point elements within each 128-bit segment of the first source vector by the single numbered element within the corresponding segment of the second source vector.

For the FMLA and FMLS instructions, the products are destructively added or subtracted from the corresponding elements of the addend and destination vector, without intermediate rounding.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMLA</td>
<td>Floating-point fused multiply-add by indexed elements</td>
<td>FMLA (indexed) on page C8-3292</td>
</tr>
<tr>
<td>FMLS</td>
<td>Floating-point fused multiply-subtract by indexed elements</td>
<td>FMLS (indexed) on page C8-3305</td>
</tr>
<tr>
<td>FMUL</td>
<td>Floating-point multiply by indexed elements</td>
<td>FMUL (indexed) on page C8-3329</td>
</tr>
</tbody>
</table>

SVE Floating-point matrix multiply operations

The Floating-point matrix multiply instructions facilitate matrix multiplication and include floating-point matrix multiply-accumulate instructions, and companion instructions that support data rearrangements in vector registers as required by the double-precision matrix multiply-accumulate instructions.

The matrix multiply-accumulate instructions delimit source and destination vectors into segments. Within each segment:

- The first source vector 2x2 matrix is organized in row-by-row order.
- The second source vector 2x2 matrix is organized in a column-by-column order.
- The destination vector 2x2 matrix is organized in row-by-row order.

One matrix multiplication is performed per vector segment and accumulated into the destination vector segment. For the double-precision matrix multiply-accumulate instructions, the vector segment length and minimum vector length is 256 bits. Double-precision matrix multiply-accumulate instructions are not supported when the vector length is 128 bits. For the single-precision matrix multiply-accumulate instruction, the vector segment length is 128 bits.

The floating-point matrix multiply-accumulate instructions strictly define the order of accumulations, and the multiplications and additions are not fused, so intermediate rounding is performed after every multiplication and every addition.

The following table shows the floating-point matrix multiplication instructions and companion instructions that are supported if ID_AA64ZFR0_EL1.F64MM is 1:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMMLA</td>
<td>Floating-point matrix multiply-accumulate into 2x2 matrix (double-precision)</td>
<td>FMMLA on page C8-3318</td>
</tr>
<tr>
<td>LD1ROB</td>
<td>Contiguous load and replicate thirty-two bytes, scalar plus scalar</td>
<td>LD1ROB (scalar plus scalar) on page C8-3451</td>
</tr>
<tr>
<td></td>
<td>Contiguous load and replicate thirty-two bytes, scalar plus immediate</td>
<td>LD1ROB (scalar plus immediate) on page C8-3449</td>
</tr>
<tr>
<td>LD1ROD</td>
<td>Contiguous load and replicate four doublewords, scalar plus scalar</td>
<td>LD1ROD (scalar plus scalar) on page C8-3455</td>
</tr>
<tr>
<td></td>
<td>Contiguous load and replicate four doublewords, scalar plus immediate</td>
<td>LD1ROD (scalar plus immediate) on page C8-3453</td>
</tr>
</tbody>
</table>
The following table shows the floating-point matrix multiplication instructions and companion instructions that are supported if ID_AA64ZFR0_EL1.F32MM is 1:

### Table C3-131 Floating-point matrix multiply instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD1ROH</td>
<td>Contiguous load and replicate sixteen halfwords, scalar plus scalar</td>
<td><strong>LD1ROH (scalar plus scalar)</strong> on page C8-3459</td>
</tr>
<tr>
<td></td>
<td>Contiguous load and replicate sixteen halfwords, scalar plus immediate</td>
<td><strong>LD1ROH (scalar plus immediate)</strong> on page C8-3457</td>
</tr>
<tr>
<td>LD1ROW</td>
<td>Contiguous load and replicate eight words, scalar plus scalar</td>
<td><strong>LD1ROW (scalar plus scalar)</strong> on page C8-3463</td>
</tr>
<tr>
<td></td>
<td>Contiguous load and replicate eight words, scalar plus immediate</td>
<td><strong>LD1ROW (scalar plus immediate)</strong> on page C8-3461</td>
</tr>
<tr>
<td>TRN1, TRN2</td>
<td>Interleave even or odd 128-bit elements from two vectors</td>
<td><strong>TRN1, TRN2 (vectors)</strong> on page C8-4304</td>
</tr>
<tr>
<td>UZP1, UZP2</td>
<td>Concatenate even or odd 128-bit elements from two vectors</td>
<td><strong>UZP1, UZP2 (vectors)</strong> on page C8-4510</td>
</tr>
<tr>
<td>ZIP1, ZIP2</td>
<td>Interleave 128-bit elements from two half vectors</td>
<td><strong>ZIP1, ZIP2 (vectors)</strong> on page C8-4538</td>
</tr>
</tbody>
</table>

The following table shows the floating-point matrix multiplication instructions and companion instructions that are supported if ID_AA64ZFR0_EL1.F32MM is 1:

### SVE BF16 floating-point multiply-add

The BF16 floating-point multiply-add instructions perform an implicit conversion of the bottom (even-numbered) or top (odd-numbered) BF16 source elements to IEEE 754 single-precision floating-point format before performing a fused multiply-add without intermediate rounding to the overlapping single-precision destination element. These instructions follow the normal floating-point behaviors that apply to single-precision arithmetic, controlled by the Effective value of the FPCR, and captured in the FPSR cumulative exception bits.

### Table C3-132 SVE BF16 floating-point multiply-add instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFMLALB</td>
<td>BF16 floating-point widening multiply accumulate long bottom (vector and indexed forms)</td>
<td><strong>BFMLALB (vectors)</strong> on page C8-3047</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>BFMLALB (indexed)</strong> on page C8-3045</td>
</tr>
<tr>
<td>BFMLALT</td>
<td>BF16 floating-point widening multiply accumulate long top (vector and indexed forms)</td>
<td><strong>BFMLALT (vectors)</strong> on page C8-3051</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>BFMLALT (indexed)</strong> on page C8-3049</td>
</tr>
</tbody>
</table>

### SVE BF16 floating-point dot product

The BF16 floating-point dot product instruction, BFDOT, performs an implicit conversion of vectors of BF16 input values to IEEE 754 single-precision floating-point format.

The BF16 dot product instructions delimit their source vectors into pairs of BF16 elements.
If FEAT_EBF16 is not implemented or FPCR.EBF is 0, irrespective of the other control bits in the FPCR, a $\text{BFDO}T$ instruction:

- Performs an unfused, widening sum-of-products of each pair of adjacent BF\text{loat16} elements in the first source vector with the corresponding or indexed pair in the second source vector. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BF\text{loat16} elements in the first source vector.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Does not modify the FPSR cumulative exception bits.
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits are all zero.
- Flushes denormalized inputs and results to zero, as if FPCR.FIZ and FPCR.FIZ are 0.
- Generates only the default NaN, as if FPCR.DN is 1.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, a $\text{BFDO}T$ instruction:

- Performs a fused, widening sum-of-products of each pair of adjacent BF\text{loat16} elements in the first source vector with the corresponding or indexed pair in the second source vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BF\text{loat16} elements in the first source vector.
- Generates only the default NaN, as if FPCR.DN is 1.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as controlled by the Effective value of the FPCR in the current execution mode, and captured in the FPSR cumulative exception bits.

### Table C3-133 SVE BF\text{loat16} floating-point dot product instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{BFDO}T$</td>
<td>BF\text{loat16} floating-point dot product (vector and indexed forms)</td>
<td>$\text{BFDO}T$ (vectors) on page C8-3043</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{BFDO}T$ (indexed) on page C8-3041</td>
</tr>
</tbody>
</table>

### SVE BF\text{loat16} floating-point matrix multiply

The BF\text{loat16} floating-point matrix multiply instruction, BF\text{MMLA}, delimits source and destination vectors into 128-bit segments. Within each segment:

- The first source vector 2x4 BF\text{16} matrix is organized in row-by-row order.
- The second source vector 4x2 BF\text{16} matrix is organized in a column-by-column order.
- The destination vector 2x2 single-precision matrix is organized in row-by-row order.

One matrix multiplication is performed per source vector segment and accumulated into the corresponding destination vector segment. This corresponds to accumulating two 2-way BF\text{loat16} widening dot products into each single-precision destination element, following the numeric behaviors described for $\text{BFDO}T$ instruction in SVE BF\text{loat16} floating-point dot product on page C3-353.

### Table C3-134 SVE BF\text{loat16} floating-point matrix multiply instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF\text{MMLA}</td>
<td>BF\text{loat16} floating-point matrix multiply-accumulate</td>
<td>BF\text{MMLA} on page C8-3053</td>
</tr>
</tbody>
</table>
SVE BFloat16 floating-point convert

The BFloat16 floating-point convert instructions perform accurately rounded down-conversion of IEEE 754 single-precision source vector elements to BFloat16 format.

The BFCVT instruction places its BFloat16 results in the bottom or even-numbered 16-bit elements of the destination vector, and sets the top or odd-numbered elements to zero.

The BFCVTNT instruction places its BFloat16 results in the top or odd-numbered 16-bit elements of the destination vector, leaving the bottom or even-numbered elements unchanged.

These instructions follow the normal floating-point behaviors that apply to single-precision arithmetic, controlled by the Effective value of the FPCR, and captured in the FPSR cumulative exception bits.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFCVT</td>
<td>Floating-point down convert to BFloat16 format (predicated)</td>
<td>BFCVT on page C8-3038</td>
</tr>
<tr>
<td>BFCVTNT</td>
<td>Floating-point down convert and narrow to BFloat16 (top, predicated)</td>
<td>BFCVTNT on page C8-3040</td>
</tr>
</tbody>
</table>

Predicate operations

The Predicate instructions perform operations that manipulate the predicate registers.

Some of these instructions are insensitive to the predicate element size and specify an explicit byte element size qualifier, .B, but an assembler must accept any qualifier, or none.

SVE Predicate initialization

The Predicate initialization instructions initialize predicate elements. Predicate elements can be initialized to be FALSE, or to be TRUE when their element number is less than:

- A fixed number of elements from the following range: VL1-VL8, VL16, VL32, VL64, VL128 or VL256.
- The largest power of two elements, POW2.
- The largest multiple of three or four elements, MUL3 or MUL4.
- The number of accessible elements, ALL, which is implicitly a multiple of two.

Unspecified or out of range constraint encodings generate a predicate with values that are all FALSE and do not cause an Undefined Instruction exception.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFALSE</td>
<td>Set all predicate elements to FALSE</td>
<td>PFALSE on page C8-3795</td>
</tr>
<tr>
<td>PTRUE</td>
<td>Initialize predicate elements from named constraint</td>
<td>PTRUE on page C8-3844</td>
</tr>
<tr>
<td>PTRUES</td>
<td>Initialize predicate elements from named constraint, setting the condition flags</td>
<td>PTRUES on page C8-3846</td>
</tr>
</tbody>
</table>
### SVE Predicate move operations

The Predicate move instructions operate on all bits of the predicate registers, implying a fixed, 1-bit predicate element size. The flag-setting variants set the N, Z, and C condition flags based on the predicate result, and set the V flag to zero. Because these instructions operate with a fixed, 1-bit element size, the *Governing predicate* for the flag-setting variants should be in the canonical form for a predicate element size in order to generate a meaningful set of condition flags for that element size.

#### Table C3-137 Predicate move instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEL</td>
<td>Select predicate elements from two predicates</td>
<td>SEL (predicates) on page C8-3921</td>
</tr>
<tr>
<td>MOV</td>
<td>Move predicate elements (predicated, merging)</td>
<td>MOV (predicate, predicated, merging) on page C8-3731</td>
</tr>
<tr>
<td></td>
<td>Move predicate elements (predicated, zeroing)</td>
<td>MOV (predicate, predicated, zeroing) on page C8-3732</td>
</tr>
<tr>
<td>MOV5</td>
<td>Move predicate elements (unpredicated)</td>
<td>MOV on page C8-3724</td>
</tr>
<tr>
<td></td>
<td>Move predicate elements, setting the condition flags (predicated)</td>
<td>MOV5 (predicated) on page C8-3746</td>
</tr>
<tr>
<td></td>
<td>Move predicate elements, setting the condition flags (unpredicated)</td>
<td>MOV5 (unpredicated) on page C8-3747</td>
</tr>
</tbody>
</table>

### SVE Predicate logical operations

The Predicate logical operation instructions perform bitwise logical operations on predicate registers that operate on all bits of the register, implying a fixed, 1-bit predicate element size. The flag-setting variants set the N, Z, and C condition flags based on the predicate result, and set the V flag to zero. Inactive elements in the destination Predicate register are set to zero, except for PTEST which does not specify a destination register. Because these instructions operate with a fixed, 1-bit element size, the *Governing predicate* for the flag-setting variants should be in the canonical form for a predicate element size in order to generate a meaningful set of condition flags for that element size.

#### Table C3-138 Predicate logical operation instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Bitwise AND predicates</td>
<td>AND (predicates) on page C8-3009</td>
</tr>
<tr>
<td>ANDS</td>
<td>Bitwise AND predicates, setting the condition flags</td>
<td>ANDS on page C8-3014</td>
</tr>
<tr>
<td>BIC</td>
<td>Bitwise clear predicates</td>
<td>BIC (predicates) on page C8-3059</td>
</tr>
<tr>
<td>BICS</td>
<td>Bitwise clear predicates, setting the condition flags</td>
<td>BICS on page C8-3064</td>
</tr>
<tr>
<td>EOR</td>
<td>Bitwise exclusive OR predicates</td>
<td>EOR (predicates) on page C8-3181</td>
</tr>
<tr>
<td>EORS</td>
<td>Bitwise exclusive OR predicates, setting the condition flags</td>
<td>EORS on page C8-3190</td>
</tr>
<tr>
<td>NAND</td>
<td>Bitwise NAND predicates</td>
<td>NAND on page C8-3759</td>
</tr>
<tr>
<td>NANDS</td>
<td>Bitwise NAND predicates, setting the condition flags</td>
<td>NANDS on page C8-3761</td>
</tr>
<tr>
<td>NOR</td>
<td>Bitwise NOR predicates</td>
<td>NOR on page C8-3769</td>
</tr>
<tr>
<td>NORS</td>
<td>Bitwise NOR predicates, setting the condition flags</td>
<td>NORS on page C8-3771</td>
</tr>
<tr>
<td>NOT</td>
<td>Bitwise invert predicate</td>
<td>NOT (predicate) on page C8-3773</td>
</tr>
<tr>
<td>NOTS</td>
<td>Bitwise invert predicate, setting the condition flags</td>
<td>NOTS on page C8-3776</td>
</tr>
</tbody>
</table>
**FFR predicate handling**

The FFR predicate handling instructions work with SVE First-fault and Non-fault loads using the FFR to determine which elements have been successfully loaded and which remain to be loaded on a subsequent iteration.

The RDFFRS instruction sets the N, Z, and C condition flags based on the predicate result, and sets the V flag to zero. Because these instructions operate with a fixed, 1-bit element size, the Governing predicate for the RDFFRS instruction should be in the canonical form for a predicate element size in order to generate a meaningful set of condition flags for that element size.

**Predicate counts**

The Predicate count instructions count either the number of Active predicate elements that are set to TRUE, or the number of elements implied by a named predicate constraint. The count can be placed in a general-purpose register, or used to increment or decrement a vector or general-purpose register.

Signed or unsigned saturating variants handle cases where, for example, an increment might cause a vectorized scalar loop index to overflow and therefore never satisfy a loop termination condition that compares it with a limit that is close to the maximum integer value.

The named predicate constraint limits the number of elements to:

- A fixed number of elements from the following range: VL1-VL8, VL16, VL32, VL64, VL128 or VL256.
- The largest power of two elements, POW2.
- The largest multiple of three or four elements, MUL3 or MUL4.
- The number of accessible elements, ALL, implicitly a multiple of two.

### Table C3-138 Predicate logical operation instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORN</td>
<td>Bitwise OR inverted predicate</td>
<td>ORN (predicates) on page C8-3779</td>
</tr>
<tr>
<td>ORNS</td>
<td>Bitwise OR inverted predicate, setting the condition flags</td>
<td>ORNS on page C8-3781</td>
</tr>
<tr>
<td>ORR</td>
<td>Bitwise OR predicates</td>
<td>ORR (predicates) on page C8-3785</td>
</tr>
<tr>
<td>ORRS</td>
<td>Bitwise OR predicates, setting the condition flags</td>
<td>ORRS on page C8-3791</td>
</tr>
<tr>
<td>PTEST</td>
<td>Test predicate value, setting the condition flags</td>
<td>PTEST on page C8-3843</td>
</tr>
</tbody>
</table>

### Table C3-139 FFR predicate handling instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDFFR</td>
<td>Return predicate of successfully loaded elements (unpredicated)</td>
<td>RDFFR (unpredicated) on page C8-3859</td>
</tr>
<tr>
<td></td>
<td>Return predicate of successfully loaded elements (predicated)</td>
<td>RDFFR (predicated) on page C8-3858</td>
</tr>
<tr>
<td>RDFFRS</td>
<td>Return predicate of successfully loaded elements, setting the condition flags (predicated)</td>
<td>RDFFRS on page C8-3860</td>
</tr>
<tr>
<td>SETFFR</td>
<td>Initialize the First-fault register to all TRUE</td>
<td>SETFFR on page C8-3925</td>
</tr>
<tr>
<td>WRFFR</td>
<td>Write a predicate register to the First-fault register</td>
<td>WRFFR on page C8-4533</td>
</tr>
</tbody>
</table>
Unspecified or out of range predicate constraint encodings generate a zero element count and do not cause an Undefined Instruction exception.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNTB</td>
<td>Set scalar to multiple of 8-bit predicate constraint element count</td>
<td>CNTB, CNTD, CNTH, CNTW on page C8-3144</td>
</tr>
<tr>
<td>CNTH</td>
<td>Set scalar to multiple of 16-bit predicate constraint element count</td>
<td>CNTB, CNTD, CNTH, CNTW on page C8-3144</td>
</tr>
<tr>
<td>CNTW</td>
<td>Set scalar to multiple of 32-bit predicate constraint element count</td>
<td>CNTB, CNTD, CNTH, CNTW on page C8-3144</td>
</tr>
<tr>
<td>CNTD</td>
<td>Set scalar to multiple of 64-bit predicate constraint element count</td>
<td>CNTB, CNTD, CNTH, CNTW on page C8-3144</td>
</tr>
<tr>
<td>CNTP</td>
<td>Set scalar to the number of Active predicate elements that are TRUE</td>
<td>CNTP on page C8-3147</td>
</tr>
<tr>
<td>DECB</td>
<td>Decrement scalar by multiple of 8-bit predicate constraint element count</td>
<td>DECB, DECD, DECH, DECW (scalar) on page C8-3160</td>
</tr>
<tr>
<td>DECH</td>
<td>Decrement scalar by multiple of 16-bit predicate constraint element count</td>
<td>DECB, DECD, DECH, DECW (scalar) on page C8-3160</td>
</tr>
<tr>
<td></td>
<td>Decrement vector by multiple of 16-bit predicate constraint element count</td>
<td>DECD, DECH, DECW (vector) on page C8-3163</td>
</tr>
<tr>
<td>DECW</td>
<td>Decrement scalar by multiple of 32-bit predicate constraint element count</td>
<td>DECB, DECD, DECH, DECW (scalar) on page C8-3160</td>
</tr>
<tr>
<td></td>
<td>Decrement vector by multiple of 32-bit predicate constraint element count</td>
<td>DECD, DECH, DECW (vector) on page C8-3163</td>
</tr>
<tr>
<td>DECD</td>
<td>Decrement scalar by multiple of 64-bit predicate constraint element count</td>
<td>DECB, DECD, DECH, DECW (scalar) on page C8-3160</td>
</tr>
<tr>
<td></td>
<td>Decrement vector by multiple of 64-bit predicate constraint element count</td>
<td>DECD, DECH, DECW (vector) on page C8-3163</td>
</tr>
<tr>
<td>DECP</td>
<td>Decrement scalar by the number of predicate elements that are TRUE</td>
<td>DECP (scalar) on page C8-3166</td>
</tr>
<tr>
<td></td>
<td>Decrement vector by the number of Active predicate elements that are TRUE</td>
<td>DECP (vector) on page C8-3167</td>
</tr>
<tr>
<td>INCB</td>
<td>Increment scalar by multiple of 8-bit predicate constraint element count</td>
<td>INCB, INC, INCH, INCW (scalar) on page C8-3380</td>
</tr>
<tr>
<td>INCH</td>
<td>Increment scalar by multiple of 16-bit predicate constraint element count</td>
<td>INCB, INC, INCH, INCW (scalar) on page C8-3380</td>
</tr>
<tr>
<td></td>
<td>Increment vector by multiple of 16-bit predicate constraint element count</td>
<td>INCD, INCH, INCW (vector) on page C8-3383</td>
</tr>
<tr>
<td>INCW</td>
<td>Increment scalar by multiple of 32-bit predicate constraint element count</td>
<td>INCB, INC, INCH, INCW (scalar) on page C8-3380</td>
</tr>
<tr>
<td></td>
<td>Increment vector by multiple of 32-bit predicate constraint element count</td>
<td>INCD, INCH, INCW (vector) on page C8-3383</td>
</tr>
<tr>
<td>INCD</td>
<td>Increment scalar by multiple of 64-bit predicate constraint element count</td>
<td>INCB, INC, INCH, INCW (scalar) on page C8-3380</td>
</tr>
<tr>
<td></td>
<td>Increment vector by multiple of 64-bit predicate constraint element count</td>
<td>INCD, INCH, INCW (vector) on page C8-3383</td>
</tr>
</tbody>
</table>
### Table C3-140 Predicate count instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCP</td>
<td>Increment scalar by the number of predicate elements that are TRUE</td>
<td><strong>INCP (scalar)</strong> on page C8-3386</td>
</tr>
<tr>
<td></td>
<td>Increment vector by the number of predicate elements that are TRUE</td>
<td><strong>INCP (vector)</strong> on page C8-3387</td>
</tr>
<tr>
<td>SQDECB</td>
<td>Signed saturating decrement scalar by multiple of 8-bit predicate constraint element count</td>
<td><strong>SQDECB</strong> on page C8-3999</td>
</tr>
<tr>
<td>SQDECH</td>
<td>Signed saturating decrement scalar by multiple of 16-bit predicate constraint element count</td>
<td><strong>SQDECH (scalar)</strong> on page C8-4005</td>
</tr>
<tr>
<td></td>
<td>Signed saturating decrement vector by multiple of 16-bit predicate constraint element count</td>
<td><strong>SQDECH (vector)</strong> on page C8-4007</td>
</tr>
<tr>
<td>SQDECW</td>
<td>Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count</td>
<td><strong>SQDECW (scalar)</strong> on page C8-4013</td>
</tr>
<tr>
<td></td>
<td>Signed saturating decrement vector by multiple of 32-bit predicate constraint element count</td>
<td><strong>SQDECW (vector)</strong> on page C8-4015</td>
</tr>
<tr>
<td>SQDECD</td>
<td>Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count</td>
<td><strong>SQDECD (scalar)</strong> on page C8-4001</td>
</tr>
<tr>
<td></td>
<td>Signed saturating decrement vector by multiple of 64-bit predicate constraint element count</td>
<td><strong>SQDECD (vector)</strong> on page C8-4003</td>
</tr>
<tr>
<td>SQDECP</td>
<td>Signed saturating decrement scalar the number of predicate elements that are TRUE</td>
<td><strong>SQDECP (scalar)</strong> on page C8-4009</td>
</tr>
<tr>
<td></td>
<td>Signed saturating decrement vector by the number of predicate elements that are TRUE</td>
<td><strong>SQDECP (vector)</strong> on page C8-4011</td>
</tr>
<tr>
<td>SQINCB</td>
<td>Signed saturating increment scalar by multiple of 8-bit predicate constraint element count</td>
<td><strong>SQINCB</strong> on page C8-4050</td>
</tr>
<tr>
<td>SQINCH</td>
<td>Signed saturating increment scalar by multiple of 16-bit predicate constraint element count</td>
<td><strong>SQINCH (scalar)</strong> on page C8-4056</td>
</tr>
<tr>
<td></td>
<td>Signed saturating increment vector by multiple of 16-bit predicate constraint element count</td>
<td><strong>SQINCH (vector)</strong> on page C8-4058</td>
</tr>
<tr>
<td>SQINCW</td>
<td>Signed saturating increment scalar by multiple of 32-bit predicate constraint element count</td>
<td><strong>SQINCW (scalar)</strong> on page C8-4064</td>
</tr>
<tr>
<td></td>
<td>Signed saturating increment vector by multiple of 32-bit predicate constraint element count</td>
<td><strong>SQINCW (vector)</strong> on page C8-4066</td>
</tr>
<tr>
<td>SQINCD</td>
<td>Signed saturating increment scalar by multiple of 64-bit predicate constraint element count</td>
<td><strong>SQINCD (scalar)</strong> on page C8-4052</td>
</tr>
<tr>
<td></td>
<td>Signed saturating increment vector by multiple of 64-bit predicate constraint element count</td>
<td><strong>SQINCD (vector)</strong> on page C8-4054</td>
</tr>
<tr>
<td>SQINCP</td>
<td>Signed saturating increment scalar by the number of predicate elements that are TRUE</td>
<td><strong>SQINCP (scalar)</strong> on page C8-4060</td>
</tr>
<tr>
<td></td>
<td>Signed saturating increment vector by the number of predicate elements that are TRUE</td>
<td><strong>SQINCP (vector)</strong> on page C8-4062</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Instruction</td>
<td>See</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-----</td>
</tr>
<tr>
<td>UQDECB</td>
<td>Unsigned saturating decrement scalar by multiple of 8-bit predicate constraint element count</td>
<td>UQDECB on page C8-4402</td>
</tr>
<tr>
<td>UQDECH</td>
<td>Unsigned saturating decrement scalar by multiple of 16-bit predicate constraint element count</td>
<td>UQDECH (scalar) on page C8-4408</td>
</tr>
<tr>
<td></td>
<td>Unsigned saturating decrement vector by multiple of 16-bit predicate constraint element count</td>
<td>UQDECH (vector) on page C8-4410</td>
</tr>
<tr>
<td>UQDECW</td>
<td>Unsigned saturating decrement scalar by multiple of 32-bit predicate constraint element count</td>
<td>UQDECW (scalar) on page C8-4416</td>
</tr>
<tr>
<td></td>
<td>Unsigned saturating decrement vector by multiple of 32-bit predicate constraint element count</td>
<td>UQDECW (vector) on page C8-4418</td>
</tr>
<tr>
<td>UQDECD</td>
<td>Unsigned saturating decrement scalar by multiple of 64-bit predicate constraint element count</td>
<td>UQDECD (scalar) on page C8-4404</td>
</tr>
<tr>
<td></td>
<td>Unsigned saturating decrement vector by multiple of 64-bit predicate constraint element count</td>
<td>UQDECD (vector) on page C8-4406</td>
</tr>
<tr>
<td>UQDECP</td>
<td>Unsigned saturating decrement scalar by the number of predicate elements that are TRUE</td>
<td>UQDECP (scalar) on page C8-4412</td>
</tr>
<tr>
<td></td>
<td>Unsigned saturating decrement vector by the number of predicate elements that are TRUE</td>
<td>UQDECP (vector) on page C8-4414</td>
</tr>
<tr>
<td>UQINCB</td>
<td>Unsigned saturating increment scalar by multiple of 8-bit predicate constraint element count</td>
<td>UQINCB on page C8-4420</td>
</tr>
<tr>
<td>UQINCH</td>
<td>Unsigned saturating increment scalar by multiple of 16-bit predicate constraint element count</td>
<td>UQINCH (scalar) on page C8-4426</td>
</tr>
<tr>
<td></td>
<td>Unsigned saturating increment vector by multiple of 16-bit predicate constraint element count</td>
<td>UQINCH (vector) on page C8-4428</td>
</tr>
<tr>
<td>UQINCW</td>
<td>Unsigned saturating increment scalar by multiple of 32-bit predicate constraint element count</td>
<td>UQINCW (scalar) on page C8-4434</td>
</tr>
<tr>
<td></td>
<td>Unsigned saturating increment vector by multiple of 32-bit predicate constraint element count</td>
<td>UQINCW (vector) on page C8-4436</td>
</tr>
<tr>
<td>UQINCD</td>
<td>Unsigned saturating increment scalar by multiple of 64-bit predicate constraint element count</td>
<td>UQINCD (scalar) on page C8-4422</td>
</tr>
<tr>
<td></td>
<td>Unsigned saturating increment vector by multiple of 64-bit predicate constraint element count</td>
<td>UQINCD (vector) on page C8-4424</td>
</tr>
<tr>
<td>UQINCP</td>
<td>Unsigned saturating increment scalar by the number of predicate elements that are TRUE</td>
<td>UQINCP (scalar) on page C8-4430</td>
</tr>
<tr>
<td></td>
<td>Unsigned saturating increment vector by the number of predicate elements that are TRUE</td>
<td>UQINCP (vector) on page C8-4432</td>
</tr>
</tbody>
</table>
Loop control

These instructions control counted vector loops and vector loops with data-dependent termination conditions. These instructions create a loop partition predicate with Active elements set to TRUE up to the point where the loop should terminate, and FALSE thereafter. Two loop concepts are supported, simple loops and data-dependent loops.

Simple loops

An up-counting WHILE instruction that increments the value of the first scalar operand and compares the value with a second, fixed scalar operand. The instruction generates a destination predicate with all of the following characteristics:

- The predicate elements starting from the lowest numbered element are true while the comparison is true.
- The predicate elements thereafter, up to the highest numbered element, are false when the comparison becomes false.

All 32 bits or 64 bits of the scalar operands are significant for the purposes of comparison. The full 32-bit or 64-bit value of the first operand is incremented by 1 for each destination predicate element, irrespective of the element size. The first general-purpose register operand is not updated.

If all of the following occur, a comparison can never fail, resulting in an all-true predicate:

- The comparison includes an equality test.
- The second scalar operand is equal to the maximum integer value of the selected size and type of comparison.

The N, Z, C, and V condition flags are unconditionally set to control a subsequent conditional branch.

Data-independent loops

For data-dependent termination conditions, it is necessary to convert the result of a vector comparison into a loop partition predicate. The new partition truncates the current vector partition immediately before or after the first active TRUE comparison. The N, Z, C, and V condition flags are optionally set to control a subsequent conditional branch.

The BRK instructions set active destination predicate elements to TRUE up to and including the first active TRUE element in their source predicate register, setting subsequent elements to FALSE.

The BRKP instructions set active destination predicate elements to TRUE up to but excluding the first active TRUE element in their source predicate register, setting subsequent elements to FALSE.

The BRKPA and BRKPB instructions propagate the result of a previous BRKB or BRKPB instruction, by setting their destination predicate register to all FALSE if the Last active element of their first source predicate register is not TRUE, but otherwise generate the destination predicate from their second source predicate as described for the BRK and BRKB instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHILELE</td>
<td>While incrementing signed scalar less than or equal to scalar</td>
<td>WHILELE on page C8-4521</td>
</tr>
<tr>
<td>WHILELO</td>
<td>While incrementing unsigned scalar lower than scalar</td>
<td>WHILELO on page C8-4523</td>
</tr>
<tr>
<td>WHILELS</td>
<td>While incrementing unsigned scalar lower than or the same as scalar</td>
<td>WHILELS on page C8-4525</td>
</tr>
<tr>
<td>WHILELT</td>
<td>While incrementing signed scalar less than scalar</td>
<td>WHILELT on page C8-4527</td>
</tr>
</tbody>
</table>

Table C3-141 Single loop instructions

C3-360 Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. ARM DDI 0487I.a Non-Confidential
The BRKN instructions propagate the result of a previous BRK8 or BRKPB instruction by setting the destination predicate register to all FALSE if the Last active element of their first source predicate register is not TRUE, but otherwise leave the destination predicate unchanged. The destination and second source predicate must have been created by another instruction, such as RDFFR or WHILE.

### Table C3-142 Data-independent loop instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRKA</td>
<td>Break after the first true condition</td>
<td>BRKA on page C8-3066</td>
</tr>
<tr>
<td>BRKAS</td>
<td>Break after the first true condition, setting the condition flags</td>
<td>BRKAS on page C8-3068</td>
</tr>
<tr>
<td>BRKB</td>
<td>Break before the first true condition</td>
<td>BRKB on page C8-3070</td>
</tr>
<tr>
<td>BRKBS</td>
<td>Break before the first true condition, setting the condition flags</td>
<td>BRKBS on page C8-3072</td>
</tr>
<tr>
<td>BRKN</td>
<td>Propagate break to next partition</td>
<td>BRKN on page C8-3074</td>
</tr>
<tr>
<td>BRKNS</td>
<td>Propagate break to next partition, setting the condition flags</td>
<td>BRKNS on page C8-3075</td>
</tr>
<tr>
<td>BRKPA</td>
<td>Break after the first true condition, propagating from previous partition</td>
<td>BRKPA on page C8-3075</td>
</tr>
<tr>
<td>BRKPAS</td>
<td>Break after the first true condition, propagating from previous partition, setting the condition flags</td>
<td>BRKPAS on page C8-3077</td>
</tr>
<tr>
<td>BRKP8</td>
<td>Break before the first true condition, propagating from the previous partition</td>
<td>BRKP8 on page C8-3079</td>
</tr>
<tr>
<td>BRKPBS</td>
<td>Break before the first true condition, propagating from the previous partition, setting the condition flags</td>
<td>BRKPBS on page C8-3080</td>
</tr>
</tbody>
</table>

#### Serialized operations

TheSerialized operation instructions permit Active elements within a vector to be processed sequentially without unpacking the vector. The condition flags are unconditionally set to control a subsequent conditional branch.

### Table C3-143 Serialized operation instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFIRST</td>
<td>Set the First active element to TRUE</td>
<td>PFIRST on page C8-3796</td>
</tr>
<tr>
<td>PNEXT</td>
<td>Find next Active element</td>
<td>PNEXT on page C8-3803</td>
</tr>
<tr>
<td>CTERMNE</td>
<td>Compare and terminate loop when not equal</td>
<td>CTERMNE on page C8-3158</td>
</tr>
</tbody>
</table>

### C3.7.2 Move operations

#### Element permute and shuffle

The Element permute and shuffle instructions move data between different vector elements, or between vector elements and scalar registers.

These instructions perform the following operations:

- Conditionally extract the Last active element of a vector or the following element.
  - The supported instructions are CLASTA and CLASTB.
- Unconditionally extract the Last active element of a vector or the following element.
  - The supported instructions are LASTA and LASTB.
• Variable permute instructions where the permutation is determined by the values in a predicate register or a table of element index values.
  — The supported instructions are COMPACT, SPLICE, and TBL.
• Fixed permute instructions where the form of the permutation is encoded in the instruction.
  — The supported instructions are: DUP, EXT, INSR, REV, REVb, REVh, REW, SUNPKHI, SUNPKLO, TRN1, TRN2, UUNPKHT, UUNPKLO, UZP1, UZP2, ZIP1, and ZIP2.

Table C3-144 Element permute and shuffle instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASTA</td>
<td>Conditionally extract element after the Last active element to general-purpose register</td>
<td>CLASTA (scalar) on page C8-3095</td>
</tr>
<tr>
<td></td>
<td>Conditionally extract element after the Last active element to SIMD&amp;FP scalar</td>
<td>CLASTA (SIMD&amp;FP scalar) on page C8-3097</td>
</tr>
<tr>
<td></td>
<td>Conditionally extract element after the Last active element to vector</td>
<td>CLASTA (vectors) on page C8-3099</td>
</tr>
<tr>
<td>CLASTB</td>
<td>Conditionally extract Last active element to general-purpose register</td>
<td>CLASTB (scalar) on page C8-3101</td>
</tr>
<tr>
<td></td>
<td>Conditionally extract Last active element to SIMD&amp;FP scalar</td>
<td>CLASTB (SIMD&amp;FP scalar) on page C8-3103</td>
</tr>
<tr>
<td></td>
<td>Conditionally extract Last active element to vector</td>
<td>CLASTB (vectors) on page C8-3105</td>
</tr>
<tr>
<td>LASTA</td>
<td>Extract element after the Last active element to general-purpose register</td>
<td>LASTA (scalar) on page C8-3401</td>
</tr>
<tr>
<td></td>
<td>Extract element after the Last active element to SIMD&amp;FP scalar</td>
<td>LASTA (SIMD&amp;FP scalar) on page C8-3403</td>
</tr>
<tr>
<td>LASTB</td>
<td>Extract Last active element to general-purpose register</td>
<td>LASTB (scalar) on page C8-3405</td>
</tr>
<tr>
<td></td>
<td>Extract Last active element to SIMD&amp;FP scalar</td>
<td>LASTA (SIMD&amp;FP scalar) on page C8-3403</td>
</tr>
<tr>
<td>COMPACT</td>
<td>Shuffle Active elements of vector to the right and fill with zeros</td>
<td>COMPACT on page C8-3149</td>
</tr>
<tr>
<td>SPLICE</td>
<td>Splice two vectors under predicate control</td>
<td>SPLICE on page C8-3988</td>
</tr>
<tr>
<td>TBL</td>
<td>Programmable table lookup using vector of element indexes</td>
<td>TBL on page C8-4298</td>
</tr>
<tr>
<td>DUP</td>
<td>Broadcast indexed vector element</td>
<td>DUP (indexed) on page C8-3171</td>
</tr>
<tr>
<td>EXT</td>
<td>Extract vector from pair of vectors</td>
<td>EXT on page C8-3196</td>
</tr>
<tr>
<td>INSR</td>
<td>Insert general-purpose register into shifted vector</td>
<td>INSR (scalar) on page C8-3397</td>
</tr>
<tr>
<td></td>
<td>Insert SIMD&amp;FP scalar register into shifted vector</td>
<td>INSR (SIMD&amp;FP scalar) on page C8-3399</td>
</tr>
<tr>
<td>MOV</td>
<td>Move indexed element or SIMD&amp;FP scalar to vector (unpredicated)</td>
<td>MOV (SIMD&amp;FP scalar, unpredicated) on page C8-3739</td>
</tr>
<tr>
<td></td>
<td>Move SIMD&amp;FP scalar register to vector elements (predicated)</td>
<td>MOV (SIMD&amp;FP scalar, predicated) on page C8-3737</td>
</tr>
<tr>
<td>REV</td>
<td>Reverse all elements in vector</td>
<td>REV (vector) on page C8-3863</td>
</tr>
<tr>
<td>REVb</td>
<td>Reverse 8-bit bytes in elements</td>
<td>REVb, REVh, REW on page C8-3864</td>
</tr>
<tr>
<td>REVh</td>
<td>Reverse 16-bit halfwords in elements</td>
<td>REVb, REVh, REW on page C8-3864</td>
</tr>
<tr>
<td>REW</td>
<td>Reverse 32-bit words in elements</td>
<td>REVb, REVh, REW on page C8-3864</td>
</tr>
</tbody>
</table>
The Unpacking instructions unpack half of the elements from the source vector register or predicate register, widen the unpacked elements to twice the width, and place the result in the destination register.

### Table C3-145 Unpacking instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNPKHI</td>
<td>Unpack and sign-extend elements from high half of vector</td>
<td>SUNPKHI, SUNPKLO on page C8-4291</td>
</tr>
<tr>
<td>SUNPKLO</td>
<td>Unpack and sign-extend elements from low half of vector</td>
<td>SUNPKHI, SUNPKLO on page C8-4291</td>
</tr>
<tr>
<td>UUNPKHI</td>
<td>Unpack and zero-extend elements from high half of vector</td>
<td>UUNPKHI, UUNPKLO on page C8-4503</td>
</tr>
<tr>
<td>UUNPKLO</td>
<td>Unpack and zero-extend elements from low half of vector</td>
<td>UUNPKHI, UUNPKLO on page C8-4503</td>
</tr>
<tr>
<td>PUNPKHI</td>
<td>Unpack and widen elements from high half of predicate</td>
<td>PUNPKHI, PUNPKLO on page C8-3848</td>
</tr>
<tr>
<td>PUNPKLO</td>
<td>Unpack and widen elements from low half of predicate</td>
<td>PUNPKHI, PUNPKLO on page C8-3848</td>
</tr>
</tbody>
</table>

The Predicate instructions are used to move and permute predicate elements. These instructions generally mirror the fixed vector permutes to allow predicates to follow their data. The permutes move all of the bits in a predicate element, not just the canonical bits.

### Table C3-146 Predicate instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>REV</td>
<td>Reverse all elements in predicate</td>
<td>REV (predicate) on page C8-3862</td>
</tr>
<tr>
<td>TRN1</td>
<td>Interleave even elements from two predicates</td>
<td>TRN1, TRN2 (predicates) on page C8-4302</td>
</tr>
<tr>
<td>TRN2</td>
<td>Interleave odd elements from two predicates</td>
<td>TRN1, TRN2 (predicates) on page C8-4302</td>
</tr>
<tr>
<td>UZP1</td>
<td>Select even elements from two predicates</td>
<td>UZP1, UZP2 (predicates) on page C8-4508</td>
</tr>
<tr>
<td>UZP2</td>
<td>Select odd elements from two predicates</td>
<td>UZP1, UZP2 (predicates) on page C8-4508</td>
</tr>
<tr>
<td>ZIP1</td>
<td>Interleave elements from low halves of two predicates</td>
<td>ZIP1, ZIP2 (predicates) on page C8-4536</td>
</tr>
<tr>
<td>ZIP2</td>
<td>Interleave elements from high halves of two predicates</td>
<td>ZIP1, ZIP2 (predicates) on page C8-4536</td>
</tr>
</tbody>
</table>
**Index vector generation**

The **INDEX** instruction initializes a vector horizontally by setting its first element to an integer value, and then repeatedly incrementing it by a second integer value to generate the subsequent elements. Each integer value can be specified as a signed immediate or a general-purpose register.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX</td>
<td>Create index vector starting from and incremented by immediates</td>
<td><strong>INDEX</strong> <em>(immediates)</em> on page C8-3391</td>
</tr>
<tr>
<td></td>
<td>Create index vector starting from immediate and incremented by general-purpose register</td>
<td><strong>INDEX</strong> <em>(scalars)</em> on page C8-3395</td>
</tr>
<tr>
<td></td>
<td>Create index vector starting from general-purpose register and incremented by immediate</td>
<td><strong>INDEX</strong> <em>(scalar, immediate)</em> on page C8-3393</td>
</tr>
<tr>
<td></td>
<td>Create index vector starting from immediate and incremented by general-purpose registers</td>
<td><strong>INDEX</strong> <em>(immediate, scalar)</em> on page C8-3389</td>
</tr>
</tbody>
</table>

**Move prefix**

The **MOVPRFX** (predicated) instruction is a predicated vector move that can be combined with a predicated destructive instruction that immediately follows it, in program order, to create a single constructive operation, or to convert an instruction with merging predication to use zeroing predication.

The **MOVPRFX** (unpredicated) instruction is an unpredicated vector move that can be combined with a predicated or unpredicated destructive instruction that immediately follows it, in program order, to create a single constructive operation.

The *Operational information* section of an SVE instruction description indicates whether or not an instruction can be predictably prefixed by a **MOVPRFX** instruction. If the *Operational information* of an SVE instruction description does not mention **MOVPRFX** or if the section does not exist, then the instruction cannot be predictably prefixed by a **MOVPRFX** instruction.

The prefixed instruction that immediately follows a **MOVPRFX** instruction in program order must be an implemented SVE instruction that can be predictably prefixed by a **MOVPRFX** instruction, or an A64 **HLT** instruction, or an A64 **BRK** instruction. For an SVE instruction that can be predictably prefixed by a **MOVPRFX** instruction, all of the following apply:

- The destination register field implicitly specifies one of the source operands, which means that it is a destructive binary or ternary vector operation or unary operation with merging predication, excluding **MOVPRFX**.
- The destination register is the same as the **MOVPRFX** destination register.
- The prefixed instruction does not use the **MOVPRFX** destination register in any of its other source register fields, even if it has a different name but refers to the same architectural register state. For example, Z1, V1, and D1 all refer to the same architectural register.
- If the **MOVPRFX** instruction is predicated, then the prefixed instruction is predicated using the same *Governing predicate* register, and the maximum encoded element size is the same as the **MOVPRFX** element size, excluding the fixed-size 64-bit elements of the wide elements form of bitwise shift and integer compare operations.
- If the **MOVPRFX** instruction is unpredicated, then the prefixed instruction can use any *Governing predicate* register and element size, or it can be unpredicated. A predicated **MOVPRFX** cannot be used with an unpredicated instruction.

If the instruction that follows a **MOVPRFX** instruction is not an implemented SVE instruction that can be predictably prefixed by a **MOVPRFX** instruction, the two instructions behave in one of the following **CONSTRAINED UNPREDICTABLE** ways:

- Either or both instructions can execute with their individually described effects.
Either instruction can generate an Undefined Instruction exception.
Either or both instructions can execute as a NOP.
The second instruction can execute with an UNKNOWN value for any of its source registers.
Any register that is written by either or both instructions can be set to an unknown value.
A control flow instruction that writes the PC can set the PC to an unknown value.

Table C3-148 Move prefix instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVPRFX</td>
<td>Move prefix (predicated)</td>
<td>MOVPRFX (predicated) on page C8-3743</td>
</tr>
<tr>
<td></td>
<td>Move prefix (unpredicated)</td>
<td>MOVPRFX (unpredicated) on page C8-3745</td>
</tr>
</tbody>
</table>

Unless the combination of a constructive operation with merging predication is specifically required, it is strongly recommended that, for performance reasons, software should prefer to use the zeroing form of predicated MOVPRFX or the unpredicated MOVPRFX instruction.

When a MOVPRFX instruction is executed, except for PMU events SVE_MVPREFX_SPEC, SVE_MVPREFX_Z_SPEC, SVE_MVPREFX_M_SPEC, and SVE_MVPREFX_U_SPEC, 0x807C-0x807F, it is IMPLEMENTATION DEFINED for each execution of the instruction whether or not any Performance Monitor counts the instruction. This can vary dynamically for each execution of the same instruction.

When a microarchitectural operation is executed because of a MOVPRFX instruction, except for PMU events SVE_MVPREFX_SPEC, SVE_MVPREFX_Z_SPEC, SVE_MVPREFX_M_SPEC, and SVE_MVPREFX_U_SPEC, 0x807C-0x807F, it is implementation defined for each execution of the operation whether or not the Performance Monitor counts the operation. This can vary dynamically for each execution of the same instruction.

MOVPRFX instruction behavior in self-hosted debug

A MOVPRFX instruction can legally prefix a BRK or HLT instruction.

If a hardware breakpoint is programmed with the address of a legal MOVPRFX instruction, when any of the following events occur, the hardware breakpoint generates a Breakpoint exception:

- The MOVPRFX instruction is committed for execution.
- The combined MOVPRFX and Prefixed instruction is committed for execution.

If a hardware breakpoint is programmed with the address of an illegal MOVPRFX instruction or a Prefixed instruction, when any of the MOVPRFX instruction and Prefixed instruction are committed for execution, it is CONSTRAINED UNPREDICTABLE whether or not the hardware breakpoint generates a Breakpoint exception.

If a single-step is performed for a MOVPRFX instruction, it is CONSTRAINED UNPREDICTABLE whether the PE steps over the pair of instructions or steps over only the MOVPRFX instruction.

C3.7.3 Reduction operations

Horizontal reductions

The Horizontal reduction instructions perform arithmetic horizontally across Active elements of a single source vector and deliver a scalar result.

The floating-point horizontal accumulating sum instruction, FADDA, operates strictly in order of increasing element number across a vector, using the scalar destination register as a source for the initial value of the accumulator. This preserves the original program evaluation order where non-associativity is required.

The other floating-point reductions calculate their result using a recursive pair-wise algorithm that does not preserve the original program order, but permits increased parallelism for code that does not require strict order of evaluation.

Integer reductions are fully associative, and the order of evaluation is not specified by the architecture.
### Table C3-149 Horizontal reduction instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANDV</td>
<td>Bitwise AND reduction, treating <em>Inactive elements</em> as all ones</td>
<td><em>ANDV</em> on page C8-3016</td>
</tr>
<tr>
<td>EORV</td>
<td>Bitwise XOR reduction, treating <em>Inactive elements</em> as zero</td>
<td><em>EORV</em> on page C8-3194</td>
</tr>
<tr>
<td>FADDA</td>
<td>Floating-point add strictly-ordered reduction, accumulating in scalar, ignoring <em>Inactive elements</em></td>
<td><em>FADDA</em> on page C8-3211</td>
</tr>
<tr>
<td>FADDV</td>
<td>Floating-point add recursive reduction, treating <em>Inactive elements</em> as +0.0</td>
<td><em>FADDV</em> on page C8-3215</td>
</tr>
<tr>
<td>FMAXNMV</td>
<td>Floating-point maximum number recursive reduction, treating <em>Inactive elements</em> as the default NaN</td>
<td><em>FMAXNMV</em> on page C8-3274</td>
</tr>
<tr>
<td>FMAXV</td>
<td>Floating-point maximum recursive reduction, treating <em>Inactive elements</em> as negative infinity</td>
<td><em>FMAXV</em> on page C8-3277</td>
</tr>
<tr>
<td>FMINNMV</td>
<td>Floating-point minimum number recursive reduction, treating <em>Inactive elements</em> as the default NaN</td>
<td><em>FMINNMV</em> on page C8-3288</td>
</tr>
<tr>
<td>FMINV</td>
<td>Floating-point minimum recursive reduction, treating <em>Inactive elements</em> as positive infinity</td>
<td><em>FMINV</em> on page C8-3291</td>
</tr>
<tr>
<td>ORV</td>
<td>Bitwise OR reduction, treating <em>Inactive elements</em> as zero</td>
<td><em>ORV</em> on page C8-3793</td>
</tr>
<tr>
<td>SADDV</td>
<td>Signed add reduction, treating <em>Inactive elements</em> as zero</td>
<td><em>SADDV</em> on page C8-3897</td>
</tr>
<tr>
<td>SMAXV</td>
<td>Signed maximum reduction, treating <em>Inactive elements</em> as the minimum signed integer</td>
<td><em>SMAXV</em> on page C8-3948</td>
</tr>
<tr>
<td>SMINV</td>
<td>Signed minimum reduction, treating <em>Inactive elements</em> the maximum signed integer</td>
<td><em>SMINV</em> on page C8-3956</td>
</tr>
<tr>
<td>UADDV</td>
<td>Unsigned add reduction, treating <em>Inactive elements</em> as zero</td>
<td><em>UADDV</em> on page C8-4325</td>
</tr>
<tr>
<td>UMAXV</td>
<td>Unsigned maximum reduction, treating <em>Inactive elements</em> as zero</td>
<td><em>UMAXV</em> on page C8-4357</td>
</tr>
<tr>
<td>UMINV</td>
<td>Unsigned minimum reduction, treating <em>Inactive elements</em> as the maximum unsigned integer</td>
<td><em>UMINV</em> on page C8-4365</td>
</tr>
</tbody>
</table>
C3.8 Data processing - SVE2

The following subsections describe the SVE2 processing instructions:

- Down-counting loops.
- Constructive multiply on page C3-369.
- Uniform DSP operations on page C3-369.
- Widening DSP operations on page C3-371.
- Narrowing DSP operation on page C3-372.
- Unary narrowing operations on page C3-373.
- Non-widening pairwise arithmetic on page C3-374.
- Widening pairwise arithmetic on page C3-374.
- Bitwise ternary logical instructions on page C3-375.
- Large integer arithmetic on page C3-375.
- Multiplication by indexed elements on page C3-375.
- Complex integer arithmetic on page C3-376.
- Floating-point extra conversions on page C3-379.
- Floating-point widening multiply-accumulate on page C3-379.
- Floating-point integer binary logarithm on page C3-380.
- Cross-lane match detect on page C3-380.
- Bit permutation on page C3-382.
- Polynomial arithmetic on page C3-382.
- Vector concatenation on page C3-383.
- Extended table lookup permute on page C3-383.
- Non-temporal gather/scatter on page C3-383.
- Cryptography support on page C3-384.

C3.8.1 Down-counting loops

A down-counting WHILE instruction decrements the value of the first scalar operand and compares the value with a second, fixed scalar operand. The instruction generates a destination predicate with all of the following characteristics:

- The predicate elements starting from the highest-numbered element are true while the comparison remains true.
- The predicate elements thereafter, down to the lowest-numbered element, are false when the comparison becomes false.

All 32 bits or 64 bits of the scalar operands are significant for the purposes of comparison. The full 32-bit or 64-bit value of the first operand is decremented by 1 for each destination predicate element, irrespective of the element size. The first general-purpose register operand is not updated.

If all of the following occur, a comparison can never fail, resulting in an all-true predicate:

- The comparison includes an equality test.
- The second scalar operand is equal to the minimum integer value of the selected size and type of comparison.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHILEGE</td>
<td>While decrementing signed 32-bit or 64-bit scalar greater than or equal to scalar</td>
<td>WHILEGE on page C8-4513</td>
</tr>
<tr>
<td>WHILEGT</td>
<td>While decrementing signed 32-bit or 64-bit scalar greater than scalar</td>
<td>WHILEGT on page C8-4515</td>
</tr>
<tr>
<td>WHILEHI</td>
<td>While decrementing unsigned 32-bit or 64-bit scalar higher than scalar</td>
<td>WHILEHI on page C8-4517</td>
</tr>
<tr>
<td>WHILEHS</td>
<td>While decrementing unsigned 32-bit or 64-bit scalar higher or same as scalar</td>
<td>WHILEHS on page C8-4519</td>
</tr>
</tbody>
</table>
C3.8.2 Constructive multiply

SVE2 includes the following constructive, three-operand versions of the integer multiply instructions:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>Multiply vectors (unpredicated)</td>
<td>MUL (vectors, unpredicated) on page C8-3757</td>
</tr>
<tr>
<td>SMULH</td>
<td>Signed multiply returning high half (unpredicated)</td>
<td>SMULH (unpredicated) on page C8-3978</td>
</tr>
<tr>
<td>UMULH</td>
<td>Unsigned multiply returning high half (unpredicated)</td>
<td>UMULH (unpredicated) on page C8-4387</td>
</tr>
</tbody>
</table>

C3.8.3 Uniform DSP operations

The uniform DSP instructions are based on AArch64 Advanced SIMD instructions with the same mnemonic. The instructions operate on integer operands and produce results with a uniform element size. The operation of an instruction might include one or more of rounding, halving, saturation, and accumulation.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SABA</td>
<td>Signed absolute difference and accumulate</td>
<td>SABA on page C8-3877</td>
</tr>
<tr>
<td>SHADD</td>
<td>Signed halving addition</td>
<td>SHADD on page C8-3926</td>
</tr>
<tr>
<td>SHSUB</td>
<td>Signed halving subtract</td>
<td>SHSUB on page C8-3932</td>
</tr>
<tr>
<td>SHSUBR</td>
<td>Signed halving subtract reversed vectors</td>
<td>SHSUBR on page C8-3934</td>
</tr>
<tr>
<td>SLT</td>
<td>Shift left and insert (immediate)</td>
<td>SLT on page C8-3936</td>
</tr>
<tr>
<td>SQABS</td>
<td>Signed saturating absolute value</td>
<td>SQABS on page C8-3990</td>
</tr>
<tr>
<td>SQADD</td>
<td>Signed saturating addition (predicated)</td>
<td>SQADD (vectors, predicated) on page C8-3994</td>
</tr>
<tr>
<td>SQDMULH</td>
<td>Signed saturating doubling multiply high (unpredicated)</td>
<td>SQDMULH (vectors) on page C8-4040</td>
</tr>
<tr>
<td>SQNEG</td>
<td>Signed saturating negate</td>
<td>SQNEG on page C8-4068</td>
</tr>
<tr>
<td>SQRDMLAH</td>
<td>Signed saturating rounding doubling multiply-add high to accumulator (unpredicated)</td>
<td>SQRDMLAH (indexed) on page C8-4075</td>
</tr>
<tr>
<td>SQRDMLSH</td>
<td>Signed saturating rounding doubling multiply-subtract high from accumulator (unpredicated)</td>
<td>SQRDMLSH (vectors) on page C8-4083</td>
</tr>
<tr>
<td>SQRDMULH</td>
<td>Signed saturating rounding doubling multiply high (unpredicated)</td>
<td>SQRDMULH (vectors) on page C8-4088</td>
</tr>
<tr>
<td>SQRSHL</td>
<td>Signed saturating rounding shift left by vector (predicated)</td>
<td>SQRSHL on page C8-4090</td>
</tr>
<tr>
<td>SQRSHLR</td>
<td>Signed saturating rounding shift left reversed vectors (predicated)</td>
<td>SQRSHLR on page C8-4092</td>
</tr>
<tr>
<td>SQSHL</td>
<td>Signed saturating shift left by immediate</td>
<td>SQSHL (immediate) on page C8-4102</td>
</tr>
<tr>
<td>SQSHL</td>
<td>Signed saturating shift left by vector (predicated)</td>
<td>SQSHL (vectors) on page C8-4104</td>
</tr>
<tr>
<td>SQSHLR</td>
<td>Signed saturating shift left reversed vectors (predicated)</td>
<td>SQSHLR on page C8-4106</td>
</tr>
<tr>
<td>SQSHLU</td>
<td>Signed saturating shift left unsigned by immediate</td>
<td>SQSHLU on page C8-4108</td>
</tr>
<tr>
<td>SQSUB</td>
<td>Signed saturating subtraction (predicated)</td>
<td>SQSUB (vectors, predicated) on page C8-4120</td>
</tr>
</tbody>
</table>
Table C3-152 Uniform DSP instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQSUBR</td>
<td>Signed saturating subtraction reversed vectors (predicated)</td>
<td>SQSUBR on page C8-4123</td>
</tr>
<tr>
<td>SRHADD</td>
<td>Signed rounding halving addition</td>
<td>SRHADD on page C8-4133</td>
</tr>
<tr>
<td>SRI</td>
<td>Shift right and insert (immediate)</td>
<td>SRI on page C8-4135</td>
</tr>
<tr>
<td>SRSHL</td>
<td>Signed rounding shift left by vector (predicated)</td>
<td>SRSHL on page C8-4137</td>
</tr>
<tr>
<td>SRSHLR</td>
<td>Signed rounding shift left reversed vectors (predicated)</td>
<td>SRSHLR on page C8-4139</td>
</tr>
<tr>
<td>SRSHR</td>
<td>Signed rounding shift right by immediate</td>
<td>SRSHR on page C8-4141</td>
</tr>
<tr>
<td>SRSRA</td>
<td>Signed rounding shift right and accumulate (immediate)</td>
<td>SRSRA on page C8-4143</td>
</tr>
<tr>
<td>SSRA</td>
<td>Signed shift right and accumulate (immediate)</td>
<td>SSRA on page C8-4149</td>
</tr>
<tr>
<td>SUQADD</td>
<td>Signed saturating addition of unsigned value</td>
<td>SUQADD on page C8-4293</td>
</tr>
<tr>
<td>UABA</td>
<td>Unsigned absolute difference and accumulate</td>
<td>UABA on page C8-4307</td>
</tr>
<tr>
<td>UHADD</td>
<td>Unsigned halving addition</td>
<td>UHADD on page C8-4345</td>
</tr>
<tr>
<td>UHSUB</td>
<td>Unsigned halving subtract</td>
<td>UHSUB on page C8-4347</td>
</tr>
<tr>
<td>UHSUBR</td>
<td>Unsigned halving subtract reversed vectors</td>
<td>UHSUBR on page C8-4349</td>
</tr>
<tr>
<td>UQADD</td>
<td>Unsigned saturating addition (predicated)</td>
<td>UQADD (vectors, predicated) on page C8-4399</td>
</tr>
<tr>
<td>UQRSHL</td>
<td>Unsigned saturating rounding shift left by vector (predicated)</td>
<td>UQRSHL on page C8-4438</td>
</tr>
<tr>
<td>UQRSHLR</td>
<td>Unsigned saturating rounding shift left reversed vectors (predicated)</td>
<td>UQRSHLR on page C8-4440</td>
</tr>
<tr>
<td>UQSHL</td>
<td>Unsigned saturating shift left by immediate</td>
<td>UQSHL (immediate) on page C8-4446</td>
</tr>
<tr>
<td>UQSLH</td>
<td>Unsigned saturating shift left by vector (predicated)</td>
<td>UQSLH (vectors) on page C8-4448</td>
</tr>
<tr>
<td>UQSHLR</td>
<td>Unsigned saturating shift left reversed vectors (predicated)</td>
<td>UQSHLR on page C8-4450</td>
</tr>
<tr>
<td>UQSUB</td>
<td>Unsigned saturating subtraction (predicated)</td>
<td>UQSUB (vectors, predicated) on page C8-4458</td>
</tr>
<tr>
<td>UQSUBR</td>
<td>Unsigned saturating subtraction reversed vectors (predicated)</td>
<td>UQSUBR on page C8-4461</td>
</tr>
<tr>
<td>URECPE</td>
<td>Unsigned reciprocal estimate (predicated)</td>
<td>URECPE on page C8-4467</td>
</tr>
<tr>
<td>URHADD</td>
<td>Unsigned rounding halving addition.</td>
<td>URHADD on page C8-4469</td>
</tr>
<tr>
<td>URSHL</td>
<td>Unsigned rounding shift left by vector (predicated)</td>
<td>URSHL on page C8-4471</td>
</tr>
<tr>
<td>URSHLR</td>
<td>Unsigned rounding shift left reversed vectors (predicated)</td>
<td>URSHLR on page C8-4473</td>
</tr>
<tr>
<td>URSHR</td>
<td>Unsigned rounding shift right by immediate</td>
<td>URSHR on page C8-4475</td>
</tr>
<tr>
<td>URSQRT</td>
<td>Unsigned reciprocal square root estimate (predicated)</td>
<td>URSQRT on page C8-4477</td>
</tr>
<tr>
<td>URSRA</td>
<td>Unsigned rounding shift right and accumulate (immediate)</td>
<td>URSRA on page C8-4479</td>
</tr>
<tr>
<td>USQADD</td>
<td>Unsigned saturating addition of signed value</td>
<td>USQADD on page C8-4491</td>
</tr>
<tr>
<td>USRA</td>
<td>Unsigned shift right and accumulate (immediate)</td>
<td>USRA on page C8-4493</td>
</tr>
</tbody>
</table>
### C3.8.4 Widening DSP operations

The widening DSP instructions are based on AArch64 Advanced SIMD instructions with similar mnemonics. The instructions operate on integer values and produce results that are twice the width of some or all of the inputs. The instructions read the narrow inputs from either the even-numbered (bottom) or odd-numbered (top) source elements and place each result in the double-width destination elements that overlap the narrow source elements.

#### Table C3-153 Widening DSP instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SABALB</td>
<td>Signed absolute difference and accumulate long (bottom)</td>
<td>SABALB on page C8-3879</td>
</tr>
<tr>
<td>SABALT</td>
<td>Signed absolute difference and accumulate long (top)</td>
<td>SABALT on page C8-3881</td>
</tr>
<tr>
<td>SABDLB</td>
<td>Signed absolute difference long (bottom)</td>
<td>SABDLB on page C8-3885</td>
</tr>
<tr>
<td>SABDLT</td>
<td>Signed absolute difference long (top)</td>
<td>SABDLT on page C8-3887</td>
</tr>
<tr>
<td>SADDB</td>
<td>Signed add long (bottom)</td>
<td>SADDB on page C8-3891</td>
</tr>
<tr>
<td>SADDLT</td>
<td>Signed add long (top)</td>
<td>SADDLT on page C8-3895</td>
</tr>
<tr>
<td>SADDWb</td>
<td>Signed add wide (bottom)</td>
<td>SADDWb on page C8-3899</td>
</tr>
<tr>
<td>SADDT</td>
<td>Signed add wide (top)</td>
<td>SADDT on page C8-3901</td>
</tr>
<tr>
<td>SMLALB</td>
<td>Signed multiply-add long to accumulator (bottom)</td>
<td>SMLALB (vectors) on page C8-3960</td>
</tr>
<tr>
<td>SMLALT</td>
<td>Signed multiply-add long to accumulator (top)</td>
<td>SMLALT (vectors) on page C8-3964</td>
</tr>
<tr>
<td>SMLSLB</td>
<td>Signed multiply-subtract long from accumulator (bottom)</td>
<td>SMLSLB (vectors) on page C8-3968</td>
</tr>
<tr>
<td>SMLSLT</td>
<td>Signed multiply-subtract long from accumulator (top)</td>
<td>SMLSLT (vectors) on page C8-3972</td>
</tr>
<tr>
<td>SMLLB</td>
<td>Signed multiply long (bottom)</td>
<td>SMLLB (vectors) on page C8-3982</td>
</tr>
<tr>
<td>SMLLT</td>
<td>Signed multiply long (top)</td>
<td>SMLLT (vectors) on page C8-3986</td>
</tr>
<tr>
<td>SQDMLALB</td>
<td>Signed saturating doubling multiply-add long to accumulator (bottom)</td>
<td>SQDMLALB (vectors) on page C8-4019</td>
</tr>
<tr>
<td>SQDMLALT</td>
<td>Signed saturating doubling multiply-add long to accumulator (top)</td>
<td>SQDMLALT (vectors) on page C8-4025</td>
</tr>
<tr>
<td>SQDMLSLB</td>
<td>Signed saturating doubling multiply-subtract long from accumulator (bottom)</td>
<td>SQDMLSLB (vectors) on page C8-4029</td>
</tr>
<tr>
<td>SQDMLSLT</td>
<td>Signed saturating doubling multiply-subtract long from accumulator (top)</td>
<td>SQDMLSLT (vectors) on page C8-4035</td>
</tr>
<tr>
<td>SQDMULLB</td>
<td>Signed saturating doubling multiply long (bottom)</td>
<td>SQDMULLB (vectors) on page C8-4044</td>
</tr>
<tr>
<td>SQDMULLT</td>
<td>Signed saturating doubling multiply long (top)</td>
<td>SQDMULLT (vectors) on page C8-4048</td>
</tr>
<tr>
<td>SSHLLB</td>
<td>Signed shift left long by immediate (bottom)</td>
<td>SSHLLB on page C8-4145</td>
</tr>
<tr>
<td>SSHLLT</td>
<td>Signed shift left long by immediate (top)</td>
<td>SSHLLT on page C8-4147</td>
</tr>
<tr>
<td>SSUBLB</td>
<td>Signed subtract long (bottom)</td>
<td>SSUBLB on page C8-4151</td>
</tr>
<tr>
<td>SSUBLT</td>
<td>Signed subtract long (top)</td>
<td>SSUBLT on page C8-4155</td>
</tr>
<tr>
<td>SSUBWB</td>
<td>Signed subtract wide (bottom)</td>
<td>SSUBWB on page C8-4159</td>
</tr>
<tr>
<td>SSUBWT</td>
<td>Signed subtract wide (top)</td>
<td>SSUBWT on page C8-4161</td>
</tr>
<tr>
<td>UABALB</td>
<td>Unsigned absolute difference and accumulate long (bottom)</td>
<td>UABALB on page C8-4309</td>
</tr>
</tbody>
</table>
C3.8.5 Narrowing DSP operation

The narrowing DSP instructions are based on AArch64 Advanced SIMD instructions with similar mnemonics. The instructions operate on integer values and produce results that are half the width of the inputs. The instructions read wide source elements and place each result in one of the following:

- The overlapped even-numbered (bottom) half-width destination elements. The odd-numbered destination elements are set to zero.
- The overlapped odd-numbered (top) half-width destination elements. The even-numbered destination elements are unchanged, which means that the instructions are implicitly merging operations.

### Table C3-153 Widening DSP instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>UABALT</td>
<td>Unsigned absolute difference and accumulate long (top)</td>
<td>UABALT on page C8-4311</td>
</tr>
<tr>
<td>UABDLB</td>
<td>Unsigned absolute difference long (bottom)</td>
<td>UABDLB on page C8-4315</td>
</tr>
<tr>
<td>UABDLT</td>
<td>Unsigned absolute difference long (top)</td>
<td>UABDLT on page C8-4317</td>
</tr>
<tr>
<td>UADDLB</td>
<td>Unsigned add long (bottom)</td>
<td>UADDLB on page C8-4321</td>
</tr>
<tr>
<td>UADDLT</td>
<td>Unsigned add long (top)</td>
<td>UADDLT on page C8-4323</td>
</tr>
<tr>
<td>UADDWB</td>
<td>Unsigned add wide (bottom)</td>
<td>UADDWB on page C8-4327</td>
</tr>
<tr>
<td>UADDWT</td>
<td>Unsigned add wide (top)</td>
<td>UADDWT on page C8-4329</td>
</tr>
<tr>
<td>UMLALB</td>
<td>Unsigned multiply-add long to accumulator (bottom)</td>
<td>UMLALB (vectors) on page C8-4369</td>
</tr>
<tr>
<td>UMLALT</td>
<td>Unsigned multiply-add long to accumulator (top)</td>
<td>UMLALT (vectors) on page C8-4373</td>
</tr>
<tr>
<td>UMLSLB</td>
<td>Unsigned multiply-subtract long from accumulator (bottom)</td>
<td>UMLSLB (vectors) on page C8-4377</td>
</tr>
<tr>
<td>UMLSLT</td>
<td>Unsigned multiply-subtract long from accumulator (top)</td>
<td>UMLSLT (vectors) on page C8-4381</td>
</tr>
<tr>
<td>UMULLB</td>
<td>Unsigned multiply long (bottom)</td>
<td>UMULLB (vectors) on page C8-4391</td>
</tr>
<tr>
<td>UMULLT</td>
<td>Unsigned multiply long (top)</td>
<td>UMULLT (vectors) on page C8-4395</td>
</tr>
<tr>
<td>USHLLB</td>
<td>Unsigned shift left long by immediate (bottom)</td>
<td>USHLLB on page C8-4485</td>
</tr>
<tr>
<td>USHLLT</td>
<td>Unsigned shift left long by immediate (top)</td>
<td>USHLLT on page C8-4487</td>
</tr>
<tr>
<td>USUBLB</td>
<td>Unsigned subtract long (bottom)</td>
<td>USUBLB on page C8-4495</td>
</tr>
<tr>
<td>USUBLT</td>
<td>Unsigned subtract long (top)</td>
<td>USUBLT on page C8-4497</td>
</tr>
<tr>
<td>USUBWB</td>
<td>Unsigned subtract wide (bottom)</td>
<td>USUBWB on page C8-4499</td>
</tr>
<tr>
<td>USUBWT</td>
<td>Unsigned subtract wide (top)</td>
<td>USUBWT on page C8-4501</td>
</tr>
</tbody>
</table>

### Table C3-154 Narrowing DSP instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDHNB</td>
<td>Add narrow high part (bottom)</td>
<td>ADDHNB on page C8-2988</td>
</tr>
<tr>
<td>ADDHNT</td>
<td>Add narrow high part (top)</td>
<td>ADDHNT on page C8-2990</td>
</tr>
<tr>
<td>RADDHNB</td>
<td>Rounding add narrow high part (bottom)</td>
<td>RADDHNB on page C8-3850</td>
</tr>
<tr>
<td>RADDHNT</td>
<td>Rounding add narrow high part (top)</td>
<td>RADDHNT on page C8-3852</td>
</tr>
</tbody>
</table>
C3.8.6 Unary narrowing operations

The unary narrowing instructions are unpredicated and do not write to the source register. The instructions read elements from the source vector and saturate each value to the half-width destination element size. The instructions place the narrow results in one of the following:

- The overlapped even-numbered (bottom) half-width elements. The odd-numbered elements are set to zero.
- The overlapped odd-numbered (top) half-width elements. The even-numbered elements are unchanged.

Non-saturating (truncating) conversions can be performed using existing SVE instructions such as shifts, masks, and permutes.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSHRNB</td>
<td>Rounding shift right narrow by immediate (bottom)</td>
<td>RSHRNB on page C8-3869</td>
</tr>
<tr>
<td>RSHRNT</td>
<td>Rounding shift right narrow by immediate (top)</td>
<td>RSHRNT on page C8-3871</td>
</tr>
<tr>
<td>RSUBHNB</td>
<td>Rounding subtract narrow high part (bottom)</td>
<td>RSUBHNB on page C8-3873</td>
</tr>
<tr>
<td>RSUBHNT</td>
<td>Rounding subtract narrow high part (top)</td>
<td>RSUBHNT on page C8-3875</td>
</tr>
<tr>
<td>SHRNB</td>
<td>Shift right narrow by immediate (bottom)</td>
<td>SHRNB on page C8-3928</td>
</tr>
<tr>
<td>SHRNT</td>
<td>Shift right narrow by immediate (top)</td>
<td>SHRNT on page C8-3930</td>
</tr>
<tr>
<td>SQRSHRNB</td>
<td>Signed saturating rounding shift right narrow by immediate (bottom)</td>
<td>SQRSHRNB on page C8-4094</td>
</tr>
<tr>
<td>SQRSHRNT</td>
<td>Signed saturating rounding shift right narrow by immediate (top)</td>
<td>SQRSHRNT on page C8-4096</td>
</tr>
<tr>
<td>SQRSHRUNB</td>
<td>Signed saturating rounding shift right unsigned narrow by immediate (bottom)</td>
<td>SQRSHRUNB on page C8-4098</td>
</tr>
<tr>
<td>SQRSHRUNT</td>
<td>Signed saturating rounding shift right unsigned narrow by immediate (top)</td>
<td>SQRSHRUNT on page C8-4100</td>
</tr>
<tr>
<td>SQSHRNB</td>
<td>Signed saturating shift right narrow by immediate (bottom)</td>
<td>SQSHRNB on page C8-4110</td>
</tr>
<tr>
<td>SQSHRNT</td>
<td>Signed saturating shift right narrow by immediate (top)</td>
<td>SQSHRNT on page C8-4112</td>
</tr>
<tr>
<td>SQSHRUNB</td>
<td>Signed saturating shift right unsigned narrow by immediate (bottom)</td>
<td>SQSHRUNB on page C8-4114</td>
</tr>
<tr>
<td>SQSHRUNT</td>
<td>Signed saturating shift right unsigned narrow by immediate (top)</td>
<td>SQSHRUNT on page C8-4116</td>
</tr>
<tr>
<td>SUBHNB</td>
<td>Subtract narrow high part (bottom)</td>
<td>SUBHNB on page C8-4281</td>
</tr>
<tr>
<td>SUBHNT</td>
<td>Subtract narrow high part (top)</td>
<td>SUBHNT on page C8-4283</td>
</tr>
<tr>
<td>UQRSHRNB</td>
<td>Unsigned saturating rounding shift right narrow by immediate (bottom)</td>
<td>UQRSHRNB on page C8-4442</td>
</tr>
<tr>
<td>UQRSHRNT</td>
<td>Unsigned saturating rounding shift right narrow by immediate (top)</td>
<td>UQRSHRNT on page C8-4444</td>
</tr>
<tr>
<td>UQSHRNB</td>
<td>Unsigned saturating shift right narrow by immediate (bottom)</td>
<td>UQSHRNB on page C8-4452</td>
</tr>
<tr>
<td>UQSHRNT</td>
<td>Unsigned saturating shift right narrow by immediate (top)</td>
<td>UQSHRNT on page C8-4454</td>
</tr>
</tbody>
</table>

Table C3-155 Unary narrowing instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQXTNB</td>
<td>Signed saturating extract narrow (bottom)</td>
<td>SQXTNB on page C8-4125</td>
</tr>
<tr>
<td>SQXTNT</td>
<td>Signed saturating extract narrow (top)</td>
<td>SQXTNT on page C8-4127</td>
</tr>
<tr>
<td>SQXTUNB</td>
<td>Signed saturating unsigned extract narrow (bottom)</td>
<td>SQXTUNB on page C8-4129</td>
</tr>
</tbody>
</table>
### C3.8.7 Non-widening pairwise arithmetic

The non-widening pairwise arithmetic instructions operate on pairs of adjacent elements in each source vector and produce a result element that is the same size as a single input element. The results from the first and second source vectors are interleaved, so that the source and result elements overlap. The result is destructively placed in the first source vector. The AArch64 Advanced SIMD instructions do not interleave the results from the first and second source vectors. Predication applies to the destination vector. The even-numbered predicate elements enable an operation on a pair of elements in the first source vector. The odd-numbered predicate elements enable an operation on a pair of elements in the second source vector. Inactive elements in the destination vector register are not modified.

### Table C3-155 Unary narrowing instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQXTUNT</td>
<td>Signed saturating unsigned extract narrow (top)</td>
<td>SQXTUNT on page C8-4131</td>
</tr>
<tr>
<td>UQXTNB</td>
<td>Unsigned saturating extract narrow (bottom)</td>
<td>UQXTNB on page C8-4463</td>
</tr>
<tr>
<td>UQXTNT</td>
<td>Unsigned saturating extract narrow (top)</td>
<td>UQXTNT on page C8-4465</td>
</tr>
</tbody>
</table>

### C3.8.8 Widening pairwise arithmetic

The widening pairwise arithmetic instructions operate on pairs of adjacent elements in a single source vector and produce a double-width result element that is accumulated into the destination vector. Inactive elements in the destination vector register are not modified.

### Table C3-156 Non-widening pairwise arithmetic instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDP</td>
<td>Add pairwise</td>
<td>ADDP on page C8-2992</td>
</tr>
<tr>
<td>FADDP</td>
<td>Floating-point add pairwise</td>
<td>FADDP on page C8-3213</td>
</tr>
<tr>
<td>FMAXNMP</td>
<td>Floating-point maximum number pairwise</td>
<td>FMAXNMP on page C8-3272</td>
</tr>
<tr>
<td>FMAXP</td>
<td>Floating-point maximum pairwise</td>
<td>FMAXP on page C8-3275</td>
</tr>
<tr>
<td>FMINNMP</td>
<td>Floating-point minimum number pairwise</td>
<td>FMINNMP on page C8-3286</td>
</tr>
<tr>
<td>FMINP</td>
<td>Floating-point minimum pairwise</td>
<td>FMINP on page C8-3289</td>
</tr>
<tr>
<td>SMAXP</td>
<td>Signed maximum pairwise</td>
<td>SMAXP on page C8-3946</td>
</tr>
<tr>
<td>SMINP</td>
<td>Signed minimum pairwise</td>
<td>SMINP on page C8-3954</td>
</tr>
<tr>
<td>UMAXP</td>
<td>Unsigned maximum pairwise</td>
<td>UMAXP on page C8-4355</td>
</tr>
<tr>
<td>UMINTP</td>
<td>Unsigned minimum pairwise</td>
<td>UMINTP on page C8-4363</td>
</tr>
</tbody>
</table>

### Table C3-157 Widening pairwise arithmetic instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SADALP</td>
<td>Signed add and accumulate long pairwise</td>
<td>SADALP on page C8-3889</td>
</tr>
<tr>
<td>UADALP</td>
<td>Unsigned add and accumulate long pairwise</td>
<td>UADALP on page C8-4319</td>
</tr>
</tbody>
</table>
C3.8.9 Bitwise ternary logical instructions

The bitwise ternary logical instructions enable complex bit processing codes to be accelerated using multiple bitwise logical operations in a shorter instruction sequence. All of the following operations are supported by the bitwise ternary logical instructions:

- The BCAX instruction combines a ternary bitwise clear with an exclusive OR.
- The EOR3 instruction provides a ternary exclusive OR.
- The XAR instruction combines an exclusive OR with rotation by a constant amount.
- The bitwise select instructions, BSL, BSL1N, BSL2N, and NBSL, can be used with other bitwise logical instructions to generate all 256 possible bitwise combinations of three input bits using at most three instructions.

The bitwise ternary logical instructions are unpredicated.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCAX</td>
<td>Bitwise clear and exclusive OR</td>
<td>BCAX on page C8-3032</td>
</tr>
<tr>
<td>BSL</td>
<td>Bitwise select</td>
<td>BSL on page C8-3082</td>
</tr>
<tr>
<td>BSL1N</td>
<td>Bitwise select with first input inverted</td>
<td>BSL1N on page C8-3084</td>
</tr>
<tr>
<td>BSL2N</td>
<td>Bitwise select with second input inverted</td>
<td>BSL2N on page C8-3086</td>
</tr>
<tr>
<td>EOR3</td>
<td>Bitwise exclusive OR of three vectors</td>
<td>EOR3 on page C8-3186</td>
</tr>
<tr>
<td>NBSL</td>
<td>Bitwise inverted select</td>
<td>NBSL on page C8-3763</td>
</tr>
<tr>
<td>XAR</td>
<td>Bitwise exclusive OR and rotate right by immediate</td>
<td>XAR on page C8-4534</td>
</tr>
</tbody>
</table>

C3.8.10 Large integer arithmetic

The large integer arithmetic instructions aid the processing of large integers in vector registers by maintaining multiple carry chains that are interleaved in accumulator vectors. A large integer arithmetic instruction takes as input all of the following:

- Either the even-numbered (bottom) or odd-numbered (top) elements of the first source vector.
- A 1-bit carry input from the least-significant bit of the odd-numbered elements of the second source vector.

The inputs to the instruction are added to or subtracted from the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADCLB</td>
<td>Add with carry long (bottom)</td>
<td>ADCLB on page C8-2978</td>
</tr>
<tr>
<td>ADCLT</td>
<td>Add with carry long (top)</td>
<td>ADCLT on page C8-2980</td>
</tr>
<tr>
<td>SBCLB</td>
<td>Subtract with carry long (bottom)</td>
<td>SBCLB on page C8-3903</td>
</tr>
<tr>
<td>SBCLT</td>
<td>Subtract with carry long (top)</td>
<td>SBCLT on page C8-3905</td>
</tr>
</tbody>
</table>

C3.8.11 Multiplication by indexed elements

The multiplication by indexed elements instructions take all integer elements in each 128-bit vector segment of the first source vector and multiplies them by the indexed element in the corresponding segment of the second source vector.
The products might be destructively added to or subtracted from the corresponding elements of an addend vector.

The second source vector elements are specified using an immediate index that selects the same element position in each 128-bit vector segment. The index range is 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the element size.

### Table C3-160 Multiplication by indexed elements instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLA</td>
<td>Multiply-add to accumulator (indexed)</td>
<td>MLA (indexed) on page C8-3712</td>
</tr>
<tr>
<td>MLS</td>
<td>Multiply-subtract from accumulator (indexed)</td>
<td>MLS (indexed) on page C8-3717</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply (indexed)</td>
<td>MUL (indexed) on page C8-3752</td>
</tr>
<tr>
<td>SMLALB</td>
<td>Signed multiply-add long to accumulator (bottom, indexed)</td>
<td>SMLALB (indexed) on page C8-3958</td>
</tr>
<tr>
<td>SMLALT</td>
<td>Signed multiply-add long to accumulator (top, indexed)</td>
<td>SMLALT (indexed) on page C8-3962</td>
</tr>
<tr>
<td>SMLSLB</td>
<td>Signed multiply-subtract long from accumulator (bottom, indexed)</td>
<td>SMLSLB (indexed) on page C8-3966</td>
</tr>
<tr>
<td>SMLSLT</td>
<td>Signed multiply-subtract long from accumulator (top, indexed)</td>
<td>SMLSLT (indexed) on page C8-3970</td>
</tr>
<tr>
<td>SQDMLALT</td>
<td>Signed saturating doubling multiply-add long to accumulator (top, indexed)</td>
<td>SQDMLALT (indexed) on page C8-4023</td>
</tr>
<tr>
<td>SQDMLSLB</td>
<td>Signed saturating doubling multiply-subtract long from accumulator (bottom, indexed)</td>
<td>SQDMLSLB (indexed) on page C8-4027</td>
</tr>
<tr>
<td>SQDMLSLT</td>
<td>Signed saturating doubling multiply-subtract long from accumulator (top, indexed)</td>
<td>SQDMLSLT (indexed) on page C8-4027</td>
</tr>
<tr>
<td>SQDMULLT</td>
<td>Signed saturating doubling multiply long (top, indexed)</td>
<td>SQDMULLT (indexed) on page C8-4046</td>
</tr>
<tr>
<td>SQRDMLAH</td>
<td>Signed saturating rounding doubling multiply-add high to accumulator (indexed)</td>
<td>SQRDMLAH (indexed) on page C8-4075</td>
</tr>
<tr>
<td>SQRDMLSH</td>
<td>Signed saturating rounding doubling multiply-subtract high from accumulator (indexed)</td>
<td>SQRDMLSH (indexed) on page C8-4080</td>
</tr>
<tr>
<td>SQRDMULH</td>
<td>Signed saturating rounding doubling multiply high (indexed)</td>
<td>SQRDMULH (indexed) on page C8-4085</td>
</tr>
<tr>
<td>UMLALB</td>
<td>Unsigned multiply-add long to accumulator (bottom, indexed)</td>
<td>UMLALB (indexed) on page C8-4367</td>
</tr>
<tr>
<td>UMLALT</td>
<td>Unsigned multiply-add long to accumulator (top, indexed)</td>
<td>UMLALT (indexed) on page C8-4371</td>
</tr>
<tr>
<td>UMLSLB</td>
<td>Unsigned multiply-subtract long from accumulator (bottom, indexed)</td>
<td>UMLSLB (indexed) on page C8-4375</td>
</tr>
<tr>
<td>UMLSLT</td>
<td>Unsigned multiply-subtract long from accumulator (top, indexed)</td>
<td>UMLSLT (indexed) on page C8-4379</td>
</tr>
<tr>
<td>UMULLB</td>
<td>Unsigned multiply long (bottom, indexed)</td>
<td>UMULLB (indexed) on page C8-4389</td>
</tr>
<tr>
<td>UMULLT</td>
<td>Unsigned multiply long (top, indexed)</td>
<td>UMULLT (indexed) on page C8-4393</td>
</tr>
</tbody>
</table>

### C3.8.12 Complex integer arithmetic

Complex integer arithmetic instructions operate on signed integer complex numbers in vectors containing the following interleaved element pairs:

- The even-numbered elements contain the real parts of the complex numbers.
- The odd-numbered elements contain the imaginary parts of the complex numbers.
Uniform complex integer arithmetic

The uniform complex integer arithmetic instructions operate on vectors containing integral complex numbers. The instructions operate on complex numbers that are in polar form.

The **CADD** instructions rotate the complex numbers in the second source vector before adding element pairs to the corresponding elements of the first source vector, in a destructive manner. The rotation direction is 90 degrees or 270 degrees from the positive real axis towards the positive imaginary axis.

The **CMLA** instructions transform the operands to enable multiply-add or multiply-subtract operations on complex numbers by combining two of the instructions. The following transformations are done:

- The complex numbers in the second source vector are rotated by 0 degrees or 180 degrees before multiplying by the duplicated real components of the first source vector.
- The complex numbers in the second source vector are rotated by 90 degrees or 270 degrees before multiplying by the duplicated imaginary components of the first source vector.

The resulting products are added to the corresponding components of the destination and addend vector.

Two **CMLA** instructions can be used, as follows:

- \( CMLA \) \( Zda.S, Zn.S, Zm.S, #A \)
- \( CMLA \) \( Zda.S, Zn.S, Zm.S, #B \)

Some meaningful combinations of \( A \) and \( B \) are:

- \( A=0, B=90 \). The complex number vectors, \( Zn \) and \( Zm \), are multiplied and the products are added to the complex numbers in \( Zda \).
- \( A=0, B=270 \). The complex number conjugates in \( Zn \) are multiplied by the complex numbers in \( Zm \) and the products are added to the complex numbers in \( Zda \).
- \( A=180, B=270 \). The two complex number vectors, \( Zn \) and \( Zm \), are multiplied and the products are subtracted from the complex numbers in \( Zda \).
- \( A=180, B=90 \). The complex number conjugates in \( Zn \) are multiplied by the complex numbers in \( Zm \) and the products are subtracted from the complex numbers in \( Zda \).

The **CMLA** indexed form uses a single complex number in each 128-bit segment of the second source vector as the multiplier for all complex numbers in the corresponding first source vector segment. The complex numbers in the second source vector are specified using an immediate index that selects the same complex number position in each 128-bit vector segment. The index range is 0 to one less than the number of complex numbers per 128-bit segment, encoded in 1 to 2 bits depending on the complex number size.

### Table C3-161 Uniform complex integer arithmetic instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CADD</strong></td>
<td>Complex integer add with rotate</td>
<td>( CADD ) on page C8-3088</td>
</tr>
<tr>
<td><strong>CMLA</strong></td>
<td>Complex integer multiply-add with rotate</td>
<td>( CMLA ) (vectors) on page C8-3114</td>
</tr>
<tr>
<td><strong>SQCADD</strong></td>
<td>Saturating complex integer add with rotate</td>
<td>( SQCADD ) on page C8-3997</td>
</tr>
<tr>
<td><strong>SQRDMLAH</strong></td>
<td>Saturating rounding doubling complex integer multiply-add high with rotate</td>
<td>( SQRDCMLAH ) (vectors) on page C8-4073</td>
</tr>
<tr>
<td><strong>SQRDCMLAH</strong></td>
<td>Saturating rounding doubling complex integer multiply-add high with rotate (indexed)</td>
<td>( SQRDCMLAH ) (indexed) on page C8-4070</td>
</tr>
</tbody>
</table>
Widening complex integer arithmetic

The widening complex integer instructions deinterleave the real and imaginary components of integral complex numbers, and generate complex result components that have a higher numeric precision than the input values. The instructions differ from other complex instructions that process the real and imaginary components of complex numbers and write the complex result components to the destination.

The following instructions are useful when generating the widened components of the result of a complex multiply-add:

- \texttt{SQDMLALBT} - the imaginary results.
- \texttt{SQDMLS LT} - the real results.
- \texttt{SQDMLALB} - the conjugate real results.
- \texttt{SQDMLSLT} - the conjugate imaginary results.

The following instructions are useful when generating the widened components of the result of a complex addition \((X + jY)\) or \((X - jY)\), given complex numbers \(X\) and \(Y\):

- \texttt{SADDLT} - the imaginary results when computing \((X + jY)\) or real values when computing \((X - jY)\).
- \texttt{SSUBLT} - the real results when computing \((X + jY)\) or imaginary values when computing \((X - jY)\).

### Table C3-162 Widening complex integer arithmetic instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SADDLT</td>
<td>Signed add long (bottom + top)</td>
<td>\texttt{SADDLT} on page C8-3893</td>
</tr>
<tr>
<td>SQDMLALBT</td>
<td>Signed saturating doubling multiply-add long to accumulator (bottom × top)</td>
<td>\texttt{SQDMLALBT} on page C8-4021</td>
</tr>
<tr>
<td>SQDMLS LT</td>
<td>Signed saturating doubling multiply-subtract long from accumulator (bottom × top)</td>
<td>\texttt{SQDMLS LT} on page C8-4031</td>
</tr>
<tr>
<td>SSUBLT</td>
<td>Signed subtract long (bottom - top)</td>
<td>\texttt{SSUBLT} on page C8-4153</td>
</tr>
<tr>
<td>SSUBLTB</td>
<td>Signed subtract long (top - bottom)</td>
<td>\texttt{SSUBLTB} on page C8-4157</td>
</tr>
</tbody>
</table>

Complex integer dot product

The complex integer dot product instructions delimit the source vectors into pairs of 8-bit or 16-bit signed integer complex numbers. The complex numbers in the first source vector are multiplied by the corresponding complex numbers in the second source vector. The wide real or wide imaginary part of the product is accumulated into a 32-bit or 64-bit destination vector element that overlaps all four of the elements that form a pair of complex number values in the first source vector.

Each instruction implicitly deinterleaves the real and imaginary components of their complex number inputs, so that the destination vector accumulates four wide real sums or four wide imaginary sums.

The complex numbers in the second source vector are rotated by 0, 90, 180, or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, considered in polar form, by applying the following transformations before the dot product operations:

- If the rotation is \#0, the imaginary parts of the complex numbers in the second source vector are negated. The destination vector accumulates the real parts of a complex dot product.
- If the rotation is \#90, the real and imaginary parts of the complex numbers the second source vector are swapped. The destination vector accumulates the imaginary parts of a complex dot product.
- If the rotation is \#180, there is no transformation. The destination vector accumulates the real parts of a complex conjugate dot product.
- If the rotation is \#270, the real parts of the complex numbers in the second source vector are negated and then swapped with the imaginary parts. The destination vector accumulates the imaginary parts of a complex conjugate dot product.
The indexed form of the instruction selects a single complex number pair within each 128-bit segment of the second source vector to multiply with all complex number pairs within the corresponding 128-bit segment of the first source vector. The complex number pairs within the second source vector are specified using an immediate index which selects the same complex number pair position within each 128-bit vector segment. The index range is from 0 to one less than the number of complex number pairs per 128-bit segment, encoded in 1 or 2 bits depending on the size of the complex number pair.

Each complex number is represented in a vector register as an even and odd pair of elements. The real part is the even-numbered element and the imaginary part is the odd-numbered element.

C3.8.13 Floating-point extra conversions

The floating-point extra conversion instructions convert to and from fully packed vectors of narrower floating-point elements.

The FCVTLT instruction converts the top or odd-numbered narrow floating-point vector elements to wider elements of the next higher precision. The conversion is similar to what is done by the widening integer instructions.

The FCVTNT and FCVXNT instructions convert wider floating-point vector elements to the top or odd-numbered narrower elements of the next lower precision. The conversion is similar to what is done by the narrowing integer instructions.

The FCVTXNT and FCVTX instructions convert from double-precision to fully packed half-precision in two narrowing steps, double-precision to single-precision and then single-precision to half-precision. The two-step conversion is done without an intermediate rounding error by using von Neumann rounding, which rounds an inexact mantissa to an odd value.

The existing SVE FCVT instructions implement the corresponding widening and narrowing conversions on the bottom or even-numbered half-width elements.

C3.8.14 Floating-point widening multiply-accumulate

The floating-point widening multiply-accumulate instructions multiply the even-numbered or odd-numbered half-precision elements of the two source vectors and then destructively add or subtract the single-precision intermediate products. Intermediate rounding is not done. The result is placed into the overlapping single-precision elements of the addend vector.

The instructions implicitly convert the half-precision inputs to single-precision and can be used to mitigate the impact of round-off errors when accumulating half-precision floating-point values over many iterations.
The instructions are unpredicated and preserve the multiplier and multiplicand source vectors.

**Table C3-165 Floating-point widening multiply-accumulate instructions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMLALB</td>
<td>Floating-point fused multiply-add long to accumulator (bottom)</td>
<td>FMLALB (vectors) on page C8-3299</td>
</tr>
<tr>
<td>FMLALB</td>
<td>Floating-point fused multiply-add long to accumulator (bottom, indexed)</td>
<td>FMLALB (indexed) on page C8-3297</td>
</tr>
<tr>
<td>FMLALT</td>
<td>Floating-point fused multiply-add long to accumulator (top)</td>
<td>FMLALT (vectors) on page C8-3303</td>
</tr>
<tr>
<td>FMLALT</td>
<td>Floating-point fused multiply-add long to accumulator (top, indexed)</td>
<td>FMLALT (indexed) on page C8-3301</td>
</tr>
<tr>
<td>FMLSLB</td>
<td>Floating-point fused multiply-subtract long from accumulator (bottom)</td>
<td>FMLSLB (vectors) on page C8-3312</td>
</tr>
<tr>
<td>FMLSLB</td>
<td>Floating-point fused multiply-subtract long from accumulator (bottom, indexed)</td>
<td>FMLSLB (indexed) on page C8-3310</td>
</tr>
<tr>
<td>FMLSLT</td>
<td>Floating-point fused multiply-subtract long from accumulator (top)</td>
<td>FMLSLT (vectors) on page C8-3316</td>
</tr>
<tr>
<td>FMLSLT</td>
<td>Floating-point fused multiply-subtract long from accumulator (top, indexed)</td>
<td>FMLSLT (indexed) on page C8-3314</td>
</tr>
</tbody>
</table>

**C3.8.15 Floating-point integer binary logarithm**

The floating-point integer binary logarithm instruction returns the signed integer base 2 logarithm of each floating-point input element |x| after normalization.

The instruction produces the unbiased exponent of x used in the representation of the floating-point value. For positive x, \( x = \text{significand} \times 2^{\text{exponent}} \).

The integer results are placed in elements of the destination vector, which have the same width as the floating-point input elements:

- If x is normal, the result is the base 2 logarithm of x.
- If x is subnormal, the result corresponds to the normalized representation.
- If x is infinite, the result is \( 2^{(\text{esize}-1) - 1} \).
- If x is ±0.0 or NaN, the result is \( -2^{(\text{esize}-1)} \).

Inactive elements in the destination vector register are not modified.

**Table C3-166 Floating-point integer binary logarithm instruction**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOGB</td>
<td>Floating-point base 2 logarithm as integer</td>
<td>FLOGB on page C8-3260</td>
</tr>
</tbody>
</table>

**C3.8.16 Cross-lane match detect**

This section includes instructions that detect or count matching elements within another vector, or within a 128-bit vector segment.
Vector Histogram Count

The vector histogram count instructions create vector histograms.

- **HISTSEG** compares each 8-bit byte element in the first source vector with all of the elements in the corresponding 128-bit segment of the second source vector. The instruction counts the matching elements and places the result in the corresponding destination vector element. The instruction is unpredicated.

- **HISTCNT** compares each active 32-bit or 64-bit element in the first source vector with all elements in the second source vector that have an element number less than or equal to the *Active element* in the first source vector. The number of matching elements is counted and the result is placed in the corresponding destination vector element. *Inactive elements* in the destination vector are set to zero. *Inactive elements* in the second source vector do not cause a match.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>HISTCNT</td>
<td>Count matching elements in vector</td>
<td>HISTCNT on page C8-3377</td>
</tr>
<tr>
<td>HISTSEG</td>
<td>Count matching elements in vector segments</td>
<td>HISTSEG on page C8-3379</td>
</tr>
</tbody>
</table>

Character match

The character match instructions can be used to scan each 128-bit segment of the second source vector for an 8-bit or 16-bit character string from the first source vector.

The **MATCH** and **NMATCH** instructions compare each active 8-bit or 16-bit character in the first source vector with all of the characters in the corresponding 128-bit segment of the second source vector. When the first source character matches any (MATCH) or does not match any (NMATCH) character in the second segment, a true value is placed in the corresponding destination predicate element, otherwise a false value is placed in the destination predicate element. *Inactive elements* in the destination predicate register are set to zero.

The instruction sets the First (N), None (Z), !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATCH</td>
<td>Detect any matching elements, setting the condition flag</td>
<td>MATCH on page C8-3710</td>
</tr>
<tr>
<td>NMATCH</td>
<td>Detect no matching elements, setting the condition flags</td>
<td>NMATCH on page C8-3767</td>
</tr>
</tbody>
</table>

Contiguous conflict detection

The contiguous conflict detection instructions check two addresses for a conflict or overlap between address ranges that could result in a loop-carried dependency through memory. The address range has the form [addr, addr+VL+8], where VL is the accessible vector length in bits. A conflict can occur when contiguous load and store instructions use these addresses within the same loop iteration.
The instructions generate a predicate with elements that are true when the addresses cannot conflict within the same iteration, and false thereafter. The instructions set the First (N), None (Z), !Last (C) condition flags based on the predicate result, and the V flag is set to zero.

The bit permutation instructions are optional. The bit permutation instructions are configured by the ID_AA64ZFR0_EL1.BitPerm bit. The instructions can be used to scatter, gather, or separate a set of bits within each first source vector element under the control of a bit mask or sieve in the corresponding second source vector elements. The instructions are unpredicated.

The BDEP instruction scatters the lowest-numbered contiguous bits within each first source vector element to the bit positions that are indicated by nonzero bits in the corresponding mask element of the second source vector. The order of the bits is preserved. The bits corresponding to a zero mask bit are set to zero.

The BEXT instruction gathers bits in each first source vector element from the bit positions that are indicated by nonzero bits in the corresponding mask element of the second source vector. The bits are gathered to the lowest-numbered contiguous bits of the corresponding destination element, preserving their order. The remaining higher-numbered bits are set to zero.

The BGRP instruction selects bits from each first source vector element and groups them into the corresponding destination element, using a corresponding mask element in the second source vector, as follows:

- The nonzero bits in the mask element select the bit positions from the corresponding first source vector element. The selected bits are gathered into the lowest-numbered contiguous bits of the destination element.
- The zero bits in the mask element select the bit positions from the corresponding first source vector element. The selected bits are gathered into the highest-numbered contiguous bits of the destination element.

The bit order within each group is preserved.

The polynomial arithmetic instructions support polynomial arithmetic over [0, 1], where exclusive-OR takes the place of addition. The instructions can be used in applications such as CRC calculations, AES-GCM, elliptic curve cryptography, Diffie-Hellman key exchange, and others.

The PMUL and widening PMULL instructions perform a polynomial multiplication over [0, 1]. The PMULL instructions read the source operands from either the even-numbered (bottom) or odd-numbered (top) narrow elements. Each double-width result is placed in the destination elements that overlap the narrow source elements.

### Table C3-169 Contiguous conflict detection instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHILERW</td>
<td>While free of read-after-write conflicts</td>
<td>WHILERW on page C8-4529</td>
</tr>
<tr>
<td>WHILEWR</td>
<td>While free of write-after-read/write conflicts</td>
<td>WHILEWR on page C8-4531</td>
</tr>
</tbody>
</table>

### C3.8.17 Bit permutation

The bit permutation instructions are optional. The bit permutation instructions are configured by the ID_AA64ZFR0_EL1.BitPerm bit. The instructions can be used to scatter, gather, or separate a set of bits within each first source vector element under the control of a bit mask or sieve in the corresponding second source vector elements. The instructions are unpredicated.

The BDEP instruction scatters the lowest-numbered contiguous bits within each first source vector element to the bit positions that are indicated by nonzero bits in the corresponding mask element of the second source vector. The order of the bits is preserved. The bits corresponding to a zero mask bit are set to zero.

The BEXT instruction gathers bits in each first source vector element from the bit positions that are indicated by nonzero bits in the corresponding mask element of the second source vector. The bits are gathered to the lowest-numbered contiguous bits of the corresponding destination element, preserving their order. The remaining higher-numbered bits are set to zero.

The BGRP instruction selects bits from each first source vector element and groups them into the corresponding destination element, using a corresponding mask element in the second source vector, as follows:

- The nonzero bits in the mask element select the bit positions from the corresponding first source vector element. The selected bits are gathered into the lowest-numbered contiguous bits of the destination element.
- The zero bits in the mask element select the bit positions from the corresponding first source vector element. The selected bits are gathered into the highest-numbered contiguous bits of the destination element.

The bit order within each group is preserved.

### Table C3-170 Bit permutation instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDEP</td>
<td>Scatter lower bits into positions selected by bitmask</td>
<td>BDEP on page C8-3034</td>
</tr>
<tr>
<td>BEXT</td>
<td>Gather lower bits from positions selected by bitmask</td>
<td>BEXT on page C8-3036</td>
</tr>
<tr>
<td>BGRP</td>
<td>Group bits to right or left as selected by bitmask</td>
<td>BGRP on page C8-3055</td>
</tr>
</tbody>
</table>

### C3.8.18 Polynomial arithmetic

The polynomial arithmetic instructions support polynomial arithmetic over [0, 1], where exclusive-OR takes the place of addition. The instructions can be used in applications such as CRC calculations, AES-GCM, elliptic curve cryptography, Diffie-Hellman key exchange, and others.

The PMUL and widening PMULL instructions perform a polynomial multiplication over [0, 1]. The PMULL instructions read the source operands from either the even-numbered (bottom) or odd-numbered (top) narrow elements. Each double-width result is placed in the destination elements that overlap the narrow source elements.
The interleaving bitwise exclusive-OR instructions operate on the even-numbered (bottom) elements of the first source vector register and the odd-numbered (top) elements of the second source vector register. The result is either placed in the even-numbered elements of the destination vector, leaving the odd-numbered elements unchanged, or placed in the odd-numbered elements of the destination vector, leaving the even-numbered elements unchanged.

These instructions are unpredicated.

### Table C3-171 Polynomial arithmetic instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>EORBT</td>
<td>Interleaving exclusive OR (bottom, top)</td>
<td>EORBT on page C8-3188</td>
</tr>
<tr>
<td>EORTB</td>
<td>Interleaving exclusive OR (top, bottom)</td>
<td>EORTB on page C8-3192</td>
</tr>
<tr>
<td>PMUL</td>
<td>Polynomial multiply vectors (unpredicated)</td>
<td>PMUL on page C8-3797</td>
</tr>
<tr>
<td>PMULLB</td>
<td>Polynomial multiply long (bottom)</td>
<td>PMULLB on page C8-3799</td>
</tr>
<tr>
<td>PMULLT</td>
<td>Polynomial multiply long (top)</td>
<td>PMULLT on page C8-3801</td>
</tr>
</tbody>
</table>

### C3.8.19 Vector concatenation

The vector concatenation instructions have new constructive versions that are introduced in SVE2 that preserve both of the source operands. In the constructive versions of the instruction, only the first source vector register number is encoded, which requires the source vectors to be in consecutively numbered registers (modulo 32).

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXT</td>
<td>Extract vector from pair of vectors</td>
<td>EXT on page C8-3196</td>
</tr>
<tr>
<td>SPLICE</td>
<td>Splice two vectors under predicate control</td>
<td>SPLICE on page C8-3988</td>
</tr>
</tbody>
</table>

### C3.8.20 Extended table lookup permute

The SVE2 extended table lookup instructions, TBL and TBX enable the construction of table lookups or programmable vector permutes where the table consists of two or more vector registers.

Because the index values can select any element in a vector, the instructions are not naturally vector length agnostic.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBL</td>
<td>Programmable table lookup in one or two vector table (zeroing)</td>
<td>TBL on page C8-4298</td>
</tr>
<tr>
<td>TBX</td>
<td>Programmable table lookup in single vector table (zeroing)</td>
<td>TBX on page C8-4300</td>
</tr>
</tbody>
</table>

### C3.8.21 Non-temporal gather/scatter

The non-temporal gather load and scatter store instructions provide a hint to the memory system that the data structure being accessed has a low reuse frequency. The memory system can use the hint to avoid retaining the data or evicting more frequently-used data from the caches.
These instructions support a single addressing mode consisting of 64-bit or 32-bit vector base addresses plus an unscaled 64-bit scalar offset that defaults to the zero register, XZR. Other addressing modes can be constructed using extra instructions.

### Table C3-174 Non-temporal gather load and scatter store instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDNT1B</td>
<td>Gather load non-temporal unsigned bytes</td>
<td>LDNT1B (vector plus scalar) on page C8-3656</td>
</tr>
<tr>
<td>LDNT1D</td>
<td>Gather load non-temporal unsigned doublewords</td>
<td>LDNT1D (vector plus scalar) on page C8-3662</td>
</tr>
<tr>
<td>LDNT1H</td>
<td>Gather load non-temporal unsigned halfwords</td>
<td>LDNT1H (vector plus scalar) on page C8-3668</td>
</tr>
<tr>
<td>LDNT1SB</td>
<td>Gather load non-temporal signed bytes</td>
<td>LDNT1SB on page C8-3670</td>
</tr>
<tr>
<td>LDNT1SH</td>
<td>Gather load non-temporal signed halfwords</td>
<td>LDNT1SH on page C8-3672</td>
</tr>
<tr>
<td>LDNT1SW</td>
<td>Gather load non-temporal signed words</td>
<td>LDNT1SW on page C8-3674</td>
</tr>
<tr>
<td>LDNT1W</td>
<td>Gather load non-temporal unsigned words</td>
<td>LDNT1W (vector plus scalar) on page C8-3680</td>
</tr>
<tr>
<td>STNT1B</td>
<td>Scatter store non-temporal bytes</td>
<td>STNT1B (vector plus scalar) on page C8-4253</td>
</tr>
<tr>
<td>STNT1D</td>
<td>Scatter store non-temporal doublewords</td>
<td>STNT1D (vector plus scalar) on page C8-4259</td>
</tr>
<tr>
<td>STNT1H</td>
<td>Scatter store non-temporal halfwords</td>
<td>STNT1H (vector plus scalar) on page C8-4265</td>
</tr>
<tr>
<td>STNT1W</td>
<td>Scatter store non-temporal words</td>
<td>STNT1W (vector plus scalar) on page C8-4271</td>
</tr>
</tbody>
</table>

### C3.8.22 Cryptography support

Implementation of cryptography acceleration instructions is optional and controlled by the ID_AA64ZFR0_EL1.{SM4, SHA3, AES} bit fields. Implementation of the instructions requires consistency is maintained with the existing Armv8 cryptographic functionality support, as follows:

- If none of the SVE2 cryptographic instructions are implemented, then the Armv8 AES, SHA1, and SHA256 instructions and the Armv8.4 SHA12, SHA3, SM3, and SM4 instructions can be implemented.
- If the SVE2 SHA3 instructions are implemented, then implementation of the Armv8.4 SHA3 instructions is required.
- If the SVE2 SM4 instructions are implemented, then implementation of the Armv8.4 SM4 instructions is required, but implementing any of the following instructions is optional:
  - The Armv8 AES, SHA1, and SHA256 instructions.
  - The Armv8.4 SHA12 and SHA3 instructions.
- If the SVE2 AES instructions are implemented, then implementation of the Armv8 AES instructions is required, but implementing any of the Armv8 SHA256, SHA12, SHA3, SM3, and SM4 instructions is optional.
- If all of the SVE2 cryptographic instructions are implemented, then implementation of the equivalent Armv8 and Armv8.4 instructions is required.

### AES-128 instructions

AES-128 is a 128-bit block cipher that is computed using a combination of linear XOR operations, the use of rotations by fixed values, and a set of 8-bit non-linear substitutions.
The following instructions accelerate a single encryption round:

- The `AES E` instruction reads a 16-byte state array from each 128-bit segment of the first source vector and a round key from the corresponding 128-bit segment of the second source vector. A single round of the `AddRoundKey()`, `SubBytes()`, and `ShiftRows()` transformations, in accordance with the AES standard, is applied to each state array.

- The `AES IMC` instruction reads a 16-byte state array from each 128-bit segment of the source register and performs a single round of the `MixColumns()` transformation on each state array in accordance with the AES standard.

The following instructions accelerate a single decryption round:

- The `AES D` instruction reads a 16-byte state array from each 128-bit segment of the first source vector and a round key from the corresponding 128-bit segment of the second source vector. A single round of the `AddRoundKey()`, `InvSubBytes()`, and `InvShiftRows()` transformations in accordance with the AES standard, is applied to each state array.

- The `AES IMC` instruction reads a 16-byte state array from each 128-bit segment of the source register and performs a single round of the `InvMixColumns()` transformation on each state array in accordance with the AES standard.

Each updated state array is destructively placed in the corresponding segment of the first source vector. The AES instructions are unpredicated.

### SHA-3 instructions

The SHA-3 instructions accelerate the SHA-3 hash algorithm.

The SHA-3 hash is based on a running digest of 1600 bits, arranged as a five by five array of 64-bit values. The instructions map the 25 64-bit values into 25 vector registers, with each 64-bit value occupying the same 64-bit element in each vector. A series of transformations is done on these registers during a round of the SHA-3 hash calculation.

Two or more parallel SHA-3 hash calculations are combined as a SIMD operation, where one calculation operates on the 0th 64-bit element of each vector, and the other calculation operates on the first 64-bit element of each vector. The SIMD operation is useful for the fast parallel hash algorithm recently introduced into the SHA-3 standard that allows a single input stream to be computed using multiple SHA-3 hashes in parallel.

The SHA3 instructions are unpredicated.

### Table C3-175 AES-128 instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>AESD</td>
<td>AES single round decryption</td>
<td>AESD on page C8-2999</td>
</tr>
<tr>
<td>AESE</td>
<td>AES single round encryption</td>
<td>AESE on page C8-3001</td>
</tr>
<tr>
<td>AES IMC</td>
<td>AES inverse mix columns</td>
<td>AES IMC on page C8-3003</td>
</tr>
<tr>
<td>AES MC</td>
<td>AES mix columns</td>
<td>AES MC on page C8-3005</td>
</tr>
</tbody>
</table>

### Table C3-176 SHA-3 instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX1</td>
<td>Bitwise rotate left by 1 and exclusive OR</td>
<td>RAX1 on page C8-3854</td>
</tr>
</tbody>
</table>

See also *Bitwise ternary logical instructions*. 
SM4 instructions

SM4 is the standard Chinese symmetric encryption algorithm which can be accelerated using a similar approach to that used for AES.

SM4 is a 128-bit wide block cipher that is computed using a combination of linear XOR operations, the use of fixed-value rotations, and a set of 8-bit non-linear substitutions.

- The SM4E instruction reads 16 bytes of input data from each 128-bit segment of the first source vector, and four iterations of 32-bit round keys from the corresponding 128-bit segments of the second source vector. Each input data block is encrypted by four rounds in accordance with the SM4 standard, and destructively placed in the corresponding segments of the first source vector.

- The SM4EKEY instruction reads four rounds of 32-bit input key values from each 128-bit segment of the first source vector, and four rounds of 32-bit constants from the corresponding 128-bit segment of the second source vector. The four rounds of output key values are derived in accordance with the SM4 standard, and placed in the corresponding segments of the destination vector.

The SM4 instructions are unpredicated.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM4E</td>
<td>SM4 encryption and decryption</td>
<td>SM4E on page C8-3938</td>
</tr>
<tr>
<td>SM4EKEY</td>
<td>SM4 key updates</td>
<td>SM4EKEY on page C8-3940</td>
</tr>
</tbody>
</table>
Chapter C4
A64 Instruction Set Encoding

This chapter describes the encoding of the A64 instruction set. It contains the following section:

•  *A64 instruction set encoding on page C4-388.*

In this chapter:

•  In the decode tables, an entry of - for a field value means the value of the field does not affect the decoding.
•  In the decode diagrams, a shaded field indicates that the bits in that field are not used in that level of decode.
C4.1 A64 instruction set encoding

The A64 instruction encoding is:

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 09 08</th>
<th>07 06 05 04</th>
<th>03 02 01 00</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>op1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>op0</td>
</tr>
</tbody>
</table>

Table C4-1 Main encoding table for the A64 instruction set

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>Reserved on page C4-388</td>
</tr>
<tr>
<td>0000</td>
<td>SME encodings on page C4-389</td>
</tr>
<tr>
<td>0001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0010</td>
<td>SVE encodings on page C4-399</td>
</tr>
<tr>
<td>0001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100x</td>
<td>Data Processing -- Immediate on page C4-530</td>
</tr>
<tr>
<td>101x</td>
<td>Branches, Exception Generating and System instructions on page C4-534</td>
</tr>
<tr>
<td>x100</td>
<td>Loads and Stores on page C4-547</td>
</tr>
<tr>
<td>x101</td>
<td>Data Processing -- Register on page C4-590</td>
</tr>
<tr>
<td>x111</td>
<td>Data Processing -- Scalar Floating-Point and Advanced SIMD on page C4-601</td>
</tr>
</tbody>
</table>

C4.1.1 Reserved

This section describes the encoding of the Reserved group. The encodings in this section are decoded from A64 instruction set encoding.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>25 24</th>
<th>16 15</th>
<th>14 13 12 11</th>
<th>10 09 08 07</th>
<th>06 05 04 03</th>
<th>02 01 00</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>op1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C4-2 Encoding table for the Reserved group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>000000000</td>
</tr>
<tr>
<td>!= 00</td>
<td>!= 00000000</td>
</tr>
<tr>
<td>! = 00</td>
<td>-</td>
</tr>
</tbody>
</table>
### SME encodings

This section describes the encoding of the SME encodings group. The encodings in this section are decoded from "A64 instruction set encoding on page C4-388."

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x x0xxxx - -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0x x11xxx - x0x</td>
<td><strong>SME Outer Product - 64 bit on page C4-390</strong></td>
</tr>
<tr>
<td>00 x1xxxx - x1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00 x10xxx - x00</td>
<td><strong>SME FP Outer Product - 32 bit on page C4-391</strong></td>
</tr>
<tr>
<td>00 x10xxx - x01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01 x10xxx - xx0</td>
<td><strong>SME Integer Outer Product - 32 bit on page C4-393</strong></td>
</tr>
<tr>
<td>01 x10xxx - xx1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01 x11xxx - x1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10 0xx000 0 0xx</td>
<td><strong>SME Move into Array on page C4-394</strong></td>
</tr>
<tr>
<td>10 0xx000 0 1xx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10 0xx000 1 -</td>
<td><strong>SME Move from Array on page C4-395</strong></td>
</tr>
<tr>
<td>10 0xx001 - -</td>
<td><strong>SME Misc on page C4-396</strong></td>
</tr>
<tr>
<td>10 0xx010 - x0x</td>
<td><strong>SME Add Vector to Array on page C4-396</strong></td>
</tr>
<tr>
<td>10 0xx010 - x1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10 0xx011 - -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10 0xx1xx - -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10 1xxxxx - -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11 - - -</td>
<td><strong>SME Memory on page C4-397</strong></td>
</tr>
</tbody>
</table>
C4.1.3 SME Outer Product - 64 bit

This section describes the encoding of the SME Outer Product - 64 bit group. The encodings in this section are decoded from SME encodings.

Table C4-4 Encoding table for the SME Outer Product - 64 bit group

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SME FP64 outer product</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>SME Int16 outer product</td>
</tr>
</tbody>
</table>

SME FP64 outer product

This section describes the encoding of the SME FP64 outer product instruction class. The encodings in this section are decoded from SME Outer Product - 64 bit.

<table>
<thead>
<tr>
<th>S</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FMOPA (non-widening)</td>
<td>FEAT_SME_F64F64</td>
</tr>
<tr>
<td>1</td>
<td>FMOPS (non-widening)</td>
<td>FEAT_SME_F64F64</td>
</tr>
</tbody>
</table>
SME Int16 outer product

This section describes the encoding of the SME Int16 outer product instruction class. The encodings in this section are decoded from SME Outer Product - 64 bit on page C4-390.

```
[31 30 29 28 27 26 25 24 23 22 21 20 16 15 13 12 | 10 9 | 8 | 5 4 | 3 2 1 0 ]
1 0 1 0 0 0 0 1 1 u1 Zm Pm Pn Zn S 0 ZAda
```

### Decode fields

<table>
<thead>
<tr>
<th>u0</th>
<th>u1</th>
<th>S</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>SMOPA FEAT_SME_I16I64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>SMOPS FEAT_SME_I16I64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0</td>
<td>SUMOPA FEAT_SME_I16I64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td>SUMOPS FEAT_SME_I16I64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td>USMOPA FEAT_SME_I16I64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1</td>
<td>USMOPS FEAT_SME_I16I64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0</td>
<td>UMOPA FEAT_SME_I16I64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>UMOPS FEAT_SME_I16I64</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### C4.1.4 SME FP Outer Product - 32 bit

This section describes the encoding of the SME FP Outer Product - 32 bit group. The encodings in this section are decoded from SME encodings on page C4-389.

```
[31 29 28 27 26 25 24 23 22 21 20 | 16 | 15 13 12 | 10 9 | 4 3 2 1 0 ]
1000000 10 00
```

Table C4-5 Encoding table for the SME FP Outer Product - 32 bit group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
</tr>
<tr>
<td>0 0</td>
<td>SME FP32 outer product on page C4-392</td>
</tr>
<tr>
<td>0 1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 0</td>
<td>SME widening BF16 outer product on page C4-392</td>
</tr>
<tr>
<td>1 1</td>
<td>SME FP16 widening outer product on page C4-392</td>
</tr>
</tbody>
</table>
SME FP32 outer product

This section describes the encoding of the SME FP32 outer product instruction class. The encodings in this section are decoded from *SME FP Outer Product - 32 bit on page C4-391.*

\[
\begin{array}{ccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & Z & m & P & m & P & n & Z & n & S & 0 & 0 & Z & A & d & a \\
\end{array}
\]

Decode fields

<table>
<thead>
<tr>
<th>S</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FMOPA (non-widening)</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>1</td>
<td>FMOPS (non-widening)</td>
<td>FEAT_SME</td>
</tr>
</tbody>
</table>

SME widening BF16 outer product

This section describes the encoding of the SME widening BF16 outer product instruction class. The encodings in this section are decoded from *SME FP Outer Product - 32 bit on page C4-391.*

\[
\begin{array}{ccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & Z & m & P & m & P & n & Z & n & S & 0 & 0 & Z & A & d & a \\
\end{array}
\]

Decode fields

<table>
<thead>
<tr>
<th>S</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BFMOPA</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>1</td>
<td>BFMOPS</td>
<td>FEAT_SME</td>
</tr>
</tbody>
</table>

SME FP16 widening outer product

This section describes the encoding of the SME FP16 widening outer product instruction class. The encodings in this section are decoded from *SME FP Outer Product - 32 bit on page C4-391.*

\[
\begin{array}{ccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & Z & m & P & m & P & n & Z & n & S & 0 & 0 & Z & A & d & a \\
\end{array}
\]

Decode fields

<table>
<thead>
<tr>
<th>S</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FMOPA (widening)</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>1</td>
<td>FMOPS (widening)</td>
<td>FEAT_SME</td>
</tr>
</tbody>
</table>
C4.1.5 SME Integer Outer Product - 32 bit

This section describes the encoding of the SME Integer Outer Product - 32 bit group. The encodings in this section are decoded from SME encodings on page C4-389.

| 31 | 25 24|23 22 21 | | | | | 4 | 3 | 2 | 1 | 0 |
|----|-----|--------|---|---|---|---|---|---|---|---|---|---|
| 1010000 | 10 | | | | | | 0 | | |

Table C4-6 Encoding table for the SME Integer Outer Product - 32 bit group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SME Int8 outer product</td>
</tr>
<tr>
<td>1</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SME Int8 outer product

This section describes the encoding of the SME Int8 outer product instruction class. The encodings in this section are decoded from SME Integer Outer Product - 32 bit.

| 31 30 29 28|27 26 25 24|23 22 21 20| 16 15 |13 12| 10 9 | | 5 4 3 2 1 0 |
|------------|---------|----------|------|-----|-----|---|------|------|------|------|------|------|
| 1 0 1 0 0 0 0 | u0 | 1 0 u1 | Zm | Pm | Pn | Zn | S | 0 0 | ZAda |

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>u0 u1 S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>SMOPA</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>0 0 1</td>
<td>SMOPS</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>0 1 0</td>
<td>SUMOPA</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>0 1 1</td>
<td>SUMOPS</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>1 0 0</td>
<td>USMOPA</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>1 0 1</td>
<td>USMOPS</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>1 1 0</td>
<td>UMOPA</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>1 1 1</td>
<td>UMOPS</td>
<td>FEAT_SME</td>
</tr>
</tbody>
</table>
C4.1.6 SME Move into Array

This section describes the encoding of the SME Move into Array group. The encodings in this section are decoded from SME encodings on page C4-389.

<table>
<thead>
<tr>
<th>31</th>
<th>24 23 22 21</th>
<th>19 18 17 16</th>
<th></th>
<th>5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11000000</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table C4-7 Encoding table for the SME Move into Array group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SME move vector to array</td>
</tr>
<tr>
<td>1</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SME move vector to array

This section describes the encoding of the SME move vector to array instruction class. The encodings in this section are decoded from SME Move into Array.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12</th>
<th>10 9</th>
<th>5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 0 0 0</td>
<td>size</td>
<td>0 0 0 0 Q V Rs Pg Zn 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>size Q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x 1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>00 0</td>
<td>MOVA (vector to tile) - Encoding</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>01 0</td>
<td>MOVA (vector to tile) - Encoding</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>10 0</td>
<td>MOVA (vector to tile) - Encoding</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>10 1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>11 0</td>
<td>MOVA (vector to tile) - Encoding</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>11 1</td>
<td>MOVA (vector to tile) - Encoding</td>
<td>FEAT_SME</td>
</tr>
</tbody>
</table>
C4.1.7 SME Move from Array

This section describes the encoding of the SME Move from Array group. The encodings in this section are decoded from SME encodings on page C4-389.

<table>
<thead>
<tr>
<th>31</th>
<th>24[23 22 21]</th>
<th>19 18 17 16</th>
<th>10 9 8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11000000</td>
<td>000</td>
<td>1</td>
<td></td>
<td>op0 op1</td>
</tr>
</tbody>
</table>

Table C4-8 Encoding table for the SME Move from Array group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

SME move array to vector

This section describes the encoding of the SME move array to vector instruction class. The encodings in this section are decoded from SME Move from Array.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Q</td>
<td>V</td>
<td>Rs</td>
<td>Pg</td>
<td>0</td>
<td>opc</td>
<td>( Zd )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>( 0x )</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>( 00 )</td>
<td>0</td>
<td>MOVA (tile to vector) - Encoding</td>
</tr>
<tr>
<td>( 01 )</td>
<td>0</td>
<td>MOVA (tile to vector) - Encoding</td>
</tr>
<tr>
<td>( 10 )</td>
<td>0</td>
<td>MOVA (tile to vector) - Encoding</td>
</tr>
<tr>
<td>( 10 )</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>( 11 )</td>
<td>0</td>
<td>MOVA (tile to vector) - Encoding</td>
</tr>
<tr>
<td>( 11 )</td>
<td>1</td>
<td>MOVA (tile to vector) - Encoding</td>
</tr>
</tbody>
</table>
C4.1.8 SME Misc

This section describes the encoding of the SME Misc group. The encodings in this section are decoded from SME encodings on page C4-389.

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>19</th>
<th>18</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11000000</td>
<td>op0</td>
<td>001</td>
<td>op1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C4-9 Encoding table for the SME Misc group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td>ZERO</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>00 00000000000</td>
<td>Unallocated. -</td>
<td></td>
</tr>
<tr>
<td>!= 00</td>
<td>- Unallocated. -</td>
<td></td>
</tr>
</tbody>
</table>

C4.1.9 SME Add Vector to Array

This section describes the encoding of the SME Add Vector to Array group. The encodings in this section are decoded from SME encodings on page C4-389.

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11000000</td>
<td>op0</td>
<td>010</td>
<td>op1</td>
<td>0</td>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C4-10 Encoding table for the SME Add Vector to Array group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2</td>
<td></td>
</tr>
<tr>
<td>0 - -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 00 0</td>
<td>SME add vector to array on page C4-397</td>
</tr>
<tr>
<td>1 00 1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 != 00 -</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
SME add vector to array

This section describes the encoding of the SME add vector to array instruction class. The encodings in this section are decoded from "SME Add Vector to Array".

C4.1.10 SME Memory

This section describes the encoding of the SME Memory group. The encodings in this section are decoded from "SME encodings" on page C4-389.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2 op3</td>
<td>op</td>
<td>V</td>
</tr>
<tr>
<td>0xx0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0xx1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0xxx</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>100x 000000 000 0</td>
<td>SMA load array vector (elements) on page C4-398</td>
<td>-</td>
</tr>
<tr>
<td>100x 000000 000 1</td>
<td>SMA store array vector (elements) on page C4-398</td>
<td>-</td>
</tr>
<tr>
<td>100x != 000000 != 000</td>
<td>SMA save and restore array on page C4-399</td>
<td>-</td>
</tr>
<tr>
<td>101x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>110x</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table C4-11 Encoding table for the SME Memory group
### SME load array vector (elements)

This section describes the encoding of the SME load array vector (elements) instruction class. The encodings in this section are decoded from *SME Memory* on page C4-397.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>- - 0</td>
<td>LD1Q FEAT_SME</td>
</tr>
<tr>
<td>1111</td>
<td>- - 0</td>
<td>ST1Q FEAT_SME</td>
</tr>
<tr>
<td>111x</td>
<td>- - 1</td>
<td>Unallocated. -</td>
</tr>
</tbody>
</table>

### SME store array vector (elements)

This section describes the encoding of the SME store array vector (elements) instruction class. The encodings in this section are decoded from *SME Memory* on page C4-397.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>LD1B FEAT_SME</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>LD1H FEAT_SME</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>LD1W FEAT_SME</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>LD1D FEAT_SME</td>
<td></td>
</tr>
</tbody>
</table>
SME save and restore array

This section describes the encoding of the SME save and restore array instruction class. The encodings in this section are decoded from SME Memory on page C4-397.

| [31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 3 0 | 1 1 1 0 0 0 0 0 | 0 0 0 0 0 0 | Rv 0 0 0 | Rn 0 | imm4 |

C4.11 SVE encodings

This section describes the encoding of the SVE encodings group. The encodings in this section are decoded from A64 instruction set encoding on page C4-388.

| [31 29 28|25 24|23 22 21|17 16|15|10 9 | 5 4 3 0 | 0010 | op1 | 000xxx | op2 | x1xxxx | op3 | 000xxx | op4 |

Table C4-12 Encoding table for the SVE encodings group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 0x 0xxx x1xxx</td>
<td>SVE Integer Multiply-Add - Predicated on page C4-412</td>
</tr>
<tr>
<td>000 0x 0xxx 000xxx</td>
<td>SVE Integer Binary Arithmetic - Predicated on page C4-413</td>
</tr>
<tr>
<td>000 0x 0xxx 001xxx</td>
<td>SVE Integer Reduction on page C4-416</td>
</tr>
<tr>
<td>000 0x 0xxx 100xxx</td>
<td>SVE Bitwise Shift - Predicated on page C4-418</td>
</tr>
<tr>
<td>000 0x 0xxx 101xxx</td>
<td>SVE Integer Unary Arithmetic - Predicated on page C4-420</td>
</tr>
<tr>
<td>000 0x 1xxx 000xxx</td>
<td>SVE integer add/subtract vectors (unpredicated) on page C4-403</td>
</tr>
<tr>
<td>000 0x 1xxx 001xxx</td>
<td>SVE Bitwise Logical - Unpredicated on page C4-422</td>
</tr>
<tr>
<td>000 0x 1xxx 0100xx</td>
<td>SVE Index Generation on page C4-423</td>
</tr>
<tr>
<td>000 0x 1xxx 0101xx</td>
<td>SVE Stack Allocation on page C4-423</td>
</tr>
<tr>
<td>000 0x 1xxx 011xxx</td>
<td>SVE2 Integer Multiply - Unpredicated on page C4-425</td>
</tr>
<tr>
<td>000 0x 1xxx 100xxx</td>
<td>SVE Bitwise Shift - Unpredicated on page C4-427</td>
</tr>
<tr>
<td>000 0x 1xxx 1010xx</td>
<td>SVE address generation on page C4-403</td>
</tr>
<tr>
<td>000 0x 1xxx 1011xx</td>
<td>SVE Integer Misc - Unpredicated on page C4-428</td>
</tr>
<tr>
<td>000 0x 1xxx 11xxx</td>
<td>SVE Element Count on page C4-430</td>
</tr>
</tbody>
</table>
Table C4-12 Encoding table for the SVE encodings group (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>1x</td>
<td>00xx</td>
<td>-</td>
<td>-</td>
<td>SVE Bitwise Immediate on page C4-434</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>01xx</td>
<td>-</td>
<td>-</td>
<td>SVE Integer Wide Immediate - Predicated on page C4-435</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxx</td>
<td>001000</td>
<td>-</td>
<td>DUP (indexed)</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxx</td>
<td>001001</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxx</td>
<td>00101x</td>
<td>-</td>
<td>SVE table lookup (three sources) on page C4-403</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxx</td>
<td>001100</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxx</td>
<td>001110</td>
<td>-</td>
<td>SVE Permute Vector - Unpredicated on page C4-436</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxx</td>
<td>010xxx</td>
<td>-</td>
<td>SVE Permute Predicate on page C4-437</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxx</td>
<td>011xxx</td>
<td>-</td>
<td>SVE permute vector elements on page C4-404</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxx</td>
<td>10xxx</td>
<td>-</td>
<td>SVE Permute Vector - Predicated on page C4-438</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxx</td>
<td>11xxx</td>
<td>-</td>
<td>SEL (vectors)</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxx</td>
<td>000xxx</td>
<td>-</td>
<td>SVE Permute Vector - Extract on page C4-442</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxx</td>
<td>000xxx</td>
<td>-</td>
<td>SVE Permute Vector - Segments on page C4-442</td>
</tr>
<tr>
<td>001</td>
<td>0x</td>
<td>0xxx</td>
<td>-</td>
<td>-</td>
<td>SVE Integer Compare - Vectors on page C4-443</td>
</tr>
<tr>
<td>001</td>
<td>0x</td>
<td>1xxx</td>
<td>-</td>
<td>-</td>
<td>SVE integer compare with unsigned immediate on page C4-405</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>0xxx</td>
<td>x0xxxx</td>
<td>-</td>
<td>SVE integer compare with signed immediate on page C4-405</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>00xxx</td>
<td>01xxx</td>
<td>-</td>
<td>SVE predicate logical operations on page C4-406</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>00xxx</td>
<td>11xxx</td>
<td>-</td>
<td>SVE Propagate Break on page C4-445</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>01xx</td>
<td>01xxx</td>
<td>-</td>
<td>SVE Partition Break on page C4-446</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>01xx</td>
<td>11xxx</td>
<td>-</td>
<td>SVE Predicate Misc on page C4-447</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>1xxx</td>
<td>00xxx</td>
<td>-</td>
<td>SVE Integer Compare - Scalars on page C4-451</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>1xxx</td>
<td>01xxx</td>
<td>0</td>
<td>SVE broadcast predicate element on page C4-407</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>1xxx</td>
<td>01xxx</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>1xxx</td>
<td>11xxx</td>
<td>-</td>
<td>SVE Integer Wide Immediate - Unpredicated on page C4-453</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>100xx</td>
<td>10xxx</td>
<td>-</td>
<td>SVE Predicate Count on page C4-456</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>101xx</td>
<td>1000xx</td>
<td>-</td>
<td>SVE Inc/Dec by Predicate Count on page C4-457</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>101xx</td>
<td>1001xx</td>
<td>-</td>
<td>SVE Write FFR on page C4-459</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>101xx</td>
<td>101xxx</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>11xxx</td>
<td>10xxx</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>010</td>
<td>0x</td>
<td>0xxx</td>
<td>0xxxx</td>
<td>-</td>
<td>SVE Integer Multiply-Add - Unpredicated on page C4-461</td>
</tr>
</tbody>
</table>
Table C4-12 Encoding table for the SVE encodings group (continued)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>010 0x 0xxx 10xxx -</td>
<td>SVE2 Integer - Predicated on page C4-464</td>
</tr>
<tr>
<td>010 0x 0xxx 11000x -</td>
<td>SVE integer clamp on page C4-407</td>
</tr>
<tr>
<td>010 0x 0xxx 11001x -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>010 0x 0xxx 1101xx -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>010 0x 0xxx 111xx -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>010 1x 1xxx - -</td>
<td>SVE Multiply - Indexed on page C4-468</td>
</tr>
<tr>
<td>010 1x 0xxx 0xxxxx -</td>
<td>SVE2 Widening Integer Arithmetic on page C4-475</td>
</tr>
<tr>
<td>010 1x 0xxx 10xxxx -</td>
<td>SVE Misc on page C4-477</td>
</tr>
<tr>
<td>010 1x 0xxx 11xxxx -</td>
<td>SVE2 Accumulate on page C4-479</td>
</tr>
<tr>
<td>010 1x 1xxxx 0xxxxx -</td>
<td>SVE2 Narrowing on page C4-483</td>
</tr>
<tr>
<td>010 1x 1xxxx 100xxx -</td>
<td>SVE2 character match on page C4-407</td>
</tr>
<tr>
<td>010 1x 1xxxx 101xxx -</td>
<td>SVE2 Histogram Computation - Segment on page C4-485</td>
</tr>
<tr>
<td>010 1x 1xxxx 110xxx -</td>
<td>HISTCNT</td>
</tr>
<tr>
<td>010 1x 1xxxx 111xxx -</td>
<td>SVE2 Crypto Extensions on page C4-485</td>
</tr>
<tr>
<td>011 0x 0xxx 0xxxxx -</td>
<td>FCMLA (vectors)</td>
</tr>
<tr>
<td>011 0x 001xx 1xxxxx -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011 0x 00000 100xxx -</td>
<td>FCADD</td>
</tr>
<tr>
<td>011 0x 00000 101xxx -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011 0x 00001 1xxxxx -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011 0x 0010x 100xxx -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011 0x 0010x 101xxx -</td>
<td>SVE floating-point convert precision odd elements on page C4-408</td>
</tr>
<tr>
<td>011 0x 0010x 110xxx -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011 0x 010xx 100xxx -</td>
<td>SVE2 floating-point pairwise operations on page C4-408</td>
</tr>
<tr>
<td>011 0x 010xx 101xxx -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011 0x 010xx 110xxx -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011 0x 011xx 1xxxxx -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011 0x 1xxxx 0000xx -</td>
<td>SVE floating-point multiply-add (indexed) on page C4-409</td>
</tr>
<tr>
<td>011 0x 1xxxx 001xx -</td>
<td>SVE floating-point complex multiply-add (indexed) on page C4-409</td>
</tr>
<tr>
<td>011 0x 1xxxx 0010x -</td>
<td>SVE floating-point multiply (indexed) on page C4-409</td>
</tr>
<tr>
<td>011 0x 1xxxx 00101 -</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
Table C4-12 Encoding table for the SVE encodings group (continued)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
</tr>
<tr>
<td>011</td>
<td>1x</td>
</tr>
<tr>
<td>011</td>
<td>1x</td>
</tr>
<tr>
<td>011</td>
<td>1x</td>
</tr>
<tr>
<td>011</td>
<td>1x</td>
</tr>
<tr>
<td>011</td>
<td>1x</td>
</tr>
<tr>
<td>011</td>
<td>1x</td>
</tr>
<tr>
<td>011</td>
<td>1x</td>
</tr>
<tr>
<td>011</td>
<td>1x</td>
</tr>
<tr>
<td>011</td>
<td>1x</td>
</tr>
<tr>
<td>011</td>
<td>1x</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td>101</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>-</td>
</tr>
<tr>
<td>111</td>
<td>-</td>
</tr>
<tr>
<td>111</td>
<td>-</td>
</tr>
<tr>
<td>111</td>
<td>-</td>
</tr>
<tr>
<td>111</td>
<td>-</td>
</tr>
<tr>
<td>111</td>
<td>-</td>
</tr>
</tbody>
</table>
SVE integer add/subtract vectors (unpredicated)

This section describes the encoding of the SVE integer add/subtract vectors (unpredicated) instruction class. The encodings in this section are decoded from SVE encodings on page C4-399.

\[
\begin{array}{cccccccccc}
\text{opc} & \text{ADD (vectors, unpredicated)} & 000 \\
\text{opc} & \text{SUB (vectors, unpredicated)} & 001 \\
\text{opc} & \text{Unallocated} & 01x \\
\text{opc} & \text{SQADD (vectors, unpredicated)} & 100 \\
\text{opc} & \text{UQADD (vectors, unpredicated)} & 101 \\
\text{opc} & \text{SQSUB (vectors, unpredicated)} & 110 \\
\text{opc} & \text{UQSUB (vectors, unpredicated)} & 111 \\
\end{array}
\]

SVE address generation

This section describes the encoding of the SVE address generation instruction class. The encodings in this section are decoded from SVE encodings on page C4-399.

\[
\begin{array}{ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
SVE permute vector elements

This section describes the encoding of the SVE permute vector elements instruction class. The encodings in this section are decoded from SVE encodings on page C4-399.

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 16 15 14 13 12 | 11 10 9 | 5 4 | 0 |
| 0 0 0 0 0 1 0 1 | size 1 | Zm 0 0 1 0 1 op | Zn | Zd |

### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TBL</td>
</tr>
<tr>
<td>1</td>
<td>TBX</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ZIP1, ZIP2 (vectors) - Encoding</td>
</tr>
<tr>
<td>001</td>
<td>ZIP1, ZIP2 (vectors) - Encoding</td>
</tr>
<tr>
<td>010</td>
<td>UZP1, UZP2 (vectors) - Encoding</td>
</tr>
<tr>
<td>011</td>
<td>UZP1, UZP2 (vectors) - Encoding</td>
</tr>
<tr>
<td>100</td>
<td>TRN1, TRN2 (vectors) - Encoding</td>
</tr>
<tr>
<td>101</td>
<td>TRN1, TRN2 (vectors) - Encoding</td>
</tr>
<tr>
<td>11x</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
SVE integer compare with unsigned immediate

This section describes the encoding of the SVE integer compare with unsigned immediate instruction class. The encodings in this section are decoded from SVE encodings on page C4-399.

```
[31 30 29 28 27 26 25 24 23 22 21 20 | 14 13 12 | 10 9 | 5 4 3 0]
 0 0 1 0 0 1 0 0 size 1 imm7 | lt | Pg | Zn ne | Pd
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>lt ne</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>CMP&lt;cc&gt; (immediate) - Encoding</td>
</tr>
<tr>
<td>0 1</td>
<td>CMP&lt;cc&gt; (immediate) - Encoding</td>
</tr>
<tr>
<td>1 0</td>
<td>CMP&lt;cc&gt; (immediate) - Encoding</td>
</tr>
<tr>
<td>1 1</td>
<td>CMP&lt;cc&gt; (immediate) - Encoding</td>
</tr>
</tbody>
</table>

SVE integer compare with signed immediate

This section describes the encoding of the SVE integer compare with signed immediate instruction class. The encodings in this section are decoded from SVE encodings on page C4-399.

```
[31 30 29 28 27 26 25 24 23 22 21 20 | 16 | 15 14 13 12 | 10 9 | 5 4 3 0]
 0 0 1 0 0 1 0 0 size 0 imm5 op 0 o2 Pg | Zn ne | Pd
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op o2 ne</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>CMP&lt;cc&gt; (immediate) - Encoding</td>
</tr>
<tr>
<td>0 0 1</td>
<td>CMP&lt;cc&gt; (immediate) - Encoding</td>
</tr>
<tr>
<td>0 1 0</td>
<td>CMP&lt;cc&gt; (immediate) - Encoding</td>
</tr>
<tr>
<td>0 1 1</td>
<td>CMP&lt;cc&gt; (immediate) - Encoding</td>
</tr>
<tr>
<td>1 0 0</td>
<td>CMP&lt;cc&gt; (immediate) - Encoding</td>
</tr>
<tr>
<td>1 0 1</td>
<td>CMP&lt;cc&gt; (immediate) - Encoding</td>
</tr>
<tr>
<td>1 1</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
SVE predicate logical operations

This section describes the encoding of the SVE predicate logical operations instruction class. The encodings in this section are decoded from *SVE encodings on page C4-399*.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16 15 14 13</th>
<th>10 9 8</th>
<th>5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1</td>
<td>op</td>
<td>S</td>
<td>0 0</td>
<td>Pm</td>
<td>0 1</td>
<td>Pg</td>
<td>o2</td>
</tr>
</tbody>
</table>

**Decode fields**

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>o3</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>AND (predicates)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>BIC (predicates)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>EOR (predicates)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>SEL (predicates)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>ANDS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>BICS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>EORS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>Unallocated.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>ORR (predicates)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>ORN (predicates)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>NOR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>NAND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>ORRS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>ORNS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>NORS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>NANDS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SVE broadcast predicate element

This section describes the encoding of the SVE broadcast predicate element instruction class. The encodings in this section are decoded from SVE encodings on page C4-399.

```
[31 30 29 28][27 26 25 24][23 22 21 20] 18 17 16|15 14 13 | 10 9 8 | 5 4 3 0 |
0 0 1 0 0 1 0 1 |1 1 | tszh  Rv 0 1 | Pn S | Pm 0 | Pd
```

tszh

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>PSEL</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

SVE integer clamp

This section describes the encoding of the SVE integer clamp instruction class. The encodings in this section are decoded from SVE encodings on page C4-399.

```
[31 30 29 28][27 26 25 24][23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 0 0 0 1 0 0 | size 0 | Zm 1 1 0 0 0 | U | Zn Zd
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SCLAMP</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>1</td>
<td>UCLAMP</td>
<td>FEAT_SME</td>
</tr>
</tbody>
</table>

SVE2 character match

This section describes the encoding of the SVE2 character match instruction class. The encodings in this section are decoded from SVE encodings on page C4-399.

```
[31 30 29 28][27 26 25 24][23 22 21 20] 16|15 14 13 12|10 9 | 5 4 | 3 0 |
0 1 0 0 0 1 0 1 | size 1 | Zm 1 0 0 | Pg Zn | op Pd
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>MATCH</td>
</tr>
<tr>
<td>1</td>
<td>NMATCH</td>
</tr>
</tbody>
</table>
SVE floating-point convert precision odd elements

This section describes the encoding of the SVE floating-point convert precision odd elements instruction class. The encodings in this section are decoded from *SVE encodings on page C4-399.*

\[
\begin{array}{cccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & opc & 0 & 0 & 1 & 0 & opc2 & 1 & 0 & 1 & \hline
\end{array}
\]

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>opc2</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>11</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>FCVTXNT</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>FCVTNT - Encoding</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>FCVTLT - Encoding</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>BFCVTNT</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>11</td>
<td>0x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>FCVTNT - Encoding</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>FCVTLT - Encoding</td>
<td>-</td>
</tr>
</tbody>
</table>

SVE2 floating-point pairwise operations

This section describes the encoding of the SVE2 floating-point pairwise operations instruction class. The encodings in this section are decoded from *SVE encodings on page C4-399.*

\[
\begin{array}{cccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & opc & 1 & 0 & 1 & 0 & Pg & 1 & 0 & 1 & \hline
\end{array}
\]

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>FADDP</td>
</tr>
<tr>
<td>001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100</td>
<td>FMAXXNMP</td>
</tr>
<tr>
<td>101</td>
<td>FMINNMP</td>
</tr>
<tr>
<td>110</td>
<td>FMAXP</td>
</tr>
<tr>
<td>111</td>
<td>FMINP</td>
</tr>
</tbody>
</table>
SVE floating-point multiply-add (indexed)

This section describes the encoding of the SVE floating-point multiply-add (indexed) instruction class. The encodings in this section are decoded from SVE encodings on page C4-399.

\[
\begin{array}{cccccc}
& \text{size} & \text{o2} & \text{op} & \text{Instruction page} \\
0 & 1 & 0 & 0 & 0 & \text{Unallocated.} \\
0 & 0 & 0 & 0 & \text{FMLA (indexed) - Encoding} \\
0 & 0 & 1 & \text{FMLS (indexed) - Encoding} \\
10 & 0 & 0 & \text{FMLA (indexed) - Encoding} \\
10 & 0 & 1 & \text{FMLS (indexed) - Encoding} \\
11 & 0 & 0 & \text{FMLA (indexed) - Encoding} \\
11 & 0 & 1 & \text{FMLS (indexed) - Encoding} \\
\end{array}
\]

SVE floating-point complex multiply-add (indexed)

This section describes the encoding of the SVE floating-point complex multiply-add (indexed) instruction class. The encodings in this section are decoded from SVE encodings on page C4-399.

\[
\begin{array}{cccccccccc}
& \text{size} & \text{o2} & \text{op} & \text{Instruction page} \\
0 & 1 & 0 & 0 & 0 & \text{Unallocated.} \\
0 & 0 & 0 & 0 & \text{FCMLA (indexed) - Encoding} \\
0 & 0 & 1 & \text{FCMLA (indexed) - Encoding} \\
\end{array}
\]

SVE floating-point multiply (indexed)

This section describes the encoding of the SVE floating-point multiply (indexed) instruction class. The encodings in this section are decoded from SVE encodings on page C4-399.
SVE floating point matrix multiply accumulate

This section describes the encoding of the SVE floating point matrix multiply accumulate instruction class. The encodings in this section are decoded from SVE encodings on page C4-399.

SVE floating-point compare vectors

This section describes the encoding of the SVE floating-point compare vectors instruction class. The encodings in this section are decoded from SVE encodings on page C4-399.
SVE floating-point arithmetic (unpredicated)

This section describes the encoding of the SVE floating-point arithmetic (unpredicated) instruction class. The encodings in this section are decoded from SVE encodings on page C4-399.

SVE floating-point recursive reduction

This section describes the encoding of the SVE floating-point recursive reduction instruction class. The encodings in this section are decoded from SVE encodings on page C4-399.
C4.1.12 SVE Integer Multiply-Add - Predicated

This section describes the encoding of the SVE Integer Multiply-Add - Predicated group. The encodings in this section are decoded from SVE encodings on page C4-399.

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[19 18]</th>
<th>[16 15 14 13 12]</th>
<th>[10 9]</th>
<th>[5 4]</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 1 0 1</td>
<td>size 0 0 0</td>
<td>opc 0 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Vd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C4-13 Encoding table for the SVE Integer Multiply-Add - Predicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>Instruction page</td>
</tr>
<tr>
<td>000</td>
<td>FADDV</td>
</tr>
<tr>
<td>001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100</td>
<td>FMAXNMV</td>
</tr>
<tr>
<td>101</td>
<td>FMINNMV</td>
</tr>
<tr>
<td>110</td>
<td>FMAXV</td>
</tr>
<tr>
<td>111</td>
<td>FMINV</td>
</tr>
</tbody>
</table>

SVE integer multiply-accumulate writing addend (predicated)

This section describes the encoding of the SVE integer multiply-accumulate writing addend (predicated) instruction class. The encodings in this section are decoded from SVE Integer Multiply-Add - Predicated.
SVE integer multiply-add writing multiplicand (predicated)

This section describes the encoding of the SVE integer multiply-add writing multiplicand (predicated) instruction class. The encodings in this section are decoded from SVE Integer Multiply-Add - Predicated on page C4-412.

C4.1.13   SVE Integer Binary Arithmetic - Predicated

This section describes the encoding of the SVE Integer Binary Arithmetic - Predicated group. The encodings in this section are decoded from SVE encodings on page C4-399.

Table C4-14 Encoding table for the SVE Integer Binary Arithmetic - Predicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>00x</td>
<td>SVE integer add/subtract vectors (predicated) on page C4-414</td>
</tr>
<tr>
<td>01x</td>
<td>SVE integer min/max/difference (predicated) on page C4-414</td>
</tr>
<tr>
<td>100</td>
<td>SVE integer multiply vectors (predicated) on page C4-414</td>
</tr>
<tr>
<td>101</td>
<td>SVE integer divide vectors (predicated) on page C4-415</td>
</tr>
<tr>
<td>11x</td>
<td>SVE bitwise logical operations (predicated) on page C4-415</td>
</tr>
</tbody>
</table>
SVE integer add/subtract vectors (predicated)

This section describes the encoding of the SVE integer add/subtract vectors (predicated) instruction class. The encodings in this section are decoded from *SVE Integer Binary Arithmetic - Predicated* on page C4-413.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0</td>
<td>0 0 0</td>
<td>opc</td>
<td>0 0</td>
</tr>
</tbody>
</table>
```

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ADD (vectors, predicated)</td>
</tr>
<tr>
<td>001</td>
<td>SUB (vectors, predicated)</td>
</tr>
<tr>
<td>010</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011</td>
<td>SUBR (vectors)</td>
</tr>
<tr>
<td>1xx</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SVE integer min/max/difference (predicated)

This section describes the encoding of the SVE integer min/max/difference (predicated) instruction class. The encodings in this section are decoded from *SVE Integer Binary Arithmetic - Predicated* on page C4-413.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0</td>
<td>0 1 0</td>
<td>opc</td>
<td>U 0 0</td>
</tr>
</tbody>
</table>
```

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>U</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>SMAX (vectors)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>UMAX (vectors)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>SMIN (vectors)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>UMIN (vectors)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>SABD</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>UABD</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SVE integer multiply vectors (predicated)

This section describes the encoding of the SVE integer multiply vectors (predicated) instruction class. The encodings in this section are decoded from *SVE Integer Binary Arithmetic - Predicated* on page C4-413.
This section describes the encoding of the SVE integer divide vectors (predicated) instruction class. The encodings in this section are decoded from *SVE Integer Binary Arithmetic - Predicated* on page C4-413.

![Binary representation of a SVE instruction](image)

### SVE Integer Divide Vectors (Predicated)

The binary representation of the SVE integer divide vectors (predicated) instruction is shown above. The instruction's fields are decoded as follows:

- **H** (High): Determines the function (MUL, SMULH, or UMULH).
- **U** (Upper): Determines the predication.

### SVE Bitwise Logical Operations (Predicated)

This section describes the encoding of the SVE bitwise logical operations (predicated) instruction class. The encodings in this section are decoded from *SVE Integer Binary Arithmetic - Predicated* on page C4-413.

The binary representation of the SVE bitwise logical operations (predicated) instruction is shown above. The instruction's fields are decoded as follows:

- **R** (Register): Determines the function (SDIV, UDIV, SDIVR, or UDIVR).
- **U** (Upper): Determines the predication.
C4.1.14 SVE Integer Reduction

This section describes the encoding of the SVE Integer Reduction group. The encodings in this section are decoded from SVE encodings on page C4-399.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 24 23 21 20 18 17 16 15 13 12</th>
<th>10 9 5 4 0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0 size 0 1 1 opc 0 0 0 Pg Zm Zdn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ORR (vectors, predicated)</td>
</tr>
<tr>
<td>001</td>
<td>EOR (vectors, predicated)</td>
</tr>
<tr>
<td>010</td>
<td>AND (vectors, predicated)</td>
</tr>
<tr>
<td>011</td>
<td>BIC (vectors, predicated)</td>
</tr>
<tr>
<td>1xx</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

---

**SVE Integer add reduction (predicated)**

This section describes the encoding of the SVE integer add reduction (predicated) instruction class. The encodings in this section are decoded from *SVE Integer Reduction.*
SVE integer min/max reduction (predicated)

This section describes the encoding of the SVE integer min/max reduction (predicated) instruction class. The encodings in this section are decoded from *SVE Integer Reduction* on page C4-416.

\[
\begin{array}{|c|c|c|c|}
\hline
31 & 30 & 29 & 28 \\
27 & 26 & 25 & 24 \\
23 & 22 & 21 & 20 \\
19 & 18 & 17 & 16 \\
15 & 14 & 13 & 12 \\
10 & 9 &  & 5 \\
0 & 4 &  & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\text{size} & \text{op} & \text{U} & \text{Pg} \\
0 & 0 & 0 & Zn \\
0 & 1 & 0 & Vd \\
\hline
\end{array}
\]

SVE constructive prefix (predicated)

This section describes the encoding of the SVE constructive prefix (predicated) instruction class. The encodings in this section are decoded from *SVE Integer Reduction* on page C4-416.

\[
\begin{array}{|c|c|c|c|}
\hline
31 & 30 & 29 & 28 \\
27 & 26 & 25 & 24 \\
23 & 22 & 21 & 20 \\
19 & 18 & 17 & 16 \\
15 & 14 & 13 & 12 \\
10 & 9 &  & 5 \\
4 &  & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\text{opc} & \text{op} & \text{U} \\
00 & 0 & SADDV \\
01 & 1 & UADDV \\
1x & - & Unallocated. \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
31 & 30 & 29 & 28 \\
27 & 26 & 25 & 24 \\
23 & 22 & 21 & 20 \\
19 & 18 & 17 & 16 \\
15 & 14 & 13 & 12 \\
10 & 9 &  & 5 \\
4 &  & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\text{size} & \text{op} & \text{U} & \text{Pg} \\
0 & 0 & 0 & Zn \\
0 & 1 & 0 & Vd \\
1 & 0 & 1 & Zd \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\text{opc} & \text{Instruction page} \\
00 & MOVPRFX (predicated) \\
01 & Unallocated. \\
1x & Unallocated. \\
\hline
\end{array}
\]
SVE bitwise logical reduction (predicated)

This section describes the encoding of the SVE bitwise logical reduction (predicated) instruction class. The encodings in this section are decoded from SVE Integer Reduction on page C4-416.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0</td>
<td>0 1 1 0</td>
<td>opc</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>

C4.1.15 SVE Bitwise Shift - Predicated

This section describes the encoding of the SVE Bitwise Shift - Predicated group. The encodings in this section are decoded from SVE encodings on page C4-399.

Table C4-16 Encoding table for the SVE Bitwise Shift - Predicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0x</td>
<td>SVE bitwise shift by immediate (predicated) on page C4-418</td>
</tr>
<tr>
<td>10</td>
<td>SVE bitwise shift by vector (predicated) on page C4-419</td>
</tr>
<tr>
<td>11</td>
<td>SVE bitwise shift by wide elements (predicated) on page C4-420</td>
</tr>
</tbody>
</table>

SVE bitwise shift by immediate (predicated)

This section describes the encoding of the SVE bitwise shift by immediate (predicated) instruction class. The encodings in this section are decoded from SVE Bitwise Shift - Predicated.
SVE bitwise shift by vector (predicated)

This section describes the encoding of the SVE bitwise shift by vector (predicated) instruction class. The encodings in this section are decoded from SVE Bitwise Shift - Predicated on page C4-418.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9 8 7</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>tszh</td>
<td>0 0</td>
<td>opc</td>
<td>L</td>
<td>U</td>
<td>1 0</td>
<td>0</td>
</tr>
</tbody>
</table>

Decoding fields

<table>
<thead>
<tr>
<th>opc</th>
<th>L</th>
<th>U</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>ASR (immediate, predicated)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>LSR (immediate, predicated)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>LSL (immediate, predicated)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>ASRD</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>SQSHL (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>UQSHL (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>SRSHR</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>URSHR</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>SQSHLU</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>size</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>R</td>
<td>L</td>
<td>U</td>
</tr>
</tbody>
</table>

Decoding fields

<table>
<thead>
<tr>
<th>R</th>
<th>L</th>
<th>U</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>1</td>
<td>0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 0 0</td>
<td>ASR (vectors)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>LSR (vectors)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td>LSL (vectors)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SVE bitwise shift by wide elements (predicated)

This section describes the encoding of the SVE bitwise shift by wide elements (predicated) instruction class. The encodings in this section are decoded from *SVE Bitwise Shift - Predicated on page C4-418*.

![Instruction Encoding Table](image)

C4.1.16 SVE Integer Unary Arithmetic - Predicated

This section describes the encoding of the SVE Integer Unary Arithmetic - Predicated group. The encodings in this section are decoded from *SVE encodings on page C4-399*.

![Instruction Encoding Table](image)

**Table C4-17 Encoding table for the SVE Integer Unary Arithmetic - Predicated group**

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10</td>
<td><em>SVE integer unary operations (predicated) on page C4-421</em></td>
</tr>
<tr>
<td>11</td>
<td><em>SVE bitwise unary operations (predicated) on page C4-421</em></td>
</tr>
</tbody>
</table>
SVE integer unary operations (predicated)

This section describes the encoding of the SVE integer unary operations (predicated) instruction class. The encodings in this section are decoded from "SVE Integer Unary Arithmetic - Predicated on page C4-420."

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0   0   0   0</td>
<td>1   0   0</td>
<td>size</td>
<td>0   1   0</td>
<td>opc</td>
<td>1   0   1</td>
<td>Pg</td>
<td>Zn</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>SXTB, SXTH, SXTW - Encoding</td>
</tr>
<tr>
<td>001</td>
<td>UXTB, UXTH, UXTW - Encoding</td>
</tr>
<tr>
<td>010</td>
<td>SXTB, SXTH, SXTW - Encoding</td>
</tr>
<tr>
<td>011</td>
<td>UXTB, UXTH, UXTW - Encoding</td>
</tr>
<tr>
<td>100</td>
<td>SXTB, SXTH, SXTW - Encoding</td>
</tr>
<tr>
<td>101</td>
<td>UXTB, UXTH, UXTW - Encoding</td>
</tr>
<tr>
<td>110</td>
<td>ABS</td>
</tr>
<tr>
<td>111</td>
<td>NEG</td>
</tr>
</tbody>
</table>

SVE bitwise unary operations (predicated)

This section describes the encoding of the SVE bitwise unary operations (predicated) instruction class. The encodings in this section are decoded from "SVE Integer Unary Arithmetic - Predicated on page C4-420."

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0   0   0   0</td>
<td>1   0   0</td>
<td>size</td>
<td>0   1   1</td>
<td>opc</td>
<td>1   0   1</td>
<td>Pg</td>
<td>Zn</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>CLS</td>
</tr>
<tr>
<td>001</td>
<td>CLZ</td>
</tr>
<tr>
<td>010</td>
<td>CNT</td>
</tr>
<tr>
<td>011</td>
<td>CNOT</td>
</tr>
<tr>
<td>100</td>
<td>FABS</td>
</tr>
<tr>
<td>101</td>
<td>FNEG</td>
</tr>
<tr>
<td>110</td>
<td>NOT (vector)</td>
</tr>
<tr>
<td>111</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
C4.1.17 SVE Bitwise Logical - Unpredicated

This section describes the encoding of the SVE Bitwise Logical - Unpredicated group. The encodings in this section are decoded from SVE encodings on page C4-399.

<p>| | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>24</td>
<td>23 22 21 20</td>
<td>16</td>
<td>15 13 12</td>
<td>10 9</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000100</td>
<td>1</td>
<td>001</td>
<td>op0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C4-18 Encoding table for the SVE Bitwise Logical - Unpredicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0xx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100</td>
<td>SVE bitwise logical operations (unpredicated) on page C4-422</td>
</tr>
<tr>
<td>101</td>
<td>XAR</td>
</tr>
<tr>
<td>11x</td>
<td>SVE2 bitwise ternary operations on page C4-422</td>
</tr>
</tbody>
</table>

SVE bitwise logical operations (unpredicated)

This section describes the encoding of the SVE bitwise logical operations (unpredicated) instruction class. The encodings in this section are decoded from SVE Bitwise Logical - Unpredicated.

<p>| | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28</td>
<td>27 26 25 24</td>
<td>23 22 21 20</td>
<td>16</td>
<td>15 14 13 12</td>
<td>11 10 9</td>
<td>5 4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 1 0 0</td>
<td>opc</td>
<td>1</td>
<td>Zm</td>
<td>0 0 1 1 0 0</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>AND (vectors, unpredicated)</td>
</tr>
<tr>
<td>01</td>
<td>ORR (vectors, unpredicated)</td>
</tr>
<tr>
<td>10</td>
<td>EOR (vectors, unpredicated)</td>
</tr>
<tr>
<td>11</td>
<td>BIC (vectors, unpredicated)</td>
</tr>
</tbody>
</table>

SVE2 bitwise ternary operations

This section describes the encoding of the SVE2 bitwise ternary operations instruction class. The encodings in this section are decoded from SVE Bitwise Logical - Unpredicated.
C4.1.18   SVE Index Generation

This section describes the encoding of the SVE Index Generation group. The encodings in this section are decoded from SVE encodings on page C4-399.

Table C4-19 Encoding table for the SVE Index Generation group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>INDEX (immediates)</td>
</tr>
<tr>
<td>00</td>
<td>INDEX (scalar, immediate)</td>
</tr>
<tr>
<td>10</td>
<td>INDEX (immediate, scalar)</td>
</tr>
<tr>
<td>11</td>
<td>INDEX (scalars)</td>
</tr>
</tbody>
</table>

C4.1.19   SVE Stack Allocation

This section describes the encoding of the SVE Stack Allocation group. The encodings in this section are decoded from SVE encodings on page C4-399.
SVE stack frame adjustment

This section describes the encoding of the SVE stack frame adjustment instruction class. The encodings in this section are decoded from *SVE Stack Allocation on page C4-423*.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>ADDVL</td>
</tr>
<tr>
<td>1</td>
<td>ADDPL</td>
</tr>
</tbody>
</table>

Streaming SVE stack frame adjustment

This section describes the encoding of the Streaming SVE stack frame adjustment instruction class. The encodings in this section are decoded from *SVE Stack Allocation on page C4-423*.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>ADDSVL</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ADDSPL</td>
<td>FEAT_SME</td>
</tr>
</tbody>
</table>

Table C4-20 Encoding table for the SVE Stack Allocation group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td><em>SVE stack frame adjustment</em></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td><em>Streaming SVE stack frame adjustment</em></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td><em>SVE stack frame size on page C4-425</em></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td><em>Streaming SVE stack frame size on page C4-425</em></td>
</tr>
</tbody>
</table>


SVE stack frame size

This section describes the encoding of the SVE stack frame size instruction class. The encodings in this section are decoded from *SVE Stack Allocation on page C4-423*.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11</th>
<th>10</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>op</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>opc2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>imm6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Decode fields**

<table>
<thead>
<tr>
<th>op</th>
<th>opc2</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>10xxx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>110xx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1110x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>11110</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>11111</td>
<td>RDSVL</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

Streaming SVE stack frame size

This section describes the encoding of the Streaming SVE stack frame size instruction class. The encodings in this section are decoded from *SVE Stack Allocation on page C4-423*.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11</th>
<th>10</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>op</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>opc2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>imm6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Decode fields**

<table>
<thead>
<tr>
<th>op</th>
<th>opc2</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10xxx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>110xx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1110x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11110</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11111</td>
<td>RDSVL</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

C4.1.20  SVE2 Integer Multiply - Unpredicated

This section describes the encoding of the SVE2 Integer Multiply - Unpredicated group. The encodings in this section are decoded from *SVE encodings on page C4-399*. 
This section describes the encoding of the SVE2 integer multiply vectors (unpredicated) instruction class. The encodings in this section are decoded from "SVE2 Integer Multiply - Unpredicated" on page C4-425.

### Table C4-21 Encoding table for the SVE2 Integer Multiply - Unpredicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0x</td>
<td>SVE2 integer multiply vectors (unpredicated)</td>
</tr>
<tr>
<td>10</td>
<td>SVE2 signed saturating doubling multiply high (unpredicated)</td>
</tr>
<tr>
<td>11</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

### SVE2 integer multiply vectors (unpredicated)

This section describes the encoding of the SVE2 integer multiply vectors (unpredicated) instruction class. The encodings in this section are decoded from "SVE2 Integer Multiply - Unpredicated" on page C4-425.

### SVE2 signed saturating doubling multiply high (unpredicated)

This section describes the encoding of the SVE2 signed saturating doubling multiply high (unpredicated) instruction class. The encodings in this section are decoded from "SVE2 Integer Multiply - Unpredicated" on page C4-425.
C4.1.21 SVE Bitwise Shift - Unpredicated

This section describes the encoding of the SVE Bitwise Shift - Unpredicated group. The encodings in this section are decoded from SVE encodings on page C4-399.

Table C4-22 Encoding table for the SVE Bitwise Shift - Unpredicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SVE bitwise shift by wide elements (unpredicated) (op0) on page C4-427</td>
</tr>
<tr>
<td>1</td>
<td>SVE bitwise shift by immediate (unpredicated) (op0) on page C4-428</td>
</tr>
</tbody>
</table>

SVE bitwise shift by wide elements (unpredicated)

This section describes the encoding of the SVE bitwise shift by wide elements (unpredicated) instruction class. The encodings in this section are decoded from SVE Bitwise Shift - Unpredicated.

Table C4-22 Encoding table for the SVE Bitwise Shift - Unpredicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>ASR (wide elements, unpredicated)</td>
</tr>
<tr>
<td>01</td>
<td>LSR (wide elements, unpredicated)</td>
</tr>
<tr>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11</td>
<td>LSL (wide elements, unpredicated)</td>
</tr>
</tbody>
</table>
SVE bitwise shift by immediate (unpredicated)

This section describes the encoding of the SVE bitwise shift by immediate (unpredicated) instruction class. The encodings in this section are decoded from *SVE Bitwise Shift - Unpredicated* on page C4-427.

|        | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| opc    | 00 | 01 | 10 | 11 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|        | ASR (immediate, unpredicated) | LSR (immediate, unpredicated) | Unallocated. | LSL (immediate, unpredicated) |

C4.22 SVE Integer Misc - Unpredicated

This section describes the encoding of the SVE Integer Misc - Unpredicated group. The encodings in this section are decoded from *SVE encodings* on page C4-399.

|        | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|        | 00000100 | 1 | 1011 |      |      |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Table C4-23 Encoding table for the SVE Integer Misc - Unpredicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0x</td>
<td><em>SVE floating-point trig select coefficient</em> on page C4-428</td>
</tr>
<tr>
<td>10</td>
<td><em>SVE floating-point exponential accelerator</em> on page C4-429</td>
</tr>
<tr>
<td>11</td>
<td><em>SVE constructive prefix (unpredicated)</em> on page C4-429</td>
</tr>
</tbody>
</table>

SVE floating-point trig select coefficient

This section describes the encoding of the SVE floating-point trig select coefficient instruction class. The encodings in this section are decoded from *SVE Integer Misc - Unpredicated*. 
### SVE floating-point exponential accelerator

This section describes the encoding of the SVE floating-point exponential accelerator instruction class. The encodings in this section are decoded from *SVE Integer Misc - Unpredicated on page C4-428*.

```
[31 30 29 28][27 26 25 24][23 22 21 20] [16][15 14 13 12][11 10 9] 5 4 0
0 0 0 0 0 1 0 0 | size | 1 | Zm | 1 | 0 | 1 | 1 | 0 | op | Zn | Zd
```

**Decode fields**

- **op**
  - 0: FTSSEL
  - 1: Unallocated.

### SVE constructive prefix (unpredicated)

This section describes the encoding of the SVE constructive prefix (unpredicated) instruction class. The encodings in this section are decoded from *SVE Integer Misc - Unpredicated on page C4-428*.

```
[31 30 29 28][27 26 25 24][23 22 21 20] [16][15 14 13 12][11 10 9] 5 4 0
0 0 0 0 0 1 0 0 | size | 1 | opc | 1 | 0 | 1 | 1 | 0 | Zn | Zd
```

**Decode fields**

- **opc**
  - 00000: FEXPA
  - 00001: Unallocated.
  - 0001x: Unallocated.
  - 001xx: Unallocated.
  - 01xxx: Unallocated.
  - 1xxxx: Unallocated.
C4.1.23 SVE Element Count

This section describes the encoding of the SVE Element Count group. The encodings in this section are decoded from SVE encodings on page C4-399.

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[19]</th>
<th>[16 15 14 13]</th>
<th>[12 11 10 9]</th>
<th>[5 4]</th>
<th>[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000100</td>
<td>1</td>
<td>11</td>
<td>op1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

op0

Table C4-24 Encoding table for the SVE Element Count group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>0 00x</td>
<td>SVE saturating inc/dec vector by element count on page C4-431</td>
</tr>
<tr>
<td>0 100</td>
<td>SVE element count on page C4-431</td>
</tr>
<tr>
<td>0 101</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 000</td>
<td>SVE inc/dec vector by element count on page C4-432</td>
</tr>
<tr>
<td>1 100</td>
<td>SVE inc/dec register by element count on page C4-432</td>
</tr>
<tr>
<td>1 x01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- 01x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- 11x</td>
<td>SVE saturating inc/dec register by element count on page C4-433</td>
</tr>
</tbody>
</table>
### SVE saturating inc/dec vector by element count

This section describes the encoding of the SVE saturating inc/dec vector by element count instruction class. The encodings in this section are decoded from `SVE Element Count` on page C4-430.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>size</td>
<td>1 0</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>imm4</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>pattern</td>
<td>Zdn</td>
<td></td>
</tr>
</tbody>
</table>
```

**Decode fields**

<table>
<thead>
<tr>
<th>size</th>
<th>D</th>
<th>U</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>SQINCH (vector)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>UQINCH (vector)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>SQDECH (vector)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>UQDECH (vector)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>SQINCW (vector)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>UQINCW (vector)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>SQDECW (vector)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>UQDECW (vector)</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>SQINCD (vector)</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>UQINCD (vector)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>SQDECD (vector)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>UQDECD (vector)</td>
</tr>
</tbody>
</table>

### SVE element count

This section describes the encoding of the SVE element count instruction class. The encodings in this section are decoded from `SVE Element Count` on page C4-430.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>size</td>
<td>1 0</td>
</tr>
<tr>
<td>1 1 1 0 0</td>
<td>op</td>
<td></td>
</tr>
<tr>
<td>pattern</td>
<td>Rd</td>
<td></td>
</tr>
</tbody>
</table>
```

**Decode fields**

<table>
<thead>
<tr>
<th>size</th>
<th>op</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>CNTB, CNTD, CNTH, CNTW - Encoding</td>
</tr>
</tbody>
</table>
SVE inc/dec vector by element count

This section describes the encoding of the SVE inc/dec vector by element count instruction class. The encodings in this section are decoded from *SVE Element Count* on page C4-430.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>op</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

SVE inc/dec register by element count

This section describes the encoding of the SVE inc/dec register by element count instruction class. The encodings in this section are decoded from *SVE Element Count* on page C4-430.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>D</td>
</tr>
<tr>
<td>00</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>D</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
</tbody>
</table>
### SVE saturating inc/dec register by element count

This section describes the encoding of the SVE saturating inc/dec register by element count instruction class. The encodings in this section are decoded from \textit{SVE Element Count} on page C4-430.

<table>
<thead>
<tr>
<th>size</th>
<th>sf</th>
<th>imm4</th>
<th>D</th>
<th>U</th>
<th>pattern</th>
<th>Rdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>1111</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0000</td>
<td>1</td>
<td>0</td>
<td>1111</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0000</td>
<td>1</td>
<td>1</td>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>sf</th>
<th>D</th>
<th>U</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SQINCB - Encoding</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>UQINCB - Encoding</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SQDECB - Encoding</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>UQDECB - Encoding</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SQINCH - Encoding</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>UQINCH - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SQINCH (scalar) - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>UQINCH (scalar) - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SQDECH (scalar) - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>UQDECH (scalar) - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SQINCH (scalar) - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>UQINCH (scalar) - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>SQDECH (scalar) - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>UQDECH (scalar) - Encoding</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SQINCW (scalar) - Encoding</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>UQINCW (scalar) - Encoding</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SQDECW (scalar) - Encoding</td>
</tr>
</tbody>
</table>
C4.1.24 SVE Bitwise Immediate

This section describes the encoding of the SVE Bitwise Immediate group. The encodings in this section are decoded from SVE encodings on page C4-399.

<table>
<thead>
<tr>
<th>31</th>
<th>24 23 22 21 20 19 18 17</th>
<th></th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000101</td>
<td>op0</td>
<td>00</td>
<td>op1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C4-25 Encoding table for the SVE Bitwise Immediate group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>11 00</td>
<td>DUPM</td>
</tr>
<tr>
<td>!1 !0</td>
<td>SVE bitwise logical with immediate (unpredicated) on page C4-434</td>
</tr>
<tr>
<td>- !0</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SVE bitwise logical with immediate (unpredicated)

This section describes the encoding of the SVE bitwise logical with immediate (unpredicated) instruction class. The encodings in this section are decoded from SVE Bitwise Immediate.
C4.1.25 SVE Integer Wide Immediate - Predicated

This section describes the encoding of the SVE Integer Wide Immediate - Predicated group. The encodings in this section are decoded from SVE encodings on page C4-399.

Table C4-26 Encoding table for the SVE Integer Wide Immediate - Predicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0xx</td>
<td>SVE copy integer immediate (predicated) on page C4-335</td>
</tr>
<tr>
<td>10x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>110</td>
<td>FCPY</td>
</tr>
<tr>
<td>111</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SVE copy integer immediate (predicated)

This section describes the encoding of the SVE copy integer immediate (predicted) instruction class. The encodings in this section are decoded from SVE Integer Wide Immediate - Predicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>CPY (immediate, zeroing)</td>
</tr>
<tr>
<td>1</td>
<td>CPY (immediate, merging)</td>
</tr>
</tbody>
</table>
C4.1.26 SVE Permute Vector - Unpredicated

This section describes the encoding of the SVE Permute Vector - Unpredicated group. The encodings in this section are decoded from *SVE encodings* on page C4-399.

| 31 | 24|23 22 21 20|19 18 16|15 | 10 9 | 0 |
|----|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 00000101 | 1 | op0 | op1 | 001110 | 0 |

Table C4-27 Encoding table for the SVE Permute Vector - Unpredicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>00 000</td>
<td>DUP (scalar)</td>
</tr>
<tr>
<td>00 100</td>
<td>INSR (scalar)</td>
</tr>
<tr>
<td>00 x10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00 xx1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01 -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10 0xx</td>
<td><em>SVE unpack vector elements</em> on page C4-436</td>
</tr>
<tr>
<td>10 100</td>
<td>INSR (SIMD&amp;FP scalar)</td>
</tr>
<tr>
<td>10 110</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10 1x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11 000</td>
<td>REV (vector)</td>
</tr>
<tr>
<td>11 != 000</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SVE unpack vector elements

This section describes the encoding of the SVE unpack vector elements instruction class. The encodings in this section are decoded from *SVE Permute Vector - Unpredicated*.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 1</td>
<td>size</td>
<td>1 1 0 0</td>
<td>U</td>
<td>H</td>
<td>0 0 1 1 1 0</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>U H</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>SUNPKHI, SUNPKLO - Encoding</td>
</tr>
<tr>
<td>0 1</td>
<td>SUNPKHI, SUNPKLO - Encoding</td>
</tr>
<tr>
<td>1 0</td>
<td>UUNPKHI, UUNPKLO - Encoding</td>
</tr>
<tr>
<td>1 1</td>
<td>UUNPKHI, UUNPKLO - Encoding</td>
</tr>
</tbody>
</table>
C4.1.27 SVE Permute Predicate

This section describes the encoding of the SVE Permute Predicate group. The encodings in this section are decoded from SVE encodings on page C4-399.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000101</td>
<td>op0 1</td>
<td>op1</td>
<td>010</td>
<td>op2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C4-28 Encoding table for the SVE Permute Predicate group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2 op3</td>
<td>SVE unpack predicate elements on page C4-437</td>
</tr>
<tr>
<td>00 1000x 0000 0</td>
<td>SVE unpack predicate elements on page C4-437</td>
</tr>
<tr>
<td>01 1000x 0000 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10 1000x 0000 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11 1000x 0000 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- 0xxxx xxx0 0</td>
<td>SVE permute predicate elements on page C4-438</td>
</tr>
<tr>
<td>- 0xxxx xxx1 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- 10100 0000 0</td>
<td>REV (predicate)</td>
</tr>
<tr>
<td>- 10101 0000 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- 10x0x 1000 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- 10x0x x100 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- 10x0x xx10 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- 10x0x xxx1 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- 10x1x - 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- 11xxx - 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- - - 1</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SVE unpack predicate elements

This section describes the encoding of the SVE unpack predicate elements instruction class. The encodings in this section are decoded from SVE Permute Predicate.
This section describes the encoding of the SVE permute predicate elements instruction class. The encodings in this section are decoded from \textit{SVE Permute Predicate on page C4-437}.

\textbf{SVE permute predicate elements}

This section describes the encoding of the SVE permute predicate elements instruction class. The encodings in this section are decoded from \textit{SVE Permute Predicate on page C4-437}.

\textbf{C4.1.28 SVE Permute Vector - Predicated}

This section describes the encoding of the SVE Permute Vector - Predicated group. The encodings in this section are decoded from \textit{SVE encodings on page C4-399}.
This section describes the encoding of the SVE extract element to general register instruction class. The encodings in this section are decoded from *SVE Permute Vector - Predicated* on page C4-438.

### Table C4-29 Encoding table for the SVE Permute Vector - Predicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
</tr>
<tr>
<td>0</td>
<td>01x</td>
</tr>
<tr>
<td>0</td>
<td>01x</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
</tr>
<tr>
<td>0</td>
<td>x01</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>!= 000</td>
</tr>
</tbody>
</table>
SVE extract element to SIMD&FP scalar register

This section describes the encoding of the SVE extract element to SIMD&FP scalar register instruction class. The encodings in this section are decoded from SVE Permute Vector - Predicated on page C4-438.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 | 10 9 | 5 4 | 0 |
| 0 0 0 0 0 1 0 1 | size | 1 0 0 0 | B | 1 0 1 | Pg | Zn | Rd |
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LASTA (scalar)</td>
</tr>
<tr>
<td>1</td>
<td>LASTB (scalar)</td>
</tr>
</tbody>
</table>

SVE reverse within elements

This section describes the encoding of the SVE reverse within elements instruction class. The encodings in this section are decoded from SVE Permute Vector - Predicated on page C4-438.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 | 10 9 | 5 4 | 0 |
| 0 0 0 0 0 1 0 1 | size | 1 0 0 0 | B | 1 0 0 | Pg | Zn | Vd |
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LASTA (SIMD&amp;FP scalar)</td>
</tr>
<tr>
<td>1</td>
<td>LASTB (SIMD&amp;FP scalar)</td>
</tr>
</tbody>
</table>

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 | 10 9 | 5 4 | 0 |
| 0 0 0 0 0 1 0 1 | size | 1 0 0 1 | opc | 1 0 0 | Pg | Zn | Zd |
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>REVB, REVH, REVW - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>REVB, REVH, REVW - Encoding</td>
</tr>
<tr>
<td>10</td>
<td>REVB, REVH, REVW - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>RBIT</td>
</tr>
</tbody>
</table>
SVE conditionally broadcast element to vector

This section describes the encoding of the SVE conditionally broadcast element to vector instruction class. The encodings in this section are decoded from *SVE Permute Vector - Predicated* on page C4-438.

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 |
| 0 0 0 0 1 0 1 | size | 1 0 1 0 0 | B | 1 0 0 | Pg | Zm | Zdn |
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>CLASTA (vectors)</td>
</tr>
<tr>
<td>1</td>
<td>CLASTB (vectors)</td>
</tr>
</tbody>
</table>

SVE conditionally extract element to SIMD&FP scalar

This section describes the encoding of the SVE conditionally extract element to SIMD&FP scalar instruction class. The encodings in this section are decoded from *SVE Permute Vector - Predicated* on page C4-438.

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 |
| 0 0 0 0 1 0 1 | size | 1 0 1 0 1 | B | 1 0 0 | Pg | Zm | Vdn |
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>CLASTA (SIMD&amp;FP scalar)</td>
</tr>
<tr>
<td>1</td>
<td>CLASTB (SIMD&amp;FP scalar)</td>
</tr>
</tbody>
</table>

SVE reverse doublewords

This section describes the encoding of the SVE reverse doublewords instruction class. The encodings in this section are decoded from *SVE Permute Vector - Predicated* on page C4-438.

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 |
| 0 0 0 0 1 0 1 | size | 1 0 1 1 0 1 0 0 | Pg | Zn | Zd |
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>REVD</td>
<td>FEAT_SME</td>
</tr>
<tr>
<td>01</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
SVE conditionally extract element to general register

This section describes the encoding of the SVE conditionally extract element to general register instruction class. The encodings in this section are decoded from SVE Permute Vector - Predicated on page C4-438.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 1</td>
<td>1 1 0 0 0 B</td>
<td>1 0 1 Pg</td>
<td>Zm Rdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

### C4.1.29 SVE Permute Vector - Extract

This section describes the encoding of the SVE Permute Vector - Extract group. The encodings in this section are decoded from SVE encodings on page C4-399.

```
<table>
<thead>
<tr>
<th>31</th>
<th>23 22 21 20</th>
<th>16 15 13 12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001010</td>
<td>1</td>
<td>000</td>
<td></td>
</tr>
</tbody>
</table>
```

Table C4-30 Encoding table for the SVE Permute Vector - Extract group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>EXT - Encoding</td>
</tr>
</tbody>
</table>

### C4.1.30 SVE Permute Vector - Segments

This section describes the encoding of the SVE Permute Vector - Segments group. The encodings in this section are decoded from SVE encodings on page C4-399.
This section describes the encoding of the SVE permute vector segments instruction class. The encodings in this section are decoded from "SVE Permute Vector - Segments on page C4-442."

This section describes the encoding of the SVE Integer Compare - Vectors group. The encodings in this section are decoded from "SVE encodings on page C4-399."
This section describes the encoding of the SVE integer compare vectors instruction class. The encodings in this section are decoded from *SVE Integer Compare - Vectors* on page C4-443.

### SVE integer compare vectors

This section describes the encoding of the SVE integer compare vectors instruction class. The encodings in this section are decoded from *SVE Integer Compare - Vectors* on page C4-443.

### SVE integer compare with wide elements

This section describes the encoding of the SVE integer compare with wide elements instruction class. The encodings in this section are decoded from *SVE Integer Compare - Vectors* on page C4-443.
C4.1.32 SVE Propagate Break

This section describes the encoding of the SVE Propagate Break group. The encodings in this section are decoded from *SVE encodings* on page C4-399.

![Table C4-33 Encoding table for the SVE Propagate Break group](image)

**SVE propagate break from previous partition**

This section describes the encoding of the SVE propagate break from previous partition instruction class. The encodings in this section are decoded from *SVE Propagate Break*. 
C4.1.33 SVE Partition Break

This section describes the encoding of the SVE Partition Break group. The encodings in this section are decoded from SVE encodings on page C4-399.

Table C4-34 Encoding table for the SVE Partition Break group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2 op3</td>
<td>SVE propagate break to next partition on page C4-446</td>
</tr>
<tr>
<td>0 1000 0 0</td>
<td>SVE propagate break to next partition on page C4-446</td>
</tr>
<tr>
<td>0 1000 0 1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 x000 1 -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 x1xx - -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 xx1x - -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 xxx1 - -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 0000 1 -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 != 0000 - -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- 0000 0 -</td>
<td>SVE partition break condition on page C4-447</td>
</tr>
</tbody>
</table>

SVE propagate break to next partition

This section describes the encoding of the SVE propagate break to next partition instruction class. The encodings in this section are decoded from SVE Partition Break.
SVE partition break condition

This section describes the encoding of the SVE partition break condition instruction class. The encodings in this section are decoded from SVE Partition Break on page C4-446.

C4.1.34 SVE Predicate Misc

This section describes the encoding of the SVE Predicate Misc group. The encodings in this section are decoded from SVE encodings on page C4-399.
This section describes the encoding of the SVE predicate test instruction class. The encodings in this section are decoded from *SVE Predicate Misc on page C4-447*.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2 op3 op4</td>
<td></td>
</tr>
<tr>
<td>0000 - x0 - - 0</td>
<td><em>SVE predicate test</em></td>
</tr>
<tr>
<td>0100 - x0 - - 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0x10 - x0 - - 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0xx1 - x0 - - 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0xxx - xl - - 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1000 000 00 - - 0</td>
<td><em>SVE predicate first active on page C4-449</em></td>
</tr>
<tr>
<td>1000 000 != 00 - - 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1000 100 10 0000 0</td>
<td><em>SVE predicate zero on page C4-449</em></td>
</tr>
<tr>
<td>1000 100 10 != 0000 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1000 110 00 - - 0</td>
<td><em>SVE predicate read from FFR (predicated) on page C4-450</em></td>
</tr>
<tr>
<td>1001 000 0x - - 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1001 000 10 - - 0</td>
<td>PNEXT</td>
</tr>
<tr>
<td>1001 000 11 - - 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1001 100 10 - - 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1001 110 00 0000 0</td>
<td><em>SVE predicate read from FFR (unpredicated) on page C4-450</em></td>
</tr>
<tr>
<td>1001 110 00 != 0000 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100x 010 - - - 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100x 100 0x - - 0</td>
<td><em>SVE predicate initialize on page C4-451</em></td>
</tr>
<tr>
<td>100x 100 11 - - 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100x 110 != 00 - - 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100x xx1 - - - 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>110x - - - - 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1x1x - - - - 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- - - - - 1</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

**SVE predicate test**

This section describes the encoding of the SVE predicate test instruction class. The encodings in this section are decoded from *SVE Predicate Misc on page C4-447*. 
SVE predicate first active

This section describes the encoding of the SVE predicate first active instruction class. The encodings in this section are decoded from *SVE Predicate Misc* on page C4-447.

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] [11 10 9 8] [7 6 5 4] [3 2 1 0]
| 0 0 1 0 0 1 0 1 | op | S | 0 1 0 0 0 1 1 | Pg | 0 | Pn | 0 | opc2 |
```

Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>opc2</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 1 0000</td>
<td></td>
<td></td>
<td>PTEST</td>
</tr>
<tr>
<td>0 1 0001</td>
<td></td>
<td></td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 1 001x</td>
<td></td>
<td></td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 1 01xx</td>
<td></td>
<td></td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 1 1xxx</td>
<td></td>
<td></td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 - -</td>
<td></td>
<td></td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SVE predicate zero

This section describes the encoding of the SVE predicate zero instruction class. The encodings in this section are decoded from *SVE Predicate Misc* on page C4-447.

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] [11 10 9 8] [7 6 5 4] [3 2 1 0]
| 0 0 1 0 0 1 0 1 | op | S | 0 1 1 0 0 0 1 1 | Pg | 0 | Pdn |
```

Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 -</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
This section describes the encoding of the SVE predicate read from FFR (predicated) instruction class. The encodings in this section are decoded from SVE Predicate Misc on page C4-447.

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0]
0 0 1 0 0 1 0 1 op S 0 1 1 0 0 0 1 1 1 0 0 1 0 0 0 0 0 Pd
```

**Decode fields**

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>PFALSE</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

**SVE predicate read from FFR (unpredicated)**

This section describes the encoding of the SVE predicate read from FFR (unpredicated) instruction class. The encodings in this section are decoded from SVE Predicate Misc on page C4-447.

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0]
0 0 1 0 0 1 0 1 op S 0 1 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 Pd
```

**Decode fields**

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>RDFFR (unpredicated)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RDFFRS</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
SVE predicate initialize

This section describes the encoding of the SVE predicate initialize instruction class. The encodings in this section are decoded from SVE Predicate Misc on page C4-447.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>PTRUE</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pattern</th>
<th>0</th>
</tr>
</thead>
</table>

Table C4-36 Encoding table for the SVE Integer Compare - Scalars group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>op0</th>
<th>op1</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>-</td>
<td>-</td>
<td>SVE integer compare scalar count and limit on page C4-451</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>0000</td>
<td>SVE conditionally terminate scalars on page C4-452</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>!= 0000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>-</td>
<td>SVE pointer conflict compare on page C4-452</td>
</tr>
<tr>
<td>1x</td>
<td>!= 00</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SVE integer compare scalar count and limit

This section describes the encoding of the SVE integer compare scalar count and limit instruction class. The encodings in this section are decoded from SVE Integer Compare - Scalars.
SVE conditionally terminate scalars

This section describes the encoding of the SVE conditionally terminate scalars instruction class. The encodings in this section are decoded from *SVE Integer Compare - Scalars on page C4-451.*

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] [11 10 9 8] [7 6 5 4] [3 2 1 0]
0 0 1 0 0 1 0 1 size 1 Rm 0 0 0 sf Ul It Rn eq Pd
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>U  It eq</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>WHILEGE</td>
</tr>
<tr>
<td>0 0 1</td>
<td>WHILEGT</td>
</tr>
<tr>
<td>0 1 0</td>
<td>WHILELT</td>
</tr>
<tr>
<td>0 1 1</td>
<td>WHILELE</td>
</tr>
<tr>
<td>1 0 0</td>
<td>WHILEHS</td>
</tr>
<tr>
<td>1 0 1</td>
<td>WHILEHI</td>
</tr>
<tr>
<td>1 1 0</td>
<td>WHILELO</td>
</tr>
<tr>
<td>1 1 1</td>
<td>WHILELS</td>
</tr>
</tbody>
</table>

SVE pointer conflict compare

This section describes the encoding of the SVE pointer conflict compare instruction class. The encodings in this section are decoded from *SVE Integer Compare - Scalars on page C4-451.*

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] [11 10 9 8] [7 6 5 4] [3 2 1 0]
0 0 1 0 0 1 0 1 op sz 1 Rm 0 0 1 0 0 0 Rn ne 0 0 0 0
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op ne</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 0</td>
<td>CTERMEQ, CTERMNE - Encoding</td>
</tr>
<tr>
<td>1 1</td>
<td>CTERMEQ, CTERMNE - Encoding</td>
</tr>
</tbody>
</table>
This section describes the encoding of the SVE Integer Wide Immediate - Unpredicated group. The encodings in this section are decoded from **SVE encodings on page C4-399**.

**SVE Integer Wide Immediate - Unpredicated**

This section describes the encoding of the SVE Integer Wide Immediate - Unpredicated group. The encodings in this section are decoded from **SVE encodings on page C4-399**.

```
| 31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13 12|11 10 9 | 5 4 3 | 0 |
|--------------|--------------|--------------|----|----------|----------|----|----|----|
| 0 0 1 0 0 1 0 1 | size | 1 | Rm | 0 0 1 1 0 0 | Rn | rw | Pd |
```

**Table C4-37 Encoding table for the SVE Integer Wide Immediate - Unpredicated group**

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td><strong>SVE integer add/subtract immediate (unpredicated)</strong> on page C4-453</td>
</tr>
<tr>
<td>01</td>
<td><strong>SVE integer min/max immediate (unpredicated)</strong> on page C4-454</td>
</tr>
<tr>
<td>10</td>
<td><strong>SVE integer multiply immediate (unpredicated)</strong> on page C4-454</td>
</tr>
<tr>
<td>11 0</td>
<td><strong>SVE broadcast integer immediate (unpredicated)</strong> on page C4-455</td>
</tr>
<tr>
<td>11 1</td>
<td><strong>SVE broadcast floating-point immediate (unpredicated)</strong> on page C4-455</td>
</tr>
</tbody>
</table>

**SVE integer add/subtract immediate (unpredicated)**

This section describes the encoding of the SVE integer add/subtract immediate (unpredicated) instruction class. The encodings in this section are decoded from **SVE Integer Wide Immediate - Unpredicated**.
SVE integer min/max immediate (unpredicated)

This section describes the encoding of the SVE integer min/max immediate (unpredicated) instruction class. The encodings in this section are decoded from *SVE Integer Wide Immediate - Unpredicated* on page C4-453.

SVE integer multiply immediate (unpredicated)

This section describes the encoding of the SVE integer multiply immediate (unpredicated) instruction class. The encodings in this section are decoded from *SVE Integer Wide Immediate - Unpredicated* on page C4-453.
SVE broadcast integer immediate (unpredicated)

This section describes the encoding of the SVE broadcast integer immediate (unpredicated) instruction class. The encodings in this section are decoded from SVE Integer Wide Immediate - Unpredicated on page C4-453.

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 16 15 14 13 12] | 5 4 | 0 |
0 0 1 0 0 1 0 1 size 1 1 0 opc 1 1 o2 imm8 Zdn
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>o2</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>1</td>
</tr>
<tr>
<td>001</td>
<td>-</td>
</tr>
<tr>
<td>01x</td>
<td>-</td>
</tr>
<tr>
<td>1xx</td>
<td>-</td>
</tr>
</tbody>
</table>

SVE broadcast floating-point immediate (unpredicated)

This section describes the encoding of the SVE broadcast floating-point immediate (unpredicated) instruction class. The encodings in this section are decoded from SVE Integer Wide Immediate - Unpredicated on page C4-453.

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] | 5 4 | 0 |
0 0 1 0 0 1 0 1 size 1 1 1 opc 0 1 sh imm8 Zd
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>sh</td>
</tr>
<tr>
<td>00</td>
<td>DUP (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1x</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
This section describes the encoding of the SVE Predicate Count group. The encodings in this section are decoded from SVE encodings on page C4-399.

### C4.1.37 SVE Predicate Count

This section describes the encoding of the SVE predicate count instruction class. The encodings in this section are decoded from SVE Predicate Count.

---

Table C4-38 Encoding table for the SVE Predicate Count group

<table>
<thead>
<tr>
<th>op0</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><em>SVE predicate count</em> on page C4-456</td>
</tr>
<tr>
<td>1</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SVE predicate count

This section describes the encoding of the SVE predicate count instruction class. The encodings in this section are decoded from *SVE Predicate Count*.
C4.1.38   SVE Inc/Dec by Predicate Count

This section describes the encoding of the SVE Inc/Dec by Predicate Count group. The encodings in this section are decoded from SVE encodings on page C4-399.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>SVE saturating inc/dec vector by predicate count on page C4-457</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>SVE saturating inc/dec register by predicate count on page C4-458</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>SVE inc/dec vector by predicate count on page C4-458</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SVE inc/dec register by predicate count on page C4-459</td>
</tr>
</tbody>
</table>

SVE saturating inc/dec vector by predicate count

This section describes the encoding of the SVE saturating inc/dec vector by predicate count instruction class. The encodings in this section are decoded from SVE Inc/Dec by Predicate Count.

<table>
<thead>
<tr>
<th>31 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 10 9 8 5 4</th>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1</td>
<td>size</td>
<td>1 0 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th>U</th>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>SQINCP (vector)</td>
</tr>
</tbody>
</table>
SVE saturating inc/dec register by predicate count

This section describes the encoding of the SVE saturating inc/dec register by predicate count instruction class. The encodings in this section are decoded from SVE Inc/Dec by Predicate Count on page C4-457.

```
<table>
<thead>
<tr>
<th></th>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>0 1 0 0 1 0 1</td>
<td>size 1 0 1 0</td>
<td>D</td>
<td>U</td>
<td>1 0 0 0 1</td>
<td>sf op</td>
<td>Pm</td>
<td>Rdn</td>
</tr>
</tbody>
</table>
```

SVE inc/dec vector by predicate count

This section describes the encoding of the SVE inc/dec vector by predicate count instruction class. The encodings in this section are decoded from SVE Inc/Dec by Predicate Count on page C4-457.
SVE inc/dec register by predicate count

This section describes the encoding of the SVE inc/dec register by predicate count instruction class. The encodings in this section are decoded from SVE Inc/Dec by Predicate Count on page C4-457.

D31 D30 D29 D28 D27 D26 D25 D24 D23 D22 D21 D20 D19 D18 D17 D16 D15 D14 D13 D12 D11 D10 D9 D8 | 5 4 | 0 |
0 0 1 0 0 1 0 1 | size | 1 0 1 1 | op | D | 0 0 0 0 | opc2 | Pm | Zdn

Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>D</th>
<th>opc2</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  - 01</td>
<td>Unallocated.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0  - 1x</td>
<td>Unallocated.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0  0 00</td>
<td>INCP (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0  1 00</td>
<td>DECP (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1  -  -</td>
<td>Unallocated.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C4.1.39   SVE Write FFR

This section describes the encoding of the SVE Write FFR group. The encodings in this section are decoded from SVE encodings on page C4-399.
This section describes the encoding of the SVE FFR write from predicate instruction class. The encodings in this section are decoded from *SVE Write FFR* on page C4-459.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00 000 - 00000</td>
<td><em>SVE FFR write from predicate</em></td>
</tr>
<tr>
<td>1 00 000 0000 00000</td>
<td><em>SVE FFR initialise</em></td>
</tr>
<tr>
<td>1 00 000 lxxx 00000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 00 000 x1xx 00000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 00 000 xx1x 00000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 00 000 xxx1 00000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- 00 000 - != 00000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- 00 != 000 - -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- != 00 - - -</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

### SVE FFR initialise

This section describes the encoding of the SVE FFR initialise instruction class. The encodings in this section are decoded from *SVE Write FFR* on page C4-459.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>WRFFR</td>
</tr>
<tr>
<td>00</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1x</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
C4.1.40 SVE Integer Multiply-Add - Unpredicated

This section describes the encoding of the SVE Integer Multiply-Add - Unpredicated group. The encodings in this section are decoded from SVE encodings on page C4-399.

Table C4-41 Encoding table for the SVE Integer Multiply-Add - Unpredicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0000x</td>
<td>SVE integer dot product (unpredicated) on page C4-461</td>
</tr>
<tr>
<td>0001x</td>
<td>SVE2 saturating multiply-add interleaved long on page C4-462</td>
</tr>
<tr>
<td>001xx</td>
<td>CDOT (vectors)</td>
</tr>
<tr>
<td>01xxx</td>
<td>SVE2 complex integer multiply-add on page C4-462</td>
</tr>
<tr>
<td>10xxx</td>
<td>SVE2 integer multiply-add long on page C4-462</td>
</tr>
<tr>
<td>110xx</td>
<td>SVE2 saturating multiply-add long on page C4-463</td>
</tr>
<tr>
<td>1110x</td>
<td>SVE2 saturating multiply-add high on page C4-463</td>
</tr>
<tr>
<td>11110</td>
<td>SVE mixed sign dot product on page C4-464</td>
</tr>
<tr>
<td>11111</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SVE integer dot product (unpredicated)

This section describes the encoding of the SVE integer dot product (unpredicated) instruction class. The encodings in this section are decoded from SVE Integer Multiply-Add - Unpredicated.
SVE2 saturating multiply-add interleaved long

This section describes the encoding of the SVE2 saturating multiply-add interleaved long instruction class. The encodings in this section are decoded from *SVE Integer Multiply-Add - Unpredicated* on page C4-461.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>size</td>
<td>0</td>
<td>Zm</td>
<td>0 0 0 0 0</td>
<td>U</td>
<td>Zn</td>
<td>Zda</td>
</tr>
</tbody>
</table>
```

**Decode fields**  
**Instruction page**  

| U |  
|---|---
| 0 | SDOT (vectors)  
| 1 | UDOT (vectors) |

SVE2 complex integer multiply-add

This section describes the encoding of the SVE2 complex integer multiply-add instruction class. The encodings in this section are decoded from *SVE Integer Multiply-Add - Unpredicated* on page C4-461.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>size</td>
<td>0</td>
<td>Zm</td>
<td>0 0 0 0 1</td>
<td>S</td>
<td>Zn</td>
<td>Zda</td>
</tr>
</tbody>
</table>
```

**Decode fields**  
**Instruction page**  

| S |  
|---|---
| 0 | SQDMLALBT  
| 1 | SQDMLSLBT |

SVE2 integer multiply-add long

This section describes the encoding of the SVE2 integer multiply-add long instruction class. The encodings in this section are decoded from *SVE Integer Multiply-Add - Unpredicated* on page C4-461.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>size</td>
<td>0</td>
<td>Zm</td>
<td>0 0 1</td>
<td>op</td>
<td>rot</td>
<td>Zn</td>
</tr>
</tbody>
</table>
```

**Decode fields**  
**Instruction page**  

| op |  
|----|---
| 0 | CMLA (vectors)  
| 1 | SQRDCMLAH (vectors) |

---

C4-462  
Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  
Non-Confidential  
ARM DDI 0487I.a  
ID081822
C4.1 A64 instruction set encoding

This section describes the encoding of the SVE2 saturating multiply-add long instruction class. The encodings in this section are decoded from *SVE Integer Multiply-Add - Unpredicated* on page C4-461.

### SVE2 saturating multiply-add long

This section describes the encoding of the SVE2 saturating multiply-add long instruction class. The encodings in this section are decoded from *SVE Integer Multiply-Add - Unpredicated* on page C4-461.

### SVE2 saturating multiply-add high

This section describes the encoding of the SVE2 saturating multiply-add high instruction class. The encodings in this section are decoded from *SVE Integer Multiply-Add - Unpredicated* on page C4-461.
SVE mixed sign dot product

This section describes the encoding of the SVE mixed sign dot product instruction class. The encodings in this section are decoded from *SVE Integer Multiply-Add - Unpredicated* on page C4-461.

C4.1.41 SVE2 Integer - Predicated

This section describes the encoding of the SVE2 Integer - Predicated group. The encodings in this section are decoded from *SVE encodings* on page C4-399.

### Table C4-42 Encoding table for the SVE2 Integer - Predicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>SVE2 integer pairwise add and accumulate long on page C4-465</td>
</tr>
<tr>
<td>op1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0x000100</td>
<td>SVE2 integer unary operations (predicated) on page C4-465</td>
</tr>
</tbody>
</table>
### SVE2 integer pairwise add and accumulate long

This section describes the encoding of the SVE2 integer pairwise add and accumulate long instruction class. The encodings in this section are decoded from *SVE2 Integer - Predicated* on page C4-464.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>size</td>
<td>0 0 0 1 0</td>
<td>U</td>
</tr>
</tbody>
</table>

### SVE2 integer unary operations (predicated)

This section describes the encoding of the SVE2 integer unary operations (predicated) instruction class. The encodings in this section are decoded from *SVE2 Integer - Predicated* on page C4-464.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>size</td>
<td>0 0</td>
<td>Q</td>
</tr>
</tbody>
</table>

### Table C4-42 Encoding table for the SVE2 Integer - Predicated group (continued)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>      </td>
<td></td>
</tr>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>0xxx 0</td>
<td><em>SVE2 saturating/rounding bitwise shift left</em> (predicated) on page C4-466</td>
</tr>
<tr>
<td>10xx 0</td>
<td><em>SVE2 integer halving add/subtract</em> (predicated) on page C4-466</td>
</tr>
<tr>
<td>10xx 1</td>
<td><em>SVE2 integer pairwise arithmetic</em> on page C4-467</td>
</tr>
<tr>
<td>11xx 0</td>
<td><em>SVE2 saturating add/subtract</em> on page C4-467</td>
</tr>
<tr>
<td>11xx 1</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SADALP</td>
</tr>
<tr>
<td>1</td>
<td>UADALP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>opc</td>
</tr>
<tr>
<td>-</td>
<td>1x</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
</tbody>
</table>
SVE2 saturating/rounding bitwise shift left (predicated)

This section describes the encoding of the SVE2 saturating/rounding bitwise shift left (predicated) instruction class. The encodings in this section are decoded from SVE2 Integer - Predicated on page C4-464.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>size</td>
<td>0 0</td>
<td>Q</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q R N U</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>SRSHL</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>URSHL</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>SRSHLR</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>URSHLR</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>SQSHL (vectors)</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>UQSHL (vectors)</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>SQRSRL</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>UQRSRL</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>SQSHLR</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>UQSHLR</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>SQRSRL</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>UQRSRL</td>
</tr>
</tbody>
</table>

SVE2 integer halving add/subtract (predicated)

This section describes the encoding of the SVE2 integer halving add/subtract (predicated) instruction class. The encodings in this section are decoded from SVE2 Integer - Predicated on page C4-464.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>size</td>
<td>0 0</td>
<td>R</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>R S U</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>SHADD</td>
</tr>
<tr>
<td>0 0 1</td>
<td>UHADD</td>
</tr>
<tr>
<td>0 1 0</td>
<td>SHSUB</td>
</tr>
<tr>
<td>0 1 1</td>
<td>UHSUB</td>
</tr>
</tbody>
</table>
### SVE2 integer pairwise arithmetic

This section describes the encoding of the SVE2 integer pairwise arithmetic instruction class. The encodings in this section are decoded from *SVE2 Integer - Predicated* on page C4-464.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>size</td>
<td>0 1 0</td>
<td>opc</td>
<td>U</td>
<td>1 0 1</td>
<td>Pg</td>
<td>Zm</td>
</tr>
</tbody>
</table>

### SVE2 saturating add/subtract

This section describes the encoding of the SVE2 saturating add/subtract instruction class. The encodings in this section are decoded from *SVE2 Integer - Predicated* on page C4-464.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>size</td>
<td>0 1 1</td>
<td>op</td>
<td>S</td>
<td>U</td>
<td>1 0 0</td>
<td>Pg</td>
</tr>
</tbody>
</table>

---

#### Decode fields

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
<th>U</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SRHADD</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>URHADD</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>SHSUBR</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>UHSUBR</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>U</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>ADDP</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>SMAXP</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>UMAXP</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>SMINP</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>UMINP</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>U</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SQADD (vectors, predicated)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>UQADD (vectors, predicated)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SQSUB (vectors, predicated)</td>
</tr>
</tbody>
</table>
C4.1.42 SVE Multiply - Indexed

This section describes the encoding of the SVE Multiply - Indexed group. The encodings in this section are decoded from SVE encodings on page C4-399.

<table>
<thead>
<tr>
<th>[31]</th>
<th>24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15</th>
<th>10 9</th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000100</td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
<td>op0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C4-43 Encoding table for the SVE Multiply - Indexed group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>00000x</td>
<td>SVE integer dot product (indexed) on page C4-469</td>
</tr>
<tr>
<td>00001x</td>
<td>SVE2 integer multiply-add (indexed) on page C4-469</td>
</tr>
<tr>
<td>00010x</td>
<td>SVE2 saturating multiply-add high (indexed) on page C4-469</td>
</tr>
<tr>
<td>00011x</td>
<td>SVE mixed sign dot product (indexed) on page C4-470</td>
</tr>
<tr>
<td>001xxx</td>
<td>SVE2 saturating multiply-add (indexed) on page C4-470</td>
</tr>
<tr>
<td>0100xx</td>
<td>SVE2 complex integer dot product (indexed) on page C4-471</td>
</tr>
<tr>
<td>0101xx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0110xx</td>
<td>SVE2 complex integer multiply-add (indexed) on page C4-471</td>
</tr>
<tr>
<td>0111xx</td>
<td>SVE2 complex saturating multiply-add (indexed) on page C4-472</td>
</tr>
<tr>
<td>10xxxx</td>
<td>SVE2 integer multiply-add long (indexed) on page C4-472</td>
</tr>
<tr>
<td>110xxx</td>
<td>SVE2 integer multiply long (indexed) on page C4-473</td>
</tr>
<tr>
<td>1110xx</td>
<td>SVE2 saturating multiply (indexed) on page C4-474</td>
</tr>
<tr>
<td>11110x</td>
<td>SVE2 saturating multiply high (indexed) on page C4-474</td>
</tr>
<tr>
<td>111110</td>
<td>SVE2 integer multiply (indexed) on page C4-474</td>
</tr>
<tr>
<td>111111</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
**SVE integer dot product (indexed)**

This section describes the encoding of the SVE integer dot product (indexed) instruction class. The encodings in this section are decoded from *SVE Multiply - Indexed* on page C4-468.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 size 1 opc 0 0 0 0 0 U Zn Zda</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SVE2 integer multiply-add (indexed)**

This section describes the encoding of the SVE2 integer multiply-add (indexed) instruction class. The encodings in this section are decoded from *SVE Multiply - Indexed* on page C4-468.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 size 1 opc 0 0 0 0 1 S Zn Zda</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SVE2 saturating multiply-add high (indexed)**

This section describes the encoding of the SVE2 saturating multiply-add high (indexed) instruction class. The encodings in this section are decoded from *SVE Multiply - Indexed* on page C4-468.
This section describes the encoding of the SVE mixed sign dot product (indexed) instruction class. The encodings in this section are decoded from *SVE Multiply - Indexed* on page C4-468.

### SVE mixed sign dot product (indexed)

This section describes the encoding of the SVE mixed sign dot product (indexed) instruction class. The encodings in this section are decoded from *SVE Multiply - Indexed* on page C4-468.

### SVE2 saturating multiply-add (indexed)

This section describes the encoding of the SVE2 saturating multiply-add (indexed) instruction class. The encodings in this section are decoded from *SVE Multiply - Indexed* on page C4-468.
## SVE2 complex integer dot product (indexed)

This section describes the encoding of the SVE2 complex integer dot product (indexed) instruction class. The encodings in this section are decoded from *SVE Multiply - Indexed* on page C4-468.

<table>
<thead>
<tr>
<th>size</th>
<th>S</th>
<th>T</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0×</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>SQDMLALB (indexed) - Encoding</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>SQDMLALT (indexed) - Encoding</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>SQDMLSLB (indexed) - Encoding</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>SQDMLSLT (indexed) - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>SQDMLALB (indexed) - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>SQDMLALT (indexed) - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>SQDMLSLB (indexed) - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>SQDMLSLT (indexed) - Encoding</td>
</tr>
</tbody>
</table>

## SVE2 complex integer multiply-add (indexed)

This section describes the encoding of the SVE2 complex integer multiply-add (indexed) instruction class. The encodings in this section are decoded from *SVE Multiply - Indexed* on page C4-468.

<table>
<thead>
<tr>
<th>size</th>
<th>S</th>
<th>T</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0×</td>
<td></td>
<td></td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>CDOT (indexed) - Encoding</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>CDOT (indexed) - Encoding</td>
</tr>
</tbody>
</table>
### SVE2 complex saturating multiply-add (indexed)

This section describes the encoding of the SVE2 complex saturating multiply-add (indexed) instruction class. The encodings in this section are decoded from *SVE Multiply - Indexed* on page C4-468.

![Binary format with decode fields](image)

### SVE2 integer multiply-add long (indexed)

This section describes the encoding of the SVE2 integer multiply-add long (indexed) instruction class. The encodings in this section are decoded from *SVE Multiply - Indexed* on page C4-468.

![Binary format with decode fields](image)
### SVE2 integer multiply long (indexed)

This section describes the encoding of the SVE2 integer multiply long (indexed) instruction class. The encodings in this section are decoded from *SVE Multiply - Indexed* on page C4-468.

\[
\begin{array}{cccc|c|c|c}
31 & 30 & 29 & 28 \Vert 27 & 26 & 25 & 24 \Vert 23 & 22 & 21 & 20 & 16 \Vert 15 & 14 & 13 & 12 \Vert 11 & 10 & 9 & 5 & 4 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & \text{size} & 1 & \text{opc} & 1 & 1 & 0 & \text{U} & \text{T} & \text{Zn} & \text{Zd} & 0 \\
\end{array}
\]

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>size S U T</td>
<td></td>
</tr>
<tr>
<td>10 0 1 1</td>
<td>UMLALT (indexed) - Encoding</td>
</tr>
<tr>
<td>10 1 0 0</td>
<td>SMLSLB (indexed) - Encoding</td>
</tr>
<tr>
<td>10 1 0 1</td>
<td>SMLSLT (indexed) - Encoding</td>
</tr>
<tr>
<td>10 1 1 0</td>
<td>UMLSLB (indexed) - Encoding</td>
</tr>
<tr>
<td>10 1 1 1</td>
<td>UMLSLT (indexed) - Encoding</td>
</tr>
<tr>
<td>11 0 0 0</td>
<td>SMLALB (indexed) - Encoding</td>
</tr>
<tr>
<td>11 0 0 1</td>
<td>SMLALT (indexed) - Encoding</td>
</tr>
<tr>
<td>11 0 1 0</td>
<td>UMLALB (indexed) - Encoding</td>
</tr>
<tr>
<td>11 0 1 1</td>
<td>UMLALT (indexed) - Encoding</td>
</tr>
<tr>
<td>11 1 0 0</td>
<td>SMLSLB (indexed) - Encoding</td>
</tr>
<tr>
<td>11 1 0 1</td>
<td>SMLSLT (indexed) - Encoding</td>
</tr>
<tr>
<td>11 1 1 0</td>
<td>UMLSLB (indexed) - Encoding</td>
</tr>
<tr>
<td>11 1 1 1</td>
<td>UMLSLT (indexed) - Encoding</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>size U T</td>
<td></td>
</tr>
<tr>
<td>0x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10 0 0</td>
<td>SMULLB (indexed) - Encoding</td>
</tr>
<tr>
<td>10 0 1</td>
<td>SMULLT (indexed) - Encoding</td>
</tr>
<tr>
<td>10 1 0</td>
<td>UMULLB (indexed) - Encoding</td>
</tr>
<tr>
<td>10 1 1</td>
<td>UMULLT (indexed) - Encoding</td>
</tr>
<tr>
<td>11 0 0</td>
<td>SMULLB (indexed) - Encoding</td>
</tr>
<tr>
<td>11 0 1</td>
<td>SMULLT (indexed) - Encoding</td>
</tr>
<tr>
<td>11 1 0</td>
<td>UMULLB (indexed) - Encoding</td>
</tr>
<tr>
<td>11 1 1</td>
<td>UMULLT (indexed) - Encoding</td>
</tr>
</tbody>
</table>
SVE2 saturating multiply (indexed)

This section describes the encoding of the SVE2 saturating multiply (indexed) instruction class. The encodings in this section are decoded from *SVE Multiply - Indexed* on page C4-468.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>size</td>
<td>1</td>
<td>opc</td>
<td>1 1 1 0</td>
<td>R</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10</td>
<td>SQDMULLB (indexed) - Encoding</td>
</tr>
<tr>
<td>10</td>
<td>SQDMULLT (indexed) - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>SQDMULLB (indexed) - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>SQDMULLT (indexed) - Encoding</td>
</tr>
</tbody>
</table>

SVE2 saturating multiply high (indexed)

This section describes the encoding of the SVE2 saturating multiply high (indexed) instruction class. The encodings in this section are decoded from *SVE Multiply - Indexed* on page C4-468.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>size</td>
<td>1</td>
<td>opc</td>
<td>1 1 1 0</td>
<td>R</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>SQDMULH (indexed) - Encoding</td>
</tr>
<tr>
<td>0x</td>
<td>SQRDMULH (indexed) - Encoding</td>
</tr>
<tr>
<td>10</td>
<td>SQDMULH (indexed) - Encoding</td>
</tr>
<tr>
<td>10</td>
<td>SQRDMULH (indexed) - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>SQDMULH (indexed) - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>SQRDMULH (indexed) - Encoding</td>
</tr>
</tbody>
</table>

SVE2 integer multiply (indexed)

This section describes the encoding of the SVE2 integer multiply (indexed) instruction class. The encodings in this section are decoded from *SVE Multiply - Indexed* on page C4-468.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>size</td>
<td>1</td>
<td>opc</td>
<td>1 1 1 0</td>
<td>R</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>SQDMULH (indexed) - Encoding</td>
</tr>
<tr>
<td>0x</td>
<td>SQRDMULH (indexed) - Encoding</td>
</tr>
<tr>
<td>10</td>
<td>SQDMULH (indexed) - Encoding</td>
</tr>
<tr>
<td>10</td>
<td>SQRDMULH (indexed) - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>SQDMULH (indexed) - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>SQRDMULH (indexed) - Encoding</td>
</tr>
</tbody>
</table>
C4.1.43 SVE2 Widening Integer Arithmetic

This section describes the encoding of the SVE2 Widening Integer Arithmetic group. The encodings in this section are decoded from SVE encodings on page C4-399.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0</td>
<td>1 0 0 1 0</td>
<td>1 1 1 1 0</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
</tr>
</tbody>
</table>

Table C4-44 Encoding table for the SVE2 Widening Integer Arithmetic group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0x</td>
<td>SVE2 integer add/subtract long on page C4-475</td>
</tr>
<tr>
<td>10</td>
<td>SVE2 integer add/subtract wide on page C4-476</td>
</tr>
<tr>
<td>11</td>
<td>SVE2 integer multiply long on page C4-476</td>
</tr>
</tbody>
</table>

SVE2 integer add/subtract long

This section describes the encoding of the SVE2 integer add/subtract long instruction class. The encodings in this section are decoded from SVE2 Widening Integer Arithmetic.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0</td>
<td>0 0 1 0</td>
<td>size 0</td>
<td>Zn 0 0 op S U T</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
</tr>
</tbody>
</table>

Decide fields

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>U</th>
<th>T</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>SADDLB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>SADDLT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>UADDLB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>UADDLT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This section describes the encoding of the SVE2 integer add/subtract wide instruction class. The encodings in this section are decoded from *SVE2 Widening Integer Arithmetic* on page C4-475.

```markdown
<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>U</th>
<th>T</th>
</tr>
</thead>
</table>
| 0  | 0 | 0 | 0 | SSUBLB
| 1  | 0 | 0 | 0 | SADDWB
| 1  | 0 | 0 | 1 | SADDWT
| 1  | 0 | 1 | 0 | SSUBWB
| 1  | 0 | 1 | 1 | SSUBWT
| 0  | 0 | 0 | 0 | Unallocated.
```

This section describes the encoding of the SVE2 integer multiply long instruction class. The encodings in this section are decoded from *SVE2 Widening Integer Arithmetic* on page C4-475.

```markdown
<table>
<thead>
<tr>
<th>S</th>
<th>U</th>
<th>T</th>
</tr>
</thead>
</table>
| 0 | 0 | 0 | SADDWB
| 0 | 0 | 1 | SADDWT
| 0 | 1 | 0 | UADDWB
| 0 | 1 | 1 | UADDWT
| 1 | 0 | 0 | SSUBWB
| 1 | 0 | 1 | SSUBWT
| 1 | 1 | 0 | USUBWB
| 1 | 1 | 1 | USUBWT
```
C4.1.44 SVE Misc

This section describes the encoding of the SVE Misc group. The encodings in this section are decoded from SVE encodings on page C4-399.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>U</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table C4-45 Encoding table for the SVE Misc group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
</tr>
<tr>
<td>0</td>
<td>10xx</td>
</tr>
<tr>
<td>1</td>
<td>10xx</td>
</tr>
<tr>
<td>-</td>
<td>00xx</td>
</tr>
<tr>
<td>-</td>
<td>010x</td>
</tr>
<tr>
<td>-</td>
<td>0110</td>
</tr>
<tr>
<td>-</td>
<td>0111</td>
</tr>
<tr>
<td>-</td>
<td>11xx</td>
</tr>
</tbody>
</table>

SVE2 bitwise shift left long

This section describes the encoding of the SVE2 bitwise shift left long instruction class. The encodings in this section are decoded from SVE Misc.
### SVE2 integer add/subtract interleaved long

This section describes the encoding of the SVE2 integer add/subtract interleaved long instruction class. The encodings in this section are decoded from *SVE Misc on page C4-477.*

![Instruction Encoding](image)

**Decode fields**

<table>
<thead>
<tr>
<th>S</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Instruction page**

- **SSHLLB**
- **SSHLLT**
- **USHLLB**
- **USHLLT**

### SVE2 bitwise exclusive-or interleaved

This section describes the encoding of the SVE2 bitwise exclusive-or interleaved instruction class. The encodings in this section are decoded from *SVE Misc on page C4-477.*

![Instruction Encoding](image)

**Decode fields**

<table>
<thead>
<tr>
<th>S</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Instruction page**

- **SADDLBT**
- **Unallocated.**
- **SSUBLBT**
- **SSUBLTB**

### SVE2 bitwise exclusive-or interleaved

This section describes the encoding of the SVE2 bitwise exclusive-or interleaved instruction class. The encodings in this section are decoded from *SVE Misc on page C4-477.*

![Instruction Encoding](image)

**Decode fields**

| T  | 0 | 1 |

**Instruction page**

- **EORBT**
- **EORTB**
SVE integer matrix multiply accumulate

This section describes the encoding of the SVE integer matrix multiply accumulate instruction class. The encodings in this section are decoded from *SVE Misc on page C4-477*.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1</td>
<td>uns 0</td>
<td>Zm 1 0 0 1 1 0</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

### Decode fields

<table>
<thead>
<tr>
<th>uns</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SMMLA</td>
<td>FEAT_I8MM</td>
</tr>
<tr>
<td>01</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>USMMLA</td>
<td>FEAT_I8MM</td>
</tr>
<tr>
<td>11</td>
<td>UMMLA</td>
<td>FEAT_I8MM</td>
</tr>
</tbody>
</table>

SVE2 bitwise permute

This section describes the encoding of the SVE2 bitwise permute instruction class. The encodings in this section are decoded from *SVE Misc on page C4-477*.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1</td>
<td>size 0</td>
<td>Zm 1 0 1 1</td>
<td>opc</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>BEXT</td>
<td>FEAT_SVE_BitPerm</td>
</tr>
<tr>
<td>01</td>
<td>BDEP</td>
<td>FEAT_SVE_BitPerm</td>
</tr>
<tr>
<td>10</td>
<td>BGRP</td>
<td>FEAT_SVE_BitPerm</td>
</tr>
<tr>
<td>11</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

C4.1.45 SVE2 Accumulate

This section describes the encoding of the SVE2 Accumulate group. The encodings in this section are decoded from *SVE encodings on page C4-399*. 
C4.1 A64 instruction set encoding

SVE2 complex integer add

This section describes the encoding of the SVE2 complex integer add instruction class. The encodings in this section are decoded from SVE2 Accumulate on page C4-479.

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 | 9 | 8 | 7 | 6 | 5 | 4 | 0 |
|-------------|-------------|-------------|-------------|-------------|-----|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 1 0 0 0 0 1 0 1 | 0 0 0 0 0 | 0 1 1 0 1 | Zm | Zdn |

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>CADD</td>
</tr>
<tr>
<td>1</td>
<td>SQCADD</td>
</tr>
</tbody>
</table>

SVE2 integer absolute difference and accumulate long

This section describes the encoding of the SVE2 integer absolute difference and accumulate long instruction class. The encodings in this section are decoded from SVE2 Accumulate on page C4-479.

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 | 9 | 8 | 7 | 6 | 5 | 4 | 0 |
|-------------|-------------|-------------|-------------|-------------|-----|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 1 0 0 0 0 1 0 1 | 0 0 0 0 0 | 0 1 1 0 1 | Zm | Zdn |

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>CADD</td>
</tr>
<tr>
<td>1</td>
<td>SQCADD</td>
</tr>
</tbody>
</table>
SVE2 integer add/subtract long with carry

This section describes the encoding of the SVE2 integer add/subtract long with carry instruction class. The encodings in this section are decoded from SVE2 Accumulate on page C4-479.

SVE2 bitwise shift right and accumulate

This section describes the encoding of the SVE2 bitwise shift right and accumulate instruction class. The encodings in this section are decoded from SVE2 Accumulate on page C4-479.
### SVE2 bitwise shift and insert

This section describes the encoding of the SVE2 bitwise shift and insert instruction class. The encodings in this section are decoded from *SVE2 Accumulate on page C4-479*.

![Instruction Set Encoding](image)

#### Decode fields

<table>
<thead>
<tr>
<th>R</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### SVE2 integer absolute difference and accumulate

This section describes the encoding of the SVE2 integer absolute difference and accumulate instruction class. The encodings in this section are decoded from *SVE2 Accumulate on page C4-479*.

![Instruction Set Encoding](image)

#### Decode fields

<table>
<thead>
<tr>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

### SVE2 integer absolute difference and accumulate

This section describes the encoding of the SVE2 integer absolute difference and accumulate instruction class. The encodings in this section are decoded from *SVE2 Accumulate on page C4-479*.

![Instruction Set Encoding](image)

#### Decode fields

<table>
<thead>
<tr>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
C4.1.46 SVE2 Narrowing

This section describes the encoding of the SVE2 Narrowing group. The encodings in this section are decoded from SVE encodings on page C4-399.

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23 22 21</th>
<th>19 18</th>
<th>16</th>
<th>15 14 13 12</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000101</td>
<td>1</td>
<td>op1</td>
<td>0</td>
<td>op2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C4-47 Encoding table for the SVE2 Narrowing group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2</td>
<td></td>
</tr>
<tr>
<td>0 0 0 10</td>
<td>SVE2 saturating extract narrow on page C4-483</td>
</tr>
<tr>
<td>0 != 0 0 10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 - 0x</td>
<td>SVE2 bitwise shift right narrow on page C4-484</td>
</tr>
<tr>
<td>1 - 0x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 - 10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- - 11</td>
<td>SVE2 integer add/subtract narrow high part on page C4-484</td>
</tr>
</tbody>
</table>

SVE2 saturating extract narrow

This section describes the encoding of the SVE2 saturating extract narrow instruction class. The encodings in this section are decoded from SVE2 Narrowing.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1 0</td>
<td>1 tszl</td>
<td>0 0 0 0 1 0</td>
<td>opc T</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

tszl

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc T</td>
<td></td>
</tr>
<tr>
<td>00 0</td>
<td>SQXTNB</td>
</tr>
<tr>
<td>00 1</td>
<td>SQXTNT</td>
</tr>
<tr>
<td>01 0</td>
<td>UQXTNB</td>
</tr>
<tr>
<td>01 1</td>
<td>UQXTNT</td>
</tr>
<tr>
<td>10 0</td>
<td>SQXTUNB</td>
</tr>
<tr>
<td>10 1</td>
<td>SQXTUNT</td>
</tr>
<tr>
<td>11 -</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
SVE2 bitwise shift right narrow

This section describes the encoding of the SVE2 bitwise shift right narrow instruction class. The encodings in this section are decoded from *SVE2 Narrowing* on page C4-483.

```
SVE2 bitwise shift right narrow
```

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18</th>
<th>16 15 14 13</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1</td>
<td>1</td>
<td>tszl</td>
<td>imm3</td>
<td>0 0</td>
<td>op</td>
<td>U</td>
<td>R</td>
<td>T</td>
</tr>
</tbody>
</table>

SVE2 integer add/subtract narrow high part

This section describes the encoding of the SVE2 integer add/subtract narrow high part instruction class. The encodings in this section are decoded from *SVE2 Narrowing* on page C4-483.
C4.1.47 SVE2 Histogram Computation - Segment

This section describes the encoding of the SVE2 Histogram Computation - Segment group. The encodings in this section are decoded from SVE encodings on page C4-399.

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[16 15 14 13]</th>
<th>[12 11 10 9]</th>
<th>[5 4]</th>
<th>[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C4-48 Encoding table for the SVE2 Histogram Computation - Segment group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>HISTSEG</td>
</tr>
<tr>
<td>!0 = 000</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

C4.1.48 SVE2 Crypto Extensions

This section describes the encoding of the SVE2 Crypto Extensions group. The encodings in this section are decoded from SVE encodings on page C4-399.
SVE2 crypto unary operations

This section describes the encoding of the SVE2 crypto unary operations instruction class. The encodings in this section are decoded from *SVE2 Crypto Extensions on page C4-485*.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 0 |
|-----------------------------------------------|-------------|
| 0 1 0 0 0 1 0 1 | size | 1 0 0 0 0 0 1 1 1 0 0 | op | 0 0 0 0 0 | Zdn |
```

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00</td>
<td>00</td>
<td>00000</td>
<td><em>SVE2 crypto unary operations</em></td>
</tr>
<tr>
<td>000</td>
<td>00</td>
<td>00</td>
<td>!= 00000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>000</td>
<td>00</td>
<td>x1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>000</td>
<td>01</td>
<td>0x</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>000</td>
<td>01</td>
<td>11</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>00</td>
<td>-</td>
<td><em>SVE2 crypto destructive binary operations</em></td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>x1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>!= 000</td>
<td>-</td>
<td>0x</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>!= 000</td>
<td>-</td>
<td>11</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SVE2 crypto destructive binary operations

This section describes the encoding of the SVE2 crypto destructive binary operations instruction class. The encodings in this section are decoded from *SVE2 Crypto Extensions on page C4-485*.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 0 |
|-----------------------------------------------|-------------|
| 0 1 0 0 0 1 0 1 | size | 1 0 0 0 0 0 1 1 1 0 0 | op | 0 0 0 0 0 | Zdn |
```

<table>
<thead>
<tr>
<th>size</th>
<th>op</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>AESMC</td>
<td>FEAT_SVE_AES</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>AESIMC</td>
<td>FEAT_SVE_AES</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1x</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
## SVE2 crypto constructive binary operations

This section describes the encoding of the SVE2 crypto constructive binary operations instruction class. The encodings in this section are decoded from *SVE2 Crypto Extensions* on page C4-485.

### C4.1.49 SVE Floating Point Widening Multiply-Add - Indexed

This section describes the encoding of the SVE Floating Point Widening Multiply-Add - Indexed group. The encodings in this section are decoded from *SVE encodings* on page C4-399.
SVE BFloat16 floating-point dot product (indexed)

This section describes the encoding of the SVE BFloat16 floating-point dot product (indexed) instruction class. The encodings in this section are decoded from SVE Floating Point Widening Multiply-Add - Indexed on page C4-487.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SVE BFloat16 floating-point dot product (indexed)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SVE floating-point multiply-add long (indexed)

This section describes the encoding of the SVE floating-point multiply-add long (indexed) instruction class. The encodings in this section are decoded from SVE Floating Point Widening Multiply-Add - Indexed on page C4-487.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>BFDOT (indexed)</td>
<td>FEAT_BF16</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>FMLALT (indexed)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>FMLSLB (indexed)</td>
</tr>
</tbody>
</table>

Table C4-50 Encoding table for the SVE Floating Point Widening Multiply-Add - Indexed group
C4.1.50 SVE Floating Point Widening Multiply-Add

This section describes the encoding of the SVE Floating Point Widening Multiply-Add group. The encodings in this section are decoded from SVE encodings on page C4-399.

Table C4-51 Encoding table for the SVE Floating Point Widening Multiply-Add group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>SVE BFloat16 floating-point dot product on page C4-489</td>
</tr>
<tr>
<td>0 0 1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>SVE floating-point multiply-add long on page C4-490</td>
</tr>
</tbody>
</table>

SVE BFloat16 floating-point dot product

This section describes the encoding of the SVE BFloat16 floating-point dot product instruction class. The encodings in this section are decoded from SVE Floating Point Widening Multiply-Add.

Table C4-51 Encoding table for the SVE Floating Point Widening Multiply-Add group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>BFDOT (vectors)</td>
<td>FEAT_BF16</td>
</tr>
</tbody>
</table>
SVE floating-point multiply-add long

This section describes the encoding of the SVE floating-point multiply-add long instruction class. The encodings in this section are decoded from *SVE Floating Point Widening Multiply-Add* on page C4-489.

```
[31 30 29 28 27 26 25 24 23 22 21 20 | 16 15 14 13 12 | 11 10 | 9 | 5 | 4 | 0 ]
0 1 1 0 0 1 0 0 1 0 2 1 Zm 1 0 op 0 0 T Zn Zda
```

Table C4-52 Encoding table for the SVE Floating Point Arithmetic - Predicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>o2 op T</td>
<td>FMLALB (vectors)</td>
<td>-</td>
</tr>
<tr>
<td>0 0 0</td>
<td>FMLALT (vectors)</td>
<td>-</td>
</tr>
<tr>
<td>0 0 1</td>
<td>FMLSLB (vectors)</td>
<td>-</td>
</tr>
<tr>
<td>0 1 0</td>
<td>FMLSLT (vectors)</td>
<td>-</td>
</tr>
<tr>
<td>1 0 0</td>
<td>BFMLALB (vectors) FEAT_BF16</td>
<td></td>
</tr>
<tr>
<td>1 0 1</td>
<td>BFMLALT (vectors) FEAT_BF16</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

C4.1.51 SVE Floating Point Arithmetic - Predicated

This section describes the encoding of the SVE Floating Point Arithmetic - Predicated group. The encodings in this section are decoded from *SVE encodings* on page C4-399.

```
[31 | 24 23 22 21 20 | 19 18 | 16 15 | 13 12 | 10 9 | 6 5 | 0 ]
01100101 0 op0 100 op1 op2
```

Table C4-52 Encoding table for the SVE Floating Point Arithmetic - Predicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2</td>
<td></td>
</tr>
<tr>
<td>0x - -</td>
<td><em>SVE floating-point arithmetic (predicated)</em> on page C4-490</td>
</tr>
<tr>
<td>10 000 -</td>
<td>FTMAD</td>
</tr>
<tr>
<td>10 != 000 -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11 - 0000</td>
<td><em>SVE floating-point arithmetic with immediate (predicated)</em> on page C4-491</td>
</tr>
<tr>
<td>11 - != 0000</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SVE floating-point arithmetic (predicated)

This section describes the encoding of the SVE floating-point arithmetic (predicated) instruction class. The encodings in this section are decoded from *SVE Floating Point Arithmetic - Predicated.*
### SVE floating-point arithmetic with immediate (predicated)

This section describes the encoding of the SVE floating-point arithmetic with immediate (predicated) instruction class. The encodings in this section are decoded from *SVE Floating Point Arithmetic - Predicated* on page C4-490.

<table>
<thead>
<tr>
<th>Size</th>
<th>opc</th>
<th>Pg</th>
<th>Zm</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>01100101</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>i1</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>FADD (immediate)</td>
</tr>
<tr>
<td>001</td>
<td>FSUB (immediate)</td>
</tr>
<tr>
<td>010</td>
<td>FMUL (immediate)</td>
</tr>
<tr>
<td>011</td>
<td>FSUBR (immediate)</td>
</tr>
<tr>
<td>0100</td>
<td>FMAXNM (vectors)</td>
</tr>
<tr>
<td>0101</td>
<td>FMINNM (vectors)</td>
</tr>
<tr>
<td>0110</td>
<td>FMAX (vectors)</td>
</tr>
<tr>
<td>0111</td>
<td>FMIN (vectors)</td>
</tr>
<tr>
<td>1000</td>
<td>FABD</td>
</tr>
<tr>
<td>1001</td>
<td>FSSCALE</td>
</tr>
<tr>
<td>1010</td>
<td>FMULX</td>
</tr>
<tr>
<td>1011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1100</td>
<td>FDIVR</td>
</tr>
<tr>
<td>1101</td>
<td>FDIV</td>
</tr>
<tr>
<td>111x</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
C4.1.52 SVE Floating Point Unary Operations - Predicated

This section describes the encoding of the SVE Floating Point Unary Operations - Predicated group. The encodings in this section are decoded from SVE encodings on page C4-399.

Table C4-53 Encoding table for the SVE Floating Point Unary Operations - Predicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>00x</td>
<td>SVE floating-point round to integral value on page C4-492</td>
</tr>
<tr>
<td>010</td>
<td>SVE floating-point convert precision on page C4-493</td>
</tr>
<tr>
<td>011</td>
<td>SVE floating-point unary operations on page C4-493</td>
</tr>
<tr>
<td>10x</td>
<td>SVE integer convert to floating-point on page C4-494</td>
</tr>
<tr>
<td>11x</td>
<td>SVE floating-point convert to integer on page C4-495</td>
</tr>
</tbody>
</table>

SVE floating-point round to integral value

This section describes the encoding of the SVE floating-point round to integral value instruction class. The encodings in this section are decoded from SVE Floating Point Unary Operations - Predicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>FRINT&lt;r&gt; - Encoding</td>
</tr>
<tr>
<td>001</td>
<td>FRINT&lt;r&gt; - Encoding</td>
</tr>
<tr>
<td>010</td>
<td>FRINT&lt;r&gt; - Encoding</td>
</tr>
<tr>
<td>011</td>
<td>FRINT&lt;r&gt; - Encoding</td>
</tr>
</tbody>
</table>
SVE floating-point convert precision

This section describes the encoding of the SVE floating-point convert precision instruction class. The encodings in this section are decoded from *SVE Floating Point Unary Operations - Predicated* on page C4-492.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>0 0 1 0</td>
<td>opc</td>
<td>opc2</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
</tr>
</tbody>
</table>

SVE floating-point unary operations

This section describes the encoding of the SVE floating-point unary operations instruction class. The encodings in this section are decoded from *SVE Floating Point Unary Operations - Predicated* on page C4-492.
SVE integer convert to floating-point

This section describes the encoding of the SVE integer convert to floating-point instruction class. The encodings in this section are decoded from *SVE Floating Point Unary Operations - Predicated* on page C4-492.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>opc 0 1 1</td>
<td>Pg Zn Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SVE integer convert to floating-point**

This section describes the encoding of the SVE integer convert to floating-point instruction class. The encodings in this section are decoded from *SVE Floating Point Unary Operations - Predicated* on page C4-492.
### SVE floating-point convert to integer

This section describes the encoding of the SVE floating-point convert to integer instruction class. The encodings in this section are decoded from *SVE Floating Point Unary Operations - Predicated* on page C4-492.

<table>
<thead>
<tr>
<th>opc</th>
<th>opc2</th>
<th>U</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
<td>1</td>
<td>UCVTF - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>0</td>
<td>SCVTTF - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1</td>
<td>UCVTF - Encoding</td>
</tr>
</tbody>
</table>

![Hexadecimal encoding](image)

<table>
<thead>
<tr>
<th>opc</th>
<th>opc2</th>
<th>U</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>FLOGB</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>0</td>
<td>FCVTZS - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>1</td>
<td>FCVTZU - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>0</td>
<td>FCVTZS - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>1</td>
<td>FCVTZU - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>0</td>
<td>FCVTZS - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>1</td>
<td>FCVTZU - Encoding</td>
</tr>
<tr>
<td>10</td>
<td>0x</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>FCVTZS - Encoding</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>FCVTZU - Encoding</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>0</td>
<td>FCVTZS - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>1</td>
<td>FCVTZU - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>0</td>
<td>FCVTZS - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>1</td>
<td>FCVTZU - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>0</td>
<td>FCVTZS - Encoding</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1</td>
<td>FCVTZU - Encoding</td>
</tr>
</tbody>
</table>
C4.1.53 SVE Floating Point Unary Operations - Unpredicated

This section describes the encoding of the SVE Floating Point Unary Operations - Unpredicated group. The encodings in this section are decoded from SVE encodings on page C4-399.

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23 22 21</th>
<th>19 18</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11 10 9</th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>01100101</td>
<td>001</td>
<td>0011</td>
<td>op0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C4-54 Encoding table for the SVE Floating Point Unary Operations - Unpredicated group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>= 00</td>
<td>SVE floating-point reciprocal estimate (unpredicated) on page C4-496</td>
</tr>
<tr>
<td>!= 00</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SVE floating-point reciprocal estimate (unpredicated)

This section describes the encoding of the SVE floating-point reciprocal estimate (unpredicated) instruction class. The encodings in this section are decoded from SVE Floating Point Unary Operations - Unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>size 0 0 1</td>
<td>opc 0 0 1 1 0 0</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>0xx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>110</td>
<td>FRECPE</td>
</tr>
<tr>
<td>111</td>
<td>FRSQRTF</td>
</tr>
</tbody>
</table>

C4.1.54 SVE Floating Point Compare - with Zero

This section describes the encoding of the SVE Floating Point Compare - with Zero group. The encodings in this section are decoded from SVE encodings on page C4-399.
SVE floating-point compare with zero

This section describes the encoding of the SVE floating-point compare with zero instruction class. The encodings in this section are decoded from SVE Floating Point Compare - with Zero on page C4-496.

<table>
<thead>
<tr>
<th>op0</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SVE floating-point compare with zero</td>
</tr>
<tr>
<td>1</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

Table C4-55 Encoding table for the SVE Floating Point Compare - with Zero group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>lt</td>
</tr>
<tr>
<td>0 0 0</td>
<td>FCM&lt;cc&gt; (zero) - Encoding</td>
</tr>
<tr>
<td>0 0 1</td>
<td>FCM&lt;cc&gt; (zero) - Encoding</td>
</tr>
<tr>
<td>0 1 0</td>
<td>FCM&lt;cc&gt; (zero) - Encoding</td>
</tr>
<tr>
<td>0 1 1</td>
<td>FCM&lt;cc&gt; (zero) - Encoding</td>
</tr>
<tr>
<td>1 1 1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 0 0</td>
<td>FCM&lt;cc&gt; (zero) - Encoding</td>
</tr>
<tr>
<td>1 1 0</td>
<td>FCM&lt;cc&gt; (zero) - Encoding</td>
</tr>
</tbody>
</table>

C4.1.55 SVE Floating Point Accumulating Reduction

This section describes the encoding of the SVE Floating Point Accumulating Reduction group. The encodings in this section are decoded from SVE encodings on page C4-399.
This section describes the encoding of the SVE floating-point serial reduction (predicated) instruction class. The encodings in this section are decoded from SVE Floating Point Accumulating Reduction on page C4-497.

Table C4-56 Encoding table for the SVE Floating Point Accumulating Reduction group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SVE floating-point serial reduction (predicated)</td>
</tr>
<tr>
<td>1</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

SVE floating-point serial reduction (predicated)

This section describes the encoding of the SVE floating-point serial reduction (predicated) instruction class. The encodings in this section are decoded from SVE Floating Point Accumulating Reduction on page C4-497.

Table C4-57 Encoding table for the SVE Floating Point Multiply-Add group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>FADDA</td>
</tr>
<tr>
<td>01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1x</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

C4.1.56      SVE Floating Point Multiply-Add

This section describes the encoding of the SVE Floating Point Multiply-Add group. The encodings in this section are decoded from SVE encodings on page C4-399.

Table C4-57 Encoding table for the SVE Floating Point Multiply-Add group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SVE floating-point multiply-accumulate writing addend on page C4-499</td>
</tr>
<tr>
<td>1</td>
<td>SVE floating-point multiply-accumulate writing multiplicand on page C4-499</td>
</tr>
</tbody>
</table>
SVE floating-point multiply-accumulate writing addend

This section describes the encoding of the SVE floating-point multiply-accumulate writing addend instruction class. The encodings in this section are decoded from SVE Floating Point Multiply-Add on page C4-498.

| 0 1 1 0 0 1 0 1 | size  | Zm | 0 | opc | Pg | Zn | Zda |

---

**SVE floating-point multiply-accumulate writing multiplicand**

This section describes the encoding of the SVE floating-point multiply-accumulate writing multiplicand instruction class. The encodings in this section are decoded from SVE Floating Point Multiply-Add on page C4-498.

| 0 1 1 0 0 1 0 1 | size  | Za | 1 | opc | Pg | Zm | Zdn |

---

**C4.1.57 SVE Memory - 32-bit Gather and Unsized Contiguous**

This section describes the encoding of the SVE Memory - 32-bit Gather and Unsized Contiguous group. The encodings in this section are decoded from SVE encodings on page C4-399.
### Table C4-58 Encoding table for the SVE Memory - 32-bit Gather and Unsized Contiguous group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
</tr>
</tbody>
</table>
| 00 | x1 | 0xx | 0 | SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets) on page C4-500  
| 00 | x1 | 0xx | 1 | Unallocated.  
| 01 | x1 | 0xx | - | SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets) on page C4-501  
| 10 | x1 | 0xx | - | SVE 32-bit gather load words (scalar plus 32-bit scaled offsets) on page C4-501  
| 11 | 0x | 000 | 0 | LDR (predicate)  
| 11 | 0x | 000 | 1 | Unallocated.  
| 11 | 0x | 010 | - | LDR (vector)  
| 11 | 0x | 0x1 | - | Unallocated.  
| 11 | 1x | 0xx | 0 | SVE contiguous prefetch (scalar plus immediate) on page C4-502  
| 11 | 1x | 0xx | 1 | Unallocated.  
| != 11 | x0 | 0xx | - | SVE 32-bit gather load (scalar plus 32-bit unscaled offsets) on page C4-502  
| - | 00 | 10x | - | SVE2 32-bit gather non-temporal load (vector plus scalar) on page C4-503  
| - | 00 | 110 | 0 | SVE contiguous prefetch (scalar plus scalar) on page C4-503  
| - | 00 | 111 | 0 | SVE 32-bit gather prefetch (vector plus immediate) on page C4-504  
| - | 00 | 11x | 1 | Unallocated.  
| - | 01 | 1xx | - | SVE 32-bit gather load (vector plus immediate) on page C4-504  
| - | 1x | 1xx | - | SVE load and broadcast element on page C4-505  

### SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)

This section describes the encoding of the SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets) instruction class. The encodings in this section are decoded from *SVE Memory - 32-bit Gather and Unsized Contiguous* on page C4-499.
SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)

This section describes the encoding of the SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets) instruction class. The encodings in this section are decoded from *SVE Memory - 32-bit Gather and Unsized Contiguous* on page C4-499.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 0 xs 1 Zm 0 msz Pg Rn 0 prfop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Decode fields**

<table>
<thead>
<tr>
<th></th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>PRFB (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>PRFH (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>PRFW (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>PRFD (scalar plus vector)</td>
</tr>
</tbody>
</table>

SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)

This section describes the encoding of the SVE 32-bit gather load words (scalar plus 32-bit scaled offsets) instruction class. The encodings in this section are decoded from *SVE Memory - 32-bit Gather and Unsized Contiguous* on page C4-499.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 1 xs 1 Zm 0 U ff Pg Rn Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Decode fields**

<table>
<thead>
<tr>
<th></th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>0 1</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>1 0</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>1 1</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
</tbody>
</table>
SVE contiguous prefetch (scalar plus immediate)

This section describes the encoding of the SVE contiguous prefetch (scalar plus immediate) instruction class. The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsize Contiguous on page C4-499.

SVE 32-bit gather load (scalar plus 32-bit unscaled offsets)

This section describes the encoding of the SVE 32-bit gather load (scalar plus 32-bit unscaled offsets) instruction class. The encodings in this section are decoded from SVE Memory - 32-bit Gather and Unsize Contiguous on page C4-499.
SVE2 32-bit gather non-temporal load (vector plus scalar)

This section describes the encoding of the SVE2 32-bit gather non-temporal load (vector plus scalar) instruction class. The encodings in this section are decoded from *SVE Memory - 32-bit Gather and Unsized Contiguous* on page C4-499.

```
<table>
<thead>
<tr>
<th>opc</th>
<th>U</th>
<th>ff</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

SVE contiguous prefetch (scalar plus scalar)

This section describes the encoding of the SVE contiguous prefetch (scalar plus scalar) instruction class. The encodings in this section are decoded from *SVE Memory - 32-bit Gather and Unsized Contiguous* on page C4-499.

```
<table>
<thead>
<tr>
<th>msz</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
</tr>
</tbody>
</table>
```
SVE 32-bit gather prefetch (vector plus immediate)

This section describes the encoding of the SVE 32-bit gather prefetch (vector plus immediate) instruction class. The encodings in this section are decoded from *SVE Memory - 32-bit Gather and Unsized Contiguous* on page C4-499.

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20]</th>
<th>16 15 14 13 12</th>
<th>10 9 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 msz 0 0</td>
<td>Rm 1 1 0</td>
<td>Pg Zn 0 prfop</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>PRFB (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>PRFH (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>PRFW (scalar plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>PRFD (scalar plus scalar)</td>
</tr>
</tbody>
</table>

SVE 32-bit gather load (vector plus immediate)

This section describes the encoding of the SVE 32-bit gather load (vector plus immediate) instruction class. The encodings in this section are decoded from *SVE Memory - 32-bit Gather and Unsized Contiguous* on page C4-499.

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20]</th>
<th>16 15 14 13 12</th>
<th>10 9 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 msz 0 0</td>
<td>imm5 1 1</td>
<td>Pg Zn 0 prfop</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>PRFB (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>PRFH (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>PRFW (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>PRFD (vector plus immediate)</td>
</tr>
</tbody>
</table>
This section describes the encoding of the SVE load and broadcast element instruction class. The encodings in this section are decoded from *SVE Memory - 32-bit Gather and Unsized Contiguous* on page C4-499.

<table>
<thead>
<tr>
<th>1 0 0 0 0 1 0 msz 0 1 imm5</th>
<th>1 1 U ff</th>
<th>Pg</th>
<th>Zn</th>
<th>Zt</th>
</tr>
</thead>
</table>

**SVE load and broadcast element**

The encodings in the instruction class are as follows:

- **LD1SB (vector plus immediate)**: 00 0 0
- **LDFF1SB (vector plus immediate)**: 00 0 1
- **LD1B (vector plus immediate)**: 00 1 0
- **LDFF1B (vector plus immediate)**: 00 1 1
- **LD1SH (vector plus immediate)**: 01 0 0
- **LDFF1SH (vector plus immediate)**: 01 0 1
- **LD1H (vector plus immediate)**: 01 1 0
- **LDFF1H (vector plus immediate)**: 01 1 1
- **LD1W (vector plus immediate)**: 10 0 - Unallocated.
- **LDFF1W (vector plus immediate)**: 10 1 - Unallocated.
- **LD1RB - Encoding**: 00 0 0
- **LD1RB - Encoding**: 00 0 1
- **LD1RB - Encoding**: 00 1 0
- **LD1RB - Encoding**: 00 1 1
- **LD1RSW**: 01 0 0
- **LD1RH - Encoding**: 01 0 1
- **LD1RH - Encoding**: 01 1 0

```
[31 30 29 28|27 26 25 24|23 22 21] 16|15 14 13 12| 10 9 | 5 4 | 0 |
1 0 0 0 0 1 0 msz 0 1 imm6 1 dtype| Rn | Zt |

```

**Decode fields**

<table>
<thead>
<tr>
<th>msz</th>
<th>U</th>
<th>ff</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>LD1SB (vector plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>LDFF1SB (vector plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>LD1B (vector plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>LDFF1B (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>LD1SH (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>LDFF1SH (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>LD1H (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDFF1H (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>LD1W (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDFF1W (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

```

**Decode fields**

<table>
<thead>
<tr>
<th>dtype</th>
<th>dtype</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>LD1RB - Encoding</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>LD1RB - Encoding</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>LD1RB - Encoding</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>LD1RB - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>LD1RSW</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>LD1RH - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>LD1RH - Encoding</td>
</tr>
</tbody>
</table>

```

**Decode fields**

<table>
<thead>
<tr>
<th>dtypeh</th>
<th>dtype</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>LD1RB - Encoding</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>LD1RB - Encoding</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>LD1RB - Encoding</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>LD1RB - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>LD1RSW</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>LD1RH - Encoding</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>LD1RH - Encoding</td>
</tr>
</tbody>
</table>
C4.1.58  SVE Memory - Contiguous Load

This section describes the encoding of the SVE Memory - Contiguous Load group. The encodings in this section are decoded from SVE encodings on page C4-399.

<table>
<thead>
<tr>
<th>op0 op1 op2</th>
<th>SVE contiguous non-temporal load (scalar plus immediate) on page C4-507</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 111</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>SVE contiguous non-temporal load (scalar plus scalar) on page C4-507</td>
</tr>
<tr>
<td>!= 00 0 111</td>
<td>SVE load multiple structures (scalar plus immediate) on page C4-508</td>
</tr>
<tr>
<td>!= 00</td>
<td>SVE load multiple structures (scalar plus scalar) on page C4-508</td>
</tr>
<tr>
<td>-</td>
<td>SVE load and broadcast quadword (scalar plus immediate) on page C4-509</td>
</tr>
<tr>
<td>-</td>
<td>SVE contiguous load (scalar plus immediate) on page C4-509</td>
</tr>
<tr>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>SVE contiguous non-fault load (scalar plus immediate) on page C4-510</td>
</tr>
<tr>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- 0 001</td>
<td>SVE load and broadcast quadword (scalar plus scalar) on page C4-511</td>
</tr>
</tbody>
</table>

Table C4-59 Encoding table for the SVE Memory - Contiguous Load group
SVE contiguous non-temporal load (scalar plus immediate)

This section describes the encoding of the SVE contiguous non-temporal load (scalar plus immediate) instruction class. The encodings in this section are decoded from SVE Memory - Contiguous Load on page C4-506.

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20]</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 msz 0 0 imm4 1 1 Pg Rn Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>LDNT1B (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>LDNT1H (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>LDNT1W (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>LDNT1D (scalar plus immediate)</td>
</tr>
</tbody>
</table>

SVE contiguous non-temporal load (scalar plus scalar)

This section describes the encoding of the SVE contiguous non-temporal load (scalar plus scalar) instruction class. The encodings in this section are decoded from SVE Memory - Contiguous Load on page C4-506.

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20]</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 msz 0 0 Rm 1 1 Pg Rn Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>LDNT1B (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>LDNT1H (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>LDNT1W (scalar plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>LDNT1D (scalar plus scalar)</td>
</tr>
</tbody>
</table>
SVE load multiple structures (scalar plus immediate)

This section describes the encoding of the SVE load multiple structures (scalar plus immediate) instruction class. The encodings in this section are decoded from *SVE Memory - Contiguous Load* on page C4-506.

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[19 16 15 14 13 12]</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
<td>msz</td>
<td>0 0</td>
<td>imm4</td>
<td>1 1</td>
<td>Pg</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**Decode fields**

<table>
<thead>
<tr>
<th>msz</th>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>LD2B (scalar plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>LD3B (scalar plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>LD4B (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>LD2H (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>LD3H (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>LD4H (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>LD2W (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>LD3W (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>LD4W (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>LD2D (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>LD3D (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>LD4D (scalar plus immediate)</td>
</tr>
</tbody>
</table>

SVE load multiple structures (scalar plus scalar)

This section describes the encoding of the SVE load multiple structures (scalar plus scalar) instruction class. The encodings in this section are decoded from *SVE Memory - Contiguous Load* on page C4-506.

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[19 16 15 14 13 12]</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
<td>msz</td>
<td>00</td>
<td>Rm</td>
<td>1 1 0</td>
<td>Pg</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**Decode fields**

<table>
<thead>
<tr>
<th>msz</th>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>LD2B (scalar plus scalar)</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>LD3B (scalar plus scalar)</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>LD4B (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>LD2H (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>LD3H (scalar plus scalar)</td>
</tr>
</tbody>
</table>
SVE load and broadcast quadword (scalar plus immediate)

This section describes the encoding of the SVE load and broadcast quadword (scalar plus immediate) instruction class. The encodings in this section are decoded from SVE Memory - Contiguous Load on page C4-506.

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12| 10 9 |  5 4 |  0 |
|---|---|---|---|---|---|---|---|---|
| 1 0 1 0 1 0 0 1 0 msz ssz 0 imm4 0 0 1 Pg Rn Zt |
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz opc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 11</td>
<td>LD4H (scalar plus scalar)</td>
<td></td>
</tr>
<tr>
<td>10 01</td>
<td>LD2W (scalar plus scalar)</td>
<td></td>
</tr>
<tr>
<td>10 10</td>
<td>LD3W (scalar plus scalar)</td>
<td></td>
</tr>
<tr>
<td>10 11</td>
<td>LD4W (scalar plus scalar)</td>
<td></td>
</tr>
<tr>
<td>11 01</td>
<td>LD2D (scalar plus scalar)</td>
<td></td>
</tr>
<tr>
<td>11 10</td>
<td>LD3D (scalar plus scalar)</td>
<td></td>
</tr>
<tr>
<td>11 11</td>
<td>LD4D (scalar plus scalar)</td>
<td></td>
</tr>
</tbody>
</table>

SVE contiguous load (scalar plus immediate)

This section describes the encoding of the SVE contiguous load (scalar plus immediate) instruction class. The encodings in this section are decoded from SVE Memory - Contiguous Load on page C4-506.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz ssz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- 1x</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>00 00</td>
<td>LD1RQB (scalar plus immediate)</td>
<td></td>
</tr>
<tr>
<td>00 01</td>
<td>LD1ROB (scalar plus immediate)</td>
<td>FEAT_F64MM</td>
</tr>
<tr>
<td>01 00</td>
<td>LD1RQH (scalar plus immediate)</td>
<td></td>
</tr>
<tr>
<td>01 01</td>
<td>LD1ROH (scalar plus immediate)</td>
<td>FEAT_F64MM</td>
</tr>
<tr>
<td>10 00</td>
<td>LD1RQW (scalar plus immediate)</td>
<td></td>
</tr>
<tr>
<td>10 01</td>
<td>LD1ROW (scalar plus immediate)</td>
<td>FEAT_F64MM</td>
</tr>
<tr>
<td>11 00</td>
<td>LD1RQD (scalar plus immediate)</td>
<td></td>
</tr>
<tr>
<td>11 01</td>
<td>LD1ROD (scalar plus immediate)</td>
<td>FEAT_F64MM</td>
</tr>
</tbody>
</table>
A64 Instruction Set Encoding
C4.1 A64 instruction set encoding

SVE contiguous non-fault load (scalar plus immediate)

This section describes the encoding of the SVE contiguous non-fault load (scalar plus immediate) instruction class. The encodings in this section are decoded from SVE Memory - Contiguous Load on page C4-506.
This section describes the encoding of the SVE load and broadcast quadword (scalar plus scalar) instruction class. The encodings in this section are decoded from SVE Memory - Contiguous Load on page C4-506.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 0 1 0 msz ssz Rm 0 0 0 Pg</td>
<td>Rn Zt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### SVE load and broadcast quadword (scalar plus scalar)

The encodings in this section are decoded from SVE Memory - Contiguous Load on page C4-506.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz ssz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 00</td>
<td>LD1RQB (scalar plus scalar)</td>
<td>-</td>
</tr>
<tr>
<td>00 01</td>
<td>LD1ROB (scalar plus scalar)</td>
<td>FEAT_F64MM</td>
</tr>
<tr>
<td>01 00</td>
<td>LD1RQH (scalar plus scalar)</td>
<td>-</td>
</tr>
<tr>
<td>01 01</td>
<td>LD1ROH (scalar plus scalar)</td>
<td>FEAT_F64MM</td>
</tr>
<tr>
<td>10 00</td>
<td>LD1RQW (scalar plus scalar)</td>
<td>-</td>
</tr>
<tr>
<td>10 01</td>
<td>LD1ROW (scalar plus scalar)</td>
<td>FEAT_F64MM</td>
</tr>
<tr>
<td>11 00</td>
<td>LD1RQD (scalar plus scalar)</td>
<td>-</td>
</tr>
<tr>
<td>11 01</td>
<td>LD1ROD (scalar plus scalar)</td>
<td>FEAT_F64MM</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>(dtype)</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td>LDNF1B - Encoding</td>
</tr>
<tr>
<td>0100</td>
<td>LDNF1SW</td>
</tr>
<tr>
<td>0101</td>
<td>LDNF1H - Encoding</td>
</tr>
<tr>
<td>0110</td>
<td>LDNF1H - Encoding</td>
</tr>
<tr>
<td>0111</td>
<td>LDNF1H - Encoding</td>
</tr>
<tr>
<td>1000</td>
<td>LDNF1SH - Encoding</td>
</tr>
<tr>
<td>1001</td>
<td>LDNF1SH - Encoding</td>
</tr>
<tr>
<td>1010</td>
<td>LDNF1W - Encoding</td>
</tr>
<tr>
<td>1011</td>
<td>LDNF1W - Encoding</td>
</tr>
<tr>
<td>1100</td>
<td>LDNF1SB - Encoding</td>
</tr>
<tr>
<td>1101</td>
<td>LDNF1SB - Encoding</td>
</tr>
<tr>
<td>1110</td>
<td>LDNF1SB - Encoding</td>
</tr>
<tr>
<td>1111</td>
<td>LDNF1D</td>
</tr>
</tbody>
</table>

### Instruction page Feature

- 1x: Unallocated.
SVE contiguous load (scalar plus scalar)

This section describes the encoding of the SVE contiguous load (scalar plus scalar) instruction class. The encodings in this section are decoded from *SVE Memory - Contiguous Load on page C4-506*.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16 15 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
<td>dtype</td>
<td>Rm</td>
<td>0 1 0</td>
<td>Pg</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**Decode fields**

<table>
<thead>
<tr>
<th>dtype</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>LD1B (scalar plus scalar) - Encoding</td>
</tr>
<tr>
<td>0001</td>
<td>LD1B (scalar plus scalar) - Encoding</td>
</tr>
<tr>
<td>0010</td>
<td>LD1B (scalar plus scalar) - Encoding</td>
</tr>
<tr>
<td>0011</td>
<td>LD1B (scalar plus scalar) - Encoding</td>
</tr>
<tr>
<td>0100</td>
<td>LD1SW (scalar plus scalar)</td>
</tr>
<tr>
<td>0101</td>
<td>LD1H (scalar plus scalar) - Encoding</td>
</tr>
<tr>
<td>0110</td>
<td>LD1H (scalar plus scalar) - Encoding</td>
</tr>
<tr>
<td>0111</td>
<td>LD1H (scalar plus scalar) - Encoding</td>
</tr>
<tr>
<td>1000</td>
<td>LD1SH (scalar plus scalar) - Encoding</td>
</tr>
<tr>
<td>1001</td>
<td>LD1SH (scalar plus scalar) - Encoding</td>
</tr>
<tr>
<td>1010</td>
<td>LD1W (scalar plus scalar) - Encoding</td>
</tr>
<tr>
<td>1011</td>
<td>LD1W (scalar plus scalar) - Encoding</td>
</tr>
<tr>
<td>1100</td>
<td>LD1SB (scalar plus scalar) - Encoding</td>
</tr>
<tr>
<td>1101</td>
<td>LD1SB (scalar plus scalar) - Encoding</td>
</tr>
<tr>
<td>1110</td>
<td>LD1SB (scalar plus scalar) - Encoding</td>
</tr>
<tr>
<td>1111</td>
<td>LD1D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

SVE contiguous first-fault load (scalar plus scalar)

This section describes the encoding of the SVE contiguous first-fault load (scalar plus scalar) instruction class. The encodings in this section are decoded from *SVE Memory - Contiguous Load on page C4-506*. 
C4.1.59  SVE Memory - 64-bit Gather

This section describes the encoding of the SVE Memory - 64-bit Gather group. The encodings in this section are decoded from *SVE encodings on page C4-399.*
SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)

This section describes the encoding of the SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets) instruction class. The encodings in this section are decoded from SVE Memory - 64-bit Gather on page C4-513.

```
[31 30 29 28 27 26 25 24 23 22 21 20] | 16|15 14 13 12| 10  9 | 5  4  3  0 |
1 1 0 0 0 1 0 0 0 1 1 Zm 1 msz Pg Rn 0 prfop
```

Table C4-60 Encoding table for the SVE Memory - 64-bit Gather group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1 op2 op3</td>
</tr>
<tr>
<td>00</td>
<td>01 0xx 1</td>
</tr>
<tr>
<td>00</td>
<td>11 1xx 0</td>
</tr>
<tr>
<td>00</td>
<td>11 - 1</td>
</tr>
<tr>
<td>00</td>
<td>x1 0xx 0</td>
</tr>
<tr>
<td>!= 00</td>
<td>11 1xx -</td>
</tr>
<tr>
<td>!= 00</td>
<td>x1 0xx -</td>
</tr>
<tr>
<td>-</td>
<td>00 101 -</td>
</tr>
<tr>
<td>-</td>
<td>00 111 0</td>
</tr>
<tr>
<td>-</td>
<td>00 111 1</td>
</tr>
<tr>
<td>-</td>
<td>00 1x0 -</td>
</tr>
<tr>
<td>-</td>
<td>01 1xx -</td>
</tr>
<tr>
<td>-</td>
<td>00 1xx -</td>
</tr>
<tr>
<td>-</td>
<td>x0 0xx -</td>
</tr>
</tbody>
</table>

SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)
SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)

This section describes the encoding of the SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets) instruction class. The encodings in this section are decoded from *SVE Memory - 64-bit Gather on page C4-513*.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0 0</td>
<td>xs 1</td>
<td>Zm 0</td>
<td>msz</td>
<td>Pg</td>
<td>Rn 0</td>
<td>prfop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

### Decode fields

<table>
<thead>
<tr>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
</tr>
<tr>
<td>00 PRFB (scalar plus vector)</td>
</tr>
<tr>
<td>01 PRFH (scalar plus vector)</td>
</tr>
<tr>
<td>10 PRFW (scalar plus vector)</td>
</tr>
<tr>
<td>11 PRFD (scalar plus vector)</td>
</tr>
</tbody>
</table>

SVE 64-bit gather load (scalar plus 64-bit scaled offsets)

This section describes the encoding of the SVE 64-bit gather load (scalar plus 64-bit scaled offsets) instruction class. The encodings in this section are decoded from *SVE Memory - 64-bit Gather on page C4-513*.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
<td>l=00</td>
<td>1 1</td>
<td>Zm 1</td>
<td>U</td>
<td>ff</td>
<td>Pg</td>
<td>Rn</td>
</tr>
</tbody>
</table>
```

### Decode fields

<table>
<thead>
<tr>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc U ff</td>
</tr>
<tr>
<td>01 0 0 0 LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01 0 1 0 LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01 1 0 0 LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>01 1 1 0 LDFF1H (scalar plus vector)</td>
</tr>
<tr>
<td>10 0 0 0 LD1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10 0 1 0 LDFF1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10 1 0 0 LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>10 1 1 0 LDFF1W (scalar plus vector)</td>
</tr>
<tr>
<td>11 0 0 - Unallocated.</td>
</tr>
<tr>
<td>11 1 0 0 LD1D (scalar plus vector)</td>
</tr>
<tr>
<td>11 1 1 0 LDFF1D (scalar plus vector)</td>
</tr>
</tbody>
</table>
SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets)

This section describes the encoding of the SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets) instruction class. The encodings in this section are decoded from SVE Memory - 64-bit Gather on page C4-513.

\[
\begin{array}{cccccccc}
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 |
\hline
1 & 1 & 0 & 0 & 1 & 0 & \text{!}=00 & \text{x}s & 1 & \text{Zm} & 0 & \text{U} & \text{ff} & \text{Pg} & \text{Rn} & \text{Zt} & \text{opc}
\end{array}
\]

<table>
<thead>
<tr>
<th>opc</th>
<th>U</th>
<th>ff</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>LD1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>LDFF1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>LD1D (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>LDFF1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

SVE 64-bit gather prefetch (vector plus immediate)

This section describes the encoding of the SVE 64-bit gather prefetch (vector plus immediate) instruction class. The encodings in this section are decoded from SVE Memory - 64-bit Gather on page C4-513.

\[
\begin{array}{ccccccccccc}
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 |
\hline
1 & 1 & 0 & 0 & 1 & 0 & \text{msz} & 0 & 0 & \text{imm5} & 1 & 1 & \text{Pg} & \text{Zn} & 0 & \text{prfop} & \text{opc}
\end{array}
\]

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>PRFB (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>PRFH (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>PRFW (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>PRFD (vector plus immediate)</td>
</tr>
</tbody>
</table>
## SVE2 64-bit gather non-temporal load (vector plus scalar)

This section describes the encoding of the SVE2 64-bit gather non-temporal load (vector plus scalar) instruction class. The encodings in this section are decoded from *SVE Memory - 64-bit Gather on page C4-513*.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0</td>
<td>0 1 0</td>
<td>msz</td>
<td>0 0</td>
<td>Rm</td>
<td>1</td>
<td>U</td>
<td>Zn</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>msz</th>
<th>U</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>LDNT1SB</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>LDNT1B (vector plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>LDNT1SH</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>LDNT1H (vector plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>LDNT1SW</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>LDNT1W (vector plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>LDNT1D (vector plus scalar)</td>
</tr>
</tbody>
</table>

## SVE 64-bit gather load (vector plus immediate)

This section describes the encoding of the SVE 64-bit gather load (vector plus immediate) instruction class. The encodings in this section are decoded from *SVE Memory - 64-bit Gather on page C4-513*.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0</td>
<td>0 1 0</td>
<td>msz</td>
<td>0 1</td>
<td>imm5</td>
<td>1</td>
<td>U</td>
<td>ff</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>msz</th>
<th>U</th>
<th>ff</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>LD1SB (vector plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>LDFF1SB (vector plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>LD1B (vector plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>LDFF1B (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>LD1SH (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>LDFF1SH (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>LD1H (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDFF1H (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>LD1SW (vector plus immediate)</td>
</tr>
</tbody>
</table>
This section describes the encoding of the SVE 64-bit gather load (scalar plus 64-bit unscaled offsets) instruction class. The encodings in this section are decoded from SVE Memory - 64-bit Gather on page C4-513.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
<td>U</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
<td>U</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>
SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)

This section describes the encoding of the SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets) instruction class. The encodings in this section are decoded from SVE Memory - 64-bit Gather on page C4-513.

```
|31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13 12| 10 9 | 5 4 | 0 |
1 1 0 0 1 0 0 0 | msz | xs | Zm | 0 | U | ff | Pg | Rn | Zt |
```

### Decode fields

<table>
<thead>
<tr>
<th>msz</th>
<th>U</th>
<th>ff</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>LD1SB (scalar plus vector)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>LDFF1SB (scalar plus vector)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>LD1B (scalar plus vector)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>LDFF1B (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>LD1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>LDFF1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>LD1D (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>LDFF1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

### C4.1.60 SVE Memory - Contiguous Store and Unsized Contiguous

This section describes the encoding of the SVE Memory - Contiguous Store and Unsized Contiguous group. The encodings in this section are decoded from SVE encodings on page C4-399.
SVE contiguous store (scalar plus scalar)

This section describes the encoding of the SVE contiguous store (scalar plus scalar) instruction class. The encodings in this section are decoded from SVE Memory - Contiguous Store and Unsized Contiguous on page C4-519.

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 0 1 0</td>
<td>! = 110</td>
<td>o2</td>
<td>Rm</td>
<td>0 1 0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
</tr>
</tbody>
</table>

Table C4-61 Encoding table for the SVE Memory - Contiguous Store and Unsized Contiguous group

<table>
<thead>
<tr>
<th>Decode fields op0</th>
<th>op1</th>
<th>op2</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xx</td>
<td>0</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10x</td>
<td>0</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>0</td>
<td>STR (predicate)</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>-</td>
<td>STR (vector)</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>!= 110</td>
<td>1</td>
<td>-</td>
<td>SVE contiguous store (scalar plus scalar)</td>
</tr>
</tbody>
</table>

C4.1.61 SVE Memory - Non-temporal and Multi-register Store

This section describes the encoding of the SVE Memory - Non-temporal and Multi-register Store group. The encodings in this section are decoded from SVE encodings on page C4-399.
This section describes the encoding of the SVE2 64-bit scatter non-temporal store (vector plus scalar) instruction class. The encodings in this section are decoded from SVE Memory - Non-temporal and Multi-register Store on page C4-520.

### SVE2 64-bit scatter non-temporal store (vector plus scalar)

This section describes the encoding of the SVE2 64-bit scatter non-temporal store (vector plus scalar) instruction class. The encodings in this section are decoded from SVE Memory - Non-temporal and Multi-register Store on page C4-520.

**Table C4-62 Encoding table for the SVE Memory - Non-temporal and Multi-register Store group**

<table>
<thead>
<tr>
<th>Decode group or instruction page</th>
<th>op0</th>
<th>op1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVE2 64-bit scatter non-temporal store (vector plus scalar)</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>SVE contiguous non-temporal store (scalar plus scalar)</td>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>SVE2 32-bit scatter non-temporal store (vector plus scalar) on page C4-522</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>SVE store multiple structures (scalar plus scalar) on page C4-522</td>
<td>!= 00</td>
<td>1</td>
</tr>
<tr>
<td>Unallocated.</td>
<td>x1</td>
<td>0</td>
</tr>
</tbody>
</table>

**SVE contiguous non-temporal store (scalar plus scalar)**

This section describes the encoding of the SVE contiguous non-temporal store (scalar plus scalar) instruction class. The encodings in this section are decoded from SVE Memory - Non-temporal and Multi-register Store on page C4-520.

## Decode fields

<table>
<thead>
<tr>
<th>Instruction page</th>
<th>msz</th>
</tr>
</thead>
<tbody>
<tr>
<td>STNT1B (vector plus scalar)</td>
<td>00</td>
</tr>
<tr>
<td>STNT1H (vector plus scalar)</td>
<td>01</td>
</tr>
<tr>
<td>STNT1W (vector plus scalar)</td>
<td>10</td>
</tr>
<tr>
<td>STNT1D (vector plus scalar)</td>
<td>11</td>
</tr>
</tbody>
</table>
SVE2 32-bit scatter non-temporal store (vector plus scalar)

This section describes the encoding of the SVE2 32-bit scatter non-temporal store (vector plus scalar) instruction class. The encodings in this section are decoded from SVE Memory - Non-temporal and Multi-register Store on page C4-520.

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STNT1B (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>STNT1H (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>STNT1W (scalar plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>STNT1D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

SVE store multiple structures (scalar plus scalar)

This section describes the encoding of the SVE store multiple structures (scalar plus scalar) instruction class. The encodings in this section are decoded from SVE Memory - Non-temporal and Multi-register Store on page C4-520.

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STNT1B (vector plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>STNT1H (vector plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>STNT1W (vector plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
C4.1.62 SVE Memory - Scatter with Optional Sign Extend

This section describes the encoding of the SVE Memory - Scatter with Optional Sign Extend group. The encodings in this section are decoded from SVE encodings on page C4-399.

| 1110010 op0 | 1 | 0
|-------------|---|---

Table C4-63 Encoding table for the SVE Memory - Scatter with Optional Sign Extend group

<table>
<thead>
<tr>
<th>op0</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets) on page C4-524</td>
</tr>
<tr>
<td>01</td>
<td>SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets) on page C4-524</td>
</tr>
<tr>
<td>10</td>
<td>SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets) on page C4-524</td>
</tr>
<tr>
<td>11</td>
<td>SVE 32-bit scatter store (scalar plus 32-bit scaled offsets) on page C4-525</td>
</tr>
</tbody>
</table>
SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)

This section describes the encoding of the SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets) instruction class. The encodings in this section are decoded from SVE Memory - Scatter with Optional Sign Extend on page C4-523.

![Instruction encoding diagram]

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0</td>
<td>msz 0 0</td>
<td>Zm 1 xs 0</td>
<td>Pg</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**Decode fields**

<table>
<thead>
<tr>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
</tr>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)

This section describes the encoding of the SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets) instruction class. The encodings in this section are decoded from SVE Memory - Scatter with Optional Sign Extend on page C4-523.

![Instruction encoding diagram]

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0</td>
<td>msz 0 1</td>
<td>Zm 1 xs 0</td>
<td>Pg</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**Decode fields**

<table>
<thead>
<tr>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
</tr>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets)

This section describes the encoding of the SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets) instruction class. The encodings in this section are decoded from SVE Memory - Scatter with Optional Sign Extend on page C4-523.
SVE 32-bit scatter store (scalar plus 32-bit scaled offsets)

This section describes the encoding of the SVE 32-bit scatter store (scalar plus 32-bit scaled offsets) instruction class. The encodings in this section are decoded from *SVE Memory - Scatter with Optional Sign Extend* on page C4-523.

C4.1.63   SVE Memory - Scatter

This section describes the encoding of the SVE Memory - Scatter group. The encodings in this section are decoded from *SVE encodings* on page C4-399.
A64 Instruction Set Encoding
C4.1 A64 instruction set encoding

This section describes the encoding of the SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets) instruction class. The encodings in this section are decoded from SVE Memory - Scatter on page C4-525.

**SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)**

This section describes the encoding of the SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets) instruction class. The encodings in this section are decoded from SVE Memory - Scatter on page C4-525.

This section describes the encoding of the SVE 64-bit scatter store (scalar plus 64-bit scaled offsets) instruction class. The encodings in this section are decoded from SVE Memory - Scatter on page C4-525.
SVE 64-bit scatter store (vector plus immediate)

This section describes the encoding of the SVE 64-bit scatter store (vector plus immediate) instruction class. The encodings in this section are decoded from SVE Memory - Scatter on page C4-525.

SVE 32-bit scatter store (vector plus immediate)

This section describes the encoding of the SVE 32-bit scatter store (vector plus immediate) instruction class. The encodings in this section are decoded from SVE Memory - Scatter on page C4-525.
C4.1.64 SVE Memory - Contiguous Store with Immediate Offset

This section describes the encoding of the SVE Memory - Contiguous Store with Immediate Offset group. The encodings in this section are decoded from *SVE encodings on page C4-399.*

\[
\begin{array}{cccccccccc}
|31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
SVE store multiple structures (scalar plus immediate)

This section describes the encoding of the SVE store multiple structures (scalar plus immediate) instruction class. The encodings in this section are decoded from SVE Memory - Contiguous Store with Immediate Offset on page C4-528.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12| 10 9 | 5 4 | 0 ]
1 1 1 0 0 1 0 msz 0 0 1 imm4 1 1 1 Pg Rn Zt
```

### Decode fields

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STNT1B (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>STNT1H (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>STNT1W (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>STNT1D (scalar plus immediate)</td>
</tr>
</tbody>
</table>

### SVE contiguous store (scalar plus immediate)

This section describes the encoding of the SVE contiguous store (scalar plus immediate) instruction class. The encodings in this section are decoded from SVE Memory - Contiguous Store with Immediate Offset on page C4-528.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12| 10 9 | 5 4 | 0 ]
1 1 1 0 0 1 0 msz !=00 imm4 1 1 1 Pg Rn Zt
```

### Decode fields

<table>
<thead>
<tr>
<th>msz</th>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>ST2B (scalar plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>ST3B (scalar plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>ST4B (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>ST2H (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>ST3H (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>ST4H (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>ST2W (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>ST3W (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>ST4W (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>ST2D (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>ST3D (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>ST4D (scalar plus immediate)</td>
</tr>
</tbody>
</table>
C4.1.65 Data Processing -- Immediate

This section describes the encoding of the Data Processing -- Immediate group. The encodings in this section are decoded from A64 instruction set encoding on page C4-388.

Table C4-66 Encoding table for the Data Processing -- Immediate group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>PC-rel. addressing on page C4-530</td>
</tr>
<tr>
<td>010</td>
<td>Add/subtract (immediate) on page C4-531</td>
</tr>
<tr>
<td>011</td>
<td>Add/subtract (immediate, with tags) on page C4-531</td>
</tr>
<tr>
<td>100</td>
<td>Logical (immediate) on page C4-532</td>
</tr>
<tr>
<td>101</td>
<td>Move wide (immediate) on page C4-532</td>
</tr>
<tr>
<td>110</td>
<td>Bitfield on page C4-533</td>
</tr>
<tr>
<td>111</td>
<td>Extract on page C4-534</td>
</tr>
</tbody>
</table>

PC-rel. addressing

This section describes the encoding of the PC-rel. addressing instruction class. The encodings in this section are decoded from Data Processing -- Immediate.
Add/subtract (immediate)

This section describes the encoding of the Add/subtract (immediate) instruction class. The encodings in this section are decoded from Data Processing -- Immediate on page C4-530.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>immlo</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADR</td>
</tr>
<tr>
<td>1</td>
<td>ADRP</td>
</tr>
</tbody>
</table>

Add/subtract (immediate, with tags)

This section describes the encoding of the Add/subtract (immediate, with tags) instruction class. The encodings in this section are decoded from Data Processing -- Immediate on page C4-530.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>op</td>
<td>S</td>
<td>imm12</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Decode fields

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>ADD (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>ADDS (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SUB (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>SUBS (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>ADD (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>ADDS (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>SUB (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>SUBS (immediate) - 64-bit variant</td>
</tr>
</tbody>
</table>
Logical (immediate)

This section describes the encoding of the Logical (immediate) instruction class. The encodings in this section are decoded from *Data Processing -- Immediate* on page C4-530.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>ADDG</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SUBG</td>
</tr>
</tbody>
</table>

Move wide (immediate)

This section describes the encoding of the Move wide (immediate) instruction class. The encodings in this section are decoded from *Data Processing -- Immediate* on page C4-530.
Bitfield

This section describes the encoding of the Bitfield instruction class. The encodings in this section are decoded from *Data Processing -- Immediate* on page C4-530.
Extract

This section describes the encoding of the Extract instruction class. The encodings in this section are decoded from Data Processing -- Immediate on page C4-530.

```
|sf| op21| 1| 0| 1| 1| N| o0| Rm| imms| Rn| Rd|
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf op21 N o0 imms</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- xl - - -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- 00 - 1 -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>- lx - - -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 - - - 1xxxx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 - 1 - -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 00 0 0 0xxxx</td>
<td>EXTR - 32-bit variant</td>
</tr>
<tr>
<td>1 - 0 - -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 00 1 0 -</td>
<td>EXTR - 64-bit variant</td>
</tr>
</tbody>
</table>

C4.1.66 Branches, Exception Generating and System instructions

This section describes the encoding of the Branches, Exception Generating and System instructions group. The encodings in this section are decoded from A64 instruction set encoding on page C4-388.

```
|31| 29| 28| 27| 26| 25| 24| 23| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| op0| 101|   | op1|   | op2|   |
```

Table C4-67 Encoding table for the Branches, Exception Generating and System instructions group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2</td>
<td>Conditional branch (immediate) on page C4-535</td>
</tr>
<tr>
<td>010 0xxxxxxxxxxx -</td>
<td>exception generation on page C4-535</td>
</tr>
<tr>
<td>110 00xxxxxxx -</td>
<td>System instructions with register argument on page C4-536</td>
</tr>
<tr>
<td>110 01000000110001 -</td>
<td>Hints on page C4-537</td>
</tr>
<tr>
<td>110 01000000110010 11111</td>
<td>Barriers on page C4-538</td>
</tr>
<tr>
<td>110 01000000110011 -</td>
<td>PSTATE on page C4-539</td>
</tr>
<tr>
<td>110 0100100xxxxxx -</td>
<td>System instructions on page C4-539</td>
</tr>
<tr>
<td>110 0100x01xxxxxx -</td>
<td>System instructions on page C4-539</td>
</tr>
</tbody>
</table>
Conditional branch (immediate)

This section describes the encoding of the Conditional branch (immediate) instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-534.

```
| 31 30 29 28|27 26 25 24|23 |  |  |  | 5 | 4 | 3 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | o1 | imm19 | o0 | cond
```

### Decode fields

<table>
<thead>
<tr>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>System register move on page C4-540</td>
</tr>
<tr>
<td>Unconditional branch (register) on page C4-540</td>
</tr>
<tr>
<td>Unconditional branch (immediate) on page C4-546</td>
</tr>
<tr>
<td>Compare and branch (immediate) on page C4-546</td>
</tr>
<tr>
<td>Test and branch (immediate) on page C4-547</td>
</tr>
</tbody>
</table>

### Exception generation

This section describes the encoding of the Exception generation instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-534.

```
| 31 30 29 28|27 26 25 24|23 | 21 | 20 |  |  |  | 5 | 4 | 2 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | opc | imm16 | op2 | LL |
```

### Decode fields

<table>
<thead>
<tr>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

### Exception generation

This section describes the encoding of the Exception generation instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-534.

```
| 31 30 29 28|27 26 25 24|23 | 21 | 20 |  |  |  | 5 | 4 | 2 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | opc | imm16 | op2 | LL |
```

### Decode fields

<table>
<thead>
<tr>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
### System instructions with register argument

This section describes the encoding of the System instructions with register argument instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-534.

<table>
<thead>
<tr>
<th>opc</th>
<th>op2</th>
<th>LL</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>01</td>
<td>SVC</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>000</td>
<td>10</td>
<td>HVC</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>000</td>
<td>11</td>
<td>SMC</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>x1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>00</td>
<td>BRK</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>1x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>000</td>
<td>x1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>000</td>
<td>00</td>
<td>HLT</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>000</td>
<td>1x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>000</td>
<td>00</td>
<td>TCANCEL</td>
<td>FEAT_TME</td>
</tr>
<tr>
<td>011</td>
<td>000</td>
<td>01</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>000</td>
<td>1x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>000</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>101</td>
<td>000</td>
<td>00</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>101</td>
<td>000</td>
<td>01</td>
<td>DCPS1</td>
<td>-</td>
</tr>
<tr>
<td>101</td>
<td>000</td>
<td>10</td>
<td>DCPS2</td>
<td>-</td>
</tr>
<tr>
<td>101</td>
<td>000</td>
<td>11</td>
<td>DCPS3</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>000</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>111</td>
<td>000</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
<th>Rt</th>
<th>Decode fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>000</td>
<td>WFET</td>
<td>FEAT_WFxT</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 0000</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
Hints

This section describes the encoding of the Hints instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-534.

<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>001</td>
<td>WFIT</td>
<td>FEAT_WFxT</td>
</tr>
<tr>
<td>0000</td>
<td>01x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0000</td>
<td>1xx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

*Decode fields*

<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td></td>
<td>DGH</td>
</tr>
<tr>
<td>000</td>
<td>111</td>
<td>XPACD, XPACI, XPACLRI</td>
</tr>
<tr>
<td>0001</td>
<td>000</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA - PACIA1716 variant</td>
</tr>
<tr>
<td>0001</td>
<td>010</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB - PACIB1716 variant</td>
</tr>
<tr>
<td>0001</td>
<td>100</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA - AUTIA1716 variant</td>
</tr>
<tr>
<td>0001</td>
<td>110</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB - AUTIB1716 variant</td>
</tr>
<tr>
<td>0010</td>
<td>000</td>
<td>ESB</td>
</tr>
<tr>
<td>0010</td>
<td>001</td>
<td>PSB CSYNC</td>
</tr>
<tr>
<td>0010</td>
<td>010</td>
<td>TSB CSYNC</td>
</tr>
<tr>
<td>0010</td>
<td>100</td>
<td>CSDB</td>
</tr>
<tr>
<td>0011</td>
<td>000</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA - PACIAZ variant</td>
</tr>
<tr>
<td>0011</td>
<td>001</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ - PACIASP variant</td>
</tr>
<tr>
<td>0011</td>
<td>010</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB - PACIBZ variant</td>
</tr>
</tbody>
</table>
Barriers

This section describes the encoding of the Barriers instruction class. The encodings in this section are decoded from Branches, Exception Generating and System instructions on page C4-534.

<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td>011</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB - PACIBSP variant</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>0011</td>
<td>100</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA - AUTIAZ variant</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>0011</td>
<td>101</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA - AUTIASP variant</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>0011</td>
<td>110</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB - AUTIBZ variant</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>0011</td>
<td>111</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB - AUTIBSP variant</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>0100</td>
<td>xx0</td>
<td>BTI</td>
<td>FEAT_BTI</td>
</tr>
</tbody>
</table>

### Barriers

This section describes the encoding of the Barriers instruction class. The encodings in this section are decoded from Branches, Exception Generating and System instructions on page C4-534.

```
<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>011</td>
<td>TCOMMIT</td>
<td>FEAT_TME</td>
</tr>
<tr>
<td>0001</td>
<td>011</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>001x</td>
<td>011</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01xx</td>
<td>011</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1xxx</td>
<td>011</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
```

### Decode fields

#### Instruction page

- **CRm**: 31 30 29 28 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 8 7 5 4 | 0
- **op2**: 1 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 1 1
- **Rt**: 31 30 29 28 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 8 7 5 4 | 0

### Decode fields

#### Instruction page

- **CRm**: 31 30 29 28 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 8 7 5 4 | 0
- **op2**: 1 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 1 1
- **Rt**: 31 30 29 28 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 8 7 5 4 | 0

### Feature

- **FEAT_PAuth**: PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB - PACIBSP variant
- **FEAT_BTI**: BTI
- **FEAT_TME**: TCOMMIT
### PSTATE

This section describes the encoding of the PSTATE instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions on page C4-534.*

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18] [16 15 14 13 12] [11] [ 8  7  5  4] [ 0 ]
1 1 0 1 0 1 0 0 0 0 0 | op1 | 0 1 0 0 | CRm | op2 | Rt
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1 op2 Rt</td>
<td></td>
<td>Unallocated.</td>
</tr>
<tr>
<td>000 000 000</td>
<td>CFINV</td>
<td>FEAT_FlagM</td>
</tr>
<tr>
<td>000 001 000</td>
<td>XAFLAG</td>
<td>FEAT_FlagM2</td>
</tr>
<tr>
<td>000 010 000</td>
<td>AXFLAG</td>
<td>FEAT_FlagM2</td>
</tr>
</tbody>
</table>

### System with result

This section describes the encoding of the System with result instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions on page C4-534.*

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18] [16 15 12] [11] [ 8  7  5  4] [ 0 ]
1 1 0 1 0 1 0 0 0 0 0 | op1 | CRn | CRm | op2 | Rt
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1 CRn CRm op2</td>
<td></td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011 != 001</td>
<td>TSTART</td>
<td>FEAT_TME</td>
</tr>
<tr>
<td>001 000 011</td>
<td>TTEST</td>
<td>FEAT_TME</td>
</tr>
</tbody>
</table>

### System instructions

This section describes the encoding of the System instructions instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions on page C4-534.*
System register move

This section describes the encoding of the System register move instruction class. The encodings in this section are decoded from Branches, Exception Generating and System instructions on page C4-534.

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 12 11 8 7 5 4 0 ]
1 1 0 1 0 1 0 0 L 0 1 0 1 0 1 0 1 0 0 0 0

Decode fields
Instruction page
L

0    SYS
1    SYSL
```

Unconditional branch (register)

This section describes the encoding of the Unconditional branch (register) instruction class. The encodings in this section are decoded from Branches, Exception Generating and System instructions on page C4-534.

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 12 11 8 7 5 4 0 ]
1 1 0 1 0 1 0 0 L 1 0 0 op1 CRn CRm op2 Rt

Decode fields
Instruction page
L

0    MSR (register)
1    MRS
```

<table>
<thead>
<tr>
<th>op</th>
<th>op2</th>
<th>op3</th>
<th>Rn</th>
<th>op4</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

Decode fields
Instruction page

<table>
<thead>
<tr>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated.</td>
</tr>
<tr>
<td>Unallocated.</td>
</tr>
<tr>
<td>BR</td>
</tr>
</tbody>
</table>

Feature
## Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>op2</th>
<th>op3</th>
<th>Rn</th>
<th>op4</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 0</td>
<td>1111</td>
<td>000001</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>000 0</td>
<td>1111</td>
<td>000010</td>
<td>-</td>
<td>!= 1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>000 0</td>
<td>1111</td>
<td>000010</td>
<td>-</td>
<td>11111</td>
<td>BRAA, BRAAZ, BRAB, BRABZ - Key A, zero modifier variant</td>
<td>FEAT_PAu th</td>
</tr>
<tr>
<td>000 0</td>
<td>1111</td>
<td>000011</td>
<td>-</td>
<td>!= 1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>000 0</td>
<td>1111</td>
<td>000011</td>
<td>-</td>
<td>11111</td>
<td>BRAA, BRAAZ, BRAB, BRABZ - Key B, zero modifier variant</td>
<td>FEAT_PAu th</td>
</tr>
<tr>
<td>000 0</td>
<td>1111</td>
<td>001xxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>000 0</td>
<td>1111</td>
<td>01xxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>000 0</td>
<td>1111</td>
<td>01xxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>000 0</td>
<td>1111</td>
<td>1xxxxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>000 1</td>
<td>1111</td>
<td>000000</td>
<td>-</td>
<td>!= 0000</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>000 1</td>
<td>1111</td>
<td>000000</td>
<td>-</td>
<td>00000</td>
<td>BLR</td>
<td>-</td>
</tr>
<tr>
<td>000 1</td>
<td>1111</td>
<td>000001</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>000 1</td>
<td>1111</td>
<td>000010</td>
<td>-</td>
<td>!= 1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>000 1</td>
<td>1111</td>
<td>000010</td>
<td>-</td>
<td>11111</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ - Key A, zero modifier variant</td>
<td>FEAT_PAu th</td>
</tr>
</tbody>
</table>

---

C4.1 A64 instruction set encoding

ARM DDI 0487I.a
ID081822
Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential
### Decode fields

<table>
<thead>
<tr>
<th>op&lt;sub&gt;c&lt;/sub&gt;</th>
<th>op2</th>
<th>op3</th>
<th>Rn</th>
<th>op4</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 1</td>
<td>1111</td>
<td>000011</td>
<td>-</td>
<td>!= 1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>000 1</td>
<td>1111</td>
<td>000011</td>
<td>-</td>
<td>1111</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ - Key B, zero modifier variant</td>
<td>FEAT_PAu</td>
</tr>
<tr>
<td>000 1</td>
<td>1111</td>
<td>001xx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>000 1</td>
<td>1111</td>
<td>01xxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>000 1</td>
<td>1111</td>
<td>1xxxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>001 0</td>
<td>1111</td>
<td>00000</td>
<td>-</td>
<td>!= 0000</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>001 0</td>
<td>1111</td>
<td>00000</td>
<td>-</td>
<td>0000</td>
<td>RET</td>
<td>-</td>
</tr>
<tr>
<td>001 0</td>
<td>1111</td>
<td>00001</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>001 0</td>
<td>1111</td>
<td>00010</td>
<td>!= 1111</td>
<td>!= 1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>001 0</td>
<td>1111</td>
<td>00010</td>
<td>!= 1111</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>001 0</td>
<td>1111</td>
<td>00010</td>
<td>1111</td>
<td>!= 1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>001 0</td>
<td>1111</td>
<td>00010</td>
<td>1111</td>
<td>1111</td>
<td>RETAA, RETAB - RETAA variant</td>
<td>FEAT_PAu</td>
</tr>
<tr>
<td>001 0</td>
<td>1111</td>
<td>00011</td>
<td>!= 1111</td>
<td>!= 1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>op2</th>
<th>op3</th>
<th>Rn</th>
<th>op4</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>0</td>
<td>000011</td>
<td>!=</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>000011</td>
<td>1111</td>
<td>!=</td>
<td>1111</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>000011</td>
<td>1111</td>
<td>1111</td>
<td>RETAA, RETAB - RETAB variant</td>
<td>FEAT_PAu th</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>1111</td>
<td>001xxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>1111</td>
<td>001xxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>01xxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>1xxxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>1xxxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>00000</td>
<td>!=</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>00000</td>
<td>!=</td>
<td>0000</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>00000</td>
<td>1111</td>
<td>!=</td>
<td>0000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>00000</td>
<td>1111</td>
<td>0000</td>
<td>ERET</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>00000</td>
<td>1111</td>
<td>0000</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>00000</td>
<td>1111</td>
<td>0000</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>op&lt;sub&gt;c&lt;/sub&gt;</th>
<th>op2</th>
<th>op3</th>
<th>Rn</th>
<th>op4</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>010 0</td>
<td>1111</td>
<td>000010</td>
<td>!= 1111</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>000010</td>
<td>1111</td>
<td>!= 1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>000010</td>
<td>1111</td>
<td>1111</td>
<td>ERETA, ERETAB - ERETA variant</td>
<td>FEAT_PAu th</td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>000011</td>
<td>!= 1111</td>
<td>!= 1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>000011</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>000011</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>000011</td>
<td>1111</td>
<td>1111</td>
<td>ERETA, ERETAB - ERETA variant</td>
<td>FEAT_PAu th</td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>000011</td>
<td>1111</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>000011</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>000011</td>
<td>1111</td>
<td>1111</td>
<td>ERETA, ERETAB - ERETA variant</td>
<td>FEAT_PAu th</td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>001xxx</td>
<td>1111</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>01xxx</td>
<td>1111</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>01xxx</td>
<td>1111</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>1xxxx</td>
<td>1111</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>1xxxx</td>
<td>1111</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>0xxxx</td>
<td>1111</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>0xxxx</td>
<td>1111</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>010 0</td>
<td>1111</td>
<td>0xxxx</td>
<td>1111</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
## Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>op2</th>
<th>op3</th>
<th>Rn</th>
<th>op4</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>1</td>
<td>1111</td>
<td>00000</td>
<td>1111</td>
<td>!= 0000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>1111</td>
<td>00000</td>
<td>1111</td>
<td>0000</td>
<td>DRPS</td>
</tr>
<tr>
<td>011</td>
<td>x</td>
<td>1111</td>
<td>00000</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1111</td>
<td>00000</td>
<td>1111</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1111</td>
<td>00010</td>
<td>-</td>
<td>-</td>
<td>BRAA, BRAAZ, BRAB, BRABZ - Key A, register modifier variant</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1111</td>
<td>00011</td>
<td>-</td>
<td>-</td>
<td>BRAA, BRAAZ, BRAB, BRABZ - Key B, register modifier variant</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1111</td>
<td>001xx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1111</td>
<td>01xxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1111</td>
<td>1xxxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1111</td>
<td>0xxxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1111</td>
<td>00010</td>
<td>-</td>
<td>-</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ - Key A, register modifier variant</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1111</td>
<td>00011</td>
<td>-</td>
<td>-</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ - Key B, register modifier variant</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1111</td>
<td>001xx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
Unconditional branch (immediate)

This section describes the encoding of the Unconditional branch (immediate) instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-534.

\[
\begin{array}{cccccccc}
\text{Instruction page} & \text{Feature} \\
\text{op} & \text{op2} & \text{op3} & \text{Rn} & \text{op4} & \hline
100 & 1 & 11111 & 001xxx & - & - & \text{Unallocated.} & - \\
100 & 1 & 11111 & 01xxxx & - & - & \text{Unallocated.} & - \\
100 & 1 & 11111 & 1xxxxx & - & - & \text{Unallocated.} & - \\
101 & x & 11111 & - & - & - & \text{Unallocated.} & - \\
11x & x & 11111 & - & - & - & \text{Unallocated.} & - \\
\end{array}
\]

Compare and branch (immediate)

This section describes the encoding of the Compare and branch (immediate) instruction class. The encodings in this section are decoded from *Branches, Exception Generating and System instructions* on page C4-534.

\[
\begin{array}{cccccccc}
\text{Instruction page} & \\
\text{op} & \hline
0 & \text{B} \\
1 & \text{BL} \\
\end{array}
\]
Test and branch (immediate)

This section describes the encoding of the Test and branch (immediate) instruction class. The encodings in this section are decoded from Branches, Exception Generating and System instructions on page C4-534.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>CBZ - 32-bit variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>CBNZ - 32-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>CBZ - 64-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CBNZ - 64-bit variant</td>
</tr>
</tbody>
</table>

C4.1.67 Loads and Stores

This section describes the encoding of the Loads and Stores group. The encodings in this section are decoded from A64 instruction set encoding on page C4-388.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0</td>
<td>0</td>
<td>1xxxxx</td>
<td>-</td>
<td>Compare and swap pair on page C4-549</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>0</td>
<td>000000</td>
<td>-</td>
<td>Advanced SIMD load/store multiple structures on page C4-549</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>01</td>
<td>0xxxxx</td>
<td>-</td>
<td>Advanced SIMD load/store multiple structures (post-indexed) on page C4-550</td>
</tr>
</tbody>
</table>
### Table C4-68 Encoding table for the Loads and Stores group (continued)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
</tr>
<tr>
<td>1x00</td>
<td>0</td>
</tr>
<tr>
<td>1x00</td>
<td>1</td>
</tr>
<tr>
<td>xx00</td>
<td>0</td>
</tr>
<tr>
<td>xx00</td>
<td>0</td>
</tr>
<tr>
<td>xx00</td>
<td>0</td>
</tr>
<tr>
<td>xx01</td>
<td>0</td>
</tr>
<tr>
<td>xx01</td>
<td>0</td>
</tr>
<tr>
<td>xx01</td>
<td>1</td>
</tr>
<tr>
<td>xx10</td>
<td>-</td>
</tr>
<tr>
<td>xx10</td>
<td>-</td>
</tr>
<tr>
<td>xx10</td>
<td>-</td>
</tr>
<tr>
<td>xx10</td>
<td>-</td>
</tr>
<tr>
<td>xx11</td>
<td>-</td>
</tr>
<tr>
<td>xx11</td>
<td>-</td>
</tr>
<tr>
<td>xx11</td>
<td>-</td>
</tr>
<tr>
<td>xx11</td>
<td>-</td>
</tr>
<tr>
<td>xx11</td>
<td>-</td>
</tr>
<tr>
<td>xx11</td>
<td>-</td>
</tr>
<tr>
<td>xx11</td>
<td>-</td>
</tr>
<tr>
<td>xx11</td>
<td>-</td>
</tr>
</tbody>
</table>
Compare and swap pair

This section describes the encoding of the Compare and swap pair instruction class. The encodings in this section are decoded from Loads and Stores on page C4-547.

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [16 15 14] [10 9] [5 4] [0]
[31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 10 9 5 4 0]
```

<table>
<thead>
<tr>
<th>sz</th>
<th>o0</th>
<th>Rt2</th>
<th>Rs</th>
<th>o0</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>Unallocated.</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL - 32-bit CASP variant</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL - 32-bit CASPL variant</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL - 32-bit CASPA variant</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL - 32-bit CASPAL variant</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL - 64-bit CASP variant</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL - 64-bit CASPL variant</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL - 64-bit CASPA variant</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL - 64-bit CASPAL variant</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Advanced SIMD load/store multiple structures

This section describes the encoding of the Advanced SIMD load/store multiple structures instruction class. The encodings in this section are decoded from Loads and Stores on page C4-547.

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 12 11 10 9 5 4 0]
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 10 9 5 4 0]
```

<table>
<thead>
<tr>
<th>sz</th>
<th>L</th>
<th>o0</th>
<th>opcode</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00000</td>
<td>ST4 (multiple structures)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0010</td>
<td>ST1 (multiple structures) - Four registers variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0100</td>
<td>ST3 (multiple structures)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0101</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0110</td>
<td>ST1 (multiple structures) - Three registers variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0111</td>
<td>ST1 (multiple structures) - One register variant</td>
</tr>
</tbody>
</table>
### Advanced SIMD load/store multiple structures (post-indexed)

This section describes the encoding of the Advanced SIMD load/store multiple structures (post-indexed) instruction class. The encodings in this section are decoded from *Loads and Stores on page C4-547.*

<table>
<thead>
<tr>
<th>L</th>
<th>opcode</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>ST2 (multiple structures)</td>
</tr>
<tr>
<td>0</td>
<td>1001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td>ST1 (multiple structures) - Two registers variant</td>
</tr>
<tr>
<td>0</td>
<td>1011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>11xx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>LD4 (multiple structures)</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>LD1 (multiple structures) - Four registers variant</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
<td>LD3 (multiple structures)</td>
</tr>
<tr>
<td>1</td>
<td>0101</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
<td>LD1 (multiple structures) - Three registers variant</td>
</tr>
<tr>
<td>1</td>
<td>0111</td>
<td>LD1 (multiple structures) - One register variant</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>LD2 (multiple structures)</td>
</tr>
<tr>
<td>1</td>
<td>1001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
<td>LD1 (multiple structures) - Two registers variant</td>
</tr>
<tr>
<td>1</td>
<td>1011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>11xx</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

---

**Decode fields**

<table>
<thead>
<tr>
<th>Instruction page</th>
<th>L</th>
<th>Rm</th>
<th>opcode</th>
<th>size</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>Q</td>
<td>0011001</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0011001</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---

**Decode fields**

<table>
<thead>
<tr>
<th>Instruction page</th>
<th>L</th>
<th>Rm</th>
<th>opcode</th>
<th>size</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>0101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>1001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>1011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>Rm</td>
<td>opcode</td>
<td>Instruction page</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>----</td>
<td>--------</td>
<td>------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>11xx</td>
<td>Unallocated.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!=1111</td>
<td>0000</td>
<td>ST4 (multiple structures) - Register offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!=1111</td>
<td>0010</td>
<td>ST1 (multiple structures) - Four registers, register offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!=1111</td>
<td>0100</td>
<td>ST3 (multiple structures) - Register offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!=1111</td>
<td>0110</td>
<td>ST1 (multiple structures) - Three registers, register offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!=1111</td>
<td>0111</td>
<td>ST1 (multiple structures) - One register, register offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!=1111</td>
<td>1000</td>
<td>ST2 (multiple structures) - Register offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!=1111</td>
<td>1010</td>
<td>ST1 (multiple structures) - Two registers, register offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>0000</td>
<td>ST4 (multiple structures) - Immediate offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>0010</td>
<td>ST1 (multiple structures) - Four registers, immediate offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>0100</td>
<td>ST3 (multiple structures) - Immediate offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>0110</td>
<td>ST1 (multiple structures) - Three registers, immediate offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>1000</td>
<td>ST2 (multiple structures) - Immediate offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>1010</td>
<td>ST1 (multiple structures) - Two registers, immediate offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0001</td>
<td>Unallocated.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0011</td>
<td>Unallocated.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0101</td>
<td>Unallocated.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1001</td>
<td>Unallocated.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1011</td>
<td>Unallocated.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>11xx</td>
<td>Unallocated.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!=1111</td>
<td>0000</td>
<td>LD4 (multiple structures) - Register offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!=1111</td>
<td>0010</td>
<td>LD1 (multiple structures) - Four registers, register offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!=1111</td>
<td>0100</td>
<td>LD3 (multiple structures) - Register offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!=1111</td>
<td>0110</td>
<td>LD1 (multiple structures) - Three registers, register offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!=1111</td>
<td>0111</td>
<td>LD1 (multiple structures) - One register, register offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!=1111</td>
<td>1000</td>
<td>LD2 (multiple structures) - Register offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!=1111</td>
<td>1010</td>
<td>LD1 (multiple structures) - Two registers, register offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>0000</td>
<td>LD4 (multiple structures) - Immediate offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>0010</td>
<td>LD1 (multiple structures) - Four registers, immediate offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>0100</td>
<td>LD3 (multiple structures) - Immediate offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>0110</td>
<td>LD1 (multiple structures) - Three registers, immediate offset variant</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Advanced SIMD load/store single structure

This section describes the encoding of the Advanced SIMD load/store single structure instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-547.

<table>
<thead>
<tr>
<th>L</th>
<th>Rm</th>
<th>opcode</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1111</td>
<td>0111</td>
<td>LD1 (multiple structures) - One register, immediate offset variant</td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>1000</td>
<td>LD2 (multiple structures) - Immediate offset variant</td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>1010</td>
<td>LD1 (multiple structures) - Two registers, immediate offset variant</td>
</tr>
</tbody>
</table>

### Decode fields

```
<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>opcode</th>
<th>S</th>
<th>size</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>-</td>
<td>x0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>-</td>
<td>x1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011</td>
<td>-</td>
<td>x0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011</td>
<td>-</td>
<td>x1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>000</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>001</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>-</td>
<td>x0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>-</td>
<td>x1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
### Decode fields

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>opcode</th>
<th>S</th>
<th>size</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>-</td>
<td>x0</td>
<td>ST4 (single structure) - 16-bit variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>-</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>-</td>
<td>00</td>
<td>ST2 (single structure) - 32-bit variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>-</td>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>0</td>
<td>01</td>
<td>ST2 (single structure) - 64-bit variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>0</td>
<td>11</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>-</td>
<td>00</td>
<td>ST4 (single structure) - 32-bit variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>-</td>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>0</td>
<td>01</td>
<td>ST4 (single structure) - 64-bit variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>1</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000</td>
<td>-</td>
<td>-</td>
<td>LD1 (single structure) - 8-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>001</td>
<td>-</td>
<td>-</td>
<td>LD3 (single structure) - 8-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>-</td>
<td>x0</td>
<td>LD1 (single structure) - 16-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>-</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>-</td>
<td>x0</td>
<td>LD3 (single structure) - 16-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>-</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>-</td>
<td>00</td>
<td>LD1 (single structure) - 32-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>-</td>
<td>1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>01</td>
<td>LD1 (single structure) - 64-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1</td>
<td>01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>-</td>
<td>00</td>
<td>LD3 (single structure) - 32-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>-</td>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>0</td>
<td>01</td>
<td>LD3 (single structure) - 64-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>1</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td>0</td>
<td>-</td>
<td>LD1R</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td>1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
<td>0</td>
<td>-</td>
<td>LD3R</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
<td>1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>000</td>
<td>-</td>
<td>-</td>
<td>LD2 (single structure) - 8-bit variant</td>
</tr>
</tbody>
</table>
## Advanced SIMD load/store single structure (post-indexed)

This section describes the encoding of the Advanced SIMD load/store single structure (post-indexed) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-547.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>L R opcode S size</td>
<td>LD4 (single structure) - 8-bit variant</td>
</tr>
<tr>
<td>1 1 001 - -</td>
<td></td>
</tr>
<tr>
<td>1 1 010 - x0</td>
<td>LD2 (single structure) - 16-bit variant</td>
</tr>
<tr>
<td>1 1 010 - x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 1 011 - x0</td>
<td>LD4 (single structure) - 16-bit variant</td>
</tr>
<tr>
<td>1 1 011 - x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 1 100 - 00</td>
<td>LD2 (single structure) - 32-bit variant</td>
</tr>
<tr>
<td>1 1 100 - 10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 1 100 0 01</td>
<td>LD2 (single structure) - 64-bit variant</td>
</tr>
<tr>
<td>1 1 100 0 11</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 1 100 1 x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 1 101 - 00</td>
<td>LD4 (single structure) - 32-bit variant</td>
</tr>
<tr>
<td>1 1 101 - 10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 1 101 0 01</td>
<td>LD4 (single structure) - 64-bit variant</td>
</tr>
<tr>
<td>1 1 101 0 11</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 1 101 1 x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 1 110 0 -</td>
<td>LD2R</td>
</tr>
<tr>
<td>1 1 110 1 -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 1 111 0 -</td>
<td>LD4R</td>
</tr>
<tr>
<td>1 1 111 1 -</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
## A64 Instruction Set Encoding

### C4.1 A64 instruction set encoding

#### Decode fields

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>Rm</th>
<th>opcode</th>
<th>S</th>
<th>size</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>100</td>
<td>1</td>
<td>01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>101</td>
<td>-</td>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>101</td>
<td>1</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 1111</td>
<td>000</td>
<td>-</td>
<td>-</td>
<td>ST1 (single structure) - 8-bit, register offset variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 1111</td>
<td>001</td>
<td>-</td>
<td>-</td>
<td>ST3 (single structure) - 8-bit, register offset variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 1111</td>
<td>010</td>
<td>-</td>
<td>x0</td>
<td>ST1 (single structure) - 16-bit, register offset variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 1111</td>
<td>011</td>
<td>-</td>
<td>x0</td>
<td>ST3 (single structure) - 16-bit, register offset variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 1111</td>
<td>100</td>
<td>-</td>
<td>00</td>
<td>ST1 (single structure) - 32-bit, register offset variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 1111</td>
<td>100</td>
<td>0</td>
<td>01</td>
<td>ST1 (single structure) - 64-bit, register offset variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 1111</td>
<td>101</td>
<td>-</td>
<td>00</td>
<td>ST3 (single structure) - 32-bit, register offset variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 1111</td>
<td>101</td>
<td>0</td>
<td>01</td>
<td>ST3 (single structure) - 64-bit, register offset variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 1111</td>
<td>111</td>
<td>0</td>
<td>11</td>
<td>ST1 (single structure) - 8-bit, immediate offset variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 1111</td>
<td>111</td>
<td>0</td>
<td>11</td>
<td>ST3 (single structure) - 8-bit, immediate offset variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 1111</td>
<td>111</td>
<td>1</td>
<td>x0</td>
<td>ST1 (single structure) - 16-bit, immediate offset variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 1111</td>
<td>111</td>
<td>0</td>
<td>01</td>
<td>ST3 (single structure) - 16-bit, immediate offset variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 1111</td>
<td>111</td>
<td>1</td>
<td>00</td>
<td>ST1 (single structure) - 32-bit, immediate offset variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 1111</td>
<td>111</td>
<td>0</td>
<td>01</td>
<td>ST3 (single structure) - 32-bit, immediate offset variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>010</td>
<td>-</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>011</td>
<td>-</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>100</td>
<td>-</td>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>100</td>
<td>0</td>
<td>11</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>100</td>
<td>1</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>101</td>
<td>-</td>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>101</td>
<td>1</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 1111</td>
<td>000</td>
<td>-</td>
<td>-</td>
<td>ST2 (single structure) - 8-bit, register offset variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 1111</td>
<td>001</td>
<td>-</td>
<td>-</td>
<td>ST4 (single structure) - 8-bit, register offset variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 1111</td>
<td>010</td>
<td>-</td>
<td>x0</td>
<td>ST2 (single structure) - 16-bit, register offset variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 1111</td>
<td>011</td>
<td>-</td>
<td>x0</td>
<td>ST4 (single structure) - 16-bit, register offset variant</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>Rm</th>
<th>opcode</th>
<th>S</th>
<th>size</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>!= 1111</td>
<td>100</td>
<td>-</td>
<td>00</td>
<td>ST2 (single structure) - 32-bit, register offset variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 1111</td>
<td>100</td>
<td>0</td>
<td>01</td>
<td>ST2 (single structure) - 64-bit, register offset variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 1111</td>
<td>101</td>
<td>-</td>
<td>00</td>
<td>ST4 (single structure) - 32-bit, register offset variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>000</td>
<td>-</td>
<td>-</td>
<td>ST2 (single structure) - 8-bit, immediate offset variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>001</td>
<td>-</td>
<td>-</td>
<td>ST4 (single structure) - 8-bit, immediate offset variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>010</td>
<td>-</td>
<td>x0</td>
<td>ST2 (single structure) - 16-bit, immediate offset variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>011</td>
<td>-</td>
<td>x0</td>
<td>ST4 (single structure) - 16-bit, immediate offset variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>-</td>
<td>00</td>
<td>ST2 (single structure) - 32-bit, immediate offset variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>0</td>
<td>01</td>
<td>ST4 (single structure) - 32-bit, immediate offset variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>-</td>
<td>00</td>
<td>ST4 (single structure) - 32-bit, immediate offset variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>0</td>
<td>01</td>
<td>ST4 (single structure) - 32-bit, immediate offset variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>0</td>
<td>01</td>
<td>ST4 (single structure) - 32-bit, immediate offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>010</td>
<td>-</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>011</td>
<td>-</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>100</td>
<td>-</td>
<td>1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>100</td>
<td>1</td>
<td>01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>101</td>
<td>-</td>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>101</td>
<td>1</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>110</td>
<td>1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>111</td>
<td>1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 1111</td>
<td>000</td>
<td>-</td>
<td>-</td>
<td>LD1 (single structure) - 8-bit, register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 1111</td>
<td>001</td>
<td>-</td>
<td>-</td>
<td>LD3 (single structure) - 8-bit, register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 1111</td>
<td>010</td>
<td>-</td>
<td>x0</td>
<td>LD1 (single structure) - 16-bit, register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 1111</td>
<td>011</td>
<td>-</td>
<td>x0</td>
<td>LD3 (single structure) - 16-bit, register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 1111</td>
<td>100</td>
<td>-</td>
<td>00</td>
<td>LD1 (single structure) - 32-bit, register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 1111</td>
<td>100</td>
<td>0</td>
<td>01</td>
<td>LD1 (single structure) - 64-bit, register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 1111</td>
<td>101</td>
<td>-</td>
<td>00</td>
<td>LD3 (single structure) - 32-bit, register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 1111</td>
<td>101</td>
<td>0</td>
<td>01</td>
<td>LD3 (single structure) - 64-bit, register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 1111</td>
<td>110</td>
<td>0</td>
<td>-</td>
<td>LD1R - Register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 1111</td>
<td>111</td>
<td>0</td>
<td>-</td>
<td>LD3R - Register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>000</td>
<td>-</td>
<td>-</td>
<td>LD1 (single structure) - 8-bit, immediate offset variant</td>
</tr>
<tr>
<td>L</td>
<td>R</td>
<td>Rm</td>
<td>opcode</td>
<td>S</td>
<td>size</td>
<td>Instruction page</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>----</td>
<td>--------</td>
<td>---</td>
<td>------</td>
<td>------------------</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>001</td>
<td>-</td>
<td>-</td>
<td>LD3 (single structure) - 8-bit, immediate offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>010</td>
<td>-</td>
<td>x0</td>
<td>LD1 (single structure) - 16-bit, immediate offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>011</td>
<td>-</td>
<td>x0</td>
<td>LD3 (single structure) - 16-bit, immediate offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>100</td>
<td>-</td>
<td>00</td>
<td>LD1 (single structure) - 32-bit, immediate offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>100</td>
<td>0</td>
<td>01</td>
<td>LD1 (single structure) - 64-bit, immediate offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>101</td>
<td>-</td>
<td>00</td>
<td>LD3 (single structure) - 32-bit, immediate offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>101</td>
<td>0</td>
<td>01</td>
<td>LD3 (single structure) - 64-bit, immediate offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>110</td>
<td>0</td>
<td>-</td>
<td>LD1R - Immediate offset variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>111</td>
<td>0</td>
<td>-</td>
<td>LD3R - Immediate offset variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>010</td>
<td>-</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>011</td>
<td>-</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>100</td>
<td>-</td>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>100</td>
<td>0</td>
<td>11</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>100</td>
<td>1</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>101</td>
<td>-</td>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>101</td>
<td>1</td>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>110</td>
<td>1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>111</td>
<td>1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>000</td>
<td>-</td>
<td>-</td>
<td>LD2 (single structure) - 8-bit, register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>001</td>
<td>-</td>
<td>-</td>
<td>LD4 (single structure) - 8-bit, register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>010</td>
<td>-</td>
<td>x0</td>
<td>LD2 (single structure) - 16-bit, register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>011</td>
<td>-</td>
<td>x0</td>
<td>LD4 (single structure) - 16-bit, register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>100</td>
<td>-</td>
<td>00</td>
<td>LD2 (single structure) - 32-bit, register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>100</td>
<td>0</td>
<td>01</td>
<td>LD2 (single structure) - 64-bit, register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>101</td>
<td>-</td>
<td>00</td>
<td>LD4 (single structure) - 32-bit, register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>101</td>
<td>0</td>
<td>01</td>
<td>LD4 (single structure) - 64-bit, register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>110</td>
<td>0</td>
<td>-</td>
<td>LD2R - Register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>111</td>
<td>0</td>
<td>-</td>
<td>LD4R - Register offset variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>000</td>
<td>-</td>
<td>-</td>
<td>LD2 (single structure) - 8-bit, immediate offset variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>001</td>
<td>-</td>
<td>-</td>
<td>LD4 (single structure) - 8-bit, immediate offset variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>010</td>
<td>-</td>
<td>x0</td>
<td>LD2 (single structure) - 16-bit, immediate offset variant</td>
</tr>
</tbody>
</table>
Load/store memory tags

This section describes the encoding of the Load/store memory tags instruction class. The encodings in this section are decoded from Loads and Stores on page C4-547.

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>Rm</th>
<th>opcode</th>
<th>S</th>
<th>size</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>011</td>
<td>-</td>
<td>x0</td>
<td>LD4 (single structure) - 16-bit, immediate offset variant</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>-</td>
<td>00</td>
<td>LD2 (single structure) - 32-bit, immediate offset variant</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>0</td>
<td>01</td>
<td>LD2 (single structure) - 64-bit, immediate offset variant</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>-</td>
<td>00</td>
<td>LD4 (single structure) - 32-bit, immediate offset variant</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>0</td>
<td>01</td>
<td>LD4 (single structure) - 64-bit, immediate offset variant</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>110</td>
<td>0</td>
<td>-</td>
<td>LD2R - Immediate offset variant</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>111</td>
<td>0</td>
<td>-</td>
<td>LD4R - Immediate offset variant</td>
<td></td>
</tr>
</tbody>
</table>

Load/store memory tags

This section describes the encoding of the Load/store memory tags instruction class. The encodings in this section are decoded from Loads and Stores on page C4-547.
Load/store exclusive pair

This section describes the encoding of the Load/store exclusive pair instruction class. The encodings in this section are decoded from Loads and Stores on page C4-547.

<table>
<thead>
<tr>
<th>opc</th>
<th>imm9</th>
<th>op2</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>-</td>
<td>11</td>
<td>STZ2G - Encoding</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>11</td>
<td>!= 000000000</td>
<td>00</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>000000000</td>
<td>00</td>
<td>LDGM</td>
<td>FEAT_MTE2</td>
</tr>
</tbody>
</table>

Load/store exclusive register

This section describes the encoding of the Load/store exclusive register instruction class. The encodings in this section are decoded from Loads and Stores on page C4-547.

<table>
<thead>
<tr>
<th>opc</th>
<th>imm9</th>
<th>op2</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>L</td>
<td>o0</td>
<td>STXRB</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>STXRB</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>STLXRB</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>LDXRB</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>LDAXP - 32-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>STXP - 64-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>STLXRB</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>LDXRB</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>LDAXP - 64-bit variant</td>
</tr>
</tbody>
</table>
**Load/store ordered**

This section describes the encoding of the Load/store ordered instruction class. The encodings in this section are decoded from *Loads and Stores on page C4-547.*

<table>
<thead>
<tr>
<th>size</th>
<th>L</th>
<th>o0</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>LDAXRB</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>STXRHB</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>STLXRHB</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>LDXRB</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDAXRB</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>STXR - 32-bit variant</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>STLXR - 32-bit variant</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>LDXR - 32-bit variant</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDAXR - 32-bit variant</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>STXR - 64-bit variant</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>STLXR - 64-bit variant</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>LDXR - 64-bit variant</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>LDAXR - 64-bit variant</td>
<td></td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>L</th>
<th>o0</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>STLLRB</td>
<td>FEAT_LOR</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>STLRB</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>LDLARB</td>
<td>FEAT_LOR</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>LDARB</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>STLLRH</td>
<td>FEAT_LOR</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>STLRH</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>LDLARH</td>
<td>FEAT_LOR</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDARH</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>STLLR - 32-bit variant</td>
<td>FEAT_LOR</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>STLR - 32-bit variant</td>
<td></td>
</tr>
</tbody>
</table>
Compare and swap

This section describes the encoding of the Compare and swap instruction class. The encodings in this section are decoded from "Loads and Stores" on page C4-547.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>L</td>
<td>o0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>L</td>
<td>o0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
LDAPR/STLR (unscaled immediate)

This section describes the encoding of the LDAPR/STLR (unscaled immediate) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-547.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>opc</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>size opc</td>
<td>STLURB</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>00 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 01</td>
<td>LDAPURB</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>00 10</td>
<td>LDAPURSB - 64-bit variant</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>00 11</td>
<td>LDAPURSB - 32-bit variant</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>01 00</td>
<td>STLURH</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>01 01</td>
<td>LDAPURH</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>01 10</td>
<td>LDAPURSH - 64-bit variant</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>01 11</td>
<td>LDAPURSH - 32-bit variant</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>10 00</td>
<td>STLUR - 32-bit variant</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>10 01</td>
<td>LDAPUR - 32-bit variant</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>10 10</td>
<td>LDAPURSW</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>10 11</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>11 00</td>
<td>STLUR - 64-bit variant</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>11 01</td>
<td>LDAPUR - 64-bit variant</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>11 10</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>11 11</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

Load register (literal)

This section describes the encoding of the Load register (literal) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-547.
Memory Copy and Memory Set

This section describes the encoding of the Memory Copy and Memory Set instruction class. The encodings in this section are decoded from Loads and Stores on page C4-547.

| 31 30 29 28|27 26 25 24|23 22 21 20| 16|15|12|11|10| 9 | 5 4 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| size | 0 | 1 | 1 | o0 | 0 | 0 | op1 | 0 | op2 | 0 | Rd |

**Decode fields**

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>LDR (literal) - 32-bit variant</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>LDR (literal, SIMD&amp;FP) - 32-bit variant</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>LDR (literal) - 64-bit variant</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>LDR (literal, SIMD&amp;FP) - 64-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>LDRSW (literal)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>LDR (literal, SIMD&amp;FP) - 128-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>PRFM (literal)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

**Instruction page**

- **CPYFP, CPYFM, CPYFE** - Prologue variant
- **CPYFPWT, CPYFMWT, CPYFEWT** - Prologue variant
- **CPYFPRT, CPYFMRT, CPYFERT** - Prologue variant
- **CPYFPT, CPYFMT, CPYFET** - Prologue variant
- **CPYFPWN, CPYFMWN, CPYFEWN** - Prologue variant
- **CPYFPWTWN, CPYFMWTWN, CPYFEWTWN** - Prologue variant
- **CPYFPRTN, CPYFMRN, CPYFERN** - Prologue variant
- **CPYFPWTRN, CPYFMWTRN, CPYFEWTRN** - Prologue variant
- **CPYFPRTN, CPYFMRTN, CPYFERTN** - Prologue variant
## A64 Instruction Set Encoding

### C4.1 A64 instruction set encoding

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>o0 op1 op2</td>
<td>CPYFPTRN, CPYFMTRN, CPYFETRN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 0 1011</td>
<td>CPYFPN, CPYFMN, CPYFEN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 0 1100</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 0 1101</td>
<td>CPYFPRTN, CPYFMRTN, CPYFERTN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 0 1110</td>
<td>CPYFPRTN, CPYFMRTN, CPYFERTN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 0 1111</td>
<td>CPYFPN, CPYFMN, CPYFEN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 0 1111</td>
<td>CPYFPN, CPYFMN, CPYFEN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 0 1111</td>
<td>CPYFPN, CPYFMN, CPYFEN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 0 1111</td>
<td>CPYFPN, CPYFMN, CPYFEN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 0000</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 0001</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 0010</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 0010</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 0011</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 0011</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 0100</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 0100</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 0101</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 0101</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 0110</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 0110</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 0111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 0111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1000</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1000</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1001</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1001</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1010</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1010</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1011</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1011</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1100</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1100</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1101</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1101</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1110</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1110</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 1 1111</td>
<td>CPYFPWTN, CPYFMWTON, CPYFEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>Decode fields</td>
<td>Instruction page</td>
<td>Feature</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------</td>
<td>---------</td>
</tr>
<tr>
<td>0 10 1011</td>
<td>CPYFPTRN, CPYFMTRN, CPYFETRN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 10 1100</td>
<td>CPYFPN, CPYFMN, CPYFEN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 10 1101</td>
<td>CPYFPWTN, CPYFMWTN, CPYFEWTN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 10 1110</td>
<td>CPYFPRTN, CPYFMRTN, CPYFERTN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 10 1111</td>
<td>CPYFPNTN, CPYFMNTN, CPYFETNTN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 11 0000</td>
<td>SETP, SETM, SETE - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 11 0001</td>
<td>SETPT, SETMT, SETET - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 11 0010</td>
<td>SETPN, SETMN, SETEN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 11 0011</td>
<td>SETPTN, SETMTN, SETETN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 11 0100</td>
<td>SETP, SETM, SETE - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 11 0101</td>
<td>SETPT, SETMT, SETET - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 11 0110</td>
<td>SETPN, SETMN, SETEN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 11 0111</td>
<td>SETPTN, SETMTN, SETETN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 11 1000</td>
<td>SETP, SETM, SETE - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 11 1001</td>
<td>SETPT, SETMT, SETET - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 11 1010</td>
<td>SETPN, SETMN, SETEN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 11 1011</td>
<td>SETPTN, SETMTN, SETETN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>0 11 11xx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1 00 0000</td>
<td>CPYP, CPYM, CPYE - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 00 0001</td>
<td>CPYPWT, CPYMWT, CPYEWT - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 00 0010</td>
<td>CPYPRT, CPYMR, CPYERT - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 00 0011</td>
<td>CPYPT, CPYMT, CPYET - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 00 0100</td>
<td>CPYPWN, CPYMWN, CPYWN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 00 0101</td>
<td>CPYPWTWN, CPYMWTN, CPYEWTN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 00 0110</td>
<td>CPYPRTWN, CPYMRWTN, CPYERTWN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 00 0111</td>
<td>CPYPTWN, CPYMTWN, CPYETWN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 00 1000</td>
<td>CPYPRN, CPYMRN, CPYERN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 00 1001</td>
<td>CPYPWTRN, CPYMWTRN, CPYEWTRN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 00 1010</td>
<td>CPYPRTRN, CPYMRTRN, CPYERTRN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 00 1011</td>
<td>CPYPRTRN, CPYMRTRN, CPYERTRN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 00 1100</td>
<td>CPYPN, CPYMN, CPYEN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 00 1101</td>
<td>CPYPWTN, CPYMWTN, CPYEWTN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>Decode fields</td>
<td>Instruction page</td>
<td>Feature</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------</td>
<td>---------</td>
</tr>
<tr>
<td>o0 op1 op2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00 1110</td>
<td>CPYPRTN, CPYMRTN, CPYERTN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 00 1111</td>
<td>CPYPTN, CPYMRTN, CPYERTN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 01 0000</td>
<td>CPYP, CPYM, CPYE - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 01 0001</td>
<td>CPYPWT, CPYMWT, CPYEWT - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 01 0010</td>
<td>CPYPRT, CPYMRT, CPYERT - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 01 0011</td>
<td>CPYPT, CPYMT, CPYET - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 01 0100</td>
<td>CPYPWN, CPYMWN, CPYEWN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 01 0101</td>
<td>CPYPWTWN, CPYMWTWN, CPYEWTWN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 01 0110</td>
<td>CPYPRTWN, CPYMRTWN, CPYERTWN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 01 0111</td>
<td>CPYPWTN, CPYMWTN, CPYEWTWN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 01 1000</td>
<td>CPYPRTN, CPYMRTN, CPYERTN - Prologue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 01 1001</td>
<td>CPYPWTN, CPYMWTN, CPYEWTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 01 1010</td>
<td>CPYPRTN, CPYMRTN, CPYERTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 01 1011</td>
<td>CPYPRTN, CPYMRTN, CPYERTN - Main variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 0000</td>
<td>CPYP, CPYM, CPYE - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 0001</td>
<td>CPYPWT, CPYMWT, CPYEWT - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 0010</td>
<td>CPYPRT, CPYMRT, CPYERT - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 0011</td>
<td>CPYPT, CPYMT, CPYET - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 0100</td>
<td>CPYPWN, CPYMWN, CPYEWN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 0101</td>
<td>CPYPWTWN, CPYMWTWN, CPYEWTWN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 0110</td>
<td>CPYPRTWN, CPYMRTWN, CPYERTWN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 0111</td>
<td>CPYPWTN, CPYMWTN, CPYEWTN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 1000</td>
<td>CPYPRTN, CPYMRTN, CPYERTN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 1001</td>
<td>CPYPWTN, CPYMWTN, CPYEWTN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 1010</td>
<td>CPYPRTN, CPYMRTN, CPYERTN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 1011</td>
<td>CPYPRTN, CPYMRTN, CPYERTN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 1100</td>
<td>CPYPN, CPYMN, CPYEN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 1101</td>
<td>CPYPWTN, CPYMWTN, CPYEWTN - Epilogue variant</td>
<td>FEAT_MOPS</td>
</tr>
</tbody>
</table>
Load/store no-allocate pair (offset)

This section describes the encoding of the Load/store no-allocate pair (offset) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-547.

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Load/store register pair (post-indexed)

This section describes the encoding of the Load/store register pair (post-indexed) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-547.

![Decode fields](image)

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>LDNP - 64-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>STNP (SIMD&amp;FP) - 128-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDNP (SIMD&amp;FP) - 128-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

Load/store register pair (offset)

This section describes the encoding of the Load/store register pair (offset) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-547.

![Decode fields](image)

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>STP - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>LDP - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>STP (SIMD&amp;FP) - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>LDP (SIMD&amp;FP) - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>STGP</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>LDPSW</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>STP (SIMD&amp;FP) - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDP (SIMD&amp;FP) - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>STP - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>LDP - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>STP (SIMD&amp;FP) - 128-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDP (SIMD&amp;FP) - 128-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
Load/store register pair (pre-indexed)

This section describes the encoding of the Load/store register pair (pre-indexed) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-547.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>1 0 1</td>
<td>V</td>
<td>0 1 0</td>
<td>L</td>
<td>imm7</td>
<td>Rt2</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc  V  L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 0 0</td>
<td>STP - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>00 0 1</td>
<td>LDP - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>00 1 0</td>
<td>STP (SIMD&amp;FP) - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>00 1 1</td>
<td>LDP (SIMD&amp;FP) - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>01 0 0</td>
<td>STGP</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>01 0 1</td>
<td>LDPSW</td>
<td>-</td>
</tr>
<tr>
<td>01 1 0</td>
<td>STP (SIMD&amp;FP) - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>01 1 1</td>
<td>LDP (SIMD&amp;FP) - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>10 0 0</td>
<td>STP - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>10 0 1</td>
<td>LDP - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>10 1 0</td>
<td>STP (SIMD&amp;FP) - 128-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>10 1 1</td>
<td>LDP (SIMD&amp;FP) - 128-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
## Load/store register (unscaled immediate)

This section describes the encoding of the Load/store register (unscaled immediate) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-547.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc V L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 1 0</td>
<td>STP (SIMD&amp;FP) - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>01 1 1</td>
<td>LDP (SIMD&amp;FP) - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>10 0 0</td>
<td>STP - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>10 0 1</td>
<td>LDP - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>10 1 0</td>
<td>STP (SIMD&amp;FP) - 128-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>10 1 1</td>
<td>LDP (SIMD&amp;FP) - 128-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>11 - -</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

### Load/store register (unscaled immediate)

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>imm9</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1</td>
<td>1</td>
<td>1x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>STURB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td>LDURB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>LDURSB - 64-bit variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>LDURSB - 32-bit variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td>STUR (SIMD&amp;FP) - 8-bit variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>LDUR (SIMD&amp;FP) - 8-bit variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
<td>STUR (SIMD&amp;FP) - 128-bit variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td>LDUR (SIMD&amp;FP) - 128-bit variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>STURH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>LDURH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>LDURSH - 64-bit variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>LDURSH - 32-bit variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>STUR (SIMD&amp;FP) - 16-bit variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td>LDUR (SIMD&amp;FP) - 16-bit variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>Unallocated.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Load/store register (immediate post-indexed)

This section describes the encoding of the Load/store register (immediate post-indexed) instruction class. The encodings in this section are decoded from `Loads and Stores on page C4-547`.

### Instruction page

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STUR - 32-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>LDUR - 32-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>LDURSW</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>STUR (SIMD&amp;FP) - 32-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>LDUR (SIMD&amp;FP) - 32-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STUR - 64-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDUR - 64-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>PRFUM</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>STUR (SIMD&amp;FP) - 64-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>LDUR (SIMD&amp;FP) - 64-bit variant</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>LDRSB (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>LDRSB (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) - 8-bit variant</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) - 8-bit variant</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
<td>STR (immediate, SIMD&amp;FP) - 128-bit variant</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td>LDR (immediate, SIMD&amp;FP) - 128-bit variant</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>LDRSH (immediate) - 64-bit variant</td>
</tr>
</tbody>
</table>
### Load/store register (unprivileged)

This section describes the encoding of the Load/store register (unprivileged) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-547.

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>LDRSH (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) - 16-bit variant</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) - 16-bit variant</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>LDRSW (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) - 32-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) - 32-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) - 64-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) - 64-bit variant</td>
</tr>
</tbody>
</table>
Load/store register (immediate pre-indexed)

This section describes the encoding of the Load/store register (immediate pre-indexed) instruction class. The encodings in this section are decoded from Loads and Stores on page C4-547.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>1 1 1 V</td>
<td>0 0</td>
<td>opc 0</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>size V opc</td>
<td></td>
</tr>
<tr>
<td>01 0 11</td>
<td>LDTRSH - 32-bit variant</td>
</tr>
<tr>
<td>1x 0 11</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10 0 00</td>
<td>STTR - 32-bit variant</td>
</tr>
<tr>
<td>10 0 01</td>
<td>LDTR - 32-bit variant</td>
</tr>
<tr>
<td>10 0 10</td>
<td>LDTRSW</td>
</tr>
<tr>
<td>11 0 00</td>
<td>STTR - 64-bit variant</td>
</tr>
<tr>
<td>11 0 01</td>
<td>LDTR - 64-bit variant</td>
</tr>
<tr>
<td>11 0 10</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

Decode fields

<table>
<thead>
<tr>
<th>size V opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1 1 1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00 0 00</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00 0 01</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00 0 10</td>
<td>LDRSB (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>00 0 11</td>
<td>LDRSB (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>00 1 00</td>
<td>STR (immediate, SIMD&amp;FP) - 8-bit variant</td>
</tr>
<tr>
<td>00 1 01</td>
<td>LDR (immediate, SIMD&amp;FP) - 8-bit variant</td>
</tr>
<tr>
<td>00 1 10</td>
<td>STR (immediate, SIMD&amp;FP) - 128-bit variant</td>
</tr>
<tr>
<td>00 1 11</td>
<td>LDR (immediate, SIMD&amp;FP) - 128-bit variant</td>
</tr>
<tr>
<td>01 0 00</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01 0 01</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>01 0 10</td>
<td>LDRSH (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>01 0 11</td>
<td>LDRSH (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>01 1 00</td>
<td>STR (immediate, SIMD&amp;FP) - 16-bit variant</td>
</tr>
<tr>
<td>01 1 01</td>
<td>LDR (immediate, SIMD&amp;FP) - 16-bit variant</td>
</tr>
</tbody>
</table>
## Atomic memory operations

This section describes the encoding of the Atomic memory operations instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-547.

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>LDRSW (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) - 32-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) - 32-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) - 64-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) - 64-bit variant</td>
</tr>
</tbody>
</table>

### Atomic memory operations

```
[31 30 29 28 27 26 25 24 23 22 21 20 16 15 14]   [12 11 10 9]   [5 4]   [0]
size 1 1 V 0 0 A R 1 | Rs o3 opc 0 0 Rn | Rt
```

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>o3</th>
<th>opc</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>11</td>
<td>x</td>
<td></td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td></td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>1</td>
<td></td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>0</td>
<td></td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
## Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>o 3</th>
<th>opc</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td></td>
<td>01</td>
<td>10</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td></td>
<td>00</td>
<td>10</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td></td>
<td>01</td>
<td>10</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td></td>
<td>01</td>
<td>10</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td></td>
<td>01</td>
<td>10</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td></td>
<td>01</td>
<td>10</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td></td>
<td>01</td>
<td>10</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td></td>
<td>01</td>
<td>10</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td></td>
<td>01</td>
<td>10</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td></td>
<td>01</td>
<td>10</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td></td>
<td>01</td>
<td>10</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td></td>
<td>01</td>
<td>10</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>LDADDB, LDADDAB, LDADDALB, LDADDB - LDADDLB - LDADB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>LDCLUDRB, LDCLUDRAB, LDCLUDRALB, LDCLUDRLB - LDCLUDRB variant</td>
<td>FEAT_LSE</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>Size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>O</th>
<th>3</th>
<th>Op</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>LDEORB, LDEORAB, LDEORALB, LDEORLB - LDEORB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>LDSETB, LDSETAB, LDSETALB, LDSETLB - LDSETB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB - LDSMAXB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB - LDSMINB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB - LDUMAXB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB - LDUMINB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>SWPB, SWPAB, SWPALB, SWPLB - SWPB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>Unallocated. -</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>Unallocated. -</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>1</td>
<td>0</td>
<td>Unallocated. -</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>Unallocated. -</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>LDADDB, LDADDAB, LDADDBL, LDADDB - LDADDB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB - LDCLRB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>LDEORB, LDEORAB, LDEORALB, LDEORLB - LDEORB variant</td>
<td>FEAT_LSE</td>
</tr>
</tbody>
</table>

**Note:**
- **VARR**: VARIANTS
- **pc**: pseudo counter
## Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>o</th>
<th>3</th>
<th>o pc</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>1</td>
<td>LDSETB, LDSETAB, LDSETALB, LDSETLB - LDSETLB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB - LDSMAXLB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>LDMINB, LDMINAB, LDMINALB, LDMINLB - LDMINLB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>LDMAXB, LDMAXAB, LDMAXALB, LDMAXLB - LDMAXLB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>LDMINB, LDMINAB, LDMINALB, LDMINLB - LDMINLB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>SWPB, SWPAB, SWPALB, SWPLB - SWPLB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>LDADDB, LDADDAB, LDADDALB, LDADDLB - LDADDAB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>1</td>
<td>LDCRLB, LDCRLAB, LDCRLALB, LDCRLLB - LDCRLAB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>LDEORB, LDEORAB, LDEORALB, LDEORLB - LDEORAB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>1</td>
<td>LDSETB, LDSETAB, LDSETALB, LDSETLB - LDSETAB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB - LDSMAXAB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>LDMINB, LDMINAB, LDMINALB, LDMINLB - LDMINAB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>LDMAXB, LDMAXAB, LDMAXALB, LDMAXLB - LDMAXAB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>LDMINB, LDMINAB, LDMINALB, LDMINLB - LDMINAB variant</td>
<td>FEAT_LSE</td>
</tr>
</tbody>
</table>
## Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>o 3</th>
<th>o pc</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td>SWPB, SWPAB, SWPALB, SWPLB - SWPAB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td>LDAPRB</td>
<td>FEAT_LRCPC</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td>00</td>
<td>1</td>
<td>LDADDB, LDADDAB, LDADDALB, LDADDLB - LDADDALB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td>00</td>
<td>1</td>
<td>LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB - LDCLRALB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td>00</td>
<td>1</td>
<td>LDEORB, LDEORAB, LDEORALB, LDEORLB - LDEORALB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td>01</td>
<td>0</td>
<td>LDSETB, LDSETAB, LDSETALB, LDSETLB - LDSETALB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td>10</td>
<td>0</td>
<td>LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB - LDSMAXALB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td>10</td>
<td>1</td>
<td>LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB - LDSMINALB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td>11</td>
<td>0</td>
<td>LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB - LDUMAXALB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td>11</td>
<td>1</td>
<td>LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB - LDUMINALB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td>SWPB, SWPAB, SWPALB, SWPLB - SWPALB variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td>00</td>
<td>1</td>
<td>LDADDH, LDADDAH, LDADDALH, LDADDLH - LDADDH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td>00</td>
<td>1</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH - LDCLRH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td>01</td>
<td>0</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH - LDEORALH variant</td>
<td>FEAT_LSE</td>
</tr>
</tbody>
</table>

C4-578 Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. Non-Confidential
### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>o3</th>
<th>o pc</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>LDSET, LDSETAH, LDSETALH, LDSETLH - LDSET variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH - LDSMAXH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH - LDSMINH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH - LDUMAXH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH - LDUMINH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>SWPH, SWPAH, SWPALH, SWPLH - SWPH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>LDADDDH, LDADDAH, LDADDALH, LDADDLH - LDADDLH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH - LDCLRLH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH - LDEORLH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>LDSET, LDSETAH, LDSETALH, LDSETLH - LDSETLH variant</td>
<td>FEAT_LSE</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>0</th>
<th>3</th>
<th>opc</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH - LDSMAXLH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH - LDSMINLH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH - LDUMAXLH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH - LDUMINLH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>SWPH, SWPAH, SWPALH, SWPLH - SWPLH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>LDADDH, LDADDAH, LDADDALH, LDADDLH - LDADDAH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH - LDCLRAH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH - LDEORAH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>LDSETH, LDSETAH, LDSETALH, LDSETLH - LDSETAH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH - LDSMAXAH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH - LDSMINAH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH - LDUMAXAH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH - LDUMINAH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>1</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>SWPH, SWPAH, SWPALH, SWPLH - SWPAH variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>Decode fields</td>
<td>Instruction page</td>
<td>Feature</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>------------------</td>
<td>---------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>01</strong></td>
<td>V A R Rs o 3 o pc</td>
<td>LDAPRH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>FEAT_LRCPC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>01</strong></td>
<td>V A R Rs o 3 o pc</td>
<td>LDADDH, LDADDAH, LDADDALH, LDADDLH - LDADDLH variant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>01</strong></td>
<td>V A R Rs o 3 o pc</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH - LDCLRALH variant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>01</strong></td>
<td>V A R Rs o 3 o pc</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH - LDEORALH variant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>01</strong></td>
<td>V A R Rs o 3 o pc</td>
<td>LDSETH, LDSETAH, LDSEATALH, LDSETLH - LDSEATALH variant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>01</strong></td>
<td>V A R Rs o 3 o pc</td>
<td>LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH - LDSMAXALH variant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>01</strong></td>
<td>V A R Rs o 3 o pc</td>
<td>LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH - LDSMINALH variant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>01</strong></td>
<td>V A R Rs o 3 o pc</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH - LDUMAXALH variant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>01</strong></td>
<td>V A R Rs o 3 o pc</td>
<td>LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH - LDUMINALH variant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>01</strong></td>
<td>V A R Rs o 3 o pc</td>
<td>SWPH, SWPAH, SWPALH, SWPLH - SWPALH variant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>10</strong></td>
<td>V A R Rs o 3 o pc</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL - 32-bit LDADD variant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>10</strong></td>
<td>V A R Rs o 3 o pc</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL - 32-bit LDCLR variant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>10</strong></td>
<td>V A R Rs o 3 o pc</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL - 32-bit LDEOR variant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>10</strong></td>
<td>V A R Rs o 3 o pc</td>
<td>LDSET, LDSETA, LDSEATAL, LDSETL - 32-bit LDSET variant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>o</th>
<th>pc</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>10</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL - 32-bit LDSMAX variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>10</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL - 32-bit LDSMIN variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>11</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL - 32-bit LDUMAX variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>11</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL - 32-bit LDUMIN variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>00</td>
<td>SWP, SWPA, SWPAL, SWPL - 32-bit SWP variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>01</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>01</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>01</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>10</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>00</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL - 32-bit LDADDL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>00</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL - 32-bit LDCLRL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>01</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL - 32-bit LDEORL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>01</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL - 32-bit LDSETL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>10</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL - 32-bit LDSMAXL variant</td>
<td>FEAT_LSE</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>Size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>Op</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL - 32-bit LDSMINL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL - 32-bit LDUMAXL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL - 32-bit LDUMINL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL - 32-bit LDADDA variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDCLR, LDCLA, LDCLAR, LDCLR - 32-bit LDCLR variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDEOR, LDEORA, LDEORAL, LDEOR - 32-bit LDEOR variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL - 32-bit LDSETA variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL - 32-bit LDSMAXA variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL - 32-bit LDSMINA variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL - 32-bit LDUMAXA variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL - 32-bit LDUMINA variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SWP, SWPA, SWPAL, SWPL - 32-bit SWPL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDAPR - 32-bit variant</td>
<td>FEAT_LRCPC</td>
</tr>
</tbody>
</table>
## Decode fields

<table>
<thead>
<tr>
<th>Size</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>Op</th>
<th>pc</th>
<th>Instruction</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL - 32-bit LDADDAL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL - 32-bit LDCLRL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL - 32-bit LDEORAL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL - 32-bit LDSETAL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL - 32-bit LDSMAXAL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL - 32-bit LDSMINAL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL - 32-bit LDUMAXAL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL - 32-bit LDUMINAL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>SWP, SWPA, SWPAL, SWPL - 32-bit SWPAL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL - 64-bit LDADD variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL - 64-bit LDCLRL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL - 64-bit LDEOR variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL - 64-bit LDSET variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL - 64-bit LDSMAX variant</td>
<td>FEAT_LSE</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>o 3</th>
<th>o pc</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL - 64-bit LDSMIN variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL - 64-bit LDUMAX variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL - 64-bit LDUMIN variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SWP, SWPA, SWPAL, SWPL - 64-bit SWP variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>ST64BV0</td>
<td>FEAT_LS64_ACC DATA</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>1</td>
<td>ST64BV</td>
<td>FEAT_LS64_V</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1111</td>
<td>00</td>
<td>ST64B</td>
<td>FEAT_LS64</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1111</td>
<td>10</td>
<td>LD64B</td>
<td>FEAT_LS64</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL - 64-bit LDADDL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL - 64-bit LDCLRL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL - 64-bit LDEORL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL - 64-bit LDSETL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL - 64-bit LDSMAXL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL - 64-bit LDSMINL variant</td>
<td>FEAT_LSE</td>
</tr>
</tbody>
</table>
## Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>o3</th>
<th>o pc</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL - 64-bit LDUMAXL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL - 64-bit LDUMINL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>-</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>SWP, SWPA, SWPAL, SWPL - 64-bit SWPL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL - 64-bit LDADDA variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL - 64-bit LDCLRA variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL - 64-bit LDEORA variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL - 64-bit LDSETA variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL - 64-bit LDSMAXA variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL - 64-bit LDSMINA variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL - 64-bit LDUMAXA variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL - 64-bit LDUMINA variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>-</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>SWP, SWPA, SWPAL, SWPL - 64-bit SWPA variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>-</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>LDAPR - 64-bit variant</td>
<td>FEAT_LRCPC</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL - 64-bit LDADDA variant</td>
<td>FEAT_LSE</td>
</tr>
</tbody>
</table>

**C4-586**

*Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. ARM DDI 0487I.a*

*Non-Confidential*

ID081822
Load/store register (register offset)

This section describes the encoding of the Load/store register (register offset) instruction class. The encodings in this section are decoded from Loads and Stores on page C4-547.

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>o3</th>
<th>opc</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>LDCLR, LDCLRA, LDCLRL - 64-bit LDCLRL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL - 64-bit LDEORAL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL - 64-bit LDSETL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL - 64-bit LDSMAXAL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL - 64-bit LDSMINAL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL - 64-bit LDUMAXAL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>0</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL - 64-bit LDUMINAL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>SWP, SWPA, SWPAL, SWPL - 64-bit SWPAL variant</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>00</td>
<td>Unallocated.</td>
<td></td>
</tr>
</tbody>
</table>

Decoding fields

- size: 11 0 1 1
- V: 0
- A: 0
- R: 0
- Rs: 1
- o3: 0
- opc: 00
- Unallocated.
- feature: FEAT_LSE
### Load/store register (pac)

This section describes the encoding of the Load/store register (pac) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-547.

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>option</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>! = 011</td>
<td>LDRSB (register) - 64-bit with extended register offset variant</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>011</td>
<td>LDRSB (register) - 64-bit with shifted register offset variant</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>! = 011</td>
<td>LDRSB (register) - 32-bit with extended register offset variant</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>011</td>
<td>LDRSB (register) - 32-bit with shifted register offset variant</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td>! = 011</td>
<td>STR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td>011</td>
<td>STR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>! = 011</td>
<td>LDR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>011</td>
<td>LDR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>-</td>
<td>STRH (register)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>-</td>
<td>LDRH (register)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>-</td>
<td>LDRSH (register) - 64-bit variant</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>-</td>
<td>LDRSH (register) - 32-bit variant</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>-</td>
<td>STR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td>-</td>
<td>LDR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>-</td>
<td>STR (register) - 32-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>-</td>
<td>LDR (register) - 32-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>-</td>
<td>LDRSW (register)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>-</td>
<td>STR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>-</td>
<td>LDR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>-</td>
<td>STR (register) - 64-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>-</td>
<td>LDR (register) - 64-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>-</td>
<td>PRFM (register)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>-</td>
<td>STR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>-</td>
<td>LDR (register, SIMD&amp;FP)</td>
</tr>
</tbody>
</table>
### Load/store register (unsigned immediate)

This section describes the encoding of the Load/store register (unsigned immediate) instruction class. The encodings in this section are decoded from *Loads and Stores* on page C4-547.

```
<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[12 11 10 9]</th>
<th>[5 4]</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size V M W imm9</td>
<td>W Rn Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>size V M W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!= 11 - - -</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>11 0 0 0</td>
<td>LDRAA, LDRAB - Key A, offset variant</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>11 0 0 1</td>
<td>LDRAA, LDRAB - Key A, pre-indexed variant</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>11 0 1 0</td>
<td>LDRAA, LDRAB - Key B, offset variant</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>11 0 1 1</td>
<td>LDRAA, LDRAB - Key B, pre-indexed variant</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>11 1 - -</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

```
| [31 30 29 28] | [27 26 25 24] | [23 22 21] | [10 9] | [5 4] | 0 |
|----------------|----------------|-------------|------|-----|
| size V opc imm12 | Rn Rt |
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>size V opc</td>
<td></td>
</tr>
<tr>
<td>x1 1 1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00 0 00</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00 0 01</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00 0 10</td>
<td>LDRSB (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>00 0 11</td>
<td>LDRSB (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>00 1 00</td>
<td>STR (immediate, SIMD&amp;FP) - 8-bit variant</td>
</tr>
<tr>
<td>00 1 01</td>
<td>LDR (immediate, SIMD&amp;FP) - 8-bit variant</td>
</tr>
<tr>
<td>00 1 10</td>
<td>STR (immediate, SIMD&amp;FP) - 128-bit variant</td>
</tr>
<tr>
<td>00 1 11</td>
<td>LDR (immediate, SIMD&amp;FP) - 128-bit variant</td>
</tr>
<tr>
<td>01 0 00</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01 0 01</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>01 0 10</td>
<td>LDRSH (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>01 0 11</td>
<td>LDRSH (immediate) - 32-bit variant</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) - 16-bit variant</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) - 16-bit variant</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>LDRSW (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) - 32-bit variant</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) - 32-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>PRFM (immediate)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) - 64-bit variant</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) - 64-bit variant</td>
</tr>
</tbody>
</table>

#### C4.1.68 Data Processing -- Register

This section describes the encoding of the Data Processing -- Register group. The encodings in this section are decoded from *A64 instruction set encoding* on page C4-388.

![Table C4-69 Encoding table for the Data Processing -- Register group](image)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0110</td>
<td>-</td>
<td>Data-processing (2 source) on page C4-591</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0110</td>
<td>-</td>
<td>Data-processing (1 source) on page C4-592</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0xx</td>
<td>-</td>
<td>Logical (shifted register) on page C4-594</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>1xx0</td>
<td>-</td>
<td>Add/subtract (shifted register) on page C4-595</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>1xx1</td>
<td>-</td>
<td>Add/subtract (extended register) on page C4-596</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>00000</td>
<td>000000</td>
<td>Add/subtract (with carry) on page C4-597</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>0000</td>
<td>x00001</td>
<td>Rotate right into flags on page C4-597</td>
</tr>
</tbody>
</table>
This section describes the encoding of the Data-processing (2 source) instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-590.

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>000001</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>011xxx</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1xxxxx</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0011x</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>001101</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>00111x</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>00001x</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0001xx</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>001xxx</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>000000</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>000010</td>
<td>UDIV - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>000011</td>
<td>SDIV - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>00010x</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>001000</td>
<td>LSLV - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>001001</td>
<td>LSRV - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>001010</td>
<td>ASRV - 32-bit variant</td>
<td>-</td>
</tr>
</tbody>
</table>
Data-processing (1 source)

This section describes the encoding of the Data-processing (1 source) instruction class. The encodings in this section are decoded from Data Processing -- Register on page C4-590.
### Decode fields

<table>
<thead>
<tr>
<th>s</th>
<th>f</th>
<th>S</th>
<th>opcode2</th>
<th>opcode</th>
<th>Rn</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1xxxxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>xxx1x</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>xx1xx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x1xxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>lxxxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>00000</td>
<td>00011x</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>00000</td>
<td>001xxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>00000</td>
<td>01xxx</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>00001</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000</td>
<td>00000</td>
<td>-</td>
<td>-</td>
<td>RBIT - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000</td>
<td>000000</td>
<td>-</td>
<td>-</td>
<td>REV16 - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000</td>
<td>000010</td>
<td>-</td>
<td>-</td>
<td>REV - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000</td>
<td>000011</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000</td>
<td>000100</td>
<td>-</td>
<td>-</td>
<td>CLZ - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000</td>
<td>000101</td>
<td>-</td>
<td>-</td>
<td>CLS - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000000</td>
<td>-</td>
<td>-</td>
<td>RBIT - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000001</td>
<td>-</td>
<td>-</td>
<td>REV16 - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000010</td>
<td>-</td>
<td>-</td>
<td>REV32</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000011</td>
<td>-</td>
<td>-</td>
<td>REV - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000100</td>
<td>-</td>
<td>-</td>
<td>CLZ - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000101</td>
<td>-</td>
<td>-</td>
<td>CLS - 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>000000</td>
<td>-</td>
<td>-</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA - PACIA variant</td>
<td>FEAT_PAult</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>000001</td>
<td>-</td>
<td>-</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB - PACIB variant</td>
<td>FEAT_PAult</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>000010</td>
<td>-</td>
<td>-</td>
<td>PACDA, PACDZA - PACDA variant</td>
<td>FEAT_PAult</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>000011</td>
<td>-</td>
<td>-</td>
<td>PACDB, PACDZB - PACDB variant</td>
<td>FEAT_PAult</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>opcode 2</th>
<th>opcode</th>
<th>Rn</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>00100</td>
<td>-</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA - AUTIA variant</td>
<td>FEAT_PAut</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>00101</td>
<td>-</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB - AUTIB variant</td>
<td>FEAT_PAut</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>00110</td>
<td>-</td>
<td>AUTDA, AUTDZA - AUTDA variant</td>
<td>FEAT_PAut</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>00111</td>
<td>-</td>
<td>AUTDB, AUTDZB - AUTDB variant</td>
<td>FEAT_PAut</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>00100</td>
<td>1111</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA - PACIZA variant</td>
<td>FEAT_PAut</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>00101</td>
<td>1111</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB - PACIZB variant</td>
<td>FEAT_PAut</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>00101</td>
<td>1111</td>
<td>PACDA, PACDZA - PACDZA variant</td>
<td>FEAT_PAut</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>00111</td>
<td>1111</td>
<td>PACDB, PACDZB - PACDZB variant</td>
<td>FEAT_PAut</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>00110</td>
<td>1111</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA - AUTIZA variant</td>
<td>FEAT_PAut</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>00110</td>
<td>1111</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB - AUTIZB variant</td>
<td>FEAT_PAut</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>00111</td>
<td>1111</td>
<td>AUTDA, AUTDZA - AUTDA variant</td>
<td>FEAT_PAut</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>00111</td>
<td>1111</td>
<td>AUTDB, AUTDZB - AUTDB variant</td>
<td>FEAT_PAut</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>01000</td>
<td>1111</td>
<td>XPACD, XPACI, XPACLRI - XPACI variant</td>
<td>FEAT_PAut</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>01001</td>
<td>1111</td>
<td>XPACD, XPACI, XPACLRI - XPACD variant</td>
<td>FEAT_PAut</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>0101x</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>011xx</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>011xx</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

### Logical (shifted register)

This section describes the encoding of the Logical (shifted register) instruction class. The encodings in this section are decoded from *Data Processing -- Register on page C4-590.*
Add/subtract (shifted register)

This section describes the encoding of the Add/subtract (shifted register) instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-590.
Add/subtract (extended register)

This section describes the encoding of the Add/subtract (extended register) instruction class. The encodings in this section are decoded from Data Processing -- Register on page C4-590.

### Instruction page

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>op</td>
<td>S</td>
<td>shift</td>
<td>imm6</td>
<td>Rm</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>shift</th>
<th>imm6</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>11</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1xxxxx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 0 0 -</td>
<td>-</td>
<td>ADD (shifted register) - 32-bit variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 -</td>
<td>-</td>
<td>ADDS (shifted register) - 32-bit variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 -</td>
<td>-</td>
<td>SUB (shifted register) - 32-bit variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 -</td>
<td>-</td>
<td>SUBS (shifted register) - 32-bit variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 -</td>
<td>-</td>
<td>ADD (shifted register) - 64-bit variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 -</td>
<td>-</td>
<td>ADDS (shifted register) - 64-bit variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 -</td>
<td>-</td>
<td>SUB (shifted register) - 64-bit variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 -</td>
<td>-</td>
<td>SUBS (shifted register) - 64-bit variant</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction page

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15</th>
<th>13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>op</td>
<td>S</td>
<td>opt</td>
<td>imm3</td>
<td>Rm</td>
<td>option</td>
<td>imm3</td>
<td>Rn</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>opt</th>
<th>imm3</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>11x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1x</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 0 0 00</td>
<td>-</td>
<td>ADD (extended register) - 32-bit variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 00</td>
<td>-</td>
<td>ADDS (extended register) - 32-bit variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 00</td>
<td>-</td>
<td>SUB (extended register) - 32-bit variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 00</td>
<td>-</td>
<td>SUBS (extended register) - 32-bit variant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 00</td>
<td>-</td>
<td>ADD (extended register) - 64-bit variant</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Add/subtract (with carry)

This section describes the encoding of the Add/subtract (with carry) instruction class. The encodings in this section are decoded from Data Processing -- Register on page C4-590.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>opt</th>
<th>imm3</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>-</td>
<td>ADDS (extended register) - 64-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>-</td>
<td>SUB (extended register) - 64-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>00</td>
<td>-</td>
<td>SUBS (extended register) - 64-bit variant</td>
</tr>
</tbody>
</table>

Rotate right into flags

This section describes the encoding of the Rotate right into flags instruction class. The encodings in this section are decoded from Data Processing -- Register on page C4-590.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>opt</th>
<th>imm3</th>
<th>o2</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td>sf</td>
<td>op</td>
<td>S</td>
<td>opt</td>
<td>imm3</td>
<td>o2</td>
<td>mask</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
</tr>
</tbody>
</table>

Unallocated
Evaluate into flags

This section describes the encoding of the Evaluate into flags instruction class. The encodings in this section are decoded from Data Processing -- Register on page C4-590.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>opcode2</th>
<th>sz</th>
<th>o3</th>
<th>mask</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>RMIF</td>
<td>FEAT_FlagM</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Conditional compare (register)

This section describes the encoding of the Conditional compare (register) instruction class. The encodings in this section are decoded from Data Processing -- Register on page C4-590.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>opcode2</th>
<th>sz</th>
<th>o2</th>
<th>Rn</th>
<th>o3</th>
<th>mask</th>
<th>cond</th>
<th>nzcv</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
<td>-</td>
<td>000000</td>
<td>SETF8, SETF16 - SETF8 variant FEAT_FlagM</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>!= 00000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00000</td>
<td>-</td>
<td>0</td>
<td>!= 1101</td>
<td>Unallocated.</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00000</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00000</td>
<td>0</td>
<td>0</td>
<td>1101</td>
<td>SETF8, SETF16 - SETF8 variant</td>
<td>FEAT_FlagM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00000</td>
<td>1</td>
<td>0</td>
<td>1101</td>
<td>SETF8, SETF16 - SETF16 variant</td>
<td>FEAT_FlagM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conditional compare (immediate)

This section describes the encoding of the Conditional compare (immediate) instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-590.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>o3</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMN (register) - 32-bit variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMP (register) - 32-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMN (register) - 64-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMP (register) - 64-bit variant</td>
</tr>
</tbody>
</table>

 Conditional select

This section describes the encoding of the Conditional select instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-590.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>o3</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMN (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMP (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMN (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMP (immediate) - 64-bit variant</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>o3</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMN (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMP (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMN (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMP (immediate) - 64-bit variant</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>o3</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMN (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMP (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMN (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMP (immediate) - 64-bit variant</td>
</tr>
</tbody>
</table>

Conditional select

This section describes the encoding of the Conditional select instruction class. The encodings in this section are decoded from *Data Processing -- Register* on page C4-590.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>o3</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMN (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMP (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMN (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMP (immediate) - 64-bit variant</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>o3</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMN (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMP (immediate) - 32-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMN (immediate) - 64-bit variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMP (immediate) - 64-bit variant</td>
</tr>
</tbody>
</table>
Data-processing (3 source)

This section describes the encoding of the Data-processing (3 source) instruction class. The encodings in this section are decoded from *Data Processing -- Register on page C4-590.*
### C4.1.69 Data Processing -- Scalar Floating-Point and Advanced SIMD

This section describes the encoding of the Data Processing -- Scalar Floating-Point and Advanced SIMD group. The encodings in this section are decoded from *A64 instruction set encoding* on page C4-388.

<table>
<thead>
<tr>
<th>[31] 28</th>
<th>27</th>
<th>25 24</th>
<th>23 22</th>
<th>19 18</th>
<th>10 9</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>111</td>
<td>op1</td>
<td>op2</td>
<td>op3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Table C4-70 Encoding table for the Data Processing -- Scalar Floating-Point and Advanced SIMD group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0x x101</td>
<td>00xxxx10 Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0010 0x x101</td>
<td>00xxxx10 Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0100 0x x101</td>
<td>00xxxx10 Cryptographic AES on page C4-603</td>
<td>-</td>
</tr>
<tr>
<td>0101 0x x0xx</td>
<td>xxx0xxxx00 Cryptographic three-register SHA on page C4-604</td>
<td>-</td>
</tr>
<tr>
<td>0101 0x x0xx</td>
<td>xxx0xxxx10 Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0101 0x x101</td>
<td>00xxxx10 Cryptographic two-register SHA on page C4-604</td>
<td>-</td>
</tr>
<tr>
<td>0110 0x x101</td>
<td>00xxxx10 Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0111 0x x0xx</td>
<td>xxx0xxxx0 Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0111 0x x101</td>
<td>00xxxx10 Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>01x1 00 00xx</td>
<td>xxx0xxxx1 Advanced SIMD scalar copy on page C4-605</td>
<td>-</td>
</tr>
<tr>
<td>01x1 01 00xx</td>
<td>xxx0xxxx1 Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>01x1 0x 0111</td>
<td>00xxxx10 Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>01x1 0x 10xx</td>
<td>xxx0xxxx1 Advanced SIMD scalar three same FP16 on page C4-605</td>
<td>-</td>
</tr>
<tr>
<td>01x1 0x 10xx</td>
<td>xxx01xxx1 Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
### Table C4-70 Encoding table for the Data Processing -- Scalar Floating-Point and Advanced SIMD group (continued)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2 op3</td>
<td>Decode group or instruction page</td>
</tr>
<tr>
<td>01x1 0x 1111 00xxxxx10</td>
<td>Advanced SIMD scalar two-register miscellaneous FP16 on page C4-606</td>
</tr>
<tr>
<td>01x1 0x x0xx xxx1xxxx0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01x1 0x x0xx xxx1xxxx1</td>
<td>Advanced SIMD three same extra on page C4-608</td>
</tr>
<tr>
<td>01x1 0x x100 00xxxxx10</td>
<td>Advanced SIMD scalar two-register miscellaneous on page C4-608</td>
</tr>
<tr>
<td>01x1 0x x110 00xxxxx10</td>
<td>Advanced SIMD scalar pairwise on page C4-610</td>
</tr>
<tr>
<td>01x1 0x x1xx 1xxxxxx10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01x1 0x x1xx x1xxxxx10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01x1 0x x1xx xxxxxxx0</td>
<td>Advanced SIMD scalar three different on page C4-611</td>
</tr>
<tr>
<td>01x1 0x x1xx xxxxxxx1</td>
<td>Advanced SIMD scalar three same on page C4-612</td>
</tr>
<tr>
<td>01x1 10 - xxxxxxxx1</td>
<td>Advanced SIMD scalar shift by immediate on page C4-614</td>
</tr>
<tr>
<td>01x1 11 - xxxxxxxx1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01x1 1x - xxxxxxxx0</td>
<td>Advanced SIMD scalar x indexed element on page C4-616</td>
</tr>
<tr>
<td>0x00 0x x0xx xxx0xxxxx0</td>
<td>Advanced SIMD table lookup on page C4-617</td>
</tr>
<tr>
<td>0x00 0x x0xx xxx0xxxxx10</td>
<td>Advanced SIMD permute on page C4-618</td>
</tr>
<tr>
<td>0x10 0x x0xx xxxxxxx0</td>
<td>Advanced SIMD extract on page C4-618</td>
</tr>
<tr>
<td>0x00 00 00xx xxx0xxxx1</td>
<td>Advanced SIMD copy on page C4-619</td>
</tr>
<tr>
<td>0x00 01 00xx xxx0xxxx1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0x00 0x 0111 00xxxxx10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0x00 0x x0xx xxx0xxxxx1</td>
<td>Advanced SIMD three same (FP16) on page C4-619</td>
</tr>
<tr>
<td>0x00 0x x0xx xxx01xxxx1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0x00 0x 1111 00xxxxx10</td>
<td>Advanced SIMD two-register miscellaneous (FP16) on page C4-621</td>
</tr>
<tr>
<td>0x00 0x x0xx xxx1xxxx0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0x00 0x x0xx xxx1xxxx1</td>
<td>Advanced SIMD three-register extension on page C4-622</td>
</tr>
<tr>
<td>0x00 0x x100 00xxxxx10</td>
<td>Advanced SIMD two-register miscellaneous on page C4-624</td>
</tr>
<tr>
<td>0x00 0x x110 00xxxxx10</td>
<td>Advanced SIMD across lanes on page C4-626</td>
</tr>
<tr>
<td>0x00 0x x1xx 1xxxxxx10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0x00 0x x1xx x1xxxxx10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0x00 0x x1xx xxxxxxx0</td>
<td>Advanced SIMD three different on page C4-628</td>
</tr>
<tr>
<td>0x00 0x x1xx xxxxxxx1</td>
<td>Advanced SIMD three same on page C4-629</td>
</tr>
<tr>
<td>0x00 10 0000 xxxxxxxx1</td>
<td>Advanced SIMD modified immediate on page C4-632</td>
</tr>
</tbody>
</table>
Table C4-70 Encoding table for the Data Processing -- Scalar Floating-Point and Advanced SIMD group (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Decode group or instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xx0</td>
<td>10</td>
<td>!= 0000</td>
<td>x0xxxx</td>
<td>Advanced SIMD shift by immediate on page C4-633</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>11</td>
<td>-</td>
<td>x0xxxx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>1x</td>
<td>-</td>
<td>x0xxxx</td>
<td>Advanced SIMD vector x indexed element on page C4-635</td>
<td>-</td>
</tr>
<tr>
<td>1100</td>
<td>00</td>
<td>10xx</td>
<td>xxx10xxxx</td>
<td>Cryptographic three-register, imm2 on page C4-637</td>
<td>-</td>
</tr>
<tr>
<td>1100</td>
<td>00</td>
<td>11xx</td>
<td>xxx1x00xx</td>
<td>Cryptographic three-register SHA 512 on page C4-637</td>
<td>-</td>
</tr>
<tr>
<td>1100</td>
<td>00</td>
<td>-</td>
<td>xxx0xxxx</td>
<td>Cryptographic four-register on page C4-638</td>
<td>-</td>
</tr>
<tr>
<td>1100</td>
<td>01</td>
<td>00xx</td>
<td>-</td>
<td>XAR</td>
<td>FEAT_SHA3</td>
</tr>
<tr>
<td>1100</td>
<td>01</td>
<td>1000</td>
<td>0001000xx</td>
<td>Cryptographic two-register SHA 512 on page C4-638</td>
<td>-</td>
</tr>
<tr>
<td>1xx0</td>
<td>1x</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>x0x1</td>
<td>0x</td>
<td>x0xx</td>
<td>-</td>
<td>Conversion between floating-point and fixed-point on page C4-639</td>
<td>-</td>
</tr>
<tr>
<td>x0x1</td>
<td>0x</td>
<td>x1xx</td>
<td>xxx000000</td>
<td>Conversion between floating-point and integer on page C4-640</td>
<td>-</td>
</tr>
<tr>
<td>x0x1</td>
<td>0x</td>
<td>x1xx</td>
<td>xxx100000</td>
<td>Floating-point data-processing (1 source) on page C4-644</td>
<td>-</td>
</tr>
<tr>
<td>x0x1</td>
<td>0x</td>
<td>x1xx</td>
<td>xxx110000</td>
<td>Floating-point compare on page C4-646</td>
<td>-</td>
</tr>
<tr>
<td>x0x1</td>
<td>0x</td>
<td>x1xx</td>
<td>xxx001000</td>
<td>Floating-point immediate on page C4-647</td>
<td>-</td>
</tr>
<tr>
<td>x0x1</td>
<td>0x</td>
<td>x1xx</td>
<td>xxx000001</td>
<td>Floating-point conditional compare on page C4-647</td>
<td>-</td>
</tr>
<tr>
<td>x0x1</td>
<td>0x</td>
<td>x1xx</td>
<td>xxxxxxx10</td>
<td>Floating-point data-processing (2 source) on page C4-648</td>
<td>-</td>
</tr>
<tr>
<td>x0x1</td>
<td>0x</td>
<td>x1xx</td>
<td>xxxxxxx11</td>
<td>Floating-point conditional select on page C4-649</td>
<td>-</td>
</tr>
<tr>
<td>x0x1</td>
<td>1x</td>
<td>-</td>
<td>-</td>
<td>Floating-point data-processing (3 source) on page C4-650</td>
<td>-</td>
</tr>
</tbody>
</table>

Cryptographic AES

This section describes the encoding of the Cryptographic AES instruction class. The encodings in this section are decoded from Data Processing -- Scalar Floating-Point and Advanced SIMD on page C4-601.

<table>
<thead>
<tr>
<th>opcode</th>
<th>size</th>
<th>Rd</th>
<th>Rn</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01001110</td>
<td>10100</td>
<td>10</td>
<td>11</td>
<td>9</td>
</tr>
</tbody>
</table>

Decoding fields

<table>
<thead>
<tr>
<th>opcode</th>
<th>size</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Unallocated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Unallocated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
Cryptographic three-register SHA

This section describes the encoding of the Cryptographic three-register SHA instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>size opcode</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>00 00100</td>
<td>AESE</td>
<td>FEAT_AES</td>
</tr>
<tr>
<td>00 00101</td>
<td>AESD</td>
<td>FEAT_AES</td>
</tr>
<tr>
<td>00 00110</td>
<td>AESMC</td>
<td>-</td>
</tr>
<tr>
<td>00 00111</td>
<td>AESIMC</td>
<td>-</td>
</tr>
<tr>
<td>1x Unallocated.</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

Cryptographic two-register SHA

This section describes the encoding of the Cryptographic two-register SHA instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>size opcode</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>- 111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>x1 Unallocated.</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>00 000</td>
<td>SHA1C</td>
<td>FEAT_SHA1</td>
</tr>
<tr>
<td>00 001</td>
<td>SHA1P</td>
<td>FEAT_SHA1</td>
</tr>
<tr>
<td>00 010</td>
<td>SHA1M</td>
<td>FEAT_SHA1</td>
</tr>
<tr>
<td>00 011</td>
<td>SHA1SU0</td>
<td>FEAT_SHA1</td>
</tr>
<tr>
<td>00 100</td>
<td>SHA256H</td>
<td>FEAT_SHA256</td>
</tr>
<tr>
<td>00 101</td>
<td>SHA256H2</td>
<td>FEAT_SHA256</td>
</tr>
<tr>
<td>00 110</td>
<td>SHA256SU1</td>
<td>FEAT_SHA256</td>
</tr>
<tr>
<td>1x Unallocated.</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
Advanced SIMD scalar copy

This section describes the encoding of the Advanced SIMD scalar copy instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

Advanced SIMD scalar three same FP16

This section describes the encoding of the Advanced SIMD scalar three same FP16 instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.
Advanced SIMD scalar two-register miscellaneous FP16

This section describes the encoding of the Advanced SIMD scalar two-register miscellaneous FP16 instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.
## Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>a</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>00xxx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>010xx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>10xxx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>1100x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>11110</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>011xx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>11111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>01111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>11100</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11010</td>
<td>FCVTNS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11011</td>
<td>FCVTMS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11100</td>
<td>FCVTAS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11101</td>
<td>SCVTF (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01100</td>
<td>FCMGT (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01101</td>
<td>FCMEQ (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01110</td>
<td>FCMLT (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11010</td>
<td>FCVTPS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11011</td>
<td>FCVTZS (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11100</td>
<td>FRECPE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>FRECPX</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11010</td>
<td>FCVTNU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11011</td>
<td>FCVTMU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11100</td>
<td>FCVTAU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11101</td>
<td>UCVTF (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01100</td>
<td>FCMGE (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01101</td>
<td>FCMLE (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01110</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11010</td>
<td>FCVTPU (vector)</td>
<td>FEAT_FP16</td>
</tr>
</tbody>
</table>
Advanced SIMD scalar three same extra

This section describes the encoding of the Advanced SIMD scalar three same extra instruction class. The encodings in this section are decoded from Data Processing -- Scalar Floating-Point and Advanced SIMD on page C4-601.

Advanced SIMD scalar two-register miscellaneous

This section describes the encoding of the Advanced SIMD scalar two-register miscellaneous instruction class. The encodings in this section are decoded from Data Processing -- Scalar Floating-Point and Advanced SIMD on page C4-601.
### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>00110</td>
<td>Unallocated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01111</td>
<td>Unallocated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10101</td>
<td>Unallocated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10111</td>
<td>Unallocated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1100x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11110</td>
<td>Unallocated.</td>
</tr>
<tr>
<td></td>
<td>0x</td>
<td>011xx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td></td>
<td>1x</td>
<td>11100</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>00011</td>
<td>SUQADD</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>00111</td>
<td>SQABS</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>01000</td>
<td>CMGT (zero)</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>01001</td>
<td>CMEQ (zero)</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>01010</td>
<td>CMLT (zero)</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>01011</td>
<td>ABS</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>10010</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>10100</td>
<td>SQXTN, SQXTN2</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>10110</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11010</td>
<td>FCVTNS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11011</td>
<td>FCVTMS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11100</td>
<td>FCVTAS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11101</td>
<td>SCVTF (vector, integer)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01100</td>
<td>FCMGT (zero)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01101</td>
<td>FCMEQ (zero)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01110</td>
<td>FCMLT (zero)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>10100</td>
<td>FCVTPS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>10111</td>
<td>FCVTZS (vector, integer)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11101</td>
<td>FRECPE</td>
</tr>
</tbody>
</table>
| 0 | 1x   | 11111  | FRECPX
This section describes the encoding of the Advanced SIMD scalar pairwise instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

### Advanced SIMD scalar pairwise

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 U 1 1 1 1 0 size 1 1 0 0 0 opcode 1 0</td>
<td>Rn Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>0011</td>
<td>USQADD</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>00111</td>
<td>SQNEG</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>01000</td>
<td>CMGE (zero)</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>01001</td>
<td>CMLE (zero)</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>01010</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>01011</td>
<td>NEG (vector)</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>10010</td>
<td>SQXTUN, SQXTUN2</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>10100</td>
<td>UQXTN, UQXTN2</td>
</tr>
<tr>
<td>@x</td>
<td>10110</td>
<td>FCVTXN, FCVTXN2</td>
<td></td>
</tr>
<tr>
<td>@x</td>
<td>11010</td>
<td>FCVTNU (vector)</td>
<td></td>
</tr>
<tr>
<td>@x</td>
<td>11011</td>
<td>FCVTMU (vector)</td>
<td></td>
</tr>
<tr>
<td>@x</td>
<td>11100</td>
<td>FCVTAU (vector)</td>
<td></td>
</tr>
<tr>
<td>@x</td>
<td>11101</td>
<td>UCVTF (vector, integer)</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>01100</td>
<td>FCMGE (zero)</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>01101</td>
<td>FCMLE (zero)</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>01110</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>11010</td>
<td>FCVTPU (vector)</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>11011</td>
<td>FCVTZU (vector, integer)</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>11101</td>
<td>FRSQRTE</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>11111</td>
<td>Unallocated.</td>
<td></td>
</tr>
</tbody>
</table>
Advanced SIMD scalar three different

This section describes the encoding of the Advanced SIMD scalar three different instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>01100</td>
<td>FMAXNMP (scalar) - Encoding</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01101</td>
<td>FADDP (scalar) - Encoding</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01111</td>
<td>FMAXP (scalar) - Encoding</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01100</td>
<td>FMINNMP (scalar) - Encoding</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01111</td>
<td>FMINP (scalar) - Encoding</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01100</td>
<td>FMAXNMP (scalar) - Encoding</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01101</td>
<td>FADDP (scalar) - Encoding</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01111</td>
<td>FMAXP (scalar) - Encoding</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01100</td>
<td>FMINNMP (scalar) - Encoding</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01111</td>
<td>FMINP (scalar) - Encoding</td>
<td>-</td>
</tr>
</tbody>
</table>
Advanced SIMD scalar three same

This section describes the encoding of the Advanced SIMD scalar three same instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

### Instruction page

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31 30 29 28 27 26 24 23 22 21 20]</td>
<td>U(opcode)</td>
</tr>
<tr>
<td>0 1 U</td>
<td>1 1 1 1 0</td>
</tr>
<tr>
<td>U size</td>
<td>opcode</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>0 -</td>
<td>10101</td>
</tr>
<tr>
<td>0 -</td>
<td>10110</td>
</tr>
<tr>
<td>0 -</td>
<td>10111</td>
</tr>
<tr>
<td>0 0x</td>
<td>11000</td>
</tr>
<tr>
<td>0 0x</td>
<td>11001</td>
</tr>
<tr>
<td>0 0x</td>
<td>11010</td>
</tr>
<tr>
<td>0 0x</td>
<td>11011</td>
</tr>
<tr>
<td>0 0x</td>
<td>11100</td>
</tr>
<tr>
<td>0 0x</td>
<td>11101</td>
</tr>
<tr>
<td>0 0x</td>
<td>11110</td>
</tr>
<tr>
<td>0 0x</td>
<td>11111</td>
</tr>
<tr>
<td>0 1x</td>
<td>11000</td>
</tr>
<tr>
<td>0 1x</td>
<td>11001</td>
</tr>
<tr>
<td>0 1x</td>
<td>11010</td>
</tr>
<tr>
<td>0 1x</td>
<td>11100</td>
</tr>
<tr>
<td>0 1x</td>
<td>11101</td>
</tr>
<tr>
<td>0 1x</td>
<td>11110</td>
</tr>
<tr>
<td>0 1x</td>
<td>11111</td>
</tr>
<tr>
<td>1 -</td>
<td>00001</td>
</tr>
<tr>
<td>1 -</td>
<td>00101</td>
</tr>
<tr>
<td>1 -</td>
<td>00110</td>
</tr>
<tr>
<td>1 -</td>
<td>00111</td>
</tr>
<tr>
<td>1 -</td>
<td>01000</td>
</tr>
<tr>
<td>1 -</td>
<td>01001</td>
</tr>
<tr>
<td>1 -</td>
<td>01010</td>
</tr>
<tr>
<td>1 -</td>
<td>01011</td>
</tr>
<tr>
<td>1 -</td>
<td>10000</td>
</tr>
<tr>
<td>1 -</td>
<td>10001</td>
</tr>
<tr>
<td>1 -</td>
<td>10100</td>
</tr>
<tr>
<td>1 -</td>
<td>10101</td>
</tr>
<tr>
<td>1 -</td>
<td>10110</td>
</tr>
<tr>
<td>1 -</td>
<td>10111</td>
</tr>
</tbody>
</table>
Advanced SIMD scalar shift by immediate

This section describes the encoding of the Advanced SIMD scalar shift by immediate instruction class. The encodings in this section are decoded from the Data Processing -- Scalar Floating-Point and Advanced SIMD on page C4-601.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22</th>
<th>19 18</th>
<th>16</th>
<th>15</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>U</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>immh</td>
</tr>
</tbody>
</table>
```

**Decode fields**

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x</td>
<td>11000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11010</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11100</td>
<td>FCMGE (register)</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11101</td>
<td>FACGE</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11110</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11111</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11010</td>
<td>FABD</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11100</td>
<td>FCMGT (register)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11101</td>
<td>FACGT</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11110</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11111</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

**Advanced SIMD scalar shift by immediate**

This section describes the encoding of the Advanced SIMD scalar shift by immediate instruction class. The encodings in this section are decoded from Data Processing -- Scalar Floating-Point and Advanced SIMD on page C4-601.
### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>immh</th>
<th>opcode</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>!= 0000 00000</td>
<td>SSHR</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 00010</td>
<td>SSRA</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 00100</td>
<td>SRSHR</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 00110</td>
<td>SRSRA</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 01000</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 01010</td>
<td>SHL</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 01100</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 01110</td>
<td>SQSHL (immediate)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 10000</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 10001</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 10010</td>
<td>SQSHRN, SQSHRN2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 10011</td>
<td>SQRSRN, SQRSRN2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 11100</td>
<td>SCVT (vector, fixed-point)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 11111</td>
<td>FCVTZS (vector, fixed-point)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 00000</td>
<td>USHR</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 00010</td>
<td>USRA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 00100</td>
<td>URSHR</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 00110</td>
<td>URSRA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 01000</td>
<td>SRI</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 01010</td>
<td>SLI</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 01100</td>
<td>SQSHLU</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 01110</td>
<td>UQSHL (immediate)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 10000</td>
<td>SQSHRUN, SQSHRUN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 10001</td>
<td>SQRSRUN, SQRSRUN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 10010</td>
<td>UQSHRN, UQSHRN2</td>
<td></td>
</tr>
</tbody>
</table>
This section describes the encoding of the Advanced SIMD scalar x indexed element instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD on page C4-601.*

![Image of instruction encoding]

### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>immh</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>10011</td>
<td>UQRSHRN, UQRSHRN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>11100</td>
<td>UCVTF (vector, fixed-point)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>11111</td>
<td>FCVTZU (vector, fixed-point)</td>
<td></td>
</tr>
</tbody>
</table>

### Advanced SIMD scalar x indexed element

[The text continues with the detailed encoding information as shown in the image.]
Advanced SIMD table lookup

This section describes the encoding of the Advanced SIMD table lookup instruction class. The encodings in this section are decoded from Data Processing -- Scalar Floating-Point and Advanced SIMD on page C4-601.

[31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 | Q | 0 0 1 1 1 0 | op2 | 0 | Rm | 0 | len | op | 0 0 | Rd |

Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>lx</td>
<td>0001</td>
<td>FMLA (by element) - Encoding</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>lx</td>
<td>0101</td>
<td>FMLS (by element) - Encoding</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>lx</td>
<td>1001</td>
<td>FMUL (by element) - Encoding</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1100</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1101</td>
<td>SQRDMLAH (by element)</td>
<td>FEAT_RDM</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1111</td>
<td>SQRDMLSH (by element)</td>
<td>FEAT_RDM</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0001</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1001</td>
<td>FMULX (by element) - Encoding</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>lx</td>
<td>0001</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>lx</td>
<td>0101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>lx</td>
<td>1001</td>
<td>FMULX (by element) - Encoding</td>
<td>-</td>
</tr>
</tbody>
</table>

Decode fields

<table>
<thead>
<tr>
<th>op2</th>
<th>len</th>
<th>op</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>0</td>
<td>TBL - Single register table variant</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>TBX - Single register table variant</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>0</td>
<td>TBL - Two register table variant</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>1</td>
<td>TBX - Two register table variant</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>0</td>
<td>TBL - Three register table variant</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>1</td>
<td>TBX - Three register table variant</td>
</tr>
</tbody>
</table>
Advanced SIMD permute

This section describes the encoding of the Advanced SIMD permute instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 1 1 0</td>
<td>size 0</td>
<td>Rm 0</td>
<td>opcode 1 0</td>
<td>Rn 0</td>
<td>Rd</td>
<td></td>
</tr>
</tbody>
</table>
```

### Decode fields

<table>
<thead>
<tr>
<th>op2</th>
<th>len</th>
<th>op</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>11</td>
<td>0</td>
<td>TBL - Four register table variant</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>1</td>
<td>TBX - Four register table variant</td>
</tr>
<tr>
<td>1x</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

Advanced SIMD extract

This section describes the encoding of the Advanced SIMD extract instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 1 0 1 1 0</td>
<td>op2 0</td>
<td>Rm 0</td>
<td>imm4 0</td>
<td>Rn 0</td>
<td>Rd</td>
<td></td>
</tr>
</tbody>
</table>
```

### Decode fields

<table>
<thead>
<tr>
<th>opcode</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>001</td>
<td>UZP1</td>
</tr>
<tr>
<td>010</td>
<td>TRN1</td>
</tr>
<tr>
<td>011</td>
<td>ZIP1</td>
</tr>
<tr>
<td>100</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>101</td>
<td>UZP2</td>
</tr>
<tr>
<td>110</td>
<td>TRN2</td>
</tr>
<tr>
<td>111</td>
<td>ZIP2</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>op2</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00</td>
<td>EXT</td>
</tr>
<tr>
<td>1x</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
Advanced SIMD copy

This section describes the encoding of the Advanced SIMD copy instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>op</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Decode fields**

<table>
<thead>
<tr>
<th>Q</th>
<th>op</th>
<th>imm5</th>
<th>imm4</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0000</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0000</td>
<td>DUP (element)</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0001</td>
<td>DUP (general)</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0010</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0100</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0110</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>1xxx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0101</td>
<td>SMOV</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0111</td>
<td>UMOV</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>0011</td>
<td>INS (general)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>0101</td>
<td>SMOV</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>x1000</td>
<td>0111</td>
<td>UMOV</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>INS (element)</td>
</tr>
</tbody>
</table>

Advanced SIMD three same (FP16)

This section describes the encoding of the Advanced SIMD three same (FP16) instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.
### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>a</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td>FMAXNM (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001</td>
<td>FMLA (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>FADD (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011</td>
<td>FMULX</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>FCMEQ (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>110</td>
<td>FMAX (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>111</td>
<td>FRECPS</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>000</td>
<td>FMINNM (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>001</td>
<td>FMLS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>FSUB (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>110</td>
<td>FMIN (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>111</td>
<td>FRSQRTS</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000</td>
<td>FMAXNMP (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>001</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>FADDP (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>FMUL (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>FCMGE (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>FACGE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td>FMAXP (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
<td>FDIV (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>000</td>
<td>FMINNMP (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>001</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>FABD</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
### Advanced SIMD two-register miscellaneous (FP16)

This section describes the encoding of the Advanced SIMD two-register miscellaneous (FP16) instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>U</strong></td>
<td><strong>a</strong></td>
<td><strong>opcode</strong></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>110</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>U</strong></td>
<td><strong>a</strong></td>
<td><strong>opcode</strong></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1100</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1101</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1110</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11100</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11101</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01100</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01101</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01110</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01111</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11000</td>
</tr>
<tr>
<td>Decode fields</td>
<td>Instruction page</td>
<td>Feature</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------</td>
<td>---------</td>
</tr>
<tr>
<td>Ua opcode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 11001</td>
<td>FRINTZ (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11010</td>
<td>FCVTPS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11011</td>
<td>FCVTZS (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11101</td>
<td>FRECPE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11111</td>
<td>Unallocated. -</td>
<td></td>
</tr>
<tr>
<td>1 0 11000</td>
<td>FRINTA (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 11001</td>
<td>FRINTX (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 11010</td>
<td>FCVTNU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 11011</td>
<td>FCVTMU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 11100</td>
<td>FCVTAU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 11101</td>
<td>UCVTF (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 01100</td>
<td>FCMGE (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 01101</td>
<td>FCMLE (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 01110</td>
<td>Unallocated. -</td>
<td></td>
</tr>
<tr>
<td>1 1 01111</td>
<td>FNEG (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 11000</td>
<td>Unallocated. -</td>
<td></td>
</tr>
<tr>
<td>1 1 11001</td>
<td>FRINTI (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 11010</td>
<td>FCVTPU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 11011</td>
<td>FCVTZU (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 11101</td>
<td>FRSQRTE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 11111</td>
<td>FSQRT (vector)</td>
<td>FEAT_FP16</td>
</tr>
</tbody>
</table>

**Advanced SIMD three-register extension**

This section describes the encoding of the Advanced SIMD three-register extension instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.
### Decode fields

<table>
<thead>
<tr>
<th>Q</th>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0x</td>
<td>0011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>0011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>Unallocated.</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>Unallocated.</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0010</td>
<td>SDOT (vector)</td>
<td>FEAT_DotProd</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1xxx</td>
<td>Unallocated.</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0011</td>
<td>USDOT (vector)</td>
<td>FEAT_I8MM</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0000</td>
<td>SQRDMLAH (vector)</td>
<td>FEAT_RDM</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0001</td>
<td>SQRDMLSH (vector)</td>
<td>FEAT_RDM</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0010</td>
<td>UDOT (vector)</td>
<td>FEAT_DotProd</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10xx</td>
<td>FCMLA</td>
<td>FEAT_FCMA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11x0</td>
<td>FCADD</td>
<td>FEAT_FCMA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00</td>
<td>1101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01</td>
<td>1111</td>
<td>BFDOT (vector)</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1x</td>
<td>1101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>1111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
<td>1111</td>
<td>BFMLALB, BFMLALT (vector)</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01xx</td>
<td>Unallocated.</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01XX</td>
<td>1101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0x</td>
<td>01xx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1x</td>
<td>011x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0100</td>
<td>SMMLA (vector)</td>
<td>FEAT_I8MM</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0101</td>
<td>USMMLA (vector)</td>
<td>FEAT_I8MM</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01</td>
<td>1101</td>
<td>BFMLLA</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0100</td>
<td>UMMLA (vector)</td>
<td>FEAT_I8MM</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
Advanced SIMD two-register miscellaneous

This section describes the encoding of the Advanced SIMD two-register miscellaneous instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD on page C4-601.*

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q U 0 1 1</td>
<td>1 0 0 0 0</td>
<td>opcode</td>
<td>1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>1000x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>10101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0x</td>
<td>011xx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>1x</td>
<td>10111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>1x</td>
<td>11110</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>11</td>
<td>10110</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>00000</td>
<td>REV64</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>00001</td>
<td>REV16 (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>00010</td>
<td>SADDLP</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>00011</td>
<td>SUQADD</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>00100</td>
<td>CLS (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>00101</td>
<td>CNT</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>00110</td>
<td>SADALP</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>00111</td>
<td>SQABS</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>01000</td>
<td>CMGT (zero)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>01001</td>
<td>CMEQ (zero)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>01010</td>
<td>CMLT (zero)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>01011</td>
<td>ABS</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>10010</td>
<td>XTN, XTN2</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>10011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>10100</td>
<td>SQXTN, SQXTN2</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>10110</td>
<td>FCVTN, FCVTN2</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>10111</td>
<td>FCVTL, FCVTL2</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11000</td>
<td>FRINTN (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11001</td>
<td>FRINTM (vector)</td>
<td>-</td>
</tr>
<tr>
<td>Decode fields</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>size</td>
<td>opcode</td>
<td>Instruction page</td>
<td>Feature</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11010</td>
<td>FCVTNS (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11011</td>
<td>FCVTMS (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11100</td>
<td>FCVTAS (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11101</td>
<td>SCVT (vector, integer)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11110</td>
<td>FRINT32Z (vector)</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11111</td>
<td>FRINT64Z (vector)</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01100</td>
<td>FCMGT (zero)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01101</td>
<td>FCMEQ (zero)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01110</td>
<td>FCMLT (zero)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01111</td>
<td>FABS (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11000</td>
<td>FRINTP (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11001</td>
<td>FRINTZ (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11010</td>
<td>FCVTPS (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11011</td>
<td>FCVTZS (vector, integer)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11100</td>
<td>URECPE</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11101</td>
<td>FRECPE</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10110</td>
<td>BFCVTN, BFCVTN2</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>00000</td>
<td>REV32 (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>00001</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>00010</td>
<td>UADDLP</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>00011</td>
<td>USQADD</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>00100</td>
<td>CLZ (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>00110</td>
<td>UADALP</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>00111</td>
<td>SQNEG</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>01000</td>
<td>CMGE (zero)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>01001</td>
<td>CMLE (zero)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>01010</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>01011</td>
<td>NEG (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>10010</td>
<td>SQXTUN, SQXTUN2</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>10011</td>
<td>SHLL, SHLL2</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>10100</td>
<td>UQXTN, UQXTN2</td>
<td>-</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>U size</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0x</td>
<td>10110</td>
<td>FCVTXN, FCVTXN2</td>
<td>-</td>
</tr>
<tr>
<td>1 0x</td>
<td>10111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1 0x</td>
<td>11000</td>
<td>FRINTA (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x</td>
<td>11001</td>
<td>FRINTX (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x</td>
<td>11010</td>
<td>FCVTNU (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x</td>
<td>11011</td>
<td>FCVTMU (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x</td>
<td>11100</td>
<td>FCVTAU (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x</td>
<td>11101</td>
<td>UCVTF (vector, integer)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x</td>
<td>11110</td>
<td>FRINT32X (vector)</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>1 0x</td>
<td>11111</td>
<td>FRINT64X (vector)</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>1 00</td>
<td>00101</td>
<td>NOT</td>
<td>-</td>
</tr>
<tr>
<td>1 01</td>
<td>00101</td>
<td>RBIT (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 1x</td>
<td>00101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1 1x</td>
<td>01100</td>
<td>FCMGE (zero)</td>
<td>-</td>
</tr>
<tr>
<td>1 1x</td>
<td>01101</td>
<td>FCMLE (zero)</td>
<td>-</td>
</tr>
<tr>
<td>1 1x</td>
<td>01110</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1 1x</td>
<td>01111</td>
<td>FNEG (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 1x</td>
<td>11000</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1 1x</td>
<td>11001</td>
<td>FRINTI (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 1x</td>
<td>11010</td>
<td>FCVTPU (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 1x</td>
<td>11011</td>
<td>FCVTZU (vector, integer)</td>
<td>-</td>
</tr>
<tr>
<td>1 1x</td>
<td>11100</td>
<td>URSQRTE</td>
<td>-</td>
</tr>
<tr>
<td>1 1x</td>
<td>11101</td>
<td>FRSQRTE</td>
<td>-</td>
</tr>
<tr>
<td>1 1x</td>
<td>11111</td>
<td>FSQRT (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 10</td>
<td>10110</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

### Advanced SIMD across lanes

This section describes the encoding of the Advanced SIMD across lanes instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.
### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0000x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00010</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>001xx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01110</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10xxx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1100x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>111xx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>00011</td>
<td>SADDLV</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>01010</td>
<td>SMAXV</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>11010</td>
<td>SMINV</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>11011</td>
<td>ADDV</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>01100</td>
<td>FMAXNMV - Encoding</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>01111</td>
<td>FMAXV - Encoding</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01100</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01100</td>
<td>FMINNMV - Encoding</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01111</td>
<td>FMINV - Encoding</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>01100</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>01111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>00011</td>
<td>UADDLV</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>01010</td>
<td>UMAXV</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>11010</td>
<td>UMINV</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>11011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>01100</td>
<td>FMAXNMV - Encoding</td>
<td>-</td>
</tr>
</tbody>
</table>
Advanced SIMD three different

This section describes the encoding of the Advanced SIMD three different instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x</td>
<td>01111</td>
<td>FMAXV - Encoding</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01100</td>
<td>FMINNMV - Encoding</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01111</td>
<td>FMINV - Encoding</td>
<td>-</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>[31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>U</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>opcode</td>
<td>0</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>U</th>
<th>opcode</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>1111</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>SADDL, SADDL2</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>SADDW, SADDW2</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>SSUBL, SSUBL2</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>SSUBW, SSUBW2</td>
</tr>
<tr>
<td>0</td>
<td>0100</td>
<td>ADDHN, ADDHN2</td>
</tr>
<tr>
<td>0</td>
<td>0101</td>
<td>Sabal, Sabal2</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>SUBHN, SUBHN2</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>SABDL, SABDL2</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>SMLAL, SMLAL2 (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1001</td>
<td>SQDMLAL, SQDMLAL2 (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td>SMLSL, SMLSL2 (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1011</td>
<td>SQDMLSL, SQDMLSL2 (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1100</td>
<td>SMULL, SMULL2 (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1101</td>
<td>SQDMULL, SQDMULL2 (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>PMULL, PMULL2</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>UADDL, UADDL2</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>UADDW, UADDW2</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>USUBL, USUBL2</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>USUBW, USUBW2</td>
</tr>
</tbody>
</table>
Advanced SIMD three same

This section describes the encoding of the Advanced SIMD three same instruction class. The encodings in this section are decoded from Data Processing -- Scalar Floating-Point and Advanced SIMD on page C4-601.
### Decode fields

<table>
<thead>
<tr>
<th>U size</th>
<th>Opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 -</td>
<td>01101</td>
<td>SMIN</td>
<td>-</td>
</tr>
<tr>
<td>0 -</td>
<td>01110</td>
<td>SABD</td>
<td>-</td>
</tr>
<tr>
<td>0 -</td>
<td>01111</td>
<td>SABA</td>
<td>-</td>
</tr>
<tr>
<td>0 -</td>
<td>10000</td>
<td>ADD (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 -</td>
<td>10001</td>
<td>CMTST</td>
<td>-</td>
</tr>
<tr>
<td>0 -</td>
<td>10010</td>
<td>MLA (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 -</td>
<td>10011</td>
<td>MUL (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 -</td>
<td>10100</td>
<td>SMAXP</td>
<td>-</td>
</tr>
<tr>
<td>0 -</td>
<td>10101</td>
<td>SMINP</td>
<td>-</td>
</tr>
<tr>
<td>0 -</td>
<td>10110</td>
<td>SQDMULH (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 -</td>
<td>10111</td>
<td>ADDP (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 0x</td>
<td>11000</td>
<td>FMAXNM (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 0x</td>
<td>11001</td>
<td>FMLA (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 0x</td>
<td>11010</td>
<td>FADD (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 0x</td>
<td>11011</td>
<td>FMULX</td>
<td>-</td>
</tr>
<tr>
<td>0 0x</td>
<td>11100</td>
<td>FCMEQ (register)</td>
<td>-</td>
</tr>
<tr>
<td>0 0x</td>
<td>11110</td>
<td>FMAX (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 0x</td>
<td>11111</td>
<td>FRECPS</td>
<td>-</td>
</tr>
<tr>
<td>0 00</td>
<td>00011</td>
<td>AND (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 00</td>
<td>11101</td>
<td>FMLAL, FMLAL2 (vector) - Encoding</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>0 01</td>
<td>00011</td>
<td>BIC (vector, register)</td>
<td>-</td>
</tr>
<tr>
<td>0 01</td>
<td>11101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0 1x</td>
<td>11000</td>
<td>FMINNM (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 1x</td>
<td>11001</td>
<td>FMLS (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 1x</td>
<td>11010</td>
<td>FSUB (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 1x</td>
<td>11011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0 1x</td>
<td>11100</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0 1x</td>
<td>11110</td>
<td>FMIN (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 1x</td>
<td>11111</td>
<td>FRSQRTS</td>
<td>-</td>
</tr>
<tr>
<td>0 10</td>
<td>00011</td>
<td>ORR (vector, register)</td>
<td>-</td>
</tr>
<tr>
<td>0 10</td>
<td>11101</td>
<td>FMLSL, FMLSL2 (vector) - Encoding</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>0 11</td>
<td>00011</td>
<td>ORN (vector)</td>
<td>-</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>U size</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 11</td>
<td>11101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>00000</td>
<td>UHADD</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>00001</td>
<td>UQADD</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>00010</td>
<td>URHADD</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>00100</td>
<td>UHSUB</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>00101</td>
<td>UQSUB</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>00110</td>
<td>CMHI (register)</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>00111</td>
<td>CMHS (register)</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>01000</td>
<td>USHL</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>01001</td>
<td>UQSHL (register)</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>01010</td>
<td>URSHL</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>01011</td>
<td>UQRSHL</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>01100</td>
<td>UMAX</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>01101</td>
<td>UMIN</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>01110</td>
<td>UABD</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>01111</td>
<td>UABA</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>10000</td>
<td>SUB (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>10001</td>
<td>CMEQ (register)</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>10010</td>
<td>MLS (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>10011</td>
<td>PMUL</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>10100</td>
<td>UMAXP</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>10101</td>
<td>UMINP</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>10110</td>
<td>SQRDMULH (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 -</td>
<td>10111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1 0x</td>
<td>11000</td>
<td>FMAXNMP (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x</td>
<td>11010</td>
<td>FADDP (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x</td>
<td>11011</td>
<td>FMUL (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x</td>
<td>11100</td>
<td>FCMGE (register)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x</td>
<td>11101</td>
<td>FACGE</td>
<td>-</td>
</tr>
<tr>
<td>1 0x</td>
<td>11110</td>
<td>FMAXP (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x</td>
<td>11111</td>
<td>FDIV (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 00</td>
<td>00011</td>
<td>EOR (vector)</td>
<td>-</td>
</tr>
</tbody>
</table>
Advanced SIMD modified immediate

This section describes the encoding of the Advanced SIMD modified immediate instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

---

### Advanced SIMD modified immediate

This section describes the encoding of the Advanced SIMD modified immediate instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 12|11 10 9 8 |7 6 5 4 |0 ]
0|Q|op|0|1|1|1|0|0|0|0|0|a|b|c| cmode |o2|1|d|e|f|g|h|Rd
```

### Decode fields

<table>
<thead>
<tr>
<th>Q</th>
<th>op</th>
<th>cmode</th>
<th>o2</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0</td>
<td>0xxx</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0x0</td>
<td>0</td>
<td>MOV1 - 32-bit shifted immediate variant</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0x1</td>
<td>0</td>
<td>ORR (vector, immediate) - 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>1xx</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>1x0</td>
<td>0</td>
<td>MOV1 - 16-bit shifted immediate variant</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>1x1</td>
<td>0</td>
<td>ORR (vector, immediate) - 16-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>10x</td>
<td>0</td>
<td>MOV1 - 32-bit shifting ones variant</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>11x</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>110</td>
<td>0</td>
<td>MOV1 - 8-bit variant</td>
<td>-</td>
</tr>
</tbody>
</table>
Advanced SIMD shift by immediate

This section describes the encoding of the Advanced SIMD shift by immediate instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22</th>
<th>19 18</th>
<th>16</th>
<th>15</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>!=0000</td>
<td>immh</td>
<td>opcode</td>
</tr>
</tbody>
</table>
```

Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>opcode</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>00001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>00011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>00101</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>00111</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>01001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>01011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>01101</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>01111</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>10011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>10101</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>1011x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>Decode fields</td>
<td>Instruction page</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>------------------</td>
<td></td>
</tr>
<tr>
<td>U opcode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 00000</td>
<td>SSHR</td>
<td></td>
</tr>
<tr>
<td>0 00010</td>
<td>SSRA</td>
<td></td>
</tr>
<tr>
<td>0 00100</td>
<td>SRSR</td>
<td></td>
</tr>
<tr>
<td>0 00110</td>
<td>SRSRA</td>
<td></td>
</tr>
<tr>
<td>0 01000</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0 01010</td>
<td>SHL</td>
<td></td>
</tr>
<tr>
<td>0 01100</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0 01110</td>
<td>SQSHL (immediate)</td>
<td></td>
</tr>
<tr>
<td>0 10000</td>
<td>SHRN, SHRN2</td>
<td></td>
</tr>
<tr>
<td>0 10001</td>
<td>RSHRN, RSHRN2</td>
<td></td>
</tr>
<tr>
<td>0 10010</td>
<td>SQSHRN, SQSHRN2</td>
<td></td>
</tr>
<tr>
<td>0 10011</td>
<td>SQRSHRN, SQRSHRN2</td>
<td></td>
</tr>
<tr>
<td>0 10100</td>
<td>SSHLL, SSHLL2</td>
<td></td>
</tr>
<tr>
<td>0 11100</td>
<td>SCVTFS (vector, fixed-point)</td>
<td></td>
</tr>
<tr>
<td>0 11111</td>
<td>FCVTZS (vector, fixed-point)</td>
<td></td>
</tr>
<tr>
<td>1 00000</td>
<td>USHR</td>
<td></td>
</tr>
<tr>
<td>1 00010</td>
<td>USRA</td>
<td></td>
</tr>
<tr>
<td>1 00100</td>
<td>URSHR</td>
<td></td>
</tr>
<tr>
<td>1 00110</td>
<td>URSRA</td>
<td></td>
</tr>
<tr>
<td>1 01000</td>
<td>SRI</td>
<td></td>
</tr>
<tr>
<td>1 01010</td>
<td>SLI</td>
<td></td>
</tr>
<tr>
<td>1 01100</td>
<td>SQSHLU</td>
<td></td>
</tr>
<tr>
<td>1 01110</td>
<td>UQSHL (immediate)</td>
<td></td>
</tr>
<tr>
<td>1 10000</td>
<td>SQSHRUN, SQSHRUN2</td>
<td></td>
</tr>
<tr>
<td>1 10001</td>
<td>SQRSHRUN, SQRSHRUN2</td>
<td></td>
</tr>
<tr>
<td>1 10010</td>
<td>UQSHRN, UQSHRN2</td>
<td></td>
</tr>
<tr>
<td>1 10011</td>
<td>UQRSHRN, UQRSHRN2</td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD vector x indexed element

This section describes the encoding of the Advanced SIMD vector x indexed element instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>U</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>size</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>opcode</td>
<td>H</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

#### Decode fields

<table>
<thead>
<tr>
<th>U opcode</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100</td>
<td>USHLL, USHLL2</td>
</tr>
<tr>
<td>11100</td>
<td>UCVTF (vector, fixed-point)</td>
</tr>
<tr>
<td>11111</td>
<td>FCV TZU (vector, fixed-point)</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>U size opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>- 01 1001</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0 00 0010</td>
<td>SMLAL, SMLAL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0 00 0011</td>
<td>SQDMLAL, SQDMLAL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0 00 1010</td>
<td>SMLS, SMLSL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0 00 1011</td>
<td>SQDMLS, SQDMLSL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0 00 1100</td>
<td>MUL (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0 00 1101</td>
<td>SMUL, SMULL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0 00 1110</td>
<td>SQDMULL, SQDMULL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0 00 1111</td>
<td>SQDMULH (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0 00 1110</td>
<td>SDOT (by element)</td>
<td>FEAT_DotProd</td>
</tr>
<tr>
<td>0 00 0000</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0 00 0010</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0 00 0001</td>
<td>FMLA (by element)</td>
<td>Encoding FEAT_FP16</td>
</tr>
<tr>
<td>0 00 0101</td>
<td>FMLS (by element)</td>
<td>Encoding FEAT_FP16</td>
</tr>
<tr>
<td>0 00 1001</td>
<td>FMUL (by element)</td>
<td>Encoding FEAT_FP16</td>
</tr>
<tr>
<td>0 00 1111</td>
<td>SUDOT (by element)</td>
<td>Encoding FEAT_18MM</td>
</tr>
<tr>
<td>0 01 0000</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0 01 0101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>1111</td>
<td>BFDOT (by element)</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>0001</td>
<td>FMLA (by element) - Encoding</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>0101</td>
<td>FMLS (by element) - Encoding</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>1001</td>
<td>FMUL (by element) - Encoding</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0000</td>
<td>FMLAL, FMLAL2 (by element) - Encoding</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0100</td>
<td>FMLS, FMLS2 (by element) - Encoding</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1111</td>
<td>USDOT (by element)</td>
<td>FEAT_I8MM</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>0000</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>0100</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>1111</td>
<td>BFMLALB, BFMLALT (by element)</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0000</td>
<td>MLA (by element)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0010</td>
<td>UMLAL, UMLAL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0100</td>
<td>MLS (by element)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0110</td>
<td>UMLSL, UMLSL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1010</td>
<td>UMULL, UMULL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1101</td>
<td>SQRDMLAH (by element)</td>
<td>FEAT_RDM</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1110</td>
<td>UDOT (by element)</td>
<td>FEAT.DotProd</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1111</td>
<td>SQRDMLSH (by element)</td>
<td>FEAT_RDM</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>1000</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>1100</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0001</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1001</td>
<td>FMULX (by element) - Encoding</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>0xx1</td>
<td>FCMLA (by element)</td>
<td>FEAT_FCMA</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>1001</td>
<td>FMULX (by element) - Encoding</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0xx1</td>
<td>FCMLA (by element)</td>
<td>FEAT_FCMA</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1000</td>
<td>FMLAL, FMLAL2 (by element) - Encoding</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1100</td>
<td>FMLS, FMLS2 (by element) - Encoding</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0001</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
### Cryptographic three-register, imm2

This section describes the encoding of the Cryptographic three-register, imm2 instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

<table>
<thead>
<tr>
<th>opcode</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td>Unallocated. -</td>
</tr>
<tr>
<td>0101</td>
<td>Unallocated. -</td>
</tr>
<tr>
<td>0111</td>
<td>Unallocated. -</td>
</tr>
<tr>
<td>1000</td>
<td>Unallocated. -</td>
</tr>
<tr>
<td>1100</td>
<td>Unallocated. -</td>
</tr>
</tbody>
</table>

### Cryptographic three-register SHA 512

This section describes the encoding of the Cryptographic three-register SHA 512 instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

<table>
<thead>
<tr>
<th>opcode</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SM3TT1A FEAT_SM3</td>
</tr>
<tr>
<td>01</td>
<td>SM3TT1B FEAT_SM3</td>
</tr>
<tr>
<td>10</td>
<td>SM3TT2A FEAT_SM3</td>
</tr>
<tr>
<td>11</td>
<td>SM3TT2B FEAT_SM3</td>
</tr>
</tbody>
</table>
Cryptographic four-register

This section describes the encoding of the Cryptographic four-register instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>RAX1</td>
<td>FEAT_SHA3</td>
</tr>
<tr>
<td>0100</td>
<td>SM3PARTW1</td>
<td>FEAT_SM3</td>
</tr>
<tr>
<td>0101</td>
<td>SM3PARTW2</td>
<td>FEAT_SM3</td>
</tr>
<tr>
<td>0110</td>
<td>SM4EKEY</td>
<td>FEAT_SM4</td>
</tr>
<tr>
<td>0111</td>
<td>Unallocated</td>
<td>-</td>
</tr>
</tbody>
</table>

Cryptographic two-register SHA 512

This section describes the encoding of the Cryptographic two-register SHA 512 instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SHA512SU0</td>
<td>FEAT_SHA512</td>
</tr>
<tr>
<td>01</td>
<td>SM4E</td>
<td>FEAT_SM4</td>
</tr>
<tr>
<td>1x</td>
<td>Unallocated</td>
<td>-</td>
</tr>
</tbody>
</table>
Conversion between floating-point and fixed-point

This section describes the encoding of the Conversion between floating-point and fixed-point instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24 23 22 21 20</th>
<th>19 18 16 15</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>S</td>
<td>ptype</td>
<td>rmode</td>
<td>opcode</td>
<td>scale</td>
</tr>
</tbody>
</table>

**Decode fields**

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>ptype</th>
<th>rmode</th>
<th>opcode</th>
<th>scale</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1xx</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x0</td>
<td>00x</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x1</td>
<td>01x</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0x</td>
<td>00x</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td></td>
<td>1x</td>
<td>01x</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0xxxx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>010</td>
<td>-</td>
<td>SCVTFT (scalar, fixed-point) - 32-bit to single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>011</td>
<td>-</td>
<td>UCVTFT (scalar, fixed-point) - 32-bit to single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>000</td>
<td>-</td>
<td>FCVTZS (scalar, fixed-point) - Single-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>001</td>
<td>-</td>
<td>FCVTZU (scalar, fixed-point) - Single-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>010</td>
<td>-</td>
<td>SCVTFT (scalar, fixed-point) - 32-bit to double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>011</td>
<td>-</td>
<td>UCVTFT (scalar, fixed-point) - 32-bit to double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>000</td>
<td>-</td>
<td>FCVTZS (scalar, fixed-point) - Double-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>001</td>
<td>-</td>
<td>FCVTZU (scalar, fixed-point) - Double-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>010</td>
<td>-</td>
<td>SCVTFT (scalar, fixed-point) - 32-bit to half-precision variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>011</td>
<td>-</td>
<td>UCVTFT (scalar, fixed-point) - 32-bit to half-precision variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>000</td>
<td>-</td>
<td>FCVTZS (scalar, fixed-point) - Half-precision to 32-bit variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>001</td>
<td>-</td>
<td>FCVTZU (scalar, fixed-point) - Half-precision to 32-bit variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>010</td>
<td>-</td>
<td>SCVTFT (scalar, fixed-point) - 64-bit to single-precision variant</td>
<td>-</td>
</tr>
</tbody>
</table>
### Conversion between floating-point and integer

This section describes the encoding of the Conversion between floating-point and integer instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

```
 31 30 29 28 27 26 25 24 23 22 21 20 19 18
 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 sf 0 | S | rmode | ptype | opcode | scale |
```

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>ptype</th>
<th>rmode</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>011</td>
<td>UCVTF (scalar, fixed-point) - 64-bit to single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>000</td>
<td>FCVTZS (scalar, fixed-point) - Single-precision to 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>001</td>
<td>FCVTZU (scalar, fixed-point) - Single-precision to 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>010</td>
<td>SCVTF (scalar, fixed-point) - 64-bit to double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>011</td>
<td>UCVTF (scalar, fixed-point) - 64-bit to double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>000</td>
<td>FCVTZS (scalar, fixed-point) - Double-precision to 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>001</td>
<td>FCVTZU (scalar, fixed-point) - Double-precision to 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>010</td>
<td>SCVTF (scalar, fixed-point) - 64-bit to half-precision variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>011</td>
<td>UCVTF (scalar, fixed-point) - 64-bit to half-precision variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>000</td>
<td>FCVTZS (scalar, fixed-point) - Half-precision to 64-bit variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>001</td>
<td>FCVTZU (scalar, fixed-point) - Half-precision to 64-bit variant</td>
<td>FEAT_FP16</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>ptype</th>
<th>rmode</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00 x1</td>
<td>11x</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00</td>
<td>000</td>
<td>FCVTNS (scalar) - Single-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00</td>
<td>001</td>
<td>FCVTNU (scalar) - Single-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00</td>
<td>010</td>
<td>SCVTF (scalar, integer) - 32-bit to single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00</td>
<td>011</td>
<td>UCVTF (scalar, integer) - 32-bit to single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00</td>
<td>100</td>
<td>FCVTAS (scalar) - Single-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00</td>
<td>101</td>
<td>FCVTAU (scalar) - Single-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>10</td>
<td>000</td>
<td>FMOV (general) - Single-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>10</td>
<td>001</td>
<td>FMOV (general) - 32-bit to single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 01</td>
<td>01</td>
<td>000</td>
<td>FCVTMS (scalar) - Single-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 01</td>
<td>01</td>
<td>001</td>
<td>FCVTMU (scalar) - Single-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 11</td>
<td>00</td>
<td>000</td>
<td>FCVTZS (scalar, integer) - Single-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 11</td>
<td>00</td>
<td>001</td>
<td>FCVTZU (scalar, integer) - Single-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 0x</td>
<td>11x</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00</td>
<td>00</td>
<td>000</td>
<td>FCVTNS (scalar) - Double-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00</td>
<td>00</td>
<td>001</td>
<td>FCVTNU (scalar) - Double-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00</td>
<td>00</td>
<td>010</td>
<td>SCVTF (scalar, integer) - 32-bit to double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00</td>
<td>00</td>
<td>011</td>
<td>UCVTF (scalar, integer) - 32-bit to double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00</td>
<td>10</td>
<td>000</td>
<td>FMOV (general) - Double-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00</td>
<td>10</td>
<td>001</td>
<td>FMOV (general) - 32-bit to double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 01</td>
<td>01</td>
<td>000</td>
<td>FCVTMS (scalar) - Double-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 01</td>
<td>01</td>
<td>001</td>
<td>FCVTMU (scalar) - Double-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 10</td>
<td>11x</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11</td>
<td>00</td>
<td>000</td>
<td>FCVTZS (scalar, integer) - Double-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11</td>
<td>00</td>
<td>001</td>
<td>FCVTZU (scalar, integer) - Double-precision to 32-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11</td>
<td>110</td>
<td>-</td>
<td>FCVTZS FEAT_JSCVT</td>
<td>-</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>ptype</th>
<th>rmode</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>-</td>
<td>11x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>000</td>
<td>FCVTNS (scalar) - Half-precision to 32-bit variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>001</td>
<td>FCVTNU (scalar) - Half-precision to 32-bit variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>010</td>
<td>SCVT (scalar, integer) - 32-bit to half-precision variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>011</td>
<td>UCVT (scalar, integer) - 32-bit to half-precision variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>100</td>
<td>FMOV (general) - Half-precision to 32-bit variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>101</td>
<td>FCVT (scalar) - Single-precision to 64-bit variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>10</td>
<td>000</td>
<td>FCVT (scalar) - Half-precision to 32-bit variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>10</td>
<td>001</td>
<td>FCVT (scalar) - Half-precision to 32-bit variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>000</td>
<td>FCVT (scalar) - Half-precision to 32-bit variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>001</td>
<td>FMOV (general) - 32-bit to half-precision variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>00</td>
<td>-</td>
<td>11x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>000</td>
<td>FCVTNS (scalar) - Single-precision to 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>001</td>
<td>FCVTNU (scalar) - Single-precision to 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>010</td>
<td>SCVT (scalar, integer) - 64-bit to single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>011</td>
<td>UCVT (scalar, integer) - 64-bit to single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>100</td>
<td>FMOV (general) - 64-bit to single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>101</td>
<td>FMOV (general) - Single-precision to 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>01</td>
<td>000</td>
<td>FCVT (scalar) - Single-precision to 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>01</td>
<td>001</td>
<td>FMOV (general) - Single-precision to 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>10</td>
<td>000</td>
<td>FCVT (scalar) - Single-precision to 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>10</td>
<td>001</td>
<td>FCVT (scalar) - Single-precision to 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>000</td>
<td>FCVT (scalar, integer) - Single-precision to 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>001</td>
<td>FCVT (scalar, integer) - Single-precision to 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>x1</td>
<td>11x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>000</td>
<td>FCVTNS (scalar) - Double-precision to 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>001</td>
<td>FCVTNU (scalar) - Double-precision to 64-bit variant</td>
<td>-</td>
</tr>
<tr>
<td>Decode fields</td>
<td>Instruction page</td>
<td>Feature</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------</td>
<td>------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sf S ptype rmode opcode</td>
<td>SCVT (scalar, integer) - 64-bit to double-precision variant</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 01 00 010</td>
<td>UCVTF (scalar, integer) - 64-bit to double-precision variant</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 01 00 011</td>
<td>FCVTAS (scalar) - Double-precision to 64-bit variant</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 01 00 100</td>
<td>FCVTAU (scalar) - Double-precision to 64-bit variant</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 01 00 110</td>
<td>FMOV (general) - Double-precision to 64-bit variant</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 01 00 111</td>
<td>FMOV (general) - 64-bit to double-precision variant</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 01 01 000</td>
<td>FCVTPS (scalar) - Double-precision to 64-bit variant</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 01 01 001</td>
<td>FCVTPU (scalar) - Double-precision to 64-bit variant</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 01 1x 11x</td>
<td>Unallocated.</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 01 10 000</td>
<td>FCVTMS (scalar) - Double-precision to 64-bit variant</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 01 10 001</td>
<td>FCVTMU (scalar) - Double-precision to 64-bit variant</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 01 11 000</td>
<td>FCVTZS (scalar, integer) - Double-precision to 64-bit variant</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 01 11 001</td>
<td>FCVTZU (scalar, integer) - Double-precision to 64-bit variant</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 10 x0 11x</td>
<td>Unallocated.</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 10 01 110</td>
<td>FMOV (general) - Top half of 128-bit to 64-bit variant</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 10 01 111</td>
<td>FMOV (general) - 64-bit to top half of 128-bit variant</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 10 1x 11x</td>
<td>Unallocated.</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 11 00 000</td>
<td>FCVTNS (scalar) - Half-precision to 64-bit variant</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 11 00 001</td>
<td>FCVTNU (scalar) - Half-precision to 64-bit variant</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 11 00 010</td>
<td>SCVT (scalar, integer) - 64-bit to half-precision variant</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 11 00 011</td>
<td>UCVTF (scalar, integer) - 64-bit to half-precision variant</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 11 00 100</td>
<td>FCVTAS (scalar) - Half-precision to 64-bit variant</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 11 00 101</td>
<td>FCVTAU (scalar) - Half-precision to 64-bit variant</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 11 00 110</td>
<td>FMOV (general) - Half-precision to 64-bit variant</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 11 00 111</td>
<td>FMOV (general) - 64-bit to half-precision variant</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 11 01 000</td>
<td>FCVTPS (scalar) - Half-precision to 64-bit variant</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 11 01 001</td>
<td>FCVTPU (scalar) - Half-precision to 64-bit variant</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 11 10 000</td>
<td>FCVTMS (scalar) - Half-precision to 64-bit variant</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 11 10 001</td>
<td>FCVTMU (scalar) - Half-precision to 64-bit variant</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 11 11 000</td>
<td>FCVTZS (scalar, integer) - Half-precision to 64-bit variant</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 11 11 001</td>
<td>FCVTZU (scalar, integer) - Half-precision to 64-bit variant</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Floating-point data-processing (1 source)

This section describes the encoding of the Floating-point data-processing (1 source) instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>M S ptype opcode</td>
<td>100000</td>
<td></td>
</tr>
<tr>
<td>0 0 00 000000</td>
<td>FMOV (register) - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 000001</td>
<td>FABS (scalar) - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 000010</td>
<td>FNEG (scalar) - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 000011</td>
<td>FSQRT (scalar) - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 000100</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 000101</td>
<td>FCVT - Single-precision to double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 000110</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 000111</td>
<td>FCVT - Single-precision to half-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 001000</td>
<td>FRINTN (scalar) - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 001001</td>
<td>FRINTP (scalar) - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 001010</td>
<td>FRINTM (scalar) - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 001011</td>
<td>FRINTZ (scalar) - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 001100</td>
<td>FRINTA (scalar) - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 001101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 001110</td>
<td>FRINTX (scalar) - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 001111</td>
<td>FRINTI (scalar) - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 010000</td>
<td>FRINT32Z (scalar) - Single-precision variant</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0 0 00 010001</td>
<td>FRINT32X (scalar) - Single-precision variant</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0 0 00 010010</td>
<td>FRINT64Z (scalar) - Single-precision variant</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0 0 00 010011</td>
<td>FRINT64X (scalar) - Single-precision variant</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0 0 00 0101xx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 011xxx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 000000</td>
<td>FMOV (register) - Double-precision variant</td>
<td>-</td>
</tr>
</tbody>
</table>
### A64 Instruction Set Encoding

#### C4.1 A64 instruction set encoding

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>ptype</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000001</td>
<td>FABS (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000101</td>
<td>FNEG (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000111</td>
<td>FSQRT (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000100</td>
<td>FCVT - Double-precision to single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000110</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000110</td>
<td>BFCVT FEAT_BF16</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001000</td>
<td>FRINTN (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001001</td>
<td>FRINTP (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001010</td>
<td>FRINTM (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001011</td>
<td>FRINTZ (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001100</td>
<td>FRINTA (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001101</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001110</td>
<td>FRINTX (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001111</td>
<td>FRINTI (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010000</td>
<td>FRINT32Z (scalar) - Double-precision variant FEAT_FRINTTS</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010001</td>
<td>FRINT32X (scalar) - Double-precision variant FEAT_FRINTTS</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010010</td>
<td>FRINT64Z (scalar) - Double-precision variant FEAT_FRINTTS</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010011</td>
<td>FRINT64X (scalar) - Double-precision variant FEAT_FRINTTS</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0101xx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>011xxx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0xxxxx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000000</td>
<td>FMOV (register) - Half-precision variant FEAT_FP16</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000001</td>
<td>FABS (scalar) - Half-precision variant FEAT_FP16</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000010</td>
<td>FNEG (scalar) - Half-precision variant FEAT_FP16</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000011</td>
<td>FSQRT (scalar) - Half-precision variant FEAT_FP16</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000100</td>
<td>FCVT - Half-precision to single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000101</td>
<td>FCVT - Half-precision to double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00011x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001000</td>
<td>FRINTN (scalar) - Half-precision variant FEAT_FP16</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001001</td>
<td>FRINTP (scalar) - Half-precision variant FEAT_FP16</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001010</td>
<td>FRINTM (scalar) - Half-precision variant FEAT_FP16</td>
<td>-</td>
</tr>
</tbody>
</table>
Floating-point compare

This section describes the encoding of the Floating-point compare instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>ptype</th>
<th>opcode</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001011</td>
<td>FRINTZ (scalar) - Half-precision variant FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001100</td>
<td>FRINTA (scalar) - Half-precision variant FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001101</td>
<td>Unallocated. -</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001110</td>
<td>FRINTX (scalar) - Half-precision variant FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001111</td>
<td>FRINTI (scalar) - Half-precision variant FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>01xxxx</td>
<td>Unallocated. -</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated. -</td>
</tr>
</tbody>
</table>

---

**Decode fields**

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>ptype</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001011</td>
<td>FRINTZ (scalar) - Half-precision variant FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001100</td>
<td>FRINTA (scalar) - Half-precision variant FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001101</td>
<td>Unallocated. -</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001110</td>
<td>FRINTX (scalar) - Half-precision variant FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001111</td>
<td>FRINTI (scalar) - Half-precision variant FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>01xxxx</td>
<td>Unallocated. -</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated. -</td>
<td></td>
</tr>
</tbody>
</table>

---

**Decode fields**

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>ptype</th>
<th>opcode2</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>000000</td>
<td>FCMP -</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>010000</td>
<td>FCMP -</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>100000</td>
<td>FCMP -</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>110000</td>
<td>FCMP -</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>000000</td>
<td>FCMP -</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>010000</td>
<td>FCMP -</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>100000</td>
<td>FCMP -</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>000000</td>
<td>FCMP FEAT_FP16</td>
</tr>
</tbody>
</table>
### Floating-point immediate

This section describes the encoding of the Floating-point immediate instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>S</td>
<td>ptype</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### Floating-point conditional compare

This section describes the encoding of the Floating-point conditional compare instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>S</td>
<td>ptype</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Floating-point data-processing (2 source)

This section describes the encoding of the Floating-point data-processing (2 source) instruction class. The encodings in this section are decoded from Data Processing -- Scalar Floating-Point and Advanced SIMD on page C4-601.
### Floating-point conditional select

This section describes the encoding of the Floating-point conditional select instruction class. The encodings in this section are decoded from *Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-601.

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>ptype</th>
<th>opcode</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0101</td>
<td>FMIN (scalar) - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0110</td>
<td>FMAXNM (scalar) - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0111</td>
<td>FMINNM (scalar) - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>1000</td>
<td>FNMUL (scalar) - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0000</td>
<td>FMUL (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0001</td>
<td>FDIV (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0010</td>
<td>FADD (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0011</td>
<td>FSUB (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0100</td>
<td>FMAX (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0101</td>
<td>FMIN (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0110</td>
<td>FMAXNM (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0111</td>
<td>FMINNM (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>1000</td>
<td>FNMUL (scalar) - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0000</td>
<td>FMUL (scalar) - Half-precision variant FEAT_FP16</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0001</td>
<td>FDIV (scalar) - Half-precision variant FEAT_FP16</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0010</td>
<td>FADD (scalar) - Half-precision variant FEAT_FP16</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0011</td>
<td>FSUB (scalar) - Half-precision variant FEAT_FP16</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0100</td>
<td>FMAX (scalar) - Half-precision variant FEAT_FP16</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0101</td>
<td>FMIN (scalar) - Half-precision variant FEAT_FP16</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0110</td>
<td>FMAXNM (scalar) - Half-precision variant FEAT_FP16</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0111</td>
<td>FMINNM (scalar) - Half-precision variant FEAT_FP16</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>1000</td>
<td>FNMUL (scalar) - Half-precision variant FEAT_FP16</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
Floating-point data-processing (3 source)

This section describes the encoding of the Floating-point data-processing (3 source) instruction class. The encodings in this section are decoded from Data Processing -- Scalar Floating-Point and Advanced SIMD on page C4-601.

[31 30 29 28 | 27 26 25 24 | 23 22 21 20] | 16 | 15 | 12 | 11 | 10 | 9 | 5 | 4 | 0 | M | 0 | S | 1 | 1 | 1 | 1 | ptype | 1 | Rm | cond | 1 | 1 | Rn | Rd

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>M S ptype</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- - 10</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>- 1 -</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00</td>
<td>FCSEL - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 01</td>
<td>FCSEL - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 11</td>
<td>FCSEL - Half-precision variant FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>1 - -</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M S ptype o1 00</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- - 10 -</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>- 1 - -</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 00</td>
<td>FMADD - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 01</td>
<td>FMSUB - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 10</td>
<td>FNMADD - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 00 11</td>
<td>FNMSUB - Single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 01 00</td>
<td>FMADD - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 01 01</td>
<td>FMSUB - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 01 10</td>
<td>FNMADD - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 01 11</td>
<td>FNMSUB - Double-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>0 0 11 00</td>
<td>FMADD - Half-precision variant FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0 0 11 01</td>
<td>FMSUB - Half-precision variant FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>Decode fields</td>
<td>Instruction page</td>
<td>Feature</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------</td>
<td>---------</td>
</tr>
<tr>
<td>M  S  ptype  o1  o0</td>
<td>FNMADD - Half-precision variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0  0  11  1  0</td>
<td>FNMSUB - Half-precision variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1  -  -  -  -</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
A64 Instruction Set Encoding
C4.1 A64 instruction set encoding
Chapter C5
The A64 System Instruction Class

This chapter describes the A64 System instruction class, and the System instruction class encoding space, that is a subset of the System registers encoding space. It contains the following sections:

- The System instruction class encoding space on page C5-654.
- Special-purpose registers on page C5-671.
- A64 System instructions for cache maintenance on page C5-776.
- A64 System instructions for address translation on page C5-841.
- A64 System instructions for TLB maintenance on page C5-866.
- A64 System instructions for prediction restriction on page C5-1146.
- A64 System instructions for the Branch Record Buffer Extension on page C5-1159.

See General information about the A64 instruction descriptions on page C2-243 for information about entries used in the instruction encoding descriptions.
C5.1 The System instruction class encoding space

Part of the A64 instruction encoding space is assigned to instructions that access the System register encoding space. These instructions provide:

• Access to System registers, including the debug registers, that provide system control, and system status information.
• Access to Special-purpose registers such as SPSR_ELx, ELR_ELx, and the equivalent fields of the Process State.
• The cache and TLB maintenance instructions and address translation instructions.
• Barriers and the CLREX instruction.
• Architectural hint instructions.

This section describes the general model for accessing this functionality.

--- Note ---

• See Fixed values in AArch64 instruction and System register descriptions on page C2-243 for information about abbreviations used in the System instruction descriptions.

• In AArch32 state much of this functionality is provided through the System register interface described in The AArch32 System register interface on page G1-8993. In AArch64 state, the parameters used to characterize the System register encoding space are \{op0, op1, Crn, Crn, op2\}. These are based on the parameters that characterize the AArch32 System register encoding space, which reflect the original implementation of these registers, as described in Background to the System register interface on page G1-8994. There is no particular significance to the naming of these parameters, and no functional distinction between the opn parameters and the CRx parameters.

Principles of the System instruction class encoding describes some general properties of these encodings. System instruction class encoding overview on page C5-655 then describes the top-level encoding of these instructions, and the following sections then describe the next level of the encoding hierarchy of System instructions and Special-purpose registers:

• \(op0==0b00\), architectural hints, barriers and CLREX, and PSTATE access on page C5-656.
• \(op0==0b01\), cache maintenance, TLB maintenance, address translation, prediction restriction, and BRBE instructions on page C5-659.
• \(op0==0b11\), Moves to and from Special-purpose registers on page C5-668.

For the description of the next level of encoding hierarchy of System registers, see:

• \(op0==0b10\), Moves to and from debug and trace System registers on page D16-5519.
• \(op0==0b11\), Moves to and from non-debug System registers, Special-purpose registers on page D16-5523.
• Reserved encodings for IMPLEMENTATION DEFINED registers on page D16-5542.

C5.1.1 Principles of the System instruction class encoding

An encoding in the System instruction space is identified by a set of arguments, \{op0, op1, Crn, Crn, op2\}. These form an encoding hierarchy, where:

\(op0\) Defines the top-level division of the encoding space, see System instruction class encoding overview on page C5-655.

\(op1\) Identifies the lowest Exception level at which the encoding is accessible, as follows:

Accessible at EL0 \( op1 \) has the value 3.
Accessible at EL1 \( op1 \) has the value 0, 1, or 2. The value is the same as the \( op1 \) value used to access the equivalent AArch32 register.

Accessible at Secure EL1 \( op1 \) has the value 7.

Accessible at EL2 \( op1 \) has the value 4 or 5. The value 5 is used for the EL12 encodings that access EL1 System registers used when FEAT_VHE is implemented and HCR_EL2.E2H is 1.
Accessible at EL3  op1 has the value 6.

Arm strongly recommends that implementers adopt this use of op1 when using the IMPLEMENTATION DEFINED regions of the encoding space described in Reserved encodings for IMPLEMENTATION DEFINED registers on page D16-5542.

C5.1.2  System instruction class encoding overview

The encoding of the System instruction class describes each instruction as being either:
- A transfer to a System register. This is a System instruction with the semantics of a write.
- A transfer from a System register. This is a System instruction with the semantics of a read.

A System instruction that initiates an operation operates as if it was making a transfer to a register.

In the AArch64 instruction set, the decode structure for the System instruction class is:

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16 15</th>
<th>12 11 8 7</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 1 0</td>
<td>L</td>
<td>op0</td>
<td>op1</td>
<td>CRn</td>
<td>CRm</td>
<td>op2</td>
</tr>
</tbody>
</table>
```

The value of L indicates the transfer direction:
- 0  Transfer to System register.
- 1  Transfer from System register.

The op0 field is the top level encoding of the System instruction type. Its possible values are:

- **0b00** These encodings provide:
  - Instructions with an immediate field for accessing PSTATE, the current PE state.
  - The architectural hint instructions.
  - Barriers and the CLREX instruction.

  For more information about these encodings, see op0==0b00, architectural hints, barriers and CLREX, and PSTATE access on page C5-656.

- **0b01** These encodings provide:
  - Cache maintenance instructions.
  - TLB maintenance instructions.
  - Address translation instructions.
  - Prediction restriction instructions.
  - BRBE instructions.

  **Note** These are equivalent to operations in the AArch32 (coproc==0b1111) encoding space.

  For more information, see op0==0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, and BRBE instructions on page C5-659.

- **0b10** These encodings provide moves to and from:
  - Legacy AArch32 System registers for execution environments, to provide access to these registers from higher Exception levels that are using AArch64.
  - Debug and trace registers.

  **Note** These are equivalent to the registers in the AArch32 (coproc==0b1110) encoding space.

  For more information, see op0==0b10, Moves to and from debug and trace System registers on page D16-5519.
The A64 System Instruction Class
C5.1 The System instruction class encoding space

0b11

These encodings provide:
• Moves to and from Non-debug System registers. The accessed registers provide system control, and system status information.

Note
The accessed registers are equivalent to the registers in the AArch32 (coproc==0b1111) encoding space.

• Access to Special-purpose registers.

For more information, see Instructions for accessing Special-purpose registers on page C5-668 and Instructions for accessing non-debug System registers on page D16-5523.

UNDEFINED behaviors

In the System register instruction encoding space, the following principles apply:

• All unallocated encodings are treated as UNDEFINED.
• All encodings with \( L = 1 \) and \( op0 = 0b0x \) are UNDEFINED, except for encodings in the area reserved for IMPLEMENTATION DEFINED use, see Reserved encoding space for IMPLEMENTATION DEFINED instructions on page C5-667.

For registers and operations that are accessible from a particular Exception level, any attempt to access those registers from a lower Exception level is UNDEFINED.

If a particular Exception level:

• Defines a register to be RO, then any attempt to write to that register, at that Exception level, is UNDEFINED. This means that any access to that register with \( L = 0 \) is UNDEFINED.
• Defines a register to be WO, then any attempt to read from that register, at that Exception level, is UNDEFINED. This means that any access to that register with \( L = 1 \) is UNDEFINED.

For IMPLEMENTATION DEFINED encoding spaces, the treatment of the encodings is IMPLEMENTATION DEFINED, but see the recommendation in Principles of the System instruction class encoding on page C5-654.

C5.1.3 \( op0 == 0b00 \), architectural hints, barriers and CLREX, and PSTATE access

The different groups of System register instructions with \( op0 == 0b00 \):

• Are identified by the value of \( CRn \).
• Are always encoded with a value of 0b11111 in the \( Rt \) field.

The encoding of these instructions is:

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 12 11 8 7 5 4 0 |
|------------------------|----------|----------|----------|----------|----------|
| 1 1 0 1 0 1 0 1 0 0 L 0 0 op1 CRn CRm op2 1 1 1 1 1 |

The encoding of the \( CRn \) field is as follows:

0b0010  See Architectural hint instructions.
0b0011  See Barriers and CLREX on page C5-657.
0b0100  See Instructions for accessing the PSTATE fields on page C5-658.

Architectural hint instructions

Within the \( op0 == 0b00 \) encodings, the architectural hint instructions are identified by \( CRn \) having the value 0b0010. The encoding of these instructions is:

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 12 11 5 4 0 |
|------------------------|----------|----------|----------|----------|----------|
| 1 1 0 1 0 1 0 1 0 0 L 0 0 CRn CRm op2 1 1 1 1 1 |
The value of op<6:0>, formed by concatenating the CRm and op2 fields, determines the hint instruction as follows:

0b000000  NOP instruction.
0b000001  YIELD instruction.
0b000010  WFE instruction.
0b000011  WFI instruction.
0b000010  SEV instruction.
0b000101  SEVL instruction.
0b000110  DGH instruction.
0b000111  XPACD, XPACI, XPACLRI instruction.
0b001000  PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA instruction, PACIA1716 variant.
0b001010  PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB instruction, PACIB1716 variant.
0b001100  AUTIA, AUTIA1716, AUTIASP, AUTIAZ instruction, AUTIA1716 variant.
0b001110  AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB instruction, AUTIB1716 variant.
0b010000  ESB instruction.
0b010001  PSB CSYNC instruction.
0b010010  TSB CSYNC instruction.
0b010100  CSDB instruction.
0b011000  PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA instruction, PACIAZ variant.
0b011001  PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA instruction, PACIASP variant.
0b011010  PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB instruction, PACIBZ variant.
0b011011  PACIB, PACIB176, PACIBSP, PACIBZ, PACIZB instruction, PACIBSP variant.
0b011100  AUTIA, AUTIA1716, AUTIASP, AUTIAZ instruction, AUTIAZ variant.
0b011101  AUTIA, AUTIA1716, AUTIASP, AUTIAZ instruction, AUTIASP variant.
0b011110  AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB instruction, AUTIBZ variant.
0b011111  AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB instruction, AUTIBSP variant.
0b100xx0  BTI instruction.

These instructions are described in Chapter C6 A64 Base Instruction Descriptions.

--- Note ---
- Instruction encodings with bits[4:0] not set to 0b11111 are UNDEFINED.
- The operation of the A64 instructions for architectural hints are identical to the corresponding A32 and T32 instructions.

For more information about:
- The WFE, WFI, SEV, and SEVL instructions, see Mechanisms for entering a low-power state on page D1-4676.
- The YIELD instruction, see Software control features and EL0 on page B1-146.

**Barriers and CLREX**

Within the op0==0b0 encodings, the barriers and CLREX instructions are identified by CRn having the value 0b1011. The encoding of these instructions is:

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 12 11 8 7 5 4 0 |
|------------------------|------------------|------------------|------------------|------------------|
| 1 1 0 1 0 1 0 1 0 0 0 0 0 1 1 0 1 1 1 0 1 1 1 1 0 |
| op0      op1      CRn      op2      1 1 1 1 1 1 |

The value of op2 determines the instruction, as follows:

0b001  DSB instruction, Memory nXS barrier variant.
0b010  CLREX instruction.
0b100  DSB instruction, Memory barrier variant.
0b101  DMB instruction.
The A64 System Instruction Class
C5.1 The System instruction class encoding space

0b110 ISB instruction.
0b000, 0b011, 0b111 UNDEFINED.

These instructions are described in Chapter C6 A64 Base Instruction Descriptions.

Note
- Instruction encodings with bits[4:0] not set to 0b11111 are UNDEFINED.
- The operation of the A64 instructions for barriers and CLREX are identical to the corresponding A32 and T32 instructions.

For more information about:
- The barrier instructions, see Memory barriers on page B2-174.
- The CLREX instruction, see Synchronization and semaphores on page B2-211.

Instructions for accessing the PSTATE fields

Within the op0=0b00 encodings, the instructions that can be used to modify PSTATE fields directly are identified by CRn having the value 0b0100. The encoding of these instructions is shown in Figure C5-1 and Figure C5-2:

Figure C5-1 Instructions using #Imm4

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 12 11 8 7 5 4 0 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | op1 | 0 | 1 | 0 | Imm4 | op2 | 1 | 1 | 1 | 1 |
| op0 | CRn | CRm | Rt |

Figure C5-2 Instructions using #Imm1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 12 11 8 7 5 4 0 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Imm1 | op2 | 1 | 1 | 1 | 1 |
| op0 | CRn | CRm | Rt |

These instructions are:
- CFINV ; Inverts the value of PSTATE.C
- MSR DAIFSet, #Imm4 ; Used to set any or all of DAIF to 1
- MSR DAIFClr, #Imm4 ; Used to clear any or all of DAIF to 0
- MSR SPSel, #Imm4 ; Used to select the Stack Pointer, between SP_EL0 and SP_ELx
- MSR UA0, #Imm4 ; Used to set the value of PSTATE.UA0
- MSR PAN, #Imm4 ; Used to set the value of PSTATE.PAN
- MSR DIT, #Imm4 ; Used to set the value of PSTATE.DIT
- MSR SSBS, #Imm4 ; Used to set the value of PSTATE.SSBS
- MSR TCO, #Imm4 ; Used to set the value of PSTATE.TCO
- MSR ALLINT, #Imm1 ; Used to set the value of PSTATE.ALLINT

The value of op2 selects the instruction form, which defines the constraints on the values of the op1 and Imm4 or Imm1 arguments, as follows:

op2==0b000 Selects the CFINV instruction.
op2==0b011 Selects the MSR UA0 instruction.
op2==0b100 Selects the MSR PAN instruction.
op2==0b101 Selects the MSR SPSel instruction.
op2==0b001 Selects the MSR SSBS instruction.
op2==0b010 Selects the MSR DIT instruction.
op2==0b100 Selects the MSR TCO instruction.
C5.1 The System instruction class encoding space

op2==0b110 Selects the MSR DAIFSet instruction, that sets the specified PSTATE.\{D, A, I, F\} bits to 1.

op2==0b111 Selects the MSR DAIFClr instruction, that clears the specified PSTATE.\{D, A, I, F\} bits to 0.

op2==0b000 Selects the MSR ALLINT instruction.

All other combinations of op1 and op2 are reserved, and the corresponding instructions are UNDEFINED.

--- Note ---

For PSTATE updates, instruction encodings with bits[4:0] not set to 0b11111 are UNDEFINED.

Writes to PSTATE occur in program order without the need for additional synchronization. Changing PSTATE.SPSel to use SP_EL0 synchronizes any updates to SP_EL0 that have been written by an MSR to SP_EL0, without the need for additional synchronization.

C5.1.4 op0==0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, and BRBE instructions

The System instructions are encoded with op0==0b01. The different groups of System instructions are identified by the values of CRn and Crm, except that some of this encoding space is reserved for IMPLEMENTATION DEFINED functionality. The encoding of these instructions is:

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 12 11 8 7 5 4 0 |
|-------------------------|-----------------|-----------------|-----------------|
| 1 1 0 1 0 1 0 0 0 1 0 1  | CRn CRm op2 Xt  |
| op0                     |

The grouping of these instructions depending on the CRn and Crm fields is as follows:

CRn==7 The instruction group is determined by the value of CRn, as follows:

CRn=={1, 5} Instruction cache maintenance instructions.
See Cache maintenance instructions, and data cache zero operation.

CRn==2 Branch Record Buffer instructions.
See Branch Record Buffer instructions on page C5-667.

CRn==3 Prediction restriction instructions.
See Prediction restriction instructions on page C5-661.

CRn==4 Data cache zero operation.
See Cache maintenance instructions, and data cache zero operation.

CRn=={6, 10, 11, 12, 14} Data cache maintenance instructions.
See Cache maintenance instructions, and data cache zero operation.

CRn==8 See Address translation instructions on page C5-661.

CRn=={8, 9} See TLB maintenance instructions on page C5-662.

CRn=={11, 15} See Reserved encoding space for IMPLEMENTATION DEFINED instructions on page C5-667.

Cache maintenance instructions, and data cache zero operation

Table C5-1 on page C5-660 lists the Cache maintenance instructions and their encodings. Instructions that take an argument include Xt in the instruction syntax. For instructions that do not take an argument, the Xt field is encoded as 0b11111. For these instructions, if the Xt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:

• The instruction is UNDEFINED.

ARM DDI 0487I.a Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. C5-659
ID081822 Non-Confidential
The instruction behaves as if the Xt field is set to 0b11111.

### Table C5-1 Cache maintenance instructions and data cache zero operation

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>000</td>
<td>0111</td>
<td>0001</td>
<td>000</td>
<td>IC IALLUIS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>0111</td>
<td>0101</td>
<td>000</td>
<td>IC IALLU</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>0101</td>
<td>001</td>
<td>IC IVAU</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>0111</td>
<td>0110</td>
<td>001</td>
<td>DC IVAC</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>0111</td>
<td>0110</td>
<td>010</td>
<td>DC ISW</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>0111</td>
<td>0110</td>
<td>011</td>
<td>DC IGVAC</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>0111</td>
<td>0110</td>
<td>100</td>
<td>DC IGSW</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>0111</td>
<td>0110</td>
<td>101</td>
<td>DC IGDVAC</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>0111</td>
<td>1110</td>
<td>010</td>
<td>DC CISW</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>0111</td>
<td>1110</td>
<td>100</td>
<td>DC CIGSW</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>0111</td>
<td>1110</td>
<td>110</td>
<td>DC CIGDSW</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>0100</td>
<td>001</td>
<td>DC ZVA</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>0100</td>
<td>011</td>
<td>DC GVA</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>0100</td>
<td>100</td>
<td>DC GZVA</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>1010</td>
<td>001</td>
<td>DC CVAC</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>1010</td>
<td>011</td>
<td>DC CGVAC</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>1010</td>
<td>101</td>
<td>DC CGDVAC</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>1011</td>
<td>001</td>
<td>DC CVAU</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>1100</td>
<td>001</td>
<td>DC CVAP</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>1100</td>
<td>011</td>
<td>DC CGVAP</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>1100</td>
<td>101</td>
<td>DC CGDVAP</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>1101</td>
<td>001</td>
<td>DC CVADP</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>1110</td>
<td>011</td>
<td>DC CGVADP</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>1110</td>
<td>101</td>
<td>DC CGDVADP</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>1110</td>
<td>001</td>
<td>DC CICVAC</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>1110</td>
<td>011</td>
<td>DC CIGVAC</td>
</tr>
</tbody>
</table>
Table C5-1 Cache maintenance instructions and data cache zero operation (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>1110</td>
<td>101</td>
<td>DC CIGDVAC</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>0111</td>
<td>1110</td>
<td>001</td>
<td>DC CIPAPA</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>0111</td>
<td>1110</td>
<td>101</td>
<td>DC CIGDPAPA</td>
</tr>
</tbody>
</table>

For more information about these instructions, see About cache maintenance in AArch64 state on page D7-5050 and A64 Cache maintenance instructions on page D7-5054.

Prediction restriction instructions

Table C5-2 on page C5-661 lists the Prediction restriction instructions and their encodings. Instructions that take an argument include Xt in the instruction syntax.

Table C5-2 Prediction restriction instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Prediction restriction encoding</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFP RCTX, Xt</td>
<td>01 3 7 3 4</td>
<td>When FEAT_SPECRES is implemented, accessible from EL0 or higher.</td>
</tr>
<tr>
<td>DVP RCTX, Xt</td>
<td>01 5</td>
<td></td>
</tr>
<tr>
<td>CPP RCTX, Xt</td>
<td>01 7</td>
<td></td>
</tr>
</tbody>
</table>

For more information about these instructions, see Execution, data prediction and prefetching restriction System instructions on page D7-5070.

Address translation instructions

Table C5-3 lists the Address translation instructions and their encodings. The syntax of the instructions includes Xt, that provides the address to be translated.

Table C5-3 Address translation instructions

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>000</td>
<td>0111</td>
<td>1000</td>
<td>000</td>
<td>AT S1E1R</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>0111</td>
<td>1000</td>
<td>001</td>
<td>AT S1E1W</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>0111</td>
<td>1000</td>
<td>010</td>
<td>AT S1E0R</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>0111</td>
<td>1000</td>
<td>011</td>
<td>AT S1E0W</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>0111</td>
<td>1001</td>
<td>000</td>
<td>AT S1E1RP</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>0111</td>
<td>1001</td>
<td>001</td>
<td>AT S1E1WP</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>0111</td>
<td>1000</td>
<td>000</td>
<td>AT S1E2R</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>0111</td>
<td>1000</td>
<td>001</td>
<td>AT S1E2W</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>0111</td>
<td>1000</td>
<td>100</td>
<td>AT S12E1R</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>0111</td>
<td>1000</td>
<td>101</td>
<td>AT S12E1W</td>
</tr>
</tbody>
</table>
The A64 System Instruction Class
C5.1 The System instruction class encoding space

Table C5-4 Address translation instructions (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>100</td>
<td>0111</td>
<td>1000</td>
<td>110</td>
<td>AT S12E0R</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>0111</td>
<td>1000</td>
<td>111</td>
<td>AT S12E0W</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>0111</td>
<td>1000</td>
<td>000</td>
<td>AT S1E3R</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>0111</td>
<td>1000</td>
<td>001</td>
<td>AT S1E3W</td>
</tr>
</tbody>
</table>

For more information about these instructions, see Address translation instructions on page D8-5120.

**TLB maintenance instructions**

Table C5-4 lists the TLB maintenance instructions and their encodings. Instructions that take an argument include Xt in the instruction syntax. For instructions that do not take an argument, the Xt field is encoded as 0b11111. For these instructions, if the Xt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to 0b11111.

Table C5-4 TLB maintenance instructions

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0001</td>
<td>000</td>
<td>TLBI VMALLE1OS, TLBI VMALLE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0001</td>
<td>001</td>
<td>TLBI VAE1OS, TLBI VAE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0001</td>
<td>010</td>
<td>TLBI ASIDE1OS, TLBI ASIDE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0001</td>
<td>011</td>
<td>TLBI VAAE1OS, TLBI VAAE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0001</td>
<td>101</td>
<td>TLBI VALE1OS, TLBI VALE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0001</td>
<td>111</td>
<td>TLBI VAALE1OS, TLBI VAALE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0010</td>
<td>001</td>
<td>TLBI RVAE1IS, TLBI RVAE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0010</td>
<td>011</td>
<td>TLBI RVAAE1IS, TLBI RVAAE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0010</td>
<td>101</td>
<td>TLBI RVALE1IS, TLBI RVALE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0010</td>
<td>111</td>
<td>TLBI RVAALE1IS, TLBI RVAALE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0011</td>
<td>000</td>
<td>TLBI VMALLE1IS, TLBI VMALLE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0011</td>
<td>001</td>
<td>TLBI VAE1IS, TLBI VAE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0011</td>
<td>010</td>
<td>TLBI ASIDE1IS, TLBI ASIDE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0011</td>
<td>011</td>
<td>TLBI VAAE1IS, TLBI VAAE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0011</td>
<td>101</td>
<td>TLBI VALE1IS, TLBI VALE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0011</td>
<td>111</td>
<td>TLBI VAALE1IS, TLBI VAALE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0101</td>
<td>001</td>
<td>TLBI RVAE1OS, TLBI RVAE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0101</td>
<td>011</td>
<td>TLBI RVAAE1OS, TLBI RVAAE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0101</td>
<td>101</td>
<td>TLBI RVALE1OS, TLBI RVALE1OSNXS</td>
</tr>
</tbody>
</table>
Table C5-4 TLB maintenance instructions (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0101</td>
<td>111</td>
<td>TLBI RVALE1OS, TLBI RVALE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0110</td>
<td>001</td>
<td>TLBI RVAE1, TLBI RVAE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0110</td>
<td>011</td>
<td>TLBI RVAE1, TLBI RVAE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0110</td>
<td>101</td>
<td>TLBI RVALE1, TLBI RVALE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0110</td>
<td>111</td>
<td>TLBI RVAE1, TLBI RVAE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0111</td>
<td>000</td>
<td>TLBI VMALE1, TLBI VMALE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0111</td>
<td>001</td>
<td>TLBI VAE1, TLBI VAE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0111</td>
<td>010</td>
<td>TLBI ASIDE1, TLBI ASIDE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0111</td>
<td>011</td>
<td>TLBI VAAE1, TLBI VAAE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0111</td>
<td>101</td>
<td>TLBI VALE1, TLBI VALE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1000</td>
<td>0111</td>
<td>111</td>
<td>TLBI VAAE1, TLBI VAAE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0001</td>
<td>000</td>
<td>TLBI VMALLE1OS, TLBI VMALLE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0001</td>
<td>001</td>
<td>TLBI VAE1OS, TLBI VAE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0001</td>
<td>010</td>
<td>TLBI ASIDE1OS, TLBI ASIDE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0001</td>
<td>011</td>
<td>TLBI VAAE1OS, TLBI VAAE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0001</td>
<td>101</td>
<td>TLBI VALE1OS, TLBI VALE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0001</td>
<td>111</td>
<td>TLBI VAAE1OS, TLBI VAAE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0010</td>
<td>001</td>
<td>TLBI RVAE1IS, TLBI RVAE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0010</td>
<td>011</td>
<td>TLBI RVAAE1IS, TLBI RVAAE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0010</td>
<td>101</td>
<td>TLBI RVALE1IS, TLBI RVALE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0010</td>
<td>111</td>
<td>TLBI RVAALE1IS, TLBI RVAALE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0011</td>
<td>000</td>
<td>TLBI VMALLE1IS, TLBI VMALLE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0011</td>
<td>001</td>
<td>TLBI VAE1IS, TLBI VAE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0011</td>
<td>010</td>
<td>TLBI ASIDE1IS, TLBI ASIDE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0011</td>
<td>011</td>
<td>TLBI VAAE1IS, TLBI VAAE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0011</td>
<td>101</td>
<td>TLBI VALE1IS, TLBI VALE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0011</td>
<td>111</td>
<td>TLBI VAAE1IS, TLBI VAAE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0101</td>
<td>001</td>
<td>TLBI RVAE1OS, TLBI RVAE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0101</td>
<td>011</td>
<td>TLBI RVAAE1OS, TLBI RVAAE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0101</td>
<td>101</td>
<td>TLBI RVALE1OS, TLBI RVALE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0101</td>
<td>111</td>
<td>TLBI RVAALE1OS, TLBI RVAALE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0110</td>
<td>001</td>
<td>TLBI RVAE1, TLBI RVAE1NXS</td>
</tr>
<tr>
<td>op0</td>
<td>op1</td>
<td>CRn</td>
<td>CRm</td>
<td>op2</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0110</td>
<td>011</td>
<td>TLBI RVAE1, TLBI RVAE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0110</td>
<td>101</td>
<td>TLBI RVALE1, TLBI RVALE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0110</td>
<td>111</td>
<td>TLBI RVAALE1, TLBI RVAALE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0111</td>
<td>000</td>
<td>TLBI VMALLE1, TLBI VMALLE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0111</td>
<td>001</td>
<td>TLBI VAE1, TLBI VAE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0111</td>
<td>010</td>
<td>TLBI ASIDE1, TLBI ASIDE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0111</td>
<td>011</td>
<td>TLBI VAAE1, TLBI VAAE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0111</td>
<td>101</td>
<td>TLBI VALE1, TLBI VALE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>1001</td>
<td>0111</td>
<td>111</td>
<td>TLBI VAALE1, TLBI VAALE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0000</td>
<td>001</td>
<td>TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0000</td>
<td>010</td>
<td>TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0000</td>
<td>101</td>
<td>TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0000</td>
<td>110</td>
<td>TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0001</td>
<td>000</td>
<td>TLBI ALLE2OS, TLBI ALLE2OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0001</td>
<td>001</td>
<td>TLBI VAE2OS, TLBI VAE2OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0001</td>
<td>100</td>
<td>TLBI ALLE1OS, TLBI ALLE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0001</td>
<td>101</td>
<td>TLBI VALE2OS, TLBI VALE2OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0001</td>
<td>110</td>
<td>TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0010</td>
<td>001</td>
<td>TLBI RVAE2IS, TLBI RVAE2ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0010</td>
<td>101</td>
<td>TLBI RVALE2IS, TLBI RVALE2ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0011</td>
<td>000</td>
<td>TLBI ALLE2IS, TLBI ALLE2ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0011</td>
<td>001</td>
<td>TLBI VAE2IS, TLBI VAE2ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0011</td>
<td>100</td>
<td>TLBI ALLE1IS, TLBI ALLE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0011</td>
<td>101</td>
<td>TLBI VALE2IS, TLBI VALE2ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0011</td>
<td>110</td>
<td>TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0100</td>
<td>000</td>
<td>TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0100</td>
<td>001</td>
<td>TLBI IPAS2E1, TLBI IPAS2E1NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0100</td>
<td>010</td>
<td>TLBI RIPAS2E1, TLBI RIPAS2E1NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0100</td>
<td>011</td>
<td>TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0100</td>
<td>100</td>
<td>TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0100</td>
<td>101</td>
<td>TLBI IPAS2E1, TLBI IPAS2E1NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0100</td>
<td>110</td>
<td>TLBI RIPAS2E1, TLBI RIPAS2E1NXS</td>
</tr>
<tr>
<td>op0</td>
<td>op1</td>
<td>CRn</td>
<td>CRm</td>
<td>op2</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>------------------------------</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0100</td>
<td>111</td>
<td>TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0101</td>
<td>001</td>
<td>TLBI RVAE2OS, TLBI RVAE2OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0101</td>
<td>101</td>
<td>TLBI RVALE2OS, TLBI RVALE2OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0110</td>
<td>001</td>
<td>TLBI RVAE2, TLBI RVAE2NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0110</td>
<td>101</td>
<td>TLBI RVALE2, TLBI RVALE2NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0111</td>
<td>000</td>
<td>TLBI ALLE2, TLBI ALLE2NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0111</td>
<td>001</td>
<td>TLBI VAE2, TLBI VAE2NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0111</td>
<td>100</td>
<td>TLBI ALLE1, TLBI ALLE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0111</td>
<td>101</td>
<td>TLBI VAE2, TLBI VAE2NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0000</td>
<td>001</td>
<td>TLBI ALLE2OS, TLBI ALLE2OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0000</td>
<td>010</td>
<td>TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0000</td>
<td>101</td>
<td>TLBI VAE2OS, TLBI VAE2OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0001</td>
<td>000</td>
<td>TLBI ALLE2OS, TLBI ALLE2OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0001</td>
<td>100</td>
<td>TLBI VAE2OS, TLBI VAE2OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0001</td>
<td>101</td>
<td>TLBI ALLE1OS, TLBI ALLE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0001</td>
<td>110</td>
<td>TLBI VALE2OS, TLBI VALE2OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0001</td>
<td>110</td>
<td>TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0010</td>
<td>001</td>
<td>TLBI RVAE2IS, TLBI RVAE2ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0010</td>
<td>101</td>
<td>TLBI VALE2IS, TLBI VALE2ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0011</td>
<td>000</td>
<td>TLBI ALLE2IS, TLBI ALLE2ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0011</td>
<td>001</td>
<td>TLBI VAE2IS, TLBI VAE2ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0011</td>
<td>100</td>
<td>TLBI ALLE1IS, TLBI ALLE1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0011</td>
<td>101</td>
<td>TLBI VALE2IS, TLBI VALE2ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0011</td>
<td>110</td>
<td>TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0100</td>
<td>000</td>
<td>TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0100</td>
<td>001</td>
<td>TLBI IPAS2E1, TLBI IPAS2E1NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0100</td>
<td>010</td>
<td>TLBI RIPAS2E1, TLBI RIPAS2E1NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0100</td>
<td>011</td>
<td>TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0100</td>
<td>100</td>
<td>TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0100</td>
<td>101</td>
<td>TLBI IPAS2LE1, TLBI IPAS2LE1NXS</td>
</tr>
</tbody>
</table>
### Table C5-4 TLB maintenance instructions (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0100</td>
<td>110</td>
<td>TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0100</td>
<td>111</td>
<td>TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0101</td>
<td>001</td>
<td>TLBI RVAE2OS, TLBI RVAE2OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1001</td>
<td>0101</td>
<td>101</td>
<td>TLBI RVALE2OS, TLBI RVALE2OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1011</td>
<td>111</td>
<td>000</td>
<td>TLBI ALLE2, TLBI ALLE2NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1011</td>
<td>111</td>
<td>001</td>
<td>TLBI VAE2, TLBI VAE2NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1011</td>
<td>111</td>
<td>100</td>
<td>TLBI ALLE1, TLBI ALLE1NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1011</td>
<td>111</td>
<td>101</td>
<td>TLBI VALE2, TLBI VALE2NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>111</td>
<td>110</td>
<td>101</td>
<td>TLBI RVALE2, TLBI RVALE2NXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0001</td>
<td>000</td>
<td>TLBI ALLE3OS, TLBI ALLE3OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0001</td>
<td>100</td>
<td>TLBI PAALLOS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0010</td>
<td>001</td>
<td>TLBI VALE3OS, TLBI VALE3OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>1000</td>
<td>0010</td>
<td>011</td>
<td>TLBI RVAE3IS, TLBI RVAE3ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1000</td>
<td>0011</td>
<td>000</td>
<td>TLBI ALLE3IS, TLBI ALLE3ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1000</td>
<td>0011</td>
<td>011</td>
<td>TLBI VAE3IS, TLBI VAE3ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1000</td>
<td>0110</td>
<td>101</td>
<td>TLBI VALE3IS, TLBI VALE3ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1000</td>
<td>0110</td>
<td>010</td>
<td>TLBI RPAOS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1000</td>
<td>0100</td>
<td>111</td>
<td>TLBI RPALOS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1000</td>
<td>0100</td>
<td>101</td>
<td>TLBI RVAE3OS, TLBI RVAE3OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1000</td>
<td>0110</td>
<td>001</td>
<td>TLBI RVAE3, TLBI RVAE3NXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1000</td>
<td>0110</td>
<td>101</td>
<td>TLBI RVALE3, TLBI RVALE3NXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1000</td>
<td>0111</td>
<td>000</td>
<td>TLBI ALLE3, TLBI ALLE3NXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1000</td>
<td>0111</td>
<td>001</td>
<td>TLBI VAE3, TLBI VAE3NXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1000</td>
<td>0111</td>
<td>100</td>
<td>TLBI PAALL</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1000</td>
<td>0111</td>
<td>101</td>
<td>TLBI VALE3, TLBI VALE3NXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>0001</td>
<td>0001</td>
<td>001</td>
<td>TLBI ALLE3OS, TLBI ALLE3OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>0001</td>
<td>0001</td>
<td>100</td>
<td>TLBI VAE3OS, TLBI VAE3OSNXS</td>
</tr>
</tbody>
</table>
The A64 System Instruction Class

C5.1 The System instruction class encoding space

Table C5-4 TLB maintenance instructions (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>110</td>
<td>1001</td>
<td>0001</td>
<td>101</td>
<td>TLBI VALE3OS, TLBI VALE3OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1001</td>
<td>0010</td>
<td>001</td>
<td>TLBI RVAE3IS, TLBI RVAE3ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1001</td>
<td>0010</td>
<td>101</td>
<td>TLBI RVAE3IS, TLBI RVAE3ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1001</td>
<td>0011</td>
<td>000</td>
<td>TLBI ALLE3IS, TLBI ALLE3ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1001</td>
<td>0011</td>
<td>001</td>
<td>TLBI VAE3IS, TLBI VAE3ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1001</td>
<td>0011</td>
<td>101</td>
<td>TLBI VALE3IS, TLBI VALE3ISNXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1001</td>
<td>1010</td>
<td>001</td>
<td>TLBI RVAE3OS, TLBI RVAE3OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1001</td>
<td>1010</td>
<td>101</td>
<td>TLBI RVALE3OS, TLBI RVALE3OSNXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1001</td>
<td>0110</td>
<td>001</td>
<td>TLBI RVAE3, TLBI RVAE3NXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1001</td>
<td>0110</td>
<td>101</td>
<td>TLBI RVAE3, TLBI RVAE3NXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1001</td>
<td>0111</td>
<td>000</td>
<td>TLBI ALLE3, TLBI ALLE3NXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1001</td>
<td>0111</td>
<td>001</td>
<td>TLBI VAE3, TLBI VAE3NXS</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>1001</td>
<td>0111</td>
<td>101</td>
<td>TLBI VALE3, TLBI VALE3NXS</td>
</tr>
</tbody>
</table>

For more information about these instructions, see *TLB maintenance instructions* on page D8-5201.

Branch Record Buffer instructions

Table C5-5 lists the Branch Record Buffer instructions and their encodings.

Table C5-5 Branch Record Buffer instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Access instruction encoding</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>op0</td>
<td>op1</td>
</tr>
<tr>
<td>BRB IALL</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>BRB INJ</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

For more information, see *Chapter D15 The Branch Record Buffer Extension.*

Reserved encoding space for IMPLEMENTATION DEFINED instructions

The A64 instruction set reserves the following encoding space for IMPLEMENTATION DEFINED instructions:

The value of \( L \) defines the use of \( Rt \) as follows:

- **0**: \( Rt \) is an argument supplied to the instruction.
- **1**: \( Rt \) is a result returned by the instruction.

IMPLEMENTATION DEFINED instructions in this encoding space are accessed using the SYS and SYSL instructions, see *SYS* on page C6-1993 and *SYSL* on page C6-1995.
C5.1.5  **op0==0b11, Moves to and from Special-purpose registers**

The instructions that move data to and from non-debug System registers are encoded with \( \text{op0} == 0b11 \), except that some of this encoding space is reserved for IMPLEMENTATION DEFINED functionality. The encoding of these instructions is:

\[
\begin{array}{c|cccccccccc|c|c|c|c|c|c|c|c|c|c|c|c}
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & |L| & 1 & 1 & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} & \text{Rt} \\
\end{array}
\]

### Instructions for accessing Special-purpose registers

The value of \( \text{CRn} \) provides the next level of decode of these instructions. For Special-purpose registers, the value of \( \text{CRn} \) is 4.

The A64 instructions for accessing Special-purpose registers are:

- **MSR** `<Special-purpose register>`, \( \text{Xt} \); Write to Special-purpose register
- **MRS** \( \text{Xt}, <\text{Special-purpose register}> \); Read from Special-purpose register

For these accesses, \( \text{CRn} \) has the value 4. The encoding for Special-purpose register accesses is:

\[
\begin{array}{c|cccccccccc|c|c|c|c|c|c|c|c|c|c|c|c}
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & |L| & 1 & 1 & \text{op1} & 0 & 0 & \text{Rm} & \text{op2} & \text{Rt} \\
\end{array}
\]

The full list of Special-purpose registers is in Table C5-6 on page C5-668. The characteristic of a Special-purpose register is that all direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization.

Table C5-6 on page C5-668 lists the encodings for \( \text{op1} \), \( \text{CRn} \), and \( \text{op2} \) fields for accesses to the Special-purpose registers in AArch64.

### Table C5-6 Special-purpose register accesses

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL1</td>
<td>SPSR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL1</td>
<td>SPSR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL2</td>
<td>SPSR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL2</td>
<td>SPSR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ELR_EL1</td>
<td>ELR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ELR_EL1</td>
<td>ELR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ELR_EL2</td>
<td>ELR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ELR_EL2</td>
<td>ELR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>SP_EL0</td>
<td>SP_EL0</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0010</td>
<td>000</td>
<td>-</td>
<td>SPSel</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0010</td>
<td>010</td>
<td>-</td>
<td>CurrentEL</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0010</td>
<td>011</td>
<td>-</td>
<td>PAN</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0010</td>
<td>100</td>
<td>-</td>
<td>UAO</td>
<td>-</td>
</tr>
<tr>
<td>op0</td>
<td>op1</td>
<td>CRn</td>
<td>CRm</td>
<td>op2</td>
<td>Access</td>
<td>Mnemonic</td>
<td>Register</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>--------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0011</td>
<td>000</td>
<td>ALLINT</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0011</td>
<td>001</td>
<td>PM</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0110</td>
<td>000</td>
<td>RW</td>
<td>ICC_PMR_EL1</td>
<td>ICC_PMR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0110</td>
<td>000</td>
<td>RW</td>
<td>ICV_PMR_EL1</td>
<td>ICC_PMR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0110</td>
<td>000</td>
<td>RW</td>
<td>ICV_PMR_EL1</td>
<td>ICV_PMR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0010</td>
<td>000</td>
<td>-</td>
<td>NZCV</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0010</td>
<td>001</td>
<td>-</td>
<td>DAIF</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0010</td>
<td>010</td>
<td>-</td>
<td>SVCR</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0010</td>
<td>010</td>
<td>-</td>
<td>DIT</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0010</td>
<td>110</td>
<td>-</td>
<td>SSBS</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0010</td>
<td>111</td>
<td>-</td>
<td>TCO</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0100</td>
<td>000</td>
<td>RW</td>
<td>FPCR</td>
<td>FPCR</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0100</td>
<td>001</td>
<td>RW</td>
<td>FPSR</td>
<td>FPSR</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0101</td>
<td>000</td>
<td>RW</td>
<td>DSPSR_EL0</td>
<td>DSPSR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0101</td>
<td>001</td>
<td>RW</td>
<td>DLR_EL0</td>
<td>DLR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL1</td>
<td>SPSR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL2</td>
<td>SPSR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL2</td>
<td>SPSR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ELR_EL1</td>
<td>ELR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ELR_EL1</td>
<td>ELR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ELR_EL2</td>
<td>ELR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>SP_EL1</td>
<td>SP_EL1</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0011</td>
<td>000</td>
<td>RW</td>
<td>SPSR_irq</td>
<td>SPSR_irq</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0011</td>
<td>001</td>
<td>RW</td>
<td>SPSR_abt</td>
<td>SPSR_abt</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>SPSR_und</td>
<td>SPSR_und</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0011</td>
<td>011</td>
<td>RW</td>
<td>SPSR_fiq</td>
<td>SPSR_fiq</td>
</tr>
</tbody>
</table>
All direct and indirect reads and writes to Special-purpose registers appear to occur in program order relative to other instructions.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>101</td>
<td>0100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL1</td>
<td>SPSR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ELR_EL1</td>
<td>ELR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL3</td>
<td>SPSR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ELR_EL3</td>
<td>ELR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0100</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>SP_EL2</td>
<td>SP_EL2</td>
</tr>
<tr>
<td>11</td>
<td>111</td>
<td>0100</td>
<td>0001</td>
<td>011</td>
<td>RW</td>
<td>RSP_EL0</td>
<td>RSP_EL0</td>
</tr>
</tbody>
</table>
C5.2 Special-purpose registers

This section describes the following Special-purpose registers:

- **ALLINT**, that holds the PSTATE.ALLINT bit.
- **CurrentEL**, that holds PSTATE.EL, and that software can read to determine the current Exception level.
- **DAIF**, that holds the current PSTATE.{D, A, I, F} interrupt mask bits.
- **DIT**, that holds the PSTATE.DIT bit.
- **ELR_EL1**, that holds the address to return to for an exception return from EL1.
- **ELR_EL2**, that holds the address to return to for an exception return from EL2.
- **ELR_EL3**, that holds the address to return to for an exception return from EL3.
- **FPCR**, that provides control of floating-point operation.
- **FPSR**, that provides floating-point status information.
- **NZCV**, that holds the PSTATE.{N, Z, C, V} condition flags.
- **PAN**, that holds the PSTATE.PAN state bit.
- **SP_EL0**, that holds the stack pointer for EL0.
- **SP_EL1**, that holds the stack pointer for EL1.
- **SP_EL2**, that holds the stack pointer for EL2.
- **SP_EL3**, that holds the stack pointer for EL3.
- **SPSel**, that holds PSTATE.SP, that at EL1 or higher selects the current SP.
- **SPSR_abt**, that holds process state on taking an exception to AArch32 Abort mode.
- **SPSR_EL1**, that holds process state on taking an exception to AArch64 EL1.
- **SPSR_EL2**, that holds process state on taking an exception to AArch64 EL2.
- **SPSR_EL3**, that holds process state on taking an exception to AArch64 EL3.
- **SPSR_fiq**, that holds process state on taking an exception to AArch32 FIQ mode.
- **SPSR_irq**, that holds process state on taking an exception to AArch32 IRQ mode.
- **SPSR_und**, that holds process state on taking an exception to AArch32 Undefined mode.
- **SSBS**, that holds the PSTATE.SSBS bit.
- **SVCR**, controls Streaming SVE mode and SME behavior.
- **TCO**, that holds the PSTATE.TCO bit.
- **UAO**, that holds the PSTATE.UAO bit.

The following registers are also Special-purpose registers:

- **DLR_EL0**, that holds the address to return to for a return from Debug state.
- **DSPSR_EL0**, that holds process state on entry to Debug state.
C5.2.1 ALLINT, All Interrupt Mask Bit

The ALLINT characteristics are:

Purpose

Allows access to the all interrupt mask bit.

Configurations

This register is present only when FEAT_NMI is implemented. Otherwise, direct accesses to ALLINT are UNDEFINED.

Attributes

ALLINT is a 64-bit register.

Field descriptions

![Field diagram]

Bits [63:14]

Reserved, RES0.

ALLINT, bit [13]

All interrupt mask. An interrupt is controlled by PSTATE.ALLINT when all of the following apply:

- SCTLR_ELx.NMI is 1.
- The interrupt is targeted at ELx.
- Execution is at ELx.

0b0 This control does not cause any interrupts to be masked.

0b1 If SCTLR_ELx.NMI is 1 and execution is at ELx, an IRQ or FIQ interrupt that is targeted to ELx, with or without Superpriority, is masked.

The value of this bit is set to the inverse value in the SCTLR_ELx.SPINTMASK field on taking an exception to ELx.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [12:0]

Reserved, RES0.

Accessing ALLINT

Accesses to this register use the following encodings in the System register encoding space:
### MRS \(<Xt>, ALLINT\)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b00</td>
<td>0b0100</td>
<td>0b0011</td>
<td>0b00</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsesif PSTATE.EL == EL1 then
X\([t, 64]\) = Zeros(50):PSTATE.ALLINT:Zeros(13);
elsesif PSTATE.EL == EL2 then
X\([t, 64]\) = Zeros(50):PSTATE.ALLINT:Zeros(13);
elsesif PSTATE.EL == EL3 then
X\([t, 64]\) = Zeros(50):PSTATE.ALLINT:Zeros(13);

### MSR ALLINT, \(<Xt>\)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b00</td>
<td>0b0100</td>
<td>0b0011</td>
<td>0b00</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsesif PSTATE.EL == EL1 then
  if EL2Enabled() && IsHCRXEL2Enabled() && HCRX_EL2.TALLINT == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    PSTATE.ALLINT = X\([t, 64]\)<13>;
elsesif PSTATE.EL == EL2 then
  PSTATE.ALLINT = X\([t, 64]\)<13>;
elsesif PSTATE.EL == EL3 then
  PSTATE.ALLINT = X\([t, 64]\)<13>;

### MSR ALLINT, \(#<imm>\)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>0b001</td>
<td>0b0100</td>
<td>0b000x</td>
<td>0b00</td>
</tr>
</tbody>
</table>

C5.2.2 CurrentEL, Current Exception Level

The CurrentEL characteristics are:

**Purpose**

Holds the current Exception level.

**Configurations**

There are no configuration notes.

**Attributes**

CurrentEL is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit positions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-4</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>31-2</td>
<td>Current Exception level.</td>
</tr>
<tr>
<td>1-0</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**EL, bits [3:2]**

Current Exception level.

- 0b00: EL0.
- 0b01: EL1.
- 0b10: EL2.
- 0b11: EL3.

When the HCR_EL2.NV bit is 1, EL1 read accesses to the CurrentEL register return the value of 0b10 in this field.

The reset behavior of this field is:

- On a Warm reset:
  - When the highest implemented Exception level is EL1, this field resets to 1.
  - When the highest implemented Exception level is EL2, this field resets to 2.
  - Otherwise, this field resets to 3.

**Accessing CurrentEL**

Accesses to this register use the following encodings in the System register encoding space:
MRS \(X_t\), CurrentEL

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        \(X[t, 64]\) = Zeros(60):'10':Zeros(2);
    else
        \(X[t, 64]\) = Zeros(60):PSTATE.EL:Zeros(2);
elsif PSTATE.EL == EL2 then
    \(X[t, 64]\) = Zeros(60):PSTATE.EL:Zeros(2);
elsif PSTATE.EL == EL3 then
    \(X[t, 64]\) = Zeros(60):PSTATE.EL:Zeros(2);
C5.2.3 DAIF, Interrupt Mask Bits

The DAIF characteristics are:

**Purpose**
Allows access to the interrupt mask bits.

**Configurations**
There are no configuration notes.

**Attributes**
DAIF is a 64-bit register.

**Field descriptions**

### Bits [63:10]
Reserved, RES0.

### D, bit [9]
Process state D mask.
- **0b0** Watchpoint, Breakpoint, and Software Step exceptions targeted at the current Exception level are not masked.
- **0b1** Watchpoint, Breakpoint, and Software Step exceptions targeted at the current Exception level are masked.

When the target Exception level of the debug exception is higher than the current Exception level, the exception is not masked by this bit.

The reset behavior of this field is:
- On a Warm reset, this field resets to 1.

### A, bit [8]
SError interrupt mask bit.
- **0b0** Exception not masked.
- **0b1** Exception masked.

The reset behavior of this field is:
- On a Warm reset, this field resets to 1.

### I, bit [7]
IRQ mask bit.
- **0b0** Exception not masked.
- **0b1** Exception masked.

The reset behavior of this field is:
- On a Warm reset, this field resets to 1.
F, bit [6]
FIQ mask bit.
0b0 Exception not masked.
0b1 Exception masked.
The reset behavior of this field is:
• On a Warm reset, this field resets to 1.

Bits [5:0]
Reserved, RES0.

Accessing DAIF
Accesses to this register use the following encodings in the System register encoding space:

*MRS <Xt>, DAIF*

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if (EL2Enabled() \&\& HCR_EL2.<E2H,TGE> == ’11’ || SCTLR_EL1.UMA == ’0’ then
    if EL2Enabled() \&\& HCR_EL2.TGE == ’1’ then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    X[t, 64] = Zeros(54):PSTATE.<D,A,I,F>:Zeros(6);
  elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(54):PSTATE.<D,A,I,F>:Zeros(6);
  elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(54):PSTATE.<D,A,I,F>:Zeros(6);
  elsif PSTATE.EL == EL3 then
    X[t, 64] = Zeros(54):PSTATE.<D,A,I,F>:Zeros(6);

*MSR DAIF, <Xt>*

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if (EL2Enabled() \&\& HCR_EL2.<E2H,TGE> == ’11’ || SCTLR_EL1.UMA == ’0’ then
    if EL2Enabled() \&\& HCR_EL2.TGE == ’1’ then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      PSTATE.<D,A,I,F> = X[t, 64]<9:6>;
  elsif PSTATE.EL == EL1 then
    PSTATE.<D,A,I,F> = X[t, 64]<9:6>;
  elsif PSTATE.EL == EL2 then
    PSTATE.<D,A,I,F> = X[t, 64]<9:6>;
  elsif PSTATE.EL == EL3 then
    PSTATE.<D,A,I,F> = X[t, 64]<9:6>;
### MSR DAIFSet, #<imm>:

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b110</td>
</tr>
</tbody>
</table>

### MSR DAIFClr, #<imm>:

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b111</td>
</tr>
</tbody>
</table>
C5.2.4 DIT, Data Independent Timing

The DIT characteristics are:

**Purpose**

Allows access to the Data Independent Timing bit.

**Configurations**

This register is present only when FEAT_DIT is implemented. Otherwise, direct accesses to DIT are **UNDEFINED**.

**Attributes**

DIT is a 64-bit register.

**Field descriptions**

![DIT Field Description Diagram]

**Bits [63:25]**

Reserved, RES0.

**DIT, bit [24]**

Data Independent Timing.

0b0 The architecture makes no statement about the timing properties of any instructions.

0b1 The architecture requires that:

- The timing of every load and store instruction is insensitive to the value of the data being loaded or stored.

- For certain data processing instructions, the instruction takes a time which is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- For certain data processing instructions, the response of the instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

The data processing instructions affected by this bit are:

- All cryptographic instructions. These instructions are:
  - AESD, AESE, AESIMC, AESMC, SHA1C, SHA1H, SHA1M, SHA1P, SHA1SU0, SHA1SU1, SHA256H, SHA256H2, SHA256SU0, SHA256SU1, SHA512H, SHA512H2, SHA512SU0, SHA512SU1, EOR3, RAX1, XAR, BCAX, SM3SS1, SM3TT1A, SM3TT1B, SM3TT2A, SM3TT2B, SM3PARTW1, SM3PARTW2, SME, and SM4EKEY.

- A subset of those instructions which use the general-purpose register file. These instructions are:
  - ADC, ADCS, ADD, ADDS, AND, ANDS, ASR, ASRV, BFC, BFI, BPM, BFXIL, BIC, BICS, COMN, CMP, CFINV, CINC, CINV, CLS, CLZ, CM, CNP, CMP, CNEG, CSEL, CSET, CSETM, CSINC, CSINV, CSNEG, EON, EOR, EXTR, LSL, LSLV, LSR, LSRV, MADD, MNEG, MOV, MOVK, MOVN, MOVZ, MSUB, MUL, MN, NEG, NEGS, NG, NGCS,
C5.2 Special-purpose registers

The A64 System Instruction Class

### Notes

The architecture makes no statement about the timing properties when the PSTATE.DIT bit is not set. However, it is likely that many of these instructions have timing that is invariant of the data in many situations.

In particular, Arm strongly recommends that the Armv8.3 pointer authentication instructions do not have their timing dependent on the key value used in the pointer authentication in all cases, regardless of the PSTATE.DIT bit.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**Bits [23:0]**

Reserved, RES0.

**Accessing DIT**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, DIT**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
X[t, 64] = Zeros(39):PSTATE.DIT:Zeros(24);
elsif PSTATE.EL == EL1 then
X[t, 64] = Zeros(39):PSTATE.DIT:Zeros(24);
elsif PSTATE.EL == EL2 then
X[t, 64] = Zeros(39):PSTATE.DIT:Zeros(24);
elsif PSTATE.EL == EL3 then
X[t, 64] = Zeros(39):PSTATE.DIT:Zeros(24);
### MSR DIT, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    PSTATE.DIT = X[t, 64]<24>
elsif PSTATE_EL == EL1 then
    PSTATE.DIT = X[t, 64]<24>
elsif PSTATE_EL == EL2 then
    PSTATE.DIT = X[t, 64]<24>
elsif PSTATE_EL == EL3 then
    PSTATE.DIT = X[t, 64]<24>

### MSR DIT, #<imm>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b010</td>
</tr>
</tbody>
</table>

```
C5.2.5 ELR_EL1, Exception Link Register (EL1)

The ELR_EL1 characteristics are:

**Purpose**
When taking an exception to EL1, holds the address to return to.

**Configurations**
There are no configuration notes.

**Attributes**
ELR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Return address</td>
</tr>
<tr>
<td></td>
<td>An exception return from EL1 using AArch64 makes ELR_EL1 become UNKNOWN.</td>
</tr>
<tr>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
</tbody>
</table>

**Accessing ELR_EL1**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic ELR_EL1 or ELR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ELR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b00</td>
<td>0b1000</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x230];
  else
    X[t, 64] = ELR_EL1;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = ELR_EL12;
  else
    X[t, 64] = ELR_EL1;
```
elsif PSTATE.EL == EL3 then
    X[t, 64] = ELR_EL1;

**MSR ELR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x230] = X[t, 64];
    else
        ELR_EL1 = X[t, 64];
    elsif PSTATE.EL == EL2 then
        if HCR_EL2.E2H == '1' then
            ELR_EL2 = X[t, 64];
        else
            UNDEFINED;
    elsif PSTATE.EL == EL3 then
        ELR_EL1 = X[t, 64];

**MRS <Xt>, ELR_EL12**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        X[t, 64] = NVMem[0x230];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        X[t, 64] = ELR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        X[t, 64] = ELR_EL1;
    else
        UNDEFINED;
### MSR ELR_EL12, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        MSR ELR_EL2, <Xt>
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        ELR_EL1 = X[t, 64];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        ELR_EL1 = X[t, 64];
    else
        UNDEFINED;
MRS <Xt>, ELR_EL2
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = ELR_EL1;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ELR_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = ELR_EL2;
```

### MSR ELR_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        ELR_EL1 = X[t, 64];
```

---

**C5-684**

*Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.*

*Non-Confidential*
elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    ELR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    ELR_EL2 = X[t, 64];
C5.2.6 ELR_EL2, Exception Link Register (EL2)

The ELR_EL2 characteristics are:

**Purpose**

When taking an exception to EL2, holds the address to return to.

**Configurations**

AArch64 System register ELR_EL2 bits [31:0] are architecturally mapped to AArch32 System register ELR_hyp[31:0].

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

ELR_EL2 is a 64-bit register.

**Field descriptions**

Bits [63:0]  
Return address.

An exception return from EL2 using AArch64 makes ELR_EL2 become UNKNOWN.

When EL2 is in AArch32 Execution state and an exception is taken from EL0, EL1, or EL2 to EL3 and AArch64 execution, the upper 32-bits of ELR_EL2 are either set to 0 or hold the same value that they did before AArch32 execution. Which option is adopted is determined by an implementation, and might vary dynamically within an implementation. Correspondingly software must regard the value as being an UNKNOWN choice between the two values.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing ELR_EL2**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic ELR_EL2 or ELR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsiF PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = ELR_EL1;
    elsif EL2Enabled() && HCR_EL2.<NV2,NV> == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ELR_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = ELR_EL2;

**MSR ELR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.<NV2,NV> == '11' then
        ELR_EL1 = X[t, 64];
    elsif EL2Enabled() & HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    ELR_EL2 = X[t, 64];
else
    if EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x230];
    else
        X[t, 64] = ELR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = ELR_EL1;
else
    if HCR_EL2.E2H == '1' then
        X[t, 64] = ELR_EL2;
    else
        X[t, 64] = ELR_EL1;
else
    if HCR_EL2.E2H == '0' then
        X[t, 64] = ELR_EL2;
else
    X[t, 64] = ELR_EL1;

**MRS <Xt>, ELR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x230];
    else
        X[t, 64] = ELR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        X[t, 64] = ELR_EL2;
else
    if HCR_EL2.E2H == '0' then
        X[t, 64] = ELR_EL2;
else
    X[t, 64] = ELR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = ELR_EL1;
### MSR ELR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>obo00</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NMMem[0x230] = X[t, 64];
    else
        ELR_EL1 = X[t, 64];
    end
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        ELR_EL2 = X[t, 64];
    else
        ELR_EL1 = X[t, 64];
    end
elsif PSTATE.EL == EL3 then
    ELR_EL1 = X[t, 64];
C5.2.7 ELR_EL3, Exception Link Register (EL3)

The ELR_EL3 characteristics are:

**Purpose**

When taking an exception to EL3, holds the address to return to.

**Configurations**

This register is present only when EL3 is implemented. Otherwise, direct accesses to ELR_EL3 are UNDEFINED.

**Attributes**

ELR_EL3 is a 64-bit register.

**Field descriptions**

```
Return address

Bits [63:0]
```

Return address.

An exception return from EL3 using AArch64 makes ELR_EL3 become UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing ELR_EL3**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ELR_EL3**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsi if PSTATE.EL == EL1 then
    UNDEFINED;
elsi if PSTATE.EL == EL2 then
    UNDEFINED;
elsi if PSTATE.EL == EL3 then
    X[t, 64] = ELR_EL3;
MSR ELR_EL3, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  ELR_EL3 = X[t, 64];
C5.2.8 FPCR, Floating-point Control Register

The FPCR characteristics are:

**Purpose**

Controls floating-point behavior.

**Configurations**

AArch64 System register FPCR bits [26:15] are architecturally mapped to AArch32 System register FPSCR[26:15].

AArch64 System register FPCR bits [12:8] are architecturally mapped to AArch32 System register FPSCR[12:8].

It is IMPLEMENTATION DEFINED whether the Len and Stride fields can be programmed to non-zero values, which will cause some AArch32 floating-point instruction encodings to be UNDEFINED, or whether these fields are RAZ.

**Attributes**

FPCR is a 64-bit register.

**Field descriptions**

![Field description diagram]

**Bits [63:27]**

Reserved, RES0.

**AHP, bit [26]**

Alternative half-precision control bit.

- 0b0 IEEE half-precision format selected.
- 0b1 Alternative half-precision format selected.

This bit is used only for conversions between half-precision floating-point and other floating-point formats.

The data-processing instructions added as part of the FEAT_FP16 extension always use the IEEE half-precision format, and ignore the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**DN, bit [25]**

Default NaN use for NaN propagation.

- 0b0 NaN operands propagate through to the output of a floating-point operation.
- 0b1 Any operation involving one or more NaNs returns the Default NaN.

This bit has no effect on the output of FABS, FMAX*, FMIN*, and FNEG instructions, and a default NaN is never returned as a result of these instructions.
The value of this bit controls both scalar and Advanced SIMD floating-point arithmetic.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**FZ, bit [24]**

Flushing denormalized numbers to zero control bit.

- **0b0**  If FPCR.AH is 0, the flushing to zero of single-precision and double-precision
denormalized inputs to, and outputs of, floating-point instructions not enabled by this
control, but other factors might cause the input denormalized numbers to be flushed to
zero.
  - If FPCR.AH is 1, the flushing to zero of single-precision and double-precision
denormalized outputs of floating-point instructions not enabled by this control, but
other factors might cause the input denormalized numbers to be flushed to zero.

- **0b1**  If FPCR.AH is 0, denormalized single-precision and double-precision inputs to, and
outputs from, floating-point instructions are flushed to zero.
  - If FPCR.AH is 1, denormalized single-precision and double-precision outputs from
floating-point instructions are flushed to zero.

For more information, see Flushing denormalized numbers to zero on page A1-58 and the
pseudocode of the floating-point instructions.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**RMode, bits [23:22]**

Rounding Mode control field.

- **0b00**  Round to Nearest (RN) mode.
- **0b01**  Round towards Plus Infinity (RP) mode.
- **0b10**  Round towards Minus Infinity (RM) mode.
- **0b11**  Round towards Zero (RZ) mode.

The specified rounding mode is used by both scalar and Advanced SIMD floating-point
instructions.

If FPCR.AH is 1, then the following instructions use Round to Nearest mode regardless of the value
of this bit:
- The FRECPE, FRECPS, FRECPIX, FRSQRT, and FRSQRTS instructions.
- The BFCT, BFCTN, BFCTN2, BFCTTN, BFCTTL, and BFCTALT instructions.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Stride, bits [21:20]**

This field has no function in AArch64 state, and non-zero values are ignored during execution in
AArch64 state.

This field is included only for context saving and restoration of the AArch32 FPSCR.Stride field.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**FZ16, bit [19]**

When FEAT_FP16 is implemented:

Flushing denormalized numbers to zero control bit on half-precision data-processing instructions.

- **0b0**  For some instructions, this bit disables flushing to zero of inputs and outputs that are
half-precision denormalized numbers.
- **0b1**  Flushing denormalized numbers to zero enabled.
For some instructions that do not convert a half-precision input to a higher precision output, this bit enables flushing to zero of inputs and outputs that are half-precision denormalized numbers.

The value of this bit applies to both scalar and Advanced SIMD floating-point half-precision calculations.

For more information, see Flushing denormalized numbers to zero on page A1-58 and the pseudocode of the floating-point instructions.

The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
  Reserved, RES0.

Len, bits [18:16]

This field has no function in AArch64 state, and non-zero values are ignored during execution in AArch64 state.

This field is included only for context saving and restoration of the AArch32 FPSCR.Len field.

The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

IDE, bit [15]

Input Denormal floating-point exception trap enable.

0b0 Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.IDC bit is set to 1.

0b1 Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IDC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level, the value of FPCR.IDE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The value of this bit controls both scalar and vector floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [14]

Reserved, RES0.

EBF, bit [13]

When **FEAT_EBF16 is implemented:**

The value of this bit controls the numeric behaviors of BFloat16 dot product calculations performed by the BFDOT, BFMMMLA, BFMOPA, and BFMOPS instructions.

When ID_AA64ISAR1_EL1.BF16 and ID_AA64ZFR0_EL1.BF16 are 0b0010, the PE supports the FPCR.EBF field. Otherwise, FPCR.EBF is RES0.

0b0 These instructions use the standard BFloat16 behaviors:
        • Ignoring the FPCR.RMode control and using the rounding mode defined for BFloat16. For more information, see Rounding on page E1-7146.
        • Flushing denormalized inputs and outputs to zero, as if the FPCR.FZ and FPCR.FIZ controls had the value '1'.
        • Performing unfused multiplies and additions with intermediate rounding of all products and sums.
These instructions use the extended BFloat16 behaviors:

- Supporting all four IEEE 754 rounding modes selected by the FPCR.RMode control.
- Optionally, flushing denormalized inputs and outputs to zero, as governed by the FPCR.FZ and FPCR.FIZ controls.
- Performing a fused two-way sum-of-products for each pair of adjacent BFloat16 elements, without intermediate rounding of the products, but rounding the single-precision sum before addition to the accumulator.
- Generating the default NaN as intermediate sum-of-products when any multiplier input is a NaN, or any product is infinity × 0.0, or there are infinite products with differing signs.
- Generating an intermediate sum-of-products of the same infinity when there are infinite products all with the same sign.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**IXE, bit [12]**

Inexact floating-point exception trap enable.

- 0b0 Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.IXC bit is set to 1.
- 0b1 Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IXC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level, the value of FPCR.IXE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The value of this bit controls both scalar and vector floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**UFE, bit [11]**

Underflow floating-point exception trap enable.

- 0b0 Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.UFC bit is set to 1.
- 0b1 Trapped exception handling selected. If the floating-point exception occurs and Flush-to-zero is not enabled, the PE does not update the FPSR.UFC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level, the value of FPCR.UFE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The value of this bit controls both scalar and vector floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**OFE, bit [10]**

Overflow floating-point exception trap enable.

- 0b0 Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.OFC bit is set to 1.
DZE, bit [9]
Divide by Zero floating-point exception trap enable.

0b0  Untrapped exception handling selected. If the floating-point exception occurs, the FPCR.DZE bit is set to 1.

0b1  Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPCR.DZE bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level, the value of FPCR.DZE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The value of this bit controls both scalar and vector floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

IOE, bit [8]
Invalid Operation floating-point exception trap enable.

0b0  Untrapped exception handling selected. If the floating-point exception occurs, the FPCR.IOE bit is set to 1.

0b1  Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPCR.IOE bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level, the value of FPCR.IOE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The value of this bit controls both scalar and vector floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [7:3]
Reserved, RES0.

NEP, bit [2]

When FEAT_AFP is implemented:

Controls how the output elements other than the lowest element of the vector are determined for Advanced SIMD scalar instructions.

0b0  Does not affect how the output elements other than the lowest are determined for Advanced SIMD scalar instructions.

0b1  The output elements other than the lowest are taken from the following registers:

• For 3-input scalar versions of the FMLA (by element) and FMLS (by element) instructions, the <Hd>, <Sd>, or <Dd> register.
• For 3-input versions of the FMADD, FMSUB, FNADD, and FNNSUB instructions, the \(<Ha>\), \(<Sa>\), or \(<Da>\) register.

• For 2-input scalar versions of the FACGE, FACGT, FCMEQ (register), FCMGE (register), and FCMGT (register) instructions, the \(<Hm>\), \(<Sm>\), or \(<Dm>\) register.

• For 2-input scalar versions of the FABD, FADD (scalar), FDIV (scalar), FMAX (scalar), FMAXNM (scalar), FMIN (scalar), FMINNM (scalar), FMUL (by element), FMULX (scalar), FMULX (by element), FNMUL (scalar), FRECPS, FRSQRTS, and FSUB (scalar) instructions, the \(<Hn>\), \(<Sn>\), or \(<Dn>\) register.

• For 1-input scalar versions of the following instructions, the \(<Hd>\), \(<Sd>\), or \(<Dd>\) register:
  — The (vector) versions of the FCVTAS, FCVTASU, FCVTMS, FCVTMSU, FCVTNS, FCVTNU, FCVTPS, and FCVTTPU instructions.
  — The (vector, fixed-point) and (vector, integer) versions of the FCVTZS, FCVTZSU, SCVT, and UCVTF instructions.
  — The (scalar) versions of the FABS, FNEG, FRINT32X, FRINT32Z, FRINT64X, FRINT64Z, FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX, FRINTZ, and FSQRT instructions.
  — The (scalar, fixed-point) and (scalar, integer) versions of the SCVT, and UCVTF instructions.
  — The BFCTV, FCVT, FCVTXN, FRECPE, FRECPIX, and FRSQRTE instructions.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level, the value of FPCR.NEP is treated as 0 for all purposes other than a direct read or write of the FPCR.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**AH, bit [1]**

**When FEAT_AFP is implemented:**

Alternate Handling. Controls alternate handling of floating-point numbers.

The Arm architecture supports two models for handling some of the corner cases of the floating-point behaviors, such as the nature of flushing of denormalized numbers, the detection of tininess and other exceptions and a range of other behaviors. The value of the FPCR.AH bit selects between these models.

For more information on the FPCR.AH bit, see *Flushing denormalized numbers to zero* on page A1-58, *Floating-point exceptions and exception traps* on page A1-64 and the pseudocode of the floating-point instructions.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.
**FIZ, bit [0]**

*When FEAT_AFP is implemented:*

Flush Inputs to Zero. Controls whether single-precision, double-precision and BFloat16 input operands that are denormalized numbers are flushed to zero.

- **0b0** The flushing to zero of single-precision and double-precision denormalized inputs to floating-point instructions not enabled by this control, but other factors might cause the input denormalized numbers to be flushed to zero.

- **0b1** Denormalized single-precision and double-precision inputs to most floating-point instructions flushed to zero.

For more information, see *Flushing denormalized numbers to zero* on page A1-58 and the pseudocode of the floating-point instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**

Reserved, RES0.

### Accessing FPCR

Accesses to this register use the following encodings in the System register encoding space:

#### MRS <Xt>, FPCR

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elsif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.FPEN != '11' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x00);
    else
      AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CPACR_EL1.FPEN != '11' then
      if EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
      else
        AArch64.SystemAccessTrap(EL2, 0x07);
      elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
      elsif EL2Enabled() && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
      elsif EL2Enabled() && HCR_EL2.E2H != '1' && CPACR_EL1.FPEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x07);
      elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if Halted() && EDSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL3, 0x07);
        else
          X[t, 64] = FPCR;
        elsif PSTATE.EL == EL1 then
          if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
            UNDEFINED;
          elsif CPACR_EL1.FPEN == '0' then
            AArch64.SystemAccessTrap(EL1, 0x07);
          elsif EL2Enabled() && HCR_EL2.TGE == '1' && CPTR_EL2.TFP == '1' then
            AArch64.SystemAccessTrap(EL2, 0x07);
          elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
            AArch64.SystemAccessTrap(EL2, 0x07);
```
elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x07);
  end;
else
  X[t, 64] = FPCR;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATIONDEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elsif HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x07);
  elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x07);
  elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x07);
    end;
else
  AArch64.SystemAccessTrap(EL3, 0x07);
else
  X[t, 64] = FPCR;
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TFP == '1' then
    AArch64.SystemAccessTrap(EL3, 0x07);
  else
    X[t, 64] = FPCR;
  end;
if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATIONDEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elsif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.FPEN != '11' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x00);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CPACR_EL1.FPEN != '11' then
      AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
      AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
      AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x07);
      end;
    else
      AArch64.SystemAccessTrap(EL3, 0x07);
    end;
else
  FPCR = X[t, 64];
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATIONDEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elsif CPACR_EL1.FPEN == 'x0' then
    AArch64.SystemAccessTrap(EL1, 0x07);
  elsif EL2Enabled() && HCR_EL2.E2H != '1' && CPTR_EL2.FPEN == '1' then
    AArch64.SystemAccessTrap(EL2, 0x07);
  else
    FPCR = X[t, 64];
end;

**MSR FPCR, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATIONDEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elsif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.FPEN != '11' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x00);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CPACR_EL1.FPEN != '11' then
      AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
      AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
      AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x07);
      end;
    else
      AArch64.SystemAccessTrap(EL3, 0x07);
    end;
else
  FPCR = X[t, 64];
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATIONDEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elsif CPACR_EL1.FPEN == 'x0' then
    AArch64.SystemAccessTrap(EL1, 0x07);
  elsif EL2Enabled() && HCR_EL2.E2H != '1' && CPTR_EL2.FPEN == '1' then
    AArch64.SystemAccessTrap(EL2, 0x07);
elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x07);
elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x07);
    end if
else
    FPCR = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
        end if
    else
        FPCR = X[t, 64];
    end if
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPCR = X[t, 64];
C5.2.9 FPSR, Floating-point Status Register

The FPSR characteristics are:

**Purpose**

Provides floating-point system status information.

**Configurations**

AArch64 System register FPSR bits [31:27] are architecturally mapped to AArch32 System register FPSCR[31:27].

AArch64 System register FPSR bit [7] is architecturally mapped to AArch32 System register FPSCR[7].

AArch64 System register FPSR bits [4:0] are architecturally mapped to AArch32 System register FPSCR[4:0].

**Attributes**

FPSR is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>31</td>
<td>N, bit [31]</td>
</tr>
<tr>
<td>30</td>
<td>Z, bit [30]</td>
</tr>
<tr>
<td>29-28</td>
<td>V, bit [28]</td>
</tr>
<tr>
<td>27</td>
<td>QC, bit [27]</td>
</tr>
<tr>
<td>26-8</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>7</td>
<td>PSTATE.N</td>
</tr>
<tr>
<td>6-0</td>
<td>PSTATE.Z</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**N, bit [31]**

*When AArch32 is supported and AArch32 floating-point is implemented:*

Negative condition flag for AArch32 floating-point comparison operations.

--- **Note** ---

AArch64 floating-point comparisons set the PSTATE.N flag instead.

---

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Z, bit [30]**

*When AArch32 is supported and AArch32 floating-point is implemented:*

Zero condition flag for AArch32 floating-point comparison operations.

--- **Note** ---

AArch64 floating-point comparisons set the PSTATE.Z flag instead.

---

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
C, bit [29]

When AArch32 is supported and AArch32 floating-point is implemented:

Carry condition flag for AArch32 floating-point comparison operations.

--- Note ---

AArch64 floating-point comparisons set the PSTATE.C flag instead.

---

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

V, bit [28]

When AArch32 is supported and AArch32 floating-point is implemented:

Overflow condition flag for AArch32 floating-point comparison operations.

--- Note ---

AArch64 floating-point comparisons set the PSTATE.V flag instead.

---

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

QC, bit [27]

Cumulative saturation bit, Advanced SIMD only. This bit is set to 1 to indicate that an Advanced SIMD integer operation has saturated since 0 was last written to this bit.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [26:8]

Reserved, RES0.

IDC, bit [7]

Input Denormal cumulative floating-point exception bit. This bit is set to 1 to indicate that the Input Denormal floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IDE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.IDE is 0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXC, bit [4]

Inexact cumulative floating-point exception bit. This bit is set to 1 to indicate that the Inexact floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IXE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.IXE is 0.
The criteria for the Inexact floating-point exception to occur are affected by whether denormalized numbers are flushed to zero and by the value of the FPCR.AH bit. For more information, see Floating-point exceptions and exception traps on page A1-64.

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally UNKNOWN value.

**UFC, bit [3]**

Underflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Underflow floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.UFE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.UFE is 0 or if flushing denormalized numbers to zero is enabled.

The criteria for the Underflow floating-point exception to occur are affected by whether denormalized numbers are flushed to zero and by the value of the FPCR.AH bit. For more information, see Floating-point exceptions and exception traps on page A1-64.

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally UNKNOWN value.

**OFC, bit [2]**

Overflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Overflow floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.OFE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.OFE is 0.

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally UNKNOWN value.

**DZC, bit [1]**

Divide by Zero cumulative floating-point exception bit. This bit is set to 1 to indicate that the Divide by Zero floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.DZE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.DZE is 0.

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IOC, bit [0]**

Invalid Operation cumulative floating-point exception bit. This bit is set to 1 to indicate that the Invalid Operation floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IOE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.IOE is 0.

The criteria for the Invalid Operation floating-point exception to occur are affected by the value of the FPCR.AH bit. For more information, see Floating-point exceptions and exception traps on page A1-64.

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing FPSR**

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, FPSR

```
if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '1'1') && CPACR_EL1.FPEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x00);
    else
        AArch64.SystemAccessTrap(EL1, 0x07);
    endif
else
    if EL2Enabled() && HCR_EL2.<E2H,TGE> == '1'1' && CPACR_EL1.FPEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x00);
    else
        AArch64.SystemAccessTrap(EL1, 0x07);
    endif
endif
else
    X[t, 64] = FPSR;
else
    PSTATE.EL == EL1 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
            UNDEFINED;
        elsif HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
            AArch64.SystemAccessTrap(EL2, 0x07);
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
        endif
    else
        PSTATE.EL == EL2 then
            if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
                UNDEFINED;
            elsif HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
                AArch64.SystemAccessTrap(EL2, 0x07);
            else
                AArch64.SystemAccessTrap(EL3, 0x07);
            endif
        else
            if Halted() && CPTR_EL3.TFP == '1' then
                AArch64.SystemAccessTrap(EL3, 0x07);
            else
                X[t, 64] = FPSR;
            endif
        else
            PSTATE.EL == EL3 then
                if CPTR_EL3.TFP == '1' then
                    AArch64.SystemAccessTrap(EL3, 0x07);
                else
                    X[t, 64] = FPSR;
                endif
```

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

The A64 System Instruction Class
C5.2 Special-purpose registers

ARM DDI 04871.a
Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential
else
    X[t, 64] = FPSR;

### MSR FPSR, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.FPEN != '11' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x00);
        else
            AArch64.SystemAccessTrap(EL1, 0x07);
        end
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.FPEN == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    else
        AArch64.SystemAccessTrap(EL3, 0x07);
    end
else
    FPSR = X[t, 64];
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif CPACR_EL1.FPEN == '0' then
        AArch64.SystemAccessTrap(EL1, 0x07);
    else
        AArch64.SystemAccessTrap(EL1, 0x07);
    end
else
    FPR = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    else
        AArch64.SystemAccessTrap(EL2, 0x07);
    end
else
    AArch64.SystemAccessTrap(EL3, 0x07);
else
    FPSR = X[t, 64];
elsif PSTATE.EL == EL3 then
if CPTR_EL3.TFP == '1' then
    AArch64.SystemAccessTrap(EL3, 0x07);
else
    FPSR = X[t, 64];
C5.2.10 NZCV, Condition Flags

The NZCV characteristics are:

**Purpose**

Allows access to the condition flags.

**Configurations**

There are no configuration notes.

**Attributes**

NZCV is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:32]</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>
| [31] | N, bit
Negative condition flag. Set to 1 if the result of the last flag-setting instruction was negative. |
| [30] | Z, bit
Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison. |
| [29] | C, bit
Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition. |
| [28] | V, bit
Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition. |
| [27:0] | Reserved, RES0. |

**Accessing NZCV**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS} \ <Xt>, \ NZCV
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
\[X[t, 64] = \text{zeros}(32):\text{PSTATE.}<N,Z,C,V>:\text{zeros}(28);\]
elsif PSTATE.EL == EL1 then
  X[t, 64] = Zeros(32) : PSTATE.<N,Z,C,V> : Zeros(28);
elsif PSTATE.EL == EL2 then
  X[t, 64] = Zeros(32) : PSTATE.<N,Z,C,V> : Zeros(28);
elsif PSTATE.EL == EL3 then
  X[t, 64] = Zeros(32) : PSTATE.<N,Z,C,V> : Zeros(28);

**MSR NZCV, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  PSTATE.<N,Z,C,V> = X[t, 64]<31:28>;
elsif PSTATE.EL == EL1 then
  PSTATE.<N,Z,C,V> = X[t, 64]<31:28>;
elsif PSTATE.EL == EL2 then
  PSTATE.<N,Z,C,V> = X[t, 64]<31:28>;
elsif PSTATE.EL == EL3 then
  PSTATE.<N,Z,C,V> = X[t, 64]<31:28>;

else if PSTATE.EL == EL0 then
  PSTATE.<N,Z,C,V> = X[t, 64]<31:28>;
elsif PSTATE.EL == EL1 then
  PSTATE.<N,Z,C,V> = X[t, 64]<31:28>;
elsif PSTATE.EL == EL2 then
  PSTATE.<N,Z,C,V> = X[t, 64]<31:28>;
elsif PSTATE.EL == EL3 then
  PSTATE.<N,Z,C,V> = X[t, 64]<31:28>;

C5.2.11 PAN, Privileged Access Never

The PAN characteristics are:

**Purpose**

Allows access to the Privileged Access Never bit.

**Configurations**

This register is present only when FEAT_PAN is implemented. Otherwise, direct accesses to PAN are UNDEFINED.

**Attributes**

PAN is a 64-bit register.

**Field descriptions**

![Field descriptions diagram]

**Bits [63:23]**

Reserved, RES0.

**PAN, bit [22]**

Privileged Access Never.

- **0b0** Privileged reads and write are not disabled by this mechanism.
- **0b1** Disables privileged read and write accesses to addresses accessible at EL0 for an enabled stage 1 translation regime that defines the EL0 permissions.

The value of this bit is usually preserved on taking an exception, except in the following situations:

- When the target of the exception is EL1, and the value of the SCTLR_EL1.SPAN bit is 0, this bit is set to 1.
- When the target of the exception is EL2, HCR_EL2.{E2H, TGE} is \{1, 1\}, and the value of the SCTLR_EL2.SPAN bit is 0, this bit is set to 1.

**Bits [21:0]**

Reserved, RES0.

**Accessing PAN**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, PAN**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsi if PSTATE.EL == EL1 then
The A64 System Instruction Class
C5.2 Special-purpose registers

X[t, 64] = Zeros(41):PSTATE.PAN:Zeros(22);
elsif PSTATE.EL == EL2 then
X[t, 64] = Zeros(41):PSTATE.PAN:Zeros(22);
elsif PSTATE.EL == EL3 then
X[t, 64] = Zeros(41):PSTATE.PAN:Zeros(22);

**MSR PAN, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
PSTATE.PAN = X[t, 64]<22>;
elsif PSTATE.EL == EL2 then
PSTATE.PAN = X[t, 64]<22>;
elsif PSTATE.EL == EL3 then
PSTATE.PAN = X[t, 64]<22>;

**MSR PAN, #<imm>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>0b000</td>
<td>0b0100</td>
<td>0b100</td>
</tr>
</tbody>
</table>
C5.2.12   SP_EL0, Stack Pointer (EL0)

The SP_EL0 characteristics are:

**Purpose**

Holds the stack pointer associated with EL0. At higher Exception levels, this is used as the current stack pointer when the value of SPSel.SP is 0.

**Configurations**

There are no configuration notes.

**Attributes**

SP_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Stack pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>63 32</td>
<td></td>
</tr>
<tr>
<td>31 0</td>
<td></td>
</tr>
</tbody>
</table>

**Accessing SP_EL0**

When the value of PSTATE.SP is 0, this register is accessible at all Exception levels as the current stack pointer. Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, SP_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0100</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if PSTATE.SP == '0' then
    UNDEFINED;
  else
    X[t, 64] = SP_EL0;
elsif PSTATE.EL == EL2 then
  if PSTATE.SP == '0' then
    UNDEFINED;
  else
    X[t, 64] = SP_EL0;
elsif PSTATE.EL == EL3 then
  if PSTATE.SP == '0' then
    UNDEFINED;
else
  X[t, 64] = SP_EL0;
```
else
    X[t, 64] = SP_EL0;

**MSR SP_EL0, <Xt>**

<table>
<thead>
<tr>
<th></th>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0100</td>
<td>0b0001</td>
<td>0b000</td>
<td></td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        SP_EL0 = X[t, 64];
    endif
else PSTATE.EL == EL2 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        SP_EL0 = X[t, 64];
    endif
elseif PSTATE.EL == EL3 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        SP_EL0 = X[t, 64];
    endif
else PSTATE.EL == EL4 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        SP_EL0 = X[t, 64];
    endif
else PSTATE.EL == EL5 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        SP_EL0 = X[t, 64];
    endif
else PSTATE.EL == EL6 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        SP_EL0 = X[t, 64];
    endif
else PSTATE.EL == EL7 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        SP_EL0 = X[t, 64];
    endif
else
    UNDEFINED;
endif
C5.2.13 SP_EL1, Stack Pointer (EL1)

The SP_EL1 characteristics are:

**Purpose**

Holds the stack pointer associated with EL1. When executing at EL1, the value of SPSel.SP determines the current stack pointer:

<table>
<thead>
<tr>
<th>SPSel.SP</th>
<th>Current stack pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>SP_EL0</td>
</tr>
<tr>
<td>0b1</td>
<td>SP_EL1</td>
</tr>
</tbody>
</table>

**Configurations**

There are no configuration notes.

**Attributes**

SP_EL1 is a 64-bit register.

**Field descriptions**

Bits [63:0]

Stack pointer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing SP_EL1**

This accessibility information only applies to accesses using the MRS or MSR instructions.

When the value of SPSel.SP is 1, this register is also accessible at EL1 as the current stack pointer.

Note

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

Accesses to this register use the following encodings in the System register encoding space:
MRS \langle Xt \rangle, SP_EL1

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b100 & 0b0100 & 0b0001 & 0b000
\end{array}
\]

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() \&\& HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = NVMem[0x240];
    elsif EL2Enabled() \&\& HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE_EL == EL2 then
        X[t, 64] = SP_EL1;
    elsif PSTATE_EL == EL3 then
        X[t, 64] = SP_EL1;

MSR SP_EL1, \langle Xt \rangle

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b100 & 0b0100 & 0b0001 & 0b000
\end{array}
\]

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() \&\& HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x240] = X[t, 64];
    elsif EL2Enabled() \&\& HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE_EL == EL2 then
        SP_EL1 = X[t, 64];
    elsif PSTATE_EL == EL3 then
        SP_EL1 = X[t, 64];
C5.2.14   SP_EL2, Stack Pointer (EL2)

The SP_EL2 characteristics are:

**Purpose**

Holds the stack pointer associated with EL2. When executing at EL2, the value of SPSel.SP determines the current stack pointer:

<table>
<thead>
<tr>
<th>SPSel.SP</th>
<th>Current stack pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>SP_EL0</td>
</tr>
<tr>
<td>0b1</td>
<td>SP_EL2</td>
</tr>
</tbody>
</table>

**Configurations**

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

SP_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Stack pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:0]**

Stack pointer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing SP_EL2**

This accessibility information only applies to accesses using the MRS or MSR instructions.

When the value of SPSel.SP is 1, this register is also accessible at EL2 as the current stack pointer.

**Note**

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

Accesses to this register use the following encodings in the System register encoding space:
**MRS \(<X_t>, SP_{EL2}\)**

```
MRS <X_t>, SP_EL2

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0100</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>
```

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  X[t, 64] = SP_EL2;

**MSR SP_EL2, <X*>**

```
MSR SP_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0100</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>
```

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  SP_EL2 = X[t, 64];
C5.2.15 SP_EL3, Stack Pointer (EL3)

The SP_EL3 characteristics are:

**Purpose**

Holds the stack pointer associated with EL3. When executing at EL3, the value of SPSel.SP determines the current stack pointer:

<table>
<thead>
<tr>
<th>SPSel.SP</th>
<th>Current stack pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>SP_EL0</td>
</tr>
<tr>
<td>0b1</td>
<td>SP_EL3</td>
</tr>
</tbody>
</table>

**Configurations**

This register is present only when EL3 is implemented. Otherwise, direct accesses to SP_EL3 are UNDEFINED.

**Attributes**

SP_EL3 is a 64-bit register.

**Field descriptions**

Bits [63:0]  
Stack pointer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
C5.2.16 SPSel, Stack Pointer Select

The SPSel characteristics are:

**Purpose**
Allows the Stack Pointer to be selected between SP_EL0 and SP_ELx.

**Configurations**
There are no configuration notes.

**Attributes**
SPSel is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:1]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>0</td>
<td>Stack pointer to use. Possible values of this bit are:</td>
</tr>
<tr>
<td>0</td>
<td>Use SP_EL0 at all Exception levels.</td>
</tr>
<tr>
<td>1</td>
<td>Use SP_ELx for Exception level ELx.</td>
</tr>
</tbody>
</table>

When FEAT_NMI is implemented and SCTLR_ELx.SPINTMASK is 1, if execution is at ELx, an IRQ or FIQ interrupt that is targeted to ELx is masked regardless of any denotation of Superpriority.

The reset behavior of this field is:
- On a Warm reset, this field resets to 1.

**Accessing SPSel**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, SPSel**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(63):PSTATE.SP;
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(63):PSTATE.SP;
elsif PSTATE.EL == EL3 then
    X[t, 64] = Zeros(63):PSTATE.SP;
**MSR SPSel, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  PSTATE.SP = X[t, 64]<0>;
elsif PSTATE_EL == EL2 then
  PSTATE.SP = X[t, 64]<0>;
elsif PSTATE_EL == EL3 then
  PSTATE.SP = X[t, 64]<0>;

**MSR SPSel, #<imm>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>0b000</td>
<td>0b0100</td>
<td>0b101</td>
</tr>
</tbody>
</table>

C5.2.17  SPSR_abt, Saved Program Status Register (Abort mode)

The SPSR_abt characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to Abort mode.

**Configurations**

AArch64 System register SPSR_abt bits [31:0] are architecturally mapped to AArch32 System register SPSR_abt[31:0].

If EL1 only supports execution in AArch64 state, this register is RES0 from EL2 and EL3.

**Attributes**

SPSR_abt is a 64-bit register.

**Field descriptions**

*When EL1 can only use AArch64:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>Reserved, RES0</td>
</tr>
</tbody>
</table>

*Otherwise:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>Reserved, RES0</td>
</tr>
</tbody>
</table>
| 31:0 | N, Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Abort mode, and copied to PSTATE.N on executing an illegal exception return operation in Abort mode. The reset behavior of this field is:
|      | • On a Warm reset, this field resets to an architecturally UNKNOWN value. |
| 30   | Z, Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Abort mode, and copied to PSTATE.Z on executing an illegal exception return operation in Abort mode. The reset behavior of this field is:
|      | • On a Warm reset, this field resets to an architecturally UNKNOWN value. |
C, bit [29]
Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Abort mode, and
copied to PSTATE.C on executing an illegal exception return operation in Abort mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]
Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Abort mode, and
copied to PSTATE.V on executing an illegal exception return operation in Abort mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]
Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Abort mode,
and copied to PSTATE.Q on executing an illegal exception return operation in Abort mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]
If-Then. Set to the value of PSTATE.IT on taking an exception to Abort mode, and copied to
PSTATE.IT on executing an illegal exception return operation in Abort mode.

SPSR_abt.IT must contain a value that is valid for the instruction being returned to.

The IT field is split as follows:
• IT[1:0] is SPSR_abt[26:25].
• IT[7:2] is SPSR_abt[15:10].

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]
RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.
Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction
set state.

SSBS, bit [23]
When FEAT_SSBS is implemented:
Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Abort mode, and
copied to PSTATE.SSBS on executing an illegal exception return operation in Abort mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

PAN, bit [22]
When FEAT_PAN is implemented:
Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Abort mode, and
copied to PSTATE.PAN on executing an illegal exception return operation in Abort mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Otherwise:

Reserved, RES0.

DIT, bit [21]

**When FEAT_DIT is implemented:**

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Abort mode, and copied to PSTATE.DIT on executing an illegal exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Abort mode, and copied to PSTATE.IL on executing an illegal exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Abort mode, and copied to PSTATE.GE on executing an illegal exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Abort mode, and copied to PSTATE.E on executing an illegal exception return operation in Abort mode.

If the implementation does not support big-endian operation, SPSR_abt.E is RES0. If the implementation does not support little-endian operation, SPSR_abt.E is RES1. On executing an illegal exception return operation in Abort mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_abt.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_abt.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to Abort mode, and copied to PSTATE.A on executing an illegal exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Abort mode, and copied to PSTATE.I on executing an illegal exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Abort mode, and copied to PSTATE.F on executing an illegal exception return operation in Abort mode.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Abort mode, and copied to PSTATE.T on executing an illegal exception return operation in Abort mode.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Abort mode, and copied to PSTATE.M[4:0] on executing an illegal exception return operation in Abort mode.
0b10000 User.
0b10001 FIQ.
0b10010 IRQ.
0b10011 Supervisor.
0b10111 Abort.
0b11011 Undefined.
0b11111 System.

Other values are reserved. If SPSR_abt.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an illegal exception return operation in Abort mode is an illegal return event, as described in *Illegal return events from AArch32 state* on page G1-8952.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_abt

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, SPSR_abt**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0100</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elseif PSTATE.EL == EL2 then
    X[t, 64] = SPSR_abt;
elseif PSTATE.EL == EL3 then
    X[t, 64] = SPSR_abt;
### MSR SPSR_abt, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0100</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    endif
elsif PSTATE.EL == EL2 then
    SPSR_abt = X[t, 64];
elsif PSTATE.EL == EL3 then
    SPSR_abt = X[t, 64];
C5.2.18 SPSR_EL1, Saved Program Status Register (EL1)

The SPSR_EL1 characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to EL1.

**Configurations**

AArch64 System register SPSR_EL1 bits [31:0] are architecturally mapped to AArch32 System register SPSR_svc[31:0].

**Attributes**

SPSR_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported and exception taken from AArch32 state:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL1, and copied to PSTATE.N on executing an exception return operation in EL1. The reset behavior of this field is:</td>
</tr>
<tr>
<td>Z</td>
<td>Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL1, and copied to PSTATE.Z on executing an exception return operation in EL1. The reset behavior of this field is:</td>
</tr>
<tr>
<td>V</td>
<td>Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL1, and copied to PSTATE.V on executing an exception return operation in EL1. The reset behavior of this field is:</td>
</tr>
<tr>
<td>C</td>
<td>Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL1, and copied to PSTATE.C on executing an exception return operation in EL1. The reset behavior of this field is:</td>
</tr>
</tbody>
</table>

An exception return from EL1 using AArch64 makes SPSR_EL1 become UNKNOWN.

**Bits [63:32]**

Reserved, RES0.

**N, bit [31]**

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL1, and copied to PSTATE.N on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Z, bit [30]**

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL1, and copied to PSTATE.Z on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**C, bit [29]**

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL1, and copied to PSTATE.C on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**V, bit [28]**

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL1, and copied to PSTATE.V on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Q, bit [27]
Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to EL1, and copied to PSTATE.Q on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]
If-Then. Set to the value of PSTATE.IT on taking an exception to EL1, and copied to PSTATE.IT on executing an exception return operation in EL1.

SPSR_EL1.IT must contain a value that is valid for the instruction being returned to.

The IT field is split as follows:
• IT[1:0] is SPSR_EL1[26:25].
• IT[7:2] is SPSR_EL1[15:10].

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

DIT, bit [24]
When FEAT_DIT is implemented:
Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL1, and copied to PSTATE.DIT on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

SSBS, bit [23]
When FEAT_SSBS is implemented:
Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL1, and copied to PSTATE.SSBS on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

PAN, bit [22]
When FEAT_PAN is implemented:
Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL1, and copied to PSTATE.PAN on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

SS, bit [21]
Software Step. Set to the value of PSTATE.SS on taking an exception to EL1, and conditionally copied to PSTATE.SS on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL1, and copied to
PSTATE.IL on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to EL1, and
copied to PSTATE.GE on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to EL1, and copied to PSTATE.E
on executing an exception return operation in EL1.

If the implementation does not support big-endian operation, SPSR_EL1.E is RES0. If the
implementation does not support little-endian operation, SPSR_EL1.E is RES1. On executing an
exception return operation in EL1, if the implementation does not support big-endian operation at
the Exception level being returned to, SPSR_EL1.E is RES0, and if the implementation does not
support little-endian operation at the Exception level being returned to, SPSR_EL1.E is RES1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL1, and copied to
PSTATE.A on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL1, and copied to
PSTATE.I on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL1, and copied to
PSTATE.F on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to EL1, and copied to
PSTATE.T on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4], bit [4]

Execution state. Set to 0b1, the value of PSTATE.nRW, on taking an exception to EL1 from AArch32
state, and copied to PSTATE.nRW on executing an exception return operation in EL1.

0b1 AArch32 execution state.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]
AArch32 Mode. Set to the value of PSTATE.M[3:0] on taking an exception to EL1, and copied to PSTATE.M[3:0] on executing an exception return operation in EL1.

0b0000 User.
0b0001 FIQ.
0b0010 IRQ.
0b0011 Supervisor.
0b0111 Abort.
0b1011 Undefined.
0b1111 System.

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL1 is an illegal return event, as described in Illegal exception returns from AArch64 state on page D1-4645.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When exception taken from AArch64 state:

An exception return from EL1 using AArch64 makes SPSR_EL1 become UNKNOWN.

Bits [63:32]
Reserved, RES0.

N, bit [31]
Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL1, and copied to PSTATE.N on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]
Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL1, and copied to PSTATE.Z on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]
Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL1, and copied to PSTATE.C on executing an exception return operation in EL1.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**V, bit [28]**

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL1, and copied to PSTATE.V on executing an exception return operation in EL1.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [27:26]**

Reserved, RES0.

**TCO, bit [25]**

When **FEAT_MTE is implemented**:
Tag Check Override. Set to the value of PSTATE.TCO on taking an exception to EL1, and copied to PSTATE.TCO on executing an exception return operation in EL1.
When **FEAT_MTE is not implemented**, it is **CONSTRAINED UNPREDICTABLE** whether this field is RES0 or behaves as if **FEAT_MTE is implemented**.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**DIT, bit [24]**

When **FEAT_DIT is implemented**:
Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL1, and copied to PSTATE.DIT on executing an exception return operation in EL1.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**UAO, bit [23]**

When **FEAT_UAO is implemented**:
User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL1, and copied to PSTATE.UAO on executing an exception return operation in EL1.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**PAN, bit [22]**

When **FEAT_PAN is implemented**:
Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL1, and copied to PSTATE.PAN on executing an exception return operation in EL1.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.
SS, bit [21]
Software Step. Set to the value of PSTATE.SS on taking an exception to EL1, and conditionally copied to PSTATE.SS on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]
Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL1, and copied to PSTATE.IL on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:14]
Reserved, RES0.

ALLINT, bit [13]
When FEAT_NMI is implemented:
All IRQ or FIQ interrupts mask. Set to the value of PSTATE.ALLINT on taking an exception to EL1, and copied to PSTATE.ALLINT on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

SSBS, bit [12]
When FEAT_SSBS is implemented:
Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL1, and copied to PSTATE.SSBS on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

BTYPE, bits [11:10]
When FEAT_BTI is implemented:
Branch Type Indicator. Set to the value of PSTATE.BTYPE on taking an exception to EL1, and copied to PSTATE.BTYPE on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

D, bit [9]
Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL1, and copied to PSTATE.D on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]
SError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL1, and copied to PSTATE.A on executing an exception return operation in EL1.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**I, bit [7]**

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL1, and copied to PSTATE.I on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**F, bit [6]**

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL1, and copied to PSTATE.F on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [5]**

Reserved, RES0.

**M[4], bit [4]**

Execution state. Set to 0b0, the value of PSTATE.nRW, on taking an exception to EL1 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL1.

0b0  AArch64 execution state.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**M[3:0], bits [3:0]**

AArch64 Exception level and selected Stack Pointer.

0b0000  EL0t.
0b0100  EL1t.
0b0101  EL1h.

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL1 is an illegal return event, as described in **Illegal exception returns from AArch64 state on page D1-4645**.

The bits in this field are interpreted as follows:
• M[3:2]: On an exception to EL1:
  — If the Effective value of HCR_EL2.{NV, NV1} != {1,0} or the exception is not taken from EL1, then M[3:2] is set to the value of PSTATE.EL on taking an exception to EL1.
  — If the Effective value of HCR_EL2.{NV, NV1} == {1,0} and the exception is not taken from EL1, then M[3:2] is set to 0b10.
  — M[3:2] is copied to PSTATE.EL on executing a legal exception return operation in EL1.
• M[1] is unused and is 0 for all non-reserved values.
• M[0] is set to the value of PSTATE.SP on taking an exception to EL1 and copied to PSTATE.SP on executing an exception return operation in EL1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Accessing SPSR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic SPSR_EL1 or SPSR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, SPSR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x160];
  else
    X[t, 64] = SPSR_EL1;
  endif
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = SPSR_EL2;
  else
    X[t, 64] = SPSR_EL1;
  endif
elsif PSTATE.EL == EL3 then
  X[t, 64] = SPSR_EL1;
```

**MSR SPSR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    NVMem[0x160] = X[t, 64];
  else
    SPSR_EL1 = X[t, 64];
  endif
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    SPSR_EL2 = X[t, 64];
  else
    SPSR_EL1 = X[t, 64];
  endif
elsif PSTATE.EL == EL3 then
  SPSR_EL1 = X[t, 64];
```
MRS <Xt>, SPSR_EL12

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        X[t, 64] = NVMem[0x160];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        X[t, 64] = SPSR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        X[t, 64] = SPSR_EL1;
    else
        UNDEFINED;

MSR SPSR_EL12, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x160] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        SPSR_EL1 = X[t, 64];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        SPSR_EL1 = X[t, 64];
    else
        UNDEFINED;
### MRS <Xt>, SPSR_EL2

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = SPSR_EL1;
elseif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elseif PSTATE.EL == EL2 then
  X[t, 64] = SPSR_EL2;
elseif PSTATE.EL == EL3 then
  X[t, 64] = SPSR_EL2;
```

### MSR SPSR_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    SPSR_EL1 = X[t, 64];
elseif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elseif PSTATE.EL == EL2 then
  SPSR_EL2 = X[t, 64];
elseif PSTATE.EL == EL3 then
  SPSR_EL2 = X[t, 64];
```
C5.2.19 SPSR_EL2, Saved Program Status Register (EL2)

The SPSR_EL2 characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to EL2.

**Configurations**

AArch64 System register SPSR_EL2 bits [31:0] are architecturally mapped to AArch32 System register SPSR_hyp[31:0].

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

SPSR_EL2 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported and exception taken from AArch32 state:*

An exception return from EL2 using AArch64 makes SPSR_EL2 become **UNKNOWN**.

**Bits [63:32]**

Reserved, **RES0**.

**N, bit [31]**

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL2, and copied to PSTATE.N on executing an exception return operation in EL2.

The reset behavior of this field is:

* On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Z, bit [30]**

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL2, and copied to PSTATE.Z on executing an exception return operation in EL2.

The reset behavior of this field is:

* On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**C, bit [29]**

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL2, and copied to PSTATE.C on executing an exception return operation in EL2.

The reset behavior of this field is:

* On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**V, bit [28]**

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL2, and copied to PSTATE.V on executing an exception return operation in EL2.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]
Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to EL2, and copied to PSTATE.Q on executing an exception return operation in EL2.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]
If-Then. Set to the value of PSTATE.IT on taking an exception to EL2, and copied to PSTATE.IT on executing an exception return operation in EL2.
SPSR_EL2.IT must contain a value that is valid for the instruction being returned to.
The IT field is split as follows:
• IT[1:0] is SPSR_EL2[26:25].
• IT[7:2] is SPSR_EL2[15:10].
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

DIT, bit [24]
When FEAT_DIT is implemented:
Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL2, and copied to PSTATE.DIT on executing an exception return operation in EL2.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

SSBS, bit [23]
When FEAT_SSBS is implemented:
Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL2, and copied to PSTATE.SSBS on executing an exception return operation in EL2.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

PAN, bit [22]
When FEAT_PAN is implemented:
Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL2, and copied to PSTATE.PAN on executing an exception return operation in EL2.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

SS, bit [21]
Software Step. Set to the value of PSTATE.SS on taking an exception to EL2, and conditionally copied to PSTATE.SS on executing an exception return operation in EL2.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**IL, bit [20]**

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL2, and copied to PSTATE.IL on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**GE, bits [19:16]**

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to EL2, and copied to PSTATE.GE on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**E, bit [9]**

Endianness. Set to the value of PSTATE.E on taking an exception to EL2, and copied to PSTATE.E on executing an exception return operation in EL2.

If the implementation does not support big-endian operation, SPSR_EL2.E is **RES0**. If the implementation does not support little-endian operation, SPSR_EL2.E is **RES1**. On executing an exception return operation in EL2, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_EL2.E is **RES0**, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_EL2.E is **RES1**.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**A, bit [8]**

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL2, and copied to PSTATE.A on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**I, bit [7]**

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL2, and copied to PSTATE.I on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**F, bit [6]**

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL2, and copied to PSTATE.F on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**T, bit [5]**

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to EL2, and copied to PSTATE.T on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
M[4], bit [4]

Execution state. Set to 0b1, the value of PSTATE.nRW, on taking an exception to EL2 from AArch32 state, and copied to PSTATE.nRW on executing an exception return operation in EL2.

0b1 AArch32 execution state.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch32 Mode. Set to the value of PSTATE.M[3:0] on taking an exception to EL2, and copied to PSTATE.M[3:0] on executing an exception return operation in EL2.

0b0000 User.
0b0001 FIQ.
0b0010 IRQ.
0b0011 Supervisor.
0b0111 Abort.
0b1010 Hyp.
0b1011 Undefined.
0b1111 System.

Other values are reserved. If SPSR_EL2.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL2 is an illegal return event, as described in Illegal exception returns from AArch64 state on page D1-4645.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When exception taken from AArch64 state:

An exception return from EL2 using AArch64 makes SPSR_EL2 become UNKNOWN.

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL2, and copied to PSTATE.N on executing an exception return operation in EL2.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL2, and copied to PSTATE.Z on executing an exception return operation in EL2.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.
C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL2, and copied to PSTATE.C on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL2, and copied to PSTATE.V on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:26]

Reserved, RES0.

TCO, bit [25]

When FEAT_MTE is implemented:

Tag Check Override. Set to the value of PSTATE.TCO on taking an exception to EL2, and copied to PSTATE.TCO on executing an exception return operation in EL2.

When FEAT_MTE is not implemented, it is CONstrained UNpredictable whether this field is RES0 or behaves as if FEAT_MTE is implemented.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL2, and copied to PSTATE.DIT on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UAO, bit [23]

When FEAT_UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL2, and copied to PSTATE.UAO on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL2, and copied to PSTATE.PAN on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Otherwise:
Reserved, RES0.

SS, bit [21]
Software Step. Set to the value of PSTATE.SS on taking an exception to EL2, and conditionally copied to PSTATE.SS on executing an exception return operation in EL2.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]
Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL2, and copied to PSTATE.IL on executing an exception return operation in EL2.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:14]
Reserved, RES0.

ALLINT, bit [13]
When FEAT_NMI is implemented:
All IRQ or FIQ interrupts mask. Set to the value of PSTATE.ALLINT on taking an exception to EL2, and copied to PSTATE.ALLINT on executing an exception return operation in EL2.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Otherwise:
Reserved, RES0.

SSBS, bit [12]
When FEAT_SSBS is implemented:
Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL2, and copied to PSTATE.SSBS on executing an exception return operation in EL2.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Otherwise:
Reserved, RES0.

BTYPE, bits [11:10]
When FEAT_BTI is implemented:
Branch Type Indicator. Set to the value of PSTATE.BTYPE on taking an exception to EL2, and copied to PSTATE.BTYPE on executing an exception return operation in EL2.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Otherwise:
Reserved, RES0.

D, bit [9]
Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL2, and copied to PSTATE.D on executing an exception return operation in EL2.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
A, bit [8]

S_Error interrupt mask. Set to the value of PSTATE.A on taking an exception to EL2, and copied to PSTATE.A on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL2, and copied to PSTATE.I on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL2, and copied to PSTATE.F on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to 0b0, the value of PSTATE.nRW, on taking an exception to EL2 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL2.

0b0  AArch64 execution state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

0b0000  EL0t.
0b0100  EL1t.
0b0101  EL1h.
0b1000  EL2t.
0b1001  EL2h.

Other values are reserved. If SPSR_EL2.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL2 is an illegal return event, as described in Illegal exception returns from AArch64 state on page D1-4645.

The bits in this field are interpreted as follows:

- M[3:2] is set to the value of PSTATE.EL on taking an exception to EL2 and copied to PSTATE.EL on executing an exception return operation in EL2.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on taking an exception to EL2 and copied to PSTATE.SP on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic SPSR_EL2 or SPSR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.
Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, SPSR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
 UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = SPSR_EL1;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        X[t, 64] = SPSR_EL2;
    elsif PSTATE.EL == EL3 then
        X[t, 64] = SPSR_EL2;

**MSR SPSR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
 UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        SPSR_EL1 = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        SPSR_EL2 = X[t, 64];
    elsif PSTATE.EL == EL3 then
        SPSR_EL2 = X[t, 64];

**MRS <Xt>, SPSR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
 UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x160];
    else
        UNDEFINED;
X[t, 64] = SPSR_EL1;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = SPSR_EL2;
  else
    X[t, 64] = SPSR_EL1;
elsif PSTATE_EL == EL3 then
  X[t, 64] = SPSR_EL1;

MSR SPSR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    NVMem[0x160] = X[t, 64];
  else
    SPSR_EL1 = X[t, 64];
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    SPSR_EL2 = X[t, 64];
  else
    SPSR_EL1 = X[t, 64];
elsif PSTATE_EL == EL3 then
  SPSR_EL1 = X[t, 64];
C5.2.20  SPSR_EL3, Saved Program Status Register (EL3)

The SPSR_EL3 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL3.

Configurations

This register is present only when EL3 is implemented. Otherwise, direct accesses to SPSR_EL3 are UNDEFINED.

Attributes

SPSR_EL3 is a 64-bit register.

Field descriptions

*When AArch32 is supported and exception taken from AArch32 state:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL3, and copied to PSTATE.N on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL3, and copied to PSTATE.Z on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL3, and copied to PSTATE.C on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL3, and copied to PSTATE.V on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
Q, bit [27]
Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to EL3, and copied to PSTATE.Q on executing an exception return operation in EL3.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]
If-Then. Set to the value of PSTATE.IT on taking an exception to EL3, and copied to PSTATE.IT on executing an exception return operation in EL3.

SPSR_EL1.IT must contain a value that is valid for the instruction being returned to.

The IT field is split as follows:
- IT[1:0] is SPSR_EL3[26:25].
- IT[7:2] is SPSR_EL3[15:10].

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DIT, bit [24]

When FEAT_DIT is implemented:
Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL3, and copied to PSTATE.DIT on executing an exception return operation in EL3.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

SSBS, bit [23]

When FEAT_SSBS is implemented:
Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL3, and copied to PSTATE.SSBS on executing an exception return operation in EL3.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:
Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL3, and copied to PSTATE.PAN on executing an exception return operation in EL3.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

SS, bit [21]
Software Step. Set to the value of PSTATE.SS on taking an exception to EL3, and conditionally copied to PSTATE.SS on executing an exception return operation in EL3.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
IL, bit [20]
Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL3, and copied to PSTATE.IL on executing an exception return operation in EL3.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]
Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to EL3, and copied to PSTATE.GE on executing an exception return operation in EL3.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]
Endianness. Set to the value of PSTATE.E on taking an exception to EL3, and copied to PSTATE.E on executing an exception return operation in EL3.

If the implementation does not support big-endian operation, SPSR_EL1.E is RES0. If the implementation does not support little-endian operation, SPSR_EL1.E is RES1. On executing an exception return operation in EL3, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_EL1.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_EL1.E is RES1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]
SError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL3, and copied to PSTATE.A on executing an exception return operation in EL3.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]
IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL3, and copied to PSTATE.I on executing an exception return operation in EL3.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]
FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL3, and copied to PSTATE.F on executing an exception return operation in EL3.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]
T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to EL3, and copied to PSTATE.T on executing an exception return operation in EL3.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4], bit [4]
Execution state. Set to \(0b1\), the value of PSTATE.nRW, on taking an exception to EL3 from AArch32 state, and copied to PSTATE.nRW on executing an exception return operation in EL3.

\(0b1\) AArch32 execution state.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**M[3:0], bits [3:0]**

AArch32 Mode. Set to the value of PSTATE.M[3:0] on taking an exception to EL3, and copied to PSTATE.M[3:0] on executing an exception return operation in EL3.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>User</td>
</tr>
<tr>
<td>0b0001</td>
<td>FIQ</td>
</tr>
<tr>
<td>0b0010</td>
<td>IRQ</td>
</tr>
<tr>
<td>0b0110</td>
<td>Monitor</td>
</tr>
<tr>
<td>0b0111</td>
<td>Abort</td>
</tr>
<tr>
<td>0b1010</td>
<td>Hyp</td>
</tr>
<tr>
<td>0b1011</td>
<td>Undefined</td>
</tr>
<tr>
<td>0b1111</td>
<td>System</td>
</tr>
</tbody>
</table>

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL3 is an illegal return event, as described in *Illegal exception returns from AArch64 state* on page D1-4645.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### When exception taken from AArch64 state:

An exception return from EL3 using AArch64 makes SPSR_EL1 become UNKNOWN.

**Bits [63:32]**

Reserved, RES0.

**N, bit [31]**

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL3, and copied to PSTATE.N on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Z, bit [30]**

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL3, and copied to PSTATE.Z on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
C, bit [29]
Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL3, and copied to PSTATE.C on executing an exception return operation in EL3.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]
Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL3, and copied to PSTATE.V on executing an exception return operation in EL3.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:26]
Reserved, RES0.

TCO, bit [25]
When FEAT_MTE is implemented:
Tag Check Override. Set to the value of PSTATE.TCO on taking an exception to EL3, and copied to PSTATE.TCO on executing an exception return operation in EL3.

When FEAT_MTE is not implemented, it is CONSTRAINED UNPREDICTABLE whether this field is RES0 or behaves as if FEAT_MTE is implemented.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

DIT, bit [24]
When FEAT_DIT is implemented:
Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL3, and copied to PSTATE.DIT on executing an exception return operation in EL3.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

UAO, bit [23]
When FEAT_UAO is implemented:
User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL3, and copied to PSTATE.UAO on executing an exception return operation in EL3.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

PAN, bit [22]
When FEAT_PAN is implemented:
Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL3, and copied to PSTATE.PAN on executing an exception return operation in EL3.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**SSBS**, bit [12]

When **FEAT_SBS** is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL3, and copied to PSTATE.SSBS on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

Otherwise:

Reserved, RES0.

**BTYPE**, bits [11:10]

When **FEAT_BT** is implemented:

Branch Type Indicator. Set to the value of PSTATE.BTYPE on taking an exception to EL3, and copied to PSTATE.BTYPE on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

Otherwise:

Reserved, RES0.

**D**, bit [9]

Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL3, and copied to PSTATE.D on executing an exception return operation in EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
A, bit [8]
SEError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL3, and copied to PSTATE.A on executing an exception return operation in EL3.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]
IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL3, and copied to PSTATE.I on executing an exception return operation in EL3.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]
FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL3, and copied to PSTATE.F on executing an exception return operation in EL3.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]
Reserved, RES0.

M[4], bit [4]
Execution state. Set to 0b0, the value of PSTATE.nRW, on taking an exception to EL3 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL3.
0b0 AArch64 execution state.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]
AArch64 Exception level and selected Stack Pointer.
0b0000 EL0t.
0b0100 EL1t.
0b0101 EL1h.
0b1000 EL2t.
0b1001 EL2h.
0b1100 EL3t.
0b1101 EL3h.

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL3 is an illegal return event, as described in Illegal exception returns from AArch64 state on page D1-4645.

The bits in this field are interpreted as follows:
• M[3:2] is set to the value of PSTATE.EL on taking an exception to EL3 and copied to PSTATE.EL on executing an exception return operation in EL3.
• M[1] is unused and is 0 for all non-reserved values.
• M[0] is set to the value of PSTATE.SP on taking an exception to EL3 and copied to PSTATE.SP on executing an exception return operation in EL3.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Accessing SPSR_EL3

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, SPSR_EL3**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  X[t, 64] = SPSR_EL3;

**MSR SPSR_EL3, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  SPSR_EL3 = X[t, 64];
C5.2.21 SPSR_fiq, Saved Program Status Register (FIQ mode)

The SPSR_fiq characteristics are:

Purpose

Holds the saved process state when an exception is taken to FIQ mode.

Configurations

AArch64 System register SPSR_fiq bits [31:0] are architecturally mapped to AArch32 System register SPSR_fiq[31:0].

If EL1 only supports execution in AArch64 state, this register is RES0 from EL2 and EL3.

Attributes

SPSR_fiq is a 64-bit register.

Field descriptions

When EL1 can only use AArch64:

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td></td>
</tr>
</tbody>
</table>

Bits [63:0]

Reserved, RES0.

Otherwise:

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td></td>
</tr>
</tbody>
</table>

| 31 | 28 | 27 | 26 | 25 | 22 | 21 | 20 | 19 | 16 | 15 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| N  | Z  | C  | V  | Q | J | IL | GE | IT[7:2] | E | A | I | F | T | M[4:0] |
| IT[1:0] | SSBS | DIT | PAN |

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to FIQ mode, and copied to PSTATE.N on executing an illegal exception return operation in FIQ mode.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to FIQ mode, and copied to PSTATE.Z on executing an illegal exception return operation in FIQ mode.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.
C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to FIQ mode, and copied to PSTATE.C on executing an illegal exception return operation in FIQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to FIQ mode, and copied to PSTATE.V on executing an illegal exception return operation in FIQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to FIQ mode, and copied to PSTATE.Q on executing an illegal exception return operation in FIQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to FIQ mode, and copied to PSTATE.IT on executing an illegal exception return operation in FIQ mode.

SPSR_fiq.IT must contain a value that is valid for the instruction being returned to.

The IT field is split as follows:
• IT[1:0] is SPSR_fiq[26:25].
• IT[7:2] is SPSR_fiq[15:10].

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the \{J, T\} bits determined the AArch32 Instruction set state. Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:
Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to FIQ mode, and copied to PSTATE.SSBS on executing an illegal exception return operation in FIQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:
Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to FIQ mode, and copied to PSTATE.PAN on executing an illegal exception return operation in FIQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to FIQ mode, and copied to PSTATE.DIT on executing an illegal exception return operation in FIQ mode.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to FIQ mode, and copied to PSTATE.IL on executing an illegal exception return operation in FIQ mode.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to FIQ mode, and copied to PSTATE.GE on executing an illegal exception return operation in FIQ mode.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to FIQ mode, and copied to PSTATE.E on executing an illegal exception return operation in FIQ mode.

If the implementation does not support big-endian operation, SPSR_fiq.E is RES0. If the implementation does not support little-endian operation, SPSR_fiq.E is RES1. On executing an illegal exception return operation in FIQ mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_fiq.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_fiq.E is RES1.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to FIQ mode, and copied to PSTATE.A on executing an illegal exception return operation in FIQ mode.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to FIQ mode, and copied to PSTATE.I on executing an illegal exception return operation in FIQ mode.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to FIQ mode, and copied to PSTATE.F on executing an illegal exception return operation in FIQ mode.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to FIQ mode, and copied to PSTATE.T on executing an illegal exception return operation in FIQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to FIQ mode, and copied to PSTATE.M[4:0] on executing an illegal exception return operation in FIQ mode.

- 0b10000: User
- 0b10001: FIQ
- 0b10010: IRQ
- 0b10011: Supervisor
- 0b10111: Abort
- 0b11011: Undefined
- 0b11111: System

Other values are reserved. If SPSR_fiq.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an illegal exception return operation in FIQ mode is an illegal return event, as described in Illegal return events from AArch32 state on page G1-8952.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_fiq

Accesses to this register use the following encodings in the System register encoding space:

\[ \text{MRS <Xt>, SPSR_fiq} \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0100</td>
<td>0b0011</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = SPSR_fiq;
elsif PSTATE.EL == EL3 then
  X[t, 64] = SPSR_fiq;
### MSR SPSR_fiq, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0100</td>
<td>0b0011</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    end if;
elsif PSTATE.EL == EL2 then
    SPSR_fiq = X[t, 64];
elsif PSTATE.EL == EL3 then
    SPSR_fiq = X[t, 64];
C5.2.22  **SPSR_irq, Saved Program Status Register (IRQ mode)**

The SPSR_irq characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to IRQ mode.

**Configurations**

AArch64 System register SPSR_irq bits [31:0] are architecturally mapped to AArch32 System register SPSR_irq[31:0].

If EL1 only supports execution in AArch64 state, this register is RES0 from EL2 and EL3.

**Attributes**

SPSR_irq is a 64-bit register.

**Field descriptions**

*When EL1 can only use AArch64:*

![Diagram of SPSR_irq bits]

**Bits [63:0]**

Reserved, RES0.

**Otherwise:**

![Diagram of SPSR_irq bits]

**Bits [63:32]**

Reserved, RES0.

**N, bit [31]**

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to IRQ mode, and copied to PSTATE.N on executing an illegal exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Z, bit [30]**

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to IRQ mode, and copied to PSTATE.Z on executing an illegal exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to IRQ mode, and copied to PSTATE.C on executing an illegal exception return operation in IRQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to IRQ mode, and copied to PSTATE.V on executing an illegal exception return operation in IRQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to IRQ mode, and copied to PSTATE.Q on executing an illegal exception return operation in IRQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to IRQ mode, and copied to PSTATE.IT on executing an illegal exception return operation in IRQ mode.

SPSR_irq.IT must contain a value that is valid for the instruction being returned to.

The IT field is split as follows:
• IT[1:0] is SPSR_irq[26:25].
• IT[7:2] is SPSR_irq[15:10].

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:
Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to IRQ mode, and copied to PSTATE.SSBS on executing an illegal exception return operation in IRQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:
Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to IRQ mode, and copied to PSTATE.PAN on executing an illegal exception return operation in IRQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Otherwise:
    Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:
    Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to IRQ mode, and copied to PSTATE.DIT on executing an illegal exception return operation in IRQ mode.
    The reset behavior of this field is:
    • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
    Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to IRQ mode, and copied to PSTATE.IL on executing an illegal exception return operation in IRQ mode.
    The reset behavior of this field is:
    • On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to IRQ mode, and copied to PSTATE.GE on executing an illegal exception return operation in IRQ mode.
    The reset behavior of this field is:
    • On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to IRQ mode, and copied to PSTATE.E on executing an illegal exception return operation in IRQ mode.

If the implementation does not support big-endian operation, SPSR_irq.E is RES0. If the implementation does not support little-endian operation, SPSR_irq.E is RES1. On executing an illegal exception return operation in IRQ mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_irq.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_irq.E is RES1.
    The reset behavior of this field is:
    • On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to IRQ mode, and copied to PSTATE.A on executing an illegal exception return operation in IRQ mode.
    The reset behavior of this field is:
    • On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to IRQ mode, and copied to PSTATE.I on executing an illegal exception return operation in IRQ mode.
    The reset behavior of this field is:
    • On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to IRQ mode, and copied to PSTATE.F on executing an illegal exception return operation in IRQ mode.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**T, bit [5]**

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to IRQ mode, and copied to PSTATE.T on executing an illegal exception return operation in IRQ mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**M[4:0], bits [4:0]**

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to IRQ mode, and copied to PSTATE.M[4:0] on executing an illegal exception return operation in IRQ mode.

- 0b10000 User.
- 0b10001 FIQ.
- 0b10010 IRQ.
- 0b10011 Supervisor.
- 0b10111 Abort.
- 0b11011 Undefined.
- 0b11111 System.

Other values are reserved. If SPSR_irq.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an illegal exception return operation in IRQ mode is an illegal return event, as described in _Illegal return events from AArch32 state_ on page G1-8952.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing SPSR_irq**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, SPSR_irq**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0100</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SPSR_irq;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SPSR_irq;
The A64 System Instruction Class
C5.2 Special-purpose registers

**MSR SPSR_irq, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0100</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elseif PSTATE.EL == EL2 then
  SPSR_irq = X[t, 64];
elseif PSTATE.EL == EL3 then
  SPSR_irq = X[t, 64];
C5.2.23 SPSR_und, Saved Program Status Register (Undefined mode)

The SPSR_und characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to Undefined mode.

**Configurations**

AArch64 System register SPSR_und bits [31:0] are architecturally mapped to AArch32 System register SPSR_und[31:0].

If EL1 only supports execution in AArch64 state, this register is RES0 from EL2 and EL3.

**Attributes**

SPSR_und is a 64-bit register.

**Field descriptions**

*When EL1 can only use AArch64:*

```
<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td></td>
</tr>
</tbody>
</table>
```

Bits [63:0]
Reserved, RES0.

*Otherwise:*

```
<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Z</td>
<td>C</td>
<td>V</td>
<td>Q</td>
<td>J</td>
<td>IL</td>
<td>GE</td>
<td>IT[7:2]</td>
<td>E</td>
<td>A</td>
<td>I</td>
<td>F</td>
<td>T</td>
<td>M[4:0]</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>SSBS</td>
<td>DIT</td>
<td>PAN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Bits [63:32]
Reserved, RES0.

**N, bit [31]**

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Undefined mode, and copied to PSTATE.N on executing an illegal exception return operation in Undefined mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Z, bit [30]**

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Undefined mode, and copied to PSTATE.Z on executing an illegal exception return operation in Undefined mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Undefined mode, and copied to PSTATE.C on executing an illegal exception return operation in Undefined mode.

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Undefined mode, and copied to PSTATE.V on executing an illegal exception return operation in Undefined mode.

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Undefined mode, and copied to PSTATE.Q on executing an illegal exception return operation in Undefined mode.

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Undefined mode, and copied to PSTATE.IT on executing an illegal exception return operation in Undefined mode.

SPSR_und.IT must contain a value that is valid for the instruction being returned to.

The IT field is split as follows:
* IT[1:0] is SPSR_und[26:25].
* IT[7:2] is SPSR_und[15:10].

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the \{J, T\} bits determined the AArch32 Instruction set state. Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

*When FEAT_SSBS is implemented:*

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Undefined mode, and copied to PSTATE.SSBS on executing an illegal exception return operation in Undefined mode.

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

PAN, bit [22]

*When FEAT_PAN is implemented:*

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Undefined mode, and copied to PSTATE.PAN on executing an illegal exception return operation in Undefined mode.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**DIT, bit [21]**

*When FEAT_DIT is implemented:*

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Undefined mode, and copied to PSTATE.DIT on executing an illegal exception return operation in Undefined mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**IL, bit [20]**

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Undefined mode, and copied to PSTATE.IL on executing an illegal exception return operation in Undefined mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**GE, bits [19:16]**

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Undefined mode, and copied to PSTATE.GE on executing an illegal exception return operation in Undefined mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**E, bit [9]**

Endianness. Set to the value of PSTATE.E on taking an exception to Undefined mode, and copied to PSTATE.E on executing an illegal exception return operation in Undefined mode.

If the implementation does not support big-endian operation, SPSR_und.E is RES0. If the implementation does not support little-endian operation, SPSR_und.E is RES1. On executing an illegal exception return operation in Undefined mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_und.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_und.E is RES1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**A, bit [8]**

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to Undefined mode, and copied to PSTATE.A on executing an illegal exception return operation in Undefined mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**I, bit [7]**

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Undefined mode, and copied to PSTATE.I on executing an illegal exception return operation in Undefined mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
F, bit [6]
FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Undefined mode, and copied to PSTATE.F on executing an illegal exception return operation in Undefined mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]
T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Undefined mode, and copied to PSTATE.T on executing an illegal exception return operation in Undefined mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]
Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Undefined mode, and copied to PSTATE.M[4:0] on executing an illegal exception return operation in Undefined mode.

<table>
<thead>
<tr>
<th>Value</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10000</td>
<td>User.</td>
</tr>
<tr>
<td>0b10001</td>
<td>FIQ.</td>
</tr>
<tr>
<td>0b10010</td>
<td>IRQ.</td>
</tr>
<tr>
<td>0b10011</td>
<td>Supervisor.</td>
</tr>
<tr>
<td>0b10111</td>
<td>Abort.</td>
</tr>
<tr>
<td>0b11011</td>
<td>Undefined.</td>
</tr>
<tr>
<td>0b11111</td>
<td>System.</td>
</tr>
</tbody>
</table>

Other values are reserved. If SPSR_und.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an illegal exception return operation in Undefined mode is an illegal return event, as described in Illegal return events from AArch32 state on page G1-8952.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_und
Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SPSR_und

<table>
<thead>
<tr>
<th>$op0$</th>
<th>$op1$</th>
<th>$CRn$</th>
<th>$CRm$</th>
<th>$op2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0100</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = SPSR_und;
elsif PSTATE.EL == EL3 then
  X[t, 64] = SPSR_und;
### MSR SPSR_und, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0100</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        SPSR_und = X[t, 64];
    elsif PSTATE.EL == EL3 then
        SPSR_und = X[t, 64];

C5.2.24 SSBS, Speculative Store Bypass Safe

The SSBS characteristics are:

**Purpose**

Allows access to the Speculative Store Bypass Safe bit.

**Configurations**

This register is present only when FEAT_SSBS is implemented. Otherwise, direct accesses to SSBS are UNDEFINED.

**Attributes**

SSBS is a 64-bit register.

**Field descriptions**

![Register Layout]

**Bits [63:13]**

Reserved, RES0.

**SSBS, bit [12]**

Speculative Store Bypass Safe.

Prohibits speculative loads or stores which might practically allow a cache timing side channel.

A cache timing side channel might be exploited where a load or store uses an address that is derived from a register that is being loaded from memory using a load instruction speculatively read from a memory location. If PSTATE.SSBS is enabled, the address derived from the load instruction might be from earlier in the coherence order than the latest store to that memory location with the same virtual address.

- **0b0** Hardware is not permitted to load or store speculatively, in a manner that could practically give rise to a cache timing side channel, using an address derived from a register value that has been loaded from memory using a load instruction (L) that speculatively reads an entry from earlier in the coherence order from that location being loaded from than the entry generated by the latest store (S) to that location using the same virtual address as L.

- **0b1** Hardware is permitted to load or store speculatively, in a manner that could practically give rise to a cache timing side channel, using an address derived from a register value that has been loaded from memory using a load instruction (L) that speculatively reads an entry from earlier in the coherence order from that location being loaded from than the entry generated by the latest store (S) to that location using the same virtual address as L.

The value of this bit is set to the value in the SCTLR_ELx.DSSBS field on taking an exception to ELx.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

**Bits [11:0]**

Reserved, RES0.
Accessing SSBS

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, SSBS**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
X[t, 64] = Zeros(51):PSTATE.SSBS:Zeros(12);
else if PSTATE.EL == EL1 then
X[t, 64] = Zeros(51):PSTATE.SSBS:Zeros(12);
else if PSTATE.EL == EL2 then
X[t, 64] = Zeros(51):PSTATE.SSBS:Zeros(12);
else if PSTATE.EL == EL3 then
X[t, 64] = Zeros(51):PSTATE.SSBS:Zeros(12);

**MSR SSBS, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
PSTATE.SSBS = X[t, 64]<12>;
else if PSTATE.EL == EL1 then
PSTATE.SSBS = X[t, 64]<12>;
else if PSTATE.EL == EL2 then
PSTATE.SSBS = X[t, 64]<12>;
else if PSTATE.EL == EL3 then
PSTATE.SSBS = X[t, 64]<12>;

**MSR SSBS, #<imm>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

C5.2.25 SVCR, Streaming Vector Control Register

The SVCR characteristics are:

**Purpose**

Controls Streaming SVE mode and SME behavior.

**Configurations**

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SVCR are UNDEFINED.

**Attributes**

SVCR is a 64-bit register.

**Field descriptions**

```
<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
<th>31</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>RES0</td>
<td>ZA</td>
<td>SM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Bits [63:2]**

Reserved, RES0.

**ZA, bit [1]**

Enables SME ZA storage.

When this storage is disabled, execution of an instruction which can access it is trapped. The exception is reported using an ESR_ELx.{EC, SMTC} value of \{0x1D, 0x3\}.

The possible values of this bit are:

- \(0b0\) SME ZA storage is invalid and not accessible.
  - This control causes execution at any Exception level of instructions that can access this storage to be trapped.

- \(0b1\) SME ZA storage is valid and accessible.
  - This control does not cause execution of any instructions to be trapped.

When a write to SVCR.ZA changes the value of PSTATE.ZA, the following applies:

- When changed from 0 to 1, all implemented bits of the storage are set to zero.
- When changed from 1 to 0, there is no observable change to the storage.

Changes to this field do not have an affect on the SVE vector and predicate registers and FPSR.

A direct or indirect read of ZA appears to occur in program order relative to a direct write of SVCR, and to MSR SVCRZA and MSR SVCRSMZA instructions, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

**SM, bit [0]**

Enables Streaming SVE mode.

When the PE is in Streaming SVE mode, the Streaming SVE vector length (SVL) applies to SVE instructions, and execution at any Exception level of an instruction which is illegal in that mode is trapped. The exception is reported using an ESR_ELx.{EC, SMTC} value of \{0x1D, 0x1\}. 

---

The A64 System Instruction Class
C5.2 Special-purpose registers
When the PE is not in Streaming SVE mode, the SVE vector length (VL) applies to SVE instructions, and execution at any Exception level of an instruction which is only legal in that mode is trapped. The exception is reported using an ESR_ELx.\{EC, SMTC\} value of \{0x1D, 0x2\}.

The possible values of this bit are:

0b0  The PE is not in Streaming SVE mode.
0b1  The PE is in Streaming SVE mode.

When a write to SVCR.SM changes the value of PSTATE.SM, the following applies:

- When changed from 0 to 1, an entry to Streaming SVE mode is performed.
- When changed from 1 to 0, an exit from Streaming SVE mode is performed.
- All implemented bits of the SVE registers Z0-Z31, P0-P15, and FFR in the new mode are set to zero.
- FPSR in the new mode is set to 0x0000_0000_0800_009f, in which all cumulative status bits are set to 1.

Changes to this field do not have an affect on SME ZA storage.

A direct or indirect read of SM appears to occur in program order relative to a direct write of SVCR, and to MSR SVCSM and MSR SVCSMZA instructions, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

### Accessing SVCR

SVCR is read/write and can be accessed from any Exception level.

Accesses to this register use the following encodings in the System register encoding space:

\[
\begin{array}{cccccc}
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>
\end{array}
\]

if PSTATE_EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" then
    UNDEFINED;
  elsif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.SMEN != '11' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x1D);
    elseif EL2Enabled() && HCR_EL2.E2H == '1' && CPACR_EL1.SMEN != '11' then
      AArch64.SystemAccessTrap(EL2, 0x1D);
    elseif EL2Enabled() && HCR_EL2.E2H == '0' && CPACR_EL2.TSM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x1D);
    elseif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x1D);
      end if;
    else
      X[t, 64] = Zeros(62):PSTATE.<ZA,SM>;
    end if;
  elseif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" then
      UNDEFINED;
    else
      X[t, 64] = Zeros(62):PSTATE.<ZA,SM>;
    end if;
  end if;
else
  X[t, 64] = Zeros(62):PSTATE.<ZA,SM>;
end if;
elsif CPACR_EL1.SMEN == 'x0' then
  AArch64.SystemAccessTrap(EL1, 0x1D);
estif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
  AArch64.SystemAccessTrap(EL2, 0x1D);
estif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
  AArch64.SystemAccessTrap(EL2, 0x1D);
estif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x1D);
  else
    X[t, 64] = Zeros(62):PSTATE.<ZA,SM>;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
    UNDEFINED;
estif HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
  AArch64.SystemAccessTrap(EL2, 0x1D);
estif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
  AArch64.SystemAccessTrap(EL2, 0x1D);
estif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x1D);
else
  X[t, 64] = Zeros(62):PSTATE.<ZA,SM>;
estif PSTATE.EL == EL3 then
  if CPTR_EL3.ESM == '0' then
    AArch64.SystemAccessTrap(EL3, 0x1D);
else
  X[t, 64] = Zeros(62):PSTATE.<ZA,SM>;

MSR SVCR, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
    UNDEFINED;
estif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.SMEN != '11' then
  if EL2Enabled() && HCR_EL2.TGE == '1' then
    AArch64.SystemAccessTrap(EL2, 0x1D);
  else
    AArch64.SystemAccessTrap(EL1, 0x1D);
else
  if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.SMEN != '11' then
    AArch64.SystemAccessTrap(EL2, 0x1D);
estif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
  AArch64.SystemAccessTrap(EL2, 0x1D);
estif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
  AArch64.SystemAccessTrap(EL2, 0x1D);
estif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x1D);
else
  SetPSTATE_SVCR(X[t, 32]);
estif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1' && CPTR_EL3.ESM == '0' then
  UNDEFINED;
else if CPACR_EL1.SMEN == 'x0' then
  AArch64.SystemAccessTrap(EL1, 0x1D);
else if EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
  AArch64.SystemAccessTrap(EL2, 0x1D);
else if EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
  AArch64.SystemAccessTrap(EL2, 0x1D);
else if HaveEL(EL3) && CPTR_EL3.ESM == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x1D);
  else
    SetPSTATE_SVCR(X[t, 32]);
endif;
endif;
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && CPTR_EL3.ESM == '0' then
    UNDEFINED;
  elsif HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x1D);
  elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x1D);
  elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x1D);
    else
      SetPSTATE_SVCR(X[t, 32]);
    endif;
  endif;
else
  if PSTATE_EL == EL3 then
    if CPTR_EL3.ESM == '0' then
      AArch64.SystemAccessTrap(EL3, 0x1D);
    else
      SetPSTATE_SVCR(X[t, 32]);
    endif;
  endif;
endif;

**MSR SVCRSM, #<imm>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b001x</td>
<td>0b011</td>
</tr>
</tbody>
</table>

**MSR SVCRZA, #<imm>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b010x</td>
<td>0b011</td>
</tr>
</tbody>
</table>

**MSR SVCRSMZA, #<imm>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b011x</td>
<td>0b011</td>
</tr>
</tbody>
</table>
C5.2.26   TCO, Tag Check Override

The TCO characteristics are:

**Purpose**

When `FEAT_MTE` is implemented, this register allows tag checks to be disabled globally.

When `FEAT_MTE` is not implemented, it is **CONSTRAINED UNPREDICTABLE** whether this register is `RES0` or behaves as if `FEAT_MTE` is implemented.

**Configurations**

This register is present only when `FEAT_MTE` is implemented. Otherwise, direct accesses to TCO are **UNDEFINED**.

**Attributes**

TCO is a 64-bit register.

**Field descriptions**

- **Bits [63:26]**
  - Reserved, `RES0`.

- **TCO, bit [25]**
  - Allows memory tag checks to be globally disabled.
  - 0b0: Loads and Stores are not affected by this control.
  - 0b1: Loads and Stores are unchecked.

- **Bits [24:0]**
  - Reserved, `RES0`.

**Accessing TCO**

For information about the operation of the MSR (immediate) accessor, see **MSR (immediate)**.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TCO**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b11</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  X[t, 64] = Zeros(38):PSTATE.TCO:Zeros(25);
elsif PSTATE_EL == EL1 then
  X[t, 64] = Zeros(38):PSTATE.TCO:Zeros(25);
elsif PSTATE_EL == EL2 then
  X[t, 64] = Zeros(38):PSTATE.TCO:Zeros(25);

elsif PSTATE.EL == EL3 then
    X[t, 64] = Zeros(38):PSTATE.TCO:Zeros(25);

**MSR TCO, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0010</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    PSTATE.TCO = X[t, 64]<25>;
elsif PSTATE.EL == EL1 then
    PSTATE.TCO = X[t, 64]<25>;
elsif PSTATE.EL == EL2 then
    PSTATE.TCO = X[t, 64]<25>;
elsif PSTATE.EL == EL3 then
    PSTATE.TCO = X[t, 64]<25>;

**MSR TCO, #<imm>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b100</td>
</tr>
</tbody>
</table>
C5.2.27 UAO, User Access Override

The UAO characteristics are:

Purpose

Allows access to the User Access Override bit.

Configurations

This register is present only when FEAT_UAO is implemented. Otherwise, direct accesses to UAO are UNDEFINED.

Attributes

UAO is a 64-bit register.

Field descriptions

Bits [63:24] Reserved, RES0.

UAO, bit [23]

User Access Override.

0b0 The behavior of LDTR* and STTR* instructions is as defined in the base Armv8 architecture.

0b1 When executed at the following Exception levels, LDTR* and STTR* instructions behave as the equivalent LDRs and STRs instructions:

- EL1.
- EL2 with HCR_EL2.{E2H, TGE} == {1, 1}.

When executed at EL3, or at EL2 with HCR_EL2.E2H == 0 or HCR_EL2.TGE == 0, the LDTR* and STTR* instructions behave as the equivalent LDRs and STRs instructions, regardless of the setting of the PSTATE.UAO bit.

Bits [22:0] Reserved, RES0.

Accessing UAO

For more information about the operation of the MSR (immediate) accessor, see MSR (immediate).

Accesses to this register use the following encodings in the System register encoding space:
**MRS <Xt>, UAO**

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b000 & 0b0100 & 0b0010 & 0b100 \\
\end{array}
\]

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(40):PSTATE.UAO:Zeros(23);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(40):PSTATE.UAO:Zeros(23);
elsif PSTATE.EL == EL3 then
    X[t, 64] = Zeros(40):PSTATE.UAO:Zeros(23);

**MSR UAO, <Xt>**

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b000 & 0b0100 & 0b0010 & 0b100 \\
\end{array}
\]

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    PSTATE.UAO = X[t, 64]<23>;
elsif PSTATE.EL == EL2 then
    PSTATE.UAO = X[t, 64]<23>;
elsif PSTATE.EL == EL3 then
    PSTATE.UAO = X[t, 64]<23>;

**MSR UAO, #<imm>**

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{op2} \\
0b00 & 0b000 & 0b0100 & 0b011 \\
\end{array}
\]
C5.3 A64 System instructions for cache maintenance

This section lists the A64 System instructions for cache maintenance.
C5.3.1   DC CGDSW, Clean of Data and Allocation Tags by Set/Way

The DC CGDSW characteristics are:

Purpose

Clean data and Allocation Tags in data cache by set/way.

Configurations

This instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC CGDSW are UNDEFINED.

Attributes

DC CGDSW is a 64-bit System instruction.

Field descriptions

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log2(ASSOCIATIVITY), L = Log2(LINELEN), B = (L + S), S = Log2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC CGDSW instruction

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.
Accesses to this instruction use the following encodings in the System instruction encoding space:

\[ \text{DC CGDSW, }<\text{Xt}> \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1010</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCSW == '1') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_SetWay);
    endif
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_SetWay);
C5.3.2 DC CGDVAC, Clean of Data and Allocation Tags by VA to PoC

The DC CGDVAC characteristics are:

**Purpose**

Clean data and Allocation Tags in data cache by address to Point of Coherency.

**Configurations**

This instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CGDVAC are UNDEFINED.

**Attributes**

DC CGDVAC is a 64-bit System instruction.

**Field descriptions**

**Bits [63:0]**

Virtual address to use. No alignment restrictions apply to this VA.

**Executing DC CGDVAC instruction**

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission fault, subject to the constraints described in Permission fault.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D7-5056.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC CGDVAC, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b011</td>
<td>0b0111</td>
<td>0b1010</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        endif
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        HFGITR_EL2.DCCVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC);
    endif
else
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC);
else
    if PSTATE.EL == EL1 then

if EL2Enabled() && HCR_EL2.TPCP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoC);
C5.3.3 DC CGDVADP, Clean of Data and Allocation Tags by VA to PoDP

The DC CGDVADP characteristics are:

Purpose

Clean Allocation Tags and data in data cache by address to Point of Deep Persistence.

If the memory system does not identify a Point of Deep Persistence, then this instruction behaves as a DC CGDVAP.

Configurations

This instruction is present only when FEAT_DPB2 is implemented and FEAT_MTE is implemented. Otherwise, direct accesses to DC CGDVADP are UNDEFINED.

Attributes

DC CGDVADP is a 64-bit System instruction.

Field descriptions

Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CGDVADP instruction

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission fault, see Permission fault.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC) on page D7-5056.

Accesses to this instruction use the following encodings in the System instruction encoding space:

DC CGDVADP, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b011</td>
<td>0b0111</td>
<td>0b1010</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    end
  else
    if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
      HFGITR_EL2.DCCVADP == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    end
  end
AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP);
elif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.TPCP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVADP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP);
  endif
else
  AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoDP);
endif
C5.3.4 **DC CGDVAP, Clean of Data and Allocation Tags by VA to PoP**

The DC CGDVAP characteristics are:

**Purpose**

Clean data and Allocation Tags in data cache by address to Point of Persistence.

If the memory system does not identify a Point of Persistence, then this instruction behaves as a DC CGDVAC.

**Configurations**

This instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CGDVAP are UNDEFINED.

**Attributes**

DC CGDVAP is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Virtual address to use</th>
</tr>
</thead>
</table>

**Virtual address to use**

Virtual address to use. No alignment restrictions apply to this VA.

**Executing DC CGDVAP instruction**

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission fault, see Permission fault.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D7-5056.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC CGDVAP, <Xi>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b011</td>
<td>0b0111</td>
<td>0b1100</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
else
  AArch64.SystemAccessTrap(EL1, 0x18);
AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP);

elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.TPCP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP);
  end

elsif PSTATE_EL == EL2 then
  AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP);
elsif PSTATE_EL == EL3 then
  AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Clean, CacheOpScope_PoP);
C5.3.5  DC CGSW, Clean of Allocation Tags by Set/Way

The DC CGSW characteristics are:

**Purpose**

Clean Allocation Tags in data cache by set/way.

**Configurations**

This instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC CGSW are UNDEFINED.

**Attributes**

DC CGSW is a 64-bit System instruction.

**Field descriptions**

- **Bits [63:32]**
  - Reserved, RES0.

- **SetWay, bits [31:4]**
  - Contains two fields:
    - Way, bits[31:32-A], the number of the way to operate on.
    - Set, bits[B-1:L], the number of the set to operate on.
  - Bits[L-1:4] are RES0.
  - \( A = \log_2(\text{ASSOCIATIVITY}) \), \( L = \log_2(\text{LINELEN}) \), \( B = (L + S) \), \( S = \log_2(\text{NSETS}) \).
  - ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

- **Level, bits [3:1]**
  - Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

- **Bit [0]**
  - Reserved, RES0.

**Executing DC CGSW instruction**

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.
Accesses to this instruction use the following encodings in the System instruction encoding space:

\[ \text{DC CGSW, \langle Xt \rangle} \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1010</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TSx == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && !HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCSW == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_SetWay);
  endif
elsif PSTATE.EL == EL2 then
  AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
  AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_SetWay);
C5.3.6 DC CGVAC, Clean of Allocation Tags by VA to PoC

The DC CGVAC characteristics are:

**Purpose**
Clean Allocation Tags in data cache by address to Point of Coherency.

**Configurations**
This instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CGVAC are UNDEFINED.

**Attributes**
DC CGVAC is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Virtual address to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:32]</td>
<td>Virtual address to use</td>
</tr>
</tbody>
</table>

**Bits [63:0]**
Virtual address to use. No alignment restrictions apply to this VA.

**Executing DC CGVAC instruction**

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission fault, subject to the constraints described in Permission fault.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D7-5056.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC CGVAC, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b011</td>
<td>0b0111</td>
<td>0b1010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
    HFGITR_EL2.DCCVAC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC);
  end if

elsif PSTATE.EL == EL1 then
if EL2Enabled() && HCR_EL2.TPCP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVC == '1') then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoC);
C5.3.7 DC CGVADP, Clean of Allocation Tags by VA to PoDP

The DC CGVADP characteristics are:

**Purpose**

Clean Allocation tags by address to Point of Deep Persistence.

If the memory system does not identify a Point of Deep Persistence, then this instruction behaves as a DC CGVAP.

**Configurations**

This instruction is present only when FEAT_DPBP2 is implemented and FEAT_MTE is implemented. Otherwise, direct accesses to DC CGVADP are UNDEFINED.

**Attributes**

DC CGVADP is a 64-bit System instruction.

**Field descriptions**

Virtual address to use

No alignment restrictions apply to this VA.

**Executing DC CGVADP instruction**

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission fault, see Permission fault.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D7-5056.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC CGVADP, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b01</td>
<td>0b0111</td>
<td>0b1101</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if !(EL2Enabled() & HCR_EL2.<E2H,TGE> == '11') & SCTL_EL1.UCI == '0' then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
      else
        AArch64.SystemAccessTrap(EL1, 0x18);
    else if EL2Enabled() & HCR_EL2.<E2H,TGE> != '11' & HCR_EL2.TPCP == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & !HaveEL(EL3) || SCR_EL3.FGTEn == '1' &
      HFGITR_EL2.DCCVADP == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL2, 0x18);
  end if
else
  AArch64.SystemAccessTrap(EL1, 0x18);
end if
AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP);
elsif PSTATE_EL == EL1 then
 if EL2Enabled() & HCR_EL2.TPCP == '1' then
   AArch64.SystemAccessTrap(EL2, 0x18);
 elsif EL2Enabled() & ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFGITR_EL2.DCCVADP == '1') then
   AArch64.SystemAccessTrap(EL2, 0x18);
 else
   AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP);
elsif PSTATE_EL == EL2 then
   AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP);
elsif PSTATE_EL == EL3 then
   AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoDP);
C5.3.8 DC CGVAP, Clean of Allocation Tags by VA to PoP

The DC CGVAP characteristics are:

**Purpose**

Clean Allocation Tags in data cache by address to Point of Persistence.

If the memory system does not identify a Point of Persistence, then this instruction behaves as a DC CGVC.

**Configurations**

This instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CGVAP are UNDEFINED.

**Attributes**

DC CGVAP is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Virtual address to use</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>63 32</td>
</tr>
</tbody>
</table>

**Bits [63:0]**

Virtual address to use. No alignment restrictions apply to this VA.

**Executing DC CGVAP instruction**

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission fault, see Permission fault.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC) on page D7-5056.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC CGVAP, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b01</td>
<td>0b011</td>
<td>0b1100</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
    HFGITR_EL2.DCCVAP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
else

The A64 System Instruction Class
C5.3 A64 System instructions for cache maintenance

AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP);
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.TPCP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP);
  endif
elsif PSTATE_EL == EL2 then
  AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP);
elsif PSTATE_EL == EL3 then
  AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Clean, CacheOpScope_PoP);
C5.3.9  DC CIGDPAPA, Clean and Invalidate of Data and Allocation Tags by PA to PoPA

The DC CIGDPAPA characteristics are:

**Purpose**

Clean and Invalidate data and Allocation Tags in data cache by physical address to the Point of Physical Aliasing.

**Configurations**

This instruction is present only when FEAT_RME is implemented and FEAT_MTE2 is implemented. Otherwise, direct accesses to DC CIGDPAPA are UNDEFINED.

**Attributes**

DC CIGDPAPA is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>NS</th>
<th>RES0</th>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>62</td>
<td>52 51 32 31 0</td>
</tr>
</tbody>
</table>

NS, bit [63]

Together with the NSE field, this field specifies the target physical address space.

<table>
<thead>
<tr>
<th>NSE</th>
<th>NS</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b0</td>
<td>Secure.</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>Non-secure.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>Root.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>Realm.</td>
</tr>
</tbody>
</table>

NSE, bit [62]

Together with the NS field, this field specifies the target physical address space.

For a description of the values derived by evaluating NS and NSE together, see DC CIGDPAPA.NS.

**Bits [61:52]**

Reserved, RES0.

**Bits [51:0]**

Physical address to use. No alignment restrictions apply to this PA.

**Executing DC CIGDPAPA instruction**

- This instruction is not subject to any translation, permission checks, or granule protection checks.
- This instruction affects all caches in the Outer Shareable shareability domain.
- This instruction has the same ordering, observability, and completion behavior as VA-based cache maintenance instructions issued to the Outer Shareable shareability domain.
Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC CIGDPAPA, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b0111</td>
<td>0b1110</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  DC_CIGDPAPA(X[t, 64]);
C5.3.10 DC CIGDSW, Clean and Invalidate of Data and Allocation Tags by Set/Way

The DC CIGDSW characteristics are:

**Purpose**
Clean and Invalidate data and Allocation Tags in data cache by set/way.

**Configurations**
This instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC CIGDSW are UNDEFINED.

**Attributes**
DC CIGDSW is a 64-bit System instruction.

**Field descriptions**

Bits [63:32]
Reserved, RES0.

SetWay, bits [31:4]
Contains two fields:
- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = \( \log_2(\text{ASSOCIATIVITY}) \), \( L = \log_2(\text{LINELEN}) \), \( B = (L + S) \), \( S = \log_2(\text{NSETS}) \).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]
Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]
Reserved, RES0.

**Executing DC CIGDSW instruction**

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.
Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC CIGDSW, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1110</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TSW == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCISW == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
  end if;
elsif PSTATE.EL == EL2 then
  AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
  AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
C5.3.11 DC CIGDVAC, Clean and Invalidate of Data and Allocation Tags by VA to PoC

The DC CIGDVAC characteristics are:

**Purpose**

Clean and Invalidate data and Allocation Tags in data cache by address to Point of Coherency.

**Configurations**

This instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CIGDVAC are UNDEFINED.

**Attributes**

DC CIGDVAC is a 64-bit System instruction.

**Field descriptions**

![Field descriptions diagram]

**Bits [63:0]**

Virtual address to use. No alignment restrictions apply to this VA.

**Executing DC CIGDVAC instruction**

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D7-5056.

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission fault.

For more information, see Permission fault.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC CIGDVAC, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b011</td>
<td>0b0111</td>
<td>0b1110</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
    HFGITR_EL2.DCCIVAC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
    endif
elsif PSTATE_EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE_EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
C5.3.12 DC CIGSW, Clean and Invalidate of Allocation Tags by Set/Way

The DC CIGSW characteristics are:

**Purpose**

Clean and Invalidate Allocation Tags in data cache by set/way.

**Configurations**

This instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC CIGSW are UNDEFINED.

**Attributes**

DC CIGSW is a 64-bit System instruction.

**Field descriptions**

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log2(ASSOCIATIVITY), L = Log2(LINELEN), B = (L + S), S = Log2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

**Executing DC CIGSW instruction**

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.
Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC CIGSW, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1110</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TSW == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() & ((HaveEL(EL3)) | SCR_EL3.FGTEn == '1') & HFGITR_EL2.DCCISW == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
C5.3.13  DC CIGVAC, Clean and Invalidate of Allocation Tags by VA to PoC

The DC CIGVAC characteristics are:

Purpose

Clean and Invalidate Allocation Tags in data cache by address to Point of Coherency.

Configurations

This instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC CIGVAC are UNDEFINED.

Attributes

DC CIGVAC is a 64-bit System instruction.

Field descriptions

Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CIGVAC instruction

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC) on page D7-5056.

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission fault.

For more information, see Permission fault.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\begin{array}{cccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b01 & 0b011 & 0b1111 & 0b1110 & 0b011 \\
\end{array}
\]

if PSTATE.EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    endif
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  endif
else
  if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCIVAC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  endif
endif
AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TCP_P == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && ((HaveEL(EL3) && SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCIVAC == '1') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
    end
elsif PSTATE_EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE_EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_CleanInvalidate, CacheOpScope_PoC);
C5.3.14 DC CIPAPA, Data or unified Cache line Clean and Invalidate by PA to PoPA

The DC CIPAPA characteristics are:

**Purpose**

Clean and Invalidate data cache by physical address to the Point of Physical Aliasing.

**Configurations**

This instruction is present only when FEAT_RME is implemented. Otherwise, direct accesses to DC CIPAPA are UNDEFINED.

**Attributes**

DC CIPAPA is a 64-bit System instruction.

**Field descriptions**

NS, bit [63]

Together with the NSE field, this field specifies the target physical address space.

<table>
<thead>
<tr>
<th>NSE</th>
<th>NS</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b0</td>
<td>Secure.</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>Non-secure.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>Root.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>Realm.</td>
</tr>
</tbody>
</table>

NSE, bit [62]

Together with the NS field, this field specifies the target physical address space.

For a description of the values derived by evaluating NS and NSE together, see DC CIPAPA.NS.

Bits [61:52]

Reserved, RES0.

Bits [51:0]

Physical address to use. No alignment restrictions apply to this PA.

**Executing DC CIPAPA instruction**

- This instruction is not subject to any translation, permission checks, or granule protection checks.
- This instruction affects all caches in the Outer Shareable shareability domain.
- This instruction has the same ordering, observability, and completion behavior as VA-based cache maintenance instructions issued to the Outer Shareable shareability domain.

Accesses to this instruction use the following encodings in the System instruction encoding space:
**DC CIPAPA, <Xt>**

```
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  DC_CIPAPA(X[t, 64]);
```

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b0111</td>
<td>0b1110</td>
<td>0b001</td>
</tr>
</tbody>
</table>
C5.3.15 DC CISW, Data or unified Cache line Clean and Invalidate by Set/Way

The DC CISW characteristics are:

**Purpose**
Clean and Invalidate data cache by set/way.

**Configurations**
AArch64 System register DC CISW performs the same function as AArch32 System register DCCISW.

**Attributes**
DC CISW is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:32]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetWay bits [31:4]</td>
<td></td>
</tr>
<tr>
<td>Contains two fields:</td>
<td></td>
</tr>
<tr>
<td>• Way, bits[31:32-A], the number of the way to operate on.</td>
<td></td>
</tr>
<tr>
<td>• Set, bits[B-1:L], the number of the set to operate on.</td>
<td></td>
</tr>
<tr>
<td>Bits[L-1:4] are RES0.</td>
<td></td>
</tr>
<tr>
<td>A = ( \log_2(\text{ASSOCIATIVITY}) ), L = ( \log_2(\text{LINELEN}) ), B = ( (L + S) ), S = ( \log_2(\text{NSETS}) ).</td>
<td></td>
</tr>
<tr>
<td>ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.</td>
<td></td>
</tr>
</tbody>
</table>

| Level, bits [3:1] |
| Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache. |

| Bit [0] |
| Reserved, RES0. |

**Executing DC CISW instruction**

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.
Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC CISW, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1110</td>
<td>0b010</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCISW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
    end
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
```

```plaintext
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCISW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
    end
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
```
DC CIVAC, Data or unified Cache line Clean and Invalidate by VA to PoC

The DC CIVAC characteristics are:

**Purpose**

Clean and Invalidate data cache by address to Point of Coherency.

**Configurations**

AArch64 System register DC CIVAC performs the same function as AArch32 System register DCCIMVAC.

**Attributes**

DC CIVAC is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>63 32</th>
<th>31 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual address to use</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Executing DC CIVAC instruction**

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D7-5056.

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission fault.

For more information, see *Permission fault*.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC CIVAC, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b011</td>
<td>0b111</td>
<td>0b1110</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
if EL2Enabled() && HCR_EL2.TGE == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGITR_EL2.DCCIVAC == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
AArch64.SystemAccessTrap(EL2, 0x18);
endif
endif
endif
AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.TCP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!(HaveEL(EL3)) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCIVAC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC);
  endif
else
  AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC);
endif
C5.3.17  DC CSW, Data or unified Cache line Clean by Set/Way

The DC CSW characteristics are:

**Purpose**
Clean data cache by set/way.

**Configurations**
AArch64 System register DC CSW performs the same function as AArch32 System register DCCSW.

**Attributes**
DC CSW is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>31:4</td>
<td>SetWay</td>
</tr>
<tr>
<td>3:1</td>
<td>Level</td>
</tr>
<tr>
<td>0</td>
<td>Bit</td>
</tr>
</tbody>
</table>

**Bits [63:32]**
Reserved, RES0.

**SetWay, bits [31:4]**
Contains two fields:
- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[31:4], the number of the set to operate on.

**Level, bits [3:1]**
Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

**Bit [0]**
Reserved, RES0.

**Executing DC CSW instruction**

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.
Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CSW, <Xt>
```

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TSW == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCSW == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL2 then
  AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
  AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_SetWay);
C5.3.18 DC CVAC, Data or unified Cache line Clean by VA to PoC

The DC CVAC characteristics are:

**Purpose**
Clean data cache by address to Point of Coherency.

**Configurations**
AArch64 System register DC CVAC performs the same function as AArch32 System register DCCMVAC.

**Attributes**
DC CVAC is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:0]</td>
<td>Virtual address to use</td>
</tr>
</tbody>
</table>

**Virtual address to use**
No alignment restrictions apply to this VA.

**Executing DC CVAC instruction**

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission fault, subject to the constraints described in Permission fault.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D7-5056.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC CVAC, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b011</td>
<td>0b0111</td>
<td>0b1010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
    if !(EL2Enabled() & HCR_EL2.<E2H,TGE> == '11') & SCTLR_EL1.UCI == '0' then
        if EL2Enabled() & HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        endif
    else
        AArch64.SystemAccessTrap(EL2, 0x18);
    endif
else
    AArch64.SystemAccessTrap(EL2, 0x18);
endif
```
if EL2Enabled() && HCR_EL2.TPCP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoC);
C5.3.19   DC CVADP, Data or unified Cache line Clean by VA to PoDP

The DC CVADP characteristics are:

**Purpose**

Clean data cache by address to Point of Deep Persistence.

If the memory system does not identify a Point of Deep Persistence, then this instruction behaves as a DC CVAP.

**Configurations**

This instruction is present only when FEAT_DPB2 is implemented. Otherwise, direct accesses to DC CVADP are UNDEFINED.

**Attributes**

DC CVADP is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:0]</td>
<td>Virtual address to use. No alignment restrictions apply to this VA.</td>
</tr>
</tbody>
</table>

**Executing DC CVADP instruction**

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission fault, see Permission fault.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC) on page D7-5056.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC CVADP, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b011</td>
<td>0b0111</td>
<td>0b1101</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```
if PSTATE_EL == EL0 then
    if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        endif
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        HFGITR_EL2.DCCVADP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.SystemAccessTrap(EL1, 0x18);
    endif
```
AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP);
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.TPCE == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVADP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP);
  endif
elsif PSTATE_EL == EL2 then
  AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP);
elsif PSTATE_EL == EL3 then
  AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP);
C5.3.20 DC CVAP, Data or unified Cache line Clean by VA to PoP

The DC CVAP characteristics are:

**Purpose**

Clean data cache by address to Point of Persistence.

If the memory system does not identify a Point of Persistence, then this instruction behaves as a DC CVAC.

**Configurations**

This instruction is present only when FEAT_DPB is implemented. Otherwise, direct accesses to DC CVAP are UNDEFINED.

**Attributes**

DC CVAP is a 64-bit System instruction.

**Field descriptions**

Bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

**Executing DC CVAP instruction**

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission fault, see permission fault.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC) on page D7-5056.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC CVAP, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b011</td>
<td>0b0111</td>
<td>0b1100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.UCI == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL1, 0x18);
  elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPCP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL1, 0x18);
AArch64.DC(X[t], 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoP);

elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.TPCP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.DC(X[t], 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoP);
  endif
endif
C5.3.21 DC CVAU, Data or unified Cache line Clean by VA to PoU

The DC CVAU characteristics are:

**Purpose**

Clean data cache by address to Point of Unification.

**Configurations**

AArch64 System register DC CVAU performs the same function as AArch32 System register DCCMVAU.

**Attributes**

DC CVAU is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Virtual address to use</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th>Virtual address to use</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

** Bits [63:0] **

Virtual address to use. No alignment restrictions apply to this VA.

** Executing DC CVAU instruction **

If EL0 access is enabled, when executed at EL0, this instruction requires read access permission to the VA, otherwise it generates a Permission fault, subject to the constraints described in Permission fault.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC) on page D7-5056.

Accesses to this instruction use the following encodings in the System instruction encoding space:

** DC CVAU, <Xt> **

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b01</td>
<td>0b011</td>
<td>0b101</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !(EL2Enabled() & HCR_EL2.<E2H,TGE> == '1') & SCTLR_EL1.UCI == '0' then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() & HCR_EL2.<E2H,TGE> != '1' & HCR_EL2.TPU == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & HCR_EL2.<E2H,TGE> != '1' & HCR_EL2.TOCU == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & HCR_EL2.<E2H,TGE> != '1' & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFGITR_EL2.DCCVAU == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '1' & SCTLR_EL2.UCI == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoU);
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TPU == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.TOCU == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCCVAU == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoU);
  endif
elsif PSTATE.EL == EL2 then
  AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoU);
elsif PSTATE.EL == EL3 then
  AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoU);
C5.3.22  DC GVA, Data Cache set Allocation Tag by VA

The DC GVA characteristics are:

**Purpose**
Write a value to the Allocation Tags of a naturally aligned block of N bytes, where the size of N is identified in DCZID_EL0. The Allocation Tag used is determined by the input address.

**Configurations**
This instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC GVA are **UNDEFINED**.

**Attributes**
DC GVA is a 64-bit System instruction.

**Field descriptions**

Boosts [63:0]
Virtual address to use. There is no alignment restriction on the address within the block of N bytes that is used.

**Executing DC GVA instruction**
When this instruction is executed, it can generate memory faults or watchpoints which are prioritized in the same way as other memory-related faults or watchpoints. If a synchronous data abort fault or a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is not set.

If the memory region being modified is any type of Device memory, this instruction can give an alignment fault that is prioritized in the same way as other alignment faults that are determined by the memory type.

This instruction applies to Normal memory regardless of cacheability attributes.

This instruction behaves as a set of stores to each Allocation Tag within the block being accessed, and so it:

- Generates a Permission fault if the translation system does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instructions.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC GVA, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b011</td>
<td>0b0111</td>
<td>0b0100</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.DZE == '0' then
if EL2Enabled() && HCR_EL2.TGE == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
  AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TDZ == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
  HFGITR_EL2.DCZVA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.DZE == '0' then
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  AArch64.MemZero(X[t, 64], CacheType_Tag);
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TDZ == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCZVA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.MemZero(X[t, 64], CacheType_Tag);
  endif
elsif PSTATE.EL == EL2 then
  AArch64.MemZero(X[t, 64], CacheType_Tag);
elsif PSTATE.EL == EL3 then
  AArch64.MemZero(X[t, 64], CacheType_Tag);
C5.3.23 DC GZVA, Data Cache set Allocation Tags and Zero by VA

The DC GZVA characteristics are:

**Purpose**

Zero data and write a value to the Allocation Tags of a naturally aligned block of N bytes, where the size of N is identified in DCZID_EL0. The Allocation Tag used is determined by the input address.

**Configurations**

This instruction is present only when FEAT_MTE is implemented. Otherwise, direct accesses to DC GZVA are UNDEFINED.

**Attributes**

DC GZVA is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Virtual address to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>63 32</td>
<td></td>
</tr>
<tr>
<td>31 0</td>
<td></td>
</tr>
</tbody>
</table>

**Executing DC GZVA instruction**

When this instruction is executed, it can generate memory faults or watchpoints which are prioritized in the same way as other memory-related faults or watchpoints. If a synchronous data abort fault or a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is not set.

If the memory region being zeroed is any type of Device memory, this instruction can give an alignment fault which is prioritized in the same way as other alignment faults that are determined by the memory type.

This instruction applies to Normal memory regardless of cacheability attributes.

This instruction behaves as a set of Stores to each byte and Allocation tag within the block being accessed, and so it:

- Generates a Permission fault if the translation system does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instructions.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC GZVA, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b011</td>
<td>0b0111</td>
<td>0b0100</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
 if !(EL2Enabled() & HCR_EL2.<E2H,TGE> == '11') & SCTLR_EL1.DZE == '0' then
   if EL2Enabled() & HCR_EL2.TGE == '1' then
     AArch64.SystemAccessTrap(EL2, 0x18);
else
  AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TDZ == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
  HFGITR_EL2.DCZVA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.DZE == '0' then
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  AArch64.MemZero(X[t, 64], CacheType_Data_Tag);
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TDZ == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCZVA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.MemZero(X[t, 64], CacheType_Data_Tag);
  end if;
elsif PSTATE.EL == EL2 then
  AArch64.MemZero(X[t, 64], CacheType_Data_Tag);
elsif PSTATE.EL == EL3 then
  AArch64.MemZero(X[t, 64], CacheType_Data_Tag);
C5.3.24 DC IGDSW, Invalidate of Data and Allocation Tags by Set/Way

The DC IGDSW characteristics are:

**Purpose**

Invalidate data and Allocation Tags in data cache by set/way.

**Configurations**

This instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC IGDSW are UNDEFINED.

**Attributes**

DC IGDSW is a 64-bit System instruction.

**Field descriptions**

Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log2(ASSOCIATIVITY), L = Log2(LINELEN), B = (L + S), S = Log2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

**Executing DC IGDSW instruction**

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.
Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
DC \text{ IGDSW, } <Xt> 
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0110</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TSW == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCISW == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL2 then
  AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
  AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
C5.3.25 DC IGDVAC, Invalidate of Data and Allocation Tags by VA to PoC

The DC IGDVAC characteristics are:

**Purpose**

Invalidate data and Allocation Tags in data cache by address to Point of Coherency.

**Configurations**

This instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC IGDVAC are UNDEFINED.

**Attributes**

DC IGDVAC is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Virtual address to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>63 32</td>
<td></td>
</tr>
<tr>
<td>31 0</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:0]**

Virtual address to use. No alignment restrictions apply to this VA.

**Executing DC IGDVAC instruction**

When the instruction is executed, it can generate a watchpoint, which is prioritized in the same way as other watchpoints. If a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 1.

This instruction requires write access permission to the VA, otherwise it generates a Permission fault, subject to the constraints described in Permission fault.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D7-5056.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC IGDVAC, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0110</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFGITR_EL2.DCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
elsif PSTATE_EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
C5.3.26 DC IGSW, Invalidate of Allocation Tags by Set/Way

The DC IGSW characteristics are:

**Purpose**
Invalidate Allocation Tags in data cache by set/way.

**Configurations**
This instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC IGSW are UNDEFINED.

**Attributes**
DC IGSW is a 64-bit System instruction.

**Field descriptions**

![Field Descriptions Diagram]

**Bits [63:32]**
Reserved, RES0.

**SetWay, bits [31:4]**
Contains two fields:
- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

A = Log2(ASSOCIATIVITY), L = Log2(LINELEN), B = (L + S), S = Log2(NSETS).

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

**Level, bits [3:1]**
Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

**Bit [0]**
Reserved, RES0.

**Executing DC IGSW instruction**
If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.
Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC IGSW, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0110</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TSW == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && !HaveEL(EL3) || SCR_EL3.FGTEn == '1' && HFGITR_EL2.DCISW == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
  endif
elsif PSTATE.EL == EL2 then
  AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
  AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Invalidate, CacheOpScope_SetWay);
C5.3.27 DC IGVAC, Invalidate of Allocation Tags by VA to PoC

The DC IGVAC characteristics are:

**Purpose**

Invalidate Allocation Tags in data cache by address to Point of Coherency.

**Configurations**

This instruction is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to DC IGVAC are UNDEFINED.

**Attributes**

DC IGVAC is a 64-bit System instruction.

**Field descriptions**

**Bits [63:0]**

Virtual address to use. No alignment restrictions apply to this VA.

**Executing DC IGVAC instruction**

When the instruction is executed, it can generate a watchpoint, which is prioritized in the same way as other watchpoints. If a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 1.

This instruction requires write access permission to the VA, otherwise it generates a Permission fault, subject to the constraints described in Permission fault.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *The data cache maintenance instruction (DC)* on page D7-5056.

Accesses to this instruction use the following encodings in the System instruction encoding space:

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0110</td>
<td>0b01</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() &\& HCR_EL2.TPCP == '1' then
    AArch64.SystemAccessTrap(El2, 0x18);
  elsif EL2Enabled() &\& (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &\& HFGITR_EL2.DCIVAC == '1' then
    AArch64.SystemAccessTrap(El2, 0x18);
  else
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
  AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
elsif PSTATE_EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Tag, CacheOp_Invalidate, CacheOpScope_PoC);
C5.3.28 DC ISW, Data or unified Cache line Invalidate by Set/Way

The DC ISW characteristics are:

**Purpose**

Invalidate data cache by set/way.

When `FEAT_MTE2` is implemented, this instruction might invalidate Allocation Tags from caches. When it invalidates Allocation Tags from caches, it also cleans them.

**Configurations**

AArch64 System register DC ISW performs the same function as AArch32 System register `DCISW`.

**Attributes**

DC ISW is a 64-bit System instruction.

**Field descriptions**

Bits [63:32]

Reserved, RES0.

**SetWay, bits [31:4]**

Contains two fields:

- **Way**, bits [31:32-A], the number of the way to operate on.
- **Set**, bits [B-1:L], the number of the set to operate on.

Bits [L-1:4] are RES0.

\[ A = \log_2(\text{ASSOCIATIVITY}), \quad L = \log_2(\text{LINELEN}), \quad B = (L + S), \quad S = \log_2(\text{NSETS}). \]

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

**Level, bits [3:1]**

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

**Bit [0]**

Reserved, RES0.

**Executing DC ISW instruction**

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is **CONSTRAINED UNPREDICTABLE** and one of the following occurs:

- The instruction is **UNDEFINED**.
- The instruction performs cache maintenance on one of:
  - No cache lines.
— A single arbitrary cache line.
— Multiple arbitrary cache lines.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[ DC\ ISW, \langle Xt \rangle \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b111</td>
<td>0b110</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TSW == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCISW == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Invalidate, CacheOpScope_SetWay);
  endif
elsif PSTATE.EL == EL2 then
  AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Invalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
  AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Invalidate, CacheOpScope_SetWay);
C5.3.29 DC IVAC, Data or unified Cache line Invalidate by VA to PoC

The DC IVAC characteristics are:

**Purpose**

Invalidate data cache by address to Point of Coherency.

When FEAT_MTE2 is implemented, this instruction might invalidate Allocation Tags from caches. When it invalidates Allocation Tags from caches, it also cleans them.

**Configurations**

AArch64 System register DC IVAC performs the same function as AArch32 System register DCIMVAC.

**Attributes**

DC IVAC is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Virtual address to use</th>
</tr>
</thead>
</table>

Virtual address to use. No alignment restrictions apply to this VA.

**Executing DC IVAC instruction**

When the instruction is executed, it can generate a watchpoint, which is prioritized in the same way as other watchpoints. If a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 1.

This instruction requires write access permission to the VA, otherwise it generates a Permission fault, subject to the constraints described in Permission fault.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The data cache maintenance instruction (DC) on page D7-5056.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**DC IVAC, <xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0110</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCIVAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC);
    elsif PSTATE.EL == EL2 then
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC);
elsif PSTATE_EL == EL3 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC);
C5.3.30 DC ZVA, Data Cache Zero by VA

The DC ZVA characteristics are:

Purpose
Zero data cache by address. Zeroes a naturally aligned block of N bytes, where the size of N is identified in DCZID_EL0.

Configurations
There are no configuration notes.

Attributes
DC ZVA is a 64-bit System instruction.

Field descriptions

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Virtual address to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>63 32</td>
<td>Virtual address to use</td>
</tr>
<tr>
<td>31 0</td>
<td>Virtual address to use</td>
</tr>
</tbody>
</table>

Bits [63:0]
Virtual address to use. There is no alignment restriction on the address within the block of N bytes that is used.

Executing DC ZVA instruction

When this instruction is executed, it can generate memory faults or watchpoints which are prioritized in the same way as other memory-related faults or watchpoints. If a synchronous data abort fault or a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 0.

If the memory region being zeroed is any type of Device memory, this instruction can give an Alignment fault which is prioritized in the same way as other Alignment faults that are determined by the memory type.

This instruction applies to Normal memory regardless of cacheability attributes.

This instruction behaves as a set of Stores to each byte within the block being accessed, and so it:

• Generates a Permission fault if the translation system does not permit writes to the locations.
• Requires the same considerations for ordering and the management of coherency as any other store instructions.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{DC ZVA, } <\text{Xt}> \\
\begin{array}{cccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
\hline
0b01 & 0b011 & 0b0111 & 0b0100 & 0b001 \\
\end{array}
\]

if PSTATE_EL == EL0 then
  if !(EL2Enabled() & HCR_EL2.<E2H,TGE> == '11') & SCTL_EL1.DZE == '0' then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else

AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TDZ == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> || !HaveEL(EL3) || SCR_EL3.FGTEn == '1' &&
  HFGITR_EL2.DCZVA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.DZE == '0' then
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  AArch64.MemZero(X[t, 64], CacheType_Data);
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TDZ == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DCZVA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.MemZero(X[t, 64], CacheType_Data);
  elsif PSTATE.EL == EL2 then
    AArch64.MemZero(X[t, 64], CacheType_Data);
elsif PSTATE.EL == EL3 then
  AArch64.MemZero(X[t, 64], CacheType_Data);
C5.3.31 IC IALLU, Instruction Cache Invalidate All to PoU

The IC IALLU characteristics are:

**Purpose**
Invalidate all instruction caches of the PE executing the instruction to the Point of Unification.

**Configurations**
AArch64 System register IC IALLU performs the same function as AArch32 System register ICIALLU.

**Attributes**
IC IALLU is a 64-bit System instruction.

**Field descriptions**
This instruction has no applicable fields.
The value in the register specified by <Xt> is ignored.

**Executing IC IALLU instruction**
The Rt field should be set to 0b11111. If the Rt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to 0b11111.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**IC IALLU{, <Xt>]**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0101</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TOCU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.ICIALLU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
        AArch64.IC(CacheOpScope_ALLUIS);
    else
        AArch64.IC(CacheOpScope_ALLUI);
    endif
elsif PSTATE.EL == EL2 then
    AArch64.IC(CacheOpScope_ALLU);
elsif PSTATE.EL == EL3 then
    AArch64.IC(CacheOpScope_ALLU);
C5.3.32 IC IALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable

The IC IALLUIS characteristics are:

**Purpose**

Invalidate all instruction caches in the Inner Shareable domain of the PE executing the instruction to the Point of Unification.

**Configurations**

AArch64 System register IC IALLUIS performs the same function as AArch32 System register ICIALLUIS.

**Attributes**

IC IALLUIS is a 64-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

**Executing IC IALLUIS instruction**

The Rt field should be set to 0b11111. If the Rt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to 0b11111.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{IC IALLUIS}(, <Xt>)
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TPU == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TICAB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.ICIALLUIS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.IC(CacheOpScope_ALLUIS);
elsif PSTATE.EL == EL2 then
  AArch64.IC(CacheOpScope_ALLUIS);
elsif PSTATE.EL == EL3 then
  AArch64.IC(CacheOpScope_ALLUIS);
C5.3.33 IC IVAU, Instruction Cache line Invalidate by VA to PoU

The IC IVAU characteristics are:

**Purpose**
Invalidate instruction cache by address to Point of Unification.

**Configurations**
AArch64 System register IC IVAU performs the same function as AArch32 System register ICIMVAU.

**Attributes**
IC IVAU is a 64-bit System instruction.

**Field descriptions**

Abbreviation: IC IVAU

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>Virtual address to use. No alignment restrictions apply to this VA.</td>
</tr>
<tr>
<td>31-0</td>
<td>Virtual address to use</td>
</tr>
</tbody>
</table>

**Bits [63:0]**
Virtual address to use. No alignment restrictions apply to this VA.

**Executing IC IVAU instruction**

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see The instruction cache maintenance instruction (IC) on page D7-5056.

If EL0 access is enabled, when executed at EL0, if this instruction does not have read access permission to the VA, it is IMPLEMENTATION DEFINED whether it generates a Permission fault.

For more information, see Permission fault.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**IC IVAU(, <Xt>)**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b01</td>
<td>0b11</td>
<td>0b10</td>
<td>0b01</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '1') && SCTLR_EL1.UCI == '0' then
if EL2Enabled() && HCR_EL2.TGE == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TPU == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TOCU == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.HTEn == '1') &&
HFGITR_EL2.ICIVAU == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.UCI == '0' then

AArch64.SystemAccessTrap(El2, 0x18);
else
  AArch64.IC(X[t, 64], CacheOpScope_PoU);
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TPU == '1' then
    AArch64.SystemAccessTrap(El2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TOCU == '1' then
    AArch64.SystemAccessTrap(El2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.ICIVAU == '1' then
    AArch64.SystemAccessTrap(El2, 0x18);
  else
    AArch64.IC(X[t, 64], CacheOpScope_PoU);
  end if
elsif PSTATE.EL == EL2 then
  AArch64.IC(X[t, 64], CacheOpScope_PoU);
elsif PSTATE.EL == EL3 then
  AArch64.IC(X[t, 64], CacheOpScope_PoU);
C5.4 A64 System instructions for address translation

This section lists the A64 System instructions for address translation.
C5.4.1  AT S12E0R, Address Translate Stages 1 and 2 EL0 Read

The AT S12E0R characteristics are:

**Purpose**

Performs stage 1 and 2 address translations from EL0, with permissions as if reading from the given virtual address from EL0, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.\{E2H, TGE\} is not \{1, 1\}, the EL1&0 translation regime.
  - If HCR_EL2.\{E2H, TGE\} is \{1, 1\}, the EL2&0 translation regime.
- Otherwise, the EL1&0 translation regime.

**Configurations**

There are no configuration notes.

**Attributes**

AT S12E0R is a 64-bit System instruction.

**Field descriptions**

![Input address for translation](image)

**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

**Executing AT S12E0R instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**AT S12E0R, \:<Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        if HCR_EL2.\:<E2H,TGE> == '11' || HCR_EL2.\:<DC,VM> == '00' then
            AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Read);
        else
            AArch64.AT(X[t, 64], TranslationStage_12, EL0, ATAccess_Read);
    end
endl
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Read);
    elsif EL2Enabled() && (HCR_EL2.<E2H,TGE> == '11' || HCR_EL2.<DC,VM> == '00') then
        AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Read);
    else
        AArch64.AT(X[t, 64], TranslationStage_12, EL0, ATAccess_Read);
    endif
endif
C5.4.2 AT S12E0W, Address Translate Stages 1 and 2 EL0 Write

The AT S12E0W characteristics are:

**Purpose**

Performs stage 1 and 2 address translations from EL0, with permissions as if writing to the given virtual address from EL0, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime.
- Otherwise, the EL1&0 translation regime.

**Configurations**

There are no configuration notes.

**Attributes**

AT S12E0W is a 64-bit System instruction.

**Field descriptions**

Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

**Executing AT S12E0W instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**AT S12E0W, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' || HCR_EL2.<DC,VM> == '00' then
    AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Write);
  else
    AArch64.AT(X[t, 64], TranslationStage_12, EL0, ATAccess_Write);
elsif PSTATE_EL == EL3 then
  if !EL2Enabled() then
    AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Write);
  elsif EL2Enabled() && (HCR_EL2.<E2H,TGE> == '11' || HCR_EL2.<DC,VM> == '00') then
    AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Write);
  else
    AArch64.AT(X[t, 64], TranslationStage_12, EL0, ATAccess_Write);
C5.4.3 AT S12E1R, Address Translate Stages 1 and 2 EL1 Read

The AT S12E1R characteristics are:

**Purpose**

Performs stage 1 and 2 address translation, with permissions as if reading from the given virtual address from EL1, or from EL2 if the Effective value of HCR_EL2.{E2H, TGE} is \{1, 1\}, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not \{1, 1\}, the EL1&0 translation regime, accessed from EL1.
  - If HCR_EL2.{E2H, TGE} is \{1, 1\}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

**Configurations**

There are no configuration notes.

**Attributes**

AT S12E1R is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Input address for translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>Input address for translation</td>
</tr>
</tbody>
</table>

**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

**Executing AT S12E1R instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**AT S12E1R, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b111</td>
<td>0b100</td>
<td>0b100</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0
  UNDEFINED;
elsif PSTATE.EL == EL1
  if EL2Enabled() && HCR_EL2.NV == '1'
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2
  if HCR_EL2.<E2H,TGE> == '11' || HCR_EL2.<DC,VM> == '00'
```
AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Read);
else
  AArch64.AT(X[t, 64], TranslationStage_12, EL1, ATAccess_Read);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Read);
  elsif EL2Enabled() && (HCR_EL2.<E2H,TGE> == '11' || HCR_EL2.<DC,VM> == '00') then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Read);
  else
    AArch64.AT(X[t, 64], TranslationStage_12, EL1, ATAccess_Read);
C5.4.4 AT S12E1W, Address Translate Stages 1 and 2 EL1 Write

The AT S12E1W characteristics are:

**Purpose**

Performs stage 1 and 2 address translation, with permissions as if writing to the given virtual address from EL1, or from EL2 if the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

**Configurations**

There are no configuration notes.

**Attributes**

AT S12E1W is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Input address for translation</th>
</tr>
</thead>
</table>

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

**Executing AT S12E1W instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**AT S12E1W, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' || HCR_EL2.<DC,VM> == '00' then
        AT S12E1W, <Xt> // For EL2
    else
        UNDEFINED;
```

C5-848 Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. Non-Confidential  ARM DDI 0487I.a ID081822
AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Write);
else
  AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Write);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Write);
  elsif EL2Enabled() && (HCR_EL2.<E2H,TGE> == '11' || HCR_EL2.<DC,VM> == '00') then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Write);
  else
    AArch64.AT(X[t, 64], TranslationStage_12, EL1, ATAccess_Write);
  fi
fi
C5.4.5   AT S1E0R, Address Translate Stage 1 EL0 Read

The AT S1E0R characteristics are:

**Purpose**

Performs stage 1 address translation from EL0, with permissions as if reading from the given virtual address from EL0, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime.
- Otherwise, the EL1&0 translation regime.

**Configurations**

There are no configuration notes.

**Attributes**

AT S1E0R is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Input address for translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:32]</td>
<td></td>
</tr>
<tr>
<td>[31:0]</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

**Executing AT S1E0R instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**AT S1E0R, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b011</td>
<td>0b100</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then

UNDEFINED;

elsif PSTATE.EL == EL1 then

if EL2Enabled() && HCR_EL2.AT == '1' then

AArch64.SystemAccessTrap(EL2, 0x18);

elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.ATS1E0R == '1' then

AArch64.SystemAccessTrap(EL2, 0x18);

else

AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Read);

elsif PSTATE.EL == EL2 then

AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Read);
elsif PSTATE.EL == EL3 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Read);
C5.4.6 AT S1E0W, Address Translate Stage 1 EL0 Write

The AT S1E0W characteristics are:

Purpose

Performs stage 1 address translation from EL0, with permissions as if writing to the given virtual address from EL0, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime.
- Otherwise, the EL1&0 translation regime.

Configurations

There are no configuration notes.

Attributes

AT S1E0W is a 64-bit System instruction.

Field descriptions

| 63 | Input address for translation |
| 32 |                              |
| 31 | Input address for translation |
| 0  |                              |

Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E0W instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

AT S1E0W, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.AT == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.ATS1E0W == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Write);
elsif PSTATE.EL == EL2 then
  AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Write);
elsif PSTATE.EL == EL3 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Write);
### C5.4.7 AT S1E1R, Address Translate Stage 1 EL1 Read

The AT S1E1R characteristics are:

**Purpose**

Performs stage 1 address translation, with permissions as if reading from the given virtual address from EL1, or from EL2 if the Effective value of HCR_EL2.{E2H, TGE} is \{1, 1\}, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not \{1, 1\}, the EL1&0 translation regime, accessed from EL1.
  - If HCR_EL2.{E2H, TGE} is \{1, 1\}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

**Configurations**

There are no configuration notes.

**Attributes**

AT S1E1R is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Input address for translation</td>
</tr>
</tbody>
</table>

**Executing AT S1E1R instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**AT S1E1R, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.AT == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.ATS1E1R == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Read);
elsif PSTATE.EI == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Read);
elsif PSTATE.EI == EL3 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Read);
C5.4.8  AT S1E1RP, Address Translate Stage 1 EL1 Read PAN

The AT S1E1RP characteristics are:

**Purpose**

Performs a stage 1 address translation, where the value of PSTATE.PAN determines if a read from a location will generate a Permission fault for a privileged access, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

**Configurations**

This instruction is present only when FEAT_PAN2 is implemented. Otherwise, direct accesses to AT S1E1RP are UNDEFINED.

**Attributes**

AT S1E1RP is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63–32</td>
<td>Input address for translation</td>
</tr>
<tr>
<td>31–20</td>
<td>Input address for translation</td>
</tr>
</tbody>
</table>

**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

**Executing AT S1E1RP instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**AT S1E1RP, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.AT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.ATS1E1RP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.SystemAccessTrap(EL2, 0x18);
```
AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_ReadPAN);
elsif PSTATE.EL == EL2 then
  AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_ReadPAN);
elsif PSTATE.EL == EL3 then
  AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_ReadPAN);
C5.4.9 AT S1E1W, Address Translate Stage 1 EL1 Write

The AT S1E1W characteristics are:

**Purpose**

Performs stage 1 address translation, with permissions as if writing to the given virtual address from EL1, or from EL2 if the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the EL1&0 translation regime, accessed from EL1.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

**Configurations**

There are no configuration notes.

**Attributes**

AT S1E1W is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Input address for translation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input address for translation. The resulting address can be read from the PAR_EL1. If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

**Executing AT S1E1W instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**AT S1E1W, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.AT == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) | SCR_EL3.FGTEn == '1') && HFGITR_EL2.ATS1E1W == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Write);
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Write);
elsif PSTATE.EL == EL3 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Write);
C5.4.10 AT S1E1WP, Address Translate Stage 1 EL1 Write PAN

The AT S1E1WP characteristics are:

**Purpose**

Performs a stage 1 address translation, where the value of PSTATE.PAN determines if a write to a location will generate a Permission fault for a privileged access, using the following translation regime:

- When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS:
  - If HCR_EL2.{E2H, TGE} is not \{1, 1\}, the EL1&0 translation regime, accessed from EL1.
  - If HCR_EL2.{E2H, TGE} is \{1, 1\}, the EL2&0 translation regime, accessed from EL2.
- Otherwise, the EL1&0 translation regime, accessed from EL1.

**Configurations**

This instruction is present only when FEAT_PAN2 is implemented. Otherwise, direct accesses to AT S1E1WP are UNDEFINED.

**Attributes**

AT S1E1WP is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:0]</td>
<td>Input address for translation</td>
</tr>
</tbody>
</table>

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is res0.

** Executing AT S1E1WP instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**AT S1E1WP, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.AT == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCITR_EL2.ATS1E1WP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
else
AArch64.AT(\{t, 64\}, TranslationStage_1, EL1, ATAccess_WritePAN);

elsif PSTATE.EL == EL2 then
    AArch64.AT(\{t, 64\}, TranslationStage_1, EL1, ATAccess_WritePAN);
elsif PSTATE.EL == EL3 then
    AArch64.AT(\{t, 64\}, TranslationStage_1, EL1, ATAccess_WritePAN);
C5.4.11  AT S1E2R, Address Translate Stage 1 EL2 Read

The AT S1E2R characteristics are:

**Purpose**

Performs stage 1 address translation as defined for EL2, with permissions as if reading from the given virtual address.

**Configurations**

There are no configuration notes.

**Attributes**

AT S1E2R is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Input address for translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>Input address for translation</td>
</tr>
</tbody>
</table>

**Executes AT S1E2R instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**AT S1E2R, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
Undefined;
elsif PSTATE.EL == EL1 then
if EL2Enabled() && HCR_EL2.NV == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
Undefined;
elsif PSTATE.EL == EL2 then
AArch64.AT(X[t, 64], TranslationStage_1, EL2, ATAccess_Read);
elsif PSTATE.EL == EL3 then
if !EL2Enabled() then
Undefined;
else
AArch64.AT(X[t, 64], TranslationStage_1, EL2, ATAccess_Read);
C5.4.12 AT S1E2W, Address Translate Stage 1 EL2 Write

The AT S1E2W characteristics are:

Purpose
Performs stage 1 address translation as defined for EL2, with permissions as if writing to the given virtual address.

Configurations
There are no configuration notes.

Attributes
AT S1E2W is a 64-bit System instruction.

Field descriptions

Bits [63:0]
Input address for translation. The resulting address can be read from the PAR_EL1.
If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E2W instruction
Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{AT S1E2W, } <\text{Xt}> \quad \begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b01 & 0b100 & 0b0111 & 0b1000 & 0b001 \\
\end{array}
\]

if PSTATE.EL == EL0 then
  UNDEFINED;
elsiif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsiif PSTATE.EL == EL2 then
  AArch64.AT(X[t, 64], TranslationStage_1, EL2, ATAccess_Write);
elsiif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    UNDEFINED;
elsielse
  AArch64.AT(X[t, 64], TranslationStage_1, EL2, ATAccess_Write);
C5.4.13 AT S1E3R, Address Translate Stage 1 EL3 Read

The AT S1E3R characteristics are:

Purpose

Performs stage 1 address translation as defined for EL3, with permissions as if reading from the given virtual address.

Configurations

There are no configuration notes.

Attributes

AT S1E3R is a 64-bit System instruction.

Field descriptions

Input address for translation

Bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1. If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

Executing AT S1E3R instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{AT S1E3R, } \langle\text{Xt}\rangle
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL3, ATAccess_Read);
C5.4.14 AT S1E3W, Address Translate Stage 1 EL3 Write

The AT S1E3W characteristics are:

**Purpose**

Performs stage 1 address translation as defined for EL3, with permissions as if writing to the given virtual address.

**Configurations**

There are no configuration notes.

**Attributes**

AT S1E3W is a 64-bit System instruction.

**Field descriptions**

---

**Bits [63:0]**

Input address for translation. The resulting address can be read from the PAR_EL1.

If the address translation instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then VA[63:32] is RES0.

**Executing AT S1E3W instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**AT S1E3W, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL3, ATAccess_Write);
C5.5 A64 System instructions for TLB maintenance

This section lists the A64 System instructions for TLB maintenance.

For more information about these instructions see TLB maintenance instructions on page D8-5201. In particular, for the full description of the scope of each instruction see TLB maintenance instruction scope on page D8-5206.
C5.5.1 TLBI ALLE1, TLBI ALLE1NXS, TLB Invalidate All, EL1

The TLBI ALLE1 and TLBI ALLE1NXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate an address using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.

The invalidation applies to entries with any VMID.

The invalidation only applies to the PE that executes this System instruction.

**Note**

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI ALLE1, TLBI ALLE1NXS is a 64-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

**Executing TLBI ALLE1, TLBI ALLE1NXS instruction**

The Rt field should be set to 0b11111. If the Rt field is not set to 0b11111, it is CONstrained UNpredictable whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to 0b11111.
Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI ALLE1{, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_NSH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_NSH, TLBI_AllAttr);

**TLBI ALLE1NXS{, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0111</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_NSH, TLBI_ExcludeXS);
elsif PSTATE.EL == EL3 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_NSH, TLBI_ExcludeXS);
C5.5.2 TLBI ALLE1IS, TLBI ALLE1ISNXS, TLB Invalidate All, EL1, Inner Shareable

The TLBI ALLE1IS and TLBI ALLE1ISNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - \( \text{SCR}_{\text{EL3}} \cdot \{\text{NSE, NS}\} \) is \( \{0, 0\} \) and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - \( \text{SCR}_{\text{EL3}} \cdot \{\text{NSE, NS}\} \) is \( \{0, 1\} \) and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.
  - \( \text{SCR}_{\text{EL3}} \cdot \{\text{NSE, NS}\} \) is \( \{1, 1\} \) and the entry would be required to translate an address using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - \( \text{SCR}_{\text{EL3}} \).NS is 0 and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - \( \text{SCR}_{\text{EL3}} \).NS is 1 and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.

The invalidation applies to entries with any VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

--- Note ---

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

---

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI ALLE1IS, TLBI ALLE1ISNXS is a 64-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by \(<Xt>\) is ignored.

**Executing TLBI ALLE1IS, TLBI ALLE1ISNXS instruction**

The Rt field should be set to \(0b11111\). If the Rt field is not set to \(0b11111\), it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
The instruction behaves as if the Rt field is set to 0b11111.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI ALLE1IS{, <Xt>]**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_ISH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_ISH, TLBI_AllAttr);

**TLBI ALLE1ISNXS{, <Xt>]**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0011</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_ISH, TLBI_ExcludeXS);
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_ISH, TLBI_ExcludeXS);
C5.5.3 TLBI ALLE1OS, TLBI ALLE1OSNXS, TLB Invalidate All, EL1, Outer Shareable

The TLBI ALLE1OS and TLBI ALLE1OSNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is \{0, 0\} and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is \{0, 1\} and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is \{1, 1\} and the entry would be required to translate an address using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate an address using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate an address using the Non-secure EL1&0 translation regime.

The invalidation applies to entries with any VMID.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

**Note**

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI ALLE1OS, TLBI ALLE1OSNXS are UNDEFINED.

**Attributes**

TLBI ALLE1OS, TLBI ALLE1OSNXS is a 64-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

**Executing TLBI ALLE1OS, TLBI ALLE1OSNXS instruction**

The Rt field should be set to 0b11111. If the Rt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
The instruction behaves as if the Rt field is set to 0b11111.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI ALLE1OS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0001</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_OSH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_OSH, TLBI_AllAttr);

**TLBI ALLE1OSNXS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
UNDEFINED;
elsif PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_OSH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_OSH, TLBI_AllAttr);
C5.5.4   TLBI ALLE2, TLBI ALLE2NXS, TLB Invalidate All, EL2

The TLBI ALLE2 and TLBI ALLE2NXS characteristics are:

**Purpose**

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any address using the Realm EL2&0 or EL2 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.

The invalidation only applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.
Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI ALLE2, TLBI ALLE2NXS is a 64-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

**Executing TLBI ALLE2, TLBI ALLE2NXS instruction**

The Rt field should be set to 0b11111. If the Rt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to 0b11111.

Accesses to this instruction use the following encodings in the System instruction encoding space:
TLB ALLE2{, <Xt>}\)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Shareability_NSH, TLBI_AllAttr);
  else
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Shareability_NSH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    UNDEFINED;
  elsif HCR_EL2.E2H == '1' then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Shareability_NSH, TLBI_AllAttr);
  else
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Shareability_NSH, TLBI_AllAttr);

TLB ALLE2NXS{, <Xt>}\)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0111</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Shareability_NSH, TLBI_ExcludeXS);
  else
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Shareability_NSH, TLBI_ExcludeXS);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    UNDEFINED;
  elsif HCR_EL2.E2H == '1' then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Shareability_NSH, TLBI_ExcludeXS);
  else
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Shareability_NSH, TLBI_ExcludeXS);
The A64 System Instruction Class

C5.5 A64 System instructions for TLB maintenance

C5.5.5 TLBI ALLE2IS, TLBI ALLE2ISNXS, TLB Invalidate All, EL2, Inner Shareable

The TLBI ALLE2IS and TLBI ALLE2ISNXS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any address using the Realm EL2&0 or EL2 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.
Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configurations

There are no configuration notes.

Attributes

TLBI ALLE2IS, TLBI ALLE2ISNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE2IS, TLBI ALLE2ISNXS instruction

The Rt field should be set to 0b11111. If the Rt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to 0b11111.

Accesses to this instruction use the following encodings in the System instruction encoding space:
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

**TLBI ALLE2IS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Shareability_ISH, TLBI_AllAttr);
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Shareability_ISH, TLBI_AllAttr);
elsif PSTATE_EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '1' then
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Shareability_ISH, TLBI_AllAttr);
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Shareability_ISH, TLBI_AllAttr);

**TLBI ALLE2ISNXS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Shareability_ISH, TLBI_ExcludeXS);
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Shareability_ISH, TLBI_ExcludeXS);
elsif PSTATE_EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '1' then
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Shareability_ISH, TLBI_ExcludeXS);
    else
        AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Shareability_ISH, TLBI_ExcludeXS);
C5.5.6 TLBI ALLE2OS, TLBI ALLE2OSNXS, TLB Invalidate All, EL2, Outer Shareable

The TLBI ALLE2OS and TLBI ALLE2OSNXS characteristics are:

**Purpose**

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any address using the Realm EL2&0 or EL2 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any address using the Secure EL2&0 or EL2 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any address using the Non-secure EL2&0 or EL2 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined. Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI ALLE2OS, TLBI ALLE2OSNXS are UNDEFINED.

**Attributes**

TLBI ALLE2OS, TLBI ALLE2OSNXS is a 64-bit System instruction.

**Field descriptions**

This instruction has no applicable fields. The value in the register specified by <Xt> is ignored.

**Executing TLBI ALLE2OS, TLBI ALLE2OSNXS instruction**

The Rt field should be set to 0b11111. If the Rt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to 0b11111.

Accesses to this instruction use the following encodings in the System instruction encoding space:
**TLBI ALLE2OS**, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Shareability_OSH, TLBI_AllAttr);
  else
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Shareability_OSH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    UNDEFINED;
  elsif HCR_EL2.E2H == '1' then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Shareability_OSH, TLBI_AllAttr);
  else
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Shareability_OSH, TLBI_AllAttr);

**TLBI ALLE2OSNXS**, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Shareability_OSH, TLBI_ExcludeXS);
  else
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Shareability_OSH, TLBI_ExcludeXS);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    UNDEFINED;
  elsif HCR_EL2.E2H == '1' then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL20, Shareability_OSH, TLBI_ExcludeXS);
  else
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Shareability_OSH, TLBI_ExcludeXS);
C5.5.7 TLBI ALLE3, TLBI ALLE3NXS, TLB Invalidate All, EL3

The TLBI ALLE3 and TLBI ALLE3NXS characteristics are:

**Purpose**

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI ALLE3, TLBI ALLE3NXS is a 64-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by \(<Xt>\) is ignored.

**Executing TLBI ALLE3, TLBI ALLE3NXS instruction**

The Rt field should be set to \(0b11111\). If the Rt field is not set to \(0b11111\), it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to \(0b11111\).

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{TLBI ALLE3}\{, <Xt>\}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Shareability_NSH, TLBI_AllAttr);
TLBI ALLE3NXS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b10</td>
<td>0b1001</td>
<td>0b0111</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Shareability_NSX, TLBIExcludeXS);
C5.5.8 TLBI ALLE3IS, TLBI ALLE3ISNXS, TLB Invalidate All, EL3, Inner Shareable

The TLBI ALLE3IS and TLBI ALLE3ISNXS characteristics are:

**Purpose**

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI ALLE3IS, TLBI ALLE3ISNXS is a 64-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

**Executing TLBI ALLE3IS, TLBI ALLE3ISNXS instruction**

The Rt field should be set to 0b11111. If the Rt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to 0b11111.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{TLBI ALLE3IS}\{, <Xt>\}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
UNDEFINED;
elsif PSTATE.EL == EL2 then
UNDEFINED;
elsif PSTATE.EL == EL3 then
AArch64.TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Shareability_ISH, TLBI_AllAttr);
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

TLBI ALLE3ISNXS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1001</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Shareability_ISH, TLBI_ExcludeXS);
C5.5.9 TLBI ALLE3OS, TLBI ALLE3OSNXS, TLB Invalidate All, EL3, Outer Shareable

The TLBI ALLE3OS and TLBI ALLE3OSNXS characteristics are:

**Purpose**

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI ALLE3OS, TLBI ALLE3OSNXS are UNDEFINED.

**Attributes**

TLBI ALLE3OS, TLBI ALLE3OSNXS is a 64-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

**Executing TLBI ALLE3OS, TLBI ALLE3OSNXS instruction**

The Rt field should be set to 0b11111. If the Rt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to 0b11111.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI ALLE3OS(<Xt>)**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1000</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
UNDEFINED;
elsif PSTATE.EL == EL2 then
UNDEFINED;
elsif PSTATE.EL == EL3 then
AAAch64.TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Shareability_OSH, TLBI_AllAttr);
TLBI ALLE3OSNXS{, <X>[r]}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented( FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL3, Shareability_OSH, TLBI_ExcludeXS);
C5.5.10 TLBI ASIDE1, TLBI ASIDE1NXS, TLB Invalidate by ASID, EL1

The TLBI ASIDE1 and TLBI ASIDE1NXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- When EL2 is implemented and enabled in the current Security state:
  - If HCR_EL2.{E2H, TGE} is not \{1, 1\}, the entry would be used with the current VMID and would be required to translate an address using the EL1&0 translation regime for the Security state.
  - If HCR_EL2.{E2H, TGE} is \{1, 1\}, the entry would be required to translate an address using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate an address using the EL1&0 translation regime for the Security state.

The invalidation applies to the PE that executes this System instruction. If FEAT_XS is implemented, the nXS variant of this System instruction is defined. Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete. The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI ASIDE1, TLBI ASIDE1NXS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>ASID, bits [63:48]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this System instruction. If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits [47:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>
Executing TLBI ASIDE1, TLBI ASIDE1NXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI ASIDE1{, <Xt>)**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGEn == '1') & HFGITR_EL2.TLBIASIDE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) & IsFeatureImplemented(FEAT_HCX) & HCR_EL2.FnXS == '1' then
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_AlAllAttr, X[t, 64]);
        end if
        if IsFeatureImplemented(FEAT_XS) & IsFeatureImplemented(FEAT_HCX) & IsHCRXEL2Enabled() & HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AlAllAttr, X[t, 64]);
        end if
    else
        if PSTATE.EL == EL2 then
            if HCR_EL2.<E2H,TGE> == '11' then
                AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBI_AlAllAttr, X[t, 64]);
            else
                AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AlAllAttr, X[t, 64]);
            end if
        elsif PSTATE.EL == EL3 then
            if HCR_EL2.<E2H,TGE> == '11' then
                AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBI_AlAllAttr, X[t, 64]);
            else
                AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AlAllAttr, X[t, 64]);
            end if
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBI_AlAllAttr, X[t, 64]);
        end if
    end if
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBI_AlAllAttr, X[t, 64]);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AlAllAttr, X[t, 64]);
    end if
elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBI_AlAllAttr, X[t, 64]);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AlAllAttr, X[t, 64]);
    end if
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if !IsFeatureImplemented(FEAT_XS) then
            UNDEFINED;
        elsif PSTATE.EL == EL0 then
            UNDEFINED;
        elsif PSTATE.EL == EL1 then
            if EL2Enabled() & HCR_EL2.TTLB == '1' then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                if !IsFeatureImplemented(FEAT_XS) then
                    UNDEFINED;
                elsif PSTATE.EL == EL0 then
                    UNDEFINED;
                elsif PSTATE.EL == EL1 then
                    if EL2Enabled() & HCR_EL2.TTLB == '1' then
                        AArch64.SystemAccessTrap(EL2, 0x18);
                    else
                        if IsFeatureImplemented(FEAT_XS) & IsFeatureImplemented(FEAT_HCX) & IsHCRXEL2Enabled() & HCRX_EL2.FnXS == '1' then
                            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_ExcludeXS, X[t, 64]);
                        else
                            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_AlAllAttr, X[t, 64]);
                        end if
                    end if
                else
                    if IsFeatureImplemented(FEAT_XS) & IsFeatureImplemented(FEAT_HCX) & IsHCRXEL2Enabled() & HCRX_EL2.FnXS == '1' then
                        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_ExcludeXS, X[t, 64]);
                    else
                        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AlAllAttr, X[t, 64]);
                    end if
                end if
            else
                AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AlAllAttr, X[t, 64]);
            end if
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AlAllAttr, X[t, 64]);
        end if
    end if
end if

**TLBI ASIDE1NXS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0111</td>
<td>0b010</td>
</tr>
</tbody>
</table>
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTXS == '0') && HFGITR_EL2.TLBIASIDE1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.FB == '1' then
    AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_ExcludeXS, X[t, 64]);
else
    AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_ExcludeXS, X[t, 64]);

C5.5.11 TLBI ASIDE1IS, TLBI ASIDE1ISNXS, TLB Invalidate by ASID, EL1, Inner Shareable

The TLBI ASIDE1IS and TLBI ASIDE1ISNXS characteristics are:

Purpose
Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- When EL2 is implemented and enabled in the current Security state:
  - If HCR_EL2.{E2H, TGE} is not \{1, 1\}, the entry would be used with the current VMID and would be required to translate an address using the EL1&0 translation regime for the Security state.
  - If HCR_EL2.{E2H, TGE} is \{1, 1\}, the entry would be required to translate an address using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate an address using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configurations
There are no configuration notes.

Attributes
TLBI ASIDE1IS, TLBI ASIDE1ISNXS is a 64-bit System instruction.

Field descriptions

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:48]</td>
<td>ASID</td>
</tr>
<tr>
<td>[47:0]</td>
<td>RES0</td>
</tr>
</tbody>
</table>

ASID, bits [63:48]
ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this System instruction.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.
Bits [47:0]

Reserved, RES0.

**Executing TLBI ASIDE1IS, TLBI ASIDE1ISNXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI ASIDE1IS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.TLBIASIDE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HC) && IsHCRXEL2Enabled() &&
            HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
                TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
                TLBI_AllAttr, X[t, 64]);
        end
    end
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Shareability_ISH, TLBI_AllAttr, X[t, 64]);
    end
elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Shareability_ISH, TLBI_AllAttr, X[t, 64]);
    end

**TLBI ASIDE1ISNXS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEl(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&
(!IsHCRXEl2Enabled()) || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIASIDE1IS == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_ExcludeXS,
  X[t, 64]);
elsif PSTATE.EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
    TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_ExcludeXS,
    X[t, 64]);
  endif
elsif PSTATE.EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
    TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_ExcludeXS,
    X[t, 64]);
C5.5.12 TLBI ASIDE1OS, TLBI ASIDE1OSNXS, TLB Invalidate by ASID, EL1, Outer Shareable

The TLBI ASIDE1OS and TLBI ASIDE1OSNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- When EL2 is implemented and enabled in the current Security state:
  - If \( \text{HCR\_EL2}.\{\text{E2H, TGE}\} \) is not \( \{1, 1\} \), the entry would be used with the current VMID and would be required to translate an address using the EL1&0 translation regime for the Security state.
  - If \( \text{HCR\_EL2}.\{\text{E2H, TGE}\} \) is \( \{1, 1\} \), the entry would be required to translate an address using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate an address using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of \( \text{SCR\_EL3.NS} \) if \( \text{FEAT\_RME} \) is not implemented, or \( \text{SCR\_EL3.\{NSE, NS\}} \) if \( \text{FEAT\_RME} \) is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If \( \text{FEAT\_XS} \) is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when \( \text{FEAT\_TLBIOS} \) is implemented. Otherwise, direct accesses to TLBI ASIDE1OS, TLBI ASIDE1OSNXS are UNDEFINED.

**Attributes**

TLBI ASIDE1OS, TLBI ASIDE1OSNXS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit positions</th>
<th>Field description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:48</td>
<td>ASID</td>
</tr>
<tr>
<td>47:32</td>
<td>RES0</td>
</tr>
<tr>
<td>31:0</td>
<td>RES0</td>
</tr>
</tbody>
</table>

**ASID, bits [63:48]**

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this System instruction.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

Bits [47:0]
Reserved, RES0.

Executing TLBI ASIDE1OS, TLBI ASIDE1OSNXS instruction
Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI ASIDE1OS{, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) & IsFeatureImplemented(FEAT_HCX) & IsHCRXEL2Enabled() &
            HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH,
                TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH,
                TLBI_AllAttr, X[t, 64]);
    end
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBI_AllAttr,
            X[t, 64]);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBI_AllAttr,
            X[t, 64]);
    end
elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBI_AllAttr,
            X[t, 64]);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBI_AllAttr,
            X[t, 64]);

**TLBI ASIDE1OSNXS{, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
AArch64.SystemAccessTrap(El2, 0x18);
elsif EL2Enabled() && (HaveEl(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&
(!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIASIDE1OS == '1' then
  AArch64.SystemAccessTrap(El2, 0x18);
else
  AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBI_ExcludeXS, X[t, 64]);
  endif
elsif PSTATE.EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_ASID(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBI_ExcludeXS, X[t, 64]);
  endif
endif
else
  AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBI_ExcludeXS, X[t, 64]);
C5.5.13   TLBI IPAS2E1, TLBI IPAS2E1NXS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1

The TLBI IPAS2E1 and TLBI IPAS2E1NXS characteristics are:

**Purpose**

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see *Invalidating TLB entries from stage 2 translations* on page D8-5211.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI IPAS2E1, TLBI IPAS2E1NXS is a 64-bit System instruction.

**Field descriptions**

```
<table>
<thead>
<tr>
<th>NS</th>
<th>RES0</th>
<th>TTL</th>
<th>RES0</th>
<th>IPA[51:48]</th>
<th>IPA[47:12]</th>
<th>IPA[47:12]</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>62</td>
<td>48</td>
<td>47</td>
<td>44 43 40 39 36 35 32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

IPA[51:48]: 
IPA[47:12]: 
IPA[47:12]: 
IPA[47:12]:
NS, bit [63]

When FEAT_RME is implemented:

- When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.
  - 0b0: IPA is in the Secure IPA space.
  - 0b1: IPA is in the Non-secure IPA space.

- When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

- When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

- Not Secure. Specifies the IPA space.
  - 0b0: IPA is in the Secure IPA space.
  - 0b1: IPA is in the Non-secure IPA space.

- When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

- When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

- Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

- 0b00xx: No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

- 0b01xx: The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b01: Level 1.
  - 0b10: Level 2.
  - 0b11: Level 3.

- 0b10xx: The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b10: Level 2.
  - 0b11: Level 3.

- 0b11xx: The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01: Level 1.
  - 0b10: Level 2.
  - 0b11: Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.
Otherwise:
  Reserved, RES0.

Bits [43:40]
  Reserved, RES0.

IPA[51:48], bits [39:36]

When FEAT_LPA is implemented:
  Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:
  Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48
bits, the upper bits of this field are RES0.

When FEAT_LPA is implemented, and 52-bit addresses and a 64KB translation granule are in use,
IPA[51:48] form the upper part of the address value. Otherwise, IPA[51:48] are RES0.

Executing TLBI IPAS2E1, TLBI IPAS2E1NXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI IPAS2E1{, <Xt>}**

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0100</td>
<td>0b001</td>
</tr>
</tbody>
</table>
```

```
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any,
  TLBI_AllAttr, X[t, 64]);
elsif PSTATE_EL == EL3 then
  if !EL2Enabled() then
    return;
  else
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any,
    TLBI_AllAttr, X[t, 64]);
```

**TLBI IPAS2E1NXS{, <Xt>}**

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0100</td>
<td>0b001</td>
</tr>
</tbody>
</table>
```

```
if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
```
elif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elseif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_ExcludeXS, X[p, 64]);
elseif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_ExcludeXS, X[p, 64]);
The TLBI IPAS2E1IS and TLBI IPAS2E1ISNXS characteristics are:

**Purpose**

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see *Invalidating TLB entries from stage 2 translations* on page D8-5211.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS is a 64-bit System instruction.

**Field descriptions**
NS, bit [63]

When **FEAT_RME** is implemented:

When the instruction is executed and \( \text{SCR}_\text{EL3}.\{\text{NSE, NS}\} = \{0, 0\} \), NS selects the IPA space.
- 0b0  IPA is in the Secure IPA space.
- 0b1  IPA is in the Non-secure IPA space.

When the instruction is executed and \( \text{SCR}_\text{EL3}.\{\text{NSE, NS}\} = \{1, 1\} \), this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and \( \text{SCR}_\text{EL3}.\{\text{NSE, NS}\} = \{0, 1\} \), this field is RES0, and the instruction applies only to the Non-secure IPA space.

When **FEAT_SEL2** is implemented and **FEAT_RME** is not implemented:

Not Secure. Specifies the IPA space.
- 0b0  IPA is in the Secure IPA space.
- 0b1  IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When **FEAT_SEL2** is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When **FEAT_TTL** is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.
- 0b00xx  No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
- 0b01xx  The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00  : If **FEAT_LPA2** is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b01  : Level 1.
  - 0b10  : Level 2.
  - 0b11  : Level 3.
- 0b10xx  The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00  : Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01  : If **FEAT_LPA2** is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b10  : Level 2.
  - 0b11  : Level 3.
- 0b11xx  The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00  : Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01  : Level 1.
  - 0b10  : Level 2.
  - 0b11  : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.
Reserved, RES0.

IPA[51:48], bits [39:36]

When FEAT_LPA is implemented:
Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:
Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

When FEAT_LPA is implemented, and 52-bit addresses and a 64KB translation granule are in use, IPA[51:48] form the upper part of the address value. Otherwise, IPA[51:48] are RES0.

Executing TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI IPAS2E1IS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE_EL == EL2 then
        AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    elsif PSTATE_EL == EL3 then
        if !EL2Enabled() then
            return;
        else
            AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    TLBI IPAS2E1ISNXS{, <Xt>} |

if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    endif
elsif PSTATE_EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE_EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    endif

C5.5.15 TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable

The TLBI IPAS2E1OS and TLBI IPAS2E1OSNXS characteristics are:

**Purpose**

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see *Invalidating TLB entries from stage 2 translations* on page D8-5211.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS are undefined.

**Attributes**

TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS</td>
<td></td>
</tr>
<tr>
<td>RES0</td>
<td></td>
</tr>
<tr>
<td>TTL</td>
<td></td>
</tr>
<tr>
<td>RES0</td>
<td></td>
</tr>
<tr>
<td>IPA[51:48]</td>
<td></td>
</tr>
<tr>
<td>IPA[47:12]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IPA[47:12]</td>
</tr>
</tbody>
</table>
NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.
0b0 IPA is in the Secure IPA space.
0b1 IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.
0b0 IPA is in the Secure IPA space.
0b1 IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

0b00xx No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

0b01xx The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
0b00 : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
0b01 : Level 1.
0b10 : Level 2.
0b11 : Level 3.

0b10xx The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
0b01 : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
0b10 : Level 2.
0b11 : Level 3.

0b11xx The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
0b01 : Level 1.
0b10 : Level 2.
0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.
Otherwise:
  Reserved, RES0.

Bits [43:40]
  Reserved, RES0.

IPA[51:48], bits [39:36]
  Extension to IPA[47:12]. For more information, see IPA[47:12].

IPA[47:12], bits [35:0]
  Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.
  When FEAT_LPA is implemented, and 52-bit addresses and a 64KB translation granule are in use, IPA[51:48] form the upper part of the address value. Otherwise, IPA[51:48] are RES0.

Executing TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI IPAS2E1OS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    return;
  else
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);

**TLBI IPAS2E1OSNXS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        AArch64.TLBIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
        TLBI_ExcludeXS, X[t, 64]);

C5.5.16 TLBI IPAS2LE1, TLBI IPAS2LE1NXS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1

The TLBI IPAS2LE1 and TLBI IPAS2LE1NXS characteristics are:

**Purpose**

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see *Invalidating TLB entries from stage 2 translations* on page D8-5211.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI IPAS2LE1, TLBI IPAS2LE1NXS is a 64-bit System instruction.

**Field descriptions**

![Field descriptions diagram]
NS, bit [63]

When *FEAT_RME* is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.
- 0b0 IPA is in the Secure IPA space.
- 0b1 IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When *FEAT_SEL2* is implemented and *FEAT_RME* is not implemented:

Not Secure. Specifies the IPA space.
- 0b0 IPA is in the Secure IPA space.
- 0b1 IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When *FEAT_SEL2* is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When *FEAT_TTL* is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

- 0b00xx No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
- 0b01xx The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: If *FEAT_LPA2* is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b01: Level 1.
  - 0b10: Level 2.
  - 0b11: Level 3.
- 0b10xx The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01: If *FEAT_LPA2* is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b10: Level 2.
  - 0b11: Level 3.
- 0b11xx The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01: Level 1.
  - 0b10: Level 2.
  - 0b11: Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.
Otherwje:

Reserved, RES0.

Bits [43:40]

Reserved, RES0.

IPA[51:48], bits [39:36]

When FEAT_LPA is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

When FEAT_LPA is implemented, and 52-bit addresses and a 64KB translation granule are in use, IPA[51:48] form the upper part of the address value. Otherwise, IPA[51:48] are RES0.

Executing TLBI IPAS2LE1, TLBI IPAS2LE1NXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI IPAS2LE1{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0100</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if !EL2Enable() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
elsif PSTATE_EL == EL3 then
  if !EL2Enable() then
    return;
  else
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);

**TLBI IPAS2LE1NXS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0100</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & & HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    elsif PSTATE.EL == EL3 then
        if !EL2Enabled() then
            return;
        else
            AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        return;
    endif;
C5.5.17 TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable

The TLBI IPAS2LE1IS and TLBI IPAS2LE1ISNXS characteristics are:

**Purpose**

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is \{0, 0\} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is \{0, 1\} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is \{1, 1\} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see Invalidating TLB entries from stage 2 translations on page D8-5211.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS is a 64-bit System instruction.

**Field descriptions**

```
<table>
<thead>
<tr>
<th>NS</th>
<th>RES0</th>
<th>TTL</th>
<th>RES0</th>
<th>IPA[51:48]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IPA[47:12]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
```
NS, bit [63]

When `FEAT_RME` is implemented:
- When the instruction is executed and `SCR_EL3.{NSE, NS} == {0, 0}`, NS selects the IPA space.
  - 0b0 IPA is in the Secure IPA space.
  - 0b1 IPA is in the Non-secure IPA space.
- When the instruction is executed and `SCR_EL3.{NSE, NS} == {1, 1}`, this field is RES0, and the instruction applies only to the Realm IPA space.
- When the instruction is executed and `SCR_EL3.{NSE, NS} == {0, 1}`, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When `FEAT_SEL2` is implemented and `FEAT_RME` is not implemented:
- Not Secure. Specifies the IPA space.
  - 0b0 IPA is in the Secure IPA space.
  - 0b1 IPA is in the Non-secure IPA space.
- When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.
- When `FEAT_SEL2` is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:
- Reserved, RES0.

Bits [62:48]
Reserved, RES0.

TTL, bits [47:44]

When `FEAT_TTL` is implemented:
Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.
- 0b00xx No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
- 0b01xx The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00 : If `FEAT_LPA2` is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b01 : Level 1.
  - 0b10 : Level 2.
  - 0b11 : Level 3.
- 0b10xx The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01 : If `FEAT_LPA2` is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b10 : Level 2.
  - 0b11 : Level 3.
- 0b11xx The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01 : Level 1.
  - 0b10 : Level 2.
  - 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.
Otherwise:
Reserved, RES0.

Bits [43:40]
Reserved, RES0.

IPA[51:48], bits [39:36]

When FEAT_LPA is implemented:
Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:
Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

When FEAT_LPA is implemented, and 52-bit addresses and a 64KB translation granule are in use,IPA[51:48] form the upper part of the address value. Otherwise, IPA[51:48] are RES0.

Executing TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI IPAS2LE1IS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDONEF;elif PSTATE.EL == EL1 then
if EL2Enabled() && HCR_EL2.NV == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
UNDONEF;elif PSTATE.EL == EL2 then
AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
else if !EL2Enabled() then
return;
else
AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);

**TLBI IPAS2LE1ISNXS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
UNDONEF;elif PSTATE.EL == EL0 then
UNDONEF;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE_EL == EL2 then
        AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last,
            TLBI_IncludeXS, X[t, 64]);
    elsif PSTATE_EL == EL3 then
        if !EL2Enabled() then
            return;
        else
            AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last,
                TLBI_IncludeXS, X[t, 64]);
C5.5.18 TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable

The TLBI IPAS2LE1OS and TLBI IPAS2LE1OSNXS characteristics are:

**Purpose**

If EL2 is implemented and enabled in the current Security state, invalidate cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is \{0, 0\} and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is \{0, 1\} and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is \{1, 1\} and the entry would be required to translate the specified IPA using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate the specified IPA using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate the specified IPA using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see *Invalidating TLB entries from stage 2 translations* on page D8-5211.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS are UNDEFINED.

**Attributes**

TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS is a 64-bit System instruction.

**Field descriptions**
NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.
0b0 IPA is in the Secure IPA space.
0b1 IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.
0b0 IPA is in the Secure IPA space.
0b1 IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:
Reserved, RES0.

Bits [62:48]
Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

0b00xx No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

0b01xx The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
0b00 : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
0b01 : Level 1.
0b10 : Level 2.
0b11 : Level 3.

0b10xx The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
0b01 : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
0b10 : Level 2.
0b11 : Level 3.

0b11xx The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
0b01 : Level 1.
0b10 : Level 2.
0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.
Otherwise:

Reserved, RES0.

Bits [43:40]

Reserved, RES0.

IPA[51:48], bits [39:36]
Extension to IPA[47:12]. For more information, see IPA[47:12].

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

When FEAT_LPA is implemented, and 52-bit addresses and a 64KB translation granule are in use, IPA[51:48] form the upper part of the address value. Otherwise, IPA[51:48] are RES0.

Executing TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI IPAS2LE1OS\{, \langle Xt \rangle\}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0100</td>
<td>0b100</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() \&\& HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);

TLBI IPAS2LE1OSNXS\{, \langle Xt \rangle\}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0100</td>
<td>0b100</td>
</tr>
</tbody>
</table>

```plaintext
if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() \&\& HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
```
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last,
        TLBI_ExcludeXS, X[t, 64]);
C5.5.19 TLBI PAALL, TLB Invalidate GPT Information by PA, All Entries, Local

The TLBI PAALL characteristics are:

**Purpose**

Invalidates cached copies of GPT entries from TLBs. Details:

- The invalidation applies to TLB entries containing GPT information that relates to a physical address.
- The invalidation applies to all TLB entries containing GPT information.
- The invalidation affects only the TLBs for the PE executing the operation.

The full set of TLB maintenance instructions that invalidate cached GPT entries is: TLBI PAALL, TLBI PAALLOS, TLBI RPALOS, and TLBI RPAOS.

These instructions have the same ordering, observability, and completion behavior as all other TLBI instructions.

**Configurations**

This instruction is present only when FEAT_RME is implemented. Otherwise, direct accesses to TLBI PAALL are UNDEFINED.

**Attributes**

TLBI PAALL is a 64-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

**Executing TLBI PAALL instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{TLBI PAALL}(\#, <Xt>)
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b11</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AArch64.TLBI_PAALL(Shareability_NSH);
C5.5.20 TLBI PAALLOS, TLB Invalidate GPT Information by PA, All Entries, Outer Shareable

The TLBI PAALLOS characteristics are:

**Purpose**

Invalidates cached copies of GPT entries from TLBs. Details:

- The invalidation applies to TLB entries containing GPT information that relates to a physical address.
- The invalidation applies to all TLB entries containing GPT information.
- The invalidation affects all TLBs in the Outer Shareable domain.

The full set of TLB maintenance instructions that invalidate cached GPT entries is: TLBI PAALL, TLBI PAALLOS, TLBI RPALOS, and TLBI RPAOS.

These instructions have the same ordering, observability, and completion behavior as all other TLBI instructions.

**Configurations**

This instruction is present only when FEAT_RME is implemented. Otherwise, direct accesses to TLBI PAALLOS are UNDEFINED.

**Attributes**

TLBI PAALLOS is a 64-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

**Executing TLBI PAALLOS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{TLBI PAALLOS}(\text{<Xt>})
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1000</td>
<td>0b0001</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_PAALL(Shareability_OSH);
C5.5.21 TLBI RIPAS2E1, TLBI RIPAS2E1NXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1

The TLBI RIPAS2E1 and TLBI RIPAS2E1NXS characteristics are:

**Purpose**

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is \{0, 0\} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is \{0, 1\} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is \{1, 1\} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula \([BaseADDR <= VA < BaseADDR + ((NUM +1) \times 2^{5 \times SCALE +1}) \times Translation_Granule_Size]\).

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 0000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 0000000000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 0000000000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000000.

For more information about the architectural requirements for this System instruction, see *Invalidating TLB entries from stage 2 translations* on page D8-5211.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RIPAS2E1, TLBI RIPAS2E1NXS are UNDEFINED.
Attributes

TLBI RIPAS2E1, TLBI RIPAS2E1NXS is a 64-bit System instruction.

Field descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| NS, bit [63] | When FEAT_RME is implemented:  
When the instruction is executed and SCR_EL3.{NSE, NS} == \{0, 0\}, NS selects the IPA space.  
0b0 IPA is in the Secure IPA space.  
0b1 IPA is in the Non-secure IPA space.  
When the instruction is executed and SCR_EL3.{NSE, NS} == \{1, 1\}, this field is RES0, and the instruction applies only to the Realm IPA space.  
When the instruction is executed and SCR_EL3.{NSE, NS} == \{0, 1\}, this field is RES0, and the instruction applies only to the Non-secure IPA space.  
When FEAT_SEL2 is implemented and FEAT_RME is not implemented:  
Not Secure. Specifies the IPA space.  
0b0 IPA is in the Secure IPA space.  
0b1 IPA is in the Non-secure IPA space.  
When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.  
When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.  
Otherwise:  
Reserved, RES0. |
| Bits [62:48] | Reserved, RES0. |
0b00 Reserved.  
0b01 4K translation granule.  
0b10 16K translation granule.  
0b11 64K translation granule.  
The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries. |
| SCALE, bits [45:44] | The exponent element of the calculation that is used to produce the upper range. |
NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.
- 0b00: The entries in the range can be using any level for the translation table entries.
- 0b01: All entries to invalidate are Level 1 translation table entries.
  - If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
- 0b10: All entries to invalidate are Level 2 translation table entries.
- 0b11: All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

When FEAT_LPA2 is implemented and TCR_EL1.DS == 1:

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.
- When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
- When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.
- When using a 4KB translation granule, this field is BaseADDR[48:12].
- When using a 16KB translation granule, this field is BaseADDR[50:14].
- When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RIPAS2E1, TLBI RIPAS2E1NXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RIPAS2E1{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0100</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  AArch64.TLBI_RIPAS2(SourceStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    return;
  else
    AArch64.TLBI_RIPAS2(SourceStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
**TLBI RIPAS2E1NXS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0100</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
  endif
elsif PSTATE.EL == EL2 then
  AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBLevel_Any,
                     TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    return;
  else
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBLevel_Any,
                         TLBI_ExcludeXS, X[t, 64]);
C5.5.22  TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable

The TLBI RIPAS2E1IS and TLBI RIPAS2E1ISNXS characteristics are:

**Purpose**

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula $[\text{BaseADDR} \leq \text{VA} < \text{BaseADDR} + ((\text{NUM} + 1) * 2^{5 \times \text{SCALE} + 1} \times \text{Translation_Granule_Size})]$. The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.
- The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is **unpredictable** when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 0000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see *Invalidating TLB entries from stage 2 translations* on page D8-5211.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined. Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS are **undefined**.
Attributes

TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS is a 64-bit System instruction.

Field descriptions

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

0b0 IPA is in the Secure IPA space.

0b1 IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

0b0 IPA is in the Secure IPA space.

0b1 IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

0b00 Reserved.

0b01 4K translation granule.

0b10 16K translation granule.

0b11 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

NUM, bits [43:39]
The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]
TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.
- 0b00: The entries in the range can be using any level for the translation table entries.
- 0b01: All entries to invalidate are Level 1 translation table entries.
  If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
- 0b10: All entries to invalidate are Level 2 translation table entries.
- 0b11: All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]
When FEAT_LPA2 is implemented and TCR_EL1.DS == 1:
The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.
- When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
- When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.
Otherwise:
The starting address for the range of the maintenance instruction.
- When using a 4KB translation granule, this field is BaseADDR[48:12].
- When using a 16KB translation granule, this field is BaseADDR[50:14].
- When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS instruction
Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RIPAS2E1IS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
  end if;
else if PSTATE.EL == EL2 then
  AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
else if PSTATE.EL == EL3 then
  if !EL2Enabled() then
    return;
  else
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
end if;
### TLBI RIPAS2E1SNXS(\(<Xt>\))

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

```plaintext
if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
  endif
elsif PSTATE.EL == EL2 then
  AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
  TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    return;
  endif
  else
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
```

C5.5.23 TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable

The TLBI RIPAS2E1OS and TLBI RIPAS2E1OSNXS characteristics are:

**Purpose**

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is \{0, 0\} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is \{0, 1\} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is \{1, 1\} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula \[\text{BaseADDR} \leq VA < \text{BaseADDR} + ((\text{NUM} +1)*2^{5\text{SCALE} +1} * \text{Translation_Granule_Size})\].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is **UNPREDICTABLE** when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 0000000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 00000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 00000000000000000000000000.

For more information about the architectural requirements for this System instruction, see *Invalidating TLB entries from stage 2 translations* on page D8-5211.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS are **UNDEFINED**.
Attributes

TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS is a 64-bit System instruction.

Field descriptions

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.

0b0 IPA is in the Secure IPA space.
0b1 IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.

0b0 IPA is in the Secure IPA space.
0b1 IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

0b00 Reserved.
0b01 4K translation granule.
0b10 16K translation granule.
0b11 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.
NUM, bits [43:39]
   The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]
   TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the
   level described by the TTL hint.
   
   0b00   The entries in the range can be using any level for the translation table entries.
   0b01   All entries to invalidate are Level 1 translation table entries.
   \color{red}{\textit{If FEAT\_LPA2 is not implemented, when using a 16KB translation granule, this value}}
   \color{red}{\textit{is reserved and hardware should treat this field as 0b00.}}
   0b10   All entries to invalidate are Level 2 translation table entries.
   0b11   All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]
   \textit{When FEAT\_LPA2 is implemented and TCR\_EL1.DS == 1:}
   
   The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16]
   for all translation granules.
   
   When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
   When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

   \textit{Otherwise:}
   
   The starting address for the range of the maintenance instruction.
   
   When using a 4KB translation granule, this field is BaseADDR[48:12].
   When using a 16KB translation granule, this field is BaseADDR[50:14].
   When using a 64KB translation granule, this field is BaseADDR[52:16].

**Executing TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\textbf{TLBI RIPAS2E1OS\{, <Xt>\}}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0100</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR.EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    end if;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
    TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
        TLBI_AllAttr, X[t, 64]);
    end if;
TLBI RIPAS2E1OSNXS{, <Xt>}:

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0100</td>
<td>0b011</td>
</tr>
</tbody>
</table>
```

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    return;
  else
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
C5.5.24 TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1

The TLBI RIPAS2LE1 and TLBI RIPAS2LE1NXS characteristics are:

**Purpose**

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is \{0, 0\} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is \{0, 1\} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is \{1, 1\} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula \[\text{BaseADDR} \leq VA < \text{BaseADDR} + ((\text{NUM} +1) \cdot 2^{5\times\text{SCALE} +1}) \cdot \text{Translation_Granule_Size}\].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation only applies to the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 0000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 0000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see *Invalidating TLB entries from stage 2 translations* on page D8-5211.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS are UNDEFINED.
Attributes

TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS is a 64-bit System instruction.

Field descriptions

NS, bit [63]

When FEAT_RME is implemented:

When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.
0b0 IPA is in the Secure IPA space.
0b1 IPA is in the Non-secure IPA space.

When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the instruction applies only to the Realm IPA space.
When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:

Not Secure. Specifies the IPA space.
0b0 IPA is in the Secure IPA space.
0b1 IPA is in the Non-secure IPA space.

When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies only to the Non-secure IPA space.
When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field is RES0.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.
0b00 Reserved.
0b01 4K translation granule.
0b10 16K translation granule.
0b11 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.
NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

- 0b00: The entries in the range can be using any level for the translation table entries.
- 0b01: All entries to invalidate are Level 1 translation table entries.
- 0b10: All entries to invalidate are Level 2 translation table entries.
- 0b11: All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

When FEAT_LPA2 is implemented and TCR_EL1.DS == 1:

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

- When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
- When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

- When using a 4KB translation granule, this field is BaseADDR[48:12].
- When using a 16KB translation granule, this field is BaseADDR[50:14].
- When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RIPAS2LE1{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0100</td>
<td>0b110</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    end
elsif PSTATE_EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last, TLBIAllAttr, X[t, 64]);
elsif PSTATE_EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last, TLBIAllAttr, X[t, 64]);
```

```
TLB I RIPAS2LE1NXS{, <Xt>}
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

C5.5.25 TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable

The TLBI RIPAS2LE1IS and TLBI RIPAS2LE1ISNXS characteristics are:

**Purpose**

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is {0, 0} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {0, 1} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is {1, 1} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM+1)*2(5*SCALE+1) * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 0000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 00000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 0000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000000.

For more information about the architectural requirements for this System instruction, see **Invalidating TLB entries from stage 2 translations** on page D8-5211.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS are UNDEFINED.
Attributes

TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS is a 64-bit System instruction.

Field descriptions

NS, bit [63]
When FEAT_RME is implemented:
When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.
0b0 IPA is in the Secure IPA space.
0b1 IPA is in the Non-secure IPA space.
When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the
instruction applies only to the Realm IPA space.
When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the
instruction applies only to the Non-secure IPA space.
When FEAT_SEL2 is implemented and FEAT_RME is not implemented:
Not Secure. Specifies the IPA space.
0b0 IPA is in the Secure IPA space.
0b1 IPA is in the Non-secure IPA space.
When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies
only to the Non-secure IPA space.
When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field
is RES0.
Otherwise:
Reserved, RES0.

Bits [62:48]
Reserved, RES0.

TG, bits [47:46]
Translation granule size.
0b00 Reserved.
0b01 4K translation granule.
0b10 16K translation granule.
0b11 64K translation granule.
The instruction takes a translation granule size for the translations that are being invalidated. If the
translations used a different translation granule size than the one being specified, then the
architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]
The exponent element of the calculation that is used to produce the upper range.
NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

<table>
<thead>
<tr>
<th>TTL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>The entries in the range can be using any level for the translation table entries.</td>
</tr>
<tr>
<td>0b01</td>
<td>All entries to invalidate are Level 1 translation table entries.</td>
</tr>
<tr>
<td>0b10</td>
<td>All entries to invalidate are Level 2 translation table entries.</td>
</tr>
<tr>
<td>0b11</td>
<td>All entries to invalidate are Level 3 translation table entries.</td>
</tr>
</tbody>
</table>

BaseADDR, bits [36:0]

When FEAT_LPA2 is implemented and TCR_EL1.DS == 1:

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granularity, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granularity, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granularity, this field is BaseADDR[48:12].
When using a 16KB translation granularity, this field is BaseADDR[50:14].
When using a 64KB translation granularity, this field is BaseADDR[52:16].

Executing TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RIPAS2LE1IS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0000</td>
<td>0b110</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() & HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    end
elsif PSTATE_EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
elsif PSTATE_EL == EL3 then
    if !EL2Enabled() then
        return;
    else
        AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
```
### TLBI RIPAS2LE1ISNXS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0000</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
  endif;
elsif PSTATE.EL == EL2 then
  AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
else
  UNDEFINED;
endif;}
The TLBI RIPAS2LE1OS and TLBI RIPAS2LE1OSNXS characteristics are:

**Purpose**

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - SCR_EL3.{NSE, NS} is \{0, 0\} and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is \{0, 1\} and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
  - SCR_EL3.{NSE, NS} is \{1, 1\} and the entry would be required to translate any IPA in the specified address range using the Realm EL1&0 translation regime.
- If FEAT_RME is not implemented, one of the following applies:
  - SCR_EL3.NS is 0 and the entry would be required to translate any IPA in the specified address range using the Secure EL1&0 translation regime.
  - SCR_EL3.NS is 1 and the entry would be required to translate any IPA in the specified address range using the Non-secure EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^{5*SCALE +1} * Translation_Granule_Size)].

**Note**

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 0000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see
Invalidating TLB entries from stage 2 translations on page D8-5211.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.
Both variants perform the same invalidation, but the TLBI System instruction without the nXS
qualifier waits for all memory accesses using in-scope old translation information to complete
before it is considered complete.
The TLBI System instruction with the nXS qualifier is considered complete when the subset of these
memory accesses with XS attribute set to 0 are complete.

Configurations
This instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is
implemented. Otherwise, direct accesses to TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS are
UNDEFINED.

Attributes
TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS is a 64-bit System instruction.

Field descriptions

<table>
<thead>
<tr>
<th>NS</th>
<th>RES0</th>
<th>TG</th>
<th>NUM</th>
<th>TTL</th>
<th>BaseADDR</th>
</tr>
</thead>
</table>

NS, bit [63]

When FEAT_RME is implemented:
When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 0}, NS selects the IPA space.
0b0 IPA is in the Secure IPA space.
0b1 IPA is in the Non-secure IPA space.
When the instruction is executed and SCR_EL3.{NSE, NS} == {1, 1}, this field is RES0, and the
instruction applies only to the Realm IPA space.
When the instruction is executed and SCR_EL3.{NSE, NS} == {0, 1}, this field is RES0, and the
instruction applies only to the Non-secure IPA space.

When FEAT_SEL2 is implemented and FEAT_RME is not implemented:
Not Secure. Specifies the IPA space.
0b0 IPA is in the Secure IPA space.
0b1 IPA is in the Non-secure IPA space.
When the instruction is executed in Non-secure state, this field is RES0, and the instruction applies
only to the Non-secure IPA space.
When FEAT_SEL2 is not implemented, or if EL2 is disabled in the current Security state, this field
is RES0.

Otherwise:
Reserved, RES0.

Bits [62:48]
Reserved, RES0.
TG, bits [47:46]
Translation granule size.
0b00  Reserved.
0b01  4K translation granule.
0b10  16K translation granule.
0b11  64K translation granule.
The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]
The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]
The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]
TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.
0b00  The entries in the range can be using any level for the translation table entries.
0b01  All entries to invalidate are Level 1 translation table entries.
      If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10  All entries to invalidate are Level 2 translation table entries.
0b11  All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]
When FEAT_LPA2 is implemented and TCR_EL1.DS == 1:
The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.
When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.
Otherwise:
The starting address for the range of the maintenance instruction.
When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS instruction
Accesses to this instruction use the following encodings in the System instruction encoding space:

TLBI RIPAS2LE1OS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b10</td>
<td>0b100</td>
<td>0b0100</td>
<td>0b11</td>
</tr>
</tbody>
</table>
if EL2Enabled() \&\& \text{HCR\_EL2.NV} = '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last,
  TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    return;
  else
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH,
    TLBILevel_Last, TLBI_AllAttr, X[t, 64]);

\textbf{TLBI RIPAS2LE10SNXS{, <Xt>}}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0100</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() \&\& \text{HCR\_EL2.NV} = '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last,
  TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    return;
  else
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH,
    TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
The TLBI RPALOS characteristics are:

**Purpose**

Invalidates cached copies of GPT entries from TLBs. Details:

- The invalidation applies to TLB entries containing GPT information that relates to a physical address.
- The invalidation affects all TLBs in the Outer Shareable domain.
- Invalidates TLB entries containing GPT information from the final level of the GPT walk that relates to the supplied physical address.
- Invalidations are range-based, invalidating TLB entries starting from the address in BaseADDR, within the range as specified by SIZE.

The full set of TLB maintenance instructions that invalidate cached GPT entries is: TLBI PAALL, TLBI PAALLOS, TLBI RPALOS, and TLBI RPAOS.

These instructions have the same ordering, observability, and completion behavior as all other TLBI instructions.

**Configurations**

This instruction is present only when FEAT_RME is implemented. Otherwise, direct accesses to TLBI RPALOS are UNDEFINED.

**Attributes**

TLBI RPALOS is a 64-bit System instruction.

**Field descriptions**

```
<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>48</th>
<th>47</th>
<th>44</th>
<th>43</th>
<th>40</th>
<th>39</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>SIZE</td>
<td>RES0</td>
<td>Address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Bits [63:48]**

Reserved, RES0.

**SIZE, bits [47:44]**

Size of the range for invalidation.

If SIZE is a reserved value, no TLB entries are required to be invalidated.

- 0b0000: 4KB.
- 0b0001: 16KB.
- 0b0010: 64KB.
- 0b0011: 2MB.
- 0b0100: 32MB.
- 0b0101: 512MB.
- 0b0110: 1GB.
- 0b0111: 16GB.
- 0b1000: 64GB.
- 0b1001: 512GB.

All other values are reserved.
If SIZE gives a range smaller than the configured physical granule size in GPCCR_EL3.PGS, then the effective value of SIZE is taken to be the size configured by GPCCR_EL3.PGS.

If GPCCR_EL3.PGS is configured to a reserved value, no TLB entries are required to be invalidated.

**Bits [43:40]**

Reserved, RES0.

**Address, bits [39:0]**

The starting address for the range of the maintenance instruction.

This field is decoded with reference to the value of GPCCR_EL3.PGS to give BaseADDR as follows:

```
<table>
<thead>
<tr>
<th>GPCCR_EL3.PGS</th>
<th>BaseADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00 (4KB)</td>
<td>BaseADDR [51:12] = Xt[39:0]</td>
</tr>
<tr>
<td>0b10 (16KB)</td>
<td>BaseADDR [51:14] = Xt[39:2]</td>
</tr>
<tr>
<td>0b01 (64KB)</td>
<td>BaseADDR [51:16] = Xt[39:4]</td>
</tr>
</tbody>
</table>
```

Other bits of BaseADDR are treated as zero, to give the effective value of BaseADDR.

If the effective value of BaseADDR is not aligned to the size of the effective value of SIZE, no TLB entries are required to be invalidated.

If GPCCR_EL3.PGS is configured to a reserved value, no TLB entries are required to be invalidated.

**Executing TLBI RPALOS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RPALOS{, <Xt>}  
```

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b10</td>
<td>0b1000</td>
<td>0b0100</td>
<td>0b111</td>
</tr>
</tbody>
</table>
```

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_RPA(TLBILevel_Last, X[t, 64], Shareability_OSH);
C5.5.28 TLBI RPAOS, TLB Range Invalidate GPT Information by PA, Outer Shareable

The TLBI RPAOS characteristics are:

**Purpose**

Invalidates cached copies of GPT entries from TLBs. Details:
- The invalidation applies to TLB entries containing GPT information that relates to a physical address.
- The invalidation affects all TLBs in the Outer Shareable domain.
- Invalidates TLB entries containing GPT information from all levels of the GPT walk that relates to the supplied physical address.
- Invalidations are range-based, invalidating TLB entries starting from the address in BaseADDR, within the range as specified by SIZE.

The full set of TLB maintenance instructions that invalidate cached GPT entries is: **TLBI PAALL, TLBI PAALLOS, TLBI RPALOS, and TLBI RPAOS**.

These instructions have the same ordering, observability, and completion behavior as all other TLBI instructions.

**Configurations**

This instruction is present only when FEAT_RME is implemented. Otherwise, direct accesses to TLBI RPAOS are UNDEFINED.

**Attributes**

TLBI RPAOS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Address (31:0)</th>
<th>SIZE (47:44)</th>
<th>RES0 (43:40)</th>
<th>RES0 (63:48)</th>
</tr>
</thead>
</table>

**Bits [63:48]**

Reserved, RES0.

**SIZE, bits [47:44]**

Size of the range for invalidation.

If SIZE is a reserved value, no TLB entries are required to be invalidated.

- 0b0000  - 4KB.
- 0b0001  - 16KB.
- 0b0010  - 64KB.
- 0b0011  - 2MB.
- 0b0100  - 32MB.
- 0b0101  - 512MB.
- 0b0110  - 1GB.
- 0b0111  - 16GB.
- 0b1000  - 64GB.
- 0b1001  - 512GB.

All other values are reserved.
If SIZE gives a range smaller than the configured physical granule size in GPCCR_EL3.PGS, then the effective value of SIZE is taken to be the size configured by GPCCR_EL3.PGS.

If GPCCR_EL3.PGS is configured to a reserved value, no TLB entries are required to be invalidated.

**Bits [43:40]**

Reserved, RES0.

**Address, bits [39:0]**

The starting address for the range of the maintenance instruction.

This field is decoded with reference to the value of GPCCR_EL3.PGS to give BaseADDR as follows:

| GPCCR_EL3.PGS | BaseADDR     |
|              |             |
| 0b0 (4KB)    | BaseADDR[51:12] = Xt[39:0] |
| 0b10 (16KB)  | BaseADDR[51:14] = Xt[39:2] |
| 0b1 (64KB)   | BaseADDR[51:16] = Xt[39:4] |

Other bits of BaseADDR are treated as zero, to give the effective value of BaseADDR.

If the effective value of BaseADDR is not aligned to the size of the effective value of SIZE, no TLB entries are required to be invalidated.

If GPCCR_EL3.PGS is configured to a reserved value, no TLB entries are required to be invalidated.

**Executing TLBI RPAOS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RPAOS{, <Xt>}
```

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1000</td>
<td>0b0100</td>
<td>0b011</td>
</tr>
</tbody>
</table>

```
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_RPA(TLBILevel_Any, X[t, 64], Shareability_OSH);
```
C5.5.29   TLBI RVAAE1, TLBI RVAAE1NXS, TLB Range Invalidate by VA, All ASID, EL1

The TLBI RVAAE1 and TLBI RVAAE1NXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry is within the address range determined by the formula \[\text{BaseADDR} \leq \text{VA} < \text{BaseADDR} + ((\text{NUM} +1)*2^{(5*\text{SCALE} +1)} * \text{Translation_Granule_Size})\].
- When EL2 is implemented and enabled in the current Security state:
  - If \(\text{HCR\_EL2\_[E2H, TGE]}\) is not \(\{1, 1\}\), the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If \(\text{HCR\_EL2\_[E2H, TGE]}\) is \(\{1, 1\}\), the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of \(\text{SCR\_EL3\_NS}\) if \(\text{FEAT\_RME}\) is not implemented, or \(\text{SCR\_EL3\_[NSE, NS]}\) if \(\text{FEAT\_RME}\) is implemented.

The invalidation applies to the PE that executes this System instruction.

--- **Note** ---

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If \(\text{TTL} = 01\) and \(\text{BaseADDR}[29:12]\) is not equal to \(00000000000000000000000000\).
  - If \(\text{TTL} = 10\) and \(\text{BaseADDR}[20:12]\) is not equal to \(00000000000\).
- For the 16K translation granule:
  - If \(\text{TTL} = 10\) and \(\text{BaseADDR}[24:14]\) is not equal to \(000000000000\).
- For the 64K translation granule:
  - If \(\text{TTL} = 01\) and \(\text{BaseADDR}[41:16]\) is not equal to \(0000000000000000000000000000000000\).
  - If \(\text{TTL} = 10\) and \(\text{BaseADDR}[28:16]\) is not equal to \(0000000000000000000000000000000000\).

If \(\text{FEAT\_XS}\) is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when \(\text{FEAT\_TLBIRANGE}\) is implemented. Otherwise, direct accesses to TLBI RVAAE1, TLBI RVAAE1NXS are UNDEFINED.

**Attributes**

TLBI RVAAE1, TLBI RVAAE1NXS is a 64-bit System instruction.
Field descriptions

Bits [63:48]
Reserved, RES0.

TG, bits [47:46]
Translation granule size.
0b00 Reserved.
0b01 4K translation granule.
0b10 16K translation granule.
0b11 64K translation granule.
The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]
The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]
The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]
TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.
0b00 The entries in the range can be using any level for the translation table entries.
0b01 All entries to invalidate are Level 1 translation table entries.
If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10 All entries to invalidate are Level 2 translation table entries.
0b11 All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

When FEAT_LPA2 is implemented and TCR_EL1.DS == 1:
The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.
When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:
The starting address for the range of the maintenance instruction.
When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].
Executing TLBI RVAAE1, TLBI RVAAE1NXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RVAAE1(, <Xt>)**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0110</td>
<td>0b01</td>
</tr>
</tbody>
</table>

```
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() & HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFCITR_EL2.TLBIRVAAE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) & IsFeatureImplemented(FEAT_HCX) & HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
            TLBILevel_ANY, TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
            TLBILevel_ANY, TLBI_AllAttr, X[t, 64]);
    else
        if IsFeatureImplemented(FEAT_XS) & IsFeatureImplemented(FEAT_HCX) & HCRX_EL2.Enabled() &
            HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH,
            TLBILevel_ANY, TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH,
            TLBILevel_ANY, TLBI_AllAttr, X[t, 64]);
    endif
elsif PSTATE_EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH,
        TLBILevel_ANY, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH,
        TLBILevel_ANY, TLBI_AllAttr, X[t, 64]);
    endif
elsif PSTATE_EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH,
        TLBILevel_ANY, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH,
        TLBILevel_ANY, TLBI_AllAttr, X[t, 64]);
    endif
```

**TLBI RVAAE1NXS(, <Xt>)**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0110</td>
<td>0b01</td>
</tr>
</tbody>
</table>

```
if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() & HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&
((!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIRVAE1 == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.FB == '1' then
  AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
  TLBI_ExcludeXS, X[t, 64]);
else
  AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any,
  TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH,
    TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH,
    TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
The TLBI RVAAE1IS and TLBI RVAAE1ISNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry is within the address range determined by the formula $[\text{BaseADDR} \leq \text{VA} < \text{BaseADDR} + (\text{NUM} +1) \times 2^{(5\times \text{SCALE} +1) \times \text{Translation_Granule_Size}}]$.
- When EL2 is implemented and enabled in the current Security state:
  - If $HCR\_EL2\{E2H, TGE\}$ is not $\{1, 1\}$, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If $HCR\_EL2\{E2H, TGE\}$ is $\{1, 1\}$, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
- The Security state is indicated by the value of $SCR\_EL3\{\text{NS, NS}\}$ if FEAT_RME is not implemented, or $SCR\_EL3\{\text{NSE, NS}\}$ if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

**Note**

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if $SCR\_EL3\{\text{EEL2}\}==1$, then:

- A PE with $SCR\_EL3\{\text{EEL2}\}==1$ is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with $SCR\_EL3\{\text{EEL2}\}==0$.
- A PE with $SCR\_EL3\{\text{EEL2}\}==0$ is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with $SCR\_EL3\{\text{EEL2}\}==1$.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

**Note**

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If $TTL==01$ and $\text{BaseADDR}[29:12]$ is not equal to $000000000000000000$.
  - If $TTL==10$ and $\text{BaseADDR}[20:12]$ is not equal to $0000000000$.
- For the 16K translation granule:
  - If $TTL==10$ and $\text{BaseADDR}[24:14]$ is not equal to $000000000000$.
- For the 64K translation granule:
  - If $TTL==01$ and $\text{BaseADDR}[41:16]$ is not equal to $00000000000000000000000000$. 
If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configurations

This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAAE1IS, TLBI RVAAE1ISNXS are UNDEFINED.

Attributes

TLBI RVAAE1IS, TLBI RVAAE1ISNXS is a 64-bit System instruction.

Field descriptions

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:48</td>
<td>RES0</td>
</tr>
<tr>
<td>47:46</td>
<td>TG</td>
</tr>
<tr>
<td>45:44</td>
<td>SCALE</td>
</tr>
<tr>
<td>43:39</td>
<td>NUM</td>
</tr>
<tr>
<td>38:37</td>
<td>TTL</td>
</tr>
<tr>
<td>36:32</td>
<td>BaseADDR</td>
</tr>
</tbody>
</table>

**Bits [63:48]**

Reserved, RES0.

**TG, bits [47:46]**

Translation granule size.

0b00  Reserved.

0b01  4K translation granule.

0b10  16K translation granule.

0b11  64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

**SCALE, bits [45:44]**

The exponent element of the calculation that is used to produce the upper range.

**NUM, bits [43:39]**

The base element of the calculation that is used to produce the upper range.

**TTL, bits [38:37]**

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

0b00  The entries in the range can be using any level for the translation table entries.

0b01  All entries to invalidate are Level 1 translation table entries.

If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.

0b10  All entries to invalidate are Level 2 translation table entries.

0b11  All entries to invalidate are Level 3 translation table entries.
BaseADDR, bits [36:0]

When FEAT_LPA2 is implemented and TCR_EL1.DS == 1:

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:
The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAAE1IS, TLBI RVAAE1ISNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RVAAE1IS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b0000</td>
<td>0b1000</td>
<td>0b0010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
    UNDEFINED;
elseif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.TLBIRVAAE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
            HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
                TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
        elseif PSTATE_EL == EL1 then
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
                TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
        endif
    endif
elseif PSTATE_EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
            TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    elseif PSTATE_EL == EL3 then
        if HCR_EL2.<E2H,TGE> == '11' then
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
                TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
                TLBI_AllAttr, X[t, 64]);
        endif
    endif
elseif PSTATE_EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
            TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
            TLBI_AllAttr, X[t, 64]);
    endif
endif
```
ARM DDI 0487I.a
Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TLBI RVAAE1ISNXS( <Xt> )

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&
    (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIRVAAE1IS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
    TLBI_ExcludeXS, X[<t, 64>]);
  elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
      AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
      TLBILevel_Any, TLBI_ExcludeXS, X[<t, 64>]);
    else
      AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
      TLBI_ExcludeXS, X[<t, 64>]);
    elsif PSTATE.EL == EL3 then
      if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
        TLBILevel_Any, TLBI_ExcludeXS, X[<t, 64>]);
      else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
        TLBI_ExcludeXS, X[<t, 64>]);
      end
    end
end
C5.5.31 TLBI RVAAE1OS, TLBI RVAAE1OSNXS, TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable

The TLBI RVAAE1OS and TLBI RVAAE1OSNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry is within the address range determined by the formula \([\text{BaseADDR} \leq \text{VA} < \text{BaseADDR} + ((\text{NUM} +1)\times2^{(5\times\text{SCALE} +1)} \times \text{Translation_Granule_Size})]\).
- When EL2 is implemented and enabled in the current Security state:
  - If \(\text{HCR\_EL2.\{E2H, TGE}\}}\) is not \{1, 1\}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If \(\text{HCR\_EL2.\{E2H, TGE\}}\) is \{1, 1\}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of \(\text{SCR\_EL3.NS}\) if \(\text{FEAT\_RME}\) is not implemented, or \(\text{SCR\_EL3.\{NSE, NS\}}\) if \(\text{FEAT\_RME}\) is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

—— Note ———

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if \(\text{SCR\_EL3.EEL2} == 1\), then:

- A PE with \(\text{SCR\_EL3.EEL2} == 1\) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \(\text{SCR\_EL3.EEL2} == 0\).
- A PE with \(\text{SCR\_EL3.EEL2} == 0\) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \(\text{SCR\_EL3.EEL2} == 1\).
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

—— Note ———

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

The range of addresses invalidated is **UNPREDICTABLE** when:

- For the 4K translation granule:
  - If \(\text{TTL} == 01\) and \(\text{BaseADDR[29:12]}\) is not equal to \(000000000000000000000000\).
  - If \(\text{TTL} == 10\) and \(\text{BaseADDR[20:12]}\) is not equal to \(00000000\).
- For the 16K translation granule:
  - If \(\text{TTL} == 10\) and \(\text{BaseADDR[24:14]}\) is not equal to \(00000000000\).
- For the 64K translation granule:
  - If \(\text{TTL} == 01\) and \(\text{BaseADDR[41:16]}\) is not equal to \(00000000000000000000000000\).
If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configurations

This instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RVAAE10S, TLBI RVAAE10SNXS are UNDEFINED.

Attributes

TLBI RVAAE10S, TLBI RVAAE10SNXS is a 64-bit System instruction.

Field descriptions

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

0b00  Reserved.

0b01  4K translation granule.

0b10  16K translation granule.

0b11  64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

0b00  The entries in the range can be using any level for the translation table entries.

0b01  All entries to invalidate are Level 1 translation table entries.

If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.

0b10  All entries to invalidate are Level 2 translation table entries.
0b11 All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

When FEAT_LPA2 is implemented and TCR_EL1.DS == 1:

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAAE1OS, TLBI RVAAE1OSNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RVAAE1OS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0101</td>
<td>0b011</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (SCR_EL3.FGTEn == '1') && HFGITR_EL2.TLBIRVAAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HC) && IsHCRXEL2Enabled() &&
            HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH,
                TLBLevelAny, TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH,
                TLBLevelAny, TLBI_AllAttr, X[t, 64]);
        elsif PSTATE_EL == EL2 then
            if HCR_EL2.<E2H,TGE> == '1' then
                AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,
                    TLBLevelAny, TLBI_AllAttr, X[t, 64]);
            else
                AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,
                    TLBLevelAny, TLBI_AllAttr, X[t, 64]);
            elsif PSTATE_EL == EL3 then
                if HCR_EL2.<E2H,TGE> == '1' then
                    AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,
                        TLBLevelAny, TLBI_AllAttr, X[t, 64]);
                else
                    AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,
                        TLBLevelAny, TLBI_AllAttr, X[t, 64]);
                
```
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

TLBI RVAAE10SNX$<Xt>$

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0101</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&
        (!IsHCREL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIRVAAE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
            TLBIExcludeXS, X[t, 64]);
    endif
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,
            TLBILevel_Any, TLBIExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
            TLBIExcludeXS, X[t, 64]);
    endif
elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,
            TLBILevel_Any, TLBIExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
            TLBIExcludeXS, X[t, 64]);
    endif
else
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
        TLBIExcludeXS, X[t, 64]);
end
C5.5.32 TLBI RVAALE1, TLBI RVAALE1NXS, TLB Range Invalidate by VA, All ASID, Last level, EL1

The TLBI RVAALE1 and TLBI RVAALE1NXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry is within the address range determined by the formula \[ \text{BaseADDR} \leq VA < \text{BaseADDR} + ((\text{NUM} + 1) \times 2^{5\times \text{SCALE} + 1} \times \text{Translation Granule Size}) \].
- When EL2 is implemented and enabled in the current Security state:
  - If \( \text{HCR\_EL2.} \{\text{E2H, TGE}\} \) is not \{1, 1\}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If \( \text{HCR\_EL2.} \{\text{E2H, TGE}\} \) is \{1, 1\}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of \( \text{SCR\_EL3.} \{\text{NSE, NS}\} \) if \( \text{FEAT\_RME} \) is not implemented, or \( \text{SCR\_EL3.} \{\text{NSE, NS}\} \) if \( \text{FEAT\_RME} \) is implemented.

The invalidation applies to the PE that executes this System instruction.

--- **Note** ---

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If \( \text{TTL==01} \) and \( \text{BaseADDR[29:12]} \) is not equal to 0000000000000000.
  - If \( \text{TTL==10} \) and \( \text{BaseADDR[20:12]} \) is not equal to 00000000.
- For the 16K translation granule:
  - If \( \text{TTL==10} \) and \( \text{BaseADDR[24:14]} \) is not equal to 000000000000.
- For the 64K translation granule:
  - If \( \text{TTL==01} \) and \( \text{BaseADDR[41:16]} \) is not equal to 00000000000000000000.
  - If \( \text{TTL==10} \) and \( \text{BaseADDR[28:16]} \) is not equal to 000000000000.

If \( \text{FEAT\_XS} \) is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when \( \text{FEAT\_TLBIRANGE} \) is implemented. Otherwise, direct accesses to TLBI RVAALE1, TLBI RVAALE1NXS are UNDEFINED.

**Attributes**

TLBI RVAALE1, TLBI RVAALE1NXS is a 64-bit System instruction.
Field descriptions

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:48]</td>
<td>Reserved, RES0</td>
<td>0b00, 0b01, 0b10, 0b11</td>
</tr>
<tr>
<td>[47:46]</td>
<td>Translation granule size</td>
<td>0b00, 0b01, 0b10, 0b11</td>
</tr>
<tr>
<td>[45:44]</td>
<td>SCALE, exponent element of calculation</td>
<td>The exponent element of the calculation</td>
</tr>
<tr>
<td>[43:39]</td>
<td>NUM, base element of calculation</td>
<td>The base element of the calculation</td>
</tr>
<tr>
<td>[38:37]</td>
<td>TTL, TTL Level hint</td>
<td>TTL Level hint</td>
</tr>
<tr>
<td>[36:0]</td>
<td>BaseADDR, starting address for range</td>
<td>The starting address for the range of the</td>
</tr>
</tbody>
</table>

Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

- 0b00: Reserved.
- 0b01: 4K translation granule.
- 0b10: 16K translation granule.
- 0b11: 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

- 0b00: The entries in the range can be using any level for the translation table entries.
- 0b01: All entries to invalidate are Level 1 translation table entries. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
- 0b10: All entries to invalidate are Level 2 translation table entries.
- 0b11: All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

When FEAT_LPA2 is implemented and TCR_EL1.DS == 1:

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

- When using a 4KB translation granule, BaseADDR[52:16] is treated as 0b0000.
- When using a 16KB translation granule, BaseADDR[52:16] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

- When using a 4KB translation granule, this field is BaseADDR[48:12].
- When using a 16KB translation granule, this field is BaseADDR[50:14].
- When using a 64KB translation granule, this field is BaseADDR[52:16].
Executing TLBI RVAALE1, TLBI RVAALE1NXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RVAALE1{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0110</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsi
if PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsi
if PSTATE.EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBLevel_Last, TLBI_AllAttr, X[t, 64]);
elsi
if PSTATE.EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBLevel_Last, TLBI_AllAttr, X[t, 64]);
elsi
if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsi
if PSTATE.EL == EL0 then
  UNDEFINED;
elsi
if PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsi
if PSTATE.EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBLevel_Last, TLBI_AllAttr, X[t, 64]);
elsi
if PSTATE.EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBLevel_Last, TLBI_AllAttr, X[t, 64]);
elsi
if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsi
if PSTATE.EL == EL0 then
  UNDEFINED;
elsi
if PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);

**TLBI RVAALE1NXS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0110</td>
<td>0b111</td>
</tr>
</tbody>
</table>
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&
(!IsHCRXEL2Enabled() || HCRX_EL2.FGTXS == '0') && HFGITR_EL2.TLBIRVALE1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.FB == '1' then
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[ ], Shareability_ISH, TLBILevel_Last,
                      TLBI_ExcludeXS, X[t, 64]);
else
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[ ], Shareability_NSH, TLBILevel_Last,
                      TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH,
                          TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[ ], Shareability_NSH, TLBILevel_Last,
                          TLBI_ExcludeXS, X[t, 64]);
    endif
elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH,
                          TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[ ], Shareability_NSH, TLBILevel_Last,
                          TLBI_ExcludeXS, X[t, 64]);
    endif
else
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[ ], Shareability_NSH, TLBILevel_Last,
                      TLBI_ExcludeXS, X[t, 64]);
C5.5.33 TLBI RVAALE1IS, TLBI RVAALE1ISNXS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable

The TLBI RVAALE1IS and TLBI RVAALE1ISNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry is within the address range determined by the formula \[\text{BaseADDR} < \text{VA} < \text{BaseADDR} + ((\text{NUM} + 1) \times 2^{5 \times \text{SCALE} + 1} \times \text{Translation Granule Size})\].
- When EL2 is implemented and enabled in the current Security state:
  - If \(\text{HCR_EL2.E2H, TGE}\) is not \(\{1, 1\}\), the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If \(\text{HCR_EL2.E2H, TGE}\) is \(\{1, 1\}\), the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
- The Security state is indicated by the value of \(\text{SCR_EL3.NS}\) if \(\text{FEAT_RME}\) is not implemented, or \(\text{SCR_EL3.NSE, NS}\) if \(\text{FEAT_RME}\) is implemented.
- The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

**Note**

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if \(\text{SCR_EL3.EEL2}==1\), then:

- A PE with \(\text{SCR_EL3.EEL2}==1\) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \(\text{SCR_EL3.EEL2}==0\).
- A PE with \(\text{SCR_EL3.EEL2}==0\) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \(\text{SCR_EL3.EEL2}==1\).
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

**Note**

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

The range of addresses invalidated is **UNPREDICTABLE** when:

- For the 4K translation granularity:
  - If \(\text{TTL}==01\) and BaseADDR\([29:12]\) is not equal to \(0000000000000000\).
  - If \(\text{TTL}==10\) and BaseADDR\([20:12]\) is not equal to \(000000000\).
- For the 16K translation granularity:
  - If \(\text{TTL}==10\) and BaseADDR\([24:14]\) is not equal to \(0000000000\).
- For the 64K translation granularity:
  - If \(\text{TTL}==01\) and BaseADDR\([41:16]\) is not equal to \(00000000000000000000000000\).
--- If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

### Configurations

This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAALEI1IS, TLBI RVAALEI1ISNXS are **UNDEFINED**.

### Attributes

TLBI RVAALEI1IS, TLBI RVAALEI1ISNXS is a 64-bit System instruction.

### Field descriptions

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Field Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-48</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>47-46</td>
<td>Translation granule size</td>
<td></td>
</tr>
<tr>
<td>45-44</td>
<td>The exponent element of the calculation that is used to produce the upper range.</td>
<td>SCALE</td>
</tr>
<tr>
<td>43-39</td>
<td>The base element of the calculation that is used to produce the upper range.</td>
<td>NUM</td>
</tr>
<tr>
<td>38-37</td>
<td>TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.</td>
<td>TTL</td>
</tr>
<tr>
<td>36-32</td>
<td>基ADC</td>
<td>BaseADDR</td>
</tr>
<tr>
<td>31-0</td>
<td>Base ADDR</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:48]**

- Reserved, RES0.

**TG, bits [47:46]**

- Translation granule size.
  - 0b00: Reserved.
  - 0b01: 4K translation granule.
  - 0b10: 16K translation granule.
  - 0b11: 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

**SCALE, bits [45:44]**

The exponent element of the calculation that is used to produce the upper range.

**NUM, bits [43:39]**

The base element of the calculation that is used to produce the upper range.

**TTL, bits [38:37]**

- TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.
  - 0b00: The entries in the range can be using any level for the translation table entries.
  - 0b01: All entries to invalidate are Level 1 translation table entries.
    - If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
  - 0b10: All entries to invalidate are Level 2 translation table entries.
  - 0b11: All entries to invalidate are Level 3 translation table entries.
BaseADDR, bits [36:0]

When FEAT_LPA2 is implemented and TCR_EL1.DS == 1:

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:
The starting address for the range of the maintenance instruction.
When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAALE1IS, TLBI RVAALE1ISNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{TLBI RVAALE1IS}, \langle \text{Xt}\rangle
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0010</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.TLBIRVAALE1IS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
      HCRX_EL2.FnXS == '1' then
      AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
        TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
      AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
        TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL2 then
      if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
          TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
      else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
          TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
      elsif PSTATE.EL == EL3 then
        if HCR_EL2.<E2H,TGE> == '11' then
          AArch64.TLBI_RVAA(SecurityStateAtEL(EL3), Regime_EL10, VMID[], Shareability_ISH,
            TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
        else
          AArch64.TLBI_RVAA(SecurityStateAtEL(EL3), Regime_EL10, VMID[], Shareability_ISH,
            TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
      \]
The A64 System Instruction Class

C5.5 A64 System instructions for TLB maintenance

TLBI RVAALE1ISNXS{, <Xt>}:

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0010</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&
        (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIRVAALE1IS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last,
            TLBI_ExcludeXS, X[t, 64]);
    else
        PSTATE.EL == EL2 then
            if HCR_EL2.<E2H,TGE> == '11' then
                AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Last,
                    TLBI_ExcludeXS, X[t, 64]);
            else
                AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last,
                    TLBI_ExcludeXS, X[t, 64]);
            else
                PSTATE.EL == EL3 then
                    if HCR_EL2.<E2H,TGE> == '11' then
                        AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Last,
                            TLBI_ExcludeXS, X[t, 64]);
                    else
                        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last,
                            TLBI_ExcludeXS, X[t, 64]);
                    else
                        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last,
                            TLBI_ExcludeXS, X[t, 64]);
                    else
                        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last,
                            TLBI_ExcludeXS, X[t, 64]);
                    end
                end
            end
        end
    end
end

The A64 System Instruction Class

C5.5 A64 System instructions for TLB maintenance

TLBI RVAALE1ISNXS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0010</td>
<td>0b111</td>
</tr>
</tbody>
</table>
The TLBI RVAALE1OS and TLBI RVAALE1OSNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry is within the address range determined by the formula \[\text{BaseADDR} \leq VA < \text{BaseADDR} + ((\text{NUM} + 1) \times 2^{(5 \times \text{SCALE} + 1)} \times \text{Translation Granite Size})\].
- When EL2 is implemented and enabled in the current Security state:
  - If \(\text{HCR EL2.E2H, TGE}\) is not \(\{1, 1\}\), the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If \(\text{HCR EL2.E2H, TGE}\) is \(\{1, 1\}\), the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
- The Security state is indicated by the value of \(\text{SCR EL3.NS}\) if \(\text{FEAT RME}\) is not implemented, or \(\text{SCR EL3.NSE, NS}\) if \(\text{FEAT RME}\) is implemented.
- The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

**Note**

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if \(\text{SCR EL3.EEL2}==1\), then:

- A PE with \(\text{SCR EL3.EEL2}==1\) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \(\text{SCR EL3.EEL2}==0\).
- A PE with \(\text{SCR EL3.EEL2}==0\) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \(\text{SCR EL3.EEL2}==1\).
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

**Note**

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

The range of addresses invalidated is **UNPREDICTABLE** when:

- For the 4K translation granule:
  - If \(\text{TTL}==01\) and BaseADDR\[29:12\] is not equal to 00000000000000000000000000.
  - If \(\text{TTL}==10\) and BaseADDR\[20:12\] is not equal to 0000000000.
- For the 16K translation granule:
  - If \(\text{TTL}==10\) and BaseADDR\[24:14\] is not equal to 00000000000.
- For the 64K translation granule:
  - If \(\text{TTL}==01\) and BaseADDR\[41:16\] is not equal to 0000000000000000000000000000000000.
If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RVAALE1OS, TLBI RVAALE1OSNXS are UNDEFINED.

**Attributes**

TLBI RVAALE1OS, TLBI RVAALE1OSNXS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:48]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[47:46]</td>
<td>TG</td>
</tr>
<tr>
<td>[45:44]</td>
<td>SCALE</td>
</tr>
<tr>
<td>[43:39]</td>
<td>NUM</td>
</tr>
<tr>
<td>[38:37]</td>
<td>TTL</td>
</tr>
<tr>
<td>[36:32]</td>
<td>BaseADDR</td>
</tr>
</tbody>
</table>

**Bits [63:48]**

Reserved, RES0.

**TG, bits [47:46]**

Translation granule size.

- **0b00**  Reserved.
- **0b01**  4K translation granule.
- **0b10**  16K translation granule.
- **0b11**  64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

**SCALE, bits [45:44]**

The exponent element of the calculation that is used to produce the upper range.

**NUM, bits [43:39]**

The base element of the calculation that is used to produce the upper range.

**TTL, bits [38:37]**

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

- **0b00**  The entries in the range can be using any level for the translation table entries.
- **0b01**  All entries to invalidate are Level 1 translation table entries.
  
  **IF FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.**
- **0b10**  All entries to invalidate are Level 2 translation table entries.
All entries to invalidate are Level 3 translation table entries.

**BaseADDR, bits [36:0]**

*When FEAT_LPA2 is implemented and TCR_EL1.DS == 1:*

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

*Otherwise:*

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

## Executing TLBI RVAALE1OS, TLBI RVAALE1OSNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RVAALE1OS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0101</td>
<td>0b11</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.TLBIRVAALE1OS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then
      AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
      AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    endif
  endif
elsif PSTATE_EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
  else
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
  endif
elsif PSTATE_EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
  else
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
  endif
```

```
TLBI RVAALE1OSNXS{, <Xt>}  

<table>
<thead>
<tr>
<th></th>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0101</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then  
  UNDEFINED;
elsif PSTATE.EL == EL0 then  
  UNDEFINED;
elsif PSTATE.EL == EL1 then  
  if EL2Enabled() && HCR_EL2.TTLB == '1' then  
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then  
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&  
    (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIRVAALE1OS == '1' then  
    AArch64.SystemAccessTrap(EL2, 0x18);
  else  
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last,  
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then  
  if HCR_EL2.<E2H,TGE> == '11' then  
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,  
    TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
  else  
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last,  
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then  
  if HCR_EL2.<E2H,TGE> == '11' then  
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,  
    TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
  else  
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last,  
    TLBI_ExcludeXS, X[t, 64]);
C5.5.35 TLBI RVAE1, TLBI RVAE1NXS, TLB Range Invalidate by VA, EL1

The TLBI RVAE1 and TLBI RVAE1NXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula: \[ \text{BaseADDR} \leq \text{VA} < \text{BaseADDR} + ((\text{NUM} + 1) \times 2^{(5 \times \text{SCALE} + 1)} \times \text{Translation\_Granule\_Size}) \].
- When EL2 is implemented and enabled in the current Security state:
  - If \text{HCR\_EL2.}{E2H, TGE} is not \{1, 1\}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If \text{HCR\_EL2.}{E2H, TGE} is \{1, 1\}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 0000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 00000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 0000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 000000000000.

**Configurations**

This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAE1, TLBI RVAE1NXS are UNDEFINED.

**Attributes**

TLBI RVAE1, TLBI RVAE1NXS is a 64-bit System instruction.
Field descriptions

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-48</td>
<td>ASID</td>
<td>ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction. Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field. If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.</td>
</tr>
<tr>
<td>47-46</td>
<td>TG</td>
<td>Translation granule size. 0b00 Reserved. 0b01 4K translation granule. 0b10 16K translation granule. 0b11 64K translation granule. The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.</td>
</tr>
<tr>
<td>45-44</td>
<td>SCALE</td>
<td>The exponent element of the calculation that is used to produce the upper range.</td>
</tr>
<tr>
<td>43-39</td>
<td>NUM</td>
<td>The base element of the calculation that is used to produce the upper range.</td>
</tr>
<tr>
<td>38-37</td>
<td>TTL</td>
<td>TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint. 0b00 The entries in the range can be using any level for the translation table entries. 0b01 All entries to invalidate are Level 1 translation table entries. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00. 0b10 All entries to invalidate are Level 2 translation table entries. 0b11 All entries to invalidate are Level 3 translation table entries.</td>
</tr>
<tr>
<td>36-32</td>
<td>BaseADDR</td>
<td>The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules. When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000. When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.</td>
</tr>
</tbody>
</table>

**When FEAT_LPA2 is implemented and TCR_EL1.DS == 1:**

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

**Otherwise:**

The starting address for the range of the maintenance instruction.
When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

**Executing TLBI RVAE1, TLBI RVAE1NXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RVAE1{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0110</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCITR_EL2.TLBIRVAE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
        end
    end
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    end
elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    end

**TLBI RVAE1NXS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0110</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE_EL == EL0 then
        UNDEFINED;
elsif PSTATE_EL == EL1 then
       if EL2Enabled() && HCR_EL2.TTLB == '1' then
                AArch64.SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTeN == '1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIRVAE1 == '1' then
                AArch64.SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && HCR_EL2.FB == '1' then
                AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
        elsif PSTATE_EL == EL2 then
              if HCR_EL2.<E2H,TGE> == '11' then
                    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
        else
                    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
        elsif PSTATE_EL == EL3 then
              if HCR_EL2.<E2H,TGE> == '11' then
                    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
        else
                    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
        

C5.5.36 TLBI RVAE1IS, TLBI RVAE1ISNXS, TLB Range Invalidate by VA, EL1, Inner Shareable

The TLBI RVAE1IS and TLBI RVAE1ISNXS characteristics are:

**Purpose**

Invalidate cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula 
  \[ \text{BaseADDR} \leq \text{VA} < \text{BaseADDR} + ((\text{NUM} + 1) \cdot 2^{5 \cdot \text{SCALE} + 1} \cdot \text{Translation Granule Size}) \].
- When EL2 is implemented and enabled in the current Security state:
  - If \( \text{HCR_EL2.\{E2H, TGE\}} \) is not \( \{1, 1\} \), the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If \( \text{HCR_EL2.\{E2H, TGE\}} \) is \( \{1, 1\} \), the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of \( \text{SCR_EL3.NS} \) if \( \text{FEAT_RME} \) is not implemented, or \( \text{SCR_EL3.\{NSE, NS\}} \) if \( \text{FEAT_RME} \) is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

--- **Note** ---

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if \( \text{SCR_EL3.EEL2}==1 \), then:

- A PE with \( \text{SCR_EL3.EEL2}==1 \) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \( \text{SCR_EL3.EEL2}==0 \).
- A PE with \( \text{SCR_EL3.EEL2}==0 \) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \( \text{SCR_EL3.EEL2}==1 \).
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 0000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 0000000000.
For the 64K translation granule:
- If TTL == 01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
- If TTL == 10 and BaseADDR[28:16] is not equal to 0000000000000.

Configurations
This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAE1IS, TLBI RVAE1ISNXS are UNDEFINED.

Attributes
TLBI RVAE1IS, TLBI RVAE1ISNXS is a 64-bit System instruction.

Field descriptions

<table>
<thead>
<tr>
<th>63</th>
<th>48</th>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ASID, bits [63:48]
ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.
Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.
If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]
Translation granule size.
0b00  Reserved.
0b01  4K translation granule.
0b10  16K translation granule.
0b11  64K translation granule.
The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]
The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]
The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]
TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.
0b00  The entries in the range can be using any level for the translation table entries.
0b01  All entries to invalidate are Level 1 translation table entries.
      If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10  All entries to invalidate are Level 2 translation table entries.
All entries to invalidate are Level 3 translation table entries.

**BaseADDR, bits [36:0]**

*When FEAT_LPA2 is implemented and TCR_EL1.DS == 1:*

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16]
for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as \texttt{0b0000}.
When using a 16KB translation granule, BaseADDR[15:14] is treated as \texttt{0b00}.

*Otherwise:*

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

**Executing TLBI RVAE1IS, TLBI RVAE1ISNXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RVAE1IS\{, \texttt{<Xt>}\}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{0b01}</td>
<td>\texttt{0b000}</td>
<td>\texttt{0b1000}</td>
<td>\texttt{0b0010}</td>
<td>\texttt{0b001}</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.TLBIRVAE1IS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
    HCRX_EL2.FnXS == '1' then
      AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
    else
      AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    end
  end
elsif PSTATE.EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
  else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
  end
elsif PSTATE.EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
  else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
  end
```
TLBI RVAE1SNXS{,<X>}
C5.5.37 TLBI RVAE1OS, TLBI RVAE1OSNXS, TLB Range Invalidate by VA, EL1, Outer Shareable

The TLBI RVAE1OS and TLBI RVAE1OSNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula \([\text{BaseADDR} \leq \text{VA} < \text{BaseADDR} + ((\text{NUM} + 1) \times 2^{5 \times \text{SCALE} + 1} \times \text{Translation_Granule_Size})]\).
- When EL2 is implemented and enabled in the current Security state:
  - If \(HCR_{EL2}.\{E2H, TGE\} \neq \{1, 1\}\), the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If \(HCR_{EL2}.\{E2H, TGE\} = \{1, 1\}\), the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of \(SCR_{EL3}.\{NS, NSE, NS\}\) if \(FEAT_RME\) is not implemented, or \(SCR_{EL3}.\{NSE, NS\}\) if \(FEAT_RME\) is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

--- **Note** ---

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if \(SCR_{EL3}.\{EEL2\}==1\), then:

- A PE with \(SCR_{EL3}.\{EEL2\}==1\) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \(SCR_{EL3}.\{EEL2\}==0\).
- A PE with \(SCR_{EL3}.\{EEL2\}==0\) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \(SCR_{EL3}.\{EEL2\}==1\).
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

---

The range of addresses invalidated is **UNPREDICTABLE** when:

- For the 4K translation granule:
  - If \(TTL==01\) and \(\text{BaseADDR}[29:12]\) is not equal to \(0000000000000000\).
  - If \(TTL==10\) and \(\text{BaseADDR}[20:12]\) is not equal to \(000000000\).
- For the 16K translation granule:
  - If \(TTL==10\) and \(\text{BaseADDR}[24:14]\) is not equal to \(0000000000\).
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

For the 64K translation granule:
  — If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  — If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000000.

Configurations

This instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RVAE1OS, TLBI RVAE1OSNXS are UNDEFINED.

Attributes

TLBI RVAE1OS, TLBI RVAE1OSNXS is a 64-bit System instruction.

Field descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASID</td>
<td>63-48</td>
<td>ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction. Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field. If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.</td>
</tr>
<tr>
<td>TG</td>
<td>47-46</td>
<td>Translation granule size.</td>
</tr>
<tr>
<td>NUM</td>
<td>43-39</td>
<td>The base element of the calculation that is used to produce the upper range.</td>
</tr>
<tr>
<td>SCALE</td>
<td>45-44</td>
<td>The exponent element of the calculation that is used to produce the upper range.</td>
</tr>
<tr>
<td>TTL</td>
<td>38-37</td>
<td>TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.</td>
</tr>
<tr>
<td>BaseADDR</td>
<td>36-0</td>
<td></td>
</tr>
</tbody>
</table>

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

0b00  Reserved.
0b01  4K translation granule.
0b10  16K translation granule.
0b11  64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

0b00  The entries in the range can be using any level for the translation table entries.
0b01  All entries to invalidate are Level 1 translation table entries.

If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

0b10 All entries to invalidate are Level 2 translation table entries.
0b11 All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

When \textit{FEAT\_LPA2} is implemented and TCR\_EL1.DS == 1:
The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.
When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:
The starting address for the range of the maintenance instruction.
When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE1OS, TLBI RVAE1OSNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

\textit{TLBI RVAE1OS(<Xt>)}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0101</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR\_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR\_EL2.TTLB05 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR\_EL3.FGTEn == '1') && HFGITR\_EL2.TLBIRVAE1OS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if IsFeatureImplemented(FEAT\_XS) && IsFeatureImplemented(FEAT\_HCK) && IsHCRXEL2Enabled() &&
       HCR\_EL2.FnXS == '1' then
      AArch64.TLBI\_RVA(SecurityStateAtEL(EL1), Regime\_EL10, VMID[], Shareability\_OSH,
       TLBLevel\_Any, TLBI\_ExcludeXS, X[t, 64]);
    else
      AArch64.TLBI\_RVA(SecurityStateAtEL(EL1), Regime\_EL10, VMID[], Shareability\_OSH,
       TLBLevel\_Any, TLBI\_AllAttr, X[t, 64]);
    endif
  endif
elsif PSTATE.EL == EL2 then
  if HCR\_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI\_RVA(SecurityStateAtEL(EL2), Regime\_EL20, VMID\_NONE, Shareability\_OSH, TLBLevel\_Any, TLBI\_AllAttr, X[t, 64]);
  else
    AArch64.TLBI\_RVA(SecurityStateAtEL(EL2), Regime\_EL20, VMID\_NONE, Shareability\_OSH, TLBLevel\_Any, TLBI\_AllAttr, X[t, 64]);
  endif
elsif PSTATE.EL == EL3 then
  if HCR\_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI\_RVA(SecurityStateAtEL(EL2), Regime\_EL20, VMID\_NONE, Shareability\_OSH, TLBLevel\_Any, TLBI\_AllAttr, X[t, 64]);
  else
    AArch64.TLBI\_RVA(SecurityStateAtEL(EL2), Regime\_EL20, VMID\_NONE, Shareability\_OSH, TLBLevel\_Any, TLBI\_AllAttr, X[t, 64]);
  endif
else
  if HCR\_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI\_RVA(SecurityStateAtEL(EL2), Regime\_EL20, VMID\_NONE, Shareability\_OSH, TLBLevel\_Any, TLBI\_AllAttr, X[t, 64]);
  else
    AArch64.TLBI\_RVA(SecurityStateAtEL(EL2), Regime\_EL20, VMID\_NONE, Shareability\_OSH, TLBLevel\_Any, TLBI\_AllAttr, X[t, 64]);
  endif
else
  if HCR\_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI\_RVA(SecurityStateAtEL(EL2), Regime\_EL20, VMID\_NONE, Shareability\_OSH, TLBLevel\_Any, TLBI\_AllAttr, X[t, 64]);
  else
    AArch64.TLBI\_RVA(SecurityStateAtEL(EL2), Regime\_EL20, VMID\_NONE, Shareability\_OSH, TLBLevel\_Any, TLBI\_AllAttr, X[t, 64]);
  endif
endif
TLBI RVAE1OSNXS( <Xt>)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0101</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
UNDEFINED;
elsif PSTATE_EL == EL0 then
UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&
    (!IsHCRXEl2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIRVAAIOS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
    TLBI_excludeXS, X[t, 64]);
  endif
elsif PSTATE_EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any,
    TLBI_excludeXS, X[t, 64]);
  elseif PSTATE_EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
      AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any,
      TLBI_excludeXS, X[t, 64]);
    else
      AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
      TLBI_excludeXS, X[t, 64]);
    endif
  endif
else
  AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
  TLBI_excludeXS, X[t, 64]);
C5.5.38 TLBI RVAE2, TLBI RVAE2NXS, TLB Range Invalidate by VA, EL2

The TLBI RVAE2 and TLBI RVAE2NXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate any VA in the range determined by the formula
  \[
  \text{BaseADDR} \leq \text{VA} < \text{BaseADDR} + ((\text{NUM} + 1) \times 2^{(5 \times \text{SCALE} + 1) \times \text{Translation_Granule_Size}})
  \]
  using the EL2 or EL2&0 translation regime, as determined by the current value of the HCR_EL2.E2H bit, for the Security state.
- If HCR_EL2.E2H == 0, the entry is from any level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configurations

This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAE2, TLBI RVAE2NXS are UNDEFINED.

Attributes

TLBI RVAE2, TLBI RVAE2NXS is a 64-bit System instruction.
Field descriptions

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASID [63:48]</td>
<td>ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction. Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field. If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.</td>
<td>Variable</td>
</tr>
<tr>
<td>TG [47:46]</td>
<td>Translation granule size.</td>
<td>0b00: Reserved, 0b01: 4K translation granule, 0b10: 16K translation granule, 0b11: 64K translation granule</td>
</tr>
<tr>
<td>SCALE [45:44]</td>
<td>The exponent element of the calculation that is used to produce the upper range.</td>
<td>Variable</td>
</tr>
<tr>
<td>NUM [43:39]</td>
<td>The base element of the calculation that is used to produce the upper range.</td>
<td>Variable</td>
</tr>
<tr>
<td>TTL [38:37]</td>
<td>TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.</td>
<td>0b00: The entries in the range can be using any level for the translation table entries, 0b01: All entries to invalidate are Level 1 translation table entries, 0b10: All entries to invalidate are Level 2 translation table entries, 0b11: All entries to invalidate are Level 3 translation table entries</td>
</tr>
<tr>
<td>BaseADDR [36:0]</td>
<td>The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules. When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.</td>
<td>Variable</td>
</tr>
</tbody>
</table>
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

**Otherwise:**

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

**Executing TLBI RVAE2, TLBI RVAE2NXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

### TLBI RVAE2{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0110</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE_EL == EL2 then
        if HCR_EL2.E2H == '1' then
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    elsif PSTATE_EL == EL3 then
        if !EL2Enabled() then
            UNDEFINED;
        elsif HCR_EL2.E2H == '1' then
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
        else
            AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
```

### TLBI RVAE2NXS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0110</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```c
if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    UNDEFINED;
elsif PSTATE_EL == EL2 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE_EL == EL2 then
```
if HCR_EL2.E2H == '1' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NSH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
else if PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    else if HCR_EL2.E2H == '1' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Any,
        TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NSH, TLBILevel_Any,
        TLBI_ExcludeXS, X[t, 64]);
C5.5.39   TLBI RVAE2IS, TLBI RVAE2ISNXS, TLB Range Invalidate by VA, EL2, Inner Shareable

The TLBI RVAE2IS and TLBI RVAE2ISNXS characteristics are:

**Purpose**

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate any VA in the range determined by the formula
  \[ \text{BaseADDR} \leq \text{VA} \leq \text{BaseADDR} + ((\text{NUM} + 1) \times 2^{(5\times\text{SCALE} + 1)} \times \text{Translation Granule Size}) \]
  using the EL2 or EL2&0 translation regime, as determined by the current value of the HCR_EL2.E2H bit, for the Security state.
- If HCR_EL2.E2H == 0, the entry is from any level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 0000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAE2IS, TLBI RVAE2ISNXS are UNDEFINED.

**Attributes**

TLBI RVAE2IS, TLBI RVAE2ISNXS is a 64-bit System instruction.
### Field descriptions

#### ASID, bits [63:48]

- **When HCR_EL2.E2H == 1:**
  - ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.
  - Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.
  - If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

- **Otherwise:**
  - Reserved, RES0.

#### TG, bits [47:46]

Translation granule size.
- 0b00  Reserved.
- 0b01  4K translation granule.
- 0b10  16K translation granule.
- 0b11  64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

#### SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

#### NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

#### TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.
- 0b00  The entries in the range can be using any level for the translation table entries.
- 0b01  All entries to invalidate are Level 1 translation table entries. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
- 0b10  All entries to invalidate are Level 2 translation table entries.
- 0b11  All entries to invalidate are Level 3 translation table entries.

#### BaseADDR, bits [36:0]

- **When FEAT_LPA2 is implemented and TCR_EL2.DS == 1:**
  - The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.
  - When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:
The starting address for the range of the maintenance instruction.
When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE2IS, TLBI RVAE2ISNXS instruction
Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RVAE2IS{, <Xt>)**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    UNDEFINED;
elseif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
  else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
  elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
      UNDEFINED;
elseif HCR_EL2.E2H == '1' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
  else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);

**TLBI RVAE2ISNXS{, <Xt>)**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elseif PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    UNDEFINED;
elseif PSTATE.EL == EL2 then
if HCR_EL2.E2H == '1' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '1' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
        TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Any,
        TLBI_ExcludeXS, X[t, 64]);
    endif
else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
endif
C5.5.40 TLBI RVAE2OS, TLBI RVAE2OSNXS, TLB Range Invalidate by VA, EL2, Outer Shareable

The TLBI RVAE2OS and TLBI RVAE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate any VA in the range determined by the formula
  \[\text{BaseADDR} \leq \text{VA} \leq \text{BaseADDR} + ((\text{NUM} + 1) \times 2^{(\text{SCALE} + 1)} \times \text{Translation Granule Size})\]
  using the EL2 or EL2&0 translation regime, as determined by the current value of the \text{HCR_EL2.E2H} bit, for the Security state.
- If \text{HCR_EL2.E2H} == 0, the entry is from any level of the translation table walk.
- If \text{HCR_EL2.E2H} == 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of \text{SCR_EL3.NS} if \text{FEAT_RME} is not implemented, or \text{SCR_EL3.[NSE, NS]} if \text{FEAT_RME} is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If \text{TTL==01} and \text{BaseADDR[29:12]} is not equal to 0000000000000000.
  - If \text{TTL==10} and \text{BaseADDR[20:12]} is not equal to 00000000.
- For the 16K translation granule:
  - If \text{TTL==10} and \text{BaseADDR[24:14]} is not equal to 00000000.
- For the 64K translation granule:
  - If \text{TTL==01} and \text{BaseADDR[41:16]} is not equal to 00000000000000000000000.
  - If \text{TTL==10} and \text{BaseADDR[28:16]} is not equal to 000000000000.

If \text{FEAT_XS} is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configurations

This instruction is present only when \text{FEAT_TLBIRANGE} is implemented and \text{FEAT_TLBIOS} is implemented. Otherwise, direct accesses to TLBI RVAE2OS, TLBI RVAE2OSNXS are UNDEFINED.

Attributes

TLBI RVAE2OS, TLBI RVAE2OSNXS is a 64-bit System instruction.
Field descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASID, bits [63:48]</td>
<td></td>
<td>When HCR_EL2.E2H == 1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.</td>
</tr>
<tr>
<td></td>
<td>0b00</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>0b01</td>
<td>4K translation granule.</td>
</tr>
<tr>
<td></td>
<td>0b10</td>
<td>16K translation granule.</td>
</tr>
<tr>
<td></td>
<td>0b11</td>
<td>64K translation granule.</td>
</tr>
<tr>
<td>SCALE, bits [45:44]</td>
<td></td>
<td>The exponent element of the calculation that is used to produce the upper range.</td>
</tr>
<tr>
<td>NUM, bits [43:39]</td>
<td></td>
<td>The base element of the calculation that is used to produce the upper range.</td>
</tr>
<tr>
<td>TTL, bits [38:37]</td>
<td></td>
<td>TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.</td>
</tr>
<tr>
<td></td>
<td>0b00</td>
<td>The entries in the range can be using any level for the translation table entries.</td>
</tr>
<tr>
<td></td>
<td>0b01</td>
<td>All entries to invalidate are Level 1 translation table entries.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.</td>
</tr>
<tr>
<td></td>
<td>0b10</td>
<td>All entries to invalidate are Level 2 translation table entries.</td>
</tr>
<tr>
<td></td>
<td>0b11</td>
<td>All entries to invalidate are Level 3 translation table entries.</td>
</tr>
<tr>
<td>BaseADDR, bits [36:0]</td>
<td></td>
<td>When FEAT_LPA2 is implemented and TCR_EL2.DS == 1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.</td>
</tr>
</tbody>
</table>
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

**Otherwise:**

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

**Executing TLBI RVAE2OS, TLBI RVAE2OSNXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RVAE2OS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0101</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsesi PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsesi PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Shareability_OSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
elsesi PSTATE.EL == EL3 then
  if !EL2Enabled() then
    UNDEFINED;
elsesi HCR_EL2.E2H == '1' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Shareability_OSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);

**TLBI RVAE2OSNXS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0101</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsesi PSTATE.EL == EL0 then
  UNDEFINED;
elsesi PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsesi PSTATE.EL == EL2 then

if HCR_EL2.E2H == '1' then
   AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any,
   TLBI_ExcludeXS, X[t, 64]);
else
   AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBILevel_Any,
   TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
   if !EL2Enabled() then
      UNDEFINED;
   elsif HCR_EL2.E2H == '1' then
      AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any,
      TLBI_ExcludeXS, X[t, 64]);
   else
      AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBILevel_Any,
      TLBI_ExcludeXS, X[t, 64]);
C5.5.41 TLBI RVAE3, TLBI RVAE3NXS, TLB Range Invalidate by VA, EL3

The TLBI RVAE3 and TLBI RVAE3NXS characteristics are:

**Purpose**

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula \[ \text{BaseADDR} \leq \text{VA} < \text{BaseADDR} + ((\text{NUM} + 1) \times 2^{(5 \times \text{SCALE} + 1) \times \text{Translation Granule Size}}) \].

The invalidation applies to the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAE3, TLBI RVAE3NXS are UNDEFINED.

**Attributes**

TLBI RVAE3, TLBI RVAE3NXS is a 64-bit System instruction.

**Field descriptions**

```
<table>
<thead>
<tr>
<th>63</th>
<th>48</th>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>TG</td>
<td>NUM</td>
<td>TTL</td>
<td>BaseADDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Bits [63:48]**

Reserved, RES0.
TG, bits [47:46]
Translation granule size.
0b00  Reserved.
0b01  4K translation granule.
0b10  16K translation granule.
0b11  64K translation granule.
The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]
The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]
The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]
TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.
0b00  The entries in the range can be using any level for the translation table entries.
0b01  All entries to invalidate are Level 1 translation table entries.
      If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10  All entries to invalidate are Level 2 translation table entries.
0b11  All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

When FEAT_LPA2 is implemented and TCR_EL3.DS == 1:
The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.
When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.
Otherwise:
The starting address for the range of the maintenance instruction.
When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE3, TLBI RVAE3NXS instruction
Accesses to this instruction use the following encodings in the System instruction encoding space:

TLBI RVAE3{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1000</td>
<td>0b0110</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_NSH, TLBILevel_Any,
  TLBI_AllAttr, X[t, 64]);

**TLBI RVAE3NXS{, <XP>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1001</td>
<td>0b0110</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_NSH, TLBILevel_Any,
  TLBI_ExcludeXS, X[t, 64]);
The TLBI RVAE3IS and TLBI RVAE3ISNXS characteristics are:

**Purpose**

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula \([BaseADDR <= VA < BaseADDR + ((NUM +1)*2^{(5*SCALE +1)} * Translation_Granule_Size)]\).

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 00000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 0000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 00000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAE3IS, TLBI RVAE3ISNXS are UNDEFINED.

**Attributes**

TLBI RVAE3IS, TLBI RVAE3ISNXS is a 64-bit System instruction.

**Field descriptions**

```
| 63  | 48  | 47  | 46  | 45  | 44  | 43  | 39  | 38  | 37  | 36  | 35  | 34  | 33  | 32  | 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| RES0|     | TG  | NUM | TTL | BaseADDR |
|     |     |     |     | SCALE |
|     | BaseADDR |
```

**Bits [63:48]**

Reserved, RES0.
TG, bits [47:46]
Translation granule size.
0b00  Reserved.
0b01  4K translation granule.
0b10  16K translation granule.
0b11  64K translation granule.
The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]
The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]
The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]
TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.
0b00  The entries in the range can be using any level for the translation table entries.
0b01  All entries to invalidate are Level 1 translation table entries.
0b10  All entries to invalidate are Level 2 translation table entries.
0b11  All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]
When FEAT_LPA2 is implemented and TCR_EL3.DS == 1:
The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.
When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.
Otherwise:
The starting address for the range of the maintenance instruction.
When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE3IS, TLBI RVAE3ISNXS instruction
Accesses to this instruction use the following encodings in the System instruction encoding space:

TLBI RVAE3IS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1000</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_ISH, TLBILevel_Any,
  TLBI_AllAttr, X[t, 64]);

\textit{TLBI \texttt{RVA3ISNX3}, \texttt{<Xt>}}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b10</td>
<td>0b1001</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_ISH, TLBILevel_Any,
  TLBI_ExludeXS, X[t, 64]);
C5.5.43 TLBI RVAE3OS, TLBI RVAE3OSNXS, TLB Range Invalidate by VA, EL3, Outer Shareable

The TLBI RVAE3OS and TLBI RVAE3OSNXS characteristics are:

**Purpose**

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula $[\text{BaseADDR} \leq \text{VA} < \text{BaseADDR} + ((\text{NUM} + 1) \times 2^{(5 \times \text{SCALE} + 1) \times \text{Translation_Granule_Size}})$.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is **UNPREDICTABLE** when:

- For the 4K translation granule:
  - If $\text{TTL}==01$ and $\text{BaseADDR}[29:12]$ is not equal to 00000000000000000000000000000000.
  - If $\text{TTL}==10$ and $\text{BaseADDR}[20:12]$ is not equal to 0000000000000000000000000000000000.
- For the 16K translation granule:
  - If $\text{TTL}==10$ and $\text{BaseADDR}[24:14]$ is not equal to 000000000000000000000000000000000000000000000.
- For the 64K translation granule:
  - If $\text{TTL}==01$ and $\text{BaseADDR}[41:16]$ is not equal to 0000000000000000000000000000000000000000000000000000000000000000.
  - If $\text{TTL}==10$ and $\text{BaseADDR}[28:16]$ is not equal to 0000000000000000000000000000000000000000000000000000000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RVAE3OS, TLBI RVAE3OSNXS are UNDEFINED.

**Attributes**

TLBI RVAE3OS, TLBI RVAE3OSNXS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>48-46</td>
<td>TG</td>
</tr>
<tr>
<td>43-39</td>
<td>NUM, TTL, BaseADDR</td>
</tr>
<tr>
<td>31-28</td>
<td>SCALE</td>
</tr>
<tr>
<td>28-0</td>
<td>BaseADDR</td>
</tr>
</tbody>
</table>

**Bits [63:48]**

Reserved, RES0.
TG, bits [47:46]
Translation granule size.
- 0b00: Reserved.
- 0b01: 4K translation granule.
- 0b10: 16K translation granule.
- 0b11: 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]
The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]
The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]
TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.
- 0b00: The entries in the range can be using any level for the translation table entries.
- 0b01: All entries to invalidate are Level 1 translation table entries.
- 0b10: All entries to invalidate are Level 2 translation table entries.
- 0b11: All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

When FEAT_LPA2 is implemented and TCR_EL3.DS == 1:
The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.
- When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
- When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:
The starting address for the range of the maintenance instruction.
- When using a 4KB translation granule, this field is BaseADDR[48:12].
- When using a 16KB translation granule, this field is BaseADDR[50:14].
- When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE3OS, TLBI RVAE3OSNXS instruction
Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RVAE3OS({ <X> })**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1000</td>
<td>0b0101</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_RVA(SecurityStateAtEL(REL3), Regime_EL3, VMID[], Shareability_OSH, TLBILevel_Any,
  TLBI_AllAttr, X[t, 64]);

TLBI RVAE3OSNXS(, <Xt>)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1001</td>
<td>0b0101</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_RVA(SecurityStateAtEL(REL3), Regime_EL3, VMID[], Shareability_OSH, TLBILevel_Any,
  TLBI_ExcludeXS, X[t, 64]);
C5.5.44 TLBI RVALE1, TLBI RVALE1NXS, TLB Range Invalidate by VA, Last level, EL1

The TLBI RVALE1 and TLBI RVALE1NXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula \([\text{BaseADDR} \leq \text{VA} < \text{BaseADDR} + ((\text{NUM} +1) \times 2^{5 \cdot \text{SCALE} + 1}) \times \text{Translation_Granule_Size}]\).
- When EL2 is implemented and enabled in the current Security state:
  - If \(\text{HCR\_EL2}.\{\text{E2H, TGE}\} = \{0, 1\}\), the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If \(\text{HCR\_EL2}.\{\text{E2H, TGE}\} = \{1, 1\}\), the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of \(\text{SCR\_EL3}.\text{NS}\) if \(\text{FEAT\_RME}\) is not implemented, or \(\text{SCR\_EL3}.\{\text{NSE, NS}\}\) if \(\text{FEAT\_RME}\) is implemented.

The invalidation applies to the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If \(\text{TTL} = 01\) and \(\text{BaseADDR}[29:12]\) is not equal to 0000000000000000.
  - If \(\text{TTL} = 10\) and \(\text{BaseADDR}[20:12]\) is not equal to 00000000.
- For the 16K translation granule:
  - If \(\text{TTL} = 10\) and \(\text{BaseADDR}[24:14]\) is not equal to 0000000000.
- For the 64K translation granule:
  - If \(\text{TTL} = 01\) and \(\text{BaseADDR}[41:16]\) is not equal to 0000000000000000.
  - If \(\text{TTL} = 10\) and \(\text{BaseADDR}[28:16]\) is not equal to 0000000000.

For more information about the architectural requirements for this System instruction, see Invalidating TLB entries from stage 2 translations on page D8-5211.

If \(\text{FEAT\_XS}\) is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when \(\text{FEAT\_TLBIRANGE}\) is implemented. Otherwise, direct accesses to TLBI RVALE1, TLBI RVALE1NXS are UNDEFINED.
Attributes

TLBI RVALE1, TLBI RVALE1NXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

- 0b00: Reserved.
- 0b01: 4K translation granule.
- 0b10: 16K translation granule.
- 0b11: 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

- 0b00: The entries in the range can be using any level for the translation table entries.
- 0b01: All entries to invalidate are Level 1 translation table entries.
  If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
- 0b10: All entries to invalidate are Level 2 translation table entries.
- 0b11: All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

When FEAT_LPA2 is implemented and TCR_EL1.DS == 1:

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

 Executing TLBI RVALE1, TLBI RVALE1NXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RVALE1{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0110</td>
<td>0b101</td>
</tr>
</tbody>
</table>

```
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HCR_EL2.TLBIRVALE1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.FB == '1' then
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && HCRX_EL2.FnXS == '1' then
      AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
      AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    end
  elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
      AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
      AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    end
  elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
      AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
      AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
  end
```
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

### TLBI RVALE1NXS( <Xt> )

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0110</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&
    (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIRVAE1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.FB == '1' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
  endif
elsif PSTATE.EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH,
    TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
  endif
elsif PSTATE.EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH,
    TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
  endif
TLBI RVALE1IS, TLBI RVALE1ISNXS, TLB Range Invalidate by VA, Last level, EL1, Inner Shareable

The TLBI RVALE1IS and TLBI RVALE1ISNXS characteristics are:

**Purpose**

Invalidate cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula: \[ \text{BaseADDR} \leq \text{VA} < \text{BaseADDR} + ((\text{NUM} + 1) \times 2^{5 \times \text{SCALE} + 1} \times \text{Translation_Granule_Size}) \].
- When EL2 is implemented and enabled in the current Security state:
  - If \( \text{HCR}_\text{EL2}.\{\text{E2H}, \text{TGE}\} \) is not \( \{1, 1\} \), the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If \( \text{HCR}_\text{EL2}.\{\text{E2H}, \text{TGE}\} \) is \( \{1, 1\} \), the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of \( \text{SCR}_\text{EL3}.\text{NS} \) if \( \text{FEAT}_\text{RME} \) is not implemented, or \( \text{SCR}_\text{EL3}.\{\text{NSE}, \text{NS}\} \) if \( \text{FEAT}_\text{RME} \) is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

---

**Note**

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if \( \text{SCR}_\text{EL3}.\text{EEL2}==1 \), then:

- A PE with \( \text{SCR}_\text{EL3}.\text{EEL2}==1 \) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \( \text{SCR}_\text{EL3}.\text{EEL2}==0 \).
- A PE with \( \text{SCR}_\text{EL3}.\text{EEL2}==0 \) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \( \text{SCR}_\text{EL3}.\text{EEL2}==1 \).
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

---

The range of addresses invalidated is **UNPREDICTABLE** when:

- For the 4K translation granule:
  - If \( \text{TTL}==01 \) and \( \text{BaseADDR}[29:12] \) is not equal to 000000000000000000000000.
  - If \( \text{TTL}==10 \) and \( \text{BaseADDR}[20:12] \) is not equal to 0000000000.
- For the 16K translation granule:
  - If \( \text{TTL}==10 \) and \( \text{BaseADDR}[24:14] \) is not equal to 00000000000.
- For the 64K translation granule:
  - If \( \text{TTL}==01 \) and \( \text{BaseADDR}[41:16] \) is not equal to 00000000000000000000000000.
If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configurations

This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVALE1IS, TLBI RVALE1ISNXS are UNDEFINED.

Attributes

TLBI RVALE1IS, TLBI RVALE1ISNXS is a 64-bit System instruction.

Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

0b00  Reserved.

0b01  4K translation granule.

0b10  16K translation granule.

0b11  64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

0b00  The entries in the range can be using any level for the translation table entries.
All entries to invalidate are Level 1 translation table entries.

If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.

All entries to invalidate are Level 2 translation table entries.

All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

When FEAT_LPA2 is implemented and TCR_EL1.DS == 1:
The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules. When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000. When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:
The starting address for the range of the maintenance instruction. When using a 4KB translation granule, this field is BaseADDR[48:12]. When using a 16KB translation granule, this field is BaseADDR[50:14]. When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE1IS, TLBI RVALE1ISNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

<table>
<thead>
<tr>
<th>TLBI RVALE1IS{, &lt;Xt&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
</tr>
<tr>
<td>0b01</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then 
  UNDEFINED;
elsif PSTATE.EL == EL1 then 
  if EL2Enabled() && HCR_EL2.TTLB == '1' then 
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then 
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.TLBIRVALE1IS == '1' then 
    AArch64.SystemAccessTrap(EL2, 0x18);
  else 
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then 
      AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else 
      AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    endforeach;
elsif PSTATE.EL == EL2 then 
  if HCR_EL2.<E2H,TGE> == '11' then 
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
  else 
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
  endif;
elsif PSTATE.EL == EL3 then 
  if HCR_EL2.<E2H,TGE> == '11' then 
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
  else 
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
  endif;
else 
  AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
endif;
AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);

TLBI RVAE1ISNXS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0010</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&
    (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIRVAE1IS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
  endif
elsif PSTATE.EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
  endif
elsif PSTATE.EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
  endif
C5.5.46 TLBI RVALE1OS, TLBI RVALE1OSNXS, TLB Range Invalidate by VA, Last level, EL1, Outer Shareable

The TLBI RVALE1OS and TLBI RVALE1OSNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula \(\text{BaseADDR} \leq \text{VA} < \text{BaseADDR} + ((\text{NUM} +1) \times 2^{(5\times\text{SCALE} +1}) \times \text{Translation Granule Size})\).
- When EL2 is implemented and enabled in the current Security state:
  - If \(\text{HCR_EL2.E2H, TGE}\) is not \{1, 1\}, the entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.
  - If \(\text{HCR_EL2.E2H, TGE}\) is \{1, 1\}, the entry would be required to translate any of the VAs in the specified address range using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of \(\text{SCR_EL3.NS}\) if \(\text{FEAT_RME}\) is not implemented, or \(\text{SCR_EL3.NSE, NS}\) if \(\text{FEAT_RME}\) is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

---

**Note**

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if \(\text{SCR_EL3.EEL2}==1\), then:

- A PE with \(\text{SCR_EL3.EEL2}==1\) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \(\text{SCR_EL3.EEL2}==0\).
- A PE with \(\text{SCR_EL3.EEL2}==0\) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \(\text{SCR_EL3.EEL2}==1\).
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

---

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If \(\text{TTL}==01\) and \(\text{BaseADDR}[29:12]\) is not equal to 0000000000000000.
  - If \(\text{TTL}==10\) and \(\text{BaseADDR}[20:12]\) is not equal to 0000000000.
- For the 16K translation granule:
  - If \(\text{TTL}==10\) and \(\text{BaseADDR}[24:14]\) is not equal to 00000000000.
- For the 64K translation granule:
  - If \(\text{TTL}==01\) and \(\text{BaseADDR}[41:16]\) is not equal to 00000000000000000000000000.
If \( \text{TTL} = 10 \) and \( \text{BaseADDR}[28:16] \) is not equal to \( 0000000000000000 \).

If \text{FEAT\_XS} is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when \text{FEAT\_TLMIRANGE} is implemented and \text{FEAT\_TLMIBIOS} is implemented. Otherwise, direct accesses to TLBI RVALE1OS, TLBI RVALE1OSNXS are UNDEFINED.

**Attributes**

TLBI RVALE1OS, TLBI RVALE1OSNXS is a 64-bit System instruction.

**Field descriptions**

- **ASID**, bits [63:48]
  - ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.
  - Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.
  - If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

- **TG**, bits [47:46]
  - Translation granule size.
  - \( 0b00 \) Reserved.
  - \( 0b01 \) 4K translation granule.
  - \( 0b10 \) 16K translation granule.
  - \( 0b11 \) 64K translation granule.
  - The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

- **SCALE**, bits [45:44]
  - The exponent element of the calculation that is used to produce the upper range.

- **NUM**, bits [43:39]
  - The base element of the calculation that is used to produce the upper range.
TTL, bits [38:37]
TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

- **0b00**: The entries in the range can be using any level for the translation table entries.
- **0b01**: All entries to invalidate are Level 1 translation table entries. If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
- **0b10**: All entries to invalidate are Level 2 translation table entries.
- **0b11**: All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

*When FEAT_LPA2 is implemented and TCR_EL1.DS == 1:*

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

*Otherwise:*

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

**Executing TLBI RVALE1OS, TLBI RVALE1OSNXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{TLBI RVALE1OS\{, <Xt>\}}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0101</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.TLBIRVALE1OS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
      HCRX_EL2.FnXS == '1' then
      AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
      AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
      if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
      else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last,}
TLBI_AllAttr, X[t, 64]);
elseif PSTATE.EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,
    TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
  else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last,
    TLBI_AllAttr, X[t, 64]);

TLBI RVALE1OSNXS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0101</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() & HCR_EL2.TTLBS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() & (!HaveEL(EL3) | SCR_EL3.FGTEn == '1') & IsFeatureImplemented(FEAT_HCX) &
  (IsHCRXEL2Enabled() | HCRX_EL2.FGTnxS == '0') & HFGITR_EL2.TLBIRVAE1OS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
elseif PSTATE.EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,
    TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
elseif PSTATE.EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,
    TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
C5.5.47 TLBI RVALE2, TLBI RVALE2NXS, TLB Range Invalidate by VA, Last level, EL2

The TLBI RVALE2 and TLBI RVALE2NXS characteristics are:

**Purpose**

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate any VA in the range determined by the formula \[ \text{BaseADDR} \leq VA \leq \text{BaseADDR} + ((\text{NUM} + 1) \times 2^{(5 \times \text{SCALE} + 1)} \times \text{Translation Granule Size}) \] using the EL2 or EL2&0 translation regime, as determined by the current value of the HCR_EL2.E2H bit, for the Security state.
- If HCR_EL2.E2H == 0, the entry is from the final level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 0000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 00000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVALE2, TLBI RVALE2NXS are UNDEFINED.

**Attributes**

TLBI RVALE2, TLBI RVALE2NXS is a 64-bit System instruction.
Field descriptions

<table>
<thead>
<tr>
<th>63</th>
<th>48</th>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASID</td>
<td>TG</td>
<td>NUM</td>
<td>TTL</td>
<td>BaseADDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCALE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ASID, bits [63:48]

When \texttt{HCR\_EL2.E2H} == 1:

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

Reserved, \texttt{RES0}.

TG, bits [47:46]

Translation granule size.

- \texttt{0b00}: Reserved.
- \texttt{0b01}: 4K translation granule.
- \texttt{0b10}: 16K translation granule.
- \texttt{0b11}: 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

- \texttt{0b00}: The entries in the range can be using any level for the translation table entries.
- \texttt{0b01}: All entries to invalidate are Level 1 translation table entries.
  
  \textbf{If \texttt{FEAT\_LPA2} is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as \texttt{0b00}.}

- \texttt{0b10}: All entries to invalidate are Level 2 translation table entries.
- \texttt{0b11}: All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

When \texttt{FEAT\_LPA2} is implemented and \texttt{TCR\_EL2.DS} == 1:

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as \texttt{0b0000}. 
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE2, TLBI RVALE2NXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RVALE2{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0110</td>
<td>0b10</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH,
            TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NSH, TLBILevel_Last,
            TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '1' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH,
            TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NSH, TLBILevel_Last,
            TLBI_AllAttr, X[t, 64]);
```

**TLBI RVALE2NXS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0110</td>
<td>0b10</td>
</tr>
</tbody>
</table>

```plaintext
if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
```
if HCR_EL2.E2H == '1' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH,
    TLBLevel_Last, TLBI_ExcludeXS, X[t, 64]);
else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NSH, TLBLevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '1' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH,
        TLBLevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NSH, TLBLevel_Last,
        TLBI_ExcludeXS, X[t, 64]);
C5.5.48 TLBI RVALE2IS, TLBI RVALE2ISNXS, TLB Range Invalidate by VA, Last level, EL2, Inner Shareable

The TLBI RVALE2IS and TLBI RVALE2ISNXS characteristics are:

**Purpose**

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)] using the EL2 or EL2&0 translation regime, as determined by the current value of the HCR_EL2.E2H bit, for the Security state.
- If HCR_EL2.E2H == 0, the entry is from the final level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is **UNPREDICTABLE** when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000000000.
  - If TTL==10 and BaseADDR[20: 12] is not equal to 00000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 00000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVALE2IS, TLBI RVALE2ISNXS are UNDEFINED.

**Attributes**

TLBI RVALE2IS, TLBI RVALE2ISNXS is a 64-bit System instruction.
### Field descriptions

![Field descriptions diagram](image-url)

**ASID, bits [63:48]**  
*When HCR_EL2.E2H == 1:*  
ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.  
Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.  
If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

**Otherwise:**  
Reserved, RES0.

**TG, bits [47:46]**  
Translation granule size.  
0b00  Reserved.  
0b01  4K translation granule.  
0b10  16K translation granule.  
0b11  64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

**SCALE, bits [45:44]**  
The exponent element of the calculation that is used to produce the upper range.

**NUM, bits [43:39]**  
The base element of the calculation that is used to produce the upper range.

**TTL, bits [38:37]**  
TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.  
0b00  The entries in the range can be using any level for the translation table entries.  
0b01  All entries to invalidate are Level 1 translation table entries.  
If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.  
0b10  All entries to invalidate are Level 2 translation table entries.  
0b11  All entries to invalidate are Level 3 translation table entries.

**BaseADDR, bits [36:0]**  
*When FEAT_LPA2 is implemented and TCR_EL2.DS == 1:*  
The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.  
When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granule, BaseADDR[15:14] is treated as \(0b00\).

**Otherwise:**

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

### Executing TLBI RVALE2IS, TLBI RVALE2ISNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RVALE2IS\{, <Xt>\}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0010</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    endif
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    endif
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '1' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    endif
elsif !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elseif PSTATE.EL == EL2 then

**TLBI RVALE2ISNXS\{, <Xt>\}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0010</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elseif PSTATE.EL == EL0 then
    UNDEFINED;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elseif PSTATE.EL == EL2 then

// The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance
if HCR_EL2.E2H == '1' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
    TLBLevel_Last, TLBI_ExcludeXS, X[t, 64]);
else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBLevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '1' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
        TLBLevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBLevel_Last,
        TLBI_ExcludeXS, X[t, 64]);

C5.5.49   TLBI RVALE2OS, TLBI RVALE2OSNXS, TLB Range Invalidate by VA, Last level, EL2, Outer Shareable

The TLBI RVALE2OS and TLBI RVALE2OSNXS characteristics are:

**Purpose**

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate any VA in the range determined by the formula
  \[ \text{BaseADDR} <= \text{VA} < \text{BaseADDR} + ((\text{NUM} + 1) \times 2^{(5 \times \text{SCALE} + 1) \times \text{Translation Granule Size}}) \]
  using the EL2 or EL2&0 translation regime, as determined by the current value of the HCR_EL2.E2H bit, for the Security state.
- If HCR_EL2.E2H == 0, the entry is from the final level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is **UNPREDICTABLE** when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 0000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 00000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 0000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 00000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RVALE2OS, TLBI RVALE2OSNXS are **UNDEFINED**.

**Attributes**

TLBI RVALE2OS, TLBI RVALE2OSNXS is a 64-bit System instruction.
Field descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASID</td>
<td>63-48</td>
</tr>
<tr>
<td>TG</td>
<td>47-46</td>
</tr>
<tr>
<td>NUM</td>
<td>43-39</td>
</tr>
<tr>
<td>TTL</td>
<td>38-37</td>
</tr>
<tr>
<td>BaseADDR</td>
<td>36-0</td>
</tr>
</tbody>
</table>

**ASID, bits [63:48]**

*When HCR_EL2.E2H == 1:*

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

*Otherwise:*

Reserved, RES0.

**TG, bits [47:46]**

Translation granule size.

- **0b00**: Reserved.
- **0b01**: 4K translation granule.
- **0b10**: 16K translation granule.
- **0b11**: 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

**SCALE, bits [45:44]**

The exponent element of the calculation that is used to produce the upper range.

**NUM, bits [43:39]**

The base element of the calculation that is used to produce the upper range.

**TTL, bits [38:37]**

TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.

- **0b00**: The entries in the range can be using any level for the translation table entries.
- **0b01**: All entries to invalidate are Level 1 translation table entries. 
  *If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.*
- **0b10**: All entries to invalidate are Level 2 translation table entries.
- **0b11**: All entries to invalidate are Level 3 translation table entries.

**BaseADDR, bits [36:0]**

*When FEAT_LPA2 is implemented and TCR_EL2.DS == 1:*

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

Otherwise:

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

**Executing TLBI RVALE2OS, TLBI RVALE2OSNXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RVALE2OS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0101</td>
<td>0b10</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if !EL2Enabled() then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,
        TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    elsif HCR_EL2.E2H == '1' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBILevel_Last,
        TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last,
        TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '1' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBILevel_Last,
        TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last,
        TLBI_AllAttr, X[t, 64]);

**TLBI RVALE2OSNXS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0101</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    }
if HCR_EL2.E2H == '1' then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,
    TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '1' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,
        TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBILevel_Last,
        TLBI_ExcludeXS, X[t, 64]);
    endif
else
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
C5.5.50 TLBI RVALE3, TLBI RVALE3NXS, TLB Range Invalidate by VA, Last level, EL3

The TLBI RVALE3 and TLBI RVALE3NXS characteristics are:

**Purpose**

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula \([\text{BaseADDR} \leq \text{VA} < \text{BaseADDR} + ((\text{NUM} + 1) \times 2^{(5 \times \text{SCALE} + 1)} \times \text{Translation_Granule_Size})]\).

The invalidation applies to the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.

- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 0000000000.

- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVALE3, TLBI RVALE3NXS are UNDEFINED.

**Attributes**

TLBI RVALE3, TLBI RVALE3NXS is a 64-bit System instruction.

**Field descriptions**

```
<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>61</th>
<th>60</th>
<th>59</th>
<th>58</th>
<th>57</th>
<th>56</th>
<th>55</th>
<th>54</th>
<th>53</th>
<th>48</th>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>TG</td>
<td>NUM</td>
<td>TTL</td>
<td>BaseADDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCALE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BaseADDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits [63:48]

Reserved, RES0.
```
TG, bits [47:46]
Translation granule size.
0b00  Reserved.
0b01  4K translation granule.
0b10  16K translation granule.
0b11  64K translation granule.
The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]
The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]
The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]
TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.
0b00  The entries in the range can be using any level for the translation table entries.
0b01  All entries to invalidate are Level 1 translation table entries.
0b10  All entries to invalidate are Level 2 translation table entries.
0b11  All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]
When FEAT_LPA2 is implemented and TCR_EL3.DS == 1:
The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.
When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.
Otherwise:
The starting address for the range of the maintenance instruction.
When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE3, TLBI RVALE3NXS instruction
Accesses to this instruction use the following encodings in the System instruction encoding space:

TLBI RVALE3{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1000</td>
<td>0b0110</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
UNDEFINED;
elsif PSTATE_EL == EL2 then
  UNDEFINED;
elsif PSTATE_EL == EL3 then
  AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);

TLI_RVALE3NXS(, <xt>)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1001</td>
<td>0b0110</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  UNDEFINED;
elsif PSTATE_EL == EL2 then
  UNDEFINED;
elsif PSTATE_EL == EL3 then
  AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
C5.5.51 TLBI RVALE3IS, TLBI RVALE3ISNXS, TLB Range Invalidate by VA, Last level, EL3, Inner Shareable

The TLBI RVALE3IS and TLBI RVALE3ISNXS characteristics are:

**Purpose**

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula \([\text{BaseADDR} \leq \text{VA} < \text{BaseADDR} + ((\text{NUM} +1)*2^{5 \times \text{SCALE}} +1) \times \text{Translation\_Granule\_Size})\].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If \(\text{TTL}==01\) and \(\text{BaseADDR}[29:12]\) is not equal to 000000000000000000.
  - If \(\text{TTL}==10\) and \(\text{BaseADDR}[20:12]\) is not equal to 0000000000.
- For the 16K translation granule:
  - If \(\text{TTL}==10\) and \(\text{BaseADDR}[24:14]\) is not equal to 0000000000.
- For the 64K translation granule:
  - If \(\text{TTL}==01\) and \(\text{BaseADDR}[41:16]\) is not equal to 00000000000000000000000000.
  - If \(\text{TTL}==10\) and \(\text{BaseADDR}[28:16]\) is not equal to 000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVALE3IS, TLBI RVALE3ISNXS are UNDEFINED.

**Attributes**

TLBI RVALE3IS, TLBI RVALE3ISNXS is a 64-bit System instruction.

**Field descriptions**

```
+---+---+---+---+---+---+---+---+
| 63| 62| 61| 60| 47| 46| 45| 44| 43| 39| 38| 37| 36| 35| 34| 33| 32| 31| 0   |
+---+---+---+---+---+---+---+---+
| RES0| TG| NUM| TTL| BaseADDR |
+---+---+---+---+---+---+---+---+
| SCALE| BaseADDR |
+---+---+---+---+---+---+---+---+
```

**Bits [63:48]**

Reserved, RES0.
TG, bits [47:46]
Translation granule size.
0b00  Reserved.
0b01  4K translation granule.
0b10  16K translation granule.
0b11  64K translation granule.
The instruction takes a translation granule size for the translations that are being invalidated. If the
translations used a different translation granule size than the one being specified, then the
architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]
The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]
The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]
TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the
level described by the TTL hint.
0b00  The entries in the range can be using any level for the translation table entries.
0b01  All entries to invalidate are Level 1 translation table entries.
      If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value
      is reserved and hardware should treat this field as 0b00.
0b10  All entries to invalidate are Level 2 translation table entries.
0b11  All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

When FEAT_LPA2 is implemented and TCR_EL3.DS == 1:
The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16]
for all translation granules.
   When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
   When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.
Otherwise:
The starting address for the range of the maintenance instruction.
   When using a 4KB translation granule, this field is BaseADDR[48:12].
   When using a 16KB translation granule, this field is BaseADDR[50:14].
   When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE3IS, TLBI RVALE3ISNXS instruction
Accesses to this instruction use the following encodings in the System instruction encoding space:

TLBI RVALE3IS({, <Xt>})

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1000</td>
<td>0b0010</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsi if PSTATE.EL == EL1 then
UNDEFINED;
elseif PSTATE.EL == EL2 then
  UNDEFINED;
elseif PSTATE.EL == EL3 then
  AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_ISH, TLBILevel_Last,
  TLBI_AllAttr, X[t, 64]);

**TLBI RVA\(EL_{3}\)NS\{, \(<X_t>\)}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1001</td>
<td>0b0010</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elseif PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  UNDEFINED;
elseif PSTATE.EL == EL2 then
  UNDEFINED;
elseif PSTATE.EL == EL3 then
  AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_ISH, TLBILevel_Last,
  TLBI_ExcludeXS, X[t, 64]);
**TLBI RVALE3OS, TLBI RVALE3OSNXS, TLB Range Invalidate by VA, Last level, EL3, Outer Shareable**

The TLBI RVALE3OS and TLBI RVALE3OSNXS characteristics are:

### Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula \[\text{BaseADDR} \leq \text{VA} < \text{BaseADDR} + ((\text{NUM} + 1) \times 2^{(5 \times \text{SCALE} - 1) \times \text{Translation_Granule_Size}})\].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

The range of addresses invalidated is **UNPREDICTABLE** when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 00000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 0000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 000000000000.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

### Configurations

This instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RVALE3OS, TLBI RVALE3OSNXS are **UNDEFINED**.

### Attributes

TLBI RVALE3OS, TLBI RVALE3OSNXS is a 64-bit System instruction.

### Field descriptions

```
+---+---+---+---+---+---+---+---+
|63|62|61|60|47|46|45|44|43|42|41|40|
+---+---+---+---+---+---+---+---+
|   |   |   |   | TG|   |   |   | NUM|TTL|BaseADDR|
+---+---+---+---+---+---+---+---+
|39|38|37|36|35|34|33|32|31|30|29|28|
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |SCALE|   |   |
+---+---+---+---+---+---+---+---+
|27|26|25|24|23|22|21|20|19|18|17|16|
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |BaseADDR|   |   |
+---+---+---+---+---+---+---+---+
```

**Bits [63:48]**

Reserved, RES0.
TG, bits [47:46]
Translation granule size.
- 0b00: Reserved.
- 0b01: 4K translation granule.
- 0b10: 16K translation granule.
- 0b11: 64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]
The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]
The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]
TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.
- 0b00: The entries in the range can be using any level for the translation table entries.
- 0b01: All entries to invalidate are Level 1 translation table entries.
  If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
- 0b10: All entries to invalidate are Level 2 translation table entries.
- 0b11: All entries to invalidate are Level 3 translation table entries.

BaseADDR, bits [36:0]

*When FEAT_LPA2 is implemented and TCR_EL3.DS == 1:*

The starting address for the range of the maintenance instructions. This field is BaseADDR[52:16] for all translation granules.

When using a 4KB translation granule, BaseADDR[15:12] is treated as 0b0000.
When using a 16KB translation granule, BaseADDR[15:14] is treated as 0b00.

*Otherwise:*

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].
When using a 16KB translation granule, this field is BaseADDR[50:14].
When using a 64KB translation granule, this field is BaseADDR[52:16].

**Executing TLBI RVALE3OS, TLBI RVALE3OSNXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI RVALE3OS{, < Xt> }**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b10</td>
<td>0b1000</td>
<td>0b0101</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elif PSTATE.EL == EL1 then
UNDEFINED;
elseif PSTATE.EL == EL2 then
    UNDEFINED;
elseif PSTATE.EL == EL3 then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_OSH, TLBILevel_Last,
    TLBI_AllAttr, X[t, 64]);

**TLBI RVALE3OSNXS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1001</td>
<td>0b0101</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elseif PSTATE.EL == EL0 then
    UNDEFINED;
elseif PSTATE.EL == EL1 then
    UNDEFINED;
elseif PSTATE.EL == EL2 then
    UNDEFINED;
elseif PSTATE.EL == EL3 then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_OSH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
C5.5.53 TLBI VAAE1, TLBI VAAE1NXS, TLB Invalidate by VA, All ASID, EL1

The TLBI VAAE1 and TLBI VAAE1NXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the current Security state:
  - If HCR_EL2.{E2H, TGE} is not \{1, 1\}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If HCR_EL2.{E2H, TGE} is \{1, 1\}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

**Note**

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VAAE1, TLBI VAAE1NXS is a 64-bit System instruction.

**Field descriptions**

- **Bits [63:48]**
  - Reserved, RES0.
- **Bits [47:44]**
  - TTL
- **Bits [43:32]**
  - VA[55:12]
- **Bits [31:0]**
  - VA[55:12]
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

TTL, bits [47:44]

**When FEAT_TTL is implemented:**

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

- **0b00xx** No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

- **0b01xx** The entry comes from a 4KB translation granule. The level of walk for the leaf level 0b0x is encoded as:
  - 0b00 : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b01 : Level 1.
  - 0b10 : Level 2.
  - 0b11 : Level 3.

- **0b10xx** The entry comes from a 16KB translation granule. The level of walk for the leaf level 0b0x is encoded as:
  - 0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01 : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b10 : Level 2.
  - 0b11 : Level 3.

- **0b11xx** The entry comes from a 64KB translation granule. The level of walk for the leaf level 0b0x is encoded as:
  - 0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01 : Level 1.
  - 0b10 : Level 2.
  - 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

**Otherwise:**

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

**Executing TLBI VAAE1, TLBI VAAE1NXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:
TLBI VAAE1{, <Xt>}

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b01 & 0b000 & 0b1000 & 0b0111 & 0b011 \\
\end{array}
\]

if \text{PSTATE.EL} == ELO then
UNDEFINED;
elsif \text{PSTATE.EL} == EL1 then
  if \text{EL2Enabled()} \&\& \text{HCR_EL2.TTLB} == '1' then
    \text{AArch64.SystemAccessTrap(EL2, 0x18)};
  elsif \text{EL2Enabled()} \&\& \left(\text{HaveEL(EL3)} \mid \text{SCR_EL3.FGETn == '1'}\right) \&\& \text{HFCITR_EL2.TLBVAE1 == '1'} then
    \text{AArch64.SystemAccessTrap(EL2, 0x18)};
  elseif \text{EL2Enabled()} \&\& \text{HCR_EL2.FB} == '1' then
    if \text{IsFeatureImplemented(FEAT_XS)} \&\& \text{IsFeatureImplemented(FEAT_HCX)} \&\& \text{HCRX_EL2.FnXS} == '1' then
      \text{AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], ShareabilityISH, TLBLevelAny, TLBI_ExcludeXS, X[t, 64])};
    else
      \text{AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], ShareabilityISH, TLBLevelAny, TLBI_AllAttr, X[t, 64])};
    end
  elsif \text{EL2Enabled()} \&\& \left(\text{HaveEL(EL3)} \mid \text{SCR_EL3.FGETn == '1'}\right) \&\& \text{IsFeatureImplemented(FEAT_HCX)} \&\& \left(\text{IsHCRX_EL2Enabled()} \&\& \text{HCRX_EL2.FGEn} == '1'\right) then
    \text{AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], ShareabilityNSH, TLBLevelAny, TLBI_ExcludeXS, X[t, 64])};
  else
    \text{AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], ShareabilityNSH, TLBLevelAny, TLBI_AllAttr, X[t, 64])};
  end
elsif \text{PSTATE.EL} == EL2 then
  if \text{HCR_EL2.<E2H,TGE> == '11'} then
    \text{AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, ShareabilityNSH, TLBLevelAny, TLBI_AllAttr, X[t, 64])};
  else
    \text{AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], ShareabilityISH, TLBLevelAny, TLBI_AllAttr, X[t, 64])};
  end
elsif \text{PSTATE.EL} == EL3 then
  if \text{HCR_EL2.<E2H,TGE> == '11'} then
    \text{AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, ShareabilityNSH, TLBLevelAny, TLBI_AllAttr, X[t, 64])};
  else
    \text{AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], ShareabilityNSH, TLBLevelAny, TLBI_AllAttr, X[t, 64])};
end

TLBI VAAE1NXS{, <Xt>}

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b01 & 0b000 & 0b1001 & 0b0111 & 0b011 \\
\end{array}
\]

if \text{IsFeatureImplemented(FEAT_XS)} then
  UNDEFINED;
elsif \text{PSTATE.EL} == ELO then
  UNDEFINED;
elsif \text{PSTATE.EL} == EL1 then
  if \text{EL2Enabled()} \&\& \text{HCR_EL2.TTLB} == '1' then
    \text{AArch64.SystemAccessTrap(EL2, 0x18)};
  elsif \text{EL2Enabled()} \&\& \left(\text{HaveEL(EL3)} \mid \text{SCR_EL3.FGETn == '1'}\right) \&\& \text{IsFeatureImplemented(FEAT_HCX)} \&\& \left(\text{IsHCRX_EL2Enabled()} \&\& \text{HCRX_EL2.FGEn} == '0'\right) \&\& \text{HFCITR_EL2.TLBVAE1 == '1'} then
    \text{AArch64.SystemAccessTrap(EL2, 0x18)};
  elseif \text{EL2Enabled()} \&\& \text{HCR_EL2.FB} == '1' then
    \text{AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], ShareabilityISH, TLBLevelAny,
TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBLevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
  elseif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
      AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBLevel_Any,
      TLBI_ExcludeXS, X[t, 64]);
    else
      AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBLevel_Any,
      TLBI_ExcludeXS, X[t, 64]);
    endif
  else
    if HCR_EL2.<E2H,TGE> == '11' then
      AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBLevel_Any,
      TLBI_ExcludeXS, X[t, 64]);
    else
      AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBLevel_Any,
      TLBI_ExcludeXS, X[t, 64]);
    endif
  endif
endif
C5.5.54  TLBI VAAE1IS, TLBI VAAE1ISNXS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable

The TLBI VAAE1IS and TLBI VAAE1ISNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the current Security state:
  - If 
    
    HCR_EL2.{E2H, TGE} is not \{1, 1\}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If 
    
    HCR_EL2.{E2H, TGE} is \{1, 1\}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of 

SCR_EL3.NS if FEAT_RME is not implemented, or

SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

--- Note ---

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if 

SCR_EL3.EEL2==1, then:

- A PE with 
  
  SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with 
  
  SCR_EL3.EEL2==0.
- A PE with 
  
  SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with 
  
  SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

--- Note ---

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VAAE1IS, TLBI VAAE1ISNXS is a 64-bit System instruction.
Field descriptions

Bits [63:48]
Reserved, RES0.

TTL, bits [47:44]

*When FEAT_TTL is implemented:*

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

- **0b00xx**: No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
- **0b01xx**: The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b01: Level 1.
  - 0b10: Level 2.
  - 0b11: Level 3.
- **0b10xx**: The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b10: Level 2.
  - 0b11: Level 3.
- **0b11xx**: The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01: Level 1.
  - 0b10: Level 2.
  - 0b11: Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

*Otherwise:*
Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:
- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

**Executing TLBI VAAE1IS, TLBI VAAE1ISNXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI VAAE1IS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() & HCR_EL2.TTLBIS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCITR_EL2.TLBIVAAE1IS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if IsFeatureImplemented(FEAT_XS) & IsFeatureImplemented(FEAT_HC) & IsHCRXEL2Enabled() &
    HCRX_EL2.FnXS == '1' then
      AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
    TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
      AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
    TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL2 then
      if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
      TLBI_AllAttr, X[t, 64]);
    else
      AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
      TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL3 then
      if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
      TLBI_AllAttr, X[t, 64]);
    else
      AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
      TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
  AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
  TLBI_AllAttr, X[t, 64]);

**TLBI VAAE1ISNXS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0011</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTen == '1') && IsFeatureImplemented(FEAT_HCX) &&
    (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIVAEIIS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
        TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
        TLBI_ExcludeXS, X[t, 64]);
    else
        if HCR_EL2.<E2H,TGE> == '11' then
            AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
            TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
            TLBI_ExcludeXS, X[t, 64]);
        end
    end
end
The TLBI VAAE1OS and TLBI VAAE1OSNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the current Security state:
  - If \( HCR\_EL2.\{E2H, TGE\} \) is not \( \{1, 1\} \), the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If \( HCR\_EL2.\{E2H, TGE\} \) is \( \{1, 1\} \), the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of \( SCR\_EL3.NS \) if FEAT_RME is not implemented, or \( SCR\_EL3.\{NSE, NS\} \) if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

**Note**

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if \( SCR\_EL3.EEL2==1 \), then:

- A PE with \( SCR\_EL3.EEL2==1 \) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \( SCR\_EL3.EEL2==0 \).
- A PE with \( SCR\_EL3.EEL2==0 \) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \( SCR\_EL3.EEL2==1 \).
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

**Note**

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined. Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VAAE1OS, TLBI VAAE1OSNXS are UNDEFINED.

**Attributes**

TLBI VAAE1OS, TLBI VAAE1OSNXS is a 64-bit System instruction.
Field descriptions

Bits [63:48]
Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:
Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

- 0b00xx: No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
- 0b01xx: The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b0: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b1: Level 1.
  - 0b10: Level 2.
  - 0b11: Level 3.
- 0b10xx: The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b0: Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b1: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b10: Level 2.
  - 0b11: Level 3.
- 0b11xx: The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b0: Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b1: Level 1.
  - 0b10: Level 2.
  - 0b11: Level 3.

Otherwise:
Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
• Where a 64KB translation granule is being used, bits [3:0] of this field are \texttt{RES0} and ignored when the instruction is executed, because \texttt{VA[15:12]} have no effect on the operation of the instruction.

**Executing TLBI VAAE1OS, TLBI VAAE1OSNXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\textbf{TLBI VAAE1OS\{, <Xt>\}}

\begin{center}
\begin{tabular}{cccccc}
\hline
\textbf{op0} & \textbf{op1} & \textbf{CRn} & \textbf{CRm} & \textbf{op2} \\
\hline
\texttt{0b01} & \texttt{0b000} & \texttt{0b1000} & \texttt{0b0001} & \texttt{0b011} \\
\hline
\end{tabular}
\end{center}

\begin{verbatim}
if \texttt{PSTATE.EL} == \texttt{EL0} then
  UNDEFINED;
elsif \texttt{PSTATE.EL} == \texttt{EL1} then
  if \texttt{EL2Enabled()} \&\& \texttt{HCR_EL2.TTLB} == '1' then
    \texttt{AArch64.SystemAccessTrap(EL2, 0x18)};
  elsif \texttt{EL2Enabled()} \&\& \texttt{HCR_EL2.TTLBOS} == '1' then
    \texttt{AArch64.SystemAccessTrap(EL2, 0x18)};
  elsif \texttt{EL2Enabled()} \&\& (!HaveEL(EL3) \texttt{|| SCR_EL3.FGTEn} == '1') \&\& \texttt{HFGITR_EL2.TLBIVAAE1OS} == '1' then
    \texttt{AArch64.SystemAccessTrap(EL2, 0x18)};
  else
    if \texttt{IsFeatureImplemented(FEAT_XS)} \&\& \texttt{IsFeatureImplemented(FEAT_HCX)} \&\& \texttt{IsHCRXEL2Enabled()} \&\& \texttt{HCRX_EL2.FnXS} == '1' then
      \texttt{AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);}
    else
      \texttt{AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);}
    else
      if \texttt{PSTATE.EL} == \texttt{EL2} then
        if \texttt{HCR_EL2.<E2H,TGE>} == '11' then
          \texttt{AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);}
        else
          \texttt{AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);}
        else
          \texttt{AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);}
        else
          \texttt{AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);}
      else
        if \texttt{HCR_EL2.<E2H,TGE>} == '11' then
          \texttt{AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);}
        else
          \texttt{AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);}
      else
        if \texttt{PSTATE.EL} == \texttt{EL3} then
          if \texttt{HCR_EL2.<E2H,TGE>} == '11' then
            \texttt{AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);}
          else
            \texttt{AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);}
      else
        if \texttt{PSTATE.EL} == \texttt{EL0} then
          UNDEFINED;
        elsif \texttt{PSTATE.EL} == \texttt{EL1} then
          UNDEFINED;
      else
        if \texttt{EL2Enabled()} \&\& \texttt{HCR_EL2.TTLB} == '1' then
          \texttt{AArch64.SystemAccessTrap(EL2, 0x18)};
      \end{verbatim}

\textbf{TLBI VAAE1OSNXS\{, <Xt>\}}

\begin{center}
\begin{tabular}{cccccc}
\hline
\textbf{op0} & \textbf{op1} & \textbf{CRn} & \textbf{CRm} & \textbf{op2} \\
\hline
\texttt{0b01} & \texttt{0b000} & \texttt{0b1001} & \texttt{0b0001} & \texttt{0b011} \\
\hline
\end{tabular}
\end{center}

\begin{verbatim}
if !\texttt{IsFeatureImplemented(FEAT_XS)} then
  UNDEFINED;
elsif \texttt{PSTATE.EL} == \texttt{EL0} then
  UNDEFINED;
elsif \texttt{PSTATE.EL} == \texttt{EL1} then
  if \texttt{EL2Enabled()} \&\& \texttt{HCR_EL2.TTLB} == '1' then
    \texttt{AArch64.SystemAccessTrap(EL2, 0x18)};
\end{verbatim}
elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&
    (!IsHCRXEL2Enabled() || HCRX_EL2.FGTXS == '0') && HFGITR_EL2.TLBIVAAEI0S == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any,
        TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
        TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any,
        TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
        TLBI_ExcludeXS, X[t, 64]);
else
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
C5.5.56 TLBI VAALE1, TLBI VAALE1NXS, TLB Invalidate by VA, All ASID, Last level, EL1

The TLBI VAALE1 and TLBI VAALE1NXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- When EL2 is implemented and enabled in the current Security state:
  - If HCR_EL2.{E2H, TGE} is not \{1, 1\}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If HCR_EL2.{E2H, TGE} is \{1, 1\}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

**Note**

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VAALE1, TLBI VAALE1NXS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>48</th>
<th>47</th>
<th>44</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>TTL</td>
<td>VA[55:12]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>0</td>
<td>VA[55:12]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:48]**

Reserved, RES0.
TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

- **0b00xx**: No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
- **0b01xx**: The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b01: Level 1.
  - 0b10: Level 2.
  - 0b11: Level 3.
- **0b10xx**: The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b10: Level 2.
  - 0b11: Level 3.
- **0b11xx**: The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01: Level 1.
  - 0b10: Level 2.
  - 0b11: Level 3.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAALE1, TLBI VAALE1NXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:
TLBI VAALE1{, <Xt>}

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b11</td>
</tr>
</tbody>
</table>
```

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.TLBIVAALE1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.FB == '1' then
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && HCRX_EL2.FnXS == '1' then
      AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBLevel_Last, TLBIExcludeXS, X[t, 64]);
    else
      AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBLevel_Last, TLBIAllAttr, X[t, 64]);
    end;
  else
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && HCRX_EL2.FnXS == '1' then
      AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBLevel_Last, TLBIExcludeXS, X[t, 64]);
    else
      AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBLevel_Last, TLBIAllAttr, X[t, 64]);
    end;
  endif;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBLevel_Last, TLBIAllAttr, X[t, 64]);
  else
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBLevel_Last, TLBIAllAttr, X[t, 64]);
  endif;
elsif PSTATE_EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBLevel_Last, TLBIAllAttr, X[t, 64]);
  else
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBLevel_Last, TLBIAllAttr, X[t, 64]);
  endif;
else
  if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
  elsif PSTATE_EL == EL0 then
    UNDEFINED;
  elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnx == '0') && HFGITR_EL2.TLBIVAALE1 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.FB == '1' then
      AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBLevel_Last, TLBIExcludeXS, X[t, 64]);
    else
      AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBLevel_Last, TLBIAllAttr, X[t, 64]);
    endif;
```

TLBI VAALE1NXS{, <Xt>}

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0111</td>
<td>0b11</td>
</tr>
</tbody>
</table>
```

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnx == '0') && HFGITR_EL2.TLBIVAALE1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.FB == '1' then
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBLevel_Last, TLBIExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBLevel_Last, TLBIAllAttr, X[t, 64]);
```

TLBI VAALE1NXS{, <Xt>}

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0111</td>
<td>0b11</td>
</tr>
</tbody>
</table>
```
TLBI_ExcludeXS, X[t, 64]);
else
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH,
        TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last,
        TLBI_ExcludeXS, X[t, 64]);
    endif
elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH,
        TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last,
        TLBI_ExcludeXS, X[t, 64]);
    endif
else
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
C5.5.57 TLBI VAALE1IS, TLBI VAALE1ISNXS, TLB Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable

The TLBI VAALE1IS and TLBI VAALE1ISNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- When EL2 is implemented and enabled in the current Security state:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

--- **Note** ---

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

--- **Note** ---

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined. Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VAALE1IS, TLBI VAALE1ISNXS is a 64-bit System instruction.
Field descriptions

Bits [63:48]
Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:
Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

0b00xx  No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

0b01xx  The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
           0b00 : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
           0b01 : Level 1.
           0b10 : Level 2.
           0b11 : Level 3.

0b10xx  The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
           0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
           0b01 : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
           0b10 : Level 2.
           0b11 : Level 3.

0b11xx  The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
           0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
           0b01 : Level 1.
           0b10 : Level 2.
           0b11 : Level 3.

Otherwise:
Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
• Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

**Executing TLBI VAALE1IS, TLBI VAALE1ISNXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI VAALE1IS(, <Xt>)**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Crn</th>
<th>Crm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & HCR_EL2.TTLBIS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFGITR_EL2.TLBIVAALE1IS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if IsFeatureImplemented(FEAT_XS) & IsFeatureImplemented(FEAT_HCX) & IsHCRXE1L2Enabled() &
      HCRX_EL2.FnXS == '1' then
      AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
      AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL2 then
      if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
      else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
      end
    elsif PSTATE.EL == EL3 then
      if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
      else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
      AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    end

TLBI VAALE1ISNXS(, <Xt>)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Crn</th>
<th>Crm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0011</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&
    (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIVALEL1IS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBLevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
        TLBLevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBLevel_Last,
        TLBI_ExcludeXS, X[t, 64]);
    endif
elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
        TLBLevel_Last, TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBLevel_Last,
        TLBI_ExcludeXS, X[t, 64]);
The TLBI VAALE1OS and TLBI VAALE1OSNXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- When EL2 is implemented and enabled in the current Security state:
  - If HCR_EL2.{E2H, TGE} is not \{1, 1\}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If HCR_EL2.{E2H, TGE} is \{1, 1\}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 and EL2&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configurations

This instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VAALE1OS, TLBI VAALE1OSNXS are UNDEFINED.

Attributes

TLBI VAALE1OS, TLBI VAALE1OSNXS is a 64-bit System instruction.
Field descriptions

Bits [63:48]:
Reserved, RES0.

TTL, bits [47:44]:
When FEAT_TTL is implemented:
Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

0b00xx: No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

0b01xx: The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
0b00: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
0b01: Level 1.
0b10: Level 2.
0b11: Level 3.

0b10xx: The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
0b00: Reserved. Treat as if TTL<3:2> is 0b00.
0b01: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
0b10: Level 2.
0b11: Level 3.

0b11xx: The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
0b00: Reserved. Treat as if TTL<3:2> is 0b00.
0b01: Level 1.
0b10: Level 2.
0b11: Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:
Reserved, RES0.

VA[55:12], bits [43:0]:
Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:
• Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
• Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

### Executing TLBI VAALE1OS, TLBI VAALE1OSNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

#### TLBI VAALE1OS( <xt> )

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b0000</td>
<td>0b1000</td>
<td>0b0001</td>
<td>0b111</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.TLBIVAALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
        else
            AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
        end
    end
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    end
elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    end
else
    if PSTATE.EL == EL0 then
        UNDEFINED;
    elsif PSTATE.EL == EL1 then
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        if EL2Enabled() && HCR_EL2.TTLB == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
    end
```

#### TLBI VAALE1OSNXS( <xt> )

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b0000</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b111</td>
</tr>
</tbody>
</table>

```c
if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```
elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) && SCR_EL3.FG7En == '1') && IsFeatureImplemented(FEAT_HCX) &&
  (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIVALEL1OS == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last,
                     TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,
                      TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Shareability_OSH, TLBILevel_Last,
                      TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VAA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,
                      TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Shareability_OSH, TLBILevel_Last,
                      TLBI_ExcludeXS, X[t, 64]);
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

C5.5.59 TLBI VAE1, TLBI VAE1NXS, TLB Invalidate by VA, EL1

The TLBI VAE1 and TLBI VAE1NXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If $HCR\_EL2\_{[E2H, TGE]}$ is not $\{1, 1\}$, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If $HCR\_EL2\_{[E2H, TGE]}$ is $\{1, 1\}$, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of $SCR\_EL3\_{[NS]}$ if FEAT_RME is not implemented, or $SCR\_EL3\_{[NSE, NS]}$ if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VAE1, TLBI VAE1NXS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>ASID</th>
<th>TTL</th>
<th>VA[55:12]</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>48</td>
<td>47</td>
</tr>
<tr>
<td>46</td>
<td>44</td>
<td>33</td>
</tr>
<tr>
<td>32</td>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

**ASID, bits [63:48]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.
Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

**TTL, bits [47:44]**

*When FEAT_TTL is implemented:*

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

- **0b00xx**  No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
- **0b01xx**  The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - **0b00**: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - **0b01**: Level 1.
  - **0b10**: Level 2.
  - **0b11**: Level 3.
- **0b10xx**  The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - **0b00**: Reserved. Treat as if TTL<3:2> is 0b00.
  - **0b01**: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
  - **0b10**: Level 2.
  - **0b11**: Level 3.
- **0b11xx**  The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - **0b00**: Reserved. Treat as if TTL<3:2> is 0b00.
  - **0b01**: Level 1.
  - **0b10**: Level 2.
  - **0b11**: Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

*Otherwise:*

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

**Executing TLBI VAE1, TLBI VAE1NXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:
**TLBI VAE1({, <Xt>})**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
   UNDEFINED;
elsif PSTATE.EL == EL1 then
   if EL2Enabled() && HCR_EL2.TTLB == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTen == '1') && HFGITR_EL2.TLBIVAE1 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif EL2Enabled() && HCR_EL2.FB == '1' then
      if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && HCRX_EL2.FnXS == '1' then
         AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
      else
         AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
      end if
   else
      if PSTATE.EL == EL2 then
         if HCR_EL2.<E2H,TGE> == '11' then
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NS, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
         else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NS, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
         end if
      elsif PSTATE.EL == EL3 then
         if HCR_EL2.<E2H,TGE> == '11' then
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NS, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
         else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NS, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
         end if
      end if
   end if
elsif PSTATE.EL == EL2 then
   if HCR_EL2.<E2H,TGE> == '11' then
      AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NS, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
   else
      AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NS, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
   end if
elsif PSTATE.EL == EL3 then
   if HCR_EL2.<E2H,TGE> == '11' then
      AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NS, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
   else
      AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NS, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
   end if
end if

**TLBI VAE1NXS({, <Xt>})**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0111</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
   UNDEFINED;
elsif PSTATE.EL == EL0 then
   UNDEFINED;
elsif PSTATE.EL == EL1 then
   if EL2Enabled() && HCR_EL2.TTLB == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTen == '1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FnXS == '0') && HFGITR_EL2.TLBIVAE1 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif EL2Enabled() && HCR_EL2.FB == '1' then
      if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && HCRX_EL2.FnXS == '1' then
         AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
      else
         AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
      end if
   else
      if PSTATE.EL == EL2 then
         if HCR_EL2.<E2H,TGE> == '11' then
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NS, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
         else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NS, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
         end if
      elsif PSTATE.EL == EL3 then
         if HCR_EL2.<E2H,TGE> == '11' then
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NS, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
         else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NS, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
         end if
      end if
   end if
elsif PSTATE.EL == EL2 then
   if HCR_EL2.<E2H,TGE> == '11' then
      AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NS, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
   else
      AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NS, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
   end if
elsif PSTATE.EL == EL3 then
   if HCR_EL2.<E2H,TGE> == '11' then
      AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NS, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
   else
      AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NS, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
   end if
end if
TLBI_ExcludeXS, X[t, 64]);
else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Any,
        TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any,
        TLBI_ExcludeXS, X[t, 64]);
    endif
else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
    if PSTATE.EL == EL3 then
        if HCR_EL2.<E2H,TGE> == '11' then
            AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Any,
            TLBI_ExcludeXS, X[t, 64]);
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any,
            TLBI_ExcludeXS, X[t, 64]);
        endif
    endif
else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
The TLBI VAE1IS and TLBI VAE1ISNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

**Note**

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined. Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VAE1IS, TLBI VAE1ISNXS is a 64-bit System instruction.
Field descriptions

**ASID, bits [63:48]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

**TTL, bits [47:44]**

*When FEAT_TTL is implemented:*

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

- **0b00xx**: No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
- **0b01xx**: The entry comes from a 4KB translation granule. The level of walk for the leaf level 0b0xx is encoded as:
  - **0b00**: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - **0b01**: Level 1.
  - **0b10**: Level 2.
  - **0b11**: Level 3.
- **0b10xx**: The entry comes from a 16KB translation granule. The level of walk for the leaf level 0b0xx is encoded as:
  - **0b00**: Reserved. Treat as if TTL<3:2> is 0b00.
  - **0b01**: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
  - **0b10**: Level 2.
  - **0b11**: Level 3.
- **0b11xx**: The entry comes from a 64KB translation granule. The level of walk for the leaf level 0b0xx is encoded as:
  - **0b00**: Reserved. Treat as if TTL<3:2> is 0b00.
  - **0b01**: Level 1.
  - **0b10**: Level 2.
  - **0b11**: Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

*Otherwise:*

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.
The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

**Executing TLBI VAE1IS, TLBI VAE1ISNXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI VAE1IS\{, <Xt>\}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & HCR_EL2.TTLBIS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.GFTEn == '1') & HFGITR_EL2.TLBIVAE1IS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if IsFeatureImplemented(FEAT_XS) & IsFeatureImplemented(FEAT_HCX) & IsHCRXEL2Enabled() &
    HCRX_EL2.FnXS == '1' then
      AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
      TLBI_ExcludeXS, X[t, 64]);
    else
      AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
      TLBI_AllAttr, X[t, 64]);
    elsif PSTATE.EL == EL2 then
      if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
        TLBI_AllAttr, X[t, 64]);
      else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
        TLBI_AllAttr, X[t, 64]);
      elsif PSTATE.EL == EL3 then
        if HCR_EL2.<E2H,TGE> == '11' then
          AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
          TLBI_AllAttr, X[t, 64]);
        else
          AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
          TLBI_AllAttr, X[t, 64]);
      end
    end
TLBI VAE1ISNXS( "<Xt>"

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&
    (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIVAE1IS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
else
  AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
  TLBI_ExcludeXS, X[t, 64]);
C5.5.61 TLBI VAE1OS, TLBI VAE1OSNXS, TLB Invalidate by VA, EL1, Outer Shareable

The TLBI VAE1OS and TLBI VAE1OSNXS characteristics are:

**Purpose**

Invalidate cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If HCR_EL2.{E2H, TGE} is not \{1, 1\}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If HCR_EL2.{E2H, TGE} is \{1, 1\}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

**Note**

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VAE1OS, TLBI VAE1OSNXS are undefined.
Attributes

TLBI VAE1OS, TLBI VAE1OSNXS is a 64-bit System instruction.

Field descriptions

<table>
<thead>
<tr>
<th>ASID</th>
<th></th>
<th>TTL</th>
<th></th>
<th>VA[55:12]</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>48</td>
<td>47</td>
<td>44</td>
<td>32</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
<td>30</td>
<td>29</td>
<td>28</td>
</tr>
</tbody>
</table>

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

<table>
<thead>
<tr>
<th>TTL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00xx</td>
<td>No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL&lt;1:0&gt; is RES0.</td>
</tr>
<tr>
<td>0b01xx</td>
<td>The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL&lt;3:2&gt; is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.</td>
</tr>
<tr>
<td>0b10xx</td>
<td>The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL&lt;3:2&gt; is 0b00. 0b01 : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL&lt;3:2&gt; is 0b00. 0b10 : Level 2. 0b11 : Level 3.</td>
</tr>
<tr>
<td>0b11xx</td>
<td>The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL&lt;3:2&gt; is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.</td>
</tr>
</tbody>
</table>

Otherwise:

Reserved, RES0.
**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

**Executing TLBI VAE1OS, TLBI VAE1OSNXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI VAE1OS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
        HCRX_EL2.FnXS == '1' then
      AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
                      TLBI_ExcludeXS, X[t, 64]);
    else
      AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
                      TLBI_AllAttr, X[t, 64]);
    end

  elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
      AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any,
                      TLBI_AllAttr, X[t, 64]);
    else
      AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any,
                      TLBI_AllAttr, X[t, 64]);
    end

  elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
      AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any,
                      TLBI_AllAttr, X[t, 64]);
    else
      AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any,
                      TLBI_AllAttr, X[t, 64]);
    end
```

C5-1072  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. ARM DDI 0487I.a Non-Confidential

ID081822
### TLBI VAE1OSNXS{, <Xt>}:

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&
    (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIVAE1OS == '1') then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
  endfi;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
  endfi;
elsif PSTATE.EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Any,
    TLBI_ExcludeXS, X[t, 64]);
  endfi;
### C5.5.62 TLBI VAE2, TLBI VAE2NXS, TLB Invalidate by VA, EL2

The TLBI VAE2 and TLBI VAE2NXS characteristics are:

**Purpose**

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be required to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the current value of the HCR_EL2.E2H bit, for the Security state.
- If HCR_EL2.E2H == 0, the entry is from any level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VAE2, TLBI VAE2NXS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>ASID, bits [63:48]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.</td>
</tr>
<tr>
<td>Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.</td>
</tr>
<tr>
<td>If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.</td>
</tr>
</tbody>
</table>
TTL, bits [47:44]

When **FEAT_TTL** is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

- **0b00xx**   No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
- **0b01xx**   The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - **0b00**: If **FEAT_LPA2** is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - **0b01**: Level 1.
  - **0b10**: Level 2.
  - **0b11**: Level 3.
- **0b10xx**   The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - **0b00**: Reserved. Treat as if TTL<3:2> is 0b00.
  - **0b01**: If **FEAT_LPA2** is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
  - **0b10**: Level 2.
  - **0b11**: Level 3.
- **0b11xx**   The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - **0b00**: Reserved. Treat as if TTL<3:2> is 0b00.
  - **0b01**: Level 1.
  - **0b10**: Level 2.
  - **0b11**: Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

**Executing TLBI VAE2, TLBI VAE2NXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:
### TLBI VAE2(, <Xr>)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);

### TLBI VAE2NXS(, <Xr>)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NSH, TLBILevel_Any,
TLBI_ExcludeXS, X[t, 64]);
C5.5.63 TLBI VAE2IS, TLBI VAE2ISNXS, TLB Invalidate by VA, EL2, Inner Shareable

The TLBI VAE2IS and TLBI VAE2ISNXS characteristics are:

**Purpose**

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be required to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the current value of the HCR_EL2.E2H bit, for the Security state.
- If HCR_EL2.E2H == 0, the entry is from any level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VAE2IS, TLBI VAE2ISNXS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>ASID</td>
</tr>
<tr>
<td>48</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>TTL</td>
</tr>
<tr>
<td>43</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>VA[55:12]</td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>VA[55:12]</td>
</tr>
</tbody>
</table>

**ASID, bits [63:48]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.
TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

0b00xx No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

0b01xx The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:

0b00 : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
0b01 : Level 1.
0b10 : Level 2.
0b11 : Level 3.

0b10xx The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:

0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
0b01 : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
0b10 : Level 2.
0b11 : Level 3.

0b11xx The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:

0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
0b01 : Level 1.
0b10 : Level 2.
0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE2IS, TLBI VAE2ISNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:
TLBI VAE2IS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
                    TLBI_AllAttr, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Any,
                    TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    UNDEFINED;
elsif HCR_EL2.E2H == '1' then
  AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
                  TLBI_AllAttr, X[t, 64]);
else
  AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Any,
                  TLBI_AllAttr, X[t, 64]);

TLBI VAE2ISNXS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
                    TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Any,
                    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    UNDEFINED;
elsif HCR_EL2.E2H == '1' then
  AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Any,
                  TLBI_ExcludeXS, X[t, 64]);
else
  AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Any,
                  TLBI_ExcludeXS, X[t, 64]);
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

C5.5.64 TLBI VAE2OS, TLBI VAE2OSNXS, TLB Invalidate by VA, EL2, Outer Shareable

The TLBI VAE2OS and TLBI VAE2OSNXS characteristics are:

**Purpose**

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be required to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the current value of the HCR_EL2.E2H bit, for the Security state.
- If HCR_EL2.E2H == 0, the entry is from any level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is from a level of the translation table walk above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VAE2OS, TLBI VAE2OSNXS are UNDEFINED.

**Attributes**

TLBI VAE2OS, TLBI VAE2OSNXS is a 64-bit System instruction.

**Field descriptions**

```
<table>
<thead>
<tr>
<th>ASID</th>
<th>TTL</th>
<th>VA[55:12]</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>48</td>
<td>47</td>
</tr>
<tr>
<td>44</td>
<td>43</td>
<td>32</td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**ASID, bits [63:48]**

*When HCR_EL2.E2H == 1:*

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.
Otherwise:
Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:
Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

0b00xx No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

0b01xx The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
0b00: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
0b01: Level 1.
0b10: Level 2.
0b11: Level 3.

0b10xx The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
0b00: Reserved. Treat as if TTL<3:2> is 0b00.
0b01: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
0b10: Level 2.
0b11: Level 3.

0b11xx The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
0b00: Reserved. Treat as if TTL<3:2> is 0b00.
0b01: Level 1.
0b10: Level 2.
0b11: Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:
Reserved, RES0.

VA[55:12], bits [43:0]
Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

• Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
• Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
• Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE2OS, TLBI VAE2OSNXS instruction
Accesses to this instruction use the following encodings in the System instruction encoding space:
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

**TLBI VAЕ2OS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any,
                   TLBI_AllAttr, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBILevel_Any,
                   TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any,
                   TLBI_AllAttr, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBILevel_Any,
                   TLBI_AllAttr, X[t, 64]);

**TLBI VAЕ2OSNXS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any,
                   TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBILevel_Any,
                   TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Any,
                   TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBILevel_Any,
                   TLBI_ExcludeXS, X[t, 64]);
  else

C5-1084  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. ARM DDI 0487l.a Non-Confidential
AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
The TLBI VAE3, TLBI VAE3NXS, TLB Invalidate by VA, EL3

The TLBI VAE3 and TLBI VAE3NXS characteristics are:

**Purpose**

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VAE3, TLBI VAE3NXS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:48]</td>
<td>RES0</td>
</tr>
<tr>
<td>[47:44]</td>
<td>TTL</td>
</tr>
<tr>
<td>[31:0]</td>
<td>VA[55:12]</td>
</tr>
</tbody>
</table>

**Bits [63:48]**

Reserved, RES0.

**TTL, bits [47:44]**

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

- 0b00xx: No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
- 0b01xx: The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b01: Level 1.
  - 0b10: Level 2.
  - 0b11: Level 3.
- 0b10xx: The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b10: Level 2.
0b11: Level 3.
0b11xx The entry comes from a 64KB translation granule. The level of walk for the leaf level
0bxx is encoded as:
0b00: Reserved. Treat as if TTL<3:2> is 0b00.
0b01: Level 1.
0b10: Level 2.
0b11: Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction,
then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:
Reserved, RES0.

VA[55:12], bits [43:0]
Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value
(if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and
so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:
• Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
• Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored
when the instruction is executed, because VA[13:12] have no effect on the operation of the
instruction.
• Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored
when the instruction is executed, because VA[15:12] have no effect on the operation of the
instruction.

Executing TLBI VAE3, TLBI VAE3NXS instruction
Accesses to this instruction use the following encodings in the System instruction encoding space:

| TLBI VAE3{, <Xt>} |
|-------------------|--|--|--|--|
| op0   | op1 | CRn   | CRm | op2 |
| 0b01  | 0b11 | 0b1000 | 0b0111 | 0b001 |

if PSTATE.EL == EL0 then
  UNDEFINED;
elsiF PSTATE.EL == EL1 then
  UNDEFINED;
elsiF PSTATE.EL == EL2 then
  UNDEFINED;
elsiF PSTATE.EL == EL3 then
  AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_NSH, TLBILevel_Any,
  TLBI_AllAttr, X[t, 64]);
**TLBI VAE3NXS\{<Xt>\}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1001</td>
<td>0b0111</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_NSH, TLBLevel_Any, TLBI_ExcludeXS, X[t, 64]);
C5.5.66 TLBI VAE3IS, TLBI VAE3ISNXS, TLB Invalidate by VA, EL3, Inner Shareable

The TLBI VAE3IS and TLBI VAE3ISNXS characteristics are:

**Purpose**

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VAE3IS, TLBI VAE3ISNXS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:48]</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>[47:44]</td>
<td>TTL</td>
</tr>
<tr>
<td>[31:0]</td>
<td>VA[55:12]</td>
</tr>
</tbody>
</table>

**Bits [63:48]**

Reserved, RES0.

**TTL, bits [47:44]**

*When FEAT_TTL is implemented:*

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

- **0b00xx** No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
- **0b01xx** The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - **0b00**: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - **0b01**: Level 1.
  - **0b10**: Level 2.
  - **0b11**: Level 3.
- **0b10xx** The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - **0b00**: Reserved. Treat as if TTL<3:2> is 0b00.
  - **0b01**: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
  - **0b10**: Level 2.
0b11: Level 3.

0b11xx  The entry comes from a 64KB translation granule. The level of walk for the leaf level
0xxx is encoded as:
0b00: Reserved. Treat as if TTL<3:2> is 0b00.
0b01: Level 1.
0b10: Level 2.
0b11: Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction,
then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:
Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value
(if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and
so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:
• Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
• Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored
  when the instruction is executed, because VA[13:12] have no effect on the operation of the
  instruction.
• Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored
  when the instruction is executed, because VA[15:12] have no effect on the operation of the
  instruction.

Executing TLBI VAE3IS, TLBI VAE3ISNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

TLBI VAE3IS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_ISH, TLBLevel_Any,
    TLBI_AllAttr, X[t, 64]);
TLBI VAE3ISNXS(, <Xt>)

```c
if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    UNDEFINED;
elsif PSTATE_EL == EL2 then
    UNDEFINED;
elsif PSTATE_EL == EL3 then
    AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_ExcludeXS, X[t, 64]);
```

---

**TLBI VAE3ISNXS(, <Xt>)**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1001</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>
C5.5.67 TLBI VAE3OS, TLBI VAE3OSNXS, TLB Invalidate by VA, EL3, Outer Shareable

The TLBI VAE3OS and TLBI VAE3OSNXS characteristics are:

**Purpose**

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VAE3OS, TLBI VAE3OSNXS are UNDEFINED.

**Attributes**

TLBI VAE3OS, TLBI VAE3OSNXS is a 64-bit System instruction.

**Field descriptions**

![Field descriptions diagram]

**Bits [63:48]**

Reserved, RES0.

**TTL, bits [47:44]**

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

- 0b00xx: No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
- 0b01xx: The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b01: Level 1.
  - 0b10: Level 2.
  - 0b11: Level 3.
- 0b10xx: The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
0b10 : Level 2.
0b11 : Level 3.

0b11xx  The entry comes from a 64KB translation granule. The level of walk for the leaf level
0bxx is encoded as:
0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
0b01 : Level 1.
0b10 : Level 2.
0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction,
then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:
Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value
(if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and
so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored
  when the instruction is executed, because VA[13:12] have no effect on the operation of the
  instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored
  when the instruction is executed, because VA[15:12] have no effect on the operation of the
  instruction.

Executing TLBI VAE3OS, TLBI VAE3OSNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI VAE3OS{, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1000</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  UNDEFINED;
elseif PSTATE.EL == EL2 then
  UNDEFINED;
elseif PSTATE.EL == EL3 then
  AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_OSH, TLBILevel_Any,
  TLBI_AllAttr, X[t, 64]);
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

TLBI VAE3OSNXS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_OSH, TLBILevel_Any, TLBIExcludeXS, X[t, 64]);
C5.5.68 TLBI VALE1, TLBI VALE1NXS, TLB Invalidate by VA, Last level, EL1

The TLBI VALE1 and TLBI VALE1NXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VALE1, TLBI VALE1NXS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>48</th>
<th>47</th>
<th>44</th>
<th>43</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASID</td>
<td>TTL</td>
<td>VA[55:12]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA[55:12]</td>
<td></td>
</tr>
</tbody>
</table>

**ASID, bits [63:48]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.
If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

**TTL, bits [47:44]**

*When FEAT_TTL is implemented:*

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

- **0b00xx** No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
- **0b01xx** The entry comes from a 4KB translation granule. The level of walk for the leaf level 0xx is encoded as:
  - 0b00 : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b01 : Level 1.
  - 0b10 : Level 2.
  - 0b11 : Level 3.
- **0b10xx** The entry comes from a 16KB translation granule. The level of walk for the leaf level 0xx is encoded as:
  - 0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01 : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b10 : Level 2.
  - 0b11 : Level 3.
- **0b11xx** The entry comes from a 64KB translation granule. The level of walk for the leaf level 0xx is encoded as:
  - 0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01 : Level 1.
  - 0b10 : Level 2.
  - 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

*Otherwise:*

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

**Executing TLBI VALE1, TLBI VALE1NXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:
TLBI VALE1{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTrEn == '1') && HFCITR_EL2.TLBIVALE1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.FB == '1' then
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && HCRX_EL2.FnXS == '1' then
      AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBIVALE1, TLBI_ExcludeXS, X[t, 64]);
    else
      AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBIVALE1, TLBI_AllAttr, X[t, 64]);
    endif
  else
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then
      AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBIVALE1, TLBI_ExcludeXS, X[t, 64]);
    else
      AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBIVALE1, TLBI_AllAttr, X[t, 64]);
    endif
  endif
else
  if PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
      AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBIVALE1, TLBI_AllAttr, X[t, 64]);
    else
      AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBIVALE1, TLBI_AllAttr, X[t, 64]);
    endif
  elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
      AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBIVALE1, TLBI_AllAttr, X[t, 64]);
    else
      AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBIVALE1, TLBI_AllAttr, X[t, 64]);
    endif
  else
    if !IsFeatureImplemented(FEAT_XS) then
      UNDEFINED;
    elsif PSTATE.EL == EL0 then
      UNDEFINED;
    elsif PSTATE.EL == EL1 then
      if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTrEn == '1') && HFCITR_EL2.TLBIVALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBIVALE1, TLBI_AllAttr, X[t, 64]);
      endif
    endif
  endif
endif

TLBI VALE1NXS{, <Xp>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0111</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTrEn == '1') && IsFeatureImplemented(FEAT_HCX) && !IsHCRXEL2Enabled() && HCRX_EL2.FGTrNsX == '0' && HFCITR_EL2.TLBIVALE1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.FB == '1' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBIVALE1, TLBI_AllAttr, X[t, 64]);
  else
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then
      AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBIVALE1, TLBI_ExcludeXS, X[t, 64]);
    else
      AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBIVALE1, TLBI_AllAttr, X[t, 64]);
    endif
  endif
else
  if PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
      AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBIVALE1, TLBI_AllAttr, X[t, 64]);
    else
      AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBIVALE1, TLBI_AllAttr, X[t, 64]);
    endif
  elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
      AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBIVALE1, TLBI_AllAttr, X[t, 64]);
    else
      AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBIVALE1, TLBI_AllAttr, X[t, 64]);
    endif
  else
    if !IsFeatureImplemented(FEAT_XS) then
      UNDEFINED;
    elsif PSTATE.EL == EL0 then
      UNDEFINED;
    elsif PSTATE.EL == EL1 then
      if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTrEn == '1') && HFCITR_EL2.TLBIVALE1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBIVALE1, TLBI_AllAttr, X[t, 64]);
      endif
    endif
  endif
endif

ARM DDI 0487I.a
Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5-1097
ID081822
TLBI_ExcludeXS, X[t, 64]);
else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H, TGE> == '11' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Last,
        TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last,
        TLBI_ExcludeXS, X[t, 64]);
    endif
elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H, TGE> == '11' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBILevel_Last,
        TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last,
        TLBI_ExcludeXS, X[t, 64]);
    endif
endif
C5.5.69 TLBI VALE1IS, TLBI VALE1ISNXS, TLB Invalidate by VA, Last level, EL1, Inner Shareable

The TLBI VALE1IS and TLBI VALE1ISNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If \( HCR_{EL2}.\{E2H, TGE\} \) is not \( \{1,1\} \), the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If \( HCR_{EL2}.\{E2H, TGE\} \) is \( \{1,1\} \), the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of \( SCR_{EL3}.NS \) if \( FEAT\_RME \) is not implemented, or \( SCR_{EL3}.\{NSE, NS\} \) if \( FEAT\_RME \) is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

**Note**

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if \( SCR_{EL3}.EEL2==1 \), then:

- A PE with \( SCR_{EL3}.EEL2==1 \) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \( SCR_{EL3}.EEL2==0 \).

- A PE with \( SCR_{EL3}.EEL2==0 \) is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with \( SCR_{EL3}.EEL2==1 \).

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If \( FEAT\_XS \) is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VALE1IS, TLBI VALE1ISNXS is a 64-bit System instruction.
Field descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASID</td>
<td>[63:48]</td>
<td>ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction. Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field. If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.</td>
</tr>
<tr>
<td>TTL</td>
<td>[47:44]</td>
<td>When FEAT_TTL is implemented: Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated. 0b00xx: No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL&lt;1:0&gt; is RES0. 0b01xx: The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL&lt;3:2&gt; is 0b00. 0b01: Level 1. 0b10: Level 2. 0b11: Level 3. 0b10xx: The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00: Reserved. Treat as if TTL&lt;3:2&gt; is 0b00. 0b01: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL&lt;3:2&gt; is 0b00. 0b10: Level 2. 0b11: Level 3. 0b11xx: The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00: Reserved. Treat as if TTL&lt;3:2&gt; is 0b00. 0b01: Level 1. 0b10: Level 2. 0b11: Level 3. If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB. Otherwise: Reserved, RES0.</td>
</tr>
<tr>
<td>VA[55:12]</td>
<td>[43:0]</td>
<td>Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction. If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.</td>
</tr>
</tbody>
</table>
The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

### Executing TLBI VALE1IS, TLBI VALE1ISNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{TLBI VALE1IS, } \langle \text{Xt} \rangle
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elifs PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elifs PSTATE.EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBLevel_Last, TLBI_AllAttr, X[t, 64]);
eelse
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Shareability_ISH, TLBLevel_Last, TLBI_AllAttr, X[t, 64]);
elifs PSTATE.EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBLevel_Last, TLBI_AllAttr, X[t, 64]);
eelse
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Shareability_ISH, TLBLevel_Last, TLBI_AllAttr, X[t, 64]);

The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

TLBI VAEL1ISNXS{, <xT>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0011</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&
  (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIVALE1IS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
  endif
elsif PSTATE_EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
  endif
elsif PSTATE_EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last,
    TLBI_ExcludeXS, X[t, 64]);
C5.5.70 TLBI VALE1OS, TLBI VALE1OSNXS, TLB Invalidate by VA, Last level, EL1, Outer Shareable

The TLBI VALE1OS and TLBI VALE1OSNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- When EL2 is implemented and enabled in the current Security state:
  - If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

**Note**

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VALE1OS, TLBI VALE1OSNXS are UNDEFINED.

**Attributes**

TLBI VALE1OS, TLBI VALE1OSNXS is a 64-bit System instruction.
Field descriptions

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:
Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

0b00xx No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

0b01xx The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
   0b00: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
   0b01: Level 1.
   0b10: Level 2.
   0b11: Level 3.

0b10xx The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
   0b00: Reserved. Treat as if TTL<3:2> is 0b00.
   0b01: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
   0b10: Level 2.
   0b11: Level 3.

0b11xx The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
   0b00: Reserved. Treat as if TTL<3:2> is 0b00.
   0b01: Level 1.
   0b10: Level 2.
   0b11: Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:
Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.
The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE1OS, TLBI VALE1OSNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{TLBI VALE1OS} \{<Xt>\}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0001</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TTL == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.TTL0S == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTE == '1') & HFGITR_EL2.TLBIVALE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_XS) & IsFeatureImplemented(FEAT_HCX) & IsHCRXEL2Enabled() &
            HCRX_EL2.FnXS == '1' then
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH,
            TLBILevel_Last, TLBI_ExpandX, X[t, 64]);
        else
            AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH,
            TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
        end if
    end if
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last,
        TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last,
        TLBI_AllAttr, X[t, 64]);
    end if
elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_OSH, TLBILevel_Last,
        TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_OSH, TLBILevel_Last,
        TLBI_AllAttr, X[t, 64]);
    end if
TLBI VALE1OSNXS(<X[t]>)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) &&
    (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIVALE1OS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last,
      TLBI_ExcludeXS, X[t, 64]);
  endif
elsif PSTATE.EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last,
      TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last,
      TLBI_ExcludeXS, X[t, 64]);
  endif
elsif PSTATE.EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last,
      TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBILevel_Last,
      TLBI_ExcludeXS, X[t, 64]);
  endif
TLBI VALE2, TLBI VALE2NXS, TLB Invalidate by VA, Last level, EL2

The TLBI VALE2 and TLBI VALE2NXS characteristics are:

**Purpose**

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the current value of the HCR_EL2.E2H bit, for the Security state.
- If HCR_EL2.E2H == 0, the entry is from the final level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined. Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VALE2, TLBI VALE2NXS is a 64-bit System instruction.

**Field descriptions**

\[
\begin{array}{c|c|c}
|   & ASID & TTL | VA[55:12] \\
\hline
|   & 48 | 47 | 44 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
\end{array}
\]

ASID, bits [63:48]

*When HCR_EL2.E2H == 1:*

- ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.
- Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.
- If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

*Otherwise:*

- Reserved, RES0.
TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for
the address being invalidated.

- **0b00xx**: No information supplied as to the translation table level. Hardware must assume that the
  entry can be from any level. In this case, TTL<1:0> is RES0.
- **0b01xx**: The entry comes from a 4KB translation granule. The level of walk for the leaf level
  0bxx is encoded as:
    - **0b00**: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
    - **0b01**: Level 1.
    - **0b10**: Level 2.
    - **0b11**: Level 3.
- **0b10xx**: The entry comes from a 16KB translation granule. The level of walk for the leaf level
  0bxx is encoded as:
    - **0b00**: Reserved. Treat as if TTL<3:2> is 0b00.
    - **0b01**: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
    - **0b10**: Level 2.
    - **0b11**: Level 3.
- **0b11xx**: The entry comes from a 64KB translation granule. The level of walk for the leaf level
  0bxx is encoded as:
    - **0b00**: Reserved. Treat as if TTL<3:2> is 0b00.
    - **0b01**: Level 1.
    - **0b10**: Level 2.
    - **0b11**: Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction,
then no entries are required by the architecture to be invalidated from the TLB.

**Otherwise:**

Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value
(if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and
so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored
  when the instruction is executed, because VA[13:12] have no effect on the operation of the
  instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored
  when the instruction is executed, because VA[15:12] have no effect on the operation of the
  instruction.

**Executing TLBI VALE2, TLBI VALE2NXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:
**TLBI VALE2(, <Xt>)**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsiF PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsiF PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NS, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NS, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
eIFl PSTATE.EL == EL3 then
    fiF !EL2Enabled() then
        UNDEFINED;
elsiF HCR_EL2.E2H == '1' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NS, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NS, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);

**TLBI VALE2NXS(, <Xt>)**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsiF PSTATE.EL == EL0 then
    UNDEFINED;
elsiF PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsiF PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NS, TLBILevel_Last, TLBIExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NS, TLBILevel_Last, TLBIExcludeXS, X[t, 64]);
eIFl PSTATE.EL == EL3 then
    fiF !EL2Enabled() then
        UNDEFINED;
elsiF HCR_EL2.E2H == '1' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NS, TLBILevel_Last, TLBIExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NS, TLBILevel_Last, TLBIExcludeXS, X[t, 64]);
AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
C5.5.72 TLBI VALE2IS, TLBI VALE2ISNXS, TLB Invalidate by VA, Last level, EL2, Inner Shareable

The TLBI VALE2IS and TLBI VALE2ISNXS characteristics are:

**Purpose**

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the current value of the HCR_EL2.E2H bit, for the Security state.
- If HCR_EL2.E2H == 0, the entry is from the final level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VALE2IS, TLBI VALE2ISNXS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>40</th>
<th>47</th>
<th>44</th>
<th>43</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASID</td>
<td>TTL</td>
<td>VA[55:12]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>20</td>
<td>17</td>
<td>14</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>VA[55:12]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ASID, bits [63:48]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.
TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

0b00xx  No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

0b01xx  The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
          0b00 : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
          0b01 : Level 1.
          0b10 : Level 2.
          0b11 : Level 3.

0b10xx  The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
          0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
          0b01 : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
          0b10 : Level 2.
          0b11 : Level 3.

0b11xx  The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
          0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
          0b01 : Level 1.
          0b10 : Level 2.
          0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

• Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
• Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
• Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE2IS, TLBI VALE2ISNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:
**TLBI VALE2IS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
  else
    AArch64.TLBI VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
elsif PSTATE_EL == EL3 then
  if !EL2Enabled() then
    UNDEFINED;
  elsif HCR_EL2.E2H == '1' then
    AArch64.TLBI VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
  else
    AArch64.TLBI VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, X[t, 64]);

**TLBI VALE2ISNXS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AArch64.TLBI VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE_EL == EL3 then
  if !EL2Enabled() then
    UNDEFINED;
  elsif HCR_EL2.E2H == '1' then
    AArch64.TLBI VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
  else
    AArch64.TLBI VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
  else

AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
C5.5.73  TLBI VALE2OS, TLBI VALE2OSNXS, TLB Invalidate by VA, Last level, EL2, Outer Shareable

The TLBI VALE2OS and TLBI VALE2OSNXS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA using the EL2 or EL2&0 translation regime, as determined by the current value of the HCR_EL2.E2H bit, for the Security state.
- If HCR_EL2.E2H == 0, the entry is from the final level of the translation table walk.
- If HCR_EL2.E2H == 1, one of the following applies:
  - The entry is a global entry from the final level of the translation table walk.
  - The entry is a non-global entry from the final level of the translation table walk that matches the specified ASID.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configurations

This instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VALE2OS, TLBI VALE2OSNXS are UNDEFINED.

Attributes

TLBI VALE2OS, TLBI VALE2OSNXS is a 64-bit System instruction.

Field descriptions

<table>
<thead>
<tr>
<th>63</th>
<th>48</th>
<th>47</th>
<th>44</th>
<th>43</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASID</td>
<td>TTL</td>
<td>VA[55:12]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ASID, bits [63:48]

When HCR_EL2.E2H == 1:

- ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.
- Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.
- If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Otherwise:

- Reserved, RES0.
TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for
the address being invalidated.

0b00xx  No information supplied as to the translation table level. Hardware must assume that the
entry can be from any level. In this case, TTL<1:0> is RES0.

0b01xx  The entry comes from a 4KB translation granule. The level of walk for the leaf level
0bxx is encoded as:
    0b00 : If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
    0b01 : Level 1.
    0b10 : Level 2.
    0b11 : Level 3.

0b10xx  The entry comes from a 16KB translation granule. The level of walk for the leaf level
0bxx is encoded as:
    0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
    0b01 : If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
    0b10 : Level 2.
    0b11 : Level 3.

0b11xx  The entry comes from a 64KB translation granule. The level of walk for the leaf level
0bxx is encoded as:
    0b00 : Reserved. Treat as if TTL<3:2> is 0b00.
    0b01 : Level 1.
    0b10 : Level 2.
    0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction,
then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value
(if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and
so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored
  when the instruction is executed, because VA[13:12] have no effect on the operation of the
  instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored
  when the instruction is executed, because VA[15:12] have no effect on the operation of the
  instruction.

Executing TLBI VALE2OS, TLBI VALE2OSNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:
TLBI VALE2OS{, <Xt>}

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last,
                        TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBILevel_Last,
                        TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
elsif HCR_EL2.E2H == '1' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last,
                        TLBI_ExcludeXS, X[t, 64]);
else
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBILevel_Last,
                        TLBI_ExcludeXS, X[t, 64]);

TLBI VALE2OSNXS{, <Xt>}

if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last,
                        TLBI_ExcludeXS, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBILevel_Last,
                        TLBI_ExcludeXS, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
elsif HCR_EL2.E2H == '1' then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBILevel_Last,
                        TLBI_ExcludeXS, X[t, 64]);
else
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBILevel_Last,
                        TLBI_ExcludeXS, X[t, 64]);
AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[], Shareability_OSH, TLBLevel_Last,
TLBI_ExcludeXS, X[t, 64]);
C5.5.74 TLBI VALE3, TLBI VALE3NXS, TLB Invalidate by VA, Last level, EL3

The TLBI VALE3 and TLBI VALE3NXS characteristics are:

**Purpose**

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VALE3, TLBI VALE3NXS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-48</td>
<td>RES0</td>
</tr>
<tr>
<td>47-44</td>
<td>TTL</td>
</tr>
<tr>
<td>43-32</td>
<td>VA[55:12]</td>
</tr>
<tr>
<td>31-0</td>
<td>VA[55:12]</td>
</tr>
</tbody>
</table>

**Bits [63:48]**

Reserved, RES0.

**TTL, bits [47:44]**

*When FEAT_TTL is implemented:*

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

- **0b00xx**: No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
- **0b01xx**: The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - **0b00**: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - **0b01**: Level 1.
  - **0b10**: Level 2.
  - **0b11**: Level 3.
- **0b10xx**: The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - **0b00**: Reserved. Treat as if TTL<3:2> is 0b00.
  - **0b01**: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
  - **0b10**: Level 2.
The entry comes from a 64KB translation granule. The level of walk for the leaf level
0bxx is encoded as:

- 0b00: Reserved. Treat as if TTL<3:2> is 0b00.
- 0b01: Level 1.
- 0b10: Level 2.
- 0b11: Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction,
then no entries are required by the architecture to be invalidated from the TLB.

**Otherwise:**

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value
(if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and
so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored
  when the instruction is executed, because VA[13:12] have no effect on the operation of the
  instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored
  when the instruction is executed, because VA[15:12] have no effect on the operation of the
  instruction.

**Executing TLBI VALE3, TLBI VALE3NXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI VALE3{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_NSH, TLBILevel_Last,
  TLBI_AllAttr, X[t, 64]);
TLBI VAELNXS\{, <Xt>\}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1001</td>
<td>0b0111</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_ExcludeXS, X[t, 64]);
C5.5.75 TLBI VALE3IS, TLBI VALE3ISNXS, TLB Invalidate by VA, Last level, EL3, Inner Shareable

The TLBI VALE3IS and TLBI VALE3ISNXS characteristics are:

**Purpose**

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VALE3IS, TLBI VALE3ISNXS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-48</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>47-44</td>
<td>TTL</td>
</tr>
<tr>
<td>43-32</td>
<td>VA[55:12]</td>
</tr>
<tr>
<td>31-0</td>
<td>VA[55:12]</td>
</tr>
</tbody>
</table>

**Bits [63:48]**

Reserved, RES0.

**TTL, bits [47:44]**

*When FEAT_TTL is implemented:*

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

- **0b00xx** No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
- **0b01xx** The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b01: Level 1.
  - 0b10: Level 2.
  - 0b11: Level 3.
- **0b10xx** The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b10: Level 2.
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

0b11: Level 3.
0b11xx The entry comes from a 64KB translation granule. The level of walk for the leaf level
0bxx is encoded as:
  0b00: Reserved. Treat as if TTL<3:2> is 0b00.
  0b01: Level 1.
  0b10: Level 2.
  0b11: Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction,
then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:
Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value
(if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and
so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

• Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
• Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored
  when the instruction is executed, because VA[13:12] have no effect on the operation of the
  instruction.
• Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored
  when the instruction is executed, because VA[15:12] have no effect on the operation of the
  instruction.

Executing TLBI VALE3IS, TLBI VALE3ISNXS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI VALE3IS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_ISH, TLBLevel_Last,
  TLBI_AllAttr, X[t, 64]);
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

**TLBI VAEL3ISNXS{, <Xt>**}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1001</td>
<td>0b0011</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_ISH, TLBILevel_Last,
  TLBI_ExcludeXS, X[t, 64]);
The TLBI VALE3OS and TLBI VALE3OSNXS characteristics are:

**Purpose**

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VALE3OS, TLBI VALE3OSNXS are UNDEFINED.

**Attributes**

TLBI VALE3OS, TLBI VALE3OSNXS is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:48</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>47:44</td>
<td>TTL, indicates the level of the translation table walk</td>
</tr>
<tr>
<td>43:32</td>
<td>VA[55:12], the address being invalidated</td>
</tr>
</tbody>
</table>

**Bits [63:48]**

Reserved, RES0.

**TTL, bits [47:44]**

*When FEAT_TTL is implemented:*

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

- **0b00xx** No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
- **0b01xx** The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: If FEAT_LPA2 is implemented, level 0. Otherwise, treat as if TTL<3:2> is 0b00.
  - 0b01: Level 1.
  - 0b10: Level 2.
  - 0b11: Level 3.
- **0b10xx** The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as:
  - 0b00: Reserved. Treat as if TTL<3:2> is 0b00.
  - 0b01: If FEAT_LPA2 is implemented, level 1. Otherwise, treat as if TTL<3:2> is 0b00.
The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as:

\- 0b00: Reserved. Treat as if TTL<3:2> is 0b00.
\- 0b01: Level 1.
\- 0b10: Level 2.
\- 0b11: Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

**Otherwise:**
Reserved, RES0.

**VA[55:12], bits [43:0]**

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

If the TLB maintenance instructions are targeting a translation regime that is using AArch32, and so has a VA of only 32 bits, then the software must treat bits[55:32] as RES0.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

**Executing TLBI VALE3OS, TLBI VALE3OSNXS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI VALE3OS{, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1000</td>
<td>0b0001</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elif PSTATE.EL == EL1 then
    UNDEFINED;
elif PSTATE.EL == EL2 then
    UNDEFINED;
elif PSTATE.EL == EL3 then
    AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_OSH, TLBLevel_Last, TLBI_AllAttr, X[<t, 64>]);
TLBI VALE3OSNXS{, <X[t]>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b110</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AArch64.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL3, VMID[], Shareability_OSH, TLBILevel_Last, TLBIExcludeXS, X[t, 64]);
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

C5.5.77 TLBI VMALLE1, TLBI VMALLE1NXS, TLB Invalidate by VMID, All at stage 1, EL1

The TLBI VMALLE1 and TLBI VMALLE1NXS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

• The entry is a stage 1 translation table entry, from any level of the translation table walk.

• When EL2 is implemented and enabled in the current Security state:
  — If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  — If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.

• When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configurations

There are no configuration notes.

Attributes

TLBI VMALLE1, TLBI VMALLE1NXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLE1, TLBI VMALLE1NXS instruction

The Rt field should be set to 0b11111. If the Rt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:

• The instruction is UNDEFINED.

• The instruction behaves as if the Rt field is set to 0b11111.

Accesses to this instruction use the following encodings in the System instruction encoding space:
### TLBI VMALLE1{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.TLBIVMALLE1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_ExcludeXS);
  endif
else
  AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_AllAttr);
endif

### TLBI VMALLE1NXS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.TLBIVMALLE1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_ExcludeXS);
  endif
else
  AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_AllAttr);
endif

AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_ExcludeXS);

elsif PSTATE.EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBI_ExcludeXS);
  else
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_ExcludeXS);
  end
elsif PSTATE.EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_NSH, TLBI_ExcludeXS);
  else
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_ExcludeXS);
C5.5.78 TLBI VMALLE1IS, TLBI VMALLE1ISNXS, TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable

The TLBI VMALLE1IS and TLBI VMALLE1ISNXS characteristics are:

**Purpose**

Invalidate cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the current Security state:
  - If $HCR_{EL2}\cdot E2H, TGE$ is not $\{1, 1\}$, the entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If $HCR_{EL2}\cdot E2H, TGE$ is $\{1, 1\}$, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
- The Security state is indicated by the value of $SCR_{EL3}\cdot NS$ if $FEAT_RME$ is not implemented, or $SCR_{EL3}\cdot NSE, NS$ if $FEAT_RME$ is implemented.
- The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

--- **Note**

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if $SCR_{EL3}\cdot EEL2==1$, then:

- A PE with $SCR_{EL3}\cdot EEL2==1$ is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with $SCR_{EL3}\cdot EEL2==0$.
- A PE with $SCR_{EL3}\cdot EEL2==0$ is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with $SCR_{EL3}\cdot EEL2==1$.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

--- **Note**

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

---

If $FEAT_XS$ is implemented, the nXS variant of this System instruction is defined. Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.

**Attributes**

TLBI VMALLE1IS, TLBI VMALLE1ISNXS is a 64-bit System instruction.
Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLE1IS, TLBI VMALLE1ISNXS instruction

The Rt field should be set to 0b11111. If the Rt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to 0b11111.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI VMALLE1IS{, <Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.TLBIVMALLE1IS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() &&
      HCRX_EL2.FnXS == '1' then
      AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
      TLBI_ExcludeXS);
    else
      AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
      TLBI_AllAttr);
    else
      if PSTATE.EL == EL2 then
        if HCR_EL2.<E2H,TGE> == '11' then
          AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
          TLBI_AllAttr);
        else
          AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
          TLBI_AllAttr);
        else
          if PSTATE.EL == EL3 then
            if HCR_EL2.<E2H,TGE> == '11' then
              AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
              TLBI_AllAttr);
            else
              AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
              TLBI_AllAttr);
            else
              AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
              TLBI_AllAttr);
          else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
            TLBI_AllAttr);
          else
            AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
            TLBI_AllAttr);
        else
          AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
          TLBI_AllAttr);
        else
          AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH,
          TLBI_AllAttr);
      ```
TLBI VMALLE1ISNXS\{ <Xt>\}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(_FEAT_XS) then
  UNDEFINED;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.TTLB == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.TTLBIS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(_FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIVMALLE1IS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_ExcludeXS);
  endif
elsif PSTATE_EL == EL2 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBI_ExcludeXS);
  else
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_ExcludeXS);
  endif
elsif PSTATE_EL == EL3 then
  if HCR_EL2.<E2H,TGE> == '11' then
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_ISH, TLBI_ExcludeXS);
  else
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_ExcludeXS);
  endif

The TLBI VMALLE1OS and TLBI VMALLE1OSNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- When EL2 is implemented and enabled in the current Security state:
  - If HCR_EL2.{E2H, TGE} is not \{1, 1\}, the entry would be used with the current VMD and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.
  - If HCR_EL2.{E2H, TGE} is \{1, 1\}, the entry would be required to translate the specified VA using the EL2&0 translation regime for the Security state.
- When EL2 is not implemented or is disabled in the current Security state, the entry would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The Security state is indicated by the value of SCR_EL3.NS if FEAT_RME is not implemented, or SCR_EL3.{NSE, NS} if FEAT_RME is implemented.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

**Note**

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

**Note**

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined.

Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete.

The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

This instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VMALLE1OS, TLBI VMALLE1OSNXS are **UNDEFINED**.

**Attributes**

TLBI VMALLE1OS, TLBI VMALLE1OSNXS is a 64-bit System instruction.
Field descriptions
This instruction has no applicable fields.
The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLE1OS, TLBI VMALLE1OSNXS instruction
The Rt field should be set to 0b11111. If the Rt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:
• The instruction is UNDEFINED.
• The instruction behaves as if the Rt field is set to 0b11111.
Accesses to this instruction use the following encodings in the System instruction encoding space:

TLBI VMALLE1OS(, <Xt>)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() & HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() & ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.TLBIVMALLE1OS == '1') then
        AArch64.SystemAccessTrap(EL2, 0x18);
else
    if IsFeatureImplemented(FEAT_XS) & IsFeatureImplemented(FEAT_HCX) & IsHCRXEL2Enabled() &
        HCRX_EL2.FnXS == '1' then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH,
        TLBI_ExcludeXS);
else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH,
        TLBI_AllAttr);
elif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,
        TLBI_AllAttr);
else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBI_AllAttr);
elif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH,
        TLBI_AllAttr);
else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability_OSH, TLBI_AllAttr);
**TLBI VMALLE1OSNXS{,<X>}%**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TTLBOS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && IsFeatureImplemented(FEAT_HCX) && (!IsHCRXEL2Enabled() || HCRX_EL2.FGTnXS == '0') && HFGITR_EL2.TLBIVMALLE1OS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability OSH, TLBI ExcludeXS);
    endif
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability OSH, TLBI ExcludeXS);
    else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability OSH, TLBI ExcludeXS);
    endif
elsif PSTATE.EL == EL3 then
    if HCR_EL2.<E2H,TGE> == '11' then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE, Shareability OSH, TLBI ExcludeXS);
    else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability OSH, TLBI ExcludeXS);
    endif
else
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability OSH, TLBI ExcludeXS);
end if
C5.5.80 TLBI VMALLS12E1, TLBI VMALLS12E1NXS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1

The TLBI VMALLS12E1 and TLBI VMALLS12E1NXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - If SCR_EL3.{NSE, NS} is {0, 0}, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {0, 1}, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {1, 1}, then:
    - The entry would be required to translate an address using the Realm EL1&0 translation regime.
    - The entry would be used with the current VMID.
- If FEAT_RME is not implemented, one of the following applies:
  - If SCR_EL3.NS is 0, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.NS is 1, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

The invalidation applies to the PE that executes this System instruction.

**Note**

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined. Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete. The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

**Configurations**

There are no configuration notes.
Attributes

TLBI VMALLS12E1, TLBI VMALLS12E1NXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLS12E1, TLBI VMALLS12E1NXS instruction

The Rt field should be set to 0b11111. If the Rt field is not set to 0b11111, it is CONstrained UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to 0b11111.

Accesses to this instruction use the following encodings in the System instruction encoding space:

**TLBI VMALLS12E1<, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() & HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AllAttr);
elsif PSTATE_EL == EL3 then
  if !EL2Enabled() then
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AllAttr);
  else
    AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AllAttr);

**TLBI VMALLS12E1NXS<, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0111</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() & HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_ExcludeXS);
elsif PSTATE_EL == EL3 then
  if !EL2Enabled() then
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH,
    TLBI_ExcludeXS);
  else
    AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH,
    TLBI_ExcludeXS);
C5.5.81 TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable

The TLBI VMALLS12E1IS and TLBI VMALLS12E1ISNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - If SCR_EL3.{NSE, NS} is {0, 0}, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {0, 1}, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {1, 1}, then:
    - The entry would be required to translate an address using the Realm EL1&0 translation regime.
    - The entry would be used with the current VMID.
- If FEAT_RME is not implemented, one of the following applies:
  - If SCR_EL3.NS is 0, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.NS is 1, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

---

**Note**

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.
Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined. Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete. The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configurations

There are no configuration notes.

Attributes

TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS instruction

The Rt field should be set to 0b11111. If the Rt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:

• The instruction is UNDEFINED.
• The instruction behaves as if the Rt field is set to 0b11111.

Accesses to this instruction use the following encodings in the System instruction encoding space:

<table>
<thead>
<tr>
<th>TLBI VMALLS12E1IS{, &lt;Xt&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
</tr>
<tr>
<td>0b01</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    endif
elsif PSTATE_EL == EL2 then
    AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_AllAttr);
elsif PSTATE_EL == EL3 then
    if !EL2Enabled() then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_AllAttr);
    else
        AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_AllAttr);
    endif
else
    AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_AllAttr);
The A64 System Instruction Class
C5.5 A64 System instructions for TLB maintenance

TLBI VMALLS12E1ISNXS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0011</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
  endif
elsif PSTATE_EL == EL2 then
  AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_ExcludeXS);
elsif PSTATE_EL == EL3 then
  if !EL2Enabled() then
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_ExcludeXS);
  else
    AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_ExcludeXS);
  endif
end
The TLBI VMALLS12E1OS and TLBI VMALLS12E1OSNXS characteristics are:

**Purpose**

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- If FEAT_RME is implemented, one of the following applies:
  - If SCR_EL3.{NSE, NS} is {0, 0}, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {0, 1}, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.
  - If SCR_EL3.{NSE, NS} is {1, 1}, then:
    - The entry would be required to translate an address using the Realm EL1&0 translation regime.
    - The entry would be used with the current VMID.
- If FEAT_RME is not implemented, one of the following applies:
  - If SCR_EL3.NS is 0, then:
    - The entry would be required to translate an address using the Secure EL1&0 translation regime.
    - If FEAT_SEL2 is implemented and enabled, the entry would be used with the current VMID.
  - If SCR_EL3.NS is 1, then:
    - The entry would be required to translate an address using the Non-secure EL1&0 translation regime.
    - If Non-secure EL2 is implemented, the entry would be used with the current VMID.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

**Note**

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, or would be defined to pass a VMID argument if SCR_EL3.EEL2==1, then:

- A PE with SCR_EL3.EEL2==1 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==0.
- A PE with SCR_EL3.EEL2==0 is not architecturally required to invalidate any entries in the Secure EL1&0 translation of a PE in the same required shareability domain with SCR_EL3.EEL2==1.
- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.
Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

If FEAT_XS is implemented, the nXS variant of this System instruction is defined. Both variants perform the same invalidation, but the TLBI System instruction without the nXS qualifier waits for all memory accesses using in-scope old translation information to complete before it is considered complete. The TLBI System instruction with the nXS qualifier is considered complete when the subset of these memory accesses with XS attribute set to 0 are complete.

Configurations

This instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI_VMALLS12E1OS, TLBI_VMALLS12E1OSNXS are UNDEFINED.

Attributes

TLBI_VMALLS12E1OS, TLBI_VMALLS12E1OSNXS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI_VMALLS12E1OS, TLBI_VMALLS12E1OSNXS instruction

The Rt field should be set to 0b11111. If the Rt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to 0b11111.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI_VMALLS12E1OS{, <Xt>}
```

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0001</td>
<td>0b110</td>
</tr>
</tbody>
</table>

```
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBI_AllAttr);
elsif PSTATE_EL == EL3 then
  if !EL2Enabled() then
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBI_AllAttr);
  else
    AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBI_AllAttr);
```

TLBI VMALLS12E1OSNXS{, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if !IsFeatureImplemented(FEAT_XS) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
  end
elsif PSTATE.EL == EL2 then
  AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBI_ExcludeXS);
elsif PSTATE.EL == EL3 then
  if !EL2Enabled() then
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBI_ExcludeXS);
  else
    AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_OSH, TLBI_ExcludeXS);
  end
else
  UNDEFINED;
end
C5.6 A64 System instructions for prediction restriction

This section lists the A64 System instructions for prediction restriction.
C5.6.1 CFP RCTX, Control Flow Prediction Restriction by Context

The CFP RCTX characteristics are:

**Purpose**

Control Flow Prediction Restriction by Context applies to all Control Flow Prediction Resources that predict execution based on information gathered within the target execution context or contexts. Control flow predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control speculative execution occurring after the instruction is complete and synchronized.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

--- **Note** ---

This instruction does not require the invalidation of prediction structures so long as the behavior described for completion of this instruction is met by the implementation.

On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

--- **Configurations** ---

This instruction is present only when FEAT_SPECRES is implemented. Otherwise, direct accesses to CFP RCTX are UNDEFINED.

--- **Attributes** ---

CFP RCTX is a 64-bit System instruction.

--- **Field descriptions** ---

Bits [63:49] reserved, RES0.

GVMID, bit [48]

Execution of this instruction applies to all VMIDs or a specified VMID.

- 0b0 Applies to specified VMID for an EL0 or EL1 target execution context.
- 0b1 Applies to all VMIDs for an EL0 or EL1 target execution context.

For target execution contexts other than EL0 or EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, this field has an Effective value of 0.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.
VMID, bits [47:32]
Only applies when bit[48] is 0 and the target execution context is either:
- EL1.
- EL0 when (HCR_EL2.E2H==0 or HCR_EL2.TGE==0).
Otherwise this field is RES0.
When the instruction is executed at EL1, this field is treated as the current VMID.
When the instruction is executed at EL0 and (HCR_EL2.E2H==0 or HCR_EL2.TGE==0), this field is treated as the current VMID.
When the instruction is executed at EL0 and (HCR_EL2.E2H==1 and HCR_EL2.TGE==1), this field is ignored.
If EL2 is not implemented or not enabled for the target Security state, this field is RES0.
If the implementation supports 16 bits of VMID, then the upper 8 bits of the VMID must be written to 0 by software when the context being affected only uses 8 bits.

Bits [31:28]
Reserved, RES0.

NSE, bit [27]
When FEAT_RME is implemented:
Together with the NS field, selects the Security state.
For a description of the values derived by evaluating NS and NSE together, see CFP_RCTX.NS.
Otherwise:
Reserved, RES0.

NS, bit [26]
When FEAT_RME is implemented:
Together with the NSE field, selects the Security state. Defined values are:

<table>
<thead>
<tr>
<th>NSE</th>
<th>NS</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b0</td>
<td>Secure.</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>Non-secure.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>Root.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>Realm.</td>
</tr>
</tbody>
</table>

Some Effective values are determined by the current Security state:
- When executed in Secure state, the Effective value of NSE is 0.
- When executed in Non-secure state, the Effective value of {NSE, NS} is {0, 1}.
- When executed in Realm state, the Effective value of {NSE, NS} is {1, 1}.
An instruction with an EL field that has a value other than 0b11 (EL3) is treated as a NOP when executed at EL3 with CFP_RCTX.{NSE, NS} == {1, 0}.

Otherwise:
Security State. Defined values are:
- 0b0 Secure state.
- 0b1 Non-secure state.
When executed in Non-secure state, the Effective value of NS is 1.
EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>EL0</td>
</tr>
<tr>
<td>0b01</td>
<td>EL1</td>
</tr>
<tr>
<td>0b10</td>
<td>EL2</td>
</tr>
<tr>
<td>0b11</td>
<td>EL3</td>
</tr>
</tbody>
</table>

If the instruction is executed at an Exception level lower than the specified level, this instruction is treated as a NOP.

Bits [23:17]

Reserved, RES0.

GASID, bit [16]

Execution of this instruction applies to all ASIDs or a specified ASID.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Applies to specified ASID for an EL0 target execution context.</td>
</tr>
<tr>
<td>0b1</td>
<td>Applies to all ASIDs for an EL0 target execution context.</td>
</tr>
</tbody>
</table>

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field has an Effective value of 0.

ASID, bits [15:0]

Only applies for an EL0 target execution context and when bit[16] is 0.

Otherwise, this field is RES0.

When the instruction is executed at EL0, this field is treated as the current ASID.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being affected only uses 8 bits.

**Executing CFP RCTX instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

**CFP RCTX, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b011</td>
<td>0b0111</td>
<td>0b0011</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !(EL2Enabled() && HCR_EL2,<E2H,TGE> == '11') && SCTLR_EL1.EnRCTX == '0' then
    if EL2Enabled() && HCR_EL2,TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    endif
  else
    AArch64.RestricPrediction(X[t, 64], RestrictType_ControlFlow);
  endif
else
  if EL2Enabled() && HCR_EL2,NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.CFPRCTX == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.RestricPrediction(X[t, 64], RestrictType_ControlFlow);
  endif
endif
AArch64.RestrictPrediction(X[t, 64], RestrictType_ControlFlow);
elsif PSTATE_EL == EL2 then
  AArch64.RestrictPrediction(X[t, 64], RestrictType_ControlFlow);
elsif PSTATE_EL == EL3 then
  AArch64.RestrictPrediction(X[t, 64], RestrictType_ControlFlow);
C5.6.2 CPP RCTX, Cache Prefetch Prediction Restriction by Context

The CPP RCTX characteristics are:

**Purpose**

Cache Prefetch Prediction Restriction by Context applies to all Cache Allocation Resources that predict cache allocations based on information gathered within the target execution context or contexts.

The actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitively control cache prefetch predictions occurring after the instruction is complete and synchronized.

This instruction applies to all:
- Instruction caches.
- Data caches.
- TLB prefetching hardware used by the executing PE that applies to the supplied context or contexts.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

**Note**

This instruction does not require the invalidation of Cache Allocation Resources so long as the behavior described for completion of this instruction is met by the implementation.

On some implementations the instruction is likely to take a significant number of cycles to execute.

This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

**Configurations**

This instruction is present only when FEAT_SPECRES is implemented. Otherwise, direct accesses to CPP RCTX are UNDEFINED.

**Attributes**

CPP RCTX is a 64-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:49</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>48:32</td>
<td>VMID</td>
</tr>
<tr>
<td>31:23</td>
<td>NS EL RES0</td>
</tr>
<tr>
<td>22:0</td>
<td>ASID</td>
</tr>
</tbody>
</table>

**Bits [63:49]**

Reserved, RES0.

**GVMID, bit [48]**

Execution of this instruction applies to all VMIDs or a specified VMID.

- **0b0**: Applies to specified VMID for an EL0 or EL1 target execution context.
- **0b1**: Applies to all VMIDs for an EL0 or EL1 target execution context.
For target execution contexts other than EL0 and EL1, this field is RES0.
If the instruction is executed at EL0 or EL1, this field has an Effective value of 0.
If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

VMID, bits [47:32]
Only applies when bit[48] is 0 and the target execution context is either:
• EL1.
• EL0 when (HCR_EL2.E2H==0 or HCR_EL2.TGE==0).
Otherwise this field is RES0.
When the instruction is executed at EL1, this field is treated as the current VMID.
When the instruction is executed at EL0 and (HCR_EL2.E2H==0 or HCR_EL2.TGE==0), this field is treated as the current VMID.
When the instruction is executed at EL0 and (HCR_EL2.E2H==1 and HCR_EL2.TGE==1), this field is ignored.
If EL2 is not implemented or not enabled for the target Security state, this field is RES0.
If the implementation supports 16 bits of VMID, then the upper 8 bits of the VMID must be written to 0 by software when the context being affected only uses 8 bits.

Bits [31:28]
Reserved, RES0.

NSE, bit [27]
When FEAT_RME is implemented:
Together with the NS field, selects the Security state.
For a description of the values derived by evaluating NS and NSE together, see CPP_RCTX.NS.
Otherwise:
Reserved, RES0.

NS, bit [26]
When FEAT_RME is implemented:
Together with the NSE field, selects the Security state. Defined values are:

<table>
<thead>
<tr>
<th>NSE</th>
<th>NS</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b0</td>
<td>Secure.</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>Non-secure.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>Root.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>Realm.</td>
</tr>
</tbody>
</table>

Some Effective values are determined by the current Security state:
• When executed in Secure state, the Effective value of NSE is 0.
• When executed in Non-secure state, the Effective value of {NSE, NS} is \{0, 1\}.
• When executed in Realm state, the Effective value of {NSE, NS} is \{1, 1\}.
An instruction with an EL field that has a value other than 0b11 (EL3) is treated as a NOP when executed at EL3 with CPP_RCTX.{NSE, NS} == \{1, 0\}.

Otherwise:
Security State. Defined values are:
0b0  Secure state.
0b1  Non-secure state.
When executed in Non-secure state, the Effective value of NS is 1.

**EL, bits [25:24]**
Exception Level. Indicates the Exception level of the target execution context.

0b00  EL0.
0b01  EL1.
0b10  EL2.
0b11  EL3.

If the instruction is executed at an Exception level lower than the specified level, this instruction is treated as a NOP.

**Bits [23:17]**
Reserved, RES0.

**GASID, bit [16]**
Execution of this instruction applies to all ASIDs or a specified ASID.

0b0  Applies to specified ASID for an EL0 target execution context.
0b1  Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.
If the instruction is executed at EL0, this field has an Effective value of 0.

**ASID, bits [15:0]**
Only applies for an EL0 target execution context and when bit[16] is 0. Otherwise, this field is RES0.
When the instruction is executed at EL0, this field is treated as the current ASID.
If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being affected only uses 8 bits.

**Executing CPP RCTX instruction**
Accesses to this instruction use the following encodings in the System instruction encoding space:

**CPP RCTX, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b011</td>
<td>0b0111</td>
<td>0b0011</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.EnRCTX == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
      HFGITR_EL2.CPPRCTX == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.RestrictPrediction(X[t, 64], RestrictType_CachePrefetch);
  else
    if PSTATE.EL == EL1 then
      if EL2Enabled() && HCR_EL2.NV == '1' then

AArch64.SystemAccessTrap(EL2, 0x18);
elseif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCITR_EL2.CPPRCTX == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  AArch64.RestrictPrediction(X[t, 64], RestrictType_CachePrefetch);
elseif PSTATE.EL == EL2 then
  AArch64.RestrictPrediction(X[t, 64], RestrictType_CachePrefetch);
elseif PSTATE.EL == EL3 then
  AArch64.RestrictPrediction(X[t, 64], RestrictType_CachePrefetch);
C5.6.3 DVP RCTX, Data Value Prediction Restriction by Context

The DVP RCTX characteristics are:

**Purpose**

Data Value Prediction Restriction by Context applies to all Data Value Prediction Resources that predict execution based on information gathered within the target execution context or contexts. Data value predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitively control speculative execution occurring after the instruction is complete and synchronized.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

--- **Note** ---

This instruction does not require the invalidation of prediction structures so long as the behavior described for completion of this instruction is met by the implementation.

On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

---

**Configurations**

This instruction is present only when FEAT_SPECRES is implemented. Otherwise, direct accesses to DVP RCTX are UNDEFINED.

**Attributes**

DVP RCTX is a 64-bit System instruction.

**Field descriptions**

```
       63    49  48  47  32
       RES0                            VMID

       31    28  27  26  25  24  23  17  16  15  0
       RES0  NS  EL  RES0              ASID
```

**Bits [63:49]**

Reserved, RES0.

**GVMID, bit [48]**

Execution of this instruction applies to all VMIDs or a specified VMID.

- 0b0 Applies to specified VMID for an EL0 or EL1 target execution context.
- 0b1 Applies to all VMIDs for an EL0 or EL1 target execution context.

For target execution contexts other than EL0 or EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, then this field has an Effective value of 0.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.
VMID, bits [47:32]

Only applies when bit[48] is 0 and the target execution context is either:

- EL1.
- EL0 when (HCR_EL2.E2H==0 or HCR_EL2.TGE==0).

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0 and (HCR_EL2.E2H==0 or HCR_EL2.TGE==0), this field is treated as the current VMID.

When the instruction is executed at EL0 and (HCR_EL2.E2H==1 and HCR_EL2.TGE==1), this field is ignored.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

If the implementation supports 16 bits of VMID, then the upper 8 bits of the VMID must be written to 0 by software when the context being affected only uses 8 bits.

Bits [31:28]

Reserved, RES0.

NSE, bit [27]

When FEAT_RME is implemented:

Together with the NS field, selects the Security state.

For a description of the values derived by evaluating NS and NSE together, see DVP_RCTX.NS.

Otherwise:

Reserved, RES0.

NS, bit [26]

When FEAT_RME is implemented:

Together with the NSE field, selects the Security state. Defined values are:

<table>
<thead>
<tr>
<th>NSE</th>
<th>NS</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b0</td>
<td>Secure.</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>Non-secure.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>Root.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>Realm.</td>
</tr>
</tbody>
</table>

Some Effective values are determined by the current Security state:

- When executed in Secure state, the Effective value of NSE is 0.
- When executed in Non-secure state, the Effective value of {NSE, NS} is {0, 1}.
- When executed in Realm state, the Effective value of {NSE, NS} is {1, 1}.

An instruction with an EL field that has a value other than 0b11 (EL3) is treated as a NOP when executed at EL3 with DVP_RCTX.{NSE, NS} == {1, 0}.

Otherwise:

Security State. Defined values are:

- 0b0 Secure state.
- 0b1 Non-secure state.

When executed in Non-secure state, the Effective value of NS is 1.
EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

- 0b00: EL0.
- 0b01: EL1.
- 0b10: EL2.
- 0b11: EL3.

If the instruction is executed at an Exception level lower than the specified level, this instruction is treated as a NOP.

Bits [23:17]

Reserved, RES0.

GASID, bit [16]

Execution of this instruction applies to all ASIDs or a specified ASID.

- 0b0: Applies to specified ASID for an EL0 target execution context.
- 0b1: Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field has an Effective value of 0.

ASID, bits [15:0]

Only applies for an EL0 target execution context and when bit[16] is 0. Otherwise this field is RES0.

When the instruction is executed at EL0, this field is treated as the current ASID.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being affected only uses 8 bits.

Executing DVP RCTX instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

**DVP RCTX, <X>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b011</td>
<td>0b0111</td>
<td>0b0011</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.EnRCTX == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && !HaveEL(EL3) || SCR_EL3.FGTEn == '1' && HFGITR_EL2.DVPRCTX == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if PSTATE.EL == EL1 then
      if EL2Enabled() && HCR_EL2.ENV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      else
        if EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DVPRCTX == '1' then
          AArch64.RestRICTPrediction(X[t, 64], RestrictType_DataValue);
        else
          AArch64.SystemAccessTrap(EL2, 0x18);
        end
      end
    else
      if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      else
        if EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.DVPRCTX == '1' then
          AArch64.SystemAccessTrap(EL2, 0x18);
        else
          AArch64.SystemAccessTrap(EL2, 0x18);
        end
      end
    else
      AArch64.SystemAccessTrap(EL2, 0x18);
  end
end
AArch64.RestrictPrediction(X[t, 64], RestrictType_DataValue);
elsif PSTATE.EL == EL2 then
  AArch64.RestrictPrediction(X[t, 64], RestrictType_DataValue);
elsif PSTATE.EL == EL3 then
  AArch64.RestrictPrediction(X[t, 64], RestrictType_DataValue);
C5.7 A64 System instructions for the Branch Record Buffer Extension

This section lists the A64 System instructions for the Branch Record Buffer Extension.
C5.7.1 BRB IALL, Invalidate the Branch Record Buffer

The BRB IALL characteristics are:

**Purpose**

Invalidates all Branch records in the Branch Record Buffer.

**Configurations**

This instruction is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRB IALL are UNDEFINED.

**Attributes**

BRB IALL is a 64-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

**Executing BRB IALL instruction**

Rt should be encoded as 0b11111. If the Rt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to 0b11111.

Accesses to this instruction use the following encodings in the System instruction encoding space:

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b001</td>
<td>0b1111</td>
<td>0b0010</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED EL3 trap priority when SDD == '1' & MDCR.EL3.SBRBE != '1' & SCR.EL3.NS == '0' then
        UNDEFINED;
    elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED EL3 trap priority when SDD == '1' & MDCR.EL3.SBRBE == 'x0' & SCR.EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() & (!HaveEL(EL3) || SCR.EL3.FGTEn == '1') & HFGITR_EL2.nRBRIALL == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) & MDCR.EL3.SBRBE == '11' & SCR.EL3.NS == '0' then
        if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) & MDCR.EL3.SBRBE == 'x0' & SCR.EL3.NS == '1' then
        if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        BRB_IALL();
elsif PSTATE_EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED EL3 trap priority when SDD == '1' & MDCR.EL3.SBRBE != '11' & SCR.EL3.NS == '0' then

UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    UNDEFINED;
elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    endif
elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    endif
else
    BRB_IALL();
elsif PSTATE.EL == EL3 then
    BRB_IALL();
C5.7.2 BRB INJ, Branch Record Injection into the Branch Record Buffer

The BRB INJ characteristics are:

**Purpose**

Injects the Branch Record held in BRBINJ_EL1, BRBSRCINJ_EL1, and BRBTGTINJ_EL1 into the Branch Record Buffer.

**Configurations**

This instruction is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRB INJ are UNDEFINED.

**Attributes**

BRB INJ is a 64-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

**Executing BRB INJ instruction**

Rt should be encoded as 0b11111. If the Rt field is not set to 0b11111, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to 0b11111.

Accesses to this instruction use the following encodings in the System instruction encoding space:

### BRB INJ

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>0b001</td>
<td>0b0111</td>
<td>0b0010</td>
<td>0b101</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & MDACR_EL3.SBRBE != '1' & SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & MDACR_EL3.SBRBE == 'x0' & SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif !Enabled() & (HaveEL(EL3) || SCR_EL3.TGTE == '1') & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & MDACR_EL3.SBRBE == 'x0' & SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    endif
else
    if !Enabled() & haveEL(EL3) & MDACR_EL3.SBRBE == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    endif
else
    BRB_INJ();
elsif PSTATE_EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & MDACR_EL3.SBRBE == 'x0' & SCR_EL3.NS == '0' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    endif
else
    AArch64.SystemAccessTrap(EL3, 0x18);
endif
```
when SDD == '1' && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    UNDEFINED;
elsif Halted() && HaveEL3() && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap
priority when SODD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    UNDEFINED;
elsif HaveEL3() && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    endif
elsif HaveEL3() && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    endif
else
    BRB_INJ();
elsif PSTATE.EL == EL3 then
    BRB_INJ();
Chapter C6
A64 Base Instruction Descriptions

This chapter describes the A64 base instructions.

It contains the following sections:

- *About the A64 base instructions on page C6-1166.*
- *Alphabetical list of A64 base instructions on page C6-1169.*
C6.1 About the A64 base instructions

Alphabetical list of A64 base instructions on page C6-1169 gives full descriptions of the A64 instructions that are in the following instruction groups:

- Branch, Exception generation, and System instructions.
- Loads and stores associated with the general-purpose registers.
- Data processing (immediate).
- Data processing (register).

A64 instruction set encoding on page C4-388 provides an overview of the instruction encodings as well as of the instruction classes within their functional groups.

The rest of this section is general description of the base instructions. It contains the following subsections:

- Register size.
- Use of the PC.
- Use of the stack pointer on page C6-1167.
- Condition flags and related instructions on page C6-1167.

C6.1.1 Register size

Most data processing, comparison, and conversion instructions that use the general-purpose registers as the source or destination operand have two instruction variants that operate on either a 32-bit or a 64-bit value.

Where a 32-bit instruction form is selected, the following holds:

- The upper 32 bits of the source registers are ignored.
- The upper 32 bits of the destination register are set to zero.
- Right shifts and right rotates inject at bit[31], not at bit[63].
- The Condition flags, where set by the instruction, are computed from the lower 32 bits.

This distinction applies even when the results of a 32-bit instruction form are indistinguishable from the lower 32 bits computed by the equivalent 64-bit instruction form. For example, a 32-bit bitwise ORR could be performed using a 64-bit ORR and simply ignoring the top 32 bits of the result. However, the A64 instruction set includes separate 32-bit and 64-bit forms of the ORR instruction.

As well as distinct sign-extend or zero-extend instructions, the A64 instruction set also provides the ability to extend and shift the final source register of an ADD, SUB, ADDS, or SUBS instruction and the index register of a load/store instruction. This enables array index calculations involving a 64-bit array pointer and a 32-bit array index to be implemented efficiently.

The assembly language notation enables the distinct identification of registers holding 32-bit values and registers holding 64-bit values. See Register names on page C1-230 and Register indexed addressing on page C1-234.

C6.1.2 Use of the PC

A64 instructions have limited access to the PC. The only instructions that can read the PC are those that generate a PC relative address:

- ADR and ADRP.
- The Load register (literal) instruction class.
- Direct branches that use an immediate offset.
- The unconditional branch with link instructions, BL and BLR, that use the PC to create the return link address.

Only explicit control flow instructions can modify the PC:

- Conditional and unconditional branch and return instructions.
- Exception generation and exception return instructions.

For more details of instructions that can modify the PC, see Branches, Exception generating, and System instructions on page C3-248.
C6.1.3  **Use of the stack pointer**

A64 instructions can use the stack pointer only in a limited number of cases:

- Load/store instructions use the current stack pointer as the base address:
  - When stack alignment checking is enabled by system software and the base register is SP, the current stack pointer must be initially quadword aligned. That is, it must be aligned to 16 bytes. Misalignment generates an SP alignment fault. See [SP alignment checking](#) on page D1-4668 for more information.

- Add and subtract data processing instructions in their immediate and extended register forms, use the current stack pointer as a source register or the destination register or both.

- Logical data processing instructions in their immediate form use the current stack pointer as the destination register.

C6.1.4  **Condition flags and related instructions**

The A64 base instructions that use the Condition flags as an input are:

- Conditional branch. The conditional branch instruction is `B.cond`.

- Add or subtract with carry. These instruction types include instructions to perform multi-precision arithmetic and calculate checksums. The add or subtract with carry instructions are `ADC`, `ADCS`, `SBC`, and `SBCS`, or an architectural alias for these instructions.

- Conditional select with increment, negate, or invert. This instruction type conditionally selects between one source register and a second, incremented, negated, inverted, or unmodified source register. The conditional select with increment, negate, or invert instructions are `CSINC`, `CSINV`, and `CSNEG`.

  These instructions also implement:
  - Conditional select or move. The Condition flags select one of two source registers as the destination register. Short conditional sequences can be replaced by unconditional instructions followed by a conditional select, `CSEL`.
  - Conditional set. Conditionally selects between 0 and 1, or 0 and -1. This can be used to convert the Condition flags to a Boolean value or mask in a general-purpose register, for example. These instructions include `CSET` and `CSETM`.

- Conditional compare. This instruction type sets the Condition flags to the result of a comparison if the original condition is true, otherwise it sets the Condition flags to an immediate value. It permits the flattening of nested conditional expressions without using conditional branches or performing Boolean arithmetic within the general-purpose registers. The conditional compare instructions are `CCMP` and `CMN`.

The A64 base instructions that update the Condition flags as an output are:

- Flag-setting data processing instructions, such as `ADCS`, `ADDS`, `ANDS`, `BICS`, `RMIF`, `SBCS`, `SETF8`, `SETF16`, and `SUBS`, and the aliases `CMN`, `CMP`, and `TST`.

- Conditional compare instructions such as `CMN`, `COMP`.

- The random number generation instructions MRS RNDR and MRS RNDRRS, see [Effect of random number generation instructions on Condition flags](#) on page C6-1168.

The A64 base instructions that manipulate the Condition flags are:

- The flag manipulation instruction `CFINV`, which inverts the value of the Carry flag.
• If `FEAT_FlagM2` is implemented, the base instructions `AXFLAG` and `XAFLAG`. These instructions convert between the Arm floating point comparison PSTATE condition flag format and an alternative format shown in Table C6-1.

Table C6-1 Relationship between ARM format and alternative format PSTATE condition flags

<table>
<thead>
<tr>
<th>Result</th>
<th>ARM format</th>
<th>Alternative format</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N Z C V</td>
<td>N Z C V</td>
</tr>
<tr>
<td>Greater than</td>
<td>0 0 1 0</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>Less than</td>
<td>1 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>Equal</td>
<td>0 1 1 0</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>Unordered</td>
<td>0 0 1 1</td>
<td>0 1 0 0</td>
</tr>
</tbody>
</table>

The flags can be directly accessed for a read/write using the `NZCV, Condition Flags` on page C5-706.

The A64 base instructions also include conditional branch instructions that do not use the Condition flags as an input:

• Compare and branch if a register is zero or nonzero, `CBZ` and `CBNZ`.
• Test a single bit in a register and branch if the bit is zero or nonzero, `TBZ` and `TBNZ`.

Effect of random number generation instructions on Condition flags

If `FEAT_RNG` is implemented, then:

• When a valid random number is returned, the `PSTATE.NZCV` flags are set to 0b0000.
• If the random number hardware is not capable of returning a random number in a reasonable period of time, the `PSTATE.NZCV` flags are set to 0b0100, and the random number generation instructions return the value 0.

_____ Note _____

The definition of “reasonable period of time” is IMPLEMENTATION DEFINED. The expectation is that software might use this as an opportunity to reschedule or run a different routine, perhaps after a small number of retries have failed to return a valid value.
C6.2 Alphabetical list of A64 base instructions

This section lists every instruction in the base category of the A64 instruction set. For details of the format used, see Understanding the A64 instruction descriptions on page C2-240.
C6.2.1 ADC

Add with Carry adds two register values and the Carry flag value, and writes the result to the destination register.

32-bit variant
Applies when \( sf = 0 \).
ADC \(<Wd>, <Wn>, <Wm>\)

64-bit variant
Applies when \( sf = 1 \).
ADC \(<Xd>, <Xn>, <Xm>\)

Decode for all variants of this encoding
\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{integer } \text{datasize} &= \text{if } sf = '1' \text{ then } 64 \text{ else } 32;
\end{align*}
\]

Assembler symbols
\(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
\(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
\(<Wm>\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
\(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
\(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
\(<Xm>\) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation
\[
\begin{align*}
\text{bits(datasize)} \text{ result;} \\
\text{bits(datasize)} \text{ operand1} &= X[n, \text{datasize}]; \\
\text{bits(datasize)} \text{ operand2} &= X[m, \text{datasize}]; \\
(\text{result, -}) &= \text{AddWithCarry}(\text{operand1, operand2, PSTATE.C}); \\
X[d, \text{datasize}] &= \text{result};
\end{align*}
\]

Operational information
If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.2 ADCS

Add with Carry, setting flags, adds two register values and the Carry flag value, and writes the result to the destination register. It updates the condition flags based on the result.

For 32-bit variant:
Applies when \( sf = 0 \).

\[
ADCS \ <Wd>, \ <Wn>, \ <Wm> 
\]

For 64-bit variant:
Applies when \( sf = 1 \).

\[
ADCS \ <Xd>, \ <Xn>, \ <Xm> 
\]

**Decode for all variants of this encoding**

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{integer } \text{datasize} &= \text{if } sf = '1' \text{ then } 64 \text{ else } 32;
\end{align*}
\]

**Assembler symbols**

- \(<Wd>\): Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\): Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Wm>\): Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<Xd>\): Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\): Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Xm>\): Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

**Operation**

\[
\begin{align*}
\text{bits(datasize) result}; \\
\text{bits(datasize) operand1} &= X[n, \text{datasize}]; \\
\text{bits(datasize) operand2} &= X[m, \text{datasize}]; \\
\text{bits(4) nzcv}; \\
(\text{result, } nzcv) &= \text{AddWithCarry(operand1, operand2, PSTATE.C)}; \\
\text{PSTATE.<N,Z,C,V> } &= nzcv; \\
X[d, \text{datasize}] &= \text{result};
\end{align*}
\]
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.3 ADD (extended register)

Add (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the \(<Rm>\) register can be a byte, halfword, word, or doubleword.

32-bit variant

Applies when \(sf == 0\).

\[
\text{ADD} \; <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
\]

64-bit variant

Applies when \(sf == 1\).

\[
\text{ADD} \; <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{integer } \text{datasize} &= \text{if } sf == '1' \text{ then } 64 \text{ else } 32; \\
\text{ExtendType } \text{extend_type} &= \text{DecodeRegExtend}(\text{option}); \\
\text{integer } \text{shift} &= \text{UInt}(\text{imm3}); \\
\text{if } \text{shift} > 4 \text{ then UNDEFINED; }
\end{align*}
\]

Assembler symbols

\(\text{<Wd|WSP>}\) Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

\(\text{<Wn|WSP>}\) Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

\(\text{<Wm>}\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

\(\text{<Xd|SP>}\) Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

\(\text{<Xn|SP>}\) Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

\(\text={<R>}\) Is a width specifier, encoded in the "option" field. It can have the following values:

\[
\begin{align*}
\text{W} & \quad \text{when option} = 00x \\
\text{W} & \quad \text{when option} = 010 \\
\text{X} & \quad \text{when option} = 111 \\
\text{W} & \quad \text{when option} = 10x \\
\text{W} & \quad \text{when option} = 110
\end{align*}
\]

\(\text{<m>}\) Is the number \([0-30]\) of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the
"option" field. It can have the following values:

- UXTB when option = 000
- UXTH when option = 001
- LSL|UXTW when option = 010
- UTX when option = 011
- SXTB when option = 100
- SXTH when option = 101
- SXTW when option = 110
- SXTX when option = 111

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted
when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is
'010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the
"option" field. It can have the following values:

- UXTB when option = 000
- UXTH when option = 001
- UTX when option = 010
- LSL|UXTX when option = 011
- SXTB when option = 100
- SXTH when option = 101
- SXTW when option = 110
- SXTX when option = 111

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when
"imm3" is '000'. In all other cases <extend> is required and must be UTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in
the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL,
and is optional when <extend> is present but not LSL.

Operation

```plaintext
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[<datasize-1:0>] else X[n, datasize];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);
(result, -) = AddWithCarry(operand1, operand2, '0');
if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 0:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.4 ADD (immediate)

Add (immediate) adds a register value and an optionally-shifted immediate value, and writes the result to the destination register.

This instruction is used by the alias MOV (to/from SP). See Alias conditions for details of when each alias is preferred.

32-bit variant
Applies when \( sf == 0 \).

\[
\text{ADD} \ <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>} 
\]

64-bit variant
Applies when \( sf == 1 \).

\[
\text{ADD} \ <Xd|SP>, <Xn|SP>, #<imm>{, <shift>} 
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } \text{datasize} &= \text{if } sf == '1' \text{ then } 64 \text{ else } 32; \\
\text{bits} (\text{datasize}) \text{ imm};
\end{align*}
\]

\[
\begin{align*}
\text{case } \text{sh} \text{ of}
\end{align*}
\]

\[
\begin{align*}
\text{when } '0' \text{ imm} &= \text{ZeroExtend}(\text{imm12}, \text{datasize}); \\
\text{when } '1' \text{ imm} &= \text{ZeroExtend}(\text{imm12}:\text{Zeros}(12), \text{datasize});
\end{align*}
\]

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (to/from SP)</td>
<td>( sh == '0' &amp;&amp; imm12 == '000000000000' &amp;&amp; (Rd == '11111' &amp;&amp; Rn == '11111') )</td>
</tr>
</tbody>
</table>

Assembler symbols

| <Wd|WSP> | Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field. |
|------|---------------------------------------------------------------|
| <Wn|WSP> | Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.         |
| <Xd|SP>  | Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.     |
| <Xn|SP>  | Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.         |
| <imm> | Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field. |
<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the “sh” field. It can have the following values:

- LSL #0 when sh = 0
- LSL #12 when sh = 1

**Operation**

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[<datasize-1:0> else X[n, datasize];

(result, -) = AddWithCarry(operand1, imm, '0');

if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.5   ADD (shifted register)

Add (shifted register) adds a register value and an optionally-shifted register value, and writes the result to the destination register.

### 32-bit variant

Applies when sf == 0.

ADD <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

### 64-bit variant

Applies when sf == 1.

ADD <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

### Assembler symbols

- `<Wd>`  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`  Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>`  Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>`  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`  Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>`  Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>`  Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:
  - LSL when shift = 00
  - LSR when shift = 01
  - ASR when shift = 10
  - The encoding shift = 11 is reserved.
- `<amount>`  For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Operation**

```plaintext
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);

(result, -) = AddWithCarry(operand1, operand2, '0');
X[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
### C6.2.6 ADDG

Add with Tag adds an immediate value scaled by the Tag granule to the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

#### Integer

(FEAT_MTE)

```plaintext
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>16</th>
<th>15 14 13</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0</td>
<td>1 0 0 1 1 0</td>
<td>uimm6</td>
<td>(0)</td>
<td>uimm4</td>
<td>Xn</td>
<td>Xd</td>
<td></td>
</tr>
</tbody>
</table>
```

#### Encoding

ADDG <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>

#### Decode for this encoding

- if !HaveMTEExt() then UNDEFINED;
- integer d = UInt(Xd);
- integer n = UInt(Xn);
- bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);

#### Assembler symbols

- `<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
- `<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Xn" field.
- `<uimm6>` Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the "uimm6" field.
- `<uimm4>` Is an unsigned immediate, in the range 0 to 15, encoded in the "uimm4" field.

#### Operation

```plaintext
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;
if AArch64.AllocationTagAccessTypeIsEnabled(AccType_NORMAL) then
    rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
else
    rtag = '0000';
(result, -) = AddWithCarry(operand1, offset, '0');
result = AArch64.AddressWithAllocationTag(result, AccType_NORMAL, rtag);
if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```
C6.2.7 ADDS (extended register)

Add (extended register), setting flags, adds a register value and a sign or zero-extended register value, followed by
an optional left shift amount, and writes the result to the destination register. The argument that is extended from
the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias CMN (extended register). See Alias conditions for details of when each alias is
preferred.

### 32-bit variant

Applies when \( sf == 0 \).

\[
\text{ADD} <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
\]

### 64-bit variant

Applies when \( sf == 1 \).

\[
\text{ADD} <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
\]

#### Decode for all variants of this encoding

```
integer d = UInt(Rd);
ingen n = UInt(Rn);
ingen m = UInt(Rm);
datatype datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

#### Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN (extended register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

#### Assembler symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn|WSP>` Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<R>` Is a width specifier, encoded in the "option" field. It can have the following values:
  - \( w \) when option = 00x
W when option = 010
X when option = x11
W when option = 10x
W when option = 110

Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000
UXTH when option = 001
LSL|UXTW when option = 010
UXTX when option = 011
SXTB when option = 100
SXTH when option = 101
SXTW when option = 110
SXTX when option = 111

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000
UXTH when option = 001
UXTW when option = 010
LSL|UXTX when option = 011
SXTB when option = 100
SXTH when option = 101
SXTW when option = 110
SXTX when option = 111

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

**Operation**

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[<datasize-1:0>] else X[n, datasize];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, '0');

PSTATE.<N,Z,C,V> = nzcv;
X[d, datasize] = result;
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
**C6.2.8 ADDS (immediate)**

Add (immediate), setting flags, adds a register value and an optionally-shifted immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias **CMN (immediate)**. See **Alias conditions** for details of when each alias is preferred.

---

**32-bit variant**

Applies when $sf == 0$.

```
ADDS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

**64-bit variant**

Applies when $sf == 1$.

```
ADDS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

**Decode for all variants of this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case sh of
  when '0' imm = ZeroExtend(imm12, datasize);
  when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

**Alias conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN (immediate)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

**Assembler symbols**

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn|WSP>`: Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn|SP>`: Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- `<shift>`: Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:
  - LSL #0 when $sh = 0$
  - LSL #12 when $sh = 1$
Operation

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, imm, '0');
PSTATE.<N,Z,C,V> = nzcv;
X[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.9 ADDS (shifted register)

Add (shifted register), setting flags, adds a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMN (shifted register). See Alias conditions for details of when each alias is preferred.

32-bit variant
Applies when \( sf == 0 \).

\[
\text{ADDS} \ <Wd>, \ <Wn>, \ <Wm>\{, \ <\text{shift}> \ #<\text{amount}>\}
\]

64-bit variant
Applies when \( sf == 1 \).

\[
\text{ADDS} \ <Xd>, \ <Xn>, \ <Xm>\{, \ <\text{shift}> \ #<\text{amount}>\}
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{integer } \text{datasize} &= \text{if } sf == '1' \text{ then } 64 \text{ else } 32; \\
\text{if } \text{shift} == '11' \text{ then UNDEFINED; } \\
\text{if } sf == '0' \&\& \text{imm6<5> == '1'} \text{ then UNDEFINED; }
\end{align*}
\]

\[
\text{ShiftType shift_type} = \text{DecodeShift(shift);} \\
\text{integer shift_amount} = \text{UInt}(\text{imm6});
\]

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN (shifted register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Wm>\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Xm>\) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- LSL when shift = 00
- LSR when shift = 01
- ASR when shift = 10

The encoding shift = 11 is reserved.

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

bits(datasize) result;
bins(datasize) operand1 = X[n, datasize];
bins(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bins(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, '0');
PSTATE.<N,Z,C,V> = nzcv;
X[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.10   ADR

Form PC-relative address adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.

![Instruction Format](image)

**Encoding**

ADR <Xd>, <label>

**Decode for this encoding**

integer d = UInt(Rd);
bits(64) imm;

imm = SignExtend(immhi:immlo, 64);

**Assembler symbols**

Xd: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

label: Is the program label whose address is to be calculated. Its offset from the address of this instruction, in the range +/-1MB, is encoded in "immhi:immlo".

**Operation**

bits(64) base = PC[];

X[d, 64] = base + imm;
C6.2.11 ADRP

Form PC-relative address to 4KB page adds an immediate value that is shifted left by 12 bits, to the PC value to form a PC-relative address, with the bottom 12 bits masked out, and writes the result to the destination register.

```
[31 30 29 28|27 26 25 24|23 |    |    |    | 5 4 | 0 ]
   1 | immlo 1 0 0 0 0     immhi     Rd
op
```

**Encoding**

ADRP <Xd>, <label>

**Decode for this encoding**

integer d = UInt(Rd);
bits(64) imm;

\[
imm = \text{SignExtend}(immhi:immlo:Zeros(12), 64);
\]

**Assembler symbols**

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<label> Is the program label whose 4KB page address is to be calculated. Its offset from the page address of this instruction, in the range +/-4GB, is encoded as "immhi:immlo" times 4096.

**Operation**

bits(64) base = PC[];
base<11:0> = Zeros(12);
X[d, 64] = base + imm;
C6.2.12 **AND (immediate)**

Bitwise AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

```
[31 30 29 28|27 26 25 24|23 22 21 | 16|15 | 10 9 | 5 4 | 0 ]
```

<table>
<thead>
<tr>
<th>sf</th>
<th>0 0</th>
<th>1 0 0 1 0 0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit variant**

Applies when sf == 0 && N == 0.

AND <Wd|WSP>, <Wn>, #<imm>

**64-bit variant**

Applies when sf == 1.

AND <Xd|SP>, <Xn>, #<imm>

**Decode for all variants of this encoding**

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
b bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

**Assembler symbols**

- `<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<imm>` For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
  For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

**Operation**

```java
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
result = operand1 AND imm;
if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.13   **AND (shifted register)**

Bitwise AND (shifted register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.

### 32-bit variant

Applies when `sf == 0`.

\[
\text{AND } \langle Wd \rangle, \langle Wn \rangle, \langle Wm \rangle\{, \langle shift \rangle \#<\text{amount}>\}
\]

### 64-bit variant

Applies when `sf == 1`.

\[
\text{AND } \langleXd \rangle, \langleXn \rangle, \langleXm \rangle\{, \langle shift \rangle \#<\text{amount}>\}
\]

**Decode for all variants of this encoding**

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{integer } \text{datasize} &= \text{if } sf == '1' \text{ then } 64 \text{ else } 32; \\
&\text{if } sf == '0' \&\& \text{imm6} < 5 > == '1' \text{ then UNDEFINED;} \\
\text{ShiftType } \text{shift_type} &= \text{DecodeShift}(\text{shift}); \\
\text{integer } \text{shift_amount} &= \text{UInt}(\text{imm6});
\end{align*}
\]

**Assembler symbols**

\[
\begin{align*}
\langle Wd \rangle & \quad \text{Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.} \\
\langle Wn \rangle & \quad \text{Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.} \\
\langle Wm \rangle & \quad \text{Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.} \\
\langleXd \rangle & \quad \text{Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.} \\
\langleXn \rangle & \quad \text{Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.} \\
\langleXm \rangle & \quad \text{Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.} \\
\langle shift \rangle & \quad \text{Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:} \\
\text{LSL} & \quad \text{when } \text{shift} = 00 \\
\text{LSR} & \quad \text{when } \text{shift} = 01 \\
\text{ASR} & \quad \text{when } \text{shift} = 10 \\
\text{ROR} & \quad \text{when } \text{shift} = 11 \\
\langle amount \rangle & \quad \text{For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.} \\
&\text{For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,}
\end{align*}
\]
Operation

bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(datasize) result;

result = operand1 AND operand2;
X[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.14  **ANDS (immediate)**

Bitwise AND (immediate), setting flags, performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias TST (immediate). See *Alias conditions* for details of when each alias is preferred.

### 32-bit variant

Applies when \( sf = 0 \) \&\& \( N = 0 \).

\[
\text{ANDS}<Wd>,<Wn>,#<imm>
\]

### 64-bit variant

Applies when \( sf = 1 \).

\[
\text{ANDS}<Xd>,<Xn>,#<imm>
\]

#### Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } d &= \text{UInt}(\text{Rd}); \\
\text{integer } n &= \text{UInt}(\text{Rn}); \\
\text{integer } \text{datasize} &= \text{if } sf = '1' \text{ then } 64 \text{ else } 32; \\
\text{bits(datasize) } \text{imm}; \\
\text{if } sf = '0' \&\& N != '0' \text{ then UNDEFINED; } \\
(\text{imm}, -) &= \text{DecodeBitMasks}(N, \text{imms}, \text{immr}, \text{TRUE}, \text{datasize});
\end{align*}
\]

### Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>TST (immediate)</td>
<td>( \text{Rd} = '11111' )</td>
</tr>
</tbody>
</table>

### Assembler symbols

- **<Wd>** Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xn>** Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- **<imm>** For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".
Operation

bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];

result = operand1 AND imm;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.15   ANDS (shifted register)

Bitwise AND (shifted register), setting flags, performs a bitwise AND of a register value and an optionally-shifted
register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias TST (shifted register). See Alias conditions for details of when each alias is
preferred.

32-bit variant

Applies when sf == 0.

ANDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit variant

Applies when sf == 1.

ANDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>TST (shifted register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<id>   Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<in>   Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<im>   Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<id>   Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn>   Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm>   Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- **LSL** when shift = 00
- **LSR** when shift = 01
- **ASR** when shift = 10
- **ROR** when shift = 11

For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Operation**

```
bits(datasize) oper1 = X[n, datasize];
bisg(datasize) oper2 = ShiftReg(m, shift_type, shift_amount, datasize);
bisg(datasize) result;
result = oper1 AND oper2;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';
X[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.16   **ASR (register)**

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is an alias of the **ASRV** instruction. This means that:

- The encodings in this description are named to match the encodings of **ASRV**.
- The description of **ASRV** gives the operational pseudocode for this instruction.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

32-bit variant

Applies when $sf == 0$.

`ASR <Wd>, <Wn>, <Wm>`

is equivalent to

`ASRV <Wd>, <Wn>, <Wm>`

and is always the preferred disassembly.

64-bit variant

Applies when $sf == 1$.

`ASR <Xd>, <Xn>, <Xm>`

is equivalent to

`ASRV <Xd>, <Xn>, <Xm>`

and is always the preferred disassembly.

**Assembler symbols**

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

**Operation**

The description of **ASRV** gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.17  ASR (immediate)

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of the sign bit in the upper bits and zeros in the lower bits, and writes the result to the destination register.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

**32-bit variant**

Applies when sf == 0 \&\& N == 0 \&\& imms == 011111.

ASR <Wd>, <Wn>, #<shift>

is equivalent to

SBFM <Wd>, <Wn>, #<shift>, #31

and is always the preferred disassembly.

**64-bit variant**

Applies when sf == 1 \&\& N == 1 \&\& imms == 111111.

ASR <Xd>, <Xn>, #<shift>

is equivalent to

SBFM <Xd>, <Xn>, #<shift>, #63

and is always the preferred disassembly.

**Assembler symbols**

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <shift> For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field.

**Operation**

The description of SBFM gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.18 ASRV

Arithmetic Shift Right Variable shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ASR (register). The alias is always the preferred disassembly.

32-bit variant
Applies when \( sf = 0 \).
ASRV <Wd>, <Wn>, <Wm>

64-bit variant
Applies when \( sf = 1 \).
ASRV <Xd>, <Xn>, <Xm>

Decode for all variants of this encoding

\[
\begin{array}{cccccccccccccccc}
\hline
\text{sf} & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & \text{Rm} & 0 & 0 & 1 & 0 & 1 & 0 & \text{Rn} & \text{Rd} \\
\text{op2} & & & & & & & & & & & & & & & & & & & & & \\
\end{array}
\]

Assembler symbols

\(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
\(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
\(<Wm>\) Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
\(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
\(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
\(<Xm>\) Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

\[
\begin{align*}
\text{bits(datasize)} & \text{ result;} \\
\text{bits(datasize)} & \text{ operand2} = X[m, \text{datasize}]; \\
\text{result} & = \text{ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);} \\
X[d, \text{datasize}] & = \text{result}; \\
\end{align*}
\]
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.19 AT

Address Translate. For more information, see op0==0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, and BRBE instructions on page C5-659.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>AT &lt;at_op&gt;, &lt;Xt&gt; is equivalent to</th>
<th>SYS #&lt;op1&gt;, C7, &lt;Cm&gt;, #&lt;op2&gt;, &lt;Xt&gt; and is the preferred disassembly when SysOp(op1,'0111',CRm,op2) == Sys_AT.</th>
</tr>
</thead>
</table>

### Encoder

<at_op> Is an AT instruction name, as listed for the AT system instruction group, encoded in the "op1:CRm<0>:op2" field. It can have the following values:

- S1E1R when op1 = 000, CRm<0> = 0, op2 = 000
- S1E1W when op1 = 000, CRm<0> = 0, op2 = 001
- S1E0R when op1 = 000, CRm<0> = 0, op2 = 010
- S1E0W when op1 = 000, CRm<0> = 0, op2 = 011
- S1E2R when op1 = 100, CRm<0> = 0, op2 = 000
- S1E2W when op1 = 100, CRm<0> = 0, op2 = 001
- S12E1R when op1 = 100, CRm<0> = 0, op2 = 100
- S12E1W when op1 = 100, CRm<0> = 0, op2 = 101
- S12E0R when op1 = 100, CRm<0> = 0, op2 = 110
- S12E0W when op1 = 100, CRm<0> = 0, op2 = 111
- S1E3R when op1 = 110, CRm<0> = 0, op2 = 000
- S1E3W when op1 = 110, CRm<0> = 0, op2 = 001

When FEAT_PAN2 is implemented, the following values are also valid:

- S1E1RP when op1 = 000, CRm<0> = 1, op2 = 000
- S1E1WP when op1 = 000, CRm<0> = 1, op2 = 001

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.
Operation

The description of SYS gives the operational pseudocode for this instruction.
AUTDA, AUTDZA

Authenticate Data address, using key A. This instruction authenticates a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTDA.
- The value zero, for AUTDZA.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

Integer

(_FEAT_PAuth)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>31 19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**AUTDA variant**

Applies when Z == 0.

AUTDA <Xd>, <Xn|SP>

**AUTDZA variant**

Applies when Z == 1 && Rn == 11111.

AUTDZA <Xd>

Decode for all variants of this encoding

```java
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // AUTDA
    if n == 31 then source_is_sp = TRUE;
else // AUTDZA
    if n != 31 then UNDEFINED;
```

Assemble symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```java
if HavePACExt() then
    if source_is_sp then
        X[d, 64] = AuthDA(X[d, 64], SP[], FALSE);
    else
        X[d, 64] = AuthDA(X[d, 64], X[n, 64], FALSE);
```
C6.2.21 AUTDB, AUTDZB

Authenticate Data address, using key B. This instruction authenticates a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by `<Xd>`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by `<Xn|SP>` for AUTDB.
- The value zero, for AUTDZB.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

**Integer**

(FEAT_PAuth)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 1 1 0 1 0 0 0 0 0 1 0 0 0 1 1 1 1</td>
<td></td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**AUTDB variant**

Applies when `Z == 0`.

AUTDB `<Xd>, <Xn|SP>`

**AUTDZB variant**

Applies when `Z == 1 && Rn == 11111`.

AUTDZB `<Xd>`

**Decode for all variants of this encoding**

```cpp
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;
if Z == '0' then // AUTDB
    if n == 31 then source_is_sp = TRUE;
else // AUTDZB
    if n != 31 then UNDEFINED;
```

**Assembler symbols**

- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

**Operation**

```cpp
if HavePACExt() then
    if source_is_sp then
        X[d, 64] = AuthDB(X[d, 64], SP[], FALSE);
    else
        X[d, 64] = AuthDB(X[d, 64], X[n, 64], FALSE);
```
C6.2.22 AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA

Authenticate Instruction address, using key A. This instruction authenticates an instruction address, using a modifier and key A.

The address is:

• In the general-purpose register that is specified by <Xd> for AUTIA and AUTIZA.
• In X17, for AUTIA1716.
• In X30, for AUTIASP and AUTIAZ.

The modifier is:

• In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTIA.
• The value zero, for AUTIZA and AUTIAZ.
• In X16, for AUTIA1716.
• In SP, for AUTIASP.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

Integer

(FEAT_PAuth)

\[
\begin{array}{cccccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & Rn & Rd
\end{array}
\]

AUTIA variant

Applies when \( Z == 0 \).

AUTIA <Xd>, <Xn|SP>

AUTIZA variant

Applies when \( Z == 1 \) \&\& \( Rn == 11111 \).

AUTIZA <Xd>

Decode for all variants of this encoding

```c
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
    UNDEFINED;
if Z == '0' then // AUTIA
    if n == 31 then source_is_sp = TRUE;
else // AUTIZA
    if n != 31 then UNDEFINED;
```

System

(FEAT_PAuth)
A64 Base Instruction Descriptions
C6.2 Alphabetical list of A64 base instructions

AUTIA1716 variant
Applies when CRm == 0001 && op2 == 100.

AUTIA1716

AUTIASP variant
Applies when CRm == 0011 && op2 == 101.

AUTIASP

AUTIAZ variant
Applies when CRm == 0011 && op2 == 100.

AUTIAZ

Decode for all variants of this encoding

integer d;
integer n;
boolean source_is_sp = FALSE;
case CRm:op2 of
    when '0011 100'    // AUTIAZ
        d = 30;
        n = 31;
    when '0011 101'    // AUTIASP
        d = 30;
        source_is_sp = TRUE;
    when '0001 100'    // AUTIA1716
        d = 17;
        n = 16;
    when '0001 000' SEE "PACIA";
    when '0001 010' SEE "PACIB";
    when '0001 110' SEE "AUTIB";
    when '0011 00x' SEE "PACIA";
    when '0011 01x' SEE "PACIB";
    when '0011 11x' SEE "AUTIB";
    when '0000 111' SEE "XPACLRI";
    otherwise SEE "HINT";

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation for all encodings

if HavePACExt() then
    if source_is_sp then
        X[d, 64] = AuthIA(X[d, 64], SP[], FALSE);
    else
        X[d, 64] = AuthIA(X[d, 64], X[n, 64], FALSE);
C6.2.23   AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB

Authenticate Instruction address, using key B. This instruction authenticates an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by <Xd> for AUTIB and AUTIZB.
- In X17, for AUTIB1716.
- In X30, for AUTIBSP and AUTIBZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTIB.
- The value zero, for AUTIZB and AUTIBZ.
- In X16, for AUTIB1716.
- In SP, for AUTIBSP.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

**Integer**

(FEAT_PAuth)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**AUTIB variant**

Applies when Z == 0.

AUTIB <Xd>, <Xn|SP>

**AUTIZB variant**

Applies when Z == 1 && Rn == 11111.

AUTIZB <Xd>

**Decode for all variants of this encoding**

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // AUTIB
    if n == 31 then source_is_sp = TRUE;
else // AUTIZB
    if n != 31 then UNDEFINED;

**System**

(FEAT_PAuth)
AUTIB1716 variant

Applies when CRm == 0001 && op2 == 110.

AUTIB1716

AUTIBSP variant

Applies when CRm == 0011 && op2 == 111.

AUTIBSP

AUTIBZ variant

Applies when CRm == 0011 && op2 == 110.

AUTIBZ

Decode for all variants of this encoding

integer d;
integer n;
boolean source_is_sp = FALSE;
case CRm:op2 of
  when '0011 110' // AUTIBZ
    d = 30;
    n = 31;
  when '0011 111' // AUTIBSP
    d = 30;
    source_is_sp = TRUE;
  when '0001 110' // AUTIB1716
    d = 17;
    n = 16;
when '0001 000' SEE "PACIA";
when '0001 010' SEE "PACIB";
when '0001 100' SEE "AUTIA";
when '0011 00x' SEE "PACIA";
when '0011 01x' SEE "PACIB";
when '0011 10x' SEE "AUTIA";
when '0000 111' SEE "XPACLRI";
otherwise SEE "HINT";

Assembler symbols

<Xd> is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP> is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation for all encodings

if HavePACExt() then
  if source_is_sp then
    \[X[d, 64] = AuthIB(X[d, 64], SP[], FALSE)\];
  else
    \[X[d, 64] = AuthIB(X[d, 64], X[n, 64], FALSE)\];
C6.2.24  AXFLAG

Convert floating-point condition flags from Arm to external format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from a form representing the result of an Arm floating-point scalar compare instruction to an alternative representation required by some software.

System
(FEAT_FlagM2)


Encoding
AXFLAG

Decode for this encoding
if !HaveFlagFormatExt() then UNDEFINED;

Operation
    bit z = PSTATE.Z OR PSTATE.V;
    bit c = PSTATE.C AND NOT(PSTATE.V);

    PSTATE.N = '0';
    PSTATE.Z = z;
    PSTATE.C = c;
    PSTATE.V = '0';
C6.2.25  B

Branch causes an unconditional branch to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

\[
\begin{array}{c|c|c|c|c|c|c|c}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & \text{imm26} & \text{op}
\end{array}
\]

**Encoding**

B <label>

**Decode for this encoding**

bits(64) offset = SignExtend(imm26:’00’, 64);

**Assembler symbols**

<label> Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

**Operation**

BranchTo(PC[] + offset, BranchType_DIR, FALSE);
C6.2.26   B.cond

Branch conditionally to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

![Instruction Encoding](https://example.com/encoding.png)

**Encoding**

B.<cond> <label>

**Decode for this encoding**

bits(64) offset = SignExtend(imm19:'00', 64);

**Assembler symbols**

<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

**Operation**

if ConditionHolds(cond) then
    BranchTo(PC[] + offset, BranchType_DIR, TRUE);
C6.2.27  BC.cond

Branch Consistent conditionally to a label at a PC-relative offset, with a hint that this branch will behave very consistently and is very unlikely to change direction.

19-bit signed PC-relative branch offset

(FEAT_HBC)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23</th>
<th>5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 0 1 0 0</td>
<td>imm19</td>
<td>1</td>
<td>cond</td>
</tr>
</tbody>
</table>

Encoding

BC.<cond> <label>

Decode for this encoding

if !HaveFeatHBC() then UNDEFINED;
bits(64) offset = SignExtend(imm19:'00', 64);

Assembler symbols

<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

if ConditionHolds(cond) then
   BranchTo(PC[] + offset, BranchType_DIR, TRUE);
C6.2.28 BFC

Bitfield Clear sets a bitfield of <width> bits at bit position <lsb> of the destination register to zero, leaving the other destination bits unchanged.

This instruction is an alias of the BFM instruction. This means that:

• The encodings in this description are named to match the encodings of BFM.
• The description of BFM gives the operational pseudocode for this instruction.

Leaving other bits unchanged

(Feature_ASMv8p2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21</th>
<th>16 15</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf 0 1 0 1 0 1 0 N immr</td>
<td>imms 1 1 1 1 Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit variant

Applies when sf == 0 && N == 0.

BFC <Wd>, #<lsb>, #<width>

is equivalent to

BFM <Wd>, WZR, #(-<lsb> MOD 32), #(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

64-bit variant

Applies when sf == 1 && N == 1.

BFC <Xd>, #<lsb>, #<width>

is equivalent to

BFM <Xd>, XZR, #(-<lsb> MOD 64), #(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<lsb> For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

<width> For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of BFM gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.29   BFI

Bitfield Insert copies a bitfield of `<width>` bits from the least significant bits of the source register to bit position `<lsb>` of the destination register, leaving the other destination bits unchanged.

This instruction is an alias of the BFM instruction. This means that:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when `<sf == 0 && N == 0>`.

BFI `<Wd>, <Wn>, #<lsb>, #<width>`

is equivalent to

BFM `<Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)`

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

### 64-bit variant

Applies when `<sf == 1 && N == 1>`.

BFI `<Xd>, <Xn>, #<lsb>, #<width>`

is equivalent to

BFM `<Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)`

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

### Assembler symbols

- `<Wd>`  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`  Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd>`  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`  Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<lsb>`  For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.
  For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
- `<width>`  For the 32-bit variant: is the width of the bitfield, in the range 1 to 32.<lsb>.
  For the 64-bit variant: is the width of the bitfield, in the range 1 to 64.<lsb>.

### Operation

The description of BFM gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.30 BFM

Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases the other bits of the destination register remain unchanged.

This instruction is used by the aliases BFC, BFI, and BFXIL. See Alias conditions on page C6-1222 for details of when each alias is preferred.

32-bit variant

Applies when sf == 0 && N == 0.

BFM <Wd>, <Wn>, #<immr>, #<imms>

64-bit variant

Applies when sf == 1 && N == 1.

BFM <Xd>, <Xn>, #<immr>, #<imms>

Decode for all variants of this encoding

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

integer r;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;

r = UInt(immr);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
```


**Alias conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFC</td>
<td>Rn == '11111' &amp;&amp; UInt(imms) &lt; UInt(immr)</td>
</tr>
<tr>
<td>BFI</td>
<td>Rn != '11111' &amp;&amp; UInt(imms) &lt; UInt(immr)</td>
</tr>
<tr>
<td>BFXIL</td>
<td>UInt(imms) &gt;= UInt(immr)</td>
</tr>
</tbody>
</table>

**Assembler symbols**

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<immr>`: For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
- `<imms>`: For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

**Operation**

```
bits(datasize) dst = X[d, datasize];
bits(datasize) src = X[n, datasize];

// perform bitfield move on low bits
bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, r) AND wmask);

// combine extension bits and result bits
X[d, datasize] = (dst AND NOT(tmask)) OR (bot AND tmask);
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.31 BFXIL

Bitfield Extract and Insert Low copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, leaving the other destination bits unchanged.

This instruction is an alias of the BFM instruction. This means that:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when sf == 0 && N == 0.

BFXIL <Wd>, <Wn>, #<lsb>, #<width>

is equivalent to

BFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)

and is the preferred disassembly when UInt(imms) >= UInt(immr).

### 64-bit variant

Applies when sf == 1 && N == 1.

BFXIL <Xd>, <Xn>, #<lsb>, #<width>

is equivalent to

BFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)

and is the preferred disassembly when UInt(imms) >= UInt(immr).

### Assembler symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <lsb> For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.
  For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
- <width> For the 32-bit variant: is the width of the bitfield, in the range 1 to 32.<lsb>.
  For the 64-bit variant: is the width of the bitfield, in the range 1 to 64.<lsb>.

### Operation

The description of BFM gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.32   BIC (shifted register)

Bitwise Bit Clear (shifted register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

32-bit variant

Applies when \( sf == 0 \).

\[ \text{BIC } <\text{Wd}>, <\text{Wn}>, <\text{Wm}>{{, \text{<shift> #<amount>}}} \]

64-bit variant

Applies when \( sf == 1 \).

\[ \text{BIC } <\text{Xd}>, <\text{Xn}>, <\text{Xm}>{{, \text{<shift> #<amount>}}} \]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } d &= \text{UInt}\left(\text{Rd}\right); \\
\text{integer } n &= \text{UInt}\left(\text{Rn}\right); \\
\text{integer } m &= \text{UInt}\left(\text{Rm}\right); \\
\text{integer } \text{datasize} &= \text{if } sf == '1' \text{ then 64 else 32}; \\
&\text{if } sf == '0' \text{ & & imm6<5> == '1' then UNDEFINED;} \\
\text{ShiftType } \text{shift_type} &= \text{DecodeShift}(\text{shift}); \\
\text{integer } \text{shift_amount} &= \text{UInt}(\text{imm6}); \\
\end{align*}
\]

Assembler symbols

- \(<\text{Wd}>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<\text{Wn}>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<\text{Wm}>\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<\text{Xd}>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<\text{Xn}>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<\text{Xm}>\) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<\text{<shift>}>\) Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:
  - LSL when \( \text{shift} = 00 \)
  - LSR when \( \text{shift} = 01 \)
  - ASR when \( \text{shift} = 10 \)
  - ROR when \( \text{shift} = 11 \)
- \(<\text{<amount>}>\) For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.
**Operation**

```plaintext
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(datasize) result;
operand2 = NOT(operand2);
result = operand1 AND operand2;
X[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.33 BICS (shifted register)

Bitwise Bit Clear (shifted register), setting flags, performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
</tr>
</tbody>
</table>

32-bit variant

Applies when \( sf == 0 \).

\[
\text{BICS } <Wd>, <Wn>, <Wm\{, <shift> #<amount>}\}
\]

64-bit variant

Applies when \( sf == 1 \).

\[
\text{BICS } <Xd>, <Xn>, <Xm\{, <shift> #<amount>}\}
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{integer } \text{datasize} &= \text{if } sf == '1' \text{ then } 64 \text{ else } 32; \\
\text{if } sf == '0' \&\& \text{imm6<5>} == '1' \text{ then UNDEFINED}; \\
\text{ShiftType } \text{shift_type} &= \text{DecodeShift(shift)}; \\
\text{integer } \text{shift_amount} &= \text{UInt}(\text{imm6});
\end{align*}
\]

Assembler symbols

\[
\begin{align*}
\text{<Wd>} & \quad \text{Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.} \\
\text{<Wn>} & \quad \text{Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.} \\
\text{<Wm>} & \quad \text{Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.} \\
\text{<Xd>} & \quad \text{Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.} \\
\text{<Xn>} & \quad \text{Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.} \\
\text{<Xm>} & \quad \text{Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.} \\
\text{<shift>} & \quad \text{Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:} \\
\text{LSL} & \quad \text{when } \text{shift} = 00 \\
\text{LSR} & \quad \text{when } \text{shift} = 01 \\
\text{ASR} & \quad \text{when } \text{shift} = 10 \\
\text{ROR} & \quad \text{when } \text{shift} = 11 \\
\text{<amount>} & \quad \text{For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.}
\end{align*}
\]
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

**Operation**

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(datasize) result;
operand2 = NOT(operand2);
result = operand1 AND operand2;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';
X[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.34   BL

Branch with Link branches to a PC-relative offset, setting the register X30 to PC+4. It provides a hint that this is a subroutine call.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25</th>
<th>24 23 22</th>
<th>21 20 19</th>
<th>18 17 16</th>
<th>15 14 13</th>
<th>12 11 10</th>
<th>9 8 7 6</th>
<th>5 4 3 2</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 0</td>
<td>1 0 1</td>
<td>imm26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

BL <label>

**Decode for this encoding**

```
bits(64) offset = SignExtend(imm26:'00', 64);
```

**Assembler symbols**

<label> Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

**Operation**

```
X[30, 64] = PC[] + 4;

BranchTo(PC[] + offset, BranchType_DIRCALL, FALSE);
```
C6.2.35   BLR

Branch with Link to Register calls a subroutine at an address in a register, setting register X30 to PC+4.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9  | 5 4  | 0 ]
  1 1 0 1 0 1 1 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  Z  op  A  M  Rn  Rm
```

**Encoding**

BLR <Xn>

**Decode for this encoding**

```python
integer n = UInt(Rn);
```

**Assembler symbols**

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

**Operation**

```python
bits(64) target = X[n, 64];
X[30, 64] = PC[] + 4;
// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10';
BranchTo(target, BranchType_INDCALL, FALSE);
```
C6.2.36   BLRAA, BLRAAZ, BLRAB, BLRABZ

Branch with Link to Register, with pointer authentication. This instruction authenticates the address in the
general-purpose register that is specified by <Xn>, using a modifier and the specified key, and calls a subroutine at
the authenticated address, setting register X30 to PC+4.

The modifier is:

• In the general-purpose register or stack pointer that is specified by <Xm|SP> for BLRAA and BLRAB.
• The value zero, for BLRAAZ and BLRABZ.

Key A is used for BLRAA and BLRAAZ, and key B is used for BLRAB and BLRABZ.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a
Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

Integer

(FEAT_PAuth)

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|1 1 0 1 0 1 1 | Z 0 0 1 | 1 1 1 1 | 0 0 0 | 0 | 1 | M | Rn | Rm |
```

Key A, zero modifier variant

Applies when \( Z == 0 \land M == 0 \land Rm == 11111 \).

BLRAA <Xn>

Key A, register modifier variant

Applies when \( Z == 1 \land M == 0 \).

BLRAA <Xn>, <Xm|SP>

Key B, zero modifier variant

Applies when \( Z == 0 \land M == 1 \land Rm == 11111 \).

BLRABZ <Xn>

Key B, register modifier variant

Applies when \( Z == 1 \land M == 1 \).

BLRAB <Xn>, <Xm|SP>

Decode for all variants of this encoding

```python
define n = UInt(Rn);
define m = UInt(Rm);
define use_key_a = (M == '0');
define source_is_sp = ((Z == '1') && (m == 31));

if !HavePACExt() then
    UNDEFINED;

if Z == '0' && m != 31 then
    UNDEFINED;
```

[1101011]
0x0 Z0 01 11111 00001 M Rn Rm
Assembler symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

<Xm|SP> Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.

Operation

bits(64) target = X[n, 64];
bits(64) modifier = if source_is_sp then SP[] else X[m, 64];

if use_key_a then
  target = AuthIA(target, modifier, TRUE);
else
  target = AuthIB(target, modifier, TRUE);

X[30, 64] = PC[] + 4;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10';
BranchTo(target, BranchType_INDCALL, FALSE);
C6.2.37 BR

Branch to Register branches unconditionally to an address in a register, with a hint that this is not a subroutine return.

Encoding
BR <Xn>

Decode for this encoding
integer n = UInt(Rn);

Assembler symbols
<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Operation
bits(64) target = X[n, 64];

// Value in BTypeNext will be used to set PSTATE.BTYPE
if InGuardedPage then
  if n == 16 || n == 17 then
    BTypeNext = '01';
  else
    BTypeNext = '11';
else
  BTypeNext = '01';
BranchTo(target, BranchType_INDIR, FALSE);
C6.2.38  BRAA, BRAAZ, BRAB, BRABZ

Branch to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by <Xn>, using a modifier and the specified key, and branches to the authenticated address.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xm|SP> for BRAA and BRAB.
- The value zero, for BRAAZ and BRABZ.

Key A is used for BRAA and BRAAZ, and key B is used for BRAB and BRABZ.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

**Integer**

(FEAT_PAuth)

```
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 5 | 4 | 0 |
```

**Key A, zero modifier variant**

Applies when Z == 0 && M == 0 && Rm == 11111.

BRAA <Xn>, <Xm|SP>

**Key A, register modifier variant**

Applies when Z == 1 && M == 0.

BRAAZ <Xn>

**Key B, zero modifier variant**

Applies when Z == 0 && M == 1 && Rm == 11111.

BRAB <Xn>, <Xm|SP>

**Key B, register modifier variant**

Applies when Z == 1 && M == 1.

BRABZ <Xn>, <Xm|SP>

**Decode for all variants of this encoding**

```
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !HavePACExt() then
    UNDEFINED;

if Z == '0' && m != 31 then
    UNDEFINED;
```
Assembler symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

<Xm|SP> Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.

Operation

\[
\begin{align*}
\text{bits}(64) \ target &= \ X[n, 64]; \\
\text{bits}(64) \ modifier &= \text{if } \text{source}\_\text{is}\_\text{sp} \ \text{then} \ SP[] \ \text{else} \ X[m, 64]; \\
\text{if use_key_a then} \\
\text{\quad target} &= \text{AuthIA}(target, modifier, \text{TRUE}); \\
\text{else} \\
\text{\quad target} &= \text{AuthIB}(target, modifier, \text{TRUE}); \\
\text{// Value in BTypeNext will be used to set PSTATE.BTYPE} \\
\text{if InGuardedPage then} \\
\text{\quad if } n == 16 || n == 17 \ \text{then} \\
\text{\quad \quad BTypeNext} &= '01'; \\
\text{\quad else} \\
\text{\quad \quad BTypeNext} &= '11'; \\
\text{else} \\
\text{\quad BTypeNext} &= '01'; \\
\text{BranchTo} (target, \text{BranchType_INDIR, FALSE});
\end{align*}
\]
C6.2.39   BRB

Branch Record Buffer. For more information, see \textit{op0==0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, and BRBE instructions} on page C5-659.

This instruction is an alias of the \textit{SYS} instruction. This means that:

\begin{itemize}
  \item The encodings in this description are named to match the encodings of \textit{SYS}.
  \item The description of \textit{SYS} gives the operational pseudocode for this instruction.
\end{itemize}

\textbf{System}

(\texttt{FEAT\_BRBE})

\begin{verbatim}
0100 0001 0111 0010 op2 Rt
L   op1  CRn  CRm
\end{verbatim}

\textbf{Encoding}

\texttt{BRB \{\texttt{brb\_op}\}, \{\texttt{Xt}\}}

is equivalent to

\texttt{SYS \#1, C7, C2, \#op2\{, \texttt{Xt}\}}

and is the preferred disassembly when \texttt{SysOp('001', '0111', '0010', op2) == Sys\_BRB}.

\textbf{Assembler symbols}

\begin{itemize}
  \item \texttt{\{\texttt{brb\_op}\}} is a BRB instruction name, as listed for the BRB system instruction group, encoded in the "op2" field. It can have the following values:
  \begin{itemize}
    \item IALL when \texttt{op2 = 100}
    \item INJ when \texttt{op2 = 101}
  \end{itemize}
  \item \texttt{\{\texttt{op2}\}} is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
  \item \texttt{\{\texttt{Xt}\}} is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.
\end{itemize}

\textbf{Operation}

The description of \textit{SYS} gives the operational pseudocode for this instruction.
C6.2.40  BRK

Breakpoint instruction. A BRK instruction generates a Breakpoint Instruction exception. The PE records the exception in ESR_ELx, using the EC value 0x3c, and captures the value of the immediate argument in ESR_ELx.ISS.

\[
\begin{array}{cccccccccc}
| & | & | & | & | & | & | & | & | & \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & imm\text{16} & 0 & 0 & 0 & 0
\end{array}
\]

**Encoding**

BRK #<imm>

**Decode for this encoding**

if HaveBTIEExt() then
    SetBTypeCompatible(TRUE);

**Assembler symbols**

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

**Operation**

AArch64.SoftwareBreakpoint(imm16);
C6.2.41 BTI

Branch Target Identification. A BTI instruction is used to guard against the execution of instructions which are not the intended target of a branch.

Outside of a guarded memory region, a BTI instruction executes as a NOP. Within a guarded memory region while \( \text{PSTATE.BTYPE} \neq 0b00 \), a BTI instruction compatible with the current value of PSTATE.BTYPE will not generate a Branch Target Exception and will allow execution of subsequent instructions within the memory region.

The operand \(<\text{targets}>\) passed to a BTI instruction determines the values of PSTATE.BTYPE which the BTI instruction is compatible with.

\[\text{Note}\]
Within a guarded memory region, when \( \text{PSTATE.BTYPE} \neq 0b00 \), all instructions will generate a Branch Target Exception, other than BRK, BTI,HLT, PACIASP, and PACIBSP, which might not. See the individual instructions for more information.

**System**

(FEAT_BTI)

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 8 7 5 4 3 2 1 0 ]
1 1 0 1 0 1 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 1 1 1 1
```

**Encoding**

BTI \(<\text{targets}>\)

**Decode for this encoding**

```
SystemHintOp op;
if CRm:op2 == '0100 xx0' then
  op = SystemHintOp_BTI;
  // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
  SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
else
  EndOfInstruction();
```

**Assemble symbols**

\(<\text{targets}>\) Is the type of indirection, encoded in the "op2<2:1>" field. It can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(omitted)</td>
<td>when op2&lt;2:1&gt; = 00</td>
</tr>
<tr>
<td>c</td>
<td>when op2&lt;2:1&gt; = 01</td>
</tr>
<tr>
<td>j</td>
<td>when op2&lt;2:1&gt; = 10</td>
</tr>
<tr>
<td>jc</td>
<td>when op2&lt;2:1&gt; = 11</td>
</tr>
</tbody>
</table>

**Operation**

```
case op of
  when SystemHintOp_YIELD
    Hint_Yield();
  when SystemHintOp_DGH
    Hint_DGH();
  when SystemHintOp_WFE
```
integer localtimeout = 1 << 64; // No local timeout event is generated
Hint_WFE(localtimeout, WFxType_WFE);

when SystemHintOp_WFI
integer localtimeout = 1 << 64; // No local timeout event is generated
Hint_WFI(localtimeout, WFxType_WFI);

when SystemHintOp_SEV
SendEvent();

when SystemHintOp_SEVL
SendEventLocal();

when SystemHintOp_ESB
if HaveTME() && TSTATE.depth > 0 then
  FailTransaction(TMFailure_ERR, FALSE);
  SynchronizeErrors();
  AArch64.ESB0peration();
  if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.ESB0peration();
  TakeUnmaskedSErrorInterrupts();

when SystemHintOp_PSB
  ProfilingSynchronizationBarrier();

when SystemHintOp_TSB
  TraceSynchronizationBarrier();

when SystemHintOp_CSDB
  ConsumptionOfSpeculativeDataBarrier();

when SystemHintOp_BTI
  SetBTypeNext('00');

otherwise // do nothing
C6.2.42 CASB, CASAB, CASALB, CASLB

Compare and Swap byte in memory reads an 8-bit byte from memory, and compares it against the value held in a
first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed,
the read and write occur atomically such that no other modification of the memory location can take place between
the read and write.

- CASAB and CASALB load from memory with acquire semantics.
- CASLB and CASALB store to memory with release semantics.
- CASB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on
page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the
compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is
restored to the values held in the register before the instruction was executed.

No offset

(FEAT_LSE)

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9  |  5  4  |  0  
   0  0 | 0  1  0  0  0  1 | L  1 | Rs  o0  1  1  1  1 | Rn  | Rt  
size
```

CASAB variant

Applies when L == 1 && o0 == 0.
CASAB <Ws>, <Wt>, [<Xn|SP>{,#0}]

CASALB variant

Applies when L == 1 && o0 == 1.
CASALB <Ws>, <Wt>, [<Xn|SP>{,#0}]

CASB variant

Applies when L == 0 && o0 == 0.
CASB <Ws>, <Wt>, [<Xn|SP>{,#0}]

CASLB variant

Applies when L == 0 && o0 == 1.
CASLB <Ws>, <Wt>, [<Xn|SP>{,#0}]

Decode for all variants of this encoding

```c
if (HaveAtomicExt()) then UNDEFINED;
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
```
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) comparevalue;
bits(8) newvalue;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

    comparevalue = X[s, 8];
    newvalue = X[t, 8];

    if n == 31 then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n, 64];

    data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);

    X[s, 32] = ZeroExtend(data, 32);
C6.2.43  CASH, CASAH, CASALH, CASLH

Compare and Swap halfword in memory reads a 16-bit halfword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASAH and CASALH load from memory with acquire semantics.
- CASLH and CASALH store to memory with release semantics.
- CAS has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

No offset

(FEAT_LSE)

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 0 0 1 0 0 1 | L 1 | Rs o0 1 1 1 1 | Rn | Rt
```

CASAH variant

Applies when L == 1 && o0 == 0.

CASAH <Ws>, <Wt>, [<Xn|SP>{,#0}]

CASALH variant

Applies when L == 1 && o0 == 1.

CASALH <Ws>, <Wt>, [<Xn|SP>{,#0}]

CASH variant

Applies when L == 0 && o0 == 0.

CASH <Ws>, <Wt>, [<Xn|SP>{,#0}]

CASLH variant

Applies when L == 0 && o0 == 1.

CASLH <Ws>, <Wt>, [<Xn|SP>{,#0}]

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UNDEFINED;
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
```
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) comparevalue;
bits(16) newvalue;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

comparevalue = X[s, 16];
newvalue = X[t, 16];

if n == 31 then
    CheckSPAAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);

X[s, 32] = ZeroExtend(data, 32);
C6.2.44 CASP, CASPA, CASPAL, CASPL

Compare and Swap Pair of words or doublewords in memory reads a pair of 32-bit words or 64-bit doublewords from memory, and compares them against the values held in the first pair of registers. If the comparison is equal, the values in the second pair of registers are written to memory. If the writes are performed, the reads and writes occur atomically such that no other modification of the memory location can take place between the reads and writes.

- CASPA and CASPAL load from memory with acquire semantics.
- CASPL and CASPAL store to memory with release semantics.
- CAS has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the registers which are compared and loaded, that is \(<W_s>\) and \(<W_{s+1}>\), or \(<X_s>\) and \(<X_{s+1}>\), are restored to the values held in the registers before the instruction was executed.

**No offset**

(FEAT_LSE)

```
[31 30 29 28][27 26 25 24][23 22 21 20] 16 15 14 10 9 5 4 0 0 | L | Rs | o0 | Rn | Rt |
```

32-bit CASP variant

Applies when \(sz = 0 \&\& L = 0 \&\& o0 = 0\).

\(\text{CASP}\ <W_s>, <W_{s+1}>, <W_t>, <W_{t+1}>, [<Xn|SP>{, #0}]\)

32-bit CASPA variant

Applies when \(sz = 0 \&\& L = 1 \&\& o0 = 0\).

\(\text{CASPA}\ <W_s>, <W_{s+1}>, <W_t>, <W_{t+1}>, [<Xn|SP>{, #0}]\)

32-bit CASPAL variant

Applies when \(sz = 0 \&\& L = 1 \&\& o0 = 1\).

\(\text{CASPAL}\ <W_s>, <W_{s+1}>, <W_t>, <W_{t+1}>, [<Xn|SP>{, #0}]\)

32-bit CASPL variant

Applies when \(sz = 0 \&\& L = 0 \&\& o0 = 1\).

\(\text{CASPL}\ <W_s>, <W_{s+1}>, <W_t>, <W_{t+1}>, [<Xn|SP>{, #0}]\)

64-bit CASP variant

Applies when \(sz = 1 \&\& L = 0 \&\& o0 = 0\).

\(\text{CASP}\ <X_s>, <X_{s+1}>, <X_t>, <X_{t+1}>, [<Xn|SP>{, #0}]\)
64-bit CASPA variant
Applies when sz == 1 && L == 1 && o0 == 0.
CASPA <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]

64-bit CASPAL variant
Applies when sz == 1 && L == 1 && o0 == 1.
CASPAL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]

64-bit CASPL variant
Applies when sz == 1 && L == 0 && o0 == 1.
CASPL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]

Decode for all variants of this encoding

if !HaveAtomicExt() then UNDEFINED;
if Rs<0> == '1' then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
integer datasize = 32 << UInt(sz);
AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler symbols

<Ws> Is the 32-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Ws> must be an even-numbered register.

<W(s+1)> Is the 32-bit name of the second general-purpose register to be compared and loaded.

<Xt> Is the 32-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Xt> must be an even-numbered register.

<X(t+1)> Is the 32-bit name of the second general-purpose register to be conditionally stored.

<Xs> Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Xs> must be an even-numbered register.

<X(s+1)> Is the 64-bit name of the second general-purpose register to be compared and loaded.

<XT> Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Xt> must be an even-numbered register.

<X(t+1)> Is the 64-bit name of the second general-purpose register to be conditionally stored.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(2*datasize) comparevalue;
b't(2*datasize) newvalue;
b't(2*datasize) data;
bits(datasize) s1 = X[s, datasize];
bits(datasize) s2 = X[s+1, datasize];
bits(datasize) t1 = X[t, datasize];
bits(datasize) t2 = X[t+1, datasize];
comparevalue = if BigEndian(ldacctype) then s1:s2 else s2:s1;
newvalue = if BigEndian(stacctype) then t1:t2 else t2:t1;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAAlignment();
    address = SP[];
else
    address = X[n, 64];
data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);

if BigEndian(ldacctype) then
    X[s, datasize] = data<2*datasize-1:datasize>;
    X[s+1, datasize] = data<datasize-1:0>;
else
    X[s, datasize] = data<datasize-1:0>;
    X[s+1, datasize] = data<2*datasize-1:datasize>;
C6.2.45 CAS, CASA, CASAL, CASL

Compare and Swap word or doubleword in memory reads a 32-bit word or 64-bit doubleword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASA and CASAL load from memory with acquire semantics.
- CASL and CASAL store to memory with release semantics.
- CAS has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, or <Xs>, is restored to the value held in the register before the instruction was executed.

No offset

(FEAT_LSE)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x 0 0 1 0 0 1 L 1 Rs o0 1 1 1 1 Rn</td>
<td>Rn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size

32-bit CAS variant

Applies when size == 10 && L == 0 && o0 == 0.

CAS <Ws>, <Wt>, [<Xn|SP>{,#0}]

32-bit CASA variant

Applies when size == 10 && L == 1 && o0 == 0.

CASA <Ws>, <Wt>, [<Xn|SP>{,#0}]

32-bit CASAL variant

Applies when size == 10 && L == 1 && o0 == 1.

CASAL <Ws>, <Wt>, [<Xn|SP>{,#0}]

32-bit CASL variant

Applies when size == 10 && L == 0 && o0 == 1.

CASL <Ws>, <Wt>, [<Xn|SP>{,#0}]

64-bit CAS variant

Applies when size == 11 && L == 0 && o0 == 0.

CAS <Xs>, <Xt>, [<Xn|SP>{,#0}]

64-bit CASA variant

Applies when size == 11 && L == 1 && o0 == 0.
CASA <Xs>, <Xt>, [<Xn|SP>{,#0}]

64-bit CASAL variant

Applies when size == 11 && L == 1 && o0 == 1.

CASA <Xs>, <Xt>, [<Xn|SP>{,#0}]

64-bit CASL variant

Applies when size == 11 && L == 0 && o0 == 1.

CASL <Xs>, <Xt>, [<Xn|SP>{,#0}]

Decode for all variants of this encoding

if !HaveAtomicExt() then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;

AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(datasize) comparevalue;
bits(datasize) newvalue;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

comparevalue = X[s, datasize];
newvalue = X[t, datasize];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);
$X[s, \text{regsize}] = \text{ZeroExtend}(\text{data}, \text{regsize});$
C6.2.46    CBNZ

Compare and Branch on Nonzero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

![Instruction Format](image)

**32-bit variant**
Applies when \( sf = 0 \).

\[
\text{CBNZ } <Wt>, <label>
\]

**64-bit variant**
Applies when \( sf = 1 \).

\[
\text{CBNZ } <Xt>, <label>
\]

**Decode for all variants of this encoding**

integer \( t = \text{UInt}(Rt) \);
integer \( \text{datasize} = \text{if } sf = '1' \text{ then } 64 \text{ else } 32 \);

bits(64) \( \text{offset} = \text{SignExtend}(\text{imm19}:'00', 64) \);

**Assembler symbols**

\( <Wt> \)  
Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.

\( <Xt> \)  
Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.

\( <\text{label}> \)  
Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

**Operation**

\[
\text{bits(\text{datasize}) operand1 = } X[t, \text{datasize}];
\text{if } \text{IsZero(operand1)} = \text{FALSE then}
\text{BranchTo(PC[] + \text{offset, } BranchType_DIR, TRUE)};
\]
C6.2.47   CBZ

Compare and Branch on Zero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

```
| 31 30 29 28|27 26 25 24|23 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| sf | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | imm19 | | 5 | 4 | 0 |   |   |
|     | op |
```

**32-bit variant**

Applies when $sf == 0$.

CBZ <Wt>, <label>

**64-bit variant**

Applies when $sf == 1$.

CBZ <Xt>, <label>

**Decode for all variants of this encoding**

```plaintext
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
bite(64) offset = SignExtend(imm19:'00', 64);
```

**Assembler symbols**

- `<Wt>`: Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- `<Xt>`: Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- `<label>`: Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

**Operation**

```plaintext
bits(datasize) operand1 = X[t, datasize];
if IsZero(operand1) == TRUE then
    BranchTo(PC[] + offset, BranchType_DIR, TRUE);
```
C6.2.48  CCMN (immediate)

Conditional Compare Negative (immediate) sets the value of the condition flags to the result of the comparison of a register value and a negated immediate value if the condition is TRUE, and an immediate value otherwise.

32-bit variant
Applies when \( sf = 0 \).
\[
\text{CCMN} \ <Wn>, \ #\text{imm}, \ #\text{nzcv}, \ <\text{cond}>
\]

64-bit variant
Applies when \( sf = 1 \).
\[
\text{CCMN} \ <Xn>, \ #\text{imm}, \ #\text{nzcv}, \ <\text{cond}>
\]

Decode for all variants of this encoding
\[
\text{integer } n = \text{UInt}(Rn);
\text{integer } \text{datasize} = \text{if } sf = '1' \text{ then } 64 \text{ else } 32;
\text{bits}(4) \text{ flags} = \text{nzcv};
\text{bits}(\text{datasize}) \text{ imm} = \text{ZeroExtend}(\text{imm5}, \text{datasize});
\]

Assembler symbols
\(<Wn> \quad \text{Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.}\n\<Xn> \quad \text{Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.}\n\<\text{imm}> \quad \text{Is a five bit unsigned (positive) immediate encoded in the "imm5" field.}\n\<\text{nzcv}> \quad \text{Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.}\n\<\text{cond}> \quad \text{Is one of the standard conditions, encoded in the "cond" field in the standard way.}\n
Operation
\[
\text{if } \text{ConditionHolds}(\text{cond}) \text{ then }
\text{bits}(\text{datasize}) \text{ operand1} = X[n, \text{datasize}];
\text{(-, flags)} = \text{AddWithCarry}(\text{operand1}, \text{imm}, '0');
\text{PSTATE.}<N,Z,C,V> = \text{flags};
\]

Operational information
If PSTATE.DIT is 1:
\[
\text{The execution time of this instruction is independent of:}
\text{— The values of the data supplied in any of its registers.}
\text{— The values of the NZCV flags.}
\]
The response of this instruction to asynchronous exceptions does not vary based on:
— The values of the data supplied in any of its registers.
— The values of the NZCV flags.
C6.2.49  CCMN (register)

Conditional Compare Negative (register) sets the value of the condition flags to the result of the comparison of a register value and the inverse of another register value if the condition is TRUE, and an immediate value otherwise.

32-bit variant
Applies when \( sf = 0 \).

\[
\text{CCMN <Wn>, <Wm>, #<nzcv>, <cond>}
\]

64-bit variant
Applies when \( sf = 1 \).

\[
\text{CCMN <Xn>, <Xm>, #<nzcv>, <cond>}
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{integer } \text{datasize} &= \text{if } sf == \text{'1'} \text{ then } 64 \text{ else } 32; \\
\text{bits(4) flags} &= \text{nzcv};
\end{align*}
\]

Assembler symbols

\[
\begin{align*}
<Wn> & \quad \text{Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.} \\
<Wm> & \quad \text{Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.} \\
<Xn> & \quad \text{Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.} \\
<Xm> & \quad \text{Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.} \\
<nzcv> & \quad \text{Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.} \\
<\text{cond}> & \quad \text{Is one of the standard conditions, encoded in the "cond" field in the standard way.}
\end{align*}
\]

Operation

\[
\begin{align*}
\text{if } \text{ConditionHolds}(\text{cond}) \text{ then} \\
\text{bits(\text{datasize}) operand1} &= X[n, \text{datasize}]; \\
\text{bits(\text{datasize}) operand2} &= X[m, \text{datasize}]; \\
(-, \text{flags}) &= \text{AddWithCarry}(\text{operand1, operand2, '0'}); \\
PSTATE.<N,Z,C,V> &= \text{flags};
\end{align*}
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
— The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.50   CCMP (immediate)

Conditional Compare (immediate) sets the value of the condition flags to the result of the comparison of a register value and an immediate value if the condition is TRUE, and an immediate value otherwise.

32-bit variant
Applies when \( sf = 0 \).

\[
\text{COMP } <Wn>, \ #<imm>, \ #<nzcv>, \ <cond>
\]

64-bit variant
Applies when \( sf = 1 \).

\[
\text{COMP } <Xn>, \ #<imm>, \ #<nzcv>, \ <cond>
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } n & = \text{ UInt}(Rn); \\
\text{integer } \text{datasize} & = \text{ if } sf = '1' \text{ then 64 else 32}; \\
\text{bits(4) flags} & = \text{ nzcv}; \\
\text{bits(\text{datasize}) } \text{imm} & = \text{ ZeroExtend}(\text{imm5}, \text{datasize});
\end{align*}
\]

Assembler symbols

\(<Wn>\)       Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
\(<Xn>\)       Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
\(<imm>\)       Is a five bit unsigned (positive) immediate encoded in the "imm5" field.
\(<nzcv>\)    Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
\(<\text{cond}>\)       Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

\[
\begin{align*}
\text{if ConditionHolds}(\text{cond}) \text{ then} \\
\text{bits(\text{datasize}) operand1} & = X[n, \text{datasize}]; \\
\text{bits(\text{datasize}) operand2} & = \text{ NOT}(\text{imm}); \\
(-, \text{flags}) & = \text{ AddWithCarry}(\text{operand1}, \text{operand2}, '1'); \\
PSTATE.<N,Z,C,V> & = \text{flags};
\end{align*}
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.51  CCMP (register)

Conditional Compare (register) sets the value of the condition flags to the result of the comparison of two registers if the condition is TRUE, and an immediate value otherwise.

32-bit variant

Applies when \( sf = 0 \).

\[
\text{COMP } <Wn>, <Wm>, #<nzcv>, <\text{cond}>
\]

64-bit variant

Applies when \( sf = 1 \).

\[
\text{COMP } <Xn>, <Xm>, #<nzcv>, <\text{cond}>
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{integer } \text{datasize} &= \text{if } sf == '1' \text{ then } 64 \text{ else } 32; \\
\text{bits}(4) \text{ flags} &= \text{nzcv};
\end{align*}
\]

Assembler symbols

- \(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Wm>\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Xm>\) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<\text{nzcv}>\) Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
- \(<\text{cond}>\) Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

\[
\begin{align*}
\text{if } \text{ConditionHolds}(\text{cond}) \text{ then} \\
\text{bits(digitsize) operand1} &= X[n, \text{datasize}]; \\
\text{bits(digitsize) operand2} &= X[m, \text{datasize}]; \\
\text{operand2} &= \text{NOT}(\text{operand2}); \\
(-, \text{flags}) &= \text{AddWithCarry}(\text{operand1}, \text{operand2}, '1'); \\
\text{PSTATE.<N,Z,C,V>} &= \text{flags};
\end{align*}
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
— The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.52  CFINV

Invert Carry Flag. This instruction inverts the value of the PSTATE.C flag.

System

(FEAT_FlagM)

Encoding

CFINV

Decode for this encoding

if !HaveFlagManipulateExt() then UNDEFINED;

Operation

PSTATE.C = NOT(PSTATE.C);

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.53  CFP

Control Flow Prediction Restriction by Context prevents control flow predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Control flow predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot be used to exploitatively control speculative execution occurring after the instruction is complete and synchronized.

For more information, see CFP RCTX.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

**System**

(FEAT_SPECRES)

```
[31 30 29 28][27 26 25 24][23 22 21 20][19 18 16][15 12][11 8][7][5][4][0] 0
1 1 1 0 1 0 1 0 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 1 1 1 0 0 1 0 0 1 0 0 Rt
```

**Encoding**

CFP RCTX, <Xt>

is equivalent to

SYS #3, C7, C3, #4, <Xt>

and is always the preferred disassembly.

**Assembler symbols**

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

**Operation**

The description of SYS gives the operational pseudocode for this instruction.
C6.2.54   CINC

Conditional Increment returns, in the destination register, the value of the source register incremented by 1 if the condition is TRUE, and otherwise returns the value of the source register.

This instruction is an alias of the CSINC instruction. This means that:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when \( sf = 0 \).

\[
\text{CINC} \ <Wd>, \ <Wn>, \ <\text{cond}>
\]

is equivalent to

\[
\text{CSINC} \ <Wd>, \ <Wn>, \ <Wn>, \ \text{invert}(<\text{cond})
\]

and is the preferred disassembly when \( Rn = Rm \).

### 64-bit variant

Applies when \( sf = 1 \).

\[
\text{CINC} \ <Xd>, \ <Xn>, \ <\text{cond}>
\]

is equivalent to

\[
\text{CSINC} \ <Xd>, \ <Xn>, \ <Xn>, \ \text{invert}(<\text{cond})
\]

and is the preferred disassembly when \( Rn = Rm \).

### Assembler symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;Wd&gt;)</td>
<td>Is the 32-bit name of the general-purpose destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>(&lt;Wn&gt;)</td>
<td>Is the 32-bit name of the general-purpose source register, encoded in the &quot;Rn&quot; and &quot;Rm&quot; fields.</td>
</tr>
<tr>
<td>(&lt;Xd&gt;)</td>
<td>Is the 64-bit name of the general-purpose destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>(&lt;Xn&gt;)</td>
<td>Is the 64-bit name of the general-purpose source register, encoded in the &quot;Rn&quot; and &quot;Rm&quot; fields.</td>
</tr>
<tr>
<td>(&lt;\text{cond}&gt;)</td>
<td>Is one of the standard conditions, excluding AL and NV, encoded in the &quot;cond&quot; field with its least significant bit inverted.</td>
</tr>
</tbody>
</table>

### Operation

The description of CSINC gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.55  CINV

Conditional Invert returns, in the destination register, the bitwise inversion of the value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This instruction is an alias of the CSINV instruction. This means that:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode for this instruction.

32-bit variant

Applies when $sf = 0$.

CINV <Wd>, <Wn>, <cond>

is equivalent to

CSINV <Wd>, <Wn>, <Wn>, invert(<cond>)

and is the preferred disassembly when $Rn = Rm$.

64-bit variant

Applies when $sf = 1$.

CINV <Xd>, <Xn>, <cond>

is equivalent to

CSINV <Xd>, <Xn>, <Xn>, invert(<cond>)

and is the preferred disassembly when $Rn = Rm$.

Assembler symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <cond> Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSINV gives the operational pseudocode for this instruction.
**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.56  CLREX

Clear Exclusive clears the local monitor of the executing PE.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 8 7 6 5 4 3 2 1 0]
1 1 0 1 0 1 0 0 0 0 0 1 1 0 1 1 1 1
```

**Encoding**

CLREX {#<imm>}

**Decode for this encoding**

// CRm field is ignored

**Assembler symbols**

<imm>  Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

**Operation**

```
ClearExclusiveLocal(ProcessorID());
```
C6.2.57  CLS

Count Leading Sign bits counts the number of leading bits of the source register that have the same value as the most significant bit of the register, and writes the result to the destination register. This count does not include the most significant bit of the source register.

32-bit variant
Applies when sf == 0.
CLS <Wd>, <Wn>

64-bit variant
Applies when sf == 1.
CLS <Xd>, <Xn>

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

integer result;
bits(datasize) operand1 = X[n, datasize];
result = CountLeadingSignBits(operand1);
X[d, datasize] = result<datasize-1:0>;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
— The values of the NZCV flags.
C6.2.58 CLZ

Count Leading Zeros counts the number of binary zero bits before the first binary one bit in the value of the source register, and writes the result to the destination register.

32-bit variant

Applies when \( sf = 0 \).

\[
\text{CLZ} \ <Wd>, \ <Wn> 
\]

64-bit variant

Applies when \( sf = 1 \).

\[
\text{CLZ} \ <Xd>, \ <Xn> 
\]

Decoding for all variants of this encoding

\[
\text{integer } d = \text{ UInt}(Rd); \\
\text{integer } n = \text{ UInt}(Rn); \\
\text{integer } \text{datasize} = \text{if } sf = '1' \text{ then } 64 \text{ else } 32; 
\]

Assembler symbols

\(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Wn>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

\(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Xn>\) Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

\[
\text{integer } \text{result}; \\
\text{bits}(\text{datasize}) \text{operand1} = X[n, \text{datasize}]; \\
\text{result} = \text{CountLeadingZeroBits}(\text{operand1}); \\
X[d, \text{datasize}] = \text{result}<\text{datasize}-1:0>; 
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.59   CMN (extended register)

Compare Negative (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the ADDS (extended register) instruction. This means that:

- The encodings in this description are named to match the encodings of ADDS (extended register).
- The description of ADDS (extended register) gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when sf == 0.

CMN <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

is equivalent to

ADDS WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

and is always the preferred disassembly.

### 64-bit variant

Applies when sf == 1.

CMN <Xn|SP>, <R><m>{, <extend> {#<amount>}}

is equivalent to

ADDS XZR, <Xn|SP>, <R><m>{, <extend> {#<amount>}}

and is always the preferred disassembly.

### Assembler symbols

- `<Wn|WSP>` is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Wm>` is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xn|SP>` is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<R>` is a width specifier, encoded in the "option" field. It can have the following values:
  - `W` when option = 00x
  - `W` when option = 010
  - `X` when option = x11
  - `W` when option = 10x
  - `W` when option = 110
- `<m>` is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

- UXTB when option = 000
- UXTH when option = 001
- LSL|UXTW when option = 010
- UTX when option = 011
- SXTB when option = 100
- SXTH when option = 101
- SXTW when option = 110
- SXTX when option = 111

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

- UXTB when option = 000
- UXTH when option = 001
- UXTW when option = 010
- LSL|UXTX when option = 011
- SXTB when option = 100
- SXTH when option = 101
- SXTW when option = 110
- SXTX when option = 111

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

**Operation**

The description of **ADDS (extended register)** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.60  CMN (immediate)

Compare Negative (immediate) adds a register value and an optionally-shifted immediate value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the ADDS (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ADDS (immediate).
- The description of ADDS (immediate) gives the operational pseudocode for this instruction.

32-bit variant

Applies when \( sf = 0 \).

CMN \(<Wn|WSP>, \#<imm>{, <shift}>\)

is equivalent to

ADDS WZR, <Wn|WSP>, \#<imm> {, <shift>}

and is always the preferred disassembly.

64-bit variant

Applies when \( sf = 1 \).

CMN \(<Xn|SP>, \#<imm>{, <shift}>\)

is equivalent to

ADDS XZR, <Xn|SP>, \#<imm> {, <shift>}

and is always the preferred disassembly.

Assembler symbols

- \(<Wn|WSP>\) Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- \(<Xn|SP>\) Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- \(<imm>\) Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- \(<shift>\) Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:
  - LSL #0 when \( sh = 0 \)
  - LSL #12 when \( sh = 1 \)

Operation

The description of ADDS (immediate) gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.61   CMN (shifted register)

Compare Negative (shifted register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the ADDS (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of ADDS (shifted register).
- The description of ADDS (shifted register) gives the operational pseudocode for this instruction.

**32-bit variant**

Applies when $s_f == 0$.

CMN $<Wn>$, $<Wm>$

is equivalent to

ADDS WZR, $<Wn>$, $<Wm>$

and is always the preferred disassembly.

**64-bit variant**

Applies when $s_f == 1$.

CMN $<Xn>$, $<Xm>$

is equivalent to

ADDS XZR, $<Xn>$, $<Xm>$

and is always the preferred disassembly.

**Assembler symbols**

- $<Wn>$ Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- $<Wm>$ Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- $<Xn>$ Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- $<Xm>$ Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- $<shift>$ Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:
  - LSL when $shift = 00$
  - LSR when $shift = 01$
  - ASR when $shift = 10$
  - The encoding $shift = 11$ is reserved.
- $<amount>$ For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  - For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.
Operation

The description of ADDS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.62    CMP (extended register)

Compare (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the SUBS (extended register) instruction. This means that:

- The encodings in this description are named to match the encodings of SUBS (extended register).
- The description of SUBS (extended register) gives the operational pseudocode for this instruction.

32-bit variant

Applies when \( sf == 0 \).

\[
\text{CMP} \ <Wn|WSP>, <Wm>{, <extend> {#<amount>}} \\
\text{is equivalent to} \\
\text{SUBS} \ WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}} \\
\text{and is always the preferred disassembly.}
\]

64-bit variant

Applies when \( sf == 1 \).

\[
\text{CMP} \ <Xn|SP>, <R><m>{, <extend> {#<amount>}} \\
\text{is equivalent to} \\
\text{SUBS} \ XZR, <Xn|SP>, <R><m>{, <extend> {#<amount>}} \\
\text{and is always the preferred disassembly.}
\]

Assembler symbols

- `<Wn|WSP>` Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<R>` Is a width specifier, encoded in the "option" field. It can have the following values:
  - `W` when option = 00x
  - `W` when option = 010
  - `X` when option = x11
  - `W` when option = 10x
  - `W` when option = 110
- `<m>` Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

---

C6-1276  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  ARM DDI 0487I.a  Non-Confidential  ID081822
For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

- **UXTB** when option = 000
- **UXTH** when option = 001
- **LSL|UXTW** when option = 010
- **UXTX** when option = 011
- **SXTB** when option = 100
- **SXTH** when option = 101
- **SXTW** when option = 110
- **SXTX** when option = 111

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

- **UXTB** when option = 000
- **UXTH** when option = 001
- **UXTW** when option = 010
- **LSL|UXTX** when option = 011
- **SXTB** when option = 100
- **SXTH** when option = 101
- **SXTW** when option = 110
- **SXTX** when option = 111

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

**Operation**

The description of **SUBS (extended register)** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
### C6.2.63  **CMP (immediate)**

Compare (immediate) subtracts an optionally-shifted immediate value from a register value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the **SUBS (immediate)** instruction. This means that:

- The encodings in this description are named to match the encodings of **SUBS (immediate)**.
- The description of **SUBS (immediate)** gives the operational pseudocode for this instruction.

#### 32-bit variant

Applies when $sf == 0$.

CMP $<Wn|WSP>$, #<imm>{, <shift>}

is equivalent to

SUBS WZR, $<Wn|WSP>$, #<imm> {, <shift>}

and is always the preferred disassembly.

#### 64-bit variant

Applies when $sf == 1$.

CMP $<Xn|SP>$, #<imm>{, <shift>}

is equivalent to

SUBS XZR, $<Xn|SP>$, #<imm> {, <shift>}

and is always the preferred disassembly.

#### Assembler symbols

- $<Wn|WSP>$: Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- $<Xn|SP>$: Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- $<imm>$: Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- $<shift>$: Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:
  
  - LSL #0 when $sh = 0$
  - LSL #12 when $sh = 1$

#### Operation

The description of **SUBS (immediate)** gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.64  **CMP (shifted register)**

Compare (shifted register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the **SUBS (shifted register)** instruction. This means that:

- The encodings in this description are named to match the encodings of **SUBS (shifted register)**.
- The description of **SUBS (shifted register)** gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when \( sf = 0 \).

CMP \(<Wn>, <Wm>{, <shift> #<amount>}\)

is equivalent to

SUBS WZR, <Wn>, <Wm> {, <shift> #<amount>}

and is always the preferred disassembly.

### 64-bit variant

Applies when \( sf = 1 \).

CMP \(<Xn>, <Xm>{, <shift> #<amount>}\)

is equivalent to

SUBS XZR, <Xn>, <Xm> {, <shift> #<amount>}

and is always the preferred disassembly.

### Assembler symbols

- \(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Wm>\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Xm>\) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<shift>\) Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:
  - LSL when \( shift = 00 \)
  - LSR when \( shift = 01 \)
  - ASR when \( shift = 10 \)
  - The encoding \( shift = 11 \) is reserved.
- \(<amount>\) For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.
Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.65   CMPP

Compare with Tag subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, updates the condition flags based on the result of the subtraction, and discards the result.

This instruction is an alias of the SUBPS instruction. This means that:

- The encodings in this description are named to match the encodings of SUBPS.
- The description of SUBPS gives the operational pseudocode for this instruction.

**Integer**

(FEAT_MTE)

| 31 30 29 28 27 26 25 24 23 22 21 20 | 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | Xn 0 0 0 0 Xn 1 1 1 1 Xd |

**Encoding**

CMPP <Xn|SP>, <Xm|SP>

is equivalent to

SUBPS XZR, <Xn|SP>, <Xm|SP>

and is always the preferred disassembly.

**Assembler symbols**

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.

<Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

**Operation**

The description of SUBPS gives the operational pseudocode for this instruction.
C6.2.66 CNEG

Conditional Negate returns, in the destination register, the negated value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This instruction is an alias of the CSNEG instruction. This means that:

- The encodings in this description are named to match the encodings of CSNEG.
- The description of CSNEG gives the operational pseudocode for this instruction.

32-bit variant

Applies when sf == 0.

CNEG <Wd>, <Wn>, <cond>

is equivalent to

CSNEG <Wd>, <Wn>, <Wn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

64-bit variant

Applies when sf == 1.

CNEG <Xd>, <Xn>, <cond>

is equivalent to

CSNEG <Xd>, <Xn>, <Xn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

Assembler symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <cond> Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSNEG gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.67 CPP

Cache Prefetch Prediction Restriction by Context prevents cache allocation predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. The actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control cache prefetch predictions occurring after the instruction is complete and synchronized.

For more information, see CPP RCTX.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

System

(FEAT_SPECRES)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 8 7 5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 0</td>
<td>0 0 1 0 1 1 0 1 1 0 0 1 1 1 1</td>
</tr>
<tr>
<td>L op1 CRn CRm op2</td>
<td></td>
</tr>
</tbody>
</table>

Encoding

CPP RCTX, <Xt>

is equivalent to

SYS #3, C7, C3, #7, <Xt>

and is always the preferred disassembly.

Assembler symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.
C6.2.68  CPYFP, CPYFM, CPYFE

Memory Copy Forward-only. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFP, then CPYFM, and then CPYFE.

CPYFP performs some preconditioning of the arguments suitable for using the CPYFM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFM performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFE performs the last part of the memory copy.

___ Note ___

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

___ Note ___

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

___ Note ___

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFP, option A (which results in encoding PSTATE.C = 0):

• If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
• Xs holds the original Xs + saturated Xn.
• Xd holds the original Xd + saturated Xn.
• Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
• PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFP, option B (which results in encoding PSTATE.C = 1):

• If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
• Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
• Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
• Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
• PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFM, option A (encoded by PSTATE.C = 0), the format of the arguments is:

• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from -Xn.
• Xd holds the lowest address that the copy is made to -Xn.
• At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFM, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

For CPYFE, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number and holds \(-1\) the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from \(-Xn\).
• Xd holds the lowest address that the copy is made to \(-Xn\).
• At the end of the instruction, the value of Xn is written back with 0.

For CPYFE, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

**Integer**

(FEAT_MOPS)

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz 0 1 1 0 0 1</td>
<td>op1 0</td>
<td>Rs 0 0 0 0 0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Epilogue variant**

Applies when op1 == 10.

CPYFE \([<Xd>], [<Xs>], <Xn>\)

**Main variant**

Applies when op1 == 01.

CPYFM \([<Xd>], [<Xs>], <Xn>\)

**Prologue variant**

Applies when op1 == 00.

CPYFP \([<Xd>], [<Xs>], <Xn>\)
Decode for all variants of this encoding

```c
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler symbols

**<Xd>**
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

**<Xs>**
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

**<Xn>**
For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```c
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFF<63:0>;
```

if supports_option_a then
  nzcv = '0000';
  // Copy in the forward direction offsets the arguments.
  toaddress = toaddress + cpysize;
  fromaddress = fromaddress + cpysize;
  cpysize = Zeros(64) - cpysize;
else
  nzcv = '0010';
// IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);
if SInt(cpysize) > 0 then
  assert SInt(stagecpysize) <= SInt(cpysize);
else
  assert SInt(stagecpysize) >= SInt(cpysize);
else
  boolean zero_size_exceptions = MemCpyZeroSizeCheck();
// Check if this version is consistent with the state of the call.
if zero_size_exceptions || SInt(cpysize) != 0 then
  if supports_option_a then
    if nzcv<1> == '1' then // PSTATE.C
      boolean wrong_option = TRUE;
      boolean from_epilogue = stage == MOPSStage_Epilogue;
      MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
      if nzcv<1> == '0' then // PSTATE.C
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
if stage == MOPSStage_Main then
  stagecpysize = cpysize - postsize;
  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;
    // Check if the parameters to this instruction are valid for the epilogue.
    if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
      boolean wrong_option = FALSE;
      boolean from_epilogue = TRUE;
      MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  if supports_option_a then
    while SInt(stagecpysize) != 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = CPYSizeChoice(toaddress, fromaddress, cpysize);
      assert B <= -1 * SInt(stagecpysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
    Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
  cpysize = cpysize + B;
  stagecpysize = stagecpysize + B;
if stage != MOPSStage_Prologue then
  X[n, 64] = cpysize;
else
  while UInt(stagecpysize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSIZEChoice(toaddress, fromaddress, cpysize);
    assert B <= UInt(stagecpysize);

    readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
    Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
    cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;

if stage != MOPSStage_Prologue then
  X[n, 64] = cpysize;
  X[d, 64] = toaddress;
  X[s, 64] = fromaddress;
if stage == MOPSStage_Prologue then
  X[n, 64] = cpysize;
  X[d, 64] = toaddress;
  X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;
C6.2.69 CPYFPN, CPYFMN, CPYFEN

Memory Copy Forward-only, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPN, then CPYFMN, and then CPYFEN.

CPYFPN performs some preconditioning of the arguments suitable for using the CPYFMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEN performs the last part of the memory copy.

--- Note ---

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

---

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

--- Note ---

Portable software should not assume that the choice of algorithm is constant.

---

After execution of CPYFPN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.

• At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.

• Xs holds the lowest address that the copy is copied from -Xn.

• Xd holds the lowest address that the copy is made to -Xn.

• At the end of the instruction, the value of Xn is written back with 0.

For CPYFEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes remaining to be copied in the memory copy in total.

• Xs holds the lowest address that the copy is copied from.

• Xd holds the lowest address that the copy is copied to.

• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

**Integer**

(FEAT_MOPS)

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>op1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Epilogue variant**

Applies when op1 == 10.

CPYFEN [Xd]!, [Xs]!, <Xn>!

**Main variant**

Applies when op1 == 01.

CPYFMN [Xd]!, [Xs]!, <Xn>!

**Prologue variant**

Applies when op1 == 00.

CPYFPN [Xd]!, [Xs]!, <Xn>!
Decode for all variants of this encoding

```c
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```c
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFF<63:0>;
```
if supports_option_a then
    nzc = '0000';
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
else
    nzc = '0010';
    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
    else
        boolean zero_size_exceptions = MemCpyZeroSizeCheck();
        // Check if this version is consistent with the state of the call.
        if zero_size_exceptions || SInt(cpysize) != 0 then
            if supports_option_a then
                if nzc<1> == '1' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                else
                    if nzc<1> == '0' then // PSTATE.C
                        boolean wrong_option = TRUE;
                        boolean from_epilogue = stage == MOPSStage_Epilogue;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
            assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
            if stage == MOPSStage_Main then
                stagecpysize = cpysize - postsize;
                // Check if the parameters to this instruction are valid.
                if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                    boolean wrong_option = FALSE;
                    boolean from_epilogue = FALSE;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                else
                    stagecpysize = postsize;
                    // Check if the parameters to this instruction are valid for the epilogue.
                    if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                        boolean wrong_option = FALSE;
                        boolean from_epilogue = TRUE;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            if supports_option_a then
                while SInt(stagecpysize) != 0 do
                    // IMP DEF selection of the block size that is worked on. While many
                    // implementations might make this constant, that is not assumed.
                    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                    assert B <= -1 * SInt(stagecpysize);
                    readda8&lt;8-1:0&gt; = Mem[fromaddress+cpysize, B, racctype];
                    Mem[toaddress+cpysize, B, wacctype] = readda8&lt;8-1:0&gt;;
                    cpysize = cpysize + B;
                    stagecpysize = stagecpysize + B;
if stage != MOPSStage_Prologue then
  X[n, 64] = cpysize;
else
  while UInt(stagecpysize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
    assert B <= UInt(stagecpysize);

    readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
    Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
    toaddress = toaddress + B;
    fromaddress = fromaddress + B;
    cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;

if stage != MOPSStage_Prologue then
  X[n, 64] = cpysize;
  X[d, 64] = toaddress;
  X[s, 64] = fromaddress;
else
  X[n, 64] = cpysize;
  X[d, 64] = toaddress;
  X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;
C6.2.70 CPYFPRN, CPYFMRN, CPYFERN

Memory Copy Forward-only, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRN, then CPYFMRN, and then CPYFERN.

CPYFPRN performs some preconditioning of the arguments suitable for using the CPYFMRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERN performs the last part of the memory copy.

--- Note ---
The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

--- Note ---
The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

--- Note ---
Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRN, option A (which results in encoding PSTATE.C = 0):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRN, option B (which results in encoding PSTATE.C = 1):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from -Xn.
• Xd holds the lowest address that the copy is made to -Xn.
• At the end of the instruction, the value of Xn is written back with 0.

For CPYFERN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

**Integer**

(FEAT_MOPS)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz 0 1 1 0 0 1</td>
<td>op1 0</td>
<td>Rs 1 0 0 0 0 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Epilogue variant**

Applies when op1 = 10.

CPYFERN [Xd]!, [Xs]!, <Xn>!

**Main variant**

Applies when op1 = 01.

CPYFRM [Xd]!, [Xs]!, <Xn>!

**Prologue variant**

Applies when op1 = 00.

CPYFPFR [Xd]!, [Xs]!, <Xn>!
Decode for all variants of this encoding

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);
boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);
if stage == MOPSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;
if supports_option_a then
    nzciv = '0000';
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
else
    nzciv = '0010';
    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPretSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);
if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
else
    assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzciv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue,
                options);
            else
                if nzciv<1> == '0' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue,
                    options);
        bits(64) postsize = CPYPretSizeChoice(toaddress, fromaddress, cpysize);
        assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
        if stage == MOPSStage_Main then
            stagecpysize = cpysize - postsize;
            // Check if the parameters to this instruction are valid.
            if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = FALSE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue,
                options);
            else
                stagecpysize = postsize;
                // Check if the parameters to this instruction are valid for the epilogue.
                if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                    boolean wrong_option = FALSE;
                    boolean from_epilogue = TRUE;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue,
                    options);
            if supports_option_a then
                while SInt(stagecpysize) != 0 do
                    // IMP DEF selection of the block size that is worked on. While many
                    // implementations might make this constant, that is not assumed.
                    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                    assert B <= -1 * SInt(stagecpysize);
                    readdata<8*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
                    Mem[toaddress+cpysize, B, wacctype] = readdata<8*8-1:0>;
                    cpysize = cpysize + B;
                    stagecpysize = stagecpysize + B;
if stage != MOPSStage_Prologue then
  X[n, 64] = cpysize;
else
  while UInt(stagecpysize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
    assert B <= UInt(stagecpysize);
    readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
    Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
    toaddress = toaddress + B;
    fromaddress = fromaddress + B;
    cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;
  if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
  if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
C6.2.71   CPYFPRT, CPYFMRT, CPYFERT

Memory Copy Forward-only, reads unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRT, then CPYFMRT, and then CPYFERT.

CPYFPRT performs some preconditioning of the arguments suitable for using the CPYFMRT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERT performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRT, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> === 1, the copy size is saturated to 0x7FFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRT, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> === 1, the copy size is saturated to 0x7FFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.

• At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERT, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from -Xn.
• Xd holds the lowest address that the copy is made to -Xn.
• At the end of the instruction, the value of Xn is written back with 0.

For CPYFERT, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

**Integer**

(FEAT_MOPS)

<table>
<thead>
<tr>
<th>[31] [30] [29] [28] [27] [26] [25] [24] [23] [22] [21] [20]</th>
<th>[16] [15]</th>
<th>[12] [11] [10] [9]</th>
<th>[5] [4]</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Epilogue variant**

Applies when op1 == 10.
CPYFERT [Xd!], [Xs!], <Xn>!

**Main variant**

Applies when op1 == 01.
CPYFMT [Xd!], [Xs!], <Xn>!

**Prologue variant**

Applies when op1 == 00.
CPYFPRT [Xd!], [Xs!], <Xn>!
Decode for all variants of this encoding

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(64*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFF<63:0>;
if supports_option_a then
  nzcv = '0000';
  // Copy in the forward direction offsets the arguments.
  toaddress = toaddress + cpysize;
  fromaddress = fromaddress + cpysize;
  cpysize = Zeros(64) - cpysize;
else
  nzcv = '0010';

  // IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);
if SInt(cpysize) > 0 then
  assert SInt(stagecpysize) <= SInt(cpysize);
else
  assert SInt(stagecpysize) >= SInt(cpysize);
else
  boolean zero_size_exceptions = MemCpyZeroSizeCheck();
  // Check if this version is consistent with the state of the call.
  if zero_size_exceptions || SInt(cpysize) != 0 then
    if supports_option_a then
      if nzcv<1> == '1' then // PSTATE.C
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        if nzcv<1> == '0' then // PSTATE.C
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        
      bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
      
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
      stagecpysize = cpysize - postsize;
      
      // Check if the parameters to this instruction are valid.
      if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = FALSE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        stagecpysize = postsize;

      // Check if the parameters to this instruction are valid for the epilogue.
      if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = TRUE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        if supports_option_a then
          while SInt(stagecpysize) != 0 do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(toaddress, fromaddress, cpysize);
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
if stage != MOPSStage_Prologue then
    X[n, 64] = cpsize;
else
    while UInt(stagecpsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpsize);
        assert B <= UInt(stagecpsize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        toaddress = toaddress + B;
        fromaddress = fromaddress + B;
        cpsize = cpsize - B;
        stagecpsize = stagecpsize - B;

    if stage != MOPSStage_Prologue then
        X[n, 64] = cpsize;
        X[d, 64] = toaddress;
        X[s, 64] = fromaddress;
    if stage == MOPSStage_Prologue then
        X[n, 64] = cpsize;
        X[d, 64] = toaddress;
        X[s, 64] = fromaddress;
        PSTATE.<N,Z,C,V> = nzcv;
C6.2.72  CPYFPRTN, CPYFMRTN, CPYFERTN

Memory Copy Forward-only, reads unprivileged, reads and writes non-temporal. These instructions perform a
memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear
consecutively in memory: CPYFPRTN, then CPYFMRTN, and then CPYFERTN.

CPYFPRTN performs some preconditioning of the arguments suitable for using the CPYFMRTN instruction, and
performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRTN performs an IMPLEMENTATION
DEFINED amount of the memory copy. CPYFERTN performs the last part of the memory copy.

------ Note ------
The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can
be performed.

------ Note ------
The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable
for a memory copy only where there is no overlap between the source and destination locations, or where the source
address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is
IMPLEMENTATION DEFINED.

------ Note ------
Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the
memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be
copied in the memory copy in total.

For CPYFMRTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.

• At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from -Xn.
• Xd holds the lowest address that the copy is made to -Xn.
• At the end of the instruction, the value of Xn is written back with 0.

For CPYFERTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

**Integer**

(FEAT_MOPS)

| [31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 | 12|11 10 9 | 5 4 | 0 |
|---|---|---|---|---|---|---|
| sz | 0 | 1 | 1 | 0 | 0 | 1 | op1 | 0 | Rs | 1 | 1 | 1 | 0 | 0 | 1 | Rn | Rd |
| op2 |

**Epilogue variant**

Applies when op1 == 10.

CPYFERTN [Xd]!, [Xs]!, <Xn>!

**Main variant**

Applies when op1 == 01.

CPYFMRTN [Xd]!, [Xs]!, <Xn>!

**Prologue variant**

Applies when op1 == 00.

CPYFPRTN [Xd]!, [Xs]!, <Xn>!
Decode for all variants of this encoding

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;
if supports_option_a then
    nzcv = '0000';
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
else
    nzcv = '0010';
    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);
if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
else
    assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if nzcv<1> == '0' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        bits(64) postsize = CPYPPostSizeChoice(toaddress, fromaddress, cpysize);
        assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
        if stage == MOPSStage_Main then
            stagecpysize = cpysize - postsize;
            // Check if the parameters to this instruction are valid.
            if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = FALSE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                stagecpysize = postsize;
                // Check if the parameters to this instruction are valid for the epilogue.
                if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                    boolean wrong_option = FALSE;
                    boolean from_epilogue = TRUE;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            boolean zero_size_exceptions = MemCpyZeroSizeCheck();
            // Check if this version is consistent with the state of the call.
            if zero_size_exceptions || SInt(cpysize) != 0 then
                if supports_option_a then
                    if nzcv<1> == '1' then // PSTATE.C
                        boolean wrong_option = TRUE;
                        boolean from_epilogue = stage == MOPSStage_Epilogue;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                    else
                        if nzcv<1> == '0' then // PSTATE.C
                            boolean wrong_option = TRUE;
                            boolean from_epilogue = stage == MOPSStage_Epilogue;
                            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        bits(64) postsize = CPYPPostSizeChoice(toaddress, fromaddress, cpysize);
        assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
        if stage == MOPSStage_Main then
            stagecpysize = cpysize - postsize;
            // Check if the parameters to this instruction are valid.
            if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = FALSE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                stagecpysize = postsize;
                // Check if the parameters to this instruction are valid for the epilogue.
                if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                    boolean wrong_option = FALSE;
                    boolean from_epilogue = TRUE;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            if supports_option_a then
                while SInt(stagecpysize) != 0 do
                    // IMP DEF selection of the block size that is worked on. While many
                    // implementations might make this constant, that is not assumed.
                    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                    assert B <= -1 || SInt(stagecpysize);
                    readdata<8*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
                    Mem[toaddress+cpysize, B, wacctype] = readdata<8*8-1:0>;
                    cpysize = cpysize + B;
                    stagecpysize = stagecpysize + B;
if stage != MOPSStage_Prologue then
  X[n, 64] = cpysize;
else
  while Uint(stagecpysize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
    assert B <= Uint(stagecpysize);

    readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
    Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
    toaddress = toaddress + B;
    fromaddress = fromaddress + B;
    cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;

  if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
  if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;
**C6.2.73 CPYFPRTRN, CPYFMRTRN, CPYFERTRN**

Memory Copy Forward-only, reads unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTRN, then CPYFMRTRN, and then CPYFERTRN.

CPYFPRTRN performs some preconditioning of the arguments suitable for using the CPYFMRTRN instruction, and performs an *IMPLEMENTATION DEFINED* amount of the memory copy. CPYFMRTRN performs an *IMPLEMENTATION DEFINED* amount of the memory copy. CPYFERTRN performs the last part of the memory copy.

--- **Note**

The inclusion of *IMPLEMENTATION DEFINED* amounts of memory copy allows some optimization of the size that can be performed.

---

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is *IMPLEMENTATION DEFINED.*

--- **Note**

Portable software should not assume that the choice of algorithm is constant.

---

After execution of CPYFPRTRN, option A (which results in encoding PSTATE.C = 0):

- If $Xn_{<63>}$ == 1, the copy size is saturated to $0x7FFFFFFFFFFFFFFF$.
- $Xs$ holds the original $Xs +$ saturated $Xn$.
- $Xd$ holds the original $Xd +$ saturated $Xn$.
- $Xn$ holds $-1* $ saturated $Xn +$ an *IMPLEMENTATION DEFINED* number of bytes copied.
- PSTATE.{$N,Z,V}$ are set to {0,0,0}.

After execution of CPYFPRTRN, option B (which results in encoding PSTATE.C = 1):

- If $Xn_{<63>}$ == 1, the copy size is saturated to $0x7FFFFFFFFFFFFFFF$.
- $Xs$ holds the original $Xs +$ an *IMPLEMENTATION DEFINED* number of bytes copied.
- $Xd$ holds the original $Xd +$ an *IMPLEMENTATION DEFINED* number of bytes copied.
- $Xn$ holds the saturated $Xn -$ an *IMPLEMENTATION DEFINED* number of bytes copied.
- PSTATE.{$N,Z,V}$ are set to {0,0,0}.

For CPYFMRTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $Xn$ is treated as a signed 64-bit number and holds $-1* $ the number of bytes remaining to be copied in the memory copy in total.
- $Xs$ holds the lowest address that the copy is copied from -$Xn$.
- $Xd$ holds the lowest address that the copy is made to -$Xn$.
- At the end of the instruction, the value of $Xn$ is written back with $-1* $ the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $Xn$ holds the number of bytes remaining to be copied in the memory copy in total.
- $Xs$ holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.

• At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.

• Xs holds the lowest address that the copy is copied from -Xn.

• Xd holds the lowest address that the copy is made to -Xn.

• At the end of the instruction, the value of Xn is written back with 0.

For CPYFERTRN option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes remaining to be copied in the memory copy in total.

• Xs holds the lowest address that the copy is copied from.

• Xd holds the lowest address that the copy is copied to.

• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

**Integer**

**(FEAT_MOPS)**

\[
\begin{array}{cccccccccc}
& & & & & & & & & & \\
\text{sz} & 0 & 1 & 1 & 0 & 0 & 1 & \text{op1} & 0 & \text{Rs} & 1 & 0 & 1 & 0 & 0 & 1 & \text{Rn} & \text{Rd} & 0 \\
\end{array}
\]

**Epilogue variant**

Applies when op1 == 10.

CPYFERTRN [Xd], [Xs], Xn!

**Main variant**

Applies when op1 == 01.

CPYFMTRN [Xd], [Xs], Xn!

**Prologue variant**

Applies when op1 == 00.

CPYFPTRN [Xd], [Xs], Xn!
**Decode for all variants of this encoding**

```plaintext
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

**Assembler symbols**

- `<Xd>`
  - For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
  - For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

- `<Xs>`
  - For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
  - For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

- `<Xn>`
  - For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
  - For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
  - For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

**Operation**

```plaintext
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;
```
if supports_option_a then
    nzc_v = '0000';
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
else
    nzc_v = '0010';
    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);
if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
else
    assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzc_v<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if nzc_v<1> == '0' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
            assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
if stage == MOPSStage_Main then
    stagecpysize = cpysize - postsize;
    // Check if the parameters to this instruction are valid.
    if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = FALSE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;
        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            if supports_option_a then
                while SInt(stagecpysize) != 0 do
                    // IMP DEF selection of the block size that is worked on. While many
                    // implementations might make this constant, that is not assumed.
                    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                    assert B <= -1 * SInt(stagecpysize);
                    readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
                    Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
                    cpysize = cpysize + B;
                    stagecpysize = stagecpysize + B;
if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSIZEChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        toaddress = toaddress + B;
        fromaddress = fromaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

    if stage != MOPSStage_Prologue then
        X[n, 64] = cpysize;
        X[d, 64] = toaddress;
        X[s, 64] = fromaddress;
    if stage == MOPSStage_Prologue then
        X[n, 64] = cpysize;
        X[d, 64] = toaddress;
        X[s, 64] = fromaddress;
        PSTATE.<N,Z,C,V> = nzcv;
C6.2.74 CPYFPRTWN, CPYFMRTWN, CPYFERTWN

Memory Copy Forward-only, reads unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTWN, then CPYFMRTWN, and then CPYFERTWN.

CPYFPRTWN performs some preconditioning of the arguments suitable for using the CPYFMRTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERTWN performs the last part of the memory copy.

--- Note ---
The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

--- Note ---
Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRTWN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRTWN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from -Xn.
• Xd holds the lowest address that the copy is made to -Xn.
• At the end of the instruction, the value of Xn is written back with 0.

For CPYFERTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)

```
| 31 30 29 28|27 26 25 24|23 22 21 20| 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
| sz | 0 1 1 | 0 0 | op1 | 0 | Rs | 0 1 1 | 0 1 | Rn | Rd |
```

Epilogue variant
Applies when op1 == 10.
CPYFERTWN [Xd]!, [Xs]!, <Xn>!

Main variant
Applies when op1 == 01.
CPYFMRTWN [Xd]!, [Xs]!, <Xn>!

Prologue variant
Applies when op1 == 00.
CPYFPRTWN [Xd]!, [Xs]!, <Xn>!
Decode for all variants of this encoding

```c
if (!HaveFeatMOPS()) then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler symbols

<Xd>  
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs>  
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn>  
For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```c
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;
```
if supports_option_a then
  nzcv = '0000';
  // Copy in the forward direction offsets the arguments.
  toaddress = toaddress + cpysize;
  fromaddress = fromaddress + cpysize;
  cpysize = Zeros(64) - cpysize;
else
  nzcv = '0010';

  // IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

if SInt(cpysize) > 0 then
  assert SInt(stagecpysize) <= SInt(cpysize);
else
  assert SInt(stagecpysize) >= SInt(cpysize);
else
  boolean zero_size_exceptions = MemCpyZeroSizeCheck();

  // Check if this version is consistent with the state of the call.
  if zero_size_exceptions || SInt(cpysize) != 0 then
    if supports_option_a then
      if nzcv<1> == '1' then // PSTATE.C
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        if nzcv<1> == '0' then // PSTATE.C
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSStage_Main then
  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
else
  stagecpysize = postsize;

  // Check if the parameters to this instruction are valid for the epilogue.
  if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
else
  if supports_option_a then
    while SInt(stagecpysize) != 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = CPYSizeChoice(toaddress, fromaddress, cpysize);
      assert B <= -1 * SInt(stagecpysize);

      readdata<8*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<8*8-1:0>;
      cpysize = cpysize + B;
      stagecpysize = stagecpysize + B;
if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);
        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;
        if stage == MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;
        PSTATE.<N,Z,C,V> = nzcv;
C6.2.75 CPYFPT, CPYFMT, CPYFET

Memory Copy Forward-only, reads and writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPT, then CPYFMT, and then CPYFET.

CPYFPT performs some preconditioning of the arguments suitable for using the CPYFMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFET performs the last part of the memory copy.

--- Note ---
The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

--- Note ---
The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

--- Note ---
Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPT, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0xffffffffffffffff.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPT, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0xffffffffffffffff.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

For CPYFET, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from -Xn.
• Xd holds the lowest address that the copy is made to -Xn.
• At the end of the instruction, the value of Xn is written back with 0.

For CPYFET, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

**Integer**

(FEAT_MOPS)

<table>
<thead>
<tr>
<th>sz</th>
<th>0 1 1 0 0 1</th>
<th>op1 0</th>
<th>Rs</th>
<th>0 0 1 1 0 1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Epilogue variant**

Applies when op1 == 10.

CPYFET [Xd]!, [Xs]!, <Xn>!

**Main variant**

Applies when op1 == 01.

CPYFMT [Xd]!, [Xs]!, <Xn>!

**Prologue variant**

Applies when op1 == 00.

CPYFPT [Xd]!, [Xs]!, <Xn>!
Decode for all variants of this encoding

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFF<63:0>;}
if supports_option_a then
    nzcv = '0000';
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
else
    nzcv = '0010';
    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);
if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
else
    assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if nzcv<1> == '0' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
        assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
        if stage == MOPSStage_Main then
            stagecpysize = cpysize - postsize;
            // Check if the parameters to this instruction are valid.
            if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = FALSE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                stagecpysize = postsize;
            // Check if the parameters to this instruction are valid for the epilogue.
            if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = TRUE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        if supports_option_a then
            while SInt(stagecpysize) != 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= -1 * SInt(stagecpysize);
                readdata<8*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
                Mem[toaddress+cpysize, B, wacctype] = readdata<8*8-1:0>;
                cpysize = cpysize + B;
                stagecpysize = stagecpysize + B;
if stage != MOPSStage_Prologue then
  X[n, 64] = cpysize;
else
  while UInt(stagecpysize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
    assert B <= UInt(stagecpysize);

    readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
    Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
    toaddress = toaddress + B;

    cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;

  if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
  if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
C6.2.76 CPYFPTN, CPYFMTN, CPYFETN

Memory Copy Forward-only, reads and writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTN, then CPYFMTN, and then CPYFETN.

CPYFPTN performs some preconditioning of the arguments suitable for using the CPYFMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFETN performs the last part of the memory copy.

--- Note ---

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

--- Note ---

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

--- Note ---

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.

• At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

For CPYFETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.

• Xs holds the lowest address that the copy is copied from -Xn.

• Xd holds the lowest address that the copy is made to -Xn.

• At the end of the instruction, the value of Xn is written back with 0.

For CPYFETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes remaining to be copied in the memory copy in total.

• Xs holds the lowest address that the copy is copied from.

• Xd holds the lowest address that the copy is copied to.

• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

**Integer**

(FEAT_MOPS)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 16 | 15 | 12 | 11 | 10 | 9 | 5 | 4 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sz | 0 1 | 1 0 | 0 1 | op1 | 0 | Rs | 1 1 | 1 1 | 0 1 | Rn | Rd |

**Epilogue variant**

Applies when op1 == 10.

CPYFETN [Xd]!, [Xs]!, <Xn>!

**Main variant**

Applies when op1 == 01.

CPYFMTN [Xd]!, [Xs]!, <Xn>!

**Prologue variant**

Applies when op1 == 00.

CPYFPTN [Xd]!, [Xs]!, <Xn>!
Decode for all variants of this encoding

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";
if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);
if stage == MOPSSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;
if supports_option_a then
    nzcv = '0000';
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
else
    nzcv = '0010';
    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);
if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
else
    assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if nzcv<1> == '0' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
            assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;
        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            stagecpysize = postsize;
            // Check if the parameters to this instruction are valid for the epilogue.
            if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = TRUE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        if supports_option_a then
            while SInt(stagecpysize) != 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= -1 * SInt(stagecpysize);
                readdata<8*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
                Mem[toaddress+cpysize, B, wacctype] = readdata<8*8-1:0>;
                cpysize = cpysize + B;
                stagecpysize = stagecpysize + B;
if stage != MOPSStage_Prologue then
  X[n, 64] = cpysize;
else
  while UInt(stagecpysize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
    assert B <= UInt(stagecpysize);

    readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
    Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
    toaddress = toaddress + B;
    cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;

  if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
  if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
C6.2.77 CPYFPTRN, CPYFMTRN, CPYFETRN

Memory Copy Forward-only, reads and writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTRN, then CPYFMTRN, and then CPYFETRN.

CPYFPTRN performs some preconditioning of the arguments suitable for using the CPYFMTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFETRN performs the last part of the memory copy.

— Note —

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

— Note —

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPTRN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPTRN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFETRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number and holds -1 * the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from - Xn.
- Xd holds the lowest address that the copy is made to - Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFETRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

**Integer**

(FEAT_MOPS)

<table>
<thead>
<tr>
<th>sz</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>op1</th>
<th>0</th>
<th>Rs</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Epilogue variant**

Applies when op1 == 10.

CPYFETRN [<Xd>]!, [<Xs>]!, <Xn>!

**Main variant**

Applies when op1 == 01.

CPYFMTTRN [<Xd>]!, [<Xs>]!, <Xn>!

**Prologue variant**

Applies when op1 == 00.

CPYFPTRN [<Xd>]!, [<Xs>]!, <Xn>!
Decode for all variants of this encoding

```c
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
  case op1 of
    when '00' stage = MOPSSStage_Prologue;
    when '01' stage = MOPSSStage_Main;
    when '10' stage = MOPSSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";
  end case;

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```c
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFF<63:0>;
```
if supports_option_a then
    nzc\textsubscript{v} = '0000';
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
else
    nzc\textsubscript{v} = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
else
    assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzc\textsubscript{v}<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if nzc\textsubscript{v}<1> == '0' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
            assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

            if stage == MOPSStage_Main then
                stagecpysize = cpysize - postsize;

            // Check if the parameters to this instruction are valid.
            if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = FALSE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                stagecpysize = postsize;

            // Check if the parameters to this instruction are valid for the epilogue.
            if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = TRUE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if supports_option_a then
                    while SInt(stagecpysize) != 0 do
                        // IMP DEF selection of the block size that is worked on. While many
                        // implementations might make this constant, that is not assumed.
                        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                        assert B <= -1 * SInt(stagecpysize);

                        readda\textsubscript{t}a<8\times8-1:0> = Mem[fromaddress+cpysize, B, racctype];
                        Mem[toaddress+cpysize, B, wacctype] = readda\textsubscript{t}a<8\times8-1:0>;
                        cpysize = cpysize + B;
                        stagecpysize = stagecpysize + B;
if stage != MOPSStage_Prologue then
  X[n, 64] = cpysize;
else
  while UInt(stagecpysize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizEChoice(toaddress, fromaddress, cpysize);
    assert B <= UInt(stagecpysize);

    readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
    Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
    toaddress = toaddress + B;
    fromaddress = fromaddress + B;
    cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;

  if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
  if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
C6.2.78 CPYFPTWN, CPYFMTWN, CPYFETWN

Memory Copy Forward-only, reads and writes unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTWN, then CPYFMTWN, and then CPYFETWN.

CPYFPTWN performs some preconditioning of the arguments suitable for using the CPYFMTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFETWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPTWN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPTWN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.

• At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

For CPYFETWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.

• Xs holds the lowest address that the copy is copied from -Xn.

• Xd holds the lowest address that the copy is made to -Xn.

• At the end of the instruction, the value of Xn is written back with 0.

For CPYFETWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes remaining to be copied in the memory copy in total.

• Xs holds the lowest address that the copy is copied from.

• Xd holds the lowest address that the copy is copied to.

• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

**Integer**

(FEAT_MOPS)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>op1</td>
</tr>
<tr>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
</tr>
<tr>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Epilogue variant**

Applies when op1 == 10.

CPYFETWN [Xd], [Xs], <Xn>

**Main variant**

Applies when op1 == 01.

CPYFMTWN [X<bd)], [X<es)], <Xn>

**Prologue variant**

Applies when op1 == 00.

CPYFPTWN [X<bd)], [X<es)], <Xn>
Decode for all variants of this encoding

```c
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
    when '00' stage = MOPSSStage_Prologue;
    when '01' stage = MOPSSStage_Main;
    when '10' stage = MOPSSStage_Epilogue;
otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler symbols

- `<Xd>`: For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

- `<Xs>`: For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

- `<Xn>`: For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```c
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFF<63:0>;
```
if supports_option_a then
    nzcv = '0000';
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
else
    nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
else
    assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if nzcv<1> == '0' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
            assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

        if stage == MOPSStage_Main then
            stagecpysize = cpysize - postsize;
            // Check if the parameters to this instruction are valid.
            if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = FALSE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                stagecpysize = postsize;
                // Check if the parameters to this instruction are valid for the epilogue.
                if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                    boolean wrong_option = FALSE;
                    boolean from_epilogue = TRUE;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            if supports_option_a then
                while SInt(stagecpysize) != 0 do
                    // IMP DEF selection of the block size that is worked on. While many
                    // implementations might make this constant, that is not assumed.
                    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                    assert B <= -1 * SInt(stagecpysize);
                    readdata<8*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
                    Mem[toaddress+cpysize, B, wacctype] = readdata<8*8-1:0>;
                    cpysize = cpysize + B;
                    stagecpysize = stagecpysize + B;
if stage != MOPSStage_Prologue then
  X[n, 64] = cpysize;
else
  while UInt(stagecpysize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPysizeChoice(toaddress, fromaddress, cpysize);
    assert B <= UInt(stagecpysize);

    readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
    Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
    toaddress = toaddress + B;
    fromaddress = fromaddress + B;
    stagecpysize = stagecpysize - B;

    if stage != MOPSStage_Prologue then
      X[n, 64] = cpysize;
      X[d, 64] = toaddress;
      X[s, 64] = fromaddress;

    if stage == MOPSStage_Prologue then
      X[n, 64] = cpysize;
      X[d, 64] = toaddress;
      X[s, 64] = fromaddress;
      PSTATE.<N,Z,C,V> = nzcv;
C6.2.79  CPYFPWN, CPYFMWN, CPYFEWN

Memory Copy Forward-only, writes non-temporal. These instructions perform a memory copy. The prologue, main,
and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWN,
then CPYFMWN, and then CPYFEWN.

CPYFPWN performs some preconditioning of the arguments suitable for using the CPYFMWN instruction, and
performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWN performs an IMPLEMENTATION
DEFINIED amount of the memory copy. CPYFEWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can
be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable
for a memory copy only where there is no overlap between the source and destination locations, or where the source
address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is
IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWN, option A (which results in encoding PSTATE.C = 0):

• If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
• Xs holds the original Xs + saturated Xn.
• Xd holds the original Xd + saturated Xn.
• Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
• PSTATE.{N,Z,V} are set to \{0,0,0\}.

After execution of CPYFPWN, option B (which results in encoding PSTATE.C = 1):

• If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
• Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
• Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
• Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
• PSTATE.{N,Z,V} are set to \{0,0,0\}.

For CPYFMWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the
memory copy in total.
• Xs holds the lowest address that the copy is copied from -Xn.
• Xd holds the lowest address that the copy is made to -Xn.
• At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be
copied in the memory copy in total.

For CPYFMWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.

• At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

• Xn is treated as a signed 64-bit number and holds \(-1 \times\) the number of bytes remaining to be copied in the memory copy in total.

• Xs holds the lowest address that the copy is copied from \(-Xn\).

• Xd holds the lowest address that the copy is made to \(-Xn\).

• At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes remaining to be copied in the memory copy in total.

• Xs holds the lowest address that the copy is copied from.

• Xd holds the lowest address that the copy is copied to.

• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

**Integer**

(FEAT_MOPS)

<table>
<thead>
<tr>
<th>[31 16 15 12 10 9]</th>
<th>[5 4]</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz 0 1</td>
<td>1 0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>Rs 0 1</td>
<td>1 0 1</td>
<td>0 1</td>
</tr>
</tbody>
</table>

**Epilogue variant**

Applies when op1 == 10.

CPYFEWN \([Xd]!\), \([Xs]!\), \(<Xn>!\)

**Main variant**

Applies when op1 == 01.

CPYFMWN \([Xd]!\), \([Xs]!\), \(<Xn>!\)

**Prologue variant**

Applies when op1 == 00.

CPYFPWN \([Xd]!\), \([Xs]!\), \(<Xn>!\)
Decode for all variants of this encoding

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFF<63:0>;}
if supports_option_a then
    nzc = '0000';
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
else
    nzc = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
    else
        boolean zero_size_exceptions = MemCpyZeroSizeCheck();

        // Check if this version is consistent with the state of the call.
        if zero_size_exceptions || SInt(cpysize) != 0 then
            if supports_option_a then
                if nzc<1> == '1' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                else
                    if nzc<1> == '0' then // PSTATE.C
                        boolean wrong_option = TRUE;
                        boolean from_epilogue = stage == MOPSStage_Epilogue;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
                assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

                if stage == MOPSStage_Main then
                    stagecpysize = cpysize - postsize;

                    // Check if the parameters to this instruction are valid.
                    if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                        boolean wrong_option = FALSE;
                        boolean from_epilogue = FALSE;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                    else
                        stagecpysize = postsize;

                        // Check if the parameters to this instruction are valid for the epilogue.
                        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                            boolean wrong_option = FALSE;
                            boolean from_epilogue = TRUE;
                            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                        else
                            if supports_option_a then
                                while SInt(stagecpysize) != 0 do
                                    // IMP DEF selection of the block size that is worked on. While many
                                    // implementations might make this constant, that is not assumed.
                                    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                                    assert B <= -1 * SInt(stagecpysize);

                                    readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
                                    Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
                                    cpysize = cpysize + B;
                                    stagecpysize = stagecpysize + B;
if stage \neq \text{MOPSStage\_Prologue} then
X[n, 64] = cpysize;
else
    while \text{UInt}(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = \text{CPYSizeChoice}(toaddress, fromaddress, cpysize);
        assert B \leq \text{UInt}(stagecpysize);

        readdata<B*8-1:0> = \text{Mem}[fromaddress, B, racctype];
        \text{Mem}[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage \neq \text{MOPSStage\_Prologue} then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;
        if stage == \text{MOPSStage\_Prologue} then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;
            PSTATE.<N,Z,C,V> = nzcv;
C6.2.80 CPYFPWT, CPYFMWT, CPYFEWT

Memory Copy Forward-only, writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWT, then CPYFMWT, and then CPYFEWT.

CPYFPWT performs some preconditioning of the arguments suitable for using the CPYFMWT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWT performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWT, option A (which results in encoding PSTATE.C = 0):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWT, option B (which results in encoding PSTATE.C = 1):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from -Xn.
• Xd holds the lowest address that the copy is made to -Xn.
• At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

**Integer**

(FEAT_MOPS)

<table>
<thead>
<tr>
<th>sz</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>op1</th>
<th>0</th>
<th>Rs</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

**Epilogue variant**

Applies when op1 = 10.

CPYFEWT [Xd]!, [Xs]!, <Xn>!

**Main variant**

Applies when op1 = 01.

CPYFMT [Xd]!, [Xs]!, <Xn>!

**Prologue variant**

Applies when op1 = 00.

CPYFPWT [Xd]!, [Xs]!, <Xn>!
Decode for all variants of this encoding

if !HaveFeatMOPPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
    when '00' stage = MOPSSStage_Prologue;
    when '01' stage = MOPSSStage_Main;
    when '10' stage = MOPSSStage_Epilogue;
otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFF<63:0>;
if supports_option_a then
  nzcv = '0000';
  // Copy in the forward direction offsets the arguments.
  toaddress = toaddress + cpysize;
  fromaddress = fromaddress + cpysize;
  cpysize = Zeros(64) - cpysize;
else
  nzcv = '0010';

  // IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

if SInt(cpysize) > 0 then
  assert SInt(stagecpysize) <= SInt(cpysize);
else
  assert SInt(stagecpysize) >= SInt(cpysize);
else
  boolean zero_size_exceptions = MemCpyZeroSizeCheck();

  // Check if this version is consistent with the state of the call.
  if zero_size_exceptions || SInt(cpysize) != 0 then
    if supports_option_a then
      if nzcv<1> == '1' then // PSTATE.C
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        if nzcv<1> == '0' then // PSTATE.C
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
  if stage == MOPSStage_Main then
    stagecpysize = cpysize - postsize;
  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;
  // Check if the parameters to this instruction are valid for the epilogue.
  if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    if supports_option_a then
      while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<8*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<8*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;
if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

    if stage != MOPSStage_Prologue then
        X[n, 64] = cpysize;
        X[d, 64] = toaddress;
        X[s, 64] = fromaddress;
    if stage == MOPSStage_Prologue then
        X[n, 64] = cpysize;
        X[d, 64] = toaddress;
        X[s, 64] = fromaddress;
        PSTATE.<N,Z,C,V> = nzcv;
C6.2.81 CPYFPWTN, CPYFMWTN, CPYFEWTN

Memory Copy Forward-only, writes unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTN, then CPYFMWTN, and then CPYFEWTN.

CPYFPWTN performs some preconditioning of the arguments suitable for using the CPYFMWTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWTN performs the last part of the memory copy.

——— Note ————
The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

——— Note ————
The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

——— Note ————
Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.

• At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.

• Xs holds the lowest address that the copy is copied from -Xn.

• Xd holds the lowest address that the copy is made to -Xn.

• At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes remaining to be copied in the memory copy in total.

• Xs holds the lowest address that the copy is copied from.

• Xd holds the lowest address that the copy is copied to.

• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

**Integer**

(FEAT_MOPS)

<table>
<thead>
<tr>
<th>sz</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>op1</th>
<th>0</th>
<th>Rs</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
<th>0</th>
<th>op2</th>
</tr>
</thead>
</table>

**Epilogue variant**

Applies when op1 == 10.

CPYFEWTN [Xd], [Xs], Xn!

**Main variant**

Applies when op1 == 01.

CPYFPWTN [Xd], [Xs], Xn!

**Prologue variant**

Applies when op1 == 00.

CPYFPWTN [Xd], [Xs], Xn!
Decode for all variants of this encoding

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler symbols

<Xd>
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs>
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn>
For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFF<63:0>;
if supports_option_a then
  nzcv = '0000';
  // Copy in the forward direction offsets the arguments.
  toaddress = toaddress + cpysize;
  fromaddress = fromaddress + cpysize;
  cpysize = Zeros(64) - cpysize;
else
  nzcv = '0010';
  // IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

if SInt(cpysize) > 0 then
  assert SInt(stagecpysize) <= SInt(cpysize);
else
  assert SInt(stagecpysize) >= SInt(cpysize);
else
  boolean zero_size_exceptions = MemCpyZeroSizeCheck();

  // Check if this version is consistent with the state of the call.
  if zero_size_exceptions || SInt(cpysize) != 0 then
    if supports_option_a then
      if nzcv<1> == '1' then // PSTATE.C
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue,
        options);
      else
        if nzcv<1> == '0' then // PSTATE.C
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue,
          options);
    bits(64) postsize = CPYPPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPStage_Main then
      stagecpysize = cpysize - postsize;
      // Check if the parameters to this instruction are valid.
      if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = FALSE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue,
        options);
      else
        stagecpysize = postsize;
      // Check if the parameters to this instruction are valid for the epilogue.
      if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = TRUE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue,
        options);
      if supports_option_a then
        while SInt(stagecpysize) != 0 do
          // IMP DEF selection of the block size that is worked on. While many
          // implementations might make this constant, that is not assumed.
          B = CPYSizeChoice(toaddress, fromaddress, cpysize);
          assert B <= -1 * SInt(stagecpysize);

          readdata<8*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
          Mem[toaddress+cpysize, B, wacctype] = readdata<8*8-1:0>;
          cpysize = cpysize + B;
          stagecpysize = stagecpysize + B;
if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        toaddress = toaddress + B;
        fromaddress = fromaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

    if stage != MOPSStage_Prologue then
        X[n, 64] = cpysize;
        X[d, 64] = toaddress;
        X[s, 64] = fromaddress;
    if stage == MOPSStage_Prologue then
        X[n, 64] = cpysize;
        X[d, 64] = toaddress;
        X[s, 64] = fromaddress;
        PSTATE.<N,Z,C,V> = nzcv;
C6.2.82 CPYFPWTRN, CPYFMWTRN, CPYFEWTRN

Memory Copy Forward-only, writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTRN, then CPYFMWTRN, and then CPYFEWTRN.

CPYFPWTRN performs some preconditioning of the arguments suitable for using the CPYFMWTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWTRN performs the last part of the memory copy.

--- Note ---

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

---

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

--- Note ---

Portable software should not assume that the choice of algorithm is constant.

---

After execution of CPYFPWTRN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWTRN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.

• At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

• Xn is treated as a signed 64-bit number and holds \(-1^* the number of bytes remaining to be copied in the memory copy in total.

• Xs holds the lowest address that the copy is copied from \(-Xn.

• Xd holds the lowest address that the copy is made to \(-Xn.

• At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes remaining to be copied in the memory copy in total.

• Xs holds the lowest address that the copy is copied from.

• Xd holds the lowest address that the copy is copied to.

• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)

<table>
<thead>
<tr>
<th>[31 30 29 28] [27 26 25 24] [23 22 21 20]</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>op1</td>
<td>0</td>
<td>Rs</td>
</tr>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Epilogue variant

Applies when op1 == 10.
CPYFEWTRN [<Xd>], [<Xs>], <Xn>!

Main variant

Applies when op1 == 01.
CPYFMWTRN [<Xd>], [<Xs>], <Xn>!

Prologue variant

Applies when op1 == 00.
CPYFPWTRN [<Xd>], [<Xs>], <Xn>!
Decode for all variants of this encoding

if (!HaveFeatMOPS()) then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";
if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);
if stage == MOPSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFF<63:0>;
if supports_option_a then
  nzcv = '0000';
  // Copy in the forward direction offsets the arguments.
  toaddress = toaddress + cpysize;
  fromaddress = fromaddress + cpysize;
  cpysize = Zeros(64) - cpysize;
else
  nzcv = '0010';

  // IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

  if SIInt(cpysize) > 0 then
    assert SIInt(stagecpysize) <= SIInt(cpysize);
  else
    assert SIInt(stagecpysize) >= SIInt(cpysize);
  
else
  boolean zero_size_exceptions = MemCpyZeroSizeCheck();

  // Check if this version is consistent with the state of the call.
  if zero_size_exceptions || SIInt(cpysize) != 0 then
    if supports_option_a then
      if nzcv<1> == '1' then // PSTATE.C
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        if nzcv<1> == '0' then // PSTATE.C
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SIInt(postsize) == 0;

if stage == MOPSStage_Main then
  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

  // Check if the parameters to this instruction are valid for the epilogue.
  if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
else
  if supports_option_a then
    while SIInt(stagecpysize) != 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = CPYSizeChoice(toaddress, fromaddress, cpysize);
      assert B <= -1 || SIInt(stagecpysize);

readdata<8*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
Mem[toaddress+cpysize, B, wacctype] = readdata<8*8-1:0>;
  cpysize = cpysize + B;
  stagecpysize = stagecpysize + B;
if stage != MOPSStage_Prologue then
X[n, 64] = cpysize;
else
  while Uint(stagecpysize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
    assert B <= Uint(stagecpysize);
    readdata<0:8-1:0> = Mem[fromaddress, B, racctype];
    Mem[toaddress, B, wacctype] = readdata<0:8-1:0>;
    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
    cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;
  end
if stage != MOPSStage_Prologue then
  X[n, 64] = cpysize;
  X[d, 64] = toaddress;
  X[s, 64] = fromaddress;
if stage == MOPSStage_Prologue then
  X[n, 64] = cpysize;
  X[d, 64] = toaddress;
  X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;
C6.2.83 CPYFPWTWN, CPYFMWTWN, CPYFEWTWN

Memory Copy Forward-only, writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTWN, then CPYFMWTWN, and then CPYFEWTWN.

CPYFPWTWN performs some preconditioning of the arguments suitable for using the CPYFMWTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWTWN performs the last part of the memory copy.

Note
The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note
Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWTWN, option A (which results in encoding PSTATE.C = 0):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWTWN, option B (which results in encoding PSTATE.C = 1):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from -Xn.
• Xd holds the lowest address that the copy is made to -Xn.
• At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xs is written back with the lowest address that has not been copied from.
  — the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)

| [31 30 29 28][27 26 25 24|23 22 21 20] | 16|15 | 12|11 10 9 | 5|4 | 0 | 0110 01 01 |
| sz | 0 | 1 | 1 | 0 | 0 | op1 | 0 | Rs | 0 | 1 | 0 | 1 | 0 | Rd |
| op2 |

Epilogue variant
Applies when op1 == 10.
CPYFEWTWN [<Xd>]!, [<Xs>]!, <Xn>!

Main variant
Applies when op1 == 01.
CPYFMWTWN [<Xd>]!, [<Xs>]!, <Xn>!

Prologue variant
Applies when op1 == 00.
CPYFPWTWN [<Xd>]!, [<Xs>]!, <Xn>!
Decode for all variants of this encoding

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);
if stage == MOPSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFF<63:0>;
if supports_option_a then
    nzc = '0000';
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
else
    nzc = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
else
    assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzc<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if nzc<1> == '0' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            stagecpysize = postsize;

            // Check if the parameters to this instruction are valid for the epilogue.
            if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = TRUE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

        if supports_option_a then
            while SInt(stagecpysize) != 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= -1 * SInt(stagecpysize);

                readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
                Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
                cpysize = cpysize + B;
                stagecpysize = stagecpysize + B;
if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
else
    while UInt(stagecpsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpsize);

        readdata<8B-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<8B-1:0>;
        toaddress = toaddress + B;
        fromaddress = fromaddress + B;

        cpysize = cpysize - B;
        stagecpsize = stagecpsize - B;

    if stage != MOPSStage_Prologue then
        X[n, 64] = cpysize;
        X[d, 64] = toaddress;
        X[s, 64] = fromaddress;
    if stage == MOPSStage_Prologue then
        X[n, 64] = cpysize;
        X[d, 64] = toaddress;
        X[s, 64] = fromaddress;
        PSTATE.<N,Z,C,V> = nzcv;
C6.2.84 CPYP, CPYM, CPYE

Memory Copy. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYP, then CPYM, and then CPYE.

CPYP performs some preconditioning of the arguments suitable for using the CPYM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYM performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYE performs the last part of the memory copy.

--- Note ---
The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYP, the following saturation logic is applied:

If \( X_n \text{<63:55>} \neq 000000000, \) the copy size \( X_n \) is saturated to \( 0x007FFFFFFFFFFFFF \).

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If \((X_s > X_d) \&\& (X_d + \text{saturated} X_n) > X_s\), then direction = forward

Elsif \((X_s < X_d) \&\& (X_s + \text{saturated} X_n) > X_d\), then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

--- Note ---
Portable software should not assume that the choice of algorithm is constant.

After execution of CPYP, option A (which results in encoding PSTATE.C = 0):

- PSTATE.\( \{N,Z,V\} \) are set to \( \{0,0,0\} \).
- If the copy is in the forward direction, then:
  - \( X_s \) holds the original \( X_s + \text{saturated} X_n \).
  - \( X_d \) holds the original \( X_d + \text{saturated} X_n \).
  - \( X_n \) holds \(-1\)\* saturated \( X_n \) + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - \( X_s \) and \( X_d \) are unchanged.
  - \( X_n \) holds the saturated value of \( X_n \) - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYP, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - \( X_s \) holds the original \( X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  - \( X_d \) holds the original \( X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  - \( X_n \) holds the saturated \( X_n \) - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.\( \{N,Z,V\} \) are set to \( \{0,0,0\} \).
- If the copy is in the backward direction, then:
  - \( X_s \) holds the original \( X_s + \text{saturated} X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  - \( X_d \) holds the original \( X_d + \text{saturated} X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  - \( X_n \) holds the saturated \( X_n \) - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.\( \{N,Z,V\} \) are set to \( \{1,0,0\} \).
For CPYM, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is copied to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYM, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYE, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is made to.
  - Xd holds the lowest address that the copy is made to.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
— Xd holds the highest address that the copy is copied to -Xn+1.
— At the end of the instruction, the value of Xn is written back with 0.

For CPYE, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.

• If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from.
  — Xd holds the highest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz [0 1 1 1 0 1 0 1 0]</td>
<td>Rs [0 0 0 0 1]</td>
<td>Rn Rd [0 1]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Epilogue variant

Applies when op1 == 10.

CPYE [Xd]!, [Xs]!, <Xn>!

Main variant

Applies when op1 == 01.

CPYM [Xd]!, [Xs]!, <Xn>!

Prologue variant

Applies when op1 == 00.

CPYP [Xd]!, [Xs]!, <Xn>!

Decode for all variants of this encoding

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP   EndOfInstruction();

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
  if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFF<63:0>;

boolean forward;
if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) & (UInt(fromaddress<55:0>) <
  UInt(toaddress<55:0> + cpysize<55:0>)) then
  forward = TRUE;
elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>))) then
    forward = FALSE;
else
    forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports_option_a then
    nzcv = '0000';
    if forward then
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';
    end
    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
else
    assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if nzcv<1> == '0' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            end
        end
        bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
        assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;
        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            stagecpysize = postsize;
            // Check if the parameters to the epilogue are valid.
            if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = TRUE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                stagecpysize = postsize;
            end
        end
    end
end
if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        
        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);
            
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        
        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);
                
                if nzcv<3> == '0' then // PSTATE.N
                    readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
                    Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
                    fromaddress = fromaddress + B;
                    toaddress = toaddress + B;
                else
                    readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
                    Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
                    fromaddress = fromaddress - B;
                    toaddress = toaddress - B;
                
                cpysize = cpysize - B;
                stagecpysize = stagecpysize - B;
                
                if stage != MOPSStage_Prologue then
                    X[n, 64] = cpysize;
                    X[d, 64] = toaddress;
                    X[s, 64] = fromaddress;
        
        if stage == MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;
            PSTATE.<N,Z,C,V> = nzcv;
C6.2.85 CPYN, CPYMN, CPYEN

Memory Copy, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPN, then CPYMN, and then CPYEN.

CPYPN performs some preconditioning of the arguments suitable for using the CPYMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEN performs the last part of the memory copy.

--- Note ---

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

--- Note ---

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.
For CPYMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
— Xd holds the highest address that the copy is copied to -Xn+1.
— At the end of the instruction, the value of Xn is written back with 0.

For CPYEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes to be copied in the memory copy in total.

• If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.

• If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

Integer
(FEAT_MOPS)

| [31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 12|11 10 9 | 5 4 | 0 |
| sz | 0 1 | 1 | 0 1 | op1 | 0 | Rs | 1 | 1 | 0 | 0 1 | Rn | Rd |

Epilogue variant
Applies when op1 == 10.
CPYEN [<Xd>], [<Xs>], <Xn>!

Main variant
Applies when op1 == 01.
CPYMN [<Xd>], [<Xs>], <Xn>!

Prologue variant
Applies when op1 == 00.
CPYPN [<Xd>], [<Xs>], <Xn>!

Decode for all variants of this encoding

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;
integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";
CheckMOPSEnabled();
if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
case c of
  when Constraint_UNDEF UNDEFINED;
  when Constraint_NOP   EndOfInstruction();
Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);
boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);
if stage == MOPSStage_Prologue then
  if cpysize<63:55> != '00000000' then cpysize = 0x007FFFFFFFFFFFF<63:0>;
  boolean forward;
  if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) & (UInt(fromaddress<55:0>) <
    UInt(toaddress<55:0> + cpysize<55:0>))) then
    forward = TRUE;
elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>))) then
  forward = FALSE;
else
  forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports_option_a then
  nzcv = '0000';
  if forward then
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
  else
    // Copy in the reverse direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    nzcv = '1010';
  else
    nzcv = '0010';

  // IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

  if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
  else
    assert SInt(stagecpysize) >= SInt(cpysize);
  else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
      if supports_option_a then
        if nzcv<1> == '1' then // PSTATE.C
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
          if nzcv<1> == '0' then // PSTATE.C
            boolean wrong_option = TRUE;
            boolean from_epilogue = stage == MOPSStage_Epilogue;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
          else
            bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
            assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

            if stage == MOPSStage_Main then
              stagecpysize = cpysize - postsize;

              // Check if the parameters to this instruction are valid.
              if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = FALSE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
              else
                stagecpysize = postsize;

                // Check if the parameters to the epilogue are valid.
                if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                  boolean wrong_option = FALSE;
                  boolean from_epilogue = TRUE;
                  MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
              else
            end if
          end if
        end if
      end if
    end if
  end if
end if
if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);

    if SInt(cpysize) < 0 then
      assert B <= -1 * SInt(stagecpysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
      cpysize = cpysize + B;
      stagecpysize = stagecpysize + B;
    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
    end
    readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
    Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
    cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;
  end
  if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
  end
else
  while UInt(stagecpysize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
    assert B <= UInt(stagecpysize);
    if nzcv<3> == '0' then // PSTATE.N
      readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
      Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
      fromaddress = fromaddress + B;
      toaddress = toaddress + B;
    else
      readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
      Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
      fromaddress = fromaddress - B;
      toaddress = toaddress - B;
    end
    cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;
    if stage != MOPSStage_Prologue then
      X[n, 64] = cpysize;
      X[d, 64] = toaddress;
      X[s, 64] = fromaddress;
    end
    if stage == MOPSStage_Prologue then
      X[n, 64] = cpysize;
      X[d, 64] = toaddress;
      X[s, 64] = fromaddress;
      PSTATE.<N,Z,C,V> = nzcv;
C6.2.86 CPYPRN, CPYMRN, CPYERN

Memory Copy, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRN, then CPYMRN, and then CPYERN.

CPYPRN performs some preconditioning of the arguments suitable for using the CPYMRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERN performs the last part of the memory copy.

--- Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

---

For CPYPRN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

--- Note

Portable software should not assume that the choice of algorithm is constant.

---

After execution of CPYPRN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.
For CPYMRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $X_n$ is treated as a signed 64-bit number.
- If the copy is in the forward direction ($X_n$ is a negative number), then:
  - $X_n$ holds $-1\times$ the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$ holds the lowest address that the copy is copied from -$X_n$.
  - $X_d$ holds the lowest address that the copy is made to -$X_n$.
  - At the end of the instruction, the value of $X_n$ is written back with $-1\times$ the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction ($X_n$ is a positive number), then:
  - $X_n$ holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$ holds the highest address that the copy is copied from -$X_n+1$.
  - $X_d$ holds the highest address that the copy is copied to -$X_n+1$.
  - At the end of the instruction, the value of $X_n$ is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- $X_n$ holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - $X_s$ holds the lowest address that the copy is copied from.
  - $X_d$ holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of $X_n$ is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of $X_s$ is written back with the lowest address that has not been copied from.
    - the value of $X_d$ is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - $X_s$ holds the highest address that the copy is copied from +1.
  - $X_d$ holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of $X_n$ is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of $X_s$ is written back with the highest address that has not been copied from +1.
    - the value of $X_d$ is written back with the highest address that has not been copied to +1.

For CPYERN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- $X_n$ is treated as a signed 64-bit number.
- If the copy is in the forward direction ($X_n$ is a negative number), then:
  - $X_n$ holds $-1\times$ the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$ holds the lowest address that the copy is copied from -$X_n$.
  - $X_d$ holds the lowest address that the copy is made to -$X_n$.
  - At the end of the instruction, the value of $X_n$ is written back with 0.
- If the copy is in the backward direction ($X_n$ is a positive number), then:
  - $X_n$ holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$ holds the highest address that the copy is copied from -$X_n+1$. 

— Xd holds the highest address that the copy is copied to -Xn+1.
— At the end of the instruction, the value of Xn is written back with 0.

For CPYERN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes to be copied in the memory copy in total.

• If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.

• If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

**Integer**

(FEAT_MOPS)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz 0 1 1 1 0 1 0 0 0 1 0 1</td>
<td>Rs 1 0 0 0 0 1</td>
<td>Rn</td>
<td>Rd</td>
</tr>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Epilogue variant**

Applies when op1 == 10.

CPYERN [<Xd>], [<Xs>], <Xn>!

**Main variant**

Applies when op1 == 01.

CPYMNRN [<Xd>], [<Xs>], <Xn>!

**Prologue variant**

Applies when op1 == 00.

CPYPRN [<Xd>], [<Xs>], <Xn>!

**Decode for all variants of this encoding**

```plaintext
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
```
MOPSStage stage;
  case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

  CheckMOPSEnabled();
  if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP   EndOfInstruction();

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

  integer N = MaxBlockSizeCopiedBytes();
  bits(64) toaddress = X[d, 64];
  bits(64) fromaddress = X[s, 64];
  bits(64) cpysize = X[n, 64];
  bits(4) nzcv = PSTATE.<N,Z,C,V>;
  bits(64) stagecpysize;
  bits(8*N) readdata;
  integer B;
  if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);
  boolean supports_option_a = MemCpyOptionA();
  (racctype, wacctype) = MemCpyAccessTypes(options);
  if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;
    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < Uint(toaddress<55:0> + cpysize<55:0>))) then
      forward = TRUE;
elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>))) then
    forward = FALSE;
else
    forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports_option_a then
    nzcv = '0000';
    if forward then
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
    else
        boolean zero_size_exceptions = MemCpyZeroSizeCheck();

        // Check if this version is consistent with the state of the call.
        if zero_size_exceptions || SInt(cpysize) != 0 then
            if supports_option_a then
                if nzcv<1> == '1' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                else
                    if nzcv<1> == '0' then // PSTATE.C
                        boolean wrong_option = TRUE;
                        boolean from_epilogue = stage == MOPSStage_Epilogue;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                     
            bits<64> postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
            assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
        if stage == MOPSStage_Main then
            stagecpysize = cpysize - postsize;

            // Check if the parameters to this instruction are valid.
            if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = FALSE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                stagecpysize = postsize;

            // Check if the parameters to the epilogue are valid.
            if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = TRUE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);

                if nzcv<3> == '0' then // PSTATE.N
                    readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
                    Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
                    fromaddress = fromaddress + B;
                    toaddress = toaddress + B;
                else
                    readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
                    Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
                    fromaddress = fromaddress - B;
                    toaddress = toaddress - B;

                cpysize = cpysize - B;
                stagecpysize = stagecpysize - B;

                if stage != MOPSStage_Prologue then
                    X[n, 64] = cpysize;
                    X[d, 64] = toaddress;
                    X[s, 64] = fromaddress;
                else
                    X[n, 64] = cpysize;
                    X[d, 64] = toaddress;
                    X[s, 64] = fromaddress;
                    PSTATE.<N,Z,C,V> = nzcv;
C6.2.87  CPYPRT, CPYMRT, CPYERT

Memory Copy, reads unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRT, then CPYMRT, and then CPYERT.

CPYPRT performs some preconditioning of the arguments suitable for using the CPYMRT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERT performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRT, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRT, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRT, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.
For CPYMRT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds \(-1^*\) the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with \(-1^*\) the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds \(-1^*\) the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
— Xd holds the highest address that the copy is copied to -Xn+1.
— At the end of the instruction, the value of Xn is written back with 0.

For CPYERT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes to be copied in the memory copy in total.

• If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.

• If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 16 | 15 | 12 | 11 | 10 | 9 | 5 | 4 | 0 |
| sz | 0 | 1 | 1 | 1 | 0 | 1 | op1 | 0 | Rs | 0 | 0 | 1 | 0 | 0 | 1 | Rn | Rd |

Epilogue variant

Applies when op1 == 10.

CPYERT [<Xd>], [<Xs>], <Xn>!

Main variant

Applies when op1 == 01.

CPYMRT [<Xd>], [<Xs>], <Xn>!

Prologue variant

Applies when op1 == 00.

CPYPRT [<Xd>], [<Xs>], <Xn>!

Decode for all variants of this encoding

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";
CheckMOPSEnabled();
if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
assert c IN {Constraint_UNDEF, Constraint_NOP};
case c of
  when Constraint_UNDEF UNDEFINED;
  when Constraint_NOP   EndOfInstruction();
Assembler symbols
<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
Operation
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);
boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);
if stage == MOPSSStage_Prologue then
  if cpysize<63:55> != '0000000000' then cpysize = 0x007FFFFFFFFFFFF<63:0>;
  boolean forward;
  if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55:0> + cpysize<55:0>))) then
    forward = TRUE;
"
elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>)) then
    forward = FALSE;
else
    forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports_option_a then
    nzcvc = '0000';
    if forward then
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcvc = '1010';
        else
            nzcvc = '0010';
    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
    else
        boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcvc<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if nzcvc<1> == '0' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
            assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

            if stage == MOPSStage_Main then
                stagecpysize = cpysize - postsize;

                // Check if the parameters to this instruction are valid.
                if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                    boolean wrong_option = FALSE;
                    boolean from_epilogue = FALSE;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                else
                    stagecpysize = postsize;

                    // Check if the parameters to the epilogue are valid.
                    if ( cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize) ) then
                        boolean wrong_option = FALSE;
                        boolean from_epilogue = TRUE;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else

if supports_option_a then
    while \( \text{SInt}(\text{stagecpysize}) \neq 0 \) do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        \( B = \text{CPYSiz}e\text{Choice}(\text{toaddress}, \text{fromaddress}, \text{cpysize}) \);

        if \( \text{SInt}(\text{cpysize}) < 0 \) then
            assert \( B <= -1 \times \text{SInt}(\text{stagecpysize}) \);

            \( \text{readdata}_{B*8-1:0} = \text{Mem}[\text{fromaddress}+\text{cpysize}, B, \text{racctype}] \);
            \( \text{Mem}[\text{toaddress}+\text{cpysize}, B, \text{wacctype}] = \text{readdata}_{B*8-1:0} \);
            \( \text{cpysize} = \text{cpysize} + B \);
            \( \text{stagecpysize} = \text{stagecpysize} + B \);
        else
            assert \( B <= \text{SInt}(\text{stagecpysize}) \);

            \( \text{cpysize} = \text{cpysize} - B \);
            \( \text{stagecpysize} = \text{stagecpysize} - B \);
            \( \text{readdata}_{B*8-1:0} = \text{Mem}[\text{fromaddress}+\text{cpysize}, B, \text{racctype}] \);
            \( \text{Mem}[\text{toaddress}+\text{cpysize}, B, \text{wacctype}] = \text{readdata}_{B*8-1:0} \);

        if \( \text{stage} \neq \text{MOPSStage}_\text{Prologue} \) then
            \( X[n, 64] = \text{cpysize} \);
        else

        while \( \text{UInt}(\text{stagecpysize}) > 0 \) do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            \( B = \text{CPYSiz}e\text{Choice}(\text{toaddress}, \text{fromaddress}, \text{cpysize}) \);
            assert \( B <= \text{UInt}(\text{stagecpysize}) \);

            if \( \text{nzcv}_{3} == '0' \) then // PSTATE.<N
                \( \text{readdata}_{B*8-1:0} = \text{Mem}[\text{fromaddress}, B, \text{racctype}] \);
                \( \text{Mem}[\text{toaddress}, B, \text{wacctype}] = \text{readdata}_{B*8-1:0} \);
                \( \text{fromaddress} = \text{fromaddress} + B \);
                \( \text{toaddress} = \text{toaddress} + B \);
            else
                \( \text{readdata}_{B*8-1:0} = \text{Mem}[\text{fromaddress}-B, B, \text{racctype}] \);
                \( \text{Mem}[\text{toaddress}-B, B, \text{wacctype}] = \text{readdata}_{B*8-1:0} \);
                \( \text{fromaddress} = \text{fromaddress} - B \);
                \( \text{toaddress} = \text{toaddress} - B \);

            \( \text{cpysize} = \text{cpysize} - B \);
            \( \text{stagecpysize} = \text{stagecpysize} - B \);

            if \( \text{stage} \neq \text{MOPSStage}_\text{Prologue} \) then
            \( X[n, 64] = \text{cpysize} \);
            \( X[d, 64] = \text{toaddress} \);
            \( X[s, 64] = \text{fromaddress} \);
            \( \text{PSTATE.}<N,Z,C,V> = \text{nzcv} \);
            \( \text{X}[n, 64] = \text{cpysize} \);
            \( X[d, 64] = \text{toaddress} \);
            \( X[s, 64] = \text{fromaddress} \);
            \( \text{PSTATE.}<N,Z,C,V> = \text{nzcv} \);
C6.2.88 CPYPRTN, CPYMRTN, CPYERTN

Memory Copy, reads unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTN, then CPYMRTN, and then CPYERTN.

CPYPRTN performs some preconditioning of the arguments suitable for using the CPYMRTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERTN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRTN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRTN, option A (which results in encoding PSTATE.C = 0):

• PSTATE.{N,Z,V} are set to {0,0,0}.
• If the copy is in the forward direction, then:
  — Xs holds the original Xs + saturated Xn.
  — Xd holds the original Xd + saturated Xn.
  — Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
• If the copy is in the backward direction, then:
  — Xs and Xd are unchanged.
  — Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRTN, option B (which results in encoding PSTATE.C = 1):

• If the copy is in the forward direction, then:
  — Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  — Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  — Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  — PSTATE.{N,Z,V} are set to {0,0,0}.
• If the copy is in the backward direction, then:
  — Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  — Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  — Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  — PSTATE.{N,Z,V} are set to {1,0,0}.
For CPYMRTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
— Xd holds the highest address that the copy is copied to \(-X_n+1\).
— At the end of the instruction, the value of Xn is written back with 0.

For CPYERTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

**Integer**

(\(\text{FEAT}_\text{MOPS}\))

\[
\begin{array}{cccccccccccccccccc}
\hline
sz & 0 & 1 & 1 & | & 0 & 1 & op1 & 0 & | & Rs & 1 & 1 & 1 & 0 & | & 0 & 1 & | & Rn & | & Rd & \hline
\hline
\end{array}
\]

**Epilogue variant**

Applies when \(\text{op1} == 10\).

CPYERTN \([<Xd>]!, [<Xs>]!, <Xn>!\)

**Main variant**

Applies when \(\text{op1} == 01\).

CPYMRTN \([<Xd>]!, [<Xs>]!, <Xn>!\)

**Prologue variant**

Applies when \(\text{op1} == 00\).

CPYPRTN \([<Xd>]!, [<Xs>]!, <Xn>!\)

**Decode for all variants of this encoding**

\[
\text{if } !\text{HaveFeatMOPS() then UNDEFINED;}
\text{if } sz != '00' \text{ then UNDEFINED;}
\]

\[
\text{integer } d = \text{UInt(Rd)}; \\
\text{integer } s = \text{UInt(Rs)}; \\
\text{integer } n = \text{UInt(Rn)}; \\
\text{bits(4) options} = \text{op2}; \\
\]
MOPSSStage stage;
    case op1 of
        when '00' stage = MOPSSStage_Prologue;
        when '01' stage = MOPSSStage_Main;
        when '10' stage = MOPSSStage_Epilogue;
        otherwise SEE "Memory Copy and Memory Set";
    CheckMOPSEnabled();
    if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
        c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP   EndOfInstruction();

Asmber symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);
boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);
if stage == MOPSSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFF<63:0>;
    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) <
        UInt(toaddress<55:0> + cpysize<55:0>)) then
        forward = TRUE;
elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>))) then
    forward = FALSE;
else
    forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports_option_a then
    nzcv = '0000';
    if forward then
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
        nzcv = '1010';
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
else
    assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if nzcv<1> == '0' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        bits(64) postsize = CPYPPostSizeChoice(toaddress, fromaddress, cpysize);
        assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

        if stage == MOPSStage_Main then
            stagecpysize = cpysize - postsize;

            // Check if the parameters to this instruction are valid.
            if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = FALSE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                stagecpysize = postsize;

                // Check if the parameters to the epilogue are valid.
                if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                    boolean wrong_option = FALSE;
                    boolean from_epilogue = TRUE;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    // B = CPYSIZEChoice(toaddress, fromaddress, cpysize);

    if SInt(cpysize) < 0 then
      assert B <= -1 * SInt(stagecpysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;
    if stage != MOPSStage_Prologue then
      X[n, 64] = cpysize;
    end if
  end while
else
  while UInt(stagecpysize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    // B = CPYSIZEChoice(toaddress, fromaddress, cpysize);
    assert B <= UInt(stagecpysize);

    if nzcv<3> == '0' then // PSTATE.N
      readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
      Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
      fromaddress = fromaddress + B;
      toaddress = toaddress + B;
    else
      readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
      Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
      fromaddress = fromaddress - B;
      toaddress = toaddress - B;
    end if
    cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;
    if stage != MOPSStage_Prologue then
      X[n, 64] = cpysize;
      X[d, 64] = toaddress;
      X[s, 64] = fromaddress;
    end if
    if stage == MOPSStage_Prologue then
      X[n, 64] = cpysize;
      X[d, 64] = toaddress;
      X[s, 64] = fromaddress;
      PSTATE.<N,Z,C,V> = nzcv;
  end while
end if
C6.2.89 CPYPRTRN, CPYMRTRN, CPYERTRN

Memory Copy, reads unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTRN, then CPYMRTRN, and then CPYERTRN.

CPYPRTRN performs some preconditioning of the arguments suitable for using the CPYMRTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERTRN performs the last part of the memory copy.

--- Note ---
The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRTRN, the following saturation logic is applied:
If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:
If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

--- Note ---
Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRTRN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRTRN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.
For CPYMRTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
— Xd holds the highest address that the copy is copied to \(-X_n+1\).
— At the end of the instruction, the value of Xn is written back with 0.

For CPYERTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

**Integer**

(FEAT_MOPS)

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  | | | | | | | | | | | | | | | | | | | | | |
  sz 0 1 1 1 0 1 0 1 0 1 0 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  op2
```

**Epilogue variant**

Applies when op1 == 10.

CPYERTRN \[['Xd>]', ['Xs>]', Xn]!

**Main variant**

Applies when op1 == 01.

CPYMRTRN \[['Xd>]', ['Xs>]', Xn]!

**Prologue variant**

Applies when op1 == 00.

CPYPRTRN \[['Xd>]', ['Xs>]', Xn]!

**Decode for all variants of this encoding**

```cpp
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
```
MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP   EndOfInstruction();

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
  if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFF<63:0>;

  boolean forward;
  if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) &
    (UInt(fromaddress<55:0>) <
    UInt(toaddress<55:0> + cpysize<55:0>))) then
    forward = TRUE;
elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>))) then
  forward = FALSE;
else
  forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports_option_a then
  nzcv = '0000';
  if forward then
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
  else
    if !forward then
      // Copy in the reverse direction offsets the arguments.
      toaddress = toaddress + cpysize;
      fromaddress = fromaddress + cpysize;
      nzcv = '1010';
    else
      nzcv = '0010';
    // IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);
  if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
  else
    assert SInt(stagecpysize) >= SInt(cpysize);
  else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
  if zero_size_exceptions || SInt(cpysize) != 0 then
    if supports_option_a then
      if nzcv<1> == '1' then // PSTATE.C
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        if nzcv<1> == '0' then // PSTATE.C
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
        assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
        if stage == MOPSStage_Main then
          stagecpysize = cpysize - postsize;
          // Check if the parameters to this instruction are valid.
          if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
          else
            stagecpysize = postsize;
            // Check if the parameters to the epilogue are valid.
          if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizChoice(toaddress, fromaddress, cpysize);
    if SInt(cpysize) < 0 then
      assert B <= -1 * SInt(stagecpysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
      cpysize = cpysize + B;
      stagecpysize = stagecpysize + B;
    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
    readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
    Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
  end while
if stage != MOPSStage_Prologue then
  X[n, 64] = cpysize;
else
  while UInt(stagecpysize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizChoice(toaddress, fromaddress, cpysize);
    if nzcv<3> == '0' then // PSTATE.N
      readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
      Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
      fromaddress = fromaddress + B;
      toaddress = toaddress + B;
    else
      readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
      Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
      fromaddress = fromaddress - B;
      toaddress = toaddress - B;
    cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;
  end while
if stage != MOPSStage_Prologue then
  X[n, 64] = cpysize;
  X[d, 64] = toaddress;
  X[s, 64] = fromaddress;
if stage == MOPSStage_Prologue then
  X[n, 64] = cpysize;
  X[d, 64] = toaddress;
  X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;
C6.2.90 CPYPRTWN, CPYMRTWN, CPYERTWN

Memory Copy, reads unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTWN, then CPYMRTWN, and then CPYERTWN.

CPYPRTWN performs some preconditioning of the arguments suitable for using the CPYMRTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERTWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRTWN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRTWN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.\{N,Z,V\} are set to \{0,0,0\}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRTWN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.\{N,Z,V\} are set to \{0,0,0\}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.\{N,Z,V\} are set to \{1,0,0\}.
For CPYMRTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds \(-1 \times \) the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with \(-1 \times \) the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds \(-1 \times \) the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
— Xd holds the highest address that the copy is copied to -Xn+1.
— At the end of the instruction, the value of Xn is written back with 0.

For CPYERTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

**Integer**

(FEAT_MOPS)

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>op1</td>
<td>0</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>op2</td>
<td>---------------</td>
<td>---------------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

**Epilogue variant**

Applies when op1 == 10.

CPYERTWN [Xd]!, [Xs]!, <Xn>!

**Main variant**

Applies when op1 == 01.

CPYMRTWN [Xd]!, [Xs]!, <Xn>!

**Prologue variant**

Applies when op1 == 00.

CPYPRTWN [Xd]!, [Xs]!, <Xn>!

**Decode for all variants of this encoding**

```plaintext
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
```
MOPSSStage stage;
case op1 of
    when '00' stage = MOPSSStage_Prologue;
    when '01' stage = MOPSSStage_Main;
    when '10' stage = MOPSSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();

Assembler symbols

<Xd>
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs>
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn>
For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessType(options);

if stage == MOPSSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFF<63:0>;

boolean forward;
if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) <UInt(toaddress<55:0> + cpysize<55:0>))) then
    forward = TRUE;
elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>))) then
  forward = FALSE;
else
  forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports_option_a then
  nzcv = '0000';
  if forward then
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
  else
    if !forward then
      // Copy in the reverse direction offsets the arguments.
      toaddress = toaddress + cpysize;
      fromaddress = fromaddress + cpysize;
      nzcv = '1010';
    else
      nzcv = '0010';
    end if;
  else
    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);
  end if;

  if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
  else
    assert SInt(stagecpysize) >= SInt(cpysize);
  end if;
else
  boolean zero_size_exceptions = MemCpyZeroSizeCheck();

  // Check if this version is consistent with the state of the call.
  if zero_size_exceptions || SInt(cpysize) != 0 then
    if supports_option_a then
      if nzcv<1> == '1' then // PSTATE.C
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        if nzcv<1> == '0' then // PSTATE.C
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        end if;
      end if;
    else
      bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
      assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
    end if;
  end if;
end if;

if stage == MOPSStage_Main then
  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;
  end if;

  // Check if the parameters to the epilogue are valid.
  if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
  end if;
if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizelchoice(toaddress, fromaddress, cpysize);
    if SInt(cpysize) < 0 then
      assert B <= -1 * SInt(stagecpysize);
      readdata<8*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<8*8-1:0>;
      cpysize = cpysize + B;
      stagecpysize = stagecpysize + B;
    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
      readdata<8*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<8*8-1:0>;
  endwhile
if stage != MOPSStage_Prologue then
  X[n, 64] = cpysize;
else
  while UInt(stagecpysize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizelchoice(toaddress, fromaddress, cpysize);
    if nzcv<3> == '0' then // PSTATE.N
      readdata<8*8-1:0> = Mem[fromaddress, B, racctype];
      Mem[toaddress, B, wacctype] = readdata<8*8-1:0>;
      fromaddress = fromaddress + B;
      toaddress = toaddress + B;
    else
      readdata<8*8-1:0> = Mem[fromaddress-B, B, racctype];
      Mem[toaddress-B, B, wacctype] = readdata<8*8-1:0>;
      fromaddress = fromaddress - B;
      toaddress = toaddress - B;
    end
    cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;
    if stage != MOPSStage_Prologue then
      X[n, 64] = cpysize;
      X[d, 64] = toaddress;
      X[s, 64] = fromaddress;
    endelse
if stage == MOPSStage_Prologue then
  X[n, 64] = cpysize;
  X[d, 64] = toaddress;
  X[s, 64] = fromaddress;
PSTATE.<N,Z,C,V> = nzcv;
C6.2.91 CPYPT, CPYMT, CPYET

Memory Copy, reads and writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPT, then CPYMT, and then CPYET.

CPYPT performs some preconditioning of the arguments suitable for using the CPYMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYET performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPT, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPT, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPT, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}. 
For CPYMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds \(-1\times\) the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with \(-1\times\) the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYET, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds \(-1\times\) the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
— Xd holds the highest address that the copy is copied to -Xn+1.
— At the end of the instruction, the value of Xn is written back with 0.

For CPYET, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.
• If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

**Integer**

(FEAT_MOPS)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>0 1 1 1</td>
<td>0 1 op1 0</td>
<td>Rs</td>
<td>0 0 1 1</td>
<td>0 1</td>
<td>Rn</td>
</tr>
<tr>
<td></td>
<td>op2</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Epilogue variant**

Applies when op1 == 10.
CPYET [<Xd>], [<Xs>], <Xn>!

**Main variant**

Applies when op1 == 01.
CPYMT [<Xd>], [<Xs>], <Xn>!

**Prologue variant**

Applies when op1 == 00.
CPYPT [<Xd>], [<Xs>], <Xn>!

**Decode for all variants of this encoding**

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";
CheckMOPSEnabled();
if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  c = ConstraintUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
case c of
  when Constraint_UNDEF UNDEFINED;
  when Constraint_NOP   EndOfInstruction();
Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

integer N = MaxBlockSizeCopiedBytes();
bite(64) toaddress = X[d, 64];
bite(64) fromaddress = X[s, 64];
bite(64) cpysize = X[n, 64];
bite(4) nzcv = PSTATE.<N,Z,C,V>;
bite(64) stagecpysize;
bite(8*N) readdata;
integer B;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);
boolean supports_option_a = MemCpyOptionA();
{racctype, wacctype} = MemCpyAccessTypes(options);
if stage == MOPSStage_Prologue then
  if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFF<63:0>;
  boolean forward;
  if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) & (UInt(fromaddress<55:0>) <
    UInt(toaddress<55:0> + cpysize<55:0>)) then
    forward = TRUE;
elsif (UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) & (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>)) then
    forward = FALSE;
else
    forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);
if supports_option_a then
    nzcv = '0000';
    if forward then
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';
// IMP DEF selection of the amount covered by pre-processing.
stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);
if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
else
    assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if nzcv<1> == '0' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
                assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
                if stage == MOPSStage_Main then
                    stagecpysize = cpysize - postsize;
                    // Check if the parameters to this instruction are valid.
                    if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                        boolean wrong_option = FALSE;
                        boolean from_epilogue = FALSE;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                    else
                        stagecpysize = postsize;
                        // Check if the parameters to the epilogue are valid.
                        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                            boolean wrong_option = FALSE;
                            boolean from_epilogue = TRUE;
                            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
if supports_option_a then  
  while SInt(stagecpysize) != 0 do  
    // IMP DEF selection of the block size that is worked on. While many  
    // implementations might make this constant, that is not assumed.  
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);  
  
  if SInt(cpysize) < 0 then  
    assert B <= -1 * SInt(stagecpysize);  
    readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];  
    Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;  
    cpysize = cpysize + B;  
    stagecpysize = stagecpysize + B;  
  else  
    assert B <= SInt(stagecpysize);  
    cpysize = cpysize - B;  
    stagecpysize = stagecpysize - B;  
    readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];  
    Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;  
  
  if stage != MOPSStage_Prologue then  
    X[n, 64] = cpysize;  
  else  
    while UInt(stagecpysize) > 0 do  
      // IMP DEF selection of the block size that is worked on. While many  
      // implementations might make this constant, that is not assumed.  
      B = CPYSizeChoice(toaddress, fromaddress, cpysize);  
      assert B <= UInt(stagecpysize);  
      if nzcv<3> == '0' then // PSTATE.N  
        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];  
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;  
        fromaddress = fromaddress + B;  
        toaddress = toaddress + B;  
      else  
        readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];  
        Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;  
        fromaddress = fromaddress - B;  
        toaddress = toaddress - B;  
      cpysize = cpysize - B;  
      stagecpysize = stagecpysize - B;  
      if stage != MOPSStage_Prologue then  
        X[n, 64] = cpysize;  
        X[d, 64] = toaddress;  
        X[s, 64] = fromaddress;  
      else  
        X[n, 64] = cpysize;  
        X[d, 64] = toaddress;  
        X[s, 64] = fromaddress;  
        PSTATE.<N,Z,C,V> = nzcv;
**C6.2.92 CPYPTN, CPYMTN, CPYETN**

Memory Copy, reads and writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTN, then CPYMTN, and then CPYETN.

CPYPTN performs some preconditioning of the arguments suitable for using the CPYMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYETN performs the last part of the memory copy.

--- **Note** ---

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

--- ---

For CPYPTN, the following saturation logic is applied:

If \( X_{n<63:55>} \neq 000000000 \), the copy size \( X_n \) is saturated to \( 0x007FFFFFFFFFFFFF \).

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

- If \( (X_{s > X_{d}}) \&\& (X_{d + saturated X_n} > X_{s}) \), then direction = forward
- Elsif \( (X_{s < X_{d}}) \&\& (X_{s + saturated X_n} > X_{d}) \), then direction = backward
- Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

--- **Note** ---

Portable software should not assume that the choice of algorithm is constant.

--- ---

After execution of CPYPTN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.\{N,Z,V\} are set to \{0,0,0\}.
- If the copy is in the forward direction, then:
  - \( X_s \) holds the original \( X_s + saturated X_n \).
  - \( X_d \) holds the original \( X_d + saturated X_n \).
  - \( X_n \) holds \( -1^* saturated X_n + an IMPLEMENTATION DEFINED number of bytes copied \).
- If the copy is in the backward direction, then:
  - \( X_s \) and \( X_d \) are unchanged.
  - \( X_n \) holds the saturated value of \( X_n - an IMPLEMENTATION DEFINED number of bytes copied \).

After execution of CPYPTN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - \( X_s \) holds the original \( X_s + an IMPLEMENTATION DEFINED number of bytes copied \).
  - \( X_d \) holds the original \( X_d + an IMPLEMENTATION DEFINED number of bytes copied \).
  - \( X_n \) holds the saturated \( X_n - an IMPLEMENTATION DEFINED number of bytes copied \).
  - PSTATE.\{N,Z,V\} are set to \{0,0,0\}.
- If the copy is in the backward direction, then:
  - \( X_s \) holds the original \( X_s + saturated X_n - an IMPLEMENTATION DEFINED number of bytes copied \).
  - \( X_d \) holds the original \( X_d + saturated X_n - an IMPLEMENTATION DEFINED number of bytes copied \).
  - \( X_n \) holds the saturated \( X_n - an IMPLEMENTATION DEFINED number of bytes copied \).
  - PSTATE.\{N,Z,V\} are set to \{1,0,0\}.  

---
For CPYMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds \(-1\times\) the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with \(-1\times\) the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds \(-1\times\) the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
— Xd holds the highest address that the copy is copied to -Xn+1.
— At the end of the instruction, the value of Xn is written back with 0.

For CPYETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.
• If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

**Integer**

(FEAT_MOPS)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 12 11 10 9</th>
<th>5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>0 1 1</td>
<td>1 0 1 op1</td>
<td>0</td>
<td>Rs</td>
</tr>
<tr>
<td></td>
<td>1 1 1</td>
<td>0 1</td>
<td>Rn</td>
<td>Rd</td>
</tr>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Epilogue variant**

Applies when op1 == 10.

CPYETN [<Xd>!]!, [<Xs>!]!, <Xn>!

**Main variant**

Applies when op1 == 01.

CPYMTN [<Xd>!]!, [<Xs>!]!, <Xn>!

**Prologue variant**

Applies when op1 == 00.

CPYPTN [<Xd>!]!, [<Xs>!]!, <Xn>!

**Decode for all variants of this encoding**

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();
if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();

Assembler symbols

Xd
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

Xs
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

Xn
For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagcpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);
if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFF<63:0>;
    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) & (UInt(fromaddress<55:0>) < UInt(toaddress<55:0> + cpysize<55:0>))) then
        forward = TRUE;
elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>))) then
  forward = FALSE;
else
  forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports_option_a then
  nzcvc = '0000';
  if forward then
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
  else
    // Copy in the reverse direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    nzcvc = '1010';
  else
    nzcvc = '0010';

  // IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

  if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
  else
    assert SInt(stagecpysize) >= SInt(cpysize);
  else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

  // Check if this version is consistent with the state of the call.
  if zero_size_exceptions || SInt(cpysize) != 0 then
    if supports_option_a then
      if nzcvc<1> == '1' then // PSTATE.C
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        if nzcvc<1> == '0' then // PSTATE.C
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
          boolean wrong_option = FALSE;
          boolean from_epilogue = FALSE;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      bits(64) postsize = CPYPPostSizeChoice(toaddress, fromaddress, cpysize);
      assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

      if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

      // Check if the parameters to this instruction are valid.
      if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = FALSE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        stagecpysize = postsize;

      // Check if the parameters to the epilogue are valid.
      if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = TRUE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        boolean wrong_option = TRUE;
        boolean from_epilogue = FALSE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
if supports_option_a then
  while \( \text{SInt}(\text{stagecpysize}) \neq 0 \) do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    \( B = \text{CPYSizeChoice}(\text{toaddress}, \text{fromaddress}, \text{cpysize}) \);

    if \( \text{SInt}(\text{cpysize}) < 0 \) then
      assert \( B \leq -1 \times \text{SInt}(\text{stagecpysize}) \);
      \( \text{readdata}_{B \times 8-1:0} = \text{Mem}[\text{fromaddress} + \text{cpysize}, B, \text{racctype}] \);
      \( \text{Mem}[\text{toaddress} + \text{cpysize}, B, \text{wacctype}] = \text{readdata}_{B \times 8-1:0} \);
      \( \text{cpysize} = \text{cpysize} + B \);
      \( \text{stagecpysize} = \text{stagecpysize} + B \);
    else
      assert \( B \leq \text{SInt}(\text{stagecpysize}) \);
      \( \text{cpysize} = \text{cpysize} - B \);
      \( \text{stagecpysize} = \text{stagecpysize} - B \);
      \( \text{readdata}_{B \times 8-1:0} = \text{Mem}[\text{fromaddress} + \text{cpysize}, B, \text{racctype}] \);
      \( \text{Mem}[\text{toaddress} + \text{cpysize}, B, \text{wacctype}] = \text{readdata}_{B \times 8-1:0} \);
    if stage \(!=\) \text{MOPSStage_Prologue} then
      \( X[n, 64] = \text{cpysize} \);

  end

  while \( \text{UInt}(\text{stagecpysize}) > 0 \) do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    \( B = \text{CPYSizeChoice}(\text{toaddress}, \text{fromaddress}, \text{cpysize}) \);
    assert \( B \leq \text{UInt}(\text{stagecpysize}) \);

    if nzcv<3> == '0' then // PSTATE.N
      \( \text{readdata}_{B \times 8-1:0} = \text{Mem}[\text{fromaddress}, B, \text{racctype}] \);
      \( \text{Mem}[\text{toaddress}, B, \text{wacctype}] = \text{readdata}_{B \times 8-1:0} \);
      \( \text{fromaddress} = \text{fromaddress} + B \);
      \( \text{toaddress} = \text{toaddress} + B \);
    else
      \( \text{readdata}_{B \times 8-1:0} = \text{Mem}[\text{fromaddress}-B, B, \text{racctype}] \);
      \( \text{Mem}[\text{toaddress}-B, B, \text{wacctype}] = \text{readdata}_{B \times 8-1:0} \);
      \( \text{fromaddress} = \text{fromaddress} - B \);
      \( \text{toaddress} = \text{toaddress} - B \);
    \( \text{cpysize} = \text{cpysize} - B \);
    \( \text{stagecpysize} = \text{stagecpysize} - B \);

    if stage \(!=\) \text{MOPSStage_Prologue} then
      \( X[n, 64] = \text{cpysize} \);
      \( X[d, 64] = \text{toaddress} \);
      \( X[s, 64] = \text{fromaddress} \);
    \text{PSTATE.<N,Z,C,V> = nzcv};
C6.2.93 CPYPTRN, CPYMTRN, CPYETRN

Memory Copy, reads and writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTRN, then CPYMTRN, and then CPYETRN.

CPYPTRN performs some preconditioning of the arguments suitable for using the CPYMTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYETRN performs the last part of the memory copy.

Note
The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPTRN, the following saturation logic is applied:
If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:
If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note
Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPTRN, option A (which results in encoding PSTATE.C = 0):
• PSTATE.{N,Z,V} are set to {0,0,0}.
• If the copy is in the forward direction, then:
  — Xs holds the original Xs + saturated Xn.
  — Xd holds the original Xd + saturated Xn.
  — Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
• If the copy is in the backward direction, then:
  — Xs and Xd are unchanged.
  — Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPTRN, option B (which results in encoding PSTATE.C = 1):
• If the copy is in the forward direction, then:
  — Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  — Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  — Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  — PSTATE.{N,Z,V} are set to {0,0,0}.
• If the copy is in the backward direction, then:
  — Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  — Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  — Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  — PSTATE.{N,Z,V} are set to {1,0,0}.
For CPYMTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds \(-1\times\) the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with \(-1\times\) the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYETRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds \(-1\times\) the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
— Xd holds the highest address that the copy is copied to -Xn+1.
— At the end of the instruction, the value of Xn is written back with 0.

For CPYETRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes to be copied in the memory copy in total.

• If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.

• If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

**Integer**

(FEAT_MOPS)

```
\[
\begin{array}{ccccccccccccccc}
\text{sz} & 0 & 1 & 1 & 1 & 0 & 1 & \text{op1} & 0 & \text{Rs} & 1 & 0 & 1 & 0 & 1 & \text{Rn} & \text{Rd} \\
\text{op2}
\end{array}
\]
```

**Epilogue variant**

Applies when op1 == 10.

CPYETRN [<Xd>], [<Xs>], <Xn>!

**Main variant**

Applies when op1 == 01.

CPYMTRN [<Xd>], [<Xs>], <Xn>!

**Prologue variant**

Applies when op1 == 00.

CPYPTRN [<Xd>], [<Xs>], <Xn>!

**Decode for all variants of this encoding**

```c
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
```
MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP   EndOfInstruction();

Assembler symbols

<Xd>
  For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
  For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs>
  For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
  For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn>
  For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
  For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
  For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

integer N = MaxBlockSizeCopiedBytes();

bits(64) toaddress = X[d, 64];

bits(64) fromaddress = X[s, 64];

bits(64) cpysize = X[n, 64];

bits(4) nzcv = PSTATE.<N,Z,C,V>;

bits(64) stagelcpysize;

bits(8*N) readdata;

integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();

(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
  if cpysize<63:55> != '00000001' then cpysize = 0x007FFFFFFFFFFFF<63:0>;

  boolean forward;
  if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) <
    UInt(toaddress<55:0> + cpysize<55:0>)) then
    forward = TRUE;
elsif (UInt(fromaddress<55:0>) < UInt(toaddress<55:0>) ) 
& ( UInt(fromaddress<55:0>) + cpysize<55:0> ) > UInt(toaddress<55:0>) )
then
    forward = FALSE;
else
    forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports_option_a then
    nzcv = '0000';
    if forward then
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';
    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);
    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
    else
        boolean zero_size_exceptions = MemCpyZeroSizeCheck();

        // Check if this version is consistent with the state of the call.
        if zero_size_exceptions || SInt(cpysize) != 0 then
            if supports_option_a then
                if nzcv<1> == '1' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                else
                    if nzcv<1> == '0' then // PSTATE.C
                        boolean wrong_option = TRUE;
                        boolean from_epilogue = stage == MOPSStage_Epilogue;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
            assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
            if stage == MOPSStage_Main then
                stagecpysize = cpysize - postsize;
                // Check if the parameters to this instruction are valid.
                if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                    boolean wrong_option = FALSE;
                    boolean from_epilogue = FALSE;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                else
                    stagecpysize = postsize;
                    // Check if the parameters to the epilogue are valid.
                    if cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize) then
                        boolean wrong_option = FALSE;
                        boolean from_epilogue = TRUE;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                    else
                        boolean from_epilogue = FALSE;
if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSIZEChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSIZEChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);

                if nzcv<3> == '0' then // PSTATE.N
                    readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
                    Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
                    fromaddress = fromaddress + B;
                    toaddress = toaddress + B;
                else
                    readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
                    Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
                    fromaddress = fromaddress - B;
                    toaddress = toaddress - B;

                cpysize = cpysize - B;
                stagecpysize = stagecpysize - B;

                if stage != MOPSStage_Prologue then
                    X[n, 64] = cpysize;
                    X[d, 64] = toaddress;
                    X[s, 64] = fromaddress;
                else
                    if stage == MOPSStage_Prologue then
                        X[n, 64] = cpysize;
                        X[d, 64] = toaddress;
                        X[s, 64] = fromaddress;
                        PSTATE.<N,Z,C,V> = nzcv;
C6.2.94 CPYPTWN, CPYMTWN, CPYETWN

Memory Copy, reads and writes unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTWN, then CPYMTWN, and then CPYETWN.

CPYPTWN performs some preconditioning of the arguments suitable for using the CPYMTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYETWN performs the last part of the memory copy.

——— Note ————
The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

——— Note ————
For CPYPTWN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

——— Note ————
Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPTWN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPTWN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.
For CPYMTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYETWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
— Xd holds the highest address that the copy is copied to -Xn+1.
— At the end of the instruction, the value of Xn is written back with 0.

For CPYETWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

**Integer**

(FEAT_MOPS)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz 0 1 1 1 0 1</td>
<td>op1 0</td>
<td>Rs 0 1 1 1</td>
<td>0 1</td>
</tr>
</tbody>
</table>

**Epilogue variant**

Applies when op1 == 10.
CPYETWN [Xd]!, [Xs]!, <Xn>!

**Main variant**

Applies when op1 == 01.
CPYMTWN [Xd]!, [Xs]!, <Xn>!

**Prologue variant**

Applies when op1 == 00.
CPYPTWN [Xd]!, [Xs]!, <Xn>!

**Decode for all variants of this encoding**

```plaintext
define
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bv4 bits options = op2;
```
MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

\[
\begin{align*}
\text{integer } N &= \text{MaxBlockSizeCopiedBytes}(); \\
\text{bits}(64) \text{ toaddress} &= X[d, 64]; \\
\text{bits}(64) \text{ fromaddress} &= X[s, 64]; \\
\text{bits}(64) \text{ cpysize} &= X[n, 64]; \\
\text{bits}(4) \text{ nzcv} &= \text{PSTATE.<N,Z,C,V>}; \\
\text{bits}(64) \text{ stagecpysize}; \\
\text{bits}(8*N) \text{ readdata}; \\
\text{integer } B;
\end{align*}
\]

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
  if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;
  boolean forward;
  if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) <
      UInt(toaddress<55:0> + cpysize<55:0>))) then
    forward = TRUE;
elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>))) then
  forward = FALSE;
else
  forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports_option_a then
  nzcv = '0000';
  if forward then
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
  else
    if !forward then
      // Copy in the reverse direction offsets the arguments.
      toaddress = toaddress + cpysize;
      fromaddress = fromaddress + cpysize;
    nzcv = '1010';
    else
      nzcv = '0010';

  // IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

  if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
  else
    assert SInt(stagecpysize) >= SInt(cpysize);
  else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

  // Check if this version is consistent with the state of the call.
  if zero_size_exceptions || SInt(cpysize) != 0 then
    if supports_option_a then
      if nzcv<1> == '1' then // PSTATE.C
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        if nzcv<1> == '0' then // PSTATE.C
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

  bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
  assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

  if stage == MOPSStage_Main then
    stagecpysize = cpysize - postsize;

    // Check if the parameters to this instruction are valid.
    if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
      boolean wrong_option = FALSE;
      boolean from_epilogue = FALSE;
      MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
      stagecpysize = postsize;

    // Check if the parameters to the epilogue are valid.
    if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
      boolean wrong_option = FALSE;
      boolean from_epilogue = TRUE;
      MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpsize);
    if SInt(cpsize) < 0 then
      assert B <= -1 * SInt(stagecpysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpsize, B, racctype];
      Mem[toaddress+cpsize, B, wacctype] = readdata<B*8-1:0>;
      cpysize =cpsize + B;
      stagecpysize = stagecpysize + B;
    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
  end while
  if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
  else
    while UInt(stagecpysize) > 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = CPYSizeChoice(toaddress, fromaddress, cpsize);
      assert B <= UInt(stagecpysize);
      if nzcv<3> == '0' then // PSTATE.N
        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
      else
        readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
        Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress - B;
        toaddress = toaddress - B;
      endif
      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
      if stage != MOPSStage_Prologue then
        X[n, 64] = cpysize;
        X[d, 64] = toaddress;
        X[s, 64] = fromaddress;
      endif
      if stage == MOPSStage_Prologue then
        X[n, 64] = cpysize;
        X[d, 64] = toaddress;
        X[s, 64] = fromaddress;
        PSTATE.<N,Z,C,V> = nzcv;
  endif
  endif
endif
CPYPWN, CPYMWN, CPYEWN

Memory Copy, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWN, then CPYMWN, and then CPYEWN.

CPYPWN performs some preconditioning of the arguments suitable for using the CPYMWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWN performs the last part of the memory copy.

Note
The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWN, the following saturation logic is applied:
If $Xn_{63:55} \neq 000000000$, the copy size $Xn$ is saturated to $0x007FFFFFFFFFFFFF$.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:
If $(Xs > Xd) \& \& (Xd + \text{saturated } Xn) > Xs$, then direction = forward
Elsif $(Xs < Xd) \& \& (Xs + \text{saturated } Xn) > Xd$, then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note
Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWN, option A (which results in encoding PSTATE.C = 0):
- PSTATE.$\{N,Z,V\}$ are set to $\{0,0,0\}$.
- If the copy is in the forward direction, then:
  - $Xs$ holds the original $Xs + \text{saturated } Xn$.
  - $Xd$ holds the original $Xd + \text{saturated } Xn$.
  - $Xn$ holds $-1 \times \text{saturated } Xn + \text{an IMPLEMENTATION DEFINED number of bytes copied}$.
- If the copy is in the backward direction, then:
  - $Xs$ and $Xd$ are unchanged.
  - $Xn$ holds the saturated value of $Xn - \text{an IMPLEMENTATION DEFINED number of bytes copied}$.

After execution of CPYPWN, option B (which results in encoding PSTATE.C = 1):
- If the copy is in the forward direction, then:
  - $Xs$ holds the original $Xs + \text{an IMPLEMENTATION DEFINED number of bytes copied}$.
  - $Xd$ holds the original $Xd + \text{an IMPLEMENTATION DEFINED number of bytes copied}$.
  - $Xn$ holds the saturated $Xn - \text{an IMPLEMENTATION DEFINED number of bytes copied}$.
  - PSTATE.$\{N,Z,V\}$ are set to $\{0,0,0\}$.
- If the copy is in the backward direction, then:
  - $Xs$ holds the original $Xs + \text{saturated } Xn - \text{an IMPLEMENTATION DEFINED number of bytes copied}$.
  - $Xd$ holds the original $Xd + \text{saturated } Xn - \text{an IMPLEMENTATION DEFINED number of bytes copied}$.
  - $Xn$ holds the saturated $Xn - \text{an IMPLEMENTATION DEFINED number of bytes copied}$.
  - PSTATE.$\{N,Z,V\}$ are set to $\{1,0,0\}$. 
For CPYMWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
— Xd holds the highest address that the copy is copied to -Xn+1.
— At the end of the instruction, the value of Xn is written back with 0.

For CPYEWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

**Integer**

(FEAT_MOPS)

| [31 30 29 28 |27 26 25 24|23 22 21 20] | 16|15 | 12|11 10 9 | 5 | 4 | 0 |
|---|---|---|---|---|---|---|---|---|
| sz | 0 | 1 | 1 | 1 | 0 | 1 | op1 | 0 |
| Rs | 0 | 1 | 0 | 0 | 0 | 1 | Rn | Rd |
| op2 | |

**Epilogue variant**

Applies when op1 == 10.

CPYEWN [Xd]!, [Xs]!, <Xn>!

**Main variant**

Applies when op1 == 01.

CPYMWN [Xd]!, [Xs]!, <Xn>!

**Prologue variant**

Applies when op1 == 00.

CPYPWN [Xd]!, [Xs]!, <Xn>!

**Decode for all variants of this encoding**

```c
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
```
MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();
if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
assert c IN {Constraint_UNDEF, Constraint_NOP};
case c of
  when Constraint_UNDEF UNDEFINED;
  when Constraint_NOP   EndOfInstruction();

Assembler symbols
<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);
boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);
if stage == MOPSStage_Prologue then
  if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;

  boolean forward;
  if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) <
      UInt(toaddress<55:0> + cpysize<55:0>)) then
    forward = TRUE;
elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>))) then
    forward = FALSE;
else
    forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports_option_a then
    nzcv = '0000';
    if forward then
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else if !forward then
        // Copy in the reverse direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        nzcv = '1010';
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
    else
        boolean zero_size_exceptions = MemCpyZeroSizeCheck();

        // Check if this version is consistent with the state of the call.
        if zero_size_exceptions || SInt(cpysize) != 0 then
            if supports_option_a then
                if nzcv<1> == '1' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                else
                    if nzcv<1> == '0' then // PSTATE.C
                        boolean wrong_option = TRUE;
                        boolean from_epilogue = stage == MOPSStage_Epilogue;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
                assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

                if stage == MOPSStage_Main then
                    stagecpysize = cpysize - postsize;

                    // Check if the parameters to this instruction are valid.
                    if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                        boolean wrong_option = FALSE;
                        boolean from_epilogue = FALSE;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                    else
                        stagecpysize = postsize;

                    // Check if the parameters to the epilogue are valid.
                    if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                        boolean wrong_option = FALSE;
                        boolean from_epilogue = TRUE;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                else
                    stagecpysize = cpysize;

                // Check if the parameters to the epilogue are valid.
                if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                    boolean wrong_option = FALSE;
                    boolean from_epilogue = TRUE;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSSizeChoice(toaddress, fromaddress, cpsize);
    if SInt(cpsize) < 0 then
      assert B <= -1 * SInt(stagecpysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
      cpysize = cpysize + B;
    stagecpysize = stagecpysize + B;
    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;
    readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
    Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
  end while
  if stage != MOPSStage_Prologue then
    X[n, 64] = cpsize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
  else
    while UInt(stagecpysize) > 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = CPYSSizeChoice(toaddress, fromaddress, cpsize);
      if nzcv<3> == '0' then // PSTATE.N
        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
      else
        readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
        Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress - B;
        toaddress = toaddress - B;
      cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;
    if stage != MOPSStage_Prologue then
      X[n, 64] = cpsize;
      X[d, 64] = toaddress;
      X[s, 64] = fromaddress;
    else
      if stage == MOPSStage_Prologue then
        X[n, 64] = cpsize;
        X[d, 64] = toaddress;
        X[s, 64] = fromaddress;
        PSTATE.<N,Z,C,V> = nzcv;
C6.2.96 CPYPWT, CPYMWT, CPYEWT

Memory Copy, writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWT, then CPYMWT, and then CPYEWT.

CPYPWT performs some preconditioning of the arguments suitable for using the CPYMWT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWT performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWT, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWT, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWT, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.
For CPYMWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
Xd holds the highest address that the copy is copied to -Xn+1.
At the end of the instruction, the value of Xn is written back with 0.

For CPYEWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

**Integer**

(FEAT_MOPS)

| [31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 | 12|11 10 9 | 5 4 | 0 |
|----------|---|---|---|---|---|---|---|
| op2 0 | 1 1 | 0 1 | 0 | Rs | 0 | 0 | 1 |
| Rd | | | | | | | Rn |

**Epilogue variant**

Applies when op1 == 10.

CPYEW [<Xd}], [<Xs]>, <Xn>!

**Main variant**

Applies when op1 == 01.

CPYMWT [<Xd>]!, [<Xs>]!, <Xn>!

**Prologue variant**

Applies when op1 == 00.

CPYPWT [<Xd>]!, [<Xs>]!, <Xn>!

**Decode for all variants of this encoding**

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bites(4) options = op2;
MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";
CheckMOPSEnabled();
if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
case c of
  when Constraint_UNDEF UNDEFINED;
  when Constraint_NOP   EndOfInstruction();
Assembler symbols
<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
Operation
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);
boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);
if stage == MOPSStage_Prologue then
  if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFF<63:0>;
  boolean forward;
  if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55:0> + cpysize<55:0>))) then
    forward = TRUE;
elsif (( UInt(fromaddress<55:0>) < UInt(toaddress<55:0>) )
  && ( UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>))) then
  forward = FALSE;
else
  forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports_option_a then
  nzcv = '0000';
  if forward then
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
  else
    if !forward then
      // Copy in the reverse direction offsets the arguments.
      toaddress = toaddress + cpysize;
      fromaddress = fromaddress + cpysize;
      nzcv = '1010';
    else
      nzcv = '0010';
    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);
  if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
  else
    assert SInt(stagecpysize) >= SInt(cpysize);
  else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
      if supports_option_a then
        if nzcv<1> == '1' then // PSTATE.C
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
          if nzcv<1> == '0' then // PSTATE.C
            boolean wrong_option = TRUE;
            boolean from_epilogue = stage == MOPSStage_Epilogue;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
          else
            bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
            assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
        if stage == MOPSStage_Main then
          stagecpysize = cpysize - postsize;
          // Check if the parameters to this instruction are valid.
          if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
          else
            stagecpysize = postsize;
            // Check if the parameters to the epilogue are valid.
            if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
              boolean wrong_option = FALSE;
              boolean from_epilogue = TRUE;
              MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);  
        end if
    else
      stagecpysize = postsize;
      // Check if the parameters to the epilogue are valid.
      if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = TRUE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      end if
    end if
  end if
end if

end if

else
  forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports_option_a then
  nzcv = '0000';
  if forward then
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
  else
    if !forward then
      // Copy in the reverse direction offsets the arguments.
      toaddress = toaddress + cpysize;
      fromaddress = fromaddress + cpysize;
      nzcv = '1010';
    else
      nzcv = '0010';
    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);
  if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
  else
    assert SInt(stagecpysize) >= SInt(cpysize);
  else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
      if supports_option_a then
        if nzcv<1> == '1' then // PSTATE.C
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
          if nzcv<1> == '0' then // PSTATE.C
            boolean wrong_option = TRUE;
            boolean from_epilogue = stage == MOPSStage_Epilogue;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
          else
            bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
            assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
        if stage == MOPSStage_Main then
          stagecpysize = cpysize - postsize;
          // Check if the parameters to this instruction are valid.
          if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
          else
            stagecpysize = postsize;
            // Check if the parameters to the epilogue are valid.
            if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
              boolean wrong_option = FALSE;
              boolean from_epilogue = TRUE;
              MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
          end if
    else
      stagecpysize = postsize;
      // Check if the parameters to the epilogue are valid.
      if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = TRUE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      end if
    end if
  end if
end if

end if
if supports_option_a then
  while $\mathsf{SInt}(\text{stagecpysize}) \neq 0$ do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    $B = \mathsf{CPYSizeChoice}(\text{toaddress}, \text{fromaddress}, \text{cpysize})$;
    
    if $\mathsf{SInt}(\text{cpysize}) < 0$ then
      assert $B \leq -1 \times \mathsf{SInt}(\text{stagecpysize})$;
      $\text{readdata}_{<B*8-1:0>} = \mathsf{Mem}[\text{fromaddress}+\text{cpysize}, B, \text{racctype}]$;
      $\mathsf{Mem}[\text{toaddress}+\text{cpysize}, B, \text{wacctype}] = \text{readdata}_{<B*8-1:0>} + B$;
      $\text{stagecpysize} = \text{stagecpysize} + B$;
    else
      assert $B \leq \mathsf{SInt}(\text{stagecpysize})$;
      $\text{cpysize} = \text{cpysize} - B$;
      $\text{stagecpysize} = \text{stagecpysize} - B$;
      $\text{readdata}_{<B*8-1:0>} = \mathsf{Mem}[\text{fromaddress}+\text{cpysize}, B, \text{racctype}]$;
      $\mathsf{Mem}[\text{toaddress}+\text{cpysize}, B, \text{wacctype}] = \text{readdata}_{<B*8-1:0>} + B$;

    $\text{stagecpysize} = \text{stagecpysize} + B$;
  end

if $\text{stage} = \text{MOPSStage-Prologue}$ then
  $X[n, 64] = \text{cpysize}$;
else
  while $\mathsf{UInt}(\text{stagecpysize}) > 0$ do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    $B = \mathsf{CPYSizeChoice}(\text{toaddress}, \text{fromaddress}, \text{cpysize})$;
    $\text{assert } B \leq \mathsf{UInt}(\text{stagecpysize})$;
    $\text{if } \text{nzcv}_{<3>} == '0' \text{ then } // \text{PSTATE}.N$
    $\text{readdata}_{<B*8-1:0>} = \mathsf{Mem}[\text{fromaddress}, B, \text{racctype}]$;
    $\mathsf{Mem}[\text{toaddress}, B, \text{wacctype}] = \text{readdata}_{<B*8-1:0>} + B$;
    $\text{fromaddress} = \text{fromaddress} + B$;
    $\text{toaddress} = \text{toaddress} + B$;
  end

  if $\text{stage} = \text{MOPSStage-Prologue}$ then
    $X[n, 64] = \text{cpysize}$;
    $X[d, 64] = \text{toaddress}$;
    $X[s, 64] = \text{fromaddress}$;
  end

if $\text{stage} = \text{MOPSStage-Prologue}$ then
  $X[n, 64] = \text{cpysize}$;
  $X[d, 64] = \text{toaddress}$;
  $X[s, 64] = \text{fromaddress}$;
  $\text{PSTATE}.<N,Z,C,V> = \text{nzcv}$;
C6.2.97  CPYPWTN, CPYMWTN, CPYEWTN

Memory Copy, writes unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTN, then CPYMWTN, and then CPYEWTN.

CPYPWTN performs some preconditioning of the arguments suitable for using the CPYMWTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWTN performs the last part of the memory copy.

--- Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

---

For CPYPWTN, the following saturation logic is applied:

If \(Xn_{63:55} \neq 000000000\), the copy size \(Xn\) is saturated to \(0x007FFFFFFFFFFFF\).

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

- If \((Xs > Xd) && (Xd + saturated Xn) > Xs\), then direction = forward
- Elsif \((Xs < Xd) && (Xs + saturated Xn) > Xd\), then direction = backward
- Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

--- Note

Portable software should not assume that the choice of algorithm is constant.

---

After execution of CPYPWTN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to \{0,0,0\}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds \(-1^*\) saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWTN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to \{0,0,0\}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to \{1,0,0\}.
For CPYMWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds \(-1\times\) the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with \(-1\times\) the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds \(-1\times\) the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
— Xd holds the highest address that the copy is copied to -Xn+1.
— At the end of the instruction, the value of Xn is written back with 0.

For CPYEWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.
• If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

Integer (FEAT_MOPS)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz 0 1 1 1 0 1 op1 0</td>
<td>Rs 1 1 0 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>op2</td>
<td>Rn Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Epilogue variant**

Applies when op1 == 10.
CPYEWTN [Xd]!, [Xs]!, <Xn>!

**Main variant**

Applies when op1 == 01.
CPYMWTN [Xd]!, [Xs]!, <Xn>!

**Prologue variant**

Applies when op1 == 00.
CPYPWTN [Xd]!, [Xs]!, <Xn>!

**Decode for all variants of this encoding**

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";
endcase;
CheckMOPSEnabled();
if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
c = ConstrLimitUnpredictable(Unpredictable_MOPSOVERLAP31);
assert c IN {Constraint_UNDEF, Constraint_NOP};
case c of
  when Constraint_UNDEF UNDEFINED;
  when Constraint_NOP   EndOfInstruction();
endcase;
Assembler symbols
<Xd>
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs>
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn>
For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);
boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);
if stage == MOPSSStage_Prologue then
  if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFF<63:0>;
endcase;
boolean forward;
if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55:0> + cpysize<55:0>))) then
  forward = TRUE;
elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) \&\&(UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>))) then
  forward = FALSE;
else
  forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports_option_a then
  nzcv = '0000';
  if forward then
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
  else
    // Copy in the reverse direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    nzcv = '1010';
  else
    nzcv = '0010';

  // IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

  if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
  else
    assert SInt(stagecpysize) >= SInt(cpysize);
  else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
      if supports_option_a then
        if nzcv<1> == '1' then // PSTATE.C
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
          if nzcv<1> == '0' then // PSTATE.C
            boolean wrong_option = TRUE;
            boolean from_epilogue = stage == MOPSStage_Epilogue;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
        assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
      else
        stagecpysize = postsize;
      // Check if the parameters to the epilogue are valid.
      if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = TRUE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      end
    stagecpysize = postsize;
  end

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  end

  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  end
if supports_option_a then
    while $\text{Int}(\text{stagecpysize}) 
        \text{!}= 0 \text{ do} \\
        // IMP DEF selection of the block size that is worked on. While many 
        // implementations might make this constant, that is not assumed.
        B = \text{CPYSizEChoice}(\text{toaddress}, \text{fromaddress}, \text{cpysize});
    
    if \text{Int}(\text{cpysize}) < 0 then
        assert B \text{ <= -1 \text{Int}(\text{stagecpysize})};
        \text{readdata}_{B*8-1:0} = \text{Mem}[\text{fromaddress}+\text{cpysize}, B, \text{racctype}];
        \text{Mem}[\text{toaddress}+\text{cpysize}, B, \text{wacctype}] = \text{readdata}_{B*8-1:0};
        \text{cpysize} = \text{cpysize} + B;
        \text{stagecpysize} = \text{stagecpysize} + B;
    else
        assert B \text{ <= \text{Int}(\text{stagecpysize})};
        \text{cpysize} = \text{cpysize} - B;
        \text{stagecpysize} = \text{stagecpysize} - B;
        \text{readdata}_{B*8-1:0} = \text{Mem}[\text{fromaddress}+\text{cpysize}, B, \text{racctype}];
        \text{Mem}[\text{toaddress}+\text{cpysize}, B, \text{wacctype}] = \text{readdata}_{B*8-1:0};
    
    if \text{stage} \text{!}= \text{MOPSStage Prologue} \text{ then}
        X[n, 64] = \text{cpysize};
    else
        while $\text{UInt}(\text{stagecpysize}) > 0 \text{ do} \\
        // IMP DEF selection of the block size that is worked on. While many 
        // implementations might make this constant, that is not assumed.
        B = \text{CPYSizEChoice}(\text{toaddress}, \text{fromaddress}, \text{cpysize});
        assert B \text{ <= \text{UInt}(\text{stagecpysize})};
    
    if nzcv<3> \text{ == '0'} then // PSTATE.N
        \text{readdata}_{B*8-1:0} = \text{Mem}[\text{fromaddress}, B, \text{racctype}];
        \text{Mem}[\text{toaddress}, B, \text{wacctype}] = \text{readdata}_{B*8-1:0};
        \text{fromaddress} = \text{fromaddress} + B;
        \text{toaddress} = \text{toaddress} + B;
    else
        \text{readdata}_{B*8-1:0} = \text{Mem}[\text{fromaddress}-B, B, \text{racctype}];
        \text{Mem}[\text{toaddress}-B, B, \text{wacctype}] = \text{readdata}_{B*8-1:0};
        \text{fromaddress} = \text{fromaddress} - B;
        \text{toaddress} = \text{toaddress} - B;
    
    \text{cpysize} = \text{cpysize} - B;
    \text{stagecpysize} = \text{stagecpysize} - B;
    
    if \text{stage} \text{!}= \text{MOPSStage Prologue} \text{ then}
        X[n, 64] = \text{cpysize};
        X[d, 64] = \text{toaddress};
        X[s, 64] = \text{fromaddress};
    
    if \text{stage} \text{== MOPSStage Prologue} \text{ then}
        X[n, 64] = \text{cpysize};
        X[d, 64] = \text{toaddress};
        X[s, 64] = \text{fromaddress};
        \text{PSTATE}.<N,Z,C,V> = \text{nzc}v;
C6.2.98 CPYPWTRN, CPYMWTRN, CPYEWTRN

Memory Copy, writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTRN, then CPYMWTRN, and then CPYEWTRN.

CPYPWTRN performs some preconditioning of the arguments suitable for using the CPYMWTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWTRN performs the last part of the memory copy.

Note
The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWTRN, the following saturation logic is applied:
If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.
After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:
If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.
The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note
Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWTRN, option A (which results in encoding PSTATE.C = 0):
- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWTRN, option B (which results in encoding PSTATE.C = 1):
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.
For CPYMWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds \(-1\times\) the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with \(-1\times\) the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds \(-1\times\) the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
— Xd holds the highest address that the copy is copied to -Xn+1.
— At the end of the instruction, the value of Xn is written back with 0.

For CPYEWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.
• If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

Integer
(FEAT_MOPS)

Epilogue variant
Applies when op1 == 10.
CPYWTRN [<Xd>], [<Xs>], <Xn>!

Main variant
Applies when op1 == 01.
CPYMWTN [<Xd>], [<Xs>], <Xn>!

Prologue variant
Applies when op1 == 00.
CPYPWTRN [<Xd>], [<Xs>], <Xn>!

Decode for all variants of this encoding
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
MOPSSStage stage;
    case op1 of
        when '00' stage = MOPSSStage_Prologue;
        when '01' stage = MOPSSStage_Main;
        when '10' stage = MOPSSStage_Epilogue;
        otherwise SEE "Memory Copy and Memory Set";
    CheckMOPSEnabled();

    if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
        c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP   EndOfInstruction();
    end;

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);
boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);
if stage == MOPSSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;
    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) & (UInt(fromaddress<55:0>) <
        UInt(toaddress<55:0> + cpysize<55:0>))) then
        forward = TRUE;
elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) \&\& (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>))) then
    forward = FALSE;
else
    forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports_option_a then
    nzcv = '0000';
    if forward then
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';
    endif
endif

// IMP DEF selection of the amount covered by pre-processing.
stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);
if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
else
    assert SInt(stagecpysize) >= SInt(cpysize);
endif
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if nzcv<1> == '0' then // PSTATE.C
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
            assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
            if stage == MOPSStage_Main then
                stagecpysize = cpysize - postsize;
                // Check if the parameters to this instruction are valid.
                if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                    boolean wrong_option = FALSE;
                    boolean from_epilogue = FALSE;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                else
                    stagecpysize = postsize;
                    // Check if the parameters to the epilogue are valid.
                    if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
                        boolean wrong_option = FALSE;
                        boolean from_epilogue = TRUE;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                endif
            endif
        endif
    endif
endif
if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizelChoice(toaddress, fromaddress, cpysize);
    if SInt(cpysize) < 0 then
      assert B <= -1 * SInt(stagecpysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
      cpysize = cpysize + B;
      stagecpysize = stagecpysize + B;
    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
    end if
  end while
  if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
  end if
end if

while UInt(stagecpysize) > 0 do
  // IMP DEF selection of the block size that is worked on. While many
  // implementations might make this constant, that is not assumed.
  B = CPYSizelChoice(toaddress, fromaddress, cpysize);
  assert B <= UInt(stagecpysize);
  if nzcv<3> == '0' then // PSTATE.N
    readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
    Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
  else
    readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
    Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
    fromaddress = fromaddress - B;
    toaddress = toaddress - B;
  end if
  cpysize = cpysize - B;
  stagecpysize = stagecpysize - B;
  if stage != MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
  end if
end while
if stage == MOPSStage_Prologue then
  X[n, 64] = cpysize;
  X[d, 64] = toaddress;
  X[s, 64] = fromaddress;
  PSTATE.<N,Z,C,V> = nzcv;
C6.2.99 CPYPWTWN, CPYMWTWN, CPYEWTWN

Memory Copy, writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTWN, then CPYMWTWN, and then CPYEWTWN.

CPYPWTWN performs some preconditioning of the arguments suitable for using the CPYMWTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWTWN performs the last part of the memory copy.

Note
The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWTWN, the following saturation logic is applied:
If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:
If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note
Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWTWN, option A (which results in encoding PSTATE.C = 0):
• PSTATE.{N,Z,V} are set to {0,0,0}.
• If the copy is in the forward direction, then:
  — Xs holds the original Xs + saturated Xn.
  — Xd holds the original Xd + saturated Xn.
  — Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
• If the copy is in the backward direction, then:
  — Xs and Xd are unchanged.
  — Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWTWN, option B (which results in encoding PSTATE.C = 1):
• If the copy is in the forward direction, then:
  — Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  — Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  — Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  — PSTATE.{N,Z,V} are set to {0,0,0}.
• If the copy is in the backward direction, then:
  — Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  — Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  — Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  — PSTATE.{N,Z,V} are set to {1,0,0}.
For CPYMWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
— Xd holds the highest address that the copy is copied to -Xn+1.
— At the end of the instruction, the value of Xn is written back with 0.

For CPYEWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

### Integer

(FEAT_MOPS)

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 12|11 10 9 | 5 4 | 0 |
   sz  0  1  1 | 0  1 | op1 | 0 | Rs  0  1  0 | 1  | Rn  Rd
```

### Epilogue variant

Applies when op1 == 10.

CPYEWTWN [<Xd>]!, [<Xs>]!, <Xn>!

### Main variant

Applies when op1 == 01.

CPYMWTWN [<Xd>]!, [<Xs>]!, <Xn>!

### Prologue variant

Applies when op1 == 00.

CPYPWTWN [<Xd>]!, [<Xs>]!, <Xn>!

### Decode for all variants of this encoding

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bites(4) options = op2;
```
MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";
EndOfInstruction();

Assembler symbols

<Xd>
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an
encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination
address and is updated by the instruction, encoded in the "Rd" field.

<Xs>
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an
encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source
address and is updated by the instruction, encoded in the "Rs" field.

<Xn>
For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding
of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in
the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of
the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of
bytes to be transferred and is updated by the instruction to encode the remaining size and
destination, encoded in the "Rn" field.

Operation

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);
if stage == MOPSStage_Prologue then
  if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;

boolean forward;
if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) & (UInt(toaddress<55:0>) + cpysize<55:0>) then
  forward = TRUE;
elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>))) then
  forward = FALSE;
else
  forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports_option_a then
  nzcv = '0000';
  if forward then
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
  else
    if !forward then
      // Copy in the reverse direction offsets the arguments.
      toaddress = toaddress + cpysize;
      fromaddress = fromaddress + cpysize;
      nzcv = '1010';
    else
      nzcv = '0010';
  // IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);
  if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
  else
    assert SInt(stagecpysize) >= SInt(cpysize);
  else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

  if zero_size_exceptions || SInt(cpysize) != 0 then
    if supports_option_a then
      if nzcv<1> == '1' then // PSTATE.C
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        if nzcv<1> == '0' then // PSTATE.C
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
  assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

  if stage == MOPSStage_Main then
    stagecpysize = cpysize - postsize;

    // Check if the parameters to this instruction are valid.
    if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
      boolean wrong_option = FALSE;
      boolean from_epilogue = FALSE;
      MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
      stagecpysize = postsize;

      // Check if the parameters to the epilogue are valid.
      if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = TRUE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);

    if SInt(cpysize) < 0 then
      assert B <= -1 * SInt(stagecpysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
      cpysize = cpysize + B;
      stagecpysize = stagecpysize + B;
    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

    if stage != MOPSStage_Prologue then
      X[n, 64] = cpysize;
    else
      while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
          readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
          Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
          fromaddress = fromaddress + B;
          toaddress = toaddress + B;
        else
          readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
          Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
          fromaddress = fromaddress - B;
          toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
          X[n, 64] = cpysize;
          X[d, 64] = toaddress;
          X[s, 64] = fromaddress;
        if stage == MOPSStage_Prologue then
          X[n, 64] = cpysize;
          X[d, 64] = toaddress;
          X[s, 64] = fromaddress;
          PSTATE.<N,Z,C,V> = nzcv;
C6.2.100  CRC32B, CRC32H, CRC32W, CRC32X

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x04C11DB7 is used for the CRC calculation.

In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note: ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

CRC

(FEAT_CRC32)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 5 | 4 | 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| sf  | Rm  | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | sz | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

CRC32B variant

Applies when sf == 0 && sz == 00.

CRC32B <Wd>, <Wn>, <Wm>

CRC32H variant

Applies when sf == 0 && sz == 01.

CRC32H <Wd>, <Wm>, <Wm>

CRC32W variant

Applies when sf == 0 && sz == 10.

CRC32w <Wd>, <Wm>, <Wm>

CRC32X variant

Applies when sf == 1 && sz == 11.

CRC32X <Wd>, <Xm>, <Xm>

Decode for all variants of this encoding

if !HaveCRCExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.

<Xm> Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.

<Wm> Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

**Operation**

```
bits(32) acc = X[n, 32]; // accumulator
bits(size) val = X[m, size]; // input value
bits(32) poly = 0x04C11DB7<31:0>;

bits(32+size) tempacc = BitReverse(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d, 32] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.101 CRC32CB, CRC32CH, CRC32CW, CRC32CX

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x1EDC6F41 is used for the CRC calculation.

In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

--- Note ---

ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

**CRC**

(FEAT_CRC32)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf 0 0 1 1 0 1 0 1 1 0</td>
<td>Rm 0 1 0 1</td>
<td>sz</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CRC32CB variant**

Applies when sf = 0 && sz = 00.

CRC32CB <Wd>, <Wn>, <Wm>

**CRC32CH variant**

Applies when sf = 0 && sz = 01.

CRC32CH <Wd>, <Wn>, <Wm>

**CRC32CW variant**

Applies when sf = 0 && sz = 10.

CRC32CW <Wd>, <Wn>, <Wm>

**CRC32CX variant**

Applies when sf = 1 && sz = 11.

CRC32CX <Wd>, <Wn>, <Xm>

**Decode for all variants of this encoding**

```c
if !HaveCRCExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);
```

**Assembler symbols**

<Wd> Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
$<Xm>$ Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.

$<Wm>$ Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

**Operation**

\[
\begin{align*}
\text{bits(32)} \ acc &= X[n, 32]; \quad \text{// accumulator} \\
\text{bits(size)} \ val &= X[m, \text{size}]; \quad \text{// input value} \\
\text{bits(32)} \ poly &= 0x1EDC6F41<31:0>;
\end{align*}
\]

\[
\begin{align*}
\text{bits(32+size)} \ tempacc &= \text{BitReverse}(acc):\text{Zeros(size)}; \\
\text{bits(size+32)} \ tempval &= \text{BitReverse}(val):\text{Zeros}(32);
\end{align*}
\]

// Poly32Mod2 on a bitstring does a polynomial Modulus over \{0,1\} operation
\[
X[d, 32] = \text{BitReverse}(\text{Poly32Mod2}(\text{tempacc} \text{ EOR} \text{tempval}, poly));
\]

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.102 CSDB

Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution and data value prediction.

No instruction other than branch instructions appearing in program order after the CSDB can be speculatively executed using the results of any:

- Data value predictions of any instructions.
- PSTATE.{N,Z,C,V} predictions of any instructions other than conditional branch instructions appearing in program order before the CSDB that have not been architecturally resolved.
- Predictions of SVE predication state for any SVE instructions.

--- Note ---

For purposes of the definition of CSDB, PSTATE.{N,Z,C,V} is not considered a data value. This definition permits:

- Control flow speculation before and after the CSDB.
- Speculative execution of conditional data processing instructions after the CSDB, unless they use the results of data value or PSTATE.{N,Z,C,V} predictions of instructions appearing in program order before the CSDB that have not been architecturally resolved.

---

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11  8  7  5  4  3  2  1  0 ]
[11 10 1 0 1 0 1 0 0 0 0 1 1 0 0 1 0 0 0 1 0 1 0 1 1 1 1 1]
```

CRm op2

**Encoding**

CSDB

**Decode for this encoding**

// Empty.

**Operation**

`ConsumptionOfSpeculativeDataBarrier();`
C6.2.103 CSEL

If the condition is true, Conditional Select writes the value of the first source register to the destination register. If the condition is false, it writes the value of the second source register to the destination register.

32-bit variant
Applies when sf == 0.
CSEL <Wd>, <Wn>, <Wm>, <cond>

64-bit variant
Applies when sf == 1.
CSEL <Xd>, <Xn>, < Xm>, <cond>

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

bits(datasize) result;
if ConditionHolds(cond) then
  result = X[n, datasize];
else
  result = X[m, datasize];
X[d, datasize] = result;
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.104 CSET

Conditional Set sets the destination register to 1 if the condition is TRUE, and otherwise sets it to 0.

This instruction is an alias of the CSINC instruction. This means that:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode for this instruction.

32-bit variant

Applies when \( sf = 0 \).

\[
\text{CSET } \langle Wd \rangle, <\text{cond}> \\
\text{is equivalent to} \\
\text{CSINC } \langle Wd \rangle, \text{WZR, WZR, invert(<cond>)} \\
\text{and is always the preferred disassembly.}
\]

64-bit variant

Applies when \( sf = 1 \).

\[
\text{CSET } \langle Xd \rangle, <\text{cond}> \\
\text{is equivalent to} \\
\text{CSINC } \langle Xd \rangle, \text{XZR, XZR, invert(<cond>)} \\
\text{and is always the preferred disassembly.}
\]

Assembler symbols

\( \langle Wd \rangle \)

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

\( \langle Xd \rangle \)

Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

\( <\text{cond}> \)

Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSINC gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.105 CSETM

Conditional Set Mask sets all bits of the destination register to 1 if the condition is TRUE, and otherwise sets all bits to 0.

This instruction is an alias of the CSINV instruction. This means that:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when \( sf = 0 \).

\[
\text{CSETM } <Wd>, <\text{cond}>
\]

is equivalent to

\[
\text{CSINV } <Wd>, \text{WZR, WZR, invert(<}\text{cond}>)
\]

and is always the preferred disassembly.

### 64-bit variant

Applies when \( sf = 1 \).

\[
\text{CSETM } <Xd>, <\text{cond}>
\]

is equivalent to

\[
\text{CSINV } <Xd>, \text{XZR, XZR, invert(<}\text{cond}>)
\]

and is always the preferred disassembly.

### Assembler symbols

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<\text{cond}>\) Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

### Operation

The description of CSINV gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
C6.2.106  CSINC

Conditional Select Increment returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register incremented by 1.

This instruction is used by the aliases CINC and CSET. See Alias conditions for details of when each alias is preferred.

32-bit variant
Applies when $\text{sf} == 0$.

$\text{CSINC} <\text{Wd}>, <\text{Wn}>, <\text{Wm}>, <\text{cond}>$

64-bit variant
Applies when $\text{sf} == 1$.

$\text{CSINC} <\text{Xd}>, <\text{Xn}>, <\text{Xm}>, <\text{cond}>$

Decode for all variants of this encoding

$$\begin{align*}
\text{integer } d &= \text{UInt}(\text{Rd}); \\
\text{integer } n &= \text{UInt}(\text{Rn}); \\
\text{integer } m &= \text{UInt}(\text{Rm}); \\
\text{integer } \text{datasize} &= \text{if } \text{sf} == '1' \text{ then } 64 \text{ else } 32;
\end{align*}$$

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CINC</td>
<td>$\text{Rm} \neq '11111'$ &amp;&amp; $\text{cond} \neq '111x'$ &amp;&amp; $\text{Rn} \neq '11111'$ &amp;&amp; $\text{Rn} == \text{Rm}$</td>
</tr>
<tr>
<td>CSET</td>
<td>$\text{Rm} == '11111'$ &amp;&amp; $\text{cond} \neq '111x'$ &amp;&amp; $\text{Rn} == '11111'$</td>
</tr>
</tbody>
</table>

Assembler symbols

- $<\text{Wd}>$ Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- $<\text{Wn}>$ Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- $<\text{Wm}>$ Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- $<\text{Xd}>$ Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- $<\text{Xn}>$ Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- $<\text{ Xm}>$ Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- $<\text{cond}>$ Is one of the standard conditions, encoded in the "cond" field in the standard way.
Operation

bits(datasize) result;
if ConditionHolds(cond) then
    result = X[n, datasize];
else
    result = X[m, datasize];
result = result + 1;
X[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.107 CSINV

Conditional Select Invert returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the bitwise inversion value of the second source register.

This instruction is used by the aliases CINV and CSETM. See Alias conditions for details of when each alias is preferred.

32-bit variant
Applies when $sf = 0$.

CSINV $<Wd>, <Wn>, <Wm>, <cond>

64-bit variant
Applies when $sf = 1$.

CSINV $<Xd>, <Xn>, <Xm>, <cond>

Decode for all variants of this encoding

integer $d = $UInt(Rd);
integer $n = $UInt(Rn);
integer $m = $UInt(Rm);
integer datasize = if $sf == '1'$ then 64 else 32;

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CINV</td>
<td>$Rm != '11111' &amp; &amp; cond != '111x' &amp; &amp; $Rn != '11111' &amp; &amp; $Rn == $Rm</td>
</tr>
<tr>
<td>CSETM</td>
<td>$Rn == '11111' &amp; &amp; cond != '111x' &amp; &amp; $Rn == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

$<Wd>$ Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

$<Wn>$ Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

$<Wm>$ Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

$<Xd>$ Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

$<Xn>$ Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

$<Xm>$ Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

$<cond>$ Is one of the standard conditions, encoded in the "cond" field in the standard way.
Operation

```plaintext
bits(datasize) result;
if ConditionHolds(cond) then
    result = X[n, datasize];
else
    result = X[m, datasize];
    result = NOT(result);
X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.108   CSNEG

Conditional Select Negation returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the negated value of the second source register.

This instruction is used by the alias CNEG. See Alias conditions for details of when each alias is preferred.

32-bit variant
Applies when sf == 0.

CSNEG <Wd>, <Wn>, <Wm>, <cond>

64-bit variant
Applies when sf == 1.

CSNEG <Xd>, <Xn>, <Xm>, <cond>

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNEG</td>
<td>cond !='111x' &amp; Rn == Rm</td>
</tr>
</tbody>
</table>

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

bits(datasize) result;
if ConditionHolds(cond) then
    result = X[n, datasize];
else
result = X[m, datasize];
result = NOT(result);
result = result + 1;
X[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.109   DC

Data Cache operation. For more information, see op0==0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, and BRBE instructions on page C5-659.

This instruction is an alias of the SYS instruction. This means that:

• The encodings in this description are named to match the encodings of SYS.
• The description of SYS gives the operational pseudocode for this instruction.

Encoding

DC <dc_op>, <Xt>

is equivalent to

SYS #<op1>, C7, <Cm>, #<op2>, <Xt>

and is the preferred disassembly when SysOp(op1,’0111’,CRm,op2) == Sys_DC.

Assembler symbols

<dc_op> Is a DC instruction name, as listed for the DC system instruction group, encoded in the "op1:CRm:op2" field. It can have the following values:

- IVAC when op1 = 000, CRm = 0110, op2 = 001
- ISW when op1 = 000, CRm = 0110, op2 = 010
- CSW when op1 = 000, CRm = 1010, op2 = 010
- CISW when op1 = 000, CRm = 1110, op2 = 010
- ZVA when op1 = 011, CRm = 0100, op2 = 001
- CVAC when op1 = 011, CRm = 0100, op2 = 001
- CVAU when op1 = 011, CRm = 1011, op2 = 001
- CIVAC when op1 = 011, CRm = 1110, op2 = 001

When FEAT_MTE2 is implemented, the following values are also valid:

- IGVAC when op1 = 000, CRm = 0110, op2 = 011
- IGWS when op1 = 000, CRm = 0110, op2 = 100
- IGDVAC when op1 = 000, CRm = 0110, op2 = 101
- IGDWS when op1 = 000, CRm = 0110, op2 = 110
- CGSW when op1 = 000, CRm = 1010, op2 = 100
- CGDSW when op1 = 000, CRm = 1010, op2 = 110
- CIGSW when op1 = 000, CRm = 1110, op2 = 100
- CIGDSW when op1 = 000, CRm = 1110, op2 = 110

When FEAT_MTE is implemented, the following values are also valid:

- GVA when op1 = 011, CRm = 0100, op2 = 011
- GZVA when op1 = 011, CRm = 0100, op2 = 100
- CGVAC when op1 = 011, CRm = 1010, op2 = 011
CGDVAC when op1 = 011, CRm = 1010, op2 = 101
CGVAP when op1 = 011, CRm = 1100, op2 = 011
CGDVAP when op1 = 011, CRm = 1100, op2 = 101
CGVADP when op1 = 011, CRm = 1101, op2 = 011
CGDVADP when op1 = 011, CRm = 1101, op2 = 101
CIGVAC when op1 = 011, CRm = 1110, op2 = 011
CIGDVAC when op1 = 011, CRm = 1110, op2 = 101

When FEAT_DPB is implemented, the following value is also valid:
CVAP when op1 = 011, CRm = 1100, op2 = 001

When FEAT_DPB2 is implemented, the following value is also valid:
CVADP when op1 = 011, CRm = 1101, op2 = 001

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation
The description of SYS gives the operational pseudocode for this instruction.
C6.2.110 DCPS1

Debug Change PE State to EL1, when executed in Debug state:

- If executed at EL0 changes the current Exception level and SP to EL1 using SP_EL1.
- Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS1 instruction is:

- EL1 if the instruction is executed at EL0.
- Otherwise, the Exception level at which the instruction is executed.

When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:

- ELR_ELx becomes UNKNOWN.
- SPSR_ELx becomes UNKNOWN.
- ESR_ELx becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_ELx.EE.

This instruction is undefined at EL0 in Non-secure state if EL2 is implemented and HCR_EL2.TGE == 1.
This instruction is always undefined in Non-debug state.

For more information on the operation of the DCPS<n> instructions, see DCPS<n> on page H2-10262.

| 31 30 29 28|27 26 25 24|23 22 21 20| 5 4| 3 2 1 0 |
|---|---|---|---|
| 1 1 0 1 0 1 0 0 1 0 1 | imm16 | 0 0 0 0 1 |

**Encoding**

DCPS1 {#<imm>}

**Decode for this encoding**

if !Halted() then UNDEFINED;

**Assembler symbols**

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

**Operation**

DCPSInstruction(LL);
C6.2.111  DCPS2

Debug Change PE State to EL2, when executed in Debug state:

- If executed at EL0 or EL1 changes the current Exception level and SP to EL2 using SP_EL2.
- Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS2 instruction is:

- EL2 if the instruction is executed at an exception level that is not EL3.
- EL3 if the instruction is executed at EL3.

When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:

- ELR_ELx becomes UNKNOWN.
- SPSR_ELx becomes UNKNOWN.
- ESR_ELx becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_ELx.EE.

This instruction is UNDEFINED at the following exception levels:

- All exception levels if EL2 is not implemented.
- At EL0 and EL1 if EL2 is disabled in the current Security state.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPS<n> instructions, see DCPS<n> on page H2-10262.

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>5 4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 0 1 1</td>
<td>imm16</td>
<td></td>
<td>0 0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

DCPS2 {#<imm>}

**Decode for this encoding**

if !Halted() then UNDEFINED;

**Assembler symbols**

<imm> is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

**Operation**

DCPSInstruction(LL);
C6.2.112 DCPS3

Debug Change PE State to EL3, when executed in Debug state:

- If executed at EL3 selects SP_EL3.
- Otherwise, changes the current Exception level and SP to EL3 using SP_EL3.

The target exception level of a DCPS3 instruction is EL3.

On executing a DCPS3 instruction:

- ELR_EL3 becomes UNKNOWN.
- SPSR_EL3 becomes UNKNOWN.
- ESR_EL3 becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_EL3.EE.

This instruction is UNDEFINED at all exception levels if either:

- ESCR.SDD == 1.
- EL3 is not implemented.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPS<\text{n}> instructions, see DCPS<\text{n}> on page H2-10262.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25</th>
<th>24</th>
<th>23 22 21 20</th>
<th>5 4</th>
<th>3 2 1 0</th>
<th>imm16</th>
<th>0 0 0</th>
<th>1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 0</td>
<td>1 0 1</td>
<td>imm16</td>
<td>0 0 0</td>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

DCPS3 {#<imm>}

Decode for this encoding

if !Halted() then UNDEFINED;

Assembler symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

DCPSInstruction(LL);
C6.2.113  DGH

Data Gathering Hint is a hint instruction that indicates that it is not expected to be performance optimal to merge memory accesses with Normal Non-cacheable or Device-GRE attributes appearing in program order before the hint instruction with any memory accesses appearing after the hint instruction into a single memory transaction on an interconnect.

System
(FEAT_DGH)

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 8 7 5 4 3 2 1 0]
 1 1 0 1 0 1 0 0 0 0 0 1 1 0 1 0 0 0 1 1 0 1 1 1 1
  CRm  op2
```

Encoding

DGH

Decode for this encoding

```c
if !HaveDGHExt() then EndOfInstruction();
```

Operation

```
Hint_DGH();
```
C6.2.114   DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see Data Memory Barrier (DMB) on page B2-175.

Encoding

DMB <option>|#<imm>

Decode for this encoding

MBReqDomain domain;
MBReqTypes types;
case CRm<3:2> of
  when '00' domain = MBReqDomain_OuterShareable;
  when '01' domain = MBReqDomain_NonShareable;
  when '10' domain = MBReqDomain_InnerShareable;
  when '11' domain = MBReqDomain_FullSystem;
case CRm<1:0> of
  when '00' types = MBReqTypes_All; domain = MBReqDomain_FullSystem;
  when '01' types = MBReqTypes_Reads;
  when '10' types = MBReqTypes_Writes;
  when '11' types = MBReqTypes_All;

Assembler symbols

<option> Specifies the limitation on the barrier operation. Values are:

SY     Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

ST     Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

LD     Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

ISH    Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.

ISHST  Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

ISHLD  Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

NSH    Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

NSHST  Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.

NSHLD  Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.
OSH  Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

OSHST Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

OSHLD Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm that are not listed above are reserved, and can be encoded using the #<imm> syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) on page B2-175 or see Data Synchronization Barrier (DSB) on page B2-178.

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

**Operation**

DataMemoryBarrier(domain, types);
C6.2.115 DRPS

Debug restore process state

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8|7 6 5 4|3 2 1 0 | 1 1 0 1 0 1 1 0 1 0 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0]
```

**Encoding**

DRPS

**Decode for this encoding**

```c
if !Halted() || PSTATE.EL == EL0 then UNDEFINED;
```

**Operation**

```c
DRPSInstruction();
```
C6.2.116  DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see *Data Synchronization Barrier (DSB)* on page B2-178.

A DSB instruction with the nXS qualifier is complete when the subset of these memory accesses with the XS attribute set to 0 are complete. It does not require that memory accesses with the XS attribute set to 1 are complete.

This instruction is used by the aliases PSSBB and SSBB. See *Alias conditions on page C6-1489* for details of when each alias is preferred.

**Memory barrier**

```
|31 30 29|28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11|8|7|6|5|4|3|2|1|0|
|1|1|0|1|0|1|0|0|0|0|0|1|1|0|0|1|1|1|1|
|CRm|1|0|0|1|1|1|1|1|
```

**Encoding**

DSB <option> | #<imm>

**Decode for this encoding**

boolean nXS = FALSE;

DSBAlias alias;
case CRm of
  when '0000' alias = DSBAlias_SSBB;
  when '0100' alias = DSBAlias_PSSBB;
  otherwise alias = DSBAlias_DSB;

MBReqDomain domain;
case CRm<3:2> of
  when '00' domain = MBReqDomain_OuterShareable;
  when '01' domain = MBReqDomain_Nonshareable;
  when '10' domain = MBReqDomain_InnerShareable;
  when '11' domain = MBReqDomain_FullSystem;

MBReqTypes types;
case CRm<1:0> of
  when '00' types = MBReqTypes_All; domain = MBReqDomain_FullSystem;
  when '01' types = MBReqTypes_Reads;
  when '10' types = MBReqTypes_Writes;
  when '11' types = MBReqTypes_All;

**Memory nXS barrier**

(FEAT_XS)

```
|31 30 29|28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11|10|9|8|7|6|5|4|3|2|1|0|imm2|1|0|0|0|1|1|1|1|1|1|
```

**Encoding**

DSB <option>nXS
Decode for this encoding

if !HaveFeatXS() then UNDEFINED;
MBReqTypes types = MBReqTypes.All;
boolean nXS = TRUE;
DSBAlias alias = DSBAlias.DSB;
MBReqDomain domain;

case imm2 of
    when '00' domain = MBReqDomain.OuterShareable;
    when '01' domain = MBReqDomain.Nonshareable;
    when '10' domain = MBReqDomain.InnerShareable;
    when '11' domain = MBReqDomain.FullSystem;

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSSBB</td>
<td>CRm == '0100'</td>
</tr>
<tr>
<td>SSBB</td>
<td>CRm == '0000'</td>
</tr>
</tbody>
</table>

Assembler symbols

<option> For the memory barrier variant: specifies the limitation on the barrier operation. Values are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SY</td>
<td>Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.</td>
</tr>
<tr>
<td>ST</td>
<td>Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.</td>
</tr>
<tr>
<td>LD</td>
<td>Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.</td>
</tr>
<tr>
<td>ISH</td>
<td>Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.</td>
</tr>
<tr>
<td>ISHST</td>
<td>Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.</td>
</tr>
<tr>
<td>ISHLD</td>
<td>Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.</td>
</tr>
<tr>
<td>NSH</td>
<td>Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.</td>
</tr>
<tr>
<td>NSHST</td>
<td>Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.</td>
</tr>
<tr>
<td>NSHLD</td>
<td>Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.</td>
</tr>
<tr>
<td>OSH</td>
<td>Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.</td>
</tr>
<tr>
<td>OSHST</td>
<td>Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.</td>
</tr>
<tr>
<td>OSHLD</td>
<td>Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.</td>
</tr>
</tbody>
</table>
All other encodings of CRm, other than the values 0b0000 and 0b0100, that are not listed above are reserved, and can be encoded using the #<imm> syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) on page B2-175 or see Data Synchronization Barrier (DSB) on page B2-178.

Note

The value 0b0000 is used to encode SSBB and the value 0b0100 is used to encode PSSBB.

For the memory nXS barrier variant: specifies the limitation on the barrier operation. Values are:

- SY: Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as imm2 = 0b11.
- ISH: Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as imm2 = 0b10.
- NSH: Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as imm2 = 0b01.
- OSH: Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as imm2 = 0b00.

<imm> is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation for all encodings

case alias of
    when DSBAlias_SSBB
        SpeculativeStoreBypassBarrierToVA();
    when DSBAlias_PSSBB
        SpeculativeStoreBypassBarrierToPA();
    when DSBAlias_DSB
        if HaveTME() && TSTATE.depth > 0 then
            FailTransaction(TMFailure_ERR, FALSE);
        if !nXS && HaveFeatXS() && HaveFeatHCX() then
            nXS = PSTATE.EL IN {EL0, EL1} && IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1';
            DataSynchronizationBarrier(domain, types, nXS);
        otherwise
            Unreachable();
    otherwise
        unreachable();
C6.2.117   DVP

Data Value Prediction Restriction by Context prevents data value predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Data value predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot be used to exploitatively control speculative execution occurring after the instruction is complete and synchronized.

For more information, see DVP RCTX.

This instruction is an alias of the SYS instruction. This means that:

• The encodings in this description are named to match the encodings of SYS.
• The description of SYS gives the operational pseudocode for this instruction.

System

(FEAT_SPECRES)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18</th>
<th>16</th>
<th>15 12</th>
<th>11 8</th>
<th>7 5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>op1</td>
<td>CRn</td>
<td>CRm</td>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

DVP RCTX, <Xt>

is equivalent to

SYS #3, C7, C3, #5, <Xt>

and is always the preferred disassembly.

Assembler symbols

<Xt>        Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.
C6.2.118   EON (shifted register)

Bitwise Exclusive OR NOT (shifted register) performs a bitwise Exclusive OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.

32-bit variant

Applies when \( sf = 0 \).

EON \(<Wd>, <Wn>, <Wm>{, <shift> #<amount>}\)

64-bit variant

Applies when \( sf = 1 \).

EON \(<Xd>, <Xn>, <Xm>{, <shift> #<amount>}\)

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } d &= \text{UInt(Rd)}; \\
\text{integer } n &= \text{UInt(Rn)}; \\
\text{integer } m &= \text{UInt(Rm)}; \\
\text{integer } datasize &= \text{if } sf = '1' \text{ then 64 else 32}; \\
\text{if } sf = '0' \text{ \&\& imm6<5> == '1' then UNDEFINED;}
\end{align*}
\]

\[
\text{ShiftType shift_type} = \text{DecodeShift(shift)}; \\
\text{integer shift_amount} = \text{UInt(imm6)};
\]

Assembler symbols

\(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

\(<Wm>\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

\(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

\(<Xm>\) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

\(<shift>\) Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- LSL when shift = 00
- LSR when shift = 01
- ASR when shift = 10
- ROR when shift = 11

\(<amount>\)

For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.
Operation

bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(datasize) result;

operand2 = NOT(operand2);

result = operand1 EOR operand2;

X[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.119   EOR (immediate)

Bitwise Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>16</th>
<th>15</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>1 0</td>
<td>0 1 0 0</td>
<td>N</td>
<td>immr</td>
<td>imms</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

### 32-bit variant

Applies when \( sf == 0 \&\& N == 0 \).

\[
\text{EOR } <\text{Wd}|\text{WSP}>, <\text{Wn}>, \#<\text{imm}>
\]

### 64-bit variant

Applies when \( sf == 1 \).

\[
\text{EOR } <\text{Xd}|\text{SP}>, <\text{Xn}>, \#<\text{imm}>
\]

#### Decode for all variants of this encoding

integer \( d = \text{UInt}(Rd); \)
integer \( n = \text{UInt}(Rn); \)
integer \( \text{datasize} = \text{if } sf == '1' \text{ then } 64 \text{ else } 32; \)
bits(\( \text{datasize} \)) \( \text{imm} \);
if \( sf == '0' \&\& N != '0' \) then UNDEFINED;
\( \langle \text{imm}, -\rangle = \text{DecodeBitMasks}(N, \text{imms}, \text{immr}, \text{TRUE, datasize}); \)

#### Assembler symbols

\( <\text{Wd}|\text{WSP}> \) Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

\( <\text{Wn}> \) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

\( <\text{Xd}|\text{SP}> \) Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

\( <\text{Xn}> \) Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

\( <\text{imm}> \) For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

#### Operation

bits(\( \text{datasize} \)) \( \text{result} \);
bv2s(bits(\( \text{datasize} \)) \( \text{operand1} = X[n, \text{datasize}]; \)
result = \( \text{operand1} \) \( \text{EOR} \) \( \text{imm} \);

if \( d == 31 \) then
    \( \text{SP}[] = \text{ZeroExtend} \text{result}, 64; \)
else
    \( X[d, \text{datasize}] = \text{result}; \)
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.120 EOR (shifted register)

Bitwise Exclusive OR (shifted register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

32-bit variant

Applies when \( sf = 0 \).

\[
\text{EOR} \ <Wd>, <Wn>, <Wm>{, <shift> #<amount>} 
\]

64-bit variant

Applies when \( sf = 1 \).

\[
\text{EOR} \ <Xd>, <Xn>, <Xm>{, <shift> #<amount>} 
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{integer } \text{datasize} &= \text{if } sf = '0' \text{ then } 32 \text{ else } 64; \\
\text{if } sf = '0' \text{ and } \text{imm6} < 5 \text{ then UNDEFINED;}
\end{align*}
\]

\[
\text{ShiftType } \text{shift_type} = \text{DecodeShift}(\text{shift}); \\
\text{integer } \text{shift_amount} = \text{UInt}(\text{imm6});
\]

Assembler symbols

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Wm>\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Xm>\) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<\text{shift}>\) Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:
  - LSL when \( \text{shift} = 00 \)
  - LSR when \( \text{shift} = 01 \)
  - ASR when \( \text{shift} = 10 \)
  - ROR when \( \text{shift} = 11 \)
- \(<\text{amount}>\) For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
**Operation**

bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(datasize) result;

result = operand1 EOR operand2;
X[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.121 ERET

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores PSTATE from the SPSR, and branches to the address held in the ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See Illegal exception returns from AArch64 state on page D1-4645.

ERET is UNDEFINED at EL0.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 1 0</td>
<td>1 0 0</td>
<td>1 1 1 1 1</td>
<td>0 0 0 0 0</td>
<td>0</td>
<td>1 1 1 1</td>
<td>0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

ERET

**Decode for this encoding**

if PSTATE.EL == EL0 then UNDEFINED;

**Operation**

```c
AArch64.CheckForERetTrap(FALSE, TRUE);
bits(64) target = ELR[];
AArch64.ExceptionReturn(target, SPSR[]);
```
C6.2.122   ERETAA, ERETAB

Exception Return, with pointer authentication. This instruction authenticates the address in ELR, using SP as the modifier and the specified key, the PE restores PSTATE from the SPSR for the current Exception level, and branches to the authenticated address.

Key A is used for ERETAA, and key B is used for ERETAB.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See Illegal exception returns from AArch64 state on page D1-4645.

ERETAA and ERETAB are UNDEFINED at EL0.

Integer

(FEAT_PAuth)

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|1 1 0 1 0 1 1 1 0|1 0 1 1 1 1 1|0 0 0 0 1|1 1 1 1 1 1|
```

A Rn op4

**ERETAA variant**

Applies when M == 0.

ERETAA

**ERETAB variant**

Applies when M == 1.

ERETAB

**Decode for all variants of this encoding**

```plaintext
if PSTATE.EL == EL0 then UNDEFINED;
boolean use_key_a = (M == '0');
if !HavePACExt() then
  UNDEFINED;
```

**Operation**

```plaintext
AArch64.CheckForERetTrap(TRUE, use_key_a);
bzts(64) target = ELR[];
bzts(64) modifier = SP[];
if use_key_a then
  target = AuthIA(target, modifier, TRUE);
else
  target = AuthIB(target, modifier, TRUE);
AArch64.ExceptionReturn(target, SPSR[]);
```
C6.2.123   ESB

Error Synchronization Barrier is an error synchronization event that might also update DISR_EL1 and VDISR_EL2.

This instruction can be used at all Exception levels and in Debug state.

In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. See Error Synchronization Barrier in the Arm(R) Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for Armv8-A architecture profile.

If the RAS Extension is not implemented, this instruction executes as a NOP.

System

(FEAT_RAS)

```
0 1 0 1 0 1 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 1 1 1 1
```

Encoding

ESB

Decode for this encoding

```
if !HaveRASExt() then EndOfInstruction();
```

Operation

```
if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);
    SynchronizeErrors();
    AArch64.ESBOperation();
if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESBOperation();
    TakeUnmaskedSErrorInterrupts();
```
C6.2.124 EXTR

Extract register extracts a register from a pair of registers.

This instruction is used by the alias ROR (immediate). See Alias conditions for details of when each alias is preferred.

32-bit variant
Applies when $sf == 0 \&\& N == 0 \&\& imms == 0xxxxx$.

EXTR $<Wd>, <Wn>, <Wm>, #<lsb>$

64-bit variant
Applies when $sf == 1 \&\& N == 1$.

EXTR $<Xd>, <Xn>, < Xm>, #<lsb>$

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if $sf == '1'$ then 64 else 32;
integer lsb;

if $N != sf$ then UNDEFINED;
if $sf == '0' \&\& imms<5> == '1'$ then UNDEFINED;
lsb = UInt(imms);

Alias conditions

<table>
<thead>
<tr>
<th>Alias is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROR (immediate) $Rn == Rm$</td>
</tr>
</tbody>
</table>

Assembler symbols

<i id> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<i in> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<i in> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<i rd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<i in> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<i in> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<lsb> For the 32-bit variant: is the least significant bit position from which to extract, in the range 0 to 31, encoded in the "imms" field.
For the 64-bit variant: is the least significant bit position from which to extract, in the range 0 to 63, encoded in the "imms" field.

**Operation**

bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
bits(2*datasize) concat = operand1:operand2;
result = concat<lsb+datasize-1:lsb>;
X[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.125  GMI

Tag Mask Insert inserts the tag in the first source register into the excluded set specified in the second source register, writing the new excluded set to the destination register.

**Integer**

(FeaT_MTE)

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9</th>
<th>0 5 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1 1 0 0 1 1 0 0 0</td>
<td>Xm 0 0 0 1 0 0 1</td>
<td>Xn</td>
</tr>
</tbody>
</table>
```

**Encoding**

GMI <Xd>, <Xn|SP>, <Xm>

**Decode for this encoding**

```
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

**Assembler symbols**

- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
- `<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Xm" field.

**Operation**

```
bits(64) address = if n == 31 then SP[] else X[n, 64];
bits(64) mask = X[m, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(address);
mask<UInt(tag)> = '1';
X[d, 64] = mask;
```
C6.2.126  HINT

Hint instruction is for the instruction set space that is reserved for architectural hint instructions.

Some encodings described here are not allocated in this revision of the architecture, and behave as NOPs. These encodings might be allocated to other hint functionality in future revisions of the architecture and therefore must not be used by software.

Encoding

HINT #<imm>

Decode for this encoding

    System HintOp op;
    
    case CRM:op2 of
        when '0000 000' op = System HintOp_NOP;
        when '0000 001' op = System HintOp_YIELD;
        when '0000 010' op = System HintOp_WFE;
        when '0000 011' op = System HintOp_WFI;
        when '0000 100' op = System HintOp_SEV;
        when '0000 101' op = System HintOp_SEVL;
        when '0000 110'
            if !HaveDGHExt() then EndOfInstruction();    // Instruction executes as NOP
            op = System HintOp_DGH;
        when '0000 111' SEE "XPACLRI";
        when '0001 xxx'
            case op2 of
                when '000' SEE "PACIA1716";
                when '010' SEE "PACIB1716";
                when '100' SEE "AUTIA1716";
                when '110' SEE "AUTIB1716";
                otherwise EndOfInstruction();
        when '0010 000'
            if !HaveRASExt() then EndOfInstruction();    // Instruction executes as NOP
            op = System HintOp_ESB;
        when '0010 001'
            if !HaveStatisticalProfiling() then EndOfInstruction();    // Instruction executes as NOP
            op = System HintOp_PSB;
        when '0010 010'
            if !HaveSelfHostedTrace() then EndOfInstruction();    // Instruction executes as NOP
            op = System HintOp_TSB;
        when '0010 100'
            op = System HintOp_CSDB;
        when '0011 xxx'
            case op2 of
                when '000' SEE "PACIAZ";
                when '001' SEE "PACIASP";
                when '010' SEE "PACIBZ";
                when '011' SEE "PACIBSP";
                when '100' SEE "AUTIAZ";
                when '101' SEE "AUTIASP";
                when '110' SEE "AUTIBZ";
                when '111' SEE "AUTIBSP";
            op = System HintOp_BTI;
                // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
A64 Base Instruction Descriptions
C6.2 Alphabetical list of A64 base instructions

Assembler symbols

<imm> Is a 7-bit unsigned immediate, in the range 0 to 127 encoded in the "CRm:op2" field. The encodings that are allocated to architectural hint functionality are described in the "Hints" table in the "Index by Encoding".

--- Note ---
For allocated encodings of "CRm:op2":
- A disassembler will disassemble the allocated instruction, rather than the HINT instruction.
- An assembler may support assembly of allocated encodings using HINT with the corresponding <imm> value, but it is not required to do so.

Operation

case op of
  when SystemHintOp_YIELD
    Hint_Yield();
  when SystemHintOp_DGH
    Hint_DGH();
  when SystemHintOp_WFE
    integer localtimeout = 1 << 64; // No local timeout event is generated
    Hint_WFE(localtimeout, WFxType_WFE);
  when SystemHintOp_WFI
    integer localtimeout = 1 << 64; // No local timeout event is generated
    Hint_WFI(localtimeout, WFxType_WFI);
  when SystemHintOp_SEV
    SendEvent();
  when SystemHintOp_SEVL
    SendEventLocal();
  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.ESBOperation();
    if PSTATE_EL IN {EL0, EL1} && EL2Enabled() then
      AArch64.eESBOperation();
      TakeUnmaskedErrorInterrupts();
    Hint_WFE(localtimeout, WFxType_WFE);
  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
    TraceSynchronizationBarrier();
  when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp_BTI
    SetBTypeNext('00');
  otherwise  // do nothing
C6.2.127   HLT

Halt instruction. An HLT instruction can generate a Halt Instruction debug event, which causes entry into Debug state.

```
   | 31 30 29 28|27 26 25 24|23 22 21 20| 5 4 3 2 1 0 |
1 1 0 1 0 1 0 0 0 1 0 | imm16 | 0 0 0 0 0 0 |
```

**Encoding**

HLT #<imm>

**Decode for this encoding**

if EDSCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
if HaveBTIExt() then
  SetBTypeCompatible(TRUE);

**Assembler symbols**

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

**Operation**

Halt(DebugHalt_HaltInstruction, FALSE);
C6.2.128   HVC

Hypervisor Call causes an exception to EL2. Software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is UNDEFINED:

- When EL3 is implemented and SCR_EL3.HCE is set to 0.
- When EL3 is not implemented and HCR_EL2.HCD is set to 1.
- When EL2 is not implemented.
- At EL1 if EL2 is not enabled in the current Security state.
- At EL0.

On executing an HVC instruction, the PE records the exception as a Hypervisor Call exception in ESR_ELx, using the EC value 0x16, and the value of the immediate argument.

```
[31 30 29 28|27 26 25 24|23 22 21 20]| | | | 5 4 3 2 1 0 |
1 1 0 1 0 1 0 0|0 0 0|imm16|0 0 0|1 0
```

**Encoding**

HVC #<imm>

**Decode for this encoding**

// Empty.

**Assembler symbols**

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

**Operation**

```cpp
if !HaveEL(EL2) || PSTATE.EL == EL0 || (PSTATE.EL == EL1 && !EL2Enabled()) then UNDEFINED;

hvc_enable = if HaveEL(EL3) then SCR_EL3.HCE else NOT(HCR_EL2.HCD);
if hvc_enable == '0' then UNDEFINED;
else AArch64.CallHypervisor(imm16);
```
IC

Instruction Cache operation. For more information, see op0==0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, and BRBE instructions on page C5-659.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td></td>
<td>CRm</td>
<td>op2</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;ic_op&gt;{, &lt;Xt&gt;}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

IC <ic_op>{, <Xt>}

is equivalent to

SYS #<op1>, C7, <Cm>, #<op2>{, <Xt>}

and is the preferred disassembly when SysOp(op1,'0111',CRm,op2) == Sys_IC.

**Assembler symbols**

<i_c_op> Is an IC instruction name, as listed for the IC system instruction pages, encoded in the "op1:CRm:op2" field. It can have the following values:

- IALLUIS  when op1 = 000, CRm = 0001, op2 = 000
- IALLU    when op1 = 000, CRm = 0101, op2 = 000
- IVAU     when op1 = 011, CRm = 0101, op2 = 001

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

**Operation**

The description of SYS gives the operational pseudocode for this instruction.
C6.2.130  IRG

Insert Random Tag inserts a random Logical Address Tag into the address in the first source register, and writes the result to the destination register. Any tags specified in the optional second source register or in GCR_EL1.Exclude are excluded from the selection of the random Logical Address Tag.

**Integer**

(FEAT_MTE)

```
[31 30 29 28|27 26 25 24|23 22 21 20]  16|15 14 13 12|11 10  9 |  5 4 |  0 |
 1 0 0 | 1 1 0 1 0 1 0 | 0 | 0 | 1 | 0 | 0 | Xn | Xd |
```

**Encoding**

IRG <Xd|SP>, <Xn|SP>{, <Xm>}

**Decode for this encoding**

```cpp
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

**Assembler symbols**

- `<Xd|SP>` is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
- `<Xn|SP>` is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
- `<Xm>` is the 64-bit name of the second general-purpose source register, encoded in the "Xm" field. Defaults to XZR if absent.

**Operation**

```cpp
bits(64) operand = if n == 31 then SP[] else X[n, 64];
bits(64) exclude_reg = X[m, 64];
bits(16) exclude = exclude_reg<15:0> OR GCR_EL1.Exclude;
bits(4) rtag;
if AArch64.AllocationTagAccessIsEnabled(AccType_NORMAL) then
  if GCR_EL1.RRND == '1' then
    if IsOnes(exclude) then
      rtag = '0000';
    else
      rtag = ChooseRandomNonExcludedTag(exclude);
  else
    bits(4) start = RGSR_EL1.TAG;
    bits(4) offset = AArch64.RandomTag();
    rtag = AArch64.ChooseNonExcludedTag(start, offset, exclude);
  else
    RGSR_EL1.TAG = rtag;
  rtag = '0000';
bits(64) result = AArch64.AddressWithAllocationTag(operand, AccType_NORMAL, rtag);
if d == 31 then
```

```cpp"
SP[] = result;
else
X[d, 64] = result;
C6.2.131 ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see *Instruction Synchronization Barrier (ISB)* on page B2-175.

```

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0</td>
<td>1 0 0 0 0</td>
<td>0 1 1 0 1</td>
<td>0 1 1 1 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CRm opc
```

*Encoding*

ISB {<option>|#<imm>}

*Decode for this encoding*

// No additional decoding required

*Assembler symbols*

<option> Specifies an optional limitation on the barrier operation. Values are:

SY Full system barrier operation, encoded as CRm = 0b1111. Can be omitted.

All other encodings of CRm are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

<imm> Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

*Operation*

```
InstructionSynchronizationBarrier();
if HaveBRBEExt() && BRBEBranchOnISB() then BRBEISB();
```
C6.2.132  LD64B

Single-copy Atomic 64-byte Load derives an address from a base register value, loads eight 64-bit doublewords from a memory location, and writes them to consecutive registers, \( X_t \) to \( X(t+7) \). The data that is loaded is atomic and is required to be 64-byte aligned.

**Integer**

(FEAT_LS64)

\[
\begin{array}{cccccccccccccccccccccccccccccc}
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 |  | 5 | 4 |  | 0 |
\end{array}
\]

Encoding

LD64B <Xt>, [<Xn|SP> {,#0}]

**Decode for this encoding**

if !HaveFeatLS64() then UNDEFINED;
if \( Rt<4:3> == '11' \) || \( Rt<0> == '1' \) then UNDEFINED;

\[
\begin{align*}
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } t &= \text{UInt}(Rt); \\
\text{boolean tag_checked} &= n != 31;
\end{align*}
\]

**Assembler symbols**

<Xt> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

CheckLDST64BEnabled();

bits(512) data;
bits(64) address;
bits(64) value;
acctype = AccType_ATOMICLS64;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if \( n == 31 \) then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
data = MemLoad64B(address, acctype);

for \( i = 0 \) to \( 7 \)
    value = data<63*64+i:64+i>;
    if BigEndian(acctype) then value = BigEndianReverse(value);
    \( X[t+i, 64] = value; \)
**C6.2.133  LDADDAB, LDADDB, LDADDALB, LDADDLB**

Atomic add on byte in memory atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAB and LDADDALB load from memory with acquire semantics.
- LDADDLB and LDADDALB store to memory with release semantics.
- LDADDB has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180*.

For information about memory accesses see *Load/store addressing modes on page C1-234*.

This instruction is used by the alias STADDB, STADDLB. See *Alias conditions on page C6-1514* for details of when each alias is preferred.

### Integer

**(FEAT_LSE)**

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20]</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
</tr>
</tbody>
</table>

**LDADDAB variant**

Applies when \(A == 1 &\& R == 0\).

LDADDAB <Ws>, <Wt>, [<Xn|SP>]

**LDADDALB variant**

Applies when \(A == 1 &\& R == 1\).

LDADDALB <Ws>, <Wt>, [<Xn|SP>]

**LDADDB variant**

Applies when \(A == 0 &\& R == 0\).

LDADDB <Ws>, <Wt>, [<Xn|SP>]

**LDADDLB variant**

Applies when \(A == 0 &\& R == 1\).

LDADDLB <Ws>, <Wt>, [<Xn|SP>]

**Decode for all variants of this encoding**

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' &\& Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STADDB, STADDLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

value = X[s, 8];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);

if t != 31 then
  X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.134  LDADDH, LDADDAH, LDADDLALH, LDADDLH

Atomic add on halfword in memory atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAH and LDADDLALH load from memory with acquire semantics.
- LDADDLH and LDADDLALH store to memory with release semantics.
- LDADDH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is used by the alias STADDH, STADDLH. See Alias conditions on page C6-1516 for details of when each alias is preferred.

**Integer**

(FEAT_LSE)

| [31 30 29 28 | [27 26 25 24 | [23 22 21 20 | [16 15 14 | [12 11 10 9 | 5 4 | 0 ] | size | opc |
| 0 | 1 | 1 | 1 | 0 | 0 | A | R | 1 | Rs | 0 | 0 | 0 | 0 | Rn | Rt |

**LDADDAH variant**

Applies when \( A = 1 \) && \( R = 0 \).

LDADDAH <Ws>, <Wt>, [Xn|SP>]

**LDADDLALH variant**

Applies when \( A = 1 \) && \( R = 1 \).

LDADDLALH <Ws>, <Wt>, [Xn|SP>]

**LDADDH variant**

Applies when \( A = 0 \) && \( R = 0 \).

LDADDH <Ws>, <Wt>, [Xn|SP>]

**LDADDLH variant**

Applies when \( A = 0 \) && \( R = 1 \).

LDADDLH <Ws>, <Wt>, [Xn|SP>]

**Decode for all variants of this encoding**

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

KarType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;

KarType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;

boolean tag_checked = n != 31;
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STADDH, STADDLH</td>
<td>A == '0' &amp; &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

value = X[s, 16];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);

if t != 31 then
  X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.135  LDADD, LDADDA, LDADDAL, LDADDL

Atomic add on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDADDA and LDADDAL load from memory with acquire semantics.
- LDADDL and LDADDAL store to memory with release semantics.
- LDADD has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is used by the alias STADD, STADDL. See Alias conditions on page C6-1518 for details of when each alias is preferred.

Integer

(FEAT_LSE)

\[
\begin{array}{cccccccccccccccccccc}
\text{size} & \text{opc} & 31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 & 23 & 22 & 21 & 20 & 16 & 15 & 14 & 12 & 11 & 10 & 9 & 5 & 4 & 0 \\
1 & x & 1 & 1 & 1 & 0 & 0 & 0 & A & R & 1 & Rn & 0 & 0 & 0 & 0 & Rn & Rt \\
\end{array}
\]

32-bit LDADD variant

Applies when size == 10 && A == 0 && R == 0.

LDADD <Ws>, <Wt>, [<Xn|SP>]

32-bit LDADDA variant

Applies when size == 10 && A == 1 && R == 0.

LDADDA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDADDAL variant

Applies when size == 10 && A == 1 && R == 1.

LDADDAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDADDL variant

Applies when size == 10 && A == 0 && R == 1.

LDADDL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDADD variant

Applies when size == 11 && A == 0 && R == 0.

LDADD <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDA variant

Applies when size == 11 && A == 1 && R == 0.

LDADDA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDL variant

Applies when size == 11 && A == 0 && R == 1.

LDADDL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDL variant

Applies when size == 11 && A == 1 && R == 1.

LDADDL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDL variant

Applies when size == 11 && A == 0 && R == 1.

LDADDL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDL variant

Applies when size == 11 && A == 1 && R == 1.

LDADDL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDL variant

Applies when size == 11 && A == 0 && R == 1.

LDADDL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDL variant

Applies when size == 11 && A == 1 && R == 1.

LDADDL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDL variant

Applies when size == 11 && A == 0 && R == 1.

LDADDL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDL variant

Applies when size == 11 && A == 1 && R == 1.

LDADDL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDL variant

Applies when size == 11 && A == 0 && R == 1.

LDADDL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDL variant

Applies when size == 11 && A == 1 && R == 1.

LDADDL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDL variant

Applies when size == 11 && A == 0 && R == 1.

LDADDL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDL variant

Applies when size == 11 && A == 1 && R == 1.
64-bit LDADDAL variant

Applies when size == 11 && A == 1 && R == 1.

LDADDAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDL variant

Applies when size == 11 && A == 0 && R == 1.

LDADDL <Xs>, <Xt>, [<Xn|SP>]

Decode for all variants of this encoding

if !HaveAtomicExt() then UNDEFINED;

t = UInt(Rt);
n = UInt(Rn);
s = UInt(Rs);
datasize = 8 << UInt(size);
regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
tag_checked = n != 31;

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STADD, STADDL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);
if \( t \neq 31 \) then
\[
X[t, \text{regsize}] = \text{ZeroExtend}(data, \text{regsize});
\]

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.136 LDAPR

Load-Acquire RCpc Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from the derived address in memory, and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180, except that:

• There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.

• The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/store addressing modes on page C1-234.

Integer

(FEAT_LRCPC)

32-bit variant

Applies when size == 10.

LDAPR <Wt>, [<Xn|SP> {,#0}]

64-bit variant

Applies when size == 11.

LDAPR <Xt>, [<Xn|SP> {,#0}]

Decode for all variants of this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTE2Ext() then
SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, dbytes, AccType_ORDERED];
X[t, regsize] = ZeroExtend(data, regsize);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.137   LDAPRB

Load-Acquire RCpc Register Byte derives an address from a base register value, loads a byte from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/store addressing modes on page C1-234.

Integer

(FEAT_LRCPC)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 0</td>
<td>0 0 1 0 1</td>
<td>1</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Encoding

LDAPRB <Wt>, [<Xn|SP> {,#0}]

Decode for this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 1, AccType_ORDERED];
X[t, 32] = ZeroExtend(data, 32);
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.138 LDAPRH

Load-Acquire RCpc Register Halfword derives an address from a base register value, loads a halfword from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180, except that:

• There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.

• The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/store addressing modes on page C1-234.

Integer

(FEAT_LRCPC)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0 0 0 1 0 1 (1) (1) (1) (1)</td>
<td>1 1 0 0 0 0</td>
<td>Rn</td>
<td>Rt</td>
</tr>
</tbody>
</table>

size Rs

Encoding

LDAPRH <Wt>, [<Xn|SP> {,#0}]

Decode for this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) data;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPLignment();
  address = SP[];
else
  address = X[n, 64];
data = Mem[address, 2, AccType_ORDERED];
X[t, 32] = ZeroExtend(data, 32);
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.139  LDAPUR

Load-Acquire RCpc Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180, except that:

• There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.

• The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/store addressing modes on page C1-234.

Unscaled offset

(FEAT_LRCPC2)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th></th>
<th>12</th>
<th>11 10 9</th>
<th></th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
<td></td>
<td>imm9</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit variant

Applies when size == 10.

LDAPUR <Wt>, [<Xn|SP>{, #<simm>}]

64-bit variant

Applies when size == 11.

LDAPUR <Xt>, [<Xn|SP>{, #<simm>}]

Decode for all variants of this encoding

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;
regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;

**Operation**

bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType.ORDERED];
X[t, regsize] = ZeroExtend(data, regsize);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.140  LDAPURB

Load-Acquire RCpc Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/store addressing modes on page C1-234.

**Unscaled offset**

(FEAT_LRCPC2)

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 0 1 0 1 0 1 0</td>
<td>imm9 0 0  Rn</td>
<td>Rt</td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

LDAPURB <Wt>, [<Xn|SP>{, #<simm>}]}

**Decode for this encoding**

bits(64) offset = SignExtend(imm9, 64);

**Assembler symbols**

- `<wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared decode for all encodings**

```pseudo
t = UInt(Rt);
n = UInt(Rn);
tag_checked = n != 31;
```

**Operation**

```pseudo
address = bits(64);
data = bits(8);
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
```
address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 1, AccType.ORDERED];
X[t, 32] = ZeroExtend(data, 32);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.141 LDAPURH

Load-Acquire RCpc Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-180, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory ordering, see *Load/store addressing modes* on page C1-234.

**Unscaled offset**

(FEAT_LRCPC2)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

LDAPURH <Wt>, [<Xn|SP>{, #<simm>}]

**Decode for this encoding**

bits(64) offset = SignExtend(imm9, 64);

**Assembler symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared decode for all encodings**

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

**Operation**

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
address = SP[];
else
    address = X[n, 64];
address = address + offset;
data = Mem[address, 2, AccType_ORDERED];
X[t, 32] = ZeroExtend(data, 32);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPURSB

Load-Acquire RCpc Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-180, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/store addressing modes* on page C1-234.

**Unscaled offset**

(FEAT_LRCPC2)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
<td></td>
<td></td>
<td>imm9</td>
<td>0 0</td>
<td>Rn</td>
</tr>
</tbody>
</table>
```

**32-bit variant**

Applies when opc == 11.

LDAPURSB <Wt>, [<Xn|SP>{, #<simm>}] 

**64-bit variant**

Applies when opc == 10.

LDAPURSB <Xt>, [<Xn|SP>{, #<simm>}] 

**Decode for all variants of this encoding**

bits(64) offset = SignExtend(imm9, 64);

**Assembler symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared decode for all encodings**

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
```
// store or zero-extending load
memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
regsize = 32;
signed = FALSE;
else
  // sign-extending load
memop = MemOp_LOAD;
regsize = if opc<0> == '1' then 32 else 64;
signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);

**Operation**

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

address = address + offset;

case memop of
  when MemOp_STORE
    data = X[t, 8];
    Mem[address, 1, AccType_ORDERED] = data;
  when MemOp_LOAD
    data = Mem[address, 1, AccType_ORDERED];
    if signed then
      X[t, regsize] = SignExtend(data, regsize);
    else
      X[t, regsize] = ZeroExtend(data, regsize);
  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.143 LDAPURSH

Load-Acquire RCpc Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180, except that:

• There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.

• The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/store addressing modes on page C1-234.

Unscaled offset

(FEAT_LRCPC2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 12 | 11 | 10 | 9 | 5 | 4 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | x  | 0  | imm9| 0  | 0  | Rn | Rt |

32-bit variant

Applies when opc == 11.

LDAPURSH <Wt>, [<Xn|SP>{, #<simm>}]

64-bit variant

Applies when opc == 10.

LDAPURSH <Xt>, [<Xn|SP>{, #<simm>}]

Decode for all variants of this encoding

bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
teger regsize;
if opc<1> == '0' then
// store or zero-extending load
memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
regsize = 32;
signed = FALSE;
else
// sign-extending load
memop = MemOp_LOAD;
regsize = if opc<0> == '1' then 32 else 64;
signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);

Operation

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t, 16];
        Mem[address, 2, AccType_ORDERED] = data;
    when MemOp_LOAD
        data = Mem[address, 2, AccType_ORDERED];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.144 LDAPURSW

Load-Acquire RCpc Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/store addressing modes on page C1-234.

Unscaled offset

(FEAT_LRCPC2)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>12 11 10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 0 0 1 1 0 0</td>
<td>imm9</td>
<td>0 0</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

LDAPURSW <Xt>, [<Xn|SP>{, #<simm>}]  

Decode for this encoding

bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Operation

bits(64) address;
bits(32) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
address = SP[];
else
  address = X[n, 64];

address = address + offset;
data = Mem[address, 4, AccType.ORDERED];
X[t, 64] = SignExtend(data, 64);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.145  LDAR

Load-Acquire Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180. For information about memory accesses, see Load/store addressing modes on page C1-234.

--- Note ---

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

32-bit variant

Applies when \( \text{size} = 10 \).

\[
\text{LDAR} <Wt>, \ [<Xn|SP>\{,#0}\]
\]

64-bit variant

Applies when \( \text{size} = 11 \).

\[
\text{LDAR} <Xt>, \ [<Xn|SP>\{,#0}\]
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } n &= \text{UInt}(\text{Rn}); \\
\text{integer } t &= \text{UInt}(\text{Rt}); \\
\text{integer } \text{elsize} &= 8 \ll \text{UInt}(\text{size}); \\
\text{integer } \text{regsize} &= \text{if } \text{elsize} = 64 \text{ then } 64 \text{ else } 32; \\
\text{boolean } \text{tag_checked} &= n \neq 31;
\end{align*}
\]

Assembler symbols

\(<Wt>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xt>\) Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

\[
\begin{align*}
\text{bits(64)} &\quad \text{address}; \\
\text{bits(elsize)} &\quad \text{data}; \\
\text{constant } \text{integer } dbytes &= \text{elsize} \ll 3;
\end{align*}
\]

\[
\begin{align*}
\text{if } &\text{HaveMTE2Ext() then} \\
&\text{SetTagCheckedInstruction(tag_checked);}
\end{align*}
\]

\[
\begin{align*}
\text{if } n &= 31 \text{ then} \\
&\text{CheckSPAAlignment();} \\
&\text{address} = \text{SP[]}; \\
\text{else} \\
&\text{address} = X[n, 64];
\end{align*}
\]
data = Mem[address, dbytes, AccType_ORDERED];
X[t, regsize] = ZeroExtend(data, regsize);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.146   LDARB

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in "Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180. For information about memory accesses, see "Load/store addressing modes on page C1-234."

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

### Encoding

LDARB <Wt>, [<Xn|SP>{,#0}]

### Decode for this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

### Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

data = Mem[address, 1, AccType.ORDERED];
X[t, 32] = ZeroExtend(data, 32);

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.147   LDARH

Load-Acquire Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180. For information about memory accesses, see Load/store addressing modes on page C1-234.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

Encoding

LDARH <Wt>, [<Xn|SP>{,#0}]

Decode for this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

data = Mem[address, 2, AccType_ORDERED];
X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.148  LDAXP

Load-Acquire Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity on page 2154 and Alignment of data accesses on page 2189. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores on page 2211. The instruction also has memory ordering semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page 2210. For information about memory accesses, see Load/store addressing modes on page 234.

32-bit variant

Applies when sz == 0.
LDAXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]

64-bit variant

Applies when sz == 1.
LDAXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]

Decode for all variants of this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE;  // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly LDAXP on page K1-11591.

Assembler symbols

<#t1>  Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<#t2>  Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>  Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t, datasize] = bits(datasize) UNKNOWN; // In this case t = t2
elseif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, AccType_ORDEREDATOMIC];
    if BigEndian(AccType_ORDEREDATOMIC) then
        X[t, datasize-elsize] = data<datasize-1:elsize>
        X[t2, elsize] = data<elsize-1:0>
    else
        X[t, elsize] = data<elsize-1:0>
        X[t2, datasize-elsize] = data<datasize-1:elsize>
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
        AArch64.Abort(address, AlignmentFault(AccType_ORDEREDATOMIC, FALSE, FALSE));
    X[t, 64] = Mem[address, 8, AccType_ORDEREDATOMIC];
    X[t2, 64] = Mem[address+8, 8, AccType_ORDEREDATOMIC];

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.149  **LDAXR**

Load-Acquire Exclusive Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores on page B2-211*. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180*. For information about memory accesses see *Load/store addressing modes on page C1-234*.

32-bit variant

Applies when `size == 10`.

LDAXR <Wt>, [<Xn|SP>{,##0}]

64-bit variant

Applies when `size == 11`.

LDAXR <Xt>, [<Xn|SP>{,##0}]

**Decode for all variants of this encoding**

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;
```

**Assembler symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

```plaintext
bits(64) address;
bias(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
```

32-bit encoding

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x 0 0 1 0 0 0 0 0 1 0</td>
<td>1 1</td>
<td>(1) (1) (1) (1)</td>
<td>1</td>
<td>(1) (1) (1) (1)</td>
</tr>
</tbody>
</table>

size L Rs o0 Rt2
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

data = Mem[address, dbytes, AccType_ORDEREDATOMIC];
X[t, regsize] = ZeroExtend(data, regsize);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.150   LDAXRB

Load-Acquire Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores on page B2-211. The instruction also has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180. For information about memory accesses see Load/store addressing modes on page C1-234.

Encoding

LDAXRB <Wt>, [<Xn|SP>{,#0}]

Decode for this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, AccType_ORDEREDATOMIC];
X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.151 LDAXRH

Load-Acquire Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores on page B2-211. The instruction also has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180. For information about memory accesses see Load/store addressing modes on page C1-234.

Encoding

LDAXRH <Wt>, [<Xn|SP>{,#0}]

Decode for this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, AccType_ORDERED_ATOMIC];
X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.152 LDCLRAB, LDCLRALB, LDCLRLB

Atomic bit clear on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAB and LDCLRALB load from memory with acquire semantics.
- LDCLRLB and LDCLRALB store to memory with release semantics.
- LDCLRAB has neither acquire nor release semantics.


For information about memory accesses see *Load/store addressing modes* on page C1-234.

This instruction is used by the alias STCLRAB, STCLRLB. See *Alias conditions* on page C6-1549 for details of when each alias is preferred.

**Integer**

(FEAT_LSE)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14</th>
<th>12 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 0</td>
<td></td>
<td>Rs</td>
<td>0 0 0 1 0 0</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
</tr>
</tbody>
</table>
```

**LDCLRAB variant**

Applies when A == 1 && R == 0.

LDCLRAB <Ws>, <Wt>, [Xn|SP]

**LDCLRALB variant**

Applies when A == 1 && R == 1.

LDCLRALB <Ws>, <Wt>, [Xn|SP]

**LDCLRB variant**

Applies when A == 0 && R == 0.

LDCLRB <Ws>, <Wt>, [Xn|SP]

**LDCLRLB variant**

Applies when A == 0 && R == 1.

LDCLRLB <Ws>, <Wt>, [Xn|SP]

**Decode for all variants of this encoding**

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STCLRB, STCLRLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```c
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.153  LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH

Atomic bit clear on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAH and LDCLRALH load from memory with acquire semantics.
- LDCLRH and LDCLRALH store to memory with release semantics.
- LDCLRH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is used by the alias STCLRH, STCLRLH. See Alias conditions on page C6-1551 for details of when each alias is preferred.

**Integer**

(FEAT_LSE)

| [31 30 29 28] [27 26 25 24] [23 22 21 20] | 16 | 15 | 14 | 12 | 11 | 10 | 9 | 5 | 4 | 0 | size opc |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 0 | A | R | 1 | Rs | 0 | 0 | 1 | 0 | 0 | Rn | Rt |

**LDCLRAH variant**

Applies when \( A = 1 \land R = 0 \).

LDCLRAH \(<Ws>, <Wt>, [<Xn|SP>]\)

**LDCLRALH variant**

Applies when \( A = 1 \land R = 1 \).

LDCLRALH \(<Ws>, <Wt>, [<Xn|SP>]\)

**LDCLRH variant**

Applies when \( A = 0 \land R = 0 \).

LDCLRH \(<Ws>, <Wt>, [<Xn|SP>]\)

**LDCLRLH variant**

Applies when \( A = 0 \land R = 1 \).

LDCLRLH \(<Ws>, <Wt>, [<Xn|SP>]\)

**Decode for all variants of this encoding**

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' \&\& Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STCLRH, STCLRLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

- `<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```plaintext
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[;];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.154 LDCLR, LDCLRA, LDCLRAL, LDCLRL

Atomic bit clear on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDCLRA and LDCLRAL load from memory with acquire semantics.
- LDCLRL and LDCLRAL store to memory with release semantics.
- LDCLR has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is used by the alias STCLR, STCLRL. See Alias conditions on page C6-1553 for details of when each alias is preferred.

### Integer

(FEAT_LSE)

| [31 30 29 28][27 26 25 24][23 22 21 20] | 16|15 14 12|11 10 9 | 5 4 | 0 |
|------------------------------------------|-----------------|
| size | opc |

#### 32-bit LDCLR variant

Applies when size == 10 && A == 0 && R == 0.

LDCLR <Ws>, <Wt>, [<Xn|SP>]

#### 32-bit LDCLRA variant

Applies when size == 10 && A == 1 && R == 0.

LDCLRA <Ws>, <Wt>, [<Xn|SP>]

#### 32-bit LDCLRAL variant

Applies when size == 10 && A == 1 && R == 1.

LDCLRAL <Ws>, <Wt>, [<Xn|SP>]

#### 32-bit LDCLRL variant

Applies when size == 10 && A == 0 && R == 1.

LDCLRL <Ws>, <Wt>, [<Xn|SP>]

#### 64-bit LDCLR variant

Applies when size == 11 && A == 0 && R == 0.

LDCLR <Xs>, <Xt>, [<Xn|SP>]

#### 64-bit LDCLRA variant

Applies when size == 11 && A == 1 && R == 0.

LDCLRA <Xs>, <Xt>, [<Xn|SP>]

#### 64-bit LDCLRAL variant

Applies when size == 11 && A == 1 && R == 1.

LDCLRAL <Xs>, <Xt>, [<Xn|SP>]

#### 64-bit LDCLRL variant

Applies when size == 11 && A == 0 && R == 1.

LDCLRL <Xs>, <Xt>, [<Xn|SP>]
64-bit LDCLRAL variant
Applies when size == 11 && A == 1 && R == 1.
LDCLRAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDCLRL variant
Applies when size == 11 && A == 0 && R == 1.
LDCLRL <Xs>, <Xt>, [<Xn|SP>]

Decode for all variants of this encoding
if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STCLR, STCLRL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation
bits(64) address;
bits(datasize) value;
bits(datasize) data;
if HaveMTE2Ext() then
   SetTagCheckedInstruction(tag_checked);
value = X[s, datasize];
if n == 31 then
   CheckSPAlignment();
   address = SP[];
else
   address = X[n, 64];
data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);
if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.155 LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic exclusive OR on byte in memory atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAB and LDEORALB load from memory with acquire semantics.
- LDEORLB and LDEORALB store to memory with release semantics.
- LDEORB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is used by the alias STEORB, STEORLB. See Alias conditions on page C6-1556 for details of when each alias is preferred.

**Integer**

(Feat_LSE)

| [31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 | 12|11 10 9 | 5 4 | 0 |        |
|-------------|-------|-------|-------|--------|-------|--------|-------|
| size | opc |       |       |       |       |       |       |

**LDEORAB variant**

Applies when \( A == 1 \) \&\& \( R == 0 \).

LDEORAB \(<Ws>, <Wt>, [<Xn|SP>]\)

**LDEORALB variant**

Applies when \( A == 1 \) \&\& \( R == 1 \).

LDEORALB \(<Ws>, <Wt>, [<Xn|SP>]\)

**LDEORB variant**

Applies when \( A == 0 \) \&\& \( R == 0 \).

LDEORB \(<Ws>, <Wt>, [<Xn|SP>]\)

**LDEORLB variant**

Applies when \( A == 0 \) \&\& \( R == 1 \).

LDEORLB \(<Ws>, <Wt>, [<Xn|SP>]\)

**Decode for all variants of this encoding**

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' \&\& Rt != '11111' then AccType.ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType.ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEORB, STEORLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.156 LDEORH, LDEORAH, LDEORALH, LDEORLH

Atomic exclusive OR on halfword in memory atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAH and LDEORALH load from memory with acquire semantics.
- LDEORLH and LDEORALH store to memory with release semantics.
- LDEORH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is used by the alias STEORH, STEORLH. See Alias conditions on page C6-1558 for details of when each alias is preferred.

Integer

(FEAT_LSE)

<table>
<thead>
<tr>
<th>[31 30 29 28][27 26 25 24][23 22 21 20]</th>
<th>16</th>
<th>15 14 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0 1 1 1 0 0]</td>
<td>Rs</td>
<td>[0 1 0 0]</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
</tr>
</tbody>
</table>

LDEORAH variant

Applies when A == 1 && R == 0.
LDEORAH <Ws>, <Wt>, [<Xn|SP>]

LDEORALH variant

Applies when A == 1 && R == 1.
LDEORALH <Ws>, <Wt>, [<Xn|SP>]

LDEORH variant

Applies when A == 0 && R == 0.
LDEORH <Ws>, <Wt>, [<Xn|SP>]

LDEORLH variant

Applies when A == 0 && R == 1.
LDEORLH <Ws>, <Wt>, [<Xn|SP>]

Decode for all variants of this encoding

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEORH, STEORLH</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.157 LDEOR, LDEORA, LDEORAL, LDEORL

Atomic exclusive OR on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDEORA and LDEORAL load from memory with acquire semantics.
- LDEORL and LDEORAL store to memory with release semantics.
- LDEOR has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is used by the alias STEOR, STEORL. See Alias conditions on page C6-1560 for details of when each alias is preferred.

**Integer**

(FEAT_LSE)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1x 111000 A 0 0 0 R1 Rs 0 0 1 0 0 Rn Rt
```

**32-bit LDEOR variant**

Applies when size == 10 && A == 0 && R == 0.

LDEOR <Ws>, <Wt>, [<Xn|SP>]

**32-bit LDEORA variant**

Applies when size == 10 && A == 1 && R == 0.

LDEORA <Ws>, <Wt>, [<Xn|SP>]

**32-bit LDEORAL variant**

Applies when size == 10 && A == 1 && R == 1.

LDEORAL <Ws>, <Wt>, [<Xn|SP>]

**32-bit LDEORL variant**

Applies when size == 10 && A == 0 && R == 1.

LDEORL <Ws>, <Wt>, [<Xn|SP>]

**64-bit LDEOR variant**

Applies when size == 11 && A == 0 && R == 0.

LDEOR <Xs>, <Xt>, [<Xn|SP>]

**64-bit LDEORA variant**

Applies when size == 11 && A == 1 && R == 0.

LDEORA <Xs>, <Xt>, [<Xn|SP>]
64-bit **LDEORAL** variant

Applies when size == 11 && A == 1 && R == 1.

LDEORAL <Xs>, <Xt>, [<Xn|SP>]

64-bit **LDEORL** variant

Applies when size == 11 && A == 0 && R == 1.

LDEORL <Xs>, <Xt>, [<Xn|SP>]

**Decode for all variants of this encoding**

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

**Alias conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEOR, STEORL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

**Assembler symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

value = X[s, datasize];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];
data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);
if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.158   LDG

Load Allocation Tag loads an Allocation Tag from a memory address, generates a Logical Address Tag from the Allocation Tag and merges it into the destination register. The address used for the load is calculated from the base register and an immediate signed offset scaled by the Tag granule.

**Integer**

(FEAT_MTE)

```
[31 30 29 28|27 26 25 24|23 22 21 20] | 12|11 10 9 | 5 4 | 0 |
  1 1 0 1 1 0 0 1 0 1 1 | imm9 | 0 0 | Xn | Xt
```

**Encoding**

LDG <Xt>, [<Xn|SP>{, #<simm>}]

**Decode for this encoding**

```c
if !HaveMTEExt() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
```

**Assembler symbols**

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Xt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

<simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

**Operation**

```c
bits(64) address;
bits(4) tag;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;
address = Align(address, TAG_GRANULE);

tag = AArch64.MemTag[address, AccType_NORMAL];
X[t, 64] = AArch64.AddressWithAllocationTag(X[t, 64], AccType_NORMAL, tag);
```
C6.2.159  LDGM

Load Tag Multiple reads a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID_EL1.BS, and writes the Allocation Tag read from address A to the destination register at 4*A<7:4>+3:4*A<7:4>. Bits of the destination register not written with an Allocation Tag are set to 0.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

**Integer**

(FEAT_MTE2)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1</td>
<td>1 1</td>
<td>0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Xn</td>
</tr>
</tbody>
</table>

**Encoding**

LDGM <Xt>, [<Xn|SP>]

**Decode for this encoding**

if !HaveMTE2Ext() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);

**Assembler symbols**

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Xt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

**Operation**

if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = Zeros(64);
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

integer size = 4 * (2 ^ (UInt(GMID_EL1.BS)));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);

for i = 0 to count-1
    bits(4) tag = AArch64.MemTag[address, AccType_NORMAL];
    data<(index*4)+3:index*4> = tag;
    address = address + TAG_GRANULE;
    index = index + 1;

X[t, 64] = data;
C6.2.160  LDLARB

Load LOAcquire Register Byte loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in LoadLOAcquire, StoreLORelease on page B2-181. For information about memory accesses, see Load/store addressing modes on page C1-234.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset

(FEAT_LOR)

Encoding

LDLARB <Wt>, [<Xn|SP>{,#0}]

Decode for this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Assembler symbols

<Wt>  Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPLignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 1, AccType_LIMITEDORDERED];
X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.161 LDLARH

Load LOAcquire Register Halfword loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in LoadLOAcquire, StoreLORelease on page B2-181. For information about memory accesses, see Load/store addressing modes on page C1-234.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset

(FEAT_LOR)

 Encoding

LDLARH <Wt>, [<Xn|SP>{,#0}]

Decode for this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Assemble symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
SetTagCheckedInstruction(tag_checked);

if n == 31 then
CheckSPAlignment();
address = SP[];
else
address = X[n, 64];

data = Mem[address, 2, AccType_LIMITEDORDERED];
X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.162 LDLAR

Load LOAcquire Register loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in LoadLOAcquire, StoreLORelease on page B2-181. For information about memory accesses, see Load/store addressing modes on page C1-234.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset

(FEAT_LOR)

| 31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 | 10 9 | 5 4 | 0 |
|-----------------|-----------------|-----------------|--------------|--------------|--------------|--------------|--------------|--------------|
| 1 | x | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | (1) (1) (1) (1) | 0 | (1) (1) (1) (1) | Rn | Rt |

32-bit variant

Applies when size == 10.

LDLAR <Wt>, [<Xn|SP>{,#0}]

64-bit variant

Applies when size == 11.

LDLAR <Xt>, [<Xn|SP>{,#0}]

Decode for all variants of this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;

Assembler symbols

<Wr> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[0];
else
address = X[n, 64];
data = Mem[address, dbytes, AccType_LIMITEDORDERED];
X[t, regsize] = ZeroExtend(data, regsize);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.163   LDNP

Load Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers.

For information about memory accesses, see Load/store addressing modes on page C1-234. For information about Non-temporal pair instructions, see Load/store non-temporal pair on page C3-259.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

opc L

32-bit variant

Applies when opc == 00.

LDNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]

64-bit variant

Applies when opc == 10.

LDNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]

Decode for all variants of this encoding

// Empty.

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly LDNP on page K1-11591.

Assembler symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<<6 == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;  // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

Operation

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

if HaveLSE2Ext() then
    bits(2*datasize) full_data;
    full_data = Mem[address, 2*dbytes, AccType_NORMAL, TRUE];
    if BigEndian(AccType_STREAM) then
        data2 = full_data<(datasize-1):0>;
        data1 = full_data<(2*datasize-1):datasize>;
    else
        data1 = full_data<(datasize-1):0>;
        data2 = full_data<(2*datasize-1):datasize>;
    else
        data1 = Mem[address, dbytes, AccType_STREAM];
        data2 = Mem[address+dbytes, dbytes, AccType_STREAM];
    if rt_unknown then
        data1 = bits(datasize) UNKNOWN;
        data2 = bits(datasize) UNKNOWN;
        X[t, datasize] = data1;
        X[t2, datasize] = data2;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.164   LDP

Load Pair of Registers calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see *Load/store addressing modes on page C1-234.*

**Post-index**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 0</td>
<td>1 0 1 0 0 1 1</td>
<td>imm7</td>
<td>Rt2</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
</tr>
<tr>
<td>opc</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 32-bit variant

Applies when opc == 00.

LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>

### 64-bit variant

Applies when opc == 10.

LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>

**Decode for all variants of this encoding**

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

**Pre-index**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 0</td>
<td>1 0 1 0 0 1 1</td>
<td>imm7</td>
<td>Rt2</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
</tr>
<tr>
<td>opc</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 32-bit variant

Applies when opc == 00.

LDP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]

### 64-bit variant

Applies when opc == 10.

LDP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]

**Decode for all variants of this encoding**

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

### Signed offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 0</td>
<td>1 0 1 0 0 1 1</td>
<td>imm7</td>
<td>Rt2</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
</tr>
</tbody>
</table>
32-bit variant

Applies when opc == 00.

LDP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]  

64-bit variant

Applies when opc == 10.

LDP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]  

Decode for all variants of this encoding

boolean wback = FALSE;
boolean postindex = FALSE;

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly LDP on page K1-11592.

Assembler symbols

<Wt1>  Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2>  Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1>  Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2>  Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>  For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.  

For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.  

For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.  

For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
assert c IN {Constraint_WSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
case c of
    when Constraint_WSUPPRESS wback = FALSE; // writeback is suppressed
    when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
case c of
    when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

Operation for all encodings

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
if !postindex then
    address = address + offset;
if HaveLSE2Ext() && !signed then
    bits(2*datasize) full_data;
    full_data = Mem[address, 2*dbytes, AccType_NORMAL, TRUE];
    if BigEndian(AccType_NORMAL) then
        data2 = full_data<(datasize-1):0>;
        data1 = full_data<(2*datasize-1):datasize>;
    else
        data1 = full_data<(datasize-1):0>;
        data2 = full_data<(2*datasize-1):datasize>;
else
    data1 = Mem[address, dbytes, AccType_NORMAL];
    data2 = Mem[address+dbytes, dbytes, AccType_NORMAL];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
if signed then
    X[t, 64] = SignExtend(data1, 64);
    X[t2, 64] = SignExtend(data2, 64);
else
    X[t, datasize] = data1;
    X[t2, datasize] = data2;
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
if n == 31 then
    SP[] = address;
else
    X[n, 64] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.165   **LDPSW**

Load Pair of Registers Signed Word calculates an address from a base register value and an immediate offset, loads two 32-bit words from memory, sign-extends them, and writes them to two registers. For information about memory accesses, see *Load/store addressing modes* on page C1-234.

**Post-index**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 1 0 0 1 1</td>
<td></td>
<td></td>
<td>imm7</td>
<td>Rt2</td>
<td>Rn</td>
<td>Rt</td>
</tr>
<tr>
<td>opc</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>

**Decode for this encoding**

boolean wback = TRUE;
boolean postindex = TRUE;

**Pre-index**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 1 0 0 1 1</td>
<td></td>
<td></td>
<td>imm7</td>
<td>Rt2</td>
<td>Rn</td>
<td>Rt</td>
</tr>
<tr>
<td>opc</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

LDPSW <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]

**Decode for this encoding**

boolean wback = TRUE;
boolean postindex = FALSE;

**Signed offset**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 1 0 0 1 1</td>
<td></td>
<td></td>
<td>imm7</td>
<td>Rt2</td>
<td>Rn</td>
<td>Rt</td>
</tr>
<tr>
<td>opc</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

LDPSW <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}

**Decode for this encoding**

boolean wback = FALSE;
boolean postindex = FALSE;
Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architec
tural Constraints on UNPREDICTABLE Behaviors, and particularly LDPSW on page K1-11593.

Assembler symbols

\(<Xt1>\)
Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xt2>\)
Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

\(<Xn|SP>\)
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<\text{imm}>\)
For the post-index and pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as \(<\text{imm}>/4\).

For the signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as \(<\text{imm}>/4\).

Shared decode for all encodings

\(
\text{integer } n = \text{UInt(Rn)}; \\
\text{integer } t = \text{UInt(Rt)}; \\
\text{integer } t2 = \text{UInt(Rt2)}; \\
\text{bits(64) offset = LSL(SignExtend(imm7, 64), 2);} \\
\text{boolean tag_checked = wback || n != 31;}
\)

\(\text{boolean rt_unknown = FALSE;}
\text{boolean wb_unknown = FALSE;}
\)

\(\text{if wback && (t == n || t2 == n) && n != 31 then}
\text{Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);}
\text{assert c IN \{Constraint_WSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP\};}
\text{case c of}
\text{when Constraint_WSUPPRESS wback = FALSE; // writeback is suppressed}
\text{when Constraint_UNKNOWN wbUnknown = TRUE; // writeback is UNKNOWN}
\text{when Constraint_UNDEF UNDEFINED;}
\text{when Constraint_NOP EndOfInstruction();}
\)

\(\text{if t == t2 then}
\text{Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);}
\text{assert c IN \{Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP\};}
\text{case c of}
\text{when Constraint_UNKNOWN rtUnknown = TRUE; // result is UNKNOWN}
\text{when Constraint_UNDEF UNDEFINED;}
\text{when Constraint_NOP EndOfInstruction();}
\)

Operation for all encodings

\(\text{bits(64) address;}
\text{bits(32) data1;}
\text{bits(32) data2;}
\)

\(\text{if HaveMTE2Ext() then}
\text{SetTagCheckedInstruction(tag_checked);}
\)

\(\text{if n == 31 then}
\text{CheckSPAlignment();}
\text{address = SP[ ];}
\text{else}
\text{address = X[n, 64];}
\text{if !postindex then}
address = address + offset;

data1 = Mem[address, 4, AccType_NORMAL];
data2 = Mem[address+4, 4, AccType_NORMAL];
if rt_unknown then
  data1 = bits(32) UNKNOWN;
data2 = bits(32) UNKNOWN;
X[t, 64] = SignExtend(data1, 64);
X[t2, 64] = SignExtend(data2, 64);
if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n, 64] = address;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.166  **LDR (immediate)**

Load Register (immediate) loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/store addressing modes on page C1-234*. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

**Post-index**

| [31 30 29 28|27 26 25 24|23 22 21 20] | [12|11 10 9 | 5 4 | 0] |
|---------|------------------|------------------|-------------------|
| 1 x 1 1 1 0 0 0 1 0 | imm9 | 0 1 | Rn | Rt |

**32-bit variant**

Applies when size == 10.

LDR <Wt>, [Xn|SP>, #<simm>

**64-bit variant**

Applies when size == 11.

LDR <Xt>, [Xn|SP>, #<simm>

**Decode for all variants of this encoding**

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bias(64) offset = SignExtend(imm9, 64);

**Pre-index**

| [31 30 29 28|27 26 25 24|23 22 21 20] | [12|11 10 9 | 5 4 | 0] |
|---------|------------------|------------------|-------------------|
| 1 x 1 1 1 0 0 0 1 0 | imm9 | 1 | 1 | Rn | Rt |

**32-bit variant**

Applies when size == 10.

LDR <Wt>, [Xn|SP>, #<simm>!]

**64-bit variant**

Applies when size == 11.

LDR <Xt>, [Xn|SP>, #<simm>!]

**Decode for all variants of this encoding**

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bias(64) offset = SignExtend(imm9, 64);
**Unsigned offset**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x</td>
<td>1 1</td>
<td>0 0</td>
<td>1 0 1</td>
<td>imm12</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**32-bit variant**

Applies when size == 10.

LDR <Wt>, [<Xn|SP>{, #<pimm>}]

**64-bit variant**

Applies when size == 11.

LDR <Xt>, [<Xn|SP>{, #<pimm>}]

**Decode for all variants of this encoding**

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);

**Notes for all encodings**

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly LDR (immediate) on page K1-11593.

**Assembler symbols**

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

**Shared decode for all encodings**

integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;
regsize = if size == '11' then 64 else 32;
integer datasize = 8 « scale;
boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
case c of
  when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
  when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
  when Constraint_UNDEF UNDEFINED;
  when Constraint_NOP EndOfInstruction();

Operation for all encodings

bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

if !postindex then
  address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t, regsize] = ZeroExtend(data, regsize);

if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n, 64] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.167   LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see Load/store addressing modes on page C1-234.

32-bit variant

Applies when \( \text{opc} == 00 \).

\[
\text{LDR} \ <Wt>, \ <\text{label}>
\]

64-bit variant

Applies when \( \text{opc} == 01 \).

\[
\text{LDR} \ <Xt>, \ <\text{label}>
\]

Decode for all variants of this encoding

```python
integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 4;
        signed = TRUE;
    when '11'
        memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

- \(<Wt>\) is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- \(<Xt>\) is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- \(<\text{label}>\) is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```python
bits(64) address = PC[] + offset;
bits(size*8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);
```
case memop of
  when MemOp_LOAD
    data = Mem[address, size, AccType_NORMAL];
    if signed then
      X[t, 64] = SignExtend(data, 64);
    else
      X[t, size*8] = data;
  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.168   LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about memory accesses, see *Load/store addressing modes on page C1-234.*

**32-bit variant**

Applies when \text{size} == 10.

\[
\text{LDR} \ <Wt>, [\langle Xn|SP\rangle, (\langle Wm\rangle|\langle Xm\rangle)\{, \langle \text{extend}\rangle \{\langle \text{amount}\rangle\}\}]
\]

**64-bit variant**

Applies when \text{size} == 11.

\[
\text{LDR} \ <Xt>, [\langle Xn|SP\rangle, (\langle Wm\rangle|\langle Xm\rangle)\{, \langle \text{extend}\rangle \{\langle \text{amount}\rangle\}\}]
\]

**Decode for all variants of this encoding**

\[
\begin{align*}
\text{integer } \text{scale} &= \text{UInt}(\text{size}); \\
\text{if } \text{option}<1> &= '0' \text{ then } \text{UNDEFINED}; \quad \text{// sub-word index} \\
\text{ExtendType} \ \text{extend\_type} &= \text{DecodeRegExtend}(\text{option}); \\
\text{integer } \text{shift} &= \text{if } S == '1' \text{ then } \text{scale} \text{ else } 0;
\end{align*}
\]

**Assembler symbols**

- \text{<Wt>} Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- \text{<Xt>} Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- \text{<Xn|SP>} Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- \text{<Wm>} When \text{option}<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- \text{<Xm>} When \text{option}<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- \text{<extend>} Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when \text{<amount>} is omitted, encoded in the "option" field. It can have the following values:
  - UXTW when \text{option} = 010
  - LSL when \text{option} = 011
  - SXTW when \text{option} = 110
  - SXTX when \text{option} = 111
- \text{<amount>} For the 32-bit variant: is the index shift amount, optional only when \text{<extend>} is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:
  - #0 when \text{S} = 0
  - #2 when \text{S} = 1

\[
\begin{array}{c|c}
\text{size} & \text{opc} \\
1 & x \\
1 & 1 \\
1 & 0 \\
1 & 0 \\
1 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
& & & & & & & \\
\text{16} & 15 & 13 & 12 & 11 & 10 & 9 & 0 \\
\text{size} & \text{opc} & \text{Rm} & \text{option} & \text{S} & 1 & 0 & \text{Rn} & \text{Rt} \\
\end{array}
\]
For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- #0 when $S = 0$
- #3 when $S = 1$

**Shared decode for all encodings**

```c
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
```

**Operation**

```c
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t, regsize] = ZeroExtend(data, regsize);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.169  **LDRAA, LDRAB**

Load Register, with pointer authentication. This instruction authenticates an address from a base register using a modifier of zero and the specified key, adds an immediate offset to the authenticated address, and loads a 64-bit doubleword from memory at this resulting address into a register.

Key A is used for LDRAA, and key B is used for LDRAB.

If the authentication passes, the PE behaves the same as for an LDR instruction. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the base register, unless the pre-indexed variant of the instruction is used. In this case, the address that is written back to the base register does not include the pointer authentication code.

For information about memory accesses, see *Load/store addressing modes* on page C1-234.

**Unscaled offset**

(FEAT_PAuth)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
<td>0 0 0</td>
<td>M S</td>
<td>imm9</td>
<td>W</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**Key A, offset variant**

Applies when \( M == 0 \) \&\& \( W == 0 \).

LDRAA <Xt>, [<Xn|SP>{, #<simm>}]

**Key A, pre-indexed variant**

Applies when \( M == 0 \) \&\& \( W == 1 \).

LDRAA <Xt>, [<Xn|SP>{, #<simm>}]!

**Key B, offset variant**

Applies when \( M == 1 \) \&\& \( W == 0 \).

LDRAB <Xt>, [<Xn|SP>{, #<simm>}]

**Key B, pre-indexed variant**

Applies when \( M == 1 \) \&\& \( W == 1 \).

LDRAB <Xt>, [<Xn|SP>{, #<simm>}]!

**Decode for all variants of this encoding**

```c
if !HavePACExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
boolean wback = (W == '1');
boolean use_key_a = (M == '0');
bits(10) SIb = S:imm9;
bits(64) offset = LSL(SignExtend(SIb, 64), 3);
boolean tag_checked = wback || n != 31;
```

**Assembler symbols**

<Xt>  Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<X|SP> is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> is the optional signed immediate byte offset, a multiple of 8 in the range -4096 to 4088, defaulting to 0 and encoded in the "S:imm9" field as <simm>/8.

**Operation**

```c
bits(64) address;
bits(64) data;
boolean wb_unknown = FALSE;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    address = SP[];
else
    address = X[n, 64];

if use_key_a then
    address = AuthDA(address, X[31, 64], TRUE);
else
    address = AuthDB(address, X[31, 64], TRUE);

if n == 31 then
    CheckSPAlignment();

address = address + offset;
data = Mem[address, 8, AccType_NORMAL];
X[t, 64] = data;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

**Operational Information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.170 LDRB (immediate)

Load Register Byte (immediate) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/store addressing modes on page C1-234.

Post-index

\[
\begin{array}{c|c|c|c}
| & | & \\
\hline
0 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 \\
| & 1 & 1 & Rn \\
| & 0 & 1 & Rt \\
\end{array}
\]

Encoding

LDRB \(<Wt>, [<Xn|SP>], #<simm>\)

Decode for this encoding

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);

Pre-index

\[
\begin{array}{c|c|c|c}
| & | & \\
\hline
0 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 \\
| & 1 & 1 & Rn \\
| & 0 & 1 & Rt \\
\end{array}
\]

Encoding

LDRB \(<Wt>, [<Xn|SP>, #<simm>]\)

Decode for this encoding

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);

Unsigned offset

\[
\begin{array}{c|c|c|c}
| & | & \\
\hline
0 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 \\
| & 1 & 0 & Rn \\
| & 0 & 1 & Rt \\
\end{array}
\]

Encoding

LDRB \(<Wt>, [<Xn|SP>{, #<pimm}>\])

Decode for this encoding

boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly LDRB (immediate) on page K1-11594.

Assembler symbols

\(<Wt>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<\text{sim}\>) Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

\(<\text{pimm}\>) Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared decode for all encodings

```c
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
    when Constraint_UNKNOWN wb_unknown = TRUE;    // writeback is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

Operation for all encodings

```c
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

if !postindex then
  address = address + offset;

if n == 31 then
  address = Mem[address, 1, AccType_NORMAL];
  X[t, 32] = ZeroExtend(data, 32);

if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elseif postindex then
    address = address + offset;
  elseif n == 31 then
    SP[] = address;
```
else
  \( x[n, 64] = \text{address}; \)

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.171   LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/store addressing modes on page C1-234.

Extended register variant

Applies when option \(!= 011\).

LDRB \(<Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> \{<amount>\}]

Shifted register variant

Applies when option \(== 011\).

LDRB \(<Wt>, [<Xn|SP>, <Xm>{, LSL <amount}>]

Decode for all variants of this encoding

if option<1> == '0' then UNDEFINED;    // sub-word index

ExtendType extend_type = DecodeRegExtend(option);  

Assembler symbols

\(<Wt>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
\(<Wm>\) When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
\(<Xm>\) When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
\(<extend>\) Is the index extend specifier, encoded in the "option" field. It can have the following values:
UXTW when option = 010
SXTW when option = 110
SXTX when option = 111
\(<amount>\) Is the index shift amount, it must be \#0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared decode for all encodings

integer n =.UInt(Rn);
integer t =(UInt(Rt);
integer m = UInt(Rm);
**Operation**

```plaintext
bits(64) offset = ExtendReg(m, extend_type, 0, 64);
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t, 32] = ZeroExtend(data, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.172   LDRH (immediate)

Load Register Halfword (immediate) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/store addressing modes on page C1-234.

**Post-index**

\[
\begin{array}{c|c|c|c|c|c}
& \text{size} & \text{opc} & \text{imm9} & 0 & 1 \\
\end{array}
\]

**Encoding**

LDRH <Wt>, [<Xn|SP>], #<simm>

**Decode for this encoding**

```java
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

**Pre-index**

\[
\begin{array}{c|c|c|c|c|c}
& \text{size} & \text{opc} & \text{imm9} & 1 & 1 \\
\end{array}
\]

**Encoding**

LDRH <Wt>, [<Xn|SP>, #<simm>]

**Decode for this encoding**

```java
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

**Unsigned offset**

\[
\begin{array}{c|c|c|c|c|c}
& \text{size} & \text{opc} & \text{mm12} & & \\
\end{array}
\]

**Encoding**

LDRH <Wt>, [<Xn|SP>{{, #<pimm>}}]

**Decode for this encoding**

```java
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```
Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly LDRH (immediate) on page K1-11594.

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;
Constraint c;
if wback && n == t && n != 31 then
  c = ConstrainsUnpredictable(Unpredictable_WBOVERLAPLD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
    when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

Operation for all encodings

```
bits(64) address;
bits(16) data;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];
if !postindex then
  address = address + offset;
data = Mem[address, 2, AccType_NORMAL];
X[t, 32] = ZeroExtend(data, 32);
if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
```
else
    \( x[n, 64] = \text{address}; \)

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.173  LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/store addressing modes on page C1-234.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>15 14 13 12 11 10  9</th>
<th>5  4  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

LDRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

**Decode for this encoding**

if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;

**Assembler symbols**

- `<Wt>`: Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Wm>`: When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<Xm>`: When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<extend>`: Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:
  - UXTW when option = 010
  - LSL when option = 011
  - SXTW when option = 110
  - SXTX when option = 111
- `<amount>`: Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:
  - #0 when S = 0
  - #1 when S = 1

**Shared decode for all encodings**

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
Operation

bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.174   LDRSB (immediate)

Load Register Signed Byte (immediate) loads a byte from memory, sign-extends it to either 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/store addressing modes on page C1-234.

**Post-index**

<table>
<thead>
<tr>
<th>32-bit variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applies when opc == 11.</td>
</tr>
<tr>
<td>LDRSB &lt;Wt&gt;, [&lt;Xn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64-bit variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applies when opc == 10.</td>
</tr>
<tr>
<td>LDRSB &lt;Xt&gt;, [&lt;Xn</td>
</tr>
</tbody>
</table>

**Decode for all variants of this encoding**

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);

**Pre-index**

<table>
<thead>
<tr>
<th>32-bit variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applies when opc == 11.</td>
</tr>
<tr>
<td>LDRSB &lt;Wt&gt;, [&lt;Xn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64-bit variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applies when opc == 10.</td>
</tr>
<tr>
<td>LDRSB &lt;Xt&gt;, [&lt;Xn</td>
</tr>
</tbody>
</table>

**Decode for all variants of this encoding**

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
Unsigned offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 0 1</td>
<td>1 x</td>
<td>imm12</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
</tr>
</tbody>
</table>

32-bit variant
Applies when opc == 11.
LDRSB <Wt>, [<Xn|SP>{, #<pimm>}]

64-bit variant
Applies when opc == 10.
LDRSB <Xt>, [<Xn|SP>{, #<pimm>}]

Decode for all variants of this encoding
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);

Notes for all encodings
For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly LDRSB (immediate) on page K1-11594.

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

Constraint c;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLPLD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE;  // writeback is suppressed
    when Constraint_UNKNOWN          wb_unknown = TRUE;  // writeback is UNKNOWN
    when Constraint_UNDEF           UNDEFINED;
    when Constraint_NOP             EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLPSST);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE rt_unknown = FALSE;  // value stored is original value
    when Constraint_UNKNOWN           rt_unknown = TRUE;  // value stored is UNKNOWN
    when Constraint_UNDEF             UNDEFINED;
    when Constraint_NOP               EndOfInstruction();

Operation for all encodings

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

if !postindex then
  address = address + offset;

case memop of
  when MemOp_STORE
    if rt_unknown then
      data = bits(8) UNKNOWN;
    else
      data = X[t, 8];
    Mem[address, 1, AccType_NORMAL] = data;
  when MemOp_LOAD
    data = Mem[address, 1, AccType_NORMAL];
    if signed then
      X[t, regsize] = SignExtend(data, regsize);
    else
      X[t, regsize] = ZeroExtend(data, regsize);
  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);

if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
else
    x[n, 64] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.175  **LDRSB (register)**

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/store addressing modes on page C1-234.*

32-bit with extended register offset variant

Applies when `opc == 11 && option != 011`.

LDRSB `<Wt>`, `<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}`

32-bit with shifted register offset variant

Applies when `opc == 11 && option == 011`.

LDRSB `<Wt>`, `<Xn|SP>, <Xm>{, LSL <amount>}`

64-bit with extended register offset variant

Applies when `opc == 10 && option != 011`.

LDRSB `<Xt>`, `<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}`

64-bit with shifted register offset variant

Applies when `opc == 10 && option == 011`.

LDRSB `<Xt>`, `<Xn|SP>, <Xm>{, LSL <amount>}`

**Decode for all variants of this encoding**

If `option<1> == '0'` then UNDEFINED;  // sub-word index

`ExtendType extend_type = DecodeRegExtend(option);`

**Assembler symbols**

- `<Wt>`  
  Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

- `<Xt>`  
  Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

- `<Xn|SP>`  
  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

- `<Wm>`  
  When `option<0>` is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

- `<Xm>`  
  When `option<0>` is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

- `<extend>`  
  Is the index extend specifier, encoded in the "option" field. It can have the following values:

  - `UXTW` when `option = 010`
  - `SXTW` when `option = 110`
  - `SXTX` when `option = 111`

- `<amount>`  
  Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.
Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH;

Operation

bits(64) offset = ExtendReg(m, extend_type, 0, 64);
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

address = address + offset;

case memop of
  when MemOp_STORE
    data = X[t, 8];
    Mem[address, 1, AccType_NORMAL] = data;

  when MemOp_LOAD
    data = Mem[address, 1, AccType_NORMAL];
    if signed then
      X[t, regsize] = SignExtend(data, regsize);
    else
      X[t, regsize] = ZeroExtend(data, regsize);

  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.176   LDRSH (immediate)

Load Register Signed Halfword (immediate) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/store addressing modes on page C1-234.

Post-index

| [31 30 29 28|27 26 25 24|23 22 21 20] | 12|11 10 9 | 5 4 | 0 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | x | 0 | imm9 | 0 | 1 | Rn | Rt |

32-bit variant

Applies when opc == 11.

LDRSH <Wt>, [<Xn|SP>], #<simm>

64-bit variant

Applies when opc == 10.

LDRSH <Xt>, [<Xn|SP>], #<simm>

Decode for all variants of this encoding

```java
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

| [31 30 29 28|27 26 25 24|23 22 21 20] | 12|11 10 9 | 5 4 | 0 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | x | 0 | imm9 | 1 | 1 | Rn | Rt |

32-bit variant

Applies when opc == 11.

LDRSH <Wt>, [<Xn|SP>], #<simm>!

64-bit variant

Applies when opc == 10.

LDRSH <Xt>, [<Xn|SP>], #<simm>!

Decode for all variants of this encoding

```java
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```
Unsigned offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 0</td>
<td>0 1 x</td>
<td>imm12</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
</tr>
</tbody>
</table>

32-bit variant

Applies when opc == 11.
LDRSH <Wt>, [<Xn|SP>{, #<pimm>}]

64-bit variant

Applies when opc == 10.
LDRSH <Xt>, [<Xn|SP>{, #<pimm>}]

Decode for all variants of this encoding

boolean wback = FALSE;
boolean postindex = FALSE;
b bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly LDRSH (immediate) on page K1-11595.

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;
Constraint c;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

Operation for all encodings

bits(64) address;
bits(16) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(16) UNKNOWN;
        else
            data = X[t, 16];
        Mem[address, 2, AccType_NORMAL] = data;
        when MemOp_LOAD
            data = Mem[address, 2, AccType_NORMAL];
            if signed then
                X[t, regsize] = SignExtend(data, regsize);
            else
                X[t, regsize] = ZeroExtend(data, regsize);
        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
else
    \( x[n, 64] = \text{address}; \)

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.177 LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses see Load/store addressing modes on page C1-234.

32-bit variant

Applies when \( \text{opc} == 11 \).

\[ \text{LDRSH} <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}] \]

64-bit variant

Applies when \( \text{opc} == 10 \).

\[ \text{LDRSH} <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}] \]

Decode for all variants of this encoding

\[
\begin{align*}
\text{if } \text{option}<1> == '0' \text{ then UNDEFINED; } & \quad / / \text{ sub-word index} \\
\text{ExtendType } \text{extend_type} = \text{DecodeRegExtend(option);} \\
\text{integer shift} = \text{if } S == '1' \text{ then 1 else 0;}
\end{align*}
\]

Assembler symbols

\(<Wt>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xt>\) Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<Wm>\) When \( \text{option}<0> \) is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

\(<Xm>\) When \( \text{option}<0> \) is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

\(<\text{extend}>\) Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when \( <\text{amount}> \) is omitted. encoded in the "option" field. It can have the following values:

\begin{align*}
\text{UXTW} & \text{ when } \text{option} = 010 \\
\text{LSL} & \text{ when } \text{option} = 011 \\
\text{SXTW} & \text{ when } \text{option} = 110 \\
\text{SXTX} & \text{ when } \text{option} = 111
\end{align*}

\(<\text{amount}>\) Is the index shift amount, optional only when \( <\text{extend}> \) is not LSL. Where it is permitted to be optional, it defaults to \#0. It is encoded in the "S" field. It can have the following values:

\begin{align*}
\#0 & \text{ when } S = 0 \\
\#1 & \text{ when } S = 1
\end{align*}
Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
   // store or zero-extending load
   memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
   regsize = 32;
   signed = FALSE;
else
   // sign-extending load
   memop = MemOp_LOAD;
   regsize = if opc<0> == '1' then 32 else 64;
   signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH;

Operation

bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
   SetTagCheckedInstruction(tag_checked);

if n == 31 then
   if memop != MemOp_PREFETCH then CheckSPAlignment();
   address = SP[];
else
   address = X[n, 64];

address = address + offset;

case memop of
when MemOp_STORE
   data = X[t, 16];
   Mem[address, 2, AccType_NORMAL] = data;

when MemOp_LOAD
   data = Mem[address, 2, AccType_NORMAL];
   if signed then
      X[t, regsize] = SignExtend(data, regsize);
   else
      X[t, regsize] = ZeroExtend(data, regsize);

when MemOp_PREFETCH
   Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.178   LDRSW (immediate)

Load Register Signed Word (immediate) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/store addressing modes on page C1-234.

Post-index

```
| 31 30 29 28 27 26 25 24 23 22 21 20 | | 12 11 10 9 | 5 4 | 0 |
|------------------------------------|-----------------|------|
| 1 0 1 1 0 0 0 1 0 0 | imm9 | 0 | 1 | Rn | Rt |
```

**Encoding**

LDRSw <Xt>, [<Xn|SP>], #<simm>

**Decode for this encoding**

```java
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

```
| 31 30 29 28 27 26 25 24 23 22 21 20 | | 12 11 10 9 | 5 4 | 0 |
|------------------------------------|-----------------|------|
| 1 0 1 1 0 0 0 1 0 0 | imm9 | 1 | 1 | Rn | Rt |
```

**Encoding**

LDRSw <Xt>, [<Xn|SP>, #<simm>]

**Decode for this encoding**

```java
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

```
| 31 30 29 28 27 26 25 24 23 22 21 | | 10 9 | 5 4 | 0 |
|---------------------------------|-----------------|------|
| 1 0 1 1 0 0 1 1 0 | imm12 | Rn | Rt |
```

**Encoding**

LDRSw <Xt>, [<Xn|SP>]{, #<pimm>}

**Decode for this encoding**

```java
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 2);
```
Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly LDRSW (immediate) on page K1-11595.

Assembler symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = wback || n != 31;
boolean wb_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
    when Constraint_UNKNOWN wb_unknown = TRUE;  // writeback is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

Operation for all encodings

bits(64) address;
bits(32) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

if !postindex then
  address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t, 64] = SignExtend(data, 64);
if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
else
    x[n, 64] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.179 LDRSW (literal)

Load Register Signed Word (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see Load/store addressing modes on page C1-234.

Encoding

LDRSw <Xt>, <label>

Decode for this encoding

integer t = UInt(Rt);
bits(64) offset;
offset = SignExtend(imm19:'00', 64);

Assembler symbols

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

bits(64) address = PC[] + offset;
bits(32) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);
data = Mem[address, 4, AccType_NORMAL];
X[t, 64] = SignExtend(data, 64);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.180  **LDRSW (register)**

Load Register Signed Word (register) calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about memory accesses, see *Load/store addressing modes on page C1-234*.

[![Instruction Format](https://example.com/instruction_format.png)](https://example.com/instruction_format.png)

**Encoding**

\[ \text{LDRSW} \ <Xt>, \ [\langle Xn|\text{SP}\rangle, \ (<Wm>|<Xm>)\{, \ <\text{extend}\} \{<\text{amount}\}\}] \]

**Decode for this encoding**

\[
\begin{align*}
\text{if } \text{option}<1> &= '0' \text{ then UNDEFINED;} \quad \text{// sub-word index} \\
\text{ExtendType} \ \text{extend\_type} &= \text{DecodeRegExtend(option);} \\
\text{integer} \ \text{shift} &= \text{if } S \text{ == '1'} \text{ then } 2 \text{ else } 0;
\end{align*}
\]

**Assembler symbols**

- `<Xt>`: Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Wm>`: When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<Xm>`: When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<extend>`: Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when `<amount>` is omitted. encoded in the "option" field. It can have the following values:
  - UXTW: when option = 010
  - LSL: when option = 011
  - SXTW: when option = 110
  - SXTX: when option = 111
- `<amount>`: Is the index shift amount, optional only when `<extend>` is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:
  - #0: when S = 0
  - #2: when S = 1

**Shared decode for all encodings**

\[
\begin{align*}
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } t &= \text{UInt}(Rt); \\
\text{integer } m &= \text{UInt}(Rm);
\end{align*}
\]
Operation

bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(32) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 4, AccType.NORMAL];
X[t, 64] = SignExt(data, 64);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.181  **LDSETB, LDSETAB, LDSETALB, LDSETLB**

Atomic bit set on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAB and LDSETALB load from memory with acquire semantics.
- LDSETLB and LDSETALB store to memory with release semantics.
- LDSETB has neither acquire nor release semantics.


For information about memory accesses see *Load/store addressing modes* on page C1-234.

This instruction is used by the alias STSETB, STSETLB. See *Alias conditions* on page C6-1615 for details of when each alias is preferred.

**Integer**

(FEAT_LSE)

```
  0  0  1  1  0  0  0 A R  1  Rs  0  0  1  1  0  0  Rn  |  Rt |
```

- **LDSETAB variant**
  Applies when \( A = 1 \land R = 0 \).
  LDSETAB \(<Ws>, <Wt>, [<Xn|SP>]\)

- **LDSETALB variant**
  Applies when \( A = 1 \land R = 1 \).
  LDSETALB \(<Ws>, <Wt>, [<Xn|SP>]\)

- **LDSETB variant**
  Applies when \( A = 0 \land R = 0 \).
  LDSETB \(<Ws>, <Wt>, [<Xn|SP>]\)

- **LDSETLB variant**
  Applies when \( A = 0 \land R = 1 \).
  LDSETLB \(<Ws>, <Wt>, [<Xn|SP>]\)

**Decode for all variants of this encoding**

```
if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSETB, STSETLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws>  Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt>  Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) value;
bits(8) data;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
value = X[s, 8];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];
data = MemAtomic(address, MemAtomicOp_ORR, value, ldacctype, stacctype);
if t != 31 then
  X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.182  **LDSETH, LDSETAH, LDSETALH, LDSETLH**

Atomic bit set on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAH and LDSETALH load from memory with acquire semantics.
- LDSETLH and LDSETALH store to memory with release semantics.
- LDSETH has neither acquire nor release semantics.


For information about memory accesses see *Load/store addressing modes* on page C1-234.

This instruction is used by the alias STSETH, STSETLH. See *Alias conditions* on page C6-1617 for details of when each alias is preferred.

**Integer**

(FEAT_LSE)

```
[31 30 29 28|27 26 25 24|23 22 21 20]  16|15 14 12|11 10 9 |  5 4 |  0 |
  0  1  1  1  0  0  0  A  R  | Rn  0  0  1  1  0  0  | Rt

size opc
```

**LDSETAH variant**

Applies when $A == 1 \&\& R == 0$.

LDSETAH <Ws>, <Wt>, [<Xn|SP>]

**LDSETALH variant**

Applies when $A == 1 \&\& R == 1$.

LDSETALH <Ws>, <Wt>, [<Xn|SP>]

**LDSETH variant**

Applies when $A == 0 \&\& R == 0$.

LDSETH <Ws>, <Wt>, [<Xn|SP>]

**LDSETLH variant**

Applies when $A == 0 \&\& R == 1$.

LDSETLH <Ws>, <Wt>, [<Xn|SP>]

**Decode for all variants of this encoding**

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' \&\& Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSETH, STSETLH</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```plaintext
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp.ORR, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.183 LDSET, LDSETA, LDSETAL, LDSETL

Atomic bit set on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSETA and LDSETAL load from memory with acquire semantics.
- LDSETL and LDSETAL store to memory with release semantics.
- LDSET has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is used by the alias STSET, STSETL. See Alias conditions on page C6-1619 for details of when each alias is preferred.

### Integer
(FEAT_LSE)

```
|31 30 29 28|27 26 25 24|23 22 21 20|16|15|14|12|11|10|9|5|4|0|
|1 x|1 1 1 0|0 0 0 A R|1|Rs|0 0|1 1|0 0|Rn|Rt|
```

### 32-bit LDSET variant
Applies when size == 10 && A == 0 && R == 0.

LDSET <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDSETA variant
Applies when size == 10 && A == 1 && R == 0.

LDSETA <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDSETAL variant
Applies when size == 10 && A == 1 && R == 1.

LDSETAL <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDSETL variant
Applies when size == 11 && A == 0 && R == 1.

LDSETL <Ws>, <Wt>, [<Xn|SP>]

### 32-bit LDSETA variant
Applies when size == 11 && A == 1 && R == 0.

LDSETA <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDSET variant
Applies when size == 11 && A == 0 && R == 0.

LDSET <Xs>, <Xt>, [<Xn|SP>]

### 64-bit LDSETA variant
Applies when size == 11 && A == 1 && R == 0.

LDSETA <Xs>, <Xt>, [<Xn|SP>]
64-bit LDSETAL variant
Applies when size == 11 && A == 1 && R == 1.
LDSETAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSETL variant
Applies when size == 11 && A == 0 && R == 1.
LDSETL <Xs>, <Xt>, [<Xn|SP>]

Decode for all variants of this encoding
if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Alias conditions

Aliasing is preferred when
STSET, STSETL  A == '0' && Rt == '11111'

Assembler symbols

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation
bits(64) address;
bits(datasize) value;
bits(datasize) data;
if HaveMTE2Ext() then
SetTagCheckedInstruction(tag_checked);
value = X[s, datasize];
if n == 31 then
CheckS ThảoAlignment();
address = SP[];
else
address = X[n, 64];
data = MemAtomic(address, MemAtomicOp_ORR, value, ldacctype, stacctype);
if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.184 LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAB and LDSMAXALB load from memory with acquire semantics.
- LDSMAXLB and LDSMAXALB store to memory with release semantics.
- LDSMAXB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is used by the alias STSMAXB, STSMAXLB. See Alias conditions on page C6-1622 for details of when each alias is preferred.

**Integer**

(FEAT_LSE)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 0 A R 1</td>
<td>Rs 0 1 0 0 0</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LDSMAXAB variant**

Applies when A == 1 && R == 0.
LDSMAXAB <Ws>, <Wt>, [<Xn|SP>]

**LDSMAXALB variant**

Applies when A == 1 && R == 1.
LDSMAXALB <Ws>, <Wt>, [<Xn|SP>]

**LDSMAXB variant**

Applies when A == 0 && R == 0.
LDSMAXB <Ws>, <Wt>, [<Xn|SP>]

**LDSMAXLB variant**

Applies when A == 0 && R == 1.
LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]

**Decode for all variants of this encoding**

```plaintext
if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMAXB, STSMAXLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
   SetTagCheckedInstruction(tag_checked);

value = X[s, 8];
if n == 31 then
   CheckSPAignment();
   address = SP[];
else
   address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);

if t != 31 then
   X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.185   LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not \texttt{WZR}, LDSMAXAH and LDSMAXALH load from memory with acquire semantics.
- LDSMAXLH and LDSMAXALH store to memory with release semantics.
- LDSMAXH has neither acquire nor release semantics.

For more information about memory ordering semantics see \textit{Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.}

For information about memory accesses see \textit{Load/store addressing modes on page C1-234.}

This instruction is used by the alias STSMAXH, STSMAXLH. See \textit{Alias conditions on page C6-1624} for details of when each alias is preferred.

\textbf{Integer}

(FEAT\_LSE)

\begin{verbatim}
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 12 11 10 9</th>
<th>8 7 5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A</td>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{LDSMAXAH variant}

Applies when $A = 1$ \&\& $R = 0$.

LDSMAXH $<\texttt{Ws}>, <\texttt{Wt}>, [<\texttt{Xn}|\texttt{SP}>]$

\textbf{LDSMAXALH variant}

Applies when $A = 1$ \&\& $R = 1$.

LDSMAXLH $<\texttt{Ws}>, <\texttt{Wt}>, [<\texttt{Xn}|\texttt{SP}>]$

\textbf{LDSMAXH variant}

Applies when $A = 0$ \&\& $R = 0$.

LDSMAXH $<\texttt{Ws}>, <\texttt{Wt}>, [<\texttt{Xn}|\texttt{SP}>]$

\textbf{LDSMAXLH variant}

Applies when $A = 0$ \&\& $R = 1$.

LDSMAXLH $<\texttt{Ws}>, <\texttt{Wt}>, [<\texttt{Xn}|\texttt{SP}>]$

\textbf{Decode for all variants of this encoding}

\begin{verbatim}
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if $A == '1'$ \&\& Rt != '11111' then AccType\_ORDEREDATOMICRW else AccType\_ATOMICRW;
AccType stacctype = if R == '1' then AccType\_ORDEREDATOMICRW else AccType\_ATOMICRW;

boolean tag_checked = n != 31;
\end{verbatim}
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMAXH, STSMAXLH</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

\(<W_s>\) Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

\(<W_t>\) Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

\(<X_n|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

\[
\begin{align*}
\text{bits(64) address;} \\
\text{bits(16) value;} \\
\text{bits(16) data;} \\
\text{if HaveMTE2Ext() then} \\
\qquad \text{SetTagCheckedInstruction(tag_checked);} \\
\text{value} = X[s, 16]; \\
\text{if n == 31 then} \\
\qquad \text{CheckSPAlignment();} \\
\text{else} \\
\qquad \text{address} = X[n, 64]; \\
\text{data} = \text{MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);} \\
\text{if t != 31 then} \\
\qquad X[t, 32] = \text{ZeroExtend(data, 32)};
\end{align*}
\]

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.186  **LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL**

Atomic signed maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSMAXA and LDSMAXAL load from memory with acquire semantics.
- LDSMAX and LDSMAXAL store to memory with release semantics.
- LDSMAX has neither acquire nor release semantics.


For information about memory accesses see *Load/store addressing modes* on page C1-234.

This instruction is used by the alias STSMAX, STSMAXL. See *Alias conditions* on page C6-1626 for details of when each alias is preferred.

### Integer

**(FEAT_LSE)**

\[
\begin{array}{ccccccccccccccccccc}
\hline
1 & x & 1 & 1 & 1 & 0 & 0 & 0 & A & R & 1 & Rs & 0 & 1 & 0 & 0 & 0 & Rn & Rt & 0
\end{array}
\]

#### 32-bit LDSMAX variant

Applies when \( \text{size} == 10 \land A == 0 \land R == 0 \).

LDSMAX \(<Ws>, <Wt>, [<Xn|SP>]\)

#### 32-bit LDSMAXA variant

Applies when \( \text{size} == 10 \land A == 1 \land R == 0 \).

LDSMAXA \(<Ws>, <Wt>, [<Xn|SP>]\)

#### 32-bit LDSMAXAL variant

Applies when \( \text{size} == 10 \land A == 1 \land R == 1 \).

LDSMAXAL \(<Ws>, <Wt>, [<Xn|SP>]\)

#### 32-bit LDSMAXL variant

Applies when \( \text{size} == 10 \land A == 0 \land R == 1 \).

LDSMAXL \(<Ws>, <Wt>, [<Xn|SP>]\)

#### 64-bit LDSMAX variant

Applies when \( \text{size} == 11 \land A == 0 \land R == 0 \).

LDSMAX \(<Xs>, <Xt>, [<Xn|SP>]\)

#### 64-bit LDSMAXA variant

Applies when \( \text{size} == 11 \land A == 1 \land R == 0 \).

LDSMAXA \(<Xs>, <Xt>, [<Xn|SP>]\)
64-bit LDSMAXAL variant
Applies when \( \text{size} = 11 \land A = 1 \land R = 1 \).
LDSMAXAL \(<Xs>, <Xt>, [<Xn|SP>]\)

64-bit LDSMAXL variant
Applies when \( \text{size} = 11 \land A = 0 \land R = 1 \).
LDSMAXL \(<Xs>, <Xt>, [<Xn|SP>]\)

Decode for all variants of this encoding
if !HaveAtomicExt() then UNDEFINED;
integer \( t = \text{UInt}(Rt) \);
integer \( n = \text{UInt}(Rn) \);
integer \( s = \text{UInt}(Rs) \);
integer datasize = 8 << \text{UInt}(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType \( \text{ldacctype} = \) if \( A = '1' \land Rt != '11111' \) then AccType\_ORDEREDATOMICRW else AccType\_ATOMICRW;
AccType \( \text{stacctype} = \) if \( R = '1' \) then AccType\_ORDEREDATOMICRW else AccType\_ATOMICRW;
boolean \( \text{tag\_checked} = n \neq 31 \);

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMAX, STSMAXL</td>
<td>( A = '0' \land Rt = '11111' )</td>
</tr>
</tbody>
</table>

Assembler symbols
<\(W_s\)> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<\(W_t\)> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<\(X_s\)> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<\(X_t\)> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<\(X_n|SP\)> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation
bits(64) address;
bits(datasize) value;
bits(datasize) data;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag\_checked);
value = X[s, datasize];
if \( n = 31 \) then
  CheckSPA\_Alignment();
  address = SP[\[];
else
  address = X[n, 64];
data = Mem\_Atomic(address, Mem\_AtomicOp\_SMAX, value, ldacctype, stacctype);
if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.187 LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAB and LDSMINALB load from memory with acquire semantics.
- LDSMINB and LDSMINALB store to memory with release semantics.
- LDSMINB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is used by the alias STSMINB, STSMINLB. See Alias conditions on page C6-1629 for details of when each alias is preferred.

**Integer**

(FEAT_LSE)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 0 0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**LDSMINAB variant**

Applies when $A == 1 \&\& R == 0$.

LDSMINAB <Ws>, <Wt>, [<Xn|SP>]

**LDSMINALB variant**

Applies when $A == 1 \&\& R == 1$.

LDSMINALB <Ws>, <Wt>, [<Xn|SP>]

**LDSMINB variant**

Applies when $A == 0 \&\& R == 0$.

LDSMINB <Ws>, <Wt>, [<Xn|SP>]

**LDSMINLB variant**

Applies when $A == 0 \&\& R == 1$.

LDSMINLB <Ws>, <Wt>, [<Xn|SP>]

**Decode for all variants of this encoding**

```c
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' \&\& Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMINB, STSMINLB</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) value;
bits(8) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);
if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.188 LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAH and LDSMINALH load from memory with acquire semantics.
- LDSMINLH and LDSMINALH store to memory with release semantics.
- LDSMINH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is used by the alias STSMINH, STSMINLH. See Alias conditions on page C6-1631 for details of when each alias is preferred.

**Integer**

(FEAT_LSE)

| [31 30 29 28][27 26 25 24][23 22 21 20] | 16|15 14 | 12|11 10 9 | 5 | 4 | 0 |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 0 | A | R | 1 | Rt |
| 0 | 1 | 0 | 1 | 0 | 0 | Rn |

**LDSMINAH variant**

Applies when A == 1 && R == 0.
LDSMINAH <Ws>, <Wt>, [<Xn|SP>]

**LDSMINALH variant**

Applies when A == 1 && R == 1.
LDSMINALH <Ws>, <Wt>, [<Xn|SP>]

**LDSMINH variant**

Applies when A == 0 && R == 0.
LDSMINH <Ws>, <Wt>, [<Xn|SP>]

**LDSMINLH variant**

Applies when A == 0 && R == 1.
LDSMINLH <Ws>, <Wt>, [<Xn|SP>]

**Decode for all variants of this encoding**

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STS MINH, STS MINLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

value = X[s, 16];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);
if t != 31 then
  X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.189 LDSMIN, LDSMINA, LDSMINAL, LDSMINL

Atomic signed minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSMINA and LDSMINAL load from memory with acquire semantics.
- LDSMIN and LDSMINAL store to memory with release semantics.
- LDSMIN has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is used by the alias STSMIN, STSMINL. See Alias conditions on page C6-1633 for details of when each alias is preferred.

### Integer

**(FEAT_LSE)**

| [31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 | 12|11 10 9 | 5 4 | 0 |
|---|---|---|---|---|---|---|
| 1 | x | 1 | 1 | 0 | 0 | A | R | 1 | Rs | 0 | 1 | 0 | 1 | 0 | 0 | Rn | Rt |

#### 32-bit LDSMIN variant

Applies when size == 10 && A == 0 && R == 0.

LDSMIN <Ws>, <Wt>, [<Xn|SP>]

#### 32-bit LDSMINA variant

Applies when size == 10 && A == 1 && R == 0.

LDSMINA <Ws>, <Wt>, [<Xn|SP>]

#### 32-bit LDSMINAL variant

Applies when size == 10 && A == 1 && R == 1.

LDSMINAL <Ws>, <Wt>, [<Xn|SP>]

#### 32-bit LDSMINL variant

Applies when size == 10 && A == 0 && R == 1.

LDSMINL <Ws>, <Wt>, [<Xn|SP>]

#### 64-bit LDSMIN variant

Applies when size == 11 && A == 0 && R == 0.

LDSMIN <Xs>, <Xt>, [<Xn|SP>]

#### 64-bit LDSMINA variant

Applies when size == 11 && A == 1 && R == 0.

LDSMINA <Xs>, <Xt>, [<Xn|SP>]

#### 64-bit LDSMINL variant

Applies when size == 11 && A == 1 && R == 1.
64-bit LDSMINAL variant

Applies when size == 11 & A == 1 & R == 1.
LDSMINAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMINL variant

Applies when size == 11 & A == 0 & R == 1.
LDSMINL <Xs>, <Xt>, [<Xn|SP>]

Decode for all variants of this encoding

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMIN, STSMINL</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);
if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.190   LDTR

Load Register (unprivileged) loads a word or doubleword from memory, and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/store addressing modes on page C1-234.

<table>
<thead>
<tr>
<th>32-bit variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applies when size == 10.</td>
</tr>
</tbody>
</table>

LDTR <Wt>, [<Xn|SP>{, #<simm>}]

<table>
<thead>
<tr>
<th>64-bit variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applies when size == 11.</td>
</tr>
</tbody>
</table>

LDTR <Xt>, [<Xn|SP>{, #<simm>}]

**Decode for all variants of this encoding**

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

**Assembler symbols**

<Wt>  Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt>  Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm>  Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared decode for all encodings**

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';
user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if user_access_override && (!unpriv_at_el1 || !unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;
integer regsize;
regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;

Operation

bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, datasize DIV 8, acctype];
X[t, regsize] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.191   LDTRB

Load Register Byte (unprivileged) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is \{1, 1\}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/store addressing modes on page C1-234.

| [31 30 29 28|27 26 25 24|23 22 21 20] | 12|11 10 9 | 5 4 | 0 |
| size | opc |

Encoding

LDTRB <Wt>, [<Xn|SP>{, #<simm>}]

Decode for this encoding

bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
t integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !((EL2Enabled()) && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';
user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;
boolean tag_checked = n != 31;

Operation

bits(64) address;
bias(8) data;
if HaveMTE2Ext() then
SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 1, acctype];
X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.192  LDTRH

Load Register Halfword (unprivileged) loads a halfword from memory, zero-extends it, and writes the result to a
register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value
of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the
instruction is executed. For information about memory accesses, see Load/store addressing modes on page C1-234.

Encoding

LDTRH <Wt>, [<Xn|SP>{, #<simm}>]

Decode for this encoding

bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded
in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';
user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;

boolean tag_checked = n != 31;

Operation

bits(64) address;
bits(16) data;
if HaveMTE2Ext() then
SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 2, acctype];
X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.193   LDTRSB

Load Register Signed Byte (unprivileged) loads a byte from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/store addressing modes on page C1-234.

32-bit variant

Applies when opc == 11.
LDTRSB <Wt>, [<Xn|SP>{, #<simm>}]

64-bit variant

Applies when opc == 10.
LDTRSB <Xt>, [<Xn|SP>{, #<simm>}]

Decode for all variants of this encoding

bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV2> == '11';
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);

Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[);
else
  address = X[n, 64];

address = address + offset;

case memop of
  when MemOp_STORE
    data = X[t, 8];
    Mem[address, 1, acctype] = data;
  when MemOp_LOAD
    data = Mem[address, 1, acctype];
    if signed then
      X[t, regsize] = SignExtend(data, regsize);
    else
      X[t, regsize] = ZeroExtend(data, regsize);
  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.194   LDTRSH

Load Register Signed Halfword (unprivileged) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is \{1, 1\}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/store addressing modes on page C1-234.

32-bit variant

Applies when opc == 11.

LDTRSH <Wt>, [<Xn|SP>{, #<simm>}]

64-bit variant

Applies when opc == 10.

LDTRSH <Xt>, [<Xn|SP>{, #<simm>}]

Decode for all variants of this encoding

bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Wt>      Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>      Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>   Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>   Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !ELEnabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11';
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';
user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);

**Operation**

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

address = address + offset;

case memop of
  when MemOp_STORE
    data = X[t, 16];
    Mem[address, 2, acctype] = data;

  when MemOp_LOAD
    data = Mem[address, 2, acctype];
    if signed then
      X[t, regsize] = SignExtend(data, regsize);
    else
      X[t, regsize] = ZeroExtend(data, regsize);

  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.195  LDTRSW

Load Register Signed Word (unprivileged) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/store addressing modes on page C1-234.

Encoding

LDTRSW <Xt>, [<Xn|SP>{, #<simm}>]

Decode for this encoding

bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Xt>  Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';
user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31;

Operation

bits(64) address;
bits(32) data;
if HaveMTE2Ext() then
SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

address = address + offset;

data = Mem[address, 4, acctype];
X[t, 64] = SignExtend(data, 64);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.196   LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB

Atomic unsigned maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against
the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.
The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAB and LDUMAXALB load from memory with acquire semantics.
- LDUMAXLB and LDUMAXALB store to memory with release semantics.
- LDUMAXB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on
page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is used by the alias STUAXMB, STUAMLB. See Alias conditions on page C6-1648 for details
of when each alias is preferred.

**Integer**

(FEAT_LSE)

```
| [31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 12|11 10 9 | 5 4 | 0 |
| 0 0 1 1 | 0 0 A | R | 1 | Rs 0 1 1 0 0 | Rn | Rt |
```

**LDUMAXAB variant**

Applies when A == 1 & R == 0.
LDUMAXAB <Ws>, <Wt>, [<Xn|SP>]

**LDUMAXALB variant**

Applies when A == 1 & R == 1.
LDUMAXALB <Ws>, <Wt>, [<Xn|SP>]

**LDUMAXB variant**

Applies when A == 0 & R == 0.
LDUMAXB <Ws>, <Wt>, [<Xn|SP>]

**LDUMAXLB variant**

Applies when A == 0 & R == 1.
LDUMAXLB <Ws>, <Wt>, [<Xn|SP>]

**Decode for all variants of this encoding**

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

int ldacctype = if A == '1' & R != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
int stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMAXB, STUMAXLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.197  **LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH**

Atomic unsigned maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAH and LDUMAXALH load from memory with acquire semantics.
- LDUMAXH and LDUMAXALH store to memory with release semantics.
- LDUMAXH has neither acquire nor release semantics.


For information about memory accesses see *Load/store addressing modes* on page C1-234.

This instruction is used by the alias STUMAXH, STUMAXLH. See *Alias conditions* on page C6-1650 for details of when each alias is preferred.

**Integer**

(Feat_LSE)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LDUMAXAH variant**

Applies when A == 1 \&\& R == 0.

LDUMAXAH <ws>, <wt>, [\<Xn\>|SP>]

**LDUMAXALH variant**

Applies when A == 1 \&\& R == 1.

LDUMAXALH <ws>, <wt>, [\<Xn\>|SP>]

**LDUMAXH variant**

Applies when A == 0 \&\& R == 0.

LDUMAXH <ws>, <wt>, [\<Xn\>|SP>]

**LDUMAXLH variant**

Applies when A == 0 \&\& R == 1.

LDUMAXLH <ws>, <wt>, [\<Xn\>|SP>]

**Decode for all variants of this encoding**

If !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' \&\& Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMAXH, STUMAXLH</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws>  Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt>  Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn\|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.198 LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL

Atomic unsigned maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, LDUMAXA and LDUMAXAL load from memory with acquire semantics.
- LDUMAX and LDUMAXAL store to memory with release semantics.
- LDUMAX has neither acquire nor release semantics.


For information about memory accesses see *Load/store addressing modes* on page C1-234.

This instruction is used by the alias STUMAX, STUMAXL. See *Alias conditions* on page C6-1652 for details of when each alias is preferred.

**Integer**

(FEAT_LSE)

| 31 30 29 28[27 26 25 24][23 22 21 20] | 16|15|14|12|11|10|9 | 5 | 4 | 0 |
|----------------------------------------|---|---|---|---|---|---|---|---|---|
| size opc                               | 0 | 1 | 1 | 1 | 0 | 0 | 0 | A | R | 1 |
| Rs                                      | 0 | 1 | 1 | 0 | 0 | Rn | Rt |

**32-bit LDUMAX variant**

Applies when `size == 10 && A == 0 && R == 0`.

LDUMAX `<ws>`, `<wt>`, `[<Xn|SP>]`

**32-bit LDUMAXA variant**

Applies when `size == 10 && A == 1 && R == 0`.

LDUMAXA `<ws>`, `<wt>`, `[<Xn|SP>]`

**32-bit LDUMAXAL variant**

Applies when `size == 10 && A == 1 && R == 1`.

LDUMAXAL `<ws>`, `<wt>`, `[<Xn|SP>]`

**32-bit LDUMAXL variant**

Applies when `size == 10 && A == 0 && R == 1`.

LDUMAXL `<ws>`, `<wt>`, `[<Xn|SP>]`

**64-bit LDUMAX variant**

Applies when `size == 11 && A == 0 && R == 0`.

LDUMAX `<xs>`, `<xt>`, `[<Xn|SP>]`

**64-bit LDUMAXA variant**

Applies when `size == 11 && A == 1 && R == 0`.

LDUMAXA `<xs>`, `<xt>`, `[<Xn|SP>]`
**64-bit LDUMAXAL variant**

Applies when \( \text{size} == 11 \&\& \text{A} == 1 \&\& \text{R} == 1 \).

LDUMAXAL \(<Xs>, <Xt>, [<Xn|SP>]\)

**64-bit LDUMAXL variant**

Applies when \( \text{size} == 11 \&\& \text{A} == 0 \&\& \text{R} == 1 \).

LDUMAXL \(<Xs>, <Xt>, [<Xn|SP>]\)

**Decode for all variants of this encoding**

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

**Alias conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMAX, STUMAXL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

**Assembler symbols**

- \(<Ws>\)
  - Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

- \(<Wt>\)
  - Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

- \(<Xs>\)
  - Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

- \(<Xt>\)
  - Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

- \(<Xn|SP>\)
  - Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);
```
if \( t \neq 31 \) then
\[
X[t, \text{regsize}] = \text{ZeroExtend}(\text{data}, \text{regsize});
\]

**Operational information**

If \( PSTATE.DIT \) is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.199   LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

Atomic unsigned minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

•   If the destination register is not WZR, LDUMINAB and LDUMINALB load from memory with acquire semantics.
•   LDUMINLB and LDUMINALB store to memory with release semantics.
•   LDUMINB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is used by the alias STUMINB, STUMINLB. See Alias conditions on page C6-1655 for details of when each alias is preferred.

**Integer**

(_FEAT_LSE)

```
[31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 12 10 9 | 5 4 | 0 ]
| 0 0 | 1 1 | 0 0 | A | R | 1 |
| Rs  | 0 1 | 1 0 | 0 | Rn | Rt |
```

**LDUMINAB variant**

Applies when \( A == 1 \) \&\& \( R == 0 \).
LDUMINAB <Ws>, <Wt>, [<Xn|SP>]

**LDUMINALB variant**

Applies when \( A == 1 \) \&\& \( R == 1 \).
LDUMINALB <Ws>, <Wt>, [<Xn|SP>]

**LDUMINB variant**

Applies when \( A == 0 \) \&\& \( R == 0 \).
LDUMINB <Ws>, <Wt>, [<Xn|SP>]

**LDUMINLB variant**

Applies when \( A == 0 \) \&\& \( R == 1 \).
LDUMINLB <Ws>, <Wt>, [<Xn|SP>]

**Decode for all variants of this encoding**

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' \&\& Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMINB, STUMINLB</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.200  **LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH**

Atomic unsigned minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAH and LDUMINALH load from memory with acquire semantics.
- LDUMINLH and LDUMINAH store to memory with release semantics.
- LDUMINH has neither acquire nor release semantics.


For information about memory accesses see *Load/store addressing modes* on page C1-234.

This instruction is used by the alias STUMINH, STUMINLH. See *Alias conditions* on page C6-1657 for details of when each alias is preferred.

**Integer**

(FEAT_LSE)

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20]</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 0 0 0 A R 1 Rs 0 1 1 0 0 Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LDUMINAH variant**

Applies when A == 1 && R == 0.

LDUMINAH <Ws>, <Wt>, [Xn|SP]

**LDUMINALH variant**

Applies when A == 1 && R == 1.

LDUMINALH <Ws>, <Wt>, [Xn|SP]

**LDUMINH variant**

Applies when A == 0 && R == 0.

LDUMINH <Ws>, <Wt>, [Xn|SP]

**LDUMINLH variant**

Applies when A == 0 && R == 1.

LDUMINLH <Ws>, <Wt>, [Xn|SP]

**Decode for all variants of this encoding**

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMINH, STUMINlh</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.201   LDUMIN, LDUMINA, LDUMINAL, LDUMINL

Atomic unsigned minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDUMINA and LDUMINAL load from memory with acquire semantics.
- LDUMINL and LDUMINAL store to memory with release semantics.
- LDUMIN has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is used by the alias STUMIN, STUMINL. See Alias conditions on page C6-1659 for details of when each alias is preferred.

**Integer**

(FEAT_LSE)

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 12|11 10 9 | 5 4 | 0 |
 1 x 1 1 1 0 0 0 A R 1 Rs 0 1 1 0 0 Rn Rt
```

**32-bit LDUMIN variant**

Applies when size == 10 && A == 0 && R == 0.

LDUMIN <Ws>, <Wt>, [<Xn|SP>]

**32-bit LDUMINA variant**

Applies when size == 10 && A == 1 && R == 0.

LDUMINA <Ws>, <Wt>, [<Xn|SP>]

**32-bit LDUMINAL variant**

Applies when size == 10 && A == 1 && R == 1.

LDUMINAL <Ws>, <Wt>, [<Xn|SP>]

**32-bit LDUMINL variant**

Applies when size == 10 && A == 0 && R == 1.

LDUMINL <Ws>, <Wt>, [<Xn|SP>]

**64-bit LDUMIN variant**

Applies when size == 11 && A == 0 && R == 0.

LDUMIN <Xs>, <Xt>, [<Xn|SP>]

**64-bit LDUMINA variant**

Applies when size == 11 && A == 1 && R == 0.

LDUMINA <Xs>, <Xt>, [<Xn|SP>]

**64-bit LDUMINAL variant**

Applies when size == 11 && A == 1 && R == 1.

LDUMINAL <Xs>, <Xt>, [<Xn|SP>]

**64-bit LDUMINL variant**

Applies when size == 11 && A == 0 && R == 1.

LDUMINL <Xs>, <Xt>, [<Xn|SP>]
64-bit LDUMINAL variant

Applies when size == 11 && A == 1 && R == 1.
LDUMINAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDUMINL variant

Applies when size == 11 && A == 0 && R == 1.
LDUMINL <Xs>, <Xt>, [<Xn|SP>]

Decode for all variants of this encoding

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMIN, STUMINL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
value = X[s, datasize];
if n == 31 then
    CheckSPAAlignment();
    address = SP[];
else
    address = X[n, 64];
data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);
if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.202  LDUR

Load Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/store addressing modes on page C1-234.

### 32-bit variant

Applies when size == 10.

LDUR <Wt>, [<Xn|SP>{, #<simm>}]

### 64-bit variant

Applies when size == 11.

LDUR <Xt>, [<Xn|SP>{, #<simm>}]

#### Decode for all variants of this encoding

integer scale = UInt(size);

bits(64) offset = SignExtend(imm9, 64);

#### Assembler symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

#### Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;

#### Operation

bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t, regsize] = ZeroExtend(data, regsize);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.203  LDURB

Load Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/store addressing modes on page C1-234.

```
[31 30 29 28][27 26 25 24][23 22 21 20] | 12|11 10 9 | 5 4 | 0 |
0  0  1  1 0  0  0  0  0  1 0  imm9  0  0  Rn  Rt
size opc
```

**Encoding**

```
LDURB <Wt>, [<Xn|SP>{, #<simm>}]   
```

**Decoding for this encoding**

```
bits(64) offset = SignExtend(imm9, 64);
```

**Assembler symbols**

- `<Wt>`  Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared decode for all encodings**

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

**Operation**

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t, 32] = ZeroExtend(data, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.204   LDURH

Load Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/store addressing modes on page C1-234.

Encoding

LDURH <Wt>, [{<Xn|SP>}, #<simm>]

Decide for this encoding

bits(64) offset = SignExtend(imm9, 64);

Assemblers symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Operation

bits(64) address;
bites(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

    address = address + offset;

    data = Mem[address, 2, AccType_NORMAL];
    X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.205   LDURSB

Load Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a
signed byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see
Load/store addressing modes on page C1-234.

32-bit variant

Applies when opc == 11.
LDURSB <Wt>, [<Xn|SP>{, #<simm>}]

64-bit variant

Applies when opc == 10.
LDURSB <Xt>, [<Xn|SP>{, #<simm>}]

Decode for all variants of this encoding

bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded
in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then  // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<8> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t, 8];
        Mem[address, 1, AccType_NORMAL] = data;
    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.206   LDURSH

Load Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see Load/store addressing modes on page C1-234.

32-bit variant

Applies when opc == 11.

LDURSH <Wt>, [<Xn|SP>{, #<simm>}]

64-bit variant

Applies when opc == 10.

LDURSH <Xt>, [<Xn|SP>{, #<simm>}]

Decode for all variants of this encoding

bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
Operation

\[
\text{bits(64) address; } \\
\text{bits(16) data;}
\]

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t, 16];
        Mem[address, 2, AccType_NORMAL] = data;
    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>); \\

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.207   LDURSW

Load Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/store addressing modes on page C1-234.*

| [31 30 29 28|27 26 25 24|23 22 21 20] | 12|11 10 9 | 5 | 4 | 0 |
|---------------|-----------------|---|---|---|
| 1 0 | 1 1 | 0 | 0 | 1 0 | 0 | imm9 | 0 0 | Rn | Rt |

**Encoding**

LDURSW <Xt>, [<Xn|SP>{, #<simm>}]

**Decode for this encoding**

bits(64) offset = SignExtend(imm9, 64);

**Assembler symbols**

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared decode for all encodings**

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

**Operation**

bits(64) address;
bits(32) data;

if HaveMTE2Ext() then
   SetTagCheckedInstruction(tag_checked);

if n == 31 then
   CheckSPAlignment();
   address = SP[];
else
   address = X[n, 64];

address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t, 64] = SignExtend(data, 64);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.208   **LDXP**

Load Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and alignment requirements on page B2-154 and Alignment of data accesses on page B2-189. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores on page B2-211. For information about memory accesses, see Load/store addressing modes on page C1-234.

### 32-bit variant

Applies when \( sz == 0 \).

\[
\text{LDXP} \ <W_{t1}> , \ <W_{t2}> , \ [<X_n|SP\{,#0}] \\
\]

### 64-bit variant

Applies when \( sz == 1 \).

\[
\text{LDXP} \ <X_{t1}> , \ <X_{t2}> , \ [<X_n|SP\{,#0}] \\
\]

**Decode for all variants of this encoding**

```c
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
if (t == t2)
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

**Notes for all encodings**

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly LDXP on page K1-11595.

**Assembler symbols**

- \( <W_{t1}> \) Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- \( <W_{t2}> \) Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- \( <X_{t1}> \) Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

```plaintext
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t, datasize] = bits(datasize) UNKNOWN;    // In this case t = t2
elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, AccType_ATOMIC];
    if BigEndian(AccType_ATOMIC) then
        X[t, datasize-elsize] = data<datasize-1:elsize>;
        X[t2, elsize] = data<elsize-1:0>;
    else
        X[t, elsize] = data<elsize-1:0>;
        X[t2, datasize-elsize] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
        AArch64.Abort(address, AlignmentFault(AccType_ATOMIC, FALSE, FALSE));
    X[t, 64] = Mem[address, 8, AccType_ATOMIC];
    X[t2, 64] = Mem[address+8, 8, AccType_ATOMIC];
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.209   LDXR

Load Exclusive Register derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores on page B2-211. For information about memory accesses see Load/store addressing modes on page C1-234.

32-bit variant

Applies when size == 10.

LDXR <Wt>, [<Xn|SP>{,#0}]

64-bit variant

Applies when size == 11.

LDXR <Xt>, [<Xn|SP>{,#0}]

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer elsize = 8L UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;
```

Assembler symbols

<Wt> is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[0];
else
    address = X[n, 64];
// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
```
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

data = Mem[address, dbytes, AccType_ATOMIC];
X[t, regsize] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.210   LDXRB

Load Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-211. For information about memory accesses see *Load/store addressing modes* on page C1-234.

**Encoding**

LDXRB <Wt>, [<Xn|SP>{,#0}]

**Decode for this encoding**

integer $n = \text{UInt}(Rn)$;
integer $t = \text{UInt}(Rt)$;

boolean tag_checked = $n \neq 31$;

**Assembler symbols**

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if $n == 31$ then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, AccType_ATOMIC];
X[t, 32] = ZeroExtend(data, 32);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.211   LDXRH

Load Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores on page B2-211. For information about memory accesses see Load/store addressing modes on page C1-234.

Encoding

LDXRH <Wt>, [<Xn|SP>{,#0}]

Decode for this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, AccType_ATOMIC];
X[t, 32] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**C6.2.212  LSL (register)**

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This instruction is an alias of the LSLV instruction. This means that:

- The encodings in this description are named to match the encodings of LSLV.
- The description of LSLV gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when \( sf = 0 \).

\[
\text{LSL} \ <\text{id}> , \ <\text{in}> , \ <\text{im}>
\]

is equivalent to

\[
\text{LSLV} \ <\text{id}> , \ <\text{in}> , \ <\text{im}>
\]

and is always the preferred disassembly.

### 64-bit variant

Applies when \( sf = 1 \).

\[
\text{LSL} \ <\text{Xd}> , \ <\text{Xn}> , \ <\text{Xm}>
\]

is equivalent to

\[
\text{LSLV} \ <\text{Xd}> , \ <\text{Xn}> , \ <\text{Xm}>
\]

and is always the preferred disassembly.

### Assembler symbols

- \(<\text{id}>\): Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<\text{in}>\): Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<\text{im}>\): Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- \(<\text{Xd}>\): Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<\text{Xn}>\): Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<\text{Xm}>\): Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

### Operation

The description of LSLV gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.213   LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when \( sf = 0 \) \&\& \( N = 0 \) \&\& \( \text{imms} \neq 011111 \).

\[
\text{LSL} \ <Wd>, \ <Wn>, \ #<shift>
\]

is equivalent to

\[
\text{UBFM} \ <Wd>, \ <Wn>, \ #(-<shift> \mod 32), \ #(31-<shift>)
\]

and is the preferred disassembly when \( \text{imms} + 1 = \text{immr} \).

### 64-bit variant

Applies when \( sf = 1 \) \&\& \( N = 1 \) \&\& \( \text{imms} \neq 111111 \).

\[
\text{LSL} \ <Xd>, \ <Xn>, \ #<shift>
\]

is equivalent to

\[
\text{UBFM} \ <Xd>, \ <Xn>, \ #(-<shift> \mod 64), \ #(63-<shift>)
\]

and is the preferred disassembly when \( \text{imms} + 1 = \text{immr} \).

### Assembler symbols

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\) Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<\text{shift}>\) For the 32-bit variant: is the shift amount, in the range 0 to 31.
  For the 64-bit variant: is the shift amount, in the range 0 to 63.

### Operation

The description of UBFM gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.214   LSLV

Logical Shift Left Variable shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This instruction is used by the alias LSL (register). The alias is always the preferred disassembly.

### 32-bit variant

Applies when \( sf = 0 \).

\[
\text{LSLV} \ <Wd>, <Wn>, <Wm>
\]

### 64-bit variant

Applies when \( sf = 1 \).

\[
\text{LSLV} \ <Xd>, <Xn>, <Xm>
\]

#### Decode for all variants of this encoding

```cpp
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

#### Assembler symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;Wd&gt;)</td>
<td>Is the 32-bit name of the general-purpose destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>(&lt;Wn&gt;)</td>
<td>Is the 32-bit name of the first general-purpose source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>(&lt;Wm&gt;)</td>
<td>Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the &quot;Rm&quot; field.</td>
</tr>
<tr>
<td>(&lt;Xd&gt;)</td>
<td>Is the 64-bit name of the general-purpose destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>(&lt;Xn&gt;)</td>
<td>Is the 64-bit name of the first general-purpose source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>(&lt;Xm&gt;)</td>
<td>Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the &quot;Rm&quot; field.</td>
</tr>
</tbody>
</table>

#### Operation

```cpp
bits(datasize) result;
bits(datasize) operand2 = X[m, datasize];
result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
X[d, datasize] = result;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.215 LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is an alias of the LSRV instruction. This means that:

- The encodings in this description are named to match the encodings of LSRV.
- The description of LSRV gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when \( sf = 0 \).

\[
\text{LSR } <Wd>, <Wn>, <Wm>
\]

is equivalent to

\[
\text{LSRV } <Wd>, <Wn>, <Wm>
\]

and is always the preferred disassembly.

### 64-bit variant

Applies when \( sf = 1 \).

\[
\text{LSR } <Xd>, <Xn>, <Xm>
\]

is equivalent to

\[
\text{LSRV } <Xd>, <Xn>, <Xm>
\]

and is always the preferred disassembly.

### Assembler symbols

- \(<Wd>\) is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Wm>\) is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- \(<Xd>\) is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\) is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Xm>\) is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

### Operation

The description of LSRV gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.216 LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when $sf == 0 \&\& N == 0 \&\& imms == 011111$.

LSR $<Wd>, <Wn>, #<shift>$

is equivalent to

UBFM $<Wd>, <Wn>, #<shift>, #31$

and is always the preferred disassembly.

### 64-bit variant

Applies when $sf == 1 \&\& N == 1 \&\& imms == 111111$.

LSR $<Xd>, <Xn>, #<shift>$

is equivalent to

UBFM $<Xd>, <Xn>, #<shift>, #63$

and is always the preferred disassembly.

### Assembler symbols

- $<Wd>$: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- $<Wn>$: Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- $<Xd>$: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- $<Xn>$: Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- $<shift>$: For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field.
  
  For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field.

### Operation

The description of UBFM gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.217   LSRV

Logical Shift Right Variable shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias LSR (register). The alias is always the preferred disassembly.

32-bit variant

Applies when \( sf = 0 \).

\[ \text{LSRV} <Wd>, <Wn>, <Wm> \]

64-bit variant

Applies when \( sf = 1 \).

\[ \text{LSRV} <Xd>, <Xn>, <Xm> \]

**Decode for all variants of this encoding**

- \( \text{integer } d = \text{UInt}(Rd); \)
- \( \text{integer } n = \text{UInt}(Rn); \)
- \( \text{integer } m = \text{UInt}(Rm); \)
- \( \text{integer } \text{datasize} = \text{if } sf = '1' \text{ then } 64 \text{ else } 32; \)
- \( \text{ShiftType } \text{shift_type} = \text{DecodeShift}(op2); \)

**Assembler symbols**

- \( <Wd> \) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \( <Wn> \) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \( <Wm> \) Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- \( <Xd> \) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \( <Xn> \) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \( <Xm> \) Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

**Operation**

\[
\text{bits(datasize) result; bits(datasize) operand2 = X[m, datasize];}
\]

\[
\text{result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize); X[d, datasize] = result;}
\]
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.218 MADD

Multiply-Add multiplies two register values, adds a third register value, and writes the result to the destination register.

This instruction is used by the alias MUL. See Alias conditions for details of when each alias is preferred.

32-bit variant
Applies when $sf = 0$.

MADD $<Wd>, <Wn>, <Wm>, <Wa>$

64-bit variant
Applies when $sf = 1$.

MADD $<Xd>, <Xn>, <Xm>, <Xa>$

Decode for all variants of this encoding

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if $sf == '1'$ then 64 else 32;
```

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>$Ra == '11111'$</td>
</tr>
</tbody>
</table>

Assembler symbols

- $<Wd>$ is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- $<Wn>$ is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- $<Wm>$ is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- $<Wa>$ is the 32-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.
- $<Xd>$ is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- $<Xn>$ is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- $<Xm>$ is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

**Operation**

\[
\begin{align*}
\text{bits}(\text{destsize}) \text{ operand1} &= X[n, \text{destsize}] ; \\
\text{bits}(\text{destsize}) \text{ operand2} &= X[m, \text{destsize}] ; \\
\text{bits}(\text{destsize}) \text{ operand3} &= X[a, \text{destsize}] ; \\
\text{integer result} &= ; \\
\text{result} &= \text{UInt}(\text{operand3}) + (\text{UInt}(\text{operand1}) \times \text{UInt}(\text{operand2})) ; \\
X[d, \text{destsize}] &= \text{result}<\text{destsize}-1:0> ;
\end{align*}
\]

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.219   MNEG

Multiply-Negate multiplies two register values, negates the product, and writes the result to the destination register.

This instruction is an alias of the MSUB instruction. This means that:

- The encodings in this description are named to match the encodings of MSUB.
- The description of MSUB gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when \( s_f = 0 \).

\[
\text{MNEG } \langle Wd \rangle, \langle Wn \rangle, \langle Wm \rangle
\]

is equivalent to

\[
\text{MSUB } \langle Wd \rangle, \langle Wn \rangle, \langle Wm \rangle, \text{WZR}
\]

and is always the preferred disassembly.

### 64-bit variant

Applies when \( s_f = 1 \).

\[
\text{MNEG } \langle Xd \rangle, \langle Xn \rangle, \langle Xm \rangle
\]

is equivalent to

\[
\text{MSUB } \langle Xd \rangle, \langle Xn \rangle, \langle Xm \rangle, \text{XZR}
\]

and is always the preferred disassembly.

### Assembler symbols

- \( \langle Wd \rangle \): Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \( \langle Wn \rangle \): Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- \( \langle Wm \rangle \): Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- \( \langle Xd \rangle \): Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \( \langle Xn \rangle \): Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- \( \langle Xm \rangle \): Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

### Operation

The description of MSUB gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.220  MOV (to/from SP)

Move between register and stack pointer: \( R_d = R_n \)

This instruction is an alias of the ADD (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ADD (immediate).
- The description of ADD (immediate) gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when \( sf = 0 \).

MOV \( <W_d|WSP>, <W_n|WSP> \)

is equivalent to

ADD \( <W_d|WSP>, <W_n|WSP>, #0 \)

and is the preferred disassembly when \( (R_d = '11111' || R_n = '11111') \).

### 64-bit variant

Applies when \( sf = 1 \).

MOV \( <X_d|SP>, <X_n|SP> \)

is equivalent to

ADD \( <X_d|SP>, <X_n|SP>, #0 \)

and is the preferred disassembly when \( (R_d = '11111' || R_n = '11111') \).

### Assembler symbols

- \( <W_d|WSP> \) Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- \( <W_n|WSP> \) Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- \( <X_d|SP> \) Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- \( <X_n|SP> \) Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

### Operation

The description of ADD (immediate) gives the operational pseudocode for this instruction.
C6.2.221 MOV (inverted wide immediate)

Move (inverted wide immediate) moves an inverted 16-bit immediate value to a register.

This instruction is an alias of the MOVN instruction. This means that:

- The encodings in this description are named to match the encodings of MOVN.
- The description of MOVN gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when \( sf = 0 \) \&\& \( hw = 0x \).

\[
\text{MOV } <Wd>, \#<imm>
\]

is equivalent to

\[
\text{MOVN } <Wd>, \#<imm16>, \text{LSL}<shift>
\]

and is the preferred disassembly when \( ! (\text{IsZero}(imm16) \&\& \text{hw} \neq '00') \&\& ! \text{IsOnes}(imm16) \).

### 64-bit variant

Applies when \( sf = 1 \).

\[
\text{MOV } <Xd>, \#<imm>
\]

is equivalent to

\[
\text{MOVN } <Xd>, \#<imm16>, \text{LSL}<shift>
\]

and is the preferred disassembly when \( ! (\text{IsZero}(imm16) \&\& \text{hw} \neq '00') \).

### Assembler symbols

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<imm>\) For the 32-bit variant: is a 32-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw", but excluding 0xffff0000 and 0x0000ffff.
  
  For the 64-bit variant: is a 64-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw".
- \(<shift>\) For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as \(<shift>/16\).
  
  For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as \(<shift>/16\).

### Operation

The description of MOVN gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.222 MOV (wide immediate)

Move (wide immediate) moves a 16-bit immediate value to a register.

This instruction is an alias of the MOV instruction. This means that:

- The encodings in this description are named to match the encodings of MOVZ.
- The description of MOVZ gives the operational pseudocode for this instruction.

32-bit variant

Applies when \( sf = 0 \) \&\& \( hw = 0x \).

\[
\text{MOV} \ <Wd>, \ #<imm>
\]

is equivalent to

\[
\text{MOVZ} \ <Wd>, \ #<imm16>, \ LSL \ #<shift>
\]

and is the preferred disassembly when \( \neg (\text{IsZero}(\text{imm16}) \&\& \ text{hw} \neq '00') \).

64-bit variant

Applies when \( sf = 1 \).

\[
\text{MOV} \ <Xd>, \ #<imm>
\]

is equivalent to

\[
\text{MOVZ} \ <Xd>, \ #<imm16>, \ LSL \ #<shift>
\]

and is the preferred disassembly when \( \neg (\text{IsZero}(\text{imm16}) \&\& \ text{hw} \neq '00') \).

Assembler symbols

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<imm>\) For the 32-bit variant: is a 32-bit immediate which can be encoded in "imm16:hw".
  For the 64-bit variant: is a 64-bit immediate which can be encoded in "imm16:hw".
- \(<shift>\) For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as \(<shift>/16\).
  For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as \(<shift>/16\).

Operation

The description of MOVZ gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.223 MOV (bitmask immediate)

Move (bitmask immediate) writes a bitmask immediate value to a register.

This instruction is an alias of the ORR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The description of ORR (immediate) gives the operational pseudocode for this instruction.

32-bit variant

Applies when \( sf == 0 \land N == 0 \).

\[
\text{MOV} \langle \text{Wd|WSP} \rangle, \#\langle \text{imm}\rangle
\]

is equivalent to

\[
\text{ORR} \langle \text{Wd|WSP} \rangle, \text{WZR}, \#\langle \text{imm}\rangle
\]

and is the preferred disassembly when \(! \text{MoveWidePreferred}(sf, N, \text{imms}, \text{immr})\).

64-bit variant

Applies when \( sf == 1 \).

\[
\text{MOV} \langle \text{Xd|SP} \rangle, \#\langle \text{imm}\rangle
\]

is equivalent to

\[
\text{ORR} \langle \text{Xd|SP} \rangle, \text{XZR}, \#\langle \text{imm}\rangle
\]

and is the preferred disassembly when \(! \text{MoveWidePreferred}(sf, N, \text{imms}, \text{immr})\).

Assembler symbols

\( \langle \text{Wd|WSP} \rangle \)  Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

\( \langle \text{Xd|SP} \rangle \)  Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

\( \langle \text{imm}\rangle \)  For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

Operation

The description of ORR (immediate) gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.224 MOV (register)

Move (register) copies the value in a source register to the destination register.

This instruction is an alias of the ORR (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of ORR (shifted register).
- The description of ORR (shifted register) gives the operational pseudocode for this instruction.

32-bit variant

Applies when \( sf = 0 \).

MOV <Wd>, <Wm>

is equivalent to

ORR <Wd>, WZR, <Wm>

and is always the preferred disassembly.

64-bit variant

Applies when \( sf = 1 \).

MOV <Xd>, <Xm>

is equivalent to

ORR <Xd>, XZR, <Xm>

and is always the preferred disassembly.

Assembler symbols

- \(<\text{Wd}>\): Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<\text{Wm}>\): Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- \(<\text{Xd}>\): Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<\text{Xm}>\): Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of ORR (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.225  MOVK

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

32-bit variant
Applies when sf == 0 && hw == 0x.

MOVK <Wd>, #<imm>{, LSL #<shift>}

64-bit variant
Applies when sf == 1.

MOVK <Xd>, #<imm>{, LSL #<shift>}

Decode for all variants of this encoding

integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;
if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');

Assembler symbols

<id> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm> Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift> For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.
For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

bits(datasize) result;
result = X[d, datasize];
result<pos+15:pos> = imm16;
X[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
   — The values of the data supplied in any of its registers.
The values of the NZCV flags.

The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
C6.2.226   MOVN

Move wide with NOT moves the inverse of an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (inverted wide immediate). See Alias conditions for details of when each alias is preferred.

32-bit variant

Applies when \( sf == 0 \) && \( hw == 0x \).

\[
\text{MOVN} \ <Wd>, \ #<imm>\{, \ LSL \ #<shift>\}
\]

64-bit variant

Applies when \( sf == 1 \).

\[
\text{MOVN} \ <Xd>, \ #<imm>\{, \ LSL \ #<shift>\}
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } datasize &= \text{if } sf == '1' \text{ then } 64 \text{ else } 32; \\
\text{integer } pos &= \text{if } sf == '0' \&\& \text{hw<1> == '1'} \text{ then UNDEFINED; } \\
&\phantom{=} \text{pos} = \text{UInt}(\text{hw:'0000'});
\end{align*}
\]

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>of variant</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (inverted wide immediate)</td>
<td>64-bit</td>
<td>! (IsZero(imm16) &amp;&amp; hw != '00')</td>
</tr>
<tr>
<td>MOV (inverted wide immediate)</td>
<td>32-bit</td>
<td>! (IsZero(imm16) &amp;&amp; hw != '00') &amp;&amp; ! IsOnes(imm16)</td>
</tr>
</tbody>
</table>

Assembler symbols

\(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<imm>\) Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

\(<shift>\) For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as \(<shift>/16\).

For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as \(<shift>/16\).
Operation

bits(datasize) result;
result = Zeros(datasize);
result<pos+15:pos> = imm16;
result = NOT(result);
X[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.227   MOVZ

Move wide with zero moves an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (wide immediate). See Alias conditions for details of when each alias is preferred.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th></th>
<th></th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>opc</td>
<td>imm16</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit variant

Applies when sf == 0 && hw == 0x.

MOVZ <Wd>, #<imm>{, LSL #<shift>}

64-bit variant

Applies when sf == 1.

MOVZ <Xd>, #<imm>{, LSL #<shift>}

Decode for all variants of this encoding

integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;
if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (wide immediate)</td>
<td>! (IsZero(imm16) &amp;&amp; hw != '00')</td>
</tr>
</tbody>
</table>

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<imm> Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

<shift> For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.

For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

bits(datasize) result;

result = Zeros(datasize);
result<pos+15:pos> = imm16;
X[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MRS

Move System Register allows the PE to read an AArch64 System register into a general-purpose register.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16</th>
<th>15 12</th>
<th>11 8</th>
<th>7 5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Encoding**

MRS <Xt>, (<systemreg>|<op0>|<op1>|<Cn>|<Cm>|<op2>)

**Decode for this encoding**

AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);
integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);

**Assembler symbols**

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

<systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".

The System register names are defined in Chapter D17 AArch64 System Register Descriptions.

<op0> Is an unsigned immediate, encoded in the "o0" field. It can have the following values:

2 when o0 = 0
3 when o0 = 1

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

**Operation**

AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
C6.2.229   MSR (immediate)

Move immediate value to Special Register moves an immediate value to selected bits of the PSTATE. For more information, see PSTATE.

The bits that can be written by this instruction are:

- PSTATE.D, PSTATE.A, PSTATE.I, PSTATE.F, and PSTATE.SP.
- If FEAT_SSBS is implemented, PSTATE.SSBS.
- If FEAT_PAN is implemented, PSTATE.PAN.
- If FEAT_UAO is implemented, PSTATE.UAO.
- If FEAT_DIT is implemented, PSTATE.DIT.
- If FEAT_MTE is implemented, PSTATE.TCO.
- If FEAT_NMI is implemented, PSTATE.ALLINT.
- If FEAT_SME is implemented, PSTATE.SM and PSTATE.ZA.

This instruction is used by the aliases SMSTART and SMSTOP. See Alias conditions on page C6-1710 for details of when each alias is preferred.

Encoding

MSR <pstatefield>, #<imm>

Decode for this encoding

if op1 == '000' && op2 == '000' then SEE "CFINV";
if op1 == '000' && op2 == '001' then SEE "XAFLAG";
if op1 == '000' && op2 == '010' then SEE "AXFLAG";

AArch64.CheckSystemAccess('00', op1, '0100', CRm, op2, '11111', '0');
bits(2) min_EL;
boolean need_secure = FALSE;

case op1 of
    when '00x'
        min_EL = EL1;
    when '010'
        min_EL = EL1;
    when '011'
        min_EL = EL0;
    when '100'
        min_EL = EL2;
    when '101'
        if !HaveVirtHostExt() then
            min_EL = EL2;
        when '110'
            min_EL = EL3;
        when '111'
            min_EL = EL1;
        need_secure = TRUE;
if (UInt(PSTATE.EL) < UInt(min_EL) || (need_secure & CurrentSecurityState() != SS_Secure)) then
    UNDEFINED;

PSTATEField field;
case op1:op2 of
    when '000 011'
        if !HaveUAOExt() then UNDEFINED;
        field = PSTATEField_UAO;
    when '000 100'
        if !HavePANExt() then UNDEFINED;
        field = PSTATEField_PAN;
    when '000 101'
        field = PSTATEField_SP;
    when '001 000'
        if !HaveFeatNMI() then UNDEFINED;
        if CRm<3:1> != '000' then UNDEFINED;
        field = PSTATEField_ALLINT;
    when '011 010'
        if !HaveDITExt() then UNDEFINED;
        field = PSTATEField_DIT;
    when '011 011'
        case CRm of
            when '001x'
                if !HaveSME() then UNDEFINED;
                field = PSTATEField_SVCRSM;
            when '010x'
                if !HaveSME() then UNDEFINED;
                field = PSTATEField_SVCRZA;
            when '011x'
                if !HaveSME() then UNDEFINED;
                field = PSTATEField_SVCRSMZA;
            otherwise
                UNDEFINED;
        end;
    when '011 100'
        if !HaveMTEExt() then UNDEFINED;
        field = PSTATEField_TCO;
    when '011 110' field = PSTATEField_DAIFSet;
    when '011 111' field = PSTATEField_DAIFClr;
    when '011 001'
        if !HaveSSBSExt() then UNDEFINED;
        field = PSTATEField_SSBS;
    otherwise
        UNDEFINED;
end;

// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted
if PSTATE.EL == EL0 && field IN {PSTATEField_DAIFSet, PSTATEField_DAIFClr} then
    if !ELUsingAArch32(EL1) & ((EL2Enabled() & HCR_EL2.<E2H,TGE> == '11') || SCTLR_EL1.UMA == '0') then
        if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMSTART</td>
<td>op1 == '011' &amp;&amp; CRm == '0xx1' &amp;&amp; op2 == '011'</td>
</tr>
<tr>
<td>SMSTOP</td>
<td>op1 == '011' &amp;&amp; CRm == '0xx0' &amp;&amp; op2 == '011'</td>
</tr>
</tbody>
</table>

Assembler symbols

<PSTATE FIELD> Is a PSTATE field name. For the MSR instruction, this is encoded in the "op1:op2:CRm" field. It can have the following values:

- **SPSel**: when op1 = 000, op2 = 101, CRm = xxxx
- **DAIFSet**: when op1 = 011, op2 = 110, CRm = xxxx
- **DAIFClr**: when op1 = 011, op2 = 111, CRm = xxxx

When FEAT_UAO is implemented, the following value is also valid:

- **UAO**: when op1 = 000, op2 = 011, CRm = xxxx

When FEAT_PAN is implemented, the following value is also valid:

- **PAN**: when op1 = 000, op2 = 100, CRm = xxxx

When FEAT_NMI is implemented, the following value is also valid:

- **ALLINT**: when op1 = 001, op2 = 000, CRm = 000x

When FEAT_SSBS is implemented, the following value is also valid:

- **SSBS**: when op1 = 011, op2 = 001, CRm = xxxx

When FEAT_DIT is implemented, the following value is also valid:

- **DIT**: when op1 = 011, op2 = 010, CRm = xxxx

When FEAT_SME is implemented, the following values are also valid:

- **SVCRSM**: when op1 = 011, op2 = 011, CRm = 001x
- **SVCRZA**: when op1 = 011, op2 = 011, CRm = 010x
- **SVCRZM2A**: when op1 = 011, op2 = 011, CRm = 011x

When FEAT_MTE is implemented, the following value is also valid:

- **TCO**: when op1 = 011, op2 = 100, CRm = xxxx

See <PSTATE> on page C4-539 when op1 = 000, op2 = 00x, CRm = xxxx.

The following encodings are reserved:

- op1 = 000, op2 = 1lx, CRm = xxxx.
- op1 = 001, op2 = 000, CRm = 001x.
- op1 = 001, op2 = 000, CRm = 01lx.
- op1 = 001, op2 = 000, CRm = 1xxx.
- op1 = 001, op2 = 001, CRm = xxxx.
- op1 = 001, op2 = 01x, CRm = xxxx.
- op1 = 001, op2 = 1xx, CRm = xxxx.
- op1 = 010, op2 = xxx, CRm = xxxx.
- op1 = 011, op2 = 000, CRm = xxxx.
- op1 = 011, op2 = 011, CRm = 000x.
- op1 = 011, op2 = 011, CRm = 1xxx.
- op1 = 011, op2 = 101, CRm = xxxx.
op1 = 1xx, op2 = xxx, CRm = xxxx.

For the SMSTART and SMSTOP aliases, this is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field. Restricted to the range 0 to 1, encoded in "CRm<0>", when <pstatefield> is ALLINT, SVCRSM, SVCRSMZA, or SVCRZA.

**Operation**

```plaintext
case field of
  when PSTATEField_SSBS
    PSTATE.SSBS = CRm<0>;
  when PSTATEField_SP
    PSTATE.SP = CRm<0>;
  when PSTATEField_DAIFSet
    PSTATE.D = PSTATE.D OR CRm<3>;
    PSTATE.A = PSTATE.A OR CRm<2>;
    PSTATE.I = PSTATE.I OR CRm<1>;
    PSTATE.F = PSTATE.F OR CRm<0>;
  when PSTATEField_DAIFClr
    PSTATE.D = PSTATE.D AND NOT(CRm<3>);
    PSTATE.A = PSTATE.A AND NOT(CRm<2>);
    PSTATE.I = PSTATE.I AND NOT(CRm<1>);
    PSTATE.F = PSTATE.F AND NOT(CRm<0>);
  when PSTATEField_PAN
    PSTATE.PAN = CRm<0>;
  when PSTATEField_UAO
    PSTATE.UAO = CRm<0>;
  when PSTATEField_DIT
    PSTATE.DIT = CRm<0>;
  when PSTATEField_TCO
    PSTATE.TCO = CRm<0>;
  when PSTATEField_ALLINT
    if (PSTATE.EL == EL1 && IsHCRXEL2Enabled() && HCRX_EL2.TALLINT == '1' && CRm<0> == '1') then
      AArch64.SystemAccessTrap(EL2, 0x18);
      PSTATE.ALLINT = CRm<0>;
  when PSTATEField_SVCRSM
    CheckSMEAccess();
    SetPSTATE_SM(CRm<0>);
  when PSTATEField_SVCRZA
    CheckSMEAccess();
    SetPSTATE_ZA(CRm<0>);
  when PSTATEField_SVCRSMZA
    CheckSMEAccess();
    SetPSTATE_SM(CRm<0>);
    SetPSTATE_ZA(CRm<0>);
```
C6.2.230 MSR (register)

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose register.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30 29 28</td>
<td>27 26 25 24</td>
</tr>
<tr>
<td>0</td>
<td>L</td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

MSR (<systemreg>|<op0>|<op1>|<Cn>|<Cm>|<op2>), <Xt>

**Decode for this encoding**

AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);

**Assembler symbols**

- `<systemreg>` is a System register name, encoded in the "o0:op1:CRn:CRm:op2". The System register names are defined in Chapter D17 AArch64 System Register Descriptions.
- `<op0>` is an unsigned immediate, encoded in the "o0" field. It can have the following values:
  - 2 when o0 = 0
  - 3 when o0 = 1
- `<op1>` is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- `<Cn>` is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- `<Cm>` is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- `<op2>` is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- `<Xt>` is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

**Operation**

AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
C6.2.231   MSUB

Multiply-Subtract multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

This instruction is used by the alias MNEG. See Alias conditions for details of when each alias is preferred.

\[
\begin{array}{ccccccccccccc}
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sf 0 0</td>
<td>1 1 0 1 1 0 0</td>
<td>Rm 1</td>
<td>Ra</td>
<td>Rn</td>
</tr>
</tbody>
</table>
\end{array}
\]

32-bit variant

Applies when \(sf == 0\).

\text{MSUB} \langle \text{Wd}, \text{Wn}, \text{Wm}, \text{Wa} \rangle

64-bit variant

Applies when \(sf == 1\).

\text{MSUB} \langle \text{Xd}, \text{Xn}, \text{Xm}, \text{Xa} \rangle

\textbf{Decode for all variants of this encoding}

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{integer } a &= \text{UInt}(Ra); \\
\text{integer } \text{destsize} &= \text{if } sf == '1' \text{ then } 64 \text{ else } 32;
\end{align*}
\]

\textbf{Alias conditions}

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNEG</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

\textbf{Assembler symbols}

\text{\langle \text{Wd} \rangle} \quad \text{Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.}

\text{\langle \text{Wn} \rangle} \quad \text{Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.}

\text{\langle \text{Wm} \rangle} \quad \text{Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.}

\text{\langle \text{Wa} \rangle} \quad \text{Is the 32-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.}

\text{\langle \text{Xd} \rangle} \quad \text{Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.}

\text{\langle \text{Xn} \rangle} \quad \text{Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.}

\text{\langle \text{Xm} \rangle} \quad \text{Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.}
<Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

**Operation**

```c
bits(destsize) operand1 = X[n, destsize];
bits(destsize) operand2 = X[m, destsize];
bits(destsize) operand3 = X[a, destsize];

integer result;
result = UInt(operand3) - (UInt(operand1) * UInt(operand2));
X[d, destsize] = result<destsize-1:0>;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.232   MUL

Multiply: \( R_d = R_n \times R_m \)

This instruction is an alias of the MADD instruction. This means that:

- The encodings in this description are named to match the encodings of MADD.
- The description of MADD gives the operational pseudocode for this instruction.

32-bit variant

Applies when \( sf = 0 \).

\[
\text{MUL } \langle W_d \rangle, \langle W_n \rangle, \langle W_m \rangle
\]

is equivalent to

\[
\text{MADD } \langle W_d \rangle, \langle W_n \rangle, \langle W_m \rangle, WZR
\]

and is always the preferred disassembly.

64-bit variant

Applies when \( sf = 1 \).

\[
\text{MUL } \langle X_d \rangle, \langle X_n \rangle, \langle X_m \rangle
\]

is equivalent to

\[
\text{MADD } \langle X_d \rangle, \langle X_n \rangle, \langle X_m \rangle, XZR
\]

and is always the preferred disassembly.

Assembler symbols

- \( \langle W_d \rangle \)
  - Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \( \langle W_n \rangle \)
  - Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- \( \langle W_m \rangle \)
  - Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- \( \langle X_d \rangle \)
  - Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \( \langle X_n \rangle \)
  - Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- \( \langle X_m \rangle \)
  - Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of MADD gives the operational pseudocode for this instruction.
C6.2.233 MVN

Bitwise NOT writes the bitwise inverse of a register value to the destination register.

This instruction is an alias of the ORN (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of ORN (shifted register).
- The description of ORN (shifted register) gives the operational pseudocode for this instruction.

32-bit variant

Applies when \( sf = 0 \).

\[
\text{MVN} <Wd>, <Wm>{, <shift> #<amount>}
\]

is equivalent to

\[
\text{ORN} <Wd>, \text{WZR}, <Wm>{, <shift> #<amount>}
\]

and is always the preferred disassembly.

64-bit variant

Applies when \( sf = 1 \).

\[
\text{MVN} <Xd>, <Xm>{, <shift> #<amount>}
\]

is equivalent to

\[
\text{ORN} <Xd>, \text{XZR}, <Xm>{, <shift> #<amount>}
\]

and is always the preferred disassembly.

Assembler symbols

- \(<Wd>\) is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wm>\) is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- \(<Xd>\) is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xm>\) is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.
- \(<\text{shift}>\) is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:
  - LSL when \( \text{shift} = 00 \)
  - LSR when \( \text{shift} = 01 \)
  - ASR when \( \text{shift} = 10 \)
  - ROR when \( \text{shift} = 11 \)
- \(<\text{amount}>\) is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- For the 32-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.
Operation
The description of ORN (shifted register) gives the operational pseudocode for this instruction.

Operational information
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.234   NEG (shifted register)

Negate (shifted register) negates an optionally-shifted register value, and writes the result to the destination register.

This instruction is an alias of the SUB (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of SUB (shifted register).
- The description of SUB (shifted register) gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when \( sf = 0 \).

\[
\text{NEG} \,<Wd>, \,<Wm>{, <shift> #<amount>}
\]

is equivalent to

\[
\text{SUB} \,<Wd>, \,WZR, \,<Wm>{, <shift> #<amount>}
\]

and is always the preferred disassembly.

### 64-bit variant

Applies when \( sf = 1 \).

\[
\text{NEG} \,<Xd>, \,<Xm>{, <shift> #<amount>}
\]

is equivalent to

\[
\text{SUB} \,<Xd>, \,XZR, \,<Xm>{, <shift> #<amount>}
\]

and is always the preferred disassembly.

### Assembler symbols

- \(<Wd>\) is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wm>\) is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- \(<Xd>\) is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xm>\) is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.
- \(<\text{shift}>\) is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:
  - LSL when \( shift = 00 \)
  - LSR when \( shift = 01 \)
  - ASR when \( shift = 10 \)
  - The encoding \( shift = 11 \) is reserved.
- \(<\text{amount}>\) is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- For the 32-bit variant: the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- For the 64-bit variant: the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.
**Operation**

The description of **SUB (shifted register)** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.235   NEGS

Negate, setting flags, negates an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is an alias of the SUBS (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

32-bit variant

Applies when \( sf = 0 \).

\[
\text{NEGS } <Wd>, <Wm>\{, <shift> \#<amount>\}
\]

is equivalent to

\[
\text{SUBS } <Wd>, WZR, <Wm> \{, <shift> \#<amount>\}
\]

and is always the preferred disassembly.

64-bit variant

Applies when \( sf = 1 \).

\[
\text{NEGS } <Xd>, <Xm>\{, <shift> \#<amount>\}
\]

is equivalent to

\[
\text{SUBS } <Xd>, XZR, <Xm> \{, <shift> \#<amount>\}
\]

and is always the preferred disassembly.

Assembler symbols

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wm>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xm>\) Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.
- \(<\text{shift}>\) Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:
  - LSL when \( \text{shift} = 00 \)
  - LSR when \( \text{shift} = 01 \)
  - ASR when \( \text{shift} = 10 \)
  - The encoding \( \text{shift} = 11 \) is reserved.
- \(<\text{amount}>\) For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.
Operation
The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational information
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.236  NGC

Negate with Carry negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register.

This instruction is an alias of the SBC instruction. This means that:

- The encodings in this description are named to match the encodings of SBC.
- The description of SBC gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>32-bit variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applies when <code>sf == 0</code>.</td>
</tr>
<tr>
<td><code>NGC &lt;Wd&gt;, &lt;Wm&gt;</code></td>
</tr>
<tr>
<td>is equivalent to</td>
</tr>
<tr>
<td><code>SBC &lt;Wd&gt;, WZR, &lt;Wm&gt;</code></td>
</tr>
<tr>
<td>and is always the preferred disassembly.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64-bit variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applies when <code>sf == 1</code>.</td>
</tr>
<tr>
<td><code>NGC &lt;Xd&gt;, &lt;Xm&gt;</code></td>
</tr>
<tr>
<td>is equivalent to</td>
</tr>
<tr>
<td><code>SBC &lt;Xd&gt;, XZR, &lt;Xm&gt;</code></td>
</tr>
<tr>
<td>and is always the preferred disassembly.</td>
</tr>
</tbody>
</table>

Assembler symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wm>` Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xm>` Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of SBC gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.237   NGCS

Negate with Carry, setting flags, negates the sum of a register value and the value of NOT (Carry flag), and writes
the result to the destination register. It updates the condition flags based on the result.

This instruction is an alias of the SBCS instruction. This means that:

• The encodings in this description are named to match the encodings of SBCS.
• The description of SBCS gives the operational pseudocode for this instruction.

32-bit variant
Applies when $sf == 0$.

NGCS $<Wd>, <Wm>$
is equivalent to
SBCS $<Wd>, WZR, <Wm>$
and is always the preferred disassembly.

64-bit variant
Applies when $sf == 1$.

NGCS $<Xd>, <Xm>$
is equivalent to
SBCS $<Xd>, XZR, <Xm>$
and is always the preferred disassembly.

Assembler symbols

$<Wd>$ Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

$<Wm>$ Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

$<Xd>$ Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

$<Xm>$ Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation
The description of SBCS gives the operational pseudocode for this instruction.

Operational information
If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.238  NOP

No Operation does nothing, other than advance the value of the program counter by 4. This instruction can be used for instruction alignment purposes.

——— Note ————
The timing effects of including a NOP instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, NOP instructions are not suitable for timing loops.

Encoding

NOP

Decode for this encoding

// Empty.

Operation

// do nothing

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.239 ORN (shifted register)

Bitwise OR NOT (shifted register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MVN. See Alias conditions for details of when each alias is preferred.

<table>
<thead>
<tr>
<th>32-bit variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applies when $sf == 0$.</td>
</tr>
<tr>
<td>ORN $&lt;Wd&gt;, &lt;Wn&gt;, &lt;Wm&gt;$</td>
</tr>
<tr>
<td>Include</td>
</tr>
<tr>
<td>$&lt;\text{shift}&gt; #&lt;\text{amount}&gt;$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64-bit variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applies when $sf == 1$.</td>
</tr>
<tr>
<td>ORN $&lt;Xd&gt;, &lt;Xn&gt;, &lt;Xm&gt;$</td>
</tr>
<tr>
<td>Include</td>
</tr>
<tr>
<td>$&lt;\text{shift}&gt; #&lt;\text{amount}&gt;$</td>
</tr>
</tbody>
</table>

**Decode for all variants of this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if $sf == '1'$ then 64 else 32;
if $sf == '0'$ & & imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

**Alias conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVN</td>
<td>$Rn == '11111'$</td>
</tr>
</tbody>
</table>

**Asmbluer symbols**

- $<\text{id}>$ Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- $<\text{in}>$ Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- $<\text{im}>$ Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- $<\text{id}>$ Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- $<\text{in}>$ Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- $<\text{im}>$ Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- $<\text{shift}>$ Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:
  - LSL when $shift = 00$
LSR  when shift = 01
ASR  when shift = 10
ROR  when shift = 11

<amount>  For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
          For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Operation

bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(datasize) result;

operand2 = NOT(operand2);
result = operand1 OR operand2;
X[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.240 ORR (immediate)

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate register value, and writes the result to the destination register.

This instruction is used by the alias MOV (bitmask immediate). See Alias conditions for details of when each alias is preferred.

### ORR <Wd|WSP>, <Wn>, #<imm>

### ORR <Xd|SP>, <Xn>, #<imm>

#### 32-bit variant

Applies when \( sf = 0 \) \&\& \( N = 0 \).

\[
\text{ORR } \langle \text{Wd}|\text{WSP} \rangle, \langle \text{Wn} \rangle, \#\langle \text{imm} \rangle
\]

#### 64-bit variant

Applies when \( sf = 1 \).

\[
\text{ORR } \langle \text{Xd}|\text{SP} \rangle, \langle \text{Xn} \rangle, \#\langle \text{imm} \rangle
\]

#### Decode for all variants of this encoding

\[
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
\text{sf} & 0 & 1 & 0 & 0 & 0 & 0 & N \\
\hline
\text{imms} & \text{imm} & \text{Rn} & \text{Rd} \\
\hline
\end{array}
\]

#### Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (bitmask immediate)</td>
<td>( Rn = \text{'11111'} ) &amp;&amp; \text{MoveWidePreferred}(sf, N, imms, immr)</td>
</tr>
</tbody>
</table>

#### Assembler symbols

- \( \langle \text{Wd}|\text{WSP} \rangle \): Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- \( \langle \text{Wn} \rangle \): Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \( \langle \text{Xd}|\text{SP} \rangle \): Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- \( \langle \text{Xn} \rangle \): Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \( \langle \text{imm} \rangle \): For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
  
  For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".
### Operation

```
bias(datasize) result;
bias(datasize) operand1 = X[n, datasize];

result = operand1 OR imm;
if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.241 ORR (shifted register)

Bitwise OR (shifted register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MOV (register). See Alias conditions for details of when each alias is preferred.

### 32-bit variant
Applies when \( sf == 0 \).

\[
\text{ORR} \ <W_d>, <W_n>, <W_m>\{, <\text{shift}\> \#<\text{amount}>\}
\]

### 64-bit variant
Applies when \( sf == 1 \).

\[
\text{ORR} \ <X_d>, <X_n>, <X_m>\{, <\text{shift}\> \#<\text{amount}>\}
\]

### Decode for all variants of this encoding

| \( sf \) | \( 0 \) | \( 1 \) | \( 0 \) | \( 1 \) | \( 0 \) | \( 0 \) | \( 1 \) | \( 0 \) | \( 0 \) | \( 1 \) | \( 0 \) | \( 0 \) | \( 1 \) | \( 0 \) | \( 0 \) | \( 0 \) | \( 0 \) | \( 0 \) | \( 0 \) | \( 0 \) | \( 0 \) | \( 0 \) | \( 0 \) | \( 0 \) |
| opc | N | Rm | imm6 | Rn | Rd |

| integer \( d \) = UInt(Rd); |
| integer \( n \) = UInt(Rn); |
| integer \( m \) = UInt(Rm); |
| integer \( \text{datasize} \) = if \( sf == '1' \) then 64 else 32; |
| if \( sf == '0' \) && \( \text{imm6}<5> == '1' \) then UNDEFINED; |

\[
\text{ShiftType} \ shift\_\text{type} = \text{DecodeShift}(\text{shift});
\]

integer \( shift\_\text{amount} = \text{UInt}(\text{imm6}); \)

### Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (register)</td>
<td>( \text{shift} == '00' ) &amp;&amp; ( \text{imm6} == '000000' ) &amp;&amp; ( \text{Rn} == '11111' )</td>
</tr>
</tbody>
</table>

### Assembler symbols

- \(<W_d>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<W_n>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<W_m>\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<X_d>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<X_n>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<X_m>\) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<\text{shift}>\) Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:
  - LSL when \( \text{shift} == 00 \)
LSR  when shift = 01
ASR  when shift = 10
ROR  when shift = 11

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Operation**

bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(datasize) result;

result = operand1 OR operand2;
X[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.242 PACDA, PACDZA

Pointer Authentication Code for Data address, using key A. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDA.
- The value zero, for PACDZA.

**Integer**

(FEAT_PAuth)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 1 1 0 0 0 0 1 0 0</td>
<td>Z</td>
<td>0 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PACDA variant**

Applies when Z == 0.

PACDA <Xd>, <Xn|SP>

**PACDZA variant**

Applies when Z == 1 && Rn == 11111.

PACDZA <Xd>

**Decode for all variants of this encoding**

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // PACDA
    if n == 31 then source_is_sp = TRUE;
else // PACDZA
    if n != 31 then UNDEFINED;

**Assembler symbols**

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

**Operation**

if source_is_sp then
    X[d, 64] = AddPACDA(X[d, 64], SP[]);
else
    X[d, 64] = AddPACDA(X[d, 64], X[n, 64]);
C6.2.243   PACDB, PACDZB

Pointer Authentication Code for Data address, using key B. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by <Xd>.

The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDB.
- The value zero, for PACDZB.

**Integer**

(FEAT_PAuth)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 1 0 1 1 0 0 0 0 1 0 0</td>
<td>Z 0 1 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PACDB variant**

Applies when Z == 0.

PACDB <Xd>, <Xn|SP>

**PACDZB variant**

Applies when Z == 1 & Rn == 1111.

PACDZB <Xd>

**Decode for all variants of this encoding**

```plaintext
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;
else // PACDB
    if Z == '0' then
        if n == 31 then source_is_sp = TRUE;
    else // PACDZB
        if n != 31 then UNDEFINED;
```

**Assembler symbols**

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

**Operation**

```plaintext
if source_is_sp then
    X[d, 64] = AddPACDB(X[d, 64], SP[]);
else
    X[d, 64] = AddPACDB(X[d, 64], X[n, 64]);
```
C6.2.244  PACGA

Pointer Authentication Code, using Generic key. This instruction computes the pointer authentication code for an address in the first source register, using a modifier in the second source register, and the Generic key. The computed pointer authentication code is returned in the upper 32 bits of the destination register.

**Integer**

(FEAT_PAuth)

| [31 30 29 28 27 26 25 24 23 22 21 20] | [16 15 14 13 12 11 10 9] | [5 4] | 0 | 
|---------------------------------------|--------------------------|------|---|---|
| 1 0 0 1 1 0 1 0 1 1 0 | Rm | 0 0 1 1 0 0 | Rd |

**Encoding**

PACGA <Xd>, <Xn>, <Xm|SP>

**Decode for this encoding**

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if !HavePACExt() then
  UNDEFINED;
if m == 31 then source_is_sp = TRUE;

**Assembler symbols**

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Rm" field.

**Operation**

if source_is_sp then
  X[d, 64] = AddPACGA(X[n, 64], SP[]);
else
  X[d, 64] = AddPACGA(X[n, 64], X[m, 64]);
C6.2.245  PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA

Pointer Authentication Code for Instruction address, using key A. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key A.

The address is:

- In the general-purpose register that is specified by <Xd> for PACIA and PACIZA.
- In X17, for PACIA1716.
- In X30, for PACIASP and PACIAZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIA.
- The value zero, for PACIZA and PACIAZ.
- In X16, for PACIA1716.
- In SP, for PACIASP.

**Integer**

(FEAT_PAuth)

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0</td>
<td>1 1 0 1 1 0</td>
<td>0 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>
```

**PACIA variant**

Applies when Z == 0.

PACIA <Xd>, <Xn|SP>

**PACIZA variant**

Applies when Z == 1 && Rn == 11111.

PACIZA <Xd>

**Decode for all variants of this encoding**

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
  UNDEFINED;

if Z == '0' then // PACIA
  if n == 31 then source_is_sp = TRUE;
else // PACIZA
  if n != 31 then UNDEFINED;

**System**

(FEAT_PAuth)
A64 Base Instruction Descriptions
C6.2 Alphabetical list of A64 base instructions

PACIA1716 variant
Applies when CRm == 0001 && op2 == 000.

PACIASP variant
Applies when CRm == 0011 && op2 == 001.

PACIAZ variant
Applies when CRm == 0011 && op2 == 000.

Decode for all variants of this encoding

```c
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
  when '0011 000'    // PACIAZ
    d = 30;
    n = 31;
  when '0011 001'    // PACIASP
    d = 30;
    source_is_sp = TRUE;
    if HaveBTIExt() then
      // Check for branch target compatibility between PSTATE.BTYPE
      // and implicit branch target of PACIASP instruction.
      SetBTypeCompatible(BTypeCompatible_PACIXSP());
    when '0001 000'    // PACIA1716
    d = 17;
    n = 16;
  when '0001 010'    // PACIB
  when '0001 100'    // AUTIA
  when '0001 110'    // AUTIB
  when '0011 01x'    // PACIB
  when '0011 10x'    // AUTIA
  when '0011 11x'    // AUTIB
  when '0000 111'    // XPACLR1
    otherwise SEE "HINT";
```

Assembler symbols

- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation for all encodings

```c
if HavePACExt() then
  if source_is_sp then
    X[d, 64] = AddPACIA(X[d, 64], SP[]);
```
else
    X[d, 64] = AddPACIA(X[d, 64], X[n, 64]);
C6.2.246   PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB

Pointer Authentication Code for Instruction address, using key B. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by \(<Xd>\) for PACIB and PACIZ8.
- In X17, for PACIB1716.
- In X30, for PACIBSP and PACIBZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by \(<Xn|SP>\) for PACIB.
- The value zero, for PACIZ8 and PACIBZ.
- In X16, for PACIB1716.
- In SP, for PACIBSP.

**Integer**

(FEAT_PAuth)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 0 0 0 0 1 0 0</td>
<td>Z</td>
<td>0 0 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PACIB variant**

Applies when \(Z == 0\).

PACIB \(<Xd>, <Xn|SP>\)

**PACIZB variant**

Applies when \(Z == 1 \&\& Rn == 11111\).

PACIZB \(<Xd>\)

**Decode for all variants of this encoding**

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
   UNDEFINED;

if \(Z == '0'\) then // PACIB
   if \(n == 31\) then source_is_sp = TRUE;
   else // PACIZB
      if \(n != 31\) then UNDEFINED;

**System**

(FEAT_PAuth)
PACIB1716 variant
Applies when CRm == 0001 && op2 == 010.
PACIB1716

PACIBSP variant
Applies when CRm == 0011 && op2 == 011.
PACIBSP

PACIBZ variant
Applies when CRm == 0011 && op2 == 010.
PACIBZ

Decode for all variants of this encoding

integer d;
integer n;
boolean source_is_sp = FALSE;
case CRm:op2 of
  when '0011 010'    // PACIBZ
    d = 30;
    n = 31;
  when '0011 011'    // PACIBSP
    d = 30;
    source_is_sp = TRUE;
    if HaveBTIExt() then
      // Check for branch target compatibility between PSTATE.BTYPE
      // and implicit branch target of PACIBSP instruction.
      SetBTypeCompatible(BTypeCompatible_PACIXSP());
  when '0001 010'    // PACIB1716
    d = 17;
    n = 16;
  when '0001 000' SEE "PACIA";
  when '0001 100' SEE "AUTIA";
  when '0001 110' SEE "AUTIB";
  when '0011 00x' SEE "PACIA";
  when '0011 10x' SEE "AUTIA";
  when '0011 11x' SEE "AUTIB";
  when '0000 111' SEE "XPACLRI";
  otherwise SEE "HINT";

Assembler symbols

<Xd>            Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP>          Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation for all encodings

if HavePACExt() then
  if source_is_sp then
    X[d, 64] = AddPACIB(X[d, 64], SP[]);
else
   \( x[d, 64] = \text{AddPACIB}(x[d, 64], x[n, 64]) \);
C6.2.247 PRFM (immediate)

Prefetch Memory (immediate) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of a PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory on page C3-267.

For information about memory accesses, see Load/store addressing modes on page C1-234.

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

PRFM (<prfop>|#<imm5>), [<Xn|SP>{, #<pimm>}]  

**Decode for this encoding**

bits(64) offset = LSL(ZeroExtend(imm12, 64), 3);

**Assembler symbols**

- `<prfop>` Is the prefetch operation, defined as `<type><target><policy>`.  
  - `<type>` is one of:
    - `PLD` Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.  
    - `PLI` Preload instructions, encoded in the "Rt<4:3>" field as 0b01.  
    - `PST` Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.  
  - `<target>` is one of:
    - `L1` Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.  
    - `L2` Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.  
    - `L3` Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.  
  - `<policy>` is one of:
    - `KEEP` Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.  
    - `STRM` Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.  

For more information on these prefetch operations, see Prefetch memory on page C3-267.

For other encodings of the "Rt" field, use `<imm5>`.

- `<imm5>` Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using `<prfop>`.

- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

- `<pimm>` Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as `<pimm>/8`.  

```
Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);

Operation

bits(64) address;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

if n == 31 then
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

Prefetch(address, t<4:0>);
C6.2.248 PRFM (literal)

Prefetch Memory (literal) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of a PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory on page C3-267.

For information about memory accesses, see Load/store addressing modes on page C1-234.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23</th>
<th>22 21 20 19 18 17 16 15 14</th>
<th>13 12 11 10 9 8 7 6 5 4</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 0 0</td>
<td>imm19</td>
<td>Rt</td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

PRFM (<prfop>|#<imm5>), <label>

**Decode for this encoding**

integer t = UInt(Rt);
bits(64) offset;
offset = SignExtend(imm19:’00’, 64);

**Assembler symbols**

- `<prfop>` Is the prefetch operation, defined as `<type><target><policy>`. `<type>` is one of:
  - PLD Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.
  - PLI Preload instructions, encoded in the "Rt<4:3>" field as 0b01.
  - PST Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.
- `<target>` is one of:
  - L1 Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.
  - L2 Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.
  - L3 Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.
- `<policy>` is one of:
  - KEEP Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.
  - STRM Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see Prefetch memory on page C3-267.

For other encodings of the "Rt" field, use `<imm5>`.

- `<imm5>` Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.
  This syntax is only for encodings that are not accessible using `<prfop>`.

- `<label>` Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.
Operation

bits(64) address = PC[] + offset;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

Prefetch(address, t<4:0>);
C6.2.249 PRFM (register)

Prefetch Memory (register) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of a PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory on page C3-267.

For information about memory accesses, see Load/store addressing modes on page C1-234.

Encoding

PRFM (<prfop>|#<imm5>), [<Xn|SP>, (<Wm>|<Xm>){{, <extend> {<amount>}}}

Decode for this encoding

if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 3 else 0;

Assembler symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.
        <type> is one of:
        PLD     Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.
        PLI     Preload instructions, encoded in the "Rt<4:3>" field as 0b01.
        PST     Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.
        <target> is one of:
        L1     Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.
        L2     Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.
        L3     Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.
        <policy> is one of:
        KEEP    Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.
        STRM    Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.
For more information on these prefetch operations, see Prefetch memory on page C3-267.
For other encodings of the "Rt" field, use <imm5>.

<iimm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
When \texttt{option=0} is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when \texttt{amount} is omitted, encoded in the "option" field. It can have the following values:

- \texttt{UXTW} when \texttt{option = 010}
- \texttt{LSL} when \texttt{option = 011}
- \texttt{SXTW} when \texttt{option = 110}
- \texttt{SXTX} when \texttt{option = 111}

Is the index shift amount, optional only when \texttt{extend} is not LSL. Where it is permitted to be optional, it defaults to \#0. It is encoded in the "S" field. It can have the following values:

- \#0 when \texttt{S = 0}
- \#3 when \texttt{S = 1}

**Shared decode for all encodings**

\begin{verbatim}
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);

Operation

bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);
if n == 31 then
    address = SP[];
else
    address = X[n, 64];
address = address + offset;
Prefetch(address, t<4:0>);
\end{verbatim}
C6.2.250  PRFUM

Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of a PRFUM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory on page C3-267.

For information about memory accesses, see Load/store addressing modes on page C1-234.

Encoding

PRFUM (<prfop>|#<imm5>), [<Xn|SP>{, #<simm>}]

Decode for this encoding

bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<prfop>  Is the prefetch operation, defined as <type><target><policy>.
  <type> is one of:
  PLD    Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.
  PLI    Preload instructions, encoded in the "Rt<4:3>" field as 0b01.
  PST    Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.
  <target> is one of:
  L1     Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.
  L2     Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.
  L3     Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.
  <policy> is one of:
  KEEP   Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.
  STRM   Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see Prefetch memory on page C3-267.

For other encodings of the "Rt" field, use <imm5>.

<imm5>  Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm>  Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.
Shared decode for all encodings

    integer n = UInt(Rn);
    integer t = UInt(Rt);

Operation

    bits(64) address;

    if HaveMTE2Ext() then
        SetTagCheckedInstruction(FALSE);

    if n == 31 then
        address = SP[];
    else
        address = X[n, 64];

    address = address + offset;
    Prefetch(address, t<4:0>);
C6.2.251   PSB CSYNC

Profiling Synchronization Barrier. This instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following DSB instruction completes when the writes to the profiling buffer have completed.

If the Statistical Profiling Extension is not implemented, this instruction executes as a NOP.

System

(FEAT_SPE)

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11  8  7  5  4  3  2  1  0 |
|-------------|-------------|-------------|-------------|-------------|-------------|-----|-----|-----|-----|
|    1        |    1        |    0        |    1        |    0        |    0        |
|    0        |    0        |    0        |    1        |    1        |    1        |
|    0        |    1        |    0        |    0        |    0        |    1        |
```

Encoding

PSB CSYNC

**Decode for this encoding**

```
if !HaveStatisticalProfiling() then EndOfInstruction();
```

Operation

```
ProfilingSynchronizationBarrier();
```
**C6.2.252 PSSBB**

Physical Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same physical address.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the PSSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the PSSBB.

- When a load to a location appears in program order before the PSSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the PSSBB.

This instruction is an alias of the DSB instruction. This means that:

- The encodings in this description are named to match the encodings of DSB.
- The description of DSB gives the operational pseudocode for this instruction.

```plaintext
0x0 00 011 0011 01001 00 11111
```

**Encoding**

PSSBB

is equivalent to

DSB #4

and is always the preferred disassembly.

**Operation**

The description of DSB gives the operational pseudocode for this instruction.
C6.2.253  RBIT

Reverse Bits reverses the bit order in a register.

32-bit variant
Applies when \( sf = 0 \).
RBIT \(<Wd>, <Wn>\)

64-bit variant
Applies when \( sf = 1 \).
RBIT \(<Xd>, <Xn>\)

Decode for all variants of this encoding
\[
\begin{align*}
\text{integer } d &= \text{ UInt}(Rd); \\
\text{integer } n &= \text{ UInt}(Rn); \\
\text{integer } \text{datasize} &= \text{ if } sf = '1' \text{ then } 64 \text{ else } 32;
\end{align*}
\]

Assembler symbols
- \(<Wd>\) is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<Xd>\) is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\) is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation
\[
\begin{align*}
\text{bits(datasize) operand} &= X[n, \text{datasize}]; \\
\text{bits(datasize) result}; \\
\text{for } i = 0 \text{ to } \text{datasize}-1 \\
& \quad \text{result}<(\text{datasize}-1)-i> = \text{operand}<i>; \\
X[d, \text{datasize}] &= \text{result};
\end{align*}
\]

Operational information
If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
— The values of the NZCV flags.
C6.2.254   RET

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.

Encoding

RET {<Xn>}

Decode for this encoding

integer n = UInt(Rn);

Assembler symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field. Defaults to X30 if absent.

Operation

bits(64) target = X[n, 64];

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

BranchTo(target, BranchType_RET, FALSE);
C6.2.255 RETAA, RETAB

Return from subroutine, with pointer authentication. This instruction authenticates the address that is held in LR, using SP as the modifier and the specified key, branches to the authenticated address, with a hint that this instruction is a subroutine return.

Key A is used for RETAA, and key B is used for RETAB.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to LR.

**Integer**

(FEAT_PAuth)

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10  9 |  5  4 |  0 |
|------------|------------|------------|------------|------------|-------|-------|
| 1 1 0 1 0 1 1 1 | 0 1 0 1 1 1 1 1 | 0 0 0 0 1 1 1 1 |

**RETAA variant**

Applies when \( M = 0 \).

RETAA

**RETAB variant**

Applies when \( M = 1 \).

RETAB

**Decode for all variants of this encoding**

```java
boolean use_key_a = (M == '0');
if !HavePACExt() then
  UNDEFINED;
```

**Operation**

```java
bits(64) target = X[30, 64];
bits(64) modifier = SP[];
if use_key_a then
  target = AuthIA(target, modifier, TRUE);
else
  target = AuthIB(target, modifier, TRUE);

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';
BranchTo(target, BranchType_RET, FALSE);
```
C6.2.256 REV

Reverse Bytes reverses the byte order in a register.

This instruction is used by the pseudo-instruction REV64. The pseudo-instruction is never the preferred disassembly.

32-bit variant

Applies when sf == 0 && opc == 10.

REV <Wd>, <Wn>

64-bit variant

Applies when sf == 1 && opc == 11.

REV <Xd>, <Xn>

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;

case opc of
  when '00'
    Unreachable();
  when '01'
    container_size = 16;
  when '10'
    container_size = 32;
  when '11'
    if sf == '0' then UNDEFINED;
   container_size = 64;

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

bits(datasize) operand = X[n, datasize];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
    rev_index = index + ((elements_per_container - 1) * 8);
    for e = 0 to elements_per_container-1
        result<rev_index+7:rev_index> = operand<index+7:index>;
        index = index + 8;
        rev_index = rev_index - 8;

X[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.257 REV16

Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of a register.

32-bit variant
Applies when \( sf = 0 \).
REV16 \(<Wd>, <Wn>\)

64-bit variant
Applies when \( sf = 1 \).
REV16 \(<Xd>, <Xn>\)

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } d & = \text{UInt}(Rd); \\
\text{integer } n & = \text{UInt}(Rn); \\
\text{integer } \text{datasize} & = \text{if } sf = '1' \text{ then } 64 \text{ else } 32; \\
\text{integer } \text{container}_\text{size} & = \text{case } \text{opc} \text{ of } \\
& \quad \text{when } '00' \rightarrow \text{Unreachable}(); \\
& \quad \text{when } '01' \rightarrow \text{container}_\text{size} = 16; \\
& \quad \text{when } '10' \rightarrow \text{container}_\text{size} = 32; \\
& \quad \text{when } '11' \rightarrow \\
& \quad \quad \text{if } sf = '0' \rightarrow \text{UNDEFINED}; \\
& \quad \quad \text{container}_\text{size} = 64; \\
\end{align*}
\]

Assembler symbols

\(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
\(<Wn>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
\(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
\(<Xn>\) Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

\[
\begin{align*}
\text{bits(datosize) operand} & = X[n, \text{datasize}]; \\
\text{bits(datosize) result}; \\
\text{integer } \text{containers} & = \text{datasize DIV container}_\text{size}; \\
\text{integer } \text{elements}\_\text{per}\_\text{container} & = \text{container}_\text{size DIV 8}; \\
\text{integer } \text{index} & = 0; \\
\text{integer } \text{rev}\_\text{index}; \\
\text{for } c = 0 \text{ to } \text{containers}-1 \rightarrow \\
\text{rev}\_\text{index} & = \text{index} + ((\text{elements}\_\text{per}\_\text{container} - 1) \times 8); \\
\end{align*}
\]
for e = 0 to elements_per_container-1
    result<rev_index+7:rev_index> = operand<index+7:index>;
    index = index + 8;
    rev_index = rev_index - 8;

X[d, datasize] = result;

Operational information
If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.258  REV32

Reverse bytes in 32-bit words reverses the byte order in each 32-bit word of a register.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 1 1 0 0 0 0 0 0 0 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

REV32 <Xd>, <Xn>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
integer container_size;
case opc of
  when '00' 
    Unreachable();
  when '01' 
    container_size = 16;
  when '10' 
    container_size = 32;
  when '11' 
    if sf == '0' then UNDEFINED;
    container_size = 64;

**Assembler symbols**

Xd  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
Xn  Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

bits(datasize) operand = X[n, datasize];
bits(datasize) result;
integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
  rev_index = index + ((elements_per_container - 1) + 8);
  for e = 0 to elements_per_container-1
    result<rev_index+7:rev_index> = operand<index+7:index>;
    index = index + 8;
    rev_index = rev_index - 8;
X[d, datasize] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.259  REV64

Reverse Bytes reverses the byte order in a 64-bit general-purpose register.

When assembling for Armv8.2, an assembler must support this pseudo-instruction. It is OPTIONAL whether an assembler supports this pseudo-instruction when assembling for an architecture earlier than Armv8.2.

This instruction is a pseudo-instruction of the REV instruction. This means that:

- The encodings in this description are named to match the encodings of REV.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of REV gives the operational pseudocode for this instruction.

64-bit variant

REV64 <Xd>, <Xn>

is equivalent to

REV  <Xd>, <Xn>

and is never the preferred disassembly.

Assembler symbols

| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
| <Xn> | Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of REV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.260  RMIF

Performs a rotation right of a value held in a general purpose register by an immediate value, and then inserts a selection of the bottom four bits of the result of the rotation into the PSTATE flags, under the control of a second immediate mask.

**Integer**

(FEAT_FlagM)

- **Encoding**
  
  RMIF <Xn>, #<shift>, #<mask>

- **Decode for this encoding**
  
  if !HaveFlagManipulateExt() then UNDEFINED;
  integer lsb = UInt(imm6);
  integer n = UInt(Rn);

- **Assembler symbols**
  
  - `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
  - `<shift>` Is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
  - `<mask>` Is the flag bit mask, an immediate in the range 0 to 15, which selects the bits that are inserted into the NZCV condition flags, encoded in the "mask" field.

- **Operation**
  
  bits(4) tmp;
  bits(64) tmpreg = X[n, 64];
  tmp = (tmpreg:tmpreg)<lsb+3:lsb>;
  if mask<3> == '1' then PSTATE.N = tmp<3>;
  if mask<2> == '1' then PSTATE.Z = tmp<2>;
  if mask<1> == '1' then PSTATE.C = tmp<1>;
  if mask<0> == '1' then PSTATE.V = tmp<0>;

- **Operational information**
  
  If PSTATE.DIT is 1:
  
  - The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
C6.2.261   ROR (immediate)

Rotate right (immediate) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This instruction is an alias of the EXTR instruction. This means that:

• The encodings in this description are named to match the encodings of EXTR.
• The description of EXTR gives the operational pseudocode for this instruction.

32-bit variant
Applies when \( sf = 0 \land N = 0 \land \text{imms} = \text{xxxxx} \).

\[ \text{ROR} \ <Wd>, <Ws>, #<shift> \]

is equivalent to

\[ \text{EXTR} \ <Wd>, <Ws>, <Ws>, #<shift> \]

and is the preferred disassembly when \( Rn = Rm \).

64-bit variant
Applies when \( sf = 1 \land N = 1 \).

\[ \text{ROR} \ <Xd>, <Xs>, #<shift> \]

is equivalent to

\[ \text{EXTR} \ <Xd>, <Xs>, <Xs>, #<shift> \]

and is the preferred disassembly when \( Rn = Rm \).

Assembler symbols

\(<Wd>\)  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Ws>\)  Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.

\(<Xd>\)  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Xs>\)  Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.

\(<\text{shift}>\)  For the 32-bit variant: is the amount by which to rotate, in the range 0 to 31, encoded in the "imms" field.

For the 64-bit variant: is the amount by which to rotate, in the range 0 to 63, encoded in the "imms" field.

Operation

The description of EXTR gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.262   ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is an alias of the RORV instruction. This means that:

- The encodings in this description are named to match the encodings of RORV.
- The description of RORV gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when \( sf = 0 \).

ROR \(<Wd>, <Wn>, <Wm>\)

is equivalent to

RORV \(<Wd>, <Wn>, <Wm>\)

and is always the preferred disassembly.

### 64-bit variant

Applies when \( sf = 1 \).

ROR \(<Xd>, <Xn>, <Xm>\)

is equivalent to

RORV \(<Xd>, <Xn>, <Xm>\)

and is always the preferred disassembly.

**Assembler symbols**

- \(<Wd>\) is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Wm>\) is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- \(<Xd>\) is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\) is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Xm>\) is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

**Operation**

The description of RORV gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.263 RORV

Rotate Right Variable provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ROR (register). The alias is always the preferred disassembly.

32-bit variant
Applies when $sf == 0$.
RORV <Wd>, <Wn>, <Wm>

64-bit variant
Applies when $sf == 1$.
RORV <Xd>, <Xn>, <Xm>

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);

Assembler symbols

<\Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<\Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<\Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

bits(datasize) result;
bits(datasize) operand2 = X[m, datasize];
result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
X[d, datasize] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.264   SB

Speculation Barrier is a barrier that controls speculation.

The semantics of the Speculation Barrier are that the execution, until the barrier completes, of any instruction that appears later in the program order than the barrier:

- Cannot be performed speculatively to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception.

In particular, any instruction that appears later in the program order than the barrier cannot cause a speculative allocation into any caching structure where the allocation of that entry could be indicative of any data value present in memory or in the registers.

The SB instruction:

- Cannot be speculatively executed as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception. The potentially exception generating instruction can complete once it is known not to be speculative, and all data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

When the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after an uncompleted SB instruction, the SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched.

System

(FEAT_SB)

Encoding

SB

Decode for this encoding

if !HaveSBExt() then UNDEFINED;

Operation

SpeculationBarrier();
C6.2.265   SBC

Subtract with Carry subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

This instruction is used by the alias NGC. See Alias conditions for details of when each alias is preferred.

32-bit variant
Applies when $sf == 0$.

SBC <Wd>, <Wn>, <Wm>

64-bit variant
Applies when $sf == 1$.

SBC <Xd>, <Xn>, <Xm>

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if $sf == '1'$ then 64 else 32;

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGC</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

bits(datasize) result;
bites(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, PSTATE.C);
X[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.266  SBCS

Subtract with Carry, setting flags, subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias NGCS. See Alias conditions for details of when each alias is preferred.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
</table>
sf | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |

op S

32-bit variant

Applies when sf == 0.

SBCS <Wd>, <Wn>, <Wm>

64-bit variant

Applies when sf == 1.

SBCS <Xd>, <Xn>, <Xm>

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGCS</td>
<td>Rn == '1111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Wd>  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>  Is the 64-bit name of the general-purpose destination register, encoded in the "Rm" field.
<Xn>  Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>  Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
bits(4) nzcv;
operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);

PSTATE.<N,Z,C,V> = nzcv;

X[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.267 SBFIZ

Signed Bitfield Insert in Zeros copies a bitfield of `<width>` bits from the least significant bits of the source register to bit position `<lsb>` of the destination register, setting the destination bits below the bitfield to zero, and the bits above the bitfield to a copy of the most significant bit of the bitfield.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

### 32-bit variant
Applies when `sf == 0 && N == 0`.

```
SBFIZ <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when ` UInt(imms) < UInt(immr)`.

### 64-bit variant
Applies when `sf == 1 && N == 1`.

```
SBFIZ <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when ` UInt(imms) < UInt(immr)`.

### Assembler symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<lsb>` For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.
  For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
- `<width>` For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.
  For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

### Operation
The description of SBFM gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If \( \text{imms} \) is greater than or equal to \( \text{immr} \), this copies a bitfield of \( (\text{imms} - \text{immr} + 1) \) bits starting from bit position \( \text{immr} \) in the source register to the least significant bits of the destination register.

If \( \text{imms} \) is less than \( \text{immr} \), this copies a bitfield of \( (\text{immr} + 1) \) bits from the least significant bits of the source register to bit position \( (\text{regsize} - \text{immr}) \) of the destination register, where \( \text{regsize} \) is the destination register size of 32 or 64 bits.

In both cases the destination bits below the bitfield are set to zero, and the bits above the bitfield are set to a copy of the most significant bit of the bitfield.

This instruction is used by the aliases ASR (immediate), SBFIZ, SBFX, SXTB, SXTH, and SXTW. See Alias conditions on page C6-1778 for details of when each alias is preferred.

32-bit variant

Applies when \( sf = 0 \land N = 0 \).

\[
\text{SBFM} \ <Wd>, \ <Wn>, \ #\text{immr}, \ #\text{imms}\]

64-bit variant

Applies when \( sf = 1 \land N = 1 \).

\[
\text{SBFM} \ <Xd>, \ <Xn>, \ #\text{immr}, \ #\text{imms}\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } d & = \text{UInt}(Rd); \\
\text{integer } n & = \text{UInt}(Rn); \\
\text{integer datasize} & = \text{if } sf = '1' \text{ then 64 else 32}; \\
\text{integer } r; \\
\text{integer } s; \\
\text{bits(datasize) wmask}; \\
\text{bits(datasize) tmask}; \\
\text{if } sf = '1' \land N \neq '1' \text{ then UNDEFINED}; \\
\text{if } sf = '0' \land (N \neq '0' \lor \text{immr}<5> \neq '0' \lor \text{imms}<5> \neq '0') \text{ then UNDEFINED}; \\
\text{r} & = \text{UInt}(\text{immr}); \\
\text{s} & = \text{UInt}(\text{imms}); \\
(\text{wmask}, \text{tmask}) & = \text{DecodeBitMasks}(N, \text{imms}, \text{immr}, \text{FALSE}, \text{datasize});
\end{align*}
\]
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>of variant</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR (immediate)</td>
<td>32-bit</td>
<td>immr == '011111'</td>
</tr>
<tr>
<td>ASR (immediate)</td>
<td>64-bit</td>
<td>immr == '111111'</td>
</tr>
<tr>
<td>SBFIZ</td>
<td>-</td>
<td>UInt(imms) &lt; UInt(immr)</td>
</tr>
<tr>
<td>SBFX</td>
<td>-</td>
<td>BFXPreferred(sf, opc&lt;1&gt;, imms, immr)</td>
</tr>
<tr>
<td>SXTB</td>
<td>-</td>
<td>immr == '000000' &amp;&amp; imms == '000111'</td>
</tr>
<tr>
<td>SXTH</td>
<td>-</td>
<td>immr == '000000' &amp;&amp; imms == '001111'</td>
</tr>
<tr>
<td>SXTW</td>
<td>-</td>
<td>immr == '000000' &amp;&amp; imms == '011111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<immr> For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.

<imms> For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Operation

bits(datasize) src = X[n, datasize];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, r) AND wmask;

// determine extension bits (sign, zero or dest register)
bhits(datasize) top = Replicate(src<>), datasize);

// combine extension bits and result bits
X[d, datasize] = (top AND NOT(tmask)) OR (bot AND tmask);

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.269  SBFX

Signed Bitfield Extract copies a bitfield of `<width>` bits starting from bit position `<lsb>` in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to a copy of the most significant bit of the bitfield.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

**32-bit variant**

Applies when `sf == 0 && N == 0`.

\[
\text{SBFX} <Wd>, <Wn>, #<lsb>, #<width>
\]

is equivalent to

\[
\text{SBFM} <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
\]

and is the preferred disassembly when \(BFXPreferred(sf, opc<1>, imms, immr)\).

**64-bit variant**

Applies when `sf == 1 && N == 1`.

\[
\text{SBFX} <Xd>, <Xn>, #<lsb>, #<width>
\]

is equivalent to

\[
\text{SBFM} <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
\]

and is the preferred disassembly when \(BFXPreferred(sf, opc<1>, imms, immr)\).

**Assembler symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<lsb>` For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
- `<width>` For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

**Operation**

The description of SBFM gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.270  SDIV

Signed Divide divides a signed integer register value by another signed integer register value, and writes the result to the destination register. The condition flags are not affected.

32-bit variant
Applies when \( sf = 0 \).

\[
\text{SDIV} <Wd>, <Wn>, <Wm>
\]

64-bit variant
Applies when \( sf = 1 \).

\[
\text{SDIV} <Xd>, <Xn>, <Xm>
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } d &= \text{UInt(Rd);} \\
\text{integer } n &= \text{UInt(Rn);} \\
\text{integer } m &= \text{UInt(Rm);} \\
\text{integer } \text{datasize} &= \text{if } sf = \text{'1'} \text{ then } 64 \text{ else } 32;
\end{align*}
\]

Assembler symbols

\(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

\(<Wm>\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

\(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

\(< Xm>\) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

\[
\begin{align*}
\text{bits(datasize) operand1} &= X[n, \text{datasize}]; \\
\text{bits(datasize) operand2} &= X[m, \text{datasize}]; \\
\text{integer } \text{result} &= ; \\
\text{if } \text{IsZero(operand2)} \text{ then } \\
\text{result} &= 0; \\
\text{else } \\
\text{result} &= \text{RoundTowardsZero(Real(Int(operand1, FALSE)) / Real(Int(operand2, FALSE)))}; \\
X[d, \text{datasize}] &= \text{result}<\text{datasize}-1:0>;
\end{align*}
\]
C6.2.271 SETF8, SETF16

Set the PSTATE.NZV flags based on the value in the specified general-purpose register. SETF8 treats the value as an 8 bit value, and SETF16 treats the value as an 16 bit value.

The PSTATE.C flag is not affected by these instructions.

**Integer**

(FEAT_FlagM)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1</td>
<td>1 1 0 1</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>sz</td>
<td>0</td>
</tr>
</tbody>
</table>

**SETF8 variant**

Applies when sz == 0.

SETF8 <Wn>

**SETF16 variant**

Applies when sz == 1.

SETF16 <Wn>

**Decode for all variants of this encoding**

```plaintext
if !HaveFlagManipulateExt() then UNDEFINED;
integer msb = if sz == '1' then 15 else 7;
integer n = UInt(Rn);
```

**Assembler symbols**

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

```
bits(32) tmpreg = X[n, 32];
PSTATE.N = tmpreg<msb>;
PSTATE.Z = if (tmpreg<msb:0> == Zeros(msb + 1)) then '1' else '0';
PSTATE.V = tmpreg<msb> EOR tmpreg<msb>;
//PSTATE.C unchanged;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.272 SETGP, SETGM, SETGE

Memory Set with tag setting. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGP, then SETGM, and then SETGE.

SETGP performs some preconditioning of the arguments suitable for using the SETGM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGM performs an IMPLEMENTATION DEFINED amount of the memory set. SETGE performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGP, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0xfffffffffffffff0.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGP, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0xfffffffffffffff0.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGM, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGM, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.
For SETGE, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -$1$ the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -$Xn$.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGE, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

### Integer

(FEAT_MOPS)

| sz | 0 1 1 1 0 1 1 0 | Rs | x x 0 0 0 1 | Rn | Rd |
|---|---|---|---|---|---|---|
| op2 |

### Epilogue variant

Applies when op2 == 1000.

SETGE [Xdb], Xn!, Xs

### Main variant

Applies when op2 == 0100.

SETGM [Xdb], Xn!, Xs

### Prologue variant

Applies when op2 == 0000.

SETGP [Xdb], Xn!, Xs

### Decode for all variants of this encoding

```c
if !HaveFeatMOPS() then UNDEFINED;
if !HaveMTEExt() then UNDEFINED;
if sz != '00' then UNDEFINED;
integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;
MOPSStage stage;
case op2<3:2> of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise UNDEFINED;
```
if $s == n || s == d || n == d$ then UNDEFINED;
if $d == 31 || n == 31$ then UNDEFINED;

**Assemble symbols**

$<Xd>$  
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

$<Xn>$  
For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

$<Xs>$  
For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

**Operation**

CheckMOPSEnabled();

bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagesetsize;
boolean is_setg = TRUE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSStage_Prologue then
    if setsize63> == '1' then setsize = 0x7FFFFFFFFFFFFF0<63:0>;
    if setsize != Zeros(64) & toaddress != Align(toaddress, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if setsize != Align(setsize, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if supports_option_a then
        nzcv = '0000';
        toaddress = toaddress + setsize;
setsize = Zeros(64) - setsize;
else
  nzcv = '0010';

stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros(64);
assert stagesetsize<3:0> == '0000';

if SInt(setsize) > 0 then
  assert SInt(stagesetsize) <= SInt(setsize);
else
  assert SInt(stagesetsize) >= SInt(setsize);
else
bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
assert postsize<63> == setsize<63> || postsize == Zeros(64);
assert postsize<3:0> == '0000';

boolean zero_size_exceptions = MemSetZeroSizeCheck();

// Check if this version is consistent with the state of the call.
if zero_size_exceptions || SInt(setsize) != 0 then
  if supports_option_a then
    if nzcv<1> == '1' then // PSTATE.C
      boolean wrong_option = TRUE;
      boolean from_epilogue = stage == MOPSStage_Epilogue;
      MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
    else
      if nzcv<1> == '0' then // PSTATE.C
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
  
  if stage == MOPSStage_Main then
    stagesetsize = setsize - postsize;
    if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
      boolean wrong_option = FALSE;
      boolean from_epilogue = FALSE;
      MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
    else
      stagesetsize = postsize;
      if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = TRUE;
        MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
  
  if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
    boolean iswrite = TRUE;
    boolean secondstage = FALSE;
    AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));
  
  if setsize != Align(setsize, TAG_GRANULE) then
    boolean iswrite = TRUE;
    boolean secondstage = FALSE;
    AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

integer tagstep;
bits(4) tag;
bits(64) tagaddr;

if supports_option_a then
  while SInt(stagesetsize) < 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = SETSizeChoice(toaddress, setsize, 16);
assert B <= \(-1 \times \text{SInt(stagesetsize)}\);
assert B<3:0> == '0000';

Mem[toaddress+setsizesize, B, acctype] = \text{Replicate}(data, B);

tagstep = B \text{DIV} 16;
tag = \text{AArch64.AllocationTagFromAddress}(toaddress + setsizesize);
while tagstep > 0 do
tagaddr = toaddress + setsizesize + (tagstep - 1) \times 16;
\text{AArch64.MemTag}[tagaddr, acctype] = tag;
tagstep = tagstep - 1;

setsizesize = setsizesize + B;
stagesetsizesize = stagesetsizesize + B;
if stage \neq \text{MOPSStage_Prologue} then
X[n, 64] = setsizesize;
else
while \text{UInt(stagesetsizesize)} > 0 do
// IMP DEF selection of the block size that is worked on. While many
// implementations might make this constant, that is not assumed.
B = \text{SETSizeChoice}(toaddress, setsizesize, 16);
assert B <= \text{UInt(stagesetsizesize)};
assert B<3:0> == '0000';

Mem[toaddress, B, acctype] = \text{Replicate}(data, B);

tagstep = B \text{DIV} 16;
tag = \text{AArch64.AllocationTagFromAddress}(toaddress);
while tagstep > 0 do
tagaddr = toaddress + (tagstep - 1) \times 16;
\text{AArch64.MemTag}[tagaddr, acctype] = tag;
tagstep = tagstep - 1;
toaddress = toaddress + B;
setsizesize = setsizesize - B;
stagesetsizesize = stagesetsizesize - B;
if stage \neq \text{MOPSStage_Prologue} then
X[n, 64] = setsizesize;
X[d, 64] = toaddress;
if stage == \text{MOPSStage_Prologue} then
X[n, 64] = setsizesize;
X[d, 64] = toaddress;
PSTATE.<N,Z,C,V> = nzcv;
C6.2.273  SETGPN, SETGMN, SETGEN

Memory Set with tag setting, non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPN, then SETGMN, and then SETGEN.

SETGPN performs some preconditioning of the arguments suitable for using the SETGMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGMN performs an IMPLEMENTATION DEFINED amount of the memory set. SETGEN performs the last part of the memory set.

—— Note ———

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

—— Note ———

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGPN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFF0.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGPN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFF0.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.
For SETGEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

### Integer

(FEAT_MOPS)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

### Epilogue variant

Applies when `op2 == 1010`.

`SETGEN [<Xd>]!, <Xn>!, <Xs>`

### Main variant

Applies when `op2 == 0110`.

`SETGMN [<Xd>]!, <Xn>!, <Xs>`

### Prologue variant

Applies when `op2 == 0010`.

`SETGPN [<Xd>]!, <Xn>!, <Xs>`

### Decode for all variants of this encoding

```c
if !HaveFeatMOPS() then UNDEFINED;
if !HaveMTEExt() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;

MOPSStage stage;
case op2<3:2> of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;
```
if \( s == n \) || \( s == d \) || \( n == d \) then UNDEFINED;
if \( d == 31 \) || \( n == 31 \) then UNDEFINED;

**Assemble symbols**

\( <X_d> \)

For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

\( <X_n> \)

For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

\( <X_s> \)

For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

**Operation**

CheckMOPSEnabled();

bits(64) toaddress = \( X[d, 64] \);
bits(64) setsize = \( X[n, 64] \);
bits(8) data = \( X[s, 8] \);
bits(4) ncvc = PSTATE.<N,Z,C,V>;
b bits(64) stagesetsize;
boolean is_setg = TRUE;
integer B;

if HaveMTE2Ext() then
   SetTagCheckedInstruction(FALSE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSStage_Prologue then
   if setsize<63> == '1' then setsize = 0x7FFFFFFFF0<63:0>;
   if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
      boolean iswrite = TRUE;
      boolean secondstage = FALSE;
      AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));
   if setsize != Align(setsize, TAG_GRANULE) then
      boolean iswrite = TRUE;
      boolean secondstage = FALSE;
      AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));
   if supports_option_a then
      ncvc = '0000';
      toaddress = toaddress + setsize;
setsize = Zeros(64) - setsize;
else
    nzcv = '0010';

stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros(64);
assert stagesetsize<3:0> == '0000';

if SInt(setsize) > 0 then
    assert SInt(stagesetsize) <= SInt(setsize);
else
    assert SInt(stagesetsize) >= SInt(setsize);
else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
assert postsize<63> == setsize<63> || postsize == Zeros(64);
assert postsize<3:0> == '0000';

boolean zero_size_exceptions = MemSetZeroSizeCheck();

if zero_size_exceptions || SInt(setsize) != 0 then
    if supports_option_a then
        if nzcv<1> == '1' then // PSTATE.C
            boolean wrong_option = TRUE;
            boolean from_epilogue = stage == MOPSStage_Epilogue;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue,
                                    options, is_setg);
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue,
                                            options, is_setg);
            if stage == MOPSStage_Main then
                stagesetsize = setsize - postsize;
                if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
                    boolean wrong_option = FALSE;
                    boolean from_epilogue = FALSE;
                    MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue,
                                                options, is_setg);
                else
                    stagesetsize = postsize;
                    if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
                        boolean wrong_option = FALSE;
                        boolean from_epilogue = TRUE;
                        MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue,
                                                  options, is_setg);
                    if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
                        boolean iswrite = TRUE;
                        boolean secondstage = FALSE;
                        AArch64 Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));
                    if setsize != Align(setsize, TAG_GRANULE) then
                        boolean iswrite = TRUE;
                        boolean secondstage = FALSE;
                        AArch64 Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    integer tagstep;
    bits(4) tag;
    bits(64) tagaddr;

    if supports_option_a then
        while SInt(stagesetsize) < 0 do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = SETSizeChoice(toaddress, setsize, 16);
assert B <= -1 * SInt(stagesetsize);
assert B<3:0> == "0000";

Mem[toaddress+setsize, B, acctype] = Replicate(data, B);

tagstep = B DIV 16;
tag = AArch64.AllocationTagFromAddress(toaddress + setsize);
while tagstep > 0 do
tagaddr = toaddress + setsize + (tagstep - 1) * 16;
AArch64.MemTag[tagaddr, acctype] = tag;
tagstep = tagstep - 1;

setsizes = setsizes + B;
stagesetsizes = stagesetsizes + B;
if stage == MOPSStage_Prologue then
  X[n, 64] = setsizes;
else
  while UInt(stagesetsizes) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = SETSizeChoice(toaddress, setsizes, 16);
    assert B <= UInt(stagesetsizes);
    assert B<3:0> == "0000";
    Mem[toaddress, B, acctype] = Replicate(data, B);

tagstep = B DIV 16;
tag = AArch64.AllocationTagFromAddress(toaddress);
while tagstep > 0 do
tagaddr = toaddress + (tagstep - 1) * 16;
AArch64.MemTag[tagaddr, acctype] = tag;
tagstep = tagstep - 1;
toaddress = toaddress + B;
setsizes = setsizes - B;
stagesetsizes = stagesetsizes - B;
if stage == MOPSStage_Prologue then
  X[n, 64] = setsizes;
  X[d, 64] = toaddress;
if stage == MOPSStage_Prologue then
  X[n, 64] = setsizes;
  X[d, 64] = toaddress;
PSTATE.<N,Z,C,V> = nzcv;
C6.2.274 SETGPT, SETGMT, SETGET

Memory Set with tag setting, unprivileged. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPT, then SETGMT, and then SETGET.

SETGPT performs some preconditioning of the arguments suitable for using the SETGMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGMT performs an IMPLEMENTATION DEFINED amount of the memory set. SETGET performs the last part of the memory set.

——— Note ————

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

——— Note ————

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGPT, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFF0.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGPT, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFF0.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  — the value of Xd is written back with the lowest address that has not been set.
For SETGET, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- \( X_n \) is treated as a signed 64-bit number.
- \( X_n \) holds \(-1 \times \) the number of bytes remaining to be set in the memory set in total.
- \( X_d \) holds the lowest address that the set is made to \(-X_n\).
- At the end of the instruction, the value of \( X_n \) is written back with 0.

For SETGET, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- \( X_n \) holds the number of bytes remaining to be set in the memory set in total.
- \( X_d \) holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of \( X_n \) is written back with 0.
  - the value of \( X_d \) is written back with the lowest address that has not been set.

**Integer**

(FEAT_MOPS)

\[
\begin{array}{ccccccccccc}
\hline
sz & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & x & x & 1 & 0 & 1 & 0 & 1 & Rn & Rd \\
\end{array}
\]

\( op2 \)

**Epilogue variant**

Applies when \( op2 = 1001 \).

\[ \text{SETGET} \ (<X_d>!), <X_n>!, <X_s> \]

**Main variant**

Applies when \( op2 = 0101 \).

\[ \text{SETGMT} \ (<X_d>!), <X_n>!, <X_s> \]

**Prologue variant**

Applies when \( op2 = 0001 \).

\[ \text{SETGPT} \ (<X_d>!), <X_n>!, <X_s> \]

**Decode for all variants of this encoding**

\[
\text{if } \text{!HaveFeatMOPS()} \text{ then UNDEFINED;}
\text{if } \text{!HaveMTEExt()} \text{ then UNDEFINED;}
\text{if } sz \neq '00' \text{ then UNDEFINED;}
\]

\[
\text{integer } d = \text{UInt}(Rd);
\text{integer } s = \text{UInt}(Rs);
\text{integer } n = \text{UInt}(Rn);
\text{bits}(2) \text{ options } = \text{op2<1:0>};
\text{MOPSStage } \text{ stage;}
\text{case op2<3:2> of}
\text{when } '00' \text{ stage } = \text{MOPSStage_Prologue};
\text{when } '01' \text{ stage } = \text{MOPSStage_Main};
\text{when } '10' \text{ stage } = \text{MOPSStage_Epilogue};
\text{otherwise UNDEFINED;}
\]

---

Printed from Arm Documentation Database on 2023-06-17
if \( s == n || s == d || n == d \) then UNDEFINED;
if \( d == 31 || n == 31 \) then UNDEFINED;

### Assembler symbols

\(<Xd>\)
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

\(<Xn>\)
For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

\(<Xs>\)
For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

### Operation

CheckMOPSEnabled();

bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagesetsize;
boolean is_setg = TRUE;
integer B;

if HaveMTE2Ext() then
SetTagCheckedInstruction(FALSE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSStage_Prologue then
if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFFF<63:0>;

if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
boolean iswrite = TRUE;
boolean secondstage = FALSE;
AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

if setsize != Align(setsize, TAG_GRANULE) then
boolean iswrite = TRUE;
boolean secondstage = FALSE;
AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

if supports_option_a then
nzcv = '0000';
toaddress = toaddress + setsize;
setsize = Zeros(64) - setsize;
else
    nzcv = '0010';

stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros(64);
assert stagesetsize<3:0> == '0000';
if SInt(setsize) > 0 then
    assert SInt(stagesetsize) <= SInt(setsize);
else
    assert SInt(stagesetsize) >= SInt(setsize);

bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
assert postsize<63> == setsize<63> || postsize == Zeros(64);
assert postsize<3:0> == '0000';

boolean zero_size_exceptions = MemSetZeroSizeCheck();

// Check if this version is consistent with the state of the call.
if zero_size_exceptions || SInt(setsize) != 0 then
    if supports_option_a then
        if nzcv<1> == '1' then // PSTATE.C
            boolean wrong_option = TRUE;
            boolean from_epilogue = stage == MOPSStage_Epilogue;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
        if stage == MOPSStage_Main then
            stagesetsize = setsize - postsize;
        if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
        else
            stagesetsize = postsize;
            if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = TRUE;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
        if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
            boolean iswrite = TRUE;
            boolean secondstage = FALSE;
            AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));
        if setsize != Align(setsize, TAG_GRANULE) then
            boolean iswrite = TRUE;
            boolean secondstage = FALSE;
            AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

        integer tagstep;
        bits(4) tag;
        bits(64) tagaddr;
        if supports_option_a then
            while SInt(stagesetsize) < 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = SETSizeChoice(toaddress, setsize, 16);
assert B <= -1 * SInt(stagesetsize);
assert B<3:0> == '0000';

Mem[toaddress+setsize, B, acctype] = Replicate(data, B);

tagstep = B DIV 16;
tag = AArch64.AllocationTagFromAddress(toaddress + setsize);
while tagstep > 0 do
  tagaddr = toaddress + setsize + (tagstep - 1) * 16;
  AArch64.MemTag[tagaddr, acctype] = tag;
  tagstep = tagstep - 1;

setsize = setsize + B;
stagesetsize = stagesetsize + B;
if stage != MOPSStage_Prologue then
  X[n, 64] = setsize;
else
  while UInt(stagesetsize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = SETSizeChoice(toaddress, setsize, 16);
    assert B <= UInt(stagesetsize);
    assert B<3:0> == '0000';
    Mem[toaddress, B, acctype] = Replicate(data, B);

    tagstep = B DIV 16;
tag = AArch64.AllocationTagFromAddress(toaddress);
    while tagstep > 0 do
      tagaddr = toaddress + (tagstep - 1) * 16;
      AArch64.MemTag[tagaddr, acctype] = tag;
      tagstep = tagstep - 1;
    end while
    toaddress = toaddress + B;
    setsize = setsize - B;
stagesetsize = stagesetsize - B;
    if stage != MOPSStage_Prologue then
      X[n, 64] = setsize;
      X[d, 64] = toaddress;
    end if
  end while
if stage == MOPSStage_Prologue then
  X[n, 64] = setsize;
  X[d, 64] = toaddress;
PSTATE.<N,Z,C,V> = nzcv;
C6.2.275   SETGPTN, SETGMTN, SETGETN

Memory Set with tag setting, unprivileged and non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPTN, then SETGMTN, and then SETGETN.

SETGPTN performs some preconditioning of the arguments suitable for using the SETGMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGMTN performs an IMPLEMENTATION DEFINED amount of the memory set. SETGETN performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

Note

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGPTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFF0.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGPTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFF0.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.
For SETGETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

**Integer**

**(FEAT_MOPS)**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11 10 9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>0 1 1</td>
<td>0 1</td>
<td>1 1</td>
<td>Rs</td>
<td>x</td>
<td>x</td>
<td>1 1</td>
<td>0 1</td>
<td>Rn</td>
</tr>
</tbody>
</table>
```

**Epilogue variant**

Applies when op2 == 1011.

SETGETN [<Xd>], <Xn>, <Xs>

**Main variant**

Applies when op2 == 0111.

SETGTMN [<Xd>], <Xn>, <Xs>

**Prologue variant**

Applies when op2 == 0011.

SETGPTN [<Xd>], <Xn>, <Xs>

**Decode for all variants of this encoding**

```
if !HaveFeatMOPS() then UNDEFINED;
if !HaveMTEExt() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;

MOPSStage stage;
case op2<3:2> of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise UNDEFINED;
```
if s == n || s == d || n == d then UNDEFINED;
if d == 31 || n == 31 then UNDEFINED;

Assembler symbols

<Xd>  For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

<Xn>  For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

<Xs>  For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

Operation

CheckMOPSEnabled();

bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagesetsize;
boolean is_setg = TRUE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSStrage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFF0<63:0>;
    if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));
    if setsize != Align(setsize, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));
    if supports_option_a then
        nzcv = '0000';
        toaddress = toaddress + setsize;
setsize = Zeros(64) - setsize;
else
  nzcv = '0010';

stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros(64);
assert stagesetsize<3:0> == '0000';
if SInt(setsize) > 0 then
  assert SInt(stagesetsize) <= SInt(setsize);
else
  assert SInt(stagesetsize) >= SInt(setsize);
else
  bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
  assert postsize<63> == setsize<63> || postsize == Zeros(64);
  assert postsize<3:0> == '0000';

boolean zero_size_exceptions = MemSetZeroSizeCheck();

// Check if this version is consistent with the state of the call.
if zero_size_exceptions || SInt(setsize) != 0 then
  if supports_option_a then
    if nzcv<1> == '1' then // PSTATE.C
      boolean wrong_option = TRUE;
      boolean from_epilogue = stage == MOPSStage_Epilogue;
      MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue,
      options, is_setg);
    else
      if nzcv<1> == '0' then // PSTATE.C
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue,
        options, is_setg);
  else
    if stage == MOPSStage_Main then
      stagesetsize = setsize - postsize;
      if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = FALSE;
        MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue,
        options, is_setg);
      else
        stagesetsize = postsize;
        if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
          boolean wrong_option = FALSE;
          boolean from_epilogue = TRUE;
          MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue,
          options, is_setg);
    else
      integer tagstep;
      bits(4) tag;
      bits(64) tagaddr;
      if supports_option_a then
        while SInt(stagesetsize) < 0 do
          // IMP DEF selection of the block size that is worked on. While many
          // implementations might make this constant, that is not assumed.
          B = SETSizeChoice(toaddress, setsize, 16);
assert B <= -1 * SInt(stagesetsize);
assert B<3:0> == '0000';

Mem[toaddress+setsize, B, acctype] = Replicate(data, B);

tagstep = B DIV 16;
tag = AArch64.AllocationTagFromAddress(toaddress + setsize);
while tagstep > 0 do 
tagaddr = toaddress + setsize + (tagstep - 1) * 16;
AArch64.MemTag[tagaddr, acctype] = tag;
tagstep = tagstep - 1;

setsize = setsize + B;
stagesetsize = stagesetsize + B;
if stage != MOPSStage_Prologue then 
X[n, 64] = setsize;
else
while UInt(stagesetsize) > 0 do 
// IMP DEF selection of the block size that is worked on. While many
// implementations might make this constant, that is not assumed.
B = SETSizeChoice(toaddress, setsize, 16);
assert B <= UInt(stagesetsize);
assert B<3:0> == '0000';
Mem[toaddress, B, acctype] = Replicate(data, B);

tagstep = B DIV 16;
tag = AArch64.AllocationTagFromAddress(toaddress);
while tagstep > 0 do 
tagaddr = toaddress + (tagstep - 1) * 16;
AArch64.MemTag[tagaddr, acctype] = tag;
tagstep = tagstep - 1;

toaddress = toaddress + B;
setsize = setsize - B;
stagesetsize = stagesetsize - B;
if stage != MOPSStage_Prologue then 
X[n, 64] = setsize;
X[d, 64] = toaddress;
if stage == MOPSStage_Prologue then 
X[n, 64] = setsize;
X[d, 64] = toaddress;
PSTATE.<N,Z,C,V> = nzcv;
C6.2.276 SETP, SETM, SETE

Memory Set. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETP, then SETM, and then SETE.

SETP performs some preconditioning of the arguments suitable for using the SETM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETM performs an IMPLEMENTATION DEFINED amount of the memory set. SETE performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETP, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.\{N,Z,V\} are set to \{0,0,0\}.

After execution of SETP, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.\{N,Z,V\} are set to \{0,0,0\}.

For SETM, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be set in the memory set in total.

For SETM, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  — the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  — the value of Xd is written back with the lowest address that has not been set.

For SETE, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
• Xn holds -$1^*$ the number of bytes remaining to be set in the memory set in total.
• Xd holds the lowest address that the set is made to -$Xn$.
• At the end of the instruction, the value of Xn is written back with 0.

For SETE, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be set in the memory set in total.
• Xd holds the lowest address that the set is made to.
• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xd is written back with the lowest address that has not been set.

**Integer**

(Feat_MOPS)

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>0 1 1 0 0 1 1 0</td>
<td>Rs  x  x  0 0 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Epilogue variant**

Applies when op2 == 1000.

SETE [Xd]!, <Xn>!, <Xs>

**Main variant**

Applies when op2 == 0100.

SETM [Xd]!, <Xn>!, <Xs>

**Prologue variant**

Applies when op2 == 0000.

SETP [Xd]!, <Xn>!, <Xs>

**Decode for all variants of this encoding**

```c
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;
MOPSStage stage;
case op2<3:2> of
when '00' stage = MOPSStage_Prologue;
when '01' stage = MOPSStage_Main;
when '10' stage = MOPSStage_Epilogue;
otherwise UNDEFINED;

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || n == 31 then
c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
assert c IN {Constraint_UNDEF, Constraint_NOP};
```
case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP   EndOfInstruction();

Assembler symbols

<Xd>  For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xn>  For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

<Xs>  Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagesetsize;
boolean is_setg = FALSE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);
if stage == MOPSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFFF<63:0>;
    if supports_option_a then
        nzcv = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcv = '0010';

    stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
    assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros(64);

    if SInt(setsize) > 0 then
        assert SInt(stagesetsize) <= SInt(setsize);
    else
        assert SInt(stagesetsize) >= SInt(setsize);
else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    assert postsize<63> == setsize<63> || postsize == Zeros(64);

    boolean zero_size_exceptions = MemSetZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(setsize) != 0 then
if supports_option_a then
  if nzcv<1> == '1' then // PSTATE.C
    boolean wrong_option = TRUE;
    boolean from_epilogue = stage == MOPSStage_Epilogue;
    MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
  else
    if nzcv<1> == '0' then // PSTATE.C
      boolean wrong_option = TRUE;
      boolean from_epilogue = stage == MOPSStage_Epilogue;
      MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
  if stage == MOPSStage_Main then
    stagesetsize = setsize - postsize;
    if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
      boolean wrong_option = FALSE;
      boolean from_epilogue = FALSE;
      MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
    else
      stagesetsize = postsize;
      if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = TRUE;
        MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
  if supports_option_a then
    while SInt(stagesetsize) < 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = SETSizeChoice(toaddress, setsize, 1);
      assert B <= -1 * SInt(stagesetsize);
      Mem[toaddress+setsize, B, acctype] = Replicate(data, B);
      setsize = setsize + B;
      stagesetsize = stagesetsize + B;
      if stage != MOPSStage_Prologue then
        X[n, 64] = setsize;
    else
      while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= UInt(stagesetsize);
        Mem[toaddress, B, acctype] = Replicate(data, B);
        toaddress = toaddress + B;
        setsize = setsize - B;
        stagesetsize = stagesetsize - B;
        if stage != MOPSStage_Prologue then
          X[n, 64] = setsize;
          X[d, 64] = toaddress;
    if stage == MOPSStage_Prologue then
      X[n, 64] = setsize;
      X[d, 64] = toaddress;
      PSTATE.<N,Z,C,V> = nzcv;
C6.2.277 SETPN, SETMN, SETEN

Memory Set, non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPN, then SETMN, and then SETEN.

SETPN performs some preconditioning of the arguments suitable for using the SETMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETMN performs an IMPLEMENTATION DEFINED amount of the memory set. SETEN performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

Note

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETPN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to \(0x7FFFFFFFFFFFFFFF\).
- Xd holds the original Xd + saturated Xn.
- Xn holds \(-1\times\text{saturated Xn} + \text{an IMPLEMENTATION DEFINED number of bytes set}\).
- PSTATE.\{N,Z,V\} are set to \{0,0,0\}.

After execution of SETPN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to \(0x7FFFFFFFFFFFFFFF\).
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.\{N,Z,V\} are set to \{0,0,0\}.

For SETMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds \(-1\times\text{number of bytes remaining to be set in the memory set in total}\).
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with \(-1\times\text{the number of bytes remaining to be set in the memory set in total}\).

For SETMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
• Xn holds -1* the number of bytes remaining to be set in the memory set in total.
• Xd holds the lowest address that the set is made to -Xn.
• At the end of the instruction, the value of Xn is written back with 0.

For SETEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be set in the memory set in total.
• Xd holds the lowest address that the set is made to.
• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xd is written back with the lowest address that has not been set.

**Integer**

(FEAT_MOPS)

| 31 30 29 28|27 26 25 24|23 22 21 20| 16|15|12|11|10 | 9 | 5 | 4 | 0 |
| sz | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | Rs | x | x | 1 | 0 | 0 | 1 | Rn | Rd |

**Epilogue variant**

Applies when `op2 == 1010`.

`SETEN [<Xd>], <Xn>, <Xs>`

**Main variant**

Applies when `op2 == 0110`.

`SETMN [<Xd>], <Xn>, <Xs>`

**Prologue variant**

Applies when `op2 == 0010`.

`SETPN [<Xd>], <Xn>, <Xs>`

**Decode for all variants of this encoding**

```c
if !HaveFeatMOPS() then UNDEFINED;
if sz !='00' then UNDEFINED;
  integer d = UInt(Rd);
  integer s = UInt(Rs);
  integer n = UInt(Rn);
  bits(2) options = op2<1:0>;
  MOPSStage stage;
  case op2<3:2> of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
  otherwise UNDEFINED;
  CheckMOPSEnabled();

  if s == n || s == d || n == d || d == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
```

```c
```
case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP   EndOfInstruction();

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.
    For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field.
    For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.
    For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

<Xs> Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagesetsize;
boolean is_setg = FALSE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);
if stage == MOPStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcv = '0010';
        stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
    assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros(64);
    if SInt(setsize) > 0 then
        assert SInt(stagesetsize) <= SInt(setsize);
    else
        assert SInt(stagesetsize) >= SInt(setsize);
    else
        bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
        assert postsize<63> == setsize<63> || postsize == Zeros(64);

        boolean zero_size_exceptions = MemSetZeroSizeCheck();
        // Check if this version is consistent with the state of the call.
        if zero_size_exceptions || SInt(setsize) != 0 then
if supports_option_a then
  if nzcv<1> == '1' then // PSTATE.C
    boolean wrong_option = TRUE;
    boolean from_epilogue = stage == MOPSStage_Epilogue;
    MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue,
    options, is_setg);
  else
    if nzcv<1> == '0' then // PSTATE.C
      boolean wrong_option = TRUE;
      boolean from_epilogue = stage == MOPSStage_Epilogue;
      MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue,
      options, is_setg);
  if stage == MOPSStage_Main then
    stagesetsize = setsize - postsize;
    if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
      boolean wrong_option = FALSE;
      boolean from_epilogue = FALSE;
      MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options,
      is_setg);
    else
      stagesetsize = postsize;
      if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = TRUE;
        MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options,
        is_setg);
  if supports_option_a then
    while SInt(stagesetsize) < 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = SETSizeChoice(toaddress, setsize, 1);
      assert B <= -1 * SInt(stagesetsize);
      Mem[toaddress+setsize, B, acctype] = Replicate(data, B);
      setsize = setsize + B;
      stagesetsize = stagesetsize + B;
      if stage != MOPSStage_Prologue then
        X[n, 64] = setsize;
  else
    while UInt(stagesetsize) > 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = SETSizeChoice(toaddress, setsize, 1);
      assert B <= UInt(stagesetsize);
      Mem[toaddress, B, acctype] = Replicate(data, B);
      toaddress = toaddress + B;
      setsize = setsize - B;
      stagesetsize = stagesetsize - B;
      if stage != MOPSStage_Prologue then
        X[n, 64] = setsize;
        X[d, 64] = toaddress;
if stage == MOPSStage_Prologue then
  X[n, 64] = setsize;
  X[d, 64] = toaddress;
PSTATE.<N,Z,C,V> = nzcv;
C6.2.278   SETPT, SETMT, SETET

Memory Set, unprivileged. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPT, then SETMT, and then SETET.

SETPT performs some preconditioning of the arguments suitable for using the SETMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETMT performs an IMPLEMENTATION DEFINED amount of the memory set. SETET performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETPT, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0xffffffffffffffff.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETPT, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0xffffffffffffffff.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be set in the memory set in total.

For SETMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETET, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
• Xn holds \(-1^n\) the number of bytes remaining to be set in the memory set in total.
• Xd holds the lowest address that the set is made to -Xn.
• At the end of the instruction, the value of Xn is written back with 0.

For SETET, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be set in the memory set in total.
• Xd holds the lowest address that the set is made to.
• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xd is written back with the lowest address that has not been set.

**Integer**

(FEAT_MOPS)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rs</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

**Epilogue variant**

Applies when op2 == 1001.
SETET [Xd]!, Xn!, Xs

**Main variant**

Applies when op2 == 0101.
SETMT [Xd]!, Xn!, Xs

**Prologue variant**

Applies when op2 == 0001.
SETPT [Xd]!, Xn!, Xs

**Decode for all variants of this encoding**

```plaintext
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
b看他 bits(2) options = op2<1:0>;

MOPSSStage stage;
case op2<3:2> of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise UNDEFINED;

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || n == 31 then
  c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
```
case c of
  when Constraint_UNDEF UNDEFINED;
  when Constraint_NOP EndOfInstruction();

Assembler symbols

<Xd>
  For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.
  For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xn>
  For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field.
  For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.
  For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

<Xs>
  Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagesetsize;
boolean is_setg = FALSE;
integer B;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);
boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);
if stage == MOPSStage_Prologue then
  if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFFF<63:0>;
  if supports_option_a then
    nzcv = '0000';
    toaddress = toaddress + setsize;
    setsize = Zeros(64) - setsize;
  else
    nzcv = '0010';
    stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
  assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros(64);
  if SInt(setsize) > 0 then
    assert SInt(stagesetsize) <= SInt(setsize);
  else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    assert postsize<63> == setsize<63> || postsize == Zeros(64);
  boolean zero_size_exceptions = MemSetZeroSizeCheck();
  // Check if this version is consistent with the state of the call.
  if zero_size_exceptions || SInt(setsize) != 0 then
if supports_option_a then
  if nzcv<1> == '1' then // PSTATE.C
    boolean wrong_option = TRUE;
    boolean from_epilogue = stage == MOPSStage_Epilogue;
    MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue,
    options, is_setg);
  else
    if nzcv<1> == '0' then // PSTATE.C
      boolean wrong_option = TRUE;
      boolean from_epilogue = stage == MOPSStage_Epilogue;
      MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue,
      options, is_setg);
if stage == MOPSStage_Main then
  stagesetsize = setsize - postsize;
  if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options,
    is_setg);
  else
    stagesetsize = postsize;
    if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
      boolean wrong_option = FALSE;
      boolean from_epilogue = TRUE;
      MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options,
      is_setg);
if supports_option_a then
  while SInt(stagesetsize) < 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = SETSizeChoice(toaddress, setsize, 1);
    assert B <= -1 * SInt(stagesetsize);
    Mem[toaddress+setsize, B, acctype] = Replicate(data, B);
    setsize = setsize + B;
    stagesetsize = stagesetsize + B;
    if stage != MOPSStage_Prologue then
      X[n, 64] = setsize;
  else
    while UInt(stagesetsize) > 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = SETSizeChoice(toaddress, setsize, 1);
      assert B <= UInt(stagesetsize);
      Mem[toaddress, B, acctype] = Replicate(data, B);
      toaddress = toaddress + B;
      setsize = setsize - B;
      stagesetsize = stagesetsize - B;
      if stage != MOPSStage_Prologue then
        X[n, 64] = setsize;
        X[d, 64] = toaddress;
if stage == MOPSStage_Prologue then
  X[n, 64] = setsize;
  X[d, 64] = toaddress;
PSTATE.<N,Z,C,V> = nzcv;
C6.2.279  SETPTN, SETMTN, SETETN

Memory Set, unprivileged and non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPTN, then SETMTN, and then SETETN.

SETPTN performs some preconditioning of the arguments suitable for using the SETMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETMTN performs an IMPLEMENTATION DEFINED amount of the memory set. SETETN performs the last part of the memory set.

--- Note ---

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

--- Note ---

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

--- Note ---

Portable software should not assume that the choice of algorithm is constant.

After execution of SETPTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1 * saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETPTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1 * number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1 * the number of bytes remaining to be set in the memory set in total.

For SETMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
• Xn holds -1\(^*\) the number of bytes remaining to be set in the memory set in total.
• Xd holds the lowest address that the set is made to -Xn.
• At the end of the instruction, the value of Xn is written back with 0.

For SETETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be set in the memory set in total.
• Xd holds the lowest address that the set is made to.
• At the end of the instruction:
  — the value of Xn is written back with 0.
  — the value of Xd is written back with the lowest address that has not been set.

**Integer**

(FEAT_MOPS)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 15</th>
<th>14 12 10 9</th>
<th>8 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>0 1 1 1</td>
<td>0 1 1 1</td>
<td>Rs</td>
<td>x x 1 1 0 1</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

**Epilogue variant**

Applies when \(\text{op2} = \texttt{1111}\).

SETETN \([\langle Xd \rangle], \langle Xn \rangle, \langle Xs \rangle\)

**Main variant**

Applies when \(\text{op2} = \texttt{0111}\).

SETMTN \([\langle Xd \rangle], \langle Xn \rangle, \langle Xs \rangle\)

**Prologue variant**

Applies when \(\text{op2} = \texttt{0011}\).

SETPTN \([\langle Xd \rangle], \langle Xn \rangle, \langle Xs \rangle\)

**Decode for all variants of this encoding**

```c
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;

MOPSStage stage;
case op2<3:2> of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
otherwise UNDEFINED;

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || n == 31 then
  c = ConstraintUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
```
case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP   EndOfInstruction();

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

<Xs> Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagesetsize;
boolean is_setg = FALSE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcv = '0010';

        stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
        assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros(64);

        if SInt(setsize) > 0 then
            assert SInt(stagesetsize) <= SInt(setsize);
        else
            assert SInt(stagesetsize) >= SInt(setsize);
    end

    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    assert postsize<63> == setsize<63> || postsize == Zeros(64);

    boolean zero_size_exceptions = MemSetZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(setsize) != 0 then

if supports_option_a then
  if nzcv<1> == '1' then // PSTATE.C
    boolean wrong_option = TRUE;
    boolean from_epilogue = stage == MOPSStage_Epilogue;
    MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue,
    options, is_setg);
  else
    if nzcv<1> == '0' then // PSTATE.C
      boolean wrong_option = TRUE;
      boolean from_epilogue = stage == MOPSStage_Epilogue;
      MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue,
      options, is_setg);

    if stage == MOPSStage_Main then
      stagesetsize = setsize - postsize;
      if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = FALSE;
        MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options,
        is_setg);
      else
        stagesetsize = postsize;
        if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
          boolean wrong_option = FALSE;
          boolean from_epilogue = TRUE;
          MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options,
          is_setg);
    end if
  end if

if supports_option_a then
  while SInt(stagesetsize) < 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = SETSizeChoice(toaddress, setsize, 1);
    assert B <= -1 * SInt(stagesetsize);

    Mem[toaddress+setsize, B, acctype] = Replicate(data, B);
    setsize = setsize + B;
    stagesetsize = stagesetsize + B;
    if stage != MOPSStage_Prologue then
      X[n, 64] = setsize;
    end if
  end while
else
  while UInt(stagesetsize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = SETSizeChoice(toaddress, setsize, 1);
    assert B <= UInt(stagesetsize);

    Mem[toaddress, B, acctype] = Replicate(data, B);
    toaddress = toaddress + B;
    setsize = setsize - B;
    stagesetsize = stagesetsize - B;
    if stage != MOPSStage_Prologue then
      X[n, 64] = setsize;
      X[d, 64] = toaddress;
    end if
  end while
end if

if stage == MOPSStage_Prologue then
  X[n, 64] = setsize;
  X[d, 64] = toaddress;
  PSTATE.<N,Z,C,V> = nzcv;
C6.2.280 SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see *Wait for Event* on page D1-4676.

<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

**Encoding**

SEV

**Decode for this encoding**

`// Empty.`

**Operation**

`SendEvent();`
C6.2.281   SEVL

Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a WFE instruction.

SendEventLocal();

Encoding
SEVL

Decode for this encoding
// Empty.

Operation
SendEventLocal();
C6.2.282  SMADDL

Signed Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMULL. See Alias conditions for details of when each alias is preferred.

```
unsigned long results = Int(operand3, FALSE) + (Int(operand1, FALSE) * Int(operand2, FALSE));
X[d, 64] = results<63:0>;
```

**Encoding**

SMADDL <Xd>, <Wn>, <Wm>, <Xa>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

**Alias conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMULL</td>
<td>Ra == '1111'</td>
</tr>
</tbody>
</table>

**Assembler symbols**

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

**Operation**

```
bits(32) operand1 = X[n, 32];
bits(32) operand2 = X[m, 32];
bits(64) operand3 = X[a, 64];

integer result;
result = Int(operand3, FALSE) + (Int(operand1, FALSE) * Int(operand2, FALSE));
X[d, 64] = result<63:0>;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.283 SMC

Secure Monitor Call causes an exception to EL3.

SMC is available only for software executing at EL1 or higher. It is UNDEFINED in EL0.

If the values of HCR_EL2.TSC and SCR_EL3.SMD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception, recording it in ESR_ELx, using the EC value 0x17, that is taken to EL3.

If the value of HCR_EL2.TSC is 1 and EL2 is enabled in the current Security state, execution of an SMC instruction at EL1 generates an exception that is taken to EL2, regardless of the value of SCR_EL3.SMD.

If the value of HCR_EL2.TSC is 0 and the value of SCR_EL3.SMD is 1, the SMC instruction is UNDEFINED.

Encoding

SMC #<imm>

Decode for this encoding

// Empty.

Assembler symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

AArch64.CheckForSMCUndefOrTrap(imm16);
AArch64.CallSecureMonitor(imm16);
C6.2.284 SMNEGL

Signed Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This instruction is an alias of the SMSUBL instruction. This means that:

- The encodings in this description are named to match the encodings of SMSUBL.
- The description of SMSUBL gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>SMNEGL &lt;Xd&gt;, &lt;Wn&gt;, &lt;Wm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>is equivalent to</td>
</tr>
<tr>
<td></td>
<td>SMSUBL &lt;Xd&gt;, &lt;Wn&gt;, &lt;Wm&gt;, XZR</td>
</tr>
</tbody>
</table>

and is always the preferred disassembly.

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of SMSUBL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**C6.2.285 SMSTART**

Enables access to Streaming SVE mode and SME architectural state.

SMSTART enters Streaming SVE mode, and enables the SME ZA storage.

SMSTART SM enters Streaming SVE mode, but does not enable the SME ZA storage.

SMSTART ZA enables the SME ZA storage, but does not cause an entry to Streaming SVE mode.

This instruction is an alias of the MSR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of MSR (immediate).
- The description of MSR (immediate) gives the operational pseudocode for this instruction.

**System**

(FEAT_SME)

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 14 13 12 11]</th>
<th>8</th>
<th>7</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

| op1 | CRm | op2 |

**Encoding**

SMSTART {<option>}

is equivalent to

MSR <pstatefield>, #1

and is always the preferred disassembly.

**Assembler symbols**

- `<option>`: Is an optional mode, encoded in the "CRm<2:1>" field. It can have the following values:
  - SM: when CRm<2:1> = 01
  - ZA: when CRm<2:1> = 10
  - [no specifier]: when CRm<2:1> = 11
  - The encoding CRm<2:1> = 00 is reserved.

- `<pstatefield>`: Is a PSTATE field name. For the MSR instruction, this is encoded in the "op1:op2:CRm" field. It can have the following values:
  - SPSel: when op1 = 000, op2 = 101, CRm = xxxx
  - DAIFSet: when op1 = 011, op2 = 110, CRm = xxxx
  - DAIFClr: when op1 = 011, op2 = 111, CRm = xxxx

When FEAT_UAO is implemented, the following value is also valid:

- UAO: when op1 = 000, op2 = 011, CRm = xxxx

When FEAT_PAN is implemented, the following value is also valid:

- PAN: when op1 = 000, op2 = 100, CRm = xxxx

When FEAT_NMI is implemented, the following value is also valid:

- ALLINT: when op1 = 001, op2 = 000, CRm = 000x

When FEAT_SBS is implemented, the following value is also valid:

- SSBS: when op1 = 011, op2 = 001, CRm = xxxx
When FEAT_DIT is implemented, the following value is also valid:
DIT when op1 = 011, op2 = 010, CRm = xxxx

When FEAT_SME is implemented, the following values are also valid:
SVCRSM when op1 = 011, op2 = 011, CRm = 001x
SVCRZA when op1 = 011, op2 = 011, CRm = 010x
SVCRSMZA when op1 = 011, op2 = 011, CRm = 011x

When FEAT_MTE is implemented, the following value is also valid:
TC0 when op1 = 011, op2 = 100, CRm = xxxx

See PSTATE on page C4-539 when op1 = 000, op2 = 00x, CRm = xxxx.
See PSTATE on page C4-539 when op1 = 000, op2 = 010, CRm = xxxx.

The following encodings are reserved:
• op1 = 000, op2 = 11x, CRm = xxxx.
• op1 = 001, op2 = 000, CRm = 001x.
• op1 = 001, op2 = 000, CRm = 01xx.
• op1 = 001, op2 = 000, CRm = 1xxx.
• op1 = 001, op2 = 001, CRm = xxxx.
• op1 = 001, op2 = 01x, CRm = xxxx.
• op1 = 001, op2 = 1xx, CRm = xxxx.
• op1 = 010, op2 = xxx, CRm = xxxx.
• op1 = 011, op2 = 000, CRm = xxxx.
• op1 = 011, op2 = 011, CRm = 000x.
• op1 = 011, op2 = 011, CRm = 1xxx.
• op1 = 011, op2 = 101, CRm = xxxx.
• op1 = 1xx, op2 = xxx, CRm = xxxx.

For the SMSTART and SMSTOP aliases, this is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

**Operation**

The description of MSR (immediate) gives the operational pseudocode for this instruction.
C6.2.286   SMSTOP

Disables access to Streaming SVE mode and SME architectural state.
SMSTOP exits Streaming SVE mode, and disables the SME ZA storage.
SMSTOP SM exits Streaming SVE mode, but does not disable the SME ZA storage.
SMSTOP ZA disables the SME ZA storage, but does not cause an exit from Streaming SVE mode.

This instruction is an alias of the MSR (immediate) instruction. This means that:
• The encodings in this description are named to match the encodings of MSR (immediate).
• The description of MSR (immediate) gives the operational pseudocode for this instruction.

System
(FEAT_SME)

Encoding
SMSTOP {<option>}

is equivalent to
MSR <$pstatefield$>, #0

and is always the preferred disassembly.

Assembler symbols

<option>  Is an optional mode, encoded in the "CRm<2:1>" field. It can have the following values:
    SM     when CRm<2:1> = 01
    ZA     when CRm<2:1> = 10
    [no specifier] when CRm<2:1> = 11
    The encoding CRm<2:1> = 00 is reserved.

<pstatefield>  Is a PSTATE field name. For the MSR instruction, this is encoded in the "op1:op2:CRm" field. It can have the following values:
    SPSel  when op1 = 000, op2 = 101, CRm = xxxx
    DAIFSset when op1 = 011, op2 = 110, CRm = xxxx
    DAIFClr when op1 = 011, op2 = 111, CRm = xxxx
    When FEAT_UAO is implemented, the following value is also valid:
    UAO    when op1 = 000, op2 = 011, CRm = xxxx
    When FEAT_PAN is implemented, the following value is also valid:
    PAN    when op1 = 000, op2 = 100, CRm = xxxx
    When FEAT_NMI is implemented, the following value is also valid:
    ALLINT when op1 = 001, op2 = 000, CRm = 000x
    When FEAT_SSBS is implemented, the following value is also valid:
    SSBS   when op1 = 011, op2 = 001, CRm = xxxx
When FEAT_DIT is implemented, the following value is also valid:

\[ \text{DIT} \quad \text{when } op1 = 011, \ op2 = 010, \ CRm = xxxx \]

When FEAT_SME is implemented, the following values are also valid:

\[ \text{SVCRSM} \quad \text{when } op1 = 011, \ op2 = 011, \ CRm = 001x \]
\[ \text{SVCRZA} \quad \text{when } op1 = 011, \ op2 = 011, \ CRm = 010x \]
\[ \text{SVCRSMZA} \quad \text{when } op1 = 011, \ op2 = 011, \ CRm = 011x \]

When FEAT_MTE is implemented, the following value is also valid:

\[ \text{TC0} \quad \text{when } op1 = 011, \ op2 = 100, \ CRm = xxxx \]

See \textit{PSTATE} on page C4-539 when \( op1 = 000, \ op2 = 00x, \ CRm = xxxx \).
See \textit{PSTATE} on page C4-539 when \( op1 = 000, \ op2 = 010, \ CRm = xxxx \).

The following encodings are reserved:

- \( op1 = 000, \ op2 = 11x, \ CRm = xxxx \).
- \( op1 = 001, \ op2 = 000, \ CRm = 001x \).
- \( op1 = 001, \ op2 = 000, \ CRm = 01xx \).
- \( op1 = 001, \ op2 = 000, \ CRm = 1xxx \).
- \( op1 = 001, \ op2 = 001, \ CRm = xxxx \).
- \( op1 = 001, \ op2 = 01x, \ CRm = xxxx \).
- \( op1 = 001, \ op2 = 1xx, \ CRm = xxxx \).
- \( op1 = 010, \ op2 = xxx, \ CRm = xxxx \).
- \( op1 = 011, \ op2 = 000, \ CRm = xxxx \).
- \( op1 = 011, \ op2 = 001, \ CRm = 000x \).
- \( op1 = 011, \ op2 = 011, \ CRm = 1xxx \).
- \( op1 = 011, \ op2 = 101, \ CRm = xxxx \).
- \( op1 = 1xx, \ op2 = xxx, \ CRm = xxxx \).

For the SMSTART and SMSTOP aliases, this is encoded in "CRm<2:1>"", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

\section*{Operation}

The description of \textit{MSR (immediate)} gives the operational pseudocode for this instruction.
C6.2.287 SMSUBL

Signed Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMNEGL. See Alias conditions for details of when each alias is preferred.

Encoding

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xd</td>
<td>64-bit name of the general-purpose destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>Wn</td>
<td>32-bit name of the first general-purpose source register holding the multiplicand, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>Wm</td>
<td>32-bit name of the second general-purpose source register holding the multiplier, encoded in the &quot;Rm&quot; field.</td>
</tr>
<tr>
<td>Xa</td>
<td>64-bit name of the third general-purpose source register holding the minuend, encoded in the &quot;Ra&quot; field.</td>
</tr>
</tbody>
</table>

Operation

```
bits(32) operand1 = X[n, 32];
bits(32) operand2 = X[m, 32];
bits(64) operand3 = X[a, 64];

integer result;
result = Int(operand3, FALSE) - (Int(operand1, FALSE) * Int(operand2, FALSE));
X[d, 64] = result<63:0>;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.288 SMULH

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

Encoding

SMULH <Xd>, <Xn>, <Xm>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

bits(64) operand1 = X[n, 64];
bits(64) operand2 = X[m, 64];

integer result;
result = Int(operand1, FALSE) * Int(operand2, FALSE);
X[d, 64] = result<127:64>;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.289   SMULL

Signed Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This instruction is an alias of the SMADDL instruction. This means that:

- The encodings in this description are named to match the encodings of SMADDL.
- The description of SMADDL gives the operational pseudocode for this instruction.

```
[31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 | 10 9 | 5 4 | 0 |
  1 0 0 | 1 1 0 1 0 | 0 1 | Rm | 0 | 1 1 1 | Rn | Rd |

Encoding
SMULL <Xd>, <Wn>, <Wm>

is equivalent to
SMADDL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler symbols

<Xd>     Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>     Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>     Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of SMADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.290  SSBB

Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the SSBB.

- When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order after the SSBB.

This instruction is an alias of the DSB instruction. This means that:

- The encodings in this description are named to match the encodings of DSB.
- The description of DSB gives the operational pseudocode for this instruction.

### Encoding

SSBB

is equivalent to

DSB #0

and is always the preferred disassembly.

### Operation

The description of DSB gives the operational pseudocode for this instruction.
C6.2.291   ST2G

Store Allocation Tags stores an Allocation Tag to two Tag granules of memory. The address used for the store is
calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is
calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

Post-index

(Feat_MTE)

```
[31 30 29 28 | 27 26 25 24 | 23 22 21 20] | 12 | 11 10 9 | 5 4 | 0 |
1 1 0 1 1 0 0 1 | 1 0 1 | imm9 | 0 1 | Xn | Xt
```

Encoding

ST2G <Xt|SP>, [<Xn|SP>], #<imm>

Decode for this encoding

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;

Pre-index

(Feat_MTE)

```
[31 30 29 28 | 27 26 25 24 | 23 22 21 20] | 12 | 11 10 9 | 5 4 | 0 |
1 1 0 1 1 0 0 1 | 1 0 1 | imm9 | 1 1 | Xn | Xt
```

Encoding

ST2G <Xt|SP>, [Xn|SP>, #<imm>]

Decode for this encoding

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;

Signed offset

(Feat_MTE)

```
[31 30 29 28 | 27 26 25 24 | 23 22 21 20] | 12 | 11 10 9 | 5 4 | 0 |
1 1 0 1 1 0 0 1 | 1 0 1 | imm9 | 1 0 | Xn | Xt
```
Encoding

ST2G <Xt|SP>, [<Xn|SP>{, #<simm>}]

Decode for this encoding

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

Assembler symbols

<Xt|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Xt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation for all encodings

bits(64) address;
bits(64) data = if t == 31 then SP[] else X[t, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
SetTagCheckedInstruction(FALSE);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
if !postindex then
    address = address + offset;
AArch64.MemTag[address, AccType_NORMAL] = tag;
AArch64.MemTag[address+TAG_GRANULE, AccType_NORMAL] = tag;
if writeback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
C6.2.292   ST64B

Single-copy Atomic 64-byte Store without Return stores eight 64-bit doublewords from consecutive registers, \(X_t\) to \(X(t+7)\), to a memory location. The data that is stored is atomic and is required to be 64-byte-aligned.

**Integer**

(Feat_LS64)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(R_t) (R_n)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

ST64B \(<X_t>, [<X_n|SP> \{,#0}\]\

**Decode for this encoding**

if !HaveFeatLS64() then UNDEFINED;
if \(R_t<4:3> == '11' || R_t<0> == '1'\) then UNDEFINED;

\(\text{integer } n = \text{UInt}(R_n);\)
\(\text{integer } t = \text{UInt}(R_t);\)
\(\text{boolean } \text{tag\_checked} = n != 31;\)

**Assembler symbols**

\(<X_t>\) Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
\(<X_n|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

CheckLDST64BEnabled();

\(\text{bits(512) } \text{data};\)
\(\text{bits(64) } \text{address};\)
\(\text{bits(64) } \text{value};\)
\(\text{acctype} = \text{AccType\_ATOMICLS64};\)

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag\_checked);

for \(i = 0\) to \(7\)
  \(\text{value} = \text{X}[t+i, 64];\)
  if BigEndian(acctype) then \(\text{value} = \text{BigEndianReverse(value)};\)
  \(\text{data}<63+64*i:64*i> = \text{value};\)

if \(n == 31\) then
    CheckSPAlignment();
    \(\text{address} = \text{SP}[];\)
else
    \(\text{address} = \text{X}[n, 64];\)
\(\text{MemStore64B(address, data, acctype)};\)
C6.2.293 ST64BV

Single-copy Atomic 64-byte Store with Return stores eight 64-bit doublewords from consecutive registers, \(X_t\) to \(X(t+7)\), to a memory location, and writes the status result of the store to a register. The data that is stored is atomic and is required to be 64-byte aligned.

**Integer**

(FEAT_LS64_V)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 0 0 0 1</td>
<td>1 0 1 1 0 0</td>
</tr>
</tbody>
</table>

**Encoding**

ST64BV <Xs>, <Xt>, [<Xn|SP>]

**Decode for this encoding**

if !HaveFeatLS64_V() then UNDEFINED;
if \(Rt<4:3> == '11'\) \(\lor\) \(Rt<0> == '1'\) then UNDEFINED;

integer \(n = \) UInt(Rn);
integer \(t = \) UInt(Rt);
integer \(s = \) UInt(Rs);
boolean tag_checked = \(n \neq 31\);

**Assembler symbols**

<Xs> Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the "Rs" field.

The value returned is:

0xFFFFFFFF_FFFFFFFF if the memory location accessed does not support this instruction. In this case, the value at the memory location is UNKNOWN.

\(!= 0xFFFFFFFF_FFFFFFFF\) if the memory location accessed does support this instruction. In this case, the peripheral that provides the response defines the returned value and provides information on the state of the memory update at the memory location.

If XZR is used, then the return value is ignored.

<Xt> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

CheckST64BVEnabled();

bits(64) data;
bits(64) address;
bits(64) value;
bits(64) status;
acctype = AccType_ATOMICLS64;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

for \(i = 0\) to 7
    value = \(X[t+i, 64]\);
if BigEndian(acctype) then value = BigEndianReverse(value);
data<63:64*i:64*i> = value;

if n == 31 then
  CheckSPA\text{Alignment}();
  address = SP[];
else
  address = X[n, 64];

status = MemStore64BWithRet(address, data, acctype);
if s != 31 then X[s, 64] = status;
C6.2.294   ST64BV0

Single-copy Atomic 64-byte EL0 Store with Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location, with the bottom 32 bits taken from ACCDATA_EL1, and writes the status result of the store to a register. The data that is stored is atomic and is required to be 64-byte aligned.

**Integer**

(FEAT_LS64_ACCDATA)

```
01 11 11 10 00 00 00 01 | Rs 10 01 00 00 | Rn | Rt
```

**Encoding**

ST64BV0 <Xs>, <Xt>, [<Xn|SP>]

**Decode for this encoding**

if !HaveFeatLS64_ACCDATA() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
boolean tag_checked = n != 31;

**Assembler symbols**

<Xs> Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the "Rs" field.

The value returned is:

- 0xFFFFFFFF_FFFFFFFF if the memory location accessed does not support this instruction. In this case, the value at the memory location is UNKNOWN.
- != 0xFFFFFFFF_FFFFFFFF if the memory location accessed does support this instruction. In this case, the peripheral that provides the response defines the returned value and provides information on the state of the memory update at the memory location.

If XZR is used, then the return value is ignored.

<Xt> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

CheckST64BV0Enabled();

bits(64) data;
bias(64) address;
bias(64) value;
bias(64) status;
acctype = AccType_ATOMICLS64;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

bits(64) Xt = X[t, 64];
value<31:0> = ACCDATA_EL1<31:0>;
value<63:32> = Xt<63:32>;
if BigEndian(acctype) then value = BigEndianReverse(value);
data<63:0> = value;
for i = 1 to 7
  value = X[t+i, 64];
  if BigEndian(acctype) then value = BigEndianReverse(value);
data<63+64*i:64*i> = value;
if n == 31 then
  CheckSPAAlignment();
  address = SP[];
else
  address = X[n, 64];
status = MemStore64BWithRet(address, data, acctype);
if s != 31 then X[s, 64] = status;
C6.2.295   STADDB, STADDLB

Atomic add on byte in memory, without return, atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDB does not have release semantics.
- STADDLB stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDADDB, LDADDAB, LDADDALB, LDADDLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDADDB, LDADDAB, LDADDALB, LDADDLB.
- The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

### Integer

(FEAT_LSE)

| 31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14|12|11 10 9 | 5 4 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| 0 0 1 1 | 0 0 0 0 | R 1 | Rs | 0 0 0 0 | 0 0 | Rn | 1 1 1 1 |

#### No memory ordering variant

Applies when \( R == 0 \).

STADDB \(<Ws>, [<Xn|SP>]\)

is equivalent to

LDADDB \(<Ws>, WZR, [<Xn|SP>]\)

and is always the preferred disassembly.

#### Release variant

Applies when \( R == 1 \).

STADDLB \(<Ws>, [<Xn|SP>]\)

is equivalent to

LDADDLB \(<Ws>, WZR, [<Xn|SP>]\)

and is always the preferred disassembly.

### Assembler symbols

- \(<Ws>\) Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- \(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.296   STADDH, STADDLH

Atomic add on halfword in memory, without return, atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDH does not have release semantics.
- STADDLH stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDADDH, LDADDAH, LDADDALH, LDADDLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDADDH, LDADDAH, LDADDALH, LDADDLH.
- The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

**Integer**

(FEAT_LSE)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 0 0 0 0</td>
<td>R</td>
<td>Rs 0 0 0 0 0</td>
<td>Rn 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**No memory ordering variant**

Applies when \( R == 0 \).

STADDH \(<Ws>, [<Xn]|SP>\) is equivalent to

LDADDH \(<Ws>, WZR, [<Xn]|SP>\)

and is always the preferred disassembly.

**Release variant**

Applies when \( R == 1 \).

STADDLH \(<Ws>, [<Xn]|SP>\) is equivalent to

LDADDLH \(<Ws>, WZR, [<Xn]|SP>\)

and is always the preferred disassembly.

**Assembler symbols**

\(<Ws>\) Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.297 **STADD, STADDL**

Atomic add on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

- **STADD** does not have release semantics.
- **STADDL** stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-180.

For information about memory accesses see *Load/store addressing modes* on page C1-234.

This instruction is an alias of the **LDADD, LDADDA, LDADDAL, LDADDL** instruction. This means that:

- The encodings in this description are named to match the encodings of **LDADD, LDADDA, LDADDAL, LDADDL**.
- The description of **LDADD, LDADDA, LDADDAL, LDADDL** gives the operational pseudocode for this instruction.

### Integer

**(FEAT_LSE)**

```
size A opc Rt
31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14| 12|11 10 9| 5 4 | 0 |
1 x 1 1 1 0 0 0 0 R 1
Rs 0 0 0 0 0 0 Rn
1 1 1 1 1
```

**32-bit LDADD alias variant**

Applies when `size == 10 && R == 0`.

**STADD** `<Ws>`, `[<Xn|SP>]`

is equivalent to

**LDADD** `<Ws>`, WZR, `[<Xn|SP>]`

and is always the preferred disassembly.

**32-bit LDADDL alias variant**

Applies when `size == 10 && R == 1`.

**STADDL** `<Ws>`, `[<Xn|SP>]`

is equivalent to

**LDADDL** `<Ws>`, WZR, `[<Xn|SP>]`

and is always the preferred disassembly.

**64-bit LDADD alias variant**

Applies when `size == 11 && R == 0`.

**STADD** `<Xs>`, `[<Xn|SP>]`

is equivalent to

**LDADD** `<Xs>`, XZR, `[<Xn|SP>]`

and is always the preferred disassembly.
64-bit LDADDL alias variant

Applies when size == 11 & R == 1.

STADDL <Xs>, [<Xn|SP>]

is equivalent to

LOADDL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.298   STCLRB, STCLRLB

Atomic bit clear on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRB does not have release semantics.

For information about memory accesses see *Load/store addressing modes* on page C1-234.

This instruction is an alias of the LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB.
- The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode for this instruction.

**Integer**

(FEAT_LSE)

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[16 15 14]</th>
<th>[12 11 10 9]</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Rn</td>
<td>Rs</td>
<td>opc</td>
<td>size</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**No memory ordering variant**

Applies when \( R == 0 \).

STCLRB \(<\text{Ws}>, [<Xn|SP>]\)

is equivalent to

LDCLRB \(<\text{Ws}>, \text{WZR}, [<Xn|SP>]\)

and is always the preferred disassembly.

**Release variant**

Applies when \( R == 1 \).

STCLRLB \(<\text{Ws}>, [<Xn|SP>]\)

is equivalent to

LDCLRLB \(<\text{Ws}>, \text{WZR}, [<Xn|SP>]\)

and is always the preferred disassembly.

**Assembler symbols**

- \(<\text{Ws}>\) Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- \(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.299  STCLRH, STCLRLH

Atomic bit clear on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRH does not have release semantics.
- STCLRLH stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH.
- The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode for this instruction.

**Integer**

(FEAT_LSE)

No memory ordering variant

Applies when R == 0.

STCLRH <Ws>, [<Xn|SP>]

is equivalent to

LDCLRH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release variant

Applies when R == 1.

STCLRLH <Ws>, [<Xn|SP>]

is equivalent to

LDCLRLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Assembler symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.300  STCLR, STCLRL

Atomic bit clear on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLR does not have release semantics.
- STCLRL stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDCLR, LDCLRA, LDCLRAL, LDCLRL instruction. This means that:

- The encodings in this description are named to match the encodings of LDCLR, LDCLRA, LDCLRAL, LDCLRL.
- The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

### Integer

(FEAT_LSE)

![Instruction Format](image)

#### 32-bit LDCLR alias variant

Applies when size == 10 && R == 0.

STCLR <Ws>, [<Xn|SP>]

is equivalent to

LDCLR <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 32-bit LDCLRL alias variant

Applies when size == 10 && R == 1.

STCLRL <Ws>, [<Xn|SP>]

is equivalent to

LDCLRL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 64-bit LDCLR alias variant

Applies when size == 11 && R == 0.

STCLR <Xs>, [<Xn|SP>]

is equivalent to

LDCLR <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
64-bit LDCLRL alias variant

Applies when size == 11 && R == 1.

STCLRL <Xs>, [<Xn|SP>]

is equivalent to

LDCLRL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler symbols

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.301 STEORB, STEORLB

Atomic exclusive OR on byte in memory, without return, atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEORB does not have release semantics.
- STEORLB stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDEORB, LDEORAB, LDEORALB, LDEORLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDEORB, LDEORAB, LDEORALB, LDEORLB.
- The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

**Integer**

(FEAT_LSE)

| [31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 | 12|11 10 9 | 5 4 | 0 | 0 0 1 1 1 0 0 0 0 R 1 | Rs 0 0 1 0 0 0 | Rn 1 1 1 1 1 | 0 | size | A | opc | Rt |

**No memory ordering variant**

Applies when \( R == 0 \).

STEORB \(<Ws>, \,[<Xn]\{SP\}>\)

is equivalent to

LDEORB \(<Ws>, \, WZR, \,[<Xn]\{SP\}>\)

and is always the preferred disassembly.

**Release variant**

Applies when \( R == 1 \).

STEORLB \(<Ws>, \,[<Xn]\{SP\}>\)

is equivalent to

LDEORLB \(<Ws>, \, WZR, \,[<Xn]\{SP\}>\)

and is always the preferred disassembly.

**Assembler symbols**

\(<Ws>\) Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.302 STEORH, STEORLH

Atomic exclusive OR on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEORH does not have release semantics.
- STEORLH stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDEORH, LDEORAH, LDEORALH, LDEORLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDEORH, LDEORAH, LDEORALH, LDEORLH.
- The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

**Integer**

(FEAT_LSE)

![Instruction encoding](image)

**No memory ordering variant**

Applies when \( R = 0 \).

```text
STEORH <Ws>, [<Xn|SP>] is equivalent to LDEORH <Ws>, WZR, [<Xn|SP>] and is always the preferred disassembly.
```

**Release variant**

Applies when \( R = 1 \).

```text
STEORLH <Ws>, [<Xn|SP>] is equivalent to LDEORLH <Ws>, WZR, [<Xn|SP>] and is always the preferred disassembly.
```

**Assembler symbols**

- `<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.303 STEOR, STEORL

Atomic exclusive OR on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEOR does not have release semantics.
- STEORL stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDEOR, LDEORA, LDEORAL, LDEORL instruction. This means that:

- The encodings in this description are named to match the encodings of LDEOR, LDEORA, LDEORAL, LDEORL.
- The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

**Integer**

(FEAT_LSE)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x 1 1 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
</tr>
<tr>
<td>size</td>
<td>A</td>
<td>opc</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit LDEOR alias variant**

Applies when size == 10 && R == 0.

STEOR <Ws>, [<Xn|SP>]

is equivalent to

LDEOR <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**32-bit LDEORL alias variant**

Applies when size == 10 && R == 1.

STEORL <Ws>, [<Xn|SP>]

is equivalent to

LDEORL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**64-bit LDEOR alias variant**

Applies when size == 11 && R == 0.

STEOR <Xs>, [<Xn|SP>]

is equivalent to

LDEOR <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
64-bit LDEORL alias variant

Applies when size == 11 && R == 1.

STEORL <Xs>, [<Xn|SP>]

is equivalent to

LDEORL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.304  STG

Store Allocation Tag stores an Allocation Tag to memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

Post-index

(FEAT_MTE)

\[
\begin{array}{cccccccc}
 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & \text{imm9} & 0 & 1 & \text{Xn} & \text{Xt} \\
\end{array}
\]

Encoding

STG \text{<Xt|SP>, [<Xn|SP>], #<simm>}

Decode for this encoding

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;

Pre-index

(FEAT_MTE)

\[
\begin{array}{cccccccc}
 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & \text{imm9} & 1 & 1 & \text{Xn} & \text{Xt} \\
\end{array}
\]

Encoding

STG \text{<Xt|SP>, [<Xn|SP>], #<simm>!}

Decode for this encoding

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;

Signed offset

(FEAT_MTE)

\[
\begin{array}{cccccccc}
 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & \text{imm9} & 1 & 0 & \text{Xn} & \text{Xt} \\
\end{array}
\]
**Encoding**

STG <Xt|SP>, [<Xn|SP>{{, #<simm>}}</simm>]

**Decode for this encoding**

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

**Assembler symbols**

<Xt|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Xt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

<simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

**Operation for all encodings**

bits(64) address;
SetTagCheckedInstruction(FALSE);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];
if !postindex then
  address = address + offset;
bits(64) data = if t == 31 then SP[] else X[t, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, AccType_NORMAL] = tag;
if writeback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n, 64] = address;
C6.2.305  STGM

Store Tag Multiple writes a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID_EL1.BS, and the Allocation Tag written to address A is taken from the source register at 4*A<7:4>+3:4*A<7:4>.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

Integer

(FEAT_MTE2)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 0 1 1</td>
<td>0 1 0 0 0 0 0 0</td>
<td>Xn</td>
<td>Xt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

STGM <Xt>, [<Xn|SP>]

Decode for this encoding

if !HaveMTE2Ext() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);

Assembler symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Xt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

Operation

if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = X[t, 64];
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

integer size = 4 * (2 ^ (UInt(GMID_EL1.BS)));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);

for i = 0 to count-1
    bits(4) tag = data<index*4+3:index*4>;
    AArch64.MemTag[address, AccType_NORMAL] = tag;
    address = address + TAG_GRANULE;
    index = index + 1;
C6.2.306   STGP

Store Allocation Tag and Pair of registers stores an Allocation Tag and two 64-bit doublewords to memory, from two registers. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the base register.

This instruction generates an Unchecked access.

**Post-index**  
(FEAT_MTE)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 1 0 0 0 1 0</td>
<td>simm7</td>
<td>Xt2</td>
<td>Xn</td>
<td>Xt</td>
</tr>
</tbody>
</table>

**Encoding**

STGP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>

**Decode for this encoding**

!HaveMTEExt() then UNDEFINED;  
integer n = UInt(Xn);  
integer t = UInt(Xt);  
integer t2 = UInt(Xt2);  
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);  
boolean writeback = TRUE;  
boolean postindex = TRUE;

**Pre-index**  
(FEAT_MTE)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 1 0 0 1 0</td>
<td>simm7</td>
<td>Xt2</td>
<td>Xn</td>
<td>Xt</td>
</tr>
</tbody>
</table>

**Encoding**

STGP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!

**Decode for this encoding**

!HaveMTEExt() then UNDEFINED;  
integer n = UInt(Xn);  
integer t = UInt(Xt);  
integer t2 = UInt(Xt2);  
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);  
boolean writeback = TRUE;  
boolean postindex = FALSE;

**Signed offset**  
( FEAT_MTE )
Encoding

STGP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]}

Decode for this encoding

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

Assembler symbols

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Xt" field.
<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Xt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<imm> For the post-index and pre-index variant: is the signed immediate offset, a multiple of 16 in the range -1024 to 1008, encoded in the "simm7" field.
For the signed offset variant: is the optional signed immediate offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "simm7" field.

Operation for all encodings

bits(64) address;
bits(64) data1;
bits(64) data2;
SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[);
else
    address = X[n, 64];
data1 = X[t, 64];
data2 = X[t2, 64];

if !postindex then
    address = address + offset;

if address != Align(address, TAG_GRANULE) then
    AArch64.Abort(address, AlignmentFault(AccType_NORMAL, TRUE, FALSE));

Mem[address, 8, AccType_NORMAL] = data1;
Mem[address+8, 8, AccType_NORMAL] = data2;

AArch64.MemTag[address, AccType_NORMAL] = AArch64.AllocationTagFromAddress(address);

if writeback then
    if postindex then
address = address + offset;
if n == 31 then
    SP[] = address;
else
    X[n, 64] = address;
C6.2.307  STLLRB

Store LORelease Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in LoadLOAcquire, StoreLORelease on page B2-181. For information about memory accesses, see Load/store addressing modes on page C1-234.

No offset

(FEAT_LOR)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0 1 0 0 0 0</td>
<td>0 0 0 0 1 0 0 1 0 0 0 0</td>
<td>0 0 0 0 1 0 0 1 0 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

STLLRB <Wt>, [<Xn|SP>{,#0}]

Decode for this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAAlignment();
    address = SP[];
else
    address = X[n, 64];

data = X[t, 8];
Mem[address, 1, AccType_LIMITEDORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.308  STLLRH

Store LORelease Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in LoadLOAcquire, StoreLORelease on page B2-181. For information about memory accesses, see Load/store addressing modes on page C1-234.

No offset

(FEAT_LOR)

| 31 30 29 28|27 26 25 24|23 22 21 20| 16 |15 14 | 10 9 | 5 4 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | (1) | (1) | (1) | (1) | 0 | (1) | (1) | (1) | (1) |
| size | L | Rs | o0 | Rt |

Encoding

STLLRH <Wt>, [<Xn|SP>{,#0}]

Decode for this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = X[t, 16];
Mem[address, 2, AccType_LIMITEDORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.309  STLLR

Store LORelease Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in LoadLOAcquire, StoreLORelease on page B2-181. For information about memory accesses, see Load/store addressing modes on page C1-234.

**No offset**

(FEAT_LOR)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>size</td>
<td>L</td>
<td>Rs</td>
<td>o0</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit variant

Applies when size == 10.

STLLR <Wt>, [<Xn|SP>{,#0}]

64-bit variant

Applies when size == 11.

STLLR <Xt>, [<Xn|SP>{,#0}]

**Decode for all variants of this encoding**

```{}
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;
```

**Assembler symbols**

- `<Wt>` is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

```{}
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = X[t, elsize];
Mem[address, dbytes, AccType_LIMITEDORDERED] = data;
```
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.310   STLR

Store-Release Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180. For information about memory accesses, see Load/store addressing modes on page C1-234.

32-bit variant
Applies when size == 10.
STLR <Wt>, [<Xn|SP>{,#0}]

64-bit variant
Applies when size == 11.
STLR <Xt>, [<Xn|SP>{,#0}]

Decode for all variants of this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);
integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
data = X[t, elsize];
Mem[address, dbytes, AccType.ORDERED] = data;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.311   STLRB

Store-Release Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180. For information about memory accesses, see Load/store addressing modes on page C1-234.

Encoding

STLRB <Wt>, [<Xn|SP>{,#0}]

Decode for this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];
data = X[t, 8];
Mem[address, 1, AccType_ORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.312 STLRH

Store-Release Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180. For information about memory accesses, see Load/store addressing modes on page C1-234.

Encoding

STLRH <Wt>, [<Xn|SP>{,#0}]

Decode for this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[);
else
    address = X[n, 64];
data = X[t, 16];
Mem[address, 2, AccType_ORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.313   STLUR

Store-Release Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180*

For information about memory accesses, see *Load/store addressing modes on page C1-234.*

**Unscaled offset**

(FEAT_LRCPC2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
<td>imm9</td>
<td>0</td>
</tr>
</tbody>
</table>

**32-bit variant**

Applies when size == 10.

STLUR <Wt>, [<Xn|SP>{, #<simm>}]

**64-bit variant**

Applies when size == 11.

STLUR <Xt>, [<Xn|SP>{, #<simm>}]

**Decode for all variants of this encoding**

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

**Assembler symbols**

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared decode for all encodings**

integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
boolean tag_checked = n != 31;

**Operation**

bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
   SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

address = address + offset;

data = X[t, datasize];
Mem[address, datasize DIV 8, AccType.ORDERED] = data;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.314 STLURB

Store-Release Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180

For information about memory accesses, see Load/store addressing modes on page C1-234.

Unscaled offset

(FEAT_LRCPC2)

| [31 30 29 28|27 26 25 24|23 22 21 20] | 12|11 10 9 | 5 4 | 0 |
|------------------|---------------|-----|-----|-----|
| 0 0 1 1 0 0 1 0 0 0 | imm9 | 0 0 | Rn | Rt |

Encoding

STLURB <Wt>, [<Xn|SP>{, #<simm}>]

Decode for this encoding

bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];
address = address + offset;
data = X[t, 8];
Mem[address, 1, AccType.ORDERED] = data;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.315   STLURH

Store-Release Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180

For information about memory accesses, see Load/store addressing modes on page C1-234.

Unscaled offset

(FEATURE_LRCPC2)

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

STLURH <Wt>, [<Xn|SP>{, #<simm}>]

Decode for this encoding

bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Operation

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;
data = X[t, 16];
Mem[address, 2, AccType_ORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.316 STLXP

Store-Release Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores on page B2-211. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity on page B2-154 and Alignment of data accesses on page B2-189. If a 64-bit pair Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180. For information about memory accesses, see Load/store addressing modes on page C1-234.

32-bit variant

Applies when sz == 0.

```
STLXP <W>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when sz == 1.

```
STLXP <W>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);  // ignored by load/store single register
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;  // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn_unknown = TRUE;  // address is UNKNOWN
            when Constraint_UNDEF   UNDEFINED;
            when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly STLXP on page K1-11596.
Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0 If the operation updates memory.
1 If the operation fails to update memory.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:
• Memory is not updated.
• <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:
• If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
• Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag Checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
elif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n, 64];
if rt_unknown then
  data = bits(datasize) UNKNOWN;
else
  bits(datasize DIV 2) el1 = X[t, datasize DIV 2];
  bits(datasize DIV 2) el2 = X[t2, datasize DIV 2];
  data = if BigEndian(AccType_ORDEREDATOMIC) then el1:el2 else el2:el1;
  bit status = '1';
  // Check whether the Exclusives monitors are set to include the
  // physical memory locations corresponding to virtual address
  // range [address, address+dbytes-1].
  if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
// to the same physical locations after address translation.
Mem[address, dbytes, AccType.ORDERED_ATOMIC] = data;
status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.317  **STLXR**

Store-Release Exclusive Register stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* on page B2-211. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-180. For information about memory accesses see *Load/store addressing modes* on page C1-234.

<table>
<thead>
<tr>
<th>32-bit variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applies when size == 10.</td>
</tr>
<tr>
<td>STLXR &lt;Ws&gt;, &lt;Wt&gt;, [&lt;Xn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64-bit variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applies when size == 11.</td>
</tr>
<tr>
<td>STLXR &lt;Ws&gt;, &lt;Xt&gt;, [&lt;Xn</td>
</tr>
</tbody>
</table>

### Decode for all variants of this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs); // ignored by all loads and store-release

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
  Constraint c = ConstrainingUnpredictable(Unpredictable_DATAOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
case c of
  when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
  when Constraint_UNDEF  UNDEFINED;
  when Constraint_NOP   EndOfInstruction();
if s == n && n != 31 then
  Constraint c = ConstrainingUnpredictable(Unpredictable_BASEOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
case c of
  when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
  when Constraint_UNDEF  UNDEFINED;
  when Constraint_NOP   EndOfInstruction();

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly STLXR on page K1-11596.
Assembler symbols

<\(s\)> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
- 0 If the operation updates memory.
- 1 If the operation fails to update memory.

<\(t\)> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<\(w\)> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<\(n|SP\)> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:
- Memory is not updated.
- <\(s\)> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:
- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

\[
\begin{align*}
\text{bits(64)} & \text{ address;} \\
\text{bits(elsize)} & \text{ data;} \\
\text{constant integer } d\text{bytes }= \frac{\text{elsize}}{8}; \\
\text{if HaveMTE2Ext()} & \text{ then} \\
\text{ SetTagCheckedInstruction}(\text{tag\_checked}); \\
\text{if } n & = 31 \text{ then} \\
\text{ CheckSPAlignment();} \\
\text{ address } & = \text{SP}[]; \\
\text{elsif } \text{rn\_unknown then} \\
\text{ address } & = \text{bits(64) UNKNOWN;} \\
\text{else} \\
\text{ address } & = \text{X}[n, 64]; \\
\text{if } \text{rt\_unknown then} \\
\text{ data } & = \text{bits(elsize) UNKNOWN;} \\
\text{else} \\
\text{ data } & = \text{X}[t, \text{elsize}]; \\
\text{bit status } & = '1'; \\
\text{// Check whether the Exclusives monitors are set to include the} \\
\text{// physical memory locations corresponding to virtual address} \\
\text{// range [address, address+dbytes-1].} \\
\text{if AArch64.ExclusiveMonitorsPass(address, dbytes) then} \\
\text{// This atomic write will be rejected if it does not refer} \\
\text{// to the same physical locations after address translation.} \\
\text{Mem}[address, dbytes, AccType\_ORDEREDATOMIC] = data;} \\
\text{status } & = \text{ExclusiveMonitorsStatus();} \\
\text{X}[s, 32] & = \text{ZeroExtend(status, 32);} \\
\end{align*}
\]
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.318 STLXRB

Store-Release Exclusive Register Byte stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores on page B2-211. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180. For information about memory accesses see Load/store addressing modes on page C1-234.

Encoding

STLXRB \(<Ws>\), \(<Wt>\), \([<Xn>|{SP}>\{,#0}\] \)

Decode for this encoding

\[
\begin{align*}
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } t &= \text{UInt}(Rt); \\
\text{integer } s &= \text{UInt}(Rs); & \quad \text{// ignored by all loads and store-release}
\end{align*}
\]

\[
\begin{align*}
\text{boolean tag_checked} &= n \neq 31; \\
\text{boolean rt\_unknown} &= \text{FALSE}; \\
\text{boolean rn\_unknown} &= \text{FALSE}; \\
\end{align*}
\]

\[
\begin{align*}
\text{if } s &= t \text{ then} \\
\text{Constraint } c &= \text{ConstrainUnpredictable(Unpredictable\_DATAOVERLAP);} \\
\text{assert } c \text{ IN \{Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP\};} \\
\text{case } c \text{ of} \\
\text{when } \text{Constraint\_UNKNOWN} & \text{ rt\_unknown = TRUE;} & \quad \text{// store UNKNOWN value} \\
\text{when } \text{Constraint\_UNDEF} & \text{ UNDEFINED;} \\
\text{when } \text{Constraint\_NOP} & \text{ EndOfInstruction();}
\end{align*}
\]

\[
\begin{align*}
\text{if } s &= n \&\& n \neq 31 \text{ then} \\
\text{Constraint } c &= \text{ConstrainUnpredictable(Unpredictable\_BASEOVERLAP);} \\
\text{assert } c \text{ IN \{Constraint\_UNKNOWN, Constraint\_UNDEF, Constraint\_NOP\};} \\
\text{case } c \text{ of} \\
\text{when } \text{Constraint\_UNKNOWN} & \text{ rn\_unknown = TRUE;} & \quad \text{// address is UNKNOWN} \\
\text{when } \text{Constraint\_UNDEF} & \text{ UNDEFINED;} \\
\text{when } \text{Constraint\_NOP} & \text{ EndOfInstruction();}
\end{align*}
\]

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly STLXRB on page K1-11597.

Assembler symbols

\(<Ws>\) Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

\[
\begin{align*}
0 & \quad \text{If the operation updates memory.} \\
1 & \quad \text{If the operation fails to update memory.}
\end{align*}
\]

\(<Wt>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn>|{SP}>\{,#0\] Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts
If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

**Operation**

```c
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[ ];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
    X[s, 32] = ZeroExtend(status, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.319   STLXRH

Store-Release Exclusive Register Halfword stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores on page B2-211. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180. For information about memory accesses see Load/store addressing modes on page C1-234.

Encoding

STLXRH <Ws>, <Wt>, [<Xn|SP>{,#0}]

Decode for this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs); // ignored by all loads and store-release

boolean tag_checked = n != 31;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    endcase;
else if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    endcase;
endif;

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly STLXRH on page K1-11597.

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment
If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- \(<w>\) is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

**Operation**

```plaintext
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];

bit status = '1';
// Check whether the Exclusives monitors are set to include the // physical memory locations corresponding to virtual address // range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer // to the same physical locations after address translation.
    Mem[address, 2, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.320  STNP

Store Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see Load/store addressing modes on page C1-234. For information about Non-temporal pair instructions, see Load/store non-temporal pair on page C3-259.

32-bit variant

Applies when opc == 00.

STNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]  

64-bit variant

Applies when opc == 10.

STNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]  

Decode for all variants of this encoding
  // Empty.

Assembler symbols

<Wt1>  Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2>  Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1>  Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2>  Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>  For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;
Operation

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMT2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data1 = X[t, datasize];
data2 = X[t2, datasize];
Mem[address, dbytes, AccType_STREAM] = data1;
Mem[address+dbytes, dbytes, AccType_STREAM] = data2;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.321   STP

Store Pair of Registers calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/store addressing modes* on page C1-234.

**Post-index**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th></th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>imm7</td>
</tr>
<tr>
<td>L</td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit variant**

Applies when opc == 00.

STP \(<Wt1>, <Wt2>, [<Xn|SP>], \#<imm>\)

**64-bit variant**

Applies when opc == 10.

STP \(<Xt1>, <Xt2>, [<Xn|SP>], \#<imm>\)

*Decode for all variants of this encoding*

boolean wback = TRUE;
boolean postindex = TRUE;

**Pre-index**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th></th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>imm7</td>
</tr>
<tr>
<td>L</td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit variant**

Applies when opc == 00.

STP \(<Wt1>, <Wt2>, [<Xn|SP>, \#<imm>]\)!

**64-bit variant**

Applies when opc == 10.

STP \(<Xt1>, <Xt2>, [<Xn|SP>, \#<imm>]\)!

*Decode for all variants of this encoding*

boolean wback = TRUE;
boolean postindex = FALSE;

**Signed offset**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th></th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>imm7</td>
</tr>
<tr>
<td>L</td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
32-bit variant
Applies when opc == 00.
STP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm}>]

64-bit variant
Applies when opc == 10.
STP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm}>]

Decode for all variants of this encoding

boolean wback = FALSE;
boolean postindex = FALSE;

Notes for all encodings
For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly STP on page K1-11596.

Assembler symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.
For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.
For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.
For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainingUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
case c of
when Constraint_NONE    rt_unknown = FALSE;    // value stored is pre-writeback
when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
when Constraint_UNDEF   UNDEFINED;
when Constraint_NOP     EndOfInstruction();

Operation for all encodings

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
else
    data1 = X[t, datasize];
if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
else
    data2 = X[t2, datasize];
if HaveLSE2Ext() then
    bits(2*datasize) full_data;
    if BigEndian(AccType_NORMAL) then
        full_data = data1:data2;
    else
        full_data = data2:data1;
    Mem[address, 2*dbytes, AccType_NORMAL, TRUE] = full_data;
else
    Mem[address, dbytes, AccType_NORMAL] = data1;
    Mem[address+dbytes, dbytes, AccType_NORMAL] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.322 STR (immediate)

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for
the store is calculated from a base register and an immediate offset. For information about memory accesses, see
Load/store addressing modes on page C1-234.

Post-index

| 31 30 29 28|27 26 25 24|23 22 21 20| | 12|11 10 9 | 5 4 | 0 |
|-------------|-------------|-------------| | | | | |
| 1 x 1 1 0 0 0 0 0 | imm9 | 0 1 | Rn | Rt |

32-bit variant

Applies when size == 10.
STR <Wt>, [<Xn|SP>], #<simm>

64-bit variant

Applies when size == 11.
STR <Xt>, [<Xn|SP>], #<simm>

Decode for all variants of this encoding

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Pre-index

| 31 30 29 28|27 26 25 24|23 22 21 20| | 12|11 10 9 | 5 4 | 0 |
|-------------|-------------|-------------| | | | | |
| 1 x 1 1 0 0 0 0 0 | imm9 | 1 1 | Rn | Rt |

32-bit variant

Applies when size == 10.
STR <Wt>, [<Xn|SP>, #<simm>]

64-bit variant

Applies when size == 11.
STR <Xt>, [<Xn|SP>, #<simm>]

Decode for all variants of this encoding

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
Signed offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

size opc

32-bit variant

Applies when size == 10.

STR <Wt>, [<Xn|SP>{, #<pimm>}]

64-bit variant

Applies when size == 11.

STR <Xt>, [<Xn|SP>{, #<pimm>}]

Decode for all variants of this encoding

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.
For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
case c of
    when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
    when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
    when Constraint_UNDEF   UNDEFINED;
    when Constraint_NOP     EndOfInstruction();

Operation for all encodings

bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = X[t, datasize];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see Load/store addressing modes on page C1-234.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

### 32-bit variant

Applies when size == 10.

STR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

### 64-bit variant

Applies when size == 11.

STR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

#### Decode for all variants of this encoding

```plaintext
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

#### Assembler symbols

- **<Wt>** Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- **<Xt>** Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<Wm>** When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- **<Xm>** When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- **<extend>** Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:
  - UXTW when option = 010
  - LSL when option = 011
  - SXTW when option = 110
  - SXTX when option = 111
- **<amount>** For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:
  - #0 when S = 0
#2 when $S = 1$

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

#0 when $S = 0$
#3 when $S = 1$

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);

integer datasize = 8 << scale;

Operation

bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;
data = X[t, datasize];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.324 STRB (immediate)

Store Register Byte (immediate) stores the least significant byte of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see Load/store addressing modes on page C1-234.

### Post-index

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 0 0 0</td>
<td>imm9</td>
<td>0 1</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

STRB <Wt>, [<Xn|SP>], #<simm>

**Decode for this encoding**

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 0 0 0</td>
<td>imm9</td>
<td>1 1</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

STRB <Wt>, [<Xn|SP>, #<simm>]

**Decode for this encoding**

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 0</th>
<th>10</th>
<th>9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0</td>
<td>imm12</td>
<td>0 1 0 0</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

STRB <Wt>, [<Xn|SP>{, #<pimm>}

**Decode for this encoding**

boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly STRB (immediate) on page K1-11598.

Assembler symbols

<\Wt> 
Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> 
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<\simm> 
Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<\pimm> 
Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
    when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
    when Constraint_UNDEF   UNDEFINED;
    when Constraint_NOP     EndOfInstruction();

Operation for all encodings

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPLalignment();
  address = SP[];
else
  address = X[n, 64];

if !postindex then
  address = address + offset;

if rt_unknown then
  data = bits(8) UNKNOWN;
else
  data = X[t, 8];
Mem[address, 1, AccType_NORMAL] = data;

if wback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
else
    X[n, 64] = address;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.325  STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a 32-bit register to the calculated address. For information about memory accesses, see Load/store addressing modes on page C1-234.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

Extended register variant

Applies when option != 011.

STRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]

Shifted register variant

Applies when option == 011.

STRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]  

Decode for all variants of this encoding

if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend> Is the index extend specifier, encoded in the "option" field. It can have the following values:
  UXTW when option = 010
  SXTW when option = 110
  SXTX when option = 111
<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
**Operation**

bits(64) offset = ExtendReg(m, extend_type, 0, 64);
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n >= 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 8];
Mem[address, 1, AccType_NORMAL] = data;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.326   STRH (immediate)

Store Register Halfword (immediate) stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see Load/store addressing modes on page C1-234.

Post-index

| 31 30 29 28|27 26 25 24|23 22 21 20 | 12|11 10 9 | 5 4 | 0 |
|-----------|-----------|-----------|-----------|
| 0 1 1 1 0 0 0 0 0 | imm9 | 0 1 | Rn | Rt |

**Encoding**

STRH <Wt>, [<Xn|SP>], #<simm>

**Decode for this encoding**

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);

Pre-index

| 31 30 29 28|27 26 25 24|23 22 21 20 | 12|11 10 9 | 5 4 | 0 |
|-----------|-----------|-----------|-----------|
| 0 1 1 1 0 0 0 0 0 | imm9 | 1 1 | Rn | Rt |

**Encoding**

STRH <Wt>, [<Xn|SP>, #<simm>]

**Decode for this encoding**

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);

Unsigned offset

| 31 30 29 28|27 26 25 24|23 22 21 | 10 9 | 5 4 | 0 |
|-----------|-----------|-----------|-----------|
| 0 1 1 1 0 0 1 0 0 | imm12 | Rn | Rt |

**Encoding**

STRH <Wt>, [<Xn|SP>{, #<pimm>}]}

**Decode for this encoding**

boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
Notes for all encodings

For information about the CONSTRANGED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly STRH (immediate) on page K1-11598.

Assembler symbols

<\Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<\simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<\pimm> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <\pimm>/2.

Shared decode for all encodings

```c
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ContrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
    when Constraint_NONE rt_unknown = FALSE; // value stored is original value
    when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

Operation for all encodings

```c
bits(64) address;
bets(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];
Mem[address, 2, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
```
else
  x[n, 64] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.327 STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a 32-bit register to the calculated address. For information about memory accesses, see Load/store addressing modes on page C1-234.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

Encoding

\[
\begin{array}{cccccccccc}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & & & & \text{Rm} & \text{option} & S & 1 & 0 & \text{Rn} & \text{Rt} \\
\end{array}
\]

\text{size} \quad \text{opc}

\textit{Decode for this encoding}

\[
\begin{align*}
\text{if } & \text{option}<1> = '0' \text{ then UNDEFINED; // sub-word index} \\
\text{ExtendType } & \text{extend_type} = \text{DecodeRegExtend(option)}; \\
\text{integer } & \text{shift} = \text{if } S = '1' \text{ then } 1 \text{ else } 0;
\end{align*}
\]

\textit{Assembler symbols}

\texttt{<Wt>}

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\texttt{<Xn|SP>}

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\texttt{<Wm>}

When \text{option}<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

\texttt{<Xm>}

When \text{option}<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

\texttt{<extend>}

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when \texttt{<amount>} is omitted, encoded in the "option" field. It can have the following values:

- \texttt{UXTW} when \text{option} = 010
- \texttt{LSL} when \text{option} = 011
- \texttt{SXTW} when \text{option} = 110
- \texttt{SXTX} when \text{option} = 111

\texttt{<amount>}

Is the index shift amount, optional only when \texttt{<extend>} is not LSL. Where it is permitted to be optional, it defaults to \#0. It is encoded in the "S" field. It can have the following values:

- \#0 when \text{S} = 0
- \#1 when \text{S} = 1

\textit{Shared decode for all encodings}

\[
\begin{align*}
\text{integer } n & = \text{UInt(Rn)}; \\
\text{integer } t & = \text{UInt(Rt)}; \\
\text{integer } m & = \text{UInt(Rm)};
\end{align*}
\]
Operation

bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 16];
Mem[address, 2, AccType_NORMAL] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.328  STSETB, STSETLB

Atomic bit set on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- **STSETB** does not have release semantics.
- **STSETLB** stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-180.

For information about memory accesses see *Load/store addressing modes* on page C1-234.

This instruction is an alias of the LDSETB, LDSETAB, LDSETALB, LDSETLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDSETB, LDSETAB, LDSETALB, LDSETLB.
- The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.

**Integer**

(FEAT_LSE)

```
| 31 30 29 28| 27 26 25 24| 23 22 21 20 | 16|15|14|12|11|10| 9 | 5| 4 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|
| 0 0 1| 1| 1| 0| 0| 0| 0|R 1|Rs| 0| 0| 1| 1| 0 0| Rn| 1|1| 1| 1|1|
```

**No memory ordering variant**

Applies when R == 0.

STSETB \(<Ws>\), \([<Xn>|SP>]\)

is equivalent to

LDSETB \(<Ws>\), WZR, \([<Xn>|SP>]\)

and is always the preferred disassembly.

**Release variant**

Applies when R == 1.

STSETLB \(<Ws>\), \([<Xn>|SP>]\)

is equivalent to

LDSETLB \(<Ws>\), WZR, \([<Xn>|SP>]\)

and is always the preferred disassembly.

**Assembler symbols**

- \(<Ws>\)   
  Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

- \(<Xn>|SP>\)  
  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.329 STSETH, STSETLH

Atomic bit set on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETH does not have release semantics.
- STSETLH stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDSETH, LDSETAH, LDSETALH, LDSETLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDSETH, LDSETAH, LDSETALH, LDSETLH.
- The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)

\[
\begin{array}{cccccc|cccc|cccc|c}
\text{size} & A & \text{opc} & \text{Rt} \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & R & 1 & R_5 & 0 & 0 & 1 & 1 & 0 & 0 & R_n & 1 & 1 & 1 & 1
\end{array}
\]

No memory ordering variant

Applies when \( R == 0 \).

STSETH \(<Ws>\), \([<Xn|SP>]\)

is equivalent to

LDSETH \(<Ws>\), WZR, \([<Xn|SP>]\)

and is always the preferred disassembly.

Release variant

Applies when \( R == 1 \).

STSETLH \(<Ws>\), \([<Xn|SP>]\)

is equivalent to

LDSETLH \(<Ws>\), WZR, \([<Xn|SP>]\)

and is always the preferred disassembly.

Assembler symbols

- \(<Ws>\) Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- \(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.330  STSET, STSETL

Atomic bit set on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit
doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back
to memory.

- STSET does not have release semantics.
- STSETL stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDSET, LDSETA, LDSETAL, LDSETL instruction. This means that:

- The encodings in this description are named to match the encodings of LDSET, LDSETA, LDSETAL,
  LDSETL.
- The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode for this
  instruction.

### Integer

(FEAT_LSE)

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 12|11 10 9 | 5 4 | 0 |
 1 x 1 1 1 0 0 0 0 R 1 Rs 0 0 1 1 0 0 Rn 1 1 1 1 1
```

#### 32-bit LDSET alias variant

Applies when `size == 10 && R == 0`.

STSET `<Ws>`, `<Xn|SP>`

is equivalent to

LDSET `<Ws>`, `WZR`, `<Xn|SP>`

and is always the preferred disassembly.

#### 32-bit LDSETL alias variant

Applies when `size == 10 && R == 1`.

STSETL `<Ws>`, `<Xn|SP>`

is equivalent to

LDSETL `<Ws>`, `WZR`, `<Xn|SP>`

and is always the preferred disassembly.

#### 64-bit LDSET alias variant

Applies when `size == 11 && R == 0`.

STSET `<Xs>`, `<Xn|SP>`

is equivalent to

LDSET `<Xs>`, `XZR`, `<Xn|SP>`

and is always the preferred disassembly.
64-bit LDSETL alias variant

Applies when size == 11 && R == 1.

STSETL <Xs>, [<Xn|SP>]

is equivalent to

LDSETL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSET, LDSETA, LDSETEAL, LDSETL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.331 STSMAXB, STSMAXLB

Atomic signed maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAXB does not have release semantics.
- STSMAXLB stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB.
- The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode for this instruction.

**Integer**

(FEAT_LSE)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 0 0 0 0</td>
<td>R 1</td>
<td>Rs 0 1 0 0 0 0</td>
<td>Rn 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**No memory ordering variant**

Applies when \( R = 0 \).

STSMAXB \(<Ws>, [<Xn|SP>]\)

is equivalent to

LDSMAXB \(<Ws>, WZR, [<Xn|SP>]\)

and is always the preferred disassembly.

**Release variant**

Applies when \( R = 1 \).

STSMAXLB \(<Ws>, [<Xn|SP>]\)

is equivalent to

LDSMAXLB \(<Ws>, WZR, [<Xn|SP>]\)

and is always the preferred disassembly.

**Assembler symbols**

- \(<Ws>\) Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- \(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
### C6.2.332 STSMAXH, STSMAXLH

Atomic signed maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- **STSMAXH** does not have release semantics.
- **STSMAXLH** stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-180.

For information about memory accesses see *Load/store addressing modes* on page C1-234.

This instruction is an alias of the LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.
- The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

#### Integer

(Feats LSE)

<table>
<thead>
<tr>
<th>[31 30 29 28]27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14</th>
<th>12</th>
<th>11 10 9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**No memory ordering variant**

Applies when \( R == 0 \).

STSMAXH \(<Ws>, [<Xn|SP>]\)

is equivalent to

LDSMAXH \(<Ws>, WZR, [<Xn|SP>]\)

and is always the preferred disassembly.

**Release variant**

Applies when \( R == 1 \).

STSMAXLH \(<Ws>, [<Xn|SP>]\)

is equivalent to

LDSMAXLH \(<Ws>, WZR, [<Xn|SP>]\)

and is always the preferred disassembly.

**Assembler symbols**

- \(<Ws>\) is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- \(<Xn|SP>\) is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.333  STSMAX, STSMAXL

Atomic signed maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAX does not have release semantics.
- STSMAXL stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL.
- The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

**Integer**

(FEAT_LSE)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>A</td>
<td>opc</td>
<td>Rn</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**32-bit LDSMAX alias variant**

Applies when size == 10 && R == 0.

STSMAX \(<Ws>\), [\(<Xn>|\text{SP}\>]\)

is equivalent to

LDSMAX \(<Ws>\), WZR, [\(<Xn>|\text{SP}\>]\)

and is always the preferred disassembly.

**32-bit LDSMAXL alias variant**

Applies when size == 10 && R == 1.

STSMAXL \(<Ws>\), [\(<Xn>|\text{SP}\>]\)

is equivalent to

LDSMAXL \(<Ws>\), WZR, [\(<Xn>|\text{SP}\>]\)

and is always the preferred disassembly.

**64-bit LDSMAX alias variant**

Applies when size == 11 && R == 0.

STSMAX \(<Xs>\), [\(<Xn>|\text{SP}\>]\)

is equivalent to

LDSMAX \(<Xs>\), XZR, [\(<Xn>|\text{SP}\>]\)

and is always the preferred disassembly.
64-bit LDSMAXL alias variant

Applies when size == 11 \&\& R == 1.

\texttt{STSMAXL <Xs>, [<Xn|SP>]}

is equivalent to

\texttt{LDSMAXL <Xs>, XZR, [<Xn|SP>]}

and is always the preferred disassembly.

Assembler symbols

\texttt{<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.}

\texttt{<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.}

\texttt{<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.}

Operation

The description of \texttt{LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL} gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.334   STSMINB, STSMINLB

Atomic signed minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMINB does not have release semantics.
- STSMINLB stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB.
- The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

**Integer**

**(FEAT_LSE)**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 0</td>
<td>0 0 0 0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0 1 0 1 0 0</td>
<td>Rn</td>
</tr>
</tbody>
</table>
```

**No memory ordering variant**

Applies when \( R = 0 \).

STSMINB \(<Ws\), \<[Xn|SP]\>

is equivalent to

LDSMINB \(<Ws\), WZR, \<[Xn|SP]\>

and is always the preferred disassembly.

**Release variant**

Applies when \( R = 1 \).

STSMINLB \(<Ws\), \<[Xn|SP]\>

is equivalent to

LDSMINLB \(<Ws\), WZR, \<[Xn|SP]\>

and is always the preferred disassembly.

**Assembler symbols**

\(<Ws>\)     Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

\(<Xn|SP>\)    Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.335  STSMINH, STSMINLH

Atomic signed minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMINH does not have release semantics.
- STSMINLH stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.
- The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

**Integer**

(FEAT_LSE)

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**No memory ordering variant**

Applies when \( R = 0 \).

STSMINH \(<Ws>, [<Xn|SP>]\)

is equivalent to

LDSMINH \(<Ws>, WZR, [<Xn|SP>]\)

and is always the preferred disassembly.

**Release variant**

Applies when \( R = 1 \).

STSMINLH \(<Ws>, [<Xn|SP>]\)

is equivalent to

LDSMINLH \(<Ws>, WZR, [<Xn|SP>]\)

and is always the preferred disassembly.

**Assembler symbols**

\(<Ws>\)  Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

\(<Xn|SP>\)  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.336  STSMIN, STSMINL

Atomic signed minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMIN does not have release semantics.
- STSMINL stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDSMIN, LDSMINA, LDSMINAL, LDSMINL instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.
- The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

Integer

(FEATURE LSE)

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>16</th>
<th>15 14</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>size</td>
<td>A</td>
<td>Rs</td>
<td>opc</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

32-bit LDSMIN alias variant

Applies when size == 10 && R == 0.

STSMIN <Ws>, [<Xn|SP>] is equivalent to

LDSMIN <Ws>, WZR, [<Xn|SP>] and is always the preferred disassembly.

32-bit LDSMINL alias variant

Applies when size == 10 && R == 1.

STSMINL <Ws>, [<Xn|SP>] is equivalent to

LDSMINL <Ws>, WZR, [<Xn|SP>] and is always the preferred disassembly.

64-bit LDSMIN alias variant

Applies when size == 11 && R == 0.

STSMIN <Xs>, [<Xn|SP>] is equivalent to

LDSMIN <Xs>, XZR, [<Xn|SP>] and is always the preferred disassembly.
64-bit LDSMINL alias variant

Applies when size == 11 && R == 1.

STSMINL <Xs>, [<Xn|SP>]

is equivalent to

LDSMINL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler symbols

<w> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.337   STTR

Store Register (unprivileged) stores a word or doubleword from a register to memory. The address that is used for
the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value
of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is \{1, 1\}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the
instruction is executed. For information about memory accesses, see Load/store addressing modes on page C1-234.

32-bit variant

Applies when size == 10.
STTR <Wt>, [<Xn|SP>{, #<simm>}]  

64-bit variant

Applies when size == 11.
STTR <Xt>, [<Xn|SP>{, #<simm>}]  

Decode for all variants of this encoding

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded
in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;

**Operation**

bits(64) address;
bits(datasize) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
address = address + offset;
data = X[t, datasize];
Mem[address, datasize DIV 8, acctype] = data;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.338  STTRB

Store Register Byte (unprivileged) stores a byte from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/store addressing modes on page C1-234.

Encoding

STTRB <Wt>, [<Xn|SP>{, #<simm>}]

Decode for this encoding

bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Wt>  Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType actype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  actype = AccType_UNPRIV;
else
  actype = AccType_NORMAL;

boolean tag_checked = n != 31;

Operation

bits(64) address;
bits(8) data;
if HaveMTE2Ext() then
SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPLignment();
  address = SP[];
else
  address = X[n, 64];

address = address + offset;

data = X[t, 8];
Mem[address, 1, acctype] = data;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.339  STTRH

Store Register Halfword (unprivileged) stores a halfword from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is \{1, 1\}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/store addressing modes on page C1-234.

Encoding

STTRH <Wt>, [<Xn|SP>{, #<simm>}]  

**Decode for this encoding**

bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt) && HCR_EL2.<NV,NV1> == '11';
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
   acctype = AccType_UNPRIV;
else
   acctype = AccType_NORMAL;

boolean tag_checked = n != 31;

Operation

bits(64) address;
bits(16) data;
if HaveMTE2Ext() then
SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 16];
Mem[address, 2, acctype] = data;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.340  STUMAXB, STUMAXLB

Atomic unsigned maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- **STUMAXB** does not have release semantics.
- **STUMAXLB** stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-180.

For information about memory accesses see *Load/store addressing modes* on page C1-234.

This instruction is an alias of the **LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB** instruction. This means that:

- The encodings in this description are named to match the encodings of **LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB**.
- The description of **LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB** gives the operational pseudocode for this instruction.

**Integer**

(FEAT_LSE)

```
[31 30 29 28|27 26 25 24|23 22 21 20]  16|15 14 12|11 10 9 | 5 4 | 0 |
  0 0 1 1 1 0 0 0 0 R 1 Rs 0 1 1 0 0 0 Rn 1 1 1 1 1
```

**No memory ordering variant**

Applies when \( R = 0 \).

**STUMAXB** <\( Ws \), [\( Xn | SP \)]

is equivalent to

**LDUMAXB** <\( Ws \), WZR, [\( Xn | SP \)]

and is always the preferred disassembly.

**Release variant**

Applies when \( R = 1 \).

**STUMAXLB** <\( Ws \), [\( Xn | SP \)]

is equivalent to

**LDUMAXLB** <\( Ws \), WZR, [\( Xn | SP \)]

and is always the preferred disassembly.

**Assembler symbols**

- <\( Ws \)> is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <\( Xn | SP \)> is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.341   STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAXH does not have release semantics.
- STUMAXLH stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.
- The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

**Integer**

(FEAT_LSE)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 0 0 0</td>
<td>0</td>
<td>0</td>
<td>R 1</td>
<td>Rs 0 1 1 0 0 0</td>
<td>Rn 1 1 1 1</td>
<td></td>
</tr>
</tbody>
</table>

**No memory ordering variant**

Applies when R == 0.

STUMAXH <Ws>, [<Xn|SP>]

is equivalent to

LDUMAXH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Release variant**

Applies when R == 1.

STUMAXLH <Ws>, [<Xn|SP>]

is equivalent to

LDUMAXLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Assembler symbols**

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

The description of `LDUMAXH`, `LDUMAXAH`, `LDUMAXALH`, `LDUMAXLH` gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.342  **STUMAX, STUMAXL**

Atomic unsigned maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- **STUMAX** does not have release semantics.
- **STUMAXL** stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release* on page B2-180.

For information about memory accesses see *Load/store addressing modes* on page C1-234.

This instruction is an alias of the LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL.
- The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

**Integer**

(FEAT_LSE)

```
size A opc Rt
|31 30 29 28|27 26 25 24|23 22 21 20|16|15 14 12|11 10 9 | | 5 4 | 0 |
1 x 1 1 1 0 0 0 0 R 1 Rn 0 1 1 0 0 0 Rn 1 1 1 1
```

**32-bit LDUMAX alias variant**

Applies when `size == 10 && R == 0`.

STUMAX `<Ws>`, `<Xn|SP>`

is equivalent to

LDUMAX `<Ws>`, WZR, `<Xn|SP>`

and is always the preferred disassembly.

**32-bit LDUMAXL alias variant**

Applies when `size == 10 && R == 1`.

STUMAXL `<Ws>`, `<Xn|SP>`

is equivalent to

LDUMAXL `<Ws>`, WZR, `<Xn|SP>`

and is always the preferred disassembly.

**64-bit LDUMAX alias variant**

Applies when `size == 11 && R == 0`.

STUMAX `<Xs>`, `<Xn|SP>`

is equivalent to

LDUMAX `<Xs>`, XZR, `<Xn|SP>`

and is always the preferred disassembly.
64-bit LDUMAXL alias variant

 Applies when size == 11 && R == 1.

 STUMAXL <Xs>, [<Xn|SP>]

 is equivalent to

 LDUMAXL <Xs>, XZR, [<Xn|SP>]

 and is always the preferred disassembly.

Assembler symbols

 <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

 <Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

 <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

 The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

Operational information

 If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.343  STUMINB, STUMINLB

Atomic unsigned minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMINB does not have release semantics.
- STUMINLB stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB.
- The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)

<table>
<thead>
<tr>
<th>size</th>
<th>A</th>
<th>opc</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>R1</td>
<td>Rs</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>28</td>
<td>27</td>
</tr>
</tbody>
</table>

No memory ordering variant

Applies when \( R = 0 \).

STUMINB \(<Ws>, [<Xn|SP>]\)

is equivalent to

LDUMINB \(<Ws>, WZR, [<Xn|SP>]\)

and is always the preferred disassembly.

Release variant

Applies when \( R = 1 \).

STUMINLB \(<Ws>, [<Xn|SP>]\)

is equivalent to

LDUMINLB \(<Ws>, WZR, [<Xn|SP>]\)

and is always the preferred disassembly.

Assembler symbols

- \(<Ws>\) Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- \(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.344  STUMINH, STUMINLH

Atomic unsigned minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMINH does not have release semantics.
- STUMINLH stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.
- The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

**Integer**

(FEAT_LSE)

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

size A opc Rt

**No memory ordering variant**

Applies when $R == 0$.

STUMINH $<\text{Ws}>$, [<Xn\text{|SP}>]

is equivalent to

LDUMINH $<\text{Ws}>$, WZR, [<Xn\text{|SP}>]

and is always the preferred disassembly.

**Release variant**

Applies when $R == 1$.

STUMINLH $<\text{Ws}>$, [<Xn\text{|SP}>]

is equivalent to

LDUMINLH $<\text{Ws}>$, WZR, [<Xn\text{|SP}>]

and is always the preferred disassembly.

**Assembler symbols**

- $<\text{Ws}>$  
  Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

- $<\text{Xn\text{|SP}>}$  
  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
**Operation**

The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.345   STUMIN, STUMINL

Atomic unsigned minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMIN does not have release semantics.
- STUMINL stores to memory with release semantics, as described in Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

This instruction is an alias of the LDUMIN, LDUMINA, LDUMINAL, LDUMINL instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMIN, LDUMINA, LDUMINAL, LDUMINL.
- The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode for this instruction.

**Integer**

(FEAT_LSE)

| [31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14| 12|11 10 9 | 5 4 | 0 ] | size A opc Rt |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | R | 1 | Rs | 0 | 1 | 1 | 1 | 0 | 0 | Rn | 1 | 1 | 1 | 1 |

**32-bit LDUMIN alias variant**

Applies when size == 10 & R == 0.

STUMIN <Ws>, [<Xn|SP>]

is equivalent to

LDUMIN <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**32-bit LDUMINL alias variant**

Applies when size == 10 & R == 1.

STUMINL <Ws>, [<Xn|SP>]

is equivalent to

LDUMINL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**64-bit LDUMIN alias variant**

Applies when size == 11 & R == 0.

STUMIN <Xs>, [<Xn|SP>]

is equivalent to

LDUMIN <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
64-bit LDUMINL alias variant

Applies when size == 11 && R == 1.
STUMINL <Xs>, [<Xn|SP>]
is equivalent to
LDUMINL <Xs>, XZR, [<Xn|SP>]
and is always the preferred disassembly.

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the
contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the
contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode for this
instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.346 STUR

Store Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see Load/store addressing modes on page C1-234.

32-bit variant

Applies when size == 10.

STUR <Wt>, [<Xn|SP>{, #<simm>}]

64-bit variant

Applies when size == 11.

STUR <Xt>, [<Xn|SP>{, #<simm>}]

Decode for all variants of this encoding

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
integer datasize = 8 ⋅ scale;
boolean tag_checked = n != 31;

Operation

bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
address = X[n, 64];
address = address + offset;
data = X[t, datasize];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;

Operational information
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.347  STURB

Store Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about memory accesses, see Load/store addressing modes on page C1-234.

**Encoding**

STURB \(<Wt>, [<Xn|SP>{, #<simm}>] \)

**Decode for this encoding**

bits(64) offset = SignExtend(imm9, 64);

**Assembler symbols**

\(<Wt>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<\text{simm}>\) Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared decode for all encodings**

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

**Operation**

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];
address = address + offset;
data = X[t, 8];
Mem[address, 1, AccType_NORMAL] = data;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.348   STURH

Store Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about memory accesses, see Load/store addressing modes on page C1-234.

Encoding

STURH <Wt>, [<Xn|SP>{, #<simm>}]  

Decode for this encoding

bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Wt>           Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>        Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>        Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Operation

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then  
    SetTagCheckedInstruction(tag_checked);

if n == 31 then  
    CheckSPAlignment();
    address = SP[];
else  
    address = X[n, 64];

address = address + offset;

data = X[t, 16];
Mem[address, 2, AccType_NORMAL] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.349 STXP

Store Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores on page B2-211. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity on page B2-154 and Alignment of data accesses on page B2-189. If a 64-bit pair Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about memory accesses, see Load/store addressing modes on page C1-234.

32-bit variant

Applies when sz == 0.

STXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]

64-bit variant

Applies when sz == 1.

STXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]

Decode for all variants of this encoding

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
  Constraint c = ConstraineUnpredictable(Unpredictable_DATAOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
  if s == n & n != 31 then
    Constraint c = ConstraineUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly STXP on page K1-11599.
Assembler symbols

\(<\text{Ws}>\)
Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0  If the operation updates memory.
1  If the operation fails to update memory.

\(<\text{Xt1}>\)
Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

\(<\text{Xt2}>\)
Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

\(<\text{Wt1}>\)
Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

\(<\text{Wt2}>\)
Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

\(<\text{Xn}|\text{SP}>\)
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- \(<\text{Ws}>\) is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If \(\text{AArch64.ExclusiveMonitorsPass() \ returns \ True}\), the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If \(\text{AArch64.ExclusiveMonitorsPass() \ returns \ False}\) and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

\begin{verbatim}
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
elif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n, 64];
if rt_unknown then
  data = bits(datasize) UNKNOWN;
else
  bits(datasize DIV 2) el1 = X[t, datasize DIV 2];
  bits(datasize DIV 2) el2 = X[t2, datasize DIV 2];
  data = if BigEndian(AccType_ATOMIC) then el1:el2 else el2:el1;
  bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
  // This atomic write will be rejected if it does not refer
\end{verbatim}
// to the same physical locations after address translation.
Mem[address, dbytes, AccType_ATOMIC] = data;
status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);

Operational information
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.350  STXR

Store Exclusive Register stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores on page B2-211. For information about memory accesses see Load/store addressing modes on page C1-234.

### 32-bit variant

Applies when size == 10.

STXR `<Ws>`, `<Wt>`, `[<Xn|SP>{,#0}]`

### 64-bit variant

Applies when size == 11.

STXR `<Ws>`, `<Xt>`, `[<Xn|SP>{,#0}]`

#### Decode for all variants of this encoding

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release
integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_UNDEF   UNDEFINED;
            when Constraint_NOP     EndOfInstruction();
```

#### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly STXR on page K1-11599.

#### Assembler symbols

`<Ws>`

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0      If the operation updates memory.
If the operation fails to update memory.

\(<X_t>\) Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<W_t>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<X_n|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- \(<W_s>\) is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If \(\text{AArch64.ExclusiveMonitorsPass()}\) returns \(\text{TRUE}\), the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If \(\text{AArch64.ExclusiveMonitorsPass()}\) returns \(\text{FALSE}\) and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

**Operation**

\[
\begin{align*}
\text{bits(64) address; } \\
\text{bits(elsize) data; } \\
\text{constant integer } dbytes = \text{elsize DIV 8; } \\
\text{if HaveMTE2Ext() then } \\
\quad \text{SetTagCheckedInstruction(tag_checked);} \\
\text{if } n == 31 \text{ then } \\
\quad \text{CheckSPAlignment(); } \\
\text{elsif } \text{rn}_\text{unknown then} \\
\quad \text{address } = \text{bits(64) UNKNOWN;} \\
\text{else} \\
\quad \text{address } = X[n, 64]; \\
\text{if } \text{rt}_\text{unknown then} \\
\quad \text{data } = \text{bits(elsize) UNKNOWN;} \\
\text{else} \\
\quad \text{data } = X[t, \text{elsize}]; \\
\text{bit status } = '1'; \\
\text{// Check whether the Exclusives monitors are set to include the } \\
\text{// physical memory locations corresponding to virtual address } \\
\text{// range [address, address+dbytes-1]. } \\
\text{if } \text{AArch64.ExclusiveMonitorsPass(address, dbytes) then } \\
\quad \text{// This atomic write will be rejected if it does not refer } \\
\quad \text{// to the same physical locations after address translation. } \\
\quad \text{Mem[address, dbytes, AccType_ATOMIC] } = \text{data;} \\
\quad \text{status } = \text{ExclusiveMonitorsStatus(); } \\
\text{X[s, 32] } = \text{ZeroExtend(status, 32);} \\
\end{align*}
\]

**Operational information**

If \(\text{PSTATE.DIT} = 1\), the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.351  STXRB

Store Exclusive Register Byte stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores on page B2-211. The memory access is atomic.

For information about memory accesses see Load/store addressing modes on page C1-234.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>1 0 0 0</td>
<td>0 0 0</td>
<td>Rs</td>
<td>0</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
</tbody>
</table>

**Encoding**

STXRB <Ws>, <Wt>, [<Xn|SP>{#0}]

**Decode for this encoding**

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);  // ignored by all loads and store-release

boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;  // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn_unknown = TRUE;  // address is UNKNOWN
            when Constraint_UNDEF   UNDEFINED;
            when Constraint_NOP     EndOfInstruction();
```

**Notes for all encodings**

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly STXRB on page K1-11599.

**Assembler symbols**

<Ws>  Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0       If the operation updates memory.
1       If the operation fails to update memory.

<Wt>  Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Aborts**
If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- \(<ws>\) is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

**Operation**

```plaintext
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[ ];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
    X[s, 32] = ZeroExtend(status, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.352 STXRH

Store Exclusive Register Halfword stores a halfword from a register to memory if the PE has exclusive access to
the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.
See *Synchronization and semaphores* on page B2-211. The memory access is atomic.

For information about memory accesses see *Load/store addressing modes* on page C1-234.

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16 15 14 10 9 5 4 0
0 1 0 0 1 0 0 0 0 0 1 1 1 1 1
size L 0 0 Rt2
```

**Encoding**

STXRH \(<Ws>\), \(<Wt>\), \([<Xn|SP>\{,#0}\] \)

**Decode for this encoding**

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs); // ignored by all loads and store-release

boolean tag_checked = n != 31;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
            when Constraint_UNDEF   UNDEFINED;
            when Constraint_NOP     EndOfInstruction();
```

**Assembler symbols**

\(<Ws>\) Is the 32-bit name of the general-purpose register into which the status result of the store exclusive
is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

\(<Wt>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Aborts and alignment**

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- \(<Ws>\) is not updated.
A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

**Operation**

```plaintext
bits(64) address;
b bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

**Operational information**

If `PSTATE.DIT` is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C6.2.353   STZ2G

Store Allocation Tags, Zeroing stores an Allocation Tag to two Tag granules of memory, zeroing the associated data locations. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

**Post-index**

(FEAT_MTE)

```
[31 30 29 28|27 26 25 24|23 22 21 20]   |   12|11 10 9 |   5 4 |   0 |
 1 1 0 1 1 0 0 1 1 1 1 1 immm9 0 1 Xn Xt
```

**Encoding**

STZ2G <Xt|SP>, [<Xn|SP>], #<imm>

**Decode for this encoding**

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;

**Pre-index**

(FEAT_MTE)

```
[31 30 29 28|27 26 25 24|23 22 21 20]   |   12|11 10 9 |   5 4 |   0 |
 1 1 0 1 1 0 0 1 1 1 1 1 immm9 1 1 Xn Xt
```

**Encoding**

STZ2G <Xt|SP>, [<Xn|SP>], #<imm>!

**Decode for this encoding**

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;

**Signed offset**

(FEAT_MTE)

```
[31 30 29 28|27 26 25 24|23 22 21 20]   |   12|11 10 9 |   5 4 |   0 |
 1 1 0 1 1 0 0 1 1 1 1 1 immm9 1 0 Xn Xt
```
**Encoding**

STZ2G <Xt|SP>, [<Xn|SP>{, #<simm>}]

**Decode for this encoding**

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

**Assembler symbols**

<Xt|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Xt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

**Operation for all encodings**

bits(64) address;
bits(64) data = if t == 31 then SP[] else X[t, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
SetTagCheckedInstruction(FALSE);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
if !postindex then
    address = address + offset;
if address != Align(address, TAG_GRANULE) then
    AArch64.Abort(address, AlignmentFault(AccType_NORMAL, TRUE, FALSE));
Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);
Mem[address+TAG_GRANULE, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);
AArch64.MemTag[address, AccType_NORMAL] = tag;
AArch64.MemTag[address+TAG_GRANULE, AccType_NORMAL] = tag;
if writeback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
C6.2.354  STZG

Store Allocation Tag. Zeroing stores an Allocation Tag to memory, zeroing the associated data location. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

**Post-index**

(Feat_MTE)

```markdown
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 0 1 0 1 1</td>
<td>imm9</td>
<td>0 1</td>
<td>Xn</td>
<td>Xt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

STZG <Xt|SP>, [<Xn|SP>], #<imm>

**Decode for this encoding**

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bias(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;

**Pre-index**

(Feat_MTE)

```markdown
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 0 1 0 1 1</td>
<td>imm9</td>
<td>1 1</td>
<td>Xn</td>
<td>Xt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

STZG <Xt|SP>, [<Xn|SP>, #<imm>]

**Decode for this encoding**

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bias(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;

**Signed offset**

(Feat_MTE)

```markdown
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 0 1 0 1 1</td>
<td>imm9</td>
<td>1 0</td>
<td>Xn</td>
<td>Xt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Encoding

STZG <Xt|SP>, [<Xn|SP>{, #<simm>}]

Decode for this encoding

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

Assembler symbols

<Xt|SP>  Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Xt" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm>  Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation for all encodings

bits(64) address;

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if address != Align(address, TAG_GRANULE) then
    AArch64.Abort(address, AlignmentFault(AccType.NORMAL, TRUE, FALSE));

Mem[address, TAG_GRANULE, AccType.NORMAL] = Zeros(TAG_GRANULE * 8);

bits(64) data = if t == 31 then SP[] else X[t, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, AccType.NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
C6.2.355  STZGM

Store Tag and Zero Multiple writes a naturally aligned block of N Allocation Tags and stores zero to the associated data locations, where the size of N is identified in DCZID_EL0.BS, and the Allocation Tag is taken from the source register bits<3:0>.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

Integer

(FEAT_MTE2)

Encode

STZGM <Xt>, [<Xn|SP>]

Decode for this encoding

if !HaveMTE2Ext() then UNDEFINED;
integer t = UInt(Xt);
in integer n = UInt(Xn);

Assembler symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Xt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

Operation

if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = X[t, 64];
bits(4) tag = data<3:0>;
bits(64) address;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

integer size = 4 * (2 ^ (UInt(DCZID_EL0.BS)));
address = Align(address, size);
in integer count = size >> LOG2_TAG_GRANULE;
for i = 0 to count-1
    AArch64.MemTag[address, AccType_NORMAL] = tag;
    Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(8 * TAG_GRANULE);
    address = address + TAG_GRANULE;
C6.2.356   SUB (extended register)

Subtract (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.

32-bit variant
Applies when sf == 0.
SUB <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

64-bit variant
Applies when sf == 1.
SUB <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
datatype datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;

Assembler symbols

<Wd|WSP>  Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP>  Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm>  Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd|SP>  Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP>  Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R>  Is a width specifier, encoded in the "option" field. It can have the following values:
   W     when option = 00x
   W     when option = 010
   X     when option = x11
   W     when option = 10x
   W     when option = 11x

<m>  Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

- UXTB when option = 000
- UXTH when option = 001
- LSL|UXTW when option = 010
- UTX when option = 011
- SXTB when option = 100
- SXTH when option = 101
- SXTW when option = 110
- SXTX when option = 111

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

- UXTB when option = 000
- UXTH when option = 001
- UTXW when option = 010
- LSL|UXTX when option = 011
- SXTB when option = 100
- SXTH when option = 101
- SXTW when option = 110
- SXTX when option = 111

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

**Operation**

```plaintext
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);
operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');
if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.357   **SUB (immediate)**

Subtract (immediate) subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**32-bit variant**

Applies when \( sf = 0 \).

`SUB \(<Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>} \)`

**64-bit variant**

Applies when \( sf = 1 \).

`SUB \(<Xd|SP>, <Xn|SP>, #<imm>{, <shift>} \)`

**Decode for all variants of this encoding**

```plaintext```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case sh of
when '0' imm = ZeroExtend(imm12, datasize);
when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

**Assembler symbols**

- `<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:
  - LSL #0 when `sh = 0`
  - LSL #12 when `sh = 1`

**Operation**

```plaintext```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[<datasize-1:0>] else X[n, datasize];
bits(datasize) operand2;

operand2 = NOT(imm);
(result, -) = AddWithCarry(operand1, operand2, '1');
```

---
if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.358 SUB (shifted register)

Subtract (shifted register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

This instruction is used by the alias NEG (shifted register). See Alias conditions for details of when each alias is preferred.

32-bit variant

Applies when sf == 0.

SUB <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit variant

Applies when sf == 1.

SUB <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEG (shifted register)</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

<table>
<thead>
<tr>
<th>Shift Type</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL</td>
<td>shift = 00</td>
</tr>
<tr>
<td>LSR</td>
<td>shift = 01</td>
</tr>
<tr>
<td>ASR</td>
<td>shift = 10</td>
</tr>
<tr>
<td>(reserved)</td>
<td>shift = 11</td>
</tr>
</tbody>
</table>

The encoding shift = 11 is reserved.

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Operation**

```c
bits(datasize) result;
basis(datasize) operand1 = X[n, datasize];
basis(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');
X[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.359 SUBG

Subtract with Tag subtracts an immediate value scaled by the Tag granule from the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

Integer

(FEAT_MTE)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>16 15 14 13</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 0 1 1</td>
<td>0 1 0 0 0 1 1 0</td>
<td>uimm6</td>
<td><a href="0">0</a></td>
<td>uimm4</td>
<td>Xn</td>
<td>Xd</td>
</tr>
</tbody>
</table>

Encoding

SUBG <Xd|SP>, <Xn|SP>, #uimm6, #uimm4

Decode for this encoding

if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);

Assembler symbols

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Xn" field.
<uimm6> Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the "uimm6" field.
<uimm4> Is an unsigned immediate, in the range 0 to 15, encoded in the "uimm4" field.

Operation

bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;
if AArch64.AllocationTagAccessIsEnabled(AccType_NORMAL) then
  rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
else
  rtag = '0000';
(result, -) = AddWithCarry(operand1, NOT(offset), '1');
result = AArch64.AddressWithAllocationTag(result, AccType_NORMAL, rtag);
if d == 31 then
  SP[] = result;
else
  X[d, 64] = result;
C6.2.360 SUBP

Subtract Pointer subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register.

**Integer**

(FEAT_MTE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 5 | 4 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |

**Encoding**

SUBP <Xd>, <Xn|SP>, <Xm|SP>

**Decode for this encoding**

if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);

**Assembler symbols**

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.

<Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

**Operation**

bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) operand2 = if m == 31 then SP[] else X[m, 64];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);

bits(64) result;
operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');
X[d, 64] = result;
C6.2.361   SUBPS

Subtract Pointer, setting Flags subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register. It updates the condition flags based on the result of the subtraction.

This instruction is used by the alias CMPP. See Alias conditions for details of when each alias is preferred.

Integer

(FEAT_MTE)

Encoding

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 0 1 0 1 0 1 0 1 0</td>
<td>0 0 0 0 0 0 0</td>
<td>0 0</td>
<td></td>
</tr>
</tbody>
</table>

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPP</td>
<td>S == '1' &amp; &amp; Xd == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.

<Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

Operation

bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) operand2 = if m == 31 then SP[] else X[m, 64];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);

bits(64) result;
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = nzcv;
X[d, 64] = result;
C6.2.362   SUBS (extended register)

Subtract (extended register), setting flags, subtracts a sign or zero-extended register value, followed by an optional
left shift amount, from a register value, and writes the result to the destination register. The argument that is extended
from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the
result.

This instruction is used by the alias CMP (extended register). See Alias conditions for details of when each alias is
preferred.

32-bit variant

Applies when sf == 0.

SUBS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

64-bit variant

Applies when sf == 1.

SUBS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP (extended register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn"
field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn"
field.
<R>   Is a width specifier, encoded in the "option" field. It can have the following values:
W    when option = 00x
W    when option = 010
X    when option = x11
W    when option = 10x
W    when option = 110

<m>  Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in
the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the
"option" field. It can have the following values:
UXTB  when option = 000
UXTH  when option = 001
LSL|UXTW when option = 010
UXTX  when option = 011
SXTB  when option = 100
SKTH  when option = 101
SXTW  when option = 110
SXTX  when option = 111

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when
"imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the
"option" field. It can have the following values:
UXTB  when option = 000
UXTH  when option = 001
UXTW  when option = 010
LSL|UXTX when option = 011
SXTB  when option = 100
SKTH  when option = 101
SXTW  when option = 110
SXTX  when option = 111

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3"
is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in
the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL,
and is optional when <extend> is present but not LSL.

Operation

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;
X[d, datasize] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.363 SUBS (immediate)

Subtract (immediate), setting flags, subtracts an optionally-shifted immediate value from a register value, and writes
the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMP (immediate). See Alias conditions for details of when each alias is
preferred.

32-bit variant
Applies when $sf == 0$.

```
SUBS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit variant
Applies when $sf == 1$.

```
SUBS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

Decode for all variants of this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case sh of
   when '0' imm = ZeroExtend(imm12, datasize);
   when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP (immediate)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

- `<id>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wd>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh"
  field. It can have the following values:
  - LSL #0 when $sh = 0$
  - LSL #12 when $sh = 1$
Operation

\[
\begin{align*}
\text{bits(\text{datasize}) result;} \\
\text{bits(\text{datasize}) operand1 = if n == 31 then SP[\text{datasize-1:0}] else X[n, \text{datasize}];} \\
\text{bits(\text{datasize}) operand2;} \\
\text{bits(4) nzcv;} \\
\text{operand2 = NOT(imm);} \\
\text{(result, nzcv) = AddWithCarry(operand1, operand2, '1');} \\
\text{PSTATE.<N,Z,C,V> = nzcv;} \\
\text{X[d, \text{datasize}] = result;}
\end{align*}
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.364 SUBS (shifted register)

Subtract (shifted register), setting flags, subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the aliases CMP (shifted register) and NEGS. See Alias conditions for details of when each alias is preferred.

32-bit variant

Applies when \( sf = 0 \).

\[ \text{SUBS } \langle \text{Wd} \rangle, \langle \text{Wn} \rangle, \langle \text{Wm} \rangle\{, <\text{shift}> \#<\text{amount}>\} \]

64-bit variant

Applies when \( sf = 1 \).

\[ \text{SUBS } \langle \text{Xd} \rangle, \langle \text{Xn} \rangle, \langle \text{Xm} \rangle\{, <\text{shift}> \#<\text{amount}>\} \]

Decode for all variants of this encoding

- \( \text{integer } d = \text{UInt}(Rd); \)
- \( \text{integer } n = \text{UInt}(Rn); \)
- \( \text{integer } m = \text{UInt}(Rm); \)
- \( \text{integer } \text{datasize} = \text{if } sf = '1' \text{ then } 64 \text{ else } 32; \)
- \( \text{if } \text{shift} = '11' \text{ then UNDEFINED}; \)
- \( \text{if } sf = '0' \&\& \text{imm6}<5> = '1' \text{ then UNDEFINED}; \)
- \( \text{ShiftType } \text{shift_type} = \text{DecodeShift}(\text{shift}); \)
- \( \text{integer } \text{shift_amount} = \text{UInt}(\text{imm6}); \)

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP (shifted register)</td>
<td>Rd == '11111'</td>
</tr>
<tr>
<td>NEGS</td>
<td>Rn == '11111' &amp;&amp; Rd != '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

- \( \langle \text{Wd} \rangle \) is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \( \langle \text{Wn} \rangle \) is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \( \langle \text{Wm} \rangle \) is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \( \langle \text{Xd} \rangle \) is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \( \langle \text{Xn} \rangle \) is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \( \langle \text{ Xm} \rangle \) is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift>  Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- **LSL** when shift = 00
- **LSR** when shift = 01
- **ASR** when shift = 10

The encoding shift = 11 is reserved.

<amount>  For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Operation**

```plaintext
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(4) nzcv;
operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = nzcv;
X[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**C6.2.365 SVC**

Supervisor Call causes an exception to be taken to EL1.

On executing an SVC instruction, the PE records the exception as a Supervisor Call exception in ESR_ELx, using the EC value 0x15, and the value of the immediate argument.

```plaintext
| 31 30 29 28 27 26 25 24 23 22 21 20 |   |   | 5 4 3 2 1 0 |
| 1 1 0 1 0 1 0 0 0 0 0 0          | imm16 | 0 0 0 0 1 |
```

**Encoding**

SVC #<imm>

**Decode for this encoding**

// Empty.

**Assembler symbols**

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

**Operation**

AArch64.CheckForSVCTrap(imm16);
AArch64.CallSupervisor(imm16);
C6.2.366  SWPB, SWPAB, SWPALB, SWPLB

Swap byte in memory atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAB and SWPALB load from memory with acquire semantics.
- SWPLB and SWPALB store to memory with release semantics.
- SWPB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

### Integer

(Integer (FEAT_LSE)

| [31 30 29 28 27 26 25 24 23 22 21 20] | 16|15|14|13|12|11|10| 9 | 5 | 4 | 0 |
|---------------------------------------------|---------------------------------|---|---|---|---|---|---|---|---|---|
| 0 0 1 1 0 0 A | R | 1 | Rs | 1 | 0 | 0 | 0 | 0 0 | Rn | Rt |

**SWPAB variant**

Applies when \( A == 1 \&\& R == 0 \).

SWPAB \(<Ws>, <Wt>, [Xn|SP]>\]

**SWPALB variant**

Applies when \( A == 1 \&\& R == 1 \).

SWPALB \(<Ws>, <Wt>, [Xn|SP]>\]

**SWPB variant**

Applies when \( A == 0 \&\& R == 0 \).

SWPB \(<Ws>, <Wt>, [Xn|SP]>\]

**SWPLB variant**

Applies when \( A == 0 \&\& R == 1 \).

SWPLB \(<Ws>, <Wt>, [Xn|SP]>\]

**Decode for all variants of this encoding**

```c
if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);
AccType ldacctype = if A == '1' \&\& Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

**Assembler symbols**

<Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

```c
bits(64) address;
bits(8) data;
bits(8) store_value;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = X[s, 8];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t, 32] = ZeroExtend(data, 32);
```
C6.2.367   SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAH and SWPALH load from memory with acquire semantics.
- SWPLH and SWPALH store to memory with release semantics.
- SWPH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Load-AcquirePC, and Store-Release on page B2-180.

For information about memory accesses see Load/store addressing modes on page C1-234.

Integer

(FEAT_LSE)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 0 0 0 A 1 Rs 1 0 0 0 0 0</td>
<td>Rn Rt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SWPAH variant

Applies when A == 1 && R == 0.

SWPAH <Ws>, <Wt>, [Xn|SP]

SWPALH variant

Applies when A == 1 && R == 1.

SWPALH <Ws>, <Wt>, [Xn|SP]

SWPH variant

Applies when A == 0 && R == 0.

SWPH <Ws>, <Wt>, [Xn|SP]

SWPLH variant

Applies when A == 0 && R == 1.

SWPLH <Ws>, <Wt>, [Xn|SP]

Decode for all variants of this encoding

```c
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```
Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) data;
bits(16) store_value;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = X[s, 16];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t, 32] = ZeroExtend(data, 32);
C6.2.368  **SWP, SWPA, SWPAL, SWPL**

Swap word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, SWPA and SWPAL load from memory with acquire semantics.
- SWP and SWPAL store to memory with release semantics.
- SWP has neither acquire nor release semantics.


For information about memory accesses see *Load/store addressing modes* on page C1-234.

**Integer**

(FEAT_LSE)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x 1 1 1</td>
<td>0 0 0 A R 1</td>
<td>Rs 1 0 0 0 0</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit SWP variant**

Applies when size == 10 && A == 0 && R == 0.

\[
\text{SWP } <W>, <Wt>, [<Xn|SP>]\]

**32-bit SWPA variant**

Applies when size == 10 && A == 1 && R == 0.

\[
\text{SWPA } <W>, <Wt>, [<Xn|SP>]\]

**32-bit SWPAL variant**

Applies when size == 10 && A == 1 && R == 1.

\[
\text{SWPAL } <W>, <Wt>, [<Xn|SP>]\]

**32-bit SWPL variant**

Applies when size == 10 && A == 0 && R == 1.

\[
\text{SWPL } <W>, <Wt>, [<Xn|SP>]\]

**64-bit SWP variant**

Applies when size == 11 && A == 0 && R == 0.

\[
\text{SWP } <X>, <Xt>, [<Xn|SP>]\]

**64-bit SWPA variant**

Applies when size == 11 && A == 1 && R == 0.

\[
\text{SWPA } <X>, <Xt>, [<Xn|SP>]\]

**64-bit SWPAL variant**

Applies when size == 11 && A == 1 && R == 1.
SWPAL \langle Xs \rangle, \langle Xt \rangle, [\langle Xn | SP \rangle]

64-bit SWPL variant

Applies when size == 11 && A == 0 && R == 1.

SWPL \langle Xs \rangle, \langle Xt \rangle, [\langle Xn | SP \rangle]

Decode for all variants of this encoding

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler symbols

\langle Ys \rangle Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
\langle Yt \rangle Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
\langle Xs \rangle Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
\langle Xt \rangle Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
\langle Xn | SP \rangle Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(datasize) data;
bits(datasize) store_value;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

store_value = X[s, datasize];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t, regsize] = ZeroExtend(data, regsize);
C6.2.369  SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to the size of the register, and writes the result to the destination register.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

32-bit variant

Applies when $sf == 0 \&\& N == 0$.

SXTB <Wd>, <Wn>

is equivalent to

SBFM <Wd>, <Wn>, #0, #7

and is always the preferred disassembly.

64-bit variant

Applies when $sf == 1 \&\& N == 1$.

SXTB <Xd>, <Wn>

is equivalent to

SBFM <Xd>, <Xn>, #0, #7

and is always the preferred disassembly.

Assembler symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.370  SXTH

Sign Extend Halfword extracts a 16-bit value, sign-extends it to the size of the register, and writes the result to the destination register.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

32-bit variant

Applies when \( sf == 0 \) \&\& \( N == 0 \).

\[ \text{SXTH} <Wd>, <Wn> \]
is equivalent to

\[ \text{SBFM} <Wd>, <Wn>, #0, #15 \]
and is always the preferred disassembly.

64-bit variant

Applies when \( sf == 1 \) \&\& \( N == 1 \).

\[ \text{SXTH} <Xd>, <Wn> \]
is equivalent to

\[ \text{SBFM} <Xd>, <Xn>, #0, #15 \]
and is always the preferred disassembly.

Assembler symbols

- \(<Wd>\): Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xd>\): Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\): Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<Wn>\): Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.371  SXTW

Sign Extend Word sign-extends a word to the size of the register, and writes the result to the destination register.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

64-bit variant

SXTW <Xd>, <Wn>

is equivalent to

SBFM <Xd>, <Xn>, #0, #31

and is always the preferred disassembly.

Assembler symbols

<Xd>  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn>  Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<Wn>  Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.372  SYS

System instruction. For more information, see `op0==0b01`, cache maintenance, TLB maintenance, address translation, prediction restriction, and BRBE instructions on page C5-659 for the encodings of System instructions.

This instruction is used by the aliases AT, BRB, CFP, CPP, DC, DVP, IC, and TLBI. See Alias conditions for details of when each alias is preferred.

Encoding
SYS #<op1>, <Cn>, <Crn>, #<op2>{, <Xt>}

Decode for this encoding
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT</td>
<td>CRn == '0111' &amp;&amp; CRm == '100x' &amp;&amp; SysOp(op1,'0111',CRm,op2) == Sys_AT</td>
</tr>
<tr>
<td>BRB</td>
<td>op1 == '001' &amp;&amp; CRn == '0111' &amp;&amp; CRm == '0010' &amp;&amp; SysOp('001','0111','0010',op2) == Sys_BRB</td>
</tr>
<tr>
<td>CFP</td>
<td>op1 == '011' &amp;&amp; CRn == '0111' &amp;&amp; CRm == '0011' &amp;&amp; op2 == '100'</td>
</tr>
<tr>
<td>CPP</td>
<td>op1 == '011' &amp;&amp; CRn == '0111' &amp;&amp; CRm == '0011' &amp;&amp; op2 == '111'</td>
</tr>
<tr>
<td>DC</td>
<td>CRn == '0111' &amp;&amp; SysOp(op1,'0111',CRm,op2) == Sys_DC</td>
</tr>
<tr>
<td>DVP</td>
<td>op1 == '011' &amp;&amp; CRn == '0111' &amp;&amp; CRm == '0011' &amp;&amp; op2 == '101'</td>
</tr>
<tr>
<td>IC</td>
<td>CRn == '0111' &amp;&amp; SysOp(op1,'0111',CRm,op2) == Sys_IC</td>
</tr>
<tr>
<td>TLBI</td>
<td>CRn == '100x' &amp;&amp; SysOp(op1,CRn,CRm,op2) == Sys_TLBI</td>
</tr>
</tbody>
</table>

Assembler symbols

<op1>  Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cn>   Is a name 'Crn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

**Operation**

```c
AArch64.SysInstr(1, sys_op1, sys_crn, sys.crm, sys_op2, t);
```
C6.2.373 SYSL

System instruction with result. For more information, see \( op0=0b01 \), cache maintenance, TLB maintenance, address translation, prediction restriction, and BRBE instructions on page C5-659 for the encodings of System instructions.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16 15</th>
<th>12</th>
<th>11</th>
<th>8 7 5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 1</td>
<td>0</td>
<td>1</td>
<td>0 1</td>
<td>op1</td>
<td>CRn</td>
<td>CRm</td>
<td>op2</td>
</tr>
</tbody>
</table>

**Encoding**

SYSL \(<Xt>\), \#<op1>, \(<Cn>\), \(<Cm>\), \#<op2>

**Decode for this encoding**

```c
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);
```

```c
integer t = UInt(Rt);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys.crm = UInt(CRm);
```

**Assembler symbols**

\(<Xt>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

\(<op1>\) Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

\(<Cn>\) Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

\(<Cm>\) Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

\(<op2>\) Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

**Operation**

// No architecturally defined instructions here.

```c
AArch64.SysInstrWithResult(1, sys_op1, sys_crn, sys.crm, sys_op2, t);
```
C6.2.374   TBNZ

Test bit and Branch if Nonzero compares the value of a bit in a general-purpose register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

```
|31 30 29 28 27 26 25 24 23|19 18 | | | 5 4 | 0 |
b5 0 1 1 0 1 1 1 b40 imm14 | | | Rt
```

**Encoding**

TBNZ <R><t>, #<imm>, <label>

**Decode for this encoding**

```c
integer t = UInt(Rt);
integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bits(64) offset = SignExtend(imm14:'00', 64);
```

**Assembler symbols**

- `<R>` Is a width specifier, encoded in the "b5" field. It can have the following values:
  - W when b5 = 0
  - X when b5 = 1
  - In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

- `<t>` Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

- `<imm>` Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

- `<label>` Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

**Operation**

```c
bits(datasize) operand = X[t, datasize];
if operand<bit_pos> == op then
   BranchTo(PC[] + offset, BranchType_DIR, TRUE);
```
C6.2.375  TBZ

Test bit and Branch if Zero compares the value of a test bit with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

Encoding

TBZ $<R><t>$, #<imm>, <label>

Decode for this encoding

integer t = UInt(Rt);
integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bits(64) offset = SignExtend(imm14:'00', 64);

Assembler symbols

$<R>$  Is a width specifier, encoded in the "b5" field. It can have the following values:
   W  when b5 = 0
   X  when b5 = 1

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

$<t>$  Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

$<imm>$  Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

$<label>$  Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

bits(datasize) operand = X[t, datasize];
if operand<bit_pos> == op then
   BranchTo(PC[] + offset, BranchType_DIR, TRUE);
C6.2.376 TCANCEL

This instruction exits Transactional state and discards all state modifications that were performed transactionally. Execution continues at the instruction that follows the TSTART instruction of the outer transaction. The destination register of the TSTART instruction of the outer transaction is written with the immediate operand of TCANCEL.

System
(FEAT_TME)

Encoding
TCANCEL #<imm>

Decode for this encoding
if !HaveTME() then UNDEFINED;
boolean retry = (imm16<15> == '1');
bits(15) reason = imm16<14:0>;

Assembler symbols
<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation
CheckTMEEnabled();
if TSTATE.depth > 0 then
    FailTransaction(TMFailure_CNCL, retry, FALSE, reason);
C6.2.377   TCOMMIT

This instruction commits the current transaction. If the current transaction is an outer transaction, then Transactional
state is exited, and all state modifications performed transactionally are committed to the architectural state.
TCOMMIT takes no inputs and returns no value.

Execution of TCOMMIT is UNDEFINED in Non-transactional state.

System

(FEAT_TME)

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 ]
[1 1 0 1 0 1 0 0 0 0 0 1 1 0 0 0 1 1 0 0 0 0 1 1 1 1 1 1]
```

Encoding

TCOMMIT

Decode for this encoding

if !HaveTME() then UNDEFINED;

Operation

CheckTMEEnabled();

if TSTATE.depth == 0 then
  UNDEFINED;

if TSTATE.depth == 1 then
  CommitTransactionalWrites();
  ClearExclusiveLocal(ProcessorID());
  TSTATE.depth = TSTATE.depth - 1;
C6.2.378 TLBI

TLB Invalidate operation. For more information, see op0==0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, and BRBE instructions on page C5-659.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16</th>
<th>15 12</th>
<th>11 8 7 5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Encoding

TLBI <tlbi_op>{, <Xt>}

is equivalent to

SYS #<op1>, <Crn>, <Cm>, #<op2>{, <Xt>}

and is the preferred disassembly when SysOp(op1,Crn,Crn,op2) == Sys_TLBI.

### Assembler symbols

- `<op1>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- `<Crn>` Is a name 'Crn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- `<Cm>` Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- `<op2>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- `<tlbi_op>` Is a TLBI instruction name, as listed for the TLBI system instruction group, encoded in the "op1:CRn:CRm:op2" field. It can have the following values:
  - VMALLE1IS when op1 = 000, Crn = 1000, Crm = 0011, op2 = 000
  - VAEI1IS when op1 = 000, Crn = 1000, Crm = 0011, op2 = 001
  - ASIDEI1IS when op1 = 000, Crn = 1000, Crm = 0011, op2 = 010
  - VAAE1IS when op1 = 000, Crn = 1000, Crm = 0011, op2 = 011
  - VALEI1IS when op1 = 000, Crn = 1000, Crm = 0011, op2 = 101
  - VAALIE1IS when op1 = 000, Crn = 1000, Crm = 0011, op2 = 111
  - VMALLE1 when op1 = 000, Crn = 1000, Crm = 0111, op2 = 000
  - VAE1 when op1 = 000, Crn = 1000, Crm = 0111, op2 = 001
  - ASIDE1 when op1 = 000, Crn = 1000, Crm = 0111, op2 = 010
  - VAAE1 when op1 = 000, Crn = 1000, Crm = 0111, op2 = 011
  - VALE1 when op1 = 000, Crn = 1000, Crm = 0111, op2 = 101
  - VAALIE1 when op1 = 000, Crn = 1000, Crm = 0111, op2 = 111
  - IPAS2E1IS when op1 = 100, Crn = 1000, Crm = 0000, op2 = 001
  - IPAS2LE1IS when op1 = 100, Crn = 1000, Crm = 0000, op2 = 101
  - ALLE2IS when op1 = 100, Crn = 1000, Crm = 0011, op2 = 000
  - VAEL2IS when op1 = 100, Crn = 1000, Crm = 0011, op2 = 001
C6.2 Alphabetical list of A64 base instructions

When FEAT_TLBIOS is implemented, the following values are also valid:

VMALLEI0S when op1 = 000, Crn = 1000, Crm = 0000, op2 = 100
VAEI0S when op1 = 000, Crn = 1000, Crm = 0001, op2 = 001
ASIDEI0S when op1 = 000, Crn = 1000, Crm = 0001, op2 = 010
VAAE10S when op1 = 000, Crn = 1000, Crm = 0001, op2 = 011
VALE10S when op1 = 000, Crn = 1000, Crm = 0001, op2 = 101
VALE10S when op1 = 000, Crn = 1000, Crm = 0001, op2 = 111
ALLE20S when op1 = 100, Crn = 1000, Crm = 0000, op2 = 000
VAE20S when op1 = 100, Crn = 1000, Crm = 0000, op2 = 001
ALLE10S when op1 = 100, Crn = 1000, Crm = 0001, op2 = 100
VALE20S when op1 = 100, Crn = 1000, Crm = 0001, op2 = 101
VMALLE12E1OS when op1 = 100, Crn = 1000, Crm = 0001, op2 = 110
IPAS2E1OS when op1 = 100, Crn = 1000, Crm = 0100, op2 = 000
IPAS2LE1OS when op1 = 100, Crn = 1000, Crm = 0100, op2 = 010
ALLE30S when op1 = 110, Crn = 1000, Crm = 0001, op2 = 100
VAE30S when op1 = 110, Crn = 1000, Crm = 0001, op2 = 101
VALE30S when op1 = 110, Crn = 1000, Crm = 0001, op2 = 111
VAE3 when op1 = 110, Crn = 1000, Crm = 0000, op2 = 101

When FEAT_TLBIIRANGE is implemented, the following values are also valid:

RVAEE1S when op1 = 000, Crn = 1000, Crm = 0010, op2 = 001
RVAEE1S when op1 = 000, Crn = 1000, Crm = 0010, op2 = 011
RVALE1S when op1 = 000, Crn = 1000, Crm = 0010, op2 = 101
RVALE1S when op1 = 000, Crn = 1000, Crm = 0010, op2 = 111
RVAE10S when op1 = 000, Crn = 1000, Crm = 0101, op2 = 001
RVAE10S when op1 = 000, Crn = 1000, Crm = 0101, op2 = 101
RVALE10S when op1 = 000, Crn = 1000, Crm = 0101, op2 = 111
RVAE when op1 = 000, Crn = 1000, Crm = 0110, op2 = 001
RVAE when op1 = 000, Crn = 1000, Crm = 0110, op2 = 101
C6.2 Alphabetical list of A64 base instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Conditions</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVALE1</td>
<td>when op1 = 000, CRn = 1000, CRm = 0110, op2 = 101</td>
<td></td>
</tr>
<tr>
<td>RVALE1</td>
<td>when op1 = 000, CRn = 1000, CRm = 0110, op2 = 111</td>
<td></td>
</tr>
<tr>
<td>RIPAS2E1IS</td>
<td>when op1 = 100, CRn = 1000, CRm = 0000, op2 = 010</td>
<td></td>
</tr>
<tr>
<td>RIPAS2LE1IS</td>
<td>when op1 = 100, CRn = 1000, CRm = 0000, op2 = 110</td>
<td></td>
</tr>
<tr>
<td>RVAE2IS</td>
<td>when op1 = 100, CRn = 1000, CRm = 0010, op2 = 001</td>
<td></td>
</tr>
<tr>
<td>RVAE2IS</td>
<td>when op1 = 100, CRn = 1000, CRm = 0010, op2 = 101</td>
<td></td>
</tr>
<tr>
<td>RIPAS2E1IS</td>
<td>when op1 = 100, CRn = 1000, CRm = 0100, op2 = 010</td>
<td></td>
</tr>
<tr>
<td>RIPAS2LE1OS</td>
<td>when op1 = 100, CRn = 1000, CRm = 0100, op2 = 011</td>
<td></td>
</tr>
<tr>
<td>RIPAS2LE1OS</td>
<td>when op1 = 100, CRn = 1000, CRm = 0100, op2 = 110</td>
<td></td>
</tr>
<tr>
<td>RVAE20S</td>
<td>when op1 = 100, CRn = 1000, CRm = 0101, op2 = 001</td>
<td></td>
</tr>
<tr>
<td>RVALE20S</td>
<td>when op1 = 100, CRn = 1000, CRm = 0101, op2 = 101</td>
<td></td>
</tr>
<tr>
<td>RVAE2</td>
<td>when op1 = 100, CRn = 1000, CRm = 0110, op2 = 001</td>
<td></td>
</tr>
<tr>
<td>RVAE2</td>
<td>when op1 = 100, CRn = 1000, CRm = 0110, op2 = 101</td>
<td></td>
</tr>
<tr>
<td>RVAE3IS</td>
<td>when op1 = 110, CRn = 1000, CRm = 0010, op2 = 001</td>
<td></td>
</tr>
<tr>
<td>RVAE3IS</td>
<td>when op1 = 110, CRn = 1000, CRm = 0010, op2 = 101</td>
<td></td>
</tr>
<tr>
<td>RVAE30S</td>
<td>when op1 = 110, CRn = 1000, CRm = 0101, op2 = 001</td>
<td></td>
</tr>
<tr>
<td>RVAE30S</td>
<td>when op1 = 110, CRn = 1000, CRm = 0101, op2 = 101</td>
<td></td>
</tr>
<tr>
<td>RVAE3</td>
<td>when op1 = 110, CRn = 1000, CRm = 0110, op2 = 001</td>
<td></td>
</tr>
<tr>
<td>RVAE3</td>
<td>when op1 = 110, CRn = 1000, CRm = 0110, op2 = 101</td>
<td></td>
</tr>
</tbody>
</table>

When FEAT_XS is implemented, the following values are also valid:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Conditions</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMALLE10SNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0001, op2 = 000</td>
<td></td>
</tr>
<tr>
<td>VAES10SNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0001, op2 = 001</td>
<td></td>
</tr>
<tr>
<td>ASIDE10SNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0001, op2 = 010</td>
<td></td>
</tr>
<tr>
<td>VAES10SNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0001, op2 = 011</td>
<td></td>
</tr>
<tr>
<td>VALE10SNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0001, op2 = 101</td>
<td></td>
</tr>
<tr>
<td>VALE10SNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0001, op2 = 111</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0010, op2 = 001</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0010, op2 = 101</td>
<td></td>
</tr>
<tr>
<td>RVALE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0010, op2 = 111</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0011, op2 = 001</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0011, op2 = 010</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0011, op2 = 011</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0011, op2 = 111</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0101, op2 = 001</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0101, op2 = 010</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0101, op2 = 101</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0101, op2 = 111</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0110, op2 = 001</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0110, op2 = 010</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0110, op2 = 101</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0110, op2 = 111</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0110, op2 = 111</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0110, op2 = 111</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0110, op2 = 111</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0110, op2 = 111</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0110, op2 = 111</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0110, op2 = 111</td>
<td></td>
</tr>
<tr>
<td>RVAE1ISNXS</td>
<td>when op1 = 000, CRn = 1001, CRm = 0110, op2 = 111</td>
<td></td>
</tr>
</tbody>
</table>
RVAALE1NXS when op1 = 000, CRn = 1001, CRm = 0110, op2 = 111
VMALE1NXS when op1 = 000, CRn = 1001, CRm = 0111, op2 = 000
VAE1NXS when op1 = 000, CRn = 1001, CRm = 0111, op2 = 001
ASIDE1NXS when op1 = 000, CRn = 1001, CRm = 0111, op2 = 010
VAE1E1NXS when op1 = 000, CRn = 1001, CRm = 0111, op2 = 011
VALE1NXS when op1 = 000, CRn = 1001, CRm = 0111, op2 = 101
VALE1E1NXS when op1 = 000, CRn = 1001, CRm = 0111, op2 = 111
IPAS2E1ISNXS when op1 = 100, CRn = 1001, CRm = 0000, op2 = 001
RIPAS2E1ISNXS when op1 = 100, CRn = 1001, CRm = 0000, op2 = 010
IPAS2E1ISNXS when op1 = 100, CRn = 1001, CRm = 0000, op2 = 011
RIPAS2E1ISNXS when op1 = 100, CRn = 1001, CRm = 0000, op2 = 110
ALLE20SNXS when op1 = 100, CRn = 1001, CRm = 0001, op2 = 000
VAE20SNXS when op1 = 100, CRn = 1001, CRm = 0001, op2 = 001
ALLE10SNXS when op1 = 100, CRn = 1001, CRm = 0001, op2 = 100
VALE20SNXS when op1 = 100, CRn = 1001, CRm = 0001, op2 = 111
VMALLS12E1OSNXS when op1 = 100, CRn = 1001, CRm = 0001, op2 = 110
RVAE21SNXS when op1 = 100, CRn = 1001, CRm = 0010, op2 = 001
RVALE21SNXS when op1 = 100, CRn = 1001, CRm = 0010, op2 = 101
ALLE21SNXS when op1 = 100, CRn = 1001, CRm = 0011, op2 = 000
VAE21SNXS when op1 = 100, CRn = 1001, CRm = 0011, op2 = 001
ALLE11SNXS when op1 = 100, CRn = 1001, CRm = 0011, op2 = 100
VALE21SNXS when op1 = 100, CRn = 1001, CRm = 0011, op2 = 111
VMALLS12E1ISNXS when op1 = 100, CRn = 1001, CRm = 0011, op2 = 110
IPAS2E1OSNXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 000
IPAS2E1NXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 001
RIPAS2E1NXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 010
RIPAS2E1OSNXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 011
IPAS2E1OSNXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 100
IPAS2E1ISNXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 101
RIPAS2E1ISNXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 110
RIPAS2E1OSNXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 111
RVAE20SNXS when op1 = 100, CRn = 1001, CRm = 0101, op2 = 001
RVALE20SNXS when op1 = 100, CRn = 1001, CRm = 0101, op2 = 101
RVAE2SNXS when op1 = 100, CRn = 1001, CRm = 0110, op2 = 001
RVALE2SNXS when op1 = 100, CRn = 1001, CRm = 0110, op2 = 101
ALLE2SNXS when op1 = 100, CRn = 1001, CRm = 0111, op2 = 000
VAE2SNXS when op1 = 100, CRn = 1001, CRm = 0111, op2 = 001
ALLE1NXS when op1 = 100, CRn = 1001, CRm = 0111, op2 = 100
VALE2SNXS when op1 = 100, CRn = 1001, CRm = 0111, op2 = 101
VMALLS12E1NXS when op1 = 100, CRn = 1001, CRm = 0111, op2 = 110
ALLE30SNXS when op1 = 110, CRn = 1001, CRm = 0001, op2 = 000
VAE30SNXS when op1 = 110, CRn = 1001, CRm = 0001, op2 = 001
VALE30SNXS when op1 = 110, CRn = 1001, CRm = 0001, op2 = 101
RVAE31SNXS when op1 = 110, CRn = 1001, CRm = 0010, op2 = 001

C6-2003
A64 Base Instruction Descriptions
C6.2 Alphabetical list of A64 base instructions
RVALE3ISNXS when \( op1 = 110, CRn = 1001, CRm = 0010, op2 = 101 \)
ALLE3ISNXS when \( op1 = 110, CRn = 1001, CRm = 0011, op2 = 000 \)
VAE3ISNXS when \( op1 = 110, CRn = 1001, CRm = 0011, op2 = 001 \)
VALE3ISNXS when \( op1 = 110, CRn = 1001, CRm = 0011, op2 = 101 \)
RVAE3OSNXS when \( op1 = 110, CRn = 1001, CRm = 0101, op2 = 001 \)
RVAE3SNXS when \( op1 = 110, CRn = 1001, CRm = 0101, op2 = 101 \)
RVAE3NXS when \( op1 = 110, CRn = 1001, CRm = 0110, op2 = 001 \)
ALLE3NXS when \( op1 = 110, CRn = 1001, CRm = 0111, op2 = 000 \)
VAE3NXS when \( op1 = 110, CRn = 1001, CRm = 0111, op2 = 001 \)
VAE3NXS when \( op1 = 110, CRn = 1001, CRm = 0111, op2 = 101 \)

\(<Xt>\) Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rx" field.

**Operation**

The description of SYS gives the operational pseudocode for this instruction.
C6.2.379 TSTART

This instruction starts a new transaction. If the transaction started successfully, the destination register is set to zero. If the transaction failed or was canceled, then all state modifications that were performed transactionally are discarded and the destination register is written with a non-zero value that encodes the cause of the failure.

System

(FEAT_TME)

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8 |7 6 5 4 |0  |
| 1 1 0 1 0 1 0 1 0 0 1 0 0 1 1 0 0 1 1 0 0 1 0 0 0 0 0 1 1 | Rt |

Encoding

TSTART <Xt>

Decode for this encoding

if !HaveTME() then UNDEFINED;
integer t =_UINT(Rt);

Assembler symbols

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

Operation

CheckTMEEnabled();

boolean IsEL1Regime;
bit tme;
bit tmt;
case PSTATE_EL of
when EL0
 IsEL1Regime = S1TranslationRegime() == EL1;
 if IsEL1Regime then
 tme = SCTLR_EL1.TME0;
 tmt = SCTLR_EL1.TMT0;
 else
 tme = SCTLR_EL2.TME0;
 tmt = SCTLR_EL2.TMT0;
 when EL1
 tme = SCTLR_EL1.TME;
 tmt = SCTLR_EL1.TMT;
 when EL2
 tme = SCTLR_EL2.TME;
 tmt = SCTLR_EL2.TMT;
 when EL3
 tme = SCTLR_EL3.TME;
 tmt = SCTLR_EL3.TMT;
 otherwise
 Unreachable();

enable = tme == '1';
trivial = tmt == '1';

if !enable then
 TransactionStartTrap(t);
elsif trivial then
 TSTATE.nPC = NextInstrAddr(64);
TSTATE.Rt = t;
FailTransaction(TMFailure_TRIVIAL, FALSE);
elsif HaveSME() && PSTATE.SM == '1' then
  FailTransaction(TMFailure_ERR, FALSE);
elsif TSTATE.depth == 255 then
  FailTransaction(TMFailure_NEST, FALSE);
elif TSTATE.depth == 0 then
  TSTATE.nPC = NextInstrAddr(64);
  TSTATE.Rt = t;
  ClearExclusiveLocal(ProcessorID());
  TakeTransactionCheckpoint();
  StartTrackingTransactionalReadsWrites();
TSTATE.depth = TSTATE.depth + 1;
X[t, 64] = Zeros(64);
C6.2.380 TTEST

This instruction writes the depth of the transaction to the destination register, or the value 0 otherwise.

**System**

(FEAT_TME)

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8|7 6 5 4 |0 |
|-------------|
| 1 1 0 1 0 1 0 0 | 1 0 0 1 1 0 0 1 1| 0 0 0 1 0 1 1 | Rt |

**Encoding**

TTEST <Xt>

**Decode for this encoding**

if !HaveTME() then UNDEFINED;
integer t = UInt(Rt);

**Assembler symbols**

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

**Operation**

CheckTMEEnabled();

X[t, 64] = (TSTATE.depth)<63:0>;
C6.2.381 TSB CSYNC

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions. If FEAT_TRF is not implemented, this instruction executes as a NOP.

System

(FEAT_TRF)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11  8  7  5  4  3  2  1  0 |
| 1 1 0 1 0 1 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 1 0 1 1 1 1 1 |

Encoding

TSB CSYNC

Decode for this encoding

if !HaveSelfHostedTrace() then EndOfInstruction();

Operation

TraceSynchronizationBarrier();
C6.2.382 TST (immediate)

Test bits (immediate), setting the condition flags and discarding the result: \( Rn \ AND \ imm \)

This instruction is an alias of the ANDS (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ANDS (immediate).
- The description of ANDS (immediate) gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when \( sf = 0 \) \&\& \( N = 0 \).

\[
\text{TST} \ <Wn>, \ #<imm>
\]

is equivalent to

\[
\text{ANDS} \ \text{WZR}, \ <Wn>, \ #<imm>
\]

and is always the preferred disassembly.

### 64-bit variant

Applies when \( sf = 1 \).

\[
\text{TST} \ <Xn>, \ #<imm>
\]

is equivalent to

\[
\text{ANDS} \ \text{XZR}, \ <Xn>, \ #<imm>
\]

and is always the preferred disassembly.

### Assembler symbols

- \(<Wn>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<Xn>\) Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<\text{imm}>\) For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
  For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

### Operation

The description of ANDS (immediate) gives the operational pseudocode for this instruction.
C6.2.383 TST (shifted register)

Test (shifted register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the ANDS (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of ANDS (shifted register).
- The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when \( sf == 0 \).

\[
\text{TST} \ <Wn>, <Wm>{, <shift> #<amount>} \]

is equivalent to

\[
\text{ANDS} \ WZR, <Wn>, <Wm>{, <shift> #<amount>} \]

and is always the preferred disassembly.

### 64-bit variant

Applies when \( sf == 1 \).

\[
\text{TST} \ <Xn>, <Xm>{, <shift> #<amount>} \]

is equivalent to

\[
\text{ANDS} \ XZR, <Xn>, <Xm>{, <shift> #<amount>} \]

and is always the preferred disassembly.

### Assembler symbols

- \( <Wn> \) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \( <Wm> \) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \( <Xn> \) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \( < Xm> \) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \( <shift> \) Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:
  - LSL when \( shift = 00 \)
  - LSR when \( shift = 01 \)
  - ASR when \( shift = 10 \)
  - ROR when \( shift = 11 \)
- \( <amount> \) For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
Operation

The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.384 UBFIZ

Unsigned Bitfield Insert in Zeros copies a bitfield of \(<width>\) bits from the least significant bits of the source register to bit position \(<\text{lsb}>\) of the destination register, setting the destination bits above and below the bitfield to zero.

This instruction is an alias of the \textbf{UBFM} instruction. This means that:

- The encodings in this description are named to match the encodings of \textbf{UBFM}.
- The description of \textbf{UBFM} gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when \(sf == 0 \land N == 0\).

\[
\text{UBFIZ } <Wd>, <Wn>, #<\text{lsb}>, #<width>
\]

is equivalent to

\[
\text{UBFM } <Wd>, <Wn>, #(-<\text{lsb} \mod 32)>, #(<width>-1)
\]

and is the preferred disassembly when \(\text{UInt}(\text{immr}) < \text{UInt}(\text{imms})\).

### 64-bit variant

Applies when \(sf == 1 \land N == 1\).

\[
\text{UBFIZ } <Xd>, <Xn>, #<\text{lsb}>, #<width>
\]

is equivalent to

\[
\text{UBFM } <Xd>, <Xn>, #(-<\text{lsb} \mod 64)>, #(<width>-1)
\]

and is the preferred disassembly when \(\text{UInt}(\text{imms}) < \text{UInt}(\text{immr})\).

### Assembler symbols

- \(<Wd>\) is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<Xd>\) is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\) is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<\text{lsb}>\) For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
- \(<width>\) For the 32-bit variant: is the width of the bitfield, in the range 1 to 32.<\text{lsb}>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64.<\text{lsb}>.

### Operation

The description of \textbf{UBFM} gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.385  UBFM

Unsigned Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If \(<\text{imms}>\) is greater than or equal to \(<\text{immr}>\), this copies a bitfield of \((<\text{imms}>-<\text{immr}>+1)\) bits starting from bit position \(<\text{immr}>\) in the source register to the least significant bits of the destination register.

If \(<\text{imms}>\) is less than \(<\text{immr}>\), this copies a bitfield of \((<\text{imms}>+1)\) bits from the least significant bits of the source register to bit position \((\text{regsize}-<\text{immr}>)\) of the destination register, where \(\text{regsize}\) is the destination register size of 32 or 64 bits.

In both cases the destination bits below and above the bitfield are set to zero.

This instruction is used by the aliases LSL (immediate), LSR (immediate), UBFIZ, UBFX, UXTB, and UXTH. See Alias conditions on page C6-2015 for details of when each alias is preferred.

32-bit variant

Applies when \(s\ f == 0 \&\& N == 0\).

\[\text{UBFM } <\text{Wd}>, <\text{Wn}>, #<\text{immr}>, #<\text{imms}>\]

64-bit variant

Applies when \(s\ f == 1 \&\& N == 1\).

\[\text{UBFM } <\text{Xd}>, <\text{Xn}>, #<\text{immr}>, #<\text{imms}>\]

Decode for all variants of this encoding

integer \(d\) = UInt(Rd);
integer \(n\) = UInt(Rn);
integer \(\text{datasize}\) = if \(s\ f == '1'\) then 64 else 32;

integer \(r\);
bits(datasize) \(wmask\);
bits(datasize) \(tmask\);

if \(s\ f == '1' \&\& N != '1'\) then UNDEFINED;
if \(s\ f == '0' \&\& (N != '0' || immr<5> != '0' || imms<5> != '0')\) then UNDEFINED;

\(r\) = UInt(immr);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>of variant</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL (immediate)</td>
<td>32-bit</td>
<td>imms != '011111' &amp; imms + 1 == immr</td>
</tr>
<tr>
<td>LSL (immediate)</td>
<td>64-bit</td>
<td>imms != '111111' &amp; imms + 1 == immr</td>
</tr>
<tr>
<td>LSR (immediate)</td>
<td>32-bit</td>
<td>imms == '011111'</td>
</tr>
<tr>
<td>LSR (immediate)</td>
<td>64-bit</td>
<td>imms == '111111'</td>
</tr>
<tr>
<td>UBFIZ</td>
<td>-</td>
<td>UInt(imms) &lt; UInt(immr)</td>
</tr>
<tr>
<td>UBFX</td>
<td>-</td>
<td>BFXPreferred(sf, opc&lt;1&gt;, imms, immr)</td>
</tr>
<tr>
<td>UXTB</td>
<td>-</td>
<td>immr == '000000' &amp; imms == '000111'</td>
</tr>
<tr>
<td>UXTH</td>
<td>-</td>
<td>immr == '000000' &amp; imms == '001111'</td>
</tr>
</tbody>
</table>

Assembler symbols

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<imm>`: For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.
- `<imms>`: For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.
- `<imm>`: For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
- `<imms>`: For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Operation

bits(datasize) src = X[n, datasize];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, r) AND wmask;

// combine extension bits and result bits
X[d, datasize] = bot AND tmask;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
— The values of the NZCV flags.
C6.2.386  UBFX

Unsigned Bitfield Extract copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to zero.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

### 32-bit variant

Applies when \( sf == 0 \&\& N == 0 \).

\[
\text{UBFX} \ <Wd>, \ <Wn>, \ #<lsb>, \ #<width>
\]

is equivalent to

\[
\text{UBFM} \ <Wd>, \ <Wn>, \ #<lsb>, \ #(<lsb>+<width>-1)
\]

and is the preferred disassembly when \( \text{BFXPreferred}(sf, \ opc<1>, \ imms, \ immr) \).

### 64-bit variant

Applies when \( sf == 1 \&\& N == 1 \).

\[
\text{UBFX} \ <Xd>, \ <Xn>, \ #<lsb>, \ #<width>
\]

is equivalent to

\[
\text{UBFM} \ <Xd>, \ <Xn>, \ #<lsb>, \ #(<lsb>+<width>-1)
\]

and is the preferred disassembly when \( \text{BFXPreferred}(sf, \ opc<1>, \ imms, \ immr) \).

### Assembler symbols

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\) Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<lsb>\) For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.
  For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
- \(<width>\) For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.
  For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

### Operation

The description of UBFM gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.387  UDF

Permanently Undefined generates an Undefined Instruction exception (ESR_ELx.EC = 0b000000). The encodings for UDF used in this section are defined as permanently UNDEFINED.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15</th>
<th>15</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>imm16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

UDF #<imm>

**Decode for this encoding**

// The imm16 field is ignored by hardware.

UNDEFINED;

**Assembler symbols**

<imm> is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field. The PE ignores the value of this constant.

**Operation**

// No operation.
C6.2.388   UDIV

Unsigned Divide divides an unsigned integer register value by another unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

32-bit variant
Applies when \(sf == 0\).
UDIV <Wd>, <Wn>, <Wm>

64-bit variant
Applies when \(sf == 1\).
UDIV <Xd>, <Xn>, < Xm>

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Wm>\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Xm>\) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
integer result;

if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, TRUE)) / Real(Int(operand2, TRUE)));
X[d, datasize] = result<datasize-1:0>;
```
C6.2.389 UMADDL

Unsigned Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMULL. See Alias conditions for details of when each alias is preferred.

Encoding

UMADDL <Xd>, <Wn>, <Wm>, <Xa>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMULL</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

bits(32) operand1 = X[n, 32];
bits(32) operand2 = X[m, 32];
bits(64) operand3 = X[a, 64];

integer result;

result = Int(operand3, TRUE) + (Int(operand1, TRUE) * Int(operand2, TRUE));

X[d, 64] = result<63:0>;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.390 UMNEGL

Unsigned Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This instruction is an alias of the UMSUBL instruction. This means that:

- The encodings in this description are named to match the encodings of UMSUBL.
- The description of UMSUBL gives the operational pseudocode for this instruction.

**Encoding**

UMNEGL <Xd>, <Wn>, <Wm>

is equivalent to

UMSUBL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

**Assembler symbols**

- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

**Operation**

The description of UMSUBL gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.391 UMSUBL

Unsigned Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMNEGL. See Alias conditions for details of when each alias is preferred.

Encoding

UMSUBL <Xd>, <Wn>, <Wm>, <Xa>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMNEGL</td>
<td>Ra == '1111'</td>
</tr>
</tbody>
</table>

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Operation

bits(32) operand1 = X[n, 32];
bits(32) operand2 = X[m, 32];
bits(64) operand3 = X[a, 64];

integer result;
result = Int(operand3, TRUE) - (Int(operand1, TRUE) * Int(operand2, TRUE));
X[d, 64] = result<63:0>;
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C6.2.392 UMULH

Unsigned Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

Encoding

UMULH <Xd>, <Xn>, <Xm>

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler symbols

- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Xm>` is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
bits(64) operand1 = X[n, 64];
bits(64) operand2 = X[m, 64];

integer result;

result = Int(operand1, TRUE) * Int(operand2, TRUE);
X[d, 64] = result<127:64>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.393  UMULL

Unsigned Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This instruction is an alias of the UMADDL instruction. This means that:

- The encodings in this description are named to match the encodings of UMADDL.
- The description of UMADDL gives the operational pseudocode for this instruction.

Encoding

UMULL <Xd>, <Wn>, <Wm>

is equivalent to

UMADDL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler symbols

<Xd>  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn>  Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm>  Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of UMADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.394 UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

### 32-bit variant

\[
\text{UXTB} \ <Wd>, \ <Wn>
\]

is equivalent to

\[
\text{UBFM} \ <Wd>, \ <Wn>, \ #0, \ #7
\]

and is always the preferred disassembly.

#### Assembler symbols

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

#### Operation

The description of UBFM gives the operational pseudocode for this instruction.

#### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.395 UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

### 32-bit variant

UXTH <\#d>, <\#n>

is equivalent to

UBFM <\#d>, <\#n>, #0, #15

and is always the preferred disassembly.

#### Assembler symbols

- \(<\#d>\) is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<\#n>\) is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

#### Operation

The description of UBFM gives the operational pseudocode for this instruction.

#### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C6.2.396   WFE

Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a
wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any
PE in the multiprocessor system. For more information, see Wait for Event on page D1-4676.

As described in Wait for Event on page D1-4676, the execution of a WFE instruction that would otherwise cause entry
to a low-power state can be trapped to a higher Exception level.

Encoding

WFE

Decode for this encoding

// Empty.

Operation

integer localtimeout = 1 << 64;    // No local timeout event is generated
Hint_WFE(localtimeout, WFxType_WFE);
C6.2.397 WFET

Wait For Event with Timeout is a hint instruction that indicates that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any PE in the multiprocessor system. For more information, see *Wait for Event* on page D1-4676.

As described in *Wait for Event* on page D1-4676, the execution of a WFET instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

**System**

(FEAT_WFxT)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

WFET <Xt>

**Decode for this encoding**

if !HaveFeatWFxT() then UNDEFINED;

integer d = UInt(Rd);

**Assembler symbols**

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rd" field.

**Operation**

integer localtimeout = UInt(X[d, 64]);

if Halted() && ConstrantUnpredictableBool(Unpredictable_WFxTDEBUG) then EndOfInstruction();

Hint_WFE(localtimeout, WFxType_WFET);
C6.2.398 WFI

Wait For Interrupt is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see Wait for Interrupt mechanism on page D1-4678.

As described in Wait for Interrupt mechanism on page D1-4678, the execution of a WFI instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

Encoding

WFI

Decode for this encoding

// Empty.

Operation

```
integer localtimeout = 1 << 64; // No local timeout event is generated
Hint_WFI(localtimeout, WFxType_WFI);
```
C6.2.399 WFIT

Wait For Interrupt with Timeout is a hint instruction that indicates that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. For more information, see Wait for Interrupt mechanism on page D1-4678.

As described in Wait for Interrupt mechanism on page D1-4678, the execution of a WFIT instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

System

( FEAT_WFxT )

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 1 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

WFIT <Xt>

Decode for this encoding

if !HaveFeatWFxT() then UNDEFINED;
integer d = UInt(Rd);

Assembler symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rd" field.

Operation

integer localtimeout = UInt(X[d, 64]);
if Halted() & & ConstrainUnpredictableBool(Unpredictable_WFxTDEBUG) then
    EndOfInstruction();
Hint_WFI(localtimeout, WFxType_WFIT);
C6.2.400  XAFLAG

Convert floating-point condition flags from external format to Arm format. This instruction converts the state of the PSTATE.\{N,Z,C,V\} flags from an alternative representation required by some software to a form representing the result of an Arm floating-point scalar compare instruction.

System

(FEAT_FlagM2)

\[
\begin{array}{cccccccccccccccccccccc}
\hline
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1
\end{array}
\]

Encoding

XAFLAG

Decode for this encoding

if !HaveFlagFormatExt() then UNDEFINED;

Operation

\[
\begin{align*}
\text{bit } n &= \text{NOT(PSTATE.C) AND NOT(PSTATE.Z);} \\
\text{bit } z &= \text{PSTATE.Z AND PSTATE.C;} \\
\text{bit } c &= \text{PSTATE.C OR PSTATE.Z;} \\
\text{bit } v &= \text{NOT(PSTATE.C) AND PSTATE.Z;} \\
\text{PSTATE.N} &= n; \\
\text{PSTATE.Z} &= z; \\
\text{PSTATE.C} &= c; \\
\text{PSTATE.V} &= v;
\end{align*}
\]
C6.2.401 XPACD, XPARI, XPARLR

Strip Pointer Authentication Code. This instruction removes the pointer authentication code from an address. The address is in the specified general-purpose register for XPARI and XPARD, and is in LR for XPARLR.

The XPARD instruction is used for data addresses, and XPARI and XPARLR are used for instruction addresses.

**Integer**

(Feat_PAuth)

```asm
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4  | 0 |  
| 1 1 0 | 1 1 0 1 1 0 | 0 0 0 0 0 1 0 0 0 0 0 | 1 0 0 0 0 0 1 1 1 1 | Rd  
```

**XPACD variant**

Applies when \( D = 1 \).

XPACD <Xd>

**XPACI variant**

Applies when \( D = 0 \).

XPACI <Xd>

**Decode for all variants of this encoding**

```cpp
boolean data = (D == '1');
integer d = UInt(Rd);
if !HavePACExt() then
  UNDEFINED;
```

**System**

(Feat_PAuth)

```asm
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 8 7 6 5 4| 3 2 1 0 |  
| 1 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 1 1 1 |  
```

**Encoding**

XPACLRI

**Decode for this encoding**

```cpp
integer d = 30;
boolean data = FALSE;
```

**Assembler symbols**

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
Operation for all encodings

if HavePACExt() then
    X[d, 64] = Strip(X[d, 64], data);
C6.2.402  YIELD

YIELD is a hint instruction. Software with a multithreading capability can use a YIELD instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction, see The YIELD instruction on page B1-146.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 | 8 7 6 5 4 3 2 1 0 |
|---------------|-----------------|--------|
| 1 1 0 1 0 1 0 0 | 0 0 0 0 1 1 0 0 | 0 0 0 0 1 1 1 1 |
```

**Encoding**

YIELD

**Decode for this encoding**

// Empty.

**Operation**

`hint_yield();`
This chapter describes the A64 Advanced SIMD and floating-point instructions.

It contains the following sections:

- About the SME instructions on page C9-4542.
- Alphabetical list of A64 Advanced SIMD and floating-point instructions on page C7-2042.
C7.1 About the A64 Advanced SIMD and floating-point instructions

Alphabetical list of A64 Advanced SIMD and floating-point instructions on page C7-2042 gives full descriptions of the A64 instructions that are in the following instruction groups:

- Loads and store instructions associated with the SIMD and floating-point registers.
- Data processing instructions with SIMD and floating-point registers.

A64 instruction set encoding on page C4-388 in the A64 Instruction Encodings chapter provides an overview of the instruction encodings as part of an instruction class within a functional group.

The rest of this section is a general description of the SIMD and floating-point instructions. It contains the following subsections:

- Register size.
- Output element control.
- Data types on page C7-2041.
- Condition flags and related instructions on page C7-2041.

C7.1.1 Register size

A64 provides a comprehensive set of packed Single Instruction Multiple Data (SIMD) and scalar operations using data held in the 32 entry 128-bit wide SIMD and floating-point register file.

Each SIMD and floating-point register can be used to hold:

- A single scalar value of the floating-point or integer type.
- A 64-bit wide vector containing one or more elements.
- A 128-bit wide vector containing two or more elements.

Where the entire 128-bit wide register is not fully utilized, the vector or scalar quantity is held in the least significant bits of the register, with the most significant bits being cleared to zero on a write, see Advanced SIMD vector formats on page A1-43.

The following instructions can insert data into individual elements within a SIMD and floating-pointer register without clearing the remaining bits to zero:

- Insert vector element from another vector element or general-purpose register, INS.
- Load structure into a single lane, for example LD3.
- All second-part narrowing operations, for example SHRN2.

C7.1.2 Output element control

When FEAT_AFP is implemented, the FPCR.NEP bit controls how output elements are determined for the scalar Advanced SIMD instructions for elements other than the lowest element of the vector.

If FPCR.NEP == 1, the following instructions determine output elements as follows:

- The 3-input floating-point scalar versions of FMLA (by element) and FMLS (by element) take output elements other than the lowest element from the <Hd>, <Sd>, or <Dd> register.
- The 3-input floating-point FMADD, FMSUB, FNADD, and FNMSUB instructions take output elements other than the lowest element from the <Ha>, <Sa>, or <Da> register.
- The 2-input floating-point scalar versions of FCMGE (register), FCMGT (register), FCMEQ (register), FACGE, FACGT, take output elements other than the lowest element from the <Hm>, <Sm>, or <Dm> register.
- The 2-input floating-point scalar versions of FMULX, FRECPS, FRSQRTS, FABD, FMUL (by element), FMUL (scalar), FDIV (scalar), FADD (scalar), FSUB (scalar), FMAX (scalar), FMIN (scalar), FMAXNM (scalar), FMINNM (scalar), FMUL (scalar), take output elements other than the lowest element from the <Hn>, <Sn>, or <Dn> register.
- For 1-input floating-point scalar versions of the instructions FCVTNS (vector), FCVTMS (vector), FCVTAS (vector), FCVTPS (vector), SCVT (vector, integer), UCVT (vector, integer) FCVTZS (vector, integer), FCVTZU (vector, integer), FCVTVU (vector), FCVTMU (vector), FCVTAU (vector), FCVTU (vector), SCVT (vector, fixed-point), UCVT (vector, fixed-point), FCVTZ (vector, fixed-point), FCVTZU
C7.1 About the A64 Advanced SIMD and floating-point instructions

C7.1.3 Data types

The A64 instruction set provides support for arithmetic, conversion, and bitwise operations on:

- Half-precision, single-precision, and double-precision floating-points.
- Signed and unsigned integers.
- Polynomials over \{0, 1\}.
- When FEAT_FCMA is implemented, complex numbers.

For all AArch64 floating-point operations, including SIMD operations, the rounding mode and exception trap handling are controlled by the FPCR.

--- Note ---

- AArch32 Advanced SIMD operations always use Arm standard floating-point arithmetic, regardless of the rounding mode specified by the AArch64 FPCR or the AArch32 FPSCR.
- In AArch64 state, floating-point multiply-add operations are always performed as fused operations, but AArch32 state provides both fused and chained multiply-add instructions.

In addition to operations that consume and produce values of the same width and type, the A64 instruction set supports SIMD and scalar operations that produce a wider or narrower vector result:

- Where a SIMD operation narrows a 128-bit vector to a 64-bit vector, the A64 instruction set provides a second-part operation, for example SHRN2, that can pack the result of a second operation into the upper part of the same destination register.
- Where a SIMD operation widens a 64-bit vector to a 128-bit vector, the A64 instruction set provides a second-part operation, for example SMLAL2, that can extract the source from the upper 64 bits of the source registers.

All SIMD operations that could produce side-effects that are not limited to the destination SIMD and floating-point register, for example a potential update of FPSR.Q or FPSR.IDC, have a dedicated scalar variant to support the use of SIMD with loops requiring specialized head or tail handling, or both.

C7.1.4 Condition flags and related instructions

The A64 instruction set provides support for flag setting and conditional operations on the SIMD and floating-point register file:

- Floating-point FCSEL and FCOMP instructions are equivalent to the integer CSEL and CCMP instructions.
- Floating-point FCMP, FCMPE, FCMPO, and FCMPO instructions set the PSTATE.{N, Z, C, V} flags based on the result of the floating-point comparison.
- Floating-point F3CVTZS instruction sets the PSTATE.Z flag if the result of the conversion, when converted back to a double-precision floating-point number, gives precisely the same value as the original. Other PSTATE flags are cleared by this instruction.
- Floating-point and integer instructions provide a means of producing either a scalar or a vector mask based on a comparison in a SIMD and floating-point register, for example FCMPZ.

--- Note ---

FCMP and FCMPE differ from the A32/T32 VCM and VCMPE instructions, which use the dedicated FPSCR.NZCV field for the result. A64 instructions store the result of an FCMP or FCMPE operation in the PSTATE.{N, Z, C, V} field.

If FEAT_FlagM2 is implemented, base instructions XAFLAG and AXFLAG convert between the PSTATE condition flag format used by the FCMP instruction and an alternative format. See Table C6-1 on page C6-1168.
C7.2 Alphabetical list of A64 Advanced SIMD and floating-point instructions

This section lists every section in the Advanced SIMD and floating-point categories of the A64 instruction set. For details of the format used, see Structure of the A64 assembler language on page C1-228.
C7.2.1 ABS

Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

Encoding

ABS \langle V \rangle \langle d \rangle, \langle V \rangle \langle n \rangle

Decode for this encoding

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd) ; \\
\text{integer } n &= \text{UInt}(Rn) ; \\
\text{if } \text{size} \neq '11' \text{ then UNDEFINED;} \\
\text{integer } \text{esize} &= 8 \ll \text{UInt(size)}; \\
\text{integer } \text{datasize} &= \text{esize}; \\
\text{integer } \text{elements} &= 1; \\
\text{boolean } \text{neg} &= (U == '1');
\end{align*}
\]

**Vector**

Encoding

ABS \langle Vd \rangle \langle T \rangle, \langle Vn \rangle \langle T \rangle

Decode for this encoding

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd) ; \\
\text{integer } n &= \text{UInt}(Rn) ; \\
\text{if } \text{size}:Q \neq '110' \text{ then UNDEFINED;} \\
\text{integer } \text{esize} &= 8 \ll \text{UInt(size)}; \\
\text{integer } \text{datasize} &= \text{if } Q == '1' \text{ then 128 else 64}; \\
\text{integer } \text{elements} &= \text{datasize} \div \text{esize}; \\
\text{boolean } \text{neg} &= (U == '1');
\end{align*}
\]

Assembler symbols

\[
\langle V \rangle \text{ Is a width specifier, encoded in the } "\text{size}" \text{ field. It can have the following values:}
\]

\[
\begin{align*}
D & \text{ when } \text{size} = 11 \\
\text{The following encodings are reserved:}
\end{align*}
\]

- size = 0x.
• size = 10.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

<table>
<thead>
<tr>
<th>T</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8B</td>
<td>when size = 00, Q = 0</td>
</tr>
<tr>
<td>16B</td>
<td>when size = 00, Q = 1</td>
</tr>
<tr>
<td>4H</td>
<td>when size = 01, Q = 0</td>
</tr>
<tr>
<td>8H</td>
<td>when size = 01, Q = 1</td>
</tr>
<tr>
<td>2S</td>
<td>when size = 10, Q = 0</td>
</tr>
<tr>
<td>4S</td>
<td>when size = 10, Q = 1</td>
</tr>
<tr>
<td>2D</td>
<td>when size = 11, Q = 1</td>
</tr>
</tbody>
</table>

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
for e = 0 to elements-1
   element = SInt(Elem[operand, e, esize]);
   if neg then
      element = -element;
   else
      element = Abs(element);
   Elem[result, e, esize] = element<esize-1:0>;
V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.2  ADD (vector)

Add (vector). This instruction adds corresponding elements in the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

```
0   1   0   1   1   1   0 | size  1 | Rm   1   0   0   0   1 | Rn   Rd
```

Encoding

ADD <V><d>, <V><n>, <V><m>

Decode for this encoding

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');
```

Vector

```
0   Q   0   0   1   1   1   0 | size  1 | Rm   1   0   0   0   1 | Rn   Rd
```

Encoding

ADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');
```

Assembler symbols

&lt;V&gt;  Is a width specifier, encoded in the "size" field. It can have the following values:

D    when size = 11

The following encodings are reserved:

•    size = 0x.
• size = 10.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8B</td>
<td>size = 00, Q = 0</td>
</tr>
<tr>
<td>16B</td>
<td>size = 00, Q = 1</td>
</tr>
<tr>
<td>4H</td>
<td>size = 01, Q = 0</td>
</tr>
<tr>
<td>8H</td>
<td>size = 01, Q = 1</td>
</tr>
<tr>
<td>2S</td>
<td>size = 10, Q = 0</td>
</tr>
<tr>
<td>4S</td>
<td>size = 10, Q = 1</td>
</tr>
<tr>
<td>2D</td>
<td>size = 11, Q = 1</td>
</tr>
</tbody>
</table>

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;
V[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.3 ADDHN, ADDHN2

Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are truncated. For rounded results, see RADDHN, RADDHN2.

The ADDHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the ADDHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

ADDHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0
[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bias(2*datasize) operand2 = V[m, 2*datasize];
bias(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bias(2*esize) element1;
bias(2*esize) element2;
bias(2*esize) sum;
for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;
Vpart[d, part, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.4   ADDP (scalar)

Add Pair of elements (scalar). This instruction adds two vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

ADDP <V><d>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize * 2;

Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:
  0 when size = 11
  The following encodings are reserved:
  • size = 0x.
  • size = 10.
<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<T> Is the source arrangement specifier, encoded in the "size" field. It can have the following values:
  2D when size = 11
  The following encodings are reserved:
  • size = 0x.
  • size = 10.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_ADD, operand, esize);
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.5   ADDP (vector)

Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0  
0 Q 0 1 1 1 0 | size 1 | Rm 1 0 1 1 1 1 | Rn Rd
```

**Encoding**

ADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

**Assembler symbols**

<\Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
```

CheckFPAdvSIMDEnabled64();
for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
    Elem[result, e, esize] = element1 + element2;

V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.6   ADDV

Add across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes
the scalar result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

Encoding
ADDV <V><d>, <Vn>.<T>

Decode for this encoding
integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

Assembler symbols
<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:
8 when size = 00
H when size = 01
S when size = 10
The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
8B when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
4S when size = 10, Q = 1
The following encodings are reserved:
• size = 10, Q = 0.
• size = 11, Q = x.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_ADD, operand, esize);

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.7 AESD

AES single round decryption.

**Advanced SIMD**

(Feat_AES)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 1 1 0</td>
<td>0 0 1 0 1 0 0 0</td>
<td>1 0 1</td>
</tr>
</tbody>
</table>

Rn Rd

**Encoding**

AESD <Vd>.16B, <Vn>.16B

**Decode for this encoding**

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;
```

**Assembler symbols**

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) result;
result = operand1 EOR operand2;
result = AESInvSubBytes(AESInvShiftRows(result));
V[d, 128] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
### C7.2.8 AESE

AES single round encryption.

#### Advanced SIMD

(FEAT_AES)

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 |
| 0 1 0 0 1 1 1 0 | 0 0 1 0 1 0 0 | 0 0 1 0 | 0 1 0 |
| Rn | Rd |
```

#### Encoding

AESE <Vd>.16b, <Vn>.16b

#### Decode for this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;
```

#### Assembler symbols

- `<Vd>` is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- `<Vn>` is the name of the second SIMD&FP source register, encoded in the "Rn" field.

#### Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) result;
result = operand1 EOR operand2;
result = AESSubBytes(AESShiftRows(result));
V[d, 128] = result;
```

#### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.9 **AESIMC**

AES inverse mix columns.

\[
\begin{array}{cccccccccccccccccccccc}
\end{array}
\]

| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |

*Encoding*

AESIMC <Vd>.16B, <Vn>.16B

*Decode for this encoding*

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd) ; \\
\text{integer } n &= \text{UInt}(Rn) ; \\
\text{if } \neg \text{HaveAESExt}() \text{ then UNDEFINED;}
\end{align*}
\]

*Assembler symbols*

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

*Operation*

\[
\begin{align*}
\text{AArch64.CheckFPAvSIMDEnabled}() ; \\
\text{bits(128) operand } &= V[n, 128] ; \\
\text{bits(128) result} ; \\
\text{result } &= \text{AESInvMixColumns}(\text{operand}) ; \\
V[d, 128] &= \text{result} ;
\end{align*}
\]

*Operational information*

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.10 AESMC

AES mix columns.

Encoding
AESMC <Vd>.16B, <Vn>.16B

Decode for this encoding

\[ \text{integer } d = \text{UInt}(Rd); \]
\[ \text{integer } n = \text{UInt}(Rn); \]
\[ \text{if } \text{!HaveAESExt()} \text{ then UNDEFINED;} \]

Assembler symbols

\[ <Vd> \quad \text{Is the name of the SIMD&FP destination register, encoded in the } "Rd" \text{ field.} \]
\[ <Vn> \quad \text{Is the name of the SIMD&FP source register, encoded in the } "Rn" \text{ field.} \]

Operation

\[ \text{AArch64.CheckFPAdvSIMDEnabled();} \]
\[ \text{bits}(128) \text{ operand} = V[n, 128]; \]
\[ \text{bits}(128) \text{ result}; \]
\[ \text{result} = \text{AESMixColumns}(\text{operand}); \]
\[ V[d, 128] = \text{result}; \]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.11   AND (vector)

Bitwise AND (vector). This instruction performs a bitwise AND between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
[31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 | 15 14 13 12 | 11 10 9 | 8 7 6 5 | 4 3 2 1 | 0 | 0 0 0 0 1 1 $0$ | 0 0 1 1 1 Rm | 0 0 0 1 1 1 Rn | Rd
```

Encoding

```
AND <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler symbols

```
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  8B when Q = 0
  16B when Q = 1
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
```

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
result = operand1 AND operand2;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.12  BCAX

Bit Clear and Exclusive OR performs a bitwise AND of the 128-bit vector in a source SIMD&FP register and the complement of the vector in another source SIMD&FP register, then performs a bitwise exclusive OR of the resulting vector and the vector in a third source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SHA3 is implemented.

Advanced SIMD

(FEAT_SHA3)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1 1 1 0</td>
<td>0 1</td>
<td>Rm</td>
<td>0</td>
<td>Ra</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
</tr>
</tbody>
</table>

Encoding

BCAX <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B

Decode for this encoding

if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Va> Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Va = V[a, 128];
V[d, 128] = Vn EOR (Vm AND NOT(Va));

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.13 BFCVT

Floating-point convert from single-precision to BFloat16 format (scalar) converts the single-precision floating-point value in the 32-bit SIMD&FP source register to BFloat16 format and writes the result in the 16-bit SIMD&FP destination register.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Single-precision to BFloat16

(FCAT_BF16)

```
|31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10| 9 | 5 | 4 | 0 |
|0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | Rn | Rd |
```

**Encoding**

BFCVT <Hd>, <Sn>

**Decode for this encoding**

```c
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
```

**Assembler symbols**

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPEnabled64();

bits(32) operand = V[n, 32];
FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

Elem[result, 0, 16] = FPConvertBF(operand, fpcr);
V[d, 128] = result;
```
C7.2.14 BFCVTN, BFCVTN2

Floating-point convert from single-precision to BFloat16 format (vector) reads each single-precision element in the SIMD&FP source vector, converts each value to BFloat16 format, and writes the results in the lower or upper half of the SIMD&FP destination vector. The result elements are half the width of the source elements.

The BFCVTN instruction writes the half-width results to the lower half of the destination vector and clears the upper half to zero, while the BFCVTN2 instruction writes the results to the upper half of the destination vector without affecting the other bits in the register.

**Vector single-precision to BFloat16**

(FEAT_BF16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Encoding**

BFCVTN(2) <Vd>,<Ta>, <Vn>.4S

**Decode for this encoding**

if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer part = UInt(Q);
integer elements = 64 DIV 16;

**Assembler symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

-absent- when Q = 0
-present- when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

-4H- when Q = 0
-8H- when Q = 1

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

CheckFPAdvSIMDEnabled64();
bits(128) operand = V[n, 128];
bits(64) result;

for e = 0 to elements-1
    Elem[operand, e, 32] = FPConvertBF(Elem[operand, e, 32], FPCR[]);
Vpart[d, part, 64] = result;
C7.2.15 BFDOT (by element)

BFloat16 floating-point dot product (vector, by element). This instruction delimits the source vectors into pairs of BFloat16 elements.

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, irrespective of the other control bits in the FPCR, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second source vector. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is \{1, 1\}.
- Generates only the default NaN, as if FPCR.DN is 1.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second source vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Generates only the default NaN, as if FPCR.DN is 1.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as controlled by the effective value of the FPCR in the current execution mode, and captured in the FPSR.

The BFloat16 pair within the second source vector is specified using an immediate index. The index range is from 0 to 3 inclusive. ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

**Vector**

(FEAT_BF16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Encoding**

BFDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.2H[index]\

**Decode for this encoding**

if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
2S when Q = 0
4S when Q = 1
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
4H when Q = 0
8H when Q = 1
<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
<index> Is the immediate index of a pair of 16-bit elements in the range 0 to 3, encoded in the "H:L" fields.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(128) operand2 = V[m, 128];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2*e+0, 16];
    bits(16) elt1_b = Elem[operand1, 2*e+1, 16];
    bits(16) elt2_a = Elem[operand2, 2*i+0, 16];
    bits(16) elt2_b = Elem[operand2, 2*i+1, 16];
    bits(32) sum = Elem[operand3, e, 32];
    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

V[d, datasize] = result;
```
C7.2.16 BFDOT (vector)

BFloat16 floating-point dot product (vector). This instruction delimits the source vectors into pairs of BFloat16 elements.

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, irrespective of the other control bits in the FPCR, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Generates only the default NaN, as if FPCR.DN is 1.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Generates only the default NaN, as if FPCR.DN is 1.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as controlled by the effective value of the FPCR in the current execution mode, and captured in the FPSR.

Vector

(FEAT_BF16)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Rm

Rn

Rd

Encoding

BFDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

Decode for this encoding

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler symbols

<vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
- 2S when Q = 0
- 4S when Q = 1

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
- 4H when Q = 0
- 8H when Q = 1

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2*e+0, 16];
    bits(16) elt1_b = Elem[operand1, 2*e+1, 16];
    bits(16) elt2_a = Elem[operand2, 2*e+0, 16];
    bits(16) elt2_b = Elem[operand2, 2*e+1, 16];
    bits(32) sum = Elem[operand3, e, 32];
    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;
V[d, datasize] = result;
```

C7.2.17  BFMLALB, BFMLALT (by element)

BFLOAT16 floating-point widening multiply-add long (by element) widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first source vector, and the indexed element in the second source vector from BFLOAT16 to single-precision format. The instruction then multiplies and adds these values without intermediate rounding to single-precision elements of the destination vector that overlap with the corresponding BFLOAT16 elements in the first source vector.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Vector

(FEAT_BF16)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>H</td>
<td>0</td>
</tr>
<tr>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

BFMLAL<bt> <Vd>.4S, <Vn>.8H, <Vm>.H[<index>]

Decode for this encoding

if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt('0':Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);
integer elements = 128 DIV 32;
integer sel = UInt(Q);

Assembler symbols

<bt> Is the bottom or top element specifier, encoded in the "Q" field. It can have the following values:
B when Q = 0
T when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

<index> Is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

Operation

CheckFPAdvSIMDEnabled64();
bits(128) result;
bits(128) operand1 = V[n, 128];
bits(128) operand2 = V[m, 128];
bits(128) operand3 = V[d, 128];
bits(16) element2 = Elem[operand2, index, 16];
for e = 0 to elements-1
  bits(16) element1 = Elem[operand1, 2*e+sel, 16];
bits(32) addend = Elem[operand3, e, 32];
  Elem[result, e, 32] = BFMulAddH(addend, element1, element2, FPCR[]);
V[d, 128] = result;
### C7.2.18 BFMLALB, BFMLALT (vector)

BFLOAT16 floating-point widening multiply-add long (vector) widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first and second source vectors from BFLOAT16 to single-precision format. The instruction then multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFLOAT16 elements in the source vectors.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

#### Vector

(FeAT_BF16)

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 | 17 16 | 15 14 13 12 | 11 10 9 | 8 7 | 6 5 | 4 | 3 | 2 | 1 | 0 |
|----------------|------------|-------------|-------|------|-------------|-------|---|---|---|---|---|---|---|---|
| Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rm | 1 | 1 | 1 | 1 | Rn | Rd |

#### Encoding

BFMLAL<bt> <Vd>.4S, <Vn>.8H, <Vm>.8H

#### Decode for this encoding

if !HaveBF16Ext() then UNDEFINED;
i = UInt(Rd);
j = UInt(Rn);
k = UInt(Rm);

integer iel = (128 DIV 32);
integer sel = UInt(Q);

#### Assembler symbols

<bt> Is the bottom or top element specifier, encoded in the "Q" field. It can have the following values:

- **B**: when Q = 0
- **T**: when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

#### Operation

CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n, 128];
bits(128) operand2 = V[m, 128];
bits(128) operand3 = V[d, 128];
bits(128) result;

for e = 0 to elements-1

| bits(16) | element1 = Elem[operand1, 2*e+sel, 16]; |
| bits(16) | element2 = Elem[operand2, 2*e+sel, 16]; |
| bits(32) | addend = Elem[operand3, e, 32]; |
| Elem[result, e, 32] = BF MulAddH(addend, element1, element2, FPCR[]); |

V[d, 128] = result;
C7.2.19 BFMMLA

BFLOAT16 floating-point matrix multiply-accumulate into 2x2 matrix.

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, irrespective of the other control bits in the FPCR, this instruction:

- Performs two unfused sums-of-products within each two pairs of adjacent BFLOAT16 elements while multiplying the 2x4 matrix of BFLOAT16 values in the first source vector with the 4x2 matrix of BFLOAT16 values in the second source vector. The intermediate single-precision products are rounded before they are summed and the intermediate sum is rounded before accumulation into the 2x2 single-precision matrix in the destination vector. This is equivalent to accumulating two 2-way unfused dot products per destination element.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Generates only the default NaN, as if FPCR.DN is 1.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs two fused sums-of-products within each two pairs of adjacent BFLOAT16 elements while multiplying the 2x4 matrix of BFLOAT16 values in the first source vector with the 4x2 matrix of BFLOAT16 values in the second source vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the 2x2 single-precision matrix in the destination vector. This is equivalent to accumulating two 2-way fused dot products per destination element.
- Generates only the default NaN, as if FPCR.DN is 1.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as controlled by the effective value of the FPCR in the current execution mode, and captured in the FPSR.

**Note**

Arm expects that the BFMMLA instruction will deliver a peak BFLOAT16 multiply throughput that is at least as high as can be achieved using two BFDOT instructions, with a goal that it should have significantly higher throughput.

**Vector**

(Feat_BF16)

```
   [31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 13 12 11 10 9 5 4 0]
   0 1 1 0 1 1 1 0 0 1 0 Rm 1 1 1 0 1 1 Rn 0 Rd
```

**Encoding**

BFMMLA <Vd>.4S, <Vn>.8H, <Vm>.8H

**Decode for this encoding**

if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
Assembler symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(128) op1 = V[n, 128];
bits(128) op2 = V[m, 128];
bits(128) acc = V[d, 128];

V[d, 128] = BFMatMulAdd(acc, op1, op2);
C7.2.20  BIC (vector, immediate)

Bitwise bit Clear (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise AND between each result and the complement of an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### 16-bit variant

Applies when `cmode == 10x1`.

```assembly
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### 32-bit variant

Applies when `cmode == 0xx1`.

```assembly
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

### Decode for all variants of this encoding

```plaintext
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx11' operation = ImmediateOp_BIC;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;
    operation = ImmediateOp_MOVI;
imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

### Assembler symbols

- `<Vd>` Is the name of the SIMD&FP register, encoded in the "Rd" field.
- `<T>` For the 16-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  - `4H` when `Q = 0`
  - `8H` when `Q = 1`
For the 32-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

25 \text{ when } Q = 0

45 \text{ when } Q = 1

\text{<imm8>} \quad \text{Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".}

\text{<amount>} \quad \text{For the 16-bit variant: is the shift amount encoded in the "cmode<1>" field. It can have the following values:}

0 \text{ when } \text{cmode}<1> = 0

8 \text{ when } \text{cmode}<1> = 1

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in the "cmode<2:1>" field. It can have the following values:

0 \text{ when } \text{cmode}<2:1> = 00

8 \text{ when } \text{cmode}<2:1> = 01

16 \text{ when } \text{cmode}<2:1> = 10

24 \text{ when } \text{cmode}<2:1> = 11

defaulting to 0 if LSL is omitted.

\textbf{Operation}

\texttt{CheckFPAdvSIMDEnabled64();}
\texttt{bits(datasize) operand;}
\texttt{bits(datasize) result;}

\texttt{case operation of}
\texttt{when \text{ImmediateOp_MOVI}}
\texttt{result = \text{imm};}
\texttt{when \text{ImmediateOp_MVNI}}
\texttt{result = \text{NOT(imm)};}\n\texttt{when \text{ImmediateOp_ORR}}
\texttt{operand = \text{V[rd, datasize]};}
\texttt{result = operand OR \text{imm};}
\texttt{when \text{ImmediateOp_BIC}}
\texttt{operand = \text{V[rd, datasize]};}
\texttt{result = operand AND \text{NOT(imm)};}\n\texttt{\text{V[rd, datasize] = result;}}

\textbf{Operational information}

\texttt{If PSTATE.DIT is 1:}

\begin{itemize}
  \item The execution time of this instruction is independent of:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
      \item The values of the NZCV flags.
    \end{itemize}
  \item The response of this instruction to asynchronous exceptions does not vary based on:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
      \item The values of the NZCV flags.
    \end{itemize}
\end{itemize}
C7.2.21 BIC (vector, register)

Bitwise bit Clear (vector, register). This instruction performs a bitwise AND between the first source SIMD&FP register and the complement of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Encoding**

BIC <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;

**Assembler symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

<table>
<thead>
<tr>
<th>Q</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

checkfpadvanced64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
operand2 = NOT(operand2);
result = operand1 AND operand2;
V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.22 BIF

Bitwise Insert if False. This instruction inserts each bit from the first source SIMD&FP register into the destination SIMD&FP register if the corresponding bit of the second source SIMD&FP register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

BIF <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

\[
\begin{align*}
\text{integer } & \text{ d } = \text{ UInt(Rd);} \\
\text{integer } & \text{ n } = \text{ UInt(Rn);} \\
\text{integer } & \text{ m } = \text{ UInt(Rm);} \\
\text{integer } & \text{ datasize } = \text{ if } Q = '1' \text{ then } 128 \text{ else } 64;
\end{align*}
\]

Assembler symbols

- \(<Vd>\) is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<T>\) is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  - 88 when \( Q = 0 \)
  - 168 when \( Q = 1 \)
- \(<Vn>\) is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- \(<Vm>\) is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

\[
\begin{align*}
\text{CheckFPAdvSIMDEnabled64();} \\
\text{bits(datasize) operand1;} \\
\text{bits(datasize) operand3;} \\
\text{bits(datasize) operand4 = V[n, datasize];} \\
\text{operand1 = V[d, datasize];} \\
\text{operand3 = NOT(V[m, datasize]);} \\
\text{V[d, datasize] = operand1 EOR ((operand1 EOR operand4) AND operand3);} \\
\end{align*}
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.23   BIT

Bitwise Insert if True. This instruction inserts each bit from the first source SIMD&FP register into the SIMD&FP destination register if the corresponding bit of the second source SIMD&FP register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Encoding**

```
0 | 1 0 1 1 0 1 0 1 | Rm
0 0 0 1 1 1 | Rn
Rd
```

**Decode for this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

**Assembler symbols**

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  - 88 when Q = 0
  - 168 when Q = 1
- `<Vn>`: Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>`: Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand3;
bits(datasize) operand4 = V[n, datasize];
operand1 = V[d, datasize];
operand3 = V[m, datasize];
V[d, datasize] = operand1 EOR ((operand1 EOR operand4) AND operand3);
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.24 BSL

Bitwise Select. This instruction sets each bit in the destination SIMD&FP register to the corresponding bit from the first source SIMD&FP register when the original destination bit was 1, otherwise from the second source SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

BSL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
88 when Q = 0
168 when Q = 1

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand3;
bits(datasize) operand4 = V[n, datasize];
operand1 = V[m, datasize];
operand3 = V[d, datasize];
V[d, datasize] = operand1 EOR ((operand1 EOR operand4) AND operand3);

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.25 CLS (vector)

Count Leading Sign bits (vector). This instruction counts the number of consecutive bits following the most significant bit that are the same as the most significant bit in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The count does not include the most significant bit itself.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

```
| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9  | 5 4  | 0 |
| 0 | Q | 0 | 1 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | Rn | Rd |
```

U

**Encoding**

CLS \(<Vd>.<T>, <Vn>.<T>\)

**Decode for this encoding**

- integer \(d = \text{UInt}(Rd)\);
- integer \(n = \text{UInt}(Rn)\);
- if size == '11' then UNDEFINED;
- integer esize = 8 << UInt(size);
- integer datasize = if Q == '1' then 128 else 64;
- integer elements = datasize DIV esize;

\[\text{CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;}\]

**Assembler symbols**

- \(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<T>\) Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - \(8B\) when size = 00, Q = 0
  - \(16B\) when size = 00, Q = 1
  - \(4H\) when size = 01, Q = 0
  - \(8H\) when size = 01, Q = 1
  - \(2S\) when size = 10, Q = 0
  - \(4S\) when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

- \(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer count;
for e = 0 to elements-1
  if countop == CountOp_CLS then
    count = CountLeadingSignBits(Elem[operand, e, esize]);
  else
```
count = CountLeadingZeroBits(Elem[operand, e, esize]);
Elem[result, e, esize] = count<esize-1:0>;
V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.26   CLZ (vector)

Count Leading Zero bits (vector). This instruction counts the number of consecutive zeros, starting from the most significant bit, in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

CLZ <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  8B  when size = 00, Q = 0
  16B when size = 00, Q = 1
  4H  when size = 01, Q = 0
  8H  when size = 01, Q = 1
  2S  when size = 10, Q = 0
  4S  when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

integer count;
for e = 0 to elements-1
  if countop == CountOp_CLS then
    count = CountLeadingSignBits(Elem[operand, e, esize]);
  else
    count = CountLeadingZeroBits(Elem[operand, e, esize]);
\[
\text{Elem}[\text{result, e, esize}] = \text{count<esize-1:0>;} \\
\text{V}[\text{d, datasize}] = \text{result};
\]

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.27  CMEQ (register)

Compare bitwise Equal (vector). This instruction compares each vector element from the first source SIMD&FP register with the corresponding vector element from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0</td>
<td>size 1</td>
<td>Rm 1 0 0 1 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

CMEQ <V><d>, <V><n>, <V><m>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1 0 1 1 0</td>
<td>size 1</td>
<td>Rm 1 0 0 1 1</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Encoding

CMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11
The following encodings are reserved:

- \( \text{size} = 0x \).
- \( \text{size} = 10 \).

\(<d>\) Is the number of the SIMD\&FP destination register, in the "Rd" field.

\(<n>\) Is the number of the first SIMD\&FP source register, encoded in the "Rn" field.

\(<m>\) Is the number of the second SIMD\&FP source register, encoded in the "Rm" field.

\(<Vd>\) Is the name of the SIMD\&FP destination register, encoded in the "Rd" field.

\(<T>\) Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- \(8B\) when \(\text{size} = 00, Q = 0\)
- \(16B\) when \(\text{size} = 00, Q = 1\)
- \(4H\) when \(\text{size} = 01, Q = 0\)
- \(8H\) when \(\text{size} = 01, Q = 1\)
- \(2S\) when \(\text{size} = 10, Q = 0\)
- \(4S\) when \(\text{size} = 10, Q = 1\)
- \(2D\) when \(\text{size} = 11, Q = 0\) is reserved.

\(<Vn>\) Is the name of the first SIMD\&FP source register, encoded in the "Rn" field.

\(<Vm>\) Is the name of the second SIMD\&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

```c
CheckFPAvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.28   CMEQ (zero)

Compare bitwise Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

```plaintext
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] [11 10 9] | 5 4 | 0 |
0 1 0 1 1 1 0 size 1 0 0 0 0 0 1 0 0 1 1 0 Rn | Rd |
U        op
```

**Encoding**

CMEQ <V><d>, <V><n>, #0

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
when '00' comparison = CompareOp_GT;
when '01' comparison = CompareOp_GE;
when '10' comparison = CompareOp_EQ;
when '11' comparison = CompareOp_LE;
end;
```

**Vector**

```plaintext
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] [11 10 9] | 5 4 | 0 |
0 Q 0 1 1 1 0 size 1 0 0 0 0 0 1 0 0 1 1 0 Rn | Rd |
U        op
```

**Encoding**

CMEQ <Vd>.<T>, <Vn>.<T>, #0

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
in integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
when '00' comparison = CompareOp_GT;
end;
```
when '01' comparison = CompareOp_GE;
when '10' comparison = CompareOp_EQ;
when '11' comparison = CompareOp_LE;

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
  D when size = 11
The following encodings are reserved:
• size = 0x.
• size = 10.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  8B when size = 00, Q = 0
  16B when size = 00, Q = 1
  4H when size = 01, Q = 0
  8H when size = 01, Q = 1
  2S when size = 10, Q = 0
  4S when size = 10, Q = 1
  2D when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.

<n> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
  element = SInt(Elem[operand, e, esize]);
case comparison of
  when CompareOp_GT test_passed = element > 0;
  when CompareOp_GE test_passed = element >= 0;
  when CompareOp_EQ test_passed = element == 0;
  when CompareOp_LE test_passed = element <= 0;
  when CompareOp_LT test_passed = element < 0;
  Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);
V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
— The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.29  CMGE (register)

Compare signed Greater than or Equal (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than or equal to the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 1 1 1 0</td>
<td>size 1</td>
<td>Rm 0 1 1 1</td>
<td>1</td>
<td>Rn 1 1</td>
<td>Rd 0</td>
<td></td>
</tr>
<tr>
<td>U eq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

CMGE <V><d>, <V><n>, <V><m>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 1 1 1 0</td>
<td>size 1</td>
<td>Rm 0 1 1 1</td>
<td>1</td>
<td>Rn 1 1</td>
<td>Rd 0</td>
</tr>
<tr>
<td>U eq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

CMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
Assembler symbols

<\'> Is a width specifier, encoded in the "size" field. It can have the following values:

0    when size = 11

The following encodings are reserved:

• size = 0x.
• size = 10.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B    when size = 00, Q = 0
16B   when size = 00, Q = 1
4H    when size = 01, Q = 0
8H    when size = 01, Q = 1
2S    when size = 10, Q = 0
4S    when size = 10, Q = 1
2D    when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize); 

V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.30   CMGE (zero)

Compare signed Greater than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1</td>
<td>1 1 1 0</td>
<td>size 1 0 0 0 0 0 1 0 0 0 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U op
```

Encoding

CMGE <V><d>, <V><n>, #0

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
when '00' comparison = CompareOp_GT;
when '01' comparison = CompareOp_GE;
when '10' comparison = CompareOp_EQ;
when '11' comparison = CompareOp_LE;

Vector

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1</td>
<td>0 1 1 0</td>
<td>size 1 0 0 0 0 0 1 0 0 0 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U op
```

Encoding

CMGE <Vd>.<T>, <Vn>.<T>, #0

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
when '00' comparison = CompareOp_GT;
when '01' comparison = CompareOp_GE;
when '10' comparison = CompareOp_EQ;
when '11' comparison = CompareOp_LE;

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
   0 when size = 11
The following encodings are reserved:
   • size = 0x.
   • size = 10.
<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
   8B when size = 00, Q = 0
   16B when size = 00, Q = 1
   4H when size = 01, Q = 0
   8H when size = 01, Q = 1
   2S when size = 10, Q = 0
   4S when size = 10, Q = 1
   2D when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.
<n> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
   element = SInt(Elem[operand, e, esize]);
   case comparison of
      when CompareOp_GT test_passed = element > 0;
      when CompareOp_GE test_passed = element >= 0;
      when CompareOp_EQ test_passed = element == 0;
      when CompareOp_LE test_passed = element <= 0;
      when CompareOp_LT test_passed = element < 0;
      Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);
V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:
   • The execution time of this instruction is independent of:
      — The values of the data supplied in any of its registers.
— The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.31 CMGT (register)

Compare signed Greater than (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 0 1 1 1 1 0 | size 1 | Rm 0 0 1 1 0 1 | Rn | Rd |
```

U  eq

**Encoding**

CMGT <V><d>, <V><n>, <V><m>

**Decode for this encoding**

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

**Vector**

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 | Q 0 0 1 1 1 0 | size 1 | Rm 0 0 1 1 0 1 | Rn | Rd |
```

U  eq

**Encoding**

CMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```
Assembler symbols

\(<\text{V}>\) Is a width specifier, encoded in the "size" field. It can have the following values:
\begin{itemize}
  \item 0 when size = 11
\end{itemize}
The following encodings are reserved:
\begin{itemize}
  \item size = 0x.
  \item size = 10.
\end{itemize}

\(<\text{d}>\) Is the number of the SIMD&FP destination register, in the "Rd" field.

\(<\text{n}>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<\text{m}>\) Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

\(<\text{Vd}>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<\text{T}>\) Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
\begin{itemize}
  \item 8B when size = 00, Q = 0
  \item 16B when size = 00, Q = 1
  \item 4H when size = 01, Q = 0
  \item 8H when size = 01, Q = 1
  \item 2S when size = 10, Q = 0
  \item 4S when size = 10, Q = 1
  \item 2D when size = 11, Q = 1
\end{itemize}
The encoding size = 11, Q = 0 is reserved.

\(<\text{Vn}>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<\text{Vm}>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
  Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);
V[d, datasize] = result;
\end{verbatim}

Operational information

If PSTATE.DIT is 1:
\begin{itemize}
  \item The execution time of this instruction is independent of:
  \begin{itemize}
    \item The values of the data supplied in any of its registers.
    \item The values of the NZCV flags.
  \end{itemize}
\end{itemize}
The response of this instruction to asynchronous exceptions does not vary based on:

— The values of the data supplied in any of its registers.
— The values of the NZCV flags.
C7.2.32   CMGT (zero)

Compare signed Greater than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
0 1 0 |1 1 1 0 |size 1 0 0 0 |0 1 0 0 |0 1 0 | Rd
U   op
```

**Encoding**

CMGT <V><d>, <V><n>, #0

**Decode for this encoding**

```java
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

```

Vector

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
0 |Q|0 |0 1 1 1 0 |size 1 0 0 0 |0 1 0 0 |0 1 0 | Rd
U   op
```

**Encoding**

CMGT <Vd>.<T>, <Vn>.<T>, #0

**Decode for this encoding**

```java
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  ...```
when '01' comparison = CompareOp_GE;
when '10' comparison = CompareOp_EQ;
when '11' comparison = CompareOp_LE;

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
   0  when size = 11
The following encodings are reserved:
   •  size = 0x.
   •  size = 10.
<\d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
   8B  when size = 00, Q = 0
   16B when size = 00, Q = 1
   4H  when size = 01, Q = 0
   8H  when size = 01, Q = 1
   2S  when size = 10, Q = 0
   4S  when size = 10, Q = 1
   2D  when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
   element = SInt(Elem[operand, e, esize]);
   case comparison of
      when CompareOp_GT test_passed = element > 0;
      when CompareOp_GE test_passed = element >= 0;
      when CompareOp_EQ test_passed = element == 0;
      when CompareOp_LE test_passed = element <= 0;
      when CompareOp_LT test_passed = element < 0;
Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);
V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:
   • The execution time of this instruction is independent of:
      — The values of the data supplied in any of its registers.
— The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.33   CMHI (register)

Compare unsigned Higher (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

```
| 31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13 12|11 10 9 | 5 4 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | size | 1 | Rm | 0 | 0 | 1 | 1 | 0 | 1 | Rn | Rd |
```

**Encoding**

CMHI <V><d>, <V><n>, <V><m>

**Decode for this encoding**

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Vector

```
| 31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13 12|11 10 9 | 5 4 | 0 |
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm | 0 | 0 | 1 | 1 | 0 | 1 | Rn | Rd |
```

**Encoding**

CMHI <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```
Assembler symbols

\(<V>\) Is a width specifier, encoded in the "size" field. It can have the following values:

- \(0\) when size = 11

The following encodings are reserved:

- size = 0x
- size = 10

\(<d>\) Is the number of the SIMD&FP destination register, in the "Rd" field.

\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<m>\) Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

\(<V_d>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- \(8B\) when size = 00, Q = 0
- \(16B\) when size = 00, Q = 1
- \(4H\) when size = 01, Q = 0
- \(8H\) when size = 01, Q = 1
- \(2S\) when size = 10, Q = 0
- \(4S\) when size = 10, Q = 1
- \(2D\) when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

\(<V_n>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<V_m>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
  Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.34  CMHS (register)

Compare unsigned Higher or Same (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than or equal to the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**Encoding**

CMHS <V><d>, <V><n>, <V><m>

**Decode for this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Vector

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**Encoding**

CMHS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```
Assembler symbols

\<V> \quad \text{Is a width specifier, encoded in the "size" field. It can have the following values:}
\begin{align*}
0 & \quad \text{when size = 11} \\
\\text{The following encodings are reserved:} & \\
\quad & \quad \begin{cases} 
\text{size = 0x.} \\
\text{size = 10.}
\end{cases}
\end{align*}

\<\text{d}> \quad \text{Is the number of the SIMD\&FP destination register, in the "Rd" field.}

\<\text{n}> \quad \text{Is the number of the first SIMD\&FP source register, encoded in the "Rn" field.}

\<\text{m}> \quad \text{Is the number of the second SIMD\&FP source register, encoded in the "Rm" field.}

\<\text{Vd}> \quad \text{Is the name of the SIMD\&FP destination register, encoded in the "Rd" field.}

\<\text{T}> \quad \text{Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:}
\begin{align*}
8B & \quad \text{when size = 00, Q = 0} \\
16B & \quad \text{when size = 00, Q = 1} \\
4H & \quad \text{when size = 01, Q = 0} \\
8H & \quad \text{when size = 01, Q = 1} \\
2S & \quad \text{when size = 10, Q = 0} \\
4S & \quad \text{when size = 10, Q = 1} \\
2D & \quad \text{when size = 11, Q = 1} \\
\text{The encoding size = 11, Q = 0 is reserved.}
\end{align*}

\<\text{Vn}> \quad \text{Is the name of the first SIMD\&FP source register, encoded in the "Rn" field.}

\<\text{Vm}> \quad \text{Is the name of the second SIMD\&FP source register, encoded in the "Rm" field.}

Operation for all encodings

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);
V[d, datasize] = result;
\end{verbatim}

Operational information

If PSTATE.DIT is 1:
\begin{itemize}
\item The execution time of this instruction is independent of:
  \begin{itemize}
  \item The values of the data supplied in any of its registers.
  \item The values of the NZCV flags.
  \end{itemize}
\end{itemize}
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.35 CMLE (zero)

Compare signed Less than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

C

Encoding

CMLE <V><d>, <V><n>, #0

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;

case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

**Vector**

C

Encoding

CMLE <Vd>.<T>, <Vn>.<T>, #0

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;

case op:U of
  when '00' comparison = CompareOp_GT;
when '01' comparison = CompareOp_GE;
when '10' comparison = CompareOp_EQ;
when '11' comparison = CompareOp_LE;

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
- 0 when size = 11
The following encodings are reserved:
  - size = 0x.
  - size = 10.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.

<n> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
  element = SInt(Elem[operand, e, esize]);
  case comparison of
    when CompareOp_GT test_passed = element > 0;
    when CompareOp_GE test_passed = element >= 0;
    when CompareOp_EQ test_passed = element == 0;
    when CompareOp_LE test_passed = element <= 0;
    when CompareOp_LT test_passed = element < 0;
    when CompareOp_IE test_passed = element <= 0;
    when CompareOp_LT test_passed = element < 0;
  Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
— The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.36   CMLT (zero)

Compare signed Less than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0</td>
<td>1 1 1 0</td>
<td>size</td>
<td>1 0 0 0 0</td>
<td>0 1 0 1 0</td>
<td>1 0</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Encoding

CMLT <V><d>, <V><n>, #0

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;

Vector

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
</tr>
</tbody>
</table>

Encoding

CMLT <Vd>.<T>, <Vn>.<T>, #0

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11
The following encodings are reserved:
- size = 0x.
- size = 10.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
8B when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
        Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.37   CMTST

Compare bitwise Test bits nonzero (vector). This instruction reads each vector element in the first source SIMD&FP register, performs an AND with the corresponding vector element in the second source SIMD&FP register, and if the result is not zero, sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

```
[31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 13 12|11 10 9 | 5 | 4 | 0 |
0 1 0 | 1 1 1 0 | size 1 | Rm 1 0 0 1 1 | Rn | Rd 1
```

**Encoding**

CMTST <V><d>, <V><n>, <V><m>

**Decode for this encoding**

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');
```

**Vector**

```
[31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 13 12|11 10 9 | 5 | 4 | 0 |
0 | Q | 0 1 1 0 | size 1 | Rm 1 0 0 1 1 | Rn | Rd 1
```

**Encoding**

CMTST <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');
```

**Assembler symbols**

<\V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11
The following encodings are reserved:

- size = 0x
- size = 10.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

</d> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

</n> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

</m> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

The encoding size = 11, Q = 0 is reserved.

</v> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

</v> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;
for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  if and_test then
    test_passed = !IsZero(element1 AND element2);
  else
    test_passed = (element1 == element2);
  Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);
V[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.38  CNT

Population Count per byte. This instruction counts the number of bits that have a value of one in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

CNT <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '00' then UNDEFINED;
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- **8B** when size = 00, Q = 0
- **16B** when size = 00, Q = 1

The following encodings are reserved:

- size = 01, Q = x.
- size = 1x, Q = x.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

integer count;
for e = 0 to elements-1
  count = BitCount(Elem[operand, e, esize]);
  Elem[result, e, esize] = count<esize-1:0>;
V[d, datasize] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.39   DUP (element)

Duplicate vector element to vector or scalar. This instruction duplicates the vector element at the specified element index in the source SIMD&FP register into a scalar or each element in a vector, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (scalar). The alias is always the preferred disassembly.

Scalar

```
| 31 30 29 28 27 26 25 24 23 22 21 20 | 15 14 13 12 11 10  9 |  5  4 |  0 |
|  0  1  0  1  1  1  0  0  0  0 | imm5  0  0  0  0  1 | Rn | Rd |
```

Encoding

DUP <V><d>, <Vn>.<T>[<index>]

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

integer esize = 8 << size;
integer datasize = esize;
integer elements = 1;
```

Vector

```
| 31 30 29 28 27 26 25 24 23 22 21 20 | 15 14 13 12 11 10  9 |  5  4 |  0 |
|  0  1  0  0  0  0  1  0  0  0  0  0 | imm5  0  0  0  0  1 | Rn | Rd |
```

Encoding

DUP <Vd>.<T>, <Vn>.<Ts>[<index>]

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

if size == 3 && Q == '0' then UNDEFINED;
integer esize = 8 << size;
integer elements = 1;
```
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler symbols

<T>
For the scalar variant: is the element width specifier, encoded in the "imm5" field. It can have the following values:

- B when imm5 = xxxx1
- H when imm5 = xxx10
- S when imm5 = xx100
- D when imm5 = x1000

The encoding imm5 = x0000 is reserved.

For the vector variant: is an arrangement specifier, encoded in the "imm5:Q" field. It can have the following values:

- 8B when imm5 = xxxx1, Q = 0
- 16B when imm5 = xxxx1, Q = 1
- 4H when imm5 = xxx10, Q = 0
- 8H when imm5 = xxx10, Q = 1
- 2S when imm5 = xx100, Q = 0
- 4S when imm5 = xx100, Q = 1
- 2D when imm5 = x1000, Q = 1

The following encodings are reserved:

- imm5 = x0000, Q = x.
- imm5 = x1000, Q = 0.

<Ts>
Is an element size specifier, encoded in the "imm5" field. It can have the following values:

- B when imm5 = xxxx1
- H when imm5 = xxx10
- S when imm5 = xx100
- D when imm5 = x1000

The encoding imm5 = x0000 is reserved.

<V>
Is the destination width specifier, encoded in the "imm5" field. It can have the following values:

- B when imm5 = xxxx1
- H when imm5 = xxx10
- S when imm5 = xx100
- D when imm5 = x1000

The encoding imm5 = x0000 is reserved.

<Vn>
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index>
Is the element index encoded in the "imm5" field. It can have the following values:

- imm5<4:1> when imm5 = xxxx1
- imm5<4:2> when imm5 = xxx10
- imm5<4:3> when imm5 = xx100
- imm5<4> when imm5 = x1000

The encoding imm5 = x0000 is reserved.

<d>
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n, idxdsize];
bits(datasize) result;
bits(esize) element;

    element = Elem[operand, index, esize];
    for e = 0 to elements-1
        Elem[result, e, esize] = element;
    V[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**C7.2.40  DUP (general)**

Duplicate general-purpose register to vector. This instruction duplicates the contents of the source general-purpose register into a scalar or each element in a vector, and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Encoding**

DUP <Vd>.<T>, <R><n>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

// imm5<4:size+1> is IGNORED

if size == 3 && Q == '0' then UNDEFINED;
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**Assembler symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "imm5:Q" field. It can have the following values:

- **8B** when imm5 = xxxx1, Q = 0
- **16B** when imm5 = xxxx1, Q = 1
- **4H** when imm5 = xxx10, Q = 0
- **8H** when imm5 = xxx10, Q = 1
- **2S** when imm5 = xx100, Q = 0
- **4S** when imm5 = xx100, Q = 1
- **2D** when imm5 = x1000, Q = 1

The following encodings are reserved:

- imm5 = x0000, Q = x.
- imm5 = x1000, Q = 0.

<R> Is the width specifier for the general-purpose source register, encoded in the "imm5" field. It can have the following values:

- **W** when imm5 = xxxx1
- **W** when imm5 = xxxx10
- **W** when imm5 = xx100
- **X** when imm5 = x1000
The encoding imm5 = x0000 is reserved. Unspecified bits in "imm5" are ignored but should be set to zero by an assembler.

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n, esize];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = element;
    V[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.41 EOR (vector)

Bitwise Exclusive OR (vector). This instruction performs a bitwise Exclusive OR operation between the two source SIMD&FP registers, and places the result in the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

EOR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
88 when Q = 0
168 when Q = 1
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n, datasize];
operand1 = V[m, datasize];
operand2 = Zeros(datasize);
operand3 = Ones(datasize);
V[d, datasize] = operand1 EOR ((operand2 EOR operand4) AND operand3);

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.42   EOR3

Three-way Exclusive OR performs a three-way exclusive OR of the values in the three source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SHA3 is implemented.

Advanced SIMD

(Feat_SHA3)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1 1 1 0 0</td>
<td>0 0</td>
<td>Rm</td>
<td>Ra</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

EOR3 <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B

Decode for this encoding

if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Va> Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Va = V[a, 128];
V[d, 128] = Vn EOR Vm EOR Va;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.43  EXT

Extract vector from pair of vectors. This instruction extracts the lowest vector elements from the second source SIMD&FP register and the highest vector elements from the first source SIMD&FP register, concatenates the results into a vector, and writes the vector to the destination SIMD&FP register vector. The index value specifies the lowest vector element to extract from the first source register, and consecutive elements are extracted from the first, then second, source registers until the destination vector is filled.

The following figure shows the operation of EXT doubleword operation for Q = 0 and imm4<2:0> = 3.

![Diagram showing the operation of EXT doubleword operation for Q = 0 and imm4<2:0> = 3.]

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
[31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 | 11 10 9 | 5 4 | 0 |
0 | Q | 1 | 0 | 1 | 1 | 0 | 0 | 0 | Rm | 0 | imm4 | 0 | Rn | Rd |
```

**Encoding**

EXT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<index>

**Decode for this encoding**

```plaintext
d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);

if Q == '0' && imm4<3> == '1' then UNDEFINED;

datasize = if Q == '1' then 128 else 64;
position = UInt(imm4) << 3;
```

**Assembler symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  - 8B when Q = 0
  - 16B when Q = 1
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<index>` Is the lowest numbered byte element to be extracted, encoded in the "Q:imm4" field. It can have the following values:
  - imm4<2:0> when Q = 0, imm4<3> = 0
  - imm4 when Q = 1, imm4<3> = x
The encoding $Q = 0$, imm4<3> = 1 is reserved.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) hi = V[m, datasize];
bits(datasize) lo = V[n, datasize];
bits(datasize*2) concat = hi:lo;

V[d, datasize] = concat<position+datasize-1:position>;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.44   FABD

Floating-point Absolute Difference (vector). This instruction subtracts the floating-point values in the elements of the second source SIMD&FP register, from the corresponding floating-point values in the elements of the first source SIMD&FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(FEAT_FP16)

Encoding

FABD <Hd>, <Hn>, <Hm>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;

Scalar single-precision and double-precision

Encoding

FABD <V>d>, <V>n>, <V>m>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;
Vector half precision

(FEAT_FP16)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**Encoding**

FABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```c
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Vector single-precision and double-precision

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**Encoding**

FABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

**Assembler symbols**

- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:
  - `S` when `sz = 0`
  - `D` when `sz = 1`
- `<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
bits(esize) diff;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, fpcr);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;
V[d, 128] = result;
```
C7.2.45 FABS (vector)

Floating-point Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(FEAT_FP16)

![Encoding](image)

Encoding

FABS <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

Single-precision and double-precision

![Encoding](image)

Encoding

FABS <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>
For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0
8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn>
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;
V[d, datasize] = result;
```
**C7.2.46 FABS (scalar)**

Floating-point Absolute value (scalar). This instruction calculates the absolute value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Half-precision variant**

Applies when $ftype == 11$.

FABS $<Hd>$, $<Hn>$

**Single-precision variant**

Applies when $ftype == 00$.

FABS $<Sd>$, $<Sn>$

**Double-precision variant**

Applies when $ftype == 01$.

FABS $<Dd>$, $<Dn>$

**Decode for all variants of this encoding**

```c
integer d = Uint(Rd);
integer n = Uint(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
end case;
```

**Assembler symbols**

- $<Dd>$ is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- $<Dn>$ is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- $<Hd>$ is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- $<Hn>$ is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- $<Sd>$ is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- $<Sn>$ is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else 0<127:0>;

bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPAbs(operand);
V[d, 128] = result;
C7.2.47 FACGE

Floating-point Absolute Compare Greater than or Equal (vector). This instruction compares the absolute value of each floating-point value in the first source SIMD&FP register with the absolute value of the corresponding floating-point value in the second source SIMD&FP register and if the first value is greater than or equal to the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(FeAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1</td>
<td>1 1 1 0</td>
<td>0 1 0</td>
<td>Rm</td>
<td>0 0 1 0</td>
<td>1 1</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

U E ac

Encoding

FACGE <Hd>, <Hn>, <Hm>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1</td>
<td>1 1 1 0</td>
<td>0 sz 1</td>
<td>Rm</td>
<td>1 1 1 0</td>
<td>1 1</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

U E ac

Encoding

FACGE <V><d>, <V><n>, <V><m>
Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Vector half precision

(FEAT_FP16)

Encoding

FACGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Vector single-precision and double-precision
Encoding

FACGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<V> Is a width specifier, encoded in the "sz" field. It can have the following values:
  S when sz = 0
  D when sz = 1
<vd> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  4H when Q = 0
  8H when Q = 1
For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
  2S when sz = 0, Q = 0
  4S when sz = 0, Q = 1
  2D when sz = 1, Q = 1
The encoding sz = 1, Q = 0 is reserved.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);
V[d, 128] = result;
C7.2.48 FACGT

Floating-point Absolute Compare Greater than (vector). This instruction compares the absolute value of each vector element in the first source SIMD&FP register with the absolute value of the corresponding vector element in the second source SIMD&FP register and if the first value is greater than the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(FEAT_FP16)

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>U</td>
<td>E</td>
<td>ac</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FACGT <Hd>, <Hn>, <Hm>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
when '000' cmp = CompareOp_EQ; abs = FALSE;
when '010' cmp = CompareOp_GE; abs = FALSE;
when '011' cmp = CompareOp_GE; abs = TRUE;
when '110' cmp = CompareOp_GT; abs = FALSE;
when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>U</td>
<td>E</td>
<td>ac</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FACGT <V<d>, <V<n>, <V>m>
Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;

Vector half precision

(FEAT_FP16)

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[16 15 14 13 12 11 10 9]</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Decoding

FACGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;

Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[16 15 14 13 12 11 10 9]</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

C7-2140 Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  ARM DDI 0487I.a  Non-Confidential  ID081822
**Encoding**

FACGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;
```

**Assembler symbols**

- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:
  - S when sz = 0
  - D when sz = 1
- `<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.
- `<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  - 4H when Q = 0
  - 8H when Q = 1
For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
  - 2S when sz = 0, Q = 0
  - 4S when sz = 0, Q = 1
  - 2D when sz = 1, Q = 1
The encoding sz = 1, Q = 0 is reserved.
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
            Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, 128] = result;
C7.2.49   FADD (vector)

Floating-point Add (vector). This instruction adds corresponding vector elements in the two source SIMD&FP registers, writes the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

( FEAT_FP16 )

[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 | Q | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 1 | 0 | 1 | Rn | Rd |

Encoding

FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');

Single-precision and double-precision

[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 | Q | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | Rm | 1 | 1 | 0 | 1 | 0 | 1 | Rn | Rd |

Encoding

FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
Assembler symbols

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
- 4H when \(Q = 0\)
- 8H when \(Q = 1\)

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
- 2S when \(sz = 0, Q = 0\)
- 4S when \(sz = 0, Q = 1\)
- 2D when \(sz = 1, Q = 1\)
- The encoding \(sz = 1, Q = 0\) is reserved.

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);
V[d, datasize] = result;
```

C7.2.50   FADD (scalar)

Floating-point Add (scalar). This instruction adds the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant

Applies when ftype == 11.

FADD <Hd>, <Hn>, <Hm>

Single-precision variant

Applies when ftype == 00.

FADD <Sd>, <Sn>, <Sm>

Double-precision variant

Applies when ftype == 01.

FADD <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
when '00' esize = 32;
when '01' esize = 64;
when '10' UNDEFINED;
when '11'
    if HaveFP16Ext() then
        esize = 16;
    else
        UNDEFINED;

Assembler symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

%Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

Elem[result, 0, esize] = FPAdd(operand1, operand2, fpcr);
V[d, 128] = result;
```
C7.2.51 FADDP (scalar)

Floating-point Add Pair of elements (scalar). This instruction adds two floating-point vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(_FEAT_FP16)

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9     | 5 4 | 0 |
0 1 | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | Rd |
```

Encoding

FADDP <V><d>, <Vn>.<T>

Decode for this encoding

```c
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;
```

Single-precision and double-precision

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9     | 5 4 | 0 |
0 1 | 1 | 1 | 1 | 1 | 0 | 0 | sz | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | Rd |
```

Encoding

FADDP <V><d>, <Vn>.<T>

Decode for this encoding

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
```

Assembler symbols

<V> For the half-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

- H when sz = 0
The encoding \( sz = 1 \) is reserved.

For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

- \( S \) when \( sz = 0 \)
- \( D \) when \( sz = 1 \)

\(<d>\) is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Vn>\) is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<T>\) For the half-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

- \( 2H \) when \( sz = 0 \)
- The encoding \( sz = 1 \) is reserved.

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

- \( 2S \) when \( sz = 0 \)
- \( 2D \) when \( sz = 1 \)

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FADD, operand, esize);
```
C7.2.52 FADDP (vector)

Floating-point Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision

(Feat_Fp16)

![Half-precision Instruction Format](image)

**Encoding**

FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```plaintext
integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean pair = (U == '1');
```

### Single-precision and double-precision

![Single-precision and Double-precision Instruction Format](image)

**Encoding**

FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```plaintext
integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << Uint(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```
boolean pair = (U == '1');

**Assembler symbols**

<Vd>  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>  
For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H  when Q = 0
- 8H  when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S  when sz = 0, Q = 0
- 4S  when sz = 0, Q = 1
- 2D  when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn>  
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>  
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);
V[d, datasize] = result;
```
C7.2.53 FCADD

Floating-point Complex Add.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with
the more significant element holding the imaginary part of the number and the less significant element holding
the real part of the number. Each element holds a floating-point value. It performs the following computation on the
 corresponding complex number element pairs from the two source registers:

• Considering the complex number from the second source register on an Argand diagram, the number is
rotated counterclockwise by 90 or 270 degrees.

• The rotated complex number is added to the complex number from the first source register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results
in either a flag being set in FPSR or a synchronous exception being generated. For more information, see
Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

Vector

( FEAT_FCMA )

Encoding

FCADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10  9</th>
<th>5  4  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  Q  1  0  1  1  1  0</td>
<td>size 0</td>
<td>Rm 1 1</td>
</tr>
</tbody>
</table>

Decode for this encoding

if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then UNDEFINED;
if Q == '0' && size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  4H when size == 01, Q == 0
  8H when size == 01, Q == 1
  2S when size == 10, Q == 0
  4S when size == 10, Q == 1
  2D when size == 11, Q == 1

The following encodings are reserved:
• size == 00, Q == x.
size = 11, Q = 0.

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<rotate>  Is the rotation, encoded in the "rot" field. It can have the following values:

90 when rot = 0
270 when rot = 1

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element3;

for e = 0 to (elements DIV 2)-1
  case rot of
    when '0'
      element1 = FPNeg(Elem[operand2, e*2+1, esize]);
      element3 = Elem[operand2, e*2, esize];
    when '1'
      element1 = Elem[operand2, e*2+1, esize];
      element3 = FPNeg(Elem[operand2, e*2, esize]);
  Elem[result, e*2, esize] = FPAdd(Elem[operand1, e*2, esize], element1, FPCR[]);
  Elem[result, e*2+1, esize] = FPAdd(Elem[operand1, e*2+1, esize], element3, FPCR[]);

V[d, datasize] = result;
C7.2.54   FCCMP

Floating-point Conditional quiet Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the PSTATE.\{N, Z, C, V\} flags. If the condition does not pass then the PSTATE.\{N, Z, C, V\} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant

Applies when ftype == 11.

FCCMP <Hn>, <Hm>, #<nzcv>, <cond>

Single-precision variant

Applies when ftype == 00.

FCCMP <Sn>, <Sm>, #<nzcv>, <cond>

Double-precision variant

Applies when ftype == 01.

FCCMP <Dn>, <Dm>, #<nzcv>, <cond>

Decode for all variants of this encoding

integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
datasize = 16;
else
  UNDEFINED;

bits(4) flags = nzcv;

Assembler symbols

<Dn>     Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>     Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<nzcv> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.

<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

CheckFPEnabled64();

bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2;
operand2 = V[m, datasize];
if ConditionHolds(cond) then
    flags = FPCompare(operand1, operand2, FALSE, FPCR[]);
PSTATE.<N,Z,C,V> = flags;

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C7.2.55  **FCCMPE**

Floating-point Conditional signaling Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the **PSTATE.** {N, Z, C, V} flags. If the condition does not pass then the **PSTATE.** {N, Z, C, V} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision variant

Applies when *ftype == 11*.

FCCMPE <Hn>, <Hm>, #<nzcv>, <cond>

### Single-precision variant

Applies when *ftype == 00*.

FCCMPE <Sn>, <Sm>, #<nzcv>, <cond>

### Double-precision variant

Applies when *ftype == 01*.

FCCMPE <Dn>, <Dm>, #<nzcv>, <cond>

### Decode for all variants of this encoding

```plaintext
decode = Integer(Rn);
decode = Integer(Rm);

decode = datatype;
case ftype of
  when '00' datatype = 32;
  when '01' datatype = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
data = 16;
  else
    UNDEFINED;

bits(4) flags = nzcv;
```

### Assembler symbols

- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<\text{n}> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<\text{m}> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<\text{s}> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<\text{m}> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<nzcv> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.

<\text{cond}> Is one of the standard conditions, encoded in the "cond" field in the standard way.

**Operation**

\begin{verbatim}
CheckFPEnabled64();

bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2;
operand2 = V[m, datasize];
if ConditionHolds(cond) then
    flags = FPCompare(operand1, operand2, TRUE, FPCR[]);
PSTATE.<\text{N,Z,C,V}> = flags;
\end{verbatim}

**Operational information**

The IEEE 754 standard specifies that the result of a comparison is precisely one of \(<, ==, >\) or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the PSTATE condition flags to \(N=0, Z=0, C=1,\) and \(V=1.\)

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C7.2.56 FCMEQ (register)

Floating-point Compare Equal (vector). This instruction compares each floating-point value from the first source SIMD&FP register, with the corresponding floating-point value from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(FEAT_FP16)

```
[31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 13 12 11 10 9] | 5 4 | 0 |
0 1 0 1 1 1 0 0 1 0 Rm 0 0 1 0 0 1 Rn Rd
0 1 0 1 1 E ac
```

Encoding

FCMEQ <Hd>, <Hn>, <Hm>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;

Scalar single-precision and double-precision

```
[31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 13 12 11 10 9] | 5 4 | 0 |
0 1 0 1 1 1 1 0 0 az 1 Rm 1 1 1 0 0 1 Rn Rd
0 1 0 1 1 E ac
```

Encoding

FCMEQ <V<d>, <V<n>, <V>m>
### Decode for this encoding

```plaintext
decode for this encoding
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

Case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

### Vector half precision

(FEAT_FP16)

[31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 13 12 11 10 9 5 4 0]

| 0 | Q | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 1 | 0 | 0 | 1 | Rn | Rd |
|---|---|---|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|---|---|
| U | E | ac |

### Encoding

FCMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

### Decode for this encoding

```plaintext
decode for this encoding
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

Case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

### Vector single-precision and double-precision

[31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 13 12 11 10 9 5 4 0]

| 0 | Q | 0 | 1 | 1 | 1 | 0 | 0 | az | 1 | Rm | 1 | 1 | 0 | 0 | 0 | 1 | Rn | Rd |
|---|---|---|---|---|---|---|---|----|---|----|---|---|---|---|---|---|---|---|---|
| U | E | ac |
**Encoding**

FCMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;
```

```c
case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

**Assembler symbols**

- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:
  - S when sz = 0
  - D when sz = 1
- `<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.
- `<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  - 4H when Q = 0
  - 8H when Q = 1
  For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
  - 2S when sz = 0, Q = 0
  - 4S when sz = 0, Q = 1
  - 2D when sz = 1, Q = 1
  The encoding sz = 1, Q = 0 is reserved.
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 & & IsMerging(fpcr);
bits(128) result = if merge then V[m, 128] else Zeros(128);

for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  if abs then
    element1 = FPAbs(element1);
    element2 = FPAbs(element2);
  case cmp of
    when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
    when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
    when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, 128] = result;
C7.2.57  FCMEQ (zero)

Floating-point Compare Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(FEAT_FP16)

```plaintext
[31 30 29 28][27 26 25 24][23 22 21 20][19 18 17 16][15 14 13 12][11 10 9 ]| 5 4 | 0 |
 0 1 0 1 1 1 1 0 | 1 1 1 1 0 0 | 0 1 1 0 | 1 0 | Rn | Rd
```

Encoding

FCMEQ <Hd>, <Hn>, #0.0

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Scalar single-precision and double-precision

```plaintext
[31 30 29 28][27 26 25 24][23 22 21 20][19 18 17 16][15 14 13 12][11 10 9 ]| 5 4 | 0 |
 0 1 0 1 1 1 1 0 | 1 1 1 1 0 0 | 0 1 1 0 | 1 0 | Rn | Rd
```

Encoding

FCMEQ <V<d>, <V<n>, #0.0

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
when '00' comparison = CompareOp_GT;
when '01' comparison = CompareOp_GE;
when '10' comparison = CompareOp_EQ;
when '11' comparison = CompareOp_LE;

Vector half precision

(FEAT_FP16)

[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 ]

| 0 | Q | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
|    |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| U | op |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Encoding
FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0

Decode for this encoding
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
when '00' comparison = CompareOp_GT;
when '01' comparison = CompareOp_GE;
when '10' comparison = CompareOp_EQ;
when '11' comparison = CompareOp_LE;

Vector single-precision and double-precision

[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 ]

| 0 | Q | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
|    |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| U | op |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Encoding
FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0

Decode for this encoding
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;

Assembler symbols

\(<Hd>\) Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<Hn>\) Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
\(<V>\) Is a width specifier, encoded in the "sz" field. It can have the following values:
\(S\) when \(sz = 0\)
\(D\) when \(sz = 1\)
\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
\(<n>\) Is the number of the SIMD&FP source register, encoded in the "Rn" field.
\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
\(4H\) when \(Q = 0\)
\(8H\) when \(Q = 1\)
For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
\(2S\) when \(sz = 0, Q = 0\)
\(4S\) when \(sz = 0, Q = 1\)
\(2D\) when \(sz = 1, Q = 1\)
The encoding \(sz = 1, Q = 0\) is reserved.
\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) zero = FPZero('0', esize);
barese: element;
boolean test_passed;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareLE(element, zero, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
        Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
C7.2.58 FCMGE (register)

Floating-point Compare Greater than or Equal (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than or equal to the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(FEAT_FP16)

```
[31 30 29 28][27 26 25 24][23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 1 1 1 1 0 0 1 0 | Rm 0 0 1 0 | 0 1 | Rd
U E ac
```

**Encoding**

FCMGE <Hd>, <Hn>, <Hm>

**Decode for this encoding**

```c
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Scalar single-precision and double-precision

```
[31 30 29 28][27 26 25 24][23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 1 1 1 1 1 0 0 sz 1 | Rm 1 1 1 0 | 0 1 | Rd
U E ac
```

**Encoding**

FCMGE <V>d>, <V>n>, <V>m>
Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Vector half precision

(Encoding)

FCMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Vector single-precision and double-precision

(Encoding)

Vector half precision

(Encoding)
Encoding

FCMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

Assembler symbols

- `<Hd>` is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<V>` is a width specifier, encoded in the "sz" field. It can have the following values:
  - `S` when `sz = 0`
  - `D` when `sz = 1`
- `<d>` is the number of the SIMD&FP destination register, in the "Rd" field.
- `<n>` is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<m>` is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` is the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  - `4H` when `Q = 0`
  - `8H` when `Q = 1`
  - `2S` when `sz = 0, Q = 0`
  - `4S` when `sz = 0, Q = 1`
  - `2D` when `sz = 1, Q = 1`

  For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
  - `2S` when `sz = 0, Q = 0`
  - `4S` when `sz = 0, Q = 1`
  - `2D` when `sz = 1, Q = 1`

  The encoding `sz = 1, Q = 0` is reserved.
- `<Vn>` is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation for all encodings

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;

boolean test_passed;

FPCRType fpcr = FPCR[];

boolean merge = elements == 1 && IsMerging(fpcr);

bits(128) result = if merge then V[m, 128] else Zeros(128);

for e = 0 to elements-1

    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    
    case cmp of
    
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);

    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, 128] = result;
C7.2.59   FCMGE (zero)

Floating-point Compare Greater than or Equal to zero (vector). This instruction reads each floating-point value in
the source SIMD&FP register and if the value is greater than or equal to zero sets every bit of the corresponding
vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector
element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results
in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(FeAT_Fp16)

\[
\begin{array}{cccccccccccccccccccccc}
\hline
0&1&1&1&1&1&1&1&0&1&1&1&1&0&0&0&1&1&0&1&0
\end{array}
\]

Encoding

FCMGE <Hd>, <Hn>, #0.0

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;

\[
\text{case op:U of}
\]

when '00' comparison = CompareOp_GT;
when '01' comparison = CompareOp_GE;
when '10' comparison = CompareOp_EQ;
when '11' comparison = CompareOp_LE;

Scalar single-precision and double-precision

\[
\begin{array}{cccccccccccccccccccccc}
\hline
0&1&1&1&1&1&1&1&0&1&1&1&1&0&0&0&1&1&0&1&0
\end{array}
\]

Encoding

FCMGE <V><d>, <V><n>, #0.0

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Vector half precision
(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

FCMGE <Vd>.<T>, <Vn>.<T>, #0.0

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

FCMGE <Vd>.<T>, <Vn>.<T>, #0.0

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in the "sz" field. It can have the following values:
  S when sz = 0
  D when sz = 1
<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  4H when Q = 0
  8H when Q = 1
For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
  2S when sz = 0, Q = 0
  4S when sz = 0, Q = 1
  2D when sz = 1, Q = 1
The encoding sz = 1, Q = 0 is reserved.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
case comparison of
    when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
    when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
    when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
    when CompareOp_LE test_passed = FPCompareLE(element, zero, FPCR[]);
    when CompareOp_LT test_passed = FPCompareLT(element, zero, FPCR[]);
  Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);
V[d, datasize] = result;
C7.2.60 FCMGT (register)

Floating-point Compare Greater than (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(Feat_FP16)

Encoding

FCMGTHd>, <Hn>, <Hm>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
when '000' cmp = CompareOp_EQ; abs = FALSE;
when '010' cmp = CompareOp_GE; abs = FALSE;
when '011' cmp = CompareOp_GE; abs = TRUE;
when '110' cmp = CompareOp_GT; abs = FALSE;
when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;

Scalar single-precision and double-precision

Encoding

FCMGTV<d>, V<n>, V<m>
Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

\[ \text{case E:U:ac of} \]
\[ \text{when '000' cmp = CompareOp_EQ; abs = FALSE;} \]
\[ \text{when '010' cmp = CompareOp_GE; abs = FALSE;} \]
\[ \text{when '011' cmp = CompareOp_GE; abs = TRUE;} \]
\[ \text{when '110' cmp = CompareOp_GT; abs = FALSE;} \]
\[ \text{when '111' cmp = CompareOp_GT; abs = TRUE;} \]
\[ \text{otherwise UNDEFINED;} \]

Vector half precision

(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 1 0 1 1 1 0 1 1 0</td>
<td>1 0 0 0 1 0</td>
<td>Rm</td>
<td>Rn</td>
</tr>
</tbody>
</table>

Encoding

FCMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

\[ \text{case E:U:ac of} \]
\[ \text{when '000' cmp = CompareOp_EQ; abs = FALSE;} \]
\[ \text{when '010' cmp = CompareOp_GE; abs = FALSE;} \]
\[ \text{when '011' cmp = CompareOp_GE; abs = TRUE;} \]
\[ \text{when '110' cmp = CompareOp_GT; abs = FALSE;} \]
\[ \text{when '111' cmp = CompareOp_GT; abs = TRUE;} \]
\[ \text{otherwise UNDEFINED;} \]

Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 1 0 1 1 1 0 1 sz 1</td>
<td>1 1 1 0 0 1</td>
<td>Rm</td>
<td>Rn</td>
</tr>
</tbody>
</table>
**Encoding**

FCMG <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

**Assembler symbols**

- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:
  - S when sz = 0
  - D when sz = 1
- `<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.
- `<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  - 4H when Q = 0
  - 8H when Q = 1
  For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
  - 2S when sz = 0, Q = 0
  - 4S when sz = 0, Q = 1
  - 2D when sz = 1, Q = 1
  The encoding sz = 1, Q = 0 is reserved.
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation for all encodings

    CheckFPAdvSIMDEnabled64();
    bits(datasize) operand1 = V[n, datasize];
    bits(datasize) operand2 = V[m, datasize];

    bits(esize) element1;
    bits(esize) element2;
    boolean test_passed;
    FPCRType fpcr = FPCR[];
    boolean merge = elements == 1 && IsMerging(fpcr);
    bits(128) result = if merge then V[m, 128] else Zeros(128);

    for e = 0 to elements-1
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
        if abs then
            element1 = FPAbs(element1);
            element2 = FPAbs(element2);
        case cmp of
            when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
            when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
            when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
            Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

    V[d, 128] = result;
C7.2.61  FCMGT (zero)

Floating-point Compare Greater than zero (vector). This instruction reads each floating-point value in the source
SIMD&FP register and if the value is greater than zero sets every bit of the corresponding vector element in the
destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the
destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results
in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see
Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
<td>1 1 1 1 0 0</td>
<td>0 1 1 0 0 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FCMG <Hd>, <Hn>, #0.0

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
<td>1 1 0 0 0 0 0 1 1 0 0 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FCMG <V<d>, <V<n>, #0.0

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

Comparison comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;

Vector half precision

(FEAT_FP16)

Encoding

FCMGT <Vd>.<T>, <Vn>.<T>, #0.0

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Comparison comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;

Vector single-precision and double-precision

Encoding

FCMGT <Vd>.<T>, <Vn>.<T>, #0.0

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:
  S when sz = 0
  D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  4H when Q = 0
  8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
  2S when sz = 0, Q = 0
  4S when sz = 0, Q = 1
  2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) zero = FPZero('0', esize);
bvsize(element); boolean test_passed;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  case comparison of
    when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
    when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
    when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
    when CompareOp_LE test_passed = FPCompareLE(element, zero, FPCR[]);
    when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);
V[d, datasize] = result;
C7.2.62 FCMLA (by element)

Floating-point Complex Multiply Accumulate (by element).

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with
the more significant element holding the imaginary part of the number and the less significant element holding the
real part of the number. Each element holds a floating-point value. It performs the following computation on
complex numbers from the first source register and the destination register with the specified complex number from
the second source register:

- Considering the complex number from the second source register on an Argand diagram, the number is
  rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a
    rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was
    a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination
  register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate
rounding.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results
in either a flag being set in FPSR or a synchronous exception being generated. For more information, see
Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_FCMA)

Encoding

Applies when size == 01.

FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>

Encoding

Applies when size == 10.

FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>

Decode for all variants of this encoding

if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index;
if size == '00' || size == '11' then UNDEFINED;
if size == '01' then index = UInt(H:L);
if size == '10' then index = UInt(H);
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
if size == '10' && (L == '1' || Q == '0') then UNDEFINED;
if size == '01' && H == '1' && Q == '0' then UNDEFINED;

**Assembler symbols**

`<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- **4H** when size = 01, Q = 0
- **8H** when size = 01, Q = 1
- **4S** when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 10, Q = 0.
- size = 11, Q = x.

`<Vn>` is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>` is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

`<Ts>` is an element size specifier, encoded in the "size" field. It can have the following values:

- **H** when size = 01
- **S** when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

`<index>` is the element index, encoded in the "size:H:L" field. It can have the following values:

- **H:L** when size = 01
- **H** when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

`<rotate>` is the rotation, encoded in the "rot" field. It can have the following values:

- **0** when rot = 00
- **90** when rot = 01
- **180** when rot = 10
- **270** when rot = 11

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bias(datasize) operand1 = V[n, datasize];
bias(datasize) operand2 = V[m, datasize];
bias(datasize) operand3 = V[d, datasize];
bias(datasize) result;
FPCRType fpcr = FPCR[];
for e = 0 to (elements DIV 2)-1
  bits(size) element1;
```
bits(esize) element2;
bites(esize) element3;
bites(esize) element4;
case rot of
  when '00'
    element1 = Elem[operand2, index*2, esize];
    element2 = Elem[operand1, e*2, esize];
    element3 = Elem[operand2, index*2+1, esize];
    element4 = Elem[operand1, e*2, esize];
  when '01'
    element1 = FPNeg(Elem[operand2, index*2+1, esize]);
    element2 = Elem[operand1, e*2+1, esize];
    element3 = Elem[operand2, index*2, esize];
    element4 = Elem[operand1, e*2+1, esize];
  when '10'
    element1 = FPNeg(Elem[operand2, index*2, esize]);
    element2 = Elem[operand1, e*2, esize];
    element3 = FPNeg(Elem[operand2, index*2+1, esize]);
    element4 = Elem[operand1, e*2, esize];
  when '11'
    element1 = Elem[operand2, index*2+1, esize];
    element2 = Elem[operand1, e*2+1, esize];
    element3 = FPNeg(Elem[operand2, index*2, esize]);
    element4 = Elem[operand1, e*2+1, esize];
Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, fpcr);
Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, fpcr);
V[d, datasize] = result;
C7.2.63   FCMLA

Floating-point Complex Multiply Accumulate.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with
the more significant element holding the imaginary part of the number and the less significant element holding the
real part of the number. Each element holds a floating-point value. It performs the following computation on the
corresponding complex number element pairs from the two source registers and the destination register:

- Considering the complex number from the second source register on an Argand diagram, the number is
  rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a
    rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was
    a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination
  register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate
rounding.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results
in either a flag being set in FPSR or a synchronous exception being generated. For more information, see
Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_FCMA)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>

Decode for this encoding

if !HaveFCADDext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then UNDEFINED;
if Q == '0' & size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
if !HaveFP16Ext() & esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1

The following encodings are reserved:

- size = 00, Q = x
- size = 11, Q = 0.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<rotate> Is the rotation, encoded in the "rot" field. It can have the following values:

- 0 when rot = 00
- 90 when rot = 01
- 180 when rot = 10
- 270 when rot = 11

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) element3;
bits(esize) element4;
FPCRType fpcr = FPCR[];
for e = 0 to (elements DIV 2)-1
  case rot of
    when '00'
      element1 = Elem[operand2, e*2, esize];
      element2 = Elem[operand1, e*2, esize];
      element3 = Elem[operand2, e*2+1, esize];
      element4 = Elem[operand1, e*2, esize];
    when '01'
      element1 = FPNeg(Elem[operand2, e*2+1, esize]);
      element2 = Elem[operand1, e*2+1, esize];
      element3 = Elem[operand2, e*2, esize];
      element4 = Elem[operand1, e*2, esize];
    when '10'
      element1 = FPNeg(Elem[operand2, e*2, esize]);
      element2 = Elem[operand1, e*2, esize];
      element3 = FPNeg(Elem[operand2, e*2+1, esize]);
      element4 = Elem[operand1, e*2, esize];
    when '11'
      element1 = Elem[operand2, e*2+1, esize];
      element2 = Elem[operand1, e*2+1, esize];
      element3 = FPNeg(Elem[operand2, e*2, esize]);
      element4 = Elem[operand1, e*2+1, esize];
   Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, fpcr);
   Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, fpcr);
```

V[d, datasize] = result;
C7.2.64 FCMLE (zero)

Floating-point Compare Less than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 1 0 1 1 1 1 0 0 0 1 1 0 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FCMLE <Hd>, <Hn>, #0.0

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

t = Int(Rn);
integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;

CompareOp comparison;

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 1 0 1 1 1 1 0 0 0 1 1 0 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FCMLE <V<d>, <V>n>, #0.0

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Vector half precision

(Encoding)

FCMLE <Vd>.<T>, <Vn>.<T>, #0.0

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Vector single-precision and double-precision

(Encoding)

FCMLE <Vd>.<T>, <Vn>.<T>, #0.0

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:
    S when sz = 0
    D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
    4H when Q = 0
    8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
    2S when sz = 0, Q = 0
    4S when sz = 0, Q = 1
    2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  case comparison of
    when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
    when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
    when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
    when CompareOp_LE test_passed = FPCompareLE(element, zero, FPCR[]);
    when CompareOp_LT test_passed = FPCompareLT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
C7.2.65 FCMLT (zero)

Floating-point Compare Less than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(FEAT_FP16)

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
[0 1 0 1 1 1 0 1 1 1 1 0 0 0 1 1 1 0 1 0] Rn Rd
```

**Encoding**

FCMLT <Hd>, <Hn>, #0.0

**Decode for this encoding**

```c
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

Scalar single-precision and double-precision

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
[0 1 0 1 1 1 0 1 1 1 1 0 0 0 0 1 1 1 0 1 0] Rn Rd
```

**Encoding**

FCMLT <V><d>, <V><n>, #0.0

**Decode for this encoding**

```c
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```
Vector half precision

(_FEAT_FP16)

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
```

<table>
<thead>
<tr>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

**Encoding**

FCMLT <Vd>.<T>, <Vn>.<T>, #0.0

**Decoding for this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp comparison = CompareOp_LT;
```

Vector single-precision and double-precision

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
```

<table>
<thead>
<tr>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

**Encoding**

FCMLT <Vd>.<T>, <Vn>.<T>, #0.0

**Decoding for this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp comparison = CompareOp_LT;
```

**Assembler symbols**

- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<V>` Is a width specifier, encoded in the "sz" field. It can have the following values:
  - `S` when `sz = 0`
  - `D` when `sz = 1`
- `<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0
8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation for all encodings

CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n, datasize];

bits(datasize) result;

bits(esize) zero = FPZero('0', esize);

bits(esize) element;

boolean test_passed;

for e = 0 to elements-1

  element = Elem[operand, e, esize];
  case comparison of
    when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
    when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
    when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
    when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
    when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
  
  Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
C7.2.66   FCMP

Floating-point quiet Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the PSTATE.{N, Z, C, V} flags.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant

Applies when $\text{ftype} = 11$ && $\text{opc} == 00$.

FCMP <Hn>, <Hm>

Half-precision, zero variant

Applies when $\text{ftype} = 11$ && $Rm == (00000)$ && $\text{opc} == 01$.

FCMP <Hn>, #0.0

Single-precision variant

Applies when $\text{ftype} = 00$ && $\text{opc} == 00$.

FCMP <Sn>, <Sm>

Single-precision, zero variant

Applies when $\text{ftype} = 00$ && $Rm == (00000)$ && $\text{opc} == 01$.

FCMP <Sn>, #0.0

Double-precision variant

Applies when $\text{ftype} = 01$ && $\text{opc} == 00$.

FCMP <Dn>, <Dm>

Double-precision, zero variant

Applies when $\text{ftype} = 01$ && $Rm == (00000)$ && $\text{opc} == 01$.

FCMP <Dn>, #0.0

Decode for all variants of this encoding

```plaintext
integer n = UInt(Rn);
integer m = UInt(Rm);   // ignored when opc<6> == '1'

integer datasize;
case ftype of
  when '00' datasize = 32;
```

| [31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 13 12|11 10 9 | 5 4 |3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | ftype | 1 | Rm | 0 | 0 | 1 | 0 | 0 | 0 | Rn | 0 | x | 0 | 0 | 0 |
| opc |
when '01' datasize = 64;
when '10' UNDEFINED;
when '11'
  if HaveFP16Ext() then
datasize = 16;
else
  UNDEFINED;

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');

Assembler symbols

<\Dn> For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
  For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<\Hn> For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
  For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<\Sn> For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
  For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPEnabled64();

bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2;
operand2 = if cmp_with_zero then FPZero('0', datasize) else V[m, datasize];
PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR[]);

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C7.2.67  FCMPE

Floating-point signaling Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the PSTATE.{N, Z, C, V} flags.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision variant

Applies when \( ftype = 11 \) \&\& \( opc = 10 \).

\[ 	ext{FCMPE} \quad <Hn>, \quad <Hm> \]

### Half-precision, zero variant

Applies when \( ftype = 11 \) \&\& \( Rm = (00000) \) \&\& \( opc = 11 \).

\[ 	ext{FCMPE} \quad <Hn>, \quad #0.0 \]

### Single-precision variant

Applies when \( ftype = 00 \) \&\& \( opc = 10 \).

\[ 	ext{FCMPE} \quad <Sn>, \quad <Sm> \]

### Single-precision, zero variant

Applies when \( ftype = 00 \) \&\& \( Rm = (00000) \) \&\& \( opc = 11 \).

\[ 	ext{FCMPE} \quad <Sn>, \quad #0.0 \]

### Double-precision variant

Applies when \( ftype = 01 \) \&\& \( opc = 10 \).

\[ 	ext{FCMPE} \quad <Dn>, \quad <Dm> \]

### Double-precision, zero variant

Applies when \( ftype = 01 \) \&\& \( Rm = (00000) \) \&\& \( opc = 11 \).

\[ 	ext{FCMPE} \quad <Dn>, \quad #0.0 \]

### Decode for all variants of this encoding

Integer \( n = \text{UInt}(Rn); \)

Integer \( m = \text{UInt}(Rm); \quad // \text{ignored when} \ opc<0> = '1' \)

Integer \( \text{datatype}; \)

Case \( ftype \) of

When '00' \( \text{datatype} = 32; \)
when '01' datasize = 64;
when '10' UNDEFINED;
when '11'  
  if HaveFP16Ext() then  
    datasize = 16;  
  else  
    UNDEFINED;  

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');

Assembler symbols

<Dn> For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Hn> For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Sn> For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPEnabled64();

bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2;
operand2 = if cmp_with_zero then FPZero('0', datasize) else V[m, datasize];
PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR[]);

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C7.2.68   FCSEL

Floating-point Conditional Select (scalar). This instruction allows the SIMD&FP destination register to take the value from either one or the other of two SIMD&FP source registers. If the condition passes, the first SIMD&FP source register value is taken, otherwise the second SIMD&FP source register value is taken.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 1 1 0</td>
<td>ftype 1</td>
<td>Rm</td>
<td>cond 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

**Half-precision variant**

Applies when ftype == 11.

FCSEL <Hd>, <Hn>, <Hm>, <cond>

**Single-precision variant**

Applies when ftype == 00.

FCSEL <Sd>, <Sn>, <Sm>, <cond>

**Double-precision variant**

Applies when ftype == 01.

FCSEL <Dd>, <Dn>, <Dm>, <cond>

**Decode for all variants of this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UNDEFINED;

**Assembler symbols**

<dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

**Operation**

```c
CheckFPEnabled64();
bits(datasize) result;
result = if ConditionHolds(cond) then V[n, datasize] else V[m, datasize];
V[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.69   FCVT

Floating-point Convert precision (scalar). This instruction converts the floating-point value in the SIMD&FP source register to the precision for the destination register data type using the rounding mode that is determined by the FPCR and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision to single-precision variant
Applies when ftype == 11 & opc == 00.
FCVT <Sd>, <Hn>

Half-precision to double-precision variant
Applies when ftype == 11 & opc == 01.
FCVT <Dd>, <Hn>

Single-precision to half-precision variant
Applies when ftype == 00 & opc == 11.
FCVT <Hd>, <Sn>

Single-precision to double-precision variant
Applies when ftype == 00 & opc == 01.
FCVT <Dd>, <Sn>

Double-precision to half-precision variant
Applies when ftype == 01 & opc == 11.
FCVT <Hd>, <Dn>

Double-precision to single-precision variant
Applies when ftype == 01 & opc == 00.
FCVT <Sd>, <Dn>

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer srcsize;
integer dstsize;
if ftype == opc then UNDEFINED;
case ftype of
when '00' srcsize = 32;
when '01' srcsize = 64;
when '10' UNDEFINED;
when '11' srcsize = 16;
case opc of
    when '00' dstsize = 32;
    when '01' dstsize = 64;
    when '10' UNDEFINED;
    when '11' dstsize = 16;

Assembler symbols

<DD> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dr> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

    CheckFPEnabled64();

    bits(srcsize) operand = V[n, srcsize];
    FPCRType fpcr = FPCR[];
    boolean merge = IsMerging(fpcr);
    bits(128) result = if merge then V[d, 128] else Zeros(128);

    Elem[result, 0, dstsize] = FPConvert(operand, fpcr, dstsize);

    V[d, 128] = result;
C7.2.70   FCVTAS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to a signed integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(FEAT_FP16)

[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9  |  5  4 |  0  ]
[ 0 1 0|1 1 1 1 0 0 1 1 1 0 0 1 1 0 0 1 1 0 0 1 0 | Rn   | Rd   ]

U

Encoding

FCVTAS <Hd>, <Hn>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9  |  5  4 |  0  ]
[ 0 1 0|1 1 1 1 0 0 | 0 | sz| 1 0 0 0 0 1 1 1 0 0 1 0 | Rn   | Rd   ]

U

Encoding

FCVTAS <V>d>, <V>n>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
Vector half precision

(FEAT_FP16)

```
FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

**Encoding**

FCVTAS <Vd>.<T>, <Vn>.<T>

**Decode for this encoding**

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 |
 0 1 1 1 0 0 1 1 1 0 0 1 1 0 0 1 0 0 0 | 1 1 1 0 0 1 0 1 0 Rn Rd
```

**Encoding**

FCVTAS <Vd>.<T>, <Vn>.<T>

**Decode for this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

**Assembler symbols**

- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> is a width specifier, encoded in the "sz" field. It can have the following values:
S when sz = 0
D when sz = 1

d is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
4H when Q = 0
8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
C7.2.71  **FCVTAS (scalar)**

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps on page A1-64*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf 0 0 1 1 1 1 0</td>
<td>type 1 0 0 1 0 0 0 0 0 0 0 0</td>
<td>Rn</td>
</tr>
</tbody>
</table>
```

### Half-precision to 32-bit variant

Applies when `sf == 0 && ftype == 11`.

FCVTAS `<Wd>`, `<Hn>`

### Half-precision to 64-bit variant

Applies when `sf == 1 && ftype == 11`.

FCVTAS `<Xd>`, `<Hn>`

### Single-precision to 32-bit variant

Applies when `sf == 0 && ftype == 00`.

FCVTAS `<Wd>`, `<Sn>`

### Single-precision to 64-bit variant

Applies when `sf == 1 && ftype == 00`.

FCVTAS `<Xd>`, `<Sn>`

### Double-precision to 32-bit variant

Applies when `sf == 0 && ftype == 01`.

FCVTAS `<Wd>`, `<Dn>`

### Double-precision to 64-bit variant

Applies when `sf == 1 && ftype == 01`.

FCVTAS `<Xd>`, `<Dn>`

### Decode for all variants of this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
```
when '00'
  fltsize = 32;
when '01'
  fltsize = 64;
when '10'
  UNDEFINED;
when '11'
  if HaveFP16Ext() then
    fltsize = 16;
  else
    UNDEFINED;

Assembler symbols

<Wd>  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>  Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>  Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>  Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, FALSE, fpcr, FPRounding_TIEAWAY, intsize);
X[d, intsize] = intval;

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is
dependent on the general-purpose register written by this instruction might be significantly delayed.
C7.2.72 FCVTAU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(Feat_FP16)

```
[31 30 29 28 27 26 25 24 | 31 30 29 28 27 26 25 24 | 5 4 | 0 ]
0 1 1 1 1 1 0 0 | 0 1 1 1 1 0 0 1 | 1 1 1 0 0 | 1 0 | Rd | Rn | U
```

Encoding

FCVTAU <Hd>, <Hn>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

```
[31 30 29 28 27 26 25 24 | 31 30 29 28 27 26 25 24 | 5 4 | 0 ]
0 1 1 1 1 1 1 0 | 0 1 1 1 1 0 0 | 1 1 1 0 0 | 1 0 | Rd | Rn | U
```

Encoding

FCVTAU <V>d>, <V<n>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

Vector half precision

(FEAT_FP16)

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Encoding

FCVTAU <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

Vector single-precision and double-precision

Encoding

FCVTAU <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Is a width specifier, encoded in the "sz" field. It can have the following values:

- S  when \( sz = 0 \)
- D  when \( sz = 1 \)

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H  when \( Q = 0 \)
- 8H  when \( Q = 1 \)

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S  when \( sz = 0, Q = 0 \)
- 4S  when \( sz = 0, Q = 1 \)
- 2D  when \( sz = 1, Q = 1 \)

The encoding \( sz = 1, Q = 0 \) is reserved.

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
   element = Elem[operand, e, esize];
   Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```
FCVTAU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|---|---|---|---|---|---|---|---|---|
| sf 0 0 1 1 1 0 | ftype 1 0 0 1 0 1 0 0 0 0 0 0 | Rn | Rd |

**Half-precision to 32-bit variant**

Applies when `sf == 0 && ftype == 11`.

FCVTAU <Wd>, <Hn>

**Half-precision to 64-bit variant**

Applies when `sf == 1 && ftype == 11`.

FCVTAU <Xd>, <Hn>

**Single-precision to 32-bit variant**

Applies when `sf == 0 && ftype == 00`.

FCVTAU <Wd>, <Sn>

**Single-precision to 64-bit variant**

Applies when `sf == 1 && ftype == 00`.

FCVTAU <Xd>, <Sn>

**Double-precision to 32-bit variant**

Applies when `sf == 0 && ftype == 01`.

FCVTAU <Wd>, <Dn>

**Double-precision to 64-bit variant**

Applies when `sf == 1 && ftype == 01`.

FCVTAU <Xd>, <Dn>

**Decode for all variants of this encoding**

```python
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
```

C7-2206
when '00'
  fltsize = 32;
when '01'
  fltsize = 64;
when '10'
  UNDEFINED;
when '11'
  if HaveFP16Ext() then
    fltsize = 16;
  else
    UNDEFINED;

**Assembler symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPEnabled64();
FPCRType fp cr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;
fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, TRUE, fp cr, FPRounding_TIEAWAY, intsize);
X[d, intsize] = intval;
```

**Operational information**

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C7.2.74 FCVTL, FCVTL2

Floating-point Convert to higher precision Long (vector). This instruction reads each element in a vector in the SIMD&FP source register, converts each value to double the precision of the source element using the rounding mode that is determined by the FPCR, and writes each result to the equivalent element of the vector in the SIMD&FP destination register.

Where the operation lengthens a 64-bit vector to a 128-bit vector, the FCVTL2 variant operates on the elements in the top 64 bits of the source register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Rn Rd

Encoding

FCVTL(2) <Vd>,<Ta>, <Vn>,<Tb>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16 <<UInt(sz);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0
[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "sz" field. It can have the following values:

4S when sz = 0
2D when sz = 1

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

4H when sz = 0, Q = 0
8H when sz = 0, Q = 1
2S when sz = 1, Q = 0
4S when sz = 1, Q = 1
Operation

```c
CheckFPAAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part, datasize];
bits(2*datasize) result;

for e = 0 to elements-1
    Elem[result, e, 2*esize] = FPConvert(Elem[operand, e, esize], FPCR[], 2 * esize);

V[d, 2*datasize] = result;
```
C7.2.75   FCVTMS (vector)

Floating-point Convert to Signed integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9 5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 0 0 1 1 0 0 1 1 1 0 0 1 1 0 1 0 0 1 1 0 0 1 1 0 1 0 0 1 1 0 1 0 1 0 0 1 1 0 1 0 1 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U o2 o1 Rn Rd

Encoding

FCVTMS <Hd>, <Hn>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9 5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 0 0 1 1 0 0 1 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 0 0 1 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U o2 o1 Rn Rd

Encoding

FCVTMS <V>d>, <V>n>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector half precision

(FEAT_FP16)

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] [11 10 9 ] [ 5  4 ] [ 0 ]
  0     0     0     0     1     1     1     0     0     1     1     0     0     1     1     0     Rn | Rd
  U     o2   o1
```

Encoding

FCVTMS <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] [11 10 9 ] [ 5  4 ] [ 0 ]
  0     0     0     0     1     1     1     0     0     1     0     0     0     1     1     0     Rn | Rd
  U     o2   o1
```

Encoding

FCVTMS <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V>  Is a width specifier, encoded in the "sz" field. It can have the following values:
S when sz = 0
D when sz = 1

<d>  Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n>  Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>  For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
4H when Q = 0
8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn>  Is the name of the SIMD&FP source register, encoded in the "Rn" field.

---

**Operation for all encodings**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);
V[d, 128] = result;
```
C7.2.76  FCVTMS (scalar)

Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![binary representation](image_url)

**Half-precision to 32-bit variant**
Applies when \( sf == 0 \) \& \( ftype == 11 \).

FCVTMS <Wd>, <Hn>

**Half-precision to 64-bit variant**
Applies when \( sf == 1 \) \& \( ftype == 11 \).

FCVTMS <Xd>, <Hn>

**Single-precision to 32-bit variant**
Applies when \( sf == 0 \) \& \( ftype == 00 \).

FCVTMS <Wd>, <Sn>

**Single-precision to 64-bit variant**
Applies when \( sf == 1 \) \& \( ftype == 00 \).

FCVTMS <Xd>, <Sn>

**Double-precision to 32-bit variant**
Applies when \( sf == 0 \) \& \( ftype == 01 \).

FCVTMS <Wd>, <Dn>

**Double-precision to 64-bit variant**
Applies when \( sf == 1 \) \& \( ftype == 01 \).

FCVTMS <Xd>, <Dn>

**Decode for all variants of this encoding**

```cpp
integer d = UInt(Rd);
integer n = UInt(Rn);
integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
```
case ftype of  
  when '00'  
    fltsize = 32;  
  when '01'  
    fltsize = 64;  
  when '10'  
    UNDEFINED;  
  when '11'  
    if HaveFP16Ext() then  
      fltsize = 16;  
    else  
      UNDEFINED;  
  rounding = FPDecodeRounding(rmode);

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPEnabled64();

FPCRTypet fpcr = FPCR[];
bits(fltsize) fltval;
b bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToF  
FPtoFixed(fltval, 0, FALSE, fpcr, rounding, intsize);
X[d, intsize] = intval;

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C7.2.77 FCVTMU (vector)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

(FEAT_FP16)

```
011 111100
 0 1 1 1 1 0 1 0 1 0 1 0  
  | 5 4 | 0 |
     U   o2  o1  Rd Rn  
```

**Encoding**

FCVTMU <Hd>, <Hn>

**Decode for this encoding**

```plaintext
if !HaveFP16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

```
011 111100
 0 1 1 1 0 0 sz 1 0 0 0 1 1 0 1 1 0  
  | 5 4 | 0 |
     U   o2  o1  Rd Rn  
```

**Encoding**

FCVTMU <V><d>, <V><n>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```
FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector half precision

(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FCVTMU <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FCVTMU <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in the "sz" field. It can have the following values:
- S when sz = 0
- D when sz = 1
</d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
</n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
</Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
</T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
- 4H when Q = 0
- 8H when Q = 1
For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1
The encoding sz = 1, Q = 0 is reserved.
</Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);
V[d, 128] = result;
```
C7.2.78 FCVTMU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision to 32-bit variant

Applies when \( sf == 0 \) && \( ftype == 11 \).

**FCVTMU \(<Wd>, <Hn>\)**

### Half-precision to 64-bit variant

Applies when \( sf == 1 \) && \( ftype == 11 \).

**FCVTMU \(<Xd>, <Hn>\)**

### Single-precision to 32-bit variant

Applies when \( sf == 0 \) && \( ftype == 00 \).

**FCVTMU \(<Wd>, <Sn>\)**

### Single-precision to 64-bit variant

Applies when \( sf == 1 \) && \( ftype == 00 \).

**FCVTMU \(<Xd>, <Sn>\)**

### Double-precision to 32-bit variant

Applies when \( sf == 0 \) && \( ftype == 01 \).

**FCVTMU \(<Wd>, <Dn>\)**

### Double-precision to 64-bit variant

Applies when \( sf == 1 \) && \( ftype == 01 \).

**FCVTMU \(<Xd>, <Dn>\)**

### Decode for all variants of this encoding

```c
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fsize;
FPRounding rounding;
```

```c
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18</th>
<th>17 16 15 14</th>
<th>13 12 11 10</th>
<th>9 8 7 6</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf 0 0 1 1 1 0</td>
<td>ftype 1 1 0 0 1 0 0 0 0 0</td>
<td>Rn Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

mode opcode
case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;
    rounding = FPDecodeRounding(rmode);

Assembler symbols

<\w>  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<\x>  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<\s>  Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<\h>  Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<\d>  Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding, intsize);
X[d, intsize] = intval;

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is
dependent on the general-purpose register written by this instruction might be significantly delayed.
C7.2.79 FCVTN, FCVTN2

Floating-point Convert to lower precision Narrow (vector). This instruction reads each vector element in the SIMD&FP source register, converts each result to half the precision of the source element, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The rounding mode is determined by the FPCR.

The FCVTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16 << UInt(sz);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0
[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

4H when sz = 0, Q = 0
8H when sz = 0, Q = 1
2S when sz = 1, Q = 0
4S when sz = 1, Q = 1

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "sz" field. It can have the following values:

4S when sz = 0
2D when sz = 1

**Operation**

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;

for e = 0 to elements-1
   Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR[], esize);

Vpart[d, part, datasize] = result;
C7.2.80  FCVTNS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

(FeAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FCVTNS <Hd>, <Hn>

**Decode for this encoding**

if !HaveFP16Ext() then UNDEFINED;

integer d = Uint(Rd);
integer n = Uint(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0 0 sz</td>
<td>1 0 0 0 0 1 1 0 1 0 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FCVTNS <V>d, <V>n>

**Decode for this encoding**

integer d = Uint(Rd);
integer n = Uint(Rn);

integer esize = 32 << Uint(sz);
integer datasize = esize;
integer elements = 1;
Vector half precision

(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 1 1 1 0 0 1 1 1 1 0 0 1 1 0 1 0 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FCVTNS <Vd>.<T>, <Vn>.<T>

**Decode for this encoding**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 1 1 1 0 0 sz 1 0 0 0 1 1 0 1 0 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FCVTNS <Vd>.<T>, <Vn>.<T>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

**Assembler symbols**

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<\V>  Is a width specifier, encoded in the "sz" field. It can have the following values:
  S  when sz = 0
  D  when sz = 1

<d>  Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n>  Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>  For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  4H  when Q = 0
  8H  when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
  2S  when sz = 0, Q = 0
  4S  when sz = 0, Q = 1
  2D  when sz = 1, Q = 1
The encoding sz = 1, Q = 0 is reserved.

<Vn>  Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
   element = Elem[operand, e, esize];
   Elem[result, e, esize] = FTPToFixed(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```
C7.2.81  FCVTNS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>type</td>
</tr>
</tbody>
</table>

**Half-precision to 32-bit variant**
Applies when \( sf == 0 \) \&\& \( ftype == 11 \).
FCVTNS <Wd>, <Hn>

**Half-precision to 64-bit variant**
Applies when \( sf == 1 \) \&\& \( ftype == 11 \).
FCVTNS <Xd>, <Hn>

**Single-precision to 32-bit variant**
Applies when \( sf == 0 \) \&\& \( ftype == 00 \).
FCVTNS <Wd>, <Sn>

**Single-precision to 64-bit variant**
Applies when \( sf == 1 \) \&\& \( ftype == 00 \).
FCVTNS <Xd>, <Sn>

**Double-precision to 32-bit variant**
Applies when \( sf == 0 \) \&\& \( ftype == 01 \).
FCVTNS <Wd>, <Dn>

**Double-precision to 64-bit variant**
Applies when \( sf == 1 \) \&\& \( ftype == 01 \).
FCVTNS <Xd>, <Dn>

**Decode for all variants of this encoding**

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
```
case ftype of
   when '00'
      fltsize = 32;
   when '01'
      fltsize = 64;
   when '10'
      UNDEFINED;
   when '11'
      if HaveFP16Ext() then
         fltsize = 16;
      else
         UNDEFINED;
   end;

   rounding = FPDecodeRounding(rmode);

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPEnabled64();

FPCRTypelfcr = FPCR[];
bv(fltsize) fltval;
bv(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding, intsize);
X[d, intsize] = intval;

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C7.2.82 FCVTNU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

(HEAD_FP16)

\[
\begin{array}{ccccccccccccccccccccccc}
| & \text{31} & \text{30} & \text{29} & \text{28} & \text{27} & \text{26} & \text{25} & \text{24} & \text{23} & \text{22} & \text{21} & \text{20} & \text{19} & \text{18} & \text{17} & \text{16} & \text{15} & \text{14} & \text{13} & \text{12} & \text{11} & \text{10} & \text{9} & | & \text{5} & \text{4} & | & \text{0} | \\
| 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & Rn & Rd |
\end{array}
\]

\textbf{Encoding}

FCVTNU <Hd>, <Hn>

\textbf{Decode for this encoding}

\begin{verbatim}
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
\end{verbatim}

Scalar single-precision and double-precision

\[
\begin{array}{ccccccccccccccccccccccc}
| & \text{31} & \text{30} & \text{29} & \text{28} & \text{27} & \text{26} & \text{25} & \text{24} & \text{23} & \text{22} & \text{21} & \text{20} & \text{19} & \text{18} & \text{17} & \text{16} & \text{15} & \text{14} & \text{13} & \text{12} & \text{11} & \text{10} & \text{9} & | & \text{5} & \text{4} & | & \text{0} | \\
| 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & Rn & Rd |
\end{array}
\]

\textbf{Encoding}

FCVTNU <V><d>, <V><n>

\textbf{Decode for this encoding}

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
\end{verbatim}
FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector half precision

(_FEAT_FP16)

Encoding

FCVTNU <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

Encoding

FCVTNU <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in the "sz" field. It can have the following values:
S when sz = 0
D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
4H when Q = 0
8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n, datasize];
bits(esize) element;

FPCRTypelfpcr = FPCR[];

boolean merge = elements == 1 && IsMerging(fpcr);

bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1

    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
C7.2.83   FCVTNU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Decoding for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
case ftype of  
  when '00' 
    fltsize = 32; 
  when '01' 
    fltsize = 64; 
  when '10' 
    UNDEFINED; 
  when '11'  
    if HaveFP16Ext() then 
      fltsize = 16; 
    else 
      UNDEFINED; 
  rounding = FPDecodeRounding(rmode);

**Assemble symbols**

<\Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<\Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<\Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding, intsize);
X[d, intsize] = intval;

**Operational information**

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C7.2.84 FCVTPS (vector)

Floating-point Convert to Signed integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

(FEAT_FP16)

| 31 30 29 28| 27 26 25 24| 23 22 21 20| 19 18 17 16| 15 14 13 12| 11 10 9 | 5 4 | 0 |
|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |

Encoding

FCVTPS <Hd>, <Hn>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

| 31 30 29 28| 27 26 25 24| 23 22 21 20| 19 18 17 16| 15 14 13 12| 11 10 9 | 5 4 | 0 |
|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

Encoding

FCVTPS <V<d>, <V<n>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector half precision

(�EAT_FFP16)

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | Rn | Rd |

Encoding

FCVTPS <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | Rn | Rd |

Encoding

FCVTPS <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> is a width specifier, encoded in the "sz" field. It can have the following values:
S when sz = 0
D when sz = 1

<d> is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
4H when Q = 0
8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bv<datasize> operand = V[n, datasize];

bv<esize> element;
FPCRTypedef fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bv<128> result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```
C7.2.85   FCVTPS (scalar)

Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 14 13 12 11 10 9 | 5 4 | 0 |
| sf 0 0 1 1 1 1 0 | type 1 0 1 0 0 0 0 0 0 | Rd |
```

### Half-precision to 32-bit variant
Applies when \(sf == 0 \&\& ftype == 11\).

\[
FCVTPS \langle Wd \rangle, \langle Hn \rangle
\]

### Half-precision to 64-bit variant
Applies when \(sf == 1 \&\& ftype == 11\).

\[
FCVTPS \langle Xd \rangle, \langle Hn \rangle
\]

### Single-precision to 32-bit variant
Applies when \(sf == 0 \&\& ftype == 00\).

\[
FCVTPS \langle Wd \rangle, \langle Sn \rangle
\]

### Single-precision to 64-bit variant
Applies when \(sf == 1 \&\& ftype == 00\).

\[
FCVTPS \langle Xd \rangle, \langle Sn \rangle
\]

### Double-precision to 32-bit variant
Applies when \(sf == 0 \&\& ftype == 01\).

\[
FCVTPS \langle Wd \rangle, \langle Dn \rangle
\]

### Double-precision to 64-bit variant
Applies when \(sf == 1 \&\& ftype == 01\).

\[
FCVTPS \langle Xd \rangle, \langle Dn \rangle
\]

### Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } & \text{d} = \text{UInt(Rd)}; \\
\text{integer } & \text{n} = \text{UInt(Rn)}; \\
\text{integer } & \text{intsize} = \text{if } \text{sf} = '1' \text{ then 64 else 32}; \\
\text{integer } & \text{fltsize}; \\
\text{FPRounding } & \text{rrounding};
\end{align*}
\]
case ftype of
  when '00'  
    fltsize = 32;
  when '01'  
    fltsize = 64;
  when '10'  
    UNDEFINED;
  when '11'  
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
    end if;
  end case;

rounding = FPDecodeRounding(rmode);

Assembler symbols

<\Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPEnabled64();

FPCRType fpcr = FPCR[];
b_bits(fltsize) fltval;
b_bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding, intsize);
X[d, intsize] = intval;

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C7.2.86 FCVTPU (vector)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

(Feat_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0 1 1 1 0 0 1 1 0 0 0 1 0 1 0</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**Encoding**

FCVTPU <Hd>, <Hn>

**Decode for this encoding**

if (!HaveFP16Ext()) then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0 0 1 1 0 0 1 1 0 0 0 1 0 1 0</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**Encoding**

FCVTPU <V><d>, <V><n>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector half precision

(_FEAT_FP16)

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9  | 5 4  | 0  |
|-----------|-----------|-----------|-----------|-----------|-------|-----|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | Rn | Rd |
| U | o2 | o1 |
```

Encoding

FCVTPU <Vb>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasetize = if Q == '1' then 128 else 64;
integer elements = datasetize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9  | 5 4  | 0  |
|-----------|-----------|-----------|-----------|-----------|-------|-----|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | sz | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | Rn | Rd |
| U | o2 | o1 |
```

Encoding

FCVTPU <Vb>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasetize = if Q == '1' then 128 else 64;
integer elements = datasetize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in the "sz" field. It can have the following values:
S when sz = 0
D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

</d> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: an arrangement specifier, encoded in the "Q" field. It can have the following values:
4H when Q = 0
8H when Q = 1

For the single-precision and double-precision variant: an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

</n> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation for all encodings

```c
CheckFPAdvSIMEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);
V[d, 128] = result;
```
C7.2.87 FCVTPU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18</th>
<th>17 16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Half-precision to 32-bit variant
Applies when sf == 0 && ftype == 11.
FCVTPU <Wd>, <Hn>

Half-precision to 64-bit variant
Applies when sf == 1 && ftype == 11.
FCVTPU <Xd>, <Hn>

Single-precision to 32-bit variant
Applies when sf == 0 && ftype == 00.
FCVTPU <Wd>, <Sn>

Single-precision to 64-bit variant
Applies when sf == 1 && ftype == 00.
FCVTPU <Xd>, <Sn>

Double-precision to 32-bit variant
Applies when sf == 0 && ftype == 01.
FCVTPU <Wd>, <Dn>

Double-precision to 64-bit variant
Applies when sf == 1 && ftype == 01.
FCVTPU <Xd>, <Dn>

Decode for all variants of this encoding

```plaintext
d = UInt(Rd);
n = UInt(Rn);
intsize = if sf == '1' then 64 else 32;
fltsize = FPRounding rounding;
```

Decoding
case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;
    rounding = FPDecodeRounding(rmode);

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPEnabled64();

FPCRTyp e fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;
fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding, intsize);
X[d, intsize] = intval;

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C7.2.88 FCVTXN, FCVTXN2

Floating-point Convert to lower precision Narrow, rounding to odd (vector). This instruction reads each vector element in the source SIMD&FP register, narrows each value to half the precision of the source element using the Round to Odd rounding mode, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Note

This instruction uses the Round to Odd rounding mode which is not defined by the IEEE 754-2008 standard. This rounding mode ensures that if the result of the conversion is inexact the least significant bit of the mantissa is forced to 1. This rounding mode enables a floating-point value to be converted to a lower precision format via an intermediate precision format while avoiding double rounding errors. For example, a 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value by first using this instruction to produce a 32-bit value and then using another instruction with the wanted rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

The FCVTXN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTXN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 0 0 sz 1 0 0 0 0 1 0 1 1 0 1 0</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Encoding

FCVTXN <Vb><d>, <Va><n>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '0' then UNDEFINED;
integer esize = 32;
integer datasize = esize;
integer elements = 1;
integer part = 0;

Vector

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 1 1 1 1 1 0 0 sz 1 0 0 0 0 1 0 1 1 0 1 0</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Encoding

FCVTXN{2} <Vd>.<Tb>, <Vn>.<Ta>
**Decode for this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '0' then UNDEFINED;
integer esize = 32;
integer datasize = 64;
integer elements = 2;
integer part = UInt(Q);
```

**Assembler symbols**

- `2` is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
  - [absent] when Q = 0
  - [present] when Q = 1

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- `<Tb>` is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
  - `2S` when sz = 1, Q = 0
  - `4S` when sz = 1, Q = 1
  The encoding sz = 0, Q = x is reserved.

- `<Vn>` is the name of the SIMD&FP source register, encoded in the "Rn" field.

- `<Ta>` is an arrangement specifier, encoded in the "sz" field. It can have the following values:
  - `2D` when sz = 1
  The encoding sz = 0 is reserved.

- `<Vb>` is the destination width specifier, encoded in the "sz" field. It can have the following values:
  - `S` when sz = 1
  The encoding sz = 0 is reserved.

- `<d>` is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- `<Va>` is the source width specifier, encoded in the "sz" field. It can have the following values:
  - `D` when sz = 1
  The encoding sz = 0 is reserved.

- `<n>` is the number of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

```
CheckFPAdvSIMDEnabled64();

bits(2*datasize) operand = V[n, 2*datasize];
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bihets(128) result = if merge then V[d, 128] else Zeros(128);
for e = 0 to elements-1
  Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], fpcr, FPRounding_ODD, esize);
if merge then
  V[d, 128] = result;
```
else
    \text{Vpart}[d, \text{part}, \text{datasize}] = \text{Elem}[\text{result}, 0, \text{datasize}];
C7.2.89   FCVTZS (vector, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22</th>
<th>19 18 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
<td>!=0000</td>
<td>immh 1 1 1 1 1 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FCVTZS <V><d>, <V><n>, #<fbits>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;

Vector

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22</th>
<th>19 18 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
<td>!=0000</td>
<td>immh 1 1 1 1 1 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FCVTZS <Vd>.<T>, <Vn>.<T>, #<fbits>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh IN {'000x'} then "Advanced SIMD modified immediate";
if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = esize;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;

Assembler symbols

\(<V>\) Is a width specifier, encoded in the "immh" field. It can have the following values:
H when immh = 001x
S when immh = 01xx
D when immh = 1xxx
The encoding immh = 000x is reserved.

\(<d>\) Is the number of the SIMD&FP destination register, in the "Rd" field.

\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:
4H when immh = 001x, Q = 0
8H when immh = 001x, Q = 1
2S when immh = 01xx, Q = 0
4S when immh = 01xx, Q = 1
2D when immh = 1xxx, Q = 1
See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.
The following encodings are reserved:
• immh = 0001, Q = x.
• immh = 1xxx, Q = 0.

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<fbits>\) For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded
in the "immh:immb" field. It can have the following values:
(32-Uint(immh:immb)) when immh = 001x
(64-Uint(immh:immb)) when immh = 01xx
(128-Uint(immh:immb)) when immh = 1xxx
The encoding immh = 000x is reserved.
For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded
in the "immh:immb" field. It can have the following values:
(32-Uint(immh:immb)) when immh = 001x
(64-Uint(immh:immb)) when immh = 01xx
(128-Uint(immh:immb)) when immh = 1xxx
See Advanced SIMD modified immediate on page C4-632 when immh = 0000.
The encoding immh = 0001 is reserved.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, fpcr, rounding, esize);
V[d, 128] = result;
C7.2.90   FCVTZS (vector, integer)

Floating-point Convert to Signed integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

(_FEAT_FP16)

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
| 0 1 | 1 1 1 1 0 1 1 1 1 0 0 1 1 0 1 1 0 | Rn | Rd |
```

**Encoding**

FCVTZS <Hd>, <Hn>

**Decode for this encoding**

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
| 0 1 | 1 1 1 1 0 1 1 1 1 0 0 1 1 0 1 1 0 | Rn | Rd |
```

**Encoding**

FCVTZS <V>d>, <V>n>

**Decode for this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```
FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector half precision

(FEATURE_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0 Q 0 1 1 1 0 1 1 1 1 0 1 1 0 0 1 1 0 1 1 0]</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Encoding

FCVTZS <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0 Q 0 1 1 1 0 1 sz 1 0 0 0 1 1 0 1 1 0]</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Encoding

FCVTZS <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in the "sz" field. It can have the following values:
S when sz = 0
0 when sz = 1
<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
4H when Q = 0
8H when Q = 1
For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1
The encoding sz = 1, Q = 0 is reserved.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRTypedefpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
C7.2.91   FCVTZS (scalar, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar). This instruction converts the
floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point signed integer using the Round
towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception
results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see
Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and
Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Half-precision to 32-bit variant

Applies when $sf == 0 && ftype == 11$.

FCVTZS <Wd>, <Hn>, #<fbits>

Half-precision to 64-bit variant

Applies when $sf == 1 && ftype == 11$.

FCVTZS <Xd>, <Hn>, #<fbits>

Single-precision to 32-bit variant

Applies when $sf == 0 && ftype == 00$.

FCVTZS <Wd>, <Sn>, #<fbits>

Single-precision to 64-bit variant

Applies when $sf == 1 && ftype == 00$.

FCVTZS <Xd>, <Sn>, #<fbits>

Double-precision to 32-bit variant

Applies when $sf == 0 && ftype == 01$.

FCVTZS <Wd>, <Dn>, #<fbits>

Double-precision to 64-bit variant

Applies when $sf == 1 && ftype == 01$.

FCVTZS <Xd>, <Dn>, #<fbits>

Decode for all variants of this encoding

```c
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
```
when '00' fltsize = 32;
when '01' fltsize = 64;
when '10' UNDEFINED;
when '11'
  if HaveFP16Ext() then
    fltsize = 16;
  else
    UNDEFINED;
if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<fbits> For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".
For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

Operation

CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;
fltval = V[n, fltsize];
intval = FPtoFixed(fltval, fracbits, FALSE, fpcr, FPRounding_ZERO, intsize);
X[d, intsize] = intval;

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C7.2.92  FCVTZS (scalar, integer)

Floating-point Convert to Signed integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| sf  | 0   | 0   | 1   | 1   | 1   | 1   | 0   | Rn |
| ftype | 1   | 1   | 0   | 0   | 0   | 0   | 0   | Rd |

Half-precision to 32-bit variant
Applies when sf == 0 && ftype == 11.
FCVTZS <Wd>, <Hn>

Half-precision to 64-bit variant
Applies when sf == 1 && ftype == 11.
FCVTZS <Xd>, <Hn>

Single-precision to 32-bit variant
Applies when sf == 0 && ftype == 00.
FCVTZS <Wd>, <Sn>

Single-precision to 64-bit variant
Applies when sf == 1 && ftype == 00.
FCVTZS <Xd>, <Sn>

Double-precision to 32-bit variant
Applies when sf == 0 && ftype == 01.
FCVTZS <Wd>, <Dn>

Double-precision to 64-bit variant
Applies when sf == 1 && ftype == 01.
FCVTZS <Xd>, <Dn>

Decode for all variants of this encoding
```integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
n integer fltsize;
FPRounding rounding;
```
case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;
    rounding = FPDecodeRounding(rmode);

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<HN> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPEnabled64();

FPCRTYPE fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;
fltval = V[n, fltsize];
intval = FPtoFixed(fltval, 0, FALSE, fpcr, rounding, intsize);
X[d, intsize] = intval;

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C7.2.93   FCVTZU (vector, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

**Scalar**

![Scalar Encoding Diagram]

**Encoding**

FCVTZU <V><d>, <V><n>, #<fbits>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datatize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;

**Vector**

![Vector Encoding Diagram]

**Encoding**

FCVTZU <Vd>.<T>, <Vn>.<T>, #<fbits>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datatize = esize;
integer elements = datatize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');

FPFrounding rounding = FPFrounding_ZERO;

Assembler symbols

<\v> Is a width specifier, encoded in the "immh" field. It can have the following values:

H when immh = 001x
S when immh = 01xx
D when immh = 1xxx

The encoding immh = 000x is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<\v> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

4H when immh = 001x, Q = 0
8H when immh = 001x, Q = 1
2S when immh = 01xx, Q = 0
4S when immh = 01xx, Q = 1
2D when immh = 1xxx, Q = 1

See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.

The following encodings are reserved:

• immh = 0001, Q = x.
• immh = 1xxx, Q = 0.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in the "immh:immb" field. It can have the following values:

(32-Uint(immh:immb)) when immh = 001x
(64-Uint(immh:immb)) when immh = 01xx
(128-Uint(immh:immb)) when immh = 1xxx

The encoding immh = 0000 is reserved.

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in the "immh:immb" field. It can have the following values:

(32-Uint(immh:immb)) when immh = 001x
(64-Uint(immh:immb)) when immh = 01xx
(128-Uint(immh:immb)) when immh = 1xxx

See Advanced SIMD modified immediate on page C4-632 when immh = 0000.

The encoding immh = 0001 is reserved.

Operation for all encodings

CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n, datasize];

bits(esize) element;

FPCRType fpcr = FPCR[];

boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPToFixed(element, fracbits, signed, fpcr, rounding, esize);
V[d, 128] = result;
C7.2.94 FCVTZU (vector, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

(_FEAT_FP16)

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------| | | | | | | |
|0 1 1 1 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 0 1 1 0 | Rn | Rd |
```

**Encoding**

FCVTZU <Hd>, <Hn>

**Decode for this encoding**

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------| | | | | | | |
|0 1 1 1 1 1 0 1 | sz| 1 0 0 0 | 0 1 1 0 1 1 0 | Rn | Rd |
```

**Encoding**

FCVTZU <V><d>, <V><n>

**Decode for this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 < UInt(sz);
integer datasize = esize;
integer elements = 1;
```
**Vector half precision**

(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FCVTZU <Vd>.<T>, <Vn>.<T>

**Decode for this encoding**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

**Vector single-precision and double-precision**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FCVTZU <Vd>.<T>, <Vn>.<T>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

**Assembler symbols**

<hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in the "sz" field. It can have the following values:
S when sz = 0
D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
4H when Q = 0
8H when Q = 1
For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1
The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
b整数(datasize) operand = V[n, datasize];

整数(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
b整数(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);
V[d, 128] = result;
```
C7.2.95 FCVTZU (scalar, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Half-precision to 32-bit variant
Applies when \( sf == 0 \) && \( ftype == 11 \).
FCVTZU <Wd>, <Hn>, #<fbits>

Half-precision to 64-bit variant
Applies when \( sf == 1 \) && \( ftype == 11 \).
FCVTZU <Xd>, <Hn>, #<fbits>

Single-precision to 32-bit variant
Applies when \( sf == 0 \) && \( ftype == 00 \).
FCVTZU <Wd>, <Sn>, #<fbits>

Single-precision to 64-bit variant
Applies when \( sf == 1 \) && \( ftype == 00 \).
FCVTZU <Xd>, <Sn>, #<fbits>

Double-precision to 32-bit variant
Applies when \( sf == 0 \) && \( ftype == 01 \).
FCVTZU <Wd>, <Dn>, #<fbits>

Double-precision to 64-bit variant
Applies when \( sf == 1 \) && \( ftype == 01 \).
FCVTZU <Xd>, <Dn>, #<fbits>

Decode for all variants of this encoding
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of

when '00' fltsize = 32;
when '01' fltsize = 64;
when '10' UNDEFINED;
when '11'
  if HaveFP16Ext() then
    fltsize = 16;
  else
    UNDEFINED;
if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is
  the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded
  as 64 minus "scale".

  For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is
  the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded
  as 64 minus "scale".

Operation

CheckFPEnabled64();

FPCRTypelfpc = FPCR[];
bists(fltsize) fltval;
bists(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, fracbits, TRUE, fpc, FPRounding_ZERO, intsize);
X[d, intsize] = intval;

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is
dependent on the general-purpose register written by this instruction might be significantly delayed.
C7.2.96 FCVTZU (scalar, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision to 32-bit variant
Applies when \( sf == 0 \) \&\& \( ftype == 11 \).
FCVTZU \(<Wd>, <Hn>\)

Half-precision to 64-bit variant
Applies when \( sf == 1 \) \&\& \( ftype == 11 \).
FCVTZU \(<Xd>, <Hn>\)

Single-precision to 32-bit variant
Applies when \( sf == 0 \) \&\& \( ftype == 00 \).
FCVTZU \(<Wd>, <Sn>\)

Single-precision to 64-bit variant
Applies when \( sf == 1 \) \&\& \( ftype == 00 \).
FCVTZU \(<Xd>, <Sn>\)

Double-precision to 32-bit variant
Applies when \( sf == 0 \) \&\& \( ftype == 01 \).
FCVTZU \(<Wd>, <Dn>\)

Double-precision to 64-bit variant
Applies when \( sf == 1 \) \&\& \( ftype == 01 \).
FCVTZU \(<Xd>, <Dn>\)

Decode for all variants of this encoding

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer intsize = if sf == '1' then 64 else 32;
integer fltsize; // PPRounding rounding;
```
case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  rounding = FPDecodeRounding(rmode);

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding, intsize);
X[d, intsize] = intval;

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is
dependent on the general-purpose register written by this instruction might be significantly delayed.
C7.2.97  FDIV (vector)

Floating-point Divide (vector). This instruction divides the floating-point values in the elements in the first source SIMD&FP register, by the floating-point values in the corresponding elements in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(FEAT_FP16)

```
<table>
<thead>
<tr>
<th>31  30  29  28</th>
<th>27  26  25  24</th>
<th>23  22  21  20</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**Encoding**

FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Single-precision and double-precision

```
<table>
<thead>
<tr>
<th>31  30  29  28</th>
<th>27  26  25  24</th>
<th>23  22  21  20</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>sz</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**Encoding**

FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 SHIFT UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```
Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
4H when Q = 0
8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPDiv(element1, element2, FPCR[]);

V[d, datasize] = result;
C7.2.98   FDIV (scalar)

Floating-point Divide (scalar). This instruction divides the floating-point value of the first source SIMD&FP register by the floating-point value of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision variant

Applies when $ftype = 11$.

FDIV $<Hd>$, $<Hn>$, $<Hm>$

### Single-precision variant

Applies when $ftype = 00$.

FDIV $<Sd>$, $<Sn>$, $<Sm>$

### Double-precision variant

Applies when $ftype = 01$.

FDIV $<Dd>$, $<Dn>$, $<Dm>$

### Decode for all variants of this encoding

integer $d = $UInt(Rd);
integer $n = $UInt(Rn);
integer $m = $UInt(Rm);

integer $esize$;

$case ftype of$

$when '00' esize = 32;$
$when '01' esize = 64;$
$when '10' UNDEFINED;$
$when '11'$

$if HaveFP16Ext() then$

$esize = 16;$
$else$

UNDEFINED;

### Assembler symbols

$<Dd>$  Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

$<Dn>$  Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

$<Dm>$  Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

$<Hd>$  Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
A64 Advanced SIMD and Floating-point Instruction Descriptions
C7.2 Alphabetical list of A64 Advanced SIMD and floating-point instructions

<hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

Elem[result, 0, esize] = FPDiv(operand1, operand2, FPCR[]);
V[d, 128] = result;
```
C7.2.99 FJCVTZS

Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD&FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register. If the result is too large to be represented as a signed 32-bit integer, then the result is the integer modulo $2^{32}$, as held in a 32-bit signed integer.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Double-precision to 32-bit

(FEAT_JSCVT)

```
| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 16 15 14 13 12 | 11 10 9 | 5 4 | 0 |
| 0 0 0 0 | 1 1 1 1 | 0 0 1 0 | 1 1 1 1 | 0 0 0 0 0 0 | | | | Rn | Rd |
| sf | ftype | mode | opcode |
```

**Encoding**

FJCVTZS <Wd>, <Dn>

**Decode for this encoding**

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- if !HaveFJCVTZSExt() then UNDEFINED;

**Assembler symbols**

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
FPCRType fpcr = FPCR[];
bounds(64) fltval;
bounds(32) intval;

bit z;
fltval = V[n, 64];
(intval, z) = FPToFixedJS(fltval, fpcr, TRUE, 32);
PSTATE.<N,Z,C,V> = '0';z:'00';
X[d, 32] = intval;
```
C7.2.100   FMADD

Floating-point fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, adds the product to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| [31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 | 10 9 | 5 4 | 0 |
|---|---|---|---|---|---|---|
| 0 0 0 1 1 1 1 | ftype 0 | Rm | 0 | Ra | Rn | Rd |

**Half-precision variant**

Applies when $ftype = 11$.

FMADD <Hd>, <Hn>, <Hm>, <Ha>

**Single-precision variant**

Applies when $ftype = 00$.

FMADD <Sd>, <Sn>, <Sm>, <Sa>

**Double-precision variant**

Applies when $ftype = 01$.

FMADD <Dd>, <Dn>, <Dm>, <Da>

**Decode for all variants of this encoding**

```plaintext
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

**Assembler symbols**

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Da> Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Ha> Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Sa> Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

**Operation**

    CheckFPEnabled64();
    bits(esize) operanda = V[a, esize];
    bits(esize) operand1 = V[n, esize];
    bits(esize) operand2 = V[m, esize];

    FPCRType fpcr = FPCR[];
    boolean merge = IsMerging(fpcr);
    bits(128) result = if merge then V[a, 128] else Zeros(128);

    Elem[result, 0, esize] = FPMulAdd(operanda, operand1, operand2, fpcr);
    V[d, 128] = result;
C7.2.101   FMAX (vector)

Floating-point Maximum (vector). This instruction compares corresponding vector elements in the two source
SIMD&FP registers, places the larger of each of the two floating-point values into a vector, and writes the vector to
the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results
in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see
Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

Half-precision
(FEAT_FP16)

Encoding
FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Single-precision and double-precision

Encoding
FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean pair = (U == '1');
boolean minimum = (O1 == '1');

**Assembler symbols**

\(<V_d>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

\(<V_n>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<V_m>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];

  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);
V[d, datasize] = result;
C7.2.102  FMAX (scalar)

Floating-point Maximum (scalar). This instruction compares the two source SIMD&FP registers, and writes the larger of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant
Applies when ftype == 11.
FMAX <Hd>, <Hn>, <Hm>

Single-precision variant
Applies when ftype == 00.
FMAX <Sd>, <Sn>, <Sm>

Double-precision variant
Applies when ftype == 01.
FMAX <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

Elem[result, 0, esize] = FPMax(operand1, operand2, fpcr);
V[d, 128] = result;
```
C7.2.103  **FMAXNM (vector)**

Floating-point Maximum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the larger of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision

(FEATURE_FP16)

```
[31 30 29 28][27 26 25 24][23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 Q 0 1 1 0 0 1 0 | Rm 0 0 0 0 1 | Rn Rd
```

**Encoding**

FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```java
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

### Single-precision and double-precision

```
[31 30 29 28][27 26 25 24][23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 Q 0 1 1 0 0 0 sz 1 | Rm 1 1 0 0 1 | Rn Rd
```

**Encoding**

FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (O1 == '1');

**Assembler symbols**

*<Vd>* Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

*<T>* For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- **4H** when Q = 0
- **8H** when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- **2S** when sz = 0, Q = 0
- **4S** when sz = 0, Q = 1
- **2D** when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

*<Vn>* Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

*<Vm>* Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = ELEM[concat, 2*e, esize];
    element2 = ELEM[concat, (2*e)+1, esize];
  else
    element1 = ELEM[operand1, e, esize];
    element2 = ELEM[operand2, e, esize];

  if minimum then
    ELEM[result, e, esize] = FPMINNum(element1, element2, FPCR[]);
  else
    ELEM[result, e, esize] = FPMAXNum(element1, element2, FPCR[]);

V[d, datasize] = result;
### C7.2.104  FMAXNM (scalar)

Floating-point Maximum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the larger of the two floating-point values to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

#### Half-precision variant
Applies when $ftype == 11$.

```
FMAXNM <Hd>, <Hn>, <Hm>
```

#### Single-precision variant
Applies when $ftype == 00$.

```
FMAXNM <Sd>, <Sn>, <Sm>
```

#### Double-precision variant
Applies when $ftype == 01$.

```
FMAXNM <Dd>, <Dn>, <Dm>
```

#### Decode for all variants of this encoding

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;

Assembler symbols

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
```
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

Elem[result, 0, esize] = FPMaxNum(operand1, operand2, fpcr);
V[d, 128] = result;
```
C7.2.105   FMAXNMP (scalar)

Floating-point Maximum Number of Pair of elements (scalar). This instruction compares two vector elements in the
source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP
register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results
in either a flag being set in FPSR or a synchronous exception being generated. For more information, see
Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(_FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1</td>
<td>1 1 1 1</td>
<td>1 0 0 0</td>
<td>1 1 1 0</td>
<td>0 1 0 0</td>
<td>1 0</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Encoding

FMAXNMP <V><d>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;

Single-precision and double-precision

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|-----------|-----------|-----------|-----------|-----------|-----------|-----|
| 0 1 1 | 1 1 1 1 | 0 0 1 1 0 0 | 0 1 1 0 | 0 1 0 0 | 1 0 | Rn | Rd |

Encoding

FMAXNMP <V><d>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
Assembler symbols

\(<V>\) For the half-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

- \(H\) when \(sz = 0\)
- The encoding \(sz = 1\) is reserved.

For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

- \(S\) when \(sz = 0\)
- \(D\) when \(sz = 1\)

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<T>\) For the half-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

- \(2H\) when \(sz = 0\)
- The encoding \(sz = 1\) is reserved.

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

- \(2S\) when \(sz = 0\)
- \(2D\) when \(sz = 1\)

Operation for all encodings

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FMAXNUM, operand, esize, FALSE);
```
C7.2.106  FMAXNMP (vector)

Floating-point Maximum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(FeaT_FP16)

\[
\begin{array}{cccccccc|c|c|c}
\hline
0 & Q & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
U & 0 & 0 & 0 & 0 & 0 & 1 & Rn & Rd & 0
\end{array}
\]

Encoding

FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');

Single-precision and double-precision

\[
\begin{array}{cccccccc|c|c|c}
\hline
0 & Q & 1 & 0 & 1 & 1 & 1 & 0 & 0 & sz & 1 \\
U & 0 & 1 & 0 & 0 & 0 & 1 & Rm & Rd & 0
\end{array}
\]

Encoding

FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Assembler symbols

<A6>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>
For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H  when Q = 0
8H  when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S  when sz = 0, Q = 0
4S  when sz = 0, Q = 1
2D  when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn>
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
if pair then
  element1 = Elem[concat, 2*e, esize];
  element2 = Elem[concat, (2*e)+1, esize];
else
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
if minimum then
  Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
else
  Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);
V[d, datasize] = result;
C7.2.107  FMAXNMV

Floating-point Maximum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(FEAT_FP16)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Rn  Rd

Encoding

FMAXNMV <V><d>, <Vn>.<T>

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Rn  Rd

Encoding

FMAXNMV <V><d>, <Vn>.<T>

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;    // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```
Assembler symbols

<\V> For the half-precision variant: is the destination width specifier, \(H\).
- For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:
  - \(S\) when \(sz = 0\)
  - The encoding \(sz = 1\) is reserved.

<\d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<\Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<\T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  - \(4H\) when \(Q = 0\)
  - \(8H\) when \(Q = 1\)
- For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:
  - \(4S\) when \(Q = 1\), \(sz = 0\)
  - The following encodings are reserved:
    - \(Q = 0\), \(sz = x\).
    - \(Q = 1\), \(sz = 1\).

Operation for all encodings

\[
\begin{align*}
\text{CheckFPAdvSIMDEnabled64}(); \\
\text{bits(datasize)} \text{ operand} = \text{V}[n, \text{datasize}]; \\
\text{V}[d, \text{esize}] = \text{Reduce}(\text{ReduceOp_FMAXNUM}, \text{operand}, \text{esize}, \text{FALSE});
\end{align*}
\]
### C7.2.108 FMAXP (scalar)

Floating-point Maximum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

#### Half-precision

(Feat_FP16)

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0 0 0</td>
<td>sz 1 1 0 0 0 0 1 1 1 1 1 0</td>
<td>Rn</td>
</tr>
</tbody>
</table>
```

#### Encoding

FMAXP <V><d>, <Vn>.<T>

#### Decode for this encoding

```plaintext
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;
```

#### Single-precision and double-precision

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 0 0 0</td>
<td>sz 1 1 0 0 0 0 1 1 1 1 1 0</td>
<td>Rn</td>
</tr>
</tbody>
</table>
```

#### Encoding

FMAXP <V><d>, <Vn>.<T>

#### Decode for this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
```
Assembler symbols

\(<V>\) For the half-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:
\(H\) when \(sz = 0\)
The encoding \(sz = 1\) is reserved.

For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:
\(S\) when \(sz = 0\)
\(D\) when \(sz = 1\)

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<T>\) For the half-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:
\(2H\) when \(sz = 0\)
The encoding \(sz = 1\) is reserved.

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:
\(2S\) when \(sz = 0\)
\(2D\) when \(sz = 1\)

Operation for all encodings

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

V[d, esize] = Reduce(ReduceOp_FMAX, operand, esize);
```
C7.2.109 FMAXP (vector)

Floating-point Maximum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the larger of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(FEAT_FP16)

\[
\begin{array}{cccccccccc|c|c}
\text{|31 30 29 28|27 26 25 24|23 22 21 20|} & \text{16|15 14 13 12|11 10 9|} & \text{5 4|} & \text{0|} \\
\text{0|Q|1|0 1 1 0 0 1 0|} & \text{Rm} & 0 0 1 1 0 1 & \text{Rn} & \text{Rd} \\
\text{U} & 0 1
\end{array}
\]

Encoding

FMAXP \text{<Vd>.<T>, <Vn>.<T>, <Vm>.<T>}

Decode for this encoding

\[
\text{if !HaveFP16Ext() then UNDEFINED;}
\]
\[
\text{integer d = UInt(Rd);} \\
\text{integer n = UInt(Rn);} \\
\text{integer m = UInt(Rm);} \\
\text{integer esize = 16;} \\
\text{integer datasize = if Q == '1' then 128 else 64;} \\
\text{integer elements = datasize DIV esize;} \\
\text{boolean pair = (U == '1');} \\
\text{boolean minimum = (o1 == '1');}
\]

Single-precision and double-precision

\[
\begin{array}{cccccccccc|c|c}
\text{|31 30 29 28|27 26 25 24|23 22 21 20|} & \text{16|15 14 13 12|11 10 9|} & \text{5 4|} & \text{0|} \\
\text{0|Q|1|0 1 1 1 0 0|} & \text{sz|1|} & \text{1 1 1 1 0 1|} & \text{Rn} & \text{Rd} \\
\text{U} & 0 1
\end{array}
\]

Encoding

FMAXP \text{<Vd>.<T>, <Vn>.<T>, <Vm>.<T>}

Decode for this encoding

\[
\text{integer d = UInt(Rd);} \\
\text{integer n = UInt(Rn);} \\
\text{integer m = UInt(Rm);} \\
\text{if sz:Q == '10' then UNDEFINED;} \\
\text{integer esize = 32 << UInt(sz);} \\
\text{integer datasize = if Q == '1' then 128 else 64;} \\
\text{integer elements = datasize DIV esize;}
\]
boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
   4H when Q = 0
   8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
   2S when sz = 0, Q = 0
   4S when sz = 0, Q = 1
   2D when sz = 1, Q = 1
   The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);
V[d, datasize] = result;
C7.2.110   FMAXV

Floating-point Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Half-precision**

(FEAT_FP16)

\[
\begin{array}{cccccccccc|cccc}
0 & Q & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & Rn & Rd
\end{array}
\]

**Encoding**

FMAXV <V><d>, <Vn>.<T>

**Decode for this encoding**

\[
\text{if } \neg\text{HaveFP16Ext}() \text{ then UNDEFINED;}
\]

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } esize &= 16; \\
\text{integer } datasize &= \text{if } Q == '1' \text{ then 128 else 64;}
\end{align*}
\]

**Single-precision and double-precision**

\[
\begin{array}{cccccccccc|cccc}
0 & Q & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & Rn & Rd
\end{array}
\]

**Encoding**

FMAXV <V><d>, <Vn>.<T>

**Decode for this encoding**

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{if } sz:Q \neq '01' \text{ then UNDEFINED;}
\end{align*}
\]

\[
\begin{align*}
\text{integer } esize &= 32 << \text{UInt}(sz); \\
\text{integer } datasize &= \text{if } Q == '1' \text{ then 128 else 64;}
\end{align*}
\]

**Assembler symbols**

\(<V>\quad \text{For the half-precision variant: is the destination width specifier, } H.\)
For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

The encoding sz = 1 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0
8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

4S when Q = 1, sz = 0

The following encodings are reserved:
• Q = 0, sz = x.
• Q = 1, sz = 1.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FMAX, operand, esize);
```
C7.2.111   FMIN (vector)

Floating-point minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(FEAT_FP16)

```
0 Q 0 1 1 1 0 1 1 0 | Rm 0 0 1 1 0 1 | Rn Rd
```

**Encoding**

FMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Single-precision and double-precision

```
0 Q 0 1 1 1 0 1 | 1 sz 1 | Rm 1 1 1 0 1 | Rn Rd
```

**Encoding**

FMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) elem1;
bits(esize) elem2;
for e = 0 to elements-1
  if pair then
    elem1 = Elem[concat, 2*e, esize];
    elem2 = Elem[concat, (2*e)+1, esize];
  else
    elem1 = Elem[operand1, e, esize];
    elem2 = Elem[operand2, e, esize];
  if minimum then
    Elem[result, e, esize] = FPMin(elem1, elem2, FPCR[]);
  else
    Elem[result, e, esize] = FPMax(elem1, elem2, FPCR[]);
V[d, datasize] = result;
C7.2.112   FMIN (scalar)

Floating-point Minimum (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Half-precision variant**
Applies when ftype == 11.
FMIN <Hd>, <Hn>, <Hm>

**Single-precision variant**
Applies when ftype == 00.
FMIN <Sd>, <Sn>, <Sm>

**Double-precision variant**
Applies when ftype == 01.
FMIN <Dd>, <Dn>, <Dm>

**Decode for all variants of this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
    case ftype of
        when '00' esize = 32;
        when '01' esize = 64;
        when '10' UNDEFINED;
        when '11'
            if HaveFP16Ext() then
                esize = 16;
            else
                UNDEFINED;
```

**Assembler symbols**

<Db>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Db>  Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Db>  Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Db>  Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<s> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

Elem[result, 0, esize] = FPMin(operand1, operand2, fpcr);
V[d, 128] = result;
```
C7.2.113   FMINNM (vector)

Floating-point Minimum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the smaller of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(_FEAT_FP16)

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');

Single-precision and double-precision

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>sz</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

**Assembler symbols**

<\text{Vd}> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<\text{T}> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
- \text{4H} when Q = 0
- \text{8H} when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
- \text{2S} when sz = 0, Q = 0
- \text{4S} when sz = 0, Q = 1
- \text{2D} when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<\text{Vn}> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<\text{Vm}> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

\text{CheckFPAdvSIMDEnabled64();}

\text{bits(datasize) operand1 = V[n, datasize];}
\text{bits(datasize) operand2 = V[m, datasize];}
\text{bits(datasize) result;}
\text{bits(2*datasize) concat = operand2:operand1;}
\text{bits(esize) element1;}
\text{bits(esize) element2;}

\text{for e = 0 to elements-1}
\text{if pair then}
\text{element1 = Elem[concat, 2*e, esize];}
\text{element2 = Elem[concat, (2*e)+1, esize];}
\text{else}
\text{element1 = Elem[operand1, e, esize];}
\text{element2 = Elem[operand2, e, esize];}

\text{if minimum then}
\text{Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);}
\text{else}
\text{Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);}

\text{V[d, datasize] = result;}
C7.2.114   FMINNM (scalar)

Floating-point Minimum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant
Applies when ftype == 11.
FMINNM <Hd>, <Hn>, <Hm>

Single-precision variant
Applies when ftype == 00.
FMINNM <Sd>, <Sn>, <Sm>

Double-precision variant
Applies when ftype == 01.
FMINNM <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler symbols

<Db>   Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dr>   Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dr>   Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hi> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```assembly
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

Elem[result, 0, esize] = FPMinNum(operand1, operand2, fpcr);
V[d, 128] = result;
```
C7.2.115   FMINNMP (scalar)

Floating-point Minimum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(FEAT_FP16)

Decoding for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;
```

Single-precision and double-precision

Decoding for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
```
Assembler symbols

\(<V>\)  
For the half-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

- H  when \( sz = 0 \)
- The encoding \( sz = 1 \) is reserved.

For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

- S  when \( sz = 0 \)
- D  when \( sz = 1 \)

\(<d>\)  
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Vn>\)  
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<T>\)  
For the half-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

- 2H  when \( sz = 0 \)
- The encoding \( sz = 1 \) is reserved.

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:

- 2S  when \( sz = 0 \)
- 2D  when \( sz = 1 \)

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_PMINNUM, operand, esize, FALSE);
```
C7.2.116   **FMINNMP (vector)**

Floating-point Minimum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of floating-point values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Half-precision**
(_FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Encoding**

FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');

**Single-precision and double-precision**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Encoding**

FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

**Assembler symbols**

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0
8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);

V[d, datasize] = result;
C7.2.117   FMINNMV

Floating-point Minimum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(FEAT_FP16)

```

Half-precision

(FEAT_FP16)

0 1 1 0 0 1 1 1 0 0 1 0 1 0 0 0 0 0 1 1 0 0 1 0 1 0

0 Q 0 1 1 0 0 1 0 1 0 0 0 0 0 1 1 0 0 1 0 1 0

Rn       Rd
```

Encoding

FMINNMV <V><d>, <Vn>.<T>

Decode for this encoding

```plaintext
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision

```

Single-precision and double-precision

0 1 1 0 0 1 1 1 0 0 1 0 1 0 0 0 0 0 0 1 1 0 0 1 0 1 0

0 Q 1 0 1 1 1 0 1 0 1 0 0 0 0 0 1 1 0 0 1 0 1 0

Rn       Rd
```

Encoding

FMINNMV <V><d>, <Vn>.<T>

Decode for this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;   // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```
Assembler symbols

\(<V>\) For the half-precision variant: is the destination width specifier, H.
For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:
\[ S \quad \text{when } sz = 0 \]
The encoding \( sz = 1 \) is reserved.

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<\text{Vn}>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
\[ 4H \quad \text{when } Q = 0 \]
\[ 8H \quad \text{when } Q = 1 \]
For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:
\[ 4S \quad \text{when } Q = 1, sz = 0 \]
The following encodings are reserved:
- \( Q = 0, sz = x \).
- \( Q = 1, sz = 1 \).

Operation for all encodings

\texttt{CheckFPAdvSIMEnabled64();}
\texttt{bits(datasize) operand = V[n, datasize];}
\texttt{V[d, esize] = Reduce(ReduceOp_PMINNUM, operand, esize, FALSE);}
C7.2.118   FMINP (scalar)

Floating-point Minimum of Pair of elements (scalar). This instruction compares two vector elements in the source
SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP
register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results
in either a flag being set in FPSR or a synchronous exception being generated. For more information, see
Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(_FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 0 1</td>
<td>sz 1 1 0 0 0</td>
<td>0 1 1 1 1</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encodings

FMINP <V><d>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;

Single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0 1</td>
<td>sz 1 1 0 0 0</td>
<td>0 1 1 1 1</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FMINP <V><d>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize + 2;
Assembler symbols

\begin{itemize}
\item \texttt{<V>} For the half-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:
  \begin{itemize}
  \item \texttt{H} when \( sz = 0 \)
  \item The encoding \( sz = 1 \) is reserved.
  \end{itemize}
  For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:
  \begin{itemize}
  \item \texttt{S} when \( sz = 0 \)
  \item \texttt{D} when \( sz = 1 \)
  \end{itemize}
\end{itemize}

\begin{itemize}
\item \texttt{<d>} Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
\item \texttt{<Vn>} Is the name of the SIMD&FP source register, encoded in the "Rn" field.
\item \texttt{<T>} For the half-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:
  \begin{itemize}
  \item \texttt{2H} when \( sz = 0 \)
  \item The encoding \( sz = 1 \) is reserved.
  \end{itemize}
  For the single-precision and double-precision variant: is the source arrangement specifier, encoded in the "sz" field. It can have the following values:
  \begin{itemize}
  \item \texttt{2S} when \( sz = 0 \)
  \item \texttt{2D} when \( sz = 1 \)
  \end{itemize}
\end{itemize}

Operation for all encodings

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FMIN, operand, esize);
\end{verbatim}
C7.2.119  **FMINP (vector)**

Floating-point Minimum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the smaller of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision

(_FEAT_FP16)

![Half-precision encoding](image)

**Encoding**

\[
\text{FMINP } <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
\]

**Decode for this encoding**

\[
\text{if !HaveFP16Ext() then UNDEFINED;}
\]

\[
\text{integer d = UInt(Rd);}
\]

\[
\text{integer n = UInt(Rn);}
\]

\[
\text{integer m = UInt(Rm);}
\]

\[
\text{integer esize = 16;}
\]

\[
\text{integer datasize = if Q == '1' then 128 else 64;}
\]

\[
\text{integer elements = datasize DIV esize;}
\]

\[
\text{boolean pair = (U == '1');}
\]

\[
\text{boolean minimum = (o1 == '1');}
\]

### Single-precision and double-precision

![Single-precision and double-precision encoding](image)

**Encoding**

\[
\text{FMINP } <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
\]

**Decode for this encoding**

\[
\text{integer d = UInt(Rd);}
\]

\[
\text{integer n = UInt(Rn);}
\]

\[
\text{integer m = UInt(Rm);}
\]

\[
\text{if sz:Q == '10' then UNDEFINED;}
\]

\[
\text{integer esize = 32 << UInt(sz);}
\]

\[
\text{integer datasize = if Q == '1' then 128 else 64;}
\]

\[
\text{integer elements = datasize DIV esize;}
\]
boolean pair = (U == '1');
boolean minimum = (o1 == '1');

**Assembler symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);
V[d, datasize] = result;
C7.2.120  FMINV

Floating-point Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Half-precision**

(FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 | 5 4 | 0 |
|---------------------------------------------------------------------|
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |

**Encoding**

FMINV <V><d>, <Vn>.<T>

**Decode for this encoding**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;

**Single-precision and double-precision**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 | 5 4 | 0 |
|---------------------------------------------------------------------|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |

**Encoding**

FMINV <V><d>, <Vn>.<T>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;

**Assembler symbols**

<V>  For the half-precision variant: is the destination width specifier, H.
For the single-precision and double-precision variant: is the destination width specifier, encoded in the "sz" field. It can have the following values:

\[ S \quad \text{when } sz = 0 \]

The encoding \( sz = 1 \) is reserved.

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

\[ 4H \quad \text{when } Q = 0 \]

\[ 8H \quad \text{when } Q = 1 \]

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

\[ 4S \quad \text{when } Q = 1, \; sz = 0 \]

The following encodings are reserved:

- \( Q = 0, \; sz = x \).
- \( Q = 1, \; sz = 1 \).

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

V[d, esize] = Reduce(ReduceOp_FMIN, operand, esize);
```
C7.2.121 FMLA (by element)

Floating-point fused Multiply-Add to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results in the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar, half-precision

(FEAT_FP16)

Encoding

FMLA <Hd>, <Hn>, <Vm>.H[<index>]

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');

Scalar, single-precision and double-precision

Encoding

FMLA <V><d>, <V><n>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

Vector, half-precision

(Encoding)

FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');

Vector, single-precision and double-precision

(Encoding)

FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (!o2 == '1');

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:
    S when sz = 0
    D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
    4H when Q = 0
    8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:
    2S when Q = 0, sz = 0
    4S when Q = 1, sz = 0
    2D when Q = 1, sz = 1

The encoding Q = 0, sz = 1 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in the "sz" field. It can have the following values:
    S when sz = 0
    D when sz = 1

For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in the "sz:L:H" field. It can have the following values:
    H:L when sz = 0, L = x
    H when sz = 1, L = 0

The encoding sz = 1, L = 1 is reserved.
Operation for all encodings

    CheckFPAdvSIMDEnabled64();
    bits(datasize) operand1 = V[n, datasize];
    bits(idxdsize) operand2 = V[m, idxdsize];
    bits(datasize) operand3 = V[d, datasize];
    bits(esize) element1;
    bits(esize) element2 = Elem[operand2, index, esize];
    FPCRType fpcr = FPCR[];
    boolean merge = elements == 1 && IsMerging(fpcr);
    bits(128) result = if merge then V[d, 128] else Zeros(128);

    for e = 0 to elements-1
        element1 = Elem[operand1, e, esize];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, fpcr);

    V[d, 128] = result;
C7.2.122  FMLA (vector)

Floating-point fused Multiply-Add to accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, adds the product to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(FEAT_FP16)

ENCODING

\[
\begin{array}{cccccccc|ccc|c}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & Rm & 0 & 0 & 0 & 1 & Rn & Rd \\
\end{array}
\]

Encoding

FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (a == '1');

Single-precision and double-precision

ENCODING

\[
\begin{array}{cccccccc|ccc|c}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & Rn & 1 & 1 & 0 & 1 & 1 & Rd \\
\end{array}
\]

Encoding

FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (op == '1');
Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
   4H when Q = 0
   8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
   2S when sz = 0, Q = 0
   4S when sz = 0, Q = 1
   2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
   element1 = Elem[operand1, e, esize];
   element2 = Elem[operand2, e, esize];
   if sub_op then element1 = FPNeg(element1);
   Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d, datasize] = result;
C7.2.123 FMLAL, FMLAL2 (by element)

Floating-point fused Multiply-Add Long to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note
ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

FMLAL

( FEAT_FHM )

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 0 1 1 1 1 1 0 L M Rn 0 0 0 H 0 Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Encoding
FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[index]

Decode for this encoding

```c
if !HaveFP10MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;
```

FMLAL2

( FEAT_FHM )

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 1 0 1 1 1 1 1 0 L M Rn 1 0 0 H 0 Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Encoding
FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[index]
Decode for this encoding

```plaintext
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 1;
```

Assembler symbols

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Ta>`: Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  - `2S` when Q = 0
  - `4S` when Q = 1
- `<Vn>`: Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Tb>`: Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  - `2H` when Q = 0
  - `4H` when Q = 1
- `<Vm>`: Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<index>`: Is the element index, encoded in the "H:L:M" fields.

Operation for all encodings

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part, datasize DIV 2];
bits(128) operand2 = V[m, 128];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d, datasize] = result;
```
C7.2.124 FMLAL, FMLAL2 (vector)

Floating-point fused Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding half-precision floating-point values in the vectors in the two source SIMD&FP registers, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

FMLAL

(FeAT_FHM)

\[
\begin{array}{ccccccccc}
| S | sz | 0 | Q | 0 | 1 | 1 | 0 | 0 | 1 | Rm & 1 & 1 & 0 & 1 & 1 & Rn & Rd \\
\hline
\end{array}
\]

Encoding

FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

Decode for this encoding

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 0;

FMLAL2

(FeAT_FHM)

\[
\begin{array}{ccccccccc}
| S | sz | 0 | Q | 1 | 0 | 1 | 1 | 0 | 0 | 1 | Rm & 1 & 1 & 0 & 1 & 1 & Rn & Rd \\
\hline
\end{array}
\]

Encoding

FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
Decode for this encoding

\[
\text{if } \neg \text{HaveFP16MulNoRoundingToFP32Ext}() \text{ then UNDEFINED;} \\
\text{integer } d = \text{UInt}(Rd); \\
\text{integer } n = \text{UInt}(Rn); \\
\text{integer } m = \text{UInt}(Rm); \\
\text{if } sz = '1' \text{ then UNDEFINED;} \\
\text{integer } \text{esize} = 32; \\
\text{integer } \text{datasize} = \text{if } Q = '1' \text{ then 128 else } 64; \\
\text{integer } \text{elements} = \text{datasize} \div \text{esize}; \\
\text{boolean } \text{sub_op} = (S = '1'); \\
\text{integer } \text{part} = 1;
\]

Assembler symbols

\[<Vd>\] Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\[<Ta>\] Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

2S \hspace{1cm} \text{when } Q = 0 \\
4S \hspace{1cm} \text{when } Q = 1

\[<Vn>\] Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\[<Tb>\] Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

2H \hspace{1cm} \text{when } Q = 0 \\
4H \hspace{1cm} \text{when } Q = 1

\[<Vm>\] Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

\[
\text{CheckFPAdvSIMDEnabled64}(); \\
\text{bits}(\text{datasize} \div 2) \text{ operand1} = V\text{part}[m, \text{part}, \text{datasize} \div 2]; \\
\text{bits}(\text{datasize} \div 2) \text{ operand2} = V\text{part}[m, \text{part}, \text{datasize} \div 2]; \\
\text{bits}(\text{datasize}) \text{ operand3} = V[d, \text{datasize}]; \\
\text{bits}(\text{datasize}) \text{ result}; \\
\text{bits}(\text{esize} \div 2) \text{ element1}; \\
\text{bits}(\text{esize} \div 2) \text{ element2};
\]

\[
\text{for } e = 0 \text{ to } \text{elements}-1 \\
\text{element1} = \text{Elem}[\text{operand1}, e, \text{esize} \div 2]; \\
\text{element2} = \text{Elem}[\text{operand2}, e, \text{esize} \div 2]; \\
\text{if } \text{sub_op} \text{ then } \text{element1} = \text{FPNeg}(\text{element1}); \\
\text{Elem}[\text{result}, e, \text{esize}] = \text{FPMulAddH}(\text{Elem}[\text{operand3}, e, \text{esize}], \text{element1}, \text{element2}, \text{FPCR}[]); \\
V[d, \text{datasize}] = \text{result};
\]
C7.2.125  **FMLS (by element)**

Floating-point fused Multiply-Subtract from accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar, half-precision**

(_FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 1 0 0 L M</td>
<td>Rm</td>
<td>0 1 0 1 H 0</td>
<td>Rn</td>
<td>Rd</td>
<td>o2</td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FMLS <Hd>, <In>, <Vm>.H[<index>]

**Decode for this encoding**

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');

**Scalar, single-precision and double-precision**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 1 1 1 sz L M</td>
<td>Rm</td>
<td>0 1 0 1 H 0</td>
<td>Rn</td>
<td>Rd</td>
<td>o2</td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FMLS <V><d>, <V><n>, <Vm>.<Ts>[<index>]

**Decode for this encoding**

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');

Vector, half-precision

(FEAT_FP16)

Encoding

FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');

Vector, single-precision and double-precision

Encoding

FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
when '0x' index = UInt(H:L);
when '10' index = UInt(H);
when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in the "sz" field. It can have the following values:
S when sz = 0
D when sz = 1
<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
4H when Q = 0
8H when Q = 1
For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:
2S when Q = 0, sz = 0
4S when Q = 1, sz = 0
2D when Q = 1, sz = 1
The encoding Q = 0, sz = 1 is reserved.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.
For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
<Ts> Is an element size specifier, encoded in the "sz" field. It can have the following values:
S when sz = 0
D when sz = 1
<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.
For the single-precision and double-precision variant: is the element index, encoded in the "sz:L:H" field. It can have the following values:
H:L when sz = 0, L = x
H when sz = 1, L = 0
The encoding sz = 1, L = 1 is reserved.
Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) operand3 = V[d, datasize];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, fpcr);

V[d, 128] = result;
C7.2.126   FMLS (vector)

Floating-point fused Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding
floating-point values in the vectors in the two source SIMD&FP registers, negates the product, adds the result to the
 corresponding vector element of the destination SIMD&FP register, and writes the result to the destination
SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception
results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see
Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

**Half-precision**

(FEAT_FP16)

<table>
<thead>
<tr>
<th>[31 30 29 28]27 26 25 2423 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Encoding**

FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (a == '1');

**Single-precision and double-precision**

<table>
<thead>
<tr>
<th>[31 30 29 28]27 26 25 2423 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Encoding**

FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (op == '1');

**Assembler symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d, datasize] = result;
```
C7.2.127   FMLSL, FMLSL2 (by element)

Floating-point fused Multiply-Subtract Long from accumulator (by element). This instruction multiplies the
negated vector elements in the first source SIMD&FP register by the specified value in the second source
SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP
register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception
results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see
Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations
to support it.

Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

FMLSL

(FEAT_FHM)

Encoding

FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]

Decode for this encoding

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
  integer d = UInt(Rd);
  integer n = UInt(Rn);
  integer m = UInt('0':Rm);    // Vm can only be in bottom 16 registers.
  if sz == '1' then UNDEFINED;
  integer index = UInt(H:L:M);

  integer esize = 32;
  integer datasize = if Q == '1' then 128 else 64;
  integer elements = datasize DIV esize;

  boolean sub_op = (S == '1');
  integer part = 0;

FMLSL2

(FEAT_FHM)

Encoding

FMLSL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]
Decode for this encoding

if ![HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;]
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M); 

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 1;

Assembler symbols

<\Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
2S when Q = 0
4S when Q = 1

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
2H when Q = 0
4H when Q = 1

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<index> Is the element index, encoded in the "H:L:M" fields.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part, datasize DIV 2];
bits(128) operand2 = V[m, 128];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
  element1 = Elem[operand1, e, esize DIV 2];
  if sub_op then element1 = FPNeg(element1);
  Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d, datasize] = result;
C7.2.128 FMLSL, FMLSL2 (vector)

Floating-point fused Multiply-Subtract Long from accumulator (vector). This instruction negates the values in the vector of one SIMD&FP register, multiplies these with the corresponding values in another vector, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.FH indicates whether this instruction is supported.

FMLSL

(FEAT_FHM)

\[\begin{array}{cccccccc|ccc}
\hline
0 & Q & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & & Rm & 1 & 1 & 0 & 1 & 1 & Rn & Rd & \hline
\end{array}\]

Encoding

FMLSL \(<Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>\)

Decode for this encoding

\[
\text{if } \neg \text{HaveFP10MulNoRoundingToFP32Ext()} \text{ then UNDEFINED;}
\text{integer } d = \text{UInt}(Rd);
\text{integer } n = \text{UInt}(Rn);
\text{integer } m = \text{UInt}(Rm);
\text{if } sz == '1' \text{ then UNDEFINED;}
\text{integer } esize = 32;
\text{integer } datasize = \text{if } Q == '1' \text{ then 128 else 64;}
\text{integer } elements = \text{datasize DIV esize;}
\text{boolean } sub_op = (S == '1');
\text{integer } part = 0;
\]

FMLSL2

(FEAT_FHM)

\[\begin{array}{cccccccc|ccc}
\hline
0 & Q & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & & Rm & 1 & 1 & 0 & 1 & 1 & Rn & Rd & \hline
\end{array}\]

Encoding

FMLSL2 \(<Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>\)
Decode for this encoding

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;

Assembler symbols

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta>  Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
       2S when Q = 0
       4S when Q = 1
<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb>  Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
       2H when Q = 0
       4H when Q = 1
<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[m, part, datasize DIV 2];
bits(datasize DIV 2) operand2 = Vpart[m, part, datasize DIV 2];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    element2 = Elem[operand2, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d, datasize] = result;
C7.2.129  FMOV (vector, immediate)

Floating-point move immediate (vector). This instruction copies an immediate floating-point constant into every element of the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(FEAT_FP16)

| [31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8|7 6 5 4] | 0 |
|-------------------|---|
| Q | a | b | c | d | e | f | g | h | Rd |

Encoding

FMOV <Vd>.<T>, #<imm>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;
integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
imm8 = a:b:c:d:e:f:g:h;
imm16 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 2):imm8<5:0>:Zeros(6);
imm = Replicate(imm16, datasize DIV 16);

Single-precision and double-precision

| [31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8|7 6 5 4] | 0 |
|-------------------|---|
| Q | op | a | b | c | d | e | f | g | h | Rd |

Single-precision variant

Applies when op == 0.
FMOV <Vd>.<T>, #<imm>

Double-precision variant

Applies when Q == 1 && op == 1.
FMOV <Vd>.2D, #<imm>

Decode for all variants of this encoding

integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
bites(datasize) imm;
bites(64) imm64;
if cmode:op == '11111' then
// FMOV Dn,#imm is in main FP instruction set
if Q == '0' then UNDEFINED;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0
8H when Q = 1

For the single-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

2S when Q = 0
4S when Q = 1

<imm> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "a:b:c:d:e:f:g:h". For details of the range of constants available and the encoding of <imm>, see Modified immediate constants in A64 floating-point instructions on page C2-244.

Operation for all encodings

CheckFPAdvSIMDEnabled64();

V[rd, datasize] = imm;
C7.2.130  FMOV (register)

Floating-point Move register without conversion. This instruction copies the floating-point value in the SIMD&FP source register to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant
Applies when ftype == 11.
FMOV <Hd>, <Hn>

Single-precision variant
Applies when ftype == 00.
FMOV <Sd>, <Sn>

Double-precision variant
Applies when ftype == 01.
FMOV <Dd>, <Dn>

Decode for all variants of this encoding

```c
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;

    case ftype of
        when '00' esize = 32;
        when '01' esize = 64;
        when '10' UNDEFINED;
        when '11'
            if HaveFP16Ext() then
                esize = 16;
            else
                UNDEFINED;
```

Assembler symbols

|<Dd> | Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
|<Dn> | Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field. |
|<Hd> | Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
|<Hn> | Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field. |
|<Sd> | Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
|<Sn> | Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field. |
Operation

CheckFPEnabled64();

bits(128) result = 0<127:0>;

bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = operand;
V[d, 128] = result;
C7.2.131  FMOV (general)

Floating-point Move to or from general-purpose register without conversion. This instruction transfers the contents of a SIMD&FP register to a general-purpose register, or the contents of a general-purpose register to a SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision to 32-bit variant

Applies when \(sf == 0 \&\& ftype == 11 \&\& rmode == 00 \&\& opcode == 110\).

FMOV \(<Wd>, <Hn>\)

Half-precision to 64-bit variant

Applies when \(sf == 1 \&\& ftype == 11 \&\& rmode == 00 \&\& opcode == 110\).

FMOV \(<Xd>, <Hn>\)

32-bit to half-precision variant

Applies when \(sf == 0 \&\& ftype == 11 \&\& rmode == 00 \&\& opcode == 111\).

FMOV \(<Hd>, <Wn>\)

32-bit to single-precision variant

Applies when \(sf == 0 \&\& ftype == 00 \&\& rmode == 00 \&\& opcode == 111\).

FMOV \(<Sd>, <Wn>\)

Single-precision to 32-bit variant

Applies when \(sf == 0 \&\& ftype == 00 \&\& rmode == 00 \&\& opcode == 110\).

FMOV \(<Wd>, <Sn>\)

64-bit to half-precision variant

Applies when \(sf == 1 \&\& ftype == 11 \&\& rmode == 00 \&\& opcode == 111\).

FMOV \(<Hd>, <Xn>\)

64-bit to double-precision variant

Applies when \(sf == 1 \&\& ftype == 01 \&\& rmode == 00 \&\& opcode == 111\).

FMOV \(<Dd>, <Xn>\)

64-bit to top half of 128-bit variant

Applies when \(sf == 1 \&\& ftype == 10 \&\& rmode == 01 \&\& opcode == 111\).

FMOV \(<Vd>.D[1], <Xn>\)
Double-precision to 64-bit variant

Applies when \( sf == 1 \) \&\& \( ftype == 01 \) \&\& \( rmode == 00 \) \&\& \( opcode == 110 \).

\[ \text{FMOV} \langle Xd \rangle, \langle Dn \rangle \]

Top half of 128-bit to 64-bit variant

Applies when \( sf == 1 \) \&\& \( ftype == 10 \) \&\& \( rmode == 01 \) \&\& \( opcode == 110 \).

\[ \text{FMOV} \langle Xd \rangle, \langle Vn\rangle.D[1] \]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer intsize} &= \text{if } sf == '1' \text{ then } 64 \text{ else } 32; \\
\text{integer fltsize} &= \text{FPCvtOp } op; \\
\text{FPRounding } &\text{ rounding}; \\
\text{boolean unsigned} &= \text{false}; \\
\text{integer part} &= \text{false}; \\
\end{align*}
\]

\[
\begin{align*}
\text{case } ftype \text{ of} \\
&\text{when } '00' \text{ then } \text{fltsize} = 32; \\
&\text{when } '01' \text{ then } \text{fltsize} = 64; \\
&\text{when } '10' \text{ then } \text{fltsize} = 128; \\
&\text{when } '11' \text{ then } \text{fltsize} = 16; \\
\end{align*}
\]

\[
\begin{align*}
\text{case } opcode<2:1>:rmode \text{ of} \\
&\text{when } '00 xx' \text{ then } \text{rounding} = \text{FPCvtOp } \text{op; } \text{unsigned} = \text{false}; \text{op} = \text{FPCvtOp } \text{op} \text{; } \\
&\text{when } '01 00' \text{ then } \text{rounding} = \text{FPCvtOp } \text{op; } \text{unsigned} = \text{false}; \text{op} = \text{FPCvtOp } \text{op} \text{; } \\
&\text{when } '10 00' \text{ then } \text{rounding} = \text{FPCvtOp } \text{op; } \text{unsigned} = \text{false}; \text{op} = \text{FPCvtOp } \text{op} \text{; } \\
&\text{when } '11 00' \text{ then } \text{rounding} = \text{FPCvtOp } \text{op; } \text{unsigned} = \text{false}; \text{op} = \text{FPCvtOp } \text{op} \text{; } \\
&\text{otherwise } \text{UNDEFINED;}
\end{align*}
\]
Assembler symbols

&Dd Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
&Hd Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
&Sd Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
&Wn Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
&Vd Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
&Dn Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Xn> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
&Sn Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

if op == FPConvOp_CVT_FtoI_JS then
  CheckFPAdvSIMDEnabled64();
else
  CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if op == FPConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;
case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n, fsize];
    intval = FPToFixed(fltval, 0, unsigned, fpcr, rounding, intsize);
    X[d, intsize] = intval;
  when FPConvOp_CVT_ItoF
    intval = X[n, intsize];
    fltval = if merge then V[d, fsize] else Zeros(fsize);
    Eml(fltval, 0, fltsize) = FixedToFP(intval, 0, unsigned, fpcr, rounding, fltsize);
    V[d, fsize] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n, part, fsize];
    intval = ZeroExtend(fltval, intsize);
    X[d, intsize] = intval;
  when FPConvOp_MOV_ItoF
    intval = X[n, intsize];
    fltval = intval<fltsize-1:0>;
    Vpart[d, part, fsize] = fltval;
  when FPConvOp_CVT_FtoI_JS
    bit z;
    fltval = V[n, fsize];
    (intval, z) = FPToFixedJS(fltval, fpcr, TRUE, intsize);
    PSTATE.<N,Z,C,V> = '0':z:'00';
    X[d, intsize] = intval;
Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C7.2.132   FMOV (scalar, immediate)

Floating-point move immediate (scalar). This instruction copies a floating-point immediate constant into the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant

Applies when \( ftype == 11 \).

\[
\text{FMOV } <\text{Hd}>, \#<\text{imm}>
\]

Single-precision variant

Applies when \( ftype == 00 \).

\[
\text{FMOV } <\text{Sd}>, \#<\text{imm}>
\]

Double-precision variant

Applies when \( ftype == 01 \).

\[
\text{FMOV } <\text{Dd}>, \#<\text{imm}>
\]

Decode for all variants of this encoding

```plaintext
integer d = UInt(Rd);
integer datasize;

case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
datasize = 16;
    else
     UNDEFINED;
    end

bits(datasize) imm = VFPExpandImm(imm8, datasize);
```

Assembler symbols

\(<\text{Dd}>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<\text{Hd}>\) Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<\text{Sd}>\) Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<\text{imm}>\) Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in the "imm8" field. For details of the range of constants available and the encoding of \(<\text{imm}>\), see Modified immediate constants in A64 floating-point instructions on page C2-244.
Operation

CheckFPEnabled64();

V[d, datasize] = imm;
C7.2.133  FMSUB

Floating-point Fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, adds that to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant
Applies when ftype == 11.
FMSUB <Hd>, <Hn>, <Hm>, <Ha>

Single-precision variant
Applies when ftype == 00.
FMSUB <Sd>, <Sn>, <Sm>, <Sa>

Double-precision variant
Applies when ftype == 01.
FMSUB <Dd>, <Dn>, <Dm>, <Da>

Decode for all variants of this encoding

integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Da> Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Ha> Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Sa> Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

**Operation**

```c
CheckFPEnabled64();

bits(esize) operand = V[a, esize];
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a, 128] else Zeros(128);

operand1 = FPNeg(operand1);
Elem[result, 0, esize] = FPMulAdd(operand, operand1, operand2, fpcr);
V[d, 128] = result;
```
C7.2.134  FMUL (by element)

Floating-point Multiply (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar, half-precision

(FEAT_FP16)

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 14 13 12 | 11 10 9 | 5 4 | 0 |
|-------------|-------------|-------------|----|---|-------------|     |    |    |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | L | M | Rm | 1 | 0 | 0 | 1 | H | 0 | Rd |
| U |

Encoding

FMUL <Hd>, <Hn>, <Vm>.H<[index]>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');

Scalar, single-precision and double-precision

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 14 13 12 | 11 10 9 | 5 4 | 0 |
|-------------|-------------|-------------|----|---|-------------|     |    |    |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | sz | L | M | Rm | 1 | 0 | 0 | 1 | H | 0 | Rd |
| U |

Encoding

FMUL <V><d>, <V><n>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');

Vector, half-precision

(FEAT_FP16)

[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12|11 10 9 | 5 4 | 0 ]

U

Encoding

FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');

Vector, single-precision and double-precision

[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12|11 10 9 | 5 4 | 0 ]

U

Encoding

FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:
    S when sz = 0
    D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
    4H when Q = 0
    8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:
    2S when Q = 0, sz = 0
    4S when Q = 1, sz = 0
    2D when Q = 1, sz = 1

The encoding Q = 0, sz = 1 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.
For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in the "sz" field. It can have the following values:
    S when sz = 0
    D when sz = 1

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.
For the single-precision and double-precision variant: is the element index, encoded in the "sz:L:H" field. It can have the following values:
    H:L when sz = 0, L = x
    H when sz = 1, L = 0

The encoding sz = 1, L = 1 is reserved.
Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, fpcr);
    else
        Elem[result, e, esize] = FPMul(element1, element2, fpcr);
V[d, 128] = result;
C7.2.135   FMUL (vector)

Floating-point Multiply (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision
(FEAT_FP16)

```
| 31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 1 | 1 | 1 | Rn | Rd |
```

**Encoding**

FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Single-precision and double-precision

```
| 31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
| 0 | Q | 1 | 0 | 1 | 1 | 0 | 0 | sz | 1 | Rm | 1 | 1 | 0 | 1 | 1 | Rn | Rd |
```

**Encoding**

FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
Assembler symbols

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>  For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H  when Q = 0
8H  when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S  when sz = 0, Q = 0
4S  when sz = 0, Q = 1
2D  when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);

V[d, datasize] = result;
C7.2.136  **FMUL (scalar)**

Floating-point Multiply (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Half-precision variant**

Applies when ftype == 11.

FMUL <Hd>, <Hn>, <Hm>

**Single-precision variant**

Applies when ftype == 00.

FMUL <Sd>, <Sn>, <Sm>

**Double-precision variant**

Applies when ftype == 01.

FMUL <Dd>, <Dn>, <Dm>

**Decode for all variants of this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
end case;
```

**Assembler symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
basis(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
basis(128) result = if merge then V[n, 128] else Zeros(128);

bits(esize) product = FPMul(operand1, operand2, fpcr);
Elem[result, 0, esize] = product;
V[d, 128] = result;
```
C7.2.137   **FMULX (by element)**

Floating-point Multiply extended (by element). This instruction multiplies the floating-point values in the vector elements in the first source SIMD&FP register by the specified floating-point value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Scalar, half-precision

(FAIL_FP16)

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20 19]</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0 0 L M Rm 1 0 0 1 H 0 Rn Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FMULX <Hd>, <Hn>, <Vm>.H[<index>]

**Decode for this encoding**

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');

### Scalar, single-precision and double-precision

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20 19]</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 1 1 1 1 sz L M Rm 1 0 0 1 H 0 Rn Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FMULX <V<d>, <V<n>, <Vm>.<Ts>[<index>]

**Decode for this encoding**

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
   when '0x' index = UInt(H:L);
when '10' index = UInt(H);
when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');

Vector, half-precision

(FEAT_FP16)

Encoding

FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');

Vector, single-precision and double-precision

Encoding

FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');

**Assembler symbols**

 Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

 Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

 Is a width specifier, encoded in the "sz" field. It can have the following values:

 S when sz = 0
 D when sz = 1

 Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

 Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

 For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

 4H when Q = 0
 8H when Q = 1

 For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "Q:sz" field. It can have the following values:

 2S when Q = 0, sz = 0
 4S when Q = 1, sz = 0
 2D when Q = 1, sz = 1

 The encoding Q = 0, sz = 1 is reserved.

 Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

 For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

 For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

 Is an element size specifier, encoded in the "sz" field. It can have the following values:

 S when sz = 0
 D when sz = 1

 For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

 For the single-precision and double-precision variant: is the element index, encoded in the "sz:L:H" field. It can have the following values:

 H:L when sz = 0, L = x
 H when sz = 1, L = 0

 The encoding sz = 1, L = 1 is reserved.
Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);
for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  if mulx_op then
    Elem[result, e, esize] = FPMulX(element1, element2, fpcr);
  else
    Elem[result, e, esize] = FPMul(element1, element2, fpcr);
V[d, 128] = result;
C7.2.138   FMULX

Floating-point Multiply extended. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(FEAT_FP16)

| 31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|-----------|-----------|-----------|---|-----------|---|---|
| 0 1 0 1 1 1 1 0 0 1 0 | Rm 0 0 1 1 1 | Rn | Rd |

Encoding

FMULX <Hd>, <Hn>, <Hm>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;

Scalar single-precision and double-precision

| 31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|-----------|-----------|-----------|---|-----------|---|---|
| 0 1 0 1 1 1 1 0 0 sz 1 | Rm 1 1 0 1 1 1 | Rn | Rd |

Encoding

FMULX <S>d>, <S>n>, <S>m>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
Vector half precision

(_FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 0 0 1 0</td>
<td>Rm 0 0 1 1 1</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Encoding

FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 0 0 sz 1</td>
<td>Rm 1 1 0 1 1 1</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Encoding

FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0
D when sz = 1

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when \( Q = 0 \)

8H when \( Q = 1 \)

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when \( sz = 0, Q = 0 \)

4S when \( sz = 0, Q = 1 \)

2D when \( sz = 1, Q = 1 \)

The encoding \( sz = 1, Q = 0 \) is reserved.

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

```c
if elements == 1 then
    CheckFPEnabled64();
else
    CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMulX(element1, element2, fpcr);
V[d, 128] = result;
```
C7.2.139  FNEG (vector)

Floating-point Negate (vector). This instruction negates the value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Half-precision**

(FEAT_FP16)

$$[31 \ 30 \ 29 \ 28] [27 \ 26 \ 25 \ 24] [23 \ 22 \ 21 \ 20] [19 \ 18 \ 17 \ 16] [15 \ 14 \ 13 \ 12] [11 \ 10 \ 9 \ 8] [5 \ 4] \ 0 \ U$$

| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| Rn | Rd |

**Encoding**

FNEG <Vd>.<T>, <Vn>.<T>

**Decode for this encoding**

```c
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

**Single-precision and double-precision**

$$[31 \ 30 \ 29 \ 28] [27 \ 26 \ 25 \ 24] [23 \ 22 \ 21 \ 20] [19 \ 18 \ 17 \ 16] [15 \ 14 \ 13 \ 12] [11 \ 10 \ 9 \ 8] [5 \ 4] \ 0 \ U$$

| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| Rn | Rd |

**Encoding**

FNEG <Vd>.<T>, <Vn>.<T>

**Decode for this encoding**

```c
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

**Assembler symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;
V[d, datasize] = result;
```
C7.2.140   FNEG (scalar)

Floating-point Negate (scalar). This instruction negates the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant
Applies when $ftype == 11$.
FNEG $<Hd>$, $<Hn>$

Single-precision variant
Applies when $ftype == 00$.
FNEG $<Sd>$, $<Sn>$

Double-precision variant
Applies when $ftype == 01$.
FNEG $<Dd>$, $<Dn>$

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
  case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
      if HaveFP16Ext() then
        esize = 16;
      else
        UNDEFINED;
  end;
```

Assembler symbols

- $<Dd>$ Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- $<Dn>$ Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- $<Hd>$ Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- $<Hn>$ Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- $<Sd>$ Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- $<Sn>$ Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else 0<127:0>;

bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPNeg(operand);
V[d, 128] = result;
C7.2.141  FNMADD

Floating-point Negated fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1</td>
<td>1 1 1 1</td>
<td>type 1</td>
<td>Rm</td>
<td>0</td>
<td>Ra</td>
<td>Rn</td>
<td>Rd</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Half-precision variant**
Applies when \( ftype = 11 \).

\[
\text{FNMADD } <\text{Hd}>, <\text{Hn}>, <\text{Hm}>, <\text{Ha}>
\]

**Single-precision variant**
Applies when \( ftype = 00 \).

\[
\text{FNMADD } <\text{Sd}>, <\text{Sn}>, <\text{Sm}>, <\text{Sa}>
\]

**Double-precision variant**
Applies when \( ftype = 01 \).

\[
\text{FNMADD } <\text{Dd}>, <\text{Dn}>, <\text{Dm}>, <\text{Da}>
\]

**Decode for all variants of this encoding**

\[
\text{integer } d = \text{UInt(Rd)}; \\
\text{integer } a = \text{UInt(Ra)}; \\
\text{integer } n = \text{UInt(Rn)}; \\
\text{integer } m = \text{UInt(Rm)}; \\
\text{integer } esize; \\
\text{case } ftype \text{ of} \\
\text{when '00' esize = 32; } \\
\text{when '01' esize = 64; } \\
\text{when '10' UNDEFINED; } \\
\text{when '11' } \\
\text{if HaveFP16Ext() then} \\
\text{esize = 16; } \\
\text{else} \\
\text{UNDEFINED;}
\]

**Assembler symbols**

\(<\text{Dd}>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<\text{Dn}>\) Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Da> Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Ha> Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Sa> Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

**Operation**

```c
CheckFPEnabled64();

bits(esize) operanda = V[a, esize];
bis(esize) operand1 = V[n, esize];
bis(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bis(128) result = if merge then V[a, 128] else Zeros(128);

operanda = FPNeg(operanda);
operand1 = FPNeg(operand1);
Elem[result, 0, esize] = FPMulAdd(operanda, operand1, operand2, fpcr);

V[d, 128] = result;
```
C7.2.142  FNMSUB

Floating-point Negated fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Instruction Format](image)

**Half-precision variant**

Applies when ftype == 11.

\[
\text{FNMSUB} \ <Hd>, <Hn>, <Hm>, <Ha>
\]

**Single-precision variant**

Applies when ftype == 00.

\[
\text{FNMSUB} \ <Sd>, <Sn>, <Sm>, <Sa>
\]

**Double-precision variant**

Applies when ftype == 01.

\[
\text{FNMSUB} \ <Dd>, <Dn>, <Dm>, <Da>
\]

**Decode for all variants of this encoding**

\[
\begin{align*}
\text{integer } & \ d = \ \text{UInt}(Rd); \\
\text{integer } & \ a = \ \text{UInt}(Ra); \\
\text{integer } & \ n = \ \text{UInt}(Rn); \\
\text{integer } & \ m = \ \text{UInt}(Rm); \\
\text{integer } & \ esize; \\
\text{case } & \ ftype \ of \\
\text{when } & \ '00' \ esize = 32; \\
\text{when } & \ '01' \ esize = 64; \\
\text{when } & \ '10' \ \text{UNDEFINED}; \\
\text{when } & \ '11' \\
\text{if } & \ \text{HaveFP16Ext}() \ \text{then} \\
& \ \ esize = 16; \\
\text{else } & \ \text{UNDEFINED};
\end{align*}
\]

**Assembler symbols**

\[
\begin{align*}
<Dd> & \quad \text{Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.} \\
<Dn> & \quad \text{Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.}
\end{align*}
\]
<Dm> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Da> Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Ha> Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Sa> Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

**Operation**

```c
CheckFPEnabled64();

bits(esize) operanda = V[a, esize];
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a, 128] else Zeros(128);

operanda = FPNeg(operanda);
Elem[result, 0, esize] = FPMulAdd(operanda, operand1, operand2, fpcr);

V[d, 128] = result;
```
C7.2.143  FNMUL (scalar)

Floating-point Multiply-Negate (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the negation of the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant
Applies when ftype == 11.
FNMUL <Hd>, <Hn>, <Hm>

Single-precision variant
Applies when ftype == 00.
FNMUL <Sd>, <Sn>, <Sm>

Double-precision variant
Applies when ftype == 01.
FNMUL <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
when '00' esize = 32;
when '01' esize = 64;
when '10' UNDEFINED;
when '11'
    if HaveFP16Ext() then
        esize = 16;
    else
        UNDEFINED;

Assembler symbols

<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>  Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>  Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>  Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

bits(esize) product = FPMul(operand1, operand2, fpcr);
product = FPNeg(product);
Elem[result, 0, esize] = product;

V[d, 128] = result;
```
C7.2.144 FRECPE

Floating-point Reciprocal Estimate. This instruction finds an approximate reciprocal estimate for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 1 0 1 1 1 1 0 0 1 1 1 0 1 1 0</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Encoding

FRECPE <Hd>, <Hn>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 1 0 1 1 1 1 0 0 1 1 1 0 1 1 0</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Encoding

FRECPE <V><d>, <V><n>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

Vector half precision

(Feat_FP16)
Encoding

FRECPE <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Vector single-precision and double-precision

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in the "sz" field. It can have the following values:
  S when sz = 0
  D when sz = 1
<q> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<q> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  4H when Q = 0
8H when Q = 1
For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1
The encoding sz = 1, Q = 0 is reserved.

<Vn> is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

```c
if elements == 1 then
    CheckFPEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bias(128) result = if merge then V[d, 128] else Zeros(128);
bias(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecipEstimate(element, FPCR[]);
V[d, 128] = result;
```

```c```
C7.2.145  FRECPS

Floating-point Reciprocal Step. This instruction multiplies the corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 2.0, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(FEAT_FP16)

If !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;

Scalar single-precision and double-precision

(FEAT_FP16)

Vector half precision

(FEAT_FP16)
C7.2 Alphabetical list of A64 Advanced SIMD and floating-point instructions

Encoding
FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding
if !HaveFP16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Vector single-precision and double-precision

Encoding
FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:
    S when sz = 0
    D when sz = 1

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- `4H` when \( Q = 0 \)
- `8H` when \( Q = 1 \)

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- `2S` when \( sz = 0, Q = 0 \)
- `4S` when \( sz = 0, Q = 1 \)
- `2D` when \( sz = 1, Q = 1 \)

The encoding \( sz = 1, Q = 0 \) is reserved.

\(<\text{T}>\)

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<\text{Vm}>\)

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

```plaintext
if elements == 1 then
    CheckFPEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRecipStepFused(element1, element2);
V[d, 128] = result;
```

```
C7.2.146   FRECPX

Floating-point Reciprocal exponent (scalar). This instruction finds an approximate reciprocal exponent for the
source SIMD&FP register and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results
in either a flag being set in FPSR or a synchronous exception being generated. For more information, see
Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

**Half-precision**

(FEAT_FP16)

```
| 31 30 29 28| 27 26 25 24| 23 22 21 20| 19 18 17 16| 15 14 13 12| 11 10 9  |  5 4  | 0 |
| 0 1 0 | 1 1 1 1 | 0 | 1 1 1 1 | 0 0 1 1 1 1 | 1 0 | Rn | Rd |
```

**Encoding**

FRECPX <Hd>, <Hn>

**Decode for this encoding**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;

**Single-precision and double-precision**

```
| 31 30 29 28| 27 26 25 24| 23 22 21 20| 19 18 17 16| 15 14 13 12| 11 10 9  |  5 4  | 0 |
| 0 1 0 | 1 1 1 1 | 0 | 1 0 0 | 0 1 1 1 1 1 | 1 0 | Rn | Rd |
```

**Encoding**

FRECPX <V><d>, <V><n>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 32 << UInt(sz);

**Assembler symbols**

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

- 5 when sz = 0
D when $sz = 1$

$d$ is the number of the SIMD&FP destination register, encoded in the "Rd" field.

$n$ is the number of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

```c
CheckFPEnabled64();
bits(esize) operand = V[n, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

Elem[result, 0, esize] = FPRecpX(operand, fpcr);

V[d, 128] = result;
```
C7.2.147 FRINT32X (vector)

Floating-point Round to 32-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision

(FEAT_FRINTTS)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
</tr>
</tbody>
</table>

U  op

Encoding

FRINT32X <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2S</td>
<td>when sz = 0, Q = 0</td>
</tr>
<tr>
<td>4S</td>
<td>when sz = 0, Q = 1</td>
</tr>
<tr>
<td>2D</td>
<td>when sz = 1, Q = 1</td>
</tr>
</tbody>
</table>

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);

V[d, datasize] = result;
C7.2.148   FRINT32X (scalar)

Floating-point Round to 32-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Floating-point**

(FEAT_FRINTTS)

```
[ 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  0 ]
```

- **ftype**
- **op**

**Single-precision variant**

Applies when `ftype == 00`.

`FRINT32X <Sd>, <Sn>`

**Double-precision variant**

Applies when `ftype == 01`.

`FRINT32X <Dd>, <Dn>`

**Decode for all variants of this encoding**

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '1x' UNDEFINED;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

**Assembler symbols**

- `<Dd>`: Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>`: Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>`: Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>`: Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPEnabled64();

FPCRTyp fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem [result, 0, esize] = FPRoundIntN(operand, fpcr, rounding, 32);

V[d, 128] = result;
C7.2.149 FRINT32Z (vector)

Floating-point Round to 32-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision
(FeaT_FRINTTS)

Encoding
FRINT32Z <Vd>.<T>, <Vn>.<T>

Decode for this encoding

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);
```

Assembler symbols

**<Vd>**
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<T>**
Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

**<Vn>**
Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);

V[d, datasize] = result;
C7.2.150   FRINT32Z (scalar)

Floating-point Round to 32-bit Integer toward Zero (scalar). This instruction rounds a floating-point value in the
SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the Round
toward Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the
{corresponding} input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the
instruction returns {for the corresponding result value} the most negative integer representable in the destination
size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception
results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see
Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

**Floating-point**

(FEAT_FRINTTS)

```
|31| 30| 29| 28| 27| 26| 25| 24|23| 22| 21| 20|19| 18| 17| 16|15| 14| 13| 12|11| 10| 9 | 5 | 4 | 0 |
|0| 0| 0| 1| 1| 1| 1| 0| 0| x | 1| 0| 1| 0| 0| 0| 0| 1| 0| 0| 0| 0|Rn| Rd|
```

**Single-precision variant**

Applies when ftype == 00.

FRINT32Z <Sd>, <Sn>

**Double-precision variant**

Applies when ftype == 01.

FRINT32Z <Dd>, <Dn>

**Decode for all variants of this encoding**

```java
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '1x' UNDEFINED;
```

**Assembler symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPEnabled64();

FPCRTypedef pcr = FPCR[];
boolean merge = IsMerging(pcr);
bite(128) result = if merge then V[d, 128] else Zeros(128);
bite(esize) operand = V[p, esize];

elem[results, 0, esize] = FPRoundIntN(operand, pcr, FPRounding.ZERO, 32);

V[d, 128] = result;
C7.2.151 FRINT64X (vector)

Floating-point Round to 64-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision

(FEAT_FRINTTS)

Encode

FRINT64X <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
  2S when sz = 0, Q = 0
  4S when sz = 0, Q = 1
  2D when sz = 1, Q = 1
The encoding sz = 1, Q = 0 is reserved.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);

V[d, datasize] = result;
C7.2.152 FRINT64X (scalar)

Floating-point Round to 64-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FRINTTS)

Single-precision variant

Applies when \( ftype = 00 \).

\[
\text{FRINT64X} \ <Sd>, \ <Sn>
\]

Double-precision variant

Applies when \( ftype = 01 \).

\[
\text{FRINT64X} \ <Dd>, \ <Dn>
\]

Decode for all variants of this encoding

\[
\text{if} \ \text{!HaveFrintExt()} \ \text{then UNDEFINED;}
\]

\[
\text{integer} \ d = \text{UInt}(\text{Rd});
\]

\[
\text{integer} \ n = \text{UInt}(\text{Rn});
\]

\[
\text{integer} \ esize;
\]

\[
\text{case} \ ftype \ \text{of}
\]

\[
\text{when} \ '00' \ esize = 32;
\]

\[
\text{when} \ '01' \ esize = 64;
\]

\[
\text{when} \ '1x' \ \text{UNDEFINED;}
\]

\[
\text{FPRounding} \ \text{rounding} = \text{FPRoundingMode}(\text{FPCR});
\]

Assembler symbols

\[
\text{<Dd>} \quad \text{Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.}
\]

\[
\text{<Dn>} \quad \text{Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.}
\]

\[
\text{<Sd>} \quad \text{Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.}
\]

\[
\text{<Sn>} \quad \text{Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.}
\]
Operation

CheckFPEnabled64();

FPCRTypedefpcr = FPCR[];
boolean merge = IsMerging(fpcr);
b128 result = int merge then V[d, 128] else Zeros(128);
b1esize) operand = V[n, esize];

Elem(result, 0, esize) = FPRoundIntN(operand, fpcr, rounding, 64);
V[d, 128] = result;
C7.2.153  FRINT64Z (vector)

Floating-point Round to 64-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision

(_FEAT_FRINTTS)

Encoding
FRINT64Z <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1
The encoding sz = 1, Q = Θ is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);

V[d, datasize] = result;
C7.2.154   FRINT64Z (scalar)

Floating-point Round to 64-bit Integer toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the corresponding input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point

(FEAT_FRINTTS)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 1 0 0 0 1 1 0 1 0 0 0 0 Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ftype  op

Single-precision variant

Applies when ftype == 00.
FRINT64Z <Sd>, <Sn>

Double-precision variant

Applies when ftype == 01.
FRINT64Z <Dd>, <Dn>

Decode for all variants of this encoding

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
when '00' esize = 32;
when '01' esize = 64;
when '1x' UNDEFINED;

Assembler symbols

<Db> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Db> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFP();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, FPRounding.ZERO, 64);

V[d, 128] = result;
C7.2.155 FRINTA (vector)

Floating-point Round to Integral, to nearest with ties to Away (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Half-precision**

(FEAT_FP16)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

FRINTA <Vd>.<T>, <Vn>.<T>

**Decode for this encoding**

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;

```
```
case U:o1:o2 of
when '0xx' rounding = FPDecodeRounding(o1:o2);
when '100' rounding = FPRounding_TIEAWAY;
when '101' UNDEFINED;
when '110' rounding = FPRoundingMode(FPCR[0]); exact = TRUE;
when '111' rounding = FPRoundingMode(FPCR[0]);
```
```
```
```

**Single-precision and double-precision**

```
| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 | 5 4 | 0 |
|-------------|-------------|-------------|-------------|-------------|-------------|-----|
| 0           | Q           | 1           | 0           | 1           | 1           | 0   |
| U           | o2           | o1           | Rn           | Rd           |
```

**Encoding**

FRINTA <Vd>.<T>, <Vn>.<T>
Decode for this encoding

```c
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);
```

Assembler symbols

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  - 4H when Q = 0
  - 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

- `<Vn>`: Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d, datasize] = result;
```
C7.2.156 FRINTA (scalar)

Floating-point Round to Integral, to nearest with ties to Away (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant

Applies when ftype == 11.

FRINTA <Hd>, <Hn>

Single-precision variant

Applies when ftype == 00.

FRINTA <Sd>, <Sn>

Double-precision variant

Applies when ftype == 01.

FRINTA <Dd>, <Dn>

Decode for all variants of this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
   case ftype of
      when '00' esize = 32;
      when '01' esize = 64;
      when '10' UNDEFINED;
      when '11'
         if HaveFP16Ext() then
            esize = 16;
         else
            UNDEFINED;
   end
```

Assembler symbols

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```
CheckFPEnabled64();
FPCRType fp = FPCR[];
boolean merge = IsMerging(fp);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPATInt(operand, fp, FPRounding_TIEAWAY, FALSE);
V[d, 128] = result;
```
C7.2.157  FRINTI (vector)

Floating-point Round to Integral, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Half-precision**

(FEAT_FP16)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q  0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rd</td>
</tr>
</tbody>
</table>
```

**Encoding**

FRINTI <Vd>.<T>, <Vn>.<T>

**Decode for this encoding**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);

**Single-precision and double-precision**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U  0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>sz</td>
<td>Rn</td>
</tr>
<tr>
<td>o2 1 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rd</td>
</tr>
</tbody>
</table>
```

**Encoding**

FRINTI <Vd>.<T>, <Vn>.<T>
**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

**Assembler symbols**

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>  For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>4H</td>
<td>Q = 0</td>
</tr>
<tr>
<td>8H</td>
<td>Q = 1</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>2S</td>
<td>sz = 0, Q = 0</td>
</tr>
<tr>
<td>4S</td>
<td>sz = 0, Q = 1</td>
</tr>
<tr>
<td>2D</td>
<td>sz = 1, Q = 1</td>
</tr>
</tbody>
</table>

The encoding sz = 1, Q = 0 is reserved.

<Vn>  Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);
V[d, datasize] = result;
C7.2.158   FRINTI (scalar)

Floating-point Round to Integral, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant

Applies when ftype == 11.

FRINTI <Hd>, <Hn>

Single-precision variant

Applies when ftype == 00.

FRINTI <Sd>, <Sn>

Double-precision variant

Applies when ftype == 01.

FRINTI <Dd>, <Dn>

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
  FPRounding rounding;
  rounding = FPRoundingMode(FPCR[]);

Assembler symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```java
CheckFPEnable64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);
V[d, 128] = result;
```
C7.2.159  FRINTM (vector)

Floating-point Round to Integral, toward Minus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

FRINTM <Vd><T>, <Vn><T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

Single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

FRINTM <Vd><T>, <Vn><T>
**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

**Assembler symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0
8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d, datasize] = result;
C7.2.160 FRINTM (scalar)

Floating-point Round to Integral, toward Minus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant

Applies when \( ftype = 11 \).

\[ \text{FRINTM} <\text{Hd}>, <\text{Hn}> \]

Single-precision variant

Applies when \( ftype = 00 \).

\[ \text{FRINTM} <\text{Sd}>, <\text{Sn}> \]

Double-precision variant

Applies when \( ftype = 01 \).

\[ \text{FRINTM} <\text{Dd}>, <\text{Dn}> \]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } esize; \\
\text{case } ftype \text{ of} \\
\text{when } '00' \text{ esize } &= 32; \\
\text{when } '01' \text{ esize } &= 64; \\
\text{when } '10' \text{  UNDEFINED; } \\
\text{when } '11' \\
&\quad \text{if HaveFP16Ext() then} \\
&\quad \text{esize } = 16; \\
&\quad \text{else} \\
&\quad \text{UNDEFINED;} \\
\text{FPRounding } &= \text{rounding}; \\
\text{rounding } &= \text{FPDecodeRounding('10');}
\end{align*}
\]

Assembler symbols

\(<\text{Dd}>\quad \text{Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.}\)
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);

V[d, 128] = result;
FRINTN (vector)

Floating-point Round to Integral, to nearest with ties to even (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FRINTN <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean exact = FALSE;
FPRounding rounding;
case U:0:o1:02 of
when '0xx' rounding = FPDecodeRounding(o1:o2);
when '100' rounding = FPRounding_TIEAWAY;
when '101' UNDEFINED;
when '110' rounding = FPRoundingMode(FPCR[.]); exact = TRUE;
when '111' rounding = FPRoundingMode(FPCR[.]);

Single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FRINTN <Vd>.<T>, <Vn>.<T>
Decode for this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);
```

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4H</td>
<td>when Q = 0</td>
</tr>
<tr>
<td>8H</td>
<td>when Q = 1</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2S</td>
<td>when sz = 0, Q = 0</td>
</tr>
<tr>
<td>4S</td>
<td>when sz = 0, Q = 1</td>
</tr>
<tr>
<td>2D</td>
<td>when sz = 1, Q = 1</td>
</tr>
</tbody>
</table>

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);
V[d, datasize] = result;
```
C7.2.162  FRINTN (scalar)

Floating-point Round to Integral, to nearest with ties to even (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant

Applies when ftype == 11.

FRINTN <Hd>, <Hn>

Single-precision variant

Applies when ftype == 00.

FRINTN <Sd>, <Sn>

Double-precision variant

Applies when ftype == 01.

FRINTN <Dd>, <Dn>

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
when '00' esize = 32;
when '01' esize = 64;
when '10' UNDEFINED;
when '11'
if HaveFP16Ext() then
    esize = 16;
else
    UNDEFINED;
FPRounding rounding;
rounding = FPDecodeRounding('00');

Assembler symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```java
CheckFPEnabled64();

FPCRTyperefcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);
V[d, 128] = result;
```
C7.2.163 FRINTP (vector)

Floating-point Round to Integral, toward Plus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision

(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Encoding

FRINTP <Vd>.<T>, <Vn>.<T>

#### Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o2:o1 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '101' rounding = FPRoundingMode(FPCR[0]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[0]);

### Single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Encoding

FRINTP <Vd>.<T>, <Vn>.<T>
Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

Assembler symbols

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>   For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
    4H    when Q = 0
    8H    when Q = 1
For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
    2S    when sz = 0, Q = 0
    4S    when sz = 0, Q = 1
    2D    when sz = 1, Q = 1
The encoding sz = 1, Q = 0 is reserved.
<Vn>  Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);
V[d, datasize] = result;
C7.2.164  FRINTP (scalar)

Floating-point Round to Integral, toward Plus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Half-precision variant
Applies when ftype == 11.
FRINTP <Hd>, <Hn>
```

```
Single-precision variant
Applies when ftype == 00.
FRINTP <Sd>, <Sn>
```

```
Double-precision variant
Applies when ftype == 01.
FRINTP <Dd>, <Dn>
```

```
Decode for all variants of this encoding
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
FP_Rounding rounding;
rounding = FPDecodeRounding('01');
```

```
Assembler symbols
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
```
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bots(128) result = if merge then V[d, 128] else Zeros(128);
bots(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);
V[d, 128] = result;
```
C7.2.165  FRINTX (vector)

Floating-point Round to Integral exact, using current rounding mode (vector). This instruction rounds a vector of
floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the
rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

When a result value is not numerically equal to the corresponding input value, an Inexact exception is raised. A zero
input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN
is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception
results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

FRINTX <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;

case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);

Single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

FRINTX <Vd>.<T>, <Vn>.<T>
**Decode for this encoding**

```plaintext
decimal d = UInt(Rd);
decimal n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
decimal esize = 32 << UInt(sz);
decimal datasize = if Q == '1' then 128 else 64;
decimal elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]): exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]):
endcase
```

**Assembler symbols**

<v>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<t>
For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
- 4H  when Q = 0
- 8H  when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
- 2S  when sz = 0, Q = 0
- 4S  when sz = 0, Q = 1
- 2D  when sz = 1, Q = 1
The encoding sz = 1, Q = 0 is reserved.

<n>
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);
V[d, datasize] = result;
```
C7.2.166  FRINTX (scalar)

Floating-point Round to Integral exact, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

When the result value is not numerically equal to the input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant

Applies when ftype == 11.

FRINTX <Hd>, <Hn>

Single-precision variant

Applies when ftype == 00.

FRINTX <Sd>, <Sn>

Double-precision variant

Applies when ftype == 01.

FRINTX <Dd>, <Dn>

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
FPRounding rounding;
rounding = FPRoundingMode(FPCR[ ]);
Assembler symbols

<\texttt{Dd}> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<\texttt{Dn}> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\texttt{Hd}> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<\texttt{Hn}> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\texttt{Sd}> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<\texttt{Sn}> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPEnabled64();

FPCRT\texttt{ype} fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

\texttt{Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, TRUE)};

V[d, 128] = result;
C7.2.167  FRINTZ (vector)

Floating-point Round to Integral, toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision

(FEAT_FP16)

\[
\begin{array}{c|cccccc|c}
U & 0 & Q & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & Rn & Rd \\
\end{array}
\]

Encoding

FRINTZ <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

Single-precision and double-precision

\[
\begin{array}{c|cccccc|c}
U & 0 & Q & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & Rn & Rd \\
\end{array}
\]

Encoding

FRINTZ <Vd>.<T>, <Vn>.<T>
**Decode for this encoding**

```plaintext
type integer d = U32(Rd);
type integer n = U32(Rn);

if sz:Q == '10' then UNDEFINED;
type integer esize = 32 << U32(sz);
type integer datasize = if Q == '1' then 128 else 64;
type integer elements = datasize DIV esize;

boolean exact = FALSE;

FPRounding rounding;
case U32:O1:O2 of
when '0xx' rounding = FPDecodeRounding(o1:O2);
when '100' rounding = FPRounding_TIEAWAY;
when '101' UNDEFINED;
when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
when '111' rounding = FPRoundingMode(FPCR[]);
end;
```

**Assembler symbols**

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  - `4H` when Q = 0
  - `8H` when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- `2S` when sz = 0, Q = 0
- `4S` when sz = 0, Q = 1
- `2D` when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

- `<Vn>` is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);
V[d, datasize] = result;
```
C7.2.168  FRINTZ (scalar)

Floating-point Round to Integral, toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant
Applies when ftype == 11.
FRINTZ <Hd>, <Hn>

Single-precision variant
Applies when ftype == 00.
FRINTZ <Sd>, <Sn>

Double-precision variant
Applies when ftype == 01.
FRINTZ <Dd>, <Dn>

Decode for all variants of this encoding
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
  FPRounding rounding;
  rounding = FPDecodeRounding('11');

Assembler symbols
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<5d> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<5n> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```java
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);
V[d, 128] = result;
```
C7.2.169  FRSQRTE

Floating-point Reciprocal Square Root Estimate. This instruction calculates an approximate square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(Feat_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

FRSQRTE <Hd>, <Hn>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Encoding

FRSQRTE <V><d>, <V><n>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

Vector half precision

(Feat_FP16)
Encodings
FRSQRT.E <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Vector single-precision and double-precision

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0
D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0
8H when Q = 1
For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:
2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1
The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

```c
if elements == 1 then
    CheckFPEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRSqrtEstimate(element, fpcr);
V[d, 128] = result;
```
C7.2.170   FRSQRTS

Floating-point Reciprocal Square Root Step. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 3.0, divides these results by 2.0, places the results into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar half precision

(Feat_FP16)

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 0 1 1 1 1 0 | 0 1 1 0 Rm | 0 0 1 1 1 | 1 | Rn | Rd |
```

Encoding

FRSQRTS <Hd>, <Hn>, <Hm>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;

Scalar single-precision and double-precision

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 0 1 1 1 1 0 | 1 | sz 1 | Rm | 1 1 1 1 1 | 1 | Rn | Rd |
```

Encoding

FRSQRTS <V>d>, <V>n>, <V>m>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

Vector half precision

(Feat_FP16)
Encoding
FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding
if !HaveFP16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Vector single-precision and double-precision

Encoding
FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler symbols
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<V> Is a width specifier, encoded in the "sz" field. It can have the following values:
    S  when sz = 0
    D  when sz = 1
<#> Is the number of the SIMD&FP destination register, in the "Rd" field.
<#> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<#> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0
8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

```
if elements == 1 then
    CheckFPEnabled64();
else
    CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
FPCRTyp e fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPKSqrtStepFused(element1, element2);

V[d, 128] = result;
```
C7.2.171  FSQRT (vector)

Floating-point Square Root (vector). This instruction calculates the square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Half-precision**

(FEAT_FP16)

Encoding

FSQRT <Vd>.<T>, <Vn>.<T>

Decode for this encoding

\[
\text{if } !\text{HaveFP16Ext()} \text{ then UNDEFINED;}
\]
\[
\text{integer } d = \text{UInt(Rd)};
\]
\[
\text{integer } n = \text{UInt(Rn)};
\]
\[
\text{integer esize} = 16;
\]
\[
\text{integer datasize} = \text{if } Q = '1' \text{ then 128 else 64};
\]
\[
\text{integer elements} = \text{datasize DIV esize} ;
\]

Single-precision and double-precision

Encoding

FSQRT <Vd>.<T>, <Vn>.<T>

Decode for this encoding

\[
\text{integer } d = \text{UInt(Rd)};
\]
\[
\text{integer } n = \text{UInt(Rn)};
\]
\[
\text{if sz}:Q = '10' \text{ then UNDEFINED;}
\]
\[
\text{integer esize} = 32 \times \text{UInt(sz)};
\]
\[
\text{integer datasize} = \text{if } Q = '1' \text{ then 128 else 64};
\]
\[
\text{integer elements} = \text{datasize DIV esize} ;
\]

Assembler symbols

\(<Vd>\)  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 4H when Q = 0
- 8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- 2S when sz = 0, Q = 0
- 4S when sz = 0, Q = 1
- 2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPSqrt(element, FPCR[]);

V[d, datasize] = result;
```
C7.2.172  FSQRT (scalar)

Floating-point Square Root (scalar). This instruction calculates the square root of the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision variant
Applies when $ftype == 11$.

FSQRT <Hd>, <Hn>

Single-precision variant
Applies when $ftype == 00$.

FSQRT <Sd>, <Sn>

Double-precision variant
Applies when $ftype == 01$.

FSQRT <Dd>, <Dn>

Decode for all variants of this encoding

```c
integer d = UInt(Rd);
integer n = UInt(Rn);

d =UInt(Rd);
integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dr> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hr> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
b启迪(128) result = if merge then V[d, 128] else 0x127:0;

bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPSqrt(operand, fpcr);

V[d, 128] = result;
```
C7.2.173  FSUB (vector)

Floating-point Subtract (vector). This instruction subtracts the elements in the vector in the second source SIMD&FP register, from the corresponding elements in the vector in the first source SIMD&FP register, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Half-precision**

(FeaT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Encoding**

FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');

**Single-precision and double-precision**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Encoding**

FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
Assembler symbols

\(<V_d>\)
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\)
For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- \(4H\) when \(Q = 0\)
- \(8H\) when \(Q = 1\)

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

- \(2S\) when \(sz = 0, Q = 0\)
- \(4S\) when \(sz = 0, Q = 1\)
- \(2D\) when \(sz = 1, Q = 1\)

The encoding \(sz = 1, Q = 0\) is reserved.

\(<V_n>\)
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<V_m>\)
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```assembly
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
bits(esize) diff;
FPCRTy pfc = FPCR[];
bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, fpc);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d, datasize] = result;
```
C7.2.174 FSUB (scalar)

Floating-point Subtract (scalar). This instruction subtracts the floating-point value of the second source SIMD&FP register from the floating-point value of the first source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

FSUB <Hd>, <Hn>, <Hm>

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 1 1 0</td>
<td>ftype 1</td>
<td>Rm 0 0 1 1 1 0</td>
<td>Rn Rd</td>
</tr>
</tbody>
</table>

**Half-precision variant**
Applies when ftype == 11.
FSUB <Hd>, <Hn>, <Hm>

**Single-precision variant**
Applies when ftype == 00.
FSUB <Sd>, <Sn>, <Sm>

**Double-precision variant**
Applies when ftype == 01.
FSUB <Dd>, <Dn>, <Dm>

**Decode for all variants of this encoding**

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

**Assembler symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<h> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<h> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<s> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<s> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<s> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```cpp
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

Elem[result, 0, esize] = FPSub(operand1, operand2, fpcr);
V[d, 128] = result;
```
C7.2.175 INS (element)

Insert vector element from another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (element). The alias is always the preferred disassembly.

Encoding

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Decode for this encoding

```python
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer dst_index = UInt(imm5<4:size+1>);
integer src_index = UInt(imm4<3:size>);
integer index_size = if imm4<3> == '1' then 128 else 64;
// imm4<size-1:0> is IGNORED

integer esize = 8 << size;
```

Assembler symbols

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Ts>`: Is an element size specifier, encoded in the "imm5" field. It can have the following values:
  - `B` when imm5 = xxxx1
  - `H` when imm5 = xxx10
  - `S` when imm5 = xx100
  - `D` when imm5 = x1000
  - The encoding imm5 = x0000 is reserved.
- `<index1>`: Is the destination element index encoded in the "imm5" field. It can have the following values:
  - imm5<4:1> when imm5 = xxxx1
  - imm5<4:2> when imm5 = xxx10
  - imm5<4:3> when imm5 = xx100
  - imm5<4> when imm5 = x1000
  - The encoding imm5 = x0000 is reserved.
- `<Vn>`: Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Is the source element index encoded in the "imm5:imm4" field. It can have the following values:

- imm4<3:0> when imm5 = xxxx1
- imm4<3:1> when imm5 = xxx10
- imm4<3:2> when imm5 = xx100
- imm4<3> when imm5 = x1000

The encoding imm5 = x0000 is reserved.

Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n, idxdsize];
bits(128) result;
result = V[d, 128];
Elem[result, dst_index, esize] = Elem[operand, src_index, esize];
V[d, 128] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.176   INS (general)

Insert vector element from general-purpose register. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (from general). The alias is always the preferred disassembly.

Encoding

INS <Vd>.<Ts>[<index>], <R><n>

Decode for this encoding

```plaintext
data = UInt(Rd);
n = UInt(Rn);
size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;
index = UInt(imm5<4:size+1>);
esize = 8 << size;
```

Assembler symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Ts>` Is an element size specifier, encoded in the "imm5" field. It can have the following values:
  - B when imm5 = xxxx1
  - H when imm5 = xxx10
  - S when imm5 = xx100
  - D when imm5 = x1000
  The encoding imm5 = x0000 is reserved.
- `<index>` Is the element index encoded in the "imm5" field. It can have the following values:
  - imm5<4:1> when imm5 = xxxx1
  - imm5<4:2> when imm5 = xxx10
  - imm5<4:3> when imm5 = xx100
  - imm5<4> when imm5 = x1000
  The encoding imm5 = x0000 is reserved.
- `<R>` Is the width specifier for the general-purpose source register, encoded in the "imm5" field. It can have the following values:
  - W when imm5 = xxxx1
W when imm5 = xxx10
W when imm5 = xx100
X when imm5 = x1000

The encoding imm5 = x0000 is reserved.

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n, esize];
bits(128) result;

result = V[d, 128];
Elem[result, index, esize] = element;
V[d, 128] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.177   LD1 (multiple structures)

Load multiple single-element structures to one, two, three, or four registers. This instruction loads multiple single-element structures from memory and writes the result to one, two, three, or four SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**No offset**

![Instruction Format](image)

**One register variant**
Applies when opcode == 0111.

LD1 { <Vt>.<T> }, [<Xn|SP>]

**Two registers variant**
Applies when opcode == 1010.

LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

**Three registers variant**
Applies when opcode == 0110.

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

**Four registers variant**
Applies when opcode == 0010.

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]

**Decode for all variants of this encoding**

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

**Post-index**

![Instruction Format](image)

**One register, immediate offset variant**
Applies when Rm == 11111 && opcode == 0111.

LD1 { <Vt>.<T> }, [Xn], <imm>

**One register, register offset variant**
Applies when Rm != 11111 && opcode == 0111.
LD1 { <Vt>.<T> }, [<Xn|SP>], <Xm>

**Two registers, immediate offset variant**
Applies when Rm == 11111 && opcode == 1010.
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>

**Two registers, register offset variant**
Applies when Rm != 11111 && opcode == 1010.
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>

**Three registers, immediate offset variant**
Applies when Rm == 11111 && opcode == 0110.
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>

**Three registers, register offset variant**
Applies when Rm != 11111 && opcode == 0110.
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>

**Four registers, immediate offset variant**
Applies when Rm == 11111 && opcode == 0010.
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

**Four registers, register offset variant**
Applies when Rm != 11111 && opcode == 0010.
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

**Decode for all variants of this encoding**
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

**Assembler symbols**

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 1D when size = 11, Q = 0
- 2D when size = 11, Q = 1

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<\textit{vt3}> is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<\textit{vt4}> is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<\textit{Xn}\mid SP> is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<\textit{imm}> For the one register, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

\begin{align*}
\#8 & \quad \text{when } Q = 0 \\
\#16 & \quad \text{when } Q = 1
\end{align*}

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

\begin{align*}
\#16 & \quad \text{when } Q = 0 \\
\#32 & \quad \text{when } Q = 1
\end{align*}

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

\begin{align*}
\#24 & \quad \text{when } Q = 0 \\
\#48 & \quad \text{when } Q = 1
\end{align*}

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

\begin{align*}
\#32 & \quad \text{when } Q = 0 \\
\#64 & \quad \text{when } Q = 1
\end{align*}

<\textit{Xm}> is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared decode for all encodings**

\begin{verbatim}
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;
integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
\end{verbatim}

**Operation for all encodings**

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;
\end{verbatim}
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAPropertyDescriptor();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.178   LD1 (single structure)

Load one single-element structure to one lane of one register. This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD&FP register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

LD1 { <Vt>.B }[<index>], [<Xn|SP>]

8-bit variant

Applies when opcode == 000.

LD1 { <Vt>.B }[<index>], [Xn|SP>]

16-bit variant

Applies when opcode == 010 && size == x0.

LD1 { <Vt>.H }[<index>], [Xn|SP>]

32-bit variant

Applies when opcode == 100 && size == 00.

LD1 { <Vt>.S }[<index>], [Xn|SP>]

64-bit variant

Applies when opcode == 100 && S == 0 && size == 01.

LD1 { <Vt>.D }[<index>], [Xn|SP>]

Decode for all variants of this encoding

Integer t = UInt(Rt);
Integer n = UInt(Rn);
Integer m = integer UNKNOWN;
Boolean wback = FALSE;
Boolean tag_checked = wback || n != 31;

Post-index

LD1 { <Vt>.B }[<index>], [Xn|SP>, #1

8-bit, immediate offset variant

Applies when Rm == 11111 && opcode == 000.

LD1 { <Vt>.B }[<index>], [Xn|SP>, #1
8-bit, register offset variant
Applies when Rm != 11111 && opcode == 000.
LD1 [ <Vt>.B ]([<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 010 && size == x0.
LD1 [ <Vt>.H ]([<index>], [<Xn|SP>], #2

16-bit, register offset variant
Applies when Rm != 11111 && opcode == 010 && size == x0.
LD1 [ <Vt>.H ]([<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 100 && size == 00.
LD1 [ <Vt>.S ]([<index>], [<Xn|SP>], #4

32-bit, register offset variant
Applies when Rm != 11111 && opcode == 100 && size == 00.
LD1 [ <Vt>.S ]([<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 100 && S == 0 && size == 01.
LD1 [ <Vt>.D ]([<index>], [<Xn|SP>], #8

64-bit, register offset variant
Applies when Rm != 11111 && opcode == 100 && S == 0 && size == 01.
LD1 [ <Vt>.D ]([<index>], [<Xn|SP>], <Xm>

Decode for all variants of this encoding
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
</index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
Shared decode for all encodings

integer init_scale = Uint(opcode<2:1>);
integer scale = init_scale;
integer selem = Uint(opcode<0:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = Uint(size);
        replicate = TRUE;
    when 0
        index = Uint(Q:S:size);  // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = Uint(Q:S:size<1>);  // H[0-7]
    when 2
        if size<3> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = Uint(Q:S);  // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = Uint(Q);  // D[0-1]
        scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;

Operation for all encodings

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            // ...
\[
\text{Elem}\[\text{rval}, \text{index}, \text{esize}\] = \text{Mem}\[\text{address}+\text{offs}, \text{ebytes}, \text{AccType} \_\text{VEC}\];
\]
\[
\text{V}\[\text{t}, 128\] = \text{rval};
\]
\[
\text{else} // \text{memop} = \text{MemOp} \_\text{STORE}
\]
\[
// \text{extract from one lane of 128-bit register}
\]
\[
\text{memop} = \text{MemOp} \_\text{STORE}
\]
\[
\text{Mem}\[\text{address}+\text{offs}, \text{ebytes}, \text{AccType} \_\text{VEC}\] = \text{Elem}\[\text{rval}, \text{index}, \text{esize}\];
\]
\[
\text{offs} = \text{offs} + \text{ebytes};
\]
\[
\text{t} = (\text{t} + 1) \mod 32;
\]
\[
\text{if wback then}
\]
\[
\text{if m \neq 31 then}
\]
\[
\text{offs} = \text{X}[\text{m}, 64];
\]
\[
\text{if n == 31 then}
\]
\[
\text{SP}[\] = address + offs;
\]
\[
\text{else}
\]
\[
\text{X}[\text{n}, 64] = address + offs;
\]

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.179   LD1R

Load one single-element structure and Replicate to all lanes (of one register). This instruction loads a single-element structure from memory and replicates the structure to all the lanes of the SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**No offset**

\[
\begin{array}{ccccccccccccccccc}
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 13 | 12 | 11 | 10 | 9 | 5 | 4 | 0 |
\hline
0 & Q & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & size & Rn & Rt \\
\end{array}
\]

**Encoding**

LD1R { <Vt>..<T> }, [Xn|SP]

**Decode for this encoding**

integer \( t = \text{UInt}(\text{Rt}) \);
integer \( n = \text{UInt}(\text{Rn}) \);
integer \( m = \text{integer UNKNOWN} \);
boolean \( \text{wback} = \text{FALSE} \);
boolean \( \text{tag_checked} = \text{wback} || n \neq 31 \);

**Post-index**

\[
\begin{array}{ccccccccccccccccc}
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 16 | 15 | 13 | 12 | 11 | 10 | 9 | 5 | 4 | 0 |
\hline
0 & Q & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & size & Rm | Rn | Rt \\
\end{array}
\]

**Immediate offset variant**

Applies when \( Rm = 11111 \).
LD1R { <Vt>..<T> }, [Xn|SP], <imm>

**Register offset variant**

Applies when \( Rm \neq 11111 \).
LD1R { <Vt>..<T> }, [Xn|SP], Xm

**Decode for all variants of this encoding**

integer \( t = \text{UInt}(\text{Rt}) \);
integer \( n = \text{UInt}(\text{Rn}) \);
integer \( m = \text{UInt}(\text{Rm}) \);
boolean \( \text{wback} = \text{TRUE} \);
boolean \( \text{tag_checked} = \text{wback} || n \neq 31 \);

**Assembler symbols**

\(<\text{Vt}>\)  \quad \text{Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.}

\(<\text{T}>\)  \quad \text{Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:}

88  \quad \text{when size = 00, Q = 0} \]
16B     when size = 00, Q = 1
4H      when size = 01, Q = 0
8H      when size = 01, Q = 1
2S      when size = 10, Q = 0
4S      when size = 10, Q = 1
1D      when size = 11, Q = 0
2D      when size = 11, Q = 1

<Xn>|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>  Is the post-index immediate offset, encoded in the "size" field. It can have the following values:
#1     when size = 00
#2     when size = 01
#4     when size = 10
#8     when size = 11
<Xm>   Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;
case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
    when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);    // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;

Operation for all encodings

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bias(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
  CheckSPAAlignment();
  address = SP[);
else
  address = X[n, 64];
offs = Zeros(64);
if replicate then
  // load and replicate to all elements
  for s = 0 to selem-1
    element = Mem[address+offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t, datasize] = Replicate(element, datasize DIV esize);
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
else
  // load/store one element per register
  for s = 0 to selem-1
    rval = V[t, 128];
    if memop == MemOp_LOAD then
      // insert into one lane of 128-bit register
      Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
      V[t, 128] = rval;
    else // memop == MemOp_STORE
      // extract from one lane of 128-bit register
      Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
      offs = offs + ebytes;
      t = (t + 1) MOD 32;
if wbacck then
  if m != 31 then
    offs = X[m, 64];
  if n == 31 then
    SP[] = address + offs;
  else
    X[n, 64] = address + offs;

Operational information
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.180 LD2 (multiple structures)

Load multiple 2-element structures to two registers. This instruction loads multiple 2-element structures from memory and writes the result to the two SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see LD3 (multiple structures).

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**No offset**

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 10 9]</th>
<th>5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 0 1 1 0 0 1 1 0 0 0 0 0 0 1 0 0 0</td>
<td>size Rn Rt</td>
</tr>
</tbody>
</table>

### Encoding

LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

### Decode for this encoding

```cpp
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

### Post-index

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20 16 15 12 11 10 9]</th>
<th>5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 0 1 1 0 0 1 1 0</td>
<td>size Rn Rt</td>
</tr>
</tbody>
</table>

### Immediate offset variant

Applies when Rm == 11111.

LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>

### Register offset variant

Applies when Rm != 11111.

LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>

### Decode for all variants of this encoding

```cpp
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

### Assembler symbols

<VT> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

- #16 when Q = 0
- #32 when Q = 1

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```c
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation for all encodings

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
```

---

(Courtesy of ARM)
else
    address = X[n, 64];

    offs = Zeros(64);
    for r = 0 to rpt-1
        for e = 0 to elements-1
            tt = (t + r) MOD 32;
            for s = 0 to selem-1
                rval = V[tt, datasize];
                if memop == MemOp_LOAD then
                    Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                    V[tt, datasize] = rval;
                else // memop == MemOp_STORE
                    Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                    offs = offs + ebytes;
                    tt = (tt + 1) MOD 32;

            if wback then
                if m != 31 then
                    offs = X[m, 64];
                if n == 31 then
                    SP[] = address + offs;
                else
                    X[n, 64] = address + offs;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**C7.2.181 LD2 (single structure)**

Load single 2-element structure to one lane of two registers. This instruction loads a 2-element structure from memory and writes the result to the corresponding elements of the two SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**No offset**

```
[31 30 29 28][27 26 25 24][23 22 21 20][19 18 17 16][15 13 12][11 10 9] | 5 4 | 0 |
0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 S size | Rn | Rt |
L R opcode
```

**8-bit variant**

Applies when opcode == 000.

LD2 \( \{ <Vt>.B, <Vt2>.B \}[<index>], [Xn|SP] \)

**16-bit variant**

Applies when opcode == 010 \&\& size == x0.

LD2 \( \{ <Vt>.H, <Vt2>.H \}[<index>], [Xn|SP] \)

**32-bit variant**

Applies when opcode == 100 \&\& size == 00.

LD2 \( \{ <Vt>.S, <Vt2>.S \}[<index>], [Xn|SP] \)

**64-bit variant**

Applies when opcode == 100 \&\& S == 0 \&\& size == 01.

LD2 \( \{ <Vt>.D, <Vt2>.D \}[<index>], [Xn|SP] \)

**Decode for all variants of this encoding**

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

**Post-index**

```
0 0 0 0 1 1 0 0 1 1 1 1 Rm | x x 0 S size | Rn | Rt |
L R opcode
```

**8-bit, immediate offset variant**

Applies when Rm == 11111 \&\& opcode == 000.

LD2 \( \{ <Vt>.B, <Vt2>.B \}[<index>], [Xn|SP], #2 \)
**8-bit, register offset variant**

Applies when $Rm \nequiv 11111$ && $opcode = 000$.

```
LD2 \{ <Vt>.B, <Vt2>.B \}[@<index>], [Xn|SP], <Xm>
```

**16-bit, immediate offset variant**

Applies when $Rm = 11111$ && $opcode = 010$ && $size = 00$.

```
LD2 \{ <Vt>.H, <Vt2>.H \}[@<index>], [Xn|SP], #4
```

**16-bit, register offset variant**

Applies when $Rm \nequiv 11111$ && $opcode = 010$ && $size = 00$.

```
LD2 \{ <Vt>.H, <Vt2>.H \}[@<index>], [Xn|SP], <Xm>
```

**32-bit, immediate offset variant**

Applies when $Rm = 11111$ && $opcode = 100$ && $size = 00$.

```
LD2 \{ <Vt>.S, <Vt2>.S \}[@<index>], [Xn|SP], #8
```

**32-bit, register offset variant**

Applies when $Rm \nequiv 11111$ && $opcode = 100$ && $size = 00$.

```
LD2 \{ <Vt>.S, <Vt2>.S \}[@<index>], [Xn|SP], <Xm>
```

**64-bit, immediate offset variant**

Applies when $Rm = 11111$ && $opcode = 100$ && $S = 0$ && $size = 01$.

```
LD2 \{ <Vt>.D, <Vt2>.D \}[@<index>], [Xn|SP], #16
```

**64-bit, register offset variant**

Applies when $Rm \nequiv 11111$ && $opcode = 100$ && $S = 0$ && $size = 01$.

```
LD2 \{ <Vt>.D, <Vt2>.D \}[@<index>], [Xn|SP], <Xm>
```

**Decode for all variants of this encoding**

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

**Assembler symbols**

- `<Vt>`  
  Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

- `<Vt2>`  
  Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

- `<index>`  
  For the 8-bit variant: is the element index, encoded in "Q:S:size".
  For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
  For the 32-bit variant: is the element index, encoded in "Q:S".
  For the 64-bit variant: is the element index, encoded in "Q".

- `<Xn|SP>`  
  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared decode for all encodings**

```plaintext
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);    // H[0-7]
  when 2
    if size<> == '1' then UNDEFINED;
    if size<> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
    scale = 3;
  else
    MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
  end if

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

**Operation for all encodings**

```plaintext
if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
EndIf;

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];
endif

offs = Zeros(64);
if replicate then
  // load and replicate to all elements
  for s = 0 to selem-1
    element = Mem[address+offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t, datasize] = Replicate(element, datasize DIV esize);
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
  end for
else
  // load/store one element per register
```
for s = 0 to selem-1
    rval = V[t, 128];
    if memop == MemOp_LOAD then
        // insert into one lane of 128-bit register
        Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
        V[t, 128] = rval;
    else // memop == MemOp_STORE
        // extract from one lane of 128-bit register
        Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
    t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.182   LD2R

Load single 2-element structure and Replicate to all lanes of two registers. This instruction loads a 2-element structure from memory and replicates the structure to all the lanes of the two SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15</th>
<th>13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 0 1 1 0 1 0 1 1 0 0 0 0 1 1 0 0</td>
<td>size</td>
<td>Rn</td>
<td>Rt</td>
</tr>
</tbody>
</table>

Encoding

LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]  

Decode for this encoding

integer t = UInt(Rt);  
integer n = UInt(Rn);  
integer m = integer UNKNOWN;  
boolean wback = FALSE;  
boolean tag_checked = wback || n != 31;

Post-index

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15</th>
<th>13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 0 1 1 0 1 1 1</td>
<td>Rm</td>
<td>1 1 0 0</td>
<td>size</td>
</tr>
</tbody>
</table>

Immediate offset variant

 Applies when Rm == 11111.
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>

Register offset variant

 Applies when Rm != 11111.
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>

Decode for all variants of this encoding

integer t = UInt(Rt);  
integer n = UInt(Rn);  
integer m = UInt(Rm);  
boolean wback = TRUE;  
boolean tag_checked = wback || n != 31;

Assembler symbols

<Vt>   Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<T>   Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

88 when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
1D when size = 11, Q = 0
2D when size = 11, Q = 1

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the post-index immediate offset, encoded in the "size" field. It can have the following values:
#2 when size = 00
#4 when size = 01
#8 when size = 10
#16 when size = 11
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;
case scale of
when 3
   // load and replicate
   if L == '0' || S == '1' then UNDEFINED;
scale = UInt(size);
   replicate = TRUE;
when 0
   index = UInt(Q:S:size);    // B[0-15]
when 1
   if size<0> == '1' then UNDEFINED;
   index = UInt(Q:S:size<1>);    // H[0-7]
when 2
   if size<1> == '1' then UNDEFINED;
   if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
   else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
scale = 3;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;

Operation for all encodings

if HaveMTE2Ext() then
   SetTagCheckedInstruction(tag_checked);
CheckFPAdvSIMEnabled64();
bits(64) address;
bites(64) offs;
bites(128) rval;
bite(esimal) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];
offs = Zeros(64);
if replicate then
  // load and replicate to all elements
  for s = 0 to selem-1
    element = Mem[address+offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t, datasize] = Replicate(element, datasize DIV esize);
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
else
  // load/store one element per register
  for s = 0 to selem-1
    rval = V[t, 128];
    if memop == MemOp_LOAD then
      // insert into one lane of 128-bit register
      Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
      V[t, 128] = rval;
    else // memop == MemOp_STORE
      // extract from one lane of 128-bit register
      Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
      offs = offs + ebytes;
      t = (t + 1) MOD 32;
if wback then
  if m != 31 then
    offs = X[m, 64];
  if n == 31 then
    SP[] = address + offs;
  else
    X[n, 64] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.183   LD3 (multiple structures)

Load multiple 3-element structures to three registers. This instruction loads multiple 3-element structures from memory and writes the result to the three SIMD&FP registers, with de-interleaving.

The following figure shows the operation of de-interleaving of a LD3.16 (multiple 3-element structures) instruction:

![Diagram of LD3 operation]

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**No offset**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 10 9 5 4 0 |
|--------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 | Q | 0 0 1 1 0 0 0 1 0 0 0 0 0 0 1 0 0  | size | Rn | Rt |
```

**Encoding**

LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [ <Xn|SP> ]

**Decode for this encoding**

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

**Post-index**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 |
|--------------------------|-----------------|-----------------|-----------------|-----------------|
| 0 | Q | 0 0 1 1 0 0 1 0 1 0 0 | size | Rn | Rt |
```

**Immediate offset variant**

Applies when `Rm == 11111`.

LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [ <Xn|SP>, <imm> ]
Register offset variant

Applies when \( Rm \neq 11111 \).

\[
LD3 \{ <Vt>.<I>, <Vt2>.<I>, <Vt3>.<I> \}, [<Xn|SP>], <Xm>
\]

Decode for all variants of this encoding

\[
integer \ t = \text{UInt}(Rt);
integer \ n = \text{UInt}(Rn);
integer \ m = \text{UInt}(Rm);
boolean \ wback = \text{TRUE};
boolean \ tag\_checked = wback || n \neq 31;
\]

Assembler symbols

- \(<Vt>\) is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- \(<I>\) is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - 8B when \( size = 00, Q = 0 \)
  - 16B when \( size = 00, Q = 1 \)
  - 4H when \( size = 01, Q = 0 \)
  - 8H when \( size = 01, Q = 1 \)
  - 2S when \( size = 10, Q = 0 \)
  - 4S when \( size = 10, Q = 1 \)
  - 2D when \( size = 11, Q = 1 \)

The encoding \( size = 11, Q = 0 \) is reserved.

- \(<Vt2>\) is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- \(<Vt3>\) is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- \(<Xn|SP>\) is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- \(<imm>\) is the post-index immediate offset, encoded in the "Q" field. It can have the following values:
  - #24 when \( Q = 0 \)
  - #48 when \( Q = 1 \)
- \(<Xm>\) is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

\[
\text{MemOp} \ \text{memop} = \begin{cases} 
\text{MemOp LOAD} & \text{if } L = '1' \\
\text{MemOp STORE} & \text{else}
\end{cases};
\]

\[
\text{integer} \ \text{datasize} = \begin{cases} 
128 & \text{if } Q = '1' \\
64 & \text{else}
\end{cases};
\]

\[
\text{integer} \ \text{esize} = 8 \times \text{UInt}(\text{size});
\]

\[
\text{integer} \ \text{rpt}; \quad \text{// number of iterations}
\]

\[
\text{integer} \ \text{selem}; \quad \text{// structure elements}
\]

\[
\text{case} \ \text{opcode} \ \text{of}\
\text{when} \ '0000' \ \text{rpt} = 1; \ \text{selem} = 4; \quad \text{// LD/ST4 (4 registers)}
\text{when} \ '0010' \ \text{rpt} = 4; \ \text{selem} = 1; \quad \text{// LD/ST1 (4 registers)}
\text{when} \ '0100' \ \text{rpt} = 1; \ \text{selem} = 3; \quad \text{// LD/ST3 (3 registers)}
\text{when} \ '0110' \ \text{rpt} = 3; \ \text{selem} = 1; \quad \text{// LD/ST1 (3 registers)}
\text{when} \ '0111' \ \text{rpt} = 1; \ \text{selem} = 1; \quad \text{// LD/ST1 (1 register)}
\text{when} \ '1000' \ \text{rpt} = 1; \ \text{selem} = 2; \quad \text{// LD/ST2 (2 registers)}
\text{when} \ '1010' \ \text{rpt} = 2; \ \text{selem} = 1; \quad \text{// LD/ST1 (2 registers)}
\text{otherwise} \ \text{UNDEFINED};
\]
// .1D format only permitted with LD1 & ST1
if size:Q == '110' \&\& selem != 1 then UNDEFINED;

Operation for all encodings

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.184   LD3 (single structure)

Load single 3-element structure to one lane of three registers. This instruction loads a 3-element structure from memory and writes the result to the corresponding elements of the three SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

[<index>], [<Xn|SP>]

8-bit variant

Applies when opcode == 001.


16-bit variant

Applies when opcode == 011 && size == x0.

LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]

32-bit variant

Applies when opcode == 101 && size == 00.


64-bit variant

Applies when opcode == 101 && S == 0 && size == 01.

LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]

Decode for all variants of this encoding

```plaintext
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

```
0 Q 0 0 1 1 0 1 0 1 0 0 0 0 0 0 0 0 x x 1 S size Rn Rt
L R opcode
```

8-bit, immediate offset variant

Applies when Rm == 11111 && opcode == 001.

LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
8-bit, register offset variant
Applies when Rm != 11111 && opcode == 001.
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 011 && size == x0.
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6

16-bit, register offset variant
Applies when Rm != 11111 && opcode == 011 && size == x0.
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 101 && size == 00.
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12

32-bit, register offset variant
Applies when Rm != 11111 && opcode == 101 && size == 00.
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 101 && S == 0 && size == 01.
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24

64-bit, register offset variant
Applies when Rm != 11111 && opcode == 101 && S == 0 && size == 01.
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>

Decode for all variants of this encoding
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared decode for all encodings**

```plaintext
case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q); // D[0-1]
  scale = 3;
```

**Operation for all encodings**

```plaintext
if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
CheckFPAdvSIMDEnabled64();
```
```plaintext
bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;
```
```plaintext
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];
```
```plaintext
offs = Zeros(64);
if replicate then
  // load and replicate to all elements
  for s = 0 to selem-1
    element = Mem[address+offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t, datasize] = Replicate(element, datasize DIV esize);
  offs = offs + ebytes;
  t = (t + 1) MOD 32;
else
  // load/store one element per register
```
```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```
for s = 0 to selem-1
    rval = V[t, 128];
    if memop == MemOp_LOAD then
        // insert into one lane of 128-bit register
        Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
        V[t, 128] = rval;
    else // memop == MemOp_STORE
        // extract from one lane of 128-bit register
        Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
    if wback then
        if m != 31 then
            offs = X[m, 64];
        if n == 31 then
            SP[] = address + offs;
        else
            X[n, 64] = address + offs;
    Operational information
    If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.185  LD3R

Load single 3-element structure and Replicate to all lanes of three registers. This instruction loads a 3-element
structure from memory and replicates the structure to all the lanes of the three SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

No offset

Immediate offset variant

Applies when \( Rm = 11111 \).

LD3R \( \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> \}, [Xn|SP], \langle \text{imm} \rangle \)

Register offset variant

Applies when \( Rm != 11111 \).

LD3R \( \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> \}, [Xn|SP], \langle Xm \rangle \)

Decode for all variants of this encoding

\begin{align*}
\text{integer } t &= \text{UInt}(Rt); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{integer UNKNOWN}; \\
\text{boolean } wback &= \text{FALSE}; \\
\text{boolean } tag\_checked &= wback \| n != 31;
\end{align*}

Assembler symbols

\(<Vt>\)  Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

\(<T>\)  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

\begin{align*}
88 & \quad \text{when size = 00, Q = 0}
\end{align*}
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
1D when size = 11, Q = 0
2D when size = 11, Q = 1

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the post-index immediate offset, encoded in the "size" field. It can have the following values:
#3 when size = 00
#6 when size = 01
#12 when size = 10
#24 when size = 11
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```plaintext
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);    // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;
  MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
  integer datasize = if Q == '1' then 128 else 64;
  integer esize = 8 << scale;

Operation for all encodings

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
```


CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t, datasize] = Replicate(element, datasize DIV esize);
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t, 128] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.186  LD4 (multiple structures)

Load multiple 4-element structures to four registers. This instruction loads multiple 4-element structures from memory and writes the result to the four SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see LD3 (multiple structures).

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**No offset**

![Instruction format](image)

**Encoding**

LD4 (Vt<T>, Vt2<T>, Vt3<T>, Vt4<T>), [Xn|SP]

**Decode for this encoding**

```plaintext
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

**Post-index**

![Instruction format](image)

**Immediate offset variant**

Applies when Rm == 11111.

LD4 (Vt<T>, Vt2<T>, Vt3<T>, Vt4<T>), [Xn|SP], <imm>

**Register offset variant**

Applies when Rm != 11111.

LD4 (Vt<T>, Vt2<T>, Vt3<T>, Vt4<T>), [Xn|SP], <Xm>

**Decode for all variants of this encoding**

```plaintext
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

**Assembler symbols**

\(<Vt>\)  
Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<T>  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
8B  when size = 00, Q = 0
16B when size = 00, Q = 1
4H  when size = 01, Q = 0
8H  when size = 01, Q = 1
2S  when size = 10, Q = 0
4S  when size = 10, Q = 1
2D  when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.

<Vt2>  Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3>  Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4>  Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>  Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:
#32  when Q = 0
#64  when Q = 1
<Xm>  Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;

Operation for all encodings

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.187   LD4 (single structure)

Load single 4-element structure to one lane of four registers. This instruction loads a 4-element structure from memory and writes the result to the corresponding elements of the four SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**No offset**

```
```

**8-bit variant**

Applies when opcode == 001.

```
```

**16-bit variant**

Applies when opcode == 011 && size == x0.

```
```

**32-bit variant**

Applies when opcode == 101 && size == 00.

```
```

**64-bit variant**

Applies when opcode == 101 && S == 0 && size == 01.

```
```

**Decode for all variants of this encoding**

```java
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

**Post-index**

```
```

**8-bit, immediate offset variant**

Applies when Rn == 11111 && opcode == 001.
8-bit, register offset variant
Applies when Rm != 11111 && opcode == 001.

16-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 011 && size == x0.

16-bit, register offset variant
Applies when Rm != 11111 && opcode == 011 && size == x0.

32-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 101 && size == 00.

32-bit, register offset variant
Applies when Rm != 11111 && opcode == 101 && size == 00.

64-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 101 && S == 0 && size == 01.

64-bit, register offset variant
Applies when Rm != 11111 && opcode == 101 && S == 0 && size == 01.

Decode for all variants of this encoding
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler symbols
<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);    // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;

Operation for all encodings

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;

constant integer ebytes = esize DIV 8;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

offs = Zeros(64);
if replicate then
  // load and replicate to all elements
  for s = 0 to selem-1
    element = Mem[address+offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t, datasize] = Replicate(element, datasize DIV esize);
    offs = offs + ebytes;
  t = (t + 1) MOD 32;
else
// load/store one element per register
for s = 0 to selem-1
  rval = V[t, 128];
  if memop == MemOp_LOAD then
    // insert into one lane of 128-bit register
    Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
    V[t, 128] = rval;
  else // memop == MemOp_STORE
    // extract from one lane of 128-bit register
    Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
    offs = offs + ebytes;
  t = (t + 1) MOD 32;
if wback then
  if m != 31 then
    offs = X[m, 64];
  if n == 31 then
    SP[] = address + offs;
  else
    X[n, 64] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.188   LD4R

Load single 4-element structure and Replicate to all lanes of four registers. This instruction loads a 4-element structure from memory and replicates the structure to all the lanes of the four SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 13 12|11 10 9 | 5 4 | 0 ]
0 Q 0 0 1 1 0 1 0 1 1 0 0 0 0 1 1 1 0 size Rn Rt
L R opcode S
```

Encoding

```
```

Decode for this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

```
[31 30 29 28|27 26 25 24|23 22 21 20|16|15 13 12|11 10 9 | 5 4 | 0 ]
0 Q 0 0 1 1 0 1 1 1 | Rm 1 1 1 0 size Rn Rt
L R opcode S
```

Immediate offset variant

Applies when 

```
```

Register offset variant

Applies when 

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

- `<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

  - 88 when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
1D when size = 11, Q = 0
2D when size = 11, Q = 1

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the post-index immediate offset, encoded in the "size" field. It can have the following values:
#4 when size = 00
#8 when size = 01
#16 when size = 10
#32 when size = 11
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```plaintext
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q); // D[0-1]
    scale = 3;
endcase

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
intrinsic datasize = if Q == '1' then 128 else 64;
intrinsic esize = 8 << scale;
```
Operation for all encodings

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t, 128] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.189  LDNP (SIMD&FP)

Load Pair of SIMD&FP registers, with Non-temporal hint. This instruction loads a pair of SIMD&FP registers from memory, issuing a hint to the memory system that the access is non-temporal. The address that is used for the load is calculated from a base register value and an optional immediate offset.

For information about non-temporal pair instructions, see Load/store SIMD and floating-point non-temporal pair on page C3-265.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### 32-bit variant
Applies when \( \text{opc} == 00 \).

LDNP <St1>, <St2>, [<Xn|SP>{, #imm}]

### 64-bit variant
Applies when \( \text{opc} == 01 \).

LDNP <Dt1>, <Dt2>, [<Xn|SP>{, #imm}]

### 128-bit variant
Applies when \( \text{opc} == 10 \).

LDNP <Qt1>, <Qt2>, [<Xn|SP>{, #imm}]

**Decode for all variants of this encoding**

// Empty.

**Notes for all encodings**

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly LDNP (SIMD&FP) on page K1-11592.

**Assembler symbols**

- \(<\text{Dt1}\>)  Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- \(<\text{Dt2}\>)  Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- \(<\text{Qt1}\>)  Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- \(<\text{Qt2}\>)  Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- \(<\text{St1}\>)  Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- \(<\text{St2}\>)  Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- \(<\text{Xn|SP}\>)  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as \(<\text{imm}>/4\).

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as \(<\text{imm}>/8\).

For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as \(<\text{imm}>/16\).

### Shared decode for all encodings

\[
\text{integer } n = \text{UInt}(Rn); \\
\text{integer } t = \text{UInt}(Rt); \\
\text{integer } t2 = \text{UInt}(Rt2); \\
\text{if } opc == '11' \text{ then UNDEFINED}; \\
\text{integer } scale = 2 + \text{UInt}(opc); \\
\text{integer } datasize = 8 \ll scale; \\
\text{bits}(64) \text{ offset} = \text{LSL}(\text{SignExtend}(\text{imm7}, 64), \text{scale}); \\
\text{boolean } \text{tag\_checked} = n != 31; \\
\text{boolean } \text{rt\_unknown} = \text{FALSE}; \\
\]

\[
\text{if } t == t2 \text{ then} \\
\text{Constraint } c = \text{ConstrainUnpredictable}((\text{Unpredictable}\_\text{LDPOVERLAP}); \\
\text{assert } c \in \{\text{Constraint}\_\text{UNKNOWN}, \text{Constraint}\_\text{UNDEF}, \text{Constraint}\_\text{NOP}\}; \\
\text{case } c \text{ of} \\
\text{ when } \text{Constraint}\_\text{UNKNOWN} \text{ rt\_unknown} = \text{TRUE}; \quad \// \text{result is UNKNOWN} \\
\text{ when } \text{Constraint}\_\text{UNDEF} \text{ UNDEFINED}; \\
\text{ when } \text{Constraint}\_\text{NOP} \text{ EndOfInstruction();}
\]

### Operation

\[
\text{CheckFPEnabled64();} \\
\text{bits}(64) \text{ address}; \\
\text{bits}(\text{datasize}) \text{ data1}; \\
\text{bits}(\text{datasize}) \text{ data2}; \\
\text{constant integer } d\text{bytes} = \text{datasize} \text{ DIV} 8; \\
\text{if } \text{HaveMTE2Ext()} \text{ then} \\
\text{SetTagCheckedInstruction}(\text{tag\_checked}); \\
\text{if } n == 31 \text{ then} \\
\text{CheckSPAlignment();} \\
\text{address} = \text{SP}[]; \\
\text{else} \\
\text{address} = \text{X}[n, 64]; \\
\text{address} = \text{address} + \text{offset}; \\
\text{data1} = \text{Mem}[\text{address}, d\text{bytes}, \text{AccType}\_\text{VECEXTRACT}]; \\
\text{data2} = \text{Mem}[\text{address}+d\text{bytes}, d\text{bytes}, \text{AccType}\_\text{VECEXTRACT}]; \\
\text{if } \text{rt\_unknown} \text{ then} \\
\text{data1} = \text{bits}(\text{datasize}) \text{ UNKNOWN}; \\
\text{data2} = \text{bits}(\text{datasize}) \text{ UNKNOWN}; \\
\text{V}[t, \text{datasize}] = \text{data1}; \\
\text{V}[t2, \text{datasize}] = \text{data2};
\]

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.190 LDP (SIMD&FP)

Load Pair of SIMD&FP registers. This instruction loads a pair of SIMD&FP registers from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Post-index

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**32-bit variant**

Applies when opc == 00.

LDP <St1>, <St2>, [<Xn|SP>], #<imm>

**64-bit variant**

Applies when opc == 01.

LDP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>

**128-bit variant**

Applies when opc == 10.

LDP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>

**Decode for all variants of this encoding**

boolean wback = TRUE;
boolean postindex = TRUE;

Pre-index

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**32-bit variant**

Applies when opc == 00.

LDP <St1>, <St2>, [<Xn|SP>], #<imm>!

**64-bit variant**

Applies when opc == 01.

LDP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>!

**128-bit variant**

Applies when opc == 10.

LDP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>!
**Decode for all variants of this encoding**

boolean wback = TRUE;
boolean postindex = FALSE;

### Signed offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>1 0 1</td>
<td>1 0 0</td>
<td>1 imm7</td>
<td>Rt2</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**32-bit variant**

Applies when opc == 00.

LDP <St1>, <St2>, [.<Xn|SP>], #<imm>]

**64-bit variant**

Applies when opc == 01.

LDP <Dt1>, <Dt2>, [.<Xn|SP>], #<imm>]

**128-bit variant**

Applies when opc == 10.

LDP <Qt1>, <Qt2>, [.<Xn|SP>], #<imm>]

**Decode for all variants of this encoding**

boolean wback = FALSE;
boolean postindex = FALSE;

### Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *LDP (SIMD&FP)* on page K1-11592, and particularly *LDNP (SIMD&FP)* on page K1-11592.

### Assembler symbols

- **<Dt1>** Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- **<Dt2>** Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- **<Qt1>** Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- **<Qt2>** Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- **<St1>** Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- **<St2>** Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<imm>** For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.
For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.

For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.

For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared decode for all encodings

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainsUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation for all encodings

```plaintext
CheckFPEnabled64();
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
if !postindex then
    address = address + offset;
data1 = Mem[address, dbytes, AccType_VEC];
data2 = Mem[address+dbytes, dbytes, AccType_VEC];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
data2 = bits(datasize) UNKNOWN;
V[t, datasize] = data1;
V[t2, datasize] = data2;
if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
```

C7-2484  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. Non-Confidential
else
    x[n, 64] = address;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.191 LDR (immediate, SIMD&FP)

Load SIMD&FP Register (immediate offset). This instruction loads an element from memory, and writes the result as a scalar to the SIMD&FP register. The address that is used for the load is calculated from a base register value, a signed immediate offset, and an optional offset that is a multiple of the element size.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Post-index**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size: 1111</td>
<td>imm9: 010</td>
<td>Rn: x10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opc: 01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**8-bit variant**

Applies when size == 00 && opc == 01.
LDR <Bt>, [<Xn|SP>], #<simm>

**16-bit variant**

Applies when size == 01 && opc == 01.
LDR <Ht>, [<Xn|SP>], #<simm>

**32-bit variant**

Applies when size == 10 && opc == 01.
LDR <St>, [<Xn|SP>], #<simm>

**64-bit variant**

Applies when size == 11 && opc == 01.
LDR <Dt>, [<Xn|SP>], #<simm>

**128-bit variant**

Applies when size == 00 && opc == 11.
LDR <Qt>, [<Xn|SP>], #<simm>

**Decode for all variants of this encoding**

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bmts(64) offset = SignExtend(imm9, 64);

**Pre-index**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size: 1111</td>
<td>imm9: 110</td>
<td>Rn: x10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opc: 01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8-bit variant
Applies when size == 00 && opc == 01.
LDR <Bt>, [<Xn|SP>, #<simm>]!

16-bit variant
Applies when size == 01 && opc == 01.
LDR <Ht>, [<Xn|SP>, #<simm>]!

32-bit variant
Applies when size == 10 && opc == 01.
LDR <St>, [<Xn|SP>, #<simm>]!

64-bit variant
Applies when size == 11 && opc == 01.
LDR <Dt>, [<Xn|SP>, #<simm>]!

128-bit variant
Applies when size == 00 && opc == 11.
LDR <Qt>, [<Xn|SP>, #<simm>]!

Decode for all variants of this encoding
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bounds(64) offset = SignExtend(imm9, 64);

Unsigned offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size 1 1 1 1 0 1 x 1</td>
<td>imm12</td>
<td>Rn</td>
<td>Rt</td>
</tr>
</tbody>
</table>

opc

8-bit variant
Applies when size == 00 && opc == 01.
LDR <Bt>, [<Xn|SP>{, #<pimm>}]

16-bit variant
Applies when size == 01 && opc == 01.
LDR <Ht>, [<Xn|SP>{, #<pimm>}]

32-bit variant
Applies when size == 10 && opc == 01.
LDR <St>, [<Xn|SP>{, #<pimm>}]
### 64-bit variant

Applies when size == 11 && opc == 01.

LDR <Dt>, [<Xn|SP>{, #<pimm>}]

### 128-bit variant

Applies when size == 00 && opc == 11.

LDR <Qt>, [<Xn|SP>{, #<pimm>}]

### Decode for all variants of this encoding

```plaintext
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

### Assembler symbols

- `<Bt>`: Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Dt>`: Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Ht>`: Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Qt>`: Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<St>`: Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>`: Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- `<pimm>`: For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as `<pimm>/2`.

For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as `<pimm>/4`.

For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as `<pimm>/8`.

For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as `<pimm>/16`.

### Shared decode for all encodings

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
```

### Operation for all encodings

```plaintext
CheckFPEnable64();
bits(64) address;
bits(datasize) data;
```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t, datasize];
        Mem[address, datasize DIV 8, AccType_VEC] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t, datasize] = data;

    if wback then
        if postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n, 64] = address;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.192  LDR (literal, SIMD&FP)

Load SIMD&FP Register (PC-relative literal). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from the PC value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

32-bit variant

Applies when \( \text{opc} == 00 \).

\[
\text{LDR} \; \text{<St>}, \; \text{<label>}
\]

64-bit variant

Applies when \( \text{opc} == 01 \).

\[
\text{LDR} \; \text{<Dt>}, \; \text{<label>}
\]

128-bit variant

Applies when \( \text{opc} == 10 \).

\[
\text{LDR} \; \text{<Qt>}, \; \text{<label>}
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } t &= \text{UInt}(\text{Rt}); \\
\text{integer } \text{size} &= \text{bits}(64) \text{ offset}; \\
\text{case } \text{opc} \text{ of} \\
& \quad \text{when } '00' \\
& \quad \quad \text{size} = 4; \\
& \quad \text{when } '01' \\
& \quad \quad \text{size} = 8; \\
& \quad \text{when } '10' \\
& \quad \quad \text{size} = 16; \\
& \quad \text{when } '11' \\
& \quad \quad \text{UNDEFINED}; \\
\text{offset} &= \text{SignExtend}(\text{imm19}:'00', 64);
\end{align*}
\]

Assembler symbols

- \( \text{<Dt>} \) Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
- \( \text{<Qt>} \) Is the 128-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
- \( \text{<St>} \) Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
- \( \text{<label>} \) Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.
Operation

bits(64) address = PC[] + offset;
bits(size=8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);
CheckFPEnabled64();

data = Mem[address, size, AccType_VEC];
V[t, size=8] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.193  **LDR (register, SIMD&FP)**

Load SIMD&FP Register (register offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### 8-bit variant

Applies when `size == 00 && opc == 01 && option != 011`.

LDR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]  

### 8-bit variant

Applies when `size == 00 && opc == 01 && option == 011`.

LDR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]  

### 16-bit variant

Applies when `size == 01 && opc == 01`.

LDR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]  

### 32-bit variant

Applies when `size == 10 && opc == 01`.

LDR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]  

### 64-bit variant

Applies when `size == 11 && opc == 01`.

LDR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]  

### 128-bit variant

Applies when `size == 00 && opc == 11`.

LDR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]  

### Decode for all variants of this encoding

```c
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
if option<1> == '0' then UNDEFINED;  // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

### Assembler symbols

- **<Bt>** is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>
Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<ht>
Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt>
Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St>
Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Xn|SP>
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm>
When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend>
For the 8-bit variant: is the index extend specifier, encoded in the "option" field. It can have the following values:

UXTW when option = 010
SXTW when option = 110
SXTX when option = 111

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

UXTW when option = 010
LSL when option = 011
SXTW when option = 110
SXTX when option = 111

<amount>
For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

#0 when S = 0
#1 when S = 1

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

#0 when S = 0
#2 when S = 1

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

#0 when S = 0
#3 when S = 1

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

#0 when S = 0
#4 when S = 1
Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;

Operation

bits(64) offset = ExtendReg(m, extend_type, shift, 64);
CheckFPEnabled64();
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

address = address + offset;

case memop of
  when MemOp_STORE
    data = V[t, datasize];
    Mem[address, datasize DIV 8, AccType_VEC] = data;
  when MemOp_LOAD
    data = Mem[address, datasize DIV 8, AccType_VEC];
    V[t, datasize] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.194   LDUR (SIMD&FP)

Load SIMD&FP Register (unscaled offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

8-bit variant
Applies when \( \text{size} == 00 \) && \( \text{opc} == 01 \).

LDUR <Bt>, [<Xn|SP>{, #<simm>}]

16-bit variant
Applies when \( \text{size} == 01 \) && \( \text{opc} == 01 \).

LDUR <Ht>, [<Xn|SP>{, #<simm>}]

32-bit variant
Applies when \( \text{size} == 10 \) && \( \text{opc} == 01 \).

LDUR <St>, [<Xn|SP>{, #<simm>}]

64-bit variant
Applies when \( \text{size} == 11 \) && \( \text{opc} == 01 \).

LDUR <Dt>, [<Xn|SP>{, #<simm>}]

128-bit variant
Applies when \( \text{size} == 00 \) && \( \text{opc} == 11 \).

LDUR <Qt>, [<Xn|SP>{, #<simm>}]

**Decode for all variants of this encoding**

\[
\text{integer scale} = \text{UInt}(\text{opc}<1>:\text{size});
\]
\[
\text{if scale > 4 then UNDEFINED;}
\]
\[
\text{bits(64) offset} = \text{SignExtend}(\text{imm9}, 64);
\]

**Assembler symbols**

- \(<\text{Bt}>\): Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- \(<\text{Dt}>\): Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- \(<\text{Ht}>\): Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- \(<\text{Qt}>\): Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- \(<\text{St}>\): Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- \(<\text{Xn}|\text{SP}>\): Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```plaintext
CheckFPEnabled64();
bits(64) address;
bits(datasize) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
address = address + offset;
case memop of
    when MemOp_STORE
        data = V[t, datasize];
        Mem[address, datasize DIV 8, AccType_VEC] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t, datasize] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.195 MLA (by element)

Multiply-Add to accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

 Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  4H when size = 01, Q = 0
  8H when size = 01, Q = 1
  2S when size = 10, Q = 0
  4S when size = 10, Q = 1
The following encodings are reserved:
  • size = 00, Q = x.
  • size = 11, Q = x.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:
  0:Rm when size = 01
M:Rm when size = 10
The following encodings are reserved:
• size = 00.
• size = 11.
Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
The following encodings are reserved:
• size = 00.
• size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:
H:L:M when size = 01
H:L when size = 10
The following encodings are reserved:
• size = 00.
• size = 11.

Operation

CheckFPAdvSIMDEnabled64();
bias(datasize) operand1 = V[n, datasize];
bias(idxdsize) operand2 = V[m, idxdsize];
bias(datasize) operand3 = V[d, datasize];
bias(datasize) result;
integer element1;
integer element2;
bias(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1+element2)<<(esize-1:0);
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.196 MLA (vector)

Multiply-Add to accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  8B when size = 00, Q = 0
  16B when size = 00, Q = 1
  4H when size = 01, Q = 0
  8H when size = 01, Q = 1
  2S when size = 10, Q = 0
  4S when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  product = (UInt(element1) * UInt(element2))<esize-1:0>;
  if sub_op then
    Elem[result, e, esize] = Elem[operand3, e, esize] - product;
  else
    Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.197  MLS (by element)

Multiply-Subtract from accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding
MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

Decode for this encoding
integer idxdsz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');

Assembler symbols
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  4H  when size = 01, Q = 0
  8H  when size = 01, Q = 1
  2S  when size = 10, Q = 0
  4S  when size = 10, Q = 1
The following encodings are reserved:
  •  size = 00, Q = x.
  •  size = 11, Q = x.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:
  0:Rm  when size = 01
M:Rm when size = 10

The following encodings are reserved:

• size = 00.
• size = 11.

Restricted to V0-V15 when element size \(<T_s>\) is H.

<\(T_s\)> is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10

The following encodings are reserved:

• size = 00.
• size = 11.

<index> is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01
H:L when size = 10

The following encodings are reserved:

• size = 00.
• size = 11.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.198  MLS (vector)

Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');

Assembler symbols

<Vd>    Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>    Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
        8B when size = 00, Q = 0
        16B when size = 00, Q = 1
        4H when size = 01, Q = 0
        8H when size = 01, Q = 1
        2S when size = 10, Q = 0
        4S when size = 10, Q = 1
        The encoding size = 11, Q = x is reserved.
<Vn>    Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>    Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.199 MOV (scalar)

Move vector element to scalar. This instruction duplicates the specified vector element in the SIMD&FP source register into a scalar, and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is an alias of the DUP (element) instruction. This means that:

- The encodings in this description are named to match the encodings of DUP (element).
- The description of DUP (element) gives the operational pseudocode for this instruction.

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\hline
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & Rn & Rd \\
\hline
\end{array}
\]

**Encoding**

MOV <V><d>, <Vn>.<T>[<index>]

is equivalent to

DUP <V><d>, <Vn>.<T>[<index>]

and is always the preferred disassembly.

**Assembler symbols**

- **<V>** Is the destination width specifier, encoded in the "imm5" field. It can have the following values:
  - B when \( imm5 = xxxx1 \)
  - H when \( imm5 = xxx10 \)
  - S when \( imm5 = xx100 \)
  - D when \( imm5 = x1000 \)
  - The encoding \( imm5 = x0000 \) is reserved.

- **<d>** Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- **<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- **<T>** Is the element width specifier, encoded in the "imm5" field. It can have the following values:
  - B when \( imm5 = xxxx1 \)
  - H when \( imm5 = xxx10 \)
  - S when \( imm5 = xx100 \)
  - D when \( imm5 = x1000 \)
  - The encoding \( imm5 = x0000 \) is reserved.

- **<index>** Is the element index encoded in the "imm5" field. It can have the following values:
  - \( imm5<4:1> \) when \( imm5 = xxxx1 \)
  - \( imm5<4:2> \) when \( imm5 = xxx10 \)
  - \( imm5<4:3> \) when \( imm5 = xx100 \)
  - \( imm5<4> \) when \( imm5 = x1000 \)
  - The encoding \( imm5 = x0000 \) is reserved.
Operation

The description of DUP (element) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.200 MOV (element)

Move vector element to another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is an alias of the INS (element) instruction. This means that:

- The encodings in this description are named to match the encodings of INS (element).
- The description of INS (element) gives the operational pseudocode for this instruction.

```
Encoding
MOV <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

is equivalent to

```
INS <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

and is always the preferred disassembly.

**Assembler symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Ts>` Is an element size specifier, encoded in the "imm5" field. It can have the following values:
  - `B` when `imm5 = xxxx1`
  - `H` when `imm5 = xxx10`
  - `S` when `imm5 = xx100`
  - `D` when `imm5 = x1000`
    The encoding `imm5 = x0000` is reserved.
- `<index1>` Is the destination element index encoded in the "imm5" field. It can have the following values:
  - `imm5<4:1>` when `imm5 = xxxx1`
  - `imm5<4:2>` when `imm5 = xxx10`
  - `imm5<4:3>` when `imm5 = xx100`
  - `imm5<4>` when `imm5 = x1000`
    The encoding `imm5 = x0000` is reserved.
- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- `<index2>` Is the source element index encoded in the "imm5:imm4" field. It can have the following values:
  - `imm4<3:0>` when `imm5 = xxxx1`
  - `imm4<3:1>` when `imm5 = xxx10`
  - `imm4<3:2>` when `imm5 = xx100`
imm<4:3> when imm5 = x1000
The encoding imm5 = x0000 is reserved.
Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.

**Operation**

The description of INS (element) gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.201 MOV (from general)

Move general-purpose register to a vector element. This instruction copies the contents of the source
general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining
bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

This instruction is an alias of the INS (general) instruction. This means that:

• The encodings in this description are named to match the encodings of INS (general).

• The description of INS (general) gives the operational pseudocode for this instruction.

Encoding

MOV <Vd>.<Ts>[<index>], <R><n>

is equivalent to

INS <Vd>.<Ts>[<index>], <R><n>

and is always the preferred disassembly.

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in the "imm5" field. It can have the following values:

B    when imm5 = xxxx1
H    when imm5 = xxx10
S    when imm5 = xx100
D    when imm5 = x1000

The encoding imm5 = x0000 is reserved.

<index> Is the element index encoded in the "imm5" field. It can have the following values:

imm5<4:1>    when imm5 = xxxx1
imm5<4:2>    when imm5 = xxx10
imm5<4:3>    when imm5 = xx100
imm5<4>      when imm5 = x1000

The encoding imm5 = x0000 is reserved.

<R> Is the width specifier for the general-purpose source register, encoded in the "imm5" field. It can have the following values:

W    when imm5 = xxxx1
W    when imm5 = xxx10
W    when imm5 = xx100
X    when imm5 = x1000
The encoding \( \text{imm5} = \text{x0000} \) is reserved.

\(<n>\) Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

**Operation**

The description of INS (general) gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.202 MOV (vector)

Move vector. This instruction copies the vector in the source SIMD&FP register into the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is an alias of the ORR (vector, register) instruction. This means that:

- The encodings in this description are named to match the encodings of ORR (vector, register).
- The description of ORR (vector, register) gives the operational pseudocode for this instruction.

\[
\begin{array}{c|c|c}
\text{Encoding} & \text{Operation} & \text{Assembler symbols} \\
\end{array}
\]

| MOV <Vd>.<T>, <Vn>.<T> | ORR <Vd>.<T>, <Vn>.<T>, <Vn>.<T> | Rd |

and is the preferred disassembly when \( Rm = Rn \).

**Assembler symbols**

- \(<Vd>\) is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<T>\) is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  - 8B when \( Q = 0 \)
  - 16B when \( Q = 1 \)
- \(<Vn>\) is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**Operation**

The description of ORR (vector, register) gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.203   MOV (to general)

Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is an alias of the UMOV instruction. This means that:

- The encodings in this description are named to match the encodings of UMOV.
- The description of UMOV gives the operational pseudocode for this instruction.

\[
\text{32-bit variant}
\]

Applies when \( Q == 0 \) \&\& \( \text{imm5} == xx100 \).

\[
\text{MOV } <Wd>, <Vn>.S[<index>]
\]

is equivalent to

\[
\text{UMOV } <Wd>, <Vn>.S[<index>]
\]

and is always the preferred disassembly.

\[
\text{64-bit variant}
\]

Applies when \( Q == 1 \) \&\& \( \text{imm5} == x1000 \).

\[
\text{MOV } <Xd>, <Vn>.D[<index>]
\]

is equivalent to

\[
\text{UMOV } <Xd>, <Vn>.D[<index>]
\]

and is always the preferred disassembly.

**Assembler symbols**

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- \(<\text{index}>\) For the 32-bit variant: is the element index encoded in "imm5<4:3>".
  For the 64-bit variant: is the element index encoded in "imm5<4>".

**Operation**

The description of UMOV gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.204   MOVI

Move Immediate (vector). This instruction places an immediate constant into every vector element of the
destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

8-bit variant
Applies when \( \text{op} == 0 \) \&\& \( \text{cmode} == 1110 \).
MOVI \( <Vd>.<T>, \#<imm_8>{, \text{LSL #0}} \)

16-bit shifted immediate variant
Applies when \( \text{op} == 0 \) \&\& \( \text{cmode} == 10x0 \).
MOVI \( <Vd>.<T>, \#<imm_8>{, \text{LSL #<amount>}} \)

32-bit shifted immediate variant
Applies when \( \text{op} == 0 \) \&\& \( \text{cmode} == 0xx0 \).
MOVI \( <Vd>.<T>, \#<imm_8>{, \text{LSL #<amount>}} \)

32-bit shifting ones variant
Applies when \( \text{op} == 0 \) \&\& \( \text{cmode} == 110x \).
MOVI \( <Vd>.<T>, \#<imm_8>, \text{MSL #<amount>} \)

64-bit scalar variant
Applies when \( Q == 0 \) \&\& \( \text{op} == 1 \) \&\& \( \text{cmode} == 1110 \).
MOVI \( <Dd>, \#<imm> \)

64-bit vector variant
Applies when \( Q == 1 \) \&\& \( \text{op} == 1 \) \&\& \( \text{cmode} == 1110 \).
MOVI \( <Vd>.2D, \#<imm> \)

Decode for all variants of this encoding

integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
 when '0xx00' operation = ImmediateOp_MOVI;
 when '0xx01' operation = ImmediateOp_MVNI;
 when '0xx10' operation = ImmediateOp_ORR;
 when '0xx11' operation = ImmediateOp_BIC;
 when '10x00' operation = ImmediateOp_MOVI;
when '10x01' operation = ImmediateOp_MVNI;
when '10x10' operation = ImmediateOp_ORR;
when '10x11' operation = ImmediateOp_BIC;
when '110x0' operation = ImmediateOp_MOVI;
when '110x1' operation = ImmediateOp_MVNI;
when '1110x' operation = ImmediateOp_MOVI;
when '11110' operation = ImmediateOp_MOVI;
when '11111'
  // FMOV Dn,#imm is in main FP instruction set
  if Q == '0' then UNDEFINED;
  operation = ImmediateOp_MOVI;
imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);

Assembler symbols

<d> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<d> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<imm> Is a 64-bit immediate 'aaaaaaaabbbbbbbccccccccddddddeeeeeeefgggggggghhhhhhh', encoded in "a:b:c:d:e:f:g:h".

<T> For the 8-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  8B when Q = 0
  16B when Q = 1
For the 16-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  4H when Q = 0
  8H when Q = 1
For the 32-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  2S when Q = 0
  4S when Q = 1

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit shifted immediate variant: is the shift amount encoded in the "cmode<1>" field. It can have the following values:
  0 when cmode<1> = 0
  8 when cmode<1> = 1
defaulting to 0 if LSL is omitted.
For the 32-bit shifted immediate variant: is the shift amount encoded in the "cmode<2:1>" field. It can have the following values:
  0 when cmode<2:1> = 00
  8 when cmode<2:1> = 01
  16 when cmode<2:1> = 10
  24 when cmode<2:1> = 11
defaulting to 0 if LSL is omitted.
For the 32-bit shifting ones variant: is the shift amount encoded in the "cmode<0>" field. It can have the following values:
  8 when cmode<0> = 0
when \( \text{cmode}<0 \) = 1

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MVNI
        result = \text{NOT}(\text{imm});
    when ImmediateOp_ORR
        operand = V[rd, datasize];
        result = operand \text{OR} \text{imm};
    when ImmediateOp_BIC
        operand = V[rd, datasize];
        result = operand \text{AND} \text{NOT}(\text{imm});

V[rd, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.205  MUL (by element)

Multiply (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler symbols

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

4H  when size = 01, Q = 0
8H  when size = 01, Q = 1
2S  when size = 10, Q = 0
4S  when size = 10, Q = 1

The following encodings are reserved:

• size = 00, Q = x.
• size = 11, Q = x.

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>  Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm  when size = 01
M:Rm  when size = 10
The following encodings are reserved:
- size = 00.
- size = 11.

Restricted to V0-V15 when element size $<T_s>$ is H.

$<T_s>$

Is an element size specifier, encoded in the "size" field. It can have the following values:
- H when size = 01
- S when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

$<\text{index}>$

Is the element index, encoded in the "size:L:H:M" field. It can have the following values:
- H:L:M when size = 01
- H:L when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    Elem[result, e, esize] = product;
V[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.206 MUL (vector)

Multiply (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean poly = (U == '1');

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
8B when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  if poly then
    product = PolynomialMult(element1, element2)<esize-1:0>;
  else
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
  Elem[result, e, esize] = product;
V[d, datasize] = result;

 Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.207  MVN

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is an alias of the NOT instruction. This means that:

- The encodings in this description are named to match the encodings of NOT.
- The description of NOT gives the operational pseudocode for this instruction.

**Encoding**

MVN <Vd>.<T>, <Vn>.<T>

is equivalent to

NOT  <Vd>.<T>, <Vn>.<T>

and is always the preferred disassembly.

**Assembler symbols**

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>  Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- 8B  when Q = 0
- 16B  when Q = 1

<Vn>  Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

The description of NOT gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.208 MVNI

Move inverted Immediate (vector). This instruction places the inverse of an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

16-bit shifted immediate variant
Applies when cmode == 10x0.
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}

32-bit shifted immediate variant
Applies when cmode == 0xx0.
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}

32-bit shifting ones variant
Applies when cmode == 110x.
MVNI <Vd>.<T>, #<imm8>, MSL #<amount>

Decode for all variants of this encoding

integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
when '0xx01' operation = ImmediateOp_MVNI;
when '0xx11' operation = ImmediateOp_BIC;
when '10x01' operation = ImmediateOp_MVNI;
when '10x11' operation = ImmediateOp_BIC;
when '110x1' operation = ImmediateOp_MVNI;
when '1110x' operation = ImmediateOp_MOVI;
when '11111'
   // FMOV Dn,#imm is in main FP instruction set
   if Q == '0' then UNDEFINED;
   operation = ImmediateOp_MOVI;
end;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
For the 16-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- **4H** when \( Q = 0 \)
- **8H** when \( Q = 1 \)

For the 32-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

- **2S** when \( Q = 0 \)
- **4S** when \( Q = 1 \)

Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

For the 16-bit shifted immediate variant: is the shift amount encoded in the "cmode<1>" field. It can have the following values:

- **0** when \( \text{cmode}<1> = 0 \)
- **8** when \( \text{cmode}<1> = 1 \)
  
  defaulting to 0 if LSL is omitted.

For the 32-bit shifted immediate variant: is the shift amount encoded in the "cmode<2:1>" field. It can have the following values:

- **0** when \( \text{cmode}<2:1> = 00 \)
- **8** when \( \text{cmode}<2:1> = 01 \)
- **16** when \( \text{cmode}<2:1> = 10 \)
- **24** when \( \text{cmode}<2:1> = 11 \)
  
  defaulting to 0 if LSL is omitted.

For the 32-bit shifting ones variant: is the shift amount encoded in the "cmode<0>" field. It can have the following values:

- **8** when \( \text{cmode}<0> = 0 \)
- **16** when \( \text{cmode}<0> = 1 \)

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
  when ImmediateOp_MOVI
    result = imm;
  when ImmediateOp_MVNI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd, datasize];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd, datasize];
    result = operand AND NOT(imm);

V[rd, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
— The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.209  NEG (vector)

Negate (vector). This instruction reads each vector element from the source SIMD&FP register, negates each value, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar


table

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1</td>
<td>1 1 1 1 0</td>
<td>size 1 0 0 0 0 0 1 0 1 1 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

NEG <V>d>, <V>n>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');

Vector


table

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>0 1 1 1 0</td>
<td>size 1 0 0 0 0 0 1 0 1 1 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

NEG <Vd>,<T>, <Vn>,<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

D when size = 11

The following encodings are reserved:

• size = 0x.
• size = 10.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 0

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.210 NOT

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MVN. The alias is always the preferred disassembly.

Encoding

\[
\text{NOT } <Vd>.<T>, <Vn>.<T>
\]

Decode for this encoding

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } esize &= 8; \\
\text{integer } databsize &= \text{if } Q = '1' \text{ then } 128 \text{ else } 64; \\
\text{integer } elements &= \text{databsize DIV } 8;
\end{align*}
\]

Assembler symbols

\[
<\text{Vd}> \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.}
\]

\[
<\text{T}> \quad \text{Is an arrangement specifier, encoded in the "Q" field. It can have the following values:}
\]

\[
\begin{align*}
8B & \quad \text{when } Q = 0 \\
16B & \quad \text{when } Q = 1
\end{align*}
\]

\[
<\text{Vn}> \quad \text{Is the name of the SIMD&FP source register, encoded in the "Rn" field.}
\]

Operation

\[
\begin{align*}
\text{CheckFPAdvSIMDEnabled64();} \\
\text{bits(databsize) operand} &= V[n, \text{databsize}]; \\
\text{bits(databsize) result}; \\
\text{bits(esize) element}; \\
\text{for } e = 0 \text{ to elements-1} \\
& \quad \text{element} = \text{Elem}[\text{operand}, e, \text{esize}]; \\
& \quad \text{Elem}[\text{result}, e, \text{esize}] = \text{NOT(element)}; \\
\text{V}[d, \text{databsize}] &= \text{result};
\end{align*}
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
— The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.211 ORN (vector)

Bitwise inclusive OR NOT (vector). This instruction performs a bitwise OR NOT between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

ORN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  - 8B when Q = 0
  - 16B when Q = 1
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
operand2 = NOT(operand2);
result = operand1 OR operand2;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.212  ORR (vector, immediate)

Bitwise inclusive OR (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise OR between each result and an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

16-bit variant

Applies when cmode == 10x1.

ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}

32-bit variant

Applies when cmode == 0xx1.

ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}

Decode for all variants of this encoding

integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx00' operation = ImmediateOp_MOVI;
  when '0xx10' operation = ImmediateOp_ORR;
  when '10x00' operation = ImmediateOp_MOVI;
  when '10x10' operation = ImmediateOp_ORR;
  when '110x0' operation = ImmediateOp_MOVI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '11110' operation = ImmediateOp_MOVI;
imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);

Assembler symbols

<Vd>  Is the name of the SIMD&FP register, encoded in the "Rd" field.
<T>  For the 16-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  4H   when Q = 0
  8H   when Q = 1
For the 32-bit variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:
  2S   when Q = 0
  4S   when Q = 1
<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit variant: is the shift amount encoded in the "cmode<1>" field. It can have the following values:
0  when cmode<1> = 0
8  when cmode<1> = 1
defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in the "cmode<2:1>" field. It can have the following values:
0  when cmode<2:1> = 00
8  when cmode<2:1> = 01
16 when cmode<2:1> = 10
24 when cmode<2:1> = 11
defaulting to 0 if LSL is omitted.

Operation

CheckFPAdvSIMDEnabled64();
bias(datasize) operand;
bias(datasize) result;
case operation of
  when ImmediateOp_MOVI
    result = imm;
  when ImmediateOp_NVMI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd, datasize];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd, datasize];
    result = operand AND NOT(imm);
V[rd, datasize] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.213  ORR (vector, register)

Bitwise inclusive OR (vector, register). This instruction performs a bitwise OR between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (vector). See Alias conditions for details of when each alias is preferred.

Encoding

ORR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (vector)</td>
<td>Rm == Rn</td>
</tr>
</tbody>
</table>

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
8B    when Q = 0
16B   when Q = 1
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
result = operand1 OR operand2;
V[d, datasize] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.214   PMUL

Polynomial Multiply. This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

For information about multiplying polynomials see *Polynomial arithmetic over \{0, 1\}* on page A1-54.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

PMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean poly = (U == '1');

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
8B when size = 00, Q = 0
16B when size = 00, Q = 1
The following encodings are reserved:
* size = 01, Q = x.
* size = 1x, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
for e = 0 to elements-1
element1 = Elem[operand1, e, esize];
Element2 = Elem[operand2, e, esize];
if poly then
    product = PolynomialMult(element1, element2)<esize-1:0>;
else
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
Elem[result, e, esize] = product;

V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.215  PMULL, PMULL2

Polynomial Multiply Long. This instruction multiplies corresponding elements in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials, see Polynomial arithmetic over \{0, 1\} on page A1-54.

The PMULL instruction extracts each source vector from the lower half of each source register. The PMULL2 instruction extracts each source vector from the upper half of each source register.

The PMULL, PMULL2 variants that operate on 64-bit data quantities are defined only when FEAT_PMULL is implemented.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Encoding

PMULL\{2\} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

### Decode for this encoding

```cpp
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '01' || size == '10' then UNDEFINED;
if size == '11' && !HaveBit128PMULLExt() then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

### Assembler symbols

2  

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd>  

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  

Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 1Q when size = 11

The following encodings are reserved:

- size = 01.
- size = 10.

The '1Q' arrangement is only allocated in an implementation that includes the Cryptographic Extension, and is otherwise RESERVED.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 1D when size = 11, Q = 0
- 2D when size = 11, Q = 1

The following encodings are reserved:

- size = 01, Q = x.
- size = 10, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, 2*esize] = PolynomialMult(element1, element2);

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.216 RADDHN, RADDHN2

Rounding Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see ADDHN, ADDHN2.

The RADDHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RADDHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

RADDHN(2) <Vd>..<Tb>, <Vn>..<Ta>, <Vm>..<Ta>

Decode for this encoding

```plaintext
ingen int d = UInt(Rd);
gen int n = UInt(Rn);
gen int m = UInt(Rm);

if size == '11' then UNDEFINED;
gen int esize = 8 << UInt(size);
gen int datasize = 64;
gen int part = UInt(Q);
gen int elements = datasize DIV esize;

bool sub_op = (o1 == '1');
gen bool round = (U == '1');
```

Assembler symbols

- 2: Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
  - [absent] when Q = 0
  - [present] when Q = 1

- <Vd>: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Tb>: Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - 8B when size = 00, Q = 0
  - 16B when size = 00, Q = 1
  - 4H when size = 01, Q = 0
  - 8H when size = 01, Q = 1
  - 2S when size = 10, Q = 0
  - 4S when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.

\( <Vn> \) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\( <Ta> \) Is an arrangement specifier, encoded in the "size" field. It can have the following values:

\( 8H \) when size = 00
\( 4S \) when size = 01
\( 2D \) when size = 10

The encoding size = 11 is reserved.

\( <Vm> \) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(2*datasize) operand2 = V[m, 2*datasize];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;
for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;
Vpart[d, part, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.217   RAX1

Rotate and Exclusive OR rotates each 64-bit element of the 128-bit vector in a source SIMD&FP register left by 1, performs a bitwise exclusive OR of the resulting 128-bit vector and the vector in another source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SHA3 is implemented.

**Advanced SIMD**

(_FEAT_SHA3_

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20]</th>
<th>[16 15 14 13 12 11 10 9 ]</th>
<th>[5 4 ]</th>
<th>[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1 1 0 0 1 1</td>
<td>Rm</td>
<td>1 0 0 1 1</td>
<td>Rd</td>
</tr>
</tbody>
</table>

**Encoding**

RAX1 <Vd>.2D, <Vn>.2D, <Vm>.2D

**Decode for this encoding**

if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

**Assembler symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

AArch64.CheckFPAdvSIMDEnabled();
bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
V[d, 128] = Vn EOR (ROL(Vm<127:64>, 1):ROL(Vm<63:0>, 1));

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.218   RBIT (vector)

Reverse Bit order (vector). This instruction reads each vector element from the source SIMD&FP register, reverses the bits of the element, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding
RBIT <Vd>.<T>, <Vn>.<T>

Decode for this encoding
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;

Assembler symbols
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
     8B when Q = 0
     16B when Q = 1
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;
bits(esize) rev;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    for i = 0 to esize-1
        rev<(esize-1)-i> = element<i>;
        Elem[result, e, esize] = rev;
V[d, datasize] = result;

Operational information
If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
— The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
**C7.2.219 REV16 (vector)**

Reverse elements in 16-bit halfwords (vector). This instruction reverses the order of 8-bit elements in each halfword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Encoding**

REV16 <Vd>.<T>, <Vn>.<T>

**Decode for this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:  B(0),  H(1),  S(1),  D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx:  64(0),  32(1),  16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//    8+B = X,  8+H = X,  8+S = X,  8+D = X
// =>  3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//    8+B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
end_case

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

**Assembler symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - `88` when size = `00`, Q = `0`
  - `168` when size = `00`, Q = `1`
The following encodings are reserved:
- size = 01, Q = x.
- size = 1x, Q = x.

<\n> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
  rev_element = element + elements_per_container - 1;
  for e = 0 to elements_per_container-1
    Elem[result, rev_element, esize] = Elem[operand, element, esize];
    element = element + 1;
    rev_element = rev_element - 1;

V[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.220 REV32 (vector)

Reverse elements in 32-bit words (vector). This instruction reverses the order of 8-bit or 16-bit elements in each word of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

REV32 <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:  B(0),  H(1),  S(1),  D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//    8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32/B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16/B = 1, 16+H = X, 16+S = X, 16+D = X
//    8/B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;
integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  88 when size = 00, Q = 0
  168 when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1

The encoding size = 1x, Q = x is reserved.

<\n>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
  rev_element = element + elements_per_container - 1;
  for e = 0 to elements_per_container-1
    Elem[result, rev_element, esize] = Elem[operand, element, esize];
    element = element + 1;
    rev_element = rev_element - 1;
V[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.221 REV64

Reverse elements in 64-bit doublewords (vector). This instruction reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Encoding

REV64 <Vd>.<T>, <Vn>.<T>

### Decode for this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:  B(0), H(1), S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx:  64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//    8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32/B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16/B = 1, 16+H = X, 16+S = X, 16+D = X
//    8/B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
endcase;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

### Assembler symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - 88 when size = 00, Q = 0
  - 168 when size = 00, Q = 1
4H  when size = 01, Q = 0
8H  when size = 01, Q = 1
2S  when size = 10, Q = 0
4S  when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<\n>  Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
  rev_element = element + elements_per_container - 1;
for e = 0 to elements_per_container-1
  Elem[result, rev_element, esize] = Elem[operand, element, esize];
  element = element + 1;
  rev_element = rev_element - 1;

V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

•  The execution time of this instruction is independent of:
   —  The values of the data supplied in any of its registers.
   —  The values of the NZCV flags.

•  The response of this instruction to asynchronous exceptions does not vary based on:
   —  The values of the data supplied in any of its registers.
   —  The values of the NZCV flags.
C7.2.222  RSHRN, RSHRN2

Rounding Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the vector in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see SHRN, SHRN2.

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Encoding

\[
\begin{array}{cccccccccccccccccc}
0 & Q & 0 & 0 & 1 & 1 & 1 & 0 & !=0000 & immb & 1 & 0 & 0 & 0 & 1 & 1 & Rn & Rd & \hline
\end{array}
\]

**immh**  

**op**

#### Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

#### Assembler symbols

2  

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent]  when Q = 0
- [present]  when Q = 1

\(<Vd>\)  

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Tb>\)  

Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

- 8B  when immh = 0001, Q = 0
- 16B  when immh = 0001, Q = 1
- 4H  when immh = 001x, Q = 0
- 8H  when immh = 001x, Q = 1
- 2S  when immh = 01xx, Q = 0
- 4S  when immh = 01xx, Q = 1

See *Advanced SIMD modified immediate* on page C4-632 when immh = 0000, Q = x.
The encoding \( \text{immh} = 1xxx, Q = x \) is reserved.

\(<Vn>\)
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<Ta>\)
Is an arrangement specifier, encoded in the "immh" field. It can have the following values:
- \( 8H \) when \( \text{immh} = 0001 \)
- \( 4S \) when \( \text{immh} = 001x \)
- \( 2D \) when \( \text{immh} = 01xx \)

See Advanced SIMD modified immediate on page C4-632 when \( \text{immh} = 0000 \).

The encoding \( \text{immh} = 1xxx \) is reserved.

\(<\text{shift}>\)
Is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:
- \( (16-\text{UInt} (\text{immh:immb})) \) when \( \text{immh} = 0001 \)
- \( (32-\text{UInt} (\text{immh:immb})) \) when \( \text{immh} = 001x \)
- \( (64-\text{UInt} (\text{immh:immb})) \) when \( \text{immh} = 01xx \)

See Advanced SIMD modified immediate on page C4-632 when \( \text{immh} = 0000 \).

The encoding \( \text{immh} = 1xxx \) is reserved.

**Operation**

CheckFPAdvSIMDEnabled64();

bits(datasize*2) operand = V[n, datasize*2];

bits(datasize) result;

integer round_const = if round then \((1 << (\text{shift} - 1))\) else 0;

integer element;

for e = 0 to elements - 1
   element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> \text{shift};
   Elem[result, e, esize] = element<esize - 1:0>;

Vpart[d, part, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RSUBHN, RSUBHN2

Rounding Subtract returning High Narrow. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see SUBHN, SUBHN2.

The RSUBHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSUBHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

RSUBHN(2) <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0
[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Ta>\) Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- \(8\) when size = 00
- \(4\) when size = 01
- \(2\) when size = 10

The encoding size = 11 is reserved.

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(2*datasize) operand2 = V[m, 2*datasize];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;
for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;
Vpart[d, part, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.224   SABA

Signed Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
[31 30 29 28][27 26 25 24][23 22 21 20]  16|15 14 13 12|11 10  9 |  5  4 |  0 |
  0 Q 0 0 1 1 1 0 |  size 1 | Rm  0 1 1 1 1 1 | Rn  | Rd |
  U ac
```

**Encoding**

SABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');

**Assembler symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;
result = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.225  SABAL, SABAL2

Signed Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABAL instruction extracts each source vector from the lower half of each source register. The SABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

SABAL[2] <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');

Assembler symbols

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

-absent- when Q = 0
-present- when Q = 1

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  Is an arrangement specifier, encoded in the "size" field. It can have the following values:

8H when size = 00
4S when size = 01
2D when size = 10

The encoding size = 11 is reserved.

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

88 when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.

<\m>

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d, 2*datasize] else Zeros(2 * datasize);
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  absdiff = Abs(element1-element2)<2*esize-1:0>;
  Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
  V[d, 2*datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.226   SABD

Signed Absolute Difference. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding
SABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  8B when size == 00, Q = 0
  16B when size == 00, Q = 1
  4H when size == 01, Q = 0
  8H when size == 01, Q = 1
  2S when size == 10, Q = 0
  4S when size == 10, Q = 1
The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;
result = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  absdiff = Abs(element1-element2)≤esize-1:0>
  Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d, datasize] = result;

Operational information
If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.227 SABDL, SABDL2

Signed Absolute Difference Long. This instruction subtracts the vector elements of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the results into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABDL instruction extracts each source vector from the lower half of each source register, while the SABDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

SABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0
[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

8H when size = 00
45 when size = 01
2D when size = 10

The encoding size = 11 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

88 when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<\m> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d, 2*datasize] else Zeros(2 * datasize);
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  absdiff = Abs(element1-element2)<<esize-1:0;
  Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d, 2*datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.228 SADALP

Signed Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register and accumulates the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

SADALP <Vd>.<Ta>, <Vn>.<Tb>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
4H when size = 00, Q = 0
8H when size = 00, Q = 1
2S when size = 01, Q = 0
4S when size = 01, Q = 1
1D when size = 10, Q = 0
2D when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
8B when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.
Operation

\[
\text{CheckFPAdvSIMDEnabled64();}
\]
\[
\text{bits(datasize) operand = V[n, datasize];}
\]
\[
\text{bits(datasize) result;}
\]
\[
\text{bits(2*esize) sum;}
\]
\[
\text{integer op1;}
\]
\[
\text{integer op2;}
\]
\[
\text{if acc then result = V[d, datasize];}
\]
\[
\text{for e = 0 to elements-1}
\]
\[
\text{op1 = Int(Elem[operand, 2*e+0, esize], unsigned);}
\]
\[
\text{op2 = Int(Elem[operand, 2*e+1, esize], unsigned);}
\]
\[
\text{sum = (op1+op2)+2*esize-1:0;}
\]
\[
\text{if acc then}
\]
\[
\text{Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;}
\]
\[
\text{else}
\]
\[
\text{Elem[result, e, 2*esize] = sum;}
\]
\[
\text{V[d, datasize] = result;}
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.229 SADDL, SADDL2

Signed Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SADDL instruction extracts each source vector from the lower half of each source register. The SADDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTER_EL2, and CPTER_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
</tr>
</tbody>
</table>

U

o1

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

Assembler symbols

\[2\]
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0
[present] when Q = 1

\langle Vd \rangle
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\langle Ta \rangle
Is an arrangement specifier, encoded in the "size" field. It can have the following values:

\[8H\] when size = 00
\[4S\] when size = 01
\[2D\] when size = 10

The encoding \[size = 11\] is reserved.

\langle Vn \rangle
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\langle Tb \rangle
Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

\[88\] when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<\m> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.230  **SADDLP**

Signed Add Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The instruction is encoded as follows:

```
+---------------------------------+------------+-----------------+--------------------+
| 31 30 29 28 27 26 25 24 23 22 21 20 | 19 18 17 16 | 15 14 13 12 11 10 9 |
+---------------------------------+------------+-----------------+--------------------+
|       0 0 1 1 1 0               | 1 0 0 0 0 0 | 0 0 0 1 0 1 0    |
+---------------------------------+------------+-----------------+--------------------+
|        U                         |            |                |
+---------------------------------+            +-----------------+--------------------+
|        Rn                        |            |                |
+---------------------------------+            +-----------------+--------------------+
|        Rd                        |            |                |
+---------------------------------+            +-----------------+--------------------+
```

**Encoding**

SADDLP \(<Vd>.<Ta>, <Vn>.<Tb>\)

**Decode for this encoding**

```python
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

**Assembler symbols**

- **<Vd>**  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Ta>**  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - 4H  when size = 00, Q = 0
  - 8H  when size = 00, Q = 1
  - 2S  when size = 01, Q = 0
  - 4S  when size = 01, Q = 1
  - 1D  when size = 10, Q = 0
  - 2D  when size = 10, Q = 1
  The encoding size = 11, Q = x is reserved.
- **<Vn>**  Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- **<Tb>**  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - 8B  when size = 00, Q = 0
  - 16B when size = 00, Q = 1
  - 4H  when size = 01, Q = 0
  - 8H  when size = 01, Q = 1
  - 2S  when size = 10, Q = 0
  - 4S  when size = 10, Q = 1
  The encoding size = 11, Q = x is reserved.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d, datasize];
for e = 0 to elements-1
  op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
  op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
  sum = (op1+op2)<2*esize-1:0>;
  if acc then
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
  else
    Elem[result, e, 2*esize] = sum;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.231 SADDLV

Signed Add Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

SADDLV <V><d>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:
H when size = 00
S when size = 01
D when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
8B when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
4S when size = 10, Q = 1

The following encodings are reserved:
- size = 10, Q = 0.
- size = 11, Q = x.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
   sum = sum + Int(Elem[operand, e, esize], unsigned);
V[d, 2*esize] = sum<2*esize-1:0>;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.232 SADDW, SADDW2

Signed Add Wide. This instruction adds vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the results in a vector, and writes the vector to the SIMD&FP destination register.

The SADDW instruction extracts the second source vector from the lower half of the second source register. The SADDW2 instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Encoding**

SADDW<2> <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

**Assembler symbols**

- 2: Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
  - [absent] when Q = 0
  - [present] when Q = 1

- <Vd>: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Ta>: Is an arrangement specifier, encoded in the "size" field. It can have the following values:
  - 8H when size = 00
  - 4S when size = 01
  - 2D when size = 10

  The encoding size = 11 is reserved.

- <Vn>: Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

- <Vm>: Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, 2*esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  if sub_op then
    sum = element1 - element2;
  else
    sum = element1 + element2;
  Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d, 2*datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.233 SCVTF (vector, fixed-point)

Signed fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

**Scalar**

```
| 31 30 29 28|27 26 25 24|23 22 | 19 18 | 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|-------------|-------------|-------------|-------------|-------------|
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | !=0000 | immh | 1 | 1 | 1 | 0 | 0 | 1 | Rn | Rd |
```

**Encoding**

SCVTF <V><d>, <V><n>, #<fbits>

**Decode for this encoding**

- integer d = UInt(Rd);
- integer n = UInt(Rn);

if immh IN ('000x') || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer dataszize = esize;
integer elements = 1;

integer fracbits = (esize + 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);

**Vector**

```
| 31 30 29 28|27 26 25 24|23 22 | 19 18 | 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|-------------|-------------|-------------|-------------|-------------|
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | !=0000 | immh | 1 | 1 | 1 | 0 | 0 | 1 | Rn | Rd |
```

**Encoding**

SCVTF <Vd>.<T>, <Vn>.<T>, #<fbits>

**Decode for this encoding**

- integer d = UInt(Rd);
- integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh IN ('000x') || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
if immh>Q == '10' then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if imm IN {'01xx'} then 32 else 16;
integer dataszize = if Q == '1' then 128 else 64;
integer elements = dataszize DIV esize;

integer fracbits = (esize + 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);

**Assembler symbols**

<\V> Is a width specifier, encoded in the "immh" field. It can have the following values:

- **H** when immh = 001x
- **S** when immh = 01xx
- **D** when immh = 1xxx

The encoding immh = 000x is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

</d> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

</T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

- **4H** when immh = 001x, Q = 0
- **8H** when immh = 001x, Q = 1
- **2S** when immh = 01xx, Q = 0
- **4S** when immh = 01xx, Q = 1
- **2D** when immh = 1xxx, Q = 1

See *Advanced SIMD modified immediate* on page C4-632 when immh = 0000, Q = x.

The following encodings are reserved:

- immh = 0001, Q = x.
- immh = 1xxx, Q = 0.

</Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in the "immh:immb" field. It can have the following values:

- (32-Uint(immh:immb)) when immh = 001x
- (64-Uint(immh:immb)) when immh = 01xx
- (128-Uint(immh:immb)) when immh = 1xxx

The encoding immh = 000x is reserved.

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in the "immh:immb" field. It can have the following values:

- (32-Uint(immh:immb)) when immh = 001x
- (64-Uint(immh:immb)) when immh = 01xx
- (128-Uint(immh:immb)) when immh = 1xxx

See *Advanced SIMD modified immediate* on page C4-632 when immh = 0000.

The encoding immh = 0001 is reserved.

**Operation for all encodings**

CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, fpcr, rounding, esize);
V[d, 128] = result;
C7.2.234  **SCVTF (vector, integer)**

Signed integer Convert to Floating-point (vector). This instruction converts each element in a vector from signed integer to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps* on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

**Scalar half precision**

(Feat_Fp16)

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 0|1 1 1 1 0 0|1 1 1 0 1 1 0|1 0 Rn | Rd
```

*Encoding*

SCVTF <Hd>, <Hn>

*Decode for this encoding*

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

**Scalar single-precision and double-precision**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 0|1 1 1 1 0 0|sz 1 0 0 0 0 1 1 1 0 1 1 0|1 0 Rn | Rd
```

*Encoding*

SCVTF <V>d>, <V>n>

*Decode for this encoding*

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```
Vector half precision

(Feat_FP16)

\[
\begin{array}{cccccccccccccccccc}
\hline
0 & Q & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & Rn & Rd \\
\end{array}
\]

Encoding

SCVT <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Vector single-precision and double-precision

\[
\begin{array}{cccccccccccccccccc}
\hline
0 & Q & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & Rn & Rd \\
\end{array}
\]

Encoding

SCVT <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0
8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

FPCRTypelfcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

FPRounding rounding = FPRoundingMode(fpcr);
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
C7.2.235  SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (scalar). This instruction converts the signed value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

32-bit to half-precision variant
Applies when \( sf = 0 \) && \( ftype = 11 \).

SCVTF <Hd>, <Wn>, #<fbits>

32-bit to single-precision variant
Applies when \( sf = 0 \) && \( ftype = 00 \).

SCVTF <Sd>, <Wn>, #<fbits>

32-bit to double-precision variant
Applies when \( sf = 0 \) && \( ftype = 01 \).

SCVTF <Dd>, <Wn>, #<fbits>

64-bit to half-precision variant
Applies when \( sf = 1 \) && \( ftype = 11 \).

SCVTF <Hd>, <Xn>, #<fbits>

64-bit to single-precision variant
Applies when \( sf = 1 \) && \( ftype = 00 \).

SCVTF <Sd>, <Xn>, #<fbits>

64-bit to double-precision variant
Applies when \( sf = 1 \) && \( ftype = 01 \).

SCVTF <Dd>, <Xn>, #<fbits>

Decode for all variants of this encoding

\[
\begin{align*}
d &= \text{UInt}(Rd); \\
n &= \text{UInt}(Rn); \\
\text{intsize} &= \text{if } sf = '1' \text{ then } 64 \text{ else } 32; \\
f\text{ltsize} &= \text{intsize}; \\
r\text{ounding} &= \text{PPRounding}(\text{rounding});
\end{align*}
\]
case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  if sf == '0' && scale<5> == '0' then UNDEFINED;
  integer fracbits = 64 - UInt(scale);
  rounding = FPRoundingMode(FPCR[]);

Assembler symbols

<DD> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<SDD> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<SD> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<XN> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<fbits> For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".
  For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

Operation

CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
basisbits(fsize) fltval;
basisbits(intsize) intval;
intval = X[n, intsize];
fltval = if merge then V[d, fsize] else Zeros(fsize);
Elem[fltval, 0, fltsize] = FixedToFP(intval, fracbits, FALSE, fpcr, rounding, fltsize);
V[d, fsize] = fltval;
C7.2.236   SCVTF (scalar, integer)

Signed integer Convert to Floating-point (scalar). This instruction converts the signed integer value in the
general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and
writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception
results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see
Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

32-bit to half-precision variant
Applies when sf == 0 && ftype == 11.
SCVTFRd, Rn

32-bit to single-precision variant
Applies when sf == 0 && ftype == 00.
SCVTFRd, Rn

32-bit to double-precision variant
Applies when sf == 0 && ftype == 01.
SCVTFRd, Rn

64-bit to half-precision variant
Applies when sf == 1 && ftype == 11.
SCVTFRd, Rn

64-bit to single-precision variant
Applies when sf == 1 && ftype == 00.
SCVTFRd, Rn

64-bit to double-precision variant
Applies when sf == 1 && ftype == 01.
SCVTFRd, Rn

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

FPRounding rounding;

sf 0 0 1 1 1 0 ftype 1 0 0 0 1 0 0 0 0 0 0 0 | Rn | Rd
mode opcode

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 16 15 14 13 12 | 11 10 9 | 5 4 | 0 |
case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  rounding = FPRoundingMode(FPCR[]);

Assembler symbols

<Db> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

CheckFPEnabled64();

FPRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

intval = X[n, intsize];
fltval = if merge then V[d, fsize] else Zeros(fsize);
Elem[fltval, 0, fsize] = FixedToFP(intval, 0, FALSE, fpcr, rounding, fltsize);
V[d, fsize] = fltval;
C7.2.237  SDOT (by element)

Dot Product signed arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note  
ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector

(FEAT_DotProc)

<table>
<thead>
<tr>
<th>Vd</th>
<th>Tn</th>
<th>Tb</th>
<th>M</th>
<th>L</th>
<th>Rm</th>
<th>H</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
</tr>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>size</td>
<td>L</td>
</tr>
</tbody>
</table>

Encoding

SDOT <Vd>,<Ta>, <Vm>,<Tb>, <Vm>,4B[<index>]

Decode for this encoding

if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index = UInt(H:L);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler symbols

<Vd>  Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Ta>  Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

2S when Q = 0
4S when Q = 1

<Vm>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>  Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

8B when Q = 0
16B when Q = 1

<Vm>  Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
Is the element index, encoded in the "H:L" fields.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(128) operand2 = V[m, 128];
bits(datasize) result = V[d, datasize];
for e = 0 to elements-1
  integer res = 0;
  integer element1, element2;
  for i = 0 to 3
    if signed then
      element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
      element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
    else
      element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
      element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
    res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d, datasize] = result;
```
C7.2.238   SDOT (vector)

Dot Product signed arithmetic (vector). This instruction performs the dot product of the four signed 8-bit elements in each 32-bit element of the first source register with the four signed 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an optional instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector

(FEAT_DotProd)

<table>
<thead>
<tr>
<th></th>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{U}</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

Decode for this encoding

\[
\text{if !HaveDOTPExt()} \text{ then UNDEFINED;} \\
\text{if } \text{size} \neq '10' \text{ then UNDEFINED;} \\
\text{boolean signed = (U == '0');} \\
\text{integer d = UInt(Rd);} \\
\text{integer n = UInt(Rn);} \\
\text{integer m = UInt(Rm);} \\
\text{integer esize = 8 \ll UInt(size);} \\
\text{integer datasize = if } Q = '1' \text{ then 128 else 64;} \\
\text{integer elements = datasize} \div \text{esize;}
\]

Assembler symbols

\(<\text{Vd}>\): Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

\(<\text{Ta}>\): Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

\begin{align*}
2S & \quad \text{when } Q = 0 \\
4S & \quad \text{when } Q = 1
\end{align*}

\(<\text{Vn}>\): Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<\text{Tb}>\): Is an arrangement specifier, encoded in the "Q" field. It can have the following values:

\begin{align*}
8B & \quad \text{when } Q = 0 \\
16B & \quad \text{when } Q = 1
\end{align*}

\(<\text{Vm}>\): Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

result = V[d, datasize];

for e = 0 to elements-1
integer res = 0;
integer element1, element2;
for i = 0 to 3
if signed then
    element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
    element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
else
    element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
    element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
res = res + element1 * element2;
Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d, datasize] = result;
C7.2.239   SHA1C

SHA1 hash update (choose).

Advanced SIMD

(_FEAT_SHA1)

Encoding

SHA1C <Qd>, <Sn>, <Vm>.4S

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;

Assembler symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) x = V[d, 128];
bits(32) y = V[n, 32];    // Note: 32 not 128 bits wide
bits(128) w = V[m, 128];
bits(32) t;
for e = 0 to 3
  t = SHAchoose(x<63:32>, x<95:64>, x<127:96>);
  y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
  x<63:32> = ROL(x<63:32>, 30);
  <y, x> = ROL(y:x, 32);
V[d, 128] = x;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.240 SHA1H

SHA1 fixed rotate.

Advanced SIMD

(FEAT_SHA1)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0 0 0 1 0 0 0 0 0 0 0 1 0</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Encoding

SHA1H <Sd>, <Sn>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA1Ext() then UNDEFINED;

Assembler symbols

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(32) operand = V[n, 32]; // read element [0] only, [1-3] zeroed
V[d, 32] = ROL(operand, 30);

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.241 SHA1M

SHA1 hash update (majority).

**Advanced SIMD**

(FEAT_SHA1)

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 0 0 0 0</td>
<td>Rm 0 0 1 0 0 0</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>
```

**Encoding**

SHA1M <Qd>, <Sn>, <Vm>.4S

**Decode for this encoding**

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

**Assembler symbols**

- `<Qd>` is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- `<Sn>` is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` is the name of the third SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```java
AArch64.CheckFPAdvSIMDEnabled();
```

```java
bits(128) x = V[d, 128];
bits(32) y = V[n, 32];    // Note: 32 not 128 bits wide
bits(128) w = V[m, 128];
bits(32) t;
for e = 0 to 3
  t = SHAmajority(x<63:32>, x<95:64>, x<127:96>);
  y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
  x<63:32> = ROL(x<63:32>, 30);
  <y, x> = ROL(y<x, 32);
V[d, 128] = x;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.242 SHA1P

SHA1 hash update (parity).

**Advanced SIMD**

(FEAT_SHA1)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 0 0 0</td>
<td>Rm</td>
<td>0 0 0 1 0 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

SHA1P <Qd>, <Sn>, <Vm>.4S

**Decode for this encoding**

```cpp
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

**Assembler symbols**

- `<Qd>` is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- `<Sn>` is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` is the name of the third SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```cpp
AArch64.CheckFPAdvSIMDEnabled();

bits(128) x = V[d, 128];
bits(32) y = V[n, 32];   // Note: 32 not 128 bits wide
bits(128) w = V[m, 128];
bits(32) t;
for e = 0 to 3
    t = SHAparity(x<63:32>, x<95:64>, x<127:96>);
    y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
    x<63:32> = ROL(x<63:32>, 30);
    y, x = ROL(y:x, 32);
V[d, 128] = x;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.243   SHA1SU0

SHA1 schedule update 0.

Advanced SIMD

(FEAT_SHA1)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>8 7 6 5</th>
<th>4 3 2 1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 0 0 0</td>
<td>0 0 1 1 0 0</td>
<td>Rd</td>
<td>Rm</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SHA1SU0 <Vd>.4S, <Vn>.4S, <Vm>.4S

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;

Assembler symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) operand3 = V[m, 128];
bits(128) result;

result = operand2<63:0>:operand1<127:64>;
result = result EOR operand1 EOR operand3;
V[d, 128] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.244 SHA1SU1

SHA1 schedule update 1.

**Advanced SIMD**

(FEAT_SHA1)

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 5 4 | 0 ]
0 1 0 1 1 1 1 0 0 0 1 0 0 0 0 0 0 0 1 1 0
```

**Encoding**

SHA1SU1 <Vd>.4S, <Vn>.4S

**Decode for this encoding**

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
if (!HaveSHA1Ext()) then UNDEFINED;
```

**Assembler symbols**

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) result;
bits(128) T = operand1 EOR LSR(operand2, 32);
result<31:0> = ROL(T<31:0>, 1);
result<63:32> = ROL(T<63:32>, 1);
result<95:64> = ROL(T<95:64>, 1);
result<127:96> = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);
V[d, 128] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
## C7.2.245 SHA256H2

SHA256 hash update (part 2).

### Advanced SIMD

(FEAT_SHA256)

```
0 1 0 1 1 1 0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0
```

### Encoding

SHA256H2 <Qd>, <Qn>, <Vm>.45

### Decode for this encoding

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- if !HaveSHA256Ext() then UNDEFINED;

### Assembler symbols

- <Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- <Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) result;
result = SHA256hash(V[n, 128], V[d, 128], V[m, 128], FALSE);
V[d, 128] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.246 SHA256H

SHA256 hash update (part 1).

**Advanced SIMD**

(FEAT_SHA256)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 0 0 0 0</td>
<td>Rm 0 1 0 0 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

SHA256H <Qd>, <Qn>, <Vm>.4S

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;

**Assembler symbols**

<Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.

<Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

**Operation**

AArch64.CheckFPAdvSIMDEnabled();

bits(128) result;
result = SHA256hash(V[d, 128], V[n, 128], V[m, 128], TRUE);
V[d, 128] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.247   SHA256SU0

SHA256 schedule update 0.

Advanced SIMD

(FEAT_SHA256)

\[
\begin{array}{ccccccccccccc|ccc}
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & & & & \\
\end{array}
\]

Encoding

SHA256SU0 \langle Vd \rangle.4S, \langle Vn \rangle.4S

Decode for this encoding

\[
\begin{aligned}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{if !HaveSHA256Ext()} \text{ then UNDEFINED;}
\end{aligned}
\]

Assembler symbols

\langle Vd \rangle        \text{ Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.}

\langle Vn \rangle        \text{ Is the name of the second SIMD&FP source register, encoded in the "Rn" field.}

Operation

\[
\begin{aligned}
\text{AArch64.CheckFPAdvSIMDEnabled();} \\
\text{bits(128) operand1 = V[d, 128];} \\
\text{bits(128) operand2 = V[n, 128];} \\
\text{bits(128) result;} \\
\text{bits(128) T = operand2<31:0>:operand1<127:32>;} \\
\text{bits(32) elt;} \\
\text{for } e = 0 \text{ to } 3 \\
\text{ \quad elt = Elem[T, e, 32];} \\
\text{ \quad elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);} \\
\text{ \quad Elem[result, e, 32] = elt + Elem[operand1, e, 32];} \\
\text{V[d, 128] = result;}
\end{aligned}
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.248 SHA256SU1

SHA256 schedule update 1.

Advanced SIMD

(_FEAT_SHA256)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 0 0 0</td>
<td>Rm 0 1 1 0 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SHA256SU1 <Vd>.4S, <Vn>.4S, <Vm>.4S

Decode for this encoding

int d = UInt(Rd);
int n = UInt(Rn);
int m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;

Assembler symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

u64 operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) operand3 = V[m, 128];
bits(128) result;

bits(64) T0 = operand3<31:0>:operand2<127:32>;
bits(64) T1;
bits(32) elt;

T1 = operand3<127:64>;
for e = 0 to 1
elt = Elem[T1, e, 32];
elt = ROR(elt, 17) XOR ROR(elt, 19) XOR LSR(elt, 10);
elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
Elem[result, e, 32] = elt;

T1 = result<63:0>;
for e = 2 to 3
elt = Elem[T1, e-2, 32];
elt = ROR(elt, 17) XOR ROR(elt, 19) XOR LSR(elt, 10);
elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
Elem[result, e, 32] = elt;

V[d, 128] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.249 SHA512H

SHA512 Hash update part 1 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma1 and chi functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SHA512 is implemented.

Advanced SIMD

(FEAT_SHA512)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0</td>
<td>1 1 1 1</td>
<td>0 1 0 1</td>
<td>1 0</td>
<td>0 0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SHA512H <Qd>, <Qn>, <Vm>.2D

Decode for this encoding

if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.

<Qm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAvSIMDEnabled();

bits(128) Vtmp;
bits(64) MSigma1;
bits(64) tmp;
bits(128) x = V[n, 128];
bits(128) y = V[m, 128];
bits(128) w = V[d, 128];

MSigma1 = ROR(y<127:64>, 14) EOR ROR(y<127:64>, 18) EOR ROR(y<127:64>, 41);
Vtmp<127:64> = (y<127:64> AND x<63:0>) EOR (NOT(y<127:64>) AND x<127:64>);
Vtmp<127:64> = (Vtmp<127:64> + MSigma1 + w<127:64>);
tmp = Vtmp<127:64> + y<63:0>;
MSigma1 = ROR(tmp, 14) EOR ROR(tmp, 18) EOR ROR(tmp, 41);
Vtmp<63:0> = (tmp AND y<127:64>) EOR (NOT(tmp) AND x<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + MSigma1 + w<63:0>);
V[d, 128] = Vtmp;
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.250 SHA512H2

SHA512 Hash update part 2 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma0 and majority functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SHA512 is implemented.

**Advanced SIMD**

(FeaT_SHA512)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 1 0 0 1 1</td>
<td>Rm</td>
<td>1 0</td>
<td>0 0</td>
<td>0 1</td>
</tr>
</tbody>
</table>

**Encoding**

SHA512H2 <Qd>, <Qn>, <Vm>.2D

**Decode for this encoding**

if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

**Assembler symbols**

<Qd> Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(64) NSigma0;
bits(128) x = V[n, 128];
bits(128) y = V[m, 128];
bits(128) w = V[d, 128];

NSigma0 = ROR(y<63:0>, 28) EOR ROR(y<63:0>, 34) EOR ROR(y<63:0>, 39);
Vtmp<127:64> = (x<63:0> AND y<127:64>) EOR (x<63:0> AND y<63:0>) EOR (y<127:64> AND y<63:0>);
Vtmp<127:64> = (Vtmp<127:64> + NSigma0 + w<127:64>);
NSigma0 = ROR(Vtmp<127:64>, 28) EOR ROR(Vtmp<127:64>, 34) EOR ROR(Vtmp<127:64>, 39);
Vtmp<63:0> = (Vtmp<127:64> AND y<63:0>) EOR (Vtmp<127:64> AND y<127:64>) EOR (y<127:64> AND y<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + NSigma0 + w<63:0>);

V[d, 128] = Vtmp;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
— The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.251 SHA512SU0

SHA512 Schedule Update 0 takes the values from the two 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the gamma0 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SHA512 is implemented.

**Advanced SIMD**

(FEAT_SHA512)

\[
\text{SHA512SU0} <V_d>.2D, <V_n>.2D
\]

**Encoding**

SHA512SU0 <V_d>.2D, <V_n>.2D

**Decode for this encoding**

\[
\text{if !HaveSHA512Ext() then UNDEFINED;}
\]

\[
\text{integer } d = \text{UInt}(Rd);
\]

\[
\text{integer } n = \text{UInt}(Rn);
\]

**Assembler symbols**

- `<V_d>` is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- `<V_n>` is the name of the second SIMD&FP source register, encoded in the "Rn" field.

**Operation**

\[
\text{AArch64.CheckFPAdvSIMDEnabled();}
\]

\[
\text{bits(64) } \text{sig0;}
\]

\[
\text{bits(128) Vtmp;}
\]

\[
\text{bits(128) } x = V[n, 128];
\]

\[
\text{bits(128) } w = V[d, 128];
\]

\[
\text{sig0} = \text{ROR}(w<127:64>, 1) \text{ EOR } \text{ROR}(w<127:64>, 8) \text{ EOR } ('0000000':w<127:71>);
\]

\[
\text{Vtmp}<63:0> = w<63:0> + \text{sig0};
\]

\[
\text{sig0} = \text{ROR}(x<63:0>, 1) \text{ EOR } \text{ROR}(x<63:0>, 8) \text{ EOR } ('0000000':x<63:7>);
\]

\[
\text{Vtmp}<127:64> = w<127:64> + \text{sig0};
\]

\[
V[d, 128] = \text{Vtmp;}
\]

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.252 SHA512SU1

SHA512 Schedule Update 1 takes the values from the three source SIMD&FP registers and produces a 128-bit output value that combines the gamma1 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SHA512 is implemented.

Advanced SIMD

(FEAT_SHA512)

| [31 30 29 28 | 27 26 25 24 | 23 22 21 20] | 16|15|14|13|12|11| 9 | 5 | 4 | 0 |
|-----------------|-----------------|-----------------|
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | Rm | 1 | 0 | 0 | 0 | 1 | 0 | Rn | Rd |

Encoding

SHA512SU1 <Vd>.2D, <Vn>.2D, <Vm>.2D

Decode for this encoding

if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler symbols

<Vd> is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig1;
bits(128) Vtmp;
bits(128) x = V[n, 128];
bits(128) y = V[m, 128];
bits(128) w = V[d, 128];

sig1 = ROR(x<127:64>, 19) EOR ROR(x<127:64>, 61) EOR ('000000':x<127:70>);
Vtmp<127:64> = w<127:64> + sig1 + y<127:64>;
sig1 = ROR(x<63:0>, 19) EOR ROR(x<63:0>, 61) EOR ('000000':x<63:6>);
Vtmp<63:0> = w<63:0> + sig1 + y<63:0>;
V[d, 128] = Vtmp;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.253 SHADD

Signed Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see SRHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

SHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  8B when size = 00, Q = 0
  16B when size = 00, Q = 1
  4H when size = 01, Q = 0
  8H when size = 01, Q = 1
  2S when size = 10, Q = 0
  4S when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
b bits(datasize) operand1 = V[n, datasize];
b bits(datasize) operand2 = V[m, datasize];
b bits(datasize) result;
integer element1;
integer element2;
integer sum;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = (element1 + element2) >> 1;
    Elem[result, e, esize] = sum<esize-1:0>;

V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
### C7.2.254 SHL

Shift Left (immediate). This instruction reads each value from a vector, left shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

```
[31 30 29 28 27 26 25 24 23 22 19 18 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0]
[0 1 0 1 1 1 1 0 !=0000 immh 0 1 0 1 0 1 Rn Rd]
```

**Encoding**

SHL <V><d>, <V><n>, #<shift>

**Decode for this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;
```

**Vector**

```
[31 30 29 28 27 26 25 24 23 22 19 18 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0]
[0 Q 0 0 1 1 1 1 0 !=0000 immh 0 1 0 1 0 1 Rn Rd]
```

**Encoding**

SHL <Vd>.<T>, <Vn>.<T>, #<shift>

**Decode for this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
```

**Assembler symbols**

<V> Is a width specifier, encoded in the "immh" field. It can have the following values:

D when immh = lxxx
The encoding $\text{immh} = 0xxx$ is reserved.

$d$ Is the number of the SIMD&FP destination register, in the "Rd" field.

$n$ Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

$Vd$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$T$ Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

- $8B$ when $\text{immh} = 0001$, $Q = 0$
- $16B$ when $\text{immh} = 0001$, $Q = 1$
- $4H$ when $\text{immh} = 001x$, $Q = 0$
- $8H$ when $\text{immh} = 001x$, $Q = 1$
- $2S$ when $\text{immh} = 01xx$, $Q = 0$
- $4S$ when $\text{immh} = 01xx$, $Q = 1$
- $2D$ when $\text{immh} = 1xxx$, $Q = 1$

See Advanced SIMD modified immediate on page C4-632 when $\text{immh} = 0000$, $Q = x$. The encoding $\text{immh} = 1xxx$, $Q = 0$ is reserved.

$Vn$ Is the name of the SIMD&FP source register, encoded in the "Rn" field.

$\text{shift}$ For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in the "immh:immb" field. It can have the following values:

- $(\text{UInt}(\text{immh:immb})-64)$ when $\text{immh} = 1xxx$

The encoding $\text{immh} = 0xxx$ is reserved.

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

- $(\text{UInt}(\text{immh:immb})-8)$ when $\text{immh} = 0001$
- $(\text{UInt}(\text{immh:immb})-16)$ when $\text{immh} = 001x$
- $(\text{UInt}(\text{immh:immb})-32)$ when $\text{immh} = 01xx$
- $(\text{UInt}(\text{immh:immb})-64)$ when $\text{immh} = 1xxx$

See Advanced SIMD modified immediate on page C4-632 when $\text{immh} = 0000$.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bias(datasize) result;

for $e = 0$ to $\text{elements}-1$
   \[\text{Elem}[result, e, esize] = \text{LSL}(\text{Elem}[operand, e, esize], \text{shift});\]

\[V[d, datasize] = result;\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.
C7.2.255   SHLL, SHLL2

Shift Left Long (by element size). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, left shifts each result by the element size, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SHLL instruction extracts vector elements from the lower half of the source register. The SHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

SHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = esize;
boolean unsigned = FALSE;   // Or TRUE without change of functionality

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
[absent] when Q = 0
[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:
8H when size = 00
4S when size = 01
2D when size = 10

The encoding size = 11 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
8B when size = 00, Q = 0
16B when size = 00, Q = 1
4H  when size = 01, Q = 0
8H  when size = 01, Q = 1
2S  when size = 10, Q = 0
4S  when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<shift>  Is the left shift amount, which must be equal to the source element width in bits, encoded in the "size" field. It can have the following values:
8     when size = 00
16    when size = 01
32    when size = 10

The encoding size = 11 is reserved.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part, datasize];
bits(2*datasize) result;
integer element;
for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d, 2*datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.256    SHRN, SHRN2

Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are truncated. For rounded results, see RSHRN, RSHRN2.

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

\[
\begin{array}{cccccccccccccccccccc}
\text{SHRN}\{2\} & & \text{<Vd>}, & \text{<Tb>}, & \text{<Vn>}, & \text{<Ta>}, & \text{#<shift>} \\
\end{array}
\]

Decode for this encoding

\[
\begin{align*}
\text{integer } d & = \text{UInt}(Rd); \\
\text{integer } n & = \text{UInt}(Rn); \\
\text{if } \text{immh} == '0000' \text{ then } & \text{SEE "Advanced SIMD modified immediate";} \\
\text{if } \text{immh<3>} == '1' \text{ then } & \text{UNDEFINED;} \\
\text{integer } \text{esize} & = 8 \times \text{HighestSetBit}(\text{immh}); \\
\text{integer } \text{datasize} & = 64; \\
\text{integer } \text{part} & = \text{UInt}(Q); \\
\text{integer } \text{elements} & = \text{datasize} \text{ DIV esize}; \\
\text{integer } \text{shift} & = (2 \times \text{esize}) - \text{UInt}(\text{immh}:\text{immb}); \\
\text{boolean } \text{round} & = (\text{op} == '1'); \\
\end{align*}
\]

Assembler symbols

2    Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent]    when Q = 0
[present]    when Q = 1

<Vd>    Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb>    Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B    when immh = 0001, Q = 0
16B    when immh = 0001, Q = 1
4H    when immh = 001x, Q = 0
8H    when immh = 001x, Q = 1
2S    when immh = 01xx, Q = 0
4S    when immh = 01xx, Q = 1

See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.
The encoding \( \text{immh} = 1xxx \), \( Q = x \) is reserved.

\(<Vn>\)
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<Ta>\)
Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

- \( \text{8H} \) when \( \text{immh} = 0001 \)
- \( \text{4S} \) when \( \text{immh} = 001x \)
- \( \text{2D} \) when \( \text{immh} = 01xx \)

See Advanced SIMD modified immediate on page C4-632 when \( \text{immh} = 0000 \).

The encoding \( \text{immh} = 1xxx \) is reserved.

\(<\text{shift}>\)
Is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

- \( (16-\text{UInt}(\text{immh:immb})) \) when \( \text{immh} = 0001 \)
- \( (32-\text{UInt}(\text{immh:immb})) \) when \( \text{immh} = 001x \)
- \( (64-\text{UInt}(\text{immh:immb})) \) when \( \text{immh} = 01xx \)

See Advanced SIMD modified immediate on page C4-632 when \( \text{immh} = 0000 \).

The encoding \( \text{immh} = 1xxx \) is reserved.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bhits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

for e = 0 to elements-1
    element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.257 SHSUB

Signed Halving Subtract. This instruction subtracts the elements in the vector in the second source SIMD&FP register from the corresponding elements in the vector in the first source SIMD&FP register, shifts each result right one bit, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
</tr>
</tbody>
</table>

U

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - 8B when size == 00, Q == 0
  - 16B when size == 00, Q == 1
  - 4H when size == 01, Q == 0
  - 8H when size == 01, Q == 1
  - 2S when size == 10, Q == 0
  - 4S when size == 10, Q == 1
  - The encoding size = 11, Q = x is reserved.
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
b bits(datasize) operand1 = V[n, datasize];
b bits(datasize) operand2 = V[m, datasize];
b bits(datasize) result;
integer element1;
integer element2;
integer diff;
for e = 0 to elements-1
```
element1 = Int(Elem[operand1, e, esize], unsigned);
element2 = Int(Elem[operand2, e, esize], unsigned);
diff = (element1 - element2) >> 1;
Elem[result, e, esize] = diff<esize-1:0>;
V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.258   **SLI**

Shift Left and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, left shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the left of each vector element in the source register are lost.

The following figure shows the operation of shift left by 3 for an 8-bit vector element.

![Figure showing SLI operation](image)

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

<table>
<thead>
<tr>
<th>[31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22</th>
<th>19 18</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>!=0000 immh</td>
</tr>
</tbody>
</table>

**Encoding**

SLI <V><d>, <V><n>, #<shift>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;
```

**Vector**

<table>
<thead>
<tr>
<th>[31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22</th>
<th>19 18</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>!=0000 immh</td>
</tr>
</tbody>
</table>

**Encoding**

SLI <Vd>.<T>, <Vn>.<T>, #<shift>
Decode for this encoding

```plaintext
type integer d = UInt(Rd);
type integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
type integer esize = 8 << HighestSetBit(immh);
type integer datasize = if Q == '1' then 128 else 64;
type integer elements = datasize DIV esize;
type integer shift = UInt(immh:immb) - esize;
```

Assembler symbols

- `<V>`: Is a width specifier, encoded in the "immh" field. It can have the following values:
  - D when immh = 1xxx
  - The encoding immh = 0xxx is reserved.

- `<d>`: Is the number of the SIMD&FP destination register, in the "Rd" field.

- `<n>`: Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- `<T>`: Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:
  - 8B when immh = 0001, Q = 0
  - 16B when immh = 0001, Q = 1
  - 4H when immh = 001x, Q = 0
  - 8H when immh = 001x, Q = 1
  - 2S when immh = 01xx, Q = 0
  - 4S when immh = 01xx, Q = 1
  - 2D when immh = 1xxx, Q = 1

  See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.

  - The encoding immh = 1xxx, Q = 0 is reserved.

- `<Vn>`: Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- `<shift>`: For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in the "immh:immb" field. It can have the following values:
  - (UInt(immh:immb)-64) when immh = 1xxx

  The encoding immh = 0xxx is reserved.

  For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:
  - (UInt(immh:immb)-8) when immh = 0001
  - (UInt(immh:immb)-16) when immh = 001x
  - (UInt(immh:immb)-32) when immh = 01xx
  - (UInt(immh:immb)-64) when immh = 1xxx

  See Advanced SIMD modified immediate on page C4-632 when immh = 0000.

Operation for all encodings

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2 = V[d, datasize];
```
bits(datasize) result;
bits(esize) mask = LSL(Ones(esize), shift);
bits(esize) shifted;
for e = 0 to elements-1
    shifted = LSL(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d, datasize] = result;

Operational information
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.259   SM3PARTW1

SM3PARTW1 takes three 128-bit vectors from the three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD

(FEAT_SM3)

| [31 30 29 28] | [27 26 25 24] | [23 22 21 20] | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 5 | 4 | 0 |
|---------------|---------------|---------------|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | Rm | 1 | 1 | 0 | 0 | 0 | Rn | Rd |

Encoding

SM3PARTW1 <Vd>.4S, <Vn>.4S, <Vm>.4S

Decode for this encoding

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();
bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(128) result;
result<95:0> = (Vd EOR Vn)<95:0> EOR (ROL(Vm<127:96>, 15):ROL(Vm<95:64>, 15):ROL(Vm<63:32>, 15));
for i = 0 to 3
  if i == 3 then
    result<127:96> = (Vd EOR Vn)<127:96> EOR (ROL(result<31:0>, 15));
  result<(32*i)+1:31:(32*i)> = result<(32*i)+1:31:(32*i)> EOR ROL(result<(32*i)+1:31:(32*i)>, 15) EOR ROL(result<(32*i)+1:31:(32*i)>, 23);
V[d, 128] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
— The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.260    SM3PARTW2

SM3PARTW2 takes three 128-bit vectors from three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when FEAT_SM3 is implemented.

**Advanced SIMD**

(FEAT_SM3)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|31 |30 |29 |28 |27 |26 |25 |24 |23 |22 |21 |20 |19 |18 |17 |16 |15 |14 |13 |12 |11 |10 | 9 | 8 | 7 | 6 |
|1  |1  |0  |0  |0  |0  |1  |1  |1  |1  |0  |0  |0  |0  |0  |0  |1  |1  |0  |0  |0  |0  |1  |1  |0  |

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|Rm | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  |

**Encoding**

SM3PARTW2 <Vd>.4S, <Vn>.4S, <Vm>.4S

**Decode for this encoding**

if !HaveSM3Ext() then UNDEFINED;

integer d = UInt(Rd);

integer n = UInt(Rn);

integer m = UInt(Rm);

**Assembler symbols**

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

**Operation**

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];

bits(128) Vn = V[n, 128];

bits(128) Vd = V[d, 128];

bits(128) result;

bits(128) tmp;

bits(32) tmp2;

tmp<127:0> = Vn EOR (ROL(Vm<127:96>, 7):ROL(Vm<95:64>, 7):ROL(Vm<63:32>, 7):ROL(Vm<31:0>, 7));

result<127:0> = Vd<127:0> EOR tmp<127:0>;

tmp2 = ROL(tmp<31:0>, 15);

tmp2 = tmp2 EOR ROL(tmp2, 15) EOR ROL(tmp2, 23);

result<127:96> = result<127:96> EOR tmp2;

V[d, 128] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.261   SM3SS1

SM3SS1 rotates the top 32 bits of the 128-bit vector in the first source SIMD&FP register by 12, and adds that 32-bit value to the two other 32-bit values held in the top 32 bits of each of the 128-bit vectors in the second and third source SIMD&FP registers, rotating this result left by 7 and writing the final result into the top 32 bits of the vector in the destination SIMD&FP register, with the bottom 96 bits of the vector being written to 0.

This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD

(FEAT_SM3)

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 16 | 15 | 14 | 10 | 9 | 5 | 4 | 0 |
|-------------|-------------|-------------|----|----|----|----|----|---|---|---|---|
| 1           | 1           | 0           | 1   | 1  | 0  | 0   | 1  | 1 | 0 |

Encoding

SM3SS1 <Vd>.4S, <Vm>.4S, <Vm>.4S, <Va>.4S

Decode for this encoding

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Va> Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Va = V[a, 128];
bits(128) result;
result<127:96> = ROL((ROL(Vn<127:96>, 12) + Vm<127:96> + Va<127:96>), 7);
result<95:0> = Zeros(96);
V[d, 128] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.262  SM3TT1A

SM3TT1A takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive OR.
- The result of the exclusive OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD

(FEAT_SM3)

Encoding

SM3TT1A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

Decode for this encoding

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);

Assembler symbols

<Vd>  Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>  Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>  Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2> Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;
WjPrime = Elem[Vm, i, 32];
SS2 = Vn<127:96> EOR ROL(Vd<127:96>, 12);
TT1 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT1 = (TT1+Vd<31:0>+SS2+WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d, 128] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SM3TT1B takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The result of the exclusive OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when FEAT_SM3 is implemented.

**Advanced SIMD**

(FEAT_SM3)

$$\begin{array}{cccccccccccccccccccc} 31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 & 23 & 22 & 21 & 20 & 16 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 5 & 4 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & Rm & 1 & 0 & \text{imm2} & 0 & 1 & Rn & Rd \end{array}$$

**Encoding**

SM3TT1B <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

**Decode for this encoding**

```c
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);
```

**Assembler symbols**

- `<Vd>` Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- `<Vn>` Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
- `<imm2>` Is a 32-bit element indexed out of `<Vm>`, encoded in "imm2".

**Operation**

```c
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;
WjPrime = Elem[Vm, i, 32];
```
SS2 = V<127:96> EOR ROL(V<127:96>, 12);
TT1 = (V<127:96> AND V<63:32>) OR (V<127:96> AND V<95:64>) OR (V<63:32> AND V<95:64>);
TT1 = (TT1+V<31:0>+SS2+WjPrime)<31:0>;
result<31:0> = V<63:32>;
result<63:32> = ROL(V<95:64>, 9);
result<95:64> = V<127:96>;
result<127:96> = TT1;
V[d, 128] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.264 SM3TT2A

SM3TT2A takes three 128-bit vectors from three source SIMD&FP register and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive OR.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD

(FEAT_SM3)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1 1 0 0 1 0</td>
<td>Rm</td>
<td>1 0</td>
<td>imm2</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SM3TT2A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

Decode for this encoding

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);

Assembler symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2> Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(32) Wj;
bits(128) result;
bits(32) TT2;

Wj = Elem[Vm, i, 32];
TT2 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT2 = (TT2+Vd<31:0>+Vn<127:96>+Wj)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR ROL(TT2, 17);
V[d, 128] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.265   SM3TT2B

SM3TT2B takes three 128-bit vectors from three source SIMD&FP registers, and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD

(FEAT_SM3)

```
| 31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|-------------|-------------|-------------|-------------|-------------|-------------|
| 1 1 0 0 1 1 0 0 1 0 | Rm 1 0 |imm2 1 1 | Rn | Rd |
```

Encoding

SM3TT2B <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

Decode for this encoding

```c
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);
```

Assembler symbols

- `<Vd>` Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- `<Vn>` Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
- `<imm2>` Is a 32-bit element indexed out of `<Vm>`, encoded in "imm2".

Operation

```c
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(32) Wj;
bits(128) result;
bits(32) TT2;

Wj = Elem[Vm, i, 32];
```
\[
TT2 = (Vd<127:96> \text{ AND } Vd<95:64>) \text{ OR } (\text{NOT}(Vd<127:96>) \text{ AND } Vd<63:32>); \\
TT2 = (TT2+Vd<31:0>+Vn<127:96>+Wj)<31:0>;
\]

\[
\begin{align*}
\text{result}<31:0> &= Vd<63:32>; \\
\text{result}<63:32> &= \text{ROL}(Vd<95:64>, 19); \\
\text{result}<95:64> &= Vd<127:96>; \\
\text{result}<127:96> &= TT2 \text{ EOR ROL}(TT2, 9) \text{ EOR ROL}(TT2, 17); \\
V[d, 128] &= \text{result};
\end{align*}
\]

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.266 SM4E

SM4 Encode takes input data as a 128-bit vector from the first source SIMD&FP register, and four iterations of the round key held as the elements of the 128-bit vector in the second source SIMD&FP register. It encrypts the data by four rounds, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SM4 is implemented.

**Advanced SIMD**

(_FEAT_SM4)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1 1 1 0 1 0 0 0 0 0 0 1 0 0 0</td>
<td>0 1 Rn Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

SM4E <Vd>.4S, <Vn>.4S

**Decode for this encoding**

```c
if (!HaveSM4Ext()) then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
```

**Assembler symbols**

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
AArch64.CheckFPAdvSIMDEnabled();
bits(128) Vn = V[n, 128];
bits(32) intval;
bits(128) roundresult;
bits(32) roundkey;
roundresult = V[d, 128];
for index = 0 to 3
    roundkey = Elem[Vn, index, 32];
    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;
for i = 0 to 3
    Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);
intval = intval EOR ROL(intval, 2) EOR ROL(intval, 10) EOR ROL(intval, 18) EOR ROL(intval, 24);
intval = intval EOR ROL(roundresult<31:0>);
roundresult<31:0> = roundresult<63:32>;
roundresult<63:32> = roundresult<95:64>;
roundresult<95:64> = roundresult<127:96>;
roundresult<127:96> = intval;
V[d, 128] = roundresult;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.267 SM4EKEY

SM4 Key takes an input as a 128-bit vector from the first source SIMD&FP register and a 128-bit constant from the second SIMD&FP register. It derives four iterations of the output key, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SM4 is implemented.

**Advanced SIMD**

(_FEAT_SM4)

Encoding

SM4EKEY <Vd>.4S, <Vn>.4S, <Vm>.4S

**Decode for this encoding**

```
if !HaveSM4Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

**Assembler symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(32) intval;
bits(128) result;
bits(32) const;
bits(128) roundresult;

roundresult = V[n, 128];
for index = 0 to 3
    const = Elem[Vm, index, 32];
    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;
    for i = 0 to 3
        Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);
    intval = intval EOR ROL(intval, 13) EOR ROL(intval, 23);
    intval = intval EOR roundresult<31:0>;
    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;
```
V[d, 128] = roundresult;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( U )</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
</tr>
</tbody>
</table>

**Decoding for this encoding**

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

**Assembler symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - 8B when size = 00, Q = 0
  - 16B when size = 00, Q = 1
  - 4H when size = 01, Q = 0
  - 8H when size = 01, Q = 1
  - 2S when size = 10, Q = 0
  - 4S when size = 10, Q = 1
  - The encoding size = 11, Q = x is reserved.
- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;
```
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
  Elem[result, e, esize] = maxmin<esize-1:0>;
V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.269  SMAXP

Signed Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13</th>
<th>12 11 10 9</th>
<th>8 7 6 5</th>
<th>4 3 2 1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
</tr>
<tr>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
</tr>
</tbody>
</table>

Encoding

SMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');

Assembler symbols

<Vd>   Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>   Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
     8B when size = 00, Q = 0
     16B when size = 00, Q = 1
     4H when size = 01, Q = 0
     8H when size = 01, Q = 1
     2S when size = 10, Q = 0
     4S when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.

<Vn>   Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>   Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
Signed Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Encoding

SMAXV `<V><d>, <Vn>.<T>

### Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');

### Assembler symbols

`<V>` Is the destination width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10

The encoding size = 11 is reserved.

`<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 4S when size = 10, Q = 1

The following encodings are reserved:

- size = 10, Q = 0.
- size = 11, Q = x.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);
V[d, esize] = maxmin<esize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.271  SMIN

Signed Minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
[31 30 29 28 27 26 25 24 23 22 21 20] 16 15 14 13 12 11 10 9 5 4 0
  0 0 1 1 1 0 size 1 1 Rm 0 1 1 0 1 1 Rn Rd 0
```

**Encoding**

SMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');

**Asmmer symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;
    V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.272  SMINP

Signed Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

\[ \text{SMINP} \quad <Vd>.<T>, <Vn>.<T>, <Vm>.<T> \]

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');

Assembler symbols

\(<Vd>\)  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\)  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

\(<Vn>\)  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vm>\)  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;
V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.273  SMINV

Signed Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0  1  1  1  1  0  size  1  1  0  0  0  1  1  0  1  0  Rd  0  Rn
U  op
```

Encoding

SMINV <V><d>, <Vn>.<T>

Decode for this encoding

- `integer d = Uint(Rd);`
- `integer n = Uint(Rn);`
- `if size:Q == '100' then UNDEFINED;`
- `if size == '11' then UNDEFINED;`
- `integer esize = 8 << Uint(size);`
- `integer datasize = if Q == '1' then 128 else 64;`
- `integer elements = datasize DIV esize;`
- `boolean unsigned = (U == '1');`
- `boolean min = (op == '1');`

Assembler symbols

- `<V>` Is the destination width specifier, encoded in the "size" field. It can have the following values:
  - `B` when size = 00
  - `H` when size = 01
  - `S` when size = 10
  - The encoding size = 11 is reserved.
- `<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- `<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - `8B` when size = 00, Q = 0
  - `16B` when size = 00, Q = 1
  - `4H` when size = 01, Q = 0
  - `8H` when size = 01, Q = 1
  - `4S` when size = 10, Q = 1
  - The following encodings are reserved:
    - `size = 10, Q = 0.`
    - `size = 11, Q = x.`
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);
V[d, esize] = maxmin<esize-1:0>;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.274 SMLAL, SMLAL2 (by element)

Signed Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

The SMLAL instruction extracts vector elements from the lower half of the first source register. The SMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
  [absent] when Q = 0
  [present] when Q = 1
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:
  4S when size = 01
  2D when size = 10
The following encodings are reserved:
- size = 00.
- size = 11.

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Tb>\) Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The following encodings are reserved:
- size = 00, Q = x.
- size = 11, Q = x.

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:
- 0:Rm when size = 01
- M:Rm when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

Restricted to V0-V15 when element size \(<Ts>\) is H.

\(<Ts>\) Is an element size specifier, encoded in the "size" field. It can have the following values:
- H when size = 01
- S when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

\(<index>\) Is the element index, encoded in the "size:L:H:M" field. It can have the following values:
- H:L:M when size = 01
- H:L when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

Operation

\ CHECKFPAdvSIMDEnabled64();
\ bits(datasize) operand1 = Vpart[n, part, datasize];
\ bits(idxdsize) operand2 = V[m, idxdsize];
\ bits(2*datasize) operand3 = V[d, 2*datasize];
\ bits(2*datasize) result;
\ integer element1;
\ integer element2;
\ bits(2*esize) product;

\ element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
  \ element1 = Int(Elem[operand1, e, esize], unsigned);
  \ product = (element1*element2)<2*esize-1:0>;
if sub_op then
    Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
else
    Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;
V[d, 2*datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.275  SMLAL, SMLAL2 (vector)

Signed Multiply-Add Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLAL instruction extracts each source vector from the lower half of each source register. The SMLAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

Decode for this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler symbols

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

| [absent]     | when Q = 0 |
| [present]    | when Q = 1 |

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  Is an arrangement specifier, encoded in the "size" field. It can have the following values:

| 8H  | when size = 00 |
| 4S  | when size = 01 |
| 2D  | when size = 10 |

The encoding size = 11 is reserved.

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| 8B  | when size = 00, Q = 0 |
| 16B | when size = 00, Q = 1 |
When size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.

<\m>
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<<2*esize-1:0;  
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;
V[d, 2*datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.276   SMLSL, SMLSL2 (by element)

Signed Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLSL instruction extracts vector elements from the lower half of the first source register. The SMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsizem = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esizem = 8 << UInt(size);
integer datasizem = 64;
integer part = UInt(Q);
integer elements = datasizem DIV esizem;
boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');

Assembler symbols

2   Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
    [absent] when Q = 0
    [present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:
    4S when size = 01
    2D when size = 10
The following encodings are reserved:
- size = 00.
- size = 11.

<Tn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The following encodings are reserved:
- size = 00, Q = x.
- size = 11, Q = x.

<Tm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:
- 0:Rm when size = 01
- M:Rm when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:
- H when size = 01
- S when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:
- H:L:M when size = 01
- H:L when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bis(2*datasize) operand3 = V[d, 2*datasize];
bis(2*datasize) result;
integer element1;
integer element2;
bis(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<<2*esize-1:0;  
```
if sub_op then
    Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
else
    Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d, 2*datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.277  SMLSL, SMLSL2 (vector)

Signed Multiply-Subtract Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLSL instruction extracts each source vector from the lower half of each source register. The SMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

Assembler symbols

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
   [absent]  when Q = 0
   [present]  when Q = 1

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  Is an arrangement specifier, encoded in the "size" field. It can have the following values:
   8H  when size = 00
   4S  when size = 01
   2D  when size = 10
   The encoding size = 11 is reserved.

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
   8B  when size = 00, Q = 0
   16B when size = 00, Q = 1
4H  when size = 01, Q = 0
8H  when size = 01, Q = 1
2S  when size = 10, Q = 0
4S  when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.

<\m>

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  product = (element1*element2)<2*esize-1:0>;
  if sub_op then
    accum = Elem[operand3, e, 2*esize] - product;
  else
    accum = Elem[operand3, e, 2*esize] + product;
  Elem[result, e, 2*esize] = accum;
V[d, 2*datasize] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.278   SMMLA (vector)

Signed 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of signed 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

**Vector**

(_FEAT_I8MM)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Rm 1 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>Rn 1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>Rd 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

SMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

**Decode for this encoding**

if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);

**Assembler symbols**

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n, 128];
bits(128) operand2 = V[m, 128];
bits(128) addend = V[d, 128];

V[d, 128] = MatMulAdd(addend, operand1, operand2, FALSE, FALSE);
C7.2.279   SMOV

Signed Move vector element to general-purpose register. This instruction reads the signed integer from the source SIMD&FP register, sign-extends it to form a 32-bit or 64-bit value, and writes the result to destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

32-bit variant

Applies when Q == 0.

SMOV <Wd>, <Vn>.<Ts>[<index>]

64-bit variant

Applies when Q == 1.

SMOV <Xd>, <Vn>.<Ts>[<index>]

Decode for all variants of this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
    when 'xxxxx1' size = 0;    // SMOV [WX]d, Vn.B
    when 'xxxx10' size = 1;    // SMOV [WX]d, Vn.H
    when '1xx100' size = 2;    // SMOV Xd, Vn.S
    otherwise UNDEFINED;

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ts> For the 32-bit variant: is an element size specifier, encoded in the "imm5" field. It can have the following values:

   B when imm5 = xxxx1
   H when imm5 = xxxx10

   The encoding imm5 = xxxx00 is reserved.

For the 64-bit variant: is an element size specifier, encoded in the "imm5" field. It can have the following values:

   B when imm5 = xxxx1
H when \( \text{imm5} = \text{xxx10} \)

S when \( \text{imm5} = \text{xx100} \)

The encoding \( \text{imm5} = \text{xx000} \) is reserved.

\(<\text{index}\>\)

For the 32-bit variant: is the element index encoded in the "imm5" field. It can have the following values:

\( \text{imm5}<4:1> \) when \( \text{imm5} = \text{xxxx1} \)

\( \text{imm5}<4:2> \) when \( \text{imm5} = \text{xxx10} \)

The encoding \( \text{imm5} = \text{xxx00} \) is reserved.

For the 64-bit variant: is the element index encoded in the "imm5" field. It can have the following values:

\( \text{imm5}<4:1> \) when \( \text{imm5} = \text{xxxx1} \)

\( \text{imm5}<4:2> \) when \( \text{imm5} = \text{xxx10} \)

\( \text{imm5}<4:3> \) when \( \text{imm5} = \text{xx100} \)

The encoding \( \text{imm5} = \text{xx000} \) is reserved.

**Operation**

```c
if \( \text{index} == 0 \) then
    \text{CheckFPEnabled64}();
else
    \text{CheckFPAdvSIMDEnabled64}();
bits(\text{idxdsize}) \text{operand} = \text{V}[n, \text{idxdsize}];

\text{X}[d, \text{datasize}] = \text{SignExtend}((\text{Elem}\text{operand, index, esize}, \text{datasize});
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.280  SMULL, SMULL2 (by element)

Signed Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts vector elements from the lower half of the first source register. The SMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 | 5 4 | 0 |
|-------------|-------------|-------------|-------------|-------------|-------------|     |
| 0           | Q           | 0           | 1           | 1           | 1           | size |
| L           | M           | Rm           | 1           | 0           | 1           | H    |
| 0           |             | Rn           |             |             | Rd          |
```

**Encoding**

SMULL[Z] <Vd>,<Ta>, <Vn>,<Tb>, <Vm>.<Ts>[<index>]

**Decode for this encoding**

```plaintext
integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;

case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

**Assembler symbols**

2  

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd>  

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  

Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 4S when size = 01
- 2D when size = 10

The following encodings are reserved:

- size = 00.
• size = 11.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
The following encodings are reserved:
• size = 00, Q = x.
• size = 11, Q = x.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:
0:Rm when size = 01
M:Rm when size = 10
The following encodings are reserved:
• size = 00.
• size = 11.
Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
The following encodings are reserved:
• size = 00.
• size = 11.

<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:
H:L:M when size = 01
H:L when size = 10
The following encodings are reserved:
• size = 00.
• size = 11.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d, 2*datasize] = result;
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.281 SMULL, SMULL2 (vector)

Signed Multiply Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts each source vector from the lower half of each source register. The SMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
\hline
0 & Q & 0 & 0 & 1 & 1 & 1 & 0 & \text{size} & 1 & \hline
0 & Rm & 1 & 1 & 0 & 0 & 0 & \hline
Vd & \hline
Rn & \hline
Rd & \hline
\end{array}
\]

**Encoding**

SMULL[2] <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');

**Assembler symbols**

2  
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0
[present] when Q = 1

<Vd>  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  
Is an arrangement specifier, encoded in the "size" field. It can have the following values:

8H when size = 00
4S when size = 01
2D when size = 10

The encoding size = 11 is reserved.

<Vn>  
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>  
Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.

<Vm> is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
for e = 0 to elements-1
   element1 = Int(Elem[operand1, e, esize], unsigned);
   element2 = Int(Elem[operand2, e, esize], unsigned);
   Elem[result, e, 2*esize] = (element1*element2)<2*esize-1:0>;
V[d, 2*datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.282   SQABS

Signed saturating Absolute value. This instruction reads each vector element from the source SIMD&FP register, puts the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

\[
\begin{array}{|ccccccccccccc|cc|}
\hline
0 & 1 & 0 & 1 & 1 & 1 & 0 & | & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & | & Rn & Rd \\
\hline
\end{array}
\]

Encoding

SQABS <V><d>, <V><n>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');

Vector

\[
\begin{array}{|cccccccccccccccccc|cc|}
\hline
0 & Q & 0 & 0 & 1 & 1 & 1 & 0 & | & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & | & Rn & Rd \\
\hline
\end{array}
\]

Encoding

SQABS <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

Assembler symbols

<\text{V}> Is a width specifier, encoded in the "size" field. It can have the following values:

- 8 when size = 00
H  when size = 01
S  when size = 10
D  when size = 11

<d>  Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n>  Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
8B  when size = 00, Q = 0
16B when size = 00, Q = 1
4H  when size = 01, Q = 0
8H  when size = 01, Q = 1
2S  when size = 10, Q = 0
4S  when size = 10, Q = 1
2D  when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
  element = SInt(Elem[operand, e, esize]);
  if neg then
    element = -element;
  else
    element = Abs(element);
  (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
  if sat then FPSR.QC = '1';
V[d, datasize] = result;
C7.2.283   SQADD

Signed saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 5 | 4 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | size | 1  |     | Rm  | 0  | 0  | 0  | 1  | 1  |     | Rn  |     | Rd  | 0  |
| U  |    |    |    |    |    |    |      |     |    |     |    |    |    |    |    |    |    |    |    |    |

**Encoding**

SQADD <V><d>, <V><n>, <V><m>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 5 | 4 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 0  | size | 1  |     | Rm  | 0  | 0  | 0  | 1  | 1  |     | Rn  |     | Rd  | 0  |
| U  |    |    |    |    |    |      |     |    |     |    |    |    |    |    |    |    |    |    |    |    |

**Encoding**

SQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

**Assembler symbols**

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B  when size = 00
- H  when size = 01
<table>
<thead>
<tr>
<th>&lt;d&gt;</th>
<th>Is the number of the SIMD&amp;FP destination register, in the &quot;Rd&quot; field.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;n&gt;</td>
<td>Is the number of the first SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>&lt;m&gt;</td>
<td>Is the number of the second SIMD&amp;FP source register, encoded in the &quot;Rm&quot; field.</td>
</tr>
<tr>
<td>&lt;Vd&gt;</td>
<td>Is the name of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>&lt;T&gt;</td>
<td>Is an arrangement specifier, encoded in the &quot;size:Q&quot; field. It can have the following values:</td>
</tr>
<tr>
<td>8B</td>
<td>when size = 00, Q = 0</td>
</tr>
<tr>
<td>16B</td>
<td>when size = 00, Q = 1</td>
</tr>
<tr>
<td>4H</td>
<td>when size = 01, Q = 0</td>
</tr>
<tr>
<td>8H</td>
<td>when size = 01, Q = 1</td>
</tr>
<tr>
<td>2S</td>
<td>when size = 10, Q = 0</td>
</tr>
<tr>
<td>4S</td>
<td>when size = 10, Q = 1</td>
</tr>
<tr>
<td>2D</td>
<td>when size = 11, Q = 1</td>
</tr>
<tr>
<td></td>
<td>The encoding size = 11, Q = 0 is reserved.</td>
</tr>
<tr>
<td>&lt;Vn&gt;</td>
<td>Is the name of the first SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>&lt;Vm&gt;</td>
<td>Is the name of the second SIMD&amp;FP source register, encoded in the &quot;Rm&quot; field.</td>
</tr>
</tbody>
</table>

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d, datasize] = result;
```
C7.2.284  SQDMLAL, SQDMLAL2 (by element)

Signed saturating Doubling Multiply-Add Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMLAL instruction extracts vector elements from the lower half of the first source register. The SQDMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|14 13 12|11 10 9 | 5 4 | 0 ]
0 1 0 1 1 1 1 | size L M | Rm 0 0 1 1 | H 0 | Rn | Rd
```

Encoding

SQDMLAL `<Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]`

Decode for this encoding

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
boolean sub_op = (o2 == '1');
```

Vector

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|14 13 12|11 10 9 | 5 4 | 0 ]
0 0 0 1 1 1 1 | size L M | Rm 0 0 1 1 | H 0 | Rn | Rd
```

Encoding

SQDMLAL(2) `<Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]`
**Decode for this encoding**

```plaintext
decode for this encoding

integer idxsize = if H == '1' then 128 else 64;
decode index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

**Assembler symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[absent]</td>
<td>when Q = 0</td>
</tr>
<tr>
<td>[present]</td>
<td>when Q = 1</td>
</tr>
</tbody>
</table>

&lt;Vd&gt; Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

&lt;Ta&gt; Is an arrangement specifier, encoded in the "size" field. It can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4S</td>
<td>when size = 01</td>
</tr>
<tr>
<td>2D</td>
<td>when size = 10</td>
</tr>
</tbody>
</table>

The following encodings are reserved:

- size = 00.
- size = 11.

&lt;Vn&gt; Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

&lt;Tb&gt; Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4H</td>
<td>when size = 01, Q = 0</td>
</tr>
<tr>
<td>8H</td>
<td>when size = 01, Q = 1</td>
</tr>
<tr>
<td>2S</td>
<td>when size = 10, Q = 0</td>
</tr>
<tr>
<td>4S</td>
<td>when size = 10, Q = 1</td>
</tr>
</tbody>
</table>

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

&lt;Va&gt; Is the destination width specifier, encoded in the "size" field. It can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>when size = 01</td>
</tr>
<tr>
<td>D</td>
<td>when size = 10</td>
</tr>
</tbody>
</table>

The following encodings are reserved:

- size = 00.
- size = 11.
<d>
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb>
Is the source width specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10

The following encodings are reserved:
• size = 00.
• size = 11.

<n>
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>
Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

0:Rm when size = 01
M:Rm when size = 10

The following encodings are reserved:
• size = 00.
• size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts>
Is an element size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10

The following encodings are reserved:
• size = 00.
• size = 11.

<index>
Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

H:L:M when size = 01
H:L when size = 10

The following encodings are reserved:
• size = 00.
• size = 11.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
  element1 = SInt(Elem[operand1, e, esize]);
  (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
  if sub_op then
    accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
  else

ARM DDI 04871.a
Non-Confidential
\[
\text{accum} = \text{SInt}(\text{Elem}[\text{operand3}, e, 2*\text{esize}]) + \text{SInt}(\text{product});
\]
\[
(\text{Elem}[\text{result}, e, 2*\text{esize}], \text{sat2}) = \text{SignedSatQ}(\text{accum}, 2*\text{esize});
\]
\[
\text{if sat1 || sat2 then FPSR.QC = '1';}
\]
\[
\text{V}[d, 2*\text{datasize}] = \text{result};
\]
C7.2.285  SQDMLAL, SQDMLAL2 (vector)

Signed saturating Doubling Multiply-Add Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMLAL instruction extracts each source vector from the lower half of each source register. The SQDMLAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

\[
\begin{array}{cccccccccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccccccccc}
0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & size & 1 & Rm & 1 & 0 & 1 & 0 & 0 & Rn & Rd & 01
\end{array}
\]

**Encoding**

SQDMLAL <Va><d>, <Vb><n>, <Vb><m>

**Decode for this encoding**

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{if } \text{size} &= '00' \text{ or } \text{size} &= '11' \text{ then UNDEFINED;} \\
\text{integer } esize &= 8 \times \text{UInt}(\text{size}); \\
\text{integer } datasize &= \text{esize}; \\
\text{integer } elements &= 1; \\
\text{integer } part &= 0; \\
\text{boolean } sub_op &= (o1 &= '1');
\end{align*}
\]

**Vector**

\[
\begin{array}{cccccccccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccccccccc}
0 & Q & 0 & 0 & 1 & 1 & 1 & 0 & size & 1 & Rm & 1 & 0 & 1 & 0 & 0 & Rn & Rd & 01
\end{array}
\]

**Encoding**

SQDMLAL{2} <Vd>.<Ta>, <Vn>.< Tb>, <Vm>.<Tb>

**Decode for this encoding**

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{if } \text{size} &= '00' \text{ or } \text{size} &= '11' \text{ then UNDEFINED;} \\
\text{integer } esize &= 8 \times \text{UInt}(\text{size}); \\
\text{integer } datasize &= 64; \\
\text{integer } part &= \text{UInt}(Q);
\end{align*}
\]
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');

**Assembler symbols**

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 4S when size = 01
- 2D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va>  Is the destination width specifier, encoded in the "size" field. It can have the following values:

- S when size = 01
- D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<d>  Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<vb>  Is the source width specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<n>  Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<mm>  Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';
V[d, 2*datasize] = result;
C7.2.286  SQDMLSL, SQDMLSL2 (by element)

Signed saturating Doubling Multiply-Subtract Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMLSL instruction extracts vector elements from the lower half of the first source register. The SQDMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[19 16 15 14 13 12]</th>
<th>[11 10 9</th>
<th>5 4</th>
<th>0 ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1</td>
<td>0 1 1</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>0 1 1 1</td>
<td>H 0</td>
</tr>
</tbody>
</table>

Encoding

SQDMLSL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
boolean sub_op = (o2 == '1');

Vector

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[19 16 15 14 13 12]</th>
<th>[11 10 9</th>
<th>5 4</th>
<th>0 ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 1 1 1</td>
<td>0 1 1</td>
<td>size</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>0 1 1 1</td>
</tr>
</tbody>
</table>

Encoding

SQDMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]
**Decode for this encoding**

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');

**Assembler symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 4S when size = 01
- 2D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

<Va> Is the destination width specifier, encoded in the "size" field. It can have the following values:

- S when size = 01
- D when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.
<d>
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb>
Is the source width specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
The following encodings are reserved:
• size = 00.
• size = 11.

<n>
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>
Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have
the following values:
0:Rm when size = 01
M:Rm when size = 10
The following encodings are reserved:
• size = 00.
• size = 11.
Restricted to V0-V15 when element size <Ts> is H.

<Ts>
Is an element size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
The following encodings are reserved:
• size = 00.
• size = 11.

<index>
Is the element index, encoded in the "size:L:H:M" field. It can have the following values:
H:L:M when size = 01
H:L when size = 10
The following encodings are reserved:
• size = 00.
• size = 11.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
biteS(datasize) operand1 = Vpart[n, part, datasize];
biteS(idxdsize) operand2 = V[m, idxdsize];
biteS(2+datasize) operand3 = V[d, 2+datasize];
biteS(2+datasize) result;
integer element1;
integer element2;
biteS(2+esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2+esize]) - SInt(product);
    else
\[
\text{accum} = \text{SInt(Elem[operand3, e, 2*esize])} + \text{SInt(product)};
\]
\[
(\text{Elem[result, e, 2*esize], sat2}) = \text{SignedSatQ(accum, 2 * esize)};
\]
\[
\text{if sat1 || sat2 then FPSR.QC = '1';}
\]
\[
\text{V[d, 2*datasize] = result;}
\]
C7.2.287   SQDMLSL, SQDMLSL2 (vector)

Signed saturating Doubling Multiply-Subtract Long. This instruction multiplies corresponding signed integer
values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and
subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector
elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation
bit FPSR.QC is set.

The SQDMLSL instruction extracts each source vector from the lower half of each source register. The SQDMLSL2
instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

Scalar

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 0 1 1 1 0 | size 1 | Rm 1 0 1 1 0 0 | Rn | Rd
```

Encoding

SQDMLSL <Va><d>, <Vb><n>, <Vb><m>

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

Vector

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 Q 0 1 1 1 0 | size 1 | Rm 1 0 1 1 0 0 | Rn | Rd
```

Encoding

SQDMLSL[2] <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
```
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');

**Assembler symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- **[absent]** when Q = 0
- **[present]** when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- **4S** when size = 01
- **2D** when size = 10

The following encodings are reserved:

- `size = 00`
- `size = 11`

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- **4H** when size = 01, Q = 0
- **8H** when size = 01, Q = 1
- **2S** when size = 10, Q = 0
- **4S** when size = 10, Q = 1

The following encodings are reserved:

- `size = 00, Q = x`
- `size = 11, Q = x`

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in the "size" field. It can have the following values:

- **S** when size = 01
- **D** when size = 10

The following encodings are reserved:

- `size = 00`
- `size = 11`

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<vb> Is the source width specifier, encoded in the "size" field. It can have the following values:

- **H** when size = 01
- **S** when size = 10

The following encodings are reserved:

- `size = 00`
- `size = 11`

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<no> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;
for e = 0 to elements-1
  element1 = SInt(Elem[operand1, e, esize]);
  element2 = SInt(Elem[operand2, e, esize]);
  (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
  if sub_op then
    accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
  else
    accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
  (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
if sat1 || sat2 then FPSR.QC = '1';
V[d, 2*datasize] = result;
C7.2.288  SQDMULH (by element)

Signed saturating Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see SQRDMULH (by element).

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

| 31 30 29 28| 27 26 25 24| 23 22 21 20| 19 16| 15 14 13 12| 11 10 9 | 5 4 | 0 |
| 0 1 0 1 1 1 1 | size L M Rm 1 1 0 0 H 0 Rn Rd |

Encoding

SQDMULH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');

Vector

| 31 30 29 28| 27 26 25 24| 23 22 21 20| 19 16| 15 14 13 12| 11 10 9 | 5 4 | 0 |
| 0 Q 0 0 1 1 1 1 | size L M Rm 1 1 0 0 H 0 Rn Rd |

Encoding

SQDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
   H when size = 01
   S when size = 10

The following encodings are reserved:
   • size = 00.
   • size = 11.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
   4H when size = 01, Q = 0
   8H when size = 01, Q = 1
   2S when size = 10, Q = 0
   4S when size = 10, Q = 1

The following encodings are reserved:
   • size = 00, Q = x.
   • size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:
   0:Rm when size = 01
   M:Rm when size = 10

The following encodings are reserved:
   • size = 00.
   • size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:
   H when size = 01
   S when size = 10

The following encodings are reserved:
   • size = 00.
   • size = 11.
<index> Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

<table>
<thead>
<tr>
<th>size</th>
<th>L</th>
<th>H</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following encodings are reserved:

- size = 00
- size = 11

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (2 * element1 * element2) + round_const;
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.289  SQDMULH (vector)

Signed saturating Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see SQRDMULH (vector).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

<table>
<thead>
<tr>
<th>[31 30 29 28][27 26 25 24][23 22 21 20]</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0           1          0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size 1</td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SQDMULH <V><d>, <V><n>, <V><m>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');

Vector

<table>
<thead>
<tr>
<th>[31 30 29 28][27 26 25 24][23 22 21 20]</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0           Q          0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SQDMULH <V>d>.<T>, <V>n>.<T>, <V>m>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');
Assembler symbols

\(<V>\) Is a width specifier, encoded in the "size" field. It can have the following values:
- \(H\) when \(size = 01\)
- \(S\) when \(size = 10\)

The following encodings are reserved:
- \(size = 00\).
- \(size = 11\).

\(<d>\) Is the number of the SIMD&FP destination register, in the "Rd" field.

\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<m>\) Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
- \(4H\) when \(size = 01, Q = 0\)
- \(8H\) when \(size = 01, Q = 1\)
- \(2S\) when \(size = 10, Q = 0\)
- \(4S\) when \(size = 10, Q = 1\)

The following encodings are reserved:
- \(size = 00, Q = x\).
- \(size = 11, Q = x\).

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

for e = 0 to elements-1
  element1 = SInt(Elem[operand1, e, esize]);
  element2 = SInt(Elem[operand2, e, esize]);
  product = (2 * element1 * element2) + round_const;
  (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
  if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.290  SQDMULL, SQDMULL2 (by element)

Signed saturating Doubling Multiply Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMULL instruction extracts the first source vector from the lower half of the first source register. The SQDMULL2 instruction extracts the first source vector from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16 15 14 13 12|11 10 9 | 5 4 | 0 ]
```

```
0 1 0 1 1 1 1 | size L M | Rm 1 0 1 1 | H 0 | Rn | Rd
```

**Encoding**

SQDMULL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

**Decode for this encoding**

```plaintext
integer iddxsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

**Vector**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16 15 14 13 12|11 10 9 | 5 4 | 0 ]
```

```
0 Q 0 1 1 1 1 | size L M | Rm 1 0 1 1 | H 0 | Rn | Rd
```

**Encoding**

SQDMULL[2] <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

**Decode for this encoding**

```plaintext
integer iddxsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
```
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper
64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have
the following values:
  [absent] when Q = 0
  [present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

  4S when size = 01
  2D when size = 10

The following encodings are reserved:
  * size = 00.
  * size = 11.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

  4H when size = 01, Q = 0
  8H when size = 01, Q = 1
  2S when size = 10, Q = 0
  4S when size = 10, Q = 1

The following encodings are reserved:
  * size = 00, Q = x.
  * size = 11, Q = x.

<VA> Is the destination width specifier, encoded in the "size" field. It can have the following values:

  S when size = 01
  D when size = 10

The following encodings are reserved:
  * size = 00.
  * size = 11.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<vb> Is the source width specifier, encoded in the "size" field. It can have the following values:

  H when size = 01
  S when size = 10
The following encodings are reserved:
- size = 00.
- size = 11.

<n>
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>
Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:
- 0:Rm when size = 01
- M:Rm when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<Ts>
Is an element size specifier, encoded in the "size" field. It can have the following values:
- H when size = 01
- S when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

<index>
Is the element index, encoded in the "size:L:H:M" field. It can have the following values:
- H:L:M when size = 01
- H:L when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

Operation for all encodings

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
  element1 = SInt(Elem[operand1, e, esize]);
  (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize);
  Elem[result, e, 2*esize] = product;
  if sat then FPSR.QC = '1';

V[d, 2*datasize] = result;
C7.2.291 SQDMULL, SQDMULL2 (vector)

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMULL instruction extracts each source vector from the lower half of each source register. The SQDMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

C7.2.291 SQDMULL, SQDMULL2 (vector)

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMULL instruction extracts each source vector from the lower half of each source register. The SQDMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

C7.2.291 SQDMULL, SQDMULL2 (vector)

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMULL instruction extracts each source vector from the lower half of each source register. The SQDMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

C7.2.291 SQDMULL, SQDMULL2 (vector)

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMULL instruction extracts each source vector from the lower half of each source register. The SQDMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

C7.2.291 SQDMULL, SQDMULL2 (vector)

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMULL instruction extracts each source vector from the lower half of each source register. The SQDMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

C7.2.291 SQDMULL, SQDMULL2 (vector)

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMULL instruction extracts each source vector from the lower half of each source register. The SQDMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
Assembler symbols

\textbf{2} Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- \textbf{[absent]} when Q = 0
- \textbf{[present]} when Q = 1

\textbf{<Vd>} Is the name of the SIMD\&FP destination register, encoded in the "Rd" field.

\textbf{<Ta>} Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- \textbf{4S} when size = 01
- \textbf{2D} when size = 10

The following encodings are reserved:

- \textbf{size = 00.}
- \textbf{size = 11.}

\textbf{<Vn>} Is the name of the first SIMD\&FP source register, encoded in the "Rn" field.

\textbf{<Tb>} Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- \textbf{4H} when size = 01, Q = 0
- \textbf{8H} when size = 01, Q = 1
- \textbf{2S} when size = 10, Q = 0
- \textbf{4S} when size = 10, Q = 1

The following encodings are reserved:

- \textbf{size = 00, Q = x.}
- \textbf{size = 11, Q = x.}

\textbf{<Vm>} Is the name of the second SIMD\&FP source register, encoded in the "Rm" field.

\textbf{<Va>} Is the destination width specifier, encoded in the "size" field. It can have the following values:

- \textbf{S} when size = 01
- \textbf{D} when size = 10

The following encodings are reserved:

- \textbf{size = 00.}
- \textbf{size = 11.}

\textbf{<d>} Is the number of the SIMD\&FP destination register, encoded in the "Rd" field.

\textbf{<Vb>} Is the source width specifier, encoded in the "size" field. It can have the following values:

- \textbf{H} when size = 01
- \textbf{S} when size = 10

The following encodings are reserved:

- \textbf{size = 00.}
- \textbf{size = 11.}

\textbf{<n>} Is the number of the first SIMD\&FP source register, encoded in the "Rn" field.

\textbf{<>} Is the number of the second SIMD\&FP source register, encoded in the "Rm" field.
Operation for all encodings

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d, 2*datasize] = result;
```
C7.2.292   SQNEG

Signed saturating Negate. This instruction reads each vector element from the source SIMD&FP register, negates each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>19 18 17 16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Encoding

SQNEG <Vd>, <Vn>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>19 18 17 16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Encoding

SQNEG <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation for all encodings**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements - 1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';
V[d, datasize] = result;
C7.2.293 SQRDMLAH (by element)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

(_FEAT_RDM)

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
0 1 1 1 1 1 1 | size L  M | Rm  1 1 0 1 | H 0 | Rn  | Rd |
```

**Encoding**

SQRDMLAH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

**Decode for this encoding**

if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
  case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;
  integer d = UInt(Rd);
  integer n = UInt(Rn);
  integer m = UInt(Rmhi:Rm);
  integer esize = 8 << UInt(size);
  integer datasize = esize;
  integer elements = 1;
  boolean rounding = TRUE;
  boolean sub_op = (S == '1');

**Vector**

(_FEAT_RDM)

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
0 | 0 1 0 1 1 1 1 | size L  M | Rm  1 1 0 1 | H 0 | Rn  | Rd |
```

**Encoding**

SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
Decode for this encoding

if !HaveQRDLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

Assembler symbols

\textless V \rangle \quad \text{Is a width specifier, encoded in the "size" field. It can have the following values:}
  \begin{align*}
  &H \quad \text{when size = 01} \\
  &S \quad \text{when size = 10}
  \end{align*}

The following encodings are reserved:
  \begin{itemize}
  \item size = 00.
  \item size = 11.
  \end{itemize}

\textless d \rangle \quad \text{Is the number of the SIMD&FP destination register, encoded in the "Rd" field.}

\textless n \rangle \quad \text{Is the number of the first SIMD&FP source register, encoded in the "Rn" field.}

\textless Vd \rangle \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.}

\textless T \rangle \quad \text{Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:}
  \begin{align*}
  &4H \quad \text{when size = 01, Q = 0} \\
  &8H \quad \text{when size = 01, Q = 1} \\
  &2S \quad \text{when size = 10, Q = 0} \\
  &4S \quad \text{when size = 10, Q = 1}
  \end{align*}

The following encodings are reserved:
  \begin{itemize}
  \item size = 00, Q = x.
  \item size = 11, Q = x.
  \end{itemize}

\textless Vn \rangle \quad \text{Is the name of the first SIMD&FP source register, encoded in the "Rn" field.}

\textless Vm \rangle \quad \text{Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:}
  \begin{align*}
  &0:Rm \quad \text{when size = 01} \\
  &M:Rm \quad \text{when size = 10}
  \end{align*}

The following encodings are reserved:
  \begin{itemize}
  \item size = 00.
  \item size = 11.
  \end{itemize}
Restricted to V0-V15 when element size $<T_s>$ is H.

$<T_s>$
Is an element size specifier, encoded in the "size" field. It can have the following values:

- **H** when size = 01
- **S** when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

$<\text{index}>$
Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

- **H:L:M** when size = 01
- **H:L** when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
integer accum;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.294  SQRDMLAH (vector)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

(FEAT_RDM)

Encoding

SQRDMLAH <V><d>, <V><n>, <V><m>

Decode for this encoding

if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer data_size = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');

Vector

(_FEAT_RDM)

Encoding

SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');

Assembler symbols

<\V> Is a width specifier, encoded in the "size" field. It can have the following values:
    H    when size = 01
    S    when size = 10
The following encodings are reserved:
    • size = 00.
    • size = 11.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<v>d> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<v> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
    4H    when size = 01, Q = 0
    8H    when size = 01, Q = 1
    2S    when size = 10, Q = 0
    4S    when size = 10, Q = 1
The following encodings are reserved:
    • size = 00, Q = x.
    • size = 11, Q = x.

<v>n> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<v>m> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
integer accum;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
if sat then FPSR.QC = '1';

∀[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.295  SQRDMLSH (by element)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

( FEAT_RDM )

Encoding

SQRDMLSH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

Decode for this encoding

if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

Vector

( FEAT_RDM )

Encoding

SQRDMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
Decide for this encoding

if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

Assembler symbols

<d> Is a width specifier, encoded in the "size" field. It can have the following values:
    H when size = 01
    S when size = 10

The following encodings are reserved:
    • size = 00.
    • size = 11.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
    4H when size = 01, Q = 0
    8H when size = 01, Q = 1
    2S when size = 10, Q = 0
    4S when size = 10, Q = 1

The following encodings are reserved:
    • size = 00, Q = x.
    • size = 11, Q = x.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:
    0:Rm when size = 01
    M:Rm when size = 10

The following encodings are reserved:
    • size = 00.
    • size = 11.
Restricted to V0-V15 when element size \(<Ts>\) is H.

\(<Ts>\) is an element size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

\(<index>\) is the element index, encoded in the "size:L:H:M" field. It can have the following values:

- H:L:M when size = 01
- H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;

integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
integer accum;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```
C7.2.296   SQRDMLSH (vector)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

(FEAT_RDM)

```
0 1 1 1 1 1 0 | size 0 | Rm 1 0 0 1 1 | Rn Rd
```

Encoding

SQRMLSH <V><d>, <V><n>, <V><m>

Decode for this encoding

if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');

Vector

(FEAT_RDM)

```
0 0 1 0 1 1 0 | size 0 | Rm 1 0 0 1 1 | Rn Rd
```

Encoding

SQRMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');

Assembler symbols

<\v> Is a width specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
The following encodings are reserved:
• size = 00.
• size = 11.
<\d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<\n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<\m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<\Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<\T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
The following encodings are reserved:
• size = 00, Q = x.
• size = 11, Q = x.
<\Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<\Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
integer accum;
boolean sat;
for e = 0 to elements-1
   element1 = SInt(Elem[operand1, e, esize]);
   element2 = SInt(Elem[operand2, e, esize]);
   element3 = SInt(Elem[operand3, e, esize]);
   if sub_op then
      accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
   else
      accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
   (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
if sat then FPSR.QC = '1';

V[d, datasize] = result;
C7.2.297   SQRDMULH (by element)

Signed saturating Rounding Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see SQRDMULH (by element).

If any of the results overflows, they are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

Encoding

SQRDMULH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean round = (op == '1');

Vector

Encoding

SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
when '01' index = Uint(H:L:M); Rmhi = '0';
when '10' index = Uint(H:L); Rmhi = M;
otherwise UNDEFINED;

integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rmhi:Rm);

integer esize = 8 << Uint(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');

### Assembler symbols

<\V> Is a width specifier, encoded in the "size" field. It can have the following values:
- H when size = 01
- S when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

<\d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<\n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<\Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<\T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The following encodings are reserved:
- size = 00, Q = x.
- size = 11, Q = x.

<\Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<\Vm> Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:
- 0:Rm when size = 01
- M:Rm when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

<\Ts> Is an element size specifier, encoded in the "size" field. It can have the following values:
- H when size = 01
- S when size = 10

The following encodings are reserved:
- size = 00.
• size = 11.

<i>index</i> is the element index, encoded in the "size:L:H:M" field. It can have the following values:

- H:L:M when size = 01
- H:L when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;
element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (2 * element1 * element2) + round_const;
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
if sat then FPSR.QC = '1';
V[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.298  **SQRDMULH (vector)**

Signed saturating Rounding Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see **SQDMULH (vector)**.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit **FPSCR.QC** is set.

Depending on the settings in the **CPACR_EL1, CPTR_EL2, and CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

```plaintext
| [31 30 29 28 | 27 26 25 24 | 23 22 21 20] | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 5 | 4 | 0 |
|---------------|---------------|-----------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | size | 1 | Rm | 1 | 0 | 1 | 1 | 0 | 1 | Rn | Rd |

Encoding

SQRDMULH <V><d>, <V><n>, <V><m>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');
```

**Vector**

```plaintext
| [31 30 29 28 | 27 26 25 24 | 23 22 21 20] | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 5 | 4 | 0 |
|---------------|---------------|-----------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm | 1 | 0 | 1 | 1 | 0 | 1 | Rn | Rd |

Encoding

SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');
```
Assembler symbols

\(<v>\)  Is a width specifier, encoded in the "size" field. It can have the following values:

- **H** when size = 01
- **S** when size = 10

The following encodings are reserved:

- size = 00.
- size = 11.

\(<d>\)  Is the number of the SIMD&FP destination register, in the "Rd" field.

\(<n>\)  Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<m>\)  Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

\(<Vd>\)  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\)  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- **4H** when size = 01, Q = 0
- **8H** when size = 01, Q = 1
- **2S** when size = 10, Q = 0
- **4S** when size = 10, Q = 1

The following encodings are reserved:

- size = 00, Q = x.
- size = 11, Q = x.

\(<Vn>\)  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vm>\)  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.299  SQRSHL

Signed saturating Rounding Shift Left (register). This instruction takes each vector element in the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see SQSHL (register).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

Encoding

SQRSHL <V><d>, <V><n>, <V><m>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;

Vector

Encoding

SQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

**Assembler symbols**

\(<V>\) Is a width specifier, encoded in the "size" field. It can have the following values:

- B when \(size = 00\)
- H when \(size = 01\)
- S when \(size = 10\)
- D when \(size = 11\)

\(<d>\) Is the number of the SIMD&FP destination register, in the "Rd" field.

\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<m>\) Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when \(size = 00, Q = 0\)
- 16B when \(size = 00, Q = 1\)
- 4H when \(size = 01, Q = 0\)
- 8H when \(size = 01, Q = 1\)
- 2S when \(size = 10, Q = 0\)
- 4S when \(size = 10, Q = 1\)
- 2D when \(size = 11, Q = 1\)

The encoding \(size = 11, Q = 0\) is reserved.

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

`CheckFPAdvSIMDEnabled64();`
`bits(datasize) operand1 = V[n, datasize];`
`bits(datasize) operand2 = V[m, datasize];`
`bits(datasize) result;`
`.boolean sat;`

for e = 0 to elements-1
  integer element = Int(Elem[operand1, e, esize], unsigned);
  integer shift = SInt(Elem[operand2, e, esize]<7:0>);
  if shift >= 0 then // left shift
    element = element << shift;
  else // right shift
    shift = -shift;
    if rounding then
      element = (element + (1 << (shift - 1))) >> shift;
    else
      element = element >> shift;
  
  if saturating then
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
  if sat then FPSR.QC = '1';
  else
    Elem[result, e, esize] = element<esize-1:0>;`
V[d, datasize] = result;
C7.2.300 SQRSHRN, SQRSHRN2

Signed saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see SQRSHRN, SQRSHRN2.

The SQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

```
<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22]</th>
<th>[19 18]</th>
<th>[16 15 14 13 12]</th>
<th>[11 10 9]</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
<td>!=0000</td>
<td>immb 1 0 0 1 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

SQRSHRN <Vb><d>, <Va><n>, #<shift>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Vector

```
<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22]</th>
<th>[19 18]</th>
<th>[16 15 14 13 12]</th>
<th>[11 10 9]</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 1 1 1 1 0</td>
<td>!=0000</td>
<td>immb 1 0 0 1 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

SQRSHRN[2] <Vb><Tb>, <Vn><Ta>, #<shift>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
```
if immh[3] == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
[absent] when Q = 0
[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:
8B when immh = 0001, Q = 0
16B when immh = 0001, Q = 1
4H when immh = 001x, Q = 0
8H when immh = 001x, Q = 1
2S when immh = 01xx, Q = 0
4S when immh = 01xx, Q = 1

See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.
The encoding immh = 1xxx, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Ta> Is an arrangement specifier, encoded in the "immh" field. It can have the following values:
8H when immh = 0001
4S when immh = 001x
2D when immh = 01xx

See Advanced SIMD modified immediate on page C4-632 when immh = 0000.
The encoding immh = 1xxx is reserved.

<Vb> Is the destination width specifier, encoded in the "immh" field. It can have the following values:
B when immh = 0001
H when immh = 001x
S when immh = 01xx

The following encodings are reserved:
• immh = 0000.
• immh = 1xxx.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<Va> Is the source width specifier, encoded in the "immh" field. It can have the following values:
H when immh = 0001
S when immh = 001x
D when immh = 01xx
The following encodings are reserved:
- \( \text{immh} = 0000 \).
- \( \text{immh} = 1xxx \).

\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<\text{shift}>\) For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:
- \( (16-\text{UInt}(\text{immh}:\text{immb})) \) when \( \text{immh} = 0001 \)
- \( (32-\text{UInt}(\text{immh}:\text{immb})) \) when \( \text{immh} = 001x \)
- \( (64-\text{UInt}(\text{immh}:\text{immb})) \) when \( \text{immh} = 01xx \)

The following encodings are reserved:
- \( \text{immh} = 0000 \).
- \( \text{immh} = 1xxx \).

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:
- \( (16-\text{UInt}(\text{immh}:\text{immb})) \) when \( \text{immh} = 0001 \)
- \( (32-\text{UInt}(\text{immh}:\text{immb})) \) when \( \text{immh} = 001x \)
- \( (64-\text{UInt}(\text{immh}:\text{immb})) \) when \( \text{immh} = 01xx \)

See Advanced SIMD modified immediate on page C4-632 when \( \text{immh} = 0000 \).

The encoding \( \text{immh} = 1xxx \) is reserved.

Operation for all encodings

```c
CheckFPAdvsIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bias(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';
Vpart[d, part, datasize] = result;
```

C7.2.301   SQRSRSHRUN, SQRSRSHRUN2

Signed saturating Rounded Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are rounded. For truncated results, see SQRSRSHRUN, SQRSRSHRUN2.

The SQRSRSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSRSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

```
[31 30 29 28] [27 26 25 24] [23 22] [19 18] [16 15 14 13] [12 11 10 9] [ 5  4] [ 0 ]
 0 1 1 1 1 1 1 0 !=0000  immbh  1 0 0 0 1 1  Rd
   immh   op    Rn
```

**Encoding**

SQRSRSHRUN <Vb><d>, <Va><n>, #<shift>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Vector

```
[31 30 29 28] [27 26 25 24] [23 22] [19 18] [16 15 14 13] [12 11 10 9] [ 5  4] [ 0 ]
 0 1 0 1 1 1 1 0 !=0000  immbh  1 0 0 0 1 1  Rd
   immh   op    Rn
```

**Encoding**

SQRSRSHRUN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then "Advanced SIMD modified immediate";
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
```
integer datysize = 64;
integer part = UInt(Q);
integer elements = datysize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');

**Assembler symbols**

2  
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
- [absent] when Q = 0
- [present] when Q = 1

<Vd>  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb>  
Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:
- 8B when immh = 0001, Q = 0
- 16B when immh = 0001, Q = 1
- 4H when immh = 001x, Q = 0
- 8H when immh = 001x, Q = 1
- 2S when immh = 01xx, Q = 0
- 4S when immh = 01xx, Q = 1

See *Advanced SIMD modified immediate on page C4-632* when immh = 0000, Q = x.
The encoding immh = 1xxx, Q = x is reserved.

<Vn>  
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta>  
Is an arrangement specifier, encoded in the "immh" field. It can have the following values:
- 8H when immh = 0001
- 4S when immh = 001x
- 2D when immh = 01xx

See *Advanced SIMD modified immediate on page C4-632* when immh = 0000.
The encoding immh = 1xxx is reserved.

<Vb>  
Is the destination width specifier, encoded in the "immh" field. It can have the following values:
- B when immh = 0001
- H when immh = 001x
- S when immh = 01xx

The following encodings are reserved:
- immh = 0000.
- immh = 1xxx.

<d>  
Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va>  
Is the source width specifier, encoded in the "immh" field. It can have the following values:
- H when immh = 0001
- S when immh = 001x
- D when immh = 01xx
The following encodings are reserved:
  • \( \text{immh} = 0000 \).
  • \( \text{immh} = 1xxx \).

\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<\text{shift}>\) For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:
  - \((16-\text{UInt}(\text{immh:immb}))\) when \( \text{immh} = 0001 \)
  - \((32-\text{UInt}(\text{immh:immb}))\) when \( \text{immh} = 001x \)
  - \((64-\text{UInt}(\text{immh:immb}))\) when \( \text{immh} = 01xx \)

The following encodings are reserved:
  • \( \text{immh} = 0000 \).
  • \( \text{immh} = 1xxx \).

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:
  - \((16-\text{UInt}(\text{immh:immb}))\) when \( \text{immh} = 0001 \)
  - \((32-\text{UInt}(\text{immh:immb}))\) when \( \text{immh} = 001x \)
  - \((64-\text{UInt}(\text{immh:immb}))\) when \( \text{immh} = 01xx \)

See Advanced SIMD modified immediate on page C4-632 when \( \text{immh} = 0000 \).

The encoding \( \text{immh} = 1xxx \) is reserved.

Operation for all encodings

\[
\text{CheckFPAdvSIMDEnabled64}();
\text{bits}(\text{datasize} \times 2) \ \text{operand} = V[n, \ \text{datasize} \times 2];
\text{bits}(\text{datasize}) \ \text{result};
\text{integer} \ \text{round\_const} = \text{if round then} (1 \ll (\text{shift} - 1)) \ \text{else} \ 0;
\text{integer} \ \text{element};
\text{boolean} \ \text{sat};
\]

\[
\text{for} \ e = 0 \ \text{to elements}-1
\]

\[
\text{element} = (\text{SInt}(\text{Elem}[\text{operand}, e, 2 \times \text{esize}]) + \text{round\_const}) \ll \text{shift};
(\text{Elem}[\text{result}, e, \text{esize}, \text{sat}]) = \text{UnsignedSatQ}(\text{element}, \text{esize});
\text{if sat then FPSR.QC} = '1';
\]

\[
\text{Vpart}[d, \text{part}, \text{datasize}] = \text{result};
\]
C7.2.302   SQSHL (immediate)

Signed saturating Shift Left (immediate). This instruction reads each vector element in the source SIMD&FP register, shifts each result by an immediate value, places the final result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

Encoding

SQSHL <V><d>, <V><n>, #<shift>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;

Vector

Encoding

SQSHL <Vd>.<T>, <Vn>.<T>, #<shift>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh[3:Q] == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;

case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
end;

**Assembler symbols**

<v> Is a width specifier, encoded in the "immh" field. It can have the following values:

- B  when immh = 0001
- H  when immh = 001x
- S  when immh = 01xx
- D  when immh = 1xxx

The encoding immh = 0000 is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

- 8B  when immh = 0001, Q = 0
- 16B when immh = 0001, Q = 1
- 4H  when immh = 001x, Q = 0
- 8H  when immh = 001x, Q = 1
- 2S  when immh = 01xx, Q = 0
- 4S  when immh = 01xx, Q = 1
- 2D  when immh = 1xxx, Q = 1

See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

( UInt(immh:immb) - 8 ) when immh = 0001
( UInt(immh:immb) - 16) when immh = 001x
( UInt(immh:immb) - 32) when immh = 01xx
( UInt(immh:immb) - 64) when immh = 1xxx

The encoding immh = 0000 is reserved.

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

( UInt(immh:immb) - 8 ) when immh = 0001
( UInt(immh:immb) - 16) when immh = 001x
( UInt(immh:immb) - 32) when immh = 01xx
(UInt(immh:immb)-64) when immh = 1xxx
See *Advanced SIMD modified immediate* on page C4-632 when immh = 0000.

**Operation for all encodings**

```
CheckFPAdvSIMDEnabled64();
bitsu(data) operand = V[n, datasize];
bitsu(data) result;
integer element;
boolean sat;

for e = 0 to elements-1
   element = Int(Elem[operand, e, esize], src_unsigned) << shift;
   (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
   if sat then FPSR.QC = '1';

V[d, datasize] = result;
```
C7.2.303   SQSHL (register)

Signed saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see SQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

| [31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 0 1 1 1 0 | size | 1 | Rm | 0 1 0 | 1 1 | Rn | Rd |
| U | R | S |

Encoding

SQSHL <V><d>, <V><n>, <V><m>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;

Vector

| [31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|----------------|-----------------|-----------------|-----------------|-----------------|
| 0 | Q | 0 | 0 1 1 1 0 | size | 1 | Rm | 0 1 0 | 1 1 | Rn | Rd |
| U | R | S |

Encoding

SQSHL <V>d.<T>, <V>n.<T>, <V>m.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

Assembler symbols

<\> Is a width specifier, encoded in the "size" field. It can have the following values:
   B when size = 00
   H when size = 01
   S when size = 10
   D when size = 11
<\d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<\n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<\m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<\Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<\T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
   8B when size = 00, Q = 0
   16B when size = 00, Q = 1
   4H when size = 01, Q = 0
   8H when size = 01, Q = 1
   2S when size = 10, Q = 0
   4S when size = 10, Q = 1
   2D when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.
<\Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<\Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
boolean sat;
for e = 0 to elements-1
   integer element = Int(Elem[operand1, e, esize], unsigned);
   integer shift = SInt(Elem[operand2, e, esize]<7:0>);
   if shift >= 0 then // left shift
      element = element << shift;
   else // right shift
      shift = -shift;
      if rounding then
         element = (element + (1 << (shift - 1))) >> shift;
      else
         element = element >> shift;
   if saturating then
      (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
   if sat then FPSR.QC = '1';
   else
      Elem[result, e, esize] = element<esize-1:0>;
\[ V[d, d\text{datasize}] = \text{result}; \]
### C7.2.304 SQSHLU

Signed saturating Shift Left Unsigned (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, shifts each value by an immediate value, saturates the shifted result to an unsigned integer value, places the result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

#### Scalar

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22</th>
<th>19 18</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Encoding

SQSHLU `<V><d>, <Vn><n>, #<shift>`

#### Decode for this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UNDEFINED;
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
endcase
```

#### Vector

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22</th>
<th>19 18</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Encoding

SQSHLU `<Vd><T>, <Vn><T>, #<shift>`

#### Decode for this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
```
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;

case op:U of
    when '00' UNDEFINED;
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;

Assembler symbols

<\V> Is a width specifier, encoded in the "immmh" field. It can have the following values:
    B when immh = 0001
    H when immh = 001x
    S when immh = 01xx
    D when immh = 1xxx

The encoding immh = 0000 is reserved.

<\d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<\n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<\Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<\T> Is an arrangement specifier, encoded in the "immmh:Q" field. It can have the following values:
    8B when immh = 0001, Q = 0
    16B when immh = 0001, Q = 1
    4H when immh = 001x, Q = 0
    8H when immh = 001x, Q = 1
    2S when immh = 01xx, Q = 0
    4S when immh = 01xx, Q = 1
    2D when immh = 1xxx, Q = 1

See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<\Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in the "immmh:immb" field. It can have the following values:
    (UInt(immh:immb)-8) when immh = 0001
    (UInt(immh:immb)-16) when immh = 001x
    (UInt(immh:immb)-32) when immh = 01xx
    (UInt(immh:immb)-64) when immh = 1xxx

The encoding immh = 0000 is reserved.

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in the "immmh:immb" field. It can have the following values:
    (UInt(immh:immb)-8) when immh = 0001
    (UInt(immh:immb)-16) when immh = 001x
    (UInt(immh:immb)-32) when immh = 01xx
(UInt(immh:immb)-64) when immh = 1xxx

See Advanced SIMD modified immediate on page C4-632 when immh = 0000.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
**C7.2.305 SQSHRN, SQSHRN2**

Signed saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts and truncates each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. For rounded results, see SQRSRN, SQRSRN2.

The SQSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Scalar

**Encoding**

SQSHRN <Vb><d>, <Va><n>, #<shift>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);

if imm == '0000' then UNDEFINED;
if imm<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');

### Vector

**Encoding**


**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);

if imm == '0000' then SEE "Advanced SIMD modified immediate";
if imm<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');

## Assembler symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Td>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb>

Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

- 8B when immh = 0001, Q = 0
- 16B when immh = 0001, Q = 1
- 4H when immh = 001x, Q = 0
- 8H when immh = 001x, Q = 1
- 2S when immh = 01xx, Q = 0
- 4S when immh = 01xx, Q = 1

See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = x is reserved.

<Vn>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta>

Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

- 8H when immh = 0001
- 4S when immh = 001x
- 2D when immh = 01xx

See Advanced SIMD modified immediate on page C4-632 when immh = 0000.

The encoding immh = 1xxx is reserved.

<Vb>

Is the destination width specifier, encoded in the "immh" field. It can have the following values:

- 8 when immh = 0001
- H when immh = 001x
- S when immh = 01xx

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.

<d>

Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va>

Is the source width specifier, encoded in the "immh" field. It can have the following values:

- H when immh = 0001
- S when immh = 001x
- D when immh = 01xx
The following encodings are reserved:
- $\text{immh} = 0000$
- $\text{immh} = 1xxx$

$n$ is the number of the first SIMD&FP source register, encoded in the "Rn" field.

$\text{<shift>}$ For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:
- $(16-\text{UInt}(\text{immh:immb}))$ when $\text{immh} = 0001$
- $(32-\text{UInt}(\text{immh:immb}))$ when $\text{immh} = 001x$
- $(64-\text{UInt}(\text{immh:immb}))$ when $\text{immh} = 01xx$

The following encodings are reserved:
- $\text{immh} = 0000$
- $\text{immh} = 1xxx$

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:
- $(16-\text{UInt}(\text{immh:immb}))$ when $\text{immh} = 0001$
- $(32-\text{UInt}(\text{immh:immb}))$ when $\text{immh} = 001x$
- $(64-\text{UInt}(\text{immh:immb}))$ when $\text{immh} = 01xx$

See Advanced SIMD modified immediate on page C4-632 when $\text{immh} = 0000$.
The encoding $\text{immh} = 1xxx$ is reserved.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bists(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';
Vpart[d, part, datasize] = result;
```
C7.2.306  SQSHRUN, SQSHRUN2

Signed saturating Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see SQRSHRUN, SQRSHRUN2.

The SQSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22]</th>
<th>[19 18]</th>
<th>[16 15 14 13]</th>
<th>[12 11 10 9]</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 1 0</td>
<td>!=0000</td>
<td>immh:0 0 0 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

SQSHRUN <Vb><d>, <Va><n>, #<shift>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

**Vector**

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22]</th>
<th>[19 18]</th>
<th>[16 15 14 13]</th>
<th>[12 11 10 9]</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
<td>!=0000</td>
<td>immh:0 0 0 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

SQSHRUN[2] <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
```
\[
\text{integer part} = \text{UInt}(Q) ;
\text{integer elements} = \text{datasize} \div \text{esize} ;
\text{integer shift} = (2 \times \text{esize}) - \text{UInt(immh:immb)} ;
\text{boolean round} = (op == \text{'}1\text{'}) ;
\]

**Assembler symbols**

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when \( Q = 0 \)
- [present] when \( Q = 1 \)

\(<Vd>\)

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Tb>\)

Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

- \( 8H \) when \( \text{immh} = 0001, Q = 0 \)
- \( 16B \) when \( \text{immh} = 0001, Q = 1 \)
- \( 4H \) when \( \text{immh} = 001x, Q = 0 \)
- \( 8H \) when \( \text{immh} = 001x, Q = 1 \)
- \( 2S \) when \( \text{immh} = 01xx, Q = 0 \)
- \( 4S \) when \( \text{immh} = 01xx, Q = 1 \)

See *Advanced SIMD modified immediate on page C4-632* when \( \text{immh} = 0000, Q = x \).

The encoding \( \text{immh} = 1xxx, Q = x \) is reserved.

\(<Va>\)

Is the source width specifier, encoded in the "immh" field. It can have the following values:

- \( H \) when \( \text{immh} = 0001 \)
- \( S \) when \( \text{immh} = 001x \)
- \( D \) when \( \text{immh} = 01xx \)

The following encodings are reserved:

- \( \text{immh} = 0000 \)
- \( \text{immh} = 1xxx \).

\(<d>\)

Is the number of the SIMD&FP destination register, in the "Rd" field.

\(<Vb>\)

Is the destination width specifier, encoded in the "immh" field. It can have the following values:

- \( D \) when \( \text{immh} = 0001 \)
- \( S \) when \( \text{immh} = 001x \)
- \( D \) when \( \text{immh} = 01xx \)

The following encodings are reserved:

- \( \text{immh} = 0000 \).
• \( \text{immh} = 1xxx \).

\(<\text{n}>\)

Is the number of the first SIMD\&FP source register, encoded in the "Rn" field.

\(<\text{shift}>\)

For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:

- \((16-\text{UInt}(\text{immh:immb}))\) when \( \text{immh} = 0001 \)
- \((32-\text{UInt}(\text{immh:immb}))\) when \( \text{immh} = 001x \)
- \((64-\text{UInt}(\text{immh:immb}))\) when \( \text{immh} = 01xx \)

The following encodings are reserved:

- \( \text{immh} = 0000 \).
- \( \text{immh} = 1xxx \).

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

- \((16-\text{UInt}(\text{immh:immb}))\) when \( \text{immh} = 0001 \)
- \((32-\text{UInt}(\text{immh:immb}))\) when \( \text{immh} = 001x \)
- \((64-\text{UInt}(\text{immh:immb}))\) when \( \text{immh} = 01xx \)

See Advanced SIMD modified immediate on page C4-632 when \( \text{immh} = 0000 \).

The encoding \( \text{immh} = 1xxx \) is reserved.

### Operation for all encodings

```assembly
CheckFPAdvSIMDEnabled64();
bias(datasize\times2) \text{operand} = V[n, datasize\times2];
bias(datasize) result;
integer round_const = if round then \((1 \ll (\text{shift} - 1))\) else 0;
integer element;
boolean sat;
for e = 0 to elements -1
  element = (SInt(Elem[\text{operand}, e, 2*esize]) + round_const) >> shift;
  (Elem[\text{result}, e, esize], sat) = UnsignedSatQ(element, esize);
  if sat then FPSR.QC = '1';
Vpart[d, part, datasize] = result;
```

```
C7.2.307   SQSUB

Signed saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 0</td>
<td>size</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rm</td>
<td>0 0 1 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

SQSUB <V><d>, <V><n>, <V><m>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

**Vector**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>Q</td>
<td>0 1 1 1 0</td>
<td>size</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rm</td>
<td>0 0 1 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

SQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

**Assembler symbols**

<V>  

Is a width specifier, encoded in the "size" field. It can have the following values:

- 8 when size = 00
H when size = 01
S when size = 10
D when size = 11

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  diff = element1 - element2;
  (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
  if sat then FPSR.QC = '1';
V[d, datasize] = result;
```
C7.2.308   SQXTN, SQXTN2

Signed saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates the value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQXTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQXTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

Encoding

SQXTN <Vb><d>, <Va><n>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');

Vector

Encoding

SQXTN{2} <Vd>.<Tb>, <Vn>.<Ta>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler symbols

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0
[present] when Q = 1

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb>
Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn>
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta>
Is an arrangement specifier, encoded in the "size" field. It can have the following values:

8H when size = 00
4S when size = 01
2D when size = 10

The encoding size = 11 is reserved.

<Vb>
Is the destination width specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10

The encoding size = 11 is reserved.

<d>
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va>
Is the source width specifier, encoded in the "size" field. It can have the following values:

H when size = 00
S when size = 01
D when size = 10

The encoding size = 11 is reserved.

<n>
Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
b bits(2*datasize) operand = V[n, 2*datasize];
b bits(datasize) result;
b bits(2*esize) element;
boolean sat;
for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
C7.2.309 SQXTUN, SQXTUN2

Signed saturating extract Unsigned Narrow. This instruction reads each signed integer value in the vector of the source SIMD&FP register, saturates the value to an unsigned integer value that is half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQXTUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQXTUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1</td>
<td>1 1 1 0</td>
<td>size</td>
<td>1 0 0 0 1 1 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SQXTUN <Vb><d>, <Va><n>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

Vector

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>0 1 1 1 0</td>
<td>size</td>
<td>1 0 0 0 1 1 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SQXTUN2 <Vd>.<Tb>, <Vn>.<Ta>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
Assembler symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb>

Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta>

Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vb>

Is the destination width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10

The encoding size = 11 is reserved.

<d>

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va>

Is the source width specifier, encoded in the "size" field. It can have the following values:

- H when size = 00
- S when size = 01
- D when size = 10

The encoding size = 11 is reserved.

<n>

Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
  element = Elem[operand, e, 2*esize];
  (Elem[result, e, esize], sat) = UnsignedSatQ(SInt(element), esize);
  if sat then FPSR.QC = '1';
Vpart[d, part, datasize] = result;
C7.2.310  SRHADD

Signed Rounding Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see SHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5  4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

SRHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

**Assembler symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```python
CheckFPAdvSIMDEnabled64();
bis(datasize) operand1 = V[n, datasize];
bis(datasize) operand2 = V[m, datasize];
bis(datasize) result;
integer element1;
integer element2;
```
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = (element1 + element2 + 1) >> 1;
    Elem[result, e, esize] = sum<esize-1:0>;

V[d, datasize] = result;
C7.2.311 SRI

Shift Right and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the right of each vector element of the source register are lost.

The following figure shows the operation of shift right by 3 for an 8-bit vector element.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>19 18 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 1 1 0</td>
<td>!0000</td>
<td>immb</td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SRI <V><d>, <V><n>, #<shift>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
```

Vector

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>19 18 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 1 0 1 1 1 1 1 0</td>
<td>!0000</td>
<td>immb</td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SRI <Vd>.<T>, <Vn>.<T>, #<shift>
Decode for this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer shift = (esize * 2) - UInt(immh:immb);
```

Assembler symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;V&gt;</td>
<td>Is a width specifier, encoded in the &quot;immh&quot; field. It can have the following values:</td>
</tr>
<tr>
<td>D</td>
<td>when immh = 1xxx</td>
</tr>
<tr>
<td></td>
<td>8B</td>
</tr>
<tr>
<td></td>
<td>16B</td>
</tr>
<tr>
<td></td>
<td>4H</td>
</tr>
<tr>
<td></td>
<td>8H</td>
</tr>
<tr>
<td></td>
<td>2S</td>
</tr>
<tr>
<td></td>
<td>4S</td>
</tr>
<tr>
<td></td>
<td>2D</td>
</tr>
</tbody>
</table>

See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x. The encoding immh = 1xxx, Q = x is reserved.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Vn&gt;</td>
<td>Is the name of the SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
</tbody>
</table>

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

- (128-UInt(immh:immb)) when immh = 1xxx

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

- (16-UInt(immh:immb)) when immh = 0001
- (32-UInt(immh:immb)) when immh = 001x
- (64-UInt(immh:immb)) when immh = 01xx
- (128-UInt(immh:immb)) when immh = 1xxx

See Advanced SIMD modified immediate on page C4-632 when immh = 0000.

Operation for all encodings

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2 = V[d, datasize];
```
bits(datasize) result;
bits(esize) mask = LSR(Ones(esize), shift);
bits(esize) shifted;

for e = 0 to elements-1
    shifted = LSR(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
    V[d, datasize] = result;

Operational information
If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.312 SRSHL

Signed Rounding Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see SSSH.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>0 1 0 1</td>
<td>0 1</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>
```

Encoding

SRSHL <V><d>, <V><n>, <V><m>

Decode for this encoding

- `integer d = UInt(Rd);`
- `integer n = UInt(Rn);`
- `integer m = UInt(Rm);`
- `integer esize = 8 << UInt(size);`
- `integer datasize = esize;`
- `integer elements = 1;`
- `boolean unsigned = (U == '1');`
- `boolean rounding = (R == '1');`
- `boolean saturating = (S == '1');`
- `if S == '0' && size != '11' then UNDEFINED;`

Vector

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1 1 1 0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>0 1 0 1</td>
</tr>
</tbody>
</table>
```

Encoding

SRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

- `integer d = UInt(Rd);`
- `integer n = UInt(Rn);`
- `integer m = UInt(Rm);`
- `if size:Q == '110' then UNDEFINED;`
- `integer esize = 8 << UInt(size);`
- `integer datasize = if Q == '1' then 128 else 64;`
- `integer elements = datasize DIV esize;`
- `boolean unsigned = (U == '1');`
- `boolean rounding = (R == '1');`
- `boolean saturating = (S == '1');`
Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- 0  when size = 11
The following encodings are reserved:
- size = 0x.
- size = 10.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B  when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H  when size = 01, Q = 0
- 8H  when size = 01, Q = 1
- 2S  when size = 10, Q = 0
- 4S  when size = 10, Q = 1
- 2D  when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        if rounding then
            element = (element + (1 << (shift - 1))) >> shift;
        else
            element = element >> shift;

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;
    V[d, datasize] = result;
C7.2.313  SRSHR

Signed Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see SSHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22</th>
<th>19 18 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>!=0000</td>
<td>immh</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**Encoding**

SRSHR <V><d>, <V><n>, #<shift>

**Decode for this encoding**

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

**Vector**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22</th>
<th>19 18 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>!=0000</td>
<td>immh</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**Encoding**

SRSHR <Vd>, <T>, <Vn>, <T>, #<shift>

**Decode for this encoding**

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
```
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler symbols

\(<V>\) Is a width specifier, encoded in the "immh" field. It can have the following values:
\(D\) when \(\text{immh} = 1xxx\)
The encoding \(\text{immh} = 0xxx\) is reserved.
\(<d>\) Is the number of the SIMD&FP destination register, in the "Rd" field.
\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<T>\) Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:
\(8B\) when \(\text{immh} = 0001, Q = 0\)
\(16B\) when \(\text{immh} = 0001, Q = 1\)
\(4H\) when \(\text{immh} = 001x, Q = 0\)
\(8H\) when \(\text{immh} = 001x, Q = 1\)
\(2S\) when \(\text{immh} = 01xx, Q = 0\)
\(4S\) when \(\text{immh} = 01xx, Q = 1\)
\(2D\) when \(\text{immh} = 1xxx, Q = 1\)
See Advanced SIMD modified immediate on page C4-632 when \(\text{immh} = 0000, Q = x\).
The encoding \(\text{immh} = 1xxx, Q = 0\) is reserved.
\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.
\(<shift>\) For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:
\((128-\text{UInt}(\text{immh:immb}))\) when \(\text{immh} = 1xxx\)
The encoding \(\text{immh} = 0xxx\) is reserved.
For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:
\((16-\text{UInt}(\text{immh:immb}))\) when \(\text{immh} = 0001\)
\((32-\text{UInt}(\text{immh:immb}))\) when \(\text{immh} = 001x\)
\((64-\text{UInt}(\text{immh:immb}))\) when \(\text{immh} = 01xx\)
\((128-\text{UInt}(\text{immh:immb}))\) when \(\text{immh} = 1xxx\)
See Advanced SIMD modified immediate on page C4-632 when \(\text{immh} = 0000\).

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then \((1 \ll (\text{shift} - 1))\) else 0;
integer element;
operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
   element = (\(\text{Int}(\text{Elem}[\text{operand}, e, \text{esize}], \text{unsigned}) + \text{round\_const}\)) \ll \text{shift};
   Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d, datasize] = result;
C7.2.314   SRSRA

Signed Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see SSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>19 18 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 1 0 !=0000 immb</td>
<td>0 0 1 1 0 1</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Encoding

SRSRA <V><d>, <V><n>, #<shift>

Decode for this encoding

```plaintext
ingter d = UInt(Rd);
ingter n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
ingter esize = 8 << 3;
ingenter datasize = esize;
ingenter elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolea unsigned = (U == '1');
genre round = (o1 == '1');
genre accumulate = (o0 == '1');
```

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>19 18 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 1 1 1 1 0 !=0000 immb</td>
<td>0 0 1 1 0 1</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Encoding

SRSRA <Vd>.<T>, <Vn>.<T>, #<shift>

Decode for this encoding

```plaintext
ingter d = UInt(Rd);
ingter n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
ingenter esize = 8 << HighestSetBit(immh);
genre datasize = if Q == '1' then 128 else 64;
genre elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
genre unsigned = (U == '1');
```
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler symbols

<V>  Is a width specifier, encoded in the "immh" field. It can have the following values:
  D  when immh = 1xxx
The encoding immh = 0xxx is reserved.
<\d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n>  Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>  Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:
  8B  when immh = 0001, Q = 0
  16B when immh = 0001, Q = 1
  4H  when immh = 001x, Q = 0
  8H  when immh = 001x, Q = 1
  2S  when immh = 01xx, Q = 0
  4S  when immh = 01xx, Q = 1
  2D  when immh = 1xxx, Q = 1
See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.
The encoding immh = 1xxx, Q = 0 is reserved.
<Vn>  Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<\shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:
  (128-UInt(immh:immb)) when immh = 1xxx
The encoding immh = 0xxx is reserved.
For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:
  (16-UInt(immh:immb)) when immh = 0001
  (32-UInt(immh:immb)) when immh = 001x
  (64-UInt(immh:immb)) when immh = 01xx
  (128-UInt(immh:immb)) when immh = 1xxx
See Advanced SIMD modified immediate on page C4-632 when immh = 0000.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
broadcast operands = V[n, datasize];
broadcast operand2;
broadcast datasize result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements - 1
  element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
  Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;


V[d, datasize] = result;
C7.2.315 SSHL

Signed Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts each value by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see SRSHL.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

```
[31| 30| 29|28|27| 26| 25|24|23| 22| 21| 20| 19| 18| 17| 16|15|14|13|12| 11| 10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0]
0   1   0   1   1   1   0   | size  1    | Rm  0   1   0   0   0   1    | Rn  | Rd |
 U   R   S
```

**Encoding**

SSHL <V><d>, <V><n>, <V><m>

**Decode for this encoding**

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

**Vector**

```
[31| 30| 29|28|27| 26| 25|24|23| 22| 21| 20| 19| 18| 17| 16|15|14|13|12| 11| 10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0]
0   1   0   0   1   1   1   0   | size  1    | Rm  0   1   0   0   0   1    | Rn  | Rd |
 U   R   S
```

**Encoding**

SSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```
Assembler symbols

<\V>  Is a width specifier, encoded in the "size" field. It can have the following values:
   0 when size = 11
   The following encodings are reserved:
      * size = 0x.
      * size = 10.

<d>  Is the number of the SIMD&FP destination register, in the "Rd" field.

<n>  Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m>  Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
   8B when size = 00, Q = 0
   16B when size = 00, Q = 1
   4H when size = 01, Q = 0
   8H when size = 01, Q = 1
   2S when size = 10, Q = 0
   4S when size = 10, Q = 1
   2D when size = 11, Q = 1
   The encoding size = 11, Q = 0 is reserved.

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
   integer element = Int(Elem[operand1, e, esize], unsigned);
   integer shift = SInt(Elem[operand2, e, esize]<7:0>);
   if shift >= 0 then // left shift
      element = element << shift;
   else // right shift
      shift = -shift;
   if rounding then
      element = (element + (1 << (shift - 1))) >> shift;
   else
      element = element >> shift;

   if saturating then
      (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
   if sat then FPSR.QC = '1';
   else
      Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.316 SSHLL, SSHLL2

Signed Shift Left Long (immediate). This instruction reads each vector element from the source SIMD&FP register, left shifts each vector element by the specified shift amount, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SSHLL instruction extracts vector elements from the lower half of the source register. The SSHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias SXTL, SXTL2. See Alias conditions for details of when each alias is preferred.

Encoding

\[
\begin{array}{cccccccccccccccc}
\text{SSHLL}\{2\} & \text{<Vd>}.& \text{<Ta>}, & \text{<Vn>}.& \text{<Tb>}, & \#<\text{shift}>
\end{array}
\]

Decode for this encoding

\[
\begin{align*}
\text{integer } d &= \text{UInt(Rd)}; \\
\text{integer } n &= \text{UInt(Rn)}; \\
\text{if } \text{immh} &= '0000' \text{ then SEE "Advanced SIMD modified immediate";} \\
\text{if } \text{immh}<3> &= '1' \text{ then UNDEFINED;} \\
\text{integer } esize &= 8 \times \text{HighestSetBit}(\text{immh}); \\
\text{integer } datasize &= 64; \\
\text{integer } part &= \text{UInt}(Q); \\
\text{integer } elements &= \text{datasize} \div \text{esize}; \\
\text{integer } shift &= \text{UInt(immh:immb)} - \text{esize}; \\
\text{boolean } unsigned &= (U == '1'); \\
\end{align*}
\]

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>SXTL, SXTL2</td>
<td>immmb = '000' &amp;&amp; BitCount(immh) = 1</td>
</tr>
</tbody>
</table>

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when \( Q = 0 \)

[present] when \( Q = 1 \)

\(<\text{Vd}>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<\text{Ta}>\) Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

8H when \( \text{immb} = 0001 \)
4S when immh = 001x  
2D when immh = 01xx  
See Advanced SIMD modified immediate on page C4-632 when imm = 0000.
The encoding immh = 1xxx is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

- 8B when immh = 0001, Q = 0
- 16B when immh = 0001, Q = 1
- 4H when immh = 001x, Q = 0
- 8H when immh = 001x, Q = 1
- 2S when immh = 01xx, Q = 0
- 4S when immh = 01xx, Q = 1

See Advanced SIMD modified immediate on page C4-632 when imm = 0000, Q = x.
The encoding immh = 1xxx, Q = x is reserved.

<shift> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in the
"immh:immb" field. It can have the following values:

- (UInt(immh:immb)-8) when immh = 0001
- (UInt(immh:immb)-16) when immh = 001x
- (UInt(immh:immb)-32) when immh = 01xx

See Advanced SIMD modified immediate on page C4-632 when imm = 0000.
The encoding immh = 1xxx is reserved.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part, datasize];
bits(datasize*2) result;
integer element;
for e = 0 to elements-1  
  element = Int(Elem[operand, e, esize], unsigned) << shift;
  Elem[result, e, 2*esize] = element<2*esize-1:0>;
V[d, datasize*2] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.317 SSHR

Signed Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see SRSHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>19 18 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0 !=0000 immh 0 0 0 0 0 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

SSHR <V><d>, <V><n>, #<shift>

**Decode for this encoding**

1. integer d = UInt(Rd);
2. integer n = UInt(Rn);
3. if immh<3> != '1' then UNDEFINED;
4. integer esize = 8 << 3;
5. integer datasize = esize;
6. integer elements = 1;
7. integer shift = (esize * 2) - UInt(immh:immb);
8. boolean unsigned = (U == '1');
9. boolean round = (o1 == '1');
10. boolean accumulate = (o0 == '1');

**Vector**

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>19 18 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 1 0 !=0000 immh 0 0 0 0 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

SSHR <Vd>.<T>, <Vn>.<T>, #<shift>

**Decode for this encoding**

1. integer d = UInt(Rd);
2. integer n = UInt(Rn);
3. if immh == '0000' then SEE "Advanced SIMD modified immediate";
4. if immh<3>:Q == '10' then UNDEFINED;
5. integer esize = 8 << HighestSetBit(immh);
6. integer datasize = if Q == '1' then 128 else 64;
7. integer elements = datasize DIV esize;
8. integer shift = (esize * 2) - UInt(immh:immb);
9. boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

**Assembler symbols**

<\V> Is a width specifier, encoded in the "immh" field. It can have the following values:

\begin{itemize}
\item \textbf{D} when immh = 1xxx
\item \textbf{0} when immh = 0xxx
\end{itemize}

The encoding immh = 0000 is reserved.

<\d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<\n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<\Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

\begin{itemize}
\item \textbf{8B} when immh = 0001, Q = 0
\item \textbf{16B} when immh = 0001, Q = 1
\item \textbf{4H} when immh = 001x, Q = 0
\item \textbf{8H} when immh = 001x, Q = 1
\item \textbf{2S} when immh = 01xx, Q = 0
\item \textbf{4S} when immh = 01xx, Q = 1
\item \textbf{2D} when immh = 1xxx, Q = 1
\end{itemize}

See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<\Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<\shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

\begin{itemize}
\item (128-\textbf{UInt}(immh:immb)) when immh = 1xxx
\end{itemize}

The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

\begin{itemize}
\item (16-\textbf{UInt}(immh:immb)) when immh = 0001
\item (32-\textbf{UInt}(immh:immb)) when immh = 001x
\item (64-\textbf{UInt}(immh:immb)) when immh = 01xx
\item (128-\textbf{UInt}(immh:immb)) when immh = 1xxx
\end{itemize}

See Advanced SIMD modified immediate on page C4-632 when immh = 0000.

**Operation for all encodings**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 \ll (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
\begin{itemize}
\item element = (\textbf{Int}(Elem[operand, e, esize], unsigned) + round_const) \gg shift;
\item Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
\end{itemize}
\forall[d, \text{datasize}] = \text{result};

**Operational information**

If \text{PSTATE.DIT} is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.318 SSRA

Signed Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see SRSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

```
|[31 30 29 28|27 26 25 24|23 22 | 19 18 | 16|15 14 13 12|11 10 9 ] 5 4 | 0 |
 0 1 0 1 1 1 1 0 | !=0000 | immh 0 0 0 1 0 1 | Rn | Rd |

U   immh  o1 o0
```

Encoding

SSRA <V><d>, <V><n>, #<shift>

Decode for this encoding

- integer \( d = \text{UInt}(Rd) \);
- integer \( n = \text{UInt}(Rn) \);
- if \( \text{immh} < 3 \) != '1' then UNDEFINED;
- integer \( \text{esize} = 8 << 3 \);
- integer \( \text{datasize} = \text{esize} \);
- integer \( \text{elements} = 1 \);
- integer \( \text{shift} = (\text{esize} * 2) - \text{UInt}(\text{immh}:\text{immb}) \);
- boolean \( \text{unsigned} = (U == '1') \);
- boolean \( \text{round} = (o1 == '1') \);
- boolean \( \text{accumulate} = (o0 == '1') \);

Vector

```
|[31 30 29 28|27 26 25 24|23 22 | 19 18 | 16|15 14 13 12|11 10 9 ] 5 4 | 0 |
 0 Q 0 1 1 1 1 0 | !=0000 | immh 0 0 0 1 0 1 | Rn | Rd |

U   immh  o1 o0
```

Encoding

SSRA <Vd>.<T>, <Vn>.<T>, #<shift>

Decode for this encoding

- integer \( d = \text{UInt}(Rd) \);
- integer \( n = \text{UInt}(Rn) \);
- if \( \text{immh} == '0000' \) then SEE "Advanced SIMD modified immediate";
- if \( \text{immh}:Q == '10' \) then UNDEFINED;
- integer \( \text{esize} = 8 << \text{HighestSetBit}(\text{immm}) \);
- integer \( \text{datasize} = \text{if} \ Q == '1' \ then 128 \ else 64 \);
- integer \( \text{elements} = \text{datasize} \ DIV \ \text{esize} \);
- integer \( \text{shift} = (\text{esize} * 2) - \text{UInt}(\text{immh}:\text{immb}) \);
- boolean \( \text{unsigned} = (U == '1') \);
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler symbols

\[ <V> \]
Is a width specifier, encoded in the "immh" field. It can have the following values:

- \( D \) when \( \text{immh} = 1\text{xxx} \)
  The encoding \( \text{immh} = 0\text{xxx} \) is reserved.

\[ <d> \]
Is the number of the SIMD&FP destination register, in the "Rd" field.

\[ <n> \]
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\[ <Vd> \]
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\[ <T> \]
Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

- \( 8B \) when \( \text{immh} = 0001, Q = 0 \)
- \( 16B \) when \( \text{immh} = 0001, Q = 1 \)
- \( 4H \) when \( \text{immh} = 001x, Q = 0 \)
- \( 8H \) when \( \text{immh} = 001x, Q = 1 \)
- \( 2S \) when \( \text{immh} = 01xx, Q = 0 \)
- \( 4S \) when \( \text{immh} = 01xx, Q = 1 \)
- \( 2D \) when \( \text{immh} = 1\text{xxx}, Q = 1 \)

See Advanced SIMD modified immediate on page C4-632 when \( \text{immh} = 0000, Q = x \).

The encoding \( \text{immh} = 1\text{xxx}, Q = 0 \) is reserved.

\[ <Vn> \]
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\[ <\text{shift}> \]
For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

- \( (128-\text{UInt}(\text{immh:immb})) \) when \( \text{immh} = 1\text{xxx} \)
  The encoding \( \text{immh} = 0\text{xxx} \) is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

- \( (16-\text{UInt}(\text{immh:immb})) \) when \( \text{immh} = 0001 \)
- \( (32-\text{UInt}(\text{immh:immb})) \) when \( \text{immh} = 001x \)
- \( (64-\text{UInt}(\text{immh:immb})) \) when \( \text{immh} = 01xx \)
- \( (128-\text{UInt}(\text{immh:immb})) \) when \( \text{immh} = 1\text{xxx} \)

See Advanced SIMD modified immediate on page C4-632 when \( \text{immh} = 0000 \).

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
  element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
  Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.319  SSUBL, SSUBL2

Signed Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are twice as long as the source vector elements.

The SSUBL instruction extracts each source vector from the lower half of each source register. The SSUBL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
</tr>
<tr>
<td>U</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Decoding for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

Assembler symbols

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

| [absent] | when Q = 0 |
| [present] | when Q = 1 |

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  Is an arrangement specifier, encoded in the "size" field. It can have the following values:

| 8H | 4S | 2D |
| when size = 00 | when size = 01 | when size = 10 |

The encoding size = 11 is reserved.

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

| 88 |
| when size = 00, Q = 0 |
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bite(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d, 2*datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
### C7.2.320 SSUBW, SSUBW2

Signed Subtract Wide. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The SSUBW instruction extracts the second source vector from the lower half of the second source register. The SSUBW2 instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Encoding

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Decoding for this encoding

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

#### Assembler symbols

- 2
  - 2
    - 0
      - Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
      - [absent] when Q = 0
      - [present] when Q = 1

- <Vd>
  - Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Ta>
  - Is an arrangement specifier, encoded in the "size" field. It can have the following values:
    - 8H when size = 00
    - 4S when size = 01
    - 2D when size = 10

  The encoding size = 11 is reserved.

- <Vn>
  - Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

- <Vm>
  - Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<<esize-1:0>;
V[d, 2*datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.321 ST1 (multiple structures)

Store multiple single-element structures from one, two, three, or four registers. This instruction stores elements to memory from one, two, three, or four SIMD&FP registers, without interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

![Instruction Format](image)

One register variant

Applies when opcode == 0111.

ST1 { <Vt>.<T> }, [<Xn|SP>]

Two registers variant

Applies when opcode == 1010.

ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

Three registers variant

Applies when opcode == 0110.

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

Four registers variant

Applies when opcode == 0010.

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]

Decode for all variants of this encoding

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

![Instruction Format](image)

One register, immediate offset variant

Applies when Rm == 11111 && opcode == 0111.

ST1 { <Vt>.<T> }, [<Xn|SP>], <imm>
One register, register offset variant
Applies when \( Rm != 11111 \) \&\& \( \text{opcode} == 0111 \).

\[ \text{ST1} \{ <Vt>.<T> \}, [<Xn|SP>], <Xm> \]

Two registers, immediate offset variant
Applies when \( Rm == 11111 \) \&\& \( \text{opcode} == 1010 \).

\[ \text{ST1} \{ <Vt>.<T>, <Vt2>.<T> \}, [<Xn|SP>], <\text{imm}> \]

Two registers, register offset variant
Applies when \( Rm != 11111 \) \&\& \( \text{opcode} == 1010 \).

\[ \text{ST1} \{ <Vt>.<T>, <Vt2>.<T> \}, [<Xn|SP>], <Xm> \]

Three registers, immediate offset variant
Applies when \( Rm == 11111 \) \&\& \( \text{opcode} == 0110 \).

\[ \text{ST1} \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> \}, [<Xn|SP>], <\text{imm}> \]

Three registers, register offset variant
Applies when \( Rm != 11111 \) \&\& \( \text{opcode} == 0110 \).

\[ \text{ST1} \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> \}, [<Xn|SP>], <Xm> \]

Four registers, immediate offset variant
Applies when \( Rm == 11111 \) \&\& \( \text{opcode} == 0010 \).

\[ \text{ST1} \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> \}, [<Xn|SP>], <\text{imm}> \]

Four registers, register offset variant
Applies when \( Rm != 11111 \) \&\& \( \text{opcode} == 0010 \).

\[ \text{ST1} \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> \}, [<Xn|SP>], <Xm> \]

Decode for all variants of this encoding
\[
\begin{align*}
\text{integer } t & = \text{UInt}(Rt); \\
\text{integer } n & = \text{UInt}(Rn); \\
\text{integer } m & = \text{UInt}(Rm); \\
\text{boolean } wback & = \text{TRUE}; \\
\text{boolean } \text{tag\_checked} & = wback \text{ || } n != 31; 
\end{align*}
\]

Assembler symbols
- \(<Vt>\) is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- \(<T>\) is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - 8B when size = 00, Q = 0
  - 16B when size = 00, Q = 1
  - 4H when size = 01, Q = 0
  - 8H when size = 01, Q = 1
  - 2S when size = 10, Q = 0
  - 4S when size = 10, Q = 1
  - 1D when size = 11, Q = 0
2D when size = 11, Q = 1

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#8 when Q = 0
#16 when Q = 1

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#16 when Q = 0
#32 when Q = 1

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#24 when Q = 0
#48 when Q = 1

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in the "Q" field. It can have the following values:

#32 when Q = 0
#64 when Q = 1

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

\[
\text{MemOp} \text{ memop} = \begin{cases} 
\text{MemOp\_LOAD} & \text{if } L = '1' \\
\text{MemOp\_STORE} & \text{else} 
\end{cases};
\]

integer datasize = if Q = '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;

Operation for all encodings

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
  for e = 0 to elements-1
    tt = (t + r) MOD 32;
    for s = 0 to selem-1
      rval = V[tt, datasize];
      if memop == MemOp_LOAD then
        Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
        V[tt, datasize] = rval;
      else // memop == MemOp_STORE
        Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
      offs = offs + ebytes;
      tt = (tt + 1) MOD 32;
  if wback then
    if m != 31 then
      offs = X[m, 64];
    if n == 31 then
      SP[] = address + offs;
    else
      X[n, 64] = address + offs;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.322   ST1 (single structure)

Store a single-element structure from one lane of one register. This instruction stores the specified element of a SIMD&FP register to memory.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

8-bit variant
Applies when opcode == 000.
ST1 { <Vt>.B }[<index>], [Xn|SP]

16-bit variant
Applies when opcode == 010 && size == x0.
ST1 { <Vt>.H }[<index>], [Xn|SP]

32-bit variant
Applies when opcode == 100 && size == 00.
ST1 { <Vt>.S }[<index>], [Xn|SP]

64-bit variant
Applies when opcode == 100 && S == 0 && size == 01.
ST1 { <Vt>.D }[<index>], [Xn|SP]

Decode for all variants of this encoding

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wbback = FALSE;
boolean tag_checked = wbback || n != 31;

Post-index

8-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 000.
ST1 { <Vt>.B }[<index>], [Xn|SP], #1

8-bit, register offset variant
Applies when Rm != 11111 && opcode == 000.
ST1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 010 && size == x0.
ST1 { <Vt>.H }[<index>], [<Xn|SP>], #2

16-bit, register offset variant
Applies when Rm != 11111 && opcode == 010 && size == x0.
ST1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 100 && size == 00.
ST1 { <Vt>.S }[<index>], [<Xn|SP>], #4

32-bit, register offset variant
Applies when Rm != 11111 && opcode == 100 && size == 00.
ST1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 100 && S == 0 && size == 01.
ST1 { <Vt>.D }[<index>], [<Xn|SP>], #8

64-bit, register offset variant
Applies when Rm != 11111 && opcode == 100 && S == 0 && size == 01.
ST1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>

Decode for all variants of this encoding

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
@index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
Shared decode for all encodings

integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);    // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datysize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;

Operation for all encodings

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

offs = Zeros(64);
if replicate then
  // load and replicate to all elements
  for s = 0 to selem-1
    element = Mem[address+offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t, datysize] = Replicate(element, datysize DIV esize);
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
else
  // load/store one element per register
  for s = 0 to selem-1
    rval = V[t, 128];
    if memop == MemOp_LOAD then
      // insert into one lane of 128-bit register

      // More code here
\texttt{Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];}
\texttt{V[t, 128] = rval;}
\texttt{else // memop == MemOp\_STORE}
\texttt{// extract from one lane of 128-bit register}
\texttt{Mem[address+offs, ebytes, AccType\_VEC] = Elem[rval, index, esize];}
\texttt{offs = offs + ebytes;}
\texttt{t = (t + 1) \texttt{MOD 32;}}
\texttt{if wback then}
\texttt{if m != 31 then}
\texttt{offs = X[m, 64];}
\texttt{if n == 31 then}
\texttt{SP[] = address + offs;}
\texttt{else}
\texttt{X[n, 64] = address + offs;}

\textbf{Operational information}

If \texttt{PSTATE.DIT} is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.323   ST2 (multiple structures)

Store multiple 2-element structures from two registers. This instruction stores multiple 2-element structures from
two SIMD&FP registers to memory, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

No offset

\[
\begin{array}{cccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \text{size} & \text{Rn} & \text{Rt} & \\
\end{array}
\]

Encoding

ST2 \{ <Vt>.<T>, <Vt2>.<T> \}, \{<Xn|SP>\}

Decode for this encoding

\[
\begin{align*}
\text{integer } t &= \text{UInt}(Rt); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{integer UNKNOWN}; \\
\text{boolean } \text{wback} &= \text{FALSE}; \\
\text{boolean } \tag\_\text{checked} &= \text{wback} || n != 31;
\end{align*}
\]

Post-index

\[
\begin{array}{cccccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & \text{Rm} & 1 & 0 & 0 & 0 & \text{size} & \text{Rn} & \text{Rt} & \\
\end{array}
\]

Immediate offset variant

 Applies when \( Rm == 11111 \).

ST2 \{ <Vt>.<T>, <Vt2>.<T> \}, \{<Xn|SP>\}, <imm>

Register offset variant

 Applies when \( Rm != 11111 \).

ST2 \{ <Vt>.<T>, <Vt2>.<T> \}, \{<Xn|SP>\}, <Xm>

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } t &= \text{UInt}(Rt); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{boolean } \text{wback} &= \text{TRUE}; \\
\text{boolean } \tag\_\text{checked} &= \text{wback} || n != 31;
\end{align*}
\]

Assembler symbols

\begin{align*}
<Vt> & \text{ Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.} \\
<T> & \text{ Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:}
\end{align*}

\[
\begin{array}{c}
\begin{array}{c}
\text{88 when } \text{size} = 00, Q = 0
\end{array}
\end{array}
\]

C7-2784   Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  ARM DDI 0487l.a
Non-Confidential  ID081822
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:
#16 when Q = 0
#32 when Q = 1
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements
case opcode of
when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;

Operation for all encodings

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];
offs = Zeros(64);
for r = 0 to rpt-1
  for e = 0 to elements-1
    tt = (t + r) MOD 32;
    for s = 0 to selem-1
      rval = V[tt, datasize];
      if memop == MemOp_LOAD then
        Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
        V[tt, datasize] = rval;
      else // memop == MemOpgetStore
        Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
        offs = offs + ebytes;
      tt = (tt + 1) MOD 32;
      if wback then
        if m != 31 then
          offs = X[m, 64];
        if n == 31 then
          SP[] = address + offs;
        else
          X[n, 64] = address + offs;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.324   ST2 (single structure)

Store single 2-element structure from one lane of two registers. This instruction stores a 2-element structure to memory from corresponding elements of two SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 13 12 11 10 9 5 4 0 0]
```

8-bit variant

Applies when opcode == 000.

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

16-bit variant

Applies when opcode == 010 && size == x0.

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

32-bit variant

Applies when opcode == 100 && size == 00.

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

64-bit variant

Applies when opcode == 100 && S == 0 && size == 01.

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 13 12 11 10 9 5 4 0 0]
```

8-bit, immediate offset variant

Applies when Rm == 11111 && opcode == 000.

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

8-bit, register offset variant

Applies when Rm != 11111 && opcode == 000.
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 010 && size == x0.
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4

16-bit, register offset variant
Applies when Rm != 11111 && opcode == 010 && size == x0.
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 100 && size == 00.
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #8

32-bit, register offset variant
Applies when Rm != 11111 && opcode == 100 && size == 00.
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 100 && S == 0 && size == 01.
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16

64-bit, register offset variant
Applies when Rm != 11111 && opcode == 100 && S == 0 && size == 01.
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>

Decode for all variants of this encoding
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
Shared decode for all encodings

integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
when 3
  // load and replicate
  if L == '0' || S == '1' then UNDEFINED;
  scale = UInt(size);
  replicate = TRUE;
when 0
  index = UInt(Q:S:size);    // B[0-15]
when 1
  if size<0> == '1' then UNDEFINED;
  index = UInt(Q:S:size<1>);    // H[0-7]
when 2
  if size<1> == '1' then UNDEFINED;
  if size<0> == '0' then
    index = UInt(Q:S);    // S[0-3]
  else
    if S == '1' then UNDEFINED;
    index = UInt(Q);    // D[0-1]
  scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;

Operation for all encodings

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

offs = Zeros(64);
if replicate then
  // load and replicate to all elements
  for s = 0 to selem-1
    element = Mem[address+offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t, datasize] = Replicate(element, datasize DIV esize);
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
else
  // load/store one element per register
  for s = 0 to selem-1
    rval = V[t, 128];
    if memop == MemOp_LOAD then
      // insert into one lane of 128-bit register

\[
\text{Elem[rval, index, esize]} = \text{Mem[address+offs, ebytes, AccType_VEC]};
\]
\[
\text{V[t, 128] = rval};
\]
\[
\text{else // memop == MemOp\_STORE}
\]
\[
\text{// extract from one lane of 128-bit register}
\]
\[
\text{Mem[address+offs, ebytes, AccType\_VEC]} = \text{Elem[rval, index, esize]};
\]
\[
\text{offs = offs + ebytes;}
\]
\[
\text{t = (t + 1) MOD 32;}
\]
\[
\text{if wback then}
\]
\[
\text{if m != 31 then}
\]
\[
\text{offs = X[m, 64];}
\]
\[
\text{if n == 31 then}
\]
\[
\text{SP[]} = \text{address + offs;}
\]
\[
\text{else}
\]
\[
\text{X[n, 64] = address + offs;}
\]

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.325  ST3 (multiple structures)

Store multiple 3-element structures from three registers. This instruction stores multiple 3-element structures to memory from three SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

\[
\begin{array}{ccccccc|c|c}
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \text{size} & Rn & Rt
\end{array}
\]

L opcode

Encoding

ST3 \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> \}, [<Xn|SP>]

Decode for this encoding

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

\[
\begin{array}{ccccccc|c|c}
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \text{size} & Rm & Rn & Rt
\end{array}
\]

L opcode

Immediate offset variant

Applies when Rm == 11111.

ST3 \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> \}, [<Xn|SP>], <imm>

Register offset variant

Applies when Rm != 11111.

ST3 \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> \}, [<Xn|SP>], <Xm>

Decode for all variants of this encoding

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler symbols

\(<Vt>\)  Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

\(<T>\)  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

88  when size = 00, Q = 0
16B  when size = 00, Q = 1
4H   when size = 01, Q = 0
8H   when size = 01, Q = 1
2S   when size = 10, Q = 0
4S   when size = 10, Q = 1
2D   when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:
   #24 when Q = 0
   #48 when Q = 1
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

MemOp menop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;  // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;  // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;  // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;  // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;  // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;  // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;  // LD/ST1 (2 registers)
otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;

Operation for all encodings

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
address = X[n, 64];
offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.326  ST3 (single structure)

Store single 3-element structure from one lane of three registers. This instruction stores a 3-element structure to memory from corresponding elements of three SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**No offset**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 1 0 1</td>
<td>0 0 0 0 0 0</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>S</td>
<td>size</td>
</tr>
</tbody>
</table>

L R opcode
```

**8-bit variant**

Applies when opcode == 001.

```
```

**16-bit variant**

Applies when opcode == 011 && size == x0.

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

**32-bit variant**

Applies when opcode == 101 && size == 00.

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

**64-bit variant**

Applies when opcode == 101 && S == 0 && size == 01.

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

**Decode for all variants of this encoding**

```java
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

**Post-index**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 1 0 1</td>
<td>0 0 0 0 0 0</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>S</td>
<td>size</td>
</tr>
</tbody>
</table>

L R opcode
```

**8-bit, immediate offset variant**

Applies when Rm == 11111 && opcode == 001.

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

**8-bit, register offset variant**

Applies when Rm != 11111 && opcode == 001.
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 011 && size == x0.
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6

16-bit, register offset variant
Applies when Rm != 11111 && opcode == 011 && size == x0.
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [Xn|SP], <Xm>

32-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 101 && size == 00.
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12

32-bit, register offset variant
Applies when Rm != 11111 && opcode == 101 && size == 00.
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [Xn|SP], <Xm>

64-bit, immediate offset variant
Applies when Rm == 11111 && opcode == 101 && S == 0 && size == 01.
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [Xn|SP], #24

64-bit, register offset variant
Applies when Rm != 11111 && opcode == 101 && S == 0 && size == 01.
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [Xn|SP], <Xm>

Decode for all variants of this encoding
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler symbols
<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
Shared decode for all encodings

```plaintext
text
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
        scale = 3;

    MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
    integer datasize = if Q == '1' then 128 else 64;
    integer esize = 8 << scale;

Operation for all encodings

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

    CheckFPAdvSIMDEnabled64();

    bits(64) address;
    bits(64) offs;
    bits(128) rval;
    bits(esize) element;
    constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[ ];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
```

Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
V[t, 128] = rval;
else // memop == MemOp_STORE
   // extract from one lane of 128-bit register
   Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
offs = offs + ebytes;
t = (t + 1) MOD 32;

if wback then
   if m != 31 then
      offs = X[m, 64];
   if n == 31 then
      SP[] = address + offs;
   else
      X[n, 64] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.327  ST4 (multiple structures)

Store multiple 4-element structures from four registers. This instruction stores multiple 4-element structures to memory from four SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 12|11 10 9 | 5 4 | 0 |
|0 |Q|0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 |size| Rm | Rn | Rt |
```

Encoding


Decode for this encoding

```plaintext
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

```
|31 30 29 28|27 26 25 24|23 22 21 20|15 12|11 10 9 | 5 4 | 0 |
|0 |Q|0 0 1 1 0 0 1 0 0 |size| Rm | Rn | Rt |
```

Immediate offset variant

Applies when Rm == 11111.

ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [ <Xn>|SP> ], <imm>

Register offset variant

Applies when Rm != 11111.

ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [ <Xn>|SP> ], <Xm>

Decode for all variants of this encoding

```plaintext
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler symbols

- `<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - 88 when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the post-index immediate offset, encoded in the "Q" field. It can have the following values:
#32 when Q = 0
#64 when Q = 1
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared decode for all encodings**

\[
\text{MemOp memop = if } L == '1' \text{ then MemOp\_LOAD else MemOp\_STORE;}
\]
\[
\text{integer datasize = if } Q == '1' \text{ then 128 else 64;}
\]
\[
\text{integer esize = 8} \ll \text{UInt(size);}
\]
\[
\text{integer elements = datasize DIV esize;}
\]
\[
\text{integer rpt; // number of iterations}
\]
\[
\text{integer selem; // structure elements}
\]

case opcode of
  when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
  otherwise UNDEFINED;

  // .ID format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;

**Operation for all encodings**

\[
\text{CheckFPAdvSIMDEnabled64();}
\]
\[
\text{bits(64) address;}
\]
\[
\text{bits(64) offs;}
\]
\[
\text{bits(datasize) rval;}
\]
\[
\text{integer tt;}
\]
\[
\text{constant integer ebytes = esize DIV 8;}
\]
\[
\text{if HaveMTE2Ext() then}
\]
\[
\text{SetTagCheckedInstruction(tag\_checked);}
\]
\[
\text{if n == 31 then}
\]
\[
\text{CheckSPAlignment();}
\]
address = \textit{SP}[]; \\
else \\
    address = X[n, 64];

offs = \textit{Zeros}(64); \\
for \( r = 0 \) to \( \text{rpt}-1 \) \\
    for \( e = 0 \) to \( \text{elements}-1 \) \\
        \( tt = (t + r) \text{ MOD } 32; \) \\
        for \( s = 0 \) to \( \text{selem}-1 \) \\
            rval = V[tt, \text{datasize}]; \\
            if \text{memop} == \text{MemOp\_LOAD} then \\
                Elem[rval, e, esize] = \text{Mem}[\text{address}+\text{offs}, e\text{bytes}, \text{AccType\_VEC}]; \\
                V[tt, \text{datasize}] = rval; \\
            else // memop == MemOp\_STORE \\
                \text{Mem}[\text{address}+\text{offs}, e\text{bytes}, \text{AccType\_VEC}] = Elem[rval, e, esize]; \\
                offs = offs + e\text{bytes}; \\
                tt = (tt + 1) \text{ MOD } 32;

if \text{wbback} then \\
    if \( m \neq 31 \) then \\
        offs = X[m, 64]; \\
    if n == 31 then \\
        \textit{SP}[] = address + offs; \\
    else \\
        X[n, 64] = address + offs;

\textbf{Operational information}

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.328  ST4 (single structure)

Store single 4-element structure from one lane of four registers. This instruction stores a 4-element structure to memory from corresponding elements of four SIMD&FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

No offset

[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 13 12 11 10 9 ] | 5 4 | 0 |
0 Q 0 0 1 1 0 1 0 0 0 0 0 0 x x 1 S size Rn Rt
L R opcode

8-bit variant

Applies when opcode == 001.


16-bit variant

Applies when opcode == 011 && size == x0.


32-bit variant

Applies when opcode == 101 && size == 00.


64-bit variant

Applies when opcode == 101 && S == 0 && size == 01.


Decode for all variants of this encoding

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 13 12 11 10 9 ] | 5 4 | 0 |
0 Q 0 0 1 1 0 1 1 0 1 1 0 0 x x 1 S size Rn Rt
L R opcode

8-bit, immediate offset variant

Applies when Rm == 11111 && opcode == 001.


8-bit, register offset variant

Applies when Rm != 11111 && opcode == 001.

16-bit, immediate offset variant
Applies when \( Rm == 11111 && \text{opcode} == 011 && size == x0 \).

16-bit, register offset variant
Applies when \( Rm != 11111 && \text{opcode} == 011 && size == x0 \).

32-bit, immediate offset variant
Applies when \( Rm == 11111 && \text{opcode} == 101 && size == 00 \).

32-bit, register offset variant
Applies when \( Rm != 11111 && \text{opcode} == 101 && size == 00 \).
ST4 \{ <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S \}[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset variant
Applies when \( Rm == 11111 && \text{opcode} == 101 && S == 0 && size == 01 \).
ST4 \{ <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D \}[<index>], [<Xn|SP>], #32

64-bit, register offset variant
Applies when \( Rm != 11111 && \text{opcode} == 101 && S == 0 && size == 01 \).
ST4 \{ <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D \}[<index>], [<Xn|SP>], <Xm>

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } t &= \text{UInt}(Rt); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{boolean } wback &= \text{TRUE}; \\
\text{boolean } \text{tag\_checked} &= wback \lor n != 31;
\end{align*}
\]

Assembler symbols

\(<Vt>\) Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
\(<Vt2>\) Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
\(<Vt3>\) Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
\(<Vt4>\) Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
\(<\text{index}>\) For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared decode for all encodings

```plaintext
type init_scale = UInt(opcode<2:1>);
type scale = init_scale;
type selem = UInt(opcode<0>:R) + 1;
type replicate = FALSE;
type index;

    case scale of
        when 3
            // load and replicate
            if L == '0' || S == '1' then UNDEFINED;
            scale = UInt(size);
            replicate = TRUE;
        when 0
            index = UInt(Q:S:size);    // B[0-15]
        when 1
            if size<0> == '1' then UNDEFINED;
            index = UInt(Q:S:size<1>);    // H[0-7]
        when 2
            if size<1> == '1' then UNDEFINED;
            if size<0> == '0' then
                index = UInt(Q:S);    // S[0-3]
            else
                if S == '1' then UNDEFINED;
                index = UInt(Q);    // D[0-1]
            scale = 3;

    MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
type datasize = if Q == '1' then 128 else 64;
type esize = 8 << scale;
```

Operation for all encodings

```plaintext
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

    CheckFPAdvSIMDEnabled64();

    bits(64) address;
    bits(64) offs;
    bits(128) rval;
    bits(esize) element;
    constant integer ebytes = esize DIV 8;

    if n == 31 then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n, 64];

    offs = Zeros(64);
    if replicate then
        // load and replicate to all elements
        for s = 0 to selem-1
            element = Mem[address+offs, ebytes, AccType_VEC];
            // replicate to fill 128- or 64-bit register
            V[t, datasize] = Replicate(element, datasize DIV esize);
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
    else
        // load/store one element per register
```
for s = 0 to selem-1
  rval = V[t, 128];
  if memop == MemOp_LOAD then
    // insert into one lane of 128-bit register
    Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
    V[t, 128] = rval;
  else // memop == MemOp_STORE
    // extract from one lane of 128-bit register
    Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
  offs = offs + ebytes;
  t = (t + 1) MOD 32;
if wback then
  if m != 31 then
    offs = X[m, 64];
  if n == 31 then
    SP[] = address + offs;
  else
    X[n, 64] = address + offs;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.329 STNP (SIMD&FP)

Store Pair of SIMD&FP registers, with Non-temporal hint. This instruction stores a pair of SIMD&FP registers to memory, issuing a hint to the memory system that the access is non-temporal. The address used for the store is calculated from an address from a base register value and an immediate offset. For information about non-temporal pair instructions, see Load/store SIMD and floating-point non-temporal pair on page C3-265.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

32-bit variant
Applies when opc == 00.

STNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]

64-bit variant
Applies when opc == 01.

STNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]

128-bit variant
Applies when opc == 10.

STNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]

Decode for all variants of this encoding

// Empty.

Assembler symbols

<Dt1> Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt2> Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<Qt1> Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt2> Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<St1> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<St2> Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.
Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;

Operation

CheckFPEnabled64();
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
else
    address = X[n, 64];
address = address + offset;
data1 = V[t, datasize];
data2 = V[t2, datasize];
Mem[address, dbytes, AccType_VECSTREAM] = data1;
Mem[address+dbytes, dbytes, AccType_VECSTREAM] = data2;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.330   STP (SIMD&FP)

Store Pair of SIMD&FP registers. This instruction stores a pair of SIMD&FP registers to memory. The address used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Post-index

32-bit variant
Applies when opc == 00.

STP <St1>, <St2>, [<Xn|SP>], #<imm>

64-bit variant
Applies when opc == 01.

STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>

128-bit variant
Applies when opc == 10.

STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>

Decode for all variants of this encoding

boolean wback = TRUE;
boolean postindex = TRUE;

Pre-index

32-bit variant
Applies when opc == 00.

STP <St1>, <St2>, [<Xn|SP>, #<imm>]

64-bit variant
Applies when opc == 01.

STP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]

128-bit variant
Applies when opc == 10.

STP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]
Decode for all variants of this encoding

boolean wback = TRUE;
boolean postindex = FALSE;

Signed offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>15 14</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>1 0 1</td>
<td>1 0 0</td>
<td>imm7</td>
<td>Rt2</td>
<td>Rn</td>
<td>Rt</td>
</tr>
</tbody>
</table>

32-bit variant

Applies when opc == 00.

STP <St1>, <St2>, [{<Xn|SP}>{, #<imm}>]

64-bit variant

Applies when opc == 01.

STP <Dt1>, <Dt2>, [{<Xn|SP}>{, #<imm}>]

128-bit variant

Applies when opc == 10.

STP <Qt1>, <Qt2>, [{<Xn|SP}>{, #<imm}>]

Decode for all variants of this encoding

boolean wback = FALSE;
boolean postindex = FALSE;

Assembler symbols

<Dt1> Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt2> Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<Qt1> Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt2> Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<St1> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<St2> Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.

For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.
For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.
For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared decode for all encodings

define integer n = UInt(Rn);
define integer t = UInt(Rt);
define integer t2 = UInt(Rt2);
define integer scale = 2 + UInt(opc);
define integer datasize = 8 << scale;
define bits(64) offset = LSL(SignExtend(imm7, 64), scale);
define boolean tag_checked = wback || n != 31;

Operation for all encodings

CheckFPEnabled64();
define bits(64) address;
define bits(datasize) data1;
define bits(datasize) data2;
define constant integer dbytes = datasize DIV 8;
define if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
define if n == 31 then
    CheckSPAlignment();
define else
    address = X[n, 64];
define if !postindex then
    address = address + offset;
define data1 = V[t, datasize];
define data2 = V[t2, datasize];
define Mem[address, dbytes, AccType_VEC] = data1;
define Mem[address+dbytes, dbytes, AccType_VEC] = data2;
define if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
define else
        X[n, 64] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.331   STR (immediate, SIMD&FP)

Store SIMD&FP register (immediate offset). This instruction stores a single SIMD&FP register to memory. The
address that is used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state
and Exception level, an attempt to execute the instruction might be trapped.

Post-index

[31 30 29 28|27 26 25 24|23 22 21 20] | 12|11 10 9 | 5 4 | 0 |
size     1 1 1 1 0 0 x 0 0 _imm9  0 1  Rn  Rt
          opc

8-bit variant

Applies when size == 00 && opc == 00.
STR <Bt>, [<Xn|SP>], #<simm>

16-bit variant

Applies when size == 01 && opc == 00.
STR <Ht>, [<Xn|SP>], #<simm>

32-bit variant

Applies when size == 10 && opc == 00.
STR <St>, [<Xn|SP>], #<simm>

64-bit variant

Applies when size == 11 && opc == 00.
STR <Dt>, [<Xn|SP>], #<simm>

128-bit variant

Applies when size == 00 && opc == 10.
STR <Qt>, [<Xn|SP>], #<simm>

Decode for all variants of this encoding

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);

Pre-index

[31 30 29 28|27 26 25 24|23 22 21 20] | 12|11 10 9 | 5 4 | 0 |
size     1 1 1 1 0 0 x 0 0 _imm9  1  Rn  Rt
          opc

8-bit variant

Applies when size == 00 && opc == 00.
STR <Bt>, [<Xn|SP>, #<simm>]!

**16-bit variant**
Applies when size == 01 && opc == 00.
STR <Ht>, [<Xn|SP>, #<simm>]!

**32-bit variant**
Applies when size == 10 && opc == 00.
STR <St>, [<Xn|SP>, #<simm>]!

**64-bit variant**
Applies when size == 11 && opc == 00.
STR <Dt>, [<Xn|SP>, #<simm>]!

**128-bit variant**
Applies when size == 00 && opc == 10.
STR <Qt>, [<Xn|SP>, #<simm>]!

**Decode for all variants of this encoding**

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bites(64) offset = SignExtend(imm9, 64);

**Unsigned offset**

| [31 30 29 28]27 26 25 24|23 22 21 | 10 9 | 5 4 | 0 |
| size | 1 1 | 1 0 | 1 x 0 | imm12 | Rn | Rt |

**8-bit variant**
Applies when size == 00 && opc == 00.
STR <Bt>, [<Xn|SP>{, #<pimm}>]

**16-bit variant**
Applies when size == 01 && opc == 00.
STR <Ht>, [<Xn|SP>{, #<pimm}>]

**32-bit variant**
Applies when size == 10 && opc == 00.
STR <St>, [<Xn|SP>{, #<pimm}>]

**64-bit variant**
Applies when size == 11 && opc == 00.
STR <Dt>, [<Xn|SP>{, #<pimm}>]

**128-bit variant**
Applies when size == 11 && opc == 00.
**128-bit variant**

Applies when `size == 00 & opc == 10`.

\[ \text{STR} <Qt>, [<Xn|SP>{, #<pimm>}] \]

**Decode for all variants of this encoding**

```plaintext
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

**Assemble symbols**

- `<Bt>`: Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Dt>`: Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Ht>`: Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Qt>`: Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<St>`: Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>`: Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- `<pimm>`: For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
  - For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as `<pimm>/2`.
  - For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as `<pimm>/4`.
  - For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as `<pimm>/8`.
  - For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as `<pimm>/16`.

**Shared decode for all encodings**

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasource = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
```

**Operation for all encodings**

```plaintext
CheckFPEnabled64();
bias(64) address;
bias(datasize) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
```
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t, datasize];
        Mem[address, datasize DIV 8, AccType_VEC] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t, datasize] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[0] = address;
    else
        X[n, 64] = address;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.332  STR (register, SIMD&FP)

Store SIMD&FP register (register offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

8-bit variant

Applies when \( \text{size} = 00 \) && \( \text{opc} = 00 \) && \( \text{option} \neq 011 \).

\[
\text{STR} \ <Bt>, \ [\langle Xn|SP\rangle, \ (<Wm>|<Xm>), \ <extend> \ {<amount>}]\]

8-bit variant

Applies when \( \text{size} = 00 \) && \( \text{opc} = 00 \) && \( \text{option} = 011 \).

\[
\text{STR} \ <Bt>, \ [\langle Xn|SP\rangle, \ <Xm>{, \ LSL \ <amount}>]\]

16-bit variant

Applies when \( \text{size} = 01 \) && \( \text{opc} = 00 \).

\[
\text{STR} \ <Ht>, \ [\langle Xn|SP\rangle, \ (<Wm>|<Xm>)\{, \ <extend> \ {<amount>}}\]

32-bit variant

Applies when \( \text{size} = 10 \) && \( \text{opc} = 00 \).

\[
\text{STR} \ <St>, \ [\langle Xn|SP\rangle, \ (<Wm>|<Xm>)\{, \ <extend> \ {<amount>}}\]

64-bit variant

Applies when \( \text{size} = 11 \) && \( \text{opc} = 00 \).

\[
\text{STR} \ <Dt>, \ [\langle Xn|SP\rangle, \ (<Wm>|<Xm>)\{, \ <extend> \ {<amount>}}\]

128-bit variant

Applies when \( \text{size} = 00 \) && \( \text{opc} = 10 \).

\[
\text{STR} \ <Qt>, \ [\langle Xn|SP\rangle, \ (<Wm>|<Xm>)\{, \ <extend> \ {<amount>}}\]

Decode for all variants of this encoding

\[
\text{integer scale} = \text{UInt}(\text{opc}<1>:\text{size});
\]

\[
\text{if} \ \text{scale} > 4 \ \text{then UNDEFINED};
\]

\[
\text{if} \ \text{option}<1> = '0' \ \text{then UNDEFINED}; \quad // \text{sub-word index}
\]

\[
\text{ExtendType extend_type} = \text{DecodeRegExtend}(\text{option});
\]

\[
\text{integer shift} = \text{if} \ S = '1' \ \text{then scale else 0};
\]

Assembler symbols

\(<Bt> \quad \text{Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.}\)
<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<It> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<S> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> For the 8-bit variant: is the index extend specifier, encoded in the "option" field. It can have the following values:

- UXTW when option = 010
- SXTW when option = 110
- SXTX when option = 111

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- UXTW when option = 010
- LSL when option = 011
- SXTW when option = 110
- SXTX when option = 111

<amount> For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- #0 when S = 0
- #1 when S = 1

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- #0 when S = 0
- #2 when S = 1

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- #0 when S = 0
- #3 when S = 1

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- #0 when S = 0
- #4 when S = 1
Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;

Operation

bits(64) offset = ExtendReg(m, extend_type, shift, 64);
CheckFPEnabled64();
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

address = address + offset;

case memop of
  when MemOp_STORE
    data = V[t, datasize];
    Mem[address, datasize DIV 8, AccType_VEC] = data;
  when MemOp_LOAD
    data = Mem[address, datasize DIV 8, AccType_VEC];
    V[t, datasize] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.333   STUR (SIMD&FP)

Store SIMD&FP register (unscaled offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

8-bit variant
Applies when size == 00 && opc == 00.

STUR <Bt>, [<Xn|SP>{, #<simm>}]

16-bit variant
Applies when size == 01 && opc == 00.

STUR <Ht>, [<Xn|SP>{, #<simm>}]

32-bit variant
Applies when size == 10 && opc == 00.

STUR <St>, [<Xn|SP>{, #<simm>}]

64-bit variant
Applies when size == 11 && opc == 00.

STUR <Dt>, [<Xn|SP>{, #<simm>}]

128-bit variant
Applies when size == 00 && opc == 10.

STUR <Qt>, [<Xn|SP>{, #<simm>}]

Decode for all variants of this encoding

integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);

Assembler symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<6> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);

Operation

CheckFPEnabled64();
bits(64) address;
bits(datasize) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
address = address + offset;
case memop of
    when MemOp_STORE
        data = V[t, datasize];
        Mem[address, datasize DIV 8, AccType_VEC] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t, datasize] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
C7.2.334   SUB (vector)

Subtract (vector). This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

![Scalar Encoding](image)

**Encoding**

SUB <V>d>, <V>n>, <V>m>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');

Vector

![Vector Encoding](image)

**Encoding**

SUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');

**Assembler symbols**

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
- D when size = 11

The following encodings are reserved:
- size = 0x.
• size = 10.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;
V[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.335  **SUBHN, SUBHN2**

Subtract returning High Narrow. This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values.

The results are truncated. For rounded results, see **RSUBHN, RSUBHN2**.

The **SUBHN** instruction writes the vector to the lower half of the destination register and clears the upper half, while the **SUBHN2** instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Instruction Encoding](image)

**Encoding**

SUBHN\{2\} <Vd>,<Vn>,<Ta>, <Vm>,<Ta>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

**Assembler symbols**

2  
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- **[absent]** when Q = 0
- **[present]** when Q = 1

<Vd>  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb>  
Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- **8B** when size = 00, Q = 0
- **16B** when size = 00, Q = 1
- **4H** when size = 01, Q = 0
- **8H** when size = 01, Q = 1
- **2S** when size = 10, Q = 0
- **4S** when size = 10, Q = 1
The encoding size = 11, \( Q = x \) is reserved.

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Ta>\) Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- \( 8H \) when size = 00
- \( 4S \) when size = 01
- \( 2D \) when size = 10

The encoding size = 11 is reserved.

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

CheckFPAdvSIMDEnabled64();

\[ \text{bits}(2*\text{datasize}) \text{ operand1} = V[n, 2*\text{datasize}] ; \]

\[ \text{bits}(2*\text{datasize}) \text{ operand2} = V[m, 2*\text{datasize}] ; \]

\[ \text{bits(\text{datasize}) result} ; \]

\[ \text{integer round\_const} = \text{if round then 1} \ll (\text{esize} - 1) \text{ else 0} ; \]

\[ \text{bits(2*esize)} \text{ element1} ; \]

\[ \text{bits(2*esize)} \text{ element2} ; \]

\[ \text{bits(2*esize)} \text{ sum} ; \]

\[
\text{for e = 0 to elements-1} \\
\quad \text{element1} = \text{Elem[operand1, e, 2*esize]} ; \\
\quad \text{element2} = \text{Elem[operand2, e, 2*esize]} ; \\
\quad \text{if sub\_op then} \\
\quad \quad \text{sum} = \text{element1} - \text{element2} ; \\
\quad \text{else} \\
\quad \quad \text{sum} = \text{element1} + \text{element2} ; \\
\quad \quad \text{sum} = \text{sum} + \text{round\_const} ; \\
\quad \quad \text{Elem[result, e, esize]} = \text{sum<2*esize-1:esize>} ; \\
\]

\[ \text{Vpart[d, part, datasize]} = \text{result} ; \]

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.336  SUDOT (by element)

Dot product index form with signed and unsigned integers. This instruction performs the dot product of the four signed 8-bit integer values in each 32-bit element of the first source register with the four unsigned 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination vector.

From Armv8.2 to Armv8.5, this is an optional instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

Vector

( FEAT_I8MM )

Encoding

SUDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B.<index>

Decode for this encoding

if !HaveInt8MatMulExt() then UNDEFINED;
boolean op1_unsigned = (US == '1');
boolean op2_unsigned = (US == '0');
n = UInt(Rn);
m = UInt(M:Rm);
d = UInt(Rd);
i = UInt(H:L);
datasize = if Q == '1' then 128 else 64;
elements = datasize DIV 32;

Assembler symbols

<Vd>  Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
<Ta>  Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
   2S when Q = 0
   4S when Q = 1
<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb>  Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
   8B when Q = 0
   16B when Q = 1
<Vm>  Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
<index>  Is the immediate index of a 32-bit group of four 8-bit values in the range 0 to 3, encoded in the "H:L" fields.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(128) operand2 = V[m, 128];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
for e = 0 to elements-1
    bits(32) res = Elem[operand3, e, 32];
    for b = 0 to 3
        integer element1 = Int(Elem[operand1, 4*e+b, 8], op1_unsigned);
        integer element2 = Int(Elem[operand2, 4*i+b, 8], op2_unsigned);
        res = res + element1 * element2;
    Elem[result, e, 32] = res;
V[d, datasize] = result;
C7.2.337  SUQADD

Signed saturating Accumulate of Unsigned value. This instruction adds the unsigned integer values of the vector elements in the source SIMD&FP register to corresponding signed integer values of the vector elements in the destination SIMD&FP register, and writes the resulting signed integer values to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
<td>size 1 0 0 0 0 0 0 1 1 1 0</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SUQADD <V><d>, <V><n>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 1 1 1 0</td>
<td>size 1 0 0 0 0 0 0 1 1 1 0</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SUQADD <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
Assembler symbols

⟨V⟩  Is a width specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11

⟨d⟩  Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

⟨n⟩  Is the number of the SIMD&FP source register, encoded in the "Rn" field.

⟨Vd⟩ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

⟨T⟩  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  8B when size = 00, Q = 0
  16B when size = 00, Q = 1
  4H when size = 01, Q = 0
  8H when size = 01, Q = 1
  2S when size = 10, Q = 0
  4S when size = 10, Q = 1
  2D when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.

⟨Vn⟩ Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

    CheckFPAdvSIMDEnabled64();
    bits(datasize) operand = V[n, datasize];
    bits(datasize) result;

    bits(datasize) operand2 = V[d, datasize];
    integer op1;
    integer op2;
    boolean sat;

    for e = 0 to elements-1
        op1 = Int(Elem[operand, e, esize], !unsigned);
        op2 = Int(Elem[operand2, e, esize], unsigned);
        (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
        if sat then FPSR.QC = '1';
        V[d, datasize] = result;
C7.2.338  SXTL, SXTL2

Signed extend Long. This instruction duplicates each vector element in the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SXTL instruction extracts the source vector from the lower half of the source register. The SXTL2 instruction extracts the source vector from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is an alias of the SSHLL, SSHLL2 instruction. This means that:

- The encodings in this description are named to match the encodings of SSHLL, SSHLL2.
- The description of SSHLL, SSHLL2 gives the operational pseudocode for this instruction.

![Encoding](image)

**Encoding**

SXTL[2]  \(<Vd>\cdot<Ta>, <Vn>\cdot<Tb>\)

is equivalent to

SSHLL[2]  \(<Vd>\cdot<Ta>, <Vn>\cdot<Tb>, #0\)

and is the preferred disassembly when BitCount(immh) == 1.

**Assembler symbols**

2  
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd>  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  
Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

- 8H when immh = 0001
- 4S when immh = 001x
- 2D when immh = 01xx

See Advanced SIMD modified immediate on page C4-632 when immh = 0000.

The encoding immh = 1xxx is reserved.

<Vn>  
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb>  
Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

- 8B when immh = 0001, Q = 0
- 16B when immh = 0001, Q = 1
- 4H when immh = 001x, Q = 0
- 8H when immh = 001x, Q = 1
2S when immh = 01xx, Q = 0
4S when immh = 01xx, Q = 1
See *Advanced SIMD modified immediate* on page C4-632 when immh = 0000, Q = x.
The encoding immh = 1xxx, Q = x is reserved.

**Operation**

The description of **SSHLL, SSHLL2** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.339  TBL

Table vector Lookup. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the result for that lookup is 0. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Two register table variant
Applies when \( \text{len} == 01 \).

\[
\text{TBL} \ <Vd>.<Ta>, \ { <Vn>.16B, <Vn+1>.16B } , \ <Vm>.<Ta>
\]

Three register table variant
Applies when \( \text{len} == 10 \).

\[
\text{TBL} <Vd>.<Ta>, \ { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B } , \ <Vm>.<Ta>
\]

Four register table variant
Applies when \( \text{len} == 11 \).

\[
\text{TBL} \ <Vd>.<Ta>, \ { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B } , \ <Vm>.<Ta>
\]

Single register table variant
Applies when \( \text{len} == 00 \).

\[
\text{TBL} \ <Vd>.<Ta>, \ { <Vn>.16B } , \ <Vm>.<Ta>
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{integer } d & = \text{UInt}(\text{Rd}); \\
\text{integer } n & = \text{UInt}(\text{Rn}); \\
\text{integer } m & = \text{UInt}(\text{Rm}); \\
\text{integer } \text{datasize} & = \text{if } Q == '1' \text{ then 128 else 64}; \\
\text{integer } \text{elements} & = \text{datasize} \text{ DIV 8}; \\
\text{integer } \text{regs} & = \text{UInt}(\text{len}) + 1; \\
\text{boolean } \text{is_tbl} & = (\text{op} == '0');
\end{align*}
\]

Assembler symbols

\[
\begin{align*}
<Vd> & \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.} \\
<Ta> & \quad \text{Is an arrangement specifier, encoded in the "Q" field. It can have the following values:} \\
& \quad 88 \quad \text{when } Q = 0 \\
& \quad 168 \quad \text{when } Q = 1
\end{align*}
\]
<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.
For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m, datasize];
bits(128*regs) table = Zeros(128 * regs);
bits(datasize) result;
integer index;

// Create table from registers
for i = 0 to regs-1
    table<128*i+127:128*i> = V[n, 128];
    n = (n + 1) MOD 32;
result = if is_tbl then Zeros(datasize) else V[d, datasize];
for i = 0 to elements-1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];
V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.340   TBX

Table vector lookup extension. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the existing value in the vector element of the destination register is left unchanged. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Two register table variant
Applies when \( \text{len} == 01 \).
TBX \(<\text{Vd}>.<\text{Ta}>, \{ <\text{Vn}>.16B, <\text{Vn+1}>.16B \}, <\text{Vm}>.<\text{Ta}> \)

Three register table variant
Applies when \( \text{len} == 10 \).
TBX \(<\text{Vd}>.<\text{Ta}>, \{ <\text{Vn}>.16B, <\text{Vn+1}>.16B, <\text{Vn+2}>.16B \}, <\text{Vm}>.<\text{Ta}> \)

Four register table variant
Applies when \( \text{len} == 11 \).
TBX \(<\text{Vd}>.<\text{Ta}>, \{ <\text{Vn}>.16B, <\text{Vn+1}>.16B, <\text{Vn+2}>.16B, <\text{Vn+3}>.16B \}, <\text{Vm}>.<\text{Ta}> \)

Single register table variant
Applies when \( \text{len} == 00 \).
TBX \(<\text{Vd}>.<\text{Ta}>, \{ <\text{Vn}>.16B \}, <\text{Vm}>.<\text{Ta}> \)

Decode for all variants of this encoding
\[
\begin{align*}
\text{integer } d &= \text{UInt}(\text{Rd}); \\
\text{integer } n &= \text{UInt}(\text{Rn}); \\
\text{integer } m &= \text{UInt}(\text{Rm}); \\
\text{integer } \text{datasize} &= \text{if } Q == '1' \text{ then 128 else 64}; \\
\text{integer } \text{elements} &= \text{datasize DIV 8}; \\
\text{integer } \text{regs} &= \text{UInt}(\text{len}) + 1; \\
\text{boolean } \text{is_tbl} &= (op == '0');
\end{align*}
\]

Assembler symbols
\(<\text{Vd}>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<\text{Ta}>\) Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
\[
\begin{align*}
88 & \quad \text{when } Q = 0 \\
168 & \quad \text{when } Q = 1
\end{align*}
\]
<Vn> is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+2> is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

<Vn+3> is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> is the name of the SIMD&FP index register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bairs(datasize) indices = V[m, datasize];
bairs(128*regs) table = Zeros(128 * regs);
bairs(datasize) result;
integer index;

// Create table from registers
for i = 0 to regs-1
    table<128*i+127:128*i> = V[n, 128];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros(datasize) else V[d, datasize];
for i = 0 to elements-1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
    — The values of the data supplied in any of its registers.
    — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
    — The values of the data supplied in any of its registers.
    — The values of the NZCV flags.
C7.2.341   TRN1

Transpose vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

--- Note ---

By using this instruction with TRN2, a 2 x 2 matrix can be transposed.

The following figure shows the operation of TRN1 and TRN2 halfword operations where Q = 0.

TRN1.16

\[
\begin{array}{c}
\text{Vn} \\
3 & 2 & 1 & 0 \\
\text{Vd} \\
\text{Vm}
\end{array}
\]

TRN2.16

\[
\begin{array}{c}
\text{Vn} \\
3 & 2 & 1 & 0 \\
\text{Vd} \\
\text{Vm}
\end{array}
\]

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

TRN1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

```cpp
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - 8B when size = 00, Q = 0
  - 16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.342 TRN2

Transpose vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

Note

By using this instruction with TRN1, a 2 x 2 matrix can be transposed.

The following figure shows the operation of TRN1 and TRN2 halfword operations where Q = 0.

The encoding for this instruction is:

```
TRN2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - `8B` when size = 00, Q = 0
  - `16B` when size = 00, Q = 1
The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.343 UABA

Unsigned Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

UABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
   8B when size = 00, Q = 0
   16B when size = 00, Q = 1
   4H when size = 01, Q = 0
   8H when size = 01, Q = 1
   2S when size = 10, Q = 0
   4S when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
bias(esize) absdiff;
result = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  absdiff = Abs(element1-element2)<esize-1:0>;
  Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.344  UABAL, UABAL2

Unsigned Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABAL instruction extracts each source vector from the lower half of each source register. The UABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [16] [15 14 13] [12] [11 10 9] [5] [4] [3] [2] [1] [0]

0 Q 1 0 1 1 0 | size 1 | Rm 0 1 0 1 0 0 | Rn Rd
```

**Encoding**

UABAL[2] <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

**Decode for this encoding**

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

**Assembler symbols**

2  
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd>  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  
Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

<Vn>  
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d, 2*datasize] else Zeros(2 * datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.345   UABD

Unsigned Absolute Difference (vector). This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

```
UABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler symbols

`<Vd>`  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>`  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

`<Vn>`  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>`  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;
```
result = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.346   UABDL, UABDL2

Unsigned Absolute Difference Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABDL instruction extracts each source vector from the lower half of each source register. The UABDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

UABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');

Assembler symbols

2   Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
   [absent] when Q = 0
   [present] when Q = 1

<Vd>   Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>   Is an arrangement specifier, encoded in the "size" field. It can have the following values:
       8H when size = 00
       4S when size = 01
       2D when size = 10
       The encoding size = 11 is reserved.

<Vn>   Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bites(2*esize) absdiff;
result = if accumulate then V[d, 2*datasize] else Zeros(2 + datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d, 2*datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.347  UADALP

Unsigned Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

UADALP <Vd>.<Ta>, <Vn>.<Tb>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
    4H when size = 00, Q = 0
    8H when size = 00, Q = 1
    2S when size = 01, Q = 0
    4S when size = 01, Q = 1
    1D when size = 10, Q = 0
    2D when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
    8B when size = 00, Q = 0
    16B when size = 00, Q = 1
    4H when size = 01, Q = 0
    8H when size = 01, Q = 1
    2S when size = 10, Q = 0
    4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.
**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d, datasize];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)\mod{2*esize-1};
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;

V[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.348  UADDL, UADDL2

Unsigned Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UADDL instruction extracts each source vector from the lower half of each source register. The UADDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

UADDL\{2\} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0
[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:

8H when size = 00
4S when size = 01
2D when size = 10

The encoding size = 11 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

88 when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<\Vm> is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

checkfpadvSIMDenabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  if sub_op then
    sum = element1 - element2;
  else
    sum = element1 + element2;
  Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.349 UADDLP

Unsigned Add Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

UADDLP <Vd>.<Ta>, <Vn>.<Tb>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
   4H when size = 00, Q = 0
   8H when size = 00, Q = 1
   2S when size = 01, Q = 0
   4S when size = 01, Q = 1
   1D when size = 10, Q = 0
   2D when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
   8B when size = 00, Q = 0
   16B when size = 00, Q = 1
   4H when size = 01, Q = 0
   8H when size = 01, Q = 1
   2S when size = 10, Q = 0
   4S when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d, datasize];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;

V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.350 UADDLV

Unsigned sum Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

UADDLV <V><d>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:
   H when size = 00
   S when size = 01
   D when size = 10
The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
   8B when size = 00, Q = 0
   16B when size = 00, Q = 1
   4H when size = 01, Q = 0
   8H when size = 01, Q = 1
   4S when size = 10, Q = 1
The following encodings are reserved:
   * size = 10, Q = 0.
   * size = 11, Q = x.
Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n, datasize];
integer sum;

sum = \text{Int(Elem[operand, 0, esize], unsigned)};
for e = 1 to elements-1
  sum = sum + \text{Int(Elem[operand, e, esize], unsigned)};

V[d, 2*esize] = \text{sum<2*esize-1:0>};

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.351  UADDW, UADDW2

Unsigned Add Wide. This instruction adds the vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register. All the values in this instruction are unsigned integer values.

The UADDW instruction extracts vector elements from the lower half of the second source register. The UADDW2 instruction extracts vector elements from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

UADDW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

Assembler symbols

2   Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
    [absent]  when Q = 0
    [present]  when Q = 1

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  Is an arrangement specifier, encoded in the "size" field. It can have the following values:
    8H  when size = 00
    4S  when size = 01
    2D  when size = 10

The encoding size = 11 is reserved.

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d, 2*datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.352  UCVTF (vector, fixed-point)

Unsigned fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>19 18 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 1 1 0</td>
<td>!=0000</td>
<td>1 1 1 0 0 1</td>
<td>Rn</td>
</tr>
</tbody>
</table>
```

**Encoding**

UCVTF <V><db>, <V><n>, #<fbits>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh IN ('000x') || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Vector

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>19 18 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 Q 1 1 1 1 1 1 0</td>
<td>!=0000</td>
<td>1 1 1 0 0 1</td>
<td>Rn</td>
</tr>
</tbody>
</table>
```

**Encoding**

UCVTF <Vd>.<t>, <Vn>.<t>, #<fbits>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh IN {'000x'} then SEE "Advanced SIMD modified immediate";
if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = esize;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
```
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);

Assembler symbols

<V> Is a width specifier, encoded in the "immm" field. It can have the following values:

H when immh = 001x
S when immh = 01xx
D when immh = 1xxx

The encoding immh = 000x is reserved.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

</d> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immm:Q" field. It can have the following values:

4H when immh = 001x, Q = 0
8H when immh = 001x, Q = 1
2S when immh = 01xx, Q = 0
4S when immh = 01xx, Q = 1
2D when immh = 1xxx, Q = 1

See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.

The following encodings are reserved:

- immh = 0001, Q = x.
- immh = 1xxx, Q = x.

</n> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in the "immm:immb" field. It can have the following values:

(32-Uint(immm:immb)) when immh = 001x
(64-Uint(immm:immb)) when immh = 01xx
(128-Uint(immm:immb)) when immh = 1xxx

The encoding immh = 000x is reserved.

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in the "immm:immb" field. It can have the following values:

(32-Uint(immm:immb)) when immh = 001x
(64-Uint(immm:immb)) when immh = 01xx
(128-Uint(immm:immb)) when immh = 1xxx

See Advanced SIMD modified immediate on page C4-632 when immh = 0000.

The encoding immh = 0001 is reserved.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(size) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, fpcr, rounding, esize);
V[d, 128] = result;
C7.2.353  UCVTF (vector, integer)

Unsigned integer Convert to Floating-point (vector). This instruction converts each element in a vector from an unsigned integer value to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Scalar half precision

(FEAT_FP16)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0 0 1 1 1 0 1 1 0 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Encoding

UCVTF <Hd>, <Hn>

Decode for this encoding

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0 0 1 0 0 0 0 1 1 1 0 1 1 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Encoding

UCVTF <V<d>, <V<n>

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```
Vector half precision

(FEAT_FP16)

Encoding

UCVTF <Vd>.<T>, <Vn>.<T>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Vector single-precision and double-precision

Encoding

UCVTF <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in the "sz" field. It can have the following values:

S when sz = 0

D when sz = 1

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in the "Q" field. It can have the following values:

4H when Q = 0
8H when Q = 1

For the single-precision and double-precision variant: is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0
4S when sz = 0, Q = 1
2D when sz = 1, Q = 1

The encoding sz = 1, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

FPCRType fpcr = FPCR[];
boolean merge = elements == 1 &amp; IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

FPRounding rounding = FPRoundingMode(fpcr);
bits(esize) element;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FixedToFP(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
C7.2.354   **UCVTF (scalar, fixed-point)**

Unsigned fixed-point Convert to Floating-point (scalar). This instruction converts the unsigned value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

```
31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 | 10 9 | 5 4 | 0 |
  sf 0 0 1 1 1 0 | type 0 0 0 0 1 1 | scale | Rn | Rd |

mode opcode
```

### 32-bit to half-precision variant

Applies when `sf == 0 && ftype == 11`.

```
UCVTF <Hd>, <Wn>, #<fbits>
```

### 32-bit to single-precision variant

Applies when `sf == 0 && ftype == 00`.

```
UCVTF <Sd>, <Wn>, #<fbits>
```

### 32-bit to double-precision variant

Applies when `sf == 0 && ftype == 01`.

```
UCVTF <Dd>, <Wn>, #<fbits>
```

### 64-bit to half-precision variant

Applies when `sf == 1 && ftype == 11`.

```
UCVTF <Hd>, <Xn>, #<fbits>
```

### 64-bit to single-precision variant

Applies when `sf == 1 && ftype == 00`.

```
UCVTF <Sd>, <Xn>, #<fbits>
```

### 64-bit to double-precision variant

Applies when `sf == 1 && ftype == 01`.

```
UCVTF <Dd>, <Xn>, #<fbits>
```

### Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
PPRounding rounding;
```
case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  if sf == '0' && scale<> == '0' then UNDEFINED;
  integer fracbites = 64 - UInt(scale);
  rounding = FPRoundingMode(FPCR[]);

Assembler symbols

<DD> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<HD> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<SF> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<XN> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<WN> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<FBITS> For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".
For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

Operation

CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

intval = X[n, intsize];
fltval = if merge then V[d, fsize] else Zeros(fsize);
Elem[fltval, 0, fltsize] = FixedToFP(intval, fracbites, TRUE, fpcr, rounding, fltsize);
V[d, fsize] = fltval;
C7.2.355  UCVTF (scalar, integer)

Unsigned integer Convert to Floating-point (scalar). This instruction converts the unsigned integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page A1-64.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

32-bit to half-precision variant
Applies when $sf == 0 \&\& ftype == 11$.
UCVTF $<Hd>, <Wn>$

32-bit to single-precision variant
Applies when $sf == 0 \&\& ftype == 00$.
UCVTF $<Sd>, <Wn>$

32-bit to double-precision variant
Applies when $sf == 0 \&\& ftype == 01$.
UCVTF $<Dd>, <Wn>$

64-bit to half-precision variant
Applies when $sf == 1 \&\& ftype == 11$.
UCVTF $<Hd>, <Xn>$

64-bit to single-precision variant
Applies when $sf == 1 \&\& ftype == 00$.
UCVTF $<Sd>, <Xn>$

64-bit to double-precision variant
Applies when $sf == 1 \&\& ftype == 01$.
UCVTF $<Dd>, <Xn>$

Decode for all variants of this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;
```

---

A64 Advanced SIMD and Floating-point Instruction Descriptions
C7.2 Alphabetical list of A64 Advanced SIMD and floating-point instructions

ARM DDI 0487I.a
Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential
case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  rounding = FPRoundingMode(FPCR());

Assembler symbols

<Db> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

CheckFPEnabled64();

FPCRType fpcr = FPCR();
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;
intval = X[n, intsize];
fltval = if merge then V[d, fsize] else Zeros(fsize);
Elem[fltval, 0, fltsize] = FixedToFP(intval, 0, TRUE, fpcr, rounding, fltsize);
V[d, fsize] = fltval;
C7.2.356   UDOT (by element)

Dot Product unsigned arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

______ Note ________
ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector

(FEAT_DotProd)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

UDOT <Vd>,<Ta>, <Vn>,<Tb>, <Vm>,4B<index>

Decode for this encoding

if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index = UInt(H:L);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
    2S when Q = 0
    4S when Q = 1
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
    8B when Q = 0
    16B when Q = 1
<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
<index> Is the element index, encoded in the "H:L" fields.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(128) operand2 = V[m, 128];
bits(datasize) result = V[d, datasize];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
        res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
    V[d, datasize] = result;
```
C7.2.357   UDOT (vector)

Dot Product unsigned arithmetic (vector). This instruction performs the dot product of the four unsigned 8-bit elements in each 32-bit element of the first source register with the four unsigned 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an optional instruction. From Armv8.4 it is mandatory for all implementations to support it.

--- Note ---
ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector

(FEAT_DotProd)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

UDOT <Vd>..<Ta>., <Vn>..<Tb>., <Vm>..<Tb>

Decode for this encoding

if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
2S when Q = 0
4S when Q = 1
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
8B when Q = 0
16B when Q = 1
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

result = V[d, datasize];
for e = 0 to elements-1
  integer res = 0;
  integer element1, element2;
  for i = 0 to 3
    if signed then
      element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
      element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
    else
      element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
      element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
    res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
  V[d, datasize] = result;
C7.2.358   UHADD

Unsigned Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see URHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5  4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
</tr>
<tr>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U
```

Decode for this encoding

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - `8B` when size = 00, Q = 0
  - `16B` when size = 00, Q = 1
  - `4H` when size = 01, Q = 0
  - `8H` when size = 01, Q = 1
  - `2S` when size = 10, Q = 0
  - `4S` when size = 10, Q = 1
  The encoding size = 11, Q = x is reserved.
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```java
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
```
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = (element1 + element2) >> 1;
    Elem[result, e, esize] = sum<esize-1:0>;
V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.359 UHSUB

Unsigned Halving Subtract. This instruction subtracts the vector elements in the second source SIMD&FP register from the corresponding vector elements in the first source SIMD&FP register, shifts each result right one bit, places each result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

```
UHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler symbols

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - 8B when size = 00, Q = 0
  - 16B when size = 00, Q = 1
  - 4H when size = 01, Q = 0
  - 8H when size = 01, Q = 1
  - 2S when size = 10, Q = 0
  - 4S when size = 10, Q = 1
  - The encoding size = 11, Q = x is reserved.
- `<Vn>` is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
for e = 0 to elements-1
```
element1 = Int(Elem[operand1, e, esize], unsigned);

element2 = Int(Elem[operand2, e, esize], unsigned);

diff = (element1 - element2) >> 1;

Elem[result, e, esize] = diff<esize-1:0>;

V[d, datasize] = result;

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.360  UMAX

Unsigned Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

UMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');

Assembler symbols

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  8B  when size = 00, Q = 0
  16B when size = 00, Q = 1
  4H  when size = 01, Q = 0
  8H  when size = 01, Q = 1
  2S  when size = 10, Q = 0
  4S  when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnable64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
  Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.361 UMAXP

Unsigned Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding
UMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');

Assembler symbols
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
8B when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
b bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements - 1
  element1 = Int(Elem[concat, 2*e, esize], unsigned);
  element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
  maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
  Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.362 UMAXV

Unsigned Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

UMAXV <V><d>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean min = (op == '1');

Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:
- B when size = 00
- H when size = 01
- S when size = 10

The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 4S when size = 10, Q = 1

The following encodings are reserved:
- size = 10, Q = 0.
- size = 11, Q = x.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);
V[d, esize] = maxmin<esize-1:0>;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.363 UMIN

Unsigned Minimum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the smaller of each of the two unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

```
UMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 « UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - 8B when size = 00, Q = 0
  - 16B when size = 00, Q = 1
  - 4H when size = 01, Q = 0
  - 8H when size = 01, Q = 1
  - 2S when size = 10, Q = 0
  - 4S when size = 10, Q = 1
  The encoding size = 11, Q = x is reserved.
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bias(datasize) operand1 = V[n, datasize];
bias(datasize) operand2 = V[m, datasize];
bias(datasize) result;
ingter element1;
ingter element2;
ingter maxmin;
```
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.364 UMINP

Unsigned Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

UMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.365  UMINV

Unsigned Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

UMINV <V><d>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean min = (op == '1');

Assembler symbols

<V> Is the destination width specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
The encoding size = 11 is reserved.
<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
8B when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
4S when size = 10, Q = 1
The following encodings are reserved:
• size = 10, Q = 0.
• size = 11, Q = x.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);
V[d, esize] = maxmin<esize-1:0>;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.366  **UMLAL, UMLAL2 (by element)**

Unsigned Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register. The UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Encoding

UMLAL\{2\} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

### Decode for this encoding

```plaintext
integer idxdsiz = if \texttt{H} == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

### Assembler symbols

- **Z**  
  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
  - [absent] when \texttt{Q} = 0
  - [present] when \texttt{Q} = 1

- **<Vd>**  
  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- **<Ta>**  
  Is an arrangement specifier, encoded in the "size" field. It can have the following values:
  - 4S when \texttt{size} = 01
  - 2D when \texttt{size} = 10
The following encodings are reserved:
- size = 00.
- size = 11.

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Tb>\) Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The following encodings are reserved:
- size = 00, Q = x.
- size = 11, Q = x.

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:
- 0:Rm when size = 01
- M:Rm when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

Restricted to V0-V15 when element size <Ts> is H.

\(<Ts>\) Is an element size specifier, encoded in the "size" field. It can have the following values:
- H when size = 01
- S when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

\(<index>\) Is the element index, encoded in the "size:L:H:M" field. It can have the following values:
- H:L:M when size = 01
- H:L when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<<2*esize-1:0>;
```
if sub_op then
    Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
else
    Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;
V[d, 2*datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.367  UMLAL, UMLAL2 (vector)

Unsigned Multiply-Add Long (vector). This instruction multiplies the vector elements in the lower or upper half of the first source SIMD&FP register by the corresponding vector elements of the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register. The UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

UMLAL\{2\} \langle Vd\rangle, \langle Ta\rangle, \langle Vn\rangle, \langle Tb\rangle, \langle Vm\rangle, \langle Tb\rangle

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

Assembler symbols

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
   [absent]  when Q = 0
   [present]  when Q = 1

\langle Vd\rangle  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\langle Ta\rangle  Is an arrangement specifier, encoded in the "size" field. It can have the following values:
   8H  when size = 00
   4S  when size = 01
   2D  when size = 10

The encoding size = 11 is reserved.

\langle Vn\rangle  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\langle Tb\rangle  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
   8B  when size = 00, Q = 0
   16B when size = 00, Q = 1
4H  
8H  
2S  
4S  
The encoding size = 11, Q = x is reserved.

<\m>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

- `CheckFPAdvSIMDEnabled64();`
- `bits(datasize) operand1 = Vpart[n, part, datasize];`
- `bits(datasize) operand2 = Vpart[m, part, datasize];`
- `bits(2*datasize) operand3 = V[d, 2*datasize];`
- `bits(2*datasize) result;`
- `integer element1;`
- `integer element2;`
- `bits(2*esize) product;`
- `bits(2*esize) accum;`

```
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;
V[d, 2*datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.368  UMLSL, UMLSL2 (by element)

Unsigned Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLSL instruction extracts vector elements from the lower half of the first source register. The UMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');

Assembler symbols

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
  [absent]  when Q = 0
  [present]  when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:
  4S  when size = 01
  2D  when size = 10
The following encodings are reserved:
- size = 00.
- size = 11.

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Tb>\) Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The following encodings are reserved:
- size = 00, Q = x.
- size = 11, Q = x.

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:
- 0:Rm when size = 01
- M:Rm when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

Restricted to V0-V15 when element size \(<Ts>\) is H.

\(<Ts>\) Is an element size specifier, encoded in the "size" field. It can have the following values:
- H when size = 01
- S when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

\(<index>\) Is the element index, encoded in the "size:L:H:M" field. It can have the following values:
- H:L:M when size = 01
- H:L when size = 10

The following encodings are reserved:
- size = 00.
- size = 11.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  product += (element1*element2)<2*esize-1:0>;
```
if sub_op then
    Elem[result, e, 2*size] = Elem[operand3, e, 2*size] - product;
else
    Elem[result, e, 2*size] = Elem[operand3, e, 2*size] + product;

V[d, 2*datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.369  **UMLSL, UMLSL2 (vector)**

Unsigned Multiply-Subtract Long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The **UMLSL** instruction extracts each source vector from the lower half of each source register. The **UMLSL2** instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>19 18 17 16 15 14 13 12 11  10  9</th>
<th>8  7  6  5  4  3  2  1  0  0  0  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0   1   0   0   1   1   1   0</td>
<td>size = 1</td>
<td>Rm = 1</td>
</tr>
<tr>
<td>1   0   1   0   0   0   0   0</td>
<td>Rn = 0</td>
<td>Rd = 0</td>
</tr>
</tbody>
</table>
```

**Encoding**

**UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>**

**Decode for this encoding**

```plaintext```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

**Assembler symbols**

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[-] absent when Q = 0
[ ] present when Q = 1

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  Is an arrangement specifier, encoded in the "size" field. It can have the following values:

<table>
<thead>
<tr>
<th>8H</th>
<th>4S</th>
<th>2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>when size = 00</td>
<td>when size = 01</td>
<td>when size = 10</td>
</tr>
</tbody>
</table>

The encoding size = 11 is reserved.

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

<table>
<thead>
<tr>
<th>88</th>
<th>168</th>
</tr>
</thead>
<tbody>
<tr>
<td>when size = 00, Q = 0</td>
<td>when size = 00, Q = 1</td>
</tr>
</tbody>
</table>
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<\m> is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  product = (element1*element2)<<2*esize-1:0>;
  if sub_op then
    accum = Elem[operand3, e, 2*esize] - product;
  else
    accum = Elem[operand3, e, 2*esize] + product;
  Elem[result, e, 2*esize] = accum;
V[d, 2*datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.370   UMMLA (vector)

Unsigned 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of unsigned 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an optional instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

Vector

(_FEAT_I8MM)

Encoding

UMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

Decode for this encoding

if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);

Assembler symbols

<Vd>        Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
<Vn>        Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>        Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n, 128];
bits(128) operand2 = V[m, 128];
bits(128) addend = V[d, 128];
V[d, 128] = MatMulAdd(addend, operand1, operand2, TRUE, TRUE);
C7.2.371 UMOV

Unsigned Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (to general). See Alias conditions for details of when each alias is preferred.

### 32-bit variant

Applies when \( Q == 0 \).

\[
\text{UMOV} \ <Wd>, \ <Vn>.<Ts>[<index>]
\]

### 64-bit variant

Applies when \( Q == 1 \) \&\& \( \text{imm5} == \text{x1000} \).

\[
\text{UMOV} \ <Xd>, \ <Vn>.<Ts>[<index>]
\]

**Decode for all variants of this encoding**

\[
\text{integer } d = \text{UInt}(Rd);
\text{integer } n = \text{UInt}(Rn);
\]

\[
\text{integer } size;
\text{case } \text{Q:imm5 of}
\text{when '0xxxx1' size = 0; \ // UMOV Wd, Vn.B}
\text{when '0xxx10' size = 1; \ // UMOV Wd, Vn.H}
\text{when '0xx100' size = 2; \ // UMOV Wd, Vn.S}
\text{when '1x1000' size = 3; \ // UMOV Xd, Vn.D}
\text{otherwise UNDEFINED;}
\]

\[
\text{integer idxdsize = if } \text{imm5<4> == '1' then 128 else 64;}
\text{integer index = UInt(imm5<4:size+1>);}
\text{integer esize = 8 << size;}
\text{integer datasize = if } Q == '1' \text{ then 64 else 32;}
\]

**Alias conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (to general)</td>
<td>( \text{imm5} == \text{'x1000'} )</td>
</tr>
<tr>
<td>MOV (to general)</td>
<td>( \text{imm5} == \text{'xx100'} )</td>
</tr>
</tbody>
</table>

**Assembler symbols**

\[
<Wd>
\]

Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ts> For the 32-bit variant: is an element size specifier, encoded in the "imm5" field. It can have the following values:

- B when \( \text{imm5} = xxxx1 \)
- H when \( \text{imm5} = xxx10 \)
- S when \( \text{imm5} = xx100 \)

The encoding \( \text{imm5} = xx000 \) is reserved.

For the 64-bit variant: is an element size specifier, encoded in the "imm5" field. It can have the following values:

- D when \( \text{imm5} = x1000 \)

The following encodings are reserved:

- \( \text{imm5} = x0000 \).
- \( \text{imm5} = xxxx1 \).
- \( \text{imm5} = xxx10 \).
- \( \text{imm5} = xx100 \).

<index> For the 32-bit variant: is the element index encoded in the "imm5" field. It can have the following values:

- \( \text{imm5}<4:1> \) when \( \text{imm5} = xxxx1 \)
- \( \text{imm5}<4:2> \) when \( \text{imm5} = xxx10 \)
- \( \text{imm5}<4:3> \) when \( \text{imm5} = xx100 \)

The encoding \( \text{imm5} = xx000 \) is reserved.

For the 64-bit variant: is the element index encoded in "imm5<4>".

**Operation**

```plaintext
if index == 0 then
    CheckFPEnabled64();
else
    CheckFPAdvSIMDEnabled64();

bits(idxdsiz) operand = V[n, idxdsiz];

X[d, datasiz] = ZeroExtend(Elem[operand, index, esize], datasiz);
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.372  UMULL, UMULL2 (by element)

Unsigned Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMULL instruction extracts vector elements from the lower half of the first source register. The UMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

Decode for this encoding

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler symbols

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
  [absent] when Q = 0
  [present] when Q = 1

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  Is an arrangement specifier, encoded in the "size" field. It can have the following values:
  4S  when size = 01
  2D  when size = 10
The following encodings are reserved:
  •  size = 00.
• size = 11.

\(<Vn>\)
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Tb>\)
Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

\[ 4H \quad \text{when } size = 01, Q = 0 \]
\[ 8H \quad \text{when } size = 01, Q = 1 \]
\[ 2S \quad \text{when } size = 10, Q = 0 \]
\[ 4S \quad \text{when } size = 10, Q = 1 \]

The following encodings are reserved:

• size = 00, Q = x.
• size = 11, Q = x.

\(<Vm>\)
Is the name of the second SIMD&FP source register, encoded in the "size:M:Rm" field. It can have the following values:

\[ 0:Rm \quad \text{when } size = 01 \]
\[ M:Rm \quad \text{when } size = 10 \]

The following encodings are reserved:

• size = 00.
• size = 11.

Restricted to V0-V15 when element size \(<Ts>\) is H.

\(<Ts>\)
Is an element size specifier, encoded in the "size" field. It can have the following values:

\[ H \quad \text{when } size = 01 \]
\[ S \quad \text{when } size = 10 \]

The following encodings are reserved:

• size = 00.
• size = 11.

\(<index>\)
Is the element index, encoded in the "size:L:H:M" field. It can have the following values:

\[ H:L:M \quad \text{when } size = 01 \]
\[ H:L \quad \text{when } size = 10 \]

The following encodings are reserved:

• size = 00.
• size = 11.

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  product = (element1*element2)<2*esize-1:0>;
  Elem[result, e, 2*esize] = product;

V[d, 2*datasize] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.373 UMULL, UMULL2 (vector)

Unsigned Multiply long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The UMULL instruction extracts each source vector from the lower half of each source register. The UMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
[absent] when Q = 0
[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:
8H when size = 00
4S when size = 01
2D when size = 10

The encoding size = 11 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
88 when size = 00, Q = 0
168 when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<\m> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
b década(size) operand1 = Vpart[n, part, datasize];
b década(size) operand2 = Vpart[m, part, datasize];
b década(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1*element2)<2*esize-1:0>;

V[d, 2*datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.374  UQADD

Unsigned saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0</td>
<td>Rm</td>
<td>0 0 0 1 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

UQADD <V><d>, <V><n>, <V><m>

**Decode for this encoding**

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- integer esize = 8 << UInt(size);
- integer datasize = esize;
- integer elements = 1;
- boolean unsigned = (U == '1');

**Vector**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 1 1 1 0</td>
<td>Rm</td>
<td>0 0 0 1 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

UQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- if size:Q == '110' then UNDEFINED;
- integer esize = 8 << UInt(size);
- integer datasize = if Q == '1' then 128 else 64;
- integer elements = datasize DIV esize;
- boolean unsigned = (U == '1');

**Assembler symbols**

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
S when size = 10
D when size = 11

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  8B when size = 00, Q = 0
  16B when size = 00, Q = 1
  4H when size = 01, Q = 0
  8H when size = 01, Q = 1
  2S when size = 10, Q = 0
  4S when size = 10, Q = 1
  2D when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d, datasize] = result;
```

Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential
C7.2.375  **UQRSHL**

Unsigned saturating Rounding Shift Left (register). This instruction takes each vector element of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see UQSHL (immediate).

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 1 0</td>
<td>Rm</td>
<td>0 1 0 1 0 1 1</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**Encoding**

UQRSHL <V><d>, <V><n>, <V><m>

**Decode for this encoding**

```python
def decode_encoding_UQRSHL(data):
    # Integer calculations
    d = UInt(Rd);
    n = UInt(Rn);
    m = UInt(Rm);
    esize = 8 << UInt(size);
    datasize = esize;
    elements = 1;
    unsigned = (U == '1');
    rounding = (R == '1');
    saturating = (S == '1');
    # Check for undefined
    if S == '0' && size != '11' then UNDEFINED;
```

**Vector**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 1 0 1 1 1 0</td>
<td>Rm</td>
<td>0 1 0 1 1 1</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**Encoding**

UQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```python
def decode_encoding_UQRSHL(data):
    # Integer calculations
    d = UInt(Rd);
    n = UInt(Rn);
    m = UInt(Rm);
    if size:Q == '110' then UNDEFINED;
    esize = 8 << UInt(size);
    datasize = if Q == '1' then 128 else 64;
    elements = datasize DIV esize;
    unsigned = (U == '1');
```
boolean rounding = (R == '1');
boolean saturating = (S == '1');

**Assembler symbols**

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1
- 2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        if rounding then
            element = (element + (1 << (shift - 1))) >> shift;
        else
            element = element >> shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;
V[d, datasize] = result;
C7.2.376 UQRSHRN, UQRSHRN2

Unsigned saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see UQSHRN, UQSHRN2.

The UQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>19 18 16 15 14 13 12 11 10 9</th>
<th>5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0</td>
<td>!=0000 1 0 0 1 1 Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Encoding

UQRSHRN <Vb><d>, <Va><n>, #<shift>

Decode for this encoding

integer d = UInt(Rd);
n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>19 18 16 15 14 13 12 11 10 9</th>
<th>5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 1 1 1 0</td>
<td>!=0000 1 0 0 1 1 Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Encoding

UQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

Decode for this encoding

integer d = UInt(Rd);
n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');

Assembler symbols

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0
[present] when Q = 1

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<7b>
Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when immh = 0001, Q = 0
16B when immh = 0001, Q = 1
4H when immh = 001x, Q = 0
8H when immh = 001x, Q = 1
2S when immh = 01xx, Q = 0
4S when immh = 01xx, Q = 1

See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = x is reserved.

<Vn>
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<7a>
Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

8H when immh = 0001
4S when immh = 001x
2D when immh = 01xx

See Advanced SIMD modified immediate on page C4-632 when immh = 0000.

The encoding immh = 1xxx is reserved.

<Vb>
Is the destination width specifier, encoded in the "immh" field. It can have the following values:

B when immh = 0001
H when immh = 001x
S when immh = 01xx

The following encodings are reserved:

• immh = 0000.
• immh = 1xxx.

<d>
Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va>
Is the source width specifier, encoded in the "immh" field. It can have the following values:

H when immh = 0001
S when immh = 001x
D when immh = 01xx
The following encodings are reserved:
- \( \text{immh} = 0000 \).
- \( \text{immh} = 1xxx \).

\(<n>\) is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<\text{shift}>\) For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:
- \(16\text{-UInt}(\text{immh:immb})\) when \(\text{immh} = 0001\)
- \(32\text{-UInt}(\text{immh:immb})\) when \(\text{immh} = 001x\)
- \(64\text{-UInt}(\text{immh:immb})\) when \(\text{immh} = 01xx\)

The following encodings are reserved:
- \( \text{immh} = 0000 \).
- \( \text{immh} = 1xxx \).

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:
- \(16\text{-UInt}(\text{immh:immb})\) when \(\text{immh} = 0001\)
- \(32\text{-UInt}(\text{immh:immb})\) when \(\text{immh} = 001x\)
- \(64\text{-UInt}(\text{immh:immb})\) when \(\text{immh} = 01xx\)

See Advanced SIMD modified immediate on page C4-632 when \(\text{immh} = 0000\).

The encoding \(\text{immh} = 1xxx\) is reserved.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';
Vpart[d, part, datasize] = result;
```
C7.2.377  UQSHL (immediate)

Unsigned saturating Shift Left (immediate). This instruction takes each vector element in the source SIMD&FP register, shifts it by an immediate value, places the results in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>19 18 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Encoding

UQSHL <V><d>, <V><n>, #<shift>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>19 18 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

UQSHL <Vd>.<T>, <Vn>.<T>, #<shift>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<>Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;  
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;  
boolean dst_unsigned;  

case op:U of  
when '00' UNDEFINED;  
when '01' src_unsigned = FALSE; dst_unsigned = TRUE;  
when '10' src_unsigned = FALSE; dst_unsigned = FALSE;  
when '11' src_unsigned = TRUE; dst_unsigned = TRUE;

Assembler symbols

<\V> Is a width specifier, encoded in the "immh" field. It can have the following values:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>immh = 0001</td>
</tr>
<tr>
<td>H</td>
<td>immh = 001x</td>
</tr>
<tr>
<td>S</td>
<td>immh = 01xx</td>
</tr>
<tr>
<td>D</td>
<td>immh = 1xxx</td>
</tr>
</tbody>
</table>

The encoding immh = 0000 is reserved.

<\d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<\Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8B</td>
<td>immh = 0001, Q = 0</td>
</tr>
<tr>
<td>16B</td>
<td>immh = 0001, Q = 1</td>
</tr>
<tr>
<td>4H</td>
<td>immh = 001x, Q = 0</td>
</tr>
<tr>
<td>8H</td>
<td>immh = 001x, Q = 1</td>
</tr>
<tr>
<td>2S</td>
<td>immh = 01xx, Q = 0</td>
</tr>
<tr>
<td>4S</td>
<td>immh = 01xx, Q = 1</td>
</tr>
<tr>
<td>2D</td>
<td>immh = 1xxx, Q = 1</td>
</tr>
</tbody>
</table>

See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

<\Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<\shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

- (UInt(immh:immb)-8) when immh = 0001
- (UInt(immh:immb)-16) when immh = 001x
- (UInt(immh:immb)-32) when immh = 01xx
- (UInt(immh:immb)-64) when immh = 1xxx

The encoding immh = 0000 is reserved.

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in the "immh:immb" field. It can have the following values:

- (UInt(immh:immb)-8) when immh = 0001
- (UInt(immh:immb)-16) when immh = 001x
- (UInt(immh:immb)-32) when immh = 01xx
\((\text{UInt}(\text{immh:imb})-64)\) when \text{immh} = 1xxx

See Advanced SIMD modified immediate on page C4-632 when \text{immh} = 0000.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```
C7.2.378  UQSHL (register)

Unsigned saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSCR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0</td>
<td>size 1</td>
<td>Rm 0 1 0</td>
<td>0 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
</tr>
<tr>
<td>U R S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

UQSHL <V><d>, <V><n>, <V><m>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;

Vector

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 1 0</td>
<td>size 1</td>
<td>Rm 0 1 0</td>
<td>0 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
</tr>
<tr>
<td>U R S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

UQSHL <Vd>.<T>, <Vm>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

**Assembler symbols**

- `<V>` is a width specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11

- `<d>` is the number of the SIMD&FP destination register, in the "Rd" field.
- `<n>` is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<m>` is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - 8B when size = 00, Q = 0
  - 16B when size = 00, Q = 1
  - 4H when size = 01, Q = 0
  - 8H when size = 01, Q = 1
  - 2S when size = 10, Q = 0
  - 4S when size = 10, Q = 1
  - 2D when size = 11, Q = 1
  The encoding size = 11, Q = 0 is reserved.
- `<Vn>` is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;
for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        if rounding then
            element = (element + (1 << (shift - 1))) >> shift;
        else
            element = element >> shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;
```

---

**Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.**

**Non-Confidential**
V[d, datasize] = result;
C7.2.379  UQSHRN, UQSHRN2

Unsigned saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see UQRSHRN, UQRSHRN2.

The UQSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>19 18 16 15 14 13 12</th>
<th>11 10 9 5 4 0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0</td>
<td>!=0000</td>
<td>immh</td>
<td>1 0 0 1 0 1 1</td>
</tr>
</tbody>
</table>

Encoding

UQSHRN <Vb><d>, <Va><n>, #<shift>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>19 18 16 15 14 13 12</th>
<th>11 10 9 5 4 0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 1 0 1 1 1 1 0</td>
<td>!=0000</td>
<td>immh</td>
<td>1 0 0 1 0 1 1</td>
</tr>
</tbody>
</table>

Encoding

UQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0
[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8B</td>
<td>immh = 0001, Q = 0</td>
</tr>
<tr>
<td>16B</td>
<td>immh = 0001, Q = 1</td>
</tr>
<tr>
<td>4H</td>
<td>immh = 001x, Q = 0</td>
</tr>
<tr>
<td>8H</td>
<td>immh = 001x, Q = 1</td>
</tr>
<tr>
<td>2S</td>
<td>immh = 01xx, Q = 0</td>
</tr>
<tr>
<td>4S</td>
<td>immh = 01xx, Q = 1</td>
</tr>
</tbody>
</table>

See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8H</td>
<td>immh = 0001</td>
</tr>
<tr>
<td>4S</td>
<td>immh = 001x</td>
</tr>
<tr>
<td>2D</td>
<td>immh = 01xx</td>
</tr>
</tbody>
</table>

See Advanced SIMD modified immediate on page C4-632 when immh = 0000.

The encoding immh = 1xxx is reserved.

<Vb> Is the destination width specifier, encoded in the "immh" field. It can have the following values:

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>immh = 0001</td>
</tr>
<tr>
<td>H</td>
<td>immh = 001x</td>
</tr>
<tr>
<td>S</td>
<td>immh = 01xx</td>
</tr>
</tbody>
</table>

The following encodings are reserved:

- immh = 0000.
- immh = 1xxx.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in the "immh" field. It can have the following values:

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>immh = 0001</td>
</tr>
<tr>
<td>S</td>
<td>immh = 001x</td>
</tr>
<tr>
<td>D</td>
<td>immh = 01xx</td>
</tr>
</tbody>
</table>
The following encodings are reserved:

- \( \text{immh} = 0000 \).
- \( \text{immh} = 1xxx \).

\(<n>\)  

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<\text{shift}>\)  

For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in the "immh:immb" field. It can have the following values:

\( (16-\text{UInt}(\text{immh}:\text{immb})) \) when \( \text{immh} = 0001 \)
\( (32-\text{UInt}(\text{immh}:\text{immb})) \) when \( \text{immh} = 001x \)
\( (64-\text{UInt}(\text{immh}:\text{immb})) \) when \( \text{immh} = 01xx \)

The following encodings are reserved:

- \( \text{immh} = 0000 \).
- \( \text{immh} = 1xxx \).

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in the "immh:immb" field. It can have the following values:

\( (16-\text{UInt}(\text{immh}:\text{immb})) \) when \( \text{immh} = 0001 \)
\( (32-\text{UInt}(\text{immh}:\text{immb})) \) when \( \text{immh} = 001x \)
\( (64-\text{UInt}(\text{immh}:\text{immb})) \) when \( \text{immh} = 01xx \)

See Advanced SIMD modified immediate on page C4-632 when \( \text{immh} = 0000 \).

The encoding \( \text{immh} = 1xxx \) is reserved.

Operation for all encodings

```c
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bias(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
  element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
  (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
  if sat then FPSR.QC = '1';
Vpart[d, part, datasize] = result;
```
C7.2.380 UQSUB

Unsigned saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 0</td>
<td><strong>size</strong></td>
<td>1</td>
<td>Rm</td>
</tr>
</tbody>
</table>

#### Encoding

UQSUB <V><d>, <V><n>, <V><m>

#### Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

### Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1 1 1 0</td>
</tr>
</tbody>
</table>

#### Encoding

UQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

#### Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

### Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- **8** when size = 00
H when size = 01
S when size = 10
D when size = 11

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
C7.2.381  UQXTN, UQXTN2

Unsigned saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates each value to half the original width, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The UQXTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQXTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**Encoding**

UQXTN <Vb><d>, <Va><n>

**Decode for this encoding**

```plaintext```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer databit = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');
```

**Vector**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**Encoding**

UQXTN{2} <Vd>.< Tb>, <Vn>.<Ta>

**Decode for this encoding**

```plaintext```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer databit = esize;
integer part = UInt(Q);
integer elements = data_size DIV esize;
```
boolean unsigned = (U == '1');

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:
   [absent] when Q = 0
   [present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
   8B when size = 00, Q = 0
   16B when size = 00, Q = 1
   4H when size = 01, Q = 0
   8H when size = 01, Q = 1
   2S when size = 10, Q = 0
   4S when size = 10, Q = 1
   The encoding size = 11, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in the "size" field. It can have the following values:
   8H when size = 00
   4S when size = 01
   2D when size = 10
   The encoding size = 11 is reserved.

<Vb> Is the destination width specifier, encoded in the "size" field. It can have the following values:
   B when size = 00
   H when size = 01
   S when size = 10
   The encoding size = 11 is reserved.

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in the "size" field. It can have the following values:
   H when size = 00
   S when size = 01
   D when size = 10
   The encoding size = 11 is reserved.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;
bits(2*esize) element;
boolean sat;
for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';

    Vpart[d, part, datasize] = result;
C7.2.382 URECPE

Unsigned Reciprocal Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse for the unsigned integer value, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

URECPE <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(32) element;
for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRecipEstimate(element);
V[d, datasize] = result;
C7.2.383 URHADD

Unsigned Rounding Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see UHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

URHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  8B when size = 00, Q = 0
  16B when size = 00, Q = 1
  4H when size = 01, Q = 0
  8H when size = 01, Q = 1
  2S when size = 10, Q = 0
  4S when size = 10, Q = 1
The encoding size = 11, Q = x is reserved.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  sum = (element1 + element2 + 1) >> 1;
  Elem[result, e, esize] = sum<esize-1:0>;

V[d, datasize] = result;
C7.2.384 URSHL

Unsigned Rounding Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

```plaintext
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Encoding

URSHL <V><d>, <V><n>, <V><m>

Decode for this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector

```plaintext
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Encoding

URSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```
Assembler symbols

\langle V \rangle  
Is a width specifier, encoded in the "size" field. It can have the following values:

- 0  when size = 11

The following encodings are reserved:

- size = 0x.
- size = 10.

\langle d \rangle  
Is the number of the SIMD&FP destination register, in the "Rd" field.

\langle n \rangle  
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\langle m \rangle  
Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

\langle Vd \rangle  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\langle T \rangle  
Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B  when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H  when size = 01, Q = 0
- 8H  when size = 01, Q = 1
- 2S  when size = 10, Q = 0
- 4S  when size = 10, Q = 1
- 2D  when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

\langle Vn \rangle  
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\langle Vm \rangle  
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = \langle V \rangle[n, datasize];
bits(datasize) operand2 = \langle V \rangle[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(\langle V \rangle[operand1, e, esize], unsigned);
    integer shift = SInt(\langle V \rangle[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        if rounding then
            element = (element + (1 << (shift - 1))) >> shift;
        else
            element = element >> shift;
    int saturating then
        (\langle V \rangle[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        \langle V \rangle[result, e, esize] = element<esize-1:0>;

\langle V \rangle[d, datasize] = result;
C7.2.385  URSHR

Unsigned Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see USHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

```
<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[19 18 16]</th>
<th>[15 14 13 12]</th>
<th>[11 10 9]</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>immh</td>
<td>o1 o0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

URSHR <V><d>, <V><n>, #<shift>

**Decode for this encoding**

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

**Vector**

```
<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[19 18 16]</th>
<th>[15 14 13 12]</th>
<th>[11 10 9]</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>immh</td>
<td>o1 o0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

URSHR <Vd>,<T>, <Vn>,<T>, #<shift>

**Decode for this encoding**

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
```
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

### Assembler symbols

<\v> Is a width specifier, encoded in the "immh" field. It can have the following values:

- D when immh = 1xxx
- The encoding immh = 0xxx is reserved.

<\d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<\n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<\V> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

- 8B when immh = 0001, Q = 0
- 16B when immh = 0001, Q = 1
- 4H when immh = 001x, Q = 0
- 8H when immh = 001x, Q = 1
- 2S when immh = 01xx, Q = 0
- 4S when immh = 01xx, Q = 1
- 2D when immh = 1xxx, Q = 1

See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.
The encoding immh = 1xxx, Q = 0 is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

- \(128-\text{UInt}(\text{immh:immb})\) when immh = 1xxx
- The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

- \(16-\text{UInt}(\text{immh:immb})\) when immh = 0001
- \(32-\text{UInt}(\text{immh:immb})\) when immh = 001x
- \(64-\text{UInt}(\text{immh:immb})\) when immh = 01xx
- \(128-\text{UInt}(\text{immh:immb})\) when immh = 1xxx

See Advanced SIMD modified immediate on page C4-632 when immh = 0000.

### Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements - 1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;}
V[d, datasize] = result;
C7.2.386   URSQRTE

Unsigned Reciprocal Square Root Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse square root for each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

URSQRTE <Vd>.<T>, <Vn>.<T>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "sz:Q" field. It can have the following values:

2S when sz = 0, Q = 0
4S when sz = 0, Q = 1

The encoding sz = 1, Q = x is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(32) element;
for e = 0 to elements-1
  element = Elem[operand, e, 32];
  Elem[result, e, 32] = UnsignedRSqrtEstimate(element);
V[d, datasize] = result;
C7.2.387 URSRA

Ununsigned Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see USRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Scalar

```
| 31 30 29 28|27 26 25 24|23 22 |
| 0 1 1 1 1 1 1 1 0 | !=0000 | immh | 0 0 1 1 0 1 | Rn | Rd |
```

**Encoding**

URSRA <V><d>, <V><n>, #<shift>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector

```
| 31 30 29 28|27 26 25 24|23 22 |
| 0 Q 1 0 1 1 1 1 1 0 | !=0000 | immh | 0 0 1 1 0 1 | Rn | Rd |
```

**Encoding**

URSRA <Vd>,<T>, <Vn><T>, #<shift>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
```
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler symbols

\(<V>\) Is a width specifier, encoded in the "immh" field. It can have the following values:

\(D\) when \(\text{immh} = 1xxx\)
The encoding \(\text{immh} = 0xxx\) is reserved.

\(<d>\) Is the number of the SIMD&FP destination register, in the "Rd" field.

\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<V_d>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

8B when \(\text{immh} = 0001, Q = 0\)
16B when \(\text{immh} = 0001, Q = 1\)
4H when \(\text{immh} = 001x, Q = 0\)
8H when \(\text{immh} = 001x, Q = 1\)
2S when \(\text{immh} = 01xx, Q = 0\)
4S when \(\text{immh} = 01xx, Q = 1\)
2D when \(\text{immh} = 1xxx, Q = 1\)

See Advanced SIMD modified immediate on page C4-632 when \(\text{immh} = 0000, Q = x\).
The encoding \(\text{immh} = 1xxx\), \(Q = 0\) is reserved.

\(<V_n>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<\text{shift}>\) For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

\((128-\text{UInt}(\text{immb}))\) when \(\text{immh} = 1xxx\)
The encoding \(\text{immh} = 0xxx\) is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

\((16-\text{UInt}(\text{immb}))\) when \(\text{immh} = 0001\)
\((32-\text{UInt}(\text{immb}))\) when \(\text{immh} = 001x\)
\((64-\text{UInt}(\text{immb}))\) when \(\text{immh} = 01xx\)
\((128-\text{UInt}(\text{immb}))\) when \(\text{immh} = 1xxx\)

See Advanced SIMD modified immediate on page C4-632 when \(\text{immh} = 0000\).

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then \((1 << (\text{shift} - 1))\) else 0;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for \(e = 0\) to \(\text{elements-1}\)
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> \text{shift};
Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;'
V[d, datasize] = result;
USDOT (vector)

Dot Product vector form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in the corresponding 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

Vector

( feat_i8mm )

Encoding

USDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

Decode for this encoding

if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;

Assembler symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
    2S when Q = 0
    4S when Q = 1
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
    8B when Q = 0
    16B when Q = 1
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
for e = 0 to elements-1
    bits(32) res = Elem[operand3, e, 32];
    for b = 0 to 3
        integer element1 = UInt(Elem[operand1, 4+e+b, 8]);
        integer element2 = SInt(Elem[operand2, 4+e+b, 8]);
res = res + element1 * element2;
Elem[result, e, 32] = res;

V[d, datasize] = result;
C7.2.389 USDOT (by element)

Dot Product index form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

Vector

(_FEAT_I8MM)

<table>
<thead>
<tr>
<th></th>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>US</td>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Encoding

USDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[index]

Decode for this encoding

if !HaveInt8MatMulExt() then UNDEFINED;
boolean op1_unsigned = (US == '1');
boolean op2_unsigned = (US == '0');
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;

Assembler symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
2S when Q = 0
4S when Q = 1
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
8B when Q = 0
16B when Q = 1
<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
<index> Is the immediate index of a 32-bit group of four 8-bit values in the range 0 to 3, encoded in the "H:L" fields.

Operation

CheckFPAdvSIMDEnabled64();
broadcast operand1 = V[n, datasize];
broadcast operand2 = V[m, 128];
broadcast operand3 = V[d, datasize];
broadcast result;

Assembly

USDOT Vd, Ta, Vn, Tb, Vm, 4B[i]

Decode for this encoding

if !HaveInt8MatMulExt() then UNDEFINED;
boolean op1_unsigned = (US == '1');
boolean op2_unsigned = (US == '0');
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;

Assembler symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
2S when Q = 0
4S when Q = 1
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in the "Q" field. It can have the following values:
8B when Q = 0
16B when Q = 1
<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
<index> Is the immediate index of a 32-bit group of four 8-bit values in the range 0 to 3, encoded in the "H:L" fields.
for e = 0 to elements-1
    bits(32) res = Elem[operand3, e, 32];
    for b = 0 to 3
        integer element1 = Int(Elem[operand1, 4*e+b, 8], op1 unsigned);
        integer element2 = Int(Elem[operand2, 4*i+b, 8], op2 unsigned);
        res = res + element1 * element2;
    Elem[result, e, 32] = res;
    V[d, datasize] = result;
C7.2.390  USHL

Unsigned Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see URSHL.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1</td>
<td>1 1 1 0</td>
<td>1</td>
<td>Rm</td>
<td>0 1 0 0</td>
<td>0 1</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>
```

**Encoding**

USHL `<V><d>, <V><n>, <V><m>`

**Decode for this encoding**

- integer `d` = UInt(Rd);
- integer `n` = UInt(Rn);
- integer `m` = UInt(Rm);
- integer `esize` = 8 << UInt(size);
- integer `datasize` = esize;
- integer `elements` = 1;
- boolean `unsigned` = (U == '1');
- boolean `rounding` = (R == '1');
- boolean `saturating` = (S == '1');

- if `S == '0'` and `size != '11'` then UNDEFINED;

**Vector**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0 1 1 1 0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>0 1 0 0</td>
</tr>
</tbody>
</table>
```

**Encoding**

USHL `<Vd>.<T>, <Vn>.<T>, <Vm>.<T>`

**Decode for this encoding**

- integer `d` = UInt(Rd);
- integer `n` = UInt(Rn);
- integer `m` = UInt(Rm);
- if `size:Q == '110'` then UNDEFINED;
- integer `esize` = 8 << UInt(size);
- integer `datasize` = if `Q == '1'` then 128 else 64;
- integer `elements` = datasize DIV esize;
- boolean `unsigned` = (U == '1');
- boolean `rounding` = (R == '1');
- boolean `saturating` = (S == '1');
Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
  D when size = 11
The following encodings are reserved:
  * size = 0x.
  * size = 10.

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  8B when size = 00, Q = 0
  16B when size = 00, Q = 1
  4H when size = 01, Q = 0
  8H when size = 01, Q = 1
  2S when size = 10, Q = 0
  4S when size = 10, Q = 1
  2D when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
  integer element = Int(Elem[operand1, e, esize], unsigned);
  integer shift = SInt(Elem[operand2, e, esize]<7:0>);
  if shift >= 0 then // left shift
    element = element << shift;
  else // right shift
    shift = -shift;
    if rounding then
      element = (element + (1 << (shift - 1))) >> shift;
    else
      element = element >> shift;
  if saturating then
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';
  else
    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.391 USHLL, USHLL2

Unsigned Shift Left Long (immediate). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, shifts the unsigned integer value left by the specified number of bits, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The USHLL instruction extracts vector elements from the lower half of the source register. The USHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias UXTL, UXTL2. See Alias conditions for details of when each alias is preferred.

| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | !=0000 | immh | 1 | 0 | 1 | 0 | 1 | Rn | Rd |

Encoding

USHLL[2] <Vd>.<Ta>, <Vn>.<Tb>, #<shift>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if imm == '0000' then SEE "Advanced SIMD modified immediate";
if imm<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasure = 64;
integer part = UInt(Q);
integer elements = datasure DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>UXTL, UXTL2</td>
<td>immh == '000' &amp;&amp; BitCount(immh) == 1</td>
</tr>
</tbody>
</table>

Assembler symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent] when Q = 0
[present] when Q = 1

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in the "immh" field. It can have the following values:

8H when immh = 0001
4S when immh = 001x
2D when immh = 01xx

See Advanced SIMD modified immediate on page C4-632 when immh = 0000.
The encoding immh = 1xxx is reserved.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:
8B when immh = 0001, Q = 0
16B when immh = 0001, Q = 1
4H when immh = 001x, Q = 0
8H when immh = 001x, Q = 1
2S when immh = 01xx, Q = 0
4S when immh = 01xx, Q = 1

See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.
The encoding immh = 1xxx, Q = x is reserved.

<shift> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in the
"immh:immb" field. It can have the following values:
(UInt(immh:immb)-8) when immh = 0001
(UInt(immh:immb)-16) when immh = 001x
(UInt(immh:immb)-32) when immh = 01xx

See Advanced SIMD modified immediate on page C4-632 when immh = 0000.
The encoding immh = 1xxx is reserved.

Operation
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part, datasize];
bits(datasize*2) result;
integer element;
for e = 0 to elements-1
  element = Int(Elem[operand, e, esize], unsigned) << shift;
  Elem[result, e, 2*esize] = element<[2*esize-1:0]>
V[d, datasize*2] = result;

Operational information
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.392 USHR

Unsigned Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see URSHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Scalar

| [31 30 29 28|27 26 25 24|23 22] | [19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 ] |
|------------------|------------------|------------------|------------------|------------------|------------------|
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | !=0000 | immh | 0 | 0 | 0 | 0 | 1 | Rn | Rd |

**Encoding**

USHR <V><d>, <V><n>, #<shift>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

### Vector

| [31 30 29 28|27 26 25 24|23 22] | [19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 ] |
|------------------|------------------|------------------|------------------|------------------|------------------|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | !=0000 | immh | 0 | 0 | 0 | 0 | 1 | Rn | Rd |

**Encoding**

USHR <Vd>.<T>, <Vn>.<T>, #<shift>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
```
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler symbols

\(<V>\) Is a width specifier, encoded in the "immh" field. It can have the following values:

- D when immh = 1xxx
  The encoding immh = 0xxx is reserved.

\(<d>\) Is the number of the SIMD&FP destination register, in the "Rd" field.

\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:

- 8B when immh = 0001, Q = 0
- 16B when immh = 0001, Q = 1
- 4H when immh = 001x, Q = 0
- 8H when immh = 001x, Q = 1
- 2S when immh = 01xx, Q = 0
- 4S when immh = 01xx, Q = 1
- 2D when immh = 1xxx, Q = 1

See Advanced SIMD modified immediate on page C4-632 when immh = 0000, Q = x.

The encoding immh = 1xxx, Q = 0 is reserved.

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<\text{shift}>\) For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:

- \((128-\text{UInt}(\text{immh:immb}))\) when immh = 1xxx
  The encoding immh = 0xxx is reserved.

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:

- \((16-\text{UInt}(\text{immh:immb}))\) when immh = 0001
- \((32-\text{UInt}(\text{immh:immb}))\) when immh = 001x
- \((64-\text{UInt}(\text{immh:immb}))\) when immh = 01xx
- \((128-\text{UInt}(\text{immh:immb}))\) when immh = 1xxx

See Advanced SIMD modified immediate on page C4-632 when immh = 0000.

Operation for all encodings

\(\text{CheckFPA}dv\text{SIMD}N\text{abled64}()\);
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then \((1 << (\text{shift} - 1))\) else 0;
integer element;
operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
  element = (\(\text{Int}(\text{Elem}[\text{operand}, e, \text{esize}], \text{unsigned}) + \text{round_const}\)) >> \text{shift};
  \(\text{Elem}[\text{result}, e, \text{esize}] = \text{Elem}[\text{operand2}, e, \text{esize}] + \text{element<esize-1:0>};\)
∀[d, datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.393 USMMLA (vector)

Unsigned and signed 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. ID_AA64ISAR1_EL1.I8MM indicates whether this instruction is supported.

**Vector**

(FEAT_I8MM)

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 0 0 1 1 0 1 0 0 Rm 1 0 1 0 1 1 Rn Rd
```

**Encoding**

USMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

**Decode for this encoding**

```c
if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

**Assembler symbols**

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n, 128];
bits(128) operand2 = V[m, 128];
bits(128) addend = V[d, 128];
V[d, 128] = MatMulAdd(addend, operand1, operand2, TRUE, FALSE);
```
C7.2.394   USQADD

Unsigned saturating Accumulate of Signed value. This instruction adds the signed integer values of the vector elements in the source SIMD&FP register to corresponding unsigned integer values of the vector elements in the destination SIMD&FP register, and accumulates the resulting unsigned integer values with the vector elements of the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Scalar**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0</td>
<td>1 0 0 0 0 0 0 1</td>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rd</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

USQADD <V><d>, <V><n>

**Decode for this encoding**

```c
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');
```

**Vector**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0 1 0</td>
<td>1 0 0 0 0 0 0 1</td>
<td>1 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rd</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

USQADD <Vd>.<T>, <Vn>.<T>

**Decode for this encoding**

```c
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```
Assembler symbols

<\v> Is a width specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

8B when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
2D when size = 11, Q = 1

The encoding size = 11, Q = 0 is reserved.

<\vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation for all encodings

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(datasize) operand2 = V[d, datasize];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d, datasize] = result;
C7.2.395  USRA

Unsigned Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see URSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>19 18 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0</td>
<td>!=0000</td>
<td>immh</td>
<td>0 0 0 1 0 1</td>
</tr>
</tbody>
</table>

Encoding

USRA <V><d>, <V><n>, #<shift>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>19 18 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 1 0 1 1 1 1 0</td>
<td>!=0000</td>
<td>immh</td>
<td>0 0 1 0 1</td>
</tr>
</tbody>
</table>

Encoding

USRA <Vd>.<T>, <Vn>.<T>, #<shift>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE "Advanced SIMD modified immediate";
if immh[3:Q] == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

**Assembler symbols**

- $\langle V \rangle$: Is a width specifier, encoded in the "immh" field. It can have the following values:
  - D when immh = 1xxx
  - The encoding immh = 0xxx is reserved.

- $\langle d \rangle$: Is the number of the SIMD&FP destination register, in the "Rd" field.

- $\langle n \rangle$: Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

- $\langle Vd \rangle$: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- $\langle T \rangle$: Is an arrangement specifier, encoded in the "immh:Q" field. It can have the following values:
  - 8B when immh = 0001, Q = 0
  - 16B when immh = 0001, Q = 1
  - 4H when immh = 001x, Q = 0
  - 8H when immh = 001x, Q = 1
  - 2S when immh = 01xx, Q = 0
  - 4S when immh = 01xx, Q = 1
  - 2D when immh = 1xxx, Q = 1
  - See *Advanced SIMD modified immediate* on page C4-632 when immh = 0000, Q = x.

- $\langle Vn \rangle$: Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- $\langle \text{shift} \rangle$: For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in the "immh:immb" field. It can have the following values:
  - (128-UInt(immh:immb)) when immh = 1xxx
  - The encoding immh = 0xxx is reserved.

  For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in the "immh:immb" field. It can have the following values:
  - (16-UInt(immh:immb)) when immh = 0001
  - (32-UInt(immh:immb)) when immh = 001x
  - (64-UInt(immh:immb)) when immh = 01xx
  - (128-UInt(immh:immb)) when immh = 1xxx
  - See *Advanced SIMD modified immediate* on page C4-632 when immh = 0000.

**Operation for all encodings**

```c
CheckFPAdvSIMDEnabled64();
bias(datasize) operand = V[n, datasize];
bias(datasize) operand2;
bias(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
```
V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.396  USUBL, USUBL2

Unsigned Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The destination vector elements are twice as long as the source vector elements.

The USUBL instruction extracts each source vector from the lower half of each source register. The USUBL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

USUBL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

Decode for this encoding

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

Assembler symbols

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

[absent]  when Q = 0
[present]  when Q = 1

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  Is an arrangement specifier, encoded in the "size" field. It can have the following values:

8H  when size = 00
4S  when size = 01
2D  when size = 10

The encoding size = 11 is reserved.

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>  Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

88  when size = 00, Q = 0
16B when size = 00, Q = 1
4H when size = 01, Q = 0
8H when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<\Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
   element1 = Int(Elem[operand1, e, esize], unsigned);
   element2 = Int(Elem[operand2, e, esize], unsigned);
   if sub_op then
      sum = element1 - element2;
   else
      sum = element1 + element2;
   Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d, 2*datasize] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.397  **USUBW, USUBW2**

Unsigned Subtract Wide. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element in the lower or upper half of the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are unsigned integer values.

The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register.

The **USUBW** instruction extracts vector elements from the lower half of the first source register. The **USUBW2** instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Encoding**

```
<table>
<thead>
<tr>
<th>[31 30 29 28][27 26 25 24][23 22 21 20]</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

U   o1
```

**Decode for this encoding**

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

**Assembler symbols**

**2**  
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

**<Vd>**  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<Ta>**  
Is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
- 4S when size = 01
- 2D when size = 10

The encoding size = 11 is reserved.

**<Vn>**  
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d, 2*datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.398  UXTL, UXTL2

Unsigned extend Long. This instruction copies each vector element from the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The UXTL instruction extracts vector elements from the lower half of the source register. The UXTL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is an alias of the USHLL, USHLL2 instruction. This means that:

- The encodings in this description are named to match the encodings of USHLL, USHLL2.
- The description of USHLL, USHLL2 gives the operational pseudocode for this instruction.

This instruction is an alias of the USHLL, USHLL2 instruction. This means that:
- The encodings in this description are named to match the encodings of USHLL, USHLL2.
- The description of USHLL, USHLL2 gives the operational pseudocode for this instruction.

Encoding

\[
\text{UXTL2} \quad <Vd>.<Ta>, <Vn>.<Tb>
\]

is equivalent to

\[
\text{USHLL2} \quad <Vd>.<Ta>, <Vn>.<Tb>, #0
\]

and is the preferred disassembly when \(\text{BitCount(immh)} = 1\).

Assembler symbols

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22</th>
<th>19 18</th>
<th>16 15 14 13</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[\text{U} \quad \text{immh} \quad \text{immb}\]

\[\text{2} \quad \text{Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:}
\]

| [absent] when Q = 0 | [present] when Q = 1 |

\[\text{<Vd>} \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.}
\]

\[\text{<Ta>} \quad \text{Is an arrangement specifier, encoded in the "immh" field. It can have the following values:}
\]

| 8H when \(\text{immh} = 0001\) |
| 4S when \(\text{immh} = 001x\) |
| 2D when \(\text{immh} = 01xx\) |

See \emph{Advanced SIMD modified immediate} on page C4-632 when \(\text{immh} = 0000\).

The encoding \(\text{immh} = 1xxx\) is reserved.

\[\text{<Vn>} \quad \text{Is the name of the SIMD&FP source register, encoded in the "Rn" field.}
\]

\[\text{<Tb>} \quad \text{Is an arrangement specifier, encoded in the "immm:Q" field. It can have the following values:}
\]

| 8B when \(\text{immm} = 0001, Q = 0\) |
| 16B when \(\text{immm} = 0001, Q = 1\) |
| 4H when \(\text{immm} = 001x, Q = 0\) |
| 8H when \(\text{immm} = 001x, Q = 1\) |
**Operation**

The description of **USHLL, USHLL2** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.399 UZP

Unzip vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

——— Note ————

This instruction can be used with UZP2 to de-interleave two vectors.

The following figure shows the operation of UZP1 and UZP2 with the arrangement specifier 8B.

![Operation of UZP1 and UZP2](image)

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Encoding

UZP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

### Decode for this encoding

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

### Assembler symbols

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - 8B when size = 00, Q = 0
  - 16B when size = 00, Q = 1
  - 4H when size = 01, Q = 0
  - 8H when size = 01, Q = 1
  - 2S when size = 10, Q = 0
  - 4S when size = 10, Q = 1
2D when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.

<\text{n}> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<\text{m}> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n, datasize];
bits(datasize) operandh = V[m, datasize];
bits(datasize) result;
bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];
V[d, datasize] = result;
\end{verbatim}

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.400 UZP2

Unzip vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

--- Note ---

This instruction can be used with UZP1 to de-interleave two vectors.

The following figure shows the operation of UZP1 and UZP2 with the arrangement specifier 8B.

```
Vn  A7  A6  A5  A4  A3  A2  A1  A0
Vm  B7  B6  B5  B4  B3  B2  B1  B0
```

UZP1.8, doubleword

```
Vd  B7  B6  B5  B4  A6  A5  A3  A2
```

UZP2.8, doubleword

```
Vd  B7  B6  B5  B4  A7  A6  A3  A2
```

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

--- Encoding ---

UZP2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

--- Decode for this encoding ---

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

--- Assembler symbols ---

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - `8B` when size = 00, Q = 0
  - `16B` when size = 00, Q = 1
  - `4H` when size = 01, Q = 0
  - `8H` when size = 01, Q = 1
  - `2S` when size = 10, Q = 0
  - `4S` when size = 10, Q = 1
2D when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n, datasize];
bits(datasize) operandh = V[m, datasize];
bits(datasize) result;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];

V[d, datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.401 XAR

Exclusive OR and Rotate performs a bitwise exclusive OR of the 128-bit vectors in the two source SIMD&FP registers, rotates each 64-bit element of the resulting 128-bit vector right by the value specified by a 6-bit immediate value, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SHA3 is implemented.

Advanced SIMD

(_FEAT_SHA3)

| [31 30 29 28]27 26 25 24|23 22 21 20] 16|15 | 10 9 | 5 4 | 0 |
|--------------------------|-----------------|-------|-------|-------|-------|-------|
| 1 1 0 0 1 1 1 0 1 0 0 | Rm | imm6 | Rn | Rd |

Encoding

XAR <Vd>.2D, <Vn>.2D, <Vm>.2D, #<imm6>

Decode for this encoding

if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<imm6> Is a rotation right, encoded in "imm6".

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) tmp;
tmp = Vn EOR Vm;
V[d, 128] = ROR(tmp<127:64>, UInt(imm6)):ROR(tmp<63:0>, UInt(imm6));

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.402 XTN, XTN2

Extract Narrow. This instruction reads each vector element from the source SIMD&FP register, narrows each value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

The XTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the XTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 1 1 1 0</td>
<td>size</td>
<td>1 0 0 0 0 1 0 1 0</td>
</tr>
</tbody>
</table>

**Encoding**

XTN{2} <Vd>, <Tb>, <Vn>, <Ta>

**Decode for this encoding**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

**Assembler symbols**

2 is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in the "Q" field. It can have the following values:

- [absent] when Q = 0
- [present] when Q = 1

<Vd> is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- 8B when size = 00, Q = 0
- 16B when size = 00, Q = 1
- 4H when size = 01, Q = 0
- 8H when size = 01, Q = 1
- 2S when size = 10, Q = 0
- 4S when size = 10, Q = 1

The encoding size = 11, Q = x is reserved.

<Vn> is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> is an arrangement specifier, encoded in the "size" field. It can have the following values:

- 8H when size = 00
4S when size = 01
2D when size = 10

The encoding size = 11 is reserved.

Operation

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;
bits(2*esize) element;

for e = 0 to elements-1
element = Elem[operand, e, 2*esize];
   Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part, datasize] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C7.2.403 ZIP1

Zip vectors (primary). This instruction reads adjacent vector elements from the lower half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

Note

This instruction can be used with ZIP2 to interleave two vectors.

The following figure shows the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.

![Diagram of ZIP1 and ZIP2 operation]

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Encoding

ZIP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

Decode for this encoding

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler symbols

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:
  - `8B` when size = 00, Q = 0
  - `16B` when size = 00, Q = 1
  - `4H` when size = 01, Q = 0
  - `8H` when size = 01, Q = 1
  - `2S` when size = 10, Q = 0
4S when size = 10, Q = 1
2D when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.

\(<Vn>\) is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vm>\) is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

\[
\text{CheckFPAdvSIMDEnabled64}();
\text{bits}(\text{datasize}) \text{ operand1} = \text{V}[n, \text{datasize}];
\text{bits}(\text{datasize}) \text{ operand2} = \text{V}[m, \text{datasize}];
\text{bits}(\text{datasize}) \text{ result};
\]

\[
\text{integer base} = \text{part} \times \text{pairs};
\]

\[
\text{for } p = 0 \text{ to pairs-1}
\]
\[
\text{Elem}[\text{result}, 2p+0, \text{esize}] = \text{Elem}[\text{operand1}, \text{base}+p, \text{esize}];
\text{Elem}[\text{result}, 2p+1, \text{esize}] = \text{Elem}[\text{operand2}, \text{base}+p, \text{esize}];
\]

\[
\text{V}[d, \text{datasize}] = \text{result};
\]

**Operational information**

If \text{PSTATE.DIT} is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C7.2.404 ZIP2

Zip vectors (secondary). This instruction reads adjacent vector elements from the upper half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

--- Note ---

This instruction can be used with ZIP1 to interleave two vectors.

The following figure shows the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.

```
<table>
<thead>
<tr>
<th>Vn</th>
<th>A7</th>
<th>A6</th>
<th>A5</th>
<th>A4</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vm</td>
<td>B7</td>
<td>B6</td>
<td>B5</td>
<td>B4</td>
<td>B3</td>
<td>B2</td>
<td>B1</td>
<td>B0</td>
</tr>
</tbody>
</table>
```

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Encoding**

ZIP2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

**Decode for this encoding**

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

**Assembler symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in the "size:Q" field. It can have the following values:

- **8B** when size = 00, Q = 0
- **16B** when size = 00, Q = 1
- **4H** when size = 01, Q = 0
- **8H** when size = 01, Q = 1
2S when size = 10, Q = 0
4S when size = 10, Q = 1
2D when size = 11, Q = 1
The encoding size = 11, Q = 0 is reserved.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer base = part * pairs;
for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];
V[d, datasize] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Chapter C8
SVE Instruction Descriptions

This chapter describes the SVE instructions. It contains the following sections:

• About the SVE instructions on page C8-2974.
• Alphabetical list of SVE instructions on page C8-2975.
C8.1 About the SVE instructions

Alphabetical list of SVE instructions on page C8-2975 gives full descriptions of the SVE instructions that are in the following instruction groups:

- Loads and store instructions associated with the SVE registers.
- Data processing instructions with SVE registers.

A64 instruction set encoding on page C4-388 provides an overview of the instruction encodings as part of an instruction class within a functional group.

The SVE instructions include details relating to the Scalable Matrix Extension (SME), such as Streaming SVE mode and FEAT_SME. For more information, see, Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A (ARM DDI 0616).
C8.2 Alphabetical list of SVE instructions

This section lists every section in the SVE category of the A64 instruction set. For details of the format used, see Structure of the A64 assembler language on page C1-228.
C8.2.1  ABS

Compute the absolute value of the signed integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0</td>
<td>1 0 0 1 0</td>
<td>1 0 1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

ABS <Zd>.<T>, <Pg>/M, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer element = SInt(Elem[operand, e, esize]);
    element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.2 **ADCLB**

Add the even-numbered elements of the first source vector and the 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector to the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.

**Encoding**

ADCLB `<Zda>.<T>, <Zn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVEZ() \&\& !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

**Assembler symbols**

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in the "sz" field. It can have the following values:

- S when sz = 0
- D when sz = 1

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
class integer VL = CurrentVL;
class integer pairs = VL DIV (esize \times 2);
bits(VL) operand = Z[n, VL];
bits(VL) carries = Z[m, VL];
bits(VL) result = Z[da, VL];

for \( p = 0 \) to pairs-1

bits(esize) element1 = Elem[result, 2*p + 0, esize];
bits(esize) element2 = Elem[operand, 2*p + 0, esize];
bits(esize) carry_in = Elem[carries, 2*p + 1, esize]<0>;

(res, nzcv) = AddWithCarry(element1, element2, carry_in);
carry_out = nzcv<1>;

Elem[result, 2*p + 0, esize] = res;
Elem[result, 2*p + 1, esize] = ZeroExtend(carry_out, esize);

Z[da, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.3   ADCLT

Add the odd-numbered elements of the first source vector and the 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector to the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.

Encoding

ADCLT <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<T> Is the size specifier, encoded in the "sz" field. It can have the following values:
S when sz = 0
D when sz = 1

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand = Z[n, VL];
bits(VL) carries = Z[m, VL];
bits(VL) result = Z[da, VL];
for p = 0 to pairs-1
  bits(esize) element1 = Elem[result, 2*p + 0, esize];
  bits(esize) element2 = Elem[operand, 2*p + 1, esize];
  bit carry_in = Elem[carries, 2*p + 1, esize]<0>;
  (res, nzcv) = AddWithCarry(element1, element2, carry_in);
  carry_out = nzcv<1>;
  Elem[result, 2*p + 0, esize] = res;
  Elem[result, 2*p + 1, esize] = ZeroExtend(carry_out, esize);
Z[da, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.4 ADD (immediate)

Add an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|  | 5 4 | 0 |
| 0 0 1 0 0 1 0 1 | size | 1 0 0 0 0 0 1 1 | sh | imm8 | Zdn |
```

Encoding

ADD <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;

Assembler symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:

LSL #0 when sh = 0
LSL #8 when sh = 1

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
\[ \text{Elem}[\text{result}, e, \text{esize}] = \text{element1} + \text{imm}; \]

\[ Z[dn, VL] = \text{result}; \]

**Operational information**

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.5 ADD (vectors, predicated)

Add active elements of the second source vector to corresponding elements of the first source vector and
destructively place the results in the corresponding elements of the first source vector. Inactive elements in the
destination vector register remain unmodified.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 ]
0 0 0 0 0 1 0 0 | 0 0 0 0 0 0 0 0 | Pg | Zm | Zdn
```

**Encoding**

ADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

**Assembler symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P\[g, PL\];
bits(VL) operand1 = Z\[dn, VL\];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z\[m, VL\] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem\[operand1, e, esize\];
bits(esize) element2 = Elem\[operand2, e, esize\];
  if Elem\[mask, e, esize\] == '1' then
    Elem\[result, e, esize\] = element1 + element2;
  else
    Elem\[result, e, esize\] = Elem\[operand1, e, esize\];
Z\[dn, VL\] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.6 ADD (vectors, unpredicated)

Add all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10  9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  0  0  0  1  0  0</td>
<td>0  0  0  0  0  0  0  0</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>
```

**Encoding**

ADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = element1 + element2;
Z[d, VL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.7 ADDHNB

Add each vector element of the first source vector to the corresponding vector element of the second source vector, and place the most significant half of the result in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. This instruction is unpredicated.

Encoding

ADDHNB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 01
  H when size = 10
  S when size = 11
  The encoding size = 00 is reserved.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
  H when size = 01
  S when size = 10
  D when size = 11
  The encoding size = 00 is reserved.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  integer element2 = UInt(Elem[operand2, e, esize]);
  integer res = (element1 + element2) >> halfesize;
\begin{align*}
\text{Elem}[\text{result}, 2\times e + 0, \text{halfesize}] &= \text{res}_h<\text{halfesize}-1:0>; \\
\text{Elem}[\text{result}, 2\times e + 1, \text{halfesize}] &= \text{Zeros}(\text{halfesize}); \\
\text{Z}[d, \text{VL}] &= \text{result};
\end{align*}

\textbf{Operational information}

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.8 ADDHNT

Add each vector element of the first source vector to the corresponding vector element of the second source vector, and place the most significant half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.

Encoding

ADDHNT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
integer element2 = UInt(Elem[operand2, e, esize]);
integer res = (element1 + element2) >> halfesize;
Elem[result, 2*e + 1, halfesize] = res<halfesize-1:0>;
Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.9   ADDP

Add pairs of adjacent elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

Encoding

ADDP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer element1;
integer element2;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '0' then
    Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    if IsEven(e) then
      element1 = UInt(Elem[operand1, e + 0, esize]);
      element2 = UInt(Elem[operand1, e + 1, esize]);
    else
      element1 = UInt(Elem[operand2, e - 1, esize]);
      element2 = UInt(Elem[operand2, e + 0, esize]);

integer res = element1 + element2;
Elem[result, e, esize] = res<esize-1:0>;
Z[dn, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.10  ADDPL

Add the current predicate register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer and place the result in the 64-bit destination general-purpose register or current stack pointer.

| 31 30 29 28| 27 26 25 24| 23 22 21 20 | 16|15|14|13|12|11|10 | 5 | 4 | 0 |
|------------|------------|--------------|----|----|----|----|----|----|----|----|----|
| 0 0 0 0 0 | 1 0 0 0 1 | 1 0 1 0 0 | imm6 | Rd |

**Encoding**

ADDPL <Xd|SP>, <Xn|SP>, #<imm>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);

**Assembler symbols**

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

<imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) result = operand1 + (imm * (PL DIV 8));
if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.11 ADDVL

Add the current vector register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer, and place the result in the 64-bit destination general-purpose register or current stack pointer.

Encoding

ADDVL <Xd|SP>, <Xn|SP>, #<imm>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;

integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);

Assembler symbols

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

<imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) result = operand1 + (imm * (VL DIV 8));
if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.12   ADR

Optionally sign or zero-extend the least significant 32-bits of each element from a vector of offsets or indices in the second source vector, scale each index by 2, 4 or 8, add to a vector of base addresses from the first source vector, and place the resulting addresses in the destination vector. This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

### Packed offsets

![Binary representation of packed offsets]

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 1 sz 1</td>
<td>Zm 1 0 1 0 msz</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
</tr>
</tbody>
</table>

### Encoding

ADR <Zd>.<T>, [<Zn>.<T>, <Zm>.<T>{, <mod> <amount>}]

### Decode for this encoding

if !HaveSVE() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
constant integer osize = esize;
boolean unsigned = TRUE;
integer mbytes = 1 << UInt(msz);

### Unpacked 32-bit signed offsets

![Binary representation of unpacked 32-bit signed offsets]

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 0 0 1 0 0 1</td>
<td>Zm 1 0 1 0 msz</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
</tr>
</tbody>
</table>

### Encoding

ADR <Zd>.D, [<Zn>.D, <Zm>.D, SXTW{ <amount>}]}

### Decode for this encoding

if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
constant integer osize = esize;
boolean unsigned = FALSE;
integer mbytes = 1 << UInt(msz);

### Unpacked 32-bit unsigned offsets

![Binary representation of unpacked 32-bit unsigned offsets]

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 0 1 1</td>
<td>Zm 1 0 1 0 msz</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
</tr>
</tbody>
</table>
Encoding

ADR <Zd>.D, [<Zn>.D, <Zm>.D, UXTW{ <amount>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
constant integer osize = 32;
boolean unsigned = TRUE;
integer mbytes = 1 << UInt(msz);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "sz" field. It can have the following values:
S when sz = 0
D when sz = 1

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in the "msz" field. It can have the following values:
[absent] when msz = 00
LSL when msz = x1
LSL when msz = 10

<amount> Is the index shift amount, encoded in the "msz" field. It can have the following values:
[absent] when msz = 00
#1 when msz = 01
#2 when msz = 10
#3 when msz = 11

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) base = Z[n, VL];
bits(VL) offs = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) addr = Elem[base, e, esize];
    integer offset = Int(Elem[offs, e, esize]<osize-1:0>, unsigned);
    Elem[result, e, esize] = addr + (offset * mbytes);
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.13 AESD

The AESD instruction reads a 16-byte state array from each 128-bit segment of the first source vector, together with a round key from the corresponding 128-bit segment of the second source vector. Each state array undergoes a single round of the ADDROUNDKEY(), INVSUBBYTES() and INVSHIFTROWS() transformations in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2

(FEAT_SVE_AES)

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 |
| 0 1 0 0 0 1 0 1 | 0 0 1 | 1 | 1 | 0 0 0 1 | 0 1 1 1 0 0 1 |

Encoding


Decode for this encoding

if !HaveSVE() || !HaveSVE2AES() then UNDEFINED;
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
result = operand1 EOR operand2;
for s = 0 to segments-1
    Elem[result, s, 128] = AESInvSubBytes(AESInvShiftRows(Elem[result, s, 128]));
Z[dn, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.14 AESE

The AESE instruction reads a 16-byte state array from each 128-bit segment of the first source vector together with a round key from the corresponding 128-bit segment of the second source vector. Each state array undergoes a single round of the ADDROUNDKEY(), SUBBYTES() and SHIFTROWS() transformations in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2

(FEAT_SVE_AES)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1 0 1 0 0 0 1 0 1 1 1 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size<1> size<0>
```

Encoding


Decode for this encoding

```
if !HaveSVE() || !HaveSVE2AES() then UNDEFINED;
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
result = operand1 EOR operand2;
for s = 0 to segments-1
    Elem[result, s, 128] = AESSubBytes(AESShiftRows(Elem[result, s, 128]));
Z[dn, VL] = result;
```
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.15   AESIMC

The AESIMC instruction reads a 16-byte state array from each 128-bit segment of the source register, and performs a single round of the INV MIXCOLUMNS() transformation on each state array in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2 (FEAT_SVE_AES)

Encoding


Decode for this encoding

if !HaveSVE() || !HaveSVE2AES() then UNDEFINED;
integer dn = UInt(Zdn);

Assembler symbols

<size<1> size<0>

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand = Z[dn, VL];
bits(VL) result;
for s = 0 to segments-1
    Elem[result, s, 128] = AESIMIXColumns(Elem[operand, s, 128]);
Z[dn, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
— The values of the NZCV flags.
C8.2.16 AESMC

The AESMC instruction reads a 16-byte state array from each 128-bit segment of the source register, and performs a single round of the MIXCOLUMNS() transformation on each state array in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2

(Feat_SVE_AES)

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8 | 7 6 5 4 | 0 |
| 0 1 0 0 0 1 0 1 0 0 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 | Zdn |

Encoding

AESMC <Zdn>.B, <Zdn>.B

Decode for this encoding

if !HaveSVE() || !HaveSVE2AES() then UNDEFINED;
i integer dn = UInt(Zdn);

Assembler symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand = Z[dn, VL];
bits(VL) result;
for s = 0 to segments-1
    Elem[result, s, 128] = AESMixColumns(Elem[operand, s, 128]);
Z[dn, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
— The values of the NZCV flags.
C8.2.17   AND (immediate)

Bitwise AND an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction BIC (immediate). The pseudo-instruction is never the preferred disassembly.

Encoding

\[
\begin{array}{c|c|c|c|c}
31 & 30 & 29 & 28 & 27 \ 26 \ 25 \ 24 \ 23 \ 22 \ 21 \ 20 \ 19 \ 18 \ 17 \ 5 \ 4 \ 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & \text{imm13} & \text{Zdn}
\end{array}
\]

Decoding for this encoding

\[
\text{if} \ !\text{HaveSVE()} \ \&\& \ !\text{HaveSME()} \ \text{then UNDEFINED};
\]
\[
\text{integer dn} = \text{UInt}(\text{Zdn});
\]
\[
\text{bits(64) imm} = \text{DecodeBitMasks(imm13<12>, imm13<5:0>, TRUE, 64});
\]

Assembler symbols

\(<\text{Zdn}>\) Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

\(<\text{T}>\) Is the size specifier, encoded in the "imm13<12>:imm13<5:0>" field. It can have the following values:

- \(S\) when \(\text{imm13}<12> = 0, \text{imm13}<5:0> = 0xxxxx\)
- \(H\) when \(\text{imm13}<12> = 0, \text{imm13}<5:0> = 10xxxx\)
- \(B\) when \(\text{imm13}<12> = 0, \text{imm13}<5:0> = 110xxx\)
- \(B\) when \(\text{imm13}<12> = 0, \text{imm13}<5:0> = 1110xx\)
- \(B\) when \(\text{imm13}<12> = 0, \text{imm13}<5:0> = 11110x\)
- \(D\) when \(\text{imm13}<12> = 1, \text{imm13}<5:0> = xxxxxx\)

The following encodings are reserved:

- \(\text{imm13}<12> = 0, \text{imm13}<5:0> = 111110\).
- \(\text{imm13}<12> = 0, \text{imm13}<5:0> = 111111\).

<\text{const}> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Operation

\[
\text{CheckSVEEnabled();}
\]
\[
\text{constant integer VL = CurrentVL;}
\]
\[
\text{constant integer PL = VL DIV 8;}
\]
\[
\text{constant integer elements = VL DIV 64;}
\]
\[
\text{bits(VL) operand} = Z[dn, VL];
\]
\[
\text{bits(VL) result;}
\]
\[
\text{for e = 0 to elements-1}
\]
\[
\text{bits(64) element1 = Elem[operand, e, 64];}
\]
\[
\text{Elem[result, e, 64] = element1 AND imm;}
\]
Z[dn, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.18   AND (predicates)

Bitwise AND active elements of the second source predicate with corresponding elements of the first source
predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the
destination predicate register are set to zero. Does not set the condition flags.

This instruction is used by the alias MOV (predicate, predicated, zeroing). See Alias conditions for details of when
each alias is preferred.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 10 9 8 5 4 3 0 |
|---------------------------|------------------|------------------|------------------|------------------|
| 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| Pm | 0 1 | Pg | 0 | Pn | 0 | Pd | 0 |
```

**Encoding**


**Decode for this encoding**

```plaintext
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

**Alias conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (predicate, predicated, zeroing)</td>
<td>S == '0' &amp;&amp; Pn == Pm</td>
</tr>
</tbody>
</table>

**Assembler symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` Is the name of the second source scalable predicate register, encoded in the "Pm" field.

**Operation**

```plaintext
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
```
if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = element1 AND element2;
else
    Elem[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.19 AND (vectors, predicated)

Bitwise AND active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>size 0 1 1 0 1 0 0 0</td>
<td>Pg</td>
<td>Zm</td>
<td>Zdn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

AND <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

**Assembler symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = element1 AND element2;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.20 AND (vectors, unpredicated)

Bitwise AND all elements of the second source vector with corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 0 0 0 1 0 0 0 0 1 | Zm 0 0 1 1 0 0 | Zn | Zd
```

**Encoding**

AND <Zd>.D, <Zn>.D, <Zm>.D

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

**Assembler symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
Z[d, VL] = operand1 AND operand2;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**C8.2.21 ANDS**

Bitwise AND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the alias MOVS (predicated). See *Alias conditions* for details of when each alias is preferred.

**Encoding**

\[
\begin{array}{cccccc}
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 14 | 13 & 10 | 9 | 8 | 5 | 4 | 3 | 0 \\
\hline
\begin{array}{c}
0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \\
\end{array}
\begin{array}{c}
Pm \\
Pg \\
Pn \\
Pd
\end{array}
\end{array}
\]

**Decode for this encoding**

- if !HaveSVE() \&\& !HaveSME() then UNDEFINED;
- constant integer esize = 8;
- integer g = UInt(Pg);
- integer n = UInt(Pn);
- integer m = UInt(Pm);
- integer d = UInt(Pd);
- boolean setflags = TRUE;

**Alias conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVS (predicated)</td>
<td>S == '1' &amp;&amp; Pn == Pm</td>
</tr>
</tbody>
</table>

**Assembler symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` Is the name of the second source scalable predicate register, encoded in the "Pm" field.

**Operation**

- `CheckSVEEnabled();`
- constant integer VL = CurrentVL;
- constant integer PL = VL DIV 8;
- constant integer elements = VL DIV esize;
- bits(PL) mask = P[g, PL];
- bits(PL) operand1 = P[n, PL];
- bits(PL) operand2 = P[m, PL];
- bits(PL) result;
- for e = 0 to elements-1
  - bit element1 = ElemP[operand1, e, esize];
bit element2 = ElemP[operand2, e, esize];
if ElemP[mask, e, esize] == '1' then
    ElemP[result, e, esize] = element1 AND element2;
else
    ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.22 ANDV

Bitwise AND horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as all ones.

Encoding

ANDV <V><d>, <Pg>, <Zn>.<T>

Decode for this encoding

if !HaveSVE() & & !HaveSME() then UNDEFINED;
constant integer esize = 8 $<<$ UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
   B when size = 00
   H when size = 01
   S when size = 10
   D when size = 11

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
   B when size = 00
   H when size = 01
   S when size = 10
   D when size = 11

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) result = Ones(esize);
for e = 0 to elements-1
   if Elem[mask, e, esize] == '1' then
      result = result AND Elem[operand, e, esize];
\[ V[d, esize] = \text{result}; \]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.23 ASR (immediate, predicated)

Shift right by immediate, preserving the sign bit, each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

Encoding

```
<Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
```

```
bits(4) tsize = tszh:tszl;
integer esize;
```

```
case tsize of
when '0000' UNDEFINED;
when '0001' esize = 8;
when '001x' esize = 16;
when '01xx' esize = 32;
when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler symbols

- `<Zdn>` is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  - B when tszh = 00, tszl = 01
  - H when tszh = 00, tszl = 1x
  - S when tszh = 01, tszl = xx
  - D when tszh = 1x, tszl = xx
- The encoding tszh = 00, tszl = 00 is reserved.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<const>` is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer PL = VL DIV 8;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;
```

```
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if Elem[mask, e, esize] == '1' then
```


Elem[result, e, esize] = ASR(element1, shift);
else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.24 ASR (immediate, unpredicated)

Shift right by immediate, preserving the sign bit, each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Encoding

ASR <Zd>.<T>, <Zn>.<T>, #<const>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  B when tszh = 00, tszl = 01
  H when tszh = 00, tszl = 1x
  S when tszh = 01, tszl = xx
  D when tszh = 1x, tszl = xx
The encoding tszh = 00, tszl = 00 is reserved.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bites(VL) operand1 = Z[n, VL];
bites(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  Elem[result, e, esize] = ASR(element1, shift);
$Z[d, VL] = \text{result};$

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.25   ASR (vectors)

Shift right, preserving the sign bit, active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

Encoding

ASR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B  when size = 00
H  when size = 01
S  when size = 10
D  when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    integer shift = Min(UInt(element2), esize);
    Elem[result, e, esize] = ASR(element1, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

\[ Z[\text{dn, VL}] = \text{result}; \]

**Operational information**

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.26   ASR (wide elements, predicated)

Shift right, preserving the sign bit, active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 0</td>
<td>size 0 1 1 0 0 1 0 0</td>
<td>Pg</td>
<td>Zm</td>
<td>Zdn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

ASR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.D

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

**Assembler symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - The encoding size = 11 is reserved.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```c
CheckSVEConfiguration();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = [Pg, PL];
bits(VL) operand1 = [dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then [m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
        integer shift = Min(UInt(element2), esize);
        Elem[result, e, esize] = ASR(element1, shift);
```
\( \text{Elem}\{\text{result}, \text{e}, \text{esize}\} = \text{Elem}\{\text{operand1}, \text{e}, \text{esize}\}; \)

\( Z[\text{dn}, \text{VL}] = \text{result}; \)

### Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.27 ASR (wide elements, unpredicated)

Shift right, preserving the sign bit, all elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and place the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. This instruction is unpredicated.

\[
\begin{array}{cccccccccccc}
|31|30|29|28|27|26|25|24|23|22|21|20| 16|15|14|13|12|11|10| 9 | 5 | 4 |
\hline
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & | & Zn & 1 & 0 & 0 & 0 & 0 & 0 & | & Zn & Zd & 0
\end{array}
\]

Encoding

ASR <Zd>.<T>, <Zn>.<T>, <Zm>.D

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
   B when size = 00
   H when size = 01
   S when size = 10
   The encoding size = 11 is reserved.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
   bits(esize) element1 = Elem[operand1, e, esize];
bits(64) element2 = Elem[operand2, (e + esize) DIV 64, 64];
   integer shift = Min(UInt(element2), esize);
   Elem[result, e, esize] = ASR(element1, shift);
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.28  ASRD

Shift right by immediate, preserving the sign bit, each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The result rounds toward zero as in a signed division. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

Encoding

ASRD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler symbols

<Zdn>  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T>    Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  B when tszh = 00, tszl = 01
  H when tszh = 00, tszl = 1x
  S when tszh = 01, tszl = xx
  D when tszh = 1x, tszl = xx

The encoding tszh = 00, tszl = 00 is reserved.

<Pg>   Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
integer element1 = SInt(Elem[operand1, e, esize]);
if element1 < 0 then
    element1 = element1 + ((1 << shift) - 1);
    Elem[result, e, esize] = (element1 >> shift)<esize-1:0>;
else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;

Operational information
If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.29   ASRR

Reversed shift right, preserving the sign bit, active elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

Encoding

ASRR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
if Elem[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    integer shift = Min(UInt(element1), esize);
    Elem[result, e, esize] = ASR(element2, shift);
else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.30 BCAX

Bitwise AND elements of the second source vector with the corresponding inverted elements of the third source vector, then exclusive OR the results with corresponding elements of the first source vector. The final results are destructively placed in the corresponding elements of the destination and first source vector. This instruction is unpredicated.

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zk> Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];
Z[dn, VL] = operand1 EOR (operand2 AND NOT(operand3));

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.31 BDEP

This instruction scatters the lowest-numbered contiguous bits within each element of the first source vector to the bit positions indicated by non-zero bits in the corresponding mask element of the second source vector, preserving their order, and set the bits corresponding to a zero mask bit to zero. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BitPerm indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2

(FEAT_SVE_BitPerm)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9 5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1 1</td>
<td>Zm</td>
</tr>
</tbody>
</table>

Encoding

BDEP <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() || !HaveSVE2BitPerm() then UNDEFINED;
constant integer esize = 8 << UIInt(size);
integer n = UIInt(Zn);
integer m = UIInt(Zm);
integer d = UIInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
    B when size = 00
    H when size = 01
    S when size = 10
    D when size = 11
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) data = Z[n, VL];
bits(VL) mask = Z[m, VL];
bits(VL) result;

for e = 0 to elements - 1
    Elem[result, e, esize] = BitDeposit(Elem[data, e, esize], Elem[mask, e, esize]);
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.32   BEXT

This instruction gathers bits in each element of the first source vector from the bit positions indicated by non-zero bits in the corresponding mask element of the second source vector to the lowest-numbered contiguous bits of the corresponding destination element, preserving their order, and sets the remaining higher-numbered bits to zero. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BitPerm indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2

(FEAT_SVE_BitPerm)

Encoding

BEXT <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() || !HaveSVE2BitPerm() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd>         Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>          Is the size specifier, encoded in the "size" field. It can have the following values:
             B when size = 00
             H when size = 01
             S when size = 10
             D when size = 11
<Zn>         Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>         Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) data = Z[n, VL];
bits(VL) mask = Z[m, VL];
bits(VL) result;

for e = 0 to elements - 1
    Elem[result, e, esize] = BitExtract(Elem[data, e, esize], Elem[mask, e, esize]);
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.33 BFCVT

Convert to BFloat16 from single-precision in each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Since the result type is smaller than the input type, the results are zero-extended to fill each destination element. ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

**SVE**

(FEATURE_BF16)

Encoding

BFCVT <Zd>.H, <Pg>/M, <Zn>.S

Decoding for this encoding

if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, 32) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if Elem[mask, e, 32] == '1' then
    bits(32) element = Elem[operand, e, 32];
    Elem[result, 2*e, 16] = FPConvertBF(element, FPCR[]);
    Elem[result, 2*e+1, 16] = Zeros(16);
  Z[d, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.34 BFCVTNT

Convert to BFloat16 from single-precision in each active floating-point element of the source vector, and place the results in the odd-numbered 16-bit elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

(FEAT_BF16)

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 | 10 9 | 5 4 | 0 |
0 1 1 0 0 1 0 0 1 0 0 1 0 1 0 1 0 1 | Pg | Zn | Zd]
```

Encoding

BFCVTNT <Zd>.H, <Pg>/M, <Zn>.S

Decode for this encoding

```
if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler symbols

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>`: Is the name of the first source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, 32) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
  if ElemP[mask, e, 32] == '1' then
    bits(32) element = Elem[operand, e, 32];
    Elem[result, 2*e+1, 16] = FPConvertBF(element, FPCR[]);
Z[d, VL] = result;
```
C8.2.35 **BFDOT (indexed)**

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, irrespective of the other control bits in the FPCR, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second vector. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Generates only the Default NaN, as if FPCR.DN is 1.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Generates only the default NaN, as if FPCR.DN is 1.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as controlled by the effective value of the FPCR in the current execution mode, and captured in the FPSR.

The BFloat16 pairs within the second source vector are specified using an immediate index which selects the same BFloat16 pair position within each 128-bit vector segment. The index range is from 0 to 3.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

**SVE**

(FEAT_BF16)

![SVE Instruction Encoding](image)

**Encoding**


**Decode for this encoding**

if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i2);
Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

<imm> Is the immediate index, in the range 0 to 3, encoded in the "i2" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
constant integer eltspersegment = 128 DIV 32;
binary(VL) operand1 = Z[n, VL];
binary(VL) operand2 = Z[m, VL];
binary(VL) operand3 = Z[da, VL];
binary(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bytes(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bytes(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bytes(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
    bytes(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
    bytes(32) sum = Elem[operand3, e, 32];
    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

Z[da, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.36  BFDOT (vectors)

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, irrespective of the other control bits in the FPCR, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Generates only the Default NaN, as if FPCR.DN is 1.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Generates only the default NaN, as if FPCR.DN is 1.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as controlled by the effective value of the FPCR in the current execution mode, and captured in the FPSR.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

---

SVE

(FEAT_BF16)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 0 1 0 0 0 0 1 1</td>
<td>Zm</td>
<td>1 0 0 0 0 0</td>
<td>Zn</td>
<td>Zda</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding


Decode for this encoding

if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler symbols

<Zda>  Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
  bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
  bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
  bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
  bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
  bits(32) sum = Elem[operand3, e, 32];
  sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
  Elem[result, e, 32] = sum;

Z[da, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.37 BFMLALB (indexed)

This BFloat16 floating-point multiply-add long instruction widens the even-numbered BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

(FEAT_BF16)

Encoding


Decode for this encoding

if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
<imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
constant integer elspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
integer segmentbase = e - (e MOD elspersegment);
integer s = 2 * segmentbase + index;
bits(16) element1 = Elem[operand1, 2 + e + 0, 16];
bits(16) element2 = Elem[operand2, s, 16];
bits(32) element3 = Elem[operand3, e, 32];
Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);
Z[da, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.38  BFMLALB (vectors)

This BFloat16 floating-point multiply-add long instruction widens the even-numbered BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

(FEAT_BF16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10  9</th>
<th>5  4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 0 1 1 1</td>
<td>Zm</td>
<td>1 0 0 0 0 0</td>
<td>Zn</td>
</tr>
</tbody>
</table>

Encoding


Decode for this encoding

if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler symbols

<Zda>  Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn>   Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>   Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
  bits(16) element1 = Elem[operand1, 2 * e + 0, 16];
  bits(16) element2 = Elem[operand2, 2 * e + 0, 16];
  bits(32) element3 = Elem[operand3, e, 32];
  Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);
Z[da, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.39  BFMLALT (indexed)

This BFloat16 floating-point multiply-add long instruction widens the odd-numbered BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

(FEAT_BF16)

Encoding


Decode for this encoding

if (!(HaveSVE() && !HaveSME()) || !HaveBF16Ext()) then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);

Assembler symbols

<Zda>  Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn>   Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm>   Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
<imm>  Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
constant integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(16) element1 = Elem[operand1, 2 * e + 1, 16];
    bits(16) element2 = Elem[operand2, s, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[1]);
Z[da, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.40 BFMLALT (vectors)

This BFloat16 floating-point multiply-add long instruction widens the odd-numbered BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

(FEAT_BF16)

```
|31 30 29 28|27 26 25 24|23 22 21 20|16|15|14|13|12|11|10|9 | 5 | 4 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | Zn | 1 | 0 | 0 | 0 | 1 | Zm | Zda |
```

**Encoding**


**Decode for this encoding**

```c
if (!HaveSVE() & !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

**Assembler symbols**

- `<Zda>`: Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- `<Zn>`: Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>`: Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, 2 * e + 1, 16];
    bits(16) element2 = Elem[operand2, 2 * e + 1, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);
Z[da, VL] = result;
```
**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.41   BFMMLA

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, irrespective of the other control bits in the FPCR, this instruction:

- Performs two unfused sums-of-products within each two pairs of adjacent BFloat16 elements while multiplying the 2×4 matrix of BFloat16 values held in each 128-bit segment of the first source vector by the 4×2 matrix of BFloat16 values in the corresponding segment of the second source vector. The intermediate single-precision products are rounded before they are summed and the intermediate sum is rounded before accumulation into the 2×2 single-precision matrix in the corresponding segment of the destination vector. This is equivalent to accumulating two 2-way unfused dot products per destination element.

- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.

- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).

- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.

- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.

- Generates only the Default NaN, as if FPCR.DN is 1.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs two fused sums-of-products within each two pairs of adjacent BFloat16 elements while multiplying the 2×4 matrix of BFloat16 values held in each 128-bit segment of the first source vector by the 4×2 matrix of BFloat16 values in the corresponding segment of the second source vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the 2×2 single-precision matrix in the corresponding segment of the destination vector. This is equivalent to accumulating two 2-way fused dot products per destination element.

- Generates only the default NaN, as if FPCR.DN is 1.

- Follows all other floating-point behaviors that apply to single-precision arithmetic, as controlled by the effective value of the FPCR in the current execution mode, and captured in the FPSR.

This instruction is unpredicated and vector length agnostic.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

**SVE**

(FEAT_BF16)

```plaintext
   [31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 5 | 4 | 0 ]
   0 1 1 0 1 0 0 0 0 1 1 Zm 1 1 1 0 0 1 Zn Zda
```

**Encoding**


**Decode for this encoding**

```plaintext
if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```
Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
bits(128) op1, op2;
bits(128) res, addend;

for s = 0 to segments-1
    op1 = Elem[operand1, s, 128];
    op2 = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
    res = BFMatMulAdd(addend, op1, op2);
    Elem[result, s, 128] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.42 BGRP

This instruction separates bits in each element of the first source vector by gathering from the bit positions indicated by non-zero bits in the corresponding mask element of the second source vector to the lowest-numbered contiguous bits of the corresponding destination element, and from positions indicated by zero bits to the highest-numbered bits of the destination element, preserving the bit order within each group. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BitPerm indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2

(FEAT_SVE_BitPerm)

Encoding

BGRP <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() || !HaveSVE2BitPerm() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) data = Z[n, VL];
bits(VL) mask = Z[m, VL];
bits(VL) result;

for e = 0 to elements - 1
   Elem[result, e, esize] = BitGroup(Elem[data, e, esize], Elem[mask, e, esize]);

Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.43  **BIC (immediate)**

Bitwise clear bits using immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is a pseudo-instruction of the **AND (immediate)** instruction. This means that:

- The encodings in this description are named to match the encodings of **AND (immediate)**.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of **AND (immediate)** gives the operational pseudocode for this instruction.

\[
\begin{array}{ccccccccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 & 23 & 22 & 21 & 20 & 19 & 18 & 17 & 5 & 4 & 0 \\
\hline
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & \text{imm13} & \text{Zdn} \\
\end{array}
\]

**Encoding**

BIC \(<\text{Zdn}.<\text{T}>, \ <\text{Zdn}.<\text{T}>, \ #<\text{const}>\)

is equivalent to

AND \(<\text{Zdn}.<\text{T}>, \ <\text{Zdn}.<\text{T}>, \ #(1-<\text{const}> - 1)\)

and is never the preferred disassembly.

**Assembler symbols**

\(<\text{Zdn}>\)  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

\(<\text{T}>\)  Is the size specifier, encoded in the "imm13<12>:imm13<5:0>" field. It can have the following values:

- **S**  when \(\text{imm13}<12> = 0, \text{imm13}<5:0> = 0xxxx\)
- **H**  when \(\text{imm13}<12> = 0, \text{imm13}<5:0> = 10xxxx\)
- **B**  when \(\text{imm13}<12> = 0, \text{imm13}<5:0> = 110xxx\)
- **B**  when \(\text{imm13}<12> = 0, \text{imm13}<5:0> = 1110xx\)
- **B**  when \(\text{imm13}<12> = 0, \text{imm13}<5:0> = 11110x\)
- **D**  when \(\text{imm13}<12> = 1, \text{imm13}<5:0> = xxxxxx\)

The following encodings are reserved:

- \(\text{imm13}<12> = 0, \text{imm13}<5:0> = 111110\).
- \(\text{imm13}<12> = 0, \text{imm13}<5:0> = 111111\).

\(<\text{const}>\)  Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

**Operation**

The description of **AND (immediate)** gives the operational pseudocode for this instruction.
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.44   BIC (predicates)

Bitwise AND inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

Encoding


Decode for this encoding

if !HaveSVE() & & !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 AND (NOT element2);
    else
        ElemP[result, e, esize] = '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.45  **BIC (vectors, predicated)**

Bitwise AND inverted active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & \text{Pg} & \text{Zm} & \text{Zdn} & \\
\end{array}
\]

**Encoding**

\[
\text{BIC } <\text{Zdn}>. <\text{T}>, <\text{Pg}>/M, <\text{Zdn}>. <\text{T}>, <\text{Zm}>. <\text{T}>
\]

**Decode for this encoding**

\[
\begin{align*}
\text{if } !\text{HaveSVE()} \&\& !\text{HaveSME()} \text{ then UNDEFINED;} \\
\text{constant integer esize = } 8 << \text{UInt(size);} \\
\text{integer } g = \text{UInt}(\text{Pg}); \\
\text{integer } dn = \text{UInt}(\text{Zdn}); \\
\text{integer } m = \text{UInt}(\text{Zm});
\end{align*}
\]

**Assembler symbols**

- \(<\text{Zdn}>\)  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- \(<\text{T}>\)  Is the size specifier, encoded in the "size" field. It can have the following values:
  - \(B\)  when size = 00
  - \(H\)  when size = 01
  - \(S\)  when size = 10
  - \(D\)  when size = 11
- \(<\text{Pg}>\)  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- \(<\text{Zm}>\)  Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

\[
\begin{align*}
\text{CheckSVEEnabled();} \\
\text{constant integer } VL = \text{CurrentVL;} \\
\text{constant integer } PL = VL \text{ DIV } 8; \\
\text{constant integer } elements = VL \text{ DIV } esize; \\
\text{bits}(PL) \text{ mask } = P[g, PL]; \\
\text{bits}(VL) \text{ operand1 } = Z[dn, VL]; \\
\text{bits}(VL) \text{ operand2 } = \text{if } \text{AnyActiveElement}(\text{mask, esize}) \text{ then } Z[m, VL] \text{ else } \text{Zeros}(VL); \\
\text{bits}(VL) \text{ result}; \\
\text{for } e = 0 \text{ to } \text{elements}-1 \\
\text{bits}(esize) \text{ element1 } = \text{Elem}(\text{operand1, } e, \text{ esize}); \\
\text{bits}(esize) \text{ element2 } = \text{Elem}(\text{operand2, } e, \text{ esize}); \\
\text{if } \text{Elem[mask, } e, \text{ esize]\ } = '1' \text{ then} \\
\quad \text{Elem[result, } e, \text{ esize] } = \text{element1 AND (NOT element2);} \\
\text{else} \\
\quad \text{Elem[result, } e, \text{ esize] } = \text{Elem(operand1, } e, \text{ esize);} \\
\text{Z[dn, VL] = result;}
\end{align*}
\]
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.46  BIC (vectors, unpredicated)

Bitwise AND inverted all elements of the second source vector with corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & Zm & 0 & 0 & 1 & 1 & 0 & 0 & Zn & Zd
\end{array}
\]

**Encoding**

BIC <Zd>.D, <Zn>.D, <Zm>.D

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
Z[d, VL] = operand1 AND (NOT operand2);

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.47   BICS

Bitwise AND inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 0 1 0 0 1 0 1 | 0 1 0 0 | Pm 0 1 | Pg 0 | Pn 1 | Pd 0
```

**Encoding**


**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

**Assembler symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` Is the name of the second source scalable predicate register, encoded in the "Pm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
for e = 0 to elements-1
  bit element1 = ElemP[operand1, e, esize];
  bit element2 = ElemP[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    ElemP[result, e, esize] = element1 AND (NOT element2);
  else
    ElemP[result, e, esize] = '0';
  
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.48   BRKA

Sets destination predicate elements up to and including the first active and true source element to true, then sets
subsequent elements to false. Inactive elements in the destination predicate register remain unmodified or are set to
zero, depending on whether merging or zeroing predication is selected. Does not set the condition flags.

Encoding

BRKA <Pd>.B, <Pg>/<ZM>, <Pn>.B

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = (M == '1');
boolean setflags = FALSE;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<ZM> Is the predication qualifier, encoded in the "M" field. It can have the following values:
Z when M = 0
M when M = 1
<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand = P[n, PL];
bits(PL) operand2 = P[d, PL];
boolean break = FALSE;
bits(PL) result;
for e = 0 to elements-1
  boolean element = ElemP[operand, e, esize] == '1';
  if ElemP[mask, e, esize] == '1' then
    ElemP[result, e, esize] = if !break then '1' else '0';
    break = break || element;
  elsif merging then
    ElemP[result, e, esize] = ElemP[operand2, e, esize];
  else
    ElemP[result, e, esize] = '0';
  if setflags then

001001010
0 0 1 0 0 1 0 1 0 0 0 0 0 0 1
B S

Pg 0
0 0 0 0

Pn 0
0 0 0 0

M 3
0 0 0 0

Pd 0
0 0 0 0
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
C8.2.49  **BRKAS**

Sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the **FIRST** (N), **NONE** (Z), ![LAST](C) condition flags based on the predicate result, and the V flag to zero.

**Encoding**

BRKAS <Pd>.B, <Pg>/Z, <Pn>.B

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = FALSE;
boolean setflags = TRUE;

**Assembler symbols**

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand = P[n, PL];
bits(PL) operand2 = P[d, PL];
boolean break = FALSE;
bits(PL) result;
for e = 0 to elements-1
  boolean element = ElemP[operand, e, esize] == '1';
  if ElemP[mask, e, esize] == '1' then
    ElemP[result, e, esize] = if !break then '1' else '0';
    break = break || element;
  elsif merging then
    ElemP[result, e, esize] = ElemP[operand2, e, esize];
  else
    ElemP[result, e, esize] = '0';
  if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.50 BRKB

Sets destination predicate elements up to but not including the first active and true source element to true, then sets
subsequent elements to false. Inactive elements in the destination predicate register remain unmodified or are set to
zero, depending on whether merging or zeroing predication is selected. Does not set the condition flags.

Encoding

BRKB <Pd>.B, <Pg>/<ZM>, <Pn>.B

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = (M == '1');
boolean setflags = FALSE;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<ZM> Is the predication qualifier, encoded in the "M" field. It can have the following values:
  Z when M = 0
  M when M = 1
<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand = P[n, PL];
bits(PL) operand2 = P[d, PL];
boolean break = FALSE;
bits(PL) result;
for e = 0 to elements-1
    boolean element = ElemP[operand, e, esize] == '1';
    if ElemP[mask, e, esize] == '1' then
        break = break || element;
    ElemP[result, e, esize] = if !break then '1' else '0';
    elsif merging then
        ElemP[result, e, esize] = ElemP[operand2, e, esize];
    else
        ElemP[result, e, esize] = '0';
    if setflags then
PSTATE.<\texttt{N,Z,C,V}> = \texttt{PredTest}(\texttt{mask}, \texttt{result}, \texttt{esize});
\texttt{P}[d, PL] = \texttt{result};
C8.2.51 BRKBS

Sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

Encoding

BRKBS <Pd>.B, <Pg>/Z, <Pn>.B

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = FALSE;
boolean setflags = TRUE;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand = P[n, PL];
bits(PL) operand2 = P[d, PL];
boolean break = FALSE;
bits(PL) result;
for e = 0 to elements-1
  boolean element = ElemP[operand, e, esize] == '1';
  if ElemP[mask, e, esize] == '1' then
    break = break || element;
  ElemP[result, e, esize] = if !break then '1' else '0';
  elsif merging then
    ElemP[result, e, esize] = ElemP[operand2, e, esize];
  else
    ElemP[result, e, esize] = '0';
  if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.52  BRKN

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise leaves the destination and second source predicate unchanged. Does not set the condition flags.

```
0 0 1 0 0 1 0 1 0 0 0 0 1 0 0 1 1 0 0 0 1 |   
Pg     0 Pn     0 Pdm

Nickel 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 | 10 9 8 | 5 4 3 0

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer dm = UInt(Pdm);
boolean setflags = FALSE;

Assembler symbols

<Pdm> Is the name of the second source and destination scalable predicate register, encoded in the "Pdm" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[dm, PL];
bits(PL) result;
if LastActive(mask, operand1, 8) == '1' then
    result = operand2;
else
    result = Zeros(PL);
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(Ones(PL), result, 8);
P[dm, PL] = result;
```
C8.2.53  **BRKNS**

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise leaves the destination and second source predicate unchanged. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 | 10  9  8 |  5  4  3  0 |
  0 0 1 0 0 1 0 1 | 0 1 1 0 0 0 0 1 |   Pg |  0  |  Pn |  0  |  Pdm |
```

**Encoding**


**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer dm = UInt(Pdm);
boolean setflags = TRUE;

**Assembler symbols**

<Pdm> Is the name of the second source and destination scalable predicate register, encoded in the "Pdm" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[dm, PL];
bits(PL) result;
if LastActive(mask, operand1, 8) == '1' then
  result = operand2;
else
  result = Zeros(PL);
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(Ones(PL), result, 8);
P[dm, PL] = result;

**Operational information**

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.54   BRKPA

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');
for e = 0 to elements-1
  if ElemP[mask, e, 8] == '1' then
    ElemP[result, e, 8] = if last then '1' else '0';
    last = last && (ElemP[operand2, e, 8] == '0');
  else
    ElemP[result, e, 8] = '0';
  if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
C8.2.55  BRKPAS

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

\[
\begin{array}{cccccccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & Pm & 1 & 1 & Pg & 0 & Pn & 0 & Pd & B
\end{array}
\]

**Encoding**


**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

**Assembler symbols**

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');
for e = 0 to elements-1
  if ElemP[mask, e, 8] == '1' then
    ElemP[result, e, 8] = if last then '1' else '0';
    last = last && (ElemP[operand2, e, 8] == '0');
  else
    ElemP[result, e, 8] = '0';
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.56 BRKPB

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12 11 10 9 8 7 6 5 4 3 0]
  0 0 1 0 0 1 0 1 0 0 0 1 1 0 0 Pm 1 1 Pg 0 Pn 1 Pd
```

**Encoding**


**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

**Assembler symbols**

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');
   for e = 0 to elements-1
      if ElemP[mask, e, 8] == '1' then
         last = last && (ElemP[operand2, e, 8] == '0');
         ElemP[result, e, 8] = if last then '1' else '0';
      else
         ElemP[result, e, 8] = '0';
   if setflags then
      PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
      P[d, PL] = result;
```
C8.2.57   BRKPBS

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler symbols

<Pd>  Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>  Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>  Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>  Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');
for e = 0 to elements-1
  if ElemP[mask, e, 8] == '1' then
    last = last && (ElemP[operand2, e, 8] == '0');
    ElemP[result, e, 8] = if last then '1' else '0';
  else
    ElemP[result, e, 8] = '0';
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.58   BSL

Selects bits from the first source vector where the corresponding bit in the third source vector is '1', and from the second source vector where the corresponding bit in the third source vector is '0'. The result is placed destructively in the destination and first source vector. This instruction is unpredicated.

Encoding


Decode for this encoding

```c
if (!HaveSVE2() && !HaveSME()) then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);
```

Assembler symbols

- `<Zdn>` is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zk>` is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];
Z[dn, VL] = (operand1 AND operand3) OR (operand2 AND NOT(operand3));
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.59   BSL1N

Selects bits from the inverted first source vector where the corresponding bit in the third source vector is '1', and from the second source vector where the corresponding bit in the third source vector is '0'. The result is placed destructively in the destination and first source vector. This instruction is unpredicated.

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm>   Is the name of the second source scalable vector register, encoded in the "Zm" field.
<Zk>   Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];

Z[dn, VL] = (NOT(operand1) AND operand3) OR (operand2 AND NOT(operand3));

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.60   BSL2N

Selects bits from the first source vector where the corresponding bit in the third source vector is '1', and from the inverted second source vector where the corresponding bit in the third source vector is '0'. The result is placed destructively in the destination and first source vector. This instruction is unpredicated.

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
<Zk> Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];
Z[dn, VL] = (operand1 AND operand3) OR (NOT(operand2) AND NOT(operand3));

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.61  CADD

Add the real and imaginary components of the integral complex numbers from the first source vector to the complex numbers from the second source vector which have first been rotated by 90 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, equivalent to multiplying the complex numbers in the second source vector by ±J beforehand. Destructively place the results in the corresponding elements of the first source vector. This instruction is unpredicated.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

Encoding

CADD <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, <const>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
calendar integer esize = 8 <<< Uint(size);
calendar integer m = Uint(Zm);
calendar integer dn = Uint(Zdn);
boolean sub_i = (rot == '0');
boolean sub_r = (rot == '1');

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in the "rot" field. It can have the following values:
#90 when rot = 0
#270 when rot = 1

Operation

CheckSVEEnabled();
calendar integer VL = CurrentVL;
calendar integer PL = VL DIV 8;
calendar integer pairs = VL DIV (2 * esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for p = 0 to pairs-1
    integer acc_r = SInt(Elem[operand1, 2 * p + 0, esize]);
integer acc_i = SInt(Elem[operand1, 2 + p + 1, esize]);
integer elt2_r = SInt(Elem[operand2, 2 + p + 0, esize]);
integer elt2_i = SInt(Elem[operand2, 2 + p + 1, esize]);
if sub_i then
  acc_r = acc_r - elt2_i;
  acc_i = acc_i + elt2_r;
if sub_r then
  acc_r = acc_r + elt2_i;
  acc_i = acc_i - elt2_r;
Elem[result, 2 + p + 0, esize] = acc_r<esize-1:0>
Elem[result, 2 + p + 1, esize] = acc_i<esize-1:0>
Z[dn, VL] = result;

Operational information
If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.62   CDOT (indexed)

The complex integer dot product instructions delimit the source vectors into pairs of 8-bit or 16-bit signed integer complex numbers. Within each pair, the complex numbers in the first source vector are multiplied by the corresponding complex numbers in the second source vector and the resulting wide real or wide imaginary part of the product is accumulated into a 32-bit or 64-bit destination vector element which overlaps all four of the elements that comprise a pair of complex number values in the first source vector.

As a result each instruction implicitly deinterleaves the real and imaginary components of their complex number inputs, so that the destination vector accumulates 4×wide real sums or 4×wide imaginary sums.

The complex numbers in the second source vector are rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, by performing the following transformations prior to the dot product operations:

- If the rotation is #0, the imaginary parts of the complex numbers in the second source vector are negated. The destination vector therefore accumulates the real parts of a complex dot product.
- If the rotation is #90, the real and imaginary parts of the complex numbers the second source vector are swapped. The destination vector therefore accumulates the imaginary parts of a complex dot product.
- If the rotation is #180, there is no transformation. The destination vector therefore accumulates the real parts of a complex conjugate dot product.
- If the rotation is #270, the real parts of the complex numbers in the second source vector are negated and then swapped with the imaginary parts. The destination vector therefore accumulates the imaginary parts of a complex conjugate dot product.

The indexed form of these instructions select a single pair of complex numbers within each 128-bit segment of the second source vector as the multiplier for all pairs of complex numbers within the corresponding 128-bit segment of the first source vector. The complex number pairs within the second source vector are specified using an immediate index which selects the same complex number pair position within each 128-bit vector segment. The index range is from 0 to one less than the number of complex number pairs per 128-bit segment, encoded in 1 or 2 bits depending on the size of the complex number pair.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

### 32-bit

```
| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 16 | 15 14 13 | 12 | 11 10 9 8 7 6 5 4 | 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 0 1 0 0 0 1 0 0 | 1 0 1 i2 Zm 0 1 0 | rot Zn Zda |

size<1> size<0>
```

**Encoding**


**Decode for this encoding**

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
```
integer sel_b = UInt(NOT(rot<0>));
boolean sub_i = (rot<0> == rot<1>);

64-bit

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0</td>
<td>1 0 0 1</td>
<td>1 1</td>
<td>i1</td>
<td>Zm</td>
<td>0 1 0 0</td>
<td>rot</td>
<td>Zn</td>
</tr>
</tbody>
</table>

size<1>
size<0>

Encoding


Decode for this encoding

if !HaveSVE2() & !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_i = (rot<0> == rot<1>);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.
For the 64-bit variant: is the immediate index of a 64-bit group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the "i1" field.

<const> Is the const specifier, encoded in the "rot" field. It can have the following values:

#0 when rot = 00
#90 when rot = 01
#180 when rot = 10
#270 when rot = 11

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 1
        integer elt1_r = SInt(Elem[operand1, 4 * e + 2 * i + 0, esize DIV 4]);
        integer elt1_i = SInt(Elem[operand1, 4 * e + 2 * i + 1, esize DIV 4]);
        integer elt2_a = SInt(Elem[operand2, 4 * s + 2 * i + sel_a, esize DIV 4]);
        integer elt2_b = SInt(Elem[operand2, 4 * s + 2 * i + sel_b, esize DIV 4]);
        if sub_i then
            res = res + (elt1_r * elt2_a) - (elt1_i * elt2_b);
        else
            res = res + (elt1_r * elt2_a) + (elt1_i * elt2_b);
        Elem[result, e, esize] = res;
    Z[da, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.63  CDOT (vectors)

The complex integer dot product instructions delimit the source vectors into pairs of 8-bit or 16-bit signed integer complex numbers. Within each pair, the complex numbers in the first source vector are multiplied by the corresponding complex numbers in the second source vector and the resulting wide real or wide imaginary part of the product is accumulated into a 32-bit or 64-bit destination vector element which overlaps all four of the elements that comprise a pair of complex number values in the first source vector.

As a result each instruction implicitly deinterleaves the real and imaginary components of their complex number inputs, so that the destination vector accumulates 4×wide real sums or 4×wide imaginary sums.

The complex numbers in the second source vector are rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, by performing the following transformations prior to the dot product operations:

- If the rotation is #0, the imaginary parts of the complex numbers in the second source vector are negated. The destination vector therefore accumulates the real parts of a complex dot product.
- If the rotation is #90, the real and imaginary parts of the complex numbers the second source vector are swapped. The destination vector therefore accumulates the imaginary parts of a complex dot product.
- If the rotation is #180, there is no transformation. The destination vector therefore accumulates the real parts of a complex conjugate dot product.
- If the rotation is #270, the real parts of the complex numbers in the second source vector are negated and then swapped with the imaginary parts. The destination vector therefore accumulates the imaginary parts of a complex conjugate dot product.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

```
[31 30 29 28][27 26 25 24][23 22 21 20]  16|15 14 13 12|11 10 9 |  5 4 |  0 |
  0 1 0 0 0 0 0 | 0 0 0 1 | 1 | Zm |  Zn | Zda
```

**Encoding**

CDOT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>, <const>

**Decode for this encoding**

```c
if !HaveSVE2() &amp; !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 &lt; UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot&lt;0&gt;);
integer sel_b = UInt(NOT(rot&lt;0&gt;));
bool sub_i = (rot&lt;0&gt; == rot&lt;1&gt;);
```

**Assembler symbols**

- `<Zda>`
  - Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- `<T>`
  - Is the size specifier, encoded in the "size&lt;0&gt;" field. It can have the following values:
    - 5\[\text{ when size}&lt;0&gt; = 0\]
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size<0>" field. It can have the following values:

- B when size<0> = 0
- H when size<0> = 1

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in the "rot" field. It can have the following values:

- #0 when rot = 00
- #90 when rot = 01
- #180 when rot = 10
- #270 when rot = 11

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements - 1
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 1
        integer elt1_r = SInt(Elem[operand1, 4 * e + 2 * i + 0, esize DIV 4]);
        integer elt1_i = SInt(Elem[operand1, 4 * e + 2 * i + 1, esize DIV 4]);
        integer elt2_a = SInt(Elem[operand2, 4 * e + 2 * i + sel_a, esize DIV 4]);
        integer elt2_b = SInt(Elem[operand2, 4 * e + 2 * i + sel_b, esize DIV 4]);
        if sub_i then
            res = res + (elt1_r * elt2_a) - (elt1_i * elt2_b);
        else
            res = res + (elt1_r * elt2_a) + (elt1_i * elt2_b);
        Elem[result, e, esize] = res;

Z[da, VL] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.64 CLASTA (scalar)

From the source vector register extract the element after the last active element, or if the last active element is the final element extract element zero, and then zero-extend that element to destructively place in the destination and first source general-purpose register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source general-purpose register.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 1 size 1 1 0 0 0 0 1 0 1 Pg Zm Rdn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

CLASTA <R><dn>, <Pg>, <R><dn>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Rdn);
integer m = UInt(Zm);
constant integer csize = if esize < 64 then 32 else 64;
boolean isBefore = FALSE;

Assembler symbols

<R> Is a width specifier, encoded in the "size" field. It can have the following values:
    W when size = 01
    W when size = x0
    X when size = 11

<dn> Is the number [0-30] of the source and destination general-purpose register or the name ZR (31), encoded in the "Rdn" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the source scalable vector register, encoded in the "Zm" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
    B when size = 00
    H when size = 01
    S when size = 10
    D when size = 11

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer esize = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = X[dn, esize];
bits(VL) operand2 = Z[m, VL];
bits(csize) result;
integer last = LastActiveElement(mask, esize);
if last < 0 then
    result = ZeroExtend(operand1, csize);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
        result = ZeroExtend(Elem[operand2, last, esize], csize);
    X[dn, csize] = result;

**Operational information**

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C8.2.65 CLASTA (SIMD&FP scalar)

From the source vector register extract the element after the last active element, or if the last active element is the final element extract element zero, and then zero-extend that element to destructively place in the destination and first source SIMD & floating-point scalar register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source SIMD & floating-point scalar register.

Encoding

CLASTA <V><dn>, <Pg>, <V><dn>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = $ << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Vdn);
integer m = UInt(Zm);
boolean isBefore = FALSE;

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

<dn> Is the number [0-31] of the source and destination SIMD&FP register, encoded in the "Vdn" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the source scalable vector register, encoded in the "Zm" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = V[dn, esize];
bits(VL) operand2 = Z[m, VL];
bits(esize) result;
integer last = LastActiveElement(mask, esize);
if last < 0 then
    result = ZeroExtend(operand1, esize);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
        result = Elem[operand2, last, esize];
    V[dn, esize] = result;
C8.2.66  CLASTA (vectors)

From the second source vector register extract the element after the last active element, or if the last active element
is the final element extract element zero, and then replicate that element to destructively fill the destination and first
source vector.

If there are no active elements then leave the destination and source vector unmodified.

Encoding

CLASTA <Zdn>.<T>, <Pg>, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean isBefore = FALSE;

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
integer last = LastActiveElement(mask, esize);
if last < 0 then
  result = operand1;
else
  if !isBefore then
    last = last + 1;
  if last >= elements then last = 0;
  for e = 0 to elements-1


\[
\text{Elem}[\text{result}, e, \text{esize}] = \text{Elem}[\text{operand2}, \text{last}, \text{esize}];
\]

\[
Z[dn, VL] = \text{result};
\]

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.67  CLASTB (scalar)

From the source vector register extract the last active element, and then zero-extend that element to destructively place in the destination and first source general-purpose register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source general-purpose register.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
<td>size</td>
<td>1 1 0 0 0 1 1 0 1</td>
<td>Pg</td>
</tr>
</tbody>
</table>

Encoding

CLASTB <R><dn>, <Pg>, <R><dn>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Rdn);
integer m = UInt(Zm);
constant integer csize = if esize < 64 then 32 else 64;
boolean isFirst = TRUE;

Assembler symbols

<R> Is a width specifier, encoded in the "size" field. It can have the following values:
W when size = 01
W when size = x0
X when size = 11

<dn> Is the number [0-30] of the source and destination general-purpose register or the name ZR (31), encoded in the "Rdn" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the source scalable vector register, encoded in the "Zm" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = X[dn, esize];
bits(VL) operand2 = Z[m, VL];
bits(csize) result;
integer last = LastActiveElement(mask, esize);
if last < 0 then
    result = ZeroExtend(operand1, csize);
else
    if !isBefore then
        last = last + 1;
    if last >= elements then last = 0;
    result = ZeroExtend(Elem[operand2, last, esize], csize);

X[dn, csize] = result;

**Operational information**

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C8.2.68   CLASTB (SIMD&FP scalar)

From the source vector register extract the last active element, and then zero-extend that element to destructively place in the destination and first source SIMD & floating-point scalar register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source SIMD & floating-point scalar register.

Encoding

CLASTB <V><dn>, <Pg>, <V><dn>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Vdn);
integer m = UInt(Zm);
boolean isBefore = TRUE;

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11

<dn> Is the number [0-31] of the source and destination SIMD&FP register, encoded in the "Vdn" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the source scalable vector register, encoded in the "Zm" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = V[dn, esize];
bits(VL) operand2 = Z[m, VL];
bits(esize) result;
integer last = LastActiveElement(mask, esize);
if last < 0 then
    result = ZeroExtend(operand1, esize);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
        result = Elem[operand2, last, esize];
    V[dn, esize] = result;
C8.2.69 CLASTB (vectors)

From the second source vector register extract the last active element, and then replicate that element to destructively fill the destination and first source vector.

If there are no active elements then leave the destination and source vector unmodified.

Encoding

CLASTB <Zdn>.<T>, <Pg>, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean isBefore = TRUE;

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
integer last = LastActiveElement(mask, esize);
if last < 0 then
    result = operand1;
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
    for e = 0 to elements-1
        Elem[result, e, esize] = Elem[operand2, last, esize];
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.70  CLS

Count leading sign bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Encoding

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0</td>
<td>1 0 0</td>
<td>0 1 1</td>
<td>0 0 0 0 1</td>
<td>0 1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = CountLeadingSignBits(element)<esize-1:0>
  
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.71   CLZ

Count leading zero bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10  9</th>
<th>5  4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0</td>
<td>1 0 1</td>
<td>0 0 1 1</td>
<td>1 1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

CLZ <Zd>.<T>, <Pg>/M, <Zn>.<T>

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

**Assembler symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = CountLeadingZeroBits(element)<esize-1:0>;
  Z[d, VL] = result;
```
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.72  CMLA (indexed)

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding the second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

Then add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the results in the corresponding elements of the addend vector. This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

16-bit

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
0 1 0 0 0 1 0 0 1 0 | 1 0 | 1 0 | i2 | Zm | 0 1 1 0 | rot | Zn | Zda
```

**Encoding**


**Decode for this encoding**

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<i2> == '1');
```

32-bit

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
0 1 0 0 0 1 0 0 1 0 | 1 1 | 1 1 | i1 | Zm | 0 1 1 0 | rot | Zn | Zda
```

**Encoding**

CMLA <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>

**Decode for this encoding**

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i1);
```
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> For the 16-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
For the 32-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm> For the 16-bit variant: is the element index, in the range 0 to 3, encoded in the “i2” field.
For the 32-bit variant: is the element index, in the range 0 to 1, encoded in the “i1” field.

<const> Is the const specifier, encoded in the “rot” field. It can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0</td>
<td>when rot = 00</td>
</tr>
<tr>
<td>#90</td>
<td>when rot = 01</td>
</tr>
<tr>
<td>#180</td>
<td>when rot = 10</td>
</tr>
<tr>
<td>#270</td>
<td>when rot = 11</td>
</tr>
</tbody>
</table>

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
constant integer pairspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for p = 0 to pairs-1
  integer segmentbase = p - (p MOD pairspersegment);
  integer s = segmentbase + index;
  integer elt1_a = SInt(Elem[operand1, 2 * p + sel_a, esize]);
  integer elt2_a = SInt(Elem[operand2, 2 * s + sel_a, esize]);
  integer elt2_b = SInt(Elem[operand2, 2 * s + sel_b, esize]);
  bits(esize) elt3_r = Elem[operand3, 2 * p + 0, esize];
  bits(esize) elt3_i = Elem[operand3, 2 * p + 1, esize];
  integer product_r = elt1_a * elt2_a;
  integer product_i = elt1_a * elt2_b;
  if sub_r then
    Elem[result, 2 * p + 0, esize] = elt3_r - product_r;
  else
    Elem[result, 2 * p + 0, esize] = elt3_r + product_r;
  if sub_i then
    Elem[result, 2 * p + 1, esize] = elt3_i - product_i;
  else
    Elem[result, 2 * p + 1, esize] = elt3_i + product_i;
Z[da, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.73   CMLA (vectors)

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in the first source vector by the corresponding complex number in the second source vector rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

Then add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the results in the corresponding elements of the addend vector. This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 15 14 13 12 11 10  9    5  4    0</td>
</tr>
<tr>
<td>0 1 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>
```

\[\text{Encoding}\]

CMLA <Zda>.<T>, <Zn>.<T>, <Zm>.<T>, <const>

\[\text{Decode for this encoding}\]

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');

\[\text{Assembler symbols}\]

<Zda>   Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<T>    Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11
<Zn>   Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>   Is the name of the second source scalable vector register, encoded in the "Zm" field.
<const> Is the const specifier, encoded in the "rot" field. It can have the following values:
#0 when rot = 00
#90 when rot = 01
#180 when rot = 10
#270 when rot = 11
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 @ esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for p = 0 to pairs-1
integer elt1_a = SInt(Elem[operand1, 2 @ p + sel_a, esize]);
integer elt2_a = SInt(Elem[operand2, 2 @ p + sel_a, esize]);
integer elt2_b = SInt(Elem[operand2, 2 @ p + sel_b, esize]);
bits(esize) elt3_r = Elem[operand3, 2 @ p + 0, esize];
bits(esize) elt3_i = Elem[operand3, 2 @ p + 1, esize];
integer product_r = elt1_a * elt2_a;
integer product_i = elt1_a * elt2_b;
if sub_r then
    Elem[result, 2 @ p + 0, esize] = elt3_r - product_r;
else
    Elem[result, 2 @ p + 0, esize] = elt3_r + product_r;
if sub_i then
    Elem[result, 2 @ p + 1, esize] = elt3_i - product_i;
else
    Elem[result, 2 @ p + 1, esize] = elt3_i + product_i;
Z[da, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand of this instruction.
C8.2.74  CMP<cc> (immediate)

Compare active integer elements in the source vector with an immediate, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

<table>
<thead>
<tr>
<th>&lt;cc&gt;</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>equal</td>
</tr>
<tr>
<td>GE</td>
<td>signed greater than or equal</td>
</tr>
<tr>
<td>GT</td>
<td>signed greater than</td>
</tr>
<tr>
<td>HI</td>
<td>unsigned higher than</td>
</tr>
<tr>
<td>HS</td>
<td>unsigned higher than or same</td>
</tr>
<tr>
<td>LE</td>
<td>signed less than or equal</td>
</tr>
<tr>
<td>LO</td>
<td>unsigned lower than</td>
</tr>
<tr>
<td>LS</td>
<td>unsigned lower than or same</td>
</tr>
<tr>
<td>LT</td>
<td>signed less than</td>
</tr>
<tr>
<td>NE</td>
<td>not equal</td>
</tr>
</tbody>
</table>

### Equal

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] [11 10 9 8] [7 6 5 4] [3 2 1 0]
  0 0 1 0 1 0 1 0 | size 0 | imm5 1 0 | Pg 0 | Zn 0 | Pd
```

### Encoding

CMPEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

### Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
integer imm = SInt(imm5);
boolean unsigned = FALSE;
```

### Greater than

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] [11 10 9 8] [7 6 5 4] [3 2 1 0]
  0 0 1 0 1 0 1 0 | size 0 | imm5 0 0 | Pg 0 | Zn 1 | Pd
```

lt
Encoding

CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

Decode for this encoding

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1</td>
<td>size 0</td>
<td>imm5 0 0 0</td>
<td>Pg</td>
<td>Zn</td>
<td>0</td>
<td>Pd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Greater than or equal

Encoding

CMPGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

Decode for this encoding

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1</td>
<td>size 1</td>
<td>imm7 0</td>
<td>Pg</td>
<td>Zn</td>
<td>1</td>
<td>Pd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Higher

Encoding

CMPHI <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

Decode for this encoding

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1</td>
<td>size 0</td>
<td>imm7 0 0</td>
<td>Pg</td>
<td>Zn</td>
<td>0</td>
<td>Pd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Higher or same

\begin{verbatim}
[31 30 29 28] [27 26 25 24] [23 22 21 20]  |  [14 13 12]  |  [10 9]  |  [5 4 3 0]  \\
0 0 1 0 | 0 1 0 0 | imm7 0 | Pg | Zn | 0 | Pd
\end{verbatim}

Encoding

CMPHS <Pd>.<T>, <Pg>/Z, <Zn>..<T>, #<imm>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEcmp op = Cmp_GE;
integer imm = UInt(imm7);
boolean unsigned = TRUE;

Less than

\begin{verbatim}
[31 30 29 28] [27 26 25 24] [23 22 21 20]  |  [16 15 13 12]  |  [10 9]  |  [5 4 3 0]  \\
0 0 1 0 | 0 1 0 1 | size 0 | imm5 0 | 0 | 1 | Pg | Zn | 0 | Pd
\end{verbatim}

Encoding

CMPLT <Pd>.<T>, <Pg>/Z, <Zn>..<T>, #<imm>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEcmp op = Cmp_LT;
integer imm = UInt(imm5);
boolean unsigned = FALSE;

Less than or equal

\begin{verbatim}
[31 30 29 28] [27 26 25 24] [23 22 21 20]  |  [16 15 13 12]  |  [10 9]  |  [5 4 3 0]  \\
0 0 1 0 | 0 1 0 1 | size 0 | imm5 0 | 0 | 1 | Pg | Zn | 1 | Pd
\end{verbatim}

Encoding

CMPLE <Pd>.<T>, <Pg>/Z, <Zn>..<T>, #<imm>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
integer imm = SInt(imm5);
boolean unsigned = FALSE;

Lower

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th>imm7</th>
<th>Pg</th>
<th>Zn</th>
<th>Pd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Encoding

CMPL <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
integer imm = UInt(imm7);
boolean unsigned = TRUE;

Lower or same

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th>imm7</th>
<th>Pg</th>
<th>Zn</th>
<th>Pd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Encoding

CMPS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
integer imm = UInt(imm7);
boolean unsigned = TRUE;

Not equal

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th>imm5</th>
<th>Pg</th>
<th>Zn</th>
<th>Pd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Encoding

`CMPNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>`

Decode for this encoding

if `!HaveSVE() && !HaveSME()` then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_NE;
integer imm = SInt(imm5);
boolean unsigned = FALSE;

Assembler symbols

`<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
`<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
- `B` when size = 00
- `H` when size = 01
- `S` when size = 10
- `D` when size = 11
`<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
`<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.
`<imm>` For the equal, greater than, greater than or equal, less than, less than or equal and not equal variant: is the signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.
For the higher, higher or same, lower and lower or same variant: is the unsigned immediate operand, in the range 0 to 127, encoded in the "imm7" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(PL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  if ElemP[mask, e, esize] == '1' then
    boolean cond;
    case op of
      when Cmp_EQ cond = element1 == imm;
      when Cmp_NE cond = element1 != imm;
      when Cmp_GE cond = element1 >= imm;
      when Cmp_LT cond = element1 <  imm;
      when Cmp_GT cond = element1 >  imm;
      when Cmp_LE cond = element1 <= imm;
    ElemP[result, e, esize] = if cond then '1' else '0';
    else
      ElemP[result, e, esize] = '0';
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.
C8.2.75   CMP<cc> (vectors)

Compare active integer elements in the first source vector with corresponding elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

<table>
<thead>
<tr>
<th>&lt;cc&gt;</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>equal</td>
</tr>
<tr>
<td>GE</td>
<td>signed greater than or equal</td>
</tr>
<tr>
<td>GT</td>
<td>signed greater than</td>
</tr>
<tr>
<td>HI</td>
<td>unsigned higher than</td>
</tr>
<tr>
<td>HS</td>
<td>unsigned higher than or same</td>
</tr>
<tr>
<td>NE</td>
<td>not equal</td>
</tr>
</tbody>
</table>

This instruction is used by the pseudo-instructions CMPLE (vectors), CMPLO (vectors), CMPLS (vectors), and CMPLT (vectors). The pseudo-instruction is never the preferred disassembly.

**Equal**

| 31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13 12| 10 9 | 5 4 | 3 | 0 | 0 0 1 0 0 1 0 0 | size 0 | Zm | 1 | 0 | 1 | Pg | Zn | 0 | Pd |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| ne  |

**Encoding**

CMPEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size); 
integer g = UInt(Pg); 
integer n = UInt(Zn); 
integer m = UInt(Zm); 
integer d = UInt(Pd); 
SVECmp op = Cmp_EQ; 
boolean unsigned = FALSE;

**Greater than**

| 31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13 12| 10 9 | 5 4 | 3 | 0 | 0 0 1 0 0 1 0 0 | size 0 | Zm | 1 | 0 | 0 | Pg | Zn | 1 | Pd |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| ne  |

**Encoding**

CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
boolean unsigned = FALSE;

Greater than or equal

Encoding
CMPGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
boolean unsigned = FALSE;

Higher

Encoding
CMPHI <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
boolean unsigned = TRUE;
Higher or same

<table>
<thead>
<tr>
<th>Size</th>
<th>Zm</th>
<th>Pg</th>
<th>Zn</th>
<th>Pd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 0</td>
<td>0 0 0</td>
<td>1 0 1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

CMPHS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
boolean unsigned = TRUE;

Not equal

<table>
<thead>
<tr>
<th>Size</th>
<th>Zm</th>
<th>Pg</th>
<th>Zn</th>
<th>Pd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 0</td>
<td>0 0 0</td>
<td>1 0 1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

CMPNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_NE;
boolean unsigned = FALSE;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;

bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(PL) result;

for e = 0 to elements-1
   integer element1 = Int(Elem[operand1, e, esize], unsigned);
   if ElemP[mask, e, esize] == '1' then
      boolean cond;
      integer element2 = Int(Elem[operand2, e, esize], unsigned);
      case op of
         when Cmp_EQ cond = element1 == element2;
         when Cmp_NE cond = element1 != element2;
         when Cmp_GE cond = element1 >= element2;
         when Cmp_LT cond = element1 < element2;
         when Cmp_GT cond = element1 > element2;
         when Cmp_LE cond = element1 <= element2;
      ElemP[result, e, esize] = if cond then '1' else '0';
      else
         ElemP[result, e, esize] = '0';
   PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.
C8.2.76   CMP<cc> (wide elements)

Compare active integer elements in the first source vector with overlapping 64-bit doubleword elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

<table>
<thead>
<tr>
<th>&lt;cc&gt;</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>equal</td>
</tr>
<tr>
<td>GE</td>
<td>signed greater than or equal</td>
</tr>
<tr>
<td>GT</td>
<td>signed greater than</td>
</tr>
<tr>
<td>HI</td>
<td>unsigned higher than</td>
</tr>
<tr>
<td>HS</td>
<td>unsigned higher than or same</td>
</tr>
<tr>
<td>LE</td>
<td>signed less than or equal</td>
</tr>
<tr>
<td>LO</td>
<td>unsigned lower than</td>
</tr>
<tr>
<td>LS</td>
<td>unsigned lower than or same</td>
</tr>
<tr>
<td>LT</td>
<td>signed less than</td>
</tr>
<tr>
<td>NE</td>
<td>not equal</td>
</tr>
</tbody>
</table>

**Equal**

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] [11 10 9 8] [7 6 5 4] [3 2 1 0]
```

![Encoding Diagram]

**Encoding**

CMPEQ <Pd>, <Pg>, <Zn>, <Zm>.D

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
boolean unsigned = FALSE;
```

**Greater than**

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] [11 10 9 8] [7 6 5 4] [3 2 1 0]
```

![Encoding Diagram]
**Encoding**

CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
boolean unsigned = FALSE;

**Greater than or equal**

CMPGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
boolean unsigned = FALSE;

**Higher**

CMPI <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
boolean unsigned = TRUE;
Higher or same

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [16 15 14 13 12] [10 9] [5 4 3] [0]
0 0 1 0 0 1 0 0 size 0 Zm 0 1 1 Pg Zn 0 Pd
```

**Encoding**

```
CMPHS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D
```

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
boolean unsigned = TRUE;
```

Less than

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [16 15 14 13 12] [10 9] [5 4 3] [0]
0 0 1 0 0 1 0 0 size 0 Zm 0 1 1 Pg Zn 0 Pd
```

**Encoding**

```
CMPLT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D
```

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
boolean unsigned = FALSE;
```

Less than or equal

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [16 15 14 13 12] [10 9] [5 4 3] [0]
0 0 1 0 0 1 0 0 size 0 Zm 0 1 1 Pg Zn 1 Pd
```

**Encoding**

```
CMPLT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D
```
**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
boolean unsigned = FALSE;

**Lower**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 0</td>
<td>size</td>
<td>0</td>
<td>Zm</td>
</tr>
</tbody>
</table>

**Encoding**

CMPL0 <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
boolean unsigned = TRUE;

**Lower or same**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 0</td>
<td>size</td>
<td>0</td>
<td>Zm</td>
</tr>
</tbody>
</table>

**Encoding**

CMPLS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
boolean unsigned = TRUE;
Not equal

```
[31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13 12]| 10| 9| 5|4|3 |0
0|0|1|0|0|1|0|0|size|0|Zm|0|0|1|Pg|Zn|1|Pd
```

Encoding

CMPNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_NE;
boolean unsigned = FALSE;
```

Assembler symbols

- `<Pd>` is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<T>` is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - The encoding size = 11 is reserved.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation for all encodings

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(PL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  if Elem[mask, e, esize] == '1' then
    boolean cond;
    integer element2 = Int(Elem[operand2, (e * esize) DIV 64, 64], unsigned);
    case op of
      when Cmp_EQ cond = element1 == element2;
      when Cmp_NE cond = element1 != element2;
      when Cmp_GE cond = element1 >= element2;
      when Cmp_LT cond = element1 <  element2;
      when Cmp_GT cond = element1 >  element2;
      when Cmp_LE cond = element1 <= element2;
```
ElemP[result, e, esize] = if cond then '1' else '0';
else
    ElemP[result, e, esize] = '0';

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to synchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.
C8.2.77 CMPE (vectors)

Compare active signed integer elements in the first source vector being less than or equal to corresponding signed
integer elements in the second source vector, and place the boolean results of the comparison in the corresponding elements
of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N),
NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is a pseudo-instruction of the CMP<cc> (vectors) instruction. This means that:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

**Encoding**

CMPLE <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

CMPGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

and is never the preferred disassembly.

**Assembler symbols**

<Pd>  Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

- B  when size = 00
- H  when size = 01
- S  when size = 10
- D  when size = 11

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

**Operation**

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register
    contains the same value for each execution.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.
C8.2.78  CMPLO (vectors)

Compare active unsigned integer elements in the first source vector being lower than corresponding unsigned
elements in the second source vector, and place the boolean results of the comparison in the corresponding elements
of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N),
NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is a pseudo-instruction of the CMP<cc> (vectors) instruction. This means that:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

Encoding

```
CMPLO <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>
```

is equivalent to

```
CMPHI   <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

and is never the preferred disassembly.

Assembler symbols

- `<Pd>`  Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Zm>`  Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zn>`  Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<T>`  Is the size specifier, encoded in the "size" field. It can have the following values:
  - `B` when size = 00
  - `H` when size = 01
  - `S` when size = 10
  - `D` when size = 11
- `<Pg>`  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register
    contains the same value for each execution.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register
    contains the same value for each execution.
  — The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is
dependent on the predicate register or NZCV condition flags written by this instruction might be significantly
delayed.
C8.2.79 CMPLS (vectors)

Compare active unsigned integer elements in the first source vector being lower than or same as corresponding unsigned elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is a pseudo-instruction of the CMP<cc> (vectors) instruction. This means that:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

### Encoding

CMPLS <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

CMPHS   <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

and is never the preferred disassembly.

### Assembler symbols

- **<Pd>** Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.
- **<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.
- **<T>** Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

### Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

### Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.
C8.2.80 CMPLT (vectors)

Compare active signed integer elements in the first source vector being less than corresponding signed elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is a pseudo-instruction of the CMP<cc> (vectors) instruction. This means that:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

![Encoding](image)

**Encoding**

CMPLT <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

and is never the preferred disassembly.

**Assembler symbols**

- **<Pd>** Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.
- **<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.
- **<T>** Is the size specifier, encoded in the "size" field. It can have the following values:
  - **B** when size = 00
  - **H** when size = 01
  - **S** when size = 10
  - **D** when size = 11
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

**Operation**

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.
C8.2.81  CNOT

Logically invert the boolean value in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Boolean TRUE is any non-zero value in a source, and one in a result element. Boolean FALSE is always zero.

Encoding

CNOT <Zd>.<T>, <Pg>/M, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1'
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = ZeroExtend(IsZeroBit(element), esize);
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.82  CNT

Count non-zero bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 ]
0 0 0 0 0 1 0 0 | size 0 1 1 0 1 0 1 | Pg Zn Zd
```

Encoding

CNT <Zd>.<T>, <Pg>/M, <Zn>.<T>

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
ing = UInt(Pg);
n = UInt(Zn);
d = UInt(Zd);
```

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn>  Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = BitCount(element)<esize-1:0>;
Z[d, VL] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.83  CNTB, CNTD, CNTH, CNTW

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then places the result in the scalar destination.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

Byte

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19</th>
<th>16 15 14 13 12 11 10 9 5 4 0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>0 0 1 0</td>
<td>imm4</td>
</tr>
</tbody>
</table>

Encoding

CNTB <Xd>{, <pattern>{, MUL #<imm>}}</p>

Decode for this encoding

if !HaveSVE() & !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Doubleword

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19</th>
<th>16 15 14 13 12 11 10 9 5 4 0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>1 1 1 0</td>
<td>imm4</td>
</tr>
</tbody>
</table>

Encoding

CNTD <Xd>{, <pattern>{, MUL #<imm>}}</p>

Decode for this encoding

if !HaveSVE() & !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
## Halfword

![Binary representation](binary_representation.png)

**Encoding**

\[ \text{CNTH} \langle Xd \rangle\{, <\text{pattern}>\{, \text{MUL} \#<\text{imm}>\}\} \]

**Decode for this encoding**

If \(!\text{HaveSVE()} \&\& \!\text{HaveSME()}\) then UNDEFINED;

- constant integer \(esize = 16\);
- integer \(d = \text{UInt}(Rd)\);
- bits(5) \(pat = \text{pattern}\);
- integer \(imm = \text{UInt}(\text{imm4}) + 1\);

## Word

![Binary representation](binary_representation.png)

**Encoding**

\[ \text{CNTW} \langle Xd \rangle\{, <\text{pattern}>\{, \text{MUL} \#<\text{imm}>\}\} \]

**Decode for this encoding**

If \(!\text{HaveSVE()} \&\& \!\text{HaveSME()}\) then UNDEFINED;

- constant integer \(esize = 32\);
- integer \(d = \text{UInt}(Rd)\);
- bits(5) \(pat = \text{pattern}\);
- integer \(imm = \text{UInt}(\text{imm4}) + 1\);

## Assembler symbols

- \(<Xd>\) is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.
- \(<\text{pattern}>\) is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:
  - \(\text{POW2}\) when pattern = 00000
  - \(\text{VL1}\) when pattern = 00001
  - \(\text{VL2}\) when pattern = 00010
  - \(\text{VL3}\) when pattern = 00011
  - \(\text{VL4}\) when pattern = 00100
  - \(\text{VL5}\) when pattern = 00101
  - \(\text{VL6}\) when pattern = 00110
  - \(\text{VL7}\) when pattern = 00111
SVE Instruction Descriptions
C8.2 Alphabetical list of SVE instructions

VL8  when pattern = 01000
VL16 when pattern = 01001
VL32 when pattern = 01010
VL64 when pattern = 01011
VL128 when pattern = 01100
VL256 when pattern = 01101
#uimm5 when pattern = 0111x
#uimm5 when pattern = 101x1
#uimm5 when pattern = 10110
#uimm5 when pattern = 1x0x1
#uimm5 when pattern = 1x010
#uimm5 when pattern = 1xx00
MUL4 when pattern = 11101
MUL3 when pattern = 11110
ALL  when pattern = 11111

<imm>  Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation for all encodings

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
X[d, 64] = (count * imm)<63:0>;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

•  The execution time of this instruction is independent of:
   —  The values of the data supplied in any of its registers.
   —  The values of the NZCV flags.

•  The response of this instruction to asynchronous exceptions does not vary based on:
   —  The values of the data supplied in any of its registers.
   —  The values of the NZCV flags.
C8.2.84  CNTP

Counts the number of active and true elements in the source predicate and places the scalar result in the destination general-purpose register. Inactive predicate elements are not counted.

Encoding

CNTP <Xd>, <Pg>, <Pn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Rd);

Assembler symbols

<Xd>  Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.
<Pg>  Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>  Is the name of the source scalable predicate register, encoded in the "Pn" field.
<T>  Is the size specifier, encoded in the "size" field. It can have the following values:
     B  when size = 00
     H  when size = 01
     S  when size = 10
     D  when size = 11

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand = P[n, PL];
bits(64) sum = Zeros(64);

for e = 0 to elements-1
   if ElemP[mask, e, esize] == '1' && ElemP[operand, e, esize] == '1' then
      sum = sum + 1;
X[d, 64] = sum;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C8.2.85 COMPACT

Read the active elements from the source vector and pack them into the lowest-numbered elements of the
destination vector. Then set any remaining elements of the destination vector to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and
enabled at the current Exception level.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 1</td>
<td>size 1 0 0 0 1 1 0 0</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

COMPACT <Zd>.<T>, <Pg>, <Zn>.<T>

Decode for this encoding

if !HaveSVE() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = $ << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size<0>" field. It can have the following values:
S when size<0> = 0
D when size<0> = 1
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Zeros(VL);
integer x = 0;
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand1, e, esize];
        Elem[result, x, esize] = element;
        x = x + 1;
Z[d, VL] = result;
C8.2.86   CPY (immediate, merging)

Copy a signed integer immediate to each active element in the destination vector. Inactive elements in the
destination vector register remain unmodified.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it
may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option
is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit
value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This instruction is used by the pseudo-instructions FMOV (zero, predicated) and MOV (immediate, predicated,
merging). The alias is always the preferred disassembly.

Encoding
CPY <Zd>.<T>, <Pg>/M, #<imm>{, <shift>}

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Zd);
boolean merging = TRUE;
integer imm = SInt(imm8);
if sh == '1' then imm = imm << 8;

Alias conditions

<table>
<thead>
<tr>
<th>Alias or pseudo-instruction</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMOV (zero, predicated)</td>
<td>Never</td>
</tr>
<tr>
<td>MOV (immediate, predicated, merging)</td>
<td>Unconditionally</td>
</tr>
</tbody>
</table>

Assembler symbols

<Zd>   Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>   Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Pg>   Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<imm>   Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.
Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:

- **LSL #0** when \( \text{sh} = 0 \)
- **LSL #8** when \( \text{sh} = 1 \)

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) dest = Z[d, VL];
bits(VL) result;

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = imm<esize-1:0>;
  elsif merging then
    Elem[result, e, esize] = Elem[dest, e, esize];
  else
    Elem[result, e, esize] = Zeros(esize);
  Z[d, VL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.87  CPY (immediate, zeroing)

Copy a signed integer immediate to each active element in the destination vector. Inactive elements in the
destination vector register are set to zero.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it
may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option
is specified is "#<imm>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit
value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This instruction is used by the alias MOV (immediate, predicated, zeroing). The alias is always the preferred
disassembly.

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12| 5  4 | 0 |
|0 0 0 0 0 1 0 1|size 0 1|Pg 0 0 sh|imm8|Zd|
```

**Encoding**

CPY <Zd>.<T>, <Pg>./Z, #<imm>{, <shift>}

**Decode for this encoding**

```java
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Zd);
boolean merging = FALSE;
integer imm = SInt(imm8);
if sh == '1' then imm = imm << 8;
```

**Assembler symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<imm>` Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh"
  field. It can have the following values:
  - LSL #0 when sh = 0
  - LSL #8 when sh = 1
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) dest = Z[d, VL];
bits(VL) result;

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = imm<esize-1:0>;
  elsif merging then
    Elem[result, e, esize] = Elem[dest, e, esize];
  else
    Elem[result, e, esize] = Zeros(esize);
  Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register
    contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register
    contains the same value for each execution.
  — The values of the NZCV flags.
**C8.2.88 CPY (scalar)**

Copy the general-purpose scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is used by the alias MOV (scalar, predicated). The alias is always the preferred disassembly.

**Encoding**

CPY <Zd>.<T>, <Pg>/M, <R><n|SP>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Rn);
integer d = UInt(Zd);

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<R> Is a width specifier, encoded in the "size" field. It can have the following values:

- W when size = 01
- W when size = x0
- X when size = 11

<n|SP> Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) result = Z[d, VL];
if AnyActiveElement(mask, esize) then
  bits(64) operand1;
  if n == 31 then
    operand1 = SP[];
  else
operand1 = X[n, 64];

for e = 0 to elements-1
  if Elem[ mask, e, esize] == '1'
    Elem[ result, e, esize] = operand1<esize-1:0>;

Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.89   CPY (SIMD&FP scalar)

Copy the SIMD & floating-point scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is used by the alias MOV (SIMD&FP scalar, predicated). The alias is always the preferred disassembly.

Encoding

```
CPY <Zd>.<T>, <Pg>/M, <V><n>
```

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Vn);
integer d = UInt(Zd);
```

Assembler symbols

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>`: Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<V>`: Is a width specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<n>`: Is the number [0-31] of the source SIMD&FP register, encoded in the "Vn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = if AnyActiveElement(mask, esize) then V[n, esize] else Zeros(esize);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    [31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 10 9 8 7 6 5 4 0]
    0 0 0 0 1 0 1 | 1 0 0 0 0 1 0 0 | Pg | Vn | Zd
```
Elt[\text{result, e, esize}] = \text{operand1};
Z[d, \text{VL}] = \text{result};

Operational information

If \text{FEAT\_SVE2} is implemented or \text{FEAT\_SME} is implemented, then when \text{PSTATE.DIT} is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.90 CTERMEQ, CTERMNE

Detect termination conditions in serialized vector loops. Tests whether the comparison between the scalar source operands holds true and if not tests the state of the !LAST condition flag (C) which indicates whether the previous flag-setting predicate instruction selected the last element of the vector partition.

The Z and C condition flags are preserved by this instruction. The N and V condition flags are set as a pair to generate one of the following conditions for a subsequent conditional instruction:

<table>
<thead>
<tr>
<th>Condition</th>
<th>N</th>
<th>V</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GE</td>
<td>0</td>
<td>0</td>
<td>Continue loop (compare failed and last element not selected)</td>
</tr>
<tr>
<td>LT</td>
<td>0</td>
<td>1</td>
<td>Terminate loop (last element selected)</td>
</tr>
<tr>
<td>LT</td>
<td>1</td>
<td>0</td>
<td>Terminate loop (compare selected)</td>
</tr>
<tr>
<td>GE</td>
<td>1</td>
<td>1</td>
<td>Never generated</td>
</tr>
</tbody>
</table>

The scalar source operands are 32-bit or 64-bit general-purpose registers of the same size.

**Equal**

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9</th>
<th>8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 1 sz 1 Rm 0 0 1 0 0</td>
<td>Rn 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

CTERMEQ <R><n>, <R><m>

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Rn);
integer m = UInt(Rm);
SVECmp op = Cmp_EQ;
```

**Not equal**

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9</th>
<th>8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 1 sz 1 Rm 0 0 1 0 0</td>
<td>Rn 1 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

CTERMNE <R><n>, <R><m>

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Rn);
integer m = UInt(Rm);
SVECmp op = Cmp_NE;
```
Assembler symbols

<r>    Is a width specifier, encoded in the "sz" field. It can have the following values:
    W    when sz = 0
    X    when sz = 1

<n>    Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m>    Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation for all encodings

    CheckSVEEnabled();
    bits(esize) operand1 = X[n, esize];
    bits(esize) operand2 = X[m, esize];
    integer element1 = UInt(operand1);
    integer element2 = UInt(operand2);
    boolean term;

    case op of
    when Cmp_EQ term = element1 == element2;
    when Cmp_NE term = element1 != element2;
    if term then
        PSTATE.N = '1';
        PSTATE.V = '0';
    else
        PSTATE.N = '0';
        PSTATE.V = (NOT PSTATE.C);

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

•    The execution time of this instruction is independent of:
    —    The values of the data supplied in any of its registers.
    —    The values of the NZCV flags.

•    The response of this instruction to asynchronous exceptions does not vary based on:
    —    The values of the data supplied in any of its registers.
    —    The values of the NZCV flags.
C8.2.91   DECB, DECD, DECH, DECW (scalar)

Determines the number of active elements implied by the named predicate constraint, multiplies that by an
immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination.

The named predicate constraint limits the number of active elements in a single predicate to:

• A fixed number (VL1 to VL256)
• The largest power of two (POW2)
• The largest multiple of three or four (MUL3 or MUL4)
• All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than
Undefined Instruction exception.

Byte

| 31 30 29 28|27 26 25 24|23 22 21 20|19 16 15 14 13 12|11 10 9 | 5 4 | 0 |
|-------------|-------------|-------------|-------------|-------------|-------------|
| 0 0 0 0 0 0 | 1 0 0 | 0 | 0 | 1 | 1 | imm4 |
| size<1>     | size<0>     | D            |

Encoding

DECB <Xdn>{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Doubleword

| 31 30 29 28|27 26 25 24|23 22 21 20|19 16 15 14 13 12|11 10 9 | 5 4 | 0 |
|-------------|-------------|-------------|-------------|-------------|-------------|
| 0 0 0 0 0 0 | 1 0 0 | 1 | 1 | 1 | imm4 |
| size<1>     | size<0>     | D            |

Encoding

DECD <Xdn>{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
Halfword

DECH <Xdn>{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Word

DECW <Xdn>{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Assembler symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:
    PDW2 when pattern = 00000
    VL1 when pattern = 00001
    VL2 when pattern = 00010
    VL3 when pattern = 00011
    VL4 when pattern = 00100
    VL5 when pattern = 00101
    VL6 when pattern = 00110
    VL7 when pattern = 00111
VL8 when pattern = 01000
VL16 when pattern = 01001
VL32 when pattern = 01010
VL64 when pattern = 01011
VL128 when pattern = 01100
VL256 when pattern = 01101
#uimm5 when pattern = 0111x
#uimm5 when pattern = 101x1
#uimm5 when pattern = 10110
#uimm5 when pattern = 1x0x1
#uimm5 when pattern = 1x010
#uimm5 when pattern = 1xx00
MUL4 when pattern = 11101
MUL3 when pattern = 11110
ALL when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation for all encodings

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
constant integer VL = CurrentVL;
bits(64) operand1 = X[dn, 64];

X[dn, 64] = operand1 - (count * imm);

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.92 DECD, DECH, DECW (vector)

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

### Doubleword

**Encoding**

DECD <Zdn>.D{, <pattern>{, MUL #<imm>}}

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

### Halfword

**Encoding**

DECH <Zdn>.H{, <pattern>{, MUL #<imm>}}

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
Word

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & \text{imm4} & 1 & 1 & 0 & 0 & 0 & 1 & \text{pattern} & \text{Zdn} & \\
\end{array}
\]

Encoding

\[
\text{DECW} \ <\text{Zdn}\>.S\{, \ <\text{pattern}\>{, \ MUL \ #<\text{imm}\>}}
\]

Decode for this encoding

\[
\text{if} \ !\text{HaveSVE()} \ & \ !\text{HaveSME()} \text{ then UNDEFINED;}
\]
\[
\text{constant integer esize = 32;}
\]
\[
\text{integer dn = UInt(Zdn);}
\]
\[
\text{bits(5) pat = pattern;}
\]
\[
\text{integer imm = UInt(imm4) + 1;}
\]

Assembler symbols

\text{<Zdn>} \quad \text{Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.}

\text{<pattern>} \quad \text{Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:}

- \text{POW2} \quad \text{when pattern = 00000}
- \text{VL1} \quad \text{when pattern = 00001}
- \text{VL2} \quad \text{when pattern = 00010}
- \text{VL3} \quad \text{when pattern = 00011}
- \text{VL4} \quad \text{when pattern = 00100}
- \text{VL5} \quad \text{when pattern = 00101}
- \text{VL6} \quad \text{when pattern = 00110}
- \text{VL7} \quad \text{when pattern = 00111}
- \text{VL8} \quad \text{when pattern = 01000}
- \text{VL16} \quad \text{when pattern = 01001}
- \text{VL32} \quad \text{when pattern = 01010}
- \text{VL64} \quad \text{when pattern = 01011}
- \text{VL128} \quad \text{when pattern = 01100}
- \text{VL256} \quad \text{when pattern = 01101}
- \#uimm5 \quad \text{when pattern = 0111x}
- \#uimm5 \quad \text{when pattern = 101x1}
- \#uimm5 \quad \text{when pattern = 10110}
- \#uimm5 \quad \text{when pattern = 1x0x1}
- \#uimm5 \quad \text{when pattern = 1x010}
- \#uimm5 \quad \text{when pattern = 1xx00}
- \text{MUL4} \quad \text{when pattern = 11101}
- \text{MUL3} \quad \text{when pattern = 11110}
- \text{ALL} \quad \text{when pattern = 11111}
<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation for all encodings**

```plaintext
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
   Elem[result, e, esize] = Elem[operand1, e, esize] - (count * imm);
Z[dn, VL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.93  DECP (scalar)

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9  8</th>
<th>5  4  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1</td>
<td>0 1 0 1 1 0 1 0</td>
<td>0 0 0 1 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

DECP <Xdn>, <Pm>.<T>

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
```

**Assembler symbols**

- `<Xdn>` Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- `<Pm>` Is the name of the source scalable predicate register, encoded in the "Pm" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) operand1 = X[dn, 64];
bits(PL) operand2 = P[m, PL];
integer count = 0;
for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;
X[dn, 64] = operand1 - count;
```

**Operational information**

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C8.2.94   DECP (vector)

Counts the number of true elements in the source predicate and then uses the result to decrement all destination vector elements.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

Encoding

DECP <Zdn>.<T>, <Pm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = $ << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<T>   Is the size specifier, encoded in the "size" field. It can have the following values:
   H    when size = 01
   S    when size = 10
   D    when size = 11
The encoding size = 00 is reserved.
<Pm>  Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
   if Elem[operand2, e, esize] == '1' then
      count = count + 1;
for e = 0 to elements-1
   Elem[result, e, esize] = Elem[operand1, e, esize] - count;
Z[dn, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.95    DUP (immediate)

Unconditionally broadcast the signed integer immediate into each element of the destination vector. This instruction is unpredicated.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This instruction is used by the pseudo-instructions FMOV (zero, unpredicated) and MOV (immediate, unpredicated). The alias is always the preferred disassembly.

Encoding

DUP <Zd>.<T>, #<imm>{, <shift>}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer d = UInt(Zd);
integer imm = SInt(imm8);
if sh == '1' then imm = imm << 8;

Alias conditions

<table>
<thead>
<tr>
<th>Alias or pseudo-instruction</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMOV (zero, unpredicated)</td>
<td>Never</td>
</tr>
<tr>
<td>MOV (immediate, unpredicated)</td>
<td>Unconditionally</td>
</tr>
</tbody>
</table>

Assembler symbols

<Zd>    Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>    Is the size specifier, encoded in the "size" field. It can have the following values:
B    when size = 00
H    when size = 01
S    when size = 10
D    when size = 11
<imm>    Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.
<shift>    Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:
LSL #0    when sh = 0
LSL #8 when sh = 1

**Operation**

CheckSVEEnabled();

constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;

bits(VL) result = Replicate(imm<esize-1:0>, VL DIV esize);

Z\[d, VL\] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.96  DUP (indexed)

Unconditionally broadcast the indexed source vector element into each element of the destination vector. This instruction is unpredicated.

The immediate element index is in the range of 0 to 63 (bytes), 31 (halfwords), 15 (words), 7 (doublewords) or 3 (quadwords). Selecting an element beyond the accessible vector length causes the destination vector to be set to zero.

This instruction is used by the alias MOV (SIMD&FP scalar, unpredicated). See Alias conditions for details of when each alias is preferred.

Encoding

DUP <Zd>.<T>, <Zn>.<T>[<imm>]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
b_bits(7) imm = imm2:tsz;
integer esize;
integer index;
case tsz of
  when '00000' UNDEFINED;
  when '10000' esize = 128; index = UInt(imm<6:5>);
  when 'x1000' esize = 64;  index = UInt(imm<6:4>);
  when 'xx100' esize = 32;  index = UInt(imm<6:3>);
  when 'xxx10' esize = 16;  index = UInt(imm<6:2>);
  when 'xxxx1' esize = 8;   index = UInt(imm<6:1>);
integer n = UInt(Zn);
integer d = UInt(Zd);

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (SIMD&amp;FP scalar, unpredicated)</td>
<td>BitCount(imm2:tsz) == 1</td>
</tr>
<tr>
<td>MOV (SIMD&amp;FP scalar, unpredicated)</td>
<td>BitCount(imm2:tsz) &gt; 1</td>
</tr>
</tbody>
</table>

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  Is the size specifier, encoded in the "tsz" field. It can have the following values:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>when tsz = xxxx1</td>
</tr>
<tr>
<td>H</td>
<td>when tsz = xxx10</td>
</tr>
<tr>
<td>S</td>
<td>when tsz = xx100</td>
</tr>
<tr>
<td>D</td>
<td>when tsz = x1000</td>
</tr>
<tr>
<td>Q</td>
<td>when tsz = 10000</td>
</tr>
</tbody>
</table>

The encoding tsz = 00000 is reserved.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<imm> Is the immediate index, in the range 0 to one less than the number of elements in 512 bits, encoded in "imm2:tsz".

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result;
bits(esize) element;
if index >= elements then
    element = Zeros(esize);
else
    element = Elem[operand1, index, esize];
result = Replicate(element, VL DIV esize);
Z[d, VL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.97 DUP (scalar)

Unconditionally broadcast the general-purpose scalar source register into each element of the destination vector. This instruction is unpredicated.

This instruction is used by the alias MOV (scalar, unpredicated). The alias is always the preferred disassembly.

Encoding

DUP <Zd>.,<T>, <R><n|SP>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11
<R> Is a width specifier, encoded in the "size" field. It can have the following values:
W when size = 01
W when size = x0
X when size = 11

<n|SP> Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer PL = VL DIV 8;
bits(64) operand;
if n == 31 then
    operand = SP[];
else
    operand = X[n, 64];
bits(VL) result;
for e = 0 to elements-1
    Elem[result, e, esize] = operand<esize-1:0>;
$Z[d, \text{VL}] = \text{result}$

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.98   DUPM

Unconditionally broadcast the logical bitmask immediate into each element of the destination vector. This instruction is unpredicated. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits.

This instruction is used by the alias MOV. See Alias conditions for details of when each alias is preferred.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17</th>
<th>1</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1 1 0 0 0 0</td>
<td>imm13</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

DUPM <Zd>.<T>, #<const>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer d = UInt(Zd);
bits(esize) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, esize);

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>SVEMoveMaskPreferred(imm13)</td>
</tr>
</tbody>
</table>

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  Is the size specifier, encoded in the "imm13<12>:imm13<5:0>" field. It can have the following values:

- **S**  when imm13<12> = 0, imm13<5:0> = 0xxxxx
- **H**  when imm13<12> = 0, imm13<5:0> = 10xxxx
- **B**  when imm13<12> = 0, imm13<5:0> = 110xx
- **B**  when imm13<12> = 0, imm13<5:0> = 1110x
- **D**  when imm13<12> = 1, imm13<5:0> = xxxxxx

The following encodings are reserved:

- · imm13<12> = 0, imm13<5:0> = 111110.
- · imm13<12> = 0, imm13<5:0> = 111111.

<const>  Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) result = Replicate(imm, VL DIV esize);
Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.99  EON

Bitwise exclusive OR an inverted immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is a pseudo-instruction of the EOR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of EOR (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of EOR (immediate) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17</th>
<th></th>
<th></th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
<td>0 1 0 0 0 0</td>
<td>imm13</td>
<td>Zdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

EON <Zdn>,<T>, <Zdn>,<T>, #<const>

is equivalent to

EOR <Zdn>,<T>, <Zdn>,<T>, #(-<const> - 1)

and is never the preferred disassembly.

**Assembler symbols**

- `<Zdn>` is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` is the size specifier, encoded in the "imm13<12>:imm13<5:0>" field. It can have the following values:
  - `S` when imm13<12> = 0, imm13<5:0> = 0xxxxx
  - `H` when imm13<12> = 0, imm13<5:0> = 10xxxx
  - `B` when imm13<12> = 0, imm13<5:0> = 110xxx
  - `B` when imm13<12> = 0, imm13<5:0> = 1110xx
  - `B` when imm13<12> = 0, imm13<5:0> = 11110x
  - `D` when imm13<12> = 1, imm13<5:0> = xxxxxx

The following encodings are reserved:

- imm13<12> = 0, imm13<5:0> = 111110.
- imm13<12> = 0, imm13<5:0> = 111111.

- `<const>` is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

**Operation**

The description of EOR (immediate) gives the operational pseudocode for this instruction.
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.100  EOR (immediate)

Bitswise exclusive OR an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction **EON**. The pseudo-instruction is never the preferred disassembly.

![Encoding Table]

**Encoding**

EOR <Zdn>,<T>, <Zdn>,<T>, #<const>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer dn = UInt(Zdn);
bits(64) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, 64);

**Assembler symbols**

<Zdn>  
Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T>  
Is the size specifier, encoded in the "imm13<12>:imm13<5:0>" field. It can have the following values:
- S when imm13<12> = 0, imm13<5:0> = 0xxxxx
- H when imm13<12> = 0, imm13<5:0> = 10xxxx
- B when imm13<12> = 0, imm13<5:0> = 110xxx
- B when imm13<12> = 0, imm13<5:0> = 1110xx
- B when imm13<12> = 0, imm13<5:0> = 11110x
- D when imm13<12> = 1, imm13<5:0> = xxxxxx

The following encodings are reserved:
- imm13<12> = 0, imm13<5:0> = 111110.
- imm13<12> = 0, imm13<5:0> = 111111.

<const>  
Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 64;
bits(VL) operand = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(64) element1 = Elem[operand, e, 64];
  Elem[result, e, 64] = element1 EOR imm;
$Z[dn, VL] = \text{result};$

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.101   EOR (predicates)

Bitwise exclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is used by the alias NOT (predicate). See Alias conditions for details of when each alias is preferred.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 14 13</th>
<th>10 9 8</th>
<th>5 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1</td>
<td>0 0 0 0</td>
<td>Pm 0 1</td>
<td>Pg 1</td>
<td>Pn 0</td>
<td>Pd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding


Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
calendar integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT (predicate)</td>
<td>Pm == Pg</td>
</tr>
</tbody>
</table>

Assembler symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```c
CheckSVEEnabled();
calendar integer VL = CurrentVL;
calendar integer PL = VL DIV 8;
calendar integer elements = VL DIV esize;
broadcasts(PL) mask = P[g, PL];
broadcasts(PL) operand1 = P[n, PL];
broadcasts(PL) operand2 = P[m, PL];
broadcasts(PL) result;
for e = 0 to elements-1
    broadcast element1 = ElemP[operand1, e, esize];
    broadcast element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 XOR element2;
```
else
    ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.
C8.2.102  EOR (vectors, predicated)

Bitwise exclusive OR active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

EOR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B  when size = 00
H  when size = 01
S  when size = 10
D  when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = element1 EOR element2;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.103  EOR (vectors, unpredicated)

Bitwise exclusive OR all elements of the second source vector with corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 |  5 4  |  0 |
  0 0 0 0 0 1 0 0 | 0 1 0 1 | Zm  | 0 0 1 1 0 0 | Zn  | Zd  
```

**Encoding**

EOR <Zd>.D, <Zn>.D, <Zm>.D

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

**Assembler symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
Z[d, VL] = operand1 EOR operand2;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.104   EOR3

Bitwise exclusive OR the corresponding elements of all three source vectors, and destructively place the results in the corresponding elements of the destination and first source vector. This instruction is unpredicated.

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zk> Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
bite(VL) operand1 = Z[dn, VL];
bite(VL) operand2 = Z[m, VL];
bite(VL) operand3 = Z[k, VL];
Z[dn, VL] = operand1 EOR operand2 EOR operand3;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.105   EORBT

Interleaving exclusive OR between the even-numbered elements of the first source vector register and the odd-numbered elements of the second source vector register, placing the result in the even-numbered elements of the destination vector, leaving the odd-numbered elements unchanged. This instruction is unpredicated.

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |  
0 1 0 0 0 1 0 1 | size 0 | Zm 1 0 0 1 0 | 0 | Zn | Zd  
```

Encoding

EORBT <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 1;
```

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, 2*e + sel1, esize];
bits(esize) element2 = Elem[operand2, 2*e + sel2, esize];
    Elem[result, 2*e + sel1, esize] = element1 EOR element2;
Z[d, VL] = result;
```
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.106   EORS

Bitwise exclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the alias NOTS. See Alias conditions for details of when each alias is preferred.

 Encoding


 Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

 Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOTS</td>
<td>Pm == Pg</td>
</tr>
</tbody>
</table>

 Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

 Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP(operand1, e, esize);
    bit element2 = ElemP(operand2, e, esize);
if $\text{Elem}[\text{mask, e, esize}] = '1'$ then
  $\text{Elem}[\text{result, e, esize}] = \text{element1 EOR element2}$;
else
  $\text{Elem}[\text{result, e, esize}] = '0'$;

if setflags then
  $\text{PSTATE.<N,Z,C,V>} = \text{PredTest(mask, result, esize)}$;
  $\text{P[d, PL]} = \text{result}$;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.107  EORTB

Interleaving exclusive OR between the odd-numbered elements of the first source vector register and the even-numbered elements of the second source vector register, placing the result in the odd-numbered elements of the destination vector, leaving the even-numbered elements unchanged. This instruction is unpredicated.

Encoding

EORTB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 0;

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

B  when size = 00
H  when size = 01
S  when size = 10
D  when size = 11

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, 2*e + sel1, esize];
bits(esize) element2 = Elem[operand2, 2*e + sel2, esize];
Elem[result, 2*e + sel1, esize] = element1 EOR element2;
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.108   EORV

Bitwise exclusive OR horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register.Inactive elements in the source vector are treated as zero.

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & Pg & Zn & Vd
\end{array}
\]

**Encoding**

EORV \(<V><d>, <Pg>, <Zn>><T>

**Decode for this encoding**

if !HaveSVE() \& \& !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

**Assembler symbols**

\(<V>\quad\text{Is a width specifier, encoded in the "size" field. It can have the following values:}\n\begin{align*}
B & \quad\text{when size} = 00 \\
H & \quad\text{when size} = 01 \\
S & \quad\text{when size} = 10 \\
D & \quad\text{when size} = 11
\end{align*}\n\(<d>\quad\text{Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.}\n\(<Pg>\quad\text{Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.}\n\(<Zn>\quad\text{Is the name of the source scalable vector register, encoded in the "Zn" field.}\n\(<T>\quad\text{Is the size specifier, encoded in the "size" field. It can have the following values:}\n\begin{align*}
B & \quad\text{when size} = 00 \\
H & \quad\text{when size} = 01 \\
S & \quad\text{when size} = 10 \\
D & \quad\text{when size} = 11
\end{align*}\n\]**

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) result = Zeros(esize);
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        result = result EOR Elem[operand, e, esize];
\[ \forall [d, \text{esize}] = \text{result}; \]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.109   EXT

Copy the indexed byte up to the last byte of the first source vector to the bottom of the result vector, then fill the
remainder of the result starting from the first byte of the second source vector. The result is placed destructively in
the destination and first source vector, or constructively in the destination vector. This instruction is unpredicated.

An index that is greater than or equal to the vector length in bytes is treated as zero, resulting in the first source
vector being copied to the result unchanged.

The Destructive encoding of this instruction might be immediately preceded in program order by a MOVPRFX
instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior
of the MOVPRFX and this instruction is UNPREDICTABLE: The MOVPRFX instruction must be unpredicated.
The MOVPRFX instruction must specify the same destination register as this instruction. The destination register
must not refer to architectural register state referenced by any other source operand register of this instruction.

Constructive

Encoding

Decode for this encoding
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dst = UInt(Zd);
integer s1 = UInt(Zn);
integer s2 = (s1 + 1) MOD 32;
integer position = UInt(imm8h:imm8l);

Destructive

Encoding

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dst = UInt(Zdn);
integer s1 = dst;
integer s2 = UInt(Zm);
integer position = UInt(imm8h:imm8l);

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>  Is the name of the first scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

<Zn2>  Is the name of the second scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

<imm>  Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8h:imm8l" fields.

**Operation for all encodings**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[s1, VL];
bits(VL) operand2 = Z[s2, VL];
bits(VL) result;

if position >= elements then
    position = 0;
position = position << 3;
bits(VL*2) concat = operand2 : operand1;
result = concat<(position+VL)-1:position>;
Z[dst, VL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.110 FABD

Compute the absolute difference of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1 0 0 1 0 0</td>
<td>Pg</td>
<td>Zm</td>
<td>Zdn</td>
</tr>
</tbody>
</table>
```

**Encoding**

FABD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Zm>.<T>

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

**Assembler symbols**

- `<Zdn>`: Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>`: Is the size specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11
  - The encoding size = 00 is reserved.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>`: Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if Elem[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPAbs(FPSub(element1, element2, FPCR[ ]);)
    else
        Elem[result, e, esize] = element1;
Z[dn, VL] = result;
```

```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.111   FABS

Take the absolute value of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This clears the sign bit and cannot signal a floating-point exception. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>size 0 1 1 0 0 1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FABS <Zd>,<T>, <Pg>/M, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11

The encoding size = 00 is reserved.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPAbs(element);
  Z[d, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.112 FAC<cc>

Compare active absolute values of floating-point elements in the first source vector with corresponding absolute values of elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is used by the pseudo-instructions FACLE and FACLT. The pseudo-instruction is never the preferred disassembly.

**Greater than**

| [31 30 29 28][27 26 25 24][23 22 21 20] | 16|15 14 13 12| 10 9 | 5 4 3 0 |
|-------------------------------------------|--------------------------|--------------------------|--------------------------|
| 0 1 1 0 0 1 0 1 1 | size 0 | Zm 1 | 1 | Pg | Zn 1 | Pd |

**Encoding**

FACGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;

**Greater than or equal**

| [31 30 29 28][27 26 25 24][23 22 21 20] | 16|15 14 13 12| 10 9 | 5 4 3 0 |
|-------------------------------------------|--------------------------|--------------------------|--------------------------|
| 0 1 1 0 0 1 0 1 1 | size 0 | Zm 1 | 0 | Pg | Zn 1 | Pd |

**Encoding**

FACGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11

The encoding size = 00 is reserved.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(PL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    boolean res;
    case op of
      when Cmp_GE res = FPCompareGE(FPAbs(element1), FPAbs(element2), FPCR[]);
      when Cmp_GT res = FPCompareGT(FPAbs(element1), FPAbs(element2), FPCR[]);
      else
        ElemP[result, e, esize] = if res then '1' else '0';
    end
  else
    ElemP[result, e, esize] = '0';
end
P[d, PL] = result;

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is
dependent on the predicate register written by this instruction might be significantly delayed.
C8.2.113 FACLE

Compare active absolute values of floating-point elements in the first source vector being less than or equal to corresponding absolute values of elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is a pseudo-instruction of the FAC<cc> instruction. This means that:

- The encodings in this description are named to match the encodings of FAC<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FAC<cc> gives the operational pseudocode for this instruction.

```
01 11 00 0 1 0 1 Zm 1 1 0 Pg Zn 1 Pd
```

**Encoding**

FACLE <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

FACGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

and is never the preferred disassembly.

**Assembler symbols**

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <T> Is the size specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11
  - The encoding size = 00 is reserved.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

**Operation**

The description of FAC<cc> gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.
C8.2.114 FACLT

Compare active absolute values of floating-point elements in the first source vector being less than corresponding absolute values of elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is a pseudo-instruction of the FAC<cc> instruction. This means that:

- The encodings in this description are named to match the encodings of FAC<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FAC<cc> gives the operational pseudocode for this instruction.

Encoding

FACLT <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

FACGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

and is never the preferred disassembly.

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of FAC<cc> gives the operational pseudocode for this instruction.

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.
C8.2.115  FADD (immediate)

Add an immediate to each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

Encoding

FADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then FPPointFive('0', esize) else FPOne('0', esize);

Assembler symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
    H when size = 01
    S when size = 10
    D when size = 11
The encoding size = 00 is reserved.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<const> Is the floating-point immediate value, encoded in the "i1" field. It can have the following values:
    #0.5 when i1 = 0
    #1.0 when i1 = 1

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPAdd(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.116 FADD (vectors, predicated)

Add active floating-point elements of the second source vector to corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

FADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
g = UInt(Pg);
dn = UInt(Zdn);
m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11

The encoding size = 00 is reserved.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if Elem[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.117   FADD (vectors, unpredicated)

Add all floating-point elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>size</td>
<td>0</td>
<td>Zm</td>
<td>0 0 0 0 0 0</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FADD <Zd>,<T>, <Zn>,<T>, <Zm>,<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler symbols**

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

- **H**  when size = 01
- **S**  when size = 10
- **D**  when size = 11

The encoding size = 00 is reserved.

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);
Z[d, VL] = result;
C8.2.118 FADDA

Floating-point add a SIMD&FP scalar source and all active lanes of the vector source and place the result destructively in the SIMD&FP scalar source register. Vector elements are processed strictly in order from low to high, with the scalar source providing the initial value. Inactive elements in the source vector are ignored.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

Encoding

FADDA <V><dn>, <Pg>, <V><dn>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Vdn);
integer m = UInt(Zm);

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
  H when size = 01
  S when size = 10
  D when size = 11
The encoding size = 00 is reserved.
<dn> Is the number [0-31] of the source and destination SIMD&FP register, encoded in the "Vdn" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the source scalable vector register, encoded in the "Zm" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  H when size = 01
  S when size = 10
  D when size = 11
The encoding size = 00 is reserved.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = V[dn, esize];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(esize) result = operand1;

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand2, e, esize];
        result = FPAdd(result, element, FPCR[]);

V[dn, esize] = result;
C8.2.119   FADDP

Add pairs of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

Encoding
FADDP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>  Is the size specifier, encoded in the "size" field. It can have the following values:
    H  when size = 01
    S  when size = 10
    D  when size = 11
    The encoding size = 00 is reserved.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result = Z[dn, VL];
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        if IsEven(e) then
            element1 = Elem[operand1, e + 0, esize];
            element2 = Elem[operand1, e + 1, esize];
        else
            element1 = Elem[operand2, e - 1, esize];
            element2 = Elem[operand2, e + 0, esize];
        Elem[result, e, esize] = FPAdd(element1, element2, FPCR[)];
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.120   FADDV

Floating-point add horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as +0.0.

```
[31 30 29 28][27 26 25 24][23 22 21 20][19 18 17 16][15 14 13 12]  10  9  |  5  4  |  0 |
 0  1  1  0  0  1  0  1  |  0  0  0  0  |  0  0  1 |  Pg |  Zn |  Vd
```

Encoding

FADDV <V><d>, <Pg>, <Zn>.<T>

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler symbols

- **<V>** Is a width specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11
  The encoding size = 00 is reserved.

- **<d>** Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

- **<Zn>** Is the name of the source scalable vector register, encoded in the "Zn" field.

- **<T>** Is the size specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11
  The encoding size = 00 is reserved.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPZero('0', esize);
V[d, esize] = ReducePredicated(ReduceOp_FADD, operand, mask, identity);
```
C8.2.121 FCADD

Add the real and imaginary components of the active floating-point complex numbers from the first source vector to the complex numbers from the second source vector which have first been rotated by 90 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, equivalent to multiplying the complex numbers in the second source vector by \( \pm j \) beforehand. Destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 0</td>
<td>size 0 0 0 0 0</td>
<td>rot 1 0 0</td>
<td>Pg</td>
<td>Zm</td>
<td>Zdn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FCADD \(<Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>, <const>\)

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean sub_i = (rot == '0');
boolean sub_r = (rot == '1');

Assembler symbols

\(<Zdn>\) Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

\(<T>\) Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

\(<const>\) Is the const specifier, encoded in the "rot" field. It can have the following values:

- #90 when rot = 0
- #270 when rot = 1

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for p = 0 to pairs-1
    acc_r = Elem[operand1, 2 * p + 0, esize];
    acc_i = Elem[operand1, 2 * p + 1, esize];
    if Elem[mask, 2 * p + 0, esize] == '1' then
        elt2_i = Elem[operand2, 2 * p + 1, esize];
        if sub_i then elt2_i = FPNeg(elt2_i);
        acc_r = FPAdd(acc_r, elt2_i, FPCR[]);
    if Elem[mask, 2 * p + 1, esize] == '1' then
        elt2_r = Elem[operand2, 2 * p + 0, esize];
        if sub_r then elt2_r = FPNeg(elt2_r);
        acc_i = FPAdd(acc_i, elt2_r, FPCR[]);
    Elem[result, 2 * p + 0, esize] = acc_r;
    Elem[result, 2 * p + 1, esize] = acc_i;

Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.122 FCM<cc> (vectors)

Compare active floating-point elements in the first source vector with corresponding elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

<table>
<thead>
<tr>
<th>&lt;cc&gt;</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>equal</td>
</tr>
<tr>
<td>GE</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>GT</td>
<td>greater than</td>
</tr>
<tr>
<td>NE</td>
<td>not equal</td>
</tr>
<tr>
<td>UO</td>
<td>unordered</td>
</tr>
</tbody>
</table>

This instruction is used by the pseudo-instructions FCMLE (vectors) and FCMLT (vectors). The pseudo-instruction is never the preferred disassembly.

**Equal**

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 3 0 |
0 1 1 0 0 1 0 1 size 0 Zm 0 1 1 Pg Zn 0 1 Pd
```

**Encoding**

FCMEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
```

**Greater than**

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 3 0 |
0 1 1 0 0 1 0 1 size 0 Zm 0 1 0 Pg Zn 1 Pd
```

**Encoding**

FCMG <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;

Greater than or equal

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>size 0</td>
<td>Zm 0 1 0</td>
<td>Pg</td>
<td>Zn 0</td>
<td>Pd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FCMGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;

Not equal

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>size 0</td>
<td>Zm 0 1 1</td>
<td>Pg</td>
<td>Zn 1</td>
<td>Pd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FCMNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_NE;
Unordered

Encoding

Decoding for this encoding

Assembler symbols

Operation for all encodings
ElemP(result, e, esize) = if res then '1' else '0';
else
    ElemP(result, e, esize) = '0';

P[d, PL] = result;

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.
C8.2.123 FCM<cc> (zero)

Compare active floating-point elements in the source vector with zero, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

<table>
<thead>
<tr>
<th>&lt;cc&gt;</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>equal</td>
</tr>
<tr>
<td>GE</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>GT</td>
<td>greater than</td>
</tr>
<tr>
<td>LE</td>
<td>less than or equal</td>
</tr>
<tr>
<td>LT</td>
<td>less than</td>
</tr>
<tr>
<td>NE</td>
<td>not equal</td>
</tr>
<tr>
<td>UO</td>
<td>unordered</td>
</tr>
</tbody>
</table>

Equal

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12] 10 9 | 5 4 3 0 |
0 1 1 0 0 1 0 1 | 0 1 0 0 1 0 0 1 | Pg | Zn | 0 | Pd |
```

Encoding

FCMEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEComp op = Cmp_EQ;
```

Greater than

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12] 10 9 | 5 4 3 0 |
0 1 1 0 0 1 0 1 | 0 1 0 0 1 0 0 1 | Pg | Zn | 1 | Pd |
```

Encoding

FCMG <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
```
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;

Greater than or equal

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] 10  9  |  5  4  3  0  
0 1 1 0 0 1 0 1  size 0 1 0 0 0 0 0 0 1  Pg  Zn 0  Pd
```

Encoding

FOMGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 ** UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
```

Less than

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] 10  9  |  5  4  3  0  
0 1 1 0 0 1 0 1  size 0 1 0 0 0 1 0 0 1  Pg  Zn 0  Pd
```

Encoding

FOMLT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 ** UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
```

Less than or equal

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] 10  9  |  5  4  3  0  
0 1 1 0 0 1 0 1  size 0 1 0 0 0 1 0 0 1  Pg  Zn 1  Pd
```

Encoding

FOME <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0
Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEOp op = Cmp_LE;

Not equal

Encoding

FCMNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEOp op = Cmp_NE;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bitsPL mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(PL) result;

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        boolean res;
        case op of
            eq:
                res = true;
            lt:
                res = false;
            ne:
                res = false;
        endcase;
    endfor;
when Cmp_EQ res = FPCompareEQ(element, 0<esize-1:0>, FPCR[]);
when Cmp_GE res = FPCompareGE(element, 0<esize-1:0>, FPCR[]);
when Cmp_GT res = FPCompareGT(element, 0<esize-1:0>, FPCR[]);
when Cmp_NE res = FPCompareNE(element, 0<esize-1:0>, FPCR[]);
when Cmp_LT res = FPCompareGT(0<esize-1:0>, element, FPCR[]);
when Cmp_LE res = FPCompareGE(0<esize-1:0>, element, FPCR[]);

ElemP[result, e, esize] = if res then '1' else '0';
else
   ElemP[result, e, esize] = '0';

P[d, PL] = result;

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.
C8.2.124   FCMLA (indexed)

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the floating-point complex numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding the second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

Then destructively add the products to the corresponding components of the complex numbers in the addend and destination vector, without intermediate rounding.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

The complex numbers within the second source vector are specified using an immediate index which selects the same complex number position within each 128-bit vector segment. The index range is from 0 to one less than the number of complex numbers per 128-bit segment, encoded in 1 to 2 bits depending on the size of the complex number. This instruction is unpredicated.

**Half-precision**

![Half-precision encoding](image)

**Encoding**


**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean neg_i = (rot<1> == '1');
boolean neg_r = (rot<0> ! = rot<1>);

**Single-precision**

![Single-precision encoding](image)

**Encoding**

FCMLA <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>
Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean neg_i = (rot<1> == '1');
boolean neg_r = (rot<0> != rot<1>);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the half-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the single-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the half-precision variant: is the index of a Real and Imaginary pair, in the range 0 to 3, encoded in the "i2" field.
For the single-precision variant: is the index of a Real and Imaginary pair, in the range 0 to 1, encoded in the "i1" field.

<const> Is the const specifier, encoded in the "rot" field. It can have the following values:
#0 when rot = 00
#90 when rot = 01
#180 when rot = 10
#270 when rot = 11

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
constant integer pairspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for p = 0 to pairs-1
  segmentbase = p - (p MOD pairspersegment);
  s = segmentbase + index;
  addend_r = Elem[operand1, 2 * p + 0, esize];
  addend_i = Elem[operand1, 2 * p + 1, esize];
  elt1_a   = Elem[operand1, 2 * p + sel_a, esize];
  elt2_a   = Elem[operand2, 2 * s + sel_a, esize];
  elt2_b   = Elem[operand2, 2 * s + sel_b, esize];
  if neg_r then elt2_a = FPNeg(elt2_a);
  if neg_i then elt2_b = FPNeg(elt2_b);
  addend_r = FPMulAdd(addend_r, elt1_a, elt2_a, FPCR[]);
  addend_i = FPMulAdd(addend_i, elt1_a, elt2_b, FPCR[]);
  Elem[result, 2 * p + 0, esize] = addend_r;
  Elem[result, 2 * p + 1, esize] = addend_i;
Z[da, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.125   FCMLA (vectors)

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the floating-point complex numbers in the first source vector by the corresponding complex number in the second source vector rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

Then destructively add the products to the corresponding components of the complex numbers in the addend and destination vector, without intermediate rounding.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element. Inactive elements in the destination vector register remain unmodified.

Encoding

FCMLA <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>, <const>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean neg_i = (rot<1> == '1');
boolean neg_r = (rot<0> != rot<1>);

Assembler symbols

<Zda>     Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<T>       Is the size specifier, encoded in the "size" field. It can have the following values:
            H when size = 01
            S when size = 10
            D when size = 11
            The encoding size = 00 is reserved.
<Pg>      Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>      Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>      Is the name of the second source scalable vector register, encoded in the "Zm" field.
<const> Is the const specifier, encoded in the "rot" field. It can have the following values:

- #0 when rot = 00
- #90 when rot = 01
- #180 when rot = 10
- #270 when rot = 11

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for p = 0 to pairs-1
    addend_r = Elem[operand3, 2 * p + 0, esize];
    addend_i = Elem[operand3, 2 * p + 1, esize];
    if ELEM(mask, 2 * p + 0, esize) == '1' then
        bits(esize) elt1_a = Elem[operand1, 2 * p + sel_a, esize];
        bits(esize) elt2_a = Elem[operand2, 2 * p + sel_a, esize];
        if neg_r then elt2_a = FPNeg(elt2_a);
        addend_r = FPMulAdd(addend_r, elt1_a, elt2_a, FPCR[]);
    if ELEM(mask, 2 * p + 1, esize) == '1' then
        bits(esize) elt1_a = Elem[operand1, 2 * p + sel_a, esize];
        bits(esize) elt2_b = Elem[operand2, 2 * p + sel_b, esize];
        if neg_i then elt2_b = FPNeg(elt2_b);
        addend_i = FPMulAdd(addend_i, elt1_a, elt2_b, FPCR[]);
    Elem[result, 2 * p + 0, esize] = addend_r;
    Elem[result, 2 * p + 1, esize] = addend_i;

Z[da, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.126  FCMLE (vectors)

Compare active floating-point elements in the first source vector being less than or equal to corresponding elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is a pseudo-instruction of the FCM<cc> (vectors) instruction. This means that:

- The encodings in this description are named to match the encodings of FCM<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.

### Encoding

FCMLE <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

FCMGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

and is never the preferred disassembly.

### Assembler symbols

- `<Pd>` is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zn>` is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<T>` is the size specifier, encoded in the "size" field. It can have the following values:
  - `H` when size = 01
  - `S` when size = 10
  - `D` when size = 11
  - The encoding size = 00 is reserved.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

### Operation

The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.

### Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.
C8.2.127   FCMLT (vectors)

Compare active floating-point elements in the first source vector being less than corresponding elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is a pseudo-instruction of the FCM<cc> (vectors) instruction. This means that:

- The encodings in this description are named to match the encodings of FCM<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.

**Encoding**

FCMLT <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

FCMGT   <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

and is never the preferred disassembly.

**Assembler symbols**

- <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <T> Is the size specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11
  
  The encoding size = 00 is reserved.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

**Operation**

The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.
C8.2.128   FCPY

Copy a floating-point immediate into each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is used by the alias FMOV (immediate, predicated). The alias is always the preferred disassembly.

Encoding

FCPY <Zd>.<T>, <Pg>/M, #<const>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = $ << UInt(size);
ing = UInt(Pg);
d = UInt(Zd);
bits(esize) imm = VFPExpandImm(imm8, esize);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11

The encoding size = 00 is reserved.

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<const> Is a floating-point immediate value expressable as ±n\times16\times2^{r}, where n and r are integers such that 16 \leq n \leq 31 and -3 \leq r \leq 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
   if Elem[mask, e, esize] == '1' then
      Elem[result, e, esize] = imm;
   Z[d, VL] = result;
**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is **UNPREDICTABLE**:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.129 FCVT

Convert the size and precision of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Since the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

**Half-precision to single-precision**

```
[31 30 29 28][27 26 25 24][23 22 21 20][19 18 17 16][15 14 13 12] 10 9 | 5 4 | 0
0 1 1 0 0 1 0 1 | 1 0 0 0 1 0 0 1 1 0 1 | Pg | Zn | Zd
```

**Encoding**

FCVT <Zd>.S, <Pg>/M, <Zn>.H

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 32;

**Half-precision to double-precision**

```
[31 30 29 28][27 26 25 24][23 22 21 20][19 18 17 16][15 14 13 12] 10 9 | 5 4 | 0
0 1 1 0 0 1 0 1 | 1 1 0 0 1 0 0 1 0 1 0 1 | Pg | Zn | Zd
```

**Encoding**

FCVT <Zd>.D, <Pg>/M, <Zn>.H

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 64;

**Single-precision to half-precision**

```
[31 30 29 28][27 26 25 24][23 22 21 20][19 18 17 16][15 14 13 12] 10 9 | 5 4 | 0
0 1 1 0 0 1 0 1 | 1 0 0 0 1 0 0 1 0 0 1 0 1 | Pg | Zn | Zd
```

**Encoding**

FCVT <Zd>.S, <Pg>/M, <Zn>.H
Encoding

FCVT <Zd>.H, <Pg>/M, <Zn>.S

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 16;

Single-precision to double-precision

Encoding

FCVT <Zd>.D, <Pg>/M, <Zn>.S

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 64;

Double-precision to half-precision

Encoding

FCVT <Zd>.H, <Pg>/M, <Zn>.D

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 16;
Double-precision to single-precision

[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] [10 9] [5 4] [0]
0 1 1 0 1 0 1 1 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Pg Zn Zd

Encoding
FCVT <Zd>.S, <Pg>/M, <Zn>.D

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 32;

Assembler symbols
<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation for all encodings
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        bits(d_esize) res = FPConvertSVE(element<s_esize-1:0>, FPCR[], d_esize);
        Elem[result, e, esize] = ZeroExtend(res, esize);
Z[d, VL] = result;

Operational information
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.130 FCVTLT

Convert odd-numbered floating-point elements from the source vector to the next higher precision, and place the results in the active overlapping double-width elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

### Half-precision to single-precision

![Encoding](image)

**Encoding**

FCVTLT `<Zd>.S`, `<Pg>/M`, `<Zn>.H

**Decode for this encoding**

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

### Single-precision to double-precision

![Encoding](image)

**Encoding**

FCVTLT `<Zd>.D`, `<Pg>/M`, `<Zn>.S

**Decode for this encoding**

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

**Assembler symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation for all encodings**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
```
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize DIV 2) element = Elem[operand, 2*e + 1, esize DIV 2];
    Elem[result, e, esize] = FPConvertSVE(element, FPCR[], esize);
  
Z[d, VL] = result;
C8.2.131 FCVTNT

Convert active floating-point elements from the source vector to the next lower precision, and place the results in the odd-numbered half-width elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified.

**Single-precision to half-precision**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 |
| 0 1 1 0 0 1 0 0 1 0 0 0 1 0 0 1 0 1 |
```

**Encoding**

FCVTNT <Zd>.H, <Pg>/M, <Zn>.S

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;

constant integer esize = 32;

integer g = UInt(Pg);

integer n = UInt(Zn);

integer d = UInt(Zd);

**Double-precision to single-precision**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 |
| 0 1 1 0 0 1 0 0 1 0 0 0 1 0 0 1 0 1 |
```

**Encoding**

FCVTNT <Zd>.S, <Pg>/M, <Zn>.D

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;

constant integer esize = 64;

integer g = UInt(Pg);

integer n = UInt(Zn);

integer d = UInt(Zd);

**Assembler symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.

**Operation for all encodings**

CheckSVEEnabled();

constant integer VL = CurrentVL;

constant integer PL = VL DIV 8;

constant integer elements = VL DIV esize;

bits(PL) mask = Pg[PL];

bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, 2*e + 1, esize DIV 2] = FPConvertSVE(element, FPCR[], esize DIV 2);
  
Z[d, VL] = result;
C8.2.132 FCVTX

Convert active double-precision floating-point elements from the source vector to single-precision, rounding to Odd, and place the results in the even-numbered 32-bit elements of the destination vector, while setting the odd-numbered elements to zero. Inactive elements in the destination vector register remain unmodified.

Rounding to Odd (aka Von Neumann rounding) permits a two-step conversion from double-precision to half-precision without incurring intermediate rounding errors.

Encoding

FCVTX <Zd>.S, <Pg>/M, <Zn>.D

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 32;

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    bits(d_esize) res = FPCConvertSVE(element<s_esize-1:0>, FPCR[], FPRounding_ODD, d_esize);
    Elem[result, e, esize] = ZeroExtend(res, esize);
Z[d, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.133 FCVTXNT

Convert active double-precision floating-point elements from the source vector to single-precision, rounding to Odd, and place the results in the odd-numbered 32-bit elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified.

Rounding to Odd (aka Von Neumann rounding) permits a two-step conversion from double-precision to half-precision without incurring intermediate rounding errors.

Encoding

```
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
```

**<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.

**<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

**<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
current integer VL = CurrentVL;
current integer PL = VL DIV 8;
current integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, 2*e + 1, esize DIV 2] = FPConvertSVE(element, FPCR[], FPRounding_ODD, esize DIV 2);
Z[d, VL] = result;
```
C8.2.134  FCVTZS

Convert to the signed integer nearer to zero from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are sign-extended to fill each destination element.

**Half-precision to 16-bit**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 ]
0 1 1 0 0 1 0 1 | 0 1 0 1 1 0 1 0 1 | Pg | Zn | Zd

int_U
```

**Encoding**

FCVTZ <Zd>.H, <Pg>/M, <Zn>.H

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

**Half-precision to 32-bit**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 ]
0 1 1 0 0 1 0 1 | 0 1 0 1 1 0 1 0 1 | Pg | Zn | Zd

int_U
```

**Encoding**

FCVTZ <Zd>.S, <Pg>/M, <Zn>.H

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```
Half-precision to 64-bit

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 |
|0 1 1 0 0 1 0 1|0 1 0 1|1 1 0 1 0 1|Pg|Zn|Zd|
```

**Int_U**

**Encoding**

FCVTZS <Zd>.D, <Pg>/M, <Zn>.H

**Decode for this encoding**

```java
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

Single-precision to 32-bit

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 |
|0 1 1 0 0 1 0 1|0 1 0 1|1 1 0 1 0 1|Pg|Zn|Zd|
```

**Int_U**

**Encoding**

FCVTZS <Zd>.S, <Pg>/M, <Zn>.S

**Decode for this encoding**

```java
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

Single-precision to 64-bit

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 |
|0 1 1 0 0 1 0 1|0 1 0 1|1 1 0 1 0 1|Pg|Zn|Zd|
```

**Int_U**

**Encoding**

FCVTZS <Zd>.D, <Pg>/M, <Zn>.S
Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;

Double-precision to 32-bit

Encoding
FCVTZS <Zd>.S, <Pg>/M, <Zn>.D

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;

Double-precision to 64-bit

Encoding
FCVTZS <Zd>.D, <Pg>/M, <Zn>.D

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    bits(d_esize) res = FPToFixed(element<s_esize-1:0>, 0, unsigned, FPCR[], rounding, d_esize);
    Elem[result, e, esize] = Extend(res, esize, unsigned);

Z[d, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.135   **FCVTZU**

Convert to the unsigned integer nearer to zero from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

### Half-precision to 16-bit

```plaintext
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 ]
0 1 1 0 0 1 0 1 0 1 0 1 1 0 1 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Pg  Zn  Zd
```

**int_U**

### Encoding

**FCVTZU <Zd>.H, <Pg>/M, <Zn>.H**

**Decode for this encoding**

```plaintext
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

### Half-precision to 32-bit

```plaintext
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 ]
0 1 1 0 0 1 0 1 0 1 0 1 1 0 1 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Pg  Zn  Zd
```

**int_U**

### Encoding

**FCVTZU <Zd>.S, <Pg>/M, <Zn>.H**

**Decode for this encoding**

```plaintext
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```
### Half-precision to 64-bit

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[19 18 17 16]</th>
<th>[15 14 13 12]</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>0 1 0 1 1 1 1 1 1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FCVTZU <Zd>.D, <Pg>/M, <Zn>.H

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

### Single-precision to 32-bit

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[19 18 17 16]</th>
<th>[15 14 13 12]</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>0 1 0 1 1 1 1 1 0 1 1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FCVTZU <Zd>.S, <Pg>/M, <Zn>.S

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

### Single-precision to 64-bit

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[19 18 17 16]</th>
<th>[15 14 13 12]</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>1 1 0 1 1 1 1 1 0 1 1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FCVTZU <Zd>.D, <Pg>/M, <Zn>.S
Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UINT(Pg);
integer n = UINT(Zn);
integer d = UINT(Zd);
constant integer s_esize = 32;
constant integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;

Double-precision to 32-bit

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1 1 1 0 1 0 0 1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FCVTZU <Zd>.S, <Pg>/M, <Zn>.D

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UINT(Pg);
integer n = UINT(Zn);
integer d = UINT(Zd);
constant integer s_esize = 64;
constant integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;

Double-precision to 64-bit

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1 1 1 0 1 1 1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FCVTZU <Zd>.D, <Pg>/M, <Zn>.D

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UINT(Pg);
integer n = UINT(Zn);
integer d = UINT(Zd);
constant integer s_esize = 64;
constant integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand  = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    bits(d_esize) res = FPToFixed(element<s_esize-1:0>, 0, unsigned, FPCR[], rounding, d_esize);
    Elem[result, e, esize] = Extend(res, esize, unsigned);
Z[d, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.136   FDIV

Divide active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

FDIV <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
ing = UInt(Pg);
dn = UInt(Zdn);
m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if Elem[mask, e, esize] == '1' then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPDIV(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.137   FDIVR

Reversed divide active floating-point elements of the second source vector by corresponding floating-point
elements of the first source vector and destructively place the quotient in the corresponding elements of the first
source vector. Inactive elements in the destination vector register remain unmodified.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10  9</th>
<th>5  4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0</td>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
<td>0 0 0 0</td>
<td>Pg</td>
<td>Zm</td>
<td>Zdn</td>
</tr>
</tbody>
</table>
```

Encoding

FDIVR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler symbols

- `<Zdn>` is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` is the size specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11

The encoding size = 00 is reserved.

- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if Elem[mask, e, esize] == '1' then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPDiv(element2, element1, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn, VL] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.138 FDUP

Unconditionally broadcast the floating-point immediate into each element of the destination vector. This instruction is unpredicated.

This instruction is used by the alias FMOV (immediate, unpredicated). The alias is always the preferred disassembly.

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12] | 5 4 | 0 |
0 0 1 0 0 1 0 1 | size | 1 1 1 | 0 0 1 1 1 0 | imm8 | Zd
```

**Encoding**

FDUP <Zd>,<T>, #<const>

**Decode for this encoding**

```java
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = $ << UInt(size);
integer d = UInt(Zd);
bits(esize) imm = VFPExpandImm(imm8, esize);
```

**Assemble symbols**

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>`: Is the size specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11
  - The encoding size = '00' is reserved.
- `<const>`: Is a floating-point immediate value expressable as ±n×16×2^r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

**Operation**

```java
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) result;
for e = 0 to elements-1
  Elem[result, e, esize] = imm;
Z[d, VL] = result;
```
C8.2.139 FEXPA

The FEXPA instruction accelerates the polynomial series calculation of the EXP(X) function.

The double-precision variant copies the low 52 bits of an entry from a hard-wired table of 64-bit coefficients, indexed by the low 6 bits of each element of the source vector, and prepends to that the next 11 bits of the source element (src<16:6>), setting the sign bit to zero.

The single-precision variant copies the low 23 bits of an entry from hard-wired table of 32-bit coefficients, indexed by the low 6 bits of each element of the source vector, and prepends to that the next 8 bits of the source element (src<13:6>), setting the sign bit to zero.

The half-precision variant copies the low 10 bits of an entry from hard-wired table of 16-bit coefficients, indexed by the low 5 bits of each element of the source vector, and prepends to that the next 5 bits of the source element (src<9:5>), setting the sign bit to zero.

A coefficient table entry with index m holds the floating-point value $2^{(m/64)}$, or for the half-precision variant $2^{(m/32)}$. This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] [11 10 9]  5  4  0
0 0 0 0 0 1 0 0 | 1 0 0 0 0 0 1 0 1 1 1 0 | Zn   Zd
```

Encoding

FEXPA <Zd>.<T>, <Zn>.<T>

Decode for this encoding

```plaintext
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = $ << UInt(size);
n = UInt(Zn);
d = UInt(Zd);
```

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size == 01
- S when size == 10
- D when size == 11

The encoding size == 00 is reserved.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```plaintext
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand = Z[n, VL];
```
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPExpA(element);

Z[d, VL] = result;
C8.2.140  FLOGB

This instruction returns the signed integer base 2 logarithm of each floating-point input element |X| after normalization.

This is the unbiased exponent of X used in the representation of the floating-point value, such that, for positive X, X = significand × 2^{exponent}.

The integer results are placed in elements of the destination vector which have the same width (ESIZE) as the floating-point input elements:

- If X is normal, the result is the base 2 logarithm of X.
- If X is subnormal, the result corresponds to the normalized representation.
- If X is infinite, the result is 2^{(esize-1)}-1.
- If X is ±0.0 or NaN, the result is -2^{(esize-1)}.

Inactive elements in the destination vector register remain unmodified.

Encoding

FLOGB <Zd>.<T>, <Pg>/M, <Zn>.<T>

Decoding for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
   H when size = 01
   S when size = 10
   D when size = 11
The encoding size = 00 is reserved.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1'
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPLogB(element, FPCR[]);
Z[d, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.141   FMAD

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third (addend) vector without intermediate rounding. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

Encoding

FMAD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
    H   when size = 01
    S   when size = 10
    D   when size = 11
The encoding size = 00 is reserved.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];
        result[e] = element1 + element2 + element3;
        if op1_neg then result[e] = ~result[e] + 1;
        if op3_neg then result[e] = ~result[e] + 1;

bits(esize) element3 = Elem[operand3, e, esize];
if op1_neg then element1 = FPNeg(element1);
if op3_neg then element3 = FPNeg(element3);
Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
else
   Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.142   FMAX (immediate)

Determine the maximum of an immediate and each active floating-point element of the source vector, and
destructively place the results in the corresponding elements of the source vector. The immediate may take the value
+0.0 or +1.0 only. If the element value is NaN then the result is NaN. Inactive elements in the destination vector
register remain unmodified.

Encoding

FMAX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

Decode for this encoding

if !HaveSVE() &amp; !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 &lt;&lt; UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros(esize) else FPOne('0', esize);

Assembler symbols

&lt;Zdn&gt;  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
&lt;T&gt;    Is the size specifier, encoded in the "size" field. It can have the following values:
        H when size = 01
        S when size = 10
        D when size = 11
        The encoding size = 00 is reserved.
&lt;Pg&gt;    Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
&lt;const&gt; Is the floating-point immediate value, encoded in the "i1" field. It can have the following values:
        #0.0 when i1 = 0
        #1.0 when i1 = 1

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = FMPMax(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;
\[ Z[dn, VL] = \text{result}; \]

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMAX (vectors)

Determine the maximum of active floating-point elements of the second source vector and corresponding
floating-point elements of the first source vector and destructively place the results in the corresponding elements
of the first source vector. If either element value is NaN then the result is NaN. Inactive elements in the destination
vector register remain unmodified.

Encoding

FMAX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if Elem[mask, e, esize] == '1' then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FMax(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.144  FMAXNM (Immediate)

Determine the maximum number value of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only. If the element value is a quiet NaN, then the result is the immediate. Inactive elements in the destination vector register remain unmodified.

Encoding

FMAXNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros(esize) else FPOne('0', esize);

Assembler symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  H when size = 01
  S when size = 10
  D when size = 11
The encoding size = 00 is reserved.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<const> Is the floating-point immediate value, encoded in the "i1" field. It can have the following values:
  #0.0 when i1 = 0
  #1.0 when i1 = 1

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPMaxNum(element1, imm, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.145  FMAXNM (vectors)

Determine the maximum number value of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If one element value is numeric and the other is a quiet NaN, then the result is the numeric value. Inactive elements in the destination vector register remain unmodified.

Encoding

FMAXNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL); bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if Elem[mask, e, esize] == '1' then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FMaxNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
$Z[dn, VL] = \text{result;}$

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.146  FMAXNMP

Compute the maximum value of each pair of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector. NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|  10  9  |  5  4  |  0  ]
 0 1 1 0 0 1 0 0 | 0 1 1 0 0 1 0 0 | Pg | Zm | Zdn
```

**Encoding**

FMAXNMP <Zdn>.<T>, <Pg>/<M>, <Zdn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

**Assembler symbols**

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>   Is the size specifier, encoded in the "size" field. It can have the following values:
H  when size = 01
S  when size = 10
D  when size = 11
The encoding size = 00 is reserved.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result = Z[dn, VL];
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    if IsEven(e) then
      element1 = Elem[operand1, e + 0, esize];
      element2 = Elem[operand1, e + 1, esize];
    else
      element1 = Elem[operand2, e - 1, esize];
  ```
element2 = \text{Elem}[\text{operand2}, e + 0, \text{esize}];
\text{Elem}[\text{result}, e, \text{esize}] = \text{FPMaxNum}(\text{element1}, \text{element2}, \text{FPCR[]});

Z[dn, VL] = \text{result};

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.147  FMAXNMV

Floating-point maximum number horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the default NaN.

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10  9 |  5  4 |  0 |
| 0  1  1  0 | 0  1  0  1 | size 0 0 1 0 0 0 1 | Pg | Zn | Vd |
```

**Encoding**

FMAXNMV <V><d>, <Pg>, <Zn><T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

**Assembler symbols**

<V> Is a width specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPDefaultNaN(esize);
V[d, esize] = ReducePredicated(ReduceOp_FMAXNUM, operand, mask, identity);
C8.2.148  FMAXP

Compute the maximum value of each pair of adjacent floating-point elements within each source vector, and
interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first
source vector.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 0</td>
<td>size 0 0 1 1 0 1 0 0</td>
<td>Pg</td>
<td>Zm</td>
<td>Zdn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

FMAXP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE2() & & !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

**Assembler symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11
  The encoding size = 00 is reserved.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result = Z[dn, VL];
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    if IsEven(e) then
      element1 = Elem[operand1, e + 0, esize];
      element2 = Elem[operand1, e + 1, esize];
    else
      element1 = Elem[operand2, e - 1, esize];
      element2 = Elem[operand2, e + 0, esize];
```
\[
\text{Elem[}\text{result, e, esize}] = \text{FPMax(element1, element2, FPCR[]);} \\
Z[\text{dn, VL}] = \text{result};
\]

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.149  FMAXV

Floating-point maximum horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as -Infinity.

Encoding

FMAXV <V><d>, <Pg>, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler symbols

<V>  Is a width specifier, encoded in the "size" field. It can have the following values:
    H    when size = 01
    S    when size = 10
    D    when size = 11

The encoding size = 00 is reserved.

<d>  Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:
    H    when size = 01
    S    when size = 10
    D    when size = 11

The encoding size = 00 is reserved.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = Pg, PL;
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPInfinity('1', esize);
V[d, esize] = ReducePredicated(ReduceOp_FMAX, operand, mask, identity);
C8.2.150   FMIN (immediate)

Determine the minimum of an immediate and each active floating-point element of the source vector, and
destructively place the results in the corresponding elements of the source vector. The immediate may take the value
+0.0 or +1.0 only. If the element value is NaN then the result is NaN. Inactive elements in the destination vector
register remain unmodified.

Encoding

FMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros(esize) else FPOne('0', esize);

Assembler symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11

The encoding size = 00 is reserved.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in the "i1" field. It can have the following values:

#0.0 when i1 = 0
#1.0 when i1 = 1

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Pg, PL; bits(PL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPMin(element1, imm, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.151  FMIN (vectors)

Determine the minimum of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If either element value is NaN then the result is NaN. Inactive elements in the destination vector register remain unmodified.

Encoding

FMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>   Is the size specifier, encoded in the "size" field. It can have the following values:
   H    when size = 01
   S    when size = 10
   D    when size = 11
   The encoding size = 00 is reserved.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
   bits(esize) element1 = Elem[operand1, e, esize];
   if Elem[mask, e, esize] == '1' then
      bits(esize) element2 = Elem[operand2, e, esize];
      Elem[result, e, esize] = FMIn(element1, element2, FPCR[{}]);
   else
      Elem[result, e, esize] = element1;
Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.152  FMINNM (immediate)

Determine the minimum number value of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only. If the element value is a quiet NaN, then the result is the immediate. Inactive elements in the destination vector register remain unmodified.

Encoding

FMINNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros(esize) else FPOne('0', esize);

Assembler symbols

<Zdn>  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<T>  Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<const>  Is the floating-point immediate value, encoded in the "i1" field. It can have the following values:
#0.0 when i1 = 0
#1.0 when i1 = 1

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPMinNum(element1, imm, FPCR[0]);
  else
    Elem[result, e, esize] = element1;

01100101
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 14 13 12 10 9 8 7 6 5 4 0]
0 1 1 0 0 1 0 1 0 1 1 0 1 0 0 i1 Zdn
\[ Z[dn, VL] = \text{result}; \]

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.153  **FMINNM (vectors)**

Determine the minimum number value of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If one element value is numeric and the other is a quiet NaN, then the result is the numeric value. Inactive elements in the destination vector register remain unmodified.

| 01100101 | 0x0 | size | 00 | 0101 | 100 | Pg | Zm | Zdn |

**Encoding**

FMINNM \(<Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>\)

**Decode for this encoding**

if \(!\text{HaveSVE}() \&\& \text{HaveSME}()\) then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

**Assembler symbols**

<Zdn>  
Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>  
Is the size specifier, encoded in the "size" field. It can have the following values:

- \(H\) when size = 01
- \(S\) when size = 10
- \(D\) when size = 11

The encoding size = 00 is reserved.

<Pg>  
Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>  
Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if Elem[mask, e, esize] == '1' then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FMinNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
\[ Z[dn, VL] = \text{result}; \]

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is **UNPREDICTABLE**:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.154  FMINNMP

Compute the minimum value of each pair of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector. NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value.

[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 ]
0 1 1 0 0 1 1 0 0 1 0 1 1 0 0 | Pg | Zm | Zdn

Encoding

FMINNMP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>  Is the size specifier, encoded in the "size" field. It can have the following values:
    H  when size = 01
    S  when size = 10
    D  when size = 11
    The encoding size = 00 is reserved.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result = Z[dn, VL];
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        if IsEven(e) then
            element1 = Elem[operand1, e + 0, esize];
            element2 = Elem[operand1, e + 1, esize];
        else
            element1 = Elem[operand2, e - 1, esize];

element2 = Elem[operand2, e + 0, esize];
Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Floating-point minimum number horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the default NaN.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>size 0 0 1 0 1 0 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Vd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FMINNMV <V><d>, <Pg>, <Zn>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

**Assemble symbols**

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPDefaultNaN(esize);
V[d, esize] = ReducePredicated(ReduceOp_FMINNUM, operand, mask, identity);
C8.2.156   FMINP

Compute the minimum value of each pair of adjacent floating-point elements within each source vector, and
interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first
source vector.

Encoding
FMINP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>   Is the size specifier, encoded in the "size" field. It can have the following values:
      H when size = 01
      S when size = 10
      D when size = 11
      The encoding size = 00 is reserved.

<Pg>   Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>   Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result = Z[dn, VL];
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    if IsEven(e) then
      element1 = Elem[operand1, e + 0, esize];
      element2 = Elem[operand1, e + 1, esize];
    else
      element1 = Elem[operand2, e - 1, esize];
      element2 = Elem[operand2, e + 0, esize];
  else
    element1 = element2 = Zeros(esize);
\[ \text{Elem}[\text{result}, e, \text{esize}] = \text{FPMin}(\text{element1}, \text{element2}, \text{FPCR[]}); \]

\[ Z[dn, VL] = \text{result}; \]

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.157   FMINV

Floating-point minimum horizontally over all lanes of a vector using a recursive pairwise reduction, and place the
result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as +Infinity.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|  10   9   |  5   4   |  0   |
  0  1  1  0  0  1  0  1|size  0  0  1  1|  1  0  0  1|Pg   Zn   Vd]
```

Encoding

FMINV <V><d>, <Pg>, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
  H when size = 01
  S when size = 10
  D when size = 11
The encoding size = 00 is reserved.
<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  H when size = 01
  S when size = 10
  D when size = 11
The encoding size = 00 is reserved.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPInfinity('0', esize);
V[d, esize] = ReducePredicated(ReduceOp_FMIN, operand, mask, identity);
C8.2.158  FMLA (indexed)

Multiply all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively added without intermediate rounding to the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

**Half-precision**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 0</td>
<td>1</td>
<td>i3l</td>
<td>Zm</td>
<td>0 0 0 0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Encoding**


**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;

**Single-precision**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 0</td>
<td>1</td>
<td>i2</td>
<td>Zm</td>
<td>0 0 0 0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Encoding**

FMLA <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;
Double-precision

```
[31 30 29 28 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 | 8 7 6 5 | 4 3 2 1 | 0]
0 1 1 0 0 1 0 0 | 1 1 1 | op
```

Encoding


Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;
```

Assembler symbols

- `<Zda>`
  - Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- `<Zn>`
  - Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>`
  - For the half-precision and single-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
  - For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- `<imm>`
  - For the half-precision variant: is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
  - For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the "i2" field.
  - For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the "i1" field.

Operation for all encodings

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
  integer segmentbase = e - (e MOD eltspersegment);
  integer s = segmentbase + index;
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, s, esize];
  bits(esize) element3 = Elem[result, e, esize];
  if op1_neg then element1 = FPNeg(element1);
  if op3_neg then element3 = FPNeg(element3);
  Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
```
Z[da, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.159  **FMLA (vectors)**

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third source (addend) vector without intermediate rounding. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

**Encoding**

FMLA <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;

**Assembler symbols**

<Zda>  Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>size</td>
<td>Zm</td>
<td>0</td>
<td>0</td>
<td>Zn</td>
<td>Zda</td>
<td></td>
</tr>
</tbody>
</table>

H  when size = 01
S  when size = 10
D  when size = 11
The encoding size = 00 is reserved.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element1 = Elem(operand1, e, esize);
    bits(esize) element2 = Elem(operand2, e, esize);
bits(esize) element3 = Elem[operand3, e, esize];

if op1_neg then element1 = FPNeg(element1);
if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
else
    Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.160 FMLALB (indexed)

This half-precision floating-point multiply-add long instruction widens the even-numbered half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the first source vector. This instruction is unpredicated.

Encoding

Decode for this encoding
if !HaveSVE2() && !HaveSME() then UNDEFINED;
    constant integer esize = 32;
    integer n = UInt(Zn);
    integer m = UInt(Zm);
    integer da = UInt(Zda);
    integer index = UInt(i3h:i3l);
    boolean op1_neg = FALSE;

Assembler symbols
<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

<imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation
CheckSVEEnabled();
    constant integer VL = CurrentVL;
    constant integer PL = VL DIV 8;
    constant integer elements = VL DIV esize;
    integer eltspersegment = 128 DIV esize;
    bits(VL) operand1 = Z[n, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = Z[da, VL];
    bits(VL) result;
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = 2 * segmentbase + index;
        bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 0, esize DIV 2];
        bits(esize DIV 2) element2 = Elem[operand2, s, esize DIV 2];
        bits(esize) element3 = Elem[operand3, e, esize];
        if op1_neg then element1 = FPNeg(element1);
        Elem[result, e, esize] = FMulAddH(element3, element1, element2, FPCR[]);
    Z[da, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.161  FMLALB (vectors)

This half-precision floating-point multiply-add long instruction widens the even-numbered half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the source vectors. This instruction is unpredicated.

Encoding

Decode for this encoding
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;

Assembler symbols
<Zda>  Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 0, esize DIV 2];
  bits(esize DIV 2) element2 = Elem[operand2, 2 * e + 0, esize DIV 2];
  bits(esize) element3 = Elem[operand3, e, esize];
  if op1_neg then element1 = FPNeg(element1);
  Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);
Z[da, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.162  FMLALT (indexed)

This half-precision floating-point multiply-add long instruction widens the odd-numbered half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the first source vector. This instruction is unpredicated.

**Encoding**


**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean op1_neg = FALSE;

**Assembler symbols**

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

<imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
  integer segmentbase = e - (e MOD eltspersegment);
  integer s = 2 * segmentbase + index;
  bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 1, esize DIV 2];
  bits(esize DIV 2) element2 = Elem[operand2, s, esize DIV 2];
  bits(esize) element3 = Elem[operand3, e, esize];
  if op1_neg then element1 = FPNeg(element1);
  Elem[result, e, esize] = FMulAddH(element3, element1, element2, FPCR[]);
Z[da, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.163 FMLALT (vectors)

This half-precision floating-point multiply-add long instruction widens the odd-numbered half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the source vectors. This instruction is unpredicated.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 | 15 14 13 12 11 10 9  5  4  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 |
|----------|-----------------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
|  Zm      |     1           |   0      |   0      |   0      |   1      |   0      |   1      |   0      |   1      |   0      |   1      |   0      |   1      |   0      |   1      |   0      |   1      |   0      |   1      |   0      |   1      |
| o2       | op  |   T   |----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
```

**Encoding**


**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;

**Assembler symbols**

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements - 1
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 1, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2 * e + 1, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);
Z[da, VL] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.164 FMLS (indexed)

Multiply all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively subtracted without intermediate rounding from the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

**Half-precision**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0]
0 1 1 0 0 1 0 0 | 0 1 | i3l | Zm 0 0 0 0 | 0 1 | Zn | Zda
```

Encoding


Decode for this encoding

```c
if (!HaveSVE() && !HaveSME()) then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
```

**Single-precision**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0]
0 1 1 0 0 1 0 0 | 0 1 | i2 | Zm 0 0 0 0 | 0 1 | Zn | Zda
```

Encoding

FMLS <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

Decode for this encoding

```c
if (!HaveSVE() && !HaveSME()) then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
```
Double-precision

Encoding


Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
c constant integer esize = 64;
c integer index = UInt(i1);
c integer n = UInt(Zn);
c integer m =UInt(Zm);
c integer da = UInt(Zda);
c boolean op1_neg = TRUE;
c boolean op3_neg = FALSE;
```

Assembler symbols

- `<Zda>` is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<Zn>` is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>` is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

- `<imm>` is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the “i2” field.

For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the “i1” field.

Operation for all encodings

```
CheckSVEEnabled();
c constant integer VL = CurrentVL;
c constant integer PL = VL DIV 8;
c constant integer elements = VL DIV esize;
c constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
better(VL) operand2 = Z[m, VL];
better(VL) result = Z[da, VL];
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, s, esize];
    bits(esize) element3 = Elem[result, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
```
Z[da, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.165 FMLS (vectors)

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third source (addend) vector without intermediate rounding. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 1 0 0 1</td>
<td>size 1</td>
<td>Zm 0 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zda</td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

FMLS <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
```

**Assembler symbols**

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
```
bits(esize) element3 = Elem[operand3, e, esize];
if op1_neg then element1 = FPNeg(element1);
if op3_neg then element3 = FPNeg(element3);
Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
else
  Elem[result, e, esize] = Elem[operand3, e, esize];
Z[da, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.166 FMULSLB (indexed)

This half-precision floating-point multiply-subtract long instruction widens the even-numbered half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the first source vector. This instruction is unpredicated.

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean op1_neg = TRUE;

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
<iimm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
  integer segmentbase = e - (e MOD eltspersegment);
  integer s = 2 * segmentbase + index;
  bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 0, esize DIV 2];
  bits(esize DIV 2) element2 = Elem[operand2, s, esize DIV 2];
  bits(esize) element3 = Elem[operand3, e, esize];
  if op1_neg then element1 = FPNeg(element1);
  Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);
Z[da, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.167 FMLSLB (vectors)

This half-precision floating-point multiply-subtract long instruction widens the even-numbered half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the source vectors. This instruction is unpredicated.

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 0, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2 * e + 0, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[{}]);
Z[da, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.168   FMLSLT (indexed)

This half-precision floating-point multiply-subtract long instruction widens the odd-numbered half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the first source vector. This instruction is unpredicated.

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean op1_neg = TRUE;

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

<iimm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 1, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, s, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);
Z[da, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.169 FMLSLT (vectors)

This half-precision floating-point multiply-subtract long instruction widens the odd-numbered half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the source vectors. This instruction is unpredicated.

Encoding

FMLSLT \(<Zda>.S, <Zn>.H, <Zm>.H\)

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 1, esize DIV 2];
  bits(esize DIV 2) element2 = Elem[operand2, 2 * e + 1, esize DIV 2];
  bits(esize) element3 = Elem[operand3, e, esize];
  if op1_neg then element1 = FPNeg(element1);
  Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);
Z[da, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.170  FMMMA

The floating-point matrix multiply-accumulate instruction supports single-precision and double-precision data types in a 2×2 matrix contained in segments of 128 or 256 bits, respectively. It multiplies the 2×2 matrix in each segment of the first source vector by the 2×2 matrix in the corresponding segment of the second source vector. The resulting 2×2 matrix product is then destructively added to the matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing a 2-way dot product per destination element. This instruction is unpredicated. The single-precision variant is vector length agnostic. The double-precision variant requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64ZFR0_EL1.F32MM indicates whether the single-precision variant is implemented.

ID_AA64ZFR0_EL1.F64MM indicates whether the double-precision variant is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

(FEAT_F32MM)

Encoding

FMMMA <Zda>.S, <Zn>.S, <Zm>.S

Decode for this encoding

if !HaveSVE() || !HaveSVEFP32MatMulExt() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

64-bit element

(FEAT_F64MM)

Encoding

FMMMA <Zda>.D, <Zn>.D, <Zm>.D

Decode for this encoding

if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < esize * 4 then UNDEFINED;
constant integer segments = VL DIV (4 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result = Zeros(VL);
bits(4*esize) op1, op2;
bits(4*esize) res, addend;
for s = 0 to segments-1
  op1 = Elem[operand1, s, 4*esize];
  op2 = Elem[operand2, s, 4*esize];
  addend = Elem[operand3, s, 4*esize];
  res = FPMatMulAdd(addend, op1, op2, esize, FPCR[]);
  Elem[result, s, 4*esize] = res;
Z[da, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.171 FMOV (immediate, predicated)

Move a floating-point immediate into each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is an alias of the FCPY instruction. This means that:

- The encodings in this description are named to match the encodings of FCPY.
- The description of FCPY gives the operational pseudocode for this instruction.

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12| 5 4 | 0 |
| 0 0 0 0 0 1 0 1 | size | 0 1 | Pg | 1 1 0 | imm8 | Zd |
```

**Encoding**

FMOV <Zd>,<T>, <Pg>/M, #<const>

is equivalent to

FCPY <Zd>,<T>, <Pg>/M, #<const>

and is always the preferred disassembly.

**Assembler symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - `H` when size = 01
  - `S` when size = 10
  - `D` when size = 11
  - The encoding size = 00 is reserved.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<const>` Is a floating-point immediate value expressable as ±n÷16×2^r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

**Operation**

The description of FCPY gives the operational pseudocode for this instruction.

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.172   FMOV (immediate, unpredicated)

Unconditionally broadcast the floating-point immediate into each element of the destination vector. This instruction is unpredicated.

This instruction is an alias of the FDUP instruction. This means that:

- The encodings in this description are named to match the encodings of FDUP.
- The description of FDUP gives the operational pseudocode for this instruction.

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 5  4  0 | 0 1 0 0 1 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 0 0 |

Encoding

FMOV <Zd>,<T>, #<const>

is equivalent to

FDUP    <Zd>,<T>, #<const>

and is always the preferred disassembly.

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

<const> Is a floating-point immediate value expressable as ±n×16×2^r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

Operation

The description of FDUP gives the operational pseudocode for this instruction.
Move floating-point constant +0.0 to to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is a pseudo-instruction of the CPY (immediate, merging) instruction. This means that:

- The encodings in this description are named to match the encodings of CPY (immediate, merging).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

The encoding

\[
\begin{array}{cccccccccc}
\text{size} & \text{Pg} & \text{M} & \text{sh} & \text{imm8} \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

is equivalent to

\[
\text{CPY} \ <Zd>.<T>, \ <Pg>/M, \ #0, \#0
\]

and is never the preferred disassembly.

**Assembler symbols**

- \(<Zd>\) is the name of the destination scalable vector register, encoded in the "Zd" field.
- \(<T>\) is the size specifier, encoded in the "size" field. It can have the following values:
  - \(H\) when \(\text{size} = 01\)
  - \(S\) when \(\text{size} = 10\)
  - \(D\) when \(\text{size} = 11\)
  - The encoding \(\text{size} = 00\) is reserved.
- \(<Pg>\) is the name of the governing scalable predicate register, encoded in the "Pg" field.

**Operation**

The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.174  FMOV (zero, unpredicated)

Unconditionally broadcast the floating-point constant +0.0 into each element of the destination vector. This instruction is unpredicated.

This instruction is a pseudo-instruction of the DUP (immediate) instruction. This means that:

• The encodings in this description are named to match the encodings of DUP (immediate).
• The assembler syntax is used only for assembly, and is not used on disassembly.
• The description of DUP (immediate) gives the operational pseudocode for this instruction.

Encoding

FMOV <Zd>.<T>, #0.0

is equivalent to

DUP <Zd>.<T>, #0

and is never the preferred disassembly.

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:

<table>
<thead>
<tr>
<th>H</th>
<th>when size = 01</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>when size = 10</td>
</tr>
<tr>
<td>D</td>
<td>when size = 11</td>
</tr>
</tbody>
</table>

The encoding size = 00 is reserved.

Operation

The description of DUP (immediate) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.175  FMSB

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third (addend) vector without intermediate rounding. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

Encoding

FMSB <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;

Assembler symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>   Is the size specifier, encoded in the "size" field. It can have the following values:

H  when size = 01
S  when size = 10
D  when size = 11

The encoding size = 00 is reserved.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za>  Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
bits(esize) element3 = Elem[operand3, e, esize];

if op1_neg then element1 = FPNeg(element1);
if op3_neg then element3 = FPNeg(element3);
Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
else
  Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.176 FMUL (immediate)

Multiply by an immediate each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +2.0 only. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28| 27 26 25 24| 23 22 21 20| 19 18 17 16| 15 14 13 12| 10 9 8 | 7 | 6 | 5 | 4 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | i1 | Zdn |

Encoding

FMUL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then FPPointFive('0', esize) else FPTwo('0', esize);

Assembler symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  
  H when size = 01
  S when size = 10
  D when size = 11
  
  The encoding size = 00 is reserved.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in the "i1" field. It can have the following values:
  
  #0.5 when i1 = 0
  #2.0 when i1 = 1

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPMul(element1, imm, FPCR[]);
  else
    Elem[result, e, esize] = element1;

\[ Z[dn, VL] = \text{result}; \]

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.177 FMUL (indexed)

Multiply all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The results are placed in the corresponding elements of the destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

**Half-precision**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 0</td>
<td>0 1 1</td>
<td>i3l</td>
<td>Zm</td>
<td>0 0 1 0</td>
<td>0 0</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>

Encoding


**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
inference integer index = UInt(i3h:i3l);
inference integer n = UInt(Zn);
inference integer m = UInt(Zm);
inference integer d = UInt(Zd);
```

**Single-precision**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 0</td>
<td>0 1 1</td>
<td>i2</td>
<td>Zm</td>
<td>0 0 1 0</td>
<td>0 0</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>

Encoding

FMUL <Zd>.S, <Zn>.S, <Zm>.S[<imm>]

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
inference integer index = UInt(i2);
inference integer n = UInt(Zn);
inference integer m = UInt(Zm);
inference integer d = UInt(Zd);
```
Double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 0 1 1 1 1</td>
<td>Zm 0 0 1 0 0 0</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>

size<1>
size<0>

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;

constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the half-precision and single-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the half-precision variant: is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the "i2" field.

For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the "i1" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
integer segmentbase = e - (e MOD eltspersegment);
integer s = segmentbase + index;
bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, s, esize];
Elem[result, e, esize] = FMUL(element1, element2, FPCR[]);

Z[d, VL] = result;
C8.2.178   FMUL (vectors, predicated)

Multiply active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>0 0 0 0 1 0 1 0</td>
<td>Pg</td>
<td>Zm</td>
<td>Zdn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FMUL <Zdn>,<T>, <Pg>/M, <Zdn>,<T>, <Zm>,<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

**Assembler symbols**

<Zdn>   Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>     Is the size specifier, encoded in the "size" field. It can have the following values:
       H when size = 01
       S when size = 10
       D when size = 11
       The encoding size = 00 is reserved.
<Pg>    Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>    Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if Elem[mask, e, esize] == '1' then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);  
  else
    Elem[result, e, esize] = element1;
Z[dn, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.179 FMUL (vectors, unpredicated)

Multiply all elements of the first source vector by corresponding floating-point elements of the second source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

```
  31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13 12|11 10 9 |  5 4 |  0  |
     0 1 1 0 0 1 0 1 | size | 0 | Zm  | 0 0 0 0 1 0 | Zn | Zd  |
```

**Encoding**

FMUL <Zd>,<T>, <Zn>,<T>, <Zm>,<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);
Z[d, VL] = result;
C8.2.180   FMULX

Multiply active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector except that $\infty \times 0.0$ gives 2.0 instead of NaN, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

The instruction can be used with FRECPX to safely convert arbitrary elements in mathematical vector space to UNIT VECTORS or DIRECTION VECTORS with length 1.

Encoding

FMULX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn>   Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>    Is the size specifier, encoded in the "size" field. It can have the following values:
        H when size = 01
        S when size = 10
        D when size = 11
        The encoding size = 00 is reserved.
<Pg>   Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>   Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if Elem[mask, e, esize] == '1' then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMulX(element1, element2, FPCR[ ]);
  else

Elem[result, e, esize] = element1;

Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.181 FNEG

Negate each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This inverts the sign bit and cannot signal a floating-point exception. Inactive elements in the destination vector register remain unmodified.

Encoding

FNEG <Zd>.<T>, <Pg>/M, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  H when size = 01
  S when size = 10
  D when size = 11
  The encoding size = 00 is reserved.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPNeg(element);
Z[d, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.182 FNMAD

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third (addend) vector without intermediate rounding. Destructively place the negated results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

Encoding

FNMAD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
ingter dn = UInt(Zdn);
ingter m = UInt(Zm);
ingter a = UInt(Za);
boolean op1_neg = TRUE;
boolean op3_neg = TRUE;

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11

The encoding size = 00 is reserved.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Pg, PL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
bits(esize) element3 = Elem[operand3, e, esize];
if op1_neg then element1 = FPNeg(element1);
if op3_neg then element3 = FPNeg(element3);
Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
else
  Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.183 FNMLA

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third source (addend) vector without intermediate rounding. Destructively place the negated results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

Encoding

FNMLA <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = TRUE;

Assembler symbols

<Zda>  Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<T>    Is the size specifier, encoded in the "size" field. It can have the following values:
        H when size = 01
        S when size = 10
        D when size = 11
        The encoding size = 00 is reserved.
<Pg>   Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>   Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>   Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Pg, PL;
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];

01100101
0x0 size 1
0x0 Zm 0
0x0 1
0x0 Pg
0x0 Zn
0x0 Zda
31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13 12| 10 9 | 5 4 | 0 |
bits(esize) element3 = Elem[operand3, e, esize];

if op1_neg then element1 = FPNeg(element1);
if op3_neg then element3 = FPNeg(element3);
  Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
else
  Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.184   FNMLS

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third source (addend) vector without intermediate rounding. Destructively place the negated results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

Encoding

FNMLS <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = TRUE;

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11

The encoding size = 00 is reserved.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
bits(esize) element3 = Elem[operand3, e, esize];

if op1_neg then element1 = FPNeg(element1);
if op3_neg then element3 = FPNeg(element3);
Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
else
    Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.185 FNMSB

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third (addend) vector without intermediate rounding. Destructively place the negated results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

**Encoding**

\[
\text{FNMSB } \langle \text{Zdn} \rangle . \langle \text{T} \rangle, \langle \text{Pg} \rangle / \langle \text{M} \rangle, \langle \text{Zm} \rangle . \langle \text{T} \rangle, \langle \text{Za} \rangle . \langle \text{T} \rangle
\]

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean op1_neg = FALSE;
boolean op3_neg = TRUE;

**Assembler symbols**

\(<\text{Zdn}>\)

Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

\(<\text{T}>\)

Is the size specifier, encoded in the "size" field. It can have the following values:

- \(H\) when size = 01
- \(S\) when size = 10
- \(D\) when size = 11

The encoding size = 00 is reserved.

\(<\text{Pg}>\)

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<\text{Zm}>\)

Is the name of the second source scalable vector register, encoded in the "Zm" field.

\(<\text{Za}>\)

Is the name of the third source scalable vector register, encoded in the "Za" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
bits(esize) element3 = Elem[operand3, e, esize];
if op1_neg then element1 = FPNeg(element1);
if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.186  FRECPE

Find the approximate reciprocal of each floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

\[
\begin{array}{cccccccccccccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & Zn & Zd
\end{array}
\]

Encoding

FRECPE <Zd>.<T>, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
n = UInt(Zn);
d = UInt(Zd);

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

H     when size = 01
S     when size = 10
D     when size = 11

The encoding size = 00 is reserved.

<Zn>  Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
const integer VL = CurrentVL;
const integer PL = VL DIV 8;
const integer elements = VL DIV esize;
bits(VL) operand = Z[n, VL];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecipEstimate[element, FPCR[]];
Z[d, VL] = result;
C8.2.187 FRECPS

Multiply corresponding floating-point elements of the first and second source vectors, subtract the products from 2.0 without intermediate rounding and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction can be used to perform a single Newton-Raphson iteration for calculating the reciprocal of a vector of floating-point values.

```
[31 30 29 28|27 26 25 24|23 22 21 20]  16|15 14 13 12|11 10  9 |  5  4 |  0 |
 0 1 1 0 0 1 0 1 | size 0 | Zm 0 0 0 1 1 0 | Zn | Zd |
```

Encoding

FRECPS <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRecipStepFused(element1, element2);
Z[d, VL] = result;
```
FRECPX

Invert the exponent and zero the fractional part of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

The result of this instruction can be used with FMULX to convert arbitrary elements in mathematical vector space to "unit vectors" or "direction vectors" of length 1.

Encoding

FRECPX <Zd>.<T>, <Pg>/M, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
   H when size = 01
   S when size = 10
   D when size = 11
   The encoding size = 00 is reserved.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
   if Elem[mask, e, esize] == '1' then
      bits(esize) element = Elem[operand, e, esize];
      Elem[result, e, esize] = FReCPx(element, FPCR[]);
   end if
Z[d, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.189  FRINT<r>

Round to an integral floating-point value with the specified rounding option from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>&lt;r&gt;</th>
<th>Rounding Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>to nearest, with ties to even</td>
</tr>
<tr>
<td>A</td>
<td>to nearest, with ties away from zero</td>
</tr>
<tr>
<td>M</td>
<td>toward minus Infinity</td>
</tr>
<tr>
<td>P</td>
<td>toward plus Infinity</td>
</tr>
<tr>
<td>Z</td>
<td>toward zero</td>
</tr>
<tr>
<td>I</td>
<td>current FPCR rounding mode</td>
</tr>
<tr>
<td>X</td>
<td>current FPCR rounding mode, signalling inexact</td>
</tr>
</tbody>
</table>

Current mode

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10  9</th>
<th>5  4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Encoding

FRINTI <Zd>.<T>, <Pg>/M, <Zn>.<T>

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Current mode signalling inexact

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10  9</th>
<th>5  4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Encoding

FRINTX <Zd>.<T>, <Pg>/M, <Zn>.<T>
Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
costant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);

Nearest with ties to away

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>size 0 0 0 1 0 0 1 0</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FRINTA <Zd>.<T>, <Pg>/M, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
costant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_TIEAWAY;

Nearest with ties to even

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>size 0 0 0 0 0 0 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FRINTN <Zd>.<T>, <Pg>/M, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
costant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_TIEEVEN;
### Toward zero

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10  9 |  5  4 |  0 |
| 0 1 1 0 0 1 0 1 | size 0 0 0 | 0 1 1 0 1 | Pg  Zn | Zd |
```

**Encoding**

`FRINTZ <Zd>.<T>, <Pg>/M, <Zn>.<T>`

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

### Toward minus infinity

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10  9 |  5  4 |  0 |
| 0 1 1 0 0 1 0 1 | size 0 0 0 | 0 1 0 1 | Pg  Zn | Zd |
```

**Encoding**

`FRINTM <Zd>.<T>, <Pg>/M, <Zn>.<T>`

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_NEGINF;
```

### Toward plus infinity

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10  9 |  5  4 |  0 |
| 0 1 1 0 0 1 0 1 | size 0 0 0 | 0 1 0 1 | Pg  Zn | Zd |
```

**Encoding**

`FRINTP <Zd>.<T>, <Pg>/M, <Zn>.<T>`

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
```
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_POSINF;

Assembler symbols

\(<Zd>\) Is the name of the destination scalable vector register, encoded in the "Zd" field.

\(<T>\) Is the size specifier, encoded in the "size" field. It can have the following values:

\(H\) when size = 01
\(S\) when size = 10
\(D\) when size = 11

The encoding size = 00 is reserved.

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Zn>\) Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

Z[d, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.190   FRSQRTE

Find the approximate reciprocal square root of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>size 0 0 1 1</td>
<td>0 0 1 1 0 0</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

FRSQRTE <Zd>,<T>, <Zn>,<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand = Z[n, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRSqrtEstimate(element, FPCR[]);
Z[d, VL] = result;
C8.2.191 FRSQRTS

Multiply corresponding floating-point elements of the first and second source vectors, subtract the products from 3.0 and divide the results by 2.0 without any intermediate rounding and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction can be used to perform a single Newton-Raphson iteration for calculating the reciprocal square root of a vector of floating-point values.

**Encoding**

FRSQRTS <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

**Assembler symbols**

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in the "size" field. It can have the following values:
  - **H** when size = 01
  - **S** when size = 10
  - **D** when size = 11
  
The encoding size = 00 is reserved.
- **<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPRSqrtStepFused(element1, element2);
Z[d, VL] = result;
```
C8.2.192 FSCALE

Multiply the active floating-point elements of the first source vector by 2.0 to the power of the signed integer values in the corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>0 0 1 0 1 1 0</td>
<td>Pg</td>
<td>Zm</td>
<td>Zdn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FSCALE <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

**Assembler symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11

The encoding size = 00 is reserved.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
bits(esize) element1 = Elem[operand1, e, esize];
if Elem[mask, e, esize] == '1' then
    integer element2 = SInt(Elem[operand2, e, esize]);
    Elem[result, e, esize] = FPScale(element1, element2, FPCR[]);
else
    Elem[result, e, esize] = element1;
Z[dn, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.193 FSQRT

Calculate the square root of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

![Table](image)

**Encoding**

FSQRT <Zd>.<T>, <Pg>/M, <Zn>.<T>

**Decode for this encoding**

```plaintext
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

**Assembler symbols**

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>`: Is the size specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11
  The encoding size = 00 is reserved.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>`: Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

```plaintext
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPSqrt(element, FPCR[]);
  Z[d, VL] = result;
```

```plaintext
31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 8 7 6 5 | 0 1 2 3 4 5 6 7
0 1 1 0 0 1 0 1 | size 0 0 1 1 0 1 0 1 | Pg | Zn | Zd
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.194  FSUB (immediate)

Subtract an immediate from each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9 8 7 6 5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>size 0 1 0 0 1 1 0 0</td>
<td>Pg 0 0 0 0</td>
<td>i1</td>
<td>Zdn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

FSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then FPPointFive('0', esize) else FPOne('0', esize);

Assembler symbols

<Zdn>  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

H  when size = 01
S  when size = 10
D  when size = 11

The encoding size = 00 is reserved.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const>  Is the floating-point immediate value, encoded in the "i1" field. It can have the following values:

#0.5  when i1 = 0
#1.0  when i1 = 1

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
bits(esize) element1 = Elem[operand1, e, esize];
if ElemP[mask, e, esize] == '1' then
  Elem[result, e, esize] = FPSub(element1, imm, FPCR[]);
else
  Elem[result, e, esize] = element1;
Z[dn, VL] = result;

Operational information
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.195 **FSUB (vectors, predicated)**

Subtract active floating-point elements of the second source vector from corresponding floating-point elements of
the first source vector and destructively place the results in the corresponding elements of the first source vector.
Inactive elements in the destination vector register remain unmodified.

```
31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 |
0 1 1 0 0 1 0 1 | size | 0 0 0 0 | 1 | 1 0 0 | Pg | Zm | Zdn
```

**Encoding**

FSUB `<Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>`

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

**Assembler symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11
  - The encoding size = 00 is reserved.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if Elem[mask, e, esize] == '1' then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPSub(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn, VL] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.196  FSUB (vectors, unpredicated)

Subtract all floating-point elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

```
| 31 30 29 28|27 26 25 24|23 22 21 20 | 16|15 14 13 12|11 10 9 |  5 4  |  0 |
|----------|----------|----------|-----|---------|-----|-----|-----|-----|
| 0 1 1 0 0 1 0 1| size 0 | Zm | 0 0 0 | 0 0 | 1 | Zn | Zd |
```

**Encoding**

FSUB <Zd>,<T>,<Zn>,<T>,<Zm>,<T>

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

**Assembler symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11
  - The encoding size = 00 is reserved.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements -1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPSub(element1, element2, FPCR[]);
Z[d, VL] = result;
```
### FSUBR (immediate)

Reversed subtract from an immediate each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

#### Encoding

FSUBR \(<Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>"

#### Decode for this encoding

```plaintext
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then FPPointFive('0', esize) else FPOne('0', esize);
```

#### Assembler symbols

- **<Zdn>** Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- **<T>** Is the size specifier, encoded in the "size" field. It can have the following values:
  - **H** when size = 01
  - **S** when size = 10
  - **D** when size = 11

  The encoding size = 00 is reserved.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<const>** Is the floating-point immediate value, encoded in the "i1" field. It can have the following values:
  - **#0.5** when i1 = 0
  - **#1.0** when i1 = 1

#### Operation

```plaintext
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPSub(imm, element1, FPCR[]);
  else
    Elem[result, e, esize] = element1;
```

---

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 10 9 8 7 6 5 4 | 0 |
|------|------|------|------|------|------|------|------|
| 0 1 1 0 0 1 0 1 | size 0 1 1 0 1 1 0 0 | Pg 0 0 0 0 i1 | Zdn |
\[ Z[dn, VL] = \text{result}; \]

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.198   FSUBR (vectors)

Reversed subtract active floating-point elements of the first source vector from corresponding floating-point elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

FSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11

The encoding size = 00 is reserved.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if Elem[mask, e, esize] == '1' then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPSub(element2, element1, FPCR[]);
  else
    Elem[result, e, esize] = element1;

Z[dn, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.199 FTMAD

The FTMAD instruction calculates the series terms for either $\sin(X)$ or $\cos(X)$, where the argument $X$ has been adjusted to be in the range $-\pi/4 < X \leq \pi/4$.

To calculate the series terms of $\sin(X)$ and $\cos(X)$ the initial source operands of FTMAD should be zero in the first source vector and $X^2$ in the second source vector. The FTMAD instruction is then executed eight times to calculate the sum of eight series terms, which gives a result of sufficient precision.

The FTMAD instruction multiplies each element of the first source vector by the absolute value of the corresponding element of the second source vector and performs a fused addition of each product with a value obtained from a table of hard-wired coefficients, and places the results destructively in the first source vector.

The coefficients are different for $\sin(X)$ and $\cos(X)$, and are selected by a combination of the sign bit in the second source element and an immediate index in the range 0 to 7.

### Double-precision coefficient table for sin(x) (s2<63> == '0')

<table>
<thead>
<tr>
<th>Index</th>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Exact Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3ff0 0000 0000 0000</td>
<td>1.0</td>
<td>= 1/1!</td>
</tr>
<tr>
<td>1</td>
<td>bfc5 5555 5555 5543</td>
<td>-0.1666666666666661</td>
<td>&gt; -1/3!</td>
</tr>
<tr>
<td>2</td>
<td>3f11 1111 1110 f30c</td>
<td>0.8333333333332002e-02</td>
<td>&lt; 1/5!</td>
</tr>
<tr>
<td>3</td>
<td>b2a0 19b9 2fc6</td>
<td>-0.1984126982840213e-03</td>
<td>&gt; -1/7!</td>
</tr>
<tr>
<td>4</td>
<td>3e51 d22b</td>
<td>0.2755731329901505e-02</td>
<td>&lt; 1/9!</td>
</tr>
<tr>
<td>5</td>
<td>bee2 7b91</td>
<td>-0.2505070584637887e-07</td>
<td>&gt; -1/11!</td>
</tr>
<tr>
<td>6</td>
<td>3de5 8868 552f</td>
<td>0.1589413637195215e-09</td>
<td>&lt; 1/13!</td>
</tr>
<tr>
<td>7</td>
<td>0000 0000 0000 0000</td>
<td>0.0</td>
<td>&gt; -1/15!</td>
</tr>
</tbody>
</table>

### Double-precision coefficient table for cos(x) (s2<63> == '1')

<table>
<thead>
<tr>
<th>Index</th>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Exact Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3ff0 0000 0000 0000</td>
<td>1.0</td>
<td>= 1/0!</td>
</tr>
<tr>
<td>1</td>
<td>bfe0 0000 0000 0000</td>
<td>-0.5000000000000000</td>
<td>= -1/2!</td>
</tr>
<tr>
<td>2</td>
<td>3fa5 5555 5555 5536</td>
<td>0.4166666666666664e-01</td>
<td>&lt; 1/4!</td>
</tr>
<tr>
<td>3</td>
<td>bf56 c16c 16c1 3a0b</td>
<td>-0.138888888888611e-02</td>
<td>&gt; -1/6!</td>
</tr>
<tr>
<td>4</td>
<td>3efa 01a0 19b1 e8d8</td>
<td>0.2480158728388683e-04</td>
<td>&lt; 1/8!</td>
</tr>
<tr>
<td>5</td>
<td>be92 7e4f 7282 f468</td>
<td>-0.2755731309913950e-06</td>
<td>&gt; -1/10!</td>
</tr>
<tr>
<td>6</td>
<td>3e21 ee96 d264 1b13</td>
<td>0.2087558253975872e-08</td>
<td>&lt; 1/12!</td>
</tr>
<tr>
<td>7</td>
<td>bda8 f763 80fb b401</td>
<td>-0.113533870720054e-10</td>
<td>&gt; -1/14!</td>
</tr>
</tbody>
</table>
### Single-precision coefficient table for \( \sin(x) \) (s2<31> == '0')

<table>
<thead>
<tr>
<th>Index</th>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Exact Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3R80 0000</td>
<td>1.0</td>
<td>= 1/1!</td>
</tr>
<tr>
<td>1</td>
<td>be2a aaab</td>
<td>-1.6666666716337e-01</td>
<td>&gt; -1/3!</td>
</tr>
<tr>
<td>2</td>
<td>3c08 8886</td>
<td>8.333330973983e-03</td>
<td>&lt; 1/5!</td>
</tr>
<tr>
<td>3</td>
<td>b950 08b9</td>
<td>-1.983967522392e-04</td>
<td>&gt; -1/7!</td>
</tr>
<tr>
<td>4</td>
<td>3636 9d6d</td>
<td>2.721174723774e-06</td>
<td>&lt; 1/9!</td>
</tr>
<tr>
<td>5</td>
<td>0000 0000</td>
<td>0.0</td>
<td>&gt; -1/11!</td>
</tr>
<tr>
<td>6</td>
<td>0000 0000</td>
<td>0.0</td>
<td>&lt; 1/13!</td>
</tr>
<tr>
<td>7</td>
<td>0000 0000</td>
<td>0.0</td>
<td>&gt; -1/15!</td>
</tr>
</tbody>
</table>

### Single-precision coefficient table for \( \cos(x) \) (s2<31> == '1')

<table>
<thead>
<tr>
<th>Index</th>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Exact Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3R80 0000</td>
<td>1.0</td>
<td>= 1/0!</td>
</tr>
<tr>
<td>1</td>
<td>bf00 0000</td>
<td>-5.000000000000e-01</td>
<td>= -1/2!</td>
</tr>
<tr>
<td>2</td>
<td>3d2a aaa6</td>
<td>4.166664928198e-02</td>
<td>&lt; 1/4!</td>
</tr>
<tr>
<td>3</td>
<td>bab6 0705</td>
<td>-1.3887590003021e-03</td>
<td>&gt; -1/6!</td>
</tr>
<tr>
<td>4</td>
<td>37cd 37cc</td>
<td>2.446388680255e-05</td>
<td>&lt; 1/8!</td>
</tr>
<tr>
<td>5</td>
<td>0000 0000</td>
<td>0.0</td>
<td>&gt; -1/10!</td>
</tr>
<tr>
<td>6</td>
<td>0000 0000</td>
<td>0.0</td>
<td>&lt; 1/12!</td>
</tr>
<tr>
<td>7</td>
<td>0000 0000</td>
<td>0.0</td>
<td>&gt; -1/14!</td>
</tr>
</tbody>
</table>

### Half-precision coefficient table for \( \sin(x) \) (s2<15> == '0')

<table>
<thead>
<tr>
<th>Index</th>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Exact Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3c00</td>
<td>1.0</td>
<td>= 1/1!</td>
</tr>
<tr>
<td>1</td>
<td>b155</td>
<td>-1.6666666716337e-01</td>
<td>&gt; -1/3!</td>
</tr>
<tr>
<td>2</td>
<td>2030</td>
<td>8.333330973983e-03</td>
<td>&lt; 1/5!</td>
</tr>
<tr>
<td>3</td>
<td>0000</td>
<td>0.0</td>
<td>&gt; -1/7!</td>
</tr>
<tr>
<td>4</td>
<td>0000</td>
<td>0.0</td>
<td>&lt; 1/9!</td>
</tr>
<tr>
<td>5</td>
<td>0000</td>
<td>0.0</td>
<td>&gt; -1/11!</td>
</tr>
<tr>
<td>6</td>
<td>0000</td>
<td>0.0</td>
<td>&lt; 1/13!</td>
</tr>
<tr>
<td>7</td>
<td>0000</td>
<td>0.0</td>
<td>&gt; -1/15!</td>
</tr>
</tbody>
</table>
Half-precision coefficient table for \( \cos(x) \) (s2<15> == '1')

<table>
<thead>
<tr>
<th>Index</th>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Exact Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3c00</td>
<td>1.0</td>
<td>= 1/0!</td>
</tr>
<tr>
<td>1</td>
<td>b800</td>
<td>-5.000000000000e-01</td>
<td>= -1/2!</td>
</tr>
<tr>
<td>2</td>
<td>293a</td>
<td>4.166664928198e-02</td>
<td>&lt; 1/4!</td>
</tr>
<tr>
<td>3</td>
<td>0000</td>
<td>0.0</td>
<td>&gt; -1/6!</td>
</tr>
<tr>
<td>4</td>
<td>0000</td>
<td>0.0</td>
<td>&lt; 1/8!</td>
</tr>
<tr>
<td>5</td>
<td>0000</td>
<td>0.0</td>
<td>&gt; -1/10!</td>
</tr>
<tr>
<td>6</td>
<td>0000</td>
<td>0.0</td>
<td>&lt; 1/12!</td>
</tr>
<tr>
<td>7</td>
<td>0000</td>
<td>0.0</td>
<td>&gt; -1/14!</td>
</tr>
</tbody>
</table>

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18] 16 15 14 13 12 11 10 9 | 5 4 | 0 |
0 1 1 0 0 1 0 1 | size 0 1 0 | imm3 1 0 0 0 0 0 | Zm | Zdn |
```

**Encoding**

FTMAD \(<Zdn>\).<\text{T}>\), \(<Zdn>\).<\text{T}>\), \(<Zm>\).<\text{T}>\), #\(<\text{imm}>\)

**Decode for this encoding**

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer imm = UInt(imm3);

**Assembler symbols**

\(<Zdn>\) Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
\(<\text{T}>\) Is the size specifier, encoded in the "size" field. It can have the following values:
- H when size = 01
- S when size = 10
- D when size = 11
The encoding size = 00 is reserved.
\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.
\(<\text{imm}>\) Is the unsigned immediate operand, in the range 0 to 7, encoded in the "imm3" field.
Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPTrigMAdd(imm, element1, element2, FPCR[]);
Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.200   FTSMUL

The FTSMUL instruction calculates the initial value for the FTMAD instruction. The instruction squares each element in the first source vector and then sets the sign bit to a copy of bit 0 of the corresponding element in the second source register, and places the results in the destination vector. This instruction is unpredicated.

To compute \( \sin(X) \) or \( \cos(X) \) the instruction is executed with elements of the first source vector set to \( X \), adjusted to be in the range \(-\pi/4 < X \leq \pi/4\).

The elements of the second source vector hold the corresponding value of the quadrant \( q \) number as an integer not a floating-point value. The value \( q \) satisfies the relationship \((2q-1) \times \pi/4 < X \leq (2q+1) \times \pi/4\).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>0</td>
<td>Zm</td>
<td>0 0 0 0 1 1</td>
<td>Zn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

FTSMUL <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
Elem[result, e, esize] = FPTrigSMul(element1, element2, FPCR[]);
Z[d, VL] = result;
C8.2.201 FTSSEL

The FTSSEL instruction selects the coefficient for the final multiplication in the polynomial series approximation. The instruction places the value 1.0 or a copy of the first source vector element in the destination element, depending on bit 0 of the quadrant number Q held in the corresponding element of the second source vector. The sign bit of the destination element is copied from bit 1 of the corresponding value of Q. This instruction is unpredicated.

To compute SIN(X) or COS(X) the instruction is executed with elements of the first source vector set to X, adjusted to be in the range \(-\pi/4 < X \leq \pi/4\).

The elements of the second source vector hold the corresponding value of the quadrant Q number as an integer not a floating-point value. The value Q satisfies the relationship \((2q-1) \times \pi/4 < X \leq (2q+1) \times \pi/4\).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

Encoding

FTSSEL <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11

The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPTrigSSel(element1, element2);
  Z[d, VL] = result;
C8.2.202  HISTCNT

This instruction compares each active 32 or 64-bit element of the first source vector with all active elements with an element number less than or equal to its own in the second source vector, and places the count of matching elements in the corresponding element of the destination vector. Inactive elements in the destination vector are set to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

Encoding

HISTCNT <Zd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = $ << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Zd);
integer n = UInt(Zn);
integer m = UInt(Zm);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size<0>" field. It can have the following values:
S when size<0> = 0
D when size<0> = 1
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
integer count = 0;
if ElemP[mask, e, esize] == '1' then
bits(esize) element1 = Elem[operand1, e, esize];
for i = 0 to e
if ElemP[mask, i, esize] == '1' then
\begin{verbatim}
\text{bits}(<\text{esize}) \text{ element2 = Elem[operand2, i, esize];}
if element1 == element2 then
    count = count + 1;
    Elem[result, e, esize] = count<esize-1:0>;
\Z[d, VL] = result;
\end{verbatim}
C8.2.203  HISTSEG

This instruction compares each 8-bit byte element of the first source vector with all of the elements in the corresponding 128-bit segment of the second source vector and places the count of matching elements in the corresponding element of the destination vector. This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

Encoding

HISTSEG <Zd>.8, <Zn>.8, <Zm>.8

Decode for this encoding

if !HaveSVE2() then UNDEFINED;
if size != "00" then UNDEFINED;
constant integer esize = $;
integer d = UInt(Zd);
integer n = UInt(Zn);
integer m = UInt(Zm);

Assembler symbols

<Zd>    Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>    Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>    Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for b = 0 to segments-1
  for s = 0 to eltspersegment-1
    integer count = 0;
    integer e = eltspersegment * b + s;
    bits(esize) element1 = Elem[operand1, e, esize];
    for i = 0 to eltspersegment-1
      integer e2 = eltspersegment * b + i;
      bits(esize) element2 = Elem[operand2, e2, esize];
      if element1 == element2 then
        count = count + 1;
      Elem[result, e, esize] = count<esize-1:0>;
    Z[d, VL] = result;
C8.2.204  INCB, INCD, INCH, INCW (scalar)

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

**Byte**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>0 1 1</td>
<td>imm4</td>
<td>1 1 1 0 0</td>
<td>0</td>
<td>pattern</td>
<td>Rdn</td>
</tr>
</tbody>
</table>

**Encoding**

INCB <Xdn>{, <pattern>{, MUL #<imm>}}

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

**Doubleword**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>1 1 1</td>
<td>imm4</td>
<td>1 1 1 0 0</td>
<td>0</td>
<td>pattern</td>
<td>Rdn</td>
</tr>
</tbody>
</table>

**Encoding**

INCD <Xdn>{, <pattern>{, MUL #<imm>}}

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
SVE Instruction Descriptions

C8.2 Alphabetical list of SVE instructions

Halfword

\[
\begin{array}{ccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & \text{imm4} & 1 & 1 & 1 & 0 & 0 & \text{pattern} & \text{Rdn} & \end{array}
\]

\text{size}<1>

\text{size}<0>

\text{Encoding}

\text{INCH} \langle Xdn\rangle \{, \langle \text{pattern}\rangle\{, \text{MUL}\ \#<\text{imm}>\}\}

\text{Decode for this encoding}

if (!\text{HaveSVE()} \&\& !\text{HaveSME()}) then UNDEFINED;
constant integer esize = 16;
integer dn = \text{UInt}(Rdn);
bits(5) pat = \text{pattern};
integer imm = \text{UInt}(\text{imm4}) + 1;

Word

\[
\begin{array}{ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
### C8.2 Alphabetical list of SVE instructions

#### VL8
- when pattern = 01000

#### VL16
- when pattern = 01001

#### VL32
- when pattern = 01010

#### VL64
- when pattern = 01011

#### VL128
- when pattern = 01100

#### VL256
- when pattern = 01101

#### #uimm5
- when pattern = 0111x

#### #uimm5
- when pattern = 101x1

#### #uimm5
- when pattern = 10110

#### #uimm5
- when pattern = 1x0x1

#### #uimm5
- when pattern = 1x010

#### #uimm5
- when pattern = 1xx00

#### MUL4
- when pattern = 11101

#### MUL3
- when pattern = 11110

#### ALL
- when pattern = 11111

<imm>  
Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

### Operation for all encodings

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
constant integer VL = CurrentVL;
bits(64) operand1 = X[dn, 64];
X[dn, 64] = operand1 + (count * imm);
```

### Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.205 INCD, INCH, INCW (vector)

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

**Doubleword**

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 1 1 1</td>
<td>imm4</td>
<td>1 1 0 0 0 0</td>
</tr>
</tbody>
</table>
```

Decoding:

```
INCD <Zdn>.D{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding:

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

**Halfword**

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 1 1</td>
<td>imm4</td>
<td>1 1 0 0 0 0</td>
</tr>
</tbody>
</table>
```

Decoding:

```
INCH <Zdn>.H{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding:

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```
Word

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20 19]</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 1 0 1 1 imm4</td>
<td>1 1 0 0 0 0 pattern</td>
<td>Zdn</td>
<td></td>
</tr>
</tbody>
</table>

size<1>

size<0>

Encoding

INCW <Zdn>.S{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bv(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Assembler symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

- POW2 when pattern = 00000
- VL1 when pattern = 00001
- VL2 when pattern = 00010
- VL3 when pattern = 00011
- VL4 when pattern = 00100
- VL5 when pattern = 00101
- VL6 when pattern = 00110
- VL7 when pattern = 00111
- VL8 when pattern = 01000
- VL16 when pattern = 01001
- VL32 when pattern = 01010
- VL64 when pattern = 01011
- VL128 when pattern = 01100
- VL256 when pattern = 01101
- #uimm5 when pattern = 01110
- #uimm5 when pattern = 01111
- #uimm5 when pattern = 1011x
- #uimm5 when pattern = 10110
- #uimm5 when pattern = 1x01x
- #uimm5 when pattern = 1x010
- #uimm5 when pattern = 1xx00
- MUL4 when pattern = 11101
- MUL3 when pattern = 11110
- ALL when pattern = 11111
<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation for all encodings**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[operand1, e, esize] + (count * imm);
Z[dn, VL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.206  INCP (scalar)

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination.

Encoding

INCP <Xd>, <Pm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);

Assembler symbols

<Xd>  Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Pm>  Is the name of the source scalable predicate register, encoded in the "Pm" field.
<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

B    when size = 00
H    when size = 01
S    when size = 10
D    when size = 11

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) operand1 = X[dn, 64];
bits(PL) operand2 = P[m, PL];
integer count = 0;

for e = 0 to elements-1
   if ElemP[operand2, e, esize] == '1' then
      count = count + 1;
X[dn, 64] = operand1 + count;

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C8.2.207  INCP (vector)

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8 | 5 4 | 0 ]
0 0 1 0 0 1 0 1 | size 1 0 1 1 0 0 1 0 0 0 0 0 0 | Pm | Zdn
```

**Encoding**

INCP <Zdn>,<T>, <Pm>,<T>

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = size << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
```

**Assembler symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11

The encoding size = 00 is reserved.

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;
for e = 0 to elements-1
  Elem[result, e, esize] = Elem[operand1, e, esize] + count;
Z[dn, VL] = result;
```
**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is **UNPREDICTABLE**:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.208   INDEX (immediate, scalar)

Populates the destination vector by setting the first element to the first signed immediate integer operand and monotonically incrementing the value by the second signed scalar integer operand for each subsequent element. The scalar source operand is a general-purpose register in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.

Encoding

INDEX <Zd>.<T>, #<imm>, <R><m>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Rm);
integer d = UInt(Zd);
integer imm = SInt(imm5);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

<imm> Is the signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.

<R> Is a width specifier, encoded in the "size" field. It can have the following values:
W when size = 01
W when size = x0
X when size = 11

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(esize) operand2 = X[m, esize];
integer element2 = SInt(operand2);
bits(VL) result;
for e = 0 to elements-1
    integer index = imm + e * element2;
    Elem[result, e, esize] = index<esize-1:0>;}
\[ Z[d, VL] = \text{result}; \]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.209 INDEX (immediates)

Populates the destination vector by setting the first element to the first signed immediate integer operand and monotonically incrementing the value by the second signed immediate integer operand for each subsequent element. This instruction is unpredicated.

Encoding

INDEX <Zd>.<T>, #<imm1>, #<imm2>

Decode for this encoding

If !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer d = UInt(Zd);
integer imm1 = SInt(imm5);
integer imm2 = SInt(imm5b);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11
<imm1> Is the first signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.
<imm2> Is the second signed immediate operand, in the range -16 to 15, encoded in the "imm5b" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bite(VL) result;
for e = 0 to elements-1
  integer index = imm1 + e * imm2;
  Elem[result, e, esize] = index<esize-1:0>;
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.210 INDEX (scalar, immediate)

Populates the destination vector by setting the first element to the first signed scalar integer operand and monotonically incrementing the value by the second signed immediate integer operand for each subsequent element. The scalar source operand is a general-purpose register in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.

Encoding

INDEX <Zd>.<T>, <R><n>, #<imm>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer d = UInt(Zd);
integer imm = SInt(imm5);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11
<R> Is a width specifier, encoded in the "size" field. It can have the following values:
- W when size = 01
- W when size = x0
- X when size = 11
<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.
<imm> Is the signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(esize) operand1 = X[n, esize];
integer element1 = SInt(operand1);
bits(VL) result;
for e = 0 to elements - 1
   integer index = element1 + e * imm;
\texttt{Elem[\text{result, e, esize}] = index<\text{esize-1:0}>;}
\texttt{Z[d, VL] = result;}

\textbf{Operational information}

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then when PSTATE\_DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.211  INDEX (scalars)

Populates the destination vector by setting the first element to the first signed scalar integer operand and monotonically incrementing the value by the second signed scalar integer operand for each subsequent element. The scalar source operands are general-purpose registers in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.

| [31 30 29 28] | [27 26 25 24] | [23 22 21 20] | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 5 | 4 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | size | 1 | Rm | 0 | 1 | 0 | 1 | 1 | Rn | Zd |

Encoding

INDEX <Zd>.<T>, <R><n>, <R><m>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11
<R> Is a width specifier, encoded in the "size" field. It can have the following values:
  W when size = 01
  W when size = x0
  X when size = 11
<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.
<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(esize) operand1 = X[n, esize];
integer element1 = SInt(operand1);
bits(esize) operand2 = X[m, esize];
integer element2 = SInt(operand2);
bits(VL) result;
for e = 0 to elements-1
   integer index = element1 + e * element2;
   Elem[result, e, esize] = index<esize-1:0>;
   Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.212  **INSR (scalar)**

Shift the destination vector left by one element, and then place a copy of the least-significant bits of the general-purpose register in element 0 of the destination vector. This instruction is unpredicated.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0]  
0 0 0 0 0 1 0 1 | 1 0 0 1 0 0 0 0 1 1 1 0 | Rm | Zdn
```

**Encoding**

INSR <Zdn>.<T>, <R><m>

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer m = UInt(Rm);
```

**Assembler symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<R>` Is a width specifier, encoded in the "size" field. It can have the following values:
  - W when size = 01
  - W when size = x0
  - X when size = 11
- `<m>` Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(VL) dest = Z[dn, VL];
bits(esize) src = X[m, esize];
Z[dn, VL] = dest<(VL-esize)-1:0> : src;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.213 INSR (SIMD&FP scalar)

Shift the destination vector left by one element, and then place a copy of the SIMD&FP scalar register in element 0 of the destination vector. This instruction is unpredicated.

Encoding

INSR <Zdn>.<T>, <V><m>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer m = UInt(Vm);

Assembler symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<m> Is the number [0-31] of the source SIMD&FP register, encoded in the "Vm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(VL) dest = Z[dn, VL];
bits(esize) src = V[m, esize];
Z[dn, VL] = dest<(VL-esize)-1:0> : src;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
— The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.214 LASTA (scalar)

If there is an active element then extract the element after the last active element modulo the number of elements from the final source vector register. If there are no active elements, extract element zero. Then zero-extend and place the extracted element in the destination general-purpose register.

Encoding

```
0 0 0 0 0 1 0 1 | 1 0 0 0 0 0 1 0 1 |
0 0 0 0 0 1 0 1 | size 1 0 0 0 0 0 | Pg Zn Rd
```

```
B
```

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = Z[n, VL];
bits(rsize) result;
integer last = LastActiveElement(mask, esize);
if isBefore then
```
if last < 0 then last = elements - 1;
else
    last = last + 1;
if last >= elements then last = 0;
result = ZeroExtend(Elem[operand, last, esize], rsize);
X[d, rsize] = result;

**Operational information**

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C8.2.215   LASTA (SIMD&FP scalar)

If there is an active element then extract the element after the last active element modulo the number of elements from the final source vector register. If there are no active elements, extract element zero. Then place the extracted element in the destination SIMD&FP scalar register.

Encoding

LASTA <V><d>, <Pg>, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean isBefore = FALSE;

Assembler symbols

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
  B   when size = 00
  H   when size = 01
  S   when size = 10
  D   when size = 11
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B   when size = 00
  H   when size = 01
  S   when size = 10
  D   when size = 11

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = Z[n, VL];
integer last = LastActiveElement(mask, esize);
if isBefore then
    if last < 0 then last = elements - 1;


else
    last = last + 1;
    if last >= elements then last = 0;
    V[d, esize] = Elem[operand, last, esize];
C8.2.216 LASTB (scalar)

If there is an active element then extract the last active element from the final source vector register. If there are no active elements, extract the highest-numbered element. Then zero-extend and place the extracted element in the destination general-purpose register.

Encoding

LASTB <R><d>, <Pg>, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;

constant integer esize = 8 << UInt(size);
constant integer rsize = if esize < 64 then 32 else 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Rd);
boolean isBefore = TRUE;

Assembler symbols

<R> Is a width specifier, encoded in the "size" field. It can have the following values:
  W when size = 01
  W when size = x0
  X when size = 11

<d> Is the number [0-30] of the destination general-purpose register or the name ZR (31), encoded in the "Rd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = Z[n, VL];
bits(rsize) result;
integer last = LastActiveElement(mask, esize);
if isBefore then
if last < 0 then last = elements - 1;
else
    last = last + 1;
if last >= elements then last = 0;
result = ZeroExtend(Elem[operand, last, esize], rsize);
X[d, rsize] = result;

Operational information
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is
dependent on the general-purpose register written by this instruction might be significantly delayed.
C8.2.217 LASTB (SIMD&FP scalar)

If there is an active element then extract the last active element from the final source vector register. If there are no active elements, extract the highest-numbered element. Then place the extracted element in the destination SIMD&FP register.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 |  5 4 |   0]
0 0 0 0 0 1 0 1 |size 1 0 0 0 1 1 0 0 | Pg | Zn | Vd
```

Encoding

LASTB <V><d>, <Pg>, <Zn>.<T>

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean isBefore = TRUE;
```

Assembler symbols

- `<V>` Is a width specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<d>` Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = Z[n, VL];
integer last = LastActiveElement(mask, esize);
if isBefore then
  if last < 0 then last = elements - 1;
```
else
    last = last + 1;
    if last >= elements then last = 0;
    \( V[d, esize] = E[\text{elem}]or, last, esize]; \)
C8.2.218   LD1B (scalar plus immediate)

Contiguous load of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

8-bit element

```
[31 30 29 28|27 26 25 24] 22 21 20|19 16|15 14 13 12| 10 9 | 5 4 | 0 |
1 0 1 0 0 0 0 0 0 imm4 1 0 1 Pg Rn Zt
```

dtype<3:1> ____________
dtype<0> ____________

**Encoding**

LD1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

**Decode for this encoding**

```java
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

16-bit element

```
[31 30 29 28|27 26 25 24] 22 21 20|19 16|15 14 13 12| 10 9 | 5 4 | 0 |
1 0 1 0 0 0 0 0 0 1 imm4 1 0 1 Pg Rn Zt
```

dtype<3:1> ____________
dtype<0> ____________

**Encoding**

LD1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

**Decode for this encoding**

```java
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```
32-bit element

```
[31 30 29 28 | 27 26 25 24 | 22 21 20 | 19 | 16 15 14 13 12 | 10 9 | 5 4 | 0]
1 0 1 0 0 1 0 0 0 1 0 0 1 1 0
```

dtype<3:1>
dtype<0>

Encoding

`LD1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]`

Decode for this encoding

```java
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

64-bit element

```
[31 30 29 28 | 27 26 25 24 | 22 21 20 | 19 | 16 14 13 12 | 10 9 | 5 4 | 0]
1 0 1 0 0 1 0 0 0 1 0 0 1 1 0
```

dtype<3:1>
dtype<0>

Encoding

`LD1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]`

Decode for this encoding

```java
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

Assembler symbols

`<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

`<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<imm>` Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.219   LD1B (scalar plus scalar)

Contiguous load of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

8-bit element

```
1 0 1 0 1 0 0 0 0 0 0 0 | Rm 0 1 0 | Pg | Rn | Zt
```

dtype<3:1> ________
dtype<0> ________

**Encoding**

LD1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer msize = 8;
boolean unsigned = TRUE;
```

16-bit element

```
1 0 1 0 1 0 0 0 0 0 0 1 | Rm 0 1 0 | Pg | Rn | Zt
```

dtype<3:1> ________
dtype<0> ________

**Encoding**

LD1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>]

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = TRUE;
```
32-bit element

```
[31 30 29 28] [27 26 25 24] [22 21 20] [16 15 14 13 12] [10 9] [5 4] 0
1 0 1 0 0 1 0 0 0 1 0 0 1 0 1 0 0 0 1 0 0 1 0
```

dtype<3:1>
dtype<0>

Encoding

```
LDIB { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
```

64-bit element

```
[31 30 29 28] [27 26 25 24] [22 21 20] [16 15 14 13 12] [10 9] [5 4] 0
1 0 1 0 0 1 0 0 0 1 0 0 1 0 1 0 0 1 0 0 1 0 0 1 0
```

dtype<3:1>
dtype<0>

Encoding

```
LDIB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
```

Assembler symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);
Z[t, VL] = result;
C8.2.220   LD1B (scalar plus vector)

Gather load of unsigned bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit unpacked unscaled offset

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 | 0 |
   1 1 0 0 0 1 0 0 0 0 xs 0 Zm 0 1 0 Pg Rn Zt
```

Encoding
LD1B {<Zt>.D}, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

Decode for this encoding
```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

32-bit unscaled offset

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 | 0 |
   1 0 0 0 0 1 0 0 0 0 xs 0 Zm 0 1 0 Pg Rn Zt
```

Encoding
LD1B {<Zt>.S}, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

Decode for this encoding
```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```
64-bit unscaled offset

Encoding

LD1B \{ <Zt>.D \}, \langle Pg\rangle/Z, [\langle Xn|SP\rangle, <Zm>.D]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

UXTW when xs = 0
SXTW when xs = 1

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
   if n == 31 & ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
   else
      if n == 31 then CheckSPAlignment();
      base = if n == 31 then SP[] else X[n, 64];

offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.221   LD1B (vector plus immediate)

Gather load of unsigned bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

### 32-bit element

![LD1B 32-bit encoding](image)

**Encoding**

```
LD1B { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]
```

**Decode for this encoding**

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

### 64-bit element

![LD1B 64-bit encoding](image)

**Encoding**

```
LD1B { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]
```

**Decode for this encoding**

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

### Assembler symbols

- `<Zt>`  
  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

**Operation for all encodings**

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
  base = Z[n, VL];
for e = 0 to elements-1
  if Elem(mask, e, esize) == '1' then
    bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);
Z[t, VL] = result;
C8.2.222   LD1D (scalar plus immediate)

Contiguous load of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

Encoding

LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

Assembler symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>  Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);
  end
Z[t, VL] = result;
C8.2.223   LD1D (scalar plus scalar)

Contiguous load of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

Encoding

LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;

Assembler symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
binary(PL) mask = P[g, PL];
binary(VL) result;
binary(msize) data;
binary(64) offset;
constant integer nbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainsUnpredictableBool(Unpredictable_CHECKSMPNONEACTIVE) then CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
base = if n == 31 then SP[] else X[n, 64];
offset = X[m, 64];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
LD1D (scalar plus vector)

Gather load of doublewords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit unpacked scaled offset

\[
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &amp; 1 &amp; 0 &amp; 0 &amp; 0 &amp; 1 &amp; 0 &amp; 1 &amp; 1 &amp; xs &amp; 1</td>
<td>Zm &amp; 0 &amp; 1 &amp; 0</td>
<td>Pg</td>
</tr>
<tr>
<td>U &amp; ff</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\]

Encoding

LD1D \{ <Zt>.D \}, <Pg>/Z, \{<Xn|SP>, <Zm>.D, <mod> #3\}

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 3;

32-bit unpacked unscaled offset

\[
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &amp; 1 &amp; 0 &amp; 0 &amp; 0 &amp; 1 &amp; 0 &amp; 1 &amp; 1 &amp; xs &amp; 0</td>
<td>Zm &amp; 0 &amp; 1 &amp; 0</td>
<td>Pg</td>
</tr>
<tr>
<td>U &amp; ff</td>
<td></td>
<td></td>
</tr>
<tr>
<td>msz&lt;1&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>msz&lt;0&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\]

Encoding

LD1D \{ <Zt>.D \}, <Pg>/Z, \{<Xn|SP>, <Zm>.D, <mod>\}

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
64-bit scaled offset

LD1D {<Zt>.D }, <Pg>/Z, [<Xn|SP>,<Zm>.D, LSL #3]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 3;

64-bit unscaled offset

LD1D {<Zt>.D }, <Pg>/Z, [<Xn|SP>,<Zm>.D]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

- **UXTW** when \( xs = 0 \)
- **SXTW** when \( xs = 1 \)

### Operation for all encodings

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];
  for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
      integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
      bits(64) addr = base + (off << scale);
      data = Mem[addr, mbytes, AccType_SVE];
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
      Elem[result, e, esize] = Zeros(esize);
  Z[t, VL] = result;
```
C8.2.225   LD1D (vector plus immediate)

Gather load of doublewords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

Encoding

LD1D { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

Assembler symbols

<Zt>       Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>       Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>       Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>      Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
for e = 0 to elements-1
if Elem[mask, e, esize] == '1' then
  bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
  data = Mem[addr, mbytes, AccType_SVE];
  Elem[result, e, esize] = Extend(data, esize, unsigned);
else
  Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.226   LD1H (scalar plus immediate)

Contiguous load of unsigned halfwords to elements of a vector register from the memory address generated by a
64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size,
irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device
memory or signal a fault, and are set to zero in the destination vector.

16-bit element

Encoding
LD1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

32-bit element

Encoding
LD1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>22 21 20 19</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 1 1 0</td>
<td>imm4</td>
<td>1 0 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
</tr>
</tbody>
</table>

dtype<3:1>
dtype<0>

Encoding

LDIH {<Zt>.D}, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrIntUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
else
  for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
      integer eoff = (offset * elements) + e;
bits(64) addr = base + eoff * mbytes;
data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
\[ \text{Elem}[\text{result}, e, \text{esize}] = \text{Zeros}(\text{esize}); \]

\[ Z[t, VL] = \text{result}; \]
C8.2.227  LD1H (scalar plus scalar)

Contiguous load of unsigned halfwords to elements of a vector register from the memory address generated by a
64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access
the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from
Device memory or signal a fault, and are set to zero in the destination vector.

16-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

dtype<3:1>  
dtype<0>

**Encoding**

LD1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 16;
boolean unsigned = TRUE;

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

dtype<3:1>  
dtype<0>

**Encoding**

LD1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
64-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 0 1 1 0 0 1 0</td>
<td>Rm</td>
<td>0 1 0</td>
<td>Pg</td>
<td>Rn</td>
</tr>
</tbody>
</table>
```

dtype<3:1>  
dtype<0>

Encoding

LD1H { \langle Zt \rangle , \langle Pg \rangle /Z, \langle Xn|SP \rangle, \langle Xm \rangle, LSL #1}

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
```

Assembler symbols

- \langle Zt \rangle: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- \langle Pg \rangle: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- \langle Xn|SP \rangle: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- \langle Xm \rangle: Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation for all encodings

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
else
```

Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.228  LD1H (scalar plus vector)

Gather load of unsigned halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit scaled offset

\[
\begin{array}{cccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
\end{array}
\]

Encoding

LD1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;

32-bit unpacked scaled offset

\[
\begin{array}{cccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccc}
1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
\end{array}
\]

Encoding

LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
### 32-bit unpacked unscaled offset

| [31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 13 12| 10 9 | 5 4 | 0 |
|----------------|----------------|-----|-----|-----|
| 1 1 0 0 0 1 0 0 1 | Zm 0 1 0 | Pg 0 | Rn 0 | Zt 0 |

**Encoding**

LD1H \( \{<Zt>.D\}, \langle Pg \rangle / Z, [<Xn|SP>, <Zm>.D, <mod>] \)

**Decode for this encoding**

```plaintext
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 32-bit unscaled offset

| [31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 13 12| 10 9 | 5 4 | 0 |
|----------------|----------------|-----|-----|-----|
| 1 0 0 0 0 1 0 0 1 | Zm 0 1 0 | Pg 0 | Rn 0 | Zt 0 |

**Encoding**

LD1H \( \{<Zt>.S\}, \langle Pg \rangle / Z, [<Xn|SP>, <Zm>.S, <mod>] \)

**Decode for this encoding**

```plaintext
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 64-bit scaled offset

| [31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 13 12| 10 9 | 5 4 | 0 |
|----------------|----------------|-----|-----|-----|
| 1 1 0 0 0 1 0 0 1 1 | Zm 1 1 0 | Pg 0 | Rn 0 | Zt 0 |
**Encoding**

LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

**Decode for this encoding**

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 1;

**64-bit unscaled offset**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9 5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

**Encoding**

LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

**Decode for this encoding**

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

**Assembler symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

- UXTW when xs = 0
- SXTW when xs = 1
Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 then CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.229   LD1H (vector plus immediate)

Gather load of unsigned halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

```
1 0 0 0 0 1 0 1 0 1 1 1 0
imm5 Pg Zn Zt
```

Encoding

LD1H { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

64-bit element

```
1 1 0 0 0 1 0 1 0 1 1 1 0
imm5 Pg Zn Zt
```

Encoding

LD1H { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

Assembler symbols

<Zt>   Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn>

Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm>

Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

**Operation for all encodings**

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```
C8.2.230  LD1RB

Load a single unsigned byte from a memory address generated by a 64-bit scalar base address plus an immediate offset which is in the range 0 to 63.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

8-bit element

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24</th>
<th>23 22 21</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 1</td>
<td>imm6 1 0 0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

dtypeh<1>               | dtypeh<0>                        
dtypeh<0>                   | dtype<1>                        

Encoding

LD1RB {<Zt>.B}, <Pg>/Z, [<Xn|SP>{, #<imm>}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

16-bit element

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24</th>
<th>23 22 21</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 1</td>
<td>imm6 1 0 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

dtypeh<1>               | dtypeh<0>                        
dtypeh<0>                   | dtype<1>                        

Encoding

LD1RB {<Zt>.H}, <Pg>/Z, [<Xn|SP>{, #<imm>}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 0 1</td>
<td>imm6</td>
<td>1 1</td>
<td>0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
</tr>
</tbody>
</table>

dtypeh<1> | dtypeh<0> | dtype<0> | dtype<1>

Encoding

LD1RB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #imm}]}

Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 0 1</td>
<td>imm6</td>
<td>1 1 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

dtypeh<1> | dtypeh<0> | dtype<0> | dtype<1>

Encoding

LD1RB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #imm}]}

Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

Assembler symbols

- `<Zt>` is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` is the optional unsigned immediate byte offset, in the range 0 to 63, defaulting to 0, encoded in the "imm6" field.
Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    bits(64) addr = base + offset * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];
  for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
      Elem[result, e, esize] = Zeros(esize);
  Z[t, VL] = result;
C8.2.231   LD1RD

Load a single doubleword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 8 in the range 0 to 504.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

| 31 30 29 28|27 26 25 24|23 22 21 | 16|15 14 13|12| 10  9 | 5  4 | 0 |
|-----------|-----------|-----------|-----|-------|-----|-------|-----|-----|-----|
| 1 0 0 0 0|1 0 1 1|imm6|1 1|Pg|Rn|Zt|

dtypeh<1> dypeh<1>
dtypeh<0>   dypeh<0>

don'type<0>
don'type<1>

Encoding
LD1RD { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

Assembler symbols
<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 504, defaulting to 0, encoded in the "imm6" field.

Operation
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
bits(64) addr = base + offset * mbytes;
data = Mem[addr, mbytes, AccType_SVE];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.232  LD1RH

Load a single unsigned halfword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 2 in the range 0 to 126.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

### 16-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0</td>
<td>1 0 1</td>
<td>imm6</td>
<td>1 0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dtypeh<1>  
dtypeh<0>  
dtype<0>  
dtype<1>

#### Encoding

LD1RH { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

#### Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

### 32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0</td>
<td>1 0 0 1</td>
<td>imm6</td>
<td>1 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dtypeh<1>  
dtypeh<0>  
dtype<0>  
dtype<1>

#### Encoding

LD1RH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

#### Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1</td>
<td>0 1 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

LD1RH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;

integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 126, defaulting to 0, encoded in the "imm6" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    bits(64) addr = base + offset * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);
Z[t, VL] = result;
C8.2.233   LD1ROB (scalar plus immediate)

Load thirty-two contiguous bytes to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first thirty-two predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE
(FEAT_F64MM)

Encoding

LD1ROB { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n ! = 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset ∙ elements) + e;
    bits(64) addr = base + eoff ∙ mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
C8.2.234   LD1ROB (scalar plus scalar)

Load thirty-two contiguous bytes to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first thirty-two predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE

(FEAT_F64MM)

Encoding

LD1ROB { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = UInt(offset) + e;
    bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
C8.2.235  LD1ROD (scalar plus immediate)

Load four contiguous doublewords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first four predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE

(FEAT_F64MM)

<table>
<thead>
<tr>
<th>[31 30 29 28]27 26 25 24 23 22 21 20 19 16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 1 0 0 1</td>
<td>imm4</td>
<td>0 0 1</td>
<td>Pg</td>
</tr>
</tbody>
</table>

Encoding

LD1ROD { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}] |

Decode for this encoding

if !HaveSVE() || !HaveSVEFP64MatMultExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);

Assembler symbols

<Zt>      Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>      Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>   Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>     Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
C8.2.236   LD1ROD (scalar plus scalar)

Load four contiguous doublewords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 8 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first four predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE

(FEAT_F64MM)

Encoding

LD1ROD { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

Decode for this encoding

if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
in integer t = UInt(Zt);
in integer n = UInt(Rn);
in integer m = UInt(Rm);
in integer g = UInt(Pg);
constant integer esize = 64;

Assembler symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = UInt(offset) + e;
    bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
C8.2.237  LD1ROH (scalar plus immediate)

Load sixteen contiguous halfwords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

**SVE**

(FEAT_F64MM)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 0 1 0 1 0 imm4</td>
<td>0 0 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

LD1ROH { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]  

**Decode for this encoding**

if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;  
integer t = UInt(Zt);  
integer n = UInt(Rn);  
integer g =UInt(Pg);  
constant integer esize = 16;  
integer offset = SInt(imm4);

**Assembler symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

**Operation**

CheckNonStreamingSVEEnabled();  
constant integer VL = CurrentVL;  
constant integer PL = VL DIV 8;  
if VL < 256 then UNDEFINED;  
constant integer elements = 256 DIV esize;  
bits(64) base;  
bits(PL) mask = P[g, PL]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrunPredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
C8.2.238 LD1ROH (scalar plus scalar)

Load sixteen contiguous halfwords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 2 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE

(FEAT_F64MM)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 1 0 1 0 1 0 0 0</td>
<td>Rm</td>
<td>0 0 0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```  

**Encoding**

LD1ROH { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

**Decode for this encoding**

if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;

**Assembler symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation**

CheckNonStreamingSVEEnabled();
count constant integer VL = CurrentVL;
count constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
count constant integer elements = 256 DIV esize;
bits(64) base;
b设立(PL) mask = P[g, PL]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstratUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n, 64];
  offset = X[m, 64];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = UInt(offset) + e;
    bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
C8.2.239  **LD1ROW (scalar plus immediate)**

Load eight contiguous words to elements of a 256-bit (octaword) vector from the memory address generated by a
64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base
address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current
vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing
bits in the destination vector are set to zero.

Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and
enabled at the current Exception level.

**SVE**

(FEAT_F64MM)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>imm4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>msz&lt;1&gt;</td>
<td>msz&lt;0&gt;</td>
<td>ssz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

LD1ROW { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

**Decode for this encoding**

if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);

**Assembler symbols**

- **<Zt>** Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<imm>** Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

**Operation**

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bull(64) base;
bull(PL) mask = P[g, PL]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
C8.2.240   LD1ROW (scalar plus scalar)

Load eight contiguous words to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 4 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE

(FEAT_F64MM)

Encoding

LD1ROW { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

Decode for this encoding

if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(UNpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = UInt(offset) + e;
    bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
C8.2.241   LD1RQB (scalar plus immediate)

Load sixteen contiguous bytes to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

Encoding

LD1RQB { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
   if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
   else
      if n == 31 then CheckSPAlignment();
      base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = base + (offset * 16) + (e * mbytes);
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);
    end if
end for

Z[t, VL] = Replicate(result, VL DIV 128);
C8.2.242   **LD1RQB (scalar plus scalar)**

Load sixteen contiguous bytes to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

**Encoding**

LD1RQB { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;

**Assembler symbols**

<Zt>    Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>    Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>    Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(64) offset;
b[128] result;
constant integer mbytes = esize DIV 8;
if !AnyActiveElement(mask, esize) then SetTagCheckedInstruction(TRUE);
if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n, 64];
  offset = X[m, 64];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

    Z[t, VL] = Replicate(result, VL DIV 128);
C8.2.243 LD1RQD (scalar plus immediate)

Load two contiguous doublewords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

Encoding

LD1RQD { Zt }.D, Pg/Z, \[Xn|SP\>{, #imm}\]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
for e = 0 to elements-1
    if Elemp[mask, e, esize] == '1' then
        bits(64) addr = base + (offset * 16) + (e * mbytes);
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);
    end

Z[t, VL] = Replicate(result, VL DIV 128);
LD1RQD (scalar plus scalar)

Load two contiguous doublewords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 8 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

Encoding

LD1RQD { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);
    end if
end for

Z[t, VL] = Replicate(result, VL DIV 128);
C8.2.245  LD1RQH (scalar plus immediate)

Load eight contiguous halfwords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

Encoding

LD1RQH { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(64) addr = base + (offset * 16) + (e * mbytes);
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros(esize);

  Z[t, VL] = Replicate(result, VL DIV 128);
C8.2.246   LD1RQH (scalar plus scalar)

Load eight contiguous halfwords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 2 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

Encoding

```
LD1RQH { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
```

Assembler symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
```

ARM DDI 0487I.a Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. C8-3475
ID081822 Non-Confidential
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);
    Z[t, VL] = Replicate(result, VL DIV 128);
C8.2.247   LD1RQW (scalar plus immediate)

Load four contiguous words to elements of a short, 128-bit (quadword) vector from the memory address generated
by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the
base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short
vector is then replicated to fill the long destination vector. Only the first four predicate elements are used and higher
numbered predicate elements are ignored.

Encoding

LD1RQW { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
basis(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = base + (offset * 16) + (e * mbytes);
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

    Z[t, VL] = Replicate(result, VL DIV 128);
C8.2.248  **LD1RQW (scalar plus scalar)**

Load four contiguous words to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 4 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first four predicate elements are used and higher numbered predicate elements are ignored.

**Encoding**

`LD1RQW { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]`

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
```

**Assembler symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 then SetTagCheckedInstruction(TRUE);
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP else X[n, 64];
  offset = X[m, 64];
```
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
C8.2.249   LD1RSB

Load a single signed byte from a memory address generated by a 64-bit scalar base address plus an immediate offset which is in the range 0 to 63.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

16-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 16 | 15 | 14 | 13 | 12 | 10 | 9  | 5  | 4  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | imm6 | 1 | 1 | 0 | Pg | Rn | Zt |

dtypeh<1>
dtypeh<0>
dype1<0>
dype1<1>

Encoding

LD1RSB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm6);

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 16 | 15 | 14 | 13 | 12 | 10 | 9  | 5  | 4  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | imm6 | 1 | 0 | 1 | Pg | Rn | Zt |

dtypeh<1>
dtypeh<0>
dype1<0>
dype1<1>

Encoding

LD1RSB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21</th>
<th>16 15 14 13</th>
<th>12 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 1 1</td>
<td>1 0 0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

LDIRSBS \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>{, #<imm>}]\

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm6);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional unsigned immediate byte offset, in the range 0 to 63, defaulting to 0, encoded in the "imm6" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n, 64];
bits(64) addr = base + offset * mbytes;
data = Mem[addr, mbytes, AccType_SVE];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);
Z[t, VL] = result;
C8.2.250  LD1RSH

Load a single signed halfword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 2 in the range 0 to 126.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

### 32-bit element

```
[31 30 29 28|27 26 25 24|23 22 21 | 16|15 14 13 12| 10 9 | 5 4 | 0]
1 0 0 0 0 1 0 1 | imm6 1 0 1 | Pg Rn Zt
```

**Encoding**

LD1RSH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm6);

### 64-bit element

```
[31 30 29 28|27 26 25 24|23 22 21 | 16|15 14 13 12| 10 9 | 5 4 | 0]
1 0 0 0 0 1 0 1 | imm6 1 0 0 | Pg Rn Zt
```

**Encoding**

LD1RSH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm6);

**Assembler symbols**

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 126, defaulting to 0, encoded in the "imm6" field.

**Operation for all encodings**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        bits(64) addr = base + offset * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);
Z[t, VL] = result;
```

C8.2.251   LD1RSW

Load a single signed word from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 4 in the range 0 to 252.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

Encoding

LD1RSW {<Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]  

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
integer offset = UInt(imm6);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 252, defaulting to 0, encoded in the "imm6" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
bits(64) addr = base + offset * mbytes;
data = Mem[addr, mbytes, AccType_SVE];

for e = 0 to elements-1
if ElemP[mask, e, esize] == '1' then
   Elem[result, e, esize] = Extend(data, esize, unsigned);
else
   Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.252   LD1RW

Load a single unsigned word from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 4 in the range 0 to 252.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

32-bit element

Encoding

LD1RW { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm);

64-bit element

Encoding

LD1RW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 252, defaulting to 0, encoded in the "imm6" field.

**Operation for all encodings**

```c
CheckSVEEnabled();
c
constant integer VL = CurrentVL;
c
constant integer PL = VL DIV 8;
c
constant integer elements = VL DIV esize;
c
bits(64) base;
c
bits(PL) mask = P[g, PL];
c
bits(VL) result;
c
bits(msize) data;
c
constant integer mbytes = msize DIV 8;
c
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
c
if !AnyActiveElement(mask, esize) then
    if n == 31 & ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
c
    else
        if n == 31 then CheckSPAlignment();
c
        base = if n == 31 then SP[] else X[n, 64];
c
        bits(64) addr = base + offset * mbytes;
c
        data = Mem[addr, mbytes, AccType_SVE];
c
        for e = 0 to elements-1
            if ElemP[mask, e, esize] == '1' then
                Elem[result, e, esize] = Extend(data, esize, unsigned);
c
            else
                Elem[result, e, esize] = Zeros(esize);
c
        Z[t, VL] = result;
```
C8.2.253   LD1SB (scalar plus immediate)

Contiguous load of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

16-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>19 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 0 1 0 1 1 1 0 0</td>
<td>imm4</td>
<td>1 0 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dtype<3:1> __________
dtype<0> __________

Encoding
LD1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>19 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 0 1 1 1 0 0</td>
<td>imm4</td>
<td>1 0 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dtype<3:1> __________
dtype<0> __________

Encoding
LD1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>19 16</th>
<th>15 14 13 12</th>
<th>10 9 5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0 1 1 0 0 0</td>
<td>imm4</td>
<td>1 0 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
</tr>
</tbody>
</table>

dtype<3:1>
dtype<0>

Encoding

LDISB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
  for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
      integer eoff = (offset * elements) + e;
bits(64) addr = base + eoff * mbytes;
data = Mem[addr, mbytes, AccType_SVE];
Elem[result, e, esize] = Extend(data, esize, unsigned);
else


\[ \text{Elem}[\text{result}, e, \text{esize}] = \text{Zeros} (\text{esize}); \]

\[ Z[t, VL] = \text{result}; \]
C8.2.254   LD1SB (scalar plus scalar)

Contiguous load of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

16-bit element

\[
\begin{array}{cccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & | & 0 & 1 & 0 & | & Pg & | & Rn & | & Zt
\end{array}
\]

dtype<3:1>  
dtype<0>

**Encoding**

LD1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>]

**Decode for this encoding**

\[
\begin{align*}
\text{if } & !\text{HaveSVE() } \&\& !\text{HaveSME() } \text{ then UNDEFINED;} \\
\text{if } & \text{Rm == '11111' then UNDEFINED;} \\
\text{integer } t & = \text{UInt(Zt);} \\
\text{integer } n & = \text{UInt(Rn);} \\
\text{integer } m & = \text{UInt(Rm);} \\
\text{integer } g & = \text{UInt(Pg)}; \\
\text{constant integer esize} & = 16; \\
\text{constant integer msize} & = 8; \\
\text{boolean unsigned} & = \text{FALSE};
\end{align*}
\]

32-bit element

\[
\begin{array}{cccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & | & 0 & 1 & 0 & | & Pg & | & Rn & | & Zt
\end{array}
\]

dtype<3:1>  
dtype<0>

**Encoding**

LD1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>]

**Decode for this encoding**

\[
\begin{align*}
\text{if } & !\text{HaveSVE() } \&\& !\text{HaveSME() } \text{ then UNDEFINED;} \\
\text{if } & \text{Rm == '11111' then UNDEFINED;} \\
\text{integer } t & = \text{UInt(Zt);} \\
\text{integer } n & = \text{UInt(Rn);} \\
\text{integer } m & = \text{UInt(Rm);} \\
\text{integer } g & = \text{UInt(Pg)}; \\
\text{constant integer esize} & = 32; \\
\text{constant integer msize} & = 8; \\
\text{boolean unsigned} & = \text{FALSE};
\end{align*}
\]
64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 22 | 21 | 20 | 16 | 15 | 14 | 13 | 12 | 10 | 9 | 5 | 4 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | Rm | 0  | 1  | 0  | Pg | Rn | Zt |

dtype<3:1>  dtype<0>

Encoding

LDISB ( <Zt>.D ), <Pg>/Z, [<Xn|SP>, <Xm>]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n, 64];
  offset = X[m, 64];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
data = Mem[addr, mbytes, AccType_SVE];
  Elem[result, e, esize] = Extend(data, esize, unsigned);
else
\texttt{Elem[result, e, esize] = Zeros(esize);}

\texttt{Z[t, VL] = result;}
C8.2.255   LD1SB (scalar plus vector)

Gather load of signed bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit unpacked unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9 5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>xs</td>
</tr>
<tr>
<td>Zm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Pg</td>
</tr>
<tr>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U ff

msz<1>
msz<0>

Encoding

LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9 5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>xs</td>
</tr>
<tr>
<td>Zm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Pg</td>
</tr>
<tr>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U ff

Encoding

LD1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
64-bit unscaled offset

Encoding

LD1SB \{<Zt>.D\}, <Pg>/Z, [<Xn|SP>, <Zm>.D]

Decode for this encoding

if !HaveSVE() then UNDEFINED;

integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm>  Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod>  Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

  UXTW     when xs = 0
  SXTW     when xs = 1

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSAlignment();
  else
    if n == 31 then CheckSAlignment();
    base = if n == 31 then SP[] else X[n, 64];
offset = Z[m, VL];

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1'
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.256 LD1SB (vector plus immediate)

Gather load of signed bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

```
|31|30|29|28|27|26|25|24|23|22|21|20|16|15|14|13|12|10|  9 |  5 |  4 |   0 |
|1  | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | imm5| 1 | 0 | 0 | Pg | Zn | Zt |
```

Encoding

LDISB {<Zt>.S}, <Pg>/Z, [<Zn>.S{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

64-bit element

```
|31|30|29|28|27|26|25|24|23|22|21|20|16|15|14|13|12|10|  9 |  5 |  4 |   0 |
|1  | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | imm5| 1 | 0 | 0 | Pg | Zn | Zt |
```

Encoding

LDISB {<Zt>.D}, <Pg>/Z, [<Zn>.D{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.257  LD1SH (scalar plus immediate)

Contiguous load of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

32-bit element

| 31 30 29 28| 27 26 25 24 | 22 21 20|19  | 16|15 14 13 12| 10 9 | 5 4 | 0 |
| 1 0 1 0 0 1 0 1 0 0 0 1 0 | imm4 | 1 0 1 | Pg | Rn | Zt |

dtype<3:1>  
dtype<0>  

Encoding

LDISH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

64-bit element

| 31 30 29 28| 27 26 25 24 | 22 21 20|19  | 16|15 14 13 12| 10 9 | 5 4 | 0 |
| 1 0 1 0 0 1 0 1 0 0 0 1 0 | imm4 | 1 0 1 | Pg | Rn | Zt |

dtype<3:1>  
dtype<0>  

Encoding

LDISH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

Assembler symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
Operation for all encodings

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
    for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
bits(64) addr = base + eoff * mbytes;
data = Mem[addr, mbytes, AccType_SVE];
Elem[result, e, esize] = Extend(data, esize, unsigned);
else
    Elem[result, e, esize] = Zeros(esize);
    Z[t, VL] = result;
```
C8.2.258 LD1SH (scalar plus scalar)

Contiguous load of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 0 0 1</td>
<td>Rm 0 1 0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dtype<3:1> 
dtype<0> 

Encoding

LDISH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;

64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 0 0 0 0</td>
<td>Rm 0 1 0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dtype<3:1> 
dtype<0> 

Encoding

LDISH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation for all encodings**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
data = Mem[addr, mbytes, AccType_SVE];
Elem[result, e, esize] = Extend(data, esize, unsigned);
else
  Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```
C8.2.259   LD1SH (scalar plus vector)

Gather load of signed halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit scaled offset

\[
\begin{array}{cccccccccccc}
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & Xs & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Pz & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Zt & 1 \\
\end{array}
\]

**Encoding**

LDISH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]

**Decode for this encoding**

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;

32-bit unpacked scaled offset

\[
\begin{array}{cccccccccccc}
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & Xs & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Pz & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Zt & 1 \\
\end{array}
\]

**Encoding**

LDISH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]

**Decode for this encoding**

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
32-bit unpacked unscaled offset

```
[31 30 29 28 27 26 25 24 23 22 21 20] 16 15 14 13 12 10 9 5 4 0
1 1 0 0 0 1 0 0 1 1 xs 0 Zm 0 0 0 Pg Rn Zt
```

Encoding

LDISH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

32-bit unscaled offset

```
[31 30 29 28 27 26 25 24 23 22 21 20] 16 15 14 13 12 10 9 5 4 0
1 0 0 0 0 1 0 0 1 1 xs 0 Zm 0 0 0 Pg Rn Zt
```

Encoding

LDISH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

64-bit scaled offset

```
[31 30 29 28 27 26 25 24 23 22 21 20] 16 15 14 13 12 10 9 5 4 0
1 1 0 0 0 1 0 0 1 1 1 Zm 1 0 0 Pg Rn Zt
```
Encoding

LDISH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 1;

64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 1 1 0</td>
<td>Zm 1 0 0</td>
<td>Pg</td>
<td>Rn</td>
</tr>
</tbody>
</table>

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:
- UXTW when xs = 0
- SXTW when xs = 1
Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    bits(64) addr = base + (off << scale);
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);
Z[t, VL] = result;
C8.2.260  LD1SH (vector plus immediate)

Gather load of signed halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>15 14 13 12</th>
<th>10  9</th>
<th>5  4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0</td>
<td>1 0 1 0 1</td>
<td>imm5</td>
<td>1 0 0</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

U ff

Encoding

LDISH { <Zt>.S }, <Pg>/Z, [Zn].S{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>15 14 13 12</th>
<th>10  9</th>
<th>5  4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0</td>
<td>1 0 1 0 1</td>
<td>imm5</td>
<td>1 0 0</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

U ff

Encoding

LDISH { <Zt>.D }, <Pg>/Z, [Zn].D{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

**Operation for all encodings**

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
data = Mem[addr, mbytes, AccType_SVE];
Elem[result, e, esize] = Extend(data, esize, unsigned);
else
    Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.261   LD1SW (scalar plus immediate)

Contiguous load of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

Encoding

LD1SW \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bias(PL) mask = P[g, PL];
bias(VL) result;
bias(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.262   LD1SW (scalar plus scalar)

Contiguous load of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

Encoding

LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
   if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
   else
      if n == 31 then CheckSPAlignment();

base = if n == 31 then SP[] else X[n, 64];
offset = X[m, 64];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.263   **LD1SW (scalar plus vector)**

Gather load of signed words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

### 32-bit unpacked scaled offset

![Binary representation](image)

**Encoding**

LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

**Decode for this encoding**

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 2;

### 32-bit unpacked unscaled offset

![Binary representation](image)

**Encoding**

LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

**Decode for this encoding**

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
**64-bit scaled offset**

```
[31 30 29 28][27 26 25 24][23 22 21 20]  16|15 14 13 12|  10 9 |  5 4 |  0 |
1 1 0 0 0 1 0 1 0 1 1 Zm 1 0 0 Pg Rn Zt
```

**Encoding**

LDISW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

**Decode for this encoding**

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 2;
```

**64-bit unscaled offset**

```
[31 30 29 28][27 26 25 24][23 22 21 20]  16|15 14 13 12|  10 9 |  5 4 |  0 |
1 1 0 0 0 1 0 1 0 1 0 Zm 1 0 0 Pg Rn Zt
```

**Encoding**

LDISW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

**Decode for this encoding**

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

**Assembler symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Zm>` Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

- **UXTW** when **xs = 0**
- **SXTW** when **xs = 1**

**Operation for all encodings**

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];
  end if;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    bits(64) addr = base + (off << scale);
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);
  end if;
end for;
Z[t, VL] = result;
```
C8.2.264   LD1SW (vector plus immediate)

Gather load of signed words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

Encoding

LD1SW { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

Assembler symbols

<Zt>         Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>         Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>         Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>        Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
   base = Z[n, VL];
for e = 0 to elements-1
if \( \text{Elem}[	ext{mask, e, esize}] = '1' \) then
    \[
    \text{bits(64) addr} = \text{ZeroExtend}(	ext{Elem}[\text{base, e, esize}], 64) + \text{offset} \times \text{mbytes};
    \]
    \[
    \text{data} = \text{Mem}[\text{addr, mbytes, AccType_SVE}];
    \]
    \[
    \text{Elem}[\text{result, e, esize}] = \text{Extend}(\text{data, esize, unsigned});
    \]
else
    \[
    \text{Elem}[\text{result, e, esize}] = \text{Zeros}(\text{esize});
    \]

\[Z[t, VL] = \text{result};\]
C8.2.265  LD1W (scalar plus immediate)

Contiguous load of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

**32-bit element**

```
[31 30 29 28|27 26 25 24| 22 21 20|19 16|15 14 13 12| 10 9 | 5 4 | 0 ]
1 0 1 0 0 1 0 1 0 1 0 0 imm4 1 0 1 Pg Rn Zt

dtype<3:1>  
dtype<0>
```

**Encoding**

LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

**Decode for this encoding**

```java
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

**64-bit element**

```
[31 30 29 28|27 26 25 24| 22 21 20|19 16|15 14 13 12| 10 9 | 5 4 | 0 ]
1 0 1 0 0 1 0 1 0 1 0 1 imm4 1 0 1 Pg Rn Zt

dtype<3:1>  
dtype<0>
```

**Encoding**

LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

**Decode for this encoding**

```java
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

**Assembler symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

**Operation for all encodings**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```
C8.2.266  LD1W (scalar plus scalar)

Contiguous load of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

32-bit element

```plaintext
LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

**Encoding**

LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

**Decode for this encoding**

```plaintext
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
```

64-bit element

```plaintext
LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

**Encoding**

LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

**Decode for this encoding**

```plaintext
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
```

**Assembler symbols**

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation for all encodings**

```c
CheckSVEEnabled();
c
constant integer VL = CurrentVL;
c
constant integer PL = VL DIV 8;
c
constant integer elements = VL DIV esize;
c
bits(64) base;
c
bits(PL) mask = P[g, PL];
c
bits(VL) result;
c
bits(msize) data;
c
bits(64) offset;
c
constant integer mbytes = msize DIV 8;
c
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
c
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
c
else
  if n == 31 then CheckSPAlignment();
c
  base = if n == 31 then SP[] else X[n, 64];
c
  offset = X[m, 64];
c
  for e = 0 to elements-1
    if Elemp[mask, e, esize] == '1' then
      bits(64) addr = base + (UInt(offset) + e) * mbytes;
c
      data = Mem[addr, mbytes, AccType_SVE];
c
      Elem[result, e, esize] = Extend(data, esize, unsigned);
c
    else
      Elem[result, e, esize] = Zeros(esize);
c
Z[t, VL] = result;
```
C8.2.267   LD1W (scalar plus vector)

Gather load of unsigned words to active elements of a vector register from memory addresses generated by a 64-bit
scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then
optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set
to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and
enabled at the current Exception level.

32-bit scaled offset

<table>
<thead>
<tr>
<th></th>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 0 0 0 0 1 0 1 0</td>
<td>xs 1</td>
<td>Zm</td>
<td>0 1 0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
</tr>
<tr>
<td>encoding</td>
<td>U ff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #2]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;

32-bit unpacked scaled offset

<table>
<thead>
<tr>
<th></th>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 1 0 0 0 1 0 1 0</td>
<td>xs 1</td>
<td>Zm</td>
<td>0 1 0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
</tr>
<tr>
<td>encoding</td>
<td>U ff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;
32-bit unpacked unscaled offset

LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled offset

LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
**Encoding**

LDIW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

**Decode for this encoding**

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 2;

64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13</th>
<th>12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 1</td>
<td>0 1 1 0</td>
<td>Zm</td>
<td>1 1 0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

LDIW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

**Decode for this encoding**

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

**Assembler symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

- UXTW when xs = 0
- SXTW when xs = 1
Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    bits(64) addr = base + (off << scale);
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.268   LD1W (vector plus immediate)

Gather load of unsigned words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

Encoding
LD1W { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

Decode for this encoding
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

64-bit element

Encoding
LD1W { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

Decode for this encoding
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

Assembler symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pp> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

**Operation for all encodings**

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

C8.2.269   LD2B (scalar plus immediate)

Contiguous load two-byte structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

Encoding

LD2B { <Zt1>.B, <Zt2>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 2;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
if n == 31 &\& ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements * nreg) + (e * nreg) + r;
        bits(64) addr = base + eoff * mbytes;
        Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
C8.2.270   LD2B (scalar plus scalar)

Contiguous load two-byte structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

Encoding

LD2B { <Zt1>.B, <Zt2>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 2;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
  end
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros(esize);
    end
  end
for r = 0 to nreg-1
  Z[(t+r) MOD 32, VL] = values[r];
C8.2.271 LD2D (scalar plus immediate)

Contiguous load two-doubleword structures, each to the same element number in two vector registers from the
memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16
to 14 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to
the two consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read
from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination
vector registers.

Encoding

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>imm4</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
msz<1>
msz<0>
```

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bases64 base;
basis(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
```
if n == 31 &
ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

    for r = 0 to nreg-1
        Z[(t+r) MOD 32, VL] = values[r];
C8.2.272    LD2D (scalar plus scalar)

Contiguous load two-doubleword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

Encoding

LD2D {<Zt1>.D, <Zt2>.D}, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 2;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainingUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros(esize);

  for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
C8.2.273   LD2H (scalar plus immediate)

Contiguous load two-halfword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

Encoding

LD2H { <Zt1>.H, <Zt2>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 2;

Assembler symbols

<Zt1>   Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>   Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>    Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>   Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
if n == 31 & ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
  CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros(esize);
  
for r = 0 to nreg-1
  Z[(t+r) MOD 32, VL] = values[r];
C8.2.274   LD2H (scalar plus scalar)

Contiguous load two-halfword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

Encoding

LD2H { <Zt1>.H, <Zt2>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 2;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

  for e = 0 to elements-1
      for r = 0 to nreg-1
          if ElemP[mask, e, esize] == '1' then
              integer eoff = UInt(offset) + (e * nreg) + r;
              bits(64) addr = base + eoff * mbytes;
              Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
          else
              Elem[values[r], e, esize] = Zeros(esize);

  for r = 0 to nreg-1
      Z[(t+r) MOD 32, VL] = values[r];
C8.2.275   LD2W (scalar plus immediate)

Contiguous load two-word structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

Encoding

LD2W { <Zt1>.S, <Zt2>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 2;

Assembler symbols

<Zt1>     Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>     Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>      Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>   Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>     Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
if n == 31 & ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
  CheckSPLignment();
else
  if n == 31 then CheckSPLignment();
  base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros(esize);
  
  for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
C8.2.276   LD2W (scalar plus scalar)

Contiguous load two-word structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

Encoding

LD2W { <Zt1>.S, <Zt2>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 2;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
count integer VL = CurrentVL;
count integer PL = VL DIV 8;
count integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, Pl];
bits(64) offset;
count integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

    for e = 0 to elements-1
        for r = 0 to nreg-1
            if ElemP[mask, e, esize] == '1' then
                integer eoff = UInt(offset) + (e * nreg) + r;
                bits(64) addr = base + eoff * mbytes;
                Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
            else
                Elem[values[r], e, esize] = Zeros(esize);

        for r = 0 to nreg-1
            Z[(t+r) MOD 32, VL] = values[r];

C8.2.277   LD3B (scalar plus immediate)

Contiguous load three-byte structures, each to the same element number in three vector registers from the memory
address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that
is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to
the three consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from
Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector
registers.

Encoding


Decoding

if !HaveSVE() && !HaveSME() then UNDEFINED;

integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 3;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to
0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, Pl];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPIalignment();
    else
        if n == 31 then CheckSPIalignment();
        base = if n == 31 then SP[] else X[n, 64];

    for e = 0 to elements-1
        for r = 0 to nreg-1
            ifElemP[mask, e, esize] == '1' then
                integer eoff = (offset * elements + nreg) + (e * nreg + r);
                bits(64) addr = base + eoff * mbytes;
                Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
            else
                Elem[values[r], e, esize] = Zeros(esize);

        for r = 0 to nreg-1
            Z[(t+r) MOD 32, VL] = values[r];
C8.2.278   LD3B (scalar plus scalar)

Contiguous load three-byte structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

Encoding

LD3B { <Zt1>.B, <Zt2>.B, <Zt3>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 3;

Assembler symbols

<Zt1>    Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>    Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>    Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>     Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>     Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bias(PL) mask = P[g, Pl];
bias(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if Elem[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
  Z[(t+r) MOD 32, VL] = values[r];
C8.2.279  LD3D (scalar plus immediate)

Contiguous load three-doubleword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector’s in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

Encoding

LD3D { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 3;

Assembler symbols

<Zt1>  Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>  Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>  Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>   Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>  Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
basis(64) base;
basis(PL) mask = P[g, Pl];
constant integer mbytes = esize DIV 8;
array [0..2] of basis(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 & ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAignment();
    else
        if n == 31 then CheckSPAignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
C8.2.280   LD3D (scalar plus scalar)

Contiguous load three-doubleword structures, each to the same element number in three vector registers from the
memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL
option) and added to the base address. After each structure access the index value is incremented by three. The index
register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to
the three consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read
from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination
vector registers.

Encoding

LD3D { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 3;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo
32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, Pl];
bits(64) offset;
countant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

    for e = 0 to elements-1
        for r = 0 to nreg-1
            if ElemP[mask, e, esize] == '1' then
                integer eoff = UInt(offset) + (e * nreg) + r;
                bits(64) addr = base + eoff * mbytes;
                Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
            else
                Elem[values[r], e, esize] = Zeros(esize);

            for r = 0 to nreg-1
                Z[(t+r) MOD 32, VL] = values[r];
C8.2.281   LD3H (scalar plus immediate)

Contiguous load three-halfword structures, each to the same element number in three vector registers from the
memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24
to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to
the three consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read
from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination
vector registers.

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 3;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo
32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to
0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
basis(64) base;
basis(PL) mask = P[g, Pl];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrunpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
C8.2.282  LD3H (scalar plus scalar)

Contiguous load three-halfword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

Encoding

LD3H { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 3;

Assembler symbols

<Zt1>  Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>  Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>  Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
banner(base);
banner(PL) mask = P[g, Pl];
banner(offset);
constant integer mbytes = esize DIV 8;
array [0..2] of banner(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
  Z[(t+r) MOD 32, VL] = values[r];
C8.2.283   LD3W (scalar plus immediate)

Contiguous load three-word structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 3;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, Pl];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
  Z[(t+r) MOD 32, VL] = values[r];
C8.2.284 LD3W (scalar plus scalar)

Contiguous load three-word structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

Encoding

LD3W { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 3;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bhits(64) base;
bhits(PL) mask = P[g, PL];
bhits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bhits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

for e = 0 to elements-1
for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + (e × nreg) + r;
        bits(64) addr = base + eoff × mbytes;
        Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
C8.2.285  LD4B (scalar plus immediate)

Contiguous load four-byte structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 4;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
  Z[(t+r) MOD 32, VL] = values[r];
C8.2.286   LD4B (scalar plus scalar)

Contiguous load four-byte structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 4;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainingUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];
for e = 0 to elements-1
    for r = 0 to nreg-1
        if Elem[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg + r);
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);
for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
C8.2.287   **LD4D (scalar plus immediate)**

Contiguous load four-doubleword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

**Encoding**


**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 4;

**Assembler symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Zt4>` Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements + nreg + (e * nreg) + r;
      bits(64) addr = base + eoff + mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros(esize);
  for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
C8.2.288   LD4D (scalar plus scalar)

Contiguous load four-doubleword structures, each to the same element number in four vector registers from the
memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL
option) and added to the base address. After each structure access the index value is incremented by four. The index
register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to
the four consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read
from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination
vector registers.

Encoding

| [31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 13 12| 10 9 | 5 4 | 0 |
|---------------------------------|----|------------|---|---|---|
| 1 0 1 0 0 1 0 1 1 1 1 | Rm | 1 1 0 | Pg | Rn | Zt |

msz<1>
msz<0>

Decoding for this encoding

\[
\begin{align*}
\text{if} & \text{ !HaveSVE() \&\& !HaveSME() then UNDEFINED; } \\
\text{if} & \text{ Rm == '11111' then UNDEFINED; } \\
\text{integer} & \text{ t = UInt(Zt); } \\
\text{integer} & \text{ n = UInt(Rn); } \\
\text{integer} & \text{ m = UInt(Rm); } \\
\text{integer} & \text{ g = UInt(Pg); } \\
\text{constant integer} & \text{ esize = 64; } \\
\text{constant integer} & \text{ nreg = 4; }
\end{align*}
\]

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAignment();
  else
    if n == 31 then CheckSPAignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros(esize);
  for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
C8.2.289   LD4H (scalar plus immediate)

Contiguous load four-halfword structures, each to the same element number in four vector registers from the
memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32
to 28 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to
the four consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read
from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination
vector registers.

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 4;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to
0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
basis(64) base;
basis(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
   if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
   else
      if n == 31 then CheckSPAlignment();
      base = if n == 31 then SP[] else X[n, 64];
for e = 0 to elements-1
   for r = 0 to nreg-1
      if ElemP[mask, e, esize] == '1' then
         integer eoff = (offset * elements * nreg) + (e * nreg) + r;
         bits(64) addr = base + eoff * mbytes;
         Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
      else
         Elem[values[r], e, esize] = Zeros(esize);
   for r = 0 to nreg-1
      Z[(t+r) MOD 32, VL] = values[r];
C8.2.290   LD4H (scalar plus scalar)

Contiguous load four-halfword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

```
1 0 1 0 0 1 0 1 1 | Rm 1 1 0 | Pg | Rn | Zt |

msz<1>
msz<0>
```

**Encoding**

```
```

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 4;
```

**Assembler symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Zt4>` Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
b its(64) base;
b its(PL) mask = P[g, PL];
b its(64) offset;
```
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros(esize);
  for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
C8.2.291  LD4W (scalar plus immediate)

Contiguous load four-word structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector’s in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 4;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
  Z[(t+r) MOD 32, VL] = values[r];
C8.2.292  LD4W (scalar plus scalar)

Contiguous load four-word structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 4;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
    for e = 0 to elements-1
      for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
          integer eoff = UInt(offset) + (e * nreg) + r;
          bits(64) addr = base + eoff * mbytes;
          Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
          Elem[values[r], e, esize] = Zeros(esize);
    for r = 0 to nreg-1
      Z[(t+r) MOD 32, VL] = values[r];
C8.2.293  **LDFF1B (scalar plus scalar)**

Contiguous load with first-faulting behavior of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

### 8-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0 0 0</td>
<td>Rm</td>
<td>0 1 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

dype<3:1>  
dtype<0>

**Encoding**

LDFF1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, <Xm}>]

**Decode for this encoding**

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer msize = 8;
boolean unsigned = TRUE;
```

### 16-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0 0 0 1</td>
<td>Rm</td>
<td>0 1 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

dype<3:1>  
dtype<0>

**Encoding**

LDFF1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, <Xm}>]

**Decode for this encoding**

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = TRUE;
```
32-bit element

LDFF1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>}]  

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;

64-bit element

LDFF1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>}]  

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 & ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
        else
            (data, fault) = (Zeros(msize), FALSE);
        // FFR elements set to FALSE following a suppressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR[e, esize] = '0';
        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elseif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);
    Z[t, VL] = result;
C8.2.294   LDFF1B (scalar plus vector)

Gather load with first-faulting behavior of unsigned bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit unpacked unscaled offset

| [31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 13 12| 10 9 | 5 4 | 0 |
| 1 1 0 0 0 1 0 0 0 xs 0 | Zm | 0 1 1 | Pg | Rn | Zt |

msz<1>
msz<0>

Encoding

LDFF1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

| [31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 13 12| 10 9 | 5 4 | 0 |
| 1 0 0 0 0 1 0 0 0 xs 0 | Zm | 0 1 1 | Pg | Rn | Zt |

Encoding

LDFF1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
### 64-bit unscaled offset

![Encoding diagram](image)

**Encoding**

LDFF1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

**Decode for this encoding**

```plaintext
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

**Assembler symbols**

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Zm>`: Is the name of the offset scalable vector register, encoded in the "Zm" field.
- `<mod>`: Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:
  - UXTW when `xs = 0`
  - SXTW when `xs = 1`

**Operation for all encodings**

```plaintext
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, Pl];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bias(VL) orig = Z[t, VL];
bias(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
```
if n == 31 &
ConstrainUnpredictableBool(Unpredictable_CheckSPNONEACTIVE) then
CheckSPAlignment();
else
if n == 31 then
CheckSPAlignment();
base = if n == 31 then SP[] else X[n, 64];
offset = Z[m, VL];
for e = 0 to elements-1
if Elem[mask, e, esize] == '1' then
integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
bits(64) addr = base + (off << scale);
if first then
// Mem[] will not return if a fault is detected for the first active element
data = Mem[addr, mbytes, AccType_SVE];
first = FALSE;
else
// MemNF[] will return fault=TRUE if access is not performed for any reason
(data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
else
(data, fault) = (Zeros(msize), FALSE);
// FFR elements set to FALSE following a suppressed access/fault
faulted = faulted || fault;
if faulted then
ElemFFR[e, esize] = '0';
// Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
unknown = unknown || ElemFFR[e, esize] == '0';
if unknown then
if !fault &
ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
Elem[result, e, esize] = Extend(data, esize, unsigned);
elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
Elem[result, e, esize] = Zeros(esize);
else // merge
Elem[result, e, esize] = Elem[orig, e, esize];
else
Elem[result, e, esize] = Extend(data, esize, unsigned);
Z[t, VL] = result;
C8.2.295   LDFF1B (vector plus immediate)

Gather load with first-faulting behavior of unsigned bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 0 1</td>
<td>imm5</td>
<td>1 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

Encoding

LDFF1B { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0 0 1</td>
<td>imm5</td>
<td>1 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

Encoding

LDFF1B { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

**Operation for all encodings**

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bis(VL) orig = Z[t, VL];
bis(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        else
            (data, fault) = (Zeros(msize), FALSE);
        // FFR elements set to FALSE following a suppressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR[e, esize] = '0';
        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR[e, esize] == '0';
        if unknown then
            if !fault & ConstrinUnpredictableBool(Unpredictable_SVELDNFDATA) then
                Elem[result, e, esize] = Extend(data, esize, unsigned);
            elsif ConstrinUnpredictableBool(Unpredictable_SVELDNFZERO) then
                Elem[result, e, esize] = Zeros(esize);
            else // merge
                Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        Z[t, VL] = result;
```
C8.2.296  LDFF1D (scalar plus scalar)

Contiguous load with first-faulting behavior of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

Encoding

LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #3}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
  end
  base = if n == 31 then SP[] else X[n, 64];
  offset = X[m, 64];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_SVE];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
    end
    else
      (data, fault) = (Zeros(msize), FALSE);
    end
    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
      ElemFFR[e, esize] = '0';
    end
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
  end
  if unknown then
    if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elseif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros(esize);
    else
      // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    end
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
  end
  Z[t, VL] = result;
C8.2.297  **LDFF1D (scalar plus vector)**

Gather load with first-faulting behavior of doublewords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

### 32-bit unpacked scaled offset

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] 16 15 14 13 12 10 9 5 4 0
1 1 0 0 0 1 0 1 1 xs 1 Zm 0 1 1 Pg Rn Zt
```

**Encoding**

```
LDFF1D { <Zt>.D }, <Pg>/Z, [ <Xn|SP>, <Zm>.D, <mod> #3 ]
```

**Decode for this encoding**

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 3;
```

### 32-bit unpacked unscaled offset

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] 16 15 14 13 12 10 9 5 4 0
1 1 0 0 0 1 0 1 1 xs 0 Zm 0 1 1 Pg Rn Zt
```

**Encoding**

```
LDFF1D { <Zt>.D }, <Pg>/Z, [ <Xn|SP>, <Zm>.D, <mod> ]
```

**Decode for this encoding**

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```
64-bit scaled offset

[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 | 0 |
  1 1 0 0 0 1 0 1 1 1 | Zm 1 1 1 | Pg Rn Zt

Encoding

LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #3]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 3;

64-bit unscaled offset

[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 | 0 |
  1 1 0 0 0 1 0 1 1 0 | Zm 1 1 1 | Pg Rn Zt

Encoding

LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

- **UXTW** when \( xs = 0 \)
- **SXTW** when \( xs = 1 \)

**Operation for all encodings**

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstranUnpredictableBool(ConstranUnpredictable_DATA) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(ElemP[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    bits(64) addr = base + (off << scale);
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = MemP[addr, mbytes, AccType_SVE];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    end
    else
      (data, fault) = (Zeros(msize), FALSE);
  end
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  end
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if fault & ConstranUnpredictableBool(ConstranUnpredictable_SVELDNFDATA) then
      ElemP[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstranUnpredictableBool(ConstranUnpredictable_SVELDNFZERO) then
      ElemP[result, e, esize] = Zeros(esize);
    else // merge
      ElemP[result, e, esize] = ElemP[orig, e, esize];
    end
  end
end
Z[t, VL] = result;
```
C8.2.298   LDFF1D (vector plus immediate)

Gather load with first-faulting behavior of doublewords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

Encoding

LDFF1D { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

Assembler symbols

<Zt>     Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>     Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>     Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>    Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        else
            (data, fault) = (Zeros(msize), FALSE);

        // FFR elements set to FALSE following a supressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR[e, esize] = '0';

        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR[e, esize] == '0';

    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else
            // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);

    Z[t, VL] = result;
### LDFF1H (scalar plus scalar)

Contiguous load with first-faulting behavior of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

#### 16-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 0 1</td>
<td>Rm</td>
<td>0 1 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

LDFF1H { <Zt>.H }, <Pg>/Z, [<Xn{|SP}{, <Xm>, LSL #1}]

**Decode for this encoding**

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 16;
boolean unsigned = TRUE;
```

#### 32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 0 1 0</td>
<td>Rm</td>
<td>0 1 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

LDFF1H { <Zt>.S }, <Pg>/Z, [<Xn{|SP}{, <Xm>, LSL #1}]

**Decode for this encoding**

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
```
64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 0</td>
<td>1 1</td>
<td>0 1 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
</tr>
</tbody>
</table>

**Encoding**

LDFF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

**Decode for this encoding**

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
```

**Assembler symbols**

- **<Zt>** Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<Xm>** Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

**Operation for all encodings**

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
```
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_SVE];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_CN0TFIRST];
    else
      (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
      ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
      if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNDATA) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
      elseif ConstrainUnpredictableBool(Unpredictable_SVELDFNZERO) then
        Elem[result, e, esize] = Zeros(esize);
      else  // merge
        Elem[result, e, esize] = Elem[orig, e, esize];
      else
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    endif
    Z[t, VL] = result;
  endif
LDFF1H (scalar plus vector)

Gather load with first-faulting behavior of unsigned halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit scaled offset

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 | 0 |
1 0 0 0 0 1 0 0 1 xs 1 Zm 0 1 1 Pg Rn Zt
```

Encoding

```
LDFF1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]
```

Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

32-bit unpacked scaled offset

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 | 0 |
1 1 0 0 0 1 0 0 1 xs 1 Zm 0 1 1 Pg Rn Zt
```

Encoding

```
LDFF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]
```

Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```
### 32-bit unpacked unscaled offset

<table>
<thead>
<tr>
<th>0 1 0 0 0 1 0 0 1 x s 0</th>
<th>Zm</th>
<th>0 1 1</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz&lt;1&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>msz&lt;0&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

`LDFF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]`

**Decode for this encoding**

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 32-bit unscaled offset

<table>
<thead>
<tr>
<th>1 0 0 0 0 1 0 0 1 x s 0</th>
<th>Zm</th>
<th>0 1 1</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>U ff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

`LDFF1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]`

**Decode for this encoding**

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 64-bit scaled offset

<table>
<thead>
<tr>
<th>1 1 0 0 0 0 1 0 0 1 1 1</th>
<th>Zm</th>
<th>1 1 1</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>U ff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

`LDFF1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]`
**Encoding**

LDFFIH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

**Decode for this encoding**

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 1;

**64-bit unscaled offset**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0 1 1 1 0</td>
<td>Zm 1 1 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Assembler symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

- **UXTW** when xs = 0
- **SXTW** when xs = 1
Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 & ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    bits(64) addr = base + (off << scale);
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_SVE];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
      (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a suppressed access/fault
    if faulted then
      ElemFFR[e, esize] = '0';
    // Value becomes CONSTRANGED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFLDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros(esize);
    else  // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
  Z[t, VL] = result;
C8.2.301 LDFF1H (vector plus immediate)

Gather load with first-faulting behavior of unsigned halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

```
| 31 30 29 28| 27 26 25 24| 23 22 21 20| 16| 15 14 13 12| 10 9 | 5 4 | 0 |
| 1 0 0 0 0 1 0 1 | 0 1 0 1 | imm5 | 1 | 1 | 1 | Pg | Zn | Zt |
```

Encoding

LDFF1H { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

64-bit element

```
| 31 30 29 28| 27 26 25 24| 23 22 21 20| 16| 15 14 13 12| 10 9 | 5 4 | 0 |
| 1 1 0 0 0 1 0 1 | 0 1 0 1 | imm5 | 1 | 1 | 1 | Pg | Zn | Zt |
```

Encoding

LDFF1H { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

Assembler symbols

```
<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
```
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

**Operation for all encodings**

```plaintext
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g,PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t,VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
  base = Z[n, VL];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_SVE];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
      (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
      ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if ![fault & ConstrainsUnpredictableBool(Unpredictable_SVELDNFDATA)] then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainsUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros(esize);
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
Z[t, VL] = result;
```


C8.2.302   **LDFF1SB (scalar plus scalar)**

Contiguous load with first-faulting behavior of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

### 16-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 1 1 0</td>
<td>Rm</td>
<td>0 1 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
</tr>
</tbody>
</table>

**Encoding**

LDFF1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, <Xm}>]

**Decode for this encoding**

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = FALSE;
```

### 32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 1 1 0 1</td>
<td>Rm</td>
<td>0 1 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
</tr>
</tbody>
</table>

**Encoding**

LDFF1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm}>]

**Decode for this encoding**

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
```
64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1</td>
<td>1 1 0 0</td>
<td>0 1 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
</tr>
</tbody>
</table>

**Encoding**

LDFF1SB { <Zt>.D }, <Pg>-/Z, [<Xn|SP>{, <Xm}>]

**Decode for this encoding**

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;

**Assembler symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

**Operation for all encodings**

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
        else
            (data, fault) = (Zeros(msize), FALSE);

        // FFR elements set to FALSE following a supressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR[e, esize] = '0';

        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR[e, esize] == '0';
        if unknown then
            if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                Elem[result, e, esize] = Extend(data, esize, unsigned);
            elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                Elem[result, e, esize] = Zeros(esize);
            else  // merge
                Elem[result, e, esize] = Elem[orig, e, esize];
            else
                Elem[result, e, esize] = Extend(data, esize, unsigned);
        end
    end
end

Z[t, VL] = result;
C8.2.303  LDFF1SB (scalar plus vector)

Gather load with first-faulting behavior of signed bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit unpacked unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0 0 xs 0</td>
<td>Zm 0 0 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

Encoding

LDFF1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 0 xs 0</td>
<td>Zm 0 0 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

LDFF1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
64-bit unscaled offset

[31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 13 12 10 9 5 4 0]
11000100 0 0 10 0 100 0 1 0
Zm 1 0 1 Pg Rn Zt
U ff
msz<1>
msz<0>

Encoding

LDFF1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:
UXTW when xs = 0
SXTW when xs = 1

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
if n == 31 && ConstrainsUnpredictableBool(Unpredictable_CHECKSPONEACTIVE) then
    CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];
for e = 0 to elements - 1
    if Elem[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        else
            (data, fault) = (Zeros(msize), FALSE);
        // FFR elements set to FALSE following a suppressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR[e, esize] = '0';
        // Value becomes CONstrained Unpredictable after an FFR element is FALSE
        unknown = unknown || ElemFFR[e, esize] == '0';
        if unknown then
            if fault && ConstrainsUnpredictableBool(Unpredictable_SVELDNFDATA) then
                Elem[result, e, esize] = Extend(data, esize, unsigned);
            elseif ConstrainsUnpredictableBool(Unpredictable_SVELDNFZERO) then
                Elem[result, e, esize] = Zeros(esize);
            else // merge
                Elem[result, e, esize] = Elem[orig, e, esize];
            else
                Elem[result, e, esize] = Extend(data, esize, unsigned);
        Z[t, VL] = result;
C8.2.304  LDFF1SB (vector plus immediate)

Gather load with first-faulting behavior of signed bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 0 1</td>
<td>imm5</td>
<td>1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

LDFF1SB { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0 0 1</td>
<td>imm5</td>
<td>1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

LDFF1SB { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

Assembler symbols

<Zt>               Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

**Operation for all encodings**

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[n, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        else
            (data, fault) = (Zeros(msize), FALSE);
        // FFR elements set to FALSE following a supressed access/fault
        faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if fault & ConstrainUnpredictableBool(Unpredictable_SVELNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELNDFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);
    Z[n, VL] = result;
```
C8.2.305   LDFF1SH (scalar plus scalar)

Contiguous load with first-faulting behavior of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

```
1  0  1  0  1  0  1  0  0  1 | Rm   0  1  1 | Pg   Rn  Zt
```

dtype<3:1>
dtype<0>

Encoding

LDFF1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
```

64-bit element

```
1  0  1  0  1  0  1  0  0  0 | Rm   0  1  1 | Pg   Rn  Zt
```

dtype<3:1>
dtype<0>

Encoding

LDFF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
```
Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
  for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
      bits(64) addr = base + (UInt(offset) + e) * mbytes;
      if first then
        // Mem[] will not return if a fault is detected for the first active element
        data = Mem[addr, mbytes, AccType_SVE];
        first = FALSE;
      else
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
        else
          (data, fault) = (Zeros(msize), FALSE);
        faulted = faulted || fault;
        if faulted then
          ElemFFR[e, esize] = '0';
          // FFR elements set to FALSE following a supressed access/fault
        unknown = unknown || ElemFFR[e, esize] == '0';
        if unknown then
          if fault && ConstrainUnpredictableBool(Unpredictable_SVELNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
          elsif ConstrainUnpredictableBool(Unpredictable_SVELNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
          else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
          // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
          unknown = unknown || ElemFFR[e, esize] == '0';
$$\text{Elem}[\text{result}, e, \text{esize}] = \text{Extend}(\text{data}, \text{esize}, \text{unsigned});$$

$$Z[t, VL] = \text{result};$$
C8.2.306  **LDFF1SH** (scalar plus vector)

Gather load with first-faulting behavior of signed halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

**32-bit scaled offset**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

LDFF1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]

Decode for this encoding

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

**32-bit unpacked scaled offset**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

LDFF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]

Decode for this encoding

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```
### 32-bit unpacked unscaled offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Zm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
msz<1>
msz<0>
Uff
```

**Encoding**

LDFFISH \( (\langle Zt \rangle.D), \langle Pg \rangle/Z, [\langle Xn \rangle|SP>, \langle Zm \rangle.D, \langle \text{mod} \rangle] \)

**Decode for this encoding**

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs._size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 32-bit unscaled offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Zm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
Uff
```

**Encoding**

LDFFISH \( (\langle Zt \rangle.S), \langle Pg \rangle/Z, [\langle Xn \rangle|SP>, \langle Zm \rangle.S, \langle \text{mod} \rangle] \)

**Decode for this encoding**

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs._size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 64-bit scaled offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Zm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
Uff
```
**Encoding**

LDFFISH \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

**Decode for this encoding**

```plaintext
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 1;
```

**64-bit unscaled offset**

| 31 30 29 28| 27 26 25 24| 23 22 21 20| 16| 15 14 13 12| 10 9 | 5 4 | 0 |
|---|---|---|---|---|---|---|---|---|
| 1 1 0 0 0 1 0 | 1 1 0 | Pg | Rn | Zt |

msz<1>

msz<0>

**Encoding**

LDFFISH \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>, <Zm>.D]

**Decode for this encoding**

```plaintext
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

**Assembler symbols**

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Zm>`: Is the name of the offset scalable vector register, encoded in the "Zm" field.
- `<mod>`: Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:
  - UXTW when xs = 0
  - SXTW when xs = 1
Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(PL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAisignment();
  else
    if n == 31 then CheckSPAisignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    bits(64) addr = base + (off << scale);
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_SVE];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
      (data, fault) = (Zeros(msize), FALSE);

  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';

  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault && ConstrainUnpredictableBool(Unpredictable_SVLENFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elseif ConstrainUnpredictableBool(Unpredictable_SVLEDFZERO) then
      Elem[result, e, esize] = Zeros(esize);
    else
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
C8.2.307   LDFF1SH (vector plus immediate)

Gather load with first-faulting behavior of signed halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

| 31 30 29 28|27 26 25 24|23 22 21 20| 16|15|14|13|12|10| 9 | 5 | 4 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | imm5 | Pg | Zn | Zt |

msz<1>      msz<0>

Encoding

LDFF1SH { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

64-bit element

| 31 30 29 28|27 26 25 24|23 22 21 20| 16|15|14|13|12|10| 9 | 5 | 4 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | imm5 | Pg | Zn | Zt |

msz<1>      msz<0>

Encoding

LDFF1SH { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

Assembler symbols

<Zt>      Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<\text{Pg}> \quad \text{Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.}

<\text{Zn}> \quad \text{Is the name of the base scalable vector register, encoded in the "Zn" field.}

<\text{imm}> \quad \text{Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.}

**Operation for all encodings**

\begin{verbatim}
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        else
            (data, fault) = (Zeros(msize), FALSE);
        // FFR elements set to FALSE following a suppressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR[e, esize] = '0';
        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR[e, esize] == '0';
        if unknown then
            if fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                Elem[result, e, esize] = Extend(data, esize, unsigned);
            elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                Elem[result, e, esize] = Zeros(esize);
            else // merge
                Elem[result, e, esize] = Elem[orig, e, esize];
            end
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        end
        Z[t, VL] = result;
    end
end
\end{verbatim}
C8.2.308  LDFF1SW (scalar plus scalar)

Contiguous load with first-faulting behavior of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

Encoding

LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;

Assembler symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>  Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];
    end
endif

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
        else
            (data, fault) = (Zeros(msize), FALSE);
        end
    end
endif

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
    end
endif

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elseif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else
            // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
        end
    end
endif

Z[t, VL] = result;
C8.2.309  LDFF1SW (scalar plus vector)

Gather load with first-faulting behavior of signed words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit unpacked scaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>10 9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0</td>
<td>1 0</td>
<td>1 0</td>
<td>xs</td>
<td>1</td>
<td>Zm</td>
<td>0 0</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

**Decode for this encoding**

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 2;

32-bit unpacked unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>10 9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0</td>
<td>1 0</td>
<td>1 0</td>
<td>xs</td>
<td>0</td>
<td>Zm</td>
<td>0 0</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

**Decode for this encoding**

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
64-bit scaled offset

Encoding

LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs Unsigned = TRUE;
integer scale = 2;

64-bit unscaled offset

Encoding

LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

- UXTW when \( xs = 0 \)
- SXTW when \( xs = 1 \)

**Operation for all encodings**

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];
else
  for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
      integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
      bits(64) addr = base + (off << scale);
      if first then
        // Mem[] will not return if a fault is detected for the first active element
        data = Mem[addr, mbytes, AccType_SVE];
        first = FALSE;
      else
        // MemNF[] will return fault=True if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        else
          (data, fault) = (Zeros(msize), FALSE);

  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';

  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elseif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros(esize);
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
  Z[t, VL] = result;
```
C8.2.310   LDFF1SW (vector plus immediate)

Gather load with first-faulting behavior of signed words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

Encoding

LDFF1SW { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}] 

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        else
            (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;

    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';

    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);

    Z[t, VL] = result;
C8.2.311   LDFF1W (scalar plus scalar)

Contiguous load with first-faulting behavior of unsigned words to elements of a vector register from the memory
address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address.
After each element access the index value is incremented, but the index register is not updated. Inactive elements
will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and
enabled at the current Exception level.

32-bit element

| [31 30 29 28|27 26 25 24| 22 21 20] | 16|15 14 13 12| 10 9 | 5 | 4 | 0 |
| 1 0 1 0 0 1 0 1 | 0 1 1 | Pg | Rn | Zt |

dtype<3:1>
dtype<0>

Encoding

LDFF1W {<Zt>.S}, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;

64-bit element

| [31 30 29 28|27 26 25 24| 22 21 20] | 16|15 14 13 12| 10 9 | 5 | 4 | 0 |
| 1 0 1 0 0 1 0 1 | 0 1 1 | Pg | Rn | Zt |

dtype<3:1>
dtype<0>

Encoding

LDFF1W {<Zt>.D}, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_SVE];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
    else
      (data, fault) = (Zeros(msize), FALSE);
  else
    ElemFFR[e, esize] = '0';
// FFR elements set to FALSE following a suppressed access/fault
faulted = faulted || fault;
if faulted then
  ElemFFR[e, esize] = '0';
// Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
unknown = unknown || ElemFFR[e, esize] == '0';
if unknown then
  if fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
    Elem[result, e, esize] = Extend(data, size, unsigned);
  elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
    Elem[result, e, esize] = Zeros(esize);
  else // merge
    Elem[result, e, esize] = Elem[orig, e, esize];
else
\[ \text{Elem}[\text{result}, e, \text{esize}] = \text{Extend}(\text{data}, \text{esize}, \text{unsigned}); \]

\[ Z[t, VL] = \text{result}; \]
LDFF1W (scalar plus vector)

Gather load with first-faulting behavior of unsigned words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

### 32-bit scaled offset

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [16 15 14 13 12] 10 9 | 5 4 | 0
1 0 0 0 0 1 0 1 0 xs 1 Zm 0 1 1 Pg Rn Zt
```

#### Encoding

```
LDFF1W { <Zt>.S }, <Pg>/Z, [Xn|SP>, <Zm>.S, <mod> #2]
```

#### Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

### 32-bit unpacked scaled offset

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [16 15 14 13 12] 10 9 | 5 4 | 0
1 1 0 0 0 1 0 1 0 xs 1 Zm 0 1 1 Pg Rn Zt
```

#### Encoding

```
LDFF1W { <Zt>.D }, <Pg>/Z, [Xn|SP>, <Zm>.D, <mod> #2]
```

#### Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```
32-bit unpacked unscaled offset

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 | 0 |
  1 1 0 0 0 1 0 1 0 xs 0 Zm 0 1 1 Pg Rn Zt
```

Encoding

LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 | 0 |
  1 0 0 0 0 1 0 1 0 xs 0 Zm 0 1 1 Pg Rn Zt
```

Encoding

LDFF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled offset

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 | 0 |
  1 1 0 0 0 1 0 1 0 1 1 Zm 1 1 1 Pg Rn Zt
```

Encoding

LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
Encoding
LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

Decode for this encoding
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 2;

64-bit unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 | 16 15 14 13 12 |
| 0  9  8  7  6  5  4  3  2  1  0  1 |
| Zm 1 1 1 |
| Pg Rn Zt |

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

UXTW when xs = 0
SXTW when xs = 1
Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bitalias mask = P[g, PL];
bitalias base;
bitalias offset;
bitalias result;
bitalias orig = Z[t, VL];
bitalias data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];
  for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
      integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
      bitalias addr = base + (off << scale);
      if first then
        // Mem[] will not return if a fault is detected for the first active element
        data = Mem[addr, mbytes, AccType_SVE];
        first = FALSE;
      else
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
      else
        (data, fault) = (Zeros(msize), FALSE);
      // FFR elements set to FALSE following a suppressed access/fault
      faulted = faulted || fault;
      if faulted then
        ElemFFR[e, esize] = '0';
        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR[e, esize] == '0';
      if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
          Elem[result, e, esize] = Extend(data, esize, unsigned);
        elseif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
          Elem[result, e, esize] = Zeros(esize);
        else // merge
          Elem[result, e, esize] = Elem[orig, e, esize];
      else
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    Z[t, VL] = result;
C8.2.313   LDFF1W (vector plus immediate)

Gather load with first-faulting behavior of unsigned words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 1 0 0 1</td>
<td>imm5</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
</tr>
</tbody>
</table>

msz<1>                                  msz<0>

Encoding

LDFF1W { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]  

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 1 0 0 1</td>
<td>imm5</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
</tr>
</tbody>
</table>

msz<1>                                  msz<0>

Encoding

LDFF1W { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]  

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
  base = Z[n, VL];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_SVE];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
      (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if fault & ConstrinUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrinUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros(esize);
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
  Z[t, VL] = result;
C8.2.314   LDNF1B

Contiguous load with non-faulting behavior of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

8-bit element

Contiguous load with non-faulting behavior of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

16-bit element

Contiguous load with non-faulting behavior of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
### 32-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1</td>
<td>0 0 1 0 1 1</td>
<td>imm4</td>
<td>1 0 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- `dtype<3:1>`
- `dtype<0>`

#### Encoding

`LDNF1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]`

#### Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

### 64-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1</td>
<td>0 0 1 0 1 1</td>
<td>imm4</td>
<td>1 0 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- `dtype<3:1>`
- `dtype<0>`

#### Encoding

`LDNF1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]`

#### Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

#### Assembler symbols

- `<Zt>`
  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `< Pg>`
  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`
  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`
  Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    // MemNF[] will return fault=TRUE if access is not performed for any reason
    (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elseif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros(esize);
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
Z[t, VL] = result;
C8.2.315   **LDNF1D**

Contiguous load with non-faulting behavior of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

### Encoding

```
LDNF1D { <Zt>.D }, <Pg>/Z, [ <Xn|SP>{, #<imm>, MUL VL} ]
```

#### Decode for this encoding

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

#### Assembler symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

#### Operation

```java
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
```
if !AnyActiveElement(mask, esize) then
  if n == 31 & ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
  for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements) + e;
      bits(64) addr = base + eoff * mbytes;
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
      (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
      ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
      if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
      else if ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
        Elem[result, e, esize] = Zeros(esize);
      else // merge
        Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    Z[t, VL] = result;
C8.2.316   LDNF1H

Contiguous load with non-faulting behavior of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

16-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>19 16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 0 1 1</td>
<td>imm4</td>
<td>1 0 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

dtype<3:1>          dtype<0>  

Encoding

LDNF1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #imm}, MUL VL]}

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>19 16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 0 1 1</td>
<td>imm4</td>
<td>1 0 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

dtype<3:1>          dtype<0>  

Encoding

LDNF1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #imm}, MUL VL]}

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
64-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>22 21 20 19</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  0  1  0  0  1  0  1</td>
<td>1  0  1  1</td>
<td>imm4</td>
<td>1  0</td>
<td>1  0</td>
<td>Zt</td>
</tr>
</tbody>
</table>
```

dtype<3:1>
dtype<0>

Encoding

LDNF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

Assembler symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation for all encodings

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t,VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainingUnpredictableBoo(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAisignment();
  else
    if n == 31 then CheckSPAisignment();
    base = if n == 31 then SP[] else X[n, 64];
for e = 0 to elements-1
  if Elem[mask,e,esize] == '1' then
    integer eoff = (offset * elements) + e;
```
bits(64) addr = base + eoff * mbytes;
// MemNF[] will return fault=TRUE if access is not performed for any reason
(data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
else
 (data, fault) = (Zeros(msize), FALSE);

// FFR elements set to FALSE following a suppressed access/fault
faulted = faulted || fault;
if faulted then
  ElemFFR[e, esize] = '0';

// Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
unknown = unknown || ElemFFR[e, esize] == '0';
if unknown then
 if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
  Elem[result, e, esize] = Extend(data, esize, unsigned);
 elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
  Elem[result, e, esize] = Zeros(esize);
 else  // merge
  Elem[result, e, esize] = Elem[orig, e, esize];
 else
  Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
C8.2.317 LDNF1SB

Contiguous load with non-faulting behavior of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

16-bit element

```
  1 0 1 0 0 1 0 1 1 1 0 1 | imm4| 1 0 1 | Pg | Rn | Zt |
```

Encoding

LDNF1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

32-bit element

```
  1 0 1 0 0 1 0 1 1 1 0 1 | imm4| 1 0 1 | Pg | Rn | Zt |
```

Encoding

LDNF1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```
64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>22 21 20</th>
<th>19</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1</td>
<td>1 1 0 0 1</td>
<td>imm4</td>
<td>1 0 1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dtype<3:1>
dtype<0>

Encoding

LDNF1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 & ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
  for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset + elements) + e;
bits(64) addr = base + eoff * mbytes;
// MemNF[] will return fault=TRUE if access is not performed for any reason
(data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
else
  (data, fault) = (Zeros(msize), FALSE);

// FFR elements set to FALSE following a suppressed access/fault
faulted = faulted || fault;
if faulted then
  ElemFFR[e, esize] = '0';

// Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
unknown = unknown || ElemFFR[e, esize] == '0';
if unknown then
  if fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
    Elem[result, e, esize] = Zeros(esize);
  else  // merge
    Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
else
  Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
C8.2.318  **LDNF1SH**

Contiguous load with non-faulting behavior of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

### 32-bit element

```
[31 30 29 28] [27 26 25 24]  [22 21 20] [19]  [16 15 14 13 12]  [10 9]  [5 4]  [0]
1 0 1 0 0 1 0 1 0 1 1 1 imm4 1 0 1 Pg Rn Zt
```

**Encoding**

LDNFISH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

#### Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

### 64-bit element

```
[31 30 29 28] [27 26 25 24]  [22 21 20] [19]  [16 15 14 13 12]  [10 9]  [5 4]  [0]
1 0 1 0 0 1 0 1 0 0 0 1 imm4 1 0 1 Pg Rn Zt
```

**Encoding**

LDNFISH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

#### Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```
Assembler symbols

\(<Zt>\) Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<imm>\) Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation for all encodings

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[Pl, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    // MemNF[] will return fault=TRUE if access is not performed for any reason
    (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
if unknown then
  if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
    Elem[result, e, esize] = Zeros(esize);
  else  // merge
    Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
Z[t, VL] = result;
```
C8.2.319   LDNF1SW

Contiguous load with non-faulting behavior of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

Encoding

```
LDNF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

Decoding for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

Assembler symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
```
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
  
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset + elements) + e;
    bits(64) addr = base + eoff * mbytes;
    // MemNF[] will return fault=TRUE if access is not performed for any reason
    (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);

  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros(esize);
    else  // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);

  Z[t, VL] = result;
C8.2.320   LDNF1W

Contiguous load with non-faulting behavior of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector’s in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

[31 30 29 28|27 26 25 24]   22 21 20|19   16|15 14 13 12|   10  9   |   5  4   |  0  |
 1 0 1 0 0 1 0 1 0 1 0 1 1  imm4  1 0 1  Pg  Rn  Zt

dtype<3:1>......................
dtype<0>......................

Encoding

LDNF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

64-bit element

[31 30 29 28|27 26 25 24]   22 21 20|19   16|15 14 13 12|   10  9   |   5  4   |  0  |
 1 0 1 0 0 1 0 1 0 1 0 1 1  imm4  1 0 1  Pg  Rn  Zt

dtype<3:1>......................
dtype<0>......................

Encoding

LDNF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
  
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    // MemNF[] will return fault=TRUE if access is not performed for any reason
    (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);
  
  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';

  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
if unknown then
  if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
    Elem[result, e, esize] = Zeros(esize);
  else // merge
    Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
Z[t, VL] = result;
C8.2.321  LDNT1B (scalar plus immediate)

Contiguous load non-temporal of bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

![Encoding Diagram]

**Encoding**

LDNT1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);

**Assembler symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
  else
    Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.322 LDNT1B (scalar plus scalar)

Contiguous load non-temporal of bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

Encoding

```
LDNT1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]
```

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
```

Assembler symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
   if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
   else
      if n == 31 then
         base = if n == 31 then SP[] else X[n, 64];
         offset = X[m, 64];
```

```
   bits(1) msz<1>
   bits(1) msz<0>
```

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
### C8.2.323 LDNT1B (vector plus scalar)

Gather load non-temporal of unsigned bytes to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

#### 32-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 0 0</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

**Encoding**

LDNT1B { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]

**Decode for this encoding**

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;

#### 64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0 0 0</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

**Encoding**

LDNT1B { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]

**Decode for this encoding**

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
  base = Z[n, VL];
  offset = X[m, 64];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
    data = Mem[addr, mbytes, AccType_SVESTREAM];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);
Z[t, VL] = result;
C8.2.324   LDNT1D (scalar plus immediate)

Contiguous load non-temporal of doublewords to elements of a vector register from the memory address generated
by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size,
irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device
memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

Encoding

LDNT1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);

Assembler symbols

<Zt>        Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>        Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>     Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>       Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the
"imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.325  **LDNT1D (scalar plus scalar)**

Contiguous load non-temporal of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

```
[31 30 29 28|27 26 25 24|23 22 21 20] [16|15 14 13|12] 10 9 | 5 4 | 0 |
 1 0 1 0 0 1 0 0 | 1 1 0 0 | Rm | 1 1 0 | Pg | Rn | Zt |
msz<1> msz<0>
```

**Encoding**

```
LDNT1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
```

**Assembler symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 \&\& ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
```
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
    else
        Elem[result, e, esize] = Zeros(esize);
    Z[t, VL] = result;
C8.2.326   LDNT1D (vector plus scalar)

Gather load non-temporal of doublewords to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

Encoding

LDNT1D { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}

Decode for this encoding

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;

Assembler symbols

<Zt>   Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>   Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>   Is the name of the base scalable vector register, encoded in the "Zn" field.
<Xm>   Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
  base = Z[n, VL];
offset = X[m, 64];

for e = 0 to elements-1
   if ElemP[mask, e, esize] == '1' then
      bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
      data = Mem[addr, mbytes, AccType_SVESTREAM];
      Elem[result, e, esize] = Extend(data, esize, unsigned);
   else
      Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.327   LDNT1H (scalar plus immediate)

Contiguous load non-temporal of halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

Encoding

LDNT1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
in integer t = UInt(Zt);
in integer n = UInt(Rn);
in integer g = UInt(Pg);
constant integer esize = 16;
in integer offset = SInt(imm);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n, 64];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
    else
        Elem[result, e, esize] = Zeros(esize);
Z[t, VL] = result;
C8.2.328  LDNT1H (scalar plus scalar)

Contiguous load non-temporal of halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

Encoding

LDNT1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;

Assembler symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n, 64];
  offset = X[m, 64];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
  else
    Elem[result, e, esize] = Zeros(esize);
  
  Z[t, VL] = result;
C8.2.329 LDNT1H (vector plus scalar)

Gather load non-temporal of unsigned halfwords to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit unscaled offset

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0</td>
<td>0 1 0 0</td>
<td>Rm 1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Encoding

LDNT1H \{ <Zt>.S \}, <Pg>/Z, \[<Zn>.S{, <Xm>}\]

Decode for this encoding

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;

64-bit unscaled offset

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0</td>
<td>0 1 0 0</td>
<td>Rm 1 1 0</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Encoding

LDNT1H \{ <Zt>.D \}, <Pg>/Z, \[<Zn>.D{, <Xm>}\]

Decode for this encoding

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g =UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
  base = Z[n, VL];
  offset = X[m, 64];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
    data = Mem[addr, mbytes, AccType_SVESTREAM];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);
Z[t, VL] = result;
C8.2.330   LDNT1SB

Gather load non-temporal of signed bytes to active elements of a vector register from memory addresses generated
by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device
memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and
enabled at the current Exception level.

32-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 0 0</td>
<td>Rm 1 0 0</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

Encoding

LDNT1SB { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm}>]

Decode for this encoding

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;

64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0 0 0</td>
<td>Rm 1 0 0</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

Encoding

LDNT1SB { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm}>]

Decode for this encoding

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
  base = Z[n, VL];
  offset = X[m, 64];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
    data = Mem[addr, mbytes, AccType_SVESTREAM];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);
Z[t, VL] = result;
C8.2.331   LDNT1SH

Gather load non-temporal of signed halfwords to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10000100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

LDNT1SH { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm}>]

Decode for this encoding

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;

64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11000100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

LDNT1SH { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm}>]

Decode for this encoding

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation for all encodings

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        data = Mem[addr, mbytes, AccType_SVESTREAM];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);
Z[t, VL] = result;
```
C8.2.332  LDNT1SW

Gather load non-temporal of signed words to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

Encoding

LDNT1SW { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]

Decode for this encoding

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
   base = Z[n, VL];
offset = X[m, 64];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
    data = Mem[addr, mbytes, AccType_SVESTREAM];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros(esize);
  end if
Z[t, VL] = result;
C8.2.333  **LDNT1W (scalar plus immediate)**

Contiguous load non-temporal of words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

```
| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 | 14 | 13 | 12 | 10 | 9  | 5 | 4 | 0 |
|-------------|-------------|-------------|---|---|---|---|---|---|---|---|---|---|---|---|
| 1           | 0           | 1           | 0 | 1 | 0 | 0 | 0 | 0 | imm4| 1 | 1 | 1 | Pg | Rn | Zt|
```

**Encoding**

`LDNT1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{},, #<imm>, MUL VL]`  

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
```

**Assembler symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAIndex();
  else
    if n == 31 then CheckSPAAlignment();
    base = if n == 31 then SP[] else X[n, 64];
```
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
  else
    Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
C8.2.334  LDNT1W (scalar plus scalar)

Contiguous load non-temporal of words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

Encoding

LDNT1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;

Assembler symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n, 64];
  offset = X[m, 64];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
  else
    Elem[result, e, esize] = Zeros(esize);
  Z[t, VL] = result;
C8.2.335 LDNT1W (vector plus scalar)

Gather load non-temporal of unsigned words to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0</td>
<td>Rm</td>
<td>1 1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

Encoding

LDNT1W { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]

Decode for this encoding

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;

64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0</td>
<td>Rm</td>
<td>1 1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

Encoding

LDNT1W { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]}

Decode for this encoding

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation for all encodings

CheckNonStreamingSVEEnabled();
c
constant integer VL = CurrentVL;
c
constant integer PL = VL DIV 8;
c
constant integer elements = VL DIV esize;
c
bits(PL) mask = P[g, PL];
c
bits(VL) base;
c
bits(64) offset;
c
bits(VL) result;
c
bits(msize) data;
c
constant integer mbytes = msize DIV 8;
c
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
c
if AnyActiveElement(mask, esize) then
c
    base = Z[n, VL];
c
    offset = X[m, 64];
c
for e = 0 to elements-1
c
    if Elem[mask, e, esize] == '1' then
c
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
c
        data = Mem[addr, mbytes, AccType_SVESTREAM];
c
        Elem[result, e, esize] = Extend(data, esize, unsigned);
c
    else
c
        Elem[result, e, esize] = Zeros(esize);
c
    Z[t, VL] = result;
C8.2.336 LDR (predicate)

Load a predicate register from a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current predicate register size in bytes. This instruction is unpredicated.

The load is performed as contiguous byte accesses, each containing 8 consecutive predicate bits in ascending element order, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then a general-purpose base register must be aligned to 2 bytes.

Encoding

LDR <Pt>, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Pt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);

Assembler symbols

<Pt> Is the name of the destination scalable predicate register, encoded in the "Pt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = PL DIV 8;
bits(64) base;
integer offset = imm * elements;
bits(PL) result;

if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n, 64];

boolean aligned = AArch64.CheckAlignment(base + offset, 2, AccType_SVE, FALSE);
for e = 0 to elements-1
    Elem[result, e, 8] = AArch64.MemSingle[base + offset, 1, AccType_SVE, aligned];
    offset = offset + 1;
P[t, PL] = result;
C8.2.337  LDR (vector)

Load a vector register from a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current vector register size in bytes. This instruction is unpredicated.

The load is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

Encoding

LDR <Zt>, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);

Assembler symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>  Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 8;
bits(64) base;
integer offset = imm * elements;
bits(VL) result;
if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n, 64];
boolean aligned = AArch64.CheckAlignment(base + offset, 16, AccType_SVE, FALSE);
for e = 0 to elements-1
    Elem[result, e, 8] = AArch64.MemSingle[base + offset, 1, AccType_SVE, aligned];
    offset = offset + 1;
Z[t, VL] = result;
C8.2.338  LSL (immediate, predicated)

Shift left by immediate each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.

Encoding

<table>
<thead>
<tr>
<th>Zdn</th>
<th>&lt;T&gt;</th>
<th>Pg/M</th>
<th>Zdn</th>
<th>#const</th>
</tr>
</thead>
</table>

Decode for this encoding

if !HaveSVE() &amp; !HaveSME() then UNDEFINED;
bits(4) tsz = tszh:tszl;
integer esize;
case tsz of
when '0000' UNDEFINED;
when '0001' esize = 8;
when '001x' esize = 16;
when '01xx' esize = 32;
when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = UInt(tsz:imm3) - esize;

Assembler symbols

&lt;Zdn&gt;  Is the name of the source and destination scalable vector register, encoded in the ”Zdn” field.
&lt;T&gt;    Is the size specifier, encoded in the ”tszh:tszl” field. It can have the following values:

<table>
<thead>
<tr>
<th>&lt;T&gt;</th>
<th>tsz</th>
<th>tszl</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>H</td>
<td>00</td>
<td>lx</td>
</tr>
<tr>
<td>S</td>
<td>01</td>
<td>xx</td>
</tr>
<tr>
<td>D</td>
<td>lx</td>
<td>xx</td>
</tr>
</tbody>
</table>

The encoding tsz = 00, tszl = 00 is reserved.

&lt;Pg&gt;   Is the name of the governing scalable predicate register P0-P7, encoded in the ”Pg” field.

&lt;const&gt;  Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in ”tsz:imm3”.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer PL = VL DIV 8;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = LSL(element1, shift);
else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.339  LSL (Immediate, unpredicated)

Shift left by immediate each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

![Binary representation of instruction](image)

**Encoding**

LSL <Zd>.<T>, <Zn>.<T>, #<const>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;

bits(4) tsize = tszh:tszl;
integer esize;

case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;

integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;

**Assembler symbols**

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

B  when tszh = 00, tszl = 01
H  when tszh = 00, tszl = 1x
S  when tszh = 01, tszl = xx
D  when tszh = 1x, tszl = xx

The encoding tszh = 00, tszl = 00 is reserved.

<Zn>  Is the name of the source scalable vector register, encoded in the "Zn" field.

<const>  Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  Elem[result, e, esize] = LSL(element1, shift);
$Z[d, \ VL] = \text{result;}$

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.340  LSL (vectors)

Shift left active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

Encoding

LSL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
   B when size = 00
   H when size = 01
   S when size = 10
   D when size = 11
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
   if Elem[mask, e, esize] == '1' then
      bits(esize) element1 = Elem[operand1, e, esize];
      bits(esize) element2 = Elem[operand2, e, esize];
      integer shift = Min(UInt(element2), esize);
      Elem[result, e, esize] = LSL(element1, shift);
   else
      Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.341  LSL (wide elements, predicated)

Shift left active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

Encoding

LSL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.D

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
g = UInt(Pg);
dn = UInt(Zdn);
m = UInt(Zm);

Assembler symbols

<Zdn>   Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>     Is the size specifier, encoded in the "size" field. It can have the following values:
        B when size = 00
        H when size = 01
        S when size = 10
        The encoding size = 11 is reserved.
<Pg>    Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>    Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
   if Elem[mask, e, esize] == '1' then
      bits(esize) element1 = Elem[operand1, e, esize];
      bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
      integer shift = Min(UInt(element2), esize);
      Elem[result, e, esize] = LSL(element1, shift);
   else
      Elem[result, e, esize] = Elem[operand1, e, esize];
\[ Z[dn, VL] = \text{result}; \]

**Operational information**

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**C8.2.342  LSL (wide elements, unpredicated)**

Shift left all elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and place the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

**Encoding**

```
LSL <Zd>.<T>, <Zn>.<T>, <Zm>.D
```

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

**Assembler symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - The encoding size = 11 is reserved.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
  integer shift = Min(UInt(element2), esize);
  Elem[result, e, esize] = LSL(element1, shift);
Z[d, VL] = result;
```
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.343   LSLR

Reversed shift left active elements of the second source vector by corresponding elements of the first source vector
and destructively place the results in the corresponding elements of the first source vector. The shift amount operand
is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive
elements in the destination vector register remain unmodified.

Encoding

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 0</td>
<td>size</td>
<td>0 1 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0 0</td>
<td>Pg</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>L</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn>            Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>              Is the size specifier, encoded in the "size" field. It can have the following values:
B                when size = 00
H                when size = 01
S                when size = 10
D                when size = 11
<Pg>             Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>             Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    integer shift = Min(UInt(element1), esize);
    Elem[result, e, esize] = LSL(element2, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.344  LSR (immediate, predicated)

Shift right by immediate, inserting zeroes, each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9 8 7</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>tszh</td>
<td>0 0 0 1 1 0 0</td>
<td>Pg</td>
<td>tszl</td>
<td>imm3</td>
<td>Zdn</td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

LSR <Zdn>..<T>, <Pg>/M, <Zdn>..<T>, #<const>

**Decode for this encoding**

```plaintext
if !HaveSVE() && !HaveSME() then UNDEFINED;

constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer PL = VL DIV 8;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
```

**Assembler symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  - B when tszh = 00, tszl = 01
  - H when tszh = 00, tszl = 1x
  - S when tszh = 01, tszl = xx
  - D when tszh = 1x, tszl = xx
  - The encoding tszh = 00, tszl = 00 is reserved.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<const>` Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

**Operation**

```plaintext
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer PL = VL DIV 8;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
```
\[\text{Elem}[\text{result, e, esize}] = \text{LSR}(\text{element1, shift});\]
\[\text{else}\]
\[\text{Elem}[\text{result, e, esize}] = \text{Elem}[\text{operand1, e, esize}];\]
\[Z[\text{dn, VL}] = \text{result};\]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is **UNPREDICTABLE**:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SVE Instruction Descriptions
C8.2 Alphabetical list of SVE instructions

C8.2.345   LSR (immediate, unpredicated)

Shift right by immediate, inserting zeroes, each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>tszh</td>
<td>tszl</td>
<td>imm3</td>
<td>1 0 0 1 0</td>
<td>1</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>

Encoding
LSR <Zd>.<T>, <Zn>.<T>, #<const>

Decode for this encoding
if !HaveSVE() & & !HaveSME() then UNDEFINED;
bite(4) tsize = tszh:tszl;
integer esize;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler symbols

<Zd>   Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>   Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  B     when tszh = 00, tszl = 01
  H     when tszh = 00, tszl = 1x
  S     when tszh = 01, tszl = xx
  D     when tszh = 1x, tszl = xx

The encoding tszh = 00, tszl = 00 is reserved.

<Zn>   Is the name of the source scalable vector register, encoded in the "Zn" field.

<const>   Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bite(VL) operand1 = Z[n, VL];
bite(VL) result;
for e = 0 to elements-1
  bite(esize) element1 = Elem[operand1, e, esize];
  Elem[result, e, esize] = LSR(element1, shift);
Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.346  LSR (vectors)

Shift right, inserting zeroes, active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

Encoding

```
LSR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decoded for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler symbols

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    integer shift = Min(UInt(element2), esize);
    Elem[result, e, esize] = LSR(element1, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
```
Z[dn, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.347  LSR (wide elements, predicated)

Shift right, inserting zeroes, active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

Encoding
LSR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.D

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>    Is the size specifier, encoded in the "size" field. It can have the following values:
        B when size = 00
        H when size = 01
        S when size = 10
        The encoding size = 11 is reserved.
<Pg>   Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>   Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
        integer shift = Min(UInt(element2), esize);
        Elem[result, e, esize] = LSR(element1, shift);
    else

```
\[
\text{Elem[result, e, esize] = Elem[operand1, e, esize]};
\]

\[
Z[dn, VL] = \text{result};
\]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.348   LSR (wide elements, unpredicated)

Shift right, inserting zeroes, all elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and place the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. This instruction is unpredicated.

```
| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 |
| 16 | 15 14 13 12 | 11 10 9 | 5 4 | 0 |
| 0 0 0 0 0 1 0 0 | size 1 | Zm | 1 0 0 0 0 0 1 | Zn | Zd |
```

**Encoding**

LSR <Zd>.<T>, <Zn>.<T>, <Zm>.D

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED; 
if size == '11' then UNDEFINED; 
constant integer esize = 8 << UInt(size); 
integer n = UInt(Zn); 
integer m = UInt(Zm); 
integer d = UInt(Zd); 
```

**Assembler symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - The encoding size = 11 is reserved.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
    integer shift = Min(UInt(element2), esize);
    Elem[result, e, esize] = LSR(element1, shift);
Z[d, VL] = result;
```
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.349  LSRR

Reversed shift right, inserting zeroes, active elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

Encoding

LSRR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn>          Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>            Is the size specifier, encoded in the "size" field. It can have the following values:
                B   when size = 00
                H   when size = 01
                S   when size = 10
                D   when size = 11
<Pg>           Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>           Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
        integer shift = Min(UInt(element1), esize);
        Elem[result, e, esize] = LSR(element2, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.350 MAD

Multiply the corresponding active elements of the first and second source vectors and add to elements of the third (addend) vector. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>Zm</td>
<td>1 1 0</td>
<td>Pg</td>
<td>Za</td>
<td>Zdn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

MAD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean sub_op = FALSE;
```

**Assembler symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Za>` Is the name of the third source scalable vector register, encoded in the "Za" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Pg, PL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer product = element1 * element2;
    if sub_op then
```
$$\text{Elem[result, e, esize]} = \text{Elem[operand3, e, esize]} - \text{product};$$
else
$$\text{Elem[result, e, esize]} = \text{Elem[operand3, e, esize]} + \text{product};$$
else
$$\text{Elem[result, e, esize]} = \text{Elem[operand1, e, esize]};$$

$$Z[dn, VL] = \text{result};$$

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.351 MATCH
This instruction compares each active 8-bit or 16-bit character in the first source vector with all of the characters in
the corresponding 128-bit segment of the second source vector. Where the first source element detects any matching
characters in the second segment it places true in the corresponding element of the destination predicate, otherwise
false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C)
condition flags based on the predicate result, and the V flag to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and
enabled at the current Exception level.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Encoding
MATCH <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Decode for this encoding
if !HaveSVE2() then UNDEFINED;
if size IN {'1x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Pd);
integer n = UInt(Zn);
integer m = UInt(Zm);

Assembler symbols
<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in the "size<0>" field. It can have the following values:
S when size<0> = 0
H when size<0> = 1
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(PL) mask = Pg[PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(PL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer segmentbase = e - (e MOD eltspersegment);
    ElemP[result, e, esize] = '0';
SVE Instruction Descriptions

C8.2 Alphabetical list of SVE instructions

bits(esize) element1 = Elem[operand1, e, esize];
for i = segmentbase to (segmentbase + eltspersegment) - 1
  bits(esize) element2 = Elem[operand2, i, esize];
  if element1 == element2 then
    ElemP[result, e, esize] = '1';
  else
    ElemP[result, e, esize] = '0';

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
C8.2.352 MLA (indexed)

Multiply all integer elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively added to the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

16-bit

![MLA 16-bit Encoding](image)

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

32-bit

![MLA 32-bit Encoding](image)

Encoding

MLA <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
64-bit


Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm> For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.
For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
  integer segmentbase = e - (e MOD eltspersegment);
  integer s = segmentbase + index;
  integer element1 = UInt(Elem[operand1, e, esize]);
  integer element2 = UInt(Elem[operand2, s, esize]);
  bits(esize) product = (element1 * element2)<esize-1:0>;
  Elem[result, e, esize] = Elem[result, e, esize] + product;
Z[da, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.353 MLA (vectors)

Multiply the corresponding active elements of the first and second source vectors and add to elements of the third source (addend) vector. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0]
0 0 0 0 0 0 1 0 0 size 0 Zm 0 1 0 Pg Zn Zda
```

Encoding

MLA <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean sub_op = FALSE;

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer product = element1 * element2;
    if sub_op then
```
Subject to the terms of an applicable ARM license (see https://www.arm.com/en/termsofuse), this information may be used by an individual for personal use only (subject to the payment of applicable fees).
C8.2.354   MLS (indexed)

Multiply all integer elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively subtracted from the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

16-bit

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 14 13 12 11 10 9 5 4 0]
0 1 0 0 0 1 0 0 0 1 | i3l | Zm 0 0 0 0 1 1 | Zn | Zda
```

Encoding


Decode for this encoding

```java
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

32-bit

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 14 13 12 11 10 9 5 4 0]
0 1 0 0 0 1 0 0 0 1 0 1 1 | i2 | Zm 0 0 0 0 1 1 | Zn | Zda
```

Encoding

MLS <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

Decode for this encoding

```java
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```
64-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 1 1 1</td>
<td>Zm 0 0 0 0 1 1</td>
<td>Zn</td>
<td>Zda</td>
</tr>
</tbody>
</table>

size<1>
size<0>

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm> For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.
For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
integer segmentbase = e - (e MOD eltspersegment);
integer s = segmentbase + index;
integer element1 = UInt(Elem[operand1, e, esize]);
integer element2 = UInt(Elem[operand2, s, esize]);
bits(esize) product = (element1 * element2)<esize-1:0>;
Elem[result, e, esize] = Elem[result, e, esize] - product;
Z[da, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.355   MLS (vectors)

Multiply the corresponding active elements of the first and second source vectors and subtract from elements of the third source (addend) vector. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

![Encoding Diagram](image)

**Encoding**

MLS <Zda>,<T>, <Pg>/M, <Zn>,<T>, <Zm>,<T>

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
c constant integer esize = 8 << UInt(size);
c integer g = UInt(Pg);
c integer n = UInt(Zn);
c integer m = UInt(Zm);
c integer da = UInt(Zda);
boolean sub_op = TRUE;
```

**Assembler symbols**

- `<Zda>` is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- `<T>` is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
c constant integer VL = CurrentVL;
c constant integer PL = VL DIV 8;
c constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer product = element1 * element2;
    if sub_op then
```

00000100

**Assemblers Symbols**

- `<Zda>` is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- `<T>` is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
c constant integer VL = CurrentVL;
c constant integer PL = VL DIV 8;
c constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer product = element1 * element2;
    if sub_op then
```

00000100

**Assemblers Symbols**

- `<Zda>` is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- `<T>` is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.
\[
\text{Elem}[\text{result}, e, \text{esize}] = \text{Elem}[\text{operand3}, e, \text{esize}] - \text{product};
\]
else
\[
\text{Elem}[\text{result}, e, \text{esize}] = \text{Elem}[\text{operand3}, e, \text{esize}] + \text{product};
\]
else
\[
\text{Elem}[\text{result}, e, \text{esize}] = \text{Elem}[\text{operand3}, e, \text{esize}];
\]
\[
\text{Z}[\text{da}, \text{VL}] = \text{result};
\]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.356   MOV

Unconditionally broadcast the logical bitmask immediate into each element of the destination vector. This instruction is unpredicated. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits.

This instruction is an alias of the DUPM instruction. This means that:

- The encodings in this description are named to match the encodings of DUPM.
- The description of DUPM gives the operational pseudocode for this instruction.

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 |  |  | 5 4 | 0 |
| 0 0 0 0 0 | 1 0 1 1 0 0 0 0 | imm13 | Zd |
```

**Encoding**

MOV <Zd>.<T>, #<const>

is equivalent to

DUPM <Zd>.<T>, #<const>

and is the preferred disassembly when SVEMoveMaskPreferred(imm13).

**Assembler symbols**

- `<Zd>` is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` is the size specifier, encoded in the "imm13<12>:imm13<5:0>" field. It can have the following values:
  - `S` when imm13<12> = 0, imm13<5:0> = 0xxxxx
  - `H` when imm13<12> = 0, imm13<5:0> = 10xxxx
  - `B` when imm13<12> = 0, imm13<5:0> = 110xxx
  - `B` when imm13<12> = 0, imm13<5:0> = 1110xx
  - `B` when imm13<12> = 0, imm13<5:0> = 11110x
  - `D` when imm13<12> = 1, imm13<5:0> = xxxxxx

The following encodings are reserved:

- imm13<12> = 0, imm13<5:0> = 111110.
- imm13<12> = 0, imm13<5:0> = 111111.

- `<const>` is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

**Operation**

The description of DUPM gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
— The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.357 MOV

Read all elements from the source predicate and place in the destination predicate. This instruction is unpredicated. Does not set the condition flags.

This instruction is an alias of the ORR (predicates) instruction. This means that:

- The encodings in this description are named to match the encodings of ORR (predicates).
- The description of ORR (predicates) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>19 18 17 16</th>
<th>09 08</th>
<th>05 04 03 02 01 00</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 10 00 1 0 1 1 0 0 0</td>
<td>Pm 0 1</td>
<td>Pg 0</td>
<td>Pn 0</td>
</tr>
</tbody>
</table>

**Encoding**

MOV <Pd>.B, <Pn>.B

is equivalent to


and is the preferred disassembly when $S == '0' \&\& Pn == Pm \&\& Pm == Pg.

**Assembler symbols**

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

**Operation**

The description of ORR (predicates) gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.358 **MOV (immediate, predicated, merging)**

Move a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This instruction is an alias of the CPY (immediate, merging) instruction. This means that:

- The encodings in this description are named to match the encodings of CPY (immediate, merging).
- The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

| 31 30 29 28|27 26 25 24|23 22 21 20|19 16 15 14 13 12| 5 4 0 |
|---|---|---|---|---|---|
| 0 0 0 0 1 0 1 | size 0 1 | Pg 0 1 | sh | imm8 | Zd |

**Encoding**

MOV <Zd>.<T>, <Pg>/M, #<imm>{, <shift>}

is equivalent to

CPY <Zd>.<T>, <Pg>/M, #<imm>{, <shift>}

and is always the preferred disassembly.

**Assembler symbols**

- **<Zd>**
  - Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>**
  - Is the size specifier, encoded in the "size" field. It can have the following values:
    - B when size = 00
    - H when size = 01
    - S when size = 10
    - D when size = 11
- **<Pg>**
  - Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- **<imm>**
  - Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.
- **<shift>**
  - Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:
    - LSL #0 when sh = 0
    - LSL #8 when sh = 1

**Operation**

The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.359\ MOV (immediate, predicated, zeroing)

Move a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register are set to zero.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This instruction is an alias of the CPY (immediate, zeroing) instruction. This means that:

- The encodings in this description are named to match the encodings of CPY (immediate, zeroing).
- The description of CPY (immediate, zeroing) gives the operational pseudocode for this instruction.

Encoding

\[
\text{MOV } <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>}
\]

is equivalent to

\[
\text{CPY } <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>}
\]

and is always the preferred disassembly.

Assembler symbols

\(<Zd>\) Is the name of the destination scalable vector register, encoded in the "Zd" field.

\(<T>\) Is the size specifier, encoded in the "size" field. It can have the following values:

- B when \(\text{size} = 00\)
- H when \(\text{size} = 01\)
- S when \(\text{size} = 10\)
- D when \(\text{size} = 11\)

\(<Pg>\) Is the name of the governing scalable predicate register, encoded in the "Pg" field.

\(<imm>\) Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

\(<\text{shift}>\) Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:

- LSL #0 when \(\text{sh} = 0\)
- LSL #8 when \(\text{sh} = 1\)

Operation

The description of CPY (immediate, zeroing) gives the operational pseudocode for this instruction.
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.360  MOV (immediate, unpredicated)

Unconditionally broadcast the signed integer immediate into each element of the destination vector. This instruction is unpredicated.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This instruction is an alias of the DUP (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of DUP (immediate).
- The description of DUP (immediate) gives the operational pseudocode for this instruction.

Encoding

MOV <Zd>.<T>, #<imm>{, <shift>}

is equivalent to

DUP     <Zd>.<T>, #<imm>{, <shift>}

and is always the preferred disassembly.

Assemble symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

<imm> Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:

LSL #0 when sh = 0
LSL #8 when sh = 1

Operation

The description of DUP (immediate) gives the operational pseudocode for this instruction.
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.361   MOV (predicate, predicated, merging)

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register remain unmodified. Does not set the condition flags.

This instruction is an alias of the SEL (predicates) instruction. This means that:

• The encodings in this description are named to match the encodings of SEL (predicates).
• The description of SEL (predicates) gives the operational pseudocode for this instruction.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 | 10 9 8 | 5 4 3 0 ]
S
0 0 1 0 0 1 0 1 0 0 0 0 0 0 1 0 1 1 1 1 0 1 1
```

**Encoding**

MOV <Pd>.B, <Pg>/M, <Pn>.B

is equivalent to


and is the preferred disassembly when Pd == Pm.

**Assembler symbols**

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

**Operation**

The description of SEL (predicates) gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.
C8.2.362   MOV (predicate, predicated, zeroing)

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is an alias of the AND (predicates) instruction. This means that:

- The encodings in this description are named to match the encodings of AND (predicates).
- The description of AND (predicates) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>MOV &lt;Pd&gt;.B, &lt;Pg&gt;/Z, &lt;Pn&gt;.B</th>
</tr>
</thead>
</table>

is equivalent to and is the preferred disassembly when \( S == '0' \) && \( Pn == Pm \).

**Assembler symbols**

- \(<Pd>\) Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- \(<Pg>\) Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- \(<Pn>\) Is the name of the first source scalable predicate register, encoded in the "Pn" field.

**Operation**

The description of AND (predicates) gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.363 MOV (scalar, predicated)

Move the general-purpose scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is an alias of the CPY (scalar) instruction. This means that:

- The encodings in this description are named to match the encodings of CPY (scalar).
- The description of CPY (scalar) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV &lt;Zd&gt;.&lt;T&gt;, &lt;Pg&gt;/M, &lt;R&gt;&lt;n</td>
<td>SP&gt;</td>
</tr>
</tbody>
</table>

**Assembler symbols**

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>`: Is the size specifier, encoded in the "size" field. It can have the following values:
  - `B` when size = 00
  - `H` when size = 01
  - `S` when size = 10
  - `D` when size = 11
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<R>`: Is a width specifier, encoded in the "size" field. It can have the following values:
  - `W` when size = 01
  - `w` when size = x0
  - `X` when size = 11
- `<n|SP>`: Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

**Operation**

The description of CPY (scalar) gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
— The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.364 MOV (scalar, unpredicated)

Unconditionally broadcast the general-purpose scalar source register into each element of the destination vector. This instruction is unpredicated.

This instruction is an alias of the DUP (scalar) instruction. This means that:

- The encodings in this description are named to match the encodings of DUP (scalar).
- The description of DUP (scalar) gives the operational pseudocode for this instruction.

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 18 17 16] [15 14 13 12] [11 10 9] | 5 4 | 0 |
0 0 0 0 0 1 0 1 | size | 1 0 0 0 0 0 0 0 1 1 1 0 | Rn | Zd |
```

**Encoding**

MOV <Zd>.<T>, <R><n|SP>

is equivalent to

DUP <Zd>.<T>, <R><n|SP>

and is always the preferred disassembly.

**Assembler symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - `B` when size = `00`
  - `H` when size = `01`
  - `S` when size = `10`
  - `D` when size = `11`
- `<R>` Is a width specifier, encoded in the "size" field. It can have the following values:
  - `W` when size = `01`
  - `W` when size = `x0`
  - `X` when size = `11`
- `<n|SP>` Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

**Operation**

The description of DUP (scalar) gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.365 MOV (SIMD&FP scalar, predicated)

Move the SIMD & floating-point scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is an alias of the CPY (SIMD&FP scalar) instruction. This means that:

- The encodings in this description are named to match the encodings of CPY (SIMD&FP scalar).
- The description of CPY (SIMD&FP scalar) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>MOV &lt;Zd&gt;.&lt;T&gt;, &lt;Pg&gt;/M, &lt;V&gt;&lt;n&gt;</th>
</tr>
</thead>
</table>

is equivalent to

<table>
<thead>
<tr>
<th>Encoding</th>
<th>CPY &lt;Zd&gt;.&lt;T&gt;, &lt;Pg&gt;/M, &lt;V&gt;&lt;n&gt;</th>
</tr>
</thead>
</table>

and is always the preferred disassembly.

### Assembler symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<V>` Is a width specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<n>` Is the number [0-31] of the source SIMD&FP register, encoded in the "Vn" field.

### Operation

The description of CPY (SIMD&FP scalar) gives the operational pseudocode for this instruction.

### Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.366  **MOV (SIMD&FP scalar, unpredicated)**

Unconditionally broadcast the SIMD&FP scalar into each element of the destination vector. This instruction is unpredicated.

This instruction is an alias of the DUP (indexed) instruction. This means that:

- The encodings in this description are named to match the encodings of DUP (indexed).
- The description of DUP (indexed) gives the operational pseudocode for this instruction.

### Encoding

\[
\text{MOV } <Zd>.<T>, <Zn>.<T>[<imm>]
\]

is equivalent to

\[
\text{DUP } <Zd>.<T>, <Zn>.<T>[<imm>]
\]

and is the preferred disassembly when \(\text{BitCount}(\text{imm2:tsz}) > 1\).

### Encoding

\[
\text{MOV } <Zd>.<T>, <V><n>
\]

is equivalent to

\[
\text{DUP } <Zd>.<T>, <Zn>.<T>[0]
\]

and is the preferred disassembly when \(\text{BitCount}(\text{imm2:tsz}) == 1\).

### Assembler symbols

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in the "tsz" field. It can have the following values:
  - B when tsz = xxxx1
  - H when tsz = xxx10
  - S when tsz = xx100
  - D when tsz = x1000
  - Q when tsz = 10000
  - The encoding tsz = 00000 is reserved.
- **<Zn>** Is the name of the source scalable vector register, encoded in the "Zn" field.
- **<imm>** Is the immediate index, in the range 0 to one less than the number of elements in 512 bits, encoded in "imm2:tsz".
- **<V>** Is a width specifier, encoded in the "tsz" field. It can have the following values:
  - B when tsz = xxxx1
  - H when tsz = xxx10
  - S when tsz = xx100
  - D when tsz = x1000
Q  when tsz = 10000
The encoding tsz = 00000 is reserved.

<n> Is the number [0-31] of the source SIMD&FP register, encoded in the "Zn" field.

Operation

The description of DUP (indexed) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.367   MOV (vector, predicated)

Move elements from the source vector to the corresponding elements of the destination vector. Inactive elements in
the destination vector register remain unmodified.

This instruction is an alias of the SEL (vectors) instruction. This means that:

- The encodings in this description are named to match the encodings of SEL (vectors).
- The description of SEL (vectors) gives the operational pseudocode for this instruction.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 13</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1 size 1</td>
<td>Zm</td>
<td>1 1</td>
<td>Pv</td>
</tr>
</tbody>
</table>
```

**Encoding**

MOV <Zd>.<T>, <Pv>/M, <Zn>.<T>

is equivalent to

SEL <Zd>.<T>, <Pv>, <Zn>.<T>, <Zd>.<T>

and is the preferred disassembly when Zd == Zm.

**Assembler symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Pv>` Is the name of the vector select predicate register, encoded in the "Pv" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.

**Operation**

The description of SEL (vectors) gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.368 MOV (vector, unpredicated)

Move vector register. This instruction is unpredicated.

This instruction is an alias of the ORR (vectors, unpredicated) instruction. This means that:

- The encodings in this description are named to match the encodings of ORR (vectors, unpredicated).
- The description of ORR (vectors, unpredicated) gives the operational pseudocode for this instruction.

```
[31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 13 12|11 10 9 | | 5 4 | | 0 |
0 0 0 0 0 1 0 0 0 1 1 | Zm 0 0 1 1 0 0 | Zn | Zd
```

**Encoding**

MOV <Zd>.D, <Zn>.D

is equivalent to

ORR <Zd>.D, <Zn>.D, <Zn>.D

and is the preferred disassembly when Zn == Zm.

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

**Operation**

The description of ORR (vectors, unpredicated) gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.369   MOVPRFX (predicated)

The predicated MOVPRFX instruction is a hint to hardware that the instruction may be combined with the destructive instruction which follows it in program order to create a single constructive operation. Since it is a hint it is also permitted to be implemented as a discrete vector copy, and the result of executing the pair of instructions with or without combining is identical. The choice of combined versus discrete operation may vary dynamically.

Unless the combination of a constructive operation with merging predication is specifically required, it is strongly recommended that for performance reasons software should prefer to use the zeroing form of predicated MOVPRFX or the unpredicated MOVPRFX instruction.

Although the operation of the instruction is defined as a simple predicated vector copy, it is required that the prefixed instruction at PC+4 must be an SVE destructive binary or ternary instruction encoding, or a unary operation with merging predication, but excluding other MOVPRFX instructions. The prefixed instruction must specify the same predicate register, and have the same maximum element size (ignoring a fixed 64-bit "wide vector" operand), and the same destination vector as the MOVPRFX instruction. The prefixed instruction must not use the destination register in any other operand position, even if they have different names but refer to the same architectural register state. Any other use is UNPREDICTABLE.

Encoding

MOVPRFX <Zd>.<T>, <Pg>/<ZM>, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean merging = (M == '1');

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
    B when size = 00
    H when size = 01
    S when size = 10
    D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<ZM> Is the predication qualifier, encoded in the "M" field. It can have the following values:
    Z when M = 0
    M when M = 1

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) dest = Z[d, VL];
bits(VL) result;

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand1, e, esize];
    Elem[result, e, esize] = element;
  elsif merging then
    Elem[result, e, esize] = Elem[dest, e, esize];
  else
    Elem[result, e, esize] = Zeros(esize);

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.370 MOVPRFX (unpredicated)

The unpredicated MOVPRFX instruction is a hint to hardware that the instruction may be combined with the destructive instruction which follows it in program order to create a single constructive operation. Since it is a hint it is also permitted to be implemented as a discrete vector copy, and the result of executing the pair of instructions with or without combining is identical. The choice of combined versus discrete operation may vary dynamically.

Although the operation of the instruction is defined as a simple unpredicated vector copy, it is required that the prefixed instruction at PC+4 must be an SVE destructive binary or ternary instruction encoding, or a unary operation with merging predication, but excluding other MOVPRFX instructions. The prefixed instruction must specify the same destination vector as the MOVPRFX instruction. The prefixed instruction must not use the destination register in any other operand position, even if they have different names but refer to the same architectural register state. Any other use is UNPREDICTABLE.

Encoding

MOVPRFX <Zd>, <Zn>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(VL) result = Z[n, VL];
Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.371 MOVS (predicated)

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is an alias of the ANDS instruction. This means that:

- The encodings in this description are named to match the encodings of ANDS.
- The description of ANDS gives the operational pseudocode for this instruction.

**Encoding**

MOVS <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to


and is the preferred disassembly when $S == '1'$ && $Pn == Pm$.

**Assembler symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.

**Operation**

The description of ANDS gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.372   **MOVS (unpredicated)**

Read all elements from the source predicate and place in the destination predicate. This instruction is unpredicated. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is an alias of the **ORRS** instruction. This means that:

- The encodings in this description are named to match the encodings of **ORRS**.
- The description of **ORRS** gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[19]</th>
<th>16</th>
<th>14</th>
<th>13</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 1 1 0 0</td>
<td>Pm</td>
<td>0 1</td>
<td>Pg</td>
<td>0</td>
<td>Pn</td>
<td>0</td>
<td>Pd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

MOVS <Pd>.B, <Pn>.B  
is equivalent to  
and is the preferred disassembly when $S = '1'$ & $Pn = Pm$ & $Pm = Pg$.

**Assembler symbols**

- **<Pd>**  Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- **<Pn>**  Is the name of the first source scalable predicate register, encoded in the "Pn" field.

**Operation**

The description of **ORRS** gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.373   MSB

Multiply the corresponding active elements of the first and second source vectors and subtract from elements of the third (addend) vector. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

Encoding

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0</td>
<td>size 0</td>
<td>Zm 1 1</td>
<td>Pg</td>
<td>Za</td>
<td>Zdn</td>
<td></td>
</tr>
</tbody>
</table>

Decoding for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean sub_op = TRUE;

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer product = element1 * element2;
    if sub_op then
      result[e] = if element1 < 0 then element2 + product else element2 - product;
    else
      result[e] = element1 * element2;
  end if;
end for;
Elem[result, e, esize] = Elem[operand3, e, esize] - product;
else
   Elem[result, e, esize] = Elem[operand3, e, esize] + product;
else
   Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**C8.2.374 MUL (immediate)**

Multiply by an immediate each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12] | 5 4 | 0 |
0 0 1 0 0 1 0 1 1 1 0 0 0 0 1 1 0 1 0 | imm8 | Zdn |
```

**Encoding**

MUL <Zdn>,<T>, <Zdn>,<T>, #<imm>

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = SInt(imm8);
```

**Assembler symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<imm>` Is the signed immediate operand, in the range -128 to 127, encoded in the "imm8" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    Elem[result, e, esize] = (element1 * imm)<esize-1:0>;
Z[dn, VL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.375 MUL (indexed)

Multiply all integer elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The results are placed in the corresponding elements of the destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

16-bit

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|-----------|-----------|-----------|-------|-------|-------|
| 0 1 0 0 0 | 1 0 0 | 0 1 3i | Zm | 1 1 1 | 1 1 0 | Zn | Zd |

i3h
```

Encoding


Decode for this encoding

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

32-bit

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|-----------|-----------|-----------|-------|-------|-------|
| 0 1 0 0 0 | 1 0 0 | 1 0 1 | i2 | Zm | 1 1 1 | 1 1 0 | Zn | Zd |

size<1>
size<0>
```

Encoding

MUL <Zd>.S, <Zn>.S, <Zm>.S[<imm>]

Decode for this encoding

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```
64-bit

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 |
| 0 1 0 0 0 1 0 0|1 1 1 1| Zm | 1 1 1 1 0| Zn | Zd |
```

Encoding

MUL <Zd>.D, <Zn>.D, <Zm>.D[<imm>]

Decode for this encoding

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler symbols

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<Zn>`: Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>`: For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
  For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- `<imm>`: For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
  For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.
  For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation for all encodings

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  integer segmentbase = e - (e MOD eltspersegment);
  integer s = segmentbase + index;
  integer element1 = UInt(Elem[operand1, e, esize]);
  integer element2 = UInt(Elem[operand2, s, esize]);
  integer res = element1 * element2;
  Elem[result, e, esize] = res<esize-1:0>;
Z[d, VL] = result;
```
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
MUL (vectors, predicated)

Multiply active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

MUL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  integer element2 = UInt(Elem[operand2, e, esize]);
  if Elem[mask, e, esize] == '1' then
    integer product = element1 * element2;
    Elem[result, e, esize] = product<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.377   MUL (vectors, unpredicated)

Multiply all elements of the first source vector by corresponding elements of the second source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Encoding**

MUL <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVEZ() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler symbols**

<Zd>    Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>    Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer product = element1 * element2;
    Elem[result, e, esize] = product<esize-1:0>;
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.378 NAND

Bitwise NAND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

![Encoding]

**Encoding**


**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;

constant integer esize = 8;

integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);

boolean setflags = FALSE;

**Assembler symbols**

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

**Operation**

CheckSVEEnabled();

constant integer VL = CurrentVL;

constant integer PL = VL DIV 8;

constant integer elements = VL DIV esize;

bits(PL) mask = P[g, PL];

bits(PL) operand1 = P[n, PL];

bits(PL) operand2 = P[m, PL];

bits(PL) result;

for e = 0 to elements-1

bit element1 = ElemP[operand1, e, esize];

bit element2 = ElemP[operand2, e, esize];

if ElemP[mask, e, esize] == '1' then

ElemP[result, e, esize] = NOT(element1 AND element2);

else

ElemP[result, e, esize] = '0';

if setflags then

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);

P[d, PL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.379 NANDS

Bitwise NAND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 AND element2);
    else
        ElemP[result, e, esize] = '0';
    if setflags then
        PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.380  NBSL

Selects bits from the first source vector where the corresponding bit in the third source vector is '1', and from the second source vector where the corresponding bit in the third source vector is '0'. The inverted result is placed destructively in the destination and first source vector. This instruction is unpredicated.

```
| 31 30 29 28|27 26 25 24|23 22 21 20 | 16|15 14 13 12|11 10 9 | 5 4  | 0  |
|------|------|------|  |------|------|  |  |
| 0 0 0 0 | 0 0 1 1 | Zm  | 0 0 1 1 | Zk  | Zdn |
```

**Encoding**


**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);

**Assembler symbols**

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zk>  Is the name of the third source scalable vector register, encoded in the "Zk" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];
Z[dn, VL] = NOT((operand1 AND operand3) OR (operand2 AND NOT(operand3)));

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.381   NEG

Negate the signed integer value in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Encoding
NEG <Zd>.<T>, <Pg>/M, <Zn>.<T>

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd>          Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>           Is the size specifier, encoded in the "size" field. It can have the following values:
                B when size = 00
                H when size = 01
                S when size = 10
                D when size = 11

<Pg>          Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn>          Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1'
        integer element = SInt(Elem[operand, e, esize]);
        element = -element;
        Elem[result, e, esize] = element<esize-1:0>;
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
This instruction compares each active 8-bit or 16-bit character in the first source vector with all of the characters in the corresponding 128-bit segment of the second source vector. Where the first source element detects no matching characters in the second segment it places true in the corresponding element of the destination predicate, otherwise false. Inactive elements in the destination predicate register are set to zero. Sets the \texttt{FIRST (N)}, \texttt{NONE (Z)}, \texttt{!LAST (C)} condition flags based on the predicate result, and the V flag to zero.

This instruction is illegal when executed in Streaming SVE mode, unless \texttt{FEAT\_SME\_FA64} is implemented and enabled at the current Exception level.

\begin{verbatim}
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</th>
<th>15 14 13 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1</td>
<td>size</td>
</tr>
</tbody>
</table>
\end{verbatim}

**Encoding**

\texttt{NMATCH <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>}

**Decode for this encoding**

if \texttt{!HaveSVE2()} then UNDEFINED;
if \texttt{size IN \{"1x"\}} then UNDEFINED;
constant integer esize = \& < UInt(size);
integer g = UInt(Pg);
integer d = UInt(Pd);
integer n = UInt(Zn);
integer m = UInt(Zm);

**Assembler symbols**

\texttt{<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.}

\texttt{<T> Is the size specifier, encoded in the "size<0>" field. It can have the following values:}

- \texttt{B} when \texttt{size<0> = 0}
- \texttt{H} when \texttt{size<0> = 1}

\texttt{<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.}

\texttt{<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.}

\texttt{<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.}

**Operation**

\texttt{CheckNonStreamingSVEEnabled();}
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if \texttt{AnyActiveElement(mask, esize)} then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if \texttt{AnyActiveElement(mask, esize)} then Z[m, VL] else Zeros(VL);
bits(PL) result;

for \texttt{e = 0} to \texttt{elements-1}
  if ElemP[mask, e, esize] == '1'
    integer segmentbase = e - (e MOD eltspersegment);
    ElemP[result, e, esize] = '1';

\texttt{01000101  size 1  Zm 1 0 0  Pg  Zn 1  Pd}
bits(esize) element1 = Elem[operand1, e, esize];
for i = segmentbase to (segmentbase + eltspersegment) - 1
  bits(esize) element2 = Elem[operand2, i, esize];
  if element1 == element2 then
    ElemP[result, e, esize] = '0';
  else
    ElemP[result, e, esize] = '0';
end for
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
C8.2.383  NOR

Bitwise NOR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 OR element2);
    else
        ElemP[result, e, esize] = '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.
C8.2.384  NORS

Bitwise NOR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

Encoding

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler symbols
<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 OR element2);
    else
        ElemP[result, e, esize] = '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.385  NOT (predicate)

Bitwise invert each active element of the source predicate, and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is an alias of the EOR (predicates) instruction. This means that:

- The encodings in this description are named to match the encodings of EOR (predicates).
- The description of EOR (predicates) gives the operational pseudocode for this instruction.

```
<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[19]</th>
<th>[16 15 14 13]</th>
<th>[10 9 8]</th>
<th>[5 4 3]</th>
<th>[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pm</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>1</td>
<td>Pn</td>
<td>0</td>
<td>Pd</td>
</tr>
</tbody>
</table>
```

**Encoding**

NOT <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to


and is the preferred disassembly when Pm == Pg.

**Assembler symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.

**Operation**

The description of EOR (predicates) gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.386   NOT (vector)

Bitwise invert each active element of the source vector, and place the results in the corresponding elements of the
destination vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0</td>
<td>0 1 1 1 0 0 1</td>
<td>0</td>
<td>5 4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

NOT <Zd>.<T>, <Pg>/M, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 <<< UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd>    Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>    Is the size specifier, encoded in the "size" field. It can have the following values:
    B    when size = 00
    H    when size = 01
    S    when size = 10
    D    when size = 11
<Pg>    Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>    Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = NOT element;

Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.387 NOTS

Bitwise invert each active element of the source predicate, and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is an alias of the EORS instruction. This means that:

- The encodings in this description are named to match the encodings of EORS.
- The description of EORS gives the operational pseudocode for this instruction.

**Encoding**

NOTS \(<Pd>.B, <Pg>/Z, <Pn>.B\)

is equivalent to

EORS \(<Pd>.B, <Pg>/Z, <Pn>.B, <Pg>.B\)

and is the preferred disassembly when \(Pm == Pg\).

**Assembler symbols**

- \(<Pd>\) Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- \(<Pg>\) Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- \(<Pn>\) Is the name of the first source scalable predicate register, encoded in the "Pn" field.

**Operation**

The description of EORS gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.388 ORN (immediate)

Bitwise inclusive OR an inverted immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is a pseudo-instruction of the ORR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of ORR (immediate) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 1 0</td>
<td>0 0 0 0 0</td>
<td>imm13</td>
<td>Zdn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

ORN `<Zdn>.<T>, <Zdn>.<T>, #<const>`

is equivalent to

ORR `<Zdn>.<T>, <Zdn>.<T>, #(-<const> - 1)`

and is never the preferred disassembly.

**Assembler symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "imm13<12>:imm13<5:0>" field. It can have the following values:
  - S when `imm13<12> = 0, imm13<5:0> = 0xxxxx`
  - H when `imm13<12> = 0, imm13<5:0> = 10xxxx`
  - B when `imm13<12> = 0, imm13<5:0> = 110xxx`
  - B when `imm13<12> = 0, imm13<5:0> = 1110xx`
  - B when `imm13<12> = 0, imm13<5:0> = 11110x`
  - D when `imm13<12> = 1, imm13<5:0> = xxxxxx`

The following encodings are reserved:

- `imm13<12> = 0, imm13<5:0> = 111110`
- `imm13<12> = 0, imm13<5:0> = 111111`

- `<const>` Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

**Operation**

The description of ORR (immediate) gives the operational pseudocode for this instruction.
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.389  ORN (predicates)

Bitwise inclusive OR inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

ENCODING


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Assembler symbols

<Pd>       Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>       Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>       Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>       Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
for e = 0 to elements-1
  bit element1 = ElemP[operand1, e, esize];
  bit element2 = ElemP[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    ElemP[result, e, esize] = element1 OR (NOT element2);
  else
    ElemP[result, e, esize] = '0';
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.
C8.2.390   ORNS

Bitwise inclusive OR inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14|13| 10| 9| 8| 5| 4| 3| 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0| 0| 1| 0| 1| 0| 1| 0| 1| 0| 1| 0| 1| 0| 0 |

S

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP(operand1, e, esize);
    bit element2 = ElemP(operand2, e, esize);
    if ElemP(mask, e, esize) == '1' then
        ElemP(result, e, esize) = element1 OR (NOT element2);
    else
        ElemP(result, e, esize) = '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.391 ORR (immediate)

Bitwise inclusive OR an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction ORN (immediate). The pseudo-instruction is never the preferred disassembly.

Encoding

```
| 31 30 29 28| 27 26 25 24| 23 22 21 20| 19 18 17 | 5 4 | 0 |
| 0 0 0 0 1 0 1 | 0 0 0 0 0 0 | imm13 | Zdn |
```

Decode for this encoding

```
if !HaveSVE() & !HaveSME() then UNDEFINED;
integer dn = UInt(Zdn);
bits(64) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, 64);
```

Assembler symbols

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "imm13<12>:imm13<5:0>" field. It can have the following values:
  - S when imm13<12> = 0, imm13<5:0> = 0xxxxx
  - H when imm13<12> = 0, imm13<5:0> = 10xxxx
  - B when imm13<12> = 0, imm13<5:0> = 110xxx
  - B when imm13<12> = 0, imm13<5:0> = 1110xx
  - B when imm13<12> = 0, imm13<5:0> = 11110x
  - D when imm13<12> = 1, imm13<5:0> = xxxxxx

The following encodings are reserved:
- `imm13<12> = 0, imm13<5:0> = 111110`.  
- `imm13<12> = 0, imm13<5:0> = 111111`.

- `<const>` Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 64;
bits(VL) operand = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(64) element1 = Elem[operand, e, 64];
  Elem[result, e, 64] = element1 OR imm;
```
$Z[dn, VL] = \text{result};$

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.392  **ORR (predicates)**

Bitwise inclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is used by the alias MOV. See *Alias conditions* for details of when each alias is preferred.

**Encoding**

```
```

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

**Alias conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>S == '0' &amp;&amp; Pn == Pm &amp;&amp; Pm == Pg</td>
</tr>
</tbody>
</table>

**Assembler symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` Is the name of the second source scalable predicate register, encoded in the "Pm" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP(operand1, e, esize);
    bit element2 = ElemP(operand2, e, esize);
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 OR element2;
```
else
    ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
    P[d, PL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.
C8.2.393 ORR (vectors, predicated)

Bitwise inclusive OR active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10  9 |  5  4 |  0 ]
0 0 0 0 0 1 0 0 size 0 1 1 0 0 0 0 0 Pg Zm Zdn
```

Encoding

ORR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = element1 OR element2;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.394  ORR (vectors, unpredicated)

Bitwise inclusive OR all elements of the second source vector with corresponding elements of the first source vector and place the first in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction is used by the alias MOV (vector, unpredicated). See Alias conditions for details of when each alias is preferred.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1</td>
<td>0 0 1 1 0 0</td>
<td>Zm</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

ORR <Zd>.D, <Zn>.D, <Zm>.D

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Alias conditions

<table>
<thead>
<tr>
<th>Alias is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (vector, unpredicated) Zn == Zm</td>
</tr>
</tbody>
</table>

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
Z[d, VL] = operand1 OR operand2;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.395  ORRS

Bitwise inclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the alias MOVS (unpredicated). See *Alias conditions* for details of when each alias is preferred.

```
Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

**Alias conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVS (unpredicated)</td>
<td>S == ’1’ &amp;&amp; Pn == Pm &amp;&amp; Pm == Pg</td>
</tr>
</tbody>
</table>

**Assembler symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` Is the name of the second source scalable predicate register, encoded in the "Pm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP(operand1, e, esize);
```
bit element2 = ElemP(operand2, e, esize);
if ElemP(mask, e, esize) == '1' then
  ElemP(result, e, esize) = element1 OR element2;
else
  ElemP(result, e, esize) = '0';

if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.396   ORV

Bitwise inclusive OR horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 |  5 4 |  0 |
| 0  0  0  0 | 0  1  1  0 | 0  0  0  1 | Pg   | Zn   | Vd   |
```

**Encoding**

ORV <V><d>, <Pg>, <Zn><T>

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

**Assembler symbols**

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) result = Zeros(esize);

for e = 0 to elements-1
   if Elem[mask, e, esize] == '1'
      result = result OR Elem[operand, e, esize];
V[d, esize] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.397 PFALSE

Set all elements in the destination predicate to false.

Encoding

PFALSE <Pd>.B

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer d = UInt(Pd);

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
P[d, PL] = Zeros(PL);

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.398 PFIRST

Sets the first active element in the destination predicate to true, otherwise elements from the source predicate are passed through unchanged. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8 | 5 4 3 0 ]
0 0 1 0 0 1 0 1 0 1 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Pd Pg Pdn
```

**Encoding**

PFIRST <Pd>.B, <Pg>, <Pd>.B

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer dn = UInt(Pdn);

**Assembler symbols**

<Pd> Is the name of the source and destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) result = P[dn, PL];
integer first = -1;

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' && first == -1 then
    first = e;
  if first >= 0 then
    ElemP[result, first, esize] = '1';
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[dn, PL] = result;

**Operational information**

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.399  PMUL

Polynomial multiply over \([0, 1]\) all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9 8 7 6 5 4</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 0 0 0 1</td>
<td>0 1 1 0 0 0 1 0 1 0 1</td>
<td>Zd</td>
</tr>
<tr>
<td>size&lt;1&gt;</td>
<td>size&lt;0&gt;</td>
<td>Zn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zm</td>
</tr>
</tbody>
</table>
```

**Encoding**

PMUL <Zd>.B, <Zn>.B, <Zm>.B

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = PolynomialMult(element1, element2)<esize-1:0>;
Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.400   PMULLB

Polynomial multiply over \([0, 1]\) the corresponding even-numbered elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether the instruction variant with 64-bit source and 128-bit destination elements is implemented.

Encoding

PMULLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' && !HaveSVE2PMULL128() then UNDEFINED;
integer esize;
  case size of
    when '00' esize = 128;
    when '01' esize = 16;
    when '10' UNDEFINED;
    when '11' esize = 64;
  integer n = UInt(Zn);
  integer m = UInt(Zm);
  integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- Q when size = 00
- H when size = 01
- D when size = 11

The encoding size = 10 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:

- D when size = 00
- B when size = 01
- S when size = 11

The encoding size = 10 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
Operation

if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2*e + 0, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2*e + 0, esize DIV 2];
    Elem[result, e, esize] = PolynomialMult(element1, element2);

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.401  PMULLT

Polynomial multiply over \([0, 1]\) the corresponding odd-numbered elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether the instruction variant with 64-bit source and 128-bit destination elements is implemented.

Encoding

PMULLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' && !HaveSVE2PMULL128() then UNDEFINED;
integer esize;
  case size of
    when '00' esize = 128;
    when '01' esize = 16;
    when '10' UNDEFINED;
    when '11' esize = 64;
  endcase;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler symbols

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>`: Is the size specifier, encoded in the "size" field. It can have the following values:
  - `Q` when size = 00
  - `H` when size = 01
  - `D` when size = 11
  The encoding size = 10 is reserved.
- `<Zn>`: Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>`: Is the size specifier, encoded in the "size" field. It can have the following values:
  - `D` when size = 00
  - `B` when size = 01
  - `S` when size = 11
  The encoding size = 10 is reserved.
- `<Zm>`: Is the name of the second source scalable vector register, encoded in the "Zm" field.
### Operation

if `esize` < 128 then `CheckSVEEnabled()`; else `CheckNonStreamingSVEEnabled()`;

constant integer `VL` = `CurrentVL`;
constant integer `elements` = `VL` DIV `esize`;

bits(`VL`) `operand1` = `Z[n, VL]`;
bits(`VL`) `operand2` = `Z[m, VL]`;
bits(`VL`) `result`;

for `e` = 0 to `elements`-1
  bits(`esize` DIV 2) `element1` = `Elem[operand1, 2*e + 1, esize DIV 2]`;
  bits(`esize` DIV 2) `element2` = `Elem[operand2, 2*e + 1, esize DIV 2]`;
  `Elem[result, e, esize]` = `PolynomialMult(element1, element2)`;

`Z[d, VL]` = `result`;

### Operational information

If `FEAT_SVE2` is implemented or `FEAT_SME` is implemented, then when `PSTATE.DIT` is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.402   PNEXT

An instruction used to construct a loop which iterates over all true elements in the vector select predicate register. If all elements in the first source predicate register are false it determines the first true element in the vector select predicate register, otherwise it determines the next true element in the vector select predicate register that follows the last true element in the first source predicate register. All elements of the destination predicate register are set to false, except the element corresponding to the determined vector select element, if any, which is set to true. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

Encoding

PNEXT <Pdn>.<T>, <Pv>, <Pdn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer v = UInt(Pv);
integer dn = UInt(Pdn);

Assembler symbols

<Pdn> Is the name of the first source and destination scalable predicate register, encoded in the "Pdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

| B | when size = 00 |
| H | when size = 01 |
| S | when size = 10 |
| D | when size = 11 |

<Pv> Is the name of the vector select predicate register, encoded in the "Pv" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[v, PL];
bits(PL) operand = P[dn, PL];
bits(PL) result;
integer next = LastActiveElement(operand, esize) + 1;
while next < elements && (ElemP[mask, next, esize] == '0') do
next = next + 1;
result = Zeros(PL);
if next < elements then
ElemP[result, next, esize] = '1';
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[dn, PL] = result;

**Operational information**

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.403 PRFB (scalar plus immediate)

Contiguous prefetch of byte elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.

Encoding

PRFB <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;
integer offset = SInt(imm6);

Assembler symbols

<prfop> Is the prefetch operation specifier, encoded in the "prfop" field. It can have the following values:

PLDL1KEEP when prfop = 0000
PLDL1STRM when prfop = 0001
PLDL2KEEP when prfop = 0010
PLDL2STRM when prfop = 0011
PLDL3KEEP when prfop = 0100
PLDL3STRM when prfop = 0101
#uimm4 when prfop = x11x
PSTL1KEEP when prfop = 1000
PSTL1STRM when prfop = 1001
PSTL2KEEP when prfop = 1010
PSTL2STRM when prfop = 1011
PSTL3KEEP when prfop = 1100
PSTL3STRM when prfop = 1101

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bounds(PL) mask = P[g, PL];
bounds(64) base;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bounds(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
C8.2.404   PRFB (scalar plus scalar)

Contiguous prefetch of byte elements from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.

Encoding

PRFB <prfop>, <Pg>, [<Xn|SP>, <Xm>]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;

Assembler symbols

<prfop>  Is the prefetch operation specifier, encoded in the "prfop" field. It can have the following values:
PLDL1KEEP when prfop = 0000
PLDL1STRM when prfop = 0001
PLDL2KEEP when prfop = 0010
PLDL2STRM when prfop = 0011
PLDL3KEEP when prfop = 0100
PLDL3STRM when prfop = 0101
#uimm4 when prfop = x1lx
PSTL1KEEP when prfop = 1000
PSTL1STRM when prfop = 1001
PSTL2KEEP when prfop = 1010
PSTL2STRM when prfop = 1011
PSTL3KEEP when prfop = 1100
PSTL3STRM when prfop = 1101

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(64) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
C8.2.405  **PRFB (scalar plus vector)**

Gather prefetch of bytes from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive addresses are not prefetched from memory.

The `<prfop>` symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

**32-bit scaled offset**

```
[31 30 29 28 27 26 25 24 23 22 21 20] 16 15 14 13 12 10 9 | 5 4 3 0
1 0 0 0 0 1 0 0 0 | xs 1 | Zm 0 0 0 | Pg 0 | Rn 0 | prfop
```

**Encoding**

```
PRFB <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]
```

**Decode for this encoding**

```c
if !HaveSVE() then UNDEFINED;
const integer esize = 32;
integer g = UINT(Pg);
integer n = UINT(Rn);
integer m = UINT(Zm);
integer level = UINT(prfop<2:1>);
bool stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
const integer offs_size = 32;
bool offs_unsigned = (xs == '0');
integer scale = 0;
```

**32-bit unpacked scaled offset**

```
[31 30 29 28 27 26 25 24 23 22 21 20] 16 15 14 13 12 10 9 | 5 4 3 0
1 1 0 0 0 1 0 0 0 | xs 1 | Zm 0 0 0 | Pg 0 | Rn 0 | prfop
```

**Encoding**

```
PRFB <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]
```

**Decode for this encoding**

```c
if !HaveSVE() then UNDEFINED;
const integer esize = 64;
integer g = UINT(Pg);
integer n = UINT(Rn);
integer m = UINT(Zm);
integer level = UINT(prfop<2:1>);
bool stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
```
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 0;

64-bit scaled offset

Encoding
PRFB <prfop>, <Pg>, [<Xn|SP>, <Zm>.D]

Decode for this encoding
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler symbols

<prfop> Is the prefetch operation specifier, encoded in the "prfop" field. It can have the following values:
PLDL1KEEP when prfop = 0000
PLDL1STRM when prfop = 0001
PLDL2KEEP when prfop = 0010
PLDL2STRM when prfop = 0011
PLDL3KEEP when prfop = 0100
PLDL3STRM when prfop = 0101
#uimm4 when prfop = x11x
PSTL1KEEP when prfop = 1000
PSTL1STRM when prfop = 1001
PSTL2KEEP when prfop = 1010
PSTL2STRM when prfop = 1011
PSTL3KEEP when prfop = 1100
PSTL3STRM when prfop = 1101

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

- **UXTW** when \( xs = 0 \)
- **SXTW** when \( xs = 1 \)

### Operation for all encodings

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```
C8.2.406   PRFB (vector plus immediate)

Gather prefetch of bytes from the active memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 0 0</td>
<td>imm5</td>
<td>1 1</td>
<td>Pg</td>
<td>Zn</td>
<td>0</td>
<td>prfop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Encoding

PRFB <prfop>, <Pg>, [<Zn>.S{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;
integer offset = UInt(imm5);

64-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0 0 0</td>
<td>imm5</td>
<td>1 1</td>
<td>Pg</td>
<td>Zn</td>
<td>0</td>
<td>prfop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Encoding

PRFB <prfop>, <Pg>, [<Zn>.D{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;
integer offset = UInt(imm5);
Assembler symbols

<prfop> Is the prefetch operation specifier, encoded in the "prfop" field. It can have the following values:

- PLDL1KEEP when prfop = 0000
- PLDL1STRM when prfop = 0001
- PLDL2KEEP when prfop = 0010
- PLDL2STRM when prfop = 0011
- PLDL3KEEP when prfop = 0100
- PLDL3STRM when prfop = 0101
- #uimm4 when prfop = x11x
- PSTL1KEEP when prfop = 1000
- PSTL1STRM when prfop = 1001
- PSTL2KEEP when prfop = 1010
- PSTL2STRM when prfop = 1011
- PSTL3KEEP when prfop = 1100
- PSTL3STRM when prfop = 1101

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Pg, PL];
bits(VL) base;

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
C8.2.407 PRFD (scalar plus immediate)

Contiguous prefetch of doubleword elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.

Encoding

PRFD <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;
integer offset = SInt(imm6);

Assembler symbols

<prfop> Is the prefetch operation specifier, encoded in the "prfop" field. It can have the following values:

- PLDL1KEEP when prfop = 0000
- PLDL1STRM when prfop = 0001
- PLDL2KEEP when prfop = 0010
- PLDL2STRM when prfop = 0011
- PLDL3KEEP when prfop = 0100
- PLDL3STRM when prfop = 0101
- #uimm4 when prfop = x1lx
- PSTL1KEEP when prfop = 1000
- PSTL1STRM when prfop = 1001
- PSTL2KEEP when prfop = 1010
- PSTL2STRM when prfop = 1011
- PSTL3KEEP when prfop = 1100
- PSTL3STRM when prfop = 1101

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
C8.2.408   PRFD (scalar plus scalar)

Contiguous prefetch of doubleword elements from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.

Encoding

PRFD <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;

Assembler symbols

<prfop>    Is the prefetch operation specifier, encoded in the "prfop" field. It can have the following values:
                PLDL1KEEP when prfop = 0000
                PLDL1STRM when prfop = 0001
                PLDL2KEEP when prfop = 0010
                PLDL2STRM when prfop = 0011
                PLDL3KEEP when prfop = 0100
                PLDL3STRM when prfop = 0101
                #uimm4 when prfop = x11x
                PSTL1KEEP when prfop = 1000
                PSTL1STRM when prfop = 1001
                PSTL2KEEP when prfop = 1010
                PSTL2STRM when prfop = 1011
                PSTL3KEEP when prfop = 1100
                PSTL3STRM when prfop = 1101

<Pg>    Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>    Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm>    Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(64) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
C8.2.409  PRFD (scalar plus vector)

Gather prefetch of doublewords from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 8. Inactive addresses are not prefetched from memory.

The `<prfop>` symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

### 32-bit scaled offset

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 0</td>
<td>xs 1</td>
<td>Zm 0 1 1</td>
<td>Pg</td>
<td>Rn 0</td>
<td>prfop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

#### Encoding

PRFD <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #3]

#### Decode for this encoding

- if !HaveSVE() then UNDEFINED;
- constant integer esize = 32;
- integer g = UInt(Pg);
- integer n = UInt(Rn);
- integer m = UInt(Zm);
- integer level = UInt(prfop<2:1>);
- boolean stream = (prfop<0> == '1');
- pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
- constant integer offs_size = 32;
- boolean offs_unsigned = (xs == '0');
- integer scale = 3;

### 32-bit unpacked scaled offset

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0 0</td>
<td>xs 1</td>
<td>Zm 0 1 1</td>
<td>Pg</td>
<td>Rn 0</td>
<td>prfop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

#### Encoding

PRFD <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #3]

#### Decode for this encoding

- if !HaveSVE() then UNDEFINED;
- constant integer esize = 64;
- integer g = UInt(Pg);
- integer n = UInt(Rn);
- integer m = UInt(Zm);
- integer level = UInt(prfop<2:1>);
- boolean stream = (prfop<0> == '1');
- pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 3;

64-bit scaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0 0 1 1</td>
<td>Zm</td>
<td>1 1</td>
<td>Pg</td>
<td>Rn</td>
<td>0</td>
<td>prfop</td>
</tr>
</tbody>
</table>

Encoding

PRFD <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, LSL #3]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 3;

Assembler symbols

<prfop>  Is the prefetch operation specifier, encoded in the "prfop" field. It can have the following values:
  PLDL1KEEP when prfop = 0000
  PLDL1STRM when prfop = 0001
  PLDL2KEEP when prfop = 0010
  PLDL2STRM when prfop = 0011
  PLDL3KEEP when prfop = 0100
  PLDL3STRM when prfop = 0101
  #uimm4 when prfop = x1lx
  PSTL1KEEP when prfop = 1000
  PSTL1STRM when prfop = 1001
  PSTL2KEEP when prfop = 1010
  PSTL2STRM when prfop = 1011
  PSTL3KEEP when prfop = 1100
  PSTL3STRM when prfop = 1101

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm>  Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

<table>
<thead>
<tr>
<th>UXTW</th>
<th>when xs = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SXTW</td>
<td>when xs = 1</td>
</tr>
</tbody>
</table>

**Operation for all encodings**

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```
C8.2.410  PRFD (vector plus immediate)

Gather prefetch of doublewords from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 1 1 0 0</td>
<td>imm5</td>
<td>1 1 1</td>
<td>Pg</td>
<td>Zn</td>
<td>0</td>
<td>prfop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Encoding

PRFD <prfop>, <Pg>, [<Zn>.S{, #<imm>}]

Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;
integer offset = UInt(imm5);
```

64-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 1 1 0 0</td>
<td>imm5</td>
<td>1 1 1</td>
<td>Pg</td>
<td>Zn</td>
<td>0</td>
<td>prfop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Encoding

PRFD <prfop>, <Pg>, [<Zn>.D{, #<imm>}]

Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;
integer offset = UInt(imm5);
```
Assembler symbols

<prfop>  Is the prefetch operation specifier, encoded in the "prfop" field. It can have the following values:

- PLDL1KEEP when prfop = 0000
- PLDL1STRM when prfop = 0001
- PLDL2KEEP when prfop = 0010
- PLDL2STRM when prfop = 0011
- PLDL3KEEP when prfop = 0100
- PLDL3STRM when prfop = 0101
- #uimm4 when prfop = x11x
- PSTL1KEEP when prfop = 1000
- PSTL1STRM when prfop = 1001
- PSTL2KEEP when prfop = 1010
- PSTL2STRM when prfop = 1011
- PSTL3KEEP when prfop = 1100
- PSTL3STRM when prfop = 1101

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn>  Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm>  Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
C8.2.411 PRFH (scalar plus immediate)

Contiguous prefetch of halfword elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.

Encoding

PRFH <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
integer offset = SInt(imm6);

Assembler symbols

<prfop> Is the prefetch operation specifier, encoded in the "prfop" field. It can have the following values:
PLDL1KEEP when prfop = 0000
PLDL1STRM when prfop = 0001
PLDL2KEEP when prfop = 0010
PLDL2STRM when prfop = 0011
PLDL3KEEP when prfop = 0100
PLDL3STRM when prfop = 0101
#uimm4 when prfop = x11x
PSTL1KEEP when prfop = 1000
PSTL1STRM when prfop = 1001
PSTL2KEEP when prfop = 1010
PSTL2STRM when prfop = 1011
PSTL3KEEP when prfop = 1100
PSTL3STRM when prfop = 1101

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.
**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

```c```
C8.2.412  PRFH (scalar plus scalar)

Contiguous prefetch of halfword elements from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.

Encoding

PRFH <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;

Assembler symbols

<prfop>  Is the prefetch operation specifier, encoded in the "prfop" field. It can have the following values:
    PLDL1KEEP when prfop = 0000
    PLDL1STRM when prfop = 0001
    PLDL2KEEP when prfop = 0010
    PLDL2STRM when prfop = 0011
    PLDL3KEEP when prfop = 0100
    PLDL3STRM when prfop = 0101
    #uimm4 when prfop = x1lx
    PSTL1KEEP when prfop = 1000
    PSTL1STRM when prfop = 1001
    PSTL2KEEP when prfop = 1010
    PSTL2STRM when prfop = 1011
    PSTL3KEEP when prfop = 1100
    PSTL3STRM when prfop = 1101

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(64) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements - 1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
C8.2.413   PRFH (scalar plus vector)

Gather prefetch of halfwords from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 2. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit scaled offset

```
[31 30 29 28][27 26 25 24][23 22 21 20] 16[15 14 13 12] 10 9 | 5 4 | 3 0 |
1 0 0 0 0 1 0 0 0 xs 1 Zm 0 0 1 Pg Rn 0 prfop
```

Encoding

PRFH <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #1]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 1;

32-bit unpacked scaled offset

```
[31 30 29 28][27 26 25 24][23 22 21 20] 16[15 14 13 12] 10 9 | 5 4 | 3 0 |
1 1 0 0 0 1 0 0 0 xs 1 Zm 0 0 1 Pg Rn 0 prfop
```

Encoding

PRFH <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #1]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 1;

64-bit scaled offset

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20]</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 1 0 0 0 1 0 0 0 1 1</td>
<td>Zm</td>
<td>1 0 1</td>
<td>Pg</td>
</tr>
</tbody>
</table>

Encoding

PRFH <prfop>, <Pg>, [<Xn>|SP>, <Zm>.D, LSL #1]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 1;

Assembler symbols

<prfop> Is the prefetch operation specifier, encoded in the "prfop" field. It can have the following values:
PLDL1KEEP when prfop = 0000
PLDL1STRM when prfop = 0001
PLDL2KEEP when prfop = 0010
PLDL2STRM when prfop = 0011
PLDL3KEEP when prfop = 0100
PLDL3STRM when prfop = 0101
#uimm4 when prfop = x11x
PSTL1KEEP when prfop = 1000
PSTL1STRM when prfop = 1001
PSTL2KEEP when prfop = 1010
PSTL2STRM when prfop = 1011
PSTL3KEEP when prfop = 1100
PSTL3STRM when prfop = 1101

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn>|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

- **UXTW** when \( xs = 0 \)
- **SXTW** when \( xs = 1 \)

### Operation for all encodings

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;

if AnyActiveElement(mask, esize) then
  base = if n == 31 then SP[] else X[n, 64];
  offset = Z[m, VL];

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    bits(64) addr = base + (off << scale);
    Hint_Prefetch(addr, pref_hint, level, stream);
```
C8.2.414  PRFH (vector plus immediate)

Gather prefetch of halfwords from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive addresses are not prefetched from memory.

The `<prfop>` symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0</td>
<td>0 1 0 0</td>
<td>imm5</td>
<td>Pg</td>
<td>Zn</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

PRFH `<prfop>`, `<Pg>`, `<Zn>.S{, #<imm>}`

Decode for this encoding

if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
integer offset = UInt(imm5);

64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0</td>
<td>0 1 0 0</td>
<td>imm5</td>
<td>Pg</td>
<td>Zn</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

PRFH `<prfop>`, `<Pg>`, `<Zn>.D{, #<imm>}`

Decode for this encoding

if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
integer offset = UInt(imm5);
Assembler symbols

<prfop> Is the prefetch operation specifier, encoded in the "prfop" field. It can have the following values:

- PLDL1KEEP when prfop = 0000
- PLDL1STRM when prfop = 0001
- PLDL2KEEP when prfop = 0010
- PLDL2STRM when prfop = 0011
- PLDL3KEEP when prfop = 0100
- PLDL3STRM when prfop = 0101
- #uimm4 when prfop = x11x
- PSTL1KEEP when prfop = 1000
- PSTL1STRM when prfop = 1001
- PSTL2KEEP when prfop = 1010
- PSTL2STRM when prfop = 1011
- PSTL3KEEP when prfop = 1100
- PSTL3STRM when prfop = 1101

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;

if AnyActiveElement(mask, esize) then
  base = Z[n, VL];

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
    Hint_Prefetch(addr, pref_hint, level, stream);
C8.2.415  PRFW (scalar plus immediate)

Contiguous prefetch of word elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.

Encoding

PRFW <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED; constant integer esize = 32; integer g = UInt(Pg); integer n = UInt(Rn); integer level = UInt(prfop<2:1>); boolean stream = (prfop<0> == '1'); pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE; integer scale = 2; integer offset = SInt(imm6);

Assembler symbols

<prfop> Is the prefetch operation specifier, encoded in the "prfop" field. It can have the following values:
   PLDL1KEEP when prfop = 0000
   PLDL1STRM when prfop = 0001
   PLDL2KEEP when prfop = 0010
   PLDL2STRM when prfop = 0011
   PLDL3KEEP when prfop = 0100
   PLDL3STRM when prfop = 0101
   #uimm4 when prfop = x1lx
   PSTL1KEEP when prfop = 1000
   PSTL1STRM when prfop = 1001
   PSTL2KEEP when prfop = 1010
   PSTL2STRM when prfop = 1011
   PSTL3KEEP when prfop = 1100
   PSTL3STRM when prfop = 1101

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP else X[n, 64];

for e = 0 to elements-1
    if Elemp[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```
C8.2.416    PRFW (scalar plus scalar)

Contiguous prefetch of word elements from the memory address generated by a 64-bit scalar base and scalar index
which is multiplied by 4 and added to the base address. After each element prefetch the index value is incremented,
but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.

Encoding

PRFW <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;

Assembler symbols

<prfop>      Is the prefetch operation specifier, encoded in the "prfop" field. It can have the following values:
              PLDL1KEEP when prfop = 0000
              PLDL1STRM when prfop = 0001
              PLDL2KEEP when prfop = 0010
              PLDL2STRM when prfop = 0011
              PLDL3KEEP when prfop = 0100
              PLDL3STRM when prfop = 0101
              #uimm4 when prfop = x11x
              PSTL1KEEP when prfop = 1000
              PSTL1STRM when prfop = 1001
              PSTL2KEEP when prfop = 1010
              PSTL2STRM when prfop = 1011
              PSTL3KEEP when prfop = 1100
              PSTL3STRM when prfop = 1101

<Pg>         Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>      Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm>         Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

checkSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(64) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if Elemp[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
C8.2.417 PRFW (scalar plus vector)

Gather prefetch of words from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 4. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit scaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 0 xs 1 Zm 0 1 0 Pg Rn 0 prfop</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

PRFW <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #2]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 2;

32-bit unpacked scaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0 0 xs 1 Zm 0 1 0 Pg Rn 0 prfop</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

PRFW <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #2]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 2;

64-bit scaled offset

Encoding

PRFW <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, LSL #2]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 2;

Assembler symbols

<prfop> Is the prefetch operation specifier, encoded in the "prfop" field. It can have the following values:
PLDL1KEEP when prfop = 0000
PLDL1STRM when prfop = 0001
PLDL2KEEP when prfop = 0010
PLDL2STRM when prfop = 0011
PLDL3KEEP when prfop = 0100
PLDL3STRM when prfop = 0101
#uimm4 when prfop = x11x
PSTL1KEEP when prfop = 1000
PSTL1STRM when prfop = 1001
PSTL2KEEP when prfop = 1010
PSTL2STRM when prfop = 1011
PSTL3KEEP when prfop = 1100
PSTL3STRM when prfop = 1101

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

- **UXTW** when \( xs = 0 \)
- **SXTW** when \( xs = 1 \)

### Operation for all encodings

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```
C8.2.418   PRFW (vector plus immediate)

Gather prefetch of words from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 1 0 1 0 0 0</td>
<td>imm5 1 1 1</td>
<td>Pg</td>
<td>Zn 0</td>
<td>prfop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

Encoding

PRFW <prfop>, <Pg>, [<Zn>.S{, #<imm}>]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
integer offset = UInt(imm5);

64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 1 0 0 0</td>
<td>imm5 1 1 1</td>
<td>Pg</td>
<td>Zn 0</td>
<td>prfop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

Encoding

PRFW <prfop>, <Pg>, [<Zn>.D{, #<imm}>]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
integer offset = UInt(imm5);
Assembler symbols

<prfop> Is the prefetch operation specifier, encoded in the "prfop" field. It can have the following values:
- PLDL1KEEP when prfop = 0000
- PLDL1STRM when prfop = 0001
- PLDL2KEEP when prfop = 0010
- PLDL2STRM when prfop = 0011
- PLDL3KEEP when prfop = 0100
- PLDL3STRM when prfop = 0101
- #uimm4 when prfop = x11x
- PSTL1KEEP when prfop = 1000
- PSTL1STRM when prfop = 1001
- PSTL2KEEP when prfop = 1010
- PSTL2STRM when prfop = 1011
- PSTL3KEEP when prfop = 1100
- PSTL3STRM when prfop = 1101

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    ifElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
C8.2.419  PSEL

If the indexed element of the second source predicate is true, place the contents of the first source predicate register into the destination predicate register, otherwise set the destination predicate to all-false. The indexed element is determined by the sum of a general-purpose index register and an immediate, modulo the number of elements. Does not set the condition flags.

SVE2

(_FEAT_SME)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16 15 14 13</th>
<th>12 11 10 9 8</th>
<th>7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 1 0</td>
<td>i1</td>
<td>tszl</td>
<td>Rv</td>
<td>0</td>
<td>Pn</td>
</tr>
</tbody>
</table>

Encoding

PSEL <Pd>, <Pn>, <Pm>, <T><Wv>, <imm>

Decode for this encoding

if !HaveSME() then UNDEFINED;
bits(5) imm5 = i1:tszh:tszl;
integer esize;
integer imm;
case tszh:tszl of
  when '0000' UNDEFINED;
  when '1000' esize = 64;  imm = UInt(imm5<4>);
  when 'x100' esize = 32;  imm = UInt(imm5<4:3>);
  when 'xx10' esize = 16;  imm = UInt(imm5<4:2>);
  when 'xxx1' esize = 8;   imm = UInt(imm5<4:1>);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer v = UInt('011':Rv);

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.
<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  B when tszh = x, tszl = xx1
  H when tszh = x, tszl = x10
  S when tszh = x, tszl = 100
  D when tszh = 1, tszl = 000
The encoding tszh = 0, tszl = 000 is reserved.
<Wv> Is the 32-bit name of the vector select register W12-W15, encoded in the "Rv" field.
<imm> Is the element index, in the range 0 to one less than the number of vector elements in a 128-bit vector register, encoded in "i1:tszh:tszl".
Operation

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(32) idx = X[v, 32];
integer element = (UInt(idx) + imm) MOD elements;
bits(PL) result;
if ElemP[operand2, element, esize] == '1' then
   result = operand1;
ext
   result = Zeros(PL);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.420  PTEST

Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate source register, and the V flag to zero.

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & \text{Pg} & 0 & \text{Pn} & 0 & 0 & 0 & 0 & 0 & 0 & S
\end{array}
\]

**Encoding**

PTEST <Pg>, <Pn>.B

**Decode for this encoding**

```plaintext
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
```

**Assembler symbols**

- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the source scalable predicate register, encoded in the "Pn" field.

**Operation**

```plaintext
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(PL) result = P[n, PL];
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.
C8.2.421  PTRUE

Set elements of the destination predicate to true if the element number satisfies the named predicate constraint, or
to false otherwise. If the constraint specifies more elements than are available at the current vector length then all
elements of the destination predicate are set to false.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than
Undefined Instruction exception. Does not set the condition flags.

Encoding

PTRUE <Pd>.<T>{, <pattern>}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer d = UInt(Pd);
boolean setflags = FALSE;
bits(5) pat = pattern;

Assembler symbols

<Pd>  Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

<table>
<thead>
<tr>
<th>B</th>
<th>H</th>
<th>S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>when size = 00</td>
<td>when size = 01</td>
<td>when size = 10</td>
<td>when size = 11</td>
</tr>
</tbody>
</table>

<pattern>  Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

<table>
<thead>
<tr>
<th>POW2</th>
<th>VL1</th>
<th>VL2</th>
<th>VL3</th>
<th>VL4</th>
<th>VL5</th>
<th>VL6</th>
</tr>
</thead>
<tbody>
<tr>
<td>when pattern = 00000</td>
<td>when pattern = 00001</td>
<td>when pattern = 00010</td>
<td>when pattern = 00011</td>
<td>when pattern = 00100</td>
<td>when pattern = 00101</td>
<td>when pattern = 00110</td>
</tr>
</tbody>
</table>
VL7  when pattern = 00111
VL8  when pattern = 01000
VL16 when pattern = 01001
VL32 when pattern = 01010
VL64 when pattern = 01011
VL128 when pattern = 01100
VL256 when pattern = 01101
#uimm5 when pattern = 0111x
#uimm5 when pattern = 101x1
#uimm5 when pattern = 10110
#uimm5 when pattern = 1x0x1
#uimm5 when pattern = 1x010
#uimm5 when pattern = 1xx00
MUL4 when pattern = 11101
MUL3 when pattern = 11110
ALL  when pattern = 11111

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(PL) result;

for e = 0 to elements-1
    ElemP[result, e, esize] = if e < count then '1' else '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(result, result, esize);
P[d, PL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.422   PTRUES

Set elements of the destination predicate to true if the element number satisfies the named predicate constraint, or to false otherwise. If the constraint specifies more elements than are available at the current vector length then all elements of the destination predicate are set to false.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

Encoding

PTRUES <Pd>.<T>{, <pattern>}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << Uint(size);
integer d = Uint(Pd);
boolean setflags = TRUE;
browse(5) pat = pattern;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:
POW2 when pattern = 00000
VL1 when pattern = 00001
VL2 when pattern = 00010
VL3 when pattern = 00011
VL4 when pattern = 00100
VL5 when pattern = 00101
VL6 when pattern = 00110
VL7 when pattern = 00111
VL8 when pattern = 01000
VL16 when pattern = 01001
VL32 when pattern = 01010
VL64 when pattern = 01011
VL128 when pattern = 01100
VL256 when pattern = 01101
#uimm5 when pattern = 0111x
#uimm5 when pattern = 101x1
#uimm5 when pattern = 10110
#uimm5 when pattern = 1x0x1
#uimm5 when pattern = 1x010
#uimm5 when pattern = 1xx00
MUL4 when pattern = 11101
MUL3 when pattern = 11110
ALL when pattern = 11111

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(PL) result;
for e = 0 to elements-1
    ElemP[result, e, esize] = if e < count then '1' else '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(result, result, esize);
P[d, PL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.
C8.2.423  PUNPKHI, PUNPKLO

Unpack elements from the lowest or highest half of the source predicate and place in elements of twice their size within the destination predicate. This instruction is unpredicated.

**High half**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8 | 5 4 3 0 |
0 0 0 0 0 1 0 1 0 0 1 1 0 0 0 | 1 0 1 0 0 0 0 0 | Pn 0 | Pd 0]
```

**Encoding**

PUNPKHI <Pd>.H, <Pn>.B

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean hi = TRUE;
```

**Low half**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8 | 5 4 3 0 |
0 0 0 0 0 1 0 1 0 0 1 1 0 0 0 | 0 1 0 0 0 0 0 | Pn 0 | Pd 0]
```

**Encoding**

PUNPKLO <Pd>.H, <Pn>.B

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean hi = FALSE;
```

**Assembler symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pn>` Is the name of the source scalable predicate register, encoded in the "Pn" field.

**Operation for all encodings**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) operand = P[n, PL];
bits(PL) result;
for e = 0 to elements-1
  ElemP[result, e, esize] = ElemP[operand, if hi then e + elements else e, esize DIV 2];
```
\[ P[d, PL] = \text{result}; \]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.424   RADDHNB

Add each vector element of the first source vector to the corresponding vector element of the second source vector, and place the most significant rounded half of the result in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. This instruction is unpredicated.

Encoding

RADDHNB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>   Is the size specifier, encoded in the "size" field. It can have the following values:
      B when size = 01
      H when size = 10
      S when size = 11
      The encoding size = 00 is reserved.
<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb>  Is the size specifier, encoded in the "size" field. It can have the following values:
      H when size = 01
      S when size = 10
      D when size = 11
      The encoding size = 00 is reserved.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
constant integer halfsize = esize DIV 2;
constant integer round_const = 1 << (halfsize - 1);
for e = 0 to elements-1
integer element1 = UInt(Elem[operand1, e, esize]);
integer element2 = UInt(Elem[operand2, e, esize]);
integer res = ((element1 + element2) + round_const) >> halfesize;
Elem[result, 2*e + 0, halfesize] = res<halfesize-1:0>;
Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.425 RADDHNT

Add each vector element of the first source vector to the corresponding vector element of the second source vector, and place the most significant rounded half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.

```
   31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13 12|11 10  9 |  5  4 |  0 |
  0 1 0 0 0 1 0 1 | size | 1 |     Zm 0 1 1 0 1 1 | Zn | Zd |
          S    R    T
```

**Encoding**

RADDHNT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 01
H when size = 10
S when size = 11

The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11

The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;
constant integer round_const = 1 << (halfesize - 1);
for e = 0 to elements-1
```
integer element1 = UInt(Elem[operand1, e, esize]);
integer element2 = UInt(Elem[operand2, e, esize]);
integer res = ((element1 + element2) + round_const) >> halfesize;
Elem[result, 2*e + 1, halfesize] = res<halfesize-1:0>;

Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.426  RAX1

Rotate each 64-bit element of the second source vector left by 1 and exclusive OR with the corresponding elements of the first source vector. The results are placed in the corresponding elements of the destination vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.SHA3 indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2

(FEAT_SVE_SHA3)

Encoding

RAX1 <Zd>.D, <Zn>.D, <Zm>.D

Decode for this encoding

if !HaveSVE2SHA3() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 64;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(64) element1 = Elem[operand1, e, 64];
  bits(64) element2 = Elem[operand2, e, 64];
  Elem[result, e, 64] = element1 EOR ROL(element2, 1);
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.427   RBIT

Reverse bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 1 1 1 0 0</td>
<td>Pg</td>
<td>Zn</td>
</tr>
</tbody>
</table>

**Encoding**

RBIT <Zd>.<T>, <Pg>/M, <Zn>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
if Elem[mask, e, esize] == '1' then
bits(esize) element = Elem[operand, e, esize];
Elem[result, e, esize] = BitReverse(element);
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.428   RDFFR (predicated)

Read the first-fault register (FFR) and place active elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

```
0 0 1 0 0 1 0 1|0 0 0 1 1 0 0 0 1 1 1 1 0 0 | Pg 0 Pd
```

**Encoding**

RDFFR <Pd>.B, <Pg>/Z

**Decode for this encoding**

if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer d = UInt(Pd);
boolean setflags = FALSE;

**Assembler symbols**

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

**Operation**

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(PL) ffr = FFR[PL];
bits(PL) result = ffr AND mask;
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, 8);
P[d, PL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.429   RDFFR (unpredicated)

Read the first-fault register (FFR) and place in the destination predicate without predication.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

Encoding
RDFFR <Pd>.B

Decode for this encoding
if !HaveSVE() then UNDEFINED;
integer d = UInt(Pd);

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

Operation
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) ffr = FFR[PL];
P[d, PL] = ffr;

Operational information
If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
## C8.2.430 RDFFRS

Read the first-fault register (FFR) and place active elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

### Encoding

RDFFRS <Pd>.B, <Pg>/Z

### Decode for this encoding

```plaintext
if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

### Assembler symbols

- `<Pd>` is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` is the name of the governing scalable predicate register, encoded in the "Pg" field.

### Operation

```plaintext
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(PL) ffr = FFR[PL];
bits(PL) result = ffr AND mask;
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, 8);
P[d, PL] = result;
```

### Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.431   RDVL

Multiply the current vector register size in bytes by an immediate in the range -32 to 31 and place the result in the 64-bit destination general-purpose register.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0</td>
<td>1 0 0 1</td>
<td>0 1 1 1 1</td>
<td>1 0 1 0 1 0</td>
<td>imm6</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**
RDVL <Xd>, #<imm>

**Decode for this encoding**
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer d = UInt(Rd);
integer imm = SInt(imm6);

**Assembler symbols**

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

**Operation**
CheckSVEEnabled();
constant integer VL = CurrentVL;
integer len = imm * (VL DIV 8);
X[d, 64] = len<63:0>;

**Operational information**
If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.432   REV (predicate)

Reverse the order of all elements in the source predicate and place in the destination predicate. This instruction is unpredicated.

Encoding

REV <Pd>.<T>, <Pn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer d = UInt(Pd);

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) operand = P[n, PL];
bits(PL) result = Reverse(operand, esize DIV 8);
P[d, PL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.433  REV (vector)

Reverse the order of all elements in the source vector and place in the destination vector. This instruction is unpredicated.

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9  | 5 4  | 0 |
| 0 0 0 0 0 0 1 0 1 | 1 1 1 0 0 0 0 0 1 1 1 0 | Zn | Zd |
```

**Encoding**

REV `<Zd>..<T>`, `<Zn>..<T>`

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

**Assembler symbols**

`<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.

`<T>` Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

`<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(VL) operand = Z[n, VL];
bits(VL) result = Reverse(operand, esize);
Z[d, VL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.434  REVB, REVH, REVW

Reverse the order of 8-bit bytes, 16-bit halfwords or 32-bit words within each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

**Byte**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12] | 10 9  | 5 4 | 0 |
0 0 0 0 0 1 0 1 | size | 1 0 0 1 0 0 1 0 0 | Pg | Zn | Zd
```

**Encoding**

REVB <Zd>.<T>, <Pg>/M, <Zn>.<T>

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer swsize = 8;
```

**Halfword**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12] | 10 9  | 5 4 | 0 |
0 0 0 0 0 1 0 1 | size | 1 0 0 1 0 1 1 0 0 | Pg | Zn | Zd
```

**Encoding**

REVH <Zd>.<T>, <Pg>/M, <Zn>.<T>

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer swsize = 16;
```

**Word**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12] | 10 9  | 5 4 | 0 |
0 0 0 0 0 1 0 1 | size | 1 0 0 1 1 0 1 0 0 | Pg | Zn | Zd
```

**Encoding**

REVW <Zd>.D, <Pg>/M, <Zn>.D
Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size != '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer swsize = 32;

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> For the byte variant: is the size specifier, encoded in the "size" field. It can have the following values:
   H when size = 01
   S when size = 10
   D when size = 11
   The encoding size = 00 is reserved.
   For the halfword variant: is the size specifier, encoded in the "size<0>" field. It can have the following values:
   S when size<0> = 0
   D when size<0> = 1

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
   if Elem[mask, e, esize] == '1' then
       bits(esize) element = Elem[operand, e, esize];
       Elem[result, e, esize] = Reverse(element, swsize);

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
— The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.435 REVD

Reverse the order of 64-bit doublewords within each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

SVE2

(FEAT_SME)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
<td>0 0 1 0 1 1 0 1 0 0</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

REVD <Zd>.Q, <Pg>/M, <Zn>.Q

Decode for this encoding

if !HaveSME() then UNDEFINED;
constant integer esize = 128;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer swsize = 64;

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = Reverse(element, swsize);

Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.436 RSHRNB

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Encoding

RSHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
   B when tszh = 0, tszl = 01
   H when tszh = 0, tszl = 1x
   S when tszh = 1, tszl = xx
The encoding tszh = 0, tszl = 00 is reserved.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
   H when tszh = 0, tszl = 01
   S when tszh = 0, tszl = 1x
   D when tszh = 1, tszl = xx
The encoding tszh = 0, tszl = 00 is reserved.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (UInt(element) + round_const) >> shift;
    Elem[result, 2*e + 0, esize] = res<esize-1:0>;
    Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.437   RSHRNT

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the rounded
results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. The
immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is
unpredicated.

Encoding

RSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bz(t3) tsize = tszh:tszl;
integer esize;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler symbols

<Zd>   Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>   Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  B   when tzh = 0, tszl = 01
  H   when tzh = 0, tszl = 1x
  S   when tzh = 1, tszl = xx
The encoding tzh = 0, tszl = 00 is reserved.

<Zn>   Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb>   Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  H   when tzh = 0, tszl = 01
  S   when tzh = 0, tszl = 1x
  D   when tzh = 1, tszl = xx
The encoding tzh = 0, tszl = 00 is reserved.

<const>   Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = (UInt(element) + round_const) >> shift;
  Elem[result, 2*e + 1, esize] = res<esize-1:0>;

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.438  RSUBHNB

Subtract each vector element of the second source vector from the corresponding vector element in the first source vector, and place the most significant rounded half of the result in the even-numbered half-width destination elements, while setting the odd-numbered half-width destination elements to zero. This instruction is unpredicated.

Encoding

RSUBHNB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
   B when size = 01
   H when size = 10
   S when size = 11
   The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
   H when size = 01
   S when size = 10
   D when size = 11
   The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;
constant integer round_const = 1 << (halfesize - 1);
for e = 0 to elements-1
integer element1 = UInt(Elem[operand1, e, esize]);
integer element2 = UInt(Elem[operand2, e, esize]);
integer res = ((element1 - element2) + round_const) >> halfesize;
Elem[result, 2*e + 0, halfesize] = res<halfesize-1:0>;
Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);

Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.439  RSUBHNT

Subtract each vector element of the second source vector from the corresponding vector element in the first source vector, and place the most significant rounded half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.

Encoding

RSUBHNT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << Uint(size);
integer n = Uint(Zn);
integer m = Uint(Zm);
integer d = Uint(Zd);

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>   Is the size specifier, encoded in the "size" field. It can have the following values:
   B    when size = 01
   H    when size = 10
   S    when size = 11
The encoding size = 00 is reserved.
<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb>  Is the size specifier, encoded in the "size" field. It can have the following values:
   H    when size = 01
   S    when size = 10
   D    when size = 11
The encoding size = 00 is reserved.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;
constant integer round_const = 1 << (halfesize - 1);
for e = 0 to elements-1
integer element1 = UInt(Elem[operand1, e, esize]);
integer element2 = UInt(Elem[operand2, e, esize]);
integer res = ((element1 - element2) + round_const) >> halfesize;
Elem[result, 2*e + 1, halfesize] = res<halfesize-1:0>;

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
**C8.2.440 SABA**

Compute the absolute difference between signed integer values in elements of the second source vector and corresponding elements of the first source vector, and add the difference to the corresponding elements of the destination vector. This instruction is unpredicated.

![Instruction Encoding](image)

**Encoding**

SABA $<Zda>.<T>$, $<Zn>.<T>$, $<Zm>.<T>$

**Decode for this encoding**

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean unsigned = FALSE;
```

**Assembler symbols**

- $<Zda>$: Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- $<T>$: Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when $size = 00$
  - H when $size = 01$
  - S when $size = 10$
  - D when $size = 11$
- $<Zn>$: Is the name of the first source scalable vector register, encoded in the "Zn" field.
- $<Zm>$: Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  integer element2 = Int(Elem[operand2, e, esize], unsigned);
  bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
  Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
Z[da, VL] = result;
```
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.441 SABALB

Compute the absolute difference between even-numbered signed integer values in elements of the second source vector and corresponding elements of the first source vector, and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

Encoding

SABALB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() & !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
n = UInt(Zn);
m = UInt(Zm);
da = UInt(Zda);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);

bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
Z[da, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.442  SABALT

Compute the absolute difference between odd-numbered signed elements of the second source vector and corresponding elements of the first source vector, and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

```
01000101
0 1 0 0 0 1 0 1 size 0 Zm 1 1 0 0 0 1 Zn Zda
```

**Encoding**

SABALT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

**Assembler symbols**

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 01
- H when size = 10
- S when size = 11

The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
  integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
```
bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
Z[da, VL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.443 SABD

Compute the absolute difference between signed integer values in active elements of the second source vector and corresponding elements of the first source vector and destructively place the difference in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

SABD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if Elem[mask, e, esize] == '1' then
        integer absdiff = Abs(element1 - element2);
        Elem[result, e, esize] = absdiff<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.444 SABDLB

Compute the absolute difference between even-numbered signed integer values in elements of the second source vector and corresponding elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

Encoding

SABDLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

\[
\text{if } \text{ !HaveSVE2() && !HaveSME() then UNDEFINED;}
\]
\[
\text{if size == '00' then UNDEFINED;}
\]
\[
\text{constant integer esize} = 8 << \text{UInt(size)};
\]
\[
\text{integer } n = \text{ UInt(Zn)};
\]
\[
\text{integer } m = \text{ UInt(Zm)};
\]
\[
\text{integer } d = \text{ UInt(Zd)};
\]

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11

The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 01
H when size = 10
S when size = 11

The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
     integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
     integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
integer res = Abs(element1 - element2);
Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.445   SABDLT

Compute the absolute difference between odd-numbered signed integer values in elements of the second source vector and corresponding elements of the first source vector, and place the results in overlapping double-width elements of the destination vector. This instruction is unpredicated.

\begin{verbatim}
[31 30 29 28|27 26 25 24|23 22 21 20]  16|15 14 13 12|11 10 9  |  5 4 |  0 |
0 1 0 0 0 1 0 1 | Zm 0 0 1 1 0 1 Zn | Zd
\end{verbatim}

\textbf{Encoding}

SABDLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

\textbf{Decide for this encoding}

\begin{verbatim}
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
\end{verbatim}

\textbf{Assembler symbols}

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

H  when size = 01
S  when size = 10
D  when size = 11

The encoding size = '00' is reserved.

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb>  Is the size specifier, encoded in the "size" field. It can have the following values:

B  when size = 01
H  when size = 10
S  when size = 11

The encoding size = '00' is reserved.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

\textbf{Operation}

\begin{verbatim}
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
  integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
\end{verbatim}
integer res = Abs(element1 - element2);
Elem[result, e, esize] = res<esize-1:0>;
Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.446   SADALP

Add pairs of adjacent signed integer values and accumulate the results into the overlapping double-width elements of the destination vector.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9  | 5 4  | 0 | 0 ]

0 1 0 0 0 1 0 0 | 0 0 0 1 0 0 1 0 | Pg | Zn | Zda
```

**Encoding**

SADALP <Zda>,<T>, <Pg>/M, <Zn>,<Tb>

**Decode for this encoding**

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer da = UInt(Zda);
```

**Assembler symbols**

- `<Zda>` Is the name of the second source and destination scalable vector register, encoded in the "Zda" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - `H` when size = 01
  - `S` when size = 10
  - `D` when size = 11
  The encoding size = 00 is reserved.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - `B` when size = 01
  - `H` when size = 10
  - `S` when size = 11
  The encoding size = 00 is reserved.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand_acc = Z[da, VL];
bits(VL) operand_src = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '0' then
    Elem[result, e, esize] = Elem[operand_acc, e, esize];
```

else
    integer element1 = SInt(Elem[operand_src, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand_src, 2*e + 1, esize DIV 2]);
    bits(esize) sum = (element1 + element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[operand_acc, e, esize] + sum;
Z[da, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.447 SADDLB

Add the corresponding even-numbered signed elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

```
[31 30 29 28|27 26 25 24|23 22 21 20]  16|15 14 13 12|11 10 9 | 5 4 | 0 |
  0 1 0 0 0 1 0 1 | size 0 | Zm 0 0 0 0 | 0 0 | Zn | Zd |
      S    U    T
```

**Encoding**

SADDLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

**Decode for this encoding**

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 0;
boolean unsigned = FALSE;
```

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 01
- H when size = 10
- S when size = 11

The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
```
integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
integer res = element1 + element2;
Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.448  SADDLBT

Add the even-numbered signed elements of the first source vector to the odd-numbered signed elements of the second source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

```
| 31 30 29 28|27 26 25 24|23 22 21 20 | 16|15 14 13|12|11 10 9 | 5 4 | 0 |
0 1 0 0 0 1 0 1| size| 0 | Zm | 1 0 0 0 | 0 0 | Zn | Zd |
S tb
```

**Encoding**

```
SADDLBT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

**Decode for this encoding**

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 1;
boolean unsigned = FALSE;
```

**Assembler symbols**

- `<Zd>` is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` is the size specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11
  - The encoding size = 00 is reserved.
- `<Zn>` is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>` is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 01
  - H when size = 10
  - S when size = 11
  - The encoding size = 00 is reserved.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
```
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 + element2;
    Elem[result, e, esize] = res<esize-1:0>;
Z[d, VL] = result;

Operational information
If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.449 SADDLT

Add the corresponding odd-numbered signed elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

Encoding

SADDLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 1;
boolean unsigned = FALSE;

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
    H when size = 01
    S when size = 10
    D when size = 11
The encoding size = 00 is reserved.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
    B when size = 01
    H when size = 10
    S when size = 11
The encoding size = 00 is reserved.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
integer res = element1 + element2;
Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.450 SADDV

Signed add horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Narrow elements are first sign-extended to 64 bits. Inactive elements in the source vector are treated as zero.

Encoding

SADDV <Dd>, <Pg>, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler symbols

<Dd> Is the 64-bit name of the destination SIMD&FP register, encoded in the "Vd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10

The encoding size = 11 is reserved.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = Z[n, VL];
integer sum = 0;

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer element = SInt(Elem[operand, e, esize]);
    sum = sum + element;

V[d, 64] = sum<63:0>;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.
C8.2.451 SADDWB

Add the even-numbered signed elements of the second source vector to the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

Encoding
SADDWB <Zd>,<T>, <Zn>,<T>, <Zm>,<Tb>

Decode for this encoding
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols
<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
   H when size = 01
   S when size = 10
   D when size = 11
   The encoding size = 00 is reserved.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
   B when size = 01
   H when size = 10
   S when size = 11
   The encoding size = 00 is reserved.

Operation
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
   integer element1 = SInt(Elem[operand1, e, esize]);
   integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
Elem[result, e, esize] = (element1 + element2)<esize-1:0>;
Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.452 SADDWT

Add the odd-numbered signed elements of the second source vector to the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

| 31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13 12|11 10 9 | 5 | 4 | 0 |
| 0 1 0 0 0|1 1 0 0 |0 1 0 0 |0 |0 1 |Zm |Zn |Zd |

Encoding

SADDWT <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
n = UInt(Zn);
m = UInt(Zm);
d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
\[ \text{Elem[result, e, esize]} = (\text{element1 + element2})<\text{esize}-1:0>; \]
\[ Z[d, VL] = \text{result}; \]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.453  SBCLB

Subtract the even-numbered elements of the first source vector and the inverted 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector from the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.

Encoding

SBCLB <Zda>,<T>, <Zn>,<T>, <Zm>,<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler symbols

<Zda>  Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in the "sz" field. It can have the following values:
S    when sz = 0
D    when sz = 1

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand = Z[n, VL];
bits(VL) carries = Z[m, VL];
bits(VL) result = Z[da, VL];

for p = 0 to pairs-1
bits(esize) element1 = Elem[result, 2*p + 0, esize];
bits(esize) element2 = Elem[operand, 2*p + 0, esize];
bit carry_in = Elem[carries, 2*p + 1, esize]<0>;

(res, nzcv) = AddWithCarry(element1, NOT(element2), carry_in);
carry_out = nzcv<1>;
Elem[result, 2*p + 0, esize] = res;
Elem[result, 2*p + 1, esize] = ZeroExtend(carry_out, esize);
Z[da, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.454   SBCLT

Subtract the odd-numbered elements of the first source vector and the inverted 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector from the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 | 1
[0 1 0 0 0 1 0 1 | 1 | sz | 0] Zm       [1 1 0 1 0 1] Zn Zda
```

**Encoding**

SBCLT `<Zda>.<T>, <Zn>.<T>, <Zm>.<T>`

**Decode for this encoding**

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

**Assembler symbols**

- `<Zda>` is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- `<T>` is the size specifier, encoded in the "sz" field. It can have the following values:
  - S when sz = 0
  - D when sz = 1
- `<Zn>` is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand = Z[n, VL];
bits(VL) carries = Z[m, VL];
bits(VL) result = Z[da, VL];
for p = 0 to pairs-1
  bits(esize) element1 = Elem[result, 2*p + 0, esize];
bits(esize) element2 = Elem[operand, 2*p + 1, esize];
  bit carry_in = Elem[carries, 2*p + 1, esize]<0>;
  (res, nzcv) = AddWithCarry(element1, NOT(element2), carry_in);
  carry_out = nzcv<1>;
  Elem[result, 2*p + 0, esize] = res;
  Elem[result, 2*p + 1, esize] = ZeroExtend(carry_out, esize);
Z[da, VL] = result;
```
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.455 SCLAMP

Clamp each signed element in the destination vector to between the signed minimum value in the corresponding element of the first source vector and the signed maximum value in the corresponding element of the second source vector and destructively write the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2
(FEAT_SME)

Encoding
SCLAMP <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding
if !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[d, VL];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, e, esize]);
  integer element2 = SInt(Elem[operand2, e, esize]);
  integer element3 = SInt(Elem[operand3, e, esize]);
  integer res = Min(Max(element1, element3), element2);
  Elem[result, e, esize] = res<esize-1:0>;
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.456  SCVTF

Convert to floating-point from the signed integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

16-bit to half-precision

```
0 1 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1
```

```
16-bit to half-precision
Encoding
SCVTF <Zd>.H, <Pg>/M, <Zn>.H

Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```
```
011001010
0x0 1
0x0 0100
0x0 1
0x0 0101
0x0 Pg
0x0 Zn
0x0 Zd
```

Encoding

SCVTF <Zd>.H, <Pg>/M, <Zn>.H

Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```
```
011001010
0x0 1
0x0 0101
0x0 0
0x0 0101
0x0 Pg
0x0 Zn
0x0 Zd
```

32-bit to half-precision

```
0 1 1 0 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

```
32-bit to half-precision
Encoding
SCVTF <Zd>.H, <Pg>/M, <Zn>.H

Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```
```
011001010
0x0 1
0x0 0100
0x0 0
0x0 0101
0x0 Pg
0x0 Zn
0x0 Zd
```

Encoding

SCVTF <Zd>.H, <Pg>/M, <Zn>.H

Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```
32-bit to single-precision

Encoding
SCVTF <Zd>.S, <Pg>/M, <Zn>.S

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

32-bit to double-precision

Encoding
SCVTF <Zd>.D, <Pg>/M, <Zn>.S

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

64-bit to half-precision

Encoding
SCVTF <Zd>.H, <Pg>/M, <Zn>.D
Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

64-bit to single-precision

Encoding
SCVTF <Zd>.S, <Pg>/M, <Zn>.D

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

64-bit to double-precision

Encoding
SCVTF <Zd>.D, <Pg>/M, <Zn>.D

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    bits(d_esize) fpval = FixedToFP(element<s_esize-1:0>, 0, unsigned, FPCR[], rounding, d_esize);
    Elem[result, e, esize] = ZeroExtend(fpval, esize);
  Z[d, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.457   SDIV

Signed divide active elements of the first source vector by corresponding elements of the second source vector and
destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the
destination vector register remain unmodified.

Encoding
SDIV <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

Assembler symbols
<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size<0>" field. It can have the following values:
    S when size<0> = 0
    D when size<0> = 1
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if Elem[mask, e, esize] == '1' then
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer quotient;
        if element2 == 0 then
            quotient = 0;
            else
                quotient = RoundTowardsZero(Real(element1) / Real(element2));
        Elem[result, e, esize] = quotient-esize-1:0;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.458  SDIVR

Signed reversed divide active elements of the second source vector by corresponding elements of the first source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10  9 |  5  4 |  0 |
| 0  0  0  0 | 1  0  1  1 | 0  0  0  0 |   Pg | Zm  | Zdn |
```

**Encoding**

SDIVR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

**Assembler symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size<0>" field. It can have the following values:
  - `S` when size<0> = 0
  - `D` when size<0> = 1
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  if Elem[mask, e, esize] == '1' then
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    integer quotient;
    if element1 == 0 then
      quotient = 0;
    else
      quotient = RoundTowardsZero(Real(element2) / Real(element1));
    end;
    Elem[result, e, esize] = quotient<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
end;
```
The instruction might be immediately preceded in program order by a MOVPREFIX instruction. The MOVPREFIX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPREFIX and this instruction is UNPREDICTABLE:

- The MOVPREFIX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPREFIX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.459  SDOT (indexed)

The signed integer indexed dot product instruction computes the dot product of a group of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four signed 8-bit or 16-bit integer values in an indexed 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group. This instruction is unpredicated.

32-bit

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 16 15 14 13 12|11 10 9 | 5 4 | 0 |
| 0 1 0 0 0 1 0 0 1 0 1 1 2 | Zm 0 0 0 0 0 0 | Zn | Zda |
```

Encoding


Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

64-bit

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 16 15 14 13 12|11 10 9 | 5 4 | 0 |
| 0 1 0 0 0 1 0 0 1 0 1 1 1 1 | Zm 0 0 0 0 0 0 | Zn | Zda |
```

Encoding


Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.
For the 64-bit variant: is the immediate index of a 64-bit group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the "i1" field.

### Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
  integer segmentbase = e - (e MOD eltspersegment);
  integer s = segmentbase + index;
  bits(esize) res = Elem[operand3, e, esize];
  for i = 0 to 3
    integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
    integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
    res = res + element1 * element2;
  Elem[result, e, esize] = res;

Z[da, VL] = result;

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.460 SDOT (vectors)

The signed integer dot product instruction computes the dot product of a group of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four signed 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

This instruction is unpredicated.

```
[31 30 29 28 27 26 25 24 23 22 21 20] 16 15 14 13 12 11 10 9  5  4  0
0 1 0 0 0 1 0 0  size 0  Zm  0 0 0 0 0 0  Zn  Zda
U
```

Encoding

```
SDOT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in the "size<0>" field. It can have the following values:

S when size<0> = 0
D when size<0> = 1

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size<0>" field. It can have the following values:

B when size<0> = 0
H when size<0> = 1

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
```
res = res + element1 * element2;
Elem[result, e, esize] = res;

Z[da, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.461 SEL (predicates)

Read active elements from the first source predicate and inactive elements from the second source predicate and place in the corresponding elements of the destination predicate. Does not set the condition flags.

This instruction is used by the alias MOV (predicate, predicated, merging). See *Alias conditions* for details of when each alias is preferred.

Encoding

SEL <Pd>.B, <Pg>, <Pn>.B, <Pm>.B

Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
```

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (predicate, predicated, merging)</td>
<td>Pd == Pm</td>
</tr>
</tbody>
</table>

Assembler symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP(operand1, e, esize);
    bit element2 = ElemP(operand2, e, esize);
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1;
    else
```

```
```
\texttt{ElemP[result, e, esize] = element2;}

\texttt{P[d, PL] = result;}

\textbf{Operational information}

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.462 SEL (vectors)

Select elements from the first source vector where the corresponding vector select predicate element is true, and from the second source vector where the predicate element is false, placing them in the corresponding elements of the destination vector.

This instruction is used by the alias MOV (vector, predicated). See Alias conditions for details of when each alias is preferred.

Encoding

SEL <Zd>.<T>, <Pv>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer v = UInt(Pv);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (vector, predicated)</td>
<td>Zd == Zm</td>
</tr>
</tbody>
</table>

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

<table>
<thead>
<tr>
<th>Character</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>size = 00</td>
</tr>
<tr>
<td>H</td>
<td>size = 01</td>
</tr>
<tr>
<td>S</td>
<td>size = 10</td>
</tr>
<tr>
<td>D</td>
<td>size = 11</td>
</tr>
</tbody>
</table>

<Pv> Is the name of the vector select predicate register, encoded in the "Pv" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```c
CheckSVEEnabled();
classic integer VL = CurrentVL;
classic integer PL = VL DIV 8;
classic integer elements = VL DIV esize;
bits(PL) mask = P[v, PL];
```
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(NOT(mask), esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    Elem[result, e, esize] = Elem[operand2, e, esize];

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.463  SETFFR

Initialise the first-fault register (FFR) to all true prior to a sequence of first-fault or non-fault loads. This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

```
[31 30 29 28][27 26 25 24][23 22 21 20][19 18 17 16][15 14 13 12][11 10 9 8][7 6 5 4][3 2 1 0]
0 0 1 0 0 1 0 1 0 0 1 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

**Encoding**

SETFFR

**Decode for this encoding**

```cpp
if !HaveSVE() then UNDEFINED;
```

**Operation**

```cpp
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
FFR[PL] = Ones(PL);
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.464  SHADD

Add active signed elements of the first source vector to corresponding signed elements of the second source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

| 0 1 0 0 0 1 0 0 | size | 0 1 0 0 0 0 1 0 | Pg | Zm | Zdn |

Decode for this encoding

if !HaveSVE2() & !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if Elem[mask, e, esize] == '1' then
        integer res = (element1 + element2) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.465 SHRNB

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Encoding

SHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
B when tszh = 0, tszl = 01
H when tszh = 0, tszl = 1x
S when tszh = 1, tszl = xx

The encoding tszh = 0, tszl = 00 is reserved.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
H when tszh = 0, tszl = 01
S when tszh = 0, tszl = 1x
D when tszh = 1, tszl = xx

The encoding tszh = 0, tszl = 00 is reserved.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = UInt(element) >> shift;
  Elem[result, 2*e + 0, esize] = res<esize-1:0>;
  Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.466   SHRNT

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Encoding

SHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
B when tszh = 0, tszl = 0
H when tszh = 0, tszl = 1
S when tszh = 1, tszl = 0
The encoding tszh = 0, tszl = 00 is reserved.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
H when tszh = 0, tszl = 0
S when tszh = 0, tszl = 1
D when tszh = 1, tszl = 0
The encoding tszh = 0, tszl = 00 is reserved.
<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = UInt(element) >> shift;
  Elem[result, 2*e + 1, esize] = res<esize-1:0>;

Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.467   SHSUB

Subtract active signed elements of the second source vector from corresponding signed elements of the first source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

<table>
<thead>
<tr>
<th>size</th>
<th>Zdn</th>
<th>Zm</th>
<th>Pg</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Encoding**

SHSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << Uint(size);
integer g = Uint(Pg);
integer dn = Uint(Zdn);
integer m = Uint(Zm);

**Assembler symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if Elem[mask, e, esize] == '1' then
        integer res = (element1 - element2) >> 1;
    Elem[result, e, esize] = res<esize-1:0>;
else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.468   SHSUBR

Subtract active signed elements of the first source vector from corresponding signed elements of the second source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

SHSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if ![HaveSVE2()] && ![HaveSME()] then UNDEFINED;

constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn>     Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>      Is the size specifier, encoded in the "size" field. It can have the following values:
       B  when size = 00
       H  when size = 01
       S  when size = 10
       D  when size = 11

<Pg>     Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>     Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if Elem[mask, e, esize] == '1' then
        integer res = (element2 - element1) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.469  SLI

Shift each source vector element left by an immediate value, and insert the result into the corresponding vector element in the destination vector register, merging the shifted bits from each source element with existing bits in each destination vector element. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

Encoding

SLI <Zd>.<T>, <Zn>.<T>, #<const>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
b(4) tsize = tszh:tszl;
integer esize;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer shift = UInt(tsize:imm3) - esize;

Assembler symbols

<Zd>    Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>    Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  B    when tszh = 00, tszl = 01
  H    when tszh = 00, tszl = 1x
  S    when tszh = 01, tszl = xx
  D    when tszh = 1x, tszl = xx
The encoding tszh = 00, tszl = 00 is reserved.
<Zn>    Is the name of the first source scalable vector register, encoded in the "Zn" field.
<const>    Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
b(VL) operand = Z[n, VL];
b(VL) result = Z[d, VL];
for e = 0 to elements-1
  bits(esize) element1 = Elem[result, e, esize];
bits(esize) element2 = Elem[operand, e, esize];
bits(esize) mask = LSL(Ones(esize), shift);
bits(esize) shiftedval = LSL(element2, shift);
Elem[result, e, esize] = (element1 AND (NOT mask)) OR shiftedval;

Z[d, VL] = result;

Operational information
If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.470  SM4E

The SM4E instruction reads 16 bytes of input data from each 128-bit segment of the first source vector, together with four iterations of 32-bit round keys from the corresponding 128-bit segments of the second source vector. Each block of data is encrypted by four rounds in accordance with the SM4 standard, and destructively placed in the corresponding segments of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.SM4 indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2

(FEAT_SVE_SM4)

Encoding

| SM4E | <Zdn>.S, <Zdn>.S, <Zm>.S |

Decode for this encoding

```c
if !HaveSVE() || !HaveSVE2SM4() then UNDEFINED;
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler symbols

- `<Zdn>` is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for s = 0 to segments-1
    bits(128) key = Elem[operand2, s, 128];
    bits(32) intval;
    bits(8) sboxout;
    bits(128) roundresult = Elem[operand1, s, 128];
    bits(32) roundkey;
    for index = 0 to 3
        roundkey = Elem[key, index, 32];
        intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;
        for i = 0 to 3
            Elem[intval, i,8] = Sbox(Elem[intval, i,8]);
        intval = intval EOR ROL(intval, 2) EOR ROL(intval,10) EOR ROL(intval,18) EOR ROL(intval, 24);
```
intval = intval EOR roundresult<31:0>;
roundresult<31:0> = roundresult<63:32>;
roundresult<63:32> = roundresult<95:64>;
roundresult<95:64> = roundresult<127:96>;
roundresult<127:96> = intval;

Elem[result, s, 128] = roundresult;

Z[dn, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.471   SM4EKEY

The SM4EKEY instruction reads four rounds of 32-bit input key values from each 128-bit segment of the first source vector, along with four rounds of 32-bit constants from the corresponding 128-bit segment of the second source vector. The four rounds of output key values are derived in accordance with the SM4 standard, and placed in the corresponding segments of the destination vector. This instruction is unpredicated.

ID_AA64ZFR0_ELM1.SM4 indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2

(Feat_SVE_SM4)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1</td>
<td>Zm</td>
<td>1 1 1 0 0</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SM4EKEY <Zd>.S, <Zn>.S, <Zm>.S

Decode for this encoding

if !HaveSVE() || !HaveSVE2SM4() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd>       Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>       Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>       Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL div 128;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for s = 0 to segments - 1
    bits(128) source = Elem[operand2, s, 128];
bits(32) intval;
bits(8) sboxout;
bits(32) const;
bits(128) roundresult = Elem[operand1, s, 128];
    for index = 0 to 3
        const = Elem[source, index, 32];
        intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;
        for i = 0 to 3
            Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);
intval = intval EOR ROL(intval, 13) EOR ROL(intval, 23);
intval = intval EOR roundresult<31:0>;
roundresult<31:0> = roundresult<63:32>;
roundresult<63:32> = roundresult<95:64>;
roundresult<95:64> = roundresult<127:96>;
roundresult<127:96> = intval;

Elem[result, s, 128] = roundresult;
Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.472  SMAX (immediate)

Determine the signed maximum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.

Encoding

```
<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>[27 26 25 24]</th>
<th>[23 22 21 20]</th>
<th>[19 18 17 16]</th>
<th>[15 14 13 12]</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1</td>
<td>1 0 1 0 0 1 1 0</td>
<td>imm8</td>
<td>Zdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;
integer imm = Int(imm8, unsigned);
```

Assembler symbols

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<imm>` Is the signed immediate operand, in the range -128 to 127, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  Elem[result, e, esize] = Max(element1, imm)<esize-1:0>;
Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.473  SMAX (vectors)

Determine the signed maximum of active elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

SMAX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  integer element2 = Int(Elem[operand2, e, esize], unsigned);
  if Elem[mask, e, esize] == '1' then
    integer maximum = Max(element1, element2);
    Elem[result, e, esize] = maximum<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
\[ Z[dn, VL] = \text{result}; \]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.474  SMAXP

Compute the maximum value of each pair of adjacent signed integer elements within each source vector, and
interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first
source vector.

Encoding

SMAXP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

B  when size = 00
H  when size = 01
S  when size = 10
D  when size = 11

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer element1;
integer element2;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '0' then
    Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    if IsEven(e) then
      element1 = SInt(Elem[operand1, e + 0, esize]);
      element2 = SInt(Elem[operand1, e + 1, esize]);
    else
      element1 = SInt(Elem[operand2, e - 1, esize]);
element2 = SInt(Elem[operand2, e + 0, esize]);
integer res = Max(element1, element2);
Elem[result, e, esize] = res<esize-1:0>;
Z[dn, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.475   SMAXV

Signed maximum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the minimum signed integer for the element size.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0</td>
<td>0 0 1 0 0</td>
<td>0 0 0 1 0</td>
<td>Pg Zn Vd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

SMAXV <V><d>, <Pg>, <Zn>.<T>

**Decode for this encoding**

if !HaveSVE() &amp; !HaveSME() then UNDEFINED;
constant integer esize = 8 &lt;&lt; UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = FALSE;

**Assembler symbols**

&lt;V&gt; Is a width specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

&lt;d&gt; Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

&lt;Pg&gt; Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

&lt;Zn&gt; Is the name of the source scalable vector register, encoded in the "Zn" field.

&lt;T&gt; Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
integer maximum = if unsigned then 0 else -(2^(esize-1));
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = Int(Elem[operand, e, esize], unsigned);
        maximum = Max(maximum, element);
$V[d, esize] = \text{maximum}_{esize-1:0}$;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.476   SMIN (immediate)

Determine the signed minimum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.

Encoding

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1</td>
<td>0 1 1 0</td>
<td>imm8</td>
<td>Zdn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

SMIN <Zdn>.<T>, <Zdn>.<T>, #<imm>

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;
integer imm = Int(imm8, unsigned);
```

**Assembler symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<imm>` Is the signed immediate operand, in the range -128 to 127, encoded in the "imm8" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
   integer element1 = Int(Elem[operand1, e, esize], unsigned);
   Elem[result, e, esize] = Min(element1, imm)<esize-1:0>;
Z[dn, VL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
The response of this instruction to asynchronous exceptions does not vary based on:

— The values of the data supplied in any of its registers.
— The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMIN (vectors)

Determine the signed minimum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

**Encoding**

SMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

**Assembler symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  integer element2 = Int(Elem[operand2, e, esize], unsigned);
  if Elem[mask, e, esize] == '1' then
    integer minimum = Min(element1, element2);
    Elem[result, e, esize] = minimum<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
\[ Z[dn, VL] = \text{result}; \]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.478 SMINP

Compute the minimum value of each pair of adjacent signed integer elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 |
|-------------|-----------|-------------|-----------|-----------|------|-----|--|---|
| 0 1 0 0 0 1 0 0 | size 0 1 0 1 0 1 0 1 | Pg | Zm | Zdn |
```

**Encoding**

SMINP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

**Assembler symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer element1;
integer element2;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '0' then
    Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    if IsEven(e) then
      element1 = SInt(Elem[operand1, e + 0, esize]);
      element2 = SInt(Elem[operand1, e + 1, esize]);
    else
      element1 = SInt(Elem[operand2, e - 1, esize]);
element2 = SInt(Elem[operand2, e + 0, esize]);
integer res = Min(element1, element2);
Elem[result, e, esize] = res<esize-1:0>;
Z[dn, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.479  SMINV

Signed minimum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the maximum signed integer for the element size.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>0 1 0</td>
<td>0 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Vd</td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

SMINV <V><d>, <Pg>, <Zn>.<T>

**Decode for this encoding**

if !HaveSVE() & !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = FALSE;

**Assembler symbols**

<V> Is a width specifier, encoded in the "size" field. It can have the following values:
- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bounds(PL) mask = P[g, PL];
bounds(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
integer minimum = if unsigned then (2^esize - 1) else (2^(esize-1) - 1);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer element = Int(Elem[operand, e, esize], unsigned);
    minimum = Min(minimum, element);
V[d, esize] = minimum<esize-1:0>;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.480  SMLALB (indexed)

Multiply the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment and destructively add to the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

32-bit

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;

64-bit

Encoding

SMLALB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
Assembler symbols

<Zda> is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation for all encodings

CheckSVEEnabled();
classic integer VL = CurrentVL;
classic integer PL = VL DIV 8;
classic integer elements = VL DIV (2 * esize);
classic integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<<2*esize-1:0;,
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + product;
Z[da, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
    — The values of the data supplied in any of its registers.
    — The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
    — The values of the data supplied in any of its registers.
    — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.481 SMLALB (vectors)

Multiply the corresponding even-numbered signed elements of the first and second source vectors and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

Encoding

SMLALB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + product;
$Z[da, VL] = \text{result};$

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SVE Instruction Descriptions
C8.2 Alphabetical list of SVE instructions

C8.2.482  SMLALT (indexed)

Multiply the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified
signed element in the corresponding second source vector segment and destructively add to the overlapping
double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same
element position within each 128-bit vector segment. The index range is from 0 to one less than the number of
elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

32-bit

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>1 0 1</td>
<td>i3h</td>
<td>Zm</td>
<td>1 0 0 0</td>
<td>3</td>
<td>1</td>
<td>Zn</td>
</tr>
</tbody>
</table>

size<1>  size<0>

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;

64-bit

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>1 0</td>
<td>1 1</td>
<td>Zm</td>
<td>1 0 0 0</td>
<td>2</td>
<td>1</td>
<td>Zn</td>
</tr>
</tbody>
</table>

size<1>  size<0>
i2h

Encoding

SMLALT <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
  integer s = e - (e MOD eltspersegment);
  integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
  integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
  bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
  Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + product;

Z[da, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.483  **SMLALT (vectors)**

Multiply the corresponding odd-numbered signed elements of the first and second source vectors and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 0 0 0 1 0 0 | size 0 | Zm 0 1 0 0 1 | Zn | Zda

S U T
```

**Encoding**

SMLALT `<Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>`

**Decode for this encoding**

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

**Assembler symbols**

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - `H` when size = 01
  - `S` when size = 10
  - `D` when size = 11
  The encoding size = 00 is reserved.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - `B` when size = 01
  - `H` when size = 10
  - `S` when size = 11
  The encoding size = 00 is reserved.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
  integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
  bits(esize) product = (element1 * element2)<esize-1:0>;
  Elem[result, e, esize] = Elem[result, e, esize] + product;
```
Z[da, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.484 SMLSLB (indexed)

Multiply the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment and destructively subtract from the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

**32-bit**

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 14 13 12 11 10 9 5 4 0]
0 1 0 0 0 1 0 0 1 0 1 i3h Zm 1 0 0 1 0 i3l Zn Zda

size<1> size<0> S U T
```

**Encoding**


**Decode for this encoding**

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

**64-bit**

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9 5 4 0]
0 1 0 0 0 1 0 0 1 1 Zm 1 0 1 0 2 0 Zn Zda

size<1> size<0> i2h S U T
```

**Encoding**

SMLSLB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

**Decode for this encoding**

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```
Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation for all encodings

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] - product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SVE Instruction Descriptions
C8.2 Alphabetical list of SVE instructions

C8.2.485 SMLSLB (vectors)

Multiply the corresponding even-numbered signed elements of the first and second source vectors and destructively subtract from the overlapping double-width elements of the addend vector. This instruction is unpredicated.

\[
\begin{array}{ccccccccccccccccccc}
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & Zm & 0 & 1 & 0 & 1 & 0 & 0 & Zn & Zda \\
\end{array}
\]

**Encoding**

SMLSLB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

**Assembler symbols**

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 01
- H when size = 10
- S when size = 11

The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
bits(esize) product = (element1 * element2)<esize-1:0>;
Elem[result, e, esize] = Elem[result, e, esize] - product;
Z[da, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is **UNPREDICTABLE**:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.486   SMLSLT (indexed)

Multiply the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment and destructively subtract from the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

32-bit

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18</th>
<th>16 15 14 13</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>i3h</td>
<td>Zm</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Encoding**


**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;

64-bit

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19</th>
<th>16 15 14 13</th>
<th>12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Zm</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Encoding**


**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] - product;
    Z[da, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SVE Instruction Descriptions
C8.2 Alphabetical list of SVE instructions

C8.2.487   SMLSLT (vectors)

Multiply the corresponding odd-numbered signed elements of the first and second source vectors and destructively subtract from the overlapping double-width elements of the addend vector. This instruction is unpredicated.

```
| 31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 5 4 | 0 |
| 0 1 0 0 0 1 0 0 | size 0 | Zm 0 1 0 1 | Zn | Zda |
```

Encoding

SMLSLT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

- if !HaveSVE2() && !HaveSME() then UNDEFINED;
- if size == '00' then UNDEFINED;
- constant integer esize = 8 << UInt(size);
- integer n = UInt(Zn);
- integer m = UInt(Zm);
- integer da = UInt(Zda);

Assembler symbols

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - `H` when size = 01
  - `S` when size = 10
  - `D` when size = 11
  - The encoding size = 00 is reserved.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - `B` when size = 01
  - `H` when size = 10
  - `S` when size = 11
  - The encoding size = 00 is reserved.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
  integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
  bits(esize) product = (element1 * element2)<<esize-1:0>
  Elem[result, e, esize] = Elem[result, e, esize] - product;
```
Z[da, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.488   SMMLA

The signed integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of signed 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of signed 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE

(FEAT_I8MM)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10  9</th>
<th>5  4  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1</td>
<td>0 0 0</td>
<td>Zm</td>
</tr>
<tr>
<td>1 0 0 1 1 0</td>
<td></td>
<td>Zn</td>
</tr>
</tbody>
</table>

Encoding


Decode for this encoding

if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result = Zeros(VL);
bits(128) op1, op2;
bits(128) res, addend;
for s = 0 to segments-1
    op1 = Elem[operand1, s, 128];
    op2 = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
res    = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
_elem[result, s, 128] = res;

Z[da, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.489 SMULH (predicated)

Widening multiply signed integer values in active elements of the first source vector by corresponding elements of
the second source vector and destructively place the high half of the result in the corresponding elements of the first
source vector. Inactive elements in the destination vector register remain unmodified.

**Encoding**

SMULH <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

**Assembler symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  integer element2 = Int(Elem[operand2, e, esize], unsigned);
  if Elem[mask, e, esize] == '1' then
    integer product = (element1 * element2) >> esize;
    Elem[result, e, esize] = product<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.490   SMULH (unpredicated)

Widening multiply signed integer values of all elements of the first source vector by corresponding elements of the second source vector and place the high half of the result in the corresponding elements of the destination vector. This instruction is unpredicated.

Encoding

SMULH <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = FALSE;

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    integer product = (element1 * element2) >> esize;
    Elem[result, e, esize] = product<esize-1:0>;
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.491 SMULLB (indexed)

Multiply the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, and place the results in the overlapping double-width elements of the destination vector register.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

32-bit

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 0 1</td>
<td>i3h</td>
<td>Zm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

size<1> size<0>

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;

64-bit

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 1</td>
<td>Zm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>Zn</td>
</tr>
</tbody>
</table>

size<1> size<0> i2h

Encoding

SMULLB <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;
Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  integer s = e - (e MOD eltspersegment);
  integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
  integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
  integer res = element1 * element2;
  Elem[result, e, 2*esize] = res<2*esize-1:0>;
Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.492   SMULLB (vectors)

Multiply the corresponding even-numbered signed elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1</td>
<td>size</td>
<td>Zm</td>
<td>0 1 1</td>
<td>1 0 0</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SMULLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
  integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
  integer res = element1 * element2;
  Elem[result, e, esize] = res<esize-1:0>;
\[ Z[d, VL] = \text{result}; \]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.493   SMULLT (indexed)

Multiply the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified
element in the corresponding second source vector segment, and place the results in the overlapping double-width
elements of the destination vector register.

The elements within the second source vector are specified using an immediate index which selects the same
element position within each 128-bit vector segment. The index range is from 0 to one less than the number of
elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

32-bit

[31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 ]

0 1 0 0 0 1 0 0 | 1 0 1 i3h Zm 1 1 0 0 | 3| 1 Zn Zd

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;

64-bit

[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12|11 10 9 | 5 4 | 0 ]

0 1 0 0 0 1 0 0 | 1 1 1 | Zm 1 1 0 0 | 2| 1 Zn Zd

Encoding

SMULLT <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;


Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>  For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm>  For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

**Operation for all encodings**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer res = element1 * element2;
    Elem[result, e, 2*esize] = res<2*esize-1:0>;
Z[d, VL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.494  SMULLT (vectors)

Multiply the corresponding odd-numbered signed elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

\[
\begin{array}{cccccccccccccccccccc}
0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

**Encoding**

SMULLT <Zd>,<T>, <Zn>,<Tb>, <Zm>,<Tb>

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
ineger n = UInt(Zn);
ineger m = UInt(Zm);
ineger d = UInt(Zd);

**Assembler symbols**

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

H  when size = 01
S  when size = 10
D  when size = 11
The encoding size = 00 is reserved.

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb>  Is the size specifier, encoded in the "size" field. It can have the following values:

B  when size = 01
H  when size = 10
S  when size = 11
The encoding size = 00 is reserved.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
integer res = element1 * element2;
Elem[result, e, esize] = res<esize-1:0>;
\[ Z[d, \text{VL}] = \text{result}; \]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.495 SPLICE

Select a region from the first source vector and copy it to the lowest-numbered elements of the result. Then set any remaining elements of the result to a copy of the lowest-numbered elements from the second source vector. The region is selected using the first and last true elements in the vector select predicate register. The result is placed destructively in the destination and first source vector, or constructively in the destination vector.

The Destructive encoding of this instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE: The MOVPRFX instruction must be unpredicated. The MOVPRFX instruction must specify the same destination register as this instruction. The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

**Constructive**

![Constructive Encoding Diagram]

**Encoding**

SPLICE <Zd>.<T>, <Pv>, { <Zn1>.<T>, <Zn2>.<T> }

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer v = UInt(Pv);
integer dst = UInt(Zd);
integer s1 = UInt(Zn);
integer s2 = (s1 + 1) MOD 32;

**Destructive**

![Destructive Encoding Diagram]

**Encoding**

SPLICE <Zdn>.<T>, <Pv>, <Zdn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer v = UInt(Pv);
integer dst = UInt(Zdn);
integer s1 = dst;
integer s2 = UInt(Zm);

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
The size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

The name of the vector select predicate register P0-P7, encoded in the "Pv" field.

Is the name of the first scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

Is the name of the second scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation for all encodings**

```plaintext
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[v, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[s1, VL] else Zeros(VL);
bits(VL) operand2 = Z[s2, VL];
bits(VL) result;
integer x = 0;
boolean active = FALSE;
constant integer lastnum = LastActiveElement(mask, esize);

if lastnum >= 0 then
    for e = 0 to lastnum
        active = active || ElemP[mask, e, esize] == '1';
    if active then
        Elem[result, x, esize] = Elem[operand1, e, esize];
        x = x + 1;

constant integer nelements = (elements - x) - 1;
for e = 0 to nelements
    Elem[result, x, esize] = Elem[operand2, e, esize];
    x = x + 1;

Z[dst, VL] = result;
```
Compute the absolute value of the signed integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Inactive elements in the destination vector register remain unmodified.

| [31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 ]
---|---|---|---|---|---|---|---|---|
 0 1 0 0 0 1 0 0 | 0 0 1 0 0 1 0 1 | Pg | Zn | Zd |

**Encoding**

SQABS <Zd>.<T>, <Pg>/M, <Zn>.<T>

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

**Asmbleer symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer element = SInt(Elem[operand, e, esize]);
    element = Abs(element);
    Elem[result, e, esize] = SignedSat(element, esize);
Z[d, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.497 SQADD (immediate)

Signed saturating add of an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

Encoding

SQADD <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = FALSE;
```

Assembler symbols

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size == 00
  - H when size == 01
  - S when size == 10
  - D when size == 11
- `<imm>` Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:
  - LSL #0 when sh == 0
  - LSL #8 when sh == 1

Operation

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
```
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + imm, esize, unsigned);
    Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.498 SQADD (vectors, predicated)

Add active signed elements of the first source vector to corresponding signed elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Inactive elements in the destination vector register remain unmodified.

Encoding

SQADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
   B when size = 00
   H when size = 01
   S when size = 10
   D when size = 11
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
   integer element1 = SInt(Elem[operand1, e, esize]);
   integer element2 = SInt(Elem[operand2, e, esize]);
   if Elem[mask, e, esize] == '1' then
      integer res = SInt(Sat(element1 + element2, esize, unsigned));
      Elem[result, e, esize] = res<esize-1:0>;
   else
      Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is **UNPREDICTABLE**:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.499  SQADD (vectors, unpredicated)

Signed saturating add all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range $-2^{{(N-1)}}$ to $(2^{{(N-1)}})-1$. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>size 1</td>
<td>Zm</td>
<td>0 0 0 1 0 0</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SQADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = $8 <<$UInt(size);
integer n = $UInt(Zn);
integer m = $UInt(Zm);
integer d = $UInt(Zd);
boolean unsigned = FALSE;

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>  Is the size specifier, encoded in the "size" field. It can have the following values:
  B  when size = 00
  H  when size = 01
  S  when size = 10
  D  when size = 11

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  integer element2 = Int(Elem[operand2, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element1 + element2, esize, unsigned);

Z[d, VL] = result;
C8.2.500 SQCADD

Add the real and imaginary components of the integral complex numbers from the first source vector to the complex numbers from the second source vector which have first been rotated by 90 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, equivalent to multiplying the complex numbers in the second source vector by ±i beforehand. Destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element’s signed integer range -2(N-1) to (2(N-1))-1. This instruction is unpredicated.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

Encoding

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
boolean sub_i = (rot == '0');
boolean sub_r = (rot == '1');

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in the "rot" field. It can have the following values:
- #90 when rot = 0
- #270 when rot = 1

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for p = 0 to pairs-1
integer acc_r = SInt((Icons[operand1, 2 * p + 0, esize]));
integer acc_i = SInt((Icons[operand1, 2 * p + 1, esize]));
integer elt2_r = SInt((Icons[operand2, 2 * p + 0, esize]));
integer elt2_i = SInt((Icons[operand2, 2 * p + 1, esize]));
if sub_i then
  acc_r = acc_r - elt2_i;
  acc_i = acc_i + elt2_r;
if sub_r then
  acc_r = acc_r + elt2_i;
  acc_i = acc_i - elt2_r;

Elem[result, 2 * p + 0, esize] = SignedSat(acc_r, esize);
Elem[result, 2 * p + 1, esize] = SignedSat(acc_i, esize);

Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.501  SQDECB

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

32-bit

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
0 0 0 0 0 1 0 0 0 0 1 0 imm4 1 1 1 1 1 0 pattern Rdn
size<1> sf D U
size<0>
```

Encoding

SQDECB <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}</td>

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

64-bit

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
0 0 0 0 0 1 0 0 0 0 1 1 imm4 1 1 1 1 1 0 pattern Rdn
size<1> sf D U
size<0>
```

Encoding

SQDECB <Xdn>{, <pattern>{, MUL #<imm>}}</td>

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
```
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

Assembler symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

- **POW2** when pattern = 00000
- **VL1** when pattern = 00001
- **VL2** when pattern = 00010
- **VL3** when pattern = 00011
- **VL4** when pattern = 00100
- **VL5** when pattern = 00101
- **VL6** when pattern = 00110
- **VL7** when pattern = 00111
- **VL8** when pattern = 01000
- **VL16** when pattern = 01001
- **VL32** when pattern = 01010
- **VL64** when pattern = 01011
- **VL128** when pattern = 01100
- **VL256** when pattern = 01101
- **#uimm5** when pattern = 0111x
- **#uimm5** when pattern = 101x1
- **#uimm5** when pattern = 10110
- **#uimm5** when pattern = 1x0x1
- **#uimm5** when pattern = 1x010
- **#uimm5** when pattern = 1xx00
- **MUL4** when pattern = 11101
- **MUL3** when pattern = 11110
- **ALL** when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation for all encodings

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```
C8.2.502 SQDECD (scalar)

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:
- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

### 32-bit

```
[31 30 29 28][27 26 25 24][23 22 21 20][19 16][15 14 13 12][11 10 9] | 5 4 | 0
0 0 0 0 0 1 0 0 1 1 1 0 imm4 1 1 1 1 1 0 pattern Rdn
```

**Encoding**

SQDECD <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

### 64-bit

```
[31 30 29 28][27 26 25 24][23 22 21 20][19 16][15 14 13 12][11 10 9] | 5 4 | 0
0 0 0 0 0 1 0 0 1 1 1 1 imm4 1 1 1 1 1 0 pattern Rdn
```

**Encoding**

SQDECD <Xdn>{, <pattern>{, MUL #<imm>}}

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
```
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
icenter integer ssize = 64;

Assembler symbols

<Xdn>  Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Wdn>  Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern>  Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>POW2</td>
<td>pattern = 00000</td>
</tr>
<tr>
<td>VL1</td>
<td>pattern = 00001</td>
</tr>
<tr>
<td>VL2</td>
<td>pattern = 00010</td>
</tr>
<tr>
<td>VL3</td>
<td>pattern = 00011</td>
</tr>
<tr>
<td>VL4</td>
<td>pattern = 00100</td>
</tr>
<tr>
<td>VL5</td>
<td>pattern = 00101</td>
</tr>
<tr>
<td>VL6</td>
<td>pattern = 00110</td>
</tr>
<tr>
<td>VL7</td>
<td>pattern = 00111</td>
</tr>
<tr>
<td>VL8</td>
<td>pattern = 01000</td>
</tr>
<tr>
<td>VL16</td>
<td>pattern = 01001</td>
</tr>
<tr>
<td>VL32</td>
<td>pattern = 01010</td>
</tr>
<tr>
<td>VL64</td>
<td>pattern = 01011</td>
</tr>
<tr>
<td>VL128</td>
<td>pattern = 01100</td>
</tr>
<tr>
<td>VL256</td>
<td>pattern = 01101</td>
</tr>
<tr>
<td>#uimm5</td>
<td>pattern = 0111x</td>
</tr>
<tr>
<td>#uimm5</td>
<td>pattern = 101x1</td>
</tr>
<tr>
<td>#uimm5</td>
<td>pattern = 10110</td>
</tr>
<tr>
<td>#uimm5</td>
<td>pattern = 1x0x1</td>
</tr>
<tr>
<td>#uimm5</td>
<td>pattern = 1x010</td>
</tr>
<tr>
<td>#uimm5</td>
<td>pattern = 1xx00</td>
</tr>
<tr>
<td>MUL4</td>
<td>pattern = 11101</td>
</tr>
<tr>
<td>MUL3</td>
<td>pattern = 11110</td>
</tr>
<tr>
<td>ALL</td>
<td>pattern = 11111</td>
</tr>
</tbody>
</table>

<imm>  Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation for all encodings

CheckSVEEnabled();
icenter integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

center integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
C8.2.503   SQDECD (vector)  

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 64-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

```
Encoding
SQDECD <Zdn>.D{, <pattern>{, MUL #<imm>}}

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

Assembler symbols

- **<Zdn>** Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- **<pattern>** Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:
  - POW2 when pattern = 00000
  - VL1 when pattern = 00001
  - VL2 when pattern = 00010
  - VL3 when pattern = 00011
  - VL4 when pattern = 00100
  - VL5 when pattern = 00101
  - VL6 when pattern = 00110
  - VL7 when pattern = 00111
  - VL8 when pattern = 01000
  - VL16 when pattern = 01001
VL32 when pattern = 01010
VL64 when pattern = 01011
VL128 when pattern = 01100
VL256 when pattern = 01101
#uimm5 when pattern = 0111x
#uimm5 when pattern = 101x1
#uimm5 when pattern = 10110
#uimm5 when pattern = 1x0x1
#uimm5 when pattern = 1x010
#uimm5 when pattern = 1xx00
MUL4 when pattern = 11101
MUL3 when pattern = 11110
ALL when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bites(VL) operand1 = Z[dn, VL];
bites(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);
Z[dn, VL] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.504  **SQDECH (scalar)**

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

### 32-bit

![32-bit Encoding](image)

Decoding for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

### 64-bit

![64-bit Encoding](image)

Decoding for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
```
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

Assembler symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

- POW2 when pattern = 00000
- VL1 when pattern = 00001
- VL2 when pattern = 00010
- VL3 when pattern = 00011
- VL4 when pattern = 00100
- VL5 when pattern = 00101
- VL6 when pattern = 00110
- VL7 when pattern = 00111
- VL8 when pattern = 01000
- VL16 when pattern = 01001
- VL32 when pattern = 01010
- VL64 when pattern = 01011
- VL128 when pattern = 01100
- VL256 when pattern = 01101
- #uimm5 when pattern = 0111x
- #uimm5 when pattern = 101x1
- #uimm5 when pattern = 10110
- #uimm5 when pattern = 1x0x1
- #uimm5 when pattern = 1x010
- #uimm5 when pattern = 1xx00
- MUL4 when pattern = 11101
- MUL3 when pattern = 11110
- ALL when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation for all encodings

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
C8.2.505    SQDECH (vector)

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 16-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

```
Encoding
SQDECH <Zdn>.H{, <pattern>{, MUL #<imm>}}
```

```
Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

**Assembler symbols**

- `<Zdn>`    Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<pattern>` Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:
  - POW2     when pattern = 00000
  - VL1      when pattern = 00001
  - VL2      when pattern = 00010
  - VL3      when pattern = 00011
  - VL4      when pattern = 00100
  - VL5      when pattern = 00101
  - VL6      when pattern = 00110
  - VL7      when pattern = 00111
  - VL8      when pattern = 01000
  - VL16     when pattern = 01001
SVE Instruction Descriptions
C8.2 Alphabetical list of SVE instructions

VL32  when pattern = 01010
VL64  when pattern = 01011
VL128 when pattern = 01100
VL256 when pattern = 01101
#uimm5 when pattern = 0111x
#uimm5 when pattern = 101x1
#uimm5 when pattern = 10110
#uimm5 when pattern = 1x0x1
#uimm5 when pattern = 1x010
#uimm5 when pattern = 1xx00
MUL4  when pattern = 11101
MUL3  when pattern = 11110
ALL   when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);
Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.506  SQDECP (scalar)

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

32-bit

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 0 1 0 1</td>
<td>1 0 1 0 1 0 0 1 0</td>
<td>0 0</td>
<td>Pm</td>
<td>Rdn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SQDECP <Xd<dn>, <Pm>.<T>, <Wdn>

Decode for this encoding

if !HaveSVE() &amp; !HaveSME() then UNDEFINED;
constant integer esize = 8 &lt;&lt; UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
integer ssize = 32;

64-bit

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1</td>
<td>1 0 1 0 1 0 0 1 1</td>
<td>1 0</td>
<td>Pm</td>
<td>Rdn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SQDECP <Xd<dn>, <Pm>.<T>

Decode for this encoding

if !HaveSVE() &amp; !HaveSME() then UNDEFINED;
constant integer esize = 8 &lt;&lt; UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
integer ssize = 64;

Assembler symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11
Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

**Operation for all encodings**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn, ssize];
bits(PL) operand2 = P[m, PL];
bits(ssize) result;
integer count = 0;

for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;

integer element = Int(operand1, unsigned);
(result, -) = SatQ(element - count, ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

**Operational information**

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C8.2.507   SQDECP (vector)

Counts the number of true elements in the source predicate and then uses the result to decrement all destination
vector elements. The results are saturated to the element signed integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited
in a future release of the architecture.

Encoding

SQDECP <Zdn>.<T>, <Pm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = $ << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;

Assembler symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11

The encoding size = 00 is reserved.

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
const integer VL = CurrentVL;
const integer PL = VL DIV 8;
const integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
  if Elem[operand2, e, esize] == '1' then
    count = count + 1;
for e = 0 to elements-1
  integer element = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element - count, esize, unsigned);
Z[dn, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.508  SQDECW (scalar)

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

**32-bit**

<table>
<thead>
<tr>
<th>000001001</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>imm4</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>pattern</th>
<th>Rdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>size&lt;1&gt;</td>
<td>D</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size&lt;0&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

```
SQDECW <Xd>, <Wd>{, <pattern>{, MUL #<imm>}}
```

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

**64-bit**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 14 13</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0 1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>imm4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>size&lt;1&gt;</td>
<td>D</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size&lt;0&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

```
SQDECW <Xd>{, <pattern>{, MUL #<imm>}}
```

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
```
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

Assembler symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

- **POW2** when pattern = 00000
- **VL1** when pattern = 00001
- **VL2** when pattern = 00010
- **VL3** when pattern = 00011
- **VL4** when pattern = 00100
- **VL5** when pattern = 00101
- **VL6** when pattern = 00110
- **VL7** when pattern = 00111
- **VL8** when pattern = 01000
- **VL16** when pattern = 01001
- **VL32** when pattern = 01010
- **VL64** when pattern = 01011
- **VL128** when pattern = 01100
- **VL256** when pattern = 01101
- **#uimm5** when pattern = 0111x
- **#uimm5** when pattern = 101x1
- **#uimm5** when pattern = 10110
- **#uimm5** when pattern = 1x010
- **#uimm5** when pattern = 1xx00
- **MUL4** when pattern = 11101
- **MUL3** when pattern = 11110
- **ALL** when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation for all encodings

    CheckSVEEnabled();
    integer count = DecodePredCount(pat, esize);
    bits(ssize) operand1 = X[dn, ssize];
    bits(ssize) result;
    integer element1 = Int(operand1, unsigned);
    (result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
    X[dn, 64] = Extend(result, 64, unsigned);
C8.2.509  SQDECW (vector)

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 32-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

Encoding

SQDECW <Zdn>.S{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
ingteger dn = UInt(Zdn);
bits(5) pat = pattern;
ingteger imm = UInt(imm4) + 1;
boolean unsigned = FALSE;

Assembler symbols

<Zdn>  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern>  Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>POW2</td>
<td>00000</td>
</tr>
<tr>
<td>VL1</td>
<td>00001</td>
</tr>
<tr>
<td>VL2</td>
<td>00010</td>
</tr>
<tr>
<td>VL3</td>
<td>00011</td>
</tr>
<tr>
<td>VL4</td>
<td>00100</td>
</tr>
<tr>
<td>VL5</td>
<td>00101</td>
</tr>
<tr>
<td>VL6</td>
<td>00110</td>
</tr>
<tr>
<td>VL7</td>
<td>00111</td>
</tr>
<tr>
<td>VL8</td>
<td>01000</td>
</tr>
<tr>
<td>VL16</td>
<td>01001</td>
</tr>
</tbody>
</table>
VL32 when pattern = 01010
VL64 when pattern = 01011
VL128 when pattern = 01100
VL256 when pattern = 01101
#uimm5 when pattern = 0111x
#uimm5 when pattern = 101x1
#uimm5 when pattern = 10110
#uimm5 when pattern = 1x0x1
#uimm5 when pattern = 1x010
#uimm5 when pattern = 1xx00
MUL4 when pattern = 11101
MUL3 when pattern = 11110
ALL when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
integer element1 = Int(Elem[operand1, e, esize], unsigned);
(Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);
Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.510   SQDMLALB (indexed)

Multiply then double the even-numbered signed elements within each 128-bit segment of the first source vector and
specified signed element in the corresponding second source vector segment. Each intermediate value is saturated
to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Then destructively add to the overlapping
double-width elements of the addend and destination vector. Each destination element is saturated to the
double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

The elements within the second source vector are specified using an immediate index which selects the same
element position within each 128-bit vector segment. The index range is from 0 to one less than the number of
elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

32-bit

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 0 0 0 1 0 0 | 1 0 1 | i3h | Zm | 0 0 1 | 0 | i3l | Zn | Zda | S | T
```

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;

64-bit

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 |16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 0 0 0 1 0 0 | 1 1 | Zm | 0 0 1 | 0 | i2i | Zn | Zda | S | T
```

Encoding

SQDMLALB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2i:i2i);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
Assembler symbols

<Zda>  Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn>   Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>   For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm>  For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
  integer s = e - (e MOD eltspersegment);
  integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
  integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
  integer element3 = SInt(Elem[result, e, 2*esize]);
  integer product = SInt(SignedSat(2 * element1 * element2, 2*esize));
  integer res = element3 + product;
  Elem[result, e, 2*esize] = SignedSat(res, 2*esize);
Z[da, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.511  SQDMLALB (vectors)

Multiply then double the corresponding even-numbered signed elements of the first and second source vectors. Each intermediate value is saturated to the double-width N-bit value's signed integer range -2^{(N-1)} to (2^{(N-1)})-1. Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range -2^{(N-1)} to (2^{(N-1)})-1. This instruction is unpredicated.

Encoding

SQDMLALB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 0;
integer sel2 = 0;

Assembler symbols

<Zda>  Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<T>   Is the size specifier, encoded in the "size" field. It can have the following values:
    H  when size = 01
    S  when size = 10
    D  when size = 11
    The encoding size = 00 is reserved.
<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb>   Is the size specifier, encoded in the "size" field. It can have the following values:
    B  when size = 01
    H  when size = 10
    S  when size = 11
    The encoding size = 00 is reserved.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
  integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
  integer element3 = SInt(Elem[result, e, esize]);
  integer product = SInt(SignedSat(2 * element1 * element2, esize));
  Elem[result, e, esize] = SignedSat(element3 + product, esize);

Z[da, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.512   SQDMLALBT

Multiply then double the corresponding even-numbered signed elements of the first and odd-numbered signed
elements of the second source vector. Each intermediate value is saturated to the double-width N-bit value's signed
integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Then destructively add to the overlapping double-width elements of the addend
and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer
range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

Encoding

SQDMLALBT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 0;
integer sel2 = 1;

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
  integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
  integer element3 = SInt(Elem[result, e, esize]);
  integer product = SInt(SignedSat(2 * element1 * element2, esize));
  Elem[result, e, esize] = SignedSat(element3 + product, esize);

Z[da, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.513   SQDMLALT (indexed)

Multiply then double the odd-numbered signed elements within each 128-bit segment of the first source vector and the specified signed element in the corresponding second source vector segment. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

32-bit

```
[31 30 29 28|27 26 25 24|23 22 21 20]|19 18 16|15 14 13|12|11 10 9 | 5 4 | 0 |
0 1 0 0 0 1 0 0 1 0 1 |i3h| Zm | 0 0 1 | 0 |i3l| Zn |
```

size<1>
size<0>

Encoding


Decode for this encoding

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

64-bit

```
[31 30 29 28|27 26 25 24|23 22 21 20]|19 18 16|15 14 13|12|11 10 9 | 5 4 | 0 |
0 1 0 0 0 1 0 0 1 1 |i2h| Zm | 0 0 1 | 0 |i2l| Zn |
```

size<1>
size<0>
i2h

Encoding


Decode for this encoding

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```
Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
  integer s = e - (e MOD eltspersegment);
  integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
  integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
  integer element3 = SInt(Elem[result, e, 2*esize]);
  integer product = SInt(SignedSat(2 * element1 * element2, 2*esize));
  integer res = element3 + product;
  Elem[result, e, 2*esize] = SignedSat(res, 2*esize);
Z[da, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.514   SQDMLALT (vectors)

Multiply then double the corresponding odd-numbered signed elements of the first and second source vectors. Each intermediate value is saturated to the double-width N-bit value's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

Encoding

SQDMLALT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
in integer n = UInt(Zn);
in integer m = UInt(Zm);
in integer da = UInt(Zda);
in integer sel1 = 1;
in integer sel2 = 1;

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
  integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
  integer element3 = SInt(Elem[result, e, esize]);
  integer product = SInt(SignedSat(2 * element1 * element2, esize));
  Elem[result, e, esize] = SignedSat(element3 + product, esize);

Z[da, VL] = result;

---

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.515  SQDMLSLB (indexed)

Multiply then double the even-numbered signed elements within each 128-bit segment of the first source vector and the specified signed element in the corresponding second source vector segment. Each intermediate value is saturated to the double-width N-bit value's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\).

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

**32-bit**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
 0 1 0 0 0 1 0 0 | 1 0 1 | i3h  Zm 0 0 1 | 1 | i3l 0 | Zn | Zda
```

**Encoding**


**Decode for this encoding**

```cpp
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

**64-bit**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
 0 1 0 0 0 1 0 0 | 1 1 1 | Zm 0 0 1 | 1 | i2l 0 | Zn | Zda
```

**Encoding**

SQDMLSLB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

**Decode for this encoding**

```cpp
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```
Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation for all encodings

    CheckSVEEnabled();
    constant integer VL = CurrentVL;
    constant integer PL = VL DIV 8;
    constant integer elements = VL DIV (2 * esize);
    constant integer eltspersegment = 128 DIV (2 * esize);
    bits(VL) operand1 = Z[n, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) result = Z[da, VL];

    for e = 0 to elements-1
        integer s = e - (e MOD eltspersegment);
        integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
        integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
        integer element3 = SInt(Elem[result, e, 2*esize]);
        integer product = SInt(SignedSat(2 * element1 * element2, 2*esize));
        integer res = element3 - product;
        Elem[result, e, 2*esize] = SignedSat(res, 2*esize);
    Z[da, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.516  SQDMLSLB (vectors)

Multiply then double the corresponding even-numbered signed elements of the first and second source vectors. Each intermediate value is saturated to the double-width N-bit value's signed integer range \(-2^{(N-1)}\) to \(2^{(N-1)}-1\). Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range \(-2^{(N-1)}\) to \(2^{(N-1)}-1\). This instruction is unpredicated.

**Encoding**

```
SQDMLSLB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

**Decode for this encoding**

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 0;
integer sel2 = 0;
```

**Assembler symbols**

- `<Zda>` is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- `<T>` is the size specifier, encoded in the "size" field. It can have the following values:
  - `H` when size = 01
  - `S` when size = 10
  - `D` when size = 11
  
  The encoding size = 00 is reserved.
- `<Zn>` is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>` is the size specifier, encoded in the "size" field. It can have the following values:
  - `B` when size = 01
  - `H` when size = 10
  - `S` when size = 11
  
  The encoding size = 00 is reserved.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
```
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 - product, esize);

Z[da, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.517  SQDMLSLBT

Multiply then double the corresponding even-numbered signed elements of the first and odd-numbered signed elements of the second source vector. Each intermediate value is saturated to the double-width N-bit value's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

Encoding

SQDMLSLBT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 0;
integer sel2 = 1;

Assembler symbols

<Zda>  Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<T>  Is the size specifier, encoded in the "size" field. It can have the following values:
H  when size = 01
S  when size = 10
D  when size = 11
The encoding size = 00 is reserved.

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb>  Is the size specifier, encoded in the "size" field. It can have the following values:
B  when size = 01
H  when size = 10
S  when size = 11
The encoding size = 00 is reserved.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 - product, esize);

Z[da, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.518  SQDMLSLT (indexed)

Multiply then double the odd-numbered signed elements within each 128-bit segment of the first source vector
and the specified signed element in the corresponding second source vector segment. Each intermediate value is
saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)}-1)$. Then destructively subtract
from the overlapping double-width elements of the addend and destination vector. Each destination element is
saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)}-1)$.

The elements within the second source vector are specified using an immediate index which selects the same
element position within each 128-bit vector segment. The index range is from 0 to one less than the number of
elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

**32-bit**

```plaintext
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**Encoding**


**Decode for this encoding**

```plaintext
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

**64-bit**

```plaintext
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

** Encoding**


**Decode for this encoding**

```plaintext
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```
Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
integer s = e - (e MOD eltspersegment);
integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
integer element3 = SInt(Elem[result, e, 2*esize]);
integer product = SInt(SignedSat(2 * element1 * element2, 2*esize));
integer res = element3 - product;
Elem[result, e, 2*esize] = SignedSat(res, 2*esize);
Z[da, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.519 SQDMLSLT (vectors)

Multiply then double the corresponding odd-numbered signed elements of the first and second source vectors. Each intermediate value is saturated to the double-width N-bit value's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

Encoding

SQDMLSLT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

\[
\begin{align*}
\text{if } & \text{ !HaveSVE2()} \&\& \text{ !HaveSME()} \text{ then UNDEFINED;} \\
\text{if } & \text{ size } = \text{ '00'} \text{ then UNDEFINED;} \\
\text{constant integer esize } = & 8 << \text{ UInt(size)}; \\
\text{integer n } = \text{ UInt(Zn)}; \\
\text{integer m } = \text{ UInt(Zm)}; \\
\text{integer da } = \text{ UInt(Zda)}; \\
\text{integer sel1 } = 1; \\
\text{integer sel2 } = 1;
\end{align*}
\]

Assembler symbols

\(<Zda>\) Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

\(<T>\) Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

\(<Zn>\) Is the size specifier, encoded in the "size" field. It can have the following values:

| \begin{align*}
H & \text{ when size } = \text{ 01} \\
S & \text{ when size } = \text{ 10} \\
D & \text{ when size } = \text{ 11}
\end{align*} |

The encoding size = 00 is reserved.

\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

\(<Tb>\) Is the size specifier, encoded in the "size" field. It can have the following values:

| \begin{align*}
B & \text{ when size } = \text{ 01} \\
H & \text{ when size } = \text{ 10} \\
S & \text{ when size } = \text{ 11}
\end{align*} |

The encoding size = 00 is reserved.

\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 - product, esize);

Z[da, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.520  **SQDMULH (indexed)**

Multiply all signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, double and place the most significant half of the result in the corresponding elements of the destination vector register. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)} - 1)$.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element.

### 16-bit

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>1</td>
<td>i3l</td>
<td>Zm</td>
<td>1 1 1 1 0</td>
<td>0</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>
```

**Encoding**


**Decode for this encoding**

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

### 32-bit

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>1</td>
<td>1</td>
<td>Zm</td>
<td>1 1 1 1 0</td>
<td>0</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>
```

**Encoding**

SQDMULH <Zd>.S, <Zn>.S, <Zm>.S[<imm>]

**Decode for this encoding**

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```
64-bit

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;

constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm> For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.
For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, s, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat(res >> esize, esize);
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.521   SQDMULH (vectors)

Multiply then double the corresponding signed elements of the first and second source vectors, and place the most
significant half of the results in the corresponding elements of the destination vector register. Each result element is
saturated to the N-bit element's signed integer range \(-2^{(N-1)} - 1\). This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>size 1</td>
</tr>
<tr>
<td>Zm</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>

Encoding

\[
\text{SQDMULH} <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
\]

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

\(<Zd>\) Is the name of the destination scalable vector register, encoded in the "Zd" field.

\(<T>\) Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

\(<Zn>\) Is the name of the first source scalable vector register, encoded in the "Zn" field.

\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, e, esize]);
  integer element2 = SInt(Elem[operand2, e, esize]);
  integer res = 2 * element1 * element2;
  Elem[result, e, esize] = SignedSat(res >> esize, esize);
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.522  SQDMULLB (indexed)

Multiply then double the even-numbered signed elements within each 128-bit segment of the first source vector and the specified element in the corresponding second source vector segment, and place the results in overlapping double-width elements of the destination vector register. Each result element is saturated to the double-width N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\).

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

32-bit

|   | 32 | 31 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  |   | 5 | 4 | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | i3h| Zm | 1  | 1  | 1  | 0  | i3l| Zn | Zd |
|size<1> |    |     |     |     |     |     |     | T |
|size<0> |    |     |     |     |     |     |     |   |

Encoding

Decode for this encoding
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;

64-bit

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  |   | 5 | 4 | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | i2h| Zm | 1  | 1  | 1  | 0  | i2l| Zn | Zd |
|size<1> |    |     |     |     |     |     |     |     |     |     |     | i2h| T |
|size<0> |    |     |     |     |     |     |     |     |     |     |     |     |   |

Encoding
SQDMULLB <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

Decode for this encoding
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;
Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>  For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
       For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
       For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation for all encodings

    CheckSVEEnabled();
    constant integer VL = CurrentVL;
    constant integer PL = VL DIV 8;
    constant integer elements = VL DIV (2 * esize);
    constant integer eltspersegment = 128 DIV (2 * esize);
    bits(VL) operand1 = Z[n, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) result;
    for e = 0 to elements-1
        integer s = e - (e MOD eltspersegment);
        integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
        integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
        integer res = 2 * element1 * element2;
        Elem[result, e, 2*esize] = SignedSat(res, 2*esize);
    Z[d, VL] = result;
C8.2.523   SQDMULLB (vectors)

Multiply the corresponding even-numbered signed elements of the first and second source vectors, double and place the results in the overlapping double-width elements of the destination vector. Each result element is saturated to the double-width N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

Encoding

\[
\text{SQDMULLB } <Zd>._<T>, <Zn>._<Tb>, <Zm>._<Tb>
\]

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd>   Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>    Is the size specifier, encoded in the "size" field. It can have the following values:
    H when size = 01
    S when size = 10
    D when size = 11
    The encoding size = 00 is reserved.
<Zn>   Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb>   Is the size specifier, encoded in the "size" field. It can have the following values:
    B when size = 01
    H when size = 10
    S when size = 11
    The encoding size = 00 is reserved.
<Zm>   Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
integer res = 2 * element1 * element2;
Elem[result, e, esize] = SignedSat(res, esize);

Z[d, VL] = result;
C8.2.524   SQDMULLT (indexed)

Multiply then double the odd-numbered signed elements within each 128-bit segment of the first source vector and
the specified element in the corresponding second source vector segment, and place the results in overlapping
double-width elements of the destination vector register. Each result element is saturated to the double-width \( N \)-bit
element's signed integer range \([-2^{(N-1)} \text{ to } (2^{(N-1)})-1]\).

The elements within the second source vector are specified using an immediate index which selects the same
element position within each 128-bit vector segment. The index range is from 0 to one less than the number of
elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

### 32-bit

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 0 0 0 1 0 0 1 0 1 i3h Zm 1 1 1 0 i3l Zn Zd
```

```
size<1>
size<0>
```

**Encoding**


**Decode for this encoding**

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;
```

### 64-bit

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 0 0 0 1 0 0 1 1 Zm 1 1 1 0 |2| 1 Zn Zd
```

```
size<1>
size<0>
i2h
```

**Encoding**

SQDMULLT <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

**Decode for this encoding**

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;
```
Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
  integer s = e - (e MOD eltspersegment);
  integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
  integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
  integer res = 2 * element1 * element2;
  Elem[result, e, 2*esize] = SignedSat(res, 2*esize);
Z[d, VL] = result;
C8.2.525   SQDMULLT (vectors)

Multiply the corresponding odd-numbered signed elements of the first and second source vectors, double and place
the results in the overlapping double-width elements of the destination vector. Each result element is saturated to
the double-width N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1</td>
<td>size 0</td>
<td>Zm 0 1 1 0 0 1</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

SQDMULLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

**Decode for this encoding**

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

**Assembler symbols**

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>`: Is the size specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11
  - The encoding size = 00 is reserved.
- `<Zn>`: Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>`: Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 01
  - H when size = 10
  - S when size = 11
  - The encoding size = 00 is reserved.
- `<Zm>`: Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bv(VL) operand1 = Z[n, VL];
bv(VL) operand2 = Z[m, VL];
bv(VL) result;
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
```
integer res = 2 * element1 * element2;
Elem[result, e, esize] = SignedSat(res, esize);

Z[d, VL] = result;
C8.2.526   SQINCB

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register’s signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

32-bit

Encoding

SQINCB <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;

64-bit

Encoding

SQINCB <Xdn>{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

Assembler symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

- **POW2** when pattern = 00000
- **VL1** when pattern = 00001
- **VL2** when pattern = 00010
- **VL3** when pattern = 00011
- **VL4** when pattern = 00100
- **VL5** when pattern = 00101
- **VL6** when pattern = 00110
- **VL7** when pattern = 00111
- **VL8** when pattern = 01000
- **VL16** when pattern = 01001
- **VL32** when pattern = 01010
- **VL64** when pattern = 01011
- **VL128** when pattern = 01100
- **VL256** when pattern = 01101
- **#uimm5** when pattern = 0111x
- **#uimm5** when pattern = 101x1
- **#uimm5** when pattern = 10110
- **#uimm5** when pattern = 1x0x1
- **#uimm5** when pattern = 1x010
- **#uimm5** when pattern = 1xx00
- **MUL4** when pattern = 11101
- **MUL3** when pattern = 11110
- **ALL** when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation for all encodings

CheckSVEEnabled();
icenter count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

center integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count + imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
C8.2.527   SQINCD (scalar)

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register’s signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL)

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

32-bit

SQINCD <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
binary(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;

64-bit

SQINCD <Xdn>{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
binary(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

Assembler symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

- Pow2 when pattern = 00000
- VL1 when pattern = 00001
- VL2 when pattern = 00010
- VL3 when pattern = 00011
- VL4 when pattern = 00100
- VL5 when pattern = 00101
- VL6 when pattern = 00110
- VL7 when pattern = 00111
- VL8 when pattern = 01000
- VL16 when pattern = 01001
- VL32 when pattern = 01010
- VL64 when pattern = 01011
- VL128 when pattern = 01100
- VL256 when pattern = 01101
- #uimm5 when pattern = 0111x
- #uimm5 when pattern = 101x1
- #uimm5 when pattern = 10110
- #uimm5 when pattern = 1x0x1
- #uimm5 when pattern = 1x010
- #uimm5 when pattern = 1xx00
- MUL4 when pattern = 11101
- MUL3 when pattern = 11110
- ALL when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation for all encodings

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;
integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
C8.2.528  SQINCD (vector)

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 64-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

Encoding

SQINCD <Zdn>.D{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;

Assembler symbols

<Zdn>  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<pattern>  Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Encoded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>POW2</td>
<td>00000</td>
</tr>
<tr>
<td>VL1</td>
<td>00001</td>
</tr>
<tr>
<td>VL2</td>
<td>00010</td>
</tr>
<tr>
<td>VL3</td>
<td>00011</td>
</tr>
<tr>
<td>VL4</td>
<td>00100</td>
</tr>
<tr>
<td>VL5</td>
<td>00101</td>
</tr>
<tr>
<td>VL6</td>
<td>00110</td>
</tr>
<tr>
<td>VL7</td>
<td>00111</td>
</tr>
<tr>
<td>VL8</td>
<td>01000</td>
</tr>
<tr>
<td>VL16</td>
<td>01001</td>
</tr>
</tbody>
</table>
VL32 when pattern = 01010
VL64 when pattern = 01011
VL128 when pattern = 01100
VL256 when pattern = 01101
#uimm5 when pattern = 01110
#uimm5 when pattern = 101x1
#uimm5 when pattern = 10110
#uimm5 when pattern = 1x0x1
#uimm5 when pattern = 1x010
#uimm5 when pattern = 1xx00
MUL4 when pattern = 11101
MUL3 when pattern = 11110
ALL when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element1 + (count + imm), esize, unsigned);
Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.529   SQINCH (scalar)

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an
immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is
saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then
sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than
Undefined Instruction exception.

32-bit

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
0 0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 | 1 1 1 1 | 0 0 | 0 | 0 | Rdn
```

size<1>

size<0>

Encoding

SQINCH <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;

64-bit

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
0 0 0 0 0 0 0 1 0 0 0 1 1 1 1 | 0 0 | 1 1 1 1 | 0 0 | 0 | 0 | Rdn
```

size<1>

size<0>

Encoding

SQINCH <Xdn>{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

Assembler symbols

<Xd> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Wd> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

- POW2 when pattern = 00000
- VL1 when pattern = 00001
- VL2 when pattern = 00010
- VL3 when pattern = 00011
- VL4 when pattern = 00100
- VL5 when pattern = 00101
- VL6 when pattern = 00110
- VL7 when pattern = 00111
- VL8 when pattern = 01000
- VL16 when pattern = 01001
- VL32 when pattern = 01010
- VL64 when pattern = 01011
- VL128 when pattern = 01100
- VL256 when pattern = 01101
- #uimm5 when pattern = 0111x
- #uimm5 when pattern = 101x1
- #uimm5 when pattern = 10110
- #uimm5 when pattern = 1x0x1
- #uimm5 when pattern = 1x010
- #uimm5 when pattern = 1xx00
- MUL4 when pattern = 11101
- MUL3 when pattern = 11110
- ALL when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation for all encodings

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
C8.2.530   SQINCH (vector)

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 16-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

```
| 31 30 29 28| 27 26 25 24| 23 22 21 20| 19 16| 15 14 13 12| 11 10 9 | 5 4 | 0 |
| 0 0 0 0 0 1 0 0 | 0 1 1 0 | imm4 | 1 1 0 0 | 0 0 | pattern | Zdn |

size<1> size<0>
```

**Encoding**

SQINCH <Zdn>.H{, <pattern>{, MUL #<imm>}}

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bias(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;

**Assembler symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<pattern>` Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:
  - POW2 when pattern = 00000
  - VL1 when pattern = 00001
  - VL2 when pattern = 00010
  - VL3 when pattern = 00011
  - VL4 when pattern = 00100
  - VL5 when pattern = 00101
  - VL6 when pattern = 00110
  - VL7 when pattern = 00111
  - VL8 when pattern = 01000
  - VL16 when pattern = 01001
C8.2 Alphabetical list of SVE instructions

VL32 when pattern = 01010
VL64 when pattern = 01011
VL128 when pattern = 01100
VL256 when pattern = 01101
#uimm5 when pattern = 0111x
#uimm5 when pattern = 101x1
#uimm5 when pattern = 10110
#uimm5 when pattern = 1x0x1
#uimm5 when pattern = 1x010
#uimm5 when pattern = 1xx00
MUL4 when pattern = 11101
MUL3 when pattern = 11110
ALL when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.531   SQINCP (scalar)

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

32-bit

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8 | 5 4 | 0 ]
0 0 1 0 0 1 0 1 | 1 0 1 0 0 1 0 0 | 1 0 0 0 1 0 1 0 | Pm | Rdn
```

**Encoding**

```
SQINCP <Xdn>, <Pm>.<T>, <Wdn>
```

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
integer ssize = 32;
```

64-bit

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8 | 5 4 | 0 ]
0 0 1 0 0 1 0 1 | 1 0 1 0 0 1 0 0 | 1 0 0 0 1 0 1 0 | Pm | Rdn
```

**Encoding**

```
SQINCP <Xdn>, <Pm>.<T>
```

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
integer ssize = 64;
```

**Assembler symbols**

- `<Xdn>`: Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- `<Pm>`: Is the name of the source scalable predicate register, encoded in the "Pm" field.
- `<T>`: Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

**Operation for all encodings**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn, ssize];
bits(PL) operand2 = P[m, PL];
bits(ssize) result;
integer count = 0;

for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;

integer element = Int(operand1, unsigned);
(result, -) = SatQ(element + count, ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

**Operational information**

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.
C8.2.532   SQINCP (vector)

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements. The results are saturated to the element signed integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

Encoding

SQINCP <Zdn>.<T>, <Pm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = $ << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;

Assembler symbols

<Zdn>   Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T>     Is the size specifier, encoded in the "size" field. It can have the following values:

H        when size = '01
S        when size = '10
D        when size = '11

The encoding size = '00 is reserved.

<Pm>    Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
    if Elem[operand2, e, esize] == '1' then
        count = count + 1;
for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element + count, esize, unsigned);
Z[dn, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.533   SQINCW (scalar)

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

### 32-bit

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 1 0 1 0</td>
<td>imm4 1 1 1 1 0 0</td>
<td>pattern</td>
<td>Rdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

SQINCW <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

### 64-bit

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>1 0 1 1</td>
<td>imm4 1 1 1 1 0 0</td>
<td>pattern</td>
<td>Rdn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

SQINCW <Xdn>{, <pattern>{, MUL #<imm>}}

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
```
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

Assembler symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

- **POW2** when pattern = 00000
- **VL1** when pattern = 00001
- **VL2** when pattern = 00010
- **VL3** when pattern = 00011
- **VL4** when pattern = 00100
- **VL5** when pattern = 00101
- **VL6** when pattern = 00110
- **VL7** when pattern = 00111
- **VL8** when pattern = 01000
- **VL16** when pattern = 01001
- **VL32** when pattern = 01010
- **VL64** when pattern = 01011
- **VL128** when pattern = 01100
- **VL256** when pattern = 01101
- **#uimm5** when pattern = 0111x
- **#uimm5** when pattern = 101x1
- **#uimm5** when pattern = 10110
- **#uimm5** when pattern = 10x01
- **#uimm5** when pattern = 1x010
- **#uimm5** when pattern = 1xx00
- **MUL4** when pattern = 11101
- **MUL3** when pattern = 11110
- **ALL** when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation for all encodings

    CheckSVEEnabled();
    integer count = DecodePredCount(pat, esize);
    bits(ssize) operand1 = X[dn, ssize];
    bits(ssize) result;
    integer element1 = Int(operand1, unsigned);
    (result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
    X[dn, 64] = Extend(result, 64, unsigned);
C8.2.534   SQINCW (vector)

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an
immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The
results are saturated to the 32-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than
Undefined Instruction exception.

![Encoding Diagram]

**Encoding**

`SQINCW <Zdn>.S{, <pattern>{, MUL #<imm>}}`

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

**Assembler symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<pattern>` Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the
  following values:
  - `POW2` when pattern = 00000
  - `VL1` when pattern = 00001
  - `VL2` when pattern = 00010
  - `VL3` when pattern = 00011
  - `VL4` when pattern = 00100
  - `VL5` when pattern = 00101
  - `VL6` when pattern = 00110
  - `VL7` when pattern = 00111
  - `VL8` when pattern = 01000
  - `VL16` when pattern = 01001
  - `ALL`
VL32 when pattern = 01010
VL64 when pattern = 01011
VL128 when pattern = 01100
VL256 when pattern = 01101
#uimm5 when pattern = 0111x
#uimm5 when pattern = 101x1
#uimm5 when pattern = 10110
#uimm5 when pattern = 1x010
#uimm5 when pattern = 1xx00
MUL4 when pattern = 11101
MUL3 when pattern = 11110
ALL when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);
Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.535 SQNEG

Negate the signed integer value in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element’s signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Inactive elements in the destination vector register remain unmodified.

Encoding

SQNEG <Zd>.<T>, <Pg>/M, <Zn>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
    B when size = 00
    H when size = 01
    S when size = 10
    D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer element = SInt(Elem[operand, e, esize]);
        element = -element;
        Elem[result, e, esize] = SignedSat(element, esize);
Z[d, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.536  SQRDCMLAH (indexed)

Multiply without saturation the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

Then double and add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the most significant rounded half of the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

16-bit

```
0 1 0 0 0 1 0 0 1 0 1 1 1 0 1 1
size<1> size<0> Zm 0 1 1 1 rot Zn Zda
```

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');

32-bit

```
0 1 0 0 0 1 0 0 1 1 1 1 0 1 1 1
size<1> size<0> Zm 0 1 1 1 rot Zn Zda
```

Encoding

SQRDCMLAH <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>
Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 16-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 32-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 16-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.
For the 32-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

<const> Is the const specifier, encoded in the "rot" field. It can have the following values:
#0 when rot = 00
#90 when rot = 01
#180 when rot = 10
#270 when rot = 11

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
constant integer pairspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

integer round_const = 1 << (esize-1);
integer res_r, res_i;
for p = 0 to pairs-1
  integer segmentbase = p - (p MOD pairspersegment);
  integer s = segmentbase + index;
  integer elt1_a = SInt(Elem[operand1, 2 * p + sel_a, esize]);
  integer elt2_a = SInt(Elem[operand2, 2 * s + sel_a, esize]);
  integer elt2_b = SInt(Elem[operand2, 2 * s + sel_b, esize]);
  bits(esize) elt3_r = Elem[operand3, 2 * p + 0, esize];
  bits(esize) elt3_i = Elem[operand3, 2 * p + 1, esize];
  integer product_r_r = elt1_a * elt2_a;
  integer product_r_i = elt1_a * elt2_b;
  if sub_r then
    res_r = (SInt(elt3_r) << esize) - 2 * product_r_r + round_const;
  else
    res_r = (SInt(elt3_r) << esize) + 2 * product_r_r + round_const;
  bits(VL) res_r = res_r;
  bits(VL) res_i = res_i;
else
if sub_i then
    res_i = (SInt(elt3_i) << esize) - 2 * product_i + round_const;
else
    res_i = (SInt(elt3_i) << esize) + 2 * product_i + round_const;
Elem[result, 2 * p + 0, esize] = SignedSat(res_r >> esize, esize);
Elem[result, 2 * p + 1, esize] = SignedSat(res_i >> esize, esize);

Z[da, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.537 SQRDCMLAH (vectors)

Multiply without saturation the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in the first source vector by the corresponding complex number in the second source vector rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

Then double and add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the most significant rounded half of the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

Encoding

SQRDCMLAH <Zda>.<T>, <Zn>.<T>, <Zm>.<T>, <const>

Decode for this encoding

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');
```

Assembler symbols

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<const>` Is the const specifier, encoded in the "rot" field. It can have the following values:
  - #0 when rot = 00
  - #90 when rot = 01
#180 when rot = 10
#270 when rot = 11

## Operation

```plaintext
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

integer round_const = 1 << (esize-1);
integer res_r, res_i;
for p = 0 to pairs-1
  integer elt1_a = SInt(Elem[operand1, 2 * p + sel_a, esize]);
  integer elt2_a = SInt(Elem[operand2, 2 * p + sel_a, esize]);
  integer elt2_b = SInt(Elem[operand2, 2 * p + sel_b, esize]);
  bits(esize) elt3_r = Elem[operand3, 2 * p + 0, esize];
  bits(esize) elt3_i = Elem[operand3, 2 * p + 1, esize];

  integer product_r = elt1_a * elt2_a;
  integer product_i = elt1_a * elt2_b;
  if sub_r then
    res_r = (SInt(elt3_r) << esize) - 2 * product_r + round_const;
  else
    res_r = (SInt(elt3_r) << esize) + 2 * product_r + round_const;
  if sub_i then
    res_i = (SInt(elt3_i) << esize) - 2 * product_i + round_const;
  else
    res_i = (SInt(elt3_i) << esize) + 2 * product_i + round_const;

  Elem[result, 2 * p + 0, esize] = SignedSat(res_r >> esize, esize);
  Elem[result, 2 * p + 1, esize] = SignedSat(res_i >> esize, esize);

Z[da, VL] = result;
```

## Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.538  SQRDMLAH (indexed)

Multiply then double all signed elements within each 128-bit segment of the first source vector and the specified signed element of the corresponding second source vector segment, and destructively add the rounded high half of each result to the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\).

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element.

### 16-bit

<table>
<thead>
<tr>
<th>S</th>
<th>i3h</th>
<th>i3l</th>
<th>Zm</th>
<th>Zn</th>
<th>Zda</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Encoding**


**Decode for this encoding**

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### 32-bit

<table>
<thead>
<tr>
<th>S</th>
<th>size&lt;1&gt;</th>
<th>size&lt;0&gt;</th>
<th>i2</th>
<th>Zm</th>
<th>Zn</th>
<th>Zda</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Encoding**

SQRDMLAH <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

**Decode for this encoding**

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```
SVE Instruction Descriptions

C8.2 Alphabetical list of SVE instructions

64-bit

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9  |  5 4  |  0 ]
0 1 0 0 0 1 0 0 1 1 1 1 Zm 0 0 0 1 0 0 Zn  Zda
```

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm> For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.
For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
constant integer round_const = 1 << (esize - 1);
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, s, esize]);
    integer element3 = SInt(Elem[operand3, e, esize]);
    integer res = (element3 << esize) + (2 * element1 * element2);
    Elem[result, e, esize] = SignedSat((res + round_const) >> esize, esize);
Z[da, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.539 SQRDM LAH (vectors)

Multiply then double the corresponding signed elements of the first and second source vectors, and destructively add the rounded high half of each result to the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element’s signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

Encoding

SQRDMLAH <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

Decoding for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = $8 <<$ UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler symbols

<Zda>  Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T>    Is the size specifier, encoded in the "size" field. It can have the following values:

B       when size = 00
H       when size = 01
S       when size = 10
D       when size = 11

<Zn>   Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>   Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
constant integer round_const = 1 << (esize - 1);
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
exteger element2 = SInt(Elem[operand2, e, esize]);
exteger element3 = SInt(Elem[operand3, e, esize]);
exteger res = (element3 << esize) + (2 * element1 * element2);
  Elem[result, e, esize] = SignedSat((res + round_const) >> esize, esize);
Z[da, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.540  SQRDMLSH (indexed)

Multiply then double all signed elements within each 128-bit segment of the first source vector and the specified
signed element of the corresponding second source vector segment, and destructively subtract the rounded high half
of each result to the corresponding elements of the addend and destination vector. Each destination element is
saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)}-1)\).

The elements within the second source vector are specified using an immediate index which selects the same
element position within each 128-bit vector segment. The index range is from 0 to one less than the number of
elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element.

16-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 14 13 12 11 10 9</th>
<th>5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 0 1 0 0 1 31</td>
<td>Zm 0 0 0 1 0 1 Zn Zda</td>
</tr>
</tbody>
</table>

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 14 13 12 11 10 9</th>
<th>5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 0 1 1 2 Zm 0 0 0 1 0 1 Zn Zda</td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SQRDMLSH <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
64-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9</th>
<th>5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 1 1</td>
<td>Zm 0 0 0 1 0 1</td>
</tr>
</tbody>
</table>

size<1>
size<0>

Encoding

\( \text{SQRDMLSH } <\text{Zda}>.D, <\text{Zn}>.D, <\text{Zm}>.D[<\text{imm}>] \)

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.
For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
constant integer round_const = 1 << (esize - 1);
for e = 0 to elements-1
  integer segmentbase = e - (e MOD eltspersegment);
  integer s = segmentbase + index;
  integer element1 = SInt(Elem[operand1, e, esize]);
  integer element2 = SInt(Elem[operand2, s, esize]);
  integer element3 = SInt(Elem[operand3, e, esize]);
  integer res = (element3 << esize) - (2 * element1 * element2);
  Elem[result, e, esize] = SignedSat((res + round_const) >> esize, esize);
Z[da, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.541  SQRDMLSH (vectors)

Multiply then double the corresponding signed elements of the first and second source vectors, and destructively subtract the rounded high half of each result from the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

```
[31 30 29 28 | 27 26 25 24 | 23 22 21 20]   [16 | 15 14 13 12 | 11 10 9 | 5 4 | 0 ]
0 1 0 0 0 1 0 0 | size 0 | Zm 0 1 1 0 1 | Zn | Zda
```

**Encoding**

SQRDMLSH <Zda><T>, <Zn><T>, <Zm><T>

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

**Assembler symbols**

<Zda>  Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

B  when size = 00
H  when size = 01
S  when size = 10
D  when size = 11

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
constant integer round_const = 1 << (esize - 1);
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, e, esize]);
  integer element2 = SInt(Elem[operand2, e, esize]);
  integer element3 = SInt(Elem[operand3, e, esize]);
  integer res = (element3 << esize) - (2 * element1 * element2);
  Elem[result, e, esize] = SignedSat((res + round_const) >> esize, esize);
Z[da, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.542 SQRDMULH (indexed)

Multiply all signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, double and place the most significant rounded half of the result in the corresponding elements of the destination vector register. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \(2^{(N-1)}-1\).

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element.

16-bit

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
  0 1 0 0 0 1 0 0 1 1 i3l Zm 1 1 1 0 1 Zn Zd
```

Encoding


Decode for this encoding

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

32-bit

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
  0 1 0 0 0 1 0 0 1 1 i2 Zm 1 1 1 0 1 Zn Zd
```

Encoding

SQRDMULH <Zd>.S, <Zn>.S, <Zm>.S[<imm>]

Decode for this encoding

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```
64-bit

| [31 30 29 28|27 26 25 24|23 22 21 20]|19 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|----------------|----------------|----------------|----------------|
| 0 1 0 0 0 1 0 1 1 1 1 1 0 | Zm | 1 1 1 1 0 | 1 | Zn | Zd |

**Encoding**


**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;

constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

**Operation for all encodings**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
constant integer round_const = 1 << (esize - 1);
for e = 0 to elements-1
integer segmentbase = e - (e MOD eltspersegment);
integer s = segmentbase + index;
integer element1 = SInt(Elem[operand1, e, esize]);
integer element2 = SInt(Elem[operand2, s, esize]);
integer res = 2 * element1 * element2;
Elem[result, e, esize] = SignedSat((res + round_const) >> esize, esize);
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.543 SQRDMULH (vectors)

Multiply then double the corresponding signed elements of the first and second source vectors, and place the most significant rounded half of the result in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

```
| 31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13 12|11 10 9 | 5 4 | 0 |
|-------------|-------------|-------------|-----|-------------|-------------|-----|
| 0 0 0 0 0 1 0 0| size 1 | Zm 0 1 1 0 1 | Zn | Zd |
```

**Encoding**

SQRDMULH <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

**Decode for this encoding**

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```c
const uint32_t vl = CurrentVL;
const uint32_t elements = vl / esize;
const uint32_t operand1 = Z[n, vl];
const uint32_t operand2 = Z[m, vl];
const uint32_t result;
const uint32_t round_const = 1 << (esize - 1);
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat((res + round_const) >> esize, esize);
Z[d, vl] = result;
```
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.544  SQRSHL

Shift active signed elements of the first source vector by corresponding elements of the second source vector and
destructively place the rounded results in the corresponding elements of the first source vector. A positive shift
amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element
is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \(2^{(N-1)}-1\). Inactive elements in the destination
vector register remain unmodified.

Encoding

SQRSHL <Zdn>..<T>, <Pg>/M, <Zdn>..<T>, <Zm>..<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>   Is the size specifier, encoded in the "size" field. It can have the following values:
   B when size = 00
   H when size = 01
   S when size = 10
   D when size = 11
<Pg>   Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>   Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
   if Elem[mask, e, esize] == '1' then
      integer element = SInt(Elem[operand1, e, esize]);
      integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
      integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
      integer res = (element + round_const) << shift;
      Elem[result, e, esize] = SignedSat(res, esize);
   else

[31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 10 9 | 5 4 | 0 |]
| 0 1 0 0 0 1 0 0 | 0 0 1 0 1 0 1 0 | Pg | Zm | Zdn |
\begin{verbatim}
Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
\end{verbatim}

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.545 SQRSHLR

Shift active signed elements of the second source vector by corresponding elements of the first source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \(2^{(N-1)}\)-1. Inactive elements in the destination vector register remain unmodified.

Encoding

<table>
<thead>
<tr>
<th>Zdn</th>
<th>T</th>
<th>Pg/M</th>
<th>Zdn</th>
<th>T</th>
<th>Zm</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;

constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    integer res = (element + round_const) << shift;
    Elem[result, e, esize] = SignedSat(res, esize);
  else
Elem[result, e, esize] = Elem[operand2, e, esize];

Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.546   SQRSHRN

Shift each signed integer value in the source vector elements right by an immediate value, and place the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Encoding

SQRSHRN <Zd>.<T>, <Zn>.<Tb>, #<const>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
when '000' UNDEFINED;
when '001' esize = 8;
when '01x' esize = 16;
when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler symbols

<Zd>   Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>   Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
       8     when tszh = 0, tszl = 01
       H     when tszh = 0, tszl = 1x
       S     when tszh = 1, tszl = xx

The encoding tszh = 0, tszl = 00 is reserved.

<Zn>   Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb>   Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
       H     when tszh = 0, tszl = 01
       S     when tszh = 0, tszl = 1x
       D     when tszh = 1, tszl = xx

The encoding tszh = 0, tszl = 00 is reserved.

<const>   Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (SInt(element) + round_const) >> shift;
    Elem[result, 2e + 0, esize] = SignedSat(res, esize);
    Elem[result, 2e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
C8.2.547  SQRSHRNT

Shift each signed integer value in the source vector elements right by an immediate value, and place the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)}) - 1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Encoding

SQRSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  B when tszh = 0, tszl = 01
  H when tszh = 0, tszl = 1x
  S when tszh = 1, tszl = xx
The encoding tszh = 0, tszl = 00 is reserved.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  H when tszh = 0, tszl = 01
  S when tszh = 0, tszl = 1x
  D when tszh = 1, tszl = xx
The encoding tszh = 0, tszl = 00 is reserved.
<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = (SInt(element) + round_const) >> shift;
  Elem[result, 2*e + 1, esize] = SignedSat(res, esize);

Z[d, VL] = result;
C8.2.548 SQRSHRUNB

Shift each signed integer value in the source vector elements right by an immediate value, and place the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Encoding
SQRSHRUNB <Zd>.<T>, <Zn>.<Tb>, #<const>

Decode for this encoding
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  B when tszh = 0, tszl = 01
  H when tszh = 0, tszl = 1x
  S when tszh = 1, tszl = xx
The encoding tszh = 0, tszl = 00 is reserved.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  H when tszh = 0, tszl = 01
  S when tszh = 0, tszl = 1x
  D when tszh = 1, tszl = xx
The encoding tszh = 0, tszl = 00 is reserved.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = (SInt(element) + round_const) >> shift;
  Elem[result, 2*e + 0, esize] = UnsignedSat(res, esize);
  Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
C8.2.549  **SQRSHRUNT**

Shift each signed integer value in the source vector elements right by an immediate value, and place the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
```

---

**Encoding**

SQRSHRUNT <Zd>.<T>, <Zn>.<Tb>, #<const>

**Decode for this encoding**

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

**Assembler symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  - B when tszh = 0, tszl = 01
  - H when tszh = 0, tszl = 1x
  - S when tszh = 1, tszl = xx
  - The encoding tszh = 0, tszl = 00 is reserved.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>` Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  - H when tszh = 0, tszl = 01
  - S when tszh = 0, tszl = 1x
  - D when tszh = 1, tszl = xx
  - The encoding tszh = 0, tszl = 00 is reserved.
- `<const>` Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".
constant integer elements = VL DIV (2 \* esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
  bits(2\*esize) element = Elem[operand, e, 2\*esize];
  integer res = (SInt(element) + round_const) >> shift;
  Elem[result, 2\*e + 1, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
C8.2.550  SQSHL (immediate)

Shift left by immediate each active signed element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)} - 1\). The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.

Encoding

SQSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
  case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
  integer g = UInt(Pg);
  integer dn = UInt(Zdn);
  integer shift = UInt(tsize:imm3) - esize;

Assembler symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
- B  when tszh = 00, tszl = 01
- H  when tszh = 00, tszl = 1x
- S  when tszh = 01, tszl = xx
- D  when tszh = 1x, tszl = xx

The encoding tszh = 00, tszl = 00 is reserved.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;
for e = 0 to elements-1
integer element1 = SInt(Elem[operand1, e, esize]);
if Elem[mask, e, esize] == '1' then
  integer res = element1 << shift;
  Elem[result, e, esize] = SignedSat(res, esize);
else
  Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.551   SQSHL (vectors)

Shift active signed elements of the first source vector by corresponding elements of the second source vector and
destructively place the results in the corresponding elements of the first source vector. A positive shift amount
performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is
saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Inactive elements in the destination vector
register remain unmodified.

Encoding

SQSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer res = element << shift;
    Elem[result, e, esize] = SignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.552       SQSHLR

Shift active signed elements of the second source vector by corresponding elements of the first source vector and
destructively place the results in the corresponding elements of the first source vector. A positive shift amount
performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is
saturated to the N-bit element's signed integer range -2^{(N-1)} to (2^{(N-1)})-1. Inactive elements in the destination vector
register remain unmodified.

Encoding

SQSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn>        Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>          Is the size specifier, encoded in the "size" field. It can have the following values:
B             when size = 00
H             when size = 01
S             when size = 10
D             when size = 11

<Pg>         Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>         Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1'
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer res = element << shift;
    Elem[result, e, esize] = SignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand2, e, esize];
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.553  SQSHLU

Shift left by immediate each active signed element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.

Encoding

SQSHLU <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

Decode for this encoding

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler symbols

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<t>` Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  - B when tszh = 00, tszl = 01
  - H when tszh = 00, tszl = 1x
  - S when tszh = 01, tszl = xx
  - D when tszh = 1x, tszl = xx
  The encoding tszh = 00, tszl = 00 is reserved.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<const>` Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

```java
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;
for e = 0 to elements-1
```
integer element1 = \texttt{SInt(Elem[operand1, e, esize])};
if \texttt{ElemP[mask, e, esize]} == '1' then
    integer res = element1 \ll shift;
    \texttt{Elem[result, e, esize]} = \texttt{UnsignedSat(res, esize)};
else
    \texttt{Elem[result, e, esize]} = \texttt{Elem[operand1, e, esize]};
\texttt{Z[dn, VL]} = result;

\textbf{Operational information}

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.554   SQSHRN B

Shift each signed integer value in the source vector elements right by an immediate value, and place the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)}) - 1\). The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Encoding

\[
\text{SQSHRN B } <Zd>.<T>, <Zn>.<Tb>, #<const>
\]

Decode for this encoding

\[
\text{if } \text{!HaveSVE2()} \&\& \text{!HaveSME()} \text{ then UNDEFINED;}
\]

\[
\text{bits(3) tsize = tszh:tszl;}
\]

\[
\text{integer esize;}
\]

\[
\text{case tsize of}
\]

\[
\text{when '000' UNDEFINED;}
\]

\[
\text{when '001' esize = 8;}
\]

\[
\text{when '01x' esize = 16;}
\]

\[
\text{when '1xx' esize = 32;}
\]

\[
\text{integer n = UInt(Zn);}
\]

\[
\text{integer d = UInt(Zd);}
\]

\[
\text{integer shift = (2 * esize) - UInt(tsize:imm3);}
\]

Assembler symbols

\(<Zd>\) Is the name of the destination scalable vector register, encoded in the "Zd" field.

\(<T>\) Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

\< 8 when tszh = 0, tszl = 01
\< H when tszh = 0, tszl = 1x
\< S when tszh = 1, tszl = xx

The encoding tszh = 0, tszl = 00 is reserved.

\(<Zn>\) Is the name of the source scalable vector register, encoded in the "Zn" field.

\(<Tb>\) Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

\< H when tszh = 0, tszl = 01
\< S when tszh = 0, tszl = 1x
\< D when tszh = 1, tszl = xx

The encoding tszh = 0, tszl = 00 is reserved.

\(<\text{const}>\) Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements - 1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = SInt(element) >> shift;
    Elem[result, 2*e + 0, esize] = SignedSat(res, esize);
    Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
C8.2.555   SQSHRNT

Shift each signed integer value in the source vector elements right by an immediate value, and place the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element’s signed integer range \(-2^{(N-1)} \) to \(2^{(N-1)} - 1\). The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Encoding

```
SQSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>
```

Decode for this encoding

```
if !HaveSVE2() & & !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
  when ’000’ UNDEFINED;
  when ’001’ esize = 8;
  when ’01x’ esize = 16;
  when ’1xx’ esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  - B when tszh = 0, tszl = 01
  - H when tszh = 0, tszl = 1x
  - S when tszh = 1, tszl = xx
  The encoding tszh = 0, tszl = 00 is reserved.
- `<Zn>` is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>` is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  - H when tszh = 0, tszl = 01
  - S when tszh = 0, tszl = 1x
  - D when tszh = 1, tszl = xx
  The encoding tszh = 0, tszl = 00 is reserved.
- `<const>` is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
```
constant integer \(\text{elements} = \text{VL} \, \text{DIV} \, (2 \times \text{esize})\);

\[\text{bits(VL) \, operand} = Z[n, \, \text{VL}]\];

\[\text{bits(VL) \, result} = Z[d, \, \text{VL}]\];

for \(e = 0\) to \(\text{elements-1}\)

\[\text{bits(2\times\text{esize}) \, element} = \text{Elem}[\text{operand, e, 2\times\text{esize}}]\];

\[\text{integer \, res} = \text{SInt(element)} \gg \text{shift};\]

\[\text{Elem}[\text{result, 2\times e + 1, \text{esize}}] = \text{SignedSat}(\text{res, esize})\];

\[Z[d, \, \text{VL}] = \text{result}\];
C8.2.556  SQSHRUNB

Shift each signed integer value in the source vector elements right by an immediate value, and place the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Encoding

SQSHRUNB <Zd>,<T>, <Zn>,<Tb>, #<const>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Asmmer symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  B  when tszh = 0, tszl = 01
  H  when tszh = 0, tszl = 1x
  S  when tszh = 1, tszl = xx
  The encoding tszh = 0, tszl = 00 is reserved.

<Zn>  Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb>  Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  H  when tszh = 0, tszl = 01
  S  when tszh = 0, tszl = 1x
  D  when tszh = 1, tszl = xx
  The encoding tszh = 0, tszl = 00 is reserved.

<const>  Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVENable();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = SInt(element) >> shift;
  Elem[result, 2*e + 0, esize] = UnsignedSat(res, esize);
  Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
C8.2.557  SQSHRUNT

Shift each signed integer value in the source vector elements right by an immediate value, and place the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to \((2^N) - 1\). The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Encoding

\[
[31\ 30\ 29\ 28\ 27\ 26\ 25\ 24\ 23\ 22\ 21\ 20\ 19\ 18\ 16\ 15\ 14\ 13\ 12\ 11\ 10\ 9\ 5\ 4\ 0]
\]

\[
0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ \text{tszh}\ \text{tszl}\ \text{imm3}\ 0\ 0\ 0\ 0\ 1\ 1\ \text{Zn}\ \text{Zd}
\]

\[\text{tszh} \quad \text{tszl} \quad \text{imm3} \quad \text{Zn} \quad \text{Zd}\]

Decoding for this encoding

\[
\text{if !haveSVE2() && !HaveSME()} \text{ then UNDEFINED};
\]

\[
\text{bits(3) tsze = tszh:tszl};
\]

\[
\text{integer esize};
\]

\[
\text{case tsze of}
\]

\[
\text{when '000' UNDEFINED};
\]

\[
\text{when '001' esize = 8};
\]

\[
\text{when '01x' esize = 16};
\]

\[
\text{when '1xx' esize = 32};
\]

\[
\text{integer n = UInt(Zn)};
\]

\[
\text{integer d = UInt(Zd)};
\]

\[
\text{integer shift = (2 * esize) - UInt(tsz:imm3)};
\]

Assembler symbols

\(<\text{Zd}>\)  
Is the name of the destination scalable vector register, encoded in the "Zd" field.

\(<\text{T}>\)  
Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

\(B\)  
when tszh = 0, tszl = 01

\(H\)  
when tszh = 0, tszl = 1x

\(S\)  
when tszh = 1, tszl = xx

The encoding tszh = 0, tszl = 00 is reserved.

\(<\text{Zn}>\)  
Is the name of the first source scalable vector register, encoded in the "Zn" field.

\(<\text{Tb}>\)  
Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

\(H\)  
when tszh = 0, tszl = 01

\(S\)  
when tszh = 0, tszl = 1x

\(D\)  
when tszh = 1, tszl = xx

The encoding tszh = 0, tszl = 00 is reserved.

\(<\text{const}>\)  
Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

\[
\text{CheckSVEEnabled();}
\]

\[
\text{constant integer VL = CurrentVL};
\]

\[
\text{constant integer PL = VL DIV 8};
\]
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = SInt(element) >> shift;
    Elem[result, 2*e + 1, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
C8.2.558 SQSUB (immediate)

Signed saturating subtract of an unsigned immediate from each element of the source vector, and destructively place
the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's
signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)}) - 1\). This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also
be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option
is specified is "\(\#<imm>, \text{LSL} \#8\)". However an assembler and disassembler may also allow use of the shifted 16-bit
value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "\(\#0, \text{LSL} \#8\)".

Encoding

\[
\text{SQSUB} <\text{Zdn}>, <\text{T}>, #<\text{imm}>, \{, <\text{shift}>\}
\]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = FALSE;

Assembler symbols

\(<\text{Zdn}>\) Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

\(<\text{T}>\) Is the size specifier, encoded in the "size" field. It can have the following values:

<table>
<thead>
<tr>
<th>(&lt;\text{T}&gt;)</th>
<th>(B)</th>
<th>(H)</th>
<th>(S)</th>
<th>(D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>when size = 00</td>
<td>when size = 01</td>
<td>when size = 10</td>
<td>when size = 11</td>
<td></td>
</tr>
</tbody>
</table>

\(<\text{imm}>\) Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

\(<\text{shift}>\) Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh"
field. It can have the following values:

<table>
<thead>
<tr>
<th>(&lt;\text{shift}&gt;)</th>
<th>(\text{LSL #0})</th>
<th>(\text{LSL #8})</th>
</tr>
</thead>
<tbody>
<tr>
<td>when sh = 0</td>
<td>when sh = 1</td>
<td></td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bias(VL) operand1 = Z[dn, VL];
bias(VL) result;
for $e = 0$ to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - imm, esize, unsigned);

$Z[dn, VL] = result$;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.559  SQSUB (vectors, predicated)

Subtract active signed elements of the second source vector from corresponding signed elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range -2(N-1) to (2(N-1))-1. Inactive elements in the destination vector register remain unmodified.

Encoding

SQSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

Assemble symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>   Is the size specifier, encoded in the "size" field. It can have the following values:
    B when size = 00
    H when size = 01
    S when size = 10
    D when size = 11
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if Elem[mask, e, esize] == '1' then
        integer res = SInt(Sat(element1 - element2, esize, unsigned));
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.560  SQSUB (vectors, unpredicated)

Signed saturating subtract all elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range -2^{(N-1)} to (2^{(N-1)}-1. This instruction is unpredicated.

Encoding

SQSUB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = FALSE;

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

<table>
<thead>
<tr>
<th>&lt;T&gt;</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>00</td>
</tr>
<tr>
<td>H</td>
<td>01</td>
</tr>
<tr>
<td>S</td>
<td>10</td>
</tr>
<tr>
<td>D</td>
<td>11</td>
</tr>
</tbody>
</table>

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - element2, esize, unsigned);
Z[d, VL] = result;
C8.2.561   SQSUBR

Subtract active signed elements of the first source vector from corresponding signed elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)}-1\). Inactive elements in the destination vector register remain unmodified.

Encoding

SQSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

Assembler symbols

<Zdn>   Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>     Is the size specifier, encoded in the "size" field. It can have the following values:
        B when size = 00
        H when size = 01
        S when size = 10
        D when size = 11
<Pg>    Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>    Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, e, esize]);
  integer element2 = SInt(Elem[operand2, e, esize]);
  if Elem[mask, e, esize] == '1' then
    integer res = SInt(Sat(element2 - element1, esize, unsigned));
    Elem[result, e, esize] = res<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
$Z[dn, VL] = \text{result};$

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.562 SQXTNB

Saturate the signed integer value in each source element to half the original source element width, and place the results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero.

||5|4|0||
|---|---|---|---|
|31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9|  |  |  |
|0 1 0 0 0 1 0 0 1 1|t|s|z|l|0 0 0 0 1 0 0 0 0|Zn|Zd|

Encoding

SQXTNB <Zd>,<T>, <Zn>,<Tb>

Decode for this encoding

```plaintext
if !HaveSVE2() && !HaveSME() then UNDEFINED;

bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
  when '001' esize = 16;
  when '010' esize = 32;
  when '100' esize = 64;
  otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  - `B` when tszh = 0, tszl = 01
  - `H` when tszh = 0, tszl = 10
  - `S` when tszh = 1, tszl = 00

The following encodings are reserved:
- tszh = 0, tszl = 00.
- tszh = x, tszl = 11.
- tszh = 1, tszl = 01.
- tszh = 1, tszl = 10.

- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.
- `<Tb>` Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  - `H` when tszh = 0, tszl = 01
  - `S` when tszh = 0, tszl = 10
  - `D` when tszh = 1, tszl = 00

The following encodings are reserved:
- tszh = 0, tszl = 00.
- tszh = x, tszl = 11.
- tszh = 1, tszl = 01.
- tszh = 1, tszl = 10.
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, e, esize]);
  bits(halfesize) res = SignedSat(element1, halfesize);
  Elem[result, 2e + 0, halfesize] = res;
  Elem[result, 2e + 1, halfesize] = Zeros(halfesize);
Z[d, VL] = result;
C8.2.563 SQXTNT

Saturate the signed integer value in each source element to half the original source element width, and place the results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1</td>
<td>1</td>
<td>tszl</td>
<td>0 0 0 1 0 0 1</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SQXTNT <Zd>.<T>, <Zn>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bts(3) tsize = tszh:tszl;
integer esize;
case tsize of
  when '001' esize = 16;
  when '010' esize = 32;
  when '100' esize = 64;
  otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

B when tszh = 0, tszl = 01
H when tszh = 0, tszl = 10
S when tszh = 1, tszl = 00

The following encodings are reserved:

- tszh = 0, tszl = 00.
- tszh = x, tszl = 11.
- tszh = 1, tszl = 01.
- tszh = 1, tszl = 10.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

H when tszh = 0, tszl = 01
S when tszh = 0, tszl = 10
D when tszh = 1, tszl = 00

The following encodings are reserved:

- tszh = 0, tszl = 00.
- tszh = x, tszl = 11.
- tszh = 1, tszl = 01.
- tszh = 1, tszl = 10.
**Operation**

```plaintext
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    bits(halfesize) res = SignedSat(element1, halfesize);
    Elem[result, 2*e + 1, halfesize] = res;

Z[d, VL] = result;
```
C8.2.564   SQXTUNB

Saturate the signed integer value in each source element to an unsigned integer value that is half the original source element width, and place the results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1 0 1</td>
<td>0 0 0 1 0 1 0</td>
<td>0 0 0 1 0 1 0 1 0 0 Zn Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

tszh

Encoding

SQXTUNB <Zd>,<T>, <Zn>,<Tb>

Decode for this encoding

```c
if (!HaveSVE2() && !HaveSME()) then UNDEFINED;

bits(3) tsize = tszh:tszl;
integer esize;

case tsize of
  when '001' esize = 16;
  when '010' esize = 32;
  when '100' esize = 64;
  otherwise UNDEFINED;

integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>   Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

- B when tszh = 0, tszl = 01
- H when tszh = 0, tszl = 10
- S when tszh = 1, tszl = 00

The following encodings are reserved:

- tszh = 0, tszl = 00.
- tszh = x, tszl = 11.
- tszh = 1, tszl = 01.
- tszh = 1, tszl = 10.

<Zn>  Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb>  Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

- H when tszh = 0, tszl = 01
- S when tszh = 0, tszl = 10
- D when tszh = 1, tszl = 00

The following encodings are reserved:

- tszh = 0, tszl = 00.
- tszh = x, tszl = 11.
- tszh = 1, tszl = 01.
• tszh = 1, tszl = 10.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, e, esize]);
  bits(halfesize) res = UnsignedSat(element1, halfesize);
  Elem[result, 2*e + 0, halfesize] = res;
  Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);
Z[d, VL] = result;
```
C8.2.565  SQXTUNT

Saturate the signed integer value in each source element to an unsigned integer value that is half the original source
element width, and place the results in the odd-numbered half-width destination elements, leaving the
even-numbered elements unchanged.

Encoding

SQXTUNT <Zd>.<T>, <Zn>.<Tb>

Decode for this encoding

if !HaveSVE2() & & !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
  when '001' esize = 16;
  when '010' esize = 32;
  when '100' esize = 64;
  otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

B when tszh = 0, tszl = 01
H when tszh = 0, tszl = 10
S when tszh = 1, tszl = 00

The following encodings are reserved:
* tszh = 0, tszl = 00.
* tszh = x, tszl = 11.
* tszh = 1, tszl = 01.
* tszh = 1, tszl = 10.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

H when tszh = 0, tszl = 01
S when tszh = 0, tszl = 10
D when tszh = 1, tszl = 00

The following encodings are reserved:
* tszh = 0, tszl = 00.
* tszh = x, tszl = 11.
* tszh = 1, tszl = 01.
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    bits(halfesize) res = UnsignedSat(element1, halfesize);
    Elem[result, 2*e + 1, halfesize] = res;

Z[d, VL] = result;
C8.2.566   SRHADD

Add active signed elements of the first source vector to corresponding signed elements of the second source vector, shift right one bit, and destructively place the rounded results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

SRHADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
count integer esize = 8 << UInt(size);
count integer g = UInt(Pg);
count integer dn = UInt(Zdn);
count integer m = UInt(Zm);

Assembler symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>   Is the size specifier, encoded in the "size" field. It can have the following values:
   B   when size = 00
   H   when size = 01
   S   when size = 10
   D   when size = 11
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
count integer VL = CurrentVL;
count integer PL = VL DIV 8;
count integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
count integer round_const = 1;
for e = 0 to elements-1 
   integer element1 = SInt(Elem[operand1, e, esize]);
   integer element2 = SInt(Elem[operand2, e, esize]);
   if Elem[mask, e, esize] == '1' then
      integer res = (element1 + element2 + round_const) >> 1;
      Elem[result, e, esize] = res<esize-1:0>;
   else
      Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.567 SRI

Shift each source vector element right by an immediate value, and insert the result into the corresponding vector element in the destination vector register, merging the shifted bits from each source element with existing bits in each destination vector element. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

**Encoding**

SRI <Zn>.<T>, <Zd>.<T>, #<const>

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer shift = (2 * esize) - UInt(tsize:imm3);

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
B when tszh = 00, tszl = 01
H when tszh = 00, tszl = 1x
S when tszh = 01, tszl = xx
D when tszh = 1x, tszl = xx
The encoding tszh = 00, tszl = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
  bits(esize) element1 = Elem[result, e, esize];
  bits(esize) element2 = Elem[operand, e, esize];
bits(esize) mask = LSR(Ones(esize), shift);
bits(esize) shiftedval = LSR(element2, shift);
Elem[result, e, esize] = (element1 AND (NOT mask)) OR shiftedval;

Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.568  SRSHL

Shift active signed elements of the first source vector by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.

Encoding

SRSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    integer res = (element + round_const) << shift;
    Elem[result, e, esize] = res<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.569 SRSHLR

Shift active signed elements of the second source vector by corresponding elements of the first source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.

Encoding

SRSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1'
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    integer res = (element + round_const) << shift;
    Elem[result, e, esize] = res<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand2, e, esize];
\[ Z[dn, VL] = \text{result}; \]

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.570  SRSHR

Shift right by immediate each active signed element of the source vector, and destructively place the rounded results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

Encoding

SRSHR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

Decode for this encoding

if !HaveSVEZ() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  B when tszh = 00, tszl = 01
  H when tszh = 00, tszl = 1x
  S when tszh = 01, tszl = xx
  D when tszh = 1x, tszl = xx
The encoding tszh = 00, tszl = 00 is reserved.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bites(VL) operand1 = Z[dn, VL];
bites(PL) mask = P[g, PL];
bites(VL) result;
integer round_const = 1 << (shift-1);
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, e, esize]);
if Elem[mask, e, esize] == '1' then
    integer res = (element1 + round_const) >> shift;
    Elem[result, e, esize] = res<esize-1:0>;
else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.571  SRSRA

Shift right by immediate each signed element of the source vector, preserving the sign bit, and add the rounded
intermediate result destructively to the corresponding elements of the addend vector. The immediate shift amount
is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16</th>
<th>15 14 13</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1</td>
<td>tszh</td>
<td>tszl</td>
<td>imm3</td>
<td>1 1 1 0</td>
<td>1 0</td>
<td>Zn</td>
<td>Zda</td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

SRSRA <Zda>.<T>, <Zn>.<T>, #<const>

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;

constant integer VL = CurrentVL;

integer esize;

case tszh of

  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;

integer n = UInt(Zn);

integer da = UInt(Zda);

integer shift = (2 * esize) - UInt(tszh:imm3);

Assembling symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

B when tszh = 00, tszl = 01
H when tszh = 00, tszl = 1x
S when tszh = 01, tszl = xx
D when tszh = 1x, tszl = xx

The encoding tszh = 00, tszl = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

**Operation**

CheckSVEEnabled();

constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;

bits(VL) operand1 = Z[n, VL];

bits(VL) operand2 = Z[da, VL];

integer round_const = 1 << (shift - 1);

for e = 0 to elements-1

  integer element = (SInt(Elem[operand1, e, esize]) + round_const) >> shift;

  Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

  01000101
Z[da, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.572 SSHLLB

Shift left by immediate each even-numbered signed element of the source vector, and place the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

Encoding

SSHLLB <Zd>.<T>, <Zn>.<Tb>, #<const>

Decoding for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bv(3) tsize = tszh:tszl;
integer esize;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
    H when tszh = 0, tszl = 01
    S when tszh = 0, tszl = 1x
    D when tszh = 1, tszl = xx
The encoding tszh = 0, tszl = 00 is reserved.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
    B when tszh = 0, tszl = 01
    H when tszh = 0, tszl = 1x
    S when tszh = 1, tszl = xx
The encoding tszh = 0, tszl = 00 is reserved.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element = Elem[operand, 2*e + 0, esize];
  integer shifted_value = SInt(element) << shift;
  Elem[result, e, 2*esize] = shifted_value<2*esize-1:0>;

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.573 **SSHLLT**

Shift left by immediate each odd-numbered signed element of the source vector, and place the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

![Instruction Encoding](image)

**Encoding**

SSHLLT `<Zd>..<T>, <Zn>..<Tb>, #<const>`

**Decode for this encoding**

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tsz:tszl;
integer esize;
  case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
  integer n = UInt(Zn);
  integer d = UInt(Zd);
  integer shift = UInt(tsize:imm3) - esize;
```

**Assembler symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in the "tsz:tszl" field. It can have the following values:
  - H when tszh = 0, tszl = 01
  - S when tszh = 0, tszl = 1x
  - D when tszh = 1, tszl = xx
  The encoding tszh = 0, tszl = 00 is reserved.
- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.
- `<Tb>` Is the size specifier, encoded in the "tsz:tszl" field. It can have the following values:
  - B when tszh = 0, tszl = 01
  - H when tszh = 0, tszl = 1x
  - S when tszh = 1, tszl = xx
  The encoding tszh = 0, tszl = 00 is reserved.
- `<const>` Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
```
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element = Elem[operand, 2*e + 1, esize];
  integer shifted_value = SInt(element) << shift;
  Elem[result, e, 2*esize] = shifted_value<2*esize-1:0>;

Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SSRA

Shift right by immediate each signed element of the source vector, preserving the sign bit, and add the truncated intermediate result destructively to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Encoding

SSRA <Zda>.<T>, <Zn>.<T>, #<const>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
  when '0000' UNDEFINED;
  when '0011' esize = 8;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer n = UInt(Zn);
integer da = UInt(Zda);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  B when tszh = 00, tszl = 01
  H when tszh = 00, tszl = 1x
  S when tszh = 01, tszl = xx
  D when tszh = 1x, tszl = xx
The encoding tszh = 00, tszl = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
  integer element = SInt(Elem[operand1, e, esize]) >> shift;
  Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
$Z[da, VL] = \text{result};$

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.575  **SSUBLB**

Subtract the even-numbered signed elements of the second source vector from the corresponding signed elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

![Encoding](image)

**Encoding**

SSUBLB `<Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>`

**Decode for this encoding**

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
n = UInt(Zn);
m = UInt(Zm);
d = UInt(Zd);
Sel1 = 0;
Sel2 = 0;
unsigned = FALSE;
```

**Assembler symbols**

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>`: Is the size specifier, encoded in the "size" field. It can have the following values:
  - **H**: when size = 01
  - **S**: when size = 10
  - **D**: when size = 11

  The encoding size = 00 is reserved.
- `<Zn>`: Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>`: Is the size specifier, encoded in the "size" field. It can have the following values:
  - **B**: when size = 01
  - **H**: when size = 10
  - **S**: when size = 11

  The encoding size = 00 is reserved.
- `<Zm>`: Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
```
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.576 SSUBLBT

Subtract the odd-numbered signed elements of the second source vector from the even-numbered signed elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

\[
\begin{array}{cccccccccccc}
\text{31} & \text{30} & \text{29} & \text{28} & \text{27} & \text{26} & \text{25} & \text{24} & \text{23} & \text{22} & \text{21} & \text{20} \\
\text{16} & \text{15} & \text{14} & \text{13} & \text{12} & \text{11} & \text{10} & \text{9} & \text{8} & \text{7} & \text{6} & \text{5} \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & \text{size} & 0 & \text{Zm} & 1 & 0 & 0 & 0 & 1 & 0 & \text{Zn} & \text{Zd} \\
\end{array}
\]

**Encoding**

SSUBLBT \(<Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>\)

**Decode for this encoding**

if !HaveSVE2() \&\& !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 1;
boolean unsigned = FALSE;

**Assembler symbols**

\(<Zd>\) Is the name of the destination scalable vector register, encoded in the "Zd" field.

\(<T>\) Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

\(<Zn>\) Is the name of the first source scalable vector register, encoded in the "Zn" field.

\(<Tb>\) Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 01
- H when size = 10
- S when size = 11

The encoding size = 00 is reserved.

\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;
    Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.577 SSUBLT

Subtract the odd-numbered signed elements of the second source vector from the corresponding signed elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

Encoding
SSUBLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 1;
boolean unsigned = FALSE;

Assembler symbols
<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
  integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
  integer res = element1 - element2;
  Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.578   SSUBLTB

Subtract the even-numbered signed elements of the second source vector from the odd-numbered signed elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>size</td>
</tr>
<tr>
<td>Zm</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>

**Encoding**

SSUBLTB <Zd>,<T>, <Zn>,<Tb>, <Zm>,<Tb>

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 0;
boolean unsigned = FALSE;

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 01
- H when size = 10
- S when size = 11

The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.579  SSUBWB

Subtract the even-numbered signed elements of the second source vector from the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

Encoding
SSUBWB <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>

Decode for this encoding
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols
<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.

Operation
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
\[
\text{Elem}(\text{result}, e, \text{esize}) = (\text{element1} - \text{element2})<\text{esize}-1:0>;
\]
\[
Z[d, VL] = \text{result};
\]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.580 SSUBWT

Subtract the even-numbered signed elements of the second source vector from the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

Encoding

SSUBWT <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, e, esize]);
  integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
\[ \text{Elem}[\text{result}, e, \text{esize}] = (\text{element1} - \text{element2})\text{<esize-1:0>}; \]

\[ Z[d, VL] = \text{result}; \]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.581  ST1B (scalar plus immediate)

Contiguous store of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

Encoding

\[
\text{ST1B \{ <Zt>.<T>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}] \}
\]

Decode for this encoding

\[
\text{if !HaveSVE() \&\& !HaveSME() then UNDEFINED;} \\
\text{integer t = UInt(Zt);} \\
\text{integer n = UInt(Rn);} \\
\text{integer g = UInt(Pg);} \\
\text{constant integer esize = 8 << UInt(size);} \\
\text{constant integer msize = 8;} \\
\text{integer offset = sint(imm4);} \\
\]

Assembler symbols

\[
\text{<Zt>} \quad \text{Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.} \\
\text{<T>} \quad \text{Is the size specifier, encoded in the "size" field. It can have the following values:} \\
\quad \text{B when size = 00} \\
\quad \text{H when size = 01} \\
\quad \text{S when size = 10} \\
\quad \text{D when size = 11} \\
\text{<Pg>} \quad \text{Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.} \\
\text{<Xn|SP>} \quad \text{Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.} \\
\text{<imm>} \quad \text{Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.} \\
\]

Operation

\[
\text{CheckSVEEnabled();} \\
\text{constant integer VL = CurrentVL;} \\
\text{constant integer PL = VL DIV 8;} \\
\text{constant integer elements = VL DIV esize;} \\
\text{bits(64) base;} \\
\text{bits(PL) mask = P[g, Pl];} \\
\text{bits(VL) src;} \\
\text{constant integer mbytes = msize DIV 8;} \\
\]

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
  CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n, 64];
  src = Z[t, VL];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer eoff = (offset + elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
C8.2.582  **ST1B (scalar plus scalar)**

Contiguous store of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 16 | 15 | 14 | 13 | 12 | 10 | 9 | 5 | 4 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 1  | 0  | 0  | size | Rm | 0  | 1  | 0  | Pg | Rn | Zt |

**Encoding**

ST1B { <Zt>, <T>, <Pg>, [<Xn|SP>, <Xm>]

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8 << UInt(size);
constant integer msize = 8;

**Assembler symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
b muchos(64) base;
b muchos(PL) mask = P[g, PL];
b muchos(64) offset;
b muchos(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();

ARM DDI 0487I.a
Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
ID081822
Non-Confidential
else

    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1

    if Elem[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;

C8.2.583 ST1B (scalar plus vector)

Scatter store of bytes from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit unpacked unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 0 0 0</td>
<td>Zm 1 xs 0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
</tr>
</tbody>
</table>

Encoding

ST1B { <Zt>.D }, <Pg>, [Xn|SP>, <Zm>.D, <mod>]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 0 1 0</td>
<td>Zm 1 xs 0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
</tr>
</tbody>
</table>

Encoding

ST1B { <Zt>.S }, <Pg>, [Xn|SP>, <Zm>.S, <mod>]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
### 64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 1 1 0 0</td>
<td>1 0 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>

msz<0>

#### Encoding

```
ST1B { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D]
```

#### Decode for this encoding

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

#### Assembler symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Zm>`: Is the name of the offset scalable vector register, encoded in the "Zm" field.
- `<mod>`: Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:
  - `UXTW` when xs = 0
  - `SXTW` when xs = 1

#### Operation for all encodings

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];
    src = Z[t, VL];
```
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
C8.2.584   ST1B (vector plus immediate)

Scatter store of bytes from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0</td>
<td>1 0</td>
<td>1</td>
<td>imm5</td>
<td>1 0</td>
<td>Pg</td>
<td>Zn</td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

Encoding

ST1B { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
integer offset = UInt(imm5);

64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0</td>
<td>0</td>
<td>0</td>
<td>1 0</td>
<td>imm5</td>
<td>1 0</td>
<td>Pg</td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

Encoding

ST1B { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offset = UInt(imm5);

Assembler symbols

<Zt>   Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>   Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "ims5" field.

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(ElemP[base, e, esize], 64) + offset * mbytes;
        Mem[addr, mbytes, AccType_SVE] = ElemP[src, e, esize]<msize-1:0>;}
C8.2.585 ST1D (scalar plus immediate)

Contiguous store of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

Encoding

ST1D { <Zt>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offset = SInt(imm4);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    src = Z[t, VL];
  for e = 0 to elements-1
if $\text{Elem}[\text{mask, e, esize}] = '1'$ then
  integer $\text{eoff} = (\text{offset} \times \text{elements}) + e$;
  bits(64) $\text{addr} = \text{base} + \text{eoff} \times \text{mbytes}$;
  $\text{Mem}[\text{addr, mbytes, AccType\_SVE}] = \text{Elem}[\text{src, e, esize}]<\text{msize}-1:0>$;
C8.2.586  ST1D (scalar plus scalar)

Contiguous store of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

```
[31 30 29 28|27 26 25 24|23 22 21 20]  16|15 14 13 12| 10 9 | 5 4 | 0 |
 1 1 1 0 0 1 0 | 1 1 1 | Rm  0 1 0 | Pg | Rn | Zt
```

Encoding

STID { <Zt>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

Decoded for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;

Assembler symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(esize) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n, 64];
  offset = X[m, 64];
  src = Z[t, VL];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
\begin{align*}
\text{bits}(64) \, \text{addr} &= \text{base} + (\text{UINT}(\text{offset}) + e) \times mbytes; \\
\text{Mem}[\text{addr}, mbytes, \text{AccType}_{\text{SVE}}] &= \text{Elem}[\text{src}, e, esize]<msize-1:0>;
\end{align*}
C8.2.587  ST1D (scalar plus vector)

Scatter store of doublewords from the active elements of a vector register to the memory addresses generated by a
64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits
and then optionally multiplied by 8. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and
enabled at the current Exception level.

32-bit unpacked scaled offset

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 | 0 |
1 1 1 0 0 1 0 |1 1 0 1 Zm 1 xs 0 Pg Rn Zt
```

Encoding

ST1D [ <Zt>.D ], <Pg>, [ <Xn|SP>, <Zm>.D, <mod> #3 ]

Decode for this encoding

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 3;
```

32-bit unpacked unscaled offset

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 | 0 |
1 1 1 0 0 1 0 |1 1 0 0 Zm 1 xs 0 Pg Rn Zt
```

Encoding

ST1D [ <Zt>.D ], <Pg>, [ <Xn|SP>, <Zm>.D, <mod> ]

Decode for this encoding

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```
64-bit scaled offset

[31 30 29 28][27 26 25 24][23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 | 0 |
1 1 1 0 0 1 0 1 1 1 0 1 Zm 1 0 1 Pg Rn Zt

msz<1>
msz<0>

Encoding

STID { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, LSL #3]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 64;
boolean offs unsigned = TRUE;
integer scale = 3;

64-bit unscaled offset

[31 30 29 28][27 26 25 24][23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 | 0 |
1 1 1 0 0 1 0 1 1 0 0 Zm 1 0 1 Pg Rn Zt

msz<1>
msz<0>

Encoding

STID { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 64;
boolean offs unsigned = TRUE;
integer scale = 0;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

- **UXTW** when \( xs = 0 \)
- **SXTW** when \( xs = 1 \)

### Operation for all encodings

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];
    src = Z[t, VL];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    bits(64) addr = base + (off << scale);
    Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```
C8.2.588   ST1D (vector plus immediate)

Scatter store of doublewords from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

Encoding

ST1D {<Zt>.D}, <Pg>, [<Zn>.D{, #<imm>}]

Decode for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offset = UInt(imm5);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    src = Z[t, VL];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;}
C8.2.589  ST1H (scalar plus immediate)

Contiguous store of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

Encoding

ST1H { <Zt>.<T> }, <Pg>, [<Xn|SP>{{, #<imm>, MUL VL}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == "00" then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8 << UInt(size);
constant integer msize = 16;
integer offset = SInt(imm4);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bites(64) base;
bites(PL) mask = P[g, Pl];
bites(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    src = Z[t, VL];
  end
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer eoff = (offset + elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
C8.2.590   ST1H (scalar plus scalar)

Contiguous store of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

\[
\begin{array}{ccccccccccccccc}
|31|30|29|28|27|26|25|24|23|22|21|20|16|15|14|13|12|10| 9 | 5 | 4 | 0 |
\end{array}
\]

Encoding

ST1H \{ <Zt>.<T> }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8 << UInt(size);
constant integer msize = 16;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11

The encoding size = 00 is reserved.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
    src = Z[t, VL];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
C8.2.591  **ST1H (scalar plus vector)**

Scatter store of halfwords from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

**32-bit scaled offset**

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 | 0 |
  1 1 1 0 0 1 0 | 0 1 1 1 | Zm   1 xs 0 | Pg  | Rn  | Zt
```

**Encoding**

ST1H { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #1]

**Decode for this encoding**

```plaintext
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

**32-bit unpacked scaled offset**

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 | 0 |
  1 1 1 0 0 1 0 | 0 1 1 1 | Zm   1 xs 0 | Pg  | Rn  | Zt
```

**Encoding**

ST1H { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #1]

**Decode for this encoding**

```plaintext
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```
32-bit unpacked unscaled offset

```
   31 30 29 28|27 26 25 24|23 22 21 20|   16|15 14 13 12|  10 9  |  5 4 |  0 |
   1 1 1 0 0 1 0 0 1 1 0 0  Zm  1 xs 0  Pg  Rn  Zt
```

```
msz<1>
msz<0>
```

**Encoding**

```
ST1H { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]
```

**Decode for this encoding**

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

32-bit unscaled offset

```
   31 30 29 28|27 26 25 24|23 22 21 20|   16|15 14 13 12|  10 9  |  5 4 |  0 |
   1 1 1 0 0 1 0 0 1 1 0 0  Zm  1 xs 0  Pg  Rn  Zt
```

```
msz<1>
msz<0>
```

**Encoding**

```
ST1H { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]
```

**Decode for this encoding**

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```
64-bit scaled offset

\[
\begin{array}{cccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & Zt \\
16 & 15 & 14 & 13 & 12 & 10 & 9 & 5 & 4 & 0 & 0 & Pg \\
11100100 & 0 & 0 & 0 & Zm \\
31 30 29 28 27 26 25 24 23 22 21 20 \\
1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & Rn \\
11100100 & 1 & 1 & 0 & Zt
\end{array}
\]

Encoding

ST1H { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, LSL #1]

Decode for this encoding

\[
\begin{array}{l}
\text{if !HaveSVE() then UNDEFINED;}
\text{integer } t = \text{UInt}(Zt);
\text{integer } n = \text{UInt}(Rn);
\text{integer } m = \text{UInt}(Zm);
\text{integer } g = \text{UInt}(Pg);
\text{constant integer esize } = 64;
\text{constant integer msize } = 16;
\text{integer offs_size } = 64;
\text{boolean offs_unsigned } = \text{TRUE};
\text{integer scale } = 1;
\end{array}
\]

64-bit unscaled offset

\[
\begin{array}{cccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & Zt \\
16 & 15 & 14 & 13 & 12 & 10 & 9 & 5 & 4 & 0 & 0 & Pg \\
11100100 & 1 & 1 & 0 & Zm \\
31 30 29 28 27 26 25 24 23 22 21 20 \\
1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & Rn \\
11100100 & 1 & 0 & Zt
\end{array}
\]

Encoding

ST1H { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D]

Decode for this encoding

\[
\begin{array}{l}
\text{if !HaveSVE() then UNDEFINED;}
\text{integer } t = \text{UInt}(Zt);
\text{integer } n = \text{UInt}(Rn);
\text{integer } m = \text{UInt}(Zm);
\text{integer } g = \text{UInt}(Pg);
\text{constant integer esize } = 64;
\text{constant integer msize } = 16;
\text{integer offs_size } = 64;
\text{boolean offs_unsigned } = \text{TRUE};
\text{integer scale } = 0;
\end{array}
\]

Assembler symbols

\(<Zt>\) Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
\(<Zm>\) Is the name of the offset scalable vector register, encoded in the "Zm" field.
Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

- **UXTW** when $xs = 0$
- **SXTW** when $xs = 1$

### Operation for all encodings

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];
    src = Z[t, VL];
  
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    bits(64) addr = base + (off << scale);
    Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

### C8.2.592 ST1H (vector plus immediate)

Scatter store of halfwords from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

#### 32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 10 9 8 7 6 5 4 0</th>
<th>1 1 1 0 0 1 0 0 1 1 1 0 1 0 1</th>
<th>imm5 1 0 1</th>
<th>Pg</th>
<th>Zn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz&lt;1&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>msz&lt;0&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

ST1H { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]

**Decode for this encoding**

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offset = UInt(imm5);
```

#### 64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 10 9 8 7 6 5 4 0</th>
<th>1 1 1 0 0 1 0 0 1 1 1 0 1 0 1</th>
<th>imm5 1 0 1</th>
<th>Pg</th>
<th>Zn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz&lt;1&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>msz&lt;0&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

ST1H { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]

**Decode for this encoding**

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offset = UInt(imm5);
```

**Assembler symbols**

- `<Zt>` is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

**Operation for all encodings**

```plaintext
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
basis(PL) mask = P[g, PL];
basis(VL) base;
basis(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        basis(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```
ST1W (scalar plus immediate)

Contiguous store of words from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

Encoding

ST1W {<Zt>.<T>}, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
i integer t = UInt(Zt);
i integer n = UInt(Rn);
i integer g = UInt(Pg);
constant integer esize = 32 << UInt(sz);
constant integer msize = 32;
i integer offset = SInt(imm4);

Assembler symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<T>  Is the size specifier, encoded in the "sz" field. It can have the following values:
  S when sz = 0
  D when sz = 1
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>  Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else

if n == 31 then CheckSPAlignment();
base = if n == 31 then SP[] else X[n, 64];
src = Z[t, VL];

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>:
C8.2.594  ST1W (scalar plus scalar)

Contiguous store of words from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

Encoding

```
ST1W { <Zt>.<T> }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]
```

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32 << UInt(sz);
constant integer msize = 32;
```

Assemble symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<T>` Is the size specifier, encoded in the "sz" field. It can have the following values:
  - `S` when `sz = 0`
  - `D` when `sz = 1`
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bv(PL) mask = P[g, PL];
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
```

```
offset = X[m, 64];
src = Z[t, VL];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;}
C8.2.595  **ST1W (scalar plus vector)**

Scatter store of words from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

### 32-bit scaled offset

![32-bit offset encoding](image)

**Encoding**

ST1W \{ <Zt>.S \}, <Pg>, \[<Xn|SP>, <Zm>.S, <mod> #2\]

**Decode for this encoding**

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

### 32-bit unpacked scaled offset

![32-bit offset encoding](image)

**Encoding**

ST1W \{ <Zt>.D \}, <Pg>, \[<Xn|SP>, <Zm>.D, <mod> #2\]

**Decode for this encoding**

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```
32-bit unpacked unscaled offset

```
[31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13 12| 10 9 | 5 4 | 0 ]
1 1 1 0 0 1 0 0 0 | Zm 1 xs 0 | Pg | Rn | Zt
```

Encoding

```
ST1W { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]
```

Decode for this encoding

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

32-bit unscaled offset

```
[31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13 12| 10 9 | 5 4 | 0 ]
1 1 1 0 0 1 0 1 0 | Zm 1 xs 0 | Pg | Rn | Zt
```

Encoding

```
ST1W { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]
```

Decode for this encoding

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```
### 64-bit scaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13</th>
<th>12</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

**Encoding**

```
ST1W { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, LSL #2]
```

**Decode for this encoding**

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 2;
```

### 64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13</th>
<th>12</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

**Encoding**

```
ST1W { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D]
```

**Decode for this encoding**

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

### Assembler symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Zm>` Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in the "xs" field. It can have the following values:

- **UXTW** when `xs = 0`
- **SXTW** when `xs = 1`

### Operation for all encodings

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = Z[m, VL];
        src = Z[t, VL];
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then
            integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
            bits(64) addr = base + (off << scale);
            Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```
C8.2.596   ST1W (vector plus immediate)

Scatter store of words from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

### 32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 1</td>
<td>0 1 1</td>
<td>imm5</td>
<td>1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

Encoding

ST1W { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]

Decoding for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
integer offset = UInt(imm5);

### 64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 1</td>
<td>0 1 1</td>
<td>imm5</td>
<td>1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>

Encoding

ST1W { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]

Decoding for this encoding

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offset = UInt(imm5);

### Assembler symbols

- <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

**Operation for all encodings**

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
basis(PL) mask = P[g, PL];
basis(VL) base;
basis(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    src = Z[t, VL];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```
C8.2.597 ST2B (scalar plus immediate)

Contiguous store two-byte structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector’s in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

Encoding

ST2B { <Zt1>.B, <Zt2>.B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 2;

Assembler symbols

<Zt1>   Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>   Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>    Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>   Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bias(64) base;
bias(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bias(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

    for r = 0 to nreg-1
        values[r] = Z[(t+r) MOD 32, VL];

    for e = 0 to elements-1
        for r = 0 to nreg-1
            if ElemP[mask, e, esize] == '1' then
                integer eoff = (offset * elements * nreg) + (e * nreg) + r;
                bits(64) addr = base + eoff * mbytes;
                Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.598   **ST2B (scalar plus scalar)**

Contiguous store two-byte structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

![Encoding Diagram](image)

**Encoding**

ST2B { <Zt1>.B, <Zt2>.B }, <Pg>, [<Xn|SP>, <Xm>]

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 2;
```

**Assembler symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
b[64] base;
b[PL] mask = P[g, PL];
b[64] offset;
constant integer mbytes = esize DIV 8;
array [0..1] of b[VL] values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrLimitUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
```

ARM DDI 04871.a  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  Non-Confidential
CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n, 64];
  offset = X[m, 64];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.599  ST2D (scalar plus immediate)

Contiguous store two-doubleword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

Encoding

ST2D { <Zt1>.D, <Zt2>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 2;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then


CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n, 64];
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32, VL];
for e = 0 to elements-1
  for r = 0 to nreg-1
    if Elem[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.600  **ST2D (scalar plus scalar)**

Contiguous store two-doubleword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

**Encoding**

```
ST2D { <Zt1>.D, <Zt2>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]
```

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 2;
```

**Assembler symbols**

- `<Zt1>` is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
```
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAignment();
    else
        if n == 31 then CheckSPAignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];
    for r = 0 to nreg-1
        values[r] = Z[(t+r) MOD 32, VL];
    for e = 0 to elements-1
        for r = 0 to nreg-1
            if ElemP[mask, e, esize] == '1' then
                integer eoff = UInt(offset) + (e * nreg) + r;
                bits(64) addr = base + eoff * mbytes;
                Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.601   ST2H (scalar plus immediate)

Contiguous store two-halfword structures, each from the same element number in two vector registers to the
memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16
to 14 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to
the two consecutive halfwords in memory which make up each structure. Inactive structures are not written to
memory.

Encoding

ST2H { <Zt1>.H, <Zt2>.H }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 2;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to
0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bites(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then

CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.602  ST2H (scalar plus scalar)

Contiguous store two-halfword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

Encoding

ST2H \{ <Zt1>.H, <Zt2>.H \}, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

Decoding for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 2;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, Pl];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainsPredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

    for r = 0 to nreg-1
        values[r] = Z[(t+r) MOD 32, VL];

    for e = 0 to elements-1
        for r = 0 to nreg-1
            if ElemP[mask, e, esize] == '1' then
                integer eoff = UInt(offset) + (e * nreg) + r;
                bits(64) addr = base + eoff * mbytes;
                Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.603  **ST2W (scalar plus immediate)**

Contiguous store two-word structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive structures are not written to memory.

```
Encoding
ST2W { <Zt1>.S, <Zt2>.S }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 2;

**Assembler symbols**

<Zt1>  Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>  Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>  Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
```

---

**Encoding Diagram**

```
                  | 5  | 4  | 0 |
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|14|13|12|10 9 | 1 1 1 0 | 0 1 | 0 1 1 | imm4| 1 1 1 | Pg | Rn | Zt

msz<1>
msz<0>
```

---

**Operation Code**

```
11100101
0x0 011
0x0 111
0x0

msz<1>
msz<0>
```
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if Elem[mask, e, esize] == '1'
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.604  ST2W (scalar plus scalar)

Contiguous store two-word structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive structures are not written to memory.

Encoding

ST2W {<Zt1>.S, <Zt2>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

Decode for this encoding

\[
\begin{align*}
\text{if } & \text{ !HaveSVE() } \&\& \text{ !HaveSME() } \text{ then UNDEFINED;} \\
\text{if } & \text{ Rm == '11111' then UNDEFINED;} \\
\text{integer } & \text{ t = UInt(Zt);} \\
\text{integer } & \text{ n = UInt(Rn);} \\
\text{integer } & \text{ m = UInt(Rm);} \\
\text{integer } & \text{ g = UInt(Pg);} \\
\text{constant integer } & \text{ esize = 32;} \\
\text{constant integer } & \text{ nreg = 2;} \\
\end{align*}
\]

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.605  **ST3B (scalar plus immediate)**

Contiguous store three-byte structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

**Encoding**

\[
\text{ST3B} \{ <Zt1>.B, <Zt2>.B, <Zt3>.B \}, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
\]

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

**Assembler symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
blob PL mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
```

```c
```
if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.606  ST3B (scalar plus scalar)

Contiguous store three-byte structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

Encoding

\[
\text{ST3B \{ <Zt1>.B, <Zt2>.B, <Zt3>.B \}, <Pg>, [<Xn|SP>, <Xm>]} 
\]

Decode for this encoding

\[
\begin{align*}
\text{if \! HaveSVE() \&\& \! HaveSME() \text{ then UNDEFINED;}} \\
\text{if \ Rm == '11111' \text{ then UNDEFINED;}} \\
\text{integer t = UInt(Zt);} \\
\text{integer n = UInt(Rn);} \\
\text{integer m = UInt(Rm);} \\
\text{integer g = UInt(Pg);} \\
\text{constant integer esize = 8;} \\
\text{constant integer nreg = 3;} \\
\end{align*}
\]

Assembler symbols

\[
\begin{align*}
\text{<Zt1>} & \quad \text{Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.} \\
\text{<Zt2>} & \quad \text{Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.} \\
\text{<Zt3>} & \quad \text{Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.} \\
\text{<Pg>} & \quad \text{Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.} \\
\text{<Xn|SP>} & \quad \text{Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.} \\
\text{<Xm>} & \quad \text{Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.} \\
\end{align*}
\]

Operation

\[
\begin{align*}
\text{CheckSVEEnabled();} \\
\text{constant integer VL = CurrentVL;} \\
\text{constant integer PL = VL DIV 8;} \\
\text{constant integer elements = VL DIV esize;} \\
\text{bits(64) base;} \\
\text{bits(PL) mask = P[g, PL];} \\
\text{bits(64) offset;} \\
\text{constant integer mbytes = esize DIV 8;} \\
\text{array \{0..2\} of bits(VL) values;} \\
\text{if HaveMTEExt() then SetTagCheckedInstruction(TRUE);} \\
\end{align*}
\]
if !AnyActiveElement(mask, esize) then
  if n == 31 & ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAignment();
  else
    if n == 31 then CheckSPAignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
  for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];
  for e = 0 to elements-1
    for r = 0 to nreg-1
      if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + (e * nreg) + r;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.607   **ST3D (scalar plus immediate)**

Contiguous store three-doubleword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

```
| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 | 14 | 13 | 12 | 10 | 9 | 5 | 4 | 0 |
| 1 1 1 0 0 0 1 0 | 1 1 1 0 1 | imm4 | 1 1 1 | Pg | Rn | Zt |
```

**Encoding**

ST3D { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

**Assembler symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, Pl];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
```
if !AnyActiveElement(mask, esize) then
  if n == 31 &
  ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then
      CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
  for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];
  for e = 0 to elements-1
    for r = 0 to nreg-1
      if ElemP[mask, e, esize] == '1'
        integer eoff = (offset * elements * nreg) + (e * nreg) + r;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.608  ST3D (scalar plus scalar)

Contiguous store three-doubleword structures, each from the same element number in three vector registers to the
memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL
option) and added to the base address. After each structure access the index value is incremented by three. The index
register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to
the three consecutive doublewords in memory which make up each structure. Inactive structures are not written to
memory.

Encoding

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13</th>
<th>12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

msz<1>
msz<0>
```

```
ST3D { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]
```

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 3;
```

Assembler symbols

```
<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
```

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bases(64) base;
bases(PL) mask = P[g, PL];
bases(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bases(VL) values;
```
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrantUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPathignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

    for r = 0 to nreg-1
        values[r] = Z[(t+r) MOD 32, VL];

    for e = 0 to elements-1
        for r = 0 to nreg-1
            if ElemP[mask, e, esize] == '1' then
                integer eoff = UInt(offset) + (e * nreg) + r;
                bits(64) addr = base + eoff * mbytes;
                Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.609  ST3H (scalar plus immediate)

Contiguous store three-halfword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

```
[31 30 29 28][27 26 25 24][23 22 21 20][19 16 15 14 13 12] 10 9 5 4 0
1 1 0 0 1 0 1 1 1 1 imm4 Pg Rn Zt
```

msz<1>
misz<0>

Encoding

ST3H { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>, [+Xn|SP]{, #imm, MUL VL}]

Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

Assembler symbols

<Zt1>  Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>  Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3>  Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>  Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Operation

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
basis(64) base;
basis(PL) mask = P[g, Pl];
constant integer mbytes = esize DIV 8;
array[0..2] of basis(PL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
```
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
    for r = 0 to nreg-1
        values[r] = Z[(t+r) MOD 32, VL];
    for e = 0 to elements-1
        for r = 0 to nreg-1
            if ElemP[mask, e, esize] == '1' then
                integer eoff = (offset * elements * nreg) + (e * nreg) + r;
                bits(64) addr = base + eoff * mbytes;
                Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.610   ST3H (scalar plus scalar)

Contiguous store three-halfword structures, each from the same element number in three vector registers to the
memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL
option) and added to the base address. After each structure access the index value is incremented by three. The index
register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to
the three consecutive halfwords in memory which make up each structure. Inactive structures are not written to
memory.

Encoding

ST3H { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 3;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
basis(64) base;
basis(PL) mask = P[g, Pl];
basis(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of basis(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 &\& ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAignment();
  else
    if n == 31 then CheckSPAignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

  for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if Elem[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.611   ST3W (scalar plus immediate)

Contiguous store three-word structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive structures are not written to memory.

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 3;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, Pl];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
if n == 31 \&\& ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.612 ST3W (scalar plus scalar)

Contiguous store three-word structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive structures are not written to memory.

Encoding

ST3W { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 3;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, Pl];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && Constrai
ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
  offset = X[m, 64];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if Elem[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.613   ST4B (scalar plus immediate)

Contiguous store four-byte structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector’s in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 4;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bite(64) base;
bite(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array {0..3} of bite(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
  end if
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32, VL];
end for
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
    end if
  end for
end for
C8.2.614  ST4B (scalar plus scalar)

Contiguous store four-byte structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

Encoding

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Pg</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
```

```
Encoding

```

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 4;
```

Assembler symbols

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Zt4>` Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bias(64) base;
bias(PL) mask = P[g, PL];
bias(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bias(VL) values;
```
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrumUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if Elem[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.615  ST4D (scalar plus immediate)

Contiguous store four-doubleword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 4;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
calendar integer PL = VL DIV 8;
calendar integer elements = VL DIV esize;
bias(64) base;
bias(PL) mask = P[g, PL];
calendar integer mbytes = esize DIV 8;
array [0..3] of bias(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
    end
for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.616   ST4D (scalar plus scalar)

Contiguous store four-doubleword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

Encoding

ST4D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 4;

Assembler symbols

<Zt1>   Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>   Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>   Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>   Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>    Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>    Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n, 64];
  offset = X[m, 64];
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32, VL];
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.617  ST4H (scalar plus immediate)

Contiguous store four-halfword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 4;

Assembler symbols

<Zt1>  Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>  Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>  Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>  Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>  Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
binary mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.618   ST4H (scalar plus scalar)

Contiguous store four-halfword structures, each from the same element number in four vector registers to the
memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL
option) and added to the base address. After each structure access the index value is incremented by four. The index
register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to
the four consecutive halfwords in memory which make up each structure. Inactive structures are not written to
memory.

```
0 1 1 0 0 1 0 0 | 1 1 1 | Rm 0 1 1 | Pg | Rn | Zt
```

Encoding


Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 4;
```

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
```
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 &\& ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then
      CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.619   ST4W (scalar plus immediate)

Contiguous store four-word structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector’s in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive structures are not written to memory.

Encoding


Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 4;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];

    for r = 0 to nreg-1
        values[r] = Z[(t+r) MOD 32, VL];

    for e = 0 to elements-1
        for r = 0 to nreg-1
            if ElemP[mask, e, esize] == '1' then
                integer eoff = (offset * elements * nreg) + (e * nreg) + r;
                bits(64) addr = base + eoff * mbytes;
                Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
C8.2.620   ST4W (scalar plus scalar)

Contiguous store four-word structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive structures are not written to memory.

Encoding

ST4W { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 4;

Assembler symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
### C8.2.621  STNT1B (scalar plus immediate)

Contiguous store non-temporal of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

![Hexadecimal and binary representation](image)

**Encoding**

STNT1B { <Zt>.B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
n = UInt(Rn);
g = UInt(Pg);
offset = SInt(imm4);
```

**Assembler symbols**

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
VL = CurrentVL;
PL = VL DIV 8;
elements = (VL DIV esize);
mask = P[g, PL];
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    src = Z[t, VL];
for e = 0 to elements-1
```

---

**Note:**

- **Encoding:** The encoding provided includes fields for the scalable vector register `<Zt>`, the governing predicate register `<Pg>`, the base register `<Xn|SP>`, and an optional signed immediate vector offset `<imm>`. The encoding also includes a multiplication by the vector's in-memory size (MUL VL).
- **Operation:** The operation involves checking the enabled state for SVE and SVE, then decoding the instructions based on the provided fields. It includes checks for active elements, base and source addresses, and a loop to process each element.
if `Elem[mask, e, esize] == '1'` then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
C8.2.622   STNT1B (scalar plus scalar)

Contiguous store non-temporal of bytes from elements of a vector register to the memory address generated by a 
64-bit scalar base and scalar index which is added to the base address. After each element access the index value is 
incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

Encoding

STNT1B { <Zt>.B }, <Pg>, [<Xn|SP>, <Xm>]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;

Assembler symbols

<Zt>       Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>       Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>    Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>       Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(VL) src;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
    src = Z[t, VL];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
C8.2.623  STNT1B (vector plus scalar)

Scatter store non-temporal of bytes from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit unscaled offset

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 0</td>
<td>0 0 1 0</td>
<td>Rm</td>
<td>0 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

STNT1B { <Zt>.S }, <Pg>, [<Zn>.S{, <Xm>}]  

**Decode for this encoding**

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;

64-bit unscaled offset

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 0</td>
<td>0 0 0 0</td>
<td>Rm</td>
<td>0 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

STNT1B { <Zt>.D }, <Pg>, [<Zn>.D{, <Xm>}]  

**Decode for this encoding**

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;

**Assembler symbols**

<Zt>  
Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

**Operation for all encodings**

```plaintext
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize]<msize-1:0>;
```
C8.2.624   STNT1D (scalar plus immediate)

Contiguous store non-temporal of doublewords from elements of a vector register to the memory address generated
by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size,
irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

Encoding

STNT1D { <Zt>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the
"imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g, PL];
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSAPAlignment();
else
  if n == 31 then CheckSAPalignment();
  base = if n == 31 then SP[] else X[n, 64];
  src = Z[t, VL];
for e = 0 to elements-1
if Elem[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
C8.2.625 STNT1D (scalar plus scalar)

Contiguous store non-temporal of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

```
[31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13|12| 10 9| 5 4| 0]
1 1 1 0 0 1 0 1 1 0 0 1 0 0 1 0

msz<1>
msz<0>
```

**Encoding**

STNT1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;

**Assembler symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(VL) src;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];
src = Z[t, VL];

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = base + (UINT(offset) + e) * mbytes;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
C8.2.626  STNT1D (vector plus scalar)

Scatter store non-temporal of doublewords from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

Encoding

STNT1D { <Zt>.D }, <Pg>, [<Zn>.D{, <Xm>}]

Decode for this encoding

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;

Assembler symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn>  Is the name of the base scalable vector register, encoded in the "Zn" field.

<Xm>  Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
  base = Z[n, VL];
  offset = X[m, 64];
  src = Z[t, VL];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
    Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize]<msize-1:0>;}
C8.2.627 STNT1H (scalar plus immediate)

Contiguous store non-temporal of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

Encoding

STNT1H { <Zt>.H }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g, PL];
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    src = Z[t, VL];
    for e = 0 to elements-1
if `Elem[mask, e, esize] == '1'` then
  integer eoff = (offset * elements) + e;
  bits(64) addr = base + eoff * mbytes;
  Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
C8.2.628 STNT1H (scalar plus scalar)

Contiguous store non-temporal of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

Encoding

STNT1H { <Zt>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(VL) src;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSVPAlignment();
    else
        if n == 31 then CheckSVPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];

src = Z[t, VL];

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    Mem[addr, mbytes, AccType_SVSTREAM] = Elem[src, e, esize];
C8.2.629  STNT1H (vector plus scalar)

Scatter store non-temporal of halfwords from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit unscaled offset

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [16] [15 14 13 12] [10 9] [5 4] 0
1 1 1 0 0 1 0 1 1 1 0 Rm 0 0 1 Pg Zn Zt
```

```
msz<1>
msz<0>
```

**Encoding**

STNT1H { <Zt>.S }, <Pg>, [<Zn>.S{, <Xm>}]}

**Decode for this encoding**

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;

64-bit unscaled offset

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [16] [15 14 13 12] [10 9] [5 4] 0
1 1 1 0 0 1 0 1 0 0 Rm 0 0 1 Pg Zn Zt
```

```
msz<1>
msz<0>
```

**Encoding**

STNT1H { <Zt>.D }, <Pg>, [<Zn>.D{, <Xm>}]

**Decode for this encoding**

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;

**Assembler symbols**

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation for all encodings

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize]<msize-1:0>;
C8.2.630  STNT1W (scalar plus immediate)

Contiguous store non-temporal of words from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

Encoding

\[
\text{STNT1W \{ <Zt>.S \}, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}}\]

Decode for this encoding

\[
\text{if } \text{!HaveSVE()} \&\& \text{!HaveSME()} \text{ then UNDEFINED;}
\]
\[
\text{integer } t = \text{UInt}(Zt); \quad \text{integer } n = \text{UInt}(Rn); \quad \text{integer } g = \text{UInt}(Pg); \quad \text{constant integer } esize = 32; \quad \text{integer offset} = \text{SInt}(\text{imm4});
\]

Assembler symbols

\(<Zt>\)  
 Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

\(<Pg>\)  
 Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Xn|SP>\)  
 Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<\text{imm}>\)  
 Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

\[
\text{CheckSVEEnabled();}
\]
\[
\text{constant integer } VL = \text{CurrentVL};
\]
\[
\text{constant integer } PL = VL \text{ DIV 8};
\]
\[
\text{constant integer } elements = VL \text{ DIV esize};
\]
\[
\text{bits(64) } base;
\]
\[
\text{constant integer } mbytes = esize \text{ DIV 8};
\]
\[
\text{bits(VL) } src;
\]
\[
\text{bits(PL) } mask = \text{P}[g, PL];
\]
\[
\text{if HaveMTEExt() then SetTagCheckedInstruction(n != 31);} \quad \text{if } \text{!AnyActiveElement(mask, esize)} \text{ then}
\]
\[
\text{if } n == 31 \&\& \text{ConstrainUnpredictableBool(\text{Unpredictable_CHECKSPNONEACTIVE}) then CheckSPadding();}
\]
\[
\text{else}
\]
\[
\text{if } n == 31 \text{ then CheckSPadding();}
\]
\[
\text{base} = \text{if } n == 31 \text{ then SP[]} \text{ else } X[n, 64];
\]
\[
\text{src} = Z[t, VL];
\]
\[
\text{for } e = 0 \text{ to elements-1}
\]
if Elem[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
C8.2.631   STNT1W (scalar plus scalar)

Contiguous store non-temporal of words from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

Encoding

STNT1W { <Zt>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) src;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n, 64];
        offset = X[m, 64];
        src = Z[t, VL];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
C8.2.632 STNT1W (vector plus scalar)

Scatter store non-temporal of words from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

32-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0</td>
<td>1 0 1 0</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

Encoding

STNT1W {<Zt>.S, <Pg>, [<Zn>.S{, <Xm>}]}

Decode for this encoding

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;

64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0</td>
<td>1 0 1 0</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

Encoding

STNT1W {<Zt>.D, <Pg>, [<Zn>.D{, <Xm>}]}

Decode for this encoding

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n =UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;

Assembler symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

**Operation for all encodings**

```c
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];
    src = Z[t, VL];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize]<msize-1:0>;
```

C8.2.633 STR (predicate)

Store a predicate register to a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current predicate register size in bytes. This instruction is unpredicated.

The store is performed as contiguous byte accesses, each containing 8 consecutive predicate bits in ascending element order, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then a general-purpose base register must be aligned to 2 bytes.

Encoding

STR <Pt>, [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Pt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);

Assembler symbols

<Pt> Is the name of the scalable predicate transfer register, encoded in the "Pt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

Operation

CheckSVEEnabled();
constant integer PL = CurrentVL DIV 8;
constant integer elements = PL DIV 8;
bits(PL) src;
bits(64) base;
integer offset = imm * elements;

if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n, 64];
src = P[t, PL];
boolean aligned = AArch64.CheckAlignment(base + offset, 2, AccType_SVE, TRUE);
for e = 0 to elements-1
    AArch64.MemSingle[base + offset, 1, AccType_SVE, aligned] = Elem[src, e, 8];
    offset = offset + 1;
C8.2.634   STR (vector)

Store a vector register to a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current vector register size in bytes. This instruction is unpredicated.

The store is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

```
| 31 30 29 28|27 26 25 24|23 22 21 | 16|15 14 13 12| 10 9 |  5 4 |  0 |
| 1 1 1 0 0 1 0 1 1 0| imm9h    | 0 1 0| imm9l | Rn  | Zt  |
```

**Encoding**

STR <Zt>, [<Xn|SP>{, #<imm>, MUL VL}]

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);

**Assembler symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>  Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 8;
bits(VL) src;
bits(64) base;
integer offset = imm * elements;
if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n, 64];
src = Z[t, VL];
boolean aligned = AArch64.CheckAlignment(base + offset, 16, AccType_SVE, TRUE);
for e = 0 to elements-1
    AArch64.MemSingle[base + offset, 1, AccType_SVE, aligned] = Elem[src, e, 8];
    offset = offset + 1;
### C8.2.635 SUB (immediate)

Subtract an unsigned immediate from each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is ";#<imm>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

**Encoding**

```plaintext
SUB <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}
```

**Decode for this encoding**

```plaintext
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
```

**Assembler symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - `B` when size = 00
  - `H` when size = 01
  - `S` when size = 10
  - `D` when size = 11
- `<imm>` Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:
  - LSL #0 when sh = 0
  - LSL #8 when sh = 1

**Operation**

```plaintext
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
```

```plaintext
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|  | 5 4 | 0]
0 0 1 0 0 1 0 1 size 1 0 0 0 0 1 1 1 sh imm8 Zdn
```
Elem[result, e, esize] = element1 - imm;
Z[dn, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.636   SUB (vectors, predicated)

Subtract active elements of the second source vector from corresponding elements of the first source vector and
destructively place the results in the corresponding elements of the first source vector. Inactive elements in the
destination vector register remain unmodified.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12</th>
<th>10 9</th>
<th>5  4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0</td>
<td>size</td>
<td>0 0 0 0 1 0 0</td>
<td>Pg</td>
</tr>
</tbody>
</table>
```

**Encoding**

SUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

**Assembler symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - `B` when `size = 00`
  - `H` when `size = 01`
  - `S` when `size = 10`
  - `D` when `size = 11`
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = element1 - element2;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
```
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register
    contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register
    contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate
  register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand
  register of this instruction.
C8.2.637  SUB (vectors, unpredicated)

Subtract all elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11  10  9</th>
<th>5  4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1</td>
<td>0 0 0 0 1</td>
<td>Zm</td>
<td>0 0 0 0 1</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

SUB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler symbols**

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

- **B**  when size = 00
- **H**  when size = 01
- **S**  when size = 10
- **D**  when size = 11

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = element1 - element2;
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.638  SUBHNB

Subtract each vector element of the second source vector from the corresponding vector element in the first source vector, and place the most significant half of the result in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. This instruction is unpredicated.

Encoding

SUBHNB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>  Is the size specifier, encoded in the "size" field. It can have the following values:
    B  when size = 01
    H  when size = 10
    S  when size = 11
    The encoding size = 00 is reserved.
<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb>  Is the size specifier, encoded in the "size" field. It can have the following values:
    H  when size = 01
    S  when size = 10
    D  when size = 11
    The encoding size = 00 is reserved.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
integer element2 = UInt(Elem[operand2, e, esize]);
integer res = (element1 - element2) >> halfesize;
Elem[result, 2*e + 0, halfesize] = res<halfesize-1:0>;
Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);

Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.639   SUBHNT

Subtract each vector element of the second source vector from the corresponding vector element in the first source vector, and place the most significant half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.

Encoding

SUBHNT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>  Is the size specifier, encoded in the "size" field. It can have the following values:
    B  when size = 01
    H  when size = 10
    S  when size = 11
    The encoding size = 00 is reserved.
<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb>  Is the size specifier, encoded in the "size" field. It can have the following values:
    H  when size = 01
    S  when size = 10
    D  when size = 11
    The encoding size = 00 is reserved.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
integer element2 = UInt(Elem[operand2, e, esize]);
integer res = (element1 - element2) >> halfesize;
Elem[result, 2*e + 1, halfesize] = res<halfesize-1:0>;
Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.640 SUBR (immediate)

Reversed subtract from an unsigned immediate each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1</td>
<td>1 0 0 0 1 1 1</td>
<td>sh</td>
<td>imm8</td>
<td>Zdn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

### Encoding

```
SUBR <Zdn>,<T>, <Zdn>,<T>, #<imm>{, <shift>}
```

### Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
```

### Assembler symbols

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<imm>` Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:
  - LSL #0 when sh = 0
  - LSL #8 when sh = 1

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
```
Elem[result, e, esize] = (imm - element1)<esize-1:0>;
Z[dn, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.641 SUBR (vectors)

Reversed subtract active elements of the first source vector from corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

SUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = element2 - element1;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.642  SUDOT

The signed by unsigned integer indexed dot product instruction computes the dot product of a group of four signed 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four unsigned 8-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE
(FEAT_I8MM)

Encoding

Decode for this encoding
if (!HaveSVE() && !HaveSME()) || !HaveInt8MatMulExt() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler symbols
<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
<imm> Is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

Operation
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
  integer segmentbase = e - (e MOD eltspersegment);
  integer s = segmentbase + index;
  bits(esize) res = Elem[operand3, e, esize];
for i = 0 to 3
    integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
    integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
    res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.643 SUNPKHI, SUNPKLO

Unpack elements from the lowest or highest half of the source vector and then sign-extend them to place in elements of twice their size within the destination vector. This instruction is unpredicated.

High half

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 1</td>
<td>size 1 1 0 0</td>
<td>0 1 0 0 1 1 1 0</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SUNPKHI <Zd>.<T>, <Zn>.<Tb>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
boolean hi = TRUE;

Low half

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 1</td>
<td>size 1 1 0 0</td>
<td>0 0 0 1 1 1 0</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SUNPKLO <Zd>.<T>, <Zn>.<Tb>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
boolean hi = FALSE;

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

H  when size = 01
S  when size = 10
D  when size = 11

The encoding size = 00 is reserved.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:

- **B** when size = 01
- **H** when size = 10
- **S** when size = 11

The encoding size = 00 is reserved.

### Operation for all encodings

```plaintext
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer hsize = esize DIV 2;
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(hsize) element = if hi then Elem[operand, e + elements, hsize] else Elem[operand, e, hsize];
    Elem[result, e, esize] = Extend(element, esize, unsigned);

Z[d, VL] = result;
```

### Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.644  SUQADD

Add active unsigned elements of the source vector to the corresponding signed elements of the addend vector, and destructively place the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's signed integer range `-2^(N-1)` to `(2^(N-1))-1`. Inactive elements in the destination vector register remain unmodified.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>9  8</th>
<th>7  6</th>
<th>5  4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>1 0 1 1 0 0 1 0</td>
<td>Pg</td>
<td>Zm</td>
<td>Zdn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SUQADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
   bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
if Elem[mask, e, esize] == '1' then
   Elem[result, e, esize] = SignedSat(SInt(element1) + UInt(element2), esize);
else
   Elem[result, e, esize] = Elem[operand1, e, esize];
```
Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.645   SXTB, SXTH, SXTW

Sign-extend the least-significant sub-element of each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

**Byte**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12] 10 9 | 5 4 | 0 |
0 0 0 0 0 1 0 0 | size 0 1 0 0 0 1 0 1 | Pg  Zn  Zd
```

**Encoding**

SXTB \(<Zd>.<T>\), \(<Pg>/M, <Zn>.<T>\)

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 8;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
```

**Halfword**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12] 10 9 | 5 4 | 0 |
0 0 0 0 0 1 0 0 | size 0 1 0 1 0 1 0 1 | Pg  Zn  Zd
```

**Encoding**

SXTH \(<Zd>.<T>\), \(<Pg>/M, <Zn>.<T>\)

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {\'0x\'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
```

**Word**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12] 10 9 | 5 4 | 0 |
0 0 0 0 0 1 0 0 | size 0 1 0 1 0 1 0 1 | Pg  Zn  Zd
```
Encoding

SXTW <Zd>.D, <Pg>/M, <Zn>.D

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size != '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> For the byte variant: is the size specifier, encoded in the "size" field. It can have the following values:
   H when size = 01
   S when size = 10
   D when size = 11
   The encoding size = 00 is reserved.
   For the halfword variant: is the size specifier, encoded in the "size<0>" field. It can have the following values:
   S when size<0> = 0
   D when size<0> = 1

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = Extend(element<s_esize-1:0>, esize, unsigned);
Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
— The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.646   TBL

Reads each element of the second source (index) vector and uses its value to select an indexed element from a table of elements consisting of one or two consecutive vector registers, where the first or only vector holds the lower numbered elements, and places the indexed table element in the destination vector element corresponding to the index vector element. If an index value is greater than or equal to the number of vector elements then it places zero in the corresponding destination vector element.

Since the index values can select any element in a vector this operation is not naturally vector length agnostic.

SVE

<table>
<thead>
<tr>
<th>size</th>
<th>Zm</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

TBL <Zd>.<T>, { <Zn>.<T> }, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = $ << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean double_table = FALSE;

SVE2

<table>
<thead>
<tr>
<th>size</th>
<th>Zm</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

TBL <Zd>.<T>, { <Zn1>.<T>, <Zn2>.<T> }, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = $ << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean double_table = TRUE;

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
D when size = 11

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation for all encodings

    CheckSVEEnabled();
    constant integer VL = CurrentVL;
    constant integer PL = VL DIV 8;
    constant integer elements = VL DIV esize;
    bits(VL) indexes = Z[m, VL];
    bits(VL) result;
    constant integer table_size = if double_table then VL*2 else VL;
    constant integer table_elements = table_size DIV esize;
    bits(table_size) table;

    if double_table then
        bits(VL) top = Z[(n + 1) MOD 32, VL];
        bits(VL) bottom = Z[n, VL];
        table = (top:bottom)<table_size-1:0>;
    else
        table = Z[n, table_size];

    for e = 0 to elements-1
        integer idx = UInt(Elem[indexes, e, esize]);
        Elem[result, e, esize] = if idx < table_elements then Elem[table, idx, esize] else Zeros(esize);

    Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.647   TBX

Reads each element of the second source (index) vector and uses its value to select an indexed element from a table of elements in the first source vector, and places the indexed element in the destination vector element corresponding to the index vector element. If an index value is greater than or equal to the number of vector elements then the corresponding destination vector element is left unchanged.

Since the index values can select any element in a vector this operation is not naturally vector length agnostic.

Encoding

TBX <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
    B when size = 00
    H when size = 01
    S when size = 10
    D when size = 11

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    integer element2 = UInt(Elem[operand2, e, esize]);
    if element2 < elements then
        Elem[result, e, esize] = Elem[operand1, element2, esize];
    Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.648 TRN1, TRN2 (predicates)

Interleave alternating even or odd-numbered elements from the first and second source predicates and place in elements of the destination predicate. This instruction is unpredicated.

### Even

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16 15 14 13 12 11 10 9 8 | 5 4 3 0]
0 0 0 0 1 0 1 | Pm 0 1 0 1 0 0 | Pn 0 | Pd 0
```

**Encoding**

TRN1 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 0;

### Odd

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16 15 14 13 12 11 10 9 8 | 5 4 3 0]
0 0 0 0 1 0 1 | Pm 0 1 0 1 0 1 | Pn 0 | Pd 0
```

**Encoding**

TRN2 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 1;

### Assembler symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation for all encodings

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (esize * 2);
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
for p = 0 to pairs-1
    Elem[result, 2*p+0, esize DIV 8] = Elem[operand1, 2*p+part, esize DIV 8];
    Elem[result, 2*p+1, esize DIV 8] = Elem[operand2, 2*p+part, esize DIV 8];
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.649  TRN1, TRN2 (vectors)

Interleave alternating even or odd-numbered elements from the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated. The 128-bit element variant of this instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant of the instruction is implemented.

Even

| [31 30 29 28][27 26 25 24][23 22 21 20] | 16|15 14 13 12|11 10 9 | 5 4 | 0 |
| 0 0 0 0 0 1 0 1 | Zm | 0 1 1 | 1 0 0 | Zn | Zd |

Encoding

TRN1 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;

Even (quadwords)

(FEAT_F64MM)

| [31 30 29 28][27 26 25 24][23 22 21 20] | 16|15 14 13 12|11 10 9 | 5 4 | 0 |
| 0 0 0 0 0 1 0 1 | Zm | 0 0 0 | 1 0 | Zn | Zd |

Encoding

TRN1 <Zd>.Q, <Zn>.Q, <Zm>.Q

Decode for this encoding

if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;

Odd

| [31 30 29 28][27 26 25 24][23 22 21 20] | 16|15 14 13 12|11 10 9 | 5 4 | 0 |
| 0 0 0 0 0 1 0 1 | Zm | 0 1 1 | 1 0 | Zn | Zd |

Even


Odd
**Encoding**

TRN2 (<Zd>,<T>, <Zn>,<T>, <Zm>,<T>)

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;

**Odd (quadwords)**

(FEAT_F64MM)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 1 0 1</td>
<td>Zm</td>
<td>0 0 0 1 1</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

TRN2 (<Zd>.Q, <Zn>.Q, <Zm>.Q)

**Decode for this encoding**

if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation for all encodings**

if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize * 2 then UNDEFINED;
constant integer pairs = VL DIV (esize * 2);
bah(VL) operand1 = Z[n, VL];
bah(VL) operand2 = Z[m, VL];
bah(VL) result = Zeros(VL);
for p = 0 to pairs-1
\[
\text{Elem[result, 2p+0, esize] = Elem[operand1, 2p+part, esize];}
\]
\[
\text{Elem[result, 2p+1, esize] = Elem[operand2, 2p+part, esize];}
\]
\[
Z[d, VL] = \text{result;}
\]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.650  UABA

Compute the absolute difference between unsigned integer values in elements of the second source vector and corresponding elements of the first source vector, and add the difference to the corresponding elements of the destination vector. This instruction is unpredicated.

Encoding

UABA <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean unsigned = TRUE;

Assembler symbols

<Zda>  Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

<table>
<thead>
<tr>
<th>&lt;T&gt;</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>size = 00</td>
</tr>
<tr>
<td>H</td>
<td>size = 01</td>
</tr>
<tr>
<td>S</td>
<td>size = 10</td>
</tr>
<tr>
<td>D</td>
<td>size = 11</td>
</tr>
</tbody>
</table>

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  integer element2 = Int(Elem[operand2, e, esize], unsigned);
  bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
  Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
Z[da, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.651 UABALB

Compute the absolute difference between even-numbered unsigned elements of the second source vector and corresponding elements of the first source vector, and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

Encoding

UABALB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
  integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
Z[da, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.652   UABALT

Compute the absolute difference between odd-numbered unsigned elements of the second source vector and corresponding elements of the first source vector, and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

Encoding

UABALT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
  integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
Z[da, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.653   UABD

Compute the absolute difference between unsigned integer values in active elements of the second source vector and corresponding elements of the first source vector and destructively place the difference in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

### Encoding

UABD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

### Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

### Assembler symbols

- `<Zdn>` is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  integer element2 = Int(Elem[operand2, e, esize], unsigned);
  if Elem[mask, e, esize] == '1' then
    integer absdiff = Abs(element1 - element2);
    Elem[result, e, esize] = absdiff<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
$Z[dn, VL] = \text{result};$

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.654  UABDLB

Compute the absolute difference between the even-numbered unsigned integer values in elements of the second source vector and the corresponding elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

```
\begin{verbatim}
[31 30 29 28 27 26 25 24 23 22 21 20 | 16 15 14 13 12 | 11 10 9 | 5 4 | 0]
0 1 0 0 0 1 0 1 size 0 Zn 0 0 1 1 1 0 Zm Zd

Encoding
UABDLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation
\hspace{1em} CheckSVEEnabled();
\hspace{1em} constant integer VL = CurrentVL;
\hspace{1em} constant integer elements = VL DIV esize;
\hspace{1em} bits(VL) operand1 = Z[n, VL];
\hspace{1em} bits(VL) operand2 = Z[m, VL];
\hspace{1em} bits(VL) result;
\hspace{1em} for e = 0 to elements-1
\hspace{2em} integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
\hspace{2em} integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
\end{verbatim}
```
integer res = Abs(element1 - element2);
Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.655 UABDLT

Compute the absolute difference between the odd-numbered unsigned integer values in elements of the second source vector and corresponding elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 |  5 4 |  0 |
  0 1 0 0 0 1 0 1 | size 0 | Zn 0 0 1 1 1 | Zn | Zd
```

Encoding

UABDLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler symbols

```
<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>  Is the size specifier, encoded in the "size" field. It can have the following values:
  H when size = 01
  S when size = 10
  D when size = 11
  The encoding size = 00 is reserved.
<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb>  Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 01
  H when size = 10
  S when size = 11
  The encoding size = 00 is reserved.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.
```

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
  integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
```
integer res = Abs(element1 - element2);
Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.656  UADALP

Add pairs of adjacent unsigned integer values and accumulate the results into the overlapping double-width elements of the destination vector.

```
| 31 30 29 28| 27 26 25 24| 23 22 21 20| 19 18 17 16| 15 14 13 12 | 10 9  | 5 4 | 0 |
| 0 1 0 0 0 0 1 0 0 | 0 0 0 1 0 1 | 1 1 0 1 | Pg | Zn | Zda |
```

**Encoding**

UADALP <Zda>, <T>, <Pg>/M, <Zn>, <Tb>

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer da = UInt(Zda);

**Assembler symbols**

- `<Zda>` Is the name of the second source and destination scalable vector register, encoded in the "Zda" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11
  - The encoding size = 00 is reserved.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 01
  - H when size = 10
  - S when size = 11
  - The encoding size = 00 is reserved.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand_acc = Z[da, VL];
bits(VL) operand_src = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '0' then
    Elem[result, e, esize] = Elem[operand_acc, e, esize];
```


else
    integer element1 = UInt(Elem[operand_src, 2*e + 0, esize DIV 2]);
    integer element2 = UInt(Elem[operand_src, 2*e + 1, esize DIV 2]);
    bits(esize) sum = (element1 + element2)<<esize-1:0>;
    Elem[result, e, esize] = Elem[operand_acc, e, esize] + sum;

Z[da, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.657  UADDLB

Add the corresponding even-numbered unsigned elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

[31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 13 12 11 10 9 | 5 4 | 0 ]
0 1 0 0 0 1 0 1 size 0 Zm 0 0 0 0 1 0 Zn Zd

Encoding
UADDLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 0;
boolean unsigned = TRUE;

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  H when size = 01
  S when size = 10
  D when size = 11
The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 01
  H when size = 10
  S when size = 11
The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
integer res = element1 + element2;
Elem[result, e, esize] = res<esize-1:0>;
Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.658  UADDLT

Add the corresponding odd-numbered unsigned elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

UADDLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Encoding

UADDLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 1;
boolean unsigned = TRUE;

Assembler symbols

<Zd>       Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>        Is the size specifier, encoded in the "size" field. It can have the following values:
            H   when size = 01
            S   when size = 10
            D   when size = 11
            The encoding size = 00 is reserved.
<Zn>       Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb>       Is the size specifier, encoded in the "size" field. It can have the following values:
            B   when size = 01
            H   when size = 10
            S   when size = 11
            The encoding size = 00 is reserved.
<Zm>       Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
integer res = element1 + element2;
Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.659   UADDV

Unsigned add horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Narrow elements are first zero-extended to 64 bits. Inactive elements in the source vector are treated as zero.

Encoding

UADDV <Dd>, <Pg>, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler symbols

<Dd> Is the 64-bit name of the destination SIMD&FP register, encoded in the "Vd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
integer sum = 0;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer element = UInt(Elem[operand, e, esize]);
    sum = sum + element;
V[d, 64] = sum<63:0>;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.
C8.2.660 UADDWB

Add the even-numbered unsigned elements of the second source vector to the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

Encoding

UADDWB <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);

\[
\text{Elem[result, e, esize]} = (\text{element1 + element2})_{\text{esize-1:0}};
\]

\[
\text{Z[d, VL]} = \text{result};
\]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
### C8.2.661 UADDWT

Add the odd-numbered unsigned elements of the second source vector to the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

![Encoding Diagram]

#### Encoding

UADDWT `<Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>

#### Decode for this encoding

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

#### Assembler symbols

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>`: Is the size specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11
  - The encoding size = 00 is reserved.
- `<Zn>`: Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>`: Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Tb>`: Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 01
  - H when size = 10
  - S when size = 11
  - The encoding size = 00 is reserved.

#### Operation

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
```

---

[01000101]

![Table]

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Zn</td>
<td>Zm</td>
<td>Zd</td>
<td>S</td>
<td>U</td>
<td>T</td>
</tr>
</tbody>
</table>
Elem[result, e, esize] = (element1 + element2)<size-1:0>;
Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.662  UCLAMP

Clamp each unsigned element in the destination vector to between the unsigned minimum value in the corresponding element of the first source vector and the unsigned maximum value in the corresponding element of the second source vector and destructively write the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2

(_FEAT_SME)

Encoding

UCLAMP <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[d, VL];
bits(VL) result;
for e = 0 to elements-1
integer element1 = UInt(Elem[operand1, e, esize]);
integer element2 = UInt(Elem[operand2, e, esize]);
integer element3 = UInt(Elem[operand3, e, esize]);
integer res = Min(Max(element1, element3), element2);
Elem[result, e, esize] = res<esize-1:0>;
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.663  UCVTF

Convert to floating-point from the unsigned integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

16-bit to half-precision

Encoding
UCVTF <Zd>.H, <Pg>/M, <Zn>.H

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);

32-bit to half-precision

Encoding
UCVTF <Zd>.H, <Pg>/M, <Zn>.S

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
### 32-bit to single-precision

![Encoding](UCVTF <Zd>.S, <Pg>/M, <Zn>.S)

**Decode for this encoding**

```plaintext
if !HaveSVE() || !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

### 32-bit to double-precision

![Encoding](UCVTF <Zd>.D, <Pg>/M, <Zn>.S)

**Decode for this encoding**

```plaintext
if !HaveSVE() || !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

### 64-bit to half-precision

![Encoding](UCVTF <Zd>.H, <Pg>/M, <Zn>.D)
Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);

64-bit to single-precision

Encoding
UCVTF <Zd>.S, <Pg>/M, <Zn>.D

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);

64-bit to double-precision

Encoding
UCVTF <Zd>.D, <Pg>/M, <Zn>.D

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    bits(d_esize) fpval = FixedToFP(element<s_esize-1:0>, 0, unsigned, FPCR[], rounding, d_esize);
    Elem[result, e, esize] = ZeroExtend(fpval, esize);
Z[d, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.664  UDIV

Unsigned divide active elements of the first source vector by corresponding elements of the second source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

UDIV <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

Assembler symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>   Is the size specifier, encoded in the "size<0>" field. It can have the following values:
  S      when size<0> = 0
  D      when size<0> = 1
<Pg>   Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>   Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL); bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  if ElementP[mask, e] == '1' then
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    integer quotient;
    if element2 == 0 then
      quotient = 0;
    else
      quotient = RoundTowardsZero(Real(element1) / Real(element2));
    end
    Elem[result, e, esize] = quotient-esize-1:0;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
$Z[dn, VL] = result$;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.665  UDIVR

Unsigned reversed divide active elements of the second source vector by corresponding elements of the first source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

UDIVR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

Assembler symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>  Is the size specifier, encoded in the "size<0>" field. It can have the following values:
S  when size<0> = 0
D  when size<0> = 1
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if Elem[mask, e, esize] == '1' then
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer quotient;
        if element1 == 0 then
            quotient = 0;
        else
            quotient = RoundTowardsZero(Real(element2) / Real(element1));
        Elem[result, e, esize] = quotient-esize-1:0;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.666 UDOT (indexed)

The unsigned integer indexed dot product instruction computes the dot product of a group of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four unsigned 8-bit or 16-bit integer values in an indexed 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group. This instruction is unpredicated.

32-bit

[31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 14 13 12 11 10 9 | 5 4 | 0]

0 1 0 0 0 1 0 0 1 0 1 i2 Zm 0 0 0 0 0 1 Zn Zda

size<1>
size<0>

Encoding

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

64-bit

[31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9 | 5 4 | 0]

0 1 0 0 0 1 0 0 1 1 1 i1 Zm 0 0 0 0 0 1 Zn Zda

size<1>
size<0>

Encoding

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
Is the name of the first source scalable vector register, encoded in the "Zn" field.

For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

For the 32-bit variant: is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.
For the 64-bit variant: is the immediate index of a 64-bit group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the "i1" field.

**Operation for all encodings**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;
Z[da, VL] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.667  UDOT (vectors)

The unsigned integer dot product instruction computes the dot product of a group of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four unsigned 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

This instruction is unpredicated.

```
[31 30 29 28|27 26 25 24|23 22 21 20]  16|15 14 13 12|11 10 9  |  5 4 |  0  
 0 1 0 0 0 1 0 0 |size 0 | Zm   0 0 0 0 0 1 | Zn   Zda
```

Encoding

UDOT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in the "size<0>" field. It can have the following values:

- S when size<0> = 0
- D when size<0> = 1

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size<0>" field. It can have the following values:

- B when size<0> = 0
- H when size<0> = 1

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) res = Elem[operand3, e, esize];
  for i = 0 to 3
    integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
    integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
```

01000100 0
0x0 size 0
0x0 Zm
0x0 Zn
0x0 Zda
\[ \text{res} = \text{res} + \text{element1} \times \text{element2}; \]
\[ \text{Elem}[\text{result}, e, \text{esize}] = \text{res}; \]
\[ Z[\text{da}, \text{VL}] = \text{result}; \]

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.668 UHADD

Add active unsigned elements of the first source vector to corresponding unsigned elements of the second source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

UHADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  integer element2 = UInt(Elem[operand2, e, esize]);
  if Elem[mask, e, esize] == '1' then
    integer res = (element1 + element2) >> 1;
    Elem[result, e, esize] = res<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.669 UHSUB

Subtract active unsigned elements of the second source vector from corresponding unsigned elements of the first source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

UHSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  integer element2 = UInt(Elem[operand2, e, esize]);
  if Elem[mask, e, esize] == '1' then
    integer res = (element1 - element2) >> 1;
    Elem[result, e, esize] = res<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.670   UHSUBR

Subtract active unsigned elements of the first source vector from corresponding unsigned elements of the second source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10  9</th>
<th>5  4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Pg</td>
<td>Zm</td>
<td>Zdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

UHSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>   Is the size specifier, encoded in the "size" field. It can have the following values:
    B when size = 00
    H when size = 01
    S when size = 10
    D when size = 11

<Pg>   Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>   Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
buts(VL) result;
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  integer element2 = UInt(Elem[operand2, e, esize]);
  if Elem[mask, e, esize] == '1' then
    integer res = (element2 - element1) >> 1;
    Elem[result, e, esize] = res<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.671 UMAX (immediate)

Determine the unsigned maximum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is an unsigned 8-bit value in the range 0 to 255, inclusive. This instruction is unpredicated.

Encoding

UMAX <Zdn>.<T>, <Zdn>.<T>, #<imm>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;
integer imm = Int(imm8, unsigned);

Assembler symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
   B when size = 00
   H when size = 01
   S when size = 10
   D when size = 11
<imm> Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    Elem[result, e, esize] = Max(element1, imm)<esize-1:0>;
Z[dn, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
The response of this instruction to asynchronous exceptions does not vary based on:

— The values of the data supplied in any of its registers.
— The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.672 **UMAX (vectors)**

Determine the unsigned maximum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

![Encoding](image)

**Encoding**

UMAX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

**Assembler symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- **B** when size = 00
- **H** when size = 01
- **S** when size = 10
- **D** when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  integer element2 = Int(Elem[operand2, e, esize], unsigned);
  if Elem[mask, e, esize] == '1' then
    integer maximum = Max(element1, element2);
    Elem[result, e, esize] = maximum<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
### C8.2.673 UMAXP

Compute the maximum value of each pair of adjacent unsigned integer elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>size 0 1 0 1 0 1 0 1</td>
<td>Pg</td>
<td>Zm</td>
<td>Zdn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

#### Encoding

UMAXP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

#### Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

#### Assembler symbols

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

#### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer element1;
integer element2;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '0' then
    Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    if IsEven(e) then
      element1 = UInt(Elem[operand1, e + 0, esize]);
      element2 = UInt(Elem[operand1, e + 1, esize]);
    else
      element1 = UInt(Elem[operand2, e - 1, esize]);
```


element2 = UInt(Elem[operand2, e + 0, esize]);
integer res = Max(element1, element2);
Elem[result, e, esize] = res<esize-1:0>;

Z[dn, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.674   UMAXV

Unsigned maximum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.

Encoding

UMAXV <V><d>, <Pg>, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = TRUE;

Assembler symbols

<V>    Is a width specifier, encoded in the "size" field. It can have the following values:
   B      when size = 00
   H      when size = 01
   S      when size = 10
   D      when size = 11

<d>    Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg>   Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn>   Is the name of the source scalable vector register, encoded in the "Zn" field.

<T>    Is the size specifier, encoded in the "size" field. It can have the following values:
   B      when size = 00
   H      when size = 01
   S      when size = 10
   D      when size = 11

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
integer maximum = if unsigned then 0 else -(2^(esize-1));
for e = 0 to elements-1
   if Elem[mask, e, esize] == '1' then
      integer element = Int(Elem[operand, e, esize], unsigned);
      maximum = Max(maximum, element);
\[ v[d, esize] = \text{maximum}<\text{esize-1:0}>; \]

**Operational information**

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C8.2.675  UMIN (immediate)

Determine the unsigned minimum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is an unsigned 8-bit value in the range 0 to 255, inclusive. This instruction is unpredicated.

Encoding

UMIN <Zdn>.<T>, <Zdn>.<T>, #<imm>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;
integer imm = Int(imm8, unsigned);

Assembler symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11
<imm> Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  Elem[result, e, esize] = Min(element1, imm)<esize-1:0>;
Z[dn, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
The response of this instruction to asynchronous exceptions does not vary based on:

— The values of the data supplied in any of its registers.

— The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.676   UMIN (vectors)

Determine the unsigned minimum of active elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

UMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

Assembler symbols

<Zdn>     Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>       Is the size specifier, encoded in the "size" field. It can have the following values:
            B    when size = 00
            H    when size = 01
            S    when size = 10
            D    when size = 11

<Pg>      Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>      Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        if Elem[mask, e, esize] == '1' then
                integer minimum = Min(element1, element2);
                Elem[result, e, esize] = minimum<esize-1:0>;
        else
                Elem[result, e, esize] = Elem[operand1, e, esize];
$Z[dn, VL] = \text{result;}$

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.677  UMINP

Compute the minimum value of each pair of adjacent unsigned integer elements within each source vector, and
interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first
source vector.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16 15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>size 0 1 0 1 1 1 0 1</td>
<td>Pg</td>
<td>Zm</td>
<td>Zdn</td>
</tr>
</tbody>
</table>

Encoding

UMINP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer element1;
integer element2;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '0' then
        Elem[result, e, esize] = Elem[operand1, e, esize];
    else
        if IsEven(e) then
            element1 = UInt(Elem[operand1, e + 0, esize]);
            element2 = UInt(Elem[operand1, e + 1, esize]);
        else
            element1 = UInt(Elem[operand2, e - 1, esize]);
element2 = UInt(Elem[operand2, e + 0, esize]);
integer res = Min(element1, element2);
Elem[result, e, esize] = res<esize-1:0>;
Z[dn, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.678  UMINV

Unsigned minimum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the maximum unsigned integer for the element size.

Encoding

UMINV <V><db>, <Pg>, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = TRUE;

Assembler symbols

<V>  Is a width specifier, encoded in the "size" field. It can have the following values:
    B     when size = 00
    H     when size = 01
    S     when size = 10
    D     when size = 11
<db> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
-Za> Is the name of the source scalable vector register, encoded in the "Zn" field.
<T>  Is the size specifier, encoded in the "size" field. It can have the following values:
    B     when size = 00
    H     when size = 01
    S     when size = 10
    D     when size = 11

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
integer minimum = if unsigned then (2^esize - 1) else (2^(esize-1) - 1);
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer element = Int(Elem[operand, e, esize], unsigned);
        minimum = Min(minimum, element);
V[d, esize] = minimum<esize-1:0>;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
Multiply the even-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment and destructively add to the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

32-bit

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 14 13 12 11 10 9]</th>
<th>5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 0 1</td>
<td>i3h</td>
</tr>
</tbody>
</table>

**Encoding**


**Decode for this encoding**

```java
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

64-bit

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9]</th>
<th>5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 0 1</td>
<td>Zm</td>
</tr>
</tbody>
</table>

**Encoding**

UMLALB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

**Decode for this encoding**

```java
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```
Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bis(VL) operand1 = Z[n, VL];
bis(VL) operand2 = Z[m, VL];
bis(VL) result = Z[da, VL];
for e = 0 to elements-1
  integer s = e - (e MOD eltspersegment);
  integer element1 = UInt(Elem[operand1, 2 * e + sel, esize]);
  integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
  bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
  Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + product;
Z[da, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.680 UMLALB (vectors)

Multiply the corresponding even-numbered unsigned elements of the first and second source vectors and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

Encoding

UMLALB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  H when size = 01
  S when size = 10
  D when size = 11
The encoding size = 00 is reserved.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 01
  H when size = 10
  S when size = 11
The encoding size = 00 is reserved.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
  integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
  bits(esize) product = (element1 * element2)<esize-1:0>;
  Elem[result, e, esize] = Elem[result, e, esize] + product;
\[ Z[da, VL] = \text{result}; \]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.681  UMLALT (indexed)

Multiply the odd-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment and destructively add to the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

32-bit

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
custom integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;

64-bit

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
custom integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
  integer s = e - (e MOD eltspersegment);
  integer element1 = UInt(Elem[operand1, 2 * e + sel, esize]);
  integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
  bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
  Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + product;
Z[da, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.682 UMLALT (vectors)

Multiply the corresponding odd-numbered unsigned elements of the first and second source vectors and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
0 1 0 0 0 1 0 0 | size 0 | Zm 0 1 0 | 0 1 | Zn | Zda
```

**Encoding**

UMLALT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

**Decode for this encoding**

```cpp
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

**Assembler symbols**

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11
  The encoding size = 00 is reserved.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 01
  - H when size = 10
  - S when size = 11
  The encoding size = 00 is reserved.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```cpp
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
  integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
  bits(esize) product = (element1 * element2)<esize-1:0>;
  Elem[result, e, esize] = Elem[result, e, esize] + product;
```
Z[da, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is **UNPREDICTABLE**:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.683 UMLSLB (indexed)

Multiply the even-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment and destructively subtract from the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 0 0 1 3</td>
<td>Zm 1 0 1</td>
<td>1</td>
</tr>
</tbody>
</table>

Encoding


Decode for this encoding

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

64-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 0 0 1 1</td>
<td>Zm 1 0 1</td>
<td>1</td>
</tr>
</tbody>
</table>

Encoding

UMLSLB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

Decode for this encoding

```c
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```
Assembler symbols

<Zda>
Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn>
Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>
For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm>
For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
  integer s = e - (e MOD eltspersegment);
  integer element1 = UInt(Elem[operand1, 2 * e + sel, esize]);
  integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
  bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
  Elem[result, e, 2*esize] = Elem[result, e, 2*esize] - product;

Z[da, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.684 UMLSLB (vectors)

Multiply the corresponding even-numbered unsigned elements of the first and second source vectors and destructively subtract from the overlapping double-width elements of the addend vector. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 13 12 11 10 9 | 5 4 | 0 |
0 1 0 0 0 1 0 0 size 0 Zm 0 1 0 1 | 0 Zn Zda 00 00
```

**Encoding**

UMLSLB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

**Decode for this encoding**

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

**Assembler symbols**

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 01
- H when size = 10
- S when size = 11

The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
  integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
```


\[
\text{bits}(\text{thesize}) \, \text{product} = (\text{element}1 \times \text{element}2)\langle\text{thesize}-1:0\rangle; \\
\text{Elem}[\text{result}, \, e, \, \text{thesize}] = \text{Elem}[\text{result}, \, e, \, \text{thesize}] - \text{product}; \\
\]

\[
Z[\text{da}, \, \text{VL}] = \text{result};
\]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.685 UMLSLT (indexed)

Multiply the odd-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment and destructively subtract from the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

32-bit

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;

64-bit

Encoding


Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
**Assembler symbols**

<Zda>
Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn>
Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>
For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm>
For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

**Operation for all encodings**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements-1
  integer s = e - (e MOD eltspersegment);
  integer element1 = UInt(Elem[operand1, 2 * e + sel, esize]);
  integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
  bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
  Elem[result, e, 2*esize] = Elem[result, e, 2*esize] - product;
Z[da, VL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.686 UMLS LT (vectors)

Multiply the corresponding odd-numbered unsigned elements of the first and second source vectors and destructively subtract from the overlapping double-width elements of the addend vector. This instruction is unpredicated.

| 31 30 29 28|27 26 25 24|23 22 21 20| 16|15 14 13|12|11 10 9 | 5 4 | 0 |
| 0 1 0 0 0 1 0 0 | 0 | 1 | 0 | 1 | 1 | 1 |

Encoding

UMLS LT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
n = UInt(Zn);
m = UInt(Zm);
da = UInt(Zda);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];
for e = 0 to elements - 1
t1 =(UInt(Elem[operand1, 2*e + 1, esize DIV 2]));
t2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
result[e] = t1 - t2;
bits(esize) product = (element1 * element2)<esize-1:0>;
Elem[result, e, esize] = Elem[result, e, esize] - product;
Z[da, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.687 UMMLA

The unsigned integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of unsigned 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of unsigned 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE

(FEAT_I8MM)

Encoding


Decode for this encoding

if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result = Zeros(VL);
bits(128) op1, op2;
bits(128) res, addend;

for s = 0 to segments-1
    op1 = Elem[operand1, s, 128];
    op2 = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
res = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
Elem[result, s, 128] = res;

Z[da, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.688  UMULH (predicated)

Widening multiply unsigned integer values in active elements of the first source vector by corresponding elements of the second source vector and destructively place the high half of the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

UMULH <Zdn>.,<T>, <Pg>/M, <Zdn>.,<T>, <Zm>.,<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

Assembler symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

B  when size = 00
H  when size = 01
S  when size = 10
D  when size = 11

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if Elem[mask, e, esize] == '1' then
        integer product = (element1 * element2) >> esize;
        Elem[result, e, esize] = product<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.689  UMULH (unpredicated)

Widening multiply unsigned integer values of all elements of the first source vector by corresponding elements of
the second source vector and place the high half of the result in the corresponding elements of the destination vector.
This instruction is unpredicated.

Encoding

UMULH <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
n = UInt(Zn);
m = UInt(Zm);
d = UInt(Zd);
boolean unsigned = TRUE;

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>   Is the size specifier, encoded in the "size" field. It can have the following values:
     B   when size = 00
     H   when size = 01
     S   when size = 10
     D   when size = 11
<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
   integer element1 = Int(Elem[operand1, e, esize], unsigned);
   integer element2 = Int(Elem[operand2, e, esize], unsigned);
   integer product = (element1 * element2) >> esize;
   Elem[result, e, esize] = product<esize-1:0>;
Z[d, VL] = result;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.690 UMULLB (indexed)

Multiply the even-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment, and place the results in the overlapping double-width elements of the destination vector register.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

32-bit

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9  8</th>
<th>7  6  5  4</th>
<th>3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>1 0 1 i3h</td>
<td>Zm</td>
<td>1 1 0 1</td>
<td>i3l</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
</tr>
</tbody>
</table>
```

Encoding


Decode for this encoding

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
c=integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;
```

64-bit

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9  8</th>
<th>7  6  5  4</th>
<th>3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>1 1 1</td>
<td>Zm</td>
<td>1 1 0 1</td>
<td>i2l</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
</tr>
</tbody>
</table>
```

Encoding

UMULLB <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

Decode for this encoding

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
c=integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;
```
Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation for all encodings

CheckSVEEnabled();
class constant integer VL = CurrentVL;
class constant integer PL = VL DIV 8;
class constant integer elements = VL DIV (2 * esize);
class constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
  integer s = e - (e MOD eltspersegment);
  integer element1 = UInt(Elem[operand1, 2 * e + sel, esize]);
  integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
  integer res = element1 * element2;
  Elem[result, e, 2*esize] = res<2*esize-1:0>;

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.691 UMULLB (vectors)

Multiply the corresponding even-numbered unsigned elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

Encoding

UMULLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVEz() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
    H when size = 01
    S when size = 10
    D when size = 11
    The encoding size = 00 is reserved.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
    B when size = 01
    H when size = 10
    S when size = 11
    The encoding size = 00 is reserved.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    integer res = element1 * element2;
    Elem[result, e, esize] = res<esize-1:0>;

\[ Z[d, \text{VL}] = \text{result}; \]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.692  UMULLT (indexed)

Multiply the odd-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment, and place the results in the overlapping double-width elements of the destination vector register.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

32-bit

\[
\begin{array}{cccccc}
\mid 31 & 30 & 29 & 28 \mid 27 & 26 & 25 & 24 \mid 23 & 22 & 21 & 20 \mid 19 & 18 & 16 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 5 & 4 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & i3h & Zm & 1 & 1 & 0 & 1 & i3l & 1 & Zn & Zd \\
\end{array}
\]

size<1>

size<0>

Encoding


Decode for this encoding

\[
\text{if !HaveSVE2() \&\& !HaveSME() then UNDEFINED;}
\]

\[
\text{constant integer esize = 16;}
\]

\[
\text{integer index = UInt(i3h:i3l);} \\
\text{integer n = UInt(Zn);} \\
\text{integer m = UInt(Zm);} \\
\text{integer d = UInt(Zd);} \\
\text{integer sel = 1;} \\
\]

64-bit

\[
\begin{array}{cccccc}
\mid 31 & 30 & 29 & 28 \mid 27 & 26 & 25 & 24 \mid 23 & 22 & 21 & 20 \mid 19 & 18 & 16 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 5 & 4 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & \mid Zm & 1 & 1 & 0 & 1 & \mid i2l & 1 & Zn & Zd \\
\end{array}
\]

size<1>

size<0>
i2h

Encoding

UMULLT <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

Decode for this encoding

\[
\text{if !HaveSVE2() \&\& !HaveSME() then UNDEFINED;}
\]

\[
\text{constant integer esize = 32;}
\]

\[
\text{integer index = UInt(i2h:i2l);} \\
\text{integer n = UInt(Zn);} \\
\text{integer m = UInt(Zm);} \\
\text{integer d = UInt(Zd);} \\
\text{integer sel = 1;} \\
\]
Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>  For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
      For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
      For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation for all encodings

    CheckSVEEnabled();
    constant integer VL = CurrentVL;
    constant integer PL = VL DIV 8;
    constant integer elements = VL DIV (2 * esize);
    constant integer eltspersegment = 128 DIV (2 * esize);
    bits(VL) operand1 = Z[n, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) result;
    for e = 0 to elements-1
        integer s = e - (e MOD eltspersegment);
        integer element1 = UInt(Elem[operand1, 2 * e + sel, esize]);
        integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
        integer res = element1 * element2;
        Elem[result, e, 2*esize] = res<2*esize-1:0>;
    Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.693  UMULLT (vectors)

Multiply the corresponding odd-numbered unsigned elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

Encoding

UMULLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    integer res = element1 * element2;
    Elem[result, e, esize] = res<esize-1:0>;
Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.694 **UQADD (immediate)**

Unsigned saturating add of an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to \((2^N)-1\). This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

---

**Encoding**

UQADD <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

**Decode for this encoding**

```c
if !HaveSVE() & !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = TRUE;
```

**Assembler symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<imm>` Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:
  - LSL #0 when sh = 0
  - LSL #8 when sh = 1

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
```
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element1 + imm, esize, unsigned);

Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.695   UQADD (vectors, predicated)

Add active unsigned elements of the first source vector to corresponding unsigned elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.

**Encoding**

UQADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

**Assembler symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[<Pg>, PL];
bits(VL) operand1 = Z[<Zdn>, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[<Zm>, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
integer element1 = UInt(Elem[operand1, e, esize]);
integer element2 = UInt(Elem[operand2, e, esize]);
if Elem[mask, e, esize] == '1' then
integer res = UInt(Sat(element1 + element2, esize, unsigned));
Elem[result, e, esize] = res<esize-1:0>;
else
Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.696   UQADD (vectors, unpredicated)

Unsigned saturating add all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to \((2^N)-1\). This instruction is unpredicated.

Encoding

UQADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = TRUE;
```

Assembler symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + element2, esize, unsigned);

Z[d, VL] = result;
```
C8.2.697   UQDECB

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

32-bit

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>0 0 1 0</td>
<td>imm4</td>
<td>1 1 1 1</td>
<td>1 1</td>
<td>pattern</td>
<td>Rdn</td>
<td></td>
</tr>
</tbody>
</table>

size<1> sf D U
size<0> 

Encoding

UQDECB <Wdn>{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>0 0 1 1</td>
<td>imm4</td>
<td>1 1 1 1</td>
<td>1 1</td>
<td>pattern</td>
<td>Rdn</td>
<td></td>
</tr>
</tbody>
</table>

size<1> sf D U
size<0> 

Encoding

UQDECB <Xdn>{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler symbols

<\(\text{idn}\)> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<\(\text{idn}\)> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:
- **POW2** when pattern = 00000
- **VL1** when pattern = 00001
- **VL2** when pattern = 00010
- **VL3** when pattern = 00011
- **VL4** when pattern = 00100
- **VL5** when pattern = 00101
- **VL6** when pattern = 00110
- **VL7** when pattern = 00111
- **VL8** when pattern = 01000
- **VL16** when pattern = 01001
- **VL32** when pattern = 01010
- **VL64** when pattern = 01011
- **VL128** when pattern = 01100
- **VL256** when pattern = 01101
- **#uimm5** when pattern = 0111x
- **#uimm5** when pattern = 101x1
- **#uimm5** when pattern = 10110
- **#uimm5** when pattern = 1x0x1
- **#uimm5** when pattern = 1x010
- **#uimm5** when pattern = 1xx00
- **MUL4** when pattern = 11101
- **MUL3** when pattern = 11110
- **ALL** when pattern = 11111

<\(\text{imm}\)> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation for all encodings

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
C8.2.698   UQDECD (scalar)

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

**32-bit**

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9 | 5 4 | 0]
0 0 0 0 0 1 0 0 1 1 1 0 imm4 1 1 1 1 1 1 pattern Rdn
```

```
sf D U
size<1>
size<0>
```

**Encoding**

UQDECD <Wdn>{, <pattern>{, MUL #<imm>}}

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bites(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;
```

**64-bit**

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9 | 5 4 | 0]
0 0 0 0 0 1 0 0 1 1 1 1 imm4 1 1 1 1 1 1 pattern Rdn
```

```
sf D U
size<1>
size<0>
```

**Encoding**

UQDECD <Xdn>{, <pattern>{, MUL #<imm>}}

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bites(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```
boolean unsigned = TRUE;
integer ssize = 64;

**Assembler symbols**

- `<idn>`  
  Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

- `<Xdn>`  
  Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

- `<pattern>`  
  Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:
  
  - `POW2` when `pattern = 00000`
  - `VL1` when `pattern = 00001`
  - `VL2` when `pattern = 00010`
  - `VL3` when `pattern = 00011`
  - `VL4` when `pattern = 00100`
  - `VL5` when `pattern = 00101`
  - `VL6` when `pattern = 00110`
  - `VL7` when `pattern = 00111`
  - `VL8` when `pattern = 01000`
  - `VL16` when `pattern = 01001`
  - `VL256` when `pattern = 01010`
  - `VL64` when `pattern = 01011`
  - `VL128` when `pattern = 01100`
  - `VL256` when `pattern = 01101`
  - `#uimm5` when `pattern = 0111x`
  - `#uimm5` when `pattern = 101x1`
  - `#uimm5` when `pattern = 10110`
  - `#uimm5` when `pattern = 1x0x1`
  - `#uimm5` when `pattern = 1x010`
  - `#uimm5` when `pattern = 1xx00`
  - `MUL4` when `pattern = 11101`
  - `MUL3` when `pattern = 11110`
  - `ALL` when `pattern = 11111`

- `<imm>`  
  Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation for all encodings**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```
C8.2.699  UQDECD (vector)

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 64-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

Encoding

UQDECD <Zdn>.D{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;

Assembler symbols

<Zdn>  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern>  Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>POW2</td>
<td>00000</td>
</tr>
<tr>
<td>VL1</td>
<td>00001</td>
</tr>
<tr>
<td>VL2</td>
<td>00010</td>
</tr>
<tr>
<td>VL3</td>
<td>00011</td>
</tr>
<tr>
<td>VL4</td>
<td>00100</td>
</tr>
<tr>
<td>VL5</td>
<td>00101</td>
</tr>
<tr>
<td>VL6</td>
<td>00110</td>
</tr>
<tr>
<td>VL7</td>
<td>00111</td>
</tr>
<tr>
<td>VL8</td>
<td>01000</td>
</tr>
<tr>
<td>VL16</td>
<td>01001</td>
</tr>
</tbody>
</table>
VL32 when pattern = 01010
VL64 when pattern = 01011
VL128 when pattern = 01100
VL256 when pattern = 01101
#uimm5 when pattern = 0111x
#uimm5 when pattern = 101x1
#uimm5 when pattern = 10110
#uimm5 when pattern = 1x0x1
#uimm5 when pattern = 1x010
#uimm5 when pattern = 1xx00
MUL4 when pattern = 11101
MUL3 when pattern = 11110
ALL when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.700   UQDECH (scalar)

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register’s unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

32-bit

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 0 1 1 0</td>
<td>imm4</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

size<1> sf D U
size<0>
```

**Encoding**

UQDECH <\$dn>{, <pattern>{, MUL #<imm>}}}

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 0 1 1 1</td>
<td>imm4</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

size<1> sf D U
size<0>
```

**Encoding**

UQDECH <\$dn>{, <pattern>{, MUL #<imm>}}}

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

### Assembler symbols

- `<idn>` Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- `<Xdn>` Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- `<pattern>` Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:
  - `POW2` when pattern = 00000
  - `VL1` when pattern = 00001
  - `VL2` when pattern = 00010
  - `VL3` when pattern = 00011
  - `VL4` when pattern = 00100
  - `VL5` when pattern = 00101
  - `VL6` when pattern = 00110
  - `VL7` when pattern = 00111
  - `VL8` when pattern = 01000
  - `VL16` when pattern = 01001
  - `VL32` when pattern = 01010
  - `VL64` when pattern = 01011
  - `VL128` when pattern = 01100
  - `VL256` when pattern = 01101
  - `VL512` when pattern = 10000
  - `VL1024` when pattern = 10001
  - `VL2048` when pattern = 10010
  - `VL4096` when pattern = 10011
  - `VL8192` when pattern = 10100
  - `VL16384` when pattern = 10101
  - `VL32768` when pattern = 10110
  - `VL65536` when pattern = 10111
  - `#uimm5` when pattern = 0111x
  - `#uimm5` when pattern = 1011x
  - `#uimm5` when pattern = 10110
  - `#uimm5` when pattern = 10101
  - `#uimm5` when pattern = 10100
  - `#uimm5` when pattern = 10101
  - `#uimm5` when pattern = 1x00
  - `#uimm5` when pattern = 1x01
  - `#uimm5` when pattern = 1xx0
  - `#uimm5` when pattern = 1xx1
  - `MUL4` when pattern = 11101
  - `MUL3` when pattern = 11110
  - `ALL` when pattern = 11111

- `<imm>` Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

### Operation for all encodings

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```
C8.2.701 UQDECH (vector)

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 16-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9 5 4 0]
0 0 0 0 0 1 0 0 0 1 1 0 1 | 1 1 0 0 | 1 1 | pattern | Zdn
size<1>
size<0>
```

**Encoding**

UQDECH <Zdn>.H{, <pattern>{, MUL #<imm>}}

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
```

**Assembler symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<pattern>` Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:
  - `POW2` when pattern = 00000
  - `VL1` when pattern = 00001
  - `VL2` when pattern = 00010
  - `VL3` when pattern = 00011
  - `VL4` when pattern = 00100
  - `VL5` when pattern = 00101
  - `VL6` when pattern = 00110
  - `VL7` when pattern = 00111
  - `VL8` when pattern = 01000
  - `VL16` when pattern = 01001
VL32 when pattern = 01010
VL64 when pattern = 01011
VL128 when pattern = 01100
VL256 when pattern = 01101
#uimm5 when pattern = 0111x
#uimm5 when pattern = 101x1
#uimm5 when pattern = 10110
#uimm5 when pattern = 1x0x1
#uimm5 when pattern = 1x010
#uimm5 when pattern = 1xx00
MUL4 when pattern = 11101
MUL3 when pattern = 11110
ALL when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);
Z[dn, VL] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.702 UQDECP (scalar)

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

32-bit

\[
\begin{array}{cccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & \text{Pm} & \text{Rdn} \\
\end{array}
\]

Encoding

UQDECP <Wdn>, <Pm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

\[
\begin{array}{cccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & \text{Pm} & \text{Rdn} \\
\end{array}
\]

Encoding

UQDECP <Xdn>, <Pm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
integer ssize = 64;

Assembler symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11
Operation for all encodings

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn, ssize];
bits(PL) operand2 = P[m, PL];
bits(ssize) result;
integer count = 0;
for e = 0 to elements-1
   if ElemP[operand2, e, esize] == '1' then
      count = count + 1;
   integer element = Int(operand1, unsigned);
   (result, -) = SatQ(element - count, ssize, unsigned);
   X[dn, 64] = Extend(result, 64, unsigned);

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is
dependent on the general-purpose register written by this instruction might be significantly delayed.
### C8.2.703 UQDECP (vector)

Counts the number of true elements in the source predicate and then uses the result to decrement all destination vector elements. The results are saturated to the element unsigned integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8 | 5 4 | 0 ]
0 0 1 0 0 1 0 1 | size 1 0 1 0 1 | 1 0 0 0 0 0 0 0 | Pm | Zdn
```

#### Encoding

UQDECP <Zdn>.<T>, <Pm>.<T>

#### Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = $ << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;
```

#### Assembler symbols

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - `H` when size = 01
  - `S` when size = 10
  - `D` when size = 11
  - The encoding size = 00 is reserved.
- `<Pm>` Is the name of the source scalable predicate register, encoded in the "Pm" field.

#### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;
for e = 0 to elements-1
  integer element = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element - count, esize, unsigned);
Z[dn, VL] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.704  UQDECW (scalar)

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

32-bit

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16 15 14 13 12|11 10 9 | 5 4 | 0 |
| 0 0 0 0 0 1 0 0 | 1 0 1 0 | imm4 | 1 1 1 1 | 1 1 | pattern | Rdn |
```

Encoding

```
UQDECW <Wdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;
```

64-bit

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16 15 14 13 12|11 10 9 | 5 4 | 0 |
| 0 0 0 0 0 1 0 0 | 1 0 1 1 | imm4 | 1 1 1 1 | 1 1 | pattern | Rdn |
```

Encoding

```
UQDECW <Xdn>{, <pattern>{, MUL #<imm>}}
```

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```
boolean unsigned = TRUE;
integer ssize = 64;

**Assembler symbols**

<Idn>  Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Xdn>  Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern>  Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:
- **POW2** when pattern = 00000
- **VL1** when pattern = 00001
- **VL2** when pattern = 00010
- **VL3** when pattern = 00011
- **VL4** when pattern = 00100
- **VL5** when pattern = 00101
- **VL6** when pattern = 00110
- **VL7** when pattern = 00111
- **VL8** when pattern = 01000
- **VL16** when pattern = 01001
- **VL32** when pattern = 01010
- **VL64** when pattern = 01011
- **VL128** when pattern = 01100
- **VL256** when pattern = 01101
- **#uimm5** when pattern = 0111x
- **#uimm5** when pattern = 101x1
- **#uimm5** when pattern = 10110
- **#uimm5** when pattern = 1x0x1
- **#uimm5** when pattern = 1x010
- **#uimm5** when pattern = 1xx00
- **MUL4** when pattern = 11101
- **MUL3** when pattern = 11110
- **ALL** when pattern = 11111

<imm>  Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation for all encodings**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```
C8.2.705 UQDECW (vector)

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 32-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

Encoding

UQDECW <Zdn>.S{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;

Assembler symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

- POW2 when pattern = 00000
- VL1 when pattern = 00001
- VL2 when pattern = 00010
- VL3 when pattern = 00011
- VL4 when pattern = 00100
- VL5 when pattern = 00101
- VL6 when pattern = 00110
- VL7 when pattern = 00111
- VL8 when pattern = 01000
- VL16 when pattern = 01001
VL32  when pattern = 01010
VL64  when pattern = 01011
VL128 when pattern = 01100
VL256 when pattern = 01101
#uimm5 when pattern = 0111x
#uimm5 when pattern = 101x1
#uimm5 when pattern = 10110
#uimm5 when pattern = 1x0x1
#uimm5 when pattern = 1x010
#uimm5 when pattern = 1xx00
MUL4  when pattern = 11101
MUL3  when pattern = 11110
ALL   when pattern = 11111

<imm>  Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
const integer VL = CurrentVL;
const integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
   integer element1 = Int(Elem[operand1, e, esize], unsigned);
   (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);
Z[dn, VL] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.706   UQINCB

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

32-bit

```
  [31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9 5 4 0 ]
  0 0 0 0 0 1 0 0 0 1 0 imm4 1 1 1 1 0 1 pattern Rdn
```

<table>
<thead>
<tr>
<th>size&lt;1&gt;</th>
<th>sf</th>
<th>D</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th></th>
</tr>
</thead>
</table>

Encoding

UQINCB <Wdn>{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

```
  [31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9 5 4 0 ]
  0 0 0 0 0 1 0 0 0 1 1 imm4 1 1 1 1 0 1 pattern Rdn
```

<table>
<thead>
<tr>
<th>size&lt;1&gt;</th>
<th>sf</th>
<th>D</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th></th>
</tr>
</thead>
</table>

Encoding

UQINCB <Xdn>{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler symbols

<\idn> 
Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Xdn> 
Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> 
Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

- **POW2** when pattern = 00000
- **VL1** when pattern = 00001
- **VL2** when pattern = 00010
- **VL3** when pattern = 00011
- **VL4** when pattern = 00100
- **VL5** when pattern = 00101
- **VL6** when pattern = 00110
- **VL7** when pattern = 00111
- **VL8** when pattern = 01000
- **VL16** when pattern = 01001
- **VL256** when pattern = 01101
- **MUL4** when pattern = 11101
- **MUL3** when pattern = 11110
- **ALL** when pattern = 11111

<\imm> 
Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation for all encodings

CheckSVEEnabled();
icenter count = DecodePredCount(pat, esize);
bite(ssize) operand1 = X[dn, ssize];
bite(ssize) result;

center element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * \imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
C8.2.707  UQINCD (scalar)

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register’s unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

32-bit

```
| [31 30 29 28] | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 5 | 4 | 0 |
| 0 0 0 0 0 0 1 0 0 1 1 1 0 | imm4 | 1 1 1 1 | 0 | 1 | pattern | Rdn |
| size<1> | sf | D | U |
| size<0> 
```

**Encoding**

UQINCD <Wdn>{, <pattern>{, MUL #<imm>}}

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;
```

64-bit

```
| [31 30 29 28] | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 5 | 4 | 0 |
| 0 0 0 0 0 0 1 0 0 1 1 1 1 | imm4 | 1 1 1 1 | 0 | 1 | pattern | Rdn |
| size<1> | sf | D | U |
| size<0> 
```

**Encoding**

UQINCD <Xdn>{, <pattern>{, MUL #<imm>}}

**Decode for this encoding**

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```
boolean unsigned = TRUE;
integer ssize = 64;

Assembler symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

POW2 when pattern = 00000
VL1 when pattern = 00001
VL2 when pattern = 00010
VL3 when pattern = 00011
VL4 when pattern = 00100
VL5 when pattern = 00101
VL6 when pattern = 01010
VL7 when pattern = 01011
VL8 when pattern = 01100
VL16 when pattern = 01101
VL32 when pattern = 01110
VL64 when pattern = 01111
VL128 when pattern = 10000
VL256 when pattern = 10001
#uimm5 when pattern = 10010
#uimm5 when pattern = 11000
#uimm5 when pattern = 11001
#uimm5 when pattern = 11010
#uimm5 when pattern = 11011
#uimm5 when pattern = 11100
#uimm5 when pattern = 11101
#uimm5 when pattern = 11110
#uimm5 when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation for all encodings

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bites(ssize) operand1 = X[dn, ssize];
bites(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
C8.2.708  UQINCD (vector)

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 64-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

```
Encoding
UQINCD <Zdn>.D{, <pattern>{, MUL #imm}}
```

```
Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
```

```
Assembler symbols
'Zdn'  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<pattern>  Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

POW2  when pattern = 00000
VL1  when pattern = 00001
VL2  when pattern = 00010
VL3  when pattern = 00011
VL4  when pattern = 00100
VL5  when pattern = 00101
VL6  when pattern = 00110
VL7  when pattern = 00111
VL8  when pattern = 01000
VL16  when pattern = 01001
```
### VL32
- when pattern = 01010

### VL64
- when pattern = 01011

### VL128
- when pattern = 01100

### VL256
- when pattern = 01101

### #uimm5
- when pattern = 0111x
- when pattern = 101x1
- when pattern = 10110
- when pattern = 1x0x1
- when pattern = 1x010
- when pattern = 1xx00

### MUL4
- when pattern = 11101

### MUL3
- when pattern = 11110

### ALL
- when pattern = 11111

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

#### Operation

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);
Z[dn, VL] = result;
```

#### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.709   UQINCH (scalar)

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an
immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is
saturated to the general-purpose register’s unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than
Undefined Instruction exception.

32-bit

Encoding

UQINCH <Wdn>{, <pattern>{, MUL #<imm>}}</p>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

Encoding

UQINCH <Xdn>{, <pattern>{, MUL #<imm>}}</p>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler symbols

<Wdn>   Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn>   Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:
  - POW2 when pattern = 00000
  - VL1 when pattern = 00001
  - VL2 when pattern = 00010
  - VL3 when pattern = 00011
  - VL4 when pattern = 00100
  - VL5 when pattern = 00101
  - VL6 when pattern = 00110
  - VL7 when pattern = 00111
  - VL8 when pattern = 01000
  - VL16 when pattern = 01001
  - VL32 when pattern = 01010
  - VL64 when pattern = 01011
  - VL128 when pattern = 01100
  - VL256 when pattern = 01101
  - #uimm5 when pattern = 0111x
  - #uimm5 when pattern = 101x1
  - #uimm5 when pattern = 10110
  - #uimm5 when pattern = 1x0x1
  - #uimm5 when pattern = 1x010
  - #uimm5 when pattern = 1xx00
  - MUL4 when pattern = 11101
  - MUL3 when pattern = 11110
  - ALL when pattern = 11111

<iimm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation for all encodings

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * iimm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
C8.2.710   UQINCH (vector)

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an
immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The
results are saturated to the 16-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than
Undefined Instruction exception.

Encoding

UQINCH <Zdn>.H{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;

Assembler symbols

<Zdn>    Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the
following values:

POW2      when pattern = 00000
VL1       when pattern = 00001
VL2       when pattern = 00010
VL3       when pattern = 00011
VL4       when pattern = 00100
VL5       when pattern = 00101
VL6       when pattern = 00110
VL7       when pattern = 00111
VL8       when pattern = 01000
VL16      when pattern = 01001
C8.2 Alphabetical list of SVE instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>VL32</td>
<td>01010</td>
</tr>
<tr>
<td>VL64</td>
<td>01011</td>
</tr>
<tr>
<td>VL128</td>
<td>01100</td>
</tr>
<tr>
<td>VL256</td>
<td>01101</td>
</tr>
<tr>
<td>#uimm5</td>
<td>0111x</td>
</tr>
<tr>
<td>#uimm5</td>
<td>101x1</td>
</tr>
<tr>
<td>#uimm5</td>
<td>10110</td>
</tr>
<tr>
<td>#uimm5</td>
<td>1x0x1</td>
</tr>
<tr>
<td>#uimm5</td>
<td>1x010</td>
</tr>
<tr>
<td>#uimm5</td>
<td>1xx00</td>
</tr>
<tr>
<td>MUL4</td>
<td>11101</td>
</tr>
<tr>
<td>MUL3</td>
<td>11110</td>
</tr>
<tr>
<td>ALL</td>
<td>11111</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);
Z[dn, VL] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.711   UQINCP (scalar)

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

32-bit

\[
\begin{array}{ccccccccccccc}
\text{size} & \text{Pm} & \text{Rdn} \\
0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 8 & 5 & 4 & 0 \\
\end{array}
\]

Encodings

UQINCP \(<Wdn>, <Pm>, <T>\)

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

\[
\begin{array}{ccccccccccccc}
\text{size} & \text{Pm} & \text{Rdn} \\
0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 8 & 5 & 4 & 0 \\
\end{array}
\]

Encoding

UQINCP \(<Xdn>, <Pm>, <T>\)

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer m = UInt(Rdn);
boolean unsigned = TRUE;
integer ssize = 64;

Assembler symbols

\(<Wdn>\) Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

\(<Xdn>\) Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

\(<Pm>\) Is the name of the source scalable predicate register, encoded in the "Pm" field.

\(<T>\) Is the size specifier, encoded in the "size" field. It can have the following values:

<table>
<thead>
<tr>
<th>(&lt;T&gt;)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>size = 00</td>
</tr>
<tr>
<td>H</td>
<td>size = 01</td>
</tr>
<tr>
<td>S</td>
<td>size = 10</td>
</tr>
<tr>
<td>D</td>
<td>size = 11</td>
</tr>
</tbody>
</table>
Operation for all encodings

CheckSVEEnabled();
const integer VL = CurrentVL;
const integer PL = VL DIV 8;
const integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn, ssize];
bits(PL) operand2 = P[m, PL];
bits(ssize) result;
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

integer element = Int(operand1, unsigned);
(result, -) = SatQ(element + count, ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is
dependent on the general-purpose register written by this instruction might be significantly delayed.
C8.2.712 **UQINCP (vector)**

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements. The results are saturated to the element unsigned integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8 | 5 4 | 0 |
|----|----|----|----|----|----|----|----|----|
| 0 0 1 0 0 1 0 1 | size 1 0 1 0 0 1 1 0 0 0 0 0 0 | Pm | Zdn |
```

**Encoding**

UQINCP <Zdn>.<T>, <Pm>.<T>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = $ << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;

**Assembler symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - H when size = 01
  - S when size = 10
  - D when size = 11
  - The encoding size = 00 is reserved.
- `<Pm>` Is the name of the source scalable predicate register, encoded in the "Pm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;
for e = 0 to elements-1
  integer element = Int(Elem[operand1, e, esize], unsigned);
  Elem[result, e, esize], -) = SatQ(element + count, esize, unsigned);
Z[dn, VL] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.713  UQINCW (scalar)

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

32-bit

| [31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12|11 10 9 | 5 4 | 0 ] |
|-----------------|-----------------|-----------------|-----------------|
| 0 0 0 0 0 1 0 1 0 | imm4 | 1 1 1 1 0 1 | pattern | Rdn |

Encoding

UQINCW <Wdn>{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

| [31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12|11 10 9 | 5 4 | 0 ] |
|-----------------|-----------------|-----------------|-----------------|
| 0 0 0 0 0 1 0 1 0 | imm4 | 1 1 1 1 0 1 | pattern | Rdn |

Encoding

UQINCW <Xdn>{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler symbols

<\idn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

POW2 when pattern = 00000
VL1 when pattern = 00001
VL2 when pattern = 00010
VL3 when pattern = 00011
VL4 when pattern = 00100
VL5 when pattern = 00101
VL6 when pattern = 00110
VL7 when pattern = 00111
VL8 when pattern = 01000
VL16 when pattern = 01001
VL32 when pattern = 01010
VL64 when pattern = 01011
VL128 when pattern = 01100
VL256 when pattern = 01101
#uimm5 when pattern = 01110
#uimm5 when pattern = 01111
#uimm5 when pattern = 10100
#uimm5 when pattern = 10101
#uimm5 when pattern = 10110
#uimm5 when pattern = 10111
#uimm5 when pattern = 11000
#uimm5 when pattern = 11001
#uimm5 when pattern = 11010
#uimm5 when pattern = 11011
#uimm5 when pattern = 11100
#uimm5 when pattern = 11101
#uimm5 when pattern = 11110
#uimm5 when pattern = 11111

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation for all encodings

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
C8.2.714  UQINCW (vector)

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 32-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

Encoding

UQINCW <Zdn>.S{, <pattern>{, MUL #<imm>}}

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;

Assembler symbols

<Zdn>  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern>  Is the optional pattern specifier, defaulting to ALL, encoded in the "pattern" field. It can have the following values:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>POW2</td>
<td>pattern = 00000</td>
</tr>
<tr>
<td>VL1</td>
<td>pattern = 00001</td>
</tr>
<tr>
<td>VL2</td>
<td>pattern = 00010</td>
</tr>
<tr>
<td>VL3</td>
<td>pattern = 00011</td>
</tr>
<tr>
<td>VL4</td>
<td>pattern = 00100</td>
</tr>
<tr>
<td>VL5</td>
<td>pattern = 00101</td>
</tr>
<tr>
<td>VL6</td>
<td>pattern = 00110</td>
</tr>
<tr>
<td>VL7</td>
<td>pattern = 00111</td>
</tr>
<tr>
<td>VL8</td>
<td>pattern = 01000</td>
</tr>
<tr>
<td>VL16</td>
<td>pattern = 01001</td>
</tr>
</tbody>
</table>
VL32  when pattern = 01010
VL64  when pattern = 01011
VL128 when pattern = 01100
VL256 when pattern = 01101
#uimm5 when pattern = 0111x
#uimm5 when pattern = 101x1
#uimm5 when pattern = 10110
#uimm5 when pattern = 1x0x1
#uimm5 when pattern = 1x010
#uimm5 when pattern = 1xx00
MUL4  when pattern = 11101
MUL3  when pattern = 11110
ALL   when pattern = 11111

<imm>  Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
   integer element1 = Int(Elem[operand1, e, esize], unsigned);
   (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.715  UQRSHL

Shift active unsigned elements of the first source vector by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.

Encoding

```
UQRSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Decode for this encoding

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler symbols

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer element = UInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    integer res = (element + round_const) << shift;
    Elem[result, e, esize] = UnsignedSat(res, esize);
  else
```
$\text{Elem}[\text{result}, e, \text{esize}] = \text{Elem}[\text{operand1}, e, \text{esize}]$;

$Z[dn, VL] = \text{result}$;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.716 UQRSHLR

Shift active unsigned elements of the second source vector by corresponding elements of the first source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.

Encoding

UQRSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer element = UInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    integer res = (element + round_const) << shift;
    Elem[result, e, esize] = UnsignedSat(res, esize);
  else
    }
\[ \text{Elem[result, e, esize]} = \text{Elem[operand2, e, esize]}; \]
\[ Z[dn, VL] = \text{result}; \]

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.717  UQRSHRNB

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to \((2^N)-1\). The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Encoding

UQRSHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  B  when tszh = 0, tszl = 01
  H  when tszh = 0, tszl = 1x
  S  when tszh = 1, tszl = xx
The encoding tszh = 0, tszl = 00 is reserved.

<Zn>  Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb>  Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  H  when tszh = 0, tszl = 01
  S  when tszh = 0, tszl = 1x
  D  when tszh = 1, tszl = xx
The encoding tszh = 0, tszl = 00 is reserved.

<const>  Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = (UInt(element) + round_const) >> shift;
  Elem[result, 2*e + 0, esize] = UnsignedSat(res, esize);
  Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
C8.2.718  UQRSHRNT

Shift each unsigned integer value in the source vector elements by an immediate value, and place the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to \(2^N-1\). The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Encoding

\[
\text{UQRSHRNT } <\text{Zd}.<\text{T}, <\text{Zn}.<\text{Tb}, #<\text{const}>
\]

Decode for this encoding

\[
\begin{align*}
\text{if } \text{!HaveSVE2() } \& \& \text{!HaveSME()} \text{ then UNDEFINED;}
\end{align*}
\]

bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler symbols

<Zd>  
Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  
Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

- B when tszh = 0, tszl = 01
- H when tszh = 0, tszl = 1x
- S when tszh = 1, tszl = xx

The encoding tszh = 0, tszl = 00 is reserved.

<Zn>  
Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb>  
Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

- H when tszh = 0, tszl = 01
- S when tszh = 0, tszl = 1x
- D when tszh = 1, tszl = xx

The encoding tszh = 0, tszl = 00 is reserved.

<const>  
Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = (UInt(element) + round_const) >> shift;
  Elem[result, 2*e + 1, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
C8.2.719  **UQSHL (immediate)**

Shift left by immediate each active unsigned element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10  9  8  7</th>
<th>5  4  3  2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 0</td>
<td>tszh</td>
<td>0 0 1 1 1 1 0 0</td>
<td>Pg</td>
<td>tszl</td>
<td>imm3</td>
<td>Zdn</td>
<td></td>
</tr>
</tbody>
</table>
```

**Encoding**

```
UQSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>
```

**Decode for this encoding**

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = UInt(tsize:imm3) - esize;
```

**Assembler symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  - B when tszh = 00, tszl = 01
  - H when tszh = 00, tszl = 1x
  - S when tszh = 01, tszl = xx
  - D when tszh = 1x, tszl = xx
  The encoding tszh = 00, tszl = 00 is reserved.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<const>` Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;
for e = 0 to elements-1
```
integer element1 = UInt(Elem[operand1, e, esize]);
if Elem[mask, e, esize] == '1' then
    integer res = element1 << shift;
    Elem[result, e, esize] = UnsignedSat(res, esize);
else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.720   UQSHL (vectors)

Shift active unsigned elements of the first source vector by corresponding elements of the second source vector and
destructively place the results in the corresponding elements of the first source vector. A positive shift amount
performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is
saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector
register remain unmodified.

Encoding

UQSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer element = UInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer res = element << shift;
    Elem[result, e, esize] = UnsignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.721  UQSHLR
Shift active unsigned elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>0 0 1 1 0 1 1 0</td>
<td>Pg</td>
<td>Zm</td>
<td>Zdn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**
UQSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

**Decoding for this encoding**
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

**Assembler symbols**

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T>  Is the size specifier, encoded in the "size" field. It can have the following values:

B  when size = 00
H  when size = 01
S  when size = 10
D  when size = 11

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer element = UInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer res = element << shift;
    Elem[result, e, esize] = UnsignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand2, e, esize];
\[ Z[dn, VL] = \text{result}; \]

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.722 UQSHRN

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Encoding

UQSHRN <Zd>.<T>, <Zn>.<Tb>, #<const>

Decode for this encoding

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bite(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

- `B` when tszh = 0, tszl = 01
- `H` when tszh = 0, tszl = 1x
- `S` when tszh = 1, tszl = xx

The encoding tszh = 0, tszl = 00 is reserved.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

- `H` when tszh = 0, tszl = 01
- `S` when tszh = 0, tszl = 1x
- `D` when tszh = 1, tszl = xx

The encoding tszh = 0, tszl = 00 is reserved.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
```
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = UInt(element) >> shift;
  Elem[result, 2*e + 0, esize] = UnsignedSat(res, esize);
  Elem[result, 2*e + 1, esize] = Zeros(esize);
Z[d, VL] = result;
C8.2.723   UQSHRNT

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 16 15 14 13 12|11 10 9 | 5 4 | 0 |
| 0 1 0 0 0 1 0 1 0 |1| tszh | tszl |imm3 | 0 0 | f | f | 0 | 1 | Zn | Zd |

U R T
```

tszh

**Encoding**

UQSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

- B when tszh = 0, tszl = 01
- H when tszh = 0, tszl = 1x
- S when tszh = 1, tszl = xx

The encoding tszh = 0, tszl = 00 is reserved.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

- H when tszh = 0, tszl = 01
- S when tszh = 0, tszl = 1x
- D when tszh = 1, tszl = xx

The encoding tszh = 0, tszl = 00 is reserved.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = UInt(element) >> shift;
  Elem[result, 2*e + 1, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
C8.2.724   UQSUB (immediate)

Unsigned saturating subtract an unsigned immediate from each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to \(2^N-1\). This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

Encoding

\[
\text{UQSUB} \quad \langle\text{Zdn}\rangle.<\text{T}>., \langle\text{Zdn}\rangle.<\text{T}>., \#<\text{imm}>, \langle\text{shift}\rangle\
\]

Decode for this encoding

if !HaveSVE() &amp; !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 &lt;&lt; UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm &lt;&lt; 8;
boolean unsigned = TRUE;

Assembler symbols

\(<\text{Zdn}>\)  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

\(<\text{T}>\)  Is the size specifier, encoded in the "size" field. It can have the following values:

- B  when size = 00
- H  when size = 01
- S  when size = 10
- D  when size = 11

\(<\text{imm}>\)  Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

\(<\text{shift}>\)  Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:

- LSL #0 when sh = 0
- LSL #8 when sh = 1

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element1 - imm, esize, unsigned);

Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.725   UQSUB (vectors, predicated)

Subtract active unsigned elements of the second source vector from corresponding unsigned elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to \((2^N)-1\). Inactive elements in the destination vector register remain unmodified.

**Encoding**

UQSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

**Decode for this encoding**

if !HaveSVE2() && !HaveSME() then UNDEFINED;

constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

**Assembler symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if Elem[mask, e, esize] == '1' then
        integer res = UInt(Sat(element1 - element2, esize, unsigned));
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | 5 4 | 0 |
 0 1 0 0 0 1 0 0 | size 0 1 1 0 1 1 1 0 0 | Pg | Zm | Zdn | S U]
$Z[dn, VL] = \text{result}$;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.726   UQSUB (vectors, unpredicated)

Unsigned saturating subtract all elements of the second source vector from corresponding elements of the first
source vector and place the results in the corresponding elements of the destination vector. Each result element is
saturated to the N-bit element's unsigned integer range 0 to \(2^N\)-1. This instruction is unpredicated.

Encoding

\[
\text{UQSUB} <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
\]

Decoding for this encoding

\[
\text{if !HaveSVE() && !HaveSME() then UNDEFINED;}
\]

\[
\begin{align*}
\text{constant integer esize} & \equiv 8 << \text{UInt(size)}; \\
\text{integer n} & \equiv \text{UInt}(Zn); \\
\text{integer m} & \equiv \text{UInt}(Zm); \\
\text{integer d} & \equiv \text{UInt}(Zd); \\
\text{boolean unsigned} & \equiv \text{TRUE};
\end{align*}
\]

Assembler symbols

\[
\begin{align*}
& <Zd> \quad \text{Is the name of the destination scalable vector register, encoded in the "Zd" field.} \\
& <T> \quad \text{Is the size specifier, encoded in the "size" field. It can have the following values:} \\
& \quad B \quad \text{when size = 00} \\
& \quad H \quad \text{when size = 01} \\
& \quad S \quad \text{when size = 10} \\
& \quad D \quad \text{when size = 11} \\
& <Zn> \quad \text{Is the name of the first source scalable vector register, encoded in the "Zn" field.} \\
& <Zm> \quad \text{Is the name of the second source scalable vector register, encoded in the "Zm" field.}
\end{align*}
\]

Operation

\[
\begin{align*}
\text{CheckSVEEnabled();} \\
\text{constant integer VL = CurrentVL;} \\
\text{constant integer elements = VL DIV esize;} \\
\text{bits(VL) operand1 = Z[n, VL];} \\
\text{bits(VL) operand2 = Z[m, VL];} \\
\text{bits(VL) result;} \\
\text{for e = 0 to elements-1} \\
\quad \text{integer element1 = Int(Elem[operand1, e, esize], unsigned);} \\
\quad \text{integer element2 = Int(Elem[operand2, e, esize], unsigned);} \\
\quad (\text{Elem[result, e, esize], -}) = \text{SatQ(element1 - element2, esize, unsigned);} \\
\text{Z[d, VL] = result;}
\end{align*}
\]
C8.2.727  UQSUBR

Subtract active unsigned elements of the first source vector from corresponding unsigned elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.

Encoding

UQSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

Assembler symbols

<Zdn>  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T>   Is the size specifier, encoded in the "size" field. It can have the following values:
   B   when size = 00
   H   when size = 01
   S   when size = 10
   D   when size = 11
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
   integer element1 = UInt(Elem[operand1, e, esize]);
   integer element2 = UInt(Elem[operand2, e, esize]);
   if Elem[mask, e, esize] == '1' then
      integer res = UInt(Sat(element2 - element1, esize, unsigned));
      Elem[result, e, esize] = res<esize-1:0>;
   else
      Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.728  UQXTNB

Saturate the unsigned integer value in each source element to half the original source element width, and place the results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero.

Encoding

UQXTNB <Zd>.<T>, <Zn>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
 when '001' esize = 16;
 when '010' esize = 32;
 when '100' esize = 64;
 otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

  B when tszh = 0, tszl = 01
  H when tszh = 0, tszl = 10
  S when tszh = 1, tszl = 00

The following encodings are reserved:

  • tszh = 0, tszl = 00.
  • tszh = x, tszl = 11.
  • tszh = 1, tszl = 01.
  • tszh = 1, tszl = 10.

<Zn>  Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb>  Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

  H when tszh = 0, tszl = 01
  S when tszh = 0, tszl = 10
  D when tszh = 1, tszl = 00

The following encodings are reserved:

  • tszh = 0, tszl = 00.
  • tszh = x, tszl = 11.
  • tszh = 1, tszl = 01.
  • tszh = 1, tszl = 10.
Operation

    CheckSVEEnabled();
    constant integer VL = CurrentVL;
    constant integer PL = VL DIV 8;
    constant integer elements = VL DIV esize;
    bits(VL) operand1 = Z[n, VL];
    bits(VL) result;
    constant integer halfesize = esize DIV 2;

    for e = 0 to elements-1
        integer element1 = UInt(Elem[operand1, e, esize]);
        bits(halfesize) res = UnsignedSat(element1, halfesize);
        Elem[result, 2*e + 0, halfesize] = res;
        Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);

    Z[d, VL] = result;
C8.2.729   UQXTNT

Saturate the unsigned integer value in each source element to half the original source element width, and place the results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged.

Encoding

UQXTNT <Zd>.<T>, <Zn>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
  case tsize of
    when '001' esize = 16;
    when '010' esize = 32;
    when '100' esize = 64;
    otherwise UNDEFINED;
  integer n = UInt(Zn);
  integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

- B when tszh = 0, tszl = 01
- H when tszh = 0, tszl = 10
- S when tszh = 1, tszl = 00

The following encodings are reserved:
- tszh = 0, tszl = 00.
- tszh = x, tszl = 11.
- tszh = 1, tszl = 01.
- tszh = 1, tszl = 10.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

- H when tszh = 0, tszl = 01
- S when tszh = 0, tszl = 10
- D when tszh = 1, tszl = 00

The following encodings are reserved:
- tszh = 0, tszl = 00.
- tszh = x, tszl = 11.
- tszh = 1, tszl = 01.
- tszh = 1, tszl = 10.
**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  bits(halfesize) res = UnsignedSat(element1, halfesize);
  Elem[result, 2*e + 1, halfesize] = res;

Z[d, VL] = result;
C8.2.730 URECPE

Find the approximate reciprocal of each active unsigned element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
<td>0 0 0 0</td>
<td>1 0 1 0</td>
<td>0 1 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>

Encoding

URECPE <Zd>.S, <Pg>/M, <Zn>.S

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size != '10' then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = UnsignedRecipEstimate(element);

Z[d, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.731 URHADD

Add active unsigned elements of the first source vector to corresponding unsigned elements of the second source vector, shift right one bit, and destructively place the rounded results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Encoding

URHADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer round_const = 1;
for e = 0 to elements-1
integer element1 = UInt(Elem[operand1, e, esize]);
integer element2 = UInt(Elem[operand2, e, esize]);
if Elem[mask, e, esize] == '1' then
integer res = (element1 + element2 + round_const) >> 1;
Elem[result, e, esize] = res<esize-1:0>;
else
Elem[result, e, esize] = Elem[operand1, e, esize];
$Z[dn, VL] = \text{result}$;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.732 \textbf{URSHL}

Shift active unsigned elements of the first source vector by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.

\begin{verbatim}
Encoding
URSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols
<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  \begin{itemize}
    \item \texttt{B} when \texttt{size} = 00
    \item \texttt{H} when \texttt{size} = 01
    \item \texttt{S} when \texttt{size} = 10
    \item \texttt{D} when \texttt{size} = 11
  \end{itemize}
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
binary(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer element = UInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    integer res = (element + round_const) << shift;
    Elem[result, e, esize] = res<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
\end{verbatim}
\[ Z[dn, VL] = \text{result}; \]

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.733 URSHLR

Shift active unsigned elements of the second source vector by corresponding elements of the first source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed.Inactive elements in the destination vector register remain unmodified.

Encoding

URSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer element = UInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    integer res = (element + round_const) << shift;
    Elem[result, e, esize] = res<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand2, e, esize];
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.734  URSHR

Shift right by immediate each active unsigned element of the source vector, and destructively place the rounded results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

Encoding

URSHR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
b = tszh:tszl;
integer esize;
  case tszh of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
  integer g = UInt(Pg);
  integer dn = UInt(Zdn);
  integer shift = (2 * esize) - UInt(tsz:imm3);

Assembler symbols

<Zdn>  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T>  Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  B  when tszh = 00, tszl = 01
  H  when tszh = 00, tszl = 1x
  S  when tszh = 01, tszl = xx
  D  when tszh = 1x, tszl = xx

The encoding tszh = 00, tszl = 00 is reserved.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const>  Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;
integer round_const = 1 << (shift-1);
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
if Elem[mask, e, esize] == '1' then
    integer res = (element1 + round_const) >> shift;
    Elem[result, e, esize] = res<esize-1:0>;
else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.735 URSQRTE

Find the approximate reciprocal square root of each active unsigned element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Encoding

URSQRTE <Zd>.S, <Pg>/M, <Zn>.S

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size != '10' then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = UnsignedRSqrtEstimate(element);
Z[d, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.736  URSRA

Shift right by immediate each unsigned element of the source vector, inserting zeroes, and add the rounded intermediate result destructively to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
0 1 0 0 0 1 0 1 | tszh 0 tszl imm3 1 1 1 0 | 1 1 | Zn | Zda
```

**Encoding**

URSRA <Zda>, <T>, <Zn>, <T>, #<const>

**Decode for this encoding**

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer n = UInt(Zn);
integer da = UInt(Zda);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

**Assembler symbols**

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- `<T>` Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
  - B when tszh = 00, tszl = 01
  - H when tszh = 00, tszl = 1x
  - S when tszh = 01, tszl = xx
  - D when tszh = 1x, tszl = xx
  The encoding tszh = 00, tszl = 00 is reserved.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<const>` Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[da, VL];
bits(VL) result;
integer round_const = 1 << (shift - 1);
for e = 0 to elements-1
  integer element = (UInt(Element[operand1, e, esize]) + round_const) >> shift;
  Element[result, e, esize] = Element[operand2, e, esize] + element<esize-1:0>;
```
Z[da, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.737 USDOT (indexed)

The unsigned by signed integer indexed dot product instruction computes the dot product of a group of four
unsigned 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four
signed 8-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds
the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group
position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE
(FEAT_I8MM)

Encoding


Decode for this encoding

if (!HaveSVE() && !HaveSME()) || !HaveInt8MatMulExt() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
<imm> Is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in
the range 0 to 3, encoded in the "i2" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];


for i = 0 to 3
    integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
    integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
    res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.738  USDOT (vectors)

The unsigned by signed integer dot product instruction computes the dot product of a group of four unsigned 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four signed 8-bit integer values in the corresponding 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

**SVE**

(FEAT_I8MM)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
<td>1 0 0 1</td>
<td>Zda</td>
<td>Zn</td>
<td>Zm</td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**


**Decode for this encoding**

if (!HaveSVE() && !HaveSME()) || !HaveInt8MatMulExt() then UNDEFINED;

constant integer esize = 32;

integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

**Assembler symbols**

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) res = Elem[operand3, e, esize];
  for i = 0 to 3
    integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
    integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
    res = res + element1 * element2;
  Elem[result, e, esize] = res;
Z[da, VL] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.739 USHLLB

Shift left by immediate each even-numbered unsigned element of the source vector, and place the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

.Encoding

USHLLB <Zd>,<T>,<Zn>,<Tb>,#<const>

.Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bite(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;

.Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
    H when tszh = 0, tszl = 01
    S when tszh = 0, tszl = 1x
    D when tszh = 1, tszl = xx
The encoding tszh = 0, tszl = 00 is reserved.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
    B when tszh = 0, tszl = 01
    H when tszh = 0, tszl = 1x
    S when tszh = 1, tszl = xx
The encoding tszh = 0, tszl = 00 is reserved.
<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

.Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element = Elem[operand, 2*e + 0, esize];
  integer shifted_value = UInt(element) << shift;
  Elem[result, e, 2*esize] = shifted_value<2*esize-1:0>;
Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.740 USHLLT

Shift left by immediate each odd-numbered unsigned element of the source vector, and place the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

Encoding

USHLLT <Zd>,<T>, <Zn>,<Tb>, #<const>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
H  when tszh = 0, tszl = 01
S  when tszh = 0, tszl = 1x
D  when tszh = 1, tszl = xx
The encoding tszh = 0, tszl = 00 is reserved.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
B  when tszh = 0, tszl = 01
H  when tszh = 0, tszl = 1x
S  when tszh = 1, tszl = xx
The encoding tszh = 0, tszl = 00 is reserved.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, 2*e + 1, esize];
    integer shifted_value = UInt(element) << shift;
    Elem[result, e, 2*esize] = shifted_value<2*esize-1:0>;

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.741 USMMLA

The unsigned by signed integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of unsigned 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of signed 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE

(FEAT_I8MM)

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 16 15 14 13 | 12 11 10 9 | 5 4 | 0 |
| 0 1 0 0 0 1 0 1 | 1 0 0 | Zm | 1 0 0 1 1 0 | Zn | Zda |

Encoding


Decode for this encoding

if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;

Assembler symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result = Zeros(VL);
basis(128) op1, op2;
basis(128) res, addend;

for s = 0 to segments-1
    op1 = Elem[operand1, s, 128];
    op2 = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];

res = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
Elem[result, s, 128] = res;

Z[da, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.742   USQADD

Add active signed elements of the source vector to the corresponding unsigned elements of the addend vector, and
destructively place the results in the corresponding elements of the addend vector. Each result element is saturated
to the N-bit element's unsigned integer range 0 to (2^N)-1.Inactive elements in the destination vector register remain
unmodified.

[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 |  5 4 |  0 |  
0 1 0 0 0 1 0 0 | size 0 1 1 | 0 1 1 0 0 | Pg | Zm | Zdn  
S U

Encoding

USQADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = UnsignedSat(UInt(element1) + SInt(element2), esize);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn, VL] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.743   USRA

Shift right by immediate each unsigned element of the source vector, inserting zeroes, and add the truncated
intermediate result destructively to the corresponding elements of the addend vector. The immediate shift amount
is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

Encoding

USRA <Zda>.<T>, <Zn>.<T>, #<const>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
  case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
  integer n = UInt(Zn);
  integer da = UInt(Zda);
  integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler symbols

<Zda>  Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T>    Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:
    B  when tszh = 00, tszl = 01
    H  when tszh = 00, tszl = 1x
    S  when tszh = 01, tszl = xx
    D  when tszh = 1x, tszl = xx
    The encoding tszh = 00, tszl = 00 is reserved.

<Zn>    Is the name of the first source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
  integer element = UInt(Elem[operand1, e, esize]) >> shift;
  Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
Z[da, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.744   USUBLB

Subtract the even-numbered unsigned elements of the second source vector from the corresponding unsigned elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

Encoding

USUBLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 0;
boolean unsigned = TRUE;

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H    when size = 01
S    when size = 10
D    when size = 11
The encoding size = 00 is reserved.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
B    when size = 01
H    when size = 10
S    when size = 11
The encoding size = 00 is reserved.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bv(VL) operand1 = Z[n, VL];
bv(VL) operand2 = Z[m, VL];
bv(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
  integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
  integer res = element1 - element2;
  Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.745   USUBLT

Subtract the odd-numbered unsigned elements of the second source vector from the corresponding unsigned elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12|11 10 9 | 5 4 | 0 |
  0 1 0 0 0 1 0 1 size 0 | Zn 0 0 0 1 1 | Zm | Zd
```

**Encoding**

USUBLT «Zd».<T>, «Zn».<Tb>, «Zm».<Tb>

**Decode for this encoding**

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 1;
boolean unsigned = TRUE;
```

**Assembler symbols**

«Zd»   Is the name of the destination scalable vector register, encoded in the "Zd" field.

«T»   Is the size specifier, encoded in the "size" field. It can have the following values:

- H when size = 01
- S when size = 10
- D when size = 11

The encoding size = 00 is reserved.

«Zn»   Is the name of the first source scalable vector register, encoded in the "Zn" field.

«Tb»   Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 01
- H when size = 10
- S when size = 11

The encoding size = 00 is reserved.

«Zm»   Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
```
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
  integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
  integer res = element1 - element2;
  Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.746  USUBWB

Subtract the even-numbered unsigned elements of the second source vector from the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1</td>
<td>Zm</td>
<td>0 1 0 1 1 0</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

USUBWB <Zd..<T>, <Zn..<T>, <Zm..<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>  Is the size specifier, encoded in the "size" field. It can have the following values:
H  when size = 01
S  when size = 10
D  when size = 11
The encoding size = 00 is reserved.

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb>  Is the size specifier, encoded in the "size" field. It can have the following values:
B  when size = 01
H  when size = 10
S  when size = 11
The encoding size = 00 is reserved.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
\[
\text{Elem}[\text{result, e, esize}] = (\text{element1} - \text{element2})_{\text{esize-1:0}};
\]
\[
\text{Z}[d, VL] = \text{result};
\]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.747    USUBWT

Subtract the odd-numbered unsigned elements of the second source vector from the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated. This instruction is unpredicated.

Encoding

USUBWT <Zd>,<T>, <Zn>,<T>, <Zm>,<Tb>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
H when size = 01
S when size = 10
D when size = 11
The encoding size = 00 is reserved.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 01
H when size = 10
S when size = 11
The encoding size = 00 is reserved.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
\texttt{Elem[result, e, esize] = (element1 - element2)_{esize-1:0};}

\texttt{Z[d, VL] = result;}

\section*{Operational information}

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then when PSTATE.DIT is 1:

\begin{itemize}
  \item The execution time of this instruction is independent of:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
      \item The values of the NZCV flags.
    \end{itemize}
  \item The response of this instruction to asynchronous exceptions does not vary based on:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
      \item The values of the NZCV flags.
    \end{itemize}
\end{itemize}
C8.2.748 UUNPKHI, UUNPKLO

Unpack elements from the lowest or highest half of the source vector and then zero-extend them to place in elements of twice their size within the destination vector. This instruction is unpredicated.

**High half**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
0 0 0 0 0 1 0 1 size 1 1 0 0 1 1 0 1 Zn Zd
```

**Encoding**

UUNPKHI <Zd>.<T>, <Zn>.< Tb>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
boolean hi = TRUE;

**Low half**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 | 5 4 | 0 ]
0 0 0 0 0 1 0 1 size 1 1 0 0 0 1 1 1 Zn Zd
```

**Encoding**

UUNPKLO <Zd>.<T>, <Zn>.< Tb>

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
boolean hi = FALSE;

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- **H** when size = 01
- **S** when size = 10
- **D** when size = 11

The encoding size = 00 is reserved.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 01
- H when size = 10
- S when size = 11

The encoding size = 00 is reserved.

**Operation for all encodings**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer hsize = esize DIV 2;
bits(VL) operand = Z[n, VL];
bits(VL) result;
for e = 0 to elements-1
    bits(hsize) element = if hi then Elem[operand, e + elements, hsize] else Elem[operand, e, hsize];
    Elem[result, e, esize] = Extend(element, esize, unsigned);
Z[d, VL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.749 UXTB, UXTH, UXTW

Zero-extend the least-significant sub-element of each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Byte

Encoding

UXTB <Zd>.<T>, <Pg>/M, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 8;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;

Halfword

Encoding

UXTH <Zd>.<T>, <Pg>/M, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;

Word

Encoding

UXTW <Zd>.<T>, <Pg>/M, <Zn>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
**Encoding**

UXTW <Zd>.D, <Pg>/M, <Zn>.D

**Decode for this encoding**

if !HaveSVE() && !HaveSME() then UNDEFINED;
if size != '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;

**Assembler symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> For the byte variant: is the size specifier, encoded in the "size" field. It can have the following values:

H when size = 01
S when size = 10
D when size = 11

The encoding size = 00 is reserved.

For the halfword variant: is the size specifier, encoded in the "size<0>" field. It can have the following values:

S when size<0> = 0
D when size<0> = 1

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation for all encodings**

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bite(VL) result = Z[d, VL];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = Extend(element<s_esize-1:0>, esize, unsigned);
Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
— The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.750 UZP1, UZP2 (predicates)

Concatenate adjacent even or odd-numbered elements from the first and second source predicates and place in elements of the destination predicate. This instruction is unpredicated.

Even

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 | 16|15|14|13|12|11|10| 9| 8 | 5| 4| 3| 0 |
|--------------|--------------|--------------|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 0 0 0 0 0 1 0 1 | size | 1 0 | Pm | 0 1 0 | 0 | 0 | Pn | 0 | Pd |
```

Encoding

UZP1 <Pd>,<T>, <Pn>,<T>, <Pm>,<T>

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 0;
```

Odd

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 | 16|15|14|13|12|11|10| 9| 8 | 5| 4| 3| 0 |
|--------------|--------------|--------------|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 0 0 0 0 0 1 0 1 | size | 1 0 | Pm | 0 1 0 | 0 | 1 | Pn | 0 | Pd |
```

Encoding

UZP2 <Pd>,<T>, <Pn>,<T>, <Pm>,<T>

Decode for this encoding

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 1;
```

Assembler symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation for all encodings

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (esize * 2);
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for p = 0 to pairs - 1
    Elem[result, p, esize DIV 8] = Elem[operand1, 2*p+part, esize DIV 8];

for p = 0 to pairs - 1
    Elem[result, pairs+p, esize DIV 8] = Elem[operand2, 2*p+part, esize DIV 8];

P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.751  UZP1, UZP2 (vectors)

Concatenate adjacent even or odd-numbered elements from the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated. The 128-bit element variant of this instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant of the instruction is implemented.

### Even

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
<td>size 1</td>
<td>Zm 0 1 1 0 1 0</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Encoding

UZP1 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

#### Decode for this encoding

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

### Even (quadwords)

(_FEAT_F64MM)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1 1 0 1</td>
<td>Zm 0 0 0 0 1 0</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Encoding

UZP1 <Zd>.Q, <Zn>.Q, <Zm>.Q

#### Decode for this encoding

```c
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

### Odd

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
<td>size 1</td>
<td>Zm 0 1 1 0 1 1</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Encoding
UZP2 <Zd>,<T>, <Zn>,<T>, <Zm>,<T>

Decode for this encoding
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;

Odd (quadwords)
(FEAT_F64MM)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 13 12 11 10 9</th>
<th>5 4 0</th>
<th>1 1 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 1 0 0</td>
<td>Zm</td>
<td>Zn</td>
</tr>
</tbody>
</table>

Encoding
UZP2 <Zd>.Q, <Zn>.Q, <Zm>.Q

Decode for this encoding
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation for all encodings
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize = 2 then UNDEFINED;
constant integer pairs = VL DIV (esize = 2);
bids(VL) operand1 = Z[n, VL];
bids(VL) operand2 = Z[m, VL];
bids(VL) result = Zeros(VL);
for p = 0 to pairs - 1
$\text{Elem[result, } p, \text{ esize]} = \text{Elem[operand1, } 2\times p + \text{ part, esize]}$;

for $p = 0$ to $\text{pairs - 1}$

$\text{Elem[result, pairs} + p, \text{ esize]} = \text{Elem[operand2, } 2\times p + \text{ part, esize]}$;

$Z[d, \text{ VL}] = \text{result};$

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.752 WHILEGE

Generate a predicate that starting from the highest numbered element is true while the decremented value of the first, signed scalar operand is greater than or equal to the second scalar operand and false thereafter down to the lowest numbered element.

If the second scalar operand is equal to the minimum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

Encoding

WHILEGE <Pd>.<T>, <R><n>, <R><m>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
n = UInt(Rn);
m = UInt(Rm);
d = UInt(Pd);
boolean unsigned = FALSE;
SVECmp op = Cmp_GE;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11
<R> Is a width specifier, encoded in the "sf" field. It can have the following values:
  W when sf = 0
  X when sf = 1
<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.
<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;

for e = elements-1 downto 0
  boolean cond;
  case op of
    when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
    when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));
  
  last = last && cond;
  ElemP[result, e, esize] = if last then '1' else '0';
  operand1 = operand1 - 1;
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
  P[d, PL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.753   WHILEGT

Generate a predicate that starting from the highest numbered element is true while the decrementsing value of the
first, signed scalar operand is greater than the second scalar operand and false thereafter down to the lowest
numbered element.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand
is decremented by one for each destination predicate element, irrespective of the predicate result element size. The
first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition
flags based on the predicate result, and the V flag to zero.

Encoding

WHILEGT <Pd>.<T>, <R><n>, <R><m>

Decode for this encoding

if !HaveSVEZ() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = FALSE;
SVECmp op = Cmp_GT;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B     when size = 00
H     when size = 01
S     when size = 10
D     when size = 11
<R> Is a width specifier, encoded in the "sf" field. It can have the following values:
W     when sf = 0
X     when sf = 1
<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the
"Rn" field.
<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the
"Rm" field.
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;

for e = elements-1 downto 0
  boolean cond;
  case op of
    when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
    when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

  last = last && cond;
  ElemP[result, e, esize] = if last then '1' else '0';
  operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.754   WHILEHI

Generate a predicate that starting from the highest numbered element is true while the decrementing value of the first, unsigned scalar operand is higher than the second scalar operand and false thereafter down to the lowest numbered element.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

Encoding

WHILEHI <Pd>.<T>, <R><n>, <R><m>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVECmp op = Cmp_GT;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11

<R> Is a width specifier, encoded in the "sf" field. It can have the following values:
W when sf = 0
X when sf = 1

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
for e = elements-1 downto 0
  boolean cond;
  case op of
    when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
    when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));
  end case;
  last = last && cond;
  ElemP[result, e, esize] = if last then '1' else '0';
  operand1 = operand1 - 1;
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.755  WHILEHS

Generate a predicate that starting from the highest numbered element is true while the decrementing value of the first, unsigned scalar operand is higher or same as the second scalar operand and false thereafter down to the lowest numbered element.

If the second scalar operand is equal to the minimum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

Encoding

WHILEHS <Pd>.<T>, <R><n>, <R><m>

Decoding for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVECmp op = Cmp_GE;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11

<R> Is a width specifier, encoded in the "sf" field. It can have the following values:
  W when sf = 0
  X when sf = 1

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.
**Operation**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    ElemP[result, e, esize] = if last then '1' else '0';
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.756 WHILELE

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, signed scalar operand is less than or equal to the second scalar operand and false thereafter up to the highest numbered element.

If the second scalar operand is equal to the maximum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

```
[31 30 29 28|27 26 25 24|23 22 21 20]  |  16|15 14 13 12|11 10 9 |  5 4 3 0
0 0 1 0 0 1 0 1  size 1  |  Rm 0 0 0 sf 0 1  Rn 1  |  Pd

\text{Encoding}
```

WHILELE <Pd>.<T>, <R><n>, <R><m>

```
\text{Decode for this encoding}
if \text{!HaveSVE()} \&\& \text{!HaveSME()} then UNDEFINED;
constant integer esize = 8 \ll\text{UInt(size)};
constant integer rsize = 32 \ll\text{UInt(sf)};
integer n = \text{UInt(Rn)};
integer m = \text{UInt(Rm)};
integer d = \text{UInt(Pd)};
boolean unsigned = FALSE;
SVE\text{Cmp} op = \text{Cmp\_LE};
```

```
\text{Assembler symbols}
<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11
<R> Is a width specifier, encoded in the "sf" field. It can have the following values:
W when sf = 0
X when sf = 1
<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.
<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.
```
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
for e = 0 to elements-1
  boolean cond;
  case op of
    when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
    when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));
  last = last && cond;
  ElemP[result, e, esize] = if last then '1' else '0';
  operand1 = operand1 + 1;
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.757  WHILELO

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, unsigned scalar operand is lower than the second scalar operand and false thereafter up to the highest numbered element.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

Encoding

WHILELO <Pd><T>, <R><n>, <R><m>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVECmp op = Cmp_LT;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11

<R> Is a width specifier, encoded in the "sf" field. It can have the following values:
  W when sf = 0
  X when sf = 1

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.
**Operation**

```plaintext
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
for e = 0 to elements-1
    boolean cond;
case op of
    when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
    when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));
    last = last && cond;
    ElemP[result, e, esize] = if last then '1' else '0';
    operand1 = operand1 + 1;
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.758 WHILELS

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, unsigned scalar operand is lower or same as the second scalar operand and false thereafter up to the highest numbered element.

If the second scalar operand is equal to the maximum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

Encoding

WHILELS <Pd>.<T>, <R><n>, <R><m>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVECmp op = Cmp_LE;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
  B when size = 00
  H when size = 01
  S when size = 10
  D when size = 11

<R> Is a width specifier, encoded in the "sf" field. It can have the following values:
  W when sf = 0
  X when sf = 1

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
for e = 0 to elements-1
  boolean cond;
  case op of
    when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
    when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));
  last = last && cond;
  ElemP[result, e, esize] = if last then '1' else '0';
  operand1 = operand1 + 1;
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.759  WHILELT

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, signed scalar operand is less than the second scalar operand and false thereafter up to the highest numbered element.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

Encoding

WHILELT <Pd>.<T>, <R><n>, <R><m>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = FALSE;
SVECmp op = Cmp_LT;

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
B when size = 00
H when size = 01
S when size = 10
D when size = 11
<R> Is a width specifier, encoded in the "sf" field. It can have the following values:
W when sf = 0
X when sf = 1
<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.
<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.
Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
for e = 0 to elements-1
boolean cond;
case op of
    when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
    when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));
last = last && cond;
   ElemP[result, e, esize] = if last then '1' else '0';
operand1 = operand1 + 1;
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
C8.2.760  WHILERW

This instruction checks two addresses for a conflict or overlap between address ranges of the form [addr, addr+VL÷8), where VL is the accessible vector length in bits, that could result in a loop-carried dependency through memory due to the use of these addresses by contiguous load and store instructions within the same iteration of a loop. Generate a predicate whose elements are true while the addresses cannot conflict within the same iteration, and false thereafter. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

Encoding

WHILERW <Pd>,<T>, <Xn>, < Xm>

Decode for this encoding

if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);

Assembler symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in the "size" field. It can have the following values:
   B when size = 00
   H when size = 01
   S when size = 10
   D when size = 11
<Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(64) src1 = X[n, 64];
bits(64) src2 = X[m, 64];
integer operand1 = UInt(src1);
integer operand2 = UInt(src2);
bits(PL) result;

integer diff = Abs(operand2 - operand1) DIV (esize DIV 8);
for e = 0 to elements-1
   if diff == 0 || e < diff then
      ElemP[result, e, esize] = '1';
   else
Element\[result, e, esize\] = '0';
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P\[d, PL\] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.761 WHILEWR

This instruction checks two addresses for a conflict or overlap between address ranges of the form [addr, addr + VL÷8), where VL is the accessible vector length in bits, that could result in a loop-carried dependency through memory due to the use of these addresses by contiguous load and store instructions within the same iteration of a loop. Generate a predicate whose elements are true while the addresses cannot conflict within the same iteration, and false thereafter. Sets the FIRST (N), NONE (Z), LAST (C) condition flags based on the predicate result, and the V flag to zero.

Encoding

WHILEWR <Pd>.<T>, <Xn>, <Xm>

Operation

CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(64) src1 = X[n, 64];
bits(64) src2 = X[m, 64];
integer operand1 = UInt(src1);
integer operand2 = UInt(src2);
bits(PL) result;

integer diff = (operand2 - operand1) DIV (esize DIV 8);
for e = 0 to elements-1
   if diff <= 0 || e < diff then
      ElemP[result, e, esize] = '1';
   else
\textbf{Operational information}

If FEAT\_SVE2 is implemented or FEAT\_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
WRFFR

Read the source predicate register and place in the first-fault register (FFR). This instruction is intended to restore a saved FFR and is not recommended for general use by applications.

This instruction requires that the source predicate contains a MONOTONIC predicate value, in which starting from bit 0 there are zero or more 1 bits, followed only by 0 bits in any remaining bit positions. If the source is not a monotonic predicate value, then the resulting value in the FFR will be UNPREDICTABLE. It is not possible to generate a non-monotonic value in FFR when using SETFFR followed by first-fault or non-fault loads.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

**Encoding**

WRFFR <Pn>.B

**Decode for this encoding**

if !HaveSVE() then UNDEFINED;
integer n = UInt(Pn);

**Assembler symbols**

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

**Operation**

CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) operand = P[n, PL];

constant integer hsb = HighestSetBit(operand);
if hsb < 0 || IsOnes(operand<hsb:0>) then
    FFR[PL] = operand;
else // not a monotonic predicate
    FFR[PL] = bits(PL) UNKNOWN;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.763   XAR

Bitwise exclusive OR the corresponding elements of the first and second source vectors, then rotate each result element right by an immediate amount. The final results are destructively placed in the corresponding elements of the destination and first source vector. This instruction is unpredicated.

```
0 0 0 0 0 1 0 0 | tszh 1 | tszl | imm3 | 0 0 1 1 0 1 | Zm | Zdn
```

**Encoding**

XAR <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, #<const>

**Decode for this encoding**

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
  case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
  integer m = UInt(Zm);
  integer dn = UInt(Zdn);
  integer rot = (2 * esize) - UInt(tsize:imm3);
```

**Assembler symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in the "tszh:tszl" field. It can have the following values:

- B when tszh = 00, tszl = 01
- H when tszh = 00, tszl = 1x
- S when tszh = 01, tszl = xx
- D when tszh = 1x, tszl = xx

The encoding tszh = 00, tszl = 00 is reserved.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

**Operation**

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
```
\[
\text{Elem}(\text{result, e, esize}) = \text{ROR}(\text{element1 EOR element2, rot});
\]
\[
Z[dn, VL] = \text{result};
\]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
C8.2.764 ZIP1, ZIP2 (predicates)

Interleave alternating elements from the lowest or highest halves of the first and second source predicates and place in elements of the destination predicate. This instruction is unpredicated.

**High halves**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12|11 10 9 8] 5 4 3 0
0 0 0 0 0 1 0 1 | Pm 0 1 0 0 1 0 | Pn 0 | Pd 0
```

**Encoding**

ZIP2 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 1;
```

**Low halves**

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12|11 10 9 8] 5 4 3 0
0 0 0 0 0 1 0 1 | Pm 0 1 0 0 0 0 | Pn 0 | Pd 0
```

**Encoding**

ZIP1 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>

**Decode for this encoding**

```c
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 0;
```

**Assembler symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<T>` Is the size specifier, encoded in the "size" field. It can have the following values:
  - B when size = 00
  - H when size = 01
  - S when size = 10
  - D when size = 11
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` Is the name of the second source scalable predicate register, encoded in the "Pm" field.
**Operation for all encodings**

```c
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (esize * 2);
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

integer base = part * pairs;
for p = 0 to pairs-1
    Elem[result, 2*p+0, esize DIV 8] = Elem[operand1, base+p, esize DIV 8];
    Elem[result, 2*p+1, esize DIV 8] = Elem[operand2, base+p, esize DIV 8];

P[d, PL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C8.2.765 ZIP1, ZIP2 (vectors)

Interleave alternating elements from the lowest or highest halves of the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated. The 128-bit element variant of this instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant of the instruction is implemented.

High halves

<table>
<thead>
<tr>
<th>[31 30 29 28] [27 26 25 24] [23 22 21 20]</th>
<th>16 [15 14 13 12] [11 10 9]</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
<td>Zm 0 1 1 0 0 1</td>
<td>Zn</td>
<td>Zd</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

ZIP2 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;

High halves (quadwords)

(FEAT_F64MM)

<table>
<thead>
<tr>
<th>[31 30 29 28] [27 26 25 24] [23 22 21 20]</th>
<th>16 [15 14 13 12] [11 10 9]</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
<td>Zm 0 0 0 0 0 1</td>
<td>Zn</td>
<td>Zd</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

ZIP2 <Zd>.Q, <Zn>.Q, <Zm>.Q

Decode for this encoding

if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;

Low halves

<table>
<thead>
<tr>
<th>[31 30 29 28] [27 26 25 24] [23 22 21 20]</th>
<th>16 [15 14 13 12] [11 10 9]</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
<td>Zm 0 1 1 0 0 0</td>
<td>Zn</td>
<td>Zd</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Encoding

ZIP1 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

Decode for this encoding

if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;

Low halves (quadwords)

(FEAT_F64MM)

\[
\begin{array}{cccccccccccc|ccc|}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & Zm & 0 & 0 & 0 & 0 & 0 & 0 & Zn & Zd \\
\end{array}
\]

Encoding

ZIP1 <Zd>.Q, <Zn>.Q, <Zm>.Q

Decode for this encoding

if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;

Assembler symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in the "size" field. It can have the following values:

- B when size = 00
- H when size = 01
- S when size = 10
- D when size = 11

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation for all encodings

if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize * 2 then UNDEFINED;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Zeros(VL);
integer base = part * pairs;
for p = 0 to pairs-1
  Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
  Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

Z[d, VL] = result;

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Chapter C9
SME Instruction Descriptions

This chapter describes the SME instructions.

It contains the following sections:
• About the SME instructions on page C9-4542.
• Alphabetical list of SME instructions on page C9-4543.
C9.1  About the SME instructions

This section is added for ease of reference. For more information, see, Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A (ARM DDI 0616).

Alphabetical list of SME instructions on page C9-4543 gives full descriptions of the SME instructions.

A64 instruction set encoding on page C4-388 in the A64 Instruction Encodings chapter provides an overview of the instruction encodings as part of an instruction class within a functional group.
C9.2 Alphabetical list of SME instructions

This section lists every section in the SME category of the A64 instruction set. For details of the format used, see Structure of the A64 assembler language on page C1-228.
C9.2.1 ADDHA

Add each element of the source vector to the corresponding active element of each horizontal slice of a ZA tile. The tile elements are predicated by a pair of governing predicates. An element of a horizontal slice is considered active if its corresponding element in the second governing predicate is TRUE and the element corresponding to its horizontal slice number in the first governing predicate is TRUE. Inactive elements in the destination tile remain unmodified.

ID_AAA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

32-bit

(FEAT_SME)

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 13 12</th>
<th>10 9</th>
<th>5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 Pm Pn Zn 0 0 ZAda</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Encoding

ADDHA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S

Decode for this encoding

if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer da = UInt(ZAda);

64-bit

(FEAT_SME_I16I64)

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 13 12</th>
<th>10 9</th>
<th>5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 0 0 0 1 0 1 0 0 0 0 Pm Pn Zn 0 0 ZAda</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Encoding

ADDHA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D

Decode for this encoding

if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer da = UInt(ZAda);

Assembler symbols

<ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation for all encodings**

```c
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand_src = Z[n, VL];
bits(dim*dim*esize) operand_acc = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for col = 0 to dim-1
    bits(esize) element = Elem[operand_src, col, esize];
    for row = 0 to dim-1
        bits(esize) res = Elem[operand_acc, row*dim+col, esize];
        if ElemP[mask1, row, esize] == '1' && ElemP[mask2, col, esize] == '1' then
            res = res + element;
        Elem[result, row*dim+col, esize] = res;
ZAtile[da, esize, dim*dim*esize] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C9.2.2 ADDSPL

Add the Streaming SVE predicate register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer and place the result in the 64-bit destination general-purpose register or current stack pointer.

This instruction does not require the PE to be in Streaming SVE mode.

SME

(FEAT_SME)

[31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 13 12 11 10 ] | 5 4 | 0 |
0 0 0 0 0 1 0 0 0 1 1 | Rn 0 1 0 1 1 | imm6 | Rd

Encoding

ADDSPL <Xd|SP>, <Xn|SP>, #<imm>

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);

Assembler symbols

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

CheckSMEEnabled();
custom integer svl = SVL;
ninteger len = imm * (svl DIV 64);
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) result = operand1 + len;
if d == 31 then
  SP[] = result;
else
  X[d, 64] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
— The values of the NZCV flags.
C9.2.3 ADDSVL

Add the Streaming SVE vector register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer, and place the result in the 64-bit destination general-purpose register or current stack pointer.

This instruction does not require the PE to be in Streaming SVE mode.

SME (FEAT_SME)

| [31 30 29 28 27 26 25 24 23 22 21 20] | 16|15 14 13 12|11 10 | 5 4 | 0 |
|-----------------------------|-----------------------------|-----------------------------|-----------------------------|-----------------------------|-----------------------------|-----------------------------|
| 0 0 0 0 0 1 0 0 0 | 0 1 0 1 1 | imm6 | Rd |

Encoding

ADDVL <Xd|SP>, <Xn|SP>, #<imm>

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);

Assembler symbols

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

CheckSMEEnabled();
constant integer svl = SVL;
integer len = imm * (svl DIV 8);
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) result = operand1 + len;
if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
— The values of the NZCV flags.
C9.2.4   ADDVA

Add each element of the source vector to the corresponding active element of each vertical slice of a ZA tile. The tile elements are predicated by a pair of governing predicates. An element of a vertical slice is considered active if its corresponding element in the first governing predicate is TRUE and the element corresponding to its vertical slice number in the second governing predicate is TRUE. Inactive elements in the destination tile remain unmodified.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

32-bit

(FEAT_SME)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1</td>
<td>Pm</td>
<td>Pn</td>
<td>Zn</td>
<td>0 0</td>
<td>ZAda</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

ADDVA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S

Decode for this encoding

if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer da = UInt(ZAda);

64-bit

(FEAT_SME_I16I64)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 0 0 0 1 0 0 1 0 0 0 1</td>
<td>Pm</td>
<td>Pn</td>
<td>Zn</td>
<td>0 0</td>
<td>ZAda</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

ADDVA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D

Decode for this encoding

if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer da = UInt(ZAda);

Assembler symbols

<ZAda>  For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
        For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn>   Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation for all encodings**

```plaintext
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand_src = Z[n, VL];
bits(dim*dim*esize) operand_acc = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for row = 0 to dim-1
  bits(esize) element = Elem[operand_src, row, esize];
  for col = 0 to dim-1
    bits(esize) res = Elem[operand_acc, row*dim+col, esize];
    if ElemP[mask1, row, esize] == '1' && ElemP[mask2, col, esize] == '1' then
      res = res + element;
    Elem[result, row*dim+col, esize] = res;

ZAtile[da, esize, dim*dim*esize] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C9.2.5  BFMOPA

The BFloat16 floating-point sum of outer products and accumulate instruction works with a 32-bit element ZA tile.

This instruction multiplies the SVLS×2 sub-matrix of BFloat16 values held in the first source vector by the 2×SVLS sub-matrix of BFloat16 values in the second source vector.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting SVLS×SVLS single-precision floating-point sum of outer products is then destructively added to the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a SVLS×2 sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a 2×SVLS sub-matrix.

This instruction follows SME BFloat16 numerical behaviors.

SME  
(FEAT_SME)

Encoding  
BFMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

Decode for this encoding  
if !HaveSME() then UNDEFINED;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;

Assembler symbols  
<ZAda>  Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
<Pn>  Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>  Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation  
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV 32;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
bits(dim*dim*32) result;
for row = 0 to dim-1
  for col = 0 to dim-1
    // determine row/col predicates
    boolean prow_0 = (ElemP[mask1, 2*row + 0, 16] == '1');
    boolean prow_1 = (ElemP[mask1, 2*row + 1, 16] == '1');
    boolean pcol_0 = (ElemP[mask2, 2*col + 0, 16] == '1');
    boolean pcol_1 = (ElemP[mask2, 2*col + 1, 16] == '1');

    bits(32) sum = Elem[operand3, row*dim+col, 32];
    if (prow_0 & pcol_0) || (prow_1 & pcol_1) then
      bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0', 16));
      bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0', 16));
      bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0', 16));
      bits(16) ecol_1 = (if pcol_1 then Elem[operand2, 2*col + 1, 16] else FPZero('0', 16));
      if sub_op then
        if prow_0 then erow_0 = BFNeg(erow_0);
        if prow_1 then erow_1 = BFNeg(erow_1);
        sum = BFDotAdd(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR[]);
    Elem[result, row*dim+col, 32] = sum;
ZAtile[da, 32, dim*dim*32] = result;
C9.2.6 BFMOPS

The BFloat16 floating-point sum of outer products and subtract instruction works with a 32-bit element ZA tile. This instruction multiplies the SVLS×2 sub-matrix of BFloat16 values held in the first source vector by the 2×SVLS sub-matrix of BFloat16 values in the second source vector.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting SVLS×SVLS single-precision floating-point sum of outer products is then destructively subtracted from the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a SVLS×2 sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a 2×SVLS sub-matrix.

This instruction follows SME BFloat16 numerical behaviors.

SME

(FEAT_SME)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15</th>
<th>13 12</th>
<th>10 9</th>
<th>5</th>
<th>4</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zm Pm Pn Zn</td>
<td>1 0 0 0 0 0</td>
<td>1 1 0 0</td>
<td>ZAda 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

BFMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;

Assembler symbols

<ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV 32;
bits(PL) mask1 = P[a, PL];
\begin{verbatim}
bits(PL) mask2 = P[b, PL]; 
bits(VL) operand1 = Z[n, VL]; 
bits(VL) operand2 = Z[m, VL]; 
bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32]; 
bits(dim*dim*32) result;
for row = 0 to dim-1 
  for col = 0 to dim-1 
    // determine row/col predicates 
    boolean prow_0 = (ElemP[mask1, 2*row + 0, 16] == '1'); 
    boolean prow_1 = (ElemP[mask1, 2*row + 1, 16] == '1'); 
    boolean pcol_0 = (ElemP[mask2, 2*col + 0, 16] == '1'); 
    boolean pcol_1 = (ElemP[mask2, 2*col + 1, 16] == '1'); 
    bits(32) sum = Elem[operand3, row*dim+col, 32]; 
    if (prow_0 & pcol_0) || (prow_1 & pcol_1) then 
      bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0', 16)); 
      bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0', 16)); 
      bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0', 16)); 
      bits(16) ecol_1 = (if pcol_1 then Elem[operand2, 2*col + 1, 16] else FPZero('0', 16)); 
      if sub_op then 
        if prow_0 then erow_0 = BFNeg(erow_0); 
        if prow_1 then erow_1 = BFNeg(erow_1); 
      sum = BFDotAdd(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR[]); 
      Elem[result, row*dim+col, 32] = sum; 
  ZAtile[da, 32, dim*dim*32] = result;
\end{verbatim}
C9.2.7   FMOPA (non-widening)

The single-precision variant works with a 32-bit element ZA tile.

The double-precision variant works with a 64-bit element ZA tile.

These instructions generate an outer product of the first source vector and the second source vector. In case of the single-precision variant, the first source is SVL₅×1 vector and the second source is 1×SVL₅ vector. In case of the double-precision variant, the first source is SVL₆×1 vector and the second source is 1×SVL₆ vector.

Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is Inactive the corresponding destination tile element remains unmodified.

The resulting outer product, SVL₅×SVL₅ in case of single-precision variant or SVL₆×SVL₆ in case of double-precision variant, is then destructively added to the destination tile. This is equivalent to performing a single multiply-accumulate to each of the destination tile elements.

This instruction follows SME ZA-targeting floating-point behaviors.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented.

**Single-precision**

(FEAT_SME)

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] 16 | 13 12 | 10 9 | 5 4 | 3 2 1 0 | Zm Pm Pn Zn 0 0 0 ZAda
1 0 0 0 0 0 0 1 0 0
S
```

**Encoding**

FMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S, <Zm>.S

**Decode for this encoding**

if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;

**Double-precision**

(FEAT_SME_F64F64)

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] 16 | 13 12 | 10 9 | 5 4 | 3 2 0 | Zm Pm Pn Zn 0 0 ZAda
1 0 0 0 0 0 0 1 1 0
S
```

**Encoding**

FMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D, <Zm>.D

**Decode for this encoding**

if !HaveSMEF64F64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;

Assembler symbols

<ZAda> For the single-precision variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the double-precision variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation for all encodings

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) element1 = Elem[operand1, row, esize];
bits(esize) element2 = Elem[operand2, col, esize];
bits(esize) element3 = Elem[operand3, row+dim+col, esize];
        if ElemP[mask1, row, esize] == '1' & ElemP[mask2, col, esize] == '1'
            if sub_op then element1 = FPNeg(element1);
            Elem[result, row+dim+col, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
        else
            Elem[result, row+dim+col, esize] = element3;
        ZAtile[da, esize, dim*dim*esize] = result;
C9.2.8 FMOPA (widening)

The half-precision floating-point sum of outer products and accumulate instruction works with a 32-bit element ZA tile.

This instruction widens the SVLS×2 sub-matrix of half-precision floating-point values held in the first source vector to single-precision floating-point values and multiplies it by the widened 2×SVLS sub-matrix of half-precision floating-point values in the second source vector to single-precision floating-point values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting SVLS×SVLS single-precision floating-point sum of outer products is then destructively added to the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a SVLS×2 sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a 2×SVLS sub-matrix.

This instruction follows SME ZA-targeting floating-point behaviors.

SME

(Feat_SME)

Encoding

FMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;

Assembler symbols

<ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
Operation

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV 32;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
bits(dim*dim*32) result;
for row = 0 to dim-1
  for col = 0 to dim-1
    // determine row/col predicates
    boolean prow_0 = (Elem[mask1, 2*row + 0, 16] == '1');
    boolean prow_1 = (Elem[mask1, 2*row + 1, 16] == '1');
    boolean pcol_0 = (Elem[mask2, 2*col + 0, 16] == '1');
    boolean pcol_1 = (Elem[mask2, 2*col + 1, 16] == '1');

    bits(32) sum = Elem[operand3, row*dim+col, 32];
    if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
      bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0', 16));
      bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0', 16));
      bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0', 16));
      bits(16) ecol_1 = (if pcol_1 then Elem[operand2, 2*col + 1, 16] else FPZero('0', 16));
      if sub_op then
        if prow_0 then erow_0 = FPNeg(erow_0);
        if prow_1 then erow_1 = FPNeg(erow_1);
      sum = FPDotAdd_ZA(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR[]);

    Elem[result, row*dim+col, 32] = sum;
    ZAtile[da, 32, dim*dim*32] = result;
C9.2.9  FMOPS (non-widening)

The single-precision variant works with a 32-bit element ZA tile.

The double-precision variant works with a 64-bit element ZA tile.

These instructions generate an outer product of the first source vector and the second source vector. In case of the single-precision variant, the first source is SVLS×1 vector and the second source is 1×SVLS vector. In case of the double-precision variant, the first source is SVLD×1 vector and the second source is 1×SVLD vector.

Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is Inactive the corresponding destination tile element remains unmodified.

The resulting outer product, SVLS×SVLS in case of single-precision variant or SVLD×SVLD in case of double-precision variant, is then destructively subtracted from the destination tile. This is equivalent to performing a single multiply-subtract from each of the destination tile elements.

This instruction follows SME ZA-targeting floating-point behaviors.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented.

**Single-precision**  
(FEAT_SME)

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 13 12| 10 9 | 5 4 | 3 2 1 0 |  
1 0 0 0 0 0 0 1 0 0  | Zm  | Pm  | Pn  | Zn  | 1 0 0 ZAda

Encoding
FMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S, <Zm>.S

Decode for this encoding
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
```

**Double-precision**  
(FEAT_SME_F64F64)

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 13 12| 10 9 | 5 4 | 3 2 0 |  
1 0 0 0 0 0 0 1 1 0  | Zm  | Pm  | Pn  | Zn  | 1 0 ZAda

Encoding
FMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D, <Zm>.D

Decode for this encoding
if !HaveSMEF64F64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;

**Assembler symbols**

<ZAda> For the single-precision variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
For the double-precision variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation for all encodings**

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) element1 = Elem[operand1, row, esize];
    bits(esize) element2 = Elem[operand2, col, esize];
    bits(esize) element3 = Elem[operand3, row*dim+col, esize];
    if ElemP[mask1, row, esize] == '1' & ElemP[mask2, col, esize] == '1' then
      if sub_op then element1 = FPNeg(element1);
      Elem[result, row*dim+col, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
    else
      Elem[result, row*dim+col, esize] = element3;
  
ZAtile[da, esize, dim*dim*esize] = result;
C9.2.10 FMOPS (widening)

The half-precision floating-point sum of outer products and subtract instruction works with a 32-bit element ZA tile.

This instruction widens the SVLS×2 sub-matrix of half-precision floating-point values held in the first source vector to single-precision floating-point values and multiplies it by the widened 2×SVLS sub-matrix of half-precision floating-point values in the second source vector to single-precision floating-point values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting SVLS×SVLS single-precision floating-point sum of outer products is then destructively subtracted from the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a SVLS×2 sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a 2×SVLS sub-matrix.

This instruction follows SME ZA-targeting floating-point behaviors.

SME

(FEAT_SME)

Encoding

FMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;

Assembler symbols

<ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV 32;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
bits(dim*dim*32) result;

for row = 0 to dim-1
   for col = 0 to dim-1
      // determine row/col predicates
      boolean prow_0 = (Elem[mask1, 2*row + 0, 16] == '1');
      boolean prow_1 = (Elem[mask1, 2*row + 1, 16] == '1');
      boolean pcol_0 = (Elem[mask2, 2*col + 0, 16] == '1');
      boolean pcol_1 = (Elem[mask2, 2*col + 1, 16] == '1');

      bits(32) sum = Elem[operand3, row*dim+col, 32];
      if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
         bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0', 16));
         bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0', 16));
         bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0', 16));
         bits(16) ecol_1 = (if pcol_1 then Elem[operand2, 2*col + 1, 16] else FPZero('0', 16));
         if sub_op then
            if prow_0 then erow_0 = FPNeg(erow_0);
            if prow_1 then erow_1 = FPNeg(erow_1);
            sum = FPDotAdd_ZA(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR[]);
      Elem[result, row*dim+col, 32] = sum;
   ZAtile[da, 32, dim*dim*32] = result;
C9.2.11  LD1B

The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of 8-bit elements in a vector. The immediate is in the range 0 to 15. The memory address is generated by scalar base and optional scalar offset which is added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME

(FEAT_SME)

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 3 0 |
  1 1 1 0 0 0 0 0 | 0 0 0 | Rm V Rs Pg Rn 0 | imm4

msz<1>
msz<0>
```

Encoding

LD1B { ZA0<Hv>.B[<Ws>, <imm>] }, <Pg>/Z, [<Xn|SP>{, <Xm}>]

Decode for this encoding

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = 0;
integer imm = UInt(imm4);
constant integer esize = 8;
boolean vertical = V == '1';
```

Assembler symbols

<table>
<thead>
<tr>
<th>&lt;hv&gt;</th>
<th>Is the horizontal or vertical slice indicator, encoded in the &quot;V&quot; field. It can have the following values:</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>when V = 0</td>
</tr>
<tr>
<td>V</td>
<td>when V = 1</td>
</tr>
</tbody>
</table>

|<ws> | Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field. |

|<imm> | Is the slice index offset, in the range 0 to 15, encoded in the "imm4" field. |

|<pg> | Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field. |

|<xn|sp> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |

|<xm> | Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field. |

Operation

```
CheckStreamingSVEAndZAEnabled();
custom integer VLL = CurrentVL;
custom integer PLL = VL DIV 8;
custom integer dim = VL DIV esize;
```

bits(64) base;
bilts(64) addr;
bilts(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(32) index = X[s, 32];
integer slice = (UInt(index) + imm) MOD dim;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if AnyActiveElement(mask, esize) ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n, 64];
offset = X[m, 64];

for e = 0 to dim - 1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SME];
  else
    Elem[result, e, esize] = Zeros(esize);
  offset = offset + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
C9.2.12  LD1D

The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of 64-bit elements in a vector. The immediate is in the range 0 to 1. The memory address is generated by scalar base and optional scalar offset which is multiplied by 8 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME

(FEAT_SME)

Encoding

LD1D { <ZAt><HV>.D[<Ws>, <imm>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #3}]

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer imm = UInt(i1);
constant integer esize = 64;
boolean vertical = V == '1';

Assembler symbols

<ZAt>  Is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAt" field.

<HV>  Is the horizontal or vertical slice indicator, encoded in the "V" field. It can have the following values:

H when V = 0
V when V = 1

<Ws>  Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm>  Is the slice index offset, in the range 0 to 1, encoded in the "i1" field.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm>  Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(32) index = X[s, 32];
integer slice = (UInt(index) + imm) MOD dim;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if AnyActiveElement(mask, esize) ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
    base = SP[];
  else
    base = X[n, 64];
  offset = X[m, 64];

for e = 0 to dim - 1
  addr = base + UInt(offset) * mbytes;
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SME];
  else
    Elem[result, e, esize] = Zeros(esize);
  offset = offset + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
C9.2.13 LD1H

The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of 16-bit elements in a vector. The immediate is in the range 0 to 7. The memory address is generated by scalar base and optional scalar offset which is multiplied by 2 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME

(FEAT_SME)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>19 18 17 16 15 14 13 12</th>
<th>11 10 9</th>
<th>8 7 6 5 4 3 2 1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
<td>V</td>
<td>Rs</td>
<td>Pg</td>
<td>Rn</td>
</tr>
</tbody>
</table>

msz<1>
msz<0>  ZAt

Encoding

LD1H { <ZAt><HV>.H[<Ws>, <imm>], <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]}

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer imm = UInt(imm3);
constant integer esize = 16;
boolean vertical = V == '1';

Assembler symbols

<ZAt> Is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAt" field.

<VH> Is the horizontal or vertical slice indicator, encoded in the "V" field. It can have the following values:

H when V = 0
V when V = 1

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> Is the slice index offset, in the range 0 to 7, encoded in the "imm3" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
b bits(64) base;
bits(64) addr;
b bits(64) offset;
b bits(PL) mask = P[g, PL];
b bits(VL) result;
b bits(32) index = X[s, 32];
integer slice = (UInt(index) + imm) MOD dim;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if AnyActiveElement(mask, esize) ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n, 64];
offset = X[m, 64];

for e = 0 to dim - 1
  addr = base + UInt(offset) * mbytes;
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SME];
  else
    Elem[result, e, esize] = Zeros(esize);
  offset = offset + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
C9.2.14   LD1Q

The slice number in the tile is selected by the slice index register, modulo the number of 128-bit elements in a Streaming SVE vector. The memory address is generated by scalar base and optional scalar offset which is multiplied by 16 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME

(FEAT_SME)

Encoding

LD1Q { <ZAt><HV>.Q[<Ws>, <imm>], <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #4}] }

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer imm = 0;
constant integer esize = 128;
boolean vertical = V == '1';

Assembler symbols

<ZAt>   Is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAt" field.
<hv>   Is the horizontal or vertical slice indicator, encoded in the "V" field. It can have the following values:
   H   when V = 0
   V   when V = 1
<ws>   Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
<imm>   Is the slice index offset 0.
<p>   Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<xn|sp>   Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<xm>   Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bases(64) addr;
bases(64) offset;
bases(PL) mask = P[g, PL];
bits(VL) result;
bits(32) index = X[s, 32];
integer slice = (UInt(index) + imm) MOD dim;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if AnyActiveElement(mask, esize) ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = X[n, 64];
    offset = X[m, 64];
else
  base = SP[];
  offset = X[n, 64];

for e = 0 to dim - 1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SME];
  else
    Elem[result, e, esize] = Zeros(esize);
  offset = offset + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
C9.2.15 LD1W

The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of 32-bit elements in a vector. The immediate is in the range 0 to 3. The memory address is generated by scalar base and optional scalar offset which is multiplied by 4 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

**SME**

(FEAT_SME)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rm</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>10</td>
<td>9</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

msz<1>
msz<0>

**Encoding**

LD1W { <ZAt><HV>.S[<Ws>, <imm>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]

**Decode for this encoding**

if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer imm = UInt(imm2);
constant integer esize = 32;
boolean vertical = V == '1';

**Assembler symbols**

- `<ZAt>` Is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAt" field.
- `<HV>` Is the horizontal or vertical slice indicator, encoded in the "V" field. It can have the following values:
  - H when V = 0
  - V when V = 1
- `<Ws>` Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- `<imm>` Is the slice index offset, in the range 0 to 3, encoded in the "imm2" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

**Operation**

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(32) index = X[s, 32];
integer slice = (UInt(index) + imm) MOD dim;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if AnyActiveElement(mask, esize) ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
    base = SP[];
  else
    base = X[n, 64];
offset = X[m, 64];

for e = 0 to dim - 1
  addr = base + UInt(offset) * mbytes;
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SME];
  else
    Elem[result, e, esize] = Zeros(esize);
  offset = offset + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
C9.2.16 LDR

The ZA array vector is selected by the sum of the vector select register and an immediate, modulo the number of bytes in a Streaming SVE vector. The immediate is in the range 0 to 15. The memory address is generated by scalar base, plus the same optional immediate offset multiplied by the current vector length in bytes. This instruction is unpredicated.

The load is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME

(FEAT_SME)

Encoding

LDR ZA[<Wv>, <imm>], [<Xn|SP>{, #<imm>, MUL VL}]

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer v = UInt('011':Rv);
integer imm = UInt(imm4);

Assembler symbols

<Wv> Is the 32-bit name of the vector select register W12-W15, encoded in the "Rv" field.
<imm> Is the vector select offset and optional memory offset, in the range 0 to 15, defaulting to 0, encoded in the "imm4" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

CheckSMEAndZAEnabled();
constant integer svl = SVL;
constant integer dim = svl DIV 8;
bits(64) base;
integer offset = imm * dim;
bits(svl) result;
bits(32) idx = X[v, 32];
integer vec = (UInt(idx) + imm) MOD dim;
if HaveTME() && TSTATE.depth > 0 then
  FailTransaction(TMFailure_ERR, FALSE);
if n == 31 then
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  CheckSPAlignment();
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n, 64];
boolean aligned = AArch64.CheckAlignment(base + offset, 16, AccType_SME, FALSE);
for e = 0 to dim-1
    Elem[result, e, 8] = AArch64.MemSingle(base + offset, 1, AccType_SME, aligned);
    offset = offset + 1;

ZAvector[vec, svl] = result;
C9.2.17 MOV (tile to vector)

The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of such elements in a vector. The immediate is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

Inactive elements in the destination vector remain unmodified.

This instruction is an alias of the MOVA (tile to vector) instruction. This means that:

- The encodings in this description are named to match the encodings of MOVA (tile to vector).
- The description of MOVA (tile to vector) gives the operational pseudocode for this instruction.

### 8-bit

\[
\text{MOV} \quad \text{<Zd>.B, <Pg>/M, ZA0<HV>.B[<Ws>, <imm>]}
\]

is equivalent to

\[
\text{MOVA} \quad \text{<Zd>.B, <Pg>/M, ZA0<HV>.B[<Ws>, <imm>]}
\]

and is always the preferred disassembly.

### 16-bit

\[
\text{MOV} \quad \text{<Zd>.H, <Pg>/M, <ZAn><HV>.H[<Ws>, <imm>]}
\]

is equivalent to

\[
\text{MOVA} \quad \text{<Zd>.H, <Pg>/M, <ZAn><HV>.H[<Ws>, <imm>]}
\]

and is always the preferred disassembly.

### 32-bit

\[
\text{MOV} \quad \text{<Zd>.H, <Pg>/M, <ZAn><HV>.H[<Ws>, <imm>]}
\]

is equivalent to

\[
\text{MOVA} \quad \text{<Zd>.H, <Pg>/M, <ZAn><HV>.H[<Ws>, <imm>]}
\]

and is always the preferred disassembly.
**Encoding**

MOV <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <imm>]

is equivalent to

MOVA <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <imm>]

and is always the preferred disassembly.

**64-bit**

64-bit Encoding

MOV <Zd>.D, <Pg>/M, <ZAn><HV>.D[<Ws>, <imm>]

is equivalent to

MOVA <Zd>.D, <Pg>/M, <ZAn><HV>.D[<Ws>, <imm>]

and is always the preferred disassembly.

**128-bit**

128-bit Encoding

MOV <Zd>.Q, <Pg>/M, <ZAn><HV>.Q[<Ws>, <imm>]

is equivalent to

MOVA <Zd>.Q, <Pg>/M, <ZAn><HV>.Q[<Ws>, <imm>]

and is always the preferred disassembly.

**Assembler symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<ZAn>` For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.
  - For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.
  - For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.
  - For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAn" field.
<HV> Is the horizontal or vertical slice indicator, encoded in the "V" field. It can have the following values:
H when V = 0
V when V = 1

<W> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "imm4" field.
For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "imm3" field.
For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "imm2" field.
For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "i1" field.
For the 128-bit variant: is the slice index offset 0.

**Operation for all encodings**

The description of MOV A (tile to vector) gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
### C9.2.18 MOV (vector to tile)

The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of such elements in a vector. The immediate is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

Inactive elements in the destination slice remain unmodified.

This instruction is an alias of the MOVA (vector to tile) instruction. This means that:

- The encodings in this description are named to match the encodings of MOVA (vector to tile).
- The description of MOVA (vector to tile) gives the operational pseudocode for this instruction.

#### 8-bit

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | | 5 | 4 | 3 | 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-------|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | V | Rs | Pg | Zn | 0 | imm4 |

**Encoding**

```
MOV ZA0<hv>.B[<w1s>, <imm>], <Pg>/M, <Zn>.B
```

is equivalent to

```
MOVA    ZA0<hv>.B[<w1s>, <imm>], <Pg>/M, <Zn>.B
```

and is always the preferred disassembly.

#### 16-bit

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | | 5 | 4 | 3 | 2 | 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-------|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | V | Rs | Pg | Zn | 0 | imm3 |

**Encoding**

```
MOV <ZAd><hv>.H[<w1s>, <imm>], <Pg>/M, <Zn>.H
```

is equivalent to

```
MOVA    <ZAd><hv>.H[<w1s>, <imm>], <Pg>/M, <Zn>.H
```

and is always the preferred disassembly.

#### 32-bit

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 | | 5 | 4 | 3 | 2 | 1 | 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-------|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | V | Rs | Pg | Zn | 0 | ZAd | imm2 |

**Encoding**

```
MOV <ZAd><hv>.H[<w1s>, <imm>], <Pg>/M, <Zn>.H
```

is equivalent to

```
MOVA    <ZAd><hv>.H[<w1s>, <imm>], <Pg>/M, <Zn>.H
```

and is always the preferred disassembly.
Encoding

MOV <ZAd><HV>.S[<Ws>, <imm>], <Pg>/M, <Zn>.S

is equivalent to

MOVA <ZAd><HV>.S[<Ws>, <imm>], <Pg>/M, <Zn>.S

and is always the preferred disassembly.

64-bit

\[
\begin{array}{cccccccccccccccccc}
|31|30|29|28|27|26|25|24|23|22|21|20|19|18|17|16|15|14|13|12|10|9|5|4|3|1|0|
\end{array}
\]

size<1> size<0>

Encoding

MOV <ZAd><HV>.D[<Ws>, <imm>], <Pg>/M, <Zn>.D

is equivalent to

MOVA <ZAd><HV>.D[<Ws>, <imm>], <Pg>/M, <Zn>.D

and is always the preferred disassembly.

128-bit

\[
\begin{array}{cccccccccccccccccc}
|31|30|29|28|27|26|25|24|23|22|21|20|19|18|17|16|15|14|13|12|10|9|5|4|3|0|
\end{array}
\]

size<1> size<0>

Encoding

MOV <ZAd><HV>.Q[<Ws>, <imm>], <Pg>/M, <Zn>.Q

is equivalent to

MOVA <ZAd><HV>.Q[<Ws>, <imm>], <Pg>/M, <Zn>.Q

and is always the preferred disassembly.

Assembler symbols

\(<\text{ZAd}\rangle\) For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.

For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.

For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAd" field.
<HV> Is the horizontal or vertical slice indicator, encoded in the "V" field. It can have the following values:
H when V = 0
V when V = 1

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "imm4" field.
For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "imm3" field.
For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "imm2" field.
For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "i1" field.
For the 128-bit variant: is the slice index offset 0.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation for all encodings**

The description of MOV A (vector to tile) gives the operational pseudocode for this instruction.

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C9.2.19 MOVA (tile to vector)

The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of such elements in a vector. The immediate is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

Inactive elements in the destination vector remain unmodified.

This instruction is used by the alias MOV (tile to vector). The alias is always the preferred disassembly.

8-bit

(FEAT_SME)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9 8</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 0 0 0</td>
<td>0 0 0 0 1 0</td>
<td>V</td>
<td>Rs</td>
<td>Pg</td>
<td>0</td>
<td>imm4</td>
<td>Zd</td>
</tr>
</tbody>
</table>

**Encoding**

MOVA <Zd>.B, <Pg>/M, ZA0<HV>.B[<Ws>], <imm>

**Decode for this encoding**

if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = 0;
integer imm = UInt(imm4);
countant integer esize = 8;
integer d = UInt(Zd);
boolean vertical = V == '1';

16-bit

(FEAT_SME)

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 8 7 5 4 | 0 |
|-----------|-----------|-----------|-----------|-----------|-----|-----|-----|-----|
| 1 1 0 0 0 0 0 0 | 0 1 0 0 0 0 1 | 0 | V | Rs | Pg | 0 | imm3 | Zd |

**Encoding**

MOVA <Zd>.H, <Pg>/M, <ZAn><HV>.H[<Ws>], <imm>

**Decode for this encoding**

if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer imm = UInt(imm3);
countant integer esize = 16;
integer d = UInt(Zd);
boolean vertical = V == '1';

32-bit

(FEAT_SME)

[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 8 | 7 6 5 4 | 0 ]
| 1 1 0 0 0 0 0 0 | 1 0 0 0 0 0 1 0 | V | Rs | Pg | 0 | ZAn | imm2 | Zd |

size<1>                         Q
size<0>                         

Encoding

MOVA <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <imm>]

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer imm = UInt(imm2);
constant integer esize = 32;
integer d = UInt(Zd);
boolean vertical = V == '1';

64-bit

(FEAT_SME)

[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9 8 | 6 5 4 | 0 ]
| 1 1 0 0 0 0 0 0 | 1 1 0 0 0 0 1 0 | V | Rs | Pg | 0 | ZAn | i1 | Zd |

size<1>                         Q
size<0>                         

Encoding

MOVA <Zd>.D, <Pg>/M, <ZAn><HV>.D[<Ws>, <imm>]

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer imm = UInt(i1);
constant integer esize = 64;
integer d = UInt(Zd);
boolean vertical = V == '1';
## 128-bit

(FEAT_SME)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 10 | 9 | 8 | 5 | 4 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | V  | Rs | 0  | ZAn | Zd |

size<1>
size<0>

### Encoding

MOVA <Zd>.Q, <Pg>/M, <ZAn>.<HV>.Q[<Ws>, <imm>]

### Decode for this encoding

if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt("011":Rs);
integer n = UInt(ZAn);
integer imm = 0;
constant integer esize = 128;
integer d = UInt(Zd);
boolean vertical = V == '1';

### Assembler symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<ZAn>` For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.
  For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.
  For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.
  For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAn" field.
- `<HV>` Is the horizontal or vertical slice indicator, encoded in the "V" field. It can have the following values:
  - H when V = 0
  - V when V = 1
- `<Ws>` Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- `<imm>` For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "imm4" field.
  For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "imm3" field.
  For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "imm2" field.
  For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "i1" field.
  For the 128-bit variant: is the slice index offset 0.
**Operation for all encodings**

```plaintext
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + imm) MOD dim;
bits(VL) operand = ZAslice[n, esize, vertical, slice, VL];
bits(VL) result = Z[d, VL];
for e = 0 to dim-1
  bits(esize) element = Elem[operand, e, esize];
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = element;
Z[d, VL] = result;
```

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C9.2.20 MOVA (vector to tile)

The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of such elements in a vector. The immediate is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

Inactive elements in the destination slice remain unmodified.

This instruction is used by the alias MOV (vector to tile). The alias is always the preferred disassembly.

8-bit
(FEATURE_SME)

MOVA ZA0<HV>.B[<dis>], <imm>, <Pg>/M, <Zn>.B

Encoding

MOVA ZA0<HV>.B[<dis>], <imm>, <Pg>/M, <Zn>.B

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = 0;
integer imm = UInt(imm4);
constant integer esize = 8;
boolean vertical = V == '1';

16-bit
(FEATURE_SME)

MOVA <ZAd><HV>.H[<dis>], <imm>, <Pg>/M, <Zn>.H

Encoding

MOVA <ZAd><HV>.H[<dis>], <imm>, <Pg>/M, <Zn>.H

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = UInt(ZAd);
integer imm = UInt(imm3);
constant integer esize = 16;
boolean vertical = V == '1';

32-bit

(FEAT_SME)

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9  | 5 4 3 2 1 0 |
|-----------|----------|-----------|----------|----------|--------|
| 1 1 0 0 0 0 0 0 | 1 0 0 0 0 0 0 0 | V Rs Pg Zn 0 ZAd imm2 |

Encoding

MOVA <ZAd><HV>.S[<Ws>, <imm>], <Pg>/M, <Zn>.S

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = UInt(ZAd);
integer imm = UInt(imm2);
constant integer esize = 32;
boolean vertical = V == '1';

64-bit

(FEAT_SME)

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12| 10 9  | 5 4 3 2 1 0 |
|-----------|----------|-----------|----------|----------|--------|
| 1 1 0 0 0 0 0 0 | 1 1 0 0 0 0 0 0 | V Rs Pg Zn 0 ZAd |1 |

Encoding

MOVA <ZAd><HV>.D[<Ws>, <imm>], <Pg>/M, <Zn>.D

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = UInt(ZAd);
integer imm = UInt(i1);
constant integer esize = 64;
boolean vertical = V == '1';
128-bit

(FEAT_SME)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12</th>
<th>10 9</th>
<th>5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 0 0 0 1</td>
<td>1 0 0 0 0 0 0 1</td>
<td>V</td>
<td>Rs</td>
<td>Pg</td>
</tr>
<tr>
<td>size&lt;1&gt;</td>
<td>size&lt;0&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

MOVA <ZAd><HV>.Q[<Ws>], <imm>, <Pg>/M, <Zn>.Q

**Decode for this encoding**

if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = UInt(ZAd);
integer imm = 0;
constant integer esize = 128;
boolean vertical = V == '1';

**Assembler symbols**

<ZAd> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.
For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAd" field.

<HV> Is the horizontal or vertical slice indicator, encoded in the "V" field. It can have the following values:
H when V = 0
V when V = 1

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "imm4" field.
For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "imm3" field.
For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "imm2" field.
For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "i1" field.
For the 128-bit variant: is the slice index offset 0.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
Operation for all encodings

`CheckStreamingSVEAndZAEnabled();`
`constant integer VL = CurrentVL;`
`constant integer PL = VL DIV 8;`
`constant integer dim = VL DIV esize;`
`bits(PL) mask = P[g, PL];`
`bits(VL) operand = Z[n, VL];`
`bits(32) index = X[s, 32];`
`integer slice = (UInt(index) + imm) MOD dim;`
`bits(VL) result = ZAslice[d, esize, vertical, slice, VL];`

for e = 0 to dim-1
  bits(esize) element = Elem[operand, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = element;

`ZAslice[d, esize, vertical, slice, VL] = result;`

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C9.2.21   RDSVL

Multiply the Streaming SVE vector register size in bytes by an immediate in the range -32 to 31 and place the result in the 64-bit destination general-purpose register.

This instruction does not require the PE to be in Streaming SVE mode.

SME

(FEAT_SME)

| 0 0 0 0 0 1 0 0 1 | 1 1 1 1 | 1 0 1 0 1 1 | imm6 | Rd |

Encoding

RDSVL <Xd>, #<imm>

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer d = UInt(Rd);
integer imm = SInt(imm6);

Assembler symbols

Xd Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.
<imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

CheckSMEEnabled();
constant integer svl = SVL;
integer len = imm \* (svl DIV 8);
X[d, 64] = len<63:0>;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
C9.2.22   SMOPA

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $\text{SVLS} \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times \text{SVLS}$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $\text{SVLD} \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $4 \times \text{SVLD}$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting $\text{SVLS} \times \text{SVLS}$ widened 32-bit integer or $\text{SVLD} \times \text{SVLD}$ widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $\text{SVLS} \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times \text{SVLS}$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $\text{SVLD} \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times \text{SVLD}$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

32-bit

(FEAT_SME)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 13 12</th>
<th>10 9</th>
<th>5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>u0</td>
<td>u1</td>
<td>Zm</td>
<td>Pm</td>
</tr>
</tbody>
</table>

Encoding

SMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

Decode for this encoding

if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;
64-bit

(FEAT_SME_I16I64)

<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20]</th>
<th>16</th>
<th>15</th>
<th>13</th>
<th>12</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zm</td>
<td>Pn</td>
<td>Pm</td>
<td>Zn</td>
<td>0</td>
<td>0</td>
<td>ZAda</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u0 u1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

SMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

Decode for this encoding

if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;

Assembler symbols

<ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation for all encodings

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 3
      if Elem[mask1, 4*row + k, esize DIV 4] == '1' &&
         Elem[mask2, 4*col + k, esize DIV 4] == '1' then
        prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                    Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
        if sub_op then prod = -prod;
        sum = sum + prod;
\[ \text{Elem}\{\text{result, row+dim+col, esize}\} = \text{sum}; \]
\[ \text{ZAtile}\{\text{da, esize, dim=dim=esize}\} = \text{result}; \]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C9.2.23 SMOPS

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds SVLS×4 sub-matrix of signed 8-bit integer values, and the second source holds 4×SVLS sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds SVLĐ×4 sub-matrix of signed 16-bit integer values, and the second source holds 4×SVLĐ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting SVLS×SVLS widened 32-bit integer or SVLĐ×SVLĐ widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a SVLS×4 sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a 4×SVLS sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a SVLĐ×4 sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a 4×SVLĐ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

32-bit

(FEAT_SME)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 13 12</th>
<th>10 9</th>
<th>5 4</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>u0</td>
<td>u1</td>
<td>Zm</td>
<td>Pm</td>
<td>Pn</td>
<td>Zn</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Encoding

SMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

Decode for this encoding

if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;
64-bit

(FEAT_SME_I16I64)

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZAda</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

SMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

**Decode for this encoding**

if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;

**Assembler symbols**

<ZAda>  For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn>    Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm>    Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn>    Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>    Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation for all encodings**

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;
for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 3
      if Elem[mask1, 4*row + k, esize DIV 4] == '1' &&
         Elem[mask2, 4*col + k, esize DIV 4] == '1' then
        prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
        if sub_op then prod = -prod;
        sum = sum + prod;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C9.2.24   ST1B

The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of 8-bit elements in a vector. The immediate is in the range 0 to 15. The memory address is generated by scalar base and optional scalar offset which is added to the base address. Inactive elements are not written to memory.

SME
(FEAT_SME)

Encoding

\[ \text{ST1B} \{ \text{ZA0}<HV>.B [<Ws>, <imm>] }, \langle Pg \rangle, \langle Xn|SP>\{, <Xm>\} \]

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = 0;
integer imm = UInt(imm4);
constant integer esize = 8;
boolean vertical = V == '1';

Assembler symbols

\(<HV>\) Is the horizontal or vertical slice indicator, encoded in the "V" field. It can have the following values:
H    when V = 0
V    when V = 1

\(<Ws>\) Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

\(<imm>\) Is the slice index offset, in the range 0 to 15, encoded in the "imm4" field.

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<Xm>\) Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
SME Instruction Descriptions
C9.2 Alphabetical list of SME instructions

bits(64) offset = X[m, 64];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + imm) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;

if HaveMTEEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if AnyActiveElement(mask, esize) ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[ ];
else
  base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_SME] = Elem[src, e, esize];
  offset = offset + 1;
C9.2.25  **ST1D**

The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of 64-bit elements in a vector. The immediate is in the range 0 to 1. The memory address is generated by scalar base and optional scalar offset which is multiplied by 8 and added to the base address. Inactive elements are not written to memory.

**SME**

(FEAT_SME)

```
[31 30 29 28|27 26 25 24|23 22 21 20] | 16|15 14 13 12| 10 9 | 5 4 | 3 | 1 | 0 |
  1 1 1 0 0 0 0 0 1 1 1 | Rm | V | Rs | Pg | Rn | 0 | ZAt | i1
```

**Encoding**

ST1D { <ZAt><HV>.D[<Ws>,<imm>],<Pg>,[<Xn|SP>{,<Xm>,LSL #3}]}

**Decode for this encoding**

```c
def decode_ST1D(const uint8_t *instruction, const SMEInstruction &inst, const Register &reg)
{
    if (!HaveSME())
        return UNDEFINED;

    integer n = UInt(Rn);
    integer m = UInt(Rm);
    integer g = UInt('0':Pg);
    integer s = UInt('011':Rs);
    integer t = UInt(ZAt);
    integer imm = UInt(i1);
    constant integer esize = 64;
    boolean vertical = V == '1';
```

**Assembler symbols**

- `<ZAt>`  Is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAt" field.
- `<HV>`  Is the horizontal or vertical slice indicator, encoded in the "V" field. It can have the following values:
  - `H` when `V = 0`
  - `V` when `V = 1`
- `<Ws>`  Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- `<imm>`  Is the slice index offset, in the range 0 to 1, encoded in the "i1" field.
- `<Pg>`  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>`  Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

**Operation**

```c
    CheckStreamingSVEAndZAEnabled();
    constant integer VL = CurrentVL;
    constant integer PL = VL DIV 8;
    constant integer dim = VL DIV esize;
    bits(64) base;
```
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) offset = X[m, 64];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + imm) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;

if HaveMTEEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if AnyActiveElement(mask, esize) ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
  addr = base + UInt(offset) + mbytes;
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_SME] = Elem[src, e, esize];
  offset = offset + 1;
C9.2.26  **ST1H**

The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of 16-bit elements in a vector. The immediate is in the range 0 to 7. The memory address is generated by scalar base and optional scalar offset which is multiplied by 2 and added to the base address. Inactive elements are not written to memory.

**SME**

*(FEAT_SME)*

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>10 9</th>
<th>5 4 3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 0 0 0</td>
<td>0 1 1</td>
<td>Rm</td>
<td>V</td>
<td>Rs</td>
<td>Pg</td>
<td>Rn</td>
<td>0</td>
<td>imm3</td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

ST1H { <ZAt><HV>.H[<Ws>, <imm>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #1}]

**Decode for this encoding**

if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer imm = UInt(imm3);
constant integer esize = 16;
boolean vertical = V == '1';

**Assembler symbols**

- `<ZAt>` Is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAt" field.
- `<HV>` Is the horizontal or vertical slice indicator, encoded in the "V" field. It can have the following values:
  - H when V = 0
  - V when V = 1
- `<Ws>` Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- `<imm>` Is the slice index offset, in the range 0 to 7, encoded in the "imm3" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

**Operation**

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) offset = X[m, 64];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + imm) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
   if AnyActiveElement(mask, esize) ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
   else
      base = SP[];
else
   base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
   addr = base + UInt(offset) * mbytes;
   if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_SME] = Elem[src, e, esize];
   offset = offset + 1;
C9.2.27 ST1Q

The slice number in the tile is selected by the slice index register, modulo the number of 128-bit elements in a Streaming SVE vector. The memory address is generated by scalar base and optional scalar offset which is multiplied by 16 and added to the base address. Inactive elements are not written to memory.

SME
(FEAT_SME)

| 31 30 29 28 27 26 25 24 23 22 21 20 | 16 15 14 13 12 10 9 5 4 3 0 |
| 1 1 1 0 0 0 1 1 1 | Rm V Rs Pg Rn 0 ZAt |

Encoding

ST1Q { <ZAt><HV>.Q[<Ws>, <imm>], <Pg>, [<Xn|SP>{, <Xm>, LSL #4}]}

Decode for this encoding

if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer imm = 0;
constant integer esize = 128;
boolean vertical = V == '1';

Assembler symbols

<ZAt> Is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator, encoded in the "V" field. It can have the following values:

H when V = 0
V when V = 1

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> Is the slice index offset 0.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) offset = X[m, 64];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + imm) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if AnyActiveElement(mask, esize) ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_SME] = Elem[src, e, esize];
  offset = offset + 1;
C9.2.28 ST1W

The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of 32-bit elements in a vector. The immediate is in the range 0 to 3. The memory address is generated by scalar base and optional scalar offset which is multiplied by 4 and added to the base address. Inactive elements are not written to memory.

SME

(FEAT_SME)

```
[31 30 29 28|27 26 25 24|23 22 21 20] 16|15 14 13 12| 10 9 | 5 4 3 2 1 0 |
 1 1 1 0 0 0 0 0 | 1 0 1   Rm | V  Rs | Pg | Rn  0 | ZAt | imm2
```

Encoding

ST1W { <ZAt>,<HV>,S[<Ws>, <imm>], <Pg>, [<Xn|SP>{, <Xm>, LSL #2}] }

Decode for this encoding

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer imm = UInt(imm2);
constant integer esize = 32;
boolean vertical = V == '1';
```

Assembler symbols

- `<ZAt>` Is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAt" field.
- `<HV>` Is the horizontal or vertical slice indicator, encoded in the "V" field. It can have the following values:
  - H when V = 0
  - V when V = 1
- `<Ws>` Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- `<imm>` Is the slice index offset, in the range 0 to 3, encoded in the "imm2" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
```
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) offset = X[m, 64];
bits(32) index = X[s, 32];
integer slice = (UInt(index) + imm) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if AnyActiveElement(mask, esize) ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_SME] = Elem[src, e, esize];
  offset = offset + 1;
C9.2.29   STR

The ZA array vector is selected by the sum of the vector select register and an immediate, modulo the number of bytes in a Streaming SVE vector. The immediate is in the range 0 to 15. The memory address is generated by scalar base, plus the same optional immediate offset multiplied by the current vector length in bytes. This instruction is unpredicated.

The store is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

**SME**

*(FEAT_SME)*

**Encoding**

\[
\text{STR } ZA[<Wv>, <imm>], [<Xn|SP>{, #<imm>, MUL VL}]
\]

**Decode for this encoding**

\[
\text{if } \neg \text{HaveSME()} \text{ then UNDEFINED;}
\]

\[
\text{integer } n = \text{UInt}(Rn);
\]

\[
\text{integer } v = \text{UInt}('011':Rv);
\]

\[
\text{integer } imm = \text{UInt}(imm4);
\]

**Assembler symbols**

\(<Wv>\) Is the 32-bit name of the vector select register W12-W15, encoded in the "Rv" field.

\(<imm>\) Is the vector select offset and optional memory offset, in the range 0 to 15, defaulting to 0, encoded in the "imm4" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

\[
\text{CheckSMEAndZAEnabled();}
\]

\[
\text{constant integer } svl = \text{SVL};
\]

\[
\text{constant integer } dim = \text{svl} \text{ DIV } 8;
\]

\[
\text{bits(32)} \text{ idx} = X[v, 32];
\]

\[
\text{integer vec} = (\text{UInt}(\text{idx}) + \text{imm}) \text{ MOD dim};
\]

\[
\text{bits(svl)} \text{ src};
\]

\[
\text{bits(64)} \text{ base};
\]

\[
\text{integer offset} = \text{imm} \times \text{dim};
\]

\[
\text{if } \text{HaveTME()} \&\& \text{TSTATE.depth} > 0 \text{ then}
\]

\[
\text{FailTransaction(TMFailure_ERR, FALSE);}
\]

\[
\text{if } n == 31 \text{ then}
\]

\[
\text{if } \text{HaveMTEExt()} \text{ then SetTagCheckedInstruction(FALSE);}
\]

\[
\text{CheckSPAlignment();}
\]

\[
\text{base} = \text{SP}[];
\]

\[
\text{else}
\]

\[
\text{if } \text{HaveMTEExt()} \text{ then SetTagCheckedInstruction(TRUE);}
\]

\[
\text{base} = X[n, 64];
\]
src = ZAvector[vec, svl];
boolean aligned = AArch64.CheckAlignment(base + offset, 16, AccType_SME, TRUE);
for e = 0 to dim-1
    AArch64.MemSingle[base + offset, 1, AccType_SME, aligned] = Elem[src, e, 8];
    offset = offset + 1;
C9.2.30  SUMOPA

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed by unsigned integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds SVLS×4 sub-matrix of signed 8-bit integer values, and the second source holds 4×SVLS sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds SVLD×4 sub-matrix of signed 16-bit integer values, and the second source holds 4×SVLD sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting SVLS×SVLS widened 32-bit integer or SVLD×SVLD widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a SVLS×4 sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a 4×SVLS sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a SVLD×4 sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a 4×SVLD sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

32-bit

(FEAT_SME)

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 16 | 15 13 12 | 10 9 | 5 4 | 3 2 1 0 |
|--------------|--------------|--------------|-----|--------|-----|-----|-----|-----|
| 1 0 1 0 0 0 0 0 | 0 1 0 1 | Zm | Pm | Pn | Zn | 0 0 0 | ZAda |
| u0 | u1 | S |

Encoding

SUMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

Decode for this encoding

if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = TRUE;
64-bit

(_FEAT_SME_I16I64)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 16 | 15 | 13 | 12 | 10 | 9 | 5 | 4 | 3 | 2 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  |    |    |    |    |    |    |    |    |    |    |
|    | Zm | Pm | Pn | Zn | 0  | 0  | 0  | ZAda |
| u0 | u1 | S  |    |    |    |    |    |    |

**Encoding**

SUMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

**Decode for this encoding**

if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = TRUE;

**Assembler symbols**

<ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation for all encodings**

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
for col = 0 to dim-1
bits(esize) sum = Elem[operand3, row*dim+col, esize];
for k = 0 to 3
if ElemP[mask1, 4*row + k, esize DIV 4] == '1' &&
ElemP[mask2, 4*col + k, esize DIV 4] == '1' then
prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
if sub_op then prod = -prod;
sum = sum + prod;

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C9.2.31 SUMOPS

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed by unsigned integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds SVLS×4 sub-matrix of signed 8-bit integer values, and the second source holds 4×SVLS sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds SVLD×4 sub-matrix of signed 16-bit integer values, and the second source holds 4×SVLD sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting SVLS×SVLS widened 32-bit integer or SVLD×SVLD widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a SVLS×4 sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a 4×SVLS sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a SVLD×4 sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a 4×SVLD sub-matrix.

ID_AA64SMFR0_EL1.116164 indicates whether the 16-bit integer variant is implemented.

32-bit

(FEAT_SME)

```
[31 30 29 28 27 26 25 24 23 22 21 20 16 15 13 12 10 9 8 7 6 5 4 3 2 1 0]
|   1 0 1 0 0 0 0 0 0 1 0 1 |
| u0 | Zm | Pm | Pn | Zn |
```

```
| 1 0 0 ZAda |
| S          |
```

**Encoding**

SUMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

**Decode for this encoding**

```c
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = TRUE;
```
64-bit

(FEAT_SME_I16I64)

```
| 31 30 29 28| 27 26 25 24| 23 22 21 20 | 16|15 | 13|12 | 10|9 | 5|4|3|2 |0 |
|------------------|------------------|------------------|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | Zm | Pm | Pn | Zn | 1 | 0 | ZAda |

u0 u1 S
```

**Encoding**

SUMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

**Decode for this encoding**

```c
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = TRUE;
```

**Assembler symbols**

- `<ZAda>`: For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
  
  For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

- `<Pn>`: Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

- `<Pm>`: Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

- `<Zn>`: Is the name of the first source scalable vector register, encoded in the "Zn" field.

- `<Zm>`: Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation for all encodings**

```c
CheckStreamingSVEAndZAEnabled();
custom integer VL = CurrentVL;
custom integer PL = VL DIV 8;
custom integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 3
      if ElemP[mask1, 4*row + k, esize DIV 4] == '1' &&
         ElemP[mask2, 4*col + k, esize DIV 4] == '1' then
        prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
        if sub_op then prod = -prod;
        sum = sum + prod;
```
lli = size;  

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds \( \text{SVL}_8 \times 4 \) sub-matrix of unsigned 8-bit integer values, and the second source holds \( 4 \times \text{SVL}_8 \) sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds \( \text{SVL}_D \times 4 \) sub-matrix of unsigned 16-bit integer values, and the second source holds \( 4 \times \text{SVL}_D \) sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting \( \text{SVL}_8 \times \text{SVL}_8 \) widened 32-bit integer or \( \text{SVL}_D \times \text{SVL}_D \) widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a \( \text{SVL}_8 \times 4 \) sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a \( 4 \times \text{SVL}_8 \) sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a \( \text{SVL}_D \times 4 \) sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a \( 4 \times \text{SVL}_D \) sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

**32-bit**

**(FEAT_SME)**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 13 12</th>
<th>10 9</th>
<th>5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 0</td>
<td>1 1 0 1</td>
<td>Zn</td>
<td>Pm</td>
<td>Pn</td>
<td>Zm</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>
```

**Encoding**

UMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

**Decode for this encoding**

```c
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```
**64-bit**

*(FEAT_SME_I16I64)*

```
[31 30 29 28 27 26 25 24 23 22 21 20] 16 | 15 13 12 | 10 9 |  8 7 6 5 4 3 2 0 | u0 u1 u2 u3 u4 u5 u6 u7
| 1 0 1 0 0 | 0 | 0 1 1 1 | Zm | Pm | Pn | Zn | 0 0 | ZAda |
```

**Encoding**

UMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

**Decode for this encoding**

```plaintext
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```

**Asmbluer symbols**

- **<ZAda>**
  For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
  For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

- **<Pn>**
  Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

- **<Pm>**
  Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

- **<Zn>**
  Is the name of the first source scalable vector register, encoded in the "Zn" field.

- **<Zm>**
  Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation for all encodings**

```
CheckStreamingSVEAndZAEenabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bins(PL) mask2 = P[b, PL];
bins(VL) operand1 = Z[n, VL];
bins(VL) operand2 = Z[m, VL];
bins(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bins(dim*dim*esize) result;
integer prod;
for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row+dim+col, esize];
    for k = 0 to 3
      if ElemP[mask1, 4*row + k, esize DIV 4] == '1' &
         ElemP[mask2, 4*col + k, esize DIV 4] == '1' then
        prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
        if sub_op then prod = -prod;
        sum = sum + prod;
```
\[\text{Elem}[\text{result, row} \times \text{dim+col}, \text{esize}] = \text{sum};\]

\[\text{ZAtile}[\text{da, esize, dim} \times \text{dim} \times \text{esize}] = \text{result};\]

**Operational information**

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C9.2.33 UMOPS

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds SVLS×4 sub-matrix of unsigned 8-bit integer values, and the second source holds 4×SVLS sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds SVLD×4 sub-matrix of unsigned 16-bit integer values, and the second source holds 4×SVLD sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting SVLS×SVLS widened 32-bit integer or SVLD×SVLD widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a SVLS×4 sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a 4×SVLS sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a SVLD×4 sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a 4×SVLD sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

32-bit

(FEAT_SME)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 13 12 10 9</th>
<th>5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 0 0 1 1 0 1 0 1 0 1 0 0 1</td>
<td>Zm Pm Pn Zn 1 0 0 ZAda</td>
<td></td>
</tr>
<tr>
<td>u0 u1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

UMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

Decode for this encoding

if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
64-bit

(FEAT_SME_I16I64)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 13 12 10 9</th>
<th>5 4 3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zm</td>
<td>Pm</td>
<td>Pn</td>
</tr>
<tr>
<td>Zn</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ZAda</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Encoding

UMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

Decode for this encoding

if !HaveSMEI6I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;

Assembler symbols

<ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation for all encodings

CheckStreamingSVEAndZAEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 3
      if Elem[mask1, 4*row + k, esize DIV 4] == '1' &&
        Elem[mask2, 4*col + k, esize DIV 4] == '1' then
        prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
      if sub_op then prod = -prod;
      sum = sum + prod;
  

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C9.2.34 USMOPA

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned by signed integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds SVLS×4 sub-matrix of unsigned 8-bit integer values, and the second source holds 4×SVLS sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds SVLD×4 sub-matrix of unsigned 16-bit integer values, and the second source holds 4×SVLD sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting SVLS×SVLS widened 32-bit integer or SVLD×SVLD widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a SVLS×4 sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a 4×SVLS sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a SVLD×4 sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a 4×SVLD sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

32-bit

(FEAT_SME)

| 31 30 29 28 |27 26 25 24|23 22 21 20| 16|15 13 12|10 9 | 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

Zm Pm Pn Zn 0 0 0 ZAda

u0 u1

Encoding

USMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

Decode for this encoding

if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
64-bit

(Feat SME_I16I64)

\[
\begin{array}{cccccccccccc}
1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
\end{array}
\]

Encoding

USMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

Decode for this encoding

if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;

Assembler symbols

<ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation for all encodings

CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile(da, esize, dim*dim*esize);
bits(dim*dim*esize) result;
integer prod;
for row = 0 to dim-1
  for col = 0 to dim-1
    bits(esize) sum = Elem[operand3, row*dim+col, esize];
    for k = 0 to 3
      if Elem[mask1, 4*row + k, esize DIV 4] == '1' &&
        Elem[mask2, 4*col + k, esize DIV 4] == '1' then
        prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
      if sub_op then prod = -prod;
      sum = sum + prod;
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
C9.2.35 USMOPS

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned by signed integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds SVL₆×₄ sub-matrix of unsigned 8-bit integer values, and the second source holds 4×SVL₆ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds SVL₉×₄ sub-matrix of unsigned 16-bit integer values, and the second source holds 4×SVL₉ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting SVL₆×SVL₆ widened 32-bit integer or SVL₉×SVL₉ widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a SVL₆×₄ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a 4×SVL₆ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a SVL₉×₄ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a 4×SVL₉ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

32-bit

(FEAT_SME)

<table>
<thead>
<tr>
<th>[31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 13 12</th>
<th>10 9</th>
<th>5 4 3 2 1 0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 0 0</td>
<td>1 1 0 0</td>
<td>Zm</td>
<td>Pm</td>
<td>Pn</td>
<td>Zn</td>
<td>1 0 0</td>
<td>ZAda</td>
</tr>
<tr>
<td>u0</td>
<td>u1</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding**

USMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

**Decode for this encoding**

if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
### 64-bit

(C9.2 Alphabetical list of SME instructions)

#### Encoding

USMOPS `<ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H`

#### Decode for this encoding

```plaintext
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
```

#### Assembler symbols

- `<ZAda>`: For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
- `<Pn>`: Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- `<Pm>`: Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- `<Zn>`: Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>`: Is the name of the second source scalable vector register, encoded in the "Zm" field.

#### Operation for all encodings

```plaintext
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;  
integer prod;
for row = 0 to dim-1
for col = 0 to dim-1
  bits(esize) sum = Elem[operand3, row*dim+col, esize];
  for k = 0 to 3
    if Elem[mask1, 4*row + k, esize DIV 4] == '1' &
      Elem[mask2, 4*col + k, esize DIV 4] == '1' then
      prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
               Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
      if sub_op then prod = -prod;
      sum = sum + prod;
```

```
Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  — The values of the NZCV flags.
C9.2.36 ZERO

Zeroes all bytes within each of the up to eight listed 64-bit element tiles named ZA0.D to ZA7.D, leaving the other
64-bit element tiles unmodified.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will
not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

For programmer convenience an assembler must also accept the names of 32-bit, 16-bit, and 8-bit element tiles
which are converted into the corresponding set of 64-bit element tiles.

In accordance with the architecturally defined mapping between different element size tiles:

- Zeroing the 8-bit element tile name ZA0.B, or the entire array name ZA, is equivalent to zeroing all eight
  64-bit element tiles named ZA0.D to ZA7.D.
- Zeroing the 16-bit element tile name ZA0.H is equivalent to zeroing 64-bit element tiles named ZA0.D,
- Zeroing the 16-bit element tile name ZA1.H is equivalent to zeroing 64-bit element tiles named ZA1.D,
- Zeroing the 32-bit element tile name ZA0.S is equivalent to zeroing 64-bit element tiles named ZA0.D and
  ZA4.D.
- Zeroing the 32-bit element tile name ZA1.S is equivalent to zeroing 64-bit element tiles named ZA1.D and
  ZA5.D.
- Zeroing the 32-bit element tile name ZA2.S is equivalent to zeroing 64-bit element tiles named ZA2.D and
  ZA6.D.
- Zeroing the 32-bit element tile name ZA3.S is equivalent to zeroing 64-bit element tiles named ZA3.D and
  ZA7.D.

The preferred disassembly of this instruction uses the shortest list of tile names that represent the encoded
immediate mask.

For example:

- An immediate which encodes 64-bit element tiles ZA0.D, ZA1.D, ZA4.D, and ZA5.D is disassembled as
  \{ZA0.S, ZA1.S\}.
  \{ZA0.H\}.
- An all-ones immediate is disassembled as \{ZA\}.
- An all-zeros immediate is disassembled as an empty list \{\}.

SME

(FEAT_SME)

\[
| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 | 0 |
\[
| 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 |

\textit{Encoding}

ZERO \{<mask>\}
**Decode for this encoding**

if !HaveSME() then UNDEFINED;
bits(8) mask = imm8;
constant integer esize = 64;

**Assembler symbols**

<mask> Is a list of up to eight 64-bit element tile names separated by commas, encoded in the "imm8" field.

**Operation**

CheckSMEAndZAEenabled();
constant integer svl = SVL;
constant integer dim = svl DIV esize;
bits(dim*dim*esize) result = Zeros(dim*dim*esize);

if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);

for i = 0 to 7
    if mask<i> == '1' then
        ZAtile[i, esize, dim*dim*esize] = result;
Part D

The AArch64 System Level Architecture
This chapter describes the AArch64 system level programmers’ model. It contains the following sections:

- Exception levels on page D1-4632.
- Exceptions on page D1-4638.
- Registers for instruction processing and exception handling on page D1-4636.
- Process state, PSTATE on page D1-4670.
- Resets and power domains on page D1-4673.
- Mechanisms for entering a low-power state on page D1-4676.
- Self-hosted debug on page D1-4681.
- Event monitors on page D1-4683.
- Interprocessing on page D1-4684.
D1.1 Exception levels

The architecture defines four Exception levels: EL0, EL1, EL2, and EL3.

EL3 is the highest Exception level and EL0 the lowest. Therefore, EL3 is higher than EL2, EL2 is higher than EL1, and EL1 is higher than EL0.

Unprivileged execution is any execution that occurs at EL0.

An implementation includes EL0 and EL1.

It is IMPLEMENTATION DEFINED whether EL2 is implemented, or not.

If implemented, EL2 provides support for the virtualization of EL0 and EL1.

It is IMPLEMENTATION DEFINED whether EL3 is implemented, or not.

A PE is not required to implement a contiguous set of Exception levels.

The current Exception level changes only when any of the following occur:
- Taking an exception.
- Returning from an exception.
- Processor reset.
- Exiting from Debug state.
- If in Debug state, executing a DCPS\(^x\) instruction.
- If in Debug state, executing a DRPS instruction.

The target Exception level is the Exception level to which an exception is taken.

Each exception type has a target Exception level that is either:
- Implicit in the type of the exception.
- Defined by configuration bits in the System registers.

An exception cannot be taken to EL0.

An exception cannot cause entry to a lower Exception level.

D1.1.1 Execution state

The architecture defines two Execution states. The AArch32 Execution state uses 32-bit processing. The AArch64 Execution state uses 64-bit processing.

The AArch32 Execution state is compatible with Armv7-A operation.

An Exception level is described as:
- Using AArch64 when execution in that Exception level is in the AArch64 Execution state.
- Using AArch32 when execution in that Exception level is in the AArch32 Execution state.

The PE can only change Execution state at reset, or when changing Exception level.

If an Exception level is using AArch32, then all lower Exception levels are using AArch32.

If an Exception level is using AArch64, then all higher Exception levels are using AArch64.

Interaction between the AArch64 and AArch32 Execution states is called interprocessing.

In this chapter, the described behavior assumes the highest implemented Exception level is using AArch64.

D1.1.2 Security state

The architecture defines two Security states, called Secure state and Non-secure state.
The architecture defines two physical address spaces, Secure and Non-secure.

If in Non-secure state, the PE can only access the Non-secure physical address space.

If in Secure state, the PE can access both the Secure physical address space and the Non-secure physical address space.

EL3 is in Secure state.

The Security state of EL2, EL1, and EL0 is determined by the Effective value of SCR_EL3.NS. If the Effective value of SCR_EL3.NS is 0, the PE is in Secure state. If the Effective value of SCR_EL3.NS is 1, the PE is in Non-secure state.

EL2 can only be in Secure state if all the following are true:
- FEAT_SEL2 is implemented.
- The Effective value of SCR_EL3.EEL2 is 0b1.
- The Effective value of SCR_EL3.NS is 0b0.

Otherwise, EL2 is in Non-secure state.

The state of SCR_EL3.NS can only change in EL3.

**D1.1.3 Effect of not implementing an Exception level**

An exception cannot be taken to an unimplemented Exception level.

Any configurable instruction control that defines an exception to an unimplemented Exception level does not cause that exception to the unimplemented Exception level. The Effective value of the configurable exception control is the value that does not cause an exception to that Exception level.

System calls made to an unimplemented Exception level from a lower Exception level are UNDEFINED.

The translation regime associated with an unimplemented Exception level is not implemented.

Any exception return that targets an unimplemented Exception level is an illegal return.

Every accessible register associated with an unimplemented Exception level is RES0 unless the register is associated with the Exception level only to provide the ability to transfer execution to a lower Exception level.

**Behavior when EL3 is not implemented**

If EL3 is not implemented, SCR_EL3.NS has a fixed Effective value that is IMPLEMENTATION DEFINED.

An implementation can provide a configuration input that determines the Effective value of SCR_EL3.NS at reset.

If EL3 is not implemented, and the Effective value of SCR_EL3.NS is 0b0, all of the following are true:
- The Effective value of MDCR_EL3.EPMAD is 0b1.
- The Effective value of MDCR_EL3.EDAD is 0b1.
- The Effective value of MDCR_EL3.SPME is 0b1.
- The Effective value of MDCR_EL3.NSPB is 0b01.
- The Effective value of MDCR_EL3.SPD32 is 0b11.
- The Effective value of MDCR_EL3.STE is 0b1.

If EL3 is not implemented, the Effective value of MDCR_EL3.STE is the inverse of the Effective value of SCR_EL3.NS.

If all of the following are true, the Effective value of SCR_EL3.EEL2 is 0b1:
- EL3 is not implemented.
- EL2 is implemented.
- The Effective value of SCR_EL3.NS is 0b0.
Behavior when EL2 is not implemented

If EL2 is not implemented, all of the following are true:
- Virtual interrupts are disabled.
- The Effective value of CNTHCTL_EL2[1:0] is 0b11.
- The Effective value of HCR_EL2.E2H is 0b0.
- The Effective value of HCR_EL2.TGE is 0b0.
- The Effective value of MDCR_EL2.HPMN is the value of PMCR_EL0.N

Behavior when EL2 is not implemented and EL3 is implemented

If EL2 is not implemented and EL3 is implemented, all of the following are true:
- Except for any of the following registers, every accessible register associated with EL2 is RES0:
  - If EL1 can use AArch32, then the following registers are not RES0:
    - DACR32_EL2.
    - DBGVCR32_EL2.
    - FPEXC32_EL2.
    - IFSR32_EL2.
  - For VPIDR_EL2:
    - Reads return the value of MIDR_EL1.
    - Writes to VPIDR_EL2 are ignored.
  - For VMPIDR_EL2:
    - Reads return the value of MPIDR_EL1.
    - Writes to VMPIDR_EL2 are ignored.
- The Effective value of HCR_EL2.RW is the value of SCR_EL3.RW.
- SCR_EL3.HCE is RES0.
- The following Address translation and TLB invalidation instructions are UNDEFINED:
  - AT S1E2R.
  - AT S1E2W.
  - TLBI VAE2.
  - TLBI VALE2.
  - TLBI VAE2IS.
  - TLBI VALE2IS.
  - TLBI VALE2OS.
  - TLBI VAE2OS.
  - TLBI RVAE2.
  - TLBI RVALE2.
  - TLBI RVAE2IS.
  - TLBI RVALE2IS.
  - TLBI RVAE2OS.
  - TLBI RVALE2OS.
Typical Exception level usage model

The architecture does not specify what software uses which Exception level. Such choices are outside the scope of the architecture. However, the following is a common usage model for the Exception levels:

<table>
<thead>
<tr>
<th>Exception level</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL0</td>
<td>Applications.</td>
</tr>
<tr>
<td>EL1</td>
<td>OS kernel and associated functions that are typically described as privileged.</td>
</tr>
<tr>
<td>EL2</td>
<td>Hypervisor.</td>
</tr>
<tr>
<td>EL3</td>
<td>Secure monitor.</td>
</tr>
</tbody>
</table>
D1.2 Registers for instruction processing and exception handling

In the Arm architecture, registers are split into two main categories:

- Registers that provide system control or status reporting.
- Registers that are used in instruction processing, for example to accumulate a result, and in handling exceptions.

For more information, see Chapter D17 AArch64 System Register Descriptions.

D1.2.1 The general-purpose registers

The general-purpose register bank comprises the 31 general-purpose registers, R0-R30.

The general-purpose register bank is associated with instructions in the base instruction set.

The general-purpose registers can be accessed as either of the following:

- 31 64-bit registers, X0-X30.
- 31 32-bit registers, W0-W30.

When a Warm reset is asserted, the general-purpose registers are reset to an architecturally UNKNOWN value.

For more information, see Register size on page C6-1166 and Registers in AArch64 state on page B1-137.

D1.2.2 The stack pointer registers

If in AArch64 state, each implemented Exception level has a dedicated stack pointer register. The stack pointer registers are:

- SP_EL0.
- SP_EL1.
- If EL2 is implemented, SP_EL2.
- If EL3 is implemented, SP_EL3.

When a Warm reset is asserted, the stack pointers are reset to an architecturally UNKNOWN value.

For more information, see Special-purpose registers on page C5-671 and SP alignment checking on page D1-4668.

Stack pointer register selection

The PE uses the following stack pointers:

- If executing at EL0, then the PE uses the EL0 stack pointer, SP_EL0.
- If executing at EL1, EL2, or EL3, then the PE uses the stack pointer determined by the PSTATE.SP bit:
  - If PSTATE.SP is 0, then the PE uses the EL0 stack pointer, SP_EL0.
  - If PSTATE.SP is 1, then the PE uses the stack pointer for the current Exception level.

When an exception is taken, the stack pointer for the target Exception level is selected.

The selected stack pointer can be indicated by a suffix to the Exception level:

- The t suffix, referring to thread, indicates use of the SP_EL0 stack pointer.
- The h suffix, referring to host, indicates use of the SP_ELx stack pointer.
The following are the AArch64 stack pointer options:

<table>
<thead>
<tr>
<th>Exception level (EL)</th>
<th>Stack pointer (SP) options</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL0</td>
<td>SP_EL0t</td>
</tr>
<tr>
<td>EL1</td>
<td>SP_EL1t, SP_EL1h</td>
</tr>
<tr>
<td>EL2</td>
<td>SP_EL2t, SP_EL2h</td>
</tr>
<tr>
<td>EL3</td>
<td>SP_EL3t, SP_EL3h</td>
</tr>
</tbody>
</table>

**D1.2.3 The SIMD and floating-point registers**

- The SIMD and floating-point register bank comprises the 32 quadword (128-bit) SIMD and floating-point registers, V0-V31.
- The SIMD and floating-point register bank is used for floating-point, vector, and other SIMD-related scalar operations.
- The SIMD and floating-point registers can be accessed as any of the following:
  - 32 quadword (128-bit) registers, Q0-Q31.
  - 32 doubleword (64-bit) registers, D0-D31.
  - 32 word (32-bit) registers, S0-S31.
  - 32 halfword (16-bit) registers, H0-H31.
  - 32 byte (8-bit) registers, B0-B31.
  - 128-bit vectors of elements.
  - 64-bit vectors of elements.
- When a Warm reset is asserted, the SIMD and floating-point registers are reset to an architecturally UNKNOWN value.

For more information, see *Registers in AArch64 state* on page B1-137.

**D1.2.4 Saved Program Status Registers**

- In AArch64 state, each implemented Exception level to which an exception can be taken has a Saved Program Status Register (SPSR). The following are the SPSRs:
  - For exceptions taken to EL1 using AArch64, SPSR_EL1.
  - For exceptions taken to EL2 using AArch64, SPSR_EL2.
  - For exceptions taken to EL3 using AArch64, SPSR_EL3.
- When a Warm reset is asserted, the SPSRs are reset to an architecturally UNKNOWN value.

For more information, see *Process state, PSTATE* on page D1-4670.

**D1.2.5 Exception Link Registers**

- Exception Link Registers hold preferred exception return addresses.
- In AArch64 state, each implemented Exception level to which an exception can be taken has an Exception Link Register (ELR). The following are the ELRs:
  - For exceptions taken to EL1, ELR_EL1.
  - For exceptions taken to EL2, ELR_EL2.
  - For exceptions taken to EL3, ELR_EL3.
- When a Warm reset is asserted, the ELRs are reset to an architecturally UNKNOWN value.

For more information, see *Preferred exception return address* on page D1-4640.
D1.3 Exceptions

D1.3.1 Exception entry terminology

*Taken, taken from, and taken to*

When the PE responds to an exception, an exception is *taken*.

The PE state immediately before taking the exception is the state the exception is *taken from*.

The PE state immediately after taking the exception is the state the exception is *taken to*.

**Exception producing instructions**

Instructions that intentionally cause an exception to be taken are called exception producing instructions.

The following are exception producing instructions:

- HVC.
- SMC.
- SVC.

Exception producing instructions produce a precise exception at the point in the instruction stream immediately after the exception producing instruction.

**Synchronous and asynchronous exceptions**

If all of the following apply, an exception is *synchronous*:

- The exception is generated as a result of direct execution or attempted execution of an instruction.
- The preferred exception return address has an architecturally defined relationship with the instruction that caused the exception.
- The exception is precise.

An exception is *asynchronous* if it is not synchronous.

Asynchronous exceptions taken to AArch64 state are also known as interrupts.

**Definition of a precise exception and imprecise exception**

An exception is *precise* if on taking the exception, the PE state and the memory system state is consistent with the PE having executed all of the instructions up to but not including the point in the instruction stream where the exception was taken from, and none afterwards. However, for an instruction executing immediately at the point in the instruction stream that the exception was taken from, the definition of precise also permits any of the following:

- For synchronous Data Abort and Watchpoint exceptions that are taken to AArch64 state generated by an instruction that performs more than one single-copy atomic memory access, the values in registers or memory affected by the instructions can be UNKNOWN, if all of the following apply:
  - The accesses affecting those registers or memory locations do not, themselves, generate exceptions or debug events.
  - The registers are not involved in the calculation of the memory address used by the instruction.
- For synchronous Data Abort and Watchpoint exceptions that are generated from load or store instructions executed in AArch64 state, all the following can occur:
  - If the instruction was a load to either the base address register or the offset register, that register is restored to the original value, and any other destination registers become UNKNOWN.
  - If the instruction was a load that does not load the base address register or the offset register, then the destination registers become UNKNOWN.
For implementations that include synchronous exception generation for floating-point exceptions, when a floating-point exception is taken, it is permitted that the cumulative floating-point exception bits are not restored.

For a synchronous exception that is taken from AArch64 state during an instruction that performs more than one single-copy atomic memory access, the values in registers or memory affected by the instructions can be UNKNOWN, if all of the following apply:
- The accesses affecting those registers or memory locations do not, themselves, generate exceptions or debug events.
- The registers are not involved in the calculation of the memory address used by the instruction.

For a synchronous exception or synchronous debug state entry, generated by a Memory Copy and Memory Set instruction, and for an asynchronous exception or asynchronous debug state entry part way through the execution of a Memory Copy and Memory Set instruction:
- The values of PSTATE.\{N,Z,C,V\} are UNKNOWN on exception entry. The values held in
  SPSR_ELx.\{N,Z,C,V\} on exception entry, and DSISR_EL0.\{N,Z,C,V\} on debug state entry, are the initial values for the instruction.
- All memory locations written by the pseudocode at the point at which the exception is taken are updated.
- If the memory locations that are not written by the pseudocode at the point at which the exception is taken are within the set of locations to be read by the instruction if it is restarted with the X[n], X[d], and X[s] register values presented as part of taking the exception, then those memory locations are not updated.
- For CPYF*T* and SET{G}*T* instructions, no memory locations that are not written by the pseudocode at the point at which the exception is taken are updated.
- For Memory Copy and Memory Set instructions other than CPYF*T* and SET{G}*T*, any memory location that does not itself generate a synchronous exception on a write becomes UNKNOWN if all of the following apply:
  - The memory location is within the set of locations to be written by the instruction if it is restarted with the X[n], X[d], and X[s] register values presented as part of taking the exception.
  - The memory location is not within the set of locations to be read by the instruction on that restart.
- Only for a synchronous exception or synchronous debug state entry, generated by a Memory Copy and Memory Set instruction:
  - On taking the exception, the X[n], X[d], and, as appropriate, X[s] register values hold a self-consistent set of values that correspond to the first element that was not copied or set, such that return to the instruction enables resumption of the memory copy or memory set. That first element does not need to be the element with a watchpoint, but can be an earlier element, including the first element.
  - If a memory location that generates a synchronous exception on a write is within the set of locations to be written by the instruction if it is restarted with the X[n], X[d], and X[s] register values presented as part of taking the exception, and if the memory location is not within the set of locations to be read by the instruction on that restart, then the memory location is unchanged.
  - For a synchronous exception, ELR_ELx points to the instruction that generated the exception.
  - For a synchronous debug state entry, DLR_EL0 points to the instruction that generated the debug state entry.
- Only for an asynchronous exception or asynchronous debug state entry part way through the execution of a Memory Copy and Memory Set instruction:
  - On taking the exception, the values of the registers that are updated by the instruction are presented as a self-consistent set of values that correspond to the first element that was not copied or set.

R\textsubscript{ZNCXP} An exception is imprecise if it is not a precise exception.
Except for SError interrupts, all exceptions taken to AArch64 state are precise. For each occurrence of an SError interrupt, whether the interrupt is precise or imprecise is IMPLEMENTATION DEFINED.

An asynchronous Data Abort exception generated by a load that causes an SError exception to be taken at some point later in the instruction stream than the load that generated the exception, is usually imprecise. The SError exception is usually imprecise because the data returned from the load is UNKNOWN, and so can have corrupted the state that is presented at the time that the exception is taken.

**Preferred exception return address**

The preferred exception return address is an address that software might return to after handling an exception in order to resume execution.

The preferred exception return address is determined by the type of the exception.

For an exception taken to an Exception level, ELx, using AArch64, the Exception Link Register for that Exception level, ELR_ELx, is set to the preferred exception return address.

For asynchronous exceptions, the preferred exception return address is the address of the instruction following the instruction boundary at which the interrupt occurs.

For synchronous exceptions other than exception producing instructions, the preferred exception return address is the address of the instruction that generates the exception.

For an exception producing instruction that is executed, the preferred exception return address is the address of the instruction that follows the exception producing instruction.

For an exception producing instruction that is trapped, disabled, or is UNDEFINED because the Exception level has insufficient privilege to execute the instruction, the preferred exception return address is the address of the exception producing instruction.

When an exception is taken from an Exception level using AArch32 to an Exception level, ELx, using AArch64, ELR_ELx[63:32] are 0.

**Exception vectors**

When an exception is taken to an Exception level that is using AArch64, execution starts at the exception vector.

The memory at the exception vector for an exception is expected to contain the handler code that corresponds to that exception category.

The vector base address for an Exception level, ELx, is defined by the Vector Base Address Register, VBAR_ELx.

Each exception category has an exception vector at a fixed offset from the vector base address.

Exceptions taken to AArch64 are categorized for the purpose of assigning exception vectors based on the following information:

- Whether the exception is one of the following:
  - Synchronous exception.
  - SError.
  - IRQ.
  - FIQ.
- Information about the Exception level that the exception came from.
- Information about the SP in use.
- The state of the register file.
The AArch64 System Level Programmers’ Model

D1.3 Exceptions

RKYXCL The following table describes the offsets that are added to the vector base address to describe the exception vector:

<table>
<thead>
<tr>
<th>Exception taken from</th>
<th>Offset for exception type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Synchronous (including synchronous External aborts if SCR_EL3.EASE is 0)</td>
</tr>
<tr>
<td>Current Exception level with SP_EL0.</td>
<td>0x000</td>
</tr>
<tr>
<td>Current Exception level with SP_ELx, x&gt;0.</td>
<td>0x200</td>
</tr>
<tr>
<td>Lower Exception level, where the implemented level immediately lower than the target level is using AArch64.</td>
<td>0x400</td>
</tr>
<tr>
<td>Lower Exception level, where the implemented level immediately lower than the target level is using AArch32.</td>
<td>0x600</td>
</tr>
</tbody>
</table>

For the preceding table, the implemented Exception level immediately lower than the target Exception level is defined by RFSZTB.

If EL3 is implemented, the Exception level immediately lower than the target Exception level is determined by whether the exception was taken from an Exception level in Secure or Non-secure state.

For an exception taken to EL3 from a lower Exception level, if EL2 is implemented and enabled for the security state of Exception level the exception was taken from, the Exception level immediately lower than EL3 is EL2.

For an exception taken to EL3 from a lower Exception level, if EL2 is not implemented and enabled for the security state of Exception level the exception was taken from, the Exception level immediately lower than EL3 is EL1.

D1.3.2 Exception entry

RDQXFW When an exception is taken to an Exception level, ELx, that is using AArch64 state, all of the following occur:

- The contents of PSTATE immediately before the exception was taken is written to SPSR_ELx.
- The preferred exception return address is written to ELR_ELx.
- The contents of PSTATE immediately after the exception is taken is as described in rule RWTXBY.
- For synchronous exceptions and SError interrupts, exception syndrome information is written to ESR_ELx.
- Execution starts from the exception vector at the target Exception level.

RWTXBY When an exception is taken to an Exception level, ELx, that is using AArch64 state, after recording the previous values in SPSR_ELx, the following PSTATE bits are set:

- PSTATE.EL is set to the target Exception level.
- All of PSTATE. {D, A, I, F, SP, TCO} are set to 1.
- PSTATE.SSBS is set to the value of SCTLR_ELx.DSSBS.
- PSTATE. {IL, nRw, UAO} are set to 0.
- PSTATE.BTYPE is set to 0b0.
- PSTATE.SS is set according to the rules in Chapter D2 AArch64 Self-hosted Debug.
- For any of the following situations, PSTATE.PAN is set to 1:
  - The target Exception level is EL1 and SCTLR_EL1.SPN is 0.
The exception is taken from EL0, the target Exception level is EL2 using AArch64, Secure EL2 is enabled, HCR_EL2.{TGE, E2H} is {1, 1}, and SCTLR_EL2.SPAN is 0.

- PSTATE.ALLINT is set to the inverse value of SCTLR_ELx.SPINTMASK.

If the Effective value of SCTLR_ELx.IESB is 1 at the target Exception level, an exception taken to AArch64 state is an error synchronization event.

If SCTLR_ELx.EIS is 1, exception entry is a context synchronization event. If SCTLR_ELx.EIS is 0, exception entry is not a context synchronization event, but the indirect writes to ESR_ELx, FAR_ELx, SPSR_ELx, ELR_ELx, and HPFAR_EL2 due to exception entry are synchronized so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.

The ordering of error synchronization and context synchronization on exception entry is described in the pseudocode.

Memory copy and memory set operations are guaranteed to function correctly only if the prologue, main, and epilogue Memory Copy and Memory Set instructions are executed in succession and are placed consecutively in memory. Failure to execute the three instructions in succession can result in the instructions causing exceptions. Software exception handlers are expected to require that the three instructions are placed consecutively in memory. If the three instructions are not placed consecutively in memory, exception returns to the wrong instructions might occur.

### Synchronous exception entry

For any of the following synchronous exceptions taken from an Exception level using AArch64, PSTATE.BTYPE is copied to SPSR_ELx.BTYPE and then PSTATE.BTYPE is set to 0:

- Software Step exception.
- PC Alignment Fault exception.
- Instruction Abort exception.
- Breakpoint exception.
- Address Matching Vector Catch exception.
- Illegal Execution state exception.
- Software Breakpoint exception.
- Branch Target exception.

When taking any other synchronous exception from an Exception level using AArch64, it is CONSTRAINED UNPREDICTABLE whether:

- SPSR_ELx.BTYPE is set to the value of PSTATE.BTYPE.
- SPSR_ELx.BTYPE is set to 0.

PSTATE.BTYPE is then set to 0.

For any of the following synchronous exception types, when an exception is taken to an Exception level, ELx, a virtual address that characterizes the exception is captured in FAR_ELx:

- An Instruction Abort exception.
- A Data Abort exception.
- A PC alignment fault exception.
- A Watchpoint exception.

For synchronous External aborts that are not caused by translation table walks, it is CONSTRAINED UNPREDICTABLE whether the FAR_ELx contains a virtual address that characterizes the exception. The ESR_ELx.FnV bit in the ISS encoding for the External abort indicates the validity of the virtual address in FAR_ELx:

- If ESR_ELx.FnV in the ISS encoding for the External abort is 1, the Faulting Virtual Address in FAR_ELx is UNKNOWN.
- If ESR_ELx.FnV in the ISS encoding for the External abort is 0, the Faulting Virtual Address in FAR_ELx is valid.

For all other exceptions taken to ELx, FAR_ELx is UNKNOWN.
For Instruction Abort or Data Abort exceptions caused by any of the following faults, when an exception is taken to EL2, an intermediate physical address that characterizes the exception is captured in HPFAR_EL2:

- A Translation fault on a stage 2 translation.
- An Access Flag fault on a stage 2 translation.
- A stage 2 Address Size fault.
- A fault on stage 2 translation of an address accessed in a stage 1 translation table walk.

For all other exceptions taken to EL2 using AArch64, HPFAR_EL2 is UNKNOWN.

If an Instruction Abort or Breakpoint exception is generated part way through execution of a Memory Copy and Memory Set instruction, presentation of the exception state is as described in R_TNVSL for a synchronous exception or synchronous debug state entry from a watchpoint, generated by a Memory Copy and Memory Set instruction.

For a CPy Memory Copy instruction, if both the read and write generate the Data Abort or Watchpoint exception, it is CONSTRAINED UNPREDICTABLE whether the exception on the read or the exception on the write is presented. ESR_ELX.ISS.WnR is set to be consistent with whether the exception was on the read or the write.

**SVE MOVPRFX exception entry behavior**

When a MOVPRFX instruction pairs legally with another instruction and the execution of the pair generates a synchronous exception, the return address that is stored in ELR_ELx is one of the following:

- When the MOVPRFX instruction did not cause a change to the architectural state, the address of the MOVPRFX instruction is stored.
- When the MOVPRFX instruction caused a change to the architectural state, the address of the prefixed instruction is stored.

When a MOVPRFX instruction pairs illegally with another instruction and execution of the pair causes entry to Debug state, the return address that is stored in DLR_EL0 is one of the following:

- When the MOVPRFX instruction did not cause a change to the architectural state, the address of the MOVPRFX instruction is stored.
- When the MOVPRFX instruction caused a change to the architectural state, the address of the prefixed instruction is stored.

When a prefixed instruction generates an Instruction Abort due to an MMU fault or synchronous External abort and the MOVPRFX does not generate an Instruction Abort, then the address of the prefixed instruction is recorded in the appropriate FAR_ELx or HPFAR_EL2 register and the address of the MOVPRFX instruction is recorded in the appropriate ELR_ELx register.

When a prefixed instruction generates an Instruction Abort due to an MMU fault or synchronous External abort and the MOVPRFX also generates an Instruction Abort, then the address of the MOVPRFX instruction is recorded in the appropriate FAR_ELx or HPFAR_EL2 register and the appropriate ELR_ELx register.
Asynchronous exception entry

When an asynchronous exception is taken from an Exception level using AArch64, PSTATE.BTYPE is copied to SPSR_ELx.BTYPE and then PSTATE.BTYPE is set to 0.

If any of the following apply, when a physical SError interrupt exception is taken to AArch64 state, the pending state of the physical SError is cleared:

- FEAT_DoubleFault is implemented.
- If the Reliability, Availability, and Serviceability Extension (RAS Extension) is implemented, and on taking the SError interrupt, the syndrome recorded in ESR_ELx indicates an SError other than IMPLEMENTATION DEFINED or uncategorized SError interrupt syndrome.

Otherwise, it is IMPLEMENTATION DEFINED whether the pending state of the physical SError is cleared. This IMPLEMENTATION DEFINED behavior might vary according to the type of the SError interrupt.

When a virtual SError interrupt exception is taken to AArch64 state, HCR_EL2.VSE is set to 0.

An asynchronous exception or asynchronous debug state entry part way through the execution of a Memory Copy and Memory Set instruction is permissible. For an asynchronous exception, ELR_ELx points to the instruction that the exception was taken on. For an asynchronous debug state entry, DLR_EL0 points to the instruction that the debug state entry was taken on.

For an imprecise asynchronous exception part way through the execution of a Memory Copy and Memory Set instruction, the following are UNKNOWN:

- The state of the X[n], X[d], and X[s] registers.
- The state of memory that was being written to by the instruction.

D1.3.3 Exception return terminology

Return, return from, return to

An exception return is caused by the execution of an exception return instruction.

The PE state immediately before an exception return instruction is executed is the state the exception returns from.

The PE state immediately after the execution of an exception return instruction is the state the exception returns to.

D1.3.4 Exception return

In AArch64 state, the Exception return instructions are ERET, ERETAA, and ERETAB.

The Exception return instructions are UNDEFINED in EL0.

An exception return is either legal or illegal.

Legal exception returns from AArch64 state

On a legal exception return from an Exception level, ELx, all of the following occur:

- If SCTLR_ELx.IESB is 1, and the Exception return instruction does not generate an exception, the PE inserts an error synchronization event before the Exception return instruction.
- The PC is set to the value in ELR_ELx.
- If returning to an Exception level using AArch32 state, all of the following apply:
  - If SPSR_ELx.T is 0, ELR_ELx[1:0] are treated as being 0 for setting the PC.
  - If SPSR_ELx.T is 1, ELR_ELx[0] is treated as being 0 for setting the PC.
- The contents of PSTATE are set to the values held in SPSR_ELx.
• If the PSTATE.IL bit copied from SPSR_ELx is 1 and the target Exception level for the return is using AArch32 state, the PSTATE.\{IT,T\} bits are determined by an IMPLEMENTATION DEFINED choice of one of the following:
  — Set to 0.
  — For returns initiated by an Exception return instruction or DRPS instruction, copied from SPSR_ELx.
  — For debug exits, copied from DSPSR_EL0.

The IMPLEMENTATION DEFINED choice might vary dynamically within the implementation. Software must regard the value as being an UNKNOWN choice between the two values.

• The Event Register for the PE executing the Exception return instruction is set.

• The local Exclusives monitor for the PE executing the Exception return instruction is cleared. It is IMPLEMENTATION DEFINED whether clearing the local Exclusives monitor also clears the global Exclusives monitor.

• After the PC is set to the value held in ELR_ELx and the contents of PSTATE are set to the values held in SPSR_ELx, ELR_ELx and SPSR_ELx become UNKNOWN.

• If the Effective value of SCTLR_ELx.EOS is 1, the exception return is a context synchronization event.

• If the Effective value of SCTLR_ELx.EOS is 0, the exception return is not a context synchronization event.

**Illegal exception returns from AArch64 state**

If in AArch64 state, any of the following situations can cause an illegal exception return:

• A return is made to an Exception level higher than the current Exception level.

• A return is made to an Exception level that is not implemented.

• If all of the following are true, and a return is made to EL1:
  — EL2 is implemented and enabled in the current Security state
  — HCR_EL2.TGE is 1.

• If all of the following are true, and a return is made to EL2:
  — EL3 is implemented.
  — SCR_EL3.NS is 0.
  — FEAT_SEL2 is not implemented.

• If all of the following are true, and a return is made to EL2:
  — EL3 is implemented.
  — SCR_EL3.NS is 0.
  — FEAT_SEL2 is implemented.
  — SCR_EL3.EEL2 is 0.

• A return where the saved PSTATE.M[4] is 0 and at least one of the following is true:
  — M[1] is 1.
  — M[3:0] are 0b0001.
  — The Exception level being returned to is using AArch32 state, as determined by SCR_EL3.RW, HCR_EL2.RW, or as configured from reset.

• A return where the saved PSTATE.M[4] bit is 1, indicating a return to AArch32 state, and at least one of the following are true:
  — AArch32 state is not supported at any Exception level.
  — The M field value is not a valid AArch32 state PE mode.
  — The Exception level being returned to is using AArch64 state, as determined by SCR_EL3.RW, HCR_EL2.RW, or as configured from reset.
On an illegal exception return from an Exception level, ELx, all of the following occur:

- If SCTLR_ELx.IESB is 1, the PE inserts an error synchronization event before the Exception return instruction.
- The PC is set to the value in ELR_ELx. If the saved PSTATE.M[4] bit is 1, for illegal exception returns to AArch32 state, all of the following are true:
  - The PC bits[31:2] are set to the value in ELR_ELx.
  - The PC bits[63:32, 1:0] are UNKNOWN.
- PSTATE.IL is set to 1.
- PSTATE.{EL, nRW, SP} are unchanged.
- PSTATE.{N, Z, C, V, D, A, I, F, PAN} are set to the associated values in SPSR_ELx.
- PSTATE.SS is handled as if the return is a legal exception return.
- PSTATE.{TCO, DIT, UAO, SSBS, BTYPE, ALLINT} are UNKNOWN.
- If PSTATE.nRW is 1, indicating a return to AArch32 state, then the following PSTATE bits are also set:
  - PSTATE.{Q, IT, GE, E} are set to the associated values in SPSR_ELx.
  - It is CONSTRAINED UNPREDICTABLE whether PSTATE.T is 0 or set to the contents of SPSR_ELx.
- The Event Register for the PE executing the Exception return instruction is set.
- The local Exclusives monitor for the PE executing the Exception return instruction is cleared. It is IMPLEMENTATION DEFINED whether clearing the local Exclusives monitor also clears the global Exclusives monitor.
- After the PC has been set to the value held in ELR_ELx and the contents of PSTATE have been set to the value held in SPSR_ELx, the ELR_ELx and SPSR_ELx become UNKNOWN.
- If the Effective value of SCTLR_ELx.EOS is 1, the exception return is a context synchronization event.
- If the Effective value of SCTLR_ELx.EOS is 0, the exception return is not a context synchronization event.

**D1.3.5 Synchronous exception types**

All of the following are Synchronous exceptions:

- Any exception generated by attempting to execute an instruction that is UNDEFINED, including:
  - Attempts to execute instructions at an inappropriate Exception level.
  - Attempts to execute instructions when they are disabled.
  - Attempts to execute instruction bit patterns that are not allocated.
- Any exception caused by attempts to execute an instruction when the value of PSTATE.IL is 1. These are called Illegal Execution State exceptions.
- Any exception caused by the use of a misaligned SP. These are called SP Alignment Fault exceptions.
- Any exception caused by attempting to execute an instruction with a misaligned PC. These are called PC Alignment Fault exceptions.
- If executing inside a guarded memory region and PSTATE.BTYPE does not equal 0, any exception caused by executing an instruction that is not compatible with the current value of PSTATE.BTYPE. These are called Branch Target exceptions.
- Any exception caused by a pointer authentication instruction authentication failure. These are called PAC Fail exceptions.
- Any exception caused by the exception producing instructions SVC, HVC, or SMC. These are respectively called Supervisor Call (SVC) exceptions, Hypervisor Call (HVC) exceptions, or Secure Monitor Call (SMC) exceptions.
- Traps on attempts to execute instructions that the System registers define as instructions that are trapped to a higher Exception level. These are called Trap exceptions.
- Any exception caused by Instruction Aborts that were generated by the memory address translation system that was associated with attempts to execute instructions from areas of memory that generate faults. These are called Instruction Abort exceptions.
• Any exception caused by Data Aborts that were generated by the memory address translation system that are associated with attempts to read or write memory that generate faults. These are called Data Abort exceptions.

• Any exception caused by Data Aborts because of a misaligned address. These are called Data Abort exceptions.

• If FEAT_MTE2 is implemented, any exception caused by a Data Abort as a result of a Tag Check Fault. These are called Data Abort exceptions.

• Memory Copy exceptions and Memory Set exceptions. For more information, see Memory Copy and Memory Set exceptions on page D1-4654.

• All of the debug exceptions:
  — Breakpoint Instruction exceptions.
  — Breakpoint exceptions.
  — Watchpoint exceptions.
  — Vector Catch exceptions.
  — Software Step exceptions.

• In an implementation that supports the trapping of floating-point exceptions, any exception caused by trapped floating-point exceptions. These are called Trapped Floating-point exceptions.

• An IMPLEMENTATION DEFINED set of exceptions caused by External aborts.

The architecture permits, but does not require, synchronous or asynchronous handling of External aborts.

The RAS architecture specifies some situations that must be handled synchronously.

Taking synchronous exceptions from EL0

If EL2 is using AArch64 and the Effective value of HCR_EL2.TGE is 1, when any of the following exceptions are taken from EL0, they are taken to EL2. If EL2 is using AArch64 and the Effective value of HCR_EL2.TGE is 0, when any of the following exceptions are taken from EL0, they are taken to EL1:

• Stage 1 Data Abort.
• Stage 1 Instruction Abort.
• PC Alignment Fault.
• SP Alignment Fault.
• Tag Check Fault.
• Branch Target Exception.
• Illegal Execution State Exception.
• Trapped Floating-point Exception.
• Supervisor Call.
• Undefined Instruction Exception.
• SVE Access Trap.
• PAC Fail Exception.
• WFE Trap.
• WFI Trap.
• Advanced SIMD and Floating-point access Trap.
• Synchronous External Aborts.
• Memory Copy and Memory Set Exceptions.

If EL2 is using AArch64 and either HCR_EL2.TGE is 1 or MDCR_EL2.TDE is 1, when any of the following debug exceptions are taken from EL0, they are taken to EL2. If EL2 is using AArch64 and both HCR_EL2.TGE is 0 and MDCR_EL2.TDE is 0, when any of the following debug exceptions are taken from EL0, they are taken to EL1:

• Breakpoint Exception.
• Software Breakpoint Exception.
• Software Step Exception.
• Watchpoint Exception.
If EL2 is using AArch64 and either HCR_EL2.TGE is 1 or MDCR_EL2.TDE is 1, when a Vector Catch exception is taken from EL0, it is taken to EL2. If EL2 is using AArch64 and both HCR_EL2.TGE is 0 and MDCR_EL2.TDE is 0, a Vector Catch exception cannot be taken.

For an exception that is taken from EL0 to EL2 because the value of HCR_EL2.TGE is 1:

- If the exception would have been reported in ESR_EL1 using any EC value other than 0x07, then the EC value and corresponding ISS encoding that would have been used to report the exception in ESR_EL1 are used to report the exception in ESR_EL2.
- If the exception would have been reported in ESR_EL1 using the EC value 0x07, then the EC value 0x00 and ISS encoding value 0x00 are reported in ESR_EL2.

Exception levels for taking a synchronous External abort

A synchronous external abort is taken to EL1 if executing at EL0 or EL1 and none of the following scenarios cause the synchronous external abort to be taken to EL2 or EL3. A synchronous external abort is taken to EL2 only if all of the following are true:

- SCR_EL3.EA is 0.
- EL2 is enabled and implemented in the current Security state.
- Execution is at an Exception level below EL3.
- If any of the following is true:
  - HCR_EL2.TEA is 1.
  - HCR_EL2.TGE is 1.
  - Execution is at EL2.

A synchronous external abort is taken to EL3 if any of the following are true:

- If executing at any Exception level and SCR_EL3.EA is 0b1.
- If executing at EL3.

SVE synchronous memory faults

In this section, the term Memory fault refers to the detection of an erroneous condition or debug event as a result of performing a data memory access for an SVE load or store instruction.

When an SVE load or store instruction results in a data memory access, the detection of any of the following conditions is considered to be a Memory fault:

- MMU fault.
- Alignment fault, excluding the SP alignment fault.
- Synchronous External abort, including synchronous parity error or ECC error.
- Watchpoint debug event.

For more information, see Memory aborts on page D8-5180.

The detection or generation of a Memory fault by an SVE load or store instruction may or may not cause a synchronous Data Abort or Watchpoint exception to be taken.

Unless otherwise specified in this section, SVE vector load and store instructions that detect a Memory fault cause a Data Abort or Watchpoint exception to be taken, as described in SVE Data Abort and Watchpoint exceptions on page D1-4649.

A Memory fault detected for a memory location that can only be accessed by an Inactive element of a predicated SVE vector load or store instruction is ignored and does not cause a Data Abort or Watchpoint exception to be taken by that instruction.
**SVE Data Abort and Watchpoint exceptions**

**RKJPTS**

Unless otherwise specified in this section, a Data Abort and Watchpoint exception caused by an SVE load or store instruction follows the behaviors described in Definition of a precise exception and imprecise exception on page D1-4638, Exception entry on page D1-4641, and Prioritization of Synchronous exceptions taken to AArch64 state on page D1-4650.

**IDGSNC**

SVE load and store instructions can generate a sequence of single-copy atomic memory accesses that might not be completed due to a Memory fault causing a Data Abort or Watchpoint exception to be taken during the memory access sequence.

**RZKBRX**

When the execution of an SVE load or store instruction detects multiple Memory faults caused by different single-copy atomic memory accesses, the Memory faults are not prioritized by the architecture.

**IMSVYK**

When an SVE load or store instruction that has not been architecturally executed is restarted after an exception return, any memory locations that it accessed before taking the exception might be accessed again. Therefore, SVE load or store instructions might perform multiple accesses to memory locations that do not cause a Memory fault but which are sensitive to the number of accesses, or have been modified between the accesses.

**RZNXNXT**

When execution of an SVE load instruction causes a Data Abort or Watchpoint exception to be taken and the destination is not a vector register that is also used as a base or index register by the instruction, then all elements of the destination register become UNKNOWN.

**RSNQJR**

When execution of an SVE load instruction causes a Data Abort or Watchpoint exception to be taken and the destination is a vector register that is also used as a base or index register by the instruction, then all elements of the destination vector register are restored to their original value prior to execution of the load instruction.

**RDWYCY**

When execution of an SVE predicated vector store instruction causes a Data Abort or Watchpoint exception to be taken, one or more of the following occurs:

- Memory locations that are associated with Active elements and which do not generate a Memory fault become UNKNOWN.
- Memory locations that are associated with Active elements and which generate a Memory fault are unchanged.
- Memory locations that are only associated with Inactive elements are unchanged.

**SVE First-fault and Non-fault loads**

**IJBGW**

When a memory access performed for the First active element of an SVE First-fault vector load instruction detects a Memory fault, this causes a synchronous exception to be taken as described in SVE Data Abort and Watchpoint exceptions.

**IDXING**

When a memory access performed for the First active element of an SVE First-fault vector load instruction does not detect a Memory fault, the other elements are handled in the same way as the elements of an SVE Non-fault vector load instruction.

**RJKGJY**

A Data Abort or Watchpoint exception is not generated when a Memory fault is detected by a memory access performed for any of the following elements:

- Any Active element of an SVE Non-fault vector load.
- Any Active element of an SVE First-fault vector load except for the First active element.

**RMNKNV**

The PE can choose to suppress a memory access performed for any of the following elements:

- Any Active element of an SVE Non-fault vector load.
- Any Active element of an SVE First-fault vector load except for the First active element.

**RHYFRN**

When a memory access performed for any of the following elements detects a Memory fault or is suppressed for any other reason, the FFR predicate elements starting from that element number, up to and including the highest-numbered element, are set to FALSE:

- Any Active element of an SVE Non-fault vector load.
- Any Active element of an SVE First-fault vector load except for the First active element.
An FFR predicate element is never set to TRUE by an SVE vector load, therefore the fault indications are cumulative.

After an SVE Non-fault vector load or First-fault vector load is executed, each destination vector element contains one of the values listed in the following table:

<table>
<thead>
<tr>
<th>Corresponding FFR element</th>
<th>Vector element status</th>
<th>Content of destination vector element</th>
</tr>
</thead>
</table>
| FALSE                     | Active                | Each byte of the element contains an independently CONSTRAINED UNPREDICTABLE choice of one of the following:  
  • 0.  
  • The previous value of that byte in the destination vector register.  
  • If and only if all of the following apply, the value read from memory:  
    — The memory access for that byte was not an access to any type of Device memory.  
    — The memory access for that byte does not return information that cannot be accessed at the current or a lower level of privilege. |
| FALSE                     | Inactive              | A CONSTRAINED UNPREDICTABLE choice of:  
  • 0.  
  • The previous value of that vector element. |
| TRUE                      | Active                | The value read from memory. |
| TRUE                      | Inactive              | 0. |

In the previous table, watchpoints are not a mechanism for preventing access to memory.

**Priority of Synchronous exceptions taken to AArch64 state**

The following list shows the priorities for Synchronous exceptions taken to an Exception level using AArch64. The highest priority is 1. The AArch64 Priority numbers correlate with the equivalent AArch32 and Debug prioritization lists.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Synchronous exception type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reset Catch debug event. See Reset Catch debug events on page H3-10293.</td>
</tr>
<tr>
<td>2</td>
<td>Exception Catch debug event if it has a priority of 2. See Reset Catch debug events on page H3-10293.</td>
</tr>
<tr>
<td>3</td>
<td>Halting Step debug event. See Halting Step debug events on page H3-10276.</td>
</tr>
<tr>
<td>4</td>
<td>Software Step Exception used during Debug. See Software Step exceptions on page D2-4742.</td>
</tr>
<tr>
<td>5</td>
<td>Exception Catch debug events. See Exception Catch debug event on page H3-10287.</td>
</tr>
<tr>
<td>6</td>
<td>PC alignment fault exception. See PC alignment checking on page D1-4668.</td>
</tr>
<tr>
<td>7</td>
<td>Instruction Abort exceptions, including exceptions generated by a Translation Table Walk not prioritized as 29. See MMU fault prioritization from a single address translation stage on page D8-5190.</td>
</tr>
<tr>
<td>8</td>
<td>Breakpoint exceptions, or AArch32 Address Matching Vector Catch exceptions. See Breakpoint exceptions on page D2-4707 and Vector Catch exceptions on page D2-4741.</td>
</tr>
<tr>
<td>9</td>
<td>Illegal Execution state exceptions. See Illegal exception returns from AArch64 state on page D1-4645.</td>
</tr>
<tr>
<td>10</td>
<td>Software Breakpoint exceptions, caused by execution of a Breakpoint instruction.</td>
</tr>
</tbody>
</table>
D1.3 Exceptions

11 Branch Target exception. See PSTATE.BTYPE on page D8-5133.

12 Exceptions taken from EL1 to EL2 because of the configuration of one of the following:
   • HSTR_EL2.Tn.
   • HCR_EL2.TIDCP.
   • If FEAT_NV is implemented, HCR_EL2.{NV1, NV}.
Exceptions taken from EL0 to EL1 or EL2 because of the configuration of the following:
   • If FEAT_TIDCP1 is implemented, SCTLR_EL1.TIDCP.
Exceptions taken from EL0 to EL2 because of the configuration of the following:
   • If FEAT_TIDCP1 is implemented, SCTLR_EL2.TIDCP.

13 Exceptions that occur as a result of attempting to execute an instruction that is UNDEFINED for one or more of the following reasons:
   • Attempting to execute a uDF instruction.
   • Attempting to execute an unallocated instruction encoding, including an encoding for an instruction that is not implemented in the PE implementation.
   • Attempting to execute an instruction that is defined to never be accessible at the current Exception level and Security state, regardless of any enables or traps.
   • Debug state execution of an instruction encoding that is not accessible in Debug state.
   • Non-debug state execution of an instruction encoding that is not accessible in Non-debug state.
   • If HVC instructions are disabled by SCR_EL3.HCE or HCR_EL2.HCD, attempting to execute an HVC instruction.
   • If SPSel.SP is 0, attempting to execute an MSR or MRS instruction to SP_EL0.
   • If HCR_EL2.E2H is 0, attempting to execute an MSR or MRS instruction using an _EL12 register name.
   • If FEAT_FGT and FEAT_PMUv3 are implemented, attempting to execute an MSR or MRS instruction in AArch64 state, or an MCR or MRC instruction in AArch32 state, that accesses a register associated with an unimplemented event counter.
   • If HLT instructions are disabled by EDSCR.HDE or halting is prohibited, attempting to execute an HLT instruction.
   • If in Debug state:
     — If HCR_EL2.TGE is 1, attempting to execute a DCPS1 instruction in Non-secure EL0.
     — If EL2 is disabled in the current Security state or is not implemented, attempting to execute a DCPS2 instruction in EL1 or EL0.
     — If EDSCR.SDD is 1 or EL3 is not implemented, attempting to execute a DCPS3 instruction.
     — If EDSCR.SDD is 1, attempting to execute in EL2, EL1, or EL0 an instruction that is configured by EL3 control registers to trap to EL3. It is IMPLEMENTATION DEFINED whether this type of exception is prioritized at this level or has the priority of the original trap exception.
   • If in AArch32 state:
     — If SCTLR_EL1.ITD is 1, attempting to execute an instruction in an IT block.
     — If SCTLR_EL1.SED is 1, attempting to execute a SETEND instructions.
     — If SCTLR_EL1.CP15BEN is 0, attempting to execute a CP15DSB, CP15SDB, or CP15SBE barrier instruction.
   • Attempting to execute an instruction that is UNDEFINED because all of the following apply:
     — EL0 is using AArch32.
     — EL1 is using AArch64.
     — Programming FPCR. {Stride, Len} to nonzero values is supported.
     — One or more of FPCR. {Stride, Len} is nonzero.

14 If all of the following apply, any exception that is generated by a configurable trap or enable:
   • The exception is not already covered by priorities 4-13.
   • The exception is taken to EL1, or if HCR_EL2.TGE is 1, is taken to EL2.
15 As the result of a configuration control in one of the following registers, an exception taken from EL0 to EL2:
   • When the exception is taken to AArch64 state:
     — HSTR_EL2.T
     — HCR_EL2.TIDCP.
   • When the exception is taken to AArch32 state:
     — HSTR.Tn.
     — HCR.TIDCP.

16 As the result of a configuration control in one of the following registers, an exception taken to EL2:
   • When the exception is taken to AArch64 state: CPTR_EL2.
   • When the exception is taken to AArch32 state: in HCPTR.

17 As the result of a configuration control in one of the following registers, an exception taken to EL2:
   • When the exception is taken to AArch64 state:
     — HCR_EL2: Other than a setting made in the {TIDCP, NV} fields, and MRS/MSR instruction using an _EL12
     register name with HCR_EL2.E2H is 0.
     — CNTHCTL_EL2: Any setting.
     — MDCR_EL2: Any setting.
     — Any of the following fine-grained traps:
       — HAFGRTR_EL2.
       — HDFGRTR_EL2.
       — HDFGWTR_EL2.
       — HFGITR_EL2.
       — HFGTR_EL2.
       — HFGWTR_EL2.
   • When the exception is taken to AArch32 state:
     — HCR: Other than the TIDCP bit.
     — CNTHCTL: Any setting.
     — HDCR: Any setting.

18 Other than an exception defined by priorities 4-17 inclusive, any exception that is the result of a configurable access to
instructions, and the exception is taken to EL2.

19 An exception caused by the SMC instruction being UNDEFINED because SCR_EL3.SMD is 1.

20 An exception caused by any of the following Exception producing instructions:
   • HVC.
   • SMC.
   • SVC.
For AArch64 State, see Branches, Exception generating, and System instructions on page C3-248.

21 An exception taken to EL3 as the result of configuration controls in CPTR_EL3. It is IMPLEMENTATION DEFINED whether
the exception is prioritized as an UNDEFINED instruction or has the priority of the original Trap exception.

22 Exceptions Trapped to EL3 from Secure EL1 using AArch32.

23 An Exception taken to EL3 from EL0, EL1, or EL2 as a result of configuration controls in MDCR_EL3. It is
IMPLEMENTATION DEFINED whether the exception is prioritized as an UNDEFINED instruction or has the priority of the
original Trap exception.

24 Other than an exception defined by priorities 4-23, inclusive, any exception taken to EL3 because of a configurable
access to an instruction. It is IMPLEMENTATION DEFINED whether the exception is prioritized as an UNDEFINED instruction
or has the priority of the original Trap exception.
Trapping of floating-point exceptions

Execution of a floating-point instruction, or execution of an Advanced SIMD or SVE instruction that performs floating-point operations, can generate an exceptional condition, called a floating-point exception.

For each of the following floating-point exceptions, it is IMPLEMENTATION DEFINED whether an implementation includes synchronous exception generation:

- Input Denormal.
- Inexact.
- Underflow.
- Overflow.
- Divide by Zero.
- Invalid Operation.

The architecture does not support asynchronous reporting of floating-point exceptions. If an implementation does not support synchronous exception generation from a floating-point exception, then that synchronous exception is never generated and all statements on when that synchronous exception is generated do not apply.

For any of the implemented floating-point exceptions listed in \texttt{R\textsubscript{BBSGN}}, FPCR provides control bits to enable synchronous exception generation.
The Exception level that a Trapped Floating-point exception is taken to is defined as follows:

- If executing at EL0:
  - If the Effective value of HCR_EL2.TGE is 0, the exception is taken to EL1.
  - If the Effective value of HCR_EL2.TGE is 1, the exception is taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.
- If executing at EL3, the exception is taken to EL3.

If an implementation includes synchronous exception generation for floating-point exceptions in AArch64 state, when the execution of separate operations in separate SIMD or SVE elements causes multiple floating-point exceptions, the ESR_ELx reports only the exceptions associated with one element that the instruction uses. The architecture does not specify which element is reported.

When a floating-point exception is trapped, all of the following apply:

- When the trapped floating-point exception is taken, it is IMPLEMENTATION DEFINED whether the FPSR is restored to the value of the FPSR immediately before the instruction that generated the trapped floating-point exception. When the trapped floating-point exception is taken, if the FPSR is not restored, it is CONSTRAINED UNPREDICTABLE which untrapped floating-point exceptions, if any, are indicated by the corresponding FPSR cumulative floating-point exception bits having the value 1.
- In ESR_ELx at the target Exception level, all of the following apply:
  - The highest priority trapped floating-point exception has a floating-point exception trapped bit set to 1.
  - If any other untrapped floating-point exceptions are generated by the same operation, each untrapped exception has a floating-point exception trapped bit set to 0. This applies to both higher priority and lower priority untrapped floating-point exceptions.
  - If any lower priority trapped floating-point exceptions are generated by the same operation, for each exception, it is CONSTRAINED UNPREDICTABLE whether the floating-point exception trapped bit is set to 1.

For trapped floating-point exceptions from Advanced SIMD or SVE instructions, the architecture does not define the floating-point exception prioritization between different elements of the instruction.

**Memory Copy and Memory Set exceptions**

If an exception is taken during the execution of Memory Copy and Memory Set instructions, and execution of these instructions restarts on a physical hardware PE implementation that is different from where the exception was taken from, then a Memory Copy or Memory Set exception can be generated.

For CPYE*, CPYFE*, CPYFM*, and CPYM* instructions, if the instruction algorithm option indicated by the value of PSTATE.C is different from the PSTATE.C value of the implementation, then a Memory Copy exception is generated.

For SETE*, SETGE*, SETGM*, and SETM* instructions, if the instruction algorithm option indicated by the value of PSTATE.C is different from the PSTATE.C value of the implementation, then a Memory Set exception is generated.

For CPYE*, CPYFE*, CPYFM*, and CPYM* instructions, if the instruction alignment and size requirements of the implementation that an exception is taken from are different from the alignment and size requirements of the implementation that is returned to, then a Memory Copy exception is generated.

For SETE*, SETGE*, SETGM*, and SETM* instructions, if the instruction alignment and size requirements of the implementation that an exception is taken from are different from the alignment and size requirements of the implementation that is returned to, then a Memory Set exception is generated.

The Memory Copy and Memory Set exceptions that are caused by alignment and size requirement differences can be avoided if the following apply:

- The CPYFP*, CPYP*, SETGP*, or SETP* instruction is executed before the appropriate CPYFM*, CPYM*, SETGM*, or SETM* on the same implementation.
• The CPYFM*, CPYM*, SETGM*, or SETM* instruction is executed before the appropriate CPYFE*, CPYE*, SETGE*, or SETE* on the same implementation.

The generic return routine for Memory Copy and Memory Set exceptions is to adjust the register arguments to be consistent with performing a new memory copy or memory set, and then to return to the CPYFP*, CPYP*, SETGP*, or SETP* instruction to restart the memory copy or memory set.

The generic return routine for Memory Copy and Memory Set exceptions can be used by relying on the Memory Copy and Memory Set instructions being placed consecutively in memory - prologue, then main, and then epilogue.

The generic return routine for Memory Copy exceptions is shown in the following pseudocode:

```c
if OptionA && WrongOption || OptionB && !WrongOption then
    // format is from option B
    if SPSR_ELx.N=='1' then
        // backward copy
        DestReg = DestReg - SizeReg;
        SrcReg = SrcReg - SizeReg;
    else // format is from option A;
        if SizeReg<63>=='1' then
            // forward copy
            DestReg = DestReg + SizeReg;
            SrcReg = SrcReg + SizeReg;
            SizeReg = -1*SizeReg;
            if FromEpilogue then
                ReturnAddress = ELR_ELx - 8;
            else
                ReturnAddress = ELR_ELx - 4;
        end
    end
end
```

The generic return routine for Memory Set exceptions is shown in the following pseudocode:

```c
if !OptionA && WrongOption || (OptionA && !WrongOption) then
    // format is from Option A
    // forward set
    DestReg = DestReg + SizeReg;
    SizeReg = -1*SizeReg;
    if FromEpilogue then
        ReturnAddress = ELR_ELx - 8;
    else
        ReturnAddress = ELR_ELx - 4;
end
```

The information in the ESR_ELx.ISS field is sufficient to allow the diagnosis of the reason for a Memory Copy or Memory Set exception being generated and to allow a generic emulation of the memory copy or memory set.

The Exception levels that Memory Copy and Memory Set exceptions are taken to are as follows:

• From EL0, to EL1 if HCR_EL2.TGE is 0. Otherwise, they are taken to EL2.
• From EL1, to either EL1 or EL2, depending on the value of HCRX_EL2.MCE2.
• From EL2 or EL3, to the same Exception level.

The HCRX_EL2.MCE2 control bit allows Memory Copy and Memory Set exceptions to be handled in a system where Memory Copy and Memory Set instructions might be used at EL1 at times when taking an exception to EL1 is not acceptable. This is necessary only if there is a possibility of the migration of EL1 code between different physical hardware PE implementations when EL1 is not tolerant of taking an exception.
D1.3.6 Asynchronous exception types

The Arm architecture exception model distinguishes two classes of asynchronous exceptions:

- Physical Interrupts
- Virtual Interrupts

There are three types of physical interrupt:

- SError (also described as a System Error).
- IRQ.
- FIQ.

There are three types of virtual interrupt:

- vSError (also described as a Virtual System Error).
- vIRQ.
- vFIQ.

IRQ, FIQ, vIRQ, and vFIQ interrupts are precise asynchronous exceptions.

If FEAT_NMI is implemented and SCTLR_ELx.NMI is 1, IRQ, FIQ, vIRQ, and vFIQ interrupts can have Superpriority as an additional attribute. When SCTLR_ELx.NMI is 0, the denotation of Superpriority has no effect on these interrupts. See also R_MHWHB, R_GFXXY, R_SNLHR, and R_XSPSG.

The mechanism by which Superpriority is assigned is IMPLEMENTATION DEFINED.

Each physical interrupt type can be assigned a target Exception level of EL1, EL2, or EL3.

ISR_EL1 shows the pending status of interrupts as follows:

- When read from EL2 or EL3, ISR_EL1 shows the pending status of each of the physical interrupts IRQ, FIQ, and SError.
- When read from EL1, ISR_EL1 shows:
  - The pending status of the virtual interrupts, vIRQ, vFIQ, and vSError if they are enabled by the Effective values of the corresponding HCR_EL2.{IMO, FMO, AMO} enables.
  - The pending status of the physical interrupts, IRQ, FIQ, and SError if the corresponding virtual interrupts are not enabled by the Effective values of the HCR_EL2.{IMO, FMO, AMO} enables.

An implementation might support other mechanisms for signaling a virtual interrupt.

The mechanism by which physical interrupts are signaled to the PE are IMPLEMENTATION DEFINED and might be either edge or level sensitive. A common implementation choice is that the IRQ and FIQ interrupts are level sensitive, and this is supported by the Generic Interrupt Controller (GIC).

The physical SError interrupt is often used, amongst other things, for communicating External aborts from the memory system that are to be taken asynchronously.

For an External abort generated by the memory system that is taken asynchronously using the SError interrupt, the SError interrupt always behaves as an edge-triggered interrupt. For any other sources of SError interrupts, it is IMPLEMENTATION DEFINED whether they are edge-triggered or level-sensitive.

When taking an SError or a vSError interrupt to an Exception level using AArch64, ESR_ELx at the target Exception level is updated to describe an SError interrupt. If the RAS extension is implemented, when taking a vSError interrupt to an Exception level using AArch64, ESR_ELx at the target Exception level is updated with exception syndrome information from VESER_EL2.

When taking an IRQ, vIRQ, FIQ or vFIQ interrupt to an Exception level using AArch64, ESR_ELx at the target Exception level is not updated.
Virtual interrupts

Each virtual interrupt type can be independently enabled from EL2. If a virtual interrupt type is enabled from EL2, the target Exception level for the corresponding physical interrupt is not EL1.

Each virtual interrupt type can be set to the pending state by EL2 using controls in HCR_EL2.

Pending vIRQ and vFIQ interrupts can be assigned Superpriority by using controls in HCRX_EL2.

If HCR_EL2.TGE is 0, setting an HCR_EL2.{FMO, IMO, AMO} routing control bit to 1 enables the corresponding virtual interrupt. If HCR_EL2.TGE is 1, all virtual interrupts are disabled and the Effective values of HCR_EL2{FMO, IMO, AMO} are 0.

If a virtual interrupt type is enabled, that type of interrupt can be generated by any one of the following:
- Execution at EL0 or EL1 if the corresponding virtual interrupt pending bit, HCR_EL2.{VSE, VI, VF}, is 1.
- For a vIRQ or a vFIQ, by an IMPLEMENTATION DEFINED mechanism. This might be a signal from an interrupt controller.

If a virtual interrupt is disabled, the virtual interrupt cannot be taken.

The following table describes the bits that enable virtual interrupts, the bits that cause virtual interrupts to be pending in HCR_EL2, and the bits that enable signaling of pending virtual interrupts with Superpriority in HCRX_EL2:

<table>
<thead>
<tr>
<th>Virtual interrupt type</th>
<th>Enable control</th>
<th>Cause a virtual interrupt to be pending</th>
<th>Pending virtual interrupt has Superpriority</th>
</tr>
</thead>
<tbody>
<tr>
<td>vSError</td>
<td>HCR_EL2.AMO</td>
<td>HCR_EL2.VSE</td>
<td>n/a</td>
</tr>
<tr>
<td>vIRQ</td>
<td>HCR_EL2.IMO</td>
<td>HCR_EL2.VI</td>
<td>HCRX_EL2.VINMI</td>
</tr>
<tr>
<td>vFIQ</td>
<td>HCR_EL2.FMO</td>
<td>HCR_EL2.VF</td>
<td>HCRX_EL2.VFNMI</td>
</tr>
</tbody>
</table>

When taking a vIRQ or a vFIQ interrupt, the corresponding virtual interrupt pending bit in HCR_EL2 retains its state.

If the virtual interrupt pending bits are used, the vIRQ or vFIQ exception handler must cause software executing at EL2 or EL3 to set their corresponding virtual interrupt pending bits to 0.

When taking a vSError interrupt, HCR_EL2.VSE is cleared to 0.

Establishing the target Exception level of an asynchronous exception

The terms used in the table in this section have the following meanings:

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCR_EL3</td>
<td>The Effective value of a field in SCR_EL3.</td>
</tr>
<tr>
<td>EEL2</td>
<td>If EL3 is not implemented, the Effective value of SCR_EL3.EEL2 is 1.</td>
</tr>
<tr>
<td>FIQ</td>
<td>The Effective value of the field that configures the asynchronous exception type in SCR_EL3. The Effective value of the field is one of the following:</td>
</tr>
<tr>
<td>IRQ</td>
<td>• If EL3 is implemented, the FIQ/IRQ/EA fields are taken from SCR_EL3.</td>
</tr>
<tr>
<td>EA</td>
<td>• If EL3 is not implemented, the Effective value of these fields is 0.</td>
</tr>
<tr>
<td>RW</td>
<td>If EL3 is not implemented, the Effective value of SCR_EL3.RW is 1.</td>
</tr>
<tr>
<td>HCR</td>
<td>If EL2 is using AArch32, this is the Effective value of a field in HCR. If EL2 is using AArch64, this is the Effective value of a field in HCR_EL2.</td>
</tr>
</tbody>
</table>
The AArch64 System Level Programmers' Model

### D1.3 Exceptions

#### RNMMXK

The following table describes the routing of physical interrupts if the highest implemented Exception level is using AArch64.

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TGE</td>
<td>If EL2 is not implemented, the Effective value of HCR.TGE or HCR_EL2.TGE is 0.</td>
</tr>
<tr>
<td>FMO</td>
<td>The Effective value of the mask override field for the asynchronous exception type in HCR or HCR_EL2. The Effective value of the field is one of the following:</td>
</tr>
<tr>
<td>IMO</td>
<td>• If EL2 is implemented, the FMO/IMO/AMO fields are taken from HCR_EL2.</td>
</tr>
<tr>
<td>AMO</td>
<td>• If EL2 is not implemented, the Effective value of these fields is 0.</td>
</tr>
<tr>
<td>E2H</td>
<td>If EL2 is not implemented, the Effective value of HCR.E2H or HCR_EL2.E2H is 0.</td>
</tr>
<tr>
<td>RW</td>
<td>If EL2 is not implemented, the Effective value of HCR_EL2.RW is the same as the Effective value of SCR_EL3.RW.</td>
</tr>
<tr>
<td>EL1</td>
<td>The exception is taken to EL1 using AArch64.</td>
</tr>
<tr>
<td>EL2</td>
<td>The exception is taken to EL2 using AArch64.</td>
</tr>
<tr>
<td>EL3</td>
<td>The exception is taken to EL3 using AArch64.</td>
</tr>
<tr>
<td>C</td>
<td>The interrupt is not taken and remains pending, regardless of the PSTATE.{A, I, F, ALLINT, SP} interrupt masks.</td>
</tr>
<tr>
<td>FIQ IRQ Abt</td>
<td>The exception is taken to AArch32 FIQ mode, the AArch32 IRQ mode or the AArch32 Abort mode according to the type of asynchronous exception.</td>
</tr>
<tr>
<td>Hyp</td>
<td>The exception is taken to AArch32 Hyp mode.</td>
</tr>
<tr>
<td>Mon</td>
<td>The exception is taken to AArch32 Monitor mode.</td>
</tr>
<tr>
<td>n/a</td>
<td>Not applicable. The field does not exist in the register in this configuration or the Exception level is not accessible in this configuration.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCR_EL3</th>
<th>HCR</th>
<th>Target when taken from</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS</td>
<td>EEL2</td>
<td>EA</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
</tr>
</tbody>
</table>
Asynchronous exception masking

An interrupt can be masked. A masked interrupt is not taken and remains pending.

Whether an interrupt is masked is determined by all of the following:
- The PSTATE.{A, I, F, ALLINT, SP} bits.
- The value of AllIntMask, as described in RXZPDT.
- Whether an IRQ or FIQ interrupt has Superpriority.
- The implemented Exception levels.
- The configuration of the Hypervisor Configuration Register.
- The configuration of the Secure Configuration Register.

If in AArch64 state and the target Exception level of an interrupt is lower than the current Exception level, the interrupt is masked.

If FEAT_NMI is implemented and SCTLR_ELx.NMI is 1, all of the following apply:
- The AllIntMask variable is created and is defined as follows:

  \[
  \text{AllIntMask} = \text{PSTATE.ALLINT} || (\text{PSTATE.SP} \&\& \text{SCTLR_ELx.SPINTMASK})
  \]
- When AllIntMask is 1 and execution is at ELx, an IRQ or FIQ interrupt that is targeted to ELx is masked regardless of Superpriority.
- When AllIntMask is 0 and execution is at ELx:
  - An IRQ or FIQ interrupt with Superpriority that is targeted to ELx is not masked by PSTATE.I or PSTATE.F, respectively.
  - An IRQ or FIQ interrupt without Superpriority that is targeted to ELx is masked by PSTATE.I or PSTATE.F, respectively, in the same way as when the Effective value of SCTLR_ELx.NMI is 0.

AllIntMask does not affect the masking or routing of interrupts to a higher Exception level.

If FEAT_NMI is not implemented or SCTLR_ELx.NMI is 0, the Effective value of AllIntMask is 0.
Physical interrupt masking

If the target Exception level of a physical IRQ interrupt is the current Exception level, ELx, the following controls determine whether the interrupt is masked:

<table>
<thead>
<tr>
<th>PSTATE.I</th>
<th>SCTLR_ELx.NMI</th>
<th>AllIntMask</th>
<th>IRQ</th>
<th>IRQ with Superpriority</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>x</td>
<td>Not masked</td>
<td>Not masked</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>Not masked</td>
<td>Not masked</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>Masked</td>
<td>Masked</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>x</td>
<td>Masked</td>
<td>Masked</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0</td>
<td>Masked</td>
<td>Not masked</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>Masked</td>
<td>Masked</td>
<td></td>
</tr>
</tbody>
</table>

If the target Exception level of a physical FIQ interrupt is the current Exception level, ELx, the following controls determine whether the interrupt is masked:

<table>
<thead>
<tr>
<th>PSTATE.F</th>
<th>SCTLR_ELx.NMI</th>
<th>AllIntMask</th>
<th>FIQ</th>
<th>FIQ with Superpriority</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>x</td>
<td>Not masked</td>
<td>Not masked</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>Not masked</td>
<td>Not masked</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>Masked</td>
<td>Masked</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>x</td>
<td>Masked</td>
<td>Masked</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0</td>
<td>Masked</td>
<td>Not masked</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>Masked</td>
<td>Masked</td>
<td></td>
</tr>
</tbody>
</table>

If the target Exception level of a physical SError interrupt is the current Exception level, the PSTATE.A control determines whether the interrupt is masked. However, if all of the following are true, the PSTATE.A control is ignored, and the interrupt is taken:

- The target Exception level is the current Exception level.
- FEAT_DoubleFault is implemented.
- SCR_EL3.NMEA is 1.

If the target Exception level of a physical interrupt is higher than the current Exception level, all of the following apply:

- If the target Exception level is EL3, the interrupt cannot be masked by the PSTATE.{A, I, F} bits.
- If the target Exception level is EL2 and any of the following are true, the interrupt cannot be masked by the PSTATE.{A, I, F} bits:
  - HCR_EL2.E2H is 0.
  - HCR_EL2.TGE is 0.
- If the target Exception level is EL2 and all of the following are true, the interrupt can be masked by the PSTATE.{A, I, F} bits:
  - HCR_EL2.E2H is 1.
  - HCR_EL2.TGE is 1.
  - The interrupt does not have Superpriority or SCTLR_EL2.NMI is 0.
• If the target Exception level is EL1 and the interrupt does not have Superpriority or SCTLR_EL1.NMI is 0, the interrupt can be masked by the PSTATE. {A, I, F} bits.

The ability to execute in EL0 with interrupts taken to EL1 masked is required by some user level driver code.

The terms used in the table in this section have the following meanings:

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCR_EL3</td>
<td>This is the Effective value of a field in SCR_EL3.</td>
</tr>
<tr>
<td>EEL2</td>
<td>If EL3 is not implemented, the Effective value of SCR_EL3.EEL2 is 1. If FEAT_SEL2 is not implemented, the Effective value of SCR_EL3.EEL2 is 0.</td>
</tr>
<tr>
<td>FIQ</td>
<td>The Effective value of the field that configures the asynchronous exception type in SCR_EL3. If EL3 is implemented, the FIQ/IRQ/EA field are taken from SCR_EL3. If EL3 is not implemented, then the Effective value of these fields is 0.</td>
</tr>
<tr>
<td>IRQ</td>
<td></td>
</tr>
<tr>
<td>EA</td>
<td></td>
</tr>
<tr>
<td>RW</td>
<td>If EL3 is not implemented, the Effective value of SCR_EL3.RW is 1.</td>
</tr>
<tr>
<td>HCR</td>
<td>If EL2 is using AArch32, this is the Effective value of a field in HCR. If EL2 is using AArch64, this is the Effective value of a field in HCR_EL2.</td>
</tr>
<tr>
<td>TGE</td>
<td>If EL2 is not implemented, the Effective value of HCR.TGE or HCR_EL2.TGE is 0.</td>
</tr>
<tr>
<td>E2H</td>
<td>If EL2 is not implemented, the Effective value of HCR.E2H or HCR_EL2.E2H is 0.</td>
</tr>
<tr>
<td>FMO</td>
<td>The Effective value of the mask override field for the asynchronous exception type in HCR or HCR_EL2. The Effective value of the field is one of the following:</td>
</tr>
<tr>
<td>IMO</td>
<td>• If EL2 is implemented, the FMO/IMO/AMO fields are taken from HCR_EL2.</td>
</tr>
<tr>
<td>AMO</td>
<td>• If EL2 is not implemented, the Effective value of these fields is 0.</td>
</tr>
<tr>
<td>A</td>
<td>When the interrupt is pending, it is taken regardless of the value of the PSTATE. {A, I, F, ALLINT, SP} interrupt masks.</td>
</tr>
<tr>
<td>B</td>
<td>When the interrupt is pending, it might be subject to masking, as defined in RMHWBP. If the interrupt is masked, it is not taken. If the interrupt is not masked, it is taken.</td>
</tr>
<tr>
<td>A/B</td>
<td>SEError interrupts behave as A. FIQ and IRQ interrupts behave as B.</td>
</tr>
<tr>
<td>C</td>
<td>When the interrupt is pending, it is not taken, regardless of the value of the PSTATE. {A, I, F, ALLINT, SP} interrupt masks.</td>
</tr>
<tr>
<td>n/a</td>
<td>Not applicable. The PE cannot be executing at this Exception level for the specified state of HCR and SCR_EL3.</td>
</tr>
</tbody>
</table>

The following table describes the target and masking of physical interrupts, if the highest implemented Exception level is using AArch64.

<table>
<thead>
<tr>
<th>SCR_EL3</th>
<th>HCR</th>
<th>Effect of the interrupt mask when executing at:</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS</td>
<td>EEL2</td>
<td>EAI</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Virtual interrupt masking

If the target Exception level of a virtual IRQ interrupt is the current Exception level, ELx, the following controls determine whether the interrupt is masked:

<table>
<thead>
<tr>
<th>PSTATE.I</th>
<th>SCTLR_ELx.NMI</th>
<th>AllIntMask</th>
<th>vIRQ</th>
<th>vIRQ with Superpriority</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 x</td>
<td></td>
<td></td>
<td></td>
<td>Not Masked</td>
</tr>
<tr>
<td>0 1 0</td>
<td></td>
<td></td>
<td></td>
<td>Not Masked</td>
</tr>
<tr>
<td>0 1 1</td>
<td></td>
<td></td>
<td></td>
<td>Masked</td>
</tr>
<tr>
<td>1 0 x</td>
<td></td>
<td>Masked</td>
<td></td>
<td>Masked</td>
</tr>
<tr>
<td>1 1 0</td>
<td></td>
<td>Masked</td>
<td></td>
<td>Not Masked</td>
</tr>
<tr>
<td>1 1 1</td>
<td></td>
<td>Masked</td>
<td></td>
<td>Masked</td>
</tr>
</tbody>
</table>

Virtual interrupt masking
RSNLJH
If the target Exception level of a virtual FIQ interrupt is the current Exception level, ELx, the following controls determine whether the interrupt is masked:

<table>
<thead>
<tr>
<th>PSTATE.F</th>
<th>SCTLR_ELx.NMI</th>
<th>AllIntMask</th>
<th>vFIQ</th>
<th>vFIQ with Superpriority</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>x</td>
<td>Not Masked</td>
<td>Not Masked</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Not Masked</td>
<td>Not Masked</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Masked</td>
<td>Masked</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>x</td>
<td>Masked</td>
<td>Masked</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Masked</td>
<td>Not Masked</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Masked</td>
<td>Masked</td>
</tr>
</tbody>
</table>

Virtual interrupts can only be taken from EL0 or EL1 and can only be taken to EL1. If EL2 is not enabled in the current Security state, all types of virtual interrupts are always masked. If executing at EL2 or EL3, all types of virtual interrupts are always masked.

If the target Exception level of a virtual SError interrupt is the current Exception level, ELx, the PSTATE.A control determines whether the interrupt is masked. However, if all of the following are true, the PSTATE.A control is ignored, and the interrupt is taken:

- The target Exception level is the current Exception level.
- FEAT_DoubleFault is implemented.
- SCR_EL3.NMEA is 1.

The terms used in the table in this section have the following meanings:

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCR_EL3</td>
<td>This is the Effective value of a field in SCR_EL3.</td>
</tr>
<tr>
<td>HCR</td>
<td>If EL2 is using AArch32, this is the Effective value of a field in HCR. If EL2 is using AArch64, this is the Effective value of a field in HCR_EL2.</td>
</tr>
<tr>
<td>E2H</td>
<td>If EL2 is using AArch32, the Effective value of HCR.E2H is 0.</td>
</tr>
<tr>
<td>B</td>
<td>When the interrupt is pending, it might be subject to masking, as defined in RSNLJH and RXSPSG. If the interrupt is masked, it is not taken. If the interrupt is not masked, it is taken.</td>
</tr>
<tr>
<td>C</td>
<td>When the interrupt is pending, it is not taken, regardless of the value of the PSTATE.{A, I, F, ALLINT, SP} interrupt masks.</td>
</tr>
<tr>
<td>n/a</td>
<td>Not applicable. The PE cannot be executing at this Exception level for the specified state of HCR and SCR_EL3.</td>
</tr>
</tbody>
</table>
The following table describes the masking of virtual interrupts when the highest implemented Exception level is using AArch64:

<table>
<thead>
<tr>
<th>SCR_EL3</th>
<th>HCR</th>
<th>Effect of the interrupt mask when executing at:</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEL2</td>
<td>NS</td>
<td>IRQ</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
</tbody>
</table>

**Prioritization of interrupts**

The prioritization of physical interrupts and virtual interrupts is IMPLEMENTATION DEFINED.

For all of the following Context synchronization events, if an interrupt is pending before the Context synchronization event, and the interrupt remains pending and is not masked after the Context synchronization event, the interrupt is taken before the first instruction after the Context synchronization event:

- Execution of an ISB instruction.
- If FEAT_ExS is not implemented, exception entry.
- If FEAT_ExS is implemented and the appropriate SCTLR_ELx.EIS bit is set, exception entry.
- If FEAT_ExS is not implemented, exception return.
- If FEAT_ExS is implemented and the appropriate SCTLR_ELx.EOS bit is set, exception exit.
- Exit from Debug state.

If the first instruction after the context synchronizing event generates a synchronous exception, the architecture does not define whether the PE takes the interrupt or the synchronous exception first.

A RAS error synchronization event defines additional requirements for taking an SError interrupt.

Other than the behaviors described in RRBZYL, an unmasked, pending interrupt must be taken in finite time.

If an unmasked interrupt was pending but is changed to not pending before it is taken, it is CONSTRAINED UNPREDICTABLE whether or not the interrupt is taken. If the interrupt is taken, it is taken before the first context synchronization event after the interrupt was changed to be not pending.

**Taking an interrupt during a multi-access load or store**

If in AArch64 state, interrupts can be taken during a sequence of memory accesses caused by a single load or store instruction. This is true regardless of the memory type being accessed.
Taking an interrupt during an SVE instruction

Permitting SVE instructions to be interrupted by asynchronous exceptions is IMPLEMENTATION DEFINED.

When returning from an asynchronous exception, an interrupted SVE instruction is restarted and cannot resume at the point the instruction was interrupted.

For Data Aborts taken asynchronously, see Definition of a precise exception and imprecise exception on page D1-4638.

D1.3.7 UNDEFINED instructions

An instruction which is UNDEFINED generates a synchronous exception for that instruction unless there is a higher priority exception generated for that instruction.

The Exception level that the synchronous Undefined Instruction exception is taken to is defined as follows:

- If executing at EL0:
  - If the Effective value of HCR_EL2.TGE is 0, the exception is taken to EL1.
  - If the Effective value of HCR_EL2.TGE is 1, the exception is taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.
- If executing at EL3, the exception is taken to EL3.

D1.3.8 Configurable instruction controls

Configurable instruction controls are control bits held in System registers that determine whether attempting to execute an instruction generates a synchronous exception at the point in the instruction stream of that instruction, and the instruction is not executed.

Configurable instruction controls might be referred to by any of the following names:

- Instruction enables.
- Instruction disables.
- Trap controls.

The definitions of each type overlap, and in some cases is historical. Describing a register control field as an instruction enable, an instruction disable, or a trap control, gives no indication of how an exception that is generated as a consequence of the value of that field is handled or reported. Each configurable instruction control defines how the exception that is generated as a consequence of the configurable instruction control is handled or reported.

An exception can only be generated as a result of a configurable instruction control if all of the following apply:

- The instruction generating the exception does not also generate a higher priority exception.
- The instruction generating the exception is not UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in the PE state in which the instruction is executed.

It is UNPREDICTABLE / CONSTRAINED UNPREDICTABLE whether configurable instruction controls generate an exception when the instruction is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in the PE state in which the instruction is executed.

UNPREDICTABLE and CONSTRAINED UNPREDICTABLE instructions can generate exceptions as a result of configurable instruction controls, but the architecture does not require them to do so.

Note: Many CONSTRAINED UNPREDICTABLE behaviors for instructions include an allowance that the CONSTRAINED UNPREDICTABLE instruction behaves the same way as a closely related instruction that is not CONSTRAINED UNPREDICTABLE. In those cases, the instructions enable, disable, or trap control that causes in exception on the closely related instruction will cause the same exception on the CONSTRAINED UNPREDICTABLE instruction.

An implementation might provide more controls, in IMPLEMENTATION DEFINED registers, to provide control of trapping of IMPLEMENTATION DEFINED features.
When a configurable instruction control causes an exception, the exception is taken and the instruction is not executed, and therefore all of the following are true:

- The preferred exception return address of the exception is the instruction that generates the exception.
- There are no changes to the registers accessed by the instruction, including as a result of side-effects of a register access.

When a configurable instruction control causes a conditional instruction to generate an exception in AArch32 state, it is IMPLEMENTATION DEFINED whether the exception applies to conditional AArch32 instructions that fail their condition code check.

**EL0 and EL1 configurable instruction controls**

The following EL0 and EL1 System registers contain configurable instruction controls:

<table>
<thead>
<tr>
<th>Register name</th>
<th>Register description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMUSERENR_EL0</td>
<td>Activity Monitors User Enable Register</td>
</tr>
<tr>
<td>CPACR_EL1</td>
<td>Architectural Feature Access Control Register</td>
</tr>
<tr>
<td>MDSCCR_EL1</td>
<td>Monitor System Debug Control Register</td>
</tr>
<tr>
<td>PMUSERENR_EL0</td>
<td>Performance Monitors User Enable Register</td>
</tr>
<tr>
<td>SCTLR_EL1</td>
<td>System Control Register (EL1)</td>
</tr>
<tr>
<td>TCR_EL1</td>
<td>Translation Control Register (EL1)</td>
</tr>
</tbody>
</table>

An exception caused by configurable instruction controls in EL1 can be taken from either AArch64 state or AArch32 state.

**EL2 configurable instruction controls**

The following EL2 System registers contain configurable instruction controls:

<table>
<thead>
<tr>
<th>Register name</th>
<th>Register description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPTR_EL2</td>
<td>Architectural Feature Trap Register, EL2</td>
</tr>
<tr>
<td>HAFGRTR_EL2</td>
<td>Hypervisor Activity Monitors Fine-Grained Read Trap Register</td>
</tr>
<tr>
<td>HCR_EL2</td>
<td>Hypervisor Configuration Register</td>
</tr>
<tr>
<td>HCRX_EL2</td>
<td>Extended Hypervisor Configuration Register</td>
</tr>
<tr>
<td>HDFGRTR_EL2</td>
<td>Hypervisor Debug Fine-Grained Read Trap Register</td>
</tr>
<tr>
<td>HDFGWTR_EL2</td>
<td>Hypervisor Debug Fine-Grained Write Trap Register</td>
</tr>
<tr>
<td>HFGITR_EL2</td>
<td>Hypervisor Fine-Grained Instruction Trap Register</td>
</tr>
<tr>
<td>HFGRTTR_EL2</td>
<td>Hypervisor Fine-Grained Read Trap Register</td>
</tr>
<tr>
<td>HFGWTR_EL2</td>
<td>Hypervisor Fine-Grained Write Trap Register</td>
</tr>
<tr>
<td>HSTR_EL2</td>
<td>Hypervisor System Trap Register</td>
</tr>
</tbody>
</table>
An exception caused by configurable instruction controls in EL2 can be taken from either AArch64 state or AArch32 state.

If Secure EL2 is implemented and enabled, configurable instruction controls available at EL2 apply in Secure state. If Secure EL2 is not implemented or not enabled, the configurable instruction controls available at EL2 are ignored in Secure state.

**EL3 configurable instruction controls**

The following EL3 System registers contain configurable instruction controls:

<table>
<thead>
<tr>
<th>Register name</th>
<th>Register description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCTLR_EL3</td>
<td>System Control Register, EL3</td>
</tr>
<tr>
<td>SCR_EL3</td>
<td>Secure Configuration Register</td>
</tr>
<tr>
<td>CPTR_EL3</td>
<td>Architectural Feature Trap Register, EL3</td>
</tr>
<tr>
<td>MDCR_EL3</td>
<td>Monitor Debug Configuration Register, EL3</td>
</tr>
<tr>
<td>TCR_EL3</td>
<td>Translation Control Register, EL3</td>
</tr>
</tbody>
</table>

An exception caused by configurable instruction controls in EL3 can be taken from either AArch64 state or AArch32 state.

**D1.3.9 Exception Producing Instructions**

The exception producing instructions are commonly called system calls and refer to any of the following synchronous exception types:

- Supervisor Call exception.
- Hypervisor Call exception.
- Secure Monitor Call exception.

A Supervisor Call is generated by executing an SVC instruction.

A Supervisor Call exception is taken to the following Exception levels:

- If executing at EL0:
  - If the Effective value of HCR_EL2.TGE is 0, the exception is taken to EL1.
  - If the Effective value of HCR_EL2.TGE is 1, the exception is taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.
- If executing at EL3, the exception is taken to EL3.

A Supervisor Call enables software executing at EL0 to make a call to an operating system or other software executing at EL1.

If EL2 is implemented, a Hypervisor Call is generated by executing an HVC instruction.
A Hypervisor Call exception is taken to the following Exception levels:

- If Secure EL2 is implemented and enabled in the current Security state, when taken from EL1, the exception is taken to EL2.
- When taken from EL2, the exception is taken to EL2.
- When taken from EL3, the exception is taken to EL3.

If any of the following is true, the HVC instruction is UNDEFINED:

- The PE is executing at EL0.
- If EL2 is not enabled in the current Security state, and the PE is executing at EL1.
- SCR_EL3.HCE is 0.

If EL3 is implemented, a Secure Monitor Call is generated by executing an SMC instruction. A Secure Monitor Call is a synchronous exception that is taken to EL3.

If any of the following are true, the SMC instruction is UNDEFINED:

- The PE is executing at EL0.
- EL3 is not implemented.

### D1.3.10 Program counter and stack pointer alignment

#### PC alignment checking

If bits [1:0] of the PC are not 0b00, there is a misaligned PC.

The execution of an instruction with a misaligned PC generates a synchronous PC Alignment exception on that instruction.

A PC Alignment exception is taken to the following Exception levels:

- If executing at EL0:
  - If HCR_EL2.TGE is 0, the exception is taken to EL1.
  - If HCR_EL2.TGE is 1, the exception is taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.
- If executing at EL3, the exception is taken to EL3.

When a PC Alignment Fault exception is taken to an Exception level, ELx, using AArch64, the ELR_ELx and the FAR_ELx both hold the entire PC in its misaligned form.

A misalignment of the PC is an indication of a serious error, for example software corruption of an address.

#### SP alignment checking

When the SP is used as the base address of a calculation, regardless of any offset applied by the instruction, if bits [3:0] of the SP are not 0b0000, there is a misaligned SP.

If SP alignment checking is enabled, then the execution of a load or store using the SP with a misaligned SP generates a synchronous SP Alignment exception on that load or store.

PRFM instructions that use the SP do not perform stack alignment checking.

The following bits enable SP alignment checking at each Exception level when that Exception level is using AArch64.

- If HCR_EL2.{E2H, TGE} is \{1, 1\}, SCTLR_EL2.SA0 controls EL0. Otherwise, SCTLR_EL1.SA0 controls EL0.
- SCTLR_EL1.SA controls EL1.
• SCTLR_EL2.SA controls EL2.
• SCTLR_EL3.SA controls EL3.

**RNCGYQ**

An SP Alignment exception is taken to the following Exception levels:

- If executing at EL0:
  - If HCR_EL2.TGE is 0, the exception is taken to EL1
  - If HCR_EL2.TGE is 1, the exception is taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.
- If executing at EL3, the exception is taken to EL3.

**RDLVL**

If SP alignment checking is enabled and an SVE predicated load or store instruction with any *Active elements* uses the current SP as the base address, then the SP register is checked for 16-byte alignment.

**RFNCJX**

If SP alignment checking is enabled and an SVE predicated load or store instruction with no *Active elements* uses the current SP as the base address, then it is CONstrained UNPredictable whether the SP register is checked for 16-byte alignment.
D1.4 Process state, PSTATE

PSTATE is an abstraction of process state information. All Arm instruction sets provide instructions that operate on elements of PSTATE. PSTATE includes all of the following:

- Fields that are meaningful only in AArch32 state.
- Fields that are meaningful only in AArch64 state.
- Fields that are meaningful in both Execution states.

PSTATE is defined in pseudocode as the PSTATE structure, of type ProcState.

D1.4.1 PSTATE fields that are meaningful in AArch64 state

PSTATE fields that are meaningful in AArch64 state are grouped into the following categories:

- Condition flags.
- Execution state controls.
- Exception mask bits.
- Access control bits.
- Timing control bits.
- Speculation control bits.

The following PSTATE bits are meaningful in AArch64 state:

<table>
<thead>
<tr>
<th>PSTATE field name</th>
<th>PSTATE field group</th>
<th>Required implemented feature</th>
<th>Value when a Warm reset is asserted</th>
<th>Additional details</th>
</tr>
</thead>
<tbody>
<tr>
<td>N, Negative condition flag</td>
<td>Condition flags</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Z, Zero condition flag</td>
<td>Condition flags</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C, Carry condition flag</td>
<td>Condition flags</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>V, Overflow condition flag</td>
<td>Condition flags</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SS, Software Step bit</td>
<td>Execution state controls</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>IL, Illegal Execution state bit</td>
<td>Execution state controls</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>nRW, Current Execution state bit</td>
<td>Execution state controls</td>
<td>-</td>
<td>0</td>
<td>If the current execution state is AArch64, the PSTATE.nRW bit is 0.</td>
</tr>
<tr>
<td>EL, Current Exception level field</td>
<td>Execution state controls</td>
<td>-</td>
<td>When a Warm reset is asserted into an Exception level using AArch64, the PSTATE.EL field holds the encoding for the highest implemented Exception level.</td>
<td>-</td>
</tr>
<tr>
<td>SP, Stack pointer register selection bit</td>
<td>Execution state controls</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>
Accessing PSTATE fields

In AArch64 state, PSTATE fields are accessed using Special-purpose registers that are directly read using the `MRS` (register) instructions, and directly written using the `MSR` (register) instructions.

If in AArch64 state, the following special-purpose registers can access the PSTATE fields that hold AArch64 state:

<table>
<thead>
<tr>
<th>Special-purpose register</th>
<th>PSTATE fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>NZCV</td>
<td>N, Z, C, V</td>
</tr>
<tr>
<td>DAIF</td>
<td>D, A, I, F</td>
</tr>
<tr>
<td>CurrentEL</td>
<td>EL</td>
</tr>
<tr>
<td>SPSel</td>
<td>SP</td>
</tr>
<tr>
<td>PAN</td>
<td>PAN</td>
</tr>
<tr>
<td>UAO</td>
<td>UAO</td>
</tr>
<tr>
<td>DIT</td>
<td>DIT</td>
</tr>
</tbody>
</table>
All other PSTATE fields do not have direct read and write access.

Software can use the MSR (immediate) instructions to directly write to PSTATE.\{D, A, I, F, SP, PAN, UAO, SSBS, TCO\}.

The following PSTATE fields can be accessed at EL0:
- PSTATE.\{N, Z, C, V, SSBS, DIT, TCO\}.
- If SCTLR_EL1.UMA is 0, PSTATE.\{D, A, I, F\}.
- All other PSTATE access instructions can be executed at EL1 or higher and are UNDEFINED at EL0.

Writes to the PSTATE fields have side-effects on various aspects of the PE operation. For side-effects caused by writes to a PSTATE field, all of the following are true:
- The side-effect is guaranteed not to be visible to earlier instructions in the Execution stream.
- The side-effect is guaranteed to be visible to later instructions in the Execution stream.

Other side-effects might occur but are not guaranteed.

<table>
<thead>
<tr>
<th>Special-purpose register</th>
<th>PSTATE fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSBS</td>
<td>SSBS</td>
</tr>
<tr>
<td>TCO</td>
<td>TCO</td>
</tr>
<tr>
<td>ALLINT</td>
<td>ALLINT</td>
</tr>
</tbody>
</table>
D1.5

Resets and power domains

IDQXXZ

The PE logic is split into a Debug power domain and a Core power domain. Cold and Warm resets, reset elements in the Core power domain. An External debug reset can reset the Debug power domain. Other resets might be implemented.

D1.5.1

Power domains and reset domains

RJMTZY

The PE logic is split into two logical power domains:

• The Core power domain.
• The Debug power domain.

IQBPTS

The power domains are described as logical because the architecture does not require two physical power domains.

RPDMKB

It is IMPLEMENTATION DEFINED whether the Core power domain and Debug power domain power up and power down separately or together.

RZDVQQ

The Core power domain contains:

• Non-debug registers and logic.
• Self-hosted debug registers and logic.
• Shared debug registers and logic.
• Some external debug registers and logic.

RGTVLN

The Debug power domain contains:

• The interface between the PE and the external debugger.
• Some external debug registers.

RMLXWM

If an external debugger is connected to the PE, the Debug power domain is powered up.

IKTSWK

For more information on access permissions for the external debug interface registers, see Access permissions for the External debug interface registers on page H8-10370 or individual register descriptions.

D1.5.2

Reset types

IHDLQB

The architecture defines the following resets:

• Warm reset.
• Cold reset.
• External debug reset.
• Trace unit reset. See Resetting the trace unit on page D4-4800.

IFMTCY

Other resets are IMPLEMENTATION DEFINED and can be mapped onto the existing resets.

RPXXYQ

Mechanisms to assert resets, other than RMR_ELx, are IMPLEMENTATION DEFINED.

RHLGGD

When a Cold reset is asserted, for each register in the Core power domain, the reset sets the register to its architecturally defined reset value.

RPJXVL

When a Warm reset is asserted, the logic on which the PE executes is reset. The Warm reset does not reset the integrated debug functionality.

RNBBKHK

When an External debug reset is asserted, the Debug power domain is reset. The Cold reset domain is not reset.

IMWTKX

If RMR_ELx is implemented, writing 1 to RMR_ELx.RR requests a Warm reset.

RDTTVZ

It is IMPLEMENTATION DEFINED whether a Warm reset can be asserted without using RMR_ELx.

ILWNC

Arm recommends that a Warm reset can be asserted independently of a Cold reset.

IQLZGF

A Warm reset allows debugging across a reset of the PE logic in the Core power domain.
It is IMPLEMENTATION DEFINED whether an External debug reset and a Cold reset can be asserted independently.

Arm recommends that when separate Core and Debug power domains are implemented, the External debug reset and Cold reset can be asserted independently.

**D1.5.3 Reset behavior**

Immediately after a reset, much of the PE state is architecturally UNKNOWN. However, some of the PE state is defined, see the individual register descriptions for more information. The state that is reset to known values is sufficient to permit predictable initial execution at the highest Exception level, such that this execution is then capable of initializing the remaining state of the system where necessary before use.

When a Cold or Warm reset is asserted, all of the following occur:

- The PE enters the highest implemented Exception level.
- The stack pointer for the highest implemented Exception level, SP_ELx, is selected.

When a Cold or Warm reset is deasserted, execution starts at an IMPLEMENTATION DEFINED address, anywhere in the physical address range. The RVBAR associated with the highest implemented Exception level RVBAR_EL1, RVBAR_EL2, or RVBAR_EL3, holds the address at which the PE starts execution.

When a Cold or Warm reset is deasserted, the IMPLEMENTATION DEFINED address from which the PE starts execution is typically set either by a hardwired configuration of the PE or by configuration input signals.

When a Cold or Warm reset is asserted, all of the following are UNKNOWN:

- All general-purpose, SIMD, and floating-point registers.
- The global exclusive monitor for the PE.
- The local exclusive monitor for the PE.
- System and Special-purpose registers with a reset value of UNKNOWN described in the AArch64 Registers.

When a Cold or Warm reset is asserted, PSTATE.{SS, IL, nRW, EL, SP, D, A, I, F, DIT} fields are set to known values.

When a Cold or Warm reset is asserted, the TLBs and caches are in an IMPLEMENTATION DEFINED state.

When a Cold or Warm reset has been asserted and before the memory management system is enabled, the TLBs, caches or both might need to be invalidated by IMPLEMENTATION DEFINED invalidation sequences.

When a Cold or Warm reset has been asserted and before Normal memory accesses are permitted to be Cacheable, the TLBs, caches or both might need to be invalidated by IMPLEMENTATION DEFINED invalidation sequences.

If IMPLEMENTATION DEFINED resets are implemented, each IMPLEMENTATION DEFINED reset can treat the cache and TLB state differently.

If IMPLEMENTATION DEFINED resets are implemented, for each IMPLEMENTATION DEFINED reset, the TLBs, caches, or both might require a different IMPLEMENTATION DEFINED invalidation sequence.

The IMPLEMENTATION DEFINED invalidation sequence might be a NOP.

**External debug access to registers in reset**

If External Debug reset is asserted, for each register that can be accessed by the external debug interface, a register access by the external debug interface has one of the following IMPLEMENTATION DEFINED results:

- The external debug interface can access the register, and the access result is IMPLEMENTATION DEFINED.
- The external debug interface cannot access the register.

If either a Cold or Warm reset is asserted, accesses of registers or register fields might be direct accesses or indirect side-effects of an access.

If either a Cold or Warm reset is asserted, external debug interface accesses of a register or register field have the following effects:
<table>
<thead>
<tr>
<th>Access</th>
<th>Register or register field reset behavior</th>
<th>Effect on register or register field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>Reset by the reset signal</td>
<td>Set to CONSTRAINED UNPREDICTABLE choice of the reset value or the value written.</td>
</tr>
<tr>
<td>Write</td>
<td>Not reset by the reset signal</td>
<td>Set to the value written.</td>
</tr>
<tr>
<td>Read</td>
<td>Reset by the reset signal</td>
<td>Returns an UNKNOWN value.</td>
</tr>
<tr>
<td>Read</td>
<td>Not reset by the reset signal</td>
<td>Returns the value of the register or register field.</td>
</tr>
</tbody>
</table>
D1.6    Mechanisms for entering a low-power state

The architecture provides the following mechanisms that software can use to indicate that the PE can enter low-power state:

- Wait for Event.
- Wait for Interrupt.

D1.6.1   Wait for Event

The Wait for Event mechanism behavior depends on the interaction of all of the following:

- The Event Register for the PE. See The Event Register.
- The Wait for Event (WFE) or Wait for Event with Timeout (WFET) instruction. See The Wait for Event and Wait for Event with Timeout instructions on page D1-4677.
- WFE wake-up events. See WFE wake-up events in AArch64 state on page D1-4677.
- The Send Event instructions, SEV and SEVL, that can cause WFE wake-up events. See The Send Event instructions on page D1-4678.

Except for all or the following, the architecture does not define the exact nature of the low-power state:

- When a WFE or WFET instruction is executed, the architecture requires that memory coherency is not lost.
- If the system is configured such that the WFE or WFET instruction can be completed, then the architecture requires that the architectural state is not lost.

The Wait for Event mechanism is associated with suspending execution on a PE for power saving, therefore Arm recommends that the Event Register is set infrequently. Software must only use the setting of the Event Register as a hint, and must not assume that any particular message is sent as a result of the setting of the Event Register.

The Event Register

The Event Register is a single bit register for each PE.

The event register is a conceptual register and cannot be explicitly accessed.

If the Event Register is set, then an event has occurred since the register was last cleared and the event might require some action by the PE.

When the PE executes a WFE or WFET instruction, all of the following apply:

- If the Event Register is clear, then the PE can enter the low-power state.
- If the Event Register is set, then all of the following apply:
  — The PE does not suspend operation.
  — The Event Register is cleared.
  — The WFE or WFET instruction completes immediately.

The reset value of the Event Register is UNKNOWN.

The Event Register for a PE is set by any of the following:

- A Send Event instruction, SEV, executed by any PE in the system.
- A Send Event Local instruction, SEVL, executed by the PE.
- An exception return.
- The clearing of the global monitor for the PE.
- An event from a Generic Timer event stream, see Event streams on page D10-5239.
- An event sent by some IMPLEMENTATION DEFINED mechanism.

The Event Register is cleared only by a Wait for Event (WFE) instruction or a Wait for Event with Timeout (WFET) instruction.
The AArch64 System Level Programmers’ Model

D1.6 Mechanisms for entering a low-power state

IZXJPD Software cannot read or write the value of the Event Register directly.

The Wait for Event and Wait for Event with Timeout instructions

RQRXWZ If FEAT_WFXT is implemented, the Wait for Event with Timeout (WFET) instruction is implemented

RCSKPF If the Event Register is clear, when a WFE or WFET instruction is executed, the PE can suspend execution and enter a low-power state.

RGZZQM If the PE enters a low-power state, the PE remains in the lower-power state until one of the following occurs:
- The PE detects a WFE wake-up event.
- An IMPLEMENTATION DEFINED wake event that is architecturally permitted to occur at any time.
- A reset is asserted.

IQGHTM The architecture permits all of the following:
- The PE is permitted to leave the low-power state for any reason.
- The PE is permitted to treat WFE as a NOP, but this is not recommended for lowest power operation.

RSLQVQ If the PE is in a low-power state, when the PE detects a WFE wake-up event, or earlier if the implementation chooses, the WFE or WFET instruction completes. If the wake-up event sets the Event Register, when execution is restarted, the state of the Event Register is IMPLEMENTATION DEFINED.

IDRSIV Software using the Wait for Event mechanism must tolerate spurious wake-up events, including multiple wake-up events.

Trapping of WFE and WFET

IKMHTB The WFE and WFET instructions are available at all Exception levels. Attempts to enter a low-power state made by software executing at EL0, EL1, or EL2 can be configured to trap to a higher Exception level.

RJWKTL If FEAT_TWED is implemented, then all of the following apply:
- A delay before taking a WFE trap can be configured.
- If a delay before taking a WFE trap is configured, then the delay does not affect the priority of the traps.

IRDYQK For example, if execution is subject to a trap at EL1 because SCTLR_EL1.nTWE is 0 and HCR_EL2.TWE is 1, the only trap that will be taken is a trap to EL1, even if the delay at EL1 is longer than the delay at EL2.

WFE wake-up events in AArch64 state

IRJRLR In this section, AllIntMask refers to the value described in RXZPDT.

ILRPVP If a WFE or WFET instruction puts a PE into low-power state, a WFE wake-up event received by the PE causes that PE to exit low-power state.

IRJRLR If a WFE or WFET instruction puts a PE into low-power state, a WFE wake-up event received by the PE causes that PE to exit low-power state.

IRLPVP If a WFE or WFET instruction puts a PE into low-power state, a WFE wake-up event received by the PE causes that PE to exit low-power state.

RKBMFC All of the following are WFE wake-up events:
- The execution of an SEV instruction on any PE in a multiprocessor system.
- Any physical SError interrupt, IRQ interrupt, or FIQ interrupt received by the PE that is not disabled by EDSR.INTdis and either of the following is true:
  - The interrupt is marked as A in RXSLWJ in Asynchronous exception masking on page D1-4659, regardless of the value of the corresponding PSTATE.{A, I, F} mask bit.
  - The interrupt is marked as B in RXSLWJ in Asynchronous exception masking on page D1-4659, AllIntMask is 0, and any of the following are true:
    - The value of the corresponding PSTATE.{A, I, F} mask bit is 0.
    - An IRQ or FIQ interrupt has Superpriority.
See RMHWBP and RGFXKY.
• If all of the following apply, any virtual SError interrupt, IRQ interrupt, or FIQ interrupt received by the PE:
  — The PE is executing at EL1 or EL0.
  — The interrupt is not disabled by EDSCR.INTdis.
  — The interrupt is marked as B in RBKHXL in Virtual interrupt masking on page D1-4662, AllIntMask is 0, and any of the following are true:
    — The value of the corresponding PSTATE.{A, I, F} mask bit is 0.
    — A vIRQ or vFIQ interrupt has Superpriority.
    See RSNLJH and RXSPSG.
• If halting is allowed, an asynchronous External Debug Request debug event. For the definition of halting, see Halting allowed and halting prohibited on page H2-10233 and External Debug Request debug event on page H3-10291.
• An event sent by the timer event stream for the PE. See Event streams on page D10-5239.
• An event caused by the clearing of the global monitor for the PE.
• An event sent by some IMPLEMENTATION DEFINED mechanism.
• When FEAT_WFxT is implemented, for WFET instructions, a local timeout event caused by the virtual count threshold value, expressed in CNTVCT_EL0, being equaled or exceeded.

Not all of the wake-up events in RKBMF set the Event Register.

The Send Event instructions

The Send Event instructions are:

- SEV, Send Event: Causes an event to be signaled to all PEs in a multiprocessor system.
- SEVL, Send Event Local: Causes an event to be signaled locally without requiring the event to be signaled to other PEs in a multiprocessor system.

An SEVL instruction might signal an event to other PEs by an IMPLEMENTATION DEFINED mechanism, but it is not required to do so.

The mechanism that signals an event to other PEs is IMPLEMENTATION DEFINED.

When an event is signaled by an SEV instruction, the ordering of the event with respect to the completion of memory accesses by instructions before the SEV instruction is not guaranteed.

Arm recommends that software includes a Data Synchronization Barrier (DSB) instruction before any SEV instruction. DSB instruction ensures that no instructions, including any SEV instructions, that appear in program order after the DSB instruction, can execute until the DSB instruction has completed. See Data Synchronization Barrier (DSB) on page B2-178.

The SEVL instruction appears to execute in program order relative to any subsequent WFE or WFET instruction executed on the same PE.

The behavior in RQJDQV applies without the need for any explicit insertion of barrier instructions.

D1.6.2 Wait for Interrupt mechanism

A PE can use the Wait for Interrupt mechanism to enter a low-power state.

When FEAT_WFxT is implemented, the Wait for Interrupt with Timeout (WFIT) instruction is implemented.

Software can use the Wait for Interrupt (WFI) and Wait for Interrupt with Timeout (WFIT) instructions to cause the PE to enter a low-power state.

If a Wait for Interrupt (WFI) and Wait for Interrupt with Timeout (WFIT) instructions causes the PE to enter a low-power state, the PE remains in that low-power state until any of the following occur:

- The PE detects a WFI wake-up event.
The AArch64 System Level Programmers' Model
D1.6 Mechanisms for entering a low-power state

- An IMPLEMENTATION DEFINED wake event that is architecturally permitted to occur at any time.
- A reset is asserted.

The architecture permits all of the following:
- The PE is permitted to leave the low-power state for any reason.
- The PE is permitted to treat WFI as a NOP, but this is not recommended for lowest power operation.

When the PE leaves a low-power state that was entered because of a WFI or WFIT instruction, the WFI or WFIT instruction completes.

Except for all or the following, the architecture does not define the exact nature of the low-power state:
- When a WFI or WFIT instruction is executed, the architecture requires that memory coherency is not lost.
- If the system is configured such that the WFI or WFIT instruction can be completed, then the architecture requires that the architectural state is not lost.

In some implementations, based on the configuration of system specific registers, WFI can be used as part of a powerdown sequence where no interrupts will cause WFI wake-up events, and restoration of power involves resetting of the PE. In those cases, the WFI is permitted to cause a loss of architectural state, as it is assumed that this state will have been saved by software as part of the powerdown sequence before the WFI.

In some implementations, based on the configuration of system specific registers, WFI can be used as part of a powerdown sequence where no interrupts will cause WFI wake-up events, and restoration of power involves resetting of the PE. In those cases, the WFI is permitted to cause a loss of architectural state, as it is assumed that this state will have been saved by software as part of the powerdown sequence before the WFI.

The WFI and WFIT instructions are available at all Exception levels. Attempts to enter a low-power state made by software executing at EL0, EL1, or EL2 can be configured to trap to a higher Exception level.

### WFI wake-up events

In this section, AllIntMask refers to the value described in RXZPDT.

If a WFI or WFIT instruction put a PE into low-power state, the PE remains in that low-power state until it receives a WFE wake-up event.

All of the following are WFI wake-up events:
- Any physical SError interrupt, IRQ interrupt, or FIQ interrupt received by the PE that is marked as A, B or A/B in the tables in Physical interrupt masking on page D1-4660, regardless of the value of the corresponding PSTATE. {A, I, F} mask bit.
- If the PE is executing at in EL0 or EL, any virtual SError interrupt, IRQ interrupt, or FIQ interrupt received by the PE, that is marked as B in the tables in Virtual interrupt masking on page D1-4662, regardless of the value of the corresponding PSTATE. {A, I, F} mask bit.
- If halting is allowed, an asynchronous External Debug Request debug event. For the definition of halting, see Halting allowed and halting prohibited on page H2-10233. See also External Debug Request debug event on page H3-10291.
- An event sent by an IMPLEMENTATION DEFINED mechanism.
- When FEAT_WFxT is implemented, for WFIT instructions, a local timeout event caused by the virtual count threshold value, expressed in CNTVCT_EL0, being equaled or exceeded.

WFI wake-up events are never disabled by EDSCR.INTdis, and are never masked by the PSTATE. {A, I, F} mask bits, or by the state of AllIntMask. If wake-up is invoked by an interrupt that is disabled or masked, the interrupt is not taken.

Because debug events are WFI wake-up events, Arm recommends that Wait for Interrupt is used as part of an idle loop rather than waiting for a single specific interrupt event to occur and then moving forward. This ensures that the intervention of debug while waiting does not significantly change the function of the program being debugged.

The architecture does not require the WFI mechanism to drain down any pending memory activity before suspending execution, and software must not rely on the WFI mechanism operating in this way.
D1.6.3 Using WFI to indicate an idle state on bus interfaces

When all of the following are true, the IMPLEMENTATION DEFINED mechanism for entering powerdown state can be completed:

- The IMPLEMENTATION DEFINED mechanism forces quiescence on a PE.
- The IMPLEMENTATION DEFINED mechanism prevents any possible WFI wakeup events.

The mechanism for entering powerdown state is IMPLEMENTATION DEFINED.

Although the mechanism for entering powerdown state is IMPLEMENTATION DEFINED, the WFI and WFE mechanisms are often used to enter powerdown state.

If a WFI powerdown mechanism is implemented, when a WFI or WFI! instructions is executed, the PE completes all current operations and any associated bus activity also completes. When all current PE operations and associated bus activity are complete, the PE can signal to an external power controller that there is no ongoing bus activity.

After a WFI instruction is executed and all currently executing instructions and related bus activity has completed, the PE is waiting for an interrupt and is in idle state. If in idle state, the PE can process memory-mapped or external debug interface accesses to debug registers.

The indication of idle state to the system normally only applies to the non-debug functional interfaces used by the PE, not the debug interfaces.

If OSDLR_EL1.DLK is 1, the PE does not signal the idle state to the control logic unless it can also guarantee that the debug interface is idle. For more information about the OS Double Lock, see Debug behavior when the OS Double Lock is locked on page H6-10346.

In a PE that implements separate Core and Debug power domains, the debug interface referred to in this section is the interface between the Core and Debug power domains, since the signal to the power controller indicates that the Core power domain is idle. For more information about the power domains, see Power domains and debug on page H6-10335.

The exact nature of the debug interface is IMPLEMENTATION DEFINED, but the use of Wait for Interrupt as the only architecturally-defined mechanism that completely suspends execution makes it suitable as the preferred powerdown entry mechanism.
D1.7 Self-hosted debug

The architecture supports both of the following:

Self-hosted debug
The PE itself hosts a debugger. The debugger programs the PE to generate debug exceptions. Debug exceptions are accommodated in the Arm Exception model.

External debug
The PE is controlled by an external debugger. The debugger programs the PE to generate debug events, that cause the PE to enter Debug state. In Debug state, the PE is halted.

D1.7.1 Debug exceptions

If a debugger has programmed the PE to generate Debug exceptions, the Debug exceptions occur during normal program flow.

For example, a software developer might use a debugger contained in an operating system to debug an application. To do this, the debugger might enable one or more debug exceptions.

The debug exceptions are all of the following:

- Breakpoint Instruction exceptions.
- Breakpoint exceptions.
- Watchpoint exceptions.
- Vector Catch exceptions.
- Software Step exceptions.

The PE can only generate a particular debug exception if all of the following are true:

- Debug exceptions are enabled from the current Exception level and Security state. See Enabling debug exceptions from the current Exception level on page D2-4699. Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.
- A debugger has enabled that particular debug exception. All of the debug exceptions except for Breakpoint Instruction exceptions have an enable control contained in the MDSCR_EL1. See The debug exception enable controls on page D2-4696.

Breakpoints and watchpoints can cause entry to Debug state instead of causing debug exceptions. See Chapter H1 About External Debug.

D1.7.2 The PSTATE debug mask bit, D

The debug exception mask bit, PSTATE.D, can mask Watchpoint, Breakpoint, and Software Step exceptions if the target Exception level is the current Exception level.

If in AArch64 state and the target Exception level of a debug exception is lower than the current Exception level, the debug exception is masked.

If the target Exception level is higher than the current Exception level, debug exceptions are not masked by PSTATE.D.

If PSTATE.D is 1 and the PE is already handling an exception in AArch64 state, the Watchpoint, Breakpoint, and Software Step exceptions are masked.

If an exception is already being handled, masking prevents software from generating another instance of an exception and prevents recursive entry into the exception handler and loss of return state.

Masking debug interrupts with PSTATE.D prevents the generation of new debug exceptions, therefore, any masked debug exceptions are not taken if PSTATE.D is later set to 0.
<table>
<thead>
<tr>
<th>IVSTYD</th>
<th>The behavior described in RNCHXS differs from the behavior for interrupts, where the PSTATE.{A, I, F} mask prevents the interrupt from being taken, but instead the interrupt remains pending.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IKWFMZ</td>
<td>When an exception is taken, PSTATE.D is set to 1.</td>
</tr>
</tbody>
</table>
D1.8 Event monitors

The architecture supports the following non-invasive architectural components that allow for event monitoring:

Performance Monitors
The Performance Monitors have a wide feature set, flexible selection of counted events, and are read/write in operation.

Activity Monitors
The Activity Monitors have a narrow feature set, limited selection of counted events, and are read-only in operation.

D1.8.1 The Performance Monitors Extension

The System registers provide access to a Performance Monitoring Unit (PMU), defined as the OPTIONAL Performance Monitors Extension to the architecture, a non-invasive debug resource that provides information about the operation of the PE. The PMU provides:

- A 64-bit cycle counter.
- An IMPLEMENTATION DEFINED number of event counters. If FEAT_PMUv3p5 is implemented, the event counters are 64-bit unsigned counters, otherwise the event counters are 32-bit event counters. Each event counter can be configured to count occurrences of a specified event. The events that can be counted are:
  
  - Architectural and microarchitectural events that are likely to be consistent across many microarchitectures. The PMU architecture uses event numbers to identify an event, and the PMU specification defines which event number must be used for each of these architectural and microarchitectural events.
  
  - Implementation-specific events. The PMU specification reserves event numbers for implementation-specific events.

Chapter D11 The Performance Monitors Extension.

D1.8.2 The Activity Monitors Extension

If the OPTIONAL Activity Monitors Extension is implemented, the System registers provide access to controls and counters for the Activity Monitors Unit (AMU).

Chapter D12 The Activity Monitors Extension.
Interprocessing

Moving between the AArch64 and AArch32 Execution states is called interprocessing.

The Execution state can only change on a change of Exception level.

Therefore, Execution state can change only on taking an exception to a higher Exception level or returning from an exception to a lower Exception level. For an exception that is taken to the same Exception level or is returning from the same Exception level, the Execution state cannot change.

When taking an exception to a higher Exception level, the Execution state is one of the following:

- Unchanged.
- Changed from AArch32 state to AArch64 state.

When returning from an exception to a lower Exception level, the Execution state is one of the following:

- Unchanged.
- Changed from AArch64 state to AArch32 state.

Register mappings between AArch32 state and AArch64 state

Register mappings between AArch32 state and AArch64 state describe the following:

- For exceptions taken from AArch32 state to AArch64 state, where the AArch32 register content is found.
- For exception returns from AArch64 state to AArch32 state, how the AArch32 register content is derived.

The AArch32 state register contents occupy the bottom 32 bits of the AArch64 state registers.

If in AArch32 state, the upper 32 bits of AArch64 state registers are inaccessible and are ignored.

For bits[63:32] of AArch64 registers that are not mapped to AArch32 registers, the unmapped bits are unchanged by AArch32 state execution.

Mapping of the general-purpose registers between the Execution states

The following table shows the mappings of general-purpose registers between Execution states:

<table>
<thead>
<tr>
<th>AArch32 register</th>
<th>AArch64 register</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>X0</td>
</tr>
<tr>
<td>R1</td>
<td>X1</td>
</tr>
<tr>
<td>R2</td>
<td>X2</td>
</tr>
<tr>
<td>R3</td>
<td>X3</td>
</tr>
<tr>
<td>R4</td>
<td>X4</td>
</tr>
<tr>
<td>R5</td>
<td>X5</td>
</tr>
<tr>
<td>R6</td>
<td>X6</td>
</tr>
<tr>
<td>R7</td>
<td>X7</td>
</tr>
<tr>
<td>R8_usr</td>
<td>X8</td>
</tr>
<tr>
<td>R9_usr</td>
<td>X9</td>
</tr>
<tr>
<td>R10_usr</td>
<td>X10</td>
</tr>
<tr>
<td>R11_usr</td>
<td>X11</td>
</tr>
</tbody>
</table>
For some exceptions, the exception syndrome given in the ESR_ELx identifies one or more register numbers from the issued instruction that generated the exception. If one of these exceptions is taken from an Exception level using AArch32, the register numbers give the AArch64 view of the register.

For example, if an exception is taken from AArch32 Abort mode and the faulting instruction specified R14, the ESR_ELx.ISS field would report this using the EC value 0b10100, because register X20 provides the AArch64 view of LR_abt, which is the copy of R14 used in Abort mode.

### Mapping of the SIMD and floating-point registers between the Execution states

The following table shows the mappings of SIMD and floating-point registers between the Execution states:

<table>
<thead>
<tr>
<th>AArch64 register</th>
<th>AArch32 register</th>
</tr>
</thead>
<tbody>
<tr>
<td>V0</td>
<td>Q0</td>
</tr>
<tr>
<td>V1</td>
<td>Q1</td>
</tr>
</tbody>
</table>
The AArch64 state registers V16-V31 are not accessible from AArch32 state.

The mapping between the V, D, and S registers in AArch64 state is not the same as the mapping between the Q, D, and S registers in AArch32 state.

In AArch64 state, there are:
- 32 64-bit D registers, D0-D31.
- 32 32-bit S registers, S0-S31.

A smaller register occupies the least significant bytes of the corresponding larger register.

The following graphic shows the mapping of V, D, and S registers in AArch64 state:

In AArch32 state, there are:
- 16 128-bit Q registers, Q0-Q15.
- 32 64-bit D registers, D0-D31.
- 32 32-bit S registers, S0-S31.

Smaller registers are packed into larger registers.

The following graphic shows the mapping of Q, D and S registers in AArch32 state:

In AArch32 state:
- There are no S registers that correspond to Q8-Q15.
- D16-D31 pack into Q8-Q15.

If software executing in AArch64 state interprets D or S registers from AArch32 state, the software must unpack the D or S registers from the V registers before it uses them.

**Mapping of the System registers between the Execution states**

For a full list of mappings of writable AArch64 System registers to the AArch32 System registers, see *Mapping of writable AArch64 System registers to the AArch32 System registers* on page K17-11840.
The relationship between the AArch64 System registers and the AArch32 System registers is architecturally defined.

Modifications made to AArch32 System registers affect only the parts of the AArch64 state registers that are mapped to the AArch32 System registers.

Supervisory code, such as a hypervisor, executing in AArch64 state, can save, restore, and interpret the System registers belonging to a lower Exception level that is using AArch32.

In some cases, two AArch32 System registers are packed into a single AArch64 System register.

If EL3 is implemented and is using AArch32, some System registers are banked between Secure and Non-secure states. In this type of banking, there is an instance of the register in Secure state and another instance of the register in Non-secure state.

If any of the following are true, banking between Secure and Non-secure states is not supported:

- EL3 is not implemented.
- EL3 is implemented and is using AArch64.

If EL3 is implemented and is using AArch64, the same, non-banked, registers are accessed in the following state:

- Secure EL1 with EL1 using AArch32.
- Non-secure EL1 with EL1 using AArch32.

If EL3 is implemented and is using AArch64, it is not possible to architecturally determine whether an AArch64 register is mapped onto the Secure or the Non-secure instance of corresponding AArch32 register.

If EL3 is implemented and is using AArch64, the interrupt asserted by the AArch64 CNTP_* timer is the same interrupt that is asserted by the Non-secure AArch32 CNTP_* timer when EL3 is using AArch32.

Although not required, Arm expects that implementations map many of the AArch64 registers for use by EL3 to the Secure instances of the banked AArch32 registers, and map many of the AArch64 registers for use by EL1 to the Non-secure instances of the banked AArch32 registers. However, if EL2 and EL3 are implemented and both support use of AArch32, this mapping is not possible for the following registers:

- IFAR - This is because when EL3 is using AArch32, HIFAR is an alias of the Secure IFAR.
- DFAR - This is because when EL3 is using AArch32, HDFAR is an alias of the Secure DFAR.

There are some AArch32 System registers that are only used in AArch32 state and do not have an equivalent AArch64 System register. However, there are AArch64 registers that can access, from a higher Exception level, the AArch32 registers that do not have an AArch64 state equivalent.

For a full list of AArch64 registers that access AArch32 System registers with no AArch64 equivalent, see AArch64 registers for accessing registers that are only used in AArch32 state on page K17-11845.

For a full list AArch64 registers that allow access to the AArch32 ID registers from AArch64 state, see AArch64 registers that access the AArch32 ID registers on page K17-11845.

State of the general-purpose registers on taking an exception to AArch64 state

If the general-purpose register was accessible from AArch32 state

When an exception is taken from AArch32 state to AArch64 state and the general-purpose register was accessible from AArch32 state, the upper 32 bits have one of the following IMPLEMENTATION DEFINED values and might vary dynamically within an implementation:

- The upper 32 bits retain the value that the same architectural register held before any AArch32 execution.
- The upper 32 bits are set to zero.

The IMPLEMENTATION DEFINED behavior applies regardless of whether any execution occurred at the Exception level that was using AArch32.
For example, the IMPLEMENTATION DEFINED behavior described in RWWJWP applies even if AArch32 state was entered by an exception return from AArch64 state, and another exception was immediately taken to AArch64 state without any instruction execution in AArch32 state.

When an exception is taken from AArch32 state to AArch64 state, software must regard the value of the upper 32 bits as a CONSTRAINED UNPREDICTABLE choice between the two values described in RWWJWP.

**If the general-purpose register was not accessible from AArch32 state**

If all of the following apply, when an exception is taken from AArch32 state to AArch64 state, the X15 register is treated as if it is accessible and therefore the upper 32 bits of X15 might either be set to zero or retain their previous value:

- The target Exception level is EL3.
- EL2 is not implemented or EL1 is in Secure state.
- SCR_EL3.RW is 0.

Otherwise, when an exception is taken from AArch32 state to AArch64 state, for a general-purpose register that was not accessible from AArch32 state, the register retains the value it had before any AArch32 execution.

**Determining the upper 32 bits of AArch64 registers on taking an exception from AArch32 state**

Whether a general-purpose register has its upper 32 bits set to zero or retained on taking an exception from AArch32 to AArch64 is affected by all of the following:

- The AArch64 state target Exception level.
- The values of both:
  - SCR_EL3.RW.
  - HCR_EL2.RW or HCR.RW, where HCR.RW is a notional bit that is RES 0.

The following table shows which general-purpose registers can have their upper 32 bits set to zero on taking an exception from AArch32 state to AArch64 state. In this table, a dash (-) indicates that the RW values are not valid for the targeted Exception level.

<table>
<thead>
<tr>
<th>SCR_EL3.RW</th>
<th>HCR_EL2.RW or HCR.RW</th>
<th>Registers when the target Exception level is:</th>
<th>EL3</th>
<th>EL2</th>
<th>EL1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X0-X30</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Not valid because the RW bit values would imply that EL2 is AArch32 and EL1 is AArch64.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X0-X14, X16-X30</td>
<td>X0-X14, X16-X30</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X0-X14</td>
<td>X0-X14</td>
<td>X0-X14</td>
<td>-</td>
</tr>
</tbody>
</table>

If EL2 is not implemented, or the SCR_EL3.NS or SCR.NS bits prevent its use, the registers that can have their upper 32 bits set to zero on taking an exception from AArch32 state to AArch64 state are the same as when HCR_EL2.RW has the same value as SCR_EL3.RW.
The following table shows which general-purpose registers can retain their upper 32 bits on taking an exception from AArch32 state to AArch64 state. In this table, a dash (-) indicates that the RW values are not valid for the target Exception level.

<table>
<thead>
<tr>
<th>SCR_EL3.RW</th>
<th>HCR_EL2.RW or HCR.RW</th>
<th>Registers when the target Exception level is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>EL3: None - EL2: - EL1: -</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Not valid because the RW bit values would imply that EL2 is AArch32 and EL1 is AArch64.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>EL3: X15 EL2: X15 EL1: -</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>EL3: X15-X30 EL2: X15-X30 EL1: X15-X30</td>
</tr>
</tbody>
</table>

If EL2 is not implemented, or SCR_EL3.NS prevents its use, the registers that can have their upper 32 bits retained on taking an exception to AArch64 state from AArch32 state are the same as when HCR_EL2.RW has the same value as SCR_EL3.RW.

SPSR, ELR, and AArch64 SP relationships on changing Execution state

The following table shows SPSR and ELR registers that are architecturally mapped between AArch32 state and AArch64 state:

<table>
<thead>
<tr>
<th>AArch32 register</th>
<th>AArch64 register</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPSR_svc</td>
<td>SPSR_EL1</td>
</tr>
<tr>
<td>SPSR_hyp</td>
<td>SPSR_EL2</td>
</tr>
<tr>
<td>ELR_hyp</td>
<td>ELR_EL2</td>
</tr>
</tbody>
</table>

When an exception is taken from an Exception level using AArch32 to EL3 using AArch64 and EL2 has been using AArch32 state, the upper 32 bits of ELR_EL2 have one of the following IMPLEMENTATION DEFINED values:
- The upper 32 bits retain the value that the same architectural register held before any AArch32 execution.
- The upper 32 bits are set to zero.

The following AArch32 registers are used only during execution in AArch32 state. However, they retain their state if there is execution at EL1 and EL1 is using AArch64 state.
- SPSR_abt.
- SPSR_und.
- SPSR_irq.
- SPSR_fiq.

For the purposes of context switching, the registers in RWFDMR are accessible during execution in AArch64 state at Exception levels higher than EL1.

If EL1 does not support execution in AArch32 state, the registers in RWFDMR are RES0.

If an exception is taken from an Exception level using AArch32 to an Exception level using AArch64, the AArch64 stack pointers and Exception Link Registers associated with an Exception level that are not accessible during AArch32 execution at the Exception level from which the exception was taken, retain the state that they had before AArch32 execution. This applies to the following registers:
- SP_EL0.
- SP_EL1.
- SP_EL2.
• ELR_EL1.
Chapter D2
AArch64 Self-hosted Debug

When the PE is using self-hosted debug, it generates *debug exceptions*. This chapter describes the AArch64 self-hosted debug exception model. It is organized as follows:

**Introductory information**
- *About self-hosted debug* on page D2-4692.
- *The debug exception enable controls* on page D2-4696.

**The debug Exception model**
- *Routing debug exceptions* on page D2-4697.
- *Enabling debug exceptions from the current Exception level* on page D2-4699.
- *The effect of powerdown on debug exceptions* on page D2-4701.
- *Summary of the routing and enabling of debug exceptions* on page D2-4702.
- *Pseudocode description of debug exceptions* on page D2-4704.

**The debug exceptions**
- *Breakpoint Instruction exceptions* on page D2-4705.
- *Breakpoint exceptions* on page D2-4707.
- *Watchpoint exceptions* on page D2-4726.
- *Vector Catch exceptions* on page D2-4741.
- *Software Step exceptions* on page D2-4742.

**Synchronization requirements**

The behavior of self-hosted debug after changes to System registers, or after changes to the authentication interface, but before a *Context synchronization event* guarantees the effects of the changes:
- *Synchronization and debug exceptions* on page D2-4755.
D2.1 About self-hosted debug

Self-hosted debug supports debugging through the generation and handling of debug exceptions, which are taken using the exception model described in Chapter D1 The AArch64 System Level Programmers’ Model. This section introduces some terms that are used in describing self-hosted debug, and then introduces the debug exceptions. See:

- Definition of a debugger in the context of self-hosted debug.
- Context ID and Process ID.
- About debug exceptions.

D2.1.1 Definition of a debugger in the context of self-hosted debug

Within this chapter, debugger means that part of an operating system, or higher level of system software, that handles debug exceptions and programs the Debug System registers. An operating system with rich application environments might provide debug services that support a debugger user interface executing at EL0. From the architectural perspective, the debug services are the debugger.

D2.1.2 Context ID and Process ID

A CONTEXTIDR_ELx identifies the current Context ID, which is used by:

- The debug logic, for breakpoint and watchpoint matching.
- Implemented trace logic, to identify the current process.

In AArch64 state, the CONTEXTIDR_ELx has a single field, PROCID, that is defined as the Process Identifier (Process ID). Therefore, in AArch64 state, the Context ID and Process ID are identical.

D2.1.3 About debug exceptions

Debug exceptions occur during normal program flow if a debugger has programmed the PE to generate them. For example, a software developer might use a debugger contained in an operating system to debug an application. To do this, the debugger enables one or more debug exceptions. The debug exceptions that can be generated in stage 1 of an AArch64 translation regime are:

- Breakpoint Instruction exceptions on page D2-4693.
- Breakpoint exceptions on page D2-4693, generated by hardware breakpoints.
- Watchpoint exceptions on page D2-4693, generated by hardware watchpoints.
- Software Step exceptions on page D2-4694.

In addition, debug exceptions generated in an AArch32 translation regime might be routed to EL2 using AArch64. See Routing debug exceptions on page D2-4697. Chapter G2 describes the debug exceptions that can be generated in an AArch32 translation regime.

Vector Catch exceptions are exceptions that cannot be generated in an AArch64 translation regime but can be generated in stage 1 of an AArch32 translation regime and routed to EL2 using AArch64. Vector Catch exceptions on page D2-4741 describes the behavior for this case.

The PE can generate a particular debug exception only when both:

1. Debug exceptions are enabled from the current Exception level and Security state.
   See Enabling debug exceptions from the current Exception level on page D2-4699. Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.

2. A debugger has enabled that particular debug exception.
   All of the debug exceptions except for Breakpoint Instruction exceptions have an enable control contained in the MDSCR_EL1. See The debug exception enable controls on page D2-4696.

Note

If halting is allowed and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause entry to Debug state instead of causing debug exceptions. In Debug state, the PE is halted.
For the definition of halting is allowed, see *Halting allowed and halting prohibited on page H2-10233.*

The following list summarizes each of the debug exceptions:

**Breakpoint Instruction exceptions**

*Breakpoint instructions* generate these. Breakpoint instructions are instructions that software developers can use to cause exceptions at particular points in the program flow.

The breakpoint instruction in the A64 instruction set is `BRK #<immediate>`. Whenever one of these is committed for execution, the PE takes a Breakpoint Instruction exception.

**PE behavior**

Breakpoint Instruction exceptions cannot be masked. The PE takes Breakpoint Instruction exceptions regardless of both of the following:

- The current Exception level.
- The current Security state.

For more information, see *Breakpoint Instruction exceptions on page D2-4705.*

**Breakpoint exceptions**

The A-profile architecture provides 2-16 hardware breakpoints. These can be programmed to generate Breakpoint exceptions based on particular instruction addresses, or based on particular PE contexts, or both.

For example, a software developer might program a hardware breakpoint to generate a Breakpoint exception whenever the instruction with address 0x1000 is committed for execution.

The A-profile architecture supports the following types of hardware breakpoint for use in stage 1 of an AArch64 translation regime:

- **Address.** Comparisons are made with the virtual address of each instruction in the program flow.
- **Context:**
  - Context ID Match. Matches with the Context ID held in the CONTEXTIDR_EL1.
  - VMID Match. Matches with the VMID value held in the VTTBR_EL2.
  - Context ID and VMID Match. Matches with both the Context ID and the VMID value.

An Address breakpoint can link to a Context breakpoint, so that the Address breakpoint generates a Breakpoint exception only if the PE is in a particular context when the address match occurs.

A breakpoint generates a Breakpoint exception whenever an instruction that causes a match is committed for execution.

**PE behavior**

If halting is allowed and EDSCR.HDE is 1, hardware breakpoints cause entry to Debug state. That is, they halt the PE. See Chapter H2 Debug State. Otherwise:

- If debug exceptions are enabled, hardware breakpoints cause Breakpoint exceptions.
- If debug exceptions are disabled, hardware breakpoints are ignored.

For more information, see *Breakpoint exceptions on page D2-4707.*

**Watchpoint exceptions**

The A-profile architecture provides 2-16 hardware watchpoints. These can be programmed to generate Watchpoint exceptions based on accesses to particular data addresses, or based on accesses to any address in a data address range.

For example, a software developer might program a hardware watchpoint to generate a Watchpoint exception on an access to any address in the data address range 0x1000 - 0x101F.
A hardware watchpoint can link to a hardware breakpoint if the hardware breakpoint is a *Linked Context* type. In this case, the watchpoint generates a Watchpoint exception only if the PE is in a particular context when the data address match occurs.

The smallest data address size that a watchpoint can be programmed to match on is a byte. A single watchpoint can be programmed to match on one or more bytes.

A watchpoint generates a Watchpoint exception whenever an instruction that initiates an access that causes a match is committed for execution.

**PE behavior**

- If halting is allowed and `EDSCR.HDE` is 1, hardware watchpoints cause entry to Debug state. That is, they halt the PE. See Chapter H2 *Debug State*.
- Otherwise:
  - If debug exceptions are enabled, hardware watchpoints cause Watchpoint exceptions.
  - If debug exceptions are disabled, hardware watchpoints are ignored.

For more information, see *Watchpoint exceptions* on page D2-4726.

**Vector Catch exceptions**

These are not generated in an AArch64 translation regime. They can be generated only in an AArch32 translation regime. See *Vector Catch exceptions* on page D2-4741.

**Software Step exceptions**

Software step is a resource that a debugger can use to make the PE single-step instructions.

For example, by using software step, debugger software executing at a higher Exception level can debug software executing at a lower Exception level, by making it single-step instructions.

After the software being debugged has single-stepped an instruction, the PE takes a Software Step exception.

**PE behavior**

Software step can be used only by a debugger executing in an Exception level that is using AArch64. However, the instruction stepped might be executed in either Execution state, and therefore Software Step exceptions can be taken from either Execution state.

If debug exceptions are enabled, Software Step exceptions can be generated.

If debug exceptions are disabled, software step is inactive.

For more information, see *Software Step exceptions* on page D2-4742.

Table D2-1 on page D2-4694 summarizes PE behavior and shows the location of the pseudocode for each of the debug exceptions.

**Table D2-1 PE behavior and pseudocode for each of the debug exceptions**

<table>
<thead>
<tr>
<th>Debug exception</th>
<th>PE behavior if debug exceptions are:</th>
<th>Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Enabled</td>
<td>Disabled</td>
</tr>
<tr>
<td>Breakpoint Instruction</td>
<td>Takes the exception</td>
<td>Takes the exception</td>
</tr>
<tr>
<td>exceptions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Breakpoint exceptions</td>
<td>Takes the exception</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pseudocode description of Breakpoint Instruction exceptions on page D2-4706

Pseudocode description of Breakpoint exceptions taken from AArch64 state on page D2-4724
<table>
<thead>
<tr>
<th>Debug exception</th>
<th>PE behavior if debug exceptions are:</th>
<th>Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Enabled</td>
<td>Disabled</td>
</tr>
<tr>
<td>Watchpoint exceptions</td>
<td>Takes the exception&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Ignored</td>
</tr>
<tr>
<td>Vector Catch exceptions</td>
<td>Takes the exception</td>
<td>Ignored</td>
</tr>
<tr>
<td>Software Step exceptions</td>
<td>Takes the exception</td>
<td>Not applicable&lt;sup&gt;b&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

<sup>a</sup> If halting is allowed and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause the PE to enter Debug state instead of causing debug exceptions. See Chapter H2 Debug State.

<sup>b</sup> Software Step is inactive if debug exceptions are disabled. No Software Step exceptions can be generated.
D2.2 The debug exception enable controls

The enable controls for each debug exception are as follows:

**Breakpoint Instruction exceptions**

None. Breakpoint Instruction exceptions are always enabled.

**Breakpoint exceptions**

MDSCR\_EL1.MDE, plus an enable control for each breakpoint, DBGBCR\_n\_EL1.E.

**Watchpoint exceptions**

MDSCR\_EL1.MDE, plus an enable control for each watchpoint, DBGWCR\_n\_EL1.E.

**Vector Catch exceptions**

MDSCR\_EL1.MDE.

**Software Step exceptions**

MDSCR\_EL1.SS.

In addition, for all debug exceptions other than Breakpoint Instruction exceptions, software must configure the controls that enable debug exceptions from the current Exception level and Security state. See *Enabling debug exceptions from the current Exception level* on page D2-4699.

The PE cannot take a debug exception if debug exceptions are disabled from either the current Exception level or the current Security state.

Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.
**D2.3 Routing debug exceptions**

If the OS Lock is locked, or if `DoubleLockStatus() == TRUE`, a Debug exception cannot be taken.

Debug exceptions are enabled and routed according to the following controls:

- `HCR_EL2.TGE`.
- `MDCR_EL2.TDE`.
- `MDCR_EL3.SDD`.
- The Security state when the exception is taken.
- The Exception level where the exception is taken.

Breakpoint Instructions are enabled in some situations where other Debug exceptions are disabled.

--- Note ---

If EL2 is not implemented, the Effective value of `HCR_EL2.TGE` is 0 and the Effective value of `MDCR_EL2.TDE` is 0. Throughout this section, references to the values of these fields are to the Effective values of the fields.

If EL3 is not implemented, and the implementation is a Secure state only implementation, the Effective value of `MDCR_EL3.SDD` is 0.

Table D2-2 on page D2-4697 shows when debug exceptions are enabled from the current Security state.

### Table D2-2 Whether debug exceptions are enabled from the current Security state

<table>
<thead>
<tr>
<th>Current Security state</th>
<th>Breakpoint Instruction exceptions</th>
<th>All other debug exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure</td>
<td>Enabled</td>
<td>Enabled</td>
</tr>
<tr>
<td>Secure</td>
<td>Enabled</td>
<td>Disabled if <code>MDCR_EL3.SDD</code> is 1. See <a href="#">Disabling debug exceptions from Secure state on page D2-4699</a>. Otherwise enabled.</td>
</tr>
</tbody>
</table>

The routing of debug exceptions is as follows:

**Debug exceptions taken when EL2 is implemented and enabled in the current Security state**

The routing of debug exceptions taken depends on the values of `MDCR_EL2.TDE` and `HCR_EL2.TGE`:

- If the Effective value of `{MDCR_EL2.TDE, HCR_EL2.TGE}` is not `{0, 0}`
  - Debug exceptions are routed to EL2, EL₀ is EL2.

- Otherwise
  - Debug exceptions behave as follows:
    - Debug exceptions taken from EL1 and EL0 are routed to EL1. EL₀ is EL1
    - Breakpoint Instruction exceptions taken from EL2 are routed to EL2.
    - All other debug exceptions are disabled from EL2 using AArch64.

**When EL3 is implemented**

Breakpoint Instruction exceptions taken from EL3 are routed to EL3.

All other debug exceptions are disabled from EL3 using AArch64.

**Otherwise**

Debug exceptions are routed to EL1.

This means that, for all debug exceptions, the debug target Exception level, EL₀, is either EL1 or EL2. When executing in the same exception level as EL₀, see [Enabling debug exceptions from the current Exception level on page D2-4699](#).
Table D2-3, Table D2-4, and Table D2-5 show the routing of debug exceptions. In these tables:

**NS**
Means the Effective value of SCR_EL3.NS.

**EEL2**
Means the Effective value of SCR_EL3.EEL2. If FEAT_SEL2 is not implemented, this is 0.

**TDE or TGE**
Means the logical OR of the Effective value of MDCR_EL2.TDE and the Effective value of HCR_EL2.TGE.

**(ELx)**
Means EL_D is ELx. However:
- All debug exceptions other than Breakpoint Instruction exceptions are disabled from this Exception level.
- Breakpoint Instruction exceptions taken when executing in this Exception level are routed to the same Exception level. This may not be the same as the EL_D Exception level.

**ELx**
Means EL_D is ELx.

### Table D2-3 Routing when both EL3 and EL2 are implemented

<table>
<thead>
<tr>
<th>NS</th>
<th>EEL2</th>
<th>TDE or TGE</th>
<th>EL_D when executing in:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>EL0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>x</td>
<td>EL1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>EL1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>EL2</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>EL1</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>EL2</td>
</tr>
</tbody>
</table>

### Table D2-4 Routing when EL3 is implemented and EL2 is not implemented

<table>
<thead>
<tr>
<th>EL_D when executing in:</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL0</td>
</tr>
<tr>
<td>EL1</td>
</tr>
</tbody>
</table>

### Table D2-5 Routing when EL3 is not implemented and EL2 is implemented

<table>
<thead>
<tr>
<th>TDE</th>
<th>EL_D when executing in:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EL0</td>
</tr>
<tr>
<td>0</td>
<td>EL1</td>
</tr>
<tr>
<td>1</td>
<td>EL2</td>
</tr>
</tbody>
</table>

### D2.3.1 Pseudocode description of routing debug exceptions

**DebugTarget()** returns the current debug target Exception level.

**DebugTargetFrom()** returns the debug target Exception level for the specified Security state.

These functions are described in Chapter J1 *Armv8 Pseudocode*. 
D2.4  Enabling debug exceptions from the current Exception level

A debug exception can be taken only if all of the following are true:

- The OS Lock is unlocked.
- `DoubleLockStatus() == FALSE.`
- The debug exception is enabled from the current Exception level.
- The debug exception is enabled from the current Security state.

Debug exceptions are enabled from the current Exception level, where EL_D is the Exception level that Table D2-3 on page D2-4698 defines, as follows:

When executing at any Exception level that is higher than EL_D:

- Breakpoint Instruction exceptions are enabled.
- All other debug exceptions are disabled.

When executing at EL_D:

- Breakpoint Instruction exceptions are enabled.
- All other debug exceptions are disabled if either of the following is true:
  - The Local (kernel) Debug Enable bit, MDSCR_EL1.KDE, is 0.
  - The Debug exception mask bit, PSTATE.D, is 1.
- Otherwise enabled.

This means that a debugger must explicitly enable these debug exceptions from EL_D by setting MDSCR_EL1.KDE to 1 and PSTATE.D to 0.

When executing at EL1, EL_D is EL2, and FEAT_NV2 is implemented:

- Watchpoint debug exceptions generated by a System register access converted to a memory access because HCR_EL2.NV2 is 1 are disabled if MDSCR_EL1.KDE is 0 and enabled if MDSCR_EL1.KDE is 1. The value of PSTATE.D is ignored.
- All other debug exceptions are enabled.

Otherwise:

- All debug exceptions are enabled.

D2.4.1  Disabling debug exceptions from Secure state

If EL3 is implemented, software executing at EL3 can set the Secure Debug Disable bit, MDCR_EL3.SDD, to 1 to disable all debug exceptions taken from AArch64 Secure state other than Breakpoint Instruction exceptions.

The A-profile architecture does not support disabling debug in Non-secure state.

--- Note ---

- If the boot software executed when reset is deasserted sets MDCR_EL3.SDD to 1, software operating at EL3 never has to switch the debug registers between Secure state and Non-secure state.
- The PE cannot take a debug exception unless it is enabled from the current Exception level. See Table D2-6.
- If either the OS Lock or the OS Double Lock is locked, debug exceptions other than Breakpoint Instruction exceptions are disabled.
- If EL3 and EL2 are not implemented, and the implementation is a Secure state only implementation, the PE behaves as if MDCR_EL3.SDD is 0.

D2.4.2  Pseudocode description of enabling debug exceptions

`AArch64.GenerateDebugExceptions()` determines whether debug exceptions other than Breakpoint Instruction exceptions are enabled from the current Exception level and Security state.
AArch64.GenerateDebugExceptionsFrom() determines whether debug exceptions other than Breakpoint Instruction exceptions are enabled from the specified Exception level and Security state.

These functions are described in Chapter J1 Armv8 Pseudocode.
D2.5 The effect of powerdown on debug exceptions

Debug OS Save and Restore sequences on page H6-10342 describes the powerdown save routine and the restore routine.

When executing either routine, software must use the OS Lock to disable generation of all of the following:

• Breakpoint exceptions.
• Watchpoint exceptions.
• Vector Catch exceptions.
• Software Step exceptions.

This is because the generation of these exceptions depends on the state of the debug registers, and the state of the debug registers might be lost over these routines.

If the OS Lock is unlocked, and DoubleLockStatus() == FALSE, debug exceptions other than Breakpoint Instruction exceptions are enabled.

If OS Lock is locked, or if DoubleLockStatus() == TRUE, debug exceptions other than Breakpoint Instruction exceptions are disabled.

Breakpoint Instruction exceptions are enabled regardless of the state of the OS Lock and the OS Double Lock.
D2.6 Summary of the routing and enabling of debug exceptions

Behavior is as follows:

**Breakpoint Instruction exceptions**

These are always enabled, regardless of the current Exception level and Security state. A Breakpoint Instruction exception taken from EL3 is always routed to EL3. A Breakpoint Instruction exception taken from EL2 is routed to EL2. A Breakpoint Instruction exception taken from EL0 or EL1 is always routed to EL_D.

**All other debug exceptions**

Table D2-6 on page D2-4702 shows the valid combinations of MDCR_EL3.SDD, MDCR_EL2.TDE, MDSCR_EL1.KDE, and PSTATE.D, and for each combination shows where these exceptions are enabled from and where they are taken to.

In the table:

- **Lock** Means the value of (OSLSR_EL1.OSLK == '1' || DoubleLockStatus()).
- **NS** Means the **Effective value** of SCR_EL3.NS.
- **SDD** Means the **Effective value** of MDCR_EL3.SDD. See Disabling debug exceptions from Secure state on page D2-4699.
- **EEL2** Means the **Effective value** of SCR_EL3.EEL2. If FEAT_SEL2 is not implemented, this is 0.
- **TGE** Means the value of HCR_EL2.TGE. If EL2 is not implemented, the PE behaves as if this is 0.
- **TDE** Means the value of MDCR_EL2.TDE. If EL2 is not implemented, the PE behaves as if this is 0.
- **KDE** Means the value of MDSCR_EL1.KDE.
- **D** Means the value of PSTATE.D.
- **n/a** Means not applicable. The PE cannot be executing at this Exception level.
- **-** Means that debug exceptions are disabled from that Exception level.

**Table D2-6 Routing of Breakpoint, Watchpoint, Software Step, and Vector Catch exceptions**

<table>
<thead>
<tr>
<th>Debug state</th>
<th>Lock</th>
<th>NS</th>
<th>SDD</th>
<th>EEL2</th>
<th>TGE</th>
<th>TDE</th>
<th>KDE</th>
<th>D</th>
<th>EL_D when enabled from:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>EL0 EL1 EL2 EL3</td>
</tr>
<tr>
<td>Yes</td>
<td>TRUE</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>- - - -</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>- - - -</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>EL1</td>
<td>n/a</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>EL1</td>
<td>EL1</td>
<td>n/a</td>
<td>-</td>
<td>EL1</td>
<td>n/a</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>EL1</td>
<td>EL1</td>
<td>n/a</td>
<td>-</td>
<td>EL1</td>
<td>n/a</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>EL1</td>
<td>EL1</td>
<td>-</td>
<td>-</td>
<td>EL1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>EL1</td>
<td>EL1</td>
<td>-</td>
<td>-</td>
<td>EL1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Table D2-6 Routing of Breakpoint, Watchpoint, Software Step, and Vector Catch exceptions

<table>
<thead>
<tr>
<th>Debug state</th>
<th>Lock</th>
<th>NS</th>
<th>SDD</th>
<th>EEL2</th>
<th>TGE</th>
<th>TDE</th>
<th>KDE</th>
<th>D</th>
<th>EL0 when enabled from:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FALSE</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>EL2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>0</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EL2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td></td>
<td></td>
<td>EL1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>0</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EL2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>0</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EL2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
D2.7 Pseudocode description of debug exceptions

`AArch64.DebugFault()` returns a `FaultRecord` object that indicates that a memory access has generated a debug exception:

The `AArch64.Abort()` function processes `FaultRecord` objects, as described in *Abort exceptions* on page D7-5076, and generates a debug exception.

Some functions called by `AArch64.Abort()` are:

- `AArch64.BreakpointException()`.
- `AArch64.WatchpointException()`.
- `AArch64.VectorCatchException()`.
- `AArch64.InstructionAbort()`.
- `AArch64.DataAbort()`.

These functions are defined in *Chapter J1 Armv8 Pseudocode*.
D2.8 Breakpoint Instruction exceptions

This section describes Breakpoint Instruction exceptions in an AArch64 translation regime.

The PE is using an AArch64 translation regime when it is executing either:

• In an Exception level that is using AArch64.
• At EL0 using AArch32 when EL1 is using AArch64.

For software executing in an Exception level that is using AArch64, a Breakpoint Instruction exception results from the execution of an A64 BRK instruction. However, within the AArch64 EL1&0 translation regime, executing a T32 or A32 BKPT instruction at EL0 using AArch32 generates a Breakpoint Instruction exception.

For more information about the T32 and A32 BKPT instructions, see:

• Breakpoint instruction in the A32 and T32 instruction sets on page G2-9051.
• BKPT instructions as the first instruction in an IT block on page G2-9052.

The following subsections describe Breakpoint Instruction exceptions in an AArch64 translation regime:

• About Breakpoint Instruction exceptions.
• Breakpoint instructions.
• Exception syndrome information and preferred return address on page D2-4706.
• Pseudocode description of Breakpoint Instruction exceptions on page D2-4706.

D2.8.1 About Breakpoint Instruction exceptions

A breakpoint is an event that results from the execution of an instruction, which is based on either:

• The instruction address, the PE context, or both. This type of breakpoint is called a hardware breakpoint.
• The instruction itself. That is, the instruction is a breakpoint instruction. These can be included in the program that the PE executes. This type of breakpoint is called a software breakpoint.

Breakpoint Instruction exceptions, which this section describes, are software breakpoints. Breakpoint exceptions on page D2-4707 describes hardware breakpoints.

There is no enable control for Breakpoint Instruction exceptions. They are always enabled, and cannot be masked. A Breakpoint Instruction exception is generated whenever a breakpoint instruction is committed for execution, regardless of all of the following:

• The current Exception level.
• The current Security state.
• Whether the debug target Exception level, EL_D, is using AArch64 or AArch32.

Note

• The debug target Exception level, EL_D, is the Exception level that debug exceptions are targeting. Routing debug exceptions on page D2-4697 describes how EL_D is derived.
• Debuggers using breakpoint instructions must be aware of the rules for concurrent modification and execution of instructions. See Concurrent modification and execution of instructions on page B2-156.

D2.8.2 Breakpoint instructions

The breakpoint instruction in the A64 instruction set is BRK #<immediate>. It is unconditional.

For details of the instruction encoding, see BRK on page C6-1237.

The breakpoint instruction in the A32 and T32 instruction sets is BKPT #<immediate>.

For more information about the A32 and T32 breakpoint instruction, see Breakpoint instruction in the A32 and T32 instruction sets on page G2-9051.
D2.8.3 Exception syndrome information and preferred return address

See the following:
• Exception syndrome information.
• Preferred return address.

Exception syndrome information

On taking a Breakpoint Instruction exception, the PE records information about the exception in the Exception Syndrome Register (ESR) at the Exception level the exception is taken to. The ESR used is one of:
• ESR_EL1.
• ESR_EL2.
• ESR_EL3.

Note
Breakpoint Instruction exceptions are the only debug exception that can be taken to EL3 using AArch64.

Table D2-7 on page D2-4706 shows the information that the PE records.

<table>
<thead>
<tr>
<th>ESR_ELx field</th>
<th>Information recorded in ESR_EL1, ESR_EL2, or ESR_EL3.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception Class, EC</td>
<td>Whether the breakpoint instruction was executed in AArch64 state or AArch32 state. The PE sets this to:</td>
</tr>
<tr>
<td></td>
<td>• 0x3C for an A64 BRK instruction.</td>
</tr>
<tr>
<td></td>
<td>• 0x38 for an A32 or T32 BKPT instruction.</td>
</tr>
<tr>
<td>Instruction Length, IL</td>
<td>The PE sets this to:</td>
</tr>
<tr>
<td></td>
<td>• 0 for a 16-bit T32 BKPT instruction.</td>
</tr>
<tr>
<td></td>
<td>• 1 for an A64 BRK instruction, or an A32 BKPT instruction.</td>
</tr>
<tr>
<td>Instruction Specific Syndrome, ISS</td>
<td>RES0.</td>
</tr>
<tr>
<td>ISS[24:16]</td>
<td>The PE copies the instruction Comment field value into here, zero extended as necessary.</td>
</tr>
<tr>
<td>ISS[15:0]</td>
<td></td>
</tr>
</tbody>
</table>

Note

• If debug exceptions are routed to EL2, it is the exception that is routed, not the instruction that is trapped. Therefore, if a Breakpoint Instruction exception is routed to EL2, ESR_EL2.EC is set to the same value as if the exception was taken to EL1.

• For information about how debug exceptions can be routed to EL2, see Routing debug exceptions on page D2-4697.

Preferred return address

The preferred return address is the address of the breakpoint instruction, not the next instruction. This is different to the behavior of other exception-generating instructions, like SVC.

D2.8.4 Pseudocode description of Breakpoint Instruction exceptions

AArch64.SoftwareBreakpoint() generates a Breakpoint Instruction exception that is taken to AArch64 state.

This function is defined in Chapter J1 Armv8 Pseudocode.
D2.9 Breakpoint exceptions

This section describes Breakpoint exceptions in stage 1 of an AArch64 translation regime.

The PE is using an AArch64 translation regime when it is executing either:
• In an Exception level that is using AArch64.
• At EL0 using AArch32 when EL1 is using AArch64.

This section contains the following subsections:
• About Breakpoint exceptions.
• Breakpoint types and linking of breakpoints on page D2-4708.
• Execution conditions for which a breakpoint generates Breakpoint exceptions on page D2-4717.
• Breakpoint instruction address comparisons on page D2-4718.
• Breakpoint context comparisons on page D2-4720.
• Breakpoint usage constraints on page D2-4721.
• Preferred return address on page D2-4724.
• Pseudocode description of Breakpoint exceptions taken from AArch64 state on page D2-4724.

D2.9.1 About Breakpoint exceptions

A breakpoint is an event that results from the execution of an instruction, which is based on either:
• The instruction address, the PE context, or both. This type of breakpoint is called a hardware breakpoint.
• The instruction itself. That is, the instruction is a breakpoint instruction. These can be included in the program that the PE executes. This type of breakpoint is called a software breakpoint.

Breakpoint exceptions are generated by Breakpoint debug events. Breakpoint debug events are generated by hardware breakpoints. Software breakpoints are described in Breakpoint Instruction exceptions on page D2-4705.

An implementation can include between 2-16 hardware breakpoints. ID_AA64DFR0_EL1.BRPs shows how many are implemented.

To use an implemented hardware breakpoint, a debugger programs the following registers for the breakpoint:
• The Breakpoint Control Register, DBGBCR<n>_EL1. This contains controls for the breakpoint, for example an enable control.
• The Breakpoint Value Register, DBGBVR<n>_EL1. This holds the value used for breakpoint matching, which is one of:
  — An instruction virtual address.
  — A Context ID.
  — A VMID value.
  — A concatenation of both a Context ID value and a VMID value.

These registers are numbered, so that:
• DBGBCR0_EL1 and DBGBVR0_EL1 are for breakpoint number zero.
• DBGBCR1_EL1 and DBGBVR1_EL1 are for breakpoint number one.
• DBGBCR2_EL1 and DBGBVR2_EL1 are for breakpoint number two.
• ...
• ...
• DBGBCR<n-1>_EL1 and DBGBVR<n-1>_EL1 are for breakpoint number (n-1).

A debugger can link a breakpoint that is programmed with an address and a breakpoint that is programmed with anything other than an address together, so that a Breakpoint debug event is generated only if both breakpoints match.
For each instruction in the program flow, all of the breakpoints are tested. When a breakpoint is tested, it generates a Breakpoint debug event if all of the following are true:

- The breakpoint is enabled. That is, the breakpoint enable control for it, DBGBCR<n>_EL1.E, is 1.
- The conditions specified in the DBGBCR<n>_EL1 are met.
- The comparison with the value held in the DBGBVR<n>_EL1 is successful.
- If the breakpoint is linked to another breakpoint, the comparisons made by that other breakpoint are also successful.
- The instruction is committed for execution.

If all of these conditions are met, the breakpoint generates the Breakpoint debug event regardless of the following:

- Whether the instruction passes its Condition code check.
- The instruction type.

If halting is allowed and EDSCR.HDE is 1, Breakpoint debug events cause entry to Debug state. Otherwise, if debug exceptions are:

- Enabled, Breakpoint debug events generate Breakpoint exceptions.
- Disabled, Breakpoint debug events are ignored.

**Note**
The remainder of this Breakpoint exceptions section, including all subsections, describes breakpoints as generating Breakpoint exceptions.

However, the behavior described also applies if breakpoints are causing entry to Debug state.

The debug exception enable controls on page D2-4696 describes the enable controls for Breakpoint debug events.

### D2.9.2 Breakpoint types and linking of breakpoints

Each implemented breakpoint is one of the following:

- A context-aware breakpoint. This is a breakpoint that can be programmed to generate a Breakpoint exception on any one of the following:
  - An instruction address match.
  - A Context ID match, with the value held in the CONTEXTIDR_EL1.
  - A VMID match, with the VMID value held in the VTTBR_EL2.
  - Both a Context ID match and a VMID match.
- A breakpoint that is not context-aware. These can only be programmed to generate a Breakpoint exception on an instruction address match.

ID_AA64DFR0_EL1.CTX_CMPs shows how many of the implemented breakpoints are context-aware breakpoints. At least one implemented breakpoint must be context-aware. The context-aware breakpoints are the highest numbered breakpoints.

Any breakpoint that is programmed to generate a Breakpoint exception on an instruction address match is categorized as an Address breakpoint. Breakpoints that are programmed to match on anything else are categorized as Context breakpoints.

When a debugger programs a breakpoint to be an Address or a Context breakpoint, it must also program that breakpoint so that it is either:

- Used in isolation. In this case, the breakpoint is called an Unlinked breakpoint.
- Enabled for linking to another breakpoint. In this case, the breakpoint is called a Linked breakpoint.
By linking an Address breakpoint and a Context breakpoint together, the debugger can create a breakpoint pair that generates a Breakpoint exception only if the PE is in a particular context when an instruction address match occurs. For example, a debugger might:

1. Program breakpoint number one to be a **Linked Address Match breakpoint**.
2. Program breakpoint number five to be a **Linked Context ID Match breakpoint**.
3. Link these two breakpoints together. A Breakpoint exception is generated only if both the instruction address matches and the Context ID matches.

The **Breakpoint Type** field for a breakpoint, DBGBCR<n>_EL1.BT, controls the breakpoint type and whether the breakpoint is enabled for linking. If BT[0] is 1, the breakpoint is enabled for linking.

If AArch32 state is implemented, Address breakpoints can be programmed to generate Breakpoint exceptions on addresses that are halfword-aligned but not word-aligned. This makes it possible to breakpoint on T32 instructions. See **Specifying the halfword-aligned address that an Address breakpoint matches on** on page D2-4719.

--- **Note**

Stage 1 of an AArch32 translation regimes supports two additional breakpoint types, Unlinked and Linked Address Mismatch breakpoints, BT == 0b0010 and BT == 0b0101. For information about these, see Chapter G2 **AArch32 Self-hosted Debug**. These types are reserved in stage 1 of an AArch64 translation regime. See **Reserved BT values** on page D2-4722.

---

### Rules for linking breakpoints

The rules for breakpoint linking are as follows:

- Only Linked breakpoint types can be linked.

- Any type of Linked Address breakpoint can link to any type of Linked Context breakpoint. The **Linked Breakpoint Number** field, DBGBCR<n>_EL1.LBN, for the Linked Address breakpoint specifies the particular Linked Context breakpoint that the Linked Address breakpoint links to, and:
  
  - **DBGBCR<n>_EL1.{SSC, HMC, PMC}** for the Linked Address breakpoint define the execution conditions that the breakpoint pair generates Breakpoint exceptions for. See **Execution conditions for which a breakpoint generates Breakpoint exceptions** on page D2-4717.
  
  - **DBGBCR<n>_EL1.{SSC, HMC, PMC}** for the Linked Context breakpoint are ignored.

- Linked Context breakpoint types can only be linked to. The LBN field for Context breakpoints is therefore ignored.

- Linked Address breakpoints cannot link to watchpoints. The LBN field can therefore only specify another breakpoint.

- If a Linked Address breakpoint links to a breakpoint that is not context-aware, the behavior of the Linked Address breakpoint is **CONSTRAINED UNPREDICTABLE**. See **Other usage constraints for Address breakpoints** on page D2-4724.

- If a Linked Address breakpoint links to an Unlinked Context breakpoint, the Linked Address breakpoint never generates any Breakpoint exceptions.

- Multiple Linked Address breakpoints can link to a single Linked Context breakpoint.

--- **Note**

Multiple Linked watchpoints can also link to a single Linked Context breakpoint. **Watchpoint exceptions on** page D2-4726 describes watchpoints.
These rules mean that a single Linked Context breakpoint might be linked to by all, or any combination of, the following:

- Multiple Linked Address Match breakpoints.
- Multiple Linked watchpoints.

**Note**

If FEAT_NV2 is implemented, the hypervisor must use the 0b1101, Linked CONTEXTIDR_EL2 Match breakpoint type to guarantee a linked match, see Interaction with self-hosted and External debug on page D8-5179.

It is also possible that a Linked Context breakpoint might have no breakpoints or watchpoints linked to it.

Figure D2-1 shows an example of permitted breakpoint and watchpoint linking.

---

**Figure D2-1 The role of linking in Breakpoint and Watchpoint exception generation**

In Figure D2-1, each Linked Address breakpoint can generate a Breakpoint exception only if the comparisons made by both it, and the Linked Context breakpoint that it links to, are successful. Similarly, each Linked watchpoint can only generate a Watchpoint exception if the comparisons made by both it, and the Linked Context breakpoint that it links to, are successful.
Breakpoint types defined by DBGBCRn_EL1.BT

The following list provides more detail about each breakpoint type:

0b0000, Unlinked Address Match breakpoint

Generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions. See Execution conditions for which a breakpoint generates Breakpoint exceptions on page D2-4717.
- A successful address match, as described in Breakpoint instruction address comparisons on page D2-4718.

DBGBCR<n>_EL1.LBN for this breakpoint is ignored.

0b0001, Linked Address Match breakpoint

Generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>_EL1.{SSC, HMC, PMC} for this breakpoint. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions on page D2-4717.
- A successful address match defined by this breakpoint, as described in Breakpoint instruction address comparisons on page D2-4718.
- A successful context match defined by the Linked Context breakpoint that this breakpoint links to.

DBGBCR<n>_EL1.LBN for this breakpoint selects the Linked Context breakpoint that this breakpoint links to.

0b0010, Unlinked Context ID Match breakpoint

BT == 0b0010 is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions. See Execution conditions for which a breakpoint generates Breakpoint exceptions on page D2-4717.
- A successful Context ID match, as described in Breakpoint context comparisons on page D2-4720.

The value of DBGBVR<n>_EL1.ContextID is compared with the current Context ID.

CONTEXTIDR_EL2 holds the current Context ID when all of:

- The implementation includes FEAT_VHE.
- EL2 is implemented and enabled in the current Security state.
- EL2 using AArch64 and HCR_EL2.E2H is set to 1.
- The PE is executing at EL0 and HCR_EL2.TGE is 1, or the PE is executing at EL2.
Otherwise, CONTEXTIDR_EL1 holds the current Context ID.

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored.

0b0011, Linked Context ID Match breakpoint

BT == 0b0011 is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons on page D2-4718.
— A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons on page D2-4720.

- Generation of a Watchpoint exception depends on both:
  — A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons on page D2-4731.
  — A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons on page D2-4720.

The value of DBGBVR<\text{n}>_EL1.ContextID is compared with the current Context ID.

CONTEXTIDR_EL2 holds the current Context ID when all of:
- The implementation includes FEAT_VHE.
- EL2 is implemented and enabled in the current Security state.
- EL2 using AArch64 and HCR_EL2.E2H is set to 1.
- The PE is executing at EL0 and HCR_EL2.TGE is 1, or the PE is executing at EL2.

Otherwise, CONTEXTIDR_EL1 holds the current Context ID.

DBGBCR<\text{n}>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

0b0100, Unlinked Address Mismatch breakpoint

BT == 0b0100 is a reserved value in stage 1 of an AArch64 translation regime. See Reserved BT values on page D2-4722.

0b0100, Unlinked Address Mismatch breakpoint on page G2-9059 describes the behavior of Address Mismatch breakpoints in stage 1 of an AArch32 translation regime.

0b0101, Linked Address Mismatch breakpoint

BT == 0b0101 is a reserved value in stage 1 of an AArch64 translation regime. See Reserved BT values on page D2-4722.

0b0101, Linked Address Mismatch breakpoint on page G2-9059 describes the behavior of Address Mismatch breakpoints in stage 1 of an AArch32 translation regime.

0b0110, Unlinked CONTEXTIDR_EL1 Match breakpoint

BT == 0b0110 is a reserved value if either:
- The breakpoint is not a context-aware breakpoint.
- The implementation does not include FEAT_VHE.

In an implementation that includes FEAT_VHE, for context-aware breakpoints, generation of a Breakpoint exception depends on both:
- DBGBCR<\text{n}>_EL1.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons on page D2-4720.

The Context ID check is made against the value in CONTEXTIDR_EL1. The value of DBGBVR<\text{n}>_EL1.ContextID is compared with the Context ID value held in CONTEXTIDR_EL1.

|| Note |
---|---|
The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

DBGBCR<\text{n}>_EL1.{LBN, BAS} for this breakpoint are ignored.

0b0111, Linked CONTEXTIDR_EL1 Match breakpoint

BT == 0b0111 is a reserved value if either:
- The breakpoint is not a context-aware breakpoint.
- The implementation does not include FEAT_VHE.
In an implementation that includes FEAT_VHE, for context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.

- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons on page D2-4718.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons on page D2-4720.

- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons on page D2-4731.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons on page D2-4720.

The Context ID check is made against the value in CONTEXTIDR_EL1. The value of DBGBVR<\(n\)_EL1.ContextID is compared with the Context ID value held in CONTEXTIDR_EL1.

--- Note ---

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

---

\(0b1000\), Unlinked VMID Match breakpoint

\(BT == 0b1000\) is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- EL2 is not implemented.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- \(DBGBCR<\(n\)>_EL1.\{SSC, HMC, BAS, PMC\}\) These define the execution conditions for which the breakpoint generates Breakpoint exceptions. See Execution conditions for which a breakpoint generates Breakpoint exceptions on page D2-4717.

- A successful VMID match, as described in Breakpoint context comparisons on page D2-4720.

\(DBGBCR<\(n\)>_EL1.\{LBN, BAS\}\) for this breakpoint are ignored.

\(0b1001\), Linked VMID Match breakpoint

\(BT == 0b1000\) is a reserved value if either:

- The breakpoint is not a context-matching breakpoint.
- EL2 is not implemented.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.

- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address Match breakpoint that links to this breakpoint. See Breakpoint instruction address comparisons on page D2-4718.
  - A successful VMID match defined by this breakpoint, as described in Breakpoint context comparisons on page D2-4720.
• Generation of a Watchpoint exception depends on both:
  — A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons on page D2-4731.
  — A successful VMID match defined by this breakpoint, as described in Breakpoint context comparisons on page D2-4720.

DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

0b1010, Unlinked Context ID and VMID Match breakpoint

BT == 0b1010 is a reserved value if either:
• The breakpoint is not a context-aware breakpoint.
• EL2 is not implemented.

When EL2 is implemented, for context-aware breakpoints, generation of a Breakpoint exception depends on all of the following:

• DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates a Breakpoint exception for. See Execution conditions for which a breakpoint generates Breakpoint exceptions on page D2-4717.
• A successful Context ID match, as described in Breakpoint context comparisons on page D2-4720.
• A successful VMID match.

The value of DBGBVR<n>_EL1.ContextID is compared with CONTEXTIDR_EL1. Breakpoint context comparisons on page D2-4720 describes the requirements for a successful Context ID match and a successful VMID match.

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored.

0b1011, Linked Context ID and VMID Match breakpoint

BT == 0b1011 is a reserved value if either:
• The breakpoint is not a context-aware breakpoint.
• EL2 is not implemented.

When EL2 is implemented, for context-aware breakpoints, one of the following applies:
• If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.
• Generation of a Breakpoint exception depends on all of the following:
  — A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons on page D2-4718.
  — A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons on page D2-4720.
  — A successful VMID match defined by this breakpoint.
• Generation of a Watchpoint exception depends on all of the following:
  — A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons on page D2-4731.
  — A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons on page D2-4720.
  — A successful VMID match defined by this breakpoint.

The value of DBGBVR<n>_EL1.ContextID is compared with CONTEXTIDR_EL1. Breakpoint context comparisons on page D2-4720 describes the requirements for a successful Context ID match and a successful VMID match by this breakpoint.

DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.
0b1100, Unlinked CONTEXTIDR_EL2 Match breakpoint

BT == 0b1100 is a reserved value if either:
- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.

In an implementation in which FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented, for context-aware breakpoints, generation of a Breakpoint exception depends on both:
- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful CONTEXTIDR_EL2 match, as described in Breakpoint context comparisons on page D2-4720.

The Context ID check is made against the value in CONTEXTIDR_EL2. The value of DBGBVR<n>_EL1 is compared with the Context ID value held in CONTEXTIDR_EL2.

Note

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

0b1101, Linked CONTEXTIDR_EL2 Match breakpoint

BT == 0b1101 is a reserved value if either:
- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.

In an implementation in which FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented, for context-aware breakpoints, either:
- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons on page D2-4718.
  - A successful CONTEXTIDR_EL2 match, as described in Breakpoint context comparisons on page D2-4720.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons on page D2-4731.
  - A successful CONTEXTIDR_EL2 match, as described in Breakpoint context comparisons on page D2-4720.

The Context ID check is made against the value in CONTEXTIDR_EL2. The value of DBGBVR<n>_EL1 is compared with the Context ID value held in CONTEXTIDR_EL2.

Note

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

0b1110, Unlinked Full Context ID Match breakpoint

BT == 0b1110 is a reserved value if either:
- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.
In an implementation in which FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented, for context-aware breakpoints, generation of a Breakpoint exception depends on both:

- \( \text{DBGBCR}^{<n>}_\text{EL1}.\{\text{SSC, HMC, PMC}\} \). These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful Context ID match, as described in Breakpoint context comparisons on page D2-4720.

The Context ID check is made against the values in both CONTEXTIDR_EL1 and CONTEXTIDR_EL2. The value of DBGBVR^{<n>}_EL1[31:0] is compared with the Context ID value held in CONTEXTIDR_EL1, and the value of DBGBVR^{<n>}_EL1[63:32] is compared with the Context ID value held in CONTEXTIDR_EL2. Both comparisons must match for the Context ID check.

--- Note ---

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

---

\( 0b1111 \), Linked Full Context ID Match breakpoint

- BT == \( 0b1111 \) is a reserved value if either:
  - The breakpoint is not a context-aware breakpoint.
  - FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.

In an implementation in which FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented, for context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons on page D2-4718.
  - A successful Context ID match, as described in Breakpoint context comparisons on page D2-4720.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons on page D2-4731.
  - A successful Context ID match, as described in Breakpoint context comparisons on page D2-4720.

The Context ID check is made against the values in both CONTEXTIDR_EL1 and CONTEXTIDR_EL2. The value of DBGBVR^{<n>}_EL1[31:0] is compared with the Context ID value held in CONTEXTIDR_EL1, and the value of DBGBVR^{<n>}_EL1[63:32] is compared with the Context ID value held in CONTEXTIDR_EL2. Both comparisons must match for the Context ID check.

--- Note ---

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

---

\( \text{DBGBCR}^{<n>}_\text{EL1}.\{\text{LBN, BAS}\} \) for this breakpoint are ignored.

--- Note ---

See Reserved DBGBCR^{<n>}_EL1.BT values on page D2-4722 for the behavior of breakpoints programmed with reserved BT values.
D2.9.3 Execution conditions for which a breakpoint generates Breakpoint exceptions

Each breakpoint can be programmed so that it generates Breakpoint exceptions only for certain execution conditions. For example, a breakpoint might be programmed to generate Breakpoint exceptions only when the PE is executing at EL0 in Secure state.

DBGBCR<\n>_EL1.{SSC, HMC, PMC} defines the execution conditions the breakpoint generates Breakpoint exceptions for, as follows:

**Security State Control, SSC**

Controls whether the breakpoint generates Breakpoint exceptions only in Secure state, only in Non-secure state, or in both Security states.

--- Note ---

This is determined by the Security state of the PE, not from the NS attribute returned by the translation of the virtual address on which the breakpoint is set.

---

**Higher Mode Control, HMC, and Privileged Mode Control, PMC**

HMC and PMC together control which Exception levels the breakpoint generates Breakpoint exceptions in.

Table D2-8 on page D2-4717 shows the valid combinations of the values of HMC, SSC, and PMC, and for each combination shows which Exception levels breakpoints generate Breakpoint exceptions in.

In the table:

- **Y** Means that a breakpoint programmed with the values of HMC, SSC, and PMC shown in that row can generate Breakpoint exceptions in that Exception level and Security state.
- **-** Means that a breakpoint programmed with the values of HMC, SSC, and PMC shown in that row cannot generate Breakpoint exceptions in that Exception level and Security state.

A combination in Table D2-8 on page D2-4717 might be reserved if an Exception level or Security state is not implemented. For information about which combinations of HMC, SSC and PMC are reserved if an Exception level or Security state are not implemented, see *Reserved DBGBCR<\n>_EL1.{SSC, HMC, PMC} values* on page D2-4722.

<table>
<thead>
<tr>
<th>HMC</th>
<th>SSC</th>
<th>PMC</th>
<th>Security state</th>
<th>EL3a</th>
<th>EL2</th>
<th>EL1</th>
<th>EL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>01</td>
<td>Both</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>10</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>11</td>
<td></td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
### D2.9 Breakpoint exceptions

#### D2.9.4 Breakpoint instruction address comparisons

In this subsection, the term `AddrTop` represents the most significant bit of a virtual address used by breakpoint data address comparisons. `AddrTop` is:

- 55, if address tagging is used for the address. See *Address tagging on page D8-5162*.
- 63, otherwise.

--- **Note**

When stage 1 translation is enabled, in AArch64 state, a virtual address has a maximum address width of either 48 bits or, when `FEAT_LVA` is implemented and the 64KB translation granularity is used, 52 bits. Software can configure a smaller address width for a virtual address, see *Input address size configuration on page D8-5090*. Attempting to translate an address that is larger than the configured input address size generates a Translation fault.

All combinations of HMC, SSC, and PMC that this table does not show are reserved. See *Reserved DBGBCR<n>_EL1.(SSC, HMC, PMC) values on page D2-4722*.

---

**Table D2-8 Summary of breakpoint HMC, SSC, and PMC encodings (continued)**

<table>
<thead>
<tr>
<th>HMC</th>
<th>SSC</th>
<th>PMC</th>
<th>Security state</th>
<th>EL3a</th>
<th>EL2</th>
<th>EL1</th>
<th>EL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>01</td>
<td>Non-secure</td>
<td>n/a</td>
<td>-</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>10</td>
<td></td>
<td>n/a</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11</td>
<td></td>
<td>n/a</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01</td>
<td>Secure</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
<td></td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>00</td>
<td>Secure</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>01</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>11</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>Both</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>11</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>00</td>
<td>Non-secure</td>
<td>n/a</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>01</td>
<td></td>
<td>n/a</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11</td>
<td></td>
<td>n/a</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>00</td>
<td>Secure</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>01</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>00</td>
<td>Both</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>01</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>11</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

---

a. Debug exceptions are not generated at EL3 using AArch64. This means that these combinations of HMC, SSC, and PMC are relevant only if breakpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSC, and PMC that generate Breakpoint exceptions at EL3 using AArch64.
When stage 1 translation is disabled, using an address that is larger than the implemented PA size generates an Address size fault. The implemented PA size is IMPLEMENTATION DEFINED up to 52 bits, see Implemented physical address size on page D8-5088.

These faults have a higher priority than breakpoints.

An address comparison is successful if bits [AddrTop:2] of the current instruction virtual address are equal to DBGBVR<\text{n}>_EL1[AddrTop:2].

\textbf{Note}

DBGBVR<\text{n}>_EL1 is a 64-bit register. The most significant bits of this register are sign-extension bits. DBGBVR<\text{n}>_EL1[1:0] are RES0 and are ignored.

If EL1 is using AArch64 and EL0 is using AArch32, A32 and T32 instructions can be executed in stage 1 of an AArch64 translation regime. In this case, the instruction addresses are zero-extended before comparison with the breakpoint.

\textbf{Specifying the halfword-aligned address that an Address breakpoint matches on}

For Address Match breakpoints, if the implementation supports AArch32 state, a debugger must program the \textit{Byte Address Selection} field, DBGBCR<\text{n}>_EL1.BAS.

<table>
<thead>
<tr>
<th>BAS</th>
<th>Match instruction at</th>
<th>Constraint for debuggers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0011</td>
<td>DBGBCR&lt;\text{n}&gt;_EL1</td>
<td>Use for T32 instructions.</td>
</tr>
<tr>
<td>0b1100</td>
<td>DBGBCR&lt;\text{n}&gt;_EL1+2</td>
<td>Use for T32 instructions.</td>
</tr>
<tr>
<td>0b1111</td>
<td>DBGBCR&lt;\text{n}&gt;_EL1</td>
<td>Use for A64 and A32 instructions.</td>
</tr>
</tbody>
</table>

If the implementation is an AArch64-only implementation, all instructions are word-aligned and DBGBCR<\text{n}>_EL1.BAS is RES1.

Figure D2-2 on page D2-4720 shows a summary of when Address Match breakpoints programmed with particular BAS values generate Breakpoint exceptions. The figure contains four parts:

- A column showing the row number, on the left.
- An instruction set and instruction size table.
- A location of instruction figure.
- A BAS field values table, on the right.

To use the figure, read across the rows. For example, row 7 shows that a breakpoint with DBGBCR<\text{n}>_EL1.BAS programmed as either 0b0011 or 0b1111 generates Breakpoint exceptions for A64 instructions. A64 instructions are always at word-aligned addresses.

\textbf{Note}

To breakpoint on an A64 instruction, Arm recommends that the debugger programs DBGBCR<\text{n}>_EL1.BAS as 0b1111.
In the figure:

Yes Means that the breakpoint generates a Breakpoint exception.
No Means that the breakpoint does not generate a Breakpoint exception.
UNP Means that it CONstrained UNPREDICTable whether the breakpoint generates a Breakpoint exception. See Other usage constraints for Address breakpoints on page D2-4724.

<table>
<thead>
<tr>
<th>Instruction set</th>
<th>Size</th>
<th>Location of instruction</th>
<th>BAS[3:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>T32</td>
<td>16-bit</td>
<td>-2 -1 0 +1 +2 +3 +4 +5</td>
<td>Yes No Yes</td>
</tr>
<tr>
<td>T32</td>
<td>32-bit</td>
<td>-2 -1 0 +1 +2 +3 +4 +5</td>
<td>UNP No UNP</td>
</tr>
<tr>
<td>T32</td>
<td>32-bit</td>
<td>-2 -1 0 +1 +2 +3 +4 +5</td>
<td>Yes UNP Yes</td>
</tr>
<tr>
<td>T32</td>
<td>32-bit</td>
<td>-2 -1 0 +1 +2 +3 +4 +5</td>
<td>No Yes UNP</td>
</tr>
<tr>
<td>A32</td>
<td>32-bit</td>
<td>-2 -1 0 +1 +2 +3 +4 +5</td>
<td>Yes UNP Yes</td>
</tr>
<tr>
<td>A64</td>
<td>32-bit</td>
<td>-2 -1 0 +1 +2 +3 +4 +5</td>
<td>Yes UNP Yes</td>
</tr>
</tbody>
</table>

a. 0 means the word-aligned address held in the DBGBVR<n>_EL1[maxAddressSize:2]:00.
The other locations are as follows:
- -2 means ((DBGBVR<n>_EL1[maxAddressSize:2]:00) –
- -1 means ((DBGBVR<n>_EL1[maxAddressSize:2]:00) –
- ... ...
- +5 means ((DBGBVR<n>_EL1[maxAddressSize:2]:00) + 5).
The solid areas show the location of the instruction.

Figure D2-2 Summary of BAS field meanings for Address Match breakpoints

D2.9.5 Breakpoint context comparisons

The breakpoint type defined by DBGBCR<n>.BT determines what context comparison is required, if any. Table D2-10 on page D2-4720 shows the BT values that require a comparison, and the match required for the comparison to be successful.

Table D2-10 Breakpoint Context ID and VMID comparison tests

<table>
<thead>
<tr>
<th>DBGBCR&lt;n&gt;.BT</th>
<th>Test required for successful context comparison</th>
</tr>
</thead>
</table>
| 0b001x       | • When FEAT_VHE is implemented, EL2 is using AArch64, the Effective value of HCR_EL2.E2H is 1, and either the PE is executing at EL0 with HCR_EL2.TGE set to 1, or the PE is executing at EL2, CONTEXTIDR_EL2 must match the DBGBVR<n>_EL1.ContextID value.  
• Otherwise, CONTEXTIDR_EL1 must match the DBGBVR<n>_EL1.ContextID value. |
| 0b011x       | CONTEXTIDR_EL1 must match the DBGBVR<n>_EL1.ContextID value. |
| 0b100x       | VTTBR_EL2.VMID must match the DBGBVR<n>_EL1.VMID value. |
| 0b101x       | CONTEXTIDR_EL1 must match the DBGBVR<n>_EL1.ContextID value and VTTBR_EL2.VMID must match the DBGBVR<n>_EL1.VMID value. |
| 0b110x       | CONTEXTIDR_EL2 must match the DBGBVR<n>_EL1.ContextID2 value, DBGBVR<n>_EL1[63:32]. |
| 0b111x       | Both:  
• CONTEXTIDR_EL1 must match the DBGBVR<n>_EL1.ContextID value, DBGBVR<n>_EL1[31:0].  
• CONTEXTIDR_EL2 must match the DBGBVR<n>_EL1.ContextID2 value, DBGBVR<n>_EL1[63:32]. |

No Context ID or VMID comparison is required for other valid DBGBCR<n>.BT values.
Context breakpoints do not generate Breakpoint exceptions when any of:

- The comparison uses the value of CONTEXTIDR_EL1 and any of:
  - The PE is executing at EL3 using AArch64.
  - The PE is executing at EL2.
  - FEAT_VHE is implemented, EL2 is using AArch64, EL2 is enabled in the current Security state, and HCR_EL2.{E2H, TGE} == {1, 1}.
- The comparison uses the value of CONTEXTIDR_EL2 and any of:
  - Neither FEAT_VHE is implemented, nor FEAT_Debugv8p2 is implemented.
  - If the PE is in Secure state, and either FEAT_SEL2 is not implemented, or Secure EL2 is disabled.
  - The PE is executing at EL3.
  - EL2 is using AArch32.
  - EL2 is not implemented.
- The comparison uses the current VMID value and any of:
  - EL2 is not implemented.
  - If the PE is in Secure state, and either FEAT_SEL2 is not implemented, or Secure EL2 is disabled.
  - The PE is executing at EL2.
  - The PE is executing at EL3.
  - FEAT_VHE is implemented, EL2 is using AArch64, EL2 is enabled in the current Security state, and HCR_EL2.{E2H, TGE} == {1, 1}.

**Note**
- For all Context breakpoints, DBGBCR<n>_EL1.BAS is RES1 and is ignored.
- For Linked Context breakpoints, DBGBCR<n>_EL1.{LBN, SSC, HMC, PMC} are RES0 and are ignored.

### D2.9.6 Breakpoint usage constraints

See the following sections:

- *Reserved DBGBCR<n>_EL1.BT values* on page D2-4722.
- *Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values* on page D2-4722.
- *Reserved DBGBCR<n>_EL1.BAS values* on page D2-4723.
- *Reserved DBGBCR<n>_EL1.LBN values* on page D2-4724.
- *Other usage constraints for Address breakpoints* on page D2-4724.
- *Other usage constraints for Context breakpoints* on page D2-4724.
Reserved DBGBCR<\text{n}>_EL1.BT values

Table D2-11 on page D2-4722 shows when particular DBGBCR<\text{n}>_EL1.BT values are reserved.

<table>
<thead>
<tr>
<th>BT value</th>
<th>Breakpoint type</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b001x</td>
<td>Context ID Match</td>
<td>If the breakpoint is not context-aware</td>
</tr>
<tr>
<td>0b010x</td>
<td>Address Mismatch</td>
<td>In stage 1 of an AArch64 translation regime, or if EDSCR.HDE is 1 and halting is allowed</td>
</tr>
<tr>
<td>0b011x</td>
<td>CONTEXTIDR_EL1 Match</td>
<td>If FEAT_VHE is not implemented, or the breakpoint is not context-aware</td>
</tr>
<tr>
<td>0b100x</td>
<td>VMID Match</td>
<td>If EL2 is not implemented, or the breakpoint is not context-aware</td>
</tr>
<tr>
<td>0b101x</td>
<td>Context ID and VMID Match</td>
<td>If EL2 is not implemented, or the breakpoint is not context-aware</td>
</tr>
<tr>
<td>0b110x</td>
<td>CONTEXTIDR_EL2 Match</td>
<td>If FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, or if the breakpoint is not context-aware</td>
</tr>
<tr>
<td>0b111x</td>
<td>Full Context ID Match</td>
<td></td>
</tr>
</tbody>
</table>

If a breakpoint is programmed with one of these reserved BT values:

- The breakpoint must behave as if it is either:
  - Disabled.
  - Programmed with a BT value that is not reserved, other than for a direct or external read of DBGBCR<\text{n}>_EL1.

- For a direct or external read of DBGBCR<\text{n}>_EL1, if the reserved BT value:
  - Has no function for any execution conditions, the value read back is UNKNOWN.
  - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the BT value so that the breakpoint functions for the other execution conditions.

The behavior of breakpoints with reserved BT values might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

Reserved DBGBCR<\text{n}>_EL1.{SSC, HMC, PMC} values

Table D2-12 on page D2-4722 shows when particular combinations of DBGBCR<\text{n}>_EL1.{SSC, HMC, PMC} are reserved in stage 1 of an AArch64 translation regime.

<table>
<thead>
<tr>
<th>HMC, SSC, and PMC combination</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>All combinations with SSC set to 0b01 or 0b10, except for the combination with HMC set to 1, SSC set to 0b01, and PMC set to 0b00.</td>
<td>When EL3 is not implemented and EL2 is implemented.</td>
</tr>
<tr>
<td>Any combination where HMC or SSC is nonzero, except for the combination with HMC set to 1, SSC set to 0b01, and PMC set to 0b00, or combinations when SSC is set to 0b11.</td>
<td>When both of EL2 and EL3 are not implemented.</td>
</tr>
<tr>
<td>The combination with HMC set to 1, SSC set to 0b11, and PMC set to 0b00.</td>
<td>When EL2 is not implemented.</td>
</tr>
</tbody>
</table>
For all breakpoints except Linked Context breakpoints, if a breakpoint is programmed with one of these reserved combinations:

- If the reserved combination has a function for other execution conditions:
  - The breakpoint must behave as if it is disabled.
  - A direct or external read of DBGBCR<\texttt{n}>_EL1.{SSC, HMC, PMC} returns the values written. This means that software can save and restore the combination so that the breakpoint can function for the other execution conditions.

- If the reserved combination does not have a function for other execution conditions:
  - It must behave either as if it is programmed with a combination that is not reserved or as if it is disabled.
  - A direct or external read of DBGBCR<\texttt{n}>_EL1.{SSC, HMC, PMC} returns UNKNOWN values.

If the breakpoint is a Linked Context breakpoint, then:
- The values of HMC, SSC, and PMC are ignored.
- A direct or external read of DBGBCR<\texttt{n}>_EL1.{SSC, HMC, PMC} returns UNKNOWN values.

The behavior of breakpoints with reserved combinations of HMC, SSC, and PMC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

### Reserved DBGBCR<\texttt{n}>_EL1.BAS values

In an AArch64-only implementation, DBGBCR<\texttt{n}>_EL1.BAS for all breakpoints is RES1.

Otherwise:

**For all Context breakpoints**

DBGBCR<\texttt{n}>_EL1.BAS is RES1 and is ignored.

**For all Address breakpoints**

Table D2-9 on page D2-4719 gives the valid values of the DBGBCR<\texttt{n}>_EL1.BAS field.

If a breakpoint is programmed with a reserved BAS value:

- The breakpoint must behave as if it is either:
  - Disabled.
  - Programmed with a BAS value that is not reserved, other than for a direct or external read of DBGBCR<\texttt{n}>_EL1.

- A direct or external read of DBGBCR<\texttt{n}>_EL1.BAS returns an UNKNOWN value.

Software must not rely on these properties as the behavior of reserved values might change in a future revision of the architecture.
Reserved DBGBCR<n>_EL1.LBN values

For all Context breakpoints

DBGBCR<n>_EL1.LBN reads UNKNOWN and its value is ignored.

For Linked Address breakpoints

A Linked Address breakpoint must link to a context-aware breakpoint. For a Linked Address breakpoint, any DBGBCR<n>_EL1.LBN value that is not for a context-aware breakpoint is reserved.

If a Linked Address breakpoint links to a breakpoint that is not implemented, or that is not context-aware, then reads of DBGBCR<n>_EL1.LBN return an unknown value and behavior is CONSTRAINED UNPREDICTABLE. The Linked Address breakpoint behaves as if it is either:

- Disabled.
- Linked to an UNKNOWN context-aware breakpoint.

If a Linked Address breakpoint links to a breakpoint that is implemented and that is context-aware, but that is either not enabled or not programmed as a Linked Context breakpoint, it behaves as if it is disabled.

For Unlinked Address breakpoints

DBGBCR<n>_EL1.LBN reads UNKNOWN and its value is ignored.

Other usage constraints for Address breakpoints

For all Address breakpoints

- DBGBVR<n>_EL1[1:0] are RES0 and are ignored.
- If the implementation supports AArch32 state:
  - For 32-bit instructions, if a breakpoint matches on the address of the second halfword but not the address of the first halfword, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception.
  - If DBGBCR<n>.BAS is 0b1111, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception for a T32 instruction starting at address ((DBGBVR<n>[48:2]:00) + 2). For T32 instructions, Arm recommends that the debugger programs the BAS field with either 0b0011 or 0b1100.

Other usage constraints for Context breakpoints

For all Context breakpoints

Any bits of DBGBVR<n>_EL1 that are not used to specify Context ID or VMID are RES0 and are ignored.

For Linked Context breakpoints

If no Linked Address breakpoints or Linked watchpoints link to a Linked Context breakpoint, the Linked Context breakpoint does not generate any Breakpoint exceptions.

D2.9.7 Preferred return address

The preferred return address of a Breakpoint exception is the address of the instruction that was not executed because the PE took the Breakpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

D2.9.8 Pseudocode description of Breakpoint exceptions taken from AArch64 state

AArch64.BreakpointValueMatch() tests the value in DBGBVR<n>_EL1.
AArch64.StateMatch() tests the values in DBGBCR<n>_EL1.\{SSC, HMC, PMC\} and, if the breakpoint links to a Linked Context breakpoint, also tests the Linked Context breakpoint.

For a watchpoint, AArch64.StateMatch() tests the values in DBGWCR<n>_EL1.\{SSC, HMC, PAC\} and, if the watchpoint links to a Linked Context breakpoint, also tests the Linked Context breakpoint.

AArch64.BreakpointMatch() tests a committed instruction against all breakpoints.

AArch64.CheckBreakpoint() generates a Breakpoint exception if all of the following are true:

- MDSCR_EL1.MDE is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See Enabling debug exceptions from the current Exception level on page D2-4699.
- All of the conditions required for Breakpoint exception generation are met. See About Breakpoint exceptions on page D2-4707.

Note

AArch64.CheckBreakpoint() might halt the PE and cause it to enter Debug state. External debug uses Debug state.

AArch64.BreakpointException() is called to generate a Breakpoint exception.

These functions are defined in Chapter J1 Armv8 Pseudocode.
**D2.10 Watchpoint exceptions**

This section describes Watchpoint exceptions in stage 1 of an AArch64 translation regime.

The PE is using an AArch64 translation regime when it is executing either:

- In an Exception level that is using AArch64.
- At EL0 using AArch32 when EL1 is using AArch64.

This section contains the following subsections:

- About Watchpoint exceptions.
- Watchpoint types and linking of watchpoints on page D2-4727.
- Execution conditions for which a watchpoint generates Watchpoint exceptions on page D2-4728.
- Watchpoint data address comparisons on page D2-4731.
- Determining the memory location that caused a Watchpoint exception on page D2-4734.
- Watchpoint behavior on other instructions on page D2-4735.
- SVE Watchpoint exceptions on page D2-4737.
- Watchpoint usage constraints on page D2-4737.
- Exception syndrome information and preferred return address on page D2-4739.
- Pseudocode description of Watchpoint exceptions taken from AArch64 state on page D2-4740.

**D2.10.1 About Watchpoint exceptions**

A watchpoint is an event that results from the execution of an instruction, based on a data address. Watchpoints are also known as data breakpoints.

A watchpoint operates as follows:

1. A debugger programs the watchpoint with a data address, or a data address range.
2. The watchpoint generates a Watchpoint debug event on an access to the address, or any address in the address range.

A watchpoint never generates a Watchpoint debug event on an instruction fetch.

An implementation can include between 2-16 watchpoints. In an implementation, ID_AA64DFR0_EL1.WRP shows how many are implemented.

To use an implemented watchpoint, a debugger programs the following registers for the watchpoint:

- The Watchpoint Control Register, DBGWCR<n>_EL1. This contains controls for the watchpoint, for example an enable control.
- The Watchpoint Value Register, DBGWVR<n>_EL1. This holds the data virtual address used for watchpoint matching.

These registers are numbered, so that:

- DBGWCR0_EL1 and DBGWVR0_EL1 are for watchpoint number zero.
- DBGWCR1_EL1 and DBGWVR1_EL1 are for watchpoint number one.
- DBGWCR2_EL1 and DBGWVR2_EL1 are for watchpoint number two.
- ...
- ...
- DBGWCR<n-1>_EL1 and DBGWVR<n-1>_EL1 are for watchpoint number (n-1).

A watchpoint can:

- Be programmed to generate Watchpoint debug events on read accesses only, on write accesses only, or on both types of access.
- Link to a Linked Context breakpoint, so that a Watchpoint debug event is generated only if the PE is in a particular context when the address match occurs.
A single watchpoint can be programmed to match on one or more address bytes. A watchpoint generates a Watchpoint debug event on an access to any byte that it is watching. The number of bytes a watchpoint is watching is either:

- One to eight bytes, provided that these bytes are contiguous and that they are all in the same naturally-aligned doubleword. A debugger uses the Byte Address Select field, DBGWCR<n>_EL1.BAS, to select the bytes. See Programming a watchpoint with eight bytes or fewer on page D2-4732.

- Eight bytes to 2GB, provided that both of the following are true:
  - The number of bytes is a power-of-two.
  - The range starts at an address that is aligned to the range size.

A debugger uses the MASK field, DBGWCR<n>_EL1.MASK, to program a watchpoint with eight bytes to 2GB. See Programming a watchpoint with eight or more bytes on page D2-4733.

A debugger must use either the BAS field or the MASK field. If it uses both, whether the watchpoint generates Watchpoint debug events is CONSTRAINED UNPREDICTABLE. See Programming dependencies of the BAS and MASK fields on page D2-4738.

For each memory access, all of the watchpoints are tested. When a watchpoint is tested, it generates a Watchpoint debug event if all of the following are true:

- The watchpoint is enabled. That is, the watchpoint enable control for it, DBGWCR<n>_EL1.E, is 1.
- The conditions specified in the DBGWCR<n>_EL1 are met.
- The comparison with the address held in the DBGWVR<n>_EL1 is successful.
- If the watchpoint links to a Linked Context breakpoint, the comparison or comparisons made by the Linked Context breakpoint also are successful. See Figure D2-1 on page D2-4710. See also Breakpoint context comparisons on page D2-4720.
- The instruction that initiates the memory access is committed for execution.
- The instruction that initiates the memory access passes its Condition code check.
- If the access is due to a System register access instruction executed at EL1 and transformed into a memory access by the mechanism described in Enhanced support for nested virtualization on page D8-5175 and one of the following is true:
  - EDSCR.HDE is set to 1 and halting is allowed.
  - Debug exceptions are enabled at EL2.

If halting is allowed and EDSCR.HDE is 1, Watchpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are:
- Enabled, Watchpoint debug events generate Watchpoint exceptions.
- Disabled, Watchpoint debug events are ignored.

--- Note ---

The remainder of this Watchpoint Exceptions section, including all subsections, describes watchpoints as generating Watchpoint exceptions.

However, the behavior described also applies if watchpoints are causing entry to Debug state.

---

**D2.10.2 Watchpoint types and linking of watchpoints**

When a debugger programs a watchpoint, it must program that watchpoint so that it is either:

- Used in isolation. In this case, the watchpoint is called an *Unlinked watchpoint*.
- Enabled for linking to a Linked Context breakpoint. In this case, the watchpoint is called a *Linked watchpoint*.
When a Linked watchpoint links to a Linked Context breakpoint, the Linked watchpoint generates a Watchpoint exception only if the PE is in a particular context when the data address match occurs. For example, a debugger might:

1. Program watchpoint number one with a data address.
2. Program breakpoint number five to be a Linked VMID Match breakpoint.
3. Link the watchpoint and the breakpoint together. A Watchpoint exception is generated only if both the data address matches and the VMID matches.

The Watchpoint Type field for a watchpoint, DBGWCR<\text{\textless}n\text{\textgreater}>_EL1.WT, controls whether the watchpoint is enabled for linking. If DBGWCR<\text{\textless}n\text{\textgreater}>_EL1.WT is 1, the watchpoint is enabled for linking.

**Rules for linking watchpoints**

The rules for watchpoint linking are as follows:

- Only Linked watchpoints can be linked.
- A Linked watchpoint can link to any type of Linked Context breakpoint. The Linked Breakpoint Number field, DBGWCR<\text{\textless}n\text{\textgreater}>_EL1.LBN, for the Linked watchpoint specifies the particular Linked Context breakpoint that the Linked watchpoint links to, and:
  - DBGWCR<\text{\textless}n\text{\textgreater}>_EL1.{SSC, HMC, PAC} for the Linked watchpoint defines the execution conditions that the watchpoint generates Watchpoint exceptions for. See *Execution conditions for which a watchpoint generates Watchpoint exceptions*.
  - DBGBCR<\text{\textless}n\text{\textgreater}>_EL1.{SSC, HMC, PMC} for the Linked Context breakpoint are ignored.
- A Linked watchpoint cannot link to another watchpoint. The LBN field can therefore only specify a breakpoint.
- If a Linked watchpoint links to a breakpoint that is not context-aware, the behavior of the Linked watchpoint is CONSTRAINED UNPREDICTABLE. See *Watchpoint usage constraints* on page D2-4737.
- If a Linked watchpoint links to an Unlinked Context breakpoint, the Linked watchpoint never generates any Watchpoint exceptions.
- If the access is due to a System register access instruction executed at EL1 and transformed into a memory access by the mechanism described in *Enhanced support for nested virtualization* on page D8-5175, and the watchpoint is linked to a context-aware breakpoint that is programmed to match the value held in CONTEXTIDR_EL1, then it is CONSTRAINED UNPREDICTABLE whether there is a watchpoint match.
- Multiple Linked watchpoints can link to a single Linked Context breakpoint.

---

**Note**

Multiple Address breakpoints can also link to a single Linked Context breakpoint. *Breakpoint exceptions* on page D2-4707 describes breakpoints.

---

Figure D2-1 on page D2-4710 shows an example of permitted watchpoint linking.

### D2.10.3 Execution conditions for which a watchpoint generates Watchpoint exceptions

Each watchpoint can be programmed so that it generates Watchpoint exceptions only for certain execution conditions. For example, a watchpoint might be programmed to generate Watchpoint exceptions only for Non-secure EL2 accesses.
DBGWCR<\textsubscript{\textasciitilde}>_EL1.\{SSC, HMC, PAC\} define the execution conditions a watchpoint generates Watchpoint exceptions for, as follows:

**Security State Control, SSC**

Controls whether the watchpoint generates Watchpoint exceptions only in Secure state, only in Non-secure state, or in both Security states.

--- Note ---
This is determined by the Security state of the PE, not from the NS attribute returned by the translation of the virtual address on which the watchpoint is set.

**Higher Mode Control, HMC, and Privileged Access Control, PAC**

HMC and PAC together control which Exception levels the watchpoint generates Watchpoint exceptions in.

The PAC control relates to the privilege of the memory access, not to the Exception level at which the access was made:

- Load unprivileged or Store unprivileged instructions executed at EL1, or executed at EL2 when HCR\_EL2.E2H is 1, are treated as EL0 accesses.
- System register accesses executed at EL1 and transformed into a memory access by the mechanism described in Enhanced support for nested virtualization on page D8-5175 are treated as EL2 accesses.

--- Note ---
This means that, if the PE executes a Load unprivileged or Store unprivileged instruction at EL1, the resulting data access triggers a watchpoint only if both:

- PAC is programmed to a value that generates watchpoints on EL0 accesses.
- All other conditions for generating the watchpoint are met.

Example A64 Load unprivileged and Store unprivileged instructions are LDTR and STTR.

Table D2-13 on page D2-4730 shows the valid combinations of HMC, SSC, and PAC, and for each combination shows which Exception levels watchpoints generate Watchpoint exceptions in.

In the table:

**Y or -**

Means that a watchpoint programmed with the values of HMC, SSC, and PAC shown in that row:

- **Y** Can generate Watchpoint exceptions in that Exception level and Security state.
- **-** Cannot generate Watchpoint exceptions in that Exception level and Security state.

For information about which combinations of HMC, SSC and PMC are reserved if an Exception level or Security state are not implemented or enabled, see Reserved DBGWCR<\textsubscript{\textasciitilde}>_EL1.\{SSC, HMC, PAC\} values on page D2-4737.
Table D2-13 Summary of watchpoint HMC, SSC, and PAC encodings

<table>
<thead>
<tr>
<th>HMC</th>
<th>SSC</th>
<th>PAC</th>
<th>Security state</th>
<th>EL3&lt;sup&gt;a&lt;/sup&gt;</th>
<th>EL2</th>
<th>EL1</th>
<th>EL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>01</td>
<td>Both</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>10</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>11</td>
<td>Non-secure</td>
<td>n/a</td>
<td>-</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01</td>
<td>Non-secure</td>
<td>n/a</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>10</td>
<td></td>
<td>n/a</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11</td>
<td></td>
<td>n/a</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01</td>
<td>Secure</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
<td></td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>00</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>01</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>11</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>Both</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>11</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>00</td>
<td>Non-secure</td>
<td>n/a</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>01</td>
<td></td>
<td>n/a</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11</td>
<td></td>
<td>n/a</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>00</td>
<td>Secure</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>01</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>00</td>
<td>Both</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>01</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>11</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

<sup>a</sup> Debug exceptions are not generated at EL3 using AArch64. This means that these combinations of HMC, SSC, and PAC are relevant only if watchpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSC, and PMC that generate Watchpoint exceptions at EL3 using AArch64.

All combinations of HMC, SSC, and PAC that this table does not show are reserved. See *Reserved DBGWCR<sub>n</sub> EL1 {SSC, HMC, PAC} values* on page D2-4737.
D2.10.4 Watchpoint data address comparisons

In this subsection, the term \textit{AddrTop} represents the most significant bit of a virtual address used by watchpoint data address comparisons. \textit{AddrTop} is:

\begin{itemize}
  \item 55, if address tagging is used for the address. See \textit{Address tagging} on page D8-5162.
  \item 63, otherwise.
\end{itemize}

\begin{note}
When stage 1 translation is enabled, in AArch64 state, a virtual address has a maximum address width of either 48 bits or, when \textit{FEAT_LVA} is implemented and the 64KB translation granule is used, 52 bits. Software can configure a smaller address width for a virtual address. See \textit{Input address size configuration} on page D8-5090. Attempting to translate an address that is larger than the configured input address size generates a Translation fault.

When stage 1 translation is disabled, using an address that is larger than the implemented PA size generates an Address size fault. The implemented PA size is \textit{IMPLEMENTATION DEFINED} up to 52 bits. See \textit{Implemented physical address size} on page D8-5088.

These faults have a higher priority than watchpoints.
\end{note}

An address comparison is successful if bits \([\textit{AddrTop}:2]\) of the current data address are equal to \texttt{DBGWVR}_\texttt{n}\_EL1[\textit{AddrTop}:2], taking into account all of the following:

\begin{itemize}
  \item The size of the access. See \textit{Size of the data access}.
  \item If EL1 is using AArch64 and EL0 is using AArch32, AArch32 instructions can be executed in stage 1 of an AArch64 translation regime. In this case, data addresses are zero-extended before comparison with the watchpoint.
  \item The bytes selected by \texttt{DBGWVR}_\texttt{n}\_EL1.BAS. See \textit{Programming a watchpoint with eight bytes or fewer} on page D2-4732.
  \item Any address ranges indicated by \texttt{DBGWVR}_\texttt{n}\_EL1.MASK. See \textit{Programming a watchpoint with eight or more bytes} on page D2-4733.
\end{itemize}

\begin{note}
\begin{itemize}
  \item \texttt{DBGWVR}_\texttt{n}\_EL1 is a 64-bit register. The most significant bits of this register are sign-extension bits.
  \item \texttt{DBGWVR}_\texttt{n}\_EL1[1:0] are \texttt{RES0} and are ignored.
\end{itemize}
\end{note}

Size of the data access

Because watchpoints can be programmed to generate Watchpoint exceptions on individual bytes, the size of each data access must be taken into account. See \textit{Example D2-1}.

\begin{example}
1. A debugger programs a watchpoint to generate Watchpoint exceptions only when the byte at address \texttt{0x1009} is accessed.
2. The PE accesses the unaligned doubleword starting at address \texttt{0x1003}.

In this scenario, the watchpoint must generate a Watchpoint exception.
\end{example}

The size of data accesses initiated by \texttt{DC ZVA} instructions is the DC ZVA block size that \texttt{DCZID\_EL0.BS} defines.

The size of data accesses initiated by \texttt{DC IVAC} instructions is an \textit{IMPLEMENTATION DEFINED} size that is both:

\begin{itemize}
  \item From the inclusive range between:
    \begin{itemize}
      \item The size that \texttt{CTR\_EL0.DminLine} defines.
      \item 2KB.
    \end{itemize}
\end{itemize}
• A power-of-two.

For both of these instructions:

• The lowest address accessed by the instruction is the address supplied to the instruction, rounded down to the nearest multiple of the access size initiated by that instruction.

• The highest address accessed is (size - 1) bytes above the lowest address accessed.

See also, Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA, DC GVA, and DC GZVA instructions on page D2-4736.

Programming a watchpoint with eight bytes or fewer

The Byte Address Select field, DBGWCR<n>_EL1.BAS, selects which bytes in the doubleword starting at the address contained in the DBGWVR<n>_EL1 the watchpoint generates Watchpoint exceptions for.

If the address programmed into the DBGWVR<n>_EL1 is:

• Doubleword-aligned:
  — All eight bits of DBGWCR<n>_EL1.BAS are used, and the descriptions given in Table D2-14 apply.

• Word-aligned but not doubleword-aligned:
  — Only DBGWCR<n>_EL1.BAS[3:0] are used, and the descriptions given in Table D2-15 on page D2-4733 apply. In this case, DBGWCR<n>_EL1.BAS[7:4] are RES0.

### Table D2-14 Supported BAS values when the DBGWVRn_EL1 address alignment is doubleword

<table>
<thead>
<tr>
<th>BAS value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000</td>
<td>Watchpoint never generates a Watchpoint exception.</td>
</tr>
<tr>
<td>BAS[0] == 1</td>
<td>Generates a Watchpoint exception if the byte at address DBGWVR&lt;n&gt;_EL1[AddrTop:3]:000 is accessed.</td>
</tr>
<tr>
<td>BAS[1] == 1</td>
<td>Generates a Watchpoint exception if the byte at address DBGWVR&lt;n&gt;_EL1[AddrTop:3]:001 is accessed.</td>
</tr>
<tr>
<td>BAS[2] == 1</td>
<td>Generates a Watchpoint exception if the byte at address DBGWVR&lt;n&gt;_EL1[AddrTop:3]:010 is accessed.</td>
</tr>
<tr>
<td>BAS[3] == 1</td>
<td>Generates a Watchpoint exception if the byte at address DBGWVR&lt;n&gt;_EL1[AddrTop:3]:011 is accessed.</td>
</tr>
<tr>
<td>BAS[4] == 1</td>
<td>Generates a Watchpoint exception if the byte at address DBGWVR&lt;n&gt;_EL1[AddrTop:3]:100 is accessed.</td>
</tr>
<tr>
<td>BAS[5] == 1</td>
<td>Generates a Watchpoint exception if the byte at address DBGWVR&lt;n&gt;_EL1[AddrTop:3]:101 is accessed.</td>
</tr>
<tr>
<td>BAS[6] == 1</td>
<td>Generates a Watchpoint exception if the byte at address DBGWVR&lt;n&gt;_EL1[AddrTop:3]:110 is accessed.</td>
</tr>
<tr>
<td>BAS[7] == 1</td>
<td>Generates a Watchpoint exception if the byte at address DBGWVR&lt;n&gt;_EL1[AddrTop:3]:111 is accessed.</td>
</tr>
</tbody>
</table>
If the BAS field is programmed with more than one byte, the bytes that it is programmed with must be contiguous. For watchpoint behavior when its BAS field is programmed with non-contiguous bytes, see Other usage constraints on page D2-4739.

When programming the BAS field with anything other than 0b11111111, a debugger must program DBGWCR<\n>_EL1.MASK to be 0b00000. See Programming dependencies of the BAS and MASK fields on page D2-4738.

A watchpoint generates a Watchpoint exception whenever a watched byte is accessed, even if:

- The access size is smaller or larger than the address region being watched.
- The access is misaligned, and the base address of the access is not in the doubleword or word of memory addressed by the DBGWVR<\n>_EL1[AddrTop:3]. See Example D2-1 on page D2-4731.

The following are some example configurations of the BAS field:

- To program a watchpoint to generate a Watchpoint exception on the byte at address 0x1003, program:
  - DBGWVR<\n>_EL1 with 0x1000.
  - DBGWCR<\n>_EL1.BAS to be 0b00001000.

- To program a watchpoint to generate a Watchpoint exception on the bytes at addresses 0x2003, 0x2004 and 0x2005, program:
  - DBGWVR<\n>_EL1 with 0x2000.
  - DBGWCR<\n>_EL1.BAS to be 0b00111000.

- If the address programmed into the DBGWVR<\n>_EL1 is doubleword-aligned:
  - To generate a Watchpoint exception when any byte in the word starting at the doubleword-aligned address is accessed, program DBGWCR<\n>_EL1.BAS to be 0b00001011.
  - To generate a Watchpoint exception when any byte in the word starting at address DBGWVR<\n>_EL1[31:3]:100 is accessed, program DBGWCR<\n>_EL1.BAS to be 0b11110000.

Note

Arm deprecates programming a DBGWVR<\n>_EL1 with an address that is not doubleword-aligned.

### Programming a watchpoint with eight or more bytes

A debugger can use the MASK field, DBGWCR<\n>_EL1.MASK, to program a single watchpoint with a data address range. The range must meet all of the following criteria:

- It is a size that is:
  - A power-of-two.
— A minimum of eight bytes.
— A maximum of 2GB.

- It starts at an address that is aligned to the size.

The MASK field specifies the number of least significant data address bits that must be masked. Up to 31 least significant bits can be masked:

<table>
<thead>
<tr>
<th>MASK</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00000</td>
<td>No bits are masked.</td>
</tr>
<tr>
<td>0b00001</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0b00010</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0b00011</td>
<td>Three least significant bits are masked.</td>
</tr>
<tr>
<td>0b00100</td>
<td>Four least significant bits are masked.</td>
</tr>
<tr>
<td>0b00101</td>
<td>Five least significant bits are masked.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0b11111</td>
<td>31 least significant bits are masked.</td>
</tr>
</tbody>
</table>

If \( n \) least significant address bits are masked, the watchpoint generates a Watchpoint exception on all of the following:

- Address \( \text{DBGWVR}^{n}\_\text{EL1}[:\text{AddrTop}:n]:000… \)
- Address \( \text{DBGWVR}^{n}\_\text{EL1}[:\text{AddrTop}:n]:111… \)
- Any address between these two addresses.

For example, if the four least significant address bits are masked, Watchpoint exceptions are generated for all addresses between \( \text{DBGWVR}^{n}\_\text{EL1}[:\text{AddrTop}:4]:0000 \) and \( \text{DBGWVR}^{n}\_\text{EL1}[:\text{AddrTop}:4]:1111 \), including these addresses.

--- Note ---

- The 17 most significant bits cannot be masked. This means that the full address cannot be masked.
- For watchpoint behavior when its MASK field is programmed with a reserved value, see Reserved \( \text{DBGWCR}^{n}\_\text{EL1}.\text{MASK} \) values on page D2-4739.

---

When masking address bits, a debugger must both:

- Program \( \text{DBGWCR}^{n}\_\text{EL1}.\text{BAS} \) to be \( 0b11111111 \). See Programming dependencies of the BAS and MASK fields on page D2-4738.
- In the \( \text{DBGWVR}^{n}\_\text{EL1} \), set the masked address bits to 0. For watchpoint behavior when any of the masked address bits are not 0, see Other usage constraints on page D2-4739.

### D2.10.5 Determining the memory location that caused a Watchpoint exception

On taking a Watchpoint exception, the PE records an address in a Fault Address Register that the debugger can use to determine the memory location that triggered the watchpoint.

The Fault Address Register (FAR) used is either:

- \( \text{FAR\_EL1} \), if the exception is taken to EL1.
- \( \text{FAR\_EL2} \), if the exception is taken to EL2.

In cases where one instruction triggers multiple watchpoints, only one address is recorded.

On entering Debug state on a Watchpoint debug event, the PE records the address in the EDWAR.

For more information, see the subsections that follow. These are:

- Address recorded for Watchpoint exceptions generated by Memory Copy and Memory Set instructions on page D2-4735.
- Address recorded for Watchpoint exceptions generated by data cache maintenance instructions on page D2-4735.
- Address recorded for Watchpoint exceptions generated by other instructions on page D2-4735.
Address recorded for Watchpoint exceptions generated by Memory Copy and Memory Set instructions

The address recorded is within an address range of the size defined by the DCZID_EL0.BS field. The start of the range is aligned to the size defined by the DCZID_EL0.BS field and its end is not greater than the address that triggered the watchpoint.

Address recorded for Watchpoint exceptions generated by data cache maintenance instructions

The address recorded is the address passed to the instruction. This means that the address recorded might be higher than the address of the location that triggered the watchpoint.

Address recorded for Watchpoint exceptions generated by other instructions

Note

Despite their mnemonics, the DC ZVA, DC GVA, and DC GZVA instructions are not data cache maintenance instructions.

For Watchpoint exceptions generated by a DC ZVA, DC GVA, or DC GZVA instruction, the address recorded is an address accessed by the instruction that triggered the watchpoint.

Otherwise, the address recorded must be both:

- From the inclusive range between:
  - The lowest address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint.
  - The highest watchpointed address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint. A watchpointed address is an address that the watchpoint is watching.

- Within a naturally-aligned block of memory that is all of the following:
  - A power-of-two size.
  - No larger than the DC ZVA block size.
  - Contains a watchpointed address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint.

The size of the block is IMPLEMENTATION DEFINED. There is no architectural means of discovering the size.

Example D2-2 Address recorded for a watchpoint programmed on 0x8019

A debugger programs a watchpoint to generate a Watchpoint exception on any access to the byte 0x8019.

An A32 load multiple instruction then loads nine registers starting from address 0x8004 upwards. This triggers the watchpoint.

If the DC ZVA block size is:

- 32 bytes, the address that the PE records must be between 0x8004 and 0x8019 inclusive.
- 16 bytes, the address that the PE records must be between 0x8010 and 0x8019 inclusive.

D2.10.6 Watchpoint behavior on other instructions

Under normal operating conditions, the following do not generate Watchpoint exceptions:

- Instruction cache maintenance instructions.
- Address translation instructions.
- TLB maintenance instructions.
• Prefetch memory instructions.
• All data cache maintenance instructions except DC IVAC.

Note

Despite their mnemonics, the DC ZVA, DC GVA, and DC GZVA instructions are not data cache maintenance instructions.

However, the debug architecture allows for implementation defined controls, such as those in ACTLR registers, to enable watchpoints on an implementation defined subset of these instructions. Whether a watchpoint treats the instruction as a load or a store, and the access size of instruction cache, address translation, and TLB operations are implementation defined.

The access size of the implementation defined instruction cache, address translation, and TLB operations which generate Watchpoint exceptions are implementation defined.

See also the following subsections:
• Watchpoint behavior on accesses by Store-Exclusive instructions.
• Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA, DC GVA, and DC GZVA instructions.

Watchpoint behavior on accesses by Store-Exclusive instructions

If a watchpoint matches on a data access caused by a Store-Exclusive instruction, then:
• If the store fails because an Exclusives monitor does not permit it, it is implementation defined whether the watchpoint generates a Watchpoint exception.
• Otherwise, the watchpoint generates a Watchpoint exception.

Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA, DC GVA, and DC GZVA instructions

DC ZVA, DC GVA and DC GZVA operations can generate Watchpoint exceptions. If the Point of Coherency is before any level of cache, it is implementation defined whether a DC IVAC instruction can generate a Watchpoint exception. Otherwise, DC IVAC operations can generate Watchpoint exceptions.

DC IVAC, DC ZVA, DC GZVA and DC GVA operations are treated as data stores by DBGWCR<\text{n}>_EL1.LSC.

Note

For the size of data accesses performed by the DC IVAC instruction and the DC ZVA instruction, see Watchpoint data address comparisons on page D2-4731. The size of all data accesses must be considered because watchpoints can be programmed to match on individual bytes.
D2.10.7 SVE Watchpoint exceptions

**RZRWTZ**
For SVE predicated vector load or store instructions which are not First-fault vector loads or Non-fault vector loads, when the instruction performs a non-speculative single-copy atomic access matching a configured watchpoint due to an *Active element*, a *Watchpoint debug event* is generated.

**RGRRCD**
For SVE predicated vector load or store instructions, if the instruction performs an access due to an *Inactive element*, a *Watchpoint debug event* is not generated.

**RDYGMS**
For SVE Non-fault vector load instructions, when the instruction performs an access, a *Watchpoint debug event* is not generated.

**RLLKQG**
For SVE Non-fault vector load instructions, when the instruction performs a non-speculative single-copy atomic access matching a configured watchpoint due to an *Active element*, the access is reported in the FFR.

**RTLSCX**
For SVE First-fault vector load instructions, when the instruction performs a non-speculative single-copy atomic access matching a configured watchpoint due to the *First active element*, a *Watchpoint event* is generated.

**RXBBLW**
For SVE First-fault vector load instructions, when the instruction performs a non-speculative single-copy atomic access matching a configured watchpoint due to an *Active element* that is not the *First active element*, a *Watchpoint debug event* is not generated and the access is reported in the FFR.

**RCKZFP**
Watchpoints are not a mechanism for preventing access to memory.

**IZHXGG**
For SVE Non-fault and First-fault vector load instructions that do not generate a *Watchpoint debug event*, an access that matches a configured watchpoint can return the data and set the appropriate FFR elements to FALSE.

D2.10.8 Watchpoint usage constraints

See the following:

- *Reserved DBGWCR<n>_EL1.{SSC, HMC, PAC} values*.
- *Reserved DBGWCR<n>_EL1.LBN values* on page D2-4738.
- *Programming dependencies of the BAS and MASK fields* on page D2-4738.
- *Reserved DBGWCR<n>_EL1.BAS values* on page D2-4738.
- *Reserved DBGWCR<n>_EL1.MASK values* on page D2-4739.
- *Other usage constraints* on page D2-4739.

**Reserved DBGWCR<n>_EL1.{SSC, HMC, PAC} values**

Table D2-16 on page D2-4737 shows when particular combinations of DBGWCR<n>_EL1.{SSC, HMC, PAC} are reserved.

<table>
<thead>
<tr>
<th>HMC, SSC, and PAC combination</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>All combinations with SSC set to 0b01 or 0b10, except for the combination with HMC set to 1, SSC set to 0b01, and PAC set to 0b00.</td>
<td>When EL3 is not implemented and EL2 is implemented.</td>
</tr>
<tr>
<td>All combinations where HMC or SSC is nonzero, except for the combination with HMC set to 1, SSC set to 0b01, and PAC set to 0b00 or combinations with SSC set to 0b11.</td>
<td>When both EL2 and EL3 are not implemented.</td>
</tr>
<tr>
<td>The combination with HMC set to 1, SSC set to 0b11, and PAC set to 0b00.</td>
<td>When EL2 is not implemented.</td>
</tr>
<tr>
<td>The combinations with SSC set to 0b11 except the combination with HMC set to 1, SSC set to 0b11, and PAC set to 0b00.</td>
<td>When Secure EL2 is not implemented.</td>
</tr>
<tr>
<td>The combination with HMC set to 1, SSC set to 0b01, and PAC set to 0b00.</td>
<td>When Secure EL2 is not implemented.</td>
</tr>
<tr>
<td>Combinations not included in Table D2-13 on page D2-4730.</td>
<td>Always.</td>
</tr>
</tbody>
</table>
If a watchpoint is programmed with one of these reserved combinations:

- The watchpoint must behave as if it is either:
  - Disabled.
  - Programmed with a combination that is not reserved, other than for a direct or external read of 
    DBGWCR<n>_EL1.
- For a direct or external read of DBGWCR<n>_EL1, if the reserved combination:
  - Has no function for any execution conditions, the value read back for each of SSC, HMC, and PMC 
    is UNKNOWN.
  - Has a function for execution conditions other than the current execution conditions, the value read 
    back is the value written. This permits software to save and restore the combination so that the 
    watchpoint functions for the other execution conditions.

The behavior of watchpoints with reserved combinations of SSC, HMC, and PAC might change in future revisions 
of the architecture. For this reason, software must not rely on the behavior described here.

**Reserved DBGWCR<n>_EL1.LBN values**

**For Linked Watchpoints**

A Linked watchpoint must link to a context-aware breakpoint. For a Linked watchpoint, any 
DBGWCR<n>_EL1.LBN value that is not for a context-aware breakpoint is reserved.

If a Linked watchpoint links to a breakpoint that is not implemented, or that is not context-aware, 
then reads of DBGWCR<n>_EL1.LBN return an UNKNOWN value and the behavior is 
CONSTRAINED UNPREDICTABLE. The Linked watchpoint behaves as if it is either:

- Disabled
- Linked to an UNKNOWN context-aware breakpoint.

If a Linked watchpoint links to a breakpoint that is implemented and is context-aware, but that is 
either not enabled or not programmed as a Linked Context breakpoint, it behaves as if it is disabled.

**For Unlinked Watchpoints** For Unlinked watchpoints, DBGWCR<n>_EL1.LBN reads UNKNOWN and its value is 
ignored.

**Programming dependencies of the BAS and MASK fields**

When programming a watchpoint, a debugger must use either:

- The MASK field, to program the watchpoint with an address range that can be eight bytes to 2GB.
- The BAS field, to select which bytes in the doubleword or word starting at the address contained in the 
  DBGWVR<n>_EL1 the watchpoint must generate Watchpoint exceptions for.

If the debugger uses the:

- MASK field, it must program BAS to be 0b11111111, so that all bytes in the doubleword or word are selected.
- BAS field, it must program MASK to be 0b000000, so that the MASK field does not indicate any address 
  ranges.

If an enabled watchpoint has a MASK field that is nonzero and a BAS field that is not set to 0b11111111, then for 
each byte in the address range, it is CONSTRAINED UNPREDICTABLE whether or not a Watchpoint exception 
is generated.

**Reserved DBGWCR<n>_EL1.BAS values**

The BAS field must be programmed with a value \( \text{Zeros}(8-n-m):\text{Ones}(n):\text{Zeros}(m) \), where:

- \( n \) is a nonzero positive integer less-than-or-equal-to 8.
- \( m \) is a positive integer less-than 8.
- \( n+m \) is less-than-or-equal-to 8.
All other values are reserved.

Note

If \( x \) is zero, then \( \text{Zeros}(x) \) is an empty bitstring.

If \( \text{DBGWVR}_{n}\_\text{EL1}[2] \) is 1, \( \text{DBGWCR}_{n}\_\text{EL1}.\text{BAS}[7:4] \) are RES0 and are ignored.

If a watchpoint is programmed with a reserved BAS value:

- It is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception for each byte in the doubleword or word of memory addressed by the \( \text{DBGWVR}_{n}\_\text{EL1} \).
- A direct or external read of \( \text{DBGWCR}_{n}\_\text{EL1}.\text{BAS} \) returns an UNKNOWN value.

Software must not rely on these properties as the behavior of reserved values might change in a future revision of the architecture.

Reserved \( \text{DBGWCR}_{n}\_\text{EL1}.\text{MASK} \) values

If a watchpoint is programmed with a reserved MASK value:

- The watchpoint must behave as if it is either:
  - Disabled.
  - Programmed with an UNKNOWN value that is not reserved, that might be 0b00000, other than for a direct or external read of \( \text{DBGWCR}_{n}\_\text{EL1} \).
- A direct or external read of \( \text{DBGWCR}_{n}\_\text{EL1}.\text{MASK} \) returns an UNKNOWN value.

Other usage constraints

For all watchpoints:

- \( \text{DBGWVR}_{n}\_\text{EL1}[1:0] \) are RES0 and are ignored.
- If \( \text{DBGWCR}_{n}\_\text{EL1}.\text{MASK} \) is nonzero, and any masked bits of \( \text{DBGWVR}_{n}\_\text{EL1} \) are not 0, it is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception when the unmasked bits match.
- A watchpoint never generates any Watchpoint exceptions if \( \text{DBGWCR}_{n}\_\text{EL1}.\text{LSC} \) is 0b00.

D2.10.9 Exception syndrome information and preferred return address

See the following:

- Exception syndrome information.
- Preferred return address on page D2-4740.

Exception syndrome information

On taking a Watchpoint exception, the PE records all of the following:

- Information about the exception in the \( \text{Exception Syndrome Register} \) (\( \text{ESR}_{\text{EL}x} \)) at the Exception level the exception is taken to.
- An address that the debugger can use to determine the memory location that caused the exception. The PE records this in a \( \text{Fault Address Register} \) (\( \text{FAR} \)).

The ESR and FAR used is either:

- \( \text{ESR}_{\text{EL1}} \) and \( \text{FAR}_{\text{EL1}} \), if the exception is taken to EL1.
- \( \text{ESR}_{\text{EL2}} \) and \( \text{FAR}_{\text{EL2}} \), if the exception is taken to EL2.
--- Note ---

Watchpoint exceptions cannot be taken to EL3 using AArch64.

See *ISS encoding for an exception from a Watchpoint exception* on page D17-5697 for more information.

**Preferred return address**

The preferred return address of a Watchpoint exception is the address of the instruction that was not executed because the PE took the Watchpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

### D2.10.10 Pseudocode description of Watchpoint exceptions taken from AArch64 state

*AArch64.WatchpointByteMatch()* tests an individual byte accessed by an operation.

*AArch64.StateMatch()* tests the values in DBGWCR<n>_EL1.{HMC, SSC, PAC}, and if the watchpoint is Linked, also tests the Linked Context breakpoint that the watchpoint links to.

*AArch64.WatchpointMatch()* tests the value in DBGWVR<n>_EL1.

*AArch64.CheckWatchpoint()* generates a FaultRecord that *AArch64.Abort()* raises a Watchpoint exception for if all of the following are true:

- MDSCR_EL1.MDE is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See *Enabling debug exceptions from the current Exception level* on page D2-4699.
- All of the conditions required for Watchpoint exception generation are met. See *About Watchpoint exceptions* on page D2-4726.

--- Note ---

*AArch64.CheckWatchpoint()* might halt the PE and cause it to enter Debug state. External debug uses Debug state.

*AArch64.WatchpointException()* is called to generate a Watchpoint exception.

These functions are defined in *Chapter J1 Armv8 Pseudocode*. 

---
D2.11 Vector Catch exceptions

Vector Catch exceptions are not generated in AArch64 translation regimes.

Note

This means that they are never taken to EL1 using AArch64 and are supported only if at least EL1 using AArch32 is supported.

A debugger that is executing in EL2 using AArch64 can route Vector Catch exceptions to EL2 using AArch64. See Routing debug exceptions on page D2-4697.

AArch64.VectorCatchException() is called to generate a Vector Catch exception.

Vector Catch exceptions on page G2-9093 describes Vector Catch exceptions.
D2.12 Software Step exceptions

The following subsections describe Software Step exceptions:

- About Software Step exceptions.
- Rules for setting MDSCR_EL1.SS to 1.
- The software step state machine.
- Entering the active-not-pending state on page D2-4744.
- Behavior in the active-not-pending state on page D2-4747.
- Entering the active-pending state on page D2-4749.
- Behavior in the active-pending state on page D2-4750.
- Stepping T32 IT instructions on page D2-4750.
- Exception syndrome information and preferred return address on page D2-4751.
- Additional considerations on page D2-4752.
- Pseudocode description of Software Step exceptions on page D2-4754.

D2.12.1 About Software Step exceptions

Software step is a resource that a debugger can use to make the PE single-step instructions. For example, by using software step, debugger software executing at a higher Exception level can single-step instructions at a lower Exception level.

Operation is as follows:

1. A debugger:
   a. Enables software step by setting MDSCR_EL1.SS to 1. See The debug exception enable controls on page D2-4696.
   b. Executes an exception return instruction, to branch to the instruction to be single-stepped in the software being debugged.

2. The PE then:
   a. Executes the instruction to be single-stepped.
   b. Takes a Software Step exception on the next instruction, returning control to the debugger.

However, another exception might be generated while the instruction is being stepped. This exception is either:

- A synchronous exception that is generated by the instruction being stepped.
- An asynchronous exception that is taken before or after the instruction being stepped.

The PE can take a Software Step exception only if debug exceptions are enabled from the current Exception level and Security state. See Enabling debug exceptions from the current Exception level on page D2-4699.

A state machine describes the behavior of software step, shown in The software step state machine. Throughout this Software Step exceptions section, including in all subsections, EL_D means the Exception level that Software Step exceptions are targeting. Routing debug exceptions on page D2-4697 defines EL_D as the debug target Exception level.

D2.12.2 Rules for setting MDSCR_EL1.SS to 1

Debugger software must be executing in an Exception level and Security state that debug exceptions are disabled from when it sets MDSCR_EL1.SS to 1.

The Exception level that hosts the debugger software must be using AArch64.

D2.12.3 The software step state machine

In Figure D2-3 on page D2-4743:

- The OS Lock is unlocked and DoubleLockStatus() == FALSE.
The PE is not in Secure state with MDCR_EL3.SDD set to 1. That is, the PE is in Non-secure state, or is in Secure state with MDCR_EL3.SDD set to 0, or the implementation does not include EL3.

**Figure D2-3 Software step state machine**

For a description of when debug exceptions are enabled or disabled from an Exception level, see *Enabling debug exceptions from the current Exception level* on page D2-4699.

For more information about how a step is completed, see *Behavior in the active-not-pending state* on page D2-4747.
The software step states are:

**Inactive**

Software step is inactive. It cannot generate any Software Step exceptions or affect PE execution. Software step is inactive whenever any of the following are true:

- MDSCR_EL1.SS is 0.
- ELD is using AArch32.
- Debug exceptions are disabled from the current Exception level or Security state.

**Active-not-pending**

None of the conditions mentioned in *Inactive* are true, therefore software step is active. The current instruction is the instruction to be stepped.

**Active-pending**

None of the conditions mentioned in *Inactive* are true, therefore software step is active. A Software Step exception is pending on the current instruction.

Whenever software step is active, whether the state machine is in the active-not-pending state or the active-pending state depends on PSTATE.SS. Table D2-17 on page D2-4744 shows this.

<table>
<thead>
<tr>
<th>ELD using:</th>
<th>Debug exception enable status in the current Exception level and Security state</th>
<th>MDSCR_EL1.SS</th>
<th>PSTATE.SS</th>
<th>State machine state</th>
</tr>
</thead>
<tbody>
<tr>
<td>AArch32</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Inactive</td>
</tr>
<tr>
<td>AArch64</td>
<td>Disabled</td>
<td>X</td>
<td>X</td>
<td>Inactive</td>
</tr>
<tr>
<td>AArch64</td>
<td>Enabled</td>
<td>0</td>
<td>X</td>
<td>Inactive</td>
</tr>
<tr>
<td>AArch64</td>
<td>Enabled</td>
<td>1</td>
<td>1</td>
<td>Active-not-pending</td>
</tr>
<tr>
<td>AArch64</td>
<td>Enabled</td>
<td>1</td>
<td>0</td>
<td>Active-pending</td>
</tr>
</tbody>
</table>

**D2.12.4 Entering the active-not-pending state**

Software step can only enter the active-not-pending state from the inactive state.

Software step:

- Enters the active-not-pending state when an Exception return instruction writes 1 to PSTATE.SS, by copying from SPSR_ELx.SS when it restores PSTATE.
- Might enter the active-not-pending state on exiting Debug state when DSPSR_EL0.SS or DSPSR.SS is 1. See *Exiting Debug state on page H2-10271*.

An Exception return instruction only copies 1 from SPSR_ELx.SS to PSTATE.SS if all of the following are true:

- MDSCR_EL1.SS is 1.
- ELD is using AArch64.
- Debug exceptions are disabled from the current Exception level.
- Debug exceptions are enabled from the Exception level that the Exception return instruction targets.

Otherwise, Exception return instructions set PSTATE.SS to 0, regardless of the value of SPSR_ELx.SS. Table D2-18 on page D2-4745 shows this. In the table:

- **Lock**  Means the value of (OSLSR_EL1.OSLK == '1' || DoubleLockStatus()).
- **NS**  Means the *Effective value of SCR_EL3.NS.*
SDD Means the Effective value of MDCR_EL3.SDD. See Disabling debug exceptions from Secure state on page D2-4699.

EEL2 Means the Effective value of SCR_EL3.EEL2. If FEAT_SEL2 is not implemented, this is 0.

TGE Means the value of HCR_EL2.TGE. If EL2 is not implemented, the PE behaves as if this is 0.

TDE Means the Effective value of MDCR_EL2.TDE. See Routing debug exceptions on page D2-4697.

**EL1 is using** The Execution state when the ELD is EL1.

**EL2 is using** The Execution state when the ELD is EL2.

<table>
<thead>
<tr>
<th>MDSCR_EL1.SS</th>
<th>Lock</th>
<th>NS</th>
<th>SDD</th>
<th>EEL2</th>
<th>TGE</th>
<th>TDE</th>
<th>EL1 is using</th>
<th>EL2 is using</th>
<th>Value an Exception return instruction writes to PSTATE.SS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>TRUE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>FALSE</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

0 0 X X AArch32 n/a 0

AArch64 n/a See Table D2-19 on page D2-4746

1 0 0 AArch32 n/a 0

AArch64 AArch64 See Table D2-19 on page D2-4746

1 AArch32 AArch32 0

X AArch64 See Table D2-20 on page D2-4747

1 X n/a AArch32 0

n/a AArch64 See Table D2-20 on page D2-4747

1 X X 0 0 AArch32 n/a 0

AArch64 AArch64 See Table D2-19 on page D2-4746

1 AArch32 AArch32 0

X AArch64 See Table D2-20 on page D2-4747

1 X n/a AArch32 0

n/a AArch64 See Table D2-20 on page D2-4747

For:

- If ELD is EL1 using AArch64, Table D2-19 on page D2-4746 shows the value an Exception return instruction writes to PSTATE.SS.
If ELₐ is EL2 using AArch64, Table D2-20 on page D2-4747 shows the value an Exception return instruction writes to PSTATE.SS.

In both tables:

- **From EL**: Means the Exception level at which the PE executes the Exception return instruction.
- **Target EL**: Is the target Exception level of the Exception return instruction.

--- **Note** ---

If the Exception return instruction is an illegal exception return, the target Exception level of the Exception return instruction is the current Exception level. See *Illegal exception returns from AArch64 state* on page D1-4645.

**KDE**

Is MDSCR_EL1.KDE. See *Enabling debug exceptions from the current Exception level* on page D2-4699.

### Table D2-19 Value an Exception return instruction writes to PSTATE.SS if ELₐ is EL1 using AArch64

<table>
<thead>
<tr>
<th>From EL</th>
<th>Target EL</th>
<th>KDE</th>
<th>PSTATE.D</th>
<th>SPSR_ELₓ.D</th>
<th>Software step enable status at:</th>
<th>Value an Exception return instruction writes to PSTATE.SS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>From EL</td>
<td>Target EL</td>
</tr>
<tr>
<td>EL₃</td>
<td>EL₃</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Disabled</td>
<td>Disabled</td>
</tr>
<tr>
<td></td>
<td>EL₂</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Disabled</td>
<td>Disabled</td>
</tr>
<tr>
<td></td>
<td>EL₁</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>Disabled</td>
<td>Disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>Disabled</td>
<td>Disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>Disabled</td>
<td>Enabled</td>
</tr>
<tr>
<td>EL₀</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Disabled</td>
<td>Enabled</td>
</tr>
<tr>
<td>EL₂</td>
<td>EL₂</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Disabled</td>
<td>Disabled</td>
</tr>
<tr>
<td></td>
<td>EL₁</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>Disabled</td>
<td>Disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>Disabled</td>
<td>Disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>Disabled</td>
<td>Enabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EL₁</td>
<td>EL₁</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>Disabled</td>
<td>Disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Enabled</td>
<td>Enabled²</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>Disabled</td>
<td>Disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>Disabled</td>
<td>Enabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EL₀</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Disabled</td>
<td>Enabled</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Enabled²</td>
<td>Enabled</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>X</td>
<td>Disabled</td>
<td>Enabled</td>
<td>SPSR_EL₁.SS</td>
</tr>
</tbody>
</table>

---

a. Because MDSCR_EL1.SS == 1, it means that the Exception return instruction is itself being stepped.
b. Depends on SPSR_EL1.D.

Table D2-20 Value an Exception return instruction writes to PSTATE.SS if EL_D is EL2 using AArch64

<table>
<thead>
<tr>
<th>From EL</th>
<th>Target EL</th>
<th>KDE</th>
<th>PSTATE.D</th>
<th>SPSR_ELx.D</th>
<th>Software step enable status at:</th>
<th>Value an Exception return instruction writes to PSTATE.SS</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL3</td>
<td>EL3</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Disabled</td>
<td>Disabled 0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>Disabled</td>
<td>Disabled 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>Disabled</td>
<td>Disabled 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>Disabled</td>
<td>Enabled SPSR_EL3.SS</td>
<td></td>
</tr>
<tr>
<td>EL1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Disabled</td>
<td>Enabled SPSR_EL3.SS</td>
<td></td>
</tr>
<tr>
<td>EL0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Disabled</td>
<td>Enabled SPSR_EL3.SS</td>
<td></td>
</tr>
<tr>
<td>EL2</td>
<td>EL2</td>
<td>0</td>
<td>X</td>
<td>Disabled</td>
<td>Disabled 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Enableda</td>
<td>Disabled 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td>Disabled</td>
<td>Disabled 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>Disabled</td>
<td>Enabled SPSR_EL2.SS</td>
<td></td>
</tr>
<tr>
<td>EL1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>Disabled</td>
<td>Enabled SPSR_EL2.SS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Enableda</td>
<td>Enabled 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>Disabled</td>
<td>Enabled SPSR_EL2.SS</td>
<td></td>
</tr>
<tr>
<td>EL0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>Disabled</td>
<td>Enabled SPSR_EL2.SS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Enableda</td>
<td>Enabled 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>Disabled</td>
<td>Enabled SPSR_EL2.SS</td>
<td></td>
</tr>
<tr>
<td>EL1</td>
<td>EL1</td>
<td>X</td>
<td>X</td>
<td>Enableda</td>
<td>Enabled 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EL0</td>
<td>X</td>
<td>X</td>
<td>Enableda</td>
<td>Enabled 0</td>
<td></td>
</tr>
</tbody>
</table>

a. Because MDSCR_EL1.SS == 1, it means that the Exception return instruction is itself being stepped.
b. Depends on SPSR_EL2.D.

---

**Note**

No AArch32 instruction can set PSTATE.SS to 1.

### D2.12.5 Behavior in the active-not-pending state

In this state, the PE does one of the following:
- Executes the instruction to be stepped and either:
  - Completes it without taking a synchronous exception.
  - Takes a synchronous exception if the instruction generates one.
- Takes an asynchronous exception without executing any instructions.
• Enters Debug state because of a Halting debug event.

If the PE executes the instruction without taking any exceptions, then the PE sets `PSTATE.SS` to 0, meaning that after the instruction has been executed:

• If the instruction has disabled debug by setting `PSTATE.D` to 1 then software step advances to the inactive state.

• If the instruction disables software step by a direct write to a System register, for example a write to `MDSCR_EL1.KDE` or `MDSCR_EL1.SS`, then software step might advance to the inactive state. These writes require explicit synchronization to guarantee their effect. See Synchronization and the software step state machine on page D2-4753.

• Otherwise, software step advances to the active-pending state. See Behavior in the active-pending state on page D2-4750.

If the PE takes either a synchronous or an asynchronous exception, behavior is as described in one of the following:

• If the PE takes an exception to an Exception level that is using AArch64.

• If the PE takes an exception to an Exception level that is using AArch32 on page D2-4749.

If the PE enters Debug state because of a Halting debug event, behavior is as described in Entering Debug state and Software Step on page H2-10241.

If the PE takes an exception to an Exception level that is using AArch64

As part of exception entry, the PE does all of the following:

• Sets `SPSR_ELx.SS` to 0 or 1, depending on the exception. See Table D2-21 on page D2-4748.

• It is UNPREDICTABLE whether `SPSR_ELx.SS` to 0 or 1 when an SError interrupt is taken to ELx without executing the instruction.

• Sets `PSTATE.SS` to 0. This causes software step to enter either the active-pending state or the inactive state, depending on whether debug exceptions are enabled or disabled from the Exception level that the exception is taken to:
  - Enabled  Software step enters the active-pending state.
  - Disabled  Software step enters the inactive state.
  - In either case, on taking the exception, a step is complete.

• Sets `PSTATE.D` to 1.

### Table D2-21 Categorization of exceptions, for setting `SPSR_ELx.SS` to 0 or 1

<table>
<thead>
<tr>
<th>Exception description</th>
<th>Exceptions</th>
<th>SPSR_ELx.SS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exceptions whose preferred return address is for the instruction that follows the instruction to be stepped.</td>
<td>Supervisor Call (SVC) exceptions, Hypervisor Call (HVIC) exceptions, Secure Monitor Call (SMC) exceptions.</td>
<td>0</td>
</tr>
<tr>
<td>Exceptions whose preferred return address is the address of the instruction to be stepped.</td>
<td>All other synchronous exceptions, and asynchronous exceptions that are taken before the instruction to be stepped.</td>
<td>1</td>
</tr>
</tbody>
</table>

--- Note ---

If an SMC instruction executed at Non-secure EL1 is trapped to EL2 because `HCR_EL2.TSC` is 1, the exception is a Trap exception, not a Secure Monitor Call exception, and so `SPSR_ELx.SS` is set to 1, not 0.
If the PE takes an exception to an Exception level that is using AArch32

This can happen only when all of the following is true:

- EL2 is implemented and is using AArch64, and the Effective value of MDCR_EL2.TDE is 1. Because MDCR_EL2.TDE is 1, ELD is EL2.
- The exception is taken to EL1 using AArch32.

As part of exception entry, the PE sets PSTATE.SS to 0. This causes software step to enter the active-pending state.

--- Note ---
- Software step always enters the active-pending state because the exception is taken to an Exception level that debug exceptions are enabled from, EL1. Debug exceptions are enabled from EL1 because ELD is EL2, and debug exceptions are always enabled from Exception levels that are lower than ELD.
- AArch32 SPSRs have no SS bit.

Summary of behavior in the active-not-pending state

Table D2-22 on page D2-4749 summarizes behavior in the active-not-pending state.

<table>
<thead>
<tr>
<th>Event</th>
<th>Value written to PSTATE.SS</th>
<th>Target Exception level is using:</th>
<th>Detailsa</th>
<th>Value written to SPSR_ELx.SS</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>No exception</td>
<td>0</td>
<td>n/a</td>
<td>Disables Software step</td>
<td>n/a</td>
<td>Inactive</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Otherwise</td>
<td>n/a</td>
<td>Active-pending</td>
</tr>
<tr>
<td>Exception</td>
<td>0</td>
<td>AArch64</td>
<td>Supervisor Call (SVC)</td>
<td>0</td>
<td>Active-pending or inactiveb</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Hypervisor Call (HVC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Secure Monitor Call (SMC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other</td>
<td>1</td>
<td></td>
<td>Active-pending</td>
</tr>
<tr>
<td>AArch32</td>
<td>All</td>
<td>0</td>
<td></td>
<td></td>
<td>Active-pending</td>
</tr>
</tbody>
</table>

a. For the No exception rows, this column shows the effect of the event. For the Exception rows, this column shows the exception taken.
b. Which state software step enters depends on whether debug exceptions are enabled or disabled from the target Exception level. See Figure D2-3 on page D2-4743.

D2.12.6 Entering the active-pending state

Software step enters the active-pending state after any of the following operations, provided that both:

- MDSCR_EL1.SS is 1.
- Debug exceptions are enabled from the Exception level and Security state that execution is in after the operation.

The operations are:

While software step is in the active-not-pending state

The PE either:

- Executing the instruction to be stepped without taking any exceptions.
- Taking an exception.
While software step is in the active-pending state
The PE takes an asynchronous exception.

While software step is in the inactive state
The PE executes either:
• An Exception return instruction when SPSR_ELx.SS is 0.
• An instruction that enables debug by setting PSTATE.D to 0.

Note
If entry to the active-pending state is because of the PE taking an exception, it means that the exception is one that is taken to EL1 when MDCR_EL2.TDE is 1 and EL2 is implemented and enabled in the current Security state. Otherwise, debug exceptions are masked by PSTATE.D, therefore they would be disabled from the target Exception level of the exception.

In addition, software step might enter the active-pending state either:
• After a direct write to a System register, for example a write to MDSCR_EL1.KDE or MDSCR_EL1.SS. These writes require explicit synchronization to guarantee their effect. See Synchronization and the software step state machine on page D2-4753.
• On exiting Debug state when DSPSR_EL0.SS or DSPSR.SS is 0. See Exiting Debug state on page H2-10271.

D2.12.7 Behavior in the active-pending state
When the PE is in the active-pending state, a Software Step exception is taken before the PE executes an instruction. The Software Step exception has higher priority than all other types of synchronous exception. However, the prioritization of this exception with respect to any unmasked pending asynchronous exception is not defined by the architecture.

For more information, see the following:
• Prioritization of Synchronous exceptions taken to AArch64 state on page D1-4650.
• Prioritization of interrupts on page D1-4664.
• Architectural requirements for taking asynchronous exceptions on page G1-8935.

D2.12.8 Stepping T32 IT instructions
The A-profile architecture permits a combination of an IT instruction and another 16-bit T32 instruction to comprise one 32-bit instruction.

For the purpose of stepping an item, it is IMPLEMENTATION DEFINED whether:
• The PE considers this combination to be one instruction.
• The PE considers this combination to be two instructions.

In an implementation that supports the ITD control, which can disable some uses of the IT instruction, it is then IMPLEMENTATION DEFINED whether this behavior depends on the value of the applicable ITD field. For example:
• The PE might consider this combination to be one instruction, regardless of the state of the applicable ITD field.
• The PE might consider this combination to be two instructions, regardless of the state of the applicable ITD field.
• The PE might consider this combination to be one instruction when the applicable ITD field is 1, and two instructions when it is 0.

The applicable ITD field is one of:
• SCTLR_EL1.ITD if execution is at EL0 using AArch32 when EL1 is using AArch64.
• SCTLR.ITD if execution is at EL0 or EL1 when EL1 is using AArch32.
• HSCTRL.ITD if execution is at Non-secure EL2 using AArch32.

D2.12.9 Exception syndrome information and preferred return address

See the following:
• Exception syndrome information.
• Preferred return address on page D2-4752.

Exception syndrome information

On taking a Software Step exception, the PE records information about the exception in the Exception Syndrome Register (ESR_ELx) at the Exception level the exception is taken to. See ISS encoding for an exception from a Software Step exception on page D17-5697 for more information.

If no instruction was stepped because software step entered the active-pending state from the inactive state without passing through the active-not-pending state, then ESR_ELx.ISV, EX are set to 0.

When an instruction has been stepped, if the stepped instruction was a conditional Load-Exclusive instruction that failed its Condition code test, then ESR_ELx.ISV is set to 1 and ESR_ELx.EX is set to a CONSTRAINED UNPREDICTABLE choice of 0 or 1.

When an instruction has been stepped, if the stepped instruction was an Exception return instruction or an ISB, then ESR_ELx.ISV is set to a CONSTRAINED UNPREDICTABLE choice of 0 or 1, and ESR_ELx.EX is set to 0.

If the Effective value of MDCR_EL2.TDE == 1, EL2 is implemented and enabled in the current Security state, and a different exception is taken before the Software Step exception, then ESR_ELx.ISV is set to a CONSTRAINED UNPREDICTABLE choice of 0 or 1. In this case:
• If ESR_ELx.ISV is set to 1, then ESR_ELx.EX is set to the correct value for the instruction.
• If ESR_ELx.ISV is set to 0, then ESR_ELx.EX is set to zero.

Other than for the cases described above, when an instruction has been stepped:
• ESR_ELx.ISV is set to 1, to indicate that the EX bit is valid.
• The value of ESR_ELx.EX is set according to the instruction stepped. When:
  — The instruction stepped was an instruction other than a Load-Exclusive instruction, an Exception Return instruction, or an ISB, and no other exception was taken before the Software Step exception, ESR_ELx.EX is set to 0.
  — The instruction stepped was a Load-Exclusive instruction that was either not conditional or did not fail its Condition code test, ESR_ELx.EX is set to 1.

Note

If the PE cannot determine the correct value of ESR_ELx.EX for the stepped instruction, then it sets ESR_ELx.{ISV, EX} to {0, 0}. For example, the exception is taken before the PE decodes the stepped instruction, or the exception means the PE has no valid stepped instruction to decode.

Note

A Load-Exclusive instruction is any one of the following:
• In the A64 instruction set, any instruction that has a mnemonic starting with either LDX or LDAX.
• In the A32 and T32 instruction sets, any instruction that has a mnemonic starting with either LDREX or LDAEX.

Note

An implementation that always sets ISV to 0 and never sets EX is not compliant.
Table D2-23 on page D2-4752 summarizes the possible values that the PE can record in ESR_ELx.{ISV, EX}.

<table>
<thead>
<tr>
<th>Description</th>
<th>ESR_ELx.ISV</th>
<th>ESR_ELx.EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syndrome data is not available because no instruction was stepped.</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Syndrome data is available because an instruction was stepped. The instruction stepped was a conditional Load-Exclusive instruction that failed its Condition code test.</td>
<td>1</td>
<td>0 or 1</td>
</tr>
<tr>
<td>Syndrome data is available because an instruction was stepped. The instruction stepped was an Exception Return instruction or an ISB.</td>
<td>0 or 1</td>
<td>0</td>
</tr>
<tr>
<td>A different exception is taken before the Software Step exception.</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Set to the correct value for the instruction.</td>
</tr>
<tr>
<td>Syndrome data is available because an instruction was stepped. The instruction stepped was an instruction other than a Load-Exclusive instruction, an Exception Return instruction, or an ISB, and no other exception was taken before the Software Step exception.</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Syndrome data is available because an instruction was stepped. The instruction stepped was a Load-Exclusive instruction that was either not conditional or did not fail its Condition code test.</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Preferred return address**

The preferred return of a Software Step exception is the address of the instruction that was not executed because the PE took the Software Step exception instead.

### D2.12.10 Additional considerations

This section contains the following:

- Behavior when an Exception return instruction is an illegal exception return.
- Behavior when the instruction stepped writes a misaligned PC value on page D2-4753.
- Stepping code that uses Exclusives monitors on page D2-4753.
- Synchronization and the software step state machine on page D2-4753.

**Behavior when an Exception return instruction is an illegal exception return**

If the conditions for entering the active-not-pending state in Entering the active-not-pending state on page D2-4744 are met, but the PE executes an Exception return instruction that is an illegal exception return, the exception return must be taken to the same Exception level that it was taken from. In this scenario, even though the Exception level remains the same before and after the Exception return instruction, software step can advance from the inactive state to one of the active states. Consider the following case:

1. MDSCR_EL1.SS is 1 and software step is inactive. The current Exception level is EL1 using AArch64, the OS Lock and OS Double Lock are unlocked, and MDCR_EL2.TDE is 0, MDSCR_EL1.KDE is 1, and PSTATE.D is 1.
   - PSTATE.D == 1 is the reason why software step is inactive, because PSTATE.D == 1 means that debug exceptions are disabled from the current Exception level.
2. The PE executes an Exception return instruction.
3. The intended target of the Exception return instruction is EL2. This means that the Exception return instruction is an illegal exception return because the intended target is higher than the Exception level the Exception return instruction it is executed at. In this case, the Exception return instruction must target EL1 instead of EL2.

If SPSR_EL1.D is 0, then on the Exception return instruction PSTATE.D becomes 0 and debug exceptions become enabled from the current Exception level. Software step therefore advances from the inactive state to one of the active states.

Which active state software step advances to depends on whether SPSR_ELx.SS is 1 or 0:

- If SPSR_ELx.SS is 1, software step advances to the active-not-pending state.
  In this case, an Illegal Execution state exception is pending on the instruction to be stepped, and the PE takes the Illegal Execution state exception instead of executing the instruction to be stepped.

- If SPSR_ELx.SS is 0, software step advances to the active-pending state.
  In this case, a Software Step exception and an Illegal Execution state exception are both pending. The Software Step exception has higher priority. On taking the Software Step exception, the PE sets SPSR_ELx.IL to 1.

Note

Prioritization of Synchronous exceptions taken to AArch64 state on page D1-4650 shows the relative priorities of synchronous exceptions.

Behavior when the instruction stepped writes a misaligned PC value

An indirect branch that writes a misaligned PC value might generate a PC alignment fault exception at the target of the branch. However, if the indirect branch is stepped using software step, the PE takes a Software Step exception instead, because the Software Step exception has higher priority. Behavior on returning from the Software Step exception depends on which Execution state the Exception level being returned to is using:

AArch64 A PC alignment fault exception is generated.

AArch32 The return from the Software Step exception forces the PC to the correct alignment, and no PC alignment fault exception is generated.

Debugger software must therefore take care when using software step to single-step an indirect branch instruction executed in AArch32 state, that it does not hide a PC alignment fault exception.

Stepping code that uses Exclusives monitors

The A-profile architecture provides no mechanism for preserving the state of the Exclusives monitors when a Load-Exclusive or a Store-Exclusive instruction is stepped.

However, for certain progressions through the software step state machine, on taking a Software Step exception, the PE provides an indication of whether the instruction stepped was a Load-Exclusive instruction.

Debugger software can use this to detect the state of the Exclusives monitors. For example, if the PE reports that the instruction stepped was a Load-Exclusive instruction, the debugger is aware that the next Store-Exclusive operation will fail, because all Exclusives monitors are cleared on returning from the Software Step exception. The debugger must then take action to ensure that the code being stepped makes forwards progress.

For more information on how the PE reports whether the instruction stepped was a Load-Exclusive instruction, see Exception syndrome information and preferred return address on page D2-4751.

Synchronization and the software step state machine

Any of the following can cause transitions between software step states:

- A direct write to a System register.
- A direct write to a Special-purpose register.
• A write to an external debug register.

The software step state machine indirectly reads some of these registers and so is not guaranteed to observe any new values until after a *Context synchronization event* has occurred.

Between a write to the register and the next *Context synchronization event*, it is CONstrained UNPREDICTABLE whether software step uses the state of the PE before the write, or the state of the PE after the write.

After a *Context synchronization event*, the state machine must use the state of the PE after the write.

**Example D2-3 Example of synchronization and software step state machine changing states**

1. Software changes MDSCR_EL1.SS from 0 to 1 when debug exceptions are enabled.
2. The PE executes some instructions.
3. A *Context synchronization event* occurs.

During step 2, it is CONstrained UNPREDICTABLE whether software step remains in the inactive state, as if MDSCR_EL1.SS is 0, or enters the active-pending state because MDSCR_EL1.SS is 1. If it is in the:
   • Inactive state, then after the *Context synchronization event*, it must enter the active-pending state.
   • Active-pending state, the PE might take a Software Step exception before the *Context synchronization event*.

---

**Note**

A direct write to a Special-purpose register does not require explicit synchronization.

---

**D2.12.11 Pseudocode description of Software Step exceptions**

`SSAdvance()` advances software step from the active-not-pending state to the active-pending state, by setting PSTATE.SS to 0. It is called on completing execution of each instruction.

`CheckSoftwareStep()` checks whether software step is in the active-pending state, and if it is, generates a Software Step exception. It is called before each instruction executed, regardless of Execution state, before checking for any other synchronous exceptions.

`DebugExceptionReturnSS()` returns the value to write to PSTATE.SS on an exception return or an exit from Debug state. See *Entering the active-not-pending state on page D2-4744*.

These functions are defined in Chapter J1 *Armv8 Pseudocode*.
**D2.13 Synchronization and debug exceptions**

The behavior of debug depends on all of the following:

- The state of the external debug authentication interface.
- Indirect reads of:
  - External debug registers.
  - System registers, including system debug registers.
  - Special-purpose registers.

If a change is made to any of these, the effect of that change on debug exception generation cannot be relied on until after a *Context synchronization event* has occurred. Similarly, the effect of the change on the software step state machine cannot be relied on until after a *Context synchronization event* has occurred.

For any instructions executed between the time when the change is made and the time when the next *Context synchronization event* occurs, it is **CONSTRAINED UNPREDICTABLE** whether debug uses the state of the PE before the change, or the state of the PE after the change.

**Example D2-4 Example of synchronization and Breakpoint exception generation**

1. Software changes MDSCR_EL1.MDE from 0 to 1.
2. An instruction is executed, that would cause a Breakpoint exception if self-hosted debug uses the state of the PE after the change.
3. A *Context synchronization event* occurs.

In this case, it is **CONSTRAINED UNPREDICTABLE** whether the instruction generates a Breakpoint exception.

**Example D2-5 Example of synchronization and debug exceptions generation**

1. Software unlocks the OS Lock.
2. The PE executes some instructions.
3. A *Context synchronization event* occurs.

During the time when the PE is executing some instructions, step 2, it is **CONSTRAINED UNPREDICTABLE** whether debug exceptions other than Breakpoint Instruction exceptions can be generated.

**Note**

Some register updates are self-synchronizing. Others require an explicit *Context synchronization event*. For more information, see:

- *Accessing PSTATE fields* on page D1-4671.
- *Synchronization requirements for AArch64 System registers* on page D17-5547.
- *Synchronization of changes to the external debug registers* on page H8-10358.
Chapter D3
AArch64 Self-hosted Trace

This chapter describes the AArch64 self-hosted trace:

Introductory information:
- About self-hosted trace on page D3-4758.
- Trace sinks on page D3-4758.
- Register controls to enable self-hosted trace on page D3-4758.

Prohibited regions in trace:
- Controls to prohibit trace at Exception levels on page D3-4759.
- Self-hosted trace and visibility of virtual data on page D3-4760.

Timestamps and Synchronization:
- Self-hosted trace timestamps on page D3-4761.
- Synchronization in self-hosted trace on page D3-4762.
D3.1 About self-hosted trace

A trace unit generates trace data to describe the program flow of the PE.

The trace unit may be an implementation of any of the following:

- In Armv9, the Embedded Trace Extension (ETE). See Chapter D4 The Embedded Trace Extension.
- An IMPLEMENTATION DEFINED trace function.

If an Armv8.4-compliant PE implements an ETM Architecture trace unit that includes the ETM System register interface, FEAT_TRF must be implemented.

If an Armv8.4-compliant PE implements a trace unit that is either not an ETM Architecture trace unit or does not implement the ETM System register interface, Arm recommends that FEAT_TRF is implemented, but this is not mandatory. This is not applicable in Armv9.

If an Armv9-compliant PE implements FEAT_ETE, FEAT_TRF must be implemented.

Self-hosted trace happens when the agent controlling the trace collection is part of the same software stack as the software being traced. The agent controls prohibited regions. The information collected by the agent is sent to a trace sink.

The trace unit and the PE must have the same view of the debug authentication interface. If FEAT_TRF is implemented, ExternalNoninvasiveDebugEnabled() is always TRUE.

D3.1.1 Trace sinks

The trace unit sends the trace data to a trace sink. A system might include multiple trace sinks, and allow software to configure which trace sink or sinks are used.

An example of an internal trace sink is an Embedded Trace Router (ETR), which allows software to define a buffer in memory. Trace data is written to this buffer.

If FEAT_TRBE is implemented, the PE includes a Trace Buffer Unit. Trace data is written directly to memory by the Trace Buffer Unit. See Chapter D6 The Trace Buffer Extension.

In Armv8, Arm recommends that a system that includes FEAT_TRF incorporates an ETR, and follows the system architecture described by the CoreSight Base System Architecture (CS-BSA).

D3.1.2 Register controls to enable self-hosted trace

If FEAT_TRF is implemented, self-hosted trace is enabled if one of the following is true:

- EDSCR.TFO == 0.
- EDSCR.TFO == 1, EL3 is implemented, MDCR_EL3.STE == 1 and ExternalSecureNoninvasiveDebugEnabled() == FALSE.
- EDSCR.TFO == 1, EL3 is not implemented, the PE executes in Secure state and ExternalSecureNoninvasiveDebugEnabled() = FALSE.

The pseudocode function SelfHostedTraceEnabled() shows these rules.

If FEAT_TRF is not implemented, SelfHostedTraceEnabled() returns FALSE.

While SelfHostedTraceEnabled() is FALSE, ExternalNoninvasiveDebugAllowed() controls whether tracing is prohibited or allowed in each Security state.

The self-hosted trace extensions do not provide any mechanism to control software access to the trace unit external debug interface.
D3.2  Prohibited regions in self-hosted trace

Trace is not generated in prohibited regions. The pseudocode function `TraceAllowed()` indicates whether tracing is allowed in the current Security state and Exception level.

The IMPLEMENTATION DEFINED debug authentication interface can allow an external agent to disable the self-hosted trace extension.

If `SelfHostedTraceEnabled()` == TRUE, tracing is prohibited in Secure state when MDCR_EL3.STE == 0.

D3.2.1 Controls to prohibit trace at Exception levels

If `SelfHostedTraceEnabled()` == TRUE, TRFCR_EL1 and TRFCR_EL2 control whether trace is prohibited at an Exception level. While `SelfHostedTraceEnabled()` == FALSE, the registers TRFCR_EL1 and TRFCR_EL2 are ignored.

If `SelfHostedTraceEnabled()` == TRUE, tracing is prohibited at EL0 if one of the following is true:
- The Effective value of HCR_EL2.TGE == 0 and TRFCR_EL1.E0TRE == 0.
- The Effective value of HCR_EL2.TGE == 1 and TRFCR_EL2.E0HTRE == 0.

If `SelfHostedTraceEnabled()` == TRUE, tracing is prohibited at EL1 if TRFCR_EL1.E1TRE == 0.

If `SelfHostedTraceEnabled()` == TRUE, tracing is prohibited at EL2 if TRFCR_EL2.E2TRE == 0.

If `SelfHostedTraceEnabled()` == TRUE, tracing is prohibited at EL3 if one of the following is true:
- EL3 is using AArch64 state.
- EL3 is using AArch32 state and TRFCR.E1TRE == 0.

The pseudocode `TraceAllowed()` shows the above rules.

If `SelfHostedTraceEnabled()` == TRUE, no events are exported to the trace unit when tracing is prohibited.

If `SelfHostedTraceEnabled()` == FALSE, no events are exported to the trace unit when the PE is in Secure state and counting in Secure state is prohibited.

If FEAT_ETE is not implemented, when PMCR_EL0.X==0 or PMCR.X==0, no PMU events are exported to the trace unit.

Otherwise, PMU events are exported to the trace unit.

If `SelfHostedTraceEnabled()` == TRUE, Table D3-1 on page D3-4760 shows the prohibited regions by Exception level and state.

In the table:

- **STE** Means the Effective value of MDCR_EL3.STE or SDCR.STE, as applicable.
- **EEL2** Means the Effective value of SCR_EL3.EEL2.
- **TGE** Means the Effective value of HCR_EL2.TGE.
- **P** Means prohibited.
- **E2TRE** Means prohibited if TRFCR_EL2.E2TRE == 0.
- **E1TRE** Means prohibited if TRFCR_EL1.E1TRE == 0.
- **E0HTRE** Means prohibited if TRFCR_EL2.E0HTRE == 0.
- **E0TRE** Means prohibited if TRFCR_EL1.E0TRE == 0.
- **n/a** Not applicable.

If FEAT_TRBE is not implemented, trace is allowed unless otherwise prohibited in Table D3-1 on page D3-4760.
If FEAT_TRBE is implemented, the requirements in this section are extended. See *The owning translation regime* on page D6-5010.

### Table D3-1 Prohibited regions

<table>
<thead>
<tr>
<th>Controls</th>
<th>Tracing prohibited at</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State</strong></td>
<td><strong>EL3 using</strong></td>
</tr>
<tr>
<td>Non-secure</td>
<td>X</td>
</tr>
<tr>
<td>Secure</td>
<td>X</td>
</tr>
<tr>
<td>Secure</td>
<td>0</td>
</tr>
<tr>
<td>Secure</td>
<td>1</td>
</tr>
<tr>
<td>Secure</td>
<td>1</td>
</tr>
<tr>
<td>Secure</td>
<td>1</td>
</tr>
<tr>
<td>AArch32</td>
<td>X</td>
</tr>
</tbody>
</table>

#### D3.2.2 Self-hosted trace and visibility of virtual data

A hypervisor can use TRFCR_EL2.CX to control visibility of CONTEXTIDR_EL2 and VTTBR_EL2.VMID. If SelfHostedTraceEnabled() == TRUE and TRFCR_EL2.CX == 0, or if EL2 is not implemented:

- The values of CONTEXTIDR_EL2 and VTTBR_EL2.VMID are not traced.
- Comparisons between CONTEXTIDR_EL2 and VTTBR_EL2.VMID do not match and results of comparison are not exposed through the comparators.

The trace unit may either prohibit trace for these values, or may record a CONTEXTIDR_EL2 or VTTBR_EL2.VMID value of zero in the trace.
D3.3 Self-hosted trace timestamps

The trace timestamp is a value that represents the passage of time in real-time. It is calculated from a counter which increments all the time, when the PE is generating trace and when the PE is in a prohibited region.

While `SelfHostedTraceEnabled()` == FALSE, the external trace provides the trace timestamp. If the external trace is a standard CoreSight system, the relationship between CoreSight time and the Generic Timer counter is IMPLEMENTATION DEFINED.

When `SelfHostedTraceEnabled()` == TRUE, the trace timestamp is one of the following:

- Physical time, which is defined by the physical count value returned by `PhysicalCountInt()`.
- If FEAT_ECV is implemented, offset physical time, which is defined as the value of (`PhysicalCountInt()` - CNTPOFF_EL2).
- Virtual time, which is defined as the value of (`PhysicalCountInt()` - CNTVOFF_EL2). The virtual offset is always CNTVOFF_EL2, including when a read of CNTVCT_EL0 at the current Exception level would treat the virtual offset as zero.

The fields TRFCR_EL2.TS, HTRFCR.TS, TRFCR_EL1.TS and TRFCR.TS control which counter is used for self-hosted trace.

The timestamp used for trace is shown in Table D3-2 on page D3-4761.

In Table D3-2 on page D3-4761, if any of the following are true, the value of physical offset is zero, otherwise the value of physical offset is the value of CNTPOFF_EL2:

- EL3 is using AArch32.
- EL2 is not implemented.
- FEAT_ECV is not implemented.
- The Effective value of SCR_EL3.{NS,RW} is {1,0}.
- CNTHCTL_EL2.ECV is 0.
- SCR_EL3.ECVEn is 0.

Note

The counter value used for the trace timestamp is not affected by the value of HCR_EL2.E2H, or whether EL2 is enabled or disabled in the current Security state.

<table>
<thead>
<tr>
<th>SelfHostedTraceEnabled()</th>
<th>TRFCR_EL2.TS</th>
<th>TRFCR_EL1.TS</th>
<th>Timestamp traced</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>xx</td>
<td>xx</td>
<td>CoreSight time</td>
</tr>
<tr>
<td>TRUE</td>
<td>0b00</td>
<td>0b01</td>
<td><code>PhysicalCountInt()</code> - CNTVOFF_EL2</td>
</tr>
<tr>
<td></td>
<td>0b00</td>
<td>0b10</td>
<td><code>PhysicalCountInt()</code> - physical offset</td>
</tr>
<tr>
<td></td>
<td>0b00</td>
<td>0b11</td>
<td><code>PhysicalCountInt()</code></td>
</tr>
<tr>
<td></td>
<td>0b01</td>
<td>xx</td>
<td><code>PhysicalCountInt()</code> - CNTVOFF_EL2</td>
</tr>
<tr>
<td></td>
<td>0b10</td>
<td>xx</td>
<td><code>PhysicalCountInt()</code> - physical offset</td>
</tr>
<tr>
<td></td>
<td>0b11</td>
<td>xx</td>
<td><code>PhysicalCountInt()</code></td>
</tr>
</tbody>
</table>
D3.4 Synchronization in self-hosted trace

The trace unit is an indirect observer of the System registers.

If FEAT_TRBE is implemented, the requirements in this section are extended. See Synchronization and the Trace Buffer Unit on page D6-5026.

While SelfHostedTraceEnabled() == TRUE, indirect reads of the trace filter control fields, TRFCR_EL1.{E1TRE, E0TRE} and TRFCR_EL2.{E2TRE, E0HTRE} are treated as indirect reads made by the instruction being traced. For these register fields, in addition to the standard requirements for synchronization of System register accesses, when a trace filter control value is changed and synchronization is not explicitly specified, one of the following occurs:

- The behavior of the PE must be consistent with the control value having the old value.
- The behavior of the PE must change the control value at a point in the simple sequential execution of the program, so that before that point, the behavior of the PE is consistent with the control value having the old value, and after that point the behavior of the PE is consistent with the control value having the new value.

If there are multiple direct writes to the register without explicit synchronization, the behavior is consistent with the writes occurring in program order.

The TSB CSYNC operation is used to ensure that a trace operation, due to a trace unit generating trace for an instruction has completed. The TSB CSYNC operation may be reordered with respect to other instructions, so must be combined with at least one Context synchronization event to ensure the operations are executed in the required order. This means that a direct write to TRFCR_EL1 or TRFCR_EL2 is guaranteed to be observed by the trace unit only after a subsequent Context synchronization event. For more information, see Trace Synchronization Barrier (TSB CSYNC) on page B2-177.

While SelfHostedTraceEnabled() == FALSE, the trace unit might impose stronger synchronization requirements.
Chapter D4
The Embedded Trace Extension

This chapter describes the Embedded Trace Extension (ETE). It contains the following sections:

- About the Embedded Trace Extension on page D4-4764.
- Programmers’ model on page D4-4771.
- Trace elements on page D4-4775.
- Instruction and exception classification on page D4-4791.
- About the ETE trace unit on page D4-4800.
- Resource operation on page D4-4843.
D4.1 About the Embedded Trace Extension

FEAT_ETE describes the operation of a trace unit. The trace unit provides details about software control flow running on a PE which can be used to aid debugging or optimization. The trace unit provides filtering functionality to allow the targeting of the information to specific code regions or periods of operation.

FEAT_ETE overlaps with the ETMv4 architecture, while also providing support for new architecture features. For more information on the ETMv4 architecture, see the ARM® Embedded Trace Macrocell Architecture Specification, ETMv4 (ARM IHI 0064). FEAT_ETE does not support all the features of ETMv4. For more information on the differences between FEAT_ETE and the ETMv4 architecture, see Differences between ETM and ETE on page K4-11671.

D4.1.1 Attributes of tracing

The attributes of PE tracing are:

Trace is generated in real time.

Trace provides a means of observing the PE operation while the PE is running. For diagnostic purposes, this is useful as some types of erroneous behavior are only solvable by observing the system during runtime. In addition, because the PE trace can include cycle counts, trace can be used for PE profiling purposes.

Trace has a minimal effect on functional performance of the PE.

Usually, trace has no effect on the functional performance of the PE. This attribute does depend on the market use of the PE being debugged, however, and on the trace requirements for the PE and the trace solution that is adopted to meet those requirements. For some markets, some impact on PE performance is acceptable but for others, most notably in real-time systems, an impact on PE performance might be unacceptable.

Trace is available for self-hosted analysis.

The trace from a PE or process is available for analysis by software running on the target. See Self-hosted Trace.

Trace is deeply embedded in an SoC.

Trace provides a method of debugging software executing on PEs that are deeply embedded within an SoC.

Trace is available for external analysis.

The trace from a PE or process can be exported off chip for analysis by external tools.

D4.1.2 Self-hosted Trace

Self-hosted trace is used for various purposes, including:

Non-invasive single stepping:

The trace provides a history of execution similar to that obtained by single-stepping through code.

Failure logging:

This is similar to a stack trace dump when a failure occurs.

Performance analysis:

The trace might be used with other trace sources or performance analysis units to analyze program performance.

D4.1.3 External debug

External debug is commonly used in trace applications that require long-term logging of behavior. In addition, external debug is more likely to be used when the impact of PE tracing on system performance must be minimized.

For example, external debug might be used:

• For debugging real-time systems.
• When analyzing programs that do not frequently vary their behavior.
• For debugging software, where a history of execution is required up to the point of failure.

Exporting the trace off-chip usually involves one of the following methodologies:

**Real-time continuous export**

This can be done using either:

- A dedicated trace port capable of sustaining the bandwidth of the trace.
- An existing interface on the SoC, such as a USB or other high-speed port.

Use of a dedicated trace port means that the trace can be exported off-chip with zero or minimum effect on system behavior. An existing interface is usually used when system constraints, such as cost or package size, mean that a dedicated trace port is not possible. However, use of an existing interface might affect system behavior, because both trace and normal interface traffic use the same port.

**Short-term on-chip capture with subsequent low speed export**

This option is used when a low-cost method of exporting the trace is required, or when system constraints prevent real-time continuous export. The trace output from the trace unit is stored temporarily on-chip, and then exported using either:

- An existing debug port on the SoC, such as a JTAG-DP or SW-DP.
- Another existing interface on the SoC, such as USB.

Typically, the temporary storage is a circular buffer. If the buffer is full, newer trace overwrites older trace, which means that the buffer always contains the most recent trace. In SoCs that employ Arm CoreSight technology, a dedicated Embedded Trace Buffer (ETB) is provided for the on-chip capture of trace.

**D4.1.4 Trace output**

The trace unit outputs the trace byte stream to one or more of the following:

- The Trace Buffer Unit.
- A CoreSight subsystem, through an AMBA ATB interface.
- One or more IMPLEMENTATION DEFINED interfaces.

If the Trace Buffer Unit is enabled, the trace byte stream is output solely to the Trace Buffer Unit.

If the Trace Buffer Unit is disabled, the trace byte stream is output to one or more of the other interfaces.

If the trace output is captured by a single capture point dedicated to a PE, and does not mix with any other trace streams, then the value programmed into TRCTRACEIDR.TRACEID does not need to be unique among all values programmed into all trace units in the system. For example, the value 0x01 could be programmed into all trace units that have their own dedicated trace capture point.

- One example of a dedicated trace capture point is the Trace Buffer Unit. When FEAT_TRBE is implemented and enabled, the value of TRCTRACEIDR.TRACEID can be the same for all PEs that are using the Trace Buffer Extension. For more information, see Chapter D6 *The Trace Buffer Extension*.
- Another example of a dedicated trace capture point is a CoreSight ETR connected through an AMBA ATB interface and dedicated to a PE. If the AMBA ATB interface connects to the ETR without mixing with any other trace streams, TRCTRACEIDR.TRACEID does not need to be unique when using the ETR, and when ensuring the trace does not go to any other trace capture points.

If an AMBA ATB interface is implemented, the trace unit must support all of the following:

- ATB triggers, indicated by TRCIDR5.ATBTRIG.
- A 7-bit trace ID, indicated by TRCIDR5.TRACEIDSIZE.

Arm recommends that if the trace unit implements either an AMBA ATB interface, or an IMPLEMENTATION DEFINED interface for trace output, then the path from the interface is not affected by a Warm reset of the PE. This ensures tracing is possible through a Warm reset of the PE, which is useful for low level debugging scenarios.
While all trace outputs are disabled, the trace unit considers any generated trace data as having been output.

### D4.1.5 Trace sessions

At any one time, the trace unit is either enabled or disabled. For more information on the possible states of the trace unit, see *Trace unit programming states*.

A *trace session* is the period between the trace unit becoming enabled, and when the trace unit next becomes disabled.

While the trace unit is enabled, the ViewInst() function is either *active* or *inactive*. While ViewInst() is *active*, the trace unit generates trace for instructions that are executed, unless trace generation is *inoperative*.

Trace generation is *operative* while neither of the following are true:
- The trace unit is disabled.
- The trace unit is recovering from a trace unit buffer overflow.

Whether ViewInst() is *active* or *inactive* is independent of whether trace generation is *operative* or *inoperative*.

Trace generation *becomes operative* when trace generation transitions from being *inoperative* to *operative*, and occurs when any of the following occur:
- When the trace unit transitions from being disabled to being enabled.
- When the trace unit recovers from a trace unit buffer overflow.

Trace generation *becomes inoperative* when trace generation transitions from being *operative* to *inoperative*, and occurs either:
- When the trace unit transitions from being enabled to being disabled.
- When the trace unit encounters a trace unit buffer overflow.

When the trace unit is unable to generate at least one trace packet which is required by the architecture, a *trace unit buffer overflow* occurs.

A trace unit buffer overflow is typically caused when any buffering in the trace unit is unable to receive more trace packets. Inability to receive trace packets is often caused by being unable to sustain output of trace packets to any trace capture infrastructure.

Note: A trace unit buffer overflow is independent of the Trace Buffer Unit filling or wrapping a trace buffer in memory. However a trace unit buffer overflow might be caused by the Trace Buffer Unit rejecting trace data due to insufficient capacity, and the limit of any trace unit internal buffers is subsequently reached.

### D4.1.6 Trace unit programming states

The trace unit is always in one of the states shown in Figure D4-1 on page D4-4767 and Table D4-1 on page D4-4767.
When the trace unit becomes enabled, the trace unit transitions from the Idle state to the Enabling state.

The trace unit transitions from the Enabling state to the Running state in a finite amount of time.

When the trace unit becomes disabled, the trace unit transitions from the Running state to the Unstable state.

The trace unit transitions from the Unstable state to the Stable state in a finite amount of time.

While the trace unit is in the Stable and Idle states, the states of the following fields do not change other than by direct writes or external writes:

- TRCVICTLR.SSSTATUS.
- TRCSEQSTR.STATE.
- TRCCNTVR<\<n>.VALUE.
- TRCSSCSR<\<n>.STATUS.
- TRCRSR.EVENT.
- TRCRSR.EXTIN.
- TRCRSR.TA.

The trace unit programmers’ model state can be safely read when in any of the Stable or Idle states.
When the trace unit becomes fully idle and both of the following are true, the trace unit transitions from the Stable state to the Idle state:

- The trace unit is drained of any trace.
- With the exception of the programming interfaces, all external interfaces on the trace unit are quiescent.

While the trace unit is not in the Idle state, direct writes and external writes to the trace unit registers are constrained unpredictable, except for the following registers:

- TRCPRGCTRL.
- TRCLAIMSET.
- TRCLAIMCLR.

This constrained unpredictable behavior is one of the following:

- The write is ignored.
- The register takes an unknown value.

The trace byte stream might also be corrupted and analysis of the byte stream might be impossible.

While the trace unit is not in the Idle or Running states, changing the value of TRCPRGCTRL.EN is constrained unpredictable.

This constrained unpredictable behavior is one of the following:

- The write is ignored.
- The register takes an unknown value.

For more information, see:

- Behavior on enabling on page D4-4800.
- Behavior on disabling on page D4-4801.
- Access permissions on the corresponding register page.

Figure D4-2 on page D4-4769 shows the procedure that must be used when programming the trace unit registers using the External Debugger interface.
Figure D4-2 External Debugger Interface programming procedure

Figure D4-3 on page D4-4770 shows the procedure that is used when programming the trace unit registers using the System instruction interface.
Figure D4-3 System instruction programming procedure
D4.2 Programmers’ model

D4.2.1 Accessing ETE registers

The ETE architecture provides registers for programming the ETE trace unit and reading back the programmed settings. These registers can be accessed by using one or more of the following access mechanisms:

• An external debugger interface, for use by an external debugger.
• System instructions, for use by self-hosted software running on the PE being traced.

When register accesses occur simultaneously from multiple access mechanisms, the trace unit behaves as if all accesses occur atomically in any order.

External debugger interface

The external debugger interface defines an address-mapped peripheral that occupies 4KB of address space.

Note: The PE does not have to be in Debug state to program the ETE registers.

The memory access sizes that are supported by any peripheral are IMPLEMENTATION DEFINED by the peripheral.

The trace unit supports the following access sizes:

• Word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register mapped to a doubleword-aligned pair of adjacent 32-bit locations.
• Doubleword-aligned 64-bit accesses to access 64-bit registers mapped to a doubleword-aligned pair of adjacent 32-bit locations. The order in which the two halves are accessed is not specified.

Note: This means that a system implementing the Debug registers using a 32-bit bus, such as AMBA APB in CoreSight systems, with a wider system interconnect must implement a bridge between the system and the debug bus that can split 64-bit accesses.

All registers are only single-copy atomic at word granularity.

The trace unit does not support the following accesses:

• Byte.
• Halfword.
• Unaligned word. These accesses are not single-copy atomic at word granularity.
• Unaligned doubleword. These accesses are not single-copy atomic at doubleword granularity.
• Doubleword accesses to a pair of 32-bit locations that are not a doubleword-aligned pair forming a 64-bit register.
• Quadword or higher.
• Exclusives.

The behavior of the accesses that are unsupported by the trace unit is CONSTRAINED UNPREDICTABLE and is one of the following:

• Accesses generate an External abort, and writes set the accessed register or registers to an UNKNOWN value or values.
• Reads return UNKNOWN data and writes are ignored.
• Reads return UNKNOWN data and writes set the accessed register or registers to an UNKNOWN value. This is the Arm preferred behavior.

For accesses from the external debugger interface which return an error response when AllowExternalTraceAccess() returns FALSE, EDPRSR.STAD is only set to 1 when this is the highest priority cause of the error. The following causes are higher priority than AllowExternalTraceAccess():

• The trace unit Core power domain is powered down.
• The OS Lock is locked and the register is defined to return an error response due to the OS Lock being locked.
RKQMKX

Accesses from the external debugger interface to unimplemented or Reserved registers behave as follows:

- For accesses in the range of offsets 0xF00 to 0xFFC, the access behaves as RES0H.
- For accesses in the range of offsets 0x000 to 0xEFC when the OS Lock is locked, the access behaves as RES0H or returns an error.
- For accesses in the range of offsets 0x000 to 0xEFC when the OS Lock is unlocked and MDCR_EL3.ETAD is 0, the access behaves as RES0H.
- For Secure accesses in the range of offsets 0x000 to 0xEFC when the OS Lock is unlocked and MDCR_EL3.ETAD is 1, the access behaves as RES0H.
- For Non-secure accesses in the range of offsets 0x000 to 0xEFC when the OS Lock is unlocked and MDCR_EL3.ETAD is 1, the access behaves as RES0H or returns an error.

RWXKDP

Reads of write-only registers are Reserved.

RSVSNR

Writes to read-only registers are Reserved.

IWTJFD

For accesses that return an error, see Behavior of a not permitted memory-mapped access on page H8-10363 for more details on how this error is handled.

System instructions

RJVLBN

MRS instructions with op0 == 0b10 and op1 == 0b001 read from trace unit registers.

RFYRRX

MSR instructions with op0 == 0b10 and op1 == 0b001 write to trace unit registers.

RVGVTS

Instructions with Crn >= 0b1000 are not allocated for accessing trace unit registers.

RSGPQB

While the PE is in EL0, all System instruction accesses to trace unit registers are UNDEFINED.

IWCXDT

For consistency with the Arm architecture, System instruction accesses to registers which are not implemented generate an Undefined Instruction exception. These accesses include:

- Writes to read-only registers.
- Reads from write-only registers.
- Accesses to registers which are not present due to the configuration of the trace unit.

D4.2.2 Synchronization of register updates

AArch64 System registers

IKWCGH

As defined in Synchronization requirements for AArch64 System registers on page D17-5547, direct writes to trace unit registers are only guaranteed to be visible to execution after a subsequent Context synchronization event.

IFNZZH

Direct reads of trace unit registers while the trace unit is not in the Stable or Idle states are not guaranteed to contain the results of the trace operation of execution previous to the direct read operation.

IOPVJQ

As defined in Synchronization requirements for AArch64 System registers on page D17-5547, a direct write to a register using the same register number that was used by a previous System instruction to write it, the final result is the value of the second write, without requiring any context synchronization between the two write instructions.

IWLGQK

As defined in Synchronization requirements for AArch64 System registers on page D17-5547, a direct read of a register using the same register number that was used by an earlier direct write is guaranteed to observe the value that was written, without requiring any context synchronization between the write and read instructions.

SHSXGZ

Context synchronization events are important when changing the value of TRCPRGCTRL.EN or when changing the OS Lock. After writing to TRCPRGCTRL to change the value of TRCPRGCTRL.EN, one read of TRCSTATR is required before programming any other registers. A Context synchronization event is required between writing to TRCPRGCTRL and reading TRCSTATR. If multiple reads of TRCSTATR are required, a Context synchronization event is required between each read of TRCSTATR to ensure any change to the trace unit state is observed.
When indirect writes or external writes to the registers in Table D4-2 on page D4-4773 occur, both of the following can observe the writes:

- Direct reads in finite time without explicit synchronization.
- Subsequent indirect reads without explicit synchronization.

### Table D4-2 Registers with a guarantee of observability

<table>
<thead>
<tr>
<th>Register</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRCCLAIMSET</td>
<td>Claim Tag Set Register</td>
</tr>
<tr>
<td>TRCCLAIMCLR</td>
<td>Claim Tag Clear Register</td>
</tr>
<tr>
<td>TRCCNTVR&lt;n&gt;</td>
<td>Counter Value Register &lt;n&gt;</td>
</tr>
<tr>
<td>TRCSEQSTR</td>
<td>Sequencer State Register</td>
</tr>
<tr>
<td>TRCSSCSR&lt;n&gt;</td>
<td>Single-shot Comparator Control Status Register</td>
</tr>
</tbody>
</table>

When the trace unit becomes enabled or disabled as a result of a direct write, for any instruction in program order before the direct write, the new state of the trace unit does not affect trace operations.

When the trace unit becomes enabled or disabled as a result of a direct write, for any instruction after a Context synchronization event in program order after the direct write, the new state of the trace unit takes effect for any trace operations.

Note: The registers which control whether the trace unit is enabled or disabled are:

- TRCPRGCTRLR.
- OSLAR_EL1.

Arm recommends that a Context synchronization event is executed after programming the Trace Unit registers, to ensure that all updates are synchronized to the trace unit before normal code execution resumes.

When a Context synchronization event occurs while the trace unit is in the Idle or Stable states, and at no other time, indirect writes to the trace unit registers are guaranteed to be visible to direct reads.

When either of the following events occurs, and at no other time, indirect writes to the trace unit registers are guaranteed to be visible to indirect reads or external reads:

- The trace unit transitions into the Stable state.
- The trace unit transitions into the Idle state.

The trace unit functions perform indirect writes to the registers and indirect reads from the registers in architectural order.

### External debugger registers

As defined in the *Synchronization of changes to the external debug registers on page H8-10358*, this section refers to accesses from the external debug interface as *external reads* and *external writes*.

As defined in *Synchronization of changes to the external debug registers on page H8-10358*, explicit synchronization is not required for an external read or an external write by an external agent to be observable to a following external read or external write by that agent to the same register using the same address, so explicit synchronization is never required for registers that are accessible only in the external debug interface.

As defined in *Synchronization of changes to the external debug registers on page H8-10358*, when an external write to a register using the same register number that was used by a previous external write is performed, the final result is the value of the second write, without requiring any context synchronization between the two write accesses.

The trace unit does not require explicit synchronization for an external write to be visible to indirect reads.
RDYRZC  The trace unit does not require explicit synchronization for an external write to be visible to subsequent external reads.

IRDFSX  As defined in Synchronization of changes to the external debug registers on page H8-10358, explicit synchronization is required for an external write to be visible to direct reads.

RMMYRJ  While the trace unit is in the Stable or Idle states, the trace unit does not require explicit synchronization for indirect writes to be visible to external reads.

**Synchronization and the authentication interface**

RWYWMJ  Changes to the authentication interface are indirect writes to TRAUTHSTATUS by the controller of the authentication interface. It is IMPLEMENTATION DEFINED whether a change on the authentication interface is guaranteed to be observable to an external read of the register only after a Context synchronization event or in finite time.
D4.3 Trace elements

Trace elements form an AST which is used to describe the control flow of program execution. Different sequences of the trace elements can be used to imply the same operation. In this way FEAT_ETE can be used by different micro-architectures. This is similar to the approach used in previous trace protocols. For more information, see the ARM® Embedded Trace Macrocell Architecture Specification, ETMv4 (ARM IHI 0064).

A trace unit compresses the information on the execution of the PE and outputs a trace byte stream that comprises multiple packets of encoded data. The compression techniques that are used include:

- **Not having a trace element for every executed instruction in the instruction trace element stream**
  - Instead, the trace unit generates *P0 elements* in the trace element stream when certain types of instruction are executed. These certain types of instructions are known as *P0 instructions*. A *P0 element* acts as a signpost in the program flow, indicating that execution is proceeding along a given path.
  - Consequently, the stream of *P0 elements* implies the execution of a greater number of instructions, and a trace analyzer can reconstruct the stream of instructions that are executed between *P0 elements* by using the *P0 element stream* and the program image.

- **Multiple trace elements in single packets**
  - Common sequences of trace elements are encoded into single packets.

- **Removal of program addresses from the trace element stream**
  - The trace analyzer can infer the addresses from the program image and previous history. This includes the targets of direct branch instructions, where the target address is encoded in the instruction itself.

- **Removal of predictable trace elements**
  - Some trace elements can be removed from the AST representation if the contents of the trace element can be predicted by previous control flow choices in the software flow. For example the Target Address element for returning from a subroutine might not be required if the branch to the subroutine has been traced.

D4.3.1 Layer model

FEAT_ETE is based on a layer model. Each layer deals with a unique aspect of tracing the PE.

![Layer model for compression and decompression](image)

The transport layer either provides either:
- A path off chip.
- A path to a memory buffer for trace to be stored.

Layer 1
Layer 1 provides compression by:
- Grouping trace elements together to form packets.
- Removing trace elements that can be implied.
- Compression against previous values.
- Leading zero compression.
- Reordering of trace elements.

**Layer 2**
Layer 2 provides:
- Speculation resolution.
- Transactional Memory resolution.

**Layer 3**
At layer 3:
- PE behavior is converted into trace elements.
- Compression is achieved by removing the trace elements which can be predicted using the program image:
  - Direct branch target addresses.
  - Return stack optimization.
- Knowledge of the application is required in order to decompress. Processes that modify the instruction opcodes require additional information to allow debugging.

**D4.3.2 Trace protocol synchronization**

The trace byte stream of a trace unit is typically stored in a circular buffer where, if the buffer is full, newer trace overwrites older trace. To ensure that a trace stream can be analyzed when the trace has been stored in a circular buffer, a trace unit must periodically generate trace protocol synchronization points in the trace byte stream.

The following trace elements or packets are used to provide synchronization information in the different layers.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 3</td>
<td>Context element and Target Address element</td>
</tr>
<tr>
<td>Layer 2</td>
<td>Trace Info element</td>
</tr>
<tr>
<td>Layer 1</td>
<td>Trace Info packet</td>
</tr>
<tr>
<td>Transport Layer</td>
<td>Alignment Synchronization packet</td>
</tr>
</tbody>
</table>

Whenever a trace analyzer receives a Trace Info packet, the trace analyzer receives information about the current state of the trace. However, the trace analyzer cannot begin analysis of program execution until it knows the context in which instructions are being executed and it has an instruction address to start analysis from.

When a Trace Info element is generated, the trace unit generates a Context element and a Target Address element soon after the Trace Info element.

Note: There are common use cases where the ratio between the number of bytes associated with trace protocol synchronization and other trace bytes increases significantly, resulting in a degradation of the usability of the trace. Therefore Arm recommends that trace protocol synchronization only occurs when required.

**Non-periodic trace protocol synchronization**

When the trace unit becomes operative, non-periodic trace protocol synchronization occurs.
When non-periodic trace protocol synchronization occurs, the trace unit generates an Alignment Synchronization packet in the trace byte stream before any other trace packets are generated.

When non-periodic trace protocol synchronization occurs, the trace unit generates a *Trace Info element* in the trace element stream before any other trace elements are generated, except *Event elements*.

Arm recommends that if a trace protocol synchronization request occurs while ViewInst() is inactive, the Alignment Synchronization packet is not output in the trace byte stream until just before either:

- ViewInst() becomes active.
- An Event packet is output.

### Periodic trace protocol synchronization

The trace unit can be programmed to generate trace protocol synchronization requests on a periodic basis, so that the trace element streams and the trace byte streams can be analyzed when stored in a circular trace buffer. TRCSYNCPR.PERIOD controls periodic trace protocol synchronization requests.

Periodic trace protocol synchronization can also be requested by the trace capture infrastructure, for example if a trace protocol synchronization request is received on an Arm AMBA ATB interface. For more information on the Arm AMBA ATB interface, see the *AMBA® ATB Protocol Specification (ARM IHI 0032)*.

When periodic trace protocol synchronization is requested, either by TRCSYNCPR.PERIOD or by other sources, the trace unit performs periodic trace protocol synchronization.

When periodic trace protocol synchronization occurs, the trace unit generates an Alignment Synchronization packet and then generates a *Trace Info element*.

Arm recommends that an Alignment Synchronization packet is only output in the trace byte stream if other trace packets have been output since the previous Alignment Synchronization packet. This reduces the risk of a circular buffer filling and overwriting trace.

If two or more periodic trace protocol synchronization requests occur, and no trace is generated between these two requests, then Arm recommends that a non-periodic trace protocol synchronization occurs before any further trace is generated. This ensures that when tracing has been inactive for a long period of time, the trace stream is fully synchronized when tracing is re-activated.

### Synchronization of instruction trace

When non-periodic trace protocol synchronization occurs, the trace unit generates a *Context element* and a *Target Address element* before any *P0 elements* are generated, to provide the trace analyzer with *Context information* and *Address information*.

When periodic trace protocol synchronization occurs, and ViewInst() is active when the corresponding *Trace Info element* is generated, the trace unit generates a *Context element* and a *Target Address element* which provide the *Context information* and *Address information* for the target of the most recent non-canceled *P0 elements*.

Note: If the trace unit generates the *Context element* and *Target Address element* immediately after the *Trace Info element*, then the most recent non-canceled *P0 elements* might have occurred before the *Trace Info element*. 
When periodic trace protocol synchronization occurs, and ViewInst() is inactive when the corresponding Trace Info element is generated, when ViewInst() becomes active and a Trace On element is generated, the trace unit generates a Context element and a Target Address element before any Atom elements, Q elements, or Exception elements are generated. This provides the trace analyzer with Context information and Address information.

If a Cancel element cancels any P0 elements before a Trace Info element, then the trace analyzer discards all of the following:

- The canceled P0 elements.
- The Trace Info element.
- All elements after the Trace Info element, up to and including the Cancel element. This includes any Context elements or Target Address elements.
Note: In this scenario, information from the canceled Trace Info element can still be used.

**Figure D4-7 Example of Target Address element after Trace Info element in a mispredicted region.**

When a Cancel element is generated which cancels any P0 elements before a Trace Info element, the trace unit generates a new Context element and a new Target Address element, which indicate the target of the most recent P0 element that has not been canceled.

The Target Address element and Context element might indicate the target of a P0 element from before the Trace Info element, or might be delayed until after the next P0 element, and therefore indicate the target of that P0 element.

Note: If the trace unit generates the new Context element and Target Address element prior to the next new P0 element, then this might prevent the indication of execution of some instructions before the Trace Info element.

If the Cancel element cancels all P0 elements after a Trace Info element, but no P0 elements prior to the Trace Info element, then it might be necessary for the trace unit to immediately generate a Context element and Target Address element. This is because a Context element and Target Address element might have been present in the trace element stream after the Trace Info element, and those Context element and Target Address element are now discarded.

### D4.3.3 Speculation in the trace element stream

FEAT_ETE supports the correction of trace. This might be because of:

- The tracing of speculative execution of instructions by a PE.
- For some implementations, the tracing of the Transactional Memory Extension.

A FEAT_ETE trace unit traces speculatively-executed instructions in the same way as all other instructions, so that both speculatively-executed instructions and architecturally-executed instructions appear in the instruction trace element stream. This means that some of the program execution information that is shown in the trace element stream might be incorrect, because some of the speculatively executed instructions might be mis-speculated.

Note: The level of speculation that is revealed in the trace is IMPLEMENTATION SPECIFIC.
The Embedded Trace Extension
D4.3 Trace elements

IXLLKT The trace unit resolves this speculation by generating trace elements to confirm the status of each instruction in the instruction trace element stream. That is, the trace unit generates trace elements to show whether each instruction has been committed for execution, or canceled because of mis-speculation. This means that a trace analyzer does not know the status of a traced instruction until the trace analyzer receives a trace element that indicates whether the instruction has been committed for execution, or canceled because the instruction was mis-speculated.

RZJKY When speculatively-executed instructions are traced, the trace unit subsequently generates trace elements that indicate whether the instructions have been committed for execution, or have been canceled.

IKYXKZ A trace analyzer takes the appropriate action, which might involve canceling some trace elements, to determine the actual program execution.

IGGFML Trace elements that resolve the status of a traced instruction are called speculation resolution elements. For more information on speculation elements, see Speculation Resolution Elements on page D4-4787.

RKYGRF When trace is generated for speculative execution, for mis-speculated execution, the trace unit does not trace any information that cannot be accessed by software executing at the same or at a lower level of privilege than the mis-speculated execution.

ROHQLY When a Context synchronization event is speculated as being taken or executed, the trace unit does not generate trace for any speculative execution after the Context synchronization event until the Context synchronization event is resolved.

RLWICQ When a speculated Context synchronization event is resolved as being not taken or not executed, the trace unit does not generate trace for mis-speculated execution that occurred after the Context synchronization event.

RYGSGJ When an exit from a Trace Prohibited region is speculated as being taken, the trace unit does not generate trace for any speculative execution after the exit from the Trace Prohibited region, until the exit from the Trace Prohibited region is resolved.

RSRLCG When a speculated exit from a Trace Prohibited region is resolved as being not taken, the trace unit does not generate trace for mis-speculated execution that occurred after the exit from a Trace Prohibited region.

Tracing transactions

IKBTIL The Transactional Memory Extension defines the Transactional state. For instructions executed in Transactional state, the trace stream indicates which instructions are executed in Transactional state, and provides indicators for a trace analyzer to determine whether the transaction was successful or failed.

IFWGBM If the instruction is executed in Transactional state then the result of the instruction is not known until the transaction succeeds or fails. Transactions can be of an arbitrary length and can be nested, so the ETE architecture does not guarantee an entire transaction is traced, if any of the transactions is traced.

IVJTLG The execution of transactions is represented in the trace element stream by 3 elements:

1. Transaction Start element.
2. Transaction Commit element.
3. Transaction Failure element.

These provide markers in the trace element stream to indicate the sections which represent transactions. The Transaction Start element indicates that any following instructions are executed in Transactional state. When the PE leaves Transactional state, either the Transaction Commit element or Transaction Failure element are traced to indicate the resolution of the transaction.

IQZNBZ An entry to Transactional state might be traced using a Transaction Start element and the subsequent exit from Transactional state might be traced, without tracing any execution in Transactional state. There might have been no execution in Transactional state, or the trace unit might have been programmed to not trace such execution.

Implementation flexibility

RVVFQZ If no speculation in the trace element stream is implemented, then TRCIDR8.MAXSPEC is 0 and TRCIDR0.COMMTRANS indicates that the Transaction Start element is a P0 element.
Filtering of trace

The ETE architecture supports filtering of the trace within a transaction.

Filtering of a transaction can be due to any of the following:

- The ViewInst function.
- Prohibited regions.
- Asynchronous events.

Due to filtering the start of the transaction might not necessarily be traced. See the Transaction Start element for details.

Due to filtering the end of a transaction might not necessarily be traced. See the Transaction Commit element and Transaction Failure element for details.

If an instruction is traced which was executed in Transactional state, then the trace analyzer must be aware, so that the effect of the instructions executed in the Transactional state can be determined.

When an instruction is traced and the PE is in Transactional state, the trace unit traces the result of the transaction unless any of the following occur:

- The trace unit becomes disabled.
- A trace unit buffer overflow occurs.
- The PE enters a Trace Prohibited region.

In the above scenarios, the trace unit generates a Transaction Failure element, and the resolution of the transaction is UNKNOWN.

Trace element stream

A Trace element stream is a sequence of elements which describe:

- The software control path of PE execution traced by the trace unit.
- Event-based trace.
- Temporal information.

A Trace Info element provides a point in the trace element stream where analysis of the trace element stream can begin.

Trace Info elements include setup information about:

- The static trace programming that does not change during a trace session, including:
  - Whether cycle counting is enabled, and if enabled, the cycle count threshold.
- Dynamic information that might change during a trace session, such as:
  - The speculation depth. This indicates how many unresolved P0 elements were traced before the Trace Info element.
  - Whether the trace unit has traced that the PE is executing in Transactional state.

P0 element

P0 elements imply the execution of instructions.

P0 elements are generated speculatively and must be either committed or canceled. For more information on speculation elements, see Speculation Resolution Elements on page D4-4787.

P0 elements must be generated in sequential execution order.
Atom element

An Atom element implies that one or more instructions have been executed, up to and including the next P0 instruction. Only certain instructions generate an Atom element. See Instruction and exception classification on page D4-4791 for more information on these instructions.

The Atom element is one of the following types:
- E Atom.
- N Atom.

The meaning of the type of an Atom element depends on the instruction it is encoding. For example, branch instructions are represented as an E Atom element if the branch was taken and an N Atom element if not taken.

Exception element

An Exception element indicates a change in program flow which cannot be calculated by the analysis of the program image, or which is caused by an instruction which is not a P0 instruction. Such a change in program flow is described as an Exceptional occurrence.

An Exceptional occurrence consists of the following:
- PE Architectural exceptions.
- ETE defined exceptions.
- IMPLEMENTATION DEFINED exceptions.

Note: Transaction failure is not classified as an Exceptional occurrence, although it is traced using an Exception packet.

An Exception element indicates:
- That an Exceptional occurrence has occurred.
- The type of Exceptional occurrence.
- The virtual address where the Exceptional occurrence was taken from, also known as the preferred exception return address.

The instruction set for the preferred exception return address for an Exception element is one of the following:
- AArch64 A64.
- AArch32 A32.
- AArch32 T32.

An Exception element is a P0 element.

PE Architectural exceptions

The following exception types are used to indicate PE Architectural exceptions:
- IRQ.
- FIQ.
- Trap.
- Call.
- Inst fault.
- Data fault.
- Inst debug.
- Data debug.
- Alignment.
- System Error.
- Debug halt.
See *Instruction and exception classification* on page D4-4791 for information on the mapping between the PE Architectural exceptions and these exception types.

Table D4-4 on page D4-4783 defines the preferred exception return address for each exception type for PE Architectural exceptions.

Table D4-4 Preferred exception return address for PE Architectural exceptions

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Preferred exception return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ</td>
<td>Instruction after the last executed instruction</td>
</tr>
<tr>
<td>FIQ</td>
<td>Instruction after the last executed instruction</td>
</tr>
<tr>
<td>Trap</td>
<td>For a trapped instruction or UNDEFINED instruction, the preferred exception return address is the address of the instruction. For a trapped exception, the preferred exception return address is the address of the instruction that caused the exception.</td>
</tr>
<tr>
<td>Call</td>
<td>Instruction after the call instruction</td>
</tr>
<tr>
<td>Inst fault</td>
<td>Instruction that caused the exception</td>
</tr>
<tr>
<td>Data fault</td>
<td>Instruction that caused the exception</td>
</tr>
<tr>
<td>Inst debug</td>
<td>Instruction that caused the exception</td>
</tr>
<tr>
<td>Data debug</td>
<td>Instruction that caused the exception</td>
</tr>
<tr>
<td>Alignment</td>
<td>Instruction that caused the alignment exception</td>
</tr>
<tr>
<td>System Error</td>
<td>Instruction after the last executed instruction</td>
</tr>
<tr>
<td>Debug halt</td>
<td>The instruction after the last executed instruction, that is, the value loaded into the DLR register.</td>
</tr>
</tbody>
</table>

The nature of System Error means that execution might not complete up to the preferred exception return address, or it might perform some operations after the preferred exception return address. This behavior is IMPLEMENTATIONDEFINED and might vary depending on the cause of the exception.

When an imprecise System Error exception occurs, the preferred exception return address is the address stored in the relevant ELR when the exception is taken.

When a System Error exception occurs, the trace analyzer must be aware that the preferred exception return address might not indicate the exact point at which program execution was interrupted. The trace analyzer should not rely on the preferred exception return address for inferring exactly which instructions were executed. This behavior only occurs for imprecise System Error exceptions.

When an imprecise Debug halt exception occurs, the preferred exception return address is the address stored in DLR or DLR_EL0 when the exception is taken.

When an imprecise Debug halt exception occurs, the trace analyzer must be aware that the preferred exception return address might not indicate the exact point at which program execution was interrupted. The trace analyzer should not rely on the preferred exception return address for inferring exactly which instructions were executed. An imprecise Debug halt exception can only occur under direct control of a debugger, usually by controlling EDRCR.CBRRQ.

**ETE defined exceptions**

In addition to the Arm Architectural exceptions, the ETE specifies the following Exceptional occurrences that are traced using Exception elements:

- PE Reset, which indicates that a PE Warm reset has occurred.
Table D4-5 on page D4-4784 defines the preferred exception return address for each exception type for ETE defined exceptions.

### Table D4-5 Preferred exception return address for ETE defined exceptions

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Preferred exception return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE Reset</td>
<td>UNKNOWN</td>
</tr>
</tbody>
</table>

When a PE Reset occurs, the preferred exception return address and context are UNKNOWN. Therefore for an Exception element indicating a PE Reset the preferred exception return address and context are UNKNOWN. No instruction execution is indicated between the previous P0 element and the Exception element.

When an Exception element indicating a PE Reset occurs:
- The target address and target context of the previous P0 element might be UNKNOWN.
- If there are no P0 elements between a Trace On element and the Exception element, then the initial address and context after the previous Trace On element might be UNKNOWN.

**IMPLEMENTATION DEFINED exceptions**

ETE defines some exception types which are IMPLEMENTATION DEFINED, including but not limited to:
- ECC error correction.
- Generic replay of program execution.

The use of the IMPLEMENTATION DEFINED exceptions is optional and IMPLEMENTATION DEFINED. IMPLEMENTATION DEFINED exceptions are not required to be traced but are intended to be used to simplify tracing of certain micro-architectural situations.

In general, the preferred exception return address is the address of the instruction after the last executed instruction, before the exception occurs.

**Source Address element**

The Source Address element indicates execution up to and including a provided P0 instruction address, and indicates the P0 instruction is taken. All P0 instructions except the final P0 instruction are not taken, which means that explicit N Atom elements are not required to be traced for those P0 instructions. A Source Address element indicates both of the following for the final P0 instruction:
- The instruction set.
- The virtual address of the instruction.

The instruction set for a Source Address element is one of the following:
- AArch64 A64.
- AArch32 A32.
- AArch32 T32.

A Source Address element is a P0 element.

**Q element**

A Q element belongs to the P0 element group in the instruction trace element stream, and must be explicitly resolved or canceled.

A Q element can optionally include a number, M. The number is a count of the instructions that are executed since the most recent P0 element, which might be a Q element. If it does not include a count of instructions, then the number of instructions that are executed since the most recent P0 element is UNKNOWN.

The trace unit generates Q elements in the program order in which they occur, and the trace protocol encode and decode process maintains this order.
A Q element does not imply Exceptional occurrences.

When a Q element implies an Exception return instruction which is taken, that instruction is the last instruction that is implied by the Q element.

When a Q element implies an executed ISB instruction, this is the last instruction implied by the Q element if execution continues from a new context after the ISB.

When execution continues from a new context after a Q element is generated, the trace unit generates a Context element after the Q element.

The Context element might be generated before or after the Target Address element that is also required after the Q element.

If a context change occurs at a point that is not a Context synchronization event, then the last instruction that is implied by a Q element must be the last instruction that is executed with the old context. The trace unit can then generate a Context element after the Q element to indicate the new context.

Transaction Start Element

TRCIDR0.COMMTRANS indicates whether the Transaction Start element is a P0 element. See Transaction Start element on page D4-4790 for more information about the Transaction Start element.

Virtual Address Space Element

Trace On element

A Trace On element indicates a discontinuity in the trace element stream. The trace unit inserts a Trace On element after a gap in the generation of the trace element stream:

• When the trace generation becomes operative and before any P0 elements.
• If some instructions are filtered out of the trace.
• The first traced instructions after:
  — A Trace Prohibited region.
  — The PE leaves Debug state.
• When instruction trace is lost because a trace unit buffer overflow occurs.

When a Trace On element is generated, the trace unit generates a Target Address element before the next P0 element.

When a Trace On element is generated, the trace unit generates a Context element before the next Atom element, Exception element or Q element, to indicate where tracing starts, unless the context has not changed since the previous Context element was output.

When the first Trace On element is generated, the trace unit outputs the corresponding Context element before the first P0 element.

Target Address element

A Target Address element indicates both of the following for the next instruction to be executed:

• The instruction set.
• The virtual address of the instruction.

The instruction set for a Target Address element is one of the following:

• AArch64 A64.
• AArch32 A32.
• AArch32 T32.

The trace unit generates Target Address elements in program order relative to other P0 elements.
The Embedded Trace Extension

**D4.3 Trace elements**

Target Address element values can be corrected by another Target Address element if both Target Address elements are generated before the next P0 element or Trace On element.

**Context element**

The Context element indicates the execution context for the next instruction to be executed.

The Context element provides the following Context information:

- The Security state.
- The Exception level.
- Whether the PE is executing in AArch64 state or AArch32 state.

The Context element can optionally provide the following Context information:

- The Context identifier.
- The Virtual context identifier.

The trace unit generates Context elements in program order relative to P0 elements.

**D4.3.7 Temporal elements**

Temporal elements provide information about the passage of time within the trace element stream. The following temporal elements are supported by ETE:

*The Cycle Count element*

Indicates the passage of PE clock cycles within the trace element stream.

*The Timestamp element*

Indicates the passage of time within the trace element stream.

*The Timestamp Marker element*

Indicates that the most recent P0 element or Event element has been timestamped and that a subsequent Timestamp element contains the timestamp value.

**Cycle Count element**

Each Cycle Count element is associated with a Commit element, and when a Commit element is generated, a Cycle Count element might also be generated.

Each Cycle Count element is associated with the most recent Commit element.

A Cycle Count element indicates the number of PE clock cycles between the two most recent Commit elements that both have an associated Cycle Count element.

Not every Commit element is required to have an associated Cycle Count element.

Cycle Count elements are generated in order relative to Commit elements.

**Timestamp element**

The Timestamp element inserts a global timestamp value into the trace element stream.

The source for timestamp reported in the timestamp element is controlled by:

- TRFCR_EL1.TS.
- TRFCR_EL2.TS.

A timestamp value of zero indicates that the timestamp value is UNKNOWN.

An UNKNOWN timestamp value might occur if the system does not support timestamping or if the timestamp is temporarily unavailable.
The Embedded Trace Extension
D4.3 Trace elements

The source for the payload of Timestamp elements is controlled by the TRFCR registers and the virtual timers. It is expected that these registers will be changed by context switch software. As a result it is possible that payloads of Timestamp elements might appear to have discontinuities, and even go backwards, if the source of the timestamp changes, or any context switching changes the System registers which control the timestamp value.

If FEAT_ETEv1p1 is implemented, when there has been a Timestamp Marker element before the Timestamp element, the Timestamp element contains the timestamp value of the most recent P0 element or Event element before the Timestamp Marker element.

If FEAT_ETEv1p1 is not implemented or if there has not been a Timestamp Marker element before the Timestamp element, the Timestamp element contains the timestamp value of the most recent P0 element or Event element before the Timestamp element.

If TRCIDR0.TSMARK is 1 and there is no previous Timestamp Marker element, the Timestamp element is for a P0 element or Event element which is before the start of the trace. This scenario might occur when trace analysis starts at a Trace Info element which is not the first Trace Info element, and the Timestamp Marker element was generated before the Trace Info element.

The requirement for a Timestamp Marker element for every Timestamp element is to avoid needing to indicate if there has been a Timestamp Marker element at a Trace Info point. This allows a trace analyzer to assume there is a Timestamp Marker element (or not) before the Trace Info, based on a static piece of information.

**Timestamp Marker element**

The Timestamp Marker element indicates the most recent P0 element or Event element has been timestamped, and that a Timestamp element will follow containing the timestamp value.

Timestamp Marker elements are generated in order with respect to P0 elements and Event elements.

Timestamp Marker elements are not canceled by Cancel elements.

A Cancel element might cause a P0 element to be canceled and if there is a Timestamp Marker element that is associated with that P0 element then the Timestamp Marker element is not associated with any P0 element. The Timestamp element which is associated with the Timestamp Marker element is unaffected, and is still usable for timestamping the approximate position in the trace stream.

If 2 Timestamp Marker elements occur without a Timestamp element between them, the oldest Timestamp Marker element is ignored.

If an Overflow element or Discard element occurs after a Timestamp Marker element and before a Timestamp element, the Timestamp Marker element is ignored.

If Timestamp Marker elements are generated by the trace unit, every Timestamp element must have a corresponding Timestamp Marker element generated before the Timestamp element.

The generation of Timestamp Marker elements is indicated in TRCIDR0.TSMARK.

**Speculation Resolution Elements**

The ETE architecture allows trace to be generated speculatively and then later committed or removed by the decompression process. Each P0 element is traced and is considered speculative until either committed by a Commit element or canceled by a Cancel element. This method of generating speculative trace allows for the tracing of speculative execution, including the tracing of transactions when the Transactional Memory Extension is implemented in the PE.

Speculation resolution elements provide a trace analyzer with information about which trace elements were correctly or incorrectly generated, and ensure the trace analyzer can reconstruct the program execution. The following speculation resolution elements are supported by ETE:

**The Mispredict element**

Corrects the most recent Atom element.
The **Cancel element**
Indicates that one or more P0 elements are canceled.

The **Commit element**
Indicates that one or more P0 elements are resolved for execution.

The **Discard element**
Removes all speculative P0 elements.

TRCIDR8.MAXSPEC specifies the maximum number of uncommitted P0 elements which can be discarded at a later stage using **Cancel elements**.

**Commit element**
A Commit element indicates that a number of unresolved P0 elements have been resolved for execution. The resolved P0 elements are the oldest P0 elements.

The Commit element resolves all types of P0 element.

Commit elements might be merged if the total number of P0 elements resolved is less than TRCIDR8.MAXSPEC. Commit elements are merged by adding their respective commit count values together.

Figure D4-8 shows an example operation for a Commit element:

![Figure D4-8 Commit Operation Example](image)

**Cancel element**
The Cancel element indicates the number of youngest unresolved and uncanceled P0 elements that are canceled from execution. A trace unit might cancel elements because of many reasons, including but not limited to:

- A P0 instruction is mis-speculated.
- An exception occurs.

The Cancel element cancels all types of P0 element.

Cancel elements might be merged if no P0 elements have been generated in between. Cancel elements are merged by adding their respective cancel numbers together.

Figure D4-9 on page D4-4789 shows an example operation for a Cancel element:
**Discard element**

A **Discard element** is generated if uncommitted **P0 elements** remain when trace generation becomes inoperative or if the resolution of uncommitted **P0 elements** cannot be output by the trace unit.

If trace generation remains inoperative, the outcomes of instructions that are traced by **P0 elements**, such as conditional **P0 instructions**, cannot be resolved, and therefore a **Discard element** indicates that all uncommitted **P0 elements** must be discarded.

**Mispredict element**

The **Mispredict element** indicates that the most recent non-canceled **Atom element** has the incorrect E or N status.

For example, if a branch instruction is predicted as taken, it is traced with an E **Atom element**. If the prediction becomes incorrect, then a **Mispredict element** is traced to indicate to a trace analyzer that the E **Atom element** changes to an N **Atom element**.

**D4.3.9 Other elements**

**Event element**

The **Event element** indicates when a programmed **ETEEvent** occurs and its payload contains a number to identify the **ETEEvent** number. See TRCEVENTCTRL0R, and TRCEVENTCTRL1R, for information about the programming of arbitrary **ETEEvents**.

**Overflow element**

The **Overflow element** indicates that the trace unit buffer has overflowed, and at least one trace element might have been lost.

**D4.3.10 Transactional Memory**

The **TSTART** instruction is a **P0 instruction**.
**Transaction Start element**

The *Transaction Start element* indicates that subsequent elements are within a transaction, until any of the following are traced:

- A *Transaction Failure element*.
- A *Transaction Commit element*.
- A *Cancel element* which cancels the *Transaction Start element*.

When the PE enters Transactional state, a *Transaction Start element* is generated before any instructions are traced. This indicates to the trace analyzer that subsequent elements have been executed in Transactional state.

Only a single *Transaction Start element* is generated for each outer transaction, unless the trace unit indicated the transaction had finished by generating a *Transaction Failure element*.

An example of when the trace unit generates a *Transaction Failure element* without the PE leaving Transactional state is when a trace unit buffer overflow occurs. In this example, tracing might resume after the trace unit buffer overflow, and if the PE is still in the same outer transaction then a new *Transaction Start element* would be generated.

The *Transaction Start element* appears in program order relative to other P0 elements.

When a TSTART instruction for an outer transaction is traced and tracing continues in Transactional state, the trace unit generates a *Transaction Start element* after the P0 element that is generated by the TSTART instruction, and before any subsequent P0 element.

When a TSTART instruction for an outer transaction is not traced and tracing becomes active while the PE is in Transactional state, the trace unit generates a *Transaction Start element* after the Trace On element and before any P0 elements.

**Transaction Commit element**

The *Transaction Commit element* indicates that the PE has exited Transactional state, that the transaction has completed successfully, and that all execution since the most recent *Transaction Start element* has been executed.

**Transaction Failure element**

The *Transaction Failure element* indicates that the transaction did not complete successfully and the trace analyzer discards all the execution since the most recent *Transaction Start element*, including any P0 elements which have been committed by Commit elements.

A sophisticated trace analyzer might be able to use the discarded elements to create a heuristic on why the transaction failed.
D4.4 Instruction and exception classification

D4.4.1 P0 instructions

This section defines all of the P0 instructions.

P0 instructions comprise all of the following:

• All direct P0 instructions.
• All indirect P0 instructions.

Direct P0 instructions comprise all of the following:

• All direct branch instructions.
• ISB instructions.
• TSTART instructions.
• WFE, WFET, WFI, and WFIT instructions, when indicated by TRCIDR2.WFXMODE.

Indirect P0 instructions comprise all of the following:

• All indirect branch instructions.

All uses of ISB in this section apply to all variants of the ISB instruction, including the CP15ISB instruction.

D4.4.2 AArc64 instructions

Direct P0 instructions

The following table describes the A64 direct P0 instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Unconditional Branch.</td>
</tr>
<tr>
<td>B.cond</td>
<td>Conditional Branch.</td>
</tr>
<tr>
<td>BC.cond</td>
<td>Branch Consistent conditionally.</td>
</tr>
<tr>
<td>BL</td>
<td>Branch with link.</td>
</tr>
<tr>
<td>CBZ or CBNZ</td>
<td>Compare with zero and branch.</td>
</tr>
<tr>
<td>ISB</td>
<td>Instruction Synchronization Barrier.</td>
</tr>
<tr>
<td>TBZ or TBNZ</td>
<td>Test and branch.</td>
</tr>
<tr>
<td>TSTART</td>
<td>Initiates a new transaction.</td>
</tr>
<tr>
<td>WFE, WFET</td>
<td>Wait For Event</td>
</tr>
<tr>
<td>WFI, WFIT</td>
<td>Wait For Interrupt</td>
</tr>
</tbody>
</table>
Indirect P0 instructions

The following table describes the A64 indirect P0 instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLR</td>
<td>Branch with link to register.</td>
</tr>
<tr>
<td>BLRAA</td>
<td>Authenticate and branch with link.</td>
</tr>
<tr>
<td>BLRAAZ</td>
<td>Authenticate and branch with link.</td>
</tr>
<tr>
<td>BLRAB</td>
<td>Authenticate and branch with link.</td>
</tr>
<tr>
<td>BLRABZ</td>
<td>Authenticate and branch with link.</td>
</tr>
<tr>
<td>BR</td>
<td>Branch to register.</td>
</tr>
<tr>
<td>BRAA</td>
<td>Authenticate and branch.</td>
</tr>
<tr>
<td>BRAAZ</td>
<td>Authenticate and branch.</td>
</tr>
<tr>
<td>BRAB</td>
<td>Authenticate and branch.</td>
</tr>
<tr>
<td>BRABZ</td>
<td>Authenticate and branch.</td>
</tr>
<tr>
<td>ERET</td>
<td>Return From Exception.</td>
</tr>
<tr>
<td>ERETAA</td>
<td>Authenticate and Exception return.</td>
</tr>
<tr>
<td>ERETAB</td>
<td>Authenticate and Exception return.</td>
</tr>
<tr>
<td>RET</td>
<td>Return From subroutine.</td>
</tr>
<tr>
<td>RETAA</td>
<td>Authenticate and function return.</td>
</tr>
<tr>
<td>RETAB</td>
<td>Authenticate and function return.</td>
</tr>
</tbody>
</table>

Return from exception instructions

The following table describes the A64 return from exception instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERET</td>
<td>Return From Exception.</td>
</tr>
<tr>
<td>ERETAA</td>
<td>Authenticate and Exception return.</td>
</tr>
<tr>
<td>ERETAB</td>
<td>Authenticate and Exception return.</td>
</tr>
</tbody>
</table>

D4-4792 Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. Non-Confidential
Branch with link instructions

The following table describes the A64 branch with link instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL</td>
<td>Branch with link.</td>
</tr>
<tr>
<td>BLR</td>
<td>Branch with link to register.</td>
</tr>
<tr>
<td>BLRAA</td>
<td>Authenticate and branch with link.</td>
</tr>
<tr>
<td>BLRAAZ</td>
<td>Authenticate and branch with link.</td>
</tr>
<tr>
<td>BLRAB</td>
<td>Authenticate and branch with link.</td>
</tr>
<tr>
<td>BLRABZ</td>
<td>Authenticate and branch with link.</td>
</tr>
</tbody>
</table>

Meaning of Atom elements

The following table describes the meaning of Atom elements in AArch64 A64.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>E</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>The branch was taken.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>B.cond</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>BC.cond</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>BL</td>
<td>The branch was taken.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>BLR</td>
<td>The branch was taken.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>BLRAA</td>
<td>The branch was taken.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>BLRAAZ</td>
<td>The branch was taken.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>BLRAB</td>
<td>The branch was taken.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>BLRABZ</td>
<td>The branch was taken.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>BR</td>
<td>The branch was taken.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>BRAA</td>
<td>The branch was taken.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>BRAAZ</td>
<td>The branch was taken.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>BRAB</td>
<td>The branch was taken.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>BRABZ</td>
<td>The branch was taken.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>CBZ or CBNZ</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>ERET</td>
<td>The branch was taken and the PE returned from the Exception.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>ERETAZ</td>
<td>The branch was taken and the PE returned from Exception.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>ERETAB</td>
<td>The branch was taken and the PE returned from Exception.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>ISB</td>
<td>The ISB performed a Context synchronization event and is considered as taken.</td>
<td>RESERVED.</td>
</tr>
</tbody>
</table>
Table D4-10 Meaning of Atom elements in AArch64 A64 (continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>E</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>RET</td>
<td>The branch was taken and the PE returned from the subroutine.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>RETAA</td>
<td>The branch was taken and the PE returned from the subroutine.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>RETAB</td>
<td>The branch was taken and the PE returned from the subroutine.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>TBZ or TBNZ</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>TSTART</td>
<td>Transaction started and the instruction is considered as taken.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>WFE, WFET</td>
<td>The instruction was executed and is considered as taken.</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>WFI, WFIT</td>
<td>The instruction was executed and is considered as taken.</td>
<td>RESERVED.</td>
</tr>
</tbody>
</table>

D4.4.3 AArch32 A32 instructions

Direct P0 instructions

The following table describes the A32 direct P0 instructions.

Table D4-11 A32 direct P0 instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Unconditional Branch.</td>
</tr>
<tr>
<td>B.cond</td>
<td>Conditional Branch.</td>
</tr>
<tr>
<td>BL</td>
<td>Branch with link</td>
</tr>
<tr>
<td>BLX &lt;immed&gt;</td>
<td>Branch with link and exchange.</td>
</tr>
<tr>
<td>ISB</td>
<td>Instruction Synchronization Barrier.</td>
</tr>
<tr>
<td>WFE</td>
<td>Wait For Event</td>
</tr>
<tr>
<td>WFI</td>
<td>Wait For Interrupt</td>
</tr>
</tbody>
</table>

Indirect P0 instructions

The following table describes the A32 indirect P0 instructions.

Table D4-12 A32 indirect P0 instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLX &lt;reg&gt;.</td>
<td>Branch with Link and Exchange.</td>
</tr>
<tr>
<td>BX</td>
<td>Branch and Exchange.</td>
</tr>
<tr>
<td>BXJ</td>
<td>Branch and Exchange.</td>
</tr>
<tr>
<td>Data processing instructions that modify the PC.</td>
<td>-</td>
</tr>
<tr>
<td>ERET</td>
<td>Exception Return.</td>
</tr>
</tbody>
</table>
### Branch with link instructions

The following table describes the A32 branch with link instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL</td>
<td>Branch with link</td>
</tr>
<tr>
<td>BLX &lt;immed&gt;</td>
<td>Branch with link and exchange.</td>
</tr>
<tr>
<td>BLX &lt;reg&gt;</td>
<td>Branch with Link and Exchange.</td>
</tr>
</tbody>
</table>

### Meaning of Atom elements

The following table describes the meaning of Atom elements in AArch32 A32.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>E</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>BCOND</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>BL</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>BLX &lt;immed&gt;</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>BLX &lt;reg&gt;</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>BX</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>BXJ</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>ISB</td>
<td>The ISB performed a Context synchronization event and is considered as taken</td>
<td>The ISB did not perform a Context synchronization event and is considered as not taken</td>
</tr>
<tr>
<td>LDM PC</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>LDR PC</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
</tbody>
</table>
### D4.4.4 AArch32 T32 instructions

#### Direct P0 instructions

The following table describes the T32 direct P0 instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Unconditional Branch.</td>
</tr>
<tr>
<td>B&lt;cc&gt;</td>
<td>Conditional Branch.</td>
</tr>
<tr>
<td>BL</td>
<td>Branch with Link.</td>
</tr>
<tr>
<td>BLX &lt;immed&gt;</td>
<td>Branch with Link and Exchange.</td>
</tr>
<tr>
<td>CBNZ</td>
<td>Compare and Branch on Nonzero.</td>
</tr>
<tr>
<td>CBZ</td>
<td>Compare and Branch on Zero.</td>
</tr>
<tr>
<td>ISB</td>
<td>Instruction Synchronization Barrier, including CP15 encodings.</td>
</tr>
<tr>
<td>WFE</td>
<td>Wait For Event</td>
</tr>
<tr>
<td>WFI</td>
<td>Wait For Interrupt</td>
</tr>
</tbody>
</table>

#### Indirect P0 instructions

The following table describes the T32 indirect P0 instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLX &lt;reg&gt;</td>
<td>Branch with Link and Exchange.</td>
</tr>
<tr>
<td>BX</td>
<td>Branch and Exchange.</td>
</tr>
<tr>
<td>BXJ</td>
<td>Branch and Exchange.</td>
</tr>
<tr>
<td>LDM including the PC.</td>
<td>Load Multiple including to the PC.</td>
</tr>
<tr>
<td>LDR to the PC.</td>
<td>Load to the PC.</td>
</tr>
<tr>
<td>POP {..PC}</td>
<td>Load the PC from the stack.</td>
</tr>
</tbody>
</table>
Branch with link instructions

The following table describes the T32 branch with link instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL</td>
<td>Branch with Link.</td>
</tr>
<tr>
<td>BLX &lt;immed&gt;</td>
<td>Branch with Link and Exchange.</td>
</tr>
<tr>
<td>BLX &lt;reg&gt;</td>
<td>Branch with Link and Exchange.</td>
</tr>
</tbody>
</table>

Meaning of Atom elements

The following table describes the meaning of Atom elements in AArch32 T32.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>E</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>B&lt;cc&gt;</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>BL</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>BLX &lt;immed&gt;</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>BLX &lt;reg&gt;</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>BX</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>BXJ</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>CBNZ</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>CBZ</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>Data processing</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>instructions that</td>
<td></td>
<td></td>
</tr>
<tr>
<td>modify the PC.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISB</td>
<td>The ISB performed a Context</td>
<td>The ISB did not perform a</td>
</tr>
<tr>
<td></td>
<td>synchronization event and is</td>
<td>Context synchronization</td>
</tr>
<tr>
<td></td>
<td>considered as taken.</td>
<td>event and is considered as</td>
</tr>
<tr>
<td></td>
<td></td>
<td>not taken.</td>
</tr>
<tr>
<td>LDM including the</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>PC.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDR to the PC.</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>POP {...,PC}</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
</tbody>
</table>
### Table D4-18 Meaning of Atom elements in AArch32 T32 (continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>E</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFE</td>
<td>The branch was taken and the PE returned from the Exception.</td>
<td>The branch was not taken and the PE did not return from the Exception.</td>
</tr>
<tr>
<td>TBB</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>TBH</td>
<td>The branch was taken.</td>
<td>The branch was not taken.</td>
</tr>
<tr>
<td>WFE</td>
<td>The instruction either passed its condition code check or failed its condition code check, but it is considered as taken.</td>
<td>The instruction failed its condition code check and is considered as not taken.</td>
</tr>
<tr>
<td>WFI</td>
<td>The instruction either passed its condition code check or failed its condition code check, but it is considered as taken.</td>
<td>The instruction failed its condition code check and is considered as not taken.</td>
</tr>
</tbody>
</table>

### D4.4.5 Exceptions to Exception element encoding

The following table shows the Exception mapping for exceptions taken to AArch64 state.

<table>
<thead>
<tr>
<th>Reason</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Target exception</td>
<td>Inst fault</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>Inst debug</td>
</tr>
<tr>
<td>FIQ</td>
<td>FIQ</td>
</tr>
<tr>
<td>HVC</td>
<td>Call</td>
</tr>
<tr>
<td>Halting debug event</td>
<td>Debug halt</td>
</tr>
<tr>
<td>IRQ</td>
<td>IRQ</td>
</tr>
<tr>
<td>Illegal execution state</td>
<td>Trap</td>
</tr>
<tr>
<td>Instruction Abort</td>
<td>Inst fault</td>
</tr>
<tr>
<td>Instruction or event trapped by a control bit</td>
<td>Trap</td>
</tr>
<tr>
<td>MemCopy or MemSet</td>
<td>Trap</td>
</tr>
<tr>
<td>Misaligned PC</td>
<td>Alignment</td>
</tr>
<tr>
<td>PAC Fail</td>
<td>Data fault</td>
</tr>
<tr>
<td>SError interrupt</td>
<td>System Error</td>
</tr>
<tr>
<td>SMC</td>
<td>Call</td>
</tr>
<tr>
<td>SVC</td>
<td>Call</td>
</tr>
<tr>
<td>Software Breakpoint Instruction</td>
<td>Inst debug</td>
</tr>
<tr>
<td>Software Step</td>
<td>Inst debug</td>
</tr>
<tr>
<td>Stack Pointer Misalignment</td>
<td>Alignment</td>
</tr>
</tbody>
</table>
The following table shows the Exception mapping for exceptions taken to AArch32 state.

### Table D4-20 Exception mapping for exceptions taken to AArch32 state

<table>
<thead>
<tr>
<th>Reason</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoint</td>
<td>Inst fault</td>
</tr>
<tr>
<td>FIQ</td>
<td>FIQ</td>
</tr>
<tr>
<td>HVC</td>
<td>Call</td>
</tr>
<tr>
<td>Halting debug event</td>
<td>Debug halt</td>
</tr>
<tr>
<td>IRQ</td>
<td>IRQ</td>
</tr>
<tr>
<td>Illegal execution state</td>
<td>Trap</td>
</tr>
<tr>
<td>Instruction or event trapped by a control bit</td>
<td>Trap</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>Inst fault</td>
</tr>
<tr>
<td>SError interrupt</td>
<td>System Error</td>
</tr>
<tr>
<td>SMC</td>
<td>Call</td>
</tr>
<tr>
<td>SVC</td>
<td>Call</td>
</tr>
<tr>
<td>Software Breakpoint Instruction</td>
<td>Inst fault</td>
</tr>
<tr>
<td>Synchronous Data Abort</td>
<td>Data fault</td>
</tr>
<tr>
<td>UNDEFINED instruction</td>
<td>Trap</td>
</tr>
<tr>
<td>Vector Catch exception</td>
<td>Inst fault</td>
</tr>
<tr>
<td>Watchpoint</td>
<td>Data fault</td>
</tr>
</tbody>
</table>
D4.5 About the ETE trace unit

Figure D4-10 shows the stages of trace generation:

![Stages of trace generation](image)

D4.5.1 Resetting the trace unit

A trace unit includes a trace unit reset, which resets all of the:

- Trace unit trace registers.
- Trace unit management registers.

When the trace unit Core power domain is powered up, a trace unit reset is applied.

It is IMPLEMENTATION DEFINED whether the system has a mechanism to initiate a trace unit reset on demand.

In a PE with FEAT_TRF, a PE Cold reset causes EDSCR.TFO to be reset to 0 which means that tracing is prohibited after the Cold reset until explicitly permitted by software. If tracing from a Cold reset is required, the debugger needs to ensure any relevant controls, including EDSCR.TFO, are programmed to permit tracing. Programming such registers might involve causing the PE to enter Debug state to ensure the registers can be programmed before the PE starts executing instructions.

When a trace unit reset is applied, the trace unit resets the values of all trace unit registers to the values described in the individual register descriptions.

Some other Arm trace architectures support multiple types of reset for the trace unit.

D4.5.2 System behaviors

The trace unit outputs all of the trace byte stream, without external stimulus, within finite time.

Behavior on enabling

While both of the following are true, the trace unit is enabled:

- TRCPRGCTRLR.EN is set to 1.
- The OS Lock is unlocked.

Some Arm trace architectures have a dedicated trace unit OS Lock, whereas ETE depends on the PE OS Lock.

While the trace unit is enabled, the trace unit can trace all PE execution, except when any of the following are true:

- A trace unit buffer overflow occurs.
- The authentication interface prohibits the tracing of certain pieces of code.
- FEAT_TRF or FEAT_TRBE prohibit the tracing of certain pieces of code.

No sequences of code or PE operations are exempt from this requirement. However, while the trace unit is transitioning from an enabled to a disabled state, or from a disabled to an enabled state, some loss of trace is permitted.

While the trace unit is enabled, writes to most trace unit trace registers might be ignored. It is UNKNOWN whether writes to these registers succeed. When the writes are successful, the behavior of the trace unit is UNPREDICTABLE.

Trace analyzers must not write to most trace unit trace registers while the trace unit is enabled, or if TRCSTATR.IDLE indicates that the trace unit is not idle.
While the trace unit is enabled or idle, all resources that are visible in the programmers’ model might have unstable values. Therefore, a trace analysis tool must be aware that the following values might be dynamically changing as they are being read:

- The Counter values, indicated by the TRCCNTVR<n>.
- The Sequencer state, indicated by TRCSEQSTR.
- The ViewInst start/stop function, indicated by TRCVICTLR.
- The Single-shot Comparator Control status, indicated by the TRCSSCSR<n>.

When the trace unit becomes enabled, the trace unit does not reset the state of any of the resources in the trace unit, including the Counters, the Sequencer, and the ViewInst start/stop function.

While the trace unit is disabled, and before it is enabled, a trace analyzer ensures the trace unit resources are programmed with a valid initial state.

### Behavior on disabling

While the trace unit is disabled:

- The trace unit cannot generate trace.
- The trace unit resources are disabled.

For more information, see [Behavior of the resources while in the Pausing state](#) on page D4-4844.

While either of the following is true, the trace unit is disabled:

- TRCPRGCTRLR.EN is 0.
- The OS Lock is locked.

Some Arm trace architectures have a dedicated trace unit OS Lock. ETE depends on the PE OS Lock.

When the trace unit becomes disabled, the trace unit stops generating trace and empties the trace buffers by outputting any data in them.

When the trace buffers are empty, after the trace unit has become idle after becoming disabled, TRCSTATR.IDLE indicates that the trace unit is idle.

When the trace unit becomes disabled, all resources that are visible in the programmers’ model retain their values and become stable at those values.

When the trace unit becomes disabled, when the resources are stable, TRCSTATR.PMSTABLE indicates that the programmers’ model is stable.

When the trace unit becomes disabled after the trace unit has generated Event elements, the trace unit outputs the Event elements before TRCSTATR.IDLE indicates that the trace unit is idle.

While the trace unit is disabled, the following are true:

- No trace is generated.
- All trace unit resources and ETEEvents are disabled.

### Behavior on flushing

The trace unit is allowed to buffer the trace byte stream to make efficient use of system infrastructure.

As the trace unit is allowed to delay the output of the trace byte stream to the system infrastructure, there are system events that require all of the trace byte stream to be observable to other observers in the system.

Making the trace byte stream visible to other observers is known as a trace unit flush.

When any of the following occur, a trace unit flush is requested:

- The trace unit transitions from an enabled to a disabled state.
- The trace capture infrastructure requests a trace unit flush.
• A TSB `CSYNC` instruction is executed in a Trace Prohibited region while the Trace Buffer Extension is implemented and enabled.

A trace unit flush might be requested for IMPLEMENTATION DEFINED reasons. For example:

• Before the trace unit enters either:
  — The low-power state.
  — A powerdown state.
• The PE enters Debug state.

An example of a trace unit flush is one requested on an Arm AMBA ATB interface. For more information on the Arm AMBA ATB interface, see the *AMBA® ATB Protocol Specification (ARM IHI 0032)*.

When a trace unit flush is requested, the trace unit performs the following tasks before responding to the flush request:

1. Encode any remaining elements into trace packets.
2. Complete any packets that are in the process of being generated.
3. Output all trace packets for all PE execution that occurred before the flush request was received.

An example of when the trace unit might need to encode remaining elements into trace packets before a trace unit flush is when there are Commit elements that are not yet encoded.

When a trace unit flush occurs while the trace unit is recovering from a trace unit buffer overflow, the trace unit outputs the corresponding Overflow element before responding to the flush request.

When a trace unit flush occurs, the trace unit either continues to generate trace or stops generating trace, depending on what condition caused the trace flush. For example, if a flush occurs because the trace unit is entering a disabled state, then tracing becomes inactive after the trace flush.

When a condition causes both a trace unit flush and the trace unit to stop generating trace, the trace unit stops generating trace before responding to the flush request, and before indicating that the trace unit is idle.

On entry to Debug state, Arm recommends that the Exception element indicating entry to Debug state is included in the flushed trace data if tracing is active.

When a trace unit flush is requested, the trace unit outputs the data within a finite period.

When a trace unit flush is requested and the cause of the flush request requires an acknowledgment, the trace unit generates the acknowledgment within a finite period.

The flush request mechanism on AMBA ATB is an example of a cause of a flush request that also requires an acknowledgment.

**Low-power state**

The low-power state in the trace unit is a mechanism to improve energy efficiency during periods where trace generation is limited.

Scenarios where the trace unit might be in the low-power state are any of the following:

• The PE is in a low-power state.
• The PE is in Debug state.

The trace unit is only permitted to be in the low-power state when any of the following are true:

• The PE is in a low-power state due to the Wait for Event mechanism.
• The PE is in a low-power state due to the Wait for Interrupt mechanism.
• The PE is in Debug state.
• The trace unit is Disabled.
**Trace unit behavior when the PE is in a low-power state**

The PE that is being traced might support a low-power state where no execution occurs. This low-power state might be invoked, for example, when the PE executes a `WFI`, `WFIT`, `WFE`, or `WFET` instruction.

When the PE is in a low-power state, it might be advantageous if the trace unit also enters a low-power state.

It is IMPLEMENTATION DEFINED whether a trace unit supports the low-power state.

While the trace unit is in the disabled state, the trace unit does not stop the PE from entering a low-power state.

While the trace unit is in Low-power Override Mode, the trace unit does not affect the operation of the PE.

**Trace unit behavior in the low-power state**

While the trace unit is enabled, when the trace unit enters the low-power state, the trace unit continues to appear enabled throughout the time it is in the low-power state.

When the trace unit enters or leaves the low-power state, the trace unit does not lose resource events that are in transition through the trace unit, except those permitted when moving through the Pausing state of the resources. For more information on the resource events that are permitted to be lost when in the Pausing state, see *Behavior of the resources while in the Pausing state* on page D4-4844.

Observation of resource events that are in transition through the trace unit when it enters the low-power state might not occur until after the trace unit leaves the low-power state.

While the trace unit is not in the low-power state, and before it enters the low-power state, the resources enter the Paused state. For more information on the pausing of resources, see *Behavior of the resources while in the Paused state* on page D4-4844.

If `WFI` and `WFE` instructions are classified as P0 instructions, see TRCIDR2.WFXMODE, and the trace unit enters the low-power state as a result of a `WFI` or `WFE` instruction, Arm strongly recommends that the following elements are generated before the trace unit enters the low-power state:

- The *Atom element* that represents a `WFI`, `WFIT`, `WFE`, or `WFET` instruction.
- Any pending *Commit elements*.

While the trace unit is in the low-power state, the trace unit does not generate trace, including ETEEEvent trace.

While the trace unit is in the low-power state, the resources remain in the state that they were in before the trace unit entered the low-power state.

The resources are:

- The Counters.
- The Sequencer.
- The ViewInst start/stop function.
- The Single-shot Comparator Controls.

While the trace unit is in the low-power state, the trace unit drives all External Outputs low.

While the trace-unit is in the low-power state, the PE and external debuggers ability to access the trace unit trace registers and trace unit management registers is unaffected.

While the trace unit is in the low-power state, when a trace protocol synchronization request occurs, the trace unit handles the trace protocol synchronization request correctly. For more information on how the trace unit handles trace protocol synchronization requests, see *Trace protocol synchronization* on page D4-4776.

While the trace unit is in a retention state, external debugger accesses to the trace unit behave as if there is no power to the trace unit Core power domain.

While the trace unit is in the low-power state, the trace unit might not recognize external events, such as the assertion of any External Inputs.
While the trace unit is in the low-power state, it is IMPLEMENTATION DEFINED whether the cycle counter continues to count or not.

While the trace unit is in the low-power state, timestamp requests might be ignored.

It is possible that the trace unit might intermittently leave and reenter the low-power state while the PE is in a low-power state. If this happens, the trace unit resources might become intermittently active during this time. In addition, trace generation might also become intermittently active, and this means that the trace unit might output some packets. This behavior is IMPLEMENTATION DEFINED.

There is no additional requirement for the trace unit to generate a Trace Info element or Trace On element when leaving the low-power state. However, if the trace unit entered the low-power state because the PE was in Debug state, the normal requirements for restarting trace after leaving Debug state apply, including generation of a Trace On element. For more information on the PE Debug state, see Trace unit behavior while the PE is in Debug state.

The trace unit can be programmed so that it does not enter the low-power state, by enabling Low-power Override Mode. Low-power Override Mode is selected using TRCEVENTCTL1R.LPOVERRIDE.

When Low-power Override Mode is enabled, the resources continue operating and the trace unit can generate trace.

Low-power Override Mode does not affect the operation of the PE, however it is not required to prevent the PE from entering a low-power state. This means that even though the trace unit can generate trace, it might only generate Event elements.

D4.5.3 Trace unit behavior while the PE is in Debug state

While ViewInst is active, when the PE enters Debug state, the trace unit generates an Exception element which indicates that the PE has entered Debug state.

When the PE enters Debug state, ViewInst becomes inactive, and remains inactive throughout the time the PE is in Debug state.

While the PE is in Debug state, the trace unit does not trace instructions that are executed.

When the PE exits Debug state and ViewInst becomes active, the trace unit generates a Trace On element.

While the PE is in Debug state, the ViewInst start/stop function maintains its state.

If an Exceptional occurrence occurs between the PE exiting Debug state and the PE executing the first instruction, the value of TRCRSR.TA is used to determine if the Exceptional occurrence is traced. In general, if the entry to Debug state was traced then TRCRSR.TA will be set to 1, and therefore this Exceptional occurrence on exit from Debug state is traced.

If a PE Reset occurs when the PE is in Debug state this is treated as leaving Debug state. This means that a Trace On element and an Exception element indicating a PE Reset are traced if tracing is not prohibited and either of the following are true:

- TRCRSR.TA is 1.
- Forced tracing of PE Resets is active.

D4.5.4 Trace unit behavior on a trace unit buffer overflow

When a trace unit buffer overflow occurs, trace generation becomes inoperative until the trace unit can recover from the overflow.

When a trace unit buffer overflow occurs, the trace unit does not output a partial trace packet, that is, the trace unit can only output complete packets.

The Overflow element indicates to a trace analyzer that a trace unit buffer overflow has occurred. For more details on the generation of an Overflow element, see Overflow element on page D4-4830.

When the trace unit recovers from a trace unit buffer overflow, the following occur:

- Trace protocol synchronization is requested.
• Trace protocol synchronization occurs before the trace unit outputs any packets.

When an Overflow packet is generated, the trace unit might output any of the following packets before it outputs an Alignment Synchronization packet:
• Event packet.
• Overflow packet.
• Discard packet.
• Ignore packet.

Arm recommends that the Alignment Synchronization packet is the first packet output after the Overflow packet.

D4.5.5 Trace unit power states

The Arm architecture defines the following power states:

Normal
The trace unit Core power domain is fully powered up and the trace unit registers are accessible.

Standby
The trace unit Core power domain is on, but there are measures to reduce energy consumption. Standby is transparent, meaning that to software and to an external debugger it is indistinguishable from normal operation.

Retention
The OS takes some measures, including IMPLEMENTATION DEFINED code sequences and registers, to reduce energy consumption. Trace unit registers cannot be accessed. A trace unit reset does not occur on leaving retention.

Powerdown
The OS takes some measures to reduce energy consumption by turning the trace unit Core power domain off. Trace registers cannot be accessed. A trace unit reset occurs on leaving powerdown.

A trace unit might support a low-power state, which is equivalent to the standby state.

A trace unit might support a retention state or a powerdown state, and both of these states are considered to be a state where the trace unit Core power domain is powered down.

If the trace unit is implemented in a power domain which is separate from the PE power domain, all of the following are true:
• The trace unit Core power domain might be able to be powered down without powering down the PE power domain.
• The trace unit Core power domain is always powered down when the PE power domain is powered down.

A read of TRCPDSR returns information about the current state of the trace unit.

Table D4-21 on page D4-4805 describes the meanings of the returned value.

<table>
<thead>
<tr>
<th>STICKYPD</th>
<th>POWER</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>The trace unit Core power domain is powered and the trace unit registers are accessible.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>The trace unit Core power domain is powered and the trace unit registers are accessible. A trace unit reset or power interruption has occurred so the trace unit register state might not be valid.</td>
</tr>
</tbody>
</table>

When the trace unit Core power domain transitions from powered down to powered up, if the trace unit register state has been preserved over the powerdown, then TRCPDSR.STICKYPD is restored to the value before powerdown.

When the trace unit Core power domain transitions from powered down to powered up, if the trace unit register state has not been preserved over the power down then TRCPDSR.STICKYPD is set to 0b1.
Note: ETE only supports a single power domain and therefore TRCPDSR.POWER is always 0b1.

A system might support a Debug power domain that contains the interface between the trace unit and the external debugger. The Debug power domain usually needs to be powered up when the external debugger needs to connect to the system. Such a Debug power domain is described in *Resets and power domains on page D1-4673.*

If the trace unit Core power domain can be powered down independently of the Debug power domain, Arm recommends that the system implements an external debug component with a powerup request mechanism which can request the trace unit Core power domain to be powered up.

Arm strongly recommends the powerup request mechanism is a CoreSight Class 0x9 ROM Table containing a parent entry for the trace unit. A parent entry of a component is one of:

- An entry in the ROM table that locates the component.
- An entry in a first ROM table that locates a second ROM table that includes a parent entry for the component.

The second ROM table is a descendant of the first ROM table.

This definition of a parent entry is recursive, and therefore the powerup request mechanism might be high up in a hierarchy of ROM tables. The ROM table and any descendants might describe other debug components, including debug components for other PEs. The ROM table might have parent entries in other ROM tables, and those parent entries might also have a powerup request mechanism in those ROM tables.

If the powerup request mechanism is implemented, in the Class 0x9 ROM Table containing the powerup request mechanism for the trace unit:

- The POWERIDVALID bit in the parent entry must be 0b1.
- The POWERID field in the parent entry has an IMPLEMENTATION DEFINED value.

It is IMPLEMENTATION DEFINED whether the trace unit Core power domain is the PE Core power domain or some other power domain. For more information on the CoreSight Class 0x9 ROM Table, see the Arm® CoreSight® Architecture Specification (ARM IHI 0029).

### D4.5.6 Visibility of the PE operation

This section describes the ability of the trace unit to trace the execution of the operation of the PE.

When the trace unit performs indirect reads of PE System registers, the trace unit complies with the rules associated with Context synchronization events.

When the trace unit performs indirect reads of PE System registers, the trace unit complies with the rules associated with the TSB CSYNC instruction as defined in *Chapter D6 The Trace Buffer Extension.*

When instructions are executed outside of any Trace Prohibited region, the trace unit observes the execution.

When observable instructions are executed, the trace unit observes all execution before a PE Context synchronization event occurs, as defined in *Chapter D6 The Trace Buffer Extension.*

When an Exceptional occurrence occurs outside of any Trace Prohibited region, the trace unit observes the Exceptional occurrence.

Executing a TSB CSYNC instruction generates a Trace synchronization event as defined in *Chapter D6 The Trace Buffer Extension.*

When a TSB CSYNC instruction is executed in a Trace Prohibited region, the TSB CSYNC instruction does not become microarchitecturally-finished until the resources are in the Paused state or the trace unit is in the Idle or Stable state.

While the PE is outside a transaction, after a TSB CSYNC instruction executed inside a Trace Prohibited region generates a Trace synchronization event, the Trace synchronization event is microarchitecturally-finished when the trace operation has microarchitecturally-finished for every instruction before the Context synchronization event before the TSB CSYNC instruction that generated the Trace synchronization event.

For more details on the TSB CSYNC instruction, see RMRVPT.
While the PE is inside a transaction, when a Trace synchronization event occurs, the Trace synchronization event becomes microarchitecturally-finished within a finite period.

While the PE is inside a transaction, the completion of a Trace synchronization event is not dependent on the resolution of the transaction. It might still depend on other aspects of the trace operation.

When a T58 CSYNC instruction executed in a Trace Prohibited region becomes microarchitecturally-finished, the trace unit generates no more trace until the PE leaves the Trace Prohibited region.

When a T58 CSYNC is executed in a Trace Prohibited region, the rules around generation of a trace flush and requiring no more trace to be generated in the Trace Prohibited region mean that only whole trace packets are output, and the last byte output is the end of a packet.

These rules ensure that no new trace is generated and allows various System registers to be changed, such as those controlling the Trace Buffer Extension, without the risk of any trace being generated while those registers are being changed.

When the trace unit becomes enabled in a Trace Prohibited region, the trace unit generates no trace until the PE leaves the Trace Prohibited region.

Note: This ensures that no trace is generated until the PE leaves the Trace Prohibited region, and therefore allows various System registers to be changed, such as those controlling the Trace Buffer Extension, without the risk of any trace being generated while those registers are being changed.

The above rule ensures that no trace is generated until the PE leaves the prohibited region, and therefore allows various system registers to be changed, such as those controlling the Trace Buffer Extension, without the risk of any trace being generated while those registers are being changed.

The operation as defined in Chapter D6 The Trace Buffer Extension can be split into operations that are performed by one of the following:

- The PE.
- The ETE trace unit.
- The trace buffer.

The operation of the trace unit is defined by the ETE trace operation.

If the Trace Buffer Unit is implemented and enabled, when a Trace synchronization event occurs, and after all of the trace byte stream generated by the trace unit is flushed to the trace buffer, the Trace synchronization event completes.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>po</td>
<td>program-order</td>
<td>head is in program order after tail.</td>
</tr>
<tr>
<td>rf</td>
<td>Reads-from</td>
<td>tail Reads-from head.</td>
</tr>
<tr>
<td>co</td>
<td>Coherence-after</td>
<td>head is Coherence-after tail.</td>
</tr>
<tr>
<td>fr</td>
<td>from-read</td>
<td>As co, except that the operation at head is a read.</td>
</tr>
<tr>
<td>ob</td>
<td>Observed-by</td>
<td>tail is Observed-by head. Only applies for different Observers.</td>
</tr>
<tr>
<td>tb</td>
<td>traced-by</td>
<td>head is the trace operation for the instruction at tail.</td>
</tr>
</tbody>
</table>

Table D4-22 Labels for ordering diagrams
ETE trace operation

Each instruction has all of the following state information:

- PC.
- PSTATE.T.
- PSTATE.EL.
- The Security state.
- CONTEXTIDR_EL1.PROCID.
- CONTEXTIDR_EL2.PROCID.
- TRFCR_EL1.
- TRFCR_EL2.
- MDCR_EL3.STE.
- TxNestingLevel.

The trace information generated contains Address information in Target Address elements, Source Address elements, Exception elements, and Q elements. The Address information contains:

- The virtual address of an instruction.
- The instruction set, known as the sub_isa.

The trace information generated contains Context information in Context elements. The Context information contains:

- The current Security state.
- The current Exception level.
- The current Execution state, which is AArch32 or AArch64.
- The current Context identifier, as stored in CONTEXTIDR_EL1.PROCID.
- The current Virtual context identifier, as stored in CONTEXTIDR_EL2.PROCID.

When an instruction is executed and all the trace elements for the instruction have been generated, the trace operation for the instruction is complete.

Trace elements generated for an instruction might include:

- Global timestamp elements.
- Cycle count elements.
- Speculation resolution elements.
- Transaction resolution elements.

For example, the tracing of PE execution is where:

- Resolved instruction A is executed in program order before a Resolved instruction B.
- tA is all the trace elements that are generated due to the tracing of instruction A.
• \( t_B \) is all the trace elements that are generated due to the tracing of instruction \( B \).
• The trace elements for \( t_A \) must be observed before \( t_B \).

This is shown in Figure D4-11.

![Figure D4-11 Trace operation](image)

### Impact on PE behavior

The ETE architecture places no requirements on the impact that trace generation has on the functional performance of a PE. Arm expects that trace unit implementations are designed according to the market requirements of the PEs being traced, and according to the trace requirements for those PEs. For some markets and trace requirements, the trace solution might always have some performance impact on the PE and the ETE architecture does not prohibit this.

### Behavior on a PE Warm reset

A PE Warm reset does not cause a Trace unit reset. Note: This ensures that tracing is possible through a PE Warm reset.

A PE Warm reset might occur at the same time as a Trace unit reset, however, these are asynchronous and unrelated events.

### Instruction block

How instructions are executed can vary significantly between PE designs. To allow for these variations the ETE architecture allows some flexibility within the filtering model. Rather than applying the filtering model to individual instructions it is applied to blocks of instructions.

An instruction block contains one or more instructions.

An instruction block can contain zero or one \( P0 \) instruction.

When an instruction block is generated which contains a \( P0 \) instruction, the instruction block has the \( P0 \) instruction as the last instruction in the block.

Exceptional occurrences do not occur between instructions in an instruction block.

The addresses of the instructions within an instruction block are sequential.

The number of instructions in a block can vary from block to block and can vary each time the same sequence of instructions are executed.

For example, the tracing of an instruction block is where:
• Resolved instruction \( A \) is executed in program order before a Resolved instruction \( B \).
• \( t_A \) is all the trace elements that are generated due to the tracing of instruction \( A \).
• \( t_B \) is all the trace elements that are generated due to the tracing of instruction \( B \).

This is shown in Figure D4-12 on page D4-4810.
Exposing speculation

For some PE microarchitectures, the tracing of execution-order only might not be achievable. The ETE architecture provides the ability to trace speculatively executed instructions.

When speculative instructions are observed, the trace unit indicates whether each instruction is resolved or canceled with a resolve operation or a cancel operation.

A resolve operation indicates that one or more instructions have, or will be, architecturally executed.

A cancel operation indicates that one or more instructions, although observed by the trace unit, did not architecturally execute.

There is no requirement to expose any speculation to the trace unit.

For example, the tracing of speculation execution is where:

- $S$ is executed in speculative execution-order after a Resolved instruction $A$.
- $A$ is executed in program order before a Resolved instruction $B$.
- $S$ is not in speculative execution-order after $B$.
- $Q$ is executed in speculative execution-order after a Resolved instruction $B$.

This is shown in Figure D4-13.

Trace Prohibited Regions

Trace Prohibited regions are instruction address regions or periods of execution by the PE that are not to be traced. Instructions and Exceptional occurrences which are not prohibited are not necessarily traced because the trace unit has a number of trace filtering functions to limit the amount of trace generated to the sections or periods of interest.

An executable program might contain regions of code that are prohibited to trace. These regions might be associated with a higher Security state, or with the PE entering a privileged mode so that it can execute the instructions that are contained within them.
Tracing might be prohibited while the PE is operating in certain states or modes. For example:

- Non-invasive debug might be prohibited while the PE is in Secure state.
- FEAT_TRF might prohibit tracing.
- FEAT_TRBE might prohibit tracing.

Trace might also become prohibited if, while tracing program execution, an authentication interface changes the currently permitted level of non-invasive debug. For example, if trace is permitted and active while the PE is operating in Secure state, and then the permitted level of non-invasive debug changes from being permitted for Secure state, to not permitted, then trace becomes prohibited.

Prohibited regions in self-hosted trace on page D3-4759 and Chapter D6 The Trace Buffer Extension describe when FEAT_TRBE prohibits tracing.

If an optional authentication interface is implemented, while Secure non-invasive debug is disabled according to the optional authentication interface and while SelfHostedTraceEnabled() returns FALSE, then for execution in Secure state, the PE executes in a Trace Prohibited region.

An example of an optional authentication interface is the CoreSight Authentication interface. For more information, see the Arm® CoreSight® Architecture Specification (ARM IHI 0029).

While the PE is executing code from a Trace Prohibited region, the trace unit does not trace instructions or Exceptional occurrences, including PE Resets.

While the PE is executing code from a Trace Prohibited region, instruction Address Comparators do not match on any instructions in the Trace Prohibited region.

While cycle counting is enabled and the PE is executing code from a Trace Prohibited region, the cycle counter continues to count.

When the PE leaves a Trace Prohibited region and tracing restarts, the cycle counter includes cycles spent in the Trace Prohibited region in the cycle count.

The behavior of the resources when entering a Trace Prohibited region is defined in Behavior of the resources while in the Paused state on page D4-4844.

While the PE is executing code from a Trace Prohibited region, the trace unit does not output any trace that might provide information about the execution in the Trace Prohibited region.

Examples of information about execution in a Trace Prohibited region that trace might provide are the context of execution, instruction addresses, and the address of the first instruction in the Trace Prohibited region.

The most common cause of an entry into a Trace Prohibited region is an Exceptional occurrence or Context synchronization event.

When an Exceptional occurrence that must be traced causes the PE to enter a Trace Prohibited region, the trace unit generates an Exception element that indicates the exception type.

When the PE enters a Trace Prohibited region and there are unresolved speculative P0 elements remaining in the trace byte stream, when the resolution of the speculative elements is known the trace unit generates the appropriate Commit elements or Cancel elements.

When the PE leaves a Trace Prohibited region and ViewInst is active, that is, any filtering applied dictates that ViewInst is active, the trace unit generates a Trace On element.

The purpose of the trace unit generating a Trace On element when the PE exits a Trace Prohibited region and ViewInst is active is to indicate to the trace analyzer that there has been a discontinuity in the trace element stream.

If the PE leaves a Trace Prohibited region other than when a Context synchronization event occurs, the Trace Prohibited region is permitted to extend up to the next Context synchronization event. Typically, a PE leaves a Trace Prohibited region via a Context synchronization event, but a PE might leave a Trace Prohibited region when the authentication interface changes, or when moving from Secure to Non-secure state without an exception return.

If an Exceptional occurrence occurs between the PE exiting a Trace Prohibited region and the PE executing the first instruction, the value of TRCRSR.TA is used to determine if the Exceptional occurrence is traced.
Multi-threaded processor

Processors with multiple threads or PEs have a trace unit for each thread or PE.

The processor might support enabling and disabling of threads, either at PE Reset time or dynamically. The trace units for the threads that are disabled might behave in one of the following ways:

- The trace unit Core power domain is powered down.
- The trace unit Core power domain is held in the trace unit reset state.

Arm recommends that the trace units for threads that are permanently disabled are not visible: either they are not included, or they are marked as not present in any ROM tables that describe the system.

Sharing between multiple PEs

Previous Trace architectures have allowed the trace unit to be shared between multiple PEs.

A trace unit only traces a single PE, that is, it cannot be shared between multiple PEs.

D4.5.7 Speculation resolution

The trace unit implements a maximum speculation depth, that is, the maximum permitted number of P0 elements that can be speculative at any instance. TRCIDR8.MAXSPEC indicates the IMPLEMENTATION DEFINED maximum speculation depth.

The trace unit never outputs more speculative P0 elements than the maximum speculation depth.

If a trace unit is not exposed to any speculative execution, then Arm recommends that the trace unit implements a maximum speculation depth of zero, and in this case:

- Cancel elements are not generated.
- Commit elements are generated after each P0 element, causing each P0 element to be immediately resolved when it is generated. The instruction trace protocol implicitly generates these Commit elements for each P0 element, meaning that explicit Commit packets are not required.
- Mispredict elements are not generated.

The ETE architecture defines Commit element and Cancel elements to allow the speculation of the P0 elements to be resolved by the trace analyzer. The trace unit is required to calculate the number of P0 elements which are committed or canceled. There are many methods by which these numbers can be calculated, but in the generic case the trace unit can use the following mathematical procedure.

The PE can speculatively indicate blocks of instructions to the trace unit. Each block of instructions is given a tag where tag belongs to 0…m and m is the number of rewind points supported by the PE.

The number of instructions per block can be random from the set N and there is a maximum of one P0 instruction per block. The order in which the tags are used can be random, but a tag cannot be reused until the previous block with that tag has been resolved, canceled or merged.

This procedure generates a transform from the potentially random sequence of core tags to a more useable space. The transform T evolves over time as the tags are reused and provides the mapping onto the new space,

\[ T_t = [c_0, \ldots, c_m] \]  

(1)

c_i is the mapping for core tag i.

c_i belongs to 0…q, where q is greater than m.
Initialization

To perform the necessary calculations, the trace unit tracks the transform of the last resolved block. \( \gamma_t \) is the last committed indicator. The algorithm is initialized at \( t=0 \) to

\[
\forall i \in 0, \ldots, m : T_0[i] = \gamma_0
\]

\[
x_0 = \gamma_0
\]

New block operation

The sequence of the numbers in the transformed space, \( x_t \), is defined by the following equation:

\[
x_{t+1} = \begin{cases} 
\left| \left(x_t + 1 \right) \mod q \right| & \text{If a traced P0 instruction} \\
x_t & \text{Otherwise}
\end{cases}
\]

\[
T_{t+1}[\text{tag}_t] = \begin{cases} 
\left| \left(x_t + 1 \right) \mod q \right| & \text{If a traced P0 instruction} \\
T_t[\text{tag}_t] & \text{Otherwise}
\end{cases}
\]

Resolved operation

The PE can resolve one or more blocks in an atomic operation. This is performed by indicating the youngest block’s tag to be resolved, and by inference all older blocks.

The number required by the Commit element is calculated by

\[
n_+ = \left| (T_t[l] - \gamma_t) \mod q \right|
\]

The state of the transform is updated by

\[
\gamma_{t+1} = T_t[l]
\]

where \( l \) is the youngest block’s tag.

Cancel operation

The PE can cancel one or more blocks in an atomic operation. This is performed by indicating the oldest block to be canceled.

The number required by the Cancel element is calculated by

\[
n_- = \left| (x_t - T_t[r]) \mod q \right|
\]

The state of the transform is updated by

\[
x_{t+1} = T_t[r]
\]

where \( r \) is the oldest block’s tag.
### D4.5.8 Filtering trace generation

The amount of trace that can be generated by the trace unit can be significant. Not all the operations of the PE are always relevant. The amount of trace generated can be reduced by the use of the trace unit filter functions.

#### ViewInst function

The filtering function of the instruction trace is expressed as a calculation evaluated for each instruction.

$$\text{ViewInst}_i = \begin{cases} 
0 & \text{When Prohibited} \\
0 & \text{When in Debug state} \\
S_i \land I_i \land E_i \land N_i & \text{Otherwise} 
\end{cases}$$

(1)

- \(S_i\) = ViewInst start/stop function
- \(I_i\) = ViewInst include/exclude function
- \(E_i\) = Exception level filtering
- \(N_i\) = Resource event based filtering

While ViewInst is high, the trace unit traces all instructions.

Instructions for which ViewInst is low might be traced. This might be as a result of tracing the next P0 element or optimizations in the trace unit.

When ViewInst is high for an instruction in an instruction block, the trace unit traces the entire instruction block.

When ViewInst becomes high, the trace unit traces the next P0 instruction or Exceptional occurrence.

Some instruction types cause the trace unit to generate P0 elements, so that they are explicitly traced. Other instruction types however are not explicitly traced. The execution of these other instruction types can be inferred from the P0 elements. This means that the following scenario is possible:

- While ViewInst is high, some instructions are executed. This means that ViewInst is indicating that those instructions must be traced. However, none of the executed instructions cause the trace unit to generate a P0 element, therefore none of the instructions are traced.
- ViewInst then goes low.
- The PE then executes an instruction that, whenever ViewInst is high, causes the trace unit to generate a P0 element.

In this scenario, although ViewInst is low when the instruction in step 3 is executed, indicating that the instruction is not traced, tracing of the instruction is permitted because this is the only way that the preceding instructions can be traced.

There is no requirement for the target address of a P0 instruction or Exceptional occurrence to be traced if ViewInst has transitioned to a low state by the time program execution has moved to the target.

Unless the target instruction block is traced, any Target Address elements indicating the target address of a P0 instruction or Exceptional occurrence cannot be relied on.
Resource event based filtering

The resource event based filtering part of the ViewInst function is expressed as the following equation:

\[ N_i = F_n(\text{TRCVICTLR.EVENT.SEL}, \text{TRCVICTLR.EVENT.TYPE}) \]  (1)

Where \( F_n(\text{TRCVICTLR.EVENT.SEL}, \text{TRCVICTLR.EVENT.TYPE}) \) selects the combination of Resource Selectors used for resource event based filtering.

Exception level filtering

This function provides a simple method of filtering out information about different Exception levels without the need to use additional resources.

ViewInst start/stop function filtering

The ViewInst start/stop function is useful when the requirement is to trace a particular piece of code with all the functions that the piece of code calls.

The ViewInst start/stop function uses the Single Address Comparators and the PE Comparator Inputs to define start points and stop points.

A start point is any of the following:

- The instruction address which matches a Single Address Comparator selected for the ViewInst start/stop function using TRCVISSCTLR.START.
- The instruction address which matches a PE Comparator selected for the ViewInst start/stop function using TRCVIPCSSCTLR.START.
A stop point is any of the following:

- The instruction address which matches a Single Address Comparator selected for the ViewInst start/stop function using TRCVISSCTLR.STOP.
- The instruction address which matches a PE Comparator selected for the ViewInst start/stop function using TRCVIPCSSCTLR.STOP.

Multiple start points can be selected. Multiple stop points can be selected.

When a start point is encountered, the ViewInst start/stop function becomes active for the instruction at the start point.

When a stop point is encountered, the ViewInst start/stop function becomes inactive immediately after the instruction at the stop point.

When the ViewInst start/stop function causes ViewInst to become active, the trace unit traces the instruction at the start address.

When the ViewInst start/stop function causes ViewInst to become inactive, the trace unit traces up to and including the instruction at the stop address.

While a ViewInst start/stop function start address is the same as a stop address, the behavior of the ViewInst start/stop function is UNPREDICTABLE.

The ViewInst start/stop function part of the ViewInst function is expressed as the following equations:

\[
\text{TRCVICTLR.SSSTATUS}_{i+1} = S_i \land \neg \text{Stop}_i
\]

\[S_i = \text{TRCVICTLR.SSSTATUS}_i \lor \text{Start}_i\]

If TRCIDR4.NUMPC == 0b0000 then

\[
\text{Start}_i = \sum_n (\text{SAC}[n] \land \text{TRCVISSCTLR.START}[n])
\]

\[
\text{Stop}_i = \sum_n (\text{SAC}[n] \land \text{TRCVISSCTLR.STOP}[n])
\]

If TRCIDR4.NUMPC != 0b0000 then

\[
\text{Start}_i = \sum_n (\text{SAC}[n] \land \text{TRCVISSCTLR.START}[n])
\lor \sum_m (\text{PECMP}[m] \land \text{TRCVIPCSSCTLR.START}[m])
\]

\[
\text{Stop}_i = \sum_n (\text{SAC}[n] \land \text{TRCVISSCTLR.STOP}[n])
\lor \sum_m (\text{PECMP}[m] \land \text{TRCVIPCSSCTLR.STOP}[m])
\]

The following have no effect on the ViewInst start/stop function:

- Exceptional occurrences.
• Execution in Debug state.
• Execution in a Trace Prohibited region.
• A trace unit buffer overflow.

When disabling the trace unit, the ViewInst start/stop function becomes static and retains its state until the trace unit is enabled again.

For each of TRCVICTLR.SSSTATUS, TRCVISSCTRL, and TRCVIPCSSCTRL, if the register is implemented, it must be programmed with an initial state when the trace unit is programmed before a trace session.

If an implementation makes speculation visible to the trace unit, the ViewInst start/stop function behaves as if no speculation has occurred. That is, when the instruction at a start or stop point is executed speculatively and is later canceled, the ViewInst start/stop function behaves as if the instruction at the start or stop point was not executed.

When the trace unit becomes disabled and there are instructions at start points or stop points which are still speculative, the behavior of the ViewInst start/stop function is IMPLEMENTATION DEFINED and one of the following:

• The ViewInst start/stop function behaves as if the instructions at the start points or stop points were incorrectly speculated. That is, the trace unit behaves as if those start points and stop points did not occur.
• The ViewInst start/stop function behaves as if the instructions at the start points or stop points were correctly speculated. That is, the trace unit updates the state of the ViewInst start/stop function as if those start points and stop points occurred.

When mis-speculation occurs and the PE returns to a point in execution before the trace unit was enabled, the ViewInst start/stop function reverts to the state it was in when the trace unit became enabled.

When a transaction failure occurs the ViewInst start/stop function reverts back to the state to point immediately after the TSTART instruction for the outer transaction.

This is the value of TRCVICTLR.SSSTATUS for the instruction block that contains the TSTART instruction for the outer transaction.

When a transaction failure causes the PE to return to a point in execution before the trace unit was enabled, the ViewInst start/stop function reverts to the state it was in when the trace unit became enabled.

When the trace unit becomes disabled and the PE is executing in Transactional state, the behavior of the ViewInst start/stop function is IMPLEMENTATION DEFINED and one of the following:

• The ViewInst start/stop function behaves as if the transaction failed. That is, the trace unit behaves as if those start points and stop points did not occur.
• The ViewInst start/stop function behaves as if the transaction was successful. That is, the trace unit updates the state of the ViewInst start/stop function as if those start points and stop points occurred.

When tracing becomes prohibited and the PE is executing in Transactional state, the behavior of the ViewInst start/stop function is IMPLEMENTATION DEFINED and one of the following:

• The ViewInst start/stop function behaves as if the transaction failed. That is, the trace unit behaves as if those start points and stop points did not occur.
• The ViewInst start/stop function behaves as if the transaction was successful. That is, the trace unit behaves as if those start points and stop points did occur.
• The ViewInst start/stop function uses the real resolution of the transaction, when that resolution is eventually known.

When the state of the ViewInst start/stop function is changed by anything other than a direct write to TRCVICTLR, the PE considers the write to be an indirect write to TRCVICTLR.SSSTATUS.

In many common usage scenarios, entry to a Trace Prohibited region or disabling of the trace unit does not occur while in a transaction.

Instruction blocks

When an instruction block that contains instructions at ViewInst start points and no instructions at ViewInst stop points is executed, the ViewInst start/stop function remains active for the entire instruction block.
While the ViewInst start/stop function is active, when an instruction block is executed that contains at least one ViewInst stop address and no ViewInst start addresses, the ViewInst start/stop function remains active for the whole instruction block and becomes inactive for the next instruction block, unless the next instruction block contains a ViewInst start address.

When an instruction block that contains at least one instruction at a ViewInst start point and at least one instruction at a ViewInst stop point is executed, the ViewInst start/stop function obeys the order of the start and stop points in the block, with the following consequences:

- The ViewInst start/stop function is active for the whole of the instruction block.
- When the final instruction in the block at a ViewInst start or stop point is at a ViewInst start point, the ViewInst start/stop function is active for the next instruction block.
- When the final instruction in the block at a ViewInst start or stop point is at a ViewInst stop point, the ViewInst start/stop function is inactive for the next instruction block, unless the next block contains an instruction at a new ViewInst start point.

The trace analyzer ensures that for all of the SACs selected for ViewInst start points or stop points, any SAC programmed with a lower address than another SAC is a lower-numbered SAC than the other SAC. That is, the SACs contain addresses in address order.

While the SACs selected for ViewInst do not contain addresses in address order, the behavior of the ViewInst start/stop function is UNPREDICTABLE.

The trace analyzer ensures that for all of the PE Comparator Inputs selected for ViewInst start points or stop points, any PE comparator programmed with a lower address than another PE comparator is a lower-numbered PE comparator than the other PE comparator. That is, the PE comparators contain addresses in address order.

While the PE Comparator Inputs selected for ViewInst do not contain addresses in address order, the behavior of the ViewInst start/stop function is UNPREDICTABLE.

If more than one instruction Address Comparator is programmed with the same instruction address, then programming one or more of those comparators as start comparators, and one or more as stop comparators, results in the following CONSTRAINED UNPREDICTABLE behavior of the ViewInst start/stop function:

- The ViewInst start/stop function is either active or inactive for the instruction at that address.
- The ViewInst start/stop function is either active or inactive after that instruction.

### ViewInst include/exclude function filtering

The ViewInst include/exclude function is useful if:

- Specific ranges of instructions are required to be included in the trace.
- Specific ranges of instructions are required to be excluded from the trace.
- A combination of including and excluding instruction ranges is required.

The ViewInst include/exclude function is comprised of two functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ViewInst include function</td>
<td>Includes one or more instruction address ranges</td>
</tr>
<tr>
<td>ViewInst exclude function</td>
<td>Excludes one or more instruction address ranges</td>
</tr>
</tbody>
</table>

There are between zero and eight instruction Address Range Comparators available for the ViewInst include/exclude function. Some of these comparators can be selected for the ViewInst include function, and some for the ViewInst exclude function.

For example, if all instructions in the address range from 0x00 to 0x2C are required, but no other instructions are required, an Address Range Comparator can be selected for the ViewInst include function and be programmed with these two addresses. All instructions that are in this address range, including those at the start and end addresses, are traced.
The ViewInst include/exclude function differs from the ViewInst start/stop function in the following ways:

- When the ViewInst start/stop function is used, the trace unit starts tracing on a specified start instruction address and stops tracing on a specified stop instruction address. However, if execution branches or jumps to an address between the start and stop points, without first accessing the instruction at the start address, then the instruction that it has branched or jumped to is not traced. Instructions in the start/stop range are only traced if the instruction at the start address is accessed, so that the trace unit is triggered to start tracing. When triggered, and as execution continues sequentially towards the stop address, all functions that the piece of code calls are traced.

- When the ViewInst include/exclude function is used, for example by programming an Address Range Comparator with an include address range, then if execution branches or jumps to any instruction address anywhere in that range, that instruction is always traced. This is true regardless of whether the instruction at the start address has been accessed or not.

In addition, unlike the ViewInst start/stop function, as program execution continues through the address range towards the end address, no functions that the piece of code calls are traced.

The ViewInst include/exclude function part of the ViewInst function is expressed as the following equations:

\[
I_i = \text{Include}_i \land \neg \text{Exclude}_i
\]

\[
\text{Include}_i = \sum_n (\text{ARC}[n] \land \text{TRCVIIECTLR.INCLUDE}[n]) \lor \prod_n \neg \text{TRCVIIECTLR.INCLUDE}[n]
\]

\[
\text{Exclude}_i = \sum_n (\text{ARC}[n] \land \text{TRCVIIECTLR.EXCLUDE}[n])
\]

**Instruction blocks**

- When an instruction in an instruction block is included to be traced by the ViewInst include/exclude function, the ViewInst include/exclude function includes all of the instruction block.

- When an instruction block contains at least one instruction excluded by the ViewInst include/exclude function, and only when all the instructions in the instruction block are excluded, the ViewInst include/exclude function excludes the instruction block.

- If a block of instructions is not entirely covered by at least one individual ARC selected by TRCVIIECTLR.EXCLUDE, it is CONSTRANGED UNPREDICTABLE whether the block is excluded or not. This applies even if other ARCs selected by TRCVIIECTLR.EXCLUDE cover the rest of the block.

**Guidelines for interpreting the ViewInst function result**

The result of the ViewInst function is either:

<table>
<thead>
<tr>
<th>Result</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Indicates that instructions being executed must be traced</td>
</tr>
<tr>
<td>Low</td>
<td>It is expected that instructions being executed are not traced</td>
</tr>
</tbody>
</table>

If it is expected that instructions being executed are not traced, then there are occasions when it is permitted to trace some of those instructions. This section provides guidelines for when it is permitted to trace instructions that ViewInst indicates are not traced.

**When ViewInst transitions from low to high**

If execution occurs while ViewInst is low, it is permitted for a trace unit to trace instructions in certain circumstances. For more information on ViewInst low, see *Occasions when tracing instructions when ViewInst is low is permitted on page D4-4820.*
Tracing of instructions is permitted while ViewInst is low, but if no instructions or Exceptional occurrences that occur are traced, then there is a discontinuity in the trace. When a discontinuity in the trace occurs, when ViewInst becomes high, a Trace On element must be generated.

Any instructions that are executed while ViewInst is high must be traced.

**Occasions when tracing instructions when ViewInst is low is permitted**

ETE permits tracing of instructions when ViewInst is low, in the following scenarios:

- When the instruction that ViewInst indicates is not to be traced is in the same instruction block as an instruction that ViewInst indicates must be traced. This is because the only way to trace the instruction that must be traced is to trace the whole instruction block.
- When the instruction that ViewInst indicates is not to be traced is in an instruction block that precedes or follows an instruction block containing an instruction that ViewInst indicates must be traced.

An implementation always traces the instruction block that contains an instruction that must be traced. However, additional blocks of instructions might also be traced. This is more likely to occur when many instructions are executed in close proximity.

Except for the scenarios that are mentioned, if the ViewInst function indicates that an instruction is not to be traced, then in general it is expected that it is not traced.

In Figure D4-14, the instruction block 1 is in execution order before instruction block 2. The ViewInst calculation for the second block returns true, as indicated by the transition labeled (a). As ViewInst is true for this instruction block then all the instructions in this block must be traced, as indicated by the transition labeled (b). Instruction block 1 might be traced as it is in the same PE cycle as block 2, as indicated by the transition labeled (c).
Rules for tracing Exceptional occurrences

When an Exceptional occurrence occurs, the Exceptional occurrence does not affect the comparators used by the ViewInst function, and none of the comparators used by the ViewInst function match.

The comparators used by the ViewInst function include the following:

- Single Address Comparators.
- Address Range Comparators.
- Context Identifier Comparators.
- Virtual Context Identifier Comparators.

When an Exception element is traced, it might indicate execution of instructions up to a specified address. These instructions might have an impact on the comparators, but the Exceptional occurrence itself does not. This means that when an Exceptional occurrence occurs, the ViewInst function does not indicate whether the Exceptional occurrence must be traced. However, it is useful to trace Exceptional occurrences, to determine why execution has departed from the normal program flow.

When an instruction executes or Exceptional occurrence occurs outside a Trace Prohibited region, the trace unit remembers whether the instruction or Exceptional occurrence was traced. The trace unit performs indirect writes to TRCRSR.TA to store this state. When an Exceptional occurrence occurs, the trace unit uses TRCRSR.TA to determine whether to trace the Exceptional occurrence.

When an instruction executes or Exceptional occurrence occurs outside a Trace Prohibited region and the instruction or Exceptional occurrence is traced, TRCRSR.TA is set to 1.

When an instruction executes or Exceptional occurrence occurs outside a Trace Prohibited region and the instruction or Exceptional occurrence is not traced, TRCRSR.TA is set to 0.

When an instruction or Exceptional occurrence is canceled, TRCRSR.TA is set to the value of TRCRSR.TA immediately before the canceled instruction or Exceptional occurrence.

When an Exceptional occurrence occurs and TRCRSR.TA is 1, the Exceptional occurrence is traced.

While any of the following are true, TRCRSR.TA is unchanged by any execution:
- The PE is in Debug state.
- The PE is in a Trace Prohibited region.

When a trace unit buffer overflow occurs, the behavior of TRCRSR.TA is IMPLEMENTATION DEFINED and is one of the following:
- TRCRSR.TA is set to 0.
- TRCRSR.TA is set to the value of TRCRSR.TA for the most recent instruction or Exceptional occurrence before the trace unit buffer overflow occurred.

Forced tracing of Exceptional occurrences

The trace unit can be programmed so that it always traces certain Exceptional occurrences, regardless of whether the instruction or Exceptional occurrence immediately before the Exceptional occurrence must be traced. This option is enabled by setting either or both:
- TRCVICTLR.TRCERR to 1. This forces the trace unit to trace System Error exceptions regardless of the value of ViewInst.
- TRCVICTLR.TRCRESET to 1. This forces the trace unit to trace PE Resets regardless of the value of ViewInst.

While the PE is executing in a Trace Prohibited region, forced tracing of System Error exceptions is inactive.

While the PE is not executing a Trace Prohibited region and forced tracing of System Error exceptions is enabled, forced tracing of System Error exceptions is active.
While forced tracing of System Error exceptions is active, when a System Error exception occurs, the trace unit generates an *Exception element* indicating a System Error exception, regardless of the value of ViewInst.

While the PE is executing in a Trace Prohibited region, forced tracing of PE Resets is inactive, regardless of whether the PE Reset causes the PE to leave a Trace Prohibited region or not.

While the PE is not executing in a Trace Prohibited region, while forced tracing of PE Resets is enabled, forced tracing of PE Resets is active.

While forced tracing of PE Resets is active, when a PE Reset occurs, the trace unit generates an *Exception element* indicating a PE Reset, regardless of the value of ViewInst.

While tracing is inactive, before an *Exception element* is generated due to forced tracing of either a PE Reset or a System Error exception, the trace unit generates a *Trace On element* and then a *Target Address element*.

When an *Exception element* is generated as a result of forced tracing, the *Trace On element* generated before the *Exception element* indicates that tracing becomes active, and the *Target Address element* indicates where tracing becomes active.

When a System Error exception occurs and TRCRSR.TA is 0 and the exception is traced because forced tracing of System Error exceptions is enabled, then it is IMPLEMENTATION DEFINED whether TRCRSR.TA is set to 1 or remains at 0.

When a PE Reset occurs and TRCRSR.TA is 0 and the PE Reset is traced because forced tracing of PE Resets is enabled, then it is IMPLEMENTATION DEFINED whether TRCRSR.TA is set to 1 or remains at 0.

In scenarios where a System Error exception occurs at approximately the same time as an exit from a Trace Prohibited region, after all execution inside the Trace Prohibited region and before any instruction execution outside the Trace Prohibited region, it is UNPREDICTABLE whether the System Error exception is considered to have occurred inside or outside the Trace Prohibited region. It is also UNPREDICTABLE whether the forced tracing of System Error exceptions is active for this exception.

These scenarios do not include scenarios where the System Error exception caused the exit from a Trace Prohibited region, because the System Error exception occurred inside the Trace Prohibited region.

In scenarios where a System Error exception occurs at approximately the same time as an entry to a Trace Prohibited region, after all execution before the Trace Prohibited region and before any instruction execution inside the Trace Prohibited region, it is UNPREDICTABLE whether the System Error exception is considered to have occurred inside or outside the Trace Prohibited region. It is also UNPREDICTABLE whether the forced tracing of System Error exceptions is active for this exception.

These scenarios do not include scenarios where the System Error exception caused the entry to a Trace Prohibited region, because the System Error exception occurred outside the Trace Prohibited region.

When a System Error exception occurs immediately after the PE exits a Trace Prohibited region and the System Error exception is traced, the preferred exception return address in the *Exception element* indicating the System Error exception does not include information about the Trace Prohibited region.

In scenarios where a PE Reset occurs at approximately the same time as an exit from a Trace Prohibited region, after all execution inside the Trace Prohibited region and before any instruction execution outside the Trace Prohibited region, it is UNPREDICTABLE whether the PE Reset is considered to have occurred inside or outside the Trace Prohibited region. It is UNPREDICTABLE whether the forced tracing of PE Resets is active for this PE Reset.

These scenarios do not include scenarios where the PE Reset caused the exit from a Trace Prohibited region, because the PE Reset occurred inside the Trace Prohibited region.

In scenarios where a PE Reset occurs at approximately the same time as an entry to a Trace Prohibited region, after all execution before the Trace Prohibited region and before any instruction execution inside the Trace Prohibited region, it is UNPREDICTABLE whether the PE Reset is considered to have occurred inside or outside the Trace Prohibited region. It is UNPREDICTABLE whether the forced tracing of PE Resets is active for this PE Reset.

These scenarios do not include scenarios where the PE Reset caused the entry to a Trace Prohibited region, because the PE Reset occurred outside the Trace Prohibited region.
When a PE Reset occurs immediately after the PE exits a Trace Prohibited region and the PE Reset is traced, the preferred exception return address in the Exception element indicating the PE Reset does not include information about the Trace Prohibited region.

### D4.5.9 Element Generation

#### Trace Info element generation

When a trace protocol synchronization request is serviced, the trace unit generates a Trace Info element.

Note: There is no requirement to generate a new Trace Info element every time that ViewInst becomes active. This is because, despite the discontinuity in the trace that is caused by the filtering, the programming of the trace remains the same.

While the PE is in Transactional state and the trace unit has previously generated a Transaction Start element for this transaction, when a Trace Info element is generated, the trace unit sets the Transactional state indicator in the Trace Info element to 1.

While the PE is not in Transactional state, or the PE is in Transactional state but the trace unit has not generated a Transaction Start element for this transaction, when a Trace Info element is generated, the trace unit sets the Transactional state indicator in the Trace Info element to 0.

When the trace unit generates the first Trace Info element after an Overflow element, the Transactional state indicator is set to 0.

When an Overflow element is generated, before any subsequent P0 elements indicating execution in Transactional state are traced, the trace unit generates a new Transaction Start element, even if a Transaction Start element has previously been traced for this transaction prior to the Overflow element.

#### Atom element

When a P0 instruction is taken, the trace unit generates one of the following:

- An E Atom element.
- A Source Address element.

When a P0 instruction is not taken, the trace unit generates one of the following:

- An N Atom element.
- Nothing.

When a P0 instruction is not taken and the trace unit does not generate an N Atom element, for all future not taken P0 instructions until the next taken P0 instruction or Exceptional occurrence, the trace unit does not generate an N Atom element.

When a P0 instruction is not taken and the trace unit does not generate an N Atom element, when an Exceptional occurrence occurs before the next taken P0 instruction, the trace unit generates an Exception element.

When a P0 instruction is not taken and the trace unit does not generate an N Atom element, when no Exceptional occurrence occurs before the next taken P0 instruction, the trace unit generates a Source Address element for the next taken P0 instruction.

When a P0 instruction is not taken and the trace unit does not generate an N Atom element, and the P0 instruction is subsequently mispredicted, the trace unit generates a Source Address element and does not generate a Mispredict element.

The trace unit generates Atom elements in the program order in which they occur, and the trace protocol encode and decode process maintains this order.

For conditional branch instructions, an E Atom element indicates that the instruction passed its condition code check, and an N Atom element indicates that the instruction failed its condition code check, although a trace unit might use a Mispredict element to modify the Atom element.
The trace unit might trace unconditional P0 instructions using an E Atom element or an N Atom element.

When an unconditional P0 instruction is traced using an N Atom element, the trace unit generates either a Mispredict element or a Cancel element to correct the N Atom element.

When an ISB instruction does not pass the condition code check, and the ISB instruction does not perform a Context synchronization event, the trace unit treats the ISB instruction as a not taken instruction.

When an ISB instruction does not pass the condition code check, and the ISB instruction performs a Context synchronization event, the trace unit treats the ISB instruction as a taken instruction.

When an ISB instruction passes the condition code check, the trace unit treats the ISB instruction as a taken instruction.

Note: For an ISB instruction, a trace analyzer must not infer the value of the PSTATE condition flags from an Atom element.

It is IMPLEMENTATION DEFINED whether the trace unit classifies WFI, WFIT, WFE, and WFET instructions as P0 instructions. When WFI, WFIT, WFE, and WFET are classified as P0 instructions, execution of these instructions generates an Atom element. See Low-power state on page D4-4802 and TRCIDR2.WFXMODE.

When WFI, WFIT, WFE, and WFET instructions are classified as P0 instructions and a conditional WFE or WFI instruction is executed, if the instruction passes its condition code check then an E Atom element is generated.

Note: For a WFI, WFIT, WFE, or WFET instruction classified as a P0 instruction, a trace analyzer must not infer the value of the PSTATE condition flags from an E Atom element.

P0 instructions that fail or are predicted to fail their condition code check either generate an Undefined Instruction exception or are executed as a NOP, if the instruction is also UNDEFINED.

When a P0 instruction fails or is predicted to fail its condition code check, and the P0 instruction is executed as a NOP, the trace unit generates an N Atom element for the P0 instruction.

When a P0 instruction fails or is predicted to fail its condition code check, and the P0 instruction generates an Undefined Instruction exception, the trace unit does not generate an Atom element for the instruction and generates an Exception element instead. The preferred exception return address for the generated Exception element is the UNDEFINED instruction, which indicates that the instruction did not execute.

The trace unit generates all Atom elements speculatively, and explicitly resolves or cancels each Atom element by generating Commit elements or Cancel elements.

A trace analyzer can infer execution from an Atom element, but only after the Atom element has been resolved by a Commit element.

For taken direct P0 instructions, a trace analyzer must infer the target address and instruction set of the instruction from the opcode in the program image.

If a taken direct P0 instruction is from a branch broadcasting region, the trace analyzer does not need to infer the target address and instruction set because this is explicitly traced using a Target Address element.

**Exception element**

When an Exceptional occurrence occurs, if the Exceptional occurrence is required to be traced, the trace unit generates an Exception element.

The trace unit generates Exception elements in program order relative to other P0 elements.
To be consistent with the rules for generating Target Address elements, under the following scenarios the trace unit must generate a Target Address element before an Exception element, unless the Target Address element would be removed due to a return stack match:

- The Exceptional occurrence is taken from the target of a taken indirect P0 instruction.
- The Exceptional occurrence is taken from the target of a taken direct P0 instruction in a branch broadcasting region.
- The Exceptional occurrence is taken from the target of another Exception element.

When an Exceptional occurrence occurs, if the Context information changes at the target of the P0 element preceding the Exceptional occurrence, then the trace unit generates a Context element before the Exception element. The Context element provides Context information about the address and context where the Exceptional occurrence was taken from.

An invalid address is one where bits [63:P] are not all zeros or all ones, where P is defined as the maximum virtual address size supported by the PE.

When the PE attempts to execute an instruction at an invalid address and the trace unit generates an Exception element, the preferred exception return address in the Exception element indicates one of the following:

- The full 64-bit invalid address.
- Any other invalid address, with address bits [P-1:0] the same as the full invalid address.

Arm recommends that when the PE attempts to execute an instruction at an invalid address and the trace unit generates an Exception element, the preferred exception return address in the Exception element indicates the full 64-bit invalid address.

**Source address element**

When a P0 instruction which must be traced is not taken and the trace unit does not generate an N Atom element, then when a subsequent P0 instruction is taken, the trace unit generates a Source Address element.

A trace unit can generate a Source Address element to imply that at least one instruction has been executed, including a taken P0 instruction.

When the trace unit generates a Source Address element to imply that a taken P0 instruction has been executed, the address associated with the Source Address element is the virtual address of the taken P0 instruction.

**Q element**

A trace unit can generate a Q element to imply that at least one instruction has been executed, possibly including P0 instructions.

When a Q element is generated, the trace unit generates a Target Address element that indicates where execution is to continue after all the instructions that are implied by the Q element have been executed.

When a Q element is generated and the last instruction implied by the Q element is a P0 instruction, the trace unit generates a Target Address element that indicates the target of the P0 instruction.

When a Q element is generated and the last instruction implied by the Q element is not a P0 instruction, the trace unit generates a Target Address element that indicates the instruction address immediately following the last instruction that is implied by the Q element.

When the PE leaves a region where Q elements are permitted, either by a P0 instruction or by sequential execution out of the region, and a Q element implies the execution of the last instruction in the region, the Q element does not imply any more instructions after the last instruction in the region.

When the PE enters a region where Q elements are permitted, either by a P0 instruction or by an Exceptional occurrence, the trace unit traces the P0 instruction or Exceptional occurrence using elements other than Q elements.

Note: Although the trace unit does not trace with Q elements a P0 instruction or Exceptional occurrence that causes the PE to enter a region where Q elements are permitted, any subsequent instructions in the region where Q elements are permitted might be traced using Q elements.
When the PE enters by sequential execution a region where $Q$ elements are permitted, any instructions that are executed since the last $P0$ element outside the permitted region might be traced using a $Q$ element. These instructions can always be inferred unambiguously from the $Q$ element.

When the PE enters by sequential execution a region where $Q$ elements are permitted, and $P0$ instructions executed since the last $P0$ element outside the permitted region are traced by a $Q$ element, the $Q$ element does not indicate execution of any $P0$ instructions outside the permitted region.

**Event element**

The trace unit generates *Event elements* independently of ViewInst.

While TRCEVENTCTL1R.INSTEN$<n>$ is 1 and the resource event selected by TRCEVENTCTL0R.EVENT$<n>$ is active, while trace generation is operative, the trace unit generates an *Event element* $<n>$ on each PE clock cycle.

When an *Event element* is generated between two $P0$ elements or at the same time as a $P0$ element that follows another, the trace unit inserts the *Event element* after the first $P0$ element but before the $P0$ element that is an IMPLEMENTATION DEFINED number of $P0$ elements after the first $P0$ element.

Arm recommends that the IMPLEMENTATION DEFINED number of $P0$ elements is less than or equal to the number of $P0$ elements the PE can process simultaneously.

While trace generation is inoperative due to a trace unit buffer overflow, when a programmed ETEEvent $<n>$ occurs, the trace unit generates at least one *Event element* $<n>$ before it generates the *Overflow element* corresponding to the trace unit buffer overflow.

**Cancel element generation**

When one or more $P0$ elements are canceled, the trace unit generates a *Cancel element*.

The trace unit generates *Cancel elements* in execution order relative to $P0$ elements.

When a *Cancel element* causes execution to return to a point in the program flow that is not adjacent to a $P0$ instruction, the trace unit generates an *Exception element* that indicates which instructions were executed up to that point in the program flow before it generates any $P0$ elements.

**Commit element generation**

When one or more traced $P0$ elements are resolved for execution, the trace unit generates a *Commit element*.

An *Atom element* might be corrected using a *Mispredict element* after it has been resolved.

The trace unit never generates more speculative $P0$ elements than the maximum speculation depth of the trace unit.

When trace generation becomes inoperative due to the trace unit being disabled, the trace unit outputs any *Commit elements* which have not been output.

If cycle counting is enabled some *Commit elements* have *Cycle Count elements* associated with them, that provide counts of processor clock cycles. The cycle count values given in *Cycle Count elements* can be used to obtain a cumulative count.

*Commit elements* with associated *Cycle Count elements* cannot be merged with later *Commit elements*.

For more information, see *Cycle Count element* on page D4-4786.

**Transaction Start element**

When the PE enters an outer transaction, before the first instruction is traced, the trace unit generates a *Transaction Start element*.

A *Transaction Start element* is not required for each *Trace On element* if the instructions are all part of the same outer transaction.
When the PE leaves Transactional state and a Transaction Start element was generated for the transaction, the trace unit traces the result of the transaction using one of the following:

- A Transaction Commit element, if the transaction was successful.
- A Transaction Failure element, if the transaction failed.
- A Cancel element which cancels the Transaction Start element.

The trace element stream only indicates that the PE is in Transactional state. It does not indicate the transactional nesting depth.

**Transaction Commit element**

When the PE exits Transactional state successfully, and a Transaction Start element was generated for the transaction, the trace unit generates a Transaction Commit element.

When a Transaction Commit element is generated, the trace unit traces the Transaction Commit element after the P0 element which is generated before the TCOMMIT instruction, and before the next Transaction Start element is traced.

Arm recommends that the Transaction Commit element is generated and output as soon as possible after the PE leaves Transactional state.

The rules on the Transaction Commit element mean that a Transaction Commit element is permitted to be output later than the P0 element which implies execution of the TCOMMIT instruction.

The TCOMMIT instruction is not a P0 instruction. This means that the Transaction Commit element might be traced before the P0 element which implies execution of the TCOMMIT instruction.

**Transaction Failure element**

When a transaction failure occurs, and a Transaction Start element was generated for the transaction, the trace unit generates a Transaction Failure element.

When the PE enters a Trace Prohibited region and is in Transactional state, and a Transaction Start element was generated for the transaction, the trace unit generates a Transaction Failure element.

When the trace unit becomes disabled and the PE is in Transactional state, and a Transaction Start element was generated for the transaction, the trace unit generates a Transaction Failure element.

When a trace unit buffer overflow occurs and the PE is in Transactional state, and a Transaction Start element was generated for the transaction, the trace unit generates a Transaction Failure element.

A Transaction Failure element is encoded as an Exception packet with a type of Transaction Failure.

When a Transaction Failure element is generated, the following behavior applies:

- The target address and target context of the previous P0 element might be UNKNOWN.
- If there are no P0 elements between a Trace On element and the Transaction Failure element, the initial address and context after the previous Trace On element might be UNKNOWN.

When a PE reset occurs and the PE is in Transactional state, and a Transaction Start element was generated for the transaction, the trace unit generates a Transaction Failure element.

**Context element**

The trace unit generates a Context element in the following situations:

- While tracing is active, when any of the Context information changes, before any P0 element which indicates execution from the new context.
- After a Trace Info element is generated due to a non-periodic trace protocol synchronization request, and before any P0 element.
- After a Trace Info element is generated due to a periodic trace protocol synchronization request.
• When mis-speculation results in an incorrect Context element being output, before any P0 element which indicates execution from the correct context.


While Virtual context identifier tracing is enabled and TRFCR_EL2.CX disallows the tracing of the Virtual context identifier, when the trace unit generates a Context element, the Virtual context identifier in the Context element has the value 0x0.

A Context element might also be output at other points, which might include after all Context synchronization events, or at any other point at which the Context information changes.

If the highest implemented Exception level is using AArch64, the Context identifier value is the value of CONTEXTIDR_EL1.

Some of the Context information might change at points other than at Context synchronization events. These changes occur when system instructions are used to change a piece of Context information, including:

• Writes to the current CONTEXTIDR_EL1.
• Writes to the CONTEXTIDR_EL2.
• Changes from Secure to Non-secure state without using an exception return.
• Changes in Exception level other than due to an exception or an exception return.

When a System instruction writes to a System register and the Context information changes, the trace unit generates a Context element containing the new context value, after the P0 element prior to the System instruction but before the P0 element following a Context synchronization event after the System instruction.

Note: If the Context element is output before the first P0 element after the System instruction, this might imply that some instructions before the System instruction were executed with the new context. This is acceptable because the code which changes the context is usually executed in a state where it does not matter whether the old or new context values are used.

If the PE takes an exception after performing a write to a System register that changes the context, but a P0 element has not been generated since the write, then a Context element indicating the new context is not required to be output before the Exception element. This is because no instructions or Exceptional occurrences are indicated to have been executed from the new context. A Context element indicating the new context must be generated after the Exception element if the Exceptional occurrence is a Context synchronization event. If the Exceptional occurrence changes the context, then the Context element must indicate the new context. This might happen if, for example, the Security state changes.

When a PE reset occurs, until the relevant PE registers are updated, the trace unit traces the Context identifier and Virtual context identifier as zero.

A trace unit is not required to generate a Context element if tracing becomes inactive before any instructions are executed in the new context.

Additional Context elements might be output by a trace unit in some scenarios, but these must only be output where they do not affect the analysis of the trace element stream. Such a scenario might include when a change in the Context information is incorrectly speculated and a subsequent Context element corrects the value of a previous incorrect Context element. Arm recommends that the generation of additional unnecessary Context elements is minimized to ensure trace bandwidth is minimized.

Target Address element

When the trace analyzer cannot infer the address or instruction set from the trace, the trace unit generates a Target Address element.

Occasions when the trace analyzer might not be able to infer the address or instruction set from previous trace include:

• At the target of an indirect P0 instruction which is taken.
• At the target of a direct P0 instruction which is taken in a branch broadcasting region, see TRCBBCTRL for more information.
• At the target of an Exceptional occurrence.
• At the target of a Transaction Failure element.
• When mis-speculation occurs and the address cannot be inferred.
• After a Q element is generated.

When the trace analyzer cannot infer the address or instruction set from the trace, the trace unit generates the resulting Target Address element before the next P0 element, unless any of the following are true:
• The Target Address element can be omitted because of a return stack match.
• Tracing is inactive at the target of the P0 instruction or Exceptional occurrence.
• A transaction failure occurs and tracing is inactive at the target of the transaction failure.

When non-periodic trace protocol synchronization occurs, the trace unit generates a Target Address element after the Trace Info element and Trace On element corresponding to the non-periodic trace protocol synchronization, and before the next P0 element is generated.

When periodic trace protocol synchronization occurs, after the corresponding Trace Info element has been generated, the trace unit generates a Target Address element containing the address of the target of the most recent P0 element before the Target Address element.

When non-periodic trace protocol synchronization occurs, the Target Address element does not need to indicate the target of the most recent P0 element, since tracing might not become active at the target of a P0 element.

When periodic trace protocol synchronization occurs, the Target Address element needs to indicate the target of the most recent P0 element, since tracing is continuing from that P0 element. Furthermore, the Target Address element might indicate the target of a P0 element from before the Trace Info element.

When a Trace On element is generated, the trace unit generates a Target Address element before the next P0 element.

Typically, a Target Address element is required after an Exception element to indicate the target of the Exceptional occurrence, since a trace analyzer is not usually able to infer the target of an Exceptional occurrence.

In some scenarios, an Exception element might be generated in the trace where the Exceptional occurrence target address is the next sequential instruction from the last instruction before the Exceptional occurrence. This behavior depends on many factors and might only occur for IMPLEMENTATION DEFINED Exceptional occurrences. If an Exceptional occurrence is taken to the next sequential instruction, the trace unit is not required to output a Target Address element indicating the target address of the Exceptional occurrence because this can be inferred from the previous execution.

A trace analyzer needs both a Target Address element and a Context element before it can determine the instruction set in use, because the Target Address element provides the instruction set and the Context element provides information on whether the PE is in AArch32 or AArch64 state.

When a change of instruction set occurs that switches between AArch32 state and AArch64 state, the trace unit generates a Context element indicating the new state.

An invalid address is one where bits [63:P] are not all zeros or all ones, where P is defined as the maximum virtual address size supported by the PE.

When the PE attempts to execute an instruction at an invalid address and the trace unit generates a Target Address element, the Target Address element indicates one of the following:
• The full 64-bit invalid address.
• Any other invalid address, with address bits [P-1:0] the same as the full invalid address.

Arm recommends that when the PE attempts to execute an instruction at an invalid address and the trace unit generates a Target Address element, the Target Address element indicates the full 64-bit invalid address.

While tagged addresses are in use, the virtual address in the trace element stream does not include the tag and is the PC value, that is, depending on the state of the PE at the address, bits[63:56] are one of the following:
• The sign-extension of bit[55].
• Zero.
The Translation Control Registers, TCR_ELx, contain the TBI field for controlling whether to ignore the top byte of an address. If the current TBI field is changed from 0 to 1, and before the next Context synchronization event the PE takes an exception because of an invalid top address byte, the branch target address to the invalid address or the preferred exception return address of the Exception element might not contain the full invalid address and might contain the address with the top byte masked. Furthermore, the branch target address might be the invalid address and therefore might be different from the preferred exception return address. Trace analysis tools must be aware that if a branch target address is substantially different from a preferred exception return address which follows, then there might have been a change in the TBI field which caused this large change in address.

When a pointer authentication check fails and an exception is taken from the resulting invalid address, the preferred exception return address is one of the following:

- The full 64-bit invalid address.
- Any other invalid address, with address bits [P-1:0] the same as the full invalid address.

Arm recommends that when a pointer authentication check fails and an exception is taken from the resulting invalid address, the preferred exception return address is the full 64-bit invalid address.

The bottom bits of an address are ignored, in the following way:

- Bits[1:0] of addresses that are used in A64 or A32 instructions are always traced as zero.
- Bit[0] of addresses that are used in T32 instructions is always traced as zero.

Additional Target Address elements might be output by a trace unit in some scenarios, but these must only be output where they do not affect the analysis of the trace element stream. These scenarios include, but are not limited to:

- When an instruction address is incorrectly speculated, and a subsequent Target Address element corrects the value of a previous incorrect Target Address element.
- When an instruction address can be inferred by the trace analyzer, for example for the target of a direct P0 instruction, but a Target Address element is output anyway with the same address.

Arm recommends that the generation of additional unnecessary Target Address elements is minimized to ensure trace bandwidth is minimized.

Mispredict element

When the status of the last non-canceled Atom element has been changed by the PE, the trace unit generates a Mispredict element.

The trace unit only generates a Mispredict element to change the status of an Atom element.

A trace unit might generate multiple Mispredict elements for the same Atom element. A trace analyzer must use each Mispredict element to determine the final status of the Atom element. For example, if an E Atom element has two Mispredict elements, the first Mispredict element indicates the Atom element is an N Atom element and the second Mispredict element indicates it is an E Atom element.

If a PE mispredicts only the target address of a P0 element, then it does not generate a Mispredict element.

The trace unit uses a Target Address element to correct the mispredicted target address. When analyzing a Mispredict element, any Target Address elements between the mispredicted Atom element and the Mispredict element must be discarded.

Overflow element

When a trace unit buffer overflow occurs, after all trace elements that were generated prior to the trace unit buffer overflow are output, the trace unit outputs an Overflow element.

When a trace unit buffer overflow occurs, and the trace unit is disabled after recovering from the trace unit buffer overflow, the trace unit outputs the corresponding Overflow element before the trace unit becomes idle.
**Timestamp element**

*RYYWTR* While TRCCONFIGR.TS is 1 and any of the following occur, a timestamp request occurs:

- The timestamp resource event occurs, as controlled by TRCTSCCTRL.
- The trace unit generates a Trace Info element.
- The trace unit recovers from a trace unit buffer overflow.
- When not in a Trace Prohibited region and a Context synchronization event is caused by any of the following:
  - The PE takes an exception.
  - The PE returns from an exception.
  - An ISB instruction is executed.
- A trace unit flush is requested.

*RCKNFV* While TRCCONFIGR.TS is 1 and when not in a Trace Prohibited region, a timestamp request might occur when any of the following occur but do not cause a Context synchronization event:

- The PE takes an exception.
- The PE returns from an exception.
- An ISB instruction is executed.

*RNNGXNQ* The state of the ViewInst function does not affect whether a timestamp request occurs.

*RHZSVP* When a timestamp request occurs and ViewInst is inactive, the timestamp request is permitted to be delayed until after the first of the following occurs:

- A P0 element is generated.
- An Event element is generated.

*IWFXVV* There is no requirement for a Timestamp element to be generated in the trace element stream on each occasion that ViewInst becomes active.

*RDRMSV* When a timestamp request occurs and is not ignored, the trace unit generates a Timestamp element.

*RDRWCYP* When a timestamp request occurs but the trace unit does not have the capacity to generate the Timestamp element immediately, then the generation of the Timestamp element is delayed until there is capacity.

*IFTQDHY* A trace unit might not have the capacity to generate a Timestamp element for multiple reasons, including avoiding a trace unit buffer overflow. A delayed Timestamp element means that a timestamp value might not be the exact time of the incident that resulted in the timestamp request. A timestamp is only a time indicator inserted in the trace element stream somewhere near the time of the request.

*RXMSGY* When a timestamp request occurs while in a Trace Prohibited region, then the generation of the Timestamp element is delayed until the PE leaves the Trace Prohibited region.

*RCCWHIIW* When the first timestamp request occurs after trace generation becomes operative, the trace unit delays generation of the corresponding Timestamp element until after the trace unit has generated either a P0 element or an Event element.

*ISDPZZ* This is so that the timestamp value can correspond to the most recent of these elements.

*RSZNMB* A timestamp request is permitted to be ignored if a previous timestamp request has not yet generated a Timestamp element, due to a delay in the generation.

*RDWFTT* A trace unit might ignore the second request of two successive timestamp requests if all of the following are true:

- The second request is not caused as a result of a trace protocol synchronization request.
- No P0 elements or Event elements have been generated between the two requests.
- The timestamp value would be the same in both Timestamp elements.

*RZMQILT* While TRCCONFIGR.CCI is 1, each Timestamp element contains a cycle count that indicates the number of cycles between the previous Cycle Count element and the element with which the Timestamp is associated.
The cycle count associated with a Timestamp element is different from the Cycle Count element in the following ways:

- The cycle count does not affect the cumulative cycle count.
- The cycle count value can be zero.

When the cycle count associated with a Timestamp element is zero, this indicates that no cycles passed between the previous Cycle Count element and the element with which the Timestamp element is associated.

When the trace unit is first enabled, while cycle counting is enabled, when a Timestamp element is generated before any Cycle Count elements, the Timestamp element reports the cycle count as UNKNOWN.

When a Timestamp element is generated and the cycle counter has exceeded the maximum supported value, the Timestamp element reports the cycle count as UNKNOWN.

**Trace On element**

When an instruction block is traced immediately after an instruction block was not traced or a trace unit buffer overflow occurred, the trace unit generates a Trace On element.

When an Exception element indicating a PE reset is traced, the preferred exception return address is UNKNOWN. Any instructions since the most recent unresolved P0 element are not traced. If ViewInst was active for these instructions, this is not considered a gap in the trace element stream and a Trace On element is not required.

In some scenarios where mis-speculation occurs or instructions are canceled, after Cancel elements have been processed there might be Trace On elements in the trace element stream even though no trace discontinuity occurred in the architecturally-executed instruction trace. This typically only occurs when the trace is filtered using the ViewInst function, which causes the Trace On element to be inserted.

Trace analyzers must be aware that these additional Trace On elements might be traced.

**Cycle Count element**

The cycle counter has an IMPLEMENTATION DEFINED size of between 12 and 20 bits, as indicated by TRCIDR2.CCSIZE. The cycle counter therefore supports values from 1 to $2^{20} - 1$.

While TRCCONFIGR.CCI is 1 and the cycle count is equal to or greater than the value of TRCCCCTLR.THRESHOLD, when a Commit element is generated, a Cycle Count element request occurs.

While TRCCCCTLR.THRESHOLD is programmed with a value less than TRCIDR3.CCITMIN, the generation of Cycle Count elements is CONSTRAINED UNPREDICTABLE.

When a request for a Cycle Count element occurs, one of the following occurs:

- The trace unit generates a Cycle Count element immediately and before any future Commit element.
- The trace unit delays generation of the Cycle Count element until after one or more further Commit elements have been generated.

Arm recommends that when a request for a Cycle Count element occurs, the Cycle Count element is generated immediately, and that Cycle Count element generation is only delayed in rare and non-repetitive circumstances.

When a Cycle Count element is generated, the Cycle Count element contains the value of the cycle counter at the time the most recent Commit element was generated, and the cycle counter is reset to the number of cycles since the most recent Commit element was generated.

A value of 0 indicates that the cycle count value is UNKNOWN.

When the cycle counter exceeds the maximum supported value, the cycle count value is UNKNOWN.

When the trace unit becomes enabled, an UNKNOWN cycle count value occurs for the first Cycle Count element generated.

When a trace unit buffer overflow occurs, an UNKNOWN cycle count value occurs for the first Cycle Count element generated.
The first Cycle Count element after the PE clock has been restarted should have an UNKNOWN cycle count.

**Discard element**

When trace generation becomes inoperative and any of the following are true, the trace unit generates a Discard element:

- P0 elements have been generated but the trace unit is unable to output the resolution of those P0 elements.
- A Transaction Start element has been generated and trace generation becomes inoperative before the transaction either succeeds or fails.

When trace generation becomes inoperative due to the trace unit becoming disabled, and a Discard element is generated, the trace unit outputs the Discard element after all other elements.

When a PE reset occurs and any of the following are true, the trace unit generates a Discard element:

- P0 elements have been generated but the trace unit is unable to output the resolution of those P0 elements.
- A Transaction Start element has been generated and the PE reset occurs before the transaction either succeeds or fails.

A trace unit might not generate a Discard element if no P0 elements are speculative.

A trace unit might generate a Discard element even if no P0 elements are speculative.

When a Discard element is generated, all uncommitted P0 elements are discarded, that is, the current speculation depth is set to zero.

When a Discard element is generated, and a Transaction Start element has been traced but the transaction has not succeeded or failed, the trace unit does not indicate the resolution of the transaction.

When a Discard element is generated and tracing subsequently becomes operative for the same transaction, the trace unit generates a new Transaction Start element before any P0 elements are generated for the transaction.

**Trace unit features**

The architecture defines a number of optional and mandatory features that are provided to modify the trace element stream to provide additional information to aid debugging. These features include the following:

- Q element regions.
- Branch broadcasting.
- Context identifier tracing.
- Cycle counting.
- Event trace.
- No overflow.
- PE Stalling and overflow avoidance.
- Timestamping.
- Virtual context identifier tracing.

For the optional features, the inclusion of these optional features is indicated in TRCIDR0-TRCIDR13.

**Q regions**

The support for Q element is OPTIONAL, as indicated by TRCIDR0.QSUPP.

The use of Q elements must be explicitly enabled if the trace unit is to use them.

While Q elements are enabled, the trace element stream might not contain enough information to determine the complete program flow, because some changes in flow might not be explicitly indicated.
Arm recommends that *Q elements* are only used in cases where generating the full ETE instruction trace element stream might cause the performance of the PE being traced to degrade significantly.

The use of *Q elements* degrades the information that can be extracted from the trace element stream. Arm recommends that *Q element* filtering, as indicated by TRCIDR0.QFILT, is also implemented.

If TRCQCTRLR is implemented, the trace unit supports the ability to control when *Q elements* are permitted in the trace element stream using ARCs. The *Q element* filtering operates in either Exclude mode, or Include mode, selected by TRCQCTRLR.MODE.

If *Q elements* are enabled and *Q element* filtering is in Include mode, the ARCs selected by TRCQCTRLR.RANGE define where *Q elements* are permitted.

If *Q elements* are enabled and *Q element* filtering is in Exclude mode, the ARCs selected by TRCQCTRLR.RANGE define where *Q elements* are not permitted.

When an instruction block contains at least one instruction where *Q elements* are permitted, the entire instruction block is permitted to generate *Q elements*.

The following equation is calculated for each instruction block and defines when *Q elements* are permitted.

\[
Q_{PERMITTED_i} = \begin{cases} 
\sum_n ARC_i(n) \land TRCQCTRLR.RANGE[n] & \text{Include mode} \\
\sum_n ARC_i(n) \land \neg TRCQCTRLR.RANGE[n] & \text{Exclude mode}
\end{cases}
\]

While TRCCONFIGR.QE indicates that *Q elements* are disabled, the trace unit does not generate any *Q elements*.

While TRCCONFIGR.QE indicates that *Q elements* are disabled, the trace unit is able to generate all of the elements required to trace the instruction sequence.

**Branch broadcasting**

The branch broadcasting feature forces the trace unit to explicitly trace the target addresses of taken direct *P0 instructions*.

The target addresses are traced using *Target Address elements* in the instruction trace stream.

Branch broadcasting is enabled by performing both of the following actions:

- Setting TRCCONFIGR.BB to 1.
- Programming TRCBBCTLR to specify how branch broadcasting behaves. TRCBBCTLR selects Address Range Comparators to define when branch broadcasting is active, and selects the operating mode of branch broadcasting:
  - Branch broadcasting is active for all instructions inside the selected ranges. This is known as Include mode.
  - Branch broadcasting is active for all instructions outside the selected ranges. This is known as Exclude mode.

When a direct *P0 instruction* for which branch broadcasting is active is taken, the trace unit generates a *Target Address element* to explicitly trace the target of the instruction, regardless of whether the *P0 instruction* is mispredicted.

While branch broadcasting is enabled, while the return stack is enabled, the trace unit prioritizes branch broadcasting over the return stack, that is, the return stack does not match on the target of any instruction for which branch broadcasting is active.

If TRCBBCTLR is not implemented, while branch broadcasting is enabled, branch broadcasting is active for all instructions.
Context identifier tracing

The trace unit can be programmed to include information about the current execution context of the program being executed on the PE, including:

- The current process identifier, stored in CONTEXTIDR_EL1. This is known as the Context identifier.
- The current VMID, stored in CONTEXTIDR_EL2. This is known as the Virtual context identifier.

The trace unit supports tracing of the Context identifier, with TRCIDR2.CIDSIZE indicating a 32-bit Context identifier size.

Cycle counting

The use of the cycle counting feature introduces Cycle Count elements into the trace element stream to indicate the passage of PE clock cycles.

Counting the number of clock cycles the PE uses to perform a certain function can be useful as a way of measuring program performance, or for profiling the PE.

While cycle counting is enabled, the trace unit outputs Cycle Count packets that contain processor clock cycle count values.

Cycle Count elements are associated with Commit elements, so that when a Commit element is generated, a Cycle Count element might also be generated. Whether a Cycle Count element is generated when a Commit element is generated depends on what cycle count threshold has been specified when programming TRCCCCTRL.THRESHOLD.

When a Commit element is generated and the cycle count value is equal to or more than the threshold value, then a Cycle Count element is generated and a Cycle Count packet is output. The cycle count value that is contained in that packet is associated with the Commit element that triggered it.

While cycle counting is enabled, and when a Commit element is generated and the cycle count value is greater than or equal to the threshold value that is programmed in TRCCCCTRL.THRESHOLD, the trace unit generates a Cycle Count element.

Also, because cycle counting is associated with Commit elements, a Cycle Count packet might imply the generation of Commit elements, and so in addition to the cycle count value, some Cycle Count packets also contain a value for the number of Commit elements that the trace unit has generated.

The value of cycle count that is given in a new Cycle Count packet indicates the number of processor clock cycles between the new Commit element that the packet is associated with, and the most recent Commit element prior to the new Commit element that had a Cycle Count element associated with it.

This means that if there is a requirement for a cumulative cycle count total, the cycle count values from the successive Cycle Count packets can be added together to obtain this.

Event trace

The ETE architecture supports the tracing of additional information in the trace stream. These are known as ETEEvents, also known as Event trace. The trace unit supports up to 4 ETEEvents. The generation of ETEEvents is controlled by selecting resources selectors. The occurrence of ETEEvents can be communicated in the following ways:

- To the system by External Outputs on page D4-4869.
- To the trace analyzer by Event Packet.

No overflow

A trace unit might include an optional feature to prevent overflows, which is indicated by TRCIDR3.NOOVERFLOW.

TRCSTALLCTRL.NOOVERFLOW controls the no overflow feature.
Enabling the no overflow feature might have a significant impact on PE performance.

While the no overflow feature is enabled, and while the number or frequency of ETEEvents is below an IMPLEMENTATION DEFINED threshold, the trace unit does not overflow.

The threshold is greater than or equal to one of each numbered ETEEvent, for each trace session.

When TRCIDR3.SYSSTALL is 0 the Effective value of TRCSTALLCTRL.NOOVERFLOW is 0 which means the no overflow feature is disabled.

When TRCSTALLCTRL.ISTALL is 0 and TRCSTALLCTRL.NOOVERFLOW is 1, it is CONSTRAINED UNPREDICTABLE whether any stalling is disabled or whether the no overflow feature is enabled.

**Stalling the execution of the PE**

The trace unit can be programmed to reduce the likelihood of a trace unit buffer overflow. If the trace unit is configured to support PE stalling, TRCIDR3.STALLCTL indicates that PE stalling is implemented and TRCIDR3.SYSSTALL indicates that PE stalling is permitted, then the execution of the PE can be slowed.

It is permissible that the operation of the PE can be affected by the programming of the trace unit. The amount of intrusion and when stalling occurs is IMPLEMENTATION DEFINED. Additional stalling of the PE execution can be achieved by enabling this feature.

Trace unit stalling of the PE is independent of the operation of the PE.

PE operations which explicitly interact with the trace unit complete independently of the programming of the ability of the trace unit to stall the PE.

The trace unit does not stall the PE while any of the following are true:

- The trace unit is in the Disabled state.
- The PE is executing in a Trace Prohibited region (see *Trace Prohibited Regions on page D4-4810*).
- The PE is in Debug State.
- The PE does not allow stalling, that is, TRCIDR3.SYSSTALL == 0b0.
- SelfHostedTraceEnabled() == FALSE and ExternalInvasiveDebugEnabled() == FALSE.
- When TRCSTALLCTRL.ISTALL == 0b0 and TRCSTALLCTRL.NOOVERFLOW == 0b0.
- Trace output is disabled.

When all of the following are true, the trace unit is permitted to stall the PE:

- Stalling of the PE is not prohibited by SCLVV.
- TRCSTALLCTRL.ISTALL == 0b1.
- Any of the following are true:
  - TRCSTALLCTRL.NOOVERFLOW == 0b1.
  - The available space in the internal storage of the trace unit is below the level indicated in TRCSTALLCTRL.LEVEL.

Otherwise, the trace unit does not stall the PE due to the stalling feature or no overflow feature.

The trace unit does not indefinitely stall the operation of the PE.

In a multi-threaded processor, if the trace unit stalls a PE, Arm recommends that stalling or disruption of the processing of other PEs is minimized. In particular, if tracing of one or more PEs in a multi-threaded processor is enabled but tracing of other PEs in the multi-threaded processor is disabled, Arm recommends that if the PEs being traced are stalled by their respective trace units then the stall has minimal effect on the PEs which are not being traced.

The levels indicated in TRCSTALLCTRL.LEVEL are the levels of intrusion allowed.
A summary of the stalling and no overflow scenarios is shown in Table D4-23 on page D4-4837, when TRCIDR3.STALLCTL is 1 and TRCIDR3.SYSSTALL is 1.

### Table D4-23 Summary of TRCSTALLCTRL stalling and no overflow features

<table>
<thead>
<tr>
<th>ISTALL</th>
<th>NOOVERFLOW</th>
<th>LEVEL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>x</td>
<td>Stalling is disabled.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>x</td>
<td>It is CONSTRAINED UNPREDICTABLE whether the no overflow feature is enabled or stalling is disabled.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Zero</td>
<td>Stalling is enabled at the minimum level.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Nonzero</td>
<td>Stalling is enabled and is based on the value in TRCSTALLCTRL.LEVEL.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Zero</td>
<td>The no overflow feature is enabled, preventing overflows.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Nonzero</td>
<td>The no overflow feature is enabled, preventing overflows, and TRCSTALLCTRL.LEVEL might cause stalling earlier than necessary to prevent overflows.</td>
</tr>
</tbody>
</table>

### Timestamping

The trace unit supports timestamping, where a common global time value is inserted into the trace stream. These timestamps may be used to correlate between multiple trace streams, including those from other PEs or other sources of trace. These timestamps may be used to determine the passage of time, for analyzing performance.

When timestamping is enabled, the trace unit inserts Timestamp elements into the trace stream. Each Timestamping element indicates the time that a recent P0 element or Event element occurred, and can be used to accurately determine when that element occurred.

The time value included in Timestamp elements is selected by TRFCR_EL1 and TRFCR_EL2 and is one of:

- Physical time, as seen by the generic timers in the PE.
- Virtual time, as seen by the generic timers in the PE.
- An IMPLEMENTATION DEFINED time value, often supplied by a CoreSight system.

The insertion of Timestamp elements is controlled by TRCCONFIGR.TS and TRCTSCTRL.

### Virtual context identifier tracing

Whether an implementation supports Virtual context identifier tracing is IMPLEMENTATION DEFINED. If it does, the trace unit can be programmed to output the identifier of a virtual machine that the PE is executing.

This option is enabled by setting TRCCONFIGR.VMID to 0b1.

If the PE implements EL2, the trace unit supports a 32-bit Virtual context identifier, with TRCIDR2.VMIDSIZE indicating a 32-bit Virtual context identifier size. The source of the Virtual context identifier is CONTEXTIDR_EL2.PROCID.

If the PE does not implement EL2, the trace unit does not support a Virtual context identifier, with TRCIDR2.VMIDSIZE indicating Virtual context identifier tracing is not supported.

Note: Previous trace architectures from Arm supported the ability to select the source of the Virtual context identifier. This specification does not support Virtual context identifier selection, and only permits CONTEXTIDR_EL2.PROCID as the source of the Virtual context identifier. See TRCIDR2.VMIDOPT for more details.
D4.5.11 Compression

Additional compression of the trace byte stream is achieved by the following methods:

- Removing elements that can be implied by the trace analyzer:
  - Implying the existence of Commit elements based on the tracing of other elements.
  - Removing Target Address elements that can be calculated by the trace analyzer by analysis of previous traced PE execution.
- Combining multiple elements together into a single packet:
  - Combining Atom elements into a single packet.
  - Combining Cancel elements and Mispredict elements into a single packet.

Implied commits

The ETE trace protocol provides mechanisms to minimize the number of Commit elements which need to be explicitly output in the trace byte stream. When a P0 element is output in the trace byte stream, if the number of speculative P0 elements output exceeds TRCIDR8.MAXSPEC, then a Commit element is implied which resolves the oldest speculative P0 element.

The trace unit does not generate commit packets for Commit elements that have been implied by the trace protocol.

Atom packing

The ETE trace protocol provides packets which allow groups of consecutive Atom elements to be packed into a single trace packet. Figure D4-15 on page D4-4839 shows the decision tree for generating the different formats of Atom packets.
Cancel Packets can indicate a number of Atom elements and the Cancel element.

The Mispredict Packets can indicate a number of Atom elements and the Mispredict element.

**Address compression**

The trace unit can remove program addresses from the trace stream. The trace analyzer can infer the addresses from the program image and previous history.

This includes the targets of direct P0 instructions, where the target address is encoded in the instruction itself.

The trace unit retains the Address information of up to the last three addresses that were explicitly output in the trace protocol, as contained in:

- Target Address packets.
• Source Address packets.
• Exception packets.
• Transaction Failure packets.
• PE Reset packets.
• Q packets.

The explicitly output addresses that the trace unit retains are known as the address history buffer.

For optimized trace protocol efficiency, Arm recommends that the address history buffer is three entries deep.

When any of the following packets are generated, the trace unit pushes the address value and sub_isa to the address history buffer:
• Target Address packet.
• Source Address packet.
• Q packet that implies a Target Address element.

When an Exception packet is generated, the trace unit pushes the preferred exception return address value and sub_isa to the address history buffer.

When one of the following packets is generated with an unknown address, the trace unit pushes an address value of 0x0 and sub_isa of IS0 to the address history buffer.
• Transaction Failure packet.
• PE Reset packet.

When a Target Address packet is generated, the trace unit uses the address history buffer to identify when a Target Address Exact Match packet can be used. When a Target Address Exact Match packet cannot be used, the most recent entry in the address history buffer is used for the Target Address packet selection.

When a Source Address packet is generated, the trace unit uses the address history buffer to identify when a Source Address Exact Match packet can be used. When a Source Address Exact Match packet cannot be used, the most recent entry in the address history buffer is used for the Source Address packet selection.

When an Exception packet is generated, the trace unit uses the address history buffer to identify when an Exception Exact Match Address packet can be used. When an Exception Exact Match Address packet cannot be used, the most recent entry in the address history buffer is used for the Exception Address packet selection.

When a Q packet is generated which implies a Target Address element, the trace unit uses the address history buffer to identify when a Q with Exact Match Address packet can be used. When a Q with Exact Match Address packet cannot be used, the most recent entry in the address history buffer is used for the Q Address packet selection.

When a Trace Info packet is generated, the trace unit sets all entries of the address history buffer to have an address value of 0x0 and sub_isa of IS0.

Return stack address matching

The trace unit might contain the optional return stack function. The return stack operates when Branch with Link instructions or indirect P0 instructions are taken, and provides a mechanism to allow the trace unit to remove certain Target Address elements from the trace element stream. The trace analyzer maintains an independent copy of the return stack which is used to determine when Target Address elements have been removed and then infer the target of indirect P0 instructions.

The depth of the return stack is IMPLEMENTATION DEFINED from 0 to 15 entries.

For optimized trace protocol efficiency, Arm recommends the trace unit implements the return stack with at least 3 entries.

While TRCCONFIG.RS is 1, when a Branch with Link instruction is predicted as taken and is traced, the trace unit pushes the following Address information to the return stack:
• The instruction address + the instruction size, that is, the return address for the Branch with Link instruction.
• The sub_isa from the instruction set encoding.

When a return stack push occurs, all existing entries are shifted down one place on the return stack and the new entry is pushed to the top entry of the return stack.

While the return stack is full, when a return stack push occurs, the oldest entry on the return stack is discarded.

When a Branch with Link instruction is predicted as taken and traced with an E Atom element, when a return stack push occurs, the trace unit pushes to the return stack, even if the prediction is incorrect and is subsequently corrected to an N Atom element.

When a Branch with Link instruction is predicted as not taken and traced with an N Atom element, the trace unit does not push to the return stack, even if the prediction is incorrect and is subsequently corrected to an E Atom element.

When a Branch with Link instruction is implied by a Q element, the trace unit does not push to the return stack.

When a Branch with Link instruction is executed in a branch broadcasting region, the trace unit does not push to the return stack.

When an indirect P0 instruction is taken and traced, and the Address information in the resultant Target Address element matches the address and sub_isa on the top of the return stack, the trace unit performs a return stack pop.

When a return stack pop occurs, both of the following occur:
• The trace unit discards the Target Address element that matches the address and sub_isa on the top of the return stack.
• The trace unit removes the top entry of the return stack, and shifts each older entry up one position.

When an indirect P0 instruction is implied by a Q element, the trace unit does not perform a return stack pop.

When an indirect P0 instruction is taken, it is possible that the target address is predicted incorrectly by the PE.

When the target address of a taken indirect P0 instruction is incorrectly predicted, and the incorrect target address is traced with a Target Address element, the trace unit corrects the incorrect address by generating a new Target Address element with the correct target address, and neither of the target addresses cause a return stack pop.

When the target address of a taken indirect P0 instruction is incorrectly predicted, and the incorrect target address matches the top entry of the return stack, the trace unit subsequently generates a Target Address element with the correct target address, and neither of the target addresses cause a return stack pop.

When the final status of the Atom element corresponding to an indirect P0 instruction is E, including when one or more Mispredict elements change the status of the Atom element, the trace unit performs a return stack pop.

Note: A return stack push only occurs if the initial Atom element state for the Branch with Link instruction is E. Conversely, a return stack pop only occurs if the final Atom element state for the indirect P0 instruction is E.

When an instruction that is both a Branch with Link instruction and an indirect P0 instruction is executed, the trace unit performs the following actions on the return stack, in order:
1. Determine whether a return stack push is possible and push if required.
2. Determine whether a return stack pop is possible and pop if required.

Note: Some Arm trace architectures use a different order of operations.

When any of the following occur, the trace unit discards the contents of the return stack:
• The trace unit generates a Trace Info element.
• The trace unit generates a Trace On element.
• The PE enters a branch broadcasting region.

A trace unit might discard the contents of the return stack at any time.

When the return stack contents are discarded, there is no requirement for the trace analyzer to be aware that this discard operation has occurred. This is because even though the contents of the trace unit return stack are discarded, there are no adverse consequences if the contents of the trace analyzer return stack are retained, but never used.
After a Trace Info element, a Target Address element and a Context element are required but might not be generated immediately. If the Target Address element and the Context element are not generated before the next P0 element, then any Branch with Link instructions must not push on to the return stack until both the Target Address element and the Context element have been generated.

Note: This restriction prevents the trace unit from performing return stack pushes for instructions that the trace analyzer cannot analyze, because it is not yet fully synchronized.

**Timestamp value compression**

The trace analyzer maintains a copy of the last Timestamp element value broadcast. The Timestamp element value might be compressed relative to the last value and only the bits that have changed need to be encoded.

When a Trace Info packet is generated, the trace unit sets its maintained value of the last Timestamp element to zero, and when the trace unit generates a subsequent Timestamp packet the value is compressed relative to this new zero value. This means that the first Timestamp packet after a Trace Info packet contains all non-zero bits of the Timestamp value.

Cancel Packets can indicate a number of Atom elements and the Cancel element.

The Mispredict Packets can indicate a number of Atom elements and the Mispredict element.
### D4.6 Resource operation

FEAT_ETE has a number of resources that can be used to provide advanced filtering functionality. The resources operate in one of the following states:

- **Running**
  - All the resources are active.

- **Pausing**
  - The resources are progressing to the Paused state.

- **Paused**
  - All the resources are static and inactive except for External Input Selectors.

---

As described in *System behaviors on page D4-4800*, the trace unit can be disabled by either:

- Setting TRCPRGCTRL.EN to 0.
- Locking the OS Lock, by setting OSLAR_EL1.OSLK to 1.

While the resources are in the Running state, and when any of the following occur, the resources enter the Pausing state:

- The trace unit becomes disabled.
- The trace unit enters the low-power state.
- The PE begins executing in a Trace Prohibited region.

While the resources are in the Pausing state, the resources enter the Paused state in finite time.

While the trace unit is in the Paused state, when all of the following are true, the resources enter the Running state:

- The trace unit is enabled.
- The trace unit is not in the low-power state.
- The PE is not executing in a Trace Prohibited region.

A trace unit buffer overflow has no impact on the behavior of the resources.

### D4.6.1 Behavior of the resources while in the Running state

The time taken for the resources to operate might vary between different trace unit implementations.
D4.6.2 Behavior of the resources while in the Pausing state

When the resources enter the Pausing state, the resources perform the following procedure:

1. All resources, except for the Sequencer and any Counters, are driven low as inputs to the Resource Selector logic. The Counters and the Sequencer behave as normal.
2. The states that the inputs were at before they were driven low are propagated through the Resource Selector logic.
3. The states of the Counters and the Sequencer are propagated through the Resource Selector logic one more time. That is, the states of the Counters and the Sequencer are propagated through the Resource Selector logic for the length of time that it takes for the state of a resource to be propagated through the Resource Selector logic.
4. The resources enter the Paused state.

The procedure that the resources perform when the resources are in the Pausing state has the result that, for resource events that are activated by a resource that is not a Counter or a Sequencer, no activity is lost, because all those resource events are updated.

When Counter and Sequencer states are propagated back as resources, so that a loop is created, then the following are true:

- If a Counter at zero resource is being used to activate the Sequencer or a Counter, then that Counter at zero resource might be propagating through the Resource Selector logic at the time when the procedure ends. In this case, the Sequencer state resource or other Counter at zero resource that is activated by that Counter at zero resource might be lost.
- If a Sequencer state resource is being used to activate a Counter, then that Sequencer state resource might be propagating through the Resource Selector logic at the time when the procedure ends. In this case, the Counter at zero resource that is activated by that Sequencer state resource might be lost.

When the trace unit becomes disabled, the behavior of the resources in the Pausing state ensures that the programmers' model provides a consistent view of the state of the trace unit resources. That is, regarding the Counters and the Sequencer, the following are true:

- If the state of the Sequencer propagated back as a resource, then the view of the Sequencer as a resource event and the view of the Sequencer resource state both show the same Sequencer state.
- If the state of a Counter is propagated back as a resource, then the view of the Counter as a resource event and the view of the Counter resource state both show the same Counter state.

The Counter state might be one of the following:

- The Counter is at zero.
- The Counter is not at zero.

D4.6.3 Behavior of the resources while in the Paused state

The behavior of the resources when the PE enters the low-power state or a Trace Prohibited region differs from other trace architectures defined by Arm.

While the resources are in the Paused state and the trace unit is not disabled, the resources do not lose resource events that are in transition, except those permitted when moving through the Pausing state of the resources. See Behavior of the resources while in the Pausing state for details on the resource events that are permitted to be lost when in the Pausing state.

While the resources are in the Paused state, the resources might not observe resource events that are in transition until after the resources leave the Paused state.

While the resources are in the Paused state, the resources remain in the state they are in.

While the resources are in the Paused state, the trace unit drives all External Outputs low.

When the trace unit becomes disabled and the resources enter the Paused state, and not before, the trace unit might set TRCSTATR.PMSTABLE to 1.
While TRCSTATR.PMSTABLE is set to 1, all resources and resource events remain in a quiescent state.

**D4.6.4 Behavior of resources on a Trace synchronization event**

When the following resources have finished calculations for all instructions prior to the previous Context synchronization event, a Trace synchronization event completes:
- Address Comparators.
- Context Identifier Comparators.
- Virtual Context Identifier Comparators.
- Single-shot Comparator Controls.

**D4.6.5 Resource organization**

There are 2 types of resources:
- Precise resources.
- Imprecise resources.

Each resource has a current state, which is output as a Resource state. The Resource state is selected by Resource Selectors, and then used by various trace unit functions as a Resource event, see Figure D4-17.

**Precise resources**

The precise resources are used in the evaluation of the ViewInst include/exclude function and the ViewInst start/stop function.

The trace unit evaluates the precise resources for each instruction block. See Instruction block on page D4-4809 for more details.

The trace unit maintains execution order of the precise resources.
Imprecise resources

If a Resource is not a precise resource, it is an imprecise resource.

![Diagram of Imprecise resource path]

Selecting a resource or a pair of resources

A resource is selected by using a Resource Selector.

Each Resource Selector uses one of the 30 TRCRSCTL<n> registers. The trace unit implements Resource Selectors in pairs, so that a maximum of 15 programmable pairs can be implemented.

Resource Selector 0 always provides a FALSE result.

While the resources are in the Running state, Resource Selector 1 provides a TRUE result.

TRCIDR4.NUMRSPAIR indicates how many pairs of Resource Selectors are implemented.

Resource Selectors can be used in pairs or used individually. When a pair of Resource Selectors is used, a Boolean function can be applied to the outputs of the combination of selected resources. See Figure D4-20 on page D4-4849.
While TRCRSCTLR<\(n\).SELECT[\(m\)] is 1, the Resource Selector selects the Resource Number \(m\) of the group selected by TRCRSCTLR<\(n\>.GROUP as described in Table D4-24 on page D4-4847.

### Table D4-24 Resource grouping

<table>
<thead>
<tr>
<th>Group</th>
<th>Resource Number</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>0-3</td>
<td>External Input Selectors 0-3</td>
</tr>
<tr>
<td></td>
<td>4-15</td>
<td>Reserved</td>
</tr>
<tr>
<td>0b0001</td>
<td>0-7</td>
<td>PE Comparator Inputs 0-7</td>
</tr>
<tr>
<td></td>
<td>8-15</td>
<td>Reserved</td>
</tr>
<tr>
<td>0b0010</td>
<td>0</td>
<td>Counter 0 at zero</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Counter 1 at zero</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Counter 2 at zero</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Counter 3 at zero</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Sequencer state 0</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Sequencer state 1</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Sequencer state 2</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Sequencer state 3</td>
</tr>
<tr>
<td></td>
<td>8-15</td>
<td>Reserved</td>
</tr>
<tr>
<td>0b0011</td>
<td>0-7</td>
<td>Single-shot Comparator Control 0-7</td>
</tr>
<tr>
<td></td>
<td>8-15</td>
<td>Reserved</td>
</tr>
<tr>
<td>0b0100</td>
<td>0-15</td>
<td>Single Address Comparator 0-15</td>
</tr>
<tr>
<td>0b0101</td>
<td>0-7</td>
<td>Address Range Comparator 0-7</td>
</tr>
<tr>
<td></td>
<td>8-15</td>
<td>Reserved</td>
</tr>
<tr>
<td>0b0110</td>
<td>0-7</td>
<td>Context Identifier Comparator 0-7</td>
</tr>
<tr>
<td></td>
<td>8-15</td>
<td>Reserved</td>
</tr>
<tr>
<td>0b0111</td>
<td>0-7</td>
<td>Virtual Context Identifier Comparator 0-7</td>
</tr>
<tr>
<td></td>
<td>8-15</td>
<td>Reserved</td>
</tr>
<tr>
<td>0b1xxx</td>
<td>0-15</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

While TRCRSCTLR<\(n\>.INV is set to 0 and one or more resources in a group are selected, when any of the outputs of the selected resources are high, the Resource Selector fires.

While TRCRSCTLR<\(n\>.INV is set to 1, when none of the outputs of the selected resources are high, the Resource Selector fires.

### A Resource Selector pair

The Resource Selectors are arranged in pairs, and the result of each of a pair of Resource Selectors can be combined using a Boolean function and used to drive other resources and events in the trace unit.

For each TRCRSCTLR<\(n\> register which is the lower register for a pair of Resource Selectors, the TRCRSCTLR<\(n\>.PAIRINV field.
For example:

- TRCRSCTRLR2 and TRCRSCTRLR3 constitute a Resource Selector pair. In this case:
  - TRCRSCTRLR2 is the lower register.
  - TRCRSCTRLR2.PAIRINV optionally inverts the result of the Boolean function that is applied to the outputs of the combination of selected resources.
  - TRCRSCTRLR3 is the upper register.
  - TRCRSCTRLR3.PAIRINV is RES0.

This means that, when a Resource Selector pair is used, the following scenario is possible:

- One TRCRSCTRL<n> might select only one resource within the group.
- The other TRCRSCTRL<n> might select more than one resource from the group, so that the result is a logical OR of the selected resources.
- A Boolean function, for example a logical AND, might be applied to the outputs of the combination of selected resources.
- The result of that Boolean function might be inverted by using PAIRINV.

In Figure D4-20 on page D4-4849, the Boolean function is selected by using the INV field for each Resource Selector, with the PAIRINV field for each Resource Selector pair, see Table D4-25 on page D4-4849.
D4.6.6   Address comparators

**ILGGVG**  
An ETE trace unit provides between 0 and 16 Single Address Comparators (SACs), each of which compares the instruction address with a user-programmed value.

**RYCRNP**  
The trace unit implements SACs in pairs, so that a trace unit implementation contains an even number of SACs.

**IMNTCY**  
TRCIDR4.NUMACPAIRS indicates how many pairs of SACs are implemented.

---

**Table D4-25 Selecting a Boolean function**

<table>
<thead>
<tr>
<th>Function</th>
<th>PAIRINV</th>
<th>Resource A INV</th>
<th>Resource B INV</th>
</tr>
</thead>
<tbody>
<tr>
<td>A AND B</td>
<td>0b0</td>
<td>0b0</td>
<td>0b0</td>
</tr>
<tr>
<td>NOT(A AND B)</td>
<td>0b1</td>
<td>0b0</td>
<td>0b0</td>
</tr>
<tr>
<td>Reserved</td>
<td>0b0</td>
<td>0b0</td>
<td>0b1</td>
</tr>
<tr>
<td>NOT(A) OR B</td>
<td>0b1</td>
<td>0b0</td>
<td>0b1</td>
</tr>
<tr>
<td>NOT(A) AND B</td>
<td>0b0</td>
<td>0b1</td>
<td>0b0</td>
</tr>
<tr>
<td>Reserved</td>
<td>0b1</td>
<td>0b1</td>
<td>0b0</td>
</tr>
<tr>
<td>NOT(A) AND NOT(B)</td>
<td>0b0</td>
<td>0b1</td>
<td>0b1</td>
</tr>
<tr>
<td>A OR B</td>
<td>0b1</td>
<td>0b1</td>
<td>0b1</td>
</tr>
</tbody>
</table>
When the PE executes instructions in Debug state, Address Comparators do not match.

When the PE executes instructions in a Trace Prohibited region, Address Comparators do not match.

Address Comparators might match in failed transactions.

Address Comparators might match on Speculative execution.

**Single Address Comparators**

A SAC can be used in the following ways:

- As inputs to the ViewInst start/stop function in the ViewInst function. For more information, see *ViewInst start/stop function filtering* on page D4-4815.
- As an individual resource.
- The comparator can be programmed so that, whenever the PE is in Non-secure state, the comparator only matches in certain Exception levels.
- The comparator can be programmed so that, whenever the PE is in Secure state, the comparator only matches in certain Exception levels.

An SAC only matches on Exception levels and Security states as indicated by TRCACATR<n>.

An SAC only matches on the context indicated by TRCACATR<n>.CONTEXT and TRCACATR<n>.CONTEXTTYPE.

\[
\text{SAC}_{\text{EL}}[n] = \begin{cases} 
\neg \text{TRCACATR}_n.\text{EXLEVEL}_\text{S}.\text{EL}0 & \text{Secure EL0} \\
\neg \text{TRCACATR}_n.\text{EXLEVEL}_\text{S}.\text{EL}1 & \text{Secure EL1} \\
\neg \text{TRCACATR}_n.\text{EXLEVEL}_\text{S}.\text{EL}2 & \text{Secure EL2} \\
\neg \text{TRCACATR}_n.\text{EXLEVEL}_\text{S}.\text{EL}3 & \text{EL3} \\
\neg \text{TRCACATR}_n.\text{EXLEVEL}_\text{NS}.\text{EL}0 & \text{Non-secure EL0} \\
\neg \text{TRCACATR}_n.\text{EXLEVEL}_\text{NS}.\text{EL}1 & \text{Non-secure EL1} \\
\neg \text{TRCACATR}_n.\text{EXLEVEL}_\text{NS}.\text{EL}2 & \text{Non-secure EL2} 
\end{cases}
\]  

\( (1) \)

\[
m = \text{TRCACATR}_n.\text{CONTEXT} \quad (1) \\
type = \text{TRCACATR}_n.\text{CONTEXTTYPE} \quad (2)
\]

\[
\text{SAC}_{\text{CONTEXT}}[n] = \begin{cases} 
1 & \text{type is 0} \\
\text{CIDCOMP}[m] & \text{type is 1} \\
\text{VMIDCOMP}[m] & \text{type is 2} \\
\text{CIDCOMP}[m] \land \text{VMIDCOMP}[m] & \text{type is 3} 
\end{cases}
\]  

\( (1) \)
When an instruction is executed, and the address of the lowest byte of the instruction exactly matches the programmed address of an SAC, the SAC matches.

\[
SAC\_ADDR_i[n] = (\text{ThisInstrAddr}(i) \equiv \text{TRCACVRn.ADDRESS}) \quad (1)
\]

For example, for a 4-byte instruction at address 0x1000:
- The lowest byte of the instruction is at 0x1000.
- The second byte of the instruction is at 0x1001.
- The third byte of the instruction is at 0x1002.
- The highest byte of the instruction is at 0x1003.

If an SAC is programmed with 0x1000, then it always matches on that instruction at address 0x1000.

It is IMPLEMENTATION DEFINED whether an SAC matches when its programmed address matches any byte of an instruction which is not the lowest byte of the instruction.

The Arm architecture supports disabling IT instructions on more than one subsequent instruction, using the ITD bits in the SCTLR, HSCTLR, and SCTLR_EL1 System registers. If any of the ITD bits are set to 1 and are affecting IT operation, and a SAC is programmed to match on the address of the instruction that is immediately after an IT instruction, when the instruction immediately after the IT instruction is executed it is CONSTRAINED UNPREDICTABLE whether that comparator matches.

If any of the ITD bits are set to 1, Arm recommends that a SAC is programmed to match on the address of the IT instruction, instead of the instruction immediately after the IT instruction.

To avoid unexpected behavior from an SAC, Arm recommends that the SAC is always programmed with an address that is for the lowest byte of an instruction.

When the instruction immediately after a MOVPRFX instruction executes, if a SAC is programmed to match on the address of this instruction, then it is CONSTRAINED UNPREDICTABLE whether that comparator matches.

Arm recommends that a SAC is programmed to match on the address of the MOVPRFX instruction, instead of the instruction immediately after the MOVPRFX instruction.

The operation of a SAC is as follows:

\[
SAC_i[n] = \begin{cases} 
0 & \text{When Prohibited} \\
0 & \text{When in Debug State} \\
\text{SAC\_ADDR}_i[n] \land \text{SAC\_EL}_i[n] \land \text{SAC\_CONTEXT}_i[n] & \text{Otherwise}
\end{cases}
\]

**Address Range Comparators**

Pairs of SACs are arranged to form one Address Range Comparator (ARC). An ARC is programmed with an address range, so that whenever any address in that range is accessed, the ARC matches. A trace unit contains between zero and eight ARCs. ARCs can be used in the following ways:
- Selected for the ViewInst include/exclude function in the ViewInst function. See ViewInst include/exclude function filtering on page D4-4818.
- As individual resources.

An ARC is programmed by programming the SACs as follows:
- The first SAC is programmed with the start address of the instruction range.
- The second SAC is programmed with the end address of the instruction range.

The address that the second SAC is programmed with must be greater than or equal to the address that the first SAC is programmed with, that is, the end address must be greater than or equal to the start address.
While the start address of an ARC is greater than the end address, the behavior of the ARC is CONSTRAINED UNPREDICTABLE, that is, at any time the ARC might do one of the following:

- Match.
- Not match.

While the TRCACATR\(<n>\) registers for the SACs in an ARC are programmed to different values, the behavior of the ARC is CONSTRAINED UNPREDICTABLE.

While an ARC is programmed with an instruction address range, when the PE executes an instruction at an address in the following range, the ARC matches:

\[
\text{start address} = \text{TRCACVR}_n.\text{ADDRESS} \quad (1) \\
\text{end address} = \text{TRCACVR}(n+1).\text{ADDRESS} \quad (2)
\]

\[
\text{ARC}_\text{ADDR}_{\lfloor n/2 \rfloor} = (\text{ThisInstrAddr}_i \geq \text{start address}) \land (\text{ThisInstrAddr}_i \leq \text{end address})
\]

When an instruction is executed, and the address of the lowest byte of the instruction is within the programmed address range of an ARC, the ARC matches.

When an instruction is executed and the programmed address range of an ARC contains addresses for one or more bytes of the instruction, but does not contain the address for the lowest byte of the instruction, it is IMPLEMENTATION SPECIFIC whether the ARC matches.

For example, for a 4-byte instruction at address 0x1000:

- The lowest byte of the instruction is at 0x1000.
- The second byte of the instruction is at 0x1001.
- The third byte of the instruction is at 0x1002.
- The highest byte of the instruction is at 0x1003.

If the programmed address range contains 0x1000, then the ARC always matches. However, if the programmed address range starts at either 0x1001, 0x1002, or 0x1003, then it is IMPLEMENTATION SPECIFIC whether the ARC matches.

To avoid unexpected behavior from an ARC, Arm recommends that the ARC is always programmed with an address range that starts with an address for the lowest byte of an instruction.

The Arm architecture supports disabling IT instructions on more than one subsequent instruction, using the ITD bits in the SCTLR, HSCTLR, and SCTLR_EL1 System registers. If any of the ITD bits are set to 1 and are affecting IT operation, and an ARC is programmed to include the address of the instruction that is immediately after an IT instruction but not include the IT instruction, when the instruction immediately after the IT instruction is executed then it is CONSTRAINED UNPREDICTABLE whether that comparator matches.

If any of the ITD bits are set to 1, Arm recommends that an ARC is programmed to include both the IT instruction and the instruction immediately after the IT instruction.

When the instruction immediately after a MOVPRFX instruction is executed, if an ARC is programmed to include the address of the instruction that is after the MOVPRFX instruction but not the MOVPRFX instruction, then it is CONSTRAINED UNPREDICTABLE whether that comparator matches.

Arm recommends that an ARC is programmed to include both the MOVPRFX instruction and the instruction immediately after the MOVPRFX instruction.
It might be possible for multiple matches to occur simultaneously. The definition of when matches occur simultaneously is implementation specific, and might vary because of runtime conditions. However, an example of when multiple matches might occur simultaneously is when multiple instructions are observed in the same processor clock cycle, so that multiple comparisons take place with each address in the programmed range. In this case, any of the following might occur:

- An address in the range is matched more than once.
- More than one address in the range is matched simultaneously.

When multiple ARC matches occur simultaneously for one ARC, both of the following are true:
- The ARC signals a match at least once.
- The ARC does not signal more matches than the number of instructions that are executed with an address that matches an address in the programmed range.

Each ARC can be used with one, or a combination of, the following:
- A Context Identifier Comparator.
- A Virtual Context Identifier Comparator.

An ARC only matches on Exception levels and Security states as indicated by TRCACATR<2n>.

\[
ARC_{EL_{i}}[n] = \begin{cases}
-TRCACATR<2n>.EXLEVEL_S_EL0 & \text{Secure EL0} \\
-TRCACATR<2n>.EXLEVEL_S_EL1 & \text{Secure EL1} \\
-TRCACATR<2n>.EXLEVEL_S_EL2 & \text{Secure EL2} \\
-TRCACATR<2n>.EXLEVEL_S_EL3 & \text{EL3} \\
-TRCACATR<2n>.EXLEVEL_NS_EL0 & \text{Non-Secure EL0} \\
-TRCACATR<2n>.EXLEVEL_NS_EL1 & \text{Non-Secure EL1} \\
-TRCACATR<2n>.EXLEVEL_NS_EL2 & \text{Non-Secure EL2}
\end{cases}
\]

An ARC only matches on the context indicated by TRCACATR<2n>.CONTEXT and TRCACATR<2n>.CONTEXTTYPE.

\[
m = TRCACATR<2n>.CONTEXT \\
type = TRCACATR<2n>.CONTEXTTYPE
\]

\[
ARC_{CONTEXT_{i}}[n] = \begin{cases}
1 & \text{type is 0} \\
CIDCOMP[m] & \text{type is 1} \\
VMIDCOMP[m] & \text{type is 2} \\
CIDCOMP[m]\land VMIDCOMP[m] & \text{type is 3}
\end{cases}
\]

The operation of an ARC is as follows:

\[
ARC_{i}[n] = \begin{cases}
0 & \text{When Prohibited} \\
0 & \text{When in Debug State} \\
ARC_{ADDR_{i}}[n] \land ARC_{EL_{i}}[n] \land ARC_{CONTEXT_{i}}[n] & \text{Otherwise}
\end{cases}
\]
D4.6.7 Context Identifier Comparator

An ETE trace unit provides between zero and eight Context Identifier Comparators. Each Context Identifier Comparator can be used in any of the following ways:

- Associated with a SAC.
- Associated with an ARC.
- As an individual resource.

While a Context Identifier Comparator is associated with either an SAC or an ARC, only while the PE is executing with the Context identifier that the Context Identifier Comparator is programmed with and when an address is accessed which the SAC or ARC is programmed to match on, the SAC or ARC signals a match.

While a Context Identifier Comparator is used as an individual resource, when an instruction block is executed with the Context identifier that the Context Identifier Comparator is programmed with, the Context Identifier Comparator matches.

When using a Context Identifier Comparator as an independent resource to activate a resource event, the time that the resource event is activated relative to the time that the Context Identifier Comparator becomes active might be imprecise.

It might be possible for multiple matches of a Context Identifier Comparator to occur simultaneously. The definition of when matches occur simultaneously is IMPLEMENTATION SPECIFIC, and might vary because of runtime conditions. However, an example of when multiple matches might occur simultaneously is when multiple instructions are observed in the same processor clock cycle, so that multiple comparisons take place.

When multiple Context Identifier Comparator matches occur simultaneously for one Context Identifier Comparator, all of the following are true:

- The Context Identifier Comparator signals a match at least once.
- The Context Identifier Comparator does not signal more matches than the number of instructions that are executed with the Context identifier that the Context Identifier Comparator is programmed with.

A Context Identifier Comparator might match on Speculative execution, that is, a Context Identifier Comparator might match if the PE speculatively changes the Context identifier.

When the PE executes instructions in Debug state, Context Identifier Comparators do not match.

When the PE executes instructions in a Trace Prohibited region, Context Identifier Comparators do not match.
The Context identifier might change at points that are not Context synchronization events, for example when a System instruction is used to write to the current Context identifier register. In these scenarios, the Context Identifier Comparator might compare against the old or new Context identifier value for any instruction after the P0 element before the System instruction, up to the Context synchronization event after the System instruction.

\[
m = \begin{cases} 
\text{TRCCIDCCTL0.COMP0} & n = 0 \\
\text{TRCCIDCCTL0.COMP1} & n = 1 \\
\text{TRCCIDCCTL0.COMP2} & n = 2 \\
\text{TRCCIDCCTL0.COMP3} & n = 3 \\
\text{TRCCIDCCTL1.COMP4} & n = 4 \\
\text{TRCCIDCCTL1.COMP5} & n = 5 \\
\text{TRCCIDCCTL1.COMP6} & n = 6 \\
\text{TRCCIDCCTL1.COMP7} & n = 7 
\end{cases}
\]

\[
v = \text{TRCCIDCVRn.VALUE} \\
cid = \text{CONTEXTIDR_EL1.PROCID}
\]

\[
\text{CIDCOMP}[n] = \begin{cases} 
0 & \text{When Prohibited} \\
0 & \text{When in Debug State} \\
7 & \text{Otherwise} \\
\prod_{j=0}^{\lfloor \log_2 m \rfloor - 1} \left(v[8j + 7 : 8j] \equiv(cid[8j + 7 : 8j]) \lor m[j] \right) 
\end{cases}
\]

### D4.6.8 Virtual Context Identifier Comparators

An ETE trace unit provides between zero and eight Virtual Context Identifier Comparators. Each Virtual Context Identifier Comparator can be used in any of the following ways:

- Associated with a SAC.
- Associated with an ARC.
- As an individual resource.

While a Virtual Context Identifier Comparator is associated with either an SAC or an ARC, only while the PE is executing with the Virtual context identifier that the Virtual Context Identifier Comparator is programmed with and when an address is accessed which the SAC or ARC is programmed to match on, the SAC or ARC signals a match.

While a Virtual Context Identifier Comparator is used as an individual resource, when an instruction block is executed with the Virtual context identifier that matches the Virtual Context Identifier Comparator value, the Virtual Context Identifier Comparator matches.

While TRFCR_EL2.CX indicates that Virtual Context Identifier Comparators cannot match, Virtual Context Identifier Comparators do not match.

When the PE executes instructions in Debug state, Virtual Context Identifier Comparators do not match.

When the PE executes instructions in a Trace Prohibited region, Virtual Context Identifier Comparators do not match.

When using a Virtual Context Identifier Comparator as an independent resource to activate a resource event, the time at which the resource event is activated relative to the time at which the Virtual Context Identifier Comparator becomes active might be imprecise.

A Virtual Context Identifier Comparator is associated with an SAC by programming TRCACATR<\alpha>.CONTEXT for the SAC.
It might be possible for multiple matches of a Virtual context identifier to occur simultaneously. The definition of when matches occur simultaneously is IMPLEMENTATION SPECIFIC, and might vary because of runtime conditions. However, an example of when multiple matches might occur simultaneously is when multiple instructions are observed in the same processor clock cycle, so that multiple comparisons take place.

When multiple Virtual Context Identifier Comparator matches occur simultaneously for one Virtual Context Identifier Comparator, both of the following are true:

• The Virtual Context Identifier Comparator signals a match at least once.
• The Virtual Context Identifier Comparator does not signal more matches than the number of instructions that are executed with the Virtual context identifier that the Virtual Context Identifier Comparator is programmed with.

A Virtual Context Identifier Comparator might signal a match on Speculative execution, that is, a Virtual Context Identifier Comparator might signal a match when the PE speculatively changes the Virtual context identifier.

The Virtual context identifier might change at points which are not Context synchronization events, for example when a System instruction is used to write to CONTEXTIDR_EL2. In these scenarios, the Virtual Context Identifier Comparator might compare against the old or new Virtual context identifier value for any instruction after the P0 element before the System instruction, up to the Context synchronization event after the System instruction.

The Counters that are employed by the ETE architecture are all decrement counters. The ETE architecture enables a trace unit to connect Counter outputs to resource events, so that a Counter at zero state can be used as a resource to activate a resource event. For example, a Counter at zero state might be used to assert an External Output or to make ViewInst active.

An ETE trace unit provides up to four 16-bit Counters. TRCIDR5.NUMCNTR indicates how many Counters are implemented. For each Counter, the following can be specified:

• The initial counter value. This can be programmed using TRCCNTVR<n>.
• The reload value. This can be programmed using TRCCNRDLVR<n>.
• The resource event that causes the Counter to reload with the reload value. This resource event is called RLDEVENT. In addition, if required, the Counter can be programmed so that it automatically reloads whenever it reaches zero.
• The resource event that enables the Counter to decrement. This resource event is called CNTEVENT. The Counter decrements whenever CNTEVENT is active.

The processor clock clocks the Counters in the trace unit.
While the PE is stalled, the Counters continue to count.

While the resources are in the Paused state, the Counters do not count.

When a Counter value is changed by anything other than a direct write to TRCCNTVR<n>, the trace unit considers the change to be an indirect write to TRCCNTVR<n>.VALUE.

Each Counter operates in one of the two following possible modes:
- Normal mode.
- Self-reload mode.

While the Counter is in Normal mode, when the Counter reaches zero, the Counter remains at zero until the reload resource event, RLDEVENT, occurs.

While the Counter is in Normal mode, the Counter-at-zero resource is active for the whole of the time that the Counter is at zero.

While the Counter is in Self-reload mode, when the Counter reaches zero, when the decrement resource event is next active, the trace unit reloads the Counter with the reload value.

While the Counter is in Self-reload mode, when the Counter value is zero, the decrement resource event is active and the reload resource event is not active, the Counter-at-zero resource is active for one cycle.

The following examples show various operating scenarios for a single Counter. Each Counter is programmed with a reload value of 0x3.

---

**Figure D4-21 Counter Example 1, Normal mode**
Figure D4-22 Counter Example 2, Normal mode

Figure D4-23 Counter Example 3, Self-reload mode

Figure D4-24 Counter Example 4, Self-reload mode
While the decrement resource event is inactive, the Counters do not decrement.

The trace unit prioritizes the reload resource event over the count decrement resource event.

Counter Operation in Normal mode

This table describes the counter operation in Normal mode:

<table>
<thead>
<tr>
<th>RLDEVENT</th>
<th>dec_action</th>
<th>Counter value</th>
<th>Action</th>
<th>Resource Active</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inactive</td>
<td>x</td>
<td>0</td>
<td>Stable</td>
<td>Yes</td>
<td>Resource is active while Counter is at zero and remains at zero</td>
</tr>
<tr>
<td>Inactive</td>
<td>0</td>
<td>&gt; 0</td>
<td>Stable</td>
<td>No</td>
<td>No activity</td>
</tr>
<tr>
<td>Inactive</td>
<td>1</td>
<td>&gt; 0</td>
<td>Decrement</td>
<td>No</td>
<td>Decrement when not zero</td>
</tr>
<tr>
<td>Active</td>
<td>x</td>
<td>0</td>
<td>Reload</td>
<td>Yes</td>
<td>Reload, but resource is active because Counter is at zero</td>
</tr>
<tr>
<td>Active</td>
<td>x</td>
<td>&gt; 0</td>
<td>Reload</td>
<td>No</td>
<td>Reload</td>
</tr>
</tbody>
</table>

Counter Operation in Self-reload mode

This table describes the counter operation in Self-reload mode:

<table>
<thead>
<tr>
<th>RLDEVENT</th>
<th>dec_action</th>
<th>Counter value</th>
<th>Action</th>
<th>Resource Active</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inactive</td>
<td>0</td>
<td>x</td>
<td>Stable</td>
<td>No</td>
<td>No activity, resource is not active even if the Counter is at zero</td>
</tr>
<tr>
<td>Inactive</td>
<td>1</td>
<td>0</td>
<td>Reload</td>
<td>Yes</td>
<td>Reload because dec_action is active and the Counter is at zero, resource is active only in this cycle</td>
</tr>
<tr>
<td>Inactive</td>
<td>1</td>
<td>&gt; 0</td>
<td>Decrement</td>
<td>No</td>
<td>Decrement when not zero</td>
</tr>
<tr>
<td>Active</td>
<td>x</td>
<td>x</td>
<td>Reload</td>
<td>No</td>
<td>Reload regardless of decrement action and the value of the Counter, resource is never active</td>
</tr>
</tbody>
</table>
Forming a larger Counter from two separate Counters

Some Counters can be chained together to form a larger counter, so that every time one Counter reloads, another Counter decrements.

The following example shows an operating scenario for 2 Counters chained together. Counter 0 is programmed with a reload value of \(0x2\).

![Figure D4-26 Chained Counter Example 1](image)

- **CLK**
  - Counter 0 Value: \(0x2, 0x1, 0x0, 0x2, 0x1, 0x0, 0x2, 0x1, 0x0, 0x2\)
  - Counter 1 Value: \(0x3, 0x2, 0x1, 0x0\)

- **Events**
  - Counter 0 RLDEVENT
  - Counter 0 CNTEVENT
  - Counter 1 RLDEVENT
  - Counter 1 CNTEVENT

- **Resources**
  - Counter 0 at zero
  - Counter 1 at zero

**Figure D4-26 Chained Counter Example 1**

Only certain Counters can be programmed to be chained together, as follows:
- Counter 1 can be programmed to decrement when Counter 0 reloads.
- Counter 3 can be programmed to decrement when Counter 2 reloads.

The decrement resource event for the higher Counter \(n\) is active when either of the following occurs:
- The lower Counter reloads due to one of the following:
  - The reload resource event that is selected by TRCCNTCTRL\(<n-1>.RLDEVENT\).
  - The Self-reload mechanism that is controlled by TRCCNTCTRL\(<n-1>.RLDSELF\).
- The decrement resource event that is selected by TRCCNTCTRL\(<n>.CNTEVENT\) is active.

While two Counters are chained together to form a larger counter, the larger counter appears as a 32-bit counter without any tearing of the values between the two Counters.

For example, if Counter 0 is in Self-reload mode and has a value of \(0x0000\) and a reload value of \(0xFFFF\), and Counter 1 is in Normal mode and has a value of \(0x1234\), then when a decrement resource event occurs on Counter 0, Counter 0 reloads to \(0xFFFF\). The reload of Counter 0 causes Counter 1 to decrement, resulting in a value of \(0x1233\).

Therefore the sequence on the Counters on consecutive cycles is \(0x12340000\) and \(0x1233FFFF\).

Note: The CounterAtZero resource might not be asserted at the same time that the Counter is at zero. For example, this could happen if the trace unit implementation pipelines some logic.
Pseudocode

EvalAllCounters()

// The counter-at-zero resources
array boolean CounterAtZero[0..3];

// EvalAllCounters() is called each clock cycle
EvalAllCounters()
    array boolean reload[0..3];
    reload[0] = EvalCounter(0, FALSE);
    reload[1] = EvalCounter(1, reload[0]);
    reload[2] = EvalCounter(2, FALSE);
    reload[3] = EvalCounter(3, reload[2]);

// EvalCounter() is called for each counter
EvalCounter(integer index, boolean lower_reloads)
    boolean dec_action;
    boolean resource_active;
    bits(16) next_value;
    boolean reload;
    boolean decrement;

    // A dec_action signal is constructed which indicates whether the counter
    // decrements. This is based on TRCCNTCTRL[n].CNTEVENT and, for counters
    // which support chaining, on TRCCNTCTRL[n].CNTCHAIN and on whether or not
    // the lower counter is reloading.
    dec_action = IsEventActive(TRCCNTCTRL[index].CNTEVENT) ||
        (TRCCNTCTRL[index].CNTCHAIN && lower_reloads);

    // The counter-at-zero resource is active if the counter is
    // currently at zero and is either in Normal mode or in
    // Self-Reload mode and dec_action is active and the reload
    // event is not active.
    resource_active = (TRCCNTVR[index] == 0) &&
        (!TRCCNTCTRL[index].RLDSELF ||
        (dec_action && !IsEventActive(TRCCNTCTRL[index].RLDEVENT))
    );

    // The counter reloads if the reload event is active or the self-reload
    // mechanism causes a reload.
    reload = IsEventActive(TRCCNTCTRL[index].RLDEVENT) ||
        (TRCCNTCTRL[index].RLDSELF && dec_action && TRCCNTVR[index] == 0);

    // The counter only decrements if it is non-zero and does not reload and
    // dec_action is active.
    decrement = !reload && (TRCCNTVR[index] != 0) && dec_action;

    // Determine the next value of the counter
    if reload then
        TRCCNTVR[index] = TRCCNTRLDVR[index].VALUE;
    else if decrement then
        TRCCNTVR[index] = TRCCNTVR[index] - 1;
    else
        TRCCNTVR[index] = TRCCNTVR[index];
    CounterAtZero[index] = resource_active;
    return reload;

D4.6.10   Sequencer

An ETE trace unit can contain a Sequencer state machine that has four states.
TRCIDR5.NUMSEQSTATE indicates whether the state machine is implemented.

If the Sequencer state machine is implemented, it has four states, numbered 0 to 3.

When the trace unit leaves the Disabled state, the Sequencer state machine starts in the state programmed in TRCSEQSTR.STATE.

The Sequencer can be connected to resource events, so that the Sequencer transitions from one state to another when certain resource events occur. The TRCSEQEVR<n> registers can be used to specify which resource events cause the state machine to transition.

Each register can be used to specify the following:

- A resource event that causes the state machine to progress to the next state.
- A resource event that causes the state machine to transition backwards to the previous state.

Different resource events can be chosen to cause the Sequencer to transition between different states. For example, a particular resource event might cause an F0 transition from state 0 to state 1 on one processor clock cycle, whereas a different resource event might cause an F1 transition from state 1 to state 2 on the next processor clock cycle. A third independent resource event might cause a B1 transition backwards from state 2 to state 1 on the third clock cycle.

The trace unit prioritizes forward transitions over backward transitions in the Sequencer state machine. That is, when two resource events occur that result in a forward transition conflicting with a backward transition in the same processor clock cycle, the trace unit gives priority to the forward transition and ignores the backward transition.

The Sequencer can progress through multiple states in a single processor clock cycle. For example, if the Sequencer is in state 0 and the resource events that cause an F0 and F1 transition to take place both become active in one clock cycle, then the Sequencer progresses from state 0 to state 2.

The Sequencer can be reset to state 0 from any other state. TRCSEQRSTEVR can be used to specify a resource event to reset the Sequencer.

When a resource event that causes an RST transition occurs, the Sequencer finishes the clock cycle in state 0 and does not progress to another state until the next clock cycle.

The trace unit prioritizes RST transitions over all other transitions. That is, when a resource event that causes an RST transition is active in the same clock cycle as resource events that cause other transitions, the trace unit gives priority to the RST transition and ignores all other transitions.

The following table defines all of the possible state transitions.

<table>
<thead>
<tr>
<th>From</th>
<th>To 0</th>
<th>To 1</th>
<th>To 2</th>
<th>To 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RST</td>
<td>!F0</td>
<td>F0 &amp; !F1</td>
<td>F0 &amp; F1 &amp; !F2</td>
</tr>
<tr>
<td>1</td>
<td>RST</td>
<td>(B0 &amp; !F1 &amp; !F0)</td>
<td>(!B0</td>
<td>F0) &amp; !F1</td>
</tr>
<tr>
<td>2</td>
<td>RST</td>
<td>(B0 &amp; B1 &amp; !F2 &amp; !F1 &amp; !F0)</td>
<td>B1 &amp; (!B0</td>
<td>F0) &amp; !F1 &amp; !F2</td>
</tr>
</tbody>
</table>
If multiple resource events that cause transitions become active in one processor clock cycle, there is no guarantee that the order of these resource events becoming active is observed. For example, you might program:

- F0 to be active on an instruction Address Comparator at address 0x1000.
- F1 to be active on an instruction Address Comparator at address 0x1004.

If the instruction at 0x1000 and the instruction at 0x1004 are executed in the same processor clock cycle, then the transition from state 0 to state 2 occurs regardless of the program order of the two instructions.

When the Sequencer state is changed by anything other than a direct write to TRCSEQSTR, the trace unit considers the change to be an indirect write to TRCSEQSTR.STATE.

The ETE architecture provides each Sequencer state as a resource, so that states can be used to trigger other resource events in the trace unit.

When the Sequencer progresses through multiple states in a single processor clock cycle, for each state that it passes through, the resource state that the Sequencer triggers is active for that cycle.

For example, if the Sequencer is in state 0, and in one processor clock cycle it moves to state 3, then the resource events that state 1 and state 2 are connected to must be active for that clock cycle. The same rule applies if the Sequencer is transitioning backwards, so that if it is in state 3, and in one processor clock cycle B2 and B1 cause it move to state 1, then the resource event that state 2 is connected to must be active for that clock cycle.

The exception to this is when an RST transition causes the Sequencer to return to state 0. For example, if the Sequencer is in state 3, and in one processor clock cycle it moves to state 0, then the resource events that state 2 and stage 1 are connected to must not become active.
Pseudocode

_EvalSequencer()

// The sequencer state resources
array boolean SequencerState[0..3];

// EvalSequencer()
// ===============
EvalSequencer()
    (rst, txn0, txn1, txn2, txn3) = SequencerTransitions();
    // Sequencer State resources
    SequencerState[0] = FALSE;
    SequencerState[1] = FALSE;
    SequencerResource(rst, txn0, txn1, txn2, txn3);
    SequencerNextState(rst, txn0, txn1, txn2, txn3);
SequencerTransitions()

// SequencerTransitions()
// ================

( boolean rst,
  array boolean txn0[0..3],
  array boolean txn1[0..3],
  array boolean txn2[0..3],
  array boolean txn3[0..3]) SequencerTransitions()
  boolean F0 = IsEventActive(TRCSEQEVR0.F);
  boolean B0 = IsEventActive(TRCSEQEVR0.B);
  boolean F1 = IsEventActive(TRCSEQEVR1.F);
  boolean B1 = IsEventActive(TRCSEQEVR1.B);
  boolean F2 = IsEventActive(TRCSEQEVR2.F);
  boolean B2 = IsEventActive(TRCSEQEVR2.B);
  boolean rst = IsEventActive(TRCSEQRSTEVR);

  array boolean txn0[0..3];
  array boolean txn1[0..3];
  array boolean txn2[0..3];
  array boolean txn3[0..3];

  txn0[1] = F0 && !F1;
  txn0[2] = F0 &&  F1 && !F2;
  txn0[3] = F0 &&  F1 &&  F2;

  txn1[0] = B0 && !F0  && !F1;
  txn1[1] = !(B0 ||  F0) && !F1;
  txn1[2] =                  F1 && !F2;
  txn1[3] =                  F1 &&  F2;

  txn2[0] = B0 && !F0  &&  B1 && !F1  && !F2;
  txn2[1] = !(B0 ||  F0) &&  B1 && !F1  && !F2;
  txn2[2] =                (!(B1 ||  F1) && !F2;
  txn2[3] =                                 F2;

  txn3[0] = B0 && !F0  &&  B1 && !F1  && B2 && !F2;
  txn3[1] = !(B0 ||  F0) &&  B1 && !F1  && B2 && !F2;
  txn3[3] =                              (!(B2 ||  F2);

return (rst, txn0, txn1, txn2, txn3);

SequencerResource()

// SequencerResource()
// ================

SequencerResource(boolean rst,
  array boolean txn0[0..3],
  array boolean txn1[0..3],
  array boolean txn2[0..3],
  array boolean txn3[0..3])

case TRCSEQSTR.STATE of
  0 then SequencerState[0] = TRUE;
  1 then SequencerState[1] = TRUE;
  2 then SequencerState[2] = TRUE;
  3 then SequencerState[3] = TRUE;

  // If the statemachine passes through
  // several states in one iteration then
  // set the SequencerState as appropriate.
  if !rst then
case TRCSEQSTR.STATE of
  0 then
    if txn0[2] then
      SequencerState[1] = TRUE;
    if txn0[3] then
      SequencerState[1] = TRUE;
      SequencerState[2] = TRUE;
  1 then
    if txn1[3] then
      SequencerState[1] = TRUE;
      SequencerState[2] = TRUE;
  2 then
    if txn2[0] then
      SequencerState[1] = TRUE;
  3 then
    if txn3[0] then
      SequencerState[1] = TRUE;
    if txn3[1] then
      SequencerState[2] = TRUE;

SequencerNextState()

// SequencerNextState()
// ============

SequencerNextState(boolean rst,
  array boolean txn0[0..3],
  array boolean txn1[0..3],
  array boolean txn2[0..3],
  array boolean txn3[0..3])

if rst then
  TRCSEQSTR.STATE = 0;
else
  case TRCSEQSTR.STATE of
    0 then
      if txn0[1] then
        TRCSEQSTR.STATE = 1;
      if txn0[2] then
        TRCSEQSTR.STATE = 2;
      if txn0[3] then
        TRCSEQSTR.STATE = 3;
    1 then
      if txn1[0] then
        TRCSEQSTR.STATE = 0;
      if txn1[1] then
        TRCSEQSTR.STATE = 1;
      if txn1[2] then
        TRCSEQSTR.STATE = 2;
      if txn1[3] then
        TRCSEQSTR.STATE = 3;
    2 then
      if txn2[0] then
        TRCSEQSTR.STATE = 0;
      if txn2[1] then
        TRCSEQSTR.STATE = 1;
      if txn2[2] then
        TRCSEQSTR.STATE = 2;
      if txn2[3] then
        TRCSEQSTR.STATE = 3;
    3 then
      if txn3[0] then
        TRCSEQSTR.STATE = 0;
      if txn3[1] then
        TRCSEQSTR.STATE = 1;
      if txn3[2] then
        TRCSEQSTR.STATE = 2;
if txn3[3] then
    TRCSEQSTR.STATE = 3;

D4.6.11 Single-shot Comparator Controls

IQSKXC
If a trace unit is exposed to speculative execution or execution in Transactional state, when Address Comparators
are used to activate resource events in the trace unit, then those resource events might be activated when speculative
execution occurs:

- A Single Address Comparator might signal a match on speculative execution or within a transaction.
- An Address Range Comparator might signal a match on speculative execution or within a transaction.

For example, this means that if an Address Comparator is used to activate a Counter or assert an External Output,
then that Counter might decrement, or that External Output might become asserted, as a result of speculative
execution. The Single-shot Comparator Controls for Address Comparators make it possible for resource events in
the trace unit to be activated based only on non-speculative execution, that is, only on architectural execution.

A trace unit can provide up to eight Single-shot Comparator Controls. Each Single-shot Comparator Control can be
used with one or more Address Comparators.

ITLFLF
Single-shot Comparator Controls can be used as a trace unit resource, to activate trace unit resource events. For
example, a Single-shot Comparator Control can be selected to:

- Enable or reload a trace unit Counter.
- Initiate a transition in the trace unit Sequencer state machine.
- Assert an External Output.

A Single-shot Comparator Control can therefore, if programmed to assert an External Output, be used to indicate to
a trace analyzer that a particular instruction has been resolved for execution. This means that a trace analyzer can
start or stop trace capture that is based on the architectural execution of that instruction.

IRBMXW
If a trace unit contains one or more Address Comparators, Arm recommends that at least one Single-shot
Comparator Control is implemented.

IVNBPG
A Single-shot Comparator Control works in the following way:

1. One or more Address Comparators are selected by using the TRCSSCCR<n> for the Single-shot Comparator
   Control. The selected Address Comparators can be all Single Address Comparators, all Address Range
   Comparators, or a combination of both.

2. When one of the selected Address Comparators matches, then when the instruction is confirmed to have
   architecturally executed, the Single-shot Comparator Control fires.

When a selected Address Comparator matches, but the instruction is confirmed to have not architecturally executed,
the Single-shot Comparator Control does not fire.

RXVVYX
When an instruction which matches an Address Comparator is confirmed to have architecturally executed, and the
Address Comparator is selected by TRCSSCCR<n> and TRCSSCSR<n>.STATUS is 0 or TRCSSCCR<n>.RST is
1, the Single-shot Comparator Control <n> fires.

RXFJGB
When a TSB CSYNC instruction is executed while a Single-shot Comparator Control is programmed to fire due to the
TSB CSYNC instruction, only when the related Trace synchronization event has completed, the Single-shot
Comparator fires.

RSWNFV
When a Single-shot Comparator Control fires, the trace unit considers this an indirect write to set
TRCSSCSR<n>.STATUS to 1.

RGBDCK
While the resources are in the Paused state, when the conditions for a Single-shot Comparator Control to fire are
met:

- If TRCSSCCR<n>.RST is 1 or TRCSSCSR<n>.STATUS is 0 then TRCSSCSR<n>.PENDING is indirectly
  written to 1.
- If TRCSSCCR<n>.RST is 0 and TRCSSCSR<n>.STATUS is 1 then TRCSSCSR<n>.PENDING is either
  indirectly written to 1 or is unchanged.
When one of the Address Comparators selected for a Single-shot Comparator Control matches, when the instruction that it matches on is in a Transaction which fails or is canceled, the Single-shot Comparator Control does not fire.

When the trace unit becomes disabled and an Address Comparator selected by a Single-shot Comparator Control has matched on an instruction that is still speculative, the Single-shot Comparator Control does not fire.

While the PE is executing in Transactional state, when the trace unit becomes disabled and an Address Comparator selected by a Single-shot Comparator Control has matched on an instruction in Transactional state, the Single-shot Comparator Control does not fire.

When tracing becomes prohibited and an Address Comparator selected by a Single-shot Comparator Control has matched on an instruction that is still speculative, the Single-shot Comparator Control waits until the instruction speculation is resolved and fires if the instruction is architecturally executed.

While the PE is executing in Transactional state, when tracing becomes prohibited and an Address Comparator selected by a Single-shot Comparator Control has matched on an instruction in Transactional state, the Single-shot Comparator Control does not fire.

When tracing becomes prohibited and an Address Comparator selected by a Single-shot Comparator Control has matched on an instruction that is still speculative, the Single-shot Comparator Control waits until the instruction speculation is resolved and fires if the instruction is architecturally executed.

While a Single-shot Comparator Control is used for instruction address comparisons, when the conditions for the Single-shot Comparator Control to fire are met, the Single-shot Comparator Control fires, regardless of whether either of the following are true:
- The instruction fails its condition code check.
- The instruction is a failed store-exclusive operation.

When a Single-shot Comparator Control is used to activate a resource event, the resource event might not become activated until some time after the trace unit has traced the instruction. This is because although the trace unit traces the instruction it is executed, the PE might not confirm whether the instruction was architecturally executed or canceled because of mis-speculation until some time later, and therefore the Single-shot Comparator Control might not fire until some time later.

**Single-shot Comparator Control modes**

Each Single-shot Comparator Control operates in one of the following modes:
- Single-shot mode: The Single-shot Comparator Control only fires once. That is, after it has fired, it never fires again.
- Multi-shot mode: The Single-shot Comparator Control resets after each time it fires. That is, it can fire again when a selected Address Comparator next signals an address match for an instruction is architecturally executed.

TRCSSCCR<α>.RST selects the mode.

While a Single-shot Comparator Control is in multi-shot mode, when the Single-shot Comparator Control fires, it fires for a maximum of one processor clock cycle.

While a Single-shot Comparator Control is in multi-shot mode, when multiple of the comparators selected for the Single-shot Comparator Control match in close succession, only the first match is guaranteed to cause the Single-shot Comparator Control to fire.

Examples of multiple comparator matches in close succession include:
- More than one of the Address Comparators that are selected signal an address match simultaneously.
- One Address Comparator matches multiple times while a first match is still waiting to be resolved.
Operation while in Paused state

The resolution of a speculative instruction might occur after the PE has entered a Trace Prohibited region and the resources have entered the Paused state. If the conditions for the Single-shot Comparator Control to fire are met while the resources are in the Paused state, then the Single-shot Comparator Control resource event is delayed to ensure that the Single-shot Comparator Control resource event is seen.

While the resources are in the Paused state, the Single-shot Comparator Controls do not fire.

When the resources enter the Running state while TRCSSCSR<n>.PENDING is 1, the following occur:

- If TRCSSCCR<n>.RST is 1 or TRCSSCSR<n>.STATUS is 0, the Single-shot Comparator Control fires.
- TRCSSCSR<n>.PENDING is indirectly written to 0.

D4.6.12 External Outputs

The ETE architecture supports between one and four External Outputs. The number of outputs that a trace unit has is IMPLEMENTATION DEFINED, but at least one output is always implemented.

External Outputs are used to indicate ETEEvents to a trace analyzer. ETEEvents are controlled by resources events. For example, an instruction Address Comparator can be used to drive one of the resource events.

If an External Output is programmed to be asserted based on program execution, such as an Address Comparator, the External Output might not be asserted at the same time as any trace generated by that program execution is output by the trace unit.

Typically, the generated trace might be buffered in a trace unit which means that the External Output would be asserted before the trace is output.

To program an External Output, use TRCEVENTCTL0R to select a resource.

The TRCIDR0.NUMEVENT field shows how many ETEEvents are supported for the particular implementation.

The External Outputs are connected to the CTI for the PE.

In a PE where the Trace Unit reset is independent of the PE Warm reset and the CTI reset is independent of the PE Warm reset, transmission of External Outputs to the CTI is unaffected by a PE Warm reset.

Operation while in Paused state

While the resources are in the Paused state an ETEEvent might occur, but any associated trace packets might not be generated. TRCRSR.EVENT provides a mechanism for recording this occurrence so that the trace unit state can be saved and restored.

While the resources are in the Paused state, the ETEEvent selector retains that one or more ETEEvents have been generated but not traced.

When an ETEEvent has been generated and the associated External Output has been asserted, any associated Event packets are generated.

When an ETEEvent has been generated but the associated External Output has not been asserted, any associated Event packets are not generated.

When an ETEEvent occurs while the resources are in the Paused state and the Event packet is not output, the trace unit sets the associated TRCRSR.EVENT[n] to 1.

When an ETEEvent occurs while the resources are in the Paused state, this is considered an indirect write to TRCRSR.

When the trace unit enters the Running state while TRCRSR.EVENT[n] is 1, the associated ETEEvent resource is active for a single PE clock cycle, and the trace unit clears TRCRSR.EVENT[n] to 0 and considers the action an indirect write to TRCRSR.
When the trace unit enters the Running state while TRCRSR.EVENT[n] is 1, the resource event selected by TRCEVENTCTL0R.EVENT<n> might also be active on the same PE clock cycle. If this happens, the associated ETEEvent resource is active for the single PE clock cycle and might not generate 2 separate ETEEvents for these 2 causes of the ETEEvent.

### D4.6.13 External inputs

**ITPPSC**

The trace unit uses the PMU events as External Inputs.

**RMTGBK**

If a PMU event number that is selected is not implemented, then the External Input resource event is inactive.

**RYCNCR**

Unless otherwise specified by the PMU event, the following PMU events are selectable by the trace unit:

- All PMU events required by the Performance Monitors Extension.
- If FEAT_PMUv3 is implemented, all Common architectural and microarchitectural events implemented by the Performance Monitors Extension.

Note: This includes Common events defined by other extensions, such as SVE and SPE.

**IXJBHV**

Additional ASIC-specific events can be selected by using a number in the IMPLEMENTATION DEFINED region.

**IVWHTZ**

There is no requirement that all IMPLEMENTATION DEFINED events are visible by the trace unit, PMU counters, and the PMUEVENT bus.

**RPHDQT**

For ETE, export of PMU events to the trace unit is not affected by PMCR.X or PMCR_EL0.X.

**RBFWZB**

When SelfHostedTraceEnabled() == TRUE and tracing is prohibited, only the PMU events defined by rules VBCBZ and KRSMY are exported to the trace unit.

**RWSXTC**

When SelfHostedTraceEnabled() == FALSE and the PE is in Secure state and counting in Secure state is prohibited, only the PMU events defined by rules VBCBZ and KRSMY are exported to the trace unit.

**RVCBZ**

The following PMU events are always exported to the trace unit:

- CTI_TRIGOUT4.
- CTI_TRIGOUT5.
- CTI_TRIGOUT6.
- CTI_TRIGOUT7.
- PMU_OVFS.
- TRB_WRAP.
- TRB_TRIG.

**RQPDKH**

When multiple occurrences of the same PMU event occur during the same cycle, the trace unit only observes a single occurrence of the PMU event.

**IMHHNV**

The operation of the PMU events and the generation of trace within the trace unit are not tightly coupled, and there is no guarantee that enabling ViewInst due to a PMU event will cause the instruction that caused the PMU event to be traced.

**RXGMPN**

When the PMU event SW_INCR is selected as an External Input and PMSWINC_EL0 is written from EL2 or EL3, the External Input is asserted if any bit [n] written has the value 1 and the relevant PMU counter <n> is implemented.

**R6XPZK**

When the PMU event SW_INCR is selected as an External Input and PMSWINC_EL0 is written from EL1 or EL0, the External Input is asserted if any bit [n] written has the value 1 and the relevant PMU counter <n> is implemented and any of the following are true:

- EL2 is implemented and enabled in the current Security state and <n> is less than MDCR_EL2.HPMN.
- EL2 is implemented and disabled in the current Security state.
- EL2 is not implemented.
In a PE where the trace unit reset is independent of the PE Warm reset and the CTI reset is independent of the PE Warm reset, transmission of the CTI_TRIGOUTn events from the CTI to the trace unit is unaffected by a PE Warm reset.

### Operation while in Paused state

The External Input Selectors are guaranteed to be active while in the Paused state. This is so that while the resources are Paused any cross trigger event is not lost but will occur when the resources resume running.

TRCRSR.EXTIN provides a mechanism to capture the sticky state of the External Input Selectors while in the Paused state so that the ETE state can be saved and restored.

When one or more selected External Inputs have been asserted, while the resources are in the Paused state, the trace unit retains the knowledge that one or more selected External Inputs have been asserted.

While the resources are in the Pausing or Paused states and the trace unit is not disabled and is not in the low-power state, when an External Input Selector n detects the selected External Input is asserted, the trace unit performs an indirect write to set TRCRSR.EXTIN[n] to 1.

When the resources enter the Running state while TRCRSR.EXTIN[n] is 1, the External Input Selector resource is active for a single PE clock cycle, and the trace unit clears TRCRSR.EXTIN[n] and considers the action an indirect write to TRCRSR.

### Operation while in low-power state

While the trace unit is in the low-power state, the External Input Selectors are inactive.

### D4.6.14 PE Comparator Inputs

The ETE architecture provides up to eight PE Comparator Inputs, that is, inputs that can be driven from comparators within the PE. For example, a PE might contain IMPLEMENTATION DEFINED comparators.

The number of PE Comparator Inputs is indicated by TRCIDR4.NUMPC.

While the PE is executing in a Trace Prohibited region, the PE Comparator Inputs are inactive.

The PE Comparator Inputs are only used in IMPLEMENTATION SPECIFIC code.

Each PE Comparator Input can be used in any of the following ways:

- To control the ViewInst start/stop function.
- To control the Single-shot Comparator Controls.
- As an independent resource.

The behavior of the PE Comparator Inputs on the resources and the filtering of the trace unit is IMPLEMENTATION DEFINED.
Chapter D5
ETE Protocol Descriptions

This chapter describes the ETE packets. It contains the following sections:

- *About the ETE protocol* on page D5-4874.
- *Summary list of ETE packets* on page D5-4877.
- *Alphabetical list of ETE packets* on page D5-4880.
D5.1  About the ETE protocol

An ETE trace unit generates a trace byte stream. The protocol is a byte-based packet protocol, which means that the trace byte stream is constructed of multiple packets, where each packet contains one or more bytes of data.

A packet consists of a single header byte, followed by zero or more payload bytes.

D5.1.1  Encoding schemes

Field encodings

Bit Replacement:

The packet outputs bits which update a piece of state. Bits output by the packet replace only those bits in the piece of state. Bits not output by the packet remain unchanged in the piece of state.

Unsigned LE128n:

The data is encoded as an unsigned number. The least significant bits of the number are output in the least significant bits of the packet. Bits not output by the packet are 0.

POD:

The encoding is specific to the packet.

Unary code:

The sequence for this variable is one of the following:

- A 0.
- A number of 1s followed by a 0.
- All 1s for the size of the variable, as defined by the packet.

For example, the permitted values for a 4-bit variable are:

- 0b0.
- 0b10.
- 0b110.
- 0b1110.
- 0b1111.

Instruction set encoding

For any virtual instruction address, the instruction set is output as a combination of the following two pieces of information:

- The SF bit encoded in Context packets.
- The sub_isa encoded by the type of the following groups of packets:
  - Target Address packets.
  - Exception packets.
  - Q packets.
  - Source Address packets.

The sub_isa indicates either:

- ISO.
- IS1.
Table D5-1 indicates how the combination of the SF bit and sub_isa indicate the instruction set.

<table>
<thead>
<tr>
<th>SF Bit</th>
<th>sub_isa</th>
<th>Instruction Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>IS0</td>
<td>AArch32 A32</td>
</tr>
<tr>
<td>0b0</td>
<td>IS1</td>
<td>AArch32 T32</td>
</tr>
<tr>
<td>0b1</td>
<td>IS0</td>
<td>AArch64 A64</td>
</tr>
</tbody>
</table>

The sub_isa also indicates the alignment of the virtual instruction addresses. Table D5-2 indicates the alignment of each sub_isa.

<table>
<thead>
<tr>
<th>sub_isa</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS0</td>
<td>Word-Aligned</td>
</tr>
<tr>
<td>IS1</td>
<td>Halfword-Aligned</td>
</tr>
</tbody>
</table>

The following packets encode the sub_isa:

- Exception Short Address IS0 Packet.
- Exception Short Address IS1 Packet.
- Exception 32-bit Address IS0 Packet.
- Exception 32-bit Address IS1 Packet.
- Exception 64-bit Address IS0 Packet.
- Exception 64-bit Address IS1 Packet.
- Exception 32-bit Address IS0 with Context Packet.
- Exception 32-bit Address IS1 with Context Packet.
- Exception 64-bit Address IS0 with Context Packet.
- Exception 64-bit Address IS1 with Context Packet.
- Target Address Short IS0 Packet.
- Target Address Short IS1 Packet.
- Target Address 32-bit IS0 Packet.
- Target Address 32-bit IS1 Packet.
- Target Address 64-bit IS0 Packet.
- Target Address 64-bit IS1 Packet.
- Target Address with Context 32-bit IS0 Packet.
- Target Address with Context 32-bit IS1 Packet.
- Target Address with Context 64-bit IS0 Packet.
- Target Address with Context 64-bit IS1 Packet.
- Source Address Short IS0 Packet.
- Source Address Short IS1 Packet.
- Source Address 32-bit IS0 Packet.
- Source Address 32-bit IS1 Packet.
- Source Address 64-bit IS0 Packet.
- Source Address 64-bit IS1 Packet.
• Q short address IS0 Packet.
• Q short address IS1 Packet.
• Q 32-bit address IS0 Packet.
• Q 32-bit address IS1 Packet.
D5.2 Summary list of ETE packets

Table D5-3 lists the ETE packets ordered by the header byte.

<table>
<thead>
<tr>
<th>Header byte</th>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>Alignment Synchronization Packet</td>
<td>Identifies a packet boundary.</td>
</tr>
<tr>
<td>00000000</td>
<td>Discard Packet</td>
<td>Indicates a Discard element.</td>
</tr>
<tr>
<td>00000000</td>
<td>Overflow Packet</td>
<td>Indicates that a trace unit buffer overflow has occurred.</td>
</tr>
<tr>
<td>00000001</td>
<td>Trace Info Packet</td>
<td>Resets trace compression to a known architectural state.</td>
</tr>
<tr>
<td>0000001x</td>
<td>Timestamp Packet</td>
<td>Indicates a Timestamp element.</td>
</tr>
<tr>
<td>00000100</td>
<td>Trace On Packet</td>
<td>Indicates that there has been a discontinuity in the trace element stream.</td>
</tr>
<tr>
<td>00000110</td>
<td>PE Reset Packet</td>
<td>Indicates that a PE Reset has occurred.</td>
</tr>
<tr>
<td>00000110</td>
<td>Transaction Failure Packet</td>
<td>Indicates that a Transaction Failure has occurred.</td>
</tr>
<tr>
<td>00000110</td>
<td>Exception 32-bit Address IS0 with Context Packet</td>
<td>Indicates that an exception has occurred.</td>
</tr>
<tr>
<td>00000110</td>
<td>Exception 32-bit Address IS1 with Context Packet</td>
<td>Indicates that an exception has occurred.</td>
</tr>
<tr>
<td>00000110</td>
<td>Exception 64-bit Address IS0 with Context Packet</td>
<td>Indicates that an exception has occurred.</td>
</tr>
<tr>
<td>00000110</td>
<td>Exception 64-bit Address IS1 with Context Packet</td>
<td>Indicates that an exception has occurred.</td>
</tr>
<tr>
<td>00000110</td>
<td>Exception Exact Match Address Packet</td>
<td>Indicates that an exception has occurred.</td>
</tr>
<tr>
<td>00000110</td>
<td>Exception Short Address IS0 Packet</td>
<td>Indicates that an exception has occurred.</td>
</tr>
<tr>
<td>00000110</td>
<td>Exception Short Address IS1 Packet</td>
<td>Indicates that an exception has occurred.</td>
</tr>
<tr>
<td>00000110</td>
<td>Exception 32-bit Address IS0 Packet</td>
<td>Indicates that an exception has occurred.</td>
</tr>
<tr>
<td>00000110</td>
<td>Exception 32-bit Address IS1 Packet</td>
<td>Indicates that an exception has occurred.</td>
</tr>
<tr>
<td>00000110</td>
<td>Exception 64-bit Address IS0 Packet</td>
<td>Indicates that an exception has occurred.</td>
</tr>
<tr>
<td>00000110</td>
<td>Exception 64-bit Address IS1 Packet</td>
<td>Indicates that an exception has occurred.</td>
</tr>
<tr>
<td>00001010</td>
<td>Transaction Start Packet</td>
<td>Indicates that the PE has started to execute in Transactional state.</td>
</tr>
<tr>
<td>00001011</td>
<td>Transaction Commit Packet</td>
<td>Indicates that the PE has successfully finished an outer transaction and is leaving Transactional state.</td>
</tr>
<tr>
<td>00001100</td>
<td>Cycle Count Format 2_0 small commit Packet</td>
<td>Indicates a Commit element and a Cycle Count element.</td>
</tr>
<tr>
<td>00001101</td>
<td>Cycle Count Format 2_1 Packet</td>
<td>Indicates a Cycle Count element.</td>
</tr>
<tr>
<td>00001101</td>
<td>Cycle Count Format 2_0 large commit Packet</td>
<td>Indicates a Commit element and a Cycle Count element.</td>
</tr>
<tr>
<td>00001110</td>
<td>Cycle Count Format 1_1 with count Packet</td>
<td>Indicates a Cycle Count element.</td>
</tr>
<tr>
<td>Header byte</td>
<td>Name</td>
<td>Purpose</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>00001110</td>
<td>Cycle Count Format 1_0 with count Packet</td>
<td>Indicates zero or one Commit elements followed by a Cycle Count element.</td>
</tr>
<tr>
<td>00001111</td>
<td>Cycle Count Format 1_1 unknown count Packet</td>
<td>Indicates a Cycle Count element.</td>
</tr>
<tr>
<td>00001111</td>
<td>Cycle Count Format 1_0 unknown count Packet</td>
<td>Indicates zero or one Commit elements followed by a Cycle Count element with an UNKNOWN cycle count value.</td>
</tr>
<tr>
<td>000100xx</td>
<td>Cycle Count Format 3_1 Packet</td>
<td>Indicates a Cycle Count element.</td>
</tr>
<tr>
<td>0001xxxx</td>
<td>Cycle Count Format 3_0 Packet</td>
<td>Indicates a Commit element and a Cycle Count element.</td>
</tr>
<tr>
<td>0010101</td>
<td>Commit Packet</td>
<td>Indicates a Commit element.</td>
</tr>
<tr>
<td>0010111x</td>
<td>Cancel Format 1 Packet</td>
<td>Indicates a Cancel element optionally followed by a Mispredict element.</td>
</tr>
<tr>
<td>001100xx</td>
<td>Mispredict Packet</td>
<td>Indicates 0-2 E or N Atom elements followed by one Mispredict element.</td>
</tr>
<tr>
<td>001101xx</td>
<td>Cancel Format 2 Packet</td>
<td>Indicates zero or more E or N Atom elements followed by a Cancel element and a Mispredict element.</td>
</tr>
<tr>
<td>00111xxx</td>
<td>Cancel Format 3 Packet</td>
<td>Indicates zero or one E Atom element followed by a Cancel element with a payload of 2-5 and one Mispredict element.</td>
</tr>
<tr>
<td>01110000</td>
<td>Ignore Packet</td>
<td>To align packet boundary to memory boundary.</td>
</tr>
<tr>
<td>0111xxxx</td>
<td>Event Packet</td>
<td>Indicates 1-4 Event elements.</td>
</tr>
<tr>
<td>10000000</td>
<td>Context Same Packet</td>
<td>Indicates a Context element.</td>
</tr>
<tr>
<td>10000001</td>
<td>Context Packet</td>
<td>Indicates a Context element.</td>
</tr>
<tr>
<td>1000010</td>
<td>Target Address with Context 32-bit IS0 Packet</td>
<td>Indicates a Target Address element and a Context element.</td>
</tr>
<tr>
<td>1000011</td>
<td>Target Address with Context 32-bit IS1 Packet</td>
<td>Indicates a Target Address element and a Context element.</td>
</tr>
<tr>
<td>1000101</td>
<td>Target Address with Context 64-bit IS0 Packet</td>
<td>Indicates a Target Address element and a Context element.</td>
</tr>
<tr>
<td>1000110</td>
<td>Target Address with Context 64-bit IS1 Packet</td>
<td>Indicates a Target Address element and a Context element.</td>
</tr>
<tr>
<td>1000100</td>
<td>Timestamp Marker Packet</td>
<td>Indicates a Timestamp Marker element.</td>
</tr>
<tr>
<td>100100xx</td>
<td>Target Address Exact Match Packet</td>
<td>Indicates a Target Address element.</td>
</tr>
<tr>
<td>10010101</td>
<td>Target Address Short IS0 Packet</td>
<td>Indicates a Target Address element.</td>
</tr>
<tr>
<td>10010110</td>
<td>Target Address Short IS1 Packet</td>
<td>Indicates a Target Address element.</td>
</tr>
<tr>
<td>1001101</td>
<td>Target Address 32-bit IS0 Packet</td>
<td>Indicates a Target Address element.</td>
</tr>
<tr>
<td>1001110</td>
<td>Target Address 32-bit IS1 Packet</td>
<td>Indicates a Target Address element.</td>
</tr>
<tr>
<td>1001101</td>
<td>Target Address 64-bit IS0 Packet</td>
<td>Indicates a Target Address element.</td>
</tr>
<tr>
<td>1001110</td>
<td>Target Address 64-bit IS1 Packet</td>
<td>Indicates a Target Address element.</td>
</tr>
</tbody>
</table>
## Table D5-3 ETE Packets (continued)

<table>
<thead>
<tr>
<th>Header byte</th>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>101000xx</td>
<td>Q with Exact match address Packet</td>
<td>Indicates that some instructions have executed with an address of the next instruction.</td>
</tr>
<tr>
<td>10100101</td>
<td>Q short address IS0 Packet</td>
<td>Indicates that some instructions have executed with an address of the next instruction.</td>
</tr>
<tr>
<td>10100110</td>
<td>Q short address IS1 Packet</td>
<td>Indicates that some instructions have executed with an address of the next instruction.</td>
</tr>
<tr>
<td>10101010</td>
<td>Q 32-bit address IS0 Packet</td>
<td>Indicates that some instructions have executed with an address of the next instruction.</td>
</tr>
<tr>
<td>10101011</td>
<td>Q 32-bit address IS1 Packet</td>
<td>Indicates that some instructions have executed with an address of the next instruction.</td>
</tr>
<tr>
<td>10101100</td>
<td>Q with count Packet</td>
<td>Indicates that some instructions have executed.</td>
</tr>
<tr>
<td>10101111</td>
<td>Q Packet</td>
<td>Indicates that some instructions have executed, without a count of the number of instructions.</td>
</tr>
<tr>
<td>101100xx</td>
<td>Source Address Exact Match Packet</td>
<td>Indicates the source address of a <code>P0 instruction</code>, and that the instruction was taken.</td>
</tr>
<tr>
<td>10110100</td>
<td>Source Address Short IS0 Packet</td>
<td>Indicates the source address of a <code>P0 instruction</code>, and that the instruction was taken.</td>
</tr>
<tr>
<td>10110101</td>
<td>Source Address Short IS1 Packet</td>
<td>Indicates the source address of a <code>P0 instruction</code>, and that the instruction was taken.</td>
</tr>
<tr>
<td>10110110</td>
<td>Source Address 32-bit IS0 Packet</td>
<td>Indicates the source address of a <code>P0 instruction</code>, and that the instruction was taken.</td>
</tr>
<tr>
<td>10110111</td>
<td>Source Address 32-bit IS1 Packet</td>
<td>Indicates the source address of a <code>P0 instruction</code>, and that the instruction was taken.</td>
</tr>
<tr>
<td>10111000</td>
<td>Source Address 64-bit IS0 Packet</td>
<td>Indicates the source address of a <code>P0 instruction</code>, and that the instruction was taken.</td>
</tr>
<tr>
<td>10111001</td>
<td>Source Address 64-bit IS1 Packet</td>
<td>Indicates the source address of a <code>P0 instruction</code>, and that the instruction was taken.</td>
</tr>
<tr>
<td>110101xx</td>
<td>Atom Format 5.2 Packet</td>
<td>Indicates five <code>Atom elements</code>.</td>
</tr>
<tr>
<td>110110xx</td>
<td>Atom Format 2 Packet</td>
<td>Indicates two <code>Atom elements</code>.</td>
</tr>
<tr>
<td>110111xx</td>
<td>Atom Format 4 Packet</td>
<td>Indicates four <code>Atom elements</code>.</td>
</tr>
<tr>
<td>11110101</td>
<td>Atom Format 5.1 Packet</td>
<td>Indicates five <code>Atom elements</code>.</td>
</tr>
<tr>
<td>1111011x</td>
<td>Atom Format 1 Packet</td>
<td>Indicates one <code>Atom element</code>.</td>
</tr>
<tr>
<td>11111xxx</td>
<td>Atom Format 3 Packet</td>
<td>Indicates three <code>Atom elements</code>.</td>
</tr>
<tr>
<td>11xxxxxx</td>
<td>Atom Format 6 Packet</td>
<td>Indicates 3-23 E <code>Atom elements</code>, plus a subsequent E Atom or N <code>Atom element</code>.</td>
</tr>
</tbody>
</table>
D5.3 Alphabetical list of ETE packets

This section lists each ETE packet and their description.
### D5.3.1 Alignment Synchronization Packet

**Purpose**

Identifies a packet boundary.

**Configurations**

All.

This packet forms a unique bit and byte pattern. Searching for this pattern allows the trace analyzer to identify packet boundaries.

**Packet Layout**

![Alignment Synchronization Packet](image)

**Additional information**

For more information about the decoding of this packet see `Parse_ExtensionPacket()` on page K4-11620.

For more information about the generation of this packet see `Trace protocol synchronization` on page D4-4776.

R\text{EXZZJ}  

Any byte that follows this unique sequence of bits is the header byte of a new packet.

R\text{VRKLP}  

This packet must be output before the first Trace Info packet.
### D5.3.2 Discard Packet

**Purpose**

Indicates a *Discard element*.

**Configurations**

All.

Indicates a *Discard element*.

**Packet Layout**

![Discard Packet](image)

**Element sequence**

This packet encodes the following sequence:

1. *Discard element.*

**Additional information**

For more information about the decoding of this packet see *DiscardPacket()* on page K4-11620.

For more information about the generation of this packet see *Trace protocol synchronization* on page D4-4776.

This packet is used to discard any speculative trace that the trace analyzer might still be holding onto.
D5.3.3 Overflow Packet

Purpose

Indicates that a trace unit buffer overflow has occurred.

Configurations

All.

Indicates that a trace unit buffer overflow has occurred and data might have been lost.

Packet Layout

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Figure D5-3 Overflow Packet

Element sequence

This packet encodes the following sequence:

1. Overflow element.
2. Discard element.

Additional information

For more information about the decoding of this packet see OverflowPacket() on page K4-11620.
D5.3.4  Trace Info Packet

Purpose

Resets trace compression to a known architectural state.

Configurations

All.

The trace info packet resets the trace compression to a known state. Any fields which are not output are treated as if the value is zero.

Packet Layout - Variant 1

```
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0
```

Figure D5-4 Trace Info Packet (1)

Packet Layout - Variant 2

```
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1
(0) T (0) (0) (0) (0) CC
```

Figure D5-5 Trace Info Packet (2)

Packet Layout - Variant 3

```
0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0
C0 SPEC[6:0]
C0 SPEC[13:7]
C0 SPEC[20:14]
C0 SPEC[27:21]
(0) (0) (0) SPEC[31:28]
```

Figure D5-6 Trace Info Packet (3)
Packet Layout - Variant 4

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>(0)</td>
<td>T</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>CC</td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>SPEC[6:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>SPEC[13:7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>SPEC[20:14]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>SPEC[27:21]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>SPEC[31:28]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure D5-7 Trace Info Packet (4)

Packet Layout - Variant 5

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C1</td>
<td>CYCT[6:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0)</td>
<td>(0)</td>
<td>CYCT[11:7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure D5-8 Trace Info Packet (5)

Packet Layout - Variant 6

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(0)</td>
<td>T</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>CC</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>CYCT[6:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0)</td>
<td>(0)</td>
<td>CYCT[11:7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure D5-9 Trace Info Packet (6)
Packet Layout - Variant 7

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C0</td>
<td>SPEC[6:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>SPEC[13:7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>SPEC[20:14]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>SPEC[27:21]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0) (0) (0)</td>
<td>SPEC[31:28]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>CYCT[6:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0) (0)</td>
<td>CYCT[11:7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure D5-10 Trace Info Packet (7)

Packet Layout - Variant 8

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(0)</td>
<td>T</td>
<td>(0) (0) (0) (0)</td>
<td>CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>SPEC[6:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>SPEC[13:7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>SPEC[20:14]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>SPEC[27:21]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0) (0) (0)</td>
<td>SPEC[31:28]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>CYCT[6:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0) (0)</td>
<td>CYCT[11:7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure D5-11 Trace Info Packet (8)

Field descriptions

**C0**

Continuation Bit.

The encoding for this field is Unary code.

---

0b0  Last byte in this section.

0b1  At least one more byte follows in this section.

**C1**

Continuation Bit.
The encoding for this field is Unary code.

<table>
<thead>
<tr>
<th>0b0</th>
<th>Last byte in this section.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>At least one more byte follows in this section.</td>
</tr>
</tbody>
</table>

**CC**

Cycle count enable indicator.
When this field is not output, it is treated as if it is zero.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>0b0</th>
<th>Cycle counting is not enabled.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>Cycle counting is enabled.</td>
</tr>
</tbody>
</table>

**CYCT**

The cycle count threshold.
When this field is not output, it is treated as if it is zero.
The encoding for this field is unsigned LE128n.

**SPEC**

The number of uncommitted P0 elements in the trace.
When this field is not output, it is treated as if it is zero.
The encoding for this field is unsigned LE128n.

**T**

Transactional state indicator.
When this field is not output, it is treated as if it is zero.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>0b0</th>
<th>The PE is not currently executing in Transactional state.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>The PE is currently executing in Transactional state.</td>
</tr>
</tbody>
</table>

**Element sequence**

This packet encodes the following sequence:

1. *Trace Info element.*

**Additional information**

For more information about the decoding of this packet see *TraceInfoPacket()* on page K4-11621.
D5.3.5 Trace On Packet

Purpose

Indicates that there has been a discontinuity in the trace element stream.

Configurations

All.

A Trace On packet indicates to a trace analyzer that the trace unit has generated a Trace On element.

Packet Layout

```
  7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0
```

Figure D5-12 Trace On Packet

Element sequence

This packet encodes the following sequence:

1. Trace On element.

Additional information

For more information about the decoding of this packet see TraceOnPacket() on page K4-11622.
D5.3.6 Timestamp Packet

**Purpose**
Indicates a Timestamp element.

**Configurations**
TRCIDR0.TSSIZE != 0b00000.

**Packet Layout - Variant 1**

```
0 0 0 0 0 0 1 0
C0  TS[6:0]
C0  TS[13:7]
C0  TS[20:14]
C0  TS[27:21]
C0  TS[34:28]
C0  TS[41:35]
C0  TS[48:42]
C0  TS[55:49]
    TS[63:56]
```

**Figure D5-13 Timestamp Packet (1)**

**Packet Layout - Variant 2**

```
0 0 0 0 0 0 1 1
C0  TS[6:0]
C0  TS[13:7]
C0  TS[20:14]
C0  TS[27:21]
C0  TS[34:28]
C0  TS[41:35]
C0  TS[48:42]
C0  TS[55:49]
    TS[63:56]
C1  COUNT[6:0]
C1  COUNT[13:7]
(0)(0) COUNT[19:14]
```

**Figure D5-14 Timestamp Packet (2)**
Field descriptions

C0

Continuation Bit.

The encoding for this field is Unary code.

<table>
<thead>
<tr>
<th>0b0</th>
<th>Last byte in this section.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>At least one more byte follows in this section.</td>
</tr>
</tbody>
</table>

C1

Continuation Bit.

The encoding for this field is Unary code.

<table>
<thead>
<tr>
<th>0b0</th>
<th>Last byte in this section.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>At least one more byte follows in this section.</td>
</tr>
</tbody>
</table>

COUNT

The number of PE clock cycles between the most recent Cycle Count element and the element related to the Timestamp. When the COUNT field is not present, the cycle count value is UNKNOWN.

The encoding for this field is unsigned LE128n.

TS

Timestamp Value.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. Timestamp element.

Additional information

For more information about the decoding of this packet see TimestampPacket() on page K4-11636.
D5.3.7 Timestamp Marker Packet

Purpose

Indicates a Timestamp Marker element.

Configurations

TRCIDR0.TSSIZE \neq 0b0000 and TRCIDR0.TSMARK == 0b1

Packet Layout

Figure D5-15 Timestamp Marker Packet
D5.3.8  Transaction Start Packet

**Purpose**
Indicates that the PE has started to execute in Transactional state.

**Configurations**
All.

**Packet Layout**

```
  7  6  5  4  3  2  1  0
  0  0  0  1  0  1  0
```

**Figure D5-16 Transaction Start Packet**

**Element sequence**
This packet encodes the following sequence:


**Additional information**
For more information about the decoding of this packet see `TransactionStartPacket()` on page K4-11635.
D5.3.9 Transaction Commit Packet

Purpose
Indicates that the PE has successfully finished an outer transaction and is leaving Transactional state.

Configurations
All.

Packet Layout

0 0 0 0 1 0 1 1

Figure D5-17 Transaction Commit Packet

Element sequence
This packet encodes the following sequence:
1. Transaction Commit element.

Additional information
For more information about the decoding of this packet see TransactionCommitPacket() on page K4-11635.
D5.3.10 Exception Exact Match Address Packet

Purpose
Indicates that an exception has occurred.

Configurations
All.

Packet Layout

![Figure D5-18 Exception Exact Match Address Packet](image)

Field descriptions

A
Preferred Exception Return address.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>The Preferred Exception Return is the same as address history buffer entry 0.</td>
</tr>
<tr>
<td>0b01</td>
<td>The Preferred Exception Return is the same as address history buffer entry 1.</td>
</tr>
<tr>
<td>0b10</td>
<td>The Preferred Exception Return is the same as address history buffer entry 2.</td>
</tr>
</tbody>
</table>

E
Identifies the elements that are indicated by this packet.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>1. Exception element (TYPE, ADDRESS).</td>
</tr>
<tr>
<td>0b10</td>
<td>1. Target Address element (ADDRESS).</td>
</tr>
<tr>
<td></td>
<td>2. Exception element (TYPE, ADDRESS).</td>
</tr>
</tbody>
</table>

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE
The exception type.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00000</td>
<td>PE Reset, also see PE Reset Packet.</td>
</tr>
<tr>
<td>0b00001</td>
<td>Debug halt.</td>
</tr>
<tr>
<td>0b00010</td>
<td>Call.</td>
</tr>
<tr>
<td>0b00011</td>
<td>Trap.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------</td>
</tr>
<tr>
<td>0b00100</td>
<td>System Error.</td>
</tr>
<tr>
<td>0b00110</td>
<td>Inst debug.</td>
</tr>
<tr>
<td>0b00111</td>
<td>Data debug.</td>
</tr>
<tr>
<td>0b01010</td>
<td>Alignment.</td>
</tr>
<tr>
<td>0b01011</td>
<td>Inst Fault.</td>
</tr>
<tr>
<td>0b01100</td>
<td>Data Fault.</td>
</tr>
<tr>
<td>0b01110</td>
<td>IRQ.</td>
</tr>
<tr>
<td>0b01111</td>
<td>FIQ.</td>
</tr>
<tr>
<td>0b10000</td>
<td>IMPLEMENTATION DEFINED 0.</td>
</tr>
<tr>
<td>0b10001</td>
<td>IMPLEMENTATION DEFINED 1.</td>
</tr>
<tr>
<td>0b10010</td>
<td>IMPLEMENTATION DEFINED 2.</td>
</tr>
<tr>
<td>0b10011</td>
<td>IMPLEMENTATION DEFINED 3.</td>
</tr>
<tr>
<td>0b10100</td>
<td>IMPLEMENTATION DEFINED 4.</td>
</tr>
<tr>
<td>0b10101</td>
<td>IMPLEMENTATION DEFINED 5.</td>
</tr>
<tr>
<td>0b10110</td>
<td>IMPLEMENTATION DEFINED 6.</td>
</tr>
<tr>
<td>0b10111</td>
<td>IMPLEMENTATION DEFINED 7.</td>
</tr>
<tr>
<td>0b11000</td>
<td>Reserved. See Transaction Failure Packet.</td>
</tr>
</tbody>
</table>

All other values are reserved. Reserved values might be defined in a future version of the architecture.

**Additional information**

For more information about the decoding of this packet see `Parse_ExceptionPacket()` on page K4-11629.

For more information about the generation of this packet see `Address compression` on page D4-4839.
D5.3.11 Exception Short Address IS0 Packet

Purpose
Indicates that an exception has occurred.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>E[1]</td>
<td>TYPE</td>
<td>E[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C0</td>
<td>A[8:2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[16:9]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure D5-19 Exception Short Address IS0 Packet

Field descriptions

A
Preferred Exception Return address.
Preferred Exception Return address bits[1:0] always have the value 0b00.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

C0
Continuation Bit.
The encoding for this field is Unary code.

| 0b0 | Last byte in this section. |
| 0b1 | At least one more byte follows in this section. |

E
Identifies the elements that are indicated by this packet.
The encoding for this field is POD.

| 0b01 | 1. Exception element (TYPE, ADDRESS). |
| 0b10 | 1. Target Address element (ADDRESS). |
| | 2. Exception element (TYPE, ADDRESS). |

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE
The exception type.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00000</td>
<td>PE Reset, also see PE Reset Packet.</td>
</tr>
<tr>
<td>0b00001</td>
<td>Debug halt.</td>
</tr>
<tr>
<td>0b00010</td>
<td>Call.</td>
</tr>
<tr>
<td>0b00011</td>
<td>Trap.</td>
</tr>
<tr>
<td>0b00100</td>
<td>System Error.</td>
</tr>
<tr>
<td>0b00110</td>
<td>Inst debug.</td>
</tr>
<tr>
<td>0b00111</td>
<td>Data debug.</td>
</tr>
<tr>
<td>0b01010</td>
<td>Alignment.</td>
</tr>
<tr>
<td>0b01011</td>
<td>Inst Fault.</td>
</tr>
<tr>
<td>0b01100</td>
<td>Data Fault.</td>
</tr>
<tr>
<td>0b01110</td>
<td>IRQ.</td>
</tr>
<tr>
<td>0b01111</td>
<td>FIQ.</td>
</tr>
<tr>
<td>0b10000</td>
<td>IMPLEMENTATION DEFINED 0.</td>
</tr>
<tr>
<td>0b10001</td>
<td>IMPLEMENTATION DEFINED 1.</td>
</tr>
<tr>
<td>0b10010</td>
<td>IMPLEMENTATION DEFINED 2.</td>
</tr>
<tr>
<td>0b10011</td>
<td>IMPLEMENTATION DEFINED 3.</td>
</tr>
<tr>
<td>0b10100</td>
<td>IMPLEMENTATION DEFINED 4.</td>
</tr>
<tr>
<td>0b10101</td>
<td>IMPLEMENTATION DEFINED 5.</td>
</tr>
<tr>
<td>0b10110</td>
<td>IMPLEMENTATION DEFINED 6.</td>
</tr>
<tr>
<td>0b10111</td>
<td>IMPLEMENTATION DEFINED 7.</td>
</tr>
<tr>
<td>0b11000</td>
<td>Reserved. See Transaction Failure Packet.</td>
</tr>
</tbody>
</table>

All other values are reserved. Reserved values might be defined in a future version of the architecture.

**Additional information**

For more information about the decoding of this packet see `Parse_ExceptionPacket()` on page K4-11629.

For more information about the generation of this packet see `Address compression` on page D4-4839.

For more information about the encoding of this packet see `Instruction set encoding` on page D5-4874.
## D5.3.12 Exception Short Address IS1 Packet

### Purpose
Indicates that an exception has occurred.

### Configurations
All.

### Packet Layout

![Figure D5-20 Exception Short Address IS1 Packet]

### Field descriptions

#### A
Preferred Exception Return address.
Preferred Exception Return address bit[0] always has the value 0b0.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

#### C0
Continuation Bit.
The encoding for this field is Unary code.

#### E
Identifies the elements that are indicated by this packet.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>0b0</th>
<th>Last byte in this section.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>At least one more byte follows in this section.</td>
</tr>
</tbody>
</table>

#### TYPE
The exception type.

All other values are reserved. Reserved values might be defined in a future version of the architecture.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00000</td>
<td>PE Reset, also see PE Reset Packet.</td>
</tr>
<tr>
<td>0b00001</td>
<td>Debug halt.</td>
</tr>
<tr>
<td>0b00010</td>
<td>Call.</td>
</tr>
<tr>
<td>0b00011</td>
<td>Trap.</td>
</tr>
<tr>
<td>0b00100</td>
<td>System Error.</td>
</tr>
<tr>
<td>0b00110</td>
<td>Inst debug.</td>
</tr>
<tr>
<td>0b00111</td>
<td>Data debug.</td>
</tr>
<tr>
<td>0b01010</td>
<td>Alignment.</td>
</tr>
<tr>
<td>0b01011</td>
<td>Inst Fault.</td>
</tr>
<tr>
<td>0b01100</td>
<td>Data Fault.</td>
</tr>
<tr>
<td>0b01110</td>
<td>IRQ.</td>
</tr>
<tr>
<td>0b01111</td>
<td>FIQ.</td>
</tr>
<tr>
<td>0b10000</td>
<td>IMPLEMENTATION DEFINED 0.</td>
</tr>
<tr>
<td>0b10001</td>
<td>IMPLEMENTATION DEFINED 1.</td>
</tr>
<tr>
<td>0b10010</td>
<td>IMPLEMENTATION DEFINED 2.</td>
</tr>
<tr>
<td>0b10011</td>
<td>IMPLEMENTATION DEFINED 3.</td>
</tr>
<tr>
<td>0b10100</td>
<td>IMPLEMENTATION DEFINED 4.</td>
</tr>
<tr>
<td>0b10101</td>
<td>IMPLEMENTATION DEFINED 5.</td>
</tr>
<tr>
<td>0b10110</td>
<td>IMPLEMENTATION DEFINED 6.</td>
</tr>
<tr>
<td>0b10111</td>
<td>IMPLEMENTATION DEFINED 7.</td>
</tr>
<tr>
<td>0b11000</td>
<td>Reserved. See Transaction Failure Packet.</td>
</tr>
</tbody>
</table>

All other values are reserved. Reserved values might be defined in a future version of the architecture.

**Additional information**

For more information about the decoding of this packet see `Parse_ExceptionPacket()` on page K4-11629.

For more information about the generation of this packet see `Address compression` on page D4-4839.

For more information about the encoding of this packet see `Instruction set encoding` on page D5-4874.
D5.3.13 Exception 32-bit Address IS0 Packet

Purpose
Indicates that an exception has occurred.

Configurations
All.

Packet Layout

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>E[1]</td>
<td>TYPE</td>
<td>E[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(0)</td>
<td>A[8:2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(0)</td>
<td>A[15:9]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A[23:16]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A[31:24]</td>
</tr>
</tbody>
</table>
```

Figure D5-21 Exception 32-bit Address IS0 Packet

Field descriptions

A
Preferred Exception Return address.
Preferred Exception Return address bits[1:0] always have the value 0b00.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

E
Identifies the elements that are indicated by this packet.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>0b01</th>
<th>1.</th>
<th>Exception element (TYPE, ADDRESS).</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>1.</td>
<td>Target Address element (ADDRESS).</td>
</tr>
<tr>
<td></td>
<td>2.</td>
<td>Exception element (TYPE, ADDRESS).</td>
</tr>
</tbody>
</table>

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE
The exception type.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>0b00000</th>
<th>PE Reset, also see PE Reset Packet.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00001</td>
<td>Debug halt.</td>
</tr>
<tr>
<td>0b00010</td>
<td>Call.</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------</td>
</tr>
<tr>
<td>0b00011</td>
<td>Trap.</td>
</tr>
<tr>
<td>0b00100</td>
<td>System Error.</td>
</tr>
<tr>
<td>0b00110</td>
<td>Inst debug.</td>
</tr>
<tr>
<td>0b00111</td>
<td>Data debug.</td>
</tr>
<tr>
<td>0b01010</td>
<td>Alignment.</td>
</tr>
<tr>
<td>0b01011</td>
<td>Inst Fault.</td>
</tr>
<tr>
<td>0b01100</td>
<td>Data Fault.</td>
</tr>
<tr>
<td>0b01110</td>
<td>IRQ.</td>
</tr>
<tr>
<td>0b01111</td>
<td>FIQ.</td>
</tr>
<tr>
<td>0b10000</td>
<td>IMPLEMENTATION DEFINED 0.</td>
</tr>
<tr>
<td>0b10001</td>
<td>IMPLEMENTATION DEFINED 1.</td>
</tr>
<tr>
<td>0b10010</td>
<td>IMPLEMENTATION DEFINED 2.</td>
</tr>
<tr>
<td>0b10011</td>
<td>IMPLEMENTATION DEFINED 3.</td>
</tr>
<tr>
<td>0b10100</td>
<td>IMPLEMENTATION DEFINED 4.</td>
</tr>
<tr>
<td>0b10101</td>
<td>IMPLEMENTATION DEFINED 5.</td>
</tr>
<tr>
<td>0b10110</td>
<td>IMPLEMENTATION DEFINED 6.</td>
</tr>
<tr>
<td>0b10111</td>
<td>IMPLEMENTATION DEFINED 7.</td>
</tr>
<tr>
<td>0b11000</td>
<td>Reserved. See Transaction Failure Packet.</td>
</tr>
</tbody>
</table>

All other values are reserved. Reserved values might be defined in a future version of the architecture.

**Additional information**

For more information about the decoding of this packet see `Parse_ExceptionPacket()` on page K4-11629.

For more information about the generation of this packet see `Address compression` on page D4-4839.

For more information about the encoding of this packet see `Instruction set encoding` on page D5-4874.
D5.3.14 Exception 32-bit Address IS1 Packet

**Purpose**
Indicates that an exception has occurred.

**Configurations**
All.

**Packet Layout**

![Figure D5-22 Exception 32-bit Address IS1 Packet](image)

**Field descriptions**

**A**
Preferred Exception Return address.
Preferred Exception Return address bit[0] always has the value 0b0.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

**E**
Identifies the elements that are indicated by this packet.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>0b01</th>
<th>1.</th>
<th>Exception element (TYPE, ADDRESS).</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>1.</td>
<td>Target Address element (ADDRESS).</td>
</tr>
<tr>
<td></td>
<td>2.</td>
<td>Exception element (TYPE, ADDRESS).</td>
</tr>
</tbody>
</table>

All other values are reserved. Reserved values might be defined in a future version of the architecture.

**TYPE**
The exception type.
The encoding for this field is POD.

<p>| 0b0000 | PE Reset, also see PE Reset Packet. |
|        | Debug halt.                           |
| 0b0010 | Call.                                 |</p>
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00011</td>
<td>Trap</td>
</tr>
<tr>
<td>0b00100</td>
<td>System Error</td>
</tr>
<tr>
<td>0b00110</td>
<td>Inst debug</td>
</tr>
<tr>
<td>0b00111</td>
<td>Data debug</td>
</tr>
<tr>
<td>0b01010</td>
<td>Alignment</td>
</tr>
<tr>
<td>0b01011</td>
<td>Inst Fault</td>
</tr>
<tr>
<td>0b01100</td>
<td>Data Fault</td>
</tr>
<tr>
<td>0b01110</td>
<td>IRQ</td>
</tr>
<tr>
<td>0b01111</td>
<td>FIQ</td>
</tr>
<tr>
<td>0b10000</td>
<td>IMPLEMENTATION DEFINED 0</td>
</tr>
<tr>
<td>0b10001</td>
<td>IMPLEMENTATION DEFINED 1</td>
</tr>
<tr>
<td>0b10010</td>
<td>IMPLEMENTATION DEFINED 2</td>
</tr>
<tr>
<td>0b10011</td>
<td>IMPLEMENTATION DEFINED 3</td>
</tr>
<tr>
<td>0b10100</td>
<td>IMPLEMENTATION DEFINED 4</td>
</tr>
<tr>
<td>0b10101</td>
<td>IMPLEMENTATION DEFINED 5</td>
</tr>
<tr>
<td>0b10110</td>
<td>IMPLEMENTATION DEFINED 6</td>
</tr>
<tr>
<td>0b10111</td>
<td>IMPLEMENTATION DEFINED 7</td>
</tr>
<tr>
<td>0b11000</td>
<td>Reserved. See Transaction Failure Packet.</td>
</tr>
</tbody>
</table>

All other values are reserved. Reserved values might be defined in a future version of the architecture.

**Additional information**

For more information about the decoding of this packet see *Parse_ExceptionPacket()* on page K4-11629.

For more information about the generation of this packet see *Address compression* on page D4-4839.

For more information about the encoding of this packet see *Instruction set encoding* on page D5-4874.
D5.3.15 Exception 64-bit Address IS0 Packet

Purpose
Indicates that an exception has occurred.

Configurations
All.

Packet Layout

```
+---+---+---+---+---+---+---+---+---+---+---+
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | E[1] | TYPE | E[0] |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
|(0) | A[8:2] |
|(0) | A[15:9] |
|    | A[23:16] |
|    | A[31:24] |
|    | A[39:32] |
|    | A[47:40] |
|    | A[55:48] |
|    | A[63:56] |
```

Figure D5-23 Exception 64-bit Address IS0 Packet

Field descriptions

A
Preferred Exception Return address.
Preferred Exception Return address bits[1:0] always have the value \(0b00\).
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

E
Identifies the elements that are indicated by this packet.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th></th>
<th>1.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(0b01)</td>
</tr>
<tr>
<td></td>
<td>(0b10)</td>
</tr>
<tr>
<td></td>
<td>1. Target Address element (ADDRESS).</td>
</tr>
<tr>
<td></td>
<td>2. Exception element (TYPE, ADDRESS).</td>
</tr>
</tbody>
</table>

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE
The exception type.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00000</td>
<td>PE Reset, also see PE Reset Packet.</td>
</tr>
<tr>
<td>0b00001</td>
<td>Debug halt.</td>
</tr>
<tr>
<td>0b00010</td>
<td>Call.</td>
</tr>
<tr>
<td>0b00011</td>
<td>Trap.</td>
</tr>
<tr>
<td>0b00100</td>
<td>System Error.</td>
</tr>
<tr>
<td>0b00110</td>
<td>Inst debug.</td>
</tr>
<tr>
<td>0b00111</td>
<td>Data debug.</td>
</tr>
<tr>
<td>0b01010</td>
<td>Alignment.</td>
</tr>
<tr>
<td>0b01011</td>
<td>Inst Fault.</td>
</tr>
<tr>
<td>0b01100</td>
<td>Data Fault.</td>
</tr>
<tr>
<td>0b01110</td>
<td>IRQ.</td>
</tr>
<tr>
<td>0b01111</td>
<td>FIQ.</td>
</tr>
<tr>
<td>0b10000</td>
<td>IMPLEMENTATION DEFINED 0.</td>
</tr>
<tr>
<td>0b10001</td>
<td>IMPLEMENTATION DEFINED 1.</td>
</tr>
<tr>
<td>0b10010</td>
<td>IMPLEMENTATION DEFINED 2.</td>
</tr>
<tr>
<td>0b10011</td>
<td>IMPLEMENTATION DEFINED 3.</td>
</tr>
<tr>
<td>0b10100</td>
<td>IMPLEMENTATION DEFINED 4.</td>
</tr>
<tr>
<td>0b10101</td>
<td>IMPLEMENTATION DEFINED 5.</td>
</tr>
<tr>
<td>0b10110</td>
<td>IMPLEMENTATION DEFINED 6.</td>
</tr>
<tr>
<td>0b10111</td>
<td>IMPLEMENTATION DEFINED 7.</td>
</tr>
<tr>
<td>0b11000</td>
<td>Reserved. See Transaction Failure Packet.</td>
</tr>
</tbody>
</table>

All other values are reserved. Reserved values might be defined in a future version of the architecture.

**Additional information**

For more information about the decoding of this packet see *Parse_ExceptionPacket()* on page K4-11629.

For more information about the generation of this packet see *Address compression* on page D4-4839.

For more information about the encoding of this packet see *Instruction set encoding* on page D5-4874.
## D5.3.16 Exception 64-bit Address IS1 Packet

**Purpose**

Indicates that an exception has occurred.

**Configurations**

All.

### Packet Layout

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Preferred Exception Return address. Preferred Exception Return address bit[0] always has the value 0b0. The address is compressed relative to address history buffer entry 0. The encoding for this field is Bit replacement.</td>
</tr>
<tr>
<td>E</td>
<td>Identifies the elements that are indicated by this packet. The encoding for this field is POD.</td>
</tr>
<tr>
<td>TYPE</td>
<td>The exception type.</td>
</tr>
</tbody>
</table>

All other values are reserved. Reserved values might be defined in a future version of the architecture.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00000</td>
<td>PE Reset, also see PE Reset Packet.</td>
</tr>
<tr>
<td>0b00001</td>
<td>Debug halt.</td>
</tr>
<tr>
<td>0b00010</td>
<td>Call.</td>
</tr>
<tr>
<td>0b00011</td>
<td>Trap.</td>
</tr>
<tr>
<td>0b00100</td>
<td>System Error.</td>
</tr>
<tr>
<td>0b00110</td>
<td>Inst debug.</td>
</tr>
<tr>
<td>0b00111</td>
<td>Data debug.</td>
</tr>
<tr>
<td>0b01010</td>
<td>Alignment.</td>
</tr>
<tr>
<td>0b01011</td>
<td>Inst Fault.</td>
</tr>
<tr>
<td>0b01100</td>
<td>Data Fault.</td>
</tr>
<tr>
<td>0b01110</td>
<td>IRQ.</td>
</tr>
<tr>
<td>0b01111</td>
<td>FIQ.</td>
</tr>
<tr>
<td>0b10000</td>
<td>IMPLEMENTATION DEFINED 0.</td>
</tr>
<tr>
<td>0b10001</td>
<td>IMPLEMENTATION DEFINED 1.</td>
</tr>
<tr>
<td>0b10010</td>
<td>IMPLEMENTATION DEFINED 2.</td>
</tr>
<tr>
<td>0b10011</td>
<td>IMPLEMENTATION DEFINED 3.</td>
</tr>
<tr>
<td>0b10100</td>
<td>IMPLEMENTATION DEFINED 4.</td>
</tr>
<tr>
<td>0b10101</td>
<td>IMPLEMENTATION DEFINED 5.</td>
</tr>
<tr>
<td>0b10110</td>
<td>IMPLEMENTATION DEFINED 6.</td>
</tr>
<tr>
<td>0b10111</td>
<td>IMPLEMENTATION DEFINED 7.</td>
</tr>
<tr>
<td>0b11000</td>
<td>Reserved. See Transaction Failure Packet.</td>
</tr>
</tbody>
</table>

All other values are reserved. Reserved values might be defined in a future version of the architecture.

**Additional information**

For more information about the decoding of this packet see `Parse_ExceptionPacket()` on page K4-11629.
For more information about the generation of this packet see `Address compression` on page D4-4839.
For more information about the encoding of this packet see `Instruction set encoding` on page D5-4874.
D5.3.17 Exception 32-bit Address IS0 with Context Packet

Purpose
Indicates that an exception has occurred.

Configurations
All.

Packet Layout - Variant 1

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>E[1]</td>
<td>TYPE</td>
<td>E[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(0)</td>
<td>A[8:2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0)</td>
<td>A[15:9]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A[23:16]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A[31:24]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>NS</td>
<td>SF</td>
<td>(0)</td>
<td>(0)</td>
<td>EL</td>
<td></td>
</tr>
</tbody>
</table>

Figure D5-25 Exception 32-bit Address IS0 with Context Packet (1)

Packet Layout - Variant 2

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>E[1]</td>
<td>TYPE</td>
<td>E[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(0)</td>
<td>A[8:2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0)</td>
<td>A[15:9]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A[23:16]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A[31:24]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>NS</td>
<td>SF</td>
<td>(0)</td>
<td>(0)</td>
<td>EL</td>
<td></td>
</tr>
</tbody>
</table>

| CONTEXTID[7:0] |
| CONTEXTID[15:8] |
| CONTEXTID[23:16] |
| CONTEXTID[31:24] |

Figure D5-26 Exception 32-bit Address IS0 with Context Packet (2)
Packet Layout - Variant 3

Figure D5-27 Exception 32-bit Address IS0 with Context Packet (3)

Packet Layout - Variant 4

Figure D5-28 Exception 32-bit Address IS0 with Context Packet (4)

Field descriptions

A

Preferred Exception Return address.
Preferred Exception Return address bits[1:0] always have the value 0b00. The address is compressed relative to address history buffer entry 0. The encoding for this field is Bit replacement.

**CONTEXTID**

Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See *Context identifier tracing on page D4-4835.*

**E**

Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>Context element.</td>
</tr>
<tr>
<td></td>
<td>2. Exception element (TYPE, ADDRESS).</td>
</tr>
<tr>
<td>0b10</td>
<td>1. Target Address element (ADDRESS).</td>
</tr>
<tr>
<td></td>
<td>2. Context element.</td>
</tr>
<tr>
<td></td>
<td>3. Exception element (TYPE, ADDRESS).</td>
</tr>
</tbody>
</table>

All other values are reserved. Reserved values might be defined in a future version of the architecture.

**EL**

Exception level at the Preferred Exception Return address.

The encoding for this field is POD.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>EL0.</td>
</tr>
<tr>
<td>0b1</td>
<td>EL1.</td>
</tr>
<tr>
<td>0b10</td>
<td>EL2.</td>
</tr>
<tr>
<td>0b11</td>
<td>EL3.</td>
</tr>
</tbody>
</table>

**NS**

Security state

When this field is not output, the Security state is the same as the most recently output Security state.

The encoding for this field is POD.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The PE is in Secure state.</td>
</tr>
<tr>
<td>0b1</td>
<td>The PE is in Non-secure state.</td>
</tr>
</tbody>
</table>

**SF**

AArch64 state.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The PE is in AArch32 state.</td>
</tr>
<tr>
<td>0b1</td>
<td>The PE is in AArch64 state.</td>
</tr>
</tbody>
</table>

**TYPE**

The exception type.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>PE Reset, also see PE Reset Packet.</td>
</tr>
<tr>
<td>0b0001</td>
<td>Debug halt.</td>
</tr>
<tr>
<td>0b0010</td>
<td>Call.</td>
</tr>
<tr>
<td>0b0011</td>
<td>Trap.</td>
</tr>
<tr>
<td>0b0010</td>
<td>System Error.</td>
</tr>
<tr>
<td>0b0011</td>
<td>Inst debug.</td>
</tr>
<tr>
<td>0b0011</td>
<td>Data debug.</td>
</tr>
<tr>
<td>0b0100</td>
<td>Alignment.</td>
</tr>
<tr>
<td>0b0101</td>
<td>Inst Fault.</td>
</tr>
<tr>
<td>0b0101</td>
<td>Data Fault.</td>
</tr>
<tr>
<td>0b0110</td>
<td>IRQ.</td>
</tr>
<tr>
<td>0b0111</td>
<td>FIQ.</td>
</tr>
<tr>
<td>0b1000</td>
<td>IMPLEMENTATION DEFINED 0.</td>
</tr>
<tr>
<td>0b1001</td>
<td>IMPLEMENTATION DEFINED 1.</td>
</tr>
<tr>
<td>0b1010</td>
<td>IMPLEMENTATION DEFINED 2.</td>
</tr>
<tr>
<td>0b1011</td>
<td>IMPLEMENTATION DEFINED 3.</td>
</tr>
<tr>
<td>0b1010</td>
<td>IMPLEMENTATION DEFINED 4.</td>
</tr>
<tr>
<td>0b1011</td>
<td>IMPLEMENTATION DEFINED 5.</td>
</tr>
<tr>
<td>0b1011</td>
<td>IMPLEMENTATION DEFINED 6.</td>
</tr>
<tr>
<td>0b1100</td>
<td>IMPLEMENTATION DEFINED 7.</td>
</tr>
<tr>
<td>0b1100</td>
<td>Reserved. See Transaction Failure Packet.</td>
</tr>
</tbody>
</table>

All other values are reserved. Reserved values might be defined in a future version of the architecture.

**VMID**

Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
This field contains a value of zero.
The encoding for this field is POD.
See Virtual context identifier tracing on page D4-4837.

Additional information
For more information about the decoding of this packet see Parse_ExceptionPacket() on page K4-11629.
For more information about the generation of this packet see Address compression on page D4-4839.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.18 Exception 32-bit Address IS1 with Context Packet

Purpose
Indicates that an exception has occurred.

Configurations
All.

Packet Layout - Variant 1

![Packet Layout - Variant 1](image1)

Packet Layout - Variant 2

![Packet Layout - Variant 2](image2)
Packet Layout - Variant 3

![Figure D5-31 Exception 32-bit Address IS1 with Context Packet (3)](image)

Packet Layout - Variant 4

![Figure D5-32 Exception 32-bit Address IS1 with Context Packet (4)](image)

Field descriptions
A
Preferred Exception Return address.
Preferred Exception Return address bit[0] always has the value 0b0.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

CONTEXTID

Context identifier.
When this field is not output, the Context identifier is the same as the most recently output Context identifier.
If Context identifier tracing is disabled, then one of the following must occur:
• This field is not traced.
• This field contains a value of zero.
The encoding for this field is POD.
See Context identifier tracing on page D4-4835.

E

Identifies the elements that are indicated by this packet.
The encoding for this field is POD.

| 0b01 | 1. Context element. |
| 2. Exception element (TYPE, ADDRESS). |
| 0b10 | 1. Target Address element (ADDRESS). |
| 2. Context element. |
| 3. Exception element (TYPE, ADDRESS). |

All other values are reserved. Reserved values might be defined in a future version of the architecture.

EL

Exception level at the Preferred Exception Return address.
The encoding for this field is POD.

| 0b00 | EL0. |
| 0b01 | EL1. |
| 0b10 | EL2. |
| 0b11 | EL3. |

NS

Security state.
The encoding for this field is POD.

| 0b0 | The PE is in Secure state. |
| 0b1 | The PE is in Non-secure state. |

SF

AArch64 state.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The PE is in AArch32 state.</td>
</tr>
<tr>
<td>0b1</td>
<td>The PE is in AArch64 state.</td>
</tr>
</tbody>
</table>

**VMID**

Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.

All other values are reserved. Reserved values might be defined in a future version of the architecture.
• This field contains a value of zero.
  The encoding for this field is POD.
  See Virtual context identifier tracing on page D4-4837.

Additional information
For more information about the decoding of this packet see Parse_ExceptionPacket() on page K4-11629.
For more information about the generation of this packet see Address compression on page D4-4839.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.19 Exception 64-bit Address IS0 with Context Packet

**Purpose**
Indicates that an exception has occurred.

**Configurations**
All.

**Packet Layout - Variant 1**

```
  7  6  5  4  3  2  1  0
0  0  0  0  0  1  1  0
  0  E[1]  TYPE  E[0]
1  0  0  0  0  1  0  1
(0)  A[8:2]
(0)  A[15:9]
      A[23:16]
      A[31:24]
      A[39:32]
      A[47:40]
      A[55:48]
A[63:56]
0  0  NS  SF  (0)  (0)  EL
```

Figure D5-33 Exception 64-bit Address IS0 with Context Packet (1)
Packet Layout - Variant 2

![Diagram of Packet Layout - Variant 2]

Figure D5-34 Exception 64-bit Address IS0 with Context Packet (2)

Packet Layout - Variant 3

![Diagram of Packet Layout - Variant 3]

Figure D5-35 Exception 64-bit Address IS0 with Context Packet (3)
Packet Layout - Variant 4

Figure D5-36 Exception 64-bit Address IS0 with Context Packet (4)

Field descriptions

A

Preferred Exception Return address.
Preferred Exception Return address bits[1:0] always have the value 0b00.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

CONTEXTID

Context identifier.
When this field is not output, the Context identifier is the same as the most recently output Context identifier.
If Context identifier tracing is disabled, then one of the following must occur:
- This field is not traced.
- This field contains a value of zero.
The encoding for this field is POD.
See Context identifier tracing on page D4-4835.

E

Identifies the elements that are indicated by this packet.
ETE Protocol Descriptions
D5.3 Alphabetical list of ETE packets

The encoding for this field is POD.

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>1.       Context element.</td>
</tr>
<tr>
<td></td>
<td>2.       Exception element (TYPE, ADDRESS).</td>
</tr>
<tr>
<td>0b10</td>
<td>1.       Target Address element (ADDRESS).</td>
</tr>
<tr>
<td></td>
<td>2.       Context element.</td>
</tr>
<tr>
<td></td>
<td>3.       Exception element (TYPE, ADDRESS).</td>
</tr>
</tbody>
</table>

All other values are reserved. Reserved values might be defined in a future version of the architecture.

**EL**

Exception level at the Preferred Exception Return address.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>EL0.</td>
</tr>
<tr>
<td>0b01</td>
<td>EL1.</td>
</tr>
<tr>
<td>0b10</td>
<td>EL2.</td>
</tr>
<tr>
<td>0b11</td>
<td>EL3.</td>
</tr>
</tbody>
</table>

**NS**

Security state
The encoding for this field is POD.

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The PE is in Secure state.</td>
</tr>
<tr>
<td>0b1</td>
<td>The PE is in Non-secure state.</td>
</tr>
</tbody>
</table>

**SF**

AArch64 state.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The PE is in AArch32 state.</td>
</tr>
<tr>
<td>0b1</td>
<td>The PE is in AArch64 state.</td>
</tr>
</tbody>
</table>

**TYPE**

The exception type.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000000</td>
<td>PE Reset, also see PE Reset Packet.</td>
</tr>
<tr>
<td>0b000001</td>
<td>Debug halt.</td>
</tr>
<tr>
<td>0b000010</td>
<td>Call.</td>
</tr>
<tr>
<td>0b000011</td>
<td>Trap.</td>
</tr>
<tr>
<td>0b000100</td>
<td>System Error.</td>
</tr>
<tr>
<td>0b000110</td>
<td>Inst debug.</td>
</tr>
</tbody>
</table>
All other values are reserved. Reserved values might be defined in a future version of the architecture.

**VMID**

Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See *Virtual context identifier tracing on page D4-4837*.

**Additional information**

For more information about the decoding of this packet see *Parse_ExceptionPacket() on page K4-11629*.

For more information about the generation of this packet see *Address compression on page D4-4839*.

For more information about the encoding of this packet see *Instruction set encoding on page D5-4874*. 

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00111</td>
<td>Data debug.</td>
</tr>
<tr>
<td>0b01010</td>
<td>Alignment.</td>
</tr>
<tr>
<td>0b01011</td>
<td>Inst Fault.</td>
</tr>
<tr>
<td>0b01100</td>
<td>Data Fault.</td>
</tr>
<tr>
<td>0b01110</td>
<td>IRQ.</td>
</tr>
<tr>
<td>0b01111</td>
<td>FIQ.</td>
</tr>
<tr>
<td>0b10000</td>
<td>IMPLEMENTATION DEFINED 0.</td>
</tr>
<tr>
<td>0b10001</td>
<td>IMPLEMENTATION DEFINED 1.</td>
</tr>
<tr>
<td>0b10010</td>
<td>IMPLEMENTATION DEFINED 2.</td>
</tr>
<tr>
<td>0b10011</td>
<td>IMPLEMENTATION DEFINED 3.</td>
</tr>
<tr>
<td>0b10100</td>
<td>IMPLEMENTATION DEFINED 4.</td>
</tr>
<tr>
<td>0b10101</td>
<td>IMPLEMENTATION DEFINED 5.</td>
</tr>
<tr>
<td>0b10110</td>
<td>IMPLEMENTATION DEFINED 6.</td>
</tr>
<tr>
<td>0b10111</td>
<td>IMPLEMENTATION DEFINED 7.</td>
</tr>
<tr>
<td>0b11000</td>
<td>Reserved. See Transaction Failure Packet.</td>
</tr>
</tbody>
</table>
D5.3.20 Exception 64-bit Address IS1 with Context Packet

Purpose

Indicates that an exception has occurred.

Configurations

All.

Packet Layout - Variant 1

![Packet Layout Diagram]

Figure D5-37 Exception 64-bit Address IS1 with Context Packet (1)
Packet Layout - Variant 2

![Packet Layout - Variant 2 diagram]

Figure D5-38 Exception 64-bit Address IS1 with Context Packet (2)

Packet Layout - Variant 3

![Packet Layout - Variant 3 diagram]

Figure D5-39 Exception 64-bit Address IS1 with Context Packet (3)
Packet Layout - Variant 4

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Preferred Exception Return address. Preferred Exception Return address bit[0] always has the value 0b0. The address is compressed relative to address history buffer entry 0. The encoding for this field is Bit replacement.</td>
</tr>
<tr>
<td>CONTEXTID</td>
<td>Context identifier.  When this field is not output, the Context identifier is the same as the most recently output Context identifier.  If Context identifier tracing is disabled, then one of the following must occur:  • This field is not traced.  • This field contains a value of zero.  The encoding for this field is POD.  See Context identifier tracing on page D4-4835.</td>
</tr>
<tr>
<td>E</td>
<td>Identifies the elements that are indicated by this packet.</td>
</tr>
</tbody>
</table>

Figure D5-40 Exception 64-bit Address IS1 with Context Packet (4)
The encoding for this field is POD.

| 0b01 | 1. Context element.  
|      | 2. Exception element (TYPE, ADDRESS). |
| 0b10 | 1. Target Address element (ADDRESS).  
|      | 2. Context element.  
|      | 3. Exception element (TYPE, ADDRESS). |

All other values are reserved. Reserved values might be defined in a future version of the architecture.

**EL**

Exception level at the Preferred Exception Return address.
The encoding for this field is POD.

| 0b00 | EL0. |
| 0b01 | EL1. |
| 0b10 | EL2. |
| 0b11 | EL3. |

**NS**

Security state
The encoding for this field is POD.

| 0b0 | The PE is in Secure state. |
| 0b1 | The PE is in Non-secure state. |

**SF**

AArch64 state.
The encoding for this field is POD.

| 0b0 | The PE is in AArch32 state. |
| 0b1 | The PE is in AArch64 state. |

**TYPE**

The exception type.
The encoding for this field is POD.

| 0b0000 | PE Reset, also see PE Reset Packet. |
| 0b0001 | Debug halt. |
| 0b0010 | Call. |
| 0b0011 | Trap. |
| 0b0100 | System Error. |
| 0b0110 | Inst debug. |
All other values are reserved. Reserved values might be defined in a future version of the architecture.

VMID

Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing on page D4-4837.

Additional information

For more information about the decoding of this packet see Parse_ExceptionPacket() on page K4-11629.

For more information about the generation of this packet see Address compression on page D4-4839.

For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
### D5.3.21 Transaction Failure Packet

**Purpose**
- Indicates that a Transaction Failure has occurred.

**Configurations**
- All.

**Packet Layout**

![Figure D5-41 Transaction Failure Packet](image)

**Field descriptions**

- **E**
  - Identifies the elements that are indicated by this packet.
  - The encoding for this field is POD.

<table>
<thead>
<tr>
<th>0b01</th>
<th>1. Transaction Failure element.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>1. Target Address element (UNKNOWN).</td>
</tr>
<tr>
<td></td>
<td>2. Transaction Failure element.</td>
</tr>
</tbody>
</table>

  All other values are reserved. Reserved values might be defined in a future version of the architecture.

**Additional information**

For more information about the decoding of this packet see `Parse_ExceptionPacket()` on page K4-11629.

For more information about the generation of this packet see `Address compression` on page D4-4839.
D5.3.22 PE Reset Packet

Purpose
Indicates that a PE Reset has occurred.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>E[1]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure D5-42 PE Reset Packet

Field descriptions

E
Identifies the elements that are indicated by this packet.
The encoding for this field is POD.

| 0b01 | 1. Exception element (PE_Reset, UNKNOWN). |
| 0b10 | 1. Target Address element (UNKNOWN). |
|      | 2. Exception element (PE_Reset, UNKNOWN). |

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information
For more information about the decoding of this packet see Parse_ExceptionPacket() on page K4-11629.
For more information about the generation of this packet see Address compression on page D4-4839.
D5.3.23 Cycle Count Format 1_0 unknown count Packet

Purpose
Indicates zero or one Commit elements followed by a Cycle Count element with an unknown cycle count value.

Configurations
All the following conditions must be met:
- TRCIDR0.COMMOPT == 0b0.
- TRCIDR0.TRCCCI == 0b1.

Packet Layout

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C0</td>
<td>COMMIT[6:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>COMMIT[13:7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>COMMIT[20:14]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>COMMIT[27:21]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0) (0) (0) (0)</td>
<td>COMMIT[31:28]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure D5-43 Cycle Count Format 1_0 unknown count Packet

Field descriptions

C0
Continuation Bit.
The encoding for this field is Unary code.

<table>
<thead>
<tr>
<th>0b0</th>
<th>Last byte in this section.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>At least one more byte follows in this section.</td>
</tr>
</tbody>
</table>

COMMIT
If this field is zero, there is no Commit element. Otherwise, there is a Commit element before the Cycle Count element and this field indicates the number of P0 elements committed by the Commit element.
The encoding for this field is unsigned LE128n.

Element sequence
This packet encodes the following sequence:
1. Commit element.

Additional information
For more information about the decoding of this packet see CycleCountFormat1Packet() on page K4-11636.
D5.3.24 Cycle Count Format 1_1 unknown count Packet

Purpose

Indicates a Cycle Count element.

Configurations

All the following conditions must be met:

- TRCIDR0.COMMOPT == 0b1.
- TRCIDR0.TRCCCI == 0b1.

Packet Layout

![Figure D5-44 Cycle Count Format 1_1 unknown count Packet](image)

Element sequence

This packet encodes the following sequence:


Additional information

For more information about the decoding of this packet see CycleCountFormat1Packet() on page K4-11636.
D5.3.25   Cycle Count Format 1_0 with count Packet

Purpose
Indicates zero or one Commit elements followed by a Cycle Count element.

Configurations
All the following conditions must be met:
- TRCIDR0.COMMOPT == 0b0.
- TRCIDR0.TRCCCI == 0b1.

Packet Layout

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0b0</th>
<th>Last byte in this section.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>At least one more byte follows in this section.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMIT[6:0]</td>
</tr>
<tr>
<td>COMMIT[13:7]</td>
</tr>
<tr>
<td>COMMIT[20:14]</td>
</tr>
<tr>
<td>COMMIT[27:21]</td>
</tr>
<tr>
<td>(0) (0) (0) (0) COMMIT[31:28]</td>
</tr>
<tr>
<td>COUNT[6:0]</td>
</tr>
<tr>
<td>COUNT[13:7]</td>
</tr>
<tr>
<td>(0) (0) COUNT[19:14]</td>
</tr>
</tbody>
</table>

Figure D5-45 Cycle Count Format 1_0 with count Packet

Field descriptions

C0
Continuation Bit.
The encoding for this field is Unary code.

C1
Continuation Bit.
The encoding for this field is Unary code.

COMMIT
If this field is zero, there is no Commit element. Otherwise, there is a Commit element before the Cycle Count element and this field indicates the number of P0 elements committed by the Commit element.
The encoding for this field is unsigned LE128n.
COUNT

Indicates the number of PE clock cycles that have occurred between the 2 most recent Commit elements that both had a Cycle Count element associated with them. The cycle count is COUNT+cc_threshold.
The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. Commit element.
2. Cycle Count element.

Additional information

For more information about the decoding of this packet see CycleCountFormat1Packet() on page K4-11636.
D5.3.26 Cycle Count Format 1_1 with count Packet

Purpose
Indicates a Cycle Count element.

Configurations
All the following conditions must be met:
• TRCIDR0.COMMOPT == 0b1.
• TRCIDR0.TRCCCI == 0b1.

Packet Layout

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C0</td>
<td>COUNT[6:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>COUNT[13:7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0) (0)</td>
<td>COUNT[19:14]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure D5-46 Cycle Count Format 1_1 with count Packet

Field descriptions

C0
Continuation Bit.
The encoding for this field is Unary code.

| 0b0 | Last byte in this section. |
| 0b1 | At least one more byte follows in this section. |

COUNT
Indicates the number of PE clock cycles that have occurred between the 2 most recent Commit elements that both had a Cycle Count element associated with them. The cycle count is COUNT+cc_threshold.
The encoding for this field is unsigned LE128n.

Element sequence
This packet encodes the following sequence:

Additional information
For more information about the decoding of this packet see CycleCountFormat1Packet() on page K4-11636.
D5.3.27  Cycle Count Format 2_0 small commit Packet

Purpose

Indicates a Commit element and a Cycle Count element.

Configurations

All the following conditions must be met:
- TRCIDR0.COMMOPT == 0b0.
- TRCIDR0.TRCCCI == 0b1.

Packet Layout

```
0 0 0 0 1 1 0 0

|AAAA|BBBB|
```

Figure D5-47 Cycle Count Format 2_0 small commit Packet

Field descriptions

AAAA

Indicates the number of P0 elements to be resolved indicated by this field + 1.
The encoding for this field is POD.

BBBB

Indicates the cycle value. The cycle count is calculated from \(cc\_threshold + BBBB\).
The encoding for this field is POD.

Element sequence

This packet encodes the following sequence:

1. Commit element.
2. Cycle Count element.

Additional information

For more information about the decoding of this packet see `CycleCountFormat2Packet()` on page K4-11637.
D5.3.28 Cycle Count Format 2_0 large commit Packet

Purpose
Indicates a Commit element and a Cycle Count element.

Configurations
All the following conditions must be met:
- TRCIDR0.COMMOPT == 0b0.
- TRCIDR0.TRCCCI == 0b1.

Packet Layout

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

AAAA
Indicates the number of P0 elements to be resolved indicated by TRCIDR8.MAXSPEC + field - 15.
The number of P0 elements to be resolved must be greater than 0.
If the number of P0 elements to be resolved is less than 17 then it is preferred that a Cycle Count Format 2_0 small commit Packet is used.
The encoding for this field is POD.

BBBB
Indicates the cycle value. The cycle count is calculated from cc_threshold + BBBB.
The encoding for this field is POD.

Element sequence
This packet encodes the following sequence:
1. Commit element.
2. Cycle Count element.

Additional information
For more information about the decoding of this packet see CycleCountFormat2Packet() on page K4-11637.
D5.3.29 Cycle Count Format 2_1 Packet

Purpose
Indicates a Cycle Count element.

Configurations
All the following conditions must be met:
• TRCIDR0.COMMOPT == 0b1.
• TRCIDR0.TRCCCI == 0b1.

Packet Layout

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>BBBBB</td>
</tr>
</tbody>
</table>

Figure D5-49 Cycle Count Format 2_1 Packet

Field descriptions
BBBB
Indicates the cycle value. The cycle count is calculated from cc_threshold + BBBBB.
The encoding for this field is POD.

Element sequence
This packet encodes the following sequence:

Additional information
For more information about the decoding of this packet see CycleCountFormat2Packet() on page K4-11637.
D5.3.30 Cycle Count Format 3_0 Packet

Purpose
Indicates a Commit element and a Cycle Count element.

Configurations
All the following conditions must be met:
- TRCIDR0.COMMOPT == 0b0.
- TRCIDR0.TRCCCI == 0b1.

Packet Layout

\[
\begin{array}{cccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 0 & 1 & AA & BB \\
\end{array}
\]

Figure D5-50 Cycle Count Format 3_0 Packet

Field descriptions

AA
The number of P0 elements to be resolved indicated by this field + 1.
The encoding for this field is POD.

BB
Indicates the cycle value. The cycle count is calculated from cc_threshold + BB.
The encoding for this field is POD.

Element sequence
This packet encodes the following sequence:
1. Commit element.
2. Cycle Count element.

Additional information
For more information about the decoding of this packet see CycleCountFormat3Packet() on page K4-11637.
### D5.3.31 Cycle Count Format 3_1 Packet

#### Purpose
Indicates a *Cycle Count element.*

#### Configurations
All the following conditions must be met:
- TRCIDR0.COMMOPT == 0b1.
- TRCIDR0.TRCCCI == 0b1.

#### Packet Layout

![Figure D5-51 Cycle Count Format 3_1 Packet](image)

#### Field descriptions
**BB**
Indicates the cycle value. The cycle count is calculated from cc_threshold + BB.
The encoding for this field is POD.

#### Element sequence
This packet encodes the following sequence:
1. *Cycle Count element.*

#### Additional information
For more information about the decoding of this packet see `CycleCountFormat3Packet()` on page K4-11637.
D5.3.32 Commit Packet

Purpose
Indicates a Commit element.

Configurations
TRCIDR8.MAXSPEC > 0.

Packet Layout

<table>
<thead>
<tr>
<th>Field description</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>Continuation Bit. The encoding for this field is Unary code.</td>
</tr>
<tr>
<td>COMMIT</td>
<td>The number of $P0$ elements to be resolved. The encoding for this field is unsigned LE128n.</td>
</tr>
</tbody>
</table>

Element sequence
This packet encodes the following sequence:
1. Commit element.

Additional information
For more information about the decoding of this packet see CommitPacket() on page K4-11622.
D5.3.33 Cancel Format 1 Packet

Purpose
Indicates a Cancel element optionally followed by a Mispredict element.

Configurations
TRCIDR8.MAXSPEC > 0.

Packet Layout

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>Continuation Bit. The encoding for this field is Unary code.</td>
</tr>
<tr>
<td>CANCEL</td>
<td>The number of P0 elements to be canceled. The encoding for this field is unsigned LE128n.</td>
</tr>
<tr>
<td>M</td>
<td>Mispredict element included in the packet. The encoding for this field is POD.</td>
</tr>
</tbody>
</table>

Figure D5-53 Cancel Format 1 Packet

Field descriptions

C0
Continuation Bit. The encoding for this field is Unary code.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Last byte in this section.</td>
</tr>
<tr>
<td>0b1</td>
<td>At least one more byte follows in this section.</td>
</tr>
</tbody>
</table>

CANCEL
The number of P0 elements to be canceled. The encoding for this field is unsigned LE128n.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

M
Mispredict element included in the packet. The encoding for this field is POD.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>No Mispredict element occurred</td>
</tr>
<tr>
<td>0b1</td>
<td>A Mispredict element occurred after the Cancel element</td>
</tr>
</tbody>
</table>

Additional information
For more information about the decoding of this packet see CancelFormat1Packet() on page K4-11623.
D5.3.34 Cancel Format 2 Packet

**Purpose**

Indicates zero or more E or N Atom elements followed by a Cancel element and a Mispredict element.

**Configurations**

TRCIDR8.MAXSPEC > 0.

**Packet Layout**

![Figure D5-54 Cancel Format 2 Packet](image_url)

**Field descriptions**

**A**

Indicates the number of Atom elements that occurred before the Cancel element.

The encoding for this field is POD.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>1. Cancel element.</td>
</tr>
<tr>
<td>0b01</td>
<td>1. E Atom element.</td>
</tr>
<tr>
<td></td>
<td>2. Cancel element.</td>
</tr>
<tr>
<td>0b10</td>
<td>1. E Atom element.</td>
</tr>
<tr>
<td></td>
<td>2. E Atom element.</td>
</tr>
<tr>
<td></td>
<td>3. Cancel element.</td>
</tr>
<tr>
<td>0b11</td>
<td>1. N Atom element.</td>
</tr>
<tr>
<td></td>
<td>2. Cancel element.</td>
</tr>
</tbody>
</table>

**Additional information**

For more information about the decoding of this packet see CancelFormat2Packet() on page K4-11623.
D5.3.35 Cancel Format 3 Packet

Purpose
Indicates zero or one E Atom element followed by a Cancel element with a payload of 2-5 and one Mispredict element.

Configurations
TRCIDR8.MAXSPEC > 0.

Packet Layout

![Figure D5-55 Cancel Format 3 Packet](image)

Field descriptions
A
Indicates the number of Atom elements that occurred before the Cancel element.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>A</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>1. Cancel element</td>
</tr>
<tr>
<td>0b1</td>
<td>1. E Atom element</td>
</tr>
<tr>
<td></td>
<td>2. Cancel element</td>
</tr>
</tbody>
</table>

CC
The number of P0 elements to be canceled.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>CC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>Cancel 2 P0 elements</td>
</tr>
<tr>
<td>0b01</td>
<td>Cancel 3 P0 elements</td>
</tr>
<tr>
<td>0b10</td>
<td>Cancel 4 P0 elements</td>
</tr>
<tr>
<td>0b11</td>
<td>Cancel 5 P0 elements</td>
</tr>
</tbody>
</table>

Additional information
For more information about the decoding of this packet see CancelFormat3Packet() on page K4-11624.
D5.3.36 Mispredict Packet

Purpose

Indicates 0-2 E or N Atom elements followed by one Mispredict element.

Configurations

All.

Packet Layout

![Figure D5-56 Mispredict Packet](image)

Field descriptions

A

Indicates the number of Atom elements that occurred before the Mispredict element.

The encoding for this field is POD.

<table>
<thead>
<tr>
<th>A</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

Additional information

For more information about the decoding of this packet see MispredictPacket() on page K4-11624.
D5.3.37 Atom Format 1 Packet

Purpose

Indicates one Atom element.

Configurations

All.

Packet Layout

---

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A</td>
</tr>
</tbody>
</table>

Figure D5-57 Atom Format 1 Packet

Field descriptions

A

Indicates a single Atom element.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>0b0</th>
<th>One N Atom element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>One E Atom element</td>
</tr>
</tbody>
</table>

Additional information

For more information about the decoding of this packet see AtomFormat1Packet() on page K4-11625.

For more information about the generation of this packet see Atom packing on page D4-4838.
D5.3.38 Atom Format 2 Packet

Purpose
Indicates two Atom elements.

Configurations
All.

Packet Layout

```
  7  6  5  4  3  2  1  0
0 1 1 0 1 1 0  A
```

Figure D5-58 Atom Format 2 Packet

Field descriptions
A
Indicates a specific sequence of Atom elements.
The encoding for this field is POD.

| 0b00 | 1.  N Atom element. |
| 0b01 | 1.  E Atom element. |
| 0b10 | 1.  N Atom element. |
|      | 2.  E Atom element. |
| 0b11 | 1.  E Atom element. |
|      | 2.  E Atom element. |

Additional information
For more information about the decoding of this packet see AtomFormat2Packet() on page K4-11625.
For more information about the generation of this packet see Atom packing on page D4-4838.
D5.3.39 Atom Format 3 Packet

Purpose
Indicates three Atom elements.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Figure D5-59 Atom Format 3 Packet

Field descriptions
A
Indicates a specific sequence of Atom elements.
The encoding for this field is POD.

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>1. E Atom element.</td>
<td>2. E Atom element.</td>
<td>3. E Atom element.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Additional information

For more information about the decoding of this packet see `AtomFormat3Packet()` on page K4-11625.

For more information about the generation of this packet see `Atom packing` on page D4-4838.
D5.3.40 Atom Format 4 Packet

Purpose

Indicates four Atom elements.

Configurations

All.

Packet Layout

```
    7  6  5  4  3  2  1  0
  1  1  0  1  1  1 | A
```

Figure D5-60 Atom Format 4 Packet

Field descriptions

A

Indicates a specific sequence of Atom elements.
The encoding for this field is POD.

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 0b0 | 1. N Atom element.  
   | 2. E Atom element.  
   | 3. E Atom element.  
   | 4. E Atom element.  |
| 0b1 | 1. N Atom element.  
| 0b1 | 1. N Atom element.  
   | 2. E Atom element.  
   | 4. E Atom element.  |
| 0b1 | 1. E Atom element.  
   | 3. E Atom element.  

Additional information

For more information about the decoding of this packet see AtomFormat4Packet() on page K4-11626.

For more information about the generation of this packet see Atom packing on page D4-4838.
D5.3.41 Atom Format 5.1 Packet

Purpose
Indicates five Atom elements.

Configurations
All.

Packet Layout

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Figure D5-61 Atom Format 5.1 Packet

Element sequence
This packet encodes the following sequence:
1. N Atom element.
2. E Atom element.
3. E Atom element.
4. E Atom element.
5. E Atom element.

Additional information
For more information about the decoding of this packet see AtomFormat5_1Packet() on page K4-11626.
For more information about the generation of this packet see Atom packing on page D4-4838.
D5.3.42 Atom Format 5.2 Packet

Purpose
Indicates five Atom elements.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A</td>
</tr>
</tbody>
</table>

Figure D5-62 Atom Format 5.2 Packet

Field descriptions

A
Indicates a specific sequence of Atom elements.
The encoding for this field is POD.

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0b01</td>
<td>1.</td>
<td>N Atom element.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>1.</td>
<td>N Atom element.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.</td>
<td>E Atom element.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.</td>
<td>E Atom element.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>1.</td>
<td>E Atom element.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.</td>
<td>E Atom element.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.</td>
<td>E Atom element.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Additional information

For more information about the decoding of this packet see AtomFormat5_2Packet() on page K4-11626.

For more information about the generation of this packet see Atom packing on page D4-4838.
**D5.3.43 Atom Format 6 Packet**

**Purpose**
Indicates 3-23 E Atom elements, plus a subsequent E Atom or N Atom element.

**Configurations**
All.

**Packet Layout**

<table>
<thead>
<tr>
<th>COUNT</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>One E Atom element</td>
</tr>
<tr>
<td>0b1</td>
<td>One N Atom element</td>
</tr>
</tbody>
</table>

**Field descriptions**

- **A**
  Indicates an E Atom element or N Atom element, after the E Atom elements indicated by COUNT. The encoding for this field is POD.

- **COUNT**
  Indicates a number of E Atom elements. The number is 3 + COUNT. Permitted values of COUNT are 0b00000 to 0b10100. The encoding for this field is POD.

**Additional information**

For more information about the decoding of this packet see AtomFormat6Packet() on page K4-11627.

For more information about the generation of this packet see Atom packing on page D4-4838.
D5.3.44 Target Address Short IS0 Packet

Purpose
Indicates a Target Address element.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C0</td>
<td>A[8:2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[16:9]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure D5-64 Target Address Short IS0 Packet

Field descriptions

A
Instruction virtual address.
Instruction virtual address bits[1:0] always have the value 0b00.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

C0
Continuation Bit.
The encoding for this field is Unary code.

<table>
<thead>
<tr>
<th>0b0</th>
<th>Last byte in this section.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>At least one more byte follows in this section.</td>
</tr>
</tbody>
</table>

Element sequence

This packet encodes the following sequence:

1. Target Address element.

Additional information

For more information about the decoding of this packet see Parse_TargetAddressPacket() on page K4-11631.
For more information about the generation of this packet see Address compression on page D4-4839.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.45 Target Address Short IS1 Packet

Purpose
Indicates a Target Address element.

Configurations
All.

Packet Layout

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C0</td>
<td>A[7:1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[15:8]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure D5-65 Target Address Short IS1 Packet

Field descriptions

A
Instruction virtual address.
Instruction virtual address bit[0] always has the value 0b0.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

C0
Continuation Bit.
The encoding for this field is Unary code.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Last byte in this section.</td>
</tr>
<tr>
<td>0b1</td>
<td>At least one more byte follows in this section.</td>
</tr>
</tbody>
</table>

Element sequence
This packet encodes the following sequence:

1. Target Address element.

Additional information
For more information about the decoding of this packet see Parse_TargetAddressPacket() on page K4-11631.
For more information about the generation of this packet see Address compression on page D4-4839.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.46 Target Address 32-bit IS0 Packet

**Purpose**
Indicates a Target Address element.

**Configurations**
All.

**Packet Layout**

```
  7  6  5  4  3  2  1  0
  1  0  1  1  0  1  0
   (0)   A[8:2]
     (0)   A[15:9]
          A[23:16]
              A[31:24]
```

Figure D5-66 Target Address 32-bit IS0 Packet

**Field descriptions**

A
Instruction virtual address.
Instruction virtual address bits[1:0] always have the value 0b00.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

**Element sequence**
This packet encodes the following sequence:
1. Target Address element.

**Additional information**
For more information about the decoding of this packet see Parse_TargetAddressPacket() on page K4-11631.
For more information about the generation of this packet see Address compression on page D4-4839.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
Target Address 32-bit IS1 Packet

Purpose

Indicates a Target Address element.

Configurations

All.

Packet Layout

![Field descriptions]

A

Instruction virtual address.
Instruction virtual address bit[0] always has the value 0b0.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. Target Address element.

Additional information

For more information about the decoding of this packet see Parse_TargetAddressPacket() on page K4-11631.
For more information about the generation of this packet see Address compression on page D4-4839.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.48 Target Address 64-bit IS0 Packet

Purpose
Indicates a Target Address element.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(0)</td>
<td>A[8:2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0)</td>
<td>A[15:9]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[23:16]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[31:24]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[39:32]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[47:40]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[55:48]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[63:56]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure D5-68 Target Address 64-bit IS0 Packet

Field descriptions

A

Instruction virtual address.
Instruction virtual address bits[1:0] always have the value 0b00.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. Target Address element.

Additional information

For more information about the decoding of this packet see Parse_TargetAddressPacket() on page K4-11631.
For more information about the generation of this packet see Address compression on page D4-4839.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.49 Target Address 64-bit IS1 Packet

Purpose
Indicates a Target Address element.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(0)</td>
<td>A[7:1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[15:8]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[23:16]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[31:24]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[39:32]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[47:40]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[55:48]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[63:56]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure D5-69 Target Address 64-bit IS1 Packet

Field descriptions
A

Instruction virtual address.
Instruction virtual address bit[0] always has the value 0b0.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

Element sequence
This packet encodes the following sequence:
1. Target Address element.

Additional information
For more information about the decoding of this packet see Parse_TargetAddressPacket() on page K4-11631.
For more information about the generation of this packet see Address compression on page D4-4839.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.50 Target Address Exact Match Packet

Purpose
Indicates a Target Address element.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>QE</td>
<td></td>
</tr>
</tbody>
</table>

Figure D5-70 Target Address Exact Match Packet

Field descriptions
QE
Instruction virtual address.
The encoding for this field is POD.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>The address is the same as address history buffer entry 0.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b01</td>
<td>The address is the same as address history buffer entry 1.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>The address is the same as address history buffer entry 2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Element sequence
This packet encodes the following sequence:
1. Target Address element.

Additional information
For more information about the decoding of this packet see Parse_TargetAddressPacket() on page K4-11631.
For more information about the generation of this packet see Address compression on page D4-4839.
D5.3.51 Context Same Packet

Purpose
Indicates a Context element.

Configurations
All.

Packet Layout

```
   7 6 5 4 3 2 1 0
```
```
1 0 0 0 0 0 0 0
```

Figure D5-71 Context Same Packet

Element sequence
This packet encodes the following sequence:

Additional information
For more information about the decoding of this packet see Parse_ContextBytes() on page K4-11633.
D5.3.52  Context Packet

Purpose

Indicates a Context element.

Configurations

All.

Packet Layout - Variant 1

```
01234567 1 0 0 0 0 0 0 1
 0 0 NS SF (0) (0) EL
```

Figure D5-72 Context Packet (1)

Packet Layout - Variant 2

```
01234567 1 0 0 0 0 0 0 1
 1 0 NS SF (0) (0) EL

CONTEXTID[7:0]
CONTEXTID[15:8]
CONTEXTID[23:16]
CONTEXTID[31:24]
```

Figure D5-73 Context Packet (2)

Packet Layout - Variant 3

```
01234567 1 0 0 0 0 0 0 1
 0 1 NS SF (0) (0) EL

VMID[7:0]
VMID[15:8]
VMID[23:16]
VMID[31:24]
```

Figure D5-74 Context Packet (3)
Packet Layout - Variant 4

![Figure D5-75 Context Packet (4)](image)

Field descriptions

**CONTEXTID**

Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See [Context identifier tracing on page D4-4835](#).

**EL**

Exception level.

The encoding for this field is POD.

<table>
<thead>
<tr>
<th>EL</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>EL0</td>
</tr>
<tr>
<td>0b1</td>
<td>EL1</td>
</tr>
<tr>
<td>0b10</td>
<td>EL2</td>
</tr>
<tr>
<td>0b11</td>
<td>EL3</td>
</tr>
</tbody>
</table>

**NS**

Security state.

The encoding for this field is POD.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The PE is in Secure state.</td>
</tr>
<tr>
<td>0b1</td>
<td>The PE is in Non-secure state.</td>
</tr>
</tbody>
</table>
**SF**

AArch64 state.

The encoding for this field is POD.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The PE is in AArch32 state.</td>
</tr>
<tr>
<td>0b1</td>
<td>The PE is in AArch64 state.</td>
</tr>
</tbody>
</table>

**VMID**

Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See [Virtual context identifier tracing on page D4-4837](#).

**Element sequence**

This packet encodes the following sequence:

1. **Context element.**

**Additional information**

For more information about the decoding of this packet see [Parse_ContextBytes() on page K4-11633](#).
D5.3.53 Target Address with Context 32-bit IS0 Packet

Purpose

Indicates a Target Address element and a Context element.

Configurations

All.

Packet Layout - Variant 1

Figure D5-76 Target Address with Context 32-bit IS0 Packet (1)

Packet Layout - Variant 2

Figure D5-77 Target Address with Context 32-bit IS0 Packet (2)
### Field descriptions

**A**

Instruction virtual address.

Instruction virtual address bits[1:0] always have the value 0b00.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

---

#### Figure D5-78 Target Address with Context 32-bit IS0 Packet (3)

#### Packet Layout - Variant 3

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(0)</td>
<td>A[8:2]</td>
<td>(0)</td>
<td>A[15:9]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>NS</td>
<td>SF</td>
<td>(0)</td>
<td>(0)</td>
<td>EL</td>
<td></td>
</tr>
</tbody>
</table>

#### Figure D5-79 Target Address with Context 32-bit IS0 Packet (4)

#### Packet Layout - Variant 4

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(0)</td>
<td>A[8:2]</td>
<td>(0)</td>
<td>A[15:9]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>NS</td>
<td>SF</td>
<td>(0)</td>
<td>(0)</td>
<td>EL</td>
<td></td>
</tr>
</tbody>
</table>
CONTEXTID

Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

• This field is not traced.
• This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing on page D4-4835.

EL

Exception level at this address.

The encoding for this field is POD.

<table>
<thead>
<tr>
<th>0b0</th>
<th>EL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>EL1</td>
</tr>
<tr>
<td>0b10</td>
<td>EL2</td>
</tr>
<tr>
<td>0b11</td>
<td>EL3</td>
</tr>
</tbody>
</table>

NS

Security state.

The encoding for this field is POD.

<table>
<thead>
<tr>
<th>0b0</th>
<th>The PE is in Secure state.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>The PE is in Non-secure state.</td>
</tr>
</tbody>
</table>

SF

AArch64 state.

The encoding for this field is POD.

<table>
<thead>
<tr>
<th>0b0</th>
<th>The PE is in AArch32 state.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>The PE is in AArch64 state.</td>
</tr>
</tbody>
</table>

VMID

Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

• This field is not traced.
• This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing on page D4-4837.
Element sequence
This packet encodes the following sequence:
1. Target Address element.
2. Context element.

Additional information
For more information about the decoding of this packet see Parse_TargetAddressPacket() on page K4-11631.
For more information about the generation of this packet see Address compression on page D4-4839.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.54 Target Address with Context 32-bit IS1 Packet

Purpose
Indicates a Target Address element and a Context element.

Configurations
All.

Packet Layout - Variant 1

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>(0)</td>
<td>A[7:1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[15:8]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[23:16]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[31:24]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>NS</td>
<td>SF</td>
<td>(0)</td>
<td>(0)</td>
<td>EL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure D5-80 Target Address with Context 32-bit IS1 Packet (1)

Packet Layout - Variant 2

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>(0)</td>
<td>A[7:1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[15:8]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[23:16]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A[31:24]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>NS</td>
<td>SF</td>
<td>(0)</td>
<td>(0)</td>
<td>EL</td>
<td>CONTEXTID[7:0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CONTEXTID[15:8]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CONTEXTID[23:16]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CONTEXTID[31:24]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure D5-81 Target Address with Context 32-bit IS1 Packet (2)
Packet Layout - Variant 3

Figure D5-82 Target Address with Context 32-bit IS1 Packet (3)

Packet Layout - Variant 4

Figure D5-83 Target Address with Context 32-bit IS1 Packet (4)

Field descriptions

A

Instruction virtual address.

Instruction virtual address bit[0] always has the value 0b0.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.
CONTEXTID

Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

• This field is not traced.
• This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing on page D4-4835.

EL

Exception level at this address.

The encoding for this field is POD.

<table>
<thead>
<tr>
<th>EL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>EL0</td>
</tr>
<tr>
<td>0b1</td>
<td>EL1</td>
</tr>
<tr>
<td>0b10</td>
<td>EL2</td>
</tr>
<tr>
<td>0b11</td>
<td>EL3</td>
</tr>
</tbody>
</table>

NS

Security state.

The encoding for this field is POD.

<table>
<thead>
<tr>
<th>NS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The PE is in Secure state.</td>
</tr>
<tr>
<td>0b1</td>
<td>The PE is in Non-secure state.</td>
</tr>
</tbody>
</table>

SF

AArch64 state.

The encoding for this field is POD.

<table>
<thead>
<tr>
<th>SF</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The PE is in AArch32 state.</td>
</tr>
<tr>
<td>0b1</td>
<td>The PE is in AArch64 state.</td>
</tr>
</tbody>
</table>

VMID

Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

• This field is not traced.
• This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing on page D4-4837.
Element sequence
This packet encodes the following sequence:

1. Target Address element.
2. Context element.

Additional information
For more information about the decoding of this packet see Parse_TargetAddressPacket() on page K4-11631.
For more information about the generation of this packet see Address compression on page D4-4839.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
### D5.3.55  Target Address with Context 64-bit IS0 Packet

**Purpose**

Indicates a Target Address element and a Context element.

**Configurations**

All.

**Packet Layout - Variant 1**

```
+---+---+---+---+---+---+---+
| 1 | 0 | 0 | 0 | 0 | 1 | 0 |
+---+---+---+---+---+---+---+
| (0) | A[8:2] |
| (0) | A[15:9] |
|     | A[23:16] |
|     | A[31:24] |
|     | A[39:32] |
|     | A[47:40] |
|     | A[55:48] |
|     | A[63:56] |
+---+---+---+---+---+---+---+

0 0 NS SF (0) (0) EL
```

Figure D5-84 Target Address with Context 64-bit IS0 Packet (1)

**Packet Layout - Variant 2**

```
+---+---+---+---+---+---+---+
| 1 | 0 | 0 | 0 | 0 | 1 | 0 |
+---+---+---+---+---+---+---+
| (0) | A[8:2] |
| (0) | A[15:9] |
|     | A[23:16] |
|     | A[31:24] |
|     | A[39:32] |
|     | A[47:40] |
|     | A[55:48] |
|     | A[63:56] |
+---+---+---+---+---+---+---+

1 0 NS SF (0) (0) EL
```

```
CONTEXTID[7:0]
CONTEXTID[15:8]
CONTEXTID[23:16]
CONTEXTID[31:24]
```

Figure D5-85 Target Address with Context 64-bit IS0 Packet (2)
Packet Layout - Variant 3

![Diagram of Packet Layout - Variant 3]

Figure D5-86 Target Address with Context 64-bit IS0 Packet (3)

Packet Layout - Variant 4

![Diagram of Packet Layout - Variant 4]

Figure D5-87 Target Address with Context 64-bit IS0 Packet (4)
Field descriptions

A
Instruction virtual address. Instruction virtual address bits[1:0] always have the value 0b00. The address is compressed relative to address history buffer entry 0. The encoding for this field is Bit replacement.

CONTEXTID
Context identifier. When this field is not output, the Context identifier is the same as the most recently output Context identifier. If Context identifier tracing is disabled, then one of the following must occur:
• This field is not traced.
• This field contains a value of zero. The encoding for this field is POD.
See Context identifier tracing on page D4-4835.

EL
Exception level at this address. The encoding for this field is POD.

<table>
<thead>
<tr>
<th>EL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>EL0</td>
</tr>
<tr>
<td>0b1</td>
<td>EL1</td>
</tr>
<tr>
<td>0b10</td>
<td>EL2</td>
</tr>
<tr>
<td>0b11</td>
<td>EL3</td>
</tr>
</tbody>
</table>

NS
Security state. The encoding for this field is POD.

<table>
<thead>
<tr>
<th>NS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The PE is in Secure state.</td>
</tr>
<tr>
<td>0b1</td>
<td>The PE is in Non-secure state.</td>
</tr>
</tbody>
</table>

SF
AArch64 state. The encoding for this field is POD.

<table>
<thead>
<tr>
<th>SF</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The PE is in AArch32 state.</td>
</tr>
<tr>
<td>0b1</td>
<td>The PE is in AArch64 state.</td>
</tr>
</tbody>
</table>

VMID
Virtual context identifier. When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.
If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing on page D4-4837.

**Element sequence**

This packet encodes the following sequence:

1. Target Address element.
2. Context element.

**Additional information**

For more information about the decoding of this packet see Parse_TargetAddressPacket() on page K4-11631.

For more information about the generation of this packet see Address compression on page D4-4839.

For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.56 Target Address with Context 64-bit IS1 Packet

Purpose

Indicates a Target Address element and a Context element.

Configurations

All.

Figure D5-88 Target Address with Context 64-bit IS1 Packet (1)

Packet Layout - Variant 2

Figure D5-89 Target Address with Context 64-bit IS1 Packet (2)
Packet Layout - Variant 3

![Figure D5-90 Target Address with Context 64-bit IS1 Packet (3)]

Packet Layout - Variant 4

![Figure D5-91 Target Address with Context 64-bit IS1 Packet (4)]
Field descriptions

A
Instruction virtual address.
Instruction virtual address bit[0] always has the value 0b0.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

CONTEXTID
Context identifier.
When this field is not output, the Context identifier is the same as the most recently output Context identifier.
If Context identifier tracing is disabled, then one of the following must occur:
• This field is not traced.
• This field contains a value of zero.
The encoding for this field is POD.
See Context identifier tracing on page D4-4835.

EL
Exception level at this address.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>EL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>EL0</td>
</tr>
<tr>
<td>0b1</td>
<td>EL1</td>
</tr>
<tr>
<td>0b10</td>
<td>EL2</td>
</tr>
<tr>
<td>0b11</td>
<td>EL3</td>
</tr>
</tbody>
</table>

NS
Security state.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>NS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The PE is in Secure state.</td>
</tr>
<tr>
<td>0b1</td>
<td>The PE is in Non-secure state.</td>
</tr>
</tbody>
</table>

SF
AArch64 state.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>SF</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The PE is in AArch32 state.</td>
</tr>
<tr>
<td>0b1</td>
<td>The PE is in AArch64 state.</td>
</tr>
</tbody>
</table>

VMID
Virtual context identifier.
When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.
If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing on page D4-4837.

**Element sequence**

This packet encodes the following sequence:

1. **Target Address element.**
2. **Context element.**

**Additional information**

For more information about the decoding of this packet see Parse_TargetAddressPacket() on page K4-11631.

For more information about the generation of this packet see Address compression on page D4-4839.

For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.57 Source Address Short IS0 Packet

Purpose
Indicates the source address of a P0 instruction, and that the instruction was taken.

Configurations
All.

Packet Layout

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Figure D5-92 Source Address Short IS0 Packet

Field descriptions

A
Instruction virtual address.
Instruction virtual address bits[1:0] always have the value 0b00.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

C0
Continuation Bit.
The encoding for this field is Unary code.

Element sequence
This packet encodes the following sequence:
1. Source Address element.

Additional information
For more information about the decoding of this packet see Parse_SourceAddressPacket() on page K4-11628.
For more information about the generation of this packet see Address compression on page D4-4839.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.58 Source Address Short IS1 Packet

Purpose
Indicates the source address of a P0 instruction, and that the instruction was taken.

Configurations
All.

Packet Layout

![Packet Layout Diagram]

Field descriptions

A
Instruction virtual address.
Instruction virtual address bit[0] always has the value 0b0.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

C0
Continuation Bit.
The encoding for this field is Unary code.

Element sequence
This packet encodes the following sequence:
1. Source Address element.

Additional information
For more information about the decoding of this packet see Parse_SourceAddressPacket() on page K4-11628.
For more information about the generation of this packet see Address compression on page D4-4839.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.59 Source Address 32-bit IS0 Packet

Purpose
Indicates the source address of a P0 instruction, and that the instruction was taken.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

A[8:2]
A[15:9]
A[23:16]
A[31:24]

Figure D5-94 Source Address 32-bit IS0 Packet

Field descriptions
A
Instruction virtual address.
Instruction virtual address bits[1:0] always have the value 0b00.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

Element sequence
This packet encodes the following sequence:
1. Source Address element.

Additional information
For more information about the decoding of this packet see Parse_SourceAddressPacket() on page K4-11628.
For more information about the generation of this packet see Address compression on page D4-4839.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.60   Source Address 32-bit IS1 Packet

Purpose
Indicates the source address of a P0 instruction, and that the instruction was taken.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th>Field descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>Instruction virtual address.</td>
</tr>
<tr>
<td>Instruction virtual address bit[0] always has the value 0b0.</td>
</tr>
<tr>
<td>The address is compressed relative to address history buffer entry 0.</td>
</tr>
<tr>
<td>The encoding for this field is Bit replacement.</td>
</tr>
</tbody>
</table>

Element sequence
This packet encodes the following sequence:
1. Source Address element.

Additional information
For more information about the decoding of this packet see Parse_SourceAddressPacket() on page K4-11628.
For more information about the generation of this packet see Address compression on page D4-4839.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
## D5.3.61 Source Address 64-bit IS0 Packet

### Purpose
Indicates the source address of a P0 instruction, and that the instruction was taken.

### Configurations
All.

### Packet Layout

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

A

Instruction virtual address.

Instruction virtual address bits[1:0] always have the value 0b00.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

### Field descriptions

A

Instruction virtual address.

Instruction virtual address bits[1:0] always have the value 0b00.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

### Element sequence

This packet encodes the following sequence:

1. Source Address element.

### Additional information

For more information about the decoding of this packet see Parse_SourceAddressPacket() on page K4-11628.

For more information about the generation of this packet see Address compression on page D4-4839.

For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.62 Source Address 64-bit IS1 Packet

Purpose
Indicates the source address of a P0 instruction, and that the instruction was taken.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(0) A[7:1]
A[15:8]
A[23:16]
A[31:24]
A[39:32]
A[47:40]
A[55:48]
A[63:56]

Figure D5-97 Source Address 64-bit IS1 Packet

Field descriptions
A
Instruction virtual address.
Instruction virtual address bit[0] always has the value 0b0.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

Element sequence
This packet encodes the following sequence:
1. Source Address element.

Additional information
For more information about the decoding of this packet see Parse_SourceAddressPacket() on page K4-11628.
For more information about the generation of this packet see Address compression on page D4-4839.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.63 Source Address Exact Match Packet

Purpose

Indicates the source address of a P0 instruction, and that the instruction was taken.

Configurations

All.

Packet Layout

![Packet Layout Diagram]

Figure D5-98 Source Address Exact Match Packet

Field descriptions

QE

Instruction virtual address.
The encoding for this field is POD.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>The address is the same as address history buffer entry 0.</td>
</tr>
<tr>
<td>0b01</td>
<td>The address is the same as address history buffer entry 1.</td>
</tr>
<tr>
<td>0b10</td>
<td>The address is the same as address history buffer entry 2.</td>
</tr>
</tbody>
</table>

Element sequence

This packet encodes the following sequence:

1. Source Address element.

Additional information

For more information about the decoding of this packet see Parse_SourceAddressPacket() on page K4-11628.

For more information about the generation of this packet see Address compression on page D4-4839.
D5.3.64 Ignore Packet

Purpose
To align packet boundary to memory boundary.

Configurations
All.

Packet Layout

```
  7 6 5 4 3 2 1 0
0 1 1 1 0 0 0 0
```

Figure D5-99 Ignore Packet
D5.3.65 Event Packet

Purpose
Indicates 1-4 Event elements.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>V3</td>
<td>V2</td>
<td>V1</td>
<td>V0</td>
</tr>
</tbody>
</table>

Figure D5-100 Event Packet

Field descriptions

V0
Event 0 indicator.
The encoding for this field is POD.

| 0b0 | ETEEvent 0 did not occur |
| 0b1 | ETEEvent 0 occurred |

V1
Event 1 indicator.
The encoding for this field is POD.

| 0b0 | ETEEvent 1 did not occur |
| 0b1 | ETEEvent 1 occurred |

V2
Event 2 indicator.
The encoding for this field is POD.

| 0b0 | ETEEvent 2 did not occur |
| 0b1 | ETEEvent 2 occurred |

V3
Event 3 indicator.
The encoding for this field is POD.

| 0b0 | ETEEvent 3 did not occur |
| 0b1 | ETEEvent 3 occurred |
Additional information

For more information about the decoding of this packet see EventTracingPacket() on page K4-11637.

Note

[V3, V2, V1, V0] ≠ 0b0000 as this is decoded as an Ignore Packet.
D5.3.66 Q Packet

Purpose
Indicates that some instructions have executed, without a count of the number of instructions.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure D5-101 Q Packet

Element sequence
This packet encodes the following sequence:
1. Q element.

Additional information
For more information about the decoding of this packet see Parse_QPacket() on page K4-11627.
D5.3.67 Q with count Packet

Purpose
Indicates that some instructions have executed.

Configurations
All.

Packet Layout

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C0</td>
<td>COUNT[6:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>COUNT[13:7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>COUNT[20:14]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>COUNT[27:21]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>COUNT[31:28]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure D5-102 Q with count Packet

Field descriptions

**C0**
Continuation Bit.
The encoding for this field is Unary code.

| 0b0 | Last byte in this section. |
| 0b1 | At least one more byte follows in this section. |

**COUNT**
The number of instructions executed.
The encoding for this field is unsigned LE128n.

Element sequence
This packet encodes the following sequence:

1. Q element.

Additional information
For more information about the decoding of this packet see *Parse_QPacket()* on page K4-11627.
D5.3.68  Q with Exact match address Packet

Purpose

Indicates that some instructions have executed with an address of the next instruction.

Configurations

All.

Packet Layout

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TYPE</td>
</tr>
<tr>
<td>C0</td>
<td>COUNT[6:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>COUNT[13:7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>COUNT[20:14]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>COUNT[27:21]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0) (0) (0)</td>
<td>COUNT[31:28]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure D5-103 Q with Exact match address Packet

Field descriptions

C0

Continuation Bit.

The encoding for this field is Unary code.

| 0b0 | Last byte in this section. |
| 0b1 | At least one more byte follows in this section. |

COUNT

The number of instructions executed.

The encoding for this field is unsigned LE128n.

TYPE

The TYPE field indicates what form the rest of the Packet takes.

The encoding for this field is POD.

| 0b00 | A packet with this TYPE value also indicates a Target Address element with an address the same as address history buffer entry 0. |
| 0b01 | A packet with this TYPE value also indicates a Target Address element with an address the same as address history buffer entry 1. |
| 0b10 | A packet with this TYPE value also indicates a Target Address element with an address the same as address history buffer entry 2. |
| 0b11 | RESERVED |
Element sequence

This packet encodes the following sequence:

1. *Q element.*
2. *Target Address element.*

Additional information

For more information about the decoding of this packet see *Parse_QPacket()* on page K4-11627.

For more information about the generation of this packet see *Address compression* on page D4-4839.
D5.3.69 Q short address IS0 Packet

Purpose
Indicates that some instructions have executed with an address of the next instruction.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th>Field Descriptions</th>
</tr>
</thead>
</table>

A
Instruction virtual address.
Instruction virtual address bits[1:0] always have the value 0b00.
The address is compressed relative to address history buffer entry 0.
The encoding for this field is Bit replacement.

C0
Continuation Bit.
The encoding for this field is Unary code.

COUNT
The number of instructions executed.
The encoding for this field is unsigned LE128n.
Element sequence

This packet encodes the following sequence:

1. Q element.
2. Target Address element.

Additional information

For more information about the decoding of this packet see Parse_QPacket() on page K4-11627.
For more information about the generation of this packet see Address compression on page D4-4839.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.70 Q short address IS1 Packet

Purpose
Indicates that some instructions have executed with an address of the next instruction.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th>Field descriptions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Instruction virtual address. Instruction virtual address bit[0] always has the value 0b0. The address is compressed relative to address history buffer entry 0. The encoding for this field is Bit replacement.</td>
</tr>
<tr>
<td>C0</td>
<td>Continuation Bit. The encoding for this field is Unary code.</td>
</tr>
<tr>
<td>C1</td>
<td>Continuation Bit. The encoding for this field is Unary code.</td>
</tr>
<tr>
<td>COUNT</td>
<td>The number of instructions executed. The encoding for this field is unsigned LE128n.</td>
</tr>
</tbody>
</table>

Figure D5-105 Q short address IS1 Packet
**Element sequence**

This packet encodes the following sequence:

1. Q element.
2. Target Address element.

**Additional information**

For more information about the decoding of this packet see `Parse_QPacket()` on page K4-11627.

For more information about the generation of this packet see `Address compression` on page D4-4839.

For more information about the encoding of this packet see `Instruction set encoding` on page D5-4874.
D5.3.71  Q 32-bit address IS0 Packet

Purpose
Indicates that some instructions have executed with an address of the next instruction.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Instruction virtual address. Instruction virtual address bits[1:0] always have the value 0b00. The address is compressed relative to address history buffer entry 0. The encoding for this field is Bit replacement.</td>
</tr>
<tr>
<td>C0</td>
<td>Continuation Bit. The encoding for this field is Unary code.</td>
</tr>
<tr>
<td>COUNT</td>
<td>The number of instructions executed. The encoding for this field is unsigned LE128n.</td>
</tr>
</tbody>
</table>

Element sequence
This packet encodes the following sequence:
1. Q element.
2. Target Address element.

Additional information
For more information about the decoding of this packet see Parse_QPacket() on page K4-11627.
For more information about the encoding of this packet see Instruction set encoding on page D5-4874.
D5.3.72 Q 32-bit address IS1 Packet

Purpose
Indicates that some instructions have executed with an address of the next instruction.

Configurations
All.

Packet Layout

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Instruction virtual address. Instruction virtual address bit[0] always has the value 0b0. The address is compressed relative to address history buffer entry 0. The encoding for this field is Bit replacement.</td>
</tr>
<tr>
<td>C0</td>
<td>Continuation Bit. The encoding for this field is Unary code.</td>
</tr>
<tr>
<td>COUNT</td>
<td>The number of instructions executed. The encoding for this field is unsigned LE128n.</td>
</tr>
</tbody>
</table>

Field descriptions

A
Instruction virtual address. Instruction virtual address bit[0] always has the value 0b0. The address is compressed relative to address history buffer entry 0. The encoding for this field is Bit replacement.

C0
Continuation Bit. The encoding for this field is Unary code.

COUNT
The number of instructions executed. The encoding for this field is unsigned LE128n.

Element sequence
This packet encodes the following sequence:
1. Q element.

Figure D5-107 Q 32-bit address IS1 Packet
2. **Target Address element.**

**Additional information**

For more information about the decoding of this packet see `Parse_QPacket()` on page K4-11627.

For more information about the encoding of this packet see `Instruction set encoding` on page D5-4874.
ETE Protocol Descriptions
D5.3 Alphabetical list of ETE packets
Chapter D6
The Trace Buffer Extension

This chapter describes the Trace Buffer Extension (TRBE). It contains the following sections:

- About the Trace Buffer Extension on page D6-5004.
- The trace buffer on page D6-5006.
- Trace buffer management on page D6-5018.
- Synchronization and the Trace Buffer Unit on page D6-5026.
- Trace synchronization and memory barriers on page D6-5031.
- Trace of Speculative execution on page D6-5032.
- Trace in Debug state on page D6-5033.
- Detailed synchronization litmus tests on page D6-5034.
- UNPREDICTABLE behavior on page D6-5038.
D6.1 About the Trace Buffer Extension

The Trace Buffer Extension feature is identified as FEAT_TRBE.

When FEAT_TRBE is implemented, the PE includes a Trace Buffer Unit. There is one logical Trace Buffer Unit for each PE in the processor.

When the Trace Buffer Unit is enabled, Program Flow Trace generated by the trace unit is written directly to memory by the Trace Buffer Unit, rather than routing it to a trace fabric. Figure D6-1 shows this.

![Figure D6-1 Logical organization of an Armv9-A PE including a trace unit and a Trace Buffer Unit](image)

In this figure:

- **EIS** is an internal representation of the executed instruction stream.
- The trace unit converts the EIS into formatted trace data.
- **Sink** is described by the section Trace Buffer Unit disabled on page D6-5013.

To allow use by self-hosted software in a platform OS environment, a Trace Buffer Unit or ETR must support a trace buffer that is mapped to a set of non-contiguous physical blocks in memory. The Trace Buffer Unit achieves this by using the PE VMSA-based MMU.

This means that all of the following apply:

- The trace buffer is normally virtually addressed.
- The trace buffer has an owning Exception level and owning Security state that define the translation regime the trace buffer uses.
- FEAT_TRBE provides a synchronization instruction, TSB CSYNC, that is used with a DSB operation to flush trace to the trace buffer.
- For FEAT_TRBE, trace is implicitly prohibited when the owning translation regime is not in context. That is, trace is prohibited if executing at a higher Exception level than the owning Exception level, or not executing in the owning Security state. This is in addition to the behavior in FEAT_ETE and FEAT_TRF for when trace is prohibited.

FEAT_TRBE also allows the trace buffer to be defined using physical addresses. This allows the Trace Buffer Unit to be used for debugging software that changes the virtual address mappings. In this configuration, the buffer must be contiguously mapped in physical memory.
The extent of the trace buffer is defined by a Base pointer and a Limit pointer. The Base pointer and Limit pointer are aligned to the smallest implemented translation granule, meaning that a buffer must be at least one full virtual page.

The Trace Buffer Unit supports the following operational modes:

**Trace buffer mode**
Controls how the Trace Buffer Unit uses the trace buffer.

**Trigger mode**
Controls how the Trace Buffer Unit reacts to a trigger condition signaled by the trace unit.

### D6.1 System events

The trace unit can be configured to react to PE events and events from the CTI. The CTI is for use by external debuggers.

As part of FEAT_TRBE and FEAT_ETE, the PMU and FEAT_ETE event sources are unified into a single event number space. Unless otherwise stated, all architecturally defined common events that can be counted by the PMU are usable as an event at the trace unit.

The following additional architecturally-defined events are provided:

- The CTI_TRIGOUT<\(n\)> events are defined to map the system events from the CTI into the PMU event number space. As well as defining these events for the trace unit, this also provides a standard mechanism for counting *external events passed to the PE*.
- The TRCEXTOUT<\(n\)> events are defined to allow the PMU to count the events that a FEAT_ETE implementation of the trace unit might generate.
- The PMU_OVFS and PMU_HOVFS events are defined to allow the trace unit to trigger directly from a PMU overflow without using the Performance Monitors overflow trigger for PMU counters accessible to EL1 and EL0, and EL2, respectively.
- The TRB_WRAP event is defined to allow the trace unit to trigger when the current write pointer reaches the end of the trace buffer and is wrapped.

### D6.1.2 Interrupts

An interrupt request is raised on a buffer management event, such as an abort or the trace buffer filling. This is the trace buffer management interrupt.

The interrupt request is passed to an interrupt controller, such as a GIC.

Arm recommends this is a PPI.

### D6.1.3 Specification

Other than where stated otherwise, this chapter describes a simple sequential model of the Trace Buffer Unit. That is, one which performs the simple loop of:

1. Collect a single byte of trace data from the trace unit.
2. If required, translate the current write pointer virtual address to a physical address.
3. If permitted, write the byte of trace data to the write address.
4. If collection is not stopped, increment the current write pointer.
5. If necessary, decrement the Trigger Counter.

Trace buffer management events are processed as part of this operation loop. Implementations conform with the architecture conform with the described behavior of the Trace Buffer Unit. Except where the architecture specifies differently, the programmer-visible behavior of an implementation that is compliant with this specification is the same as a simple sequential model. Trace appears to be written sequentially by the Trace Buffer Unit.
D6.2 The trace buffer

**R**YCHKJ If and only if all of the following are true, then the Trace Buffer Unit is enabled:

- `SelfHostedTraceEnabled()` is TRUE.
- `TRBLIMITR_EL1.E` is 1.

The pseudocode function `TraceBufferEnabled` shows this.

**R**JYXPM If the Trace Buffer Unit is not enabled, then the Trace Buffer Unit is disabled. See *Trace Buffer Unit disabled* on page D6-5013.

The pseudocode function `TraceBufferEnabled` shows this.

**I**SMTV `SelfHostedTraceEnabled()` is defined by `FEAT_TRF`.

**R**BGLHT If and only if all of the following are true, then the Trace Buffer Unit is *running*:

- The Trace Buffer Unit is enabled.
- `TRBSR_EL1.S` is 0.

The pseudocode function `TraceBufferRunning` shows this.

**R**FRHIXV If and only if all of the following are true, then *collection is stopped*:

- The Trace Buffer Unit is enabled.
- `TRBSR_EL1.S` is 1.

The pseudocode function `TraceBufferRunning` shows this.

**I**YSQZ While the Trace Buffer Unit is enabled, it collects trace data from the trace unit and does one of the following:

- *Accepts* the trace data and writes it to the trace buffer in memory.
- *Discards* the trace data. The trace data is lost.
- *Rejects* the trace data.

**R**YMYYQX When the Trace Buffer Unit is enabled and *running*, and the Trace Buffer Unit is able to accept the trace data, the Trace Buffer Unit *accepts* the trace data from the trace unit and writes it into the trace buffer.

**R**LNTVR When the Trace Buffer Unit is enabled and *running*, and the Trace Buffer Unit is not able to accept the trace data, the Trace Buffer Unit *rejects* the trace data from the trace unit. The trace data might be retained by the trace unit until the Trace Buffer Unit *accepts* the trace data.

**I**SQYCT For example, the Trace Buffer Unit might not be able to accept trace data while its internal buffers are full.

**I**TRCDR If the Trace Buffer Unit *rejects* trace data and the trace unit is not able to retain the trace data, then the trace unit discards it and enters an *Overflow* state. Details of *Overflow* state and how the trace unit recovers from *Overflow* state are defined by the trace unit. See *Trace unit behavior on a trace unit buffer overflow* on page D4-4804.

**R**YMVZL When the Trace Buffer Unit is enabled and *collection is stopped*, the Trace Buffer Unit *discards* trace data from the trace unit. The trace data is lost.

**R**PHSKP When used with a trace unit that implements `FEAT_ETE`, the Trace Buffer Unit ignores the value of the ETE `TRCTRACEIDR` register.

### D6.2.1 The trace buffer pointers

**R**WKBRT The trace buffer is defined by three trace buffer pointer addresses:

- The Base pointer.
- The Limit pointer.
- The current write pointer.

**R**FVPBS The trace buffer starts at the Base pointer and extends to the Limit pointer. The location at the Base pointer is included in the trace buffer. The location at the Limit pointer is not included in the trace buffer.
The Base pointer and Limit pointer must be aligned by software to the smallest implemented translation granule size.

For each byte of trace that the Trace Buffer Unit accepts and writes to the trace buffer at the address in the current write pointer, one of the following applies:

- If the current write pointer is not equal to the Limit pointer minus one, then the current write pointer is incremented by one.
- If the current write pointer is equal to the Limit pointer minus one, then all of the following occur:
  - The current write pointer is wrapped by setting it to the Base pointer.
  - TRBSR_EL1.WRAP is set to 1.
  - The TRB_WRAP event is generated.

The current write pointer is not incremented when collection is stopped.

The required alignment of the current write pointer is IMPLEMENTATION DEFINED.

The Trace Buffer Unit can write trace data to memory in quantized units. The behavior is as if the bytes are written sequentially.

The Base pointer is (TRBBASER_EL1.BASE << 12). Bits [11:0] of the Base pointer are zero.

The Limit pointer is (TRBLIMITR_EL1.LIMIT << 12). Bits [11:0] of the Limit pointer are zero.

The current write pointer is TRBPTR_EL1.PTR[63:0].

The Trigger Counter is TRBTRG_EL1.TRG.

### D6.2.2 Behavior when address translation enabled

If TRBLIMITR_EL1.nVM is 0, then the Base pointer, Limit pointer, and current write pointer are virtual addresses in the stage 1 translation regime of the owning translation regime.

If TRBLIMITR_EL1.nVM is 0, then the stage 1 translation process for translating virtual addresses and checking for MMU faults is identical to that for any other virtual address in the owning translation regime.

If TRBLIMITR_EL1.nVM is 0, then CMDTG means all of the following apply:

- The virtual addresses are translated to stage 1 output addresses by stage 1 translation, and checked for stage 1 MMU faults. The stage 1 output addresses are:
  - Physical address in the owning Security state if the owning translation regime has no stage 2 translation.
  - Intermediate physical addresses (IPAs) in the owning Security state if the owning translation regime has stage 2 translations.
- If stage 1 translation is enabled for the owning translation regime, the memory type and, as applicable, Cacheability, Shareability, and Device type attributes, for stage 1 output addresses are defined by the translation table entries for the virtual address being written to.
- If stage 1 translation is disabled for the owning translation regime, the memory type of the stage 1 output addresses is Device-nGnRnE, unless overridden by stage 2 controls.
- If SCTLR_ELx.C is 0 for the owning translation regime and stage 1 translation is enabled, then all accesses to Normal memory are Non-cacheable.
- TRBPTR_EL1[63:56] are ignored by address translation if the respective TBI bit is 1.

When the Trace Buffer Unit is enabled, the Trace Buffer Unit might prefetch and cache address translations for the translation regime of the owning Exception level, including when the owning Exception level is out-of-context.

RSIFRO means that, when the Trace Buffer Unit is enabled and the owning Exception level is a lower Exception level, then the Trace Buffer Unit might make memory accesses to translation table entries from the translation regime of the owning Exception level, using the settings of the System registers associated with that translation regime.
If the PE is not executing in the owning Security state, or the PE is executing at EL3 and SCR_EL3.NS does not indicate the owning Security state, then the translation regime of the owning Exception level might not be the owning translation regime.

These memory accesses might be observed by other observers, to the extent that those accesses are required to be observed as determined by the Shareability and Cacheability of those translation table entries.

This is an exception to the rules in the section Speculative memory accesses from out-of-context translation regimes on page D8-5087.

See also Context switching on page K4-11673.

D6.2.3 Behavior when address translation disabled

If TRBLIMITR_EL1.nVM is 1, the Base pointer, Limit pointer, and current write pointer are:

- Physical address in the owning Security state if the owning translation regime has no stage 2 translation.
- Intermediate physical addresses (IPAs) in the owning Security state if the owning translation regime has stage 2 translations.

These addresses are output directly by stage 1 without any address translation.

If TRBLIMITR_EL1.nVM is 1, then unless overridden by stage 2 controls, TRBMAR_EL1 defines the memory type, and, as applicable, Cacheability, Shareability, and Device type attributes, for the stage 1 output addresses.

If TRBLIMITR_EL1.nVM is 1, the values of SCTLR_ELx.{C, M} for the owning translation regime are ignored for the purposes of determining the trace buffer Cacheability attributes.

If TRBLIMITR_EL1.nVM is 1 it is possible to generate mismatched attributes for a location from within the same stage 1 translation regime, using TRBMAR_EL1.

Software must be aware of the consequences of and permitted behaviors when accessing a memory location with mismatched attributes. For more information, including a full definition of mismatched attributes and the permitted behaviors, see Mismatched memory attributes on page B2-208.

If TRBLIMITR_EL1.nVM is 1 and TRBPTR_EL1[top:PAmax()] is nonzero, a stage 1 Address Size fault is generated when the Trace Buffer Unit attempts to write to memory, and:

- If FEAT_LPA is implemented, top is 51.
- If FEAT_LPA is not implemented, top is 47.

If TRBLIMITR_EL1.nVM is 1 and TRBPTR_EL1[63:(top+1)] is nonzero when the Trace Buffer Unit attempts to write to the trace buffer, then one of the following occurs, and it is CONSTRAINED UNPREDICTABLE which:

- A stage 1 Address Size fault is generated.
- TRBPTR_EL1[63:(top+1)] are ignored and treated as zero.

The value of top is as defined by RMXRFD.

D6.2.4 Effect of stage 2 translation

If the owning translation regime has stage 2 translations, the stage 2 process of translating the stage 1 output intermediate physical addresses and attributes to a physical address and attributes, and checking for MMU faults, is identical to that for any other intermediate physical address generated by the owning translation regime.

For example:

- The intermediate physical addresses are translated to physical addresses by stage 2 translation, and checked for stage 2 MMU faults.
- The attributes from stage 1 are combined with the attributes from the stage 2 translation to generate the physical memory attributes.
- If the Effective value of HCR_EL2.DC in the owning translation regime is 1, then stage 1 translation is disabled and the memory type produced by stage 1 is Normal Non-shareable, Inner Write-Back Cacheable Read-Allocate Write-Allocate, Outer Write-Back Cacheable Read-Allocate Write-Allocate, regardless of the values of SCTLR_EL1.C and TRBMAR_EL1.
If the Effective value of HCR_EL2.MIOCNCE in the owning translation regime is 0, then for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there is no loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

### D6.2.5 Accesses to the trace buffer

**RJTVD**

Writes to the trace buffer by the Trace Buffer Unit are privileged writes within the owning translation regime.

**ITTKZQ**

For accesses made by the Trace Buffer Unit, the memory type and, as applicable, Cacheability, Shareability, and Device type attributes are determined by the translation tables or TRBMAP_EL1.

See:
- \( R_{CMDTG} \) and \( R_{GBKYK} \), if translation is enabled.
- \( R_{FKLW} \), if translation is disabled.
- \( R_{JCMKS} \) and \( R_{IZSME} \), if the owning translation regime has stage 2 translations.

**RFBKCC**

Address translations performed by the Trace Buffer Unit manage dirty state and the Access flag. This is discoverable by software from TRBIDR_EL1.F.

**RSFLWLS**

If all of the following apply, the Trace Buffer Unit can speculatively update the translation table descriptor for any Page or Block in the trace buffer before writing data to it:
- Hardware management of dirty state by the Trace Buffer Unit is implemented.
- Hardware management of dirty state is enabled for the owning translation regime.
- The write is otherwise permitted.

This includes the case where a trace buffer management event means the Trace Buffer Unit stops writing data before the Page or Block is written to.

**RBWNRF**

The access granule for writes to the trace buffer by the Trace Buffer Unit is IMPLEMENTATION DEFINED, up to a maximum of 2KB, and might vary from time to time.

**RCMNS**

Writes to any Device memory type by the Trace Buffer Unit occur once.

**RRZTDD**

A memory access from the Trace Buffer Unit that crosses a Page or Block boundary to a memory location that has a different memory type or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation performs one of the following behaviors:
- Each memory access generated by the Trace Buffer Unit uses the memory type and Shareability attribute associated with its own address.
- The access generates an Alignment fault caused by the memory type:
  - If only the stage 1 translation generated the mismatch, or there is only one stage of translation in the owning translation regime, the resulting trace buffer management event is a stage 1 Data Abort.
  - If only the stage 2 translation generated the mismatch, the resulting trace buffer management event is a stage 2 Data Abort.
  - If both stages of translation generate the mismatch, the resulting trace buffer management event is either a stage 1 Data Abort or a stage 2 Data Abort.
- The trace data is discarded and the current write pointer might not be updated.

**RXRQTN**

A memory access from the Trace Buffer Unit to Device memory that crosses a boundary corresponding to the smallest translation granule size of the implementation causes CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation performs one of the following behaviors:
- Each memory accesses generated by the Trace Buffer Unit is performed as if the boundary has no effect on the memory accesses.
- Each memory accesses generated by the Trace Buffer Unit is performed as if the boundary has no effect on the memory accesses except that there is no guarantee of ordering between it and other memory accesses.
• The access generates an Alignment fault caused by the memory type:
  — If only the stage 1 translation causes the boundary to be crossed, or there is only one stage of
    translation in the owning translation regime, the resulting trace buffer management event is a stage 1
    Data Abort.
  — If only the stage 2 translation causes the boundary to be crossed, the resulting trace buffer management
    event is a stage 2 Data Abort.
  — If both stages of translation cause the boundary to be crossed, the resulting trace buffer management
    event is either a stage 1 Data Abort or a stage 2 Data Abort.
• The trace data is discarded and the current write pointer might not be updated.

Note: The boundary is between two Device memory regions that are both:
• Of the size of the smallest implemented translation granule.
• Aligned to the size of the smallest implemented translation granule.

Although the Trace Buffer Unit behaves as if trace data is written a byte at a time, it is not required to do so.
For example, RWNRF and CMSNE mean that if the memory type for the trace buffer is Device-nGnRnE, then all
of the following apply:
• Writes are not repeated and not reordered.
• A write Completes only after it reaches its endpoint in the memory system.
• The access granule size at the endpoint in the memory system is not defined by the architecture. However, a
  specific implementation might define the granule to permit interoperability with specific devices.
The access granule is not required to be fixed. For example, the Trace Buffer Unit might output a smaller granule
when flushing trace data to the trace buffer.
See also IQQKZF.

D6.2.6 The owning translation regime

The owning translation regime is defined by the owning Security state and the owning Exception level.

When the Trace Buffer Unit is enabled, the owning Security state is:
• Non-secure state if and only if at least one of the following is true:
  — EL3 is not implemented and the PE executes in Non-secure state.
  — MDCR_EL3.NSTB is either 0b10 or 0b11.
• Secure state if and only if at least one of the following is true:
  — EL3 is not implemented and the PE executes in Secure state.
  — MDCR_EL3.NSTB is either 0b00 or 0b01.

When the Trace Buffer Unit is enabled, the owning Exception level is:
• EL1 if and only if at least one of the following is true:
  — EL2 is not implemented in the owning Security state.
  — EL2 is disabled in the owning Security state.
  — MDCR_EL2.E2TB is either 0b10 or 0b11.
• EL2 if and only if all of the following is true:
  — EL2 is implemented and enabled in the owning Security state.
  — MDCR_EL2.E2TB is 0b00.

When the Trace Buffer Unit is enabled and the owning Exception level is EL1, all of the following apply:
• The owning translation regime is EL1&0.
• If TRBLIMITR_EL1.nVM is 0, the trace buffer pointer addresses are virtual addresses in the EL1&0
  translation regime using the current ASID from TTBRx_EL1.
• If TRBLIMITR_EL1.nVM is 1, the trace buffer pointer addresses are intermediate physical addresses.
• Intermediate physical addresses (whether from the output of stage 1, or the pointers, as applicable) are subject to stage 2 translation using the current VMID if EL2 is implemented and enabled and HCR_EL2.VM is 1.

• The following are prohibited trace regions:
  — EL3.
  — EL2.
  — EL0, if EL2 is implemented and enabled and HCR_EL2.TGE is 1.

When the Trace Buffer Unit is enabled and the owning Exception level is EL2, all of the following apply:

• If HCR_EL2.E2H is 0, the owning translation regime is EL2.
• If HCR_EL2.E2H is 1, the owning translation regime is EL2&0.
• If HCR_EL2.E2H is 0 and TRBLIMITR_EL1.nVM is 0, the trace buffer pointer addresses are virtual addresses in the EL2 translation regime.
• If HCR_EL2.E2H is 1 and TRBLIMITR_EL1.nVM is 0, the trace buffer pointer addresses are virtual addresses in the EL2&0 translation regime using the current ASID from TTBRx_EL2.
• If TRBLIMITR_EL1.nVM is 1, the trace buffer pointer addresses are physical addresses.
• EL3 is a prohibited trace region.

The following table summarizes the owning translation regime.

<table>
<thead>
<tr>
<th>Enabled</th>
<th>NSTB</th>
<th>E2TB</th>
<th>EEL2</th>
<th>E2H</th>
<th>Owning translation regime</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>Disabled</td>
</tr>
<tr>
<td>TRUE</td>
<td>0b0x</td>
<td>x</td>
<td>0b0</td>
<td>x</td>
<td>Secure EL1&amp;0</td>
</tr>
<tr>
<td>TRUE</td>
<td>0b0x</td>
<td>0b00</td>
<td>0b1</td>
<td>0b0</td>
<td>Secure EL2</td>
</tr>
<tr>
<td>TRUE</td>
<td>0b0x</td>
<td>0b00</td>
<td>0b1</td>
<td>0b1</td>
<td>Secure EL2&amp;0</td>
</tr>
<tr>
<td>TRUE</td>
<td>0b0x</td>
<td>0b1x</td>
<td>0b1</td>
<td>x</td>
<td>Secure EL1&amp;0</td>
</tr>
<tr>
<td>TRUE</td>
<td>0b1x</td>
<td>0b00</td>
<td>x</td>
<td>0b0</td>
<td>Non-secure EL2</td>
</tr>
<tr>
<td>TRUE</td>
<td>0b1x</td>
<td>0b00</td>
<td>x</td>
<td>0b1</td>
<td>Non-secure EL2&amp;0</td>
</tr>
<tr>
<td>TRUE</td>
<td>0b1x</td>
<td>0b1x</td>
<td>x</td>
<td>x</td>
<td>Non-secure EL1&amp;0</td>
</tr>
</tbody>
</table>

When any of the following is true, then the translation of addresses generated by the Trace Buffer Unit is CONstrained UNPREDICTABLE:

• The owning Security state is Secure and SCR_EL3.NS is 1.
• The owning Security state is Non-secure and SCR_EL3.NS is 0.
The PE behaves as if one of the following is true for these translations:

- The owning Security state is Secure and SCR_EL3.NS is 0.
- The owning Security state is Non-secure and SCR_EL3.NS is 1.

Note: The behavior might differ within the same translation.

Secure and Non-secure translation regimes have different behaviors. Non-secure translation regimes only operate on Non-secure addresses, but a Secure stage 1 translation regime can generate both Secure and Non-secure output addresses and a Secure stage 2 translation regime can have both Secure and Non-secure input and output addresses. In addition, stage 2 translation is disabled when EL2 is disabled in the current Security state.

RRRCNN means that if software executing at EL3 changes the value of SCR_EL3.NS before ensuring all trace operations are complete, this might cause CONSTRAINED UNPREDICTABLE behaviors, including any of the following:

- The owning Security state is Non-secure but the PE can generate Secure output addresses at both stage 1 and, if applicable, stage 2.
- The owning Security state is Non-secure EL1&0 but the PE behaves as if stage 2 is disabled because, in this example, Secure EL2 is disabled.
- The owning Security state is Secure but the PE treats Secure output addresses as Non-secure addresses at both stage 1 and, if applicable, stage 2.
- The owning Security state is Secure EL1&0 but the PE behaves as if stage 2 is enabled even if Secure EL2 is disabled.

See also RNSFRO and RMRVFT.

RMFFGX means that when the Trace Buffer Unit is enabled and the owning Security state is Non-secure state, Secure state is a prohibited trace region.

RVGWJN means that when the Trace Buffer Unit is enabled and the owning Security state is Secure state, Non-secure state is a prohibited trace region.

The following table summarizes the Trace Prohibited regions, by Exception level and state, when all of the following apply:

- TraceBufferEnabled() is TRUE.
- EL3, Non-secure EL2, and Secure EL2 are all implemented.
- EL3 is using AArch64.

In this table:

- NS is the Effective value of SCR_EL3.NS.
- STE is the Effective value of MDCR_EL3.STE.
- NSTB is the Effective value of MDCR_EL3.NSTB.
- E2TB is the Effective value of MDCR_EL2.E2TB.
- EEL2 is the Effective value of SCR_EL3.EEL2.
- TGE is the Effective value of HCR_EL2.TGE.

The EL3, EL2, EL1, EL0 columns show which control, if any, enables tracing at the Exception level. In these columns:

- P means tracing is prohibited.
- E2TRE means tracing is allowed if TRFCR_EL2.E2TRE is 1 and prohibited otherwise.
E1TRE
means tracing is allowed if TRFCR_EL1.E1TRE is 1 and prohibited otherwise.

E0HTRE
means tracing is allowed if TRFCR_EL2.E0HTRE is 1 and prohibited otherwise.

E0TRE
means tracing is allowed if TRFCR_EL1.E0TRE is 1 and prohibited otherwise.

The pseudocode function TraceAllowed shows this.

<table>
<thead>
<tr>
<th>NS</th>
<th>STE</th>
<th>NSTB</th>
<th>E2TB</th>
<th>EEL2</th>
<th>TGE</th>
<th>EL3</th>
<th>EL2</th>
<th>EL1</th>
<th>EL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>x</td>
<td>0b0x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>0b1</td>
<td>x</td>
<td>0b1x</td>
<td>0b1x</td>
<td>x</td>
<td>0b0</td>
<td>P</td>
<td>P</td>
<td>E1TRE</td>
<td>E0TRE</td>
</tr>
<tr>
<td>0b1</td>
<td>x</td>
<td>0b1x</td>
<td>0b1x</td>
<td>x</td>
<td>0b1</td>
<td>P</td>
<td>P</td>
<td>n/a</td>
<td>P</td>
</tr>
<tr>
<td>0b1</td>
<td>x</td>
<td>0b1x</td>
<td>0b00</td>
<td>x</td>
<td>0b0</td>
<td>P</td>
<td>E2TRE</td>
<td>E1TRE</td>
<td>E0TRE</td>
</tr>
<tr>
<td>0b1</td>
<td>x</td>
<td>0b1x</td>
<td>0b00</td>
<td>x</td>
<td>0b1</td>
<td>P</td>
<td>E2TRE</td>
<td>n/a</td>
<td>E0HTRE</td>
</tr>
<tr>
<td>0b0</td>
<td>0b0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>0b1x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>0b0x</td>
<td>x</td>
<td>0b0</td>
<td>x</td>
<td>P</td>
<td>n/a</td>
<td>E1TRE</td>
<td>E0TRE</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>0b0x</td>
<td>0b1x</td>
<td>0b1</td>
<td>0b0</td>
<td>P</td>
<td>P</td>
<td>E1TRE</td>
<td>E0TRE</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>0b0x</td>
<td>0b1x</td>
<td>0b1</td>
<td>0b1</td>
<td>P</td>
<td>n/a</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>0b0x</td>
<td>0b00</td>
<td>0b1</td>
<td>0b0</td>
<td>P</td>
<td>E2TRE</td>
<td>E1TRE</td>
<td>E0TRE</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>0b0x</td>
<td>0b00</td>
<td>0b0</td>
<td>0b1</td>
<td>P</td>
<td>E2TRE</td>
<td>n/a</td>
<td>E0HTRE</td>
</tr>
</tbody>
</table>

When the Trace Buffer Unit is disabled, the owning translation regime, owning Security state, and owning Exception level are not defined.

D6.2.7 Trace Buffer Unit disabled

When the Trace Buffer Unit is disabled, the Trace Buffer Unit discards trace data from the trace unit.

The Trace Buffer Unit does not prefetch and cache address translations when the Trace Buffer Unit is disabled.

When the Trace Buffer Unit is disabled the trace unit might send trace data to an IMPLEMENTATION DEFINED trace bus.

The trace unit does not send trace data to the IMPLEMENTATION DEFINED trace bus when the Trace Buffer Unit is enabled.

Figure D6-1 on page D6-5004 shows this IMPLEMENTATION DEFINED trace bus as a dotted line to an external trace Sink.
Details of this bus are outside the scope of this architecture, and might require further configuration. For example, if the trace unit implements FEAT_ETE and the trace bus is AMBA ATB, the ATID value is configured through the trace unit external trace registers.

D6.2.8 Restrictions on programming the Trace Buffer Unit

A current write pointer value is out-of-range if any of the following are true:
- The current write pointer is less-than the Base pointer, treating both pointers as unsigned integers.
The current write pointer is greater-than-or-equal-to the Limit pointer, treating both pointers as unsigned integers.

- Bits [63:56] of the current write pointer are not equal to bits [63:56] of the Base pointer.
- Bits [63:56] of the current write pointer are not equal to bits [63:56] of the Limit pointer.

Note: R_MSPSD means the current write pointer is out-of-range if the Base pointer is not less-than the Limit pointer or bits [63:56] of the Base pointer are not equal to bits [63:56] of the Limit pointer.

A current write pointer or Trigger Counter value is misaligned if it is not a multiple of an IMPLEMENTATION DEFINED alignment specified by TRBIDR_EL1.Align.

A current write pointer or Trigger Counter value is a valid restart value if it was previously initialized with a value that was not out-of-range and not misaligned and later read from the applicable register when all of the following are true:

- The Trace Buffer Unit is disabled.
- All trace operations are complete. See R_NSFRQ for the definition of complete.
- No External abort has been reported to the Trace Buffer Unit. TRBSR_EL1.EA is 0.
- No write by the Trace Buffer Unit has generated an Alignment fault.
- No write by the Trace Buffer Unit has generated an asynchronous SError interrupt exception.

A current write pointer or Trigger Counter value is a fault value if it was previously initialized with a value that was not out-of-range and not misaligned or a value that was a valid restart value, and later read from the applicable register when all of the following are true:

- The Trace Buffer Unit is disabled.
- All trace operations are complete. See R_NSFRQ for the definition of complete.
- One of the following is true:
  - An External abort has been reported to the Trace Buffer Unit. TRBSR_EL1.EA is 1.
  - A write by the Trace Buffer Unit has generated an Alignment fault.
  - A write by the Trace Buffer Unit has generated an asynchronous SError interrupt exception.

A fault value is for error handling purposes only. Software must not cause the Trace Buffer Unit to become enabled and running with the current write pointer having a fault value.

If the current write pointer is written by a direct write with a misaligned value that is not a restart value and not a fault value, the value returned by a subsequent direct read of the current write pointer is UNKNOWN.
If the current write pointer has an out-of-range value, or a misaligned value that is not a restart value when the Trace Buffer Unit attempts to write to the trace buffer, then any of the following might occur:

- If the value is out-of-range, the current write pointer might be wrapped before or after the write, and the TRB_WRAP event might be generated.
- If the value is misaligned, the write might generate an Alignment fault.
- The Trace Buffer Unit might write the trace data to any address in memory that is writable by a privileged access in the owning translation regime. These addresses are:
  - Virtual addresses in the owning translation regime if TRBLIMITR_EL1.nVM is 0.
  - Intermediate physical addresses in the owning Security state if TRBLIMITR_EL1.nVM is 1 and the owning translation regime has stage 2 translations.
  - Physical addresses in the owning Security state if TRBLIMITR_EL1.nVM is 1 and the owning translation regime has no stage 2 translation.
- The write might generate a trace buffer management event with an UNKNOWN reason:
  - TRBSR_EL1.S is either set to 1 or unchanged.
  - TRBSR_EL1.WRAP is either set to 1 or unchanged.
  - TRBSR_EL1.EC is set to an UNKNOWN value.
  - TRBSR_EL1.MSS is set to an UNKNOWN value.
  - The TRB_WRAP event might be generated.

If the Trigger Counter is written by a direct write with a misaligned value that is not a restart value, then all of the following apply:

- If the value is not a fault value, the value returned by a subsequent direct read of the Trigger Counter register is UNKNOWN.
- The generation of a Trigger Event while the Trace Buffer Unit remains enabled and running is UNPREDICTABLE.

RXZHH and RCZDM mean an implementation that always keeps the current write pointer and/or Trigger Counter aligned to the IMPLEMENTATION DEFINED alignment specified by TRBIDR_EL1.Align, where TRBIDR_EL1.Align is greater-than-zero (byte alignment), can implement bits [M:0] of the applicable register(s) as RAZ/WI bits, where M is (TRBIDR_EL1.Align-1).

RHZXZZM allows an implementation where an External abort is reported to the Trace Buffer Unit and handled synchronously to implement TRBPTR_EL1[M:N], where N is IMPLEMENTATION SPECIFIC and typically determined by the minimum memory access granule, as read/write bits for the purpose of reporting an External abort fault address, but otherwise ignore the value in these bits. If N>0, bits [(N-1):0] can be implemented as RAZ/WI.

When TRBLIMITR_EL1.E is 1, the PE might ignore a direct write to any of the following registers, other than a direct write to TRBLIMITR_EL1 that modifies TRBLIMITR_EL1.E:

- The current write pointer, TRBPTR_EL1.
- The Base pointer, TRBBASER_EL1.
- The Limit pointer, TRBLIMITR_EL1.
- The Trigger Counter, TRBTRG_EL1.
- TRBSR_EL1.
- TRBMAR_EL1.

Software must use appropriate Context synchronization operations to order a direct write that modifies TRBLIMITR_EL1.E with respect to other direct writes to Trace Buffer Unit registers. This includes a write to enable the Trace Buffer Unit by setting TRBLIMITR_EL1.E to 1.

See also:

- Synchronization and the Trace Buffer Unit on page D6-5026.
- UNPREDICTABLE behavior on page D6-5038.
- Context switching on page K4-11673.
### D6.2.9 Cache and TLB operations

**IWYVXK** Translations used by the Trace Buffer Unit might be cached in a TLB.

**RGQJMC** TLB maintenance operations that affect the TLB of the PE also affect any TLB caching translations for the Trace Buffer Unit of that PE.

**RNPNNM** The PE is permitted, but not required, to cache all translations used by the Trace Buffer Unit in TLB caching structures that combine stage 1 and stage 2 of the translation. This includes when TRBLIMITR_EL1.nVM is 1 and the owning translation regime has stage 2 translations.

**SLSZDR** When TRBLIMITR_EL1.nVM is 1 and the owning translation regime has stage 2 translations, the Trace Buffer Unit uses intermediate physical addresses (IPAs). RNPNNM permits, but does not require, such translations to be cached in a TLB in such a way that an IPAS2 TLB maintenance operation is not sufficient to invalidate the cached copies. In this case, there is no VA for the translation.

If TRBLIMITR_EL1.nVM is 1 and the owning translation regime has stage 2 translations, then the following code executed at EL2 or above is sufficient to invalidate all cached copies of the stage 2 translations used by the Trace Buffer Unit of the IPA held in Xt for the current VMID:

```
TLBI IPAS2E1, Xt
DSB
TLBI VMALLE1
```

Equivalent architectural requirements apply to the IPAS2L instruction, except that the only TLB entries that must be invalidated by an IPAS2L instruction are those that come from the final level of the translation table lookup.

Equivalent sequences guaranteed to invalidate all entries invalidated by the above code sequence can be used, such as TLBI ALL or TLBI VMALLS1S2.

**RDNFWB** Cache maintenance operations that affect the caches of the PE also affect data caching by the Trace Buffer Unit of that PE.

**IMZQPT** RGQJMC and RDNFWB mean that the completion of any cache or TLB maintenance instruction includes its completion on all Trace Buffer Units for PEs that are affected by both the instruction and the DSB operation that is required to guarantee visibility of the maintenance instruction. See Detailed synchronization litmus tests on page D6-5034 for more information.

### D6.2.10 Effect on the exclusive monitors and transactions

**RDJCVN** If an operation between Load-Exclusive and Store-Exclusive instructions is traced, and the trace data is written to an unrelated address, then the write has no effect on the exclusive monitors.

**RMJDKN** If an operation inside a transaction is traced, and the trace data is written to an unrelated address, then the write has no effect on the transaction.

**RNWSKV** If the Trace Buffer Unit writes to the marked address of an exclusives monitor in the Exclusive Access state, then one of the following occurs, and it is CONSTRAINED UNPREDICTABLE which:

- The write has the same effect on the exclusives monitor as a store by the PE or any other Observer to that address.
- The write has no effect on the exclusives monitor.

**RFVRXJ** If the Trace Buffer Unit writes to the working set of a transaction, then one of the following occurs, and it is CONSTRAINED UNPREDICTABLE which:

- The write has the same effect on the transaction as a store by any other Observer to that address.
- The write has no effect on this transaction.

### D6.2.11 MPAM

**RWXXSYG** If The Memory Partitioning and Monitoring (MPAM) Extension on page A2-126 is implemented, then the MPAM information for accesses made by the Trace Buffer Unit to the trace buffer use the MPAM values of the owning Exception level and owning Security state.
For example, if the owning Exception level is EL2 the trace buffer writes use MPAM2_EL2.PARTID_D and MPAM2_EL2.PMG_D. MPAM_NS is set for the owning Security state.

**D6.2.12 MTE**

If **FEAT_MTE** is implemented, then the Trace Buffer Unit generates an Unchecked access for each access to the trace buffer.

Note: This is the case even when a Tagged Normal memory type is accessed.

See also Chapter D9 *The Memory Tagging Extension*. 
D6.3 Trace buffer management

IGYHBH FEAT_TRBE supports the following trace buffer modes:

Circular Buffer mode
In Circular Buffer mode, when the current write pointer reaches the Limit pointer, it is wrapped by setting it to the Base pointer.

Wrap mode
As Circular Buffer mode, except that an interrupt request is generated when the current write pointer is wrapped.

Fill mode
As Wrap mode, except that trace collection stops when the current write pointer is wrapped.

The trace buffer mode is controlled by TRBLIMITR_EL1.FM.

INFZKS A trace buffer management event occurs:
• On an Alignment fault or MMU fault.
• On an External abort.
• On a Trigger Event, if enabled.
• When the current write pointer is wrapped to the Base pointer and the trace buffer mode is not Circular Buffer mode. This event is known as:
  — A buffer wrap event, if the trace buffer mode is Wrap mode.
  — A buffer full event, if the trace buffer mode is Fill mode.
• On a programming error, when permitted as an UNPREDICTABLE behavior of the PE. For more information, see Restrictions on programming the Trace Buffer Unit on page D6-5013 and UNPREDICTABLE behavior on page D6-5038.
• On an IMPLEMENTATION DEFINED event.

Note: The trace buffer management event behavior differs from that of the SPE Profiling Buffer management event.

RHLKSG On a trace buffer management event, all of the following occurs:
• The interrupt request bit, TRBSR_EL1.IRQ, is set to 1.
• The trace buffer management interrupt signal, TRBIRQ, is asserted.
• Additional syndrome for the event might be written to TRBSR_EL1.MSS.

RLRTBP TRBIRQ is a level triggered interrupt request driven by TRBSR_EL1.IRQ. This means that all of the following apply:
• A direct write that sets TRBSR_EL1.IRQ to 1 causes the interrupt request to be asserted.
• The interrupt request remains asserted until software clears TRBSR_EL1.IRQ to 0.

RTPPCF When a GIC is implemented, TRBIRQ is configured as a PPI. TRBIRQ is signaled by the PE that implements the Trace Buffer Unit.

IHJFLC The PPI number is not defined by the architecture. Arm recommends that the PPI number is discoverable to an Operating System, for example using ACPI or devicetree interfaces.

SXLNJY Software must configure the trace buffer management interrupt to be taken to the correct Exception level.

IHJFLC Buffer full, Alignment fault, and MMU fault trace buffer management events are synchronous. This means that the effect of these trace buffer management events setting TRBSR_EL1.S to 1, collection is stopped, happens before any further trace is collected by the Trace Buffer Unit from the trace unit.

ILZDN When the trigger mode is Stop on Trigger, a Trigger Event initiates a trace unit flush meaning more trace might be written to the trace buffer before any Collection is stopped by the Trigger Event trace buffer management event. This additional trace might cause a second trace buffer management event to be generated before the Trigger Event trace buffer management event.
The TRBIRQ interrupt is always taken asynchronously by the PE, even if the event is reported synchronously to the Trace Buffer Unit.

Following an Alignment fault, MMU fault, or External abort trace buffer management event, TRBPTR_EL1 serves as a Fault Address Register.

For a fault or synchronous External abort trace buffer management event, the frozen TRBPTR_EL1 is the address that generated the fault or External abort.

For an asynchronous External abort trace buffer management event, the frozen TRBPTR_EL1 is not guaranteed to be the address that generated the External abort.

### D6.3.1 Prioritization of a trace buffer management event

Where multiple synchronous trace buffer management events occur on writing trace data, the PE prioritizes them from highest to lowest priority, reporting the highest priority event as follows:

1. Synchronous fault.
2. Synchronous External abort.
3. Buffer full event.
4. Buffer wrap event.

Do not confuse the prioritization of trace buffer management events with the prioritization of trace buffer management interrupts by an interrupt controller.

Asynchronous and IMPLEMENTATION DEFINED trace buffer management events are not prioritized relative to synchronous trace buffer management events.

### D6.3.2 Buffer full and buffer wrap events

If the current write pointer is wrapped to the Base pointer and the trace buffer mode is Fill mode, then all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.
- TRBSR_EL1.WRAP is set to 1.
- The TRB_WRAP event is generated.
- If TRBSR_EL1.S is 0, then all of the following occur:
  - TRBSR_EL1.S is set to 1, collection is stopped.
  - TRBSR_EL1.EC is set to 0x00, other buffer management event.
  - TRBSR_EL1.BSC is set to 0b000001, buffer filled.
- The other fields in TRBSR_EL1 are unchanged.

After the trace buffer management event, the current write pointer will point to the Base pointer.

If the current write pointer is wrapped to the Base pointer and the trace buffer mode is Wrap mode, then all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.
- TRBSR_EL1.WRAP is set to 1.
- The TRB_WRAP event is generated.

Because TRBSR_EL1.S is unchanged, trace continues to be collected and written to the trace buffer.

If the current write pointer is wrapped to the Base pointer and the trace buffer mode is Circular Buffer mode, then all of the following occur:

- TRBSR_EL1.WRAP is set to 1.
- The other fields in TRBSR_EL1 are unchanged.
- The TRB_WRAP event is generated.
If TRBSR_EL1.S is 1, the current write pointer is not updated, meaning the current write pointer is never wrapped when TRBSR_EL1.S is already 1.

Software can configure the PMU to count the TRB_WRAP event and monitor how many times the current write pointer has wrapped, particularly in Circular Buffer mode or Wrap mode.

See also Controlling generation of trace buffer management events on page K4-11675.

D6.3.3 Trigger event

The Trace Buffer Extension supports detection of a trigger condition from the trace unit. A trigger condition is typically used to stop trace capture to ensure trace is captured around a point of interest.

The trace unit defines how software programs the trace unit to generate trigger conditions.

A Detected Trigger is signaled to the Trace Buffer Unit by the trace unit when the trace unit detects a trigger condition. The trace unit defines whether the Detected Trigger is signaled synchronously or asynchronously to the trace data stream.

A Trigger Event occurs when the Trigger Counter has counted the specified number of trace bytes after a Detected Trigger. Software can set the Trigger Counter to zero to skip this step.

The Trigger Counter is a counter used to delay a Trigger Event for a specified number of trace bytes after a Detected Trigger.

Figure D6-2 shows this.

Figure D6-2 Trigger condition to Trigger Event

For a trace unit that implements FEAT_ETE, event 0 is the trigger condition.

When the trigger mode is set to Stop on trigger, software uses the Trigger Counter to control how trace is collected around the Detected Trigger as follows:

• Set the Trigger Counter to zero to trace before the Detected Trigger.
• Set the Trigger Counter to half the size of the trace buffer to trace around the Detected Trigger.
• Set the Trigger Counter to the size of the trace buffer to trace after the Detected Trigger.

Restrictions on programming the Trace Buffer Unit on page D6-5013 defines additional constraints on writing to the Trigger Counter.

If a Detected Trigger occurs while the Trace Buffer Unit is enabled, then all of the following occur:

• TRBSR_EL1.TRG is set to 1.
• The other fields in TRBSR_EL1 are unchanged.

If all of the following are true, then the Trace Buffer Unit decrements the Trigger Counter by 1 for each byte of trace data written to the trace buffer by the Trace Buffer Unit:

• The Trigger Counter is nonzero.
• TRBSR_EL1.TRG is set to 1.

If the write generates an Alignment fault, MMU fault, or External abort, it is UNPREDICTABLE whether the Trigger Counter is decremented.
If a Detected Trigger occurs when the Trigger Counter is nonzero and TRBSR_EL1.TRG is 0, the Trace Buffer Unit might decrement the Trigger Counter by an amount up to the value specified by TRBIDR_EL1.Align without writing any additional trace data to the trace buffer.

An implementation might include an internal buffer that collects bytes of trace data into more convenient units before writing them to memory. For example, the width of the system bus or the length of a cache line. In such an implementation, TRBIDR_EL1.Align specifies the size of this unit, and the Trigger Counter is decremented by the size of this unit when the write occurs, meaning the Trigger Counter is always aligned to the size specified by TRBIDR_EL1.Align.

However, this means that if the Detected Trigger occurs when such an internal buffer is not empty, the Trace Buffer Unit will over-decrement the counter when the internal buffer is written to memory. RQFGNQ permits this.

See also RBWNRf.

A Trigger Event is generated when the Trace Buffer Unit is enabled and one of the following occurs:

- A Detected Trigger occurs when the Trigger Counter is zero and TRBSR_EL1.TRG is 0.
- The Trace Buffer Unit decrements the Trigger Counter to zero.

A Trigger Event is not generated when a Detected Trigger occurs, the Trigger Counter is set to zero, and TRBSR_EL1.TRG is already 1.

A Trigger Event is not generated when the Trace Buffer Unit is disabled.

The Detected Trigger might be generated by a trace operation tT. This might be the trace operation generated by an instruction that also matched the trigger condition, or might be a trace operation generated asynchronously by the trace unit to mark the trigger condition in the trace data. The trace unit defines this relationship for triggers.

A Trigger Event might be generated by a trace operation as follows:

- If the Trigger Event is generated when a Detected Trigger occurs when the Trigger Counter is zero and TRBSR_EL1.TRG is 0, and the Detected Trigger is generated by a trace operation tT then the Trigger Event is generated by the same trace operation tT.
- If the Trigger Event is generated when the Trace Buffer Unit decrements the Trigger Counter to zero, then Trigger Event is generated by the trace operation that generated the trace data that caused the Trigger Counter to decrement to zero.

A Trigger Event is not generated by a specific trace operation if the Trigger Event is generated when a Detected Trigger occurs when the Trigger Counter is zero and TRBSR_EL1.TRG is 0, and the Detected Trigger is not generated by a specific trace operation.

The link between a Trigger Event and a trace operation that generated it affects when the trace operation is Microarchitecturally-finished and the behavior of the TSB CSYNC instruction. See RQJDJD.

FEAT_TRBE supports the following trigger modes:

**Stop on trigger**

Trace collection is stopped and an interrupt request is generated after a Trigger Event.

**IRQ on trigger**

An interrupt request is generated after a Trigger Event.

**Ignore trigger**

The Trace Buffer Unit ignores the trigger condition.

In the Stop on trigger and IRQ on trigger modes, software specifies the amount of trace that is collected after the trigger condition before the Trigger Event.

If a Trigger Event is generated when collection is not stopped and the trigger mode is Stop on trigger, then all of the following occur:

- The Trace Buffer Unit initiates a trace unit flush of the trace unit.
- The TRB_TRIG event is generated.

On completion of the trace unit flush, all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.
• If TRBSR_EL1.S is 0, then all of the following occur:
  — TRBSR_EL1.S is set to 1, \textit{collection is stopped}.
  — TRBSR_EL1.EC is set to 0x00, other buffer management event.
  — TRBSR_EL1.BSC is set to 0b000010, Trigger Event.
• The other fields in TRBSR_EL1 are unchanged.

After the trace buffer management event, the current write pointer will point to either the first byte after the last trace byte written to the trace buffer, or, if the last trace byte written to the trace buffer was the last byte in the trace buffer, the Base pointer.

The trace unit defines the behavior and completion of a trace unit flush, including which trace operations, if any, are accepted by the Trace Buffer Unit before the trace unit flush completes.

If the \textit{Detected Trigger} is generated by a trace operation \( t_T \), then the trace unit flush does not complete before the Trace Buffer Unit accepts the trace data for \( t_T \).

Because the Trace Buffer Unit initiates a trace unit flush before stopping this means that, before TRBSR_EL1.S is set to 1:
• More trace might be written to the trace buffer after the \textit{Trigger Event} is detected.
• This might generate other management events that set TRBSR_EL1.S to 1.

If a \textit{Trigger Event} is generated when collection is not stopped and the trigger mode is \textit{IRQ on trigger}, then all of the following occur:
• The Trace Buffer Unit initiates a \textit{trace unit flush} of the trace unit.
• The TRB_TRIG event is generated.

On completion of the trace unit flush, all of the following occur:
• A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.
• The other fields in TRBSR_EL1 are unchanged.

Because TRBSR_EL1.S is unchanged, trace continues to be collected and written to the trace buffer.

If a \textit{Trigger Event} is generated and the trigger mode is \textit{Ignore trigger}, then all of the following occur:
• TRBSR_EL1 is unchanged.
• The TRB_TRIG event is generated.

If a \textit{Trigger Event} is generated when \textit{collection is stopped}, then all of the following occur:
• TRBSR_EL1 is unchanged.
• The TRB_TRIG event is generated.

These rules mean that trace might be collected after the \textit{Trigger Event}, but are included to ensure that trace for the instructions that caused the trigger condition is not discarded in common cases.

The trigger mode is controlled by TRBLIMITR_EL1.TM.

See also \textit{Controlling generation of trace buffer management events on page K4-11675}.

### D6.3.4 Faults

A write by the Trace Buffer Unit might generate one or more of the following faults:

**Alignment fault**

If TRBPTR_EL1 is misaligned, the behavior is \textit{UNPREDICTABLE} and a write to the trace buffer by the Trace Buffer Unit might generate an Alignment fault. See also \textit{RMGZWR}.

**Translation fault**

Any access outside the virtual address or intermediate physical address space generates a Translation fault.
The translation of a virtual address or intermediate physical address to a physical address might generate Translation fault.

Writes to the trace buffer are made as privileged writes in the owning translation regime, meaning they are not affected by the TCR_ELx.E0PDy bits for the owning translation regime.

Address Size fault

The translation of a virtual address or intermediate physical address to a physical address, or use of an out-of-range physical address, might generate an Address Size fault.

Permission fault

Writes to the trace buffer are made as privileged writes in the owning translation regime. If the write does not have write permission, a Permission fault is generated. The value of PSTATE.PAN is ignored.

If the Trace Buffer Unit does not manage the dirty state in translation tables, then accesses ignore the Dirty Bit Modifier bit in Page and Block descriptors and as a result, might generate a Permission fault.

Access Flag fault

If the Trace Buffer Unit does not manage the Access flag in translation tables or hardware management of the Access flag state is disabled for the owning translation regime, then any access to a Page or Block with the Access flag bit set to 0 in a descriptor will generate an Access Flag fault.

TLB Conflict fault

IMPLEMENTATION DEFINED.

Unsupported atomic hardware update fault

If hardware update of the translation tables is not guaranteed atomic in regard to other agents that access the memory, the translation of a virtual address to a physical address might generate an Unsupported atomic hardware update fault.

External abort on translation table walk or translation table update

The translation of a virtual address to a physical address might generate an External abort on the translation table walk or translation table update, and be treated as a synchronous MMU fault. See also External aborts on page D6-5024.

MMU fault means any of these faults other than an Alignment fault.

RFYBCG

Writes to the trace buffer by the Trace Buffer Unit never generate watchpoints.

IDTKGB

Faults do not generate an actual Data Abort exception. The ESR and FAR registers are unchanged.

SGTLCY

To avoid MMU faults, Arm recommends:

- Software pins the Pages or Blocks used for the trace buffer. This includes a hypervisor pinning these Pages or Blocks in the stage 2 translation.
- If the Trace Buffer Unit does not manage the Access Flag and dirty state, software marks the Pages or Blocks as accessed and dirty. Software can discover whether address translations performed by the Trace Buffer Unit manage dirty state and the Access flag from TRBIDR_EL1.F.

RFSPBK

If a write by the Trace Buffer Unit generates an Alignment fault or MMU fault, and TRBSR_EL1.S is 0, then all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.
- TRBSR_EL1.S is set to 1, collection is stopped.
- TRBSR_EL1.EC is set to the appropriate one of the following values:
  - 0x24, stage 1 Data Abort on write to trace buffer.
  - 0x25, stage 2 Data Abort on write to trace buffer.
- TRBSR_EL1.FSC is set to indicate the type of the fault.
- TRBPTR_EL1 is set to the address that generated the fault.
- The other fields in TRBSR_EL1 are unchanged.

If a write by the Trace Buffer Unit generates an External abort on a translation table walk or translation table update, it is IMPLEMENTATION DEFINED whether TRBSR_EL1.EA is set to 1 or unchanged.
In the case of a stage 2 Data Abort on a write to trace buffer, the PE does not record whether the fault was due to a stage 2 fault on the access, or a stage 2 fault on a stage 1 translation table access.

**D6.3.5 External aborts**

A write to the trace buffer might generate an External abort, including an External abort on a translation table walk or translation table update:

**External abort**

The write might generate a synchronous or asynchronous External abort.

**External abort on translation table walk or translation table update**

The translation of a virtual address or intermediate physical address to a physical address might generate an External abort.

However, an External abort on a translation table walk or translation table update might be treated as a synchronous MMU fault. See *Faults* on page D6-5022.

When a write to the trace buffer generates an External abort, then one of the following occurs, and it is IMPLEMENTATION DEFINED which:

- The External abort is reported to the Trace Buffer Unit and treated as a trace buffer management event.
- The External abort is not reported to the Trace Buffer Unit and generates an asynchronous SError interrupt exception at the PE.

When an External abort is reported to the Trace Buffer Unit, then one of the following occurs, and it is IMPLEMENTATION DEFINED which:

- The External abort is handled synchronously by the Trace Buffer Unit.
- The External abort is handled asynchronously by the Trace Buffer Unit.

If the Trace Buffer Unit handles External aborts asynchronously, then all of the following apply:

- The External abort trace buffer management event might not be generated until after a first trace buffer management event has set TRBSR_EL1.S to 1.
- Writes to the trace buffer might generate a second trace buffer management event after the External abort trace buffer management event has set TRBSR_EL1.S to 1.
- The Trace Buffer Unit might collect further trace data from the trace unit and write it to memory before the External abort trace buffer management event sets TRBSR_EL1.S to 1.

When a write by the Trace Buffer Unit generates an External abort reported to the Trace Buffer Unit, all of the following occur:

- The External abort bit, TRBSR_EL1.EA, is set to 1.
- The Trace Buffer Unit stops writing trace data to the trace buffer.
- If the Trace Buffer Unit handles External aborts synchronously, TRBPTR_EL1 is set to the address that generated the External abort.
- If the Trace Buffer Unit handles External aborts asynchronously, TRBPTR_EL1 is not guaranteed to be set to the address that generated the External abort.
- If TRBSR_EL1.S is 0, then all of the following occur:
  - A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.
  - TRBSR_EL1.S is set to 1, *collection is stopped*.
  - TRBSR_EL1.EC is set to the appropriate one of the following values:
    - 0x24, stage 1 Data Abort on write to trace buffer.
    - 0x25, stage 2 Data Abort on write to trace buffer.
  - TRBSR_EL1.FSC is set to indicate the type of External abort.
- The other TRBSR_EL1 fields are unchanged.
Reporting the External abort to the Trace Buffer Unit and generating a trace buffer management event:

- Sets TRBSR_EL1.S to 1 and so discards further trace data.
- Allows error recovery software to isolate the event to the actions of the Trace Buffer Unit.

Not reporting the External abort to the Trace Buffer Unit and generating an asynchronous SError interrupt exception at the PE:

- Means that, while tracing is not prohibited, the trace unit might continue generating trace data that the Trace Buffer Unit accepts. However, the SError interrupt might be taken to an Exception level where tracing is prohibited.
- Might not allow error recovery software to isolate the event and error containment.

**D6.3.6 IMPLEMENTATION DEFINED management events**

When IMPLEMENTATION DEFINED conditions are met all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.
- TRBSR_EL1.S is set to 1, collection is stopped.
- TRBSR_EL1.EC is set to \( 0x1F \), IMPLEMENTATION DEFINED buffer management event.
- TRBSR_EL1.MSS is set to an IMPLEMENTATION DEFINED value.
- The other fields in TRBSR_EL1 are unchanged.

The intent of this event is for cases such as errata workarounds to allow an implementation to report any failure to write data to the buffer that is not covered by other codes. Arm recommends that such mechanisms are disabled on reset.
D6.4 Synchronization and the Trace Buffer Unit

Program Flow Trace data is generated by traced instructions. When an instruction is executed:

1. The PE decides whether to create a trace operation for the instruction.
2. If created, the trace operation generates the Program Flow Trace data.

For some trace unit implementations, Speculative instructions might generate trace operations, as well as architecturally Resolved instructions.

See on page K5-11677 for more information on these terms.

The trace unit might also generate asynchronous trace operations, that are not causally related to an executed instruction. If the trace unit implements the FEAT_ETE then the ETE Resources can generate trace operations that are not causally related to an instruction or Speculative instruction.

The architecture defines a Trace Synchronization event that synchronizes the operation of trace operations with the execution of instructions. Without correct use of the Trace Synchronization event, a trace operation might for instance read a stale value from a System register, causing trace data to be written to the wrong memory location, or the Trace Buffer Unit to otherwise generate UNPREDICTABLE software behavior. See also UNPREDICTABLE behavior on page D6-5038.

Trace operations operate independently of the instructions that are executed on the PE and make indirect reads and indirect writes of System registers as an external agent.

The synchronization requirements for direct reads, direct writes, indirect reads, and indirect writes of System registers made by instructions and external agents are defined in Synchronization requirements for AArch64 System registers on page D17-5547.

Synchronization in self-hosted trace on page D3-4762 defines that indirect reads of the TRFCR_EL1.{E1TRE, E0TRE} and TRFCR_EL2.{E2TRE, E0HTRE} trace filter controls when determining whether the current Execution stream is part of a prohibited trace region and an instruction \( A \) should generate a trace operation, are treated as indirect reads made by \( A \).

Each System register access made by the trace unit is one of the following, and the trace unit defines which:

- An indirect read or indirect write \( rw_A \) made by an instruction \( A \). For example, to determine whether to generate a trace operation for \( A \).
- An indirect read or indirect write \( rw_{A'} \) made by a trace operation \( A' \). This is in addition to System register accesses defined by this Manual as indirect reads or indirect writes made by the Trace Buffer Unit.
- An other indirect read or indirect write \( rw \), not directly related to either an instruction or trace operation. The trace unit defines the synchronization requirements for these registers.

In addition to the registers listed by RNRLDV, for the ETE, the following ETE System registers are indirectly read by an instruction \( A \) to determine whether \( A \) should generate a trace operation:

- TRCPRTL.R, the Trace unit enable bit in the Programming Control Register.
- TROSLSR.OSLK, the Trace OS Lock Status Register.

This Manual defines which System registers are indirectly read or indirectly written by the Trace Buffer Unit as part of the trace operation \( t_A \) for a traced instruction \( A \). For example:

- Trace Buffer Unit System registers.
- VMSA System registers and SCR_EL3.NS, when translating addresses generated by the Trace Buffer Unit. Other System registers are indirectly read or indirectly written by the trace unit as part of the trace operation \( t_A \) for a traced instruction \( A \). For example:
  - Other ETE System registers.
  - If context tracing is enabled, the applicable Context ID register or registers, CONTEXTIDR_EL1 or CONTEXTIDR_EL2.
  - If trace timestamping is enabled, any applicable counter offset, CNTV_OFF_EL2 or CNTPOFF_EL2.

Some ETE System registers are indirectly read or indirectly written by, for example, the ETE Resources when generating trace operations or updating the ETE Resources, and are made by neither a trace operation of an instruction nor an instruction.
The behavior of the ETE is defined by FEAT_ETE. See also *Trace synchronization and the Trace Unit* on page D6-5028.

The indirect reads and indirect writes to Trace Buffer Unit, VMSA and other System registers made by the Trace Buffer Unit are made by the trace operation tOP.

To synchronize trace operations, software must use the TSB CSYNC instruction to generate a Trace Synchronization event.

In the absence of any explicit synchronization, the trace unit generates the trace data for an instruction and the Trace Buffer Unit accepts, discards, or rejects the trace data in finite time. However:

- If the Trace Buffer Unit accepts the trace data, then the write of the trace data to memory requires explicit synchronization to Complete.
- The indirect writes to System registers made by a trace operation require explicit synchronization to guarantee they are observable.

### D6.4.1 Trace Synchronization event

Executing a TSB CSYNC instruction generates a Trace Synchronization event.

A trace operation tOP is not Microarchitecturally-finished before all of the following are true:

- All indirect reads and indirect writes of System registers made by tOP have been performed.
- If tOP generates a *Trigger Event* that in turn initiates a *trace unit flush*, then all of the following are true:
  - The trace unit flush is complete.
  - All trace operations the Trace Buffer Unit accepts before the trace unit flush completes are Microarchitecturally-finished.
  - All indirect writes to System registers made by the Trace Buffer Unit on completion of the trace unit flush have been performed.

Indirect reads and writes include but are not limited to the following:

- All indirect reads and indirect writes of the Trace Buffer Unit, VMSA System registers, and SCR_EL3.NS made by memory accesses performed by tOP.
- All indirect writes to System registers made by a trace buffer management event generated by tOP.

However, this does explicitly exclude any indirect writes of System registers made in response to an External abort by the access.

A trace operation tOP is complete when it is Microarchitecturally-finished and all memory accesses performed by tOP are Complete and any indirect writes of System registers made in response to an External abort response to the access have been performed.

If, following a Context synchronization event CSE the PE is executing in a Trace Prohibited region, a TSB CSYNC executed in the Trace Prohibited region and Non-debug state in program order after CSE is not Microarchitecturally-finished before all of the following are true:

- All trace operations tA generated by instructions A in program order before CSE are Microarchitecturally-finished.
- All trace operations tS generated by Speculative instructions S that are not in speculative execution order after CSE are Microarchitecturally-finished.
- All trace operations tR generated by the trace unit are Microarchitecturally-finished.
- The trace unit enters a state where the trace unit does not generate further trace operations and does not signal a Detected Trigger. The trace unit remains in this state while the PE is executing in the Trace Prohibited region.
- If a *trace unit flush* is initiated by a *Trigger Event* before the TSB CSYNC is Microarchitecturally-finished, the trace unit flush is complete, all trace operations the Trace Buffer Unit accepts before the trace unit flush completes are Microarchitecturally-finished, and any indirect writes made by the Trace Buffer Unit on completion of the trace unit flush have been performed.
These trace operations are synchronized by the TSB CSYNC.

A direct write \( W_2 \) to a System register made by an instruction \( B \) is Coherence-after an indirect read or indirect write \( rw_1 \) of the same System register made by a trace operation \( t_A \) for a traced instruction \( A \) if all of the following are true:

- Either \( A \) is executed in program order before a Context synchronization event \( CSE \), or \( A \) is \( CSE \).
- \( CSE \) is in program order before a Trace synchronization barrier \( TSB \).
- \( B \) is executed in program order after \( TSB \).
- After executing \( CSE \), the PE is in a Trace Prohibited region and \( TSB \) is executed in Non-debug state in the same Trace Prohibited region.

RZCDDS A direct write \( W_2 \) to a System register made by an instruction \( B \) is Coherence-after an indirect read or indirect write \( rw_1 \) of the same System register made by a trace operation \( t_A \) for a traced instruction \( A \) if all of the following are true:

- Either \( A \) is executed in program order before a Context synchronization event \( CSE \), or \( A \) is \( CSE \).
- \( CSE \) is in program order before a Trace synchronization barrier \( TSB \).
- \( B \) is executed in program order after \( TSB \).
- After executing \( CSE \), the PE is in a Trace Prohibited region and \( TSB \) is executed in Non-debug state in the same Trace Prohibited region.

RZCDDS emerges from the requirement in RMRVPT for the trace operations to be Microarchitecturally-finished by the TSB CSYNC operation.

RZCDDS A direct write \( W_2 \) to a System register made by an instruction \( B \) is Coherence-after an indirect read or indirect write \( rw_1 \) of the same System register made by a trace operation \( t_A \) for a traced instruction \( A \) if all of the following are true:

- Either \( A \) is executed in program order before a Context synchronization event \( CSE \), or \( A \) is \( CSE \).
- \( CSE \) is in program order before a Trace synchronization barrier \( TSB \).
- \( B \) is executed in program order after \( TSB \).
- After executing \( CSE \), the PE is in a Trace Prohibited region and \( TSB \) is executed in Non-debug state in the same Trace Prohibited region.

Self-hosted trace extension synchronization rules on page D6-5029 and Trace in Debug state on page D6-5033 define further rules for the operation of TSB CSYNC.

The PE does not stall indefinitely (or until interrupted) waiting for a TSB CSYNC. For example, the TSB CSYNC must not wait until there is no trace data left to write if the trace unit is capable of producing a constant stream of trace data.

A PE might abandon a TSB CSYNC executed in Non-debug state before it is Microarchitecturally-finished to take an interrupt, so long as the preferred return address is set such that the TSB CSYNC is re-executed when the interrupt handler completes. That is, the TSB CSYNC is only Speculatively executed.

Absent any Context synchronization event or DSB Data synchronization barrier, a TSB CSYNC instruction is not required to execute in program order with respect to other instructions or memory accesses. This means that software must execute additional barriers to guarantee that the trace operations are Microarchitecturally-finished and/or complete.

D6.4.2 Trace synchronization and the Trace Unit

The ETE has Resources that can generate trace operations that are not directly generated by an instruction or Speculative instruction.

The ETE specification defines the following rules:

- The Resources do not generate trace operations in the Paused state.
- If, following a Context synchronization event, the PE is executing in a Trace Prohibited region and the ETE is enabled, the ETE pauses the ETE Resources.
- How software synchronizes indirect writes to System registers made by trace operations generated by Resources.

The behavior of the ETE is defined by FEAT_ETE.

If FEAT_ETE is implemented, further rules are defined for the behavior of the trace unit when a TSB CSYNC instruction is executed.

A trace operation \( t_0 \) generated by ETE Resources inherits the synchronization requirements for a trace operation generated by an instruction \( A \), even if no trace operation is generated by \( A \), provided that one of the following applies:

- The requirement is that \( A \) is executed in program-order after \( CSE \) and tracing was prohibited before \( CSE \) and is allowed after \( CSE \).
- The requirement is that either \( A \) is executed in program-order before \( CSE \) or \( A \) is \( CSE \), and tracing was allowed before \( CSE \) and is prohibited after \( CSE \).

If the trace unit becomes enabled when the PE is executing a Trace Prohibited region, it does not generate any trace operations, including trace operations for Speculative instructions and other trace operations not generated by instructions, until the PE enters a region where tracing is allowed.

The trace unit defines the sequence by which software enables the trace unit.
D6.4.3 Self-hosted trace extension synchronization rules

ICMGRF FEAT_TRF defines further rules for the TSB CSYNC instruction. In particular, how a TSB CSYNC instruction synchronizes direct reads and indirect writes to a System register with respect to indirect reads and indirect writes of the same System register made by trace operations.

These rules restate the synchronization behavior described in Memory barriers on page B2-174 and Synchronization in self-hosted trace on page D3-4762.

RTSPXF is similar to RZCDDS and RXQVZW. However:
- RTSPXF applies whether the TSB CSYNC operation is executed in a trace prohibited or trace allowed region, in both Non-debug state and Debug state. RZCDDS applies only when the TSB CSYNC is executed in a Trace Prohibited region and the PE is in Non-debug state, and RXQVZW applies only when the TSB CSYNC is executed when the trace unit is disabled and the PE is in Debug state.
- RTSPXF applies only to System registers accessed by the trace unit as part of a trace operation. RZCDDS and RXQVZW apply to all System register accesses made by the trace operation and includes indirect reads and indirect writes made by the Trace Buffer Unit.

See RSXXQJ and IDDLVC.

RBFJKD An indirect read r1 of a System register made by a trace operation tA for a traced instruction A Reads-from a direct write W2 to the same System register made by an instruction B if all of the following are true:
- A is executed in program order after a Context synchronization event CSE.
- B is executed in program order before CSE.

See also IFVBPF.

SYPWVJ RBFJKD means that, if the PE enters a region where tracing is allowed by executing a Context synchronization event, such as an ERET instruction when SCTLR_ELx.EOS is 1, then all indirect reads and writes of System registers made by trace operations generated after entering the tracing allowed region will observe the values in those System registers written by direct writes before the Context synchronization event.

RMQXJM An indirect write w1 of a System register made by a trace operation tA for a traced instruction A is Coherence-after a direct write W2 of the same System register made by an instruction B if all of the following are true:
- A is executed in program order after a Context synchronization event CSE.
- B is executed in program order before CSE.

See also IFVBPF.

RTSPXF A direct write W2 to a System register made by an instruction B is Coherence-after an indirect read or indirect write rw1 of the same System register made by the trace unit as part of a trace operation tA for a traced instruction A if all of the following are true:
- Either A is executed in program order before a Context synchronization event CSE, or A is CSE.
- CSE is in program order before a Trace synchronization barrier TSB.
- B is executed in program order after TSB.

See also IMQTRQ.

RVTNCS A direct read R2 of a System register made by an instruction B Reads-from an indirect write w1 to the same System register made by a trace operation tA for a traced instruction A if all of the following are true:
- Either A is executed in program order before a first Context synchronization event CSE1, or A is CSE1.
- CSE1 is in program order before a Trace synchronization barrier TSB.
- TSB is executed in program order before a second Context synchronization event CSE2.
- B is executed in program order after CSE2.

See also ISLWRW.
**RNNRHD**

An instruction A in program-order after a direct write W that modifies one of the trace filter controls, TRFCR_EL1.{E1TRE, E0TRE} and TRFCR_EL2.{E2TRE, E0HTRE}, Reads-from W when determining whether A should generate a trace operation, if there is no intervening direct write to the same register and any of the following is true:

- A is in program-order after a Context synchronization event CSE and CSE is in program-order after W.
- An instruction B Reads-from W when determining whether B should generate a trace operation, and A is in program-order after B.

**IVKGRM**

RNNRHD means that for the instructions between a direct write to one of the trace filter controls to either enable or disable trace at the current Exception level and a following Context synchronization event, although it is UNPREDICTABLE whether each instruction observes the old or new values of the trace filter controls, once one instruction has observed the new value, all subsequent instructions also observe the new value.

However this might not happen, and all instructions might observe the old values until the Context synchronization event occurs.

This guarantees that trace switches either on and off cleanly, and is required by Program Flow Trace protocols.
D6.5 Trace synchronization and memory barriers

RVLWDM If, following a Context synchronization event CSE, the PE is executing in a Trace Prohibited region, a DSB with required access types of reads and write is executed in program order after a TSB CSYNC operation that is executed in the Trace Prohibited region in program order after CSE, then in addition to the requirements in this Manual, the DSB does not complete until all explicit memory accesses of the required access type made by the trace operations synchronized by the TSB CSYNC are Complete for the set of observers in the required Shareability domain.

See also RMPSXN.

RDFHWV An explicit read, explicit write, translation table walk, cache maintenance operation, or TLB invalidate operation M₁ will be Observed-by a read or a write RW₂ of a Location made by a trace operation tA relating to a traced instruction A if all of the following are true:

- A is executed in program-order after a Context synchronization event CSE.
- CSE is in program order after a Data Synchronization Barrier DSB.
- DSB does not complete before M₁ is complete.

INLHGN In RDFHWV, CSE is required to allow A to be traced as a Speculative instruction before it is Canceled or Resolved. The equivalent rule for SPE does not require CSE as SPE cannot write profiling records to memory until the profiled operation is Canceled or Resolved. See Trace of Speculative execution on page D6-5032.

RJPPZZ For the indirect writes to TRBPTR_EL1 and TRBSR_EL1 that are made as a result of an External abort on a write of trace data to memory, the synchronization rules apply only after the write to memory has completed.
D6.6 Trace of Speculative execution

In the standard model of execution for an instruction, instructions are executed as Speculative operations and then later become one of the following:

- Resolved. These instructions will then proceed to Complete.
- Canceled.
- Transaction-failed, if FEAT_TME is implemented.
- Transaction-canceled, if FEAT_TME is implemented.

A trace unit might generate trace for Speculative instructions before they are Resolved, Canceled, Transaction-failed or Transaction-canceled, and this trace can be written to memory. This means that, as well as indirectly reading System registers or memory, a trace operation $t_S$ for a Speculative instruction $S$ might perform any of the following:

- Indirectly write to System registers.
- Write to memory.

This sets trace operations apart from the normal operation of instructions, as the Arm architecture prohibits, for example, a Canceled instruction from updating a System register or memory. (Performance Monitors and Statistical Profiling can also cause speculative updates of System registers and memory.)

The preceding rules deal with the ordering of trace operations for Resolved (and ultimately Complete) instructions, and the requirements for synchronization based on the position of those instructions in the program order of the Execution stream.

This section extends the rules to cover Speculative instructions that are later Canceled, Transaction-canceled or Transaction-failed, and for trace operations generated by ETE Resources. Because these trace operations are not generated by instructions that Complete, they cannot be determined to be in program order with respect to architecturally Complete instructions.

A Speculative instruction $S$ is in speculative execution-order after an instruction $A$ if $S$ will be in program order after $A$ if $S$ is Resolved, even if $S$ is subsequently Canceled, Transaction-failed, or Transaction-canceled.

For example, a Speculative instruction $S$ is in speculative execution-order after a Resolved instruction $A$ if either of the following are true:

- The branch predictor mispredicted $A$ and, had the prediction been correct, $S$ would be in program order after $A$. Branch predictor means any structure that causes the PE to execute Speculative instructions. This includes, for example, branch history buffers, branch target caches, and instruction trace caches. It is not limited to structures that predict only the direction and/or target of branch instructions.

- $S$ forms part of a Transaction $T$ that was Transaction-canceled or Transaction-failed and $A$ is the Resolved TSTART operation for the outermost Transaction containing $T$.

Speculative execution-order does not provide a complete ordering. A pair of Speculative instructions $A$ and $B$ might not be ordered with respect to each other. For example, if $A$ and $B$ are respectively the result of different incorrect predictions by the branch predictor. However, each Speculative instruction is in speculative execution-order after at least one Resolved instruction.

The architecture requires that instructions that generate Context synchronization events do not appear to be executed speculatively.

A trace operation $t_S$ generated by a Speculative instruction $S$ that is in speculative execution-order after a Resolved instruction $A$ inherits the synchronization requirements for a trace operation generated by $A$, even if no trace operation is generated by $A$, provided that one of the following applies:

- The requirement is that $A$ is executed in program-order after CSE, and $S$ is in speculative execution-order after CSE.

- The requirement is that either $A$ is executed in program-order before CSE or $A$ is CSE, and $S$ is not in speculative execution-order after CSE.

See also $I_{WLOHQ}$ and $I_{GXWM}$. 
D6.7 Trace in Debug state

If the trace unit is disabled, then a TSB CSYNC executed in Debug state is not Microarchitecturally-finished before all of the following are true:

- All trace operations $t_A$ generated by instructions $A$ in program order before the PE entered Debug state are Microarchitecturally-finished.
- All trace operations $t_S$ generated by Speculative instructions $S$ that are not in speculative execution order after the entry to Debug state are Microarchitecturally-finished.
- All trace operations $t_R$ generated by the trace unit are Microarchitecturally-finished.
- The trace unit enters a state where the trace unit does not generate further trace operations and does not signal a Detected Trigger. The trace unit remains in this state while the PE is in Debug state.
- If a trace unit flush is initiated by a Trigger Event before the TSB CSYNC is Microarchitecturally-finished, the trace unit flush is complete, all trace operations the Trace Buffer Unit accepts before the trace unit flush completes are Microarchitecturally-finished, and any indirect writes made by the Trace Buffer Unit on completion of the trace unit flush have been performed.

These trace operations are synchronized by the TSB CSYNC.

A direct write $W_2$ to a System register made by an instruction $B$ is Coherence-after an indirect read or indirect write $rw_1$ of the same System register made by a trace operation $t_A$ for a traced instruction $A$ if all of the following are true:

- $A$ is executed in program order before the PE entered Debug state.
- $B$ is executed in program order after a Trace synchronization barrier TSB.
- TSB was executed in Debug state when the trace unit was disabled.

IVHGVC RJXQVZW emerges from the requirement in RWYHRT for the trace operations to be Microarchitecturally-finished by the TSB CSYNC operation.

Trace Synchronization event on page D6-5027 and Self-hosted trace extension synchronization rules on page D6-5029 define further rules for the operation of TSB CSYNC.

RMPXXN A DSB with required access types of reads and write is executed after a TSB CSYNC operation executed in Debug state, then in addition to the requirements in this Manual, the DSB does not complete until all explicit memory accesses of the required access type made by the trace operations synchronized by the TSB CSYNC are Complete for the set of observers in the required Shareability domain.
D6.8 Detailed synchronization litmus tests

This section details example synchronization scenarios and litmus tests for TSB CSYNC. These are derived from FEAT_TRF and Trace Synchronization event on page D6-5027.

This section uses the terms program order, Reads-from and Coherence-after to define the ordering of System register and memory accesses made by trace operations. These terms are defined for memory accesses in Definition of the Arm memory model on page B2-159. For the purposes of this section, these terms are used for System registers as well as memory accesses.

The terms external agent, Observer, and Observed-by, also used in this section, are also defined in Definition of the Arm memory model on page B2-159.

This section does not describe the synchronization rules in Debug state defined in Trace in Debug state on page D6-5033. In general, litmus tests for Debug state can be derived by applying the following modifications:

- Where a rule mentions a Context synchronization event (CSE) coming before a TSB CSYNC operation, if the TSB CSYNC is executed in Debug state, then the entry to Debug state can replace that CSE for the rule.
- Where a rule mentions the PE executing instructions in a Trace Prohibited region following the CSE, then executing the instructions in Debug state with the trace unit disabled is sufficient for the rule.

Exit from Debug state is a Context synchronization event.

Figure D6-3 shows RBFJKD and RMJQXM.

![Figure D6-3 Indirect read or indirect write by Trace operation after direct write](image)

Figure D6-4 Direct write after indirect read or indirect write by Trace operation

In RVTNCS, the second Context synchronization event CSE2 is required to ensure the direct read B is not executed before the synchronization barrier TSB. Figure D6-5 on page D6-5035 shows this.
If the trace is not prohibited after the Context synchronization event, further trace operations might be generated that are not guaranteed to be synchronized by the TSB CSYNC. For ISLWRW, this applies to the first Context synchronization event CSE1.

Trace is prohibited at higher Exception levels than the owning Exception level. This means that if the PE takes an exception to a higher Exception level than the owning Exception level then trace is prohibited at by taking the exception.

Figure D6-6 shows RDFHWV for an explicit read or a write M1 of a Location made by an instruction B in program order before DSB.

If all of the following are true, RVILWDM requires that a DSB with required access types of reads and writes does not complete until at least all reads or writes RW made by the Trace Buffer Unit for all trace operations tA relating to a traced instruction A are complete for the set of the observers in the required Shareability domain:

• Either A is executed in program order before a Context synchronization event CSE, or A is CSE.
• The PE is executing in a Trace Prohibited region after CSE.
• CSE is in program order before a Trace synchronization barrier TSB.
• TSB is executed in program order before the DSB.

Figure D6-7 on page D6-5036 shows a read or a write RW1 of a Location made by the Trace Buffer Unit for a trace operation tA relating to a traced instruction A is complete and therefore will be Observed-by a read or a write RW2 of the same Location made by an instruction B executed in program order after a DSB ISH instruction.
RHCVVS defines Trace synchronization for Speculative instructions.

For example, an indirect read \( r_1 \) of a System register made by a trace operation \( t_S \) for a traced Speculative instruction \( S \) from a direct write \( W_2 \) to the same System register made by an instruction \( B \), if all of the following are true:

- \( S \) is executed in speculative execution-order after a Resolved instruction \( A \).
- \( A \) is executed in program order after a Context synchronization event \( CSE \).
- \( B \) is executed in program order before \( CSE \).

Figure D6-8 shows this.

\[
\text{Figure D6-7 Observation of Trace operation memory access}
\]

RHCVVS defines Trace synchronization for Speculative instructions.

For example, if all of the following are true, a DSB with required access types of reads and writes does not complete until at least all reads or writes \( RW \) made by the Trace Buffer Unit for all trace operations \( t_S \) relating to a traced Speculative instructions \( S \) are complete for the required Shareability domain:

- \( S \) is executed in speculative execution-order after a Resolved instruction \( A \).
- \( A \) is executed in program order before a Context synchronization event \( CSE \).
- \( S \) is not in speculative execution-order after \( CSE \).
- \( CSE \) is in program order before a Trace synchronization barrier \( TSB \).
- The PE is executing in a Trace Prohibited region after \( CSE \).
- \( TSB \) is executed in program order before the DSB.

Figure D6-9 on page D6-5037 shows a read or a write \( RW_1 \) of a Location made by the Trace Buffer Unit for a trace operation \( t_S \) relating to a traced Speculative instruction \( S \) is Complete and therefore will be Observed-by a read or a write \( RW_2 \) of the same Location made by an instruction \( B \) executed in program order after a DSB ISH instruction.
Figure D6-9 Observation of Speculative Trace operation memory access
### D6.9 UNPREDICTABLE behavior

In the absence of correct synchronization events, it is UNPREDICTABLE whether an indirect read made by a trace operation of a System register updated by a direct write will return the old or the new values. A trace operation might make multiple indirect reads of a System register, and each might return a different one of the old or the new values.

For example, a trace operation might read MDCR_EL2.E2TB twice, as follows:

1. When the trace operation is first generated, to evaluate TraceAllowed() and determine whether trace is prohibited.
2. When the trace operation is complete and ready to be written to memory, to evaluate TraceBufferOwner() to determine the context for TRBPTR_EL1.

If MDCR_EL2.E2TB is updated without proper synchronization between these two events, both the old and new value might be used.

In the absence of correct synchronization events, it is UNPREDICTABLE whether a direct read of a System register updated by an indirect write made by a trace operation will return the old or the new values.

If an instruction is traced because the Trace Buffer Unit is enabled and running, but a later indirect read of a System register by the trace operation for the instruction determines that the trace data cannot be written to memory, because the Trace Buffer Unit now appears to be disabled, then one of the following occurs, and it is CONSTRAINED UNPREDICTABLE which:

- The trace data is written to memory.
- The trace data is sent to an IMPLEMENTATION DEFINED trace bus.
- The trace data is written to memory and sent to an IMPLEMENTATION DEFINED trace bus.
- The Trace Buffer Unit discards the trace data.

This also includes any trace data that might be flushed by the trace unit, for example due to a TSBSYNC operation.

If the Trace Buffer Unit discards the trace data, a trace buffer management event might be generated:

- TRBSR_EL1.EC is set to a CONSTRAINED UNPREDICTABLE choice of the following values:
  - 0x00, other buffer management event.
  - 0x1F, IMPLEMENTATION DEFINED buffer management event.
- If TRBSR_EL1.EC is set to 0x00, then TRBSR_EL1.BSC is set to 0b000000 to indicate that the trace buffer is not full.

It is also CONSTRAINED UNPREDICTABLE whether any of the following occur, whether or not the trace data is written to memory:

- The current write pointer and, if TRBSR_EL1.TRG is 1, the Trigger Counter are updated for the trace data.
- A trace buffer management event that would have been generated is observed and/or generated.
- A PMU event that would have been generated is generated.

See also **Context switching on page K4-11673**.
Chapter D7
The AArch64 System Level Memory Model

This chapter provides a system level view of the general features of the memory system. It contains the following sections:

- About the memory system architecture on page D7-5040.
- Address space on page D7-5041.
- Mixed-endian support on page D7-5042.
- Cache support on page D7-5043.
- External aborts on page D7-5072.
- Memory barrier instructions on page D7-5074.
- Pseudocode description of general memory System instructions on page D7-5075.
D7.1 About the memory system architecture

The Arm architecture supports different implementation choices for the memory system microarchitecture and memory hierarchy, depending on the requirements of the system being implemented. In this respect, the memory system architecture describes a design space in which an implementation is made. The architecture does not prescribe a particular form for the memory systems. Key concepts are abstracted in a way that permits implementation choices to be made while enabling the development of common software routines that do not have to be specific to a particular microarchitectural form of the memory system. For more information about the concept of a hierarchical memory system see Memory hierarchy on page B2-184.

If FEAT_MTE2 is implemented, the definitions of the memory model which apply to data accesses and data apply to Allocation Tag accesses and Allocation tags, unless otherwise specified in Chapter D9 The Memory Tagging Extension.

D7.1.1 Form of the memory system architecture

The A-profile architecture includes a Virtual Memory System Architecture (VMSA). Chapter D8 The AArch64 Virtual Memory System Architecture describes the AArch64 view of the VMSA.

D7.1.2 Memory attributes

Memory types and attributes on page B2-196 describes the memory attributes, including how different memory types have different attributes. Each location in memory has a set of memory attributes, and the translation tables define the virtual memory locations, and the attributes for each location.

Table D7-1 on page D7-5040 shows the memory attributes that are visible at the system level.

<table>
<thead>
<tr>
<th>Memory type</th>
<th>Shareability</th>
<th>Cacheability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device a</td>
<td>Outer Shareable</td>
<td>Non-cacheable.</td>
</tr>
<tr>
<td>Normal</td>
<td>One of:</td>
<td>One of b:</td>
</tr>
<tr>
<td></td>
<td>• Non-shareable.</td>
<td>• Non-cacheable.</td>
</tr>
<tr>
<td></td>
<td>• Inner Shareable.</td>
<td>• Write-Through Cacheable.</td>
</tr>
<tr>
<td></td>
<td>• Outer Shareable.</td>
<td>• Write-Back Cacheable.</td>
</tr>
</tbody>
</table>

a. Takes additional attributes, see Device memory on page B2-200.
b. See also Cacheability, cache allocation hints, and cache transient hints on page D7-5046.

D7.2 Address space

The architecture is designed to support a wide range of applications with different memory requirements. It supports a range of physical address (PA) sizes, and provides associated control and identification mechanisms. For more information, see Implemented physical address size on page D8-5088.

D7.2.1 Virtual address space overflow

When a PE performs a Simple sequential execution of instructions, it calculates:

\[(address\_of\_current\_instruction) + (size\_of\_executed\_instruction)\]

This calculation is performed after each instruction to determine which instruction to execute next.

If the address calculation performed after executing an instruction overflows 0xFFFF FFFF FFFF FFFF, the program counter becomes UNKNOWN. Similarly, if the address calculation based on the program counter for the value of the link register or exception link register overflows 0xFFFF FFFF FFFF FFFF, then those registers also become UNKNOWN.

Note

Address tags are not propagated to the program counter, so the tag does not affect the address calculation.

Where an instruction accesses a sequential set of bytes that crosses the 0xFFFF_FFFF_FFFF_FFFF boundary when tagged addresses are not used, or the 0xFFF_FFFF_FFFF_FFFF boundary when tagged addresses are used, then the virtual address accessed for the bytes above this boundary is UNKNOWN. When tagged addresses are used, the value of the tag associated with the address also becomes UNKNOWN.
D7.3 Mixed-endian support

A control bit, SCTLR_EL1.E0E is provided to allow the endianness of explicit data accesses made while executing at EL0 to be controlled independently of those made while executing at EL1. Table D7-2 on page D7-5042 shows the endianness of explicit data accesses and translation table walks.

Table D7-2 Endianness support

<table>
<thead>
<tr>
<th>Exception level</th>
<th>Explicit data accesses</th>
<th>Stage 1 translation table walks</th>
<th>Stage 2 translation table walks</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL0</td>
<td>SCTLR_EL1.E0E</td>
<td>SCTLR_EL1.EE</td>
<td>SCTLR_EL2.EE</td>
</tr>
<tr>
<td>EL1</td>
<td>SCTLR_EL1.EE</td>
<td>SCTLR_EL1.EE</td>
<td>SCTLR_EL2.EE</td>
</tr>
<tr>
<td>EL2</td>
<td>SCTLR_EL2.EE</td>
<td>SCTLR_EL2.EE</td>
<td>n/a</td>
</tr>
<tr>
<td>EL3</td>
<td>SCTLR_EL3.EE</td>
<td>SCTLR_EL3.EE</td>
<td>n/a</td>
</tr>
</tbody>
</table>

--- Note ---

SCTLR_EL1.E0E has no effect on the endianness of the LDTR, LDTRH, LDTRSH, and LDTRSW instructions, or on the endianness of the STTR and STTRH instructions, when these are executed at EL1.

AArch64 state provides the following options for endianness support:

- All Exception levels support mixed-endianness:
  - SCTLR_ELx.EE is RW and SCTLR_EL1.E0E is RW.
- Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only little-endianness:
  - SCTLR_ELx.EE is RES0 and SCTLR_EL1.E0E is RW.
- Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only big-endianness:
  - SCTLR_ELx.EE is RES1 and SCTLR_EL1.E0E is RW.
- All Exception levels support only little-endianness:
  - SCTLR_ELx.EE is RES0 and SCTLR_EL1.E0E is RES0.
- All Exception levels support only big-endianness:
  - SCTLR_ELx.EE is RES1 and SCTLR_EL1.E0E is RES1.

If mixed endian support is implemented for an Exception level using AArch32, endianness is controlled by PSTATE.E. For exception returns to AArch32 state, PSTATE.E is copied from SPSR_ELx.E. If the target Exception level supports only little-endian accesses, SPSR_ELx.E is RES0. If the target Exception level supports only big-endian accesses, SPSR_ELx.E is RES1. PSTATE.E is ignored in AArch64 state.

The BigEndian() function determines whether the current Exception level and Execution state are using big-endian data. This function is defined in Chapter J1 Armv8 Pseudocode.

For more information about endianness in the Arm architecture see Endian support on page B2-192.
D7.4 Cache support

This section describes the cache identification and control mechanisms, and the A64 cache maintenance instructions, in the following sections:

- General behavior of the caches.
- Cache identification on page D7-5044.
- Cacheability, cache allocation hints, and cache transient hints on page D7-5046.
- Enabling and disabling the caching of memory accesses on page D7-5047.
- Behavior of caches at reset on page D7-5049.
- Non-cacheable accesses and instruction caches on page D7-5049.
- About cache maintenance in AArch64 state on page D7-5050.
- A64 Cache maintenance instructions on page D7-5054.
- Data cache zero instruction on page D7-5068.
- Cache lockdown on page D7-5068.
- System level caches on page D7-5070.
- Branch prediction on page D7-5070.
- Execution, data prediction and prefetching restriction System instructions on page D7-5070.

See also Caches on page D8-5215.

D7.4.1 General behavior of the caches

When a memory location has a Normal Cacheable memory attribute, determining whether a copy of the memory location is held in a cache still depends on many aspects of the implementation. The following non-exhaustive list of factors might be involved:

- The size, line length, and associativity of the cache.
- The cache allocation algorithm.
- Activity by other elements of the system that can access the memory.
- Speculative instruction fetching algorithms.
- Speculative data fetching algorithms.
- Interrupt behaviors.

Given this range of factors, and the large variety of cache systems that might be implemented, the architecture cannot guarantee whether:

- A memory location present in the cache remains in the cache.
- A memory location not present in the cache is brought into the cache.

Instead, the following principles apply to the behavior of caches:

- The architecture has a concept of an entry locked down in the cache. How lockdown is achieved is IMPLEMENTATION DEFINED, and lockdown might not be supported by:
  - A particular implementation.
  - Some memory attributes.
- An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.
- A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

Note

For more information, see The interaction of cache lockdown with cache maintenance instructions on page D7-5069.
Any memory location that has a Normal Cacheable attribute at either the current Exception level or at a higher Exception level can be allocated to a cache at any time.

It is guaranteed that no memory location will be allocated into a Data or Unified cache if that location does not have a Normal Cacheable attribute in either:
- The translation regime at the current Exception level.
- The translation regime at any higher Exception level.

For data accesses, any memory location with a Normal Inner Shareable or Normal Outer Shareable attribute is guaranteed to be coherent with all Requesters in its shareability domain.

Any memory location is not guaranteed to remain incoherent with the rest of memory.

The eviction of a cache entry from a cache level can overwrite memory that has been written by another observer only if the entry contains a memory location that has been written to by an observer in the shareability domain of that memory location. The maximum size of the memory that can be overwritten is called the Cache Write-back Granule. In some implementations the CTR_EL0 identifies the Cache Write-back Granule.

The allocation of a memory location into a cache cannot cause the most recent value of that memory location to become invisible to an observer if it was previously visible to that observer.

Note: The Cacheability attribute of an address is determined by the applicable translation table entry for that address, as modified by any applicable System register Cacheability controls, such as the SCTLR_EL1.{I, C} controls.

For the purpose of these principles, a cache entry covers at least 16 bytes and no more than 2KB of contiguous address space, aligned to the size of the cache entry.

D7.4.2 Cache identification

The cache identification registers describe the implemented caches that are affected by cache maintenance instructions executed on the PE. This includes the cache maintenance instructions that:

- Affect the entire cache, for example IC IALU.
- Operate by VA, for example IC IVAU.
- Operate by set/way, for example DC ISW.

The cache identification registers are:

- The Cache Type Register, CTR_EL0, that defines:
  - The minimum line length of any of the instruction caches affected by the instruction cache maintenance instructions.
  - The minimum line length of any of the data or unified caches, affected by the data cache maintenance instruction.
  - The cache indexing and tagging policy of the Level 1 instruction cache.

Note: It is IMPLEMENTATION DEFINED whether caches beyond the PoC will be reported by this mechanism, and because of the possible existence of system caches some caches before the PoC might not be reported. For more information about system caches see System level caches on page D7-5070.

- A single Cache Level ID Register, CLIDR_EL1, that defines:
  - The type of cache that is implemented and can be maintained using the architected cache maintenance instructions that operate by set/way or operate on the entire cache at each cache level, up to the maximum of seven levels.
  - The Level of Coherence (LoC) for the caches. See Terms used in describing the cache maintenance instructions on page D7-5050 for the definition of LoC.
— The Level of Unification Uniprocessor (LoUU) for the caches. See Terms used in describing the cache maintenance instructions on page D7-5050 for the definition of LoUU.
— An optional ICB field to indicate the boundary between the caches use for caching Inner Cacheable memory regions and those used only for caching Outer Cacheable regions.

• A single Cache Size Selection Register, CSSELR_EL1, that selects the cache level and sort of cache (Instruction, Data/Unified/Tag) of the current Cache Size Identification Register.

• For each implemented cache that is identifiable by this mechanism, across all the levels of caching, a Cache Size Identification Register, CCSIDR_EL1, that defines:
  — Whether the cache supports Write-Through, Write-Back, Read-Allocate and Write-Allocate.
  — The number of sets, associativity and line length of the cache. See Terms used in describing the cache maintenance instructions on page D7-5050 for a definition of these terms.

----- Note ----- 
From Armv8.3, multiple formats of the Cache Size Identification Register are supported. For more information, see Possible formats of the Cache Size Identification Register, CCSIDR_EL1.

To determine the cache topology associated with a PE:

1. Read the Cache Type Register to find the indexing and tagging policy used for the Level 1 instruction cache. This register also provides the size of the smallest cache lines used for the instruction caches, and for the data and unified caches. These values are used in cache maintenance instructions.

2. Read the Cache Level ID Register to find what caches are implemented. The register includes seven Cache type fields, for cache levels 1 to 7. Scanning these fields, starting from Level 1, identifies the instruction, data or unified caches implemented at each level. This scan ends when it reaches a level at which no caches are defined. The Cache Level ID Register also specifies the Level of Unification (LoU) and the Level of Coherence (LoC) for the cache implementation.

3. For each cache identified at stage 2:
   • Write to the Cache Size Selection Register to select the required cache. A cache is identified by its level, and whether it is:
     — An instruction cache.
     — A data or unified cache.
   • Read the Cache Size Identification Register to find details of the cache.

Possible formats of the Cache Size Identification Register, CCSIDR_EL1

From Armv8.3, the Cache Size Identification Register, CCSIDR_EL1 has two different formats available for defining the number of sets and associativity of the cache. For a definition of these terms, see Terms used in describing the cache maintenance instructions on page D7-5050.

When FEAT_CCIDX is implemented:
   • CCSIDR_EL1 is a 64-bit register.
   • The length of the CCSIDR_EL1.Assoc field is 21 bits. This limits the associativity of the currently selected cache to 2^{21}.
   • The length of the CCSIDR_EL1.NumSets field is 24 bits. This limits the number of sets in the currently selected cache to 2^{24}.

This is the 64-bit format of the Cache Size Identification Register.

When FEAT_CCIDX is not implemented:
   • CCSIDR_EL1 is a 32-bit register.
   • The length of the CCSIDR_EL1.Assoc field is 10 bits. This limits the associativity of the currently selected cache to 2^{10}.
The AArch64 System Level Memory Model

D7.4 Cache support

- The length of the CCSIDR_EL1.NumSets field is 15 bits. This limits the number of sets in the currently selected cache to $2^{15}$.

This is the 32-bit format of the Cache Size Identification Register.

When one of these formats is implemented, it is implemented across all the levels of caching.

D7.4.3 Cacheability, cache allocation hints, and cache transient hints

Cacheability only applies to Normal memory, and can be defined independently for Inner and Outer cache locations. All types of Device memory are always treated as Non-cacheable.

As described in Memory types and attributes on page B2-196, the memory attributes include a cacheability attribute that is one of:
- Non-cacheable.
- Write-Through cacheable.
- Write-Back cacheable.

Cacheability attributes other than Non-cacheable can be complemented by a cache allocation hint. This is an indication to the memory system of whether allocating a value to a cache is likely to improve performance. In addition, it is IMPLEMENTATION DEFINED whether a cache transient hint is supported, see Transient cacheability hint.

The cache allocation hints are assigned independently for read and write accesses, and therefore when the Transient hint is supported the following cache allocation hints can be assigned:

For read accesses: Read-Allocate, Transient Read-Allocate, or No Read-Allocate.

For write accesses: Write-Allocate, Transient Write-Allocate, or No Write-Allocate.

--- Note ---

- A Cacheable location with both No Read-Allocate and No Write-Allocate hints is not the same as a Non-cacheable location. A Non-cacheable location has coherency guarantees for all observers within the system that do not apply for a location that is Cacheable, No Read-Allocate, No Write-Allocate.

- Implementations can use the cache allocation hints to limit cache pollution to a part of a cache, such as to a subset of ways.

- For VMSAv8-64 translation table walks, the TCR_ELx.{IRGNn, ORGNn} fields define the memory attributes of the translation tables, including the cacheability. However, this assignment supports only a subset of the cacheability attributes described in this section.

The architecture does not require an implementation to make any use of cache allocation hints. This means an implementation might not make any distinction between memory locations with attributes that differ only in their cache allocation hint.

Transient cacheability hint

In Armv8, it is IMPLEMENTATION DEFINED whether a Transient hint is supported. In an implementation that supports the Transient hint, the Transient hint is a qualifier of the cache allocation hints, and indicates that the benefit of caching is for a relatively short period. It indicates that it might be better to restrict allocation of transient entries, to avoid possibly casting-out other, less transient, entries.

--- Note ---

The architecture does not specify what is meant by a relatively short period.

The description of the AArch64 MAIR_EL1, MAIR_EL2, and MAIR_EL3 registers, and the AArch32 MAIR0, MAIR1, HMAIR0, and HMAIR1 registers, includes the assignment of the Transient hint in an implementation that supports this option. In this assignment:

- The Transient hint is defined independently for Inner Cacheable and Outer Cacheable memory regions.
D7.4 Cache support

D7.4.4 Enabling and disabling the caching of memory accesses

Cacheability control fields can force all memory locations with the Normal memory type to be treated as Non-cacheable, regardless of their assigned Cacheability attribute. Independent controls are provided for each stage of address translation, with separate controls for:

- Data accesses. These controls also apply to accesses to the translation tables.
- Instruction accesses.

Note

These Cacheability controls replace the cache enable controls provided in previous versions of the Arm architecture.

The Cacheability control fields and their effects are as follows:

For the EL1&0 translation regime

- When the value of SCTL_EL1.C is 0:
  - All stage 1 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the EL1&0 stage 1 translation tables are Non-cacheable.
- When the value of SCTL_EL1.I is 0:
  - All stage 1 translations for instruction accesses to Normal memory are Non-cacheable.
- When the value of HCR_EL2.CD is 1:
  - All stage 2 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the EL1&0 stage 2 translation tables are Non-cacheable.
- When the value of HCR_EL2.ID is 1:
  - All stage 2 translations for instruction accesses to Normal memory are Non-cacheable.
- When the value of HCR_EL2.DC is 1, all stage 1 translations and all accesses to the EL1&0 stage 1 translation tables, are treated as accesses to Normal Non-shareable Inner Write-Back Cacheable Read-Allocate Write-Allocate, Outer Write-Back Cacheable Read-Allocate Write-Allocate memory, regardless of the value of SCTL_EL1.{I, C}. This applies to translations for both data and instruction accesses.

Note

- The stage 1 and stage 2 cacheability attributes are combined as described in Combining stage 1 and stage 2 Cacheability attributes for Normal memory on page D8-5154.
- The SCTL_EL1.{C, I} and HCR_EL2.DC fields have no effect on the EL2, EL2&0, and EL3 translation regimes.
- The HCR_EL2.{ID, CD} fields affect only stage 2 of the EL1&0 translation regime.
- When EL2 is using AArch64 and EL1 is using AArch32, the HCR_EL2.{ID, CD, DC} controls apply as described here, but the EL1 controls are SCTL_EL1.{C, I}.
- When FEAT_XS is implemented, the SCTL_EL1.{C, I} and HCR_EL2.{ID, CD} fields have no effect on the value of the XS attribute.

For the EL2 translation regime

- When the value of SCTL_EL2.C is 0:
  - All data accesses to Normal memory using the EL2 translation regime are Non-cacheable.
  - All accesses to the EL2 translation tables are Non-cacheable.
- When the value of SCTL_EL2.I is 0:
  - All instruction accesses to Normal memory using the EL2 translation regime are Non-cacheable.
The AArch64 System Level Memory Model
D7.4 Cache support

--- Note ---

- The SCTLR_EL2.{I, C} fields have no effect on the EL1&0 and EL3 translation regimes.
- When FEAT_XS is implemented, the SCTLR_EL2.{I, C} fields have no effect on the value of the XS attribute.

--- For the EL2&0 translation regime ---

- When the value of SCTLR_EL2.C is 0:
  - All stage 1 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the EL2&0 stage 1 translation tables are Non-cacheable.
- When the value of SCTLR_EL2.I is 0:
  - All stage 1 translations for instruction accesses to Normal memory are Non-cacheable.

--- Note ---

When FEAT_XS is implemented, the SCTLR_EL2.{I, C} fields have no effect on the value of the XS attribute.

--- For the EL3 translation regime ---

- When the value of SCTLR_EL3.C is 0:
  - All data accesses to Normal memory using the EL3 translation regime are Non-cacheable.
  - All accesses to the EL3 translation tables are Non-cacheable.
- When the value of SCTLR_EL3.I is 0:
  - All instruction accesses to Normal memory using the EL3 translation regime are Non-cacheable.

--- Note ---

- The SCTLR_EL3{I, C} fields have no effect on the EL1&0, EL2, and EL2&0 translation regimes.
- When FEAT_XS is implemented, the SCTLR_EL3.{I, C} fields have no effect on the value of the XS attribute.

--- In addition ---

- For translation regimes other than the EL1&0 translation regime, if the value of SCTLR_ELx.M is 0, indicating that stage 1 translations are disabled for that translation regime, then:
  - If the value of SCTLR_ELx.I is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
  - If the value of SCTLR_ELx.I is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through cacheable, Outer Write-Through cacheable.
- For the EL1&0 translation regime, if the value of SCTLR_EL1.M is 0, indicating that stage 1 translations are disabled for that translation regime, and the value of HCR_EL2.DC is 0:
  - If the value of SCTLR_EL1.I is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
  - If the value of SCTLR_EL1.I is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through Cacheable, Outer Write-Through Cacheable.

The effect of SCTLR_ELx.C, HCR_EL2.DC and HCR_EL2.CD is reflected in the result of the address translation instructions in the PAR when these bits have an effect on the stages of translation being reported in the PAR.
D7.4 Cache support

D7.4.5 Behavior of caches at reset

The behavior of caches at reset is as follows:

- All caches reset to IMPLEMENTATION DEFINED states that might be UNKNOWN.
- The Cacheability control fields described in Enabling and disabling the caching of memory accesses on page D7-5047 reset to values that force all memory locations to be treated as Non-cacheable.

— Note —

This applies only to the controls that apply to the Translation regime that is used by the Exception level and Security state entered on reset.

- An implementation can require the use of a specific cache initialization routine to invalidate its storage array before caching is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, and the routine must be documented clearly as part of the documentation of the device.
- If an implementation permits cache hits when the Cacheability control fields force all memory locations to be treated as Non-cacheable then the cache initialization routine must:
  — Provide a mechanism to ensure the correct initialization of the caches.
  — Be documented clearly as part of the documentation of the device.

In particular, if an implementation permits cache hits when the Cacheability controls force all memory locations to be treated as Non-cacheable, and the cache contents are not invalidated at reset, the initialization routine must avoid any possibility of running from an uninitialized cache. It is acceptable for an initialization routine to require a fixed instruction sequence to be placed in a restricted range of memory.

- Arm recommends that whenever an invalidation routine is required, it is based on the cache maintenance instructions.

See also TLB behavior at reset on page D8-5193.

D7.4.6 Non-cacheable accesses and instruction caches

In AArch64 state, instruction accesses to Non-cacheable Normal memory can be held in instruction caches.

Correspondingly, the sequence for ensuring that modifications to instructions are available for execution must include invalidation of the modified locations from the instruction cache, even if the instructions are held in Normal Non-cacheable memory. This includes cases where System register Cacheability control fields force instruction accesses to memory to be Non-cacheable.

Therefore when using self-modified code in Non-cacheable space in a uniprocessor system, the following sequence is required:

; Enter this code with $t0 containing the new 32-bit instruction
; to be held at a location pointed to by $x0 in Normal Non-cacheable memory.
STR $t0, [$x0]
DSB ISH; Ensure visibility of the data stored
IC IVAU, [$x0]; Invalidate instruction cache by VA to PoU
DSB ISH; Ensure completion of the invalidations
ISB ;
In a multiprocessor system, the IC IVAU for a non-cacheable location is broadcast to all PEs within the Inner Shareable domain of the PE running this sequence. This is despite non-cacheable normal memory locations being treated as Outer Shared in other parts of the architecture.

Additional software steps might be required to synchronize the threads with other PEs. This might be necessary so that the PEs executing the modified instructions can execute an ISB after completing the invalidation, and to avoid issues associated with concurrent modification and execution of instruction sequences. See also Concurrent modification and execution of instructions on page B2-156 and Concurrent modification and execution of instructions on page E2-7166.

Larger blocks of instructions can be modified using the IC IALLU instruction for a uniprocessor system, or a IC IALLUIS for a multiprocessor system.

--- Note ---
This section applies even when the Cacheability control fields force instruction accesses to memory in AArch64 state to be Non-cacheable, as described in Enabling and disabling the caching of memory accesses on page D7-5047.

## D7.4.7 About cache maintenance in AArch64 state

The following sections give general information about cache maintenance:
- Terms used in describing the cache maintenance instructions.
- Abstraction of the cache hierarchy on page D7-5053.

The following sections describe the A64 cache maintenance instructions:
- The instruction cache maintenance instruction (IC) on page D7-5056.
- The data cache maintenance instruction (DC) on page D7-5056.

--- Note ---
Some descriptions of the cache maintenance instructions refer to the cacheability of the address on which the instruction operates. The Cacheability of an address is determined by the applicable translation table entry for that address, as modified by any applicable System register Cacheability controls, such as the SCTLR_EL1.{I, C} controls.

--- Terms used in describing the cache maintenance instructions ---

Cache maintenance instructions are defined to act on particular memory locations. Instruction scope can be defined:
- By the virtual address of the memory location to be maintained, referred to as operating by VA.
- By a mechanism that describes the location in the hardware of the cache, referred to as operating by set/way.

In addition, for instruction caches, there are instructions that invalidate all entries.

The following subsections define the terms used in the descriptions of the cache maintenance instructions:
- Terminology for cache maintenance instructions operating by set/way on page D7-5051.
- Terminology for Clean, Invalidate, and Clean and Invalidate instructions on page D7-5051.

--- Note ---
There is no terminology specific to cache maintenance instructions that operate by VA. When all applicable stages of translation are disabled, the VA used is identical to the PA. For more information about memory system behavior when address translation is disabled, see The effects of disabling an address translation stage on page D8-5118.
Terminology for cache maintenance instructions operating by set/way

Cache maintenance instruction that operate by set/way refer to the particular structures in a cache. Three parameters describe the location in a cache hierarchy that an instruction works on. These parameters are:

**Level**

The cache level of the hierarchy. The number of levels of cache is IMPLEMENTATION DEFINED. The cache levels that can be managed using the architected cache maintenance instructions that operate by set/way can be determined from the CLIDR_EL1.

In the Arm architecture, the lower numbered cache levels are those closest to the PE. See Memory hierarchy on page B2-184.

**Set**

Each level of a cache is split up into a number of sets. Each set is a set of locations in a cache level to which an address can be assigned. Usually, the set number is an IMPLEMENTATION DEFINED function of an address.

In the Arm architecture, sets are numbered from 0.

**Way**

The associativity of a cache is the number of locations in a set to which a specific address can be assigned. The way number specifies one of these locations.

In the Arm architecture, ways are numbered from 0.

--- **Note**

Because the allocation of a memory address to a cache location is entirely IMPLEMENTATION DEFINED, Arm expects that most portable software will use only the cache maintenance instructions by set/way as single steps in a routine to perform maintenance on the entire cache.

---

Terminology for Clean, Invalidate, and Clean and Invalidate instructions

Caches introduce coherency problems in two possible directions:

1. An update to a memory location by a PE that accesses a cache might not be visible to other observers that can access memory. This can occur because new updates are still in the cache and are not visible yet to the other observers that do not access that cache.

2. Updates to memory locations by other observers that can access memory might not be visible to a PE that accesses a cache. This can occur when the cache contains an old, or stale, copy of the memory location that has been updated.

The Clean and Invalidate instructions address these two issues. The definitions of these instructions are:

**Clean**

A cache clean instruction ensures that updates made by an observer that controls the cache are made visible to other observers that can access memory at the point to which the instruction is performed. Once the Clean has completed, the new memory values are guaranteed to be visible to the point to which the instruction is performed, for example to the Point of Unification.

The cleaning of a cache entry from a cache can overwrite memory that has been written by another observer only if the entry contains a location that has been written to by an observer in the shareability domain of that memory location.

**Invalidate**

A cache invalidate instruction ensures that updates made visible by observers that access memory at the point to which the invalidate is defined, are made visible to an observer that controls the cache. This might result in the loss of updates to the locations affected by the invalidate instruction that have been written by observers that access the cache, if those updates have not been cleaned from the cache since they were made.

If the address of an entry on which the invalidate instruction operates is Normal, Non-cacheable or any type of Device memory then an invalidate instruction also ensures that this address is not present in the cache.

--- **Note**

Entries for addresses that are Normal Cacheable can be allocated to the cache at any time, and so the cache invalidate instruction cannot ensure that the address is not present in a cache.
Clean and Invalidate

A cache **clean and invalidate** instruction behaves as the execution of a clean instruction followed immediately by an invalidate instruction. Both instructions are performed to the same location.

The points to which a cache maintenance instruction can be defined differ depending on whether the instruction operates by VA or by set/way:

- For instructions operating by set/way, the point is defined to be to the next level of caching. For the All operations, the point is defined as the Point of Unification for each location held in the cache.
- For instructions operating by VA, the following conceptual points are defined:

**Point of Coherency (PoC)**

The point at which all agents that can access memory are guaranteed to see the same copy of a memory location for accesses of any memory type or cacheability attribute. In many cases this is effectively the main system memory, although the architecture does not prohibit the implementation of caches beyond the PoC that have no effect on the coherency between memory system agents.

--- **Note** ---

The presence of system caches can affect the determination of the point of coherency as described in *System level caches* on page D7-5070.

**Point of Unification (PoU)**

The PoU for a PE is the point by which the instruction and data caches and the translation table walks of that PE are guaranteed to see the same copy of a memory location. In many cases, the Point of Unification is the point in a uniprocessor memory system by which the instruction and data caches and the translation table walks have merged.

The PoU for an Inner Shareable shareability domain is the point by which the instruction and data caches and the translation table walks of all the PEs in that Inner Shareable shareability domain are guaranteed to see the same copy of a memory location. Defining this point permits self-modifying software to ensure future instruction fetches are associated with the modified version of the software by using the standard correctness policy of:

1. Clean data cache entry by address.
2. Invalidate instruction cache entry by address.

**Point of Persistence (PoP)**

*When FEAT_DPB is implemented:*

The point in a memory system, if it exists, at or beyond the Point of Coherency, where a write to memory is maintained when system power is removed, and reliably recovered when power is restored to the affected locations in memory.

*When FEAT_DPB and FEAT_DPB2 are implemented:*

The point in a memory system where there is a system guarantee that there is sufficient energy within the system to ensure that a write to memory will be persistent if system power is removed.

--- **Note** ---

Such memory is sometimes called non-volatile memory. For example, the Storage-class memory shown in *Figure B2-1 on page B2-185* could be used as target memory for this feature.

**Point of Deep Persistence (PoDP)**

The point in a memory system where any writes that have reached that point are persistent, even in the event of an instantaneous hardware failure of the power system.
The following fields in the CLIDR_EL1 relate to the PoC and PoU:

**LoC, Level of Coherence**
This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Coherency. The LoC value is a cache level, so, for example, if LoC contains the value 3:
- A clean to the Point of Coherency operation requires the level 1, level 2 and level 3 caches to be cleaned.
- Level 4 cache is the first level that does not have to be maintained.
If the LoC field value is 0x0, this means that no levels of cache need to be cleaned or invalidated when cleaning or invalidating to the Point of Coherency.
If the LoC field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Coherency.

**LoUU, Level of Unification, uniprocessor**
This field defines the last level of data cache that must be cleaned, or the last level of instruction cache that must be invalidated, when cleaning or invalidating to the Point of Unification for the PE. As with LoC, the LoUU value is a cache level.
If the LoUU field value is 0x0, this means that no levels of data cache need to be cleaned or invalidated when cleaning or invalidating to the Point of Unification.
If the LoUU field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

**LoUIS, Level of Unification, Inner Shareable**
In any implementation:
- This field defines the last level of data or unified cache that must be cleaned, or the last level of instruction or unified cache that must be invalidated, when cleaning or invalidating to the Point of Unification for the Inner Shareable shareability domain. As with LoC, the LoUIS value is a cache level.
- If the LoUIS field value is 0x0, this means that no levels of data or unified cache need to be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable shareability domain.
- If the LoUIS field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

**Abstraction of the cache hierarchy**
The following subsections describe the abstraction of the cache hierarchy:
- *Cache maintenance instructions that operate by VA.*
- *Cache maintenance instructions that operate by set/way on page D7-5054.*

**Cache maintenance instructions that operate by VA**
The VA-based cache maintenance instructions are described as operating by VA. Each of these instructions is always qualified as being one of:
- Performed to the Point of Coherency.
- Performed to the Point of Unification.
- When FEAT_DPB is implemented, performed to the Point of Persistence.

See *Terms used in describing the cache maintenance instructions on page D7-5050* for definitions of these terms, and for more information about possible meanings of VA.

*A64 Cache maintenance instructions on page D7-5054* lists the VA-based maintenance instructions.

The CTR_EL0 holds minimum line length values for:
- The instruction caches.
- The data and unified caches.
These values support efficient invalidation of a range of VAs, because this value is the most efficient address stride to use to apply a sequence of VA-based maintenance instructions to a range of VAs.

For the Invalidate data or unified cache line by VA instruction, the Cache Write-back Granule field of the CTR_EL0 defines the maximum granule that a single invalidate instruction can invalidate. This meaning of the Cache Write-back Granule is in addition to its defining the maximum size that can be written back.

**Cache maintenance instructions that operate by set/way**

*AArch64 Cache maintenance instructions* lists the set/way-based maintenance instructions. Some encodings of these instructions include a required field that specifies the cache level for the instruction:

- A clean instruction cleans from the level of cache specified through to at least the next level of cache, moving further from the PE.
- An invalidate instruction invalidates only at the level specified.

**D7.4.8 A64 Cache maintenance instructions**

The A64 cache maintenance instructions are part of the A64 System instruction class in the register encoding space. For encoding details and other general information on these System instructions, see [System instructions on page C3-250, SYS on page C6-1993](#) and [Cache maintenance instructions, and data cache zero operation on page C5-659](#).

Table D7-3 on page D7-5054 shows the AArch64 System instructions that perform instruction or data cache maintenance. Instructions that take an argument include Xt in the entry in the *System instruction on page D7-5054* column.

<table>
<thead>
<tr>
<th><strong>Note</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>In Table D7-3 on page D7-5054 the Point of Unification is the Point of Unification of the PE executing the cache maintenance instruction.</td>
</tr>
<tr>
<td>In general, the AArch32 instruction and data cache maintenance instructions provide equivalent functionality to the AArch64 cache maintenance instructions, see <a href="#"><em>AArch32 cache and branch predictor maintenance instructions on page G4-9123</em></a>. However, the Data Cache Clean to the Point of Persistence instruction, implemented when FEAT_DPB is implemented, is supported in AArch64 state only.</td>
</tr>
</tbody>
</table>

### Table D7-3 System instructions for cache maintenance

<table>
<thead>
<tr>
<th><strong>System instruction</strong></th>
<th><strong>Instruction</strong></th>
<th><strong>Notes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction cache maintenance instructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IC IALLUIS</td>
<td>Invalidate all to Point of Unification, Inner Shareable</td>
<td>EL1 or higher access.</td>
</tr>
<tr>
<td>IC IALU</td>
<td>Invalidate all to Point of Unification</td>
<td>EL1 or higher access.</td>
</tr>
<tr>
<td>IC IVAU, Xt</td>
<td>Invalidate by virtual address to Point of Unification</td>
<td>When SCLR_EL1.UCl !== 1, EL0 access. Otherwise, EL1 or higher access.</td>
</tr>
<tr>
<td>Data cache maintenance instructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC IVAC, Xt</td>
<td>Invalidate by virtual address to Point of Coherency</td>
<td>EL1 or higher access.</td>
</tr>
<tr>
<td>DC IGVC, Xt</td>
<td>Invalidate of Allocation Tags by virtual address to Point of Coherency</td>
<td>EL1 or higher access.</td>
</tr>
<tr>
<td>DC IGDVAC, Xt</td>
<td>Invalidate of data and Allocation Tags by virtual address to Point of Coherency</td>
<td>EL1 or higher access.</td>
</tr>
</tbody>
</table>
A DSB or DMB instruction intended to ensure the completion of cache or branch predictor maintenance instructions must have an access type of both loads and stores.

The following subsections give more information about these instructions:

- The instruction cache maintenance instruction (IC) on page D7-5056.
- The data cache maintenance instruction (DC) on page D7-5056.
- EL0 accessibility of cache maintenance instructions on page D7-5058.
- General requirements for the scope of maintenance instructions on page D7-5059.
- Effects of instructions that operate by VA to the PoC on page D7-5059.
- Effects of instructions that operate by VA to the PoP on page D7-5060.

### Table D7-3 System instructions for cache maintenance (continued)

<table>
<thead>
<tr>
<th>System instruction</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC ISW, Xt</td>
<td>Invalidate by set/way</td>
<td>EL1 or higher access.</td>
</tr>
<tr>
<td>DC CVAC, Xt</td>
<td>Clean by virtual address to Point of Coherency</td>
<td>When <code>SCTRL_EL1.UCIa == 1</code>, EL0 access. Otherwise EL1 or higher access.</td>
</tr>
<tr>
<td>DC CCVAC, Xt</td>
<td>Clean of Allocation Tags by virtual address to Point of Coherency</td>
<td>When <code>SCTRL_EL1.UCIa == 1</code>, EL0 access. Otherwise EL1 or higher access.</td>
</tr>
<tr>
<td>DC CGDVAC, Xt</td>
<td>Clean of data and Allocation Tags by virtual address to Point of Coherency</td>
<td>When <code>SCTRL_EL1.UCIa == 1</code>, EL0 access. Otherwise EL1 or higher access.</td>
</tr>
<tr>
<td>DC CVADP, Xt</td>
<td>Clean by virtual address to Point of Deep Persistence</td>
<td>When <code>SCTRL_EL1.UCIa == 1</code>, EL0 access. Otherwise EL1 or higher access.</td>
</tr>
<tr>
<td>DC CCVADP, Xt</td>
<td>Clean of Allocation Tags by virtual address to Point of Deep Persistence</td>
<td>When <code>SCTRL_EL1.UCIa == 1</code>, EL0 access. Otherwise EL1 or higher access.</td>
</tr>
<tr>
<td>DC CGDVADP, Xt</td>
<td>Clean of data and Allocation Tags by virtual address to Point of Deep Persistence</td>
<td>When <code>SCTRL_EL1.UCIa == 1</code>, EL0 access. Otherwise EL1 or higher access.</td>
</tr>
<tr>
<td>DC CGVAP, Xt</td>
<td>Clean of Allocation Tags by virtual address to Point of Persistence</td>
<td>When <code>SCTRL_EL1.UCIa == 1</code>, EL0 access. Otherwise EL1 or higher access.</td>
</tr>
<tr>
<td>DC CVAP, Xt</td>
<td>Clean by virtual address to Point of Persistence (^b)</td>
<td>When <code>SCTRL_EL1.UCIa == 1</code>, EL0 access. Otherwise EL1 or higher access.</td>
</tr>
<tr>
<td>DC CSW, Xt</td>
<td>Clean by set/way</td>
<td>EL1 or higher access.</td>
</tr>
<tr>
<td>DC CVAU, Xt</td>
<td>Clean by virtual address to Point of Unification</td>
<td>When <code>SCTRL_EL1.UCIa == 1</code>, EL0 access. Otherwise EL1 or higher access.</td>
</tr>
<tr>
<td>DC CIVAC, Xt</td>
<td>Clean and invalidate by virtual address to Point of Coherency</td>
<td>When <code>SCTRL_EL1.UCIa == 1</code>, EL0 access. Otherwise EL1 or higher access.</td>
</tr>
<tr>
<td>DC CIGVAC, Xt</td>
<td>Clean and invalidate of Allocation Tags by virtual address to Point of Coherency</td>
<td>When <code>SCTRL_EL1.UCIa == 1</code>, EL0 access. Otherwise EL1 or higher access.</td>
</tr>
<tr>
<td>DC CIGDVAC, Xt</td>
<td>Clean and invalidate of data and Allocation Tags by virtual address to Point of Coherency</td>
<td>When <code>SCTRL_EL1.UCIa == 1</code>, EL0 access. Otherwise EL1 or higher access.</td>
</tr>
<tr>
<td>DC CISW, Xt</td>
<td>Clean and invalidate by set/way</td>
<td>EL1 or higher access.</td>
</tr>
</tbody>
</table>

\(^a\) When HCR_EL2.{E2H,TGE} == {1, 1}, the control is from SCTRL_EL2.
\(^b\) Supported only when FEAT_DPB is implemented.
• Effects of instructions that operate by VA to the PoU on page D7-5060.
• Effects of All and set/way maintenance instructions on page D7-5061.
• Effects of virtualization and Security state on the cache maintenance instructions on page D7-5061.
• Boundary conditions for cache maintenance instructions on page D7-5063.
• Ordering and completion of data and instruction cache instructions on page D7-5063.
• Performing cache maintenance instructions on page D7-5066.

The instruction cache maintenance instruction (IC)

System instructions on page C3-250 describes the A64 assembly syntax for this instruction.

When an IC instruction requires an address argument this takes the form of a 64-bit register that holds the VA argument. No alignment restrictions apply for this address.

Any cache maintenance instruction operating by VA includes as part of any required VA to PA translation:
• For an instruction executed at EL1, or at EL2 when HCR_EL2.E2H==1, the current ASID.
• The current Security state.
• Whether the instruction was executed at EL1 or EL2.
• For an instruction executed at EL1, the current VMID.

That VA to PA translation might fault. However, for an instruction cache maintenance instruction that operates by VA:
• It is IMPLEMENTATION DEFINED whether the instruction can generate:
  — An Access flag fault.
  — A Translation fault.
• The instruction cannot generate a Permission fault, except for:
  — The possible generation of a Permission fault by the execution of an IC IVAU instruction at EL0 when the specified address does not have read access at EL0, as described in EL0 accessibility of cache maintenance instructions on page D7-5058.
  — When FEAT_CMOW is implemented, the possible generation of a Permission fault by:
    — The execution of an IC IVAU instruction at EL0 when the specified address has stage 1 read permission, but does not have stage 1 write permission.
    — The execution of an IC IVAU instruction at EL1 or EL0 when the specified address has stage 2 read permission, but does not have stage 2 write permission.
    — The possible Permission fault on a Stage 2 fault on a stage 1 translation table walk.

For more information about possible faults on a cache maintenance instruction that operates by VA see Memory aborts on page D8-5180.

See also Ordering and completion of data and instruction cache instructions on page D7-5063.

The data cache maintenance instruction (DC)

System instructions on page C3-250 describes the A64 assembly syntax for this instruction.

When a DC instruction requires a set/way/level argument this takes the form of a 64-bit register, the upper 32 bits of which are RES0.

If a data cache maintenance by set/way instruction specifies a set, way, or level argument that is larger than the value supported by the implementation then the instruction is CONSTRAINED UNPREDICTABLE, see Out of range values of the Set/Way/Index fields in cache maintenance instructions on page K1-11600 or the instruction description.

When a DC instruction requires an address argument this takes the form of a 64-bit register that holds the VA argument. No alignment restrictions apply for this address.
Any cache maintenance instruction operating by VA includes as part of any required VA to PA translation:

- For an instruction executed at EL1, or at EL2 when HCR_EL2.E2H is 1, the current ASID.
- The current Security state.
- Whether the instruction is executed at EL1 or EL2.
- For an instruction executed at EL1, the current VMID.

That VA to PA translation might fault. However, a data or unified cache maintenance instruction that operates by VA cannot generate a Permission fault except in the following cases:

- The possible generation of a Permission fault by:
  - The execution of a `DC IVAC` instruction when the specified address does not have write permission.
  - The execution of an enabled DC * instruction at EL0 when the specified address does not have read access at EL0, as described in EL0 accessibility of cache maintenance instructions on page D7-5058.

The description of Permission faults includes possible constraints on the generation of Permission faults on cache maintenance by VA instructions.

- When FEAT_CMOW is implemented, the possible generation of a Permission fault by:
  - The execution of a `DC CIVAC`, `DC CIGVAC`, or `DC CIGDVAC` instruction at EL0 when the specified address has stage 1 read permission, but does not have stage 1 write permission.
  - The execution of a `DC CIVAC`, `DC CIGVAC`, or, `DC CIGDVAC` instruction at EL1 or EL0 when the specified address has stage 2 read permission, but does not have stage 2 write permission.

- The possible Permission fault on a Stage 2 fault on a stage 1 translation table walk.

For more information about possible faults on a VA to PA translation see Memory aborts on page D8-5180.

When executed at EL1, a `DC ISW` instruction performs a clean and invalidate, meaning it performs the same maintenance as a `DC CISW` instruction, if all of the following apply:

- EL2 is implemented and enabled in the current Security state.
- Either:
  - The value of HCR_EL2.SWIO is 1, forcing a cache clean to perform a clean and invalidate.
  - The value of HCR_EL2.VM is 1, meaning EL1&0 stage two address translation is enabled.

When executed at EL1, a `DC IVAC` instruction performs a clean and invalidate, meaning it performs the same maintenance as a `DC CIVAC` instruction, if all of the following apply:

- EL2 is implemented and enabled in the current Security state.
- The value of HCR_EL2.VM is 1, meaning EL1&0 stage two address translation is enabled.

-- Note --

The forcing of a clean instruction to perform a clean invalidate applies to the AArch32 cache maintenance instructions `DCIWAC` and `DCISW`. See AArch32 data cache maintenance instructions (DC*) on page G4-9125.

-- Note --

When FEAT_DPB is implemented, meaning the `DC CVAP` instruction is implemented, if the memory system does not support the Point of Persistence, a data cache clean to the PoP, `DC CVAP`, behaves as a data cache clean to the PoC, `DC CVAC`.

-- Note --

- Support for the Point of Persistence does not change the definition or behavior of the CLIDR_EL1 System register.
- Because a DSB SYS instruction will not complete until all previous DC CVAP instructions have completed, the following sequence can be used to ensure the completion of any store to the Point of Persistence, where the store might be to Non-cacheable memory:
  ```
  DMB ; Note this can be any DMB that applies to both loads and stores
  DC CVAP , Xt
  DSB SYS
  ```
If caches that are invisible to the programmer exist beyond the Point of Coherency but before the Point of Persistence and hold data that is marked as Non-cacheable, the **DC CVAP** operation causes the Non-cacheable locations to be cleaned from those caches.

If a memory fault that sets the **FAR** for the translation regime applicable for the cache maintenance instruction is generated from a data cache maintenance instruction, the **FAR** holds the address specified in the register argument of the instruction.

**Note**

Despite its mnemonic, **DC ZVA** is not a cache maintenance instruction.

See also [EL0 accessibility of cache maintenance instructions](#) and [Ordering and completion of data and instruction cache instructions](#) on page D7-5063.

### EL0 accessibility of cache maintenance instructions

The **SCTLR_EL1.UCI** bit enables EL0 access for the **DC CVAU, DC CVAC, DC CVAP, DC CIVAC, DC CIVAC, DC CIGVAC, DC CIGDVAC**, and **IC IVAU** instructions. When EL0 use of these instructions is disabled because **SCTLR_EL1.UCI** == 0, executing one of these instructions at EL0 generates a trap to EL1, that is reported using **EC** = 0x18. When **HCR_EL2.{E2H,TGE}** == {1,1}, the control is from **SCTLR_EL2**.

**Note**

**DC CVAP** is implemented only if **FEAT_DPB** is implemented.

For these instructions read access permission is required. When the value of **SCTLR_EL1.UCI** is 1:

- For the **DC CVAU, DC CVAC, DC CVAP, DC CIVAC, DC CIGVAC**, and **DC CIGDVAC**, instructions, if the instruction is executed at EL0 and the address specified in the argument cannot be read at EL0, a **Permission fault** might be generated.

- For the **IC IVAU** instruction, if the instruction is executed at EL0 and the address specified in the argument cannot be read at EL0, it is **IMPLEMENTATION DEFINED** whether a **Permission fault** is generated.

- When **FEAT_CMOW** is implemented, for the **DC CIVAC, DC CIGVAC, DC CIGDVAC**, and **IC IVAU** instructions, both read and write access permission is required:
  - When **SCTLR_EL1.CMOW** is 1, if the instruction executed at EL0 has stage 1 read permission, but does not have stage 1 write permission, a **Permission fault** is generated.
  - When **HCRX_EL2.CMOW** is 1, if the instruction executed at EL0 has stage 2 read permission, but does not have stage 2 write permission, a **Permission fault** is generated.

  **Note**
  
  This stage 2 access permission also applies to the **DCCIMVAC** and **ICIMVAU** AArch32 cache maintenance instructions.

For more information see the description of **Permission faults**, In the case of a **DC */* instruction executed at EL0 when the address specified cannot be read at EL0 the **Permission fault** is generated unless one of the permitted constraints described in that section applies and means the fault cannot be generated.

Software can read the **CTR_EL0** to discover the stride needed for cache maintenance instructions. The **SCTLR_EL1.UCT** bit enables EL0 access to the **CTR_EL0**. When EL0 access to the Cache Type register is disabled, a register access instruction executed at EL0 is trapped to EL1 using **EC** = 0x18.
General requirements for the scope of maintenance instructions

The specification of the cache maintenance instructions describes what each instruction is guaranteed to do in a system. It does not limit other behaviors that might occur, provided they are consistent with the requirements described in General behavior of the caches on page D7-5043, Behavior of caches at reset on page D7-5049, and Preloading caches on page B2-188.

This means that as a side-effect of a cache maintenance instruction:

- Any location in the cache might be cleaned.
- Any unlocked location in the cache might be cleaned and invalidated.

Note

Arm recommends that, for best performance, such side-effects are kept to a minimum. Arm strongly recommends that the side-effects of operations performed in Non-secure state do not have a significant performance impact on execution in Secure state.

If an implementation can overwrite Allocation Tags in memory that have been written by another observer, where the Allocation Tags have not been written by an observer in the Shareability domain of that memory location, then:

- A cache maintenance operation which cleans data from a cache level must also clean the associated Allocation Tags.
- A cache maintenance operation which invalidates, or cleans and invalidates data from a cache level, must also clean and invalidate the associated Allocation Tags.

Effects of instructions that operate by VA to the PoC

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, cache maintenance instructions that operate by VA to the PoC must affect the caches of other PEs in the shareability domain described by the shareability attributes of the VA supplied with the instruction.

For Device memory and Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions must affect the caches of all PEs in the Outer Shareable shareability domain of the PE on which the instruction is operating.

In all cases, for any affected PE, these instructions affect all data and unified caches to the PoC. Table D7-4 on page D7-5059 shows the scope of these Data and unified cache maintenance instructions.

<table>
<thead>
<tr>
<th>Shareability</th>
<th>PEs affected</th>
<th>Effective to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-shareable</td>
<td>The PE executing the instruction</td>
<td>The PoC of the entire system</td>
</tr>
<tr>
<td>Inner Shareable</td>
<td>All PEs in the same Inner Shareable shareability domain as the PE executing the instruction</td>
<td>The PoC of the entire system</td>
</tr>
<tr>
<td>Outer Shareable</td>
<td>All PEs in the same Outer Shareable shareability domain as the PE executing the instruction</td>
<td>The PoC of the entire system</td>
</tr>
</tbody>
</table>

Note

It is IMPLEMENTATION DEFINED by the system whether the cache maintenance instructions have an effect on the caches of observers that are not PEs within the affected shareability domain to which the cache maintenance instructions apply.
Effects of instructions that operate by VA to the PoP

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, cache maintenance instructions that operate by VA to the PoP must affect the caches of other PEs in the shareability domain described by the shareability attributes of the VA supplied with the instruction.

For Device memory and Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions must affect the caches of all PEs in the Outer Shareable shareability domain of the PE on which the instruction is operating.

In all cases, for any affected PE, these instructions affect all data and unified caches to the PoP. Table D7-5 on page D7-5060 shows the scope of these Data and unified cache maintenance to the PoP instructions.

Table D7-5 PEs affected by cache maintenance instructions to the PoP

<table>
<thead>
<tr>
<th>Shareability</th>
<th>PEs affected</th>
<th>Effective to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-shareable</td>
<td>The PE executing the instruction</td>
<td>The PoP of the entire system</td>
</tr>
<tr>
<td>Inner Shareable</td>
<td>All PEs in the same Inner Shareable shareability domain as the PE executing the instruction</td>
<td>The PoP of the entire system</td>
</tr>
<tr>
<td>Outer Shareable</td>
<td>All PEs in the same Outer Shareable shareability domain as the PE executing the instruction</td>
<td>The PoP of the entire system</td>
</tr>
</tbody>
</table>

Note

It is IMPLEMENTATION DEFINED by the system whether the cache maintenance instructions have an effect on the caches of observers that are not PEs within the affected shareability domain to which the cache maintenance instructions apply.

Effects of instructions that operate by VA to the PoU

For cache maintenance instructions that operate by VA to the PoU, Table D7-6 on page D7-5060 shows how, for a VA in a Normal or Device memory location, the shareability attribute of the VA determines the minimum set of PEs affected, and the point to which the instruction must be effective.

Table D7-6 PEs affected by cache maintenance instructions to the PoU

<table>
<thead>
<tr>
<th>Shareability</th>
<th>PEs affected</th>
<th>Effective to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-shareable</td>
<td>The PE executing the instruction</td>
<td>The PoU of instruction cache fills, data cache fills and write-backs, and translation table walks, on the PE executing the instruction</td>
</tr>
<tr>
<td>Inner Shareable or Outer Shareable</td>
<td>All PEs in the same Inner Shareable shareability domain as the PE executing the instruction</td>
<td>The PoU of instruction cache fills, data cache fills and write-backs, and translation table walks, of all PEs in the same Inner Shareable shareability domain as the PE executing the instruction</td>
</tr>
</tbody>
</table>

Note

- The set of PEs guaranteed to be affected is never greater than the PEs in the Inner Shareable shareability domain containing the PE executing the instruction.
- It is IMPLEMENTATION DEFINED by the system whether the cache maintenance instructions have an effect on the caches of observers that are not PEs within the affected shareability domain to which the cache maintenance instructions apply.
Effects of All and set/way maintenance instructions

The \texttt{IC IALLU} and \texttt{DC} set/way instructions apply only to the caches of the PE that performs the instruction.

The \texttt{IC IALLUIS} instruction can affect the caches of all PEs in the same Inner Shareable shareability domain as the PE that performs the instruction. This instruction has an effect to the Point of Unification of instruction cache fills, data cache fills, and write-backs, and translation table walks, of all PEs in the same Inner Shareable shareability domain.

\begin{note}
- The possible presence of system caches, as described in \textit{System level caches} on page D7-5070, means architecture does not guarantee that all levels of the cache can be maintained using set/way instructions.
- It is \textsc{IMPLEMENTATION DEFINED} by the system whether the cache maintenance instructions have an effect on the caches of observers that are not PEs within the affected shareability domain to which the cache maintenance instructions apply.
\end{note}

Effects of virtualization and Security state on the cache maintenance instructions

Each Security state has its own physical address (PA) space, therefore cache entries are associated with PA space. \textbf{Table D7-7} on page D7-5061 shows the effects of virtualization and security on the cache maintenance instructions. In the table, the \textit{Specified entries} on page D7-5061 are entries that the architecture requires the instruction to affect.

\begin{note}
The rules described in \textit{General behavior of the caches} on page D7-5043 mean that an instruction might also affect other entries.
\end{note}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{Cache maintenance instructions} & \textbf{Security state} & \textbf{Specified entries} \\
\hline
Invalidate, Clean, or Clean and Invalidate by VA: \\
\texttt{DC IVAC, DC CVAC, DC CVAP, DC CVAU, DC CIVAC, DC CVAP} & Both & All lines that hold the PA that, in the current Security state, is mapped to by the combination of all of: \\
& & \begin{itemize}
  \item The specified VA.
  \item For an instruction executed at EL1, EL0, or at EL2 when HCR\textsubscript{EL2}.E2H is set to 1 the current ASID if the location is mapped to by a non-global page.
  \item For an instruction executed at EL1 when SCR\textsubscript{EL3}.NS == 1 or SCR\textsubscript{EL3}.EEL2 == 1, the current VMID.\textsuperscript{*}
  \item For an instruction executed at EL0 when (SCR\textsubscript{EL3}.NS == 1 or SCR\textsubscript{EL3}.EEL2 == 1) and (HCR\textsubscript{EL2}.E2H == 0 or HCR\textsubscript{EL2}.TGE == 0), the current VMID.\textsuperscript{*}
\end{itemize} \\
\hline
Invalidate, Clean, or Clean and Invalidate by set/way: \\
\texttt{DC ISW, DC CSW, DC CISW} & Non-secure & Line specified by set/way provided that the entry comes from the Non-secure PA space. \\
& Secure & Line specified by set/way regardless of the PA space that the entry has come from. \\
\hline
\end{tabular}
\caption{Effects of virtualization and security on the maintenance instructions}
\end{table}
For locked entries and entries that might be locked, the behavior of cache maintenance instructions described in *The interaction of cache lockdown with cache maintenance instructions* on page D7-5069 applies.

With an implementation that generates aborts if entries are locked or might be locked in the cache, when the use of lockdown aborts is enabled, these aborts can occur on any cache maintenance instructions.

In an implementation that includes EL2:

- The architecture does not require cache cleaning when switching between virtual machines. Cache invalidation by set/way must not present an opportunity for one virtual machine to corrupt state associated with a second virtual machine. To ensure this requirement is met, invalidate by set/way instructions can, instead, perform a clean and invalidate by set/way.

- As described in *The data cache maintenance instruction (DC)* on page D7-5056, the AArch64 Data cache invalidate instructions, DC _IVAC and DC _ISW, when executed at EL1 and EL0, and the AArch32 Data cache invalidate instructions DCIMVAC and DCISW, when executed at EL1, can be configured to perform a cache clean as well as a cache invalidation.

- TLB and instruction cache invalidate instructions executed at EL1 are broadcast across the Inner Shareable domain when all of the following is true:
  - When the value of HCR_EL2.FB is 1.
— EL3 is not implemented, or EL3 is implemented and either SCR_EL3.NS == 1 or SCR_EL3.EEL2 == 1.

When EL1 is using AArch64, this applies to the IC IALLU instruction. This means the instruction performs the invalidation that would be performed by the corresponding Inner Shareable instruction IC IALLUIS.

For more information about the cache maintenance instructions, see About cache maintenance in AArch64 state on page D7-5050, A64 Cache maintenance instructions on page D7-5054, and Chapter D8 The AArch64 Virtual Memory System Architecture.

Boundary conditions for cache maintenance instructions

Cache maintenance instructions operate on the caches regardless of whether the System register Cacheability controls force all memory accesses to be Non-cacheable.

For VA-based cache maintenance instructions, the instruction operates on the caches regardless of the memory type and cacheability attributes marked for the memory address in the VMSA translation table entries. This means that the effects of the cache maintenance instructions can apply regardless of:

- Whether the address accessed:
  — Is Normal memory or Device memory.
  — Has the Cacheable attribute or the Non-cacheable attribute.
- Any applicable domain control of the address accessed.
- The access permissions for the address accessed, other than the effect of the stage two write permission on data or unified cache invalidation instructions.

Ordering and completion of data and instruction cache instructions

All data cache instructions, other than DC ZVA, DC GVA, and DC GZVA that specify an address:

- Execute in program order relative to loads or stores that have all of the following properties:
  — Access an address in Normal memory with either Inner Write Through or Inner Write Back attributes within the same cache line of minimum size, as indicated by CTR_EL0.DMinLine.
  — Use an address with the same cacheability attributes as the address passed to the data cache instruction.
- Can execute in any order relative to loads or stores that have all of the following properties:
  — Access an address in Normal memory with either Inner Write Through or Inner Write Back attributes within the same cache line of minimum size, as indicated by CTR_EL0.DMinLine.
  — Use an address with different cacheability attributes as the address passed to the data cache instruction.
  — Do not have a DMB or DSB executed between the load or store instruction and the data cache instruction.
- Can execute in any order relative to other data cache instructions, other than DC ZVA, DC GVA, and DC GZVA that specify an address within the same cache line of minimum size, as indicated by CTR_EL0.DMinLine.
- Can execute in any order relative to other data cache instructions, other than DC ZVA, DC GVA, and DC GZVA that specify an address in a different cache line of minimum size, as indicated by CTR_EL0.DMinLine, unless a DMB or DSB is executed between the instructions.
- Can execute in any order relative to other data cache instructions, other than DC ZVA, DC GVA, and DC GZVA that specify an address in a different cache line of minimum size, as indicated by CTR_EL0.DMinLine, unless a DMB or DSB is executed between the instructions.
- Can execute in any order relative to instruction cache maintenance instructions unless a DSB is executed between the instructions.
• Can execute in any order relative to data cache maintenance instructions that do not specify an address unless a DMB or DSB is executed between the instructions.

--- Note ---
Despite their mnemonics, the DC ZVA, DC GVA, and DC GZVA instructions are not data cache maintenance instructions.

--- Note ---
• Data cache ordering rules by address are consistent with physically indexed physically tagged caches. See Data and unified caches on page D8-5215.
• Data cache zero instruction on page D7-5068 describes the ordering and completion rules for Data Cache Zero.

All data cache maintenance instructions that do not specify an address:
• Can execute in any order relative to data cache maintenance instructions that do not specify an address unless a DMB or DSB is executed between the instructions.
• Can execute in any order relative to data cache maintenance instructions that specify an address, other than DC ZVA, DC GVA, and DC GZVA, unless a DMB or DSB is executed between the instructions.
• Can execute in any order relative to loads or stores unless a DMB or DSB is executed between the instructions.
• Can execute in any order relative to instruction cache maintenance instructions unless a DSB is executed between the instructions.

All instruction cache maintenance instructions can execute in any order relative to other instruction cache instructions, data cache instructions, loads, and stores unless a DSB is executed between the instructions.

A cache maintenance instruction can complete at any time after it is executed, but is only guaranteed to be complete, and its effects visible to other observers, following a DSB instruction executed by the PE that executed the cache maintenance instruction. See also the requirements for cache maintenance instructions in Completion and endpoint ordering on page B2-168.

In all cases, where the text in this section refers to a DMB or a DSB, this means a DMB or DSB whose required access type is both loads and stores.

--- Note ---
These ordering requirements are extended from the requirements in AArch32 state given in:
• Ordering of cache and branch predictor maintenance instructions on page G4-9132.
• AArch32 instruction cache maintenance instructions (IC*) on page G4-9124.

Ordering and completion of Data Cache Clean to Point of Persistence

The update of the persistent memory as a result of Data Cache Clean to the Point of Persistence is guaranteed to have occurred either after:
• The execution of a DSB applying to both reads and writes after the execution of the Data Cache Clean to the Point of Persistence.
• The update to persistent memory caused by a different Data Cache Clean to the Point of Persistence that is ordered after a DMB applying to both reads and writes that appears after the original Data Cache Clean to the Point of Persistence.

--- Note ---
This second point is an aspect of the fact that the Data Cache Clean to the Point of Persistence instructions are ordered by DMB, and this controls the order of arrival in persistent memory.
The ordering effect for the Data Cache Clean to the Point of Persistence by DMB applying to both read and writes is not sufficient to ensure that in Example D7-1, observation of the value '1' in the memory location X3 implies that the Data Cache Clean to the Point of Persistence has caused an update of persistent memory:

**Example D7-1 The ordering effect for the Data Cache Clean to the Point of Persistence**

; initial condition has [X3]=0.
DC_CVADP, X1
DMB
MOV X2, #1
STR X2, [X3]

However, in Example D7-2, the ordering effects of the DMB instruction will ensure that the location pointed by P0: X1 will reach the Point of Persistence before, or at the same time as, the location pointed by P1:X8.

**Example D7-2 The ordering effect for the Data Cache Clean to the Point of Persistence**

; initial conditions has P0: X3 and P1: X3 point to the same location, which is 0 at the start of this example
P0
DC_CVAP, X1
DMB
MOV X2, #1
STR X2, [X3]
P1
loop
LDR X2, [X3]
CBZ X2, loop
DMB
DC_CVAP, X8

**Ordering and completion of Data Cache Clean to Point of Deep Persistence**

The update of the deep persistent memory as a result of Data Cache Clean to the Point of Deep Persistence is guaranteed to have occurred either after:

- The execution of a DSB applying to both reads and writes after the execution of the Data Cache Clean to the Point of Deep Persistence.
- The update to deep persistent memory caused by a different Data Cache Clean to the Point of Deep Persistence that is ordered after a DMB applying to both reads and writes that appears after the original Data Cache Clean to the Point of Deep Persistence.

**Note**

This second point is an aspect of the fact that the Data Cache Clean to the Point of Deep Persistence instructions are ordered by DMB, and this controls the order of arrival in deep persistent memory.

**Note**

The ordering effect for the Data Cache Clean to the Point of Deep Persistence by DMB applying to both read and writes is not sufficient to ensure that in Example D7-3, observation of the value '1' in the memory location X3 implies that the Data Cache Clean to the Point of Deep Persistence has caused an update of deep persistent memory:
Example D7-3 The ordering effect for the Data Cache Clean to the Point of Deep Persistence

; initial conditions has [X3]=0.
DC CVADP, X1
DMB
MOV X2, #1
STR X2, [X3]

However, in Example D7-4, the ordering effects of the DMB instruction will ensure that the location pointed by P0: X1 will reach the Point of Deep Persistence before, or at the same time as, the location pointed by P1: X8.

Example D7-4 The ordering effect for the Data Cache Clean to the Point of Deep Persistence

; initial conditions has P0: X3 and P1: X3 point to the same location, which is 0 at the start of this example
P0
DC CVADP, X1
DMB
MOV X2, #1
STR X2, [X3]
P1
loop
LDR X2, [X3]
CBZ X2, loop
DMB
DC CVADP, X8

Performing cache maintenance instructions

To ensure all cache lines in a block of address space are maintained through all levels of cache Arm strongly recommends that software:

• For data or unified cache maintenance, uses the CTR_EL0.DMinLine value to determine the loop increment size for a loop of data cache maintenance by VA instructions.
• For instruction cache maintenance, uses the CTR_EL0.IMinLine value to determine the loop increment size for a loop of instruction cache maintenance by VA instructions.

Example code for cache maintenance instructions

The cache maintenance instructions by set/way can clean or invalidate, or both, the entirety of one or more levels of cache attached to a PE. However, unless all PEs attached to the caches regard all memory locations as Non-cacheable, it is not possible to prevent locations being allocated into the cache during such a sequence of the cache maintenance instructions.

Note

Since the set/way instructions are performed only locally, there is no guarantee of the atomicity of cache maintenance between different PEs, even if those different PEs are each executing the same cache maintenance instructions at the same time. Since any cacheable line can be allocated into the cache at any time, it is possible for a cache line to migrate from an entry in the cache of one PE to the cache of a different PE in a way that means the line is not affected by set/way based cache maintenance. Therefore, Arm strongly discourages the use of set/way
instructions to manage coherency in coherent systems. The expected use of the cache maintenance instructions that operate by set.way is limited to the cache maintenance associated with the powerdown and powerup of caches, if this is required by the implementation.

The limitations of cache maintenance by set/way mean maintenance by set/way does not happen on multiple PEs, and cannot be made to happen atomically for each address on each PE. Therefore in multiprocessor or multithreaded systems, the use of cache maintenance by set/way to clean, or clean and invalidate, the entire cache for coherency management with very large buffers or with buffers with unknown address can fail to provide the expected coherency results because of speculation by other PEs, or possibly by other threads. The only way that these instructions can be used in this way is to first ensure that all PEs that might cause speculative accesses to caches that need to be maintained are not capable of generating speculative accesses. This can be achieved by ensuring that those PEs have no memory locations with a Normal Cacheable attribute. Such an approach can have very large system performance effects, and Arm advises implementers to use hardware coherency mechanisms in systems where this will be an issue.

**System level caches on page D7-5070** refers to other limitations of cache maintenance by set/way.

The following example code for cleaning a data or unified cache to the Point of Coherency illustrates a generic mechanism for cleaning the entire data or unified cache to the Point of Coherency. It assumes that the current Cache Size Identification Register is in 32-bit format. For more information, see Possible formats of the Cache Size Identification Register, CCSIDR_EL1 on page D7-5045.

```
MRS     X0, CLIDR_EL1
AND     W3, W0, #0x07000000     // Get 2 x Level of Coherence
LSR     W3, W3, #23
CEZ     W3, Finished
MOV     W10, #0                  // W10 = 2 x cache level
MOV     W8, #1                   // W8 = constant 0b1

Loop1:  ADD     W2, W10, W10, LSR #1    // Calculate 3 x cache level
        LSR     W1, W0, W2              // extract 3-bit cache type for this level
        AND     W1, W1, #0x7            // W1 = constant 0x7
        CMP     W1, #2
        B.LT    Skip                    // No data or unified cache at this level
        MSR     CSSELR_EL1, X10         // Select this cache level
        ISB                             // Synchronize change of CSSELR
        MRS     X1, CCSIDR_EL1          // Read CCSIDR
        AND     W2, W1, #7              // W2 = log2(linelen)-4
        ADD     W2, W2, #4              // W2 = log2(linelen)
        UBFX    W4, W1, #3, #10         // W4 = max way number, right aligned
        CLZ     W5, W4                  // W5 = 32-log2(ways), bit position of way in DC operand
        LSL     W9, W4, W5              // W9 = max way number, aligned to position in DC operand
        LSL     W16, W8, W5             // W16 = amount to decrement way number per iteration

Loop2:  UBFX    W7, W1, #13, #15        // W7 = max set number, right aligned
        LSL     W7, W7, W2              // W7 = max set number, aligned to position in DC operand
        LSL     W17, W8, W2             // W17 = amount to decrement set number per iteration

Loop3:  ORR     W11, W10, W9           // W11 = combine way number and cache number...
        ORR     W11, W11, W7            // ... and set number for DC operand
        DC      CSW, X11                // Do data cache clean by set and way
        B.GE    Loop3
        SUBS    W7, W7, W17             // Decrement set number
        B.GE    Loop2
        SUBS    X9, X9, X16             // Decrement way number

Skip:   ADD     W10, W10, #2          // Increment 2 x cache level
        CMP     W3, W10
        DSB     // Ensure completion of previous cache maintenance instruction
        B.GT    Loop1

Finished:
```

Similar approaches can be used for all cache maintenance instructions.
D7.4.9 Data cache zero instruction

The Data Cache Zero by Address instruction, DC ZVA, writes 0x00 to each byte of a block of N bytes, aligned in memory to N bytes in size, where:

- The block in memory is identified by the address supplied as an argument to the DC ZVA instruction. There are no alignment restrictions on this address.

--- Note ---

This means that each byte of the block of memory that includes the supplied address is set to zero.

- The DCZID_EL0 register indicates the block size, N bytes, that is written with byte values of zero.

Software can restrict access to this instruction. See Configurable instruction controls on page D1-4665 and the description of the DC ZVA instruction.

The DC ZVA instruction behaves as a set of stores to the location being accessed, and:

- Generates a Permission fault if the translation regime being used when the instruction is executed does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instruction.

In addition:

- When the instruction is executed, it can generate memory faults or watchpoints that are prioritized in the same way as other memory related faults or watchpoints. Where a synchronous Data Abort fault or a watchpoint is generated, the CM bit in the syndrome field is not set to 1, which would be the case for all other cache maintenance instructions. See ISS encoding for an exception from a Data Abort on page D17-5685 for more information about the encoding of the associated ESR_ELx.ISS field.

- If the memory region being zeroed is any type of Device memory, then DC ZVA generates an Alignment fault which is prioritized in the same way as other alignment faults that are determined by the memory type.

--- Note ---

The architecture makes no statements about whether or not a DC ZVA instruction causes allocation to any particular level of the cache, for addresses that have a cacheable attribute for those levels of cache.

Despite its mnemonic, the DC ZVA instruction is not a data cache maintenance instruction.

D7.4.10 Cache lockdown

The concept of an entry locked in a cache is allowed, but not architecturally defined. How lockdown is achieved is IMPLEMENTATION DEFINED and might not be supported by:

- An implementation.
- Some memory attributes.

An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.

A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.
The interaction of cache lockdown with cache maintenance instructions

The interaction of cache lockdown and cache maintenance instructions is IMPLEMENTATION DEFINED. However, an architecturally-defined cache maintenance instruction on a locked cache line must comply with the following general rules:

- The effect of the following instructions on locked cache entries is IMPLEMENTATION DEFINED:
  - Cache clean by set/way, **DC CSW**.
  - Cache invalidate by set/way, **DC ISW**.
  - Cache clean and invalidate by set/way, **DC CISW**.
  - Instruction cache invalidate all, **IC IALLU** and **IC IALLUIS**.

However, one of the following approaches must be adopted in all these cases:

1. If the instruction specified an invalidation, a locked entry is not invalidated from the cache.
2. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the DFSC value defined for this purpose, see [ISS encoding for an exception from a Data Abort on page D17-5685](#).

This permits a usage model for cache invalidate routines to operate on a large range of addresses by performing the required operation on the entire cache, without having to consider whether any cache entries are locked.

The effect of the following instructions is IMPLEMENTATION DEFINED:

- Cache clean by virtual address, **DC CVAC**, **DC CVAP**, and **DC CVAU**.
- Cache invalidate by virtual address, **DC IVAC**.
- Cache clean and invalidate by virtual address, **DC CIVAC**.

However, one of the following approaches must be adopted in all these cases:

1. If the instruction specified an invalidation, a locked entry is invalidated from the cache. For the clean and invalidate instructions, the entry must be cleaned before it is invalidated.
2. If the instruction specified an invalidation, a locked entry is not invalidated from the cache. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the DFSC value defined for this purpose. See [ESR_ELx on page K17-11846](#).

In an implementation that includes EL2 enabled in the current Security state, if **HCR_EL2.TIDCP** is set to 1, any exception relating to lockdown of an entry is routed to EL2.

--- Note ---

An implementation that uses an abort mechanism for entries that can be locked down but are not actually locked down must:

- Document the IMPLEMENTATION DEFINED instruction sequences that perform the required operations on entries that are not locked down.
- Implement one of the other permitted alternatives for the locked entries.

Arm recommends that, when possible, such IMPLEMENTATION DEFINED instruction sequences use architecturally-defined instructions. This minimizes the number of customized instructions required.

In addition, an implementation that uses an abort to handle cache maintenance instructions for entries that might be locked must provide a mechanism that ensures that no entries are locked in the cache.

The reset setting of the cache must be that no cache entries are locked.
Additional cache functions for the implementation of lockdown

An implementation can add additional cache maintenance functions for the handling of lockdown in the IMPLEMENTATION DEFINED spaces reserved for Cache Lockdown, see Reserved encodings for IMPLEMENTATION DEFINED registers on page D16-5542.

D7.4.11 System level caches

The Arm Architecture defines a system cache as a cache that is not described in the PE Cache Identification registers, CCSIDR_EL1 and CLIDR_EL1, and for which the set/way cache maintenance instructions do not apply.

Conceptually, three classes of system cache can be envisaged:

1. System caches which lie before the point of coherency and cannot be managed by any cache maintenance instructions. Such systems fundamentally undermine the concept of cache maintenance instructions operating to the point of coherency, as they imply the use of non-architecture mechanisms to manage coherency. The use of such systems in the Arm architecture is explicitly prohibited.

2. System caches which lie before the point of coherency and can be managed by cache maintenance by address instructions that apply to the point of coherency, but cannot be managed by cache maintenance by set/way instructions. Where maintenance of the entirety of such a cache must be performed, as in the case for power management, it must be performed using non-architectural mechanisms.

3. System caches which lie beyond the point of coherency and so are invisible to the software. The management of such caches is outside the scope of the architecture.

D7.4.12 Branch prediction

The architecture does not define any branch predictor maintenance instructions for AArch64 state.

If branch prediction is architecturally visible, cache maintenance must also apply to branch prediction.

D7.4.13 Execution, data prediction and prefetching restriction System instructions

When FEAT_SPECRES is implemented, the System instructions listed in A64 System instructions for prediction restriction on page C5-1146 prevent predictions based on information gathered from earlier execution within a particular execution context (CTX), from affecting the later speculative execution within that CTX, to the extent that the speculative execution is observable through side-channels.

The prediction restriction System instructions being used by a particular CTX apply to:

- All control flow prediction resources that predict execution addresses.
- Data value prediction.
- Cache allocation prediction.

For these System instructions, the CTX is defined by:

- The Security state.
- The Exception level.
- When executing at EL1, if EL2 is implemented and enabled in the current Security state, the VMID.
- When executing at EL0, whether the EL1&0 or the EL2&0 translation regime is in use.
- When executing at EL0 when using the EL1&0 translation regime, the ASID and, if EL2 is implemented and enabled in the current Security state, the VMID.
- When executing at EL0 when using the EL2&0 translation regime, the ASID.

--- Note ---

- The data value prediction applies to all prediction resources that use some form of training to speculate data values as part of an execution.
- The cache allocation applies to all instruction and data caches, and TLB prefetching hardware used by the executing PE that applies to the supplied context.
The context information is passed as a register argument, and is restricted so that:

- Execution of the System instruction at EL0 only applies to the current hardware defined context.
- Execution of the System instruction at EL1 only applies to the current VMID and Security state, and does not apply to EL2 or EL3.
- Execution of the System instruction at EL2 can only apply to the current Security state, and does not apply to EL3.

If the System instruction is specified to apply to Exception levels that are not implemented, or which are higher than the Exception level that the System instruction is executed at, then the System instruction is treated as a NOP.

When the System instruction is complete and synchronized, no predictions of the restricted type for the affected context are influenced by the execution of the program before the System instruction in a manner that can be observed by the use of any side channels.

--- Note ---

- Prediction restriction System instructions do not require the invalidation of prediction structures so long as the behavior described for completion is met by an implementation.
- Prediction restriction System instructions are permitted to invalidate more prediction information than is defined by the supplied execution context.

---

These System instructions are guaranteed to be complete following a DSB that covers both read and write behavior on the same PE that executed the original instruction. A subsequent Context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

In AArch64 state, EL0 access to the System instructions is controlled by:

- When HCR_EL2.{E2H, TGE} is not {1, 1}, SCTLR_EL1.EnRCTX.
- When HCR_EL2.{E2H, TGE} == {1, 1}, SCTLR_EL2.EnRCTX.

--- Note ---

If the SCR_EL3.EEL2 is changed, in order to remove all VMID tagging from Secure EL1 and Secure EL0 entries, each prediction resource should be invalidated for:

- Secure EL0 for all ASID and VMID values.
- Secure EL1 for all VMID values.
D7.5 External aborts

The Arm architecture defines External aborts as errors that occur in the memory system, other than those that are detected by the MMU or debug logic. An External abort might signal a data corruption to the PE. For example, a memory location might have been corrupted, and this corruption is detected by hardware using a parity or error correction code (ECC). The error might have been propagated. The RAS Extension provides mechanisms for software to determine the extent of the corruption and contain propagation of the error. For more information, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile.

An External abort is one of the following:
- Synchronous.
- Precise asynchronous.
- Imprecise asynchronous.

For more information, see Exception entry terminology on page D1-4638.

The RAS Extension provides a more granular taxonomy of aborts. When the RAS Extension is not implemented, the Arm architecture does not provide any method to distinguish between precise asynchronous and imprecise asynchronous External aborts.

It is IMPLEMENTATION DEFINED which External aborts, if any, are supported.

External aborts on data accesses and translation table walks on data accesses can be either synchronous or asynchronous.

When FEAT_DoubleFault is not implemented, External aborts on instruction fetches and translation table walks on instruction fetches can be either synchronous or asynchronous.

When FEAT_DoubleFault is implemented, all External abort exceptions on instruction fetches and translation table walks on instruction fetches must be synchronous.

A synchronous External abort on an instruction fetch, including a translation table walk on an instruction fetch, is taken precisely using the Instruction Abort exception.

A synchronous External abort on a data read or write, including a translation table walk on a data read or write, is taken precisely using the Data Abort exception.

See Synchronous exception types on page D1-4646.

An asynchronous External abort is taken using the SError interrupt exception. See Asynchronous exception types on page D1-4656.

The effect of a failed memory access is described in Definition of a precise exception and imprecise exception on page D1-4638.

Normally, External aborts are rare. An imprecise asynchronous External abort is likely to be fatal to the process that is running. Arm recommends that implementations make External aborts precise wherever possible.

The following subsections give more information about possible External aborts:
- Provision for the classification of External aborts.
- Parity or ECC error reporting, RAS Extension not implemented on page D7-5073.

D7.5.1 Provision for the classification of External aborts

In AArch64 state, an implementation can use ESR_ELx.EA, ISS[9], to provide more information about synchronous External aborts. For all synchronous aborts other than synchronous External aborts, ESR_ELx.EA, ISS[9], returns a value of 0.

If the RAS Extension is implemented:
- The ESR_ELx.SET field provides information about the state of the PE following a synchronous External abort.
- The ESR ELx.AET field might contain more information following an asynchronous abort taken as an SError interrupt.
• The implementation might define error record registers.

For more information, see:
• ISS encoding for an exception from an Instruction Abort on page D17-5679.
• ISS encoding for an exception from a Data Abort on page D17-5685.
• ISS encoding for an SError interrupt on page D17-5695.
• Arm® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile.

D7.5.2 Parity or ECC error reporting, RAS Extension not implemented

The Arm architecture supports the reporting of both synchronous and asynchronous parity or ECC errors from the cache system. It is IMPLEMENTATION DEFINED what parity or ECC errors in the cache systems, if any, result in synchronous or asynchronous parity or ECC errors.

A fault code is defined for reporting parity or ECC errors. However, when parity or ECC error reporting is implemented, it is implementation defined whether a parity or ECC error is reported using the assigned fault code or using another appropriate encoding.

For all purposes other than the Fault status encoding, parity or ECC errors are treated as External aborts.
D7.6 Memory barrier instructions

Memory barriers on page B2-174 describes the memory barrier instructions. This section describes the system level controls of those instructions.

D7.6.1 EL2 control of the shareability of data barrier instructions executed at EL0 or EL1

In an implementation that includes EL2 enabled in the current Security state and supports shareability limitations on the data barrier instructions, the HCR_EL2.BSU field can modify the required shareability of an instruction that is executed at EL0 or EL1. Table D7-8 on page D7-5074 shows the encoding of this field.

<table>
<thead>
<tr>
<th>HCR_EL2.BSU</th>
<th>Minimum shareability of barrier instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No effect, shareability is as specified by the instruction</td>
</tr>
<tr>
<td>01</td>
<td>Inner Shareable</td>
</tr>
<tr>
<td>10</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td>11</td>
<td>Full system</td>
</tr>
</tbody>
</table>

For an instruction executed at EL0 or EL1, Table D7-9 on page D7-5074 shows how the HCR_EL2.BSU is combined with the shareability specified by the argument of the DMB or DSB instruction to give the scope of the instruction.

<table>
<thead>
<tr>
<th>Shareability specified by the DMB or DSB argument</th>
<th>HCR_EL2.BSU</th>
<th>Resultant shareability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full system</td>
<td>Any</td>
<td>Full system</td>
</tr>
<tr>
<td>Outer Shareable</td>
<td>00, 01, or 10</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td></td>
<td>11, Full system</td>
<td>Full system</td>
</tr>
<tr>
<td>Inner Shareable</td>
<td>00 or 01</td>
<td>Inner Shareable</td>
</tr>
<tr>
<td></td>
<td>10, Outer Shareable</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td></td>
<td>11, Full system</td>
<td>Full system</td>
</tr>
<tr>
<td>Non-shareable</td>
<td>00, No effect</td>
<td>Non-shareable</td>
</tr>
<tr>
<td></td>
<td>01, Inner Shareable</td>
<td>Inner Shareable</td>
</tr>
<tr>
<td></td>
<td>10, Outer Shareable</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td></td>
<td>11, Full system</td>
<td>Full system</td>
</tr>
</tbody>
</table>
D7.7 Pseudocode description of general memory System instructions

This section lists the pseudocode describing general memory operations:

- Memory data type definitions.
- Basic memory access.
- Aligned memory access.
- Unaligned memory access.
- Exclusives monitors operations.
- Access permission checking on page D7-5076.
- Abort exceptions on page D7-5076.
- Memory barriers on page D7-5077.

D7.7.1 Memory data type definitions

This section lists the memory data types.

The memory data types are:

- Address descriptor, defined by the AddressDescriptor type.
- Full address, defined by the FullAddress type.
- Memory attributes, defined by the MemoryAttributes type.
- Memory type, defined by the MemType enumeration.
- Device memory type, defined by the DeviceType enumeration.
- Normal memory attributes, defined by the MemAttrHints type.
- Cacheability attributes, defined by the MemAttr_NC, MemAttr_WT, and MemAttr_WB constants.
- Allocation hints, defined by the MemHint_No, MemHint_WA, MemHint_RA, and MemHint_RWA constants.
- Access permissions, defined by the Permissions type.

These types are defined in Chapter J1 Armv8 Pseudocode.

D7.7.2 Basic memory access

The PhysMemRead() and PhysMemWrite() functions perform single-copy atomic, aligned, little-endian memory accesses of size bytes to or from the underlying physical memory array of bytes.

The attributes in memaddrdesc.memattrs are used by the memory system to determine caching and ordering behaviors as described in Memory types and attributes on page B2-196, Ordering relations on page B2-163, and Atomicity in the Arm architecture on page B2-154.

D7.7.3 Aligned memory access

The two MemSingle[] accessors, non-assignment (memory read) AArch64.MemSingle[] and assignment (memory write) AArch64.MemSingle[], make atomic, little-endian accesses of size bytes.

D7.7.4 Unaligned memory access

The two Mem[] accessors, Non-assignment (memory read) Mem[] and Assignment (memory write) Mem[], make accesses of the required type. If an access is not architecturally defined to be atomic, Mem[] synthesizes accesses from multiple calls to AArch64.MemSingle[]. It also reverses the byte order if the access is big-endian.

The AArch64.CheckAlignment() function checks the alignment of memory accesses.

D7.7.5 Exclusives monitors operations

The AArch64.SetExclusiveMonitors() function sets the Exclusives monitors for a block of bytes, the size of which is determined by size, at the virtual address defined by address.
The AArch64.ExclusiveMonitorsPass() function checks whether the Exclusives monitors are set to include the location of a number of bytes specified by size, at the virtual address defined by address. The atomic write that follows after the Exclusives monitors have been set must be to the same physical address. It is permitted, but not required, for this function to return FALSE if the virtual address is not the same as that used in the previous call to AArch64.SetExclusiveMonitors().

The ExclusiveMonitorsStatus() function returns 0 if the previous atomic write was to the same physical memory locations selected by AArch64.ExclusiveMonitorsPass() and therefore succeeded. Otherwise the function returns 1, indicating that the address translation delivered a different physical address.

The MarkExclusiveGlobal() procedure takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The procedure records that the PE processorid has requested exclusive access covering at least size bytes from address paddress. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, up to a limit of 2KB and no smaller than two words, and aligned in the address space to the size of the location. It is CONSTRAINED UNPREDICTABLE whether this causes any previous request for exclusive access to any other address by the same PE to be cleared.

The MarkExclusiveLocal() procedure takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The procedure records in a local record that PE processorid has requested exclusive access to an address covering at least size bytes from address paddress. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, and can at its largest cover the whole of memory but is no smaller than two words, and is aligned in the address space to the size of the location. It is IMPLEMENTATION DEFINED whether this procedure also performs a MarkExclusiveGlobal() using the same parameters.

The IsExclusiveGlobal() function takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The function returns TRUE if the PE processorid has marked in a global record an address range as exclusive access requested that covers at least size bytes from address paddress. It is IMPLEMENTATION DEFINED whether it returns TRUE or FALSE if a global record has marked a different address as exclusive access requested. If no address is marked in a global record as exclusive access, IsExclusiveGlobal() returns FALSE. It is IMPLEMENTATION DEFINED whether this operation also clears the global record of PE processorid for which an address has had a request for an exclusive access.

The IsExclusiveLocal() function takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The function returns TRUE if the PE processorid has marked an address range as exclusive access requested that covers at least the size bytes from address paddress. It is IMPLEMENTATION DEFINED whether this function returns TRUE or FALSE if the address marked as exclusive access requested does not cover all of size bytes from address paddress. If no address is marked as exclusive access requested, then this function returns FALSE. It is IMPLEMENTATION DEFINED whether this result is ANDed with the result of IsExclusiveGlobal() with the same parameters.

The ClearExclusiveByAddress() procedure takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The procedure clears the global records of all PEs, other than processorid, for which an address region including any of size bytes starting from paddress has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether the equivalent global record of the PE processorid is also cleared if any of size bytes starting from paddress has had a request for an exclusive access, or if any other address has had a request for an exclusive access.

The ClearExclusiveLocal() procedure takes as arguments the PE identifier processorid. The procedure clears the local record of PE processorid for which an address has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether this operation also clears the global record of PE processorid that an address has had a request for an exclusive access.

D7.7.6 Access permission checking

The AArch64.S1HasPermissionsFault() and AArch64.S2HasPermissionsFault() functions are used by the architecture to perform access permission checking based on attributes derived from the Translation Tables descriptors.

The interpretation of access permission is shown in Memory access control on page D8-5132.

D7.7.7 Abort exceptions

The AArch64.Abort() function generates either a Data Abort or an Instruction Abort exception by calling AArch64.DataAbort() or AArch64.InstructionAbort(). It also can generate a debug exception for debug related faults, see Chapter D2 AArch64 Self-hosted Debug.
The `AArch64.DataAbort()` function generates a Data Abort exception, routes the exception to EL2 or EL3, and records the information required for the Exception Syndrome registers, ESR_ELx. See *ISS encoding for an exception from a Data Abort* on page D17-5685. A second stage abort might also record the intermediate physical address, IPA, but this depends on the type of the abort.

For a synchronous abort, `AArch64.DataAbort()` also sets the FAR to the VA of the abort.

The `AArch64.InstructionAbort()` function generates an Instruction Abort exception, routes the exception to EL2 or EL3, and records the information required for the Exception Syndrome registers, ESR_ELx, see *ISS encoding for an exception from an Instruction Abort* on page D17-5679. A second stage abort might also record the intermediate physical address, IPA, but this depends on the type of the abort.

For a synchronous abort, `AArch64.InstructionAbort()` also sets the FAR to the VA of the abort.

The `FaultRecord` type describes a fault. Functions that check for faults return a record of this type appropriate to the type of fault.

The function `NoFault()` returns a null record that indicates no fault. The `IsFault()` function tests whether a `FaultRecord` contains a fault.

### D7.7.8 Memory barriers

The definition for the memory barrier functions is given by the enumerations `MBReqDomain` and `MBReqTypes`.

These enumerations define the required shareability domains and required access types used as arguments for `DMB` and `DSB` instructions.

The procedures `DataMemoryBarrier`, `DataSynchronizationBarrier`, and `InstructionSynchronizationBarrier` perform the memory barriers.
The AArch64 System Level Memory Model

D7.7 Pseudocode description of general memory System instructions
Chapter D8
The AArch64 Virtual Memory System Architecture

This chapter provides a system level view of the AArch64 Virtual Memory System Architecture (VMSAv8-64), the memory system architecture of an A-profile implementation that is executing in AArch64 state. It contains the following sections:

- Address translation on page D8-5080
- Translation process on page D8-5093
- Translation table descriptor formats on page D8-5123
- Memory access control on page D8-5132
- Memory region attributes on page D8-5151
- Other descriptor fields on page D8-5158
- Address tagging on page D8-5162
- Pointer authentication on page D8-5164
- Virtualization Host Extensions on page D8-5168
- Nested virtualization on page D8-5173
- Memory aborts on page D8-5180
- Translation Lookaside Buffers on page D8-5193
- TLB maintenance on page D8-5197
- Caches on page D8-5215
- Pseudocode description of VMSAv8-64 address translation on page D8-5217
D8.1 Address translation

If an implementation is executing in AArch64 state, then it is using the Armv8 Virtual Memory System Architecture (VMSA), specifically the 64-bit VMSAv8-64.

Address translation converts the addresses used by instructions to the addresses used by the physical memory system.

When a data address or instruction address is used in an instruction, it is a virtual address (VA). This includes any address stored in one of the following registers:

- Program counter (PC).
- Stack pointers (SP).
- Link register (LR).
- Exception link register (ELR).

When an access is made to the physical memory system, a physical address (PA) is used.

An address translation maps a VA to a PA.

An address translation requires one of the following:

- A single translation stage, stage 1.
- Two sequential translation stages, stage 1 and stage 2.

An address translation stage maps an input address (IA) to an output address (OA).

If one address translation stage is used, then a VA is mapped to a PA using all of the following steps:

1. The VA is input as the IA to the translation stage.
2. The PA is output as the OA from the translation stage.

If two address translation stages are used, then a VA is mapped to a PA using all of the following steps:

1. The VA is input as the IA to the stage 1 translation.
2. The intermediate physical address (IPA) is output as the OA from the stage 1 translation.
3. The IPA is input as the IA to the stage 2 translation.
4. The PA is output as the OA from the stage 2 translation.

If an address translation stage is disabled, then the value of the OA is the same as the IA.

When an IA is translated to an OA, an address translation stage uses a set of memory mapped translation tables that hold all of the following information:

- The OA corresponding to the IA.
- For accesses made from Secure state, whether the OA access is to the Secure or Non-secure address map.
- The OA memory access permissions.
- The OA memory region attributes.

When an IA is translated by an address translation stage, all of the following apply:

- A translation table lookup reads an entry from a translation table.
- A translation table entry resolves a subset of the IA.
- Multiple translation table entries can be required to completely resolve an IA.
- An address translation can require multiple lookups across different lookup levels and multiple translation tables.

When memory is accessed, the Memory Management Unit (MMU) controls address translation, memory access permissions, memory attribute determination, and memory attribute checking.

When the MMU cannot translate the IA, an MMU fault is generated.
When an address translation stage generates an MMU fault, it is one of the following:

- When a stage 1 translation cannot translate an IA, a stage 1 MMU fault is generated.
- When a stage 2 translation cannot translate an IA, a stage 2 MMU fault is generated.

For more information, see:

- Implemented physical address size on page D8-5088.
- Output address size configuration on page D8-5088.
- Supported virtual address ranges on page D8-5089.
- Input address size configuration on page D8-5090.
- Intermediate physical address size configuration on page D8-5091.
- Translation process on page D8-5093.
- The effects of disabling an address translation stage on page D8-5118.
- Memory aborts on page D8-5180

### D8.1.1 Translation granules

The translation granule size determines the number of bits resolved by each lookup level when mapping from an IA to OA, and the maximum size of a single translation table.

The VMSA defines all of the following translation granule sizes:

- 4KB.
- 16KB.
- 64KB.

For each translation granule size, and translation stage, support is IMPLEMENTATION DEFINED.

Each address translation stage can be independently configured to use one of the supported translation granule sizes.

A page is the smallest memory region in which an IA to OA mapping can be specified.

The translation granule determines all of the following:

- The translation process used to resolve an IA to an OA.
- The page size of the address translation stage.
- The number of address bits required to address a page.
- The maximum translation table size of the address translation stage.
- The number of address bits that can be resolved in a single translation table lookup.

For all of the following reasons, a larger translation granule can reduce the number of translation lookup levels:

- The larger granule uses a larger translation table with more entries.
- A single lookup can resolve more IA bits.
- The larger page size means more of the least significant address bits are used to address within a page and do not require translation because those bits are flat-mapped from IA to OA.

Arm recommends separating memory-mapped peripherals by an integer multiple of the largest granule size supported by the PEs in the system, to allow independent management of each peripheral.

If FEAT_GTG is not implemented, then the ID_AA64MMFR0_EL1.{TGran4, TGran16, TGran64} fields specify the translation granules supported in both stage 1 and stage 2 translations.

If FEAT_GTG is implemented, then support for translation granule sizes are determined by all of the following:

- The ID_AA64MMFR0_EL1.{TGran4, TGran16, TGran64} fields specify the translation granules supported in stage 1 translations.
• The ID_AA64MMFR0_EL1.{TGran4_2, TGran16_2, TGran64_2} fields specify the translation granules supported in stage 2 translations.

D8.1.2 Translation regimes

A translation regime determines how a VA is mapped to a PA. The translation regime is affected by the current Security state, the current Exception level, the enabled Exception levels, HCR_EL2 settings, and implemented features.

The architecture defines all of the following translation regimes:
- Secure EL1&0 translation regime.
- Non-secure EL1&0 translation regime.
- Secure EL2&0 translation regime.
- Non-secure EL2&0 translation regime.
- Secure EL2 translation regime.
- Non-secure EL2 translation regime.
- Secure EL3 translation regime.

The implemented Exception levels affect the supported translation regimes and translation stages.

If a VA is accessed in Non-secure state, then the VA is mapped to Non-secure PA space.

If a VA is accessed in Secure state, then one of the following can occur:
- The VA is mapped to Non-secure PA space.
- The VA is mapped to Secure PA space.

Only the EL1&0 translation regimes support a stage 2 translation.

If a software agent, such as an operating system, uses or configures stage 1 translations, then that software agent might be unaware of all of the following:
- The stage 2 translation.
- The distinction between IPA and PA.

If the Effective value of HCR_EL2.E2H is 0, then only the EL1&0 translation regime, stage 1, can support two VA ranges.

If the Effective value of HCR_EL2.E2H is 1, then the EL2&0 translation regime, stage 1, can support two VA ranges.

If a stage 1 translation supports two VA ranges, then that translation stage also supports two privilege levels.

Secure EL1&0 translation regime

If all of the following apply, then the Secure EL1&0 translation regime is used to translate addresses:
- The memory accesses occur in Secure state.
- Memory is accessed from EL1 or EL0.
- The Effective value of HCR_EL2.{E2H, TGE} is not \{1, 1\}.

If Secure EL2 is disabled or not implemented, then the Secure EL1&0 translation regime has a single address translation stage, stage 1, that does all the following:
- Maps VAs to PAs.
- Supports two VA ranges and the use of ASIDs.
If Secure EL2 is enabled, then the Secure EL1&0 translation regime, supports all of the following:

- Translation stage 1 that does all the following:
  - Maps VAs to IPAs.
  - Supports two VA ranges and the use of ASIDs.
- Translation stage 2 that does all the following:
  - Maps stage 1 IPAs to PAs.
  - Supports a single Non-secure IPA range and a single Secure IPA range.
- Translation stage 1 and translation stage 2 can be enabled independent of each other.

For more information, see Use of ASIDs and VMIDs to reduce TLB maintenance requirements on page D8-5194.

**Non-secure EL1&0 translation regime**

If all of the following apply and EL2 is disabled, then the Non-secure EL1&0 translation regime is used to translate addresses:

- The memory accesses occur in Non-secure state.
- Memory is accessed from EL1 or EL0.
- The Effective value of HCR_EL2.{E2H, TGE} is not \( \{1, 1\} \).

If Non-secure EL2 is not implemented, then the Non-secure EL1&0 translation regime has a single address translation stage, stage 1, that does all the following:

- Maps VAs to PAs.
- Supports two VA ranges and the use of ASIDs.

If Non-secure EL2 is enabled, then the Non-secure EL1&0 translation regime supports all of the following:

- Translation stage 1 that does all the following:
  - Maps VAs to IPAs.
  - Supports two VA ranges and the use of ASIDs.
- Translation stage 2 that does all the following:
  - Maps stage 1 IPAs to PAs.
  - Supports a single IPA range.
- Translation stage 1 and translation stage 2 can be enabled independent of each other.

For more information, see Use of ASIDs and VMIDs to reduce TLB maintenance requirements on page D8-5194.

Non-secure EL2 is effectively disabled if it is not implemented.

**Secure EL2&0 translation regime**

If all of the following apply, then the Secure EL2&0 translation regime is used to translate addresses:

- The Effective value of SCR_EL3.EEL2 is 1.
- The memory accesses occur in Secure state.
- One of the following applies:
  - The Effective value of HCR_EL2.{E2H, TGE} is \( \{1, 0\} \) and memory accesses are from EL2.
  - The Effective value of HCR_EL2.{E2H, TGE} is \( \{1, 1\} \) and memory accesses are from EL2 or EL0.

For more information, see Virtualization Host Extensions on page D8-5168.

The Secure EL2&0 translation regime has a single address translation stage, stage 1, that does all the following:

- Maps VAs to PAs.
- Supports two VA ranges and the use of ASIDs.

For more information, see Use of ASIDs and VMIDs to reduce TLB maintenance requirements on page D8-5194.
Non-secure EL2&0 translation regime

If all of the following apply, then the Non-secure EL2&0 translation regime is used to translate addresses:

- The memory accesses occur in Non-secure state.
- One of the following applies:
  - The Effective value of HCR_EL2.{E2H, TGE} is \{1, 0\} and memory accesses are from EL2.
  - The Effective value of HCR_EL2.{E2H, TGE} is \{1, 1\} and memory accesses are from EL2 or EL0.

For more information, see Virtualization Host Extensions on page D8-5168.

Non-secure EL2 translation regime

If all of the following apply, then the Non-secure EL2 translation regime is used to translate addresses:

- The memory accesses occur in Non-secure state.
- The Effective value of HCR_EL2.E2H is 0.
- Memory is accessed from EL2.

The Non-secure EL2 translation regime has a single address translation stage, stage 1, that does all the following:

- Maps VAs to PAs.
- Supports a single VA range.

Secure EL2 translation regime

If all of the following apply, then the Secure EL2 translation regime is used to translate addresses:

- The Effective value of SCR_EL3.EEL2 is 1.
- The memory accesses occur in Secure state.
- The Effective value of HCR_EL2.E2H is 0.
- Memory is accessed from EL2.

The Secure EL2 translation regime has a single address translation stage, stage 1, that does all the following:

- Maps VAs to PAs.
- Supports a single VA range.

Non-secure EL3 translation regime

If memory is accessed from EL3, then the Secure EL3 translation regime is used to translate addresses.

The Secure EL3 translation regime has a single address translation stage, stage 1, that does all the following:

- Maps VAs to PAs.
- Supports a single VA range.

D8.1.3 Relationship between translation regimes and implemented Exception levels

The set of translation regimes that an implementation supports depend on the set of Exception levels that the implementation supports.
If an implementation does not include EL2, then all of the following apply:

- If the implementation does not include EL3, then the MMU provides a single EL1&0 stage 1 translation regime with an IMPLEMENTATION DEFINED Security state.
- If the implementation includes EL3, then the MMU provides an EL1&0 stage 1 translation regime in each Security state.

For each translation stage supported by a translation regime, the following table shows what is required to support that translation stage.

<table>
<thead>
<tr>
<th>Translation stage</th>
<th>Requires</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure EL3 stage 1</td>
<td>EL3 is implemented and EL3 uses AArch64.</td>
</tr>
<tr>
<td>Secure EL2 stage 1</td>
<td>EL2 is implemented and EL2 uses AArch64. EL2 is implemented and EL2 uses AArch64. The Effective value of SCR_EL3.EEL2 is 1. The Effective value of SCR_EL3.NS is 0.</td>
</tr>
<tr>
<td>Secure EL2&amp;0 stage 1</td>
<td>EL2 is implemented and EL2 uses AArch64. EL2 is implemented and EL2 uses AArch64. The Effective value of SCR_EL3.EEL2 is 1. The Effective value of SCR_EL3.NS is 0. The Effective value of HCR_EL2.E2H is 1.</td>
</tr>
<tr>
<td>Secure EL1&amp;0 stage 2</td>
<td>EL2 is implemented and EL2 uses AArch64. EL2 is implemented and EL2 uses AArch64. The Effective value of SCR_EL3.EEL2 is 1. The Effective value of SCR_EL3.NS is 0.</td>
</tr>
<tr>
<td>Secure EL1&amp;0 stage 1</td>
<td>EL1 uses AArch64. EL1 uses AArch64. The Effective value of SCR_EL3.NS is 0.</td>
</tr>
<tr>
<td>Non-secure EL2 stage 1</td>
<td>EL2 is implemented and EL2 uses AArch64. EL2 is implemented and EL2 uses AArch64. The Effective value of SCR_EL3.NS is 1.</td>
</tr>
<tr>
<td>Non-secure EL2&amp;0 stage 1</td>
<td>EL2 is implemented and EL2 uses AArch64. EL2 is implemented and EL2 uses AArch64. The Effective value of SCR_EL3.NS is 1. The Effective value of SCR_EL3.NS is 1.</td>
</tr>
<tr>
<td>Non-secure EL1&amp;0 stage 2</td>
<td>EL2 is implemented and EL2 uses AArch64. EL2 is implemented and EL2 uses AArch64. The Effective value of SCR_EL3.NS is 1.</td>
</tr>
<tr>
<td>Non-secure EL1&amp;0 stage 1</td>
<td>EL1 uses AArch64. EL1 uses AArch64. The Effective value of SCR_EL3.NS is 1.</td>
</tr>
</tbody>
</table>

For a stage 1 translation, if EL0 uses AArch32 state and one or more of the following applies, then VMSAv8-64 translation is required:

- EL1 uses AArch64 and the Effective value of HCR_EL2.{E2H, TGE} is not {1, 1}.
- EL2 uses AArch64 and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

For a stage 2 translation, if either EL1 or EL0 uses AArch32 state and EL2 uses AArch64, then VMSAv8-64 translation is required.

If EL0 uses AArch32, then accesses from EL0 have all of the following properties:

- Accesses use the EL1&0 or EL2&0 translation regime.
- Accesses use the AArch32 application-level memory model.
- Accesses are limited to a 32-bit VA range.

For more information, see Chapter B1 The AArch64 Application Level Programmers’ Model.
### D8.1.4 System registers relevant to MMU operation

If a System register name has a numeric suffix, then the suffix indicates the lowest Exception level that can access the register.

Translation stages are enabled and controlled by the registers specified in the following table.

**Table D8-2 Registers that enable and control translation stages**

<table>
<thead>
<tr>
<th>Translation Stage</th>
<th>Controlled from</th>
<th>Controlling registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure EL3 stage 1</td>
<td>EL3</td>
<td>SCTLR_EL3, SCR_EL3, TCR_EL3, MAIR_EL3, AMAIR_EL3, TTBR0_EL3</td>
</tr>
<tr>
<td>Secure EL2 stage 1</td>
<td>Secure EL2</td>
<td>SCTLR_EL2, TCR_EL2, MAIR_EL2, AMAIR_EL3, TTBR0_EL2, SCR_EL3</td>
</tr>
<tr>
<td>Secure EL2&amp;0 stage 1</td>
<td>Secure EL2</td>
<td>SCTLR_EL2, TCR_EL2, MAIR_EL2, AMAIR_EL3, TTBR0_EL2, TTBR1_EL2, HCR_EL2, SCR_EL3</td>
</tr>
<tr>
<td>Secure EL1&amp;0 stage 2</td>
<td>Secure EL2</td>
<td>SCTLR_EL2, VSTCR_EL2, VSTTBR_EL2, VTCR_EL2, VTTBR_EL2, SCR_EL3</td>
</tr>
<tr>
<td>Secure EL1&amp;0 stage 1</td>
<td>Secure EL1</td>
<td>SCTLR_EL1, TCR_EL1, MAIR_EL1, AMAIR_EL3, TTBR0_EL1, TTBR1_EL1, HCR_EL2, SCR_EL3</td>
</tr>
<tr>
<td>Non-secure EL2 stage 1</td>
<td>Non-secure EL2</td>
<td>SCTLR_EL2, TCR_EL2, MAIR_EL2, AMAIR_EL3, TTBR0_EL2</td>
</tr>
<tr>
<td>Non-secure EL2&amp;0 stage 1</td>
<td>Non-secure EL2</td>
<td>SCTLR_EL2, TCR_EL2, MAIR_EL2, AMAIR_EL3, TTBR0_EL2, TTBR1_EL2, HCR_EL2</td>
</tr>
<tr>
<td>Non-secure EL1&amp;0 stage 2</td>
<td>Non-secure EL2</td>
<td>SCTLR_EL1, TCR_EL1, MAIR_EL1, AMAIR_EL3, TTBR0_EL1, TTBR1_EL1, HCR_EL2</td>
</tr>
<tr>
<td>Non-secure EL1&amp;0 stage 1</td>
<td>Non-secure EL1</td>
<td>SCTLR_EL1, TCR_EL1, MAIR_EL1, AMAIR_EL3, TTBR0_EL1, TTBR1_EL1, HCR_EL2</td>
</tr>
</tbody>
</table>

**Table D8-3 Common system register abbreviations used in VMSAv8-64**

<table>
<thead>
<tr>
<th>Common Abbreviation</th>
<th>Translation stage</th>
<th>Exception level</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCTLR_ELx</td>
<td>-</td>
<td>SCTLR_EL1</td>
</tr>
<tr>
<td>TCR_ELx</td>
<td>Stage 1</td>
<td>TCR_EL1</td>
</tr>
<tr>
<td></td>
<td>Stage 2</td>
<td>TCR_EL2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSTCR_EL2, VSTTBR_EL2</td>
</tr>
<tr>
<td>TTBR_ELx</td>
<td>Stage 1</td>
<td>TTBR0_EL1, TTBR1_EL1</td>
</tr>
<tr>
<td></td>
<td>Stage 2</td>
<td>TTBR0_EL2, TTBR1_EL2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VTTBR_EL2, VSTTBR_EL2</td>
</tr>
<tr>
<td>TTBRn_ELx</td>
<td>Stage 1</td>
<td>TTBR0_EL1, TTBR1_EL1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TTBR0_EL2, TTBR1_EL2</td>
</tr>
<tr>
<td>TTBR0_ELx</td>
<td>Stage 1</td>
<td>TTBR0_EL1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TTBR0_EL2</td>
</tr>
<tr>
<td>TTBR1_ELx</td>
<td>Stage 1</td>
<td>TTBR1_EL1</td>
</tr>
</tbody>
</table>

When software that enables or disables an address translation stage uses a PA that differs from the VA, speculative instruction fetching can cause complications. If software controls address translations that apply to the software itself, then when that software enables or disables an address translation stage, Arm recommends that the software uses a PA that is identical to the VA.
D8.1.5 Out-of-context translation regimes

Software is required to consider the effects on registers and translation tables when a context switch is made from one translation regime to another.

Speculative memory accesses from out-of-context translation regimes

If a speculative memory access causes a translation table walk, and the translation table walk is speculative when the lower Exception level was entered, all of the following apply regarding memory accesses caused by speculative translation table walks from the lower Exception level:

- All of the speculative memory accesses are observed as required by the Shareability and Cacheability attributes of the lower Exception level translation table entries.
- If it is impossible to determine that the speculative memory accesses have been observed by the observers, then these speculative memory accesses can still occur.
- No new memory accesses using the lower Exception level translation table entries occur.

If EL2 is enabled, when a switch from one virtual machine to another occurs, software executing at EL0 or EL1 expects register fields controlling or affecting the EL1&0 regime are modified consistently before the next virtual machine starts execution.

If the registers fields controlling or affecting the EL1&0 translation regime are modified when changing a virtual machine, then all of the following apply:

- Software executing at EL2 does the modification.
- The registers are modified out-of-context from software executing at EL1 or EL0.
- When modifying the registers, no synchronization precautions are required until the exception return to EL1 or EL0.
D8.1.6 Implemented physical address size

The implemented PA size is indicated by the value of the ID_AA64MMFR0_EL1.PARange field, as shown in the following table. All other values of the ID_AA64MMFR0_EL1.PARange field are reserved.

<table>
<thead>
<tr>
<th>ID_AA64MMFR0_EL1.PARange</th>
<th>PA memory size</th>
<th>PA address size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>4GB</td>
<td>32 bits, PA[31:0]</td>
</tr>
<tr>
<td>0b0001</td>
<td>64GB</td>
<td>36 bits, PA[35:0]</td>
</tr>
<tr>
<td>0b0010</td>
<td>1TB</td>
<td>40 bits, PA[39:0]</td>
</tr>
<tr>
<td>0b0011</td>
<td>4TB</td>
<td>42 bits, PA[41:0]</td>
</tr>
<tr>
<td>0b0100</td>
<td>16TB</td>
<td>44 bits, PA[43:0]</td>
</tr>
<tr>
<td>0b0101</td>
<td>256TB</td>
<td>48 bits, PA[47:0]</td>
</tr>
<tr>
<td>0b0110</td>
<td>4PB</td>
<td>52 bits, PA[51:0]</td>
</tr>
</tbody>
</table>

If the implemented PA size is greater than 48 bits, then FEAT_LPA is required.

D8.1.7 Output address size configuration

For a translation stage, the maximum OA size that can be described by translation table entries is one of the following:

- For a translation regime controlled by TCR_ELx, if the 4KB or 16KB translation granule is used, then one of the following:
  - If the Effective value of TCR_ELx.DS is 0, then 48 bits.
  - If the Effective value of TCR_ELx.DS is 1, then 52 bits.
- For any translation regime, if the 64KB translation granule is used, then one of the following:
  - If FEAT_LPA is not implemented, then 48 bits.
  - If FEAT_LPA is implemented, then 52 bits.

The name of the TCR_ELx.{I}PS field is one of the following:

- If the register is TCR_EL1, then the field is TCR_EL1.IPS.
- If the register is TCR_EL2, then one of the following:
  - If the Effective value of HCR_EL2.E2H is 0, then the field is TCR_EL2.PS.
  - If the Effective value of HCR_EL2.E2H is 1, then the field is TCR_EL2.IPS.
- If the register is TCR_EL3, then the field is TCR_EL3.PS.

The OA size from a translation stage is configured by the associated TCR_ELx.{I}PS field, as shown in the following table. All other values of the TCR_ELx.{I}PS field are reserved.

<table>
<thead>
<tr>
<th>TCR_ELx.{I}PS</th>
<th>OA memory size</th>
<th>OA address size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000</td>
<td>4GB</td>
<td>32 bits, OA[31:0]</td>
</tr>
<tr>
<td>0b001</td>
<td>64GB</td>
<td>36 bits, OA[35:0]</td>
</tr>
<tr>
<td>0b010</td>
<td>1TB</td>
<td>40 bits, OA[39:0]</td>
</tr>
<tr>
<td>0b011</td>
<td>4TB</td>
<td>42 bits, OA[41:0]</td>
</tr>
</tbody>
</table>
The TCR_ELx.{I}PS field is three bits and has the same encoding as the three least significant bits of ID_AA64MMFR0_EL1.PARange.

The OA size from a translation stage cannot be larger than the implemented PA size. If the value of TCR_ELx.{I}PS is larger than the implemented PA size, then {I}PS is treated as being the same as the implemented PA size.

Arm strongly recommends that software avoids configuring TCR_ELx.{I}PS to a value greater than the implemented PA size.

If a translation stage is enabled and the translation stage OA size is larger than the implemented PA size, then an Address size fault is generated at the lookup level in the translation stage that generated the OA.

If all of the following apply, then a stage 2 Translation fault is generated:
- Two address translation stages are used.
- Stage 2 address translation is enabled.
- The stage 1 OA size does not generate a stage 1 Address size fault.
- The stage 1 OA size is larger than the specified stage 2 translation IA size.

### Table D8-5 Output address size configuration (continued)

<table>
<thead>
<tr>
<th>TCR_ELx.{I}PS</th>
<th>OA memory size</th>
<th>OA address size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b100</td>
<td>16TB</td>
<td>44 bits, OA[43:0]</td>
</tr>
<tr>
<td>0b101</td>
<td>256TB</td>
<td>48 bits, OA[47:0]</td>
</tr>
<tr>
<td>0b110</td>
<td>4PB</td>
<td>52 bits, OA[51:0]</td>
</tr>
</tbody>
</table>

Only stage 1 address translation in the EL1&0 and EL2&0 translation regimes support two VA ranges.

If a translation regime supports a single VA range, then the maximum VA range is one of the following:
- If the maximum VA size is 48 bits, then the maximum VA range is 0x0000000000000000 to 0x0000FFFFFFFFFFFF.
- If the maximum VA size is 52 bits, then the maximum VA range is 0x0000000000000000 to 0x000FFFFFFFFFFFFF.

If a translation regime supports two VA ranges, then all of the following apply:
- IA bit[55] determines one of the following VA ranges:
  - If IA bit[55] is 0, the lower VA range is used.
  - If IA bit[55] is 1, the upper VA range is used.
• The maximum lower VA range is one of the following:
  — If the maximum VA size is 48 bits, then the maximum VA range is 0x0000000000000000 to 0xFFFFFFFFFFFF.
  — If the maximum VA size is 52 bits, then the maximum VA range is 0x0000000000000000 to 0xFFFFFFFFFFFF.

• The maximum upper VA range is one of the following:
  — If the maximum VA size is 48 bits, then the maximum VA range is 0xFFFF000000000000 to 0xFFFFFFFFFFFFFFFF.
  — If the maximum VA size is 52 bits, then the maximum VA range is 0xFFF0000000000000 to 0xFFFFFFFFFFFFFFFF.

If the configured size of the upper VA range is less than the maximum, then all of the following apply:
• The range starting address is greater than the address used by the maximum VA size.
• The range ending address is 0xFFFFFFFFFFFFFFFF.

A 48-bit VA range defines an address space of 256TB. A 52-bit VA range defines an address space of 4PB.

D8.1.9 Input address size configuration

If a translation stage supports a single VA range, then TCR_ELx.T0SZ configures the IA size, translated using TTBR0_ELx.

If a translation stage supports two VA ranges, then all of the following TCR_ELx.TnSZ fields specify the IA size:
• TCR_ELx.T0SZ configures the IA size of the lower VA range, translated using TTBR0_ELx.
• TCR_ELx.T1SZ configures the IA size of the upper VA range, translated using TTBR1_ELx.

The stage 1 translation IA size is defined as 64-TCR_ELx.TnSZ.

For all translation stages, the following table shows how the maximum permitted TnSZ value is determined:

<table>
<thead>
<tr>
<th>Translation granule size</th>
<th>FEAT_TTST not implemented</th>
<th>FEAT_TTST implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB</td>
<td>39</td>
<td>48</td>
</tr>
<tr>
<td>16KB</td>
<td>39</td>
<td>48</td>
</tr>
<tr>
<td>64KB</td>
<td>39</td>
<td>47</td>
</tr>
</tbody>
</table>

If TCR_ELx.TnSZ configures an IA size that is smaller than the maximum size, then each one-bit reduction in the IA size has one of the following effects on the lookup level that the table walk starts with:
• The translation table size is reduced by one half.
• The table walk starts one level later.

If the value of TnSZ is greater than the maximum permitted value, then any use of the TnSZ value causes one of the following IMPLEMENTATION DEFINED behaviors to occur:
• For all purposes other than reading back the value, the implementation behaves as if the TnSZ field has the maximum permitted value.
• Any use generates a level 0 Translation fault.

For a stage 1 translation, the minimum Effective value of TnSZ is one of the following:
• For the 4KB or 16KB translation granule, one of the following:
  — If the Effective value of TCR_ELx.DS is 0, then the minimum Effective value of TnSZ is 16.
  — If the Effective value of TCR_ELx.DS is 1, then the minimum Effective value of TnSZ is 12.
• For the 64KB translation granule, one of the following:
  — If FEAT_LVA is not implemented, then the minimum Effective value of TnSZ is 16.
  — If FEAT_LVA is implemented, then the minimum Effective value of TnSZ is 12.

If all of the following apply, then a stage 1 level 0 Address size fault is generated:
• Stage 1 translation is disabled.
• The IA size is larger than the implemented PA size.

For an address translation stage, when an attempt is made to translate an address larger than the configured IA size, a level 0 Translation fault at that translation stage is generated.

For a stage 1 address translation, if the value of TnSZ is smaller than the minimum Effective value, then when using the TnSZ value to translate an address, one of the following applies:
• If FEAT_LVA is not implemented, then it is IMPLEMENTATION DEFINED which of the following behaviors occurs:
  — For all uses other than reading back the value, the implementation behaves as if the TnSZ field has a value of 16.
  — When using the TnSZ value to translate an address, a stage 1 level 0 Translation fault is generated.
• If FEAT_LVA is implemented, then when using the TnSZ value to translate an address, a stage 1 level 0 Translation fault is generated.

For more information, see Relationship between translation regimes and implemented Exception levels on page D8-5084.

**D8.1.10 Intermediate physical address size configuration**

When a stage 2 translation occurs, the configured IPA size is specified by one of the following T0SZ values:
• If the IPA is in Non-secure address space, then VTCR_EL2.T0SZ
• If the IPA is in Secure address space, then VSTCR_EL2.T0SZ

The implemented PA size constrains all of the following:
• The maximum IPA size.
• The effective minimum T0SZ value, VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ, used to specify the configured IPA size.
• The stage 2 initial lookup level.

When all of the following apply, the stage 2 translation IPA size check generates a stage 2 level 0 Translation fault:
• Stage 2 translation is enabled.
• The OA size from the stage 1 translation, whether or not stage 1 translation is enabled, is larger than the IA size specified by VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ.

When a stage 2 translation occurs, the effective minimum value of T0SZ is determined by the implemented PA size as shown in the following table:

<table>
<thead>
<tr>
<th>Implemented PA size</th>
<th>Effective minimum T0SZ value</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bits</td>
<td>32 if EL1 is using AArch64</td>
</tr>
<tr>
<td>32 bits</td>
<td>24 if EL1 is using AArch32</td>
</tr>
<tr>
<td>36 bits</td>
<td>28 if EL1 is using AArch64</td>
</tr>
<tr>
<td>36 bits</td>
<td>24 if EL1 is using AArch32</td>
</tr>
</tbody>
</table>

For more information, see Relationship between translation regimes and implemented Exception levels on page D8-5084.
For a stage 2 address translation, if FEAT_LPA is not implemented and the value of T0SZ is smaller than the effective minimum value, then it is IMPLEMENTATION_DEFINED which of the following behaviors occurs:

- For all purposes other than reading back the value of the field, the implementation treats the T0SZ field as having the effective minimum value.
- The implementation generates a stage 2 level 0 Translation fault.

For a stage 2 address translation, if FEAT_LPA is implemented and the value of the T0SZ field is smaller than the effective minimum value, then a stage 2 level 0 Translation fault is generated.

If the T0SZ field is programmed to a value smaller than the effective minimum value, then a larger address range cannot be supported because an Address size fault is generated due to one of the following reasons:

- For a VMSAv8-64 stage 1 translation, the stage 1 OA is larger than the PA size.
- For a VMSAv8-32 stage 1 translation, the stage 1 OA is larger than 40 bits.

For more information, see:

- Implemented physical address size on page D8-5088.
- Translation table walk on page D8-5093.
- Memory aborts on page D8-5180.

### Table D8-7 Determining the effective minimum T0SZ value for stage 2 translations (continued)

<table>
<thead>
<tr>
<th>Implemented PA size</th>
<th>Effective minimum T0SZ value</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 bits</td>
<td>24</td>
</tr>
<tr>
<td>42 bits</td>
<td>22</td>
</tr>
<tr>
<td>44 bits</td>
<td>20</td>
</tr>
<tr>
<td>48 bits</td>
<td>16</td>
</tr>
<tr>
<td>52 bits</td>
<td>12</td>
</tr>
</tbody>
</table>
D8.2 Translation process

The translation of a VA to a PA begins with reading the translation table base address register, followed by a walk through the translation tables to read the descriptors associated with the mapping.

D8.2.1 Translation table walk

A translation table walk is the set of translation table lookups that are required to do all of the following:

- For a single stage translation at stage 1, translate a VA to a PA.
- For two translation stages and stage 1 is disabled, translate an IPA to a PA.
- For a two stage translation, all of the following:
  - For a stage 1 translation, translate a VA to an IPA.
  - For a stage 2 translation, translate an IPA to a PA for each of the stage 1 translation table lookups.
  - For a stage 2 translation, translate an IPA to a PA for the stage 1 OA.

When a translation table walk begins, the initial translation table lookup uses the translation table base address stored in the TTBR_ELx register to locate the translation table.

When a translation table lookup occurs, the descriptor held in the translation table entry is read as an 8-byte single-copy atomic access.

For information on the translation table alignment requirements, see Translation table alignment on page D8-5098.

The descriptor type indicates one of the following:

- The translation table walk is complete and the translation table entry is the final entry.
- The translation table walk requires an extra lookup at the next higher lookup level.
- The descriptor is invalid.

When an extra lookup level is required, the descriptor contains all of the following information:

- The translation table walk base address of the next lookup level.
- For stage 1 descriptors, if the Effective value of TCR_ELx.HPD is 0, then the descriptor has hierarchical access permissions that are applied to the final translation.
- If the translation is made in a Secure translation regime, then a stage 1 descriptor indicates whether the base address is in the Secure address space or the Non-secure address space, unless a hierarchical control at a previous lookup level has indicated that it is required to be in the Non-secure address space.

For more information, see Security state of translation table lookups on page D8-5099.

For a translation lookup level, the translation table base address is one of the following:

- For the initial lookup level, the aligned value of the appropriate TTBR_ELx.BADDR field.
- For subsequent lookup levels, the next-level translation table base address returned by the previous lookup level.

When the last lookup level of a translation stage returns a valid descriptor, it contains all of the following:

- The OA from the translation stage.
- The memory region access permissions.
- The memory region attributes.

For more information, see Memory region attributes on page D8-5151 and Memory access control on page D8-5132.

For a translation regime that uses two translation stages, the stage 1 descriptor addresses require stage 2 translation from IPA to PA.

When a translation table walk succeeds, all of the following are returned:

- The OA.
• If the translation table walk is made in a Secure translation regime, then the information returned indicates whether the OA is in the Secure IPA or PA space, or to the Non-secure IPA or PA space.
• The output memory region attributes.
• The output memory region access permissions.

For more information, see Security state of translation table lookups on page D8-5099.

The following figure is a generalized view of a single address translation stage with three lookup levels.

Figure D8-1 Generalized view of a single address translation stage

If all of the following are true, then a translation table entry is permitted to be cached in a Translation Lookaside Buffer (TLB):

- The entry is valid.
- Using the entry does not generate a Translation fault, an Address size fault, or an Access flag fault.

TLB caching can reduce the number of required translation table lookups.

If two address translation stages are used and when a full translation table walk at both stages is required, then all of the following apply:

- \( S1 \) is the number of lookup levels required for a stage 1 translation.
- \( S2 \) is the number of lookup levels required for a stage 2 translation.
- The maximum number of translation table lookups required is \((S1+1)*(S2+1)-1\).

**D8.2.2 Concatenated translation tables**

For the initial lookup of a stage 2 address translation, up to 16 translation tables can be concatenated.

For a stage 2 translation, if the translation table in the initial lookup level would require 16 or fewer entries, then the stage 2 translation can be configured to have all of the following properties:

- The initial lookup of the stage 2 translation starts at the next lookup level.
- A number of translation tables corresponding to the original number of translation table entries at the previously initial lookup level are concatenated at that next lookup level.

For the 16KB translation granule, if the Effective value of VTCR_EL2.DS is 0 and a 48-bit address size is translated by the stage 2 translation, then the initial lookup is required to start at level 1 with two concatenated translation tables.
For the initial lookup in a stage 2 translation, if concatenated translation tables are used, then all of the following apply:

- Up to four additional IA bits are resolved by the initial lookup.
- For each additional $n$ IA bits resolved by the initial lookup, the number of least significant translation table base address bits held in the TTBR_ELx is reduced by $n$ bits.
- For each additional $n$ IA bits resolved by the initial lookup, $2^n$ concatenated translation tables are required at the initial lookup level.

Using a concatenated translation table eliminates the overhead of an extra lookup level.

If concatenated translation tables are used, then software is required to do all of the following:

- Align the base address of the first translation table to the sum of the size of the memory occupied by the concatenated translation tables.
- Program VTTBR_EL2 or VSTTBR_EL2 with the address of the first translation table in the set of concatenated translation tables.
- Program VTCR_EL2 or VSTCR_EL2 with the initial lookup level.

### D8.2.3 Translation table base address register

For a translation stage, the translation table base address register, TTBR_ELx, holds the translation table base address of the initial lookup.

For a translation stage that supports two VA ranges, two translation table base address registers are required.

Software can use TCR_ELx.TnSZ to configure the translation stage IA size to be smaller than the supported size.

If EL2 is disabled or not implemented, then for the stage 1 EL1&0 regime, all of the following apply:

- The translation table base address held in the TTBR_ELx register is a PA.
- The translation table base address returned by a Table descriptor is a PA.

If EL2 is enabled, then for all address translation stages other than stage 1 in EL1&0, all of the following apply:

- The translation table base address held in the TTBR_ELx register is a PA.
- The translation table base address returned by a Table descriptor is a PA.

If EL2 is enabled, then for stage 1 address translations in EL1&0, all of the following apply:

- The stage 1 translation table base address held in the TTBR_ELx register is an IPA.
- The stage 1 translation table base address returned by a Table descriptor is an IPA.
- Accesses to stage 1 translation tables are subject to a stage 2 translation.

For an address translation stage, the translation table base address in TTBR_ELx is defined for the supported OA size of that stage.

The number of OA bits held in TTBR_ELx is determined by the granule size and OA address size. The bits used for each granule size when using the maximum OA address size of 48 bits or 52 bits are shown in the following table. Software might be required to set one or more of the low-order base address bits to zero to align the table to the table size.

<table>
<thead>
<tr>
<th>Granule size</th>
<th>Maximum OA size</th>
<th>Translation table base address</th>
<th>OA bits held in TTBR_ELx</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB, 16KB, or 64KB</td>
<td>52 bits</td>
<td>TTBR_ELx[5:2]</td>
<td>OA[51:48]</td>
</tr>
</tbody>
</table>
If an address translation stage uses an OA size smaller than the maximum, then the upper bits of the translation table base address in TTBR_ELx corresponding to the upper bits of OA size are required to be set by software to zero.

If TCR_ELx.TnSZ specifies an IA size that is smaller than the maximum size resolved at the initial lookup level, then more low-order TTBR_ELx bits are used to hold the translation table base address.

If an address translation stage uses an OA size smaller than the maximum and if the bits above the configured OA size of the translation table base address in TTBR_ELx are not set to zero, then an Address size fault is generated and reports all of the following:

- A translation level 0 lookup as generating the fault, regardless of whether or not the translation stage starts with a level 0 lookup.
- The translation stage that generated the fault.

For more information, see System registers relevant to MMU operation on page D8-5086.

### Selection between TTBR0_ELx and TTBR1_ELx when two VA ranges are supported

If a stage 1 translation regime supports two VA ranges, then the translation regime TTBR_ELx registers point to all of the following address ranges:

- For the lower VA range that begins at address 0x0000000000000000, TTBR0_ELx points to the translation table for the initial lookup level.
- For the upper VA range that ends at address 0xFFFFFFFFFFFFFFFF, TTBR1_ELx points to the translation table for the initial lookup level.

If a stage 1 translation regime supports two VA ranges, then the TCR_ELx.{T0SZ, T1SZ} fields configure all of the following address range sizes:

- The lower VA range is 0x0000000000000000 to (2^(64-T0SZ) - 1).
- The upper VA range is (2^64 - 2^(64-T1SZ)) to 0xFFFFFFFFFFFFFFFF.

If a stage 1 translation regime supports two VA ranges, when an accessed address is not in the lower VA range or the upper VA range, a stage 1 level 0 Translation fault is generated.

The following figure illustrates the two address ranges translated by the tables the TTBR_ELx registers point to.
If a stage 1 translation regime supports two VA ranges, then all of the following are used to select the TTBR_ELx:

- If VA bit[55] is zero, then TTBR0_ELx is selected.
- If VA bit[55] is one, then TTBR1_ELx is selected.

If a stage 1 translation regime supports two VA ranges and TCR_ELx.EPDn is one, then when a TLB miss occurs based on TTBRn_ELx, a level 0 Translation fault is generated, and no translation table walk is done.

For more information, see Address tagging on page D8-5162.

**Preventing EL0 access to halves of the address map**

All statements in this section require implementation of FEAT_E0PD.

The TCR_ELx.{E0PD0, E0PD1} fields can be used to prevent EL0 access to the addresses translated by the corresponding TTBR0_ELx or TTBR1_ELx.

When the TCR_ELx.{E0PD0, E0PD1} fields prevent EL0 access to an address translated by TTBR0_ELx or TTBR1_ELx, then a level 0 Translation fault is generated.

When the TCR_ELx.{E0PD0, E0PD1} fields generate a level 0 Translation fault, then the time needed to take the fault should be the same whether or not the address accessed is present in a TLB, to mitigate attacks that use fault timing.

When the TCR_ELx.{E0PD0, E0PD1} fields generate a level 0 Translation fault, the fault is not counted as a TLB miss for performance monitoring features.

**D8.2.5 Translation table and translation table lookup properties**

Translation table and translation table lookup properties include the table size, table alignment, table endianness, and memory attributes.

**Translation table size**

The descriptor in a translation table entry is an eight-byte, or 64-bit, object.
The size of a translation table in bytes is determined by multiplying the number of entries by the eight-byte descriptor size.

The maximum number of translation table entries is determined by the translation granule size, which is defined by one of the following:

- For a stage 1 translation that supports one VA range, TCR_ELx.TG0.
- For a stage 1 translation that can support two VA ranges, all of the following:
  - For the lower VA range, TCR_ELx.TG0.
  - For the upper VA range, TCR_ELx.TG1.
- For a stage 2 translation in the Non-secure EL1&0 translation regime, VTCR_EL2.TG0.
- For a stage 2 translation in the Secure EL1&0 translation regime, one of the following:
  - If the stage 2 IA is a Non-secure IPA, then VTCR_EL2.TG0.
  - If the stage 2 IA is a Secure IPA, then VSTCR_EL2.TG0.

Translation table alignment

A translation table is required to be aligned to one of the following:

- If the translation table has eight or more entries, then it is aligned to the translation table size.
- If the translation table has fewer than eight entries, then it is aligned to 64 bytes.

Only when all of the following are true is it possible to have fewer than 8 translation table entries:

- The translation table is at the initial lookup level.
- Concatenated translation tables are not used.

If concatenated translation tables are used, then the concatenated translation tables are required to be aligned to the overall size of the memory occupied by the concatenated translation tables.

Translation table lookup endianness

When a translation table lookup reads a translation table entry, the read is 8-byte single-copy atomic.

The endianness of a translation table lookup is determined by the appropriate SCTLR_ELx.EE bit.

Changing an SCTLR_ELx.EE bit requires synchronization before the change is guaranteed to be visible to subsequent operations.

For more information, see *Synchronization requirements for AArch64 System registers* on page D17-5547.

Arm does not recommend changing an SCTLR_ELx.EE bit unless one or more of the following apply:

- The translation table lookups affected by the modified SCTLR_ELx.EE bit belong to an out-of-context translation regime.
- For any translation stage affected by the modified SCTLR_ELx.EE bit, address translation is disabled.

Translation table lookup memory attributes

For a translation table lookup in an address translation stage, the TCR_ELx and VTCR_EL2 registers determine the memory Cacheability and Shareability attributes that apply.

For a translation table lookup in an address translation stage, the required memory type is Normal memory.

For a two stage translation regime, when a translation table lookup from stage 1 occurs, all of the following apply:

- Arm recommends that the stage 2 translation of the stage 1 translation table lookup does not map to Device memory.
- Software can configure HCR_EL2.PTW to protect stage 2 table walks from mapping stage 1 translation tables to Device memory.
D8.2 Translation process

Stage 2 fault on a stage 1 translation table walk

Mismatched memory attributes

D8.2.6 Translation table walk properties

Ordering of memory accesses from translation table walks

A translation table walk is considered to be a separate observer. An explicit write to the translation tables might be observed by that separate observer for either of the following:
- A translation table walk caused by a different explicit write generated by the same instruction.
- A translation table walk caused by an explicit access generated by any instruction appearing in program order after the instruction doing the explicit write to the translation table.

The explicit write to the translation tables is guaranteed to be observable, to the extent required by the Shareability attributes, only after the execution of a DSB instruction. This DSB instruction and the instruction that performed the explicit write to the translation tables must have been executed by the same PE.

Any writes to the translation tables are not observed by any translation table accesses associated with an explicit memory access generated by a load or store that occurs in program order before the instruction that performs the write to the translation tables.

If FEAT_ETS is implemented, and a memory access RW1 is Ordered-before a second memory access RW2, then RW1 is also Ordered-before any translation table walk generated by RW2 that generates any of the following:
- A Translation fault.
- An Address size fault.
- An Access flag fault.

For more information, see:
- Ordering relations on page B2-163.
- Ordering constraints on page B2-165.

Security state of translation table lookups

For a Non-secure translation regime, all translation table lookups are done to Non-secure IPA or PA space.

When EL2 is enabled, the OA space of the Secure EL1&0 stage 1 translation regime is IPA space. In all other cases, the OA space for a Secure stage 1 translation regime is PA space.

For a stage 1 translation table walk in a Secure translation regime, the OA space of the translation table lookups is determined by all of the following:
- The initial translation table lookup is done to Secure OA space.
- When the Table descriptor lookup is to Secure OA space, the descriptor NSTable bit determines one of the following:
  - If NSTable is zero, the next translation table lookup is made to Secure OA space.
  - If NSTable is one, the next translation table lookup is made to Non-secure OA space.
When the Table descriptor lookup is to Non-secure OA space, the next translation table lookup is made to Non-secure OA space.

For more information, see Hierarchical control of Secure or Non-secure memory accesses on page D8-5135.

For a stage 2 translation in a Secure translation regime, the PA space of translation table lookups is determined by one of the following:

- When translating an address in Secure IPA space, the Effective value of the VSTCR_EL2.SW bit.
- When translating an address in Non-secure IPA space, the Effective value of the VTCR_EL2.NSW bit.

When an IPA from a Secure translation regime is translated by stage 2 translation, the output PA space of the translation is determined by one of the following:

- When translating an address in Secure IPA space, the Effective value of the VSTCR_EL2.SA bit.
- When translating an address in Non-secure IPA space, the Effective value of the VTCR_EL2.NSA bit.

Translation using the 4KB granule

Address translations that use the 4KB granule have a 4KB page size. Depending on the settings and supported features, up to 40 address bits are resolved using up to 5 lookup levels.

Throughout this section, if an address translation stage is not specified, then references to the Effective value of TCR_ELx.DS also apply to VTCR_EL2.DS.

For the 4KB translation granule, the maximum VA and PA supported by a translation regime is one of the following:

- If the Effective value of TCR_ELx.DS is 0, then the maximum VA and PA supported is 48 bits.
- If the Effective value of TCR_ELx.DS is 1, then the maximum VA and PA supported is 52 bits.

For the 4KB translation granule, if the Effective value of TCR_ELx.DS is 0, then OA[51:48] are treated as 0b0000.

For each lookup level supported by the 4KB translation granule, the following table describes the translation table properties at that level.

<table>
<thead>
<tr>
<th>Lookup level</th>
<th>Index into translation table</th>
<th>Maximum entries in table</th>
<th>Contents of translation table entries</th>
<th>Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-</td>
<td>-</td>
<td>Lookup level not supported</td>
<td>Effective value of TCR_ELx.DS is 0</td>
</tr>
<tr>
<td>IA[51:48]</td>
<td>16</td>
<td>Table descriptors</td>
<td></td>
<td>Effective value of TCR_ELx.DS is 1</td>
</tr>
<tr>
<td>0</td>
<td>IA[47:39]</td>
<td>512</td>
<td>Table descriptors</td>
<td>Effective value of TCR_ELx.DS is 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Table descriptors and Block descriptors</td>
<td>Effective value of TCR_ELx.DS is 1</td>
</tr>
<tr>
<td>1</td>
<td>IA[38:30]</td>
<td>512</td>
<td>Table descriptors and Block descriptors</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>IA[29:21]</td>
<td>512</td>
<td>Table descriptors and Block descriptors</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>IA[20:12]</td>
<td>512</td>
<td>Page descriptors</td>
<td>-</td>
</tr>
</tbody>
</table>

For the 4KB translation granule, a translation resolved by a Page descriptor at lookup level 3 has all of the following properties:

- The page size is 4KB.
- The translation can resolve a page using one of the following maximum address ranges:
  - If the Effective value of TCR_ELx.DS is 0, then IA[47:12].
  - If the Effective value of TCR_ELx.DS is 1, then IA[51:12].
- The page is addressed by one of the following:
  - If the Effective value of TCR_ELx.DS is 0, then OA[47:12].
If the Effective value of TCR_ELx.DS is 1, then OA[51:12].

- IA[11:0] is mapped directly to OA[11:0].

For the 4KB translation granule, a translation resolved by a Block descriptor at the specified lookup level has all of the properties shown in the following table:

### Table D8-10 4KB granule block descriptor properties at each lookup level

<table>
<thead>
<tr>
<th>Lookup level</th>
<th>Size of memory region addressed by Block descriptor</th>
<th>Bit range that is direct mapped from IA to OA</th>
<th>IA bit range that selects Block descriptor</th>
<th>Effective value of TCR_ELx.DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Not supported (512GB)</td>
<td>-</td>
<td>IA[38:0] maps to OA[38:0]</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1GB</td>
<td>IA[29:0] maps to OA[29:0]</td>
<td>IA[47:30]</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IA[51:30]</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2MB</td>
<td>IA[20:0] maps to OA[20:0]</td>
<td>IA[47:21]</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IA[51:21]</td>
<td>1</td>
</tr>
</tbody>
</table>

For the 4KB translation granule, the following figure shows how the maximum 52-bit IA size is resolved. An IA larger than 48 bits requires that the Effective value of TCR_ELx.DS is 1.

### Figure D8-3 52-bit IA resolved using 4KB translation granule
Stage 1 address translation using the 4KB translation granule

For the 4KB translation granule, when a stage 1 translation table walk is started, the initial lookup level is determined by the value of the TCR_ELx.TnSZ field as shown in the following table.

<table>
<thead>
<tr>
<th>Initial lookup level</th>
<th>TnSZ minimum value</th>
<th>Maximum IA bits resolved</th>
<th>TnSZ maximum value</th>
<th>Minimum IA bits resolved</th>
<th>Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>12</td>
<td>IA[51:12]</td>
<td>15</td>
<td>IA[48:12]</td>
<td>Effective value of TCR_ELx.DS is 1</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>IA[47:12]</td>
<td>24</td>
<td>IA[39:12]</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>25</td>
<td>IA[38:12]</td>
<td>33</td>
<td>IA[30:12]</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>34</td>
<td>IA[29:12]</td>
<td>39</td>
<td>IA[24:12]</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>IA[23:12]</td>
<td>42</td>
<td>IA[21:12]</td>
<td>FEAT_TTST is implemented</td>
</tr>
<tr>
<td>3</td>
<td>43</td>
<td>IA[20:12]</td>
<td>48</td>
<td>IA[15:12]</td>
<td>FEAT_TTST is implemented</td>
</tr>
</tbody>
</table>

For a stage 1 translation in the 4KB translation granule, depending on the IA size, the initial lookup level is indexed by up to 9 bits and all remaining lookup levels are indexed by exactly 9 bits.

For the 4KB translation granule and a 52-bit IA, the following figure is a generalized view of a stage 1 address translation. An IA and an OA larger than 48 bits requires that the Effective value of TCR_ELx.DS is 1. The 512GB block size requires that the Effective value of TCR_ELx.DS is 1.

Key:
- D_Table is a Table descriptor.
- D_Block is a Block descriptor.
- D_Page is a Page descriptor.
  - a Indexed by IA[n:48], where IA width is (n+1) bits.
  - b Indexed by IA[47:39].
  - c Indexed by IA[38:30].
  - d Indexed by IA[29:21].
  - e Indexed by IA[20:12].

Figure D8-4 Generalized view of a stage 1 address translation using the 4KB granule
Stage 2 address translation using the 4KB translation granule

For the 4KB translation granule, when a stage 2 translation table walk is started, the initial lookup level is determined by one of the following:

- If the Effective value of VTCR_EL2.DS is 0, then the initial lookup level is determined by one of VTCR_EL2.SL0 or VSTCR_EL2.SL0, and VTCR_EL2.SL2 and VSTCR_EL2.SL2 are RES0.
- If the Effective value of VTCR_EL2.DS is 1, then the initial lookup level is determined by the combination of VTCR_EL2.SL0 and VTCR_EL2.SL2, or VSTCR_EL2.SL0 and VSTCR_EL2.SL2.

For the 4KB translation granule, when a stage 2 translation table walk is started, the following table shows the initial lookup level determined by VTCR_EL2.SL0 or VSTCR_EL2.SL0 and, when the Effective value of VTCR_EL2.DS is 1, the corresponding VTCR_EL2.SL2 or VSTCR_EL2.SL2.

<table>
<thead>
<tr>
<th>SL2</th>
<th>SL0</th>
<th>Initial lookup level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b00</td>
<td>Level 2</td>
</tr>
<tr>
<td>0b0</td>
<td>0b01</td>
<td>Level 1</td>
</tr>
<tr>
<td>0b0</td>
<td>0b10</td>
<td>Level 0</td>
</tr>
<tr>
<td>0b0</td>
<td>0b11</td>
<td>Level 3</td>
</tr>
<tr>
<td>0b1</td>
<td>0b00</td>
<td>Level -1</td>
</tr>
<tr>
<td>0b1</td>
<td>0b01</td>
<td>Reserved</td>
</tr>
<tr>
<td>0b1</td>
<td>0b10</td>
<td>Reserved</td>
</tr>
<tr>
<td>0b1</td>
<td>0b11</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

For the 4KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is -1, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

<table>
<thead>
<tr>
<th>Number of concatenated translation tables</th>
<th>IA bits resolved</th>
<th>T0SZ Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (1 table)</td>
<td>IA[51:12]-IA[48:12]</td>
<td>12-15 Effective value of VTCR_EL2.DS is 1</td>
</tr>
</tbody>
</table>

For the 4KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 0, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

<table>
<thead>
<tr>
<th>Number of concatenated translation tables</th>
<th>IA bits resolved</th>
<th>T0SZ Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (1 table)</td>
<td>IA[47:12]-IA[39:12]</td>
<td>16-24 -</td>
</tr>
<tr>
<td>2</td>
<td>IA[48:12]</td>
<td>15 Effective value of VTCR_EL2.DS is 1</td>
</tr>
<tr>
<td>4</td>
<td>IA[49:12]</td>
<td>14 Effective value of VTCR_EL2.DS is 1</td>
</tr>
<tr>
<td>8</td>
<td>IA[50:12]</td>
<td>13 Effective value of VTCR_EL2.DS is 1</td>
</tr>
<tr>
<td>16</td>
<td>IA[51:12]</td>
<td>12 Effective value of VTCR_EL2.DS is 1</td>
</tr>
</tbody>
</table>
For the 4KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 1, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

### Table D8-15 4KB granule, stage 2 initial lookup at level 1

<table>
<thead>
<tr>
<th>Number of concatenated translation tables</th>
<th>IA bits resolved</th>
<th>T0SZ</th>
<th>Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (1 table)</td>
<td>IA[38:12]-IA[30:12]</td>
<td>25-33</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IA[39:12]</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>IA[40:12]</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>IA[41:12]</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>IA[42:12]</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

For the 4KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 2, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

### Table D8-16 4KB granule, stage 2 initial lookup at level 2

<table>
<thead>
<tr>
<th>Number of concatenated translation tables</th>
<th>IA bits resolved</th>
<th>T0SZ</th>
<th>Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (1 table)</td>
<td>IA[23:12]-IA[21:12]</td>
<td>40-42</td>
<td>FEAT_TTST is implemented</td>
</tr>
<tr>
<td>None (1 table)</td>
<td>IA[29:12]-IA[24:12]</td>
<td>34-39</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IA[30:12]</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>IA[31:12]</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>IA[32:12]</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>IA[33:12]</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

For the 4KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 3, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

### Table D8-17 4KB granule, stage 2 initial lookup at level 3

<table>
<thead>
<tr>
<th>Number of concatenated translation tables</th>
<th>IA bits resolved</th>
<th>T0SZ</th>
<th>Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (1 table)</td>
<td>IA[20:12]-IA[15:12]</td>
<td>43-48</td>
<td>FEAT_TTST is implemented</td>
</tr>
<tr>
<td>2</td>
<td>IA[21:12]</td>
<td>42</td>
<td>FEAT_TTST is implemented</td>
</tr>
<tr>
<td>4</td>
<td>IA[22:12]</td>
<td>41</td>
<td>FEAT_TTST is implemented</td>
</tr>
<tr>
<td>8</td>
<td>IA[23:12]</td>
<td>40</td>
<td>FEAT_TTST is implemented</td>
</tr>
<tr>
<td>16</td>
<td>IA[24:12]</td>
<td>39</td>
<td>FEAT_TTST is implemented</td>
</tr>
</tbody>
</table>

For the 4KB translation granule and a 52-bit IA, the following figure is a generalized view of a stage 2 address translation. An IA larger than 48 bits requires that the Effective value of VTCR_EL2.DS is 1. The 512GB block size requires that the Effective value of VTCR_EL2.DS is 1.
VTCCR_EL2.{SL2, SL0} defines the start level.

Starting at level -1

VTBTR_EL2

Level -1 table

D_Table

Level 0 table

D_Table

512GB region

D_Block

1GB region

D_Table

4KB memory page

Level 1 table

D_Block

D_Table

2MB region

D_Page

4KB memory page

Level 2 table

D_Block

D_Table

2MB region

D_Page

4KB memory page

Level 3 table

D_Block

D_Table

4KB memory page

Key for both diagrams:

D_Table is a Table descriptor.
D_Block is a Block descriptor.
D_Page is a Page descriptor.

- a Indexed by IA[47:39], where IA width is (n+1) bits.
- b1 Indexed by IA[47:39].
- b2 Indexed by IA[39:30], where IA width is (n+1) bits.
- c Indexed by IA[38:30].
- d Indexed by IA[29:21].
- e Indexed by IA[20:12].

For the 4KB translation granule, when a stage 2 translation table walk is started and one of the following is true, a stage 2 level 0 Translation fault is generated:

- If the Effective value of VTCR_EL2.DS is 0, then one or more of the following is true:
  - The VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value is not consistent with the corresponding VTCR_EL2.SL0 or VSTCR_EL2.SL0 value.
  - VTCR_EL2.SL0 or VSTCR_EL2.SL0 is programmed to a reserved value.

- If the Effective value of VTCR_EL2.DS is 1, then one or more of the following is true:
  - The VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value is not consistent with the corresponding VTCR_EL2.{SL2, SL0} or VSTCR_EL2.{SL2, SL0} value.
  - VTCR_EL2.{SL2, SL0} or VSTCR_EL2.{SL2, SL0} is programmed to a reserved value.
Finding the descriptor when using the 4KB translation granule

For the 4KB translation granule, the following table shows the algorithm to find the descriptor address at each supported lookup level, using all of the following information:

- The translation table base address, \textit{BaseAddr}.
- The number of bits in the supported PA size, \textit{PAsize}.
- The IA supplied to the translation stage and used as an index into the translation table.
- For each initial lookup level, the permitted range of values for TCR\_ELx.TnSZ.
- For a stage 2 translation, all of the following:
  - The value of VTCR\_EL2.T0SZ or VSTCR\_EL2.T0SZ.
  - The value of VTCR\_EL2.SL0 or VSTCR\_EL2.SL0.
  - If the Effective value of VTCR\_EL2.DS is 1, then the value of VTCR\_EL2.SL2 or VSTCR\_EL2.SL2.

### Table D8-18 4KB granule, finding the descriptor address

<table>
<thead>
<tr>
<th>Lookup level</th>
<th>Stage 1 translation table descriptor address</th>
<th>Stage 2 translation table descriptor address</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>if 12 &lt;= TnSZ &lt;= 15 then ( x = (19 - \text{TnSZ}) )(</td>
<td>\text{BaseAddr[\text{PAsize-1:x}]:IA[x+44:48]:0b000}</td>
</tr>
<tr>
<td>0</td>
<td>if 16 &lt;= TnSZ &lt;= 24 then ( x = (28 - \text{TnSZ}) )(</td>
<td>\text{BaseAddr[\text{PAsize-1:x}]:IA[x+35:39]:0b000}</td>
</tr>
<tr>
<td>1</td>
<td>if 25 &lt;= TnSZ &lt;= 33 then ( x = (37 - \text{TnSZ}) )(</td>
<td>\text{BaseAddr[\text{PAsize-1:x}]:IA[x+26:30]:0b000}</td>
</tr>
<tr>
<td>2</td>
<td>if 34 &lt;= TnSZ &lt;= 42 then ( x = (46 - \text{TnSZ}) )(</td>
<td>\text{BaseAddr[\text{PAsize-1:x}]:IA[x+17:21]:0b000}</td>
</tr>
<tr>
<td>3</td>
<td>if 43 &lt;= TnSZ &lt;= 48 then ( x = (55 - \text{TnSZ}) )(</td>
<td>\text{BaseAddr[\text{PAsize-1:x}]:IA[x+8:12]:0b000}</td>
</tr>
</tbody>
</table>

D8.2.8 Translation using the 16KB granule

Address translations that use the 16KB granule have a 16KB page size. Depending on the settings and supported features, up to 38 address bits are resolved using up to 4 lookup levels.
Throughout this section, if an address translation stage is not specified, then references to the Effective value of TCR_ELx.DS also apply to VTCR_EL2.DS.

For the 16KB translation granule, the maximum VA and PA supported by a translation regime is one of the following:

- If the Effective value of TCR_ELx.DS is 0, then the maximum VA and PA supported is 48 bits.
- If the Effective value of TCR_ELx.DS is 1, then the maximum VA and PA supported is 52 bits.

For the 16KB translation granule, if the Effective value of TCR_ELx.DS is 0, then OA[51:48] are treated as 0b0000.

For each lookup level supported by the 16KB translation granule, the following table describes the translation table properties at that level.

<table>
<thead>
<tr>
<th>Lookup level</th>
<th>Index into translation table</th>
<th>Maximum entries in table</th>
<th>Contents of translation table entries</th>
<th>Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IA[47]</td>
<td>2</td>
<td>Table descriptors</td>
<td>Effective value of TCR_ELx.DS is 0</td>
</tr>
<tr>
<td></td>
<td>IA[51:47]</td>
<td>32</td>
<td>Table descriptors</td>
<td>Effective value of TCR_ELx.DS is 1</td>
</tr>
<tr>
<td>1</td>
<td>IA[46:36]</td>
<td>2048</td>
<td>Table descriptors</td>
<td>Effective value of TCR_ELx.DS is 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Table descriptors and Block descriptors</td>
<td>Effective value of TCR_ELx.DS is 1</td>
</tr>
<tr>
<td>2</td>
<td>IA[35:25]</td>
<td>2048</td>
<td>Table descriptors and Block descriptors</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>IA[24:14]</td>
<td>2048</td>
<td>Page descriptors</td>
<td>-</td>
</tr>
</tbody>
</table>

For the 16KB translation granule, a translation resolved by a Page descriptor at lookup level 3 has all of the following properties:

- The page size is 16KB.
- The translation can resolve a page using one of the following maximum address ranges:
  - If the Effective value of TCR_ELx.DS is 0, then IA[47:14].
  - If the Effective value of TCR_ELx.DS is 1, then IA[51:14].
- The page is addressed by one of the following:
  - If the Effective value of TCR_ELx.DS is 0, then OA[47:14].
  - If the Effective value of TCR_ELx.DS is 1, then OA[51:14].
- IA[13:0] is mapped directly to OA[13:0].

For the 16KB translation granule, a translation resolved by a Block descriptor at the specified lookup level has all of the properties shown in the following table:

<table>
<thead>
<tr>
<th>Lookup level</th>
<th>Size of memory region addressed by Block descriptor</th>
<th>Bit range that is directly mapped from IA to OA</th>
<th>IA bit range that selects Block descriptor</th>
<th>Effective value of TCR_ELx.DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Not supported</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>64GB</td>
<td>IA[35:0] maps to OA[35:0]</td>
<td>IA[51:36]</td>
<td>1</td>
</tr>
</tbody>
</table>
For the 16KB translation granule, the following figure shows how the maximum 52-bit IA size is resolved. An IA larger than 48 bits requires that the Effective value of TCR_ELx.DS is 1.

![Input address (IA)](image)

**Figure D8-6 52-bit IA resolved using 16KB translation granule**

### Stage 1 address translation using the 16KB translation granule

For the 16KB translation granule, when a stage 1 translation table walk is started, the initial lookup level is determined by the value of the TCR_ELx.TnSZ field as shown in the following table:

<table>
<thead>
<tr>
<th>Initial lookup level</th>
<th>TnSZ minimum value</th>
<th>Maximum IA bits resolved</th>
<th>TnSZ maximum value</th>
<th>Minimum IA bits resolved</th>
<th>Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>IA[51:14]</td>
<td>15</td>
<td>IA[48:14]</td>
<td>Effective value of TCR_ELx.DS is 1</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>IA[47:14]</td>
<td>16</td>
<td>IA[47:14]</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>17</td>
<td>IA[46:14]</td>
<td>27</td>
<td>IA[36:14]</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>28</td>
<td>IA[35:14]</td>
<td>38</td>
<td>IA[25:14]</td>
<td>-</td>
</tr>
</tbody>
</table>

For a stage 1 translation in the 16KB translation granule, depending on the IA size, the initial lookup level is indexed by up to 11 bits and all remaining lookup levels are indexed by exactly 11 bits.

For the 16KB translation granule and a 52-bit IA, the following figure is a generalized view of a stage 1 address translation. An IA and an OA larger than 48 bits requires that the Effective value of TCR_ELx.DS is 1. The 64GB block size requires that the Effective value of TCR_ELx.DS is 1.

![Generalized view](image)
Figure D8-7 Generalized view of a stage 1 address translation using the 16KB granule

### Stage 2 address translation using the 16KB translation granule

For the 16KB translation granule, when a stage 2 translation table walk is started, the initial lookup level is determined by the corresponding VTCR_EL2.SL0 or VSTCR_EL2.SL0 value.

For the 16KB translation granule, when a stage 2 translation table walk is started, the following table shows the initial lookup level determined by SL0:

<table>
<thead>
<tr>
<th>SL0</th>
<th>Initial lookup level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>Level 3</td>
</tr>
<tr>
<td>0b01</td>
<td>Level 2</td>
</tr>
<tr>
<td>0b10</td>
<td>Level 1</td>
</tr>
<tr>
<td>0b11</td>
<td>Level 0</td>
</tr>
</tbody>
</table>

For the 16KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 0, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

<table>
<thead>
<tr>
<th>Number of concatenated translation tables</th>
<th>IA bits resolved</th>
<th>T0SZ</th>
<th>Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (1 table)</td>
<td>IA[51:14]-IA[47:14]</td>
<td>12-16</td>
<td>Effective value of VTCR_EL2.DS is 1</td>
</tr>
</tbody>
</table>
For the 16KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 1, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

**Table D8-24 16KB granule, stage 2 initial lookup at level 1**

<table>
<thead>
<tr>
<th>Number of concatenated translation tables</th>
<th>IA bits resolved</th>
<th>T0SZ</th>
<th>Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (1 table)</td>
<td>IA[46:14]-IA[36:14]</td>
<td>17-27</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>IA[47:14]</td>
<td>16</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>IA[48:14]</td>
<td>15</td>
<td>Effective value of VTCR_EL2.DS is 1</td>
</tr>
<tr>
<td>8</td>
<td>IA[49:14]</td>
<td>14</td>
<td>Effective value of VTCR_EL2.DS is 1</td>
</tr>
<tr>
<td>16</td>
<td>IA[50:14]</td>
<td>13</td>
<td>Effective value of VTCR_EL2.DS is 1</td>
</tr>
</tbody>
</table>

For the 16KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 2, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

**Table D8-25 16KB granule, stage 2 initial lookup at level 2**

<table>
<thead>
<tr>
<th>Number of concatenated translation tables</th>
<th>IA bits resolved</th>
<th>T0SZ</th>
<th>Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (1 table)</td>
<td>IA[35:14]-IA[25:14]</td>
<td>28-38</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>IA[36:14]</td>
<td>27</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>IA[37:14]</td>
<td>26</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>IA[38:14]</td>
<td>25</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>IA[39:14]</td>
<td>24</td>
<td>-</td>
</tr>
</tbody>
</table>

For the 16KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 3, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

**Table D8-26 16KB granule, stage 2 initial lookup at level 3**

<table>
<thead>
<tr>
<th>Number of concatenated translation tables</th>
<th>IA bits resolved</th>
<th>T0SZ</th>
<th>Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (1 table)</td>
<td>IA[23:14]-IA[15:14]</td>
<td>40-48</td>
<td>FEAT_TTST is implemented</td>
</tr>
<tr>
<td>None (1 table)</td>
<td>IA[24:14]</td>
<td>39</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>IA[25:14]</td>
<td>38</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>IA[26:14]</td>
<td>37</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>IA[27:14]</td>
<td>36</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>IA[28:14]</td>
<td>35</td>
<td>-</td>
</tr>
</tbody>
</table>

For the 16KB translation granule and a 52-bit IA, the following figure is a generalized view of a stage 2 address translation. An IA larger than 48 bits requires that the Effective value of VTCR_EL2.DS is 1. The 64GB block size requires that the Effective value of VTCR_EL2.DS is 1.
D8.2 Translation process

Figure D8-8 Generalized view of a stage 2 address translation using the 16KB granule

For the 16KB translation granule, when a stage 2 translation table walk is started and one of the following is true, a stage 2 level 0 Translation fault is generated:

- The VTCR_EL2.TOSZ or VSTCR_EL2.TOSZ value is not consistent with the corresponding VTCR_EL2.SL0 or VSTCR_EL2.SL0 value.
- VTCR_EL2.SL0 or VSTCR_EL2.SL0 is programmed to a reserved value.

For more information, see Concatenated translation tables on page D8-5094.

Finding the descriptor when using the 16KB translation granule

For the 16KB translation granule, the following table shows the algorithm to find the descriptor address at each supported lookup level, using all of the following information:

- The translation table base address, BaseAddr.
- The number of bits in the supported PA size, PAsize.
- The IA supplied to the translation stage and used as an index into the translation table.
For each initial lookup level, the permitted range of values for TCR_ELx.TnSZ.

For a stage 2 translation, all of the following:

- The value of VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ.
- The value of VTCR_EL2.SL0 or VSTCR_EL2.SL0.

### Table D8-27 16KB granule, finding the descriptor address

<table>
<thead>
<tr>
<th>Lookup level</th>
<th>Stage 1 translation table descriptor address</th>
<th>Stage 2 translation table descriptor address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>if 12 &lt;= TnSZ &lt;= 16 then x = (20 - TnSZ)</td>
<td>if SL0==3 then if 12 &lt;= T0SZ &lt;= 16 then x = (20 - T0SZ)</td>
</tr>
<tr>
<td></td>
<td>BaseAddr[PAsize-1:x]:IA[x+43:47]:0b000</td>
<td>BaseAddr[PAsize-1:x]:IA[x+43:47]:0b000</td>
</tr>
<tr>
<td>1</td>
<td>if 17 &lt;= TnSZ &lt;= 27 then x = (31 - TnSZ)</td>
<td>if SL0==2 then if 13 &lt;= T0SZ &lt;= 27 then x = (31 - T0SZ)</td>
</tr>
<tr>
<td></td>
<td>else x = 14</td>
<td>else x = 14</td>
</tr>
<tr>
<td></td>
<td>BaseAddr[PAsize-1:x]:IA[x+32:36]:0b000</td>
<td>BaseAddr[PAsize-1:x]:IA[x+32:36]:0b000</td>
</tr>
<tr>
<td>2</td>
<td>if 28 &lt;= TnSZ &lt;= 38 then x = (42 - TnSZ)</td>
<td>if SL0==1 then if 24 &lt;= T0SZ &lt;= 38 then x = (42 - T0SZ)</td>
</tr>
<tr>
<td></td>
<td>else x = 14</td>
<td>else x = 14</td>
</tr>
<tr>
<td></td>
<td>BaseAddr[PAsize-1:x]:IA[x+21:25]:0b000</td>
<td>BaseAddr[PAsize-1:x]:IA[x+21:25]:0b000</td>
</tr>
<tr>
<td>3</td>
<td>if 39 &lt;= TnSZ &lt;= 48 then x = (53 - TnSZ)</td>
<td>if SL0==0 then if 35 &lt;= T0SZ &lt;= 48 then x = (53 - T0SZ)</td>
</tr>
<tr>
<td></td>
<td>else x = 14</td>
<td>else x = 14</td>
</tr>
<tr>
<td></td>
<td>BaseAddr[PAsize-1:x]:IA[x+10:14]:0b000</td>
<td>BaseAddr[PAsize-1:x]:IA[x+10:14]:0b000</td>
</tr>
</tbody>
</table>

### D8.2.9 Translation using the 64KB granule

**IVDPWL** Address translations that use the 64KB granule have a 64KB page size. Depending on the settings and supported features, up to 36 address bits are resolved using up to 3 lookup levels.

**ILRFQD** For the 64KB translation granule, the maximum VA size supported by a translation regime is one of the following:

- If FEAT_LVA is not implemented, then the maximum VA supported is 48 bits.
- If FEAT_LVA is implemented, then the maximum VA supported is 52 bits.

**IYZDPC** For the 64KB translation granule, the maximum PA size supported by a translation regime is one of the following:

- If FEAT_LPA is not implemented, then the maximum PA supported is 48 bits.
- If FEAT_LPA is implemented, then the maximum PA supported is 52 bits.

**RNSQXP** For the 64KB translation granule, if FEAT_LPA is not implemented, then OA[51:48] are treated as 0b0000.
For each lookup level supported by the 64KB translation granule, the following table describes the translation table properties at that level.

### Table D8-28 64KB granule translation table properties at each lookup level

<table>
<thead>
<tr>
<th>Lookup level</th>
<th>Index into translation table</th>
<th>Maximum entries in table</th>
<th>Contents of translation table entries</th>
<th>Additional requirements for stage 1</th>
<th>Additional requirements for stage 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IA[47:42]</td>
<td>64</td>
<td>Table descriptors</td>
<td>-</td>
<td>FEAT_LPA</td>
</tr>
<tr>
<td></td>
<td>IA[51:42]</td>
<td>1024</td>
<td>Table descriptors and Block descriptors</td>
<td>FEAT_LPA</td>
<td>FEAT_LPA</td>
</tr>
<tr>
<td>2</td>
<td>IA[41:29]</td>
<td>8192</td>
<td>Table descriptors and Block descriptors</td>
<td>FEAT_LVA, FEAT_LPA</td>
<td>FEAT_LPA</td>
</tr>
<tr>
<td>3</td>
<td>IA[28:16]</td>
<td>8192</td>
<td>Page descriptors</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

For the 64KB translation granule, a translation resolved by a Page descriptor at lookup level 3 has all of the following properties:

- The page size is 64KB.
- For stage 1, the translation can resolve a page using one of the following maximum address ranges:
  - If FEAT_LVA is not implemented, then IA[47:16].
  - If FEAT_LVA is implemented, then IA[51:16].
- For stage 2, the translation can resolve a page using one of the following maximum address ranges:
  - If FEAT_LPA is not implemented, then IA[47:16].
  - If FEAT_LPA is implemented, then IA[51:16].
- For stage 1 and stage 2, the page is addressed by one of the following:
  - If FEAT_LPA is not implemented, then OA[47:16].
  - If FEAT_LPA is implemented, then OA[51:16].
- IA[15:0] is mapped directly to OA[15:0].

For the 64KB translation granule, a translation resolved by a Block descriptor at the specified lookup level has all of the properties shown in the following table:

### Table D8-29 64KB granule block descriptor properties at each lookup level

<table>
<thead>
<tr>
<th>Lookup level</th>
<th>Size of memory region addressed by Block descriptor</th>
<th>Bit range that is direct mapped from IA to OA</th>
<th>IA bit range that selects Block descriptor</th>
<th>Additional requirements for stage 1</th>
<th>Additional requirements for stage 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4TB</td>
<td>IA[41:0] maps to OA[41:0]</td>
<td>IA[47:42]</td>
<td>FEAT_LPA</td>
<td>FEAT_LPA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IA[51:42]</td>
<td>FEAT_LVA, FEAT_LPA</td>
<td>FEAT_LPA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IA[51:29]</td>
<td>FEAT_LPA</td>
<td>FEAT_LPA</td>
</tr>
</tbody>
</table>

For the 64KB translation granule, the following figure shows how the maximum 52-bit IA size is resolved. For a stage 1 translation, an IA larger than 48 bits requires the implementation of FEAT_LVA. For a stage 2 translation, an IA larger than 48 bits requires the implementation of FEAT_LPA.
### Figure D8-9 52-bit IA resolved using 64KB translation granule

**Stage 1 address translation using the 64KB translation granule**

For the 64KB translation granule, when a stage 1 translation table walk is started, the initial lookup level is determined by the value of the TCR_ELx.TnSZ field as shown in the following table:

<table>
<thead>
<tr>
<th>Initial lookup level</th>
<th>TnSZ minimum value</th>
<th>Maximum IA bits resolved</th>
<th>TnSZ maximum value</th>
<th>Minimum IA bits resolved</th>
<th>Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>IA[51:16]</td>
<td>15</td>
<td>IA[48:16]</td>
<td>FEAT_LVA is implemented</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>IA[47:16]</td>
<td>21</td>
<td>IA[42:16]</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>IA[41:16]</td>
<td>34</td>
<td>IA[29:16]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td>IA[23:16]</td>
<td>47</td>
<td>IA[16:16]</td>
<td>FEAT_TTST is implemented</td>
</tr>
</tbody>
</table>

For a stage 1 translation in the 64KB translation granule, depending on the IA size, the initial lookup level is indexed by up to 13 bits and all remaining lookup levels are indexed by exactly 13 bits.

For the 64KB translation granule and a 52-bit IA, the following figure is a generalized view of a stage 1 address translation. An IA larger than 48 bits requires implementation of FEAT_LVA, and an OA larger than 48 bits requires implementation of FEAT_LPA. The 4TB block size requires implementation of FEAT_LPA.
Stage 2 address translation using the 64KB translation granule

For the 64KB translation granule, when a stage 2 translation table walk is started, the initial lookup level is determined by the corresponding VTCR_EL2.SL0 or VSTCR_EL2.SL0 value.

For the 64KB translation granule, when a stage 2 translation table walk is started, the following table shows the initial lookup level determined by SL0:

<table>
<thead>
<tr>
<th>SL0</th>
<th>Initial lookup level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>Level 3</td>
</tr>
<tr>
<td>0b01</td>
<td>Level 2</td>
</tr>
<tr>
<td>0b10</td>
<td>Level 1</td>
</tr>
<tr>
<td>0b11</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

For the 64KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 1, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

<table>
<thead>
<tr>
<th>Number of concatenated translation tables</th>
<th>IA bits resolved</th>
<th>T0SZ</th>
<th>Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (1 table)</td>
<td>IA[51:16]-IA[48:16]</td>
<td>12-15</td>
<td>FEAT_LPA is implemented</td>
</tr>
<tr>
<td>None (1 table)</td>
<td>IA[47:16]-IA[42:16]</td>
<td>16-21</td>
<td>-</td>
</tr>
</tbody>
</table>
For the 64KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 2, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

### Table D8-33 64KB granule, stage 2 initial lookup at level 2

<table>
<thead>
<tr>
<th>Number of concatenated translation tables</th>
<th>IA bits resolved</th>
<th>T0SZ</th>
<th>Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (1 table)</td>
<td>IA[41:16]-IA[29:16]</td>
<td>22-34</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IA[42:16]</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>IA[43:16]</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>IA[44:16]</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>IA[45:16]</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>

For the 64KB translation granule, when a stage 2 translation table walk is started and the initial lookup level is 3, the following table shows all of the permitted concatenated translation table configurations and the corresponding VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value:

### Table D8-34 64KB granule, stage 2 initial lookup at level 3

<table>
<thead>
<tr>
<th>Number of concatenated translation tables</th>
<th>IA bits resolved</th>
<th>T0SZ</th>
<th>Additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (1 table)</td>
<td>IA[23:16]-IA[16:16]</td>
<td>40-47</td>
<td>FEAT_TTST is implemented</td>
</tr>
<tr>
<td>None (1 table)</td>
<td>IA[28:16]-IA[24:16]</td>
<td>35-39</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IA[29:16]</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>IA[30:16]</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>IA[31:16]</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>IA[32:16]</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

For the 64KB translation granule and a 52-bit IA, the following figure is a generalized view of a stage 2 address translation. An IA larger than 48 bits requires the implementation of FEAT_LPA. The 4TB block size requires implementation of FEAT_LPA.
Figure D8-11 Generalized view of a stage 2 address translation using the 64KB granule

For the 64KB translation granule, when a stage 2 translation table walk is started and one of the following is true, a stage 2 level 0 Translation fault is generated:

- The VTCR_EL2.T0SZ or VSTCR_EL2.T0SZ value is not consistent with the corresponding VTCR_EL2.SL0 or VSTCR_EL2.SL0 value.
- VTCR_EL2.SL0 or VSTCR_EL2.SL0 is programmed to a reserved value.

For more information, see Concatenated translation tables on page D8-5094.

Finding the descriptor when using the 64KB translation granule

For the 64KB translation granule, the following table shows the algorithm to find the descriptor address at each supported lookup level, using all of the following information:

- The translation table base address, BaseAddr.
- The number of bits in the supported PA size, PSize.
- The IA supplied to the translation stage and used as an index into the translation table.
- For each initial lookup level, the permitted range of values for TCR_ELx.TnSZ.
• For a stage 2 translation, all of the following:
  — The value of $\text{VTCR}_{\text{EL2}}.T0SZ$ or $\text{VSTCR}_{\text{EL2}}.T0SZ$.
  — The value of $\text{VTCR}_{\text{EL2}}.SL0$ or $\text{VSTCR}_{\text{EL2}}.SL0$.

### Table D8-35 64KB granule, finding the descriptor address

<table>
<thead>
<tr>
<th>Lookup level</th>
<th>Stage 1 translation table descriptor address</th>
<th>Stage 2 translation table descriptor address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>if $12 \leq TnSZ \leq 21$ then $x = (25 - TnSZ)$ BaseAddr[PASize-1:x]:IA[x+38:42]:0b000</td>
<td>if SL0==2 if $12 \leq T0SZ \leq 21$ then $x = (25 - T0SZ)$ BaseAddr[PASize-1:x]:IA[x+38:42]:0b000</td>
</tr>
<tr>
<td>2</td>
<td>if $22 \leq TnSZ \leq 34$ then $x = (38 - TnSZ)$ else $x = 16$ BaseAddr[PASize-1:x]:IA[x+25:29]:0b000</td>
<td>if SL0==1 if $18 \leq T0SZ \leq 34$ then $x = (38 - T0SZ)$ else $x = 16$ BaseAddr[PASize-1:x]:IA[x+25:29]:0b000</td>
</tr>
<tr>
<td>3</td>
<td>if $35 \leq TnSZ \leq 47$ then $x = (51 - TnSZ)$ else $x = 16$ BaseAddr[PASize-1:x]:IA[x+12:16]:0b000</td>
<td>if SL0==0 if $31 \leq T0SZ \leq 47$ then $x = (51 - T0SZ)$ else $x = 16$ BaseAddr[PASize-1:x]:IA[x+12:16]:0b000</td>
</tr>
</tbody>
</table>

### D8.2.10 The effects of disabling an address translation stage

#### IXHZJC

Stage 1 and stage 2 translations can be disabled independently, and doing so affects the MMU behavior.

#### Behavior when stage 1 address translation is disabled

If stage 1 address translation is disabled, then all of the following apply to memory accesses that would otherwise be translated at stage 1:

- The stage 1 IA is flat mapped to the OA.
- No Translation faults, Access flag faults, or Permission faults can be generated.
- Address size faults and Alignment faults can be generated.
- No memory is guarded.
- The access is to one of the following IPA or PA spaces:
  - For accesses from the Non-secure state, the access is to the Non-secure IPA or PA space.
  - For accesses from the Secure state, the access is to the Secure IPA or PA space.

#### RWFPZW

If stage 1 address translation is disabled, then the stage 1 translation assigns one of the following attributes to memory accesses:

- For memory accesses using the EL1&0 regime, if the effective value of $\text{HCR}_{\text{EL2}}.DC$ is 1, then all of the following memory attributes are assigned:
  - The Tagged attribute is set according to $\text{HCR}_{\text{EL2}}.DCT$.
  - Normal Inner Write-Back Cacheable Read-Allocate Write-Allocate.
  - Normal Outer Write-Back Cacheable Read-Allocate Write-Allocate.
  - Non-shareable.
  - If $\text{FEAT}_{\text{XS}}$ is implemented, then the XS attribute is set to 0.
• For all other memory accesses, one of the following memory attributes are assigned:
  — For a data access, the Device-nGnRnE memory attribute.
  — For an instruction access, the Normal memory attribute and one of the following:
    — If SCTLR_ELx.I is 0, then the Non-cacheable and Outer Shareable attributes.
    — If SCTLR_ELx.I is 1, then the Cacheable, Inner Write-Through Read-Allocate No
      Write-Allocate, Outer Write-Through Read-Allocate No Write-Allocate Outer Shareable
      attribute.
    — For data accesses and instruction accesses, if FEAT_XS is implemented, then the XS attribute is set to
      1.

\[\text{R}_{\text{CSDNK}}\]

If all of the following apply, then the stage 1 memory attribute assignments and OA can be modified by the stage 2 translation:

• EL1&0 stage 2 address translation is enabled.
• An access using the EL1&0 translation regime occurs.

\[\text{R}_{\text{LBPTY}}\]

If HCR_EL2.DC is 1, then all of the following apply:

• For all purposes other than reading the value of the bit, the Effective value of SCTLR_EL1.M is 0, disabling
  stage 1 address translation in the EL1&0 translation regime.
• If EL2 is enabled, then for all purposes other than reading the value of the bit, the Effective value of
  HCR_EL2.VM is 1, enabling stage 2 address translation in the EL1&0 translation regime.

**Behavior when stage 2 address translation is disabled**

\[\text{R}_{\text{QVLSD}}\]

If stage 2 address translation is disabled, then all of the following apply to memory accesses that would otherwise be translated at stage 2:

• No stage 2 MMU faults can be generated.
• The IPA is flat mapped to the PA.
• The memory attributes and permissions assigned by the stage 1 translation are assigned to the PA.
• The access is to one of the following PA spaces:
  — For accesses from the Non-secure IPA space, the access is to the Non-secure PA space.
  — For accesses from the Secure IPA space, the access is to the Secure PA space.

**Instruction fetch behavior when all translation stages are disabled**

\[\text{I}_{\text{DLNTB}}\]

If all associated address translation stages are disabled, then software is required to place instructions that will be executed in memory regions where those regions and the memory regions immediately following each have all the following properties:

• The memory region size is equal to the minimum implemented translation granule size.
• The memory region is tolerant to speculative accesses.
• The memory region is naturally aligned.

\[\text{R}_{\text{WLVZN}}\]

If execution is in AArch64 state and all associated address translation stages are disabled, a memory location might be accessed as a result of an instruction fetch, including a speculative instruction fetch, in all of the following cases:

• The memory location is in the same memory region, or in the next contiguous memory region, as an
  instruction that simple sequential program execution either requires to be fetched now or has required to be
  fetched since the last reset.
• The memory location is the target of a direct branch that simple sequential program execution would have
  taken since the most recent of one of the following:
  — The last reset.
  — The last synchronization of instruction cache maintenance targeting the branch instruction address.
If all address translation stages are disabled, then speculative instruction fetches can cause unintended memory location accesses, regardless of whether the fetched instruction is committed to execution.

**Effect of disabling address translation on maintenance and address translation instructions**

- If all address translation stages are disabled, speculative instruction fetches can cause unintended memory location accesses, regardless of whether the fetched instruction is committed to execution.

- Cache maintenance instructions act on the target location regardless of all of the following:
  - Whether or not any address translation stages are disabled.
  - The memory attribute values.

- For an address translation stage that is disabled, cache maintenance instructions use flat address mapping.

- If an address translation stage is disabled, TLB maintenance operations are not affected.

For more information, see:

- *A64 Cache maintenance instructions* on page D7-5054.
- *Address translation instructions*.
- *TLB maintenance instructions* on page D8-5201.

### D8.2.11 Address translation instructions

Address translation instructions return the result of translating an IA using a specified translation stage or regime.

An address translation instruction has all of the following properties:

- An IA is supplied as the argument to the instruction.
- The instruction encoding determines the translation stage and regime used by the translation.
- The PAR_EL1 register is updated with the translation result.
- If an instruction is executed in Non-secure state, the instruction cannot return the result of a Secure address translation stage.
- If an address translation stage is controlled by a higher Exception level than the Exception level at which the address translation instruction is executed, the instruction is UNDEFINED.

If FEAT_MTE2 is enabled, then the results of AT* instructions reflect whether the translation is tagged or untagged. For more information, see *Tagged and Untagged Addresses* on page D9-5224.

The A64 assembly language syntax of an address translation instruction is **AT <operation>, <Xt>**.

- The <operation> in **AT <operation>, <Xt>** is one of the following address translation stages:
  - S1 specifies a stage 1 translation.
  - S12 specifies a stage 1 translation followed by a stage 2 translation.

- The <el> is one of the following Exception levels that apply to the translation:
  - E0 specifies EL0.
  - E1 specifies EL1.
  - E2 specifies EL2.
  - E3 specifies EL3.

- The <read|write> is one of the following:
  - R specifies Read.
  - W specifies Write.
• If FEAT_PAN2 is implemented, then the <pan> component is included and has all of the following properties:
  — P determines whether the PSTATE.PAN value is considered when determining permissions. For more information, see PSTATE.PAN on page D8-5132.
  — The field is optional.
  — If <stages>=S1 and <el>=E1, the field is permitted.
  — If <stages>!=S1 or <el>!=E1, the field is not permitted.

If <el> is higher than the current Exception level, then the address translation instruction is UNDEFINED.

The <Xt> in AT <operation>, <Xt> specifies the 64-bit register holding the address to be translated.

When an address translation instruction applies to a translation regime that is using AArch32, VA[63:32] is RES0.

When an address is translated by an address translation instruction, no alignment restrictions exist and therefore an address translation instruction cannot generate an Alignment fault.

For more information, see MMU faults generated by address translation instructions on page D8-5183.

When an address translation instruction is executed, the specified translation stage and regime determines all of the following:
• The entries in TLB caching structures that are permitted to be used.
• How the translation table walk is done.

When an address translation instruction is executed, it is IMPLEMENTATION DEFINED whether the result can be returned from a TLB, or a translation table walk occurs.

If TLB entries might differ from the underlying translation tables held in memory, such as when waiting for a maintenance or synchronization event to complete after updating the translation tables, Arm recommends not using the address translation instructions.

Address translation instructions, successful address translation

When an address translation instruction successfully translates an address, all of the following are updated in PAR_EL1:
• PAR_EL1.F is set to 0.
• The resulting OA is returned in PAR_EL1.PA, and the resulting attributes are returned in the other fields of PAR_EL1.

The architecture defines a single PAR, PAR_EL1, that is used regardless of all of the following:
• The Exception level at which the address translation instruction was executed.
• The Exception level that controls the address translation stage or stages used by the address translation instruction.

Address translation instructions, effect of translation regime

If EL2 is disabled or not implemented, the AT S1E2R and AT S1E2W instructions are UNDEFINED.

If EL3 is implemented and EL2 is disabled or not implemented for the target Security state, then all of the following apply to the AT S1E++ instruction behavior:
• The instruction has the same behavior as the equivalent AT S1E++ instruction.
• The instruction behaves consistently with an implementation that has all of the following characteristics:
  — EL2 is implemented.
  — Stage 2 translation is disabled.

RPXNSR

If SCR_EL3.NS is 0 and Secure EL2 is disabled or not implemented, then executing the following AT instructions at EL3 are is UNDEFINED:
- AT S1E2R.
- AT S1E2W.

IKJDTT

The value of HCR_EL2.DC affects all of the following instructions:
- AT S1E0*.
- AT S1E1*.
- AT S12E0*.
- AT S12E1*.

Address translation instructions, synchronization requirements

IPTYVP

When an address translation instruction is executed, explicit synchronization is required to guarantee the result is visible to subsequent direct reads of PAR_EL1.

For more information, see Synchronization requirements for AArch64 System registers on page D17-5547.

IPCPWC

Requiring explicit synchronization after executing an address translation instruction is consistent with the general requirement that the effect of a write to a System register requires explicit synchronization before the result is guaranteed to be visible to subsequent instructions.
D8.3 Translation table descriptor formats

If an address translation stage is controlled by an Exception level that is using AArch64, then the translation table uses the VMASv8-64 format.

A translation descriptor has one of the following formats:

- An invalid descriptor format.
- A Table descriptor format that points to the next-level translation table.
- A Block descriptor or Page descriptor format that defines the memory access properties.
- A reserved format.

The value of descriptor bit[0] determines one of the following:

- If descriptor bit[0] is 0, then the descriptor is invalid.
- If descriptor bit[0] is 1, then the descriptor is valid.

If descriptor bit[0] is 1, then the value of descriptor bit[1] determines one of the following descriptor types:

- For lookup levels other than lookup level 3, one of the following:
  - If bit[1] is 0, then the descriptor is a Block descriptor.
  - If bit[1] is 1, then the descriptor is a Table descriptor.
- For lookup level 3, one of the following:
  - If bit[1] is 0, then the descriptor is reserved, and treated as invalid.
  - If bit[1] is 1, then the descriptor is a Page descriptor.

D8.3.1 Table descriptor format

Throughout this section, if an address translation stage is not specified, then references to the Effective value of TCR_ELx.DS also apply to VTCR_EL2.DS.

The following figure shows all of following stage 1 and stage 2 Table descriptor formats:

- 4KB, 16KB, and 64KB granules using a 48-bit next-level table address.
- If the Effective value of TCR_ELx.DS is 1, the 4KB and 16KB granules using a 52-bit next-level table address.
- If FEAT_LPA is implemented, the 64KB granule using a 52-bit next-level table address.
The AArch64 Virtual Memory System Architecture
D8.3 Translation table descriptor formats

4KB and 16KB granules, 52-bit OA

| Table | Attributes | RES0 | Next-level table address[51:50] | m-1 | 12 | 11 | 9 | 8 | 7 | 2 | 1 | 0 |
|-------|------------|------|----------------------------------|-----|----|----|---|---|---|---|---|---|---|
| 63    | 59 58 51 50 49 |      |                                  |     |    |    |   |   |   |   |   |   |   |

With the 4KB granule size m is 12 and with the 16KB granule size m is 14.
† When m is 12, the RES0 field shown for bits[(m-1):12] is absent.

64KB granule, 52-bit OA

| Table | Attributes | RES0 | Next-level table address[51:48] | m-1 | 12 | 11 | 9 | 8 | 7 | 2 | 1 | 0 |
|-------|------------|------|----------------------------------|-----|----|----|---|---|---|---|---|---|---|
| 63    | 59 58 51 50 48 47 |      |                                  |     |    |    |   |   |   |   |   |   |   |

† TA[51:48] indicates bits [51:48] of the next-level table address.

4KB, 16KB, and 64KB granules, 48-bit OA

| Table | Attributes | RES0 | Next-level table address[47:48] | m-1 | 12 | 11 | 9 | 8 | 7 | 2 | 1 | 0 |
|-------|------------|------|----------------------------------|-----|----|----|---|---|---|---|---|---|---|
| 63    | 59 58 51 50 47 |      |                                  |     |    |    |   |   |   |   |   |   |   |

With the 4KB granule size m is 12, with the 16KB granule size m is 14, and with the 64KB granule size m is 16.
† When m is 12, the RES0 field shown for bits[(m-1):12] is absent.

**Figure D8-12 Table descriptor formats**

For a stage 1 translation, the following figure shows the next-level attribute fields in a Table descriptor. For a stage 2 translation, bits[63:59] are RES0 and bits[58:51] are IGNORED.

<table>
<thead>
<tr>
<th>63 62 61 60 59 58 51</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTable</td>
</tr>
<tr>
<td>APTable</td>
</tr>
<tr>
<td>UXNTable or XNTable</td>
</tr>
<tr>
<td>PXNTable ‡</td>
</tr>
</tbody>
</table>

‡ UXNTable for a translation regime that supports two VA ranges, XNTable for other regimes.
† Regimes that support two VA ranges only, RES0 in the other regimes.

**Figure D8-13 Stage 1 next-level attribute fields in a Table descriptor**

Table descriptor bit[63] is one of the following:

- For stage 1 translations in Secure state, the NTable field which determines the IPA or PA space used for translation tables in subsequent lookup levels. For more information, see Hierarchical control of Secure or Non-secure memory accesses on page D8-5135.
- For stage 1 translations in Non-secure state, this bit is RES0.
- For stage 2 translations, this bit is RES0.

Table descriptor bits[62:61] are one of the following:

- For stage 1 translations, the APTable[1:0] field which determines the access permissions limit for subsequent lookup levels.
- For stage 2 translations, these bits are RES0.

For more information, see Hierarchical control of data access permissions on page D8-5138.

For EL1&0 stage 1 translations, if the Effective value of HCR_EL2.{NV, NV1} is \{1, 1\}, then APTable[0] is treated as 0 regardless of the actual value.
Table descriptor bit[60] is one of the following:

- For stage 1 translations that support one privilege level, the XNTable field which determines the execute-never limit for subsequent lookup levels.
- For stage 1 translations that support two privilege levels, the UXNTable field which determines the unprivileged execute-never limit for subsequent lookup levels at EL0.
- For EL1&0 stage 1 translations, if the Effective value of HCR_EL2.{NV, NV1} is {1, 1}, then the PXNTable field. For more information, see Additional behavior when HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1 on page D8-5174.
- For stage 2 translations, this bit is RES0.

For more information, see Additional behavior when HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1 on page D8-5174.

Table descriptor bit[59] is one of the following:

- For stage 1 translations that support one privilege level, RES0.
- For stage 1 translations that support two privilege levels, the PXNTable field which determines the privileged execute-never limit for subsequent lookup levels.
- For EL1&0 stage 1 translations, if the Effective value of HCR_EL2.{NV, NV1} is {1, 1}, then RES0. For more information, see Additional behavior when HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1 on page D8-5174.
- For stage 2 translations, this bit is RES0.

For more information, see Hierarchical control of instruction execution permissions on page D8-5141.

Table descriptor bits[58:51] are IGNORED.

Table descriptor bit[50] is RES0.

Table descriptor bits[49:48] are one of the following:

- For the 4KB and 16KB granules, if the Effective value of TCR_ELx.DS is 0, then RES0.
- For the 4KB and 16KB granules, if the Effective value of TCR_ELx.DS is 1, then bits[49:48] of the next-level table address.
- For the 64KB translation granule, RES0.

Table descriptor bits[47:12] are one of the following:

- For the 4KB granule, bits[47:12] of the next-level table address.
- For the 16KB granule, all of the following:
  - Table descriptor bits[47:14] are bits[47:14] of the next-level table address.
  - Table descriptor bits[13:12] are RES0.
- For the 64KB granule, all of the following:
  - Table descriptor bits[47:16] are bits[47:16] of the next-level table address.
  - If FEAT_LPA is not implemented, then Table descriptor bits[15:12] are RES0.
  - If FEAT_LPA is implemented, then Table descriptor bits[15:12] are bits[51:48] of the next-level table address.

Table descriptor bits[11:2] are one of the following:

- For the 4KB and 16KB granules, if the Effective value of TCR_ELx.DS is 0, then the bits are IGNORED.
- For the 4KB and 16KB granules, if the Effective value of TCR_ELx.DS is 1, then all of the following:
  - Table descriptor bits[11:10] are IGNORED.
  - Table descriptor bits[9:8] are bits[51:50] of the next-level table address.
  - Table descriptor bits[7:2] are IGNORED.
- For the 64KB granule, the bits are IGNORED.
For lookup levels other than lookup level 3, to specify a Table descriptor, the value of descriptor bit[1] is required to be 1.

Table descriptor bit[0] is required to be 1 to indicate the descriptor is valid.

The next-level table address is one of the following:
- For a level -1 Table descriptor, the address of a level 0 table.
- For a level 0 Table descriptor, the address of a level 1 table.
- For a level 1 Table descriptor, the address of a level 2 table.
- For a level 2 Table descriptor, the address of a level 3 table.

If EL2 is enabled, the next-level table address is one of the following:
- For translation regimes other than Secure EL1&0 and Non-secure EL1&0, the PA of the target table.
- For the Secure EL1&0 and Non-secure EL1&0 translation regimes, the IPA of the target table.

### D8.3.2 Block descriptor and Page descriptor formats

Throughout this section, if an address translation stage is not specified, then references to the Effective value of TCR_ELx.DS also apply to VTCR_EL2.DS.

The following figure shows all of the following Block descriptor formats:
- 4KB, 16KB, and 64KB granules using a 48-bit OA.
- If the Effective value of TCR_ELx.DS is 1, the 4KB and 16KB granules using a 52-bit OA.
- If FEAT_LPA is supported, the 64KB granule using a 52-bit OA.

For the 4KB granule size, the level 1 descriptor \( n \) is 30, and the level 2 descriptor \( n \) is 21.
For the 16KB granule size, the level 2 descriptor \( n \) is 25.
For the 64KB granule size, the level 2 descriptor \( n \) is 29.

The following figure shows all of the following Page descriptor formats:
- 4KB, 16KB, and 64KB granules using a 48-bit OA.
- If the Effective value of TCR_ELx.DS is 1, the 4KB and 16KB granules using a 52-bit OA.
- If FEAT_LPA is supported, the 64KB granule using a 52-bit OA.

---

**Figure D8-14 Block descriptor formats**
### Figure D8-15 Page descriptor formats

For a stage 1 translation, the following figure shows the attribute fields in Block descriptors and Page descriptors, split into upper attributes and lower attributes.

#### 4KB granule 52-bit OA

<table>
<thead>
<tr>
<th>Page</th>
<th>Upper attributes</th>
<th>Output address[49:12]</th>
<th>Lower attributes</th>
<th>51 50 49</th>
<th>12 11 10 9 8 7 2 1 0</th>
</tr>
</thead>
</table>

#### 4KB granule 48-bit OA

<table>
<thead>
<tr>
<th>Page</th>
<th>Upper attributes</th>
<th>RES0</th>
<th>Output address[47:12]</th>
<th>Lower attributes</th>
<th>1 1</th>
</tr>
</thead>
</table>

#### 16KB granule 52-bit OA

<table>
<thead>
<tr>
<th>Page</th>
<th>Upper attributes</th>
<th>Output address[49:14]</th>
<th>RES0</th>
<th>Lower attributes</th>
<th>51 50 49</th>
<th>14 13 12 11 10 9 8 7 2 1 0</th>
</tr>
</thead>
</table>

#### 16KB granule 48-bit OA

<table>
<thead>
<tr>
<th>Page</th>
<th>Upper attributes</th>
<th>RES0</th>
<th>Output address[47:14]</th>
<th>RES0</th>
<th>Lower attributes</th>
<th>1 1</th>
</tr>
</thead>
</table>

#### 64KB granule 52-bit OA

<table>
<thead>
<tr>
<th>Page</th>
<th>Upper attributes</th>
<th>RES0</th>
<th>Output address[47:16]</th>
<th>OA[51:48]</th>
<th>Lower attributes</th>
<th>1 1</th>
</tr>
</thead>
</table>

#### 64KB granule 48-bit OA

<table>
<thead>
<tr>
<th>Page</th>
<th>Upper attributes</th>
<th>RES0</th>
<th>Output address[47:16]</th>
<th>RES0</th>
<th>Lower attributes</th>
<th>1 1</th>
</tr>
</thead>
</table>

---

**Figure D8-16 Stage 1 attribute fields in Block and Page descriptors**

- **Upper attributes**
  - 63 62 59 58 55 54 53 52 51 50
  - PBHA
  - IGNORED
  - Reserved for software use
  - UXN or XN
  - PXN
  - Contiguous
  - DBM

- **Lower attributes**
  - 16 15 12 11 10 9 8 7 6 5 4 2
  - nT
  - OA
  - nG
  - AF
  - SH[1:0]
  - AP[2:1]
  - NS
  - AttrIndx[2:0]

- **Notes:**
  - † IGNORED if FEAT_HFDS2 is not implemented.
  - †† UXN for a translation regime that can apply to execution at EL0, otherwise XN.
  - †‡ RES0 for a translation regime that cannot apply to execution at EL0.
  - * RES0 if FEAT_HAFDBS is not implemented.
  - ‡ RES0 if FEAT_LPA is not implemented.
  - # OA[51:50] if the Effective value of TCR_Elx.DS is 1.
For a stage 2 translation, the following figure shows the attribute fields in Block descriptors and Page descriptors, split into upper attributes and lower attributes.

![Figure D8-17 Stage 2 attribute fields in Block and Page descriptors](image)

- **Upper attributes**:
  - Bits [63:51] are reserved for use by a System MMU.
  - Bits [50:22] are reserved for software use.
  - Bit [21] is `CONTIGUOUS`.
  - Bit [20] is `DBM`.
  - Bits [19:16] are reserved for System MMU.

- **Lower attributes**:
  - Bits [11:2] are `OA[51:50]` if the Effective value of VTCR_EL2.DS is 1.
  - Bit [1] is `X[1:0]`.
  - Bit [0] is `AF`.

**Notes**:
- Bit [53] is `RES0` if FEAT_XNX is not implemented.
- Bits [62:60] are reserved for use by System MMU if FEAT_HPDS2 is not implemented.
- Bit [59] is `IGNORED` if FEAT_HPDS2 is not implemented.
- *RES0* if FEAT_HAFDBS is not implemented.
- §RES0 if FEAT_LPA is not implemented.
- #OA[51:50] if the Effective value of VTCR_EL2.DS is 1.

---

**Table 8-5128**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td><code>PBHA</code></td>
</tr>
<tr>
<td>62</td>
<td><code>IGNORED</code></td>
</tr>
<tr>
<td>60-58</td>
<td>Reserved for use by System MMU</td>
</tr>
<tr>
<td>57</td>
<td><code>XN[1:0]</code></td>
</tr>
<tr>
<td>56</td>
<td><code>Contiguous</code></td>
</tr>
<tr>
<td>55-51</td>
<td>Reserved for software use</td>
</tr>
<tr>
<td>16</td>
<td><code>hT</code></td>
</tr>
<tr>
<td>15-12</td>
<td>Reserved for use by System MMU</td>
</tr>
<tr>
<td>11-10</td>
<td><code>FnXS</code></td>
</tr>
<tr>
<td>9-8</td>
<td><code>AF</code></td>
</tr>
<tr>
<td>7-6</td>
<td><code>SH[1:0]</code></td>
</tr>
<tr>
<td>5</td>
<td><code>S2AP[1:0]</code></td>
</tr>
<tr>
<td>4-2</td>
<td><code>MemAttr[3:0]</code></td>
</tr>
<tr>
<td>1-0</td>
<td><code>OA[51:50]</code></td>
</tr>
</tbody>
</table>

---

**Notes**:
- Bit [53] is `RES0` if FEAT_XNX is not implemented.
- Bits [62:60] are reserved for use by System MMU if FEAT_HPDS2 is not implemented.
- Bit [59] is `IGNORED` if FEAT_HPDS2 is not implemented.
- *RES0* if FEAT_HAFDBS is not implemented.
- §RES0 if FEAT_LPA is not implemented.
- #OA[51:50] if the Effective value of VTCR_EL2.DS is 1.
• For stage 1 translations, bit[53] is one of the following:
  — If the translation regime supports only one privilege level, then RES0.
  — If the translation regime can support two privilege levels, then the Privileged execute-never field (PXN).
  — For EL1&0 translations, if the Effective value of HCR_EL2.NV, NV1 is \{1, 1\}, then RES0.
• For stage 2 translations, bits[54:53] are one of the following:
  — If FEAT_XNX is not implemented, then bit[54] is the Execute-never field (XN) and bit[53] is RES0.
  — If FEAT_XNX is implemented, then bits[54:53] are the Execute-never field (XN[1:0]).
For more information, see Instruction execution permissions on page D8-5138 and Additional behavior when HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1 on page D8-5174.

RKMTMJ
Block descriptor and Page descriptor bit[52] is the Contiguous bit.
For more information, see The Contiguous bit on page D8-5158.

RNJYJS
Block descriptor and Page descriptor bit[51] is the Dirty Bit Modifier (DBM).
For more information, see Hardware management of the dirty state on page D8-5145.

RKHPCD
Block descriptor and Page descriptor bit[50] is one of the following:
- If FEAT_BTI is not implemented, then Block descriptor and Page descriptor bit[50] is RES0.
- If FEAT_BTI is implemented, then one of the following:
  — For stage 1 translations, the Block descriptor and Page descriptor bit[50] is Guarded Page field (GP).
  — For stage 2 translations, the Block descriptor and Page descriptor bit[50] is RES0.

RCYLDW
Block descriptor and Page descriptor bits[49:48] are one of the following:
- For the 4KB and 16KB granules, one of the following:
  — If the Effective value of TCR_ELx.DS is 0, then RES0.
  — If the Effective value of TCR_ELx.DS is 1, then bits[49:48] of the OA.
- For the 64KB granule, RES0.

RBHRKX
Block descriptor bits[47:17] are one of the following:
- For the 4KB granule, if the Effective value of TCR_ELx.DS is 1, then all of the following in a level 0 Block descriptor:
  — Block descriptor bits[47:39] are bits[47:39] of the OA.
  — Block descriptor bits[38:17] are RES0.
- For the 4KB granule, all of the following in a level 1 Block descriptor:
  — Block descriptor bits[47:30] are bits[47:30] of the OA.
  — Block descriptor bits[29:17] are RES0.
- For the 4KB granule, all of the following in a level 2 Block descriptor:
  — Block descriptor bits[47:21] are bits[47:21] of the OA.
  — Block descriptor bits[20:17] are RES0.
- For the 16KB granule, if the Effective value of TCR_ELx.DS is 1, then all of the following in a level 1 Block descriptor:
  — Block descriptor bits[47:36] are bits[47:36] of the OA.
  — Block descriptor bits[35:17] are RES0.
- For the 16KB granule, all of the following in a level 2 Block descriptor:
  — Block descriptor bits[47:25] are bits[47:25] of the OA.
  — Block descriptor bits[24:17] are RES0.
• For the 64KB granule, if FEAT_LPA is implemented, then all of the following in a level 1 Block descriptor:
  — Block descriptor bits[47:42] are bits[47:42] of the OA.
  — Block descriptor bits[41:17] are RES0.
• For the 64KB granule, all of the following in a level 2 Block descriptor:
  — Block descriptor bits[47:29] are bits[47:29] of the OA.
  — Block descriptor bits[28:17] are RES0.

BBHXV
Block descriptor bit[16] is one of the following:
• If FEAT_BBM is not implemented, then Block descriptor bit[16] is RES0.
• If FEAT_BBM is not implemented, then Block descriptor bit[16] is the nT field.
For more information, see Block translation entry on page D8-5160.

VZRKS
Block descriptor bits[15:12] are one of the following:
• If FEAT_LPA is not implemented, then Block descriptor bits[15:12] are RES0.
• If FEAT_LPA is implemented, then one of the following:
  — For the 4KB and 16KB granules, then Block descriptor bits[15:12] are RES0.
  — For the 64KB granule, then Block descriptor bits[15:12] are bits[51:48] of the OA.

QNVFN
Page descriptor bits[47:12] are one of the following:
• For the 4KB granule, bits[47:12] of the OA.
• For the 16KB granule, all of the following:
  — Page descriptor bits[47:14] are bits[47:14] of the OA.
  — Page descriptor bits[13:12] are RES0.
• For the 64KB granule, all of the following:
  — Page descriptor bits[47:16] are bits[47:16] of the OA.
  — If FEAT_LPA is not implemented, then Page descriptor bits[15:12] are RES0.
  — If FEAT_LPA is implemented, then Page descriptor bits[15:12] are bits[51:48] of the OA.

DFNPW
Block descriptor and Page descriptor bit[11] is one of the following:
• For stage 1 translations that support a single privilege level, Block descriptor and Page descriptor bit[11] is RES0.
• For stage 1 translations that support two privilege levels, Block descriptor and Page descriptor bit[11] is the not global bit (nG). For more information, see Global and process-specific translation table entries on page D8-5194.
• For stage 2 translations, one of the following:
  — If FEAT_XS is not implemented, then Block descriptor and Page descriptor bit[11] is RES0.
  — If FEAT_XS is implemented, then Block descriptor and Page descriptor bit[11] is FnXS. For more information, see XS attribute modifier on page D8-5160.

XBPJZ
Block descriptor and Page descriptor bit[10] is the Access flag (AF).
For more information, see The Access flag on page D8-5143.

MIZBIW
Block descriptor and Page descriptor bits[9:8] are one of the following:
• For the 4KB and 16KB granules, one of the following:
  — If the Effective value of TCR_ELx.DS is 0, then the Shareability field (SH[1:0]).
  — If the Effective value of TCR_ELx.DS is 1, then bits[51:50] of the OA.
• For the 64KB translation granule, the Shareability field (SH[1:0]).
For more information, see Stage 1 Shareability attributes on page D8-5152 and Stage 2 Shareability attributes on page D8-5156.
Block descriptor and Page descriptor bits[7:6] are one of the following:

- For stage 1 translations that support a single privilege level, all of the following:
  - Bit[7] is the data Access Permissions bit (AP[2]).
- For EL1 stage 1 translations, if the Effective value of HCR_EL2.{NV, NV1} is {1, 1}, then all of the following:
  - Bit[7] is the data Access Permissions bit (AP[2]).
  - Bit[6] is treated as 0 regardless of the actual value.
- For stage 1 translations that support two privilege levels, the data Access Permissions bits (AP[2:1]).
- For stage 2 translations, the stage 2 data Access Permissions field (S2AP[1:0]).

For more information, see Data access permissions for stage 1 translations on page D8-5136, Data access permissions for stage 2 translations on page D8-5137, and Additional behavior when HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1 on page D8-5174.

The Armv8 Block descriptor and Page descriptor format defines the data Access Permissions bits, AP[2:1], and does not define an AP[0] bit.

Block descriptor and Page descriptor bits[5:2] are one of the following:

- For stage 1 translations bit[5] is one of the following:
  - When the access is from Secure state, the Non-secure bit (NS). For more information, see Control of Secure or Non-secure memory access on page D8-5135.
  - When the access is from Non-secure state, the bit is res0.
- For stage 1 translations, bits[4:2] are the stage 1 memory attributes index field for the MAIR_ELx (AttrIndx[2:0]). For more information, see Stage 1 memory type and Cacheability attributes on page D8-5151.
- For stage 2 translations, if the Effective value of HCR_EL2.FWB is 0, then bits[5:2] are the stage 2 memory attributes (MemAttr[3:0]). For more information, see Stage 2 memory type and Cacheability attributes when FEAT_S2FWB is disabled on page D8-5153.
- For stage 2 translations, if the Effective value of HCR_EL2.FWB is 1, then all of the following:
  - Bit[5] is res0.
  - Bits[4:2] are the stage 2 memory attributes (MemAttr[2:0]). For more information, see Stage 2 memory type and Cacheability attributes when FEAT_S2FWB is enabled on page D8-5155.

Descriptor bit[1] is one of the following:

- For lookup levels other than lookup level 3, to specify a Block descriptor, the value of descriptor bit[1] is required to be 0.
- For lookup level 3, to specify a Page descriptor, the value of descriptor bit[1] is required to be 1.

Block descriptor and Page descriptor bit[0] is required to be 1 to indicate the descriptor is valid.

For more information, see:

- Effect of PSTATE on access permission on page D8-5132.
- Hardware updates to the translation tables on page D8-5148.
D8.4 Memory access control

Fields in the descriptors and in PSTATE provide ways to control access to memory and instruction execution.

D8.4.1 Effect of PSTATE on access permission

PSTATE.PAN affects memory access permissions decoded from memory access control fields in the descriptors. PSTATE.UAO affects the privilege level of a memory access. The PSTATE.BTYPE field is part of the determination of whether a branch access to a guarded memory region generates a Branch Target exception.

For more information on PSTATE, see Process state, PSTATE on page D1-4670.

PSTATE.PAN

PSTATE.PAN is used to prevent privileged data accesses to virtual memory addresses that are accessible from EL0.

All statements in this section require implementation of FEAT_PAN.

For a translation regime, if one or more of the following apply, then the PSTATE.PAN bit has no effect in that regime:

- Stage 1 translation is disabled.
- Stage 1 translation supports a single privilege level.

If the Effective value of PSTATE.PAN is 1, then a privileged data access from any of the following Exception levels to a virtual memory address that is accessible to data accesses at EL0 generates a stage 1 Permission fault:

- A privileged data access from EL1.
- If HCR_EL2.E2H is 1, then a privileged data access from EL2.

If the value of PSTATE.PAN is 0, then a privileged data access is permitted from the corresponding privileged Exception level to a virtual memory address that is accessible at EL0.

When taking an exception from an EL in AArch64 state to an ELx, PSTATE.PAN is saved to and restored from SPSR_ELx.PAN.

When taking an exception from an EL in AArch32 state to an ELx in AArch64 state, CPSR.PAN is saved to and restored from SPSR_ELx.PAN.

When entering and exiting Debug state, PSTATE.PAN is saved to and restored from DSPSR_EL0.PAN.

When an exception occurs, one of the following determines whether PSTATE.PAN is set to 1 or the value of PSTATE.PAN is unchanged:

- When taking an exception to EL1, SCTLR_EL1.SPAN.
- When taking an exception to EL2, SCTLR_EL2.SPAN.

If FEAT_PAN3 is implemented, PSTATE.PAN is 1, and unprivileged instruction execution is permitted to a VA, then all of the following prevent privileged data accesses to that VA:

- For a data access from EL1, when SCTLR_EL1.EPAN is 1, a Permission fault is generated.
- For a data access from EL2, when HCR_EL2.E2H is 1 and SCTLR_EL2.EPAN is 1, a Permission fault is generated.

For the purpose of PAN, if FEAT_PAN3 is implemented, then it is IMPLEMENTATION DEFINED whether SCR_EL3.SIF is used to determine instruction execution permission.

If FEAT_PAN3 is implemented, then the SCTLR_EL1.EPAN and SCTLR_EL2.EPAN bits affect the AT S1E1RP and AT S1E1WP instructions, consistent with those instructions using PSTATE.PAN to determine whether the memory access generates a Permission fault.

The PSTATE.PAN bit has no effect on all of the following:

- Instruction fetches.
• Data cache instructions, except DC ZVA.
• If FEAT_PAN2 is not implemented, then address translation instructions.
• If FEAT_PAN2 is implemented, then the address translation instructions other than AT S1E1RP and AT S1E1WP.

If the current Exception level is EL1 and HCR_EL2.{NV, NV1} is \{1, 1\}, then PSTATE.PAN is treated as 0 for all purposes except reading the bit value.

Software can access PSTATE.PAN using all of the following instructions:
• Software can use MSR (immediate) PAN, #Imm4 to modify PSTATE.PAN.
• Software can use MSR (register) PAN, Xt to modify PSTATE.PAN.
• Software can use MRS Xt, PAN to read PSTATE.PAN.

If the PE is in Debug state, then a debugger can use the DRPS instruction to modify PSTATE.PAN.

PSTATE.UAO

PSTATE.UAO is used to cause unprivileged load/store register instructions to behave as if they are load/store register instructions.

All statements in this section require implementation of FEAT_UAO.

If the value of PSTATE.UAO is 1, then a load/store unprivileged instruction executed at any of the following Exception levels is subject to the same access permissions as the corresponding load/store register instruction:
• EL1.
• If the Effective value of HCR_EL2.{E2H, TGE} is \{1, 1\}, EL2.

If the value of PSTATE.UAO is 0, then load/store unprivileged instructions are not affected.

When an exception occurs, all of the following apply to PSTATE.UAO:
• When taking an exception to ELx, PSTATE.UAO is saved to SPSR_ELx.UAO and then set to 0.
• When returning from an exception at ELx, PSTATE.UAO is restored from SPSR_ELx.UAO.

If the PE is in AArch32 state, then when an exception occurs and the exception is taken to AArch64 state, all of the following occur:
• PSTATE.UAO is set to 0.
• The corresponding SPSR_ELx.UAO is set to 0.

When entering and exiting Debug state, PSTATE.UAO is saved to and restored from DSPSR_EL0.UAO.

Software can access PSTATE.UAO using all of the following instructions:
• Software can use MSR (immediate) UAO, #Imm4 to modify PSTATE.UAO.
• Software can use MSR (register) UAO, Xt to modify PSTATE.UAO.
• Software can use MRS Xt, UAO to read PSTATE.UAO.

If the PE is in Debug state, a debugger can use the DRPS instruction to modify PSTATE.UAO.

PSTATE.BTYPE

PSTATE.BTYPE is part of a mechanism used to guard memory pages against executing instructions that are not the intended target of a branch.

All statements in this section require implementation of FEAT_BTI.

When an instruction completes execution, the value of the PSTATE.BTYPE field is determined by all of the following, as shown in the following table:
• The instruction that was executed.
• Whether or not the memory region is guarded.
• The register accessed by the instruction.

### Table D8-36 Values taken by \texttt{PSTATE.BTYPE} on execution of an instruction

<table>
<thead>
<tr>
<th>Instruction executed</th>
<th>Memory region</th>
<th>Register accessed</th>
<th>PSTATE.BTYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR, BRAA, BRAAZ, BRAB, BRABZ</td>
<td>Guarded</td>
<td>Any register except X16 or X17</td>
<td>0b11</td>
</tr>
<tr>
<td>BLR, BLRAA, BLRAAZ, BLRAB, BLRABZ</td>
<td>Any</td>
<td>Any register</td>
<td>0b10</td>
</tr>
<tr>
<td>BR, BRAA, BRAAZ, BRAB, BRABZ</td>
<td>Guarded</td>
<td>X16 or X17</td>
<td>0b01</td>
</tr>
<tr>
<td>BR, BRAA, BRAAZ, BRAB, BRABZ</td>
<td>Non-guarded</td>
<td>Any register</td>
<td>0b01</td>
</tr>
<tr>
<td>Any instruction except BR, BRAA, BRAAZ, BRAB, BRABZ, BLR, BLRAA, BLRAAZ, BLRAB, BLRABZ</td>
<td>Any</td>
<td>Any register</td>
<td>0b00</td>
</tr>
</tbody>
</table>

When the BTI instruction is executed, the \texttt{<targets>} operand determines the compatibility between the BTI instruction and different \texttt{PSTATE.BTYPE} values as shown in the following table:

### Table D8-37 Compatibility of BTI instruction to different \texttt{PSTATE.BTYPE} values

<table>
<thead>
<tr>
<th>\texttt{&lt;targets&gt;}</th>
<th>\texttt{PSTATE.BTYPE} value</th>
<th>0b00</th>
<th>0b01</th>
<th>0b10</th>
<th>0b11</th>
</tr>
</thead>
<tbody>
<tr>
<td>(omitted)</td>
<td>N/A</td>
<td>Not compatible</td>
<td>Not compatible</td>
<td>Not compatible</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>N/A</td>
<td>Compatible</td>
<td>Compatible</td>
<td>Not compatible</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>N/A</td>
<td>Compatible</td>
<td>Not compatible</td>
<td>Compatible</td>
<td></td>
</tr>
<tr>
<td>jc</td>
<td>N/A</td>
<td>Compatible</td>
<td>Compatible</td>
<td>Compatible</td>
<td></td>
</tr>
</tbody>
</table>

If stage 1 translation is enabled, the GP bit in Block and Page descriptors indicates whether the memory region is guarded. For more information, see \texttt{FEAT_BTI}.

If stage 1 translation is disabled, no memory regions are guarded. For more information, see \textit{Behavior when stage 1 address translation is disabled} on page D8-5118.

If the \texttt{PSTATE.BTYPE} field is not 0b00 and an attempt is made to execute an instruction within a guarded page, a Branch Target exception is generated unless the instruction is one of the following:

- A BTI instruction that is compatible with the \texttt{PSTATE.BTYPE} field.
- A PACIASP or PACIBSP instruction, and the \texttt{PSTATE.BTYPE} is consistent with implicit BTI behavior of these instructions.
- A Breakpoint Instruction exception.
- A Halt Instruction debug event.

When a Branch Target exception occurs, it is taken to one of the following:

- When executing at EL0 and HCR_EL2.TGE is 0, the Branch Target exception is taken to EL1.
- When executing at EL0 and HCR_EL2.TGE is 1, the Branch Target exception is taken to EL2.
- When executing at ELx, where x is 1, 2, or 3, the Branch Target exception is taken to ELx.

When a Branch Target exception occurs, ESR_ELx.EC is 0xD.

For more information, see \textit{ISS encoding for an exception from Branch Target Identification instruction} on page D17-5700.
When PACIASP and PACIBSP instructions access a guarded memory region, the instructions have an implicit branch target identification instruction that is compatible with one of the following:

- The PSTATE.BTYPE value is \(0b10\) or \(0b01\).  
- When the associated SCTLR_ELx.\{BT0, BT1, BT\} bits are 0, the PSTATE.BTYPE value is \(0b11\).

The implicit branch target identification property of PACIASP and PACIBSP is independent of the setting of the SCTLR_ELx.\{EnIA, EnIB\} bits.

The BTI instruction is a NOP in a non-guarded page.

There is no mechanism for direct reads or direct writes of the PSTATE.BTYPE field.

### D8.4.2 Control of Secure or Non-secure memory access

The Non-secure, NS, bit in Block descriptors and Page descriptors, and the Non-secure table, NSTable, bit in Table descriptors are used to provide control over accesses from Secure state to Secure and Non-secure memory.

In this section and subsections, IPA or PA space refers to the OA space of stage 1 translations.

For a Secure translation regime, if the translation table entry is in Secure IPA or PA space, then the Block descriptor and Page descriptor NS determines all of the following:

- If NS is 0, an access to the OA specified by the descriptor is to Secure IPA or PA space.
- If NS is 1, an access to the OA specified by the descriptor is to Non-secure IPA or PA space.

For a Non-secure translation regime, the NS and NSTable bits are RES0 and the OA or next-level table base address is in Non-secure IPA or PA space.

For a Secure translation regime, when the descriptor is fetched from Non-secure IPA or PA space, all of the following apply to the descriptor:

- The descriptor NS and NSTable bits exist but they are SBZ instead of RES0.
- The OA or next-level table base address specified by the descriptor is in Non-secure IPA or PA space.

Hierarchical control of Secure or Non-secure memory accesses

For a Secure translation regime, when a Table descriptor is accessed from a stage 1 translation table in Secure IPA or PA space, the NSTable field determines all of the following:

- If NSTable is 0, the next-level translation table address in the Table descriptor is in Secure IPA or PA space.
- If NSTable is 1, the next-level translation table address in the Table descriptor is in Non-secure IPA or PA space.

If the next-level translation table address in the Table descriptor is in Non-secure IPA or PA space, then the address specified by a descriptor in the next lookup-level translation table is in Non-secure IPA or PA space.

If stage 2 translation is enabled, the VSTCR_EL2.SA, VTCR_EL2.NSA, VSTCR_EL2.SW, and VTCR_EL2.NSW fields can map an IPA space to a PA space not matching the Security of the IPA space.

If all of the following apply, then a stage 1 translation is treated as non-global, meaning the Effective value of nG is 1, regardless of the actual value of the Block descriptor or Page descriptor nG bit:

- The stage 1 translation supports two privilege levels.
- The PE is in Secure state.
- NSTable is 1 at any level of the translation table walk.

For more information, see Global and process-specific translation table entries on page D8-5194.

The descriptor NSTable field affects all subsequent lookup levels and the translation IPA or PA space. When an NSTable field is changed, software is required to use a break-before-make sequence, including TLB maintenance for all lookup levels for the VA range translated by the descriptor.
For more information, see *TLB maintenance* on page D8-5197 and *Using break-before-make when updating translation table entries* on page D8-5198.

### D8.4.3 Data access permissions

The descriptors contain fields used to control data access permissions in stage 1 and stage 2 translations.

#### Data access permissions for stage 1 translations

For a stage 1 translation that supports two Exception levels, the AP[2:1] bits control all of the following stage 1 data access permissions:

- AP[2] selects between read-only and read/write access.
- AP[1] selects between Application level control at EL0 and the privileged Exception level control.

For a stage 1 translation that supports one Exception level, AP[1] is RES1.

For a stage 1 translation that supports two Exception levels, the AP[1] descriptor bit determines all of the following:

- If AP[1] is 0, data accesses are not permitted at EL0.
- If AP[1] is 1, data accesses at EL0 use the data access permission defined by AP[2].

For a stage 1 translation that supports one Exception level, the following table shows the possible data access permissions at the current Exception level.

<table>
<thead>
<tr>
<th></th>
<th>Access permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Read/write</td>
</tr>
<tr>
<td>1</td>
<td>Read-only</td>
</tr>
</tbody>
</table>

For a stage 1 translation that supports two Exception levels, the following table shows the possible data access permissions at EL0 and the corresponding higher Exception level.

<table>
<thead>
<tr>
<th></th>
<th>Access from higher Exception level</th>
<th>Access from EL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Read/write</td>
<td>None</td>
</tr>
<tr>
<td>01</td>
<td>Read/write</td>
<td>Read/write</td>
</tr>
<tr>
<td>10</td>
<td>Read-only</td>
<td>None</td>
</tr>
<tr>
<td>11</td>
<td>Read-only</td>
<td>Read-only</td>
</tr>
</tbody>
</table>

If a data access to memory is attempted that is not permitted by the value of one of the following, then in the absence of a higher-priority MMU fault, a stage 1 Permission fault is generated:

- For a stage 1 translation that supports one Exception level, the AP[2] bit.
- For a stage 1 translation that supports two Exception levels, the AP[2:1] bits.

For more information, see *MMU fault prioritization from a single address translation stage* on page D8-5190.

For the EL1&0 translation regime in the current Security state, if the Effective value of HCR_EL2.{NV, NV1} is \{1, 1\}, then AP[1] is treated as 0 regardless of the actual value.

For more information, see *Additional behavior when HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1* on page D8-5174.

If hardware management of dirty state is enabled at stage 1, then the AP[2] bit can be cleared by hardware in some situations.
For more information, see *Hardware management of the dirty state on page D8-5145.*

**Data access permissions for stage 2 translations**

For the EL1&0 translation regime, if stage 2 translation is enabled, then the S2AP descriptor field determines the stage 2 data access permissions.

The following table shows encoding of the S2AP descriptor field:

<table>
<thead>
<tr>
<th>S2AP[1:0]</th>
<th>Access permission from EL1&amp;0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No access</td>
</tr>
<tr>
<td>01</td>
<td>Read-only</td>
</tr>
<tr>
<td>10</td>
<td>Write-only</td>
</tr>
<tr>
<td>11</td>
<td>Read/Write</td>
</tr>
</tbody>
</table>

The S2AP data access permissions do not distinguish between EL1 accesses and EL0 accesses.

If an attempt is made to access memory that is not permitted by the value of the S2AP field, then a stage 2 Permission fault is generated.

If hardware management of dirty state is enabled at stage 2, then the S2AP[1] bit can be set to 1 by hardware in some situations.

The combination of both the stage 1 and stage 2 permissions determine the final data access permissions.

If EL2 is enabled, then all of the following apply to the EL1&0 translation regime:

- For enabled stage 1 translations, all of the following apply:
  - The stage 1 translation is configured and controlled from EL1.
  - Stage 1 MMU faults are taken to EL1.
- For enabled stage 2 translations, all of the following apply:
  - The stage 2 translation is configured and controlled from EL2.
  - The stage 2 data access permissions do not differentiate between accesses at EL1 or EL0.
  - Software executing at EL2 can assign a write-only permission to a memory region.
  - Stage 2 MMU faults are taken to EL2.

Stage 1 Permission faults are reported with a higher priority than stage 2 Permission faults.

For a virtualization implementation, a hypervisor can define the stage 2 translation permissions to be more restrictive than the stage 1 translation permissions defined by a Guest OS. The final access permissions are the more restrictive of the permissions defined by:

- The Guest OS, in the stage 1 translation tables.
- The hypervisor, in the stage 2 translation tables.

The effects of combining access permissions defined by the hypervisor are expected to be functionally transparent to the Guest OS.

For more information, see *MMU fault prioritization from a single address translation stage on page D8-5190.*
Hierarchical control of data access permissions

Translation table entries at a given lookup level can limit data access permissions at subsequent lookup levels.

For a stage 1 translation, the Table descriptor APTable[1:0] field limits the data access permission of subsequent stage 1 translation lookup levels, regardless of the permissions in subsequent lookup levels, as shown in the following table:

<table>
<thead>
<tr>
<th>APTable[1:0]</th>
<th>Effect at subsequent lookup levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No effect on permissions.</td>
</tr>
<tr>
<td>01</td>
<td>Unprivileged access not permitted.</td>
</tr>
<tr>
<td>10</td>
<td>Write access not permitted.</td>
</tr>
<tr>
<td>11</td>
<td>Write access not permitted.</td>
</tr>
</tbody>
</table>

Unprivileged read access not permitted.

For a Permission fault, the level of the Block descriptor or Page descriptor is reported regardless of whether the lack of permission was caused by configuration of the APTable or AP fields.

For translation regimes that support one Exception level, APTable[0] is RES0.

The APTable[1:0] settings are combined with the descriptor access permissions in subsequent lookup levels. They do not change the values entered in those descriptors, nor restrict what values can be entered.

For the translation regime controlled by a TCR_ELx, one or more of the following can be used to disable the Table descriptor APTable[1:0] field so that it is IGNORED by the PE and the behavior is as if the value is 0:

- If the Effective value of TCR_ELx.HPD{0} is 1, hierarchical data access permission control is disabled in the translation tables pointed to by TTBR0_ELx.
- If the Effective value of TCR_ELx.HPD1 is 1, hierarchical data access permission control is disabled in the translation tables pointed to by TTBR1_ELx.

The descriptor APTable field affects all subsequent lookup levels. When an APTable field is changed, software is required to use a break-before-make sequence, including TLB maintenance for all lookup levels for the VA range translated by the descriptor.

D8.4.4 Instruction execution permissions

The descriptors contain fields used to control instruction execution permissions in stage 1 and stage 2 translations.

If instruction execution is not permitted by an execute-never field in a translation table entry at any level and stage of translation, then the memory region is execute-never.

If a memory region is execute-never at the current Exception level, then speculative instruction fetch from that memory region is prohibited.

If a memory region is execute-never at the current Exception level, then when an attempt to execute from that memory region occurs, a Permission fault is generated at the translation stage that determines the region is execute-never.

If Device memory is not execute-never, then speculative instruction fetch from that memory region is permitted.

For all Exception levels from which a read-sensitive memory region can be accessed, to avoid the possibility of a speculative fetch affecting that memory region, software is expected to define that memory region as execute-never.

Except for the purposes of PSTATE.PAN, the descriptor execute-never fields do not prevent data accesses to the memory region translated by the descriptor.
Summary of instruction access and execution permissions for stage 1 translations

If a translation regime applies to EL0 and a higher Exception level, then the following table shows the possible stage 1 memory access permissions, using all of the following notations:

- **Executable.** Executing instructions in the memory location is not prevented.
- **Not Executable.** Executing instructions in the memory location is not permitted.
- **R.** Reading from the memory location is permitted.
- **W.** Writing to the memory location is permitted.
- **WXN** is in the SCTLR_ELx register at the higher Exception level to which the translation regime applies.
- **AP[2:1]** is the effective value of AP[2:1] after the effects of APTable are applied.
- If hardware update of dirty state is enabled, then AP[2] is considered to be 0 for the purposes of computing execute permission in some situations.

Table D8-42 Summary of possible memory access permissions for a stage 1 translation supporting two Exception levels

<table>
<thead>
<tr>
<th>UXN</th>
<th>PXN</th>
<th>AP[2:1]</th>
<th>WXN</th>
<th>Access from higher Exception level</th>
<th>Access from EL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>R, W, Executable</td>
<td>Executable</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>1</td>
<td>R, W, Not executable</td>
<td>Executable</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>R, W, Not executable</td>
<td>R, W, Executable</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>1</td>
<td>R, W, Not executable</td>
<td>R, W, Not executable</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>x</td>
<td>R, Executable</td>
<td>Executable</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>x</td>
<td>R, Executable</td>
<td>R, Executable</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>00</td>
<td>x</td>
<td>R, W, Not executable</td>
<td>Executable</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01</td>
<td>0</td>
<td>R, W, Not executable</td>
<td>R, W, Executable</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01</td>
<td>1</td>
<td>R, W, Not executable</td>
<td>R, W, Not executable</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>10</td>
<td>x</td>
<td>R, Not executable</td>
<td>Executable</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>x</td>
<td>R, Not executable</td>
<td>R, Executable</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>R, W, Executable</td>
<td>Not executable</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>1</td>
<td>R, W, Not executable</td>
<td>Not executable</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>x</td>
<td>R, W, Not executable</td>
<td>R, W, Not executable</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>x</td>
<td>R, Executable</td>
<td>Not executable</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>x</td>
<td>R, Executable</td>
<td>R, Not executable</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00</td>
<td>x</td>
<td>R, W, Not executable</td>
<td>Not executable</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01</td>
<td>x</td>
<td>R, W, Not executable</td>
<td>R, W, Not executable</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>x</td>
<td>R, Not executable</td>
<td>Not executable</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
<td>x</td>
<td>R, Not executable</td>
<td>R, Not executable</td>
</tr>
</tbody>
</table>

For a memory location, if a translation regime applies to a single Exception level, then the following table shows the stage 1 access permissions, using all of the following notations:

- **Executable.** Executing instructions in the memory location is permitted.
• Not Executable. Executing instructions in the memory location is not permitted.
• R. Reading from the memory location is permitted.
• W. Writing to the memory location is permitted.
• WXN is in the SCTLR_ELx register at the higher Exception level to which the translation regime applies.
• AP[2] is the effective value of AP[2] after the effects of APTable are applied.
• If hardware update of dirty state is enabled, then AP[2] is considered to be 0 for the purposes of computing execute permission in some situations.

Table D8-43 Summary of possible memory access permissions for a stage 1 translation supporting one Exception level

<table>
<thead>
<tr>
<th>XN</th>
<th>AP[2]</th>
<th>WXN</th>
<th>Access permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R, W, Executable</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>R, W, Not executable</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>x</td>
<td>R, Executable</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>x</td>
<td>R, W, Not executable</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>x</td>
<td>R, Not executable</td>
</tr>
</tbody>
</table>

For more information, see Implications of enabling the dirty state management mechanism on page D8-5147.

Instruction execution permissions for stage 1 translations

The instruction Execute-never, Unprivileged execute-never, and Privileged execute-never fields in a Block descriptor and Page descriptor are used to control instruction execution permissions by a stage 1 translation.

For a stage 1 translation that supports one Exception level, the XN field in the Block descriptor and Page descriptor has all of the following properties:
• If the Effective value of XN is 0, then instruction execution is not prevented by this bit.
• If the Effective value of XN is 1, then instruction execution is not permitted.

For a stage 1 translation that supports two Exception levels, the UXN field in the Block descriptor and Page descriptor has all of the following properties:
• If the Effective value of UXN is 0, then instruction execution is not prevented by this bit at EL0.
• If the Effective value of UXN is 1, then instruction execution is not permitted at EL0.

For a stage 1 translation that supports two Exception levels, the PXN field in the Block descriptor and Page descriptor has all of the following properties:
• If the Effective value of PXN is 0, then instruction execution is not prevented by this bit at the privileged Exception level.
• If the Effective value of PXN is 1, then instruction execution is not permitted at the privileged Exception level.

For stage 1 translations that support two Exception levels, if the value of the AP[2:1] descriptor field is 0b01, then the PXN descriptor field is treated as if it has the value 1, regardless of the actual value.

For translation regimes supporting EL0, instruction execution at a privileged Exception level is not permitted from unprivileged writable memory.

Instruction execution permissions for stage 2 translations

The instruction Execute-never field in a Block descriptor and Page descriptor is used to control instruction execution permissions by a stage 2 translation.
For a stage 2 translation, if FEAT_XNX is not implemented, then the XN field in the Block descriptor and Page descriptor controls instruction execution permissions as shown in the following table:

<table>
<thead>
<tr>
<th>XN</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The stage 2 control does not forbid execution at EL1 and EL0.</td>
</tr>
<tr>
<td>1</td>
<td>The stage 2 control does not permit execution at EL1 or EL0.</td>
</tr>
</tbody>
</table>

For a stage 2 translation, if FEAT_XNX is implemented, then the XN[1:0] field in the Block descriptor and Page descriptor controls instruction execution permissions as shown in the following table:

<table>
<thead>
<tr>
<th>XN[1]</th>
<th>XN[0]</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>The stage 2 control does not forbid execution at EL1 and EL0.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>The stage 2 control does not permit execution at EL1, but does not forbid execution at EL0.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>The stage 2 control does not permit execution at EL1 or EL0.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>The stage 2 control does not forbid execution at EL1, but does not permit execution at EL0.</td>
</tr>
</tbody>
</table>

**Effect of both stage 1 and stage 2 on instruction access permissions**

Both the stage 1 and stage 2 permissions determine the final instruction access permissions.

If both stage 1 and stage 2 translations are enabled, then in the absence of other higher-priority MMU faults, the instruction execution permissions are combined in one of the following ways:

1. If instruction execution is not permitted by the stage 1 permissions, then a stage 1 Permission fault is generated, regardless of the stage 2 permissions.
2. If instruction execution is permitted by the stage 1 permissions, but is not permitted by the stage 2 permissions, then a stage 2 Permission fault is generated.
3. If instruction execution is permitted by both the stage 1 permissions and the stage 2 permissions, then a Permission fault is not generated.

For more information, see [MMU fault prioritization from a single address translation stage on page D8-5190](#).

### Hierarchical control of instruction execution permissions

Stage 1 translation table entries at a given lookup level can limit instruction execution permissions at subsequent lookup levels.

For a stage 1 translation, the value of the XNTable Table descriptor field has one of the following effects:

- If the **effective value** of the XNTable field is 0, then the field has no effect.
- If the effective value of the XNTable field is 1, then all of the following apply:
  - The XNTable field is treated as 1 in all Table descriptors in subsequent lookup levels, regardless of the actual value of XNTable.
  - The XN field in Block descriptors and Page descriptors is treated as 1 in subsequent lookup levels, regardless of the actual value of XN.
  - The value and interpretation of the XNTable and XN fields in all subsequent lookup levels are otherwise unaffected.

For a stage 1 translation, the value of the UXNTable Table descriptor field has one of the following effects:

- If the **effective value** of the UXNTable field is 0, then the field has no effect.
If the Effective value of the UXNTable field is 1, then all of the following apply:

— The UXNTable field is treated as 1 in all Table descriptors in subsequent lookup levels, regardless of the actual value of UXNTable.
— The UXN field in Block descriptors and Page descriptors is treated as 1 in subsequent lookup levels, regardless of the actual value of UXN.
— The value and interpretation of the UXNTable and UXN fields in all subsequent lookup levels are otherwise unaffected.

For a stage 1 translation, the value of the PXNTable Table descriptor field has one of the following effects:

• If the Effective value of the PXNTable field is 0, then the field has no effect.
• If the Effective value of the PXNTable field is 1, then all of the following apply:
  — The PXNTable field is treated as 1 in all Table descriptors in subsequent lookup levels, regardless of the actual value of PXNTable.
  — The PXN field in Block descriptors and Page descriptors is treated as 1 in subsequent lookup levels, regardless of the actual value of PXN.
  — The value and interpretation of the PXNTable and PXN fields all subsequent lookup levels are otherwise unaffected.

The UXNTable, XNTable, and PXNTable Table descriptor fields control stage 1 translations. The corresponding bit positions are RES0 in the stage 2 Table descriptors.

For the translation regime controlled by a TCR_ELx, one or more of the following can be used to disable the Table descriptor XNTable, UXNTable, and PXNTable fields so they are ignored by the PE and the system behavior is as if the values are zero:

• If the Effective value of TCR_ELx.HPD{0} is 1, then hierarchical instruction execution permission control is disabled in the translation tables pointed to by TTBR0_ELx.
• If the Effective value of TCR_ELx.HPD1 is 1, then hierarchical instruction execution permission control is disabled in the translation tables pointed to by TTBR1_ELx.

The Table descriptor UXNTable, XNTable, and PXNTable fields affect all subsequent lookup levels, including stage 1 translation output. When a UXNTable, XNTable, or PXNTable field is changed, software is required to use a break-before-make sequence, including TLB maintenance for all lookup levels for the VA range translated by the descriptor.

Preventing execution from writable locations

There are register control fields that can be used to force writable memory to be treated as XN, PXN, or UXN, regardless of the value of the corresponding descriptor fields.

For a stage 1 translation in a translation regime that supports two Exception levels, the corresponding SCTLR_ELx.WXN field does all of the following:

• If the value is 0, then there is no effect on access permissions.
• If the value is 1, then all of the following apply:
  — If the memory region is writable from EL0, then it is treated as Unprivileged execute-never, regardless of the value of the Block descriptor or Page descriptor UXN field.
  — If the memory region is writable from the higher Exception level, ELx, then it is treated as Privileged execute-never, regardless of the value of the Block descriptor or Page descriptor PXN field.

For a stage 1 translation in a translation regime that applies to a single Exception level, the corresponding SCTLR_ELx.WXN field does all of the following:

• If the value is 0, then there is no effect on access permissions.
• If the value is 1 and if the memory region is writable, then it is treated as Execute-never, regardless of the value of the Block descriptor or Page descriptor XN field.
The SCTLR_ELx.WXN field reduces the memory footprint available for code injection attacks by introducing an invariant that writable memory regions are not executable.

For the lifetime of a given virtual machine, Arm expects the SCTLR_ELx.WXN field to remain static in normal operation.

SCTRL_ELx.WXN is permitted to be cached in a TLB.

**Restriction on Secure instruction fetch**

The SCR_EL3.SIF bit can be used to prevent instruction execution from Non-secure memory at stage 1 when the PE is executing in Secure state.

If all of the following apply, then an attempt to execute an instruction fetched from memory defined by the stage 1 translation as Non-secure generates a Permission fault:

- The Effective value of SCR_EL3.SIF is 1.
- Execution is in Secure state.

SCR_EL3.SIF does not prevent speculative instruction fetch. If Secure software wants to prevent speculative instruction fetch from a memory region, including Non-secure mappings of that region, then that software needs to define the region as Device, XN.

SCR_EL3.SIF is permitted to be cached in a TLB.

TLB entries might reflect the value of SCR_EL3.SIF, therefore any change to the value of that bit requires synchronization and TLB invalidation.

For an implementation that does not implement EL3, the Effective value of SCR_EL3.SIF is 0.

**D8.4.5 The Access flag**

The AF in a Block descriptor and Page descriptor indicates one of the following:

- If the value is 0, then the memory region has not been accessed since the value of AF was last set to 0.
- If the value is 1, then the memory region has been accessed since the value of AF was last set to 0.

Descriptors with AF set to zero can never be cached in a TLB.

For more information about when translation table entries are permitted to be cached in a TLB, see *Translation Lookaside Buffers on page D8-5193.*

**Software management of the Access flag**

If the AF is not managed by hardware, software management of the AF is required.

For an implementation that does not manage the AF in hardware, when an attempt is made to translate an address using a descriptor with an AF of 0, an Access flag fault is generated.

For an implementation that does not manage the AF in hardware, when an Access flag fault is generated, software is expected to set the AF to 1 in the descriptor that generated the fault.

Setting the AF to 1 prevents the Access flag fault from being generated the next time an attempt is made to translate an address using that descriptor.

When software sets the AF to 1, there is no requirement to perform TLB invalidation after setting the flag because entries with an AF set to 0 are never held in a TLB.

If a system incorporates components that can autonomously update translation table entries that are shared with the PE, then any changes by system software to translation table entries with an AF of 0 should avoid the possibility of simultaneous updates. This excludes changes to the AF value. For example, this can be done by using use a Load-Exclusive/Store-Exclusive loop or an atomic compare-and-swap operation.
Hardware management of the Access flag

All statements in this section require implementation of the optional feature, FEAT_HAFDBS.

When the PE performs a hardware update of the AF, it sets the AF to 1 in the corresponding descriptor in memory, in a coherent manner, using an atomic read-modify-write of that descriptor.

For a translation stage, if the value of the corresponding TCR_ELx.HA is 1, then AF hardware management is enabled.

For each translation stage, AF hardware management can be enabled independently.

If AF hardware management is enabled, then hardware updates the AF when a memory access is made using a Block descriptor or Page descriptor and all of the following apply:
- The access does not generate an Alignment fault based on the memory type, or a Permission fault.
- If AF hardware management was disabled or not implemented, the access would have generated an Access flag fault.

If AF hardware management is enabled, then it is constrained unpredictable whether hardware updates the AF when a memory access is made using a Block descriptor or Page descriptor and all of the following apply:
- The access generates an Alignment fault based on the memory type or a Permission fault.
- If AF hardware management were disabled or not implemented, the access would have generated an Access flag fault.

If hardware does not update the AF from 0 to 1, then the descriptor is not permitted to be cached in a TLB.

If AF hardware management is enabled in one or both translation stages, then when a memory access occurs, one or more of the following can be updated:
- The stage 1 AF.
- Each of the stage 2 AFS used to translate accesses to the stage 1 translation table walk and the OA.

For stage 1 translations, if AF hardware management is enabled, then when stage 2 translation does not permit access to the OA returned by the stage 1 translation, it is permitted, but not required, for the stage 1 AF to be updated.

For stage 2 translations, if AF hardware management is enabled, then when stage 1 translation does not permit access, it is permitted, but not required, for the stage 2 AF to be updated.

If AF hardware management is enabled and a speculative access is successfully translated, then it is permitted, but not required, for hardware to update the AF.

When a speculative access occurs, the rules determining the AF hardware update apply.

When the translation of an architecturally executed memory access occurs, the architecture requires that AF is set to 1.

When a translation occurs, the AF is permitted to be set to 1 even in the case where no architecturally executed memory access occurs. This could be due to a speculative memory access.

For an address translation stage, if AF hardware management is enabled, then when an address translation instruction is executed, all of the following apply:
- Hardware is permitted, but not required, to update the AF.
- If hardware attempts to update the AF and the update generates a fault, it is implementation defined whether this is reported as a Data Abort or not.
- If a Data Abort does not occur, then the instruction reports a result in the PAR as if the descriptors used to translate the address have an AF value of 1.

For more information, see Using break-before-make when updating translation table entries on page D8-5198.
D8.4.6 Hardware management of the dirty state

RQRTKG All statements in this section require implementation of the OPTIONAL feature, FEAT_HAFDBS, which supports both hardware update of the Access flag and hardware update of the dirty state.

IHJTWZ The dirty state is used to indicate a memory block or page has been modified. When hardware update of the dirty state is enabled, the descriptor DBM field indicates whether the descriptor is a candidate for hardware updates of the dirty state.

RPZFOC If the DBM field in a Block descriptor or Page descriptor is 0, then all of the following apply:

- For write accesses translated by the descriptor, the generation of Permission faults are unaffected by FEAT_HAFDBS.
- Hardware does not update the dirty state.

IFLYZP For a translation stage, if all of the following apply, then dirty state hardware management is enabled:

- The value of the corresponding TCR_ELx.HD is 1.
- The value of the corresponding TCR_ELx.HA is 1.

RLRMSH For the purpose of FEAT_HAFDBS, a Block descriptor or Page descriptor can be described as having one of the following states:

- Non-writeable.
- Writeable-clean.
- Writeable-dirty.

RXSTDV If all of the following apply, then a Block descriptor or Page descriptor is described as non-writeable:

- The AP or S2AP descriptor fields do not grant write access.
- One or more of the following apply:
  - The descriptor DBM field is 0.
  - For the appropriate translation stage, dirty state hardware update is disabled.

RXZFDQH For each translation stage, if all of the following apply, then a Block descriptor or Page descriptor is described as writeable-clean:

- Dirty state hardware update is enabled at that translation stage.
- The descriptor DBM field is 1.
- For a descriptor used by stage 1 translation, the AP[2] field is 1 and this is the only reason that a Permission fault would be generated due to a write access if dirty state hardware management were disabled.
- For a descriptor used by stage 2 translation, the S2AP[1] field is 0 and this is the only reason that a Permission fault would be generated due to a write access if dirty state hardware management were disabled.

RBFQG For each translation stage, if all of the following apply, then a Block descriptor or Page descriptor is described as writeable-dirty:

- Dirty state hardware update is enabled at that translation stage.
- The descriptor DBM field is 1.
- For a stage 1 translation, the AP[2] field is 0.
- For a stage 2 translation, the S2AP[1] field is 1.

IBFDYL Only if a descriptor is writeable-clean, then hardware can update the dirty state.

RPQJHR For a translation stage, when the dirty state is updated by hardware, the PE updates the corresponding descriptor in memory, in a coherent manner, using an atomic read-modify-write of that descriptor to change the descriptor from writeable-clean to writeable-dirty.

RPRJRK If the requirements to update both the Access flag and dirty state by hardware are met, the PE is permitted to update the Access flag to 1 as part of the same atomic read-modify-write of the descriptor that performs the dirty state update.
For a translation stage, Access flag faults are reported with higher priority than Permission faults. Therefore, it is not permitted for hardware to update the dirty state so that a descriptor is writeable-dirty but has the Access flag clear.

When the dirty state is updated by hardware, the PE is required to check that any copy of the descriptor cached in a TLB is not stale with regard to the descriptor in memory.

If the descriptor has changed, then the PE is required to use the new information from the descriptor in memory, regardless of whether the modified descriptor has a different OA, different attributes, or generates a fault.

For performance purposes, when the dirty state is updated by hardware, the PE should update or invalidate any TLB entry that contains a previously-cached copy of the descriptor. This is to prevent multiple writes by the PE from causing multiple updates to the descriptor.

Arm expects many PEs to cause a translation table walk to occur when dirty state hardware management causes an update of a translation table descriptor.

If a write access translated by a writeable-clean descriptor is performed architecturally, then hardware updates the dirty state of that descriptor.

Arm expects many PEs to cause a translation table walk to occur when dirty state hardware management causes an update of a translation table descriptor.

If a write access translated by a writeable-clean descriptor is not performed architecturally, then unless specified here hardware does not update the dirty state of that descriptor. In all of the following cases, hardware is permitted to update the dirty state while attempting to translate the explicit write access:

- The translation generates an MMU fault that is reported with lower priority than a Permission fault at that translation stage.
- The access is non-speculative and the translation generates an Alignment fault only because the memory type indicated by the descriptor is Device, and the hardware update would have occurred if the Alignment fault was not generated.
- The descriptor is for a stage 1 translation, the access is non-speculative, and the stage 2 translation of the OA from stage 1 generates a stage 2 MMU fault.
- The descriptor is used to translate accesses from the Statistical Profiling Buffer. For more information, see The Profiling Buffer on page D13-5458.

If both translation stages are enabled, and the conditions to update a stage 1 descriptor by hardware are met, then the stage 2 translation of the location holding that stage 1 descriptor results in one of the following:

- The stage 2 translation grants write permissions and the stage 1 hardware update occurs, including both of the following cases:
  - The stage 2 translation already grants write permission and does not need to be updated.
  - The stage 2 translation is writeable-clean and is successfully updated to writeable-dirty.
- The stage 2 translation generates a Permission fault due to lack of write access and the stage 1 hardware update does not occur. If the access that caused the stage 1 hardware update was non-speculative, then the stage 2 MMU fault is taken as a Data Abort or Instruction Abort exception.

If a speculative update of a stage 1 Access flag would otherwise be permitted, but the stage 2 translation of the stage 1 descriptor is read-only, then the speculative update of the stage 1 Access flag does not occur.

If all of the following apply, hardware can update the dirty state of a stage 2 descriptor even if the stage 1 descriptor is not updated, including the case where the stage 1 descriptor is not a Block descriptor or Page descriptor:

- Stage 1 hardware updates of the Access flag or dirty state are enabled.
- The stage 2 translation of a stage 1 translation table walk uses a writeable-clean descriptor.

The dirty state hardware update mechanism does not affect the fault reporting priority except in all of the following cases:

- If all of the following stage 2 MMU faults are generated due to a non-speculative access that causes a stage 1 hardware update, then either fault is permitted to be reported:
  - That stage 1 hardware update generates a stage 2 MMU fault.
  - The stage 2 translation of the stage 1 OA generates a stage 2 MMU fault.
• If a hardware update is permitted, that update is permitted to generate a synchronous External abort or an IMPLEMENTATION DEFINED abort caused by a memory type not supporting an atomic read-modify-write.

If an instruction that generates more than one single-copy atomic memory access has a fault on some, but not all, of those memory accesses, then all of the following apply:
• All accesses that do not fault are permitted to cause hardware updates of the dirty state.
• For faulting accesses that meet the requirements for hardware updates of the dirty state, those accesses are also permitted to cause hardware updates of the dirty state.

For more information, see:
• MMU fault prioritization from a single address translation stage on page D8-5190.
• Faults and watchpoints on page D13-5464

Implications of enabling the dirty state management mechanism

All statements in this section require implementation of the OPTIONAL feature, FEAT_HAFDBS, which supports both hardware update of the Access flag and hardware update of the dirty state.

For stage 1 translations, translations using a descriptor in the writeable-clean state are considered writeable for the purposes of the corresponding SCTLR_ELx.WXN control.

For stage 1 translations, if the corresponding SCTLR_ELx.WXN is 1, then all of the following apply:
• For a translation regime that supports a single privilege level, translations using a writeable-clean descriptor are treated as execute-never.
• For a translation regime that supports two privilege levels, translations using a privileged writable-clean descriptor are treated as privileged execute-never.
• For a translation regime that supports two privilege levels, translations using a writeable-clean descriptor are treated as unprivileged execute-never.

For a translation regime that supports two privilege levels, translations using an unprivileged writable-clean descriptor are treated as privileged execute-never.

Cache invalidation instructions and address translation instructions never cause a dirty state hardware update of either:
• The stage 1 Block descriptor or Page descriptor translating the address specified in the instruction.
• The stage 2 Block descriptor or Page descriptor translating the OA from the stage 1 translation.

For the following instructions that require write permission, if the address specified in the instruction is translated by a writeable-clean descriptor, then the descriptor is considered to grant write access:
• Data cache invalidation instruction, DC IVAC.
• Address translation instructions, AT S12E0W, AT S12E1W, AT S1E0W, AT S1E1W, AT S1E2W, AT S1E3W.

For all of the following cases, if the stage 1 descriptor is writeable-clean, then it is IMPLEMENTATION DEFINED whether hardware updates the dirty state:
• A Store-Exclusive instruction to the memory location fails because the Exclusives monitor is not in the exclusive state.
• A synchronous External abort occurs on a write to the memory location.
• A Watchpoint exception occurs on a write to the memory location.

For all of the following cases, if the stage 2 descriptor is writeable-clean, then it is IMPLEMENTATION DEFINED whether hardware updates the dirty state:
• A Store-Exclusive instruction to the memory location fails because the Exclusives monitor is not in the exclusive state.
• A synchronous External abort occurs on a write to the memory location.
• A Watchpoint exception occurs on a write to the memory location.

A hardware update of the dirty state performed by one PE does not require invalidation of any corresponding TLB entries in other PEs.

For a CAS or CASP instruction to an address translated by a writeable-clean descriptor, if the comparison fails and the location is therefore not updated, then it is constrained unpredictable whether hardware updates the dirty state.

For a stage 2 writeable-clean descriptor with the S2AP[1:0] bits set to 0b00, if an atomic instruction to the address translated by the descriptor generates a stage 2 Permission fault as a result of not having read permission, then it is constrained unpredictable whether hardware updates the dirty state.

D8.4.7 Hardware updates to the translation tables

Hardware updates to the translation tables are subject to memory ordering requirements and restrictions on the memory types.

Ordering of hardware updates to the translation tables

A hardware update to the translation table that is caused by a load or a store, including an atomic instruction, is guaranteed to be observed, to the extent required by the Shareability attributes:

• Before a load or store, including an atomic instruction, to an arbitrary address, other than the address of the translation table entry, that appears in program order after the load or store, including an atomic instruction, causing the update to the translation table entry only if a DSB with the appropriate Shareability attributes, where the DSB applies to both loads and stores, is executed between the load or store, including an atomic instruction, that caused the update to the translation table and the subsequent load or store.

• Before a load to the translation table entry that is being updated that appears in program order after the load or store, including an atomic instruction, causing the update to the translation table entry only if a DSB with the appropriate Shareability attributes, where the DSB applies to both loads and stores, is executed between the load or store, including an atomic instruction, that caused the update to the translation table and the subsequent load.

• Before a store or atomic access to the translation table entry that is being updated that appears in program order after the load or store, including an atomic instruction, causing the update to the translation table entry.

• Before a cache maintenance instruction to an arbitrary address appearing in program order after the load or store, including an atomic instruction, causing the update to the translation table entry only if a DSB with the appropriate Shareability attributes, where the DSB applies to both loads and stores, is executed between the load or store, including an atomic instruction that caused the update to the translation table entry and the subsequent cache maintenance instruction.

An update to the translation table that is caused by a load is not ordered with respect to the load itself.

An update to the translation table that is caused by a store or an atomic access is observed by all observers, to the extent required by the Shareability attributes, before the store itself in the case that the store is to the same location as the translation table update.

An update to the translation table that is caused by a store or an atomic access is not ordered with respect to the store itself in the case that the store is not the same location as the translation table update.

Restriction on memory types for hardware updates to translation tables

A translation table can be placed in Normal memory with any Cacheability.

If hardware update of translation table entries is enabled, software is required to ensure that the translation table is located in memory that supports atomic read-modify-write updates in a coherent manner.

Atomicity properties can only be met at the system level, and some system implementations might not support this functionality in all memory regions, including all of the following:

• Any system memory type that does not support hardware cache coherency.
The AArch64 Virtual Memory System Architecture

D8.4 Memory access control

The system implementation determines which memory type is treated as Non-cacheable.

The following Conventional memory types architecturally guarantee that hardware updates of the translation tables are atomic:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with read allocation hints and write allocation hints and not transient.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with read allocation hints and write allocation hints and not transient.

If a translation table hardware update is not atomic as observed by other agents that can access memory, then the update can have one or more of the following effects:

- A synchronous External abort on the translation table walk is generated.
- The instruction generates an SError interrupt.
- An MMU fault is generated, reported as an Unsupported atomic hardware update using one of the following Fault status codes:
  - For Data Abort exceptions, ESR_ELx.DFSC = 0b110001.
  - For Instruction Abort exceptions, ESR_ELx.IFSC = 0b110001.
  - For an abort on a write to the Statistical Profiling buffer, PMBSR_EL1.FSC = 0b110001.
- The hardware updates occur and all of the following apply:
  - There is no guarantee that the memory accesses are done atomically in regard to accesses to memory by other agents.
  - The instruction might also generate an SError interrupt.

If a translation table hardware update is not atomic as observed by other agents that can access memory, and the update generates an MMU fault, all of the following apply:

- If EL2 is implemented and enabled in the current Security state, then for the EL1&0 translation regime, one of the following:
  - If the atomic hardware update is not supported in the stage 1 translation due to the defined memory type, or if the stage 2 translation is not enabled, then this exception is a stage 1 abort and is taken to EL1.
  - In all other cases, the exception is a stage 2 abort and is taken to EL2.
- For a translation stage, the MMU fault priority is at an IMPLEMENTATION DEFINED point between all of the following:
  - Immediately before the priority of an Access flag fault that is generated at the same translation stage.
  - Immediately after the priority of a Permission fault that is generated at the same translation stage.

If translation table hardware updates are not atomic as observed by other agents that can access memory, then in all of the following cases, the architecture permits, but does not require, an address translation instruction to generate an MMU fault, reported as an Unsupported atomic hardware update using PAR_EL1.FST = 0b110001:

- For a stage 1 translation at EL1, stage 1 translation table hardware updates are enabled.
- For EL2 or EL3, translation table hardware updates are enabled.

For more information, see Possible implementation restrictions on using atomic instructions on page B2-157.

Use of the Contiguous bit with hardware updates to the translation tables

When hardware updates any of the AF, AP[2] or S2AP[1] bits, then the update applies to a single translation table entry.

If hardware updates a translation table entry, and if the Contiguous bit in that entry is 1, then the members in a group of contiguous translation table entries can have different AF, AP[2], and S2AP[1] values.

For more information, see Possible implementation restrictions on using atomic instructions on page B2-157.
If all of the following apply, then the architecture does not require a hardware update of the AF:

- An access is to a location translated by an entry that has the Contiguous bit set to 1.
- At least one descriptor in the set of contiguous translation table entries has AF set to 1.

If all of the following apply, then the architecture does not require a hardware update of the AP[2] or S2AP[1] bit:

- A write is done to a location translated by a descriptor that has the Contiguous bit set to 1.
- The corresponding AP[2] or S2AP[1] bit in at least one descriptor in the set of contiguous translation table entries indicates that the entry is dirty.

The Contiguous bit permits, but does not require, hardware to hold a single entry in a TLB for the set of translation table entries in the group, and to have updated only one or more of the AF and the AP[2] bit or S2AP[1] bit in the single translation table entry that resulted in the TLB entry.

Software is required to combine the AF values and AP[2] or S2AP[1] values across all translation table entries in a contiguous group to determine whether any of the entries have been accessed or are dirty.

For more information on the Contiguous bit, see *The Contiguous bit* on page D8-5158.
D8.5 Memory region attributes

The memory region attributes in a descriptor define the memory type, Cacheability, and whether the memory region is Shareable. For translation regimes that support two translation stages, the stage 1 and stage 2 memory region attributes are combined to produce a resultant memory type and Cacheability, and Shareability.

All of the following apply to the memory region attributes:

- The stage 1 OA attributes are derived from a combination of the Block descriptor and Page descriptor AttrIndx[2:0] field, and the MAIR_ELx register fields.
- The stage 1 translation table walk attributes are derived from the TCR_ELx.{IRGN_n, ORGN_n, SHn} fields.
- The SCTLR_ELx.{C, I} fields might override stage 1 attributes.
- The HCR_EL2.PTW field might override stage 2 translation attributes of a stage 1 translation table walk.
- If EL2 is enabled, then for the EL1&0 translation regime the HCR_EL2.{CD, DC} fields might override stage 1 and stage 2 attributes.
- All of the following HCR_EL2.FWB values affect the stage 2 attributes:
  - When the Effective value of HCR_EL2.FWB is 0, stage 2 attributes are derived from the stage 2 MemAttr[3:2] descriptor bits, then combined with stage 1 attributes.
  - When the Effective value of HCR_EL2.FWB is 1, stage 2 attributes are derived from the stage 2 MemAttr[2:0] descriptor bits.
- For Device memory and Normal Non-cacheable mappings from stage 1, normalization of the Shareability that is input to stage 2 is IMPLEMENTATION DEFINED.
- For Device memory and Normal Non-cacheable mappings from stage 2, Shareability is always normalized to Outer Shareable.

For a virtualization implementation, combining memory attributes from stage 1 and stage 2 translations support all of the following functions that are useful to a hypervisor executing at EL2:

- Reduce or increase the permitted Cacheability of a region.
- Increase the required Shareability of a region.

The effects of combining memory attributes defined by the hypervisor are expected to be functionally transparent to the Guest OS.

If all of the following apply, memory locations are treated as Tagged:

- The combined effects of stage 1 and stage 2 translations define the memory attributes as all of the following:
  - Normal memory.
  - Inner, and Outer Write-Back Non-Transient Read-Allocate Write-Allocate.
- The stage 1 translation is treated as Tagged.

For more information, see Tagged and Untagged Addresses on page D9-5224.

D8.5.1 Stage 1 memory type and Cacheability attributes

For a stage 1 translation, the Block descriptor and Page descriptor AttrIndx[2:0] field holds the value $n$ used to select the 8-bit field MAIR_ELx.Attr$<n>$ that specifies the memory region attributes of the descriptor OA.

For the memory region specified by a Block descriptor and Page descriptor, the MAIR_ELx.Attr$<n>$ field selected by the descriptor AttrIndx[2:0] field defines all of the following:

- The Device or Normal memory type.
- For Device memory, all of the following:
  - One of the Device-nGnRnE, Device-nGnRE, Device-nGRE, or Device-GRE Device memory types.
  - If FEAT_XS is implemented, then the XS attribute.
For Normal memory, all of the following:

- For both inner and outer Cacheability, one of the Non-cacheable, Write-Through, or Write-Back attributes.
- For Write-Through Cacheable and Write-Back Cacheable regions, the Read-Allocate and Write-Allocate hints, each of which is Allocate or No Allocate, and the Transient allocation hints.
- If FEAT_MTE2 is implemented, then the Tagged attribute.
- If FEAT_XS is implemented, then the XS attribute.

For more information, see XS attribute modifier on page D8-5160 and Tagged and Untagged Addresses on page D9-5224.

For some translation regimes, the memory region attributes determined from translation tables and MAIR_ELx values might be overridden by the configuration of SCTLR_ELx.[C, I] and HCR_EL2.DC.

### D8.5.2 Stage 1 Shareability attributes

For Normal Cacheable memory, if the Effective value of TCR_ELx.DS is 0, then the SH[1:0] field in a stage 1 translation Block descriptor and Page descriptor encodes the Shareability attributes of the descriptor OA, as shown in the following table:

<table>
<thead>
<tr>
<th>SH[1:0]</th>
<th>Normal memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Non-shareable</td>
</tr>
<tr>
<td>01</td>
<td>Reserved, CONSTRAINT UNPREDICTABLE as described in Reserved values in System and memory-mapped registers and translation table entries on page K1-11583.</td>
</tr>
<tr>
<td>10</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td>11</td>
<td>Inner Shareable</td>
</tr>
</tbody>
</table>

If a region is mapped as Device memory or Normal Non-cacheable memory after all enabled translation stages, then the region has an effective Shareability attribute of Outer Shareable.

If stage 2 translation is enabled and stage 1 maps a region as Device memory or Normal Non-cacheable memory, then it is IMPLEMENTATION DEFINED whether:

- The Shareability attribute configured for stage 1 is input into stage 2 translation.
- An effective Outer Shareable attribute is input into stage 2 translation.

If the Effective value of TCR_ELx.DS is 1, then the stage 1 translation descriptor bits[9:8] are OA[51:50] instead of SH[1:0].

For Normal Cacheable memory, if the Effective value of TCR_ELx.DS is 1, then the stage 1 translation Shareability is treated as being mapped by one of the following:

- For the EL3 translation regime, TCR_EL3.SH0.
- For the EL2 translation regime, if HCR_EL2.E2H is 0, then TCR_EL2.SH0.
- For the EL2 and EL2&0 translation regimes, if HCR_EL2.E2H is 1 and the VA is an address that is translated using tables pointed to by TTBR0_EL2, then TCR_EL2.SH0.
- For the EL2 and EL2&0 translation regimes, if HCR_EL2.E2H is 1 and the VA is an address that is translated using tables pointed to by TTBR1_EL2, then TCR_EL2.SH1.
- For the EL1&0 translation regime, if the VA is an address that is translated using tables pointed to by TTBR0_EL1, then TCR_EL1.SH0.
- For the EL1&0 translation regime, if the VA is an address that is translated using tables pointed to by TTBR1_EL1, then TCR_EL1.SH1.
D8.5.3 Stage 2 memory type and Cacheability attributes

For any stage 2 translation, if the final memory type any is Normal Cacheable type, then all of the following apply:

- If the output of stage 1 specifies a Cacheable memory type, then the final cache allocation hints are the stage 1 cache allocation hints.
- If the output of stage 1 does not specify a Cacheable memory type, then the final cache allocation hints are Read Allocate, Write Allocate.

Stage 2 translation configuration does not assign cache allocation hints.

If the memory type from a stage 2 translation causes a stage 1 translation table walk to a Device memory type, then one of the following occurs:

- If HCR_EL2.PTW is 0, then the translation table walk occurs as if it is to Normal Non-cacheable memory, and the walk can be done speculatively.
- If HCR_EL2.PTW is 1, then the memory access generates a stage 2 Permission fault.

D8.5.4 Stage 2 memory type and Cacheability attributes when FEAT_S2FWB is disabled

All statements in this section and subsections require that the Effective value of HCR_EL2.FWB is 0.

If the Effective value of HCR_EL2.FWB is 0, then the stage 1 and stage 2 memory type and Cacheability attributes are combined.

For stage 2 translations, the MemAttr[3:0] field in a Block descriptor and Page descriptor encodes the memory type and Cacheability of the memory region addressed by the descriptor OA.

For stage 2 translations, the MemAttr[3:2] descriptor bits encode the memory type and, for Normal memory, the Outer Cacheability, of the descriptor OA as shown in the following table:

<table>
<thead>
<tr>
<th>MemAttr[3:2]</th>
<th>Memory type</th>
<th>Outer Cacheability</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Device</td>
<td>Not applicable</td>
</tr>
<tr>
<td>01</td>
<td>Normal</td>
<td>Outer Non-cacheable</td>
</tr>
<tr>
<td>10</td>
<td>Normal</td>
<td>Outer Write-Through Cacheable</td>
</tr>
<tr>
<td>11</td>
<td>Normal</td>
<td>Outer Write-Back Cacheable</td>
</tr>
</tbody>
</table>

For stage 2 translations, if the MemAttr[3:2] descriptor bits are 0b00, indicating Device memory, then the MemAttr[1:0] descriptor bits encode the Device memory type as shown in the following table:

<table>
<thead>
<tr>
<th>MemAttr[1:0]</th>
<th>Encoding if MemAttr[3:2] == 0b00</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Region is Device-nGnRnE memory</td>
</tr>
<tr>
<td>01</td>
<td>Region is Device-nGnRE memory</td>
</tr>
<tr>
<td>10</td>
<td>Region is Device-nGRE memory</td>
</tr>
<tr>
<td>11</td>
<td>Region is Device-GRE memory</td>
</tr>
</tbody>
</table>
For stage 2 translations, if the MemAttr[3:2] descriptor bits are not 0b00, indicating Normal memory, then the MemAttr[1:0] descriptor bits encode the Normal memory Inner Cacheability as shown in the following table:

<table>
<thead>
<tr>
<th>MemAttr[1:0] Encoding if MemAttr[3:2] != 0b00</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

For more information, see:

- *Combining stage 1 and stage 2 memory type attributes*.

**Combining stage 1 and stage 2 memory type attributes**

If the Effective value of HCR_EL2.FWB is 0, then the stage 1 and stage 2 memory type attributes are combined as shown in the following table:

<table>
<thead>
<tr>
<th>Rule</th>
<th>If either translation stage assigns:</th>
<th>Resultant memory type is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device has precedence over Normal</td>
<td>Any Device memory type</td>
<td>Any Device memory type</td>
</tr>
<tr>
<td>non-Gathering has precedence over Gathering</td>
<td>A Device-nGxx memory type</td>
<td>A Device-nGxx memory type</td>
</tr>
<tr>
<td>non-Reordering has precedence over Reordering</td>
<td>A Device-nGnRx memory type</td>
<td>A Device-nGnRx memory type</td>
</tr>
<tr>
<td>No Early write acknowledge has precedence over Early write acknowledge</td>
<td>The Device-nGnRnE memory type</td>
<td>The Device-nGnRnE memory type</td>
</tr>
</tbody>
</table>

**Combining stage 1 and stage 2 Cacheability attributes for Normal memory**

If the Effective value of HCR_EL2.FWB is 0, then the stage 1 and stage 2 Inner Cacheability and Outer Cacheability attributes are combined as shown in the following table:

<table>
<thead>
<tr>
<th>Assignment in stage 1</th>
<th>Assignment in stage 2</th>
<th>Resultant Cacheability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-cacheable</td>
<td>Any</td>
<td>Non-cacheable</td>
</tr>
<tr>
<td>Any</td>
<td>Non-cacheable</td>
<td>Non-cacheable</td>
</tr>
<tr>
<td>Write-Through Cacheable</td>
<td>Write-Through Cacheable</td>
<td>Write-Through Cacheable</td>
</tr>
</tbody>
</table>
For stage 2 translations, if the Effective value of HCR_EL2.FWB is 0 and the stage 2 MemAttr[3:0] field is 0b1111, then the combined memory type and Cacheability attributes are the output memory type and Cacheability attributes from stage 1.

### D8.5.5 Stage 2 memory type and Cacheability attributes when FEAT_S2FWB is enabled

All statements in this section and subsections require that the Effective value of HCR_EL2.FWB is 1.

For stage 2 translations, FEAT_S2FWB has all of the following effects on the MemAttr[3:2] bits:

- MemAttr[3] is RES0.

For stage 2 translations, if MemAttr[2] is 0, then the MemAttr[1:0] bits define Device memory attributes as shown in the following table:

<table>
<thead>
<tr>
<th>Stage 2 MemAttr[1:0]</th>
<th>Device Memory Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Device-nGnRnE</td>
</tr>
<tr>
<td>01</td>
<td>Device-nGnRE</td>
</tr>
<tr>
<td>10</td>
<td>Device-nGRE</td>
</tr>
<tr>
<td>11</td>
<td>Device-GRE</td>
</tr>
</tbody>
</table>

For stage 2 translations, if MemAttr[2] is 1, then the MemAttr[1:0] bits affect the memory type and Cacheability as shown in the following table:

<table>
<thead>
<tr>
<th>Stage 1 memory type and Cacheability attribute</th>
<th>Stage 2 MemAttr[1:0]</th>
<th>Resultant memory type and Cacheability attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal Write-Back</td>
<td>11</td>
<td>Normal Write-Back</td>
</tr>
<tr>
<td>Normal Write-Through</td>
<td>11</td>
<td>Normal Write-Through</td>
</tr>
<tr>
<td>Normal Non-cacheable</td>
<td>11</td>
<td>Normal Non-cacheable</td>
</tr>
<tr>
<td>Device</td>
<td>11</td>
<td>Device</td>
</tr>
<tr>
<td>Normal Write-Back</td>
<td>10</td>
<td>Normal Write-Back</td>
</tr>
<tr>
<td>Normal Write-Through</td>
<td>10</td>
<td>Normal Write-Back</td>
</tr>
<tr>
<td>Normal Non-cacheable</td>
<td>10</td>
<td>Normal Write-Back</td>
</tr>
<tr>
<td>Device</td>
<td>10</td>
<td>Normal Write-Back</td>
</tr>
<tr>
<td>Normal Write-Back</td>
<td>01</td>
<td>Normal Non-cacheable</td>
</tr>
</tbody>
</table>
RRMTWN  If the stage 1 translation specifies a Device memory type, and the stage 2 descriptor MemAttr[2:0] field is \(0b110\), then all of the following apply:

- If an atomic memory access occurs, then it is IMPLEMENTATION DEFINED whether it is supported in the same way as accesses to memory locations with a resultant Device memory type.
- If an exclusive access occurs, then it is IMPLEMENTATION DEFINED whether it is supported in the same way as accesses to memory locations with a resultant Device memory type.
- If a misaligned access occurs, then it is CONSTRAINED UNPREDICTABLE whether the resultant stage 1 memory type generates a stage 1 Alignment fault.
- For a DC ZVA, DC GZVA, or DC GVA instruction, it is CONSTRAINED UNPREDICTABLE whether the resultant stage 1 memory type generates a stage 1 Alignment fault.

RRDBK  The architecture requires that CLIDR_EL1.{LoUU, LoIUS} are \(\{0, 0\}\) so that no data cache levels need to be cleaned to manage coherency with instruction fetches.

RDPDXS  If the stage 1 translation specifies a Normal memory type with Cacheability other than Write-Back Cacheable, and the stage 2 descriptor MemAttr[2:0] field is \(0b110\), then it is IMPLEMENTATION DEFINED whether atomic memory accesses or exclusives are supported, in the same way as accesses to a Normal memory type with Cacheability other than Write-Back Cacheable.

LRWWR  For more information, see:
- XS attribute modifier on page D8-5160.
- Reserved values in System and memory-mapped registers and translation table entries on page K1-11600.

### D8.5.6  Stage 2 Shareability attributes

ICBHV  The stage 2 Shareability attributes are not affected by the Effective value of HCR_EL2.FWB.

DNZIQ  For Normal Cacheable memory, if the Effective value of VTCR_EL2.DS is 0, then the SH[1:0] field in a stage 2 translation Block descriptor and Page descriptor encodes the Shareability attributes of the descriptor OA, as shown in the following table:

<table>
<thead>
<tr>
<th>SH[1:0]</th>
<th>Normal memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Non-shareable</td>
</tr>
<tr>
<td>01</td>
<td>Reserved, CONSTRAINED UNPREDICTABLE</td>
</tr>
<tr>
<td>10</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td>11</td>
<td>Inner Shareable</td>
</tr>
</tbody>
</table>

HTSGQ  For Normal Cacheable memory, if the Effective value of VTCR_EL2.DS is 1, then the stage 2 translation Shareability is determined by VTCR_EL2.SH0.
For Device memory and Normal Non-cacheable memory, the Shareability attributes of the stage 2 translation descriptor OA are treated as Outer Shareable.

If the Effective value of VTCR_EL2.DS is 1, then the stage 2 translation descriptor bits[9:8] are OA[51:50] instead of SH[1:0].

For more information, see Reserved values in System and memory-mapped registers and translation table entries on page K1-11600.

Combining the stage 1 and stage 2 Shareability attributes for Normal memory

The value of HCR_EL2.FWB does not affect how Shareability attributes from stage 1 and stage 2 are combined.

If the resultant memory type from a stage 1 and stage 2 translation is one of the following, the memory region is treated as Outer Shareable:

- Any Device memory type.
- Normal Inner Non-cacheable, Outer Non-cacheable.

For a memory region with a resultant memory type of Normal that is not Inner Non-cacheable, Outer Non-cacheable, the stage 1 and stage 2 Shareability attributes can be combined as shown in the following table:

<table>
<thead>
<tr>
<th>Assignment in stage 1</th>
<th>Assignment in stage 2</th>
<th>Resultant Shareability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outer Shareable</td>
<td>Any</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td>Inner Shareable</td>
<td>Outer Shareable</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td>Inner Shareable</td>
<td>Inner Shareable</td>
<td>Inner Shareable</td>
</tr>
<tr>
<td>Inner Shareable</td>
<td>Non-shareable</td>
<td>Inner Shareable</td>
</tr>
<tr>
<td>Non-shareable</td>
<td>Outer Shareable</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td>Non-shareable</td>
<td>Inner Shareable</td>
<td>Inner Shareable</td>
</tr>
<tr>
<td>Non-shareable</td>
<td>Non-shareable</td>
<td>Non-shareable</td>
</tr>
</tbody>
</table>
D8.6 Other descriptor fields

D8.6.1 The Contiguous bit

The Contiguous bit identifies a descriptor as belonging to a group of adjacent translation table entries that point to a contiguous OA range.

If the Effective value of the Contiguous bit in a Block descriptor or Page descriptor is 1, and the descriptor would otherwise be permitted to be cached in a TLB, then all of the following apply:

- The entry is permitted to be cached in a TLB as though it is one of a number of adjacent translation table entries that point to a contiguous OA range with consistent attributes and permissions.
- Software is required to ensure that all of the adjacent translation table entries for the contiguous region point to a contiguous OA range with consistent attributes and permissions.

The Contiguous bit is \texttt{RES0} and has an Effective value of 0 in all of the following Block descriptors:

- For the 4KB translation granule, if the Effective value of TCR\_ELx.DS or VTCR\_EL2.DS is 1, then the Contiguous bit is \texttt{RES0} in the level 0 Block descriptor of that translation regime.
- For the 16KB translation granule, if the Effective value of TCR\_ELx.DS or VTCR\_EL2.DS is 1, then the Contiguous bit is \texttt{RES0} in the level 1 Block descriptor of that translation regime.
- For the 64KB translation granules, if FEAT\_LPA is implemented, then the Contiguous bit is \texttt{RES0} in a level 1 Block descriptor.

For a lookup level in a translation granule, if the Effective value of the Contiguous bit is 1, then the entry is permitted to be cached in a TLB as though all of the properties shown in the following table apply:

<table>
<thead>
<tr>
<th>Translation granule</th>
<th>Block or Page lookup level</th>
<th>Number of adjacent translation table entries</th>
<th>Alignment boundary of adjacent translation table entries within the translation table</th>
<th>Alignment of contiguous OA range</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB</td>
<td>Any</td>
<td>16</td>
<td>128 bytes</td>
<td>Multiple of 16 of the block or page size at that lookup level</td>
</tr>
<tr>
<td>16KB</td>
<td>2</td>
<td>32</td>
<td>256 bytes</td>
<td>1GB</td>
</tr>
<tr>
<td>16KB</td>
<td>3</td>
<td>128</td>
<td>1KB</td>
<td>2MB</td>
</tr>
<tr>
<td>64KB</td>
<td>Any</td>
<td>32</td>
<td>256 bytes</td>
<td>Multiple of 32 of the block or page size at that lookup level</td>
</tr>
</tbody>
</table>

The architecture does not require descriptors with the Contiguous bit set to 1 to be cached as a single TLB entry for the contiguous region. To avoid TLB coherency issues, software is required to perform TLB maintenance on the entire address region that results from using the Contiguous bit.

If the Effective value of the Contiguous bit is 1, then hardware updates to the AF and dirty state can cause members in a group of contiguous translation table entries to have different AF, AP[2], and S2AP[1] values.

For more information, see *Use of the Contiguous bit with hardware updates to the translation tables* on page D8-5149.

**Misprogramming the Contiguous bit**

For all descriptors within the range indicated by one or more descriptors that have the Contiguous bit set to 1, if one or more of the following translation table programming errors is made, then a TLB might contain overlapping entries:

- One or more of the contiguous translation table entries does not have the Contiguous bit set to 1.
• One or more of the contiguous translation table entries holds an OA that is not consistent with all of the
entries pointing to the same aligned contiguous address range.
• The attributes and permissions of the contiguous entries are not the same.

For a TLB lookup in a contiguous region mapped by translation table entries that are misprogrammed, that TLB
lookup is permitted to produce one of the following:
• An OA, access permissions, and memory attributes that are consistent with any of the programmed
translation table values.
• If BBM support levels 1 and 2 are not implemented, then an OA, access permissions, or memory attributes
that are inconsistent with any of the programmed translation table values. For more information, see Support
levels for changing block size on page D8-5199.
• A TLB conflict abort.

Architectural guarantees when the Contiguous bit is misprogrammed

If the Contiguous bit is misprogrammed, then the architecture guarantees all of the following:
• If physical memory regions are inaccessible through translation tables programmed at EL1 when the
Contiguous bit is not misprogrammed, then they remain inaccessible to software executing at EL1 or EL0.
• If memory attributes and permissions are unattainable through translation tables programmed at EL1 when
the Contiguous bit is not misprogrammed, then they remain unattainable to software executing at EL1 or
EL0.
• Software executing in Non-secure state cannot access Secure physical memory.

The PE is required to generate an Address Size fault on accesses to addresses above the configured OA size, even
in the case where the Contiguous bit is used to mark a set of Block descriptors as contiguous, such that the OA region
translated by that contiguous set of descriptors exceeds the configured OA size.

An implementation is permitted to ignore the Contiguous bit to ensure that OA range checking is done.

Implementation options when the Contiguous bit is misprogrammed

If software does all of the following, it is considered a programming error:
• The Contiguous bit is used to mark a set of translation table entries as contiguous.
• The address range translated by the set of translation table entries is larger than the IA size configured for the
translation stage as determined by the TCR_ELx.TxSZ field.

If the Contiguous bit is used to mark a set of translation table entries as contiguous and if the address range translated
by the translation table entries is larger than the IA size supported by the translation stage, as defined by the
TCR_ELx.TxSZ field, then an implementation is permitted, but not required to, do one or more of the following:
• Generate a Translation fault.
• When all of the following apply, an implementation is permitted to not generate a Translation fault:
  — A translation table entry within a contiguous set of translation table entries is accessed.
  — The translation table entry is valid.

D8.6.2 Page Based Hardware attributes

The PBHA bits can be used for IMPLEMENTATION DEFINED purposes.

All statements in this section require implementation of FEAT_HPDS2.

For a stage 1 translation that uses TTBR0_ELx, if all of the following apply, then hardware can use the PBHA bit
in the corresponding Block descriptor or Page descriptor bit[nn] for IMPLEMENTATION DEFINED purposes:
• The corresponding TCR_ELx.HWU0nn bit, HWU62, HWU61, HWU60, or HWU59, is 1.
• TCR_ELx.HPD0 is 1.
For a stage 1 translation that uses TTBR1_ELx, if all of the following apply, then hardware can use the PBHA bit in the corresponding Block descriptor or Page descriptor bit[nn] for IMPLEMENTATION DEFINED purposes:

- The corresponding TCR_ELx.HWU{nn} bit, HWU162, HWU161, HWU160, or HWU159, is 1.
- TCR_ELx.HPD1 is 1.

For a stage 2 translation, if the value of VTCR_EL2.HWU{nn} is 1, then hardware can use the PBHA bit in the corresponding Block descriptor or Page descriptor bit[nn] for IMPLEMENTATION DEFINED purposes.

If the PBHA bit is used for IMPLEMENTATION DEFINED purposes, then setting the PBHA bit to 0 has the same behavior as when the PBHA bit is not used for IMPLEMENTATION DEFINED purposes.

If the Effective value of TCR_ELx.HWU{0, 1}nn is 0, then the PBHA bit in the corresponding Block descriptor or Page descriptor bit[nn] is IGNORED.

### D8.6.3 Block translation entry

Setting the nT bit in a Block descriptor guarantees that, when changing block size, accesses translated by the translation table entry do not break coherency, ordering guarantees or uniprocessor semantics, or fail to clear the Exclusives monitors.

All statements in this section require implementation of FEAT_BBM.

If the implementation meets either level 1 or level 2 support requirements for changing block size, then when using a Block descriptor with the nT bit set, the PE is permitted to do one of the following:

- Generate a Translation fault and not cache the entry in a TLB.
- If an entry that does not have the nT bit set is cached within a TLB and translates the same address to the same output address with consistent memory attributes and permissions, then the PE guarantees that accesses translated by the translation table entry with the nT bit set does not break coherency, ordering guarantees or uniprocessor semantics, or fail to clear the Exclusives monitors.

For more information, see Support levels for changing block size on page D8-5199.

If the nT bit in a Block descriptor is set, then the translation performance can be significantly impacted.

### D8.6.4 XS attribute modifier

The XS attribute indicates that an access to the memory region could take a long time to complete. Variants of DSB instructions and TLB maintenance instructions that do not depend on the completion of memory accesses with the XS attribute set to 1 are defined.

All statements in this section require implementation of FEAT_XS.

For a stage 2 translation, the FnXS bit in a Block descriptor and Page descriptor has all of the following properties:

- If the FnXS bit is 0, then the XS attribute of the resultant memory translation is not modified.
- If the FnXS bit is 1, then the XS attribute of the resultant memory translation is set to 0.

For stage 2 translations, if the resultant memory attributes are Normal Inner Write-back, Outer Write-back Cacheable, then the XS attribute is set to 0 on the resultant memory translation.

If FEAT_XS is implemented and the stage 1 memory type defined by the MAIR_ELx or TCR_ELx registers is one of the following, then the XS attribute is set to 0, otherwise the XS attribute is set to 1:

- Device memory types that use the MAIR_ELx.Attr<n> encoding `0b0000dd01`.
- For Normal memory, one of the following:
  - Inner Write-Back Cacheable, Outer Write-back Cacheable memory types defined in the MAIR_ELx or TCR_ELx registers, including any memory types that are treated as Write-Back Cacheable as a result of IMPLEMENTATION DEFINED choices in the architecture.
  - Inner Write-through Cacheable and Outer Write-through Cacheable memory types that use the MAIR_ELx.Attr<n> encoding `0b10100000`. 

For more information, see Support levels for changing block size on page D8-5199.
Inner Non-cacheable, Outer Non-cacheable memory types that use the MAIR_ELx.Attr<\text{n}> encoding 0b01000000.

The stage 2 impact of the FnXS bit applies for stage 1 translations in the EL1&0 translation regime from AArch32 or AArch64.

For more information, see:

- *Enabling and disabling the caching of memory accesses* on page D7-5047.
- *Stage 1 memory type and Cacheability attributes* on page D8-5151.
- *Stage 2 memory type and Cacheability attributes when FEAT_S2FWB is enabled* on page D8-5155.

### D8.6.5 IGNORED fields

For stage 1 and stage 2 Table descriptors, bits[58:51] and bits[11:2] are IGNORED, and the architecture guarantees that a PE does not use the bits. See Figure D8-16 on page D8-5127 and Figure D8-17 on page D8-5128.

For stage 1 and stage 2 Block descriptors and Page descriptors, all of the following bits are IGNORED, and the architecture guarantees that a PE does not use the bits:

- If FEAT_HPDS2 is not implemented, then descriptor bits[63:55].
- If FEAT_HPDS2 is implemented, then descriptor bit[63] and bits[58:55].

If FEAT_HPDS2 is not implemented, then in the stage 2 Block descriptor and Page descriptor, bits[63:59] are reserved for a System MMU.

If FEAT_HPDS2 is implemented, then in the stage 2 Block descriptor and Page descriptor, bit[63] is reserved for a System MMU.

In the stage 1 and stage 2 Block descriptor and Page descriptor, bits[58:55] have all of the following properties:

- The field is for software use.
- The architecture guarantees that hardware does not use the field.
- If the bits in the field are changed, there is no need to invalidate a TLB.
D8.7 Address tagging

When address tagging is enabled, the top eight bits of the VA are ignored for the purposes of address translation and they are instead described as an address tag. An address tag might be used for software purposes, an MTE Logical Address Tag, or both.

If address tagging is enabled in a translation regime, then the top eight bits of the VA, bits[63:56], have all of the following properties:

- Bits[63:56] are the address tag.
- The bits are ignored during address translation.
- If the translation system is enabled, then the bits are ignored when determining whether the address is out of range and therefore generates a Translation fault.
- If the translation system is not enabled, then the bits are ignored when determining whether the address is out of range and therefore generates an Address size fault.

D8.7.1 Address tag control

Address tagging is controlled using the Top Byte Ignore (TBI) field in the TCR_ELx register.

For a stage 1 translation that supports a single VA range, TCR_ELx.TBI{0} determines whether address tagging is enabled.

For a stage 1 translation that supports two VA ranges, all of the following determine whether address tagging is enabled:

- TCR_ELx.TBI0 determines whether the lower address range uses address tags.
- TCR_ELx.TBI1 determines whether the upper address range uses address tags.

The TCR_ELx.TBI{\(n\)} bit controls address tagging whether or not the corresponding stage 1 translation is enabled.

If FEAT_PAuth is implemented and a TCR_ELx.TBI{\(n\)} bit enables address tagging, then the corresponding TCR_ELx.TBID{\(n\)} bit determines whether address tagging applies to both instruction addresses and data addresses, or just data addresses. For more information, see Pointer authentication on page D8-5164.

D8.7.2 Effect of address tagging on the PC

This section describes how the top byte of an address is considered when address tagging is enabled and an address is propagated to the PC for a branch, procedure return, exception, or exception return occurs.

For a stage 1 translation that supports a single VA range, if address tagging for instruction accesses is enabled in an Exception level, ELx, then bits[63:56] of the address loaded into the PC are forced to 0x00 in all of the following cases:

- A branch or procedure return within ELx.
- An exception is taken to ELx.
- When an exception return does not generate an Illegal exception return, any exception return to ELx.
- Exiting from Debug state to ELx.

For a stage 1 translation that supports two VA ranges, if address tagging for instruction accesses is enabled in an Exception level, ELx, then bits[63:56] of the address loaded into the PC are a sign-extension of address bit[55] in all of the following cases:

- A branch or procedure return within ELx.
- An exception is taken to ELx.
- When an exception return does not generate an Illegal exception return, any exception return to ELx.
- Exiting from Debug state to ELx.

If the value of the SPSR_ELx.M[4] bit of the saved process state is 1 when an Illegal exception return occurs, indicating a return to AArch32 state, then PC bits[63:32] are UNKNOWN.
When address tagging is enabled and an address causes a Data Abort or a *Watchpoint exception*, the address tag is included in the VA returned in the FAR.
D8.8 Pointer authentication

AArch64 state supports authenticating the contents of a register before that register is used as the target of an indirect branch, or as a load.

All statements in this section and subsections require implementation of FEAT_PAuth.

Pointer authentication is supported only in AArch64 state.

The pointer authentication mechanism treats the upper bits of a pointer stored in a register in one of the following ways:

- If FEAT_PAuth2 is not implemented, then the pointer authentication mechanism replaces the upper bits of a pointer stored in a register with a pointer authentication code (PAC).
- If FEAT_PAuth2 is implemented, then the pointer authentication mechanism exclusive-ORs the upper bits of a pointer stored in a register with a PAC.
- If FEAT_EPAC is implemented, then when performing a PAC operation on a non-canonical address, the pointer authentication mechanism sets the PAC field to 0.

If FEAT_CONSTPACFIELD is implemented, then an implementation is permitted to use the value in Xn[55] to determine the size of the PAC field, even when address tagging is not used.

If the value of one or more of the ID_AA64ISAR1_EL1.{GPI, GPA, API, APA} or ID_AA64ISAR2_EL1.{GPA3, APA3} fields is non-zero, then pointer authentication is implemented.

Pointer authentication functionality is the same whether address translation is enabled or disabled, but is useful only when address translation is enabled.

D8.8.1 PAC field

The TCR_ELx.{T0SZ, T1SZ} fields are used to configure the size and location of the PAC field.

When a 64-bit register, Xn, holds an address with a PAC field, the location of the PAC field is determined by all of the following:

- The bottom_PAC_bit is 64-TCR_ELx.TnSZ.
- If FEAT_CONSTPACFIELD is implemented, or address tagging is used, then the value of Xn[55] is the value of n used in TnSZ.
- If FEAT_CONSTPACFIELD is not implemented and address tagging is not used, then the value of Xn[55] is the value of n used in TnSZ.
- If address tagging is used, the PAC field is Xn[54:bottom_PAC_bit].
- If address tagging is not used, the PAC field is Xn[63:56, 54:bottom_PAC_bit].

For more information, see Address tagging on page D8-5162.

Xn[55] always determines whether Xn[63:56] is part of the PAC field, and it is IMPLEMENTATION DEFINED whether Xn[55] or Xn[63] determines which of TCR_ELx.{T0SZ, T1SZ} are used to determine the value of bottom_PAC_bit.

If the TCR_ELx.TBID{r} bit is used to restrict address tagging to data addresses, then instruction addresses can use Xn[63:56] bits to produce larger pointer authentication code fields.

For all of the following reasons, if FEAT_PAuth2 is implemented, FEAT_FPAC is not implemented, and stage 1 translation is disabled, then Arm recommends not setting the TCR_ELx.TnSZ values to indicate an address range that is smaller than the PA size:

- For some PAC values, if TCR_ELx.TnSZ is set to indicate an address range that is smaller than the PA size, the address generated by a failed PAC authentication might be an address within the PA size because the upper address bits, those between the PA size and the size indicated by the TCR_ELx.TnSZ field, are taken from the result of the authentication process.
- Accessing memory using such an address that has failed PAC authentication does not generate an Address size fault.
• The memory access is done using upper address bits, those between the PA size and the size indicated by the TCR_ELx.TnSZ field, taken from the result of the authentication process.

If the value in TCR_ELx.TnSZ is outside the permitted range, then one of the following constrained unpredictable results occur:
• The TCR_ELx.TnSZ value configures the bottom_PAC_bit value.
• The minimum permitted TCR_ELx.TnSZ value configures the bottom_PAC_bit value.
• The maximum permitted TCR_ELx.TnSZ value configures the bottom_PAC_bit value.

If a PE treats an out of range TCR_ELx.TnSZ value as the maximum permitted field value or the minimum permitted field value for all purposes except reading the field value, then that behavior also applies to determining bottom_PAC_bit.

D8.8.2 PAC generation and verification keys

For pointer authentication, all of the following 128-bit PAC keys are provided:
• Instruction address PAC key A, APIAKey_EL1, is the concatenation of the register values APIAKeyHi_EL1:APIAKeyLo_EL1.
• Instruction address PAC key B, APIBKey_EL1, is the concatenation of the register values APIBKeyHi_EL1:APIBKeyLo_EL1.
• Data address PAC key A, APDAKey_EL1, is the concatenation of the register values APDAKeyHi_EL1:APDAKeyLo_EL1.
• Data address PAC key B, APDBKey_EL1, is the concatenation of the register values APDBKeyHi_EL1:APDBKeyLo_EL1.
• Generic authentication PAC key A, APGAKey_EL1, is the concatenation of the register values APGAKeyHi_EL1:APGAKeyLo_EL1.

For an Exception level, all of the following bits in the SCTLR_ELx registers are used to enable PAC generation and validation:
• SCTLR_ELx.EnIA enables instruction address pointer authentication using the APIAKey_EL1 key.
• SCTLR_ELx.EnIB enables instruction address pointer authentication using the APIBKey_EL1 key.
• SCTLR_ELx.EnDA enables data address pointer authentication using the APDAKey_EL1 key.
• SCTLR_ELx.EnDB enables data address pointer authentication using the APDBKey_EL1 key.

All Exception levels use the same set of registers to hold PAC keys.

When switching between Exception levels, software is expected to apply all of the following:
• Software is expected to switch the PAC keys between Exception levels.
• Software is expected to not leave the values of the current keys present in memory, typically done by zeroing those locations after switching.

D8.8.3 PAC instructions

When a register is used as the target of an indirect branch instruction or a load instruction, all of the following instructions can be used to authenticate the register contents and prepare the register to be used:
• Instructions that insert a PAC field into the upper bits of the register.
• Instructions that extract the PAC field from the upper register bits and check that the value is correct.
• Instructions that remove the PAC field in the register and replace it with the extension bits, without verification.

An instruction that inserts a PAC into the upper bits of a register does all of the following:
• The bits used are the extension bits that do not hold valid address bits.
• The inserted PAC value is calculated from the value of the register and one other 64-bit value.
IKBGF An instruction that extracts the PAC from the upper register bits and checks that the value is correct does all of the following:

- The check is based on the value of the register and one other 64-bit value.
- When the value is correct, the PAC is replaced with the extension bits.
- When the value is incorrect, all of the following occur:
  - The PAC is replaced with the extension bits.
  - Two extension bits are set to a unique, fixed value.
  - When the register is used as an indirect branch target, a Translation fault is generated because the VA is not mapped.

IRZFZG Multiple versions of pointer authentication instructions support different use cases. Some instructions combine a pointer authentication operation with another operation.

RWNKCJ If PAC generation and validation is disabled, then the PACGA and XPAC* instructions are not affected and are always enabled.

RWDYDC If PAC generation and validation is disabled, then all of the following apply:

- Instructions that insert a PAC into the address in a register execute as NOPs.
- Instructions that authenticate a PAC execute as NOPs.
- If an instruction combines pointer authentication with another operation, then all of the following apply:
  - The pointer authentication does not occur.
  - The combined operation behaves the same as the equivalent instruction that does not do pointer authentication.

IRHMIV If PAC generation and validation is disabled, all of the following are examples of the behavior of instructions that combine pointer authentication with another operation:

- A RETAA instruction operates as a RET instruction.
- A LDRAA Xt, [Xn, #<simm10>! instruction operates as a LDR Xt, [Xn, #<simm10>:000]! instruction.

IZVGYK The PACGA instruction generates a 32-bit PAC from two 64-bit values and a generic key.

IZPGRZ The PACGA instruction can be used to protect small memory blocks. PACGA instructions can be chained to protect an arbitrary-sized memory block.

IPXWQC For pointer authentication instructions other than the PACGA instruction, the PAC is generated using one of the following IMPLEMENTATION DEFINED methods:

- If FEAT_PACQARMA5 is implemented, then ID_AA64ISAR1_EL1.APA is non-zero and the QARMA5 block cipher algorithm is used.
- If FEAT_PACIMP is implemented, then ID_AA64ISAR1_EL1.API is non-zero and an IMPLEMENTATION DEFINED algorithm is used.
- If FEAT_PACQARMA3 is implemented, then ID_AA64ISAR2_EL1.APA3 is non-zero and the QARMA3 block cipher algorithm is used.

IWXMCQ For the PACGA instruction, the PAC is generated using one of the following IMPLEMENTATION DEFINED methods:

- If FEAT_PACQARMA5 is implemented, then ID_AA64ISAR1_EL1.GPA is non-zero and the QARMA5 block cipher algorithm is used.
- If FEAT_PACIMP is implemented, then ID_AA64ISAR1_EL1.GPI is non-zero and an IMPLEMENTATION DEFINED algorithm is used.
- If FEAT_PACQARMA3 is implemented, then ID_AA64ISAR2_EL1.GPA3 is non-zero and the QARMA3 block cipher algorithm is used.

RZYPJV For the PACGA instruction, if the PAC is generated using an IMPLEMENTATION DEFINED algorithm, then all of the following are required:

- The IMPLEMENTATION DEFINED algorithm uses the same arguments as the ComputePAC() pseudocode function.
The AArch64 Virtual Memory System Architecture

D8.8 Pointer authentication

For a set of arguments passed to the IMPLEMENTATION DEFINED algorithm, the same result is produced by all PEs that an execution thread could migrate between.

For more information, see aarch64/functions/pac/computepac/ComputePAC on page J1-11116.

D8.8.4 Faulting on pointer authentication

All statements in this section require implementation of FEAT_FPAC.

If an instruction only authenticates a PAC, then when the PAC is incorrect, a Translation fault is generated.

If an instruction is a combined instruction that includes pointer authentication, then when the PAC is incorrect, one of the following IMPLEMENTATION DEFINED behaviors occur:

- A Translation fault is generated due to the authentication failure.
- The address is modified in a way that generates a Translation fault when the address is accessed.

If an instruction is a combined instruction that includes pointer authentication and FEAT_FPACCOMBINE is implemented, then when the PAC is incorrect, a Translation fault is generated due to the authentication failure.

When an authentication failure occurs at EL0, the exception is taken at one of the following:

- If HCR_EL2.TGE is 0, the exception is taken at EL1.
- If HCR_EL2.TGE is 1, the exception is taken at EL2.

If the current Exception level is not EL0, then when an authentication failure occurs, the exception is taken at the current Exception level.

When an exception is generated due to an authentication failure, the ESR_ELx.EC code is set to 0x1C.
D8.9 Virtualization Host Extensions

Virtualization Host Extensions provide enhanced support for a Type 2 virtualization solution, where there is a Host OS that is either more privileged than the hypervisor or is the hypervisor.

All statements in this chapter require implementation of FEAT_VHE.

If and only if an implementation includes EL2 using AArch64, then Virtualization Host Extensions, FEAT_VHE, can apply.

FEAT_VHE adds all of the following state:

- The HCR_EL2.E2H configuration bit.
- The CONTEXTIDR_EL2 register, which has the same format and contents as the CONTEXTIDR_EL1 register.
- The TTBR1_EL2 register, which has the same format and contents as the TTBR1_EL1 register.
- The CNTHV_CTL_EL2, CNTHV_CVAL_EL2, and CNTHV_TVAL_EL2 registers, which have the same format and contents as the CNTV_CTL_EL0, CNTV_CVAL_EL0, and CNTV_TVAL_EL0 registers, respectively.
- An EL2 virtual timer with all of the following properties:
  - It is accessed using the CNTHV_CTL_EL2, CNTHV_CVAL_EL2, and CNTHV_TVAL_EL2 registers.
  - The virtual offset is treated as 0.

D8.9.1 Behavior of HCR_EL2.E2H

HCR_EL2.E2H enables a configuration where a host operating system is running in EL2, and the host operating system applications are running in EL0. The host operating system might also manage guest virtual machines that run EL1&0, with stage 2 translation enabled and controlled by the host operating system.

If HCR_EL2.E2H is 0, then all of the following apply:

- Execution at EL2 uses the EL2 translation regime.
- All of the following effects apply to the contents of TTBR1_EL2:
  - The contents can be read by an MRS instruction and written by an MSR instruction.
  - For all other hardware operations, the contents are ignored.
- The Context ID matching breakpoint has all of the following properties:
  - It is disabled at EL2.
  - It uses the value of CONTEXTIDR_EL1 at EL0 and EL1.

If HCR_EL2.E2H is 1, then the translation regime controlled by TCR_EL2 is the EL2&0 translation regime.

All of the following are properties of the EL2&0 translation regime:

- The EL2&0 translation regime behaves the same as stage 1 in the EL1&0 translation regime and uses an upper address range translated by tables pointed to by TTBR1_EL2.
- TTBR0_EL2 translates the lower address range of the EL2&0 translation regime and is extended to have the same format and contents as TTBR0_EL1.
- The Privileged Access Never mechanism applies to data accesses from EL2.
- When CNTVCT_EL0 is read from EL2, the virtual offset is reported as 0.
- All of the following registers are redefined:
  - CNTHCTL_EL2.
  - CPTR_EL2.
  - TCR_EL2.
If the value of \texttt{HCR\_EL2}.E2H is 1, then all of the following effects to the Context ID apply:

- When executing at EL2, a Context ID matching breakpoint uses \texttt{CONTEXTIDR\_EL2}.
- Both VMID and VMID plus Context ID matching breakpoints do not match at EL2.

If \texttt{HCR\_EL2}.\{E2H, TGE\} is \{1, 0\}, then accesses from EL1 and EL0 use the EL1&0 translation regime.

If the value of \texttt{HCR\_EL2}.\{E2H, TGE\} is \{1, 1\}, then when executing at EL0, all of the following apply:

- The EL2&0 translation regime is used and accesses are treated as unprivileged.
- A Context ID matching breakpoint uses \texttt{CONTEXTIDR\_EL2}.
- The following timer registers, and their equivalent AArch32 registers, are redefined to access the associated EL2 register, rather than accessing the EL0 register:
  - \texttt{CNTP\_CTL\_EL0}.
  - \texttt{CNTP\_CVAL\_EL0}.
  - \texttt{CNTP\_TVAL\_EL0}.
  - \texttt{CNTV\_CTL\_EL0}.
  - \texttt{CNTV\_CVAL\_EL0}.
  - \texttt{CNTV\_TVAL\_EL0}.
- Both VMID and VMID plus Context ID matching breakpoints do not match at EL0.

If \texttt{HCR\_EL2}.\{E2H, TGE\} is \{1, 1\}, then when executing at EL2, all of the following apply:

- The EL2&0 translation regime is used.
- For permission and watchpoint checking, if the unprivileged instructions \texttt{LDTR}, \texttt{LDTRB}, \texttt{LDTRH}, \texttt{LDTRSB}, \texttt{LDTRSH}, \texttt{LDTRSW}, \texttt{STTR}, \texttt{STTRB}, and \texttt{STTRH} are executed, then the behavior is the same as executing at EL0.

If the value of \texttt{HCR\_EL2}.\{E2H, TGE\} is \{1, 1\}, then all of the following register effects apply:

- Except for the purpose of reading the value held in the register, some fields in \texttt{HCR\_EL2} and all fields in \texttt{HSTR\_EL2} are treated as effectively having a specific value.
- \texttt{SCTLR\_EL2} is redefined in all of the following ways:
  - Additional fields from \texttt{SCTLR\_EL1} are included.
  - The register applies to execution at EL0.
- When \texttt{CNTVCT\_EL0} is read from EL0 or EL2, the virtual offset is treated as 0.

If the value of \texttt{HCR\_EL2}.\{E2H, TGE\} is \{1, 1\}, then when an exception is taken from EL0 to EL2, the value of the \texttt{HCR\_EL2}.RW bit is ignored when determining the exception vector offset to use.

If the value of \texttt{HCR\_EL2}.\{E2H, TGE\} is \{1, 1\}, then the TLB maintenance and address translation instructions that apply to the EL1&0 translation regime are redefined to apply to the EL2&0 translation regime.

If the value of \texttt{HCR\_EL2}.\{E2H, TGE\} is \{1, 1\}, then when executing at EL2 or EL0, any physical interrupt that is configured to be taken at EL2 is subject to the \texttt{PSTATE}.\{D, A, I, F\} interrupt masks.

For more information, see \textit{The PSTATE debug mask bit, D} on page D1-4681 and \textit{Asynchronous exception masking} on page D1-4659.

If the value of \texttt{HCR\_EL2}.\{E2H, TGE\} is \{1, 1\}, then all of the following apply to EL1:

- Access from EL1 is not possible.
- \texttt{CPACR\_EL1} does not cause any instructions to be trapped to EL1.
- \texttt{CNTKCTL\_EL1} does not cause any instructions to be trapped to EL1, and the event stream caused by \texttt{CNTKCTL\_EL1} is disabled.

### D8.9.2 System and Special-purpose register redirection

If all of the conditions in the following list are true, then the System register mapping in the following table applies:

- The PE is executing at EL2.
• HCR_EL2.E2H is 1.

Table D8-57 System register redirection

<table>
<thead>
<tr>
<th>Specified EL1 System register</th>
<th>Equivalent EL2 System register accessed at EL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCTLR_EL1</td>
<td>SCTLR_EL2</td>
</tr>
<tr>
<td>CPACR_EL1</td>
<td>CPTR_EL2</td>
</tr>
<tr>
<td>TRFCR_EL1</td>
<td>TRFCR_EL2</td>
</tr>
<tr>
<td>TTBR0_EL1</td>
<td>TTBR0_EL2</td>
</tr>
<tr>
<td>TTBR1_EL1</td>
<td>TTBR1_EL2</td>
</tr>
<tr>
<td>TCR_EL1</td>
<td>TCR_EL2</td>
</tr>
<tr>
<td>AFSR0_EL1</td>
<td>AFSR0_EL2</td>
</tr>
<tr>
<td>AFSR1_EL1</td>
<td>AFSR1_EL2</td>
</tr>
<tr>
<td>ESR_EL1</td>
<td>ESR_EL2</td>
</tr>
<tr>
<td>FAR_EL1</td>
<td>FAR_EL2</td>
</tr>
<tr>
<td>MAIR_EL1</td>
<td>MAIR_EL2</td>
</tr>
<tr>
<td>AMAIR_EL1</td>
<td>AMAIR_EL2</td>
</tr>
<tr>
<td>VBAR_EL1</td>
<td>VBAR_EL2</td>
</tr>
<tr>
<td>CONTEXTIDR_EL1</td>
<td>CONTEXTIDR_EL2</td>
</tr>
<tr>
<td>CNTKCTL_EL1</td>
<td>CNTKCTL_EL2</td>
</tr>
</tbody>
</table>

If all of the conditions in the following list are true, then the System register mapping in the following table applies:

• The PE is executing at EL2.
• HCR_EL2.E2H is 1.
• The Effective value of SCR_EL3.NS is 1.

Table D8-58 Additional System register redirection when the Effective value of SCR_EL3.NS is 1

<table>
<thead>
<tr>
<th>Specified System register</th>
<th>Equivalent EL2 System register accessed at EL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNTP_TVAL_EL0</td>
<td>CNTHP_TVAL_EL2</td>
</tr>
<tr>
<td>CNTP_CTL_EL0</td>
<td>CNTHP_CTL_EL2</td>
</tr>
<tr>
<td>CNTP_CVAL_EL0</td>
<td>CNTHP_CVAL_EL2</td>
</tr>
<tr>
<td>CNTV_TVAL_EL0</td>
<td>CNTHV_TVAL_EL2</td>
</tr>
<tr>
<td>CNTV_CTL_EL0</td>
<td>CNTHV_CTL_EL2</td>
</tr>
<tr>
<td>CNTV_CVAL_EL0</td>
<td>CNTHV_CVAL_EL2</td>
</tr>
</tbody>
</table>

If all of the conditions in the following list are true, then the System register mapping in the following table applies:

• The PE is executing at EL2.
• HCR_EL2.E2H is 1.
• The Effective value of SCR_EL3.NS is 0.

Table D8-59 Additional System register redirection when the Effective value of SCR_EL3.NS is 0

<table>
<thead>
<tr>
<th>Specified EL1 System register</th>
<th>Equivalent EL2 System register accessed at EL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNTP_TVAL_EL0</td>
<td>CNTHPS_TVAL_EL2</td>
</tr>
<tr>
<td>CNTP_CTL_EL0</td>
<td>CNTHPS_CTL_EL2</td>
</tr>
<tr>
<td>CNTP_CVAL_EL0</td>
<td>CNTHPS_CVAL_EL2</td>
</tr>
<tr>
<td>CNTV_TVAL_EL0</td>
<td>CNTHVS_TVAL_EL2</td>
</tr>
<tr>
<td>CNTV_CTL_EL0</td>
<td>CNTHVS_CTL_EL2</td>
</tr>
<tr>
<td>CNTV_CVAL_EL0</td>
<td>CNTHVS_CVAL_EL2</td>
</tr>
</tbody>
</table>

If all of the conditions in the following list are true, then the Special-purpose register mapping in the following table applies:

• The PE is executing at EL2.
• HCR_EL2.E2H is 1.

Table D8-60 Special-purpose register redirection

<table>
<thead>
<tr>
<th>Specified EL1 Special-purpose register</th>
<th>Equivalent EL2 Special-purpose register accessed at EL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPSR_EL1</td>
<td>SPSR_EL2</td>
</tr>
<tr>
<td>ELR_EL1</td>
<td>ELR_EL2</td>
</tr>
</tbody>
</table>

For more information, see System and Special-purpose register aliasing.

D8.9.3 System and Special-purpose register aliasing

If all of the conditions in the following list are true, then the System register mapping in the following table applies. The aliases are UNDEFINED at EL1 and EL0.

• HCR_EL2.E2H is 1.
• The PE is executing at EL2 or EL3.

Table D8-61 System register aliases

<table>
<thead>
<tr>
<th>Alias mnemonic</th>
<th>EL1 or EL0 System register accessed at EL2 or EL3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCTLR_EL12</td>
<td>SCTLR_EL1</td>
</tr>
<tr>
<td>CPACR_EL12</td>
<td>CPACR_EL1</td>
</tr>
<tr>
<td>ZCR_EL12</td>
<td>ZCR_EL1</td>
</tr>
<tr>
<td>TRFCR_EL12</td>
<td>TRFCR_EL1</td>
</tr>
<tr>
<td>TTBR0_EL12</td>
<td>TTBR0_EL1</td>
</tr>
<tr>
<td>TTBR1_EL12</td>
<td>TTBR1_EL1</td>
</tr>
<tr>
<td>TCR_EL12</td>
<td>TCR_EL1</td>
</tr>
<tr>
<td>AFSR0_EL12</td>
<td>AFSR0_EL1</td>
</tr>
<tr>
<td>AFSR1_EL12</td>
<td>AFSR1_EL1</td>
</tr>
</tbody>
</table>
If all of the conditions in the following list are true, then the Special-purpose register mapping in the following table applies. The aliases are UNDEFINED at EL1 and EL0.

- **HCR_EL2.E2H** is 1.
- The PE is executing at EL2 or EL3.

<table>
<thead>
<tr>
<th>Alias mnemonic</th>
<th>EL1 or EL0 System register accessed at EL2 or EL3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESR_EL12</td>
<td>ESR_EL1</td>
</tr>
<tr>
<td>FAR_EL12</td>
<td>FAR_EL1</td>
</tr>
<tr>
<td>PMSCR_EL12</td>
<td>PMSCR_EL1</td>
</tr>
<tr>
<td>MAIR_EL12</td>
<td>MAIR_EL1</td>
</tr>
<tr>
<td>AMAIR_EL12</td>
<td>AMAIR_EL1</td>
</tr>
<tr>
<td>VBAR_EL12</td>
<td>VBAR_EL1</td>
</tr>
<tr>
<td>CONTEXTIDR_EL12</td>
<td>CONTEXTIDR_EL1</td>
</tr>
<tr>
<td>CNTKCTL_EL12</td>
<td>CNTKCTL_EL1</td>
</tr>
<tr>
<td>CNTP_TVAL_EL02</td>
<td>CNTP_TVAL_EL0</td>
</tr>
<tr>
<td>CNTP_CTL_EL02</td>
<td>CNTP_CTL_EL0</td>
</tr>
<tr>
<td>CNTP_CVAL_EL02</td>
<td>CNTP_CVAL_EL0</td>
</tr>
<tr>
<td>CNTV_TVAL_EL02</td>
<td>CNTV_TVAL_EL0</td>
</tr>
<tr>
<td>CNTV_CTL_EL02</td>
<td>CNTV_CTL_EL0</td>
</tr>
<tr>
<td>CNTV_CVAL_EL02</td>
<td>CNTV_CVAL_EL0</td>
</tr>
</tbody>
</table>

For more information, see **System and Special-purpose register redirection on page D8-5169.**
D8.10 Nested virtualization

Nested virtualization is an OPTIONAL feature that enables a Host hypervisor executing at EL2 to run a Guest hypervisor at EL1.

All statements in this section and subsections require implementation of FEAT_NV.

Nested virtualization adds the HCR_EL2.{NV, NV1, AT} fields.

An implementation is permitted to cache the HCR_EL2.{NV, NV1} fields in a TLB.

If a Guest hypervisor is run with HCR_EL2.E2H set to 0, then the Host hypervisor is required to set HCR_EL2.TVM to 1 and CPTER_EL2.TCPAC to 1 to trap any Guest hypervisor accesses to the EL1 System registers that would be accessed from any Guest OS running under the Guest hypervisor.

Nested virtualization does not modify either self-hosted debug or the Performance Monitors Extension.

Arm assumes that the Host hypervisor traps accesses to the Breakpoint registers and Performance Monitors registers to EL2, so that it can process any accesses to these registers made by a Guest hypervisor or by a Guest OS running under the Guest hypervisor.

D8.10.1 Behavior when HCR_EL2.NV is 1

If HCR_EL2.NV is 1 and the current Exception level is EL1, then all of the following apply:

- If a read access or write access from EL1 to one of the following System registers or Special-purpose registers occurs and that access is permitted at EL2, then that access is trapped to EL2 and reported using EC syndrome value 0x18 in ESR_EL2:
  - Any System register accessed using MRS or MSR named *_EL2, except SP_EL2.
  - Any System register accessed using MRS or MSR named *_EL12.
  - Any System register accessed using MRS or MSR named *_EL02.
  - Special-purpose registers SPSR_irq, SPSR_abt, SPSR_und, and SPSR_fiq, accessed using MRS or MSR.
  - Special-purpose register SP_EL1 accessed using the dedicated MRS or MSR instruction.

- If one of the following instructions is executed from EL1, then the instruction is trapped to EL2 and reported using EC syndrome value 0x18 in ESR_EL2:
  - Address Translation instructions that are accessible only from EL2 and above.
  - TLB maintenance instructions that are accessible only from EL2 and above.

- If the ERET, ERETAA, or ERETAB instruction is executed from EL1, then the instruction is trapped to EL2 and reported using EC syndrome value 0x1a in ESR_EL2.

- If EL3 is not implemented and the Effective value of HCR_EL2.TSC is 1, then when an SMC instruction is executed at EL1, all of the following apply:
  - The SMC instruction is trapped to EL2.
  - The exception is reported using EC syndrome value 0x17 in ESR_EL2.

If HCR_EL2.NV is 1 and the current Exception level is EL1, then EL1 read accesses to the CurrentEL register return a value of 0x2 in bits[3:2].

D8.10.2 Additional behavior when HCR_EL2.NV is 1 and HCR_EL2.NV1 is 0

If HCR_EL2.{NV, NV1} is {1, 0} and the current Exception level is EL1, then any exception taken from EL1 to EL1 causes SPSR_EL1.M[3:2] to be set to 0b10 and not 0b01.
D8.10.3 Additional behavior when HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1

If HCR_EL2.{NV, NV1} is {1, 1} and the current Exception level is EL1, then all of the following apply:

- If an access to one of the following registers occurs, then the access is trapped to EL2 and reported using EC syndrome value 0x18 in ESR_EL2:
  - An access to VBAR_EL1.
  - An access to ELR_EL1.
  - An access to SPSR_EL1.
  - If implemented, then an access from EL1 to SCXTNUM_EL1.
- For Block descriptors and Page descriptors in the EL1&0 translation regime, all of the following apply:
  - Block descriptor and Page descriptor bit[54] holds PXN, not UXN.
  - Block descriptor and Page descriptor bit[53] is RES0.
  - Block descriptor and Page descriptor bit[6], AP[1], is treated as 0 regardless of the actual value.
- For Table descriptors in the EL1&0 translation regime, if hierarchical permissions are enabled, then all of the following apply:
  - Table descriptor bit[61], APTable[0], is treated as 0 regardless of the actual value.
  - Table descriptor bit[60] holds PXNTable, not UXNTable.
  - Table descriptor bit[59] is RES0.
- PSTATE.PAN is treated as 0 for all purposes except reading the value of the bit.
- If the LDTR* or the STTR* instructions are executed, then all of the following apply:
  - The LDTR* instructions behave the same as the corresponding LDR* instructions.
  - The STTR* instructions behave the same as the corresponding STR* instructions.

Arm expects software to clear the HCR_EL2.NV1 bit to 0 before permitting execution at EL0.

D8.10.4 Behavior when HCR_EL2.NV is 0 and HCR_EL2.NV1 is 1

If HCR_EL2.{NV, NV1} is {0, 1} and the current Exception level is EL1, then one of the following constrained unpredictable behaviors apply:

- For all purposes other than reading back the value of the HCR_EL2.NV bit, the implementation behaves as if HCR_EL2.{NV, NV1} is {1, 1}.
- For all purposes other than reading back the value of the HCR_EL2.NV1 bit, the implementation behaves as if HCR_EL2.{NV, NV1} is {0, 0}.
- The implementation behaves as defined when HCR_EL2.NV is 0, with HCR_EL2.NV1 set to 1 having the effect of causing accesses to VBAR_EL1, ELR_EL1, and SPSR_EL1 from EL1 to be trapped to EL2.

D8.10.5 Effect of HCR_EL2.AT

If HCR_EL2.AT is 1, then all of the following are trapped to EL2 and reported using EC syndrome value 0x18 in ESR_EL2:

- Executing AT S1E0R from EL1.
- Executing AT S1E0M from EL1.
- Executing AT S1E1R from EL1.
- Executing AT S1E1M from EL1.
- Executing AT S1E1RP from EL1.
- Executing AT S1E1WP from EL1.
Enhanced support for nested virtualization

Enhanced support for nested virtualization provides a mechanism for hardware to transform reads and writes from System registers into reads and writes from memory.

All statements in this section and subsections require implementation of FEAT_NV2.

If HCR_EL2.NV2 is 1 and the current Exception level is EL1, then all of the following apply:

- Accesses to EL2 registers are redirected to EL1 registers.
- Accesses to System registers are transformed to memory accesses.

If HCR_EL2.NV2 is 0, then the behavior of HCR_EL2.{NV, NV1} is unchanged.

Redirection of register accesses from EL2 to EL1

If HCR_EL2.{NV, NV2} is {1, 1} and the current Exception level is EL1, then accesses to all of the following EL2 Special-purpose registers are redirected to the corresponding EL1 register:

- An access from EL1 to SPSR_EL2 is redirected to SPSR_EL1.
- An access from EL1 to ELR_EL2 is redirected to ELR_EL1.

If HCR_EL2.{NV, NV2} is {1, 1} and the current Exception level is EL1, then accesses to all of the following EL2 System registers are redirected to the corresponding EL1 register:

- An access from EL1 to ESR_EL2 is redirected to ESR_EL1.
- An access from EL1 to FAR_EL2 is redirected to FAR_EL1.

For more information, see:

- \textit{op0==0b11, Moves to and from Special-purpose registers on page C5-668.}
- \textit{Instructions for accessing non-debug System registers on page D16-5523.}

Loads and stores generated by transforming register accesses

If HCR_EL2.{NV, NV2} is {1, 1} and the current Exception level is EL1, then when an MRS or MSR instruction is executed to access certain registers, all of the following apply:

- The instruction is treated as an instruction executed at EL1.
- The register accesses do not generate an exception to EL2.
- The register accesses are transformed to memory accesses.
- The memory accesses are treated as EL2 accesses.

When register accesses are transformed to memory accesses, all of the following information is used to form the memory address:

- A base address that is stored in the VNCR_EL2 register.
- For a register that is redirected to memory, a unique memory offset value.
- The memory address is a combination of the base address and the memory offset according to the formula \( \text{SignExtend}(\text{VNCR_EL2.BADDR} : \text{Offset<11:0>}, 64) \).
When register accesses are transformed to memory accesses, the following table shows the unique memory offset that is assigned to the register.

<table>
<thead>
<tr>
<th>Register access if HCR_EL2.NV1 is 0</th>
<th>Register access if HCR_EL2.NV1 is 1</th>
<th>Memory offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>VTTBR_EL2</td>
<td>VTTBR_EL2</td>
<td>0x20</td>
</tr>
<tr>
<td>VTTBR_EL2</td>
<td>VTTBR_EL2</td>
<td>0x20</td>
</tr>
<tr>
<td>VTCR_EL2</td>
<td>VTCR_EL2</td>
<td>0x40</td>
</tr>
<tr>
<td>VSTCR_EL2</td>
<td>VSTCR_EL2</td>
<td>0x48</td>
</tr>
<tr>
<td>VMPIDR_EL2</td>
<td>VMPIDR_EL2</td>
<td>0x50</td>
</tr>
<tr>
<td>CNTVOFF_EL2</td>
<td>CNTVOFF_EL2</td>
<td>0x60</td>
</tr>
<tr>
<td>HCR_EL2</td>
<td>HCR_EL2</td>
<td>0x78</td>
</tr>
<tr>
<td>HSTR_EL2</td>
<td>HSTR_EL2</td>
<td>0x80</td>
</tr>
<tr>
<td>VPIDR_EL2</td>
<td>VPIDR_EL2</td>
<td>0x88</td>
</tr>
<tr>
<td>TPIDR_EL2</td>
<td>TPIDR_EL2</td>
<td>0x90</td>
</tr>
<tr>
<td>HCRX_EL2</td>
<td>HCRX_EL2</td>
<td>0xA0</td>
</tr>
<tr>
<td>VNCR_EL2</td>
<td>VNCR_EL2</td>
<td>0xB0</td>
</tr>
<tr>
<td>CPACR_EL12</td>
<td>CPACR_EL1</td>
<td>0x100</td>
</tr>
<tr>
<td>CONTEXTIDR_EL12</td>
<td>CONTEXTIDR_EL1</td>
<td>0x108</td>
</tr>
<tr>
<td>SCTLR_EL12</td>
<td>SCTLR_EL1</td>
<td>0x110</td>
</tr>
<tr>
<td>ACTLR_EL1</td>
<td>ACTLR_EL1</td>
<td>0x118</td>
</tr>
<tr>
<td>TCR_EL12</td>
<td>TCR_EL1</td>
<td>0x120</td>
</tr>
<tr>
<td>AFSR0_EL12</td>
<td>AFSR0_EL1</td>
<td>0x128</td>
</tr>
<tr>
<td>AFSR1_EL12</td>
<td>AFSR1_EL1</td>
<td>0x130</td>
</tr>
<tr>
<td>ESR_EL12</td>
<td>ESR_EL1</td>
<td>0x138</td>
</tr>
<tr>
<td>MAIR_EL12</td>
<td>MAIR_EL1</td>
<td>0x140</td>
</tr>
<tr>
<td>AMAIR_EL12</td>
<td>AMAIR_EL1</td>
<td>0x148</td>
</tr>
<tr>
<td>MDSCR_EL1</td>
<td>MDSCR_EL1</td>
<td>0x158</td>
</tr>
<tr>
<td>SPSR_EL12</td>
<td>SPSR_EL1</td>
<td>0x160</td>
</tr>
<tr>
<td>CNTV_CVAL_EL02</td>
<td>CNTV_CVAL_EL0</td>
<td>0x168</td>
</tr>
<tr>
<td>CNTV_CTL_EL02</td>
<td>CNTV_CTL_EL0</td>
<td>0x170</td>
</tr>
<tr>
<td>CNTP_CVAL_EL02</td>
<td>CNTP_CVAL_EL0</td>
<td>0x178</td>
</tr>
<tr>
<td>CNTP_CTL_EL02</td>
<td>CNTP_CTL_EL0</td>
<td>0x180</td>
</tr>
<tr>
<td>SCXTNUM_EL12</td>
<td>SCXTNUM_EL1</td>
<td>0x188</td>
</tr>
<tr>
<td>TFSR_EL12</td>
<td>TFSR_EL1</td>
<td>0x190</td>
</tr>
<tr>
<td>Register access if HCR_EL2.NV1 is 0</td>
<td>Register access if HCR_EL2.NV1 is 1</td>
<td>Memory offset</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>CNTPOFF_EL2</td>
<td>CNTPOFF_EL2</td>
<td>0x1A8</td>
</tr>
<tr>
<td>HFGTR_EL2</td>
<td>HFGTR_EL2</td>
<td>0x1B8</td>
</tr>
<tr>
<td>HFGWTR_EL2</td>
<td>HFGWTR_EL2</td>
<td>0x1C0</td>
</tr>
<tr>
<td>HFGITR_EL2</td>
<td>HFGITR_EL2</td>
<td>0x1C8</td>
</tr>
<tr>
<td>HDFGRTR_EL2</td>
<td>HDFGRTR_EL2</td>
<td>0x1D0</td>
</tr>
<tr>
<td>HDFGWTR_EL2</td>
<td>HDFGWTR_EL2</td>
<td>0x1D8</td>
</tr>
<tr>
<td>ZCR_EL1</td>
<td>ZCR_EL1</td>
<td>0x1E0</td>
</tr>
<tr>
<td>HAFGRTR_EL2</td>
<td>HAFGRTR_EL2</td>
<td>0x1E8</td>
</tr>
<tr>
<td>TTBR0_EL12</td>
<td>TTBR0_EL1</td>
<td>0x200</td>
</tr>
<tr>
<td>TTBR1_EL12</td>
<td>TTBR1_EL1</td>
<td>0x210</td>
</tr>
<tr>
<td>FAR_EL12</td>
<td>FAR_EL1</td>
<td>0x220</td>
</tr>
<tr>
<td>ELR_EL12</td>
<td>ELR_EL1</td>
<td>0x230</td>
</tr>
<tr>
<td>SP_EL1</td>
<td>SP_EL1</td>
<td>0x240</td>
</tr>
<tr>
<td>VBAR_EL12</td>
<td>VBAR_EL1</td>
<td>0x250</td>
</tr>
<tr>
<td>ICH_LR&lt;n&gt;_EL2</td>
<td>ICH_LR&lt;n&gt;_EL2</td>
<td>0x400+8*n</td>
</tr>
<tr>
<td>ICH_AP0R&lt;n&gt;_EL2</td>
<td>ICH_AP0R&lt;n&gt;_EL2</td>
<td>0x480+8*n</td>
</tr>
<tr>
<td>ICH_AP1R&lt;n&gt;_EL2</td>
<td>ICH_AP1R&lt;n&gt;_EL2</td>
<td>0x4A0+8*n</td>
</tr>
<tr>
<td>ICH_HCR_EL2</td>
<td>ICH_HCR_EL2</td>
<td>0x4C0</td>
</tr>
<tr>
<td>ICH_VMCR_EL2</td>
<td>ICH_VMCR_EL2</td>
<td>0x4C8</td>
</tr>
<tr>
<td>VDISR_EL2</td>
<td>VDISR_EL2</td>
<td>0x500</td>
</tr>
<tr>
<td>VSESER_EL2</td>
<td>VSESER_EL2</td>
<td>0x508</td>
</tr>
<tr>
<td>PMBLIMITR_EL1</td>
<td>PMBLIMITR_EL1</td>
<td>0x800</td>
</tr>
<tr>
<td>PMBPTR_EL1</td>
<td>PMBPTR_EL1</td>
<td>0x810</td>
</tr>
<tr>
<td>PMBSR_EL1</td>
<td>PMBSR_EL1</td>
<td>0x820</td>
</tr>
<tr>
<td>PMSCR_EL12</td>
<td>PMSCR_EL1</td>
<td>0x828</td>
</tr>
<tr>
<td>PMSEVFR_EL1</td>
<td>PMSEVFR_EL1</td>
<td>0x830</td>
</tr>
<tr>
<td>PMSCR_EL1</td>
<td>PMSCR_EL1</td>
<td>0x838</td>
</tr>
<tr>
<td>PMSIRR_EL1</td>
<td>PMSIRR_EL1</td>
<td>0x840</td>
</tr>
<tr>
<td>PMSLATFR_EL1</td>
<td>PMSLATFR_EL1</td>
<td>0x848</td>
</tr>
<tr>
<td>TRFCR_EL12</td>
<td>TRFCR_EL1</td>
<td>0x880</td>
</tr>
<tr>
<td>AMEVCNTVOFF0&lt;n&gt;_EL2</td>
<td>AMEVCNTVOFF0&lt;n&gt;_EL2</td>
<td>0xA00+8*n</td>
</tr>
<tr>
<td>AMEVCNTVOFF1&lt;n&gt;_EL2</td>
<td>AMEVCNTVOFF1&lt;n&gt;_EL2</td>
<td>0xA80+8*n</td>
</tr>
</tbody>
</table>
Unallocated memory offset values up to but not including \(0x1000\) are reserved.

When a register access instruction targets a register that is not implemented, the PE treats access to that register as unallocated.

When a System register access instruction is trapped by either or both of HCR_EL2.\{NV, NV1\}, then the instruction is transformed into a memory access instruction instead of creating a trap.

When a System register access instruction is not trapped by either or both of HCR_EL2.\{NV, NV1\}, then the trap is subject to the exception prioritization rules.

Accesses to all of the following registers that affect hypervisor execution by controlling the event stream are not transformed into memory accesses:

- \(\text{CNTHCTL}_\text{EL2}\).
- If HCR_EL2.NV1 is 0, then CNTKCTL_EL12.
- If HCR_EL2.NV1 is 1, then CNTKCTL_EL1.

When a System register access is transformed into a memory access, the memory access has all of the following properties:

- The memory access is translated by one of the following:
  - If HCR_EL2.E2H is 0, then the EL2 translation regime.
  - If HCR_EL2.E2H is 1, then the EL2&0 translation regime.
- The endianness of the memory access is defined by SCTLR_EL2.EE.
- The memory access is 64-bit single-copy atomic aligned to 64 bits.
- The memory access does not have Acquire or Release semantics.
- If there is no context synchronizing operation between the register access and a load or store instruction accessing the address of the transformed memory access, then the transformed memory access can be reordered with respect to any reads or writes at EL1 caused by load or store instructions to the same address.
- The memory access behaves as if PSTATE.PAN is 0.
- For fields in a transformed System register that are defined to be RES0 or RES1, the memory access does not check that the fields are set correctly, nor are they forced to their dedicated value.

For more information, see Prioritization of Synchronous exceptions taken to AArch64 state on page D1-4650.

Exceptions from transformed register accesses

If HCR_EL2.\{NV, NV2\} is \{1, 1\}, then any exception taken from EL1 and taken to EL1 causes the SPSR_EL1.M[3:2] to be set to \(0b10\) and not \(0b01\).

If a System register access is transformed to a memory access, then when the memory access generates a Data Abort, the resulting fault has all of the following properties:

- The fault is taken to EL2, using the standard vector offset for exceptions from EL1 to EL2.
- The fault is reported as a Data Abort from the current Exception level with the ESR_EL2.EC code \(0x25\).
- FAR_EL2 is updated to hold the faulting address.

If a System register access is transformed to a memory access, then when the memory access generates a synchronous External abort and External aborts are not configured to be taken to EL3, the resulting fault has all of the following properties:

- The fault is taken to EL2, using the standard vector offset for exceptions from EL1 to EL2.
- The fault is reported as a Data Abort from the current Exception level with the ESR_EL2.EC code \(0x25\).

The VNCR field in ESR_EL2 and ESR_EL3 identifies whether the fault came from use of VNCR_EL2 by EL1.

For more information, see ISS encoding for an exception from a Data Abort on page D17-5685.
Interaction with self-hosted and External debug

When a System register access is transformed into a memory access, all of the following operations treat the instruction as being executed at EL1:

- PMU events filtered by Exception level.
- For trace or Statistical Profiling, instructions filtered by Exception level.
- Checking the instruction address against breakpoint registers or trace resources.

When a System register access is transformed into a memory access, all of the following operations treat the memory access as being executed at EL2:

- The memory access is checked against the watchpoint registers.
- The memory address is recorded in a Statistical Profiling record.

If all of the following apply, then it is CONSTRANDED UNPREDICTABLE whether there is a watchpoint match:

- A System register access is transformed into a memory access.
- The memory access matches an EL2 access in the watchpoint registers.
- A watchpoint is linked to a context-aware breakpoint that is programmed to match the value held in CONTEXTIDR_EL1 or VMID.

If all of the following apply, then a watchpoint match generates a Watchpoint debug event:

- A System register access is transformed into a memory access.
- EDSCR.HDE is 1.
- Halting is enabled.

If all of the following apply, then a watchpoint match generates a Watchpoint exception:

- A System register access is transformed into a memory access.
- EDSCR.HDE is 0.
- Debug exceptions are enabled at EL2.

If a System register access is transformed to a memory access, then when a watchpoint match generates a Watchpoint exception, the resulting exception has all of the following properties:

- The exception is taken to EL2.
- The fault is reported as a Watchpoint from the current Exception level with the ESR_EL2.EC code 0x35.
- FAR_EL2 is updated to hold the cachepointed address.

If a System register access is transformed to a memory access, then when a Watchpoint exception is generated, the VCNCR field in ESR_EL2 identifies whether the Watchpoint exception came from use of VNCR_EL2 by EL1.

If a System register access is transformed to a memory access, then the resulting loads and stores are treated by the Performance Monitors as Memory-read operations and Memory-write operations.

If a System register access is transformed to a memory access, then when the Statistical Profiling Unit selects the instruction generating the memory access for profiling, it records the operation as a Load/Store operation.

If a System register access is transformed to a memory access, then when the Statistical Profiling Unit selects the instruction generating the memory access for profiling and Statistical Profiling is disabled at EL2, the VA for the memory access is not recorded.
D8.11 Memory aborts

**D8.11 Memory aborts**

For a VMSAv8-64 implementation, all of the following abort mechanisms can cause the PE to take an exception due to a failed memory access:

- A *Debug exception* generated by the self-hosted debug system. For more information, see Chapter D2 AArch64 Self-hosted Debug.
- An *Alignment fault* generated by a memory access that is not aligned as required by the operation. For more information, see Alignment support on page B2-189.
- An *MMU fault* generated by the fault checking sequence in the appropriate translation regime. For more information, see MMU fault types.
- An *External abort* generated by the memory system and not by self-hosted debug, alignment checking, or MMU checking. For more information, see External aborts on page D7-5072.

For AArch64 state, MMU faults are synchronous exceptions that are reported as one of the following:

- Data Aborts due to data memory access faults.
- Instruction Aborts due to instruction fetch faults.

For more information, see Synchronous exception types on page D1-4646.

In some cases, an MMU fault generated as part of an instruction execution is reported in PAR_EL1 rather than as a synchronous abort.

When an MMU fault is generated and reported as an Instruction Abort or Data Abort, the Exception level that the exception is taken to depends on the translation regime and translation stage that caused the fault.

When a memory access abort generates an exception, all of the following registers are used to record context information:

- A Fault Address Register, FAR_ELx.
- An Exception Syndrome Register, ESR_ELx.

When an MMU fault is generated, the fault context saved in the ESR_ELx at the Exception level that the fault is taken to depends on all of the following:

- Whether the fault is reported as an Instruction Abort or as a Data Abort.
- Whether the exception is taken from the same or a lower Exception level.

For an Instruction Abort, only the following two exceptions have a higher priority in the exception priority hierarchy:

- Software Step exception.
- PC alignment fault exception.

For a Data Abort, only Watchpoint exceptions have a lower priority in the exception priority hierarchy.

### D8.11.1 MMU fault types

All of the following MMU fault types are supported, and there are no other MMU fault types:

- Alignment fault on a data access. For more information, see Alignment support on page B2-189.
- Translation fault.
- Address size fault.
- Synchronous External abort on a translation table walk.
- Access flag fault.
- Permission fault.
- TLB conflict abort.

When an attempt to access Device memory generates an MMU fault, no memory access occurs.
RTQDJS If the memory attributes cannot be determined and the resultant memory region could be Device memory, then when an attempt to access that region generates an MMU fault, no memory access occurs.

**Translation fault**

RDYNQ If descriptor bit[0] is 0 in any translation table entry, the descriptor is invalid. When a translation table lookup returns an invalid descriptor, a Translation fault is generated and descriptor bits[63:1] are ignored.

RKMJZ If descriptor bit[1] is 0 at a lookup level that does not support Block descriptors, then the descriptor encoding is reserved. When a translation table lookup returns that descriptor, a Translation fault is generated.

RFDQJL If FEAT_BBM is implemented and supported at level 1 or higher, and the Block descriptor has the nT bit set, then the implementation is permitted to generate a Translation fault.

IRSSLQ For any translation lookup level, when a Translation fault is generated, the reported fault code indicates the lookup level.

RVZZSZ When one or more of the following apply, a level 0 Translation fault is generated on the relevant translation stage:
- The IA does not map onto a TTBR_ELx address range.
- If the IA maps onto the TTBR0_ELx address range and the IA contains any one bits above the configured IA size as determined by TCR_ELx.T0SZ.
- If the IA maps onto the TTBR1_ELx address range and the IA contains any zero bits above the configured IA size as determined by TCR_ELx.T1SZ.
- When a TLB miss occurs, the corresponding TCR_ELx.EPDn field prevents a translation table walk using TTBRn_ELx.
- When FEAT_E0PD is implemented, the corresponding TCR_ELx.E0PDn field prevents unprivileged access to an address translated by TTBRn_ELx.
- When FEAT_SVE is implemented, the corresponding TCR_ELx.NFDy field prevents non-faulting unprivileged accesses to an address translated by TTBRy_ELx.

RMLNTS When a Translation fault is generated, the translation table entry causing the fault is not cached in a TLB.

IXFTPJ When a Translation fault is generated, the fault handler does not have to perform TLB maintenance to remove the faulting entry.

IPXJFV For more information, see *MMU faults generated by cache maintenance operations* on page D8-5185.

**Address size fault**

RCVSBH For a translation stage, if an address in one of the following has nonzero address bits above the configured OA size, then an Address size fault is generated:
- The TTBR_ELx used in the translation.
- A translation table entry.
- The translation OA.

RFTZHB An Address size fault can be generated at any translation lookup level.

RWDZFC For the 64KB translation granule, if FEAT_LPA is implemented and the implementation does not support 52-bit PAs, then all of the following apply:
- It is IMPLEMENTATION DEFINED whether nonzero address bits above the OA size generate an Address size fault.
- For this case, it is deprecated to not generate an Address size fault.

RBFHQH When an Address size fault is generated, the reported fault code indicates one of the following:
- If the fault was generated due to the TTBR_ELx used in the translation having nonzero address bits above the OA size, then a fault at level 0.
• If the fault was not generated due to the TTBR_ELx used in the translation, then the lookup level at which the fault occurred.

When an Address size fault is generated, the translation table entry causing the fault is not cached in a TLB.

When an Address size fault is generated, the fault handler does not have to perform TLB maintenance to remove the faulting entry.

An implementation is required to ensure that the Contiguous bit does not prevent generation of Address size faults. For more information, see Architectural guarantees when the Contiguous bit is misprogrammed on page D8-5159.

For more information, see:
• MMU faults generated by cache maintenance operations on page D8-5185.
• Output address size configuration on page D8-5088.

External abort on a translation table walk

If an External abort on a translation table walk is generated, then it is reported in one of the following ways:
• If the External abort is synchronous, then it is reported as one of the following:
  — If the translation table walk is due to an instruction fetch, then a synchronous Instruction Abort exception.
  — If the translation table walk is due to a data access, an address translation instruction, or a cache maintenance instruction operating by VA, then a synchronous Data Abort exception.
• If the External abort is asynchronous, then it is reported using the SError interrupt exception.

A TLB or intermediate TLB caching structure might support:
• An arbitrary number of translation table lookup levels.
• One or more translation stages that might not correspond to an address translation lookup stage.

If a synchronous External abort is generated due to a TLB or intermediate TLB caching structure, including parity or ECC errors, then all of the following are permitted:
• If the PE cannot precisely determine the translation stage at which the error occurred, then it is reported and prioritized as a stage 1 fault.
• If the PE cannot precisely determine the lookup level at which the error occurred, then the lookup level is reported and prioritized as one of the following:
  — The lowest-numbered lookup level that could have caused the error.
  — If the PE cannot determine any information about the lookup level, then level 0.

Access flag fault

For an implementation that does not manage the AF in hardware, when and only when a non-speculative access is made to an address and an attempt is made to translate that address using a descriptor with an AF of 0, an Access flag fault is generated.

For a translation lookup of a Block descriptor or Page descriptor, when an Access flag fault is generated, the reported fault code indicates the lookup level.

When an Access flag fault is generated, the translation table entry causing the fault is not cached in a TLB.

When an Access flag fault is generated, the fault handler does not have to perform TLB maintenance to remove the faulting entry.

For more information, see:
• MMU faults generated by cache maintenance operations on page D8-5185.
• Software management of the Access flag on page D8-5143.
Permission fault

When a Permission fault is generated, the reported fault code indicates the lookup level of the Block or Page descriptor used in the translation, even if the lack of permission was caused by hierarchical permission controls.

A translation table entry that generates a Permission fault is permitted to be cached in a TLB.

If software updates a stage 1 or stage 2 translation table due to a Permission fault, then the software is required to invalidate the appropriate TLB entry to prevent stale information in a TLB from being used by a subsequent memory access.

For more information, see:
- MMU faults generated by cache maintenance operations on page D8-5185.
- Memory access control on page D8-5132.
- Stage 2 fault on a stage 1 translation table walk on page D8-5189.

TLB conflict abort

When a TLB has not been properly invalidated, such as when architecturally required TLB invalidation is not done, an address lookup might hit multiple TLB entries.

When a lookup address hits multiple TLB entries, it is IMPLEMENTATION DEFINED whether a TLB conflict abort is generated.

When a TLB conflict abort is generated, it is reported as one of the following:
- On an instruction fetch it is reported as an Instruction Abort.
- On a data access or a cache management instruction, it is reported as a Data Abort.
- For an AT S1E0* or AT S1E1* instruction executed from EL1, when a TLB conflict abort is generated on a stage 2 translation, it is reported as a Data Abort.

For an AT instruction other than an AT S1E0* or AT S1E1* instruction executed from EL1, a TLB conflict does not generate an abort.

When a TLB conflict abort is generated, it is IMPLEMENTATION DEFINED whether it is a stage 1 abort or a stage 2 abort.

If EL2 is enabled and stage 2 of the EL1&0 translation regime is disabled, then a stage 2 abort cannot be generated.

When a TLB conflict abort is generated, all of the following are reported:
- A 0b110000 Fault status code.
- The lookup address that caused the fault.

If an address matches multiple entries in a TLB and does not generate a TLB conflict abort, then all of the following apply:
- The resulting behavior is CONSTRAINED UNPREDICTABLE.
- The CONSTRAINED UNPREDICTABLE behavior cannot permit access to memory regions with permissions or attributes that would not be possible in the current Security state at the current Exception level.

For more information, see CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values on page K1-11585.

D8.11.2 MMU faults generated by address translation instructions

An address translation instruction uses the translation system and can generate all of the following faults:
- Translation fault.
- Address size fault.
- Synchronous External abort on a translation table walk.
- Access flag fault.
- Permission fault.
- TLB conflict abort.
- When translating using the AArch32 translation systems, a domain fault.

When an address translation instruction generates a fault and when none of the following generated the fault, the fault is not taken as a synchronous exception, and PAR_EL1 is populated with PAR_EL1.F == 1 and the fault syndrome information:
- A synchronous External abort during a translation table walk.
- An AT S1E0* or AT S1E1* instruction executed from EL1 generates a stage 2 fault on a memory access during a translation table walk.

For more information, see Exceptions to reporting the fault in PAR_EL1.

If an address translation instruction requires two address translation stages, then a fault can be generated from either stage 1 or stage 2.

If an address translation instruction requires two address translation stages in the EL1&0 translation regime, then a fault can be generated on the stage 2 translation of an address accessed as part of the stage 1 translation table walk.

For more information, see Stage 2 fault on a stage 1 translation table walk on page D8-5189.

If FEAT_PAN2 is implemented, then the <pan> component of the AT instruction <operation> determines whether the PSTATE.PAN value is considered when determining permissions.

Exceptions to reporting the fault in PAR_EL1

For an address translation instruction executed at a particular Exception level, ELx, if a synchronous External abort is generated on a stage 1 translation table walk, then the Data Abort exception is taken to the Exception level to which a synchronous External abort on a stage 1 translation table walk for a memory access from ELx would be taken.

When an address translation instruction executed at EL3 generates a synchronous External abort on a stage 2 translation table walk, a Data Abort exception is taken to EL3.

For an address translation instruction executed at the EL2 or EL1 Exception level, if a synchronous External abort is generated on a stage 2 translation table walk, then the Data Abort exception is taken to the Exception level to which a synchronous External abort on a stage 2 translation table walk for a memory access from that same EL2 or EL1 Exception level would be taken.

When an address translation instruction causes a synchronous External abort that is reported as an exception, all of the following apply:
- PAR_EL1 is UNKNOWN.
- The ESR_ELx of the target Exception level indicates that the fault was generated by a translation table walk due to a cache maintenance operation or address translation instruction.
- The FAR_ELx of the target Exception level holds the VA supplied to the address translation instruction.

When an AT S1E0* or AT S1E1* instruction is executed, all of the following can generate a stage 2 fault on a memory access during a translation table walk:
- Stage 2 Translation fault.
- Stage 2 Address size fault.
- Synchronous External abort on a stage 2 translation table walk.
- Stage 2 Access fault.
- Stage 2 Permission fault.
- A TLB conflict abort that takes an IMPLEMENTATION DEFINED choice of reporting as a stage 2 abort.
When an AT \texttt{S1E0} or AT \texttt{S1E1} instruction executed from EL1 generates a stage 2 fault on a memory access during a translation table walk, one of the following occurs:

- If the fault is a synchronous External abort on a stage 2 translation table walk and \texttt{SCR_EL3.EA} is 1, then a synchronous External abort on a stage 2 translation table walk is taken to EL3.
- The fault is taken as an exception to EL2 and all of the following apply:
  - \texttt{PAR_EL1} is \texttt{UNKNOWN}.
  - \texttt{ESR_EL2} indicates that a cache maintenance instruction faulted during a translation table walk.
  - \texttt{HPFAR_EL2} holds the IPA that faulted.
  - \texttt{FAR_EL2} holds the VA supplied to the address translation instruction.

For more information, see \textit{Stage 2 fault on a stage 1 translation table walk} on page D8-5189.

### D8.11.3 MMU faults generated by cache maintenance operations

There are no alignment requirements in cache maintenance instructions and execution of these instructions cannot generate an Alignment fault.

If the Point of Coherency is before any cache level, then it is \texttt{IMPLEMENTATION DEFINED} whether the execution of any cache clean instruction, or clean and invalidate instruction, that operates by VA to the Point of Coherency, can generate any MMU fault.

If the Point of Unification is before any data cache level, then it is \texttt{IMPLEMENTATION DEFINED} whether the execution of any data or unified cache clean instruction, or clean and invalidate instruction, that operates by VA to the Point of Unification, can generate any MMU fault.

It is \texttt{IMPLEMENTATION DEFINED} whether the execution of any cache maintenance instruction by VA can generate an Access flag fault.

If \texttt{SCTLR_ELx.UCI} is 1, enabling EL0 execution of the data cache maintenance instructions that operate by VA, then when the instruction is executed at EL0 to a location that does not have read permission at EL0, a Permission fault can be generated.

If \texttt{SCTLR_ELx.UCI} is 1, enabling EL0 execution of the \texttt{IC IVAU} instruction, and if the instruction is executed at EL0 to a location that does not have read permission at EL0, then it is \texttt{IMPLEMENTATION DEFINED} whether a Permission fault is generated.

If the Effective value of \texttt{SCTLR_EL1.CMOV} is 1, then when executing a \texttt{IC IVAU}, \texttt{DC CVAC}, \texttt{DC CIGVAC}, or \texttt{DC CIGDVAC} instruction at EL0 that has stage 1 read permission, but does not have stage 1 write permission, a stage 1 Permission fault is generated.

If the Effective value of \texttt{HCRX_EL2.CMOV} is 1, then when executing a \texttt{IC IVAU}, \texttt{DC CVAC}, \texttt{DC CIGVAC}, or \texttt{DC CIGDVAC} instruction at EL1 and EL0 that has stage 2 read permission, but does have stage 2 write permission, a stage 2 Permission fault is generated.

If an \texttt{IC IVAU}, \texttt{DC CVAC}, \texttt{DC CIGVAC}, or \texttt{DC CIGDVAC} instruction is implemented as a NOP, then it is \texttt{IMPLEMENTATION DEFINED} whether the instruction generates a stage 1 or stage 2 Permission fault when it does not have read and write permission.

When an implementation can generate MMU faults on execution of cache maintenance operations, Permission faults are generated according to the type of cache maintenance operation and system register configuration.

If a stage 2 translation of a stage 1 translation table walk is done as part of a cache maintenance instruction, then a stage 2 Permission fault can be generated.

If a \texttt{DC IVAC} does not have write permission to the location it invalidates, then a Permission fault can be generated.

\texttt{DC ZVA} is not a cache maintenance instruction.
For more information, see:

- Data and unified caches on page D8-5215.
- Instruction caches on page D8-5215.
- Memory aborts on page D8-5180.
- Permission fault on page D8-5183.
- A64 Cache maintenance instructions on page D7-5054.

## D8.11.4 MMU fault-checking sequence

For a translation stage, when an IA is translated to an OA, the fault checking sequence is done using all of the following steps:

1. If the IA is subject to an alignment check, then check the alignment and do one of the following:
   - If the IA is not aligned, then an Alignment fault is generated.
   - If the IA is aligned, then continue to the next step.

2. Check that the IA maps to a translation table base address register, TTBR, and do one of the following:
   - If the IA does not map to a TTBR, then a Translation fault is generated.
   - If the IA maps to a TTBR and translation using that TTBR is disabled, then a Translation fault is generated.
   - If the IA maps to a TTBR, then get the translation table base address and continue to the next step.

3. Check that the translation table base address size is valid and do one of the following:
   - If the translation table base address size is not valid, then an Address size fault is generated.
   - If the translation table base address size is valid, then continue to the next step.

4. Fetch the descriptor, using all of the following steps:
   - If the descriptor address is an IPA from a stage 1 translation, then all of the following are done:
     - The fault checking sequence is done on the stage 2 translation, using the IPA as an IA to the stage 2 translation.
     - The OA from the stage 2 translation is the descriptor PA.
   - If the descriptor address is not an IPA from a stage 1 translation, then the descriptor address is the descriptor PA.
   - The descriptor fetch is initiated using the descriptor PA, and one of the following occurs:
     - A Synchronous External abort is generated.
     - The descriptor is returned.

5. Check that the descriptor is valid and do one of the following:
   - If the descriptor is not valid, then a Translation fault is generated.
   - If the descriptor is valid, then continue to the next step.

6. Check that the descriptor address size is valid and do one of the following:
   - If the descriptor address size is not valid, then an Address size fault is generated.
   - If the descriptor address size is valid, then continue to the next step.

7. Check the descriptor type and do one of the following:
   - For a Table descriptor, get the descriptor address, get the hierarchical permissions and attributes that apply to subsequent lookup levels, and go back to step 4 to fetch the next descriptor.
   - For a Block descriptor or Page descriptor, continue to the next step.

8. If FEAT_BBM is implemented and supported at level 1 or higher, and the fetched descriptor is a block descriptor with the nT bit set, then the implementation can generate a Translation fault.
9. If AF hardware management is disabled or not implemented, check the descriptor AF bit and do one of the following:
   • If AF is 0, then an Access flag fault is generated.
   • If AF is 1, then continue to the next step.
10. If AF hardware management is enabled, then the hardware attempts to update the AF and that might result in a Permission fault at stage 2, or a Synchronous External abort.
11. Get the OA and OA space from the Block descriptor or Page descriptor returned by the translation table walk.
12. Check the alignment required for the output memory type and do one of the following:
   • If the OA alignment is not valid, then an Alignment fault is generated.
   • If the OA alignment is valid, then continue to the next step.
13. Check the OA space access permissions and do one of the following:
   • If the OA space access is not permitted, then a Permission fault is generated.
   • If the OA space access is permitted, then continue to the next step.
14. If dirty state hardware management is enabled, then the hardware attempts to update the dirty state and that might result in a Permission fault at stage 2, or a Synchronous External abort.
15. The translation stage returns the OA and region attributes.

For a translation stage, the following figure illustrates the MMU fault checking sequence.
For a translation stage, the following figure illustrates the steps taken to fetch a descriptor during a translation table walk.
Stage 2 fault on a stage 1 translation table walk

For a translation regime that uses two translation stages, the memory access made as part of a stage 1 translation table lookup can generate one or more of the following during a stage 2 translation:

- A Translation fault, Address size fault, Access flag fault, or Permission fault.
- A synchronous External abort on the stage 2 descriptor access.
- A synchronous External abort on the memory access.

If an Address size fault, Translation fault, Access flag fault, or Permission fault is generated on a stage 2 translation of a stage 1 translation table walk, then all of the following apply:

- The exception is taken to EL2.
- ESR_EL2.ISS[7] is 1 to indicate a stage 2 fault during a stage 1 translation table walk.
- The part of the ISS field that might contain details of the instruction is invalid.

If a synchronous External abort is generated on a stage 2 translation of a stage 1 translation table walk, then all of the following apply:

- If SCR_EL3.EA is 0, then all of the following apply:
  - The exception is taken to EL2.
  - ESR_EL2.ISS[7] is 1 to indicate a stage 2 fault during a stage 1 translation table walk.
- If SCR_EL3.EA is 1, then all of the following apply:
  - The exception is taken to EL3.
— ESR_EL3.ISS[7] is 1 to indicate a stage 2 fault during a stage 1 translation table walk.

- The part of the ISS field that might contain details of the instruction is invalid.

IZGDIIH
If the stage 2 translation of a stage 1 translation table walk returns a Device memory type, then the value of HCR_EL2.PTW determines all of the following:

- If the value is 0, the translation table walk access is permitted and treated as an access to Normal, Non-cacheable memory.
- If the value is 1, the translation table walk access generates a stage 2 Permission fault.

IVXKVL
If the stage 2 translation of a stage 1 translation table lookup maps to Device memory, it likely indicates a Guest OS error where the stage 1 translation table is corrupted, and it is appropriate to trap this access to the hypervisor.

ILWGSL
If software updates HCR_EL2.PTW without changing the current VMID, then the software is required to invalidate the TLBs because they might hold entries that depend on the effect of HCR_EL2.PTW.

IHPGQN
For more information, see Permission fault on page D8-5183.

The lookup level associated with MMU faults

IXCCCM
For MMU faults, the Data Fault Status Code (DFSC) and Instruction Fault Status Code (IFSC) in the ESR_ELx report the translation lookup level associated with a given MMU fault type.

ICRKPZ
When an MMU fault is generated, all of the following determine the lookup level associated with the fault:

- For a fault generated by a translation table walk, the lookup level at which the fault occurred.
- For a Translation fault, one of the following:
  — The lookup level at which the fault occurred.
  — If a fault is generated because translation table walks are disabled by TCR_ELx.EPD[n], the IA is outside the range specified by the associated TTBR_ELx, or FEAT_E0PD is enabled and prevents access to addresses translated by TTBRn_ELx, then the fault is reported as a level 0 fault.
- For an Access flag fault, the lookup level at which the fault occurred.
- For a Permission fault, including a Permission fault generated by hierarchical permissions, the final lookup level that returned the Block descriptor or Page descriptor.

ICIZVP
For more information, see Permission fault on page D8-5183.

D8.11.5 MMU fault prioritization from a single address translation stage

RXCHFJ
For a single translation stage in a VMSA8-64 translation regime, the following numbered list shows the possible MMU faults on a memory access, from highest priority to lowest priority, and includes where stage 2 faults that are generated within a stage 1 translation table walk are prioritized:

1. For a stage 1 translation, an Alignment fault not caused by memory type.
2. If one or more of the following is true, then a Translation fault:
   - The IA out of the translated address range.
   - For a stage 1 translation, the IA requiring a TTBR_ELx that is disabled.
   - For a stage 2 translation, VTCR_EL2.SL0 is inconsistent with VTCR_EL2.T0SZ.
   - For a stage 2 translation, VSTCR_EL2.SL0 is inconsistent with VSTCR_EL2.T0SZ.
   - For a stage 2 translation, SL0 is programmed to a reserved value.
   - For a stage 2 translation, the Effective value of VTCR_EL2.DS is 1 and VTCR_EL2.{SL2, SL0} is inconsistent with VTCR_EL2.T0SZ.
   - For a stage 2 translation, the Effective value of VTCR_EL2.DS is 1 and VSTCR_EL2.{SL2, SL0} is inconsistent with VSTCR_EL2.T0SZ.
   - For a stage 2 translation, the Effective value of VTCR_EL2.DS is 1 and {SL2, SL0} is programmed to a reserved value.
3. If one or more of the following is true, an Address size fault on the address in TTBR_ELx:
   • The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS, TCR_EL3.PS, or VTCR_EL2.PS fails.
   • The programmed address is outside the implemented range.
4. If stage 2 address translation is enabled, then a stage 2 MMU fault on a level -1 memory access during a stage 1 table walk, including an Address size fault caused by a PA outside the implemented range.
5. A synchronous parity or ECC error on a level -1 lookup of a translation table walk.
6. A synchronous External abort on a level -1 lookup of a translation table walk.
7. Translation fault on a level -1 translation entry.
8. If one or more of the following is true, an Address size fault on the address in a level -1 lookup translation table entry:
   • The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS, TCR_EL3.PS, or VTCR_EL2.PS fails.
   • The OA is out of the implemented range.
9. If stage 2 address translation is enabled, then a stage 2 MMU fault on a level 0 memory access during a stage 1 table walk, including an Address size fault caused by a PA outside the implemented range.
10. A synchronous parity or ECC error on a level 0 lookup of a translation table walk.
11. A synchronous External abort on a level 0 lookup of a translation table walk.
12. Translation fault on a level 0 translation table entry.
13. If one or more of the following is true, an Address size fault on the address in a level 0 lookup translation table entry:
   • The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS, TCR_EL3.PS, or VTCR_EL2.PS fails.
   • The OA is out of the implemented range.
14. If stage 2 address translation is enabled, then a stage 2 MMU fault on a level 1 memory access during a stage 1 table walk, including an Address size fault caused by a PA outside the implemented range.
15. A synchronous parity or ECC error on a level 1 lookup of a translation table walk.
16. A synchronous External abort on a level 1 lookup of a translation table walk.
17. Translation fault on a level 1 translation table entry.
18. If one or more of the following is true, an Address size fault on the address in a level 1 lookup translation table entry:
   • The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS, TCR_EL3.PS, or VTCR_EL2.PS fails.
   • The OA is out of the implemented range.
19. If stage 2 address translation is enabled, then a stage 2 MMU fault on a level 2 memory access during a stage 1 table walk, including an Address size fault caused by a PA outside the implemented range.
20. A synchronous parity or ECC error on a level 2 lookup of a translation table walk.
22. Translation fault on a level 2 translation table entry.
23. If one or more of the following is true, an Address size fault on the address in a level 2 lookup translation table entry:
   • The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS, TCR_EL3.PS, or VTCR_EL2.PS fails.
   • The OA is out of the implemented range.
24. If stage 2 address translation is enabled, then a stage 2 MMU fault on a level 3 memory access during a stage 1 table walk, including an Address size fault caused by a PA outside the implemented range.
25. A synchronous parity or ECC error on a level 3 lookup of a translation table walk.
27. Translation fault on a level 3 translation table entry.
28. If one or more of the following is true, an Address size fault on the address in a level 3 lookup translation table entry:
   • The address size check on TCR_EL1.IPS, TCR_EL2.(I)PS, TCR_EL3.PS, or VTCR_EL2.PS fails.
   • The OA is out of the implemented range.
30. Alignment fault caused by the memory type.
31. Permission fault.
32. If stage 2 address translation is enabled, then a stage 2 fault on the memory access, including an Address size fault caused by a PA outside the implemented range.
33. Synchronous parity or ECC error on the memory access.
34. Synchronous External abort on the memory access.

If all the following apply, the prioritization between the stage 2 Permission fault on the stage 1 translation table walk and the stage 1 abort caused by the stage 1 Block descriptor or Page descriptor is IMPLEMENTATION DEFINED:
   • Hardware management of the stage 1 access flag or dirty state is enabled.
   • The access flag or dirty state in a stage 1 Block descriptor or Page descriptor needs to be updated.
   • For the location of the stage 1 Block descriptor or Page descriptor, the stage 2 translation has read permission but not write permission.
   • The stage 1 Block descriptor or Page descriptor generates an abort, which can be due to an Address size fault, an Alignment fault caused by memory type, or a Permission fault.

For more information, see Hardware management of the dirty state on page D8-5145.

For a TLB conflict abort, all of the following apply:
   • The priority of the abort is IMPLEMENTATION DEFINED.
   • The priority of the abort is required to be higher than any abort that depends on a value held in a TLB.

The cause of a TLB conflict abort depends on the TLB implementation.

For an IMPLEMENTATION DEFINED MMU fault caused by a Load-Exclusive or Store-Exclusive to an unsupported memory type, the priority is IMPLEMENTATION DEFINED.

For an MMU fault at a translation stage caused by an unsupported atomic hardware update, the priority of this fault lies at an IMPLEMENTATION DEFINED point between all of the following:
   • A priority immediately higher than an Access flag fault generated by the same translation stage as this MMU fault.
   • A priority immediately lower than a Permission fault generated by the same translation stage as this MMU fault.
## D8.12 Translation Lookaside Buffers

The Arm architecture does not specify any structure of *Translation Lookaside Buffers* (TLBs), and permits any structure that complies with the requirements described in this section.

**Translation table entries that generate a Translation fault, an Address size fault, or an Access flag fault are never cached in a TLB.**

When address translation is enabled, a translation table entry for an in-context translation regime that does not cause a Translation fault, an Address size fault, or an Access flag fault is permitted to be cached in a TLB or intermediate TLB caching structure as the result of an explicit or speculative access.

When address translation is enabled, if a translation table entry meets all of the following requirements, then that translation table entry is permitted to be cached in a TLB or intermediate TLB caching structure at any time:

- The translation table entry itself does not generate a Translation fault, an Address size fault, or an Access flag fault.
- The translation table entry is not from a translation regime configured by an Exception level that is lower than the current Exception level.

The Arm architecture permits TLBs to cache certain information from System control registers, including when any or all translation stages are disabled. The individual register descriptions specify System control register fields are permitted to be cached in a TLB.

For more information, see Chapter D17 *AArch64 System Register Descriptions*.

The TLB entries in all of the following translation regimes are out-of-context:

- When executing at EL3 or EL2, the TLB entries associated with the EL1&0 translation regime are out-of-context.
- When executing at EL3, the TLB entries associated with the EL2 or EL2&0 translation regime are out-of-context.

The VMSA provides TLB maintenance instructions for the management of TLB contents.

When a translation stage is disabled and then re-enabled, TLB entries are not corrupted.

### D8.12.1 TLB behavior at reset

When a reset occurs, an implementation is not required to automatically invalidate a TLB.

When a reset occurs, a TLB is affected in all of the following ways:

- All TLBs reset to an IMPLEMENTATION DEFINED state that might be UNKNOWN.
- It is IMPLEMENTATION DEFINED whether a specific TLB invalidation routine is required to invalidate a TLB before translation is enabled after a reset.

For the ELx reset is taken to, when a reset occurs, SCTLR_ELx.M is reset to 0. For the translation regime controlled by that SCTLR_ELx.M bit, when SCTLR_ELx.M is 0, TLB contents have no effect on address translation.

If an implementation requires a specific TLB invalidation routine, then all of the following apply:

- The routine is IMPLEMENTATION DEFINED.
- The implementation documentation is required to clearly document the routine.
- Arm recommends that the routine is based on the TLB maintenance instructions.

On a Cold reset or Warm reset, an implementation might require TLBs to maintain their contents from before the reset, including one or more of the following reasons:

- Power management.
- Debug requirements.

For more information on the TLB maintenance instructions used in a TLB invalidation routine, see *TLB maintenance instructions* on page D8-5201.
D8.12.2 TLB lockdown

TLB lockdown support is IMPLEMENTATION DEFINED.

If an implementation supports TLB lockdown, then all of the following apply:

- The lockdown mechanism is IMPLEMENTATION DEFINED.
- The implementation documentation is required to clearly document the interaction of the TLB lockdown mechanism with the architecture.
- A locked TLB entry is guaranteed to remain in the TLB, unless the locked TLB entry is affected by a TLBI operation.
- An unlocked TLB entry is not guaranteed to remain in the TLB.
- If a translation table entry is modified, then it is not guaranteed that a locked TLB entry remains coherent with the modified translation table entry.

If a translation table entry is modified, then it is not guaranteed that a locked TLB entry remains incoherent with the modified translation table entry because the lockdown mechanism might permit a TLB maintenance instruction to trigger an update of the locked TLB entry.

For more information, see The interaction of TLB lockdown with TLB maintenance instructions on page D8-5200.

The implementation is permitted to use the reserved IMPLEMENTATION DEFINED register encodings to implement TLB lockdown functions.

TLB lockdown functions might include, but are not limited to, all of the following:

- Unlock all locked TLB entries.
- Preload a translation table entry into a specific TLB level.

If an implementation supports TLB lockdown and EL2 is enabled, then when executing at EL1 or EL0, an exception due to TLB lockdown can be routed to one of the following:

- EL1, as a Data Abort exception.
- EL2, as a Hyp Trap exception.

D8.12.3 Use of ASIDs and VMIDs to reduce TLB maintenance requirements

TLB maintenance can be reduced during context switches by associating the translation table lookups from some translation regimes with an address space identifier (ASID), virtual machine identifier (VMID), or both.

Global and process-specific translation table entries

For translation regimes that use an ASID, it is possible to mark translations as global or non-global.

A TLB entry associated with a specific ASID value, or a specific ASID and a VMID value, can only be used to translate a VA in a context that is associated with the same ASID value, or ASID and VMID value.

For stage 1 of a translation regime that supports two privilege levels, non-global TLB entries are associated with a specific ASID.

For stage 1 of a translation regime that can support two privilege levels, Arm expects that software configures translations specific to a process to be associated with a specific ASID.

For all of the following, the translation does not support association of a translation with an ASID, and all translations are treated as global:

- A stage 1 translation that supports a single privilege level
- The stage 2 component of a two stage translation.

The ASID permits software to switch between process-specific translation table mappings without removing previous mappings cached for another ASID from a TLB.
For stage 1 of a translation regime that can support two privilege levels, each TTBR0_ELx and TTBR1_ELx has an ASID field.

All the following values of TCR_ELx.A1 determine the ASID value that is used by the translations of both the lower VA range and upper VA range:
- If the value is 0, then the TTBR0_ELx.ASID is used.
- If the value is 1, then the TTBR1_ELx.ASID is used.

The nG bit in a Block descriptor and Page descriptor indicates one of the following:
- If the value is 0, the translation is global and the TLB entry applies to all ASID values.
- If the value is 1, the translation is non-global and the TLB entry applies to only the current ASID value.

For a translation regime that supports global and non-global translations, translation table entries from lookup levels other than the final lookup level are treated as non-global, regardless of the value of the nG bit in the final lookup level.

For a translation in Secure state, if the NSTable bit in a Table descriptor is 1 at any level of the translation table walk, then the resulting translation is treated as non-global, regardless of the nG bit value in the Block descriptor or Page descriptor for the translation.

If EL2 is enabled, then for the EL1&0 translation regime, TLB entries created for the regime are associated with the current VMID.

If EL2 is enabled, then for the EL1&0 translation regime, TLB entries are associated with the current VMID in all cases, and include all of the following:
- For both global and non-global translations of any ASID value, and regardless of whether stage 2 translation is disabled, TLB entries containing information from stage 1 translation.
- For both stage 1 translation table walks and stage 1 OA, and regardless of whether stage 1 translation is disabled, TLB entries containing information from stage 2 translations.

If EL2 is enabled, then for the EL1&0 translation regime, TLB entries associated with a VMID are associated with a specific virtual machine.

The VMID permits software to switch between virtual machines that have different VMIDs without removing previous translation table mappings from a TLB.

VTTBR_EL2.VMID holds the current VMID.

ASID size

The ASID size is an IMPLEMENTATION DEFINED choice of 8 bits or 16 bits.

The maximum supported ASID size is indicated by ID_AA64MMFR0_EL1.ASIDBits.

For an implementation that supports 16-bit ASIDs, all of the following values of TCR_ELx.AS specify whether TTBR0_ELx.ASID[15:8] are used:
- If TCR_ELx.AS is 0, then TTBR0_ELx.ASID[15:8] are not used and are treated in all of the following ways:
  - For every purpose other than direct reads of TTBR0_ELx.ASID and TTBR1_ELx.ASID, the bits are ignored by hardware.
  - When used for allocating and matching entries in a TLB, the bits are treated as if they are all zeros.
- If TCR_ELx.AS is 1, then TTBR0_ELx.ASID[15:8] are used for allocating and matching entries in a TLB.

Bits[15:8] of the ASID field in TLBI instructions are considered by hardware regardless of the value of TCR_ELx.AS.

For a translation using VMSAv8-32, if the implementation supports 16-bit ASIDs, then the 8-bit ASID used is zero-extended to 16 bits.

The AArch32 ASID size is 8 bits.
VMID size

The VMID size is an IMPLEMENTATION DEFINED choice of 8 bits or 16 bits.

The maximum supported VMID size is specified by ID_AA64MMFR1_EL1.VMIDBits.

If and only if EL2 is using AArch64, then use of a 16-bit VMID is permitted.

For an implementation that supports a 16-bit VMID, all of the following values of VTCR_EL2.VS specify whether VTTBR_EL2.VMID[15:8] are used:

- If VTCR_EL2.VS is 0, then VTTBR_EL2.VMID[15:8] are not used and are treated in all of the following ways:
  - For every purpose other than direct reads of VTTBR_EL2.VMID, the bits are ignored by hardware.
  - When used for allocating and matching entries in a TLB, the bits are treated as if they are all zeros.
- If VTCR_EL2.VS is 1, then VTTBR_EL2.VMID[15:8] are used.

Common not private translations

FEAT_TTCNP allows multiple PEs in the same Inner Shareable domain and operating in the same translation regime to use the same translation table entries in a given translation stage.

If all of the following conditions are met, translation table entries pointed to by TTBR_ELx are shared with all other PEs in the Inner Shareable domain, and are referred to as common not private translation tables:

- For the PEs that share the translation tables, the Effective value of the corresponding TTBR_ELx.CnP is 1.
- For the PEs that share the translation tables, the same translation regime in the same Security state applies to the corresponding TTBR_ELx.
- If an ASID applies to the corresponding translation stage, then all of the PEs that share non-global translation table entries are required to have the same current ASID.
- If a VMID applies to the corresponding translation stage, then all of the PEs that share translation table entries are required to have the same current VMID.

For a common not private translation table, if a System register field with all of the following characteristics is set by a particular PE to a value that is different than the value set by the other PEs, then it is CONSTRAINED UNPREDICTABLE whether the System register field is interpreted using the value of the particular PE or the value of one of the other PEs that are sharing the translation table entry:

- The register field applies to the translation stage.
- The register field is permitted to be cached in a TLB.

For a translation regime with both stage 1 and stage 2 translations, a TLB entry can be shared between different PEs in one or more of the following cases only if the value of the TTBR_ELx.CnP bit is 1 for both stages of translation:

- The TLB entry holds information from stage 1 translation only.
- The TLB entry combines information from stage 1 and stage 2 translations.

The TTBR_ELx.CnP bit is permitted to be cached in a TLB.

For a common not private translation table, if a TTBR_ELx does not point to the same translation table as the other TTBR_ELx registers, then the system is misconfigured and an address translation using that TTBR_ELx causes one of the following to occur:

- Multiple hits in a TLB, which is permitted to generate a TLB conflict abort.
- A CONSTRAINED UNPREDICTABLE result due to caching of control or data values.

For more information, see TLB conflict abort on page D8-5183 and CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values on page K1-11585.
D8.13 TLB maintenance

For the purpose of TLB maintenance, a TLB entry is any structure that holds a translation table entry, including intermediate TLB caching structures and temporary working registers in translation table walk hardware.

When a translation table walk occurs, translation table entries that are permitted to be cached in a TLB might be held in TLB caching structures.

Entries held in a TLB are distinguished by all of the following context information:

- The Security state and translation regime.
- If applicable to the translation regime, then the VMID.
- If applicable to the translation regime, then whether the translation is global or non-global.
- If the translation is non-global, then the ASID.

TLBs, or TLB caching structures, are not guaranteed to remain coherent with changes to translation table entries, and are therefore distinct from data caches.

Indexing an intermediate TLB structure by the IA is permitted.

If FEAT_nTLBPA is not implemented, then indexing an intermediate TLB structure by all of the following is also permitted:

- The PA of the location holding the translation table entry.
- For stage 1 translations, the IPA of the location holding the translation table entry.

If FEAT_nTLBPA is not implemented, any TLB maintenance instruction that applies to a PE with the context information that is relevant to the translation table entry ensures cached copies of translation table entries are invalidated for that PE in all of the following implementations of intermediate TLB structures:

- Indexing the TLB using the PA of the location holding the translation table entry.
- For stage 1 translations, indexing the TLB using the IPA of the location holding the translation table entry.

FEAT_nTLBPA permits software to determine the existence of intermediate TLB caching structures that are indexed by PA or IPA and perform TLB maintenance accordingly when it would otherwise not be required.

When a TLB maintenance instruction targets only stage 1 entries, all of the following apply:

- The maintenance applies to any entries in caching structures that include stage 1 information used to translate the address or range of addresses being invalidated.
- If the TLB maintenance targets a specific ASID, then entries in caching structures that are not tagged with the specific ASID are not invalidated.
- If the stage 1 translation information contained in a single block or page has been collectively cached in multiple TLB entries, then all entries containing that stage 1 information are invalidated, regardless of whether the entry would be used to translate the address being invalidated.

When a TLB maintenance instruction applies only to stage 2 entries, all of the following apply:

- The maintenance applies to any entries in caching structures that include only stage 2 information used to translate the address or range of addresses being invalidated.
- If the stage 2 translation information contained in a single block or page has been collectively cached in multiple TLB entries, then all entries containing that stage 1 information are invalidated, regardless of whether the entry would be used to translate the address being invalidated.
- The maintenance is not required to apply to structures combining stage 1 and stage 2 information used to translate the address or range of addresses being invalidated.

When a TLB maintenance instruction applies to both stage 1 and stage 2 entries, all of the following apply:

- The maintenance applies to any entries in caching structures that include stage 1 information used to translate the address or range of addresses being invalidated.
- The maintenance applies to any entries in caching structures that include stage 2 information used to translate the address or range of addresses being invalidated.
• The maintenance applies to structures combining stage 1 and stage 2 information used to translate the address or range of addresses being invalidated.

When a translation table entry associated with a specific VMID or ASID is modified, software is required to invalidate the corresponding TLB entry to ensure that the modified translation table entry is visible to subsequent execution, including speculative execution.

When a System register field is modified and that field is permitted to be cached in a TLB, software is required to invalidate all TLB entries that might be affected by the field, at any address translation stage in the translation regime even if the translation stage is disabled, using the appropriate VMID and ASID, after any required System register synchronization.

For more information, see Synchronization requirements for AArch64 System registers on page D17-5547.

When a translation table entry that generates a Translation fault, Address size fault, or Access flag fault is changed to one that does not fault, all of the following apply to software:

• TLB invalidation is not required because an entry that generates one of the listed faults is never cached in a TLB.

• A Context synchronization event is required to ensure that the completed change to the translation table entry affects subsequent instruction fetches.

D8.13.1 Using break-before-make when updating translation table entries

If multiple execution threads use the same translation tables, then when a translation table entry is modified in one or more of the following ways, the architecture requires software to use a break-before-make sequence:

• A change to the memory type, Shareability or Cacheability.

• The translation OA is changed and one or more of the following apply:
  — Either or both of the old and new translations grant write permission, including cases where the DBM bit is set and hardware updates of the dirty state is enabled.
  — The memory contents at the new OA do not match the memory contents at the previous OA.

• If the requirements enabled by FEAT_BBM level 1 or above cannot be followed, all of the following changes to the block size used by the translation system:
  — Changing from a smaller size to a larger size, such as when a stage 2 Table descriptor is replaced with a Block descriptor.
  — Changing from a larger size to a smaller size, such as when a stage 2 Block descriptor is replaced with a Table descriptor.

• A global entry is created that might overlap with non-global entries in a TLB.

A change to the translation OA space is considered a change to the translation OA.

Use of a break-before-make sequence to ensure that old and new translation table entries are never simultaneously visible to different execution threads is guaranteed to prevent all of the following problems:

• Creating multiple TLB entries that apply to the same address.

• The effects of TLB caching breaking coherency.

• The effects of TLB caching breaking single-copy atomicity properties.

• The effects of TLB caching breaking ordering guarantees or uniprocessor semantics.

• The effects of TLB caching causing a failure to clear the Exclusives monitors.

A break-before-make sequence requires all of the following steps:

1. Replace the old translation table entry with an invalid entry.
2. Execute a DSB instruction to ensure the invalid entry is visible.
3. Invalidate TLB entries based on the translation table entry with a broadcast TLB invalidation instruction.
4. Execute a DSB instruction to ensure the invalidation completes.
5. Write the new translation table entry.
6. Execute a DSB instruction to ensure the new entry is visible.

For a translation stage with AF hardware management enabled, if a translation table entry is modified and the break-before-make sequence is not followed, then all of the following failures associated with AF hardware updates can occur:

- When a memory location associated with that translation table entry is accessed, the AF is not set.
- When hardware updates to that translation table entry are followed by stores appearing later in program order, the ordering required is not followed.

For a translation stage with dirty state hardware management enabled, if a translation table entry is modified and the break-before-make sequence is not followed, then all of the following failures associated with dirty state hardware updates can occur:

- When a memory location associated with that translation table entry is not written, hardware modifies the AP[2] or S2AP[1] descriptor bit.
- When a memory location associated with that translation table entry is written, hardware does not modify the AP[2] or S2AP[1] descriptor bit.
- When hardware updates to that translation table entry are followed by stores appearing later in program order, the ordering required is not followed.

### D8.13.2 Support levels for changing block size

All statements in this section require implementation of FEAT_BBM.

When a translation table entry is modified to change the block size, the hardware provides one of the following possible support levels affecting the break-before-make requirement to avoid breaking coherency, ordering guarantees or uniprocessor semantics, or failing to clear the Exclusives monitors:

- If level 0 is supported, then software is required to use break-before-make.
- If level 1 is supported, then software can use the level 0 approach or use the block translation entry bit, nT, in the Block descriptor.
- If level 2 is supported, then all of the following apply:
  - Software can use the level 0 approach or the level 1 approach.
  - Changing block size does not break coherency, ordering guarantees or uniprocessor semantics, or fail to clear the Exclusives monitors.

For more information, see [Block translation entry](#) on page D8-5160.

If any level is supported and the TLB entries are not invalidated after the writes that modified the translation table entries are completed, then a TLB conflict abort can be generated because in a TLB there might be multiple translation table entries that all translate the same IA.

For more information, see [TLB conflict abort](#) on page D8-5183.

If level 1 or level 2 is supported, then changing the Contiguous bit in a set of Block descriptors or Page descriptors can be done without breaking coherency, ordering guarantees or uniprocessor semantics, or failing to clear the Exclusives monitors.

If level 1 or level 2 is supported and the Contiguous bit in a set of Block descriptors or Page descriptors is changed, then a TLB conflict abort can be generated because multiple translation table entries might exist within a TLB that translates the same IA.

If all of the following apply, then a TLB conflict abort is reported to EL2:

- Level 1 or level 2 is supported.
- Stage 2 translations are enabled in the current translation regime.
- A TLB conflict abort is generated due to changing the block size or Contiguous bit.

If level 1 or level 2 is supported and a TLB conflict abort is generated, then TLB maintenance is required to remove the multiple TLB entries that translate the same address.
For more information, see *TLB maintenance due to TLB conflict*.

When a TLB conflict abort is generated, it is IMPLEMENTATION DEFINED whether it is a stage 1 abort or a stage 2 abort.

D8.13.3 TLB maintenance due to TLB conflict

If multiple TLB entries translate the same address, then the ALL or VMALL form of a TLB maintenance instruction that targets the given translation regime is guaranteed to remove all TLB entries within that regime.

All of the following instructions are guaranteed to remove the TLB entries associated with a conflict:

- For the EL1&0 translation regime, when stage 2 translations are in use, either TLBI VMALLS1E1 or TLBI ALLE1.
- For the EL1&0 translation regime, when stage 2 translations are not in use, either TLBI VMALLE1 or TLBI ALLE1.
- For the EL2&0 translation regime, either TLBI VMALLE1 or TLBI ALLE2.
- For the EL2 translation regime, TLBI ALLE2.
- For the EL3 translation regime, TLBI ALLE3.

If multiple TLB entries translate the same address, then the minimum set of TLB maintenance operations required to guarantee all TLB entries associated with that address and translation regime have been invalidated is IMPLEMENTATION DEFINED.

D8.13.4 The interaction of TLB lockdown with TLB maintenance instructions

If a TLB entry is locked and a TLB invalidate all instruction is executed and, if that entry was not locked the TLB invalidate all instruction would invalidate that entry, then one of the following IMPLEMENTATION DEFINED behaviors occur:

- The locked TLB entry is not affected.
- If the TLB entry is locked or might be locked, an IMPLEMENTATION DEFINED Data Abort exception is generated.

The IMPLEMENTATION DEFINED Data Abort on a TLB invalidate all instruction permits TLB invalidation routines to choose to invalidate a large range of addresses, without considering whether any TLB entries are locked, or to not affect locked TLB entries.

If a TLB entry is locked and a TLB invalidate by VA or invalidate by ASID instruction is executed and, if that entry was not locked the TLB invalidate by VA or invalidate by ASID instruction would invalidate that entry, then one of the following IMPLEMENTATION DEFINED behaviors occur:

- The locked TLB entry is invalidated.
- The locked TLB entry is not affected.
- If the TLB entry is locked or might be locked, an IMPLEMENTATION DEFINED Data Abort exception is generated.

The exception syndrome definitions include a fault code for cache and TLB lockdown faults.

If an implementation uses an abort mechanism when entries that can be locked down are not actually locked down, then all of the following are required:

- The IMPLEMENTATION DEFINED instruction sequences that perform the required operations on entries that are not locked down are documented.
- An IMPLEMENTATION DEFINED mechanism is documented that ensures no TLB entries are locked.
- For locked entries, one of the other specified alternatives is implemented.

If an implementation supports TLB lockdown, then Arm recommends that IMPLEMENTATION DEFINED TLB maintenance instruction sequences use the architecturally-defined operations to minimize the number of customized operations required.

For more information on TLB lockdown, see *TLB lockdown* on page D8-5194.
D8.13.5 TLB maintenance instructions

For a given translation regime, the architecture defines TLB maintenance instructions that provide all of the following functions:

- Invalidate all corresponding entries in a TLB.
- Invalidate a corresponding single TLB entry by ASID for a non-global entry.
- Invalidate all corresponding TLB entries that match a specific ASID.
- Invalidate all corresponding TLB entries that match a specific VA, regardless of the ASID.
- Invalidate all corresponding TLB entries within a range of addresses.

Each TLB maintenance instruction can be applied to one of the following:

- Only the PE that executes the instruction
- All PEs in the same Shareability domain as the PE that executes the instruction.

The A64 assembly language syntax of a TLB maintenance instruction is TLBI <operation>{, <Xt>}

The <operation> in TLBI <operation>{, <Xt>} is one of the following:

- ALLE1{NXS}, ALLE2{NXS}, ALLE3{NXS}, ALLE1IS{NXS}, ALLE2IS{NXS}, ALLE3IS{NXS}, ALLE1OS{NXS}, ALLE2OS{NXS}, or ALLE3OS{NXS}.
- VMALLE1{NXS}, VMALLE1IS{NXS}, VMALLE1OS{NXS}, VMALLE1E1{NXS}, VMALLE1E1IS{NXS}, or VMALLE1E1OS{NXS}.
- ASIDE1{NXS}, ASIDE1IS{NXS}, or ASIDE1OS{NXS}.
- {R}VA{L}E1{NXS}, {R}VA{L}E1IS{NXS}, {R}VA{L}E1OS{NXS}, {R}VA{L}E2{NXS}, {R}VA{L}E2IS{NXS}, {R}VA{L}E2OS{NXS}, or {R}VA{L}E3{NXS}.
- {R}VAA{L}E1{NXS}, {R}VAA{L}E1IS{NXS}, or {R}VAA{L}E1OS{NXS}.
- {R}IPAS2{L}E1{NXS}, {R}IPAS2{L}E1IS{NXS}, or {R}IPAS2{L}E1OS{NXS}.

The <operation> in TLBI <operation>{, <Xt>} has a structure of {R}<type><regime><shareability>{NXS} with all of the following components:

- R is optional and determines one of the following:
  - When present, the instruction applies to all TLB entries translating addresses within the address range.
  - When not present, the instruction applies to all TLB entries translating a single address that could be used by the PE that executes the TLBI instruction.
- <type> is one of the following:
  - ALL specifies all translations at <regime>.
  - VA{L} specifies all translations at <regime> using the supplied address and, if an ASID is supplied, the supplied ASID or global entries.
  - VA{L}E1{NXS}, VA{L}E1IS{NXS}, or VA{L}E1OS{NXS}.
  - VMALLE1{NXS}, VMALLE1IS{NXS}, or VMALLE1OS{NXS}.
  - ASIDE1{NXS}, ASIDE1IS{NXS}, or ASIDE1OS{NXS}.
  - {R}VA{L}E1{NXS}, {R}VA{L}E1IS{NXS}, or {R}VA{L}E1OS{NXS}.
  - {R}IPAS2{L}E1{NXS}, {R}IPAS2{L}E1IS{NXS}, or {R}IPAS2{L}E1OS{NXS}.

- <regime> is one of the following Exception levels:
  - If E1, then EL1.
  - If E2, then EL2.
  - If E3, then EL3.
<shareability> is one of the following:
- IS indicates that the instruction applies to all TLBs in the Inner Shareable domain.
- OS indicates that the instruction applies to all TLBs in the Outer Shareable domain.
- The parameter is optional, and no value indicates that the instruction applies to all TLBs that could be used by the PE that executes the instruction.

NXS has all of the following properties:
- The parameter is optional.
- When present, the instruction is considered complete after all issued memory transactions using translation information held in TLB entries that have the associated XS attribute set to 0 and are within the scope of the instruction, are complete.
- When not present, the instruction is considered complete after all issued memory transactions using translation information held in TLB entries that are within the scope of the instruction are complete, regardless of the XS attribute value.
- The parameter can only be present when FEAT_XS is implemented.

If <regime> is higher than the current Exception level, then the TLB maintenance instruction is UNDEFINED.

All TLB maintenance instructions are UNDEFINED at EL0.

TLBI ALLE1{NXS}, TLBI ALLE1IS{NXS}, TLBI ALLE1OS{NXS}, TLBI {R}IPAS2(L)E1{NXS}, TLBI {R}IPAS2(L)E1IS{NXS}, TLBI {R}IPAS2(L)E1OS{NXS}, TLBI VMALLS12E1{NXS}, and TLBI VMALLS12E1IS{NXS} and TLBI VMALLS12E1OS{NXS} are UNDEFINED at EL1.

If EL2 is not implemented, the TLBI {R}VA{L}E2{NXS}, TLBI {R}VA{L}E2IS{NXS}, TLBI {R}VA{L}E2OS{NXS}, TLBI ALLE2{NXS}, TLBI ALLE2IS{NXS}, and TLBI ALLE2OS{NXS} instructions are UNDEFINED.

When a TLB entry is invalidated by one PE, it is inconsistent with the architecture to allow another PE to refill that TLB entry so that the new entry gives the appearance to software that the invalidation did not occur.

The <Xt> in TLBI <operation>{, <Xt>} has all of the following properties:
- It specifies a register that passes one or both of an address and an ASID as an argument.
- For operations that include a VA, if FEAT_TTL is implemented, then the register passes the level hint.
- For range-based TLBI operations, then the register passes information about the address range.
- It is required by the TLBI ASID, TLBI {R}VA{L}, TLBI {R}VAA{L}, and TLBI {R}IPAS2{L} instructions.

**TLB maintenance instructions that do not apply to a range of addresses**

For TLB maintenance instructions that take a VA, an ASID, or both as an argument, and that do not apply to a range of addresses, the register specified by the Xt argument has the following format:
- Register bits[63:48] are one of the following:
  - If the instruction requires an ASID argument, the ASID.
  - If the instruction does not require an ASID argument, RES0.
- If FEAT_TTL is not implemented, then register bits[47:44] are RES0.
- If FEAT_TTL is implemented, then register bits[47:44] are one of the following:
  - If the instruction requires a VA argument, the translation table level hint, TTL.
  - If the instruction does not require a VA argument, RES0.
- Register bits[43:0] are one of the following:
  - If the instruction requires a VA argument, VA[55:12].
  - If the instruction does not require a VA argument, RES0.

For TLB maintenance instructions that take a VA, hardware interprets VA[63:56] as each having the same value as VA[55].
**RFGPTT**

For TLB maintenance instructions that take a VA, if the instruction targets a translation regime that is using AArch32, then all of the following apply:

- Software is required to treat VA[55:32] as RES0.
- VA[55:32] are ignored when the instruction is executed, and interpreted as being zero.
- VA[63:56] are interpreted as being zero.

**ILHKCP**

If the implementation supports 16-bit ASIDs, then software is required to set the upper 8 bits of the ASID to 0 when the context being invalidated uses only 8 bits.

**ICPNYZ**

For TLB maintenance instructions that take a register argument that holds an IPA and that do not apply to a range of addresses, the register specified by the Xt argument has the following format:

- Register bit[63] is one of the following:
  - If the instruction is executed in Secure state, the NS bit specifying the Secure or Non-secure IPA space.
  - If the instruction is executed in Non-secure state, RES0.
- If FEAT_TTL is not implemented, then register bits[62:40] are RES0.
- If FEAT_TTL is implemented, then all of the following apply:
  - Register bits[62:48] are RES0.
  - Register bits[47:44] are the translation table level hint, TTL.
  - Register bits[43:40] are RES0.
- Register bits[39:36] are one of the following:
  - If 52-bit addresses are supported, IPA[51:48].
  - If 52-bit addresses are not supported, RES0.
- Register bits[35:0] are IPA[47:12].

**IHQTLK**

For TLB maintenance instructions that take a register argument that holds a VA or an IPA, register bits that hold the address have all of the following properties:

- If the 4KB granule size is used, all address bits in the register bits are used by the invalidation instruction.
- If the 16KB granule size is used, register bits[1:0] are ignored because address bits[13:12] have no effect on the invalidation instruction.
- If the 64KB granule size is used, register bits[3:0] are ignored because address bits[15:12] have no effect on the invalidation instruction.

**TLB maintenance instructions that apply to a range of addresses**

**IQPHNP**

For TLB maintenance instructions that take a VA, or a VA and an ASID, and that apply to a range of addresses, the register specified by the Xt argument has the following format:

- Register bits[63:48] are one of the following:
  - If the instruction requires an ASID argument, the ASID.
  - If the instruction does not require an ASID argument, RES0.
- Register bits[47:46] are the translation granule size, TG.
- Register bits[45:44] are the SCALE field that is the exponent element of the calculation that produces the upper range.
- Register bits[43:39] are the NUM field that is the base element of the calculation that produces the upper range.
- If FEAT_TTL is not implemented, then register bits[38:37] are RES0.
- If FEAT_TTL is implemented, then register bits[38:37] are the translation table level hint, TTL.
- Register bits[36:0] give the starting address of the range of addresses, BaseADDR.
For TLB maintenance instructions that take a register argument that holds an IPA and that apply to a range of addresses, the register specified by the \texttt{Xt} argument has the following format:

- Register bit[63] is one of the following:
  - If the instruction is executed in Secure state, the NS bit specifying the Secure or Non-secure IPA space.
  - If the instruction is executed in Non-secure state, RES0.
- Register bits[62:48] are RES0.
- Register bits[47:46] are the translation granule size, TG.
- Register bits[45:44] are the SCALE field that is the exponent element of the calculation that produces the upper range.
- Register bits[43:39] are the NUM field that is the base element of the calculation that produces the upper range.
- If \texttt{FEAT_TTL} is not implemented, then register bits[38:37] are RES0.
- If \texttt{FEAT_TTL} is implemented, then register bits[38:37] are the translation table level hint, TTL.
- Register bits[36:0] specify the starting address of the range of addresses, BaseADDR.

For TLB instructions that apply to a range of addresses, the following table shows the TG field encodings that define the translation granule size for the translations that are being invalidated, and the Translation\_Granule\_Size in bytes used in determining the address range.

<table>
<thead>
<tr>
<th>TG</th>
<th>Translation granule size</th>
<th>Translation_Granule_Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Reserved</td>
<td>N/A</td>
</tr>
<tr>
<td>01</td>
<td>4KB translation granule</td>
<td>4096</td>
</tr>
<tr>
<td>10</td>
<td>16KB translation granule</td>
<td>16384</td>
</tr>
<tr>
<td>11</td>
<td>64KB translation granule</td>
<td>65536</td>
</tr>
</tbody>
</table>

For TLB instructions that apply to a range of addresses, if the translations use a different translation granule size than the one specified by the TG field, then the architecture does not require that the instruction invalidate those entries.

For all TLB range maintenance instructions, TLB entries that translate addresses within the address range determined by the following formula are invalidated:

\[ \text{BaseADDR} <= \text{input\_address} < \text{BaseADDR} + ((\text{NUM} +1)\times2^{(5\times\text{SCALE} +1)} + \text{Translation\_Granule\_Size}) \].

For a TLB range maintenance instruction, BaseADDR is derived from bits[36:0] in the register specified by the \texttt{Xt} argument as one of the following:

- For the 4KB granule size, one of the following:
  - If the \texttt{Effective value} of TCR\_EL1\_DS is 0, then BaseADDR[48:12].
  - If the \texttt{Effective value} of TCR\_EL1\_DS is 1, then BaseADDR[52:16]. In this case, BaseADDR[15:12] is treated as all zero.
- For the 16KB granule size, one of the following:
  - If the \texttt{Effective value} of TCR\_EL1\_DS is 0, then BaseADDR[50:14].
  - If the \texttt{Effective value} of TCR\_EL1\_DS is 1, then BaseADDR[52:16]. In this case, BaseADDR[15:14] is treated as all zero.
- For the 64KB granule size, BaseADDR[52:16].
For all of the following, the invalidated address range is UNPREDICTABLE:

- For the 4KB translation granularity, one of the following:
  - The Effective value of TCR_EL1.DS is 1, the TTL field is \(0b00\), and BaseADDR[38:12] are not all zero.
  - The TTL field is \(0b01\) and BaseADDR[29:12] are not all zero.
  - The TTL field is \(0b10\) and BaseADDR[20:12] are not all zero.

- For the 16KB translation granularity, one of the following:
  - The Effective value of TCR_EL1.DS is 1, the TTL field is \(0b01\), and BaseADDR[35:14] are not all zero.
  - The TTL field is \(0b10\) and BaseADDR[24:14] are not all zero.

- For the 64KB translation granularity, one of the following:
  - The TTL field is \(0b01\) and BaseADDR[41:16] are not all zero.
  - The TTL field is \(0b10\) and BaseADDR[28:16] are not all zero.

For more information, see Broadcast TLB maintenance on page D8-5211.

### Translation table level hint

All statements in this section require implementation of FEAT_TTL.

For the address being invalidated by a TLB maintenance instruction, the TTL field in the register specified by \(X_t\) indicates the lookup level of the translation table walk holding the leaf entry.

Hardware can use the TTL field to determine if the page might have been splintered into multiple TLB entries.

For an entry being invalidated by a TLB maintenance instruction, if an incorrect TTL value is specified, then the architecture does not require any entries to be invalidated from a TLB.

For TLB instructions that apply to a single address, the following table shows the 4-bit TTL field encodings.

<table>
<thead>
<tr>
<th>TTL[3:2]</th>
<th>Translation granule</th>
<th>TTL[1:0]</th>
<th>Information supplied</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>-</td>
<td>RES0</td>
<td>No information supplied about the translation level. Hardware assumes that the leaf entry can be from any level.</td>
</tr>
<tr>
<td>01</td>
<td>4KB</td>
<td>00</td>
<td>If FEAT_LPA2 is not implemented, then this value is reserved, and hardware treats this as if TTL[3:2] is (0b00). If FEAT_LPA2 is implemented, then the leaf entry for the address being invalidated is on level 0 of the translation table walk.</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>01</td>
<td>The leaf entry for the address being invalidated is on level 1 of the page table walk.</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>00</td>
<td>The leaf entry for the address being invalidated is on level 2 of the page table walk.</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>00</td>
<td>The leaf entry for the address being invalidated is on level 3 of the page table walk.</td>
</tr>
<tr>
<td>10</td>
<td>16KB</td>
<td>00</td>
<td>This value is reserved, and hardware treats this as if TTL[3:2] is (0b00).</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>01</td>
<td>If FEAT_LPA2 is not implemented, then this value is reserved, and hardware treats this as if TTL[3:2] is (0b00). If FEAT_LPA2 is implemented, then the leaf entry for the address being invalidated is on level 1 of the translation table walk.</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>10</td>
<td>The leaf entry for the address being invalidated is on level 2 of the page table walk.</td>
</tr>
</tbody>
</table>
The AArch64 Virtual Memory System Architecture

D8.13 TLB maintenance

For TLB instructions that apply to a range of addresses, the following table shows the 2-bit TTL field encodings.

**Table D8-66 TTL field encodings in TLB instructions that apply to a range of addresses**

<table>
<thead>
<tr>
<th>TTL</th>
<th>Information supplied</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>The leaf entries in the range can be using any level for the translation table entries.</td>
</tr>
</tbody>
</table>
| 01   | When the 4KB or 64KB translation granule is used, all leaf entries to invalidate are level 1 translation table entries.  
  If FEAT_LPA2 is not implemented, then when using a 16KB translation granule, this value is reserved, and hardware treats this as if TTL is 0b00.  
  If FEAT_LPA2 is implemented, then when using a 16KB translation granule, all leaf entries to invalidate are level 1 translation table entries. |
| 10   | All leaf entries to invalidate are level 2 translation table entries.                 |
| 11   | All leaf entries to invalidate are level 3 translation table entries.                 |

**TLB maintenance instruction scope**

If `<type>` is `ALL` in a TLB maintenance instruction, then all of the following apply:

- For all values of `<regime>`, the instruction exists.
- For the Security state specified by `SCR_EL3.NS` and `SCR_EL3.EEL2`, the instruction applies to all cached copies of the stage 1 and stage 2 translation table entries from any lookup level in the translation table walk required to translate any address at the specified Exception level.
- If EL2 is enabled, then for entries from the EL1&0 translation regime the instruction applies to cached copies of translation table entries with any VMID.
- For entries from a translation regime in which an ASID is valid, the instruction applies to all of the following cached copies of translation table entries from any lookup level in the translation table walk:
  - Global entries.
  - Non-global entries with any ASID.

If `<type>` is `VMALL` in a TLB maintenance instruction, then all of the following apply:

- If and only if the value of `<regime>` is E1, then the instruction applies to all of the following:
  - When `HCR_EL2.[E2H, TGE]` is not {1, 1}, then the EL1&0 translation regime.
  - When `HCR_EL2.[E2H, TGE]` is {1, 1}, then the EL2&0 translation regime.
- For the Security state specified by `SCR_EL3.NS` and `SCR_EL3.EEL2`, the instruction applies to all cached copies of the stage 1 translation table entries from any lookup level in the translation table walk required to translate any address at the specified Exception level.

<table>
<thead>
<tr>
<th>TTL[3:2]</th>
<th>Translation granule</th>
<th>TTL[1:0]</th>
<th>Information supplied</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td></td>
<td>00</td>
<td>This value is reserved, and hardware treats this as if TTL[3:2] is 0b00.</td>
</tr>
<tr>
<td>11</td>
<td>64KB</td>
<td>00</td>
<td>The leaf entry for the address being invalidated is on level 3 of the page table walk.</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>01</td>
<td>The leaf entry for the address being invalidated is on level 1 of the page table walk.</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>10</td>
<td>The leaf entry for the address being invalidated is on level 2 of the page table walk.</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>11</td>
<td>The leaf entry for the address being invalidated is on level 3 of the page table walk.</td>
</tr>
</tbody>
</table>
• If EL2 is enabled, then for entries from the EL1&0 translation regime the instruction applies only to cached copies of translation table entries with the current VMID.

• For entries from a translation regime in which an ASID is valid, the instruction applies to all of the following cached copies of translation table entries from any lookup level in the translation table walk:
  — Global entries.
  — Non-global entries with any ASID.

RBMMCC

If `<type>` is `VMALLS12` in a TLB maintenance instruction, then all of the following apply:

• If and only if the value of `<regime>` is E1, then the instruction applies to the EL1&0 translation regime.

• For the Security state specified by SCR_EL3.NS and SCR_EL3.EEL2, the instruction applies to all cached copies of the stage 1 and stage 2 translation table entries from any lookup level in the translation table walk required to translate any address at the specified Exception level.

• The instruction applies only to cached copies of translation table entries with the current VMID.

• For entries from a translation regime in which an ASID is valid, the instruction applies to all of the following cached copies of translation table entries from any lookup level in the translation table walk:
  — Global entries.
  — Non-global entries with any ASID.

• If one of the following is true and the instruction is executed at EL3, the instruction is not UNDEFINED and has the same effect as `TLBI VMALL` because there are no stage 2 translations to invalidate:
  — EL2 is not implemented.
  — The Effective value of SCR_EL3.NS is 0 and EL2 is disabled

RFKSVM

If `<type>` is `ASID` in a TLB maintenance instruction, then all of the following apply:

• If and only if the value of `<regime>` is E1, then the instruction applies to all of the following:
  — When HCR_EL2.{E2H, TGE} is not {1, 1}, then the EL1&0 translation regime when executing at EL1.
  — When HCR_EL2.{E2H, TGE} is {1, 1}, then the EL2&0 translation regime when executing at EL2.

• For the Security state specified by SCR_EL3.NS and SCR_EL3.EEL2, the instruction applies to all cached copies of the stage 1 translation table entries from any lookup level in the translation table walk required to translate any address at the specified Exception level.

• If EL2 is enabled, then for entries from the EL1&0 translation regime the instruction applies only to cached copies of translation table entries with the current VMID.

• The instruction applies only to non-global entries from the final lookup level that matches the specified ASID.

RVMSWQ

If `<type>` is `VA` in a TLB maintenance instruction, then all of the following apply:

• For all values of `<regime>`, the instruction exists.

• For the Security state specified by SCR_EL3.NS and SCR_EL3.EEL2, the instruction applies to all cached copies of the stage 1 translation table entries from any lookup level in the translation table walk required to translate the address at the specified Exception level.

• If EL2 is enabled, then for entries from the EL1&0 translation regime the instruction applies only to cached copies of translation table entries with the current VMID.

• For instructions where the value of `<regime>` is E2, the instruction applies to the EL2&0 or EL2 translation regime, as determined by the value of HCR_EL2.E2H.

• For entries from a translation regime in which an ASID is valid, the instruction applies only to all of the following cached copies of translation table entries:
  — A non-global entry from the final lookup level that matches the specified ASID.
  — A global entry from the final lookup level.

RDRCZZ

If `<type>` is `VAL` in a TLB maintenance instruction, then all of the following apply:

• For all values of `<regime>`, the instruction exists.
For the Security state specified by SCR_EL3.NS and SCR_EL3.EEL2, the instruction applies to all cached copies of the stage 1 translation table entries from the final lookup level in the translation table walk required to translate the address at the specified Exception level.

If EL2 is enabled, then for entries from the EL1&0 translation regime the instruction applies only to cached copies of translation table entries from the final lookup level with the current VMID.

For instructions where the value of <regime> is E2, the instruction applies to the EL2&0 or EL2 translation regime, as determined by the value of HCR_EL2.E2H.

For entries from a translation regime in which an ASID is valid, the instruction applies only to all of the following cached copies of translation table entries:

- A non-global entry from the final lookup level that matches the specified ASID.
- A global entry from the final lookup level.

If <type> is VAA in a TLB maintenance instruction, then all of the following apply:

- If and only if the value of <regime> is E1, then the instruction applies to all of the following:
  - When HCR_EL2.{E2H, TGE} is not {1, 1}, then the EL1&0 translation regime when executing at EL1.
  - When HCR_EL2.{E2H, TGE} is {1, 1}, then the EL2&0 translation regime when executing at EL2.
- For the Security state specified by SCR_EL3.NS and SCR_EL3.EEL2, the instruction applies to all cached copies of the stage 1 translation table entries from any lookup level in the translation table walk required to translate the address at the specified Exception level.

If EL2 is enabled, then for entries from the EL1&0 translation regime the instruction applies only to cached copies of translation table entries with the current VMID.

For instructions where the value of <regime> is E2, the instruction applies to the EL2&0 or EL2 translation regime, as determined by the value of HCR_EL2.E2H.

The instruction applies to all of the following cached copies of translation table entries from any lookup level in the translation table walk:

- Global entries.
- Non-global entries with any ASID.

If <type> is VAAL in a TLB maintenance instruction, then all of the following apply:

- If and only if the value of <regime> is E1, then the instruction applies to all of the following:
  - When HCR_EL2.{E2H, TGE} is not {1, 1}, then the EL1&0 translation regime when executing at EL1.
  - When HCR_EL2.{E2H, TGE} is {1, 1}, then the EL2&0 translation regime when executing at EL2.
- For the Security state specified by SCR_EL3.NS and SCR_EL3.EEL2, the instruction applies to all cached copies of the stage 1 translation table entries from the final lookup level in the translation table walk required to translate the address at the specified Exception level.

If EL2 is enabled, then for entries from the EL1&0 translation regime the instruction applies only to cached copies of translation table entries from the final lookup level with the current VMID.

For instructions where the value of <regime> is E2, the instruction applies to the EL2&0 or EL2 translation regime, as determined by the value of HCR_EL2.E2H.

The instruction applies to all of the following cached copies of translation table entries from the final lookup level in the translation table walk:

- Global entries.
- Non-global entries with any ASID.

If <type> is IPAS2 in a TLB maintenance instruction, then all of the following apply:

- If and only if the value of <regime> is E1, and if EL2 is enabled, then the instruction applies only to the EL1&0 translation regime.
• If all of the following apply, the instruction applies to all cached copies of the stage 2 translation table entries from any lookup level in the translation table walk required to translate the address:
  — The TLB entry holds only a stage 2 translation table entry.
  — The entry applies only to the current VMID.
• The instruction is not required to invalidate TLB entries that combine stage 1 and stage 2 translation table entries.
• If one of the following is true, then the instruction executes as a NOP:
  — EL2 is not implemented.
  — The Effective value of SCR_EL3.NS is 0 and EL2 is disabled.

If <type> is IPAS2l in a TLB maintenance instruction, then all of the following apply:
• If and only if the value of <regime> is E1, and if EL2 is enabled, then the instruction applies only to the EL1&0 translation regime.
• If all of the following apply, the instruction applies to all cached copies of the stage 2 translation table entries from the final lookup level in the translation table walk required to translate the address:
  — The TLB entry holds only a stage 2 translation table entry.
  — The entry applies only to the current VMID.
• The instruction is not required to invalidate TLB entries that combine stage 1 and stage 2 translation table entries.
• If one or more of the following is true, then the instruction executes as a NOP:
  — EL2 is not implemented.
  — The Effective value of SCR_EL3.NS is 0 and EL2 is disabled.

### D8.13.6 Operation of the TLB maintenance instructions

A TLB maintenance instruction applies whether a translation stage is enabled or disabled.

A TLB maintenance instruction can affect any TLB entries that are not locked down.

There is no guarantee that an unlocked TLB entry remains in the cache. Therefore, it is not possible to determine architecturally whether a TLB maintenance instruction has affected any TLB entries that were not specified by the instruction.

The TLB maintenance instructions specify the Exception level of the translation regime to which they apply.

A TLB maintenance instruction never generates an MMU abort.

If EL3 is implemented, then SCR_EL3.NS modifies the effect of the TLB maintenance instructions in all of the following ways:
• For instructions that apply to the EL2 or EL2&0 translation regime, if SCR_EL3.{EEL2, NS} is not {0, 0}, then SCR_EL3.NS identifies whether the maintenance instructions apply to the Secure or Non-secure variant of the corresponding regime.
• If SCR_EL3.{EEL2, NS} is {0, 0}, then instructions that apply to the EL2 or EL2&0 translation regime are UNDEFINED.
• For instructions that apply to the EL3 translation regime, SCR_EL3.NS has no effect.

A TLB maintenance instruction is not affected by the current state of all of the following control bits involved in the translation process:
• In AArch64 state, all of the following:
  — SCTLR_EL1.M.
  — SCTLR_EL2.M.
  — SCTLR_EL3.{M, RW}.
  — HCR_EL2.{VM, RW}.
— TCR_EL1.{TG1, EPD1, T1SZ, TG0, EPD0, T0SZ, AS, A1}.
— If HCR_EL2.E2H is 0, then TCR_EL2.{TG0, T0SZ}.
— If the Effective value of HCR_EL2.E2H is 1, then TCR_EL2.{TG1, EPD1, T1SZ, TG0, EPD0, T0SZ, AS, A1}.
— TCR_EL3.{TG0, T0SZ}.
— VTCR_EL2.{SL0, T0SZ}.
— TTBR0_EL1.ASID.
— TTBR1_EL1.ASID.
— If the Effective value of HCR_EL2.E2H is 1, then all of the following:
  — TTBR0_EL2.ASID.
  — TTBR1_EL2.ASID.

• In AArch32 state, all of the following:
  — SCTLR.M.
  — HCR.VM.
  — TTBCR.{EAE, PD1, PD0, N, EPD1, T1SZ, EPD0, T0SZ, A1}.
  — HTCR.T0SZ.
  — VTCR.{SL0, T0SZ}.
  — TTBR0.ASID.
  — TTBR1.ASID.
  — CONTEXTIDR.ASID.

Address-based TLBI instructions are applied to the extent of the <shareability> specified by the instruction regardless of the Shareability assigned by any translation of the IA or Shareability attribute stored in TLB entries.

For TLB maintenance instructions that operate only on instruction TLBs or only on data TLBs, Arm deprecates their use in AArch32 state.

When a TLB maintenance instruction is executed, all of the following apply:

• The minimum size of a TLB entry that is required to be invalidated from the TLB is at least the size specified by the corresponding translation table entry.
• The Contiguous bit does not affect the minimum size of a TLB entry that is required to be invalidated from the TLB.

When invalidating a mapping greater than a translation granule size or block size, for example a mapping that is specified using the Contiguous bit, software is required to issue a TLB maintenance instruction for each block or granule within the configured contiguous region size to invalidate the entire mapping.

An implementation is permitted to invalidate more than the minimum required TLB entries.

Arm recommends not invalidating entries that are not required to be invalidated to minimize the performance impact.

Arm expects software to mostly use address-based TLB invalidation instructions that apply to entries cached from the last level of a stage 1 or a stage 2 translation table walk to avoid unnecessary loss of the cached intermediate translation table entries.

For all of the following reasons, software is required to set VTTBR_EL2.VMID[7:0] to a known value as part of the PE initialization sequence:

• The VTTBR_EL2.VMID field resets to a value that is architecturally UNKNOWN.
• If EL2 is enabled, dependencies on the VMID in the EL1&0 translation regime apply whether stage 2 translation is enabled or disabled.
**Invalidating TLB entries from stage 2 translations**

The following code is required to invalidate all cached copies of the stage 2 translation of the IPA held in \( X_t \) using the current VMID, with the corresponding requirement applied to the broadcast versions of the instructions:

\[ \text{TLB} \text{ IPAS2E1, } X_t \text{ DSB TLBI VMALE1} \]

The following code is required to invalidate all cached copies of the stage 2 translations of the IPA held in \( X_t \) used to translate the VA, and the specified ASID when executing TLBI VAE1, held in \( X_t \), with the corresponding requirement applied to the broadcast versions of the instructions:

\[ \text{TLB} \text{ IPAS2E1, } X_t \text{ DSB TLBI VAE1, } X_t \text{ or TLBI VAE1, } X_t \]

The following code is required to invalidate all cached copies of the stage 2 translations of the IPA held in \( X_t \) used to translate the IPA produced by the last level of stage 1 translation table lookup for the VA, and the specified ASID when executing TLBI VAE1, held in \( X_t \), with the corresponding requirement applied to the broadcast versions of the instructions:

\[ \text{TLB} \text{ IPAS2E1, } X_t \text{ DSB TLBI VAE1, } X_t \text{ or TLBI VAE1, } X_t \]

For an EL1&0 translation regime with stage 2 translation enabled, software is required to invalidate all cached copies of the stage 2 translations of the IPA held in \( X_t \) used to translate the IPA produced by the last level of stage 1 translation table lookup for the VA, and the specified ASID when executing TLBI VAE1, held in \( X_t \), with the corresponding requirement applied to the broadcast versions of the instructions:

\[ \text{TLB} \text{ IPAS2E1, } X_t \text{ DSB TLBI VAE1, } X_t \]

Equivalent architectural requirements apply to the IPAS2L instruction, except that the only TLB entries required to be invalidated by an IPAS2L instruction are those that come from the final translation table lookup level.

**D8.13.7 Broadcast TLB maintenance**

If all of the following apply, then a TLB maintenance instruction executed by a PE in an Exception level that is using AArch64 can affect a PE that executes the same Exception level using AArch32:

- The PE lies within the target Shareability domain of the TLBI maintenance instruction.
- If VA matching is required, then the VA is 0x0000FFFFFFFF or lower in the memory map.
- If ASID matching is required, then one of the following:
  - If the PE using AArch64 is using an 8-bit ASID, then the ASIDs match.
  - If the PE using AArch64 is using a 16-bit ASID and the top 8 bits of the 16-bit ASID are zero, then the ASIDs match.
- If VMID matching is required, then one of the following:
  - If the PE using AArch64 is using an 8-bit VMID, then the VMIDs match.
  - If the PE using AArch64 is using a 16-bit VMID and the top 8 bits of the 16-bit VMID are zero, then the VMIDs match.

If all of the following apply, then a TLB maintenance instruction executed by a PE in an Exception level that is using AArch32 can affect a PE that executes the same Exception level using AArch64:

- The PE lies within the same Inner Shareable Shareability domain of the TLBI maintenance instruction.
- If VA matching is required, then when the supplied VA is zero-extended, the VAs match.
- If IPA matching is required, then when the supplied IPA is zero-extended, the IPAs match.
- If ASID matching is required, then one of the following:
  - If the PE using AArch64 is using an 8-bit ASID, then the ASIDs match.
  - If the PE using AArch64 is using a 16-bit ASID, then when the supplied ASID is zero-extended, then the ASIDs match.
- If VMID matching is required, then one of the following:
  - If the PE using AArch64 is using an 8-bit VMID, then the VMIDs match.
  - If the PE using AArch64 is using a 16-bit VMID, then when the supplied VMID is zero-extended, the VMIDs match.
If a PE with EL3 using AArch32 issues a broadcast AArch32 TLB maintenance instruction affecting entries in Secure state, then the instruction is not required to affect one or more of the following in the Inner Shareable domain:

- The EL3 translation regime of the PEs with EL3 using AArch64.
- If EL2 is disabled, then the EL1&0 translation regime of the PEs with EL3 using AArch64, regardless of whether the EL1&0 translation regime is using AArch64 or AArch32.

If a PE with EL3 using AArch64 issues a broadcast AArch64 TLB maintenance instruction affecting EL3 entries, then the instruction is not required to affect the EL3 translation regime of the PEs with EL3 using AArch32 in the Inner Shareable domain.

If a PE with EL3 using AArch64 issues a broadcast AArch64 TLB maintenance instruction affecting Secure EL1 entries, then the instruction is not required to affect the EL3 translation regime of the PEs with EL3 using AArch32 in the Inner Shareable domain.

In all of the following cases, a broadcast TLB maintenance instruction is not required to perform any invalidation on the recipient PE within the target Shareability domain:

- A TLB maintenance instruction specifying a VA and affecting one of the following translation regimes is broadcast from a PE using one translation granule size for that translation regime to a PE using a different translation granule size for that VA in the same translation regime:
  - The EL2 translation regime.
  - The EL2&0 translation regime.
  - The EL3 translation regime.

- A TLB maintenance instruction specifying a VA and affecting the EL1&0 translation regime is broadcast from a PE using a stage 1 translation granule size for that translation regime to a PE using EL1 and a different stage 1 translation granule size for that VA.

- If EL2 is enabled, then a TLB maintenance instruction specifying a VA and affecting the EL1&0 translation regime is broadcast from a PE using a stage 2 translation granule size to a PE using a different stage 2 translation granule size.

- If EL2 is enabled, then a TLB maintenance instruction specifying an IPA and affecting the EL1&0 translation regime is broadcast from a PE using a stage 2 translation granule size to a PE using a different stage 2 translation granule size.

- A TLB maintenance instruction specifying a range of VAs or IPAs is broadcast from one PE to a PE that does not support the TLB range-based operations.

The set of Requesters containing TLBs that support TLB range-based operations is defined by the system architecture.

Arm strongly recommends that within an Inner Shareable domain, all PEs support the same set of broadcast TLB range maintenance instructions.

If an Armv7 PE is in the same Inner Shareable domain as an Armv8 PE, and the Armv8 PE issues a broadcast TLB maintenance instruction that is not defined in Armv7, then that instruction is not required to affect the TLBs of the Armv7 PE.

All of the following TLB maintenance instructions are added to the Armv8 T32 and A32 instruction sets, and do not exist in Armv7:

- For stage 1 translations, all of the following instructions that operate on TLB entries for the final level of translation table walk:
  - TLBIMVALID.
  - TLBIMVALID.
  - TLBIMVAAL.
  - TLBIMVALH.
• For stage 2 translations, all of the following instructions that operate by IPA on TLB entries:
  — TLBIIPAS2IS.
  — TLBIIPAS2LIS.
  — TLBIIPAS2.
  — TLBIIPAS2L.

**D8.13.8 Ordering and completion of TLB maintenance instructions**

For all of the following instructions, the effects of a TLB maintenance instruction can be observed in any order relative to that instruction unless a DSB is executed between the instructions:

- Any load or store instruction.
- Another TLB maintenance instruction.
- A data or instruction cache maintenance instruction.

A TLB maintenance instruction executed by a PE causes a TLB maintenance operation to occur on each PE within the Shareability domain specified by the instruction.

If a TLB maintenance instruction has the nXS qualifier, then the associated TLB maintenance operations have the nXS qualifier.

If a TLB maintenance instruction does not have the nXS qualifier and the Effective value of HCRX_EL2.FnXS is 1, then all of the following apply:

- When the instruction is not executed at EL1, the associated TLB maintenance operations do not have the nXS qualifier.
- When the instruction is executed at EL1, the associated TLB maintenance operations have the nXS qualifier.

If a translation regime uses two translation stages, then the XS attribute used to determine the behavior of a TLB maintenance instruction with the nXS qualifier is the attribute determined after both translation stages have been applied.

When a TLB maintenance instruction is executed, **in-scope old translation information** is any translation information related to addresses within the scope of that instruction that is not consistent with one or more of the following:

- The architectural translation information held in the translation tables.
- Any architecture translation information that is Coherence-after the information held in the translation tables.

In-scope old translation information might be held in TLBs or other non-coherent caching structures.

For a PE that handles a TLB maintenance operation without the nXS qualifier, the TLB maintenance operation is finished when all of the following apply:

- All memory accesses generated by that PE using in-scope old translation information are complete.
- All memory accesses, RWx, generated by that PE are complete, where RWx is the set of all memory accesses generated by instructions for that PE that appear in program order before an instruction I1 executed by that PE, where all of the following apply:
  — I1 uses the in-scope old translation information.
  — The use of the in-scope old translation information generates a synchronous Data Abort.
  — If I1 did not generate an abort from use of the in-scope old translation information, I1 would generate a memory access that RWx would be locally-ordered-before.

For a PE that handles a TLB maintenance operation with the nXS qualifier, the TLB maintenance operation is finished when all of the following apply:

- All memory accesses with the XS attribute set to 0 generated by that PE using in-scope old translation information are complete.
• All memory accesses, RWx, generated by that PE are complete, where RWx is the set of all memory accesses generated by instructions for that PE that appear in program order before an instruction I1 executed by that PE, where all of the following apply:
  — I1 uses the in-scope old translation information.
  — The use of the in-scope old translation information generates a synchronous Data Abort.
  — If I1 did not generate an abort from use of the in-scope old translation information, I1 would generate a memory access with the XS attribute set to 0 that RWx would be locally-ordered-before.

IKFTWB For best real-time performance, Arm recommends that the completion of a TLB maintenance instruction with the nXS qualifier executed by a PE should not depend on the completion of any memory accesses with the XS attribute set to 1 generated by a second PE.

RDCJQM When the TLB maintenance operations specified by a TLB maintenance instruction are finished for all PEs, the TLB maintenance instruction is complete.

RLXHLD If a TLB maintenance instruction is complete, then no new memory accesses using the in-scope old translation information are architecturally performed by any observer that is affected by the TLB maintenance instruction.

ICPZSM If it is impossible for software running on any observer to determine that the memory accesses have been performed, then speculative memory accesses can be performed using in-scope old translation information.

RPSMWS When a DSB instruction is used to ensure the completion of TLB maintenance instructions, all of the following apply:
  • The DSB is required to apply to both loads and stores.
  • A DSB is sufficient to ensure completion of TLB maintenance instructions that apply to a single PE.
  • A DSB is sufficient to ensure completion of TLB maintenance instructions that apply to PEs in the same Inner Shareable domain.

RYNJCP When a PE, PEx, executes a TLB maintenance instruction, the instruction can complete at any time after it is issued, but the instruction is only guaranteed to be finished for a PE other than PEx after the execution of DSB by the PEx.

RBLDZX For the TLB of a PE that issues a TLB maintenance instruction, one of the following determines when the instruction is guaranteed to be complete:
  • If FEAT_ETS is not implemented, then the instruction is guaranteed to be complete after the execution of DSB by that PE, followed by a Context synchronization event.
  • If FEAT_ETS is implemented, all of the following determine when the instruction is guaranteed to be complete:
    — If the instruction applies only to translations without execute permission and where translation information that is coherence-after also do not have execute permission, then the instruction is guaranteed to be complete after the execution of DSB by that PE.
    — If the instruction applies to any translations with execute permission, then the instruction is guaranteed to be complete after the execution of DSB by that PE, followed by a Context synchronization event.

IMGRK For more information, see Definition of the Arm memory model on page B2-159.
D8.14 Caches

D8.14.1 Data and unified caches

For data and unified caches, if all data accesses to an address do not use mismatched memory attributes, then the use of address translation is transparent to any data access to the address.

For more information, see Mismatched memory attributes on page B2-208.

For data accesses from the same observer to different VAs, if the accesses are translated to the same PA with the same memory attributes, then they are coherent and all of the following behaviors are guaranteed without requiring the use of barrier or cache maintenance instructions:

- Two writes to the same PA occur in program order.
- A read of a PA returns the value of the last successful write to that PA.
- A write to a PA that occurs, in program order, after a read of that PA, has no effect on the value returned by that read.

When cache maintenance is done to a memory location, the effect of that cache maintenance is visible to all VA or IPA aliases of that physical memory location.

The properties of data and unified caches are consistent with implementing the caches as physically-indexed, physically-tagged caches.

D8.14.2 Instruction caches

An instruction cache has all of the following properties:

- An instruction cache is accessed only as a result of an instruction fetch.
- An instruction cache is never written to by a load or store instruction.
- Cache invalidation is the only cache maintenance that can be done on an instruction cache.

The Arm architecture permits all of the following instruction cache implementations:

- Physically-indexed, physically-tagged (PIPT) instruction cache.
- VMID-aware PIPT (VPIPT) instruction cache.
- Virtually-indexed, physically-tagged (VIPT) instruction cache.

The CTR_EL0.L1Ip field identifies the form of the instruction caches.

For all permitted instruction cache implementations, the Arm IVIPT Extension is also implemented, which reduces instruction cache maintenance requirements to requiring maintenance only after writing new data to a PA that holds an instruction.

Previous versions of the Arm architecture permitted an instruction cache option that does not implement the Arm IVIPT Extension.

For software to be portable between implementations that might use any of the permitted instruction cache implementations, that software is required to invalidate the instruction cache whenever any condition occurs that requires instruction cache maintenance on at least one of the instruction cache types.

Physically-indexed, physically-tagged instruction caches

For PIPT instruction caches, all of the following apply:

- If all instruction fetches to an address do not use mismatched memory attributes, then the use of address translation is transparent to any instruction fetch to the address.
- If instruction cache maintenance is done on a memory location, then the effect of that instruction cache maintenance is visible to all VA or IPA aliases of that physical memory location.
VMID-aware PIPT instruction caches

All statements in this section require implementation of FEAT_VPIPT.

For VPIPT instruction caches, all of the following apply:

- If all instruction fetches to an address do not use mismatched memory attributes, then the use of address translation is transparent to any instruction fetch to the address.
- If instruction cache maintenance is done on a memory location, then the effect of that instruction cache maintenance is visible to all VA or IPA aliases of that physical memory location.
- For EL1 and EL0, if VMIDs are being used for the current Security state, all of the following apply:
  - Only if an instruction fetch is made using the VMID that was used when an instruction cache entry was created, then the instruction fetch is permitted to hit that entry in the cache.
  - Only if the current VMID matches the VMID that was used when an instruction cache entry was created, then an instruction cache maintenance instruction executed at EL1 or EL0 is required to apply to the instruction cache entry.

The first two behaviors of VPIPT instruction caches are the same as the behaviors of PIPT instruction caches.

Virtually-indexed, physically-tagged instruction caches

For VIPT instruction caches, all of the following apply:

- If all instruction fetches to an address do not use mismatched memory attributes, then the use of address translation is transparent to any instruction fetch to the address.
- If instruction cache maintenance by address is done on a memory location, then the effect of that instruction cache maintenance is visible only to the VA supplied with the operation, and the effect of the invalidation might not be visible to any other VA or IPA aliases of that physical memory location.
- The only architecturally-guaranteed way to invalidate all VA or IPA aliases of a PA from a VIPT instruction cache is to invalidate the entire instruction cache.

Cache maintenance requirements due to changing memory region attributes

If changes are made to the memory region attributes in translation table entries, TLB maintenance might be required.

For more information, see TLB maintenance on page D8-5197.

The behaviors caused by mismatched memory attributes mean that if any of the following changes are made to the Inner Cacheability or Outer Cacheability attributes in translation table entries, then software is required to ensure that any cached copies of affected locations are removed from the caches, typically by cleaning and invalidating the locations from the cache levels that might hold copies of the locations affected by the attribute change:

- A change from Write-Back to Write-Through.
- A change from Write-Back to Non-cacheable.
- A change from Write-Through to Non-cacheable.
- A change from Write-Through to Write-Back.

For more information, see Mismatched memory attributes on page B2-208 and Using break-before-make when updating translation table entries on page D8-5198.

If the Shareability attributes in translation table entries are changed, then software is required to execute the following sequence to prevent possible coherency errors caused by mismatched memory attributes:

1. Make the memory location Non-cacheable, Outer Shareable.
2. Clean and invalidate the memory location from caches in the appropriate Shareability domain.
3. Change the Shareability attributes to the new values.
The Arm architecture defines pseudocode that describes the translation table walk.

**Full Physical Address**
A complete PA necessary to identify a location in physical memory is captured by the type `FullAddress`. This is composed of:
- A bitstring address, which identifies the PA.
- An enumeration `paspace` which identifies the PA space.

**Address translation**
`AArch64.TranslateAddress()` acts as the entry point to VMSAv8-64 and performs the required address translation based on the provided parameters and System register configurations. The function returns an `AddressDescriptor()` structure holding valid data for either of the following:
- Target memory address and attributes for a non-faulting translation.
- Fault details holding data to be populated in syndrome registers.

`AArch64.FullTranslate()` selects the translation regime and performs first and potentially second stage of translation returning the PA and attributes of target memory. `AArch64.S1Translate()` carries out the first stage of translation when stage 1 is not disabled, mapping the VA to the IPA and carrying out permission checks. Otherwise, `AArch64.S1DisabledOutput()` assigns the appropriate memory attributes and flat maps the input address to the output address. `AArch64.S2Translate()` carries out stage 2 translation for the EL1&0 translation regime when enabled mapping the IPA to the PA. Otherwise, the IPA is the PA.

**Translation table walk**
Each stage of translation has a separate walk function, `AArch64.S1Walk()` and `AArch64.S2Walk()`, corresponding to the first and second stage of translation respectively. Each use walk parameters extracted from related System registers. Parameters are collected based on the active translation regime. For instance, stage 1 EL2 translation regime parameters are obtained and returned by the function `AArch64.S1TTWParamsEL2()`. Given these parameters, a walk initializes a walk state of the type `TTWState`, holding the base address of the first translation table.

The walk progressively fetches and decodes translation Table descriptors, updating the walk state to the next base address as it descends through the levels of tables until a Block or Page descriptor is discovered or an invalid descriptor is fetched. Decoding the descriptor for both stage 1 and stage 2 walks is carried out by the function `AArch64.DecodeDescriptorType()`.

For a non-faulting walk, three items are returned by a translation table walk:
- The final walk state.
- The final descriptor fetched.
- The address of the final descriptor.

The final descriptor and its address are used to update the descriptor as specified by `Hardware management of the Access flag on page D8-5144` and `Hardware management of the dirty state on page D8-5145`.

A faulting walk could report one of the following at a specified level:
- Translation Fault.
- Address size Fault.
- Access flag Fault.

**Hardware update of Translation Table descriptors**
The walk parameters collected from System registers indicate the ability to update the access flag or set write permissions within descriptors. This is controlled by the Dirty Bit Modifier, and the conditions specified in `Hardware management of the Access flag on page D8-5144` and `Hardware management of the dirty state on page D8-5145`. The translation functions `AArch64.S1Translate()` or `AArch64.S2Translate()` set the appropriate descriptor bits returned by the walk functions and call `AArch64.MemSwapTableDesc()` to swap the old descriptor for the updated one in an atomic fashion.
Address decoding and calculation

The walk state is initialized to hold the base address of the first translation table, using AArch64.TTBaseAddress() to decode TTBR0_ELx and TTBR1_ELx registers. The walk progressively fetches and decodes translation Table descriptors, updating the walk state to the next base address utilizing AArch64.NextTableBase() as it descends through the levels of tables. Prior to every descriptor fetch the base address is indexed by the function AArch64.TTEntryAddress() to point to the specific table entry. Indexing at the start level of stage 2 tables is shown in AArch64.S2SLTTEntryAddress() which caters for concatenated tables. The final walk state would hold the base address for the output block or page; this is extracted from the Leaf descriptor in AArch64.BlockBase() or AArch64.PageBase() respectively.

Memory attribute decoding

If a stage of translation is enabled, fetched descriptors that are blocks or pages encode memory attributes assigned to the output of translation. Stage 1 memory attributes are decoded by the function S1DecodeMemAttrs(). Likewise, the stage 2 memory attributes are decoded by the function S2DecodeMemAttrs() followed by combining stage 1 and stage 2 attributes by the function S2CombineS1MemAttrs(). However, if FEAT_S2FWB is enabled, this behavior is overridden and memory attributes are decoded as specified in Stage 2 memory type and Cacheability attributes when FEAT_S2FWB is enabled on page D8-5155. This is captured by the function AArch64.S2ApplyFWBMemAttrs().

Fault detection

As soon as translation is invoked a reserve FaultRecord accompanies the process capturing the stage and level of translation as it proceeds. When a fault is detected, it is reflected in the FaultRecord and reported back as the result of translation with the most recent state to be reported already captured within. The following functions detect a certain type of fault, their outputs are all Boolean with a TRUE value on detection:

- The AArch64.S1HasPermissionsFault() and AArch64.S2HasPermissionsFault() function detects a permissions fault for stage 1 and stage 2 respectively. Note: For atomic instructions introduced by FEAT_LSE, these functions are called twice, once to check for read permissions and another for write allowing the correct failure to be reported.
- The AArch64.S1HasAlignmentFault() and AArch64.S2HasAlignmentFault() functions detect an alignment fault for stage 1 and stage 2 respectively.
- The AArch64.S1InvalidTxSZ() and AArch64.S2InvalidTxSZ() functions detect a Translation fault caused by erroneous configuration of TCR_ELx.TxSZ field. Also, the AArch64.S2InconsistentSL() and AArch64.S2InvalidSL() functions detect a stage 2 Translation fault caused by erroneous configuration of the VTCR_EL2.{SL2, SL0} and VSTCR_EL2.{SL2, SL0} fields.
- AArch64.VAIsOutOfRange() detects a stage 1 Translation fault caused by VAs larger than the address input size configured. Similarly, AArch64.IPAIsOutOfRange() detects a stage 2 Translation fault caused by the output of stage 1 being larger than the configured input size for stage 2.
- AArch64.ContiguousBitFaults() detects a stage 1 or 2 Translation fault caused by a mis-programmed contiguous bit within a fetched descriptor.

For more information, see aarch64/translation on page J1-11212.
Chapter D9
The Memory Tagging Extension

This chapter describes the Memory Tagging Extension. It contains the following sections:

• Introduction on page D9-5220.
• Allocation Tags on page D9-5221.
• Tag checking on page D9-5223.
• Tagged and Untagged Addresses on page D9-5224.
• PE access to Allocation Tags on page D9-5225.
• Enabling the Memory Tagging Extension on page D9-5226.
• PE handling of Tag Check Fault on page D9-5227.
• PE generation of Tag Checked and Tag Unchecked accesses on page D9-5229.
D9.1 Introduction

There are three versions of the Memory tagging extension:

**FEAT_MTE**

FEAT_MTE supports the Memory tagging instructions accessible in EL0.

When FEAT_MTE is implemented:
- A set of tag load and tag store instructions are provided.
- Instructions to generate and insert Logical Tags in addresses are provided.
- System instructions to Clean, and Clean and Invalidate Allocation Tags from caches are provided.
- If FEAT_MTE2 is not implemented, all System instructions defined by the Memory Tagging Extension are UNDEFINED.
- If FEAT_MTE2 is not implemented, all operations which read Allocation Tags treat the Allocation Tag as zero, and any traps or permission checks continue to apply.
- If FEAT_MTE2 is not implemented, instructions which insert Allocation Tags into addresses treat the Allocation Tag as zero.
- If FEAT_MTE2 is not implemented, the Tagged memory type encoding in the Memory Attribute Indirection Registers are UNPREDICTABLE.

When FEAT_MTE and FEAT_MOPS are implemented, the SETG* instructions are also provided. For more information, see Memory Copy and Memory Set instructions on page C3-274.

**FEAT_MTE2**

FEAT_MTE2 supports all instructions and System registers defined by the extension, Allocation Tags in memory, and Tag Checking of accesses to tagged memory.

When FEAT_MTE2 is implemented:
- All FEAT_MTE functionalities are available for use.
- System register and page level control over access to Allocation Tags in memory is provided.
- Allocation Tags are provided for each 16-byte granule of Conventional memory.
- The tag PA space is separate to the data physical address (data PA) space accessed by data load and store instructions to access data in normal memory and devices.
- Any associated fields in System control registers are available for use.
- All System registers defined by the extension become available for use.
- All System instructions and instructions defined by the extension become available for use.

**FEAT_MTE3**

FEAT_MTE3 adds support for asymmetric Tag Check Fault handling.

When FEAT_MTE3 is implemented:
- All FEAT_MTE and FEAT_MTE2 functionalities are available for use.
- Tag Check Faults can be configured to cause a synchronous exception on reads, and be asynchronously accumulated on writes.
- Any Tag Check Fault on an access that performs both a read and a write can be configured to cause a synchronous exception.
D9.2 Allocation Tags

The tag PA space provides access to Allocation Tags stored in memory. The data PA space provides access to data held in memory.

An Allocation Tag is 4 bits wide.

Each naturally-aligned set of 16 tag PA space locations is a Tag Granule. Each Tag Granule is associated with one Allocation Tag.

——— Note ————
The value 0b1111 may incur a higher performance overhead than other Allocation Tag encodings.

If FEAT_MTE2 is implemented, storage is provided for Allocation Tags at each tag PA where Conventional memory exists at the same physical address in the data PA space.

——— Note ————
Arm recommends that software does not use instructions which write 0b1111 as an Allocation Tag to memory.

The result of an access to the tag PA where Allocation Tag storage is not provided is IMPLEMENTATION DEFINED.

It is IMPLEMENTATION DEFINED whether Allocation Tags are permitted to be accessed through regions of the data PA space. If Allocation Tags are accessible through the data PA space, then the layout of Allocation Tags is IMPLEMENTATION DEFINED.

It is not architecturally required for an Allocation Tag accessed via the tag PA space to be coherent with the same Allocation Tag accessed via the data PA space. A write to one location can be made visible at the other location by the use of the cache maintenance operations.

Unless otherwise stated, the definitions in Chapter B2 The AArch64 Application Level Memory Model and Chapter D7 The AArch64 System Level Memory Model that apply to data accesses and data, apply separately to Allocation Tag accesses and Allocation Tags.

It is IMPLEMENTATION DEFINED whether accesses to the tag PA space are monitored by the global monitor.

D9.2.1 Cache activity and Allocation Tags

When data is evicted from a cache entry at a cache level, the evicted data can overwrite data in memory that has been written by another observer if either, or any of the following are true:

• The data has been written by an observer in the Shareability domain of that memory location, where the maximum size of the memory that can be overwritten is defined by the Cache Write-Back Granule in CTR_EL0.
• The associated Allocation Tags have been written to by an observer in the Shareability domain of that memory location, where the maximum size of the memory that can be overwritten is defined by the Cache Write-Back Granule in CTR_EL0.

When Allocation Tags are evicted from a cache entry at a cache level, the evicted Allocation Tags can overwrite Allocation Tags in memory that has been written by another observer if either, or any of the following are true:

• The Allocation Tags associated with memory within an address range of the size of the Cache Write-Back Granule, aligned to that size, have been written to by an observer in the Shareability domain of that memory location.
• Data within an address range of the size of Cache Write-Back Granule in CTR_EL0, aligned to that size, of the associated data has been written to by an observer in the Shareability domain of that memory location.

If an implementation can overwrite Allocation Tags in memory that have been written by another observer, where the Allocation Tags have not been written by an observer in the Shareability domain of that memory location, then:

• A cache maintenance operation which cleans data from a cache level must also clean the associated Allocation Tags.
A cache maintenance operation which invalidates, or cleans and invalidates data from a cache level, must also clean and invalidate the associated Allocation Tags.

**Note**

If an implementation supports both clean write-back of Allocation Tags, and storage for Allocation Tags can be repurposed to store data when not being used to store Allocation Tags, it is an implementation responsibility to prevent the clean write-back of Allocation Tags from overwriting data.

For more information on DC operations that affect Allocation Tags see *A64 System instructions for cache maintenance* on page C5-776.
D9.3 Tag checking

A memory access that is a read or write can be either Tag Checked or Tag Unchecked. An access to the data PA space can be either Tag Checked or Tag Unchecked. An access to the tag PA space is always Tag Unchecked. A data access which is performed as part of a prefetch operation is Tag Unchecked. When the value of PSTATE.TCO is 1, all loads and stores are Tag Unchecked. A Tag Checked memory access includes a Physical Address Tag. A Tag Checked access causes a Tag Check operation to be performed.

If the Allocation Tag and Physical Address Tag in a Tag Check operation do not match, the Tag Check operation generates a Tag Check Fault. The read of an Allocation Tag due to a Tag Check operation, and the dependent data access, are not required to form an atomic operation.

Software cannot rely on a Load-Exclusive/Store-Exclusive pair to eventually succeed if the Tag Checked properties of the following in a Load-Exclusive/Store-Exclusive instruction pair accessing the same location from the same PE do not match:
• A memory access due to a Store-Exclusive instruction.
• A memory access due to the preceding Load-Exclusive instruction.

D9.3.1 Tag Check Faults

A Tag Check Fault can be configured to cause one of the following:
• A synchronous exception.
• A bit to be asynchronously set in TFSR_ELx.
• To be ignored.

If a store causes a synchronous Tag Check Fault exception, the faulting memory locations being written to by the store that caused the fault are unchanged.

If a Tag Check Fault is not configured to cause a synchronous exception then the following are true:
• There is no effect on the data access, that is the load or store completes unless another exception is taken.
• There is no effect on any of the side effects caused by the completion of the data access.

If FEAT_MTE2 is implemented, a synchronous exception due to a Tag Check Fault is reported as a Data Abort with a Data Fault Status Code of Synchronous Tag Check Fault and the faulting virtual address is reported in FAR_ELx. For more information, see PE handling of Tag Check Fault on page D9-5227.
D9.4 Tagged and Untagged Addresses

Virtual addresses can either be Tagged or Untagged.

An access to memory at:
• An Untagged virtual address generates a Tag Unchecked access.
• A Tagged virtual address permits the generation of a Tag Checked or Tag Unchecked access.

A read of an Allocation Tag from an Untagged virtual address returns the value 0b0000.

A write of an Allocation Tag to an Untagged address is IGNORED.

Accesses of Allocation Tags at Tagged virtual addresses are permitted.

All virtual addresses in AArch32 state are Untagged.

D9.4.1 Virtual address translation

If stage 1 translation at the current Exception level is enabled, stage 1 translations are Tagged or Untagged depending on the Memory Attributes for the memory location being accessed.

If stage 1 translation is disabled for the EL1&0 translation regime:
• If the value of HCR_EL2.DC is 1, stage 1 translations are Tagged or Untagged depending on the value of HCR_EL2.DCT.
• If the value of HCR_EL2.DC is 0, stage 1 translations are treated as Untagged.

For all other translation regimes, if stage 1 translation is disabled, stage 1 translations are treated as Untagged.

Memory locations are treated as Tagged where all of the following is true:
• The combined effects of stage 1 and stage 2 translations define the memory attributes as:
  — Normal memory.
  — Inner, and Outer Write-Back Non-Transient Read-Allocate Write-Allocate.
• The stage 1 translation is treated as Tagged.

Otherwise memory locations are Untagged.

If a memory location is marked as Untagged, a data cache invalidation operation that would invalidate Allocation Tags at that location cleans and invalidates the Allocation Tags.

Note
If a memory location is marked as both Tagged and Non-shared, it is IMPLEMENTATION DEFINED whether the memory location is treated as Tagged or Untagged.

When the EL1&0 stage 1 translation regime is disabled and HCR_EL2.DC is 1, in the current Security state, the execution of any of the AT S1E0, AT S1E1, AT S1Z0, AT S1Z1 address translation instructions will reflect the effect of HCR_EL2.DCT in PAR_EL1.ATTR.

If SCTLR_ELx.C is 0 for a stage 1 translation regime, it is CONstrained Unpredictable between:
• The stage 1 translation is treated as Untagged.
• SCTLR_ELx.C has no effect on whether the stage 1 translation is treated as Tagged or Untagged.

Note
To ensure consistent behavior, software can set SCTLR_ELx.ATA to 0 when SCTLR_ELx.C is 0.

For more information on Virtual address translation, see Address translation on page D8-5080.
D9.5 PE access to Allocation Tags

Instructions that load or store Allocation Tags apply the same address translation and permission checks as a load or store of data to a virtual address.

Instructions that load or store Allocation Tags at a virtual address have the same effect on the Access flag and dirty state as instructions that load or store data at the same virtual address.

An instruction that loads or stores an Allocation Tag:

- Is considered a load or store of data to each location associated with the Allocation Tag for the purpose of triggering Watchpoints and PMU events, other than for events which count bytes of data transferred.
- Is treated as a load or store for the purpose of Statistical Profiling.
- Generates a tag PA with the same physical address as a load or store of data to a virtual address.

Instructions that store Allocation Tags to memory locations marked as Device memory result in a CONstrained UNpredictable choice between:

- Storing the data, if any, to the specified locations.
- Generating an Alignment Fault, which is prioritized in the same way as other alignment faults that are determined by the memory type.

DC GZVA and DC GVA are instructions that store Allocation tags.

Instructions which load or store Allocation tags are considered to perform the access for the purposes of address translation, triggering watchpoints, generating PMU events and Statistical Profiling irrespective of whether access to the Allocation tags in memory is disabled. This includes:

- When FEAT_MTE2 is not implemented.
- Access to Allocation tags in memory is disabled due to Allocation tag access controls in HCR_EL2, SCR_EL3 and SCTLR_ELx.
- The Tagged attribute is absent on the locations being accessed.

A read of an Allocation Tag that returns zero due to access to Allocation tags being disabled by HCR_EL2.ATA, SCR_EL3.ATA or SCTLR_ELx. {ATA, ATA0}, or due to the memory type not having the Tagged attribute, is permitted to generate an External abort if a read of data from the same address would generate an External abort.

For more information on which instructions can be used to access an Allocation tag, see:

- Loads and stores on page C3-256, for load and store instructions.
- Branches, Exception generating, and System instructions on page C3-248, for System instructions.
D9.6 Enabling the Memory Tagging Extension

Access to Allocation Tags in memory can be enabled by use of the following controls:

- SCR_EL3.ATA.
- HCR_EL2.ATA.
- SCTLR_ELx.ATA.
- SCTLR_ELx.ATA0.

When executed at an exception level where accesses to Allocation Tags are disabled, instructions that:

- Load or store data, are Unchecked.
- Load or store Allocation Tags treat the Allocation Tag as RAZ/WI.
- Insert Logical Address Tags into addresses treat the Allocation Tag used to generate the Logical Address Tag as zero,
- Invalidate Allocation Tags from caches, behave as the equivalent Clean and Invalidate operation on Allocation Tags.

For the purpose of determining Allocation Tag access, unprivileged load and store instructions are treated as if executed at EL0 when executed at either:

- EL1, when the Effective value of PSTATE.UAO is 0.
- EL2, when both the Effective value of HCR_EL2.{E2H, TGE} is {1, 1} and the Effective value of PSTATE.UAO is 0.

Note

Arm recommends that:

- When software requires access to Allocation Tags in a context but Tag Checking is not required, the SCTLR_ELx.TCF or SCTLR_ELx.TCF0 affecting that context is set to 0.
- When software does not require access to Allocation Tags in a context, one or more SCTLR_ELx.ATA affecting that context are set to 0.
D9.7 PE handling of Tag Check Fault

If SCTLR_ELx.TCF has the value of 0b00, a Tag Check Fault due to a load or store at ELx has no effect on the PE.

A Tag Check Fault due to a load or store at EL0 has no effect on the PE if either of the following conditions are true:

- SCTLR_EL1.TCF0 has the value of 0b00, and HCR_EL2.{E2H, TGE} does not have the value of {1,1}.
- SCTLR_EL2.TCF0 has the value of 0b00, and HCR_EL2.{E2H, TGE} has the value of {1,1}.

If SCTLR_ELx.TCF has the value of 0b01, a Tag Check Fault due to a load or store at ELx generates a synchronous exception.

If SCTLR_ELx.TCF has the value of 0b10, a Tag Check Fault due to a load or store at ELy using TTBR_ELx causes TFSR_ELy.TFx to be asynchronously set to 1.

If FEAT_MTE3 is implemented, and SCTLR_ELx.TCF has the value of 0b11, a Tag Check Fault due to a load, or an atomic operation, generates a synchronous exception.

If FEAT_MTE3 is implemented, and SCTLR_ELx.TCF has the value of 0b11, a Tag Check Fault due to a store at ELy using TTBR_ELx causes TFSR_ELy.TFx to be asynchronously set to 1.

A Tag Check Fault due to a load or store at EL0 has no effect on the PE if either of the following conditions are true:

- SCTLR_EL1.TCF0 has the value of 0b00, and HCR_EL2{E2H, TGE} does not have the value of {1,1}.
- SCTLR_EL2.TCF0 has the value of 0b00, and HCR_EL2{E2H, TGE} has the value of {1,1}.

A Tag Check Fault due to a load or store at EL0 generates a synchronous exception if either of the following conditions are true:

- SCTLR_EL1.TCF0 has the value of 0b01, and HCR_EL2{E2H, TGE} does not have the value of {1,1}.
- SCTLR_EL2.TCF0 has the value of 0b01, and HCR_EL2{E2H, TGE} has the value of {1,1}.

A Tag Check Fault due to a load or store at EL0 using TTBRy_EL1 or TTBRy_EL2, causes TFSRE0_EL1.TFy to be set to 1 if either of the following conditions are true:

- SCTLR_EL1.TCF0 has the value of 0b10, and HCR_EL2{E2H, TGE} does not have the value of {1,1}.
- SCTLR_EL2.TCF0 has the value of 0b10, and HCR_EL2{E2H, TGE} has the value of {1,1}.

If FEAT_MTE3 is implemented, a Tag Check Fault due to a load, or an atomic operation, at EL0 generates a synchronous exception, if either of the following conditions are true:

- SCTLR_EL1.TCF0 has the value of 0b11, and HCR_EL2{E2H, TGE} does not have the value of {1,1}.
- SCTLR_EL2.TCF0 has the value of 0b11, and HCR_EL2{E2H, TGE} has the value of {1,1}.

If FEAT_MTE3 is implemented, a Tag Check Fault due to a store at EL0 using TTBRy_EL1 or TTBRy_EL2 causes TFSRE0_EL1.TFy to be set to 1, if either of the following conditions are true:

- SCTLR_EL1.TCF0 has the value of 0b11, and HCR_EL2{E2H, TGE} does not have the value of {1,1}.
- SCTLR_EL2.TCF0 has the value of 0b11, and HCR_EL2{E2H, TGE} has the value of {1,1}.

TFSR_ELx and TFSRE0_EL1 are unchanged by a memory data access causing a Tag Check pass.

A synchronous exception due to a Tag Check Fault is reported as a Data Abort, with a Data Fault status code of Synchronous Tag Check Fault, and the faulting virtual address is reported in FAR_ELx.

A Data Abort due to a Tag Check Fault is taken from EL0 to one of the following exception levels:

- EL1 if HCR_EL2.TGE is 0.
- EL2 if HCR_EL2.TGE is 1.

A Data Abort due to a Tag Check Fault is taken from ELx to ELx where x is 1, 2 or 3.

A Data Abort due to a Tag Check Fault is prioritized as a Data Abort exception generated by a synchronous External abort that was not generated by a translation table walk.

If an access generates both a Data Abort due to a Synchronous Tag Check Fault, and a Data Abort due to a synchronous External abort that was not generated by a translation table walk, it is IMPLEMENTATION DEFINED which abort is reported. For more information on prioritization of exceptions see Synchronous exception types on page D1-4646.
If an instruction that accesses memory generates both a synchronous Data Abort and a Tag Check Fault, where Tag Check Faults are configured to set a bit in a TFSR_ELx or TFSRE0_EL1 register, and the bit is 0, the bit becomes UNKNOWN.

If an instruction that stores to memory generates a Data Abort that is a Synchronous Tag Check Fault, the value of each memory location that the instruction stores to is UNKNOWN for any location for which no exceptions, and no Debug event is generated. The size of a memory location is defined as being the size for which a memory access is single-copy atomic.

For the purpose of determining Tag Check Fault handling, unprivileged load and store instructions are treated as if executed at EL0 when executed at either:

- EL1, when the Effective value of PSTATE.UAO is 0.
- EL2, when both the Effective value of HCR_EL2.[E 2H, TGE] is {1, 1} and the Effective value of PSTATE.UAO is 0.

Indirect writes to TFSRE0_EL1 and any TFSR_ELx accessible at ELy that are caused by a Tag Check Fault are synchronized by any of:

- An exception entry to ELy, if SCTLR_ELy.ITFSB has the value of 0b1.
- A DSB over the Non-shareable domain at ELy in program order, after the instruction causing the Tag Check Fault.

When FEAT_SVE is implemented, if a load of an element in a SVE Non-faulting or First-faulting load instruction causes a Tag Check Fault, and is not the First active element in a First-faulting instruction, the Tag Check Fault:

- Is recorded in the corresponding FFR register.
- Does not generate a Synchronous Tag Check Fault exception.
- Does not cause any bit in any TFSR_ELx or TFSRE0_EL1 registers to be set.
- The value loaded into the element is UNKNOWN.

When FEAT_SVE is implemented, if a load of an element in a SVE Non-faulting or First-faulting load instruction causes a Tag Check Fault, and is the First active element in a First-faulting instruction, the Tag Check Fault:

- Is not recorded in the corresponding FFR register.
- Generates a Synchronous Tag Check Fault exception if configured to do so.
- Sets a bit in TFSR_ELx or TFSRE0_EL1 registers if configured to do so.
- If a synchronous Tag Check Fault is generated, the value loaded into the element is UNKNOWN.

It is CONSTRAINED UNPREDICTABLE whether the FFR element associated with the read of an Active element in an SVE Non-fault load, or an Active element which is not the First active element in an SVE First-fault load, R2, to location X, is set to FALSE if all of the following are true:

- Tag Check Faults are configured as asynchronous for both reads and writes.
- A read or write RW1 to location Y causes a Tag Check Fault.
- Tag Check Faults for locations X and Y are reported in the same status bit, either:
  - TFSR_ELx.TFy.
  - TFSRE0_EL1.TFy.
- RW1 is in program order before R2, or is the First active element in the first-fault load instruction causing R2.
- There are no other faults caused by R2 that are reported in FFR.
- There is not a DSB and a direct write of 0b0 to that status bit appearing in program order between the instruction causing RW1 and the instruction causing R2.

If a Tag check fault configured to asynchronously set a bit in TFSR_ELx or TFSRE0_EL1 has not been observed in TFSR_ELx or TFSRE0_EL1, and has not been synchronized by a DSB, or by the effects of SCTLR_ELx.ITSB, when a write to a System register affecting Tag check fault handling occurs, the effect of the Tag check fault is CONSTRAINED UNPREDICTABLE between the effect of the old and new value of the System register.
D9.8 **PE generation of Tag Checked and Tag Unchecked accesses**

A Logical Address Tag is formed by bits [59:56] of the 64-bit address that is used for a load or store instruction. The PE generates a Physical Address Tag from the Logical Address Tag for each Tag Checked access to memory. Unless an access is explicitly defined as a Tag Unchecked access, it is a Tag Checked access.

Instructions in Debug state follow the same rules for generation of Tag Checked and Tag Unchecked accesses as in Non-Debug state. See Chapter H2 [Debug State](#) for more information.

### D9.8.1 Tag Unchecked accesses

The following operations generate a Tag Unchecked access:

- An instruction fetch.
- A load instruction that loads an Allocation Tag.
- A store instruction that stores an Allocation Tag.

When PSTATE.TCO is 1, all loads and stores generate Tag Unchecked accesses.

An access due to a translation table walk generates a Tag Unchecked access.

If FEAT_NV2 is implemented, loads and stores relative to VNCR_EL2 generate a Tag Unchecked access.

If the Statistical Profiling Extension is implemented, all accesses to the Profiling Buffer are Tag Unchecked accesses. See Chapter D13 [The Statistical Profiling Extension](#) for more information.

An access which would be translated using TTBR0_ELx is Tag Unchecked, irrespective of whether the stage 1 address translation for the ELx translation regime is enabled or not, where either of the following conditions apply:

- TCR_ELx.TBI is 0.
- TCR_ELx.TBIO is 0.

If TCR_ELx.TBI1 has the value of zero, an access which would be translated using TTBR1_ELx is Tag Unchecked, irrespective of whether the stage 1 address translation for the ELx translation regime is enabled or not.

An access will be Tag Unchecked, irrespective of whether the stage 1 address translation for the ELx translation regime is enabled or not, where all of the following conditions apply:

- The access would be translated using TTBR0_ELx.
- The Logical Address Tag is 0b0000.
- TCR_ELx.TCMA is 1, or TCR_ELx.TCMA0 is 1.

An access will be Tag Unchecked, irrespective of whether the stage 1 address translation for the ELx translation regime is enabled or not, when all of the following conditions apply:

- The access would be translated using TTBR1_ELx.
- The Logical Address Tag is 0b1111.
- TCR_ELx.TCMA1 is 1.

A Tag Unchecked access will be generated for a load or store that uses either of the following:

- A base register only, with the SP as the base register.
- A base register plus immediate offset addressing form, with the SP as the base register.

Literal (PC-relative) loads generate a Tag Unchecked access.

### D9.8.2 Constrained Unpredictable behavior

When executing a Store-Exclusive instruction, that if Tag Unchecked would not perform the store, and would return a status value of one, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction generates a Tag Checked access.
- The instruction generates a Tag Unchecked access.
The Memory Tagging Extension

D9.8 PE generation of Tag Checked and Tag Unchecked accesses
Chapter D10
The Generic Timer in AArch64 state

This chapter describes the implementation of the Arm Generic Timer. It includes an overview of the AArch64 System register interface to an Arm Generic Timer.

It contains the following sections:

• About the Generic Timer on page D10-5232.
• The AArch64 view of the Generic Timer on page D10-5236.

Chapter G6 The Generic Timer in AArch32 state describes the AArch32 view of the Generic Timer, and Chapter I2 System Level Implementation of the Generic Timer describes the system level implementation of the Generic Timer.
D10.1 About the Generic Timer

Figure D10-1 shows an example system-on-chip that uses the Generic Timer as a system timer. In this figure:

- This manual defines the architecture of the individual PEs in the multiprocessor blocks.
- The ARM Generic Interrupt Controller Architecture Specification defines a possible architecture for the interrupt controllers.
- Generic Timer functionality is distributed across multiple components.

The Generic Timer:

- Provides a system counter that measures the passing of time in real-time.

--- Note ---

The Generic Timer can also provide other components at a system level, but Figure D10-1 does not show any such components.

- Supports virtual counters that measure the passing of virtual-time. That is, a virtual counter can measure the passing of time on a particular virtual machine.
- Can trigger events after a period of time has passed. The timers:
  - Can be used as count-up or as count-down timers.
  - Can operate in real-time or in virtual-time.

This chapter describes an instance of the Generic Timer component that Figure D10-1 shows as Timer_0 or Timer_1 within the Multiprocessor A or Multiprocessor B block. This component can be accessed from AArch64 state or AArch32 state, and this chapter describes access from AArch64 state. Chapter G6 The Generic Timer in AArch32 state describes access to this component from AArch32 state.

A Generic Timer implementation must also include a memory-mapped system component. This component:

- Must provide the System counter shown in Figure D10-1.
- Optionally, can provide timer components for use at a system level.

Chapter 12 System Level Implementation of the Generic Timer describes this memory-mapped component.
D10.1.1 The full set of Generic Timer components

Within a system that might include multiple PEs, a full set of Generic Timer components is as follows:

The system counter

This provides a uniform view of system time, see The system counter on page D10-5234. Because this must be implemented at the system level, it is accessed through The system level memory-mapped implementation of the Generic Timer. However, during initialization, a status register in each implemented timer in the system must be programmed with the frequency of the system counter, so that software can read this frequency.

PE implementations of the Generic Timer

Each PE implementation of the Generic Timer provides the following components:

• A physical counter, which gives access to the count value of the system counter. When FEAT_ECV is implemented, the CNTPOFF_EL2 register allows offsetting of physical timers and counters.

• A virtual counter, which gives access to virtual time. In AArch64 state, the CNTVOFF_EL2 register defines the offset between physical time, as defined by the value of the system counter, and virtual time.

• A number of timers. In an implementation where all Exception levels are implemented and can use AArch64 state, the timers that are accessible from AArch64 state are:
  — An EL1 physical timer.
  — A Non-secure EL2 physical timer.
  — An EL3 physical timer.
  — An EL1 virtual timer.
  — A Non-secure EL2 virtual timer.
  — A Secure EL2 virtual timer.
  — A Secure EL2 physical timer.

The Non-secure EL2 virtual timer is available only when FEAT_VHE is implemented.

The Secure EL2 timers are available only when FEAT_SEL2 is implemented.

The AArch64 view of the Generic Timer on page D10-5236 describes these components.

The system level memory-mapped implementation of the Generic Timer

The memory-mapped registers that control the components of the system level implementation of the Generic Timer are grouped into frames. The Generic Timer architecture defines the offset of each register within its frame, but the base address of each frame is IMPLEMENTATION DEFINED, and defined by the system.

Each system level component has one or two register frames. The possible system level components are:

The memory-mapped counter module, required

This module controls the system counter. It has two frames:

• A control frame, CNTControlBase.
• A status frame, CNTReadBase.

The memory-mapped timer control module, required

The system level implementation of the Generic Timer can provide up to eight timers, and the memory-mapped timer control module identifies:

• Which timers are implemented.
• The features of each implemented timer.

This module has a single frame, CNTCTLBase.

Memory-mapped timers, optional

An implemented memory-mapped timer:

• Must provide a privileged view of the timer, in the CNTBaseV frame.
D10.1 About the Generic Timer

- Optionally, provides an unprivileged view of the timer in the CNTELOBaseN frame.

\[ N \text{ is the timer number, and the corresponding frame number, in the range 0-7.} \]

Chapter 12 System Level Implementation of the Generic Timer describes these components.

D10.1.2 The system counter

The Generic Timer provides a system counter with the following specification:

**Width**
- From Armv8.0 to Armv8.5 inclusive, at least 56 bits wide. The value returned by any 64-bit read of the counter is zero-extended to 64 bits.
- From Armv8.6, must be 64 bits wide.

**Frequency**
- From Armv8.0 to Armv8.5 inclusive, increments at a fixed frequency, typically in the range 1-50MHz. It can support one or more alternative operating modes in which it increments by larger amounts at a lower frequency, typically for power-saving.
- From Armv8.6, increments at a fixed frequency of 1GHz.

**Roll-over**
- Roll-over time of not less than 40 years.

**Accuracy**
- Arm does not specify a required accuracy, but recommends that the counter does not gain or lose more than ten seconds in a 24-hour period.
- Use of lower-frequency modes must not affect the implemented accuracy.

**Start-up**
- Starts operating from zero.

The system counter, once configured and running, must provide a uniform view of system time. More precisely, it must be impossible for the following sequence of events to show system time going backwards:

1. Device A reads the time from the system counter.
2. Device A communicates with another agent in the system, Device B.
3. After recognizing the communication from Device A, Device B reads the time from the system counter.

The system counter must be implemented in an always-on power domain.

To support lower-power operating modes in architectures from Armv8.0 to Armv8.5, the counter can increment by larger amounts at a lower frequency. For example, a 10MHz system counter might either increment:

- By 1 at 10MHz.
- By 500 at 20kHz, when the system lowers the clock frequency, to reduce power consumption.

In this case, the counter must support transitions between high-frequency, high-precision operation, and lower-frequency, lower-precision operation, without any impact on the required accuracy of the counter.

From Armv8.6 the counter operates at a higher fixed frequency of 1GHz.

**Note**

Though each unit of the counter is set to 1ns, this does not require that the counter is incremented every 1ns. A step in the counter might be more than a single bit increment. It is recommended that the count is not incremented at a rate that is less than 50MHz in normal running operation.

The CNTFRQ_EL0 register is intended to hold a copy of the current clock frequency to allow fast reference to this frequency by software running on the PE. For more information, see Initializing and reading the system counter frequency on page D10-5235.

The mechanism by which the count from the system counter is distributed to system components is IMPLEMENTATION DEFINED, but each PE with a System register interface to the system counter must have a counter input that can capture each increment of the counter.

**Note**

So that the system counter can be clocked independently from the PE hardware, the count value might be distributed using a Gray code sequence. Gray-count scheme for timer distribution scheme on page K7-11704 gives more information about this possibility.
Initializing and reading the system counter frequency

The CNTFRQ_EL0 register must be programmed to the clock frequency of the system counter. Typically, this is done only during the system boot process, by using the System register interface to write the system counter frequency to the CNTFRQ_EL0 register. Only software executing at the highest implemented Exception level can write to CNTFRQ_EL0.

--- Note ---

The CNTFRQ_EL0 register is UNKNOWN at reset, and therefore the counter frequency must be set as part of the system boot process.

Software can read the CNTFRQ_EL0 register, to determine the current system counter frequency, in the following states:

- Secure and Non-secure EL2.
- Secure and Non-secure EL1.
- When CNTKCTL_EL1.{EL0PCTEN, EL0VCTEN} is not {0,0} and CNTHCTL_EL2.{EL0PCTEN, EL0VCTEN} is not {0,0}, Secure and Non-secure EL0.

Memory-mapped controls of the system counter

Some system counter controls are accessible only through the memory-mapped interface to the system counter. These controls are:

- Enabling and disabling the counter.
- Setting the counter value.
- Changing the operating mode, to change the update frequency and increment value.
- Enabling Halt-on-debug, which a debugger can then use to suspend counting.

For descriptions of these controls, see Chapter I2 System Level Implementation of the Generic Timer.
D10.2 The AArch64 view of the Generic Timer

The following sections describe the components and features of a PE implementation of the Generic Timer, as seen from AArch64 state:

- **The physical counter.**
- **The virtual counter** on page D10-5238.
- **Event streams** on page D10-5239.
- **Timers** on page D10-5240.

D10.2.1 The physical counter

The PE includes a physical counter that contains the count value of the system counter. The CNTPCT_EL0 register holds the current physical counter value. When FEAT_ECV is implemented, the CNTPOFF_EL2 register holds the optional physical offset that can be applied at EL0 and EL1 whether EL0 and EL1 are using AArch64 state or AArch32 state. For more information, see The physical offset register on page D10-5237.

Reads of CNTPCT_EL0 can occur speculatively and out of order relative to other instructions executed on the same PE.

The self-synchronized view of the physical counter

When FEAT_ECV is implemented, an alternative way to read the physical counter is supported. The CNTPCTSS_EL0 register is a non-speculative view of the physical counter, as seen from the Exception level that CNTPCTSS_EL0 is read from.

Accesses to the CNTPCTSS_EL0 are subject to the same traps as accesses to the CNTPCT_EL0.

Reads of CNTPCT_EL0 occur in program order relative to reads of CNTPCT_EL0 or CNTPCTSS_EL0.

Reads of CNTPCTSS_EL0 occur in program order relative to reads of CNTPCT_EL0 or CNTPCTSS_EL0.

**Example D10-1 Ensuring reads of the physical counter occur after signal read from memory**

If a read from memory is used to obtain a signal from another agent that indicates that CNTPCT_EL0 must be read, an ISB is used to ensure that the read of CNTPCT_EL0 occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop                   ; polling for some communication to indicate a requirement to read the timer
  LDR X1, [X2]        ; has the value 1 written to it
  CMP X1, #1          ; without this the CNTPCT_EL0 could be read before the memory location in [X2]
  B.NE loop
  ISB                  ; without this the CNTPCT_EL0 could be read before the memory location in [X2]
  MRS X1, CNTPCT_EL0
```

When FEAT_ECV is implemented, an access to CNTPCTSS_EL0 can be used in place of the CNTPCT_EL0 which, because it cannot be accessed speculatively, allows the ISB to be removed. This means that the following code sequence can be used:

```
loop                   ; polling for some communication to indicate a requirement to read the timer
  LDR X1, [X2]        ; has the value 1 written to it
  CMP X1, #1          ; without this the CNTPCT_EL0 could be read before the memory location in [X2]
  B.NE loop
  MRS X1, CNTPCTSS_EL0
```

Similarly where a read of the physical counter is required to take place after the completion of all loads and stores appearing in program order before the read of the counter, then the following code sequences can be used:

```
...                   ; earlier loads and stores
DSB                    ; completes the earlier loads and stores
ISB                    ; without this the CNTPCT_EL0 could be read before the completion of the earlier
  MRS X1, CNTPCT_EL0   ; loads and stores
```
Or, if FEAT_ECV is implemented:

```assembly
... ; earlier loads and stores
DSB ; completes earlier loads and stores
MRS X1, CNTPCTSS_EL0
```

Neither view of the physical counter ensures that:

- Context changes occurring in program order before the read of the counter have been synchronized.
- Accesses to memory appearing in program order after the read of the counter are executed before the counter has been read.

Where there is a Dependency through registers dependency from the read of the physical counter to a Register effect generated by a read or write, the read or write will be executed after the read of the counter.

**Example D10-2 Ensuring reads of the physical counter occur after previous memory accesses**

To ensure that all previous memory accesses have completed and all previous context changes have been synchronized before the read of the counter, one of the following sequences should be used:

**either:**

```assembly
DSB
ISB
MRS Xn, CNTPCT{SS}_EL0 ; either view of the physical counter has the same effect in this example
```

**or**

```assembly
DMB
LDR Xa, [Xd] ; this could be any memory location, for example the stack pointer
CBZ Xa, next
```

```assembly
ISB ; this ISB is not needed if the MRS is accessing CNTPCTSS_EL0
MRS Xn, CNTPCT{SS}_EL0
```

To ensure that a memory access occurs only after a read of the counter, then either of the following sequences should be used:

```assembly
MRS Xn, CNTPCT{SS}_EL0 ; either view of the physical counter has the same effect in this example
ISB
LDR Xa, [Xb] ; this load will be executed after the timer has been read
```

**or**

```assembly
MRS Xn, CNTPCT{SS}_EL0 ; either view of the physical counter has the same effect in this example
EOR Xm, Xn, Xn
LDR Xa, [Xb, Xm] ; this load will be executed after the timer has been read
```

**The physical offset register**

When FEAT_ECV is implemented, the CNTPOFF_EL2 register allows an offset to be applied to the physical counter, as viewed from EL1 and EL0, and to the EL1 physical timer. The functionality of this 64-bit register is affected by CNTHCTL_EL2.ECV.

When CNTHCTL_EL2.ECV is 1, an MRS to CNTPCT_EL0 or CNTPCTSS_EL0 from either EL0 or EL1 that is not trapped will return the value \( PCount<63:0> - CNTPOFF_EL2<63:0> \). For information on how the EL1 physical timer interrupt is triggered when CNTHCTL_EL2.ECV is 1, see *Operation of the CompareValue views of the timers* on page D10-5241.

When EL2 is not enabled for the current Security state, or when CNTHCTL_EL2.ECV is 0, then:

- An MRS to CNTPCT_EL0 from either EL0 or EL1 that is not trapped will return the value PCount<63:0>. 

---

*The Generic Timer in AArch64 state*

*D10.2 The AArch64 view of the Generic Timer*
- The physical offset is treated as zero for all timer and counter calculations involving the physical offset.

When EL2 is not enabled for the current Security state, or when CNTHCTL_EL2.ECV is 0, then the behavior of the counters and timers is as described for Armv8.5 and the optional physical offset is not used.

When SCR_EL3.ECVEn is 0, all values of CNTPOFF_EL2 are treated as 0 for all purposes other than direct reads or writes to the register from EL3.

D10.2.2 The virtual counter

An implementation of the Generic Timer always includes a virtual counter, which indicates virtual time. The virtual counter contains the value of the physical counter minus a 64-bit virtual offset. When executing at EL1 or EL0, the virtual offset value relates to the current virtual machine.

The CNTVOFF_EL2 register contains the virtual offset, see The virtual offset register on page D10-5239.

The CNTVCT_EL0 register holds the current virtual counter value. Reads of CNTVCT_EL0 can occur speculatively and out of order relative to other instructions executed on the same PE.

The self-synchronized view of the virtual counter

When FEAT_ECV is implemented, an alternative way to read the virtual counter is supported. The CNTVCTSS_EL0 register is a non-speculative view of the virtual counter, as seen from the Exception level that CNTVCTSS_EL0 is read from.

Accesses to the CNTVCTSS_EL0 are subject to the same traps as accesses to the CNTVCT_EL0.

Reads of CNTVCT_EL0 occur in program order relative to reads of CNTVCT_EL0 or CNTVCTSS_EL0.

Reads of CNTVCTSS_EL0 occur in program order relative to reads of CNTVCT_EL0 or CNTVCTSS_EL0.

Example D10-3 Ensuring reads of the virtual counter occur after signal read from memory

If a read from memory is used to obtain a signal from another agent that indicates that CNTVCT_EL0 must be read, an ISB is used to ensure that the read of CNTVCT_EL0 occurs after the signal has been read from memory, as shown in the following code sequence:

\[
\text{loop} \quad ; \text{polling for some communication to indicate a requirement to read the timer} \\
\quad \text{LDR X1, [X2]} \quad ; \text{has had the value 1 written to it} \\
\quad \text{CMP X1, #1} \\
\quad \text{B.NE loop} \\
\quad \text{ISB} \quad ; \text{without this the CNTVCT_EL0 could be read before the memory location in [X2]} \\
\quad \text{MRS X1, CNTVCT_EL0} \\
\text{When FEAT_ECV is implemented, an access to CNTVCTSS_EL0 can be used in place of the CNTVCT_EL0, which, because it cannot be accessed speculatively, allows the ISB to be removed. This means that the following code sequence can be used:} \\
\text{loop} \quad ; \text{polling for some communication to indicate a requirement to read the timer} \\
\quad \text{LDR X1, [X2]} \quad ; \text{has had the value 1 written to it} \\
\quad \text{CMP X1, #1} \\
\quad \text{B.NE loop} \\
\quad \text{MRS X1, CNTVCTSS_EL0} \\
\text{Similarly where a read of the virtual counter is required to take place after the completion of all loads and stores appearing in program order before the read of the counter, then the following two sequences can be used:} \\
\text{...} \quad ; \text{earlier loads and stores} \\
\text{DSB} \quad ; \text{completes earlier loads and stores} \\
\text{ISB} \quad ; \text{without this CNTPCT_EL0 could be read before the completion of the earlier loads and stores} \\
\text{MRS X1, CNTVCT_EL0} \]
Or, if FEAT_ECV is implemented:

```
... ; earlier loads and stores
DSB ; completes earlier loads and stores
MRS X1, CNTVCTSS_EL0
```

Neither view of the virtual counter ensures that:

- Context changes occurring in program order before the read of the counter have been synchronized.
- Accesses to memory appearing in program order after the read of the counter are executed before the counter has been read.

Where there is a Dependency through registers dependency from the read of the virtual counter to a Register effect generated by a read or write, the read or write will be executed after the read of the counter.

**Example D10-4 Ensuring reads of the virtual counter occur after previous memory accesses**

To ensure that all previous memory accesses have completed and all previous context changes have been synchronized before the read of the counter, one of the following sequences should be used:

either:

```
DSB
ISB
MRS Xn, CNTVCT{SS}_EL0 ; either view of the virtual counter has the same effect in this example
```

or

```
DMB
LDR Xa, [Xd] ; this could be any memory location, for example the stack pointer
CBZ Xa, next
```

```
ISB ; this ISB is not needed if the MRS is accessing CNTVCT{SS}_EL0
MRS Xn, CNTVCT{SS}_EL0
```

To ensure that a memory access occurs only after a read of the counter, then either of the following sequences should be used:

```
MRS Xn, CNTVCT{SS}_EL0 ; either view of the virtual counter has the same effect in this example
ISB
LDR Xa, [Xb] ; this load will be executed after the timer has been read
```

or

```
MRS Xn, CNTVCT{SS}_EL0 ; either view of the virtual counter has the same effect in this example
EOR Xm, Xn, Xn
LDR Xa, [Xb, Xm] ; this load will be executed after the timer has been read
```

**The virtual offset register**

The virtual counter is a counter that has a virtual offset relative to the physical counter as viewed from EL2 and EL3. This virtual offset is held in the register CNTVOFF_EL2. The virtual counter value is the count compared by the EL1 virtual timer.

If EL2 is not implemented and enabled, then the virtual counter uses a fixed offset of zero.

**D10.2.3 Event streams**

An implementation that includes the Generic Timer can use the system counter to generate one or more event streams, to generate periodic wake-up events as part of the mechanism described in *Wait for Event on page D1-4676*. 
Note

An event stream might be used:

- To impose a time-out on a Wait For Event polling loop.
- To safeguard against any programming error that means an expected event is not generated.

The `CNTKCTL_EL1.{EVNTEN, EVNTDIR, EVNTI, EVNTIS}` fields define an event stream that is generated from the virtual counter.

In all implementations, the `CNTHCTL_EL2.{EVNTEN, EVNTDIR, EVNTI, EVNTIS}` fields define an event stream that is generated from the physical counter.

The event stream is configured as follows:

- `EVNTI` selects the counter bit that triggers the event.
- If `FEAT_ECV` is not implemented, `EVNTI` selects between bits[0:15].
- If `FEAT_ECV` is implemented, `EVNTIS` selects whether `EVNTI` selects between bits[0:15] or bits[8:23].
- `EVNTDIR` selects whether the event is generated on each 0 to 1 transition, or each 1 to 0 transition, of the selected counter bit.

Note

If the event stream is configured to produce events from the low order bits of the counter when the counter frequency is very high (for example 1GHz), then the practical update rate of the counter might mean that the event stream is not generated as the low order bit might not change. Software can rely on an event stream rate of at least 1MHz in normal operation.

The pseudocode descriptions of the operation of an event stream are `SetEventRegister`, `TestEventCNTV`, and `TestEventCNTP`.

D10.2.4 Timers

In an implementation of the Generic Timer that includes EL3, if EL3 can use AArch64, the following timers are implemented:

- An EL1 physical timer, which:
  - In Secure state, can be accessed from EL1.
  - In Non-secure state, can be accessed from EL1 unless those accesses are trapped to EL2.
  - When this timer can be accessed from EL1, an EL1 control determines whether it can be accessed from EL0.
- A Non-secure EL2 physical timer.
- A Secure EL3 physical timer. An EL3 control determines whether this register is accessible from Secure EL1.
- An EL1 virtual timer.
- When `FEAT_VHE` is implemented, a Non-secure EL2 virtual timer.
- When `FEAT_SEL2` is implemented, a Secure EL2 physical timer.
- When `FEAT_SEL2` is implemented, a Secure EL2 virtual timer.

The output of each implemented timer:

- Provides an output signal to the system.
- If the PE interfaces to a `Generic Interrupt Controller` (GIC), signals a `Private Peripheral Interrupt` (PPI) to that GIC. In a multiprocessor implementation, each PE must use the same interrupt number for each timer.

Each timer:

- Is based around a 64-bit CompareValue that provides a 64-bit unsigned upcounter.
- Provides an alternative view of the CompareValue, called the TimerValue, that appears to operate as a 32-bit downcounter.
The Generic Timer in AArch64 state

D10.2 The AArch64 view of the Generic Timer

- Has, in addition, a 32-bit Control register.

### Table D10-1 Physical timer registers summary for the Generic Timer

<table>
<thead>
<tr>
<th>Timer register</th>
<th>EL1 physical timer</th>
<th>EL2 physical timer</th>
<th>Secure EL2 physical timer</th>
<th>EL3 physical timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV</td>
<td>CNTP_CV_AL_EL0</td>
<td>CNTH_CV_AL_EL2</td>
<td>CNTHPS_CV_AL_EL2</td>
<td>CNTPS_CV_AL_EL1</td>
</tr>
<tr>
<td>TV</td>
<td>CNTP_TV_AL_EL0</td>
<td>CNTH_TV_AL_EL2</td>
<td>CNTHPS_TV_AL_EL2</td>
<td>CNTPS_TV_AL_EL1</td>
</tr>
<tr>
<td>Control</td>
<td>CNTP_CTL_EL0</td>
<td>CNTHP_CTL_EL2</td>
<td>CNTHPS_CTL_EL2</td>
<td>CNTPS_CTL_EL1</td>
</tr>
</tbody>
</table>

a. In this column, CV indicates the CompareValue register, and TV indicates the TimerValue register.

b. Present only when the implementation includes FEAT_SEL2.

### Table D10-2 Virtual timer register summary for the Generic Timer

<table>
<thead>
<tr>
<th>Timer register</th>
<th>EL1 virtual timer</th>
<th>EL2 virtual timer</th>
<th>Secure EL2 virtual timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV</td>
<td>CNTV_CV_AL_EL0</td>
<td>CNTHV_CV_AL_EL2</td>
<td>CNTHVS_CV_AL_EL2</td>
</tr>
<tr>
<td>TV</td>
<td>CNTV_TV_AL_EL0</td>
<td>CNTHV_TV_AL_EL2</td>
<td>CNTHVS_TV_AL_EL2</td>
</tr>
<tr>
<td>Control</td>
<td>CNTV_CTL_EL0</td>
<td>CNTHV_CTL_EL2</td>
<td>CNTHVS_CTL_EL2</td>
</tr>
</tbody>
</table>

a. In this column, CV indicates the CompareValue register, and TV indicates the TimerValue register.
b. Only when the implementation includes FEAT_VHE.
c. Present only when the implementation includes FEAT_SEL2.

### Operation of the CompareValue views of the timers

The CompareValue view of a timer operates as a 64-bit upcounter. The timer condition is met when the appropriate counter reaches the value programmed into its CompareValue register. When the timer condition is met, an interrupt is generated if the interrupt is not masked in the corresponding timer control register, CNTP_CTL_EL0, CNTHP_CTL_EL2, CNTHPS_CTL_EL2, CNTPS_CTL_EL1, CNTV_CTL_EL0, CNTHV_CTL_EL2, or CNTHVS_CTL_EL2. For CNTP_CTL_EL0, the asserted interrupt is the same as the interrupt asserted by the Non-secure instance of the AArch32 register CNTP_CTL.

The operation of this view of a timer is:

\[
\text{TimerConditionMet} = (((\text{Counter}[63:0] - \text{Offset}[63:0])[63:0] - \text{CompareValue}[63:0]) >= 0)
\]

Where:

- **TimerConditionMet**: Is TRUE if the timer condition for this counter is met, and FALSE otherwise.
- **Counter**: The physical counter value, which can be read from the CNTPCT_EL0 register when read from EL2 or EL3.
- **Offset**: For the EL1 physical timer, if ID_AA64MMFR0_EL1.ECV is 0b10 and CNTHCTL_EL2.ECV is 0b1, then the offset value is held in the CNTPOFF_EL2 register. Otherwise, the offset value of the EL1 physical timer is zero.
  - For the EL1 virtual timer, the offset value is held in the CNTVOFF_EL2 register.
  - For the EL2 physical and virtual timers, the offset value is zero.
- **CompareValue**: The value of the appropriate CompareValue register, CNTP_CV_AL_EL0, CNTHP_CV_AL_EL2, CNTHPS_CV_AL_EL2, CNTV_CV_AL_EL0, CNTHV_CV_AL_EL2, or CNTHVS_CV_AL_EL2.

In this view of a timer, Counter, Offset, and CompareValue are all 64-bit unsigned values.
Note

This means that a timer with a CompareValue of, or close to, 0xFFFFFFFF_FFFF_FFFF might never meet its timer condition. However, there is no practical requirement to use values close to the counter wrap value.

Software can observe the counter value by the offset in some situations by reading CNTVCT_EL0.

Operation of the TimerValue views of the timers

The TimerValue view of a timer appears to operate as a signed 32-bit downcounter. A TimerValue register is programmed with a count value. This value decrements on each increment of the appropriate counter, and the timer condition is met when the value reaches zero. When the timer condition is met, an interrupt is generated if the interrupt is not masked in the corresponding timer control register, CNTP_CTL_EL0, CNTHP_CTL_EL2, CNTHPS_CTL_EL2, CNTPS_CTL_EL1, CNTV_CTL_EL0, CNTH_CTL_EL2, or CNTHVS_CTL_EL2.

This view of a timer depends on the following behavior of accesses to TimerValue registers:

Reads

\[
\text{TimerValue} = (\text{CompareValue} - (\text{Counter} - \text{ModOffset}))\{31:0\}
\]

Writes

\[
\text{CompareValue} = ((\text{Counter} - \text{ModOffset})\{63:0\} + \text{SignExtend} (\text{TimerValue}))\{63:0\}
\]

Where the arguments other than TimerValue and ModOffset have the definitions used in Operation of the CompareValue views of the timers, and in addition:

TimerValue

The value of a TimerValue register, CNTP_TVAL_EL0, CNTHP_TVAL_EL2, CNTHPS_TVAL_EL2, CNTPS_TVAL_EL1, CNTV_TVAL_EL0, CNTHV_TVAL_EL2, or CNTHVS_TVAL_EL2.

ModOffset

Offset for all timer registers other the EL1 physical timer accessed through CNTP_CTL_EL0.

Offset for the EL1 physical timer accessed through CNTP_CTL_EL0, if ID_AA64MMFR0_EL1.ECV is less than 0b10 or CNTHCTL_EL2.ECV is 0,

Offset for the EL1 physical timer accessed through CNTP_CTL_EL0, if ID_AA64MMFR0_EL1.ECV is 0b10 and CNTHCTL_EL2.ECV is 1, and if the access is from EL0 or EL1.

0 for the EL1 physical timer accessed through CNTP_CTL_EL0 if ID_AA64MMFR0_EL1.ECV is 0b10 and CNTHCTL_EL2.ECV is 1, and if the access is from EL2 or EL3.

In this view of a timer, values are signed in standard two’s complement form.

A read of a TimerValue register after the timer condition has been met indicates the time since the timer condition was met.

Note

- Operation of the CompareValue views of the timers on page D10-5241 gives a strict definition of TimerConditionMet. However, provided that the TimerValue is not expected to wrap as a 32-bit signed value when decremented from 0x80000000, the TimerValue view can be used as giving an effect equivalent to:

  \[\text{TimerConditionMet} = (\text{TimerValue} \leq 0)\]

- Programming TimerValue to a negative number with magnitude greater than (Counter–Offset) can lead to an arithmetic overflow that causes the CompareValue to be an extremely large positive value. This potentially delays meeting the timer condition for an extremely long period of time.
Chapter D11
The Performance Monitors Extension

This chapter describes the implementation of the Arm Performance Monitors, that are an optional non-invasive debug component. It describes version 3 of the Performance Monitor Unit (PMU) architecture, FEAT_PMUv3. It contains the following sections:

• About the Performance Monitors on page D11-5244.
• Accuracy of the Performance Monitors on page D11-5248.
• Behavior on overflow on page D11-5250.
• Attributability on page D11-5252.
• Controlling the PMU counters on page D11-5254.
• Event filtering on page D11-5260.
• Event counting threshold on page D11-5262.
• Performance Monitors and Debug state on page D11-5264.
• Enabling event counters on page D11-5254.
• Counter access on page D11-5265.
• PMU events and event numbers on page D11-5266.
• Performance Monitors Extension registers on page D11-5428.

Note
Table K17-2 on page K17-11838 disambiguates the general register references used in this chapter.
D11.1 About the Performance Monitors

The Performance Monitors Extension is an OPTIONAL feature of an implementation, but Arm strongly recommends that implementations include version 3 of the Performance Monitors Extension, FEAT_PMUv3.

Note
No previous versions of the Performance Monitors Extension can be implemented in architectures from Armv8.0.

The basic form of the Performance Monitors is:

- A 64-bit cycle counter, see Time as measured by the Performance Monitors cycle counter on page D11-5246.
- A number of 64-bit or 32-bit event counters. If FEAT_PMUv3p5 is implemented and the highest Exception level is using AArch64, the event counters are 64-bit. If FEAT_PMUv3p5 is not implemented, the event counters are 32-bit.
- The event counted by each event counter is programmable. The architecture provides space for up to 31 event counters. The actual number of event counters is IMPLEMENTATION DEFINED, and the specification includes an identification mechanism.

Note
The Performance Monitors Extension permits an implementation with no event counters (PMCR_EL0.N==0). However, Arm recommends that at least two event counters are implemented, and that hypervisors provide at least this many event counters to guest operating systems.

- When EL2 is implemented, the required controls to partition the implemented event counters into the following ranges:
  - A first range which is available for use by the guest operating system accessible at all Exception levels.
  - A second range which is available for use by the hypervisor accessible at EL3 and EL2, and, if FEAT_SEL2 is not implemented or if Secure EL2 is disabled, in Secure state.

- Controls for:
  - Enabling and resetting counters.
  - Flagging overflows.
  - Enabling interrupts on overflow.
  - Disabling or freezing counters.
  - Threshold counting.

The PMU architecture uses event numbers to identify an event. It:

- Defines event numbers for common events, for use across many architectures and microarchitectures.

Note
Implementations that include FEAT_PMUv3 must, as a minimum requirement, implement a subset of the common events. See Common event numbers on page D11-5306.

- Reserves a large event number space for IMPLEMENTATION DEFINED events.

The full set of events for an implementation is IMPLEMENTATION DEFINED. Arm recommends that implementations include all of the events that are appropriate to the architecture profile and microarchitecture of the implementation.

When an implementation includes the Performance Monitors Extension, the architecture defines the following possible interfaces to the Performance Monitors Extension registers:

- A System register interface. This interface is mandatory.

Note
In AArch32 state, the interface is in the (coproc==0b1111) encoding space.
• An external debug interface which optionally supports memory-mapped accesses. Implementation of this interface is OPTIONAL. See Chapter 13 Recommended External Interface to the Performance Monitors.

An operating system can use the System registers to access the counters.

Also, if required, the operating system can enable application software to access the counters. This enables an application to monitor its own performance with fine-grain control without requiring operating system support. For example, an application might implement per-function performance monitoring.

To enable interaction with external monitoring, an implementation might consider additional enhancements, such as providing:
• A set of events, from which a selection can be exported onto a bus for use as external events.
• The ability to count external events. This enhancement requires the implementation to include a set of external event input signals.

The Performance Monitors Extension is common to AArch64 operation and AArch32 operation. This means the architecture defines both AArch64 and AArch32 System registers to access the Performance Monitors. For example, the Performance Monitors Cycle Count Register is accessible as:
• When executing in AArch64 state, PMCCNTR_EL0.
• When executing in AArch32 state, PMCCNTR.

When executing in AArch32 state, if FEAT_PMUv3p5 is implemented, bits [63:32] of the event counters are not accessible. If the implementation does not support AArch64 at any Exception level, 64-bit event counters are not required to be implemented.

D11.1.1 Interaction with EL3

Software executing at EL3 can trap attempts by lower Exception levels to access the PMU. This means that the Secure monitor can identify any software which is using the PMU and switch contexts, if required.

Software executing at EL3 can:
• Prohibit counting of events Attributable to Secure state.
• If FEAT_PMUv3p5 is implemented, prohibit the cycle counter from counting cycles in Secure state, see Controlling the PMU counters on page D11-5254.
• If FEAT_PMUv3p7 is implemented:
  — Prohibit event counters from counting events at EL3 without affecting the rest of Secure state.
  — Prohibit the cycle counter from counting cycles at EL3 without affecting the rest of Secure state.

For more information, see Controlling the PMU counters on page D11-5254 and Freezing event counters on page D11-5255.

In AArch32 state, the Performance Monitors registers are Common registers, see Classification of System registers on page G5-9281.

If FEAT_MTPMU is implemented and EL3 is implemented, MDCR_EL3.MTPME and SDCR.MTPME enable and disable the PMEVTYPE<\alpha>.MT bit.

D11.1.2 Interaction with EL2

Software executing at EL3 or EL2 can program HDCR.HPMN to partition the event counters into two ranges:
• If HDCR.HPMN is not 0 and is less-than PMCR.N, HDCR.HPMN divides the event counters into a first range [0..(HDCR.HPMN-1)], and a second range [HDCR.HPMN..(PMCR.N-1)].
• If FEAT_HPMN0 is implemented and HDCR.HPMN is 0, all event counters are in the second range and none are in the first range.
• If HDCR.HPMN is equal to PMCR.N, all event counters are in the first range and none are in the second range.

This does not depend on whether EL2 is enabled in the current Security state. Each range of event counters has its own global controls.
If `FEAT_HPMN0` is not implemented and `HDCR.HPMN` is 0, the behavior is CONSTRAINED UNPREDICTABLE. See:

- *The Performance Monitors Extension* on page K1-11586.

Software executing at EL3 or EL2 can:

- Trap an access at EL0 or EL1 to the PMU. This means the hypervisor can identify which Guest OSs are using the PMU and intelligently employ switching of the PMU state. There is a separate trap for the `PMCR` register, and if `FEAT_FGT` is implemented and enabled, fine-grained traps are provided.

- If `FEAT_PMUv3p1` is implemented, prohibit counting of events Attributable to EL2 by the event counters in the first range.

- If `FEAT_PMUv3p5` is implemented, prohibit the cycle counter from counting cycles at EL2.

When EL2 is implemented and enabled in the current Security state, software executing at EL1 and, if enabled by `PMUSERENR`, EL0:

- Will read the value of `HDCR.HPMN` for `PMCR.N`.
- Cannot access the event counters in the second range, or the controls associated with them.

If `FEAT_MTPMU` is implemented, EL3 is not implemented, and EL2 is implemented, `MDCR_EL2.MTPME` and `HDCR.MTPME` enable and disable the `PMEVTPYPER<n>.MT` bit.

For more information, see:

- *Enabling event counters* on page D11-5254.
- *Counter access* on page D11-5265.
- *Controlling the PMU counters* on page D11-5254.
- *Multithreaded implementations* on page D11-5258.

**D11.3 Time as measured by the Performance Monitors cycle counter**

The Performance Monitors cycle counter, accessed through `PMCCNTR_EL0` or `PMCCNTR`, increments from the hardware processor clock, not PE clock cycles.

The relationship between the count recorded by the Performance Monitors cycle counter and the passage of real time is IMPLEMENTATION DEFINED.

See *Controlling the PMU counters* on page D11-5254 for information about when the cycle counter does not increment.

______ Note ______

- This means that, in an implementation where PEs are multithreaded, when enabled, the cycle counter continues to increment across all PEs, rather than only counting cycles for which the current PE is active.

- Although the architecture requires that direct reads of `PMCCNTR_EL0` or `PMCCNTR` occur in program order, there is no requirement that the count increments between two such reads. Even when the counter is incrementing on every clock cycle, software might need check that the difference between two reads of the counter is nonzero.

The architecture requires that an indirect write to the `PMCCNTR_EL0` or `PMCCNTR` is observable to direct reads of the register in finite time. The counter increments from the hardware processor clock are indirect writes to these registers.

**D11.4 Interaction with trace**

It is IMPLEMENTATION DEFINED whether the implementation exports counter events to a trace unit, or other external monitoring agent, to provide triggering information. The form of any exporting is also IMPLEMENTATION DEFINED. If implemented, this exporting might be enabled as part of the performance monitoring control functionality.
Arm recommends system designers include a mechanism for importing a set of external events to be counted, but such a feature is IMPLEMENTATION DEFINED. When implemented, this feature enables the trace unit to pass in events to be counted.

Exporting PMU events to the ETM is prohibited for some Exception levels when SelfHostedTraceEnabled() == TRUE. For more information, see Controls to prohibit trace at Exception levels on page D3-4759.
D11.2 Accuracy of the Performance Monitors

The Performance Monitors:

- Are a non-invasive debug component. See Non-invasive behavior.
- Must provide broadly accurate and statistically useful count information.

However, the Performance Monitors allow for:

- A reasonable degree of inaccuracy in the counts to keep the implementation and validation cost low. See A reasonable degree of inaccuracy on page D11-5249.
- IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, that software must configure before using certain PMU events. For example, to configure how the PE generates PMU events for components such as external caches and external memory.
- Other IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, to optionally put the PE in an operating state that might do one or both of the following:
  - Change the level of non-invasiveness of the Performance Monitors so that enabling an event counter can impact the performance or behavior of the PE.
  - Allow inaccurate counts. This includes, but is not limited to, cycle counts.

D11.2.1 Non-invasive behavior

The Performance Monitors are a non-invasive debug feature. A non-invasive debug feature permits the observation of data and program flow. Performance Monitors, PC Sample-based Profiling and Trace are non-invasive debug features.

Non-invasive debug components do not guarantee that they do not make any changes to the behavior or performance of the processor. Any changes that do occur must not be severe however, as this will reduce the usefulness of event counters for performance measurement and profiling. This does not include any change to program behavior that results from the same program being instrumented to use the Performance Monitors, or from some other performance monitoring process being run concurrently with the process being profiled in a multitasking operating system. As such, a reasonable variation in performance is permissible.

Note

Power consumption is one measure of performance. Therefore, a reasonable variation in power consumption is permissible.

Arm does not define a reasonable variation in performance, but recommends that such a variation is kept within 5% of normal operating performance, when averaged across a suite of code that is representative of the application workload.

Note

For profiles other than A-profile, there is the potential for stronger requirements. Ultimately, performance requirements are determined by end-users, and not set by the architecture.

For some common architectural events, this requirement to be non-invasive can conflict with the requirement to present an accurate value of the count under normal operating conditions. Should an implementation require more performance-invasive techniques to accurately count an event, there are the following options:

- If the event is optional, define an alternative implementation defined event that accurately counts the event and document the impact on performance of enabling the event.
- Provide an implementation defined control that disables accurate counting of the event to restore broadly accurate performance, and document the impact on performance of accurate counting.
D11.2.2 A reasonable degree of inaccuracy

The Performance Monitors provide broadly accurate and statistically useful count information. To keep the implementation and validation cost low, a reasonable degree of inaccuracy in the counts is acceptable. Arm does not define a reasonable degree of inaccuracy but recommends the following guidelines:

• Under normal operating conditions, the counters must present an accurate value of the count.

• In exceptional circumstances, such as a change in Security state or other boundary condition, it is acceptable for the count to be inaccurate.

• Under very unusual, non-repeating pathological cases, the counts can be inaccurate. These cases are likely to occur as a result of asynchronous exceptions, such as interrupts, where the chance of a systematic error in the count is very unlikely.

Note

An implementation must not introduce inaccuracies that can be triggered systematically by the execution of normal pieces of software. For example, it is not reasonable for the count of branch behavior to be inaccurate when caused by a systematic error generated by the loop structure producing a dropping in branch count.

However, dropping a single branch count as the result of a rare interaction with an interrupt is acceptable.

The permitted inaccuracy limits the possible uses of the Performance Monitors. In particular, the architecture does not define the point in a pipeline where the event counter is incremented, relative to the point where a read of the event counters is made. This means that pipelining effects can cause some imprecision, and can affect which events are counted.

Where a direct write to a Performance Monitors control register disables a counter, and is followed by a Context synchronization event, any subsequent indirect read of the control register by the Performance Monitors to determine whether the counter is enabled will return the updated value. Any subsequent direct read of the counter or counter overflow status flags will return the value at the point the counter was disabled.

Note

The imprecision means that the counter might have counted an event around the time the counter was disabled, but does not allow the event to be observed as counted after the counter was disabled.

A change of Security state can also affect the accuracy of the Performance Monitors, see Interaction with EL3 on page D11-5245.

In addition to this, entry to and exit from Debug state can disturb the normal running of the PE, causing further inaccuracy in the Performance Monitors. Disabling the counters while in Debug state limits the extent of this inaccuracy. An implementation can employ methods to limit this inaccuracy, for example by promptly disabling the counters during the Debug state entry sequence.

An implementation must document any particular scenarios where significant inaccuracies are expected.
D11.3 Behavior on overflow

The event counters, PMEVCNTR<n>, are either 32-bit or 64-bit unsigned counters that overflow in the following situations:

- If FEAT_PMUvs3p5 is not implemented, 32-bit event counters are implemented, and if incrementing PMEVCNTR<n> causes an unsigned overflow of an event counter, the PE sets PMOVSCLR[n] to 1.

- If FEAT_PMUvs3p5 is implemented, 64-bit event counters are implemented, HDCR.HPMN is not 0, and either n is in the range [0 .. (HDCR.HPMN-1)] or EL2 is not implemented, then event counter overflow is configured by PMCR.LP:
  - When PMCR.LP is set to 0, if incrementing PMEVCNTR<n> causes an unsigned overflow of bits [31:0] of the event counter, the PE sets PMOVSCLR[n] to 1.
  - When PMCR.LP is set to 1, if incrementing PMEVCNTR<n> causes an unsigned overflow of bits [63:0] of the event counter, the PE sets PMOVSCLR[n] to 1.

- If FEAT_PMUvs3p5 is implemented, 64-bit event counters are implemented, EL2 is implemented, and HDCR.HPMN is less-than PMCR.N, when n is in the range [HDCR.HPMN .. (PMCR.N-1)], event counter overflow is configured by HDCR.HLP:
  - When HDCR.HLP is set to 0, if incrementing PMEVCNTR<n> causes an unsigned overflow of bits [31:0] of the event counter, the PE sets PMOVSCLR[n] to 1.
  - When HDCR.HLP is set to 1, if incrementing PMEVCNTR<n> causes an unsigned overflow of bits [63:0] of the event counter, the PE sets PMOVSCLR[n] to 1.

The cycle counter, PMCCNTR, is a 64-bit unsigned counter, that is configured by PMCR.LC:

- If PMCR.LC is set to 0, if incrementing PMCCNTR causes an unsigned overflow of bits [31:0] of the cycle counter, the PE sets PMOVSCLR[31] to 1.
- If PMCR.LC is set to 1, if incrementing PMCCNTR causes an unsigned overflow of bits [63:0] of the cycle counter, the PE sets PMOVSCLR[31] to 1.

The update of PMOVSCLR occurs synchronously with the update of the counter.

For all 64-bit counters, incrementing the counter is the same whether an unsigned overflow occurs at [31:0] or [63:0]. If the counter increments for an event, bits [63:0] are always incremented,

When any overflow occurs, an interrupt request is generated if the PE is configured to generate counter overflow interrupts. For more information, see Generating overflow interrupt requests.

If FEAT_PMUvs3p7 is implemented, event counting can be frozen after an unsigned overflow is detected, see Freezing event counters on page D11-5255.

--- Note

Software executing at EL1 or higher must take care that setting PMCR.LP or HDCR.HLP does not cause software executing at lower Exception levels to malfunction. If legacy software accesses the PMU at lower Exception levels, software at the higher Exception levels should not set the PMCR.LP or HDCR.HLP fields to 1. However, if the legacy software does not use the counter overflow, it is not affected by setting the PMCR.LP or HDCR.HLP to 1.

D11.3.1 Generating overflow interrupt requests

Software can program the Performance Monitors so that an overflow interrupt request is generated when a counter overflows. See PMINTENSET and PMINTEGR.

--- Note

- The mechanism by which an interrupt request from the Performance Monitors generates an FIQ or IRQ exception is IMPLEMENTATION DEFINED.
- Arm recommends that the overflow interrupt requests:
  - Translate into a PMUIRQ signal, so that they are observable to external devices.
— Connect to inputs on an IMPLEMENTATION DEFINED Generic Interrupt Controller as a Private Peripheral Interrupt (PPI) for the originating processor. See the ARM Generic Interrupt Controller Architecture Specification for information about PPIs.

— Connect to a Cross Trigger Interface (CTI), see Chapter H5 The Embedded Cross-Trigger Interface.

• Arm strongly discourages implementations from connecting overflow interrupt requests from multiple PEs to the same System Peripheral Interrupt (SPI) identifier.

• From GICv3, the ARM® Generic Interrupt Controller Architecture Specification recommends that the Private Peripheral Interrupt (PPI) with ID 23 is used for overflow interrupt requests.

Software can write to the counters to control the frequency at which interrupt requests occur. For example, software might set a 32-bit counter to 0xFFFF0000, to generate another counter overflow after 65536 increments, and reset it to this value every time an overflowing interrupt occurs.

Note

If an event can occur multiple times in a single clock cycle, then counter overflow can occur without the counter registering a value of zero.

The overflow interrupt request is a level-sensitive request. The PE signals a request for:

• Any given PMEVCNTR<\text{n}> counter, when the value of PMOVSET[n] is 1, the value of PMINTENSE[n] is 1, and one of the following is true:
  — EL2 is not implemented and the value of PMCR.E is 1.
  — EL2 is implemented, \text{n} is less than the value of HDCR.HPMN, and the value of PMCR.E is 1.
  — EL2 is implemented, \text{n} is greater than or equal to the value of HDCR.HPMN, and the value of HDCR.HPME is 1.

• The cycle counter, when the values of PMOVSET[31], PMINTENSE[31], and PMCR.E are all 1.

The overflow interrupt request is active in both Secure and Non-secure states. In particular, if EL3 and EL2 are both implemented, overflow events from PMEVCNTR<\text{n}> where \text{n} is greater than or equal to the value of HDCR.HPMN can be signaled from all modes and states but only if the value of HDCR.HPME is 1.

The interrupt handler for the counter overflow request must cancel the interrupt request, by writing 1 to PMOVSCLR[n] to clear the overflow bit to 0.

Pseudocode description of overflow interrupt requests

See Chapter J1 Armv8 Pseudocode for a pseudocode description of overflow interrupt requests. The AArch64.CheckForPMUOverflow() and AArch32.CheckForPMUOverflow() pseudocode functions signal PMU overflow interrupt requests to an interrupt controller and PMU overflow trigger events to the cross-trigger interface.
D11.4 **Attributability**

An event caused by the PE counting the event is *Attributable*. If an agent other than the PE that is counting the events causes an event, these events are Unattributable.

An event is defined as being either Attributable or Unattributable. If the event is Attributable, it is further defined whether it is Attributable to:

- The current Security state of the PE.
- The current Exception level of the PE.
- When the PE is in Debug state, operations issued to the PE by the debugger through the external debug interface.

In a multithreaded implementation, an event might be generated by another PE with the same values for affinity level 1 and higher. This event is further defined as Attributable to:

- The current Security state of that PE.
- The current Exception level of that PE.
- When that PE is in Debug state, operations issued to that PE by the debugger through the external debug interface.

See *Multithreaded implementations on page D11-5258* for information about enabling and restricting counting events in a multithreaded implementation.

--- **Note** ---

- In an implementation containing multiple PEs, each PE is identified by a unique *affinity* value reported by `MPIDR_EL1[Aff3, Aff2, Aff1, Aff0]`, where the value of affinity level 0 is the most significant for determining the PE behavior, and the values of higher affinity levels are less significant. Affinity level 3 is only supported in AArch64 state.

- An implementation is described as multithreaded when the lowest level of affinity consists of logical PEs that are implemented using a multithreading type approach. In this section, when referring to a multithreaded implementation, *thread* is used to mean processing elements with:
  - `MPIDR_EL1.MT` or `MPIDR.MT` set to 1,
  - Different values for affinity level 0.
  - The same values for affinity level 1 and higher.

An event can be defined as the combination of multiple subevents, which can be either Attributable or Unattributable.

All architecturally defined events are *Attributable*, unless otherwise stated.

Unattributable events might be counted when Attributable events are not counted. See:

- *Interaction with EL3 on page D11-5245.*
- *Event filtering on page D11-5260.*
- *Performance Monitors and Debug state on page D11-5264.*
These sections are summarized by Table D11-1 on page D11-5253 for events Attributable to the processor, and Unattributable events. Table D11-1 on page D11-5253 entries apply when the counter and PMU are enabled and not frozen. Otherwise, events are not counted.

### Table D11-1 Counting events

<table>
<thead>
<tr>
<th>State</th>
<th>Allowed or prohibited</th>
<th>Filtered</th>
<th>Event type</th>
<th>If Attributable to:</th>
<th>Then</th>
<th>Else</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-debug</td>
<td>Allowed</td>
<td>Not filtered</td>
<td>X</td>
<td>Count</td>
<td>Count</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Filtered</td>
<td>Current Exception level</td>
<td>Do not count</td>
<td>IMPLEMENTATION</td>
<td>DEFINED</td>
</tr>
<tr>
<td>Prohibited</td>
<td></td>
<td>X</td>
<td>Current Security state</td>
<td>Do not count</td>
<td>IMPLEMENTATION</td>
<td>DEFINED</td>
</tr>
<tr>
<td>Debug</td>
<td>X</td>
<td>X</td>
<td>Debugger operations or raw cycles</td>
<td>Do not count</td>
<td>IMPLEMENTATION</td>
<td>DEFINED</td>
</tr>
</tbody>
</table>
D11.5 Controlling the PMU counters

This section describes the mechanisms available for controlling the PMU event and cycle counters. The following sections describe those mechanisms:

- Enabling event counters.
- Freezing event counters on page D11-5255.
- Prohibiting event and cycle counting on page D11-5256.

D11.5.1 Enabling event counters

Table D11-2 on page D11-5254 shows an implementation that does not include EL2, where the PMCR.E bit is a global counter enable bit, and PMCNTENSET provides an enable bit for each counter.

<table>
<thead>
<tr>
<th>PMCR.E</th>
<th>PMCNTENSET[n] == 0</th>
<th>PMCNTENSET[n] == 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PMEVCNTR&lt;n&gt; disabled</td>
<td>PMEVCNTR&lt;n&gt; disabled</td>
</tr>
<tr>
<td>1</td>
<td>PMEVCNTR&lt;n&gt; disabled</td>
<td>PMEVCNTR&lt;n&gt; enabled</td>
</tr>
</tbody>
</table>

If the implementation includes EL2, then in addition to the PMCR.E and PMCNTENSET enable bits:

- HDCR.HPME overrides the value of PMCR.E for counters configured for access in EL2.
- HDCR.HPMN specifies the number of event counters that the Guest OS can access. When FEAT_HPMN0 is not implemented, the minimum permitted value of HDCR.HPMN is 1, meaning there must be at least one event counter that the Guest OS can access. Otherwise, the minimum permitted value of HDCR.HPMN is 0.

Table D11-3 on page D11-5254 shows the combined effect of all the counter enable controls.

<table>
<thead>
<tr>
<th>HDCR.HPME</th>
<th>PMCR.E</th>
<th>PMCNTENSET[n] == 0</th>
<th>PMCNTENSET[n] == 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>PMEVCNTR&lt;n&gt; disabled</td>
<td>PMEVCNTR&lt;n&gt; disabled</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>PMEVCNTR&lt;n&gt; disabled</td>
<td>PMEVCNTR&lt;n&gt; enabled</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>PMEVCNTR&lt;n&gt; disabled</td>
<td>PMEVCNTR&lt;n&gt; disabled</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>PMEVCNTR&lt;n&gt; disabled</td>
<td>PMEVCNTR&lt;n&gt; enabled</td>
</tr>
</tbody>
</table>

Note:

- The effect of HDCR.{HPME, HPMN} on the counter enables applies at all Exception levels and in both Security states.
- The value returned for PMCR.N is not affected by HDCR.HPMN at:
  - EL3.
  - EL2.
  - Secure EL1, if FEAT_SEL2 is not implemented or Secure EL2 is disabled.
  - Secure EL0, if FEAT_SEL2 is not implemented or Secure EL2 is disabled.
The cycle counter, PMCCNTR, counts unless disabled or prohibited as described in *Prohibiting event and cycle counting* on page D11-5256.

### D11.5.2 Freezing event counters

When FEAT_SPEv1p2 is implemented, the PMU can be configured to freeze event counters when an SPE buffer management event occurs. A counter is disabled under the following conditions:

- If EL2 is implemented, MDCR_EL2.HPMN is not 0, \( n \) is in the range \([0 .. (MDCR_EL2.HPMN-1)]\), when PMBSR_EL1.S is 1, and PMBLIMITR_EL1.E is 1, indicating an SPE buffer management event occurred, event counter \( n \) does not count if all the following are true:
  - PMBLIMITR_EL1.PMFZ is 1.
  - PMCR_EL0.FZS is 1.
- If EL2 is not implemented, \( n \) is in the range \([0 .. (PMCR_EL0.N-1)]\), when PMBSR_EL1.S is 1 and PMBLIMITR_EL1.E is 1, indicating an SPE buffer management event occurred, event counter \( n \) does not count if all the following are true:
  - PMBLIMITR_EL1.PMFZ is 1.
  - PMCR_EL0.FZS is 1.
- If EL2 is implemented, MDCR_EL2.HPMN is not 0, \( n \) is in the range \([MDCR_EL2.HPMN .. (PMCR_EL0.N-1)]\), when PMBSR_EL1.S is 1 and PMBLIMITR_EL1.E is 1, indicating an SPE buffer management event occurred, event counter \( n \) does not count if all the following are true:
  - PMBLIMITR_EL1.PMFZ is 1.
  - MDCR_EL2.HPMFZ is 1.

**Note**

This also applies when EL2 is disabled in the current Security state.

If the highest implemented Exception level is using AArch32, then the Effective value of PMBLIMITR_EL1.E is 0 and FEAT_SPEv1p2 does not affect the PMU event counters. Otherwise, the effect of FEAT_SPEv1p2 on PMU event counters applies in AArch32 state. See *Profiling Buffer management* on page D13-5463.

When FEAT_PMUv3p7 is implemented, the PMU can be configured to freeze event counters when an unsigned overflow of a counter occurs. A counter is disabled under the following conditions:

- If EL2 is implemented, MDCR_EL2.HPMN is not 0, and \( n \) is in the range \([0 .. (MDCR_EL2.HPMN-1)]\), when PMOVSCLR_EL0[(MDCR_EL2.HPMN-1):0] is nonzero, indicating an unsigned overflow in one of the event counters in the range, event counter \( n \) does not count when PMCR_EL0.FZO is 1.
- If EL2 is implemented, MDCR_EL2.HPMN is less than PMCR_EL0.N and \( n \) is in the range \([MDCR_EL2.HPMN .. (PMCR_EL0.N-1)]\), when PMOVSCLR_EL0[(PMCR_EL0.N-1):MDCR_EL2.HPMN] is nonzero, indicating an unsigned overflow in one of the event counters in the range, event counter \( n \) does not count when MDCR_EL2.HPMFZO is 1.
- If EL2 is not implemented and \( n \) is in the range \([0 .. (PMCR_EL0.N-1)]\), when PMOVSCLR_EL0[(PMCR_EL0.N-1):0] is nonzero, indicating an unsigned overflow in one of the event counters in the range, event counter \( n \) does not count when PMCR_EL0.FZO is 1.

When the applicable PMCR_EL0.FZO or MDCR_EL2.HPMFZO bit is 1, it is **CONSTRAINED UNPREDICTABLE** whether any event happening at or about the same time as the event that caused the overflow is counted. This includes other instances of the same event.

**Note**

The architecture does not define when PMU events are counted relative to the instructions that caused the event. Events caused by an instruction might be counted before or after the instruction becomes architecturally executed, and events might be counted for operations that are not architecturally executed. Events can be counted speculatively, out-of-order, or both with respect to the simple sequential execution of the program. Events might also be counted simultaneously by other event counters when the overflow occurs, including events from different instructions. Multiple instances of an event might occur simultaneously, thus an event counter unsigned overflow can yield a nonzero value in the event counter.
Arm recommends that such counting anomalies are minimized when software uses the freeze on overflow feature. When the freeze on overflow feature is being used, software cannot assume that the event counter stops counting at zero when an overflow occurs.

If an event counter \( n \) overflows, where \( n \) is even and event counter \( n+1 \) is configured to count the CHAIN event, it is \text{CONSTRAINED UNPREDICTABLE} whether the CHAIN event observes the overflow event when the applicable \text{PMCR\_EL0.FZO} or \text{MDCR\_EL2.HPMFZO} bit is 1 and the corresponding \text{PMCR\_EL0.LP} or \text{MDCR\_EL2.HLP} bit is 0.

If a direct read of \text{PMOVSLR\_EL0} returns a nonzero value for a subset of the overflow flags, which means an event counter \( n \) should not count, then a sequence of direct reads of \text{PMEVCNTR<\(n\)} >\_EL0 ordered after the read of \text{PMOVSLR\_EL0} and before the \text{PMOVSLR\_EL0} flags are cleared to zero, will return the same value for each read, because the event counter has stopped counting.

\textbf{Note}

Direct reads of System registers require explicit synchronization for following direct reads of other System registers to be ordered after the first direct read.

\section{Prohibiting event and cycle counting}

Counting \textit{Attributable} events in Secure state is prohibited unless any one of the following is true:

\begin{itemize}
  \item EL3 is not implemented.
  \item \text{FEAT\_PMUv3p7} is not implemented, EL3 is implemented, is using AArch64, and the value of \text{MDCR\_EL3.SPME} is 1.
  \item \text{FEAT\_PMUv3p7} is implemented, EL3 is implemented, EL3 is using AArch64, the value of \text{MDCR\_EL3.SPME} is 1, and the value of \text{MDCR\_EL3.MPMX} is 0.
  \item \text{FEAT\_PMUv3p7} is implemented, EL3 is implemented, the PE is not at EL3, EL3 is using AArch64, and the value of \text{MDCR\_EL3.MPMX} is 1.
  \item EL3 is implemented, is using AArch32, and the value of \text{SDCR.SPME} is 1.
  \item EL3 is implemented, EL3 or EL1 is using AArch32, the PE is executing at EL0, and the value of \text{SDER32\_EL3.SUNIDEN} is 1.
  \item If \text{FEAT\_Debugv8p2} is not implemented, EL3 is implemented, and counting is permitted by an \text{IMPLEMENTATION DEFINED} authentication interface, \text{External\_Secure\_Noninvasive\_Debug\_Enabled()} == TRUE.
\end{itemize}

\textbf{Note}

Software can read the Authentication Status register, \text{DBGAUTHSTATUS} to determine the state of an \text{IMPLEMENTATION DEFINED} authentication interface.

If \text{FEAT\_PMUv3p7} is implemented and \text{MDCR\_EL3.MPMX} is 1, counting \textit{Attributable} events at EL3 for event counter \( n \) is prohibited if any of the following are true:

\begin{itemize}
  \item EL2 is not implemented, and \( n \) is in the range \([0 .. (\text{PMCR\_EL0.N}-1)]\).
  \item EL2 is implemented, \text{MDCR\_EL3.SPME} is 0, and \( n \) is in the range \([0 .. (\text{PMCR\_EL0.N}-1)]\).
  \item EL2 is implemented, \text{MDCR\_EL2.HPMN} is not 0, \text{MDCR\_EL3.SPME} is 1, and \( n \) is in the range \([0 .. \text{MDCR\_EL2.HPMN}-1)]\).
\end{itemize}

If EL2 is implemented and \text{MDCR\_EL3.{SPME, MPMX}} is \{1, 1\}, when \text{MDCR\_EL2.HPMN} is less than \text{PMCR\_EL0.N} and \( n \) is in the range \([\text{MDCR\_EL2.HPMN} .. (\text{PMCR\_EL0.N}-1)]\), counting \textit{Attributable} events at EL3 for event counter \( n \) is allowed.

Counting \textit{Attributable} events at EL2 is prohibited unless any of the following are true:

\begin{itemize}
  \item \text{FEAT\_PMUv3p1} is not implemented.
  \item \text{HDCR.HPMD} is not implemented.
\end{itemize}
The Performance Monitors Extension

D11.5 Controlling the PMU counters

- **MDCR_EL2.HPMN** is less than **PMCR_EL0.N**, the event is being counted by event counter *n*, and *n* is in the range \([MDCR_EL2.HPMN .. (PMCR_EL0.N-1)]\).

If **FEAT_SEL2** is implemented, counting **Attributable** events at Secure EL2 is allowed if and only if counting events is allowed in Secure state, and counting events is allowed at EL2.

The accessibility of Performance Monitors registers is unaffected by whether event counting is enabled or prohibited.

The cycle counter, **PMCCNTR**, counts unless any of the following is true:

- The cycle counter is disabled by **PMCR_EL0.E** or **PMCNTENSET_EL0[31]**.
- Event counting by event counters in the range \([0..(HDCR.HPMN-1)]\) is prohibited or frozen, and **PMCR.DP** is set to 1.

--- **Note**

When **FEAT_HPMN0** is implemented and **HDCR.HPMN** is 0, the cycle counter is disabled by **PMCR.DP** under the same conditions that would prohibit or freeze event counting by event counters in the range \([0..(HDCR.HPMN-1)]\) when **HDCR.HPMN** is not 0.

- The PE is in Debug state.
- **FEAT_PMUv3p5** is implemented, EL3 is implemented, the PE is in Secure state, and **SDCR.SCCD** is set to 1.
- **FEAT_PMUv3p5** is implemented, EL2 is implemented, the PE is executing at EL2, and **HDCR.HCCD** is set to 1.
- **FEAT_PMUv3p7** is implemented, the PE is at EL3, EL3 is using AArch64, and **MDCR_EL3.MCCD** is set to 1.

For each Unattributable event, it is IMPLEMENTATION DEFINED whether it is counted when counting **Attributable** events is prohibited.

See **AArch64.CountPMUEvents()** and **AArch32.CountPMUEvents()** in Chapter J1 Armv8 Pseudocode for more information. The CountEvents\(n\) functions return TRUE if PMEVCNTR\(<n>\) is enabled and allowed to count events at the current Exception level or state, and FALSE otherwise. The function CountEvents\(31\) returns TRUE if the cycle counter is enabled and allowed to count cycles at the current Exception level and state and FALSE otherwise. However, these functions do not completely describe the behavior for Unattributable events.

The Performance Monitors are intended to be broadly accurate and statistically useful, see **Accuracy of the Performance Monitors** on page D11-5248. Some inaccuracy is permitted at the point of changing between a state where counting is prohibited and a state where counting is allowed, however. To avoid the leaking of information, the permitted inaccuracy is that transactions that are not prohibited can be uncounted. Where possible, prohibited transactions must not be counted, but if they are counted, then that counting must not degrade security.
D11.6 Multithreaded implementations

If an implementation is multithreaded and the Effective value of PMEVTYPER<\(n\).MT == 1, events on other PEs with the same level 1 Affinity are also counted. A pair of PEs have the same level 1 Affinity if they have the same values for all fields in MPIDR_EL1 or MPIDR except the Aff0 field.

Events on other PEs are not counted when the Effective value of PMEVTYPER<\(n\).MT is 0.

If the CPU implements multithreading, and FEAT_MTPMU is not implemented, for Armv8.5 and earlier, it is IMPLEMENTATION DEFINED whether PMEVTYPER<\(n\).MT is implemented as RW or RES0. From Armv8.6, if the OPTIONAL FEAT_MTPMU feature is not implemented, the Effective value of PMEVTYPER<\(n\).MT is RES0.

If FEAT_MTPMU is implemented, EL3 is implemented, and MDCR_EL3.MTPME is 0 or SDCR.MTPME is 0, FEAT_MTPMU is disabled and the Effective value of PMEVTYPER<\(n\).MT is 0.

If FEAT_MTPMU is implemented, EL3 is not implemented, EL2 is implemented, and MDCR_EL2.MTPME is 0 or HDCR.MTPME is 0, FEAT_MTPMU is disabled and the Effective value of PMEVTYPER<\(n\).MT is 0.

If FEAT_MTPMU is disabled on a Processing Element PEA, it is IMPLEMENTATION DEFINED whether FEAT_MTPMU is disabled on another Processing Element PEB, if all the following are true:

- FEAT_MTPMU is implemented on PEA and PEB.
- PEA and PEB have the same values for Affinity level 1 and higher.
- PEA and PEB both have MPIDR_EL1.MT or MPIDR.MT set to 1.

However, even when the Effective value of PMEVTYPER<\(n\).MT is 1, PEA does not count an event that is Attributable to Secure state on PEB if counting events Attributable to Secure state is prohibited on PEA. Similarly, PEA does not count an event that is Attributable to EL2 on PEB if counting events Attributable to EL2 is prohibited on PEA.

Example D11-1 The effect of having PMEVTYPER<\(n\).MT == 1

If the value of MDCR_EL3.SPME is 0, and \(n\) is less than PMCR.N on PEA, then event counter \(n\) on PEA does not count events Attributable to Secure state on PEB, even if one or both of the following applies:

- PEA is in Non-secure state.
- MDCR_EL3.SPME == 1 on PEB.

---

Example D11-2 The effect of having PMEVTYPER<\(n\).MT == 1

When MDCR_EL2.HPMN is not 0, if the value of MDCR_EL2.HPMD is 1 and \(n\) is less than MDCR_EL2.HPMN on PEA, then event counter \(n\) on PEA does not count events Attributable to EL2 on PEB, even if one of the following applies:

- MDCR_EL2.HPMD == 0 on PEB.
- PEA is not executing at EL2.

---

When the current configuration is not multithreaded, and PEA prohibits counting of events Attributable to Secure state when PEA is in Secure state, it is IMPLEMENTATION DEFINED whether:

- Counting events Attributable to Secure state when PEA is in Non-secure state is permitted.
- Counting Unattributable events related to other Secure operations in the system when PEA is in Non-secure state is permitted.

Otherwise, counting events in Non-secure state is permitted.

When the current configuration is not multithreaded, and PEA prohibited counting of events Attributable to EL2 when PEA is at EL2, it is IMPLEMENTATION DEFINED whether:

- Counting events Attributable to EL2 when PEA is using another Exception level is permitted.
• Counting Unattributable events related to EL2 when PE_A is using another Exception level is permitted. Otherwise, counting events at another Exception level is permitted.
D11.7 Event filtering

The PMU can filter events by various combinations of Exception level and Security state. This gives software the flexibility to count events across multiple processes.

D11.7.1 Filtering by Exception level and Security state

In AArch64 state:

- For each event counter, PMEVTYPER<n>_EL0 specifies the Exception levels in which the counter counts events Attributable to Exception levels.
- PMCCFILTR_EL0 specifies the Exception levels in which the cycle counter counts.

For an event that is Attributable to an Exception level, in a multithreaded implementation:

- When the Effective value of PMEVTYPER<n>_EL0.MT is 1, the specified filtering is evaluated using the current Exception level and Security state of the thread to which the event is Attributable. See Example D11-3.
- When the Effective value of PMEVTYPER<n>_EL0.MT is 0, the event is only counted if it is Attributable to the counting thread, and the filtering is evaluated using the Exception level and Security state of the counting thread.

Example D11-3 Example of the effect of the PMEVTYPER<n>_EL0.MT control

In a multithreaded implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1 and the value of PMEVTYPER<n>_EL0.U is 1 on the counting thread, then event counter n does not count events Attributable to EL0 on another thread, even if the counting thread is not executing at EL0.

For each Unattributable event, it is implementation defined whether the filtering applies. In a multithreaded implementation, if the filtering applies to an Unattributable event, then the filtering is evaluated using the Exception level and Security state of the counting thread.

In AArch32 state, the filtering controls are provided by the PMEVTYPER<n> and PMCCFILTR registers.

For more information, see the individual register descriptions and Multithreaded implementations on page D11-5258.

D11.7.2 Accuracy of event filtering

For most events, it is acceptable that, during a transition between states, events generated by instructions executed in one state are counted in the other state. The following sections describe the cases where event counts must not be counted in the wrong state:

- Exception-related events.
- Software increment events on page D11-5261.

Exception-related events

The PMU must filter events related to exceptions and exception handling according to the Exception level in which the event occurred. These events are:

- EXC_TAKEN, Exception taken.
- EXC_RETURN, Instruction architecturally executed, Condition code check pass, exception return.
- CID_WRITE RETIRED, Instruction architecturally executed, Condition code check pass, write to CONTEXTIDR.
- TTBR_WRITE RETIRED, Instruction architecturally executed, Condition code check pass, write to translation table base.
- EXC_UNDEF, Exception taken, other synchronous.
The Performance Monitors Extension

D11.7 Event filtering

- EXC_SVC, Exception taken, Supervisor Call.
- EXC_PABORT, Exception taken, Instruction Abort.
- EXC_DABORT, Exception taken, Data Abort or SError.
- EXC_IRQ, Exception taken,IRQ.
- EXC_FIQ, Exception taken,FIQ.
- EXC_SMC, Exception taken, Secure Monitor Call.
- EXC_HVC, Exception taken, Hypervisor Call.
- EXC_TRAP_PABORT, Exception taken, Instruction Abort not Taken locally.
- EXC_TRAP_DABORT, Exception taken, Data Abort or SError not Taken locally.
- EXC_TRAP_OTHER, Exception taken, other traps not Taken locally.
- EXC_TRAP_IRQ, Exception taken, IRQ not Taken locally.
- BRB_FILTRATE, Branch record captured.

The PMU must not count an exception after it has been taken because this could systematically report a result of zero exceptions at EL0. Similarly, it is not acceptable for the PMU to count exception returns or writes to CONTEXTIDR after the return from the exception.

Software increment events

The PMU must filter software increment events according to the Exception level in which the software increment occurred. Software increment counting must also be precise, meaning the PMU must count every architecturally executed software increment event, and must not count any Speculatively executed software increment.

Software increment events must also be counted without the need for explicit synchronization. For example, two software increments executed without an intervening Context synchronization event must increment the event counter twice.

For more information, see SW_INCR, Instruction architecturally executed, Condition code check pass, software increment.

D11.7.3 Pseudocode description of event filtering

See AArch64.CountPMUEvents() and AArch32.CountPMUEvents() in Chapter J1 Armv8 Pseudocode for a pseudocode description of event filtering. However, this function does not completely describe the behavior for Unattributable events.
The Performance Monitors Extension

D11.8 Event counting threshold

When `FEAT_PMUv3_TH` is implemented, threshold condition controls are accessible through each `PMEVTYPER<n>_EL0` register. This gives software the ability to count events described by `PMEVTYPER<n>` only when they meet a threshold condition.

D11.8.1 Enabling event counting threshold

When `FEAT_PMUv3_TH` is implemented, threshold counting for event counter `n` is disabled if both of the following are true, and enabled otherwise:

- `PMEVTYPER<n>_EL0.TC` is `0b000`.
- `PMEVTYPER<n>_EL0.TH` is zero.

D11.8.2 Threshold conditions

The `PMEVTYPER<n>_EL0.{TC, TH}` fields define the threshold condition.

If `FEAT_PMUv3_TH` is not implemented, or threshold counting for event counter `n` is disabled, `V` is the amount that the event defined by `PMEVTYPER<n>.{MT, evtCount}` counts by in a given processor cycle.

Otherwise, on each processor cycle, `V` is compared with the value in `PMEVTYPER<n>_EL0.TH` to determine whether it meets the threshold condition. `PMEVTYPER<n>_EL0.TC` determines the threshold condition, and whether the counter increments by `V` or 1 when the threshold condition is met.

`PMMIR_EL1.THWIDTH` describes the maximum value that can be written to `PMEVTYPER<n>_EL0.TH`. The supported threshold conditions are:

- Less-than.
- Greater-than-or-equal-to.
- Not-equals.
- Equal-to.

Example D11-4 Incrementing event counter `n` by `V` when `V` meets the threshold condition

When all of the following are true, the event counter `n` will increment by four:

- `PMEVTYPER<n>_EL0.TC` is `0b010`, equals, meaning threshold counting for event counter `n` is enabled.
- `PMEVTYPER<n>_EL0.TH` is 4.
- `PMEVTYPER<n>.evtCount` is `0x003F`, `STALL SLOT`.
- There are exactly four operation Slots not occupied by an operation Attributable to the PE on the cycle.

Example D11-5 Incrementing event counter `n` by 1 when `V` meets the threshold condition

When all of the following are true, the event counter `n` will increment by one:

- `PMEVTYPER<n>_EL0.TC` is `0b101`, greater-than-or-equal, count, meaning threshold counting for event counter `n` is enabled.
- `PMEVTYPER<n>_EL0.TH` is 2.
- `PMEVTYPER<n>.evtCount` is `0x80C1`, `FP_FIXED_OPS_SPEC`.
- At least one floating-point multiply-add instruction is issued on the cycle.

--- Note ---

The event counter `n` also increments by 1 if, for example, two or more independent floating-point add operations are issued on the cycle.
D11.8.3  Accessing event counting threshold functionality

The PMEVTYPER<n>_EL0.{TC, TH} fields are not accessible through the AArch32 PMEVTYPER<n> System register. However, the threshold condition still applies in AArch32 state, and PMMIR_EL1.THWIDTH is readable in the AArch32 PMMIR System register.

When FEAT_PMUv3_TH is implemented, the PMEVTYPER<n>_EL0.{TC, TH} fields are accessible through the AArch64 PMEVTYPER<n>_EL0 System registers and the external interface PMEVTYPER<n>_EL0 registers. See Chapter 13 Recommended External Interface to the Performance Monitors.

PMMIR_EL1.THWIDTH is readable in the external PMMIR register.

D11.8.4  Pseudocode description of event counting threshold

See PMUCountValue() in Chapter J1 Armv8 Pseudocode for a pseudocode description of the operation of the threshold condition.
**D11.9 Performance Monitors and Debug state**

Events that count cycles are not counted in Debug state.

Events Attributable to the operations issued by the debugger through the external debug interface are not counted in Debug state.

In an implementation that supports multithreading, when the *Effective value* of PMEVTYPE<n>_EL0.MT is 1, if an event is Attributable to an operation issued by the debugger through the external debug interface to another thread that is in Debug state, then the event is not counted, and it is IMPLEMENTATION DEFINED whether the event is counted when the counting thread is in Debug state.

For each Unattributable event, it is IMPLEMENTATION DEFINED whether it is counted when the counting PE is in Debug state. If the event might be counted, then the rules in *Filtering by Exception level and Security state* on page D11-5260 apply for the current Security state in Debug state.
D11.10 Counter access

All implemented event counters are accessible in EL3 and EL2. If EL2 is implemented the hypervisor uses HDCR.HPMN to reserve an event counter, with the effect that if EL2 is enabled in the current Security state, software cannot access that counter and its associated state from EL0 or EL1.

If FEAT_FGT is implemented, if PMSELR.SEL or $n$ indicates an unimplemented event counter, access to PMXEVTYPEP, PMXEVCNTR, PMEVTYPEP<$n$>, or PMEVCNTR<$n$> is UNDEFINED.

--- Note ---

Whether software can access an event counter at an Exception level does not affect whether the counter counts events at that Exception level. For more information, see Controlling the PMU counters on page D11-5254 and Enabling event counters on page D11-5254.

D11.10.1 PMEVCNTR<$n$> event counters

Table D11-4 on page D11-5265 shows how the number of implemented event counters, PMCR.N, and if EL2 is implemented, the value of the HDCR.HPMN field affects the behavior of permitted accesses to the PMEVCNTR<$n$> event counter registers for values of $n$ from 0 to 30.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Access at Exception level</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$ &lt; PMCR.N and either EL2 is not implemented or EL2 is disabled in the</td>
<td>EL3</td>
</tr>
<tr>
<td>current Security state</td>
<td>Succeeds</td>
</tr>
<tr>
<td>$n$ &lt; HDCR.HPMN and EL2 is implemented and enabled in the current</td>
<td>Succeeds</td>
</tr>
<tr>
<td>Security state</td>
<td></td>
</tr>
<tr>
<td>$n$ ≥ HDCR.HPMN and $n$ &lt; PMCR.N and EL2 is implemented and enabled in</td>
<td>Succeeds</td>
</tr>
<tr>
<td>the current Security state</td>
<td></td>
</tr>
<tr>
<td>$n$ ≥ PMCR.N</td>
<td>No access</td>
</tr>
</tbody>
</table>

Where Table D11-4 on page D11-5265 shows access succeeds for an event counter $n$, the access might be UNDEFINED or generate a trap exception. See the descriptions of PMEVCNTR<$n$> and PMXEVCNTR for details.

Where Table D11-4 on page D11-5265 shows no access for an event counter $n$:

- When PMSELR.SEL is $n$, the PE prevents direct reads and direct writes of PMXEVTYPEP or PMXEVCNTR. See the register descriptions for more information.
- The PE prevents direct reads and direct writes of PMEVTYPEP<$n$> or PMEVCNTR<$n$>. See the register descriptions for more information.
- Direct reads and direct writes of the following registers are RAZ/WI, PMOVSCLR[$n$], PMOVSET[$n$], PMCNTENSET[$n$], PMCNTENCLR[$n$], PMINTENSET[$n$], and PMINTENCLR[$n$].
- Direct writes to PMSWINC[$n$] are ignored.
- A direct write of 1 to PMCR.P does not reset PMEVCNTR<$n$>.

D11.10.2 Cycle counter

The PMU does not provide any control that a hypervisor can use to reserve the cycle counter for its own use. However, access to the PMU registers are subject to the access permissions described in Configurable instruction controls on page D1-4665.
D11.11 PMU events and event numbers

The following sections describe the events that can be counted and their associated event numbers, and the mnemonics for the events:

• **Definitions.**
• **The PMU event number space and common events** on page D11-5275.
• **Common event numbers** on page D11-5306.
• **Cycle event counting** on page D11-5421.
• **Meaningful combinations of common microarchitectural events** on page D11-5422.
• **Meaningful combinations of SVE events** on page D11-5424.
• **Required events** on page D11-5425.
• **IMPLEMENTATION DEFINED event numbers** on page D11-5427.

D11.11.1 Definitions

The following subsections give more information about terms used in the event definitions:

• **Definition of terms.**
• **Levels of caches and TLBs** on page D11-5272.
• **Shared caches and buses** on page D11-5272.
• **Counting exceptions taken locally or not taken locally** on page D11-5273.

Definition of terms

This section describes terms used by PMU events as they relate to the implementation of the PMU. For more definitions, see **Glossary.**

**ALU operation counts**

The PMU events 0x80C0 to 0x80C9 count the number of arithmetic logic unit operations performed by each instruction.

Table D11-5 on page D11-5267 gives the ALU operation counts for complex ALU operations. In this table:

- **Input size** The element size of input operands other than the accumulator.
- **Acc size** The element size of the accumulator operand.
- **Count** The number of addition and multiply operations per 128 bits of input:
  • Scalable vector operations increment the counter by the Count value for an applicable *_SCALE_OPS_SPEC event.
  • Advanced SIMD operations operating on a 128-bit register increment the counter by the Count value for an applicable *_FIXED_OPS_SPEC event.
  • Advanced SIMD operations operating on a 64-bit register increment the counter by half the Count value for an applicable *_FIXED_OPS_SPEC event.

- **Type** The data type classification for the operations. This determines for which events the event counter counts the operation.

Predicated operations are counted even if the Governing predicate for the element is FALSE.

**Note**

The FP64 FMUL instruction works on 256-bit segments, and performs 16 operations per 256-bit segment. The table represents counts per 128 bits of input, so the counter increments by 8.
Predicated operations are counted even if the governing predicate for the element is FALSE.

For other instructions, the PMU events that count ALU operations are incremented as follows:

- Multiply-add, multiply-subtract, fused multiply-add, and fused multiply-subtract instructions generate two ALU operations of the specified type per input element. For floating-point operations, these are the instructions counted by FP_FMA_SPEC.
- All other data processing operations generate one ALU operation of the specified type per input element.

The PMU events 0x80CA to 0x80CF count the number of load or store operations performed by each instruction:

- Non-SVE load and store of a single register instructions increment the counter by 1. This includes loads and stores of Sx, Dx, and Qx SIMD&FP registers.
- Non-SVE load and store of a pair of registers instructions increment the counter by 2. This includes loads and stores of pairs of Sx, Dx, and Qx SIMD&FP registers.
- AArch32 load and store multiple registers instructions increment the counter by the number of registers transferred.
- Atomic store instructions increment the counter by 1. These are instructions that atomically update a value in memory without returning a value to a register.
- Atomic load, compare and swap of a single register, and swap instructions increment the counter by 2. Atomic load instructions are instructions that atomically update a value in memory, returning a value to a register.
- Compare and swap of a pair of registers increment the counter by 4.
- SVE and Advanced SIMD LDI instructions increment the counter by 1.
- SVE LDIQ instructions increment the counter by (128 ÷ CSIZE).
- Advanced SIMD LDI[1-4] and ST[1-4] instructions increment the counter by the number of elements transferred per vector multiplied by the number of transferred registers.
- DC ZVA and DC GZVA instructions increment the counter by an IMPLEMENTATION DEFINED amount.

CSIZE

Container size, in bits, that corresponds to the largest non-overlapping SVE or Advanced SIMD vector element size or scalar register size that is encoded in the instruction opcode. This excludes the 64-bit elements of the wide element variants of the SVE bitwise shift and integer compare instructions that overlap the narrower source and destination elements.
Event in progress

Some events count when another event or condition is in progress. This might mean that the event counts the occupancy of a queue or other microarchitectural structure tracking the event. It is usually implementation defined when an event is in progress.

For example, the MEM_ACCESS_RD_PERCYC event counts when the MEM_ACCESS_RD event is in progress, meaning on each Processor cycle, the counter increments by the number of Memory-read operations that are in progress.

These events can be used to calculate the average number of events in progress. In the case of the MEM_ACCESS_RD_PERCYC event, this is also the average read latency. However, the definition of in progress might not include all parts of the operation.

Example D11-6 Stages of an operation not included in the definition of event in progress

In an example implementation, a Memory-read operation generated by a load instruction occupies three pipeline stages in the PE before generating a MEM_ACCESS_RD event when the PE starts to access memory. The event is then considered to be in progress until the data is returned to the PE. In the case of a Normal Cacheable access, the PE first looks in the Level 1 data cache, and if the address is cached, returns data in two cycles. Once the data is returned, it is another cycle before the result can be forwarded to any other instruction.

In this example, if all loads hit in the Level 1 cache, the average read latency calculated using the MEM_ACCESS_RD_PERCYC and MEM_ACCESS_RD events might be two cycles. Although this value is correct for the implementation-specific definition of this event, it has to be adjusted by a constant four additional cycles to match the more commonly understood definition of Level 1 cache access latency, which for this example would be quoted as six cycles.

Instruction architecturally executed

An instruction that is part of the Execution stream. See also Architecturally executed.

A reasonable degree of inaccuracy on page D11-5249 allows for counts to be inaccurate in exceptional circumstances. For an event that counts instructions architecturally executed, this allows an implementation to count instructions that do not form part of the Execution stream because of an exceptional event, such as if the instruction generates a synchronous exception or entry to Debug state.

Instructions that have no visible effect on the architectural state of the PE are architecturally executed and counted even if they form part of the Execution stream.

Examples of instructions that have no visible effect are:

• A NOP.
• A conditional instruction that fails its Condition code check.
• A Compare and Branch on Zero, CBZ, instruction that does not branch.
• A Compare and Branch on Nonzero, CBNZ, instruction that does not branch.

However, for events that count only the execution of instructions that update System registers, such as CID_WRITE RETIRED, if such an instruction is executed twice without an intervening Context synchronization event, it is constrained unpredictable whether the first instruction is counted.

Instruction architecturally executed, Condition code check pass

Instruction architecturally executed, Condition code check pass is a class of events that explicitly do not occur for:

• A conditional instruction that fails its Condition code check.
• A Compare and Branch on Zero, CBZ, instruction that does not branch.
• A Compare and Branch on Nonzero, CBNZ, instruction that does not branch.
• A Test and Branch on Zero, TBZ, instruction that does not branch.
• A Test and Branch on Nonzero, TBNZ, instruction that does not branch.
• A Store-Exclusive instruction that does not write to memory.
Otherwise, the definition of architecturally executed is the same as for Instruction architecturally executed.

A branch that is architecturally executed, with condition code check pass is also described as a branch taken.

**Instruction memory access**

A PE acquires instructions for execution through instruction fetches. Instruction fetches might be due to:
- Fetching instructions that are architecturally executed.
- The result of the execution of an instruction preload instruction, PLI.
- Speculation that a particular instruction might be executed in the future.

The relationship between the fetch of an individual instruction and an instruction memory access is IMPLEMENTATION DEFINED. For example, an implementation might fetch many instructions including a non-integer number of instructions in a single instruction memory access.

**Memory-read operations**

A PE accesses memory through memory-read operations and Memory-write operations. A memory-read operation might be due to:
- The result of an architecturally executed memory-reading instructions.
- The result of a Speculatively executed memory-reading instructions.
- A translation table walk.

For levels of cache hierarchy beyond the Level 1 caches, memory-read operations also include accesses made as part of a refill of another cache closer to the PE. Such refills might be due to:
- Memory-read operations or Memory-write operations that miss in the cache
- The execution of a data preload instruction.
- The execution of a cache maintenance instruction.

--- **Note** ---

A preload instruction or cache maintenance instruction is not, in itself, an access to that cache. However, it might generate cache refills which are then treated as memory-read operations beyond that cache.

- Speculation that a future instruction might access the memory location.
- Instruction memory accesses to a unified cache, when the cache does not implement the L<n>I_CACHE event.

This list is not exhaustive.

The relationship between memory-read instructions and memory-read operations is IMPLEMENTATION DEFINED. For example, for some implementations an LDPE instruction that reads two 64-bit registers might generate one memory-read operation if the address is quadword-aligned, but for other addresses it generates two or more memory-read operations.

**Memory-write operations**

Memory-write operations might be due to:
- The result of an architecturally executed memory-writing instructions.
- The result of a Speculatively executed memory-writing instructions.

--- **Note** ---

Speculatively executed memory-writing instructions that do not become architecturally executed must not alter the architecturally defined view of memory. They can, however, generate a memory-write operation that is later undone in some implementation specific way.
For levels of cache hierarchy beyond the Level 1 caches, memory-write operations also include accesses made as part of a write-back from another cache closer to the PE. Such write-backs might be due to:

- Evicting a dirty line from the cache, to allocate a cache line for a cache refill, see [Memory-read operations](#).
- The execution of a cache maintenance instruction.

--- **Note** ---

A cache maintenance instruction is not in itself an access to that cache. However, it might generate write-backs which are then treated as memory-write operations beyond that cache.

---

- The result of a coherency request from another PE.

This list is not exhaustive.

**DC ZVA** is counted as a [Memory-write operation](#). **ST64BV** and **ST64BV0** are Store with Return instructions, but for the purpose of the PMU they are treated as Memory-write operations.

The relationship between memory-writing instructions and memory-write operations is [IMPLEMENTATION DEFINED](#). For example, for some implementations an **STP** instruction that writes two 64-bit registers might generate one memory-write operation if the address is quadword-aligned, but for other addresses it generates two or more memory-write operations. In some implementations, the result of two **STR** instructions that write to adjacent memory might be merged into a single memory-write operation.

--- **Note** ---

The data written back from a cache that is shared with other PEs might not be data that was written by the PE that performs the operation that leads to the write-back. Nevertheless, the event is counted as a write-back event for that PE.

**MSIZE**

Memory element access size, in bits, that corresponds to a load or store instruction mnemonic suffix, where B=8, H=16, W=32 and D=64. When an instruction mnemonic does not end with B, H, W or D, the memory access size is implied by the scalar transfer register size or SIMD transfer register element size.

**Non-SIMD SVE instructions**

These instructions are:

- Vector address calculation instructions, **ADDPL**, **ADDVL**, and **RDVL**.
- The scalar **Predicate counts** instructions.
- The compare and terminate instructions, **CTERMEQ**, **CTERMNE**.

**Operation**

An instruction might create one or more microarchitectural operations (μ-ops) at any point in the execution pipeline. Depending on the event definition, the μ-ops might be counted instead of instructions. The definition of a μ-op is implementation specific. An architecture instruction might create more than one μ-op for each instruction. μ-ops might also be removed or merged in the execution stream, so an architecture instruction might create no μ-ops for an instruction. Any arbitrary translation of instructions to an equivalent sequence of μ-ops is permitted.

This means the relationship between a μ-op and an architecturally executed instruction is [IMPLEMENTATION DEFINED](#).

--- **Note** ---

The architecture does not require that an implementation that generates μ-ops must count μ-ops for operations. An implementation can choose to interpret operation as instruction.
The counting of operations can indicate the workload on the PE. However, there is no requirement for operations to represent similar amounts of work, and direct comparisons between different microarchitectures are not meaningful.

Operations might be defined with reference to a particular instruction or type of instruction. In the case of operations this means operations with semantics that map to that type of instruction. For example, an implementation splits an A32 or T32 LDM instruction of six registers into six μ-ops, one for each load, and a seventh address-generation operation to determine the base address or writeback address. Also, for doubleword alignment, the six load μ-ops might combine into four operations, that is, a word load, two doubleword loads, and a second word load. This single instruction can then be counted as five, or possibly six, events:

- Four (Operations speculatively executed - Load) events.
- One (Operations speculatively executed - Integer data processing) event.
- One (Operations speculatively executed - Software change of the PC) event if the PC was one of the six registers in the LDM instruction.

Operations speculatively executed

A Operation that is Speculatively executed.

There is no architecturally guaranteed relationship between a Speculatively executed micro-op and an architecturally executed instruction.

The results of such an operation can also be discarded, if it transpires that the operation was not required, following a mispredicted branch. Therefore, the architecture defines these events as operations speculatively executed, where appropriate.

Note

In some events, operation has a more specific meaning described in the event. See ALU operation counts on page D11-5266.

Processor cycle

For a non-multithreaded implementation, this means a cycle of the processor. For a multithreaded implementation, processor cycle means each cycle of the multithreaded processor, not just those cycles for which the PE counting the event is active.

Slot

An implementation of a PE might be able to execute multiple micro-ops in a single processor cycle. The maximum number of micro-ops that can be executed might vary at different points in the execution pipeline.

To allow profiling of the utilization of the resource of the PE, an implementation specific point in the execution pipeline is chosen where the maximum number of micro-ops that can be executed is an IMPLEMENTATION DEFINED fixed value.

Each possible micro-op that can be executed at that point in a cycle is called a Slot. The maximum number of micro-ops that can be executed is defined by PMMIR.SLOTS.

Software change of the PC

Some events relate to instructions that cause a software change of the PC. This includes all:

- Branch instructions.
- Memory-reading instructions that explicitly write to the PC.
- Data-processing instructions that explicitly write to the PC.
- Exception return instructions.

It is IMPLEMENTATION DEFINED whether any or all of the following are treated as software changes of the PC:

- BRK and BKPT instructions.
- An exception generated because an instruction is UNDEFINED.
- The exception-generating instructions, SVC, HVC, and SMC.
- Context synchronization barrier ISB instructions.
Speculatively executed

An instruction or operation that is counted by an event when it might be Speculative.

The architecture does not define the point in a pipeline where an event is counted. For some events, this means the operation or instruction is counted when the operation or instruction is Speculative. The results of such an operation or instruction might later be discarded, if it transpires that the operation was not required, such as following a mispredicted branch, or might be later resolved to be Architecturally executed.

Different groups of events might be counted at different points in the pipeline and so can have different IMPLEMENTATION DEFINED definitions of speculatively executed. Such groups share a common base type, which the event name denotes. Each of the events in the previous example is of the base type, operation speculatively executed.

For groups of events with a common base type, speculatively executed operations are all counted on the same basis, which normally means at the same point in the pipeline. It is possible to compare the counts and make meaningful observations about the program being profiled.

VL

The current SVE vector length, in bits.

Levels of caches and TLBs

The mapping of levels of cache and TLB to the PMU events is IMPLEMENTATION DEFINED. Although CLIDR_EL1 and CLIDR define the implemented levels of cache, these are not required to correspond with the levels of cache defined for PMU events. The architecture does not provide any way of determining implemented levels of TLB. Also, many implementations include structures that provide some caching at a higher level than the level 1 caches or TLBs. Typically, these structures, that might be called Level 0 caches, or mini caches, or microcaches, are invisible to software. The implementation-specific nature of cache and TLB implementations mean that, in general, PMU event counts cannot be used reliably to make direct comparisons between different implementations, and Arm recommends the following implementation guidelines:

- If L3D_CACHE events are implemented, then L2D_CACHE and if applicable L2I_CACHE events should be implemented.
- If L2D_CACHE events are implemented, then L1D_CACHE and if applicable L1I_CACHE events should be implemented.
- If L2I_CACHE events are implemented, then L1I_CACHE events should be implemented.
- Where the Last Level of cache is also the Level 3 or Level 2 unified cache, the LL_CACHE events should be implemented in preference to the L3D_CACHE or L2D_CACHE events as applicable.
- For the Level <n> cache, where <n> = 1 and <n> = 2:
  - If the Level <n> cache is unified, but the cache can disambiguate between Data and Instruction accesses to the cache, then both the L<n>D_CACHE and L<n>I_CACHE events should be implemented.
  - If the cache is unified and the cache cannot disambiguate between Data and Instruction accesses, then only the L<n>D_CACHE should be implemented, counting all accesses.

This final property is IMPLEMENTATION DEFINED.

Shared caches and buses

There is no architectural concept of a shared component. However, when a cache, a bus, or any other system component that might generate countable events is implemented, and:

- The extent of the first-order effects due to an event from that component are only applicable to a single PE, then the event is not shared.
- Otherwise, the event is shared.

Second-order effects are not considered when determining if an event is shared.
Example D11-7 First and second order effects of a cache miss in a multiple-PE implementation

In an implementation that consists of two PEs, each with its own L1 cache, a cache miss by one of the PEs is a first-order effect of an access to its cache. Any snoop that is performed on the L1 cache of the other PE in the implementation as a result of that cache miss is a second order effect.

Note

Shared events are inherently linked to microarchitectures and so the implementer must make an informed decision about how such events are implemented.

Counting exceptions taken locally or not taken locally

Table D11-6 on page D11-5273 shows the events for exceptions taken to an Exception level using AArch64.

### Table D11-6 Events for exceptions taken to an Exception level using AArch64

<table>
<thead>
<tr>
<th>ESR.EC</th>
<th>Description</th>
<th>Event number and classification for exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Unknown or uncategorized</td>
<td>0x0081, EXC_UNDEF</td>
</tr>
<tr>
<td>0x01</td>
<td>!F+ traps</td>
<td>0x0081, EXC_UNDEF</td>
</tr>
<tr>
<td>0x03</td>
<td>AArch32 MCR/MRC traps on (coproc==0b1111) accesses</td>
<td>0x0081, EXC_UNDEF</td>
</tr>
<tr>
<td>0x04</td>
<td>AArch32 MRCR/MRRC traps on (coproc==0b1111) accesses</td>
<td>0x0081, EXC_UNDEF</td>
</tr>
<tr>
<td>0x05</td>
<td>AArch32 MCR/MRC traps on (coproc==0b1110) accesses</td>
<td>0x0081, EXC_UNDEF</td>
</tr>
<tr>
<td>0x06</td>
<td>AArch32 LDC/STC traps on (coproc==0b1110) accesses</td>
<td>0x0081, EXC_UNDEF</td>
</tr>
<tr>
<td>0x07</td>
<td>Advanced SIMD or FP traps</td>
<td>0x0081, EXC_UNDEF</td>
</tr>
<tr>
<td>0x08</td>
<td>AArch32 MVFR* and FPSID traps</td>
<td>0x0081, EXC_UNDEF</td>
</tr>
<tr>
<td>0x0C</td>
<td>AArch32 MRCR/MRRC traps on (coproc==0b1110) accesses</td>
<td>0x0081, EXC_UNDEF</td>
</tr>
<tr>
<td>0x0E</td>
<td>Illegal instruction set state</td>
<td>0x0081, EXC_UNDEF</td>
</tr>
<tr>
<td>0x11</td>
<td>AArch32 SVC</td>
<td>0x0082, EXC_SVC</td>
</tr>
<tr>
<td>0x12</td>
<td>AArch32 HVC that is not disabled</td>
<td>0x0082, EXC_HVC</td>
</tr>
<tr>
<td>0x13</td>
<td>AArch32 SMC that is not disabled</td>
<td>0x0082, EXC_SMC</td>
</tr>
<tr>
<td>0x15</td>
<td>AArch64 SVC</td>
<td>0x0082, EXC_SVC</td>
</tr>
<tr>
<td>0x16</td>
<td>AArch64 HVC that is not disabled</td>
<td>0x008A, EXC_HVC</td>
</tr>
</tbody>
</table>
### Table D11-6 Events for exceptions taken to an Exception level using AArch64 (continued)

<table>
<thead>
<tr>
<th>ESR,EC</th>
<th>Description</th>
<th>Event number and classification for exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Taken locally</strong></td>
</tr>
<tr>
<td>0x17</td>
<td>AArch64 SMC that is not disabled</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0088, EXC_SMC</td>
</tr>
<tr>
<td>0x18</td>
<td>AArch64 MSR, MRS and System instruction traps</td>
<td>0x0081, EXC_UNDEF</td>
</tr>
<tr>
<td>0x19</td>
<td>SVE traps</td>
<td>0x0081, EXC_UNDEF</td>
</tr>
<tr>
<td>0x1F</td>
<td>IMPLEMENTATION DEFINED exception taken to EL3</td>
<td>IMPLEMENTATION DEFINED&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>0x20</td>
<td>Instruction Abort from below</td>
<td>0x0083, EXC_PABORT</td>
</tr>
<tr>
<td>0x21</td>
<td>Instruction Abort from current Exception level</td>
<td>0x0083, EXC_PABORT</td>
</tr>
<tr>
<td>0x22</td>
<td>PC alignment</td>
<td>0x0083, EXC_PABORT</td>
</tr>
<tr>
<td>0x24</td>
<td>Data Abort from below</td>
<td>0x0084, EXC_DABORT</td>
</tr>
<tr>
<td>0x25</td>
<td>Data Abort from current Exception level</td>
<td>0x0084, EXC_DABORT</td>
</tr>
<tr>
<td>0x26</td>
<td>SP alignment fault exception</td>
<td>0x0084, EXC_DABORT</td>
</tr>
<tr>
<td>0x28</td>
<td>AArch32 FP exception</td>
<td>0x0081, EXC_UNDEF</td>
</tr>
<tr>
<td>0x2C</td>
<td>AArch64 FP exception</td>
<td>0x0081, EXC_UNDEF</td>
</tr>
<tr>
<td>0x2F</td>
<td>SError interrupt</td>
<td>0x0084, EXC_DABORT</td>
</tr>
<tr>
<td>0x30</td>
<td>Breakpoint from below</td>
<td>0x0083, EXC_PABORT</td>
</tr>
<tr>
<td>0x31</td>
<td>Breakpoint from current Exception level</td>
<td>0x0083, EXC_PABORT</td>
</tr>
<tr>
<td>0x32</td>
<td>Software step from below</td>
<td>0x0083, EXC_PABORT</td>
</tr>
<tr>
<td>0x33</td>
<td>Software step from current Exception level</td>
<td>0x0083, EXC_PABORT</td>
</tr>
<tr>
<td>0x34</td>
<td>Watchpoint from below</td>
<td>0x0084, EXC_DABORT</td>
</tr>
<tr>
<td>0x35</td>
<td>Watchpoint from current Exception level</td>
<td>0x0084, EXC_DABORT</td>
</tr>
<tr>
<td>0x38</td>
<td>AArch32 BKPT instruction</td>
<td>0x0083, EXC_PABORT</td>
</tr>
<tr>
<td>0x3A</td>
<td>AArch32 Vector Catch debug event</td>
<td>0x0083, EXC_PABORT</td>
</tr>
<tr>
<td>0x3C</td>
<td>AArch64 BK instruction</td>
<td>0x0083, EXC_PABORT</td>
</tr>
<tr>
<td>-</td>
<td>IRQ interrupt</td>
<td>0x0086, EXC_IRQ</td>
</tr>
<tr>
<td>-</td>
<td>FIQ interrupt</td>
<td>0x0087, EXC_FIQ</td>
</tr>
<tr>
<td>All other values</td>
<td>All other exceptions</td>
<td>0x0081, EXC_UNDEF</td>
</tr>
</tbody>
</table>

<sup>a</sup> Implementation defined.
The Performance Monitors Extension

D11.11 PMU events and event numbers

The Glossary defines the term Taken locally, that is used in event definitions in Common microarchitectural events on page D11-5317. See also Exception levels on page D1-4632 for more information.

D11.11.2 The PMU event number space and common events

In Armv8.0, the event number space is 10 bits. Armv8.1 extends the event number space, and therefore the PMEVTYPE<n>_EL0.evtCount field to 16 bits, and is allocated as Table D11-7 on page D11-5275 shows. For more information about the entries in the Allocation on page D11-5275 column see the text that follows this table:

<table>
<thead>
<tr>
<th>Event numbers</th>
<th>Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>In all versions from Armv8.0</td>
<td></td>
</tr>
<tr>
<td>0x0000-0x003F</td>
<td>Common architectural and microarchitectural events.</td>
</tr>
<tr>
<td>0x0040-0x00BF</td>
<td>When FEAT_PMUv3p8 is implemented, common architectural and microarchitectural events. Previously Arm-recommended common architectural and microarchitectural events.</td>
</tr>
<tr>
<td>0x00C0-0x03FF</td>
<td>IMPLEMENTATION DEFINED events.</td>
</tr>
<tr>
<td>From Armv8.1</td>
<td></td>
</tr>
<tr>
<td>0x0400-0x3FFF</td>
<td>IMPLEMENTATION DEFINED events.</td>
</tr>
<tr>
<td>0x4000-0x403F</td>
<td>Common architectural and microarchitectural events.</td>
</tr>
<tr>
<td>0x4040-0x40BF</td>
<td>When FEAT_PMUv3p8 is implemented, common architectural and microarchitectural events. Previously Arm-recommended common architectural and microarchitectural events.</td>
</tr>
<tr>
<td>0x40C0-0x7FFF</td>
<td>IMPLEMENTATION DEFINED events.</td>
</tr>
<tr>
<td>0x8000-0x80FF</td>
<td>Common architectural and microarchitectural events.</td>
</tr>
<tr>
<td>0x8100-0x81FF</td>
<td>From Armv8.6, common architectural and microarchitectural events. Previously reserved.</td>
</tr>
<tr>
<td>0x8200-0xC0BF</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0xC0C0-0xFFFF</td>
<td>IMPLEMENTATION DEFINED events.</td>
</tr>
</tbody>
</table>

The meaning of the entries in the Allocation on page D11-5275 column of Table D11-7 on page D11-5275 is as follows:

Common architectural and microarchitectural events

Arm defines the use of these event numbers. For more information see Common event numbers on page D11-5306.

a. The exception reported with EC 0x1F is IMPLEMENTATION DEFINED, and therefore it is IMPLEMENTATION DEFINED which event counts the exception, except that the event that counts the exception must correctly indicate whether the exception was Taken locally.
IMPLEMENTATION DEFINED event numbers

For more information about the use of these event numbers see IMPLEMENTATION DEFINED event numbers on page D11-5427.

See PMEVTYPERN.EVT Count for details of the PE behavior when an event number for a reserved or unimplemented PMU event is written to evtCount.

Table D11-8 on page D11-5276 lists the number and mnemonic of PMU events.

<table>
<thead>
<tr>
<th>Event number</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>SW_INCR</td>
<td>Instruction architecturally executed, Condition code check pass, software increment.</td>
</tr>
<tr>
<td>0x0001</td>
<td>L1I_CACHE_REFILL</td>
<td>Level 1 instruction cache refill.</td>
</tr>
<tr>
<td>0x0002</td>
<td>L1I_TLB_REFILL</td>
<td>Level 1 instruction TLB refill.</td>
</tr>
<tr>
<td>0x0003</td>
<td>L1D_CACHE_REFILL</td>
<td>Level 1 data cache refill.</td>
</tr>
<tr>
<td>0x0004</td>
<td>L1D_CACHE</td>
<td>Level 1 data cache access.</td>
</tr>
<tr>
<td>0x0005</td>
<td>L1D_TLB_REFILL</td>
<td>Level 1 data TLB refill.</td>
</tr>
<tr>
<td>0x0006</td>
<td>LD RETIRED</td>
<td>Instruction architecturally executed, Condition code check pass, load.</td>
</tr>
<tr>
<td>0x0007</td>
<td>ST RETIRED</td>
<td>Instruction architecturally executed, Condition code check pass, store.</td>
</tr>
<tr>
<td>0x0008</td>
<td>INST RETIRED</td>
<td>Instruction architecturally executed.</td>
</tr>
<tr>
<td>0x0009</td>
<td>EXC TAKEN</td>
<td>Exception taken.</td>
</tr>
<tr>
<td>0x000A</td>
<td>EXC RETURN</td>
<td>Instruction architecturally executed, Condition code check pass, exception return.</td>
</tr>
<tr>
<td>0x000B</td>
<td>CID WRITE RETIRED</td>
<td>Instruction architecturally executed, Condition code check pass, write to CONTEXTIDR.</td>
</tr>
<tr>
<td>0x000C</td>
<td>PC WRITE RETIRED</td>
<td>Instruction architecturally executed, Condition code check pass, Software change of the PC.</td>
</tr>
<tr>
<td>0x000D</td>
<td>BR_IMMED RETIRED</td>
<td>Branch Instruction architecturally executed, immediate.</td>
</tr>
<tr>
<td>0x000E</td>
<td>BR RETURN RETIRED</td>
<td>Branch Instruction architecturally executed, procedure return, taken.</td>
</tr>
<tr>
<td>0x000F</td>
<td>UNALIGNED LDST RETIRED</td>
<td>Instruction architecturally executed, Condition code check pass, unaligned load or store.</td>
</tr>
<tr>
<td>0x0010</td>
<td>BR MIS PRED</td>
<td>Branch instruction Speculatively executed, mispredicted or not predicted.</td>
</tr>
<tr>
<td>0x0011</td>
<td>CPU CYCLES</td>
<td>Cycle.</td>
</tr>
<tr>
<td>0x0012</td>
<td>BR PRED</td>
<td>Predictable branch instruction Speculatively executed.</td>
</tr>
<tr>
<td>0x0013</td>
<td>MEM ACCESS</td>
<td>Data memory access.</td>
</tr>
<tr>
<td>0x0014</td>
<td>L1I CACHE</td>
<td>Level 1 instruction cache access.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x0015</td>
<td>L1D_CACHE_WB</td>
<td>Level 1 data cache write-back.</td>
</tr>
<tr>
<td>0x0016</td>
<td>L2D_CACHE</td>
<td>Level 2 data cache access.</td>
</tr>
<tr>
<td>0x0017</td>
<td>L2D_CACHE_REFILL</td>
<td>Level 2 data cache refill.</td>
</tr>
<tr>
<td>0x0018</td>
<td>L2D_CACHE_WB</td>
<td>Level 2 data cache write-back.</td>
</tr>
<tr>
<td>0x0019</td>
<td>BUS_ACCESS</td>
<td>Bus access.</td>
</tr>
<tr>
<td>0x001A</td>
<td>MEMORY_ERROR</td>
<td>Local memory error.</td>
</tr>
<tr>
<td>0x001B</td>
<td>INST_SPEC</td>
<td>Operation speculatively executed.</td>
</tr>
<tr>
<td>0x001C</td>
<td>TTBR_WRITE_RETIRED</td>
<td>Instruction architecturally executed, Condition code check pass, write to TTBR.</td>
</tr>
<tr>
<td>0x001D</td>
<td>BUS_CYCLES</td>
<td>Bus cycle.</td>
</tr>
<tr>
<td>0x001E</td>
<td>CHAIN</td>
<td>Chain a pair of event counters.</td>
</tr>
<tr>
<td>0x001F</td>
<td>L1D_CACHE_ALLOCATE</td>
<td>Level 1 data cache allocation without refill.</td>
</tr>
<tr>
<td>0x0020</td>
<td>L2D_CACHE_ALLOCATE</td>
<td>Level 2 data cache allocation without refill.</td>
</tr>
<tr>
<td>0x0021</td>
<td>BR_RETIRED</td>
<td>Instruction architecturally executed, branch.</td>
</tr>
<tr>
<td>0x0022</td>
<td>BR_MIS_PRED_RETIRED</td>
<td>Branch Instruction architecturally executed, mispredicted.</td>
</tr>
<tr>
<td>0x0023</td>
<td>STALL/frontend</td>
<td>No operation sent for execution due to the frontend.</td>
</tr>
<tr>
<td>0x0024</td>
<td>STALL/backend</td>
<td>No operation sent for execution due to the backend.</td>
</tr>
<tr>
<td>0x0025</td>
<td>L1D_TLB</td>
<td>Level 1 data TLB access.</td>
</tr>
<tr>
<td>0x0026</td>
<td>L1I_TLB</td>
<td>Level 1 instruction TLB access.</td>
</tr>
<tr>
<td>0x0027</td>
<td>L2I_CACHE</td>
<td>Level 2 instruction cache access.</td>
</tr>
<tr>
<td>0x0028</td>
<td>L2I_CACHE_REFILL</td>
<td>Level 2 instruction cache refill.</td>
</tr>
<tr>
<td>0x0029</td>
<td>L3D_CACHE_ALLOCATE</td>
<td>Level 3 data cache allocation without refill.</td>
</tr>
<tr>
<td>0x002A</td>
<td>L3D_CACHE_REFILL</td>
<td>Level 3 data cache refill.</td>
</tr>
<tr>
<td>0x002B</td>
<td>L3D_CACHE</td>
<td>Level 3 data cache access.</td>
</tr>
<tr>
<td>0x002C</td>
<td>L3D_CACHE_WB</td>
<td>Level 3 data cache write-back.</td>
</tr>
<tr>
<td>0x002D</td>
<td>L2D_TLB_REFILL</td>
<td>Level 2 data TLB refill.</td>
</tr>
<tr>
<td>0x002E</td>
<td>L2I_TLB_REFILL</td>
<td>Level 2 instruction TLB refill.</td>
</tr>
<tr>
<td>0x002F</td>
<td>L2D_TLB</td>
<td>Level 2 data TLB access.</td>
</tr>
<tr>
<td>0x0030</td>
<td>L2I_TLB</td>
<td>Level 2 instruction TLB access.</td>
</tr>
<tr>
<td>0x0031</td>
<td>REMOTE_ACCESS</td>
<td>Access to another socket in a multi-socket system.</td>
</tr>
<tr>
<td>0x0032</td>
<td>LL_CACHE</td>
<td>Last level cache access.</td>
</tr>
</tbody>
</table>
### Table D11-8 Event index (continued)

<table>
<thead>
<tr>
<th>Event number</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0033</td>
<td>LL_CACHE_MISS</td>
<td>Last level cache miss.</td>
</tr>
<tr>
<td>0x0034</td>
<td>DTLB_WALK</td>
<td>Data TLB access with at least one translation table walk.</td>
</tr>
<tr>
<td>0x0035</td>
<td>ITLB_WALK</td>
<td>Instruction TLB access with at least one translation table walk.</td>
</tr>
<tr>
<td>0x0036</td>
<td>LL_CACHE_RD</td>
<td>Last level cache access, read.</td>
</tr>
<tr>
<td>0x0037</td>
<td>LL_CACHE_MISS_RD</td>
<td>Last level cache miss, read.</td>
</tr>
<tr>
<td>0x0038</td>
<td>REMOTE_ACCESS_RD</td>
<td>Access to another socket in a multi-socket system, read.</td>
</tr>
<tr>
<td>0x0039</td>
<td>L1D_CACHE_LMISS_RD</td>
<td>Level 1 data cache long-latency read miss.</td>
</tr>
<tr>
<td>0x003A</td>
<td>OP_RETIRED</td>
<td>Micro-operation architecturally executed.</td>
</tr>
<tr>
<td>0x003B</td>
<td>OP_SPEC</td>
<td>Micro-operation Speculatively executed.</td>
</tr>
<tr>
<td>0x003C</td>
<td>STALL</td>
<td>No operation sent for execution.</td>
</tr>
<tr>
<td>0x003D</td>
<td>STALL_SLOT_BACKEND</td>
<td>No operation sent for execution on a Slot due to the backend.</td>
</tr>
<tr>
<td>0x003E</td>
<td>STALL_SLOT_FRONTEND</td>
<td>No operation sent for execution on a Slot due to the frontend.</td>
</tr>
<tr>
<td>0x003F</td>
<td>STALL_SLOT</td>
<td>No operation sent for execution on a Slot.</td>
</tr>
<tr>
<td>0x0040</td>
<td>L1D_CACHE_RD</td>
<td>Level 1 data cache access, read.</td>
</tr>
<tr>
<td>0x0041</td>
<td>L1D_CACHE_WR</td>
<td>Level 1 data cache access, write.</td>
</tr>
<tr>
<td>0x0042</td>
<td>L1D_CACHE_REFILL_RD</td>
<td>Level 1 data cache refill, read.</td>
</tr>
<tr>
<td>0x0043</td>
<td>L1D_CACHE_REFILL_WR</td>
<td>Level 1 data cache refill, write.</td>
</tr>
<tr>
<td>0x0044</td>
<td>L1D_CACHE_REFILL_INNER</td>
<td>Level 1 data cache refill, inner.</td>
</tr>
<tr>
<td>0x0045</td>
<td>L1D_CACHE_REFILL_OUTER</td>
<td>Level 1 data cache refill, outer.</td>
</tr>
<tr>
<td>0x0046</td>
<td>L1D_CACHE_WB_VICTIM</td>
<td>Level 1 data cache write-back, victim.</td>
</tr>
<tr>
<td>0x0047</td>
<td>L1D_CACHE_WB_CLEAN</td>
<td>Level 1 data cache write-back, cleaning and coherency.</td>
</tr>
<tr>
<td>0x0048</td>
<td>L1D_CACHE_INVAL</td>
<td>Level 1 data cache invalidate.</td>
</tr>
<tr>
<td>0x0049</td>
<td>L1D_TLB_REFILL_RD</td>
<td>Level 1 data TLB refill, read.</td>
</tr>
<tr>
<td>0x004A</td>
<td>L1D_TLB_REFILL_WR</td>
<td>Level 1 data TLB refill, write.</td>
</tr>
<tr>
<td>0x004B</td>
<td>L1D_TLB_RD</td>
<td>Level 1 data TLB access, read.</td>
</tr>
<tr>
<td>0x004C</td>
<td>L1D_TLB_WR</td>
<td>Level 1 data TLB access, write.</td>
</tr>
<tr>
<td>0x0050</td>
<td>L2D_CACHE_RD</td>
<td>Level 2 data cache access, read.</td>
</tr>
<tr>
<td>0x0051</td>
<td>L2D_CACHE_WR</td>
<td>Level 2 data cache access, write.</td>
</tr>
<tr>
<td>0x0052</td>
<td>L2D_CACHE_REFILL_RD</td>
<td>Level 2 data cache refill, read.</td>
</tr>
<tr>
<td>0x0053</td>
<td>L2D_CACHE_REFILL_WR</td>
<td>Level 2 data cache refill, write.</td>
</tr>
</tbody>
</table>
Table D11-8 Event index (continued)

<table>
<thead>
<tr>
<th>Event number</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0056</td>
<td>L2D_CACHE_WB_VICTIM</td>
<td>Level 2 data cache write-back, victim.</td>
</tr>
<tr>
<td>0x0057</td>
<td>L2D_CACHE_WB_CLEAN</td>
<td>Level 2 data cache write-back, cleaning and coherency.</td>
</tr>
<tr>
<td>0x0058</td>
<td>L2D_CACHE_INVAL</td>
<td>Level 2 data cache invalidate.</td>
</tr>
<tr>
<td>0x005C</td>
<td>L2D_TLB_REFILL_RD</td>
<td>Level 2 data TLB refill, read.</td>
</tr>
<tr>
<td>0x005D</td>
<td>L2D_TLB_REFILL_WR</td>
<td>Level 2 data TLB refill, write.</td>
</tr>
<tr>
<td>0x005E</td>
<td>L2D_TLB_RD</td>
<td>Level 2 data TLB access, read.</td>
</tr>
<tr>
<td>0x005F</td>
<td>L2D_TLB_WR</td>
<td>Level 2 data TLB access, write.</td>
</tr>
<tr>
<td>0x0060</td>
<td>BUS_ACCESS_RD</td>
<td>Bus access, read.</td>
</tr>
<tr>
<td>0x0061</td>
<td>BUS_ACCESS_WR</td>
<td>Bus access, write.</td>
</tr>
<tr>
<td>0x0062</td>
<td>BUS_ACCESS_SHARED</td>
<td>Bus access, Normal, Cacheable, Shareable.</td>
</tr>
<tr>
<td>0x0063</td>
<td>BUS_ACCESS_NOT_SHARED</td>
<td>Bus access, not Normal, Cacheable, Shareable.</td>
</tr>
<tr>
<td>0x0064</td>
<td>BUS_ACCESS_NORMAL</td>
<td>Bus access, normal.</td>
</tr>
<tr>
<td>0x0065</td>
<td>BUS_ACCESS_PERIPH</td>
<td>Bus access, peripheral.</td>
</tr>
<tr>
<td>0x0066</td>
<td>MEM_ACCESS_RD</td>
<td>Data memory access, read.</td>
</tr>
<tr>
<td>0x0067</td>
<td>MEM_ACCESS_WR</td>
<td>Data memory access, write.</td>
</tr>
<tr>
<td>0x0068</td>
<td>UNALIGNED_LD_SPEC</td>
<td>Unaligned access, read.</td>
</tr>
<tr>
<td>0x0069</td>
<td>UNALIGNED_ST_SPEC</td>
<td>Unaligned access, write.</td>
</tr>
<tr>
<td>0x006A</td>
<td>UNALIGNED_LDST_SPEC</td>
<td>Unaligned access.</td>
</tr>
<tr>
<td>0x006C</td>
<td>LDREX_SPEC</td>
<td>Exclusive operation Speculatively executed, Load-Exclusive.</td>
</tr>
<tr>
<td>0x006D</td>
<td>STREX_PASS_SPEC</td>
<td>Exclusive operation Speculatively executed, Store-Exclusive pass.</td>
</tr>
<tr>
<td>0x006E</td>
<td>STREX_FAIL_SPEC</td>
<td>Exclusive operation Speculatively executed, Store-Exclusive fail.</td>
</tr>
<tr>
<td>0x006F</td>
<td>STREX_SPEC</td>
<td>Exclusive operation Speculatively executed, Store-Exclusive.</td>
</tr>
<tr>
<td>0x0070</td>
<td>LD_SPEC</td>
<td>Operation speculatively executed, load.</td>
</tr>
<tr>
<td>0x0071</td>
<td>ST_SPEC</td>
<td>Operation speculatively executed, store.</td>
</tr>
<tr>
<td>0x0072</td>
<td>LDST_SPEC</td>
<td>Operation speculatively executed, load or store.</td>
</tr>
<tr>
<td>0x0073</td>
<td>DP_SPEC</td>
<td>Operation speculatively executed, integer data processing.</td>
</tr>
<tr>
<td>0x0074</td>
<td>ASE_SPEC</td>
<td>Operation speculatively executed, Advanced SIMD.</td>
</tr>
<tr>
<td>0x0075</td>
<td>VFP_SPEC</td>
<td>Operation speculatively executed, scalar floating-point.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x0076</td>
<td>PC_WRITE_SPEC</td>
<td>Operation speculatively executed, <strong>Software change of the PC.</strong></td>
</tr>
<tr>
<td>0x0077</td>
<td>CRYPTO_SPEC</td>
<td>Operation speculatively executed, <strong>Cryptographic instruction.</strong></td>
</tr>
<tr>
<td>0x0078</td>
<td>BR_IMMED_SPEC</td>
<td><strong>Branch Speculatively executed</strong>, immediate branch.</td>
</tr>
<tr>
<td>0x0079</td>
<td>BR_RETURN_SPEC</td>
<td><strong>Branch Speculatively executed</strong>, procedure return.</td>
</tr>
<tr>
<td>0x007A</td>
<td>BR_INDIRECT_SPEC</td>
<td><strong>Branch Speculatively executed</strong>, indirect branch.</td>
</tr>
<tr>
<td>0x007C</td>
<td>ISB_SPEC</td>
<td>Barrier <strong>Speculatively executed</strong>, ISB.</td>
</tr>
<tr>
<td>0x007D</td>
<td>DSB_SPEC</td>
<td>Barrier <strong>Speculatively executed</strong>, DSB.</td>
</tr>
<tr>
<td>0x007E</td>
<td>DMB_SPEC</td>
<td>Barrier <strong>Speculatively executed</strong>, DMB.</td>
</tr>
<tr>
<td>0x007F</td>
<td>CSDB_SPEC</td>
<td>Barrier <strong>Speculatively executed</strong>, CSDB.</td>
</tr>
<tr>
<td>0x0081</td>
<td>EXC_UNDEF</td>
<td>Exception taken, other synchronous.</td>
</tr>
<tr>
<td>0x0082</td>
<td>EXC_SVC</td>
<td>Exception taken, Supervisor Call.</td>
</tr>
<tr>
<td>0x0083</td>
<td>EXC_PABORT</td>
<td>Exception taken, Instruction Abort.</td>
</tr>
<tr>
<td>0x0084</td>
<td>EXC_DABORT</td>
<td>Exception taken, Data Abort or SError.</td>
</tr>
<tr>
<td>0x0086</td>
<td>EXC_IRQ</td>
<td>Exception taken, IRQ.</td>
</tr>
<tr>
<td>0x0087</td>
<td>EXC_FIQ</td>
<td>Exception taken, FIQ.</td>
</tr>
<tr>
<td>0x0088</td>
<td>EXC_SMC</td>
<td>Exception taken, Secure Monitor Call.</td>
</tr>
<tr>
<td>0x008A</td>
<td>EXC_HVC</td>
<td>Exception taken, Hypervisor Call.</td>
</tr>
<tr>
<td>0x008B</td>
<td>EXC_TRAP_PABORT</td>
<td>Exception taken, Instruction Abort not Taken locally.</td>
</tr>
<tr>
<td>0x008C</td>
<td>EXC_TRAP_DABORT</td>
<td>Exception taken, Data Abort or SError not Taken locally.</td>
</tr>
<tr>
<td>0x008D</td>
<td>EXC_TRAP_OTHER</td>
<td>Exception taken, other traps not Taken locally.</td>
</tr>
<tr>
<td>0x008E</td>
<td>EXC_TRAP_IRQ</td>
<td>Exception taken, IRQ not Taken locally.</td>
</tr>
<tr>
<td>0x008F</td>
<td>EXC_TRAP_FIQ</td>
<td>Exception taken, FIQ not Taken locally.</td>
</tr>
<tr>
<td>0x0090</td>
<td>RC_LD_SPEC</td>
<td>Release consistency operation <strong>Speculatively executed</strong>, Load-Acquire.</td>
</tr>
<tr>
<td>0x0091</td>
<td>RC_ST_SPEC</td>
<td>Release consistency operation <strong>Speculatively executed</strong>, Store-Release.</td>
</tr>
<tr>
<td>0x00A0</td>
<td>L3D_CACHE_RD</td>
<td>Level 3 data cache access, read.</td>
</tr>
<tr>
<td>0x00A1</td>
<td>L3D_CACHE_WR</td>
<td>Level 3 data cache access, write.</td>
</tr>
<tr>
<td>0x00A2</td>
<td>L3D_CACHE_REFILL_RD</td>
<td>Level 3 data cache refill, read.</td>
</tr>
<tr>
<td>0x00A3</td>
<td>L3D_CACHE_REFILL_WR</td>
<td>Level 3 data cache refill, write.</td>
</tr>
<tr>
<td>0x00A6</td>
<td>L3D_CACHE WB_VICTIM</td>
<td>Level 3 data cache write-back, victim.</td>
</tr>
</tbody>
</table>
Table D11-8 Event index (continued)

<table>
<thead>
<tr>
<th>Event number</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00A7</td>
<td>L3D_CACHE_WB_CLEAN</td>
<td>Level 3 data cache write-back, cleaning and coherency.</td>
</tr>
<tr>
<td>0x00A8</td>
<td>L3D_CACHE_INVAL</td>
<td>Level 3 data cache invalidate.</td>
</tr>
<tr>
<td>0x4000</td>
<td>SAMPLE_POP</td>
<td>Statistical Profiling sample population.</td>
</tr>
<tr>
<td>0x4001</td>
<td>SAMPLE_FEED</td>
<td>Statistical Profiling sample taken.</td>
</tr>
<tr>
<td>0x4002</td>
<td>SAMPLE_FILTRATE</td>
<td>Statistical Profiling sample taken and not removed by filtering.</td>
</tr>
<tr>
<td>0x4003</td>
<td>SAMPLE_COLLISION</td>
<td>Statistical Profiling sample collided with previous sample.</td>
</tr>
<tr>
<td>0x4004</td>
<td>CNT_CYCLES</td>
<td>Constant frequency cycles.</td>
</tr>
<tr>
<td>0x4005</td>
<td>STALL_BACKEND_MEM</td>
<td>Memory stall cycles.</td>
</tr>
<tr>
<td>0x4006</td>
<td>L1I_CACHE_LMISS</td>
<td>Level 1 instruction cache long-latency miss.</td>
</tr>
<tr>
<td>0x4009</td>
<td>L2D_CACHE_LMISS_RD</td>
<td>Level 2 data cache long-latency read miss.</td>
</tr>
<tr>
<td>0x400A</td>
<td>L2I_CACHE_LMISS</td>
<td>Level 2 instruction cache long-latency miss.</td>
</tr>
<tr>
<td>0x400B</td>
<td>L3D_CACHE_LMISS_RD</td>
<td>Level 3 data cache long-latency read miss.</td>
</tr>
<tr>
<td>0x400C</td>
<td>TRB_WRAP</td>
<td>Trace buffer current write pointer wrapped.</td>
</tr>
<tr>
<td>0x400D</td>
<td>PMU_OVFS</td>
<td>PMU overflow, counters accessible to EL1 and EL0.</td>
</tr>
<tr>
<td>0x400E</td>
<td>TRB_TRIG</td>
<td>Trace buffer Trigger Event.</td>
</tr>
<tr>
<td>0x400F</td>
<td>PMU_HOVFS</td>
<td>PMU overflow, counters reserved for use by EL2.</td>
</tr>
<tr>
<td>0x4010</td>
<td>TRCEXTOUT0</td>
<td>Trace unit external output 0.</td>
</tr>
<tr>
<td>0x4011</td>
<td>TRCEXTOUT1</td>
<td>Trace unit external output 1.</td>
</tr>
<tr>
<td>0x4012</td>
<td>TRCEXTOUT2</td>
<td>Trace unit external output 2.</td>
</tr>
<tr>
<td>0x4013</td>
<td>TRCEXTOUT3</td>
<td>Trace unit external output 3.</td>
</tr>
<tr>
<td>0x4018</td>
<td>CTI_TRIGOUT4</td>
<td>Cross-trigger Interface output trigger 4.</td>
</tr>
<tr>
<td>0x4019</td>
<td>CTI_TRIGOUT5</td>
<td>Cross-trigger Interface output trigger 5.</td>
</tr>
<tr>
<td>0x401A</td>
<td>CTI_TRIGOUT6</td>
<td>Cross-trigger Interface output trigger 6.</td>
</tr>
<tr>
<td>0x401B</td>
<td>CTI_TRIGOUT7</td>
<td>Cross-trigger Interface output trigger 7.</td>
</tr>
<tr>
<td>0x4020</td>
<td>LDST_ALIGN_LAT</td>
<td>Access with additional latency from alignment.</td>
</tr>
<tr>
<td>0x4021</td>
<td>LD_ALIGN_LAT</td>
<td>Load with additional latency from alignment.</td>
</tr>
<tr>
<td>0x4022</td>
<td>ST_ALIGN_LAT</td>
<td>Store with additional latency from alignment.</td>
</tr>
<tr>
<td>0x4024</td>
<td>MEM_ACCESS_CHECKED</td>
<td>Checked data memory access.</td>
</tr>
<tr>
<td>0x4025</td>
<td>MEM_ACCESS_RD_CHECKED</td>
<td>Checked data memory access, read.</td>
</tr>
<tr>
<td>0x4026</td>
<td>MEM_ACCESS_WR_CHECKED</td>
<td>Checked data memory access, write.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x8000</td>
<td>SIMD_INST_RETIRED</td>
<td>Instruction architecturally executed, SIMD.</td>
</tr>
<tr>
<td>0x8001</td>
<td>ASE_INST_RETIRED</td>
<td>Instruction architecturally executed, Advanced SIMD.</td>
</tr>
<tr>
<td>0x8002</td>
<td>SVE_INST_RETIRED</td>
<td>Instruction architecturally executed, SVE.</td>
</tr>
<tr>
<td>0x8003</td>
<td>ASE_SVE_INST_RETIRED</td>
<td>Instruction architecturally executed, Advanced SIMD or SVE.</td>
</tr>
<tr>
<td>0x8004</td>
<td>SIMD_INST_SPEC</td>
<td>Operation speculatively executed, SIMD.</td>
</tr>
<tr>
<td>0x8005</td>
<td>ASE_INST_SPEC</td>
<td>Operation speculatively executed, Advanced SIMD.</td>
</tr>
<tr>
<td>0x8006</td>
<td>SVE_INST_SPEC</td>
<td>Operation speculatively executed, SVE, including load and store.</td>
</tr>
<tr>
<td>0x8007</td>
<td>ASE_SVE_INST_SPEC</td>
<td>Operation speculatively executed, Advanced SIMD or SVE.</td>
</tr>
<tr>
<td>0x8008</td>
<td>UOP_SPEC</td>
<td>Microarchitectural operation speculatively executed.</td>
</tr>
<tr>
<td>0x8009</td>
<td>ASE_UOP_SPEC</td>
<td>Microarchitectural operation speculatively executed, Advanced SIMD.</td>
</tr>
<tr>
<td>0x800A</td>
<td>SVE_UOP_SPEC</td>
<td>Microarchitectural operation speculatively executed, SVE.</td>
</tr>
<tr>
<td>0x800B</td>
<td>ASE_SVE_UOP_SPEC</td>
<td>Microarchitectural operation speculatively executed, Advanced SIMD or SVE.</td>
</tr>
<tr>
<td>0x800C</td>
<td>SIMD_UOP_SPEC</td>
<td>Microarchitectural operation speculatively executed, SIMD.</td>
</tr>
<tr>
<td>0x800E</td>
<td>SVE_MATH_SPEC</td>
<td>Operation speculatively executed, SVE math accelerator.</td>
</tr>
<tr>
<td>0x8010</td>
<td>FP_SPEC</td>
<td>Floating-point operation speculatively executed, including SIMD.</td>
</tr>
<tr>
<td>0x8011</td>
<td>ASE_FP_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD.</td>
</tr>
<tr>
<td>0x8012</td>
<td>SVE_FP_SPEC</td>
<td>Floating-point operation speculatively executed, SVE.</td>
</tr>
<tr>
<td>0x8013</td>
<td>ASE_SVE_FP_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD or SVE.</td>
</tr>
<tr>
<td>0x8014</td>
<td>FP_HP_SPEC</td>
<td>Floating-point operation speculatively executed, half precision.</td>
</tr>
<tr>
<td>0x8015</td>
<td>ASE_FP_HP_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD half precision.</td>
</tr>
<tr>
<td>0x8016</td>
<td>SVE_FP_HP_SPEC</td>
<td>Floating-point operation speculatively executed, SVE half precision.</td>
</tr>
<tr>
<td>0x8017</td>
<td>ASE_SVE_FP_HP_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD or SVE half precision.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>0x8018</td>
<td>FP_SP_SPEC</td>
<td>Floating-point operation speculatively executed, single precision.</td>
</tr>
<tr>
<td>0x8019</td>
<td>ASE_FP_SP_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD single precision.</td>
</tr>
<tr>
<td>0x801A</td>
<td>SVE_FP_SP_SPEC</td>
<td>Floating-point operation speculatively executed, SVE single precision.</td>
</tr>
<tr>
<td>0x801B</td>
<td>ASE_SVE_FP_SP_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD or SVE single precision.</td>
</tr>
<tr>
<td>0x801C</td>
<td>FP_DP_SPEC</td>
<td>Floating-point operation speculatively executed, double precision.</td>
</tr>
<tr>
<td>0x801D</td>
<td>ASE_FP_DP_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD double precision.</td>
</tr>
<tr>
<td>0x801E</td>
<td>SVE_FP_DP_SPEC</td>
<td>Floating-point operation speculatively executed, SVE double precision.</td>
</tr>
<tr>
<td>0x801F</td>
<td>ASE_SVE_FP_DP_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD or SVE double precision.</td>
</tr>
<tr>
<td>0x8020</td>
<td>FP_DIV_SPEC</td>
<td>Floating-point operation speculatively executed, divide.</td>
</tr>
<tr>
<td>0x8021</td>
<td>ASE_FP_DIV_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD divide.</td>
</tr>
<tr>
<td>0x8022</td>
<td>SVE_FP_DIV_SPEC</td>
<td>Floating-point operation speculatively executed, SVE divide.</td>
</tr>
<tr>
<td>0x8023</td>
<td>ASE_SVE_FP_DIV_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD or SVE divide.</td>
</tr>
<tr>
<td>0x8024</td>
<td>FP_SQRT_SPEC</td>
<td>Floating-point operation speculatively executed, square root.</td>
</tr>
<tr>
<td>0x8025</td>
<td>ASE_FP_SQRT_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD square root.</td>
</tr>
<tr>
<td>0x8026</td>
<td>SVE_FP_SQRT_SPEC</td>
<td>Floating-point operation speculatively executed, SVE square root.</td>
</tr>
<tr>
<td>0x8027</td>
<td>ASE_SVE_FP_SQRT_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD or SVE square-root.</td>
</tr>
<tr>
<td>0x8028</td>
<td>FP_FMA_SPEC</td>
<td>Floating-point operation speculatively executed, FMA.</td>
</tr>
<tr>
<td>0x8029</td>
<td>ASE_FP_FMA_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD FMA.</td>
</tr>
<tr>
<td>0x802A</td>
<td>SVE_FP_FMA_SPEC</td>
<td>Floating-point operation speculatively executed, SVE FMA.</td>
</tr>
<tr>
<td>0x802B</td>
<td>ASE_SVE_FP_FMA_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD or SVE FMA.</td>
</tr>
<tr>
<td>0x802C</td>
<td>FP_MUL_SPEC</td>
<td>Floating-point operation speculatively executed, multiply.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>0x802D</td>
<td>ASE_FP_MUL_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD multiply.</td>
</tr>
<tr>
<td>0x802E</td>
<td>SVE_FP_MUL_SPEC</td>
<td>Floating-point operation speculatively executed, SVE multiply.</td>
</tr>
<tr>
<td>0x802F</td>
<td>ASE_SVE_FP_MUL_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD or SVE multiply.</td>
</tr>
<tr>
<td>0x8030</td>
<td>FP_ADDSUB_SPEC</td>
<td>Floating-point operation speculatively executed, add or subtract.</td>
</tr>
<tr>
<td>0x8031</td>
<td>ASE_FP_ADDSUB_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD add or subtract.</td>
</tr>
<tr>
<td>0x8032</td>
<td>SVE_FP_ADDSUB_SPEC</td>
<td>Floating-point operation speculatively executed, SVE add or subtract.</td>
</tr>
<tr>
<td>0x8033</td>
<td>ASE_SVE_FP_ADDSUB_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD or SVE add or subtract.</td>
</tr>
<tr>
<td>0x8034</td>
<td>FP_RECPE_SPEC</td>
<td>Floating-point operation speculatively executed, reciprocal estimate.</td>
</tr>
<tr>
<td>0x8035</td>
<td>ASE_FP_RECPE_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD reciprocal estimate.</td>
</tr>
<tr>
<td>0x8036</td>
<td>SVE_FP_RECPE_SPEC</td>
<td>Floating-point operation speculatively executed, SVE reciprocal estimate.</td>
</tr>
<tr>
<td>0x8037</td>
<td>ASE_SVE_FP_RECPE_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD or SVE reciprocal estimate.</td>
</tr>
<tr>
<td>0x8038</td>
<td>FP_CVT_SPEC</td>
<td>Floating-point operation speculatively executed, convert.</td>
</tr>
<tr>
<td>0x8039</td>
<td>ASE_FP_CVT_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD convert.</td>
</tr>
<tr>
<td>0x803A</td>
<td>SVE_FP_CVT_SPEC</td>
<td>Floating-point operation speculatively executed, SVE convert.</td>
</tr>
<tr>
<td>0x803B</td>
<td>ASE_SVE_FP_CVT_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD or SVE convert.</td>
</tr>
<tr>
<td>0x803C</td>
<td>SVE_FP_AREDUCE_SPEC</td>
<td>Floating-point operation speculatively executed, SVE accumulating reduction.</td>
</tr>
<tr>
<td>0x803D</td>
<td>ASE_FP_PREDUCE_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD pairwise add step.</td>
</tr>
<tr>
<td>0x803E</td>
<td>SVE_FP_VREDUCE_SPEC</td>
<td>Floating-point operation speculatively executed, SVE vector reduction.</td>
</tr>
<tr>
<td>0x803F</td>
<td>ASE_SVE_FP_VREDUCE_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD or SVE vector reduction.</td>
</tr>
<tr>
<td>0x8040</td>
<td>INT_SPEC</td>
<td>Integer operation speculatively executed.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>0x8041</td>
<td>ASE_INT_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD.</td>
</tr>
<tr>
<td>0x8042</td>
<td>SVE_INT_SPEC</td>
<td>Integer operation speculatively executed, SVE.</td>
</tr>
<tr>
<td>0x8043</td>
<td>ASE_SVE_INT_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD or SVE.</td>
</tr>
<tr>
<td>0x8044</td>
<td>INT_DIV_SPEC</td>
<td>Integer operation speculatively executed, divide.</td>
</tr>
<tr>
<td>0x8045</td>
<td>INT_DIV64_SPEC</td>
<td>Integer operation speculatively executed, 64-bit divide.</td>
</tr>
<tr>
<td>0x8046</td>
<td>SVE_INT_DIV_SPEC</td>
<td>Integer operation speculatively executed, SVE divide.</td>
</tr>
<tr>
<td>0x8047</td>
<td>SVE_INT_DIV64_SPEC</td>
<td>Integer operation speculatively executed, SVE 64-bit divide.</td>
</tr>
<tr>
<td>0x8048</td>
<td>INT_MUL_SPEC</td>
<td>Integer operation speculatively executed, multiply.</td>
</tr>
<tr>
<td>0x8049</td>
<td>ASE_INT_MUL_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD multiply.</td>
</tr>
<tr>
<td>0x804A</td>
<td>SVE_INT_MUL_SPEC</td>
<td>Integer operation speculatively executed, SVE multiply.</td>
</tr>
<tr>
<td>0x804B</td>
<td>ASE_SVE_INT_MUL_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD or SVE multiply.</td>
</tr>
<tr>
<td>0x804C</td>
<td>INT_MUL64_SPEC</td>
<td>Integer operation speculatively executed, 64×64 multiply.</td>
</tr>
<tr>
<td>0x804D</td>
<td>SVE_INT_MUL64_SPEC</td>
<td>Integer operation speculatively executed, SVE 64×64 multiply.</td>
</tr>
<tr>
<td>0x804E</td>
<td>INT_MULH64_SPEC</td>
<td>Integer operation speculatively executed, 64×64 multiply returning high part.</td>
</tr>
<tr>
<td>0x804F</td>
<td>SVE_INT_MULH64_SPEC</td>
<td>Integer operation speculatively executed, SVE 64×64 multiply high part.</td>
</tr>
<tr>
<td>0x8058</td>
<td>NONFP_SPEC</td>
<td>Non-floating-point operation speculatively executed.</td>
</tr>
<tr>
<td>0x8059</td>
<td>ASE_NONFP_SPEC</td>
<td>Non-floating-point operation speculatively executed, Advanced SIMD.</td>
</tr>
<tr>
<td>0x805A</td>
<td>SVE_NONFP_SPEC</td>
<td>Non-floating-point operation speculatively executed, SVE.</td>
</tr>
<tr>
<td>0x805B</td>
<td>ASE_SVE_NONFP_SPEC</td>
<td>Non-floating-point operation speculatively executed, Advanced SIMD or SVE.</td>
</tr>
<tr>
<td>0x805D</td>
<td>ASE_INT_VREDUCE_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD reduction.</td>
</tr>
<tr>
<td>0x805E</td>
<td>SVE_INT_VREDUCE_SPEC</td>
<td>Integer operation speculatively executed, SVE reduction.</td>
</tr>
<tr>
<td>0x805F</td>
<td>ASE_SVE_INT_VREDUCE_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD or SVE reduction.</td>
</tr>
<tr>
<td>0x8060</td>
<td>SVE_PERM_SPEC</td>
<td>Operation speculatively executed, SVE permute.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x8061</td>
<td>SVE_PERM_IGRANULE_SPEC</td>
<td>Operation speculatively executed, SVE intra-granule permute.</td>
</tr>
<tr>
<td>0x8062</td>
<td>SVE_PERM_XGRANULE_SPEC</td>
<td>Operation speculatively executed, SVE cross-granule permute.</td>
</tr>
<tr>
<td>0x8063</td>
<td>SVE_PERM_VARIABLE_SPEC</td>
<td>Operation speculatively executed, SVE programmable permute.</td>
</tr>
<tr>
<td>0x8064</td>
<td>SVE_XPIPE_SPEC</td>
<td>Operation speculatively executed, SVE cross-pipe.</td>
</tr>
<tr>
<td>0x8065</td>
<td>SVE_XPIPE_Z2R_SPEC</td>
<td>Operation speculatively executed, SVE vector to scalar cross-pipe.</td>
</tr>
<tr>
<td>0x8066</td>
<td>SVE_XPIPE_R2Z_SPEC</td>
<td>Operation speculatively executed, SVE scalar to vector cross-pipe.</td>
</tr>
<tr>
<td>0x8067</td>
<td>SVE_PGEN_NVEC_SPEC</td>
<td>Operation speculatively executed, SVE predicate-only.</td>
</tr>
<tr>
<td>0x8068</td>
<td>SVE_PGEN_SPEC</td>
<td>Operation speculatively executed, SVE predicate generating.</td>
</tr>
<tr>
<td>0x8069</td>
<td>SVE_PGEN_FLG_SPEC</td>
<td>Operation speculatively executed, SVE predicate flag setting.</td>
</tr>
<tr>
<td>0x806A</td>
<td>SVE_PGEN_CMP_SPEC</td>
<td>Operation speculatively executed, SVE vector compare.</td>
</tr>
<tr>
<td>0x806B</td>
<td>SVE_PGEN_FCM_SPEC</td>
<td>Floating-point operation speculatively executed, SVE vector compare.</td>
</tr>
<tr>
<td>0x806C</td>
<td>SVE_PGEN_LOGIC_SPEC</td>
<td>Operation speculatively executed, SVE predicate logical.</td>
</tr>
<tr>
<td>0x806D</td>
<td>SVE_PPERM_SPEC</td>
<td>Operation speculatively executed, SVE predicate permute.</td>
</tr>
<tr>
<td>0x806E</td>
<td>SVE_PSCAN_SPEC</td>
<td>Operation speculatively executed, SVE predicate scan.</td>
</tr>
<tr>
<td>0x806F</td>
<td>SVE_PCNT_SPEC</td>
<td>Operation speculatively executed, SVE predicate count.</td>
</tr>
<tr>
<td>0x8070</td>
<td>SVE_PLOOP_WHILE_SPEC</td>
<td>Operation speculatively executed, SVE predicate loop while.</td>
</tr>
<tr>
<td>0x8071</td>
<td>SVE_PLOOP_TEST_SPEC</td>
<td>Operation speculatively executed, SVE predicate loop test.</td>
</tr>
<tr>
<td>0x8072</td>
<td>SVE_PLOOP_ELTS_SPEC</td>
<td>Operation speculatively executed, SVE predicate loop elements.</td>
</tr>
<tr>
<td>0x8073</td>
<td>SVE_PLOOP_TERM_SPEC</td>
<td>Operation speculatively executed, SVE predicate loop termination.</td>
</tr>
<tr>
<td>0x8074</td>
<td>SVE_PRED_SPEC</td>
<td>Operation speculatively executed, SVE predicated.</td>
</tr>
<tr>
<td>0x8075</td>
<td>SVE_PRED_EMPTY_SPEC</td>
<td>Operation speculatively executed, SVE predicated with no active predicates.</td>
</tr>
<tr>
<td>0x8076</td>
<td>SVE_PRED_FULL_SPEC</td>
<td>Operation speculatively executed, SVE predicated with all active predicates.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x8077</td>
<td>SVE_PRED_PARTIAL_SPEC</td>
<td>Operation speculatively executed, SVE predicated with partially active predicates.</td>
</tr>
<tr>
<td>0x8078</td>
<td>SVE_UNPRED_SPEC</td>
<td>Operation speculatively executed, SVE unpredicated.</td>
</tr>
<tr>
<td>0x8079</td>
<td>SVE_PRED_NOT_FULL_SPEC</td>
<td>SVE predicated operations Speculatively executed with no active or partially active predicates.</td>
</tr>
<tr>
<td>0x807C</td>
<td>SVE_MOVPRFX_SPEC</td>
<td>Operation speculatively executed, SVE MOVPRFX.</td>
</tr>
<tr>
<td>0x807D</td>
<td>SVE_MOVPRFX_Z_SPEC</td>
<td>Operation speculatively executed, SVE MOVPRFX zeroing predication.</td>
</tr>
<tr>
<td>0x807E</td>
<td>SVE_MOVPRFX_M_SPEC</td>
<td>Operation speculatively executed, SVE MOVPRFX merging predication.</td>
</tr>
<tr>
<td>0x807F</td>
<td>SVE_MOVPRFX_U_SPEC</td>
<td>Operation speculatively executed, SVE MOVPRFX unfused.</td>
</tr>
<tr>
<td>0x8080</td>
<td>SVE_LDST_SPEC</td>
<td>Operation speculatively executed, SVE load, store, or prefetch.</td>
</tr>
<tr>
<td>0x8081</td>
<td>SVE_LD_SPEC</td>
<td>Operation speculatively executed, SVE load.</td>
</tr>
<tr>
<td>0x8082</td>
<td>SVE_ST_SPEC</td>
<td>Operation speculatively executed, SVE store.</td>
</tr>
<tr>
<td>0x8083</td>
<td>SVE_PRF_SPEC</td>
<td>Operation speculatively executed, SVE prefetch.</td>
</tr>
<tr>
<td>0x8084</td>
<td>ASE_SVE_LDST_SPEC</td>
<td>Operation speculatively executed, Advanced SIMD or SVE load or store.</td>
</tr>
<tr>
<td>0x8085</td>
<td>ASE_SVE_LD_SPEC</td>
<td>Operation speculatively executed, Advanced SIMD or SVE load.</td>
</tr>
<tr>
<td>0x8086</td>
<td>ASE_SVE_ST_SPEC</td>
<td>Operation speculatively executed, Advanced SIMD or SVE store.</td>
</tr>
<tr>
<td>0x8087</td>
<td>PRF_SPEC</td>
<td>Operation speculatively executed, Prefetch.</td>
</tr>
<tr>
<td>0x8088</td>
<td>BASE_LDST_REG_SPEC</td>
<td>Operation speculatively executed, general-purpose register load, store, or prefetch.</td>
</tr>
<tr>
<td>0x8089</td>
<td>BASE_LD_REG_SPEC</td>
<td>Operation speculatively executed, general-purpose register load.</td>
</tr>
<tr>
<td>0x808A</td>
<td>BASE_ST_REG_SPEC</td>
<td>Operation speculatively executed, general-purpose register store.</td>
</tr>
<tr>
<td>0x808B</td>
<td>BASE_PRF_SPEC</td>
<td>Operation speculatively executed, general-purpose register prefetch.</td>
</tr>
<tr>
<td>0x808C</td>
<td>FPASE_LDST_REG_SPEC</td>
<td>Operation speculatively executed, SIMD&amp;FP register load or store.</td>
</tr>
<tr>
<td>0x808D</td>
<td>FPASE_LD_REG_SPEC</td>
<td>Operation speculatively executed, SIMD&amp;FP register load.</td>
</tr>
<tr>
<td>0x808E</td>
<td>FPASE_ST_REG_SPEC</td>
<td>Operation speculatively executed, SIMD&amp;FP register store.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x8090</td>
<td>SVE_LDSR_REG_SPEC</td>
<td>Operation speculatively executed, SVE unpredicated load or store register.</td>
</tr>
<tr>
<td>0x8091</td>
<td>SVE_LDR_REG_SPEC</td>
<td>Operation speculatively executed, SVE unpredicated load register.</td>
</tr>
<tr>
<td>0x8092</td>
<td>SVE_STR_REG_SPEC</td>
<td>Operation speculatively executed, SVE unpredicated store register.</td>
</tr>
<tr>
<td>0x8094</td>
<td>SVE_LDSR_PREG_SPEC</td>
<td>Operation speculatively executed, SVE load or store predicate register.</td>
</tr>
<tr>
<td>0x8095</td>
<td>SVE_LDR_PREG_SPEC</td>
<td>Operation speculatively executed, SVE load predicate register.</td>
</tr>
<tr>
<td>0x8096</td>
<td>SVE_STR_PREG_SPEC</td>
<td>Operation speculatively executed, SVE store predicate register.</td>
</tr>
<tr>
<td>0x8098</td>
<td>SVE_LDSR_ZREG_SPEC</td>
<td>Operation speculatively executed, SVE load or store vector register.</td>
</tr>
<tr>
<td>0x8099</td>
<td>SVE_LDR_ZREG_SPEC</td>
<td>Operation speculatively executed, SVE load vector register.</td>
</tr>
<tr>
<td>0x809A</td>
<td>SVE_STR_ZREG_SPEC</td>
<td>Operation speculatively executed, SVE store vector register.</td>
</tr>
<tr>
<td>0x809C</td>
<td>SVE_LDSR_CONTIG_SPEC</td>
<td>Operation speculatively executed, SVE contiguous load, store, or prefetch element.</td>
</tr>
<tr>
<td>0x809D</td>
<td>SVE_LDR_CONTIG_SPEC</td>
<td>Operation speculatively executed, SVE contiguous load element.</td>
</tr>
<tr>
<td>0x809E</td>
<td>SVE_ST_CONTIG_SPEC</td>
<td>Operation speculatively executed, SVE contiguous store element.</td>
</tr>
<tr>
<td>0x809F</td>
<td>SVE_PRF_CONTIG_SPEC</td>
<td>Operation speculatively executed, SVE contiguous prefetch element.</td>
</tr>
<tr>
<td>0x80A0</td>
<td>SVE_LDSRNT_CONTIG_SPEC</td>
<td>Operation speculatively executed, SVE non-temporal contiguous load or store element.</td>
</tr>
<tr>
<td>0x80A1</td>
<td>SVE_LDRNT_CONTIG_SPEC</td>
<td>Operation speculatively executed, SVE non-temporal contiguous load element.</td>
</tr>
<tr>
<td>0x80A2</td>
<td>SVE_STNT_CONTIG_SPEC</td>
<td>Operation speculatively executed, SVE non-temporal contiguous store element.</td>
</tr>
<tr>
<td>0x80A4</td>
<td>ASE_SVE_LDSR_MULTI_SPEC</td>
<td>Operation speculatively executed, Advanced SIMD or SVE contiguous load or store multiple vector.</td>
</tr>
<tr>
<td>0x80A5</td>
<td>ASE_SVE_LDR_MULTI_SPEC</td>
<td>Operation speculatively executed, Advanced SIMD or SVE contiguous load multiple vector.</td>
</tr>
<tr>
<td>0x80A6</td>
<td>ASE_SVE_ST_MULTI_SPEC</td>
<td>Operation speculatively executed, Advanced SIMD or SVE contiguous store multiple vector.</td>
</tr>
<tr>
<td>0x80A8</td>
<td>SVE_LDSR_MULTI_SPEC</td>
<td>Operation speculatively executed, SVE contiguous load or store multiple vector.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x80A9</td>
<td>SVE_LD_MULTI_SPEC</td>
<td>Operation speculatively executed, SVE contiguous load multiple vector.</td>
</tr>
<tr>
<td>0x80AA</td>
<td>SVE_ST_MULTI_SPEC</td>
<td>Operation speculatively executed, SVE contiguous store multiple vector.</td>
</tr>
<tr>
<td>0x80AC</td>
<td>SVE_LDST_NONCONTIG_SPEC</td>
<td>Operation speculatively executed, SVE non-contiguous load, store, or prefetch.</td>
</tr>
<tr>
<td>0x80AD</td>
<td>SVE_LD_GATHER_SPEC</td>
<td>Operation speculatively executed, SVE gather-load.</td>
</tr>
<tr>
<td>0x80AE</td>
<td>SVE_ST_SCATTER_SPEC</td>
<td>Operation speculatively executed, SVE scatter-store.</td>
</tr>
<tr>
<td>0x80AF</td>
<td>SVE_PRF_GATHER_SPEC</td>
<td>Operation speculatively executed, SVE gather-prefetch.</td>
</tr>
<tr>
<td>0x80B0</td>
<td>SVE_LDST64_NONCONTIG_SPEC</td>
<td>Operation speculatively executed, SVE 64-bit non-contiguous load, store, or prefetch.</td>
</tr>
<tr>
<td>0x80B1</td>
<td>SVE_LD64_GATHER_SPEC</td>
<td>Operation speculatively executed, SVE 64-bit gather-load.</td>
</tr>
<tr>
<td>0x80B2</td>
<td>SVE_ST64_SCATTER_SPEC</td>
<td>Operation speculatively executed, SVE 64-bit scatter-store.</td>
</tr>
<tr>
<td>0x80B3</td>
<td>SVE_PRF64_GATHER_SPEC</td>
<td>Operation speculatively executed, SVE 64-bit gather-prefetch.</td>
</tr>
<tr>
<td>0x80B4</td>
<td>ASE_SVE_UNALIGNED_LDS64_SPEC</td>
<td>Advanced SIMD or SVE unaligned accesses.</td>
</tr>
<tr>
<td>0x80B5</td>
<td>ASE_SVE_UNALIGNED_LDS64_SPEC</td>
<td>Advanced SIMD or SVE unaligned read accesses.</td>
</tr>
<tr>
<td>0x80B6</td>
<td>ASE_SVE_UNALIGNED_STS64_SPEC</td>
<td>Advanced SIMD or SVE unaligned write accesses.</td>
</tr>
<tr>
<td>0x80B8</td>
<td>ASE_SVE_UNALIGNED_CONTIG_LDS64_SPEC</td>
<td>Advanced SIMD or SVE unaligned contiguous accesses.</td>
</tr>
<tr>
<td>0x80B9</td>
<td>ASE_SVE_UNALIGNED_CONTIG_LDS64_SPEC</td>
<td>Advanced SIMD or SVE unaligned contiguous read accesses.</td>
</tr>
<tr>
<td>0x80BA</td>
<td>ASE_SVE_UNALIGNED_CONTIG_STS64_SPEC</td>
<td>Advanced SIMD or SVE unaligned contiguous write accesses.</td>
</tr>
<tr>
<td>0x80BC</td>
<td>SVE_LDFF_SPEC</td>
<td>Operation speculatively executed, SVE first-fault load.</td>
</tr>
<tr>
<td>0x80BD</td>
<td>SVE_LDFF_FAULT_SPEC</td>
<td>Operation speculatively executed, SVE first-fault load which set FFR bit to 0b0.</td>
</tr>
<tr>
<td>0x80C0</td>
<td>FP_SCALE_OPS_SPEC</td>
<td>Scalable floating-point element ALU operations Speculatively executed.</td>
</tr>
<tr>
<td>0x80C1</td>
<td>FP_FIXED_OPS_SPEC</td>
<td>Non-scalable floating-point element ALU operations Speculatively executed.</td>
</tr>
<tr>
<td>0x80C2</td>
<td>FP_HP_SCALE_OPS_SPEC</td>
<td>Scalable half-precision floating-point element ALU operations Speculatively executed.</td>
</tr>
<tr>
<td>0x80C3</td>
<td>FP_HP_FIXED_OPS_SPEC</td>
<td>Non-scalable half-precision floating-point element ALU operations Speculatively executed.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x80C4</td>
<td>FP_SP_SCALE_OPS_SPEC</td>
<td>Scalable single-precision floating-point element ALU operations Speculatively executed.</td>
</tr>
<tr>
<td>0x80C5</td>
<td>FP_SP_FIXED_OPS_SPEC</td>
<td>Non-scalable single-precision floating-point element ALU operations Speculatively executed.</td>
</tr>
<tr>
<td>0x80C6</td>
<td>FP_DP_SCALE_OPS_SPEC</td>
<td>Scalable double-precision floating-point element ALU operations Speculatively executed.</td>
</tr>
<tr>
<td>0x80C7</td>
<td>FP_DP_FIXED_OPS_SPEC</td>
<td>Non-scalable double-precision floating-point element ALU operations Speculatively executed.</td>
</tr>
<tr>
<td>0x80C8</td>
<td>INT_SCALE_OPS_SPEC</td>
<td>Scalable integer element ALU operations Speculatively executed.</td>
</tr>
<tr>
<td>0x80C9</td>
<td>INT_FIXED_OPS_SPEC</td>
<td>Non-scalable integer element ALU operations Speculatively executed.</td>
</tr>
<tr>
<td>0x80CA</td>
<td>LDST_SCALE_OPS_SPEC</td>
<td>Scalable load or store element Operations speculatively executed.</td>
</tr>
<tr>
<td>0x80CB</td>
<td>LDST_FIXED_OPS_SPEC</td>
<td>Non-scalable load or store element Operations speculatively executed.</td>
</tr>
<tr>
<td>0x80CC</td>
<td>LD_SCALE_OPS_SPEC</td>
<td>Scalable load element Operations speculatively executed.</td>
</tr>
<tr>
<td>0x80CD</td>
<td>LD_FIXED_OPS_SPEC</td>
<td>Non-scalable load element Operations speculatively executed.</td>
</tr>
<tr>
<td>0x80CE</td>
<td>ST_SCALE_OPS_SPEC</td>
<td>Scalable store element Operations speculatively executed.</td>
</tr>
<tr>
<td>0x80CF</td>
<td>ST_FIXED_OPS_SPEC</td>
<td>Non-scalable store element Operations speculatively executed.</td>
</tr>
<tr>
<td>0x80DA</td>
<td>LDST_SCALE_BYTES_SPEC</td>
<td>Scalable load and store bytes Speculatively executed.</td>
</tr>
<tr>
<td>0x80DB</td>
<td>LDST_FIXED_BYTES_SPEC</td>
<td>Non-scalable load and store bytes Speculatively executed.</td>
</tr>
<tr>
<td>0x80DC</td>
<td>LD_SCALE_BYTES_SPEC</td>
<td>Scalable load bytes Speculatively executed.</td>
</tr>
<tr>
<td>0x80DD</td>
<td>LD_FIXED_BYTES_SPEC</td>
<td>Non-scalable load bytes Speculatively executed.</td>
</tr>
<tr>
<td>0x80DE</td>
<td>ST_SCALE_BYTES_SPEC</td>
<td>Scalable store bytes Speculatively executed.</td>
</tr>
<tr>
<td>0x80DF</td>
<td>ST_FIXED_BYTES_SPEC</td>
<td>Non-scalable store bytes Speculatively executed.</td>
</tr>
<tr>
<td>0x80E1</td>
<td>ASE_INT8_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD 8-bit.</td>
</tr>
<tr>
<td>0x80E2</td>
<td>SVE_INT8_SPEC</td>
<td>Integer operation speculatively executed, SVE 8-bit.</td>
</tr>
<tr>
<td>0x80E3</td>
<td>ASE_SVE_INT8_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD or SVE 8-bit.</td>
</tr>
<tr>
<td>0x80E5</td>
<td>ASE_INT16_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD 16-bit.</td>
</tr>
</tbody>
</table>
### Table D11-8 Event index (continued)

<table>
<thead>
<tr>
<th>Event number</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80E6</td>
<td>SVE_INT16_SPEC</td>
<td>Integer operation speculatively executed, SVE 16-bit.</td>
</tr>
<tr>
<td>0x80E7</td>
<td>ASE_SVE_INT16_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD or SVE 16-bit.</td>
</tr>
<tr>
<td>0x80E9</td>
<td>ASE_INT32_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD 32-bit.</td>
</tr>
<tr>
<td>0x80EA</td>
<td>SVE_INT32_SPEC</td>
<td>Integer operation speculatively executed, SVE 32-bit.</td>
</tr>
<tr>
<td>0x80EB</td>
<td>ASE_SVE_INT32_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD or SVE 32-bit.</td>
</tr>
<tr>
<td>0x80ED</td>
<td>ASE_INT64_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD 64-bit.</td>
</tr>
<tr>
<td>0x80EE</td>
<td>SVE_INT64_SPEC</td>
<td>Integer operation speculatively executed, SVE 64-bit.</td>
</tr>
<tr>
<td>0x80EF</td>
<td>ASE_SVE_INT64_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD or SVE 64-bit.</td>
</tr>
<tr>
<td>0x80F1</td>
<td>ASE_FP_DOT_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD dot-product.</td>
</tr>
<tr>
<td>0x80F2</td>
<td>SVE_FP_DOT_SPEC</td>
<td>Floating-point operation speculatively executed, SVE dot-product.</td>
</tr>
<tr>
<td>0x80F3</td>
<td>ASE_SVE_FP_DOT_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD or SVE dot-product.</td>
</tr>
<tr>
<td>0x80F5</td>
<td>ASE_FP_MMLA_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD matrix multiply.</td>
</tr>
<tr>
<td>0x80F6</td>
<td>SVE_FP_MMLA_SPEC</td>
<td>Floating-point operation speculatively executed, SVE matrix multiply.</td>
</tr>
<tr>
<td>0x80F7</td>
<td>ASE_SVE_FP_MMLA_SPEC</td>
<td>Floating-point operation speculatively executed, Advanced SIMD or SVE matrix multiply.</td>
</tr>
<tr>
<td>0x80F9</td>
<td>ASE_INT_DOT_SPEC</td>
<td>Operation speculatively executed, Advanced SIMD integer dot-product.</td>
</tr>
<tr>
<td>0x80FA</td>
<td>SVE_INT_DOT_SPEC</td>
<td>Integer operation speculatively executed, SVE dot-product.</td>
</tr>
<tr>
<td>0x80FB</td>
<td>ASE_SVE_INT_DOT_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD or SVE dot-product.</td>
</tr>
<tr>
<td>0x80FD</td>
<td>ASE_INT_MMLA_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD matrix multiply.</td>
</tr>
<tr>
<td>0x80FE</td>
<td>SVE_INT_MMLA_SPEC</td>
<td>Integer operation speculatively executed, SVE matrix multiply.</td>
</tr>
<tr>
<td>0x80FF</td>
<td>ASE_SVE_INT_MMLA_SPEC</td>
<td>Integer operation speculatively executed, Advanced SIMD or SVE matrix multiply.</td>
</tr>
<tr>
<td>0x8107</td>
<td>BR_SKIP_RETIRED</td>
<td>Branch Instruction architecturally executed, not taken.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------</td>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td>0x8108</td>
<td>BR_IMMED_TAKEN_RETIRED</td>
<td>Branch Instruction architecturally executed, immediate,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>taken.</td>
</tr>
<tr>
<td>0x8109</td>
<td>BR_IMMED_SKIP_RETIRED</td>
<td>Branch Instruction architecturally executed, immediate,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>not taken.</td>
</tr>
<tr>
<td>0x810A</td>
<td>BR_IND_TAKEN_RETIRED</td>
<td>Branch Instruction architecturally executed, indirect,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>taken.</td>
</tr>
<tr>
<td>0x810B</td>
<td>BR_IND_SKIP_RETIRED</td>
<td>Branch Instruction architecturally executed, indirect,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>not taken.</td>
</tr>
<tr>
<td>0x810C</td>
<td>BR_INDNR_TAKEN_RETIRED</td>
<td>Branch Instruction architecturally executed, indirect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>excluding procedure return, taken.</td>
</tr>
<tr>
<td>0x810D</td>
<td>BR_INDNR_SKIP_RETIRED</td>
<td>Branch Instruction architecturally executed, indirect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>excluding procedure return, not taken.</td>
</tr>
<tr>
<td>0x810E</td>
<td>BR_RETURN_ANY_RETIRED</td>
<td>Branch Instruction architecturally executed, procedure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return.</td>
</tr>
<tr>
<td>0x810F</td>
<td>BR_RETURN_SKIP_RETIRED</td>
<td>Branch Instruction architecturally executed, procedure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return, not taken.</td>
</tr>
<tr>
<td>0x8110</td>
<td>BR_IMMED_PRED_RETIRED</td>
<td>Branch Instruction architecturally executed, predicted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>immediate.</td>
</tr>
<tr>
<td>0x8111</td>
<td>BR_IMMED_MIS_PRED_RETIRED</td>
<td>Branch Instruction architecturally executed,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mispredicted immediate.</td>
</tr>
<tr>
<td>0x8112</td>
<td>BR_IND_PRED_RETIRED</td>
<td>Branch Instruction architecturally executed, predicted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>indirect.</td>
</tr>
<tr>
<td>0x8113</td>
<td>BR_IND_MIS_PRED_RETIRED</td>
<td>Branch Instruction architecturally executed,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mispredicted indirect.</td>
</tr>
<tr>
<td>0x8114</td>
<td>BR_RETURN_PRED_RETIRED</td>
<td>Branch Instruction architecturally executed, predicted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>procedure return.</td>
</tr>
<tr>
<td>0x8115</td>
<td>BR_RETURN_MIS_PRED_RETIRED</td>
<td>Branch Instruction architecturally executed,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mispredicted procedure return.</td>
</tr>
<tr>
<td>0x8116</td>
<td>BR_INDNR_PRED_RETIRED</td>
<td>Branch Instruction architecturally executed, predicted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>indirect excluding procedure return.</td>
</tr>
<tr>
<td>0x8117</td>
<td>BR_INDNR_MIS_PRED_RETIRED</td>
<td>Branch Instruction architecturally executed,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mispredicted indirect excluding procedure return.</td>
</tr>
<tr>
<td>0x8118</td>
<td>BR_TAKEN_PRED_RETIRED</td>
<td>Branch Instruction architecturally executed, predicted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>branch, taken.</td>
</tr>
<tr>
<td>0x8119</td>
<td>BR_TAKEN_MIS_PRED_RETIRED</td>
<td>Branch Instruction architecturally executed,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mispredicted branch, taken.</td>
</tr>
<tr>
<td>0x811A</td>
<td>BR_SKIP_PRED_RETIRED</td>
<td>Branch Instruction architecturally executed,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>predicted branch, not taken.</td>
</tr>
<tr>
<td>0x811B</td>
<td>BR_SKIP_MIS_PRED_RETIRED</td>
<td>Branch Instruction architecturally executed,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mispredicted branch, not taken.</td>
</tr>
</tbody>
</table>
### Table D11-8 Event index (continued)

<table>
<thead>
<tr>
<th>Event number</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x811C</td>
<td>BR_PRED_RETIRED</td>
<td>Branch Instruction architecturally executed, predicted branch.</td>
</tr>
<tr>
<td>0x811D</td>
<td>BR_IND_RETIRED</td>
<td>Instruction architecturally executed, indirect branch.</td>
</tr>
<tr>
<td>0x811E</td>
<td>BR_INDNR_RETIRED</td>
<td>Branch Instruction architecturally executed, indirect excluding procedure return.</td>
</tr>
<tr>
<td>0x811F</td>
<td>BRB_FILTRATE</td>
<td>Branch Record captured.</td>
</tr>
<tr>
<td>0x8120</td>
<td>INST_FETCH_PERCYC</td>
<td>Event in progress, INST_FETCH.</td>
</tr>
<tr>
<td>0x8121</td>
<td>MEM_ACCESS_RD_PERCYC</td>
<td>Event in progress, MEM_ACCESS_RD.</td>
</tr>
<tr>
<td>0x8124</td>
<td>INST_FETCH</td>
<td>Instruction memory access.</td>
</tr>
<tr>
<td>0x8125</td>
<td>BUS_REQ_RD_PERCYC</td>
<td>Bus read transactions in progress.</td>
</tr>
<tr>
<td>0x8126</td>
<td>BUS_REQ_WR_PERCYC</td>
<td>Bus write transactions in progress.</td>
</tr>
<tr>
<td>0x8128</td>
<td>DTLB_WALK_PERCYC</td>
<td>Event in progress, DTLB_WALK.</td>
</tr>
<tr>
<td>0x8129</td>
<td>ITLB_WALK_PERCYC</td>
<td>Event in progress, ITLB_WALK.</td>
</tr>
<tr>
<td>0x812A</td>
<td>SAMPLE_FEED_BR</td>
<td>Statistical Profiling sample taken, branch.</td>
</tr>
<tr>
<td>0x812B</td>
<td>SAMPLE_FEED_LD</td>
<td>Statistical Profiling sample taken, load.</td>
</tr>
<tr>
<td>0x812C</td>
<td>SAMPLE_FEED_ST</td>
<td>Statistical Profiling sample taken, store.</td>
</tr>
<tr>
<td>0x812D</td>
<td>SAMPLE_FEED_OP</td>
<td>Statistical Profiling sample taken, matching operation type.</td>
</tr>
<tr>
<td>0x812E</td>
<td>SAMPLE_FEED_EVENT</td>
<td>Statistical Profiling sample taken, matching events.</td>
</tr>
<tr>
<td>0x812F</td>
<td>SAMPLE_FEED_LAT</td>
<td>Statistical Profiling sample taken, exceeding minimum latency.</td>
</tr>
<tr>
<td>0x8130</td>
<td>L1D_TLB_RW</td>
<td>Level 1 data TLB demand access.</td>
</tr>
<tr>
<td>0x8131</td>
<td>L1I_TLB_RD</td>
<td>Level 1 instruction TLB demand access.</td>
</tr>
<tr>
<td>0x8132</td>
<td>L1D_TLB_PRFM</td>
<td>Level 1 data TLB software preload.</td>
</tr>
<tr>
<td>0x8133</td>
<td>L1I_TLB_PRFM</td>
<td>Level 1 instruction TLB software preload.</td>
</tr>
<tr>
<td>0x8134</td>
<td>DTLB_HWUPD</td>
<td>Data TLB hardware update of translation table.</td>
</tr>
<tr>
<td>0x8135</td>
<td>ITLB_HWUPD</td>
<td>Instruction TLB hardware update of translation table.</td>
</tr>
<tr>
<td>0x8136</td>
<td>DTLB_STEP</td>
<td>Data TLB translation table walk, step.</td>
</tr>
<tr>
<td>0x8137</td>
<td>ITLB_STEP</td>
<td>Instruction TLB translation table walk, step.</td>
</tr>
<tr>
<td>0x8138</td>
<td>DTLB_WALK_LARGE</td>
<td>Data TLB large page translation table walk.</td>
</tr>
<tr>
<td>0x8139</td>
<td>ITLB_WALK_LARGE</td>
<td>Instruction TLB large page translation table walk.</td>
</tr>
<tr>
<td>0x813A</td>
<td>DTLB_WALK_SMALL</td>
<td>Data TLB small page translation table walk.</td>
</tr>
<tr>
<td>0x813B</td>
<td>ITLB_WALK_SMALL</td>
<td>Instruction TLB small page translation table walk.</td>
</tr>
</tbody>
</table>
### Table D11-8 Event index (continued)

<table>
<thead>
<tr>
<th>Event number</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x813C</td>
<td>DTLB_WALK_RW</td>
<td>Data TLB demand access with at least one translation table walk.</td>
</tr>
<tr>
<td>0x813D</td>
<td>ITLB_WALK_RD</td>
<td>Instruction TLB demand access with at least one translation table walk.</td>
</tr>
<tr>
<td>0x813E</td>
<td>DTLB_WALK_PRFM</td>
<td>Data TLB software preload access with at least one translation table walk.</td>
</tr>
<tr>
<td>0x813F</td>
<td>ITLB_WALK_PRFM</td>
<td>Instruction TLB software preload access with at least one translation table walk.</td>
</tr>
<tr>
<td>0x8140</td>
<td>L1D_CACHE_RW</td>
<td>Level 1 data cache demand access.</td>
</tr>
<tr>
<td>0x8141</td>
<td>L1I_CACHE_RD</td>
<td>Level 1 instruction cache demand fetch.</td>
</tr>
<tr>
<td>0x8142</td>
<td>L1D_CACHE_PRFM</td>
<td>Level 1 data cache software preload.</td>
</tr>
<tr>
<td>0x8143</td>
<td>L1I_CACHE_PRFM</td>
<td>Level 1 instruction cache software preload.</td>
</tr>
<tr>
<td>0x8144</td>
<td>L1D_CACHE_MISS</td>
<td>Level 1 data cache demand access miss.</td>
</tr>
<tr>
<td>0x8145</td>
<td>L1I_CACHE_HWPRF</td>
<td>Level 1 instruction cache hardware prefetch.</td>
</tr>
<tr>
<td>0x8146</td>
<td>L1D_CACHE_REFILL_PRFM</td>
<td>Level 1 data cache refill, software preload.</td>
</tr>
<tr>
<td>0x8147</td>
<td>L1I_CACHE_REFILL_PRFM</td>
<td>Level 1 instruction cache refill, software preload.</td>
</tr>
<tr>
<td>0x8148</td>
<td>L2D_CACHE_RW</td>
<td>Level 2 data cache demand access.</td>
</tr>
<tr>
<td>0x8149</td>
<td>L2I_CACHE_RD</td>
<td>Level 2 instruction cache demand fetch.</td>
</tr>
<tr>
<td>0x814A</td>
<td>L2D_CACHE_PRFM</td>
<td>Level 2 data cache software preload.</td>
</tr>
<tr>
<td>0x814B</td>
<td>L2I_CACHE_PRFM</td>
<td>Level 2 instruction cache software preload.</td>
</tr>
<tr>
<td>0x814C</td>
<td>L2D_CACHE_MISS</td>
<td>Level 2 data cache demand access miss.</td>
</tr>
<tr>
<td>0x814D</td>
<td>L2I_CACHE_HWPRF</td>
<td>Level 2 instruction cache hardware prefetch.</td>
</tr>
<tr>
<td>0x814E</td>
<td>L2D_CACHE_REFILL_PRFM</td>
<td>Level 2 data cache refill, software preload.</td>
</tr>
<tr>
<td>0x814F</td>
<td>L2I_CACHE_REFILL_PRFM</td>
<td>Level 2 instruction cache refill, software preload.</td>
</tr>
<tr>
<td>0x8150</td>
<td>L3D_CACHE_RW</td>
<td>Level 3 data cache demand access.</td>
</tr>
<tr>
<td>0x8151</td>
<td>L3D_CACHE_PRFM</td>
<td>Level 3 data cache software preload.</td>
</tr>
<tr>
<td>0x8152</td>
<td>L3D_CACHE_MISS</td>
<td>Level 3 data cache demand access miss.</td>
</tr>
<tr>
<td>0x8153</td>
<td>L3D_CACHE_REFILL_PRFM</td>
<td>Level 3 data cache refill, software preload.</td>
</tr>
<tr>
<td>0x8154</td>
<td>L1D_CACHE_HWPRF</td>
<td>Level 1 data cache hardware prefetch.</td>
</tr>
<tr>
<td>0x8155</td>
<td>L2D_CACHE_HWPRF</td>
<td>Level 2 data cache hardware prefetch.</td>
</tr>
<tr>
<td>0x8156</td>
<td>L3D_CACHE_HWPRF</td>
<td>Level 3 data cache hardware prefetch.</td>
</tr>
<tr>
<td>0x8157</td>
<td>LL_CACHE_HWPRF</td>
<td>Last level cache hardware prefetch.</td>
</tr>
<tr>
<td>0x8158</td>
<td>STALL_FRONTEND_MEMBOUND</td>
<td>Frontend stall cycles, memory bound.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>0x8159</td>
<td>STALL_FRONTEND_L1I</td>
<td>Frontend stall cycles, level 1 instruction cache.</td>
</tr>
<tr>
<td>0x815A</td>
<td>STALL_FRONTEND_L2I</td>
<td>Frontend stall cycles, level 2 instruction cache.</td>
</tr>
<tr>
<td>0x815B</td>
<td>STALL_FRONTEND_MEM</td>
<td>Frontend stall cycles, last level PE cache or memory.</td>
</tr>
<tr>
<td>0x815C</td>
<td>STALL_FRONTEND_TLB</td>
<td>Frontend stall cycles, TLB.</td>
</tr>
<tr>
<td>0x8160</td>
<td>STALL_FRONTEND_CPUBOUND</td>
<td>Frontend stall cycles, processor bound.</td>
</tr>
<tr>
<td>0x8161</td>
<td>STALL_FRONTEND_FLOW</td>
<td>Frontend stall cycles, flow control.</td>
</tr>
<tr>
<td>0x8162</td>
<td>STALL_FRONTEND_FLUSH</td>
<td>Frontend stall cycles, flush recovery.</td>
</tr>
<tr>
<td>0x8163</td>
<td>STALL_FRONTEND_RENAME</td>
<td>Frontend stall cycles, rename full.</td>
</tr>
<tr>
<td>0x8164</td>
<td>STALL_BACKEND_MEMBOUND</td>
<td>Backend stall cycles, memory bound.</td>
</tr>
<tr>
<td>0x8165</td>
<td>STALL_BACKEND_L1D</td>
<td>Backend stall cycles, level 1 data cache.</td>
</tr>
<tr>
<td>0x8166</td>
<td>STALL_BACKEND_L2D</td>
<td>Backend stall cycles, level 2 data cache.</td>
</tr>
<tr>
<td>0x8167</td>
<td>STALL_BACKEND_TLB</td>
<td>Backend stall cycles, TLB.</td>
</tr>
<tr>
<td>0x8168</td>
<td>STALL_BACKEND_ST</td>
<td>Backend stall cycles, store.</td>
</tr>
<tr>
<td>0x816A</td>
<td>STALL_BACKEND_CPUBOUND</td>
<td>Backend stall cycles, processor bound.</td>
</tr>
<tr>
<td>0x816B</td>
<td>STALL_BACKEND_BUSY</td>
<td>Backend stall cycles, backend busy.</td>
</tr>
<tr>
<td>0x816C</td>
<td>STALL_BACKEND_ILOCK</td>
<td>Backend stall cycles, input dependency.</td>
</tr>
<tr>
<td>0x816D</td>
<td>STALL_BACKEND_RENAME</td>
<td>Backend stall cycles, rename full.</td>
</tr>
<tr>
<td>0x816E</td>
<td>STALL_BACKEND_ATOMIC</td>
<td>Backend stall cycles, atomic operation.</td>
</tr>
<tr>
<td>0x816F</td>
<td>STALL_BACKEND_MEMCPYSET</td>
<td>Backend stall cycles, Memory Copy or Set operation.</td>
</tr>
<tr>
<td>0x8170</td>
<td>CAS_NEAR_FAIL</td>
<td>Atomic memory Operation speculatively executed,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compare and Swap fail.</td>
</tr>
<tr>
<td>0x8171</td>
<td>CAS_NEAR_PASS</td>
<td>Atomic memory Operation speculatively executed,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compare and Swap pass.</td>
</tr>
<tr>
<td>0x8172</td>
<td>CAS_NEAR_SPEC</td>
<td>Atomic memory Operation speculatively executed,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compare and Swap near.</td>
</tr>
<tr>
<td>0x8173</td>
<td>CAS_FAR_SPEC</td>
<td>Atomic memory Operation speculatively executed,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compare and Swap far.</td>
</tr>
<tr>
<td>0x8174</td>
<td>CAS_SPEC</td>
<td>Atomic memory Operation speculatively executed,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compare and Swap.</td>
</tr>
<tr>
<td>0x8175</td>
<td>LSE_LD_SPEC</td>
<td>Atomic memory Operation speculatively executed,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>load.</td>
</tr>
<tr>
<td>0x8176</td>
<td>LSE_ST_SPEC</td>
<td>Atomic memory Operation speculatively executed,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>store.</td>
</tr>
<tr>
<td>0x8177</td>
<td>LSE_LDST_SPEC</td>
<td>Atomic memory Operation speculatively executed,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>load or store.</td>
</tr>
</tbody>
</table>
Table D11-8 Event index (continued)

<table>
<thead>
<tr>
<th>Event number</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8178</td>
<td>REMOTE_ACCESS_WR</td>
<td>Access to another socket in a multi-socket system, write.</td>
</tr>
<tr>
<td>0x8179</td>
<td>BRNL_INDNR_TAKEN_RETIRED</td>
<td>Branch Instruction architecturally executed, indirect branch without link excluding procedure return, taken.</td>
</tr>
<tr>
<td>0x817A</td>
<td>BL_TAKEN_RETIRED</td>
<td>Branch Instruction architecturally executed, branch with link, taken.</td>
</tr>
<tr>
<td>0x817B</td>
<td>BRNL_TAKEN_RETIRED</td>
<td>Branch Instruction architecturally executed, branch without link, taken.</td>
</tr>
<tr>
<td>0x817C</td>
<td>BL_IND_TAKEN_RETIRED</td>
<td>Branch Instruction architecturally executed, indirect branch with link, taken.</td>
</tr>
<tr>
<td>0x817D</td>
<td>BRNL_IND_TAKEN_RETIRED</td>
<td>Branch Instruction architecturally executed, indirect branch without link, taken.</td>
</tr>
<tr>
<td>0x817E</td>
<td>BL_IMMED_TAKEN_RETIRED</td>
<td>Branch Instruction architecturally executed, direct branch with link, taken.</td>
</tr>
<tr>
<td>0x817F</td>
<td>BRNL_IMMED_TAKEN_RETIRED</td>
<td>Branch Instruction architecturally executed, direct branch without link, taken.</td>
</tr>
<tr>
<td>0x8180</td>
<td>BR_UNCOND_RETIRED</td>
<td>Branch Instruction architecturally executed, unconditional branch.</td>
</tr>
<tr>
<td>0x8181</td>
<td>BR_COND_RETIRED</td>
<td>Branch Instruction architecturally executed, conditional branch.</td>
</tr>
<tr>
<td>0x8182</td>
<td>BR_COND_TAKEN_RETIRED</td>
<td>Branch Instruction architecturally executed, conditional branch, taken.</td>
</tr>
<tr>
<td>0x8183</td>
<td>BR_HINT_COND_RETIRED</td>
<td>Branch Instruction architecturally executed, hinted conditional.</td>
</tr>
<tr>
<td>0x8184</td>
<td>BR_HINT_COND_PRED_RETIRED</td>
<td>Branch Instruction architecturally executed, predicted hinted conditional.</td>
</tr>
<tr>
<td>0x8185</td>
<td>BR_HINT_COND_MIS_PRED_RETIRED</td>
<td>Branch Instruction architecturally executed, mispredicted hinted conditional.</td>
</tr>
<tr>
<td>0x8186</td>
<td>UOP_RETIRED</td>
<td>Micro-operation architecturally executed.</td>
</tr>
<tr>
<td>0x8188</td>
<td>DTLB_WALK_BLOCK</td>
<td>Data TLB block translation table walk.</td>
</tr>
<tr>
<td>0x8189</td>
<td>ITLB_WALK_BLOCK</td>
<td>Instruction TLB block translation table walk.</td>
</tr>
<tr>
<td>0x818A</td>
<td>DTLB_WALK_PAGE</td>
<td>Data TLB page translation table walk.</td>
</tr>
<tr>
<td>0x818B</td>
<td>ITLB_WALK_PAGE</td>
<td>Instruction TLB page translation table walk.</td>
</tr>
<tr>
<td>0x818D</td>
<td>BUS_REQ_RD</td>
<td>Bus request, read.</td>
</tr>
<tr>
<td>0x818E</td>
<td>BUS_REQ_WR</td>
<td>Bus request, write.</td>
</tr>
<tr>
<td>0x818F</td>
<td>BUS_REQ</td>
<td>Bus request.</td>
</tr>
<tr>
<td>0x8190</td>
<td>ISNP_HIT_RD</td>
<td>Snoop hit, demand instruction fetch.</td>
</tr>
<tr>
<td>0x8191</td>
<td>ISNP_HIT_NEAR_RD</td>
<td>Snoop hit in near cache, demand instruction fetch.</td>
</tr>
</tbody>
</table>
## Table D11-8 Event index (continued)

<table>
<thead>
<tr>
<th>Event number</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8192</td>
<td>ISNP_HIT_FAR_RD</td>
<td>Snoop hit in far cache, demand instruction fetch.</td>
</tr>
<tr>
<td>0x8193</td>
<td>ISNP_HIT_REMOTE_RD</td>
<td>Snoop hit in remote cache, demand instruction fetch.</td>
</tr>
<tr>
<td>0x8194</td>
<td>DSNP_HIT_RD</td>
<td>Snoop hit, demand data read.</td>
</tr>
<tr>
<td>0x8195</td>
<td>DSNP_HIT_NEAR_RD</td>
<td>Snoop hit in near cache, demand data read.</td>
</tr>
<tr>
<td>0x8196</td>
<td>DSNP_HIT_FAR_RD</td>
<td>Snoop hit in far cache, demand data read.</td>
</tr>
<tr>
<td>0x8197</td>
<td>DSNP_HIT_REMOTE_RD</td>
<td>Snoop hit in remote cache, demand data read.</td>
</tr>
<tr>
<td>0x8198</td>
<td>DSNP_HIT_WR</td>
<td>Snoop hit, demand data write.</td>
</tr>
<tr>
<td>0x8199</td>
<td>DSNP_HIT_NEAR_WR</td>
<td>Snoop hit in near cache, demand data write.</td>
</tr>
<tr>
<td>0x819A</td>
<td>DSNP_HIT_FAR_WR</td>
<td>Snoop hit in far cache, demand data write.</td>
</tr>
<tr>
<td>0x819B</td>
<td>DSNP_HIT_REMOTE_WR</td>
<td>Snoop hit in remote cache, demand data write.</td>
</tr>
<tr>
<td>0x819C</td>
<td>DSNP_HIT_RW</td>
<td>Snoop hit, demand data access.</td>
</tr>
<tr>
<td>0x819D</td>
<td>DSNP_HIT_NEAR_RW</td>
<td>Snoop hit in near cache, demand data access.</td>
</tr>
<tr>
<td>0x819E</td>
<td>DSNP_HIT_FAR_RW</td>
<td>Snoop hit in far cache, demand data access.</td>
</tr>
<tr>
<td>0x819F</td>
<td>DSNP_HIT_REMOTE_RW</td>
<td>Snoop hit in remote cache, demand data access.</td>
</tr>
<tr>
<td>0x81A0</td>
<td>DSNP_HIT_PRFM</td>
<td>Snoop hit, software data preload.</td>
</tr>
<tr>
<td>0x81A1</td>
<td>DSNP_HIT_NEAR_PRFM</td>
<td>Snoop hit in near cache, software data preload.</td>
</tr>
<tr>
<td>0x81A2</td>
<td>DSNP_HIT_FAR_PRFM</td>
<td>Snoop hit in far cache, software data preload.</td>
</tr>
<tr>
<td>0x81A3</td>
<td>DSNP_HIT_REMOTE_PRFM</td>
<td>Snoop hit in remote cache, software data preload.</td>
</tr>
<tr>
<td>0x81A4</td>
<td>DSNP_HIT_HWPRF</td>
<td>Snoop hit, hardware data prefetch.</td>
</tr>
<tr>
<td>0x81A5</td>
<td>DSNP_HIT_NEAR_HWPRF</td>
<td>Snoop hit in near cache, hardware data prefetch.</td>
</tr>
<tr>
<td>0x81A6</td>
<td>DSNP_HIT_FAR_HWPRF</td>
<td>Snoop hit in far cache, hardware data prefetch.</td>
</tr>
<tr>
<td>0x81A7</td>
<td>DSNP_HIT_REMOTE_HWPRF</td>
<td>Snoop hit in remote cache, hardware data prefetch.</td>
</tr>
<tr>
<td>0x81A8</td>
<td>ISNP_HIT_PRFM</td>
<td>Snoop hit, software instruction preload.</td>
</tr>
<tr>
<td>0x81A9</td>
<td>ISNP_HIT_NEAR_PRFM</td>
<td>Snoop hit in near cache, software instruction preload.</td>
</tr>
<tr>
<td>0x81AA</td>
<td>ISNP_HIT_FAR_PRFM</td>
<td>Snoop hit in far cache, software instruction preload.</td>
</tr>
<tr>
<td>0x81AB</td>
<td>ISNP_HIT_REMOTE_PRFM</td>
<td>Snoop hit in remote cache, software instruction preload.</td>
</tr>
<tr>
<td>0x81AC</td>
<td>ISNP_HIT_HWPRF</td>
<td>Snoop hit, hardware instruction prefetch.</td>
</tr>
<tr>
<td>0x81AD</td>
<td>ISNP_HIT_NEAR_HWPRF</td>
<td>Snoop hit in near cache, hardware instruction prefetch.</td>
</tr>
<tr>
<td>0x81AE</td>
<td>ISNP_HIT_FAR_HWPRF</td>
<td>Snoop hit in far cache, hardware instruction prefetch.</td>
</tr>
<tr>
<td>0x81AF</td>
<td>ISNP_HIT_REMOTE_HWPRF</td>
<td>Snoop hit in remote cache, hardware instruction prefetch.</td>
</tr>
<tr>
<td>0x81B0</td>
<td>ISNP_HIT</td>
<td>Snoop hit, instruction.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>0x81B1</td>
<td>ISNP_HIT_NEAR</td>
<td>Snoop hit in near cache, instruction.</td>
</tr>
<tr>
<td>0x81B2</td>
<td>ISNP_HIT_FAR</td>
<td>Snoop hit in far cache, instruction.</td>
</tr>
<tr>
<td>0x81B3</td>
<td>ISNP_HIT_REMOTE</td>
<td>Snoop hit in remote cache, instruction.</td>
</tr>
<tr>
<td>0x81B4</td>
<td>DSNP_HIT</td>
<td>Snoop hit, data.</td>
</tr>
<tr>
<td>0x81B5</td>
<td>DSNP_HIT_NEAR</td>
<td>Snoop hit in near cache, data.</td>
</tr>
<tr>
<td>0x81B6</td>
<td>DSNP_HIT_FAR</td>
<td>Snoop hit in far cache, data.</td>
</tr>
<tr>
<td>0x81B7</td>
<td>DSNP_HIT_REMOTE</td>
<td>Snoop hit in remote cache, data.</td>
</tr>
<tr>
<td>0x81B8</td>
<td>L1I_CACHE_REFILL_HWPRF</td>
<td>Level 1 instruction cache refill, hardware prefetch.</td>
</tr>
<tr>
<td>0x81B9</td>
<td>L2I_CACHE_REFILL_HWPRF</td>
<td>Level 2 instruction cache refill, hardware prefetch.</td>
</tr>
<tr>
<td>0x81B0</td>
<td>L1D_CACHE_REFILL_HWPRF</td>
<td>Level 1 data cache refill, hardware prefetch.</td>
</tr>
<tr>
<td>0x81B1</td>
<td>L2D_CACHE_REFILL_HWPRF</td>
<td>Level 2 data cache refill, hardware prefetch.</td>
</tr>
<tr>
<td>0x81B2</td>
<td>L3D_CACHE_REFILL_HWPRF</td>
<td>Level 3 data cache refill, hardware prefetch.</td>
</tr>
<tr>
<td>0x81B3</td>
<td>L1D_CACHE_HIT_RD_FPRFM</td>
<td>Level 1 data cache demand first hit, read, fetched by software preload.</td>
</tr>
<tr>
<td>0x81B4</td>
<td>L2D_CACHE.Hit_RW</td>
<td>Level 2 data cache demand access hit, write.</td>
</tr>
<tr>
<td>0x81B5</td>
<td>L3D_CACHE_HIT_RW</td>
<td>Level 3 data cache demand access hit, write.</td>
</tr>
<tr>
<td>0x81B6</td>
<td>L1D_CACHE_HIT_RD</td>
<td>Level 1 instruction cache demand fetch hit.</td>
</tr>
<tr>
<td>0x81B7</td>
<td>L2D_CACHE_HIT_RD</td>
<td>Level 2 instruction cache demand fetch hit.</td>
</tr>
<tr>
<td>0x81B8</td>
<td>L3D_CACHE_HIT_RD</td>
<td>Level 3 instruction cache demand fetch hit.</td>
</tr>
<tr>
<td>0x81B9</td>
<td>LL_CACHE_HIT_RD</td>
<td>Last level cache demand hit, read.</td>
</tr>
<tr>
<td>0x81BC</td>
<td>L1D_CACHE_HIT_RW</td>
<td>Level 1 data cache demand access hit, write.</td>
</tr>
<tr>
<td>0x81BD</td>
<td>L2D_CACHE_HIT_RW</td>
<td>Level 2 data cache demand access hit.</td>
</tr>
<tr>
<td>0x81BE</td>
<td>L3D_CACHE_HIT_RW</td>
<td>Level 3 data cache demand access hit.</td>
</tr>
<tr>
<td>0x81BF</td>
<td>LL_CACHE_HIT_RW</td>
<td>Last level cache demand access hit.</td>
</tr>
<tr>
<td>0x81C0</td>
<td>L1I_CACHE_HIT_RD</td>
<td>Level 1 instruction cache demand hit, read.</td>
</tr>
<tr>
<td>0x81C1</td>
<td>L2I_CACHE_HIT_RD</td>
<td>Level 2 instruction cache demand hit, read.</td>
</tr>
<tr>
<td>0x81C4</td>
<td>L1D_CACHE_HIT_RD</td>
<td>Level 1 data cache demand hit, read.</td>
</tr>
<tr>
<td>0x81C5</td>
<td>L2D_CACHE_HIT_RD</td>
<td>Level 2 data cache demand hit, read.</td>
</tr>
<tr>
<td>0x81C6</td>
<td>L3D_CACHE_HIT_RD</td>
<td>Level 3 data cache demand hit, read.</td>
</tr>
<tr>
<td>0x81C7</td>
<td>LL_CACHE_HIT_RD</td>
<td>Last level cache demand hit, read.</td>
</tr>
<tr>
<td>0x81C8</td>
<td>L1D_CACHE_HIT_WR</td>
<td>Level 1 data cache demand access hit, write.</td>
</tr>
<tr>
<td>0x81C9</td>
<td>L2D_CACHE_HIT_WR</td>
<td>Level 2 data cache demand access hit.</td>
</tr>
<tr>
<td>0x81CA</td>
<td>L3D_CACHE_HIT_WR</td>
<td>Level 3 data cache demand access hit.</td>
</tr>
<tr>
<td>0x81CB</td>
<td>LL_CACHE_HIT_WR</td>
<td>Last level cache demand access hit, write.</td>
</tr>
<tr>
<td>0x81CC</td>
<td>L1D_CACHE_HIT_RW</td>
<td>Level 1 data cache demand access hit.</td>
</tr>
<tr>
<td>0x81CD</td>
<td>L2D_CACHE_HIT_RW</td>
<td>Level 2 data cache demand access hit.</td>
</tr>
<tr>
<td>0x81CE</td>
<td>L3D_CACHE_HIT_RW</td>
<td>Level 3 data cache demand access hit.</td>
</tr>
<tr>
<td>0x81CF</td>
<td>LL_CACHE_HIT_RW</td>
<td>Last level cache demand access hit.</td>
</tr>
<tr>
<td>0x81D0</td>
<td>L1I_CACHE_HIT_RD_FPRFM</td>
<td>Level 1 instruction cache demand fetch first hit, fetched by software preload.</td>
</tr>
<tr>
<td>0x81D1</td>
<td>L2I_CACHE_HIT_RD_FPRFM</td>
<td>Level 2 instruction cache demand fetch first hit, fetched by software preload.</td>
</tr>
<tr>
<td>0x81D2</td>
<td>L1D_CACHE_HIT_RD_FPRFM</td>
<td>Level 1 data cache demand first hit, read, fetched by software preload.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>0x81D5</td>
<td>L2D_CACHE_HIT_RD_FPRFM</td>
<td>Level 2 data cache demand first hit, read, fetched by software preload.</td>
</tr>
<tr>
<td>0x81D6</td>
<td>L3D_CACHE_HIT_RD_FPRFM</td>
<td>Level 3 data cache demand first hit, read, fetched by software preload.</td>
</tr>
<tr>
<td>0x81D7</td>
<td>LL_CACHE_HIT_RD_FPRFM</td>
<td>Last level cache demand first hit, read, fetched by software preload.</td>
</tr>
<tr>
<td>0x81D8</td>
<td>L1D_CACHE_HIT_WR_FPRFM</td>
<td>Level 1 data cache demand access first hit, write, fetched by software preload.</td>
</tr>
<tr>
<td>0x81D9</td>
<td>L2D_CACHE_HIT_WR_FPRFM</td>
<td>Level 2 data cache demand access first hit, write, fetched by software preload.</td>
</tr>
<tr>
<td>0x81DA</td>
<td>L3D_CACHE_HIT_WR_FPRFM</td>
<td>Level 3 data cache demand access first hit, write, fetched by software preload.</td>
</tr>
<tr>
<td>0x81DB</td>
<td>LL_CACHE_HIT_WR_FPRFM</td>
<td>Last level cache demand access first hit, write, fetched by software preload.</td>
</tr>
<tr>
<td>0x81DC</td>
<td>L1D_CACHE_HIT_RW_FPRFM</td>
<td>Level 1 data cache demand access first hit, fetched by software preload.</td>
</tr>
<tr>
<td>0x81DD</td>
<td>L2D_CACHE_HIT_RW_FPRFM</td>
<td>Level 2 data cache demand access first hit, fetched by software preload.</td>
</tr>
<tr>
<td>0x81DE</td>
<td>L3D_CACHE_HIT_RW_FPRFM</td>
<td>Level 3 data cache demand access first hit, fetched by software preload.</td>
</tr>
<tr>
<td>0x81DF</td>
<td>LL_CACHE_HIT_RW_FPRFM</td>
<td>Last level cache demand access first hit, fetched by software preload.</td>
</tr>
<tr>
<td>0x81E0</td>
<td>L1I_CACHE_HIT_RD_FHWPRF</td>
<td>Level 1 instruction cache demand fetch first hit, fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x81E1</td>
<td>L2I_CACHE_HIT_RD_FHWPRF</td>
<td>Level 2 instruction cache demand fetch first hit, fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x81E4</td>
<td>L1D_CACHE_HIT_RD_FHWPRF</td>
<td>Level 1 data cache demand first hit, read, fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x81E5</td>
<td>L2D_CACHE_HIT_RD_FHWPRF</td>
<td>Level 2 data cache demand first hit, read, fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x81E6</td>
<td>L3D_CACHE_HIT_RD_FHWPRF</td>
<td>Level 3 data cache demand first hit, read, fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x81E7</td>
<td>LL_CACHE_HIT_RD_FHWPRF</td>
<td>Last level cache demand first hit, read, fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x81E8</td>
<td>L1D_CACHE_HIT_WR_FHWPRF</td>
<td>Level 1 data cache demand access first hit, write, fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x81E9</td>
<td>L2D_CACHE_HIT_WR_FHWPRF</td>
<td>Level 2 data cache demand access first hit, write, fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x81EA</td>
<td>L3D_CACHE_HIT_WR_FHWPRF</td>
<td>Level 3 data cache demand access first hit, write, fetched by hardware prefetcher.</td>
</tr>
</tbody>
</table>
Table D11-8 Event index (continued)

<table>
<thead>
<tr>
<th>Event number</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x81EB</td>
<td>LL_CACHE_HIT_WR_FHWPRF</td>
<td>Last level cache demand access first hit, write, fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x81EC</td>
<td>L1D_CACHE_HIT_RW_FHWPRF</td>
<td>Level 1 data cache demand access first hit, fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x81ED</td>
<td>L2D_CACHE_HIT_RW_FHWPRF</td>
<td>Level 2 data cache demand access first hit, fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x81EE</td>
<td>L3D_CACHE_HIT_RW_FHWPRF</td>
<td>Level 3 data cache demand access first hit, fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x81EF</td>
<td>LL_CACHE_HIT_RW_FHWPRF</td>
<td>Last level cache demand access first hit, fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x81F0</td>
<td>L1I_CACHE_HIT_RD_FPRF</td>
<td>Level 1 instruction cache demand fetch first hit, fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x81F1</td>
<td>L2I_CACHE_HIT_RD_FPRF</td>
<td>Level 2 instruction cache demand fetch first hit, fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x81F4</td>
<td>L1D_CACHE_HIT_RD_FPRF</td>
<td>Level 1 data cache demand first hit, read, fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x81F5</td>
<td>L2D_CACHE_HIT_RD_FPRF</td>
<td>Level 2 data cache demand first hit, read, fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x81F6</td>
<td>L3D_CACHE_HIT_RD_FPRF</td>
<td>Level 3 data cache demand first hit, read, fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x81F7</td>
<td>LL_CACHE_HIT_RD_FPRF</td>
<td>Last level cache demand first hit, read, fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x81F8</td>
<td>L1D_CACHE_HIT_WR_FPRF</td>
<td>Level 1 data cache demand access first hit, write, fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x81F9</td>
<td>L2D_CACHE_HIT_WR_FPRF</td>
<td>Level 2 data cache demand access first hit, write, fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x81FA</td>
<td>L3D_CACHE_HIT_WR_FPRF</td>
<td>Level 3 data cache demand access first hit, write, fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x81FB</td>
<td>LL_CACHE_HIT_WR_FPRF</td>
<td>Last level cache demand access first hit, write, fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x81FC</td>
<td>L1D_CACHE_HIT_RW_FPRF</td>
<td>Level 1 data cache demand access first hit, fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x81FD</td>
<td>L2D_CACHE_HIT_RW_FPRF</td>
<td>Level 2 data cache demand access first hit, fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x81FE</td>
<td>L3D_CACHE_HIT_RW_FPRF</td>
<td>Level 3 data cache demand access first hit, fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x81FF</td>
<td>LL_CACHE_HIT_RW_FPRF</td>
<td>Last level cache demand access first hit, fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x8200</td>
<td>L1I_CACHE_HIT</td>
<td>Level 1 instruction cache hit.</td>
</tr>
<tr>
<td>0x8201</td>
<td>L2I_CACHE_HIT</td>
<td>Level 2 instruction cache hit.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>0x8204</td>
<td>L1D_CACHE_HIT</td>
<td>Level 1 data cache hit.</td>
</tr>
<tr>
<td>0x8205</td>
<td>L2D_CACHE_HIT</td>
<td>Level 2 data cache hit.</td>
</tr>
<tr>
<td>0x8206</td>
<td>L3D_CACHE_HIT</td>
<td>Level 3 data cache hit.</td>
</tr>
<tr>
<td>0x8207</td>
<td>LL_CACHE_HIT</td>
<td>Last level cache hit.</td>
</tr>
<tr>
<td>0x8208</td>
<td>L1I_CACHE_HIT_PRFM</td>
<td>Level 1 instruction cache software preload hit.</td>
</tr>
<tr>
<td>0x8209</td>
<td>L2I_CACHE_HIT_PRFM</td>
<td>Level 2 instruction cache software preload hit.</td>
</tr>
<tr>
<td>0x820C</td>
<td>L1D_CACHE_HIT_PRFM</td>
<td>Level 1 data cache software preload hit.</td>
</tr>
<tr>
<td>0x820D</td>
<td>L2D_CACHE_HIT_PRFM</td>
<td>Level 2 data cache software preload hit.</td>
</tr>
<tr>
<td>0x820E</td>
<td>L3D_CACHE_HIT_PRFM</td>
<td>Level 3 data cache software preload hit.</td>
</tr>
<tr>
<td>0x820F</td>
<td>LL_CACHE_HIT_PRFM</td>
<td>Last level cache software preload hit.</td>
</tr>
<tr>
<td>0x8214</td>
<td>L1D_CACHE_HITM_RD</td>
<td>Level 1 data cache demand hit modified, read.</td>
</tr>
<tr>
<td>0x8215</td>
<td>L2D_CACHE_HITM_RD</td>
<td>Level 2 data cache demand hit modified, read.</td>
</tr>
<tr>
<td>0x8216</td>
<td>L3D_CACHE_HITM_RD</td>
<td>Level 3 data cache demand hit modified, read.</td>
</tr>
<tr>
<td>0x8217</td>
<td>LL_CACHE_HITM_RD</td>
<td>Last level cache demand hit modified, read.</td>
</tr>
<tr>
<td>0x8218</td>
<td>L1D_CACHE_HITM_WR</td>
<td>Level 1 data cache demand access hit modified, write.</td>
</tr>
<tr>
<td>0x8219</td>
<td>L2D_CACHE_HITM_WR</td>
<td>Level 2 data cache demand access hit modified, write.</td>
</tr>
<tr>
<td>0x821A</td>
<td>L3D_CACHE_HITM_WR</td>
<td>Level 3 data cache demand access hit modified, write.</td>
</tr>
<tr>
<td>0x821B</td>
<td>LL_CACHE_HITM_WR</td>
<td>Last level cache demand access hit modified, write.</td>
</tr>
<tr>
<td>0x821C</td>
<td>L1D_CACHE_HITM_RW</td>
<td>Level 1 data cache demand access hit modified.</td>
</tr>
<tr>
<td>0x821D</td>
<td>L2D_CACHE_HITM_RW</td>
<td>Level 2 data cache demand access hit modified.</td>
</tr>
<tr>
<td>0x821E</td>
<td>L3D_CACHE_HITM_RW</td>
<td>Level 3 data cache demand access hit modified.</td>
</tr>
<tr>
<td>0x821F</td>
<td>LL_CACHE_HITM_RW</td>
<td>Last level cache demand access hit modified.</td>
</tr>
<tr>
<td>0x8224</td>
<td>DSNP_HITM_RD</td>
<td>Snoop hit, demand data read, modified.</td>
</tr>
<tr>
<td>0x8225</td>
<td>DSNP_HITM_NEAR_RD</td>
<td>Snoop hit in near cache, demand data read, modified.</td>
</tr>
<tr>
<td>0x8226</td>
<td>DSNP_HITM_FAR_RD</td>
<td>Snoop hit in far cache, demand data read, modified.</td>
</tr>
<tr>
<td>0x8227</td>
<td>DSNP_HITM_REMOTE_RD</td>
<td>Snoop hit in remote cache, demand data read, modified.</td>
</tr>
<tr>
<td>0x8228</td>
<td>DSNP_HITM_WR</td>
<td>Snoop hit, demand data write, modified.</td>
</tr>
<tr>
<td>0x8229</td>
<td>DSNP_HITM_NEAR_WR</td>
<td>Snoop hit in near cache, demand data write, modified.</td>
</tr>
<tr>
<td>0x822A</td>
<td>DSNP_HITM_FAR_WR</td>
<td>Snoop hit in far cache, demand data write, modified.</td>
</tr>
<tr>
<td>0x822B</td>
<td>DSNP_HITM_REMOTE_WR</td>
<td>Snoop hit in remote cache, demand data write, modified.</td>
</tr>
<tr>
<td>0x822C</td>
<td>DSNP_HITM_RW</td>
<td>Snoop hit, demand data access, modified.</td>
</tr>
<tr>
<td>0x822D</td>
<td>DSNP_HITM_NEAR_RW</td>
<td>Snoop hit in near cache, demand data access, modified.</td>
</tr>
</tbody>
</table>
Table D11-8 Event index (continued)

<table>
<thead>
<tr>
<th>Event number</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x822E</td>
<td>DSNP_HITM_FAR_RW</td>
<td>Snoop hit in far cache, demand data access, modified.</td>
</tr>
<tr>
<td>0x822F</td>
<td>DSNP_HITM_REMOTE_RW</td>
<td>Snoop hit in remote cache, demand data access, modified.</td>
</tr>
<tr>
<td>0x8230</td>
<td>LOCAL_MEM</td>
<td>Access to memory attached to this device.</td>
</tr>
<tr>
<td>0x8231</td>
<td>LOCAL_MEM_RD</td>
<td>Access to memory attached to this device, read.</td>
</tr>
<tr>
<td>0x8232</td>
<td>LOCAL_MEM_WR</td>
<td>Access to memory attached to this device, write.</td>
</tr>
<tr>
<td>0x8233</td>
<td>LOCAL_MEM_RW</td>
<td>Access to memory attached to this device, demand read or write.</td>
</tr>
<tr>
<td>0x8234</td>
<td>LOCAL_MEM_PRFM</td>
<td>Access to memory attached to this device, preload or prefetch.</td>
</tr>
<tr>
<td>0x8238</td>
<td>REMOTE_MEM</td>
<td>Access to memory attached to another socket in a multi-socket system.</td>
</tr>
<tr>
<td>0x8239</td>
<td>REMOTE_MEM_RD</td>
<td>Access to memory attached to another socket in a multi-socket system, read.</td>
</tr>
<tr>
<td>0x823A</td>
<td>REMOTE_MEM_WR</td>
<td>Access to memory attached to another socket in a multi-socket system, write.</td>
</tr>
<tr>
<td>0x823B</td>
<td>REMOTE_MEM_RW</td>
<td>Access to memory attached to another socket in a multi-socket system, demand read or write.</td>
</tr>
<tr>
<td>0x823C</td>
<td>REMOTE_MEM_PRFM</td>
<td>Access to memory attached to another socket in a multi-socket system, preload or prefetch.</td>
</tr>
<tr>
<td>0x8240</td>
<td>L1I_LFB_HIT_RD</td>
<td>Level 1 instruction cache demand fetch line-fill buffer hit.</td>
</tr>
<tr>
<td>0x8241</td>
<td>L2I_LFB_HIT_RD</td>
<td>Level 2 instruction cache demand fetch line-fill buffer hit.</td>
</tr>
<tr>
<td>0x8244</td>
<td>L1D_LFB_HIT_RD</td>
<td>Level 1 data cache demand line-fill buffer hit, read.</td>
</tr>
<tr>
<td>0x8245</td>
<td>L2D_LFB_HIT_RD</td>
<td>Level 2 data cache demand line-fill buffer hit, read.</td>
</tr>
<tr>
<td>0x8246</td>
<td>L3D_LFB_HIT_RD</td>
<td>Level 3 data cache demand line-fill buffer hit, read.</td>
</tr>
<tr>
<td>0x8247</td>
<td>LL_LFB_HIT_RD</td>
<td>Last level cache demand line-fill buffer hit, read.</td>
</tr>
<tr>
<td>0x8248</td>
<td>L1D_LFB_HIT_WR</td>
<td>Level 1 data cache demand access line-fill buffer hit, write.</td>
</tr>
<tr>
<td>0x8249</td>
<td>L2D_LFB_HIT_WR</td>
<td>Level 2 data cache demand access line-fill buffer hit, write.</td>
</tr>
<tr>
<td>0x824A</td>
<td>L3D_LFB_HIT_WR</td>
<td>Level 3 data cache demand access line-fill buffer hit, write.</td>
</tr>
<tr>
<td>0x824B</td>
<td>LL_LFB_HIT_WR</td>
<td>Last level cache demand access line-fill buffer hit, write.</td>
</tr>
<tr>
<td>0x824C</td>
<td>L1D_LFB_HIT_RW</td>
<td>Level 1 data cache demand access line-fill buffer hit.</td>
</tr>
<tr>
<td>0x824D</td>
<td>L2D_LFB_HIT_RW</td>
<td>Level 2 data cache demand access line-fill buffer hit.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>0x824E</td>
<td>L3D_LFB_HIT_RW</td>
<td>Level 3 data cache demand access line-fill buffer hit.</td>
</tr>
<tr>
<td>0x824F</td>
<td>LL_LFB_HIT_RW</td>
<td>Last level cache demand access line-fill buffer hit.</td>
</tr>
<tr>
<td>0x8250</td>
<td>L1I_LFB_HIT_RD_FPRFM</td>
<td>Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by software preload.</td>
</tr>
<tr>
<td>0x8251</td>
<td>L2I_LFB_HIT_RD_FPRFM</td>
<td>Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by software preload.</td>
</tr>
<tr>
<td>0x8254</td>
<td>L1D_LFB_HIT_RD_FPRFM</td>
<td>Level 1 data cache demand line-fill buffer first hit, read, recently fetched by software preload.</td>
</tr>
<tr>
<td>0x8255</td>
<td>L2D_LFB_HIT_RD_FPRFM</td>
<td>Level 2 data cache demand line-fill buffer first hit, read, recently fetched by software preload.</td>
</tr>
<tr>
<td>0x8256</td>
<td>L3D_LFB_HIT_RD_FPRFM</td>
<td>Level 3 data cache demand line-fill buffer first hit, read, recently fetched by software preload.</td>
</tr>
<tr>
<td>0x8257</td>
<td>LL_LFB_HIT_RD_FPRFM</td>
<td>Last level cache demand line-fill buffer first hit, read, recently fetched by software preload.</td>
</tr>
<tr>
<td>0x8258</td>
<td>L1D_LFB_HIT_WR_FPRFM</td>
<td>Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by software preload.</td>
</tr>
<tr>
<td>0x8259</td>
<td>L2D_LFB_HIT_WR_FPRFM</td>
<td>Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by software preload.</td>
</tr>
<tr>
<td>0x825A</td>
<td>L3D_LFB_HIT_WR_FPRFM</td>
<td>Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by software preload.</td>
</tr>
<tr>
<td>0x825B</td>
<td>LL_LFB_HIT_WR_FPRFM</td>
<td>Last level cache demand access line-fill buffer first hit, write, recently fetched by software preload.</td>
</tr>
<tr>
<td>0x825C</td>
<td>L1D_LFB_HIT_RW_FPRFM</td>
<td>Level 1 data cache demand access line-fill buffer first hit, recently fetched by software preload.</td>
</tr>
<tr>
<td>0x825D</td>
<td>L2D_LFB_HIT_RW_FPRFM</td>
<td>Level 2 data cache demand access line-fill buffer first hit, recently fetched by software preload.</td>
</tr>
<tr>
<td>0x825E</td>
<td>L3D_LFB_HIT_RW_FPRFM</td>
<td>Level 3 data cache demand access line-fill buffer first hit, recently fetched by software preload.</td>
</tr>
<tr>
<td>0x825F</td>
<td>LL_LFB_HIT_RW_FPRFM</td>
<td>Last level cache demand access line-fill buffer first hit, recently fetched by software preload.</td>
</tr>
<tr>
<td>0x8260</td>
<td>L1I_LFB_HIT_RD_FHWPRF</td>
<td>Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x8261</td>
<td>L2I_LFB_HIT_RD_FHWPRF</td>
<td>Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x8264</td>
<td>L1D_LFB_HIT_RD_FHWPRF</td>
<td>Level 1 data cache demand line-fill buffer first hit, read, recently fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x8265</td>
<td>L2D_LFB_HIT_RD_FHWPRF</td>
<td>Level 2 data cache demand line-fill buffer first hit, read, recently fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x8266</td>
<td>L3D_LFB_HIT_RD_FHWPRF</td>
<td>Level 3 data cache demand line-fill buffer first hit, read, recently fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>Event number</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x8267</td>
<td>LL_LFB_HIT_RD_FHWPRF</td>
<td>Last level cache demand line-fill buffer first hit, read, recently fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x8268</td>
<td>L1D_LFB_HIT_WR_FHWPRF</td>
<td>Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x8269</td>
<td>L2D_LFB_HIT_WR_FHWPRF</td>
<td>Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x826A</td>
<td>L3D_LFB_HIT_WR_FHWPRF</td>
<td>Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x826B</td>
<td>LL_LFB_HIT_WR_FHWPRF</td>
<td>Last level cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x826C</td>
<td>L1D_LFB_HIT_RW_FHWPRF</td>
<td>Level 1 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x826D</td>
<td>L2D_LFB_HIT_RW_FHWPRF</td>
<td>Level 2 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x826E</td>
<td>L3D_LFB_HIT_RW_FHWPRF</td>
<td>Level 3 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x826F</td>
<td>LL_LFB_HIT_RW_FHWPRF</td>
<td>Last level cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher.</td>
</tr>
<tr>
<td>0x8270</td>
<td>L1I_LFB_HIT_RD_FPRF</td>
<td>Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x8271</td>
<td>L2I_LFB_HIT_RD_FPRF</td>
<td>Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x8274</td>
<td>L1D_LFB_HIT_RD_FPRF</td>
<td>Level 1 data cache demand line-fill buffer first hit, read, recently fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x8275</td>
<td>L2D_LFB_HIT_RD_FPRF</td>
<td>Level 2 data cache demand line-fill buffer first hit, read, recently fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x8276</td>
<td>L3D_LFB_HIT_RD_FPRF</td>
<td>Level 3 data cache demand line-fill buffer first hit, read, recently fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x8277</td>
<td>LL_LFB_HIT_RD_FPRF</td>
<td>Last level cache demand line-fill buffer first hit, read, recently fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x8278</td>
<td>L1D_LFB_HIT_WR_FPRF</td>
<td>Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x8279</td>
<td>L2D_LFB_HIT_WR_FPRF</td>
<td>Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x827A</td>
<td>L3D_LFB_HIT_WR_FPRF</td>
<td>Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x827B</td>
<td>LL_LFB_HIT_WR_FPRF</td>
<td>Last level cache demand access line-fill buffer first hit, write, recently fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x827C</td>
<td>L1D_LFB_HIT_RW_FPRF</td>
<td>Level 1 data cache demand access line-fill buffer first hit, recently fetched by preload or prefetch.</td>
</tr>
</tbody>
</table>
Table D11-8 Event index (continued)

<table>
<thead>
<tr>
<th>Event number</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x827D</td>
<td>L2D_LFB_HIT_RW_FPRF</td>
<td>Level 2 data cache demand access line-fill buffer first hit, recently fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x827E</td>
<td>L3D_LFB_HIT_RW_FPRF</td>
<td>Level 3 data cache demand access line-fill buffer first hit, recently fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x827F</td>
<td>LL_LFB_HIT_RW_FPRF</td>
<td>Last level cache demand access line-fill buffer first hit, recently fetched by preload or prefetch.</td>
</tr>
<tr>
<td>0x8280</td>
<td>L1I_CACHE_PRF</td>
<td>Level 1 instruction cache, preload or prefetch hit.</td>
</tr>
<tr>
<td>0x8281</td>
<td>L2I_CACHE_PRF</td>
<td>Level 2 instruction cache, preload or prefetch hit.</td>
</tr>
<tr>
<td>0x8284</td>
<td>L1D_CACHE_PRF</td>
<td>Level 1 data cache, preload or prefetch hit.</td>
</tr>
<tr>
<td>0x8285</td>
<td>L2D_CACHE_PRF</td>
<td>Level 2 data cache, preload or prefetch hit.</td>
</tr>
<tr>
<td>0x8286</td>
<td>L3D_CACHE_PRF</td>
<td>Level 3 data cache, preload or prefetch hit.</td>
</tr>
<tr>
<td>0x8287</td>
<td>LL_CACHE_PRF</td>
<td>Last level cache, preload or prefetch hit.</td>
</tr>
<tr>
<td>0x8288</td>
<td>L1I_CACHE_REFILL_PRF</td>
<td>Level 1 instruction cache refill, preload or prefetch hit.</td>
</tr>
<tr>
<td>0x8289</td>
<td>L2I_CACHE_REFILL_PRF</td>
<td>Level 2 instruction cache refill, preload or prefetch hit.</td>
</tr>
<tr>
<td>0x828C</td>
<td>L1D_CACHE_REFILL_PRF</td>
<td>Level 1 data cache refill, preload or prefetch hit.</td>
</tr>
<tr>
<td>0x828D</td>
<td>L2D_CACHE_REFILL_PRF</td>
<td>Level 2 data cache refill, preload or prefetch hit.</td>
</tr>
<tr>
<td>0x828E</td>
<td>L3D_CACHE_REFILL_PRF</td>
<td>Level 3 data cache refill, preload or prefetch hit.</td>
</tr>
<tr>
<td>0x828F</td>
<td>LL_CACHE_REFILL_PRF</td>
<td>Last level cache refill, preload or prefetch hit.</td>
</tr>
<tr>
<td>0x8290</td>
<td>ISNP_HIT_PRF</td>
<td>Snoop hit, any instruction prefetch.</td>
</tr>
<tr>
<td>0x8291</td>
<td>ISNP_HIT_NEAR_PRF</td>
<td>Snoop hit in near cache, instruction preload or prefetch.</td>
</tr>
<tr>
<td>0x8292</td>
<td>ISNP_HIT_FAR_PRF</td>
<td>Snoop hit in far cache, instruction preload or prefetch.</td>
</tr>
<tr>
<td>0x8293</td>
<td>ISNP_HIT_REMOTE_PRF</td>
<td>Snoop hit in remote cache, instruction preload or prefetch.</td>
</tr>
<tr>
<td>0x8294</td>
<td>DSNP_HIT_PRF</td>
<td>Snoop hit, any data prefetch.</td>
</tr>
<tr>
<td>0x8295</td>
<td>DSNP_HIT_NEAR_PRF</td>
<td>Snoop hit in near cache, data preload or prefetch.</td>
</tr>
<tr>
<td>0x8296</td>
<td>DSNP_HIT_FAR_PRF</td>
<td>Snoop hit in far cache, data preload or prefetch.</td>
</tr>
<tr>
<td>0x8297</td>
<td>DSNP_HIT_REMOTE_PRF</td>
<td>Snoop hit in remote cache, data preload or prefetch.</td>
</tr>
<tr>
<td>0x8298</td>
<td>LL_CACHE_RW</td>
<td>Last level cache demand access.</td>
</tr>
<tr>
<td>0x8299</td>
<td>LL_CACHE_PRFM</td>
<td>Last level cache software preload.</td>
</tr>
<tr>
<td>0x829A</td>
<td>LL_CACHE_REFILL</td>
<td>Last level cache refill.</td>
</tr>
<tr>
<td>0x829B</td>
<td>LL_CACHE_REFILL_PRFM</td>
<td>Last level cache refill, software preload.</td>
</tr>
<tr>
<td>0x829C</td>
<td>LL_CACHE_WR</td>
<td>Last level cache write-back.</td>
</tr>
<tr>
<td>0x829D</td>
<td>LL_CACHE_WR</td>
<td>Last level cache access, write.</td>
</tr>
</tbody>
</table>
The event numbers of the common architectural and microarchitectural events are reserved for the specified events. Each of these event numbers must either:

- Be used for its assigned event.
- Not be used.

However, see *Required events* on page D11-5425.

When an implementation supports monitoring of an event that is assigned a common architectural or microarchitectural event number, Arm strongly recommends that it uses that number for the event. However, software might encounter implementations where an event assigned a number in this range is monitored using an event number from an IMPLEMENTATION DEFINED range.

**Note**

Arm might define other common architectural and microarchitectural event numbers. This is one reason why software must not assume that an event with an assigned common architectural or microarchitectural event number is never monitored using an event number from the IMPLEMENTATION DEFINED range.

It is IMPLEMENTATION DEFINED which events, including Common events, are generated by IMPLEMENTATION DEFINED extensions to the architecture, including accesses to IMPLEMENTATION DEFINED System registers and IMPLEMENTATION DEFINED System instructions. However, the functionality of the IMPLEMENTATION DEFINED extension must be appropriate for the generated events.

The common events are described in the following sections:

- *Common architectural events* on page D11-5307.
- *Common microarchitectural events* on page D11-5317.

The supported common architectural and microarchitectural events in the ranges 0x0000-0x003F and 0x4000-0x403F are discoverable to software through:

- The PMCEID0_EL0 and PMCEID1_EL0 registers in AArch64 state.
- The PMCEID0, PMCEID1, PMCEID2, and PMCEID3 registers in AArch32 state.

Arm recommends that the value of 0 is used for the PMCEID0_EL0 or PMCEID1_EL0 bit corresponding to any event that an implementation never generates, even if the implementation is considered to support but never count the event.

**Note**

- For example, if an implementation never generates the L1D_CACHE_ALLOCATE event, event 31, Arm recommends that PMCEID0_EL0[31] is RAZ.
- In an implementation that supports both Execution states, each bit in the AArch64 PMCEID0_EL0 and PMCEID1_EL0 registers corresponds to a single bit in the AArch32 PMCEID0, PMCEID1, PMCEID2, and PMCEID3 registers, and corresponding bits must have the same behavior.
However, for some implementations, an event in the common events range might be generated by the system, meaning behavior can vary between systems. In such a case, the corresponding PMCEIDn_EL0 bit might be RAO.

Event numbers that Table D11-7 on page D11-5275 shows as allocated for common architectural and microarchitectural events that are not described in Common architectural events and Common microarchitectural events on page D11-5317 are reserved. Future revisions of this Manual, or of the architecture, might assign these reserved values to additional common events. Events that do not require additional features in the PMU can be implemented retrospectively, meaning an implementation of a particular version of the PMU specification might support common events that are first defined in a later version of the PMU specification.

--- Note ---

The requirement that an event that is implemented retrospectively does not require additional features in the PMU means that it must be possible to represent the event $n$ the PMEVTPER<$n$>_EL0.evtCount field. This means, for example, that an implementation with a 10-bit PMEVTPER<$n$>_EL0.evtCount field can only implement events with event numbers $0x0000$-$0x03FF$.

**Common architectural events**

The Common architectural events are:

$0x0000$, **SW_INCR, Instruction architecturally executed, Condition code check pass, software increment**

The counter counts each write to the PMSWINC_EL0 and PMSWINC register, for each implemented event counter $<n>$:

- In AArch64 state, if PMEVTPER<$n$>_EL0.evtCount is $0x0000$ then the counter counts each MSR write to PMSWINC_EL0 with bit $[n]$ set to 1.
- In AArch32 state, if PMEVTPERN.evtCount is $0x0000$ then the counter counts each MCR write to PMSWINC with bit $[n]$ set to 1.

If the PE performs two architecturally executed writes to the PMSWINC_EL0 or PMSWINC register without an intervening Context synchronization event, then the counter is incremented twice.

--- Note ---

In a multithreaded implementation, if the Effective value of PMEVTPER<$n$>_EL0.MT is 1, then the counter counts writes by all PEs that have the same affinity at level 1 and above.

PMCEID0_EL0[0] reads as $0b1$ if this event is implemented and $0b0$ otherwise.
This event must be implemented.

$0x0006$, **LD_RETIRED, Instruction architecturally executed, Condition code check pass, load**

The counter counts each architecturally-executed memory-reading instruction.

--- Note ---

This event $0x006$ does not count the return status value of a Store-Exclusive instruction.

Whether the preload instructions PREM, PLD, PLDW, PLI, count as memory-reading instructions is IMPLEMENTATION DEFINED. Arm recommends that if the instruction is not implemented as a NOP then it is counted as a memory-reading instruction.

PMCEID0_EL0[6] reads as $0b1$ if this event is implemented and $0b0$ otherwise.

$0x0007$, **ST_RETIRED, Instruction architecturally executed, Condition code check pass, store**

The counter counts each architecturally-executed memory-writing instruction.
DC ZVA is counted as a store.

The counter does not count a Store-Exclusive instruction that fails.

PMCEID0_EL0[7] reads as $0b1$ if this event is implemented and $0b0$ otherwise.
0x0008, INST_RETIRED, Instruction architecturally executed

The counter counts each architecturally-executed instruction.

It is IMPLEMENTATION DEFINED whether the counter counts architecturally-executed MOVPRFX instructions.

PMCEID0_EL0[8] reads as 0b1 if this event is implemented and 0b0 otherwise.

This event must be implemented if event INST_SPEC is not implemented.

0x0009, EXC_TAKEN, Exception taken

The counter counts each exception taken.

--- Note ---

The counter counts the PE exceptions described in:

- For exceptions taken to an Exception level using AArch64, Exception entry on page D1-4641.
- For exceptions taken to an Exception level using AArch32, AArch32 state exception descriptions on page G1-8964.

PMCEID0_EL0[9] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x000A, EXC_RETURN, Instruction architecturally executed, Condition code check pass, exception return

The counter counts each architecturally-executed exception return instruction. The following sections define the counted instructions:

- For an exception return from an Exception level using AArch64, Exception return on page D1-4644.
- For an exception return from an Exception level using AArch32, Exception return to an Exception level using AArch32 on page G1-8951.

However, it is CONSTRAINED UNPREDICTABLE whether this event counts the execution of an exception return instruction if either:

- Execution of the instruction is, itself, CONSTRAINED UNPREDICTABLE.
- Execution of the instruction sets PSTATE.IL and does not generate an exception return.

--- Note ---

Examples of when an exception return instruction is CONSTRAINED UNPREDICTABLE are if the instruction is executed at EL0, or in AArch32 state in System mode.

A particular consequence of this CONSTRAINED UNPREDICTABLE behavior is that an implementation that does not support AArch32 state at EL1 or higher does not have to treat AArch32 MOV PS, LR instructions, and related instructions, as exception return instructions.

PMCEID0_EL0[10] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x000B, CID_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to CONTEXTIDR

The counter counts each MSR write to CONTEXTIDR_EL1 and each MCR write to CONTEXTIDR.

If the PE performs two architecturally-executed writes to CONTEXTIDR without an intervening Context synchronization event, it is CONSTRAINED UNPREDICTABLE whether the first write is counted.

When FEAT_VHE is implemented, the counter:

- Counts each architecturally-executed instruction accessing the named register CONTEXTIDR_EL1, including when executing at EL2 when HCR_EL2.E2H is 0b1.
- Does not count instructions accessing the named register CONTEXTIDR_EL12.
The event is defined by the name used to access the register. The counter does not count writes to the named register CONTEXTIDR_EL2.

PMCEID0_EL0[11] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x000C, PC_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, Software change of the PC
The counter counts each architecturally-executed Software change of the PC that is taken. This includes all:
• Branch instructions.
• Memory-reading instructions that explicitly write to the PC.
• Data-processing instructions that explicitly write to the PC.
• Exception return instructions.

Conditional branches are only counted if the branch is taken.

It is IMPLEMENTATION DEFINED whether the following are included as a Software change of the PC:
• BRK and BKPT instructions.
• An UNDEFINED instruction that generates an exception.
• An exception-generating instruction, SVC, HVC, and SMC, that generates an exception.
• Context synchronization barrier instructions, ISB.

The counter does not increment for exceptions other than those explicitly identified in these lists. If PC_WRITE_RETIRED and BR_SKIP_RETIRED are both implemented, the PE must treat the following types of instruction in the same way for both events:
• BRK and BKPT instructions.
• UNDEFINED instructions.
• The exception-generating instructions, SVC, HVC, and SMC.
• Context synchronization barrier instructions.

From Armv8.6, if BR_RETIRED is also implemented, the PE must also treat these types of instruction in the same way for the BR_RETIRED, PC_WRITE_RETIRED, and BR_SKIP_RETIRED events.

PMCEID0_EL0[12] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x000D, BR_IMMED_RETIRED, Branch Instruction architecturally executed, immediate
The counter counts all architecturally-executed immediate branch instructions.

In AArch32 state, the counter increments each time the PE executes one of the following instructions:
• B{<c> } <label>.
• BL{<c>} <label>.
• BLX{<c>} <label>.
• CBZ <Rn>, <label>.
• CBNZ <label>.

In AArch64 state, the counter increments each time the PE executes one of the following instructions:
• B <label>.
• B.cond <label>.
• BL <label>.
• CBZ <Rn>, <label>.
• CBNZ <Rn>, <label>.
• TBZ <Rn>, <label>.
• TBNZ <Rn>, <label>.

--- Note ---
Conditional branches are always counted, regardless of whether the branch is taken or not taken.

---
If the Context synchronization barrier instruction ISB is counted as a Software change of the PC instruction by PC_WRITE RETIRED, then it is IMPLEMENTATION DEFINED whether ISB is counted as an immediate branch instruction.

PMCEID0_EL0[13] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x000E, BR_RETURN RETIRED, Branch Instruction architecturally executed, procedure return, taken
In AArch32 state, the counter counts the following architecturally executed procedure return instructions:
• BX R14.
• MOV PC, LR.
• POP {..., PC}.
• LDR PC, [SP], #offset.

--- Note ---
The counter counts only the listed instructions as procedure returns. For example, it does not count the following as procedure return instructions:
• BX R0, because Rm ≠ R14.
• MOV PC, R0, because Rm ≠ R14.
• LDM SP, {..., PC}, because writeback is not specified.
• LDR PC, [SP, #offset], because this specifies the wrong addressing mode.

---
In AArch64 state, the counter counts architecturally executed RET, RETAA, and RETAB instructions.
PMCEID0_EL0[14] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x000F, UNALIGNED_LDST RETIRED, Instruction architecturally executed, Condition code check pass, unaligned load or store
The counter counts each memory-reading instruction or memory-writing instruction access that would generate an Alignment fault when Alignment fault checking is enabled.
The counter does not count accesses that would generate an SP alignment fault exception if the applicable stack pointer alignment check is enabled, unless that access would also generate an Alignment fault Data Abort exception if Alignment fault checking is enabled.
It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether this event counts accesses that generate an exception, including accesses that do generate Alignment fault Data Abort exceptions.
PMCEID0_EL0[15] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x001C, TTBR_WRITE RETIRED, Instruction architecturally executed, Condition code check pass, write to TTBR
The counter counts MSR writes to TTBR0_EL1 and TTBR1_EL1 in AArch64 state and MCR and MCRR writes to TTBR0 and TTBR1 in AArch32 state. When EL3 is implemented and using AArch32, this includes counting writes to both banked copies of TTBR0 and TTBR1.
If the PE executes two writes to the same TTBR, without an intervening Context synchronization event, it is CONSTRAINED UNPREDICTABLE whether the first write to the TTBR, is counted.
If EL3 is implemented and using AArch64, the counter does not count writes to TTBR0_EL3.
If EL2 is implemented and using AArch64, the counter does not count writes to TTBR0_EL2 and VTTBR_EL2.

If EL2 is implemented and using AArch32, the counter does not count writes to HTTBR and VTTBR.

When FEAT_VHE is implemented, the counter:
• Counts each architecturally-executed instruction accessing the named registers TTBR0_EL1 and TTBR1_EL1, including when executing at EL2 when HCR_EL2.E2H is 0b1.
• Does not count instructions accessing the named registers TTBR0_EL12 and TTBR1_EL12.

PMCEID0_EL0[28] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x001E, CHAIN, Chain a pair of event counters

Even-numbered counters never increment as a result of this event.
For an odd-numbered counter <n+1>, the counter increments when an event increments the preceding even-numbered counter <n> on the same PE causing unsigned overflow of bits [31:0] of the event counter <n>, and any of the following are true:
• FEAT_PMUv3p5 is not implemented.
• EL2 is not implemented and PMCR.LP is 0b1.
• EL2 is implemented, <n> is less than HDCR.HPMN, and PMCR.LP is 0b0.
• EL2 is implemented, <n> is greater than or equal to HDCR.HPMN, and HDCR.HLP is 0b0.
This means the CHAIN event can be used to link the odd-numbered counter with the preceding even-numbered counter to provide a 64-bit counter.

Note

When FEAT_PMUv3p5 is not implemented, the CHAIN event allows software to use the N event counters as N 32-bit counters, N/2 64-bit counters, or a mixture of 32-bit counters and 64-bit counters.
The CHAIN event only counts overflows from the preceding even-numbered counter on the same PE. This means it ignores the Effective value of PMEVTYPER<n>_EL0.MT.
The architecture does not provide atomic access to a pair of counters.

To filter the Exception levels and Security states in which the event is counted, software:
• Programs PMEVTYPER<n>_EL0 to count the event in the required conditions.
• Programs PMEVTYPER<n+1>_EL0 to count the CHAIN event in all Exception levels and states.
The PE might ignore the filter settings for the CHAIN event and behave as if they are set to count in all Exception levels and states. If software does not program the event in this way, the count becomes UNPREDICTABLE.
The architecture does not define the latency between the low counter overflowing and the high counter incrementing the CHAIN event. There is no requirement for updates to occur synchronously, but software reading or enabling the counter pair by reading the low counter first and the high counter second, with an intervening Context synchronization event, will not observe the low counter incrementing and overflowing for the event and the high counter not incrementing for the resulting CHAIN event. This means that the ISB executed after reading the low counter ensures the completion of the update of the high counter by the CHAIN event.
PMCEID0_EL0[30] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0021, BR_RETIRED, Instruction architecturally executed, branch

The counter counts all branches on the architecturally executed path that would incur cost if mispredicted.

It is IMPLEMENTATION DEFINED whether the counter increments for any or all of:
• Unconditional direct branch instructions.
The Performance Monitors Extension

D11.11 PMU events and event numbers

- Exception-generating instructions.
- Exception return instructions.
- Context synchronization instructions.

The counter counts all other branch instructions, memory-reading and data-processing instructions that explicitly write to the PC, at retirement.

Note

Conditional branches are always counted, regardless of whether the branch is taken.

Arm recommends that BR_RETIRED counts Unconditional direct branch instructions and Exception return instructions.

From Armv8.6, if PC_WRITE RETIRED and BR_RETIRED are both implemented, the PE must treat the following types of instruction in the same way for both events:
- BRK and BKPT instructions.
- UNDEFINED instructions.
- The exception-generating instructions, SVC, HVC, and SMC.
- Context synchronization barrier instructions.

PMCEID1_EL0[1] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x400C, TRB_WRAP, Trace buffer current write pointer wrapped

The event is generated each time the current write pointer is wrapped to the base pointer.

PMCEID0_EL0[44] reads as 0b1 if this event is implemented and 0b0 otherwise.

This event must be implemented if FEAT_TRBE is implemented.

0x400D, PMU_OVFS, PMU overflow, counters accessible to EL1 and EL0

The event is generated each time one of the following occurs:
- An event is counted by an event counter <n> and all of the following are true:
  - PMINTENSET_EL1[n] is 0b1.
  - One of the following is true:
    - Counting the event causes unsigned overflow of PMEVCNTR<n>_EL0[31:0], and either FEAT_PMUv3p5 is not implemented or PMCR_EL0.LP is 0b0.
    - Counting the event causes unsigned overflow of PMEVCNTR<n>_EL0[63:0], FEAT_PMUv3p5 is implemented, and PMCR_EL0.LP is 0b1.
    - Either EL2 is implemented and <n> in the range [0 .. (MDCR_EL2.HPMN-1)], or EL2 is not implemented and <n> is in the range [0 .. (PMCR_EL0.N-1)].
- A cycle is counted by PMCCNTR_EL0, PMINTENSET_EL1[31] is 0b1, and one of the following is true:
  - Counting the cycle causes unsigned overflow of PMCCNTR_EL0[31:0] and PMCR_EL0.LC is 0b0.
  - Counting the cycle causes unsigned overflow of PMCCNTR_EL0[63:0] and PMCR_EL0.LC is 0b1.

This event cannot be counted by the PMU. PMCEID0_EL0[45] reads as 0b0.

This event must be implemented if all of the following are true:
- FEAT_PMUv3 is implemented.
- FEAT_ETE is implemented.

0x400E, TRB_TRIG, Trace buffer Trigger Event

The event is generated when a Trace Buffer Extension Trigger Event occurs.

It is IMPLEMENTATION DEFINED whether this event can be counted by the PMU.
PMCEID0_EL0[46] reads as 0b1 if this event is implemented and can be counted by the PMU, and 0b0 otherwise.

This event must be implemented if FEAT_TRBE is implemented.

\[0x400F\], **PMU_HOVFS, PMU overflow, counters reserved for use by EL2**

The event is generated each time an event is counted by an event counter \(<n>\) and all of the following are true:

- EL2 is implemented.
- PMINTENSET_EL1[\(n\)] is 0b1.
- One of the following is true:
  - Counting the event causes unsigned overflow of PMEVCNTR\(<n>_EL0[31:0]\), and either FEAT_PMUv3p5 is not implemented or MDCR_EL2.HLP is 0b0.
  - Counting the event causes unsigned overflow of PMEVCNTR\(<n>_EL0[63:0]\), FEAT_PMUv3p5 is implemented, and MDCR_EL2.HLP is 0b1.
- \(<n>\) in the range [MDCR_EL2.HPMN .. (PMCR_EL0.N-1)].

The event is not transmitted to a trace unit if SelfHostedTraceEnabled() is TRUE and TRFCR_EL2.E2TRE is 0b0.

--- Note ---

This is in addition to the rules for the export of all events to a trace unit. See Controls to prohibit trace at Exception levels on page D3-4759.

This event cannot be counted by the PMU. PMCEID0_EL0[47] reads as 0b0.

This event must be implemented if all of the following are true:

- FEAT_PMUv3 is implemented.
- FEAT_ETE is implemented.
- EL2 is implemented.

\[0x8000\], **SIMD_INST_RETIRED, Instruction architecturally executed, SIMD**

The counter counts each architecturally executed SIMD instruction.

That is, the counter counts:

- SVE instructions other than non-SIMD SVE instructions.
- Advanced SIMD instructions other than Advanced SIMD scalar instructions.

\[0x8001\], **ASE_INST_RETIRED, Instruction architecturally executed, Advanced SIMD**

The counter counts each architecturally executed Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED whether the counter counts Advanced SIMD scalar instructions.

\[0x8002\], **SVE_INST_RETIRED, Instruction architecturally executed, SVE**

The counter counts each architecturally executed SVE instruction.

It is IMPLEMENTATION DEFINED whether the counter counts non-SIMD SVE instructions.

This event must be implemented if event SVE_INST_SPEC is not implemented.

Implementation of this event requires that FEAT_SVE is implemented.

\[0x8003\], **ASE_SVE_INST_RETIRED, Instruction architecturally executed, Advanced SIMD or SVE**

The counter counts each architecturally executed instruction counted by either ASE_INST_RETIRED or SVE_INST_RETIRED.
0x8107, **BR_SKIP_RETIRED, Branch Instruction architecturally executed, not taken**

The counter counts each conditional **Software change of the PC** instruction that is not taken. This includes all:

- Conditional branch instructions.
- Conditional memory-reading instructions that explicitly write to the PC.
- Conditional data-processing instructions that explicitly write to the PC.
- Conditional exception return instructions.

These are the same instructions which, if unconditional, or conditional and taken, are counted by the **PC_WRITE_RETIRED** event.

It is **IMPLEMENTATION DEFINED** whether the following are included as a **Software change of the PC**, which the counter increments for:

- A conditional **UNDEFINED** instruction that does not generate an exception.
- A conditional exception-generating instruction, **SVC**, **HVC**, or **SMC**, that does not generate an exception.
- A conditional Context synchronization barrier instruction, **ISB**, that fails its condition code check.

**Note**

Many of these instructions can only be conditional in the AArch32 instruction sets.

The counter does not increment for exceptions other than those explicitly identified in these lists. If **PC_WRITE_RETIRED** and **BR_SKIP_RETIRED** are both implemented, the PE must treat the following types of instruction in the same way for both events:

- **BRK** and **BKPT** instructions.
- **UNDEFINED** instructions.
- The exception-generating instructions, **SVC**, **HVC**, and **SMC**.
- Context synchronization barrier instructions.

From Armv8.6, if **BR_RETIRED** is also implemented, the PE must also treat these types of instruction in the same way for the **BR_RETIRED** event.

0x8108, **BR_IMMED_TAKEN_RETIRED, Branch Instruction architecturally executed, immediate, taken**

The counter counts each instruction counted by both **BR_IMMED_RETIRED** and **PC_WRITE_RETIRED**.

These are all direct branch instructions on the architecturally executed path, where the branch is taken.

0x8109, **BR_IMMED_SKIP_RETIRED, Branch Instruction architecturally executed, immediate, not taken**

The counter counts each instruction counted by both **BR_IMMED_RETIRED** and **BR_SKIP_RETIRED**.

These are all direct branch instructions on the architecturally executed path, where the branch is not taken.

Implementation of this optional event requires that AArch32 is implemented at any Exception level.

0x810A, **BR_IND_TAKEN_RETIRED, Branch Instruction architecturally executed, indirect, taken**

The counter counts each instruction counted by both **BR_IND_RETIRED** and **PC_WRITE_RETIRED**.

These are all indirect branch instructions, including return instructions, on the architecturally executed path, where the branch is taken.

Implementation of this optional event requires that AArch32 is implemented at any Exception level.
0x8108, **BR_IND_SKIP_RETIRED**, Branch Instruction architecturally executed, indirect, not taken

The counter counts each instruction counted by both **BR_IND_RETIRED** and **BR_SKIP_RETIRED**.

These are all indirect branch instructions, including return instructions, on the architecturally executed path, where the branch is not taken.

Implementation of this optional event requires that AArch32 is implemented at any Exception level.

0x810C, **BR_INDNR_TAKEN_RETIRED**, Branch Instruction architecturally executed, indirect excluding procedure return, taken

The counter counts each instruction counted by both **BR_IND_RETIRED** and **PC_WRITE_RETIRED**, that is not counted by **BR_RETURN_RETIRED**.

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path, where the branch is taken.

0x810D, **BR_INDNR_SKIP_RETIRED**, Branch Instruction architecturally executed, indirect excluding procedure return, not taken

The counter counts each instruction counted by both **BR_INDNR_RETIRED** and **BR_SKIP_RETIRED**.

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path, where the branch is not taken.

Implementation of this optional event requires that AArch32 is implemented at any Exception level.

0x810E, **BR_RETURN_ANY_RETIRED**, Branch Instruction architecturally executed, procedure return

The counter counts each instruction counted by **BR_IND_RETIRED** where if taken, the branch would be counted by **BR_RETURN_RETIRED**.

These are all return instructions on the architecturally executed path.

Implementation of this optional event requires that AArch32 is implemented at any Exception level.

0x810F, **BR_RETURN_SKIP_RETIRED**, Branch Instruction architecturally executed, procedure return, not taken

The counter counts each instruction counted by both **BR_RETURN_ANY_RETIRED** and **BR_SKIP_RETIRED**.

These are all return instructions on the architecturally executed path, where the branch is not taken.

Implementation of this optional event requires that AArch32 is implemented at any Exception level.

0x8110, **BR_IND_RETIRED**, Instruction architecturally executed, indirect branch

The counter counts each Software change of the PC that is not counted by **BR_IMMED_RETIRED**.

Software change of the PC has the same definition as for the **PC_WRITE_RETIRED** and **BR_SKIP_RETIRED** events.

--- Note

Conditional branches are always counted, regardless of whether the branch is taken.

---

0x811E, **BR_INDNR_RETIRED**, Branch Instruction architecturally executed, indirect excluding procedure return

The counter counts each instruction counted by **BR_IND_RETIRED** that is not counted by **BR_RETURN_ANY_RETIRED**.

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path.

Implementation of this optional event requires that AArch32 is implemented at any Exception level.
0x811F, BRB_FILTRATE, Branch Record captured
The counter counts each branch record captured in the branch record buffer. Branch records that are not captured because they are removed by filtering are not counted.
When BRB_FILTRATE is generated for an exception or an exception return, it is an Exception-related event.
This event must be implemented if FEAT_BRBE is implemented.

0x8179, BRNL_INDNR_TAKEN_RETIRED, Branch Instruction architecturally executed, indirect branch without link excluding procedure return, taken
The counter counts each branch counted by BRNL_IND_TAKEN_RETIRED that is not counted by BR_RETURN_RETIRED.
These are all indirect branch without link excluding procedure return instructions on the architecturally executed path, where the branch is taken.

0x817A, BL_TAKEN_RETIRED, Branch Instruction architecturally executed, branch with link, taken
The counter counts each Software change of the PC counted by PC_WRITE_RETIRED that calls a subroutine and sets LR to return address.
These are all branch with link instructions on the architecturally executed path, where the branch is taken.

0x817B, BRNL_TAKEN_RETIRED, Branch Instruction architecturally executed, branch without link, taken
The counter counts each Software change of the PC counted by PC_WRITE_RETIRED that is not counted by BL_TAKEN_RETIRED.
These are all branch without link instructions on the architecturally executed path, where the branch is taken.

0x817C, BRNL_IND_TAKEN_RETIRED, Branch Instruction architecturally executed, indirect branch without link, taken
The counter counts each branch counted by both BRNL_TAKEN_RETIRED and BR_IND_TAKEN_RETIRED.
These are all indirect branch without link instructions on the architecturally executed path, where the branch is taken.

0x817D, BRNL_IND_TAKEN_RETIRED, Branch Instruction architecturally executed, indirect branch without link, taken
The counter counts each branch counted by both BRNL_TAKEN_RETIRED and BR_IND_TAKEN_RETIRED.
These are all indirect branch without link instructions on the architecturally executed path, where the branch is taken.

0x817E, BL_IMMED_TAKEN_RETIRED, Branch Instruction architecturally executed, direct branch with link, taken
The counter counts each branch counted by both BL_TAKEN_RETIRED and BR_IMMED_TAKEN_RETIRED.
These are all direct branch with link instructions on the architecturally executed path, where the branch is taken.

0x817F, BRNL_IMMED_TAKEN_RETIRED, Branch Instruction architecturally executed, direct branch without link, taken
The counter counts each branch counted by both BRNL_TAKEN_RETIRED and BR_IMMED_TAKEN_RETIRED.
These are all direct branch without link instructions on the architecturally executed path, where the branch is taken.
**0x8180, BR_UNCOND_RETIRED, Branch Instruction architecturally executed, unconditional branch**

The counter counts each Software change of the PC counted by BR_RETIRED that is not counted by BR_COND_RETIRED.

These are all unconditional branch instructions on the architecturally executed path.

**0x8181, BR_COND_RETIRED, Branch Instruction architecturally executed, conditional branch**

The counter counts each Software change of the PC counted by BR_RETIRED that is a conditional branch.

These are all conditional branch instructions on the architecturally executed path.

In the A64 instruction set, conditional branches are B.cond, BC.cond, CBNZ, CBZ, TBNZ, and T8Z. This includes B.cond and BC.cond instructions with the AL or W condition code.

In the A32 instruction set, conditional branches are Software change of the PC instructions that have bits [31:29] of the instruction opcode not equal to 0b111.

In the T32 instruction set, conditional branches are B<<>, B<<>.W, Software change of the PC instructions executed in an IT block, CBNZ, and CBZ. Branch instruction encodings, including B{<<>}, W, where << is either omitted or AL are only considered conditional branches when they are the last instruction in an IT block.

**0x8182, BR_COND_TAKEN_RETIRED, Branch Instruction architecturally executed, conditional branch, taken**

The counter counts each Software change of the PC counted by both BR_COND_RETIRED and PC_WRITE_RETIRED.

These are all conditional branch instructions on the architecturally executed path, where the branch is taken.

**0x8183, BR_HINT_COND_RETIRED, Branch Instruction architecturally executed, hinted conditional**

The counter counts each branch counted by BR_COND_RETIRED that is a hinted conditional branch.

These are all hinted conditional branch instructions on the architecturally executed path.

The hinted conditional branch instruction is BC.cond.

Implementation of this optional event requires that FEAT_HBC is implemented.

**Common microarchitectural events**

The Common microarchitectural events are:

**0x0001, L1I_CACHE_REFILL, Level 1 instruction cache refill**

The counter counts each access counted by L1I_CACHE that causes a refill of the Level 1 instruction or unified cache from outside of the Level 1 instruction or unified cache.

A refill includes any access that causes data to be fetched from outside of the Level 1 caches, even if the data is ultimately not allocated into the Level 1 instruction cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts accesses to the Level 1 instruction cache that cause a refill that is satisfied by fetching data from memory, a Level 2 cache, or a Level 1 cache of another PE.

The counter does not count accesses that miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

PMCEID0_EL0[1] reads as 0b1 if this event is implemented and 0b0 otherwise.
0x0002, **L1I_TLB_REFILL, Level 1 instruction TLB refill**

The counter counts each Instruction memory access counted by L1I_TLB that causes a TLB refill of the Level 1 instruction TLB.

This includes each Instruction memory access that causes a memory access due to a translation table walk or an access to another level of TLB caching.

The counter does not count the access if any of the following are true:

- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.
- The access is due to a TLB maintenance instruction.
- The access misses in the TLB and generates a translation table walk but does not cause a refill of the TLB.

It is IMPLEMENTATION DEFINED whether the counter counts the access if any of the following are true:

- The access generates a Translation fault for any other reason.
- The refill is not allocated in the TLB.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

PMCEID0_EL0[2] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0003, **L1D_CACHE_REFILL, Level 1 data cache refill**

The counter counts each access counted by L1D_CACHE that causes a refill of the Level 1 data or unified cache from outside of the Level 1 data or unified cache.

A refill includes any access that causes data to be fetched from outside of the Level 1 caches, even if the data is ultimately not allocated into the Level 1 data cache. Example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts accesses to the Level 1 data cache that cause a refill that is satisfied by fetching data from memory, a Level 2 cache, or a Level 1 cache of another PE.

The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

PMCEID0_EL0[3] reads as 0b1 if this event is implemented and 0b0 otherwise.

This event must be implemented if the implementation includes a Level 1 data or unified cache.

0x0004, **L1D_CACHE, Level 1 data cache access**

The counter counts each Memory-read operation or Memory-write operation that causes a cache access to at least the Level 1 data or unified cache.

Each access to a cache line is counted including the multiple accesses caused by single instructions such as LDM or STM. Each access to other Level 1 data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.
If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

An example of cache status information is whether the cached data is held in an exclusive or shared state.

When the L1D_CACHE_RW event is implemented:

- If the L1D_CACHE_PRFM event is implemented, accesses to the Level 1 data cache due to a preload or prefetch instruction are counted. Otherwise, these are not counted.
- If the L1D_CACHE_HWPRF event is implemented, accesses to the Level 1 data cache due to a hardware prefetcher are counted. Otherwise these events are not counted.

When the L1D_CACHE_RW event is not implemented, it is IMPLEMENTATION DEFINED whether accesses to the Level 1 data cache due to preload or prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

PMCEID0_EL0[4] reads as 0b1 if this event is implemented and 0b0 otherwise.

This event must be implemented if the implementation includes a Level 1 data or unified cache.

0x0005, L1D_TLB_REFILL, Level 1 data TLB refill

The counter counts each access counted by L1D_TLB that causes a TLB refill of the Level 1 data or unified TLB.

This includes each access that causes a memory access due to a translation table walk or an access to another level of TLB caching.

The counter does not count the access if any of the following are true:

- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 0b1.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.
- The access is due to a TLB maintenance instruction.
- The access misses in the TLB and generates a translation table walk but does not cause a refill of the TLB.

It is IMPLEMENTATION DEFINED whether the counter counts the access if any of the following are true:

- The access generates a Translation fault for any other reason.
- The refill is not allocated in the TLB.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

PMCEID0_EL0[5] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0010, BR_MIS_PRED, Branch instruction Speculatively executed, mispredicted or not predicted

The counter counts each correction to the predicted program flow that occurs because of a misprediction from, or no prediction from, the branch prediction resources and that relates to instructions that the branch prediction resources are capable of predicting.

If no program-flow prediction resources are implemented, Arm recommends that the counter counts all branches that are not taken.

PMCEID0_EL0[16] reads as 0b1 if this event is implemented and 0b0 otherwise.
This event must be implemented if the implementation includes program-flow prediction.

0x0011, **CPU_CYCLES, Cycle**

The counter increments on every cycle.

All counters are subject to changes in clock frequency, including when a \texttt{WFI} or \texttt{WFE} instruction stops the clock. This means that it is \texttt{CONSTRAINED UNPREDICTABLE} whether or not CPU_CYCLES continues to increment when the clocks are stopped by \texttt{WFI} and \texttt{WFE} instructions.

---

Note

Unlike PMCCNTR or PMCCNTR_EL0, this count is not affected by the cycle counter controls:

- The counter is not incremented in prohibited regions, so is not affected by PMCR.DP or PMCR_EL0.DP.
- The counter increments on every cycle, regardless of the setting of PMCR.D or PMCR_EL0.D.
- The counter is reset when event counters are reset by PMCR.P and PMCR_EL0.P, never by PMCR.C or PMCR_EL0.C.
- If \texttt{FEAT\_PMUv3p5} is implemented, the counter ignores SDCR.SCCD, MDCR_EL3.SCCD, HDCR.HCCD, and MDCR_EL2.HDCCD.

---

In a multithreaded implementation:

- If the Effective value of \texttt{PMEVTYPER<\texttt{n}\_EL0\_MT} for the event counter is 0, then the counter counts each Processor cycle when the counting PE is active.
- If the Effective value of \texttt{PMEVTYPER<\texttt{n}\_EL0\_MT} for the event counter is 1, then the counter counts each Processor cycle.

PMCEID0_EL0[17] reads as 0b1 if this event is implemented and 0b0 otherwise.

This event must be implemented.

0x0012, **BR_PRED, Predictable branch instruction Speculatively executed**

The counter counts every branch or other change in the program flow that the branch prediction resources are capable of predicting.

If all branches are subject to prediction, for example a BTB or BTAC, then all branches are predictable branches.

If branches are decoded before the predictor, so that the branch prediction logic dynamically predicts only some branches, for example conditional and indirect branches, then it is \texttt{IMPLEMENTATION DEFINED} whether other branches are counted as predictable branches. Arm recommends that all branches are counted.

An implementation might include other structures that predict branches, such as a loop buffer that predicts short backwards direct branches as taken. Each execution of such a branch is a predictable branch. Terminating the loop might generate a misprediction event that is counted by \texttt{BR\_MIS\_PRED}.

If no program-flow prediction resources are implemented, this event is optional, but Arm recommends that \texttt{BR\_PRED} counts all branches.

PMCEID0_EL0[18] reads as 0b1 if this event is implemented and 0b0 otherwise.

This event must be implemented if the implementation includes program-flow prediction.

0x0013, **MEM_ACCESS, Data memory access**

The counter counts each Memory-read operation or Memory-write operation that the PE makes.

The counter increments whether the access results in an access to a Level 1 data or unified cache, a Level 2 data or unified cache, or neither of these.

The number of accesses generated by each instruction is \texttt{IMPLEMENTATION DEFINED}.

The counter does not count:

- Instruction memory accesses.
• Translation table walks.
• Write-back from any cache.
• Refilling of any cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

PMCEID0_EL0[19] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0014, L1I_CACHE, Level 1 instruction cache access

The counter counts each Instruction memory access to at least the Level 1 instruction or unified cache.

Each Instruction memory access to other Level 1 instruction or unified memory structures, such as refill buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

An example of cache status information is whether the cached data is held in an exclusive or shared state.

When the L1I_CACHE_RD event is implemented:
  • If the L1I_CACHE_PRFM event is implemented, accesses to the Level 1 instruction cache due to a preload or prefetch instruction are counted. Otherwise, these are not counted.
  • If the L1I_CACHE_HWPRF event is implemented, accesses to the Level 1 instruction cache due to a hardware prefetcher are counted. Otherwise these events are not counted.

When the L1I_CACHE_RD event is not implemented, it implements whether accesses to the Level 1 instruction cache due to preload or prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

PMCEID0_EL0[20] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0015, L1D_CACHE_WB, Level 1 data cache write-back

The counter counts each write-back of data from the Level 1 data or unified cache to outside of the Level 1 caches. For example:
  • A write-back of a dirty cache line to a Level 2 cache or memory.
  • A write-back of a recently fetched cache line that has not been allocated to the Level 1 data cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:
  • A transfer of a dirty cache line from the Level 1 data cache to outside of Level 1 caches made as a result of a coherency request.
  • Write-backs made as a result of cache maintenance instructions.

The counter does not count:
  • The invalidation of a cache line without any write-back to outside of the Level 1 caches or memory.
  • Writes that write through the Level 1 data cache to outside of the Level 1 caches.
A write-back is attributable to the agent that generated the request that caused the write-back. This might not be the same agent that caused the data being written back to be allocated into the cache.

An Unattributable write-back event occurs when a requestor outside of the PE makes a coherency request that results in write-back. If the cache is shared, then an Unattributable write-back event is not counted. If the cache is not shared, then the event is counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines software is streaming writes to memory and does not allocate lines to the cache, or by a DC ZVA operation.

PMCEID0_EL0[21] reads as \texttt{0b1} if this event is implemented and \texttt{0b0} otherwise.

\textbf{0x0016, L2D\_CACHE, Level 2 data cache access}

The counter counts each Memory-read operation or Memory-write operation that causes a cache access to at least the Level 2 data or unified cache.

Each access to a cache line is counted including refills of and write-backs from other caches. Each access to other Level 2 data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If \texttt{FEAT\_PMUv3p4} is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If \texttt{FEAT\_PMUv3p4} is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

For example, cache status information is whether the cached data is held in an exclusive or shared state.

When the \texttt{L2D\_CACHE\_RW} event is implemented:

- If the \texttt{L2D\_CACHE\_PRFM} event is implemented, accesses to the Level 2 data cache due to a preload or prefetch instruction are counted. Otherwise, these are not counted.
- If the \texttt{L2D\_CACHE\_HWPRF} event is implemented, accesses to the Level 2 data cache due to a hardware prefetcher are counted. Otherwise these events are not counted.

When the \texttt{L2D\_CACHE\_RW} event is not implemented, it is IMPLEMENTATION DEFINED whether accesses to the Level 2 data cache due to preload or prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of \texttt{PMEVTYPER<n>_EL0.MT} for the counter is \texttt{0b1}, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

PMCEID0_EL0[22] reads as \texttt{0b1} if this event is implemented and \texttt{0b0} otherwise.

\textbf{0x0017, L2D\_CACHE\_REFILL, Level 2 data cache refill}

The counter counts each access counted by \texttt{L2D\_CACHE} that causes a refill of the Level 2 data or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

A refill includes any access that causes data to be fetched from outside of the Level 1 and Level 2 caches, even if the data is ultimately not allocated into the Level 2 data cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Level 2 data cache that cause a refill that is satisfied by fetching data from memory, a Level 3 cache, or a Level 2 cache of another PE.
- Refills of and write-backs from any Level 1 data, instruction, or unified cache of this PE that cause a refill from outside of the Level 1 and Level 2 caches of this PE.
• Accesses to the Level 2 data cache that cause a refill of a Level 1 cache of this PE from outside of the Level 1 and Level 2 caches of this PE, even if there is no refill of the Level 2 data cache.

The counter does not count accesses that:
• Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
• Miss in the cache but do not generate a refill, such as a write through the cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPE<\n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

PMCEID0_EL0[23] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0018, L2D_CACHE_WB, Level 2 data cache write-back

The counter counts each write-back of data from the Level 2 data or unified cache to outside of the Level 1 and Level 2 caches. For example:
• A write-back of a dirty cache line to a Level 3 cache or memory.
• A write-back of a recently fetched cache line that has not been allocated to the Level 2 data cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:
• A transfer of a dirty cache line from the Level 2 data cache to outside of Level 1 and Level 2 caches made as a result of a coherency request.
• Write-backs made as a result of cache maintenance instructions.

The counter does not count:
• The invalidation of a cache line without any write-back to outside of the Level 1 and Level 2 caches or memory.
• Writes that write through the Level 2 data cache to outside of the Level 1 and Level 2 caches.
• Transfers of data from the Level 2 data cache to a Level 1 cache to satisfy a refill of the other cache.

A write-back is attributable to the agent that generated the request that caused the write-back. This might not be the same agent that caused the data being written back to be allocated into the cache.

An Unattributable write-back event occurs when a requestor outside of the PE makes a coherency request that results in write-back. If the cache is shared, then an Unattributable write-back event is not counted. If the cache is not shared, then the event is counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines software is streaming writes to memory and does not allocate lines to the cache, or by a DC ZVA operation.

PMCEID0_EL0[24] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0019, BUS_ACCESS, Bus access

The counter counts each Memory-read operation or Memory-write operation that accesses outside of the boundary of the PE and its closely-coupled caches. Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED. Where an implementation has multiple buses at this boundary, this event counts the sum of accesses across all buses.

A bus access is part of a bus transaction. The exact nature of a bus transaction is IMPLEMENTATION DEFINED, but for the purposes of event monitoring consists of a single access comprising one or more cycles, or beats, when the transaction occupies the bus. The BUS_ACCESS event counts each beat of each transaction. That is, each bus cycle counted by BUS_CYCLES for which the bus is active.
Bus transactions include refills of and write-backs from data, instruction, and unified caches. Whether bus transactions include operations that use the bus but do not explicitly transfer data is IMPLEMENTATION DEFINED.

An Unattributable bus transaction occurs when a requestor outside the PE makes a request that results in a bus access, for example, a coherency request.

If a bus supports multiple accesses per cycle, for example through multiple channels, the counter increments once for each channel that is active on a cycle, and so it might increment by more than one in any given cycle.

The maximum increment in any given cycle is IMPLEMENTATION DEFINED.

PMCEID0_EL0[25] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x001A, MEMORY_ERROR, Local memory error
The counter counts every occurrence of a memory error signaled by a memory closely coupled to this PE. The definition of local memories is IMPLEMENTATION DEFINED but includes caches, tightly-coupled memories, and TLB arrays.

Memory error refers to a physical error detected by the hardware, such as a parity or ECC error. It includes errors that are correctable and those that are not. It does not include errors as defined in the architecture, such as MMU faults.

PMCEID0_EL0[26] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x001B, INST_SPEC, Operation speculatively executed
The counter counts instructions that are Speculatively executed by the PE. This includes instructions that are subsequently not architecturally executed. The definition of Speculatively executed is IMPLEMENTATION DEFINED.

PMCEID0_EL0[27] reads as 0b1 if this event is implemented and 0b0 otherwise.

This event must be implemented if event INST_RETIRED is not implemented.

0x001D, BUS_CYCLES, Bus cycle
The counter increments on every cycle of the interface at the boundary of the PE and its closely-coupled caches. Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED.

------- Note -------
If the implementation clocks the external memory interface at the same rate as the processor hardware, the counter counts every cycle.

PMCEID0_EL0[29] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x001F, L1D_CACHE_ALLOCATE, Level 1 data cache allocation without refill
The counter counts each Memory-write operation that writes an entire line into the Level 1 data or unified cache without fetching data from outside the Level 1 data or unified cache.

These are allocations of cache lines in the Level 1 data or unified cache that are not refills counted by L1D_CACHE_REFILL. For example:

• A write of an entire cache line from a coalescing write buffer.

• A DC ZVA operation.

The counter counts only Memory-write operations Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation.

PMCEID0_EL0[31] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0020, L2D_CACHE_ALLOCATE, Level 2 data cache allocation without refill
The counter counts each Memory-write operation that writes an entire line into the Level 2 data or unified cache without fetching data from outside the Level 2 data or unified cache.
These are allocations of cache lines in the Level 2 data or unified cache that are not refills counted by L2D_CACHE_REFILL. For example:

- A write-back of an entire cache line from a Level 1 cache to the Level 2 data cache.
- A write of an entire cache line from a coalescing write buffer.
- A DC_ZVA operation.

The counter counts only Memory-write operations Attributable to the PE counting the event, and, if the Effective value of PMEVTYPE<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation.

PMCEID1_EL0[0] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0022, BR_MIS_PRED RETIRED, Branch Instruction architecturally executed, mispredicted

The counter counts each instructions counted by BR_RETIRED that were not correctly predicted. If no program-flow prediction resources are implemented, this event counts all retired not-taken branches.

PMCEID1_EL0[0] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0023, STALL_FRONTEND, No operation sent for execution due to the frontend

The counter counts each cycle counted by CPU_CYCLES where no Attributable instruction or operation was sent for execution and there was no Attributable instruction or operation available to dispatch for the PE from the frontend.

The division between frontend and backend is IMPLEMENTATION DEFINED. All STALL events must count at the same point in the pipeline.

--- Note ---

For a simplified pipeline model of Fetch-Decode-Issue-Execute-Retire, Arm recommends that the events are counted when instructions are dispatched from Decode to Issue.

In a single cycle, both the STALL_BACKEND and STALL_FRONTEND events might be counted, if both the backend is unable to accept any operations and there are no operations available to issue from the frontend.

PMCEID1_EL0[3] reads as 0b1 if this event is implemented and 0b0 otherwise. This event must be implemented if FEAT_PMUv3p1 is implemented.

0x0024, STALL_BACKEND, No operation sent for execution due to the backend

The counter counts each cycle counted by CPU_CYCLES where no Attributable instruction or operation was sent for execution and either:

- The backend is unable to accept any of the instruction operations available for the PE.
- The backend is unable to accept any operations for the PE.

For example, the backend might be unable to accept operations because of a resource conflict or non-availability.

The division between frontend and backend is IMPLEMENTATION DEFINED. For more information, see STALL_FRONTEND. All STALL events must count at the same point in the pipeline.

--- Note ---

In a single cycle, both the STALL_BACKEND and STALL_FRONTEND events might be counted, if both the backend is unable to accept any operations and there are no operations available to issue from the frontend.

PMCEID1_EL0[4] reads as 0b1 if this event is implemented and 0b0 otherwise. This event must be implemented if FEAT_PMUv3p1 is implemented.
0x0025, **L1D_TLB, Level 1 data TLB access**  
The counter counts each Memory-read operation or Memory-write operation that causes a TLB access to at least the Level 1 data or unified TLB.  
Each access to a TLB entry is counted including multiple accesses caused by single instructions such as LDM or STM.  
The counter does not count the access if the access is due to a TLB maintenance instruction.  
If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\text{n}>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.  
PMCEID1_EL0[5] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0026, **L1I_TLB, Level 1 instruction TLB access**  
The counter counts each Instruction memory access that causes a TLB access to at least the Level 1 instruction TLB.  
The counter does not count the access if the access is due to a TLB maintenance instruction.  
If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\text{n}>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.  
PMCEID1_EL0[6] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0027, **L2I_CACHE, Level 2 instruction cache access**  
The counter counts each Instruction memory access to at least the Level 2 instruction or unified cache.  
Each Instruction memory access to other Level 2 instruction or unified memory structures, such as refill buffers, is also counted.  
It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.  
If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.  
If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.  
An example of cache status information is whether the cached data is held in an exclusive or shared state.  
When the L2I_CACHE_RD event is implemented:  
• If the L2I_CACHE_PRFM event is implemented, accesses to the Level 2 instruction cache due to a preload or prefetch instruction are counted. Otherwise, these are not counted.  
• If the L2I_CACHE_HWPRF event is implemented, accesses to the Level 2 instruction cache due to a hardware prefetcher are counted. Otherwise these events are not counted.  
When the L2I_CACHE_RD event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 2 instruction cache due to preload or prefetch instructions or due to a hardware prefetcher are counted.  
If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\text{n}>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.  
PMCEID1_EL0[7] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0028, **L2I_CACHE_REFILL, Level 2 instruction cache refill**  
The counter counts each access counted by L2I_CACHE that causes a refill of the Level 2 instruction or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.
A refill includes any access that causes data to be fetched from outside of the Level 1 and Level 2 caches, even if the data is ultimately not allocated into the Level 2 instruction cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Level 2 instruction cache that cause a refill that is satisfied by fetching data from memory, a Level 3 cache, or a Level 2 cache of another PE.
- Refills of any Level 1 instruction or unified cache of this PE that cause a refill from outside of the Level 1 and Level 2 caches of this PE.
- Accesses to the Level 2 instruction cache that cause a refill of a Level 1 cache of this PE from outside of the Level 1 and Level 2 caches of this PE, even if there is no refill of the Level 2 instruction cache.

The counter does not count accesses that miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

PMCEID1_EL0[8] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0029, L3D_CACHE_ALLOCATE, Level 3 data cache allocation without refill

The counter counts each Memory-write operation that writes an entire line into the Level 3 data or unified cache without fetching data from outside the Level 3 data or unified cache. These are allocations of cache lines in the Level 3 data or unified cache that are not refills counted by L3D_CACHE_REFILL. For example:

- A write-back of an entire cache line from a Level 2 cache to the Level 3 data cache.
- A write of an entire cache line from a coalescing write buffer.
- A DC ZVA operation.

The counter counts only Memory-write operations Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation.

PMCEID1_EL0[9] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x002A, L3D_CACHE_REFILL, Level 3 data cache refill

The counter counts each access counted by L3D_CACHE that causes a refill of the Level 3 data or unified cache, or any Level 1 or Level 2 data, instruction, or unified cache of this PE, from outside of those caches.

A refill includes any access that causes data to be fetched from outside of the Level 1 to Level 3 caches, even if the data is ultimately not allocated into the Level 3 data cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Level 3 data cache that cause a refill that is satisfied by fetching data from memory, a lower level cache, or a Level 3 cache of another PE.
- Refills of and write-backs from any Level 1 or Level 2 data, instruction, or unified caches of this PE that cause a refill from outside of the Level 1 to Level 3 caches of this PE.
- Accesses to the Level 3 data cache that cause a refill of a Level 2 cache of this PE from outside of the Level 1 to Level 3 caches of this PE, even if there is no refill of the Level 3 data cache.
The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

PMCEID1_EL0[10] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x002B, L3D_CACHE, Level 3 data cache access

The counter counts each Memory-read operation or Memory-write operation that causes a cache access to at least the Level 3 data or unified cache.

Each access to a cache line is counted including refills of and write-backs from other caches. Each access to other Level 3 data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

An example of cache status information is whether the cached data is held in an exclusive or shared state.

When the L3D_CACHE_RW event is implemented:

- If the L3D_CACHE_PRFM event is implemented, accesses to the Level 3 data cache due to a preload or prefetch instruction are counted. Otherwise, these are not counted.
- If the L3D_CACHE_HWPRF event is implemented, accesses to the Level 3 data cache due to a hardware prefetcher are counted. Otherwise these events are not counted.

When the L3D_CACHE_RW event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 3 data cache due to preload or prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

PMCEID1_EL0[11] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x002C, L3D_CACHE_WB, Level 3 data cache write-back

The counter counts each write-back of data from the Level 3 data or unified cache to outside of the Level 1 to Level 3 caches. For example:

- A write-back of a dirty cache line to a lower level cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level 3 data cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

- A transfer of a dirty cache line from the Level 3 data cache to outside of Level 1 to Level 3 caches made as a result of a coherency request.
- Write-backs made as a result of cache maintenance instructions.
The counter does not count:

- The invalidation of a cache line without any write-back to outside of the Level 1 to Level 3 caches or memory.
- Writes that write through the Level 3 data cache to outside of the Level 1 to Level 3 caches.
- Transfers of data from the Level 3 data cache to a Level 1 or Level 2 cache to satisfy a refill of the other cache.

A write-back is attributable to the agent that generated the request that caused the write-back. This might not be the same agent that caused the data being written back to be allocated into the cache.

An Unattributable write-back event occurs when a requestor outside of the PE makes a coherency request that results in write-back. If the cache is shared, then an Unattributable write-back event is not counted. If the cache is not shared, then the event is counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines software is streaming writes to memory and does not allocate lines to the cache, or by a DC ZVA operation.

PMCEID1_EL0[12] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x002D, L2D_TLB_REFILL, Level 2 data TLB refill

The counter counts each access counted by L2D_TLB that causes a TLB refill of the Level 2 data or unified TLB, or any Level 1 data, instruction, or unified TLBs of this PE.

This includes each access that causes a memory access due to a translation table walk or an access to another level of TLB caching.

The counter does not count the access if any of the following are true:

- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 0b1.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.
- The access is due to a TLB maintenance instruction.
- The access misses in the TLB and generates a translation table walk but does not cause a refill of the TLB.

It is IMPLEMENTATION DEFINED whether the counter counts the access if any of the following are true:

- The access generates a Translation fault for any other reason.
- The refill is not allocated in the TLB.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

PMCEID1_EL0[13] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x002E, L2I_TLB_REFILL, Level 2 instruction TLB refill

The counter counts each Instruction memory access counted by L2I_TLB that causes a TLB refill of the Level 2 instruction TLB, or any Level 1 data, instruction, or unified TLBs of this PE.

This includes each Instruction memory access that causes a memory access due to a translation table walk or an access to another level of TLB caching.

The counter does not count the access if any of the following are true:

- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.
- The access is due to a TLB maintenance instruction.
• The access misses in the TLB and generates a translation table walk but does not cause a refill of the TLB.

It is IMPLEMENTATION DEFINED whether the counter counts the access if any of the following are true:
• The access generates a Translation fault for any other reason.
• The refill is not allocated in the TLB.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

PMCEID1_EL0[14] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x002F, L2D_TLB, Level 2 data TLB access

The counter counts each Memory-read operation or Memory-write operation that causes a TLB access to at least the Level 2 data or unified TLB.

Each access to a TLB entry is counted including refills of Level 1 TLBs.

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

PMCEID1_EL0[15] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0030, L2I_TLB, Level 2 instruction TLB access

The counter counts each Instruction memory access that causes a TLB access to at least the Level 2 instruction TLB.

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

PMCEID1_EL0[16] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0031, REMOTE_ACCESS, Access to another socket in a multi-socket system

The counter counts each Memory-read operation or Memory-write operation that causes an access to a different device or socket in a multi-socket system. That is, a socket that does not contain the PE.

System topology is IMPLEMENTATION DEFINED. This means that it is IMPLEMENTATION DEFINED which systems are defined as multi-socket systems, and, in systems defined as multi-socket systems, which components are defined as being in the same or different sockets. Arm recommends that devices where an access through that device incurs a significant latency penalty compared to other accesses are treated as being in a different socket.

For example, in a system comprising multiple integrated circuits in a multi-chip module, an access to a different integrated circuit in the same module might be treated as an access to another socket, even though the multi-chip module is physically connected to a single socket at the motherboard. However, in another system with many such multi-chip modules, an access to a different integrated system in the same module might be treated as an access to the same socket because an access to an integrated circuit on a different module has much higher latency.

The count includes all accesses to external memory counted by REMOTE_MEM. For more information, see REMOTE_MEM.

It is IMPLEMENTATION DEFINED whether an access that causes a snoop into a different socket but does not return data from or pass data to the remote socket is counted.

PMCEID1_EL0[17] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0032, LL_CACHE, Last level cache access

The counter counts each Memory-read operation or Memory-write operation that causes a cache access to at least the Last level cache.
Each access to a cache line is counted including refills of and write-backs from other caches. Each access to other Last level data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

An example of cache status information is whether the cached data is held in an exclusive or shared state.

When the LL_CACHE_RW event is implemented:

- If the LL_CACHE_PRFM event is implemented, accesses to the Last level cache due to a preload or prefetch instruction are counted. Otherwise, these are not counted.
- If the LL_CACHE_HWPRF event is implemented, accesses to the Last level cache due to a hardware prefetcher are counted. Otherwise these events are not counted.

When the LL_CACHE_RW event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Last level cache due to preload or prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

PMCEID1_EL0[19] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0033, LL_CACHE_MISS, Last level cache miss

If the LL_CACHE_RW event is implemented, the counter counts each access counted by LL_CACHE_RW that is not completed by the Last level cache.

If the LL_CACHE_RW event is not implemented, the counter counts each access counted by LL_CACHE that is not completed by the Last level cache.

That is, the access is one of the following:

- A Memory-read operation that does not return data from the Last level cache.
- A Memory-write operation that does not update the Last level cache.

The counter does not count operations that are completed by a cache above the Last level cache. It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

PMCEID1_EL0[19] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0034, DTLB_WALK, Data TLB access with at least one translation table walk

The counter counts each access counted by L1D_TLB that causes a refill of a data or unified TLB involving at least one translation table walk access.

This includes each complete or partial translation table walk that causes an access to memory, including to data or translation table walk caches.

If Armv8.7 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that cause an update of an existing TLB entry involving at least one translation table walk access are counted. If Armv8.7 is implemented, these accesses are counted.
The counter does not count the access if any of the following are true:

- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 0b1.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.
- The access is due to a TLB maintenance instruction.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

PMCEID1_EL0[20] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0035, ITLB_WALK, Instruction TLB access with at least one translation table walk

The counter counts each access counted by L1I_TLB that causes a refill of an instruction TLB involving at least one translation table walk access.

This includes each complete or partial translation table walk that causes an access to memory, including to data or translation table walk caches.

If Armv8.7 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that cause an update of an existing TLB entry involving at least one translation table walk access are counted. If Armv8.7 is implemented, these accesses are counted.

The counter does not count the access if any of the following are true:

- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.
- The access is due to a TLB maintenance instruction.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

PMCEID1_EL0[21] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0036, LL_CACHE_RD, Last level cache access, read

If the LL_CACHE_RW event is implemented, the counter counts each access counted by LL_CACHE_RW that is a Memory-read operation.

If the LL_CACHE_RW event is not implemented, the counter counts each access counted by LL_CACHE that is a Memory-read operation.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

PMCEID1_EL0[22] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0037, LL_CACHE_MISS_RD, Last level cache miss, read

The counter counts each access counted by both LL_CACHE_MISS and LL_CACHE_RD.

That is, every miss of the Last level cache counted by LL_CACHE_MISS that is caused by a Memory-read operation.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.
PMCEID1_EL0[23] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0038, `REMOTE_ACCESS_RD`, Access to another socket in a multi-socket system, read
The counter counts each access counted by `REMOTE_ACCESS` that is a Memory-read operation.
PMCEID1_EL0[24] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0039, `L1D_CACHE_LMISS_RD`, Level 1 data cache long-latency read miss
The counter counts each Memory-read operation to the Level 1 data or unified cache counted by `L1D_CACHE` that incurs additional latency because it returns data from outside of the Level 1 data or unified cache of this PE.
The event indicates to software that the access missed in the Level 1 data or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 1 data or unified cache.
The counter does not count:
- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 1 caches that the access looks up in and results in an access being made to a Level 2 cache or elsewhere beyond the Level 1 data or unified cache.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.
An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.
It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.
If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.
PMCEID1_EL0[25] reads as 0b1 if this event is implemented and 0b0 otherwise.
This event must be implemented if FEAT_PMUv3p4 is implemented.

0x003A, `OP_RETIRED`, Micro-operation architecturally executed
The counter counts each operation counted by `OP_SPEC` that would be executed in a Simple sequential execution of the program.
PMCEID1_EL0[26] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x003B, `OP_SPEC`, Micro-operation Speculatively executed
The counter counts the number of operations executed by the PE, including those that are executed speculatively and would not be executed in a Simple sequential execution of the program.
PMCEID1_EL0[27] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x003C, `STALL`, No operation sent for execution
The counter counts each cycle counted by `CPU_CYCLES` where no operation was sent for execution.
On a multithreaded implementation:
- If the Effective value of PMEVTYPER<n>_EL0.MT is 0b0, then the counter counts cycles when the PE is active and no operation for the PE is sent for execution, even if operations Attributable to other PEs in the multithreaded implementation are sent for execution in that cycle. The counter does not count cycles when the PE is not active.
- If the Effective value of PMEVTYPER<n>_EL0.MT is 0b1, then the counter counts all cycles when no instructions or operations for any PE in the multithreaded implementation were sent for execution.
All STALL events must count at the same point in the pipeline.

PMCEID1_EL0[28] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x0030, **STALL_SLOT_BACKEND, No operation sent for execution on a Slot due to the backend**

Counts each Slot counted by STALL_SLOT where no Attributable instruction or operation was sent for execution and either:

- The backend is unable to accept any of the instruction operations available on the Slot for the PE.
- The backend is unable to accept any operations on the Slot for the PE.

The division between frontend and backend is IMPLEMENTATION DEFINED. For more information, see STALL_FRONTEND. All STALL events must count at the same point in the pipeline. The maximum value by which STALL_SLOT_BACKEND can count in a single cycle is an IMPLEMENTATION DEFINED fixed value, \( s1\)ots. For more information, see STALL_SLOT.

—— Note ——

In a single cycle, the sum of values counted by STALL_SLOT_BACKEND and STALL_SLOT_FRONTEND might be greater-than \( s1\)ots, if both the backend is unable to accept some operations and there are fewer-than \( s1\)ots operations available to dispatch from the frontend.

PMCEID1_EL0[29] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x003E, **STALL_SLOT_FRONTEND, No operation sent for execution on a Slot due to the frontend**

Counts each Slot counted by STALL_SLOT where no Attributable instruction or operation was sent for execution and there was no Attributable instruction or operation available to dispatch for the PE from the frontend for the Slot.

The division between frontend and backend is IMPLEMENTATION DEFINED. For more information, see STALL_FRONTEND. All STALL events must count at the same point in the pipeline. The maximum value by which STALL_SLOT_FRONTEND can count in a single cycle is an IMPLEMENTATION DEFINED fixed value, \( s1\)ots. For more information, see STALL_SLOT.

—— Note ——

In a single cycle, the sum of values counted by STALL_SLOT_BACKEND and STALL_SLOT_FRONTEND might be greater-than \( s1\)ots, if both the backend is unable to accept some operations and there are fewer-than \( s1\)ots operations available to dispatch from the frontend.

PMCEID1_EL0[30] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x003F, **STALL_SLOT, No operation sent for execution on a Slot**

The counter counts on each cycle the number of instruction or operation Slots that were not occupied by an instruction or operation Attributable to the PE.

All STALL events must count at the same point in the pipeline. The maximum value by which STALL_SLOT can count in a single cycle is an IMPLEMENTATION DEFINED fixed value, \( s1\)ots. The definition of a Slot is IMPLEMENTATION DEFINED. The formula

\[
STALL_SLOT = \frac{(CPU_CYCLES \times s1\)ots)}{TIME}
\]

gives the utilization of the Slots of the processor by Attributable instruction or operations of this PE. Each Slot holds at most one instruction or operation each cycle.

On a multithreaded implementation:

- If the Effective value of PMEVTYPER<\(n\)>_EL0.MT is 0b0, then the counter counts Slots occupied by an instruction or operation Attributable to other PEs of the multithreaded implementation only when the PE was active in that cycle. The counter does not count Slots on cycles when the PE was not active.
- If the Effective value of PMEVTYPER<\(n\)>_EL0.MT is 0b1, then, for every cycle, the counter counts all Slots not occupied by any instruction or operation for any PE of the multithreaded implementation.

PMCEID1_EL0[31] reads as 0b1 if this event is implemented and 0b0 otherwise.
The Performance Monitors Extension
D11.11 PMU events and event numbers

0x0040, L1D_CACHE_RD, Level 1 data cache access, read
If the L1D_CACHE_RW event is implemented, the counter counts each access counted by L1D_CACHE_RW that is a Memory-read operation.
If the L1D_CACHE_RW event is not implemented, the counter counts each access counted by L1D_CACHE that is a Memory-read operation.
If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0041, L1D_CACHE_WR, Level 1 data cache access, write
If the L1D_CACHE_RW event is implemented, the counter counts each access counted by L1D_CACHE_RW that is a Memory-write operation.
If the L1D_CACHE_RW event is not implemented, the counter counts each access counted by L1D_CACHE that is a Memory-write operation.
If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0042, L1D_CACHE_REFILL_RD, Level 1 data cache refill, read
The counter counts each access counted by both L1D_CACHE_RD and L1D_CACHE_REFILL. That is, every refill of the Level 1 data or unified cache counted by L1D_CACHE_REFILL that is caused by a Memory-read operation.
If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0043, L1D_CACHE_REFILL_WR, Level 1 data cache refill, write
The counter counts each access counted by both L1D_CACHE_REFILL and L1D_CACHE_WR. That is, every refill of the Level 1 data or unified cache counted by L1D_CACHE_REFILL that is caused by a Memory-write operation.
If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0044, L1D_CACHE_REFILL_INNER, Level 1 data cache refill, inner
The counter counts each access counted by L1D_CACHE_REFILL that generates a refill satisfied by transfer from another cache inside of the immediate cluster.

Note
The boundary between inner and outer is IMPLEMENTATION DEFINED, and it is not necessarily linked to other similar boundaries, such as the boundary between Inner Cacheable and Outer Cacheable or the boundary between Inner Shareable and Outer Shareable.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.
0x0045, L1D_CACHE_REFILL_OUTER, Level 1 data cache refill, outer
The counter counts each access counted by L1D_CACHE_REFILL that generates a refill satisfied by transfer from outside of the immediate cluster.

Note
The boundary between inner and outer is IMPLEMENTATION DEFINED, and it is not necessarily linked to other similar boundaries, such as the boundary between Inner Cacheable and Outer Cacheable or the boundary between Inner Shareable and Outer Shareable.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0046, L1D_CACHE_WB_VICTIM, Level 1 data cache write-back, victim
The counter counts each write-back counted by L1D_CACHE_WB that occurs because the line is allocated for an access made by the PE.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0047, L1D_CACHE_WB_CLEAN, Level 1 data cache write-back, cleaning and coherency
The counter counts each write-back counted by L1D_CACHE_WB that occurs because of a coherency operation made by another PE or, optionally, the execution of a cache maintenance instruction.
Whether write-backs that are caused by the execution of a cache maintenance instruction are counted is IMPLEMENTATION DEFINED.

Note
The transfer of a dirty cache line from the Level 1 data cache of this PE to the data cache of another PE due to a hardware coherency operation is not counted unless the dirty cache line is also written back to a Level 2 cache or memory.

If a coherency request from a requestor outside of the PE results in a write-back, it is an Unattributable event.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0048, L1D_CACHE_INV_AL, Level 1 data cache invalidate
The counter counts each invalidation of a cache line in the Level 1 data or unified cache. For example:
• Invalidation of a cache line because of a cache maintenance operation.
• Transfer of ownership of a cache line to another cache because of a coherency or refill request.

The counter does not count events if a cache refill of the Level 1 data or unified cache invalidates a line in the Level 1 data or unified cache.
If FEAT_PMUv3p4 is not implemented, the counter does not count locally-executed cache maintenance instructions that operate by set/way.
If FEAT_PMUv3p4 is implemented, it is IMPLEMENTATION DEFINED whether the counter counts locally-executed cache maintenance instructions that operate by set/way.

Note
Software that uses this event must know whether the Level 1 data cache is shared with other PEs. This event does not follow the general rule of Level 1 data cache events of only counting Attributable events.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.
0x004C, L1D_TLB_REFILL_RD, Level 1 data TLB refill, read

The counter counts each access counted by both L1D_TLB_RD and L1D_TLB_REFILL.

The counter does not count the access if any of the following are true:

- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 0b1.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x004D, L1D_TLB_REFILL_WR, Level 1 data TLB refill, write

The counter counts each access counted by both L1D_TLB_REFILL and L1D_TLB_WR.

The counter does not count the access if any of the following are true:

- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 0b1.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x004E, L1D_TLB_RD, Level 1 data TLB access, read

If the L1D_TLB_RW event is implemented, the counter counts each access counted by L1D_TLB_RW that is a Memory-read operation.

If the L1D_TLB_RW event is not implemented, the counter counts each access counted by L1D_TLB that is a Memory-read operation.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x004F, L1D_TLB_WR, Level 1 data TLB access, write

If the L1D_TLB_RW event is implemented, the counter counts each access counted by L1D_TLB_RW that is a Memory-write operation.

If the L1D_TLB_RW event is not implemented, the counter counts each access counted by L1D_TLB that is a Memory-write operation.

DC ZVA is counted as a Memory-write operation.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.
0x0050, L2D_CACHE_RD, Level 2 data cache access, read
   If the L2D_CACHE_RW event is implemented, the counter counts each access counted by L2D_CACHE_RW that is a Memory-read operation.
   If the L2D_CACHE_RW event is not implemented, the counter counts each access counted by L2D_CACHE that is a Memory-read operation.
   If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted. When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0051, L2D_CACHE_WR, Level 2 data cache access, write
   If the L2D_CACHE_RW event is implemented, the counter counts each access counted by L2D_CACHE_RW that is a Memory-write operation.
   If the L2D_CACHE_RW event is not implemented, the counter counts each access counted by L2D_CACHE that is a Memory-write operation.
   If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted. When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0052, L2D_CACHE_REFILL_RD, Level 2 data cache refill, read
   The counter counts each access counted by both L2D_CACHE_RD and L2D_CACHE_REFILL. That is, every refill of the Level 2 data or unified cache counted by L2D_CACHE_REFILL that is caused by a Memory-read operation.
   If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted. When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0053, L2D_CACHE_REFILL_WR, Level 2 data cache refill, write
   The counter counts each access counted by both L2D_CACHE_REFILL and L2D_CACHE_WR. That is, every refill of the Level 2 data or unified cache counted by L2D_CACHE_REFILL that is caused by a Memory-write operation.
   If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted. When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0056, L2D_CACHE_WB_VICTIM, Level 2 data cache write-back, victim
   The counter counts each write-back counted by L2D_CACHE_WB that occurs because the line is allocated for an access made by the PE.
   When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0057, L2D_CACHE_WB_CLEAN, Level 2 data cache write-back, cleaning and coherency
   The counter counts each write-back counted by L2D_CACHE_WB that occurs because of a coherency operation made by another PE or, optionally, the execution of a cache maintenance instruction.
   Whether write-backs that are caused by the execution of a cache maintenance instruction are counted is IMPLEMENTATION DEFINED.
Note
The transfer of a dirty cache line from the Level 2 data cache of this PE to the data cache of another PE due to a hardware coherency operation is not counted unless the dirty cache line is also written back to a Level 3 cache or memory.

If a coherency request from a requestor outside of the PE results in a write-back, it is an Unattributable event.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0058, L2D_CACHE_INV, Level 2 data cache invalidate
The counter counts each invalidation of a cache line in the Level 2 data or unified cache. For example:
• Invalidation of a cache line because of a cache maintenance operation.
• Transfer of ownership of a cache line to another cache because of a coherency or refill request.

The counter does not count events if a cache refill of the Level 2 data or unified cache invalidates a line in the Level 2 data or unified cache.

If FEAT_PMUv3p4 is not implemented, the counter does not count locally-executed cache maintenance instructions that operate by set/way.

If FEAT_PMUv3p4 is implemented, it is IMPLEMENTATION DEFINED whether the counter counts locally-executed cache maintenance instructions that operate by set/way.

Note
Software that uses this event must know whether the Level 2 data cache is shared with other PEs. This event does not follow the general rule of Level 2 data cache events of only counting Attributable events.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x005C, L2D_TLB_REFILL_RD, Level 2 data TLB refill, read
The counter counts each access counted by both L2D_TLB_RD and L2D_TLB_REFILL.

The counter does not count the access if any of the following are true:
• FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.
• FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 0b1.
• The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x005D, L2D_TLB_REFILL_WR, Level 2 data TLB refill, write
The counter counts each access counted by both L2D_TLB_REFILL and L2D_TLB_WR.

The counter does not count the access if any of the following are true:
• FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.
• FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 0b1.
• The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.
It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\text{n}>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

\textbf{0x005E, L2D\_TLB\_RD, Level 2 data TLB access, read}

The counter counts each access that is a Memory-read operation.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\text{n}>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

\textbf{0x005F, L2D\_TLB\_WR, Level 2 data TLB access, write}

The counter counts each access that is a Memory-write operation.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\text{n}>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

\textbf{0x0060, BUS\_ACCESS\_RD, Bus access, read}

The counter counts each access counted by BUS\_ACCESS that is a Memory-read operation.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

\textbf{0x0061, BUS\_ACCESS\_WR, Bus access, write}

The counter counts each access counted by BUS\_ACCESS that is a Memory-write operation.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

\textbf{0x0062, BUS\_ACCESS\_SHARED, Bus access, Normal, Cacheable, Shareable}

The counter counts each access counted by BUS\_ACCESS that is Normal, Cacheable, Shareable.

\textbf{Note}

It is IMPLEMENTATION DEFINED how the PE translates the attributes from the translation table entry for a region to the attributes on the bus.

In particular, a region of memory designated as Normal, Cacheable, Inner Shareable, Not Outer Shareable by a translation table entry, might be marked as either shareable or Non-shareable at the boundary of the PE and its closely-coupled caches. This depends on where the IMPLEMENTATION DEFINED boundary lies, between Inner and Outer Shareable.

If the Inner Shareable extends beyond the PE boundary, and the bus indicates the distinction between Inner and Outer Shareable, then either is counted as shareable for the purposes of defining this event.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

\textbf{0x0063, BUS\_ACCESS\_NOT\_SHARED, Bus access, not Normal, Cacheable, Shareable}

The counter counts each access counted by BUS\_ACCESS that is not counted by BUS\_ACCESS\_SHARED.

For example, the counter counts accesses marked as:

- Normal, Cacheable, Non-shareable.
- Normal, Non-cacheable.
- Device.
_____ Note _____

It is IMPLEMENTATION DEFINED, how the PE translates the attributes from the translation table entries for a region to the attributes on the bus.

In particular, a region of memory designated as Normal, Cacheable, Inner Shareable, Not Outer Shareable by a translation table entry, might be marked as either shareable or Non-shareable at the boundary of the PE and its closely-coupled caches. This depends on where the IMPLEMENTATION DEFINED boundary lies, between Inner and Outer Shareable.

If the Inner Shareable extends beyond the PE boundary, and the bus indicates the distinction between Inner and Outer Shareable, then either is counted as shareable for the purposes of defining this event.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0064, BUS_ACCESS_NORMAL, Bus access, normal

The counter counts each access counted by BUS_ACCESS that is to Normal or bulk memory.

For example, the counter counts Normal, Cacheable and Normal, Non-cacheable accesses but does not count Device accesses.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0065, BUS_ACCESS_PERIPH, Bus access, peripheral

The counter counts each access counted by BUS_ACCESS that is not counted by BUS_ACCESS_NORMAL.

For example, the counter counts accesses to Device memory.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0066, MEM_ACCESS_RD, Data memory access, read

If the MEM_ACCESS_RW event is implemented, the counter counts each access counted by MEM_ACCESS_RW that is a Memory-read operation.

If the MEM_ACCESS_RW event is not implemented, the counter counts each access counted by MEM_ACCESS that is a Memory-read operation.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0067, MEM_ACCESS_WR, Data memory access, write

If the MEM_ACCESS_RW event is implemented, the counter counts each access counted by MEM_ACCESS_RW that is a Memory-write operation.

If the MEM_ACCESS_RW event is not implemented, the counter counts each access counted by MEM_ACCESS that is a Memory-write operation.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0068, UNALIGNED_LD_SPEC, Unaligned access, read

The counter counts each access counted by MEM_ACCESS that is an unaligned Memory-read operation.

The unaligned access is counted even if it is subsequently transformed into multiple aligned accesses.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0069, UNALIGNED_ST_SPEC, Unaligned access, write

The counter counts each access counted by MEM_ACCESS that is an unaligned Memory-write operation.

The unaligned access is counted even if it is subsequently transformed into multiple aligned accesses.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.
0x006A, UNALIGNED_LDST_SPEC, Unaligned access
The counter counts each access counted by either UNALIGNED_LD_SPEC or UNALIGNED_ST_SPEC.
The unaligned access is counted even if it is subsequently transformed into multiple aligned accesses.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x006C, LDREX_SPEC, Exclusive operation Speculatively executed, Load-Exclusive
The counter counts Load-Exclusive instructions Speculatively executed.
The definition of Speculatively executed is IMPLEMENTATION DEFINED.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x006D, STREX_PASS_SPEC, Exclusive operation Speculatively executed, Store-Exclusive pass
The counter counts Store-Exclusive instructions Speculatively executed that completed a write.
The definition of Speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the LDREX_SPEC event.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x006E, STREX_FAIL_SPEC, Exclusive operation Speculatively executed, Store-Exclusive fail
The counter counts Store-Exclusive instructions Speculatively executed that fail to complete a write. It is within the IMPLEMENTATION DEFINED definition of Speculatively executed whether this includes conditional instructions that fail the condition code check.
The definition of Speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the LDREX_SPEC event.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x006F, STREX_SPEC, Exclusive operation Speculatively executed, Store-Exclusive
The counter counts Store-Exclusive instructions Speculatively executed.
The definition of Speculatively executed is IMPLEMENTATION DEFINED but it must be the same as for the LDREX_SPEC event.
Arm recommends that this event is implemented if it is not possible to implement the exclusive operation Speculatively executed, Store-Exclusive pass, and exclusive operation Speculatively executed, Store-Exclusive fail, events with the same degree of speculation as the LDREX_SPEC event.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0070, LD_SPEC, Operation speculatively executed, load
The counter counts each operation counted by INST_SPEC that is a memory-reading operation.
Memory-reading instructions are defined by the LD_RETIRED event.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0071, ST_SPEC, Operation speculatively executed, store
The counter counts each operation counted by INST_SPEC that is a memory-writing operation.
Memory-writing instructions are defined by the ST_RETIRED event.
The counter counts DC ZVA as a store operation.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0072, LDST_SPEC, Operation speculatively executed, load or store
The counter counts each operation counted by either LD_SPEC or ST_SPEC.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.
The counter counts each operation counted by **INST_SPEC** that is an integer data-processing operation.

The counter counts the following operations that operate on the general-purpose registers:

- In AArch64 state, *Data processing - immediate* on page C3-297 and *Data processing - register* on page C3-302.
- In AArch32 state, *Data-processing instructions* on page F2-7260.

This includes MOV and MVN operations.

This event also counts the following miscellaneous instructions:

- In AArch32 state, *PSTATE and banked register access instructions* on page F2-7268, *Miscellaneous instructions* on page F2-7273, other than ISB and preloads, and *System register access instructions* on page F2-7277, other than LDC and STC instructions.

If the preload instructions PRFM, PLD, PLDW, and PLI, do not count as memory-reading instructions then they must count as integer data-processing instructions.

If ISBs do not count as Software change of the PC then they must count as integer data-processing instructions.

It is IMPLEMENTATION DEFINED whether the following instructions are counted as integer data-processing operations, SIMD operations, or floating-point operations, but Arm recommends that the instructions are all counted as integer data-processing operations:

- For AArch64 state, from the A64 floating-point convert to integer class, operations that move a value between a general-purpose register and a SIMD and floating-point register without type conversion:
  - FMOV (general).

- For AArch64 state, from the SIMD Move group, operations that move a values between a general-purpose register and an element or elements in a SIMD and floating-point register:
  - DUP (general).
  - SMOV.
  - UM0V.
  - INS (general).

- For AArch32 state:
  - VDUP (general-purpose register) and all VMOV instructions that transfer data between a general-purpose register and a SIMD and floating-point register.
  - VMRS.
  - VMSR.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

**0x0074, ASE_SPEC, Operation speculatively executed, Advanced SIMD**

The counter counts each operation counted by **INST_SPEC** that is an Advanced SIMD data-processing operation.

See:

- For AArch64 state, the SIMD operations listed in *Data processing - SIMD and floating-point* on page C3-310.
- For AArch32 state, *Advanced SIMD data-processing instructions* on page F2-7281.

This includes all operations that operate on the SIMD and floating-point registers, except those that are counted as:

- Integer data-processing operations.
- Floating-point data-processing operations.
• Memory-reading operations.
• Memory-writing operations.
• Cryptographic operations other than PMULL, in AArch64 state.
• VMULL, in AArch32 state.

Advanced SIMD scalar operations are counted as Advanced SIMD operations, including those which operate on floating-point values. In AArch64 state, PMULL, and in AArch32 state, VMULL are counted as Advanced SIMD operations.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0075, VFP_SPEC, Operation speculatively executed, scalar floating-point

The counter counts each operation counted by INST_SPEC that is a scalar floating-point data-processing operation.

See:
• In AArch64 state, only the scalar floating-point operations listed in Data processing - SIMD and floating-point on page C3-310.
• In AArch32 state, Floating-point data-processing instructions on page F2-7292.

Note
This event does not count the SIMD floating-point operations listed in Data processing - SIMD and floating-point on page C3-310.

This includes all operations that operate on the SIMD and floating-point registers as floating-point values, except for SIMD scalar operations and those that are counted as one of:
• Integer data processing.
• Memory-reading operations.
• Memory-writing operations.

The following instructions that take both an integer register and a floating-point register argument and perform a type conversion (to/from integer or to/from fixed-point), are counted as floating-point data-processing operations:
• In AArch64 state, FCVT{<mode>}, UCVTF, and SCVTF.
• In AArch32 state, VCVT{<mode>}, (floating-point), VCVT, VCVT1, and VCVTB.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0076, PC_WRITE_SPEC, Operation speculatively executed, Software change of the PC

The counter counts each operation counted by INST_SPEC that is a Software change of the PC.

Software change of the PC operations are defined by the PC_WRITE_RETIRED event.

When FEAT_PMUv3p8 is implemented, the counter counts the operation even if the branch is not taken. Otherwise, it is IMPLEMENTATION DEFINED whether the counter counts the operation when the branch is not taken.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0077, CRYPTO_SPEC, Operation speculatively executed, Cryptographic instruction

The counter counts each operation counted by INST_SPEC that is a cryptographic operation other than PMULL or VMULL.

See The Cryptographic Extension on page C3-333.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0078, BR_IMMED_SPEC, Branch Speculatively executed, immediate branch

The counter counts immediate branch instructions Speculatively executed. Defined by the instruction architecturally executed, immediate branch event, see Common event numbers on page D11-5306.
The definition of Speculatively executed is IMPLEMENTATION DEFINED.
See also BR_IMMED_RETIRED.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0079, BR_RETURN_SPEC, Branch Speculatively executed, procedure return

The counter counts procedure return instructions Speculatively executed. Defined by the BR_RETURN_RETIRED event.
The definition of Speculatively executed is IMPLEMENTATION DEFINED.
See also BR_RETURN_RETIRED.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x007A, BR_INDIRECT_SPEC, Branch Speculatively executed, indirect branch

The counter counts indirect branch instructions Speculatively executed. This includes Software change of the PC other than exception-generating instructions and immediate branch instructions.
The definition of Speculatively executed is IMPLEMENTATION DEFINED.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x007C, ISB_SPEC, Barrier Speculatively executed, ISB

The counter counts Instruction Synchronization Barrier instructions Speculatively executed, including CP15ISB.
The definition of Speculatively executed is IMPLEMENTATION DEFINED.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x007D, DSB_SPEC, Barrier Speculatively executed, DSB

The counter counts data synchronization and speculative load barrier instructions Speculatively executed, including CP15DSB, SS88 and P5SB8.
The definition of Speculatively executed is IMPLEMENTATION DEFINED.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x007E, DMB_SPEC, Barrier Speculatively executed, DMB

The counter counts data memory barrier instructions Speculatively executed, including CP15DSB.
It does not include the implied barrier operations of load/store operations with release consistency semantics.
The definition of Speculatively executed is IMPLEMENTATION DEFINED.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x007F, CSDB_SPEC, Barrier Speculatively executed, CSDB

The counter counts control speculation barrier instructions Speculatively executed.
The definition of Speculatively executed is IMPLEMENTATION DEFINED.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0081, EXC_UNDEF, Exception taken, other synchronous

The counter counts each exception counted by EXC_TAKEN that is Taken locally and is not counted as any of the following:
• Exception taken, Supervisor Call (EXC_SVC).
• Exception taken, Secure Monitor Call (EXC_SMC).
• Exception taken, Hypervisor Call (EXC_HVC).
• Exception taken, Instruction Abort (EXC_PABORT).
• Exception taken, Data Abort or SError (EXC_DABORT).
• Exception taken, IRQ (EXC_IRQ).
• Exception taken, FIQ (EXC_FIQ).
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0082, EXC_SVC, Exception taken, Supervisor Call
The counter counts each Supervisor Call exception counted by EXC_TAKEN that is Taken locally.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0083, EXC_PABORT, Exception taken, Instruction Abort
The counter counts each Instruction Abort exception counted by EXC_TAKEN that is Taken locally.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0084, EXC_DABORT, Exception taken, Data Abort or SError
The counter counts each Data Abort, SError interrupt, or virtual SError interrupt exception counted by EXC_TAKEN that is Taken locally.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0086, EXC_IRQ, Exception taken, IRQ
The counter counts each IRQ or virtual IRQ exception counted by EXC_TAKEN that is Taken locally.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0087, EXC_FIQ, Exception taken, FIQ
The counter counts each FIQ or virtual FIQ exception counted by EXC_TAKEN that is Taken locally.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0088, EXC_SMC, Exception taken, Secure Monitor Call
The counter counts each Secure Monitor Call exception counted by EXC_TAKEN. The counter does not count SMC instructions that generate other exceptions, including Trap exceptions and Undefined Instruction exceptions.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x008A, EXC_HVC, Exception taken, Hypervisor Call
The counter counts each Hypervisor Call exception counted by EXC_TAKEN. The counter does not count HMC instructions that are UNDEFINED.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x008B, EXC_TRAP_PABORT, Exception taken, Instruction Abort not Taken locally
The counter counts each Instruction Abort exception counted by EXC_TAKEN that is not Taken locally.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x008C, EXC_TRAP_DABORT, Exception taken, Data Abort or SError not Taken locally
The counter counts each Data Abort, SError interrupt, or virtual SError interrupt exception counted by EXC_TAKEN, that is not Taken locally.
When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x008D, EXC_TRAP_OTHER, Exception taken, other traps not Taken locally
The counter counts each exception counted by EXC_TAKEN that is not Taken locally and not counted as any of the following:
• Exception taken, Secure Monitor Call (EXC_SMC).
• Exception taken, Hypervisor Call (EXC_HVC).
• Exception taken, Instruction Abort not Taken locally (EXC_TRAP_PABORT).
• Exception taken, Data Abort or SError not Taken locally (EXC_TRAP_DABORT).
• Exception taken, IRQ not Taken locally \(\text{EXC\_TRAP\_IRQ}\).
• Exception taken, FIQ not Taken locally \(\text{EXC\_TRAP\_FIQ}\).

When \(\text{FEAT\_PMUv3p8}\) is not implemented, this is an IMPLEMENTATION DEFINED event.

0x008E, \text{EXC\_TRAP\_IRQ}, \text{Exception taken, IRQ not Taken locally}

The counter counts each IRQ or virtual IRQ exception counted by \(\text{EXC\_TAKEN}\) that is not Taken locally.

When \(\text{FEAT\_PMUv3p8}\) is not implemented, this is an IMPLEMENTATION DEFINED event.

0x008F, \text{EXC\_TRAP\_FIQ}, \text{Exception taken, FIQ not Taken locally}

The counter counts each FIQ or virtual FIQ exception counted by \(\text{EXC\_TAKEN}\) that is not Taken locally.

When \(\text{FEAT\_PMUv3p8}\) is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0090, \text{RC\_LD\_SPEC}, \text{Release consistency operation Speculatively executed, Load-Acquire}

The counter counts Memory-read operations with acquire or AcquirePC semantics that are Speculatively executed.

When \(\text{FEAT\_PMUv3p8}\) is not implemented, this is an IMPLEMENTATION DEFINED event.

0x0091, \text{RC\_ST\_SPEC}, \text{Release consistency operation Speculatively executed, Store-Release}

The counter counts Memory-write operations with release semantics that are Speculatively executed.

When \(\text{FEAT\_PMUv3p8}\) is not implemented, this is an IMPLEMENTATION DEFINED event.

0x00A0, \text{L3D\_CACHE\_RD}, \text{Level 3 data cache access, read}

If the \text{L3D\_CACHE\_RW} event is implemented, the counter counts each access counted by \text{L3D\_CACHE\_RW} that is a Memory-read operation.

If the \text{L3D\_CACHE\_RW} event is not implemented, the counter counts each access counted by \text{L3D\_CACHE} that is a Memory-read operation.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of \text{PMEVTYPER<0>_EL0.MT} for the counter is \(0b1\), other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

When \(\text{FEAT\_PMUv3p8}\) is not implemented, this is an IMPLEMENTATION DEFINED event.

0x00A1, \text{L3D\_CACHE\_WR}, \text{Level 3 data cache access, write}

If the \text{L3D\_CACHE\_RW} event is implemented, the counter counts each access counted by \text{L3D\_CACHE\_RW} that is a Memory-write operation.

If the \text{L3D\_CACHE\_RW} event is not implemented, the counter counts each access counted by \text{L3D\_CACHE} that is a Memory-write operation.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of \text{PMEVTYPER<0>_EL0.MT} for the counter is \(0b1\), other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

When \(\text{FEAT\_PMUv3p8}\) is not implemented, this is an IMPLEMENTATION DEFINED event.

0x00A2, \text{L3D\_CACHE\_REFILL\_RD}, \text{Level 3 data cache refill, read}

The counter counts each access counted by both \text{L3D\_CACHE\_RD} and \text{L3D\_CACHE\_REFILL}.

That is, every refill of the Level 3 data or unified cache counted by \text{L3D\_CACHE\_REFILL} that is caused by a Memory-read operation.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of \text{PMEVTYPER<0>_EL0.MT} for the counter is \(0b1\), other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

When \(\text{FEAT\_PMUv3p8}\) is not implemented, this is an IMPLEMENTATION DEFINED event.
0x00A3, **L3D_CACHE_REFILL_WR, Level 3 data cache refill, write**

The counter counts each access counted by both **L3D_CACHE_REFILL** and **L3D_CACHE_WR**.

That is, every refill of the Level 3 data or unified cache counted by **L3D_CACHE_REFILL** that is caused by a Memory-write operation.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x00A6, **L3D_CACHE_WB_VICTIM, Level 3 data cache write-back, victim**

The counter counts each write-back counted by **L3D_CACHE_WB** that occurs because the line is allocated for an access made by the PE.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x00A7, **L3D_CACHE_WB_CLEAN, Level 3 data cache write-back, cleaning and coherency**

The counter counts each write-back counted by **L3D_CACHE_WB** that occurs because of a coherency operation made by another PE or, optionally, the execution of a cache maintenance instruction.

Whether write-backs that are caused by the execution of a cache maintenance instruction are counted is IMPLEMENTATION DEFINED.

Note

The transfer of a dirty cache line from the Level 3 data cache of this PE to the data cache of another PE due to a hardware coherency operation is not counted unless the dirty cache line is also written back to a lower level cache or memory.

If a coherency request from a requestor outside of the PE results in a write-back, it is an Unattributable event.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x00A8, **L3D_CACHE_INV, Level 3 data cache invalidate**

The counter counts each invalidation of a cache line in the Level 3 data or unified cache. For example:

- Invalidation of a cache line because of a cache maintenance operation.
- Transfer of ownership of a cache line to another cache because of a coherency or refill request.

The counter does not count events if a cache refill of the Level 3 data or unified cache invalidates a line in the Level 3 data or unified cache.

If FEAT_PMUv3p4 is not implemented, the counter does not count locally-executed cache maintenance instructions that operate by set/way.

If FEAT_PMUv3p4 is implemented, it is IMPLEMENTATION DEFINED whether the counter counts locally-executed cache maintenance instructions that operate by set/way.

Note

Software that uses this event must know whether the Level 3 data cache is shared with other PEs. This event does not follow the general rule of Level 3 data cache events of only counting Attributable events.

When FEAT_PMUv3p8 is not implemented, this is an IMPLEMENTATION DEFINED event.

0x4000, **SAMPLE_POP, Statistical Profiling sample population**

The counter counts each operation that might be sampled, whether or not the operation was sampled. Operations that are executed at an Exception level or Security state in which the Statistical Profiling Extension is disabled are not counted.
PMCEID0_EL0[32] reads as 0b1 if this event is implemented and 0b0 otherwise.
This event must be implemented if FEAT_SPE is implemented.

0x4001, SAMPLE_FEED, Statistical Profiling sample taken
The counter counts each time the sample interval counter reaches zero and is reloaded, and the sample does not collide with the previous sample. Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.
PMCEID0_EL0[33] reads as 0b1 if this event is implemented and 0b0 otherwise.
This event must be implemented if FEAT_SPE is implemented.

0x4002, SAMPLE_FILTRATE, Statistical Profiling sample taken and not removed by filtering
The counter counts each sample counted by SAMPLE_FEED that is not removed by filtering.
Sample records that are not removed by filtering, but are discarded before being written to the Profiling Buffer because of a Profiling Buffer management event or because Discard mode is implemented and enabled, are counted.
PMCEID0_EL0[34] reads as 0b1 if this event is implemented and 0b0 otherwise.
This event must be implemented if FEAT_SPE is implemented.

0x4003, SAMPLE_COLLISION, Statistical Profiling sample collided with previous sample
The counter counts each time the sample interval counter reaches zero and is reloaded, and the sample collides with the previous sample because the previous sampled operation has not completed generating its sample record.
PMCEID0_EL0[35] reads as 0b1 if this event is implemented and 0b0 otherwise.
This event must be implemented if FEAT_SPE is implemented.

0x4004, CNT_CYCLES, Constant frequency cycles
The counter increments at a constant frequency equal to the rate of increment of the System Counter, CNTPCT_EL0.
PMCEID0_EL0[36] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x4005, STALL_BACKEND_MEM, Memory stall cycles
The counter counts each cycle counted by STALL_BACKEND_MEMBOUND where there is a demand data miss in the last level of data or unified cache within the PE clock domain.
If Armv8.7 is implemented, the counter also counts backend stall cycles when a non-cacheable data access is in progress.
If Armv8.7 is not implemented, it is IMPLEMENTATION DEFINED whether the counter counts backend stall cycles when a non-cacheable data access is in progress.
PMCEID0_EL0[37] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x4006, L1I_CACHE_LMISS, Level 1 instruction cache long-latency miss
If the L1I CACHE_RD event is implemented, the counter counts each access to the Level 1 instruction or unified cache counted by L1I CACHE_RD that incurs additional latency because it returns instructions from outside of the Level 1 instruction or unified cache of this PE.
If the L1I CACHE_RD event is not implemented, the counter counts each access to the Level 1 instruction or unified cache counted by L1I CACHE that incurs additional latency because it returns instructions from outside of the Level 1 instruction or unified cache of this PE.
The event indicates to software that the access missed in the Level 1 instruction or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 1 instruction or unified cache.
The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 1 caches that the access looks up in and results in instructions being returned from a Level 2 cache or elsewhere beyond the Level 1 instruction or unified cache.

- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PM_EVTYPER<\n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

PMCEID0_EL0[38] reads as 0b1 if this event is implemented and 0b0 otherwise.

This event must be implemented if FEAT_PMUv3p4 is implemented.

0x4009, L2D_CACHE_LMISS_RD, Level 2 data cache long-latency read miss

The counter counts each Memory-read operation to the Level 2 data or unified cache counted by L2D_CACHE that incurs additional latency because it returns data from outside of the Level 1 and Level 2 data or unified caches of this PE.

The event indicates to software that the access missed in the Level 2 data or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 2 data or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 2 caches that the access looks up in and results in an access being made to a Level 3 cache or elsewhere beyond the Level 2 data or unified cache. This might be counted as a Level 1 cache miss.

- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PM_EVTYPER<\n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

PMCEID0_EL0[41] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x400A, L2I_CACHE_LMISS, Level 2 instruction cache long-latency miss

If the L2I_CACHE_RD event is implemented, the counter counts each access to the Level 2 instruction or unified cache counted by L2I_CACHE_RD that incurs additional latency because it returns instructions from outside of the Level 1 and Level 2 instruction or unified caches of this PE.

If the L2I_CACHE_RD event is not implemented, the counter counts each access to the Level 2 instruction or unified cache counted by L2I_CACHE that incurs additional latency because it returns instructions from outside of the Level 1 and Level 2 instruction or unified caches of this PE.
The event indicates to software that the access missed in the Level 2 instruction or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 2 instruction or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 2 caches that the access looks up in and results in instructions being returned from a Level 3 cache or elsewhere beyond the Level 2 instruction or unified cache. This might be counted as a Level 1 cache miss.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

PMCEID0_EL0[42] reads as 0b1 if this event is implemented and 0b0 otherwise.

**0x400B, L3D_CACHE_LMISS_RD, Level 3 data cache long-latency read miss**

The counter counts each Memory-read operation to the Level 3 data or unified cache counted by L3D_CACHE that incurs additional latency because it returns data from outside of the Level 1 to Level 3 data or unified caches of this PE.

The event indicates to software that the access missed in the Level 3 data or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 3 data or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 3 caches that the access looks up in and results in an access being made to a lower level cache or elsewhere beyond the Level 3 data or unified cache. This might be counted as a Level 2 cache miss.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

PMCEID0_EL0[43] reads as 0b1 if this event is implemented and 0b0 otherwise.

**0x4010, TRCEXTOUT0, Trace unit external output 0**

The event is generated each time an event is signaled by the trace unit external event 0.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[48] reads as 0b1 if this event is implemented and 0b0 otherwise.

This event must be implemented if FEAT_ETE is implemented.
0x4011, TRCEXTOUT1, Trace unit external output 1
The event is generated each time an event is signaled by the trace unit external event 1.
It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.
PMCEID0_EL0[49] reads as 0b1 if this event is implemented and 0b0 otherwise.
This event must be implemented if FEAT_ETE is implemented.

0x4012, TRCEXTOUT2, Trace unit external output 2
The event is generated each time an event is signaled by the trace unit external event 2.
It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.
PMCEID0_EL0[50] reads as 0b1 if this event is implemented and 0b0 otherwise.
This event must be implemented if FEAT_ETE is implemented.

0x4013, TRCEXTOUT3, Trace unit external output 3
The event is generated each time an event is signaled by the trace unit external event 3.
It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.
PMCEID0_EL0[51] reads as 0b1 if this event is implemented and 0b0 otherwise.
This event must be implemented if FEAT_ETE is implemented.

0x4018, CTI_TRIGOUT4, Cross-trigger Interface output trigger 4
The event is generated each time an event is signaled on CTI output trigger 4.

Note
CTI output triggers are input events to the PMU and trace unit.

PMCEID0_EL0[56] reads as 0b1 if this event is implemented and 0b0 otherwise.
This event must be implemented if FEAT_ETE is implemented.

0x4019, CTI_TRIGOUT5, Cross-trigger Interface output trigger 5
The event is generated each time an event is signaled on CTI output trigger 5.

Note
CTI output triggers are input events to the PMU and trace unit.

PMCEID0_EL0[57] reads as 0b1 if this event is implemented and 0b0 otherwise.
This event must be implemented if FEAT_ETE is implemented.

0x401A, CTI_TRIGOUT6, Cross-trigger Interface output trigger 6
The event is generated each time an event is signaled on CTI output trigger 6.

Note
CTI output triggers are input events to the PMU and trace unit.

PMCEID0_EL0[58] reads as 0b1 if this event is implemented and 0b0 otherwise.
This event must be implemented if FEAT_ETE is implemented.

0x401B, CTI_TRIGOUT7, Cross-trigger Interface output trigger 7
The event is generated each time an event is signaled on CTI output trigger 7.

Note
CTI output triggers are input events to the PMU and trace unit.

PMCEID0_EL0[59] reads as 0b1 if this event is implemented and 0b0 otherwise.
This event must be implemented if FEAT_ETE is implemented.
0x4020, **LDST_ALIGN_LAT, Access with additional latency from alignment**

The counter counts each access counted by MEM_ACCESS that, due to the alignment of the address and size of data being accessed, incurred additional latency.

PMCEID1_EL0[32] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x4021, **LD_ALIGN_LAT, Load with additional latency from alignment**

The counter counts each Memory-read operation counted by LDST_ALIGN_LAT.
PMCEID1_EL0[33] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x4022, **ST_ALIGN_LAT, Store with additional latency from alignment**

The counter counts each Memory-write operation counted by LDST_ALIGN_LAT.
PMCEID1_EL0[34] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x4024, **MEM_ACCESS_CHECKED, Checked data memory access**

The counter counts each memory access counted by MEM_ACCESS that is checked by the Memory Tagging Extension.
PMCEID1_EL0[36] reads as 0b1 if this event is implemented and 0b0 otherwise.

0x4025, **MEM_ACCESS_RD_CHECKED, Checked data memory access, read**

The counter counts each Memory-read operation counted by MEM_ACCESS_CHECKED.
PMCEID1_EL0[37] reads as 0b1 if this event is implemented and 0b0 otherwise.
Implementation of this optional event requires that FEAT_MTE is implemented.

0x4026, **MEM_ACCESS_WR_CHECKED, Checked data memory access, write**

The counter counts each Memory-write operation counted by MEM_ACCESS_CHECKED.
PMCEID1_EL0[38] reads as 0b1 if this event is implemented and 0b0 otherwise.
Implementation of this optional event requires that FEAT_MTE is implemented.

0x8004, **SIMD_INST_SPEC, Operation speculatively executed, SIMD**

The counter counts each Speculatively executed operation due to either:

- An SVE instruction that is not a non-SIMD SVE instruction.
- An A64 Advanced SIMD instruction that is not an Advanced SIMD scalar instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD operations are counted in AArch32 state.

0x8005, **ASE_INST_SPEC, Operation speculatively executed, Advanced SIMD**

The counter counts each Speculatively executed operation due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED whether the counter counts operations due to Advanced SIMD scalar instructions.

It is IMPLEMENTATION DEFINED which Advanced SIMD operations are counted in AArch32 state.

0x8006, **SVE_INST_SPEC, Operation speculatively executed, SVE, including load and store**

The counter counts each Speculatively executed operation due to an SVE instruction.

It is IMPLEMENTATION DEFINED whether the counter counts operations due to non-SIMD SVE instructions.

This event must be implemented if event SVE_INST_RETIRED is not implemented.
Implementation of this event requires that FEAT_SVE is implemented.

0x8007, **ASE_SVE_INST_SPEC, Operation speculatively executed, Advanced SIMD or SVE**

The counter counts each Speculatively executed operation counted by either ASE_INST_SPEC or SVE_INST_SPEC.

It is IMPLEMENTATION DEFINED which Advanced SIMD operations are counted in AArch32 state.
0x8008, **UOP_SPEC, Microarchitectural operation speculatively executed**

The counter counts each Speculatively executed microarchitectural operation.

The counter counts the operations irrespective of the IMPLEMENTATION DEFINED interpretation of operation speculatively executed.

0x8009, **ASE_UOP_SPEC, Microarchitectural operation speculatively executed, Advanced SIMD**

The counter counts each Speculatively executed microarchitectural operation counted by UOP_SPEC due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED whether the counter counts microarchitectural operations due to Advanced SIMD scalar instructions.

It is IMPLEMENTATION DEFINED which Advanced SIMD microarchitectural operations are counted in AArch32 state.

0x800A, **SVE_UOP_SPEC, Microarchitectural operation speculatively executed, SVE**

The counter counts each Speculatively executed microarchitectural operation counted by UOP_SPEC due to an SVE instruction.

It is IMPLEMENTATION DEFINED whether the counter counts microarchitectural operations due to the instructions listed in Non-SIMD SVE instructions on page D11-5270.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x800B, **ASE_SVE_UOP_SPEC, Microarchitectural operation speculatively executed, Advanced SIMD or SVE**

The counter counts each Speculatively executed microarchitectural operation counted by either ASE_UOP_SPEC or SVE_UOP_SPEC.

It is IMPLEMENTATION DEFINED which Advanced SIMD microarchitectural operations are counted in AArch32 state.

0x800C, **SIMD_UOP_SPEC, Microarchitectural operation speculatively executed, SIMD**

The counter counts each Speculatively executed microarchitectural operation counted by UOP_SPEC due to either:

- An SVE instruction other than any instruction listed in Non-SIMD SVE instructions on page D11-5270.
- An A64 Advanced SIMD instruction other than an Advanced SIMD scalar instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD microarchitectural operations are counted in AArch32 state.

0x800E, **SVE_MATH_SPEC, Operation speculatively executed, SVE math accelerator**

The counter counts each Speculatively executed math function operation due to any of the following A64 instructions:

- SVE: FEXPA, FTMAD, FTSML, or FTSEL.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8010, **FP_SPEC, Floating-point operation speculatively executed, including SIMD**

The counter counts each Speculatively executed floating-point operation due to an A64 scalar, Advanced SIMD, or SVE instruction.

It is IMPLEMENTATION DEFINED whether the counter counts operations due to a scalar, Advanced SIMD, or SVE instruction.

It is IMPLEMENTATION DEFINED which floating-point operations are counted in AArch32 state.

——— Note ————

This event differs from the VFP_SPEC event which does not count SIMD operations.
0x8011, **ASE_FP_SPEC, Floating-point operation speculatively executed, Advanced SIMD**

The counter counts each Speculatively executed floating-point operation counted by FP_SPEC due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD floating-point operations are counted in AArch32 state.

0x8012, **SVE_FP_SPEC, Floating-point operation speculatively executed, SVE**

The counter counts each Speculatively executed floating-point operation counted by FP_SPEC due to an SVE instruction.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8013, **ASE_SVE_FP_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE**

The counter counts each Speculatively executed floating-point operation counted by either ASE_FP_SPEC or SVE_FP_SPEC.

It is IMPLEMENTATION DEFINED which Advanced SIMD floating-point operations are counted in AArch32 state.

0x8014, **FP_HP_SPEC, Floating-point operation speculatively executed, half precision**

The counter counts each Speculatively executed floating-point operation counted by FP_SPEC where the largest type is half precision.

0x8015, **ASE_FP_HP_SPEC, Floating-point operation speculatively executed, Advanced SIMD half precision**

The counter counts each Speculatively executed half-precision floating-point operation counted by FP_HP_SPEC due to an Advanced SIMD instruction.

0x8016, **SVE_FP_HP_SPEC, Floating-point operation speculatively executed, SVE half precision**

The counter counts each Speculatively executed half-precision floating-point operation counted by FP_HP_SPEC due to an SVE instruction.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8017, **ASE_SVE_FP_HP_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE half precision**

The counter counts each Speculatively executed floating-point operation counted by either ASE_FP_HP_SPEC or SVE_FP_HP_SPEC.

0x8018, **FP_SP_SPEC, Floating-point operation speculatively executed, single precision**

The counter counts each Speculatively executed floating-point operation counted by FP_SPEC where the largest type is single precision.

0x8019, **ASE_FP_SP_SPEC, Floating-point operation speculatively executed, Advanced SIMD single precision**

The counter counts each Speculatively executed single-precision floating-point operation counted by FP_SP_SPEC due to an Advanced SIMD instruction.

0x801A, **SVE_FP_SP_SPEC, Floating-point operation speculatively executed, SVE single precision**

The counter counts each Speculatively executed single-precision floating-point operation counted by FP_SP_SPEC due to an SVE instruction.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x801B, **ASE_SVE_FP_SP_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE single precision**

The counter counts each Speculatively executed floating-point operation counted by either ASE_FP_SP_SPEC or SVE_FP_SP_SPEC.
The Performance Monitors Extension

D11.11 PMU events and event numbers

0x801C, FP_DP_SPEC, Floating-point operation speculatively executed, double precision
The counter counts each Speculatively executed floating-point operation counted by FP_SPEC where the largest type is double precision.

0x801D, ASE_FP_DP_SPEC, Floating-point operation speculatively executed, Advanced SIMD double precision
The counter counts each Speculatively executed double-precision floating-point operation counted by FP_DP_SPEC due to an Advanced SIMD instruction.

0x801E, SVE_FP_DP_SPEC, Floating-point operation speculatively executed, SVE double precision
The counter counts each Speculatively executed double-precision floating-point operation counted by FP_DP_SPEC due to an SVE instruction.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x801F, ASE_SVE_FP_DP_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE double precision
The counter counts each Speculatively executed floating-point operation counted by either ASE_FP_DP_SPEC or SVE_FP_DP_SPEC.

0x8020, FP_DIV_SPEC, Floating-point operation speculatively executed, divide
The counter counts each Speculatively executed floating-point divide operation counted by FP_SPEC due to any of the following A64 instructions:
• Scalar: FDIV.
• Advanced SIMD: FDIIV.
• SVE: FDIV or FDIVR.
It is IMPLEMENTATION DEFINED which floating-point divide operations are counted in AArch32 state.

0x8021, ASE_FP_DIV_SPEC, Floating-point operation speculatively executed, Advanced SIMD divide
The counter counts each Speculatively executed floating-point divide operation counted by FP_DIV_SPEC due to any of the following A64 instructions:
• Advanced SIMD: FDIIV.
It is IMPLEMENTATION DEFINED which floating-point divide operations are counted in AArch32 state.

0x8022, SVE_FP_DIV_SPEC, Floating-point operation speculatively executed, SVE divide
The counter counts each Speculatively executed floating-point divide operation counted by FP_DIV_SPEC due to any of the following instructions:
• SVE: FDIIV or FDIVR.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x8023, ASE_SVE_FP_DIV_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE divide
The counter counts each Speculatively executed floating-point divide operation counted by either ASE_FP_DIV_SPEC or SVE_FP_DIV_SPEC.
That is, due to any of the following A64 instructions:
• Advanced SIMD: FDIIV.
• SVE: FDIIV or FDIVR.
It is IMPLEMENTATION DEFINED which floating-point divide operations are counted in AArch32 state.
The counter counts each Speculatively executed floating-point square-root operation counted by \texttt{FP\_SPEC} due to any of the following A64 instructions:

- Scalar: \texttt{FSQRT}.
- Advanced SIMD: \texttt{FSQRT}.
- SVE: \texttt{FSQRT}.

It is IMPLEMENTATION DEFINED which floating-point square-root operations are counted in AArch32 state.

The counter counts each Speculatively executed floating-point square-root operation counted by \texttt{FP\_SPEC} due to any of the following A64 instructions:

- Advanced SIMD: \texttt{FSQRT}.

It is IMPLEMENTATION DEFINED which floating-point square-root operations are counted in AArch32 state.

The counter counts each Speculatively executed floating-point square-root operation counted by \texttt{FP\_SPEC} due to any of the following A64 instructions:

- SVE: \texttt{FSQRT}.

Implementation of this optional event requires that FEAT\_SVE is implemented.

The counter counts each Speculatively executed floating-point square-root operation counted by \texttt{FP\_SPEC} due to any of the following A64 instructions:

- Advanced SIMD: \texttt{FSQRT}.
- SVE: \texttt{FSQRT}.

That is, due to any of the following A64 instructions:

- Advanced SIMD: \texttt{FSQRT}.
- SVE: \texttt{FSQRT}.

It is IMPLEMENTATION DEFINED which floating-point square-root operations are counted in AArch32 state.

The counter counts each Speculatively executed floating-point fused multiply-add or multiply-subtract operation counted by \texttt{FP\_SPEC} due to any of the following A64 instructions:

- Scalar: \texttt{FMADD}, \texttt{FMSUB}, \texttt{FNMADD}, or \texttt{FNMSUB}.
- Advanced SIMD: \texttt{BFMLALB}, \texttt{BFMLALT}, \texttt{FCMLA}, \texttt{FMLA}, or \texttt{FMLS}.
- SVE: \texttt{BFMLALB (vectors)}, \texttt{BFMLALT (vectors)}, \texttt{FCMLA (vectors)}, \texttt{FMAD}, \texttt{FMLA (vectors)}, \texttt{FMLS (vectors)}, \texttt{FMAD}, \texttt{FMAL}, \texttt{FMLS}, \texttt{FMFSB}, or \texttt{FTMAD}.
- SVE2: \texttt{BFMLALB (vectors)}, \texttt{BFMLALT (vectors)}, \texttt{FMLAL (vectors)}, \texttt{FMLALT (vectors)}, \texttt{FMLS (vectors)}, \texttt{FMLAL (vectors)}, or \texttt{FMLSLT (vectors)}.

It is IMPLEMENTATION DEFINED which floating-point fused multiply-add or multiply-subtract operations are counted in AArch32 state.

The counter counts each Speculatively executed floating-point fused multiply-add or multiply-subtract operation counted by \texttt{FP\_SPEC} due to any of the following A64 instructions:

- Advanced SIMD: \texttt{BFMLALB}, \texttt{BFMLALT}, \texttt{FCMLA}, \texttt{FMLA}, or \texttt{FMLS}.

It is IMPLEMENTATION DEFINED which floating-point fused multiply-add or multiply-subtract operations are counted in AArch32 state.
The counter counts each Speculatively executed floating-point fused multiply-add or multiply-subtract operation counted by FP_FMA_SPEC due to any of the following instructions:

- SVE: BFMLALB (vectors), BFMLALT (vectors), FCMLA (vectors), FMAD, FMLA (vectors), FMLS (vectors), FMSB, FNMAD, FMCLA, FMULS, FMULSLB, or FTMAD.
- SVE2: BFMLALB (vectors), BFMLALT (vectors), FMLALB (vectors), FMLALT (vectors), FMLSLB (vectors), or FMLSLT (vectors).

Implementation of this optional event requires that FEAT_SVE is implemented.

The counter counts each Speculatively executed floating-point fused multiply-add or multiply-subtract operation counted by either ASE_FP_FMA_SPEC or SVE_FP_FMA_SPEC.

That is, due to any of the following A64 instructions:

- Advanced SIMD: BFMLALB, BFMLALT, FCMLA, FMLA, or FMLS.
- SVE: BFMLALB (vectors), BFMLALT (vectors), FCMLA (vectors), FMAD, FMLA (vectors), FMLS (vectors), FMSB, FNMAD, FMCLA, FMULS, FMULSLB, or FTMAD.
- SVE2: BFMLALB (vectors), BFMLALT (vectors), FMLALB (vectors), FMLALT (vectors), FMLSLB (vectors), or FMLSLT (vectors).

It is IMPLEMENTATION DEFINED which floating-point fused multiply-add or multiply-subtract operations are counted in AArch32 state.

The counter counts each Speculatively executed floating-point multiply operation counted by FP_MUL_SPEC due to any of the following A64 instructions:

- Scalar: FMUL or FMULX.
- Advanced SIMD: FMUL or FMULX.
- SVE: FMUL, FMULX, or FTSMUL.

It is IMPLEMENTATION DEFINED which floating-point multiply operations are counted in AArch32 state.

The counter counts each Speculatively executed floating-point multiply operation counted by either ASE_FP_MUL_SPEC or SVE_FP_MUL_SPEC.

That is, due to any of the following A64 instructions:

- Advanced SIMD: FMUL or FMULX.
- SVE: FMUL, FMULX, or FTSMUL.

Implementation of this optional event requires that FEAT_SVE is implemented.

The counter counts each Speculatively executed floating-point fused multiply-add or multiply-subtract operation counted by either ASE_FP_MUL_SPEC or SVE_FP_MUL_SPEC.

That is, due to any of the following A64 instructions:

- Advanced SIMD: FMUL or FMULX.
- SVE: FMUL, FMULX, or FTSMUL.
It is IMPLEMENTATION DEFINED which floating-point fused multiply-add or multiply-subtract operations are counted in AArch32 state.

0x8030, **FP_ADDSUB_SPEC, Floating-point operation speculatively executed, add or subtract**

The counter counts each Speculatively executed floating-point add or subtract operation counted by FP_SPEC due to any of the following A64 instructions:

- Scalar: FADD or FSUB.
- Advanced SIMD: FADD, FADD, or FSUB.
- SVE: FABD, FADD, FSUB, or FSUBR.

It is IMPLEMENTATION DEFINED which floating-point add or subtract operations are counted in AArch32 state.

0x8031, **ASE_FP_ADDSUB_SPEC, Floating-point operation speculatively executed, Advanced SIMD add or subtract**

The counter counts each Speculatively executed floating-point add or subtract operation counted by FP_ADDSUB_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FABD, FADD, or FSUB.

It is IMPLEMENTATION DEFINED which floating-point add or subtract operations are counted in AArch32 state.

0x8032, **SVE_FP_ADDSUB_SPEC, Floating-point operation speculatively executed, SVE add or subtract**

The counter counts each Speculatively executed floating-point add or subtract operation counted by FP_ADDSUB_SPEC due to any of the following instructions:

- SVE: FABD, FADD, FSUB, or FSUBR.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8033, **ASE_SVE_FP_ADDSUB_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE add or subtract**

The counter counts each Speculatively executed floating-point add or subtract operation counted by either ASE_FP_ADDSUB_SPEC or SVE_FP_ADDSUB_SPEC.

That is, due to any of the following A64 instructions:

- Advanced SIMD: FABD, FADD, or FSUB.
- SVE: FABD, FADD, FSUB, or FSUBR.

It is IMPLEMENTATION DEFINED which floating-point add or subtract operations are counted in AArch32 state.

0x8034, **FP_RECPE_SPEC, Floating-point operation speculatively executed, reciprocal estimate**

The counter counts each Speculatively executed floating-point reciprocal estimate operation counted by FP_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FRECPE or FRSQRT.
- SVE: FRECPE or FRSQRT.

It is IMPLEMENTATION DEFINED which floating-point reciprocal estimate operations are counted in AArch32 state.

0x8035, **ASE_FP_RECPE_SPEC, Floating-point operation speculatively executed, Advanced SIMD reciprocal estimate**

The counter counts each Speculatively executed floating-point reciprocal estimate operation counted by FP_RECPE_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FRECPE or FRSQRT.

It is IMPLEMENTATION DEFINED which floating-point reciprocal estimate operations are counted in AArch32 state.
0x8036, SVE_FP_RECPE_SPEC, Floating-point operation speculatively executed, SVE reciprocal estimate
The counter counts each Speculatively executed floating-point reciprocal estimate operation counted by FP_RECPE_SPEC due to any of the following instructions:
- SVE: FRECPE or FRSQRT.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8037, ASE_SVE_FP_RECPE_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE reciprocal estimate
The counter counts each Speculatively executed floating-point reciprocal estimate operation counted by either ASE_FP_RECPE_SPEC or SVE_FP_RECPE_SPEC.
That is, due to any of the following A64 instructions:
- Advanced SIMD: FRECPE or FRSQRT.
- SVE: FRECPE or FRSQRT.

It is IMPLEMENTATION DEFINED which floating-point reciprocal estimate operations are counted in AArch32 state.

0x8038, FP_CVT_SPEC, Floating-point operation speculatively executed, convert
The counter counts each Speculatively executed floating-point convert operation due to an A64 scalar, Advanced SIMD, or SVE floating-point conversion instruction.
It is IMPLEMENTATION DEFINED which floating-point convert operations are counted in AArch32 state.

0x8039, ASE_FP_CVT_SPEC, Floating-point operation speculatively executed, Advanced SIMD convert
The counter counts each Speculatively executed floating-point convert operation counted by FP_CVT_SPEC due to an A64 Advanced SIMD instruction.
It is IMPLEMENTATION DEFINED which Advanced SIMD floating-point convert operations are counted in AArch32 state.

0x803A, SVE_FP_CVT_SPEC, Floating-point operation speculatively executed, SVE convert
The counter counts each Speculatively executed floating-point convert operation counted by FP_CVT_SPEC due to an SVE instruction.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x803B, ASE_SVE_FP_CVT_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE convert
The counter counts each Speculatively executed floating-point convert operation counted by either ASE_FP_CVT_SPEC or SVE_FP_CVT_SPEC.
It is IMPLEMENTATION DEFINED which Advanced SIMD floating-point convert operations are counted in AArch32 state.

0x803C, SVE_FP_AREDUCE_SPEC, Floating-point operation speculatively executed, SVE accumulating reduction
The counter counts each Speculatively executed floating-point accumulating reduction operation counted by SVE_FP_SPEC due to any of the following A64 instructions:
- SVE: FADD.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x803D, ASE_FP_PREDUCE_SPEC, Floating-point operation speculatively executed, Advanced SIMD pairwise add step
The counter counts each Speculatively executed floating-point pairwise add operation counted by ASE_FP_SPEC due to any of the following A64 instructions:
- Advanced SIMD: FADD.
It is IMPLEMENTATION DEFINED which floating-point pairwise add operations are counted in AArch32 state.

0x803E, SVE_FP_VREDUCE_SPEC, Floating-point operation speculatively executed, SVE vector reduction

The counter counts each Speculatively executed floating-point tree-wise reduction operation counted by SVE_FP_SPEC due to any of the following A64 instructions:
- SVE: FADDV, FMAXNW, FMAXV, FMINNW, or FMINV.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x803F, ASE_SVE_FP_VREDUCE_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE vector reduction

The counter counts each Speculatively executed floating-point reduction operation counted by ASE_SVE_FP_SPEC due to any of the following A64 instructions:
- Advanced SIMD: FADDP, FMAXNW, FMAXV, FMINNW, or FMINV.
- SVE: FADDV, FMAXNW, FMAXV, FMINNW, or FMINV.

It is IMPLEMENTATION DEFINED which floating-point reduction operations are counted in AArch32 state.

0x8040, INT_SPEC, Integer operation speculatively executed

The counter counts each Speculatively executed integer arithmetic operation due to an A64 scalar, Advanced SIMD, and SVE data-processing instruction.

It is IMPLEMENTATION DEFINED whether the counter counts operations due to a scalar, Advanced SIMD, and SVE data-processing instruction.

It is IMPLEMENTATION DEFINED which integer arithmetic operations are counted in AArch32 state.

0x8041, ASE_INT_SPEC, Integer operation speculatively executed, Advanced SIMD

The counter counts each Speculatively executed integer operation counted by INT_SPEC due to an Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD integer operations are counted in AArch32 state.

0x8042, SVE_INT_SPEC, Integer operation speculatively executed, SVE

The counter counts each Speculatively executed integer operation counted by INT_SPEC due to an SVE instruction.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8043, ASE_SVE_INT_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE

The counter counts each Speculatively executed integer arithmetic operation counted by either ASE_INT_SPEC or SVE_INT_SPEC.

It is IMPLEMENTATION DEFINED which Advanced SIMD integer operations are counted in AArch32 state.

0x8044, INT_DIV_SPEC, Integer operation speculatively executed, divide

The counter counts each Speculatively executed integer divide operation counted by INT_SPEC due to any of the following A64 instructions:
- Scalar: SDIV or UDIV.
- SVE: SDIV, SDIVR, UDIV, or UDIVR.

It is IMPLEMENTATION DEFINED which integer divide operations are counted in AArch32 state.

0x8045, INT_DIV64_SPEC, Integer operation speculatively executed, 64-bit divide

The counter counts each Speculatively executed integer divide operation counted by INT_SPEC due to any of the following A64 instructions:
- Scalar: SDIV or UDIV.
The Performance Monitors Extension
D11.11 PMU events and event numbers

- SVE: SDIV, SDIVR, UDIV, or UDIVR.
The counter only counts operations with 64-bit operands or vector elements.

0x8046, SVE_INT_DIV_SPEC, Integer operation speculatively executed, SVE divide
The counter counts each Speculatively executed integer divide operation counted by INT_DIV_SPEC due to any of the following instructions:
- SVE: SDIV, SDIVR, UDIV, or UDIVR.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x8047, SVE_INT_DIV64_SPEC, Integer operation speculatively executed, SVE 64-bit divide
The counter counts each Speculatively executed integer 64-bit divide operation counted by INT_DIV64_SPEC due to any of the following instructions:
- SVE: SDIV, SDIVR, UDIV, or UDIVR.
The counter only counts operations with 64-bit operands or vector elements.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x8048, INT_MUL_SPEC, Integer operation speculatively executed, multiply
The counter counts each Speculatively executed integer multiply or multiply-accumulate operation counted by INT_SPEC due to any of the following A64 instructions:
- Scalar: MADD, MSUB, MUL, SMADD, SMULH, UMAADD, or UMLH.
- Advanced SIMD: MLA, MLS, MUL, PMULL, SMLAL, SMLS, SMULL, SQDMULAL, SQDMULSL, SQDMULH, SQDMULL, SQRDMULAH, SQRDMULSH, SQ-shadow, or UMLH.
- SVE: MAD, MIA, MLS, MSB, ML, SMULH, or UMLH.
- SVE2: CMLA, MLA, MLS, MUL, PMULL, SMLAL, SMLS, SMULL, SQDMULAL, SQDMULSL, SQDMULH, SQDMULL, SQRDMULAH, SQRDMULSH, SQRDMULH, UMLH, or UMLH.

It is IMPLEMENTATION DEFINED which integer multiply or multiply-accumulate operations are counted in AArch32 state.

0x8049, ASE_INT_MUL_SPEC, Integer operation speculatively executed, Advanced SIMD multiply
The counter counts each Speculatively executed integer multiply or multiply-accumulate operation counted by INT_MUL_SPEC due to any of the following A64 instructions:
- Advanced SIMD: MLA, MLS, MUL, PMULL, SMLAL, SMLS, SMULL, SQDMULAL, SQDMULSL, SQDMULH, SQDMULL, SQRDMULAH, SQRDMULSH, SQRDMULH, UMLH, or UMLH.

It is IMPLEMENTATION DEFINED which integer multiply or multiply-accumulate operations are counted in AArch32 state.

0x804A, SVE_INT_MUL_SPEC, Integer operation speculatively executed, SVE multiply
The counter counts each Speculatively executed integer multiply or multiply-accumulate operation counted by INT_MUL_SPEC due to any of the following instructions:
- SVE: MAD, MIA, MLS, MSB, ML, SMULH, or UMLH.
- SVE2: CMLA, MLA, MLS, MUL, PMULL, SMLAL, SMLS, SMULL, SQDMULAL, SQDMULSL, SQDMULH, SQDMULL, SQRDMULAH, SQRDMULSH, SQRDMULH, UMLH, or UMLH.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x804B, ASE_SVE_INT_MUL_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE multiply
The counter counts each Speculatively executed integer multiply or multiply-accumulate operation counted by either ASE_INT_MUL_SPEC or SVE_INT_MUL_SPEC.
That is, due to any of the following A64 instructions:

- Advanced SIMD: MLA, MLS, MUL, PMULL, SMLAL, SMLS, SMLSL, SMULL, SQDMLAL, SQDMLSL, SQDMLUL, SQDMLULL, SQDMLSH, SQDMLSLH, SQDMLUH, UMLAL, UMLSL, or UMLUL.
- SVE: MAD, MLA, MLS, MSB, MUL, SMULH, or UMLUL.
- SVE2: CMLA, MLA, MLS, MUL, SMLALB, SMLALT, SMLSLB, SMLSLT, SMULH, SMULLB, SMULLT, SQDMLALB, SQDMLALBT, SQDMLALT, SQDMLSLB, SQDMLSLBT, SQDMLSLT, SQDMLUH, SQDMLULL, SQDMLULLT, SQDMLUH, SQRDMULH, SQRDMULLH, SQDMLUH, UMLALB, UMLALT, UMLSLB, UMLSLT, UMLUL, or UMLULL.

It is IMPLEMENTATION DEFINED which integer multiply or multiply-accumulate operations are counted in AArch32 state.

0x804C, **INT_MUL64_SPEC**, Integer operation speculatively executed, 64×64 multiply

The counter counts each Speculatively executed 64×64 integer multiply operation counted by INT_SPEC due to any of the following A64 instructions:

- Scalar: MADD, MSUB, MUL, or SMULH.
- SVE: MAD, MLA, MLS, MSB, MUL, SMULH, or UMLUL.
- SVE2: CMLA (vectors), MLA, MLS, MUL, SMLALB, SMLALT, SMLSLB, SMLSLT, SMULH, SMULLB, SMULLT, SQDMLALB, SQDMLALBT, SQDMLALT, SQDMLSLB, SQDMLSLBT, SQDMLSLT, SQDMLUH, SQDMLULL, SQDMLULLT, SQDMLUH, SQRDMULH (vectors), SQDMLAH, SQDMLSH, SQRDMULH, UMLALB, UMLALT, UMLSLB, UMLSLT, UMLUL, or UMLULL.

The counter only counts operations that perform a 64-bit × 64-bit integer multiply operations.

0x804D, **SVE_INT_MUL64_SPEC**, Integer operation speculatively executed, SVE 64×64 multiply

The counter counts each Speculatively executed 64×64 integer multiply operation counted by INT_MUL64_SPEC due to any of the following instructions:

- SVE: MAD, MLA, MLS, MSB, MUL, SMULH, or UMLUL.
- SVE2: CMLA (vectors), MLA, MLS, MUL, SMLALB, SMLALT, SMLSLB, SMLSLT, SMULH, SMULLB, SMULLT, SQDMLALB, SQDMLALBT, SQDMLALT, SQDMLSLB, SQDMLSLBT, SQDMLSLT, SQDMLUH, SQDMLULL, SQDMLULLT, SQDMLUH, SQRDMULH (vectors), SQDMLAH, SQDMLSH, SQRDMULH, UMLALB, UMLALT, UMLSLB, UMLSLT, UMLUL, or UMLULL.

The counter only counts operations that perform a 64-bit × 64-bit integer multiply operations. Implementation of this optional event requires that FEAT_SVE is implemented.

0x804E, **INT_MULH64_SPEC**, Integer operation speculatively executed, 64×64 multiply returning high part

The counter counts each Speculatively executed widening 64×64 integer multiply operation counted by INT_SPEC due to any of the following A64 instructions:

- Scalar: SMULH or UMLUL.
- SVE: SMULH or UMLUL.
- SVE2: SMULH, SQDMLUL, SQRDMULH, or UMLUL.

These instructions perform 64-bit × 64-bit integer multiply operations.

0x804F, **SVE_INT_MULH64_SPEC**, Integer operation speculatively executed, SVE 64×64 multiply high part

The counter counts each Speculatively executed 64×64 integer multiply returning high part operation counted by INT_MULH64_SPEC due to any of the following instructions:

- SVE: SMULH or UMLUL.
- SVE2: SMULH, SQDMLUL, SQRDMULH, or UMLUL.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8058, **NONFP_SPEC**, Non-floating-point operation speculatively executed

The counter counts each Speculatively executed operation due to any of:

- A scalar instruction that would be counted by the DP_SPEC event.
**The Performance Monitors Extension**

**D11.11 PMU events and event numbers**

- An A64 Advanced SIMD data processing instruction defined in the section titled *Data processing - SIMD and floating-point* on page C3-310 that would not be counted by *FP_SPEC*.
- An SVE instruction with vector source or destination registers, that would not be counted by *FP_SPEC*.

It is IMPLEMENTATION DEFINED which non-floating-point data processing operations are counted in AArch32 state.

0x8059, **ASE_NONFP_SPEC, Non-floating-point operation speculatively executed, Advanced SIMD**

The counter counts each Speculatively executed non-floating-point operation counted by NONFP_SPEC due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD non-floating-point data processing operations are counted in AArch32 state.

0x805A, **SVE_NONFP_SPEC, Non-floating-point operation speculatively executed, SVE**

The counter counts each Speculatively executed non-floating-point operation counted by NONFP_SPEC due to an SVE instruction.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x805B, **ASE_SVE_NONFP_SPEC, Non-floating-point operation speculatively executed, Advanced SIMD or SVE**

The counter counts each Speculatively executed non-floating-point operation counted by either ASE_NONFP_SPEC or SVE_NONFP_SPEC.

It is IMPLEMENTATION DEFINED which Advanced SIMD non-floating-point data processing operations are counted in AArch32 state.

0x805D, **ASE_INT_VREDUCE_SPEC, Integer operation speculatively executed, Advanced SIMD reduction**

The counter counts each Speculatively executed across-vector and pairwise integer reduction operation counted by ASE_INT_SPEC due to any of the following A64 instructions:

- Advanced SIMD: SADDLP, SADDLV, SMAXP, SMAXV, SMINP, SMINV, UADDVL, UMAXV, or UMINV.

It is IMPLEMENTATION DEFINED which across-vector and pairwise integer reduction operations are counted in AArch32 state.

0x805E, **SVE_INT_VREDUCE_SPEC, Integer operation speculatively executed, SVE reduction**

The counter counts each Speculatively executed across-vector integer reduction operation counted by SVE_INT_SPEC due to any of the following A64 instructions:

- SVE: ANDV, EORV, ORV, SADDV, SMAXV, SMINV, UADDV, UMAXV, or UMINV.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x805F, **ASE_SVE_INT_VREDUCE_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE reduction**

The counter counts each Speculatively executed across-vector and pairwise integer reduction operation counted by either ASE_INT_VREDUCE_SPEC or SVE_INT_VREDUCE_SPEC.

That is, due to any of the following A64 instructions:

- Advanced SIMD: SADDLP, SADDLV, SMAXP, SMAXV, SMINP, SMINV, UADDVL, UMAXV, or UMINV.
- SVE: ANDV, EORV, ORV, SADDV, SMAXV, SMINV, UADDV, UMAXV, or UMINV.

It is IMPLEMENTATION DEFINED which across-vector and pairwise integer reduction operations are counted in AArch32 state.
0x8060, SVE_PERM_SPEC, Operation speculatively executed, SVE permute

The counter counts each Speculatively executed vector or predicate permute operation due to any of the following A64 instructions:

- SVE: CLASTA, CLASTB, COMPACT, CPY (SIMD&FP scalar), CPY (scalar), DUP (indexed), DUP (scalar), EXT, INSR, LASTA, LASTB, PUNPKHI, PUNPKLO, REV (vector), REV8, REVH, REW, SPLICE, SUNPKHI, SUNPKLO, TBL, TRN1 (vectors), TRN2 (vectors), UUNPKHI, UUNPKLO, UZP1 (vectors), UZP2 (vectors), ZIP1 (vectors), or ZIP2 (vectors).

- SVE2: EXT, SPLICE, TBL, TBX, TRN1, TRN2, UZP1, UZP2, ZIP1, or ZIP2.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8061, SVE_PERM_VARIABLE_SPEC, Operation speculatively executed, SVE intra-granule permute

The counter counts each Speculatively executed vector or predicate permute operation within a 128-bit vector granule or 16-bit predicate granule due to any of the following A64 instructions:

- SVE: REV8, REVH, REW, TRN1 (vectors), or TRN2 (vectors).

- SVE2: TRN1 or TRN2.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8062, SVE_PERM_XGRANULE_SPEC, Operation speculatively executed, SVE cross-granule permute

The counter counts each Speculatively executed vector or predicate permute operation that can cross between 128-bit vector granules or 16-bit predicate granules due to any of the following A64 instructions:

- SVE: CLASTA, CLASTB, COMPACT, CPY (SIMD&FP scalar), CPY (scalar), DUP (indexed), DUP (scalar), EXT, INSR, LASTA, LASTB, PUNPKHI, PUNPKLO, REV (vector), SPLICE, SUNPKHI, SUNPKLO, TBL, UUNPKHI, UUNPKLO, UZP1 (vectors), UZP2 (vectors), ZIP1 (vectors), or ZIP2 (vectors).

- SVE2: EXT, SPLICE, TBL, TBX, UZP1, UZP2, ZIP1, or ZIP2.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8063, SVE_PERM_VARIABLE_SPEC, Operation speculatively executed, SVE programmable permute

The counter counts each Speculatively executed variable vector permute operation due to any of the following A64 instructions:

- SVE: CLASTA, CLASTB, COMPACT, LASTA, LASTB, SPLICE, or TBL.

- SVE2: SPLICE or TBL.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8064, SVE_XPIPE_SPEC, Operation speculatively executed, SVE cross-pipe

The counter counts each Speculatively executed cross-pipeline transfer operation counted by either SVE_XPIPE_RZ2_SPEC or SVE_XPIPE_Z2R_SPEC.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8065, SVE_XPIPE_Z2R_SPEC, Operation speculatively executed, SVE vector to scalar cross-pipe

The counter counts each Speculatively executed vector to general-purpose scalar cross-pipeline transfer operation due to any of the following A64 instructions:

- SVE: CLASTA (scalar), CLASTB (scalar), CNTP, DECP (scalar), INCP (scalar), LASTA (scalar), LASTB (scalar), SQDECP (scalar), SQINCP (scalar), UQDECP (scalar), or UQINCP (scalar).

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8066, SVE_XPIPE_RZ2_SPEC, Operation speculatively executed, SVE scalar to vector cross-pipe

The counter counts each Speculatively executed general-purpose scalar to vector cross-pipeline transfer operation due to any of the following A64 instructions:

- SVE: CPY (scalar), DUP (scalar), INDEX (immediate, scalar), INDEX (scalar, immediate), INDEX (scalars), INSR (scalar), WHILEE, WHILELO, WHILELS, or WHILELT.

- SVE2: WHILEGE, WHILEGT, WHILEHI, WHILEHS, WHILERQ, or WHILEWR.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x8067, SVE_PGEN_NVEC_SPEC, Operation speculatively executed, SVE predicate-only

The counter counts each Speculatively executed predicate-generating operation that does not read vector registers due to any of the following A64 instructions:

- SVE: AND (predicates), ANDS, BIC (predicates), BICS, BRKA, BRKAS, BRKB, BRKBS, BRKN, BRKNS, BRKPA, BRKPAS, BRKPB, BRKPBPS, EOR (predicates), EORS, NAND, NANDS, NOR, NORS, ORN (predicates), ORNS, ORR (predicates), ORRS, PFALSE, PFIRST, PNEXT, PTRUE, PTRUEES, PUNPKHI, PUNPKLO, RDFR, RDFFRS, REV (predicate), SEL (predicates), TRN1 (predicates), TRN2 (predicates), UZP1 (predicates), UZP2 (predicates), ZIP1 (predicates), or ZIP2 (predicates).

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8068, SVE_PGEN_SPEC, Operation speculatively executed, SVE predicate generating

The counter counts each Speculatively executed predicate-generating operation due to any of the following A64 instructions:

- SVE: AND (predicates), ANDS, BIC (predicates), BICS, BRKA, BRKAS, BRKB, BRKBS, BRKN, BRKNS, BRKPA, BRKPAS, BRKPB, BRKPBPS, CMPEQ, CMPGE, CMPGT, CMPHI, CMPSH, CMPLE (immediate), CMPLE (wide elements), CMPL0 (immediate), CMPL0 (wide elements), CMPLS (immediate), CMPLS (wide elements), CMPLT (immediate), CMPLT (wide elements), CMPNE, EOR (predicates), EORS, FACGE, FACGT, FCMEE, FCOME, FCOME (zero), FCMT (zero), FCME, FCMUO, NAND, NANDS, NOR, NORS, ORN (predicates), ORNS, ORR (predicates), ORRS, PFALSE, PFIRST, PNEXT, PTRUE, PTRUEES, PUNPKHI, PUNPKLO, RDFR, RDFFRS, REV (predicate), SEL (predicates), TRN1 (predicates), TRN2 (predicates), UZP1 (predicates), UZP2 (predicates), WHILELE, WHILELO, WHILELS, WHILELT, ZIP1 (predicates), or ZIP2 (predicates).

- SVE2: MATCH, NMATCH, WHILEGE, WHILEGT, WHILEHS, WHILEHW, or WHILELR.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8069, SVE_PGEN_FLG_SPEC, Operation speculatively executed, SVE predicate flag setting

The counter counts each Speculatively executed predicate-generating operation that sets condition flags due to any of the following A64 instructions:

- SVE: ANDS, BICS, BRKAS, BRKBS, BRKN, BRKNS, BRKPA, BRKPAS, BRKPB, BRKPBPS, CMPEQ, CMPGE, CMPGT, CMPHI, CMPSH, CMPLE (immediate), CMPL0 (immediate), CMPL0 (wide elements), CMPL0 (immediate), CMPL0 (wide elements), CMPLS (immediate), CMPLT (immediate), CMPLT (wide elements), CMPNE, EORS, NANDS, NORS, ORNS, ORRS, PFALSE, PFIRST, PNEXT, PTRUE, PTRUEES, RDFR, RDFFRS, WHILELE, WHILELO, WHILELS, or WHILELT.

- SVE2: MATCH, NMATCH, WHILEGE, WHILEGT, WHILEHS, WHILERW, or WHILERW.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x806A, SVE_PGEN_CMP_SPEC, Operation speculatively executed, SVE vector compare

The counter counts each Speculatively executed vector compare operation due to any of the following A64 instructions:

- SVE: CMPEQ, CMPGE, CMPGT, CMPHI, CMPSH, CMPLE (immediate), CMPL0 (immediate), CMPL0 (wide elements), CMPL0 (immediate), CMPLT (immediate), CMPLT (wide elements), CMPNE, FACGE, FACGT, FCOME, FCOME, FCOME (zero), FCMT (zero), FCME, FCMUO.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x806B, SVE_PGEN_FCM_SPEC, Floating-point operation speculatively executed, SVE vector compare

The counter counts each Speculatively executed vector floating-point compare operation due to any of the following A64 instructions:

- SVE: FACGE, FACGT, FCOME, FCOME, FCOME (zero), FCMT (zero), FCME, or FCMUO.

Implementation of this optional event requires that FEAT_SVE is implemented.
0x806C, SVE_PGEN_LOGIC_SPEC, Operation speculatively executed, SVE predicate logical
The counter counts each Speculatively executed predicate logical operation due to any of the following A64 instructions:

- SVE: AND (predicates), ANDS, BIC (predicates), BICS, EOR (predicates), EORS, NAND, NANDS, NOR, NOR, ORN (predicates), ORNS, ORR (predicates), or ORRS.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x806D, SVE_PPERM_SPEC, Operation speculatively executed, SVE predicate permute
The counter counts each Speculatively executed predicate permute operation due to any of the following A64 instructions:

- SVE: PUNPKHI, PUNPKLO, REV (predicate), TRN1 (predicate), TRN2 (predicate), UZP1 (predicate), ZIP2 (predicate), ZIP2 (predicate), UZP2 (predicate), or ZIP2 (predicate).

Implementation of this optional event requires that FEAT_SVE is implemented.

0x806E, SVE_PSCAN_SPEC, Operation speculatively executed, SVE predicate scan
The counter counts each Speculatively executed predicate scanning and generation operation due to any of the following A64 instructions:

- SVE: BRKA, BRKAS, BRKB, BRKBS, BRKN, BRKPA, BRKPS, BRKPB, BRKPBS, PFIRST, or PNEXT.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x806F, SVE_PCNT_SPEC, Operation speculatively executed, SVE predicate count
The counter counts each Speculatively executed predicate population count operation due to any of the following A64 instructions:

- SVE: CNTP, DECP, INCP, SQDECP, SQINCP, UQDECP, or UQINCP.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8070, SVE_PLOOP_WHILE_SPEC, Operation speculatively executed, SVE predicate loop while
The counter counts each Speculatively executed counted predicate generation operation due to any of the following A64 instructions:

- SVE: WHILELE, WHILELO, WHILELS, or WHILELT.
- SVE2: WHILEGE, WHILEGT, WHILEHI, or WHILEHS.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8071, SVE_PLOOP_TEST_SPEC, Operation speculatively executed, SVE predicate loop test
The counter counts each Speculatively executed loop predicate test operation due to any of the following A64 instructions:

- SVE: BRKAS, BRKBS, BRKN, BRKPA, BRKPS, BRKPB, BRKPBS, WHILELE, WHILELO, WHILELS, or WHILELT.
- SVE2: WHILEGE, WHILEGT, WHILEHI, or WHILEHS.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8072, SVE_PLOOP_ELTS_SPEC, Operation speculatively executed, SVE predicate loop elements
The counter counts each Speculatively executed loop predicate generation operation due to any of the following A64 instructions:

- SVE: WHILELE, WHILELO, WHILELS, or WHILELT.
- SVE2: WHILEGE, WHILEGT, WHILEHI, or WHILEHS.

The counter increments by (128 ÷ CSIZE).

--- Note ---

Multiplying the counter value by (VL ÷ 128) determines the number of vector elements speculatively processed by while loops.

Implementation of this optional event requires that FEAT_SVE is implemented.
0x8073, **SVE_PLOOP_TERM_SPEC, Operation speculatively executed, SVE predicate loop termination**

The counter counts each Speculatively executed loop-terminating predicate generation operation due to any of:

- An SVE **WHILEL**, **WHILES**, or **WHILELT** instruction which sets PSTATE.N to \(0b0\).
- An SVE **BRKAS**, **BRKBS**, **BRKNS**, **BRKPAS**, or **BRKPBS** instruction which sets PSTATE.C to \(0b1\).
- An SVE **CTERMEQ** or **CTERMNE** instruction which sets PSTATE.N to \(0b1\) and PSTATE.V to \(0b0\).
- An SVE **WHILEGE**, **WHILEGT**, **WHILEHL**, or **WHILEHS** instruction which sets PSTATE.N to \(0b0\).

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8074, **SVE_PRED_SPEC, Operation speculatively executed, SVE predicated**

The counter counts each Speculatively executed SIMD data-processing, load, or store operation due to an SVE instruction with a Governing predicate operand that determines the Active elements.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8075, **SVE_PRED_EMPTY_SPEC, Operation speculatively executed, SVE predicated with no active predicates**

The counter counts each Speculatively executed predicated SIMD data-processing, load, or store operation counted by **SVE_PRED_SPEC** where all elements are Inactive.

That is, all elements in the Governing predicate are FALSE.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8076, **SVE_PRED_FULL_SPEC, Operation speculatively executed, SVE predicated with all active predicates**

The counter counts each Speculatively executed predicated SIMD data-processing, load, or store operation counted by **SVE_PRED_SPEC** where all elements are Active.

That is, all elements in the Governing predicate are TRUE.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8077, **SVE_PRED_PARTIAL_SPEC, Operation speculatively executed, SVE predicated with partially active predicates**

The counter counts each Speculatively executed predicated SIMD data-processing, load, or store operation counted by **SVE_PRED_SPEC** that is not counted by either **SVE_PRED_EMPTY_SPEC** or **SVE_PRED_FULL_SPEC**.

That is, the elements in the Governing predicate are neither all TRUE nor all FALSE.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8078, **SVE_UNPRED_SPEC, Operation speculatively executed, SVE unpredicated**

The counter counts each Speculatively executed SIMD data-processing, load, or store operation due to an SVE instruction without a Governing predicate operand.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8079, **SVE_PRED_NOT_FULL_SPEC, SVE predicated operations Speculatively executed with no active or partially active predicates**

The counter counts Speculatively executed SIMD data-processing and load and store operations due to SVE instructions with a Governing predicate in which at least one element is FALSE. This includes both those counted by **SVE_PRED_PARTIAL_SPEC** and those counted by **SVE_PRED_EMPTY_SPEC**.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x807C, **SVE_MOVPRFX_SPEC, Operation speculatively executed, SVE MOVPRFX**

The counter counts each Speculatively executed operation due to any of the following A64 instructions:

- SVE: MOVPRFX.
The instruction is counted whether or not it is fused with the prefixed instruction. Implementation of this optional event requires that FEAT_SVE is implemented.

0x807D, SVE_MOVPRFX_Z_SPEC, Operation speculatively executed, SVE MOVPRFX zeroing predication

The counter counts each Speculatively executed operation counted by SVE_MOVPRFX_SPEC where the operation uses zeroing predication. Implementation of this optional event requires that FEAT_SVE is implemented.

0x807E, SVE_MOVPRFX_M_SPEC, Operation speculatively executed, SVE MOVPRFX merging predication

The counter counts each Speculatively executed operation counted by SVE_MOVPRFX_SPEC where the operation uses merging predication. Implementation of this optional event requires that FEAT_SVE is implemented.

0x807F, SVE_MOVPRFX_U_SPEC, Operation speculatively executed, SVE MOVPRFX unfused

The counter counts each Speculatively executed operation counted by SVE_MOVPRFX_SPEC where the MOVPRFX is not fused with the prefixed instruction. Implementation of this optional event requires that FEAT_SVE is implemented.

0x8080, SVE_LDST_SPEC, Operation speculatively executed, SVE load, store, or prefetch

The counter counts each Speculatively executed load, store, or prefetch operation counted by any of SVE_LD_SPEC, SVE_PRF_SPEC, or SVE_ST_SPEC. Implementation of this optional event requires that FEAT_SVE is implemented.

0x8081, SVE_LD_SPEC, Operation speculatively executed, SVE load

The counter counts each Speculatively executed operation that reads from memory due to an SVE load instruction. Implementation of this optional event requires that FEAT_SVE is implemented.

0x8082, SVE_ST_SPEC, Operation speculatively executed, SVE store

The counter counts each Speculatively executed operation that writes to memory due to an SVE store instruction. Implementation of this optional event requires that FEAT_SVE is implemented.

0x8083, SVE_PRF_SPEC, Operation speculatively executed, SVE prefetch

The counter counts each Speculatively executed prefetch operation due to any of the following A64 instructions:

• SVE: PRFB, PRFD, PRFH, or PRFW.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8084, ASE_SVE_LDST_SPEC, Operation speculatively executed, Advanced SIMD or SVE load or store

The counter counts each Speculatively executed load or store operation counted by any of ASE_SVE_LD_SPEC, ASE_SVE_ST_SPEC, or PRF_SPEC. It is IMPLEMENTATION DEFINED which Advanced SIMD load or store operations are counted in AArch32 state.

0x8085, ASE_SVE_LD_SPEC, Operation speculatively executed, Advanced SIMD or SVE load

The counter counts each Speculatively executed operation that reads from memory due to an A64 SVE or Advanced SIMD load instruction. It is IMPLEMENTATION DEFINED which Advanced SIMD load operations are counted in AArch32 state.

0x8086, ASE_SVE_ST_SPEC, Operation speculatively executed, Advanced SIMD or SVE store

The counter counts each Speculatively executed operation that writes to memory due to an A64 SVE or Advanced SIMD store instruction.
It is IMPLEMENTATION DEFINED which Advanced SIMD store operations are counted in AArch32 state.

0x8087, PRF_SPEC, Operation speculatively executed, Prefetch

The counter counts each Speculatively executed operation that prefetches memory due to any of the following A64 instructions:

• Scalar: PRFM.
• SVE: PRFB, PRFD, PRFH, or PRFW.

It is IMPLEMENTATION DEFINED which prefetch operations are counted in AArch32 state.

0x8088, BASE_L DST_REG_SPEC, Operation speculatively executed, general-purpose register load, store, or prefetch

The counter counts each Speculatively executed general-purpose register load, store, or prefetch operation counted by any of BASE_LD_REG_SPEC, BASE_PRF_SPEC, or BASE_ST_REG_SPEC.

0x8089, BASE_LD_REG_SPEC, Operation speculatively executed, general-purpose register load

The counter counts each Speculatively executed operation that loads a general-purpose register from memory.

It is IMPLEMENTATION DEFINED which load operations are counted in AArch32 state.

0x808A, BASE_ST_REG_SPEC, Operation speculatively executed, general-purpose register store

The counter counts each Speculatively executed operation that stores a general-purpose register to memory.

It is IMPLEMENTATION DEFINED whether the counter counts operations due to DC ZVA instructions.

It is IMPLEMENTATION DEFINED which store operations are counted in AArch32 state.

0x808B, BASE_PRF_SPEC, Operation speculatively executed, general-purpose register prefetch

The counter counts each Speculatively executed prefetch operation due to any of the following A64 instructions:

• Scalar: PRFM.

It is IMPLEMENTATION DEFINED which prefetch operations are counted in AArch32 state.

0x808C, FPASE_L DST_REG_SPEC, Operation speculatively executed, SIMD&FP register load or store

The counter counts each Speculatively executed SIMD&FP register load or store operation counted by either FPASE_LD_REG_SPEC or FPASE_ST_REG_SPEC.

0x808D, FPASE_LD_REG_SPEC, Operation speculatively executed, SIMD&FP register load

The counter counts each Speculatively executed operation that reads from memory due to any of the following A64 instructions:

• Scalar: LDP (SIMD&FP) or LDR (SIMD&FP).
• Advanced SIMD: LD1 or LD1R.

It is IMPLEMENTATION DEFINED which load operations are counted in AArch32 state.

0x808E, FPASE_ST_REG_SPEC, Operation speculatively executed, SIMD&FP register store

The counter counts each Speculatively executed operation that writes to memory due to any of the following A64 instructions:

• Scalar: STP (SIMD&FP) or STR (SIMD&FP).
• Advanced SIMD: ST1.

It is IMPLEMENTATION DEFINED which store operations are counted in AArch32 state.

0x8090, SVE_L DST_REG_SPEC, Operation speculatively executed, SVE unpredicated load or store register

The counter counts each Speculatively executed SVE register load or store operation counted by either SVE_LDR_REG_SPEC or SVE_STR_REG_SPEC.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x8091, **SVE_LDR_REG_SPEC, Operation speculatively executed, SVE unpredicated load register**

The counter counts each speculatively executed SVE operation that reads from memory due to any of the following A64 instructions:

- SVE: LDR.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8092, **SVE_STR_REG_SPEC, Operation speculatively executed, SVE unpredicated store register**

The counter counts each speculatively executed SVE operation that writes to memory due to any of the following A64 instructions:

- SVE: STR.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8094, **SVE_LDR_PREG_SPEC, Operation speculatively executed, SVE load or store predicate register**

The counter counts each speculatively executed SVE predicate register load or store operation counted by either SVE_LDR_PREG_SPEC or SVE_STR_PREG_SPEC.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8095, **SVE_STR_PREG_SPEC, Operation speculatively executed, SVE load predicate register**

The counter counts each speculatively executed SVE load predicate register operation that reads from memory due to any of the following A64 instructions:

- SVE: LDR (predicate).

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8096, **SVE_STR_PREG_SPEC, Operation speculatively executed, SVE store predicate register**

The counter counts each speculatively executed SVE store predicate register operation that writes to memory due to any of the following A64 instructions:

- SVE: STR (predicate).

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8098, **SVE_LDR_ZREG_SPEC, Operation speculatively executed, SVE load or store vector register**

The counter counts each speculatively executed SVE vector register load or store operation counted by either SVE_LDR_ZREG_SPEC or SVE_STR_ZREG_SPEC.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x8099, **SVE_LDR_ZREG_SPEC, Operation speculatively executed, SVE load vector register**

The counter counts each speculatively executed SVE load vector register operation that reads from memory due to any of the following A64 instructions:

- SVE: LDR (vector).

Implementation of this optional event requires that FEAT_SVE is implemented.

0x809A, **SVE_STR_ZREG_SPEC, Operation speculatively executed, SVE store vector register**

The counter counts each speculatively executed SVE store vector register operation that writes to memory due to any of the following A64 instructions:

- SVE: STR (vector).

Implementation of this optional event requires that FEAT_SVE is implemented.

0x809C, **SVE_LDST_CONTIG_SPEC, Operation speculatively executed, SVE contiguous load, store, or prefetch element**

The counter counts each speculatively executed SVE contiguous element load, store, or prefetch operation counted by any of SVE_LD_CONTIG_SPEC, SVE_PRF_CONTIG_SPEC, or SVE_ST_CONTIG_SPEC.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x809D, SVE_LD_CONTIG_SPEC, Operation speculatively executed, SVE contiguous load element
The counter counts each Speculatively executed operation that reads from memory due to either:
  • An SVE predicated single vector contiguous element load instruction.
  • An SVE load and replicate LD1R or LD1RQ instruction.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x809E, SVE_ST_CONTIG_SPEC, Operation speculatively executed, SVE contiguous store element
The counter counts each Speculatively executed operation that writes to memory due to an SVE predicated single vector contiguous element store instruction.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x809F, SVE_PRF_CONTIG_SPEC, Operation speculatively executed, SVE contiguous prefetch element
The counter counts each Speculatively executed operation that prefetches memory due to an SVE predicated single contiguous element prefetch instruction.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80A0, SVE_LDSNT_CONTIG_SPEC, Operation speculatively executed, SVE non-temporal contiguous load or store element
The counter counts each Speculatively executed SVE non-temporal contiguous element load or store operation counted by either SVE_LDNT_CONTIG_SPEC or SVE_STNT_CONTIG_SPEC.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80A1, SVE_LDNT_CONTIG_SPEC, Operation speculatively executed, SVE non-temporal contiguous load element
The counter counts each Speculatively executed operation that reads from memory with a non-temporal hint due to an SVE non-temporal contiguous element load instruction.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80A2, SVE_STNT_CONTIG_SPEC, Operation speculatively executed, SVE non-temporal contiguous store element
The counter counts each Speculatively executed operation that writes to memory with a non-temporal hint due to an SVE non-temporal contiguous element store instruction.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80A4, ASE_SVE_LDSNT_MULTI_SPEC, Operation speculatively executed, Advanced SIMD or SVE contiguous load or store multiple vector
The counter counts each Speculatively executed multiple vector contiguous structure load or store operation counted by either ASE_SVE_LD_MULTI_SPEC or ASE_SVE_ST_MULTI_SPEC.
It is IMPLEMENTATION DEFINED which Advanced SIMD contiguous structure load or store operations are counted in AArch32 state.

0x80A5, ASE_SVE_LD_MULTI_SPEC, Operation speculatively executed, Advanced SIMD or SVE contiguous load multiple vector
The counter counts each Speculatively executed operation that reads from memory due to an A64 SVE or Advanced SIMD multiple vector contiguous structure load instruction.
It is IMPLEMENTATION DEFINED which Advanced SIMD contiguous structure load operations are counted in AArch32 state.

0x80A6, ASE_SVE_ST_MULTI_SPEC, Operation speculatively executed, Advanced SIMD or SVE contiguous store multiple vector
The counter counts each Speculatively executed operation that writes to memory due to an A64 SVE or Advanced SIMD multiple vector contiguous structure store instruction.
It is IMPLEMENTATION DEFINED which Advanced SIMD contiguous structure store operations are counted in AArch32 state.

0x80A8, SVE_LDST_MULTI_SPEC, Operation speculatively executed, SVE contiguous load or store multiple vector

The counter counts each Speculatively executed SVE contiguous load or store multiple vector operation counted by either SVE_LD_MULTI_SPEC or SVE_ST_MULTI_SPEC.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80A9, SVE_LD_MULTI_SPEC, Operation speculatively executed, SVE contiguous load multiple vector

The counter counts each Speculatively executed load or store operation counted by either SVE_LD_MULTI_SPEC or SVE_ST_MULTI_SPEC.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80AA, SVE_ST_MULTI_SPEC, Operation speculatively executed, SVE contiguous store multiple vector

The counter counts each Speculatively executed operation that writes to memory due to an SVE multiple vector contiguous structure store instruction.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80AC, SVE_LDST_NONCONTIG_SPEC, Operation speculatively executed, SVE non-contiguous load, store, or prefetch

The counter counts each Speculatively executed gather-load, scatter-store, or gather-prefetch operation counted by any of SVE_LD_GATHER_SPEC, SVE_PRF_GATHER_SPEC, or SVE_ST_SCATTER_SPEC.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80AD, SVE_LD_GATHER_SPEC, Operation speculatively executed, SVE gather-load

The counter counts each Speculatively executed operation that reads from memory due to an SVE non-contiguous gather-load instruction.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80AE, SVE_ST_SCATTER_SPEC, Operation speculatively executed, SVE scatter-store

The counter counts each Speculatively executed operation that writes to memory due to an SVE non-contiguous scatter-store instruction.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80AF, SVE_PRF_GATHER_SPEC, Operation speculatively executed, SVE gather-prefetch

The counter counts each Speculatively executed operation that prefetches memory due to an SVE non-contiguous gather-prefetch instruction.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80B0, SVE_LDST64_NONCONTIG_SPEC, Operation speculatively executed, SVE 64-bit non-contiguous load, store, or prefetch

The counter counts each Speculatively executed 64-bit gather-load, scatter-store, or gather-prefetch operation counted by any of SVE_LD64_GATHER_SPEC, SVE_PRF64_GATHER_SPEC, or SVE_ST64_SCATTER_SPEC.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80B1, SVE_LD64_GATHER_SPEC, Operation speculatively executed, SVE 64-bit gather-load

The counter counts each Speculatively executed operation that reads from memory due to an SVE non-contiguous gather-load instruction with 64-bit vector elements in the address.
Implementation of this optional event requires that FEAT_SVE is implemented.
0x80B2, **SVE_ST64_SCATTER_SPEC**, Operation speculatively executed, SVE 64-bit scatter-store
The counter counts each Speculatively executed operation that writes to memory due to an SVE non-contiguous scatter-store instruction with 64-bit vector elements in the address.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80B3, **SVE_PRF64_GATHER_SPEC**, Operation speculatively executed, SVE 64-bit gather-prefetch
The counter counts each Speculatively executed operation that prefetches memory due to an SVE non-contiguous gather-prefetch instructions with 64-bit vector elements in the address.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80B4, **ASE_SVE_UNALIGNED_LDST_SPEC**, Advanced SIMD or SVE unaligned accesses
The counter counts each Speculatively executed 64-bit unaligned load or store operation counted by either **ASE_SVE_UNALIGNED_LD_SPEC** or **ASE_SVE_UNALIGNED_ST_SPEC**.
It is IMPLEMENTATION DEFINED which unaligned Advanced SIMD load or store operations are counted in AArch32 state.

0x80B5, **ASE_SVE_UNALIGNED_LD_SPEC**, Advanced SIMD or SVE unaligned read accesses
The counter counts each memory read access due to an A64 SVE or Advanced SIMD load instruction where either:
• A contiguous vector address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.
• A gather, scatter, or single element address is not aligned to the memory element access size, in bytes.
The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.
It is IMPLEMENTATION DEFINED which unaligned Advanced SIMD load operations are counted in AArch32 state.

0x80B6, **ASE_SVE_UNALIGNED_ST_SPEC**, Advanced SIMD or SVE unaligned write accesses
The counter counts each memory write access due to an A64 SVE or Advanced SIMD store instruction where either:
• A contiguous vector address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.
• A gather, scatter, or single element address is not aligned to the memory element access size, in bytes.
The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.
It is IMPLEMENTATION DEFINED which unaligned Advanced SIMD store operations are counted in AArch32 state.

0x80B8, **ASE_SVE_UNALIGNED_CONTIG_LDST_SPEC**, Advanced SIMD or SVE unaligned contiguous accesses
The counter counts each Speculatively executed 64-bit unaligned contiguous load or store operation counted by either **ASE_SVE_UNALIGNED_CONTIG_LD_SPEC** or **ASE_SVE_UNALIGNED_CONTIG_ST_SPEC**.
It is IMPLEMENTATION DEFINED which unaligned contiguous Advanced SIMD load or store operations are counted in AArch32 state.

0x80B9, **ASE_SVE_UNALIGNED_CONTIG_LD_SPEC**, Advanced SIMD or SVE unaligned contiguous read accesses
The counter counts each Memory-read operation due to an A64 SVE or Advanced SIMD contiguous load instruction where the address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.
The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.
It is IMPLEMENTATION DEFINED which unaligned contiguous Advanced SIMD load operations are counted in AArch32 state.

0x80BA, ASE_SVE_UNALIGNED_CONTIG_ST_SPEC, Advanced SIMD or SVE unaligned contiguous write accesses

The counter counts each Memory-write operation due to an A64 SVE or Advanced SIMD contiguous store instruction where the address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

It is IMPLEMENTATION DEFINED which unaligned contiguous Advanced SIMD store operations are counted in AArch32 state.

0x80BC, SVE_LDFF_SPEC, Operation speculatively executed, SVE first-fault load

The counter counts each Speculatively executed Memory-read operation due to an SVE First-fault or Non-fault load instruction.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x80BD, SVE_LDFF_FAULT_SPEC, Operation speculatively executed, SVE first-fault load which set FFR bit to 0b0

The counter counts each Speculatively executed Memory-read operation due to an SVE First-fault or Non-fault load instruction that writes 0b0 to at least one bit in FFR.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x80C0, FP_SCALE_OPS_SPEC, Scalable floating-point element ALU operations Speculatively executed

The counter counts each Speculatively executed floating-point ALU operation counted by SVE_FP_SPEC, except that it is IMPLEMENTATION DEFINED whether operations due to instructions other than those listed in Floating-point arithmetic (SVE) in Arm Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A_ are counted.

See ALU operation counts on page D11-5266 for information on the counter increment for different types of instruction.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x80C1, FP_FIXED_OPS_SPEC, Non-scalable floating-point element ALU operations Speculatively executed

The counter counts each Speculatively executed floating-point ALU operation counted by FP_SPEC but not by SVE_FP_SPEC, and it is IMPLEMENTATION DEFINED whether operations due to instructions other than those listed in Floating-point arithmetic (scalar) in Arm Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A_ and Floating-point arithmetic (Advanced SIMD) in Arm Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for Armv8-A_ are counted.

It does not count operations that are counted by FP_SCALE_OPS_SPEC.

See ALU operation counts on page D11-5266 for information on the counter increment for different types of instruction.

0x80C2, FP_HP_SCALE_OPS_SPEC, Scalable half-precision floating-point element ALU operations Speculatively executed

The counter counts each Speculatively executed scalable floating-point element ALU operation counted by FP_SCALE_OPS_SPEC where the largest type is half precision.

See ALU operation counts on page D11-5266 for information on the counter increment for different types of instruction.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x80C3, FP_HP_FIXED_OPS_SPEC, Non-scalable half-precision floating-point element ALU operations Speculatively executed

The counter counts each Speculatively executed non-scalable floating-point ALU operation counted by FP_FIXED_OPS_SPEC where the largest type is half precision.
See ALU operation counts on page D11-5266 for information on the counter increment for different types of instruction.

0x80C4, FP_SP_SCALE_OPS_SPEC, Scalable single-precision floating-point element ALU operations Speculatively executed
The counter counts each Speculatively executed scalable floating-point element ALU operation counted by FP_SCALE_OPS_SPEC where the largest type is single precision.
See ALU operation counts on page D11-5266 for information on the counter increment for different types of instruction.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80C5, FP_SP_FIXED_OPS_SPEC, Non-scalable single-precision floating-point element ALU operations Speculatively executed
The counter counts each Speculatively executed non-scalable floating-point ALU operation counted by FP_FIXED_OPS_SPEC where the largest type is single precision.
See ALU operation counts on page D11-5266 for information on the counter increment for different types of instruction.

0x80C6, FP_DP_SCALE_OPS_SPEC, Scalable double-precision floating-point element ALU operations Speculatively executed
The counter counts each Speculatively executed scalable floating-point element ALU operation counted by FP_SCALE_OPS_SPEC where the largest type is double precision.
See ALU operation counts on page D11-5266 for information on the counter increment for different types of instruction.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80C7, FP_DP_FIXED_OPS_SPEC, Non-scalable double-precision floating-point element ALU operations Speculatively executed
The counter counts each Speculatively executed non-scalable floating-point ALU operation counted by FP_FIXED_OPS_SPEC where the largest type is double precision.
See ALU operation counts on page D11-5266 for information on the counter increment for different types of instruction.

0x80C8, INT_SCALE_OPS_SPEC, Scalable integer element ALU operations Speculatively executed
The counter counts each integer ALU operation counted by SVE_INT_SPEC.
See ALU operation counts on page D11-5266 for information on the counter increment for different types of instruction.
Implementation of this optional event requires that FEAT_SVE is implemented.

0x80C9, INT_FIXED_OPS_SPEC, Non-scalable integer element ALU operations Speculatively executed
The counter counts each integer ALU operation counted by INT_SPEC that is not counted by SVE_INT_SPEC.
See ALU operation counts on page D11-5266 for information on the counter increment for different types of instruction.

0x80CA, LDST_SCALE_OPS_SPEC, Scalable load or store element Operations speculatively executed
The counter counts each Speculatively executed Memory-read operation or Memory-write operation due to an SVE predicated vector load or store instruction other than a replicating LD1R or LD1RQ instruction.
See ALU operation counts on page D11-5266 for information on the counter increment for different types of instruction.
The counter does not count tag loads or tag stores.
Implementation of this optional event requires that FEAT_SVE is implemented.
0x80CB, **LDST_FIXED_OPS_SPEC**, Non-scalable load or store element Operations speculatively executed

The counter counts each Speculatively executed Memory-read operation or Memory-write operation due to any of:

- Any non-SVE load, store, or atomic operation.
- Any SVE non-vector load or store operation.
- An SVE replicating LD1R or LD1RQ instruction.

See *ALU operation counts* on page D11-5266 for information on the counter increment for different types of instruction.

0x80CC, **LD_SCALE_OPS_SPEC**, Scalable load element Operations speculatively executed

The counter counts each Speculatively executed Memory-read operation due to an SVE predicated vector load instruction other than a replicating LD1R or LD1RQ instruction.

See *ALU operation counts* on page D11-5266 for information on the counter increment for different types of instruction.

The counter does not count tag loads.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x80CD, **LD_FIXED_OPS_SPEC**, Non-scalable load element Operations speculatively executed

The counter counts each Speculatively executed Memory-read operation due to any of:

- Any non-SVE load or atomic operation.
- Any SVE non-vector load operation.
- An SVE replicating LD1R or LD1RQ instruction.

See *ALU operation counts* on page D11-5266 for information on the counter increment for different types of instruction.

0x80CE, **ST_SCALE_OPS_SPEC**, Scalable store element Operations speculatively executed

The counter counts each Speculatively executed Memory-write operation due to an SVE predicated vector store instruction.

See *ALU operation counts* on page D11-5266 for information on the counter increment for different types of instruction.

The counter does not count tag stores.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x80CF, **ST_FIXED_OPS_SPEC**, Non-scalable store element Operations speculatively executed

The counter counts each Memory-write operation due to either:

- Any non-SVE store or atomic operation.
- Any SVE non-vector store operation.

See *ALU operation counts* on page D11-5266 for information on the counter increment for different types of instruction.

0x80DA, **LDST_SCALE_BYTES_SPEC**, Scalable load and store bytes Speculatively executed

The counter counts each byte speculatively read or written due to an SVE vector load or store instruction other than a replicating LD1R or LD1RQ instruction.

For each instruction, the counter is incremented by \((16 ÷ (CSIZE ÷ MSIZE))\), multiplied by the number of transferred vector registers.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x80DB, **LDST_FIXED_BYTES_SPEC**, Non-scalable load and store bytes Speculatively executed

The counter counts each byte speculatively read or written due to any of:

- Any non-SVE load, store, or atomic operation.
- Any SVE non-vector load or store operation.
• An SVE replicating LD1R or LD1RQ instruction.

For each instruction, the counter is incremented by the number of bytes transferred per register multiplied by the number of registers transferred multiplied by the number of transfers made per register. For example, the counter counts bytes as follows:

• Non-SVE load and store of a single register instructions increment the counter by \((\text{MSIZE} \div 8)\).

• Non-SVE load and store of a pair of registers instructions increment the counter by \(2 \times (\text{MSIZE} \div 8)\).

• AArch32 load and store multiple registers instructions increment the counter by the number of registers transferred multiplied by \((\text{MSIZE} \div 8)\).

• Atomic store instructions increment the counter by \((\text{MSIZE} \div 8)\). These are instructions that atomically update a value in memory without returning a value to a register.

• Atomic load, compare and swap of a single register, and swap instructions increment the counter by \(2 \times (\text{MSIZE} \div 8)\). Atomic load instructions are instructions that atomically update a value in memory, returning a value to a register.

• Compare and swap of a pair of registers increment the counter by \(4 \times (\text{MSIZE} \div 8)\).

• SVE and Advanced SIMD LD1R instructions increment the counter by \((\text{MSIZE} \div 8)\).

• SVE LD1RQ instructions increment the counter by \(16\).

• Advanced SIMD LD[1-4] and ST[1-4] instructions increment the counter by the number of registers transferred multiplied by the number of bytes being transferred per register.

• DC ZVA and DC GZVA instructions increment the counter by \(2^{(\text{DCZID.EL0.BS})}\).

0x80DC, LD_SCALE_BYTES_SPEC, Scalable load bytes Speculatively executed
The counter counts each byte speculatively read due to an SVE vector load instruction other than a replicating LD1R or LD1RQ instruction.

For each instruction, the counter is incremented by \((16 \div (\text{CSIZE} \div \text{MSIZE}))\), multiplied by the number of transferred vector registers.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x80DD, LD_FIXED_BYTES_SPEC, Non-scalable load bytes Speculatively executed
The counter counts each byte speculatively read due to any of:

• Any non-SVE load or atomic operation.

• Any SVE non-vector load operation.

• An SVE replicating LD1R or LD1RQ instruction.

For each instruction, the counter is incremented by the number of bytes transferred per register multiplied by the number of registers transferred. That is, the counter is incremented by:

• Half the value that the LDST_FIXED_BYTES_SPEC event counts if the operation is a load atomic, compare and swap, or compare operation.

• The same as for LDST_FIXED_BYTES_SPEC if the operation is any other load operation.

0x80DE, ST_SCALE_BYTES_SPEC, Scalable store bytes Speculatively executed
The counter counts each byte speculatively written due to an SVE vector store instruction.

For each instruction, the counter is incremented by \((16 \div (\text{CSIZE} \div \text{MSIZE}))\), multiplied by the number of transferred vector registers.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x80DF, ST_FIXED_BYTES_SPEC, Non-scalable store bytes Speculatively executed
The counter counts each byte written due to either:

• Any non-SVE store or atomic operation.

• Any SVE non-vector store operation.
For each instruction, the counter is incremented by the number of bytes transferred per register multiplied by the number of registers transferred. That is, the counter is incremented by:

- Half the value that the LDST\_FIXED\_BYTES\_SPEC event counts if the operation is a compare and swap or compare operation.
- The same as for LDST\_FIXED\_BYTES\_SPEC if the operation is any other store operation, including an atomic store operation.

0x80E1, ASE\_INT8\_SPEC, Integer operation speculatively executed, Advanced SIMD 8-bit
The counter counts each operation counted by ASE\_SVE\_INT8\_SPEC that is an Advanced SIMD operation.

0x80E2, SVE\_INT8\_SPEC, Integer operation speculatively executed, SVE 8-bit
The counter counts each operation counted by ASE\_SVE\_INT8\_SPEC that is an SVE operation. Implementation of this optional event requires that FEAT\_SVE is implemented.

0x80E3, ASE\_SVE\_INT8\_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE 8-bit
The counter counts each operation counted by ASE\_SVE\_INT\_SPEC where the largest type is 8-bit integer.

0x80E5, ASE\_SVE\_INT16\_SPEC, Integer operation speculatively executed, Advanced SIMD 16-bit
The counter counts each operation counted by ASE\_SVE\_INT16\_SPEC that is an Advanced SIMD operation.

0x80E6, SVE\_INT16\_SPEC, Integer operation speculatively executed, SVE 16-bit
The counter counts each operation counted by ASE\_SVE\_INT16\_SPEC that is an SVE operation. Implementation of this optional event requires that FEAT\_SVE is implemented.

0x80E7, ASE\_SVE\_INT16\_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE 16-bit
The counter counts each operation counted by ASE\_SVE\_INT\_SPEC where the largest type is 16-bit integer.

0x80E9, ASE\_INT32\_SPEC, Integer operation speculatively executed, Advanced SIMD 32-bit
The counter counts each operation counted by ASE\_SVE\_INT32\_SPEC that is an Advanced SIMD operation.

0x80EA, SVE\_INT32\_SPEC, Integer operation speculatively executed, SVE 32-bit
The counter counts each operation counted by ASE\_SVE\_INT32\_SPEC that is an SVE operation. Implementation of this optional event requires that FEAT\_SVE is implemented.

0x80EB, ASE\_SVE\_INT32\_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE 32-bit
The counter counts each operation counted by ASE\_SVE\_INT\_SPEC where the largest type is 32-bit integer.

0x80ED, ASE\_INT64\_SPEC, Integer operation speculatively executed, Advanced SIMD 64-bit
The counter counts each operation counted by ASE\_SVE\_INT64\_SPEC that is an Advanced SIMD operation.

0x80EE, SVE\_INT64\_SPEC, Integer operation speculatively executed, SVE 64-bit
The counter counts each operation counted by ASE\_SVE\_INT64\_SPEC that is an SVE operation. Implementation of this optional event requires that FEAT\_SVE is implemented.

0x80EF, ASE\_SVE\_INT64\_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE 64-bit
The counter counts each operation counted by ASE\_SVE\_INT\_SPEC where the largest type is 64-bit integer.
The counter counts each dot-product operation counted by ASE_SVE_FP_DOT_SPEC due to any of the following A64 instructions:

- Advanced SIMD: BFDOT.

It is IMPLEMENTATION DEFINED which dot-product operations are counted in AArch32 state.

The counter counts each dot-product operation counted by ASE_SVE_FP_DOT_SPEC due to any of the following instructions:

- SVE: BFDOT.
- SVE2: BFDOT.

Implementation of this optional event requires that FEAT_SVE is implemented.

The counter counts each dot-product operation counted by FP_SPEC due to any of the following A64 instructions:

- Advanced SIMD: BFDOT.
- SVE: BFDOT.
- SVE2: BFDOT.

It is IMPLEMENTATION DEFINED which dot-product operations are counted in AArch32 state.

The counter counts each floating-point matrix multiply operation counted by ASE_SVE_FP_MMLA_SPEC due to any of the following A64 instructions:

- Advanced SIMD: BFMMUL.

It is IMPLEMENTATION DEFINED which floating-point matrix multiply operations are counted in AArch32 state.

The counter counts each floating-point matrix multiply operation counted by ASE_SVE_FP_MMLA_SPEC due to any of the following instructions:

- SVE: BFMMUL or FMUL.
- SVE2: BFMMUL or FMUL.

Implementation of this optional event requires that FEAT_SVE is implemented.

The counter counts each floating-point matrix multiply operation counted by FP_SPEC due to any of the following A64 instructions:

- Advanced SIMD: BFMMUL.
- SVE: BFMMUL or FMUL.
- SVE2: BFMMUL or FMUL.

It is IMPLEMENTATION DEFINED which floating-point matrix multiply operations are counted in AArch32 state.

The counter counts each integer dot product operation counted by ASE_SVE_INT_DOT_SPEC due to any of the following A64 instructions:

- Advanced SIMD: SDOT, SUDDOT, UDOT, or USDOT.
It is IMPLEMENTATION DEFINED which integer dot product operations are counted in AArch32 state.

0x80FA, SVE_INT_DOT_SPEC, Integer operation speculatively executed, SVE dot-product

The counter counts each integer dot product operation counted by ASE_SVE_INT_DOT_SPEC due to any of the following instructions:

- SVE: SDOT, SUDOT, UDOT, or USDOT.
- SVE2: CDOT, SUDOT, or USDOT.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x80FB, ASE_SVE_INT_DOT_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE dot-product

The counter counts each integer dot product operation counted by INT_SPEC due to any of the following A64 instructions:

- Advanced SIMD: SDOT, SUDOT, UDOT, or USDOT.
- SVE: SDOT, SUDOT, UDOT, or USDOT.
- SVE2: CDOT, SUDOT, or USDOT.

It is IMPLEMENTATION DEFINED which integer dot product operations are counted in AArch32 state.

0x80FD, ASE_INT_MMLA_SPEC, Integer operation speculatively executed, Advanced SIMD matrix multiply

The counter counts each integer matrix multiply operation counted by ASE_SVE_INT_MMLA_SPEC due to any of the following A64 instructions:

- Advanced SIMD: SIMLA, UMMLA, or USMMLA.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x80FE, SVE_INT_MMLA_SPEC, Integer operation speculatively executed, SVE matrix multiply

The counter counts each integer matrix multiply operation counted by ASE_SVE_INT_MMLA_SPEC due to any of the following instructions:

- SVE: SIMLA, UMMLA, or USMMLA.
- SVE2: SIMLA, UMMLA, or USMMLA.

Implementation of this optional event requires that FEAT_SVE is implemented.

0x80FF, ASE_SVE_INT_MMLA_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE matrix multiply

The counter counts each integer matrix multiply operation counted by INT_SPEC due to any of the following A64 instructions:

- Advanced SIMD: SIMLA, UMMLA, or USMMLA.
- SVE: SIMLA, UMMLA, or USMMLA.
- SVE2: SIMLA, UMMLA, or USMMLA.

It is IMPLEMENTATION DEFINED which integer matrix multiply operations are counted in AArch32 state.

0x8110, BR_IMMED_PRED_RETIRED, Branch Instruction architecturally executed, predicted immediate

The counter counts each instruction counted by both BR_IMMED_RETIRED and BR_PRED_RETIRED.

These are all direct branch instructions on the architecturally executed path, where the branch is correctly predicted.

0x8111, BR_IMMED_MIS_PRED_RETIRED, Branch Instruction architecturally executed, mispredicted immediate

The counter counts each instruction counted by both BR_IMMED_RETIRED and BR_MIS_PRED_RETIRED.
These are all direct branch instructions on the architecturally executed path, where the branch is mispredicted.

0x8112, BR_IND_PRED_RETIRED, Branch Instruction architecturally executed, predicted indirect

The counter counts each instruction counted by both BR_IND_RETIRED and BR_PRED_RETIRED.

These are all indirect branch instructions, including return instructions, on the architecturally executed path, where the branch is correctly predicted.

0x8113, BR_IND_MIS_PRED_RETIRED, Branch Instruction architecturally executed, mispredicted indirect

The counter counts each instruction counted by both BR_IND_RETIRED and BR_MIS_PRED_RETIRED.

These are all indirect branch instructions, including return instructions, on the architecturally executed path, where the branch is mispredicted.

0x8114, BR_RETURN_PRED_RETIRED, Branch Instruction architecturally executed, predicted procedure return

The counter counts each instruction counted by BR_IND_PRED_RETIRED where if taken, the branch would be counted by BR_PRED_RETIRED.

These are all return instructions on the architecturally executed path, where the branch is correctly predicted.

0x8115, BR_RETURN_MIS_PRED_RETIRED, Branch Instruction architecturally executed, mispredicted procedure return

The counter counts each instruction counted by BR_IND_MIS_PRED_RETIRED where if taken, the branch would also be counted by BR_RETURN_RETIRED.

These are all return instructions on the architecturally executed path, where the branch is mispredicted.

0x8116, BR_INDRNR_PRED_RETIRED, Branch Instruction architecturally executed, predicted indirect excluding procedure return

The counter counts each instruction counted by BR_IND_PRED_RETIRED where if taken, the branch would not be counted by BR_RETURN_RETIRED.

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path, where the branch is correctly predicted.

0x8117, BR_INDRNR_MIS_PRED_RETIRED, Branch Instruction architecturally executed, mispredicted indirect excluding procedure return

The counter counts each instruction counted by BR_IND_MIS_PRED_RETIRED where if taken, the branch would not be counted by BR_RETURN_RETIRED.

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path, where the branch is mispredicted.

0x8118, BR_TAKEN_PRED_RETIRED, Branch Instruction architecturally executed, predicted branch, taken

The counter counts each instruction counted by both BR_PRED_RETIRED and PC_WRITE_RETIRED.

These are all branch instructions on the architecturally executed path, where the branch is correctly predicted and taken.

0x8119, BR_TAKEN_MIS_PRED_RETIRED, Branch Instruction architecturally executed, mispredicted branch, taken

The counter counts each instruction counted by both BR_MIS_PRED_RETIRED and PC_WRITE_RETIRED.
These are all branch instructions on the architecturally executed path, where the branch is mispredicted and taken.

0x811A, **BR_SKIP_PRED_RETIRED**, Branch **Instruction architecturally executed**, predicted branch, not taken

The counter counts each instruction counted by both BR_PRED_RETIRED and BR_SKIP_RETIRED.

These are all branch instructions on the architecturally executed path, where the branch is correctly predicted and not taken.

0x811B, **BR_SKIP_MIS_PRED_RETIRED**, Branch **Instruction architecturally executed**, mispredicted branch, not taken

The counter counts each instruction counted by both BR_MIS_PRED_RETIRED and BR_SKIP_RETIRED.

These are all branch instructions on the architecturally executed path, where the branch is mispredicted and not taken.

0x811C, **BR_PRED_RETIRED**, Branch **Instruction architecturally executed**, predicted branch

The counter counts each instruction counted by BR_RETIRED that is not counted by BR_MIS_PRED_RETIRED.

These are all branch instructions on the architecturally executed path, where the branch is correctly predicted.

0x8120, **INST_FETCH_PERCYC**, Event in progress, **INST_FETCH**

The counter counts by the number of INST_FETCH events in progress on each Processor cycle.

The ratio INST_FETCH_PERCYC ÷ INST_FETCH is the mean duration of INST_FETCH events in Processor cycles.

0x8121, **MEM_ACCESS_RD_PERCYC**, Event in progress, **MEM_ACCESS_RD**

The counter counts by the number of MEM_ACCESS_RD events in progress on each Processor cycle.

The ratio MEM_ACCESS_RD_PERCYC ÷ MEM_ACCESS_RD is the mean duration of MEM_ACCESS_RD events in Processor cycles.

0x8124, **INST_FETCH**, Instruction memory access

The counter counts each Instruction memory access that the PE makes. The counter increments whether the access results in an access to a Level 1 instruction cache, a Level 2 instruction, data or unified cache, or none of these.

The counter does not increment as a result of:

- Data memory accesses.
- Translation table walks.
- Refilling of any cache.
- Accesses that result from cache maintenance instructions.

0x8125, **BUS_REQ_RD_PERCYC**, Bus read transactions in progress

The counter counts by the number of bus read transactions counted by BUS_REQ_RD in progress on each Processor cycle.

The ratio BUS_REQ_RD_PERCYC ÷ BUS_REQ_RD is the mean duration of bus read transactions in Processor cycles.

A bus read transaction is in progress between the transaction starting and it completing. This includes cycles when the transaction is not occupying the bus.

If the bus is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPE<\n>EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the bus is not shared, all events are counted.
0x8126, **BUS_REQ_WR_PERCYC, Bus write transactions in progress**

The counter counts the number of bus write transactions counted by BUS_REQ_WR in progress on each Processor cycle.

The ratio BUS_REQ_WR_PERCYC ÷ BUS_REQ_WR is the mean duration of bus write transactions in Processor cycles.

A bus write transaction is in progress between the transaction starting and it completing. This includes cycles when the transaction is not occupying the bus.

A mean duration value calculated using this event only measures the latency from the perspective of the bus. This is not necessarily the same as the latency for the write to be Complete, as the write might be completed on the bus before reaching its endpoint.

For some bus implementations, writes are posted, meaning that, from the perspective of the bus, the write completes immediately. There is no response when the write reaches its Completion endpoint. This event is not included for such implementations.

If the bus is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the bus is not shared, all events are counted.

0x8128, **DTLB_WALK_PERCYC, Event in progress, DTLB_WALK**

The counter counts the number of DTLB_WALK events in progress on each Processor cycle.

The ratio DTLB_WALK_PERCYC ÷ DTLB_WALK is the mean duration of DTLB_WALK events in Processor cycles.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

0x8129, **ITLB_WALK_PERCYC, Event in progress, ITLB_WALK**

The counter counts the number of ITLB_WALK events in progress on each Processor cycle.

The ratio ITLB_WALK_PERCYC ÷ ITLB_WALK is the mean duration of ITLB_WALK events in Processor cycles.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

0x812A, **SAMPLE_FEED_BR, Statistical Profiling sample taken, branch**

The counter counts each sample counted by SAMPLE_FEED that are branch operations. The values of PMSFCR_EL1.{B,FT} are ignored when generating this event.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

This event must be implemented if FEAT_SPEv1p2 is implemented.

Implementation of this event requires that FEAT_SPE is implemented.

0x812B, **SAMPLE_FEED_LD, Statistical Profiling sample taken, load**

The counter counts each sample counted by SAMPLE_FEED that are load or load atomic operations. The values of PMSFCR_EL1.{LD,FT} are ignored when generating this event.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

This event must be implemented if FEAT_SPEv1p2 is implemented.

Implementation of this event requires that FEAT_SPE is implemented.
The counter counts each sample counted by Sample Feed that are store or atomic operations, including load atomic operations.

The values of PMSFCR_EL1.{ST,FT} are ignored when generating this event.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

This event must be implemented if FEAT_SPEv1p2 is implemented.

Implementation of this event requires that FEAT_SPE is implemented.

The counter counts each sample counted by Sample Feed that meets the operation type filter constraints.

That is, one of the following is true:

- The operation is a branch and PMSFCR_EL1.B is 0b1.
- The operation is a load or load atomic, and PMSFCR_EL1.LD is 0b1.
- The operation is a store or atomic operation, and PMSFCR_EL1.ST is 0b1.

The value of PMSFCR_EL1.FT is ignored when generating this event.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

This event must be implemented if FEAT_SPEv1p2 is implemented.

Implementation of this event requires that FEAT_SPE is implemented.

The counter counts each sample counted by Sample Feed that meets the Events packet filter constraints.

That is, each sampled operation with all the events in the filter sets defined by PMSEVFR_EL1 and, if implemented, PMSNEVFR_EL1 are counted. The values of PMSFCR_EL1.{FnE,FE} are ignored when generating this event.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

This event must be implemented if FEAT_SPEv1p2 is implemented.

Implementation of this event requires that FEAT_SPE is implemented.

The counter counts each sample counted by Sample Feed that meets the operation latency filter constraints.

That is, each sampled operation with a total latency greater than or equal to the minimum latency defined by PMSLATFR_EL1.MINLAT are counted. The value of PMSFCR_EL1.FL is ignored when generating this event.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

This event must be implemented if FEAT_SPEv1p2 is implemented.

Implementation of this event requires that FEAT_SPE is implemented.

The counter counts each access counted by L1D_TLB that is due to a demand Memory-read operation or demand Memory-write operation.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

This event must be implemented if event L1D_TLB_PRFM is implemented.
0x8131, **L1I_TLB_RD, Level 1 instruction TLB demand access**

The counter counts each access counted by L1I_TLB that is due to a demand Instruction memory access.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\alpha>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

This event must be implemented if event L1I_TLB_PRFM is implemented.

0x8132, **L1D_TLB_PRFM, Level 1 data TLB software preload**

The counter counts each access counted by L1D_TLB that is due to a preload or prefetch instruction.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\alpha>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

This event must be implemented if event L1D_TLB_RW is implemented.

0x8133, **L1I_TLB_PRFM, Level 1 instruction TLB software preload**

The counter counts each access counted by L1I_TLB that is due to a preload or prefetch instruction.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\alpha>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

This event must be implemented if event L1I_TLB_RD is implemented.

0x8134, **DTLB_HWUPD, Data TLB hardware update of translation table**

The counter counts each access counted by L1D_TLB that causes a hardware update of a translation table entry.

Each attempted hardware update of a translation table entry is counted once. If the PE requires multiple translation table walk accesses to perform an update, this counts as a single update. If the update fails because it would not be atomic and has to be retried, each retry is counted.

The counter does not count the access if any of the following are true:

- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 0b1.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.
- The access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\alpha>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

Implementation of this optional event requires that FEAT_E0PD is implemented.

0x8135, **ITLB_HWUPD, Instruction TLB hardware update of translation table**

The counter counts each access counted by L1I_TLB that causes a hardware update of a translation table entry.

Each attempted hardware update of a translation table entry is counted once. If the PE requires multiple translation table walk accesses to perform an update, this counts as a single update. If the update fails because it would not be atomic and has to be retried, each retry is counted.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.
- The access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.
It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<el>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

Implementation of this optional event requires that FEAT_E0PD is implemented.

0x8136, DTLB_STEP, Data TLB translation table walk, step

The counter counts each translation table walk access made by a refill of the data or unified TLB. The event is Attributable to the access that missed in the TLB and caused the walk, not to the owner of the translation tables being accessed. For example, this means that if an EL0 access causes a translation table walk consisting of accesses to both stage 1 and stage 2 translation tables, all accesses are counted if event counting is allowed at EL0, regardless of whether event counting is allowed at EL1 and/or EL2.

The counter does not count the event if any of the following are true:
- FEAT_SVE is implemented and the access causing the refill is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 0b1.
- The access causing the refill generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.
- The access causing the refill is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.

It is IMPLEMENTATION DEFINED whether the counter counts the event if the access causing the refill generates a Translation fault for any other reason.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<el>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

Implementation of this optional event requires that FEAT_E0PD is implemented.

0x8137, ITLB_STEP, Instruction TLB translation table walk, step

The counter counts each translation table walk access made by a refill of the instruction TLB. The event is Attributable to the access that missed in the TLB and caused the walk, not to the owner of the translation tables being accessed. For example, this means that if an EL0 access causes a translation table walk consisting of accesses to both stage 1 and stage 2 translation tables, all accesses are counted if event counting is allowed at EL0, regardless of whether event counting is allowed at EL1 and/or EL2.

The counter does not count the event if any of the following are true:
- The access causing the refill generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.
- The access causing the refill is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.

It is IMPLEMENTATION DEFINED whether the counter counts the event if the access causing the refill generates a Translation fault for any other reason.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<el>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

Implementation of this optional event requires that FEAT_E0PD is implemented.

0x8138, DTLB_WALK_LARGE, Data TLB large page translation table walk

The counter counts each translation table walk counted by DTLB_WALK where the result of the walk yields a large page size.
The set of large page sizes is the complement of the set of small page sizes defined by the **DTLB_WALK_SMALL** event. For example, these translations might be cached by dedicated TLB resources. This set is IMPLEMENTATION DEFINED and might differ between instruction and data TLBs.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared, the counter counts only events attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\(\text{E}_n\) EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

Implementation of this optional event requires that FEAT_E0PD is implemented.

**0x8139, ITLB_WALK_LARGE, Instruction TLB large page translation table walk**

The counter counts each translation table walk counted by **ITLB_WALK** where the result of the walk yields a large page size.

The set of large page sizes is the complement of the set of small page sizes defined by the **ITLB_WALK_SMALL** event. For example, these translations might be cached by dedicated TLB resources. This set is IMPLEMENTATION DEFINED and might differ between instruction and data TLBs.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared, the counter counts only events attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\(\text{E}_n\) EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

Implementation of this optional event requires that FEAT_E0PD is implemented.

**0x813A, DTLB_WALK_SMALL, Data TLB small page translation table walk**

The counter counts each translation table walk counted by **DTLB_WALK** where the result of the walk yields a small page size.

The set of small page sizes is the complement of the set of large page sizes defined by the **DTLB_WALK_LARGE** event. For example, these translations might be cached by dedicated TLB resources. This set is IMPLEMENTATION DEFINED and might differ between instruction and data TLBs.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared, the counter counts only events attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\(\text{E}_n\) EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

Implementation of this optional event requires that FEAT_E0PD is implemented.

**0x813B, ITLB_WALK_SMALL, Instruction TLB small page translation table walk**

The counter counts each translation table walk counted by **ITLB_WALK** where the result of the walk yields a small page size.

The set of small page sizes is the complement of the set of large page sizes defined by the **ITLB_WALK_LARGE** event. For example, these translations might be cached by dedicated TLB resources. This set is IMPLEMENTATION DEFINED and might differ between instruction and data TLBs.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared, the counter counts only events attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\(\text{E}_n\) EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

Implementation of this optional event requires that FEAT_E0PD is implemented.

**0x813C, DTLB_WALK_RW, Data TLB demand access with at least one translation table walk**

The counter counts each demand access counted by **L1D_TLB_RW** that causes a refill or update of a data or unified TLB involving at least one translation table walk access.
The counter does not count the demand access if any of the following are true:

- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 0b1.
- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.

It is IMPLEMENTATION DEFINED whether the counter counts the demand access if the access generates a Translation fault for any other reason.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\rangle_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

0x813D, ITLB_WALK_RD, Instruction TLB demand access with at least one translation table walk

The counter counts each demand access counted by L1I_TLB_RD that causes a refill or update of an instruction TLB involving at least one translation table walk access.

The counter does not count the demand access if any of the following are true:

- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.
- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.

It is IMPLEMENTATION DEFINED whether the counter counts the demand access if the access generates a Translation fault for any other reason.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\rangle_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

0x813E, DTLB_WALK_PRFM, Data TLB software preload access with at least one translation table walk

The counter counts each access counted by L1D_TLB_PRFM that causes a refill or update of a data or unified TLB involving at least one translation table walk access.

The counter does not count the access if any of the following are true:

- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.
- FEAT_SVE is implemented and the access is a non-fault access that fails because the applicable TCR_ELx.NFDy bit is 0b1.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<\rangle_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

0x813F, ITLB_WALK_PRFM, Instruction TLB software preload access with at least one translation table walk

The counter counts each access counted by L1I_TLB_PRFM that causes a refill or update of an instruction TLB involving at least one translation table walk access.

The counter does not count the access if any of the following are true:

- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 0b1.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 0b1.
It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPE<\alpha>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.

0x8140, L1D_CACHE_RW, Level 1 data cache demand access
The counter counts each access counted by L1D_CACHE that is due to a demand Memory-read operation or demand Memory-write operation.
This includes accesses made by Speculatively executed instructions.
This event must be implemented if any of the following are true:
• Event L1D_CACHE_PRFM is implemented.
• Event L1D_CACHE_HWPRF is implemented.

0x8141, L1I_CACHE_RD, Level 1 instruction cache demand fetch
The counter counts each access counted by L1I_CACHE that is due to a demand Instruction memory access.
This includes instruction prefetches made by the PE for Speculatively executed instructions.
This event must be implemented if any of the following are true:
• Event L1I_CACHE_PRFM is implemented.
• Event L1I_CACHE_HWPRF is implemented.

0x8142, L1D_CACHE_PRFM, Level 1 data cache software preload
The counter counts each access counted by L1D_CACHE that is due to a preload or prefetch instruction.
This event must be implemented if event L1D_CACHE_RW is implemented.

0x8143, L1I_CACHE_PRFM, Level 1 instruction cache software preload
The counter counts each access counted by L1I_CACHE that is due to a preload or prefetch instruction.
This event must be implemented if event L1I_CACHE_RD is implemented.

0x8144, L1D_CACHE_MISS, Level 1 data cache demand access miss
The counter counts each demand access counted by L1D_CACHE_RW that misses in the Level 1 data or unified cache, causing an access to outside of the Level 1 caches of this PE.

0x8145, L1I_CACHE_HWPRF, Level 1 instruction cache hardware prefetch
The counter counts each access counted by L1I_CACHE that is due to a hardware prefetch.
The hardware prefetch is generated by a hardware prefetcher at the Level 1 instruction or unified cache.
The PE might prefetch instructions as part of instruction pipelining, and might do so for Speculatively executed instruction paths. PE instruction prefetching is not counted as hardware prefetching. Unlike PE instruction prefetching, a hardware prefetch only allocates instructions into the cache.

0x8146, L1D_CACHE_REFILL_PRFM, Level 1 data cache refill, software preload
The counter counts each access counted by L1D_CACHE_PRFM that causes a refill of the Level 1 data or unified cache from outside of the Level 1 data or unified cache.

0x8147, L1I_CACHE_REFILL_PRFM, Level 1 instruction cache refill, software preload
The counter counts each access counted by L1I_CACHE_PRFM that causes a refill of the Level 1 instruction or unified cache from outside of the Level 1 instruction or unified cache.
The counter counts each access counted by L2D_CACHE that is due to a demand Memory-read operation or demand Memory-write operation.

This includes:
- Accesses made by Speculatively executed instructions.
- Accesses to the Level 2 data or unified cache due to a refill of another cache caused by a demand Memory-read operation or demand Memory-write operation.

This event must be implemented if any of the following are true:
- Event L2D_CACHE_PRFM is implemented.
- Event L2D_CACHE_HWPRF is implemented.

The counter counts each access counted by L2I_CACHE that is due to a demand Instruction memory access.

This includes:
- Instruction prefetches made by the PE for Speculatively executed instructions.
- Accesses to the Level 2 instruction or unified cache due to a refill of another cache caused by a demand Instruction memory access.

This event must be implemented if any of the following are true:
- Event L2I_CACHE_PRFM is implemented.
- Event L2I_CACHE_HWPRF is implemented.

The counter counts each access counted by L2D_CACHE that is due to a preload or prefetch instruction.

This includes accesses to the Level 2 data or unified cache due to a refill of another cache caused by a preload or prefetch instruction.

This event must be implemented if event L2D_CACHE_RW is implemented.

The counter counts each access counted by L2I_CACHE that is due to a preload or prefetch instruction.

This includes accesses to the Level 2 instruction or unified cache due to a refill of another cache caused by a preload or prefetch instruction.

This event must be implemented if event L2I_CACHE_RD is implemented.

The counter counts each demand access counted by L2D_CACHE_RW that misses in the Level 1 and Level 2 data or unified caches, causing an access to outside of the Level 1 and Level 2 caches of this PE.

The counter counts each access counted by L2I_CACHE that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Level 2 instruction or unified cache.

The counter counts each access counted by L2D_CACHE_PRFM that causes a refill of the Level 2 data or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.
The counter counts each access counted by `L2I_CACHE_PRFM` that causes a refill of the Level 2 instruction or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

This event must be implemented if any of the following are true:
- Event `L3D_CACHE_PRFM` is implemented.
- Event `L3D_CACHE_HWPRF` is implemented.

The counter counts each access counted by `L3D_CACHE` that is due to a preload or prefetch instruction.

This includes accesses to the Level 3 data or unified cache due to a refill of another cache caused by a preload or prefetch instruction.

This event must be implemented if event `L3D_CACHE_RW` is implemented.

The counter counts each demand access counted by `L3D_CACHE_RW` that misses in the Level 1 to Level 3 data or unified caches, causing an access to outside of the Level 1 to Level 3 caches of this PE.

The counter counts each access counted by `L3D_CACHE_PRFM` that causes a refill of the Level 3 data or unified cache, or any Level 1 or Level 2 data, instruction, or unified cache of this PE, from outside of those caches.

The counter counts each access counted by `L3D_CACHE_HWPRF` that is due to a hardware prefetch. The hardware prefetch is generated by a hardware prefetcher at the Level 3 data or unified cache.

The counter counts each access counted by `L2D_CACHE_HWPRF` that is due to a hardware prefetch. The hardware prefetch is generated by a hardware prefetcher at the Level 2 data or unified cache.

The counter counts each access counted by `L3D_CACHE_HWPRF` that is due to a hardware prefetch. The hardware prefetch is generated by a hardware prefetcher at the Level 3 data or unified cache.

The counter counts each access counted by `LL_CACHE_HWPRF` that is due to a hardware prefetch. The hardware prefetch is generated by a hardware prefetcher at the Last level cache.

The counter counts each cycle counted by `STALL_FRONTEND` when no instructions are delivered from the memory system.
The counter counts stalls that occur when the frontend interface to memory is busy or stalled. This includes the stalls counted by STALL_FRONTEND_L1I, STALL_FRONTEND_L2I, STALL_FRONTEND_MEM, and STALL_FRONTEND_TLB, and any other IMPLEMENTATION DEFINED memory stalls.

It does not include stalls that are counted by STALL_FRONTEND_CPUBOUND. However both events will count the same cycle counted by STALL_FRONTEND if there are both memory and processor-resource stall conditions active.

0x8159, STALL_FRONTEND_L1I, Frontend stall cycles, level 1 instruction cache
The counter counts each cycle counted by STALL_FRONTEND_MEMBOUND when there is a demand instruction miss in the first level of instruction cache.

The counter does not count the cycle if any of the following are true:

- The STALL_FRONTEND_L2I event is implemented and there is a demand instruction miss in the second level of instruction cache, meaning the STALL_FRONTEND_L2I event counts the cycle.
- There is a demand instruction miss in the last level of instruction cache within the PE clock domain, meaning the STALL_FRONTEND_MEM event counts the cycle.

Implementation of this optional event requires that the first level of instruction cache is implemented within the PE clock domain and is not the last level of instruction cache within the PE clock domain.

0x815A, STALL_FRONTEND_L2I, Frontend stall cycles, level 2 instruction cache
The counter counts each cycle counted by STALL_FRONTEND_MEMBOUND when there is a demand instruction miss in the second level of instruction or unified cache.

The counter does not count the cycle if there is a demand instruction miss in the last level of instruction cache within the PE clock domain, meaning the STALL_FRONTEND_MEM event counts the cycle.

Implementation of this optional event requires that the second level of instruction cache is implemented within the PE clock domain and is not the last level of instruction cache within the PE clock domain.

0x815B, STALL_FRONTEND_MEM, Frontend stall cycles, last level PE cache or memory
The counter counts each cycle counted by STALL_FRONTEND_MEMBOUND when there is a demand instruction miss in the last level of instruction or unified cache within the PE clock domain or a non-cacheable instruction fetch in progress.

0x815C, STALL_FRONTEND_TLB, Frontend stall cycles, TLB
The counter counts each cycle counted by STALL_FRONTEND_MEMBOUND when there is a demand instruction miss in the instruction or unified TLB.

0x8160, STALL_FRONTEND_CPUBOUND, Frontend stall cycles, processor bound
The counter counts each cycle counted by STALL_FRONTEND when the frontend is stalled on a frontend processor resource, not including memory.

The counter counts stalls that occur when a frontend processor resource is busy. This includes the stalls counted by STALL_FRONTEND_FLOW, STALL_FRONTEND_FLUSH, and STALL_FRONTEND_RENAME, and any other IMPLEMENTATION DEFINED processor resource stalls.

It does not include stalls that are counted by STALL_FRONTEND_MEMBOUND. However both events will count the same cycle counted by STALL_FRONTEND if there are both memory and processor-resource stall conditions active.

0x8161, STALL_FRONTEND_FLOW, Frontend stall cycles, flow control
The counter counts each cycle counted by STALL_FRONTEND_CPUBOUND when the frontend is stalled on unavailability of prediction flow resources.
The Performance Monitors Extension

D11.11 PMU events and event numbers

--- Note ---

This event is not counting stalls due to mispredictions, but rather stalls when the frontend is unable to make a prediction.

---

0x8162, STALL_FRONTEND_FLUSH, Frontend stall cycles, flush recovery

The counter counts each cycle counted by STALL_FRONTEND_CPUBOUND when the frontend is recovering from a flush or resteer.

The situations where the frontend is flushed are IMPLEMENTATION DEFINED. For example, the frontend might be flushed on a branch misprediction or on a Context synchronization event.

0x8163, STALL_FRONTEND_RENAME, Frontend stall cycles, rename full

The counter counts each cycle counted by STALL_FRONTEND_CPUBOUND when operations are available from the frontend but at least one is not ready to be sent to the backend because no rename register is available.

If this event is implemented and counts such stalls then the STALL_BACKEND_RENAME event counts as zero.

0x8164, STALL_BACKEND_MEMBOUND, Backend stall cycles, memory bound

The counter counts each cycle counted by STALL_BACKEND when the backend is waiting for a memory access to complete.

The counter counts stalls that occur when the backend interface to memory is busy or stalled. This includes the stalls counted by STALL_BACKEND_ATOMIC, STALL_BACKEND_L1D, STALL_BACKEND_L2D, STALL_BACKEND_MEM, STALL_BACKEND_MEMCPYSET, STALL_BACKEND_ST, and STALL_BACKEND_TLB, and any other IMPLEMENTATION DEFINED memory stalls.

It does not include stalls that are counted by STALL_BACKEND_CPUBOUND, although both events might count on the same cycle counted by STALL_BACKEND if there are both memory and processor-resource stall conditions active.

0x8165, STALL_BACKEND_L1D, Backend stall cycles, level 1 data cache

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when there is a demand data miss in the first level of data or unified cache.

The counter does not count the cycle if any of the following are true:

- The STALL_BACKEND_L2D event is implemented and there is a demand data miss in the second level of data or unified cache, meaning the STALL_BACKEND_L2D event counts the cycle.
- There is a demand data miss in the last level of data or unified cache within the PE clock domain, meaning the STALL_BACKEND_MEM event counts the cycle.

Implementation of this optional event requires that the first level of data or unified cache is implemented within the PE clock domain and is not the last level of data or unified cache within the PE clock domain.

0x8166, STALL_BACKEND_L2D, Backend stall cycles, level 2 data cache

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when there is a demand data miss in the second level of data or unified cache.

The counter does not count the cycle if there is a demand data miss in the last level of data or unified cache within the PE clock domain, meaning the STALL_BACKEND_MEM event counts the cycle.

Implementation of this optional event requires that the second level of data or unified cache is implemented within the PE clock domain and is not the last level of data or unified cache within the PE clock domain.

0x8167, STALL_BACKEND_TLB, Backend stall cycles, TLB

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when there is a demand data miss in the data or unified TLB.
0x8168, **STALL_BACKEND_ST**, Backend stall cycles, store
The counter counts each cycle counted by **STALL_BACKEND_MEMBOUND** when the backend is stalled waiting for a store.

0x816A, **STALL_BACKEND_CPUBOUND**, Backend stall cycles, processor bound
The counter counts each cycle counted by **STALL_BACKEND** when the backend is stalled on a processor resource, not including memory.

The counter counts stalls that occur when a backend processor resource is busy. This includes the stalls counted by **STALL_BACKEND_RENAME**, and any other **IMPLEMENTATION DEFINED** processor resource stalls.

It does not include stalls that are counted by **STALL_BACKEND_MEMBOUND**, although both events might count on the same cycle counted by **STALL_BACKEND** if there are both memory and processor-resource stall conditions active.

0x816B, **STALL_BACKEND_BUSY**, Backend stall cycles, backend busy
The counter counts each cycle counted by **STALL_BACKEND** when operations are available from the frontend but the backend is not able to accept an operation because an execution unit is busy.

For example a complex operation unit such as a divider is executing a previous operation and cannot accept a new operation.

0x816C, **STALL_BACKEND_ILOCK**, Backend stall cycles, input dependency
The counter counts each cycle counted by **STALL_BACKEND** when operations are available from the frontend but at least one is not ready to be sent to the backend because of an input dependency.

0x816D, **STALL_BACKEND_RENAME**, Backend stall cycles, rename full
The counter counts each cycle counted by **STALL_BACKEND_CPUBOUND** when operations are available from the frontend but at least one is not ready to be sent to the backend because no rename register is available.

If this event is implemented and counts such stalls then the **STALL_FRONTEND_RENAME** event counts as zero.

0x816E, **STALL_BACKEND_ATOMIC**, Backend stall cycles, atomic operation
The counter counts each cycle counted by **STALL_BACKEND_MEMBOUND** when the backend is processing an Atomic operation.

Implementation of this optional event requires that **FEAT_LSE** is implemented.

0x816F, **STALL_BACKEND_MEMCPYSET**, Backend stall cycles, Memory Copy or Set operation
The counter counts each cycle counted by **STALL_BACKEND_MEMBOUND** when the backend is processing an Memory Copy or Set instruction.

The Memory Copy instructions are **CPY** and **CPYF**. The Memory Set instructions are **SET** and **SETG**.

Implementation of this optional event requires that **FEAT_MOPS** is implemented.

0x8170, **CAS_NEAR_FAIL**, Atomic memory Operation speculatively executed, Compare and Swap fail
The counter counts each Compare and Swap operation counted by **CAS_NEAR_SPEC** that does not update the location accessed.

Implementation of this optional event requires that **FEAT_LSE** is implemented.

0x8171, **CAS_NEAR_PASS**, Atomic memory Operation speculatively executed, Compare and Swap pass
The counter counts each Compare and Swap operation counted by **CAS_NEAR_SPEC** that updates the location accessed.

Implementation of this optional event requires that **FEAT_LSE** is implemented.

0x8172, **CAS_NEAR_SPEC**, Atomic memory Operation speculatively executed, Compare and Swap near
The counter counts each Compare and Swap operation counted by **CAS_SPEC** that executes locally to the PE.
The definition of *locally* is IMPLEMENTATION DEFINED. Operations counted by CAS_NEAR_SPEC also generate one of the CAS_NEAR_PASS or CAS_NEAR_FAIL events.

Implementation of this optional event requires that FEAT_LSE is implemented.

0x8173, **CAS_FAR_SPEC**, Atomic memory Operation speculatively executed, Compare and Swap far

The counter counts each Compare and Swap operation counted by CAS_SPEC that does not execute locally to the PE.

The definition of *locally* is IMPLEMENTATION DEFINED. Operations counted by CAS_FAR_SPEC do not generate CAS_NEAR_PASS or CAS_NEAR_FAIL events.

Implementation of this optional event requires that FEAT_LSE is implemented.

0x8174, **CAS_SPEC**, Atomic memory Operation speculatively executed, Compare and Swap

The counter counts each load atomic operation counted by LSE_LD_SPEC that is a Compare and Swap operation.

Implementation of this optional event requires that FEAT_LSE is implemented.

0x8175, **LSE_LD_SPEC**, Atomic memory Operation speculatively executed, load

The counter counts each operation counted by INST_SPEC that is an Atomic memory operation that returns a value to the PE, including Swap and Compare-and-Swap operations.

Implementation of this optional event requires that FEAT_LSE is implemented.

0x8176, **LSE_ST_SPEC**, Atomic memory Operation speculatively executed, store

The counter counts each operation counted by INST_SPEC that is an Atomic memory operation that does not return a value to the PE.

Implementation of this optional event requires that FEAT_LSE is implemented.

0x8177, **LSE_LDST_SPEC**, Atomic memory Operation speculatively executed, load or store

The counter counts each operation counted by either LSE_LD_SPEC or LSE_ST_SPEC.

Implementation of this optional event requires that FEAT_LSE is implemented.

0x8178, **REMOTE_ACCESS_WR**, Access to another socket in a multi-socket system, write

The counter counts each access counted by REMOTE_ACCESS that is a Memory-write operation.

0x8184, **BR_HINT_COND_PRED_RETIRED**, Branch Instruction architecturally executed, predicted hinted conditional

The counter counts each branch counted by both BR_HINT_COND_RETIRED and BR_PRED_RETIRED.

These are all hinted conditional branch instructions on the architecturally executed path, where the branch is correctly predicted.

Implementation of this optional event requires that FEAT_HBC is implemented.

0x8185, **BR_HINT_COND_MIS_PRED_RETIRED**, Branch Instruction architecturally executed, mispredicted hinted conditional

The counter counts each branch counted by both BR_HINT_COND_RETIRED and BR_MIS_PRED_RETIRED.

These are all hinted conditional branch instructions on the architecturally executed path, where the branch is mispredicted.

Implementation of this optional event requires that FEAT_HBC is implemented.

0x8186, **UOP_RETIRED**, Micro-operation architecturally executed

The counter counts each micro-operation that would be executed in a Simple sequential execution of the program.
Unlike \texttt{OP\_RETIRED}, this event is not linked to the definition of \texttt{OP\_SPEC}, meaning it counts micro-operations that are created from other operations after those operations are counted by \texttt{OP\_SPEC}.

\texttt{0x8188, DTLB\_WALK\_BLOCK, Data TLB block translation table walk}
The counter counts each translation table walk counted by \texttt{DTLB\_WALK} where the result of the walk yields a Block.
That is, the translation table walk ends at a Block descriptor, at other than the last level of translation table.
If two stages of translation are used and Stage 1 and Stage 2 yield different granule sizes, the counter counts the translation table walk only if both are a Block.
The counter does not count the walk if the access generates a Translation fault.
If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER\langle n\rangle\_EL0.MT for the counter is $0b1$, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.
Implementation of this optional event requires that FEAT\_E0PD is implemented.

\texttt{0x8189, ITLB\_WALK\_BLOCK, Instruction TLB block translation table walk}
The counter counts each translation table walk counted by \texttt{ITLB\_WALK} where the result of the walk yields a Block.
That is, the translation table walk ends at a Page descriptor, at other than the last level of translation table.
If two stages of translation are used and Stage 1 and Stage 2 yield different granule sizes, the counter counts the translation table walk only if both are a Block.
The counter does not count the walk if the access generates a Translation fault.
If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER\langle n\rangle\_EL0.MT for the counter is $0b1$, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.
Implementation of this optional event requires that FEAT\_E0PD is implemented.

\texttt{0x818A, DTLB\_WALK\_PAGE, Data TLB page translation table walk}
The counter counts each translation table walk counted by \texttt{DTLB\_WALK} where the result of the walk yields a Page.
That is, the translation table walk ends at a Block descriptor, at the last level of translation table.
If two stages of translation are used and Stage 1 and Stage 2 yield different granule sizes, the counter counts the translation table walk only if either is a Page.
The counter does not count the walk if the access generates a Translation fault.
If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER\langle n\rangle\_EL0.MT for the counter is $0b1$, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.
Implementation of this optional event requires that FEAT\_E0PD is implemented.

\texttt{0x818B, ITLB\_WALK\_PAGE, Instruction TLB page translation table walk}
The counter counts each translation table walk counted by \texttt{ITLB\_WALK} where the result of the walk yields a Page.
That is, the translation table walk ends at a Page descriptor, at the last level of translation table.
If two stages of translation are used and Stage 1 and Stage 2 yield different granule sizes, the counter counts the translation table walk only if either is a Page.
The counter does not count the walk if the access generates a Translation fault.
If the TLB is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER\langle n\rangle\_EL0.MT for the counter is $0b1$, other PEs in the multithreaded implementation. If the TLB is not shared, all events are counted.
Implementation of this optional event requires that FEAT_E0PD is implemented.

0x818D, **BUS_REQ_RD, Bus request, read**

The counter counts each transaction counted by BUS_REQ that is a Memory-read operation.

0x818E, **BUS_REQ_WR, Bus request, write**

The counter counts each transaction counted by BUS_REQ that is a Memory-write operation.

0x818F, **BUS_REQ, Bus request**

The counter counts each request generated by a Memory-read operation or Memory-write operation that accesses outside of the boundary of the PE and its closely-coupled caches. Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED. Where an implementation has multiple buses at this boundary, this event counts the sum of requests across all buses.

A bus request is the start of a bus transaction. The exact nature of a bus transaction is IMPLEMENTATION DEFINED, but for the purposes of event monitoring consists of a single access comprising one or more cycles, or beats, when the transaction occupies the bus. The BUS_REQ event therefore counts the number of transactions. The BUS_ACCESS event counts the occupancy of the transaction on the bus.

Bus transactions include refills of and write-backs from data, instruction, and unified caches. Whether bus transactions include operations that use the bus but do not explicitly transfer data is IMPLEMENTATION DEFINED. This must be the same for the BUS_REQ and BUS_ACCESS events.

An Unattributable bus transaction occurs when a requestor outside the PE makes a request that results in a bus transaction, for example, a coherency request.

The maximum increment in any given cycle is IMPLEMENTATION DEFINED.

If the bus is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the bus is not shared, all events are counted.

0x8190, **ISNP_HIT_RD, Snoop hit, demand instruction fetch**

The counter counts each snoop generated in response to a demand Instruction memory access that hits in a cache outside of the cache hierarchy of this PE.

0x8191, **ISNP_HIT_NEAR_RD, Snoop hit in near cache, demand instruction fetch**

The counter counts each snoop generated in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

0x8192, **ISNP_HIT_FAR_RD, Snoop hit in far cache, demand instruction fetch**

The counter counts each snoop generated in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache outside the local PE cluster on the same device.

0x8193, **ISNP_HIT_REMOTE_RD, Snoop hit in remote cache, demand instruction fetch**

The counter counts each snoop generated in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache on a remote device.

0x8194, **DSNP_HIT_RD, Snoop hit, demand data read**

The counter counts each snoop generated in response to a demand Memory-read operation counted by DSNP_HIT_RW that hits in a cache outside of the cache hierarchy of this PE.

0x8195, **DSNP_HIT_NEAR_RD, Snoop hit in near cache, demand data read**

The counter counts each snoop generated in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.
The Performance Monitors Extension

D11.11 PMU events and event numbers

ID081822

Non-Confidential

0x8196, DSNP_HIT_FAR_RD, Snoop hit in far cache, demand data read
The counter counts each snoop generated in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache outside the local PE cluster on the same device.

0x8197, DSNP_HIT_REMOTE_RD, Snoop hit in remote cache, demand data read
The counter counts each snoop generated in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache on a remote device.

0x8198, DSNP_HIT_WR, Snoop hit, demand data write
The counter counts each snoop generated in response to a demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache outside of the cache hierarchy of this PE.

0x8199, DSNP_HIT_NEAR_WR, Snoop hit in near cache, demand data write
The counter counts each snoop generated in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

0x819A, DSNP_HIT_FAR_WR, Snoop hit in far cache, demand data write
The counter counts each snoop generated in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache outside the local PE cluster on the same device.

0x819B, DSNP_HIT_REMOTE_WR, Snoop hit in remote cache, demand data write
The counter counts each snoop generated in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache on a remote device.

0x819C, DSNP_HIT_RW, Snoop hit, demand data access
The counter counts each snoop generated in response to a demand Memory-read operation or demand Memory-write operation that hits in a cache outside of the cache hierarchy of this PE.

0x819D, DSNP_HIT_NEAR_RW, Snoop hit in near cache, demand data access
The counter counts each snoop generated in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

0x819E, DSNP_HIT_FAR_RW, Snoop hit in far cache, demand data access
The counter counts each snoop generated in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache outside the local PE cluster on the same device.

0x819F, DSNP_HIT_REMOTE_RW, Snoop hit in remote cache, demand data access
The counter counts each snoop generated in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache on a remote device.

0x81A0, DSNP_HIT_PRFM, Snoop hit, software data preload
The counter counts each snoop generated in response to a data preload or prefetch instruction that hits in a cache outside of the cache hierarchy of this PE.

0x81A1, DSNP_HIT_NEAR_PRFM, Snoop hit in near cache, software data preload
The counter counts each snoop generated in response to a data preload or prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

0x81A2, DSNP_HIT_FAR_PRFM, Snoop hit in far cache, software data preload
The counter counts each snoop generated in response to a data preload or prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache outside the local PE cluster on the same device.
0x81A3, DSNP_HIT_REMOTE_PRFM, Snoop hit in remote cache, software data preload
The counter counts each snoop generated in response to a data preload or prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache on a remote device.

0x81A4, DSNP_HIT_HWPRF, Snoop hit, hardware data prefetch
The counter counts each snoop generated by a data hardware prefetch that hits in a cache outside of the cache hierarchy of this PE.

0x81A5, DSNP_HIT_NEAR_HWPRF, Snoop hit in near cache, hardware data prefetch
The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

0x81A6, DSNP_HIT_FAR_HWPRF, Snoop hit in far cache, hardware data prefetch
The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache outside the local PE cluster on the same device.

0x81A7, DSNP_HIT_REMOTE_HWPRF, Snoop hit in remote cache, hardware data prefetch
The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache on a remote device.

0x81A8, ISNP_HIT_PRFM, Snoop hit, software instruction preload
The counter counts each snoop generated in response to an instruction preload or prefetch instruction that hits in a cache outside of the cache hierarchy of this PE.

0x81A9, ISNP_HIT_NEAR_PRFM, Snoop hit in near cache, software instruction preload
The counter counts each snoop generated in response to an instruction preload or prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

0x81AA, ISNP_HIT_FAR_PRFM, Snoop hit in far cache, software instruction preload
The counter counts each snoop generated in response to an instruction preload or prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache outside the local PE cluster on the same device.

0x81AB, ISNP_HIT_REMOTE_PRFM, Snoop hit in remote cache, software instruction preload
The counter counts each snoop generated in response to an instruction preload or prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache on a remote device.

0x81AC, ISNP_HIT_HWPRF, Snoop hit, hardware instruction prefetch
The counter counts each snoop generated by an instruction hardware prefetch that hits in a cache outside of the cache hierarchy of this PE.

0x81AD, ISNP_HIT_NEAR_HWPRF, Snoop hit in near cache, hardware instruction prefetch
The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

0x81AE, ISNP_HIT_FAR_HWPRF, Snoop hit in far cache, hardware instruction prefetch
The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache outside the local PE cluster on the same device.

0x81AF, ISNP_HIT_REMOTE_HWPRF, Snoop hit in remote cache, hardware instruction prefetch
The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache on a remote device.

0x81B0, ISNP_HIT, Snoop hit, instruction
The counter counts each snoop that hits in a cache outside of the cache hierarchy of this PE.
0x81B1, ISNP_HIT_NEAR, Snoop hit in near cache, instruction
The counter counts each snoop counted by ISNP_HIT that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

0x81B2, ISNP_HIT_FAR, Snoop hit in far cache, instruction
The counter counts each snoop counted by ISNP_HIT that hits in a cache outside the local PE cluster on the same device.

0x81B3, ISNP_HIT_REMOTE, Snoop hit in remote cache, instruction
The counter counts each snoop counted by ISNP_HIT that hits in a cache on a remote device.

0x81B4, DSNP_HIT, Snoop hit, data
The counter counts each snoop that hits in a cache outside of the cache hierarchy of this PE.

0x81B5, DSNP_HIT_NEAR, Snoop hit in near cache, data
The counter counts each snoop counted by DSNP_HIT that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

0x81B6, DSNP_HIT_FAR, Snoop hit in far cache, data
The counter counts each snoop counted by DSNP_HIT that hits in a cache outside the local PE cluster on the same device.

0x81B7, DSNP_HIT_REMOTE, Snoop hit in remote cache, data
The counter counts each snoop counted by DSNP_HIT that hits in a cache on a remote device.

0x81B8, L1I_CACHE_REFILL_HWPRF, Level 1 instruction cache refill, hardware prefetch
The counter counts each hardware prefetch counted by L1I_CACHE_HWPRF that causes a refill of the Level 1 instruction or unified cache from outside of the Level 1 instruction or unified cache.

0x81B9, L2I_CACHE_REFILL_HWPRF, Level 2 instruction cache refill, hardware prefetch
The counter counts each hardware prefetch counted by L2I_CACHE_HWPRF that causes a refill of the Level 2 instruction or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

0x81BC, L1D_CACHE_REFILL_HWPRF, Level 1 data cache refill, hardware prefetch
The counter counts each hardware prefetch counted by L1D_CACHE_HWPRF that causes a refill of the Level 1 data or unified cache from outside of the Level 1 data or unified cache.

0x81BD, L2D_CACHE_REFILL_HWPRF, Level 2 data cache refill, hardware prefetch
The counter counts each hardware prefetch counted by L2D_CACHE_HWPRF that causes a refill of the Level 2 data or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

0x81BE, L3D_CACHE_REFILL_HWPRF, Level 3 data cache refill, hardware prefetch
The counter counts each hardware prefetch counted by L3D_CACHE_HWPRF that causes a refill of the Level 3 data or unified cache, or any Level 1 or Level 2 data, instruction, or unified cache of this PE, from outside of those caches.

0x81BF, LL_CACHE_REFILL_HWPRF, Last level cache refill, hardware prefetch
The counter counts each hardware prefetch counted by LL_CACHE_HWPRF that causes a refill of the Last level cache, or any other data, instruction, or unified cache of this PE, from outside of those caches.

0x81C0, L1I_CACHE_HIT_RD, Level 1 instruction cache demand fetch hit
The counter counts each demand fetch counted by L1I_CACHE_RD that hits in the Level 1 instruction or unified cache.
0x81C1, L2I_CACHE_HIT_RD, Level 2 instruction cache demand fetch hit
   The counter counts each demand fetch counted by L2I_CACHE_RD that hits in the Level 2
   instruction or unified cache.

0x81C4, L1D_CACHE_HIT_RD, Level 1 data cache demand hit, read
   The counter counts each demand read counted by L1D_CACHE_RD that hits in the Level 1 data or
   unified cache.

0x81C5, L2D_CACHE_HIT_RD, Level 2 data cache demand hit, read
   The counter counts each demand read counted by L2D_CACHE_RD that hits in the Level 2 data or
   unified cache.

0x81C6, L3D_CACHE_HIT_RD, Level 3 data cache demand hit, read
   The counter counts each demand read counted by L3D_CACHE_RD that hits in the Level 3 data or
   unified cache.

0x81C7, LL_CACHE_HIT_RD, Last level cache demand hit, read
   The counter counts each demand read counted by LL_CACHE_RD that hits in the Last level cache.

0x81C8, L1D_CACHE_HIT_WR, Level 1 data cache demand access hit, write
   The counter counts each demand write counted by L1D_CACHE_WR that hits in the Level 1 data
   or unified cache.

0x81C9, L2D_CACHE_HIT_WR, Level 2 data cache demand access hit, write
   The counter counts each demand write counted by L2D_CACHE_WR that hits in the Level 2 data
   or unified cache.

0x81CA, L3D_CACHE_HIT_WR, Level 3 data cache demand access hit, write
   The counter counts each demand write counted by L3D_CACHE_WR that hits in the Level 3 data
   or unified cache.

0x81CB, LL_CACHE_HIT_WR, Last level cache demand access hit, write
   The counter counts each demand write counted by LL_CACHE_WR that hits in the Last level
   cache.

0x81CC, L1D_CACHE_HIT_RW, Level 1 data cache demand access hit
   The counter counts each demand access counted by L1D_CACHE_RW that hits in the Level 1 data
   or unified cache.

0x81CD, L2D_CACHE_HIT_RW, Level 2 data cache demand access hit
   The counter counts each demand access counted by L2D_CACHE_RW that hits in the Level 2 data
   or unified cache.

0x81CE, L3D_CACHE_HIT_RW, Level 3 data cache demand access hit
   The counter counts each demand access counted by L3D_CACHE_RW that hits in the Level 3 data
   or unified cache.

0x81CF, LL_CACHE_HIT_RW, Last level cache demand access hit
   The counter counts each demand access counted by LL_CACHE_RW that hits in the Last level
   cache.

0x81D0, L1I_CACHE_HIT_RD_FPRFM, Level 1 instruction cache demand fetch first hit, fetched by
   software preload
   The counter counts each demand fetch first hit counted by L1I_CACHE_HIT_RD where the cache
   line was fetched in response to a preload or prefetch instruction.
   That is, the L1I_CACHE_REFILL_PRFM event was generated when the cache line was fetched
   into the cache.
Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81D1, L2I_CACHE_HIT_RD_FPRFM, Level 2 instruction cache demand fetch first hit, fetched by software preload
The counter counts each demand fetch first hit counted by L2I_CACHE_HIT_RD where the cache line was fetched in response to a preload or prefetch instruction.
That is, the L2I_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.
Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81D4, L1D_CACHE_HIT_RD_FPRFM, Level 1 data cache demand first hit, read, fetched by software preload
The counter counts each first hit counted by L1D_CACHE_HIT_RW_FPRFM that is a demand Memory-read operation.

0x81D5, L2D_CACHE_HIT_RD_FPRFM, Level 2 data cache demand first hit, read, fetched by software preload
The counter counts each first hit counted by L2D_CACHE_HIT_RW_FPRFM that is a demand Memory-read operation.

0x81D6, L3D_CACHE_HIT_RD_FPRFM, Level 3 data cache demand first hit, read, fetched by software preload
The counter counts each first hit counted by L3D_CACHE_HIT_RW_FPRFM that is a demand Memory-read operation.

0x81D7, LL_CACHE_HIT_RD_FPRFM, Last level cache demand first hit, read, fetched by software preload
The counter counts each first hit counted by LL_CACHE_HIT_RW_FPRFM that is a demand Memory-read operation.

0x81D8, L1D_CACHE_HIT_WR_FPRFM, Level 1 data cache demand access first hit, write, fetched by software preload
The counter counts each first hit counted by L1D_CACHE_HIT_RW_FPRFM that is a demand Memory-write operation.

0x81D9, L2D_CACHE_HIT_WR_FPRFM, Level 2 data cache demand access first hit, write, fetched by software preload
The counter counts each first hit counted by L2D_CACHE_HIT_RW_FPRFM that is a demand Memory-write operation.

0x81DA, L3D_CACHE_HIT_WR_FPRFM, Level 3 data cache demand access first hit, write, fetched by software preload
The counter counts each first hit counted by L3D_CACHE_HIT_RW_FPRFM that is a demand Memory-write operation.

0x81DB, LL_CACHE_HIT_WR_FPRFM, Last level cache demand access first hit, write, fetched by software preload
The counter counts each first hit counted by LL_CACHE_HIT_RW_FPRFM that is a demand Memory-write operation.

0x81DC, L1D_CACHE_HIT_RW_FPRFM, Level 1 data cache demand access first hit, fetched by software preload
The counter counts each demand access first hit counted by L1D_CACHE_HIT_RW where the cache line was fetched in response to a preload or prefetch instruction.
That is, the L1D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.
Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81DD, **L2D_CACHE_HIT_RW_FPRFM**, Level 2 data cache demand access first hit, fetched by software preload

The counter counts each demand access first hit counted by L2D_CACHE_HIT_RW where the cache line was fetched in response to a preload or prefetch instruction.

That is, the L2D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81DE, **L3D_CACHE_HIT_RW_FPRFM**, Level 3 data cache demand access first hit, fetched by software preload

The counter counts each demand access first hit counted by L3D_CACHE_HIT_RW where the cache line was fetched in response to a preload or prefetch instruction.

That is, the L3D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81DF, **LL_CACHE_HIT_RW_FPRFM**, Last level cache demand access first hit, fetched by software preload

The counter counts each demand access first hit counted by LL_CACHE_HIT_RW where the cache line was fetched in response to a preload or prefetch instruction.

That is, the LL_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81E0, **L1I_CACHE_HIT_RD_FHWPRF**, Level 1 instruction cache demand fetch first hit, fetched by hardware prefetcher

The counter counts each demand fetch first hit counted by L1I_CACHE_HIT_RD where the cache line was fetched by a hardware prefetcher.

That is, the L1I_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81E1, **L2I_CACHE_HIT_RD_FHWPRF**, Level 2 instruction cache demand fetch first hit, fetched by hardware prefetcher

The counter counts each demand fetch first hit counted by L2I_CACHE_HIT_RD where the cache line was fetched by a hardware prefetcher.

That is, the L2I_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81E4, **L1D_CACHE_HIT_RW_FHWPRF**, Level 1 data cache demand first hit, read, fetched by hardware prefetcher

The counter counts each first hit counted by L1D_CACHE_HIT_RW_FHWPRF that is a demand Memory-read operation.
0x81E5, **L2D_CACHE_HIT_RD_FHWPRF**, Level 2 data cache demand first hit, read, fetched by hardware prefetcher

The counter counts each first hit counted by **L2D_CACHE_HIT_RW_FHWPRF** that is a demand Memory-read operation.

0x81E6, **L3D_CACHE_HIT_RD_FHWPRF**, Level 3 data cache demand first hit, read, fetched by hardware prefetcher

The counter counts each first hit counted by **L3D_CACHE_HIT_RW_FHWPRF** that is a demand Memory-read operation.

0x81E7, **LL_CACHE_HIT_RD_FHWPRF**, Last level cache demand first hit, read, fetched by hardware prefetcher

The counter counts each first hit counted by **LL_CACHE_HIT_RW_FHWPRF** that is a demand Memory-read operation.

0x81E8, **L1D_CACHE_HIT_WR_FHWPRF**, Level 1 data cache demand access first hit, write, fetched by hardware prefetcher

The counter counts each first hit counted by **L1D_CACHE_HIT_RW_FHWPRF** that is a demand Memory-write operation.

0x81E9, **L2D_CACHE_HIT_WR_FHWPRF**, Level 2 data cache demand access first hit, write, fetched by hardware prefetcher

The counter counts each first hit counted by **L2D_CACHE_HIT_RW_FHWPRF** that is a demand Memory-write operation.

0x81EA, **L3D_CACHE_HIT_WR_FHWPRF**, Level 3 data cache demand access first hit, write, fetched by hardware prefetcher

The counter counts each first hit counted by **L3D_CACHE_HIT_RW_FHWPRF** that is a demand Memory-write operation.

0x81EB, **LL_CACHE_HIT_WR_FHWPRF**, Last level cache demand access first hit, write, fetched by hardware prefetcher

The counter counts each first hit counted by **LL_CACHE_HIT_RW_FHWPRF** that is a demand Memory-write operation.

0x81EC, **L1D_CACHE_HIT_RW_FHWPRF**, Level 1 data cache demand access first hit, fetched by hardware prefetcher

The counter counts each demand access first hit counted by **L1D_CACHE_HIT_RW** where the cache line was fetched by a hardware prefetcher.

That is, the **L1D_CACHE_REFILL_HWPRF** event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81ED, **L2D_CACHE_HIT_RW_FHWPRF**, Level 2 data cache demand access first hit, fetched by hardware prefetcher

The counter counts each demand access first hit counted by **L2D_CACHE_HIT_RW** where the cache line was fetched by a hardware prefetcher.

That is, the **L2D_CACHE_REFILL_HWPRF** event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81EE, **L3D_CACHE_HIT_RW_FHWPRF**, Level 3 data cache demand access first hit, fetched by hardware prefetcher

The counter counts each demand access first hit counted by **L3D_CACHE_HIT_RW** where the cache line was fetched by a hardware prefetcher.
That is, the L3D_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81EF, LL_CACHE_HIT_RW_FHWPRF, Last level cache demand access first hit, fetched by hardware prefetcher

The counter counts each demand access first hit counted by LL_CACHE_HIT_RW where the cache line was fetched by a hardware prefetcher.

That is, the LL_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x81F0, L1I_CACHE_HIT_RD_FPRF, Level 1 instruction cache demand access first hit, fetched by preload or prefetch

The counter counts each first hit counted by either L1I_CACHE_HIT_RD_FHWPRF or L1I_CACHE_HIT_RD_FPRFM.

0x81F1, L2I_CACHE_HIT_RD_FPRF, Level 2 instruction cache demand access first hit, fetched by preload or prefetch

The counter counts each first hit counted by either L2I_CACHE_HIT_RD_FHWPRF or L2I_CACHE_HIT_RD_FPRFM.

0x81F4, L1D_CACHE_HIT_RD_FPRF, Level 1 data cache demand access first hit, read, fetched by preload or prefetch

The counter counts each first hit counted by either L1D_CACHE_HIT_RD_FHWPRF or L1D_CACHE_HIT_RD_FPRFM.

0x81F5, L2D_CACHE_HIT_RD_FPRF, Level 2 data cache demand access first hit, read, fetched by preload or prefetch

The counter counts each first hit counted by either L2D_CACHE_HIT_RD_FHWPRF or L2D_CACHE_HIT_RD_FPRFM.

0x81F6, L3D_CACHE_HIT_RD_FPRF, Level 3 data cache demand access first hit, read, fetched by preload or prefetch

The counter counts each first hit counted by either L3D_CACHE_HIT_RD_FHWPRF or L3D_CACHE_HIT_RD_FPRFM.

0x81F7, LL_CACHE_HIT_RD_FPRF, Last level cache demand access first hit, read, fetched by preload or prefetch

The counter counts each first hit counted by either LL_CACHE_HIT_RD_FHWPRF or LL_CACHE_HIT_RD_FPRFM.

0x81F8, L1D_CACHE_HIT_WR_FPRF, Level 1 data cache demand access first hit, write, fetched by preload or prefetch

The counter counts each first hit counted by either L1D_CACHE_HIT_WR_FHWPRF or L1D_CACHE_HIT_WR_FPRFM.

0x81F9, L2D_CACHE_HIT_WR_FPRF, Level 2 data cache demand access first hit, write, fetched by preload or prefetch

The counter counts each first hit counted by either L2D_CACHE_HIT_WR_FHWPRF or L2D_CACHE_HIT_WR_FPRFM.

0x81FA, L3D_CACHE_HIT_WR_FPRF, Level 3 data cache demand access first hit, write, fetched by preload or prefetch

The counter counts each first hit counted by either L3D_CACHE_HIT_WR_FHWPRF or L3D_CACHE_HIT_WR_FPRFM.
0x81FB, LL_CACHE_HIT_WR_FPRF, Last level cache demand access first hit, write, fetched by preload or prefetch

The counter counts each first hit counted by either LL_CACHE_HIT_WR_FHWPRF or LL_CACHE_HIT_WR_FPRFM.

0x81FC, L1D_CACHE_HIT_RW_FPRF, Level 1 data cache demand access first hit, fetched by preload or prefetch

The counter counts each first hit counted by either L1D_CACHE_HIT_RW_FHWPRF or L1D_CACHE_HIT_RW_FPRFM.

0x81FD, L2D_CACHE_HIT_RW_FPRF, Level 2 data cache demand access first hit, fetched by preload or prefetch

The counter counts each first hit counted by either L2D_CACHE_HIT_RW_FHWPRF or L2D_CACHE_HIT_RW_FPRFM.

0x81FE, L3D_CACHE_HIT_RW_FPRF, Level 3 data cache demand access first hit, fetched by preload or prefetch

The counter counts each first hit counted by either L3D_CACHE_HIT_RW_FHWPRF or L3D_CACHE_HIT_RW_FPRFM.

0x81FF, LL_CACHE_HIT_RW_FPRF, Last level cache demand access first hit, fetched by preload or prefetch

The counter counts each first hit counted by either LL_CACHE_HIT_RW_FHWPRF or LL_CACHE_HIT_RW_FPRFM.

0x8200, L1I_CACHE_HIT, Level 1 instruction cache hit

The counter counts each access counted by L1I_CACHE that hits in the Level 1 instruction or unified cache.

0x8201, L2I_CACHE_HIT, Level 2 instruction cache hit

The counter counts each access counted by L2I_CACHE that hits in the Level 2 instruction or unified cache.

0x8204, L1D_CACHE_HIT, Level 1 data cache hit

The counter counts each access counted by L1D_CACHE that hits in the Level 1 data or unified cache.

0x8205, L2D_CACHE_HIT, Level 2 data cache hit

The counter counts each access counted by L2D_CACHE that hits in the Level 2 data or unified cache.

0x8206, L3D_CACHE_HIT, Level 3 data cache hit

The counter counts each access counted by L3D_CACHE that hits in the Level 3 data or unified cache.

0x8207, LL_CACHE_HIT, Last level cache hit

The counter counts each access counted by LL_CACHE that hits in the Last level cache.

0x8208, L1I_CACHE_HIT_PRFM, Level 1 instruction cache software preload hit

The counter counts each software preload counted by L1I_CACHE_PRFM that hits in the Level 1 instruction or unified cache.

0x8209, L2I_CACHE_HIT_PRFM, Level 2 instruction cache software preload hit

The counter counts each software preload counted by L2I_CACHE_PRFM that hits in the Level 2 instruction or unified cache.
0x820C, L1D_CACHE_HIT_PRFM, Level 1 data cache software preload hit
   The counter counts each software preload counted by L1D_CACHE_PRFM that hits in the Level 1
data or unified cache.

0x820D, L2D_CACHE_HIT_PRFM, Level 2 data cache software preload hit
   The counter counts each software preload counted by L2D_CACHE_PRFM that hits in the Level 2
data or unified cache.

0x820E, L3D_CACHE_HIT_PRFM, Level 3 data cache software preload hit
   The counter counts each software preload counted by L3D_CACHE_PRFM that hits in the Level 3
data or unified cache.

0x820F, LL_CACHE_HIT_PRFM, Last level cache software preload hit
   The counter counts each software preload counted by LL_CACHE_PRFM that hits in the Last level
cache.

0x8214, L1D_CACHE_HITM_RD, Level 1 data cache demand hit modified, read
   The counter counts each cache access counted by L1D_CACHE_HIT_RD that accesses a
previously modified cache location.

0x8215, L2D_CACHE_HITM_RD, Level 2 data cache demand hit modified, read
   The counter counts each cache access counted by L2D_CACHE_HIT_RD that accesses a
previously modified cache location.

0x8216, L3D_CACHE_HITM_RD, Level 3 data cache demand hit modified, read
   The counter counts each cache access counted by L3D_CACHE_HIT_RD that accesses a
previously modified cache location.

0x8217, LL_CACHE_HITM_RD, Last level cache demand hit modified, read
   The counter counts each cache access counted by LL_CACHE_HIT_RD that accesses a previously
modified cache location.

0x8218, L1D_CACHE_HITM_WR, Level 1 data cache demand access hit modified, write
   The counter counts each cache access counted by L1D_CACHE_HIT_WR that accesses a
previously modified cache location.

0x8219, L2D_CACHE_HITM_WR, Level 2 data cache demand access hit modified, write
   The counter counts each cache access counted by L2D_CACHE_HIT_WR that accesses a
previously modified cache location.

0x821A, L3D_CACHE_HITM_WR, Level 3 data cache demand access hit modified, write
   The counter counts each cache access counted by L3D_CACHE_HIT_WR that accesses a
previously modified cache location.

0x821B, LL_CACHE_HITM_WR, Last level cache demand access hit modified, write
   The counter counts each cache access counted by LL_CACHE_HIT_WR that accesses a previously
modified cache location.

0x821C, L1D_CACHE_HITM_RW, Level 1 data cache demand access hit modified
   The counter counts each cache access counted by L1D_CACHE_HIT_RW that accesses a
previously modified cache location.

0x821D, L2D_CACHE_HITM_RW, Level 2 data cache demand access hit modified
   The counter counts each cache access counted by L2D_CACHE_HIT_RW that accesses a
previously modified cache location.
0x821E, L3D_CACHE_HITM_RW, Level 3 data cache demand access hit modified
The counter counts each cache access counted by L3D_CACHE_HIT_RW that accesses a previously modified cache location.

0x821F, LL_CACHE_HITM_RW, Last level cache demand access hit modified
The counter counts each cache access counted by LL_CACHE_HIT_RW that accesses a previously modified cache location.

0x8224, DSNP_HITM_RD, Snoop hit, demand data read, modified
The counter counts each snoop generated in response to a demand Memory-read operation counted by DSNP_HIT_RD that accesses a previously modified cache location.

0x8225, DSNP_HITM_NEAR_RD, Snoop hit in near cache, demand data read, modified
The counter counts each snoop generated in response to a demand Memory-read operation counted by DSNP_HIT_NEAR_RD that accesses a previously modified cache location.

0x8226, DSNP_HITM_FAR_RD, Snoop hit in far cache, demand data read, modified
The counter counts each snoop generated in response to a demand Memory-read operation counted by DSNP_HIT_FAR_RD that accesses a previously modified cache location.

0x8227, DSNP_HITM_REMOTE_RD, Snoop hit in remote cache, demand data read, modified
The counter counts each snoop generated in response to a demand Memory-read operation counted by DSNP_HIT_REMOTE_RD that accesses a previously modified cache location.

0x8228, DSNP_HITM_RW, Snoop hit, demand data write, modified
The counter counts each snoop generated in response to a demand Memory-write operation counted by DSNP_HIT_WR that accesses a previously modified cache location.

0x8229, DSNP_HITM_NEAR_WR, Snoop hit in near cache, demand data write, modified
The counter counts each snoop generated in response to a demand Memory-write operation counted by DSNP_HIT_NEAR_WR that accesses a previously modified cache location.

0x822A, DSNP_HITM_FAR_WR, Snoop hit in far cache, demand data write, modified
The counter counts each snoop generated in response to a demand Memory-write operation counted by DSNP_HIT_FAR_WR that accesses a previously modified cache location.

0x822B, DSNP_HITM_REMOTE_WR, Snoop hit in remote cache, demand data write, modified
The counter counts each snoop generated in response to a demand Memory-write operation counted by DSNP_HIT_REMOTE_WR that accesses a previously modified cache location.

0x822C, DSNP_HITM_RW, Snoop hit, demand data access, modified
The counter counts each snoop generated in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that accesses a previously modified cache location.

0x822D, DSNP_HITM_NEAR_RW, Snoop hit in near cache, demand data access, modified
The counter counts each snoop generated in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_NEAR_RW that accesses a previously modified cache location.

0x822E, DSNP_HITM_FAR_RW, Snoop hit in far cache, demand data access, modified
The counter counts each snoop generated in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_FAR_RW that accesses a previously modified cache location.
0x822F, DSNP_HITM_REMOTE_RW, Snoop hit in remote cache, demand data access, modified
The counter counts each snoop generated in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_REMOTE_RW that accesses a previously modified cache location.

0x8230, LOCAL_MEM, Access to memory attached to this device
The counter counts each Memory-read operation or Memory-write operation access to external memory attached to this device or socket. In a multi-socket system this means the socket that contains the PE. For more information, see REMOTE_ACCESS.

In systems where there multiple types of memory attached to this device with different performance characteristics, it is IMPLEMENTATION DEFINED whether accesses to all external memory types are counted by this event, or are classified as remote accesses and counted by REMOTE_MEM.

For example, in a system with an expansion memory connected to the device that has significantly higher latency than the main system memory, accesses to the main system memory might be counted by this event and accesses to the expansion memory by REMOTE_MEM.

0x8231, LOCAL_MEM_RD, Access to memory attached to this device, read
The counter counts each demand access counted by LOCAL_MEM_RW that is a Memory-read operation.

0x8232, LOCAL_MEM_WR, Access to memory attached to this device, write
The counter counts each demand access counted by LOCAL_MEM_RW that is a Memory-write operation.

0x8233, LOCAL_MEM_RW, Access to memory attached to this device, demand read or write
The counter counts each access counted by LOCAL_MEM that is a demand Memory-read operation or demand Memory-write operation.

It is IMPLEMENTATION DEFINED whether an access to external memory due to a prefetch to a cache is counted by LOCAL_MEM_RW or LOCAL_MEM_PRFM.

0x8234, LOCAL_MEM_PRFM, Access to memory attached to this device, preload or prefetch
The counter counts each access counted by LOCAL_MEM that is due to a preload or prefetch instruction, or a prefetch to a cache.

It is IMPLEMENTATION DEFINED whether an access to external memory due to a prefetch to a cache is counted by LOCAL_MEM_RW or LOCAL_MEM_PRFM.

0x8238, REMOTE_MEM, Access to memory attached to another socket in a multi-socket system
The counter counts each Memory-read operation or Memory-write operation access counted by REMOTE_ACCESS that causes an access to external memory attached to a different socket in a multi-socket system.

The counter also counts any accesses to external memory that are not counted by LOCAL_MEM because the implementation classifies them as remote accesses. For example, accesses to expansion memory connected to the device that has significantly higher latency than the main system memory might be classified as remote accesses. For more information, see LOCAL_MEM.

0x8239, REMOTE_MEM_RD, Access to memory attached to another socket in a multi-socket system, read
The counter counts each access counted by REMOTE_MEM_RW that is a Memory-read operation.

0x823A, REMOTE_MEM_WR, Access to memory attached to another socket in a multi-socket system, write
The counter counts each access counted by REMOTE_MEM_RW that is a Memory-write operation.

0x823B, REMOTE_MEM_RW, Access to memory attached to another socket in a multi-socket system, demand read or write
The counter counts each access counted by REMOTE_MEM that is a demand Memory-read operation or demand Memory-write operation.
0x823C, REMOTE_MEM_PRFM, Access to memory attached to another socket in a multi-socket system, preload or prefetch

The counter counts each access counted by REMOTE_MEM that is due to a preload or prefetch instruction, or a prefetch to a cache.

It is IMPLEMENTATION DEFINED whether an access to external memory due to a prefetch to a cache is counted by LOCAL_MEM_RW or LOCAL_MEM_PRFM.

0x8240, L1I_LFB_HIT_RD, Level 1 instruction cache demand fetch line-fill buffer hit

The counter counts each demand access counted by L1I_CACHE_HIT_RD that hits a recently fetched line in the Level 1 instruction or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x8241, L2I_LFB_HIT_RD, Level 2 instruction cache demand fetch line-fill buffer hit

The counter counts each demand access counted by L2I_CACHE_HIT_RD that hits a recently fetched line in the Level 2 instruction or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x8244, L1D_LFB_HIT_RD, Level 1 data cache demand line-fill buffer hit, read

The counter counts each demand access counted by L1D_CACHE_HIT_RD that hits a recently fetched line in the Level 1 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x8245, L2D_LFB_HIT_RD, Level 2 data cache demand line-fill buffer hit, read

The counter counts each demand access counted by L2D_CACHE_HIT_RD that hits a recently fetched line in the Level 2 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x8246, L3D_LFB_HIT_RD, Level 3 data cache demand line-fill buffer hit, read

The counter counts each demand access counted by L3D_CACHE_HIT_RD that hits a recently fetched line in the Level 3 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x8247, LL_LFB_HIT_RD, Last level cache demand line-fill buffer hit, read

The counter counts each demand access counted by LL_CACHE_HIT_RD that hits a recently fetched line in the Last level cache.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x8248, L1D_LFB_HIT_WR, Level 1 data cache demand access line-fill buffer hit, write

The counter counts each demand access counted by L1D_CACHE_HIT_WR that hits a recently fetched line in the Level 1 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.
0x8249, **L2D_LFB_HIT_WR**, Level 2 data cache demand access line-fill buffer hit, write

The counter counts each demand access counted by L2D_CACHE_HIT_WR that hits a recently fetched line in the Level 2 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x824A, **L3D_LFB_HIT_WR**, Level 3 data cache demand access line-fill buffer hit, write

The counter counts each demand access counted by L3D_CACHE_HIT_WR that hits a recently fetched line in the Level 3 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x824B, **LL_LFB_HIT_WR**, Last level cache demand access line-fill buffer hit, write

The counter counts each demand access counted by LL_CACHE_HIT_WR that hits a recently fetched line in the Last level cache.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x824C, **L1D_LFB_HIT_RW**, Level 1 data cache demand access line-fill buffer hit

The counter counts each demand access counted by L1D_CACHE_HIT_RW that hits a recently fetched line in the Level 1 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x824D, **L2D_LFB_HIT_RW**, Level 2 data cache demand access line-fill buffer hit

The counter counts each demand access counted by L2D_CACHE_HIT_RW that hits a recently fetched line in the Level 2 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x824E, **L3D_LFB_HIT_RW**, Level 3 data cache demand access line-fill buffer hit

The counter counts each demand access counted by L3D_CACHE_HIT_RW that hits a recently fetched line in the Level 3 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x824F, **LL_LFB_HIT_RW**, Last level cache demand access line-fill buffer hit

The counter counts each demand access counted by LL_CACHE_HIT_RW that hits a recently fetched line in the Last level cache.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

0x8250, **L1I_LFB_HIT_RD_FPRFM**, Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by software preload

The counter counts each demand fetch line-fill buffer first hit counted by L1I_LFB_HIT_RD where the cache line was fetched in response to a preload or prefetch instruction.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 1 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1I_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.
Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x8251, L21_LFB_HIT_RD_FPRFM, Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by software preload

The counter counts each demand fetch line-fill buffer first hit counted by L21_LFB_HIT_RD where the cache line was fetched in response to a preload or prefetch instruction.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 2 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L2I_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x8254, L1D_LFB_HIT_RD_FPRFM, Level 1 data cache demand line-fill buffer first hit, read, recently fetched by software preload

The counter counts each line-fill buffer first hit counted by L1D_LFB_HIT_RW_FPRFM that is a demand Memory-read operation.

0x8255, L2D_LFB_HIT_RD_FPRFM, Level 2 data cache demand line-fill buffer first hit, read, recently fetched by software preload

The counter counts each line-fill buffer first hit counted by L2D_LFB_HIT_RW_FPRFM that is a demand Memory-read operation.

0x8256, L3D_LFB_HIT_RD_FPRFM, Level 3 data cache demand line-fill buffer first hit, read, recently fetched by software preload

The counter counts each line-fill buffer first hit counted by L3D_LFB_HIT_RW_FPRFM that is a demand Memory-read operation.

0x8257, LL_LFB_HIT_RD_FPRFM, Last level cache demand line-fill buffer first hit, read, recently fetched by software preload

The counter counts each line-fill buffer first hit counted by LL_LFB_HIT_RW_FPRFM that is a demand Memory-read operation.

0x8258, L1D_LFB_HIT_WR_FPRFM, Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by software preload

The counter counts each demand access line-fill buffer first hit counted by L1D_LFB_HIT_RW_FPRFM where the cache line was fetched in response to a preload or prefetch instruction.

0x8259, L2D_LFB_HIT_WR_FPRFM, Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by software preload

The counter counts each demand access line-fill buffer first hit counted by L2D_LFB_HIT_RW_FPRFM where the cache line was fetched in response to a preload or prefetch instruction.

0x825A, L3D_LFB_HIT_WR_FPRFM, Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by software preload

The counter counts each demand access line-fill buffer first hit counted by L3D_LFB_HIT_RW_FPRFM where the cache line was fetched in response to a preload or prefetch instruction.

0x825B, LL_LFB_HIT_WR_FPRFM, Last level cache demand access line-fill buffer first hit, write, recently fetched by software preload

The counter counts each demand access line-fill buffer first hit counted by LL_LFB_HIT_RW_FPRFM where the cache line was fetched in response to a preload or prefetch instruction.

0x825C, L1D_LFB_HIT_RW_FPRFM, Level 1 data cache demand access line-fill buffer first hit, recently fetched by software preload

The counter counts each demand access line-fill buffer first hit counted by L1D_LFB_HIT_RW where the cache line was fetched in response to a preload or prefetch instruction.
That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x825D, L2D_LFB_HIT_RW_FPRFM, Level 2 data cache demand access line-fill buffer first hit, recently fetched by software preload

The counter counts each demand access line-fill buffer first hit counted by L2D_LFB_HIT_RW where the cache line was fetched in response to a preload or prefetch instruction.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L2D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x825E, L3D_LFB_HIT_RW_FPRFM, Level 3 data cache demand access line-fill buffer first hit, recently fetched by software preload

The counter counts each demand access line-fill buffer first hit counted by L3D_LFB_HIT_RW where the cache line was fetched in response to a preload or prefetch instruction.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L3D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x825F, LL_LFB_HIT_RW_FPRFM, Last level cache demand access line-fill buffer first hit, recently fetched by software preload

The counter counts each demand access line-fill buffer first hit counted by LL_LFB_HIT_RW where the cache line was fetched in response to a preload or prefetch instruction.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the LL_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x8260, L1I_LFB_HIT_RD_FHWPRF, Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand fetch line-fill buffer first hit counted by L1I_LFB_HIT_RD where the cache line was fetched by a hardware prefetcher.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 1 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1I_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x8261, L2I_LFB_HIT_RD_FHWPRF, Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand fetch line-fill buffer first hit counted by L2I_LFB_HIT_RD where the cache line was fetched by a hardware prefetcher.
That is, the fetch hits a cache line that is in the process of being loaded into the Level 2 instruction
or unified cache, and so does not generate a new refill, but has to wait for the previous refill to
complete, and the L2I_CACHE_REFILL_HWPRF event was generated when the cache line was
fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the
event is not generated again for the same cache line while it remains in the cache.

0x8264, L1D_LFB_HIT_RD_FHWPRF, Level 1 data cache demand line-fill buffer first hit, read, recently
fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by L1D_LFB_HIT_RW_FHWPRF that is a demand Memory-read operation.

0x8265, L2D_LFB_HIT_RD_FHWPRF, Level 2 data cache demand line-fill buffer first hit, read, recently
fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by L2D_LFB_HIT_RW_FHWPRF that is a demand Memory-read operation.

0x8266, L3D_LFB_HIT_RD_FHWPRF, Level 3 data cache demand line-fill buffer first hit, read, recently
fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by L3D_LFB_HIT_RW_FHWPRF that is a demand Memory-read operation.

0x8267, LL_LFB_HIT_RD_FHWPRF, Last level cache demand line-fill buffer first hit, read, recently fetched
by hardware prefetcher

The counter counts each line-fill buffer first hit counted by LL_LFB_HIT_RW_FHWPRF that is a demand Memory-read operation.

0x8268, L1D_LFB_HIT_WR_FHWPRF, Level 1 data cache demand access line-fill buffer first hit, write,
recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by L1D_LFB_HIT_RW_FHWPRF that is a demand Memory-write operation.

0x8269, L2D_LFB_HIT_WR_FHWPRF, Level 2 data cache demand access line-fill buffer first hit, write,
recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by L2D_LFB_HIT_RW_FHWPRF that is a demand Memory-write operation.

0x826A, L3D_LFB_HIT_WR_FHWPRF, Level 3 data cache demand access line-fill buffer first hit, write,
recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by L3D_LFB_HIT_RW_FHWPRF that is a demand Memory-write operation.

0x826B, LL_LFB_HIT_WR_FHWPRF, Last level cache demand access line-fill buffer first hit, write, recently fetched
by hardware prefetcher

The counter counts each line-fill buffer first hit counted by LL_LFB_HIT_RW_FHWPRF that is a demand Memory-write operation.

0x826C, L1D_LFB_HIT_RW_FHWPRF, Level 1 data cache demand access line-fill buffer first hit, recently
fetched by hardware prefetcher

The counter counts each demand access line-fill buffer first hit counted by L1D_LFB_HIT_RW
where the cache line was fetched by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or
unified cache, and so does not generate a new refill, but has to wait for the previous refill to
complete, and the L1D_CACHE_REFILL_HWPRF event was generated when the cache line was
fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the
event is not generated again for the same cache line while it remains in the cache.
0x826D, L2D_LFB_HIT_RW_FHWPRF, Level 2 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand access line-fill buffer first hit counted by L2D_LFB_HIT_RW where the cache line was fetched by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L2D_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x826E, L3D_LFB_HIT_RW_FHWPRF, Level 3 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand access line-fill buffer first hit counted by L3D_LFB_HIT_RW where the cache line was fetched by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L3D_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x826F, LL_LFB_HIT_RW_FHWPRF, Last level cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand access line-fill buffer first hit counted by LL_LFB_HIT_RW where the cache line was fetched by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the LL_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

0x8270, L1L_LFB_HIT_RD_FPRF, Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by preload or prefetch

The counter counts each line-fill buffer first hit counted by either L1L_LFB_HIT_RD_FHWPRF or L1L_LFB_HIT_RD_FPRFM.

0x8271, L2L_LFB_HIT_RD_FPRF, Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by preload or prefetch

The counter counts each line-fill buffer first hit counted by either L2L_LFB_HIT_RD_FHWPRF or L2L_LFB_HIT_RD_FPRFM.

0x8274, L1D_LFB_HIT_RD_FPRF, Level 1 data cache demand line-fill buffer first hit, read, recently fetched by preload or prefetch

The counter counts each line-fill buffer first hit counted by either L1D_LFB_HIT_RD_FHWPRF or L1D_LFB_HIT_RD_FPRFM.

0x8275, L2D_LFB_HIT_RD_FPRF, Level 2 data cache demand line-fill buffer first hit, read, recently fetched by preload or prefetch

The counter counts each line-fill buffer first hit counted by either L2D_LFB_HIT_RD_FHWPRF or L2D_LFB_HIT_RD_FPRFM.
0x8276, L3D_LFB_HIT_RD_FPRF, Level 3 data cache demand line-fill buffer first hit, read, recently fetched by preload or prefetch
The counter counts each line-fill buffer first hit counted by either L3D_LFB_HIT_RD_FHWPRF or L3D_LFB_HIT_RD_FPRFM.

0x8277, LL_LFB_HIT_RD_FPRF, Last level cache demand line-fill buffer first hit, read, recently fetched by preload or prefetch
The counter counts each line-fill buffer first hit counted by either LL_LFB_HIT_RD_FHWPRF or LL_LFB_HIT_RD_FPRFM.

0x8278, L1D_LFB_HIT_WR_FPRF, Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by preload or prefetch
The counter counts each line-fill buffer first hit counted by either L1D_LFB_HIT_WR_FHWPRF or L1D_LFB_HIT_WR_FPRFM.

0x8279, L2D_LFB_HIT_WR_FPRF, Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by preload or prefetch
The counter counts each line-fill buffer first hit counted by either L2D_LFB_HIT_WR_FHWPRF or L2D_LFB_HIT_WR_FPRFM.

0x827A, L3D_LFB_HIT_WR_FPRF, Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by preload or prefetch
The counter counts each line-fill buffer first hit counted by either L3D_LFB_HIT_WR_FHWPRF or L3D_LFB_HIT_WR_FPRFM.

0x827B, LL_LFB_HIT_WR_FPRF, Last level cache demand access line-fill buffer first hit, write, recently fetched by preload or prefetch
The counter counts each line-fill buffer first hit counted by either LL_LFB_HIT_WR_FHWPRF or LL_LFB_HIT_WR_FPRFM.

0x827C, L1D_LFB_HIT_RW_FPRF, Level 1 data cache demand access line-fill buffer first hit, recently fetched by preload or prefetch
The counter counts each line-fill buffer first hit counted by either L1D_LFB_HIT_RW_FHWPRF or L1D_LFB_HIT_RW_FPRFM.

0x827D, L2D_LFB_HIT_RW_FPRF, Level 2 data cache demand access line-fill buffer first hit, recently fetched by preload or prefetch
The counter counts each line-fill buffer first hit counted by either L2D_LFB_HIT_RW_FHWPRF or L2D_LFB_HIT_RW_FPRFM.

0x827E, L3D_LFB_HIT_RW_FPRF, Level 3 data cache demand access line-fill buffer first hit, recently fetched by preload or prefetch
The counter counts each line-fill buffer first hit counted by either L3D_LFB_HIT_RW_FHWPRF or L3D_LFB_HIT_RW_FPRFM.

0x827F, LL_LFB_HIT_RW_FPRF, Last level cache demand access line-fill buffer first hit, recently fetched by preload or prefetch
The counter counts each line-fill buffer first hit counted by either LL_LFB_HIT_RW_FHWPRF or LL_LFB_HIT_RW_FPRFM.

0x8280, L1I_CACHE_PRF, Level 1 instruction cache, preload or prefetch hit
The counter counts each fetch counted by either L1I_CACHE_HWPRF or L1I_CACHE_PRFM.

0x8281, L2I_CACHE_PRF, Level 2 instruction cache, preload or prefetch hit
The counter counts each fetch counted by either L2I_CACHE_HWPRF or L2I_CACHE_PRFM.

0x8284, L1D_CACHE_PRF, Level 1 data cache, preload or prefetch hit
The counter counts each fetch counted by either L1D_CACHE_HWPRF or L1D_CACHE_PRFM.
0x8285, L2D_CACHE_PRF, Level 2 data cache, preload or prefetch hit
The counter counts each fetch counted by either L2D_CACHE_HWPRF or L2D_CACHE_PRFM.

0x8286, L3D_CACHE_PRF, Level 3 data cache, preload or prefetch hit
The counter counts each fetch counted by either L3D_CACHE_HWPRF or L3D_CACHE_PRFM.

0x8287, LL_CACHE_PRF, Last level cache, preload or prefetch hit
The counter counts each fetch counted by either LL_CACHE_HWPRF or LL_CACHE_PRFM.

0x8288, L1I_CACHE_REFILL_PRF, Level 1 instruction cache refill, preload or prefetch hit
The counter counts each refill counted by either L1I_CACHE_REFILL_HWPRF or L1I_CACHE_REFILL_PRFM.

0x8289, L2I_CACHE_REFILL_PRF, Level 2 instruction cache refill, preload or prefetch hit
The counter counts each refill counted by either L2I_CACHE_REFILL_HWPRF or L2I_CACHE_REFILL_PRFM.

0x828C, L1D_CACHE_REFILL_PRF, Level 1 data cache refill, preload or prefetch hit
The counter counts each refill counted by either L1D_CACHE_REFILL_HWPRF or L1D_CACHE_REFILL_PRFM.

0x828D, L2D_CACHE_REFILL_PRF, Level 2 data cache refill, preload or prefetch hit
The counter counts each refill counted by either L2D_CACHE_REFILL_HWPRF or L2D_CACHE_REFILL_PRFM.

0x828E, L3D_CACHE_REFILL_PRF, Level 3 data cache refill, preload or prefetch hit
The counter counts each refill counted by either L3D_CACHE_REFILL_HWPRF or L3D_CACHE_REFILL_PRFM.

0x828F, LL CACHE REFILL PRF, Last level cache refill, preload or prefetch hit
The counter counts each refill counted by either LL_CACHE_REFILL_HWPRF or LL_CACHE_REFILL_PRFM.

0x8290, ISNP_HIT_PRF, Snoop hit, any instruction prefetch
The counter counts each snoop counted by either ISNP_HIT_HWPRF or ISNP_HIT_PRFM.

0x8291, ISNP_HIT_NEAR_PRF, Snoop hit in near cache, instruction preload or prefetch
The counter counts each snoop counted by either ISNP_HIT_NEAR_HWPRF or ISNP_HIT_NEAR_PRFM, that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

0x8292, ISNP_HIT_FAR_PRF, Snoop hit in far cache, instruction preload or prefetch
The counter counts each snoop counted by either ISNP_HIT_FAR_HWPRF or ISNP_HIT_FAR_PRFM, that hits in a cache outside the local PE cluster on the same device.

0x8293, ISNP_HIT_REMOTE_PRF, Snoop hit in remote cache, instruction preload or prefetch
The counter counts each snoop counted by either ISNP_HIT_REMOTE_HWPRF or ISNP_HIT_REMOTE_PRFM, that hits in a cache on a remote device.

0x8294, DSNP_HIT_PRF, Snoop hit, any data prefetch
The counter counts each snoop counted by either DSNP_HIT_HWPRF or DSNP_HIT_PRFM.

0x8295, DSNP_HIT_NEAR_PRF, Snoop hit in near cache, data preload or prefetch
The counter counts each snoop counted by either DSNP_HIT_NEAR_HWPRF or DSNP_HIT_NEAR_PRFM, that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.
0x8296, **DSNP_HIT_FAR_PRF, Snoop hit in far cache, data preload or prefetch**

The counter counts each snoop counted by either **DSNP_HIT_FAR_HWPRF** or **DSNP_HIT_FAR_PRFM**, that hits in a cache outside the local PE cluster on the same device.

0x8297, **DSNP_HIT_REMOTE_PRF, Snoop hit in remote cache, data preload or prefetch**

The counter counts each snoop counted by either **DSNP_HIT_REMOTE_HWPRF** or **DSNP_HIT_REMOTE_PRFM**, that hits in a cache on a remote device.

0x8298, **LL_CACHE_RW, Last level cache demand access**

The counter counts each access counted by **LL_CACHE** that is due to a demand Memory-read operation or demand Memory-write operation.

This includes:
- Accesses made by Speculatively executed instructions.
- Accesses to the Last level cache due to a refill of another cache caused by a demand Memory-read operation or demand Memory-write operation.

This event must be implemented if any of the following are true:
- Event **LL_CACHE_PRFM** is implemented.
- Event **LL_CACHE_HWPRF** is implemented.

0x8299, **LL_CACHE_PRFM, Last level cache software preload**

The counter counts each access counted by **LL_CACHE** that is due to a preload or prefetch instruction.

This includes accesses to the Last level cache due to a refill of another cache caused by a preload or prefetch instruction.

This event must be implemented if event **LL_CACHE_RW** is implemented.

0x829a, **LL_CACHE_REFILL, Last level cache refill**

The counter counts each access counted by **LL_CACHE** that causes a refill of the Last level cache, or any other data, instruction, or unified cache of this PE, from outside of those caches.

A refill includes any access that causes data to be fetched from outside of the Level 1 to Last level caches, even if the data is ultimately not allocated into the Last level cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:
- Accesses to the Last level cache that cause a refill that is satisfied by fetching data from memory, or a Last level cache of another PE.
- Refills of and write-backs from any other data, instruction, or unified caches of this PE that cause a refill from outside of the Level 1 to Last level caches of this PE.
- Accesses to the Last level cache that cause a refill of a higher level cache of this PE from outside of the Level 1 to Last level caches of this PE, even if there is no refill of the Last level cache.

The counter does not count accesses that:
- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.
0x829B, LL_CACHE_REFILL_PRFM, Last level cache refill, software preload
The counter counts each access counted by LL_CACHE_PRFM that causes a refill of the Last level cache, or any other data, instruction, or unified cache of this PE, from outside of those caches.

0x829C, LL_CACHE_WB, Last level cache write-back
The counter counts each write-back of data from the Last level cache to outside of the Level 1 to Last level caches. For example:
- A write-back of a dirty cache line to memory.
- A write-back of a recently fetched cache line that has not been allocated to the Last level cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:
- A transfer of a dirty cache line from the Last level cache to outside of Level 1 to Last level caches made as a result of a coherency request.
- Write-backs made as a result of cache maintenance instructions.

The counter does not count:
- The invalidation of a cache line without any write-back to outside of the Level 1 to Last level caches or memory.
- Writes that write through the Last level cache to outside of the Level 1 to Last level caches.
- Transfers of data from the Last level cache to another cache to satisfy a refill of the other cache.

A write-back is attributable to the agent that generated the request that caused the write-back. This might not be the same agent that caused the data being written back to be allocated into the cache. An Unattributable write-back event occurs when a requestor outside of the PE makes a coherency request that results in write-back. If the cache is shared, then an Unattributable write-back event is not counted. If the cache is not shared, then the event is counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines software is streaming writes to memory and does not allocate lines to the cache, or by a DC ZVA operation.

0x829D, LL_CACHE_WR, Last level cache access, write
If the LL_CACHE_RW event is implemented, the counter counts each access counted by LL_CACHE_RW that is a Memory-write operation.

If the LL_CACHE_RW event is not implemented, the counter counts each access counted by LL_CACHE that is a Memory-write operation.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

0x829F, LL_CACHE_REFILL_WR, Last level cache refill, write
The counter counts each access counted by both LL_CACHE_REFILL and LL_CACHE_WR.
That is, every refill of the Last level cache counted by LL_CACHE_REFILL that is caused by a Memory-write operation.

If the cache is shared, the counter counts only events Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0b1, other PEs in the multithreaded implementation. If the cache is not shared, all events are counted.

0x82A0, MEM_ACCESS_RW, Data memory access, demand access
The counter counts each access counted by MEM_ACCESS that is a demand Memory-read operation or demand Memory-write operation.
This includes accesses made by speculative instructions.
0x82A1, **INST_FETCH_RD, Instruction memory access, demand fetch**

The counter counts each fetch counted by **INST_FETCH** that is a demand fetch.

This includes any fetch made for a speculative instruction.

0x82A2, **MEM_ACCESS_PRFM, Data memory access, preload**

The counter counts each access counted by **MEM_ACCESS** that is due to a software preload instruction.

0x82A3, **INST_FETCH_PRFM, Instruction memory access, preload**

The counter counts each access counted by **INST_FETCH** that is due to a software preload instruction.

### D11.11.4 Cycle event counting

The **CPU_CYCLES** event and the cycle counter, **PMCCNTR**, count cycles. The duration of a cycle is subject to any changes in clock frequency, including clock stopping caused by the WFI and WFE instructions.

It is implementation specific whether **CPU_CYCLES** and **PMCCNTR** count when the PE is in WFI or WFE state, even if the clocks are not stopped.

In addition, events such as **STALL**, **STALL_FRONTEND** and **STALL_BACKEND** that are defined to only count cycles that are counted by the **CPU_CYCLES** event have the same limitation.

### Multithreaded implementations

Multithreaded implementations can have various forms, some examples of these are:

- **Simultaneous Multithreading** (SMT), where every PE thread is active on every Processor cycle.
- **Fine-grained Multithreading** (FGMT), also known as a Barrel processor, where one PE thread is active on each Processor cycle, and this changes regularly.
- **Switch on Event Multithreading** (SoEMT), also known as **Coarse-grained Multithreading** (CGMT), where high latency events cause the processor to switch the active PE thread.

In the above examples, active means that the PE might execute the instructions. A PE can be active but not executing instructions when no instruction is available or because of limited execution resources.

It is implementation specific whether a thread is active when the thread is in WFE or WFI state. This applies for all forms of multithreaded implementation.

When the PMU implementation supports multithreading, and the **Effective value** of **PMEVTYPER<n>_EL0.MT** bit is 0, the **CPU_CYCLES** event does not count Processor cycles on which the thread was not active. For the example multithreaded implementations, this means that, if the event counter is enabled, event counting is not prohibited, and the thread is not in WFE or WFI state:

- For an SMT implementation, the **CPU_CYCLES** event counts every Processor cycle.
- For a particular FGMT implementation, that alternates between two threads on each Processor cycle, the **CPU_CYCLES** event counts every other Processor cycle.
- For a particular SoEMT implementation, that is waiting for a long latency operation, the **CPU_CYCLES** event does not count Processor cycles, as the PE thread is not active.

If the **Effective value** of **PMEVTYPER<n>_EL0.MT** bit is 1, the **CPU_CYCLES** event counts each Processor cycle, and can only count a maximum of one each Processor cycle.

Events that only count cycles that are counted by the **CPU_CYCLES** event have the same limitation. For example, in an SMT implementation, if a PE thread cannot issue an instruction because of contention with other PE threads, these are counted as **STALL_BACKEND** cycles.

If the **Effective value** of **PMEVTYPER<n>_EL0.MT** bit is 1, the PE only counts cycles on which no operation is issued from any thread.
The cycle counter, PMCCNTR, is not affected by whether the thread is active or inactive. When enabled, PMCCNTR counts every processor cycle.

See Multithreaded implementations on page D11-5258, MDCR_EL3.MTPME, SDCR.MTPME, MDCR_EL2.MTPME, and HDR.C.MTPME for more information about when the Effective value of PMEVTYPE<\n>_EL0.MT is 0.

D11.11.5 Meaningful combinations of common microarchitectural events

The architecture highlights some meaningful combinations of common microarchitectural events. The following tables list the highlighted combinations:

- Table D11-9 on page D11-5422, REFILL events and associated access events.
- Table D11-10 on page D11-5423, Meaningful combinations of latency events.
- Table D11-11 on page D11-5423, Cache hit events and associated data source events.
- Table D11-12 on page D11-5424, Meaningful combinations of cache line state tracking events.
- Table D11-13 on page D11-5424, Meaningful combinations of TLB events.

### Table D11-9 REFILL events and associated access events

<table>
<thead>
<tr>
<th>Numerator</th>
<th>Denominator</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001 L1I_CACHE_REFILL</td>
<td>0x0014 L1I_CACHE</td>
<td>Attributable Level 1 instruction cache refill rate</td>
</tr>
<tr>
<td>0x0002 L1I_TLB_REFILL</td>
<td>0x0026 L1I_TLB</td>
<td>Attributable Level 1 instruction TLB refill rate</td>
</tr>
<tr>
<td>0x0003 L1D_CACHE_REFILL</td>
<td>0x0004 L1D_CACHE</td>
<td>Attributable Level 1 data or unified cache refill rate</td>
</tr>
<tr>
<td>0x0005 L1D_TLB_REFILL</td>
<td>0x0025 L1D_TLB</td>
<td>Attributable Level 1 data or unified TLB refill rate</td>
</tr>
<tr>
<td>0x0017 L2D_CACHE_REFILL</td>
<td>0x0016 L2D_CACHE</td>
<td>Attributable Level 2 data or unified cache refill rate</td>
</tr>
<tr>
<td>0x0028 L2I_CACHE_REFILL</td>
<td>0x0027 L2I_CACHE</td>
<td>Attributable Level 2 instruction cache refill rate</td>
</tr>
<tr>
<td>0x002A L3D_CACHE_REFILL</td>
<td>0x002B L3D_CACHE</td>
<td>Attributable Level 3 data or unified cache refill rate</td>
</tr>
<tr>
<td>0x002D L2D_TLB_REFILL</td>
<td>0x002F L2D_TLB</td>
<td>Attributable Level 2 data or unified TLB refill rate</td>
</tr>
<tr>
<td>0x002E L2I_TLB_REFILL</td>
<td>0x0030 L2I_TLB</td>
<td>Attributable Level 2 instruction TLB refill rate</td>
</tr>
<tr>
<td>0x0029 BUS_ACCESS</td>
<td>0x001D BUS_CYCLES</td>
<td>Attributable Bus accesses per cycle</td>
</tr>
<tr>
<td>0x0033 LL_CACHE_MISS</td>
<td>0x0032 LL_CACHE</td>
<td>Attributable Last Level data or unified cache refill rate</td>
</tr>
<tr>
<td>0x0034 DTLB_WALK</td>
<td>0x0025 L1D_TLB</td>
<td>Attributable data TLB miss rate</td>
</tr>
<tr>
<td>0x0035 ITLB_WALK</td>
<td>0x0026 L1I_TLB</td>
<td>Attributable instruction TLB miss rate</td>
</tr>
<tr>
<td>0x0037 LL_CACHE_MISS_RD</td>
<td>0x0036 LL_CACHE_RD</td>
<td>Attributable memory read operation miss rate</td>
</tr>
<tr>
<td>0x0038 REMOTE_ACCESS_RD</td>
<td>0x0031 REMOTE_ACCESS</td>
<td>Attributable read accesses to another socket in a multi-socket system</td>
</tr>
<tr>
<td>0x0042 L1D_CACHE_REFILL_WR</td>
<td>0x0040 L1D_CACHE_WR</td>
<td>Attributable Level 1 cache refill rate, write</td>
</tr>
<tr>
<td>0x0043 L1D_CACHE_REFILL_RD</td>
<td>0x0041 L1D_CACHE_WR</td>
<td>Attributable Level 1 cache refill rate, read</td>
</tr>
<tr>
<td>0x004C L1D_TLB_REFILL_WR</td>
<td>0x004E L1D_TLB_WR</td>
<td>Attributable Level 1 TLB refill rate, write</td>
</tr>
<tr>
<td>0x004D L1D_TLB_REFILL_RD</td>
<td>0x004F L1D_TLB_WR</td>
<td>Attributable Level 1 TLB refill rate, read</td>
</tr>
<tr>
<td>0x0052 L2D_CACHE_REFILL_RD</td>
<td>0x0050 L2D_CACHE_WR</td>
<td>Attributable Level 2 data cache refill rate, read</td>
</tr>
</tbody>
</table>
## Table D11-9 REFILL events and associated access events (continued)

<table>
<thead>
<tr>
<th>Numerator</th>
<th>Denominator</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0053 L2D_CACHE_REFILL_WR</td>
<td>0x0051 L2D_CACHE_WR</td>
<td>Attributable Level 2 data cache refill rate, write</td>
</tr>
<tr>
<td>0x005C L2D_TLB_REFILL_RD</td>
<td>0x005E L2D_TLB_RD</td>
<td>Attributable Level 2 data TLB refill rate, read</td>
</tr>
<tr>
<td>0x005D L2D_TLB_REFILL_WR</td>
<td>0x005F L2D_TLB_WR</td>
<td>Attributable Level 2 data TLB refill rate, write</td>
</tr>
<tr>
<td>0x00A2 L3D_CACHE_REFILL_RD</td>
<td>0x00A0 L3D_CACHE_RD</td>
<td>Attributable Level 3 data cache refill rate, read</td>
</tr>
<tr>
<td>0x00A3 L3D_CACHE_REFILL_WR</td>
<td>0x00A1 L3D_CACHE_WR</td>
<td>Attributable Level 3 data cache refill rate, write</td>
</tr>
</tbody>
</table>

## Table D11-10 Meaningful combinations of latency events

<table>
<thead>
<tr>
<th>Numerator</th>
<th>Denominator</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8120 INST_FETCH_PERCYC</td>
<td>0x8124 INST_FETCH</td>
<td>Mean duration of instruction fetch events in processor cycles</td>
</tr>
<tr>
<td>0x8121 MEM_ACCESS_RD_PERCYC</td>
<td>0x0066 MEM_ACCESS_RD</td>
<td>Mean duration of memory read access events in processor cycles</td>
</tr>
<tr>
<td>0x8125, BUS_REQ_RD_PERCYC</td>
<td>0x818D, BUS_REQ_RD</td>
<td>Bus read transaction average latency</td>
</tr>
<tr>
<td>0x8128 DTLB_WALK_PERCYC</td>
<td>0x0034 DTLB_WALK</td>
<td>Mean duration of data or unified TLB walk events in processor cycles</td>
</tr>
<tr>
<td>0x8129 ITLB_WALK_PERCYC</td>
<td>0x0035 ITLB_WALK</td>
<td>Mean duration of instruction TLB walk events in processor cycles</td>
</tr>
</tbody>
</table>

## Table D11-11 Cache hit events and associated data source events

<table>
<thead>
<tr>
<th>Numerator</th>
<th>Denominator</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x820C, L1D_CACHE_HIT_RFM&lt;sup&gt;a&lt;/sup&gt;</td>
<td>0x8142, L1D_CACHE_PRFM&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Level 1 data or unified cache software preload hit ratio&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>0x81CC, L1D_CACHE_HIT_RW&lt;sup&gt;a&lt;/sup&gt;</td>
<td>0x82A0, MEM_ACCESS_RW</td>
<td>Level 1 data or unified cache demand cache hit rate&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>0x81CF, LL_CACHE_HIT_RW</td>
<td>0x82A0, MEM_ACCESS_RW</td>
<td>Last level cache demand cache hit rate</td>
</tr>
<tr>
<td>0x8232, LOCAL_MEM_RW</td>
<td>0x82A0, MEM_ACCESS_RW</td>
<td>Local memory demand access rate</td>
</tr>
<tr>
<td>0x823B, REMOTE_MEM_RW</td>
<td>0x82A0, MEM_ACCESS_RW</td>
<td>Remote memory demand access rate</td>
</tr>
<tr>
<td>0x821D, L2D_CACHE_HITM_RW&lt;sup&gt;a&lt;/sup&gt;</td>
<td>0x81CD, L2D_CACHE_HIT_RW&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Level 2 data or unified cache demand cache hit rate&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

<sup>a</sup> Similar ratios can be defined for other caches.
D11.11.6 Meaningful combinations of SVE events

Scalar-equivalent operations

The number of speculatively executed operations performed on individual scalar values, assuming that all SVE vector elements are active, can be determined from a pair of event counters. For example, the total number of individual floating-point operations performed can be computed as follows:

\[
\text{FP\_SCALE\_OPS\_SPEC} \times \text{VL} \div 128 + \text{FP\_FIXED\_OPS\_SPEC}
\]

A summary of these event pairs is given below. Combined multiply-add and multiply-subtract instructions are counted as two operations per element.

<table>
<thead>
<tr>
<th>Operation type</th>
<th>Scalable operations</th>
<th>Fixed width operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-point operations (any precision)</td>
<td>FP_SCALE_OPS_SPEC</td>
<td>FP_FIXED_OPS_SPEC</td>
</tr>
<tr>
<td>Half-precision floating-point operations</td>
<td>FP_HP_SCALE_OPS_SPEC</td>
<td>FP_HP_FIXED_OPS_SPEC</td>
</tr>
<tr>
<td>Single-precision floating-point operations</td>
<td>FP_SP_SCALE_OPS_SPEC</td>
<td>FP_SP_FIXED_OPS_SPEC</td>
</tr>
<tr>
<td>Double-precision floating-point operation</td>
<td>FP_DP_SCALE_OPS_SPEC</td>
<td>FP_DP_FIXED_OPS_SPEC</td>
</tr>
<tr>
<td>Integer operations (any size)</td>
<td>INT_SCALE_OPS_SPEC</td>
<td>INT_FIXED_OPS_SPEC</td>
</tr>
<tr>
<td>Load/store accesses (any size)</td>
<td>LDST_SCALE_OPS_SPEC</td>
<td>LDST_FIXED_OPS_SPEC</td>
</tr>
<tr>
<td>Load accesses (any size)</td>
<td>LD_SCALE_OPS_SPEC</td>
<td>LD_FIXED_OPS_SPEC</td>
</tr>
<tr>
<td>Store accesses (any size)</td>
<td>ST_SCALE_OPS_SPEC</td>
<td>ST_FIXED_OPS_SPEC</td>
</tr>
</tbody>
</table>
Bytes loaded and stored

The number of bytes speculatively loaded from memory or stored to memory, assuming that all SVE vector elements are active, can be determined from a pair of event counters. For example, the total number of bytes loaded from memory can be computed as follows:

\[ \text{LD\_SCALE\_BYTES\_SPEC} \times \text{VL} \div 128 + \text{LD\_FIXED\_BYTES\_SPEC} \]

A summary of the total byte count pairs is as follows:

<table>
<thead>
<tr>
<th>Operation type</th>
<th>Scalable operations</th>
<th>Fixed width operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/store byte count</td>
<td>LDST_SCALE_BYTES_SPEC</td>
<td>LDST_FIXED_BYTES_SPEC</td>
</tr>
<tr>
<td>Load byte count</td>
<td>LD_SCALE_BYTES_SPEC</td>
<td>LD_FIXED_BYTES_SPEC</td>
</tr>
<tr>
<td>Store byte count</td>
<td>ST_SCALE_BYTES_SPEC</td>
<td>ST_FIXED_BYTES_SPEC</td>
</tr>
</tbody>
</table>

Overall vector utilization

Vector utilization rates for SVE events which ignore the number of Active elements can be estimated by adjusting them using the following ratios:

<table>
<thead>
<tr>
<th>Numerator</th>
<th>Denominator</th>
<th>Utilization rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVE_PRED_FULL_SPEC</td>
<td>SVE_PRED_SPEC</td>
<td>All predicates active</td>
</tr>
<tr>
<td>SVE_PRED_PARTIAL_SPEC</td>
<td>SVE_PRED_SPEC</td>
<td>Partial predicates active</td>
</tr>
<tr>
<td>SVE_PRED_EMPTY_SPEC</td>
<td>SVE_PRED_SPEC</td>
<td>No predicates active</td>
</tr>
</tbody>
</table>

Vector loop efficiency

The effectiveness with which sequential or scalar source loops are vectorized can be estimated using ratios of the SVE\_PLOOP\_*\_SPEC predicated loop events, as shown in the following table:

<table>
<thead>
<tr>
<th>Numerator</th>
<th>Denominator</th>
<th>Vector loop metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVE_PLOOP_ELTS_SPEC \times VL \div 128</td>
<td>SVE_PLOOP_TERM_SPEC</td>
<td>Source level iterations per loop</td>
</tr>
<tr>
<td>SVE_PLOOP_TEST_SPEC</td>
<td>SVE_PLOOP_TERM_SPEC</td>
<td>Vectorized iterations per loop</td>
</tr>
<tr>
<td>SVE_PLOOP_ELTS_SPEC \times VL \div 128</td>
<td>SVE_PLOOP_TEST_SPEC</td>
<td>Parallelism per vector loop</td>
</tr>
</tbody>
</table>

D11.11.7 Required events

**FEAT\_PMUv3** requires that an implementation includes the following common events:

- \(0x0000\), SW\_INCR, Instruction architecturally executed, Condition code check pass, software increment.
- \(0x0003\), L1D\_CACHE\_REFILL, Level 1 data cache refill.

--- Note ---
Event \(0x0003\) is only required if the implementation includes a Level 1 data or unified cache.

--- Note ---
Event \(0x0004\), L1D\_CACHE, Level 1 data cache access.

--- Note ---
Event \(0x0004\) is only required if the implementation includes a Level 1 data or unified cache.
The Performance Monitors Extension

D11.11 PMU events and event numbers

• 0x0010, BR_MIS_PRED, Mispredicted or not predicted branch Speculatively executed.
  
  **Note**
  Event 0x0010 is only required if the implementation includes program-flow prediction. However, Arm strongly recommends that the event is implemented as described in Common microarchitectural events on page D11-5317.

• 0x0011, CPU_CYCLES, Cycle.

• 0x0012, BR_PRED, Predictable branch Speculatively executed.
  
  **Note**
  Event 0x0012 is only required if the implementation includes program-flow prediction. However, Arm recommends that the event is implemented as described in Common microarchitectural events on page D11-5317.

• At least one of:
  — 0x0008, INST_RETIRED, Instruction architecturally executed.
  — 0x001B, INST_SPEC, Operation Speculatively executed.
  
  **Note**
  Arm strongly recommends that event 0x008 is implemented.

• When FEAT_PMUv3p1 is implemented:
  — 0x0023, STALL_FRONTEND, No operation issued due to the frontend.
  — 0x0024, STALL_BACKEND, No operation issued due to the backend.

• When The Scalable Vector Extension (SVE) is implemented, at least one of:
  — 0x8002, SVE_INST_RETIRED, SVE instruction architecturally retired.
  — 0x8006, SVE_INST_SPEC, SVE operation speculatively executed.

• When FEAT_SPE is implemented:
  — 0x4000, SAMPLE_POP, Statistical Profiling sample population.
  — 0x4001, SAMPLE_FEED, Statistical Profiling sample taken.
  — 0x4002, SAMPLE_FILTRATE, Statistical Profiling sample filtered.
  — 0x4003, SAMPLE_COLLISION, Statistical Profiling sample collision.

• When FEAT_PMUv3p4 is implemented:
  — 0x003C, STALL_SLOT_BACKEND, No operation sent for execution on a Slot due to the backend.
  — 0x003D, STALL_SLOT_FRONTEND, No operation sent for execution on a Slot due to the frontend.
  — 0x003E, STALL_SLOT, No operation sent for execution on a Slot.

Arm strongly recommends that the following events are implemented:

• 0x0021, BR_RETIRED.
• 0x0022, BR_MIS_PRED_RETIRED.
• 0x003A, OP_RETIRED.

When any of the following common events are implemented, all three of them are implemented:

• 0x003D, STALL_SLOT_BACKEND, No operation sent for execution on a Slot due to the backend,
• 0x003E, STALL_SLOT_FRONTEND, No operation sent for execution on a Slot due to the frontend,
• 0x003F, STALL_SLOT, No operation sent for execution on a Slot.

When any of the following common events are implemented, all three of them are implemented:

• 0x812A, SAMPLE_FEED_BR, Statistical Profiling sample taken, branch.
• 0x812B, SAMPLE_FEED_LD, Statistical Profiling sample taken, load.
• 0x812C, SAMPLE_FEED_ST, Statistical Profiling sample taken, store.
• 0x812D, SAMPLE_FEED_OP, Statistical Profiling sample taken, matching operation type.
• 0x812E, SAMPLE_FEED_EVENT, Statistical Profiling sample taken, matching events.
• 0x812F, SAMPLE_FEED_LAT, Statistical Profiling sample taken, exceeding minimum latency.

Arm strongly recommends that the following events are implemented:

• 0x0021, BR_RETIRED.
• 0x0022, BR_MIS_PRED_RETIRED.
• 0x003A, OP_RETIRED.
The Performance Monitors Extension

D11.11 PMU events and event numbers

D11.11.8 IMPLEMENTATION DEFINED event numbers

Arm recommends that implementers establish a standardized numbering scheme for their IMPLEMENTATION DEFINED events, with common definitions, and common event numbers, applied to all of their implementations. In general, the recommended approach is for standardization across implementations with common features. However, Arm recognizes that attempting to standardize the encoding of microarchitectural features across too wide a range of implementations is not productive.

The Arm architecture guarantees not to define any event prefixed with IMP_ as part of the standard Arm architecture.

Arm strongly recommends that at least the following classes of event are identified in the IMPLEMENTATION DEFINED events:

- Separating each of the STALL_FRONTEND and STALL_SLOT_FRONTEND events to count holes in instruction availability.
- Separating each of the STALL_BACKEND and STALL_SLOT_BACKEND events, to count, for example, cumulative duration of stalls, unavailability of execution resources, or missed superscalar issue opportunities.
- Miss rates for additional levels of caches and TLBs.
- Any external events passed to the PE through an IMPLEMENTATION DEFINED mechanism.
- Cumulative duration of a PSTATE.{A, I, F} interrupt mask set to 1.
- Cumulative occupancy for resource queues, such as data access queues, and entry/exit counts, so that average latencies can be determined, separating out counts for key resources that might exist. An implementation might also provide registers in the IMPLEMENTATION DEFINED space to further extend such counts, for example by specifying a minimum latency for an event to be counted.
- Any other microarchitectural features that the implementer considers are valuable to count.

The range of possible IMPLEMENTATION DEFINED event numbers is described in The PMU event number space and common events on page D11-5275.
D11.12 Performance Monitors Extension registers

Further information on the Performance Monitors Extension Registers can be found in the following sections:

• Table K17-2 on page K17-11838 lists the Performance Monitors register names for AArch32 and AArch64 states.

• Performance monitors registers on page K17-11871 summarizes the Performance Monitors Extension registers in AArch64 state.

• Performance monitors registers on page K17-11901 summarizes the Performance Monitors Extension registers in AArch32 state.
This chapter describes version 1 of the Activity Monitor Unit (AMU) architecture, AMUv1, an optional non-invasive component. It contains the following sections:

- About the Activity Monitors Extension on page D12-5430.
- Properties and behavior of the activity monitors on page D12-5431.
- AMU events and event numbers on page D12-5433.
D12.1 About the Activity Monitors Extension

The Activity Monitors Extension is an OPTIONAL extension to the Armv8.4 architecture.

The Activity Monitors Extension implements version 1 of the Activity Monitors architecture, AMUv1, and interfaces to the registers defined by AMUv1, the Activity Monitors registers.

Version 1 of the Activity Monitors architecture implements:

- A counter group of four architected 64-bit event counters. The events counted by the architected event counter are fixed and architecturally defined.

  **Note**

  The Activity Monitors architecture provides space for up to 16 architected event counters. Future versions of the Activity Monitors architecture may use this space to implement additional architected event counters.

- A counter group of up to 16 auxiliary 64-bit event counters. The event counted for each auxiliary event counter may be fixed or programmable, and whether it is fixed or programmable is IMPLEMENTATION DEFINED. When the event counted by an auxiliary event counter is fixed, this event is IMPLEMENTATION DEFINED.

- Controls for enabling and disabling counters.

- When the event counted by an auxiliary event counter is programmable, controls for assigning an event to the counter.

- Controls that determine whether the activity monitor counters continue to count while the PE is halted in Debug state.

The read-only registers AMCFGR and AMCGCR provide information about features supported by the Activity Monitors Extension, the number of counter groups implemented, the total number of counters implemented, the number of counters implemented within each group, and the size of the counters.

The Activity Monitors Extension provides:

- A mandatory System register interface to the Activity Monitors registers, for both AArch64 and AArch32 states.

  *Base system registers on page K17-11883* lists the AArch64 Activity Monitors registers, and *Base system registers on page K17-11908* lists the AArch32 Activity Monitors registers. *Table K17-3 on page K17-11839* shows the relationship between the AArch64 and the AArch32 Activity Monitors register.

- Controls that allow software to enable or disable access by software running at lower Exception levels to the Activity Monitors registers.

- When FEAT_AMUv1p1 is implemented, and the hypervisor is using AArch64, offset registers that support virtualization of the Activity Monitor event counters.

- An optional external interface providing read-only memory-mapped access to the Activity Monitors registers.

  *Alphabetical index of memory-mapped registers on page K17-11911* lists the Activity Monitors memory-mapped registers. For more information on the recommended external interface, see *Chapter I4 Recommended External Interface to the Activity Monitors.*
### D12.2 Properties and behavior of the activity monitors

#### D12.2.1 Basic characteristics of the activity monitor event counters

Every activity monitor event counter is a 64-bit wrapping counter. When an activity monitor event counter wraps, the counter overflows.

**Note**

The Activity Monitor architecture does not provide support for overflow status indication or interrupts.

The state of the authentication signals do not affect counting.

Any change in clock frequency, including when a WFI and WFE instruction stops the clock, can affect any counter.

If FEAT_AMUv1p1 is implemented, for the architected event counters 0, 2 and 3, and each auxiliary event counter configured to use an offset, there is an offset register which is used to virtualize the count on a read from EL1 or EL0. At EL2, EL3 or from the memory-mapped view, permitted accesses to the counters use the physical view without any offset. See [Virtualization](page:D12-5432)

#### D12.2.2 Counter configuration and controls

For each architected event counter AMEVCNTR0<n>, there is a corresponding event type register AMEVTYPEPER0<n> which provides information on the event counted by that counter. The event type registers AMEVTYPEPER0<n> are read-only.

For each auxiliary event counter AMEVCNTR1<n>, there is a corresponding event type register AMEVTYPEPER1<n> which provides information on the event counted by that counter. When the event counted by an auxiliary event counter is fixed, the corresponding event type register AMEVTYPEPER1<n> is read-only. When the event counted by an auxiliary event counter is programmable, the corresponding event type register AMEVTYPEPER1<n> is read/write.

For each counter group, there is a pair of separate controls to enable and disable the counters in that counter group. AMCNTENCLR0 and AMCNTENSET0 are used to disable and enable the architected event counters. AMCNTENCLR1 and AMCNTENSET1 are used to disable and enable the auxiliary event counters.

While the PE is halted in Debug state, AMCR.HDBG controls whether activity monitor counting is halted.

AMUSERENR.EN controls access from EL0 to the Activity Monitor Extension System registers. CPTR_EL2.TAM and HCPTR.TAM control access from EL0 and EL1 to the Activity Monitor Extension System registers. CPTR_EL3.TAM control access from EL0, EL1, and EL2 to the Activity Monitor Extension System registers.

**Note**

These controls obey the priority order described in [Prioritization of Synchronous exceptions taken to AArch64 state](page:D1-4650) and [Synchronous exception prioritization for exceptions taken to AArch32 state](page:G1-8933).

AMUSERENR.EN is configurable at EL1, EL2, and EL3. All other controls, as well as the value of the counters, are configurable only at the highest implemented Exception level.

If FEAT_AMUv1p1 is implemented, AMCG1IDR_EL0 defines which auxiliary counters are implemented, and if virtual offsets are enabled, indicates which of the implemented auxiliary counters have a virtual offset when read from EL0 and EL1.

If FEAT_AMUv1p1 is implemented, AMCR.CG1RZ controls whether the auxiliary event counters read as zero if they are accessed at an Exception level lower than the highest implemented Exception level.

#### D12.2.3 Power and reset domains

The power domain of the Activity Monitoring Unit is IMPLEMENTATION DEFINED and named the AMU domain.

The reset domain of the Activity Monitoring Unit is IMPLEMENTATION DEFINED and named the AMU reset.
The AMU power domain may be the Core power domain.

When an AMU reset of the AMU power domain occurs, the Activity Monitoring Unit is reset and the counters are reset to zero.

When the PE is not in reset, the Activity Monitoring Unit is available.

### D12.2.4 Accuracy and non-invasive behavior

The activity monitors are a non-invasive component which must provide broadly accurate and statistically useful count information.

The implementation of an architecturally required event might create a conflict between the requirement to be non-invasive and the requirement to present an accurate value of the count under normal operating conditions. An implementation might provide an IMPLEMENTATION DEFINED control that disables accurate count of the event to restore performance and document the impact on performance of accurate counting. The expectations for non-invasive behavior and the degree of inaccuracy of the activity monitors are otherwise as described for the Performance Monitors architecture.

--- Note ---

For information on the expectations for non-invasive behavior and the degree of inaccuracy of the Performance Monitors, see Non-invasive behavior on page D11-5248 and A reasonable degree of inaccuracy on page D11-5249.

### D12.2.5 Virtualization

**FEAT_AMUv1p1** supports virtualized access to the Activity Monitors event counters at EL1 and EL0.

The fields **HCR_EL2.AMVOFFEN** and **SCR_EL3.AMVOFFEN** enable and disable virtualization. When enabled, the architected event counters 0, 2 and 3 have counter offsets. Architected event counter 1 does not have an offset. The register **AMCG1IDR_EL0** indicates which of the implemented auxiliary event counters has implemented counter offsets. An implemented event counter that does not have a defined offset has an effective offset of zero. The offset registers can be accessed only at EL2 or EL3, and affect views of the event counters at EL1 and EL0 from the System register interfaces only.

The **AMEVCNTVOFF0<n>_EL2** registers hold the offsets for the implemented and enabled architected event counters.

The **AMEVCNTVOFF1<n>_EL2** registers hold the offsets for the implemented and enabled auxiliary event counters.
D12.3 AMU events and event numbers

The Activity Monitors architecture uses the event number space defined by the Performance Monitors architecture to identify events.

The Activity Monitors architecture defines additional events and adds them to the event number space defined by the Performance Monitors architecture for common events.

If the event is counting an IMPLEMENTATION DEFINED event, it must use an event number from the IMPLEMENTATION DEFINED event space.

When an implementation supports monitoring of an event that is assigned a common architectural or microarchitectural event number, Arm strongly recommends that it uses that number for the event.

When a common event is available to both the Performance Monitors architecture and the Activity Monitors architecture within one implementation, both architectures use the same event number.

D12.3.1 Architected event counters

Version 1 of the Activity Monitors architecture, AMUv1, requires four events to be counted by the architected activity monitor event counters.

The events required to be counted are:

0x0011, CPU_CYCLES, Processor frequency cycles
The counter increments on every cycle when the PE is not in WFI or WFE state. When the PE is in WFI or WFE state, this counter does not increment.
This event is counted by AMEVCNTR0<n>, where n is 0.

0x4004, CNT_CYCLES, Constant frequency cycles
The counter increments at a constant frequency when the PE is not in WFI or WFE state, equal to the rate of increment of the System counter, CNTPCT_EL0. When the PE is in WFI or WFE state, this counter does not increment.
This event is counted by AMEVCNTR0<n>, where n is 1.

0x0008, INST_RETIRED, Instructions retired
This event is defined identically to INST_RETIRED in the FEAT_PMUv3 architecture.
This event is counted by AMEVCNTR0<n>, where n is 2.

0x4005, STALL_BACKEND_MEM, Memory stall cycles
The counter counts cycles in which the PE is unable to dispatch instructions from the frontend to the backend of the PE due to a backend stall caused by a demand data miss in the last level of data or unified cache within the PE clock domain or, if Armv8.7 is implemented, a non-cacheable data access in progress.
If Armv8.7 is not implemented, it is IMPLEMENTATION DEFINED whether the counter counts backend stall cycles when a non-cacheable data access is in progress.
This event is counted by AMEVCNTR0<n>, where n is 3.

D12.3.2 Auxiliary event counters

Auxiliary event counters can count events defined by the Performance Monitors architecture and IMPLEMENTATION DEFINED events defined specifically for activity monitoring.

Implementations must not reuse an IMPLEMENTATION DEFINED event number for different hardware events across the Performance Monitors architecture and the Activity Monitors architecture.
Chapter D13
The Statistical Profiling Extension

This chapter describes the Statistical Profiling Extension. It contains the following sections:

- About the Statistical Profiling Extension on page D13-5436.
- Defining the sample population on page D13-5438.
- Controlling when an operation is sampled on page D13-5439.
- Enabling profiling on page D13-5442.
- Filtering sample records on page D13-5444.
- The profiling data on page D13-5446.
- The Profiling Buffer on page D13-5458.
- Profiling Buffer management on page D13-5463.
- Synchronization and Statistical Profiling on page D13-5468.
D13.1 About the Statistical Profiling Extension

When the Statistical Profiling Extension is implemented, the PE includes a Statistical Profiling Unit (SPU). When profiling is enabled, the SPU does the following:

1. Chooses an operation from a sample population, that can be restricted by Exception level, at a programmable interval that might have some random, or pseudorandom, perturbation.
2. Takes a trace of the sampled operation. This includes the PC, events, timings, and data addresses, related to the sampled operation. This is the profiling operation.
3. If defined, filters out potential sample records generated by the profiling operation by reference to any or all of the following:
   a. The type of operation.
   b. Events.
   c. Latency.
4. Creates a record that contains the traced information. Sample records that meet the criteria of the filter are written to and stored in a memory buffer. These sample records can be processed by software when the memory buffer is full.

D13.1.1 Non-invasive behavior

Statistical Profiling is a non-invasive debug operation:

• While profiling is enabled, the operation and performance of the processing element (PE) must not be significantly impacted between sampled operations, that is, other than for writing out sample records and processing Profiling Buffer management interrupts.
• The performance of the sampled operation and the performance of the PE in general must not be significantly impacted. The sample records are not written to memory until after the sampled operation has finished execution. However, this does not apply if the sample records are physical addresses for data access operations. In this case, the impact is IMPLEMENTATION DEFINED.
• The profiling operation to write sample records must not be excessively impactful on the performance of the sampled operation or the performance of the PE generally.

D13.1.2 PMU extensions

If the Statistical Profiling and Performance Monitors Extensions are implemented, then the following PMU events must be implemented:

• SAMPLE_POP.
• SAMPLE_FEED.
• SAMPLE_FILTRATE.
• SAMPLE_COLLISION.

Note These events are discoverable through a read of PMCEID0_EL0[35:32].

If FEAT_SPEv1p2 is implemented, the following PMU events must be implemented:

• SAMPLE_FEED_BR.
• SAMPLE_FEED_LD.
• SAMPLE_FEED_ST.
• SAMPLE_FEED_OP.
• SAMPLE_FEED_EVENT.
• SAMPLE_FEED_LAT.
D13.1.3 **Multithreaded implementations**

In a multithreaded implementation:
- Statistical Profiling is implemented per-thread.
- The sample interval counter counts only operations for the thread that is being profiled.
- Latency and other cycle counters count each cycle for the PE for which the thread was active and could issue an operation.

The architecture does not define features for inter-thread profiling and does not support sharing the Profiling Buffer between threads.

--- **Note** ---

An implementation is described as multithreaded when the lowest level of *affinity* consists of logical processors that are implemented using a multi-threading type approach. That is, the performance of processors at the lowest affinity level is very interdependent.
D13.2 Defining the sample population

All samples are taken from a population of operations. The population is dynamic rather than static. That is, if a program executes the same operation multiple times (for example, because of loops and subroutines) then that operation appears multiple times in the population.

The operations are an implementation defined choice between:

- Architecture instructions.
- IMPLEMENTATION DEFINED microarchitectural operations (micro-ops).

Architecture instruction means a single instruction that is defined by the Armv8 instruction set architecture in AArch64 state.

An architecture instruction might create one or more micro-ops at any point in the execution pipeline. The definition of a micro-op is implementation specific. An architecture instruction might create more than one micro-op for each instruction. A micro-op might also be removed or merged with another micro-op in the execution stream, so an architecture instruction might create no micro-ops for an instruction.

Any arbitrary translation of architecture instructions to an equivalent sequence of micro-ops is permitted. In some implementations, the relationship between architecture instructions and micro-ops might vary over time.

Note
Sampling from architecture instructions does not require that the instruction is architecturally executed.

D13.2.1 Operations that might be excluded from the sample population

It is implementation defined whether each of the following operations is part of the sample population:

- Operations on misspeculated paths.
- Operations (specifically micro-ops) that do not relate to any architecture instruction.
- Operations that generate non-architectural exceptions.

If the operation is not part of the sample population, the operation does not cause the sample interval counter to decrement, is not counted by the SAMPLE_POP event and therefore is never sampled.

If the operation is part of the sample population, the operation causes the sample interval counter to decrement, is counted by the SAMPLE_POP event, and might be sampled and counted by the SAMPLE_FEED event. However, it is implementation defined whether the sample record for such a sampled operation is captured in the Profiling Buffer. For more information, see Sample operation records for misspeculated and non-architectural operations on page D13-5454 and Non-architectural exceptions on page D13-5457.

If such a sample record is not captured into the Profiling Buffer, then no packets are output and the sample is not counted by the SAMPLE_FILTRATE event.

Note
If the owning Exception level passes this data to less privileged software for processing, it can set PMSFCR_EL1.FE to 1 and PMSEVFR_EL1[1] to 1 to prevent speculative instructions from being recorded in the Profiling Buffer.
D13.3 Controlling when an operation is sampled

The sample interval counter, PMSICR_EL1.COUNT controls when an operation is selected for sampling. In some implementations, a secondary sample interval counter, PMSICR_EL1.ECOUNT, is also used.

The following sections describe the operation of the sample interval counters.

Details of the random or pseudorandom number generator used when PMSIRR_EL1.RND is set to 1 are IMPLEMENTATION DEFINED. See Generating random numbers for sampling.

D13.3.1 Operation sampling

A sample operation is as follows:

1. A sampling interval is written to PMSICR_EL1.COUNT by software. The interval is measured in operations.
2. The sample interval counter is decremented by hardware for each operation when sampling is enabled.
3. When the sample interval counter reaches zero, then:
   a. If random perturbation is enabled, the PE continues to count for a random number of further operations while sampling is enabled.
   b. An operation is chosen for profiling. The choice of operation around the sampling point is implementation-specific, but does not introduce sampling bias.
4. The sample interval counter is reloaded and the process loops to step 2. It is IMPLEMENTATION DEFINED whether the sample interval counter is reloaded before step 3.a) or at step 3.b). That is, before or after counting the random number of further operations.
5. The chosen operation is marked as the sampled operation. The PE collects information about the sampled operation as it executes by a profiling operation.
6. The sample record is created when the sampled operation has finished execution.

D13.3.2 Generating random numbers for sampling

The random number generator is IMPLEMENTATION DEFINED. Implementations might use a pseudorandom number. The random number generator must be reset into a useable state. An implementation might include IMPLEMENTATION DEFINED registers to further configure the random number generator.

It is IMPLEMENTATION DEFINED whether the PE adds the random number to the sample interval counter prior to counting down the interval, or after the counter reaches zero and the counter has been reloaded.

D13.3.3 Initializing the sample interval counters

When the PE moves from a state where profiling is disabled to a state where profiling is enabled:

- If PMSICR_EL1 is nonzero, then sampling restarts from the current values in PMSICR_EL1.
- If PMSICR_EL1 is zero, then it is loaded with an initial value. The behavior depends on PMSIRR_EL1.RND and an IMPLEMENTATION DEFINED choice discoverable by a read of PMSIDR_EL1.ERnd.
  - If PMSIRR_EL1.RND is 0:
    - PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
    - PMSICR_EL1.COUNT[7:0] is set to 0x00.
  - If PMSIRR_EL1.RND is 1 and PMSIDR_EL1.ERnd is 0:
    - PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
    - PMSICR_EL1.COUNT[7:0] is set to a random or pseudorandom value in the range 0x00 to 0xFF.
  - If PMSIRR_EL1.RND is 1 and PMSIDR_EL1.ERnd is 1:
    - PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
    - PMSICR_EL1.COUNT[7:0] is set to 0x00.
D13.3.4 Behavior of the sample interval counter while profiling is enabled

While profiling is enabled, the counters control when an operation is selected for sampling. The behavior depends on PMSIRR_EL1.RND and an IMPLEMENTATION DEFINED choice discoverable in PMSIDR_EL1.ERnd.

If PMSIRR_EL1.RND is 0:

While nonzero, the sample interval counter decrements by 1 for each member of the sample population. When the counter reaches zero:

- A member of the sampling population is selected for sampling.
- The counter is set as follows:
  - PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
  - PMSICR_EL1.COUNT[7:0] is set to 0x00.

Note

Because the counter counts down to zero, when PMSIRR_EL1.RND is 0 the interval between operations being selected for sampling is (INTERVAL×256+1).

If PMSIRR_EL1.RND is 1 and PMSIDR_EL1.ERnd is 0

While nonzero, the sample interval counter decrements by 1 for each member of the sample population. When the counter reaches zero:

- A member of the sampling population is selected for sampling.
- The counter is set as follows:
  - PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
  - PMSICR_EL1.COUNT[7:0] is set to a random or pseudorandom value in the range 0x00 to 0xFF.

Note

When PMSIRR_EL1.RND is 1 and PMSIDR_EL1.ERnd is 0, the mean interval between operations being selected for sampling is (INTERVAL×256+128), if the random number generator is uniform.

If PMSIRR_EL1.RND is 1 and PMSIDR_EL1.ERnd is 1

While nonzero, the primary sample interval counter decrements by 1 for each member of the sample population. When the primary counter reaches zero:

- The primary sample interval counter is set as follows:
  - PMSICR_EL1.COUNT[31:8] is set to PMSIRR_EL1.INTERVAL.
  - PMSICR_EL1.COUNT[7:0] is set to 0x00.
- The secondary sample interval counter, PMSICR_EL1.ECOUNT, is set to a random or pseudorandom value in the range 0x00 to 0xFF.

While the secondary sample interval counter is nonzero, the secondary sample interval counter decrements by 1 for each member of the sample population. The primary sample interval counter also continues to decrement because it is also nonzero.

When the secondary sample interval counter reaches zero, an operation is selected for sampling.

Note

When PMSIRR_EL1.RND is set to 1 and PMSIDR_EL1.ERnd is 1, the mean interval between operations being selected for sampling is (INTERVAL×256+1), if the random number generator is uniform.
D13.3.5 Behavior of the sample interval counter while profiling is disabled

When profiling is disabled:
• No operations are selected for sampling.
• No sample records are collected.
• The sample interval counters retain their values and do not decrement.

D13.3.6 Where operations are sampled

The exact point in the sampled lifespan of operations at which operations are chosen for profiling is IMPLEMENTATION DEFINED.

Note
Arm recommends that the point at which operations are sampled is linked to the definition of the Performance Monitors Extension (PMU) STALL_FRONTEND and STALL_BACKEND events, so that sampling records information for STALL_BACKEND stalls.

D13.3.7 Sample collisions

The maximum number of sampled operations that a PE can support simultaneously is IMPLEMENTATION DEFINED. If the maximum number of simultaneous sampled operations has been reached at the point when a new operation must be sampled, the new sample is said to have collided with a previous sampled operation.

The PE records the fact that a sampled operation has collided with another sampled operation. Software can also count the number of collisions and gauge the impact of the collisions.

On a sample collision:
• The PMU event SAMPLE_COLLISION is generated.
• PMBSR_EL1.COLL is set to 1.
• The new operation is not sampled.

Following a context synchronization event an indirect write to PMBSR_EL1.COLL is guaranteed to be visible to instructions in program order after the sampled operation that collided. There is no guarantee of visibility without a context synchronization event. For more information, see Synchronization and Statistical Profiling on page D13-5468.

Note
This means that following a context synchronization event PMBSR_EL1.COLL will not change on entry to a state where profiling is disabled.
D13.4 Enabling profiling

Profiling is enabled when all of the following are true:

- The PE is in AArch64 state.
- PMBLIMITR_EL1.E is 1 and PMBSR_EL1.S is 0.
- The PE is executing at either the Profiling Buffer owning Exception level or any lower Exception level.
- The PE is executing in the Security state of the Profiling Buffer owning Exception level.
- The PE is in Non-debug state.
- PMSCR_EL1.{E1SPE, E0SPE} and PMSCR_EL2.{E2SPE, E0HSPE} enable profiling at the current Exception level.

Note

The owning Exception level is controlled by MDCR_EL3.NSPB and MDCR_EL2.E2PB. See *The owning Exception level* on page D13-5459.

PMSCR_EL1.{E1SPE, E0SPE} and PMSCR_EL2.{E2SPE, E0HSPE} enable sampling by Exception level:

- In a guest operating system or Secure state, PMSCR_EL1.E1SPE enables profiling at EL1 and PMSCR_EL1.E0SPE at EL0.
- In a hypervisor or host operating system, PMSCR_EL2.E2SPE enables profiling at EL2 and PMSCR_EL2.E0HSPE at EL0.
- Sampling is always disabled at EL3.

Table D13-1 on page D13-5442 defines the valid combinations of the Effective values of SCR_EL3.NS, SCR_EL3.EEL2, MDCR_EL3.NSPB, MDCR_EL2.E2PB, and HCR_EL2.TGE that define when sampling is enabled.

In Table D13-1 on page D13-5442:

D Disabled.

E2SPE Enabled if PMSCR_EL2.E2SPE == 1, disabled otherwise.

E1SPE Enabled if PMSCR_EL1.E1SPE == 1, disabled otherwise.

E0HSPE Enabled if PMSCR_EL2.E0HSPE == 1, disabled otherwise.

E0SPE Enabled if PMSCR_EL1.E0SPE == 1, disabled otherwise.

**Table D13-1 Enabling by Exception level and Security state (for all Exception levels using AArch64 state)**

<table>
<thead>
<tr>
<th>NS</th>
<th>NSPB</th>
<th>E2PB</th>
<th>EEL2</th>
<th>TGE</th>
<th>EL3</th>
<th>EL2</th>
<th>EL1</th>
<th>EL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0b0X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>0b1X</td>
<td>0b1X</td>
<td>X</td>
<td>0</td>
<td>D</td>
<td>D</td>
<td>E1SPE</td>
<td>E0SPE</td>
<td></td>
</tr>
<tr>
<td>0b1X</td>
<td>X</td>
<td>1</td>
<td>D</td>
<td>D</td>
<td>n/a</td>
<td>E0SPE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b00</td>
<td>X</td>
<td>0</td>
<td>D</td>
<td>E2SPE</td>
<td>E1SPE</td>
<td>E0SPE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b00</td>
<td>X</td>
<td>1</td>
<td>D</td>
<td>E2SPE</td>
<td>n/a</td>
<td>E0HSPE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This is described in the pseudocode function `StatisticalProfilingEnabled()`. 

<table>
<thead>
<tr>
<th>Controls</th>
<th>Sampling enabled at</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS</td>
<td>NSPB</td>
</tr>
<tr>
<td>0</td>
<td>0b1X</td>
</tr>
<tr>
<td>0b0X</td>
<td>X</td>
</tr>
<tr>
<td>0b1X</td>
<td>1</td>
</tr>
<tr>
<td>0b1X</td>
<td>1</td>
</tr>
<tr>
<td>0b00</td>
<td>1</td>
</tr>
<tr>
<td>0b00</td>
<td>1</td>
</tr>
</tbody>
</table>
D13.5 Filtering sample records

PMSFCR_EL1.FT enables filtering by operation type. When enabled PMSFCR_EL1.{ST, LD, B} define the collected types:

- ST enables collection of store sampled operations, including all atomic operations.
- LD enables collection of load sampled operations, including atomic operations that return a value to a register.
- B enables collection of branch sampled operations, including direct and indirect branches and exception returns.

--- Note ---
When micro-op sampling is implemented, filtering is based on the micro-op type.

Table D13-2 on page D13-5444 summarizes the controls for filtering by operation type. In this table:

Load Atomic: Refers to atomic operations which return a value to a general-purpose register. Other atomic operations are classed as Store.

D: Indicates that the operation is discarded.
C: Indicates that the operation is collected.
C/D: Indicates it is CONstrained Unpredictable whether the operation is collected or discarded.

<table>
<thead>
<tr>
<th>PMSFCR_EL1 field</th>
<th>Operation type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT</td>
<td>LD</td>
</tr>
<tr>
<td>0XX</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>C</td>
</tr>
<tr>
<td>1D</td>
<td>C</td>
</tr>
<tr>
<td>10</td>
<td>C</td>
</tr>
<tr>
<td>1C</td>
<td>C</td>
</tr>
<tr>
<td>1D</td>
<td>C</td>
</tr>
</tbody>
</table>

PMSFCR_EL1.FE enables filtering by a set of events that are defined by PMSEVFR_EL1. When enabled, only sampled operations with all the events in the filter set are recorded and written to the Profiling Buffer.

If FEAT_SPEv1p2 is implemented, PMSFCR_EL1.FnE enables filtering by a set of events that are defined by PMSNEVFR_EL1. When enabled, only sampled operations with all the events in the filter clear are recorded and written to the Profiling Buffer.

PMSFCR_EL1.FL enables filtering by total latency. PMSLATFR_EL1.MINLAT defines the minimum latency. When enabled, only sampled operations with a total latency greater than or equal to the minimum latency are recorded and written to the Profiling Buffer.

These controls combine together as a logical AND.
Example D13-1 Collection of sampled operations

If PMSFCR_EL1.FE is 1, PMSFCR_EL1.FnE is 0, PMSFCR_EL1.FT is 1, and PMSFCR_EL1.FL is 1, then only sampled operations that meet all of the following criteria are recorded and written to the Profiling Buffer:

- The sampled operation is one of the selected operation types.
- The operation has all of the events in the filter set.
- The total latency is equal to or greater than the minimum latency.

This is described in the pseudocode function SPEcollectRecord().

D13.5.1 Discard mode

FEAT_SPEv1p2 adds an operating mode, Discard mode, that allows all sampled operations to be discarded and not written to the Profiling Buffer. Discard mode is enabled when PMBLIMITR_EL1.FM is 0b10, and has all of the following effects:

- All profiling data is discarded after filtering.
- The PMBLIMITR_EL1.LIMIT and PMBPTR_EL1 fields are ignored. PMBPTR_EL1 does not increment for each sampled operation.
- The restrictions on setting PMBLIMITR_EL1.LIMIT and PMBPTR_EL1 do not apply, see Restrictions on the current write pointer on page D13-5458.
- Buffer management events are not generated.

Other profiling behaviors are unchanged, including:

- The discarding of profiling data logically occurs after SAMPLE_FILTRATE and other PMU events are counted.
- Sample collisions will still set PMBSR_EL1.COLL to 1.
The profiling data

Unless otherwise stated, all sample records that are generated by a profiling operation contain:

- A timestamp, if enabled. This is one of:
  - The physical counter, CNTPCT_EL0.
  - The offset physical counter, CNTPCT_EL0 - CNTPOFF_EL2. When any of the following are true, the Effective value of CNTPOFF_EL2 is 0 for all profiling purposes:
    - EL3 is using AArch32.
    - EL2 is not implemented.
    - FEAT_ECV is not implemented.
    - The Effective value of SCR_EL3.{NS,RW} is {1,0}.
    - CNTHCTL_EL2.ECV is 0.
    - SCR_EL3.ECVEn is 0.
  - The virtual counter, CNTVCT_EL0.

It is IMPLEMENTATION DEFINED how this timestamp relates to the sampled operation. It might be the time when the sampled operation was taken or any later time during the lifetime of the sampled operation, that is, up to the time when the sampled operation finishes execution.

If the Generic Timer system counter is disabled and timestamps are enabled, then it is IMPLEMENTATION DEFINED whether:

- The SPU behaves as if timestamps are disabled.
- The timestamp that is collected in the sample record is UNKNOWN.

Note: This behavior describes when CNTCR.EN is 0, the Generic Timer system counter is disabled. This behavior does not apply when the Generic Timer system counter is enabled but not accessible at the current Exception level.

- The context, if enabled, which is one or more of:
  - CONTEXTIDR_EL1.
  - CONTEXTIDR_EL2.
  - The Exception level.
  - The Security state.

- Information about whether the sampled operation generated an exception:
  - The target address for an exception generating operation is not collected.

- Information about whether the sampled operation was Architecturally executed.

If the sampled operation is Architecturally executed and does not generate an exception, the sample record also contains:

- The PC virtual address for the sampled operation.
- Information about whether the sampled operation is a branch, a load, a load atomic, a store, or other.
- Information about whether the sampled operation is conditional, conditional select, or not.
- The total latency, a cycle count from the start of the sampled operation up to the point where the operation has finished execution and is no longer capable of stalling any instruction that consumes its output.
- The issue latency, a cycle count from the start of the sampled operation up to the point when at least one part of the sampled operation starts executing. A sampled operation might be delayed, for example, because the input operands were not available.

If the sampled operation is not Architecturally executed or generates an exception, it is UNPREDICTABLE whether the record contains all or any of this information and the other information about the operation listed in this section and the following subsections. For information on exceptions being taken in sampled operations, see Exceptions on page D13-5456.
The architecture defines a set of additional data that is collected in the sample record for each sampled operation. This is described in the following subsections, and comprises:

- Events, which are required to be implemented consistently with PMU Events. For more information, see Chapter D14 Statistical Profiling Extension Sample Record Specification and Chapter D11 The Performance Monitors Extension.
- Cycle counters. Cycle count values as described in this architecture, which, for a particular implementation, are fixed with an IMPLEMENTATION DEFINED value, might be omitted from the sample record.
- Addresses.

In addition, the architecture permits IMPLEMENTATION DEFINED events, counters, and addresses to be collected.

**D13.6.1 Information collected for micro-ops**

Because architectural instructions might create zero, one, or more micro-ops, micro-ops might have different characteristics from the architectural instructions they are created from. The data collected for each micro-op is IMPLEMENTATION DEFINED. Implementations should collect the subset of data appropriate to the micro-op.

**Example D13-2 Sampling of micro-ops**

If an architectural load instruction is split into an address generation micro-op and a load micro-op, then when generating the sample record and filtering based on operation type:

- If the address generation micro-op is sampled, the sampled operation is treated as other.
- If the load micro-op is sampled, the sampled operation is treated as a load.

**D13.6.2 Additional information for each profiled branch or exception return**

For an Architecturally executed sampled branch or exception return operation, the profiling operation records:

- The sampled operation type as an unconditional branch or a conditional branch. Sampled exception returns are treated as unconditional branches by the Statistical Profiling Extension.
- If the branch is taken, the target virtual address of the branch. The target virtual address of the branch includes the Exception level and Security state of the target. The target virtual address includes the Exception level and Security state of the target. If the sampled operation is an illegal exception return, it is CONSTRAINED UNPREDICTABLE whether the context information recorded in the target virtual address is the actual target context, or the target context that is described by the SPSR.
- If the PE implements branch prediction, whether the branch was correctly predicted or mispredicted.
- Whether the branch was taken or not taken.
- Whether the branch was direct or indirect.
- If the branch is not taken, a target virtual address might be recorded. Software must treat this value if present as UNKNOWN.
- If the optional behavior in FEAT_SPEv1p2 is implemented, the target address of the most recently executed branch that was taken and retired in program order before the sampled operation.

**Note**

A sampled operation that generates an exception is not treated as a branch.

It is IMPLEMENTATION DEFINED whether an ISB instruction is treated as a branch to the next instruction.

**Previous branch target**

FEAT_SPEv1p2 adds an optional capability to record a packet for each event that provides the target address of the previous taken branch. PMSIDR_EL1.PBT describes whether this feature is implemented.

When implemented, the profiling operation records the target address of the most recent branch that was taken and retired in program order before the sampled operation.
It is IMPLEMENTATION DEFINED whether or not the profiling operation records the target address of the most recently taken branch instruction in the following cases:

- The sampled operation is not a sampled retired taken branch operation.
- The most recently taken branch instruction was a Context synchronization operation, exception-generating instruction, or exception return.
- No branch instruction has been retired, prior to the sampled operation, since the most recent Context synchronization operation or taken exception.

The profiling operation does not record the target address of the most recently taken branch instruction in the following cases:

- The most recently taken branch instruction was executed when profiling was disabled or prohibited.
- Either the most recently taken branch instruction or the sampled operation is still speculative.

### D13.6.3 Additional information for each profiled memory access operation

For an *Architecturally executed* sampled load, store, or atomic operation that does not generate an exception, the profiling operation records:

- The data virtual and, if enabled, physical addresses being accessed.
  - If the applicable Top Byte Ignore (TBI) bit is set to one, the virtual address includes any top-byte tag.
  - The physical address is the address the PE accesses in the physical address space, and so includes the Secure address space identifier.
- The sampled operation type, which includes:
  - Whether the sampled operation is a load, store, or atomic.
  - When FEAT_SPEv1p3 and FEAT_MTE are implemented, whether the sampled operation is a Tag load or store.
  - When FEAT_NV2 is implemented, whether the sampled operation is a System register access transformed into a memory access.
  - When FEAT_SPEv1p3 and FEAT_MOPS are implemented, whether the sampled operation is a Memory Copy or Memory Set load or store.
  - Whether the sampled operation is Load-Exclusive, Store-Exclusive or Load-acquire, Store-release.
  - Whether the sampled operation accesses the general-purpose or SIMD&FP registers.
- The translation latency. Cycle count from a virtual address being passed to the MMU for translation to the result of the translation being available.
- Whether the sampled operation accessed the Level 1 data cache and the result.
- Whether the sampled operation accessed the data TLB and the result.
- An optional record of whether the sampled operation accessed Last Level data cache and the result.
- An optional record of whether the sampled operation accessed another socket in a multi-socket system.
- An optional, IMPLEMENTATION DEFINED, indicator of the data source for a load. If the sampled operation makes multiple accesses, it is IMPLEMENTATION DEFINED whether this indicator combines information for all parts of the load or applies only for a chosen part of the load.
- If FEAT_SPEv1p1 is implemented, an optional indication that the sampled memory operation is non-optimal for the access size. For more information, see *Data Alignment Flag* on page D13-5451.

For each of the cache and another socket indicators, it is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether this information is present for store accesses. The Last level cache and another socket indicators are optional and might not be present.

For more information, see *Events packet* on page D14-5486.

---

**Note**

A store might be marked as not accessing a cache or another socket because it microarchitecturally finished before doing so. For example, the write was placed into a write buffer. This behavior is IMPLEMENTATION DEFINED and might change from time to time, and such events must be interpreted with care.

If the sampled load, store, or atomic operation performs multiple accesses, it is IMPLEMENTATION DEFINED whether the implementation chooses to profile all of the access or a chosen part of that access.
If the implementation chooses to profile a chosen part of the access:

- It is IMPLEMENTATION DEFINED how the PE chooses the part of the access. The choice does not introduce any systematic bias.

--- Note ---

For an example of inadvertent systematic bias, consider an implementation where a multiple-register load operation is split into multiple accesses. If the PE systematically chooses the first operation at the lower address for sampling translation latency and data source indicator, and the operation is executed in a loop with an incrementing address, then the first access has better spatial locality with preceding accesses than later accesses and is more likely to both:

- Hit in the TLB, giving a shorter translation latency.
- Return data from the Level 1 data cache.

In this case, or if the PE systematically chooses the last access at the higher address, then sampling would be biased.

- If the accesses are architecturally contiguous, it is further IMPLEMENTATION DEFINED whether the recorded data virtual address is the lowest virtual address that is accessed by the sampled operation or applies to the chosen part of the access.
- If the accesses are not architecturally contiguous, the recorded data virtual address applies for the chosen part of the access.
- It is IMPLEMENTATION DEFINED whether the events and total operation latency apply to the whole operation or the chosen part of the operation.
- The translation latency applies to the chosen part of the operation, and is the count of cycles for which the chosen part of the operation is waiting for the MMU to complete an address translation.

Arm recommends that if the implementation chooses to profile a chosen part of the access, then the recorded addresses, events, and total operation latency apply to the chosen access. That is, the PE behaves as if the chosen part of the access is the sampled operation.

If the sampled load, store, or atomic operation performs a single access, or the implementation chooses to profile all parts of a multiple access:

- If the accesses are architecturally contiguous, the recorded data virtual addresses is the lowest virtual address that is accessed by the sampled operation.
- If the accesses are not architecturally contiguous, the recorded data virtual addresses apply for the chosen part of the access.
- The events and total operation latency apply to the whole operation. For example, when recording whether the sampled operation accessed the Level 1 data cache, the PE records whether any part of the access accessed the Level 1 data cache, and the result, and the total operation latency applies from the issue of the operation to the completion of all parts of the operation.
- The translation latency is an IMPLEMENTATION DEFINED choice between:
  - The count of cycles for which at least one part of the operation is waiting for the MMU to complete an address translation, and no part of the operation is accessing memory.
  - The count of cycles for which at least one part of the operation is waiting for the MMU to complete an address translation.

The sampled data physical address packet is not output if any of the following are true:

- The PE does not translate the address, for example because it does not perform the access or the address translation generates a Translation fault.
- The sampled data virtual address packet is not output.
- Sampling of physical addresses is prohibited by System register controls.

If a sampled virtual address packet is not output, or the PE does not perform the access, then all of the following apply:

- It is IMPLEMENTATION DEFINED whether the Translation latency Counter packet for the load or store is either not recorded, or recorded with a value of zero. If the access does not occur and the address is not translated, the packet is not recorded.
- It is IMPLEMENTATION DEFINED whether the bits corresponding to the access in the Events packet are recorded or always zero. If the access does not occur, these bits are zero.
Additional effects when FEAT_MTE2 is implemented and Allocation Tags are used

If profiling is enabled, each Allocation Tag covers multiple locations in a Tag Granule. It is IMPLEMENTATION DEFINED whether the implementation treats each Allocation Tag access as an access to the data location addressed in the operation, or the whole Tag Granule. That is, whether the data virtual address associated with the sampled access or chosen part of the access is the address of the location being accessed, or the lowest address covered by the same Allocation Tag or Allocation Tags.

If the operation is an access to an Allocation Tag or multiple Allocation Tags, the sampled data physical address has two possibilities:

- The address generated from translating the sampled data virtual address.
- The address generated from translating the lowest address covered by the same Allocation Tag or Allocation Tags.

If the two possibilities have different values, it is IMPLEMENTATION DEFINED which one is used as the sampled data physical address. Otherwise, the sampled data physical address is the address generated from translating the sampled data virtual address.

If FEAT_MTE and FEAT_SPEv1p3 are implemented, Tag load and store instructions are sampled using a dedicated Operation Type packet subclass, (Tags load/store). See Operation Type packet payload (load/store) on page D14-5493.

--- Note

Operations that store both Tags and data are recorded as store operations targeting the general-purpose registers or targeting an unspecified registers subclass.

---

For more information, see Chapter D9 The Memory Tagging Extension.

Additional effects when FEAT_NV2 is implemented

When the sampled operation is a System register access transformed into a memory access by the mechanism described in Enhanced support for nested virtualization on page D8-5175, the operation is recorded as an MRS or MSR operation at EL1 transformed into a load/store. If Statistical Profiling is disabled at EL2, the virtual address for the memory access is not recorded.

Additional effects when FEAT_MOPS is implemented

Memory Copy and Memory Set instructions are sampled as retired if they have started processing. This is true even if the execution is not completed at the time of sampling, for example, due to an interrupt.

Memory Copy operations CPY and CPYF are sampled using Operation Type packet subclass (Memory Copy load/store), and Memory Set operations SET and SETG are sampled using Operation Type packet subclass (Memory Set load/store).

If a sampled Memory Copy operation performs both a load and a store, then the following are true:

- The sampled data virtual address is:
  - The corresponding one of load and store addresses if the filter controls are set to sample only one of load and store operations, respectively.
  - An unbiased sampling of one of load and store addresses if the filter controls are set to sample both load and store operations.

- The value of the LDST bit in the Operation Type packet is 0 if the source address is sampled, and 1 if the destination address is sampled.
Additional effects when FEAT_LS64, FEAT_LS64_V, and FEAT_LS64_ACCDATA are implemented

FEAT_LS64, FEAT_LS64_V, and FEAT_LS64_ACCDATA add the LD64B, ST64B, ST64BV, and ST64BV0 instructions. For SPE:

- LD64B is treated as a load.
- ST64B, ST64BV, and ST64BV0 are treated as stores. ST64BV and ST64BV0 are store with return instructions and might have performance properties similar to a Device memory load. Sampled ST64BV and ST64BV0 instructions might have the following additional behaviors normally associated with sampled loads:
  - A Data Source packet is generated. This is optional for these instructions. For example, if the Data Source packet can only describe Normal memory sources, then they are not relevant for these instructions.
  - The total latency counter for the sampled operation preferably includes cycles to the point where the return value is returned to the PE.

**Note**

These behaviors are optional. The information in these packets might not be available or relevant for all implementations.

Data Alignment Flag

If FEAT_SPEv1p1 is implemented Events packet.E[11] is set to 1 for a sampled memory operation if the address alignment is non-optimal for the access size.

Address alignment is defined as *non-optimal* if that access incurs an additional performance penalty only because of the address alignment, and is unrelated to whether the access is architecturally misaligned for the access size.

**Example D13-3 Data Alignment Flag operation**

- A 32-bit word access that is not word aligned is architecturally misaligned, but (if Alignment faults are disabled) might not incur an additional penalty because of this alignment unless the word also happens to span a cache-line boundary.
- A contiguous load operation that loads a vector that is the length of two cache lines is optimally aligned if it has cache-line alignment, even though the operation makes two cache line accesses.
- A non-contiguous SVE load operation that makes a sequence of access is optimal only if all of the access are optimal.

The definition of non-optimal is IMPLEMENTATION DEFINED and support for the Alignment Flag is OPTIONAL.

**D13.6.4 Additional information for each profiled conditional instruction**

For an *Architecturally executed* sampled conditional operation that finishes execution, the profiling operation records:

- That the sampled operation was conditional.
- Whether the condition passed or failed.

A conditional compare operation is treated as a conditional operation.

It is IMPLEMENTATION DEFINED which conditional select operations (both integer and floating-point), including general-purpose, SIMD&FP, and SVE operations, are treated as conditional:

- Conditional select.
- Conditional select increment.
- Conditional select negate.
- Conditional select invert operations.
Predicated SVE operations are not conditional operations, as the conditionality of the operation is controlled by a predicate.

Micro-ops might have different characteristics from the architectural instructions they are created from. See Information collected for micro-ops on page D13-5447 for more information.

Note

If the implementation samples architectural instructions, for some alias instructions, the alias specifies the opposite condition to the machine instruction that they alias. For example, CSET and CSETM. The event as captured by SPE always captures the result of the machine instruction, not the alias.

For conditional branches, see Additional information for each profiled branch or exception return on page D13-5447.

D13.6.5 Additional information for each profiled Scalable Vector Extension operation

When FEAT_SPEv1p1 and The Scalable Vector Extension (SVE) are implemented, SVE operations are sampled as described in this section.

In this section the following terms are used:

Maximum implemented vector length

Means the implemented width of the vector registers. This value is IMPLEMENTATION DEFINED.

Accessible vector length

Means the accessible width of the SVE vector registers at the current Exception level, as constrained by the ZCR_EL1, ZCR_EL2 or ZCR_EL3 System registers. The Accessible vector length is always less-than-or-equal-to the Maximum implemented vector length.

Sampled SVE operation

Means an instruction or micro-operation defined by the Arm Architecture Reference Manual Supplement: The Scalable Vector Extension (SVE), for Armv8-A and sampled by the SPU that has a vector or a predicate as an input or output. This includes instructions with scalar outputs, but excludes the Non-SIMD SVE instructions.

If an implementation samples micro-operations, then it is IMPLEMENTATION DEFINED, and might vary between operation types, whether an operation for which all the following are true is treated as a Sampled SVE operation or the equivalent Advanced SIMD operation:

• The Accessible vector length is 128 bits.
• The operation is unpredicated, and does not have a predicate register as an input or output.
• The operation has an equivalent Advanced SIMD operation.

This includes SVE load and store operations where an equivalent Advanced SIMD operation is defined.

Sampled operation vector

Means the portion of the accessible vector operated on by the Sampled SVE operation.

Effective vector length

Is the length of the Sampled operation vector. The Effective vector length is always less-than-or-equal-to the Accessible vector length.

Note

The Accessible vector length is always quantized into multiples of 128-bits. However, the Sampled operation vector can be any size down to the element size of the operation.
Sampled predicated SVE operation

Means a Sampled SVE operation that is one of:

- An SVE operation that writes to a vector destination register under a Governing predicate using either zeroing or merging predication.
- A predicated store of a vector register or registers.

For an implementation that samples micro-operations, an SVE instruction might be split up into one or more micro-operations, some of which are predicated and some of which are not predicated.

Note

Sampled predicated SVE operation excludes operations that do not write a vector register, or do so but not using zeroing or merging predication, and applies to machine instructions rather than aliases. For example, the following instructions are not predicated SVE instructions under this definition:

- CNTP, LASTA, and PTRUE do not write to vector registers.
- FADDV, and SMAXV write scalar values to SIMD&FP registers.
- COMPACT and SEL (vectors) write to vector registers, and have a predicate operand, but do not use that predicate as a Governing predicate for zeroing or merging predication.
- MOV (vector, predicated) appears to be a predicated SVE instruction because it specifies merging predication through the <PG>/M operand, but it is actually an alias for the SEL (vectors) instruction.

If an implementation samples micro-operations, it is IMPLEMENTATION DEFINED whether individual elements, or groups of elements, are treated as single micro-operations.

The division of instructions into micro-operations must be fixed prior to sampling to guarantee consistently accurate statistical sampling.

Example D13-4 Vector length

For example, to support a vector length of 1024 bits, an implementation might split all instructions into four micro-operations on 256-bit vector paths. The implementation must, however, implement 1024-bit wide vector registers.

This behavior might vary based on operation type. For example, an implementation that has a full-width data-path for most operations might choose to break certain complex operations, such as non-contiguous load or stores, into shorter vectors.

Example D13-5 Accessible vector length less-than the Maximum implemented

To support an Accessible vector length less-than the Maximum implemented vector length, an implementation might choose to do all operations at the Maximum implemented vector length and discard the results above the Accessible vector length. Discarded results, arising from difference between Maximum implemented vector length and Accessible vector length, do not form part of the sampled operation and the Effective vector length must not include any discarded portions of the vector.

Results discarded because of predication are part of the sampled operation.

For a sampled SVE cache prefetch operation:

- The profiling operation captures an IMPLEMENTATION DEFINED subset of the information captured for an SVE load instruction.
- The profiling operation treats the operation type as Other when generating the sample records and filtering based on operation.
• It is IMPLEMENTATION DEFINED whether the operation is treated as a Sampled SVE operation:
  — If treated as a Sampled SVE operation, the Operation Type packet payload format is the Operation Type packet on page D14-5491.
  — If not treated as a Sampled SVE operation, the Operation Type packet format is the Operation Type packet payload (Other) on page D14-5491.

For a Sampled SVE operation, the Operation Type packet is one of:
• The SVE operation format.
• The SVE load or store format.

For a Sampled SVE operation, the Operation Type packet.EVL field records an upper bound on the Effective vector length. The value recorded in the Operation Type packet.EVL field is the Effective vector length rounded up to a power-of-two value.

For a Sampled SVE operation that is a Sampled predicated SVE operation;
• Operation Type packet.PRED, Predicated SVE operation, is set to 1.
• If any elements in the Sampled operation vector are Inactive elements, then Events packet.E[17], Partial predicate, is set to 1.
• If all elements in the Sampled operation vector are Inactive elements, then Events packet.E[18], Empty predicate, is set to 1 and Events packet.E[17] (Partial predicate) is set to 1.
• If all elements in the Sampled operation vector are Active elements then Events packet.E[18:17] is set to 0b00.

For a Sampled SVE operation that is not a Sampled predicated SVE operation:
• Operation Type packet.PRED, Predicated SVE operation, is set to 0.
• Events packet.E[18:17] is set to 0b00.

For a sampled non-contiguous SVE load or store operation that makes multiple memory accesses, the sampled data virtual address is the address accessed by a random one of the load or store operations chosen from the Sampled operation vector. If the chosen load or store operation is for an Inactive element, the data virtual address packet is not output.

For more information on memory access operations, see Additional information for each profiled memory access operation on page D13-5448.

For a sampled contiguous SVE load or store operation that makes multiple memory accesses, the sampled data virtual address is an IMPLEMENTATION DEFINED choice of:
• The address accessed for the lowest element in the Sampled operation vector.
• The address used for the access containing the lowest Active element in the Sampled operation vector.

If the corresponding element is an Inactive element, it is IMPLEMENTATION DEFINED whether the data virtual address packet is output.

**D13.6.6 Sample operation records for misspeculated and non-architectural operations**

It is IMPLEMENTATION DEFINED whether each of the following operations is part of the sample population:
• Operations on misspeculated paths.
• Operations that do not relate to any architecture instruction.

If the operation is part of the sample population, it is further IMPLEMENTATION DEFINED whether the sample record for the sampled operation is captured in the Profiling Buffer. For more information, see Operations that might be excluded from the sample population on page D13-5438.

If such an operation is part of the sample population and the sample record is captured in the Profiling Buffer, then some information for the operation might not be present. However, the Events packet and either the End packet or the Timestamp packet is always output. Neither event 0 (generated exception) nor event 1 (architecturally retired) will be set in the Events packet.

The record must not contain information that cannot be accessed by privileged software of the owning Exception level.
D13.6.7 Additional information for other operations

For cache maintenance operations by virtual address, cache prefetch, other than SVE cache prefetch, or address translation instructions, the profiling operation:

- Captures an IMPLEMENTATION DEFINED subset of the information captured for a load instruction.
- Treats the operation type as other when generating the sample record and filtering based on operation type.

See Filtering sample records on page D13-5444, Operation Type packet and Additional information for each profiled Scalable Vector Extension operation on page D13-5452.

D13.6.8 Controlling the data that is collected

Certain data in sample records is collected only if permitted by one or both of EL1 and EL2. This is to restrict exposure of data to a lower Exception level or to Non-secure state.

CONTEXTIDR_EL1 is collected only if PMSCR_EL1.CX is set to 1, the PE is executing at EL1 or EL0 and any of the following are true when an operation is sampled:

- EL2 is not implemented.
- FEAT_SEL2 is implemented and EL2 is disabled for the current Security state.
- The Effective value of HCR_EL2.TGE is 0.

CONTEXTIDR_EL2 is collected only if the Effective value of PMSCR_EL2.CX is 1 and EL2 is implemented and enabled for the current Security state.

This is described in the pseudocode functions CollectContextIDR1() and CollectContextIDR2().

Timestamps are collected only if one of the following is true:

- PMSCR_EL1.TS is set to 1 and the Profiling Buffer is owned by EL1.
- PMSCR_EL2.TS is set to 1 and the Profiling Buffer is owned by EL2.

The timestamp is a choice between:

- Physical time, which is defined by the physical count value returned by PhysicalCountInt().
- If FEAT_ECV is implemented, offset physical time, which is defined as the value of (PhysicalCountInt() - CNTPOFF_EL2). However, the physical offset is treated as zero if FEAT_ECV is disabled.
- Virtual time, which is defined as the value of (PhysicalCountInt() - CNTVOFF_EL2). However, the virtual offset is treated as zero if a read of CNTVCT_EL0 at the current Exception level would treat the virtual offset as zero.

Table D13-3 on page D13-5456 summarizes the choice of value for the Timestamp packet when FEAT_ECV is implemented and StatisticalProfilingEnabled() is TRUE. In Table D13-3 on page D13-5456:

**Owning EL** This is the Exception level that owns the Profiling Buffer. This is returned by the function ProfilingBufferOwner(). If EL2 is disabled in the current Security state, this is always EL1.

**EL2 enabled** This is TRUE when EL2 is enabled in the current Security state. When EL2 is disabled in the current Security state, this is FALSE.

**Virtual offset** If any of the following are true, the virtual offset is zero, otherwise the virtual offset is the value of CNTVOFF_EL2:

- The Effective value of HCR_EL2.E2H is 1, and the sampled operation was executed at EL2.
- The Effective values of HCR_EL2.{E2H, TGE} is {1, 1}, and the sampled operation was executed at EL0.

**Physical offset** If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2:

- EL2 is not implemented.
- FEAT_ECV is not implemented.
- CNTHCTL_EL2.ECV is 0.
SCR_EL3.ECVEn is 0.

### Table D13-3 Recorded timestamp when FEAT_ECV is implemented

<table>
<thead>
<tr>
<th>EL2 enabled</th>
<th>Owning EL</th>
<th>PMSCR_EL2.PA</th>
<th>PMSCR_EL1.PA</th>
<th>Recorded timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>EL1</td>
<td>xx</td>
<td>x</td>
<td>0b00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PhysicalCountInt() - virtual offset</td>
</tr>
<tr>
<td>TRUE</td>
<td>EL1</td>
<td>xx</td>
<td>0b01</td>
<td>0b11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PhysicalCountInt() - physical offset</td>
</tr>
<tr>
<td></td>
<td>EL2</td>
<td>xx</td>
<td>0b00</td>
<td>0b01</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PhysicalCountInt() - virtual offset</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PhysicalCountInt() - physical offset</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PhysicalCountInt() - physical offset</td>
</tr>
</tbody>
</table>

If EL2 is not implemented, see the register descriptions of PMSCR_EL1.PCT and PMSCR_EL2.PCT for details of their behavior. This behavior is described by the pseudocode function `CollectTimeStamp()`.

Physical data addresses are collected only if one of the following is true:

- PMSCR_EL1.PA is set to 1 and the Profiling Buffer is owned by Secure EL1, and Secure EL2 is disabled or not implemented.
- PMSCR_EL2.PA is set to 1 and the Profiling Buffer is owned by Secure or Non-secure EL2.
- PMSCR_EL1.PA is set to 1 and PMSCR_EL2.PA is set to 1 and either the Profiling Buffer is owned by Non-secure EL1, or the Profiling Buffer is owned by Secure EL1 and Secure EL2 is implemented and enabled.

If EL2 is not implemented or is disabled for the current Security state, the PE behaves as if PMSCR_EL2.PA is set to 1, other than for a direct read of the register.

Physical data address collection is described by the pseudocode function `CollectPhysicalAddress()`.

Enabling collection of the physical data addresses has an IMPLEMENTATION DEFINED impact on the sampled operation.

### D13.6.9 Exceptions

All sample records written to the Profiling Buffer contain the Events packet and either the End packet or the Timestamp packet.

If the sampled operation generates an exception condition, it is UNPREDICTABLE whether the sample record contains any other information. This includes operations that generate faults or other exception conditions but do not generate exceptions. For example:

- An instruction on a misspeculated path.
- A load operation that is part of a Non-fault load instruction or is not the First active element of a First-fault load instruction that generates an MMU fault or watchpoint.
- An address translation operation or prefetch instruction that generates an MMU fault.

Where a sampled operation generates an exception and the type of exception means that a particular item is not computed by the sampled operation, that information is not collected by the profiling operation. For more information, see Synchronization and Statistical Profiling on page D13-5468.
Example D13-6 Translation Faults

If a sampled operation generates a Translation Fault, the physical address for the sampled operation was not generated by the MMU and cannot be recorded.

Non-architectural exceptions

It is IMPLEMENTATION DEFINED whether operations that generate non-architectural exceptions are part of the sample population. If such an operation is part of the sample population, it is further IMPLEMENTATION DEFINED whether the sample record for a sampled operation that generates a non-architectural exception is captured in the Profiling Buffer. For more information, see Operations that might be excluded from the sample population on page D13-5438.

If such an operation is part of the sample population and the sample record is captured in the Profiling Buffer, then the sample might record handling of the non-architectural exception. If the sample record does not record handling of the non-architectural exception, then the sampled operation is not Architecturally executed because of the non-architectural exception and it is recorded using E[1] == 0 (operation is not architecturally executed) in the Events packet. Bit E[0] (operation generated an exception) might be used to indicate the operation did not complete because of the non-architectural exception.
The Statistical Profiling Extension

D13.7 The Profiling Buffer

The profile data is collected in a memory Profiling Buffer. The Profiling Buffer is defined by:

• \( \text{PMBPTR\_EL1} \), the current write pointer.
• \( \text{PMBLIMITR\_EL1} \), the write limit pointer.

The Profiling Buffer starts at the current write pointer and extends to the current limit pointer minus one. The write limit pointer must be aligned to the smallest implemented translation granule size. The alignment of the current write pointer is IMPLEMENTATION DEFINED.

\( \text{PMBLIMITR\_EL1} \) and \( \text{PMBPTR\_EL1} \) are virtual addresses in the stage 1 translation regime of the owning Exception level. This is called the owning translation regime.

Note

The translation of virtual addresses to physical addresses is identical to that for any other virtual address in the owning Exception level. For example, \( \text{PMBPTR\_EL1}[63:56] \) are ignored by address translation if the respective TBI bit is set to 1.

D13.7.1 Restrictions on the current write pointer

This section describes the software rules on setting the current write pointer, \( \text{PMBPTR\_EL1} \). If these rules are not followed, the value returned for a direct read of \( \text{PMBPTR\_EL1} \) is UNKNOWN, the behavior is UNPREDICTABLE, and the PE might do any of the following at any point after profiling is enabled:

• Write sample records to any writeable address in memory that is writable at the owning Exception level.
• Generate a Profiling Buffer management event, with or without indicating data loss, for one of the following reasons:
  — The Profiling Buffer is full.
  — Any MMU Fault.

When profiling becomes enabled, all the following must be true:

• The current write pointer must be at least one sample record below the write limit pointer. That is:
  \[
  \text{UInt}(\text{PMBPTR\_EL1.PTR}) \leq \text{UInt}(\text{PMBLIMITR\_EL1.LIMIT : Zeros(12)}) - 2\text{PMSIDR\_EL1.MaxSize}.
  \]
• \( \text{PMBPTR\_EL1.PTR}[63:56] \) must equal \( \text{PMBLIMITR\_EL1.LIMIT}[63:56] \).

When the Profiling Buffer is first configured, \( \text{PMBPTR\_EL1.PTR} \) must be aligned to \( \text{PMBIDR\_EL1.Align} \). That is, if \( \text{PMBIDR\_EL1.Align} \) is nonzero, \( \text{PMBPTR\_EL1.PTR} [\text{UInt}(\text{PMBIDR\_EL1.Align})-1:0] \) must be all zeros.

However, the current write pointer can usually be restored to the saved write pointer value it had when profiling was disabled, providing a PSB CSYNC and a context synchronization event were executed before reading \( \text{PMBPTR\_EL1} \):

• If no Profiling Buffer management event was signaled then profiling can be restarted from the saved write pointer. In this case, the saved write pointer points within one sample record of the write limit pointer.
• If a Profiling Buffer management event was signaled then:
  — If \( \text{PMBSR\_EL1.S} \) is restored to 1, then profiling is not being enabled, and there are no constraints on the value written to \( \text{PMBPTR\_EL1} \).
  — If \( \text{PMBSR\_EL1.S} \) is restored to 0, and the Profiling Buffer management event was caused by an MMU fault, profiling can be restarted from the saved write pointer; if \( \text{PMBSR\_EL1.\{EA, DL\}} \) did not also indicate an External abort or data loss, and the saved write pointer is at least one sample record below the write limit pointer.

Note

If a signaled MMU fault has not been corrected, the SPU generates a new MMU fault Profiling Buffer management event when it next tries to write a sample record.

— If \( \text{PMBSR\_EL1.S} \) is restored to 0, and the Profiling Buffer management event was caused by a buffer full event, the Profiling Buffer can be extended and profiling restarted from the saved write pointer; if \( \text{PMBSR\_EL1.\{EA, DL\}} \) did not also indicate an External abort or data loss and the saved write pointer is at least one sample record below the extended write limit pointer.
The current write pointer must not be restored from the saved write pointer following a Profiling Buffer management event if PMBSR_EL1.DL was set to 1.

The saved write pointer might not be aligned to 2^PMBDIR_EL1.Align and might point to within one sample record of the write limit pointer.

For more information, see Synchronization and Statistical Profiling on page D13-5468.

### D13.7.2 The owning Exception level

The owning Exception level is:

- **Non-secure EL1**, if all of the following are true:
  - Either EL3 is not implemented and the PE is executing in Non-Secure state, or MDCR_EL3.NSPB is either 0b10 or 0b11.
  - Either EL2 is not implemented, or MDCR_EL2.E2PB is either 0b10 or 0b11.

- **Non-secure EL2**, if all of the following are true:
  - EL2 is implemented.
  - Either EL3 is not implemented and the PE is executing in Non-secure state, or MDCR_EL3.NSPB is either 0b10 or 0b11.
  - MDCR_EL2.E2PB is 0b00.

- **Secure EL1**, if all of the following are true:
  - Either EL3 is not implemented and the PE is executing in Secure state, or MDCR_EL3.NSPB is either 0b00 or 0b01.
  - Either Secure EL2 is not implemented or is disabled, or MDCR_EL2.E2PB is either 0b10 or 0b11.

- **Secure EL2**, if all of the following are true:
  - Secure EL2 is implemented and enabled.
  - Either EL3 is not implemented and the PE is executing in Secure state, or MDCR_EL3.NSPB is either 0b00 or 0b01.
  - MDCR_EL2.E2PB is 0b00.

#### When the owning Exception level is Non-secure EL1

The Profiling Buffer addresses are in the Non-secure EL1&0 translation regime using the current ASID from TTBRx_EL1. This is a two-stage translation using the current VMID if EL2 is implemented and HCR_EL2.VM is 1.

If EL3 is implemented, then profiling is disabled in Secure state.

If EL2 is implemented, then profiling is disabled at EL2 and at Non-secure EL0 when HCR_EL2.TGE is 1.

#### When the owning Exception level is Non-secure EL2

The Profiling Buffer addresses are in the Non-secure EL2 translation regime. If HCR_EL2.E2H is 1, this is an EL2&0 translation regime using the current EL2&0 translation regime ASID from TTBRx_EL2.

If EL3 is implemented, then profiling is disabled in Secure state.

---

**Note**

If either HCR_EL2.E2H is 0 or HCR_EL2.TGE is 0, and the PE is executing at EL1 or EL0, the EL2 translation regime is not the current stage 1 translation regime because the current stage 1 translation regime is EL1&0.

---

#### When the owning Exception level is Secure EL1

The Profiling Buffer addresses are in the Secure EL1&0 translation regime using the current ASID from TTBRx_EL1. This is a two-stage translation using the current VMID if Secure EL2 is implemented and enabled and HCR_EL2.VM is 1.

If EL3 is implemented, then profiling is disabled in Non-secure state.
If Secure EL2 is implemented and enabled, then profiling is disabled at EL2 and at Secure EL0 when HCR_EL2.TGE is 1.

When the owning Exception level is Secure EL2

The Profiling Buffer addresses are in the Secure EL2 translation regime. If HCR_EL2.E2H is 1, this is an EL2&0 translation regime using the current EL2&0 translation regime ASID from TTBRx_EL2.

If EL3 is implemented, then profiling is disabled in Non-secure state.

Summary of the owning translation regime

Profiling is disabled if any of the following are true:

- The owning Exception level is using AArch32 state.
- PMBLIMITR_EL1.E is 0.

Table D13-4 on page D13-5460 summarizes the owning translation regime.

<table>
<thead>
<tr>
<th>PMBLIMITR_EL1</th>
<th>SCR_EL3</th>
<th>MDCR_EL3</th>
<th>MDCR_EL2</th>
<th>Owning translation regime</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>NS</td>
<td>EEL2</td>
<td>NSPB</td>
<td>E2PB</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>0b1x</td>
<td>0b1x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0b0x</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0b0x</td>
<td>X</td>
<td>Secure EL1&amp;0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0b0x</td>
<td>0b1x</td>
<td>Secure EL1&amp;0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0b00</td>
<td>Secure EL2 or EL2&amp;0*</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>0b1x</td>
<td>X</td>
<td>Disabled</td>
</tr>
</tbody>
</table>

a. Depending on the values of HCR_EL2.{E2H,TGE}.

D13.7.3 Memory access types and coherency

 Writes to any Device memory type by the SPU occur once.

The memory type and attributes that are used for a write by the SPU to the Profiling Buffer is taken from the translation table entries for the virtual address being written to. That is:

- The writes are treated as coming from an observer that is coherent with all observers in the Shareability domain that is defined by the translation tables.
- There is no requirement to manage coherency for observers in the same Shareability domain but coherency for other observers in the system might require explicit management.

For more information, see Synchronization and Statistical Profiling on page D13-5468.

If FEAT_MTE2 is implemented, a PE will generate a Tag Unchecked access for each access to the Profiling Buffer as part of writing a sample record.

For more information on FEAT_MTE2, see Chapter D9 The Memory Tagging Extension.
Writes to the Profiling Buffer are made as privileged writes within the owning translation regime. However, the value of PSTATE.PAN is ignored for these writes and treated as zero, see Faults and watchpoints on page D13-5464.

This means that if FEAT_E0PD is implemented, the values of TCR_ELx.E0PDy, where ELx is the owning Exception level, do not apply to accesses to the Profiling Buffer made by the SPU.

D13.7.4 Memory access and crossing page boundaries

A memory access from the SPU that crosses a page boundary to a memory location that has a different memory type or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation performs one of the following behaviors:

- Each memory access generated by the SPU uses the memory type and Shareability attribute associated with its own address.
- The access generates an Alignment fault caused by the memory type:
  - If only the stage 1 translation generated the mismatch, or there is only one stage of translation in the owning translation regime, the resulting Buffer Management event is a stage 1 Data Abort.
  - If only the stage 2 translation generated the mismatch, the resulting Buffer Management event is a stage 2 Data Abort.
  - If both stages of translation generate the mismatch, the resulting Buffer Management event is either a stage 1 Data Abort or a stage 2 Data Abort.
- Some or all of the data is discarded. The write pointer is either updated by the amount of data written not including the discarded data or the amount of data written including the discarded data.

A memory access from the SPU to Device memory that crosses a boundary corresponding to the smallest translation granule size of the implementation causes CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation performs one of the following behaviors:

- All memory accesses generated by the SPU are performed as if the boundary has no effect on the memory accesses.
- All memory accesses generated by the SPU are performed as if the boundary has no effect on the memory accesses except that there is no guarantee of ordering between memory accesses.
- The access generates an Alignment fault caused by the memory type:
  - If only the stage 1 translation causes the boundary to be crossed, or there is only one stage of translation in the owning translation regime, the resulting Buffer Management event is a stage 1 Data Abort.
  - If only the stage 2 translation causes the boundary to be crossed, the resulting Buffer Management event is a stage 2 Data Abort.
  - If both stages of translation cause the boundary to be crossed, the resulting Buffer Management event is either a stage 1 Data Abort or a stage 2 Data Abort.
- Some or all of the data is discarded. The write pointer is either updated by the amount of data written not including the discarded data or the amount of data written including the discarded data.

--- Note ---

The boundary referred to is between two Device memory regions that are both:
- Of the size of the smallest implemented translation granule.
- Aligned to the size of the smallest implemented translation granule.

If PMSIDR_EL1.MaxSize indicates the same value as PMBIDR_EL1.Align, then records are a fixed power-of-two size and never cross a page boundary.

D13.7.5 Cache and TLB operations

TLB maintenance operations that affect the TLB of the PE also affect any TLB caching translations for the SPU of that PE.

Cache maintenance operations that affect the caches of the PE also affect data caching by the SPU of that PE.
This means that the completion of any cache or TLB maintenance instruction includes its completion on all SPUs for PEs that are affected by both the instruction and the DSB operation that is required to guarantee visibility of the maintenance instruction. See Completion and endpoint ordering on page B2-168.

D13.7.6 Effect on the exclusive monitors

If a Load-exclusive instruction or an operation between Load-exclusive and Store-exclusive instructions is sampled, and the sample record is written to an unrelated address, then to avoid a probe effect, Arm recommends that the Store-exclusive does not systematically fail on account of the sampled operation.

If a Store-exclusive instruction is sampled, and the sample record is written to an unrelated address, then the Store-exclusive must not systematically fail on account of the instruction having been sampled.
D13.8 Profiling Buffer management

A Profiling Buffer management event occurs:

- On a fault, see *Faults and watchpoints on page D13-5464.*
- On an External abort, see *External aborts on page D13-5466.*
- When the Profiling Buffer fills, see *Buffer full event on page D13-5464.*

--- Note ---

The Profiling Buffer management event behavior differs from that of the FEAT TRBE trace buffer management event.

---

On a Profiling Buffer management event:

- The service bit, PMBSR_EL1.S, is set to 1.
- The data loss bit, PMBSR_EL1.DL, is set as described in the event description.
- The Profiling Buffer management interrupt request signal, PMBIRQ, is asserted:
  - **PMBIRQ** is a level-sensitive interrupt request driven by PMBSR_EL1.S. This means that a direct write that sets PMBSR_EL1.S to 1 causes the interrupt to be asserted, and PMBIRQ remains asserted until software clears PMBSR_EL1.S to 0.
  - If a Generic Interrupt Controller (GIC) is implemented, PMBIRQ must be configured as a Private Peripheral Interrupt (PPI) in a multiprocessor system. PMBIRQ is signaled by the PE that implements the SPU.

--- Note ---

A standard PPI number is allocated by the Arm® Base System Architecture (BSA).

- Additional syndrome for the event is written to PMBSR_EL1.MSS. Unless otherwise stated in the event description, other PMBSR_EL1 fields are unchanged.

While PMBSR_EL1.S is set to 1:

- The buffer is disabled and profiling is disabled.
- All remaining buffered sample records are discarded.
- The values in PMBPTR_EL1 are retained and PMSICR_EL1 does not decrement.

Buffer full events and MMU fault Profiling Buffer management events are reported synchronously.

--- Note ---

*Reported synchronously* means that profiling is disabled before the SPU samples further operations. The interrupt exception resulting from asserting the Profiling Buffer interrupt request is an asynchronous exception.

---

It is IMPLEMENTATION DEFINED whether External aborts are reported to the SPU synchronously or asynchronously. If External aborts are reported as asynchronous:

- The External abort might not be received until after a first Profiling Buffer management event has set PMBSR_EL1.S to 1.
- Writes to the buffer might generate a second Profiling Buffer management event after the External abort has set PMBSR_EL1.S to 1.

The architecture does not require that a sample record is written sequentially by the SPU, only that:

- The SPU never writes past the PMBLIMITR_EL1 limit pointer.
- On a Profiling Buffer management interrupt, PMBSR_EL1.DL indicates whether PMBPTR_EL1 points to the first byte after the last complete sample record.
- On an MMU fault or synchronous External abort, PMBPTR_EL1 serves as a Fault Address Register.

--- Note ---

- This means that it must not be assumed that:
  - There is ever any valid data beyond the current PMBPTR_EL1 write pointer.
— The PE has not written a valid sample record between the current PMBPTR_EL1 write pointer and the PMBLIMITR_EL1 limit pointer.
— If PMBSR_EL1.DL is set to 1 on a Profiling Buffer management interrupt, that there is any valid data between the end of the last complete sample record and the current PMBPTR_EL1 write pointer.
— Any valid data has been written to the Profiling Buffer if an External abort is reported asynchronously to the SPU.

• The last complete sample record must end at most 2(PMSIDR_EL1.MaxSize) bytes below PMBPTR_EL1.

D13.8.1 Prioritization of Profiling Buffer management events

Where multiple synchronous Profiling Buffer management events occur on writing a sample record, the PE prioritizes them as follows (from highest to lowest priority):
1. Synchronous fault.
2. Synchronous External abort.
3. Buffer full event.

Asynchronous External aborts are not prioritized with respect to other events.

Note
Prioritization of Profiling Buffer management interrupt requests is managed by the interrupt controller. Profiling Buffer management events are prioritized internally by the PE.

D13.8.2 Buffer full event

If, after writing a sample record, there is not sufficient space in the Profiling Buffer for a sample record of the size indicated by PMSIDR_EL1.MaxSize, and PMBSR_EL1.S is 0, a Profiling Buffer management event is generated:
• PMBSR_EL1.EC is set to 0b000000, other buffer management event.
• The BSC field of PMBSR_EL1.MSS is set as follows:
  — PMBSR_EL1.BSC is set to 0b000001, buffer filled.
• PMBPTR_EL1 is set to the first byte after the last complete sample record. PMBSR_EL1.DL is unchanged.
• The other PMBSR_EL1 fields are unchanged.
• PMBSR_EL1.S is set to 1.

That is, the Profiling Buffer management event is generated when the PE writes past the write limit pointer minus 2(PMSIDR_EL1.MaxSize). The SPU never writes beyond the write limit pointer.

For more information, see Restrictions on the current write pointer on page D13-5458.

D13.8.3 Faults and watchpoints

Table D13-5 on page D13-5464 lists the faults that might be generated by a write to the Profiling Buffer by the SPU. Writes to the Profiling Buffer never generate watchpoints.

<table>
<thead>
<tr>
<th>Fault</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>The translation of a virtual address to a physical address might generate a Translation fault.</td>
</tr>
<tr>
<td>Address Size</td>
<td>The translation of a virtual address to a physical address might generate an Address Size fault.</td>
</tr>
<tr>
<td>Alignment</td>
<td>If PMBPTR_EL1 is not aligned to an IMPLEMENTATION DEFINED minimum alignment, the behavior is UNPREDICTABLE and a write to the Profiling Buffer by the SPU might generate an Alignment fault. For more information, see Restrictions on the current write pointer on page D13-5458.</td>
</tr>
</tbody>
</table>
If a write to the Profiling Buffer generates a fault and PMBSR_EL1.S is 0, then a Profiling Buffer management event is generated:

- PMBSR_EL1.S is set to 1.
- PMBSR_EL1.EC is set to one of:
  - 0b100100, stage 1 Data Abort on write to the Profiling Buffer.
  - 0b100101, stage 2 Data Abort on write to the Profiling Buffer.
- The FSC field of PMBSR_EL1.MSS is set as follows:
  - PMBSR_EL1.FSC is set to indicate the type of the fault.
- PMBPTR_EL1 is set to the address that generated the fault.
- If PMBPTR_EL1 is not the address of the first byte after the last complete sample record written by the SPU, then PMBSR_EL1.DL is set to 1. Otherwise, PMBSR_EL1.DL is unchanged.
- The other PMBSR_EL1 fields are unchanged.

If a write to the Profiling Buffer generates an External abort on a translation table walk or translation table update, it is IMPLEMENTATION DEFINED whether PMBSR_EL1.EA is set to 1 or unchanged.

**Note**

Each of these faults gives rise to a Profiling Buffer management interrupt, not an actual MMU fault exception. The ESR and FAR registers are unchanged.

For more information, see *MMU fault-checking sequence* on page D8-5186.

---

### Hardware management of dirty state and the Access flag by the Statistical Profiling Extension

If FEAT_SPEv1p3 is implemented, address translations performed by the SPU manage dirty state and the Access flag. Otherwise, it is IMPLEMENTATION DEFINED whether address translations performed by the SPU manage dirty state and the Access flag. This is discoverable by software using PMBIDR_EL1.F. See *Hardware management of the dirty state on page D8-5145* and *Hardware management of the Access flag on page D8-5144*.
If the SPU manages dirty state and hardware management of dirty state is enabled for the owning translation regime, then the SPU can speculatively update the Translation Table descriptor for any Page or Block in the Statistical Profiling buffer before writing data to it, if the write is otherwise permitted. This includes the case where a buffer management event means the SPU stops writing data before the page or block is written to. For more information, see *The Profiling Buffer* on page D13-5458.

**D13.8.4 External aborts**

When a write to the Profiling Buffer generates an External abort, including an External abort on a translation table walk or translation table update, the permitted IMPLEMENTATION DEFINED behaviors are:

- The External abort is ignored.
- The External abort generates an SError interrupt exception.
- If FEAT_SPEv1p3 is not implemented, the External abort is reported to the SPU. The SPU generates a Profiling Buffer management event.

The choice of IMPLEMENTATION DEFINED behavior is not required to be the same for each of:

- An External abort on the write to the Profiling Buffer.
- An External abort on a translation table walk.
- An External abort on a translation table update.

An External abort on a translation table walk or translation table update might be treated as a synchronous MMU fault, as described by *Faults and watchpoints* on page D13-5464.

PMBIDR_EL1.EA indicates to software the mechanism used by the SPU when a write to memory generates an External abort.

The originator of the abort might additionally take other actions. For more information, see *Arm® Architecture Reference Manual Supplement: Reliability, Availability, and Serviceability (RAS), for Armv8-A*.

**The External abort is ignored**

If the External abort is ignored, this has the same visible behavior as when a write does not generate any External abort. PMBSR_EL1 is not modified.

**The External abort generates an SError interrupt exception**

If a write to the Profiling Buffer generates an External abort that is taken as an SError interrupt exception, the PE takes the SError interrupt exception as normal, and PMBSR_EL1 fields are unchanged.

**The External abort is reported to the SPU**

This behavior is not permitted if FEAT_SPEv1p3 is implemented.

If a write to the Profiling Buffer generates an External abort that is reported to the SPU:

- The External abort bit, PMBSR_EL1.EA, is set to 1.
- The SPU stops writing sample records to the Profiling Buffer. It is IMPLEMENTATION DEFINED whether an External abort on a write to the Profiling Buffer is reported as synchronous or asynchronous:
  - The External abort is reported as *synchronous* if PMBPTR_EL1 is set to the address that was externally aborted.
  - The External abort is reported as *asynchronous* if PMBPTR_EL1 is not guaranteed to be set to the address that was externally aborted.
- If the External abort is reported as asynchronous or PMBPTR_EL1 is not the address of the first byte of the sample record being written by the SPU, then PMBSR_EL1.DL is set to 1. Otherwise, PMBSR_EL1.DL is unchanged.

--- **Note** ---

Following an External abort reported asynchronously to the SPU, software must not assume that any valid data has been written to the Profiling Buffer.
• If PMBSR_EL1.S is 0, a buffer management event is generated:
  — PMBSR_EL1.S is set to 1.
  — PMBSR_EL1.EC is set to one of:
    — 0b100100, stage 1 Data Abort on write to buffer.
    — 0b100101, stage 2 Data Abort on write to buffer.
  — PMBSR_EL1.MSS is set as follows:
    — PMBSR_EL1.FSC is set to indicate a synchronous or asynchronous External abort.
• The other PMBSR_EL1 fields are unchanged.
The profiling operation of the SPU:

- Makes indirect reads and indirect writes of System registers.
- Writes to memory.
- Makes further indirect writes to PMBPTR_EL1 as a result of an External abort on a write to memory.

The indirect reads of the PMSCR_EL1.{E1SPE, E0SPE} and PMSCR_EL2.{E2SPE, E0HSPE} controls when determining whether to select an operation for profiling are treated as indirect reads made by the instruction being executed, and subject to the standard requirements for synchronization.

Otherwise, although the profiling operation is generated by a sampled operation, the profiling operation executes independently of the instructions that are executed on the PE.

A DSB instruction guarantees that all memory transactions that are made by the PE are observable by writes made by a profiling operation relating to a sampled operation in program order after the DSB instruction.

A Context synchronization event guarantees that a direct write to a System register made by the PE in program order before the context synchronization event are observable by indirect reads and indirect writes of the same System register made by a profiling operation relating to a sampled operation in program order after the context synchronization event.

To synchronize previous profiling operations, software must execute a PSB CSYNC Buffer Synchronization instruction.

--- Note ---

The PSB CSYNC instruction is not defined in the AArch32 instruction set architecture.

---

Following a context synchronization event, a PSB CSYNC instruction is guaranteed to synchronize the profiling operations for all instructions that are executed in program order before the context synchronization event.

Synchronized by the PSB CSYNC instruction means:

- A direct read of a System register in program order following a PSB CSYNC instruction requires explicit synchronization to observe an indirect write to the same System register made by a profiling operation synchronized by the PSB CSYNC instruction.
- An indirect write to a System register made by a profiling operation synchronized by a PSB CSYNC instruction does not affect a direct write to the same System register made in program order following the PSB CSYNC instruction.
- A direct write to a System register in program order following a PSB CSYNC instruction is not allowed to affect an indirect read of the same System register made by a profiling operation synchronized by the PSB CSYNC instruction.
- A DSB instruction in program order following a PSB CSYNC instruction does not complete before the writes to the Profiling Buffer of sample records for profiling operations synchronized by the PSB CSYNC instruction have completed. The DSB instruction must apply to both loads and stores.

For the indirect write to PMBSR_EL1 that is made as a result of an External abort on a write of a sample record to memory, the synchronization rules apply only after the write has completed.

--- Note ---

If profiling is not disabled when the context synchronization event occurs, further profiling operations might be generated that are not guaranteed to be synchronized by the PSB CSYNC instruction.

If the PE takes an exception to an Exception level where profiling is disabled, no new operations are selected for sampling. Profiling is always disabled if the owning Exception level is a lower Exception level than the current Exception level.

---

In the absence of a context synchronization event, a PSB CSYNC instruction is not required to execute in program order with respect to sampled operations.
D13.9.1 UNPREDICTABLE behavior

In the absence of correct context synchronization events, it is UNPREDICTABLE whether an indirect read of a System register made by a profiling operation will return the old or the new values.

If the indirect reads mean that ProfilingBufferEnabled() returns FALSE when a sample record or records are about to be written to memory, then it is further UNPREDICTABLE whether the sample record or records:

- Are written to memory.
- Are silently discarded and not written to memory.
- Are discarded and not written to memory, and a Profiling Buffer management event is generated:
  - PMBSR_EL1.DL is set to 1.
  - PMBSR_EL1.EC is set to 0x00.
  - PMBSR_EL1.BSC is set to 0x00 to indicate that the buffer is not full.

If SCR_EL3.NS does not match the Security state of the owning translation regime, it is UNPREDICTABLE whether an address translation made by the PE writing a sample record to memory uses the value of SCR_EL3.NS or the identity of the owning translation regime.

This means that software must execute a PSB CSYNC instruction to force any sample records to be written to the Profiling Buffer before changing context.
This chapter describes the sample records generated by the Statistical Profiling Extension. It contains the following sections:

- *About the Statistical Profiling Extension sample records* on page D14-5472.
- *Alphabetical list of Statistical Profiling Extension packets* on page D14-5475.
D14.1 About the Statistical Profiling Extension sample records

The Statistical Profiling Extension sample record format version is identified by PMSIDR_EL1.Format. The architecture currently defines only version 0.

Note

Armv8.7 defines the SPE sample record format version, allowing future architecture updates to extend or change the record format. PMSIDR_EL1.Format was previously a RES0 field in a read-only register. Software that reads and checks PMSIDR_EL1.Format on any implementation prior to Armv8.7 that includes SPE will read a value indicating format version 0 is supported.

The sample record format version 0 is self-describing and extensible. This format allows software to parse profile data even when that profile data contains extended information.

The Statistical Profiling Extension writes a series of sample records to memory, each record consisting of a sequence of packets, and each packet consisting of:

• One or two header bytes.
• Zero, 1, 2, 4, or 8 payload bytes.

D14.1.1 Headers

The first header byte encodes the number of payload bytes:

- 0x00–0x1F: Single byte header, no payload.
- 0x20–0x3F: First byte of extended header. Second byte encodes the payload length.
- 0x40–0x4F, 0x80–0x8F, 0xC0–0xCF: Header with an 8-bit payload.
- 0x50–0x5F, 0x90–0x9F, 0xD0–0xDF: Header with a 16-bit payload.
- 0x60–0x6F, 0xA0–0xAF, 0xE0–0xEF: Header with a 32-bit payload.
- 0x70–0x7F, 0xB0–0xBF, 0xF0–0xFF: Header with a 64-bit payload.

D14.1.2 Records

A record consists of multiple packets. A record comprises, in ascending address order:

• A sequence of headers, each followed by their payload byte or bytes.
• Either:
  — An End packet header.
  — A Timestamp packet.

Figures in this chapter show each packet as a sequence of bytes. Figure D14-1 on page D14-5473 shows how bytes are stored in memory in increasing addresses from left to right.
Figure D14-1 Convention for packet descriptions

In some sections, the figures are split into separate figures for the header byte and payload bytes. For instance, where the number of payload bytes varies according to a field in the header.

The header bytes and payload bytes are described in ascending memory address order. Within a payload value, values are in little-endian byte order.

The size of the access granule for writes to the Profiling Buffer by the Statistical Profiling Unit is IMPLEMENTATION DEFINED, up to a maximum of 2KB. The size of the access granule can vary from time to time.

D14.1.3 Protocol framing packets and forwards compatibility

The padding header, alignment command, Timestamp packet, and end packet are protocol framing packets that frame the records created by the Statistical Profiling Unit. Only padding headers and alignment commands are permitted between records.

--- Note ---

PMBIDR_EL1.Align defines a minimum alignment for records. However, implementations must nevertheless create a valid protocol stream that can be parsed without knowledge of the minimum alignment.

---

The packet types are described in the following sections. Software must ignore unknown packets, using the size field encoded in the header. This includes packets containing reserved values in fields.

The following sections give an overview of the Statistical Profiling Unit packets output to a memory-mapped Profiling Buffer or Device memory:

- Statistical Profiling Extension protocol packet headers

D14.1.4 Statistical Profiling Extension protocol packet headers

8-bit headers

For Address packets and Counter packets, the 8-bit header format is described as the short format.

Table D14-1 8-bit header encodings

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Padding on page D14-5497</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>End packet on page D14-5485</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Timestamp packet on page D14-5498</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Events packet on page D14-5486</td>
</tr>
</tbody>
</table>
16-bit headers

For Address packets and Counter packets, the 16-bit header format is described as the extended format.

Table D14-2 16-bit header encodings

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 0 x x</td>
<td>1 0 1 1 0 x x x</td>
<td>Address packet on page D14-5476</td>
</tr>
<tr>
<td>0 0 1 0 0 0 x x</td>
<td>1 0 0 1 1 x x x</td>
<td>Counter packet on page D14-5481</td>
</tr>
</tbody>
</table>
D14.2 Alphabetical list of Statistical Profiling Extension packets

This section lists every SPE packet and their description.
D14.2.1 Address packet

The Address packet characteristics are:

**Purpose**  Provides an address value for the record. Addresses are always 64 bits.

**Attributes**  Multi-part packet comprising:

- 8 or 16-bit header.
- 64-bit payload.

Address packet header

When Extended format is used, the Address packet header bit assignments are:

<table>
<thead>
<tr>
<th>Bit Assignments</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 0</td>
<td>0 0 1 0 0 0 0 INDEX[4:3]</td>
</tr>
<tr>
<td>Byte 1</td>
<td>1 0 0 SZ 1 1 0 INDEX[2:0]</td>
</tr>
</tbody>
</table>

When Short format is used, the Address packet header bit assignments are:

<table>
<thead>
<tr>
<th>Bit Assignments</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 0</td>
<td>1 0 0 SZ 1 1 0 INDEX</td>
</tr>
</tbody>
</table>

**Byte 1 bits [7:6], when Extended format, Byte 0 bits [7:6], when Short format**

This field reads as 0b10.

**SZ, byte 1 bits [5:4], when Extended format, SZ, byte 0 bits [5:4], when Short format**

Payload size. The defined values of this field are:

- 0b11 Doubleword.
  
  This field reads as 0b11.

**Byte 1 bit [3], when Extended format, Byte 0 bit [3], when Short format**

This bit reads as 0b0.

**Byte 0 bits [7:5], when Extended format**

This field reads as 0b001.

**Byte 0 bits [4:2], when Extended format**

This field reads-as-zero.

**INDEX, byte 0 bits [1:0], byte 1 bits [2:0], when Extended format, INDEX, byte 0 bits [2:0], when Short format**

The defined values of this field are:

- 0b00000 Issued instruction virtual address (PC). Included for all operations.
- 0b00001 Branch target address:
  - It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether this address is included for an Exception Return to an Exception level where profiling is disabled.
  - Included for all other branch and exception return instructions.
- 0b00010 Data access virtual address. Included for all load, store and atomic operations.
- 0b00011 Data access physical address:
  - Not included if disabled by CollectPhysicalAddress, or if the access generates an abort other than a Permission or Access Flag fault.
• It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether this is included for accesses that generate Permission or Access Flag faults.
• Included for all other load, store and atomic operations.

0b00100  Previous branch target address. The target virtual address of the most recently taken branch operation in program order before the sampled operation. PMSIDR_EL1.PBT indicates whether this is implemented. This value is defined when FEAT_SPEv1p2 is implemented and reserved otherwise.

0b0011x  IMPLEMENTATION DEFINED address.

0b1xxxx  IMPLEMENTATION DEFINED address.

All other values are reserved.

In the Short format header, bits [4:3] are zero.

Address packet payload

When Data access physical address, the Address packet payload bit assignments are:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR[7:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDR[15:8]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDR[23:16]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDR[31:24]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDR[39:32]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDR[47:40]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDR[55:48]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NS  CH  0  0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PAT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Byte 0

Byte 1

Byte 2

Byte 3

Byte 4

Byte 5

Byte 6

Byte 7
When Data access virtual address, the Address packet payload bit assignments are:

![Address packet payload bit assignments diagram for Data access virtual address]

When Instruction virtual address, the Address packet payload bit assignments are:

![Address packet payload bit assignments diagram for Instruction virtual address]

**TAG byte <7>, when Data access virtual address**

Top-byte tag.

If the value of the applicable TBI bit is one, a data access virtual address includes the top-byte tag. If the applicable TBI bit is zero, it is IMPLEMENTATION DEFINED whether this field reads as zero or holds the address tag of the applicable address.
NS, byte 7 bit [7], when Instruction virtual address
Non-secure state. The Security state associated with the address. For an issued instruction virtual address (PC) this is the Security state the instruction was executed in. For a branch target address, this is the Security state at the target of the branch. The defined values of this bit are:
0 Secure state.
1 Non-secure state.

NS, byte 7 bit [7], when Data access physical address
Physical address space identifier. The Security attribute for the physical address. The defined values of this bit are:
0 Secure physical address space.
1 Non-secure physical address space.

CH, byte 7 bit[6], when Data access physical address
When FEAT_MTE2 is implemented, Checked access identifier. Checked or Unchecked access. The defined values of this bit are:
0 Tag Unchecked access.
1 Tag Checked access.
For more information, see Chapter D9 The Memory Tagging Extension.
If FEAT_MTE2 is not implemented this bit is RAZ.
When Tag Check Faults are configured to be ignored by SCTLR_ELx.TCF or SCTLR_ELx.TCF0, it is IMPLEMENTATION DEFINED whether this bit is 1 or 0 on a Tag Checked access.

EL, byte 7 bits [6:5], when Instruction virtual address
Exception level. The Exception level associated with the address. For an issued instruction virtual address (PC) this is the Exception level the instruction was executed in. For a branch target address, this is the Exception level at the target of the branch. The defined values of this field are:
0b00 EL0.
0b01 EL1.
0b10 EL2.
0b11 EL3.

--- Note ---
For an Exception Return, the Exception level at the target of the branch might be different to the Exception level the instruction was executed in.

---

Byte 7 bits [5:4], when Data access physical address
This field reads as 0b00.

PAT, Byte 7 bits [3:0], when Data access physical address
When FEAT_MTE2 is implemented, this field provides the Physical Address Tag for a Tag Checked access. If the access is Unchecked this field reads as an IMPLEMENTATION DEFINED choice between 0b0000 and the Physical Address Tag used to perform the access.
For more information, see Chapter D9 The Memory Tagging Extension.
If FEAT_MTE2 is not implemented this field is RAZ.

Byte 7 bits [4:0], when Instruction virtual address
This field reads as 0b0000.

ADDR, bytes <6:0>
D14.2.2 Context packet

The Context packet characteristics are:

**Purpose**
Provides context information for the record.

**Attributes**
Multi-part packet comprising:
- 8-bit header.
- 32-bit payload.

### Context packet header

The Context packet header bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SZ, byte 0 bits [5:4]</td>
<td>0b10</td>
</tr>
<tr>
<td></td>
<td>Payload size. The defined values of this field are:</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0b10 Word.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>INDEX, byte 0 bits [1:0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Identifies the context value. The defined values of this field are:</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0b00 CONTEXTIDR_EL1. Included for all operations if enabled by CollectContextIDR1.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0b01 CONTEXTIDR_EL2. Included for all operations if enabled by CollectContextIDR2.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>All other values are reserved.</td>
<td></td>
</tr>
</tbody>
</table>

### Context packet payload

The Context packet payload bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CONTEXT[7:0]</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>CONTEXT[15:8]</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CONTEXT[23:16]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>CONTEXT[31:24]</td>
<td></td>
</tr>
</tbody>
</table>

**CONTEXT, bytes <3:0>**

The context value.
D14.2.3 Counter packet

The Counter packet characteristics are:

**Purpose**
Count of cycles the operation spent performing all or part of its behavior. The counter value occupies the least significant bits of the payload. The remaining bits are set to zero.

**Attributes**
Multi-part packet comprising:
- 8 or 16-bit header.
- 16-bit payload.

### Counter packet header

**When Extended format**, the Counter packet header bit assignments are:

```
  7  6  5  4  3  2  1  0
0 0 0 1 0 0 0 INDEX[4:3] Byte 0
1 0 SZ 0 1 INDEX[2:0] Byte 1
```

**When Short format**, the Counter packet header bit assignments are:

```
  7  6  5  4  3  2  1  0
1 0 SZ 0 1 INDEX Byte 0
```

**Byte 1 bits [7:6], when Extended format, Byte 0 bits [7:6], when Short format**

This field reads as `0b10`.

**SZ, byte 1 bits [5:4], when Extended format, SZ, byte 0 bits [5:4], when Short format**
Payload size. The defined values of this field are:
- `0b01` Halfword.
  This field reads as `0b01`.

**Byte 1 bit [3], when Extended format, Byte 0 bit [3], when Short format**

This bit reads as `0b1`.

**Byte 0 bits [7:5], when Extended format**

This field reads as `0b001`.

**Byte 0 bits [4:2], when Extended format**

This field reads-as-zero.

**INDEX, byte 0 bits [1:0], byte 1 bits [2:0], when Extended format, INDEX, byte 0 bits [2:0], when Short format**

The defined values of this field are:
- `0b0000` Total latency. Cycle count from the operation being dispatched for issue to the operation being microarchitecturally-finished. Included for all operations.
- `0b0001` Issue latency. Cycle count from the operation being dispatched for issue to the operation being issued for execution. This counts any delay in waiting the operation being ready to issue. Included for all operations.
- `0b0010` Translation latency. Cycle count from a virtual address being passed to the MMU for translation to the result of the translation being available. Included for all load, store and atomic operations.
- `0b0011x` IMPLEMENTATION DEFINED counter value.
0b1xxxx  IMPLEMENTATION DEFINED counter value.
All other values are reserved.
In the Short format header, bits [4:3] are zero.
For the purposes of defining these counter values:
  • Dispatched for issue means:
    — The operation has been decoded.
    — The operation might not be ready to start execution because it is waiting for input values. The operation might be put into a queue.
  • Issued for execution means the operation is ready to start executing:
    — For example, for a memory operation, this should be indicative of the cycle count from memory operation being dispatched for issue to access being initiated (virtual address).
  • Microarchitecturally-finished means:
    — The operation has completed execution and is no longer capable of stalling any instruction that consumes its output. The results of the operation are not required to be coherent or observable by other PEs.
    — It is IMPLEMENTATION DEFINED whether the operation is speculative, or has committed its results to the architectural state of the PE.
    — For example:
      — For an arithmetic, floating-point, or SIMD operation with variable timing, such as divide, the results of the operation are available.
      — For load and atomic operations that return data, all data have been returned from memory.
      — For store and atomic operations that do not return data, it is not required that the store is complete for other observers.
      — For branch operations, the branch has been resolved as taken or not taken.
      — For barrier operations, the barrier has completed.
For WFE and WFI operations, it is IMPLEMENTATION DEFINED whether:
  • The instruction is complete before the PE enters a low-power state or when the PE wakes from the low-power state.
  • Counters count in the low power state.
  • Sampling an operation is itself a wake-up event.

**Counter packet payload**

It is IMPLEMENTATION DEFINED whether Counter packet is 12-bit or 16-bit counters. PMSIDR_EL1.CountSize indicates the size of counter implemented.

When 12-bit counters are implemented, the Counter packet payload bit assignments are:

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT[7:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Byte 1 bits [7:4]**

This field reads-as-zero.
COUNT, byte 1 bits [3:0], byte <0>

The counter value occupies the least significant bits of the payload. The remaining bits are set to zero. The counters are:

- Unsigned numbers.
- 12 bits.
- Saturating.
- The value 0xFFF indicates the count has saturated.

When 16-bit counters are implemented, the Counter packet payload bit assignments are:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COUNT[7:0]</td>
<td>Byte 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COUNT[15:8]</td>
<td>Byte 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

COUNT, bytes <1:0>

The counter value occupies the least significant bits of the payload. The remaining bits are set to zero. The counters are:

- Unsigned numbers.
- 16 bits.
- Saturating.
- The value 0xFFFF indicates the count has saturated.
D14.2.4 Data Source packet

The Data Source packet characteristics are:

**Purpose**

If the implementation includes support for indicating the loaded data source, the Data Source packet indicates where the data returned for a load operation was sourced. It might also include other information, such as the state of the data at the source. It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether this is included for load and atomic operations that generate an External abort. It is IMPLEMENTATION DEFINED whether this is included for atomic operations that do not return data to a PE register. Included for all other load and atomic operations.

**Attributes**

Multi-part packet comprising:
- 8-bit header.
- 8 or 16-bit payload.

**Data Source packet header**

The Data Source packet header bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>0b01</td>
</tr>
<tr>
<td>5-4</td>
<td>Payload size. Defined values are: 0b00 - Byte, 0b01 - Halfword</td>
</tr>
<tr>
<td>3-0</td>
<td>0b0011</td>
</tr>
</tbody>
</table>

**Byte 0 bits [7:6]**

This field reads as 0b01.

**SZ, byte 0 bits [5:4]**

Payload size. The defined values of this field are:
- 0b00: Byte
- 0b01: Halfword

**Byte 0 bits [3:0]**

This field reads as 0b0011.

**Data Source packet payload**

When \(SZ = 0b00\), the Data Source packet payload bit assignments are:

```
7 6 5 4 3 2 1 0
SOURCE
```

When \(SZ = 0b01\), the Data Source packet payload bit assignments are:

```
7 6 5 4 3 2 1 0
  SOURCE[7:0]
```

```
7 6 5 4 3 2 1 0
  SOURCE[15:8]
```

**SOURCE, byte <0>**, when \(SZ = 0b00\), **SOURCE, bytes <1:0>**, when \(SZ = 0b01\)**

Because the list of data sources varies from system to system, the definition of this field is IMPLEMENTATION DEFINED. If a sampled operation generated multiple data accesses, it is IMPLEMENTATION DEFINED how the data source information is combined.
D14.2.5 End packet

The End packet characteristics are:

**Purpose**
Defines the end of a record if a Timestamp packet is not present.

**Attributes**
8-bit packet.

**Field descriptions**

The End packet bit assignments are:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Byte <0>**

This field reads as 0b00000001.
D14.2.6 Events packet

The Events packet characteristics are:

**Purpose**

Indicates up to 64 events generated by the sampled operation. If FEAT_PMUv3p1 is implemented and an event counter is configured to count PMU events, then a sampled operation that causes the event counter to be incremented has the event recorded as one, and conversely a sampled operation that does not cause the counter to be incremented is recorded as zero.

**Note**

Arm recommends that the Performance Monitors Extension implements the Events.

**Attributes**

Multi-part packet comprising:

- 8-bit header.
- 8, 16, 32, or 64-bit payload.

**Events packet header**

The Events packet header bit assignments are:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>SZ</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Byte 0 bits [7:6]**

This field reads as 0b01.

**SZ, byte 0 bits [5:4]**

Payload size. The defined values of this field are:

- 0b00 Byte.
- 0b01 Halfword.
- 0b10 Word.
- 0b11 Doubleword.

Software must treat bits that are not output as zero.

**Byte 0 bits [3:0]**

This field reads as 0b0010.

**Events packet payload**

When **SZ** == 0b00, the Events packet payload bit assignments are:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

When **SZ** == 0b01, the Events packet payload bit assignments are:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>
When SZ == 0b10, the Events packet payload bit assignments are:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>
| 0 | 0 | 0 | 0 | 0 | E[18] | E[17] | E[16] | Byte 2
| E[31:24] | Byte 3

When SZ == 0b11, the Events packet payload bit assignments are:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>
| 0 | 0 | 0 | 0 | 0 | E[18] | E[17] | E[16] | Byte 2
| E[31:24] | Byte 3
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | Byte 4
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | Byte 5
| E[55:48] | Byte 6
| E[63:56] | Byte 7

E[63:48], bytes <7:6>, when SZ == 0b11

Events 63 to 48. IMPLEMENTATION DEFINED.

Bytes <5:4,2>, byte 1 bit [3], when SZ == 0b11

This field reads-as-zero.

E[31:24], byte <3>, when SZ == 0b10, or when SZ == 0b11

Events 31 to 24. IMPLEMENTATION DEFINED.

E[18], byte 2 bit [18], when SZ == 0b10, or SZ == 0b11

Empty predicate.

When The Scalable Vector Extension (SVE) and FEAT_SPEv1p1 are implemented the defined values of this bit are:

0 Operation was not an SVE operation, was unpredicated or executed with all elements Active.

1 SVE operation executed with all elements Inactive.

Otherwise this bit reads-as-zero.
If \texttt{FEAT\_PMUv3} and \texttt{FEAT\_SVE} are implemented, this Event is required to be implemented consistently with \texttt{SVE\_PRED\_EMPTY\_SPEC}.

\textbf{E[17], byte 2 bit [17], when SZ == \texttt{0b10}, or SZ == \texttt{0b11}}

Partial predicate.

When The Scalable Vector Extension (SVE) and \texttt{FEAT\_SPEv1p1} are implemented the defined values of this bit are:

0 Operation was not an SVE operation, was unpredicated, or executed with all elements Active.

1 Predicated SVE operation executed with at least one Inactive element.

Otherwise this bit reads-as-zero.

If \texttt{FEAT\_PMUv3} and \texttt{FEAT\_SVE} are implemented this Event is required to be implemented consistently with \texttt{SVE\_PRED\_NOT\_FULL\_SPEC}.

\textbf{E[15:12], byte 1 bits [7:4], when SZ == \texttt{0b01, when SZ == 0b10, or when SZ == 0b11}}

Events 15 to 12. IMPLEMENTATION DEFINED.

\textbf{E[11], byte 1 bit [3], when SZ == \texttt{0b01, when SZ == 0b10, or when SZ == 0b11}}

Alignment.

When \texttt{FEAT\_SPEv1p1} is implemented the defined values of this bit are:

0 Load/store operation that was optimally aligned for the size of data being accessed.

1 Load/store operation that, due to the alignment of the address and size of data being accessed, incurred additional latency.

Otherwise this bit reads-as-zero.

If \texttt{FEAT\_PMUv3} is implemented this Event is required to be implemented consistently with \texttt{LDST\_ALIGN\_LAT}.

\textbf{E[10], byte 1 bit [2], when SZ == \texttt{0b01, when SZ == 0b10, or when SZ == 0b11}}

Remote access. The defined values of this bit are:

0 Did not cause access to another socket.

1 Load/store operation caused an access to another socket in a multi-socket system. This includes each data memory access that accesses another socket in a multi-socket system, including those that do not return data.

This event is optional. When this event is implemented, it is further IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether a store can finish execution before this event is generated, meaning this event is never recorded for stores.

If this event and \texttt{FEAT\_PMUv3} are both implemented, this event is required to be implemented consistently with \texttt{REMOTE\_ACCESS} or \texttt{REMOTE\_ACCESS\_RD}.

\textbf{E[9], byte 1 bit [1], when SZ == \texttt{0b01, when SZ == 0b10, or when SZ == 0b11}}

Last Level cache miss. The defined values of this bit are:

0 Did not miss Last Level cache.

1 Load/store operation caused an access to at least the Last Level cache but is not completed by the Last Level cache. That is, each:

- Load operation that does not return data from the Last Level cache.
- Store operation that does not update the Last Level cache.

The event is not set for operations that are completed by a cache above the Last Level cache.

This event is optional. When this event is implemented, it is further IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether a store can finish execution before this event is generated, meaning this event is never recorded for stores.
If this event and FEAT_PMUv3 are both implemented, this event is required to be implemented consistently with LL_CACHE_MISS or LL_CACHE_MISS_RD.

E[8], byte 1 bit [0], when SZ == 0b01, when SZ == 0b10, or when SZ == 0b11
Last Level cache access. The defined values of this bit are:
0  Did not access Last Level data or unified cache.
1  Load/store operation caused a cache access to at least the Last Level data or unified cache.

Note
The architecture does not define the Last Level cache. The Last Level cache is typically the largest cache on this device shared by all PEs in the Inner or Outer Shareable domain of this PE. In a multi-socket system, it is IMPLEMENTATION DEFINED whether this includes caches on other sockets.

This event is optional. When this event is implemented, it is further IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether a store can finish execution before this event is generated, meaning this event is never recorded for stores.

If this event and FEAT_PMUv3 are both implemented, this event is required to be implemented consistently with LL_CACHE or LL_CACHE_RD.

E[7], byte 0 bit [7]
Mispredicted. The defined values of this bit are:
0  Did not cause correction to the predicted program flow.
1  A branch that caused a correction to the predicted program flow.

If FEAT_PMUv3 is implemented this Event is required to be implemented consistently with either BR_MIS_PRED or BR_MIS_PRED_RETIRED.

E[6], byte 0 bit [6]
Not taken. The defined values of this bit are:
0  Did not fail condition code check.
1  A conditional instruction that failed its condition code check. This includes conditional branches, compare-and-branch, conditional select, and conditional compares:
   •  For a conditional branch or compare-and-branch instruction, this means the branch was not taken.
   •  For a conditional select, this means the second operand was written to the result.
   •  For a condition compare, this means the condition flags were set to the immediate value and not the result of the compare.

This field is valid only if the OpType defines this as either a Conditional Branch or a Conditional operation, and is RAZ otherwise.

E[5], byte 0 bit [5]
TLB walk. The defined values of this bit are:
0  Did not generate TLB walk.
1  Load/store operation that causes a refill of a data or unified TLB, involving at least one translation table walk access. This includes each complete or partial translation table walk that causes an access to memory, including to data or translation table walk caches.

If FEAT_PMUv3 is implemented this Event is required to be implemented consistently with DTLB_WALK.

E[4], byte 0 bit [4]
TLB access. The defined values of this bit are:
0  Did not access TLB.
1  Load/store operation caused an access to at least the first level of data or unified TLB.
If FEAT_PMUv3 is implemented this Event is required to be implemented consistently with L1D_TLB.

E[3], byte 0 bit [3]
Level 1 Data cache refill. The defined values of this bit are:
0  Did not cause level 1 data cache refill.
1  Load/store operation caused a refill of at least the first level of data or unified cache. This includes each data memory access that causes a refill from outside the cache. It excludes accesses that do not cause a new cache refill but are satisfied from refilling data of a previous miss.

It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether a store can finish execution before this event is generated, meaning this event is never recorded for stores.

If FEAT_PMUv3 is implemented this event is required to be implemented consistently with L1D_CACHE_REFILL or L1D_CACHE_REFILL_RD.

E[2], byte 0 bit [2]
Level 1 Data cache access. The defined values of this bit are:
0  Did not access level 1 data cache.
1  Load/store operation caused a cache access to at least the first level of data or unified cache.

It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether a store can finish execution before this event is generated, meaning this event is never recorded for stores.

If FEAT_PMUv3 is implemented this event is required to be implemented consistently with L1D_CACHE or L1D_CACHE_RD.

E[1], byte 0 bit [1]
Architecturally executed. The defined values of this bit are:
0  Did not retire.
1  Committed its results to the architectural state of the PE, or completed with a synchronous architectural exception.

Note
A conditional instruction can retire even if it fails its condition code check.

If FEAT_PMUv3 is implemented this Event is required to be implemented consistently with INST_RETIRED.

E[0], byte 0 bit [0]
Generated exception. The defined values of this bit are:
0  Did not generate an exception.
1  Completed with a synchronous exception.

If E[1] in the same Events packet is set to 0, then the meaning of this bit is IMPLEMENTATION DEFINED.

If FEAT_PMUv3 is implemented this Event is required to be implemented consistently with EXC_TAKEN.
### D14.2.7 Operation Type packet

The Operation Type packet characteristics are:

**Purpose**
- Defines the type of operation sampled. Included for all operations.

**Attributes**
- Multi-part packet comprising:
  - 8-bit header.
  - 8-bit payload.

#### Operation Type packet header

The Operation Type packet header bit assignments are:

<table>
<thead>
<tr>
<th>Bit Assignment</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 0 bits [7:6]</td>
<td>0b01</td>
<td>This field reads as 0b01.</td>
</tr>
<tr>
<td>SZ, byte 0 bits [5:4]</td>
<td>0b00</td>
<td>Payload size. The defined values of this field are: 0b00 Byte. This field reads as 0b00.</td>
</tr>
<tr>
<td>Byte 0 bits [3:2]</td>
<td>0b10</td>
<td>This field reads as 0b10.</td>
</tr>
<tr>
<td>CLASS, byte 0 bits [1:0]</td>
<td></td>
<td>Top-level instruction class. The defined values of this field are: 0b00 Other. 0b01 Load, store, or atomic. 0b10 Branch or exception return. All other values are reserved.</td>
</tr>
</tbody>
</table>

#### Operation Type packet payload (Other)

When Other operation, the Operation Type packet payload (Other) bit assignments are:  

<table>
<thead>
<tr>
<th>Bit Assignment</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 0</td>
<td>0b0000000</td>
<td></td>
</tr>
</tbody>
</table>
When an SVE operation, the Operation Type packet payload (Other) bit assignments are:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EVL</td>
<td>1</td>
<td>PRED</td>
<td>FP</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SUBCLASS, byte<0>**

Second-level instruction class. Defines the type of instruction. The defined values of this field are:

- **0b0000000x** Other operation.
- **0b0xxx1xx0** SVE operation. If FEAT_SVE is implemented, and if FEAT_SPE is implemented, bits [6:4:2:1] are further defined as the EVL, PRED, and FP fields. Otherwise this value is reserved.

**EVL, byte 0 bits [6:4], when SVE operation**

Effective Vector Length. Defines the sampled operation vector length, rounded up to a power of two. That is, the length of vector operated on by the sampled operation. The defined values of this field are:

- **0b000** 32 bits.
- **0b001** 64 bits.
- **0b010** 128 bits.
- **0b011** 256 bits.
- **0b100** 512 bits.
- **0b101** 1024 bits.
- **0b110** 2048 bits.

All other values reserved.

The accessible vector length is always quantized into multiples of 128 bits. However, the effective vector length can be any size down to the smallest element size.

If the effective vector length is not a power of two, or is less than 32 bits, the value is rounded up before it is encoded in this field.

**PRED, byte 0 bit[2], when SVE operation**

Predicated SVE operation. The defined values of this bit are:

- **0** Not predicated.
- **1** Predicated SVE operation. The operation is an SVE operation that writes to a vector destination register under a Governing predicate using either zeroing or merging predication.

**FP, byte 0 bits [6:4], when SVE operation**

Floating-point operation. The defined values of this bit are:

- **0** Integer.
- **1** Floating-point.

**COND, byte 0 bit [0], when Other operation**

Conditional. The defined values of this bit are:

- **0** Unconditional operation.
- **1** Conditional select or conditional compare operation. See Additional information for each profiled conditional instruction on page D13-5451.
**Operation Type packet payload (Branch)**

The Operation Type packet payload (Branch) bit assignments are:

```
   7 6 5 4 3 2 1 0
   SUBCLASS COND Byte 0
```

**SUBCLASS, byte <0>**

Second-level instruction class. Describes the branch type. The defined values of this field are:

- 0b0000000x Direct branch.
- 0b0000001x Indirect branch.
- All other values are reserved.

**COND, byte 0 bit [0]**

Conditional. The defined values of this bit are:

- 0 Unconditional branch.
- 1 Conditional branch.

**Operation Type packet payload (load/store)**

When FEAT_NV2 transformed System register access, the Operation Type packet payload (load/store) bit assignments are:

```
   7 6 5 4 3 2 1 0
   SUBCLASS LDST Byte 0
```

When Extended load/store, the Operation Type packet payload (load/store) bit assignments are:

```
   7 6 5 4 3 2 1 0
   SUBCLASS AR EXCL AT 1 LDST Byte 0
```

When General-purpose load/store, the Operation Type packet payload (load/store) bit assignments are:

```
   7 6 5 4 3 2 1 0
   SUBCLASS 0 0 0 0 0 0 0 LDST Byte 0
```

When SIMD&FP load/store, the Operation Type packet payload (load/store) bit assignments are:

```
   7 6 5 4 3 2 1 0
   SUBCLASS 0 0 0 0 0 1 0 LDST Byte 0
```
When SVE load/store, the Operation Type packet payload (load/store) bit assignments are:

```
<table>
<thead>
<tr>
<th>SUBCLASS</th>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SG</td>
<td>EVL</td>
</tr>
<tr>
<td>1</td>
<td>PRED</td>
</tr>
<tr>
<td>0</td>
<td>LDST</td>
</tr>
</tbody>
</table>
```

When Unspecified load/store, the Operation type packet payload (load/store) bit assignments are:

```
<table>
<thead>
<tr>
<th>SUBCLASS</th>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000</td>
</tr>
<tr>
<td>1</td>
<td>0000000</td>
</tr>
<tr>
<td>2</td>
<td>0000000</td>
</tr>
<tr>
<td>3</td>
<td>0000000</td>
</tr>
<tr>
<td>4</td>
<td>0000000</td>
</tr>
</tbody>
</table>
```

When Tags load/store, the Operation Type packet payload (load/store) bit assignments are:

```
<table>
<thead>
<tr>
<th>SUBCLASS</th>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000</td>
</tr>
<tr>
<td>1</td>
<td>0000000</td>
</tr>
<tr>
<td>2</td>
<td>0000000</td>
</tr>
<tr>
<td>3</td>
<td>0000000</td>
</tr>
<tr>
<td>4</td>
<td>0000000</td>
</tr>
</tbody>
</table>
```

When Memory Copy load/store, the Operation Type packet payload (load/store) bit assignments are:

```
<table>
<thead>
<tr>
<th>SUBCLASS</th>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000</td>
</tr>
<tr>
<td>1</td>
<td>0000000</td>
</tr>
<tr>
<td>2</td>
<td>0000000</td>
</tr>
<tr>
<td>3</td>
<td>0000000</td>
</tr>
<tr>
<td>4</td>
<td>0000000</td>
</tr>
</tbody>
</table>
```

When Memory Set load/store, the Operation Type packet payload (load/store) bit assignments are:

```
<table>
<thead>
<tr>
<th>SUBCLASS</th>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000</td>
</tr>
<tr>
<td>1</td>
<td>0000000</td>
</tr>
<tr>
<td>2</td>
<td>0000000</td>
</tr>
<tr>
<td>3</td>
<td>0000000</td>
</tr>
</tbody>
</table>
```

**SUBCLASS, byte <0>**

Second-level instruction class. Indicates the load/store type. The defined values of this field are:

- **0b0000000x** A load/store targeting the general-purpose registers, other than an atomic operation, load-acquire, store-release or exclusive.
- **0b0000001x** An atomic operation, load-acquire, store-release or exclusive. Bits [4:2] are further subdivided as described by the AR, EXCL and AT fields.
- **0b0000010x** A load/store targeting the SIMD&FP registers.
- **0b0000011x** A load/store targeting the SVE registers. Bits [7:4.2] are further defined as SG, EVL and PRED fields.
- **0b0001000x** A load/store targeting unspecified registers.
  - This value is defined only if FEAT_SPEv1p1 is implemented.
  - This value is reserved otherwise.
- **0b0011000x** An MRS or MSR operation at EL1 transformed to a load/store when HCR_EL2.NV2 is 1.
  - This value is defined only if FEAT_NV2 is implemented and reserved otherwise.
0b0001010x A load/store of an Allocation Tag or multiple Allocation Tags.
This value is defined only if FEAT_SPEv1p3 and FEAT_MTE are implemented. This value is reserved otherwise.

0b0010000x A load/store from a Memory Copy operation. This value is defined only if
FEAT_SPEv1p3 and FEAT_MOPS are implemented. This value is reserved otherwise.

0b00100101 A store from a Memory Set operation. This value is defined if FEAT_SPEv1p3 and
FEAT_MOPS are implemented. This value is reserved otherwise.

All other values are reserved.

SG, byte 0 bit [7], when SVE load/store
Gather/scatter load/store. The defined values of this bit are:
0 Not gather load or scatter store.
1 Gather load or scatter store.

EVL, byte 0 bits [6:4], when SVE load/store
Effective Vector Length. Defines the sampled operation vector length, rounded up to a power of two.
That is, the length of vector operated on by the sampled operation. The defined values of this field are:
0b00 32 bits.
0b01 64 bits.
0b10 128 bits.
0b11 256 bits.
0b100 512 bits.
0b101 1024 bits.
0b110 2048 bits.
All other values reserved.
The accessible vector length is always quantized into multiples of 128 bits. However, the effective vector length can be any size down to the smallest element size.
If the effective vector length is not a power of two, or is less than 32 bits, the value is rounded up before it is encoded in this field.

AR, byte 0 bit [4], when Extended load/store
Acquire/Release. The defined values of this bit are:
0 Load/store/atomic without Acquire or Release semantics.
1 Load/store/atomic with Acquire or Release semantics.

EXCL, byte 0 bit [3], when Extended load/store
Exclusive. The defined values of this bit are:
0 Load/store/atomic without Exclusive.
1 Load/store with Exclusive.
This bit is RES0 if AT == 1.

PRED, byte 0 bit[2], when SVE load/store
Predicated SVE operation. The defined values of this bit are:
0 Not predicated.
1 Predicated SVE operation. The operation is one of the following:
   • If FEAT_SPEv1p2 is implemented, a predicated load operation that writes to one or more vector destination registers under a Governing predicate using zeroing predication.
   • A predicated store of one or more vector registers.
If FEAT_SPEv1p2 is not implemented, it is IMPLEMENTATION DEFINED whether this field is 0 or 1 for a predicated load operation that writes to one or more vector destination registers under a Governing predicate using zeroing predication.

**AT, byte 0 bit [2], when Extended load/store**
Atomic load/store. The defined values of this bit are:
- 0: Not atomic.
- 1: Atomic.

**LDST, byte 0 bit [0], when Memory Copy operation**
Store not load. The defined values of this bit are:
- 0: Load, sampled virtual address is the source address.
- 1: Store, sampled virtual address is the destination address.

**LDST, byte 0 bit [0], otherwise**
Store not load. The defined values of this bit are:
- 0: Load or swap.
- 1: Store.
This bit is always one for a Memory Set operation.
**D14.2.8 Padding**

The Padding characteristics are:

**Purpose**
Allows the PE to create alignment in the protocol buffer.

**Attributes**
8-bit packet.

**Field descriptions**

The Padding bit assignments are:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Byte <0>

This field reads as 0b00000000.
D14.2.9  Timestamp packet

The Timestamp packet characteristics are:

**Purpose**  The 64-bit timestamp value when the operation was sampled. The Timestamp packet must come at the end of the record. If the Timestamp packet is not present, an End packet must come at the end of the record.

**Attributes**  Multi-part packet comprising:
- 8-bit header.
- 64-bit payload.

**Timestamp packet header**

The Timestamp packet header bit assignments are:

```
<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**Byte 0 bits [7:6]**

This field reads as 0b01.

**SZ, byte 0 bits [5:4]**

Payload size. The defined values of this field are:
- 0b11  Doubleword.
  This field reads as 0b11.

**Byte 0 bits [3:0]**

This field reads as 0b0001.
**Timestamp packet payload**

The Timestamp packet payload bit assignments are:

```
<table>
<thead>
<tr>
<th>Bit Assignments</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS[7:0]</td>
<td>0</td>
</tr>
<tr>
<td>TS[15:8]</td>
<td>1</td>
</tr>
<tr>
<td>TS[23:16]</td>
<td>2</td>
</tr>
<tr>
<td>TS[31:24]</td>
<td>3</td>
</tr>
<tr>
<td>TS[39:32]</td>
<td>4</td>
</tr>
<tr>
<td>TS[47:40]</td>
<td>5</td>
</tr>
<tr>
<td>TS[55:48]</td>
<td>6</td>
</tr>
<tr>
<td>TS[63:56]</td>
<td>7</td>
</tr>
</tbody>
</table>
```

**TS, bytes <7:0>**

Timestamp value when the operation was sampled. The value depends on the result of `CollectTimeStamp()`:

- If `TimeStamp_Virtual`, this is the virtual timestamp, `CNTVCT_EL0`.
- If `TimeStamp_Physical`, this is the physical timestamp, `CNTPCT_EL0`.
- If `TimeStamp_OffsetPhysical`, this is the offset physical timestamp, `CNTPCT_EL0 - CNTPOFF_EL2`.
- If `TimeStamp_None`, the Timestamp packet is not included and an End packet must come at the end of the record.

However, if the Generic Timer System counter is disabled and `CollectTimeStamp()` returns a value other than `TimeStamp_None`, then it is IMPLEMENTATION DEFINED whether:

- The Statistical Profiling Unit behaves as if `CollectTimeStamp()` returns the value `TimeStamp_None`.
- The value of this field in the record is UNKNOWN.

**Note**

This relaxation refers to when the actual System counter is disabled, that is, CNTEN.EN == 0. It does not apply when the System counter is enabled but not accessible at the current Exception level.
Chapter D15
The Branch Record Buffer Extension

This chapter describes the Branch Record Buffer Extension (BRBE). It contains the following sections:

• About the Branch Record Buffer Extension on page D15-5502.
• Branch record filtering on page D15-5507.
• Branch record buffer operation on page D15-5511.
• Branch record buffer on page D15-5513.
• Programmers’ model on page D15-5515.
D15.1 About the Branch Record Buffer Extension

The Branch Record Buffer Extension provides control path information for compiling and optimizing software. These directed optimizations extract information about hotspots and common control paths in the code. FEAT_BRBE provides a mechanism for capturing control path history in a low-cost manner. FEAT_BRBEv1p1 extends FEAT_BRBE to enable branch recording at EL3.

D15.1.1 Branch records

Each Branch record consists of 3 registers:
• BRBINF<n>_EL1.
• BRBSRC<n>_EL1.
• BRBTGT<n>_EL1.

Taken branch instructions, as defined by Branches, Exception generating, and System instructions on page C3-248, generate a Branch record.

Exceptions generate a Branch record.

A Half-source Branch record has BRBINF<n>_EL1.VALID set to 0b10.

A Half-target Branch record has BRBINF<n>_EL1.VALID set to 0b01.

A Full Branch record has BRBINF<n>_EL1.VALID set to 0b11.

BRBINF<n>_EL1.VALID indicates the validity of a Branch record:
• If BRBINF<n>_EL1.VALID is 0b00, the Branch record is invalid.
• Otherwise, the Branch record is valid.

When a Branch record is generated for any branch or exception which does not transition between a BRBE Prohibited region and a BRBE Non-prohibited region, the Branch record is a Full Branch record. See Branch records for exceptions on page D15-5504 and Branch records for exception returns on page D15-5505 for more details on when a Half-source Branch record or a Half-target Branch record is generated.

When an exception, exception return instruction, or Instruction Synchronization Barrier instruction causes a Context synchronization event which synchronizes an update to one or more System registers which are indirectly read when generating a Branch record, the synchronization of those register updates occurs before the registers are indirectly read. Such order is generally consistent with indirect reads of System registers performed by events which cause a Context synchronization event.

The reason for the Branch record is captured in BRBINF<n>_EL1.TYPE.

D15.1.2 Cycle counting

Each Branch record contains a cycle count value which is representative of the time taken between each Branch record being generated. The cycle count value can be used to determine the relative performance of the program between each Branch record. For large cycle count values, the value stored in each Branch record is encoded to use less storage, with a small loss of precision in the value.

The size of the cycle counter used to generate cycle count values is IMPLEMENTATION DEFINED, from one of the sizes indicated in BRBIDR0_EL1.CC.

Each Branch record contains a cycle count value which indicates the number of PE clock cycles that occurred between the previous Branch record being generated and this Branch record being generated.

In a multithreaded implementation, the cycle counter only counts cycles on which the thread was active.

The Branch record counts cycles in the same way as the CPU_CYCLES PMU event counts cycles when PMEVTYPE<n>_EL0.MT is 0. For more information, see Cycle event counting on page D11-5421.
For the purposes of the cycle count, a Branch record is generated only when the corresponding branch instruction or exception is guaranteed to be architecturally executed and the target address has been calculated. Arm recommends that the Branch record is generated as soon after this point as possible.

When a branch target address contains an address tag, the target address captured in the Branch record is the virtual address with the address tag removed.

The cycle count value in a Branch record is Branch cycle count unknown when any of the following are true:

- If EL2 is implemented, BRBCR_EL2.CC is 0.
- BRBCR_EL1.CC is 0.
- This is the first Branch record after the PE exited a BRBE Prohibited region.
- This is the first Branch record after cycle counting has been enabled.
- This is the first Branch record after BRBFCR_EL1.PAUSED is cleared from 1 to 0.
- This is the first Branch record after execution of a BRB IALL instruction.

Note: This applies even when EL2 is disabled in the current Security state.

When the cycle count value in a Branch record is Branch cycle count unknown:

- BRBINF<n>_EL1.CCU has the value 1.
- BRBINF<n>_EL1.CC contains a value which is all zeros.

The number of cycles indicated by this Branch record is UNKNOWN.

If the cycle count value in a Branch record would exceed the maximum value of the cycle counter, then:

- BRBINF<n>_EL1.CCU has the value 0.
- BRBINF<n>_EL1.CC contains a value which is all ones.

If the cycle count value in a Branch record is not UNKNOWN and would not exceed the maximum value of the cycle counter, then:

- BRBINF<n>_EL1.CCU has the value 0.
- BRBINF<n>_EL1.CC contains the cycle count value, encoded as defined in BRBINF<n>_EL1.CC.

### Mispredicated branches

Each Branch record generated for a branch instruction contains an indication of whether the branch was correctly or incorrectly predicted by the PE. Branch prediction behavior is IMPLEMENTATION DEFINED and this is an indication of whether such prediction succeeded, or not.

For a Branch record of a branch instruction, one of the following occurs:

- If EL2 is implemented and BRBCR_EL2.MPRED is 0, then BRBINF<n>_EL1.MPRED is set to 0.
- Else if BRBCR_EL1.MPRED is 0, then BRBINF<n>_EL1.MPRED is set to 0.
- Otherwise:
  - BRBINF<n>_EL1.MPRED is 0 for a correctly predicted branch.
  - BRBINF<n>_EL1.MPRED is 1 for an incorrectly predicted branch.

Note: This applies even when EL2 is disabled in the current Security state.

For a Branch record of an exception, BRBINF<n>_EL1.MPRED has the value 0.

An incorrectly predicted branch is when any of the following is true:

- The direction of a conditional branch was incorrectly predicted at least once during the execution of the instruction.
- The target of a branch was incorrectly predicted at least once during the execution of the instruction.
- The branch was not predicted by a branch predictor.

A correctly predicted branch is one that is not incorrectly predicted.
D15.1.4 BRBE Prohibited regions

An executable program might contain regions of code that are prohibited to generate Branch records, and these regions are called BRBE Prohibited regions. These regions are usually associated with a different Security state or Exception level.

BRBE Prohibited regions are controlled by the following:

- BRBCR_EL1.E0BRE.
- BRBCR_EL1.E1BRE.
- BRBCR_EL2.E0HBRE.
- BRBCR_EL2.E2BRE.
- MDCR_EL3.SBRBE.

While executing outside a BRBE Prohibited region, Branch records might not be generated because the Branch Record Buffer Extension has a number of filtering functions.

Execution in AArch32 state is a BRBE Prohibited region.

Execution in Debug state is a BRBE Prohibited region.

When FEAT_BRBEv1p1 and EL3 are implemented:
- When MDCR_EL3.{E3BREC, E3BREW} is {0b01, 0b10}, self-hosted EL3 branch recording is enabled.
- When MDCR_EL3.{E3BREC, E3BREW} is {0b00, 0b00} or MDCR_EL3.{E3BREC, E3BRE} is {0b11, 0b11}, self-hosted EL3 branch recording is disabled.

Execution at EL3 is a BRBE Prohibited region when any of the following are true:
- FEAT_BRBEv1p1 is not implemented.
- Self-hosted EL3 branch recording is disabled.

Execution at EL2 is a BRBE Prohibited region when any of the following are true:
- BRBCR_EL2.E2BRE is 0.
- MDCR_EL3.SBRBE is 0b00.
- MDCR_EL3.SBRBE is 0b01 and the PE is in Secure state.

Execution at EL1 is a BRBE Prohibited region when any of the following are true:
- BRBCR_EL1.E1BRE is 0.
- MDCR_EL3.SBRBE is 0b00.
- MDCR_EL3.SBRBE is 0b01 and the PE is in Secure state.

Execution at EL0 is a BRBE Prohibited region when any of the following are true:
- EL2 is disabled in the current Security state or HCR_EL2.TGE is 0, and BRBCR_EL1.E0BRE is 0.
- EL2 is enabled in the current Security state and HCR_EL2.TGE is 1, and BRBCR_EL2.E0HBRE is 0.
- MDCR_EL3.SBRBE is 0b00.
- MDCR_EL3.SBRBE is 0b01 and the PE is in Secure state.

While the PE is executing code from a BRBE Prohibited region, no data is captured in Branch records that might provide information about execution in the BRBE Prohibited region.

D15.1.5 Branch records for exceptions

When an exception is taken from a BRBE Prohibited region to a BRBE Prohibited region, no Branch record is generated.
When an exception is taken from a BRBE Non-prohibited region, or an exception is taken to a BRBE Non-prohibited region:

- If the target Exception level is EL1, a Branch record is generated only if BRBCR_EL1.EXCEPTION is 1.
- If the target Exception level is EL2, a Branch record is generated only if BRBCR_EL2.EXCEPTION is 1.
- If the target Exception level is EL3, a Branch record is generated only if FEAT_BRBEv1p1 is implemented and self-hosted EL3 branch recording is enabled.

When a Branch record is generated for an exception:

- If the exception is taken from a BRBE Prohibited region, then a Half-target Branch record is generated.
- If the exception is taken from a BRBE Non-prohibited region to a BRBE Prohibited region, then a Half-source Branch record is generated.
- If the exception is taken from a BRBE Non-prohibited region to a BRBE Non-prohibited region, then a Full Branch record is generated.

When entering Debug state:

- If the entry is from a BRBE Prohibited region, no Branch record is generated.
- If the entry is from a BRBE Non-prohibited region, then a Half-source Branch record is generated.

When a Half-source Branch record or a Full Branch record is generated for an Illegal Execution state exception, the source information in the Branch record indicates where the exception was taken from, in the same way as all other exceptions.

A Branch record for an exception which contains a valid source address has the source address set to the preferred exception return address for the exception.

A Branch record for an exception which contains a valid target address has the target address set to the address of the exception vector.

D15.1.6  Branch records for exception returns

When an exception return instruction is executed in a BRBE Prohibited region and branches to a BRBE Prohibited region, no Branch record is generated.

When an exception return instruction is executed in a BRBE Non-prohibited region, or an exception return instruction branches to a BRBE Non-prohibited region:

- If the exception return instruction is executed at EL3, a Branch record is generated only if FEAT_BRBEv1p1 is implemented and self-hosted EL3 branch recording is enabled.
- If the exception return instruction is executed at EL2, a Branch record is generated only if BRBCR_EL2.ERTN is 1.
- If the exception return instruction is executed at EL1, a Branch record is generated only if BRBCR_EL1.ERTN is 1.

When a Branch record is generated for an exception return instruction:

- If the exception return instruction is executed in a BRBE Prohibited region, then a Half-target Branch record is generated.
- If the exception return instruction is executed in a BRBE Non-prohibited region and branches to a BRBE Prohibited region, then a Half-source Branch record is generated.
- If the exception return instruction is executed in a BRBE Non-prohibited region and branches to a BRBE Non-prohibited region, then a Full Branch record is generated.

When exiting from Debug state:

- If the exit is to a BRBE Prohibited region, no Branch record is generated.
- If the exit is to a BRBE Non-prohibited region, then a Half-target Branch record is generated.
When a half-target branch record or a full branch record is generated for an exception return instruction which is an illegal return or a legal return which sets PSTATE.IL to 1, the target information in the branch record indicates the target of the branch:

- BRBTGT<\text{n}> EL1.ADDRESS contains the target of the branch.
- BRBINF<\text{n}> EL1.EL contains the value that is loaded into PSTATE.EL.

When a half-target branch record or a full branch record is generated for an exception return instruction which is an illegal return or a legal return which sets PSTATE.IL to 1, for the purposes of determining whether the target is a BRBE prohibited region, the value that is loaded into PSTATE.EL is used as the target exception level.

PSTATE.EL is unchanged on an illegal return, so the current exception level is the target of the illegal return, regardless of where the return was attempting to return to.

### D15.1.7 The Branch Record Buffer Extension and the Transactional Memory Extension

When an entire transaction is executed in a BRBE Non-prohibited region and the transaction fails or is canceled then BRBFCR_EL1.LASTFAILED is set to 1.

When an entire transaction is executed in a BRBE Prohibited region and the transaction fails or is canceled then BRBFCR_EL1.LASTFAILED is unchanged.

When a transaction is executed partially in a BRBE Prohibited region and partially in a BRBE Non-prohibited region and the transaction fails or is canceled then it is constrained unpredictable whether BRBFCR_EL1.LASTFAILED is set to 1 or is unchanged.

When a branch record is generated, other than through the injection mechanism, the value of BRBFCR_EL1.LASTFAILED is copied to the LASTFAILED field in the branch record and BRBFCR_EL1.LASTFAILED is set to 0.

When a transaction fails or is canceled, a branch record is not generated.

When a transaction fails or is canceled, branch records generated in the transaction are not removed from the branch record buffer.

Attempting to execute the BRB IALL or BRB INJ instructions in Transactional state results in the transaction failing with ERR cause.

### D15.1.8 PE speculation

The branch records only contain information for a branch, exception, or exception return that is architecturally executed.
D15.2 Branch record filtering

For Branch records generated outside a BRBE Prohibited region, it is useful to reduce the number of records that are generated to match their use. Table D15-1 on page D15-5507 lists some different use cases.

### Table D15-1 Example use cases for filtering

<table>
<thead>
<tr>
<th>Use case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control path</td>
<td>• All branches</td>
</tr>
<tr>
<td></td>
<td>• Subroutine returns</td>
</tr>
<tr>
<td></td>
<td>• Exceptions</td>
</tr>
<tr>
<td></td>
<td>• Exception returns</td>
</tr>
<tr>
<td>Call path</td>
<td>• Branch with link instructions</td>
</tr>
<tr>
<td></td>
<td>• Subroutine returns</td>
</tr>
<tr>
<td>Kernel calls</td>
<td>• Exceptions</td>
</tr>
<tr>
<td></td>
<td>• Exception returns</td>
</tr>
</tbody>
</table>

D15.2.1 Filtering on type

The Branch records can be filtered by independently enabling the generation of the following types:

- Exception.
- Exception return.
- Direct Branch with link.
- Indirect Branch with link.
- Return from subroutine.
- Indirect Branches.
- Conditional Direct Branches.
- Unconditional Direct Branches.

Control of when Branch records for exceptions are generated is controlled by BRBCR_EL1.EXCEPTION and BRBCR_EL2.EXCEPTION. See Branch records for exceptions on page D15-5504 for details.

### Table D15-2 Exception mapping for exceptions taken to AArch64 state

<table>
<thead>
<tr>
<th>Reason</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Target exception</td>
<td>Inst fault</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>Inst debug</td>
</tr>
<tr>
<td>FIQ</td>
<td>FIQ</td>
</tr>
<tr>
<td>HVC</td>
<td>Call</td>
</tr>
<tr>
<td>Halting debug event</td>
<td>Debug halt</td>
</tr>
<tr>
<td>IRQ</td>
<td>IRQ</td>
</tr>
<tr>
<td>Illegal execution state</td>
<td>Trap</td>
</tr>
<tr>
<td>Instruction Abort</td>
<td>Inst fault</td>
</tr>
<tr>
<td>Instruction or event trapped by a control bit</td>
<td>Trap</td>
</tr>
<tr>
<td>MemCopy or MemSet</td>
<td>Trap</td>
</tr>
<tr>
<td>Misaligned PC</td>
<td>Alignment</td>
</tr>
</tbody>
</table>
Control of when *Branch records* for exception return instructions are generated is controlled by BRBCR_EL1.ERTN and BRBSCR_EL2.ERTN. See *Branch records for exception returns* on page D15-5505 for details.

### Table D15-2 Exception mapping for exceptions taken to AArch64 state (continued)

<table>
<thead>
<tr>
<th>Reason</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAC Fail</td>
<td>Data fault</td>
</tr>
<tr>
<td>SError interrupt</td>
<td>System Error</td>
</tr>
<tr>
<td>SMC</td>
<td>Call</td>
</tr>
<tr>
<td>SVC</td>
<td>Call</td>
</tr>
<tr>
<td>Software Breakpoint Instruction</td>
<td>Inst debug</td>
</tr>
<tr>
<td>Software Step</td>
<td>Inst debug</td>
</tr>
<tr>
<td>Stack Pointer Misalignment</td>
<td>Alignment</td>
</tr>
<tr>
<td>Synchronous Data Abort</td>
<td>Data fault</td>
</tr>
<tr>
<td>UNDEFINED instruction</td>
<td>Trap</td>
</tr>
<tr>
<td>Watchpoint</td>
<td>Data debug</td>
</tr>
</tbody>
</table>

### Table D15-3 A64 return from exception instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERET</td>
<td>Return From Exception.</td>
</tr>
<tr>
<td>ERETTA</td>
<td>Authenticate and Exception return.</td>
</tr>
<tr>
<td>ERETTB</td>
<td>Authenticate and Exception return.</td>
</tr>
</tbody>
</table>

Each of the direct branch with link instructions only generates a *Branch record* when the instruction is executed in a BRBE Non-prohibited region and if any of the following are true:

- BRBFCR_EL1.DIRCALL is 1 and BRBFCR_EL1.EnI is 0.
- BRBFCR_EL1.DIRCALL is 0 and BRBFCR_EL1.EnI is 1.

### Table D15-4 A64 direct branch with link instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL</td>
<td>Branch with link.</td>
</tr>
</tbody>
</table>

Each of the indirect branch with link instructions only generates a *Branch record* when the instruction is executed in a BRBE Non-prohibited region and if any of the following are true:

- BRBFCR_EL1.INDCALL is 1 and BRBFCR_EL1.EnI is 0.
• BRBFCR_EL1.INDCALL is 0 and BRBFCR_EL1.EnI is 1.

Table D15-5 A64 indirect branch with link instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLR</td>
<td>Branch with link to register.</td>
</tr>
<tr>
<td>BLRAA</td>
<td>Authenticate and branch with link.</td>
</tr>
<tr>
<td>BLRAAZ</td>
<td>Authenticate and branch with link.</td>
</tr>
<tr>
<td>BLRAB</td>
<td>Authenticate and branch with link.</td>
</tr>
<tr>
<td>BLRABZ</td>
<td>Authenticate and branch with link.</td>
</tr>
</tbody>
</table>

Each of the return from subroutine instructions only generates a Branch record, when the instruction is executed in a BRBE Non-prohibited region and if any of the following are true:
• BRBFCR_EL1.RTN is 1 and BRBFCR_EL1.EnI is 0.
• BRBFCR_EL1.RTN is 0 and BRBFCR_EL1.EnI is 1.

Table D15-6 A64 return from subroutine instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RET</td>
<td>Return From subroutine.</td>
</tr>
<tr>
<td>RETAA</td>
<td>Authenticate and function return.</td>
</tr>
<tr>
<td>RETAB</td>
<td>Authenticate and function return.</td>
</tr>
</tbody>
</table>

Unless covered by other rules, each of the indirect branch instructions only generates a Branch record when the instruction is executed in a BRBE Non-prohibited region and if any of the following are true:
• BRBFCR_EL1.INDIRECT is 1 and BRBFCR_EL1.EnI is 0.
• BRBFCR_EL1.INDIRECT is 0 and BRBFCR_EL1.EnI is 1.

Table D15-7 A64 indirect branch instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR</td>
<td>Branch to register.</td>
</tr>
<tr>
<td>BRAA</td>
<td>Authenticate and branch.</td>
</tr>
<tr>
<td>BRAAZ</td>
<td>Authenticate and branch.</td>
</tr>
<tr>
<td>BRAB</td>
<td>Authenticate and branch.</td>
</tr>
<tr>
<td>BRABZ</td>
<td>Authenticate and branch.</td>
</tr>
</tbody>
</table>

Unless covered by other rules, each of the conditional direct branch instructions only generates a Branch record when the instruction is taken, is executed in a BRBE Non-prohibited region, and if any of the following are true:
• BRBFCR_EL1.CONDDIR is 1 and BRBFCR_EL1.EnI is 0.
• BRBFCR_EL1.CONDDIR is 0 and BRBFCR_EL1.EnI is 1.

**Table D15-8 A64 conditional direct branch instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.cond</td>
<td>Conditional Branch.</td>
</tr>
<tr>
<td>BC.cond</td>
<td>Branch Consistent conditionally.</td>
</tr>
<tr>
<td>CBZ or CBNZ</td>
<td>Compare with zero and branch.</td>
</tr>
<tr>
<td>TBZ or TBNZ</td>
<td>Test and branch.</td>
</tr>
</tbody>
</table>

Note: BC.cond and B.cond instructions with the AL or NW condition code are considered conditional.

Unless covered by other rules, each of the unconditional direct branch instructions only generates a *Branch record* when the instructions are executed in a BRBE Non-prohibited region and if any of the following are true:

• BRBFCR_EL1.DIRECT is 1 and BRBFCR_EL1.EnI is 0.
• BRBFCR_EL1.DIRECT is 0 and BRBFCR_EL1.EnI is 1.

Table D15-9 A64 unconditional direct branch instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Unconditional Branch.</td>
</tr>
</tbody>
</table>

It is IMPLEMENTATION DEFINED whether *Branch records* are generated for each of the following taken unconditional direct branch instructions when the instruction is executed in a BRBE Non-prohibited region and if any of the following are true:

• BRBFCR_EL1.DIRECT is 1 and BRBFCR_EL1.EnI is 0.
• BRBFCR_EL1.DIRECT is 0 and BRBFCR_EL1.EnI is 1.

Table D15-10 Optional A64 unconditional direct branch instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISB</td>
<td>Instruction Synchronization Barrier.</td>
</tr>
</tbody>
</table>

Writing a value of 0b0000_0001 to the filter controls, BRBFCR_EL1<23:16>, ensures *Branch records* are generated for all branch instructions.
D15.3 Branch record buffer operation

The Branch Record Buffer Extension operation is controlled by the BRBCR_EL1, BRBCR_EL2, and BRBFCR_EL1 registers.

Generation of Branch records is paused when BRBFCR_EL1.PAUSED is 1.

When generation of Branch records is paused, Branch records are not generated.

If EL2 is implemented, a BRBE freeze event occurs when all of the following are true:
- BRBCR_EL1.FZP is 1.
- Generation of Branch records is not paused.
- PMOVSLR_EL0[(MDCR_EL2.HPMN-1):0] is nonzero.
- The PE is in a BRBE Non-prohibited region.

If EL2 is implemented, a BRBE freeze event occurs when all of the following are true:
- BRBCR_EL2.FZP is 1.
- Generation of Branch records is not paused.
- PMOVSLR_EL0[(PMCR_EL0.N-1):MDCR_EL2.HPMN] is nonzero.
- The PE is in a BRBE Non-prohibited region.

This applies even when EL2 is disabled in the current Security state.

If EL2 is not implemented, a BRBE freeze event occurs when all of the following are true:
- BRBCR_EL1.FZP is 1.
- Generation of Branch records is not paused.
- PMOVSLR_EL0[(PMCR_EL0.N-1):0] is nonzero.
- The PE is in a BRBE Non-prohibited region.

On a BRBE freeze event:
- BRBFCR_EL1.PAUSED is set to 1.
- The current timestamp is captured in BRBTS_EL1.

The source of value of the timestamp captured in BRBTS_EL1 is selected by the combination of programming of BRBCR_EL2.TS and BRBCR_EL1.TS. See Table D15-11 on page D15-5511 and BRBETimeStamp().

<table>
<thead>
<tr>
<th>BRBCR_EL2.TS</th>
<th>BRBCR_EL1.TS</th>
<th>Captured timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00 (delegate)</td>
<td>0b01 (virtual)</td>
<td>PhysicalCountInt() - CNTVOFF_EL2</td>
</tr>
<tr>
<td>0b10 (offset physical)</td>
<td>0b11 (physical)</td>
<td>PhysicalCountInt() - physical offset</td>
</tr>
<tr>
<td>0b01 (virtual)</td>
<td>0bxx</td>
<td>PhysicalCountInt() - CNTVOFF_EL2</td>
</tr>
<tr>
<td>0b10 (offset physical)</td>
<td>0bxx</td>
<td>PhysicalCountInt() - physical offset</td>
</tr>
<tr>
<td>0b11 (physical)</td>
<td>0bxx</td>
<td>PhysicalCountInt()</td>
</tr>
</tbody>
</table>

If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2:
- EL3 is implemented and SCR_EL3.ECVEn is 0.
- EL2 is implemented and CNTHCTL_EL2.ECV is 0.

When a valid Branch record is captured in the Branch record buffer storage, the BRB_FILTRATE event is generated.
When BRB_FILTRATE is generated for an exception or an exception return, it is an Exception-related event. For more information on PMU event filtering, see Exception-related events on page D11-5260.

It is CONstrained UNpredictable whether a BRB_FILTRATE event is generated after a BRB_INJ causes a Branch record to be injected.

It is expected that the Branch record buffer storage is in a BRBE Prohibited region during software context switches. During software context switches, if the PMU event counters are not prohibited from counting events, it is expected that event counters set to count the BRB_FILTRATE event are prohibited from counting.

The architecture does not define when PMU events are counted relative to the instructions that caused the event. Events generated by an instruction might be counted before or after the instruction becomes architecturally executed, and events might be counted for operations that do not become architecturally executed. This means that events can be counted speculatively and/or out-of-order regarding the simple sequential execution of the program. Events might also be counted simultaneously by other event counters when the overflow occurs, including events from different instructions. In addition, multiple instances of an event might occur simultaneously, meaning that an event counter unsigned overflow can yield a nonzero value in the event counter.

Furthermore, the Branch records are generated only for architecturally executed operations. See RKXTKS:

These properties mean that, unless otherwise stated, on a BRBE freeze event, it is CONstrained UNpredictable whether the branches that define the basic block containing the instruction causing that event are captured in the Branch record buffer.

An exception to this relaxation applies for the BRB_FILTRATE event.

If a direct read of PMOVSLR_EL0 returns a nonzero value for a subset of the overflow flags, such that one of RNXCWF, RGXGWY, or RPKTXQ means that a BRBE freeze event should occur, then a direct read of BRBFDR_EL1 ordered after the direct read of PMOVSLR_EL0 will return BRBFDR_EL1.PAUSED is 1.

Direct reads of System registers require explicit synchronization for following direct reads of other System registers to be ordered after the first direct read. For more information, see General behavior of accesses to the AArch64 System registers on page D17-5546.

If a direct read of BRBFDR_EL1.PAUSED returns 1, then no operations ordered after the direct read will generate further Branch records until BRBFDR_EL1.PAUSED is cleared by software.

Note: The subsequent operations can be ordered by a Context synchronization event.
D15.4 Branch record buffer

IFBHCC The Branch record buffer can contain:

- 8 Branch records.
- 16 Branch records.
- 32 Branch records.
- 64 Branch records.

This is known as the Branch record buffer storage.

RKSLSM The Branch record buffer storage has a maximum number of Branch records as defined by BRBIDR0_EL1.NUMREC.

IPPBZP The Branch record buffer provides System registers to access the Branch records stored in the Branch record buffer storage. These System registers provide access to up to 32 Branch records without the need for explicit synchronization between each System register read. When more than 32 Branch records are implemented, the Branch record buffer provides a banking mechanism to provide access to multiple banks, each bank containing up to 32 Branch records. BRBFCR_EL1.BANK controls which bank is currently selected, and updates to BRBFCR_EL1.BANK require explicit synchronization before accessing the bank.

WLSQL The Branch record buffer storage has a capacity to store up to 32 Branch records. Accessing Branch records 0-31 is performed by setting BRBFCR_EL1.BANK to 0b0. Accessing Branch records 32-63 is performed by setting BRBFCR_EL1.BANK to 0b1.

TJGLK The Branch record with index 0 is the youngest captured branch.

HRTRNR The Branch record with index n is younger than Branch record with index n+1.

DTPDK On the generation of a new Branch record, if the Branch record buffer storage is full, then the oldest Branch record is lost.

SQLCX When the buffer contains M valid Branch records, where M > 0 and M is less than the maximum number of Branch records, all of the following are true:

- Branch records with index 0 to M-1 are all valid.
- All other Branch records are invalid.

PGDLX The creation of a Branch record is considered an indirect write to BRBTGT<n>_EL1, BRBSRC<n>_EL1 and BRBINF<n>_EL1, and therefore requires explicit synchronization before being read.

SFFNF The generation of Branch records performs indirect reads and indirect writes of System registers.

KFTYTV Synchronization requirements for AArch64 System registers on page D17-5547 defines the synchronization requirements for direct reads, direct writes, indirect reads, and indirect writes of System registers made by instructions and external agents.

D15.4.1 Invalidating the record buffer

LLIHYN Execution of BRR IALL causes all Branch records to be invalidated.

PFERNW A Branch record, R, is invalidated by the instruction BRR IALL, W, if all of the following are true:

- R is caused by a branch operation or exception, B.
- B is either:
  - In program order before a Context synchronization event, CSE.
  - Is the Context synchronization event.
- CSE is in program order before W.

LWPKR A Branch record R is not invalidated by the instruction BRR IALL, W, if all of the following are true:

- R is caused by a branch operation or exception, B.
- B is in program order after a Context synchronization event, CSE
- CSE is in program order after W.
It is constrained unpredictable whether a Branch record $R$ is invalidated by the instruction BRB IALL, $W$, if all of the following are true:

- $CSE_1$ is in program order before $W$.
- $R$ is caused by a branch operation or exception, $B$.
- $B$ is in program order after a Context synchronization event, $CSE_1$
- $B$ is either:
  - In program order before a Context synchronization event, $CSE_2$.
  - Is the Context synchronization event, $CSE_2$.
- $CSE_2$ is in program order after $W$ and there are no other CSEs between $CSE_1$ and $CSE_2$.

If a Branch record is invalidated, all older Branch records are invalidated.

When a Branch record has been invalidated, it remains invalid until it is overwritten by any of the following:

- A new Branch record is created.
- A Branch record is injected using the BRB INJ instruction.
D15.5 Programmers’ model

RBNGTH

Reads from an unimplemented Branch record return the value zero.

RPKZCF

All Branch records captured while generation of Branch records is not paused, must represent a continuous block of execution for all BRBE Non-prohibited regions.

IXSBPR

The captured Branch records might not represent a continuous block if generation of Branch records is paused at any time. To avoid this non-continuous nature, the BRB IALL instruction can be used to invalidate all Branch records while generation is paused.

RPMBRRL

If a Branch record cannot be captured for a branch instruction or exception that is not prohibited and has been selected to generate a record, then all the Branch records must be invalidated. The reasons for a PE being unable to capture a Branch record are IMPLEMENTATION DEFINED and Arm recommends that such reasons are rare.

IQJFSV

When a process is migrated to a PE with a smaller number of Branch records implemented then the information from the older Branch records will be lost.

IBDKJ

When FEAT_BRBE is implemented, the following fields are added to System registers to control access to the Branch record buffer functionality:

- When EL2 is implemented:
  - HDFGRTR_EL2.nBRBIDR.
  - HDFGRTR_EL2.nBRBCTL.
  - HDFGWTR_EL2.nBRBCTL.
  - HDFGRTR_EL2.nBRBDATA.
  - HDFGWTR_EL2.nBRBDATA.
  - HFGITR_EL2.nBRBINJ.
  - HFGITR_EL2.nBRBIALL.
- When EL3 is implemented:
  - MDCR_EL3.SBRBE.

SGCYTK

When self-hosted EL3 branch recording is enabled, the following registers must be programmed:

- BRBCR_EL1 and BRBCR_EL2. Software must program these registers to control the following:
  - Recording of exceptions taken to EL1 and EL2.
  - Recording of exception returns from EL1 and EL2.
  - Branch recording at EL0, EL1, and EL2.
  - Occurrence of BRBE freeze events on PMU overflows.
  - Timestamp source.
  - Misprediction information.
  - Cycle count information.
- BRBFCSR_EL1. Software must program this register to control the following:
  - Selection of Branch record buffer bank.
  - Selection of branch types to record at all Exception levels.
  - Pausing of Branch recording.
  - If FEAT_TME is implemented, recording of transaction failure information.

Software should also consider how MDCR_EL3.SBRBE should be programmed to either allow or prohibit access to the captured branch records from lower Exception levels. Even if self-hosted EL3 branch recording is not being used, software should consider whether a BRB IALL instruction should be executed before executing software at lower Exception levels.

SYLMQQ

Software must invalidate the Branch records after a PE reset to ensure that details of execution before the reset event are not leaked.

IPJPPQ

MDCR_EL3.E3BREC resets to 0 on a Cold reset, and MDCR_EL3.E3BREW resets to 0 on a Warm reset. This allows software to program MDCR_EL3.E3BREC and MDCR_EL3.E3BREW such that the BRBE continues recording after Warm reset, or stops recording at Warm reset.
When FEAT_BRBEv1p1 is implemented, certain fields in the following registers are reset to an architecturally unknown value on a Cold reset and unchanged on a Warm reset:

- BRBCR_EL1.
- BRBCR_EL2.
- BRBFCCR_EL1.

### D15.5.1 Manual injection of Branch records

The Branch Record Buffer Extension supports the ability to manually create *Branch records* and inject them in to the Branch record buffer storage. The primary purpose of the injection functionality is to support the restore of the Branch record buffer storage contents, particularly during software context switch events, including migration of software between PEs. The Branch record buffer storage contents are read out using direct reads of BRBSRC<\(n\)>_EL1, BRBTGT<\(n\)>_EL1, and BRBINF<\(n\)>_EL1.

The Branch Record Injection data registers are:

- BRBSRCINJ_EL1.
- BRBTGTINJ_EL1.
- BRBINFINJ_EL1.

*Branch record* injection consists of creating a single *Branch record* using direct writes to the Branch Record Injection data registers, then injecting the record into the Branch record buffer storage using BRB INJ. This process injects a single *Branch record* as the youngest entry in the Branch record buffer storage. This process is repeated for each *Branch record* to be added to the Branch record buffer storage.

When BRB INJ is executed outside of a BRBE Prohibited region, it is CONSTRAINED UNPREDICTABLE whether a *Branch record* is injected.

When BRB INJ is executed inside a BRBE Prohibited region, the contents of the Branch Record Injection data registers are used to create a *Branch record* which is added to the Branch record buffer storage as the youngest entry.

When a BRB INJ instruction is executed inside a BRBE Prohibited region and the contents of the Branch Record Injection data registers indicates an invalid record, it is CONSTRAINED UNPREDICTABLE whether a *Branch record* is injected to the Branch record buffer. An invalid record is one with BRBINFINJ_EL1.VALID set to 0b00.

For a BRB INJ instruction, it is CONSTRAINED UNPREDICTABLE whether a *Branch record* is injected to the Branch record buffer when all of the following are true:

- The BRB INJ instruction is executed inside a BRBE Prohibited region.
- The contents of the Branch Record Injection data registers indicates a valid record.
- The other contents of the Branch Record Injection data registers indicate an incorrectly formatted record.

If a *Branch record* is injected, then the contents of the Branch record are CONSTRAINED UNPREDICTABLE.

An example of an incorrectly formatted record is one where BRBINFINJ_EL1.VALID is 0b01 and BRBINFINJ_EL1.MPRED is 0b1.

Execution of BRB INJ does not require explicit synchronization to use the result of direct writes to the Branch Record Injection data registers in program order before BRB INJ.

The creation of a *Branch record* as a result of execution of BRB INJ does not use the result of direct writes to the Branch Record Injection data registers in program order after BRB INJ. Explicit synchronization is not required to ensure this ordering. For more information, see *Synchronization requirements for AArch64 System registers* on page D17-5547.

After the execution of BRB INJ, the contents of the Branch Record Injection data registers are UNKNOWN.

Changes to the BRB registers are subject to the rules for synchronization for System registers. See *Synchronization requirements for AArch64 System registers* on page D17-5547.
Chapter D16
AArch64 System Register Encoding

This chapter describes the AArch64 System register encoding space. It contains the following sections:

- *The System register encoding space* on page D16-5518.
- *op0*=0b10, Moves to and from debug and trace System registers* on page D16-5519.
- *op0*=0b11, Moves to and from non-debug System registers, Special-purpose registers* on page D16-5523.
D16.1 The System register encoding space

The A64 instruction set includes instructions that access the System register encoding space. These instructions provide:

- Access to System registers, including the debug registers, that provide system control, and system status information.
- Access to Special-purpose registers such as SPSR_ELx, ELR_ELx, and the equivalent fields of the Process State.
- The cache and TLB maintenance instructions and address translation instructions.
- Barriers and the CLREX instruction.
- Architectural hint instructions.

This section describes the parts of the System register encoding space that provides access to the System registers described in Chapter D17 AArch64 System Register Descriptions.

--- Note ---

- See Fixed values in AArch64 instruction and System register descriptions on page C2-243 for information about abbreviations used in the System instruction descriptions.

- In AArch32 state, much of this functionality is provided through the System register interface described in The AArch32 System register interface on page G1-8993. In AArch64 state, the parameters used to characterize the System register encoding space are \{\text{op0}}, \text{op1}, \text{CRn}, \text{CRm}, \text{op2}\). These are based on the parameters that characterize the AArch32 System register encoding space, which reflect the original implementation of these registers, as described in Background to the System register interface on page G1-8994. There is no particular significance to the naming of these parameters, and no functional distinction between the \text{opn} parameters and the \text{CRx} parameters.

---

Principles of the System instruction class encoding on page C5-654 describes some general properties of these encodings. System instruction class encoding overview on page C5-655 then describes the top-level encoding of these instructions, identifying that:

- Entries in the encoding space are characterized by the parameter set \{\text{op0}}, \text{op1}, \text{CRn}, \text{CRm}, \text{op2}\}.
- \text{op0} is the most significant parameter for determining allocations in this space.

Much of this encoding space is used for System instructions, as described in Chapter C5 The A64 System Instruction Class. This chapter describes only the part of the encoding space that is used for System registers, in the following sections:

- \text{op0}==0b10, Moves to and from debug and trace System registers on page D16-5519.
- \text{op0}==0b11, Moves to and from non-debug System registers, Special-purpose registers on page D16-5523.
D16.2  \( \text{op0} = 0b10 \), Moves to and from debug and trace System registers

The instructions that move data to and from the debug, Execution environment, and trace System registers are encoded with \( \text{op0} = 0b10 \). This means the encoding of these instructions is:

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 16 15</th>
<th>8 7 5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 1 0 0</td>
<td>L 1 0</td>
<td>op1</td>
<td>CRn</td>
<td>CRm</td>
<td>op2</td>
</tr>
</tbody>
</table>

\( \text{op0} \)

--- Note ---

- The section describes the use of all of the \( \text{op0} = 0b10 \) region of the System register encoding space.
- These encodings access the registers that are equivalent to the AArch32 System registers in the \( \text{coproc} = 0b1110 \) encoding space.

The value of \( \text{op1} \) provides the next level of decode of these instructions, as follows:

\( \text{op1} = \{0, 3, 4\} \)

Debug. See Instructions for accessing debug System registers

--- Note ---

The standard encoding of debug registers is \( \text{op0} = 0b10, \text{op1} = \{0, 3, 4\} \). The registers in the \( \text{op0} = 0b11 \) encoding space that are classified as debug registers are DLR_EL0, DSPSR_EL0, MDCR_EL2, MDCR_EL3, and SDER32_EL3. See Instructions for accessing non-debug System registers on page D16-5523 for the encodings of these registers.

\( \text{op1} = 1 \)

Trace and BRBE registers. See FEAT_ETE, FEAT_BRBE, or the appropriate trace architecture specification for more information.

D16.2.1 Instructions for accessing debug System registers

The instructions for accessing debug System registers are:

- \text{MSR <System register>, Xt} ; Write to System register
- \text{MRS Xt, <System register>} ; Read from System register

Where \(<\text{system register}>\) is the register name, for example MDCCSR_EL0.

This section includes only the System register access encodings for which both:
- \( \text{op0} \) is \( 0b10 \).
- The value of \( \text{op1} \) is one of \( \{0, 1, 3, 4, 5\} \).

--- Note ---

These encodings access the registers that are equivalent to the AArch32 System registers in the \( \text{coproc} = 0b1110 \) encoding space.

Table D16-1 shows the mapping of the System register encodings for debug System register access.

<table>
<thead>
<tr>
<th>( \text{op0} )</th>
<th>( \text{op1} )</th>
<th>CRn</th>
<th>CRm</th>
<th>( \text{op2} )</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>OSDTRRX_EL1</td>
<td>OSDTRRX_EL1</td>
</tr>
<tr>
<td>10</td>
<td>000</td>
<td>0000</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>MDCCINT_EL1</td>
<td>MDCCINT_EL1</td>
</tr>
<tr>
<td>10</td>
<td>000</td>
<td>0000</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>MDSCR_EL1</td>
<td>MDSCR_EL1</td>
</tr>
<tr>
<td>10</td>
<td>000</td>
<td>0000</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>OSDTRTX_EL1</td>
<td>OSDTRTX_EL1</td>
</tr>
</tbody>
</table>
### Table D16-1 Instruction encodings for debug System register access (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>000</td>
<td>0000</td>
<td>0110</td>
<td>010</td>
<td>RW</td>
<td>OSECCR_EL1</td>
<td>OSECCR_EL1</td>
</tr>
<tr>
<td>10</td>
<td>000</td>
<td>0000</td>
<td>m[3:0]</td>
<td>100</td>
<td>RW</td>
<td>DBGBVR&lt;n&gt;_EL1</td>
<td>DBGBVR_EL1[]</td>
</tr>
<tr>
<td>10</td>
<td>000</td>
<td>0000</td>
<td>m[3:0]</td>
<td>101</td>
<td>RW</td>
<td>DBGBCR&lt;n&gt;_EL1</td>
<td>DBGBCR_EL1[]</td>
</tr>
<tr>
<td>10</td>
<td>000</td>
<td>0000</td>
<td>m[3:0]</td>
<td>110</td>
<td>RW</td>
<td>DBGWVR&lt;n&gt;_EL1</td>
<td>DBGWVR_EL1[]</td>
</tr>
<tr>
<td>10</td>
<td>000</td>
<td>0000</td>
<td>m[3:0]</td>
<td>111</td>
<td>RW</td>
<td>DBGWCR&lt;n&gt;_EL1</td>
<td>DBGWCR_EL1[]</td>
</tr>
<tr>
<td>10</td>
<td>000</td>
<td>0001</td>
<td>0000</td>
<td>000</td>
<td>RO</td>
<td>MDRAR_EL1</td>
<td>MDRAR_EL1</td>
</tr>
<tr>
<td>10</td>
<td>000</td>
<td>0001</td>
<td>0000</td>
<td>100</td>
<td>WO</td>
<td>OSLAR_EL1</td>
<td>OSLAR_EL1</td>
</tr>
<tr>
<td>10</td>
<td>000</td>
<td>0001</td>
<td>0011</td>
<td>100</td>
<td>RO</td>
<td>OSLSR_EL1</td>
<td>OSLSR_EL1</td>
</tr>
<tr>
<td>10</td>
<td>000</td>
<td>0001</td>
<td>0011</td>
<td>110</td>
<td>RO</td>
<td>OSDLR_EL1</td>
<td>OSDLR_EL1</td>
</tr>
<tr>
<td>10</td>
<td>000</td>
<td>0001</td>
<td>0100</td>
<td>100</td>
<td>RW</td>
<td>DBGPRCR_EL1</td>
<td>DBGPRCR_EL1</td>
</tr>
<tr>
<td>10</td>
<td>000</td>
<td>0111</td>
<td>1000</td>
<td>110</td>
<td>RW</td>
<td>DBGCLAIMSET_EL1</td>
<td>DBGCLAIMSET_EL1</td>
</tr>
<tr>
<td>10</td>
<td>000</td>
<td>0111</td>
<td>1001</td>
<td>110</td>
<td>RW</td>
<td>DBGCLAIMCLR_EL1</td>
<td>DBGCLAIMCLR_EL1</td>
</tr>
<tr>
<td>10</td>
<td>000</td>
<td>0111</td>
<td>1110</td>
<td>110</td>
<td>RO</td>
<td>DBGAUTHSTATUS_EL1</td>
<td>DBGAUTHSTATUS_EL1</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>TRCTRACEIDR</td>
<td>TRCTRACEIDR</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>TRCVICTLR</td>
<td>TRCVICTLR</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0000</td>
<td>110</td>
<td>RO</td>
<td>TRCIDR8</td>
<td>TRCIDR8</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0000</td>
<td>111</td>
<td>RW</td>
<td>TRCIMSPEC0</td>
<td>TRCIMSPEC0</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0001</td>
<td>110</td>
<td>RO</td>
<td>TRCPRGCTRLR</td>
<td>TRCPRGCTRLR</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0001</td>
<td>111</td>
<td>RO</td>
<td>TRCQCTRLR</td>
<td>TRCQCTRLR</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>TRCVIIECTLR</td>
<td>TRCVIIECTLR</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0011</td>
<td>110</td>
<td>RO</td>
<td>TRCIDR9</td>
<td>TRCIDR9</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0010</td>
<td>100</td>
<td>RW</td>
<td>TRCVISSCTRLR</td>
<td>TRCVISSCTRLR</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0011</td>
<td>110</td>
<td>RO</td>
<td>TRCIDR10</td>
<td>TRCIDR10</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0110</td>
<td>010</td>
<td>RW</td>
<td>TRCSTATTRO</td>
<td>TRCSTATTRO</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0111</td>
<td>010</td>
<td>RW</td>
<td>TRCVIPCSSCTRLR</td>
<td>TRCVIPCSSCTRLR</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0111</td>
<td>110</td>
<td>RO</td>
<td>TRCIDR11</td>
<td>TRCIDR11</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>00:m[1:0]</td>
<td>100</td>
<td>RW</td>
<td>TRCSEQEVR&lt;n&gt;</td>
<td>TRCSEQEVR[]</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>00:m[1:0]</td>
<td>101</td>
<td>RW</td>
<td>TRCCNTRLDVR&lt;n&gt;</td>
<td>TRCCNTRLDVR[]</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0100</td>
<td>000</td>
<td>RW</td>
<td>TRCCONFIGR</td>
<td>TRCCONFIGR</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0100</td>
<td>110</td>
<td>RO</td>
<td>TRCIDR12</td>
<td>TRCIDR12</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0101</td>
<td>110</td>
<td>RO</td>
<td>TRCIDR13</td>
<td>TRCIDR13</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0110</td>
<td>000</td>
<td>RW</td>
<td>TRCAUXCTRLR</td>
<td>TRCAUXCTRLR</td>
</tr>
</tbody>
</table>
Table D16-1 Instruction encodings for debug System register access (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0110</td>
<td>100</td>
<td>RW</td>
<td>TRCSQESTEVR</td>
<td>TRCSQESTEVR</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0111</td>
<td>100</td>
<td>RW</td>
<td>TRCSEQSTR</td>
<td>TRCSEQSTR</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>01:m[1:0]</td>
<td>101</td>
<td>RW</td>
<td>TRCCNTCTLR&lt;n&gt;</td>
<td>TRCCNTCTLR[]</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>0:m[2:0]</td>
<td>111</td>
<td>RW</td>
<td>TRCIMSPEC&lt;n&gt;</td>
<td>TRCIMSPEC[]</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>1000</td>
<td>000</td>
<td>RW</td>
<td>TRCEVENTCTL0R</td>
<td>TRCEVENTCTL0R</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>1000</td>
<td>111</td>
<td>RO</td>
<td>TRCIDR0</td>
<td>TRCIDR0</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>1001</td>
<td>000</td>
<td>RW</td>
<td>TRCEVENTCTL1R</td>
<td>TRCEVENTCTL1R</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>1001</td>
<td>111</td>
<td>RO</td>
<td>TRCIDR1</td>
<td>TRCIDR1</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>1010</td>
<td>000</td>
<td>RW</td>
<td>TRCRSR</td>
<td>TRCRSR</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>1010</td>
<td>111</td>
<td>RO</td>
<td>TRCIDR2</td>
<td>TRCIDR2</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>1011</td>
<td>000</td>
<td>RW</td>
<td>TRCSTALLCTRLR</td>
<td>TRCSTALLCTRLR</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>1011</td>
<td>111</td>
<td>RO</td>
<td>TRCIDR3</td>
<td>TRCIDR3</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>10:m[1:0]</td>
<td>100</td>
<td>RW</td>
<td>TRCEXITINSELT&lt;n&gt;</td>
<td>TRCEXITINSELT[]</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>10:m[1:0]</td>
<td>101</td>
<td>RW</td>
<td>TRCCNTVR&lt;n&gt;</td>
<td>TRCCNTVR[]</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>1100</td>
<td>000</td>
<td>RW</td>
<td>TRCTSCCTRL</td>
<td>TRCTSCCTRL</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>1100</td>
<td>111</td>
<td>RO</td>
<td>TRCIDR4</td>
<td>TRCIDR4</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>1101</td>
<td>000</td>
<td>RW</td>
<td>TRCSYNCPR</td>
<td>TRCSYNCPR</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>1101</td>
<td>111</td>
<td>RO</td>
<td>TRCIDR5</td>
<td>TRCIDR5</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>1110</td>
<td>000</td>
<td>RW</td>
<td>TRCCCCTRLR</td>
<td>TRCCCCTRLR</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>1110</td>
<td>111</td>
<td>RO</td>
<td>TRCIDR6</td>
<td>TRCIDR6</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>1111</td>
<td>000</td>
<td>RW</td>
<td>TRCBBCTRL</td>
<td>TRCBBCTRL</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0000</td>
<td>1111</td>
<td>111</td>
<td>RO</td>
<td>TRCIDR7</td>
<td>TRCIDR7</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0001</td>
<td>0001</td>
<td>100</td>
<td>RO</td>
<td>TRCOSLSR</td>
<td>TRCOSLSR</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0001</td>
<td>0:m[2:0]</td>
<td>010</td>
<td>RW</td>
<td>TRCSSCCR&lt;n&gt;</td>
<td>TRCSSCCR[]</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0001</td>
<td>0:m[2:0]</td>
<td>011</td>
<td>RW</td>
<td>TRCSSPCICR&lt;n&gt;</td>
<td>TRCSSPCICR[]</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0001</td>
<td>1:m[2:0]</td>
<td>010</td>
<td>RW</td>
<td>TRCSSCSR&lt;n&gt;</td>
<td>TRCSSCSR[]</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0001</td>
<td>m[3:0]</td>
<td>00:m[4]</td>
<td>RW</td>
<td>TRCRSCTRL&lt;n&gt;</td>
<td>TRCRSCTRL[]</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0010</td>
<td>m[2:0]:0</td>
<td>00:m[3]</td>
<td>RW</td>
<td>TRCACVR&lt;n&gt;</td>
<td>TRCACVR[]</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0010</td>
<td>m[2:0]:0</td>
<td>01:m[3]</td>
<td>RW</td>
<td>TRACATR&lt;n&gt;</td>
<td>TRACATR[]</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0011</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>TRCCIDCTRL0</td>
<td>TRCCIDCTRL0</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0011</td>
<td>0001</td>
<td>010</td>
<td>RW</td>
<td>TRCCIDCTRL1</td>
<td>TRCCIDCTRL1</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0011</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>TRCVMIDCTRL0</td>
<td>TRCVMIDCTRL0</td>
</tr>
</tbody>
</table>
### Table D16-1 Instruction encodings for debug System register access (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>001</td>
<td>0011</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>TRCVMIDCCTRL1</td>
<td>TRCVMIDCCTRL1</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0011</td>
<td>m[2:0]:0</td>
<td>000</td>
<td>RW</td>
<td>TRCCIDCVR&lt;α&gt;</td>
<td>TRCCIDCVR[α]</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0011</td>
<td>m[2:0]:0</td>
<td>001</td>
<td>RW</td>
<td>TRCVMIDCVR&lt;α&gt;</td>
<td>TRCVMIDCVR[α]</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0111</td>
<td>0010</td>
<td>111</td>
<td>RO</td>
<td>TRCDEVID</td>
<td>TRCDEVID</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0111</td>
<td>1000</td>
<td>110</td>
<td>RW</td>
<td>TRCCLAIMSET</td>
<td>TRCCLAIMSET</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0111</td>
<td>1001</td>
<td>110</td>
<td>RW</td>
<td>TRCCLAIMCLR</td>
<td>TRCCLAIMCLR</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0111</td>
<td>1110</td>
<td>110</td>
<td>RO</td>
<td>TRCAUTHSTATUS</td>
<td>TRCAUTHSTATUS</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>0111</td>
<td>1111</td>
<td>110</td>
<td>RO</td>
<td>TRCDEVARCH</td>
<td>TRCDEVARCH</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>1000</td>
<td>m[3:0]</td>
<td>m[4]:00</td>
<td>RO</td>
<td>BRBINF&lt;α&gt;EL1</td>
<td>BRBINF&lt;α&gt;EL1</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>1000</td>
<td>m[3:0]</td>
<td>m[4]:01</td>
<td>RO</td>
<td>BRBSRC&lt;α&gt;EL1</td>
<td>BRBSRC&lt;α&gt;EL1</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>1000</td>
<td>m[3:0]</td>
<td>m[4]:10</td>
<td>RO</td>
<td>BRBTGT&lt;α&gt;EL1</td>
<td>BRBTGT&lt;α&gt;EL1</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>1001</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>BRBCR_EL1</td>
<td>BRBCR_EL1</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>1001</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>BRBCR_EL2</td>
<td>BRBCR_EL2</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>1001</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>BRBCR_EL2</td>
<td>BRBCR_EL2</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>1001</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>BRBCR_EL2</td>
<td>BRBCR_EL2</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>1001</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>BRBCR_EL1</td>
<td>BRBCR_EL1</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>1001</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>BRBTS_EL1</td>
<td>BRBTS_EL1</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>1001</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>BRBINFINJ_EL1</td>
<td>BRBINFINJ_EL1</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>1001</td>
<td>0001</td>
<td>001</td>
<td>RW</td>
<td>BRBSRCINJ_EL1</td>
<td>BRBSRCINJ_EL1</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>1001</td>
<td>0001</td>
<td>010</td>
<td>RW</td>
<td>BRBTGTINJ_EL1</td>
<td>BRBTGTINJ_EL1</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>1001</td>
<td>0010</td>
<td>000</td>
<td>RO</td>
<td>BRBIDR0_EL1</td>
<td>BRBIDR0_EL1</td>
</tr>
<tr>
<td>10</td>
<td>011</td>
<td>0000</td>
<td>0001</td>
<td>000</td>
<td>RO</td>
<td>MDCCSR_EL0</td>
<td>MDCCSR_EL0</td>
</tr>
<tr>
<td>10</td>
<td>011</td>
<td>0000</td>
<td>0100</td>
<td>000</td>
<td>RW</td>
<td>DBGDTR_EL0</td>
<td>DBGDTR_EL0</td>
</tr>
<tr>
<td>10</td>
<td>011</td>
<td>0000</td>
<td>0101</td>
<td>000</td>
<td>RO</td>
<td>DBGDTRRX_EL0</td>
<td>DBGDTRRX_EL0</td>
</tr>
<tr>
<td>10</td>
<td>011</td>
<td>0000</td>
<td>0101</td>
<td>000</td>
<td>RO</td>
<td>DBGDTRRX_EL0</td>
<td>DBGDTRRX_EL0</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>0000</td>
<td>0111</td>
<td>000</td>
<td>RW</td>
<td>DBGVCR32_EL2</td>
<td>DBGVCR32_EL2</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>1001</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>BRBCR_EL2</td>
<td>BRBCR_EL2</td>
</tr>
<tr>
<td>10</td>
<td>101</td>
<td>1001</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>BRBCR_EL1</td>
<td>BRBCR_EL1</td>
</tr>
</tbody>
</table>

For more information, see *Mapping of the System registers between the Execution states* on page D1-4686.

---

**Note**

Table D16-1 lists the Armv8 debug System registers. For encoding information of the registers introduced by Armv9, see the register descriptions in Chapter D17 *AArch64 System Register Descriptions*.
**D16.3 op0==0b11, Moves to and from non-debug System registers, Special-purpose registers**

The instructions that move data to and from non-debug System registers are encoded with op0==0b11, except that some of this encoding space is reserved for IMPLEMENTATION DEFINED functionality. The encoding of these instructions is:

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 12 11 8 7 5 4 0</th>
<th>1 1 0 1 0 1 0 1 0 0</th>
<th>L</th>
<th>1 1</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The value of CRn provides the next level of decode of these instructions, as follows:

CRn==\{0, 1, 2, 3, 5, 6, 7, 9, 10, 12, 13, 14\}

See Instructions for accessing non-debug System registers.

CRn==4 See Instructions for accessing Special-purpose registers on page C5-668.

CRn==\{11, 15\} See Reserved encodings for IMPLEMENTATION DEFINED registers on page D16-5542.

**D16.3.1 Instructions for accessing non-debug System registers**

The A64 instructions for accessing System registers are:

- MSR <system_register>, Xt ; Write to System register
- MSR Xt, <system_register> ; Read from System register

Where <system_register> is the register name, for example MIDR_EL1.

This section includes only the System register access encodings for which both:

- op0 is 0b11.
- The value of CRn is one of \{0, 1, 2, 3, 4, 5, 6, 7, 9, 10, 12, 13, 14\}.

Note

- These encodings access the registers that are equivalent to the AArch32 System registers in the (coproc==0b1111) encoding space.
- While this group is described as accessing the non-debug System registers, its correct characterization is by the \{op0, CRn\} values given in this subsection, and the group includes the debug registers DLR_EL0, DSPSR_EL0, MDCR_EL2, MDCR_EL3, and SDER32_EL3, which are described in Debug registers on page D17-6393. These registers are exceptions to the standard encoding of debug registers, which has op0==0b10, see Instructions for accessing debug System registers on page D16-5519.

The instruction encoding for these accesses is:

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 12 11 8 7 5 4 0</th>
<th>1 1 0 1 0 1 0 1 0 0</th>
<th>L</th>
<th>1 1</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See text for permitted values of CRn

Table D16-2 on page D16-5523 shows the encodings of the register access instructions.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>000</td>
<td>RO</td>
<td>MIDR_EL1</td>
<td>MIDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>000</td>
<td>RO</td>
<td>MIDR_EL1</td>
<td>VPIDR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>000</td>
<td>RO</td>
<td>MIDR_EL1</td>
<td>VPIDR_EL2</td>
</tr>
<tr>
<td>op0</td>
<td>op1</td>
<td>CRn</td>
<td>CRm</td>
<td>op2</td>
<td>Access</td>
<td>Mnemonic</td>
<td>Register</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>--------</td>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>000</td>
<td>RO</td>
<td>VPIDR_EL2</td>
<td>VPIDR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>101</td>
<td>RO</td>
<td>MPIDR_EL1</td>
<td>MPIDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>101</td>
<td>RO</td>
<td>MPIDR_EL1</td>
<td>VMMPIDR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>101</td>
<td>RO</td>
<td>VMMPIDR_EL2</td>
<td>MPIDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>110</td>
<td>RO</td>
<td>REVIDR_EL1</td>
<td>REVIDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>000</td>
<td>RO</td>
<td>ID_PFR0_EL1</td>
<td>ID_PFR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>001</td>
<td>RO</td>
<td>ID_PFR1_EL1</td>
<td>ID_PFR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>010</td>
<td>RO</td>
<td>ID_DFR0_EL1</td>
<td>ID_DFR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>011</td>
<td>RO</td>
<td>ID_AFR0_EL1</td>
<td>ID_AFR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>100</td>
<td>RO</td>
<td>ID_MMFR0_EL1</td>
<td>ID_MMFR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>101</td>
<td>RO</td>
<td>ID_MMFR1_EL1</td>
<td>ID_MMFR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>110</td>
<td>RO</td>
<td>ID_MMFR2_EL1</td>
<td>ID_MMFR2_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0000</td>
<td>111</td>
<td>RO</td>
<td>ID_MMFR3_EL1</td>
<td>ID_MMFR3_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0010</td>
<td>000</td>
<td>RO</td>
<td>ID_ISAR0_EL1</td>
<td>ID_ISAR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0010</td>
<td>001</td>
<td>RO</td>
<td>ID_ISAR1_EL1</td>
<td>ID_ISAR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0010</td>
<td>010</td>
<td>RO</td>
<td>ID_ISAR2_EL1</td>
<td>ID_ISAR2_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0010</td>
<td>011</td>
<td>RO</td>
<td>ID_ISAR3_EL1</td>
<td>ID_ISAR3_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0010</td>
<td>100</td>
<td>RO</td>
<td>ID_ISAR4_EL1</td>
<td>ID_ISAR4_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0010</td>
<td>101</td>
<td>RO</td>
<td>ID_ISAR5_EL1</td>
<td>ID_ISAR5_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0010</td>
<td>110</td>
<td>RO</td>
<td>ID_MMFR4_EL1</td>
<td>ID_MMFR4_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0010</td>
<td>111</td>
<td>RO</td>
<td>ID_ISAR6_EL1</td>
<td>ID_ISAR6_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0011</td>
<td>000</td>
<td>RO</td>
<td>MVFR0_EL1</td>
<td>MVFR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0011</td>
<td>001</td>
<td>RO</td>
<td>MVFR1_EL1</td>
<td>MVFR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0011</td>
<td>010</td>
<td>RO</td>
<td>MVFR2_EL1</td>
<td>MVFR2_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0011</td>
<td>100</td>
<td>RO</td>
<td>ID_PFR2_EL1</td>
<td>ID_PFR2_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0011</td>
<td>101</td>
<td>RO</td>
<td>ID_DFR1_EL1</td>
<td>ID_DFR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0011</td>
<td>110</td>
<td>RO</td>
<td>ID_MMFR5_EL1</td>
<td>ID_MMFR5_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0011</td>
<td>n</td>
<td>RO</td>
<td>for n={3, 7} Reserved, RAZ.</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0100</td>
<td>000</td>
<td>RO</td>
<td>ID_AA64PFR0_EL1</td>
<td>ID_AA64PFR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0100</td>
<td>001</td>
<td>RO</td>
<td>ID_AA64PFR1_EL1</td>
<td>ID_AA64PFR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0000</td>
<td>0100</td>
<td>100</td>
<td>RO</td>
<td>ID_AA64ZFR0_EL1</td>
<td>ID_AA64ZFR0_EL1</td>
</tr>
</tbody>
</table>
### Table D16-2 Instruction encodings for non-debug System register access (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>000</td>
<td>000</td>
<td>0100</td>
<td>101</td>
<td>RO</td>
<td>ID_AA64SMFR0_EL1</td>
<td>ID_AA64SMFR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>000</td>
<td>0100</td>
<td>n</td>
<td>RO, for</td>
<td>Reserved, RAZ.</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>n={2, 3, 6, 7}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>000</td>
<td>0101</td>
<td>000</td>
<td>RO</td>
<td>ID_AA64DFR0_EL1</td>
<td>ID_AA64DFR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>000</td>
<td>0101</td>
<td>001</td>
<td>RO</td>
<td>ID_AA64DFR1_EL1</td>
<td>ID_AA64DFR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>000</td>
<td>0101</td>
<td>100</td>
<td>RO</td>
<td>ID_AA64AFR0_EL1</td>
<td>ID_AA64AFR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>000</td>
<td>0101</td>
<td>101</td>
<td>RO</td>
<td>ID_AA64AFR1_EL1</td>
<td>ID_AA64AFR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>000</td>
<td>0101</td>
<td>n</td>
<td>RO, for</td>
<td>Reserved, RAZ.</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>n={2, 3, 6, 7}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>000</td>
<td>0110</td>
<td>000</td>
<td>RO</td>
<td>ID_AA64ISAR0_EL1</td>
<td>ID_AA64ISAR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>000</td>
<td>0110</td>
<td>001</td>
<td>RO</td>
<td>ID_AA64ISAR1_EL1</td>
<td>ID_AA64ISAR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>000</td>
<td>0110</td>
<td>010</td>
<td>RO</td>
<td>ID_AA64ISAR2_EL1</td>
<td>ID_AA64ISAR2_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>000</td>
<td>0110</td>
<td>n</td>
<td>RO, for</td>
<td>Reserved, RAZ.</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>n=3-7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>000</td>
<td>0111</td>
<td>000</td>
<td>RO</td>
<td>ID_AA64MMFR0_EL1</td>
<td>ID_AA64MMFR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>000</td>
<td>0111</td>
<td>001</td>
<td>RO</td>
<td>ID_AA64MMFR1_EL1</td>
<td>ID_AA64MMFR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>000</td>
<td>0111</td>
<td>010</td>
<td>RO</td>
<td>ID_AA64MMFR2_EL1</td>
<td>ID_AA64MMFR2_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>000</td>
<td>0111</td>
<td>n</td>
<td>RO, for</td>
<td>Reserved, RAZ.</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>n=3-7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SCTLR_EL1</td>
<td>SCTLR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SCTLR_EL1</td>
<td>SCTLR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SCTLR_EL2</td>
<td>SCTLR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SCTLR_EL2</td>
<td>SCTLR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ACTLR_EL1</td>
<td>ACTLR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>CPACR_EL1</td>
<td>CPACR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>CPACR_EL1</td>
<td>CPACR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>CPACR_EL1</td>
<td>CPACR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>CPACR_EL2</td>
<td>CPACR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0000</td>
<td>101</td>
<td>RW</td>
<td>RGSR_EL1</td>
<td>RGSR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0000</td>
<td>110</td>
<td>RW</td>
<td>GCR_EL1</td>
<td>GCR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>ZCR_EL1</td>
<td>ZCR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>ZCR_EL1</td>
<td>ZCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>ZCR_EL2</td>
<td>ZCR_EL1</td>
</tr>
</tbody>
</table>
### Table D16-2 Instruction encodings for non-debug System register access (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>ZCR_EL2</td>
<td>ZCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0010</td>
<td>001</td>
<td>RW</td>
<td>TRFCR_EL1</td>
<td>TRFCR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0010</td>
<td>001</td>
<td>RW</td>
<td>TRFCR_EL1</td>
<td>TRFCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0010</td>
<td>001</td>
<td>RW</td>
<td>TRFCR_EL2</td>
<td>TRFCR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0010</td>
<td>100</td>
<td>RW</td>
<td>SMPRI_EL1</td>
<td>SMPRI_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0010</td>
<td>110</td>
<td>RW</td>
<td>SMCR_EL1</td>
<td>SMCR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0010</td>
<td>110</td>
<td>RW</td>
<td>SMCR_EL1</td>
<td>SMCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0001</td>
<td>0010</td>
<td>110</td>
<td>RW</td>
<td>SMCR_EL2</td>
<td>SMCR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>TTBR0_EL1</td>
<td>TTBR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>TTBR0_EL1</td>
<td>TTBR0_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>TTBR1_EL1</td>
<td>TTBR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>TTBR1_EL1</td>
<td>TTBR1_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>TTBR1_EL1</td>
<td>TTBR1_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>TTBR1_EL2</td>
<td>TTBR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>TCR_EL1</td>
<td>TCR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>TCR_EL1</td>
<td>TCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>TCR_EL2</td>
<td>TCR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>TCR_EL2</td>
<td>TCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>APIAKeyLo_EL1</td>
<td>APIAKeyLo_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0001</td>
<td>001</td>
<td>RW</td>
<td>APIAKeyHi_EL1</td>
<td>APIAKeyHi_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0001</td>
<td>010</td>
<td>RW</td>
<td>APIBKeyLo_EL1</td>
<td>APIBKeyLo_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0001</td>
<td>011</td>
<td>RW</td>
<td>APIBKeyHi_EL1</td>
<td>APIBKeyHi_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>APDAKeyLo_EL1</td>
<td>APDAKeyLo_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0010</td>
<td>001</td>
<td>RW</td>
<td>APDAKeyHi_EL1</td>
<td>APDAKeyHi_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>APDBKeyLo_EL1</td>
<td>APDBKeyLo_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0010</td>
<td>011</td>
<td>RW</td>
<td>APDBKeyHi_EL1</td>
<td>APDBKeyHi_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0011</td>
<td>000</td>
<td>RW</td>
<td>APGAKeyLo_EL1</td>
<td>APGAKeyLo_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0010</td>
<td>0011</td>
<td>001</td>
<td>RW</td>
<td>APGAKeyHi_EL1</td>
<td>APGAKeyHi_EL1</td>
</tr>
</tbody>
</table>
Table D16-2 Instruction encodings for non-debug System register access (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL1</td>
<td>SPSR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL1</td>
<td>SPSR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL2</td>
<td>SPSR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL2</td>
<td>SPSR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ELR_EL1</td>
<td>ELR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ELR_EL1</td>
<td>ELR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ELR_EL2</td>
<td>ELR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ELR_EL2</td>
<td>ELR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>SP_EL0</td>
<td>SP_EL0</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0010</td>
<td>000</td>
<td>-</td>
<td>SPSel</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0010</td>
<td>010</td>
<td>-</td>
<td>CurrentEL</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0010</td>
<td>011</td>
<td>-</td>
<td>PAN</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0010</td>
<td>100</td>
<td>-</td>
<td>UAO</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0011</td>
<td>000</td>
<td>-</td>
<td>ALLINT</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0110</td>
<td>000</td>
<td>RW</td>
<td>ICC_PMR_EL1</td>
<td>ICC_PMR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0110</td>
<td>000</td>
<td>RW</td>
<td>ICC_PMR_EL1</td>
<td>ICC_PMR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0110</td>
<td>000</td>
<td>RW</td>
<td>ICV_PMR_EL1</td>
<td>ICV_PMR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0100</td>
<td>0110</td>
<td>000</td>
<td>RW</td>
<td>ICV_PMR_EL1</td>
<td>ICV_PMR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>AFSR0_EL1</td>
<td>AFSR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>AFSR0_EL1</td>
<td>AFSR0_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>AFSR0_EL2</td>
<td>AFSR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>AFSR0_EL2</td>
<td>AFSR0_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0001</td>
<td>001</td>
<td>RW</td>
<td>AFSR1_EL1</td>
<td>AFSR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0001</td>
<td>001</td>
<td>RW</td>
<td>AFSR1_EL1</td>
<td>AFSR1_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0001</td>
<td>001</td>
<td>RW</td>
<td>AFSR1_EL2</td>
<td>AFSR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0001</td>
<td>001</td>
<td>RW</td>
<td>AFSR1_EL2</td>
<td>AFSR1_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>ESR_EL1</td>
<td>ESR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>ESR_EL1</td>
<td>ESR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>ESR_EL2</td>
<td>ESR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>ESR_EL2</td>
<td>ESR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0011</td>
<td>000</td>
<td>RO</td>
<td>ERRIDR_EL1</td>
<td>ERRIDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0011</td>
<td>001</td>
<td>RW</td>
<td>ERRSELR_EL1</td>
<td>ERRSELR_EL1</td>
</tr>
</tbody>
</table>
### Table D16-2 Instruction encodings for non-debug System register access (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0100</td>
<td>000</td>
<td>RO</td>
<td>ERXFR_EL1</td>
<td>ERXFR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0100</td>
<td>001</td>
<td>RW</td>
<td>ERXCTRL_EL1</td>
<td>ERXCTRL_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0100</td>
<td>010</td>
<td>RW</td>
<td>ERXSTATUS_EL1</td>
<td>ERXSTATUS_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0100</td>
<td>011</td>
<td>RW</td>
<td>ERXADDR_EL1</td>
<td>ERXADDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0100</td>
<td>100</td>
<td>RO</td>
<td>ERXPFGF_EL1</td>
<td>ERXPFGF_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0100</td>
<td>101</td>
<td>RW</td>
<td>ERXPFGCTL_EL1</td>
<td>ERXPFGCTL_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0100</td>
<td>110</td>
<td>RW</td>
<td>ERXPFGCDN_EL1</td>
<td>ERXPFGCDN_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0110</td>
<td>000</td>
<td>RW</td>
<td>ERXMISC0_EL1</td>
<td>ERXMISC0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0110</td>
<td>001</td>
<td>RW</td>
<td>ERXMISC1_EL1</td>
<td>ERXMISC1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0110</td>
<td>010</td>
<td>RW</td>
<td>ERXMISC2_EL1</td>
<td>ERXMISC2_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0110</td>
<td>011</td>
<td>RW</td>
<td>ERXMISC3_EL1</td>
<td>ERXMISC3_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0101</td>
<td>0110</td>
<td>000</td>
<td>RW</td>
<td>TFSR_EL1</td>
<td>TFSR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0110</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>FAR_EL1</td>
<td>FAR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0110</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>FAR_EL2</td>
<td>FAR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0110</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>FAR_EL2</td>
<td>FAR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>0111</td>
<td>0100</td>
<td>000</td>
<td>RW</td>
<td>PAR_EL1</td>
<td>PAR_EL1</td>
</tr>
</tbody>
</table>
## Table D16-2 Instruction encodings for non-debug System register access (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>000</td>
<td>1001</td>
<td>1010</td>
<td>000</td>
<td>RW</td>
<td>PMBLIMITR_EL1</td>
<td>PMBLIMITR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1001</td>
<td>1010</td>
<td>001</td>
<td>RW</td>
<td>PMBPTR_EL1</td>
<td>PMBPTR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1001</td>
<td>1010</td>
<td>011</td>
<td>RW</td>
<td>PMBSR_EL1</td>
<td>PMBSR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1001</td>
<td>1010</td>
<td>111</td>
<td>RO</td>
<td>PMBIDR_EL1</td>
<td>PMBIDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1001</td>
<td>1011</td>
<td>000</td>
<td>RW</td>
<td>TRBLIMITR_EL1</td>
<td>TRBLIMITR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1001</td>
<td>1011</td>
<td>001</td>
<td>RW</td>
<td>TRBPTR_EL1</td>
<td>TRBPTR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1001</td>
<td>1011</td>
<td>010</td>
<td>RW</td>
<td>TRBBASER_EL1</td>
<td>TRBBASER_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1001</td>
<td>1011</td>
<td>011</td>
<td>RW</td>
<td>TRBSR_EL1</td>
<td>TRBSR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1001</td>
<td>1011</td>
<td>100</td>
<td>RW</td>
<td>TRBMAR_EL1</td>
<td>TRBMAR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1001</td>
<td>1011</td>
<td>110</td>
<td>RW</td>
<td>TRBTRG_EL1</td>
<td>TRBTRG_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1001</td>
<td>1011</td>
<td>111</td>
<td>RO</td>
<td>TRBIDR_EL1</td>
<td>TRBIDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1001</td>
<td>1110</td>
<td>001</td>
<td>RW</td>
<td>PMINTENSET_EL1</td>
<td>PMINTENSET_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1001</td>
<td>1110</td>
<td>010</td>
<td>RW</td>
<td>PMINTENCCLR_EL1</td>
<td>PMINTENCCLR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1001</td>
<td>1110</td>
<td>110</td>
<td>RO</td>
<td>PMMIR_EL1</td>
<td>PMMIR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>MAIR_EL1</td>
<td>MAIR_EL1</td>
<td>MAIR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>MAIR_EL2</td>
<td>MAIR_EL2</td>
<td>MAIR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>MAIR_EL2</td>
<td>MAIR_EL2</td>
<td>MAIR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0100</td>
<td>000</td>
<td>RW</td>
<td>AMAIR_EL1</td>
<td>AMAIR_EL1</td>
<td>AMAIR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0100</td>
<td>000</td>
<td>RW</td>
<td>AMAIR_EL1</td>
<td>AMAIR_EL1</td>
<td>AMAIR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0100</td>
<td>000</td>
<td>RW</td>
<td>AMAIR_EL2</td>
<td>AMAIR_EL2</td>
<td>AMAIR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0100</td>
<td>010</td>
<td>RW</td>
<td>LORSA_EL1</td>
<td>LORSA_EL1</td>
<td>LORSA_EL1</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0100</td>
<td>001</td>
<td>RW</td>
<td>LOREA_EL1</td>
<td>LOREA_EL1</td>
<td>LOREA_EL1</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0100</td>
<td>010</td>
<td>RW</td>
<td>LORN_EL1</td>
<td>LORN_EL1</td>
<td>LORN_EL1</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0100</td>
<td>011</td>
<td>RW</td>
<td>LORC_EL1</td>
<td>LORC_EL1</td>
<td>LORC_EL1</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0100</td>
<td>100</td>
<td>RO</td>
<td>MPAMIDR_EL1</td>
<td>MPAMIDR_EL1</td>
<td>MPAMIDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0100</td>
<td>111</td>
<td>RO</td>
<td>LORID_EL1</td>
<td>LORID_EL1</td>
<td>LORID_EL1</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0101</td>
<td>000</td>
<td>RW</td>
<td>MPAM1_EL1</td>
<td>MPAM1_EL1</td>
<td>MPAM1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0101</td>
<td>000</td>
<td>RW</td>
<td>MPAM1_EL1</td>
<td>MPAM1_EL1</td>
<td>MPAM1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0101</td>
<td>000</td>
<td>RW</td>
<td>MPAM2_EL2</td>
<td>MPAM2_EL2</td>
<td>MPAM2_EL2</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0101</td>
<td>000</td>
<td>RW</td>
<td>MPAM2_EL2</td>
<td>MPAM2_EL2</td>
<td>MPAM2_EL2</td>
</tr>
<tr>
<td>op0</td>
<td>op1</td>
<td>CRn</td>
<td>CRm</td>
<td>op2</td>
<td>Access</td>
<td>Mnemonic</td>
<td>Register</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>--------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>VBAR_EL1</td>
<td>VBAR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>VBAR_EL2</td>
<td>VBAR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>RMR_EL1</td>
<td>RMR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>0001</td>
<td>000</td>
<td>RO</td>
<td>ISR_EL1</td>
<td>ISR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>0001</td>
<td>001</td>
<td>RW</td>
<td>DISR_EL1</td>
<td>DISR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>0001</td>
<td>010</td>
<td>RW</td>
<td>VDISR_EL2</td>
<td>DISR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1000</td>
<td>000</td>
<td>RO</td>
<td>ICC_IAR0_EL1</td>
<td>ICC_IAR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1000</td>
<td>001</td>
<td>WO</td>
<td>ICC_EOIR0_EL1</td>
<td>ICC_EOIR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1000</td>
<td>010</td>
<td>RO</td>
<td>ICC_HPPIR0_EL1</td>
<td>ICC_HPPIR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1000</td>
<td>011</td>
<td>RW</td>
<td>ICC_BPR0_EL1</td>
<td>ICC_BPR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1000</td>
<td>111</td>
<td>RW</td>
<td>ICC_BPR0_EL1</td>
<td>ICC_BPR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1000</td>
<td>110</td>
<td>RW</td>
<td>ICC_BPR0_EL1</td>
<td>ICC_BPR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1000</td>
<td>110</td>
<td>RW</td>
<td>ICC_BPR0_EL1</td>
<td>ICC_BPR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1000</td>
<td>111</td>
<td>RW</td>
<td>ICC_BPR0_EL1</td>
<td>ICC_BPR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1000</td>
<td>110</td>
<td>RW</td>
<td>ICC_BPR0_EL1</td>
<td>ICC_BPR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1000</td>
<td>111</td>
<td>RW</td>
<td>ICC_BPR0_EL1</td>
<td>ICC_BPR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1000</td>
<td>110</td>
<td>RW</td>
<td>ICC_BPR0_EL1</td>
<td>ICC_BPR0_EL1</td>
</tr>
</tbody>
</table>

Table D16-2 Instruction encodings for non-debug System register access (continued)
### Table D16-2 Instruction encodings for non-debug System register access (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1001</td>
<td>0:m[1:0]</td>
<td>RW</td>
<td>ICC_AP1R&lt;n&gt;_EL1</td>
<td>ICC_AP1R_EL1[]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1001</td>
<td>0:m[1:0]</td>
<td>RW</td>
<td>ICC_AP1R&lt;n&gt;_EL1</td>
<td>ICC_AP1R_EL1_NS[1]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1001</td>
<td>0:m[1:0]</td>
<td>RW</td>
<td>ICC_AP1R&lt;n&gt;_EL1</td>
<td>ICC_AP1R_EL1_S[1]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1001</td>
<td>0:m[1:0]</td>
<td>RW</td>
<td>ICC_AP1R&lt;n&gt;_EL1</td>
<td>ICC_AP1R_EL1[]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1001</td>
<td>0:m[1:0]</td>
<td>RW</td>
<td>ICC_AP1R&lt;n&gt;_EL1</td>
<td>ICC_AP1R_EL1_NS[1]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1001</td>
<td>0:m[1:0]</td>
<td>RW</td>
<td>ICC_AP1R&lt;n&gt;_EL1</td>
<td>ICC_AP1R_EL1_S[1]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1001</td>
<td>0:m[1:0]</td>
<td>RW</td>
<td>ICC_AP1R&lt;n&gt;_EL1</td>
<td>ICC_AP1R_EL1[]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1001</td>
<td>101</td>
<td>RO</td>
<td>ICC_NMIAR1_EL1</td>
<td>ICC_NMIAR1_EL1[]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1001</td>
<td>101</td>
<td>RO</td>
<td>ICC_NMIAR1_EL1</td>
<td>ICC_NMIAR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1001</td>
<td>101</td>
<td>RO</td>
<td>ICC_NMIAR1_EL1</td>
<td>ICC_NMIAR1_EL1[]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1001</td>
<td>101</td>
<td>RO</td>
<td>ICC_NMIAR1_EL1</td>
<td>ICC_NMIAR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1011</td>
<td>001</td>
<td>WO</td>
<td>ICC_DIR_EL1</td>
<td>ICC_DIR_EL1[]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1011</td>
<td>001</td>
<td>WO</td>
<td>ICC_DIR_EL1</td>
<td>ICC_DIR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1011</td>
<td>001</td>
<td>WO</td>
<td>ICC_DIR_EL1</td>
<td>ICC_DIR_EL1[]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1011</td>
<td>001</td>
<td>WO</td>
<td>ICC_DIR_EL1</td>
<td>ICC_DIR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1011</td>
<td>011</td>
<td>RO</td>
<td>ICC_RPR_EL1</td>
<td>ICC_RPR_EL1[]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1011</td>
<td>011</td>
<td>RO</td>
<td>ICC_RPR_EL1</td>
<td>ICC_RPR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1011</td>
<td>011</td>
<td>RO</td>
<td>ICC_RPR_EL1</td>
<td>ICC_RPR_EL1[]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1011</td>
<td>011</td>
<td>RO</td>
<td>ICC_RPR_EL1</td>
<td>ICC_RPR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1011</td>
<td>101</td>
<td>WO</td>
<td>ICC_SGI1R_EL1</td>
<td>ICC_SGI1R_EL1[]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1011</td>
<td>110</td>
<td>WO</td>
<td>ICC_ASGI1R_EL1</td>
<td>ICC_ASGI1R_EL1[]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1011</td>
<td>111</td>
<td>WO</td>
<td>ICC_SGI0R_EL1</td>
<td>ICC_SGI0R_EL1[]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>000</td>
<td>RO</td>
<td>ICC_IAR1_EL1</td>
<td>ICC_IAR1_EL1[]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>000</td>
<td>RO</td>
<td>ICC_IAR1_EL1</td>
<td>ICC_IAR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>000</td>
<td>RO</td>
<td>ICC_IAR1_EL1</td>
<td>ICC_IAR1_EL1[]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>000</td>
<td>RO</td>
<td>ICC_IAR1_EL1</td>
<td>ICC_IAR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>001</td>
<td>WO</td>
<td>ICC_EOIR1_EL1</td>
<td>ICC_EOIR1_EL1[]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>001</td>
<td>WO</td>
<td>ICC_EOIR1_EL1</td>
<td>ICC_EOIR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>001</td>
<td>WO</td>
<td>ICC_EOIR1_EL1</td>
<td>ICC_EOIR1_EL1[]</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>001</td>
<td>WO</td>
<td>ICC_EOIR1_EL1</td>
<td>ICC_EOIR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>010</td>
<td>RO</td>
<td>ICC_HPPIR1_EL1</td>
<td>ICC_HPPIR1_EL1[]</td>
</tr>
</tbody>
</table>
Table D16-2 Instruction encodings for non-debug System register access (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>010</td>
<td>RO</td>
<td>ICC_HPPRI1_EL1</td>
<td>ICC_HPPRI1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>010</td>
<td>RO</td>
<td>ICV_HPPRI1_EL1</td>
<td>ICV_HPPRI1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>010</td>
<td>RO</td>
<td>ICC_HPPRI1_EL1</td>
<td>ICV_HPPRI1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>010</td>
<td>RO</td>
<td>ICC_HPPRI1_EL1</td>
<td>ICV_HPPRI1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>011</td>
<td>RW</td>
<td>ICC_BPR1_EL1</td>
<td>ICC_BPR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>011</td>
<td>RW</td>
<td>ICV_BPR1_EL1</td>
<td>ICC_BPR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>011</td>
<td>RW</td>
<td>ICV_BPR1_EL1</td>
<td>ICC_BPR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>011</td>
<td>RW</td>
<td>ICV_BPR1_EL1</td>
<td>ICC_BPR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>100</td>
<td>RW</td>
<td>ICC_CTLR_EL1</td>
<td>ICC_CTLR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>100</td>
<td>RW</td>
<td>ICC_CTLR_EL1</td>
<td>ICC_CTLR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>100</td>
<td>RW</td>
<td>ICC_CTLR_EL1</td>
<td>ICC_CTLR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>100</td>
<td>RW</td>
<td>ICC_CTLR_EL1</td>
<td>ICC_CTLR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>100</td>
<td>RW</td>
<td>ICC_CTLR_EL1</td>
<td>ICC_CTLR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>101</td>
<td>RW</td>
<td>ICC_SRE_EL1</td>
<td>ICC_SRE_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>101</td>
<td>RW</td>
<td>ICC_SRE_EL1</td>
<td>ICC_SRE_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>101</td>
<td>RW</td>
<td>ICC_SRE_EL1</td>
<td>ICC_SRE_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>101</td>
<td>RW</td>
<td>ICC_SRE_EL1</td>
<td>ICC_SRE_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>110</td>
<td>RW</td>
<td>ICC_IGRPEN0_EL1</td>
<td>ICC_IGRPEN0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>110</td>
<td>RW</td>
<td>ICC_IGRPEN0_EL1</td>
<td>ICC_IGRPEN0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>110</td>
<td>RW</td>
<td>ICC_IGRPEN0_EL1</td>
<td>ICC_IGRPEN0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>110</td>
<td>RW</td>
<td>ICC_IGRPEN0_EL1</td>
<td>ICC_IGRPEN0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>111</td>
<td>RW</td>
<td>ICC_IGRPEN1_EL1</td>
<td>ICC_IGRPEN1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>111</td>
<td>RW</td>
<td>ICC_IGRPEN1_EL1</td>
<td>ICC_IGRPEN1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>111</td>
<td>RW</td>
<td>ICC_IGRPEN1_EL1</td>
<td>ICC_IGRPEN1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>111</td>
<td>RW</td>
<td>ICC_IGRPEN1_EL1</td>
<td>ICC_IGRPEN1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>111</td>
<td>RW</td>
<td>ICC_IGRPEN1_EL1</td>
<td>ICC_IGRPEN1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>111</td>
<td>RW</td>
<td>ICC_IGRPEN1_EL1</td>
<td>ICC_IGRPEN1_EL1</td>
</tr>
</tbody>
</table>
### AArch64 System Register Encoding

**D16.3 op0==0b11, Moves to and from non-debug System registers, Special-purpose registers**

#### Table D16-2 Instruction encodings for non-debug System register access (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>111</td>
<td>RW</td>
<td>ICV_IGRPEN1_EL1</td>
<td>ICC_IGRPEN1_EL1_S</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1100</td>
<td>1100</td>
<td>111</td>
<td>RW</td>
<td>ICV_IGRPEN1_EL1</td>
<td>ICV_IGRPEN1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1101</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>CONTEXTIDR_EL1</td>
<td>CONTEXTIDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1101</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>CONTEXTIDR_EL1</td>
<td>CONTEXTIDR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1101</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>CONTEXTIDR_EL2</td>
<td>CONTEXTIDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1101</td>
<td>0000</td>
<td>100</td>
<td>RW</td>
<td>TPIDR_EL1</td>
<td>TPIDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1101</td>
<td>0000</td>
<td>101</td>
<td>RW</td>
<td>ACCDATA_EL1</td>
<td>ACCDATA_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1101</td>
<td>0000</td>
<td>111</td>
<td>RW</td>
<td>SCXTNUM_EL1</td>
<td>SCXTNUM_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1101</td>
<td>0000</td>
<td>111</td>
<td>RW</td>
<td>SCXTNUM_EL1</td>
<td>SCXTNUM_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1101</td>
<td>0000</td>
<td>111</td>
<td>RW</td>
<td>SCXTNUM_EL2</td>
<td>SCXTNUM_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1101</td>
<td>0000</td>
<td>111</td>
<td>RW</td>
<td>SCXTNUM_EL2</td>
<td>SCXTNUM_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1110</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>CNTHCTL_EL2</td>
<td>CNTHCTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1110</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>CNTHCTL_EL2</td>
<td>CNTKCTL_EL1</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1110</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>CNTKCTL_EL1</td>
<td>CNTHCTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>1110</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>CNTKCTL_EL1</td>
<td>CNTKCTL_EL1</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>0000</td>
<td>0000</td>
<td>000</td>
<td>RO</td>
<td>CCSIDR_EL1</td>
<td>CCSIDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>0000</td>
<td>0000</td>
<td>001</td>
<td>RO</td>
<td>CLIDR_EL1</td>
<td>CLIDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>0000</td>
<td>0000</td>
<td>010</td>
<td>RO</td>
<td>CCSIDR2_EL1</td>
<td>CCSIDR2_EL1</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>0000</td>
<td>0000</td>
<td>100</td>
<td>RO</td>
<td>GMID_EL1</td>
<td>GMID_EL1</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>0000</td>
<td>0000</td>
<td>110</td>
<td>RO</td>
<td>SMIDR_EL1</td>
<td>SMIDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>0000</td>
<td>0000</td>
<td>111</td>
<td>RO</td>
<td>AIDR_EL1</td>
<td>AIDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>0000</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>CSSELR_EL1</td>
<td>CSSELR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0000</td>
<td>0000</td>
<td>001</td>
<td>RO</td>
<td>CTR_EL0</td>
<td>CTR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0000</td>
<td>0000</td>
<td>010</td>
<td>RO</td>
<td>DCZID_EL0</td>
<td>DCZID_EL0</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0010</td>
<td>0100</td>
<td>000</td>
<td>RO</td>
<td>RNDR</td>
<td>RNDR</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0010</td>
<td>0100</td>
<td>001</td>
<td>RO</td>
<td>RNDRRS</td>
<td>RNDRRS</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0010</td>
<td>000</td>
<td>-</td>
<td>NZCV</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0010</td>
<td>001</td>
<td>-</td>
<td>DAIF</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0010</td>
<td>010</td>
<td>-</td>
<td>SVCR</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0010</td>
<td>101</td>
<td>-</td>
<td>DIT</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0010</td>
<td>110</td>
<td>-</td>
<td>SSBS</td>
<td>-</td>
</tr>
<tr>
<td>op0</td>
<td>op1</td>
<td>CRn</td>
<td>CRm</td>
<td>op2</td>
<td>Access</td>
<td>Mnemonic</td>
<td>Register</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>--------</td>
<td>------------</td>
<td>--------------</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0010</td>
<td>111</td>
<td>-</td>
<td>TCO</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0100</td>
<td>000</td>
<td>RW</td>
<td>FPCR</td>
<td>FPCR</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0100</td>
<td>001</td>
<td>RW</td>
<td>FPSR</td>
<td>FPSR</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0101</td>
<td>000</td>
<td>RW</td>
<td>DSPSR_EL0</td>
<td>DSPSR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0100</td>
<td>0101</td>
<td>001</td>
<td>RW</td>
<td>DLR_EL0</td>
<td>DLR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>1100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>PMCR_EL0</td>
<td>PMCR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>1100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>PMCNTENSEL0</td>
<td>PMCNTENSEL0</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>1100</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>PMCTENCLR_EL0</td>
<td>PMCTENCLR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>1100</td>
<td>0010</td>
<td>011</td>
<td>RW</td>
<td>PMOVSCLR_EL0</td>
<td>PMOVSCLR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>1100</td>
<td>0100</td>
<td>100</td>
<td>WO</td>
<td>PMSWINC_EL0</td>
<td>PMSWINC_EL0</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>1100</td>
<td>0100</td>
<td>101</td>
<td>RW</td>
<td>PMSELR_EL0</td>
<td>PMSELR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>1100</td>
<td>0100</td>
<td>110</td>
<td>RO</td>
<td>PMCEID0_EL0</td>
<td>PMCEID0_EL0</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>1100</td>
<td>0100</td>
<td>111</td>
<td>RO</td>
<td>PMCEID1_EL0</td>
<td>PMCEID1_EL0</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>1101</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>PMCCNTR_EL0</td>
<td>PMCCNTR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>1101</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>PMXEVTYPE_EL0</td>
<td>PMXEVTYPE_EL0</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>1101</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>PMXEVTYPE_EL0</td>
<td>PMXEVTYPE_EL0</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>1101</td>
<td>0000</td>
<td>011</td>
<td>RW</td>
<td>PMXEVNCNT_EL0</td>
<td>PMXEVNCNT_EL0</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>1110</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>PMUSERENR_EL0</td>
<td>PMUSERENR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>1110</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>PMOVSSSET_EL0</td>
<td>PMOVSSSET_EL0</td>
</tr>
<tr>
<td>11</td>
<td>1101</td>
<td>0000</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>TPIDR_EL0</td>
<td>TPIDR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>1101</td>
<td>0000</td>
<td>0010</td>
<td>011</td>
<td>RW</td>
<td>TPIDRRO_EL0</td>
<td>TPIDRRO_EL0</td>
</tr>
<tr>
<td>11</td>
<td>1101</td>
<td>0001</td>
<td>0010</td>
<td>010</td>
<td>RO</td>
<td>AMCR_EL0</td>
<td>AMCR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>1101</td>
<td>0001</td>
<td>0010</td>
<td>011</td>
<td>RO</td>
<td>AMCFGR_EL0</td>
<td>AMCFGR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>1101</td>
<td>0001</td>
<td>0010</td>
<td>011</td>
<td>RO</td>
<td>AMCGCR_EL0</td>
<td>AMCGCR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>1101</td>
<td>0001</td>
<td>0010</td>
<td>011</td>
<td>RO</td>
<td>AMUSERENR_EL0</td>
<td>AMUSERENR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>1101</td>
<td>0001</td>
<td>0010</td>
<td>100</td>
<td>RW</td>
<td>AMCNTENCLR0_EL0</td>
<td>AMCNTENCLR0_EL0</td>
</tr>
<tr>
<td>11</td>
<td>1101</td>
<td>0001</td>
<td>0010</td>
<td>101</td>
<td>RW</td>
<td>AMCNTENSEL0_EL0</td>
<td>AMCNTENSEL0_EL0</td>
</tr>
<tr>
<td>11</td>
<td>1101</td>
<td>0001</td>
<td>0100</td>
<td>100</td>
<td>RO</td>
<td>AMCG1IDR_EL0</td>
<td>AMCG1IDR_EL0</td>
</tr>
<tr>
<td>11</td>
<td>1101</td>
<td>0001</td>
<td>0110</td>
<td>000</td>
<td>RW</td>
<td>AMCNTENCLR1_EL0</td>
<td>AMCNTENCLR1_EL0</td>
</tr>
<tr>
<td>11</td>
<td>1101</td>
<td>0001</td>
<td>0110</td>
<td>001</td>
<td>RW</td>
<td>AMCNTENSEL1_EL0</td>
<td>AMCNTENSEL1_EL0</td>
</tr>
</tbody>
</table>

Table D16-2 Instruction encodings for non-debug System register access (continued)
### Table D16-2 Instruction encodings for non-debug System register access (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>011</td>
<td>010</td>
<td>m[3]</td>
<td>m[2:0]</td>
<td>RW</td>
<td>AMEVCNTR0(!&lt;&gt;) EL0</td>
<td>AMEVCNTR0_EL0[]</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>011</td>
<td>m[3]</td>
<td>m[2:0]</td>
<td>RO</td>
<td>AMEVTYPE0(!&lt;&gt;) EL0</td>
<td>AMEVTYPE0_EL0[]</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>110</td>
<td>m[3]</td>
<td>m[2:0]</td>
<td>RW</td>
<td>AMEVCNTR1(!&lt;&gt;) EL0</td>
<td>AMEVCNTR1_EL0[]</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>111</td>
<td>m[3]</td>
<td>m[2:0]</td>
<td>RW</td>
<td>AMEVTYPE1(!&lt;&gt;) EL0</td>
<td>AMEVTYPE1_EL0[]</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>CNTFRQ_EL0</td>
<td>CNTFRQ_EL0</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0000</td>
<td>001</td>
<td>-</td>
<td>CNTPCT_EL0</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0000</td>
<td>010</td>
<td>-</td>
<td>CNTVCT_EL0</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0000</td>
<td>101</td>
<td>-</td>
<td>CNTPCTSS_EL0</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0000</td>
<td>110</td>
<td>-</td>
<td>CNTVCTSS_EL0</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>000</td>
<td>-</td>
<td>CNTHP_TV AL_EL2</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>000</td>
<td>-</td>
<td>CNTHPS_TV AL_EL2</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>000</td>
<td>-</td>
<td>CNTP_TV AL_EL0</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>001</td>
<td>RW</td>
<td>CNTHP_CTL_EL2</td>
<td>CNTHPS_CTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>001</td>
<td>RW</td>
<td>CNTHP_CTL_EL2</td>
<td>CNTHPS_CTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>001</td>
<td>RW</td>
<td>CNTHP_CTL_EL2</td>
<td>CNTP_CTL_EL0</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>001</td>
<td>RW</td>
<td>CNTHPS_CTL_EL2</td>
<td>CNTHPS_CTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>001</td>
<td>RW</td>
<td>CNTHPS_CTL_EL2</td>
<td>CNTHP_CTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>001</td>
<td>RW</td>
<td>CNTHPS_CTL_EL2</td>
<td>CNTP_CTL_EL0</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>001</td>
<td>RW</td>
<td>CNTHPS_CTL_EL2</td>
<td>CNTHPS_CTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>CNTHP_CVAL_EL2</td>
<td>CNTHPS_CVAL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>CNTHP_CVAL_EL2</td>
<td>CNTHPS_CVAL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>CNTHP_CVAL_EL2</td>
<td>CNTP_CVAL_EL0</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>CNTHPS_CVAL_EL2</td>
<td>CNTHPS_CVAL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>CNTHPS_CVAL_EL2</td>
<td>CNTHP_CVAL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>CNTHPS_CVAL_EL2</td>
<td>CNTP_CVAL_EL0</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0010</td>
<td>000</td>
<td>-</td>
<td>CNTHV_TV AL_EL2</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>000</td>
<td>-</td>
<td>CNTHVS_TV AL_EL2</td>
<td>-</td>
</tr>
</tbody>
</table>
### Table D16-2 Instruction encodings for non-debug System register access (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>000</td>
<td>-</td>
<td>CNTV_TV_AL_EL0</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>001</td>
<td>RW</td>
<td>CNTHV_CTL_EL2</td>
<td>CNTHVS_CTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>001</td>
<td>RW</td>
<td>CNTHV_CTL_EL2</td>
<td>CNTHV_CTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>001</td>
<td>RW</td>
<td>CNTHV_CTL_EL2</td>
<td>CNTV_CTL_EL0</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>001</td>
<td>RW</td>
<td>CNTHVS_CTL_EL2</td>
<td>CNTHVS_CTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>001</td>
<td>RW</td>
<td>CNTHVS_CTL_EL2</td>
<td>CNTHV_CTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>001</td>
<td>RW</td>
<td>CNTHV_CTL_EL2</td>
<td>CNTV_CTL_EL0</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>001</td>
<td>RW</td>
<td>CNTV_CTL_EL0</td>
<td>CNTHV_CTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>001</td>
<td>RW</td>
<td>CNTV_CTL_EL0</td>
<td>CNTV_CTL_EL0</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>CNTHV_CVAL_EL2</td>
<td>CNTHVS_CVAL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>CNTHV_CVAL_EL2</td>
<td>CNTHVS_CVAL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>CNTHV_CVAL_EL2</td>
<td>CNTV_CVAL_EL0</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>CNTHV_CVAL_EL2</td>
<td>CNTHVS_CVAL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>CNTHVS_CVAL_EL2</td>
<td>CNTV_CVAL_EL0</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>CNTHVS_CVAL_EL2</td>
<td>CNTHVS_CVAL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>CNTHVS_CVAL_EL2</td>
<td>CNTV_CVAL_EL0</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>CNTHVS_CVAL_EL2</td>
<td>CNTHVS_CVAL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>CNTHVS_CVAL_EL2</td>
<td>CNTV_CVAL_EL0</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>CNTV_CVAL_EL0</td>
<td>CNTHV_CVAL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>1110</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>CNTV_CVAL_EL0</td>
<td>CNTV_CVAL_EL0</td>
</tr>
<tr>
<td>11</td>
<td>1110</td>
<td>10:m[4:3]</td>
<td>m[2:0]</td>
<td>RW</td>
<td>PMEVCNTR&lt;n&gt;_EL0</td>
<td>PMEVCNTR_EL0[]</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1110</td>
<td>1111</td>
<td>111</td>
<td>RW</td>
<td>PMCCFILTER_EL0</td>
<td>PMCCFILTER_EL0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1110</td>
<td>11:m[4:3]</td>
<td>m[2:0]</td>
<td>RW</td>
<td>PMEVTYPER&lt;n&gt;_EL0</td>
<td>PMEVTYPER_EL0[]</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>0000</td>
<td>000</td>
<td>RO</td>
<td>VPIDR_EL2</td>
<td>MIDR_EL1</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>VPIDR_EL2</td>
<td>VPIDR_EL2</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>0000</td>
<td>101</td>
<td>RO</td>
<td>VMPIDR_EL2</td>
<td>VMPIDR_EL2</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>0000</td>
<td>101</td>
<td>RW</td>
<td>VMPIDR_EL2</td>
<td>VMPIDR_EL2</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SCTLR_EL2</td>
<td>SCTLR_EL2</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ACTLR_EL2</td>
<td>ACTLR_EL2</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>HCR_EL2</td>
<td>HCR_EL2</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>0001</td>
<td>001</td>
<td>RW</td>
<td>MDCR_EL2</td>
<td>MDCR_EL2</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>0001</td>
<td>010</td>
<td>RW</td>
<td>CPTTR_EL2</td>
<td>CPTTR_EL2</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>0001</td>
<td>011</td>
<td>RW</td>
<td>HSTR_EL2</td>
<td>HSTR_EL2</td>
<td></td>
</tr>
<tr>
<td>op0</td>
<td>op1</td>
<td>CRn</td>
<td>CRm</td>
<td>op2</td>
<td>Access</td>
<td>Mnemonic</td>
<td>Register</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>--------</td>
<td>--------------</td>
<td>----------------</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0001</td>
<td>0001</td>
<td>100</td>
<td>RW</td>
<td>HDFGRTR_EL2</td>
<td>HDFGRTR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0001</td>
<td>0001</td>
<td>101</td>
<td>RW</td>
<td>HFGWTR_EL2</td>
<td>HFGWTR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0001</td>
<td>0001</td>
<td>110</td>
<td>RW</td>
<td>HFGITR_EL2</td>
<td>HFGITR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0001</td>
<td>0001</td>
<td>111</td>
<td>RW</td>
<td>HACR_EL2</td>
<td>HACR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0001</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>ZCR_EL2</td>
<td>ZCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0001</td>
<td>0010</td>
<td>001</td>
<td>RW</td>
<td>TRFCR_EL2</td>
<td>TRFCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0001</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>HCRX_EL2</td>
<td>HCRX_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0001</td>
<td>0010</td>
<td>101</td>
<td>RW</td>
<td>SMPRIMAP_EL2</td>
<td>SMPRIMAP_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0001</td>
<td>0010</td>
<td>110</td>
<td>RW</td>
<td>SMCR_EL2</td>
<td>SMCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0011</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>TTBR0_EL2</td>
<td>TTBR0_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0010</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>TTBR1_EL2</td>
<td>TTBR1_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0010</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>TCR_EL2</td>
<td>TCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0010</td>
<td>0011</td>
<td>000</td>
<td>RW</td>
<td>VTTBR_EL2</td>
<td>VTTBR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0010</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>VTCR_EL2</td>
<td>VTCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0010</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>VNCR_EL2</td>
<td>VNCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0010</td>
<td>0110</td>
<td>000</td>
<td>RW</td>
<td>VSTTBR_EL2</td>
<td>VSTTBR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0010</td>
<td>0110</td>
<td>010</td>
<td>RW</td>
<td>VSTCR_EL2</td>
<td>VSTCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0011</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>DACR32_EL2</td>
<td>DACR32_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0011</td>
<td>0001</td>
<td>100</td>
<td>RW</td>
<td>HDFGRTR_EL2</td>
<td>HDFGRTR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0011</td>
<td>0001</td>
<td>101</td>
<td>RW</td>
<td>HDFGWTR_EL2</td>
<td>HDFGWTR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0011</td>
<td>0001</td>
<td>110</td>
<td>RW</td>
<td>HAFGRTR_EL2</td>
<td>HAFGRTR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0000</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL1</td>
<td>SPSR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0000</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>SPSR_EL1</td>
<td>SPSR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0000</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL2</td>
<td>SPSR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0001</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL2</td>
<td>SPSR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ELR_EL1</td>
<td>ELR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>ELR_EL1</td>
<td>ELR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ELR_EL2</td>
<td>ELR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>ELR_EL2</td>
<td>ELR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>SP_EL1</td>
<td>SP_EL1</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0011</td>
<td>000</td>
<td>RW</td>
<td>SPSR_irq</td>
<td>SPSR_irq</td>
</tr>
</tbody>
</table>
### Table D16-2 Instruction encodings for non-debug System register access (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0011</td>
<td>001</td>
<td>RW</td>
<td>SPSR_abt</td>
<td>SPSR_abt</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>SPSR_und</td>
<td>SPSR_und</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0100</td>
<td>0011</td>
<td>011</td>
<td>RW</td>
<td>SPSR_fiq</td>
<td>SPSR_fiq</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0101</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>IFSR32_EL2</td>
<td>IFSR32_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0101</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>AFSR0_EL2</td>
<td>AFSR0_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0101</td>
<td>0001</td>
<td>001</td>
<td>RW</td>
<td>AFSR1_EL2</td>
<td>AFSR1_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0101</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>ESR_EL1</td>
<td>ESR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0101</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>ESR_EL1</td>
<td>ESR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0110</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>FAR_EL1</td>
<td>FAR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0110</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>FAR_EL1</td>
<td>FAR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0110</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>FAR_EL2</td>
<td>FAR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0110</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>FAR_EL2</td>
<td>FAR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>0110</td>
<td>0000</td>
<td>100</td>
<td>RW</td>
<td>HPFAR_EL2</td>
<td>HPFAR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1001</td>
<td>1001</td>
<td>000</td>
<td>RW</td>
<td>PMSCR_EL2</td>
<td>PMSCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1010</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>MAIR_EL2</td>
<td>MAIR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1010</td>
<td>0011</td>
<td>000</td>
<td>RW</td>
<td>AMAIR_EL2</td>
<td>AMAIR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1010</td>
<td>0100</td>
<td>000</td>
<td>RW</td>
<td>MPAMHCR_EL2</td>
<td>MPAMHCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1010</td>
<td>0100</td>
<td>001</td>
<td>RW</td>
<td>MPAMVPMV_EL2</td>
<td>MPAMVPMV_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1010</td>
<td>0101</td>
<td>000</td>
<td>RW</td>
<td>MPAM2_EL2</td>
<td>MPAM2_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1010</td>
<td>0110</td>
<td>000</td>
<td>RW</td>
<td>MPAMVPM0_EL2</td>
<td>MPAMVPM0_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1010</td>
<td>0110</td>
<td>001</td>
<td>RW</td>
<td>MPAMVPM1_EL2</td>
<td>MPAMVPM1_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1010</td>
<td>0110</td>
<td>010</td>
<td>RW</td>
<td>MPAMVPM2_EL2</td>
<td>MPAMVPM2_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1010</td>
<td>0110</td>
<td>011</td>
<td>RW</td>
<td>MPAMVPM3_EL2</td>
<td>MPAMVPM3_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1010</td>
<td>0110</td>
<td>100</td>
<td>RW</td>
<td>MPAMVPM4_EL2</td>
<td>MPAMVPM4_EL2</td>
</tr>
</tbody>
</table>
### Table D16-2 Instruction encodings for non-debug System register access (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>100</td>
<td>1010</td>
<td>0110</td>
<td>101</td>
<td>RW</td>
<td>MPAMVPM5_EL2</td>
<td>MPAMVPM5_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1010</td>
<td>0110</td>
<td>110</td>
<td>RW</td>
<td>MPAMVPM6_EL2</td>
<td>MPAMVPM6_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1010</td>
<td>0110</td>
<td>111</td>
<td>RW</td>
<td>MPAMVPM7_EL2</td>
<td>MPAMVPM7_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>VBAR_EL2</td>
<td>VBAR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1100</td>
<td>0000</td>
<td>001</td>
<td>RO</td>
<td>RVBAR_EL2</td>
<td>RVBAR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1100</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>RMR_EL2</td>
<td>RMR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1100</td>
<td>0001</td>
<td>001</td>
<td>RW</td>
<td>VDISR_EL2</td>
<td>VDISR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1100</td>
<td>1000</td>
<td>0:m[1:0]</td>
<td>RW</td>
<td>ICH_AP0R&lt;m&gt;_EL2</td>
<td>ICH_AP0R_EL2[]</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1100</td>
<td>1001</td>
<td>0:m[1:0]</td>
<td>RW</td>
<td>ICH_AP1R&lt;m&gt;_EL2</td>
<td>ICH_AP1R_EL2[]</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1100</td>
<td>1001</td>
<td>101</td>
<td>RW</td>
<td>ICC_SRE_EL2</td>
<td>ICC_SRE_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1100</td>
<td>1011</td>
<td>000</td>
<td>RW</td>
<td>ICH_HCR_EL2</td>
<td>ICH_HCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1100</td>
<td>1011</td>
<td>001</td>
<td>RO</td>
<td>ICH_VTR_EL2</td>
<td>ICH_VTR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1100</td>
<td>1011</td>
<td>010</td>
<td>RO</td>
<td>ICH_MISR_EL2</td>
<td>ICH_MISR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1100</td>
<td>1011</td>
<td>011</td>
<td>RO</td>
<td>ICH_EISR_EL2</td>
<td>ICH_EISR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1100</td>
<td>1011</td>
<td>101</td>
<td>RO</td>
<td>ICH_ELRSR_EL2</td>
<td>ICH_ELRSR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1100</td>
<td>1011</td>
<td>111</td>
<td>RW</td>
<td>ICH_VMCR_EL2</td>
<td>ICH_VMCR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1100</td>
<td>110:m[3]m[2:0]</td>
<td>RW</td>
<td>ICH_LR&lt;m&gt;_EL2</td>
<td>ICH_LR_EL2[]</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1101</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>CONTEXTIDR_EL2</td>
<td>CONTEXTIDR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1101</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>TPIDR_EL2</td>
<td>TPIDR_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1101</td>
<td>0000</td>
<td>111</td>
<td>RW</td>
<td>SCXTNUM_EL2</td>
<td>SCXTNUM_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1101</td>
<td>100:m[3]m[2:0]</td>
<td>RW</td>
<td>AMEVCNTVOFF0&lt;m&gt;_EL2</td>
<td>AMEVCNTVOFF0_EL2[]</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1101</td>
<td>101:m[3]m[2:0]</td>
<td>RW</td>
<td>AMEVCNTVOFF1&lt;m&gt;_EL2</td>
<td>AMEVCNTVOFF1_EL2[]</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1110</td>
<td>0000</td>
<td>011</td>
<td>RW</td>
<td>CNTVOFF_EL2</td>
<td>CNTVOFF_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1110</td>
<td>0000</td>
<td>110</td>
<td>RW</td>
<td>CNTPOFF_EL2</td>
<td>CNTPOFF_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1110</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>CNTHCTL_EL2</td>
<td>CNTHCTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1110</td>
<td>0010</td>
<td>000</td>
<td>-</td>
<td>CNTHP_TVAL_EL2</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1110</td>
<td>0010</td>
<td>001</td>
<td>RW</td>
<td>CNTHP_CTL_EL2</td>
<td>CNTHP_CTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1110</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>CNTHP_CVAL_EL2</td>
<td>CNTHP_CVAL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1110</td>
<td>0011</td>
<td>000</td>
<td>-</td>
<td>CNTHV_TVAL_EL2</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1110</td>
<td>0011</td>
<td>001</td>
<td>RW</td>
<td>CNTHV_CTL_EL2</td>
<td>CNTHV_CTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1110</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>CNTHV_CVAL_EL2</td>
<td>CNTHV_CVAL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1110</td>
<td>0100</td>
<td>000</td>
<td>-</td>
<td>CNTHVS_TVAL_EL2</td>
<td>-</td>
</tr>
</tbody>
</table>
### Table D16-2 Instruction encodings for non-debug System register access (continued)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>100</td>
<td>1110</td>
<td>0100</td>
<td>001</td>
<td>RW</td>
<td>CNTHVS_CTL_EL2</td>
<td>CNTHVS_CTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1110</td>
<td>0100</td>
<td>010</td>
<td>RW</td>
<td>CNTHVS_CV_AL_EL2</td>
<td>CNTHVS_CV_AL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1110</td>
<td>0101</td>
<td>000</td>
<td>-</td>
<td>CNTHPS_TV_AL_EL2</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1110</td>
<td>0101</td>
<td>001</td>
<td>RW</td>
<td>CNTHPS_CTL_EL2</td>
<td>CNTHPS_CTL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>1110</td>
<td>0101</td>
<td>010</td>
<td>RW</td>
<td>CNTHPS_CV_AL_EL2</td>
<td>CNTHPS_CV_AL_EL2</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0001</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SCTL_EL1</td>
<td>SCTL_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0001</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>CPACR_EL1</td>
<td>CPACR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0001</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>ZCR_EL1</td>
<td>ZCR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0001</td>
<td>0010</td>
<td>001</td>
<td>RW</td>
<td>TRFCR_EL1</td>
<td>TRFCR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0001</td>
<td>0010</td>
<td>110</td>
<td>RW</td>
<td>SMCR_EL1</td>
<td>SMCR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0010</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>TTBR0_EL1</td>
<td>TTBR0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0010</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>TTBR1_EL1</td>
<td>TTBR1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0010</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>TCR_EL1</td>
<td>TCR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SPSR_EL1</td>
<td>SPSR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ELR_EL1</td>
<td>ELR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0101</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>AFSL_EL1</td>
<td>AFSL_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0101</td>
<td>0001</td>
<td>001</td>
<td>RW</td>
<td>AFSL0_EL1</td>
<td>AFSL0_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0101</td>
<td>0001</td>
<td>010</td>
<td>RW</td>
<td>AFSL1_EL1</td>
<td>AFSL1_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0101</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>ESR_EL1</td>
<td>ESR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0110</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>TFSR_EL1</td>
<td>TFSR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>0110</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>FAR_EL1</td>
<td>FAR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>1001</td>
<td>1001</td>
<td>000</td>
<td>RW</td>
<td>PMSCR_EL1</td>
<td>PMSCR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>1010</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>MAIR_EL1</td>
<td>MAIR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>1010</td>
<td>0011</td>
<td>000</td>
<td>RW</td>
<td>AMAIR_EL1</td>
<td>AMAIR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>1010</td>
<td>0101</td>
<td>000</td>
<td>RW</td>
<td>MPAML_EL1</td>
<td>MPAML_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>1100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>VBAR_EL1</td>
<td>VBAR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>1101</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>CONTEXTIDR_EL1</td>
<td>CONTEXTIDR_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>1101</td>
<td>0000</td>
<td>111</td>
<td>RW</td>
<td>SCXTNUM_EL1</td>
<td>SCXTNUM_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>1110</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>CNTKCTL_EL1</td>
<td>CNTKCTL_EL1</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>1110</td>
<td>0010</td>
<td>000</td>
<td>-</td>
<td>CNTPK_TV_AL_EL0</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>1110</td>
<td>0010</td>
<td>001</td>
<td>RW</td>
<td>CNTPK_CTL_EL0</td>
<td>CNTPK_CTL_EL0</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>1110</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>CNTPK_CV_AL_EL0</td>
<td>CNTPK_CV_AL_EL0</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>1110</td>
<td>0011</td>
<td>000</td>
<td>-</td>
<td>CNTV_TV_AL_EL0</td>
<td>-</td>
</tr>
<tr>
<td>op0</td>
<td>op1</td>
<td>CRn</td>
<td>CRm</td>
<td>op2</td>
<td>Access</td>
<td>Mnemonic</td>
<td>Register</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-------</td>
<td>-------</td>
<td>-----</td>
<td>--------</td>
<td>-------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>1110</td>
<td>0011</td>
<td>001</td>
<td>RW</td>
<td>CNTV_CTL_EL0</td>
<td>CNTV_CTL_EL0</td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>1110</td>
<td>0011</td>
<td>010</td>
<td>RW</td>
<td>CNTV_CVAL_EL0</td>
<td>CNTV_CVAL_EL0</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0001</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>SCTLR_EL3</td>
<td>SCTLR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0001</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>ACTLR_EL3</td>
<td>ACTLR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0001</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>SCR_EL3</td>
<td>SCR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0001</td>
<td>0001</td>
<td>001</td>
<td>RW</td>
<td>SDER32_EL3</td>
<td>SDER32_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0001</td>
<td>0001</td>
<td>010</td>
<td>RW</td>
<td>CPTR_EL3</td>
<td>CPTR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0001</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>ZCR_EL3</td>
<td>ZCR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0001</td>
<td>0010</td>
<td>110</td>
<td>RW</td>
<td>SMCR_EL3</td>
<td>SMCR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0010</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>TTBR0_EL3</td>
<td>TTBR0_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0010</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>TCR_EL3</td>
<td>TCR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>SPSR_EL3</td>
<td>SPSR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0100</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>ELR_EL3</td>
<td>ELR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0100</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>SP_EL2</td>
<td>SP_EL2</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0101</td>
<td>0001</td>
<td>000</td>
<td>RW</td>
<td>AFSR0_EL3</td>
<td>AFSR0_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0101</td>
<td>0001</td>
<td>001</td>
<td>RW</td>
<td>AFSR1_EL3</td>
<td>AFSR1_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0101</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>ESR_EL3</td>
<td>ESR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0101</td>
<td>0110</td>
<td>000</td>
<td>RW</td>
<td>TFSR_EL3</td>
<td>TFSR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0110</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>FAR_EL3</td>
<td>FAR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>0110</td>
<td>0000</td>
<td>101</td>
<td>RW</td>
<td>MFAR_EL3</td>
<td>MFAR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>1010</td>
<td>0010</td>
<td>000</td>
<td>RW</td>
<td>MAIR_EL3</td>
<td>MAIR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>1010</td>
<td>0011</td>
<td>000</td>
<td>RW</td>
<td>AMAIR_EL3</td>
<td>AMAIR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>1010</td>
<td>0101</td>
<td>000</td>
<td>RW</td>
<td>MPAM3_EL3</td>
<td>MPAM3_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>1100</td>
<td>0000</td>
<td>000</td>
<td>RW</td>
<td>VBAR_EL3</td>
<td>VBAR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>1100</td>
<td>0000</td>
<td>001</td>
<td>RW</td>
<td>RVBAR_EL3</td>
<td>RVBAR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>1100</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>RMR_EL3</td>
<td>RMR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>1100</td>
<td>1100</td>
<td>100</td>
<td>RW</td>
<td>ICC_CTLR_EL3</td>
<td>ICC_CTLR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>1100</td>
<td>1100</td>
<td>101</td>
<td>RW</td>
<td>ICC_SRE_EL3</td>
<td>ICC_SRE_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>1100</td>
<td>1100</td>
<td>111</td>
<td>RW</td>
<td>ICC_IGRPEN1_EL3</td>
<td>ICC_IGRPEN1_EL3</td>
</tr>
</tbody>
</table>
About the GIC System registers

From version 3.0 of the GIC architecture specification, the specification defines three groups of System registers, identified by the prefix of the register name:

ICC_  GIC physical CPU interface System registers.
ICH_  GIC virtual interface control System registers.
ICV_  GIC Virtual CPU interface System registers.

Note

These registers are in addition to the GIC memory-mapped register groups GICC_, GICD_, GICH_, GICR_, GICV_, and GITS_.

When implemented, the GIC System registers form part of an Arm processor implementation, and therefore these registers are included in the register summaries. However, the registers are defined only in the GIC Architecture Specification.

As Table D16-2 on page D16-5523 shows, the ICV_* registers have the same \{op0, op1, CRn, CRm, op2\} encodings as the corresponding ICC_* registers. For these encodings, GIC register configuration fields determine which register is accessed.

For more information, see the ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0 (ARM IHI 0069).

D16.3.2  Reserved encodings for IMPLEMENTATION DEFINED registers

The System register encoding space with op0==0b11 reserves the following encodings for IMPLEMENTATION DEFINED registers:

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Access</th>
<th>Mnemonic</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>110</td>
<td>1101</td>
<td>0000</td>
<td>010</td>
<td>RW</td>
<td>TPIDR_EL3</td>
<td>TPIDR_EL3</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>1101</td>
<td>0000</td>
<td>111</td>
<td>RW</td>
<td>SCXTNUM_EL3</td>
<td>SCXTNUM_EL3</td>
</tr>
<tr>
<td>11</td>
<td>111</td>
<td>1110</td>
<td>0010</td>
<td>000</td>
<td>-</td>
<td>CNTPS_TVAL_EL1</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>111</td>
<td>1110</td>
<td>0010</td>
<td>001</td>
<td>RW</td>
<td>CNTPS_CTL_EL1</td>
<td>CNTPS_CTL_EL1</td>
</tr>
<tr>
<td>11</td>
<td>111</td>
<td>1110</td>
<td>0010</td>
<td>010</td>
<td>RW</td>
<td>CNTPS_CVAL_EL1</td>
<td>CNTPS_CVAL_EL1</td>
</tr>
<tr>
<td>11</td>
<td>op1[2:0]</td>
<td>1x11</td>
<td>Cm[3:0]</td>
<td>op2[2:0]</td>
<td>-</td>
<td>S3_&lt;op1&gt;<em>&lt;Cn&gt;</em>&lt;Cm&gt;_&lt;op2&gt;</td>
<td>-</td>
</tr>
</tbody>
</table>

The value of L defines the access type and the use of Rt as follows:

0  Write the value in Rt to the IMPLEMENTATION DEFINED register.
1  Read the value of the IMPLEMENTATION DEFINED register to Rt.

For more information about these encodings, see S3_<op1>_<Cn>_<Cm>_<op2>, IMPLEMENTATION DEFINED registers on page D17-6153. As that section describes, any IMPLEMENTATION DEFINED registers are accessed in a similar way to architecturally-defined System registers, using MRS and MSR instructions, see:

- MRS on page C6-1707.
- MSR (immediate) on page C6-1708.
- MSR (register) on page C6-1712.
The Arm architecture guarantees not to define any register name prefixed with *IMP_* as part of the standard Arm architecture.

**Note**

Arm strongly recommends that any register names created in the IMPLEMENTATION DEFINED register spaces be prefixed with *IMP_* and postfixed with *ELx*, where appropriate.
AArch64 System Register Encoding
D16.3 op0==0b11, Moves to and from non-debug System registers, Special-purpose registers
Chapter D17
AArch64 System Register Descriptions

This chapter defines the AArch64 System registers. It contains the following sections:

- About the AArch64 System registers on page D17-5546.
- General system control registers on page D17-5555.
- Debug registers on page D17-6393.
- Trace registers on page D17-6519.
- Performance Monitors registers on page D17-6740.
- Activity Monitors registers on page D17-6816.
- Statistical Profiling Extension registers on page D17-6858.
- Branch Record Buffer Extension registers on page D17-6912.
- RAS registers on page D17-6956.
- Generic Timer registers on page D17-7014.
- Realm Management Extension registers on page D17-7004.
D17.1 About the AArch64 System registers

The following sections describe common features of the AArch64 registers:

- Fixed values in the System register descriptions.
- General behavior of accesses to the AArch64 System registers.
- Principles of the ID scheme for fields in ID registers on page D17-5551.

D17.1.1 Fixed values in the System register descriptions

See Fixed values in AArch64 instruction and System register descriptions on page C2-243. This section defines how the glossary terms RAZ, RES0, RAO, and RES1 can be represented in the System register descriptions.

D17.1.2 General behavior of accesses to the AArch64 System registers

The following subsections give general information about the behavior of accesses to the System registers:

- Reset behavior of AArch64 System registers.
- Synchronization requirements for AArch64 System registers on page D17-5547.

Reset behavior of AArch64 System registers

Reset values apply only to RW registers and fields, however:

- Some RO registers or fields, including feature ID registers and some status registers or register fields, always return a known value.

- Some RW and RO registers or register fields return status information about the PE. Unless the register description indicates that the value is UNKNOWN on reset, a read of the register immediately after a reset returns valid information.

- Some RW and RO registers and fields are aliases of other registers or fields. In these cases, the reset behavior of the aliased register or field determines the value returned by a read of the register immediately after a reset.

- WO registers that only have an effect on writes do not have meaningful reset values. However, an access to a WO register might affect underlying state, and that state might have a defined reset value.

- IMPLEMENTATION DEFINED registers have IMPLEMENTATION DEFINED reset behavior.

After a reset, only a limited subset of the PE state is guaranteed to be set to defined values. Also, for debug and trace System registers, reset requirements must take account of different levels of reset. For more information about the reset behavior of System registers when the PE resets into an Exception level that is using AArch64, see:

- Reset behavior on page D1-4674.
- The appropriate Trace architecture specification, for the Trace System registers.

For a PE reset into an Exception level that is using AArch64, the architecture defines which AArch64 System registers have a defined reset value, and when that defined reset value applies. The register descriptions include this information, and Reset behavior on page D1-4674 summarizes these architectural requirements. Otherwise, RW registers that have a meaningful reset value reset to an architecturally UNKNOWN value.

Note

When the PE resets into an Exception level that is using AArch32, no PE state that relates to execution in AArch64 state is accessible until another reset causes the Execution state to change to AArch64. Therefore, on a reset into AArch32 state, PE state that relates only to execution in AArch64 state cannot have a meaningful reset value.
Pseudocode description of resetting System registers

The `AArch64.ResetSystemRegisters()` pseudocode function resets all System registers, and register fields, that have defined reset values, as described in this section and `Reset behavior` on page D1-4674.

Note

For debug and trace System registers, this function resets registers as defined for the appropriate level of reset.

Synchronization requirements for AArch64 System registers

Reads of the System registers can occur out of order with respect to earlier instructions executed on the same PE, provided that both:

- Any data dependencies between the instructions, including read-after-read dependencies, are respected.
- The reads to the register do not occur earlier than the most recent Context synchronization event to its architectural position in the instruction stream.

Note

In particular, the values read from System registers that hold self-incrementing counts, such as the Performance Monitors counters or the Generic Timer counter or timers, could be accessed from any time after the previous Context synchronization event. For example, where a memory access is used to communicate a read of such a counter, an ISB must be inserted between the read of the memory location that is known to have returned its data, either as a result of a condition on that data or of the read having completed, and the read of the counter, if it is necessary that the counter returns a count value after the memory communication.

Direct writes using the instructions in Table D16-2 on page D16-5523 require synchronization before software can rely on the effects of changes to the System registers to affect instructions appearing in program order after the direct write to the System register. Direct writes to these registers are not allowed to affect any instructions appearing in program order before the direct write. The only exceptions are:

- All direct writes to the same register, that use the same encoding for that register, are guaranteed to occur in program order relative to each other
- All direct writes to a register occur in program order with respect to all direct reads to the same register using the same encoding.
- Any System register access that an Arm Architecture Specification or equivalent specification defines as not requiring synchronization.
- When FEAT_BRBE is implemented, execution of BRB INJ does not require explicit synchronization to use the result of direct writes to the Branch record injection data registers in program order before BRB INJ.

Explicit synchronization occurs as a result of a Context synchronization event, which is one of the following events:

- Execution of an ISB instruction.
- Exception entry, if FEAT_ExS is not implemented, or if FEAT_ExS is implemented and defines that exception entries to this Exception level are context synchronization events.
- Exception return, if FEAT_ExS is not implemented, or if FEAT_ExS is implemented and defines that exception returns from this Exception level are context synchronization events.
- Execution of a DCPs instruction in Debug state.
- Execution of a DRPS instruction in Debug state.
- Exit from Debug state.

Note

The ISB and exception entry events are applicable both in Debug state and in Non-debug state.
Conceptually, explicit synchronization occurs as the first step of each of these events, so that if the event uses state that has previously been changed but was not synchronized by the time of the event, the event is guaranteed to use the state as if it had been synchronized.

--- Note ---

This explicit synchronization applies as the first step of the execution of the events, and does not apply to any effect of System registers that apply to the fetch and decode of the instructions that cause these events, such as breakpoints or changes to the translation table.

In addition, any system instructions that cause a write to a System register must be synchronized before the result is guaranteed to be visible to subsequent direct reads of that System register.

Direct reads to any one of the following registers, using the same encoding, occur in program order relative to each other:

- ISR_EL1.
- The Generic Timer registers, that is, CNTPCT_EL0 and CNTVCT_EL0, and the Counter registers CNTP_TV_AL_EL0, CNTV_TV_AL_EL0, CNTHP_TV_AL_EL2, and CNTPS_TV_AL_EL1.
- DBGCLAIMCLR_EL1.
- The PMU Counters, that is, PMCCNTR_EL0, PMEVCNTR<e> EL0, PMXEVCNTR_EL0, PMOVSCLR_EL0, and PMOVSET_EL0.
- The Debug Communications Channel registers, that is, DBGDTRRX_EL0, DBGDTR_EL0, and MDCCSR_EL0.

All other direct reads of System registers can occur in any order if synchronization has not been performed.

**Table D17-1** describes the synchronization requirements between two successive read or write accesses to the same register, where the ordering of the read or write accesses is:

1. Program order, in the event that both the reads or writes are caused by an instruction executed on this PE, other than one caused by a memory access by this PE.
2. The order of arrival of asynchronous reads and writes at the PE relative to the execution of instructions that cause reads or writes.
3. The order of arrival of asynchronous reads and writes at the PE relative to each other.

<table>
<thead>
<tr>
<th>First read/write</th>
<th>Second read/write</th>
<th>Synchronization requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct read</td>
<td>Direct read</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Direct write</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Indirect read</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Indirect write</td>
<td>None, see <em>Notes on page D17-5549</em></td>
</tr>
<tr>
<td>Direct write</td>
<td>Direct read</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Direct write</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Indirect read</td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>Indirect write</td>
<td>None, see <em>Notes on page D17-5549</em></td>
</tr>
<tr>
<td>Indirect read</td>
<td>Direct read</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Direct write</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Indirect read</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Indirect write</td>
<td>None</td>
</tr>
</tbody>
</table>
The terms Direct read, Direct write, Indirect read, and Indirect write, as used in Table D17-1 on page D17-5548, are defined as follows:

**Direct read** Where software uses an MRS system register access instruction to read that register into a general purpose register.

Where a direct read of a register has a side-effect that changes the contents of a register, the effect of a direct read on that register is defined to be an indirect write. In this case, the indirect write is only guaranteed to have occurred, and be visible to subsequent direct or indirect reads or writes, if synchronization is performed after the direct read.

**Direct write** Where software uses an MSR (register) access instruction to write to that register from a general purpose register.

Where a direct write to a register has an effect on the register that means that the value in the register is not always the last value that is written (as is the case with set and clear registers), the effect of a direct write on that register is defined to be an indirect write. In this case, the indirect write is only guaranteed to be visible to subsequent direct or indirect reads or writes if synchronization is performed after the direct write and before the subsequent direct or indirect reads or writes.

**Indirect read** Where an instruction uses a System register to establish operating conditions for the instruction, for example, the TTBR_ELx address or whether memory accesses are forced to be Non-cacheable. This includes situations where the contents of one System register selects what value is read or written using a different register. Indirect reads also include reads of the System register by external agents such as debuggers. Where an indirect read of a register has a side-effect that changes the contents of that register, that is defined to be an indirect write.

**Indirect write** Where a System register is written as the consequence of some other instruction, exception, operation, or by the asynchronous operation of an external agent, including the passage of time as seen in counters, timers, or performance counters, the assertion of interrupts, or writes from an external debugger.

**Note** Since an exception is context synchronizing, registers such as the Exception Syndrome registers that are indirectly written as part of exception entry do not require additional synchronization.

Where a direct read or write to a register is followed by an indirect write caused by an external agent, autonomous asynchronous event, or as a result of memory mapped write, synchronization is required to guarantee the order of those two accesses.

Where an indirect write caused by a direct write is followed by an indirect write caused by an external agent, autonomous asynchronous event, or as a result of memory mapped write, synchronization is required to guarantee the order of those two indirect accesses.

Where a direct read to one register causes a bit or field in a different register (or the same register using a different encoding) to be updated, the change to the different register (or same register using a different encoding) is defined to be an indirect write. In this case, the indirect write is only guaranteed to be visible to subsequent direct or indirect reads or writes if synchronization is performed after the direct read and before the subsequent direct or indirect reads or writes.

---

**Table D17-1 Synchronization requirements (continued)**

<table>
<thead>
<tr>
<th>First read/write</th>
<th>Second read/write</th>
<th>Synchronization requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indirect write</td>
<td>Direct read</td>
<td>Required, see Notes</td>
</tr>
<tr>
<td>Direct write</td>
<td>None, see Notes</td>
<td></td>
</tr>
<tr>
<td>Indirect read</td>
<td>Required, see Notes</td>
<td></td>
</tr>
<tr>
<td>Indirect write</td>
<td>None, see Notes</td>
<td></td>
</tr>
</tbody>
</table>

---

**Notes**

The terms Direct read, Direct write, Indirect read, and Indirect write, as used in Table D17-1 on page D17-5548, are defined as follows:
Where a direct write to one register causes a bit or field in a different register (or the same register using a different encoding) to be updated as a side-effect of that direct write (as opposed to simply being a direct write to the different encoding), the change to the different register (or same register using a different encoding) is defined to be an indirect write. In this case, the indirect write is only guaranteed to be visible to subsequent direct or indirect reads or writes if synchronization is performed after the direct write and before the subsequent direct or indirect reads or writes.

Where indirect writes are caused by the actions of external agents such as debuggers, or by memory-mapped reads or writes by the PE, then an indirect write by that agent and mechanism to a register, followed by an indirect read by that agent and mechanism to the same register using the same address, does not require synchronization.

Where an indirect write occurs as a side-effect of an access, this happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect.

Indirect writes caused by external agents, autonomous asynchronous events, or as a result of memory-mapped writes, to the registers shown in Table D17-2, are required to be observable to:

- Direct reads in finite time without explicit synchronization.
- Subsequent indirect reads without explicit synchronization.

Without explicit synchronization to guarantee the order of the accesses, where the same register is accessed by two or more of a System register access instruction, and external agent, and autonomous asynchronous event, or as a result of a memory-mapped access, the behavior must be as if the accesses occurred atomically and in any order. This applies even if the accesses occur simultaneously.

### Table D17-2 Registers with a guarantee of observability, VMSAv8-64

<table>
<thead>
<tr>
<th>Registers</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISR_EL1</td>
<td>Interrupt Status Register</td>
</tr>
<tr>
<td>DBGCLAIMCLR_EL1, DBGCLAIMSET_EL1</td>
<td>Debug CLAIM registers</td>
</tr>
<tr>
<td>CNTPCT_EL0, CNTVCT_EL0, CNTP_TV_AL_EL0, CNTV_TV_AL_EL0, CNTHP_TV_AL_EL2, CNTPS_TV_AL_EL1</td>
<td>Generic Timer registers</td>
</tr>
<tr>
<td>PMCCNTR_EL0, PMEVCNTR&lt;n&gt;_EL0, PMXEVCNTR_EL0, PMOVSCLR_EL0, PMOVSSET_EL0</td>
<td>PMU Counters</td>
</tr>
<tr>
<td>DBGDTRTX_EL0, DBGDTRRX_EL0, DBGDTR_EL0, and the DCC flags in MDCCSR_EL0 and EDSCR</td>
<td>Debug Communication Channel registers</td>
</tr>
<tr>
<td>EDSR.PipeAdv</td>
<td>External Debug Status and Control Register PipeAdv field</td>
</tr>
</tbody>
</table>

In addition to the requirements shown in Table D17-2 on page D17-5550:

- Indirect writes to the following registers as a result of memory-mapped writes, including accesses by external agents, are required to be observable to the indirect read made in determining the response to a subsequent memory-mapped access without explicit synchronization:
  - OSLAR_EL1. OSLAR_EL1 is indirectly read to determine whether the subsequent access is permitted.
  - EDLAR, if implemented. EDLAR is indirectly read to determine whether a subsequent write or side-effect of an access is ignored.

  **Note**

  This requirement is stricter than the general requirement for the observability of indirect writes.

- The requirement that an indirect write to the registers in Table D17-2 on page D17-5550 is observable to direct reads in finite time does not imply that all observers will observe the indirect write at the same time.
For example, an increment of the system counter is an autonomous asynchronous event that performs an indirect write to the counter. This asynchronous event might generate a timer interrupt request, resulting in a Context synchronization event. When a GIC is used, the timer interrupt might arrive at the GIC after the PE has taken an interrupt request from another source, but before software reads the current interrupt ID from the GIC. This means that the GIC might identify the timer interrupt as the current interrupt. Software must not assume that a subsequent direct read of the counter register is guaranteed to observe the updated value of that register.

Although this example uses the counter-timer registers, it applies equally to other registers that might be linked to interrupt requests, including the PMU and Statistical Profiling status registers.

- When the PE is in Debug state, there are synchronization requirements for the Debug Communication Channel and Instruction Transfer registers. See DCC and ITR access in Debug state on page H4-10313.

**Note**

- The provision of explicit synchronization requirements to System registers is provided to allow the direct access to these registers to be implemented in a small number of cycles, and that updates to multiple registers can be performed quickly with the synchronization penalty being paid only when the updates have occurred.

- Since toolkits might use registers such as the thread-local storage registers within compiled code, it is recommended that access to these registers is implemented to take a small number of cycles.

- While no synchronization is required between a direct write and a direct read, or between a direct read and an indirect write, this does not imply that a direct read causes synchronization of a previous direct write. That is, the sequence direct write → direct read → indirect read, with no intervening context synchronization, does not guarantee that the indirect read observes the result of the direct write.

- If FEAT_MTE2 is implemented, a DSB instruction over the Non-shareable domain or an exception entry to ELy with SCTLR_ELy.ITFSB = 0b1 is required between an indirect write to TFSRE0_EL1, or any TFSR_ELx accessible at ELy, and a direct read or direct write of that register.

**D17.1.3 Principles of the ID scheme for fields in ID registers**

The Arm architecture specifies a number of ID registers that are characterized as comprising a set of 4-bit ID fields. Each ID field identifies the presence, and possibly the level of support for, a particular feature in an implementation of the architecture. These fields follow an architectural model that aids their use by software and provides future compatibility. This section describes that model. ID registers to which this scheme applies on page D17-5553 identifies the set of ID registers.

A small number of ID fields do not follow the scheme described in this section. In these cases, the field description states that it does not follow this scheme.

**Note**

- The ID fields described here are distinct from register fields that enumerate the number of resources, such as the number of breakpoints, watchpoints, or performance monitors, or the amount of memory.

- ID fields that do not follow this scheme include the ID_AA64DFR0_EL1.PMUVer, ID_DFR0_EL1.PerfMon, ID_DFR0.PerfMon and EDDFR.PMUVer fields, see Alternative ID scheme used for the Performance Monitors Extension version on page D17-5553.

- The presence of an ID field for a feature does not imply that the feature is optional.

To provide forward compatibility, software can rely on the features of these fields that are described in this section.
The ID fields, which are either signed or unsigned, use increasing numerical values to indicate increases in functionality. Therefore, if a value of $0x1$ indicates the presence of some instructions, then the value $0x2$ will indicate the presence of those instructions plus some additional instructions or functionality. This means software can be written in the form:

```c
if (value >= number) {
    // do something that relies on the value of the feature
}
```

For ID fields where the value $0x0$ defines that a feature is not present, the field holds an unsigned value. This covers the vast majority of such fields.

In a few cases, the architecture has been changed to permit implementations to exclude a feature that has previously been required and for which no ID field has been defined. In these cases, a new ID field is defined and:

- The field holds a signed value.
- The field value $0xF$ indicates that the feature is not implemented.
- The field value $0x0$ indicates that the feature is implemented.
- Software that depends on the feature can use the test:
  ```c
  if value >= 0 {
    // Software features that depend on the presence of the hardware feature
  }
  ```

In some cases, it has been decided retrospectively that the increase in functionality between two consecutive numerical values is too great, and it is desirable to permit an intermediate degree of functionality, and the means to discover this. This is done by the introduction of a fractional field that both:

- Is referred to in the definition of the original field.
- Applies only when the original field is at the lower value of the step.

In principle, a fractional field can be used for two different fractional steps, with different meanings associated with each of these steps. For this reason, a fractional field must be interpreted in the context of the field to which it relates and the value of that field. Example D17-1 shows the use of such a field.

### Example D17-1 Example of the use of a fractional field

For a field describing some class of functionality:

- The value $0x1$ was defined as indicating that item A is present.
- The value $0x2$ was defined as indicating that items B and C are present, in addition to item A.

Subsequently, it might be necessary to introduce a second ID field to indicate that A and B only are present. This new field is a fractional field, and might be defined as having the value $0x1$ when A and B only are present. This fractional field is valid only when the original ID field has the value $0x1$.

This approach means that:

- Software that depends on the test if (value $>= 0x2$) can rely on features A, B, and C being present,
- Software that depends on the test if (value $>= 0x1$) can rely on feature A being present.
- If new software needs to check only that features A and B are present, then it can test:
  ```c
  if (value $>= 0x2$ || (value $== 0x1$ && fractional_value $>= 0x1$)) {
    // Software features that depend on A and B only
  }
  ```

A fractional field uses the same approach of increasing numerical values indicating increasing functionality, and the fractional approach can also be applied recursively to fractional fields.

Unused ID fields, and fractional fields that are not applicable, are $RES0$ to allow their future use when features, or fractional implementation options, are added.
**ID registers to which this scheme applies**

This scheme applies to the following registers:

**AArch64 System registers**

- The AArch64 views of the AArch32 feature ID registers given by:
  - The AArch32 Auxiliary Feature register ID_AFR0_EL1.
  - The AArch32 Processor Feature registers ID_PFR0_EL1 and ID_PFR1_EL1.
  - The AArch32 Debug Feature register ID_DFR0_EL1.
  - The AArch32 Memory Model Feature registers ID_MMFR0_EL1, ID_MMFR1_EL1, ID_MMFR2_EL1, ID_MMFR3_EL1, and ID_MMFR4_EL1.
  - The AArch32 Instruction Set Attribute registers ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.
  - The AArch32 Media and VFP Feature registers MVFR0_EL1, MVFR1_EL1, and MVFR2_EL1.
- The AArch64 Auxiliary Feature registers ID_AA64AFR0_EL1 and ID_AA64AFR1_EL1.
- The AArch64 Processor Feature registers ID_AA64PFR0_EL1 and ID_AA64PFR1_EL1.
- The AArch64 Debug Feature registers ID_AA64DFR0_EL1 and ID_AA64DFR1_EL1.
- The AArch64 Memory Model Feature registers ID_AA64MMFR0_EL1, ID_AA64MMFR1_EL1, and ID_AA64MMFR2_EL1.
- The AArch64 Instruction Set Attribute registers ID_AA64ISAR0_EL1 and ID_AA64ISAR1_EL1.

**AArch32 System registers**

- The AArch32 Auxiliary Feature register ID_AFR0.
- The AArch32 Processor Feature registers ID_PFR0 and ID_PFR1.
- The AArch32 Debug Feature register ID_DFR0.
- The AArch32 Memory Model Feature registers ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, and ID_MMFR4.
- The AArch32 Instruction Set Attribute registers ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.
- The AArch32 Media and FP Feature registers MVFR0, MVFR1, and MVFR2.

**Memory-mapped registers**

- The External Debug Processor Feature register EDPFR.
- The External Debug Feature register EDDFR.

**Alternative ID scheme used for the Performance Monitors Extension version**

The ID_AA64DFR0_EL1.PMUVer, ID_DFR0_EL1.PerfMon, ID_DFR0.PerfMon and EDDFR.PMUVer fields, that identify the version of the Performance Monitors Extension, do not follow the standard ID scheme. Software must treat these fields as follows:

- The value 0xF indicates that the Arm-architected Performance Monitors Extension is not implemented.
- If the field value is not 0xF the field is treated as an unsigned value, as described for the standard ID scheme.

This means that software that depends on the implementation of a particular version of the Arm Performance Monitors Extension must be written in the form:

```c
if (value != 0xF and value >= number) {
    // do something that relies on version 'number' of the feature
}
```

For these fields, Arm deprecates use of the value 0xF in new implementations.
Alternative ID scheme used for ID_AA64MMFR0_EL1 stage 2 granule sizes

The ID_AA64MMFR0_EL1.TGran4_2, ID_AA64MMFR0_EL1.TGran16_2 and ID_AA64MMFR0_EL1.TGran64_2 fields that identify the memory translation stage 2 granule size, do not follow the standard ID scheme. Software must treat these fields as follows:

- The value 0x0 indicates that support is identified by another field.
- If the field value is not 0x0, the value indicates the level of support provided.

This means that software should use a test of the form:

```c
if (field != 0 and field > value) {
    // do something that relies on the value of the feature
}
```
D17.2 General system control registers

This section lists the System registers in AArch64 that are not part of one of the other listed groups.
D17.2.1 ACCDATA_EL1, Accelerator Data

The ACCDATA_EL1 characteristics are:

**Purpose**

Holds the lower 32 bits of the data that is stored by an ST64BV0, Single-copy atomic 64-byte EL0 store instruction.

**Configurations**

This register is present only when FEAT_LS64_ACCDATA is implemented. Otherwise, direct accesses to ACCDATA_EL1 are UNDEFINED.

**Attributes**

ACCDATA_EL1 is a 64-bit register.

**Field descriptions**

**Bits [63:32]**

Reserved, RES0.

**ACCDATA, bits [31:0]**

Accelerator Data field. Holds bits[31:0] of the data that is stored by an ST64BV0 instruction.

**Accessing ACCDATA_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ACCDATA_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.ADEn == '0' then
        UNDEFINED;
    elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.nACCDATA_EL1 == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3.ADEn == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = ACCDATA_EL1;
    else
        PSTATE.EL == EL2 then
            if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.ADEn == '0' then
                ...
UNDEFINED;
elsif HaveEL(EL3) && SCR_EL3.ADEn == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = ACCDATA_EL1;
eelsif PSTATE.EL == EL3 then
  X[t, 64] = ACCDATA_EL1;

**MSR ACCDATA_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
eelsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "El3 trap priority when SDD == '1'" && SCR_EL3.ADEn == '0' then
    UNDEFINED;
eelsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.nACCDATA_EL1 == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
eelsif HaveEL(EL3) && SCR_EL3.ADEn == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
eelsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "El3 trap priority when SDD == '1'" && SCR_EL3.ADEn == '0' then
    UNDEFINED;
eelsif HaveEL(EL3) && SCR_EL3.ADEn == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
eelsif PSTATE.EL == EL3 then
  ACCDATA_EL1 = X[t, 64];
eelsif PSTATE.EL == EL3 then
  ACCDATA_EL1 = X[t, 64];
eelsif PSTATE.EL == EL3 then
  ACCDATA_EL1 = X[t, 64];
D17.2.2 ACTLR_EL1, Auxiliary Control Register (EL1)

The ACTLR_EL1 characteristics are:

### Purpose

Provides IMPLEMENTATION DEFINED configuration and control options for execution at EL1 and EL0.

--- Note ---

Arm recommends the contents of this register have no effect on the PE when HCR_EL2.E2H, TGE is [1, 1], and instead the configuration and control fields are provided by the ACTLR_EL2 register. This avoids the need for software to manage the contents of these register when switching between a Guest OS and a Host OS.

### Configurations

AArch64 System register ACTLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register ACTLR[31:0].

AArch64 System register ACTLR_EL1 bits [63:32] are architecturally mapped to AArch32 System register ACTLR2[31:0].

### Attributes

ACTLR_EL1 is a 64-bit register.

### Field descriptions

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>31-0</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing ACTLR_EL1

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ACTLR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TACR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = NVMem[0x118];
    end
end
else
    \(X[t, 64] = ACTLR\_EL1;\)
elsif PSTATE.EL == EL2 then
    \(X[t, 64] = ACTLR\_EL1;\)
elsif PSTATE.EL == EL3 then
    \(X[t, 64] = ACTLR\_EL1;\)

**MSR ACTLR\_EL1, \(<Xt>\)**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR\_EL2.TACR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR\_EL2.<NV2,NV> == '11' then
        NVMem[0x118] = \(X[t, 64]\);
    else
        ACTLR\_EL1 = \(X[t, 64]\);
    endif
elsif PSTATE.EL == EL2 then
    ACTLR\_EL1 = \(X[t, 64]\);
elsif PSTATE.EL == EL3 then
    ACTLR\_EL1 = \(X[t, 64]\);
D17.2.3 ACTLR_EL2, Auxiliary Control Register (EL2)

The ACTLR_EL2 characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED configuration and control options for EL2.

--- **Note** ---

Arm recommends the contents of this register are updated to apply to EL0 when HCR_EL2.E2H, TGE\} is {1, 1}, gaining configuration and control fields from the ACTLR_EL1. This avoids the need for software to manage the contents of these register when switching between a Guest OS and a Host OS.

---

**Configurations**

AArch64 System register ACTLR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HACTLR[31:0].

AArch64 System register ACTLR_EL2 bits [63:32] are architecturally mapped to AArch32 System register HACTLR2[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

ACTLR_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>31-0</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [63:0]**

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing ACTLR_EL2**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ACTLR_EL2**

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>
```

```c
if PSTATE.EL == EL0 then
  UNDEFINED;
else if PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
```
else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ACTLR_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = ACTLR_EL2;

**MSR ACTLR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    ACTLR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    ACTLR_EL2 = X[t, 64];
D17.2.4 ACTLR_EL3, Auxiliary Control Register (EL3)

The ACTLR_EL3 characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED configuration and control options for EL3.

**Configurations**

This register is present only when EL3 is implemented. Otherwise, direct accesses to ACTLR_EL3 are UNDEFINED.

**Attributes**

ACTLR_EL3 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>31-0</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing ACTLR_EL3**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ACTLR_EL3**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    UNDEFINED;
elsif PSTATE_EL == EL2 then
    UNDEFINED;
elsif PSTATE_EL == EL3 then
    X[t, 64] = ACTLR_EL3;
```
**MSR ACTLR_EL3, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  ACTLR_EL3 = X[t, 64];
D17.2.5 AFSR0_EL1, Auxiliary Fault Status Register 0 (EL1)

The AFSR0_EL1 characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL1.

**Configurations**

AArch64 System register AFSR0_EL1 bits [31:0] are architecturally mapped to AArch32 System register ADFSR[31:0].

**Attributes**

AFSR0_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>32</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>31</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>0</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [63:0]**

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing AFSR0_EL1**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic AFSR0_EL1 or AFSR0_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, AFSR0_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>
elsif PSTATE.EL == EL3 then
    X[t, 64] = AFSR0_EL1;

**MSR AFSR0_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.AFSR0_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x128] = X[t, 64];
    else
        AFSR0_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AFSR0_EL2 = X[t, 64];
    else
        AFSR0_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    AFSR0_EL1 = X[t, 64];

**MRS <Xt>, AFSR0_EL12**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '101' then
        X[t, 64] = NVMem[0x128];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        X[t, 64] = AFSR0_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        X[t, 64] = AFSR0_EL1;
    else
        UNDEFINED;
**MSR AFSR0_EL12, <xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    NVMem[0x128] = X[t, 64];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AFSR0_EL1 = X[t, 64];
  else
    UNDEFINED;
elsif PSTATE_EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    AFSR0_EL1 = X[t, 64];
  else
    UNDEFINED;

D17.2.6 AFSR0_EL2, Auxiliary Fault Status Register 0 (EL2)

The AFSR0_EL2 characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL2.

**Configurations**

AArch64 System register AFSR0_EL2 bits [31:0] are architecturally mapped to AArch32 System register HADFSR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

AFSR0_EL2 is a 64-bit register.

**Field descriptions**

 IMPLEMENTATION DEFINED, bits [63:0]

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing AFSR0_EL2**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic AFSR0_EL2 or AFSR0_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, AFSR0_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CM</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.NV = '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = AFSR0_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = AFSR0_EL2;
### MSR AFSR0_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE_EL == EL2 then
    AFSR0_EL2 = X[t, 64];
elseif PSTATE_EL == EL3 then
    AFSR0_EL2 = X[t, 64];

### MRS <Xt>, AFSR0_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.AFSR0_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x128];
    else
        X[t, 64] = AFSR0_EL1;
elseif PSTATE_EL == EL2 then
    if HCR_EL2.EZM == '1' then
        X[t, 64] = AFSR0_EL2;
    else
        X[t, 64] = AFSR0_EL1;
elseif PSTATE_EL == EL3 then
    X[t, 64] = AFSR0_EL1;

### MSR AFSR0_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.AFSR0_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x128];
    else
        X[t, 64] = AFSR0_EL1;
NVMem[0x128] = X[t, 64];
else
    AFSR0_EL1 = X[t, 64];
elsif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AFSR0_EL2 = X[t, 64];
    else
        AFSR0_EL1 = X[t, 64];
    end if
elsif PSTATE_EL == EL3 then
    AFSR0_EL1 = X[t, 64];
D17.2.7 AFSR0_EL3, Auxiliary Fault Status Register 0 (EL3)

The AFSR0_EL3 characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL3.

**Configurations**

This register is present only when EL3 is implemented. Otherwise, direct accesses to AFSR0_EL3 are UNDEFINED.

**Attributes**

AFSR0_EL3 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing AFSR0_EL3**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, AFSR0_EL3**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    UNDEFINED;
elsif PSTATE_EL == EL2 then
    UNDEFINED;
elsif PSTATE_EL == EL3 then
    X[t, 64] = AFSR0_EL3;
```
### MSR AFSR0_EL3, $<Xt>$

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b110</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AFSR0_EL3 = X[t, 64];
AFSR1_EL1, Auxiliary Fault Status Register 1 (EL1)

The AFSR1_EL1 characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL1.

**Configurations**

AArch64 System register AFSR1_EL1 bits [31:0] are architecturally mapped to AArch32 System register AIFSR[31:0].

**Attributes**

AFSR1_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>IMPLEMENTATION DEFINED</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing AFSR1_EL1**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic AFSR1_EL1 or AFSR1_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, AFSR1_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```java
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.AFSR1_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x130];
    else
        X[t, 64] = AFSR1_EL1;
    elsif PSTATE.EL == EL2 then
        if HCR_EL2.E2H == '1' then
            X[t, 64] = AFSR1_EL1;
        elsif PSTATE.EL == EL0 then
            UNDEFINED;
        else
            X[t, 64] = AFSR1_EL1;
```
elsif PSTATE.EL == EL3
    X[t, 64] = AFSR1_EL1;

**MSR AFSR1_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFGWTR_EL2.AFSR1_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x130] = X[t, 64];
    else
        AFSR1_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AFSR1_EL2 = X[t, 64];
    else
        AFSR1_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    AFSR1_EL1 = X[t, 64];

**MRS <Xt>, AFSR1_EL12**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '101' then
        X[t, 64] = NVMem[0x130];
    elsif EL2Enabled() & HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        X[t, 64] = AFSR1_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.E2H == '1' then
        X[t, 64] = AFSR1_EL1;
    else
        UNDEFINED;
**MSR AFSR1_EL12, <Xt>**

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>
```

```markdown
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() \&\& HCR_EL2.<NV2,NV1,NV> == '101' then
    NVMem[0x130] = X[t, 64];
  elsif EL2Enabled() \&\& HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AFSR1_EL1 = X[t, 64];
  else
    UNDEFINED;
elsif PSTATE.EL == EL3 then
  if EL2Enabled() \&\& !ELUsingAArch32(EL2) \&\& HCR_EL2.E2H == '1' then
    AFSR1_EL1 = X[t, 64];
  else
    UNDEFINED;
```
D17.2.9 AFSR1_EL2, Auxiliary Fault Status Register 1 (EL2)

The AFSR1_EL2 characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL2.

**Configurations**

AArch64 System register AFSR1_EL2 bits [31:0] are architecturally mapped to AArch32 System register HAIFSR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

AFSR1_EL2 is a 64-bit register.

**Field descriptions**

![Field descriptions diagram]

**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing AFSR1_EL2**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic AFSR1_EL2 or AFSR1_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, AFSR1_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = AFSR1_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = AFSR1_EL2;
**MSR AFSR1_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() & HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  AFSR1_EL2 = X[t, 64];
eelsif PSTATE_EL == EL3 then
  AFSR1_EL2 = X[t, 64];
MRS <Xt>, AFSR1_EL1
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() & HCR_EL2.TVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.AFSR1_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x130];
  else
    X[t, 64] = AFSR1_EL1;
eelsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = AFSR1_EL2;
  else
    X[t, 64] = AFSR1_EL1;
eelsif PSTATE_EL == EL3 then
  X[t, 64] = AFSR1_EL1;

**MSR AFSR1_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() & HCR_EL2.TVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.AFSR1_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
    op0 op1 CRn CRm op2
    0b11 0b000 0b0101 0b0001 0b001

NVMem[0x130] = X[t, 64];
else
  AFSR1_EL1 = X[t, 64];
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AFSR1_EL2 = X[t, 64];
  else
    AFSR1_EL1 = X[t, 64];
  endif
elsif PSTATE_EL == EL3 then
  AFSR1_EL1 = X[t, 64];
D17.2.10  AFSR1_EL3, Auxiliary Fault Status Register 1 (EL3)

The AFSR1_EL3 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL3.

Configurations

This register is present only when EL3 is implemented. Otherwise, direct accesses to AFSR1_EL3 are UNDEFINED.

Attributes

AFSR1_EL3 is a 64-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AFSR1_EL3

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, AFSR1_EL3

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    UNDEFINED;
elsif PSTATE_EL == EL2 then
    UNDEFINED;
elsif PSTATE_EL == EL3 then
    X[t, 64] = AFSR1_EL3;
### MSR AFSR1_EL3, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  AFSR1_EL3 = X[t, 64];
```
D17.2.11 AIDR_EL1, Auxiliary ID Register

The AIDR_EL1 characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED identification information.

The value of this register must be interpreted in conjunction with the value of MIDR_EL1.

**Configurations**

AArch64 System register AIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register AIDR[31:0].

**Attributes**

AIDR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>bit positions</th>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
<tr>
<td>31-0</td>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

**Accessing AIDR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, AIDR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b11</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        endif
    else
        AArch64.SystemAccessTrap(EL1, 0x18);
    endif
else
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && !(HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.AIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = AIDR_EL1;
    endif
elsif PSTATE.EL == EL2 then
    X[t, 64] = AIDR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = AIDR_EL1;
```
D17.2.12 AMAIR_EL1, Auxiliary Memory Attribute Indirection Register (EL1)

The AMAIR_EL1 characteristics are:

Purpose
Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR_EL1.

Configurations
AArch64 System register AMAIR_EL1 bits [31:0] are architecturally mapped to AArch32 System register AMAIR0[31:0].
AArch64 System register AMAIR_EL1 bits [63:32] are architecturally mapped to AArch32 System register AMAIR1[31:0].

Attributes
AMAIR_EL1 is a 64-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

AMAIR_EL1 is permitted to be cached in a TLB.

IMPLEMENTATION DEFINED, bits [63:0]
IMPLEMENTATION DEFINED.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMAIR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic AMAIR_EL1 or AMAIR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, AMAIR_EL1

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>
```
X[t, 64] = AMAIR_EL1;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = AMAIR_EL2;
  else
    X[t, 64] = AMAIR_EL1;
elsif PSTATE_EL == EL3 then
  X[t, 64] = AMAIR_EL1;

**MSR AMAIR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() & HCR_EL2.TVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFGWTR_EL2.AMAIR_EL1 == '1') then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
    NMem[0x148] = X[t, 64];
  else
    AMAIR_EL1 = X[t, 64];
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    AMAIR_EL2 = X[t, 64];
  else
    UNDEFINED;
elsif PSTATE_EL == EL3 then
  AMAIR_EL1 = X[t, 64];

**MRS <Xt>, AMAIR_EL12**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '101' then
    X[t, 64] = NMem[0x148];
  elsif EL2Enabled() & HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = AMAIR_EL1;
  else
    UNDEFINED;
elsif PSTATE_EL == EL3 then
  if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.E2H == '1' then
    X[t, 64] = AMAIR_EL1;
  else
    UNDEFINED;
### MSR AMAIR_EL12, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV2,NV1,NV == '101' then
        NVMem[0x148] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AMAIR_EL1 = X[t, 64];
    else
        UNDEFINED;
elsif PSTATE_EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        AMAIR_EL1 = X[t, 64];
    else
        UNDEFINED;
```

---

**AArch64 System Register Descriptions**  
**D17.2 General system control registers**
D17.2.13  AMAIR_EL2, Auxiliary Memory Attribute Indirection Register (EL2)

The AMAIR_EL2 characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR_EL2.

**Configurations**

AArch64 System register AMAIR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HAMAIR0[31:0].

AArch64 System register AMAIR_EL2 bits [63:32] are architecturally mapped to AArch32 System register HAMAIR1[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

AMAIR_EL2 is a 64-bit register.

**Field descriptions**

![Field diagram](image)

AMAIR_EL2 is permitted to be cached in a TLB.

**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing AMAIR_EL2**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic AMAIR_EL2 or AMAIR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, AMAIR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
```c
elsif PSTATE_EL == EL2 then
    X[t, 64] = AMAIR_EL2;
elsif PSTATE_EL == EL3 then
    X[t, 64] = AMAIR_EL2;

MSR AMAIR_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE_EL == EL2 then
    AMAIR_EL2 = X[t, 64];
elsif PSTATE_EL == EL3 then
    AMAIR_EL2 = X[t, 64];

MRS <Xt>, AMAIR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' then
        X[t, 64] = AMAIR_EL2;
    else
        X[t, 64] = AMAIR_EL1;
elsif PSTATE_EL == EL3 then
    X[t, 64] = AMAIR_EL1;
```

### MSR AMAIR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.AMAIR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x148] = X[t, 64];
    else
        AMAIR_EL1 = X[t, 64];
    elsif PSTATE.EL == EL2 then
        if HCR_EL2.E2H == '1' then
            AMAIR_EL2 = X[t, 64];
        else
            AMAIR_EL2 = X[t, 64];
        endif
    else
        AMAIR_EL1 = X[t, 64];
    elsif PSTATE.EL == EL3 then
        AMAIR_EL1 = X[t, 64];
    endif
D17.2.14 AMAIR_EL3, Auxiliary Memory Attribute Indirection Register (EL3)

The AMAIR_EL3 characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR_EL3.

**Configurations**

This register is present only when EL3 is implemented. Otherwise, direct accesses to AMAIR_EL3 are UNDEFINED.

**Attributes**

AMAIR_EL3 is a 64-bit register.

**Field descriptions**

AMAIR_EL3 is permitted to be cached in a TLB.

**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing AMAIR_EL3**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, AMAIR_EL3**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b11</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  X[t, 64] = AMAIR_EL3;
**MSR AMAIR_EL3, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    AMAIR_EL3 = X[t, 64];
D17.2.15 APDAKeyHi_EL1, Pointer Authentication Key A for Data (bits[127:64])

The APDAKeyHi_EL1 characteristics are:

**Purpose**

Holds bits[127:64] of key A used for authentication of data pointer values.

--- Note ---

The term APDAKey_EL1 is used to describe the concatenation of APDAKeyHi_EL1: APDAKeyLo_EL1.

**Configurations**

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APDAKeyHi_EL1 are UNDEFINED.

**Attributes**

APDAKeyHi_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 bit value, bits[127:64] of the 128 bit pointer authentication key value</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:0]**

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing APDAKeyHi_EL1**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRS \langle Xt \rangle, \text{APDAKeyHi_EL1} \\
\begin{array}{cccccc}
  \text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
  0b11 & 0b000 & 0b0010 & 0b0010 & 0b001
\end{array}
\]

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& SCR_EL3.APK == '0' then
        UNDEFINED;
    elsif EL2Enabled() \&\& HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() \&\& (HaveEL(EL3) || SCR_EL3.FGTEn == '1') \&\& HFRGR_EL2.APDAKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = APDAKeyHi_EL1;
else if PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = APDAKeyHi_EL1;
else if PSTATE.EL == EL3 then
  X[t, 64] = APDAKeyHi_EL1;

**MSR APDAKeyHi_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.APK == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.APDAKey == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      APDAKeyHi_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    APDAKeyHi_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
  APDAKeyHi_EL1 = X[t, 64];
D17.2.16 APDAKeyLo_EL1, Pointer Authentication Key A for Data (bits[63:0])

The APDAKeyLo_EL1 characteristics are:

**Purpose**
Holds bits[63:0] of key A used for authentication of data pointer values.

**Note**
The term APDAKey_EL1 is used to describe the concatenation of APDAKeyHi_EL1: APDAKeyLo_EL1.

**Configurations**
This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APDAKeyLo_EL1 are UNDEFINED.

**Attributes**
APDAKeyLo_EL1 is a 64-bit register.

**Field descriptions**

- **Bits [63:0]**
64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing APDAKeyLo_EL1**
Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, APDAKeyLo_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.APK == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCGRTR_EL2.APDKey == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
```plaintext
AArch64 System Register Descriptions
D17.2 General system control registers

elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
  elsif Halted() && EDSCR.SDD == '1' then
    undefine;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = APDAKeyLo_EL1;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
else
  X[t, 64] = APDAKeyLo_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = APDAKeyLo_EL1;

**MSR APDAKeyLo_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11 0b000 0b0010 0b0010 0b000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif PSTATE.EL == EL3 then
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  APDAKeyLo_EL1 = X[t, 64];
```

D17-5592  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  ARM DDI 0487I.a  Non-Confidential  ID081822
D17.2.17 APDBKeyHi_EL1, Pointer Authentication Key B for Data (bits[127:64])

The APDBKeyHi_EL1 characteristics are:

**Purpose**

Holds bits[127:64] of key B used for authentication of data pointer values.

**Note**

The term APDBKey_EL1 is used to describe the concatenation of APDBKeyHi_EL1: APDBKeyLo_EL1.

**Configurations**

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APDBKeyHi_EL1 are UNDEFINED.

**Attributes**

APDBKeyHi_EL1 is a 64-bit register.

**Field descriptions**

Bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing APDBKeyHi_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, APDBKeyHi_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCGRTR_EL2.APDBKey == '1') then
        AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        X[t, 64] = APDBKeyHi_EL1;
    endif
elsif PSTATE_EL == EL3 then
    X[t, 64] = APDBKeyHi_EL1;
endif

MSR APDBKeyHi_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
else if PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        APDBKeyHi_EL1 = X[t, 64];
    end
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        APDBKeyHi_EL1 = X[t, 64];
    endif
elsif PSTATE_EL == EL3 then
    APDBKeyHi_EL1 = X[t, 64];
endif
D17.2.18  APDBKeyLo_EL1, Pointer Authentication Key B for Data (bits[63:0])

The APDBKeyLo_EL1 characteristics are:

**Purpose**

Holds bits[63:0] of key B used for authentication of data pointer values.

**Note**

The term APDBKey_EL1 is used to describe the concatenation of APDBKeyHi_EL1:
APDBKeyLo_EL1.

**Configurations**

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APDBKeyLo_EL1 are UNDEFINED.

**Attributes**

APDBKeyLo_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>64 bit value, bits[63:0] of the 128 bit pointer authentication key value</th>
</tr>
</thead>
</table>

**Accessing APDBKeyLo_EL1**

Accesses to this register use the following encodings in the System register encoding space:

*MRS <Xt>, APDBKeyLo_EL1*

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elself PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD = '1'" && SCR_EL3.APK == '0' then
UNDEFINED;
elself EL2Enabled() && HCR_EL2.APK == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elself EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCRTR_EL2.APDBKey == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end if
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end if
elsif PSTATE_EL == EL3 then
  X[t, 64] = APDBKeyLo_EL1;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  end if
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end if
elsif PSTATE_EL == EL3 then
  APDBKeyLo_EL1 = X[t, 64];
endif

MSR APDBKeyLo_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>
### D17.2.19 APGAKeyHi_EL1, Pointer Authentication Key A for Code (bits[127:64])

The APGAKeyHi_EL1 characteristics are:

**Purpose**

Holds bits[127:64] of key used for generic pointer authentication code.

**Note**

The term APGAKey_EL1 is used to describe the concatenation of APGAKeyHi_EL1:
APGAKeyLo_EL1.

**Configurations**

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APGAKeyHi_EL1 are UNDEFINED.

**Attributes**

APGAKeyHi_EL1 is a 64-bit register.

**Field descriptions**

64 bit value, bits[127:64] of the 128 bit pointer authentication key value

**Bits [63:0]**

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing APGAKeyHi_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, APGAKeyHi_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then

UNDEFINED;

elsif PSTATE.EL == EL1 then

if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'

& SCR_EL3.APK == '0' then

UNDEFINED;

elsif EL2Enabled() && HCR_EL2.APK == '0' then

AArch64.SystemAccessTrap(EL2, 0x18);

elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCGR_EL2.APGAKey == '1' then

AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = APGAKeyHi_EL1;
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = APGAKeyHi_EL1;
elsif PSTATE_EL == EL3 then
    X[t, 64] = APGAKeyHi_EL1;

MSR APGAKeyHi_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        APGAKeyHi_EL1 = X[t, 64];
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APGAKeyHi_EL1 = X[t, 64];
elsif PSTATE_EL == EL3 then
    APGAKeyHi_EL1 = X[t, 64];
D17.2.20 APGAKeyLo_EL1, Pointer Authentication Key A for Code (bits[63:0])

The APGAKeyLo_EL1 characteristics are:

Purpose

Holds bits[63:0] of key used for generic pointer authentication code.

Note

The term APGAKey_EL1 is used to describe the concatenation of APGAKeyHi_EL1: APGAKeyLo_EL1.

Configurations

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APGAKeyLo_EL1 are UNDEFINED.

Attributes

APGAKeyLo_EL1 is a 64-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>64 bit value, bits[63:0] of the 128 bit pointer authentication key value</th>
</tr>
</thead>
</table>

Accessing APGAKeyLo_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[ MRS <Xt>, APGAKeyLo_EL1 \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD = '1' & SCR_EL3.APK == '0' then
    UNDEFINED;
  elsif EL2Enabled() & HCR_EL2.APK == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFCRTR_EL2.APGKey == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = APGAKeyLo_EL1;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
        UNDEFINED;
    elseif HaveEL(EL3) && SCR_EL3.APK == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = APGAKeyLo_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = APGAKeyLo_EL1;

**MSR APGAKeyLo_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
        UNDEFINED;
    elseif EL2Enabled() && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.APGAKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) && SCR_EL3.APK == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APGAKeyLo_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
        UNDEFINED;
    elseif HaveEL(EL3) && SCR_EL3.APK == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APGAKeyLo_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    APGAKeyLo_EL1 = X[t, 64];
D17.2.21   APIAKeyHi_EL1, Pointer Authentication Key A for Instruction (bits[127:64])

The APIAKeyHi_EL1 characteristics are:

**Purpose**

Holds bits[127:64] of key A used for authentication of instruction pointer values.

--- Note ---

The term APIAKey_EL1 is used to describe the concatenation of APIAKeyHi_EL1:
APIAKeyLo_EL1.

**Configurations**

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to
APIAKeyHi_EL1 are UNDEFINED.

**Attributes**

APIAKeyHi_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>64 bit value, bits[127:64] of the 128 bit pointer authentication key value</td>
</tr>
<tr>
<td>32</td>
<td>64 bit value, bits[127:64] of the 128 bit pointer authentication key value</td>
</tr>
</tbody>
</table>

**Bits [63:0]**

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing APIAKeyHi_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, APIAKeyHi_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elif PSTATE_EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & SCR_EL3.APK == '0' then
    UNDEFINED;
elif EL2Enabled() & HCR_EL2.APK == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFCRTR_EL2.APIAKey == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = APIAKeyHi_EL1;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
    UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = APIAKeyHi_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = APIAKeyHi_EL1;

MSR APIAKeyHi_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b010</td>
<td>0b001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
    UNDEFINED;
elsif EL2Enabled() && HCR_EL2.APK == '0' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.APIAKey == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  APIAKeyHi_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
    UNDEFINED;
elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  APIAKeyHi_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
  APIAKeyHi_EL1 = X[t, 64];
D17.2.22  APIAKeyLo_EL1, Pointer Authentication Key A for Instruction (bits[63:0])

The APIAKeyLo_EL1 characteristics are:

**Purpose**

Holds bits[63:0] of key A used for authentication of instruction pointer values.

**Note**

The term APIAKey_EL1 is used to describe the concatenation of APIAKeyHi_EL1: APIAKeyLo_EL1.

**Configurations**

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APIAKeyLo_EL1 are UNDEFINED.

**Attributes**

APIAKeyLo_EL1 is a 64-bit register.

**Field descriptions**

|| 64 bit value, bits[63:0] of the 128 bit pointer authentication key value
---|---
| 01 |
| 0 |

**Bits [63:0]**

64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing APIAKeyLo_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, APIAKeyLo_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCGRTR_EL2.APIAKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```c
elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = APIAKeyLo_EL1;
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = APIAKeyLo_EL1;
elsif PSTATE_EL == EL3 then
    X[t, 64] = APIAKeyLo_EL1;

MSR APIAKeyLo_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>
```

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
        UNDEFINED;
    elif EL2Enabled() && HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.APIAKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            APIAKeyLo_EL1 = X[t, 64];
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
        UNDEFINED;
    elif HaveEL(EL3) && SCR_EL3.APK == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            APIAKeyLo_EL1 = X[t, 64];
elsif PSTATE_EL == EL3 then
    APIAKeyLo_EL1 = X[t, 64];
```
### D17.2.23 APIBKeyHi_EL1, Pointer Authentication Key B for Instruction (bits[127:64])

The APIBKeyHi_EL1 characteristics are:

#### Purpose

Holds bits[127:64] of key B used for authentication of instruction pointer values.

---

#### Note

The term APIBKey_EL1 is used to describe the concatenation of APIBKeyHi_EL1: APIBKeyLo_EL1.

---

#### Configurations

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APIBKeyHi_EL1 are UNDEFINED.

#### Attributes

APIBKeyHi_EL1 is a 64-bit register.

#### Field descriptions

![Field descriptions diagram]

#### Bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

#### Accessing APIBKeyHi_EL1

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, APIBKeyHi_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0001</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() & haveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & SCR_EL3.APK == '0' then
        UNDEFINED;
    elsif EL2Enabled() & HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & (!haveEL(EL3) || SCR_EL3.FGTEn == '1') & HFCRTR_EL2.APIBKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = APIBKeyHi_EL1;
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = APIBKeyHi_EL1;
elsif PSTATE_EL == EL3 then
  X[t, 64] = APIBKeyHi_EL1;

**MSR APIBKeyHi_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0001</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.APK == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.APIBKey == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    APIBKeyHi_EL1 = X[t, 64];
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.APK == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    APIBKeyHi_EL1 = X[t, 64];
elsif PSTATE_EL == EL3 then
  APIBKeyHi_EL1 = X[t, 64];
D17.2.24 APIBKeyLo_EL1, Pointer Authentication Key B for Instruction (bits[63:0])

The APIBKeyLo_EL1 characteristics are:

**Purpose**

Holds bits[63:0] of key B used for authentication of instruction pointer values.

**Note**

The term APIBKey_EL1 is used to describe the concatenation of APIBKeyHi_EL1: APIBKeyLo_EL1.

**Configurations**

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APIBKeyLo_EL1 are UNDEFINED.

**Attributes**

APIBKeyLo_EL1 is a 64-bit register.

**Field descriptions**

| Bits [63:0] | 64 bit value, bits[63:0] of the 128 bit pointer authentication key value |

**Accessing APIBKeyLo_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, APIBKeyLo_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.APK == '0' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.APK == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCGRTR_EL2.APIBKey == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) \&\& SCR_EL3.APK == '0' then
    if Halted() \&\& EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = APIBKeyLo_EL1;
elsif PSTATE_EL == EL2 then
    if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' \&\& SCR_EL3.APK == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) \&\& SCR_EL3.APK == '0' then
        if Halted() \&\& EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = APIBKeyLo_EL1;
elsif PSTATE_EL == EL3 then
    X[t, 64] = APIBKeyLo_EL1;

\textbf{MSR APIBKeyLo_EL1, <Xt>}

\begin{center}
\begin{tabular}{cccccc}
\hline
\textbf{op0} & \textbf{op1} & \textbf{CRn} & \textbf{CRm} & \textbf{op2} \\
\hline
0b11 & 0b000 & 0b0010 & 0b0001 & 0b010 \\
\hline
\end{tabular}
\end{center}

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' \&\& SCR_EL3.APK == '0' then
        UNDEFINED;
    elsif EL2Enabled() \&\& HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() \&\& (!HaveEL(EL3) \&\& SCR_EL3.FGTEn == '1') \&\& HFGWTR_EL2.APIBKey == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) \&\& SCR_EL3.APK == '0' then
        if Halted() \&\& EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            APIBKeyLo_EL1 = X[t, 64];
elsif PSTATE_EL == EL2 then
    if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' \&\& SCR_EL3.APK == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) \&\& SCR_EL3.APK == '0' then
        if Halted() \&\& EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        APIBKeyLo_EL1 = X[t, 64];
elsif PSTATE_EL == EL3 then
    APIBKeyLo_EL1 = X[t, 64];
D17.2.25 CCSIDR2_EL1, Current Cache Size ID Register 2

The CCSIDR2_EL1 characteristics are:

**Purpose**

Provides the information about the architecture of the currently selected cache from bits[63:32] of CCSIDR_EL1.

**Configurations**

AArch64 System register CCSIDR2_EL1 bits [31:0] are architecturally mapped to AArch32 System register CCSIDR2[31:0].

This register is present only when FEAT_CCIDX is implemented. Otherwise, direct accesses to CCSIDR2_EL1 are UNDEFINED.

In an implementation which does not support AArch32 at EL1, it is IMPLEMENTATION DEFINED whether reading this register gives an UNKNOWN value or is UNDEFINED.

The implementation includes one CCSIDR2_EL1 for each cache that it can access. CSSELR_EL1 selects which Cache Size ID Register is accessible.

**Attributes**

CCSIDR2_EL1 is a 64-bit register.

**Field descriptions**

**Bits [63:24]**

Reserved, RES0.

**NumSets, bits [23:0]**

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

**Accessing CCSIDR2_EL1**

If CSSELR_EL1.\{TnD, Level, InD\} is programmed to a cache level that is not implemented, then on a read of the CCSIDR2_EL1 the behavior is CONSTRAINED UNPREDICTABLE, and can be one of the following:

- The CCSIDR2_EL1 read is treated as NOP.
- The CCSIDR2_EL1 read is UNDEFINED.
- The CCSIDR2_EL1 read returns an UNKNOWN value.

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, CCSIDR2_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    else
      UNDEFINED;
  elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID2 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TID4 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      X[t, 64] = CCSIDR2_EL1;
  elsif PSTATE.EL == EL2 then
    X[t, 64] = CCSIDR2_EL1;
  elsif PSTATE.EL == EL3 then
    X[t, 64] = CCSIDR2_EL1;
D17.2.26 CCSIDR_EL1, Current Cache Size ID Register

The CCSIDR_EL1 characteristics are:

**Purpose**

Provides information about the architecture of the currently selected cache.

**Configurations**

AArch64 System register CCSIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register CCSIDR[31:0].

AArch64 System register CCSIDR_EL1 bits [63:32] are architecturally mapped to AArch32 System register CCSIDR2[31:0].

The implementation includes one CCSIDR_EL1 for each cache that it can access. CSSELK_EL1 selects which Cache Size ID Register is accessible.

**Attributes**

CCSIDR_EL1 is a 64-bit register.

**Field descriptions**

*When FEAT_CCIDX is implemented:*

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:56]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[55:32]</td>
<td>NumSets, (Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.</td>
</tr>
<tr>
<td>[31:24]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[23:3]</td>
<td>Associativity, (Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.</td>
</tr>
</tbody>
</table>
| [2:0]     | LineSize, (Log2(Number of bytes in cache line)) - 4. For example:
  |   • For a line length of 16 bytes: Log2(16) = 4, LineSize entry = 0. This is the minimum line length.
  |   • For a line length of 32 bytes: Log2(32) = 5, LineSize entry = 1. |

**Note**

The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.
--- Note ---
The C++ 17 specification has two defined parameters relating to the granularity of memory that does not interfere. For generic software and tools, Arm will set the hardware_destructive_interference_size parameter to 256 bytes and the hardware_constructive_interference_size parameter to 64 bytes.

When FEAT_MTE is implemented and enabled, where a cache only holds Allocation tags, this field is RES0.

**Otherwise:**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>31:28</td>
<td>Reserved, UNKNOWN.</td>
</tr>
<tr>
<td>27:13</td>
<td>NumSets, bits</td>
</tr>
<tr>
<td>12:3</td>
<td>Associativity, bits</td>
</tr>
<tr>
<td>2:0</td>
<td>LineSize, bits</td>
</tr>
</tbody>
</table>

--- Note ---
The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

---

Bits [63:32]
Reserved, RES0.

Bits [31:28]
Reserved, UNKNOWN.

NumSets, bits [27:13]
(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Associativity, bits [12:3]
(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

LineSize, bits [2:0]
(Log2(Number of bytes in cache line)) - 4. For example:
- For a line length of 16 bytes: Log2(16) = 4, LineSize entry = 0. This is the minimum line length.
- For a line length of 32 bytes: Log2(32) = 5, LineSize entry = 1.

--- Note ---
The C++ 17 specification has two defined parameters relating to the granularity of memory that does not interfere. For generic software and tools, Arm will set the hardware_destructive_interference_size parameter to 256 bytes and the hardware_constructive_interference_size parameter to 64 bytes.
## Accessing CCSIDR_EL1

If `CSSELR_EL1.{TnD, Level, InD}` is programmed to a cache level that is not implemented, then on a read of the CCSIDR_EL1 the behavior is **CONSTRAINED UNPREDICTABLE**, and can be one of the following:

- The CCSIDR_EL1 read is treated as NOP.
- The CCSIDR_EL1 read is **UNDEFINED**.
- The CCSIDR_EL1 read returns an **UNKNOWN** value.

Accesses to this register use the following encodings in the System register encoding space:

### MRS <Xt>, CCSIDR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```
if PSTATE_EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        end
    else
        UNDEFINED;
    end
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID2 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TID4 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.CCSIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = CCSIDR_EL1;
    end
elsif PSTATE_EL == EL2 then
    X[t, 64] = CCSIDR_EL1;
elsif PSTATE_EL == EL3 then
    X[t, 64] = CCSIDR_EL1;
```
D17.2.27  CLIDR_EL1, Cache Level ID Register

The CLIDR_EL1 characteristics are:

**Purpose**

Identifies the type of cache, or caches, that are implemented at each level and can be managed using the architected cache maintenance instructions that operate by set/way, up to a maximum of seven levels. Also identifies the Level of Coherence (LoC) and Level of Unification (LoU) for the cache hierarchy.

**Configurations**

AArch64 System register CLIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register CLIDR[31:0].

**Attributes**

CLIDR_EL1 is a 64-bit register.

**Field descriptions**

![Field diagram]

**Bits [63:47]**

Reserved, RES0.

**Ttype<n>, bits [2(n-1)+34:2(n-1)+33], for n = 7 to 1**

*When FEAT_MTE2 is implemented:*

Tag cache type. Indicate the type of cache that is implemented and can be managed using the architected cache maintenance instructions that operate by set/way at each level, from Level 1 up to a maximum of seven levels of cache hierarchy.

- **0b00**: No Tag Cache.
- **0b01**: Separate Allocation Tag Cache.
- **0b10**: Unified Allocation Tag and Data cache, Allocation Tags and Data in unified lines.
- **0b11**: Unified Allocation Tag and Data cache, Allocation Tags and Data in separate lines.

**ICB, bits [32:30]**

Inner cache boundary. This field indicates the boundary for caching Inner Cacheable memory regions.

- **0b000**: Not disclosed by this mechanism.
- **0b001**: L1 cache is the highest Inner Cacheable level.
- **0b010**: L2 cache is the highest Inner Cacheable level.
- **0b011**: L3 cache is the highest Inner Cacheable level.
- **0b100**: L4 cache is the highest Inner Cacheable level.
- **0b101**: L5 cache is the highest Inner Cacheable level.
- **0b110**: L6 cache is the highest Inner Cacheable level.
- **0b111**: L7 cache is the highest Inner Cacheable level.
LoUU, bits [29:27]
Level of Unification Uniprocessor for the cache hierarchy.
For a description of the values of this field, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions on page D7-5051.

Note
When FEAT_S2FWB is implemented, the architecture requires that this field is zero so that no levels of data cache need to be cleaned in order to manage coherency with instruction fetches.

LoC, bits [26:24]
Level of Coherence for the cache hierarchy.
For a description of the values of this field, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions on page D7-5051.

LoUIS, bits [23:21]
Level of Unification Inner Shareable for the cache hierarchy.
For a description of the values of this field, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions on page D7-5051.

Note
When FEAT_S2FWB is implemented, the architecture requires that this field is zero so that no levels of data cache need to be cleaned in order to manage coherency with instruction fetches.

Ctype<n>, bits [3(n-1)+2:3(n-1)], for n = 7 to 1
Cache Type fields. Indicate the type of cache that is implemented and can be managed using the architected cache maintenance instructions that operate by set/way at each level, from Level 1 up to a maximum of seven levels of cache hierarchy. Possible values of each field are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000</td>
<td>No cache.</td>
</tr>
<tr>
<td>0b001</td>
<td>Instruction cache only.</td>
</tr>
<tr>
<td>0b010</td>
<td>Data cache only.</td>
</tr>
<tr>
<td>0b011</td>
<td>Separate instruction and data caches.</td>
</tr>
<tr>
<td>0b100</td>
<td>Unified cache.</td>
</tr>
</tbody>
</table>

All other values are reserved.

If software reads the Cache Type fields from Ctype1 upwards, once it has seen a value of 000, no caches that can be managed using the architected cache maintenance instructions that operate by set/way exist at further-out levels of the hierarchy. So, for example, if Ctype3 is the first Cache Type field with a value of 000, the values of Ctype4 to Ctype7 must be ignored.

Accessing CLIDR_EL1
Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS} \ <\text{Xt}, \ CLIDR_\text{EL1} \\
\begin{array}{cccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b001 & 0b0000 & 0b0000 & 0b001 \\
\end{array}
\]

if PSTATE.EL == EL0 then
if IsFeatureImplemented(FEAT_IDST) then
if EL2Enabled() && HCR_EL2.TGE == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
    AArch64.SystemAccessTrap(EL1, 0x18);
else
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID2 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.TID4 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.CLIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = CLIDR_EL1;
    elsif PSTATE.EL == EL2 then
        X[t, 64] = CLIDR_EL1;
    elsif PSTATE.EL == EL3 then
        X[t, 64] = CLIDR_EL1;
D17.2.28 CONTEXTIDR_EL1, Context ID Register (EL1)

The CONTEXTIDR_EL1 characteristics are:

**Purpose**

Identifies the current Process Identifier.

The value of the whole of this register is called the Context ID and is used by:

- The debug logic, for Linked and Unlinked Context ID matching.
- The trace logic, to identify the current process.

The significance of this register is for debug and trace use only.

**Configurations**

AArch64 System register CONTEXTIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register CONTEXTIDR[31:0].

**Attributes**

CONTEXTIDR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
<th>RES0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>PROCID</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**PROCID, bits [31:0]**

Process Identifier. This field must be programmed with a unique value that identifies the current process.

--- **Note** ---

In AArch32 state, when TTBCR.EAE is set to 0, CONTEXTIDR.ASID holds the ASID.

In AArch64 state, CONTEXTIDR_EL1 is independent of the ASID, and for the EL1&0 translation regime either TTBR0_EL1 or TTBR1_EL1 holds the ASID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CONTEXTIDR_EL1**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CONTEXTIDR_EL1 or CONTEXTIDR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:
MRS \langle Xt \rangle, CONTEXTIDR_EL1

\[
\begin{array}{|c|c|c|c|c|}
\hline
0b11 & 0b000 & 0b1101 & 0b0000 & 0b001 \\
\hline
\end{array}
\]

\textbf{MSR CONTEXTIDR_EL1, \langle Xt \rangle}

\[
\begin{array}{|c|c|c|c|c|}
\hline
0b11 & 0b000 & 0b1101 & 0b0000 & 0b001 \\
\hline
\end{array}
\]

if PSTATE.EL == EL0 then
    UNDEFINED;
elssif PSTATE.EL == EL1 then
    if EL2Enabled() \&\& HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
elssif EL2Enabled() \&\& (HaveEL(EL3) \| SCR_EL3.FGTEn == '1') \&\& HFGWTR_EL2.CONTEXTIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
elssif EL2Enabled() \&\& HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x108] = X[t, 64];
    eelse
        CONTEXTIDR_EL1 = X[t, 64];
elssif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CONTEXTIDR_EL2 = X[t, 64];
eelse
        CONTEXTIDR_EL1 = X[t, 64];
elssif PSTATE.EL == EL3 then
    CONTEXTIDR_EL1 = X[t, 64];
MRS <Xt>, CONTEXTIDR_EL12

if PSTATE.EL == EL0 then
  UNDEFINED;
elsf PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    X[t, 64] = NVMem[0x108];
elsf EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elsf PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = CONTEXTIDR_EL1;
else
  UNDEFINED;
elsf PSTATE.EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    X[t, 64] = CONTEXTIDR_EL1;
else
  UNDEFINED;

MSR CONTEXTIDR_EL12, <Xt>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsf PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    NVMem[0x108] = X[t, 64];
elsf EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elsf PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    CONTEXTIDR_EL1 = X[t, 64];
else
  UNDEFINED;
elsf PSTATE.EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    CONTEXTIDR_EL1 = X[t, 64];
else
  UNDEFINED;
D17.2.29 CONTEXTIDR_EL2, Context ID Register (EL2)

The CONTEXTIDR_EL2 characteristics are:

**Purpose**

Identifies the current Process Identifier for EL2.

The value of the whole of this register is called the Context ID and is used by:

- The debug logic, for Linked and Unlinked Context ID matching.
- The trace logic, to identify the current process.

The significance of this register is for debug and trace use only.

**Configurations**

This register is present only when FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented. Otherwise, direct accesses to CONTEXTIDR_EL2 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

CONTEXTIDR_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:32]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[31:0]</td>
<td>PROCID</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**PROCID, bits [31:0]**

Process Identifier. This field must be programmed with a unique value that identifies the current process.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CONTEXTIDR_EL2**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CONTEXTIDR_EL2 or CONTEXTIDR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:
**MRS <Xt>, CONTEXTIDR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = CONTEXTIDR_EL2;
elsif PSTATE.EL == EL3 then
  X[t, 64] = CONTEXTIDR_EL2;

**MSR CONTEXTIDR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  CONTEXTIDR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
  CONTEXTIDR_EL2 = X[t, 64];

**MRS <Xt>, CONTEXTIDR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCRTR_EL2.CONTEXTIDR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x108];
  else
    X[t, 64] = CONTEXTIDR_EL1;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = CONTEXTIDR_EL2;
else
  X[t, 64] = CONTEXTIDR_EL2;
X[t, 64] = CONTEXTIDR_EL1;
elsif PSTATE_EL == EL3 then
 X[t, 64] = CONTEXTIDR_EL1;

**MSR CONTEXTIDR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b00</td>
<td>0b1</td>
<td>0b000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
 UNDEFINED;
elsif PSTATE_EL == EL1 then
 if EL2Enabled() & HCR_EL2.TVM == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
 elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFGWTR_EL2.CONTEXTIDR_EL1 == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
 elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
  NVMem[0x108] = X[t, 64];
 else
  CONTEXTIDR_EL1 = X[t, 64];
 elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
   CONTEXTIDR_EL2 = X[t, 64];
  else
   CONTEXTIDR_EL1 = X[t, 64];
 elsif PSTATE_EL == EL3 then
   CONTEXTIDR_EL1 = X[t, 64];
 elsif PSTATE_EL == EL3 then
   CONTEXTIDR_EL1 = X[t, 64];
D17.2.30 **CPACR_EL1, Architectural Feature Access Control Register**

The CPACR_EL1 characteristics are:

**Purpose**

Controls access to trace, SME, Streaming SVE, SVE, and Advanced SIMD and floating-point functionality.

**Configurations**

AArch64 System register CPACR_EL1 bits [31:0] are architecturally mapped to AArch32 System register CPACR[31:0].

When EL2 is implemented and enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}, the fields in this register have no effect on execution at EL0 and EL1. In this case, the controls provided by CPTR_EL2 are used.

**Attributes**

CPACR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>RES0</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>31</td>
<td>RES0</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>28</td>
<td>TTA</td>
<td>Traps EL0 and EL1 System register accesses to all implemented trace registers from both Execution states to EL1, or to EL2 when it is implemented and enabled in the current Security state and HCR_EL2.TGE is 1, as follows:</td>
</tr>
<tr>
<td>27</td>
<td>SMEN</td>
<td>In AArch64 state, accesses to trace registers are trapped, reported using ESR_ELx.EC value 0x18.</td>
</tr>
<tr>
<td>26</td>
<td>SMEN</td>
<td>In AArch32 state, MRC and MCR accesses to trace registers are trapped, reported using ESR_ELx.EC value 0x05.</td>
</tr>
<tr>
<td>25</td>
<td>SMEN</td>
<td>In AArch32 state, MRRC and MCRR accesses to trace registers are trapped, reported using ESR_ELx.EC value 0x0C.</td>
</tr>
<tr>
<td>24</td>
<td>FPEN</td>
<td>0b0 This control does not cause any instructions to be trapped.</td>
</tr>
<tr>
<td>23</td>
<td>FPEN</td>
<td>0b1 This control causes EL0 and EL1 System register accesses to all implemented trace registers to be trapped.</td>
</tr>
<tr>
<td>22</td>
<td>ZEN</td>
<td>Note</td>
</tr>
<tr>
<td>21</td>
<td>ZEN</td>
<td>The ETMv4 architecture and ETE do not permit EL0 to access the trace registers. If the trace unit implements FEAT_ETMv4 or FEAT_ETE, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPACR_EL1.TTA is 1.</td>
</tr>
<tr>
<td>20</td>
<td>ZEN</td>
<td>The Arm architecture does not provide traps on trace register accesses through the optional memory-mapped interface.</td>
</tr>
<tr>
<td>19</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ZEN</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>ZEN</td>
<td></td>
</tr>
</tbody>
</table>
System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not implemented, this bit is RES0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [27:26]**

Reserved, RES0.

**SMEN, bits [25:24]**

*When FEAT_SME is implemented:*

Traps execution at EL1 and EL0 of SME instructions, SVE instructions when FEAT_SVE is not implemented or the PE is in Streaming SVE mode, and instructions that directly access the SVCR or SMCR_EL1 System registers to EL1, or to EL2 when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE is 1.

When instructions that directly access the SVCR System register are trapped with reference to this control, the MSR, SVCSR, MSR, SVCRZA, and MSR, SVCRSZ, instructions are also trapped.

The exception is reported using ESR_ELx.EC value of 0x1D, with an ISS code of 0x0000000.

This field does not affect whether Streaming SVE or SME register values are valid.

A trap taken as a result of CPACR_EL1.SMEN has precedence over a trap taken as a result of CPACR_EL1.FPEN.

- **0b00** This control causes execution of these instructions at EL1 and EL0 to be trapped.
- **0b01** This control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL1 to be trapped.
- **0b10** This control causes execution of these instructions at EL1 and EL0 to be trapped.
- **0b11** This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

**Bits [23:22]**

Reserved, RES0.

**FPEN, bits [21:20]**

Traps execution at EL1 and EL0 of instructions that access the Advanced SIMD and floating-point registers from both Execution states to EL1, reported using ESR_ELx.EC value 0x07, or to EL2 reported using ESR_ELx.EC value 0x00 when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to FPCR, FPSR, any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers.
- In AArch32 state, FPSCR, and any of the SIMD and floating-point registers Q0-15, including their views as D0-D31 registers or S0-31 registers.

Traps execution at EL1 and EL0 of SME and SVE instructions to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. The exception is reported using ESR_ELx.EC value 0x07.

A trap taken as a result of CPACR_EL1.SMEN has precedence over a trap taken as a result of CPACR_EL1.FPEN.
A trap taken as a result of CPACR_EL1.ZEN has precedence over a trap taken as a result of CPACR_EL1.FPEN.

0b00  This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b01  This control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL1 to be trapped.
0b10  This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b11  This control does not cause execution of any instructions to be trapped.

Writes to MVFR0, MVFR1, and MVFR2 from EL1 or higher are CONSTRAINED UNPREDICTABLE and whether these accesses can be trapped by this control depends on implemented CONSTRAINED UNPREDICTABLE behavior.

Note
• Attempts to write to the FPSID count as use of the registers for accesses from EL1 or higher.
• Accesses from EL0 to FPSID, MVFR0, MVFR1, MVFR2, and FPEXC are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPACR_EL1.FPEN is not 0b11.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:18]
Reserved, RES0.

ZEN, bits [17:16]

When FEAT_SVE is implemented:
Traps execution at EL1 and EL0 of SVE instructions when the PE is not in Streaming SVE mode, and instructions that directly access the ZCR_EL1 System register to EL1, or to EL2 when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE is 1.
The exception is reported using ESR_ELx.EC value 0x19.

A trap taken as a result of CPACR_EL1.ZEN has precedence over a trap taken as a result of CPACR_EL1.FPEN.

0b00  This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b01  This control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL1 to be trapped.
0b10  This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b11  This control does not cause execution of any instructions to be trapped.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bits [15:0]
Reserved, RES0.

Accessing CPACR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CPACR_EL1 or CPACR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:
MRS \langle X_t \rangle, CPACR_EL1

if PSTATE.EL == EL0 then
   UNDEFINED;
elsif PSTATE.EL == EL1 then
   if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
      UNDEFINED;
   elsif EL2Enabled() && CPTR_EL2.TCPAC == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.CPACR_EL1 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
      if Halted() && EDSCR.SDD == '1' then
         UNDEFINED;
      else
         AArch64.SystemAccessTrap(EL3, 0x18);
      end if;
   else
      X[t, 64] = CPACR_EL1;
   end if;
elsif PSTATE.EL == EL2 then
   if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
      UNDEFINED;
   elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
      if Halted() && EDSCR.SDD == '1' then
         UNDEFINED;
      else
         AArch64.SystemAccessTrap(EL3, 0x18);
      end if;
   elsif HCR_EL2.E2H == '1' then
      X[t, 64] = CPTR_EL2;
   else
      X[t, 64] = CPACR_EL1;
   end if;
elsif PSTATE.EL == EL3 then
   X[t, 64] = CPACR_EL1;

MSR CPACR_EL1, \langle X_t \rangle

if PSTATE.EL == EL0 then
   UNDEFINED;
elsif PSTATE.EL == EL1 then
   if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
      UNDEFINED;
   elsif EL2Enabled() && CPTR_EL2.TCPAC == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.CPACR_EL1 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
      if Halted() && EDSCR.SDD == '1' then
         UNDEFINED;
      else
         AArch64.SystemAccessTrap(EL3, 0x18);
      end if;
   else
      X[t, 64] = CPACR_EL1;
   end if;
elsif PSTATE.EL == EL2 then
   if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
      UNDEFINED;
   elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
      if Halted() && EDSCR.SDD == '1' then
         UNDEFINED;
      else
         AArch64.SystemAccessTrap(EL3, 0x18);
      end if;
   elsif HCR_EL2.E2H == '1' then
      X[t, 64] = CPTR_EL2;
   else
      X[t, 64] = CPACR_EL1;
   end if;
elsif PSTATE.EL == EL3 then
   X[t, 64] = CPACR_EL1;
AArch64.SystemAccessTrap(EL3, 0x18);
elself EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
  NVMem[0x100] = X[t, 64];
else
  CPACR_EL1 = X[t, 64];
elself PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
    UNDEFINED;
elself HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
elself HCR_EL2.E2H == '1' then
  CPTR_EL2 = X[t, 64];
else
  CPACR_EL1 = X[t, 64];
elself PSTATE_EL == EL3 then
  CPACR_EL1 = X[t, 64];

MRS <Xt>, CPACR_EL12

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elself PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    X[t, 64] = NVMem[0x100];
elself EL2Enabled() && HCR_EL2.NV == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elself PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
      UNDEFINED;
elself HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = CPACR_EL1;
else
  UNDEFINED;
elself PSTATE_EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    X[t, 64] = CPACR_EL1;
else
  UNDEFINED;
**MSR CPACR_EL12, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV2,NV1,NV == '101' then
    NVMem[0x100] = X[t, 64];
  elseif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
      UNDEFINED;
    elseif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    else
      CPACR_EL1 = X[t, 64];
    end
  elseif PSTATE_EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
      CPACR_EL1 = X[t, 64];
    else
      UNDEFINED;
  end
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV2,NV1,NV == '101' then
    NVMem[0x100] = X[t, 64];
  elseif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
end
### D17.2.31 CPTR_EL2, Architectural Feature Trap Register (EL2)

The CPTR_EL2 characteristics are:

**Purpose**

Controls trapping to EL2 of accesses to CPACR, CPACR_EL1, trace, Activity Monitor, SME, Streaming SVE, SVE, and Advanced SIMD and floating-point functionality.

**Configurations**

AArch64 System register CPTR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HCPTR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

CPTR_EL2 is a 64-bit register.

**Field descriptions**

*When FEAT_VHE is implemented and HCR_EL2.E2H == 1:*

<table>
<thead>
<tr>
<th>Field</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCPAC, bit [31]</td>
<td>In AArch64 state, traps accesses to CPACR_EL1 from EL1 to EL2, when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC value 0x18.</td>
</tr>
<tr>
<td>In AArch32 state, traps accesses to CPACR from EL1 to EL2, when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC value 0x03.</td>
<td></td>
</tr>
<tr>
<td>0b0</td>
<td>This control does not cause any instructions to be trapped.</td>
</tr>
<tr>
<td>0b1</td>
<td>EL1 accesses to CPACR_EL1 and CPACR are trapped to EL2, when EL2 is enabled in the current Security state.</td>
</tr>
</tbody>
</table>

**TCPAC, bit [31]**

In AArch64 state, traps accesses to CPACR_EL1 from EL1 to EL2, when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC value 0x18.

In AArch32 state, traps accesses to CPACR from EL1 to EL2, when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC value 0x03.

0b0 This control does not cause any instructions to be trapped.

0b1 EL1 accesses to CPACR_EL1 and CPACR are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

---

**Note**

CPACR_EL1 and CPACR are not accessible at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
TAM, bit [30]

When FEAT_AMUv1 is implemented:

Trap Activity Monitor access. Traps EL1 and EL0 accesses to all Activity Monitor registers to EL2, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using ESR_ELx.EC value 0x18:
  - AMUSERENR_EL0, AMCFGR_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0,
    AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0,
    AMCR_EL0, AMEVCNTR0<n>_EL0, AMEVCNTR1<n>_EL0,
    AMEVTYPER0<n>_EL0, and AMEVTYPER1<n>_EL0.

- In AArch32 state, MRC or MCR accesses to the following registers are trapped to EL2 and reported using ESR_ELx.EC value 0x03:
  - AMUSERENR, AMCFGR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1,
    AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<n>, and
    AMEVTYPER1<n>.

- In AArch32 state, MRRC or MCRR accesses to AMEVCNTR0<n> and AMEVCNTR1<n>, are trapped to EL2, reported using ESR_ELx.EC value 0x04.

0b0 Accesses from EL1 and EL0 to Activity Monitor registers are not trapped.

0b1 Accesses from EL1 and EL0 to Activity Monitor registers are trapped to EL2, when EL2 is enabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [29]

Reserved, RES0.

TTA, bit [28]

Traps System register accesses to all implemented trace registers from both Execution states to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to trace registers with op0=2, op1=1, and CRn<0b1000 are trapped to EL2, reported using EC syndrome value 0x18.

- In AArch64 state, MRC or MCR accesses to trace registers with cpnum=14, opc1=1, and CRn<0b1000 are trapped to EL2, reported using EC syndrome value 0x05.

0b0 This control does not cause any instructions to be trapped.

0b1 Any attempt at EL0, EL1 or EL2, to execute a System register access to an implemented trace register is trapped to EL2, when EL2 is enabled in the current Security state, unless HCR_EL2.TGE is 0 and it is trapped by CPACR.NSTRCDIS or CPACR_EL1.TTA. When HCR_EL2.TGE is 1, any attempt at EL0 or EL2 to execute a System register access to an implemented trace register is trapped to EL2, when EL2 is enabled in the current Security state.

Note

The ETMv4 architecture and ETE do not permit EL0 to access the trace registers. If the trace unit implements FEAT_ETMv4 or ETE, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.TTA is 1.

EL2 does not provide traps on trace register accesses through the optional Memory-mapped interface.
System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not supported, this bit is RES0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [27:26]**

Reserved, RES0.

**SMEN, bits [25:24]**

*When FEAT_SME is implemented:*

Traps execution at EL2, EL1, and EL0 of SME instructions, SVE instructions when FEAT_SVE is not implemented or the PE is in Streaming SVE mode, and instructions that directly access the SVCR, SMCR_EL1, or SMCR_EL2 System registers to EL2, when EL2 is enabled in the current Security state.

When instructions that directly access the SVCR System register are trapped with reference to this control, the MSR SVCRSM, MSR SVCRZA, and MSR SVCRSMZA instructions are also trapped.

The exception is reported using ESR_EL2.EC value of 0x1D, with an ISS code of 0x0000000.

This field does not affect whether Streaming SVE or SME register values are valid.

A trap taken as a result of CPTR_EL2.SMEN has precedence over a trap taken as a result of CPTR_EL2.FPEN.

- **0b00** This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
- **0b01** When HCR_EL2:TGE is 0, this control does not cause execution of any instructions to be trapped.
  
  When HCR_EL2:TGE is 1, this control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL2 to be trapped.
- **0b10** This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
- **0b11** This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

**Bits [23:22]**

Reserved, RES0.

**FPEN, bits [21:20]**

Traps execution at EL2, EL1, and EL0 of instructions that access the Advanced SIMD and floating-point registers from both Execution states to EL2, when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC value 0x87.

Traps execution at EL2, EL1, and EL0 of SME and SVE instructions to EL2, when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC value 0x87.

A trap taken as a result of CPTR_EL2.SMEN has precedence over a trap taken as a result of CPTR_EL2.FPEN.

A trap taken as a result of CPTR_EL2.ZEN has precedence over a trap taken as a result of CPTR_EL2.FPEN.

- **0b00** This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
- **0b01** When HCR_EL2:TGE is 0, this control does not cause execution of any instructions to be trapped.
When HCR_EL2.TGE is 1, this control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL2 to be trapped.

0b10 This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b11 This control does not cause execution of any instructions to be trapped.

Writes to MVFR0, MVFR1, and MVFR2 from EL1 or higher are CONSTRAINED UNPREDICTABLE and whether these accesses can be trapped by this control depends on implemented CONSTRAINED UNPREDICTABLE behavior.

Note

- Attempts to write to the FPSID count as use of the registers for accesses from EL1 or higher.
- Accesses from EL0 to FPSID, MVFR0, MVFR1, MVFR2, and FPEXC are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.FPEN is not 0b11.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:18]
Reserved, RES0.

ZEN, bits [17:16]

When FEAT_SVE is implemented:

Traps execution at EL2, EL1, and EL0 of SVE instructions when the PE is not in Streaming SVE mode, and instructions that directly access the ZCR_EL1 or ZCR_EL2 System registers to EL2, when EL2 is enabled in the current Security state.

The exception is reported using ESR_ELx.ec value 0x19.

A trap taken as a result of CPTR_EL2.ZEN has precedence over a trap taken as a result of CPTR_EL2.FPEN.

0b00 This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b01 When HCR_EL2.TGE is 0, this control does not cause execution of any instructions to be trapped.

When HCR_EL2.TGE is 1, this control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL2 to be trapped.

0b10 This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b11 This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bits [15:0]
Reserved, RES0.
Otherwise:

This format applies in all Armv8.0 implementations.

**Bits [63:32]**

Reserved, RES0.

**TCPAC, bit [31]**

In AArch64 state, traps accesses to CPACR_EL1 from EL1 to EL2, when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC value 0x18.

In AArch32 state, traps accesses to CPACR from EL1 to EL2, when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC value 0x03.

0b0 This control does not cause any instructions to be trapped.

0b1 EL1 accesses to the following registers are trapped to EL2, when EL2 is enabled in the current Security state:

- CPACR_EL1.
- CPACR.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

--- **Note** ---

CPACR_EL1 and CPACR are not accessible at EL0.

--- The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**TAM, bit [30]**

*When FEAT_AMUv1 is implemented:*

Trap Activity Monitor access. Traps EL1 and EL0 accesses to all Activity Monitor registers to EL2, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using ESR_ELx.EC value 0x18:

  - AMUSERENR_EL0, AMCFGR_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0, AMCR_EL0, AMEVCNTR0<n>_EL0, AMEVCNTR1<n>_EL0, AMEVTYPER0<n>_EL0, and AMEVTYPER1<n>_EL0.

- In AArch32 state, MCR or MRC accesses to the following registers are trapped to EL2 and reported using ESR_ELx.EC value 0x03:

  - AMUSERENR, AMCFGR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<n>, and AMEVTYPER1<n>.

- In AArch32 state, MCRR or MRRC accesses to AMEVCNTR0<n> and AMEVCNTR1<n>, are trapped to EL2, reported using ESR_ELx.EC value 0x04.

0b0 Accesses from EL1 and EL0 to Activity Monitor registers are not trapped.
Accesses from EL1 and EL0 to Activity Monitor registers are trapped to EL2, when EL2 is enabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bits [29:21]**

Reserved, RES0.

**TTA, bit [20]**

Traps System register accesses to all implemented trace registers from both Execution states to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to trace registers with op0=2, op1=1, and CRn<0b1000 are trapped to EL2, reported using EC syndrome value 0x18.
- In AArch32 state, MRC or MCR accesses to trace registers with cpnum=14, opc1=1, and CRn<0b1000 are trapped to EL2, reported using EC syndrome value 0x05.

0b0 This control does not cause any instructions to be trapped.

0b1 Any attempt at EL0, EL1, or EL2, to execute a System register access to an implemented trace register is trapped to EL2, when EL2 is enabled in the current Security state, unless it is trapped by one of the following controls:

- CPACR_EL1.TTA
- CPACR.TRCDIS

**Note**

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the trace unit implements FEAT_ETMv4, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.TTA is 1.
- EL2 does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not supported, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [19:14]**

Reserved, RES0.

**Bit [13]**

Reserved, RES1.

**TSM, bit [12]**

*When FEAT_SME is implemented:*

Traps execution at EL2, EL1, and EL0 of SME instructions, SVE instructions when FEAT_SVE is not implemented or the PE is in Streaming SVE mode, and instructions that directly access the SVCR, SMCR_EL1, or SMCR_EL2 System registers to EL2, when EL2 is enabled in the current Security state.

When instructions that directly access the SVCR System register are trapped with reference to this control, the MSR SVECSR, MSR SVCRZA, and MSR SVCRSMZA instructions are also trapped.
The exception is reported using ESR_EL2.EC value of 0x1D, with an ISS code of 0x0000000.
This field does not affect whether Streaming SVE or SME register values are valid.

A trap taken as a result of CPTR_EL2.TSM has precedence over a trap taken as a result of CPTR_EL2.TFP.

0b0  This control does not cause execution of any instructions to be trapped.
0b1  This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES1.

**Bit [11]**

Reserved, RES0.

**TFP, bit [10]**

Traps execution of instructions which access the Advanced SIMD and floating-point functionality, from both Execution states to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using ESR_ELx.EC value 0x07:
  - FPCR, FPSR, FPEXC32_EL2, any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers.
- In AArch32 state, accesses to the following registers are trapped to EL2, reported using ESR_ELx.EC value 0x07:
  - MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-15, including their views as D0-D31 registers or S0-31 registers. For the purposes of this trap, the architecture defines a VMSR access to FPSID from EL1 or higher as an access to a SIMD and floating-point register. Otherwise, permitted VMSR accesses to FPSID are ignored.

Traps execution at the same Exception levels of SME and SVE instructions to EL2, when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC value 0x07.

A trap taken as a result of CPTR_EL2.TSM has precedence over a trap taken as a result of CPTR_EL2.TFP.

A trap taken as a result of CPTR_EL2.TZ has precedence over a trap taken as a result of CPTR_EL2.TFP.

0b0  This control does not cause execution of any instructions to be trapped.
0b1  This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.

--- **Note** ---

FPEXC32_EL2 is not accessible from EL0 using AArch64.
FPSID, MVFR0, MVFR1, and FPEXC are not accessible from EL0 using AArch32.

---

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [9]**

Reserved, RES1.

**TZ, bit [8]**

*When FEAT_SVE is implemented:*

Traps execution at EL2, EL1, and EL0 of SVE instructions when the PE is not in Streaming SVE mode, and instructions that directly access the ZCR_EL2 or ZCR_EL1 System registers to EL2, when EL2 is enabled in the current Security state.
The exception is reported using ESR_ELx.EC value 0x19.
A trap taken as a result of CPTR_EL2.TZ has precedence over a trap taken as a result of CPTR_EL2.TFP.

0b0  This control does not cause execution of any instructions to be trapped.
0b1  This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES1.

Bits [7:0]
Reserved, RES1.

Accessing CPTR_EL2
Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS } \langle X_t \rangle, \text{ CPTR_EL2}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

\[
\text{MSR CPTR_EL2, } \langle X_t \rangle
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elseif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
    UNDEFINED;
elseif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = CPTR_EL2;
elseif PSTATE.EL == EL3 then
  X[t, 64] = CPTR_EL2;
else
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'' & CPTR_EL3.TCPAC == '1'
  then
    UNDEFINED;
  elsif HaveEL(EL3) & CPTR_EL3.TCPAC == '1'
  then
    if Halted() & EDSCR.SDD == '1'
    then
      UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    CPTR_EL2 = X[t, 64];
else
  if PSTATE.EL == EL3 then
    CPTR_EL2 = X[t, 64];
else
  AArch64.SystemAccessTrap(EL3, 0x18);
elsif EL2Enabled() & CPTR_EL2.TCPAC == '1'
  then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFGRTR_EL2.CPACR_EL1 == '1'
  then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TCPAC == '1'
  then
    if Halted() & EDSCR.SDD == '1'
    then
      UNDEFINED;
  elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111'
  then
    X[t, 64] = NVMem[0x100];
else
  X[t, 64] = CPACR_EL1;
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'' & CPTR_EL3.TCPAC == '1'
  then
    UNDEFINED;
  elsif HaveEL(EL3) & CPTR_EL3.TCPAC == '1'
  then
    if Halted() & EDSCR.SDD == '1'
    then
      UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
elsif EL2Enabled() & HCR_EL2.E2H == '1'
  then
    X[t, 64] = CPTR_EL2;
else
  X[t, 64] = CPACR_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = CPACR_EL1;

MRS <Xt>, CPACR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>
**MSR CPACR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
 UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
    UNDEFINED;
  elsif EL2Enabled() && CPTR_EL2.TCPAC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.CPACR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    NVMem[0x100] = X[t, 64];
  else
    CPACR_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  elsif HCR_EL2.E2H == '1' then
    CPTR_EL2 = X[t, 64];
  else
    CPACR_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL3 then
  CPACR_EL1 = X[t, 64];
D17.2.32 CPTR_EL3, Architectural Feature Trap Register (EL3)

The CPTR_EL3 characteristics are:

**Purpose**

Controls trapping to EL3 of accesses to CPACR, CPACR_EL1, HCPTR, CPTR_EL2, trace, Activity Monitor, SME, Streaming SVE, SVE, and Advanced SIMD and floating-point functionality.

**Configurations**

This register is present only when EL3 is implemented. Otherwise, direct accesses to CPTR_EL3 are UNDEFINED.

**Attributes**

CPTR_EL3 is a 64-bit register.

**Field descriptions**

![Field descriptions diagram]

**Bits [63:32]**

Reserved, RES0.

**TCPAC, bit [31]**

Traps all of the following to EL3, from both Execution states and any Security state.

- EL2 accesses to CPTR_EL2, reported using ESR_ELx.EC value 0x18, or HCPTR, reported using ESR_ELx.EC value 0x03.
- EL2 and EL1 accesses to CPACR_EL1 reported using ESR_ELx.EC value 0x18, or CPACR reported using ESR_ELx.EC value 0x03.

When CPTR_EL3.TCPAC is:

- **0b0** This control does not cause any instructions to be trapped.
- **0b1** EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR, are trapped to EL3, unless they are trapped by CPTR_EL2.TCPAC.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**TAM, bit [30]**

*When FEAT_AMUv1 is implemented:*

Trap Activity Monitor access. Traps EL2, EL1, and EL0 accesses to all Activity Monitor registers to EL3.
Accesses to the Activity Monitors registers are trapped as follows:

- In AArch64 state, the following registers are trapped to EL3 and reported with ESR_ELx.EC value 0x18:
  - AMUSERENR_EL0, AMCFG_R_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0, AMCR_EL0, AMEVCNTR0<\alpha> EL0, AMEVCNTR1<\alpha> EL0, AMEVTYPER0<\alpha> EL0, and AMEVTYPER1<\alpha> EL0.

- In AArch32 state, accesses with MRC or MCR to the following registers reported with ESR_ELx.EC value 0x03:
  - AMUSERENR, AMCFG, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<\alpha>, and AMEVTYPER1<\alpha>.

- In AArch32 state, accesses with MRRC or MCRR to the following registers, reported with ESR_ELx.EC value 0x04:
  - AMEVCNTR0<\alpha>, AMEVCNTR1<\alpha>.

0b0 Accesses from EL2, EL1, and EL0 to Activity Monitor registers are not trapped.
0b1 Accesses from EL2, EL1, and EL0 to Activity Monitor registers are trapped to EL3.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Another:
Reserved, RES0.

Bits [29:21]
Reserved, RES0.

TTA, bit [20]
Traps System register accesses. Accesses to the trace registers, from all Exception levels, any Security state, and both Execution states are trapped to EL3 as follows:
- In AArch64 state, Trace registers with op0=2, op1=1, and CRn<@b100 are trapped to EL3 and reported using EC syndrome value 0x18.
- In AArch32 state, accesses using MCR or MRC to the Trace registers with cpnum=14, opc1=1, and CRn<@b100 are reported using EC syndrome value 0x05.
0b0 This control does not cause any instructions to be trapped.
0b1 Any System register access to the trace registers is trapped to EL3, unless it is trapped by CPACR.TRCDIS, CPACR_EL1.TTA, or CPTR_EL2.TTA.

If System register access to trace functionality is not supported, this bit is RES0.

Note
The ETMv4 architecture and ETE do not permit EL0 to access the trace registers. If the trace unit implements FEAT_ETMv4 or FEAT_ETE, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than this trap exception.

EL3 does not provide traps on trace register accesses through the Memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see Configurable instruction controls on page D1-4665.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:13]
Reserved, RES0.
ESM, bit [12]

When FEAT_SME is implemented:

Traps execution of SME instructions, SVE instructions when FEAT_SVE is not implemented or the PE is in Streaming SVE mode, and instructions that directly access the SMCR_EL1, SMCR_EL2, SMCR_EL3, SMPRI_EL1, SMPRIMAP_EL2, or SVCR System registers, from all Exception levels and any Security state, to EL3.

When instructions that directly access the SVCR System register are trapped with reference to this control, the MSR SVCRSM, MSR SVCRZA, and MSR SVCRSMZA instructions are also trapped.

When direct accesses to SMPRI_EL1 and SMPRIMAP_EL2 are trapped, the exception is reported using an ESR_EL3.EC value of 0x18. Otherwise, the exception is reported using an ESR_EL3.EC value of 0x1D, with an ISS code of 0x00000000.

This field does not affect whether Streaming SVE or SME register values are valid.

A trap taken as a result of CPTR_EL3.ESM has precedence over a trap taken as a result of CPTR_EL3.TFP.

0b0 This control causes execution of these instructions at all Exception levels to be trapped.

0b1 This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [11]

Reserved, RES0.

TFP, bit [10]

Traps execution of instructions which access the Advanced SIMD and floating-point functionality, from all Exception levels, any Security state, and both Execution states, to EL3.

This includes the following registers, all reported using ESR_ELx.EC value 0x07:

- FPCR, FPSR, FPEXC32_EL2, and any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-S31 registers.
- MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers.
- VMSR accesses to FPSID.

Permitted VMSR accesses to FPSID are ignored, but for the purposes of this trap the architecture defines a VMSR access to the FPSID from EL1 or higher as an access to a SIMD and floating-point register.

Traps execution at all Exception levels of SME and SVE instructions to EL3 from any Security state. The exception is reported using ESR_ELx.EC value 0x07.

A trap taken as a result of CPTR_EL3.ESM has precedence over a trap taken as a result of CPTR_EL3.TFP.

A trap taken as a result of CPTR_EL3.ESM has precedence over a trap taken as a result of CPTR_EL3.TFP.

Defined values are:

0b0 This control does not cause execution of any instructions to be trapped.

0b1 This control causes execution of these instructions at all Exception levels to be trapped.

Note

FPEXC32_EL2 is not accessible from EL0 using AArch64.

FPSID, MVFR0, MVFR1, and FPEXC are not accessible from EL0 using AArch32.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [9]**

Reserved, RES0.

**EZ, bit [8]**

*When FEAT_SVE is implemented:*

Traps execution of SVE instructions when the PE is not in Streaming SVE mode, and instructions that directly access the ZCR_EL3, ZCR_EL2, or ZCR_EL1 System registers, from all Exception levels and any Security state, to EL3.

The exception is reported using ESR_ELx.EC value 0x19.

A trap taken as a result of CPTR_EL3.EZ has precedence over a trap taken as a result of CPTR_EL3.TFP.

  - 0b0       This control causes execution of these instructions at all Exception levels to be trapped.
  - 0b1       This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bits [7:0]**

Reserved, RES0.

**Accessing CPTR_EL3**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, CPTR_EL3**

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b1 & 0b10 & 0b0001 & 0b0001 & 0b010 \\
\end{array}
\]

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  UNDEFINED;
elsif PSTATE_EL == EL2 then
  UNDEFINED;
elsif PSTATE_EL == EL3 then
  X[t, 64] = CPTR_EL3;

**MSR CPTR_EL3, <Xt>**

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b1 & 0b110 & 0b0001 & 0b0001 & 0b010 \\
\end{array}
\]

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  UNDEFINED;

UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  CPTR_EL3 = X[t, 64];
D17.2.33 CSSELR_EL1, Cache Size Selection Register

The CSSELR_EL1 characteristics are:

**Purpose**

Selects the current Cache Size ID Register, CCSIDR_EL1, by specifying the required cache level and the cache type (either instruction or data cache).

**Configurations**

AArch64 System register CSSELR_EL1 bits [31:0] are architecturally mapped to AArch32 System register CSSELR[31:0].

**Attributes**

CSSELR_EL1 is a 64-bit register.

**Field descriptions**

![Field Diagram]

**Bits [63:5]**

Reserved, RES0.

**TnD, bit [4]**

*When FEAT_MTE2 is implemented:*

Allocation Tag not Data bit.

0b0  Data, Instruction or Unified cache.

0b1  Separate Allocation Tag cache.

When CSSELR_EL1.InD == 1, this bit is RES0.

If CSSELR_EL1. {TnD, Level, InD} is programmed to a cache level that is not implemented, then the value for this field on a read of CSSELR_EL1 is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

**Level, bits [3:1]**

Cache level of required cache.

0b000  Level 1 cache.

0b001  Level 2 cache.

0b010  Level 3 cache.

0b011  Level 4 cache.

0b100  Level 5 cache.

0b101  Level 6 cache.

0b110  Level 7 cache.

All other values are reserved.
If CSSELR_EL1.\{TnD, Level, InD\} is programmed to a cache level that is not implemented, then
the value for this field on a read of CSSELR_EL1 is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**InD, bit [0]**

Instruction not Data bit.

0b0 Data or unified cache.

0b1 Instruction cache.

If CSSELR_EL1.\{TnD, Level, InD\} is programmed to a cache level that is not implemented, then
a read of CSSELR_EL1 is CONSTRAINED UNPREDICTABLE, and returns UNKNOWN values for
CSSELR_EL1.\{Level, InD\}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architectually UNKNOWN value.

**Accessing CSSELR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, CSSELR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b010</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TID2 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TID4 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.CSSELR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = CSSELR_EL1;
  elsif PSTATE.EL == EL2 then
    X[t, 64] = CSSELR_EL1;
  elsif PSTATE.EL == EL3 then
    X[t, 64] = CSSELR_EL1;

**MSR CSSELR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b010</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TID2 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.TID4 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.CSSELR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
AArch64 System Register Descriptions
D17.2 General system control registers

```c
AArch64.SystemAccessTrap(EL2, 0x18);
else
    CSSELRL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    CSSELRL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    CSSELRL1 = X[t, 64];
```
D17.2.34 CTR_EL0, Cache Type Register

The CTR_EL0 characteristics are:

**Purpose**

Provides information about the architecture of the caches.

**Configurations**

AArch64 System register CTR_EL0 bits [31:0] are architecturally mapped to AArch32 System register CTR[31:0].

**Attributes**

CTR_EL0 is a 64-bit register.

**Field descriptions**

![Field descriptions diagram]

**Bits [63:38]**

Reserved, RES0.

**TminLine, bits [37:32]**

*When FEAT_MTE2 is implemented:*

Tag minimum Line. $\log_2$ of the number of words covered by Allocation Tags in the smallest cache line of all caches which can contain Allocation tags that are controlled by the PE.

**Note**

- For an implementation with cache lines containing 64 bytes of data and 4 Allocation Tags, this will be $\log_2(64/4) = 4$.
- For an implementation with Allocations Tags in separate cache lines of 128 Allocation Tags per line, this will be $\log_2(128*16/4) = 9$.

**Otherwise:**

Reserved, RES0.

**Bit [31]**

Reserved, RES1.

**Bit [30]**

Reserved, RES0.

**DIC, bit [29]**

Instruction cache invalidation requirements for data to instruction coherence.

- **0b0** Instruction cache invalidation to the Point of Unification is required for data to instruction coherence.
- **0b1** Instruction cache invalidation to the Point of Unification is not required for data to instruction coherence.
IDC, bit [28]

Data cache clean requirements for instruction to data coherence. The meaning of this bit is:

- **0b0** Data cache clean to the Point of Unification is required for instruction to data coherence, unless CLIDR_EL1.LoC == 0b000 or (CLIDR_EL1.LoUIS == 0b000 && CLIDR_EL1.LoUU == 0b000).
- **0b1** Data cache clean to the Point of Unification is not required for instruction to data coherence.

CWG, bits [27:24]

Cache writeback granule. \(\log_2\) of the number of words of the maximum size of memory that can be overwritten as a result of the eviction of a cache entry that has had a memory location in it modified. A value of 0b0000 indicates that this register does not provide Cache writeback granule information and either:

- The architectural maximum of 512 words (2KB) must be assumed.
- The Cache writeback granule can be determined from maximum cache line size encoded in the Cache Size ID Registers.

Values greater than 0b1001 are reserved.

Arm recommends that an implementation that does not support cache write-back implements this field as 0b0001. This applies, for example, to an implementation that supports only write-through caches.

ERG, bits [23:20]

Exclusives reservation granule, and, if FEAT_TME is implemented, transactional reservation granule. \(\log_2\) of the number of words of the maximum size of the reservation granule for the Load-Exclusive and Store-Exclusive instructions, and, if FEAT_TME is implemented, for detecting transactional conflicts.

A value of 0b0000 indicates that this register does not provide granule information and the architectural maximum of 512 words (2KB) must be assumed.

Value 0b0001 and values greater than 0b1001 are reserved.

DminLine, bits [19:16]

\(\log_2\) of the number of words in the smallest cache line of all the data caches and unified caches that are controlled by the PE.

L1Ip, bits [15:14]

Level 1 instruction cache policy. Indicates the indexing and tagging policy for the L1 instruction cache. Possible values of this field are:

- **0b0** *When FEAT_VPIPT is implemented:*
  VMID aware Physical Index, Physical tag (VPIPT).
- **0b1** ASID-tagged Virtual Index, Virtual Tag (AIVIVT).
- **0b10** Virtual Index, Physical Tag (VIPT).
- **0b11** Physical Index, Physical Tag (PIPT).

From Armv8, the value 0b01 is reserved.

The value 0b00 is permitted only in an implementation that includes FEAT_VPIPT.

Bits [13:4]

Reserved, RES0.

IminLine, bits [3:0]

\(\log_2\) of the number of words in the smallest cache line of all the instruction caches that are controlled by the PE.
Accessing CTR_EL0

Accesses to this register use the following encodings in the System register encoding space:

\[ MRS <X_t>, CTR_EL0 \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```
if PSTATE.EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '1') && SCTLR_EL1.UCT == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.TID2 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
    HFGRTR_EL2.CTR_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = CTR_EL0;
  endif
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TID2 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.CTR_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = CTR_EL0;
  endif
elsif PSTATE.EL == EL2 then
  X[t, 64] = CTR_EL0;
elsif PSTATE.EL == EL3 then
  X[t, 64] = CTR_EL0;
```

```
D17.2.35 DACR32_EL2, Domain Access Control Register

The DACR32_EL2 characteristics are:

Purpose

Allows access to the AArch32 DACR register from AArch64 state only. Its value has no effect on execution in AArch64 state.

Configurations

AArch64 System register DACR32_EL2 bits [31:0] are architecturally mapped to AArch32 System register DACR[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DACR32_EL2 are UNDEFINED.

If EL2 is not implemented but EL3 is implemented, and EL1 is capable of using AArch32, then this register is not RES0.

Attributes

DACR32_EL2 is a 64-bit register.

Field descriptions

Bits [63:32]

Reserved, RES0.

D<n>, bits [2n+1:2n], for n = 15 to 0

Domain n access permission, where n = 0 to 15. Permitted values are:

0b00 No access. Any access to the domain generates a Domain fault.

0b01 Client. Accesses are checked against the permission bits in the translation tables.

0b11 Manager. Accesses are not checked against the permission bits in the translation tables.

The value 0b10 is reserved.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DACR32_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, DACR32_EL2

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0011</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then

UNDEFINED;

elsif PSTATE.EL == EL1 then
if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = DACR32_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = DACR32_EL2;

MSR DACR32_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0011</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        DACR32_EL2 = X[t, 64];
    elsif PSTATE.EL == EL3 then
        DACR32_EL2 = X[t, 64];
D17.2.36 DCZID_EL0, Data Cache Zero ID register

The DCZID_EL0 characteristics are:

**Purpose**

Indicates the block size that is written with byte values of 0 by the DC ZVA (Data Cache Zero by Address) System instruction.

If FEAT_MTE is implemented, this register also indicates the granularity at which the DC GVA and DC GZVA instructions write.

**Configurations**

There are no configuration notes.

**Attributes**

DCZID_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:5]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DZP, bit [4]</td>
<td>Data Zero Prohibited. This field indicates whether use of DC ZVA instructions is permitted or prohibited.</td>
</tr>
<tr>
<td>BS, bits [3:0]</td>
<td>Log2 of the block size in words. The maximum size supported is 2KB (value == 9).</td>
</tr>
</tbody>
</table>

**Accessing DCZID_EL0**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, DCZID_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then

if EL2Enabled() && HCR_EL2.«E2H,TGE» != '1' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HFGRTR_EL2.DCZID_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    X[t, 64] = DCZID_EL0;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.DCZID_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = DCZID_EL0;
elsif PSTATE_EL == EL2 then
    X[t, 64] = DCZID_EL0;
elsif PSTATE_EL == EL3 then
    X[t, 64] = DCZID_EL0;
D17.2.37 ESR_EL1, Exception Syndrome Register (EL1)

The ESR_EL1 characteristics are:

**Purpose**

Holds syndrome information for an exception taken to EL1.

**Configurations**

AArch64 System register ESR_EL1 bits \([31:0]\) are architecturally mapped to AArch32 System register DFSR[31:0].

**Attributes**

ESR_EL1 is a 64-bit register.

**Field descriptions**

ESR_EL1 is made **UNKNOWN** as a result of an exception return from EL1.

When an **UNPREDICTABLE** instruction is treated as **UNDEFINED**, and the exception is taken to EL1, the value of ESR_EL1 is **UNKNOWN**. The value written to ESR_EL1 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

**Bits [63:37]**

Reserved, **RES0**.

**ISS2, bits [36:32]**

*When FEAT_LS64 is implemented:*

When FEAT_LS64 V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort exception for a Translation fault, Access flag fault, or Permission fault, then this field holds register specifier, Xs.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort exception for a Translation fault, Access flag fault, or Permission fault, then this field holds register specifier, Xs.

Otherwise, this field is **RES0**.

*Otherwise:*

Reserved, **RES0**.

**EC, bits [31:26]**

Exception Class. Indicates the reason for the exception that this register holds information about.

For each EC value, the table references a subsection that gives information about:

- The cause of the exception, for example the configuration required to enable the trap.
- The encoding of the associated ISS.

Possible values of the EC field are:

**EC == 0b00000000**

Unknown reason.

See **ISS encoding for exceptions with an unknown reason**.
EC == 0b000001
Trapped WF* instruction execution.
Conditional WF* instructions that fail their condition code check do not cause an exception.
See ISS encoding for an exception from a WF* instruction.

EC == 0b000011
When AArch32 is supported:
Trapped MCR or MRC access with (coproc==0b1111) that is not reported using EC 0b000000.
See ISS encoding for an exception from an MCR or MRC access.

EC == 0b000100
When AArch32 is supported:
Trapped MCRR or MRRC access with (coproc==0b1111) that is not reported using EC 0b000000.
See ISS encoding for an exception from an MCRR or MRRC access.

EC == 0b000101
When AArch32 is supported:
Trapped MCRR or MRRC access with (coproc==0b1110).
See ISS encoding for an exception from an MCRR or MRRC access.

EC == 0b000110
When AArch32 is supported:
Trapped LDC or STC access.
The only architected uses of these instruction are:
• An STC to write data to memory from DBGDTRRXint.
• An LDC to read data from memory to DBGDTRTXint.
See ISS encoding for an exception from an LDC or STC instruction.

EC == 0b000111
Access to SME, SVE, Advanced SIMD or floating-point functionality trapped by CPACR_EL1.FPEN, CPTR_EL2.FPEN, CPTR_EL2.TFP, or CPTR_EL3.TFP control.
Excludes exceptions resulting from CPACR_EL1 when the value of HCR_EL2.TGE is 1, or because SVE or Advanced SIMD and floating-point are not implemented. These are reported with EC value 0b000000.
See ISS encoding for an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from the FPEN and TFP traps.

EC == 0b001010
When FEAT_LS64 is implemented:
Trapped execution of an LD64B or ST64B* instruction.
See ISS encoding for an exception from an LD64B or ST64B* instruction.

EC == 0b001100
When AArch32 is supported:
Trapped MRRC access with (coproc==0b1110).
See ISS encoding for an exception from an MCRR or MRRC access.

EC == 0b001101
When FEAT_BTI is implemented:
Branch Target Exception.
See ISS encoding for an exception from Branch Target Identification instruction.

EC == 0b001110
Illegal Execution state.
See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.
EC == 0b010001
When AArch32 is supported:
SVC instruction execution in AArch32 state.
See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 0b010101
When AArch64 is supported:
SVC instruction execution in AArch64 state.
See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 0b011000
When AArch64 is supported:
Trapped MSR, MRS or System instruction execution in AArch64 state, that is not
reported using EC 0b000000, 0b000001, or 0b000111.
This includes all instructions that cause exceptions that are part of the encoding space
defined in System instruction class encoding overview on page C5-655, except for those
exceptions reported using EC values 0b000000, 0b000001, or 0b000111.
See ISS encoding for an exception from MSR, MRS, or System instruction execution in
AArch64 state.

EC == 0b011001
When FEAT_SVE is implemented:
Access to SVE functionality trapped as a result of CPACR_EL1.ZEN,
CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ, that is not reported using EC
0b000000.
See ISS encoding for an exception from an access to SVE functionality, resulting from
CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ.

EC == 0b011011
When FEAT_TME is implemented:
Exception from a TSTART instruction at EL0 when SCTLR_EL1.TME0
== 0, EL0 when SCTLR_EL2.TME0 == 0, at EL1 when SCTLR_EL1.TME == 0, at
EL2 when SCTLR_EL2.TME == 0 or at EL3 when SCTLR_EL3.TME == 0.
See ISS encoding for an exception from a TSTART instruction.

EC == 0b011100
When FEAT_FPAC is implemented:
Exception from a Pointer Authentication instruction authentication failure
See ISS encoding for an exception from a Pointer Authentication instruction
authentication failure.

EC == 0b011101
When FEAT_SME is implemented:
Access to SME functionality trapped as a result of CPACR_EL1.SMEN,
CPTR_EL2.SMEN, CPTR_EL2.TSM, CPTR_EL3.ESM, or an attempted execution of
an instruction that is illegal because of the value of PSTATE.SM or PSTATE.ZA, that is
not reported using EC 0b000000.
See ISS encoding for an exception due to SME functionality.

EC == 0b011110
When FEAT_RME is implemented:
Exception from a Granule Protection Check
See ISS encoding for an exception from a Granule Protection Check.

EC == 0b100000
Instruction Abort from a lower Exception level.
Used for MMU faults generated by instruction accesses and synchronous External
aborts, including synchronous parity or ECC errors. Not used for debug-related
exceptions.
See ISS encoding for an exception from an Instruction Abort.

**EC == 0b100001**

Instruction Abort taken without a change in Exception level.  
Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.  
See ISS encoding for an exception from an Instruction Abort.

**EC == 0b100010**

PC alignment fault exception.  
See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.

**EC == 0b100100**

Data Abort exception from a lower Exception level.  
Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.  
See ISS encoding for an exception from a Data Abort.

**EC == 0b100101**

Data Abort exception taken without a change in Exception level.  
Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.  
See ISS encoding for an exception from a Data Abort.

**EC == 0b100110**

SP alignment fault exception.  
See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.

**EC == 0b100111**

*When FEAT_MOPS is implemented:*  
Memory Operation Exception.  
See ISS encoding for an exception from the Memory Copy and Memory Set instructions.

**EC == 0b101000**

*When AArch32 is supported:*  
Trapped floating-point exception taken from AArch32 state.  
This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.  
See ISS encoding for an exception from a trapped floating-point exception.

**EC == 0b101100**

*When AArch64 is supported:*  
Trapped floating-point exception taken from AArch64 state.  
This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.  
See ISS encoding for an exception from a trapped floating-point exception.

**EC == 0b101111**

SError interrupt.  
See ISS encoding for an SError interrupt.

**EC == 0b110000**

Breakpoint exception from a lower Exception level.
See ISS encoding for an exception from a Breakpoint or Vector Catch debug exception.

EC == 0b1100001
Breakpoint exception taken without a change in Exception level.
See ISS encoding for an exception from a Breakpoint or Vector Catch debug exception.

EC == 0b110010
Software Step exception from a lower Exception level.
See ISS encoding for an exception from a Software Step exception.

EC == 0b110011
Software Step exception taken without a change in Exception level.
See ISS encoding for an exception from a Software Step exception.

EC == 0b110100
Watchpoint exception from a lower Exception level.
See ISS encoding for an exception from a Watchpoint exception.

EC == 0b110101
Watchpoint exception taken without a change in Exception level.
See ISS encoding for an exception from a Watchpoint exception.

EC == 0b111000
When AArch32 is supported:
BKPT instruction execution in AArch32 state.
See ISS encoding for an exception from execution of a Breakpoint instruction.

EC == 0b111000
When AArch64 is supported:
BRK instruction execution in AArch64 state.
See ISS encoding for an exception from execution of a Breakpoint instruction.

All other EC values are reserved by Arm, and:
- Unused values in the range 0b000000 - 0b101100 (0x00 - 0x2C) are reserved for future use for synchronous exceptions.
- Unused values in the range 0b101101 - 0b111111 (0x2D - 0x3F) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [25]
Instruction Length for synchronous exceptions. Possible values of this bit are:

0b0  16-bit instruction trapped.
0b1  32-bit instruction trapped. This value is also used when the exception is one of the following:
    - An SError interrupt.
    - An Instruction Abort exception.
    - A PC alignment fault exception.
    - An SP alignment fault exception.
    - A Data Abort exception for which the value of the ISV bit is 0.
    - An Illegal Execution state exception.
    - Any debug exception except for Breakpoint instruction exceptions. For Breakpoint instruction exceptions, this bit has its standard meaning:
      — 0b0: 16-bit T32 BKPT instruction.
— 0b1: 32-bit A32 BKPT instruction or A64 BRK instruction.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS, bits [24:0]

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

Typically, an ISS encoding has a number of subfields. When an ISS subfield holds a register number, the value returned in that field is the AArch64 view of the register number.

For an exception taken from AArch32 state, see Mapping of the general-purpose registers between the Execution states on page D1-4684.

If the AArch32 register descriptor is 0b1111, then:

• If the instruction that generated the exception was not UNPREDICTABLE, the field takes the value 0b11111.

• If the instruction that generated the exception was UNPREDICTABLE, the field takes an UNKNOWN value that must be either:
  — The AArch64 view of the register number of a register that might have been used at the Exception level from which the exception was taken.
  — The value 0b11111.

ISS encoding for exceptions with an unknown reason

<table>
<thead>
<tr>
<th>Bits [24:0]</th>
<th>Reserved, RES0</th>
</tr>
</thead>
</table>

Reserved, RES0.

When an exception is reported using this EC code the IL field is set to 1.

This EC code is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

• The attempted execution of an instruction bit pattern that has no allocated instruction or that is not accessible at the current Exception level and Security state, including:
  — A read access using a System register pattern that is not allocated for reads or that does not permit reads at the current Exception level and Security state.
  — A write access using a System register pattern that is not allocated for writes or that does not permit writes at the current Exception level and Security state.
  — Instruction encodings that are unallocated.
  — Instruction encodings for instructions or System registers that are not implemented in the implementation.

• In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.

• In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.

• In AArch32 state, attempted execution of a short vector floating-point instruction.
• In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.

• An exception generated because of the value of one of the SCTLR_EL1.{ITD, SED, CP15BEN} control bits.

• Attempted execution of:
  — An HVC instruction when disabled by HCR_EL2.HCD or SCR_EL3.HCE.
  — An SMC instruction when disabled by SCR_EL3.SMD.
  — An HLT instruction when disabled by EDSCR.HDE.

• Attempted execution of an MSR or MRS instruction to access SP_EL0 when the value of SPSel.SP is 0.

• Attempted execution of an MSR or MRS instruction using a _EL12 register name when HCR_EL2.E2H == 0.

• Attempted execution, in Debug state, of:
  — A DCPS1 instruction when the value of HCR_EL2.TGE is 1 and EL2 is disabled or not implemented in the current Security state.
  — A DCPS2 instruction from EL1 or EL0 when EL2 is disabled or not implemented in the current Security state.
  — A DCPS3 instruction when the value of EDSCR.SDD is 1, or when EL3 is not implemented.

• When EL3 is using AArch64, attempted execution from Secure EL1 of an SRS instruction using R13_mon.

• In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, or EL0 of an instruction that is configured to trap to EL3.

• In AArch32 state, the attempted execution of an MRS (banked register) or an MSR (banked register) instruction to SPSR_mon, SP_mon, or LR_mon.

• An exception that is taken to EL2 because the value of HCR_EL2.TGE is 1 that, if the value of HCR_EL2.TGE was 0 would have been reported with an ESR_ELx.EC value of 0b000111.

• In Non-transactional state, attempted execution of a TCOMMIT instruction.

**ISS encoding for an exception from a WF* instruction**

<table>
<thead>
<tr>
<th>CV</th>
<th>COND</th>
<th>RES0</th>
<th>RN</th>
<th>RES0</th>
<th>RV</th>
<th>TI</th>
</tr>
</thead>
</table>

CV, bit [24]

Condition code valid.

0b0  The COND field is not valid.

0b1  The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

• When an A32 instruction is trapped, CV is set to 1.

• When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.
COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:10]

Reserved, RES0.

RN, bits [9:5]

When FEAT_WFxT is implemented:

Register Number. Indicates the register number supplied for a WFET or WFIT instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [4:3]

Reserved, RES0.

RV, bit [2]

When FEAT_WFxT is implemented:

Register field Valid.

If TI[1] == 1, then this field indicates whether RN holds a valid register number for the register argument to the trapped WFET or WFIT instruction.

- 0b0 Register field invalid.
- 0b1 Register field valid.

If TI[1] == 0, then this field is RES0.

This field is set to 1 on a trap on WFET or WFIT.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**TI, bits [1:0]**
Trapped instruction. Possible values of this bit are:
- 0b00 WFI trapped.
- 0b01 WFE trapped.
- 0b10 *When FEAT_WFxT is implemented:* WFIT trapped.
- 0b11 *When FEAT_WFxT is implemented:* WFET trapped.

When FEAT_WFxT is implemented, this is a two bit field as shown. Otherwise, bit[1] is RES0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating this exception:
- SCTLR_EL1.{nTWE, nTWI}.
- HCR_EL2.{TWE, TWI}.
- SCR_EL3.{TWE, TWI}.

**ISS encoding for an exception from an MCR or MRC access**

```
<table>
<thead>
<tr>
<th>18</th>
<th>23</th>
<th>20</th>
<th>19</th>
<th>17</th>
<th>16</th>
<th>14</th>
<th>13</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV</td>
<td>COND</td>
<td>Opc2</td>
<td>Opc1</td>
<td>CRn</td>
<td>Rt</td>
<td>CRm</td>
<td>Direction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**CV, bit [24]**
Condition code valid.
- 0b0 The COND field is not valid.
- 0b1 The COND field is valid.
For exceptions taken from AArch64, CV is set to 1.
For exceptions taken from AArch32:
- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**COND, bits [23:20]**
For exceptions taken from AArch64, this field is set to 0b1110.
The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.
For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to \texttt{0b1110}.

- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to \texttt{0b1110}, the value for unconditional.
  - With the COND value held in the instruction.

- When a T32 instruction is trapped, it is \textbf{IMPLEMENTATION DEFINED} whether:
  - CV is set to 0 and COND is set to an \textbf{UNKNOWN} value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is \textbf{IMPLEMENTATION DEFINED} whether the COND field is set to \texttt{0b1110}, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally \textbf{UNKNOWN} value.

\textbf{Opc2, bits [19:17]}

The Opc2 value from the issued instruction.

For a trapped VMRS access, holds the value \texttt{0b000}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally \textbf{UNKNOWN} value.

\textbf{Opc1, bits [16:14]}

The Opc1 value from the issued instruction.

For a trapped VMRS access, holds the value \texttt{0b111}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally \textbf{UNKNOWN} value.

\textbf{CRn, bits [13:10]}

The CRn value from the issued instruction.

For a trapped VMRS access, holds the reg field from the VMRS instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally \textbf{UNKNOWN} value.

\textbf{Rt, bits [9:5]}

The Rt value from the issued instruction, the general-purpose register used for the transfer.

If the Rt value is not \texttt{0b1111}, then the reported value gives the AArch64 view of the register.

Otherwise, if the Rt value is \texttt{0b1111}:

- If the instruction that generated the exception is not \textbf{UNPREDICTABLE}, then the register specifier takes the value \texttt{0b11111}.

- If the instruction that generated the exception is \textbf{UNPREDICTABLE}, then the register specifier takes an \textbf{UNKNOWN} value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value \texttt{0b11111}.
See *Mapping of the general-purpose registers between the Execution states* on page D1-4684.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**CRm, bits [4:1]**
The CRm value from the issued instruction.
For a trapped VMRS access, holds the value **0b0000**.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Direction, bit [0]**
Indicates the direction of the trapped instruction.
- **0b0** Write to System register space. MCR instruction.
- **0b1** Read from System register space. MRC or VMRS instruction.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

The following fields describe configuration settings for generating exceptions that are reported using EC value **0b000011**:
- If `FEAT_TIDCP1` is implemented, `SCTLR_EL1.TIDCP`, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch32 state, MCR or MRC access (coproc == **0b1111**), trapped to EL1.
- `CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}`, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCR or MRC access (coproc == **0b1111**), trapped to EL1 or EL2.
- `PMUSERENR_EL0.{ER, CR, SW, EN}`, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCR or MRC access (coproc == **0b1111**), trapped to EL1 or EL2.
- `AMUSERENR_EL0.EN`, for accesses to Activity Monitors registers from EL0 using AArch32 state, MCR or MRC access (coproc == **0b1111**), trapped to EL1 or EL2.
- `HCR_EL12.TTLB`, for execution of TLB maintenance instructions at EL1 using AArch32 state, MCR or MRC access (coproc == **0b1111**), trapped to EL2.
- `HCR_EL2.TSVT, TPC, TPU` for execution of cache maintenance instructions at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == **0b1111**), trapped to EL2.
- `HCR_EL2.TACR`, for accesses to the Auxiliary Control Register at EL1 using AArch32 state, MCR or MRC access (coproc == **0b1111**), trapped to EL2.
- `HCR_EL2.TIDCP`, for accesses to lockdown, DMA, and TCM operations at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == **0b1111**), trapped to EL2.
- If `FEAT_TIDCP1` is implemented, `SCTLR_EL2.TIDCP`, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch32 state, MCR or MRC access (coproc == **0b1111**), trapped to EL2.
- `HCR_EL22.TID1, TID2, TID3`, for accesses to ID registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == **0b1111**), trapped to EL2.
- `CPTR_EL2.TCPAC`, for accesses to CPACR_EL1 or CPACR using AArch32 state, MCR or MRC access (coproc == **0b1111**), trapped to EL2.
- `HSTR_EL2.T<n>`, for accesses to System registers using AArch32 state, MCR or MRC access (coproc == **0b1111**), trapped to EL2.
• **CNTHCTL_EL2.EL1PCEN**, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

• **MDCR_EL2.[TPM, TPMCR]**, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

• **CPTR_EL2.TAM**, for accesses to Activity Monitors registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

• **CPTR_EL3.TCPAC**, for accesses to CPACR from EL1 and EL2, and accesses to HCPTP from EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL3.

• **CPTR_EL3.TAM**, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL3.

• If **FEAT_FGT** is implemented, MCR or MRC access to some registers at EL0, trapped to EL2.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b000101:

• **CPACR_EL1.TTA** for accesses to trace registers, MCR or MRC access (coproc == 0b1110) trapped to EL1 or EL2.

• **MDSCR_EL1.TDCC**, for accesses to the Debug Communications Channel (DCC) registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL1 or EL2.

• If **FEAT_FGT** is implemented, **MDCR_EL2.TDCC** for accesses to the DCC registers at EL0 and EL1 trapped to EL2, and **MDCR_EL3.TDCC** for accesses to the DCC registers at EL0, EL1, and EL2 trapped to EL3.

• **HCR_EL2.TID0**, for accesses to the JIDR register in the ID group 0 at EL0 and EL1 using AArch32, MRC access (coproc == 0b1110) trapped to EL2.

• **CPTR_EL2.TTA**, for accesses to trace registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL2.

• **MDCR_EL2.TDRA**, for accesses to Debug ROM registers DBGDRAR and DBGDSAR using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL2.

• **MDCR_EL2.TDOSA**, for accesses to powerdown debug registers, using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL2.

• **MDCR_EL2.TDA**, for accesses to other debug registers, using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL2.

• **CPTR_EL3.TTA**, for accesses to trace registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.

• **MDCR_EL3.TDOSA**, for accesses to powerdown debug registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.

• **MDCR_EL3.TDA**, for accesses to other debug registers, using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b001000:

• **HCR_EL2.TID0**, for accesses to the FPSID register in ID group 0 at EL1 using AArch32 state, VMRS access trapped to EL2.

• **HCR_EL2.TID3**, for accesses to registers in ID group 3 including MVFR0, MVFR1 and MVFR2, VMRS access trapped to EL2.
**ISS encoding for an exception from an LD64B or ST64B* instruction**

ISS, bits [24:0]

<table>
<thead>
<tr>
<th>ISS encoding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000000000000000000000000</td>
<td>When FEAT_LS64_V is implemented: ST64BV instruction trapped.</td>
</tr>
<tr>
<td>0b0000000000000000000000001</td>
<td>When FEAT_LS64_ACCDATA is implemented: ST64BV0 instruction trapped.</td>
</tr>
<tr>
<td>0b0000000000000000000000010</td>
<td>When FEAT_LS64 is implemented: LD64B or ST64B instruction trapped.</td>
</tr>
</tbody>
</table>

All other values are reserved.

**ISS encoding for an exception from an MCRR or MRRC access**

CV, bit [24]

Condition code valid.

- 0b0 The COND field is not valid.
- 0b1 The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.
For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc1, bits [19:16]
The Opc1 value from the issued instruction.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [15]
Reserved, RES0.

Rt2, bits [14:10]
The Rt2 value from the issued instruction, the second general-purpose register used for the transfer.
If the Rt2 value is not 0b1111, then the reported value gives the AArch64 view of the register. Otherwise, if the Rt2 value is 0b1111:
- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value 0b1111.
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value 0b1111.

See Mapping of the general-purpose registers between the Execution states on page D1-4684.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]
The Rt value from the issued instruction, the first general-purpose register used for the transfer.
If the Rt value is not 0b1111, then the reported value gives the AArch64 view of the register. Otherwise, if the Rt value is 0b1111:
- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value 0b1111.
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value 0b1111.

See Mapping of the general-purpose registers between the Execution states on page D1-4684.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
CRm, bits [4:1]
The CRm value from the issued instruction.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]
Indicates the direction of the trapped instruction.
0b0 Write to System register space. MCRR instruction.
0b1 Read from System register space. MRRC instruction.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b000100:

• CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.

• PMUSERENR_EL0.{CR, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.

• AMUSERENR_EL0.{EN}, for accesses to Activity Monitors registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.

• HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.

• HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.

• CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.

• MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.

• CPTR_EL2.TAM, for accesses to Activity Monitors registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.

• MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL3.

• CPTR_EL3.TAM, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL3.

• IF FEAT_FGT is implemented, HDFGRTR_EL2.PMCCNTR_EL0 for MRRC access and HDFGWTR_EL2.PMCCNTR_EL0 for MCRR access to PMCCNTR at EL0, trapped to EL2.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b001100:

• MDSCR_EL1.TDCC, for accesses to the Debug ROM registers DBGDSAR and DBGDRAR at EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1110) trapped to EL1 or EL2.

• MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and DBGDSAR using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL2.

• MDCR_EL3.TDA, for accesses to debug registers, using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL3.
• **CPACR_EL1.TTA** for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL1 or EL2.

• **CPTR_EL2.TTA**, for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL2.

• **CPTR_EL3.TTA**, for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL3.

--- **Note**

If the Armv8-A architecture is implemented with an ETMv4 implementation, MCRR and MRRC accesses to trace registers are UNDEFINED and the resulting exception is higher priority than an exception due to these traps.

### ISS encoding for an exception from an LDC or STC instruction

<table>
<thead>
<tr>
<th></th>
<th>CV</th>
<th>COND</th>
<th>imm8</th>
<th>RES0</th>
<th>Rn</th>
<th>AM</th>
<th>Offset</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>24</td>
<td>23</td>
<td>20-19</td>
<td>12-11</td>
<td>10-9</td>
<td>8-3</td>
<td>1-0</td>
<td></td>
</tr>
</tbody>
</table>

**CV, bit [24]**

Condition code valid.
- 0b0  The COND field is not valid.
- 0b1  The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:
- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**COND, bits [23:20]**

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:
- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IR field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
• For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to \texttt{0b1110}, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

\textbf{imm8, bits \([19:12]\)}

The immediate value from the issued instruction.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

\textbf{Bits \([11:10]\)}

Reserved, \texttt{RES0}.

\textbf{Rn, bits \([9:5]\)}

The Rn value from the issued instruction, the general-purpose register used for the transfer.

If the Rn value is not \texttt{0b1111}, then the reported value gives the AArch64 view of the register.

Otherwise, if the Rn value is \texttt{0b1111}:
• If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value \texttt{0b1111}.
• If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  — The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  — The value \texttt{0b1111}.

See \textit{Mapping of the general-purpose registers between the Execution states on page D1-4684.}

This field is valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction. When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

\textbf{Offset, bit \([4]\)}

Indicates whether the offset is added or subtracted:
\begin{itemize}
  \item \texttt{0b0} Subtract offset.
  \item \texttt{0b1} Add offset.
\end{itemize}

This bit corresponds to the U bit in the instruction encoding.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

\textbf{AM, bits \([3:1]\)}

Addressing mode. The permitted values of this field are:
\begin{itemize}
  \item \texttt{0b000} Immediate unindexed.
  \item \texttt{0b001} Immediate post-indexed.
  \item \texttt{0b010} Immediate offset.
  \item \texttt{0b011} Immediate pre-indexed.
  \item \texttt{0b100} For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.
  \item \texttt{0b110} For a trapped STC instruction, this encoding is reserved.
\end{itemize}
The values 0b101 and 0b111 are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-11600.

Bit [2] in this subfield indicates the instruction form, immediate or literal.

Bits [1:0] in this subfield correspond to the bits \{P, W\} in the instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Direction, bit [0]**

Indicates the direction of the trapped instruction.

0b0 Write to memory. STC instruction.

0b1 Read from memory. LDC instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

The following fields describe the configuration settings for the traps that are reported using EC value 0b000110:

- **MDSCR_EL1.TDCC**, for accesses using AArch32 state, LDC access to DBGDTRXint or STC access to DBGDTRXint trapped to EL1 or EL2.

- **MDCR_EL2.TDA**, for accesses using AArch32 state, LDC access to DBGDTRXint or STC access to DBGDTRXint MCR or MRC access trapped to EL2.

- **MDCR_EL3.TDA**, for accesses using AArch32 state, LDC access to DBGDTRXint or STC access to DBGDTRXint MCR or MRC access trapped to EL3.

- If FEAT_FGT is implemented, **MDCR_EL2.TDCC** for LDC and STC accesses to the DCC registers at EL0 and EL1 trapped to EL2, and **MDCR_EL3.TDCC** for accesses to the DCC registers at EL0, EL1, and EL2 trapped to EL3.

**ISS encoding for an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from the FPEN and TFP traps**

```
   28  23  20  19  0
   CV  COND RES0
```

The accesses covered by this trap include:

- Execution of SVE or Advanced SIMD and floating-point instructions.

- Accesses to the Advanced SIMD and floating-point System registers.

- Execution of SME instructions.

For an implementation that does not include either SVE or support for Advanced SIMD and floating-point, the exception is reported using the EC value 0b000000.

**CV, bit [24]**

Condition code valid.

0b0 The COND field is not valid.

0b1 The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
• When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.
The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:
• When an A32 instruction is trapped, CV is set to 1 and:
  — If the instruction is conditional, COND is set to the condition code field value from the instruction.
  — If the instruction is unconditional, COND is set to 0b1110.
• A conditional A32 instruction that is known to pass its condition code check can be presented either:
  — With COND set to 0b1110, the value for unconditional.
  — With the COND value held in the instruction.
• When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  — CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  — CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
• For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:0]
Reserved, RES0.

The following fields describe the configuration settings for the traps that are reported using EC value 0b000111:
• CPACR_EL1.FPEN, for accesses to SIMD and floating-point registers trapped to EL1.
• CPTR_EL2.FPEN and CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL2.
• CPTR_EL3.TFP, for accesses to SIMD and floating-point registers trapped to EL3.

ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ

The accesses covered by this trap include:
• Execution of SVE instructions when the PE is not in Streaming SVE mode.
• Accesses to the SVE System registers, ZCR_ELx.
For an implementation that does not include SVE, the exception is reported using the EC value 0b000000.

**Bits [24:0]**

Reserved, RES0.

The following fields describe the configuration settings for the traps that are reported using EC value 0b011001:

- **CPACR_EL1.ZEN**, for execution of SVE instructions and accesses to SVE registers at EL0 or EL1, trapped to EL1.
- **CPTR_EL2.ZEN and CPTR_EL2.TZ**, for execution of SVE instructions and accesses to SVE registers at EL0, EL1, or EL2, trapped to EL2.
- **CPTR_EL3.EZ**, for execution of SVE instructions and accesses to SVE registers from all Exception levels, trapped to EL3.

**ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault**

![ISS encoding diagram]

**Bits [24:0]**

Reserved, RES0.

There are no configuration settings for generating Illegal Execution state exceptions and PC alignment fault exceptions. For more information about PC alignment fault exceptions, see [PC alignment checking](#) on page D1-4668.

**SP alignment checking** on page D1-4668 describes the configuration settings for generating SP alignment fault exceptions.

**ISS encoding for an exception from the Memory Copy and Memory Set instructions**

![ISS encoding diagram]

**MemInst, bit [24]**

Indicates the memory instruction class causing the exception.

- 0b0: CPYFE*, CPYFM*, CPYE*, and CPYM* instructions.
- 0b1: SETE*, SETM*, SETGE*, and SETGM* instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**isSETG, bit [23]**

Indicates whether the instruction belongs to SETGM* or SETGE* class of instruction.

- 0b0: Not a SETGM* or SETGE* instruction.
- 0b1: SETGM* or SETGE* instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Options, bits [22:19]
Options : the Options field of the instruction.
For Memory Copy instructions, bits[22:19] forms the Options field, which holds the bits[15:12] of the instruction.
For Memory Set instructions:
- Bits[22:21] are RES0.
- Bits[20:19] form the Options field, which holds the bits[13:12] of the instruction.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FromEpilogue, bit [18]
Indicates whether the instruction belongs to the epilogue class of Memory Copy or Memory Set instructions.
0b0 Not an epilogue instruction.
0b1 CPYE*, CPYFE*, SETE*, or SETGE* instruction.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WrongOption, bit [17]
Algorithm option.
0b0 WrongOption is false.
0b1 WrongOption is true.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OptionA, bit [16]
Algorithm type indicated by the PSTATE.C bit.
0b0 OptionB indicated by PSTATE.C is 0.
0b1 OptionA indicated by PSTATE.C is 1.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [15]
Reserved, RES0.

destreg, bits [14:10]
The destination register value from the issued instruction, containing the destination address.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

srcreg, bits [9:5]
The source register value from the issued instruction, containing either the source address or the source data.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

sizereg, bits [4:0]
The size register value from the issued instruction, containing the number of bytes to be transfered or set.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ISS encoding for an exception from HVC or SVC instruction execution**

<table>
<thead>
<tr>
<th>Bits [24:16]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**imm16, bits [15:0]**

The value of the immediate field from the HVC or SVC instruction.

For an HVC instruction, and for an A64 SVC instruction, this is the value of the imm16 field of the issued instruction.

For an A32 or T32 SVC instruction:
- If the instruction is unconditional, then:
  - For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.
  - For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- If the instruction is conditional, this field is UNKNOWN.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

In AArch32 state, the HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

For T32 and A32 instructions, see SVC and HVC.

For A64 instructions, see SVC and HVC.

If FEAT_FGT is implemented, HFGITR_EL2.[SVC_EL1, SVC_EL0] control fine-grained traps on SVC execution.

**ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state**

<table>
<thead>
<tr>
<th>Bits [24:22]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**Op0, bits [21:20]**

The Op0 value from the issued instruction.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Op2, bits [19:17]**

The Op2 value from the issued instruction.
The reset behavior of this field is:
   • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Op1, bits [16:14]**

The Op1 value from the issued instruction.
The reset behavior of this field is:
   • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**CRn, bits [13:10]**

The CRn value from the issued instruction.
The reset behavior of this field is:
   • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Rt, bits [9:5]**

The Rt value from the issued instruction, the general-purpose register used for the transfer.
The reset behavior of this field is:
   • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**CRm, bits [4:1]**

The CRm value from the issued instruction.
The reset behavior of this field is:
   • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Direction, bit [0]**

Indicates the direction of the trapped instruction.
0b0   Write access, including MSR instructions.
0b1   Read access, including MRS instructions.
The reset behavior of this field is:
   • On a Warm reset, this field resets to an architecturally UNKNOWN value.

For exceptions caused by System instructions, see *System instructions on page C4-539* for the encoding values returned by an instruction.

The following fields describe configuration settings for generating the exception that is reported using EC value 0b011000:

- If **FEAT_TIDCP1** is implemented, **SCTLR_EL1.TIDCP**, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch64 state, MSR or MRS access trapped to EL1.
- **SCTLR_EL1.UCI**, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- **SCTLR_EL1.UCT**, for accesses to **CTR_EL0** using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- **SCTLR_EL1.DZE**, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- **SCTLR_EL1.UMA**, for accesses to the **PSTATE** interrupt masks using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- **CPACR_EL1.TTA**, for accesses to the trace registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- **MDSCR_EL1.TDCC**, for accesses to the Debug Communications Channel (DCC) registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
• If FEAT_FGT is implemented, MDCR_EL2.TDCC for accesses to the DCC registers at EL0 and EL1 trapped to EL2, and MDCR_EL3.TDCC for accesses to the DCC registers at EL0, EL1, and EL2 trapped to EL3.

• CNTKCTL_EL1.\{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN\} accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.

• PMUSERENR_EL0.\{ER, CR, SW, EN\}, for accesses to the Performance Monitor registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.

• AMUSERENR_EL0.EN, for accesses to Activity Monitors registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.

• HCR_EL2.\{TRVM, TVM\}, for accesses to virtual memory control registers using AArch64 state, MSR or MRS access trapped to EL2.

• HCR_EL2.TDZ, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL2.

• HCR_EL2.TTLB, for execution of TLB maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.

• HCR_EL2.\{TSW, TPC, TPU\}, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.

• HCR_EL2.TACR, for accesses to the Auxiliary Control Register, ACTLR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.

• HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations using AArch64 state, MSR or MRS access trapped to EL2.

• If FEAT_TIDCP1 is implemented, SCTLR_EL2.TIDCP, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch64 state, MSR or MRS access trapped to EL2.

• HCR_EL2.\{TID1, TID2, TID3\}, for accesses to ID group 1, ID group 2 or ID group 3 registers, using AArch64 state, MSR or MRS access trapped to EL2.

• CPTR_EL2.TCPAC, for accesses to CPACR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.

• CPTR_EL2.TTA, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL2.

• MDCR_EL2.TTRF, for accesses to the trace filter control register, TRFCR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.

• MDCR_EL2.TDRA, for accesses to Debug ROM registers, using AArch64 state, MSR or MRS access trapped to EL2.

• MDCR_EL2.TDOSA, for accesses to powerdown debug registers using AArch64 state, MSR or MRS access trapped to EL2.

• CNTHTCTL_EL2.\{EL1PCEN, EL1PCTEN\}, for accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2.

• MDCR_EL2.TDA, for accesses to debug registers using AArch64 state, MSR or MRS access trapped to EL2.

• MDCR_EL2.\{TPM, TPMCR\}, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL2.

• CPTR_EL2.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access trapped to EL2.

• HCR_EL2.APK, for accesses to Pointer authentication key registers. using AArch64 state, MSR or MRS access trapped to EL2.
• `HCR_EL2.{NV, NV1}`, for Nested virtualization register access, using AArch64 state, MSR or MRS access, trapped to EL2.

• `HCR_EL2.AT`, for execution of AT S1E* instructions, using AArch64 state, MSR or MRS access, trapped to EL2.

• `HCR_EL2.{TERR, FIEN}`, for accesses to RAS registers, using AArch64 state, MSR or MRS access, trapped to EL2.

• `SCR_EL3.APK`, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL3.

• `SCR_EL3.ST`, for accesses to the Counter-timer Physical Secure timer registers, using AArch64 state, MSR or MRS access trapped to EL3.

• `SCR_EL3.{TERR, FIEN}`, for accesses to RAS registers, using AArch64 state, MSR or MRS access trapped to EL3.

• `CPTR_EL3.TTCPAC`, for accesses to `CPTR_EL2` and `CPACR_EL1` using AArch64 state, MSR or MRS access trapped to EL3.

• `CPTR_EL3.TTA`, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL3.

• `MDCR_EL3.TTRF`, for accesses to the trace filter control registers, `TRFCR_EL1` and `TRFCR_EL2`, using AArch64 state, MSR or MRS access trapped to EL3.

• `MDCR_EL3.TDA`, for accesses to debug registers, using AArch64 state, MSR or MRS access trapped to EL3.

• `MDCR_EL3.TDOSA`, for accesses to powerdown debug registers, using AArch64 state, MSR or MRS access trapped to EL3.

• `MDCR_EL3.TPM`, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL3.

• `CPTR_EL3.TAM`, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access, trapped to EL3.

• If `FEAT_EVT` is implemented, the following registers control traps for EL1 and EL0 Cache controls that use this EC value:
  — `HCR_EL2.{TTLBOS, TTLBIS, TICAB, TOCU, TID4}`.
  — `HCR2.{TTLBIS, TICAB, TOCU, TID4}`.

• If `FEAT_FGT` is implemented:
  — `SCR_EL3.FGTEn`, for accesses to the fine-grained trap registers, MSR or MRS access at EL2 trapped to EL3.
  — `HFGRTR_EL2` for reads and `HFGWTR_EL2` for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 trapped to EL2.
  — `HFGITR_EL2` for execution of system instructions, MSR or MRS access trapped to EL2
  — `HDFGRTR_EL2` for reads and `HDFGWTR_EL2` for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 state trapped to EL2.
  — `HAFGRTR_EL2` for reads of Activity Monitor counters, using AArch64 state, MRS access at EL0 and EL1 trapped to EL2.

• If `FEAT_RNG_TRAP` is implemented:
  — `SCR_EL3.TRNDR` for reads of `RNDR` and `RNDRRS` using AArch64 state, MRS access trapped to EL3.

• If `FEAT_SME` is implemented:
  — `CPTR_EL3.ESM`, for MSR or MRS accesses to `SMPRI_EL1` at EL1, EL2, and EL3, trapped to EL3.
— CPTR_EL3.ESM, for MSR or MRS accesses to SMPRIMAP_EL2 at EL2 and EL3, trapped to EL3.
— SCTLR_EL1.EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, trapped to EL1 or EL2.
— SCTLR_EL2.EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, trapped to EL2.
— SCR_EL3.EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, EL1, and EL2, trapped to EL3.

• If FEAT_NMI is implemented, HCRX_EL2.TALLINT, for MSR writes of ALLINT at EL1, trapped to EL2.

**ISS encoding for an exception from an Instruction Abort**

![ISS encoding diagram]

**Bits [24:13]**

Reserved, RES0.

**SET, bits [12:11]**

*When FEAT_RAS is implemented:*

Synchronous Error Type. When IFSC is 0b010000, describes the PE error state after taking the
Instruction Abort exception.

0b00 Recoverable state (UER).
0b10 Uncontainable (UC).
0b11 Restartable state (UEO).

All other values are reserved.

--- **Note** ---

Software can use this information to determine what recovery might be possible. Taking a
synchronous External Abort exception might result in a PE state that is not recoverable.

---

This field is valid only if the IFSC code is 0b010000. It is RES0 for all other aborts.

The reset behavior of this field is:

* • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**FnV, bit [10]**

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a
translation table walk.

0b0 FAR is valid.
0b1 FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the IFSC code is 0b010000. It is RES0 for all other aborts.

The reset behavior of this field is:

* • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**EA, bit [9]**

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External
aborts.

For any abort other than an External abort this bit returns a value of 0.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [8]**

Reserved, RES0.

**S1PTW, bit [7]**

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

- 0b0 Fault not on a stage 2 translation for a stage 1 translation table walk.
- 0b1 Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [6]**

Reserved, RES0.

**IFSC, bits [5:0]**

Instruction Fault Status Code.

- 0b000000 Address size fault, level 0 of translation or translation table base register.
- 0b000001 Address size fault, level 1.
- 0b000010 Address size fault, level 2.
- 0b000011 Address size fault, level 3.
- 0b000100 Translation fault, level 0.
- 0b000101 Translation fault, level 1.
- 0b000110 Translation fault, level 2.
- 0b000111 Translation fault, level 3.
- 0b001001 Access flag fault, level 1.
- 0b001010 Access flag fault, level 2.
- 0b001011 Access flag fault, level 3.
- 0b001100 When FEAT_LP A2 is implemented:
  Access flag fault, level 0.
- 0b001101 When FEAT_LP A2 is implemented:
  Permission fault, level 0.
- 0b001110 Permission fault, level 1.
- 0b001111 Permission fault, level 2.
- 0b010000 Synchronous External abort, not on translation table walk or hardware update of translation table.
- 0b010011 When FEAT_LP A2 is implemented:
  Synchronous External abort on translation table walk or hardware update of translation table, level -1.
- 0b010100 Synchronous External abort on translation table walk or hardware update of translation table, level 0.
- 0b010101 Synchronous External abort on translation table walk or hardware update of translation table, level 1.
- 0b010110 Synchronous External abort on translation table walk or hardware update of translation table, level 2.
0b01011  Synchronous External abort on translation table walk or hardware update of translation table, level 3.

0b01100  When FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access, not on translation table walk.

0b01101  When FEAT_LPA2 is implemented and FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.

0b01110  When FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.

0b01101  When FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.

0b01110  When FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.

0b01111  When FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.

0b10001  When FEAT_RME is implemented and FEAT_LPA2 is implemented:
Granule Protection Fault on translation table walk or hardware update of translation table, level -1.

0b10010  When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation table, level 0.

0b10011  When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation table, level 1.

0b10010  When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation table, level 2.

0b10011  When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation table, level 3.

0b10100  When FEAT_RME is implemented:
Granule Protection Fault, not on translation table walk or hardware update of translation table.

0b10101  When FEAT_LPA2 is implemented:
Address size fault, level -1.

0b10111  When FEAT_LPA2 is implemented:
Translation fault, level -1.

0b11000  TLB conflict abort.

0b11001  When FEAT_HAFDBS is implemented:
Unsupported atomic hardware update fault.

All other values are reserved.

For more information about the lookup level associated with a fault, see The lookup level associated with MMU faults on page D8-5190.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ISS encoding for an exception due to SME functionality**

The accesses covered by this trap include:
- Execution of SME instructions.
- Execution of SVE and Advanced SIMD instructions, when the PE is in Streaming SVE mode.
- Direct accesses of SVCR, SMCR_EL1, SMCR_EL2, SMCR_EL3.

**Bits [24:3]**
- Reserved, RES0.

**SMTC, bits [2:0]**
- SME Trap Code. Identifies the reason for instruction trapping.
  - 0b000: Access to SME functionality trapped as a result of CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, or CPTR_EL3.ESM, that is not reported using EC 0b000000.
  - 0b001: Advanced SIMD, SVE, or SVE2 instruction trapped because PSTATE.SM is 1.
  - 0b010: SME instruction trapped because PSTATE.SM is 0.
  - 0b011: SME instruction trapped because PSTATE.ZA is 0.
- All other values are reserved.

The following fields describe the configuration settings for the traps that are reported using the EC value 0b011101:
- **CPACR_EL1.SMEN**, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR and SMCR_EL1 System registers at EL1 and EL0, trapped to EL1 or EL2.
- **CPTR_EL2.SMEN and CPTR_EL2.TSM**, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR, SMCR_EL1, SMCR_EL2 at EL2, EL1, or EL0, trapped to EL2.
- **CPTR_EL3.ESM**, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR, SMCR_EL1, SMCR_EL2, SMCR_EL3 from all Exception levels and any Security state, trapped to EL3.

**ISS encoding for an exception from a Granule Protection Check**

**Bits [24:22]**
- Reserved, RES0.
S2PTW, bit [21]
Indicates whether the Granule Protection Check exception was on an access made for a stage 2 translation table walk.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Fault not on a stage 2 translation table walk.</td>
</tr>
<tr>
<td>0b1</td>
<td>Fault on a stage 2 translation table walk.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

InD, bit [20]
Indicates whether the Granule Protection Check exception was on an instruction or data access.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Data access.</td>
</tr>
<tr>
<td>0b1</td>
<td>Instruction access.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

GPCSC, bits [19:14]
Granule Protection Check Status Code.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000000</td>
<td>GPT address size fault at level 0.</td>
</tr>
<tr>
<td>0b000100</td>
<td>GPT walk fault at level 0.</td>
</tr>
<tr>
<td>0b000101</td>
<td>GPT walk fault at level 1.</td>
</tr>
<tr>
<td>0b001100</td>
<td>Granule protection fault at level 0.</td>
</tr>
<tr>
<td>0b001101</td>
<td>Granule protection fault at level 1.</td>
</tr>
<tr>
<td>0b010100</td>
<td>Synchronous External abort on GPT fetch at level 0.</td>
</tr>
<tr>
<td>0b010101</td>
<td>Synchronous External abort on GPT fetch at level 1.</td>
</tr>
</tbody>
</table>

All other values are reserved.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

VNCR, bit [13]
*When FEAT_NV2 is implemented:*
Indicates that the fault came from use of **VNCR_EL2** register by EL1 code.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The fault was not generated by the use of <strong>VNCR_EL2</strong>, by an MRS or MSR instruction executed at EL1.</td>
</tr>
<tr>
<td>0b1</td>
<td>The fault was generated by the use of <strong>VNCR_EL2</strong>, by an MRS or MSR instruction executed at EL1.</td>
</tr>
</tbody>
</table>

This field is 0 in ESR_EL1.

When InD is '1', this field is **RES0**.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

*Otherwise:*
Reserved, **RES0**.

**Bits [12:9]**
Reserved, **RES0**.
CM, bit [8]

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

<table>
<thead>
<tr>
<th>CM Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.</td>
</tr>
<tr>
<td>0b1</td>
<td>The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA, DC GVA, and DC GZVA instructions are not classified as cache maintenance instructions, and therefore their execution cannot cause this field to be set to 1.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

Indicates whether the Granule Protection Check exception was on an access for stage 2 translation for a stage 1 translation table walk:

<table>
<thead>
<tr>
<th>S1PTW Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Fault not on a stage 2 translation for a stage 1 translation table walk.</td>
</tr>
<tr>
<td>0b1</td>
<td>Fault on the stage 2 translation of an access for a stage 1 translation table walk.</td>
</tr>
</tbody>
</table>

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

<table>
<thead>
<tr>
<th>WnR Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Abort caused by an instruction reading from a memory location.</td>
</tr>
<tr>
<td>0b1</td>
<td>Abort caused by an instruction writing to a memory location.</td>
</tr>
</tbody>
</table>

When InD is '1', this field is RES0.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

This field is UNKNOWN for:

- An External abort on an Atomic access.
- A fault reported using a DFSC value of 0b110101 or 0b110001, indicating an unsupported Exclusive or atomic access.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

xFSC, bits [5:0]

Instruction or Data Fault Status Code.

<table>
<thead>
<tr>
<th>xFSC Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b100011</td>
<td>When FEAT_RME is implemented and FEAT_LPA2 is implemented: Granule Protection Fault on translation table walk or hardware update of translation table, level -1.</td>
</tr>
<tr>
<td>0b100100</td>
<td>When FEAT_RME is implemented: Granule Protection Fault on translation table walk or hardware update of translation table, level 0.</td>
</tr>
</tbody>
</table>
When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation table, level 1.

When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation table, level 2.

When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation table, level 3.

When FEAT_RME is implemented:
Granule Protection Fault, not on translation table walk or hardware update of translation table.

All other values are reserved.

For more information about the lookup level associated with a fault, see *The lookup level associated with MMU faults on page D8-5190.*

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ISS encoding for an exception from a Data Abort**

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this ISS encoding includes ISS2, bits[36:32].

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this ISS encoding includes ISS2, bits[36:32].

**ISV, bit [24]**

Instruction Syndrome Valid. Indicates whether the syndrome information in ISS[23:14] is valid.

0b0 No valid instruction syndrome. ISS[23:14] are RES0.

0b1 ISS[23:14] hold a valid instruction syndrome.

In ESR_EL2, ISV is 1 when FEAT_LS64 is implemented and a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

In ESR_EL2, ISV is 1 when FEAT_LS64_V is implemented and a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

In ESR_EL2, ISV is 1 when FEAT_LS64_ACCDATA is implemented and a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.
For other faults reported in ESR_EL2, ISV is 0 except for the following stage 2 aborts:

- AArch64 loads and stores of a single general-purpose register (including the register specified with \texttt{0b11111}, including those with Acquire/Release semantics, but excluding Load Exclusive or Store Exclusive and excluding those with writeback).
- AArch32 instructions where the instruction:
  - Is an LDR, LDA, LDRT, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDBA, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLHB, or STRBT instruction.
  - Is not performing register writeback.
  - Is not using R15 as a source or destination register.

For these stage 2 aborts, ISV is \texttt{UNKNOWN} if the exception was generated in Debug state in memory access mode, and otherwise indicates whether ISS[23:14] hold a valid syndrome.

For faults reported in ESR_EL1 or ESR_EL3, ISV is 1 when FEAT_LS64 is implemented and a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

For faults reported in ESR_EL1 or ESR_EL3, ISV is 1 when FEAT_LS64_V is implemented and a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

For faults reported in ESR_EL1 or ESR_EL3, ISV is 1 when FEAT_LS64_ACCE is implemented and a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

When FEAT_RAS is implemented, ISV is 0 for any synchronous External abort.

When FEAT_RAS is not implemented, it is IMPLEMENTATION DEFINED whether ISV is set to 1 or 0 on a synchronous External abort on a stage 2 translation table walk.

For ISS reporting, a stage 2 abort on a stage 1 translation table walk does not return a valid instruction syndrome, and therefore ISV is 0 for these aborts.

When FEAT_MOPS is implemented, for a synchronous Data Abort on a Memory Copy and Memory Set instruction, ISV is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally \texttt{UNKNOWN} value.

\textbf{SAS, bits [23:22]}

\textit{When ISV == 1:}

\textbf{Syndrome Access Size. Indicates the size of the access attempted by the faulting operation.}

\begin{itemize}
\item \texttt{0b00}  \textbf{Byte}
\item \texttt{0b01}  \textbf{Halfword}
\item \texttt{0b10}  \textbf{Word}
\item \texttt{0b11}  \textbf{Doubleword}
\end{itemize}

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is \texttt{0b11}.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is \texttt{0b11}.

When FEAT_LS64_ACCE is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is \texttt{0b11}.

This field is \texttt{UNKNOWN} when the value of ISV is \texttt{UNKNOWN}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally \texttt{UNKNOWN} value.
SSE, bit [21]

**When ISV == 1:**
Syndrome Sign Extend. For a byte, halfword, or word load operation, indicates whether the data item must be sign extended.

- **0b0** Sign-extension not required.
- **0b1** Data item must be sign-extended.

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

For all other operations, this field is 0.

This field is **UNKNOWN** when the value of ISV is **UNKNOWN**.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**
Reserved, RES0.

SRT, bits [20:16]

**When ISV == 1:**
Syndrome Register Transfer. The register number of the Wt/Xt/Rt operand of the faulting instruction.

If the exception was taken from an Exception level that is using AArch32, then this is the AArch64 view of the register. See [Mapping of the general-purpose registers between the Execution states](#) on page D1-4684.

This field is **UNKNOWN** when the value of ISV is **UNKNOWN**.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**
Reserved, RES0.

SF, bit [15]

**When ISV == 1:**

**SF**
Sixty Four bit general-purpose register transfer. Width of the register accessed by the instruction is 64-bit.

- **0b0** Instruction loads/stores a 32-bit general-purpose register.
- **0b1** Instruction loads/stores a 64-bit general-purpose register.

--- **Note** ---
This field specifies the register width identified by the instruction, not the Execution state.
When `FEAT_LS64` is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 1.

When `FEAT_LS64_V` is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 1.

When `FEAT_LS64_ACCDATA` is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 1.

This field is `UNKNOWN` when the value of ISV is `UNKNOWN`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally `UNKNOWN` value.

**When `ISV == 0` and `FEAT_SME` is implemented:**

FnP

FAR not Precise.

0b0 The FAR holds the faulting virtual address that generated the Data Abort.

0b1 The FAR holds any virtual address within the naturally-aligned granule that contains the faulting virtual address that generated a Data Abort due to an SVE contiguous vector load/store instruction in Streaming SVE mode, or an SME load/store instruction.

For more information about the naturally-aligned fault granule, see FAR_ELx (for example, FAR_EL1).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally `UNKNOWN` value.

**Otherwise:**

Reserved, RES0.

**AR, bit [14]**

**When `ISV == 1`:**

Acquire/Release.

0b0 Instruction did not have acquire/release semantics.

0b1 Instruction did have acquire/release semantics.

When `FEAT_LS64` is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When `FEAT_LS64_V` is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When `FEAT_LS64_ACCDATA` is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

This field is `UNKNOWN` when the value of ISV is `UNKNOWN`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally `UNKNOWN` value.

**Otherwise:**

Reserved, RES0.
VNCR, bit [13]

When FEAT_NV2 is implemented:

Indicates that the fault came from use of VNCR_EL2 register by EL1 code.

0b0    The fault was not generated by the use of VNCR_EL2, by an MRS or MSR instruction executed at EL1.

0b1    The fault was generated by the use of VNCR_EL2, by an MRS or MSR instruction executed at EL1.

This field is 0 in ESR_EL1.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LST, bits [12:11]

When (DFSC == 0b00xxxx || DFSC == 0b101011) && DFSC != 0b0000xx:

LST
Load/Store Type. Used when a Translation fault, Access flag fault, or Permission fault generates a Data Abort.

0b00    The instruction that generated the Data Abort is not specified.

0b01    When FEAT_LS64_V is implemented:
An ST64BV instruction generated the Data Abort.

0b10    When FEAT_LS64 is implemented:
An LD64B or ST64B instruction generated the Data Abort.

0b11    When FEAT_LS64_ACCDATA is implemented:
An ST64BV0 instruction generated the Data Abort.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_RAS is implemented and DFSC == 0b010000:

SET
Synchronous Error Type. Used when a Synchronous External abort, not on a Translation table walk or hardware update of the Translation table, generated the Data Abort. Describes the PE error state after taking the Data Abort exception.

0b00    Recoverable state (UER).

0b10    Uncontainable (UC).

0b11    Restartable state (UEO).

All other values are reserved.

Note
Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in a PE state that is not recoverable.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

0b0  FAR is valid.
0b1  FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the DFSC code is 0b010000. It is RES0 for all other aborts.

The reset behavior of this field is:
•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:
•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

CM, bit [8]

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

0b0  The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.
0b1  The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA, DC GVA, and DC GZVA instructions are not classified as cache maintenance instructions, and therefore their execution cannot cause this field to be set to 1.

The reset behavior of this field is:
•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

0b0  Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1  Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:
•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

0b0  Abort caused by an instruction reading from a memory location.
0b1  Abort caused by an instruction writing to a memory location.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.
This field is **UNKNOWN** for:

- An External abort on an Atomic access.
- A fault reported using a DFSC value of \(0b110101\) or \(0b110001\), indicating an unsupported Exclusive or atomic access.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**DFSC, bits [5:0]**

Data Fault Status Code.

- \(0b000000\) Address size fault, level 0 of translation or translation table base register.
- \(0b000001\) Address size fault, level 1.
- \(0b000010\) Address size fault, level 2.
- \(0b000011\) Address size fault, level 3.
- \(0b000100\) Translation fault, level 0.
- \(0b000101\) Translation fault, level 1.
- \(0b000110\) Translation fault, level 2.
- \(0b000111\) Translation fault, level 3.
- \(0b001000\) Access flag fault, level 0.
- \(0b001001\) Access flag fault, level 1.
- \(0b001010\) Access flag fault, level 2.
- \(0b001011\) Access flag fault, level 3.
- \(0b001100\) Access flag fault, level 0. *When FEAT_LP A2 is implemented:*
- \(0b001101\) Access flag fault, level 1. *When FEAT_LP A2 is implemented:*
- \(0b001110\) Access flag fault, level 2. *When FEAT_LP A2 is implemented:*
- \(0b001111\) Access flag fault, level 3. *When FEAT_LP A2 is implemented:*
- \(0b010000\) Synchronous External abort, not on translation table walk or hardware update of translation table. *When FEAT_MTE2 is implemented:*
- \(0b010001\) Synchronous External abort on translation table walk or hardware update of translation table, level -1. *When FEAT_MTE2 is implemented:*
- \(0b010010\) Synchronous External abort on translation table walk or hardware update of translation table, level 0. *When FEAT_MTE2 is implemented:*
- \(0b010011\) Synchronous External abort on translation table walk or hardware update of translation table, level 1. *When FEAT_MTE2 is implemented:*
- \(0b010100\) Synchronous External abort on translation table walk or hardware update of translation table, level 0. *When FEAT_LP A2 is implemented:*
- \(0b010101\) Synchronous External abort on translation table walk or hardware update of translation table, level 1. *When FEAT_LP A2 is implemented:*
- \(0b010110\) Synchronous External abort on translation table walk or hardware update of translation table, level 2. *When FEAT_LP A2 is implemented:*
- \(0b010111\) Synchronous External abort on translation table walk or hardware update of translation table, level 3. *When FEAT_LP A2 is implemented:*
- \(0b011000\) Synchronous parity or ECC error on memory access, not on translation table walk. *When FEAT_LP A2 is implemented and FEAT_RAS is not implemented:*
- \(0b011011\) Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1. *When FEAT_LP A2 is implemented and FEAT_RAS is not implemented:*

---

**DFSC**, bits [5:0]

Data Fault Status Code.

- \(0b000000\) Address size fault, level 0 of translation or translation table base register.
- \(0b000001\) Address size fault, level 1.
- \(0b000010\) Address size fault, level 2.
- \(0b000011\) Address size fault, level 3.
- \(0b000100\) Translation fault, level 0.
- \(0b000101\) Translation fault, level 1.
- \(0b000110\) Translation fault, level 2.
- \(0b000111\) Translation fault, level 3.
- \(0b001000\) Access flag fault, level 0.
- \(0b001001\) Access flag fault, level 1.
- \(0b001010\) Access flag fault, level 2.
- \(0b001011\) Access flag fault, level 3.
- \(0b001100\) Access flag fault, level 0. *When FEAT_LP A2 is implemented:*
- \(0b001101\) Access flag fault, level 1. *When FEAT_LP A2 is implemented:*
- \(0b001110\) Access flag fault, level 2. *When FEAT_LP A2 is implemented:*
- \(0b001111\) Access flag fault, level 3. *When FEAT_LP A2 is implemented:*
- \(0b010000\) Synchronous External abort, not on translation table walk or hardware update of translation table. *When FEAT_MTE2 is implemented:*
- \(0b010001\) Synchronous External abort on translation table walk or hardware update of translation table, level -1. *When FEAT_MTE2 is implemented:*
- \(0b010010\) Synchronous External abort on translation table walk or hardware update of translation table, level 0. *When FEAT_MTE2 is implemented:*
- \(0b010011\) Synchronous External abort on translation table walk or hardware update of translation table, level 1. *When FEAT_MTE2 is implemented:*
- \(0b010100\) Synchronous External abort on translation table walk or hardware update of translation table, level 0. *When FEAT_LP A2 is implemented:*
- \(0b010101\) Synchronous External abort on translation table walk or hardware update of translation table, level 1. *When FEAT_LP A2 is implemented:*
- \(0b010110\) Synchronous External abort on translation table walk or hardware update of translation table, level 2. *When FEAT_LP A2 is implemented:*
- \(0b010111\) Synchronous External abort on translation table walk or hardware update of translation table, level 3. *When FEAT_LP A2 is implemented:*
- \(0b011000\) Synchronous parity or ECC error on memory access, not on translation table walk. *When FEAT_LP A2 is implemented and FEAT_RAS is not implemented:*
- \(0b011011\) Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1. *When FEAT_LP A2 is implemented and FEAT_RAS is not implemented:*
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0b11000 | When FEAT_RME is not implemented:  
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0. |
| 0b011110 | When FEAT_RME is not implemented:  
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1. |
| 0b011111 | When FEAT_RME is not implemented:  
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2. |
| 0b101011 | When FEAT_RME is not implemented:  
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3. |
| 0b10000 | Alignment fault. |
| 0b10001 | When FEAT_RME is implemented and FEAT_LPA2 is implemented:  
Granule Protection Fault on translation table walk or hardware update of translation table, level -1. |
| 0b10010 | When FEAT_RME is implemented:  
Granule Protection Fault on translation table walk or hardware update of translation table, level 0. |
| 0b10011 | When FEAT_RME is implemented:  
Granule Protection Fault on translation table walk or hardware update of translation table, level 1. |
| 0b10100 | When FEAT_RME is implemented:  
Granule Protection Fault on translation table walk or hardware update of translation table, level 2. |
| 0b100100 | When FEAT_RME is implemented:  
Granule Protection Fault on translation table walk or hardware update of translation table, level 3. |
| 0b101001 | When FEAT_LPA2 is implemented:  
Granule Protection Fault, not on translation table walk or hardware update of translation table. |
| 0b11010 | When FEAT_LPA2 is implemented:  
Address size fault, level -1. |
| 0b110100 | When FEAT_LPA2 is implemented:  
Translation fault, level -1. |
| 0b11000 | TLB conflict abort. |
| 0b110000 | When FEAT_HAFDBS is implemented:  
Unsupported atomic hardware update fault. |
| 0b11010 | IMPLEMENTATION DEFINED fault (Lockdown). |
| 0b110100 | IMPLEMENTATION DEFINED fault (Unsupported Exclusive or Atomic access). |

All other values are reserved.

For more information about the lookup level associated with a fault, see The lookup level associated with MMU faults on page D8-5190.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
ISS encoding for an exception from a trapped floating-point exception

Bit [24]
Reserved, RES0.

TFV, bit [23]
Trapped Fault Valid bit. Indicates whether the IDF, IXF, UFF, OFF, DZF, and IOF bits hold valid information about trapped floating-point exceptions.

0b0 The IDF, IXF, UFF, OFF, DZF, and IOF bits do not hold valid information about trapped floating-point exceptions and are UNKNOWN.

0b1 One or more floating-point exceptions occurred during an operation performed while executing the reported instruction. The IDF, IXF, UFF, OFF, DZF, and IOF bits indicate trapped floating-point exceptions that occurred. For more information, see Floating-point exceptions and exception traps on page A1-64.

It is IMPLEMENTATION DEFINED whether this field is set to 0 on an exception generated by a trapped floating-point exception from an instruction that is performing floating-point operations on more than one lane of a vector.

Note
This is not a requirement. Implementations can set this field to 1 on a trapped floating-point exception from an instruction and return valid information in the {IDF, IXF, UFF, OFF, DZF, IOF} fields.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [22:11]
Reserved, RES0.

VECITR, bits [10:8]
For a trapped floating-point exception from an instruction executed in AArch32 state this field is RES1.
For a trapped floating-point exception from an instruction executed in AArch64 state this field is UNKNOWN.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

IDF, bit [7]
Input Denormal floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

0b0 Input denormal floating-point exception has not occurred.
0b1 Input denormal floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
D17.2 General system control registers

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

0b0    Inexact floating-point exception has not occurred.
0b1    Inexact floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

UFF, bit [3]

Underflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

0b0    Underflow floating-point exception has not occurred.
0b1    Underflow floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OFF, bit [2]

Overflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

0b0    Overflow floating-point exception has not occurred.
0b1    Overflow floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DZF, bit [1]

Divide by Zero floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

0b0    Divide by Zero floating-point exception has not occurred.
0b1    Divide by Zero floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IOF, bit [0]

Invalid Operation floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

0b0    Invalid Operation floating-point exception has not occurred.
0b1    Invalid Operation floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

In an implementation that supports the trapping of floating-point exceptions:

- From an Exception level using AArch64, the FPCR. {IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.
• From an Exception level using AArch32, the FPSCR.\{IDE, IXE, UFE, OFE, DZE, IOE\} bits enable each of the floating-point exception traps.

**ISS encoding for an SError interrupt**

![ISS encoding diagram]

IDS, bit [24]

IMPLEMENTATION DEFINED syndrome.

0b0 Bits [23:0] of the ISS field holds the fields described in this encoding.

--- Note ---

If FEAT_RAS is not implemented, bits [23:0] of the ISS field are RES0.

--- Note ---

0b1 Bits [23:0] of the ISS field holds IMPLEMENTATION DEFINED syndrome information that can be used to provide additional information about the SError interrupt.

--- Note ---

This field was previously called ISV.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [23:14]

Reserved, RES0.

IESB, bit [13]

*When FEAT_IESB is implemented and DFSC == 0b010001:*

Implicit error synchronization event.

0b0 The SError interrupt was either not synchronized by the implicit error synchronization event or not taken immediately.

0b1 The SError interrupt was synchronized by the implicit error synchronization event and taken immediately.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

AET, bits [12:10]

*When FEAT_RAS is implemented and DFSC == 0b010001:*

Asynchronous Error Type.

Describes the PE error state after taking the SError interrupt exception.

0b000 Uncontainable (UC).

0b001 Unrecoverable state (UEU).

0b010 Restartable state (UEO).

0b011 Recoverable state (UER).

0b110 Corrected (CE).

All other values are reserved.
If multiple errors are taken as a single SError interrupt exception, the overall PE error state is reported.

--- Note ---

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EA, bit [9]

When FEAT_RAS is implemented and DFSC == 0b010001:
External abort type. Provides an IMPLEMENTATION DEFINED classification of External aborts.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bits [8:6]

Reserved, RES0.

DFSC, bits [5:0]

When FEAT_RAS is implemented:
Data Fault Status Code.
- 0b000000 Uncategorized error.
- 0b010001 Asynchronous SError interrupt.
All other values are reserved.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

ISS encoding for an exception from a Breakpoint or Vector Catch debug exception

<table>
<thead>
<tr>
<th>24</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td></td>
<td></td>
<td>IFSC</td>
</tr>
</tbody>
</table>

Bits [24:6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.
- 0b100010 Debug exception.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
For more information about generating these exceptions:

- For exceptions from AArch64, see *Breakpoint exceptions* on page D2-4707.
- For exceptions from AArch32, see *Breakpoint exceptions* on page G2-9054 and *Vector Catch exceptions* on page G2-9093.

**ISS encoding for an exception from a Software Step exception**

<table>
<thead>
<tr>
<th>24</th>
<th>23</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISV</td>
<td>RES0</td>
<td>EX</td>
<td>IFSC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ISV, bit [24]**

Instruction syndrome valid. Indicates whether the EX bit, ISS[6], is valid, as follows:

- 0b0  EX bit is RES0.
- 0b1  EX bit is valid.

See the EX bit description for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [23:7]**

Reserved, RES0.

**EX, bit [6]**

Exclusive operation. If the ISV bit is set to 1, this bit indicates whether a Load-Exclusive instruction was stepped.

- 0b0  An instruction other than a Load-Exclusive instruction was stepped.
- 0b1  A Load-Exclusive instruction was stepped.

If the ISV bit is set to 0, this bit is RES0, indicating no syndrome data is available.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IFSC, bits [5:0]**

Instruction Fault Status Code.

- 0b100010  Debug exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see *Software Step exceptions* on page D2-4742.

**ISS encoding for an exception from a Watchpoint exception**

For more information about generating these exceptions, see *Watchpoint exceptions* on page D2-4756.
Bit [24]
Reserved, RES0.

WPT, bits [23:18]

When FEAT_SME is implemented:
Watchpoint number, 0 to 15 inclusive.
All other values are reserved.

Otherwise:
Reserved, RES0.

WPTV, bit [17]

When FEAT_SME is implemented:
Watchpoint number Valid.
0b0 The WPT field is invalid, and holds an UNKNOWN value.
0b1 The WPT field is valid, and holds the number of a watchpoint that triggered a Watchpoint exception.

When a Watchpoint exception is triggered by a watchpoint match:
• If the PE sets any of FnV, FnP, or WPF to 1, then the PE sets WPTV to 1.
• If the PE sets all of FnV, FnP, and WPF to 0, then the PE sets WPTV to an IMPLEMENTATION DEFINED value, 0 or 1.

Otherwise:
Reserved, RES0.

WPF, bit [16]

When FEAT_SME is implemented:
Watchpoint might be false-positive.
0b0 The watchpoint matched the original access or set of contiguous accesses.
0b1 The watchpoint matched an access or set of contiguous accesses where the lowest accessed address was rounded down to the nearest multiple of 16 bytes and the highest accessed address was rounded up to the nearest multiple of 16 bytes minus 1, but the watchpoint might not have matched the original access or set of contiguous accesses.

Otherwise:
Reserved, RES0.

FnP, bit [15]

When FEAT_SME is implemented:
FAR not Precise.
This field only has meaning if the FAR is valid; that is, when the FnV field is 0. If the FnV field is 1, the FnP field is 0.
0b0 If the FnV field is 0, the FAR holds the virtual address of an access or set of contiguous accesses that triggered a Watchpoint exception.
0b1 The FAR holds any address within the smallest implemented translation granule that contains the virtual address of an access or set of contiguous accesses that triggered a Watchpoint exception.

Otherwise:
Reserved, RES0.

Bit [14]
Reserved, RES0.
**VNCR, bit [13]**

*When FEAT_NV2 is implemented:*

Indicates that the watchpoint came from use of VNCR_EL2 register by EL1 code.

0b0  The watchpoint was not generated by the use of VNCR_EL2 by EL1 code.

0b1  The watchpoint was generated by the use of VNCR_EL2 by EL1 code.

This field is 0 in ESR_EL1.

The reset behavior of this field is:

* On a Warm reset, this field resets to an architecturally UNDEFINED value.

**Bits [12:11]**

Reserved, RES0.

**FnV, bit [10]**

*When FEAT_SME is implemented:*

FAR not Valid.

0b0  The FAR is valid, and its value is as described by the FnP field.

0b1  The FAR is invalid, and holds an UNDEFINED value.

**Otherwise:**

Reserved, RES0.

**Bit [9]**

Reserved, RES0.

**CM, bit [8]**

Cache maintenance. Indicates whether the Watchpoint exception came from a cache maintenance instruction:

0b0  The Watchpoint exception was not generated by the execution of one of the System instructions identified in the description of value 1.

0b1  The Watchpoint exception was generated by the execution of a cache maintenance instruction. The DC ZVA, DC GVA, and DC GZVA instructions are not classified as a cache maintenance instructions, and therefore their execution cannot cause this field to be set to 1.

The reset behavior of this field is:

* On a Warm reset, this field resets to an architecturally UNDEFINED value.

**Bit [7]**

Reserved, RES0.

**WnR, bit [6]**

Write not Read. Indicates whether the Watchpoint exception was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

0b0  Watchpoint exception caused by an instruction reading from a memory location.

0b1  Watchpoint exception caused by an instruction writing to a memory location.

For Watchpoint exceptions on cache maintenance instructions, this bit always returns a value of 1.

For Watchpoint exceptions from an atomic instruction, this field is set to 0 if a read of the location would have generated the Watchpoint exception, otherwise it is set to 1.

If multiple watchpoints match on the same access, it is UNPREDICTABLE which watchpoint generates the Watchpoint exception.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**DFSC, bits [5:0]**

Data Fault Status Code.

0b100010 Debug exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see *Watchpoint exceptions* on page D2-4726.

**ISS encoding for an exception from execution of a Breakpoint instruction**

<table>
<thead>
<tr>
<th>24</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>Comment</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [24:16]**

Reserved, RES0.

**Comment, bits [15:0]**

Set to the instruction comment field value, zero extended as necessary.

For the AArch32 BKPT instructions, the comment field is described as the immediate field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see *Breakpoint Instruction exceptions* on page D2-4705.

**ISS encoding for an exception from a TSTART instruction**

<table>
<thead>
<tr>
<th>24</th>
<th>16</th>
<th>15</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>Rd</td>
<td>RES0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [24:10]**

Reserved, RES0.

**Rd, bits [9:5]**

The Rd value from the issued instruction, the general purpose register used for the destination.

**Bits [4:0]**

Reserved, RES0.

**ISS encoding for an exception from Branch Target Identification instruction**

<table>
<thead>
<tr>
<th>24</th>
<th>16</th>
<th>15</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>BTYPE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [24:2]**

Reserved, RES0.
BTYPE, bits [1:0]

This field is set to the PSTATE.BTYPE value that generated the Branch Target Exception.

For more information about generating these exceptions, see Chapter B1 The AArch64 Application Level Programmers’ Model.

ISS encoding for an exception from a Pointer Authentication instruction authentication failure

<table>
<thead>
<tr>
<th>Bit [24:2]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Exception as a result of an Instruction key or a Data key</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit [1]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Instruction Key</td>
</tr>
<tr>
<td>0b1</td>
<td>Data Key</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

<table>
<thead>
<tr>
<th>Bit [0]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>A key</td>
</tr>
<tr>
<td>0b1</td>
<td>B key</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

The following instructions generate an exception when the Pointer Authentication Code (PAC) is incorrect:
- AUTIASP, AUTIAZ, AUTIA1716.
- AUTIBSP, AUTIBZ, AUTIB1716.
- AUTIA, AUTDA, AUTIB, AUTDB.
- AUTiza, AUTIZB, AUTDZA, AUTDZB.

It is IMPLEMENTATION DEFINED whether the following instructions generate an exception directly from the authorization failure, rather than changing the address in a way that will generate a Translation fault when the address is accessed:
- RETAA, RETAB.
- BRAA, BRAB, BLRAA, BLRAB.
- BRAAZ, BRABZ, BLRAAZ, BLRABZ.
- ERETA, ERETAB.
- LDRAA, LDRAB, whether the authenticated address is written back to the base register or not.
Accessing ESR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic ESR_EL1 or ESR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ESR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWR_EL2.ESR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x138];
  else
    X[t, 64] = ESR_EL1;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = ESR_EL2;
  elsif PSTATE.EL == EL3 then
    X[t, 64] = ESR_EL1;
else
  MSR ESR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWR_EL2.ESR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    NVMem[0x138] = X[t, 64];
  else
    ESR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    ESR_EL2 = X[t, 64];
  else
    ESR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
  ESR_EL1 = X[t, 64];
MRS <Xt>, ESR_EL12

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0101</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    X[t, 64] = NVMem[0x138];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = ESR_EL1;
  else
    UNDEFINED;
elsif PSTATE_EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    X[t, 64] = ESR_EL1;
  else
    UNDEFINED;

MSR ESR_EL12, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0101</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    NVMem[0x138] = X[t, 64];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    ESR_EL1 = X[t, 64];
  else
    UNDEFINED;
elsif PSTATE_EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    ESR_EL1 = X[t, 64];
  else
    UNDEFINED;
**MRS <Xt>, ESR_EL2**

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = ESR_EL1;
    elsif EL2Enabled() & HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        X[t, 64] = ESR_EL2;
    elsif PSTATE.EL == EL3 then
        X[t, 64] = ESR_EL2;

**MSR ESR_EL2, <Xt>**

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.<NV2,NV> == '11' then
        ESR_EL1 = X[t, 64];
    elsif EL2Enabled() & HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        ESR_EL2 = X[t, 64];
    elsif PSTATE.EL == EL3 then
        ESR_EL2 = X[t, 64];
D17.2.38  ESR_EL2, Exception Syndrome Register (EL2)

The ESR_EL2 characteristics are:

**Purpose**

Holds syndrome information for an exception taken to EL2.

**Configurations**

AArch64 System register ESR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HSR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

ESR_EL2 is a 64-bit register.

**Field descriptions**

ESR_EL2 is made UNKNOWN as a result of an exception return from EL2.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL2, the value of ESR_EL2 is UNKNOWN. The value written to ESR_EL2 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

**Bits [63:37]**

Reserved, RES0.

**ISS2, bits [36:32]**

*When FEAT_LS64 is implemented:*

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort exception for a Translation fault, Access flag fault, or Permission fault, then this field holds register specifier, Xs.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort exception for a Translation fault, Access flag fault, or Permission fault, then this field holds register specifier, Xs.

Otherwise, this field is RES0.

*Otherwise:*

Reserved, RES0.

**EC, bits [31:26]**

Exception Class. Indicates the reason for the exception that this register holds information about.

For each EC value, the table references a subsection that gives information about:

- The cause of the exception, for example the configuration required to enable the trap.
- The encoding of the associated ISS.
Possible values of the EC field are:

**EC == 0b000000**
- Unknown reason.
  - See ISS encoding for exceptions with an unknown reason.

**EC == 0b000001**
- Trapped WF* instruction execution.
  - Conditional WF* instructions that fail their condition code check do not cause an exception.
  - See ISS encoding for an exception from a WF* instruction.

**EC == 0b000011**
- When AArch32 is supported:
  - Trapped MCR or MRC access with (coproc==0b1111) that is not reported using EC 0b000000.
  - See ISS encoding for an exception from an MCR or MRC access.

**EC == 0b000100**
- When AArch32 is supported:
  - Trapped MCRR or MRRC access with (coproc==0b1111) that is not reported using EC 0b000000.
  - See ISS encoding for an exception from an MCRR or MRRC access.

**EC == 0b000101**
- When AArch32 is supported:
  - Trapped MCR or MRC access with (coproc==0b1110).
  - See ISS encoding for an exception from an MCR or MRC access.

**EC == 0b000110**
- When AArch32 is supported:
  - Trapped LDC or STC access.
  - The only architected uses of these instruction are:
    - An STC to write data to memory from DBGDTRRXint.
    - An LDC to read data from memory to DBGDTRTXint.
  - See ISS encoding for an exception from an LDC or STC instruction.

**EC == 0b000111**
- Access to SME, SVE, Advanced SIMD or floating-point functionality trapped by CPACR_EL1.FPEN, CPTR_EL2.FPEN, CPTR_EL2.TFP, or CPTR_EL3.TFP control.
  - Excludes exceptions resulting from CPACR_EL1 when the value of HCR_EL2.TGE is 1, or because SVE or Advanced SIMD and floating-point are not implemented. These are reported with EC value 0b000000.
  - See ISS encoding for an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from the FPEN and TFP traps.

**EC == 0b001000**
- When AArch32 is supported:
  - Trapped VMRS access, from ID group trap, that is not reported using EC 0b000111.
  - See ISS encoding for an exception from an MCR or MRC access.

**EC == 0b001001**
- When FEAT_P Auth is implemented:
  - Trapped use of a Pointer authentication instruction because HCR_EL2.API == 0 || SCR_EL3.API == 0.
  - See ISS encoding for an exception from a Pointer Authentication instruction when HCR_EL2.API == 0 || SCR_EL3.API == 0.

**EC == 0b001010**
- When FEAT_LS64 is implemented:
An exception from an LD64B or ST64B* instruction.
See ISS encoding for an exception from an LD64B or ST64B* instruction.

EC == 0b0001100
When AArch32 is supported:
Trapped MRRC access with (coproc==0b1110).
See ISS encoding for an exception from an MCRR or MRRC access.

EC == 0b0001101
When FEAT_BTI is implemented:
Branch Target Exception.
See ISS encoding for an exception from Branch Target Identification instruction.

EC == 0b010001
When AArch32 is supported:
SVC instruction execution in AArch32 state.
This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TGE is 1.
See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 0b010010
When AArch32 is supported:
HVC instruction execution in AArch32 state, when HVC is not disabled.
See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 0b010011
When AArch32 is supported:
SMC instruction execution in AArch32 state, when SMC is not disabled.
This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.
See ISS encoding for an exception from SMC instruction execution in AArch32 state.

EC == 0b010101
When AArch64 is supported:
SVC instruction execution in AArch64 state.
See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 0b010110
When AArch64 is supported:
HVC instruction execution in AArch64 state, when HVC is not disabled.
See ISS encoding for an exception from HVC or SVC instruction execution.

EC == 0b010111
When AArch64 is supported:
SMC instruction execution in AArch64 state, when SMC is not disabled.
This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.
See ISS encoding for an exception from SMC instruction execution in AArch64 state.

EC == 0b011000
When AArch64 is supported:
Trapped MSR, MRS or System instruction execution in AArch64 state, that is not reported using EC 0b000000, 0b000001 or 0b000111.
This includes all instructions that cause exceptions that are part of the encoding space defined in System instruction class encoding overview on page C5-655, except for those exceptions reported using EC values 0b000000, 0b000001, or 0b000111.

See ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state.

**EC == 0b011001**

*When FEAT_SVE is implemented:*

Access to SVE functionality trapped as a result of CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ, that is not reported using EC 0b000000.

See ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ.

**EC == 0b011010**

*When FEAT_PAuth is implemented and FEAT_NV is implemented:*

Trapped ERET, ERETTA, or ERETB instruction execution.

See ISS encoding for an exception from an ERET, ERETTA, or ERETB instruction.

**EC == 0b011011**

*When FEAT_TME is implemented:*

Exception from an access to a TSTART instruction at EL0 when SCTLR_EL1.TME0 == 0, EL0 when SCTLR_EL2.TME0 == 0, at EL1 when SCTLR_EL1.TME == 0, at EL2 when SCTLR_EL2.TME == 0 or at EL3 when SCTLR_EL3.TME == 0.

See ISS encoding for an exception from a TSTART instruction.

**EC == 0b011100**

*When FEAT_FPAC is implemented:*

Exception from a Pointer Authentication instruction authentication failure

See ISS encoding for an exception from a Pointer Authentication instruction authentication failure.

**EC == 0b011101**

*When FEAT_SME is implemented:*

Access to SME functionality trapped as a result of CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, CPTR_EL3.ESM, or an attempted execution of an instruction that is illegal because of the value of PSTATE.SM or PSTATE.ZA, that is not reported using EC 0b000000.

See ISS encoding for an exception due to SME functionality.

**EC == 0b011110**

*When FEAT_RME is implemented:*

Exception from a Granule Protection Check

See ISS encoding for an exception from a Granule Protection Check.

**EC == 0b100000**

Instruction Abort from a lower Exception level.

Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.

See ISS encoding for an exception from an Instruction Abort.

**EC == 0b100001**

Instruction Abort taken without a change in Exception level.

Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.

See ISS encoding for an exception from an Instruction Abort.
EC == 0b100010
PC alignment fault exception.
See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.

EC == 0b100100
Data Abort exception from a lower Exception level, excluding Data Abort exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support.
These Data Abort exceptions might be generated from Exception levels in any Execution state.
Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.
See ISS encoding for an exception from a Data Abort.

EC == 0b100101
Data Abort exception without a change in Exception level, or Data Abort exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support.
Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.
See ISS encoding for an exception from a Data Abort.

EC == 0b100110
SP alignment fault exception.
See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.

EC == 0b100111
When FEAT_MOPS is implemented:
Memory Operation Exception.
See ISS encoding for an exception from the Memory Copy and Memory Set instructions.

EC == 0b101000
When AArch32 is supported:
Trapped floating-point exception taken from AArch32 state.
This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.
See ISS encoding for an exception from a trapped floating-point exception.

EC == 0b101111
When AArch64 is supported:
Trapped floating-point exception taken from AArch64 state.
This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.
See ISS encoding for an exception from a trapped floating-point exception.

EC == 0b101100
SError interrupt.
See ISS encoding for an SError interrupt.

EC == 0b110000
Breakpoint exception from a lower Exception level.
See ISS encoding for an exception from a Breakpoint or Vector Catch debug exception.
EC == 0b110001
Breakpoint exception taken without a change in Exception level.
See ISS encoding for an exception from a Breakpoint or Vector Catch debug exception.

EC == 0b110010
Software Step exception from a lower Exception level.
See ISS encoding for an exception from a Software Step exception.

EC == 0b110011
Software Step exception taken without a change in Exception level.
See ISS encoding for an exception from a Software Step exception.

EC == 0b110100
Watchpoint from a lower Exception level, excluding Watchpoint Exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support.
These Watchpoint Exceptions might be generated from Exception levels using any Execution state.
See ISS encoding for an exception from a Watchpoint exception.

EC == 0b110101
Watchpoint exceptions without a change in Exception level, or Watchpoint exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support.
See ISS encoding for an exception from a Watchpoint exception.

EC == 0b111000
When AArch32 is supported:
BKPT instruction execution in AArch32 state.
See ISS encoding for an exception from execution of a Breakpoint instruction.

EC == 0b111010
When AArch32 is supported:
Vector Catch exception from AArch32 state.
The only case where a Vector Catch exception is taken to an Exception level that is using AArch64 is when the exception is routed to EL2 and EL2 is using AArch64.
See ISS encoding for an exception from a Breakpoint or Vector Catch debug exception.

EC == 0b111100
When AArch64 is supported:
BRK instruction execution in AArch64 state.
See ISS encoding for an exception from execution of a Breakpoint instruction.

All other EC values are reserved by Arm, and:

- Unused values in the range 0b000000 - 0b101100 (0x00 - 0x2C) are reserved for future use for synchronous exceptions.
- Unused values in the range 0b101101 - 0b111111 (0x2D - 0x3F) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [25]
Instruction Length for synchronous exceptions. Possible values of this bit are:
0b0 16-bit instruction trapped.
0b1 32-bit instruction trapped. This value is also used when the exception is one of the following:

- An SError interrupt.
- An Instruction Abort exception.
- A PC alignment fault exception.
- An SP alignment fault exception.
- A Data Abort exception for which the value of the ISV bit is 0.
- An Illegal Execution state exception.
- Any debug exception except for Breakpoint instruction exceptions. For Breakpoint instruction exceptions, this bit has its standard meaning:
  — 0b0: 16-bit T32 BKPT instruction.
  — 0b1: 32-bit A32 BKPT instruction or A64 BRK instruction.
- An exception reported using EC value 0b000000.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ISS, bits [24:0]**

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

Typically, an ISS encoding has a number of subfields. When an ISS subfield holds a register number, the value returned in that field is the AArch64 view of the register number.

For an exception taken from AArch32 state, see *Mapping of the general-purpose registers between the Execution states on page D1-4684.*

If the AArch32 register descriptor is 0b1111, then:

- If the instruction that generated the exception was not UNPREDICTABLE, the field takes the value 0b11111.
- If the instruction that generated the exception was UNPREDICTABLE, the field takes an UNKNOWN value that must be either:
  — The AArch64 view of the register number of a register that might have been used at the Exception level from which the exception was taken.
  — The value 0b11111.

**ISS encoding for exceptions with an unknown reason**

![ISS encoding diagram]

**Bits [24:0]**

Reserved, RES0.

When an exception is reported using this EC code the IL field is set to 1.

This EC code is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

- The attempted execution of an instruction bit pattern that has no allocated instruction or that is not accessible at the current Exception level and Security state, including:
  — A read access using a System register pattern that is not allocated for reads or that does not permit reads at the current Exception level and Security state.
— A write access using a System register pattern that is not allocated for writes or that does not permit writes at the current Exception level and Security state.
— Instruction encodings that are unallocated.
— Instruction encodings for instructions or System registers that are not implemented in the implementation.

• In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.
• In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.
• In AArch32 state, attempted execution of a short vector floating-point instruction.
• In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.
• An exception generated because of the value of one of the SCTRL_EL1.{ITD, SED, CP15BEN} control bits.
• Attempted execution of:
  — An HVC instruction when disabled by HCR_EL2.HCD or SCR_EL3.HCE.
  — An SMC instruction when disabled by SCR_EL3.SMD.
  — An HLT instruction when disabled by EDSCR.HDE.
• Attempted execution of an MSR or MRS instruction to access SP_EL0 when the value of SPSel.SP is 0.
• Attempted execution of an MSR or MRS instruction using a _EL12 register name when HCR_EL2.E2H == 0.
• Attempted execution, in Debug state, of:
  — A DCPS1 instruction when the value of HCR_EL2.TGE is 1 and EL2 is disabled or not implemented in the current Security state.
  — A DCPS2 instruction from EL1 or EL0 when EL2 is disabled or not implemented in the current Security state.
  — A DCPS3 instruction when the value of EDSCR.SDD is 1, or when EL3 is not implemented.
• When EL3 is using AArch64, attempted execution from Secure EL1 of an SRS instruction using R13_mon.
• In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, or EL0 of an instruction that is configured to trap to EL3.
• In AArch32 state, the attempted execution of an MRS (banked register) or an MSR (banked register) instruction to SPSR_mon, SP_mon, or LR_mon.
• An exception that is taken to EL2 because the value of HCR_EL2.TGE is 1 that, if the value of HCR_EL2.TGE was 0 would have been reported with an ESR_ELx.EC value of 0b000111.
• In Non-transactional state, attempted execution of a TCOMMIT instruction.

**ISS encoding for an exception from a WF* instruction**

<table>
<thead>
<tr>
<th>26</th>
<th>23</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV</td>
<td>COND</td>
<td>RES0</td>
<td>RN</td>
<td>RES0</td>
<td>RV</td>
<td>TI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CV, bit [24]
Condition code valid.
0b0    The COND field is not valid.
0b1    The COND field is valid.
For exceptions taken from AArch64, CV is set to 1.
For exceptions taken from AArch32:
  • When an A32 instruction is trapped, CV is set to 1.
  • When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or
    set to 0. See the description of the COND field for more information.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]
For exceptions taken from AArch64, this field is set to 0b1110.
The condition code for the trapped instruction. This field is valid only for exceptions taken from
AArch32, and only when the value of CV is 1.
For exceptions taken from AArch32:
  • When an A32 instruction is trapped, CV is set to 1 and:
    — If the instruction is conditional, COND is set to the condition code field value from the
      instruction.
    — If the instruction is unconditional, COND is set to 0b1110.
  • A conditional A32 instruction that is known to pass its condition code check can be presented
    either:
    — With COND set to 0b1110, the value for unconditional.
    — With the COND value held in the instruction.
  • When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
    — CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the
      SPSR.IT field to determine the condition, if any, of the T32 instruction.
    — CV is set to 1 and COND is set to the condition code for the condition that applied to
      the instruction.
  • For an implementation that, for both A32 and T32 instructions, takes an exception on a
    trapped conditional instruction only if the instruction passes its condition code check, these
    definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND
    field is set to 0b1110, or to the value of any condition that applied to the instruction.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:10]
Reserved, RES0.

RN, bits [9:5]

When FEAT_WFxT is implemented:
Register Number. Indicates the register number supplied for a WFET or WFIT instruction.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.
Bits [4:3]

Reserved, RES0.

RV, bit [2]

*When FEAT_WFxT is implemented:*

Register field Valid.

If TI[1] == 1, then this field indicates whether RN holds a valid register number for the register argument to the trapped WFET or WFIT instruction.

- 0b0 Register field invalid.
- 0b1 Register field valid.

If TI[1] == 0, then this field is RES0.

This field is set to 1 on a trap on WFET or WFIT.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

TI, bits [1:0]

Trapped instruction. Possible values of this bit are:

- 0b00 WFI trapped.
- 0b01 WFE trapped.
- 0b10 *When FEAT_WFxT is implemented:* WFIT trapped.
- 0b11 *When FEAT_WFxT is implemented:* WFET trapped.

When FEAT_WFxT is implemented, this is a two bit field as shown. Otherwise, bit[1] is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating this exception:

- SCTLR_EL1.{nTWE, nTWI}.
- HCR_EL2.{TWE, TWI}.
- SCR_EL3.{TWE, TWI}.

**ISS encoding for an exception from an MCR or MRC access**

```
+---------+---------+---------+---------+---------+--------+--------+--------+--------+--------+--------+
| CV      | COND    | Opc2    | Opc1    | Cn      | Rt     | Crm    | Direction |
+---------+---------+---------+---------+---------+--------+--------+---------+---------+---------+---------+
```

CV, bit [24]

Condition code valid.

- 0b0 The COND field is not valid.
- 0b1 The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.
For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc2, bits [19:17]

The Opc2 value from the issued instruction.
For a trapped VMRS access, holds the value 0b000.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc1, bits [16:14]

The Opc1 value from the issued instruction.
For a trapped VMRS access, holds the value 0b111.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.
For a trapped VMRS access, holds the reg field from the VMRS instruction encoding.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Rt, bits [9:5]
The Rt value from the issued instruction, the general-purpose register used for the transfer.

If the Rt value is not 0b1111, then the reported value gives the AArch64 view of the register.
Otherwise, if the Rt value is 0b1111:

- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value 0b11111.
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value 0b11111.

See Mapping of the general-purpose registers between the Execution states on page D1-4684.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]
The CRm value from the issued instruction.
For a trapped VMRS access, holds the value 0b0000.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]
Indicates the direction of the trapped instruction.

0b0   Write to System register space. MCR instruction.
0b1   Read from System register space. MRC or VMRS instruction.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b000011:

- If FEAT_TIDCP1 is implemented, SCTLR_EL1.TIDCP, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1.
- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1 or EL2.
- PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1 or EL2.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors registers from EL0 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions at EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.{TSW, TPC, TPU} for execution of cache maintenance instructions at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register at EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
• **HCR_EL2.TIDCP**, for accesses to lockdown, DMA, and TCM operations at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

• **If FEAT_TIDCP1** is implemented, **SCTLR_EL2.TIDCP**, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

• **HCR_EL2.TID1, TID2, TID3**, for accesses to ID registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

• **CPTR_EL2.TCPAC**, for accesses to CPACR_EL1 or CPACR using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

• **HSTR_EL2.T<n>**, for accesses to System registers using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

• **CNTHCTL_EL2.EL1PCEN**, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

• **MDCR_EL2.TPM, TPMCR**, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

• **CPTR_EL2.TAM**, for accesses to Activity Monitors registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

• **CPTR_EL3.TCPAC**, for accesses to CPACR from EL1 and EL2, and accesses to HCPTR from EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL3.

• **MDCR_EL3.TPM**, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL3.

• **CPTR_EL3.TAM**, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL3.

• **If FEAT_FGT** is implemented, MCR or MRC access to some registers at EL0, trapped to EL2.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b0008101:

• **CPACR_EL1.TTA** for accesses to trace registers, MCR or MRC access (coproc == 0b1110) trapped to EL1 or EL2.

• **MDSCR_EL1.TDCC**, for accesses to the Debug Communications Channel (DCC) registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL1 or EL2.

• **If FEAT_FGT** is implemented, **MDCR_EL2.TDCC** for accesses to the DCC registers at EL0 and EL1 trapped to EL2, and **MDCR_EL3.TDCC** for accesses to the DCC registers at EL0, EL1, and EL2 trapped to EL3.

• **HCR_EL2.TID0**, for accesses to the JIDR register in the ID group 0 at EL0 and EL1 using AArch32, MRC access (coproc == 0b1111) trapped to EL2.

• **CPTR_EL2.TTA**, for accesses to trace registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL2.

• **MDCR_EL2.TDRA**, for accesses to Debug ROM registers DBGDRAR and DBGDSAR using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL2.

• **MDCR_EL2.TDOSA**, for accesses to powerdown debug registers, using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL2.

• **MDCR_EL2.TDA**, for accesses to other debug registers, using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL2.

• **CPTR_EL3.TTA**, for accesses to trace registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.
• **MDCR_EL3.TDOSA**, for accesses to powerdown debug registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.

• **MDCR_EL3.TDA**, for accesses to other debug registers, using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b001000:

• **HCR_EL2.TID0**, for accesses to the FPSID register in ID group 0 at EL1 using AArch32 state, VMRS access trapped to EL2.

• **HCR_EL2.TID3**, for accesses to registers in ID group 3 including MVFR0, MVFR1 and MVFR2, VMRS access trapped to EL2.

**ISS encoding for an exception from an LD64B or ST64B* instruction**

<table>
<thead>
<tr>
<th>ISS</th>
<th>0b00000000000000000000000000000000</th>
<th>ST64BV instruction trapped.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISS</td>
<td>0b00000000000000000000000000000001</td>
<td>When FEAT_LS64_ACCDATA is implemented:</td>
</tr>
<tr>
<td>ISS</td>
<td>0b00000000000000000000000000000010</td>
<td>ST64BV0 instruction trapped.</td>
</tr>
</tbody>
</table>

All other values are reserved.

**ISS encoding for an exception from an MCRR or MRRC access**

<table>
<thead>
<tr>
<th>CV</th>
<th>COND</th>
<th>Opc1</th>
<th>Rt2</th>
<th>Rt</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>Direction</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CV, bit [24]**

Condition code valid.

0b0 The COND field is not valid.

0b1 The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

• When an A32 instruction is trapped, CV is set to 1.

• When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**COND, bits [23:20]**

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.
For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.

- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.

- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Opc1, bits [19:16]**

The Opc1 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [15]**

Reserved, RES0.

**Rt2, bits [14:10]**

The Rt2 value from the issued instruction, the second general-purpose register used for the transfer.

If the Rt2 value is not 0b1111, then the reported value gives the AArch64 view of the register.

Otherwise, if the Rt2 value is 0b1111:

- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value 0b11111.
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value 0b11111.

See *Mapping of the general-purpose registers between the Execution states on page D1-4684*.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Rt, bits [9:5]**

The Rt value from the issued instruction, the first general-purpose register used for the transfer.

If the Rt value is not 0b1111, then the reported value gives the AArch64 view of the register.

Otherwise, if the Rt value is 0b1111:

- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value 0b11111.
• If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  — The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  — The value 0b11111.

See Mapping of the general-purpose registers between the Execution states on page D1-4684.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]
The CRm value from the issued instruction.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]
Indicates the direction of the trapped instruction.
0b0 Read to System register space. MCRR instruction.
0b1 Write from System register space. MRRC instruction.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b000100:
• CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.
• PMUSERENR_EL0.{CR, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.
• AMUSERENR_EL0.{EN}, for accesses to Activity Monitors registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.
• HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
• HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
• CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
• MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
• CPTR_EL2.TAM, for accesses to Activity Monitors registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
• MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL3.
• CPTR_EL3.TAM, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL3.
• If FEAT_FGT is implemented, HDFGRTR_EL2.PMCCNTR_EL0 for MRRC access and HDFGWTR_EL2.PMCCNTR_EL0 for MCRR access to PMCCNTR at EL0, trapped to EL2.
The following fields describe configuration settings for generating exceptions that are reported using EC value 0b001100:

- **MDSCR_EL1.TDCC**, for accesses to the Debug ROM registers DBGDSAR and DBGDRAR at EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1110) trapped to EL1 or EL2.
- **MDCR_EL2.TDRA**, for accesses to Debug ROM registers DBGDRAR and DBGDSAR using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL2.
- **MDCR_EL3.TDA**, for accesses to debug registers, using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL3.
- **CPACR_EL1.TTA** for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL1 or EL2.
- **CPTR_EL2.TTA**, for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL2.
- **CPTR_EL3.TTA**, for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL3.

**Note**

If the Armv8-A architecture is implemented with an ETMv4 implementation, MCRR and MRRC accesses to trace registers are UNDEFINED and the resulting exception is higher priority than an exception due to these traps.

**ISS encoding for an exception from an LDC or STC instruction**

<table>
<thead>
<tr>
<th>CV</th>
<th>COND</th>
<th>imm8</th>
<th>RES0</th>
<th>Rn</th>
<th>AM</th>
<th>Offset</th>
<th>Direction</th>
</tr>
</thead>
</table>

**CV, bit [24]**

Condition code valid.

- 0b0 The COND field is not valid.
- 0b1 The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**COND, bits [23:20]**

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
• A conditional A32 instruction that is known to pass its condition code check can be presented either:
  — With COND set to 0b1110, the value for unconditional.
  — With the COND value held in the instruction.
• When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  — CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  — CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
• For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**imm8, bits [19:12]**

The immediate value from the issued instruction.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [11:10]**

Reserved, RES0.

**Rn, bits [9:5]**

The Rn value from the issued instruction, the general-purpose register used for the transfer.
If the Rn value is not 0b1111, then the reported value gives the AArch64 view of the register. Otherwise, if the Rn value is 0b1111:
• If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value 0b11111.
• If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  — The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  — The value 0b11111.

See *Mapping of the general-purpose registers between the Execution states* on page D1-4684.
This field is valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction. When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Offset, bit [4]**

Indicates whether the offset is added or subtracted:
0b0 Subtract offset.
0b1 Add offset.
This bit corresponds to the U bit in the instruction encoding.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
AM, bits [3:1]

Addressing mode. The permitted values of this field are:

- 0b000  Immediate unindexed.
- 0b001  Immediate post-indexed.
- 0b010  Immediate offset.
- 0b011  Immediate pre-indexed.
- 0b100  For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.
- 0b110  For a trapped STC instruction, this encoding is reserved.

The values 0b101 and 0b111 are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in Reserved values in System and memory-mapped registers and translation table entries on page K1-11600.

Bit [2] in this subfield indicates the instruction form, immediate or literal.

Bits [1:0] in this subfield correspond to the bits \{P, W\} in the instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

- 0b0  Write to memory. STC instruction.
- 0b1  Read from memory. LDC instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

The following fields describe the configuration settings for the traps that are reported using EC value 0b000110:

- **MDSCR_EL1.TDCC**, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint trapped to EL1 or EL2.
- **MDCR_EL2.TDA**, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint MCR or MRC access trapped to EL2.
- **MDCR_EL3.TDA**, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint MCR or MRC access trapped to EL3.

- If FEAT_FGT is implemented, **MDCR_EL2.TDCC** for LDC and STC accesses to the DCC registers at EL0 and EL1 trapped to EL2, and **MDCR_EL3.TDCC** for accesses to the DCC registers at EL0, EL1, and EL2 trapped to EL3.

**ISS encoding for an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from the FPEN and TFP traps**

<table>
<thead>
<tr>
<th>CV</th>
<th>COND</th>
<th>RES0</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>23</td>
<td>20</td>
</tr>
</tbody>
</table>

The accesses covered by this trap include:

- Execution of SVE or Advanced SIMD and floating-point instructions.
- Accesses to the Advanced SIMD and floating-point System registers.
- Execution of SME instructions.
For an implementation that does not include either SVE or support for Advanced SIMD and floating-point, the exception is reported using the EC value 0b000000.

**CV**, bit [24]

Condition code valid.

- 0b0       The COND field is not valid.
- 0b1       The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**COND**, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [19:0]**

Reserved, RES0.

The following fields describe the configuration settings for the traps that are reported using EC value 0b000111:

- **CPACR_EL1.FPEN**, for accesses to SIMD and floating-point registers trapped to EL1.
- **CPTR_EL2.FPEN** and **CPTR_EL2.TFP**, for accesses to SIMD and floating-point registers trapped to EL2.
- **CPTR_EL3.TFP**, for accesses to SIMD and floating-point registers trapped to EL3.
**ISS encoding for an exception from an access to SVE functionality, resulting from**

*CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ*

The accesses covered by this trap include:

- Execution of SVE instructions when the PE is not in Streaming SVE mode.
- Accesses to the SVE System registers, ZCR_ELx.

For an implementation that does not include SVE, the exception is reported using the EC value `0b000000`.

**Bits [24:0]**

Reserved, RES0.

The following fields describe the configuration settings for the traps that are reported using EC value `0b011001`:

- **CPACR_EL1.ZEN**, for execution of SVE instructions and accesses to SVE registers at EL0 or EL1, trapped to EL1.
- **CPTR_EL2.ZEN** and **CPTR_EL2.TZ**, for execution of SVE instructions and accesses to SVE registers at EL0, EL1, or EL2, trapped to EL2.
- **CPTR_EL3.EZ**, for execution of SVE instructions and accesses to SVE registers from all Exception levels, trapped to EL3.

**ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault**

**Bits [24:0]**

Reserved, RES0.

There are no configuration settings for generating Illegal Execution state exceptions and PC alignment fault exceptions. For more information about PC alignment fault exceptions, see [PC alignment checking](#) on page D1-4668.

[SP alignment checking](#) on page D1-4668 describes the configuration settings for generating SP alignment fault exceptions.

**ISS encoding for an exception from the Memory Copy and Memory Set instructions**

**MemInst, bit [24]**

Indicates the memory instruction class causing the exception.

- `0b0` CPYFE*, CPYFM*, CPYE*, and CPYM* instructions.
- `0b1` SETE*, SETM*, SETGE*, and SETGM* instructions.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**isSETG, bit [23]**
Indicates whether the instruction belongs to SETGM* or SETGE* class of instruction.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Not a SETGM* or SETGE* instruction.</td>
</tr>
<tr>
<td>0b1</td>
<td>SETGM* or SETGE* instruction.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Options, bits [22:19]**
Options : the Options field of the instruction.

For Memory Copy instructions, bits[22:19] forms the Options field, which holds the bits[15:12] of the instruction.

For Memory Set instructions:

- Bits[22:21] are RES0.
- Bits[20:19] form the Options field, which holds the bits[13:12] of the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**FromEpilogue, bit [18]**
Indicates whether the instruction belongs to the epilogue class of Memory Copy or Memory Set instructions.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Not an epilogue instruction.</td>
</tr>
<tr>
<td>0b1</td>
<td>CPYE*, CPYFE*, SETE*, or SETGE* instruction.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**WrongOption, bit [17]**
Algorithm option.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>WrongOption is false.</td>
</tr>
<tr>
<td>0b1</td>
<td>WrongOption is true.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**OptionA, bit [16]**
Algorithm type indicated by the PSTATE.C bit.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>OptionB indicated by PSTATE.C is 0.</td>
</tr>
<tr>
<td>0b1</td>
<td>OptionA indicated by PSTATE.C is 1.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [15]**
Reserved, RES0.

**destreg, bits [14:10]**
The destination register value from the issued instruction, containing the destination address.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
**srcreg, bits [9:5]**

The source register value from the issued instruction, containing either the source address or the source data.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**sizereg, bits [4:0]**

The size register value from the issued instruction, containing the number of bytes to be transferred or set.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ISS encoding for an exception from HVC or SVC instruction execution**

<table>
<thead>
<tr>
<th>24</th>
<th>16</th>
<th>15</th>
<th>imm16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [24:16]**

Reserved, RES0.

**imm16, bits [15:0]**

The value of the immediate field from the HVC or SVC instruction.

For an HVC instruction, and for an A64 SVC instruction, this is the value of the imm16 field of the issued instruction.

For an A32 or T32 SVC instruction:
- If the instruction is unconditional, then:
  - For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.
  - For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- If the instruction is conditional, this field is UNKNOWN.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

In AArch32 state, the HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

For T32 and A32 instructions, see SVC and HVC.

For A64 instructions, see SVC and HVC.

If FEAT_FGT is implemented, HFGITR_EL2 {{SVC_EL1, SVC_EL0} control fine-grained traps on SVC execution.

**ISS encoding for an exception from SMC instruction execution in AArch32 state**

<table>
<thead>
<tr>
<th>24</th>
<th>23</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV</td>
<td>COND</td>
<td>RES0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CCKOWNPASS
For an SMC instruction that completes normally and generates an exception that is taken to EL3, the ISS encoding is RES0.

For an SMC instruction that is trapped to EL2 from EL1 because HCR_EL2.TSC is 1, the ISS encoding is as shown in the diagram.

**CV, bit [24]**

Condition code valid.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The COND field is not valid.</td>
</tr>
<tr>
<td>0b1</td>
<td>The COND field is valid.</td>
</tr>
</tbody>
</table>

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field is valid only if CCKNOWNPASS is 1, otherwise it is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**COND, bits [23:20]**

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field is valid only if CCKNOWNPASS is 1, otherwise it is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**CCKNOWNPASS, bit [19]**

Indicates whether the instruction might have failed its condition code check.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The instruction was unconditional, or was conditional and passed its condition code check.</td>
</tr>
<tr>
<td>0b1</td>
<td>The instruction was conditional, and might have failed its condition code check.</td>
</tr>
</tbody>
</table>
In an implementation in which an SMC instruction that fails its code check is not trapped, this field can always return the value 0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [18:0]**

Reserved, RES0.

**HCR_EL2.TSC** describes the configuration settings for trapping SMC instructions to EL2.

**ISS encoding for an exception from SMC instruction execution in AArch64 state**

```plaintext
  +-----+-----+-----+-----+
  | RES0| RES0| imm16| RES0 |
  +-----+-----+-----+-----+
      24 16 15 0
```

**Bits [24:16]**

Reserved, RES0.

**imm16, bits [15:0]**

The value of the immediate field from the issued SMC instruction.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

The value of ISS[24:0] described here is used both:
- When an SMC instruction is trapped from EL1 modes.
- When an SMC instruction is not trapped, so completes normally and generates an exception that is taken to EL3.

**HCR_EL2.TSC** describes the configuration settings for trapping SMC from EL1 modes.

**ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state**

```plaintext
  +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
  | RES0| Op0 | Op2 | Op1 | CRn | Rt  | CRm |
  +-----+-----+-----+-----+-----+-----+-----+
      24 22 21 20 19 17 16 14 13 10 9 5 4 1 0
```

**Bits [24:22]**

Reserved, RES0.

**Op0, bits [21:20]**

The Op0 value from the issued instruction.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Op2, bits [19:17]**

The Op2 value from the issued instruction.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Op1, bits [16:14]
The Op1 value from the issued instruction.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]
The CRn value from the issued instruction.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]
The Rt value from the issued instruction, the general-purpose register used for the transfer.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]
The CRm value from the issued instruction.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]
Indicates the direction of the trapped instruction.
0b0 Write access, including MSR instructions.
0b1 Read access, including MRS instructions.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

For exceptions caused by System instructions, see System instructions on page C4-539 for the encoding values returned by an instruction.

The following fields describe configuration settings for generating the exception that is reported using EC value 0b011000:
- If FEAT_TIDCP1 is implemented, SCTLR_EL1.TIDCP, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch64 state, MSR or MRS access trapped to EL1.
- SCTLR_EL1.UCI, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.UCT, for accesses to CTR_EL0 using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.DZE, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.UMA, for accesses to the PSTATE interrupt masks using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- CPACR_EL1.TTA, for accesses to the trace registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- If FEAT_FGT is implemented, MDCR_EL2.TDCC for accesses to the DCC registers at EL0 and EL1 trapped to EL2, and MDCR_EL3.TDCC for accesses to the DCC registers at EL0, EL1, and EL2 trapped to EL3.
• CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN} accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.

• PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to the Performance Monitor registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.

• AMUSERENR_EL0.EN, for accesses to Activity Monitors registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.

• HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers using AArch64 state, MSR or MRS access trapped to EL2.

• HCR_EL2.TDZ, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL2.

• HCR_EL2.TTLB, for execution of TLB maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.

• HCR_EL2.TS, for accesses to HD_CCR registers using AArch64 state, MSR or MRS access trapped to EL2.

• HCR_EL2.TACR, for accesses to the Auxiliary Control Register, ACTLR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.

• HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations using AArch64 state, MSR or MRS access trapped to EL2.

• If FEAT_TIDCP1 is implemented, SCTLR_EL2.TIDCP, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch64 state, MSR or MRS access trapped to EL2.

• HCR_EL2.TID1, TID2, TID3, for accesses to ID group 1, ID group 2 or ID group 3 registers, using AArch64 state, MSR or MRS access trapped to EL2.

• CPTR_EL2.TCPAC, for accesses to CPACR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.

• CPTR_EL2.TTA, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL2.

• MDCR_EL2.TTRF, for accesses to the trace filter control register, TRFCR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.

• MDCR_EL2.TDRA, for accesses to Debug ROM registers, using AArch64 state, MSR or MRS access trapped to EL2.

• MDCR_EL2.TDOSA, for accesses to powerdown debug registers using AArch64 state, MSR or MRS access trapped to EL2.

• CNTKCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2.

• MDCR_EL2.TDA, for accesses to debug registers using AArch64 state, MSR or MRS access trapped to EL2.

• MDCR_EL2.TPM, TPMCR, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL2.

• CPTR_EL2.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access trapped to EL2.

• HCR_EL2.APK, for accesses to Pointer authentication key registers. using AArch64 state, MSR or MRS access trapped to EL2.

• HCR_EL2.NV, NV1, for Nested virtualization register access, using AArch64 state, MSR or MRS access, trapped to EL2.
• **HCR_EL2.AT**, for execution of AT S1E* instructions, using AArch64 state, MSR or MRS access, trapped to EL2.

• **HCR_EL2.{TERR, FIEN}**, for accesses to RAS registers, using AArch64 state, MSR or MRS access, trapped to EL2.

• **SCR_EL3.APK**, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL3.

• **SCR_EL3.ST**, for accesses to the Counter-timer Physical Secure timer registers, using AArch64 state, MSR or MRS access trapped to EL3.

• **SCR_EL3.{TERR, FIEN}**, for accesses to RAS registers, using AArch64 state, MSR or MRS access trapped to EL3.

• **CPTR_EL3.TCPAC**, for accesses to **CPTR_EL2** and **CPACR_EL1** using AArch64 state, MSR or MRS access trapped to EL3.

• **CPTR_EL3.TTA**, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL3.

• **MDCR_EL3.TTRF**, for accesses to the trace filter control registers, **TRFCR_EL1** and **TRFCR_EL2**, using AArch64 state, MSR or MRS access trapped to EL3.

• **MDCR_EL3.TDA**, for accesses to debug registers, using AArch64 state, MSR or MRS access trapped to EL3.

• **MDCR_EL3.TDOSA**, for accesses to powerdown debug registers, using AArch64 state, MSR or MRS access trapped to EL3.

• **MDCR_EL3.TPM**, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL3.

• **CPTR_EL3.TAM**, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access, trapped to EL3.

• If **FEAT_EVT** is implemented, the following registers control traps for EL1 and EL0 Cache controls that use this EC value:
  — **HCR_EL2.{TTLBOS, TTLBIS, TICAB, TOCU, TID4}**.
  — **HCR2.{TTLBIS, TICAB, TOCU, TID4}**.

• If **FEAT_FGT** is implemented:
  — **SCR_EL3.FGTEn**, for accesses to the fine-grained trap registers, MSR or MRS access at EL2 trapped to EL3.
  — **HFGTR_EL2** for reads and **HFGWTR_EL2** for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 trapped to EL2.
  — **HFGITR_EL2** for execution of system instructions, MSR or MRS access trapped to EL2
  — **HDFGRTR_EL2** for reads and **HDFGWTR_EL2** for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 state trapped to EL2.
  — **HAFGRTR_EL2** for reads of Activity Monitor counters, using AArch64 state, MRS access at EL0 and EL1 trapped to EL2.

• If **FEAT_RNG_TRAP** is implemented:
  — **SCR_EL3.TRNDR** for reads of **RNDR** and **RNDRRS** using AArch64 state, MRS access trapped to EL3.

• If **FEAT_SME** is implemented:
  — **CPTR_EL3.ESM**, for MSR or MRS accesses to **SMPRI_EL1** at EL1, EL2, and EL3, trapped to EL3.
  — **CPTR_EL3.ESM**, for MSR or MRS accesses to **SMPRIMAP_EL2** at EL2 and EL3, trapped to EL3.
  — **SCTL_EL1.EnTP2**, for MSR or MRS accesses to **TPIDR2_EL0** at EL0, trapped to EL1 or EL2.
— SCTLR_EL2.EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, trapped to EL2.
— SCR_EL3.EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, EL1, and EL2, trapped to EL3.

• If FEAT_NMI is implemented, HCRX_EL2.TALLINT, for MSR writes of ALLINT at EL1, trapped to EL2.

**ISS encoding for an exception from an Instruction Abort**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>13-12</td>
<td>SET</td>
</tr>
<tr>
<td>11-10</td>
<td>EA</td>
</tr>
<tr>
<td>9</td>
<td>IFSC</td>
</tr>
<tr>
<td>8</td>
<td>FnV</td>
</tr>
<tr>
<td>7-6</td>
<td>RES0</td>
</tr>
<tr>
<td>5-0</td>
<td>S1PTW</td>
</tr>
</tbody>
</table>

**Bits [24:13]**

Reserved, RES0.

**SET, bits [12:11]**

*When FEAT_RAS is implemented:*

Synchronous Error Type. When IFSC is 0b010000, describes the PE error state after taking the Instruction Abort exception.

- 0b00: Recoverable state (UER).
- 0b10: Uncontainable (UC).
- 0b11: Restartable state (UEO).

All other values are reserved.

**Note**

Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in a PE state that is not recoverable.

This field is valid only if the IFSC code is 0b010000. It is RES0 for all other aborts.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**FnV, bit [10]**

Far not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

- 0b0: Far is valid.
- 0b1: Far is not valid, and holds an UNKNOWN value.

This field is valid only if the IFSC code is 0b010000. It is RES0 for all other aborts.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**EA, bit [9]**

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Bit [8]

Reserved, RES0.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

- **0b0** Fault not on a stage 2 translation for a stage 1 translation table walk.
- **0b1** Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.

- **0b000000** Address size fault, level 0 of translation or translation table base register.
- **0b000001** Address size fault, level 1.
- **0b000010** Address size fault, level 2.
- **0b000011** Address size fault, level 3.
- **0b000100** Translation fault, level 0.
- **0b000101** Translation fault, level 1.
- **0b000110** Translation fault, level 2.
- **0b000111** Translation fault, level 3.
- **0b001001** Access flag fault, level 1.
- **0b001010** Access flag fault, level 2.
- **0b001011** Access flag fault, level 3.
- **0b001100** When FEAT_LPA is implemented:
  Access flag fault, level 0.
- **0b001101** When FEAT_LPA is implemented:
  Permission fault, level 0.
- **0b001110** Permission fault, level 1.
- **0b001111** Permission fault, level 2.
- **0b010000** Permission fault, level 3.
- **0b010001** Synchronous External abort, not on translation table walk or hardware update of translation table.
- **0b010011** When FEAT_LPA is implemented:
  Synchronous External abort on translation table walk or hardware update of translation table, level -1.
- **0b010100** Synchronous External abort on translation table walk or hardware update of translation table, level 0.
- **0b010101** Synchronous External abort on translation table walk or hardware update of translation table, level 1.
- **0b010110** Synchronous External abort on translation table walk or hardware update of translation table, level 2.
- **0b010111** Synchronous External abort on translation table walk or hardware update of translation table, level 3.
When FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access, not on translation table walk.

When FEAT_LPA2 is implemented and FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access on translation table walk or
hardware update of translation table, level -1.

When FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access on translation table walk or
hardware update of translation table, level 0.

When FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access on translation table walk or
hardware update of translation table, level 1.

When FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access on translation table walk or
hardware update of translation table, level 2.

When FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access on translation table walk or
hardware update of translation table, level 3.

When FEAT_RME is implemented and FEAT_LPA2 is implemented:
Granule Protection Fault on translation table walk or hardware update of translation
table, level -1.

When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation
table, level 0.

When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation
table, level 1.

When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation
table, level 2.

When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation
table, level 3.

When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation
table, level 2.

When FEAT_RME is implemented:
Granule Protection Fault, not on translation table walk or hardware update of translation
table.

When FEAT_LPA2 is implemented:
Address size fault, level -1.

When FEAT_LPA2 is implemented:
Translation fault, level -1.

TLB conflict abort.

When FEAT_HAFDBS is implemented:
Unsupported atomic hardware update fault.

All other values are reserved.

For more information about the lookup level associated with a fault, see The lookup level associated
with MMU faults on page D8-5190.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a
stage 1 translation walk.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
ISS encoding for an exception due to SME functionality

The accesses covered by this trap include:

- Execution of SME instructions.
- Execution of SVE and Advanced SIMD instructions, when the PE is in Streaming SVE mode.
- Direct accesses of SVCR, SMCR_EL1, SMCR_EL2, SMCR_EL3.

Bits [24:3]

Reserved, RES0.

SMTC, bits [2:0]

SME Trap Code. Identifies the reason for instruction trapping.

0b000  Access to SME functionality trapped as a result of CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, or CPTR_EL3.ESM, that is not reported using EC 0b000000.

0b001  Advanced SIMD, SVE, or SVE2 instruction trapped because PSTATE.SM is 1.

0b010  SME instruction trapped because PSTATE.SM is 0.

0b011  SME instruction trapped because PSTATE.ZA is 0.

All other values are reserved.

The following fields describe the configuration settings for the traps that are reported using the EC value 0b011101:

- **CPACR_EL1.SMEN**, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR and SMCR_EL1 System registers at EL1 and EL0, trapped to EL1 or EL2.

- **CPTR_EL2.SMEN** and **CPTR_EL2.TSM**, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR, SMCR_EL1, SMCR_EL2 at EL2, EL1, or EL0, trapped to EL2.

- **CPTR_EL3.ESM**, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR, SMCR_EL1, SMCR_EL2, SMCR_EL3 from all Exception levels and any Security state, trapped to EL3.

ISS encoding for an exception from a Granule Protection Check

Bits [24:22]

Reserved, RES0.

S2PTW, bit [21]

Indicates whether the Granule Protection Check exception was on an access made for a stage 2 translation table walk.

0b0  Fault not on a stage 2 translation table walk.

0b1  Fault on a stage 2 translation table walk.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**InD, bit [20]**

Indicates whether the Granule Protection Check exception was on an instruction or data access.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Data access.</td>
</tr>
<tr>
<td>0b1</td>
<td>Instruction access.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**GPCSC, bits [19:14]**

Granule Protection Check Status Code.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000000</td>
<td>GPT address size fault at level 0.</td>
</tr>
<tr>
<td>0b000100</td>
<td>GPT walk fault at level 0.</td>
</tr>
<tr>
<td>0b000101</td>
<td>GPT walk fault at level 1.</td>
</tr>
<tr>
<td>0b001100</td>
<td>Granule protection fault at level 0.</td>
</tr>
<tr>
<td>0b001101</td>
<td>Granule protection fault at level 1.</td>
</tr>
<tr>
<td>0b010100</td>
<td>Synchronous External abort on GPT fetch at level 0.</td>
</tr>
<tr>
<td>0b010101</td>
<td>Synchronous External abort on GPT fetch at level 1.</td>
</tr>
</tbody>
</table>

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**VNCR, bit [13]**

When **FEAT_NV2** is implemented:

Indicates that the fault came from use of **VNCR_EL2** register by EL1 code.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The fault was not generated by the use of <strong>VNCR_EL2</strong>, by an MRS or MSR instruction executed at EL1.</td>
</tr>
<tr>
<td>0b1</td>
<td>The fault was generated by the use of <strong>VNCR_EL2</strong>, by an MRS or MSR instruction executed at EL1.</td>
</tr>
</tbody>
</table>

This field is 0 in ESR_EL1.

When InD is '1', this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

**Bits [12:9]**

Reserved, RES0.

**CM, bit [8]**

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.</td>
</tr>
<tr>
<td>0b1</td>
<td>The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The <strong>DC ZVA</strong>, <strong>DC GVA</strong>, and <strong>DC GZVA</strong> instructions are not classified as cache maintenance instructions, and therefore their execution cannot cause this field to be set to 1.</td>
</tr>
</tbody>
</table>
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**S1PTW, bit [7]**

Indicates whether the Granule Protection Check exception was on an access for stage 2 translation for a stage 1 translation table walk:

- 0b0 Fault not on a stage 2 translation for a stage 1 translation table walk.
- 0b1 Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**WnR, bit [6]**

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

- 0b0 Abort caused by an instruction reading from a memory location.
- 0b1 Abort caused by an instruction writing to a memory location.

When InD is '1', this field is RES0.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

This field is UNKNOWN for:
- An External abort on an Atomic access.
- A fault reported using a DFSC value of 0b111010 or 0b110001, indicating an unsupported Exclusive or atomic access.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**xFSC, bits [5:0]**

Instruction or Data Fault Status Code.

- 0b100011 *When FEAT_RME is implemented and FEAT_LPA2 is implemented*: Granule Protection Fault on translation table walk or hardware update of translation table, level -1.
- 0b100100 *When FEAT_RME is implemented*: Granule Protection Fault on translation table walk or hardware update of translation table, level 0.
- 0b100101 *When FEAT_RME is implemented*: Granule Protection Fault on translation table walk or hardware update of translation table, level 1.
- 0b100110 *When FEAT_RME is implemented*: Granule Protection Fault on translation table walk or hardware update of translation table, level 2.
- 0b100111 *When FEAT_RME is implemented*: Granule Protection Fault on translation table walk or hardware update of translation table, level 3.
- 0b101000 *When FEAT_RME is implemented*: Granule Protection Fault on translation table walk or hardware update of translation table, level 3.
Granule Protection Fault, not on translation table walk or hardware update of translation table.

All other values are reserved.

For more information about the lookup level associated with a fault, see The lookup level associated with MMU faults on page D8-5190.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ISS encoding for an exception from a Data Abort**

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this ISS encoding includes ISS2, bits[36:32].

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this ISS encoding includes ISS2, bits[36:32].

**ISV, bit [24]**

Instruction Syndrome Valid. Indicates whether the syndrome information in ISS[23:14] is valid.

- 0b0: No valid instruction syndrome. ISS[23:14] are RES0.

In ESR_EL2, ISV is 1 when FEAT_LS64 is implemented and a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

In ESR_EL2, ISV is 1 when FEAT_LS64_V is implemented and a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

In ESR_EL2, ISV is 1 when FEAT_LS64_ACCDATA is implemented and a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

For other faults reported in ESR_EL2, ISV is 0 except for the following stage 2 aborts:

- AArch64 loads and stores of a single general-purpose register (including the register specified with 0b11111, including those with Acquire/Release semantics, but excluding Load Exclusive or Store Exclusive and excluding those with writeback).
- AArch32 instructions where the instruction:
  - Is an LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT instruction.
  - Is not performing register writeback.
  - Is not using R15 as a source or destination register.

For these stage 2 aborts, ISV is UNKNOWN if the exception was generated in Debug state in memory access mode, and otherwise indicates whether ISS[23:14] hold a valid syndrome.
For faults reported in ESR_EL1 or ESR_EL3, ISV is 1 when FEAT_LS64 is implemented and a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

For faults reported in ESR_EL1 or ESR_EL3, ISV is 1 when FEAT_LS64_V is implemented and a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

For faults reported in ESR_EL1 or ESR_EL3, ISV is 1 when FEAT_LS64_ACCDATA is implemented and a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

When FEAT_RAS is implemented, ISV is 0 for any synchronous External abort.

When FEAT_RAS is not implemented, it is IMPLEMENTATION DEFINED whether ISV is set to 1 or 0 on a synchronous External abort on a stage 2 translation table walk.

For ISS reporting, a stage 2 abort on a stage 1 translation table walk does not return a valid instruction syndrome, and therefore ISV is 0 for these aborts.

When FEAT_MOPS is implemented, for a synchronous Data Abort on a Memory Copy and Memory Set instruction, ISV is 0.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

SAS, bits [23:22]

When ISV == 1:
Syndrome Access Size. Indicates the size of the access attempted by the faulting operation.

0b00  Byte
0b01  Halfword
0b10  Word
0b11  Doubleword

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0b11.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0b11.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0b11.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

SSE, bit [21]

When ISV == 1:
Syndrome Sign Extend. For a byte, halfword, or word load operation, indicates whether the data item must be sign extended.

0b0  Sign-extension not required.
0b1  Data item must be sign-extended.

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.
When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

For all other operations, this field is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**SRT, bits [20:16]**

*When ISV == 1:*

Syndrome Register Transfer. The register number of the Wu/Xt/Rt operand of the faulting instruction.

If the exception was taken from an Exception level that is using AArch32, then this is the AArch64 view of the register. See *Mapping of the general-purpose registers between the Execution states on page D1-4684.*

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**SF, bit [15]**

*When ISV == 1:*

SF

Sixty Four bit general-purpose register transfer. Width of the register accessed by the instruction is 64-bit.

- 0b0 Instruction loads/stores a 32-bit general-purpose register.
- 0b1 Instruction loads/stores a 64-bit general-purpose register.

--- **Note**

This field specifies the register width identified by the instruction, not the Execution state.

---

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 1.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 1.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 1.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
When ISV == 0 and FEAT_SME is implemented:

FnP

FAR not Precise.

0b0 The FAR holds the faulting virtual address that generated the Data Abort.

0b1 The FAR holds any virtual address within the naturally-aligned granule that contains the faulting virtual address that generated a Data Abort due to an SVE contiguous vector load/store instruction in Streaming SVE mode, or an SME load/store instruction.

For more information about the naturally-aligned fault granule, see FAR_ELx (for example, FAR_EL1).

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AR, bit [14]

When ISV == 1:

Acquire/Release.

0b0 Instruction did not have acquire/release semantics.

0b1 Instruction did have acquire/release semantics.

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VNCR, bit [13]

When FEAT_NV2 is implemented:

Indicates that the fault came from use of VNCR_EL2 register by EL1 code.

0b0 The fault was not generated by the use of VNCR_EL2, by an MRS or MSR instruction executed at EL1.

0b1 The fault was generated by the use of VNCR_EL2, by an MRS or MSR instruction executed at EL1.

This field is 0 in ESR_EL1.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
LST, bits [12:11]

When (DFSC == 0b00xxxx || DFSC == 0b101011) && DFSC != 0b0000xx:

LST
Load/Store Type. Used when a Translation fault, Access flag fault, or Permission fault generates a Data Abort.

0b00 The instruction that generated the Data Abort is not specified.

0b01 When FEAT_LS64_V is implemented:
An ST64BV instruction generated the Data Abort.

0b10 When FEAT_LS64 is implemented:
An LD64B or ST64B instruction generated the Data Abort.

0b11 When FEAT_LS64_ACCDATA is implemented:
An ST64BV0 instruction generated the Data Abort.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_RAS is implemented and DFSC == 0b010000:

SET
Synchronous Error Type. Used when a Syncronous External abort, not on a Translation table walk or hardware update of the Translation table, generated the Data Abort. Describes the PE error state after taking the Data Abort exception.

0b00 Recoverable state (UER).

0b10 Uncontainable (UC).

0b11 Restartable state (UEO).

All other values are reserved.

Note
Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in a PE state that is not recoverable.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

0b0 FAR is valid.

0b1 FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the DFSC code is 0b010000. It is RES0 for all other aborts.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
CM, bit [8]
Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

0b0 The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.
0b1 The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA, DC GVA, and DC GZVA instructions are not classified as cache maintenance instructions, and therefore their execution cannot cause this field to be set to 1.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]
For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

0b0 Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1 Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WnR, bit [6]
Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

0b0 Abort caused by an instruction reading from a memory location.
0b1 Abort caused by an instruction writing to a memory location.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

This field is UNKNOWN for:
- An External abort on an Atomic access.
- A fault reported using a DFSC value of 0b110101 or 0b110001, indicating an unsupported Exclusive or atomic access.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]
Data Fault Status Code.

0b000000 Address size fault, level 0 of translation or translation table base register.
0b000001 Address size fault, level 1.
0b000010 Address size fault, level 2.
0b000011 Address size fault, level 3.
0b000100 Translation fault, level 0.
0b000101 Translation fault, level 1.
0b000110 Translation fault, level 2.
0b00011  Translation fault, level 3.
0b001001  Access flag fault, level 1.
0b001010  Access flag fault, level 2.
0b001011  Access flag fault, level 3.
0b001000  When FEAT_LPA2 is implemented:
          Access flag fault, level 0.
0b001100  When FEAT_LPA2 is implemented:
          Permission fault, level 0.
0b001101  Permission fault, level 1.
0b001110  Permission fault, level 2.
0b001111  Permission fault, level 3.
0b010000  Synchronous External abort, not on translation table walk or hardware update of translation table.
0b010001  When FEAT_MTE2 is implemented:
          Synchronous Tag Check Fault.
0b010011  When FEAT_LPA2 is implemented:
          Synchronous External abort on translation table walk or hardware update of translation table, level -1.
0b010100  Synchronous External abort on translation table walk or hardware update of translation table, level 0.
0b010101  Synchronous External abort on translation table walk or hardware update of translation table, level 1.
0b010110  Synchronous External abort on translation table walk or hardware update of translation table, level 2.
0b010111  Synchronous External abort on translation table walk or hardware update of translation table, level 3.
0b011000  When FEAT_RAS is not implemented:
          Synchronous parity or ECC error on memory access, not on translation table walk.
0b011011  When FEAT_LPA2 is implemented and FEAT_RAS is not implemented:
          Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.
0b011100  When FEAT_RAS is not implemented:
          Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.
0b011101  When FEAT_RAS is not implemented:
          Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.
0b011110  When FEAT_RAS is not implemented:
          Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.
0b011111  When FEAT_RAS is not implemented:
          Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.
0b100001  Alignment fault.
0b100011  When FEAT_RME is implemented and FEAT_LPA2 is implemented:
          Granule Protection Fault on translation table walk or hardware update of translation table, level -1.
0b100100  When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation table, level 0.

0b100101 When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation table, level 1.

0b100110 When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation table, level 2.

0b100111 When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation table, level 3.

0b101000 When FEAT_RME is implemented:
Granule Protection Fault, not on translation table walk or hardware update of translation table.

0b101001 When FEAT_LPA2 is implemented:
Address size fault, level -1.

0b101011 When FEAT_LPA2 is implemented:
Translation fault, level -1.

0b110000 TLB conflict abort.

0b110001 When FEAT_HAFDBS is implemented:
Unsupported atomic hardware update fault.

0b110100 IMPLEMENTATION DEFINED fault (Lockdown).

0b110101 IMPLEMENTATION DEFINED fault (Unsupported Exclusive or Atomic access).

All other values are reserved.

For more information about the lookup level associated with a fault, see The lookup level associated with MMU faults on page D8-5190.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ISS encoding for an exception from a trapped floating-point exception**

Bit [24]

Reserved, RES0.

TFV, bit [23]

Trapped Fault Valid bit. Indicates whether the IDF, IXF, UFF, OFF, DZF, and IOF bits hold valid information about trapped floating-point exceptions.

0b0 The IDF, IXF, UFF, OFF, DZF, and IOF bits do not hold valid information about trapped floating-point exceptions and are UNKNOWN.
0b1 One or more floating-point exceptions occurred during an operation performed while executing the reported instruction. The IDF, IXF, UFF, OFF, DZF, and IOF bits indicate trapped floating-point exceptions that occurred. For more information, see Floating-point exceptions and exception traps on page A1-64.

It is IMPLEMENTATION DEFINED whether this field is set to 0 on an exception generated by a trapped floating-point exception from an instruction that is performing floating-point operations on more than one lane of a vector.

--- Note ---

This is not a requirement. Implementations can set this field to 1 on a trapped floating-point exception from an instruction and return valid information in the {IDF, IXF, UFF, OFF, DZF, IOF} fields.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [22:11]

Reserved, RES0.

VECITR, bits [10:8]

For a trapped floating-point exception from an instruction executed in AArch32 state this field is RES1.

For a trapped floating-point exception from an instruction executed in AArch64 state this field is UNKNOWN.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IDF, bit [7]

Input Denormal floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:
- 0b0 Input denormal floating-point exception has not occurred.
- 0b1 Input denormal floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:
- 0b0 Inexact floating-point exception has not occurred.
- 0b1 Inexact floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

UFF, bit [3]

Underflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:
- 0b0 Underflow floating-point exception has not occurred.
- 0b1 Underflow floating-point exception occurred during execution of the reported instruction.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**OFF, bit [2]**

Overflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

- 0b0  Overflow floating-point exception has not occurred.
- 0b1  Overflow floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**DZF, bit [1]**

Divide by Zero floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

- 0b0  Divide by Zero floating-point exception has not occurred.
- 0b1  Divide by Zero floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IOF, bit [0]**

Invalid Operation floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

- 0b0  Invalid Operation floating-point exception has not occurred.
- 0b1  Invalid Operation floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

In an implementation that supports the trapping of floating-point exceptions:

- From an Exception level using AArch64, the FPCR.\{IDE, IXE, UFE, OFE, DZE, IOE\} bits enable each of the floating-point exception traps.
- From an Exception level using AArch32, the FPSCR.\{IDE, IXE, UFE, OFE, DZE, IOE\} bits enable each of the floating-point exception traps.

**ISS encoding for an SError interrupt**

![ISS encoding diagram]

**IDS, bit [24]**

IMPLEMENTATION DEFINED syndrome.

- 0b0  Bits [23:0] of the ISS field holds the fields described in this encoding.

**Note**

If FEAT_RAS is not implemented, bits [23:0] of the ISS field are RES0.

- 0b1  Bits [23:0] of the ISS field hold IMPLEMENTATION DEFINED syndrome information that can be used to provide additional information about the SError interrupt.
Note
This field was previously called ISV.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [23:14]
Reserved, RES0.

IESB, bit [13]
When FEAT_IESB is implemented and DFSC == 0b010001:
Implicit error synchronization event.
0b0  The SError interrupt was either not synchronized by the implicit error synchronization event or not taken immediately.
0b1  The SError interrupt was synchronized by the implicit error synchronization event and taken immediately.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

AET, bits [12:10]
When FEAT_RAS is implemented and DFSC == 0b010001:
Asynchronous Error Type.
Describes the PE error state after taking the SError interrupt exception.
0b000  Uncontainable (UC).
0b001  Unrecoverable state (UEU).
0b010  Restartable state (UEO).
0b011  Recoverable state (UER).
0b100  Corrected (CE).
All other values are reserved.
If multiple errors are taken as a single SError interrupt exception, the overall PE error state is reported.

Note
Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EA, bit [9]
When FEAT_RAS is implemented and DFSC == 0b010001:
External abort type. Provides an IMPLEMENTATION DEFINED classification of External aborts.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Otherweise:
Reserved, RES0.

Bits [8:6]
Reserved, RES0.

DFSC, bits [5:0]

When FEAT_RAS is implemented:
Data Fault Status Code.
0b000000 Uncategorized error.
0b010001 Asynchronous SError interrupt.
All other values are reserved.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Otherwise:
Reserved, RES0.

ISS encoding for an exception from a Breakpoint or Vector Catch debug exception

Bits [24:6]
Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.
0b100010 Debug exception.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions:
• For exceptions from AArch64, see Breakpoint exceptions on page D2-4707.
• For exceptions from AArch32, see Breakpoint exceptions on page G2-9054 and Vector Catch exceptions on page G2-9093.

ISS encoding for an exception from a Software Step exception

ISV, bit [24]
Instruction syndrome valid. Indicates whether the EX bit, ISS[6], is valid, as follows:
0b0 EX bit is RES0.
0b1 EX bit is valid.
See the EX bit bit description for more information.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [23:7]**

Reserved, RES0.

**EX, bit [6]**

Exclusive operation. If the ISV bit is set to 1, this bit indicates whether a Load-Exclusive instruction was stepped.

- 0b0 An instruction other than a Load-Exclusive instruction was stepped.
- 0b1 A Load-Exclusive instruction was stepped.

If the ISV bit is set to 0, this bit is RES0, indicating no syndrome data is available.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IFSC, bits [5:0]**

Instruction Fault Status Code.

- 0b100010 Debug exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see *Software Step exceptions on page D2-4742.*

**ISS encoding for an exception from a Watchpoint exception**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>WPT</td>
</tr>
<tr>
<td>22</td>
<td>WPTV</td>
</tr>
<tr>
<td>18-12</td>
<td>RES0</td>
</tr>
<tr>
<td>11-8</td>
<td>CM</td>
</tr>
<tr>
<td>7-3</td>
<td>DFSC</td>
</tr>
</tbody>
</table>

**Bit [24]**

Reserved, RES0.

**WPT, bits [23:18]**

*When FEAT_SME is implemented:*

Watchpoint number, 0 to 15 inclusive.

All other values are reserved.

*Otherwise:*

Reserved, RES0.

**WPTV, bit [17]**

*When FEAT_SME is implemented:*

Watchpoint number Valid.

- 0b0 The WPT field is invalid, and holds an UNKNOWN value.
- 0b1 The WPT field is valid, and holds the number of a watchpoint that triggered a Watchpoint exception.

When a Watchpoint exception is triggered by a watchpoint match:

- If the PE sets any of FnV, FnP, or WPF to 1, then the PE sets WPTV to 1.
If the PE sets all of FnV, FnP, and WPF to 0, then the PE sets WPTV to an IMPLEMENTATION DEFINED value, 0 or 1.

**Otherwise:**
Reserved, RES0.

**WPF, bit [16]**

*When FEAT_SME is implemented:*
Watchpoint might be false-positive.

- 0b0 The watchpoint matched the original access or set of contiguous accesses.
- 0b1 The watchpoint matched an access or set of contiguous accesses where the lowest accessed address was rounded down to the nearest multiple of 16 bytes and the highest accessed address was rounded up to the nearest multiple of 16 bytes minus 1, but the watchpoint might not have matched the original access or set of contiguous accesses.

**Otherwise:**
Reserved, RES0.

**FnP, bit [15]**

*When FEAT_SME is implemented:*
FAR not Precise.

This field only has meaning if the FAR is valid; that is, when the FnV field is 0. If the FnV field is 1, the FnP field is 0.

- 0b0 If the FnV field is 0, the FAR holds the virtual address of an access or set of contiguous accesses that triggered a Watchpoint exception.
- 0b1 The FAR holds any address within the smallest implemented translation granule that contains the virtual address of an access or set of contiguous accesses that triggered a Watchpoint exception.

**Otherwise:**
Reserved, RES0.

**Bit [14]**
Reserved, RES0.

**VNCR, bit [13]**

*When FEAT_NV2 is implemented:*
Indicates that the watchpoint came from use of VNCR_EL2 register by EL1 code.

- 0b0 The watchpoint was not generated by the use of VNCR_EL2 by EL1 code.
- 0b1 The watchpoint was generated by the use of VNCR_EL2 by EL1 code.

This field is 0 in ESR_EL1.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**Bits [12:11]**
Reserved, RES0.

**FnV, bit [10]**

*When FEAT_SME is implemented:*
FAR not Valid.

- 0b0 The FAR is valid, and its value is as described by the FnP field.
The FAR is invalid, and holds an **UNKNOWN** value.

**Otherwise:**

Reserved, **RES0**.

**Bit [9]**

Reserved, **RES0**.

**CM, bit [8]**

Cache maintenance. Indicates whether the Watchpoint exception came from a cache maintenance instruction:

- **0b0** The Watchpoint exception was not generated by the execution of one of the System instructions identified in the description of value 1.
- **0b1** The Watchpoint exception was generated by the execution of a cache maintenance instruction. The `DC ZVA`, `DC GVA`, and `DC GZVA` instructions are not classified as a cache maintenance instructions, and therefore their execution cannot cause this field to be set to 1.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Bit [7]**

Reserved, **RES0**.

**WnR, bit [6]**

Write not Read. Indicates whether the Watchpoint exception was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

- **0b0** Watchpoint exception caused by an instruction reading from a memory location.
- **0b1** Watchpoint exception caused by an instruction writing to a memory location.

For Watchpoint exceptions on cache maintenance instructions, this bit always returns a value of 1.

For Watchpoint exceptions from an atomic instruction, this field is set to 0 if a read of the location would have generated the Watchpoint exception, otherwise it is set to 1.

If multiple watchpoints match on the same access, it is **UNPREDICTABLE** which watchpoint generates the Watchpoint exception.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**DFSC, bits [5:0]**

Data Fault Status Code.

- **0b100010** Debug exception.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

For more information about generating these exceptions, see *Watchpoint exceptions* on page D2-4726.

**ISS encoding for an exception from execution of a Breakpoint instruction**

<table>
<thead>
<tr>
<th>Bit 24</th>
<th>Bit 16</th>
<th>Bit 15</th>
<th>Bit 0</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RES0</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [24:16]**

Reserved, **RES0**.
Comment, bits [15:0]

Set to the instruction comment field value, zero extended as necessary.

For the AArch32 BKPT instructions, the comment field is described as the immediate field.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see *Breakpoint Instruction exceptions on page D2-4705.*

**ISS encoding for an exception from an ERET, ERETA, or ERETAB instruction**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>RES0</td>
</tr>
<tr>
<td>23</td>
<td>ERET</td>
</tr>
<tr>
<td>22</td>
<td>ERETA</td>
</tr>
</tbody>
</table>

This EC value applies when FEAT_FGT is implemented, or when HCR_EL2.NV is 1.

**Bits [24:2]**

Reserved, RES0.

**ERET, bit [1]**

Indicates whether an ERET or ERETA* instruction was trapped to EL2.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ERET instruction trapped to EL2.</td>
</tr>
<tr>
<td>1</td>
<td>ERETA or ERETAB instruction trapped to EL2.</td>
</tr>
</tbody>
</table>

If this bit is 0, the ERETA field is RES0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ERETA, bit [0]**

Indicates whether an ERETAA or ERETAB instruction was trapped to EL2.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ERETAA instruction trapped to EL2.</td>
</tr>
<tr>
<td>1</td>
<td>ERETAB instruction trapped to EL2.</td>
</tr>
</tbody>
</table>

When the ERET field is 0, this bit is RES0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see HCR_EL2.NV.

If FEAT_FGT is implemented, HFGITR_EL2.ERET controls fine-grained trap exceptions from ERET, ERETAA and ERETAB execution.

**ISS encoding for an exception from a TSTART instruction**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>RES0</td>
</tr>
<tr>
<td>23</td>
<td>Rd</td>
</tr>
<tr>
<td>22</td>
<td>RES0</td>
</tr>
</tbody>
</table>

**Bits [24:10]**

Reserved, RES0.

**Rd, bits [9:5]**

The Rd value from the issued instruction, the general purpose register used for the destination.
D17.2 General system control registers

Bits [4:0]
Reserved, RES0.

**ISS encoding for an exception from Branch Target Identification instruction**

Bits [24:2]
Reserved, RES0.

**BTYPE, bits [1:0]**
This field is set to the PSTATE.BTYPE value that generated the Branch Target Exception.

For more information about generating these exceptions, see Chapter B1 *The AArch64 Application Level Programmers’ Model*.

**ISS encoding for an exception from a Pointer Authentication instruction when HCR_EL2.API == 0 || SCR_EL3.API == 0**

Bits [24:0]
Reserved, RES0.

For more information about generating these exceptions, see:

- **HCR_EL2.API**, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL2.
- **SCR_EL3.API**, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL3.

**ISS encoding for an exception from a Pointer Authentication instruction authentication failure**

Bits [24:2]
Reserved, RES0.

Bit [1]
This field indicates whether the exception is as a result of an Instruction key or a Data key.

- 0b0 Instruction Key.
- 0b1 Data Key.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [0]**

This field indicates whether the exception is as a result of an A key or a B key.
- 0b0 A key.
- 0b1 B key.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

The following instructions generate an exception when the Pointer Authentication Code (PAC) is incorrect:
- AUTIASP, AUTIAZ, AUTIA1716.
- AUTIBSP, AUTIBZ, AUTIB1716.
- AUTIA, AUTDA, AUTIB, AUTDB.
- AUTIZA, AUTIZB, AUTDZA, AUTDZB.

It is IMPLEMENTATION DEFINED whether the following instructions generate an exception directly from the authorization failure, rather than changing the address in a way that will generate a Translation fault when the address is accessed:
- RETAA, RETAB.
- BRAA, BRAB, BLRAA, BLRAB.
- BRAAZ, BRABZ, BLRAAZ, BLRABZ.
- ERETA, ERETAB.
- LDRAA, LDRAB, whether the authenticated address is written back to the base register or not.

**Accessing ESR_EL2**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic ESR_EL2 or ESR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ESR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL = EL0 then
  UNDEFINED;
elsif PSTATE.EL = EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = ESR_EL1;
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
endif
elsif PSTATE.EL = EL2 then
  X[t, 64] = ESR_EL2;
elsif PSTATE.EL = EL3 then
  X[t, 64] = ESR_EL2;
**MSR ESR_EL2, <Xt>**

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        ESR_EL1 = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE_EL == EL2 then
    ESR_EL2 = X[t, 64];
elsif PSTATE_EL == EL3 then
    ESR_EL2 = X[t, 64];
MRS <Xt>, ESR_EL1

**MRS <Xt>, ESR_EL1**

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.ESR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x138];
    else
        X[t, 64] = ESR_EL1;
elsif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' then
        X[t, 64] = ESR_EL2;
    else
        X[t, 64] = ESR_EL1;
elsif PSTATE_EL == EL3 then
    X[t, 64] = ESR_EL1;
MSR ESR_EL1, <Xt>

**MSR ESR_EL1, <Xt>**

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.ESR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x138];
    else
        X[t, 64] = ESR_EL1;
else
    if HCR_EL2.E2H == '1' then
        X[t, 64] = ESR_EL2;
    else
        X[t, 64] = ESR_EL1;
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    NVMem[0x138] = X[t, 64];
else
    ESR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        ESR_EL2 = X[t, 64];
    else
        ESR_EL1 = X[t, 64];
    endif
elsif PSTATE.EL == EL3 then
    ESR_EL1 = X[t, 64];
endif
D17.2.39  ESR_EL3, Exception Syndrome Register (EL3)

The ESR_EL3 characteristics are:

**Purpose**

Holds syndrome information for an exception taken to EL3.

**Configurations**

This register is present only when EL3 is implemented. Otherwise, direct accesses to ESR_EL3 are UNDEFINED.

**Attributes**

ESR_EL3 is a 64-bit register.

**Field descriptions**

ESR_EL3 is made UNKNOWN as a result of an exception return from EL3.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL3, the value of ESR_EL3 is UNKNOWN. The value written to ESR_EL3 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

**Bits [63:37]**

Reserved, RES0.

**ISS2, bits [36:32]**

*When FEAT_LS64 is implemented:*

When FEAT_LS64 is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort exception for a Translation fault, Access flag fault, or Permission fault, then this field holds register specifier, Xs.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort exception for a Translation fault, Access flag fault, or Permission fault, then this field holds register specifier, Xs.

Otherwise, this field is RES0.

**Otherwise:**

Reserved, RES0.

**EC, bits [31:26]**

Exception Class. Indicates the reason for the exception that this register holds information about.

For each EC value, the table references a subsection that gives information about:

- The cause of the exception, for example the configuration required to enable the trap.
- The encoding of the associated ISS.

Possible values of the EC field are:

**EC == 0b0000000**

Unknown reason.

See ISS encoding for exceptions with an unknown reason.
EC == 0b000001
Trapped WF* instruction execution.
Conditional WF* instructions that fail their condition code check do not cause an exception.
See ISS encoding for an exception from a WF* instruction.

EC == 0b000011
When AArch32 is supported:
Trapped MCR or MRC access with (coproc==0b1111) that is not reported using EC 0b000000.
See ISS encoding for an exception from an MCR or MRC access.

EC == 0b000100
When AArch32 is supported:
Trapped MCRR or MRRC access with (coproc==0b1111) that is not reported using EC 0b000000.
See ISS encoding for an exception from an MCRR or MRRC access.

EC == 0b000101
When AArch32 is supported:
Trapped MCR or MRC access with (coproc==0b1110).
See ISS encoding for an exception from an MCR or MRC access.

EC == 0b000110
When AArch32 is supported:
Trapped LDC or STC access.
The only architected uses of these instruction are:
- An STC to write data to memory from DBGDTRRXint.
- An LDC to read data from memory to DBGDTRTXint.
See ISS encoding for an exception from an LDC or STC instruction.

EC == 0b000111
Access to SME, SVE, Advanced SIMD or floating-point functionality trapped by CPACR_EL1.FPEN, CPTR_EL2.FPEN, CPTR_EL2.TFP, or CPTR_EL3.TFP control.
Excludes exceptions resulting from CPACR_EL1 when the value of HCR_EL2.TGE is 1, or because SVE or Advanced SIMD and floating-point are not implemented. These are reported with EC value 0b000000.
See ISS encoding for an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from the FPEN and TFP traps.

EC == 0b001001
When FEAT_PAuth is implemented:
Trapped use of a Pointer authentication instruction because HCR_EL2.API == 0 || SCR_EL3.API == 0.
See ISS encoding for an exception from a Pointer Authentication instruction when HCR_EL2.API == 0 || SCR_EL3.API == 0.

EC == 0b001010
When FEAT_LS64 is implemented:
An exception from an LD64B or ST64B* instruction.
See ISS encoding for an exception from an LD64B or ST64B* instruction.

EC == 0b001100
When AArch32 is supported:
Trapped MRRC access with (coproc==0b1110).
See ISS encoding for an exception from an MCRR or MRRC access.

EC == 0b001101
When FEAT_BTI is implemented:
Branch Target Exception.
See ISS encoding for an exception from Branch Target Identification instruction.

EC == 0b0001110
Illegal Execution state.
See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.

EC == 0b0100011
When AArch32 is supported:
SMC instruction execution in AArch32 state, when SMC is not disabled.
See ISS encoding for an exception from SMC instruction execution in AArch32 state.

EC == 0b0101010
When AArch64 is supported:
SMC instruction execution in AArch64 state.
See ISS encoding for an exception from SMC instruction execution in AArch64 state.

EC == 0b010110
When AArch64 is supported:
Trapped MSR, MRS or System instruction execution in AArch64 state, that is not reported using EC 0b000000, 0b000001 or 0b000111.
This includes all instructions that cause exceptions that are part of the encoding space defined in System instruction class encoding overview on page C5-655, except for those exceptions reported using EC values 0b000000, 0b000001, or 0b000111.
See ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state.

EC == 0b0101110
When AArch64 is supported:
Access to SVE functionality trapped as a result of CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ, that is not reported using EC 0b000000.
See ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ.

EC == 0b011000
When AArch64 is supported:
SVC instruction execution in AArch64 state.
See ISS encoding for an exception from SVC instruction execution in AArch64 state.

EC == 0b0110110
When AArch64 is supported:
HVC instruction execution in AArch64 state.
See ISS encoding for an exception from HVC instruction execution in AArch64 state.

EC == 0b0110111
When AArch64 is supported:
Exception from an access to a TSTART instruction at EL0 when SCTLR_EL1.TME == 0, EL0 when SCTLR_EL2.TME == 0, at EL1 when SCTLR_EL1.TME == 0, at EL2 when SCTLR_EL2.TME == 0 or at EL3 when SCTLR_EL3.TME == 0.
See ISS encoding for an exception from a TSTART instruction.

EC == 0b011100
When FEAT_FPAC is implemented:
Exception from a Pointer Authentication instruction authentication failure
See ISS encoding for an exception from a Pointer Authentication instruction authentication failure.

EC == 0b011101
When FEAT_SME is implemented:
Access to SME functionality trapped as a result of CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, CPTR_EL3.ESM, or an attempted execution of an instruction that is illegal because of the value of PSTATE.SM or PSTATE.ZA, that is not reported using EC 0b000000.

See ISS encoding for an exception due to SME functionality.

**EC == 0b011110**

*When FEAT_RME is implemented:*

Exception from a Granule Protection Check

See ISS encoding for an exception from a Granule Protection Check.

**EC == 0b011111**

IMPLEMENTATION DEFINED exception to EL3.

See ISS encoding for an IMPLEMENTATION DEFINED exception to EL3.

**EC == 0b100000**

Instruction Abort from a lower Exception level.

Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.

See ISS encoding for an exception from an Instruction Abort.

**EC == 0b100001**

Instruction Abort taken without a change in Exception level.

Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.

See ISS encoding for an exception from an Instruction Abort.

**EC == 0b100010**

PC alignment fault exception.

See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.

**EC == 0b100100**

Data Abort exception from a lower Exception level.

Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.

See ISS encoding for an exception from a Data Abort.

**EC == 0b100101**

Data Abort exception taken without a change in Exception level.

Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.

See ISS encoding for an exception from a Data Abort.

**EC == 0b100110**

SP alignment fault exception.

See ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.

**EC == 0b100111**

*When FEAT_MOPS is implemented:*

Memory Operation Exception.

See ISS encoding for an exception from the Memory Copy and Memory Set instructions.

**EC == 0b101100**

*When AArch64 is supported:*
Trapped floating-point exception taken from AArch64 state. This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.

See ISS encoding for an exception from a trapped floating-point exception.

**EC == 0b101111**

SError interrupt.

See ISS encoding for an SError interrupt.

**EC == 0b111100**

*When AArch64 is supported:*

BRK instruction execution in AArch64 state.

This is reported in ESR_EL3 only if a BRK instruction is executed in EL3. This is the only debug exception that can be taken to EL3 when EL3 is using AArch64.

See ISS encoding for an exception from execution of a Breakpoint instruction.

All other EC values are reserved by Arm, and:

- Unused values in the range 0b000000 - 0b101100 (0x00 - 0x2C) are reserved for future use for synchronous exceptions.
- Unused values in the range 0b101101 - 0b111111 (0x2D - 0x3F) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IL, bit [25]**

Instruction Length for synchronous exceptions. Possible values of this bit are:

0b0 16-bit instruction trapped.

0b1 32-bit instruction trapped. This value is also used when the exception is one of the following:

- An SError interrupt.
- An Instruction Abort exception.
- A PC alignment fault exception.
- An SP alignment fault exception.
- A Data Abort exception for which the value of the ISV bit is 0.
- An Illegal Execution state exception.
- Any debug exception except for Breakpoint instruction exceptions.
- An exception reported using EC value 0b000000.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ISS, bits [24:0]**

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

Typically, an ISS encoding has a number of subfields. When an ISS subfield holds a register number, the value returned in that field is the AArch64 view of the register number.

For an exception taken from AArch32 state, see *Mapping of the general-purpose registers between the Execution states* on page D1-4684.
If the AArch32 register descriptor is 0b1111, then:

- If the instruction that generated the exception was not UNPREDICTABLE, the field takes the value 0b11111.
- If the instruction that generated the exception was UNPREDICTABLE, the field takes an UNKNOWN value that must be either:
  - The AArch64 view of the register number of a register that might have been used at the Exception level from which the exception was taken.
  - The value 0b11111.

**ISS encoding for exceptions with an unknown reason**

![ISS encoding for exceptions with an unknown reason](image)

**Bits [24:0]**

Reserved, RES0.

When an exception is reported using this EC code the IL field is set to 1.

This EC code is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

- The attempted execution of an instruction bit pattern that has no allocated instruction or that is not accessible at the current Exception level and Security state, including:
  - A read access using a System register pattern that is not allocated for reads or that does not permit reads at the current Exception level and Security state.
  - A write access using a System register pattern that is not allocated for writes or that does not permit writes at the current Exception level and Security state.
  - Instruction encodings that are unallocated.
  - Instruction encodings for instructions or System registers that are not implemented in the implementation.

- In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.

- In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.

- In AArch32 state, attempted execution of a short vector floating-point instruction.

- In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.

- An exception generated because of the value of one of the SCTLR_EL1.{ITD, SED, CP15BEN} control bits.

- Attempted execution of:
  - An HVC instruction when disabled by HCR_EL2.HCD or SCR_EL3.HCE.
  - An SMC instruction when disabled by SCR_EL3.SMD.
  - An HLT instruction when disabled by EDSCR.HDE.

- Attempted execution of an MSR or MRS instruction to access SP_EL0 when the value of SPSel.SP is 0.

- Attempted execution of an MSR or MRS instruction using a _EL12 register name when HCR_EL2.E2H == 0.
• Attempted execution, in Debug state, of:
  — A DCPS1 instruction when the value of HCR_EL2.TGE is 1 and EL2 is disabled or not implemented in the current Security state.
  — A DCPS2 instruction from EL1 or EL0 when EL2 is disabled or not implemented in the current Security state.
  — A DCPS3 instruction when the value of EDSCR.SDD is 1, or when EL3 is not implemented.
• When EL3 is using AArch64, attempted execution from Secure EL1 of an SRS instruction using R13_mon.
• In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, or EL0 of an instruction that is configured to trap to EL3.
• In AArch32 state, the attempted execution of an MRS (banked register) or an MSR (banked register) instruction to SPSR_mon, SP_mon, or LR_mon.
• An exception that is taken to EL2 because the value of HCR_EL2.TGE is 1 that, if the value of HCR_EL2.TGE was 0 would have been reported with an ESR_ELx.EC value of 0b000111.
• In Non-transactional state, attempted execution of a TCOMMIT instruction.

**ISS encoding for an exception from a WF* instruction**

<table>
<thead>
<tr>
<th>CV</th>
<th>COND</th>
<th>RES0</th>
<th>RN</th>
<th>RES0</th>
<th>RV</th>
<th>TI</th>
</tr>
</thead>
</table>

**CV, bit [24]**

Condition code valid.

- 0b0: The COND field is not valid.
- 0b1: The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**COND, bits [23:20]**

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
• When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  — CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the
    SPSR.IT field to determine the condition, if any, of the T32 instruction.
  — CV is set to 1 and COND is set to the condition code for the condition that applied to
    the instruction.
• For an implementation that, for both A32 and T32 instructions, takes an exception on a
  trapped conditional instruction only if the instruction passes its condition code check, these
  definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND
  field is set to \(0b1110\), or to the value of any condition that applied to the instruction.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:10]
Reserved, RES0.

RN, bits [9:5]

**When FEAT_WFxT is implemented:**
Register Number. Indicates the register number supplied for a WFET or WFIT instruction.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

Bits [4:3]
Reserved, RES0.

RV, bit [2]

**When FEAT_WFxT is implemented:**
Register field Valid.
If TI[1] == 1, then this field indicates whether RN holds a valid register number for the register
argument to the trapped WFET or WFIT instruction.

\(0b0\) Register field invalid.
\(0b1\) Register field valid.

If TI[1] == 0, then this field is RES0.
This field is set to 1 on a trap on WFET or WFIT.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

TI, bits [1:0]

Trapped instruction. Possible values of this bit are:

\(0b00\) WFI trapped.
\(0b01\) WFE trapped.
\(0b10\) **When FEAT_WFxT is implemented:**
WFIT trapped.
\(0b11\) **When FEAT_WFxT is implemented:**
WFET trapped.

When FEAT_WFxT is implemented, this is a two bit field as shown. Otherwise, bit[1] is RES0.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating this exception:

- `SCTLR_EL1.{nTWE, nTWI}`.
- `HCR_EL2.{TWE, TWI}`.
- `SCR_EL3.{TWE, TWI}`.

**ISS encoding for an exception from an MCR or MRC access**

<table>
<thead>
<tr>
<th>CV</th>
<th>COND</th>
<th>Opc2</th>
<th>Opc1</th>
<th>CRn</th>
<th>Rt</th>
<th>CRm</th>
</tr>
</thead>
</table>

**CV, bit [24]**

- **0b0** The COND field is not valid.
- **0b1** The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**COND, bits [23:20]**

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Opc2, bits [19:17]**

The Opc2 value from the issued instruction.
For a trapped VMRS access, holds the value 0b000.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Opc1, bits [16:14]**

The Opc1 value from the issued instruction.
For a trapped VMRS access, holds the value 0b111.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**CRn, bits [13:10]**

The CRn value from the issued instruction.
For a trapped VMRS access, holds the reg field from the VMRS instruction encoding.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Rt, bits [9:5]**

The Rt value from the issued instruction, the general-purpose register used for the transfer.
If the Rt value is not 0b1111, then the reported value gives the AArch64 view of the register. Otherwise, if the Rt value is 0b1111:
- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value 0b11111.
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  — The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  — The value 0b11111.

See *Mapping of the general-purpose registers between the Execution states* on page D1-4684. The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**CRm, bits [4:1]**

The CRm value from the issued instruction.
For a trapped VMRS access, holds the value 0b0000.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Direction, bit [0]**

Indicates the direction of the trapped instruction.
- 0b0 Write to System register space. MCR instruction.
- 0b1 Read from System register space. MRC or VMRS instruction.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b000011:

- If **FEAT_TIDCP1** is implemented, **SCTLR_EL1.TIDCP**, for EL0 accesses to **IMPLEMENTATION DEFINED** functionality using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1.

- **CNTKCTL_EL1.EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN**, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1 or EL2.

- **PMUSERENR_EL0.ER, CR, SW, EN**, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1 or EL2.

- **AMUSERENR_EL0.EN**, for accesses to Activity Monitors registers from EL0 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1 or EL2.

- **HCR_EL1.TRVM, TVM**, for accesses to virtual memory control registers from EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

- **HCR_EL2.TTTLB**, for execution of TLB maintenance instructions at EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

- **HCR_EL2.TSW, TPC, TPU**, for execution of cache maintenance instructions at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

- **HCR_EL2.TACR**, for accesses to the Auxiliary Control Register at EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

- **HCR_EL2.TIDCP**, for accesses to lockdown, DMA, and TCM operations at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

- If **FEAT_TIDCP1** is implemented, **SCTLR_EL2.TIDCP**, for EL0 accesses to **IMPLEMENTATION DEFINED** functionality using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

- **HCR_EL2.TID1, TID2, TID3**, for accesses to ID registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

- **CPTR_EL2.TCPAC**, for accesses to **CPACR_EL1 or CPACR** using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

- **HSTR_EL2.T<n>**, for accesses to System registers using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

- **CNTHCTL_EL2.EL1PCEN**, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

- **MDCR_EL2.TPM, TPMCR**, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

- **CPTR_EL2.TAM**, for accesses to Activity Monitors registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.

- **MDCR_EL3.TPM**, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL3.

- **CPTR_EL3.TCPAC**, for accesses to **CPACR from EL1 and EL2**, and accesses to **HCPRTR from EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111)** trapped to EL3.

- **CPTR_EL3.TAM**, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL3.

- If **FEAT_FGT** is implemented, MCR or MRC access to some registers at EL0, trapped to EL2.
The following fields describe configuration settings for generating exceptions that are reported using EC value 0b000101:

- **CPACR_EL1.TTA** for accesses to trace registers, MCR or MRC access (coproc == 0b1110) trapped to EL1 or EL2.
- **MDSCR_EL1.TDCC**, for accesses to the Debug Communications Channel (DCC) registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL1 or EL2.
- If FEAT_FGT is implemented, **MDCR_EL2.TDCC** for accesses to the DCC registers at EL0 and EL1 trapped to EL2, and **MDCR_EL3.TDCC** for accesses to the DCC registers at EL0, EL1, and EL2 trapped to EL3.
- **HCR_EL2.TID0**, for accesses to the JIDR register in the ID group 0 at EL0 and EL1 using AArch32, MRC access (coproc == 0b1110) trapped to EL2.
- **CPTR_EL2.TTA**, for accesses to trace registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL2.
- **MDCR_EL2.TDRA**, for accesses to Debug ROM registers DBGDRAR and DBGDSAR using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL2.
- **MDCR_EL2.TDOSA**, for accesses to powerdown debug registers, using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL2.
- **MDCR_EL2.TDA**, for accesses to other debug registers, using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL2.
- **CPTR_EL3.TTA**, for accesses to trace registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.
- **MDCR_EL3.TDOSA**, for accesses to powerdown debug registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.
- **MDCR_EL3.TDA**, for accesses to other debug registers, using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b001000:

- **HCR_EL2.TID0**, for accesses to the FPSID register in ID group 0 at EL1 using AArch32 state, VMRS access trapped to EL2.
- **HCR_EL2.TID3**, for accesses to registers in ID group 3 including MVFR0, MVFR1 and MVFR2, VMRS access trapped to EL2.

**ISS encoding for an exception from an LD64B or ST64B* instruction**

![ISS encoding diagram]

**ISS, bits [24:0]**

- 0b00000000000000000000000000000000 *When FEAT_LS64_V is implemented:* ST64BV instruction trapped.
- 0b00000000000000000000000000000001 *When FEAT_LS64_ACCDATA is implemented:* ST64BV0 instruction trapped.
- 0b00000000000000000000000000000010 *When FEAT_LS64 is implemented:* LD64B or ST64B instruction trapped.

All other values are reserved.
ISS encoding for an exception from an MCRR or MRRC access

CV, bit [24]

Condition code valid.
- 0b0   The COND field is not valid.
- 0b1   The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.
For exceptions taken from AArch32:
- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.
The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.
For exceptions taken from AArch32:
- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc1, bits [19:16]

The Opc1 value from the issued instruction.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Bit [15]
Reserved, RES0.

Rt2, bits [14:10]
The Rt2 value from the issued instruction, the second general-purpose register used for the transfer. If the Rt2 value is not \(0b1111\), then the reported value gives the AArch64 view of the register. Otherwise, if the Rt2 value is \(0b1111\):
- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value \(0b1111\).
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value \(0b1111\).

See Mapping of the general-purpose registers between the Execution states on page D1-4684.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]
The Rt value from the issued instruction, the first general-purpose register used for the transfer. If the Rt value is not \(0b1111\), then the reported value gives the AArch64 view of the register. Otherwise, if the Rt value is \(0b1111\):
- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value \(0b1111\).
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value \(0b1111\).

See Mapping of the general-purpose registers between the Execution states on page D1-4684.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]
The CRm value from the issued instruction.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]
Indicates the direction of the trapped instruction.
\(0b0\) Write to System register space. MCRR instruction.
\(0b1\) Read from System register space. MRRC instruction.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating exceptions that are reported using EC value \(0b000100\):
- CNTKCTL_EL1.\{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN\}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == \(0b1111\)) trapped to EL1 or EL2.
• **PMUSERENR_EL0.**{CR, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.

• **AMUSERENR_EL0.**{EN}, for accesses to Activity Monitors registers AMEVCNTR0<:n> and AMEVCNTR1<:n> from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.

• **HCR_EL2.{TRVM, TVM},** for accesses to virtual memory control registers from EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.

• **HISTR_EL2.T<:n>,** for accesses to System registers using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.

• **CNSHCTL_EL2.{EL1PCEN, EL1PTEN},** for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.

• **MDCR_EL2.{TPM, TPMCR},** for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.

• **CPTR_EL2.TAM,** for accesses to Activity Monitors registers AMEVCNTR0<:n> and AMEVCNTR1<:n> from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.

• **MDCR_EL3.TPM,** for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL3.

• **CPTR_EL3.TAM,** for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL3.

• If **FEAT_FGT** is implemented, **HDFGRTR_EL2.PMCCNTR_EL0** for MRRC access and **HDFGWTR_EL2.PMCCNTR_EL0** for MCRR access to PMCCNTR at EL0, trapped to EL2.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b001100:

• **MDSCR_EL1.TDCC,** for accesses to the Debug ROM registers DBGDSAR and DBGDRAR at EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1110) trapped to EL1 or EL2.

• **MDCR_EL2.TDRA,** for accesses to Debug ROM registers DBGDRAR and DBGDSAR using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL2.

• **MDCR_EL3.TDA,** for accesses to debug registers, using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL3.

• **CPACR_EL1.TTA** for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL1 or EL2.

• **CPTR_EL2.TTA,** for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL2.

• **CPTR_EL3.TTA,** for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL3.

--- **Note** ---

If the Armv8-A architecture is implemented with an ETMv4 implementation, MCRR and MRRC accesses to trace registers are UNDEFINED and the resulting exception is higher priority than an exception due to these traps.

--- **ISS encoding for an exception from an LDC or STC instruction** ---

```
| 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| CV | COND | imm8 | RES0 | Rn | 5  | 4  | 3  | 2  | 1  | 0  | Offset | AM | Direction |
```

ARM DDI 04871.a
Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
ID081822
Non-Confidential
CV, bit [24]
Condition code valid.

0b0 The COND field is not valid.
0b1 The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:
- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]
For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:
- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

imm8, bits [19:12]
The immediate value from the issued instruction.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:10]
Reserved, RES0.

Rn, bits [9:5]
The Rn value from the issued instruction, the general-purpose register used for the transfer.
If the Rn value is not 0b1111, then the reported value gives the AArch64 view of the register. Otherwise, if the Rn value is 0b1111:

- If the instruction that generated the exception is not UNPREDICTABLE, then the register specifier takes the value 0b1111.
- If the instruction that generated the exception is UNPREDICTABLE, then the register specifier takes an UNKNOWN value, which is restricted to either:
  - The AArch64 view of one of the registers that could have been used in AArch32 state at the Exception level that the instruction was executed at.
  - The value 0b11111.

See *Mapping of the general-purpose registers between the Execution states* on page D1-4684.

This field is valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction. When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Offset, bit [4]**

Indicates whether the offset is added or subtracted:

- 0b0  Subtract offset.
- 0b1  Add offset.

This bit corresponds to the U bit in the instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**AM, bits [3:1]**

Addressing mode. The permitted values of this field are:

- 0b000  Immediate unindexed.
- 0b001  Immediate post-indexed.
- 0b010  Immediate offset.
- 0b011  Immediate pre-indexed.
- 0b100  For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.
- 0b110  For a trapped STC instruction, this encoding is reserved.

The values 0b101 and 0b111 are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in *Reserved values in System and memory-mapped registers and translation table entries* on page K1-11600.

Bit [2] in this subfield indicates the instruction form, immediate or literal.

Bits [1:0] in this subfield correspond to the bits \{P, W\} in the instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Direction, bit [0]**

Indicates the direction of the trapped instruction.

- 0b0  Write to memory. STC instruction.
- 0b1  Read from memory. LDC instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
The following fields describe the configuration settings for the traps that are reported using EC value 0b000110:

- **MDSCR_EL1.TDCC**, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint trapped to EL1 or EL2.
- **MDCR_EL2.TDA**, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint MCR or MRC access trapped to EL2.
- **MDCR_EL3.TDA**, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint MCR or MRC access trapped to EL3.
- If FEAT_FGT is implemented, **MDCR_EL2.TDCC** for LDC and STC accesses to the DCC registers at EL0 and EL1 trapped to EL2, and **MDCR_EL3.TDCC** for accesses to the DCC registers at EL0, EL1, and EL2 trapped to EL3.

**ISS encoding for an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from the FPEN and TFP traps**

| 28 | 23 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| CV | COND | RES0 |

The accesses covered by this trap include:

- Execution of SVE or Advanced SIMD and floating-point instructions.
- Accesses to the Advanced SIMD and floating-point System registers.
- Execution of SME instructions.

For an implementation that does not include either SVE or support for Advanced SIMD and floating-point, the exception is reported using the EC value 0b000000.

**CV, bit [24]**

Condition code valid.

- **0b0** The COND field is not valid.
- **0b1** The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**COND, bits [23:20]**

For exceptions taken from AArch64, this field is set to 0b1110.

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
• A conditional A32 instruction that is known to pass its condition code check can be presented either:
  — With COND set to 0b1110, the value for unconditional.
  — With the COND value held in the instruction.
• When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  — CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  — CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
• For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:0]
Reserved, RES0.

The following fields describe the configuration settings for the traps that are reported using EC value 0b000111:
• CPACR_EL1.FPEN, for accesses to SIMD and floating-point registers trapped to EL1.
• CPACR_EL2.FPEN and CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL2.
• CPTR_EL3.TFP, for accesses to SIMD and floating-point registers trapped to EL3.

**ISS encoding for an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ**

The accesses covered by this trap include:
• Execution of SVE instructions when the PE is not in Streaming SVE mode.
• Accesses to the SVE System registers, ZCR_ELx.

For an implementation that does not include SVE, the exception is reported using the EC value 0b000000.

Bits [24:0]
Reserved, RES0.

The following fields describe the configuration settings for the traps that are reported using EC value 0b011001:
• CPACR_EL1.ZEN, for execution of SVE instructions and accesses to SVE registers at EL0 or EL1, trapped to EL1.
• CPTR_EL2.ZEN and CPTR_EL2.TZ, for execution of SVE instructions and accesses to SVE registers at EL0, EL1, or EL2, trapped to EL2.
• CPTR_EL3.EZ, for execution of SVE instructions and accesses to SVE registers from all Exception levels, trapped to EL3.
**ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault**

![ISS encoding diagram](image)

**Bits [24:0]**

Reserved, RES0.

There are no configuration settings for generating Illegal Execution state exceptions and PC alignment fault exceptions. For more information about PC alignment fault exceptions, see [PC alignment checking on page D1-4668](#).

[SP alignment checking on page D1-4668](#) describes the configuration settings for generating SP alignment fault exceptions.

**ISS encoding for an exception from the Memory Copy and Memory Set instructions**

![ISS encoding diagram](image)

**MemInst, bit [24]**

Indicates the memory instruction class causing the exception.

- **0b0**: CPYFE*, CPYFM*, CPYE*, and CPYM* instructions.
- **0b1**: SETE*, SETM*, SETGE*, and SETGM* instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**isSETG, bit [23]**

Indicates whether the instruction belongs to SETGM* or SETGE* class of instruction.

- **0b0**: Not a SETGM* or SETGE* instruction.
- **0b1**: SETGM* or SETGE* instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Options, bits [22:19]**

Options : the Options field of the instruction.

For Memory Copy instructions, bits[22:19] forms the Options field, which holds the bits[15:12] of the instruction.

For Memory Set instructions:

- **Bits[22:21]** are RES0.
- **Bits[20:19]** form the Options field, which holds the bits[13:12] of the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
FromEpilogue, bit [18]
Indicates whether the instruction belongs to the epilogue class of Memory Copy or Memory Set instructions.

- 0b0: Not an epilogue instruction.
- 0b1: CPYE*, CPYFE*, SETE*, or SETGE* instruction.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WrongOption, bit [17]
Algorithm option.

- 0b0: WrongOption is false.
- 0b1: WrongOption is true.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OptionA, bit [16]
Algorithm type indicated by the PSTATE.C bit.

- 0b0: OptionB indicated by PSTATE.C is 0.
- 0b1: OptionA indicated by PSTATE.C is 1.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [15]
Reserved, RES0.

destreg, bits [14:10]
The destination register value from the issued instruction, containing the destination address.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

srcreg, bits [9:5]
The source register value from the issued instruction, containing either the source address or the source data.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

sizereg, bits [4:0]
The size register value from the issued instruction, containing the number of bytes to be transferred or set.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from HVC or SVC instruction execution

Bits [24:16]
Reserved, RES0.
**imm16, bits [15:0]**

The value of the immediate field from the HVC or SVC instruction.

For an HVC instruction, and for an A64 SVC instruction, this is the value of the imm16 field of the issued instruction.

For an A32 or T32 SVC instruction:

- If the instruction is unconditional, then:
  - For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.
  - For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- If the instruction is conditional, this field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

In AArch32 state, the HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

For T32 and A32 instructions, see SVC and HVC.

For A64 instructions, see SVC and HVC.

If FEAT_FGT is implemented, HFGITR_EL2.{SVC_EL1, SVC_EL0} control fine-grained traps on SVC execution.

**ISS encoding for an exception from SMC instruction execution in AArch32 state**

```
+---+---+---+---+
| CV| COND| 18| 16 |
+---+---+---+---+
| CCKNOWNPASS |
+---+---+---+---+
| RES0 |
```

For an SMC instruction that completes normally and generates an exception that is taken to EL3, the ISS encoding is RES0.

For an SMC instruction that is trapped to EL2 from EL1 because HCR_EL2.TSC is 1, the ISS encoding is as shown in the diagram.

**CV, bit [24]**

Condition code valid.

- 0b0 The COND field is not valid.
- 0b1 The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. See the description of the COND field for more information.

This field is valid only if CCKNOWNPASS is 1, otherwise it is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**COND, bits [23:20]**

For exceptions taken from AArch64, this field is set to 0b1110.
The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field is valid only if CCKNOWNPASS is 1, otherwise it is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CCKNOWNPASS, bit [19]

Indicates whether the instruction might have failed its condition code check.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The instruction was unconditional, or was conditional and passed its condition code check.</td>
</tr>
<tr>
<td>0b1</td>
<td>The instruction was conditional, and might have failed its condition code check.</td>
</tr>
</tbody>
</table>

--- Note ---

In an implementation in which an SMC instruction that fails its code check is not trapped, this field can always return the value 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [18:0]

Reserved, RES0.

HCR_EL2.TSC describes the configuration settings for trapping SMC instructions to EL2.

**ISS encoding for an exception from SMC instruction execution in AArch64 state**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
</tr>
</tbody>
</table>

Bits [24:16]

Reserved, RES0.
imm16, bits [15:0]
The value of the immediate field from the issued SMC instruction.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

The value of ISS[24:0] described here is used both:
• When an SMC instruction is trapped from EL1 modes.
• When an SMC instruction is not trapped, so completes normally and generates an exception that is taken to EL3.

HCR_EL2.TSC describes the configuration settings for trapping SMC from EL1 modes.

**ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state**

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24-22</td>
<td>Bits [24:22] Reserved, RES0.</td>
</tr>
<tr>
<td>21-20</td>
<td>Op0 value from the issued instruction. On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>19-17</td>
<td>Op2 value from the issued instruction. On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>16-14</td>
<td>Op1 value from the issued instruction. On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>13-10</td>
<td>CRn value from the issued instruction. On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>9</td>
<td>Rt value from the issued instruction, the general-purpose register used for the transfer. On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>4-1</td>
<td>CRm value from the issued instruction. On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
</tbody>
</table>
Direction, bit [0]

Indicates the direction of the trapped instruction.

- **0b0** Write access, including MSR instructions.
- **0b1** Read access, including MRS instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

For exceptions caused by System instructions, see *System instructions on page C4-539* for the encoding values returned by an instruction.

The following fields describe configuration settings for generating the exception that is reported using EC value 0b011000:

- If **FEAT_TIDCP1** is implemented, **SCTLR_EL1.TIDCP**, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch64 state, MSR or MRS access trapped to EL1.

- **SCTLR_EL1.UCI**, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.

- **SCTLR_EL1.UCT**, for accesses to **CTR_EL0** using AArch64 state, MSR or MRS access trapped to EL1 or EL2.

- **SCTLR_EL1.DZE**, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.

- **SCTLR_EL1.UMA**, for accesses to the PSTATE interrupt masks using AArch64 state, MSR or MRS access trapped to EL1 or EL2.

- **CPACR_EL1.TTA**, for accesses to the trace registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.

- **MDSCR_EL1.TDCC**, for accesses to the Debug Communications Channel (DCC) registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.

- If **FEAT_FGT** is implemented, **MDCR_EL2.TDCC** for accesses to the DCC registers at EL0 and EL1 trapped to EL2, and **MDCR_EL3.TDCC** for accesses to the DCC registers at EL0, EL1, and EL2 trapped to EL3.

- **CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}** accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.

- **PMUSERENR_EL0.{ER, CR, SW, EN}**, for accesses to the Performance Monitor registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.

- **AMUSERENR_EL0.EN**, for accesses to Activity Monitors registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.

- **HCR_EL2.{TRVM, TVM}**, for accesses to virtual memory control registers using AArch64 state, MSR or MRS access trapped to EL2.

- **HCR_EL2.TDZ**, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL2.

- **HCR_EL2.TTLB**, for execution of TLB maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.

- **HCR_EL2.{TSW, TPC, TPU}**, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.

- **HCR_EL2.TACR**, for accesses to the Auxiliary Control Register, **ACTLR_EL1**, using AArch64 state, MSR or MRS access trapped to EL2.
AArch64 System Register Descriptions

D17.2 General system control registers

- **HCR_EL2.TIDCP**, for accesses to lockdown, DMA, and TCM operations using AArch64 state, MSR or MRS access trapped to EL2.

- If **FEAT_TIDCP1** is implemented, **SCTLR_EL2.TIDCP**, for EL0 accesses to IMPLEMENTATION DEFINED functionality using AArch64 state, MSR or MRS access trapped to EL2.

- **HCR_EL2.{TID1, TID2, TID3}**, for accesses to ID group 1, ID group 2 or ID group 3 registers, using AArch64 state, MSR or MRS access trapped to EL2.

- **CPTR_EL2.TCPAC**, for accesses to **CPACR_EL1**, using AArch64 state, MSR or MRS access trapped to EL2.

- **CPTR_EL2.TTA**, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL2.

- **MDCR_EL2.TTRF**, for accesses to the trace filter control register, **TRFCR_EL1**, using AArch64 state, MSR or MRS access trapped to EL2.

- **MDCR_EL2.TDRA**, for accesses to Debug ROM registers, using AArch64 state, MSR or MRS access trapped to EL2.

- **MDCR_EL2.TDOSA**, for accesses to powerdown debug registers using AArch64 state, MSR or MRS access trapped to EL2.

- **CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}**, for accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2.

- **MDCR_EL2.TDA**, for accesses to debug registers using AArch64 state, MSR or MRS access trapped to EL2.

- **MDCR_EL2.{TPM, TPMCR}**, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL2.

- **CPTR_EL2.TAM**, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access trapped to EL2.

- **HCR_EL2.APK**, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL2.

- **HCR_EL2.{NV, NV1}**, for Nested virtualization register access, using AArch64 state, MSR or MRS access, trapped to EL2.

- **HCR_EL2.AT**, for execution of AT S1E* instructions, using AArch64 state, MSR or MRS access, trapped to EL2.

- **HCR_EL2.{TERR, FIEN}**, for accesses to RAS registers, using AArch64 state, MSR or MRS access, trapped to EL2.

- **SCR_EL3.APK**, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL3.

- **SCR_EL3.ST**, for accesses to the Counter-timer Physical Secure timer registers, using AArch64 state, MSR or MRS access trapped to EL3.

- **SCR_EL3.{TERR, FIEN}**, for accesses to RAS registers, using AArch64 state, MSR or MRS access trapped to EL3.

- **CPTR_EL3.TCPAC**, for accesses to **CPTR_EL2** and **CPACR_EL1** using AArch64 state, MSR or MRS access trapped to EL3.

- **CPTR_EL3.TTA**, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL3.

- **MDCR_EL3.TTRF**, for accesses to the trace filter control registers, **TRFCR_EL1** and **TRFCR_EL2**, using AArch64 state, MSR or MRS access trapped to EL3.
• MDCR_EL3.TDA, for accesses to debug registers, using AArch64 state, MSR or MRS access trapped to EL3.

• MDCR_EL3.TDOSA, for accesses to powerdown debug registers, using AArch64 state, MSR or MRS access trapped to EL3.

• MDCR_EL3.TPM, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL3.

• CPTR_EL3.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access, trapped to EL3.

• If FEAT_EVT is implemented, the following registers control traps for EL1 and EL0 Cache controls that use this EC value:
  — HCR_EL2.{TTLBOS, TTLBIS, TICAB, TOCU, TID4}.
  — HCR2.{TTLBIS, TICAB, TOCU, TID4}.

• If FEAT_FGT is implemented:
  — SCR_EL3.FGTEn, for accesses to the fine-grained trap registers, MSR or MRS access at EL2 trapped to EL3.
  — HFGTRR_EL2 for reads and HFGWTR_EL2 for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 trapped to EL2.
  — HFGITR_EL2 for execution of system instructions, MSR or MRS access trapped to EL2
  — HDFGRTR_EL2 for reads and HDFGWTR_EL2 for writes of registers, using AArch64 state, MSR or MRS access at EL0 and EL1 state trapped to EL2.
  — HAFGRTR_EL2 for reads of Activity Monitor counters, using AArch64 state, MRS access at EL0 and EL1 trapped to EL2.

• If FEAT_RNG_TRAP is implemented:
  — SCR_EL3.TRNDR for reads of RNDR and RNDRRS using AArch64 state, MRS access trapped to EL3.

• If FEAT_SME is implemented:
  — CPTR_EL3.ESM, for MSR or MRS accesses to SMPRI_EL1 at EL1, EL2, and EL3, trapped to EL3.
  — CPTR_EL3.ESM, for MSR or MRS accesses to SMPRIMAP_EL2 at EL2 and EL3, trapped to EL3.
  — SCTLR_EL1.EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, trapped to EL1 or EL2.
  — SCTLR_EL2. EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, trapped to EL2.
  — SCR_EL3. EnTP2, for MSR or MRS accesses to TPIDR2_EL0 at EL0, EL1, and EL2, trapped to EL3.

• If FEAT_NMI is implemented, HCRX_EL2.TALLINT, for MSR writes of ALLINT at EL1, trapped to EL2.

**ISS encoding for an IMPLEMENTATION DEFINED exception to EL3**

<table>
<thead>
<tr>
<th>24</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [24:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.
ISS encoding for an exception from an Instruction Abort

<table>
<thead>
<tr>
<th>BIT</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>24:13</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>12:11</td>
<td>SET, bits [12:11]</td>
</tr>
<tr>
<td>9</td>
<td>EA, bit [9]</td>
</tr>
<tr>
<td>8</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>5</td>
<td>IFSC</td>
</tr>
<tr>
<td>10</td>
<td>FnV, bit [10]</td>
</tr>
</tbody>
</table>

**SET, bits [12:11]**

*When FEAT_RAS is implemented:*

Synchronous Error Type. When IFSC is 0b010000, describes the PE error state after taking the Instruction Abort exception.

- 0b00: Recoverable state (UER).
- 0b10: Uncontainable (UC).
- 0b11: Restartable state (UEO).

All other values are reserved.

---

**Note**

Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in a PE state that is not recoverable.

This field is valid only if the IFSC code is 0b010000. It is RES0 for all other aborts. The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**FnV, bit [10]**

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

- 0b0: FAR is valid.
- 0b1: FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the IFSC code is 0b010000. It is RES0 for all other aborts. The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**EA, bit [9]**

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [8]**

Reserved, RES0.
S1PTW, bit [7]
For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:
0b0 Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1 Fault on the stage 2 translation of an access for a stage 1 translation table walk.
For any abort other than a stage 2 fault this bit is RES0.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [6]
Reserved, RES0.

IFSC, bits [5:0]
Instruction Fault Status Code.
0b000000 Address size fault, level 0 of translation or translation table base register.
0b000001 Address size fault, level 1.
0b000010 Address size fault, level 2.
0b000011 Address size fault, level 3.
0b000100 Translation fault, level 0.
0b000101 Translation fault, level 1.
0b000110 Translation fault, level 2.
0b000111 Translation fault, level 3.
0b001000 Access flag fault, level 0.
0b001001 Access flag fault, level 1.
0b001010 Access flag fault, level 2.
0b001011 Access flag fault, level 3.
0b001100 When FEAT_LPA2 is implemented:
Access flag fault, level 0.
0b001101 When FEAT_LPA2 is implemented:
Permission fault, level 0.
0b001110 Permission fault, level 1.
0b001111 Permission fault, level 2.
0b010000 Synchronous External abort, not on translation table walk or hardware update of translation table.
0b010001 When FEAT_LPA2 is implemented:
Synchronous External abort on translation table walk or hardware update of translation table, level -1.
0b010010 Synchronous External abort on translation table walk or hardware update of translation table, level 0.
0b010011 Synchronous External abort on translation table walk or hardware update of translation table, level 1.
0b010100 Synchronous External abort on translation table walk or hardware update of translation table, level 2.
0b010101 Synchronous External abort on translation table walk or hardware update of translation table, level 3.
0b010110 When FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access, not on translation table walk.
0b010111 When FEAT_LPA2 is implemented and FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.

0b011100  *When FEAT_RAS is not implemented:*
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.

0b011101  *When FEAT_RAS is not implemented:*
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.

0b011110  *When FEAT_RAS is not implemented:*
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.

0b011111  *When FEAT_RAS is not implemented:*
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.

0b100011  *When FEAT_RME is implemented and FEAT_LPA2 is implemented:*
Granule Protection Fault on translation table walk or hardware update of translation table, level -1.

0b100100  *When FEAT_RME is implemented:*
Granule Protection Fault on translation table walk or hardware update of translation table, level 0.

0b100101  *When FEAT_RME is implemented:*
Granule Protection Fault on translation table walk or hardware update of translation table, level 1.

0b100110  *When FEAT_RME is implemented:*
Granule Protection Fault on translation table walk or hardware update of translation table, level 2.

0b100111  *When FEAT_RME is implemented:*
Granule Protection Fault on translation table walk or hardware update of translation table, level 3.

0b101000  *When FEAT_RME is implemented:*
Granule Protection Fault, not on translation table walk or hardware update of translation table.

0b101001  *When FEAT_LPA2 is implemented:*
Address size fault, level -1.

0b101011  *When FEAT_LPA2 is implemented:*
Translation fault, level -1.

0b110000  TLB conflict abort.

0b110001  *When FEAT_HAFDBS is implemented:*
Unsupported atomic hardware update fault.

All other values are reserved.

For more information about the lookup level associated with a fault, see *The lookup level associated with MMU faults* on page D8-5190.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
ISS encoding for an exception due to SME functionality

The accesses covered by this trap include:

- Execution of SME instructions.
- Execution of SVE and Advanced SIMD instructions, when the PE is in Streaming SVE mode.
- Direct accesses of SVCR, SMCR_EL1, SMCR_EL2, SMCR_EL3.

Bits [24:3]

Reserved, RES0.

SMTC, bits [2:0]

SME Trap Code. Identifies the reason for instruction trapping.

- 0b000   Access to SME functionality trapped as a result of CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, or CPTR_EL3.ESM, that is not reported using EC 0b000000.
- 0b001   Advanced SIMD, SVE, or SVE2 instruction trapped because PSTATE.SM is 1.
- 0b010   SME instruction trapped because PSTATE.SM is 0.
- 0b011   SME instruction trapped because PSTATE.ZA is 0.

All other values are reserved.

The following fields describe the configuration settings for the traps that are reported using the EC value 0b011101:

- CPACR_EL1.SMEN, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR and SMCR_EL1 System registers at EL1 and EL0, trapped to EL1 or EL2.
- CPTR_EL2.SMEN and CPTR_EL2.TSM, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR, SMCR_EL1, SMCR_EL2 at EL2, EL1, or EL0, trapped to EL2.
- CPTR_EL3.ESM, for execution of SME instructions, SVE instructions when the PE is in Streaming SVE mode, and instructions that directly access SVCR, SMCR_EL1, SMCR_EL2, SMCR_EL3 from all Exception levels and any Security state, trapped to EL3.

ISS encoding for an exception from a Granule Protection Check

Bits [24:22]

Reserved, RES0.

S2PTW, bit [21]

Indicates whether the Granule Protection Check exception was on an access made for a stage 2 translation table walk.

- 0  Fault not on a stage 2 translation table walk.
- 1  Fault on a stage 2 translation table walk.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**InD, bit [20]**

Indicates whether the Granule Protection Check exception was on an instruction or data access.
0b0 Data access.
0b1 Instruction access.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**GPCSC, bits [19:14]**

Granule Protection Check Status Code.
0b000000 GPT address size fault at level 0.
0b000100 GPT walk fault at level 0.
0b000101 GPT walk fault at level 1.
0b001100 Granule protection fault at level 0.
0b001101 Granule protection fault at level 1.
0b010100 Synchronous External abort on GPT fetch at level 0.
0b010101 Synchronous External abort on GPT fetch at level 1.
All other values are reserved.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**VNCR, bit [13]**

*When FEAT_NV2 is implemented:*

Indicates that the fault came from use of VNCb_EL2 register by EL1 code.
0b0 The fault was not generated by the use of VNCb_EL2, by an MRS or MSR instruction executed at EL1.
0b1 The fault was generated by the use of VNCb_EL2, by an MRS or MSR instruction executed at EL1.
This field is 0 in ESR_EL1.
When InD is '1', this field is RES0.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

**Bits [12:9]**

Reserved, RES0.

**CM, bit [8]**

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:
0b0 The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.
0b1 The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA, DC GVA, and DC GZVA instructions are not classified as cache maintenance instructions, and therefore their execution cannot cause this field to be set to 1.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**S1PTW, bit [7]**

Indicates whether the Granule Protection Check exception was on an access for stage 2 translation for a stage 1 translation table walk:

- 0b0 Fault not on a stage 2 translation for a stage 1 translation table walk.
- 0b1 Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**WnR, bit [6]**

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

- 0b0 Abort caused by an instruction reading from a memory location.
- 0b1 Abort caused by an instruction writing to a memory location.

When InD is '1', this field is RES0.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

This field is UNKNOWN for:
• An External abort on an Atomic access.
• A fault reported using a DFSC value of 0b110101 or 0b110001, indicating an unsupported Exclusive or atomic access.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**xFSC, bits [5:0]**

Instruction or Data Fault Status Code.

- 0b100011 *When FEAT_RME is implemented and FEAT_LPA2 is implemented:* Granule Protection Fault on translation table walk or hardware update of translation table, level -1.
- 0b100100 *When FEAT_RME is implemented:* Granule Protection Fault on translation table walk or hardware update of translation table, level 0.
- 0b100101 *When FEAT_RME is implemented:* Granule Protection Fault on translation table walk or hardware update of translation table, level 1.
- 0b100110 *When FEAT_RME is implemented:* Granule Protection Fault on translation table walk or hardware update of translation table, level 2.
- 0b100111 *When FEAT_RME is implemented:* Granule Protection Fault on translation table walk or hardware update of translation table, level 3.
- 0b101000 *When FEAT_RME is implemented:*
Granule Protection Fault, not on translation table walk or hardware update of translation table.

All other values are reserved.

For more information about the lookup level associated with a fault, see *The lookup level associated with MMU faults on page D8-5190.*

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**ISS encoding for an exception from a Data Abort**

![ISS encoding diagram](image)

When **FEAT_LS64_V** is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this ISS encoding includes ISS2, bits[36:32].

When **FEAT_LS64_ACCDATA** is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this ISS encoding includes ISS2, bits[36:32].

**ISV, bit [24]**

Instruction Syndrome Valid. Indicates whether the syndrome information in ISS[23:14] is valid.

- **0b0** No valid instruction syndrome. ISS[23:14] are **RES0**.
- **0b1** ISS[23:14] hold a valid instruction syndrome.

In ESR_EL2, ISV is 1 when **FEAT_LS64** is implemented and a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

In ESR_EL2, ISV is 1 when **FEAT_LS64_V** is implemented and a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

In ESR_EL2, ISV is 1 when **FEAT_LS64_ACCDATA** is implemented and a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

For other faults reported in ESR_EL2, ISV is 0 except for the following stage 2 aborts:

- **AArch64** loads and stores of a single general-purpose register (including the register specified with **0b111111**, including those with Acquire/Release semantics, but excluding Load Exclusive or Store Exclusive and excluding those with writeback).
- **AArch32** instructions where the instruction:
  - Is an LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT instruction.
  - Is not performing register writeback.
  - Is not using R15 as a source or destination register.

For these stage 2 aborts, ISV is **UNKNOWN** if the exception was generated in Debug state in memory access mode, and otherwise indicates whether ISS[23:14] hold a valid syndrome.
For faults reported in ESR_EL1 or ESR_EL3, ISV is 1 when FEAT_LS64 is implemented and a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

For faults reported in ESR_EL1 or ESR_EL3, ISV is 1 when FEAT_LS64_V is implemented and a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

For faults reported in ESR_EL1 or ESR_EL3, ISV is 1 when FEAT_LS64_ACCDATA is implemented and a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault.

When FEAT_RAS is implemented, ISV is 0 for any synchronous External abort.

When FEAT_RAS is not implemented, it is IMPLEMENTATION DEFINED whether ISV is set to 1 or 0 on a synchronous External abort on a stage 2 translation table walk.

For ISS reporting, a stage 2 abort on a stage 1 translation table walk does not return a valid instruction syndrome, and therefore ISV is 0 for these aborts.

When FEAT_MOPS is implemented, for a synchronous Data Abort on a Memory Copy and Memory Set instruction, ISV is 0.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**SAS, bits [23:22]**

*When ISV == 1:*

Syndrome Access Size. Indicates the size of the access attempted by the faulting operation.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>Byte</td>
</tr>
<tr>
<td>0b01</td>
<td>Halfword</td>
</tr>
<tr>
<td>0b10</td>
<td>Word</td>
</tr>
<tr>
<td>0b11</td>
<td>Doubleword</td>
</tr>
</tbody>
</table>

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0b11.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0b11.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0b11.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

**SSE, bit [21]**

*When ISV == 1:*

Syndrome Sign Extend. For a byte, halfword, or word load operation, indicates whether the data item must be sign extended.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Sign-extension not required.</td>
</tr>
<tr>
<td>0b1</td>
<td>Data item must be sign-extended.</td>
</tr>
</tbody>
</table>

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.
When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

For all other operations, this field is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

### SRT, bits [20:16]

**When ISV == 1:**

Syndrome Register Transfer. The register number of the Wu/Xt/Rt operand of the faulting instruction.

If the exception was taken from an Exception level that is using AArch32, then this is the AArch64 view of the register. See [Mapping of the general-purpose registers between the Execution states](#) on page D1-4684.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

### SF, bit [15]

**When ISV == 1:**

SF

Sixty Four bit general-purpose register transfer. Width of the register accessed by the instruction is 64-bit.

0b0  Instruction loads/stores a 32-bit general-purpose register.

0b1  Instruction loads/stores a 64-bit general-purpose register.

--- **Note** ---

This field specifies the register width identified by the instruction, not the Execution state.
When ISV == 0 and FEAT_SME is implemented:

FnP
- FAR not Precise.
- 0b0 The FAR holds the faulting virtual address that generated the Data Abort.
- 0b1 The FAR holds any virtual address within the naturally-aligned granule that contains the faulting virtual address that generated a Data Abort due to an SVE contiguous vector load/store instruction in Streaming SVE mode, or an SME load/store instruction.

For more information about the naturally-aligned fault granule, see FAR_ELx (for example, FAR_EL1).

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
- Reserved, RES0.

AR, bit [14]

When ISV == 1:
- Acquire/Release.
- 0b0 Instruction did not have acquire/release semantics.
- 0b1 Instruction did have acquire/release semantics.

When FEAT_LS64 is implemented, if a memory access generated by an LD64B or ST64B instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_V is implemented, if a memory access generated by an ST64BV instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

When FEAT_LS64_ACCDATA is implemented, if a memory access generated by an ST64BV0 instruction generates a Data Abort for a Translation fault, Access flag fault, or Permission fault, then this field is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
- Reserved, RES0.

VNCR, bit [13]

When FEAT_NV2 is implemented:
- Indicates that the fault came from use of VNCR_EL2 register by EL1 code.
- 0b0 The fault was not generated by the use of VNCR_EL2, by an MRS or MSR instruction executed at EL1.
- 0b1 The fault was generated by the use of VNCR_EL2, by an MRS or MSR instruction executed at EL1.

This field is 0 in ESR_EL1.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
- Reserved, RES0.
LST, bits [12:11]

When \( DFSC == 0b00xxxx \) \&\& \( DFSC == 0b101011 \) \&\& \( DFSC != 0b0000xx \):

**LST**
Load/Store Type. Used when a Translation fault, Access flag fault, or Permission fault generates a Data Abort.

- **0b0**: The instruction that generated the Data Abort is not specified.
- **0b1**: When FEAT_LS64_V is implemented:
  An ST64BV instruction generated the Data Abort.
- **0b10**: When FEAT_LS64 is implemented:
  An LD64B or ST64B instruction generated the Data Abort.
- **0b11**: When FEAT_LS64_ACCDATA is implemented:
  An ST64BV0 instruction generated the Data Abort.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_RAS is implemented and \( DFSC == 0b010000 \):

**SET**
Synchronous Error Type. Used when a Synchronous External abort, not on a Translation table walk or hardware update of the Translation table, generated the Data Abort. Describes the PE error state after taking the Data Abort exception.

- **0b0**: Recoverable state (UER).
- **0b10**: Uncontainable (UC).
- **0b11**: Restartable state (UEO).

All other values are reserved.

---

**Note**

Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in a PE state that is not recoverable.

---

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, **RES0**.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

- **0b**: FAR is valid.
- **0b1**: FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the DFSC code is **0b010000**. It is **RES0** for all other aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
CM, bit [8]

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

0b0  The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.
0b1  The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA, DC GVA, and DC GZVA instructions are not classified as cache maintenance instructions, and therefore their execution cannot cause this field to be set to 1.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

0b0  Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1  Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

0b0  Abort caused by an instruction reading from a memory location.
0b1  Abort caused by an instruction writing to a memory location.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

This field is UNKNOWN for:
- An External abort on an Atomic access.
- A fault reported using a DFSC value of 0b110101 or 0b110001, indicating an unsupported Exclusive or atomic access.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code.

0b000000  Address size fault, level 0 of translation or translation table base register.
0b000001  Address size fault, level 1.
0b000010  Address size fault, level 2.
0b000011  Address size fault, level 3.
0b000100  Translation fault, level 0.
0b000101  Translation fault, level 1.
0b000110  Translation fault, level 2.
Translation fault, level 3.
Access flag fault, level 1.
Access flag fault, level 2.
Access flag fault, level 3.
When FEAT_LPA2 is implemented:
Access flag fault, level 0.
When FEAT_LPA2 is implemented:
Permission fault, level 0.
Permission fault, level 1.
Permission fault, level 2.
Permission fault, level 3.
Synchronous External abort, not on translation table walk or hardware update of translation table.
When FEAT_MTE2 is implemented:
Synchronous Tag Check Fault.
When FEAT_LPA2 is implemented:
Synchronous External abort on translation table walk or hardware update of translation table, level -1.
Synchronous External abort on translation table walk or hardware update of translation table, level 0.
Synchronous External abort on translation table walk or hardware update of translation table, level 1.
Synchronous External abort on translation table walk or hardware update of translation table, level 2.
Synchronous External abort on translation table walk or hardware update of translation table, level 3.
When FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access, not on translation table walk.
When FEAT_LPA2 is implemented and FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.
When FEAT_RME is implemented and FEAT_LPA2 is implemented:
Granule Protection Fault on translation table walk or hardware update of translation table, level -1.
When FEAT_RME is implemented:
Granule Protection Fault on translation table walk or hardware update of translation table, level 0.

0b100101  *When FEAT_RME is implemented:*
Granule Protection Fault on translation table walk or hardware update of translation table, level 1.

0b100110  *When FEAT_RME is implemented:*
Granule Protection Fault on translation table walk or hardware update of translation table, level 2.

0b100111  *When FEAT_RME is implemented:*
Granule Protection Fault on translation table walk or hardware update of translation table, level 3.

0b101000  *When FEAT_RME is implemented:*
Granule Protection Fault, not on translation table walk or hardware update of translation table.

0b101010  *When FEAT_LPA2 is implemented:*
Address size fault, level -1.

0b101011  *When FEAT_LPA2 is implemented:*
Translation fault, level -1.

0b110000  TLB conflict abort.

0b110001  *When FEAT_HAFDBS is implemented:*
Unsupported atomic hardware update fault.

0b110100  IMPLEMENTATION DEFINED fault (Lockdown).

0b110101  IMPLEMENTATION DEFINED fault (Unsupported Exclusive or Atomic access).

All other values are reserved.

For more information about the lookup level associated with a fault, see *The lookup level associated with MMU faults* on page D8-5190.

If the S1PTW bit is set, then the level refers the level of the stage 2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**ISS encoding for an exception from a trapped floating-point exception**

---

**Bit [24]**

Reserved, RES0.

**TFV, bit [23]**

Trapped Fault Valid bit. Indicates whether the IDF, IXF, UFF, OFF, DZF, and IOF bits hold valid information about trapped floating-point exceptions.

0b0  The IDF, IXF, UFF, OFF, DZF, and IOF bits do not hold valid information about trapped floating-point exceptions and are **UNKNOWN**.
One or more floating-point exceptions occurred during an operation performed while executing the reported instruction. The IDF, IXF, UFF, OFF, DZF, and IOF bits indicate trapped floating-point exceptions that occurred. For more information, see Floating-point exceptions and exception traps on page A1-64.

It is IMPLEMENTATION DEFINED whether this field is set to 0 on an exception generated by a trapped floating-point exception from an instruction that is performing floating-point operations on more than one lane of a vector.

--- Note ---
This is not a requirement. Implementations can set this field to 1 on a trapped floating-point exception from an instruction and return valid information in the {IDF, IXF, UFF, OFF, DZF, IOF} fields.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [22:11]
Reserved, RES0.

VECITR, bits [10:8]
For a trapped floating-point exception from an instruction executed in AArch32 state this field is RES1.
For a trapped floating-point exception from an instruction executed in AArch64 state this field is UNKNOWN.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

IDF, bit [7]
Input Denormal floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:
0b0   Input denormal floating-point exception has not occurred.
0b1   Input denormal floating-point exception occurred during execution of the reported instruction.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]
Reserved, RES0.

IXF, bit [4]
Inexact floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:
0b0   Inexact floating-point exception has not occurred.
0b1   Inexact floating-point exception occurred during execution of the reported instruction.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

UFF, bit [3]
Underflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:
0b0   Underflow floating-point exception has not occurred.
0b1   Underflow floating-point exception occurred during execution of the reported instruction.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**OFF, bit [2]**

Overflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:
- 0b0 Overflow floating-point exception has not occurred.
- 0b1 Overflow floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**DZF, bit [1]**

Divide by Zero floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:
- 0b0 Divide by Zero floating-point exception has not occurred.
- 0b1 Divide by Zero floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IOF, bit [0]**

Invalid Operation floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:
- 0b0 Invalid Operation floating-point exception has not occurred.
- 0b1 Invalid Operation floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

In an implementation that supports the trapping of floating-point exceptions:
• From an Exception level using AArch64, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.
• From an Exception level using AArch32, the FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.

**ISS encoding for an SError interrupt**

<table>
<thead>
<tr>
<th>24</th>
<th>23</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDS</td>
<td>RES0</td>
<td>AET</td>
<td>EA</td>
<td>RES0</td>
<td>DFSC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**IDS, bit [24]**

IMPLEMENTATION DEFINED syndrome.
- 0b0 Bits [23:0] of the ISS field holds the fields described in this encoding.

**Note**
If FEAT_RAS is not implemented, bits [23:0] of the ISS field are RES0.

- 0b1 Bits [23:0] of the ISS field holds IMPLEMENTATION DEFINED syndrome information that can be used to provide additional information about the SError interrupt.
Note
This field was previously called ISV.

Bits [23:14]
Reserved, RES0.

IESB, bit [13]
When FEAT_IESB is implemented and DFSC == 0b10001:
Implicit error synchronization event.
0b0 The SError interrupt was either not synchronized by the implicit error synchronization event or not taken immediately.
0b1 The SError interrupt was synchronized by the implicit error synchronization event and taken immediately.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

AET, bits [12:10]
When FEAT_RAS is implemented and DFSC == 0b10001:
Asynchronous Error Type.
Describes the PE error state after taking the SError interrupt exception.
0b000 Uncontainable (UC).
0b001 Unrecoverable state (UEU).
0b010 Restartable state (UEO).
0b011 Recoverable state (UER).
0b110 Corrected (CE).
All other values are reserved.
If multiple errors are taken as a single SError interrupt exception, the overall PE error state is reported.

Note
Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EA, bit [9]
When FEAT_RAS is implemented and DFSC == 0b10001:
External abort type. Provides an IMPLEMENTATION DEFINED classification of External aborts.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Otherwise:
Reserved, RES0.

Bits [8:6]
Reserved, RES0.

DFSC, bits [5:0]

When FEAT_RAS is implemented:
Data Fault Status Code.

0b000000 Uncategorized error.
0b010001 Asynchronous SError interrupt.
All other values are reserved.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

ISS encoding for an exception from execution of a Breakpoint instruction

Bits [24:16]
Reserved, RES0.

Comment, bits [15:0]
Set to the instruction comment field value, zero extended as necessary.
For the AArch32 BKPT instructions, the comment field is described as the immediate field.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see Breakpoint Instruction exceptions on page D2-4705.

ISS encoding for an exception from a TSTART instruction

Bits [24:10]
Reserved, RES0.

Rd, bits [9:5]
The Rd value from the issued instruction, the general purpose register used for the destination.

Bits [4:0]
Reserved, RES0.
ISS encoding for an exception from Branch Target Identification instruction

Bits [24:2]
Reserved, RES0.

BTYPE, bits [1:0]
This field is set to the PSTATE.BTYPE value that generated the Branch Target Exception.

For more information about generating these exceptions, see Chapter B1 *The AArch64 Application Level Programmers’ Model*.

ISS encoding for an exception from a Pointer Authentication instruction when HCR_EL2.API == 0 || SCR_EL3.API == 0

Bits [24:0]
Reserved, RES0.

For more information about generating these exceptions, see:
- HCR_EL2.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL2.
- SCR_EL3.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL3.

ISS encoding for an exception from a Pointer Authentication instruction authentication failure

Bits [24:2]
Reserved, RES0.

Bit [1]
This field indicates whether the exception is as a result of an Instruction key or a Data key.
0b0 Instruction Key.
0b1 Data Key.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Bit [0]

This field indicates whether the exception is as a result of an A key or a B key.

0b0  A key.
0b1  B key.

The reset behavior of this field is:

•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

The following instructions generate an exception when the Pointer Authentication Code (PAC) is incorrect:

•  AUTIASP, AUTIAZ, AUTIA1716.
•  AUTIBSP, AUTIBZ, AUTIB1716.
•  AUTIA, AUTDA, AUTIB, AUTDB.
•  AUTIZA, AUTIZB, AUTDZA, AUTOZB.

It is IMPLEMENTATION DEFINED whether the following instructions generate an exception directly from the authorization failure, rather than changing the address in a way that will generate a Translation fault when the address is accessed:

•  RETAA, RETAB.
•  BRAA, BRAB, BLRAA, BLRAB.
•  BRAAZ, BRABZ, BLRAAZ, BLRABZ.
•  ERETTA, ERETAB.
•  LDRAA, LDRAB, whether the authenticated address is written back to the base register or not.

**Accessing ESR_EL3**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS } <Xt>, \text{ ESR_EL3}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b10</td>
<td>0b0101</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  UNDEFINED;
elsif PSTATE_EL == EL2 then
  UNDEFINED;
elsif PSTATE_EL == EL3 then
  \( X[t, 64] = \text{ESR_EL3}; \)
MSR ESR_EL3, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b10</td>
<td>0b0101</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  ESR_EL3 = X[t, 64];
D17.2.40 FAR_EL1, Fault Address Register (EL1)

The FAR_EL1 characteristics are:

**Purpose**
Holds the faulting Virtual Address for all synchronous Instruction Abort exceptions, Data Abort exceptions, PC alignment fault exceptions and Watchpoint exceptions that are taken to EL1.

**Configurations**
AArch64 System register FAR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DFAR[31:0] (DFAR_NS).
AArch64 System register FAR_EL1 bits [63:32] are architecturally mapped to AArch32 System register IFAR[31:0] (IFAR_NS).

**Attributes**
FAR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Faulting Virtual Address for synchronous exceptions taken to EL1</th>
</tr>
</thead>
</table>

Faultrng Virtual Address for synchronous exceptions taken to EL1. Exceptions that set the FAR_EL1 are Instruction Aborts (EC 0x20 or 0x21), Data Aborts (EC 0x24 or 0x25), PC alignment faults (EC 0x22), and Watchpoints (EC 0x34 or 0x35). ESR_EL1.EC holds the EC value for the exception.

For a synchronous External abort, if the VA that generated the abort was from an address range for which TCR_ELx.TBI{<0|1>} == 1 for the translation regime in use when the abort was generated, then the top eight bits of FAR_EL1 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL1.FnV is 0, and FAR_EL1 is UNKNOWN if ESR_EL1.FnV is 1.

If a memory fault that sets FAR_EL1, other than a Tag Check Fault, is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

On an exception due to a Tag Check Fault caused by a data cache maintenance or other DC instruction, the address held in FAR_EL1 is IMPLEMENTATION DEFINED as one of the following:

- The lowest address that gave rise to the fault.
- The address specified in the register argument of the instruction as generated by MMU faults caused by DC ZVA.

If the exception that updates FAR_EL1 is taken from an Exception level using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_EL1 are 0x00000001:

- The faulting address was generated by a load or store instruction that sequentially incremented from address 0xFFFFFFFF. Such a load or store instruction is CONSTRAINED UNPREDICTABLE.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.
When FEAT_SME is implemented, and the PE sets ESR_EL1.ISV to 0 and ESR_EL1.FnP to 1 on taking a Data Abort exception or Watchpoint exception, the PE sets FAR_EL1 to any address within the naturally-aligned fault granule that contains the virtual address of the memory access that generated the Data Abort or Watchpoint exception.

The naturally-aligned fault granule is one of:

- When ESR_EL1.DFSC is 0b010001, indicating a Synchronous Tag Check fault, it is a 16-byte tag granule.
- When ESR_EL1.DFSC is 0b11010x, indicating an IMPLEMENTATION DEFINED fault, it is an IMPLEMENTATION DEFINED granule.
- Otherwise, it is the smallest implemented translation granule.

When FEAT_MOPS is implemented, the value in FAR_EL1 on a synchronous exception from any of the Memory Copy and Memory Set instructions represents the first element that has not been copied or set, and is determined as follows:

- For a Data Abort generated by the MMU, the value is within the address range of the relevant translation granule, aligned to the size of the relevant translation granule of the address that generated the Data Abort. Bits[(n-1):0] of the value are UNKNOWN, where 2^n is the relevant translation granule size in bytes. For the purpose of calculating the relevant translation granule, if the MMU is disabled for a stage of translation, then the current translation granule size is equal to 2^64 for stage 1, and the PARange for stage 2. The relevant translation granule is:
  — For MMU faults generated at stage 1, the current stage 1 translation granule.
  — For MMU faults generated at stage 2, the smaller of the current stage 1 translation granule and the current stage 2 translation granule.
  — If FEAT_RME is implemented, for a synchronous data abort generated as the result of a GPF, the smallest of the current stage 1 translation granule, the current stage 2 translation granule and the configured granule size in GPCCR_EL3.PGS.
- For a Data Abort generated by a Tag Check failure, the value is the lowest address that failed the Tag Check within the block size of the load or store.
- For a Watchpoint exception, the value is an address range of the size defined by the DCZID_EL0.BS field. This address does not need to be the element with a watchpoint, but can be some earlier element.
- Otherwise, the value is the lowest address in the block size of the load or store.

For a Data Abort or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see Address tagging on page D8-5162.

For a synchronous Tag Check Fault abort, bits[63:60] are UNKNOWN.

Execution at EL0 makes FAR_EL1 become UNKNOWN.

--- Note ---

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or data abort. It is the lower address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

For all other exceptions taken to EL1, FAR_EL1 is UNKNOWN.

FAR_EL1 is made UNKNOWN on an exception return from EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Accessing FAR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic FAR_EL1 or FAR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

### MRS <Xt>, FAR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```python
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFGWTR_EL2.FAR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x220];
  else
    X[t, 64] = FAR_EL1;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = FAR_EL2;
  else
    X[t, 64] = FAR_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = FAR_EL1;

### MSR FAR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```
MRS <Xt>, FAR_EL12

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
selsif PSTATE.EL == EL1 then
if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
X[t, 64] = NVMem[0x220];
elsif EL2Enabled() && HCR_EL2.NV == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
UNDEFINED;
selsif PSTATE.EL == EL2 then
if HCR_EL2.E2H == '1' then
X[t, 64] = FAR_EL1;
else
UNDEFINED;
selsif PSTATE.EL == EL3 then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
X[t, 64] = FAR_EL1;
else
UNDEFINED;

MSR FAR_EL12, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
selsif PSTATE.EL == EL1 then
if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
NVMem[0x220] = X[t, 64];
elsif EL2Enabled() && HCR_EL2.NV == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
UNDEFINED;
selsif PSTATE.EL == EL2 then
if HCR_EL2.E2H == '1' then
FAR_EL1 = X[t, 64];
else
UNDEFINED;
selsif PSTATE.EL == EL3 then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
FAR_EL1 = X[t, 64];
else
UNDEFINED;
### MRS <xt>, FAR_EL2

\[
\begin{array}{c|cccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
\hline
0b11 & 0b100 & 0b0110 & 0b0000 & 0b000 \\
\end{array}
\]

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = FAR_EL1;
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  X[t, 64] = FAR_EL2;
elsif PSTATE_EL == EL3 then
  X[t, 64] = FAR_EL2;

### MSR FAR_EL2, <xt>

\[
\begin{array}{c|cccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
\hline
0b11 & 0b100 & 0b0110 & 0b0000 & 0b000 \\
\end{array}
\]

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    FAR_EL1 = X[t, 64];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  FAR_EL2 = X[t, 64];
elsif PSTATE_EL == EL3 then
  FAR_EL2 = X[t, 64];
D17.2.41 FAR_EL2, Fault Address Register (EL2)

The FAR_EL2 characteristics are:

**Purpose**

Holds the faulting Virtual Address for all synchronous Instruction Abort exceptions, Data Abort exceptions, PC alignment fault exceptions and Watchpoint exceptions that are taken to EL2.

**Configurations**

AArch64 System register FAR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HDFAR[31:0].

AArch64 System register FAR_EL2 bits [63:32] are architecturally mapped to AArch32 System register HIFAR[31:0].

AArch64 System register FAR_EL2 bits [31:0] are architecturally mapped to AArch32 System register DFAR[31:0] (DFAR_S) when EL2 is implemented.

AArch64 System register FAR_EL2 bits [63:32] are architecturally mapped to AArch32 System register IFAR[31:0] (IFAR_S) when EL2 is implemented.

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

FAR_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faulting Virtual Address for synchronous exceptions taken to EL2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faulting Virtual Address for synchronous exceptions taken to EL2</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:0]**

Faulting Virtual Address for synchronous exceptions taken to EL2. Exceptions that set the FAR_EL2 are Instruction Aborts (EC 0x20 or 0x21), Data Aborts (EC 0x24 or 0x25), PC alignment faults (EC 0x22), and Watchpoints (EC 0x34 or 0x35). ESR_EL2.EC holds the EC value for the exception.

For a synchronous External abort, if the VA that generated the abort was from an address range for which TCR_EL{<0|1>}.TBI{<0|1>} == 1 for the translation regime in use when the abort was generated, then the top eight bits of FAR_EL2 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL2.FnV is 0, and FAR_EL2 is UNKNOWN if ESR_EL2.FnV is 1.

If a memory fault that sets FAR_EL2, other than a Tag Check Fault, is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

On an exception due to a Tag Check Fault caused by a data cache maintenance or other DC instruction, the address held in FAR_EL2 is IMPLEMENTATION DEFINED as one of the following:

- The lowest address that gave rise to the fault.
- The address specified in the register argument of the instruction as generated by MMU faults caused by DC ZVA.
If the exception that updates FAR_EL2 is taken from an Exception level using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are 0xffffffff:

- The faulting address was generated by a load or store instruction that sequentially incremented from address 0xffffffff. Such a load or store instruction is CONSTRAINED UNPREDICTABLE.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

When FEAT_SME is implemented, and the PE sets ESR_EL2.ISV to 0 and ESR_EL2.FnP to 1 on taking a Data Abort exception or Watchpoint exception, the PE sets FAR_EL2 to any address within the naturally-aligned fault granule that contains the virtual address of the memory access that generated the Data Abort or Watchpoint exception.

The naturally-aligned fault granule is one of:

- When ESR_EL2.DFSC is 0b010001, indicating a Synchronous Tag Check fault, it is a 16-byte tag granule.
- When ESR_EL2.DFSC is 0b11010x, indicating an IMPLEMENTATION DEFINED fault, it is an IMPLEMENTATION DEFINED granule.
- Otherwise, it is the smallest implemented translation granule.

When FEAT_MIPS is implemented, the value in FAR_EL2 on a synchronous exception from any of the Memory Copy and Memory Set instructions represents the first element that has not been copied or set, and is determined as follows:

- For a Data Abort generated by the MMU, the value is within the address range of the relevant translation granule, aligned to the size of the relevant translation granule of the address that generated the Data Abort. Bits[(n-1):0] of the value are UNKNOWN, where 2^n is the relevant translation granule size in bytes. For the purpose of calculating the relevant translation granule, if the MMU is disabled for a stage of translation, then the current translation granule size is equal to 2^64 for stage 1, and the PARange for stage 2. The relevant translation granule is:
  - For MMU faults generated at stage 1, the current stage 1 translation granule.
  - For MMU faults generated at stage 2, the smaller of the current stage 1 translation granule and the current stage 2 translation granule.
  - If FEAT_RME is implemented, for a synchronous data abort generated as the result of a GPF, the smallest of the current stage 1 translation granule, the current stage 2 translation granule and the configured granule size in GPCCR_EL3.PGS.
- For a Data Abort generated by a Tag Check failure, the value is the lowest address that failed the Tag Check within the block size of the load or store.
- For a Watchpoint exception, the value is an address range of the size defined by the DCZID_EL0.BS field. This address does not need to be the element with a watchpoint, but can be some earlier element.
- Otherwise, the value is the lowest address in the block size of the load or store.

For a Data Abort or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see Address tagging on page D8-5162.

For a synchronous Tag Check Fault abort, bits[63:60] are UNKNOWN.

Execution at EL1 or EL0 makes FAR_EL2 become UNKNOWN.

--- Note ---

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or data abort. It is the lower address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize between those different faults.
For all other exceptions taken to EL2, FAR_EL2 is **UNKNOWN**.
FAR_EL2 is made **UNKNOWN** on an exception return from EL2.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

### Accessing FAR_EL2

When `HCR_EL2.E2H` is 1, without explicit synchronization, access from EL2 using the mnemonic `FAR_EL2` or `FAR_EL1` are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

#### MRS <Xt>, FAR_EL2

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = FAR_EL1;
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = FAR_EL2;
elsif PSTATE.EL == EL3 then
  X[t, 64] = FAR_EL2;
```

#### MSR FAR_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    FAR_EL1 = X[t, 64];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  FAR_EL2 = X[t, 64];
eisf PSTATE.EL == EL3 then
  FAR_EL2 = X[t, 64];
```
**MRS <Xt>, FAR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFCRTR_EL2.FAR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x220];
  else
    X[t, 64] = FAR_EL1;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = FAR_EL2;
  else
    X[t, 64] = FAR_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = FAR_EL1;

**MSR FAR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFCWTR_EL2.FAR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
    NVMem[0x220] = X[t, 64];
  else
    FAR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    FAR_EL2 = X[t, 64];
  else
    FAR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
  FAR_EL1 = X[t, 64];
D17.2.42 FAR_EL3, Fault Address Register (EL3)

The FAR_EL3 characteristics are:

**Purpose**

Holds the faulting Virtual Address for all synchronous Instruction Abort exceptions, Data Abort exceptions and PC alignment fault exceptions that are taken to EL3.

**Configurations**

This register is present only when EL3 is implemented. Otherwise, direct accesses to FAR_EL3 are UNDEFINED.

**Attributes**

FAR_EL3 is a 64-bit register.

**Field descriptions**

Faulting Virtual Address for synchronous exceptions taken to EL3

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Faulting Virtual Address for synchronous exceptions taken to EL3</td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Bits [63:0]

Faulting Virtual Address for synchronous exceptions taken to EL3. Exceptions that set the FAR_EL3 are Instruction Aborts (EC 0x20 or 0x21), Data Aborts (EC 0x24 or 0x25), and PC alignment faults (EC 0x22). ESR_EL3.EC holds the EC value for the exception.

For a synchronous External abort, if the VA that generated the abort was from an address range for which TCR_ELx.TBI{<0|1>} == 1 for the translation regime in use when the abort was generated, then the top eight bits of FAR_EL3 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL3.FnV is 0, and FAR_EL3 is UNKNOWN if ESR_EL3.FnV is 1.

If a memory fault that sets FAR_EL3, other than a Tag Check Fault, is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

On an exception due to a Tag Check Fault caused by a data cache maintenance or other DC instruction, the address held in FAR_EL3 is IMPLEMENTATION DEFINED as one of the following:

- The lowest address that gave rise to the fault.
- The address specified in the register argument of the instruction as generated by MMU faults caused by DC ZV A.

If the exception that updates FAR_EL3 is taken from an Exception level using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are 0x00000001:

- The faulting address was generated by a load or store instruction that sequentially incremented from address 0xFFFFFFFF. Such a load or store instruction is CONSTRAINED UNPREDICTABLE.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

When FEAT_SME is implemented, and the PE sets ESR_EL3.ISV to 0 and ESR_EL3.FnP to 1 on taking a Data Abort exception, the PE sets FAR_EL3 to any address within the naturally-aligned fault granule that contains the virtual address of the memory access that generated the Data Abort.
The naturally-aligned fault granule is one of:

- When ESR_EL3.DFSC is 0b010001, indicating a Synchronous Tag Check fault, it is a 16-byte tag granule.
- When ESR_EL3.DFSC is 0b11010x, indicating an IMPLEMENTATION DEFINED fault, it is an IMPLEMENTATION DEFINED granule.
- Otherwise, it is the smallest implemented translation granule.

When FEAT_MOPS is implemented, the value in FAR_EL3 on a synchronous exception from any of the Memory Copy and Memory Set instructions represents the first element that has not been copied or set, and is determined as follows:

- For a Data Abort generated by the MMU, the value is within the address range of the relevant translation granule, aligned to the size of the relevant translation granule of the address that generated the Data Abort. Bits[(n-1):0] of the value are UNKNOWN, where 2^n is the relevant translation granule size in bytes. For the purpose of calculating the relevant translation granule, if the MMU is disabled for a stage of translation, then the current translation granule size is equal to 2^64 for stage 1, and the PARange for stage 2. The relevant translation granule is:
  - For MMU faults generated at stage 1, the current stage 1 translation granule.
  - For MMU faults generated at stage 2, the smaller of the current stage 1 translation granule and the current stage 2 translation granule.
  - If FEAT_RME is implemented, for a synchronous data abort generated as the result of a GPF, the smallest of the current stage 1 translation granule, the current stage 2 translation granule and the configured granule size in GPCCR_EL3.PGS.
- For a Data Abort generated by a Tag Check failure, the value is the lowest address that failed the Tag Check within the block size of the load or store.
- Otherwise, the value is the lowest address in the block size of the load or store.

For a Data Abort exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see Address tagging on page D8-5162.

For a synchronous Tag Check Fault abort, bits[63:60] are UNKNOWN.

Execution at EL2, EL1, or EL0 makes FAR_EL3 become UNKNOWN.

--- Note ---

The address held in this register is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or data abort. It is the lowest address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

---

For all other exceptions taken to EL3, FAR_EL3 is UNKNOWN.

FAR_EL3 is made UNKNOWN on an exception return from EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FAR_EL3

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, FAR_EL3

\[
\begin{array}{c}
\text{if } \text{PSTATE.EL} = \text{EL0} \text{ then} \\
\text{UNDEFINED;}
\end{array}
\]

\[
\begin{array}{c}
\text{elsif } \text{PSTATE.EL} = \text{EL1} \text{ then} \\
\text{UNDEFINED;}
\end{array}
\]

\[
\begin{array}{c}
\text{elsif } \text{PSTATE.EL} = \text{EL2} \text{ then} \\
\text{UNDEFINED;}
\end{array}
\]

\[
\begin{array}{c}
\text{elsif } \text{PSTATE.EL} = \text{EL3} \text{ then} \\
X[t, 64] = \text{FAR_EL3;}
\end{array}
\]

MSR FAR_EL3, <Xt>

\[
\begin{array}{c}
\text{if } \text{PSTATE.EL} = \text{EL0} \text{ then} \\
\text{UNDEFINED;}
\end{array}
\]

\[
\begin{array}{c}
\text{elsif } \text{PSTATE.EL} = \text{EL1} \text{ then} \\
\text{UNDEFINED;}
\end{array}
\]

\[
\begin{array}{c}
\text{elsif } \text{PSTATE.EL} = \text{EL2} \text{ then} \\
\text{UNDEFINED;}
\end{array}
\]

\[
\begin{array}{c}
\text{elsif } \text{PSTATE.EL} = \text{EL3} \text{ then} \\
\text{FAR_EL3} = X[t, 64];
\end{array}
\]
**D17.2.43 FPEXC32_EL2, Floating-Point Exception Control register**

The FPEXC32_EL2 characteristics are:

**Purpose**

Allows access to the AArch32 register FPEXC from AArch64 state only. Its value has no effect on execution in AArch64 state.

**Configurations**

AArch64 System register FPEXC32_EL2 bits [31:0] are architecturally mapped to AArch32 System register FPEXC[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to FPEXC32_EL2 are UNDEFINED.

If EL2 is not implemented but EL3 is implemented, and EL1 is capable of using AArch32, then this register is not RES0.

Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

**Attributes**

FPEXC32_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>31</td>
<td>EX, bit [31]</td>
</tr>
<tr>
<td>30</td>
<td>EN, bit [30]</td>
</tr>
<tr>
<td>27:24</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>23:16</td>
<td>VECITR, RES0.</td>
</tr>
<tr>
<td>11:10</td>
<td>IDF, RES0.</td>
</tr>
<tr>
<td>9</td>
<td>IXF, RES0.</td>
</tr>
<tr>
<td>8</td>
<td>IOF, RES0.</td>
</tr>
<tr>
<td>7:0</td>
<td>OFF, RES0.</td>
</tr>
<tr>
<td>31:28</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>27:25</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>24</td>
<td>DEX, RES0.</td>
</tr>
<tr>
<td>23:16</td>
<td>FP2V, RES0.</td>
</tr>
<tr>
<td>15:8</td>
<td>TFV, RES0.</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**EX, bit [31]**

Exception bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RAZ/WI.

**EN, bit [30]**

Enables access to the Advanced SIMD and floating-point functionality from all Exception levels, except that setting this field to 0 does not disable the following:

- VMSR accesses to the FPEXC or FPSID.
- VMRS accesses from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2.

0b0 Accesses to the FPSCR, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers, are UNDEFINED at all Exception levels.

0b1 This control permits access to the Advanced SIMD and floating-point functionality at all Exception levels.
Execution of Advanced SIMD and floating-point instructions in AArch32 state can be disabled or trapped by the following controls:

- **CPACR.cp10**, or, if executing at EL0, **CPACR_EL1.FPEN**.
- **FPEXC.EN**.
- If executing in Non-secure state:
  - **HCPTR.TCP10**, or if EL2 is using AArch64, **CPTR_EL2.TFP**.
  - **NSACR.cp10**, or if EL3 is using AArch64, **CPTR_EL3.TFP**.
- For Advanced SIMD instructions only:
  - **CPACR.ASEDIS**.
  - If executing in Non-secure state, **HCPTR.TASE** and **NSACR.NSTRCDIS**.

See the descriptions of the controls for more information.

--- Note

When executing at EL0 using AArch32:

- If EL1 is using AArch64, then the Effective value of **FPEXC.EN** is 1.
- If EL2 is using AArch64 and is enabled in the current Security state, **HCR_EL2.TGE** is 1, and the Effective value of **HCR_EL2.RW** is 1, then the Effective value of **FPEXC.EN** is 1. However, Arm deprecates using the value of FPEXC32_EL2.EN to determine behavior.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**DEX, bit [29]**

Defined synchronous exception on floating-point execution.

This field identifies whether a synchronous exception generated by the attempted execution of an instruction was generated by an unallocated encoding. The instruction must be in the encoding space that is identified by the pseudocode function ExecutingCP10or11Instr() returning TRUE. This field also indicates whether the FPEXC32_EL2.TFV field is valid.

The meaning of this bit is:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The exception was generated by the attempted execution of an unallocated instruction in the encoding space that is identified by the pseudocode function ExecutingCP10or11Instr(). If FPEXC32_EL2.TFV is RW then it is invalid and <strong>UNKNOWN</strong>. If FPEXC32_EL2.{IDF, IXF, UFF, OFF, DZF, IOF} are RW then they are invalid and <strong>UNKNOWN</strong>.</td>
</tr>
<tr>
<td>0b1</td>
<td>The exception was generated during the execution of an allocated encoding. FPEXC32_EL2.TFV is valid and indicates the cause of the exception.</td>
</tr>
</tbody>
</table>

On an exception that sets this bit to 1 the exception-handling routine must clear this bit to 0.

On an implementation that both does not support trapping of floating-point exceptions and implements the AArch32 **FPSCR.\{Stride, Len\}** fields as RAZ, this bit is **RES0**.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**FP2V, bit [28]**

FPINST2 instruction valid bit. From Armv8, this bit is **RES0**.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

Access to this field is **RES0**.

**VV, bit [27]**

VECITR valid bit. From Armv8, this bit is **RES0**.
TFV, bit [26]

Trapped Fault Valid bit. Valid only when the value of FPEXC32_EL2.DEX is 1. When valid, it indicates the cause of the exception and therefore whether FPEXC32_EL2.{IDF, IXF, UFF, OFF, DZF, IOF} are valid.

0b0  The exception was caused by the execution of a floating-point VABS, VADD, VDIV, VFMA, VFMS, VFNMA, VFNMS, VMLA, VMLS, VMOV, VMUL, VNEG, VNMLA, VNMUL, VNSQRT, or VSUB instruction when one or both of FPEXC32_EL2.{Stride, Len} was non-zero. If FPEXC32_EL2.{IDF, IXF, UFF, OFF, DZF, IOF} are RW then they are invalid and UNKNOWN.

0b1  FPEXC32_EL2.{IDF, IXF, UFF, OFF, DZF, IOF} indicate the presence of trapped floating-point exceptions that had occurred at the time of the exception. Bits are set for all trapped exceptions that had occurred at the time of the exception.

This bit returns a status value and ignores writes.

When the value of FPEXC32_EL2.DEX is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

• When !ImpDefBool("Support trapping of floating-point exceptions"), access to this field is RAZ/WI.

• When ImpDefBool("Implemented FPSCR LEN, STRIDE as RAZ"), access to this field is RAO/WI.

Bits [25:11]

Reserved, RES0.

VECITR, bits [10:8]

Vector iteration count. From Armv8, this field is RES1.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RES1.

IDF, bit [7]

Input Denormal trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Input Denormal exception occurred while FPSCR.IDE was 1:

0b0  Input Denormal exception has not occurred.

0b1  Input Denormal exception has occurred.

Input Denormal exceptions can occur only when FPSCR.FZ is 1.

Note

A half-precision floating-point value that is flushed to zero because the value of FPSCR.FZ16 is 1 does not generate an Input Denormal exception.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC32_EL2.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Inexact exception occurred while FPSCR.IXE was 1:

0b0  Inexact exception has not occurred.
0b1  Inexact exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

UFF, bit [3]

Underflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Underflow exception occurred while FPSCR.UFE was 1:

0b0  Underflow exception has not occurred.
0b1  Underflow exception has occurred.

Underflow trapped exceptions can occur:
• On half-precision data-processing instructions only when FPSCR.FZ16 is 0.
• Otherwise only when FPSCR.FZ is 0.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC32_EL2.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

OFF, bit [2]

Overflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Overflow exception occurred while FPSCR.OFE was 1:

0b0  Overflow exception has not occurred.
0b1  Overflow exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

DZF, bit [1]

Divide by Zero trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether a Divide by Zero exception occurred while FPSCR.DZE was 1:

0b0  Divide by Zero exception has not occurred.
0b1  Divide by Zero exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.
On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IOF, bit [0]**

Invalid Operation trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Invalid Operation exception occurred while FPSCR.IOE was 1:

- $0b0$ Invalid Operation exception has not occurred.
- $0b1$ Invalid Operation exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing FPEXC32_EL2

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, FPEXC32_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() &amp; HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if Halted() &amp; HaveEL(EL3) &amp; EDSCR.SDD == '1' &amp; boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" &amp; CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elsif HCR_EL2.E2H == '0' &amp; CPTR_EL2.TFP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x07);
  elsif HCR_EL2.E2H == '1' &amp; CPTR_EL2.FPEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x07);
  elsif HaveEL(EL3) &amp; CPTR_EL3.TFP == '1' then
    if Halted() &amp; EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x07);
    else
      X[t, 64] = FPEXC32_EL2;
  else
    if CPTR_EL3.TFP == '1' then
      AArch64.SystemAccessTrap(EL3, 0x07);
    else
      X[t, 64] = FPEXC32_EL2;
```
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elsif HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
    AArch64.SystemAccessTrap(EL2, 0x07);
  elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x07);
  elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x07);
    else
      FPEXC32_EL2 = X[t, 64];
  elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TFP == '1' then
      AArch64.SystemAccessTrap(EL3, 0x07);
    else
      FPEXC32_EL2 = X[t, 64];
    op0  op1  CRn  CRm  op2
    0b11  0b100  0b0101  0b0011  0b000
D17.2.44   GCR_EL1, Tag Control Register.

The GCR_EL1 characteristics are:

**Purpose**
Tag Control Register.

**Configurations**
This register is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to GCR_EL1 are UNDEFINED.

**Attributes**
GCR_EL1 is a 64-bit register.

**Field descriptions**

**Bits [63:17]**
Reserved, RES0.

**RRND, bit [16]**
Controls generation of tag values by the IRG instruction.
0b0     IRG generates a tag value as defined by RandomTag().
0b1     IRG generates an implementation-specific tag value with a distribution of tag values no worse than generated with GCR_EL1.RRND == 0.

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Exclude, bits [15:0]**
Allocation Tag values excluded from selection by ChooseNonExcludedTag().
If all bits of GCR_EL1.Exclude are 1, then the Allocation Tag value 0 will be used.
The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing GCR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, GCR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.ATA == '0' then
        UNDEFINED;
    elseif EL2Enabled() && HCR_EL2.ATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) && SCR_EL3.ATA == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        X[t, 64] = GCR_EL1;
    end
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.ATA == '0' then
        UNDEFINED;
    elseif HaveEL(EL3) && SCR_EL3.ATA == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        X[t, 64] = GCR_EL1;
    end
elsif PSTATE.EL == EL3 then
    X[t, 64] = GCR_EL1;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.ATA == '0' then
        UNDEFINED;
    elseif EL2Enabled() && HCR_EL2.ATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) && SCR_EL3.ATA == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        GCR_EL1 = X[t, 64];
    end
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.ATA == '0' then
        UNDEFINED;
    elseif HaveEL(EL3) && SCR_EL3.ATA == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        GCR_EL1 = X[t, 64];
    end
elsif PSTATE.EL == EL3 then
    GCR_EL1 = X[t, 64];
else
    GCR_EL1 = X[t, 64];
end

MSR GCR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b110</td>
</tr>
</tbody>
</table>
D17.2.45  GMID_EL1, Multiple tag transfer ID register

The GMID_EL1 characteristics are:

Purpose
Indicates the block size that is accessed by the LDGM and STGM System instructions.

Configurations
This register is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to GMID_EL1 are UNDEFINED.

Attributes
GMID_EL1 is a 64-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:4]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[3:0]</td>
<td>BS, Log2 of the block size in words. The minimum supported size is 16B (value == 2) and the maximum is 256B (value == 6).</td>
</tr>
</tbody>
</table>

Accessing GMID_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[
MRS <Xt>, GMID_EL1
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        end;
    else
        UNDEFINED;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID5 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = GMID_EL1;
    end;
elsif PSTATE.EL == EL2 then
    X[t, 64] = GMID_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = GMID_EL1;
D17.2.46  HACR_EL2, Hypervisor Auxiliary Control Register

The HACR_EL2 characteristics are:

**Purpose**

Controls trapping to EL2 of IMPLEMENTATION DEFINED aspects of EL1 or EL0 operation.

--- Note ---

Arm recommends that the values in this register do not cause unnecessary traps to EL2 when HCR_EL2.{E2H, TGE} == {1, 1}.

**Configurations**

AArch64 System register HACR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HACR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

HACR_EL2 is a 64-bit register.

**Field descriptions**

**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing HACR_EL2**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, HACR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b11</td>
</tr>
</tbody>
</table>

```
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
else
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = HACR_EL2;
```
elsif PSTATE.EL == EL3 then
    X[t, 64] = HACR_EL2;

**MSR HACR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elseif PSTATE.EL == EL2 then
    HACR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    HACR_EL2 = X[t, 64];
D17.2.47 HAFGRTR_EL2, Hypervisor Activity Monitors Fine-Grained Read Trap Register

The HAFGRTR_EL2 characteristics are:

**Purpose**

Provides controls for traps of MRS reads of Activity Monitors System registers.

**Configurations**

This register is present only when FEAT_AMUv1 is implemented and FEAT_FGT is implemented. Otherwise, direct accesses to HAFGRTR_EL2 are UNDEFINED.

**Attributes**

HAFGRTR_EL2 is a 64-bit register.

**Field descriptions**

**Bits [63:50]**

Reserved, RES0.

**AMEVTYPER1<\(x\)>_EL0, bit [19+2x], for x = 15 to 0**

*When AMEVTPERI<\(x\)> is implemented:*

AMEVTPERI<\(x\)>_EL0

Trap MRS reads of AMEVTPERI<\(x\)>_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTPERI<\(x\)> at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of AMEVTPERI<\(x\)>_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVTPERI<\(x\)> at EL0 using AArch32 are not trapped by this mechanism.
If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} !={1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of AMEVTYPE1\(n\)\_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of AMEVTYPE1\(n\) at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

AMEVCTR1\(x\)\_EL0, bit \([18+2x]\), for \(x = 15\) to \(0\)

When AMEVCTR1\(x\) is implemented:

AMEVCNTR1\(x\)\_EL0

Trap MRS reads of AMEVCTR1\(x\)\_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCTR1\(x\)\_EL0 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

- MRS reads of AMEVCTR1\(x\)\_EL0 at EL1 and EL0 using AArch64 and MRC reads of AMEVCTR1\(x\)\_EL0 at EL0 using AArch32 are not trapped by this mechanism.
- If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} !={1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:
  - MRS reads of AMEVCTR1\(x\)\_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
  - MRC reads of AMEVCTR1\(x\) at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

AMCNTEN\(x\), bit \([17x]\), for \(x = 1\) to \(0\)

Trap MRS reads and MRC reads of multiple System registers.

Enables a trap to EL2 the following operations:

- At EL1 and EL0 using AArch64: MRS reads of AMCNTENCLR\(x\)\_EL0 and AMCNTENSET\(x\).
- At EL0 using AArch32 when EL1 is using AArch64: MRC reads of AMCNTENCLR\(x\) and AMCNTENSET\(x\).

- The operations listed above are not trapped by this mechanism.
- If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} !={1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:
  - MRS reads at EL1 and EL0 using AArch64 of AMCNTENCLR\(x\)\_EL0 and AMCNTENSET\(x\)\_EL0 are trapped to EL2 and reported with EC syndrome value 0x18.
  - MRC reads at EL0 using AArch32 of AMCNTENCLR\(x\) and AMCNTENSET\(x\) are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.
Bits [16:5]

Reserved, RES0.

AMEVCNTR0<\texttt{x}>_EL0, bit \texttt{[x+1]}, for \texttt{x} = 3 to 0

Trap \texttt{MRS} reads of AMEVCNTR0<\texttt{n}>_EL0 at EL1 and EL0 using AArch64 and \texttt{MRC} reads of AMEVCNTR0<\texttt{n}>_EL0 at EL0 using AArch32 when EL1 is using AArch64 to EL2.

- \texttt{0b0} \texttt{MRS} reads of AMEVCNTR0<\texttt{n}>_EL0 at EL1 and EL0 using AArch64 and \texttt{MRC} reads of AMEVCNTR0<\texttt{n}> at EL0 using AArch32 are not trapped by this mechanism.
- \texttt{0b1} If EL2 is implemented and enabled in the current Security state, HCR_EL2.\{E2H, TGE\} \neq \{1, 1\}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn = 1, then, unless the read generates a higher priority exception:
  - \texttt{MRS} reads of AMEVCNTR0<\texttt{n}>_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
  - \texttt{MRC} reads of AMEVCNTR0<\texttt{n}> at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Accessing HAFGRTR_EL2

Accesses to this register use the following encodings in the System register encoding space:

\texttt{MRS \langle Xt \rangle, HAFGRTR_EL2}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0011</td>
<td>0b0001</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsf if PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = NVMem[0x1E8];
e1sf EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsf if PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.FGTEn == '0' then
    UNDEFINED;
e1sf HaveEL(EL3) && SCR_EL3.FGTEn == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
e1se Halted() && EDSCR.SDD == '0' then
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
e1se
  else
    X[t, 64] = HAFGRTR_EL2;
elsf if PSTATE.EL == EL3 then
  X[t, 64] = HAFGRTR_EL2;
**MSR HAFGRTR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0011</td>
<td>0b0001</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() & HCR_EL2.NV == '11' then
        VMEM[0x1E8] = X[t, 64];
    elsif EL2Enabled() & HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE_EL == EL2 then
        if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
          when SDD == '1"
          & SCR_EL3.FGTEn == '0' then
            UNDEFINED;
        elsif HaveEL(EL3) & SCR_EL3.FGTEn == '0' then
            if Halted() & EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end
        else
            HAFGRTR_EL2 = X[t, 64];
    elsif PSTATE_EL == EL3 then
        HAFGRTR_EL2 = X[t, 64];
    else
        HAFGRTR_EL2 = X[t, 64];
D17.2.48 HCR_EL2, Hypervisor Configuration Register

The HCR_EL2 characteristics are:

**Purpose**

Provides configuration controls for virtualization, including defining whether various operations are trapped to EL2.

**Configurations**

AArch64 System register HCR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HCR[31:0].

AArch64 System register HCR_EL2 bits [63:32] are architecturally mapped to AArch32 System register HCR2[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

The bits in this register behave as if they are 0 for all purposes other than direct reads of the register if EL2 is not enabled in the current Security state.

**Attributes**

HCR_EL2 is a 64-bit register.

**Field descriptions**

**TWEDEL, bits [63:60]**

*When FEAT_TWED is implemented:*

TWE Delay. A 4-bit unsigned number that, when HCR_EL2.TWEDEn is 1, encodes the minimum delay in taking a trap of WFE* caused by HCR_EL2.TWE as \(2^{(TWEDEL + 8)}\) cycles.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.
TWEDEn, bit [59]

When FEAT_TWED is implemented:

TWE Delay Enable. Enables a configurable delayed trap of the WFE* instruction caused by HCR_EL2.TWE.

0b0   The delay for taking the trap is IMPLEMENTATION DEFINED.
0b1   The delay for taking the trap is at least the number of cycles defined in HCR_EL2.TWEDEL.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TID5, bit [58]

When FEAT_MTE2 is implemented:

Trap ID group 5. Traps the following register accesses to EL2, when EL2 is enabled in the current Security state:
AArch64:

• GMID_EL1.

0b0   This control does not cause any instructions to be trapped.
0b1   The specified EL1 and EL0 accesses to ID group 5 registers are trapped to EL2.

When the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field has an Effective value of 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DCT, bit [57]

When FEAT_MTE2 is implemented:

Default Cacheability Tagging. When HCR_EL2.DC is in effect, controls whether stage 1 translations are treated as Tagged or Untagged.

0b0   Stage 1 translations are treated as Untagged.
0b1   Stage 1 translations are treated as Tagged.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ATA, bit [56]

When FEAT_MTE2 is implemented:

Allocation Tag Access. When HCR_EL2.{E2H,TGE} != {1,1}, controls access to Allocation Tags, System registers for Memory tagging, and prevention of Tag checking, at EL1 and EL0.

0b0   Access to Allocation Tags is prevented at EL1 and EL0.

Accesses at EL1 to GCR_EL1, RGSR_EL1, TFSR_EL1, TFSR_EL2, or TFSRE0_EL1 that are not UNDEFINED are trapped to EL2.

Memory accesses at EL1 and EL0 are not subject to a Tag Check operation.

0b1   This control does not prevent access to Allocation Tags at EL1 and EL0.
This control does not prevent Tag checking at EL1 and EL0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

TTLBOS, bit [55]

When FEAT_EVT is implemented:

Trap TLB maintenance instructions that operate on the Outer Shareable domain. Traps execution of those TLB maintenance instructions at EL1 to EL2, when EL2 is enabled in the current Security state. This applies to the following instructions:

- TLBI VMALLE1OS, TLBI VMALLE1OSNXS, TLBI VAE1OS, TLBI VAEL1OSNXS, TLBI ASIDE1OS, TLBI ASIDE1OSNXS, TLBI VAAE1OS, TLBI VAEL1OS, TLBI VAEE1OSNXS, TLBI VAAE1OSNXS, TLBI VAAE1OS, TLBI VAEE1OSNXS, TLBI VAAE1OSNXS, TLBI VAAE1OS, TLBI VAEE1OSNXS, TLBI VAAE1OSNXS, TLBI RVAAE1OS, TLBI RVAAE1OSNXS, TLBI RVAAE1OS, TLBI RVAAE1OSNXS, TLBI RVAAE1OS, TLBI RVAAE1OSNXS, and TLBI RVAAE1OSNXS.

0b0  This control does not cause any instructions to be trapped.
0b1  Execution of the specified instructions are trapped to EL2.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

TTLBIS, bit [54]

When FEAT_EVT is implemented:

Trap TLB maintenance instructions that operate on the Inner Shareable domain. Traps execution of those TLB maintenance instructions at EL1 to EL2, when EL2 is enabled in the current Security state. This applies to the following instructions:

- When EL1 is using AArch64, TLBI VMALLE1IS, TLBI VMALLE1ISNXS, TLBI VAEL1IS, TLBI VAEL1ISNXS, TLBI VAAE1IS, TLBI VAEL1ISNXS, TLBI VAAE1ISNXS, TLBI VAAE1IS, TLBI VAEL1ISNXS, TLBI VAAE1ISNXS, TLBI VAAE1IS, TLBI VAEL1ISNXS, TLBI VAAE1ISNXS, TLBI VAAE1IS, TLBI VAEL1ISNXS, TLBI VAAE1ISNXS, TLBI RVAAE1IS, TLBI RVAAE1ISNXS, TLBI RVAAE1IS, TLBI RVAAE1ISNXS, and TLBI RVAAE1IS, TLBI RVAAE1ISNXS.
- When EL1 is using AArch32, TLBIALL1IS, TLBIMVAI3, TLBIASID1IS, TLBIMVAI3IS, TLBIASID1IS, and TLBIMVAI3IS.

0b0  This control does not cause any instructions to be trapped.
0b1  Execution of the specified instructions are trapped to EL2.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.
EnSCXT, bit [53]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Enable Access to the SCXTNUM_EL1 and SCXTNUM_EL0 registers. The defined values are:

- **0b0**  When HCR_EL2.E2H is 0 or HCR_EL2.TGE is 0, and EL2 is enabled in the current Security state, EL1 and EL0 access to SCXTNUM_EL0 and EL1 access to SCXTNUM_EL1 is disabled by this mechanism, causing an exception to EL2, and the values of these registers to be treated as 0.

- **0b1**  This control does not cause accesses to SCXTNUM_EL0 or SCXTNUM_EL1 to be trapped.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

TOCU, bit [52]

When FEAT_EVT is implemented:

Trap cache maintenance instructions that operate to the Point of Unification. Traps execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state. This applies to the following instructions:

- When SCTLR_EL1.UCI is 1, HCR_EL2.{TGE, E2H} is not {1, 1}, and EL0 is using AArch64, IC IV AU, DC CV AU.

- When EL1 is using AArch64, IC IVAU, IC IALLU, DC CVAU.

- When EL1 is using AArch32, ICIMVAU, ICIALLU, DCCMVAU.

_____ **Note** ______

An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:

- IC IALLUIS and IC IALLU are always UNDEFINED at EL0 using AArch64.

- ICIMVAU, ICIALLU, ICIALLUIS, and DCCMVAU are always UNDEFINED at EL0 using AArch32.

- **0b0**  This control does not cause any instructions to be trapped.

- **0b1**  Execution of the specified instructions are trapped to EL2.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.
AMVOFFEN, bit [51]

**When FEAT_AMUv1p1 is implemented:**
Activity Monitors Virtual Offsets Enable.

- **0b0** Virtualization of the Activity Monitors is disabled. Indirect reads of the virtual offset registers are zero.
- **0b1** Virtualization of the Activity Monitors is enabled.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

TICAB, bit [50]

**When FEAT_EVT is implemented:**
Trap ICIAULLUIS/IC IALLUIS cache maintenance instructions. Traps execution of those cache maintenance instructions at EL1 to EL2, when EL2 is enabled in the current Security state. This applies to the following instructions:

- When EL1 is using AArch64, IC IALLUIS.
- When EL1 is using AArch32, ICIAULLUIS.

- **0b0** This control does not cause any instructions to be trapped.
- **0b1** EL1 execution of the specified instructions is trapped to EL2.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

TID4, bit [49]

**When FEAT_EVT is implemented:**
Trap ID group 4. Traps the following register accesses to EL2, when EL2 is enabled in the current Security state:

**AArch64:**
- EL1 reads of CCSIDR_EL1, CCSIDR2_EL1, CLIDR_EL1, and CSSELR_EL1.
- EL1 writes to CSSELR_EL1.

**AArch32:**
- EL1 reads of CCSIDR, CCSIDR2, CLIDR, and CSSELR.
- EL1 writes to CSSELR.

- **0b0** This control does not cause any instructions to be trapped.
- **0b1** The specified EL1 and EL0 accesses to ID group 4 registers are trapped to EL2.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
**GPF, bit [48]**

_When FEAT_RME is implemented:_

Controls the reporting of Granule protection faults at EL0 and EL1.

- **0b0** This control does not cause exceptions to be routed from EL0 and EL1 to EL2.
- **0b1** Instruction Abort exceptions and Data Abort exceptions due to GPFs from EL0 and EL1 are routed to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**

Reserved, RES0.

**FIEN, bit [47]**

_When FEAT_RASv1p1 is implemented:_

Fault Injection Enable. Unless this bit is set to 1, accesses to the ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1 registers from EL1 generate a Trap exception to EL2, when EL2 is enabled in the current Security state, reported using EC syndrome value 0x18.

- **0b0** Accesses to the specified registers from EL1 are trapped to EL2, when EL2 is enabled in the current Security state.
- **0b1** This control does not cause any instructions to be trapped.

If EL2 is disabled in the current Security state, the Effective value of HCR_EL2.FIEN is **0b1**.

If ERRIDR_EL1.NUM is zero, meaning no error records are implemented, or no error record accessible using System registers is owned by a node that implements the RAS Common Fault Injection Model Extension, then this bit might be RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**

Reserved, RES0.

**FWB, bit [46]**

_When FEAT_S2FWB is implemented:_

Forced Write-Back. Defines the combined cacheability attributes in a 2 stage translation regime.

**Note**

When FEAT_MTE is implemented, if the stage 1 page or block descriptor specifies the Tagged attribute, the final memory type is Tagged only if the final cacheable memory type is Inner and Outer Write-back cacheable and the final allocation hints are Read-Allocate, Write-Allocate.

- **0b0** When this bit is 0, then:
  - The combination of stage 1 and stage 2 translations on memory type and cacheability attributes are as described in the Armv8.0 architecture. For more information, see _Stage 2 memory type and Cacheability attributes when FEAT_S2FWB is disabled_ on page D8-5153.
  - The encoding of the stage 2 memory type and cacheability attributes in bits[5:2] of the stage 2 page or block descriptors are as described in the Armv8.0 architecture.

- **0b1** When this bit is 1, then:
  - Bit[5] of stage 2 page or block descriptor is RES0.
• When bit[4] of stage 2 page or block descriptor is 1 and when:
  — Bits[3:2] of stage 2 page or block descriptor are 0b11, the resultant memory type and inner or outer cacheability attribute is the same as the stage 1 memory type and inner or outer cacheability attribute.
  — Bits[3:2] of stage 2 page or block descriptor are 0b10, the resultant memory type and attribute is Normal Write-Back.
  — Bits[3:2] of stage 2 page or block descriptor are 0b0x, the resultant memory type will be Normal Non-cacheable except where the stage 1 memory type was Device-<attr> the resultant memory type will be Device-<attr>.
• When bit[4] of stage 2 page or block descriptor is 0 the memory type is Device, and when:
  — Bits[3:2] of stage 2 page or block descriptor are 0b00, the stage 2 memory type is Device-nGnRnE.
  — Bits[3:2] of stage 2 page or block descriptor are 0b01, the stage 2 memory type is Device-nGnRE.
  — Bits[3:2] of stage 2 page or block descriptor are 0b10, the stage 2 memory type is Device-nGRE.
  — Bits[3:2] of stage 2 page or block descriptor are 0b11, the stage 2 memory type is Device-GRE.
• If the stage 1 translation specifies a cacheable memory type, then the stage 1 cache allocation hint is applied to the final cache allocation hint where the final memory type is cacheable.
• If the stage 1 translation does not specify a cacheable memory type, then if the final memory type is cacheable, it is treated as read allocate, write allocate.

The stage 1 and stage 2 memory types are combined in the manner described in *Stage 2 memory type and Cacheability attributes when FEAT_S2FWB is disabled* on page D8-5153.

In Secure state, this bit applies to both the Secure stage 2 translation and the Non-secure stage 2 translation.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**NV2, bit [45]**

*When FEAT_NV2 is implemented:*

Nested Virtualization. Changes the behaviors of HCR_EL2.{NV1, NV} to provide a mechanism for hardware to transform reads and writes from System registers into reads and writes from memory.

- **0b0** This bit has no effect on the behavior of HCR_EL2.{NV1, NV}. The behavior of HCR_EL2.{NV1, NV} is as defined for FEAT_NV.
- **0b1** Redefines behavior of HCR_EL2{NV1, NV} to enable:
  - Transformation of read/writes to registers into read/writes to memory.
  - Redirection of EL2 registers to EL1 registers.

Any exception taken from EL1 and taken to EL1 causes SPSR_EL1.M[3:2] to be set to 0b10 and not 0b01.

When HCR_EL2.NV is 0, the Effective value of this field is 0 and this field is treated as 0 for all purposes other than direct reads and writes of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Otherwise:
Reserved, RES0.

AT, bit [44]

When FEAT_NV is implemented:
Address Translation. EL1 execution of the following address translation instructions is trapped to EL2, when EL2 is enabled in the current Security state, reported using EC syndrome value 0x18:
- AT S1E0R, AT S1E0W, AT S1E1R, AT S1E1W, AT S1E1RP, AT S1E1WP.

0b0 This control does not cause any instructions to be trapped.
0b1 EL1 execution of the specified instructions is trapped to EL2.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

NV1, bit [43]

When FEAT_NV2 is implemented:
Nested Virtualization.
0b0 If HCR_EL2.{NV2, NV} are both 1, accesses executed from EL1 to implemented EL2, EL02, or EL2 registers are transformed to loads and stores.
If HCR_EL2.NV2 is 0 or HCR_EL2.{NV2, NV} == {1, 0}, this control does not cause any instructions to be trapped.
0b1 If HCR_EL2.NV2 is 1, accesses executed from EL1 to implemented EL2 registers are transformed to loads and stores.
If HCR_EL2.NV2 is 0, EL1 accesses to VBAR_EL1, ELR_EL1, SPSR_EL1, and, when FEAT_CSV2_2 or FEAT_CSV2_1p2 is implemented, SCXTNUM_EL1, are trapped to EL2, when EL2 is enabled in the current Security state, and are reported using EC syndrome value 0x18.

If HCR_EL2.NV2 is 1, the value of HCR_EL2.NV1 defines which EL1 register accesses are transformed to loads and stores. These transformed accesses have priority over the trapping of registers.

The trapping of EL1 registers caused by other control bits has priority over the transformation of these accesses.

If a register is specified that is not implemented by an implementation, then access to that register are UNDEFINED.

For the list of registers affected, see Enhanced support for nested virtualization on page D8-5175.

If HCR_EL2.{NV1, NV} is {0, 1}, any exception taken from EL1, and taken to EL1, causes the SPSR_EL1.M[3:2] to be set to 0b10, and not 0b01.

If HCR_EL2.{NV1, NV} is {1, 1}, then:

- The EL1 translation table Block and Page descriptors:
  - Bit[54] holds the PXN instead of the UXN.
  - Bit[53] is RES0.
  - Bit[6] is treated as 0 regardless of the actual value.
- If Hierarchical Permissions are enabled, the EL1 translation table Table descriptors are as follows:
  - Bit[61] is treated as 0 regardless of the actual value.
  - Bit[60] holds the PXNTable instead of the UXNTable.
  - Bit[59] is RES0.
• When executing at EL1, the PSTATE.PAN bit is treated as zero for all purposes except reading the value of the bit.

• When executing at EL1, the LDTR* instructions are treated as the equivalent LDR* instructions, and the STTR* instructions are treated as the equivalent STR* instructions.

If HCR_EL2.{NV1, NV} are \{1, 0\}, then the behavior is a constrained unpredictable choice of:

• Behaving as if HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1 for all purposes other than reading back the value of the HCR_EL2.NV bit.

• Behaving as if HCR_EL2.NV is 0 and HCR_EL2.NV1 is 0 for all purposes other than reading back the value of the HCR_EL2.NV1 bit.

• Behaving with regard to the HCR_EL2.NV and HCR_EL2.NV1 bits behavior as defined in the rest of this description.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally unknown value.

When FEAT_NV is implemented:

Nested Virtualization. EL1 accesses to certain registers are trapped to EL2, when EL2 is enabled in the current Security state.

0b0 This control does not cause any instructions to be trapped.

0b1 EL1 accesses to VBAR_EL1, ELR_EL1, SPSR_EL1, and, when FEAT_CSV2_2 or FEAT_CSV2_1p2 is implemented, SCXTNUM_EL1, are trapped to EL2, when EL2 is enabled in the current Security state, and are reported using EC syndrome value 0x18.

If HCR_EL2.NV is 1 and HCR_EL2.NV1 is 0, then the following effects also apply:

• Any exception taken from EL1, and taken to EL1, causes the SPSR_EL1.M[3:2] to be set to 0b10, and not 0b01.

If HCR_EL2.NV and HCR_EL2.NV1 are both set to 1, then the following effects also apply:

• The EL1 translation table Block and Page descriptors:
  — Bit[54] holds the PXN instead of the UXN.
  — Bit[53] is RES0.
  — Bit[6] is treated as 0 regardless of the actual value.

• If Hierarchical Permissions are enabled, the EL1 translation table Table descriptors are as follows:
  — Bit[61] is treated as 0 regardless of the actual value.
  — Bit[60] holds the PXNTable instead of the UXNTable.
  — Bit[59] is RES0.

• When executing at EL1, the PSTATE.PAN bit is treated as zero for all purposes except reading the value of the bit.

• When executing at EL1, the LDTR* instructions are treated as the equivalent LDR* instructions, and the STTR* instructions are treated as the equivalent STR* instructions.

If HCR_EL2.NV is 0 and HCR_EL2.NV1 is 1, then the behavior is a constrained unpredictable choice of:

• Behaving as if HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1 for all purposes other than reading back the value of the HCR_EL2.NV bit.

• Behaving as if HCR_EL2.NV is 0 and HCR_EL2.NV1 is 0 for all purposes other than reading back the value of the HCR_EL2.NV1 bit.

• Behaving with regard to the HCR_EL2.NV and HCR_EL2.NV1 bits behavior as defined in the rest of this description.

This bit is permitted to be cached in a TLB.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**

Reserved, RES0.

**NV, bit [42]**

**When FEAT_NV2 is implemented:**

Nested Virtualization.

When HCR_EL2.NV2 is 1, redefines register accesses so that:

- Instructions accessing the Special purpose registers SPSR_EL2 and ELR_EL2 instead access SPSR_EL1 and ELR_EL1 respectively.
- Instructions accessing the System registers ESR_EL2 and FAR_EL2 instead access ESR_EL1 and FAR_EL1.

When HCR_EL2.NV2 is 0, or if FEAT_NV2 is not implemented, traps functionality that is permitted at EL2 and would be **UNDEFINED** at EL1 if this field was 0, when EL2 is enabled in the current Security state. This applies to the following operations:

- EL1 accesses to Special-purpose registers that are not **UNDEFINED** at EL2.
- EL1 accesses to System registers that are not **UNDEFINED** at EL2.
- Execution of EL1 or EL2 translation regime address translation and TLB maintenance instructions for EL2 and above.

0b0 When this bit is set to 0, then the PE behaves as if HCR_EL2.NV2 is 0 for all purposes other than reading this register. This control does not cause any instructions to be trapped.

When HCR_EL2.NV2 is 1, no FEAT_NV2 functionality is implemented.

0b1 When HCR_EL2.NV2 is 0, or if FEAT_NV2 is not implemented, EL1 accesses to the specified registers or the execution of the specified instructions are trapped to EL2, when EL2 is enabled in the current Security state. EL1 read accesses to the CurrentEL register return a value of 0x2.

When HCR_EL2.NV2 is 1, this control redefines EL1 register accesses so that instructions accessing SPSR_EL2, ELR_EL2, ESR_EL2, and FAR_EL2 instead access SPSR_EL1, ELR_EL1, ESR_EL1, and FAR_EL1 respectively.

When HCR_EL2.NV2 is 0, or if FEAT_NV2 is not implemented, then:

- The System or Special-purpose registers for which accesses are trapped and reported using EC syndrome value 0x18 are as follows:
  - Registers accessed using MRS or MSR with a name ending in _EL2, except SP_EL2.
  - Registers accessed using MRS or MSR with a name ending in _EL12.
  - Registers accessed using MRS or MSR with a name ending in _EL02.
  - Special-purpose registers SPSR_irq, SPSR_abt, SPSR_und and SPSR_fiq, accessed using MRS or MSR.
  - Special-purpose register SP_EL1 accessed using the dedicated MRS or MSR instruction.

- The instructions for which the execution is trapped and reported using EC syndrome value 0x18 are as follows:
  - EL2 translation regime Address Translation instructions and TLB maintenance instructions.
  - EL1 translation regime Address Translation instructions and TLB maintenance instructions that are accessible only from EL2 and EL3.
The instructions for which the execution is trapped as follows:
- SMC in an implementation that does not include EL3 and when HCR_EL2.TSC is 1. HCR_EL2.TSC bit is not RES0 in this case. This is reported using EC syndrome value 0x17.
- The ERET, ERETTAA, and ERETTAB instructions, reported using EC syndrome value 0x1A.

--- Note ---

The priority of this trap is higher than the priority of the HCR_EL2.API trap. If both of these bits are set so that EL1 execution of an ERETTAA or ERETTAB instruction is trapped to EL2, then the syndrome reported is 0x1A.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*When FEAT_NV is implemented:*

Nested Virtualization. Traps functionality that is permitted at EL2 and would be UNDEFINED at EL1 if this field was 0, when EL2 is enabled in the current Security state. This applies to the following operations:
- EL1 accesses to Special-purpose registers that are not UNDEFINED at EL2.
- EL1 accesses to System registers that are not UNDEFINED at EL2.
- Execution of EL1 or EL2 translation regime address translation and TLB maintenance instructions for EL2 and above.

0b0  This control does not cause any instructions to be trapped.
0b1  EL1 accesses to the specified registers or the execution of the specified instructions are trapped to EL2, when EL2 is enabled in the current Security state. EL1 read accesses to the CurrentEL register return a value of 0x2.

The System or Special-purpose registers for which accesses are trapped and reported using EC syndrome value 0x18 are as follows:
- Registers accessed using MRS or MSR with a name ending in _EL2, except SP_EL2.
- Registers accessed using MRS or MSR with a name ending in _EL12.
- Registers accessed using MRS or MSR with a name ending in _EL02.
- Special-purpose registers SPSR_irq, SPSR_abt, SPSR_und and SPSR_fiq, accessed using MRS or MSR.
- Special-purpose register SP_EL1 accessed using the dedicated MRS or MSR instruction.

The instructions for which the execution is trapped and reported using EC syndrome value 0x18 are as follows:
- EL2 translation regime Address Translation instructions and TLB maintenance instructions.
- EL1 translation regime Address Translation instructions and TLB maintenance instructions that are accessible only from EL2 and EL3.

The execution of the ERET, ERETTAA, and ERETTAB instructions are trapped and reported using EC syndrome value 0x1A.

--- Note ---

The priority of this trap is higher than the priority of the HCR_EL2.API trap. If both of these bits are set so that EL1 execution of an ERETTAA or ERETTAB instruction is trapped to EL2, then the syndrome reported is 0x1A.

The execution of the SMC instructions in an implementation that does not include EL3 and when HCR_EL2.TSC is 1 are trapped and reported using EC syndrome value 0x17. HCR_EL2.TSC bit is not RES0 in this case.

This bit is permitted to be cached in a TLB.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

**API, bit [41]**

When *FEAT_PAuth* is implemented:

Controls the use of instructions related to Pointer Authentication:

- In EL0, when HCR_EL2.TGE==0 or HCR_EL2.E2H==0, and the associated SCTLR_EL1.En<N><M>=1.
- In EL1, the associated SCTLR_EL1.En<N><M>=1.

Traps are reported using EC syndrome value 0x09. The Pointer Authentication instructions trapped are:

- AUTDA, AUTDB, AUTDZA, AUTDZB, AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZA, AUTIZB.
- PACGA, PACDA, PACDB, PACDZA, PACDZB, PACIA, PACIA1716, PACIASP, PACIAZ, PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZA, PACIZB.
- RETAA, RETAB, BRAA, BRAB, BLRAA, BLRAB, BRAAZ, BRABZ, BLRAAZ, BLRABZ.
- ERETTAA, ERETTAB, LDRAA, and LDRAB.

0b0  The instructions related to Pointer Authentication are trapped to EL2, when EL2 is enabled in the current Security state and the instructions are enabled for the EL1&0 translation regime, from:

- EL0 when HCR_EL2.TGE==0 or HCR_EL2.E2H==0.
- EL1.

If HCR_EL2.NV is 1, the HCR_EL2.NV trap takes precedence over the HCR_EL2.API trap for the ERETTA and ERETTB instructions.

If EL2 is implemented and enabled in the current Security state and *HFGITR_EL2.ERET==1*, execution at EL1 using AArch64 of ERETTA or ERETTB instructions is reported with EC syndrome value 0x1A with its associated ISS field, as the fine-grained trap has higher priority than the HCR_EL2.API == 0.

0b1  This control does not cause any instructions to be trapped.

If FEAT_PAuth is implemented but EL2 is not implemented or disabled in the current Security state, the system behaves as if this bit is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

**APK, bit [40]**

When *FEAT_PAuth* is implemented:

Trap registers holding "key" values for Pointer Authentication. Traps accesses to the following registers from EL1 to EL2, when EL2 is enabled in the current Security state, reported using EC syndrome value 0x18:

- APIAKeyLo_EL1, APIAKeyHi_EL1, APIBKeyLo_EL1, APIBKeyHi_EL1, APDAKeyLo_EL1, APDAKeyHi_EL1, APDBKeyLo_EL1, APDBKeyHi_EL1, APGAKeyLo_EL1, and APGAKeyHi_EL1.

0b0  Access to the registers holding "key" values for pointer authentication from EL1 are trapped to EL2, when EL2 is enabled in the current Security state.

0b1  This control does not cause any instructions to be trapped.
Note

If FEAT_PAuth is implemented but EL2 is not implemented or is disabled in the current Security state, the system behaves as if this bit is 1.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

TME, bit [39]

When FEAT_TME is implemented:
Enables access to the TSTART, TCOMMIT, TTEST, and TCANCEL instructions at EL0 and EL1.
- 0b0 EL0 and EL1 accesses to TSTART, TCOMMIT, TTEST, and TCANCEL instructions are UNDEFINED.
- 0b1 This control does not cause any instruction to be UNDEFINED.

If EL2 is not implemented or is disabled in the current Security state, the Effective value of this bit is 0b1.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

MIOCNCE, bit [38]

Mismatched Inner/Outer Cacheable Non-Coherency Enable, for the EL1&0 translation regimes.
- 0b0 For the EL1&0 translation regimes, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there must be no loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.
- 0b1 For the EL1&0 translation regimes, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there might be a loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

For more information, see Mismatched memory attributes on page B2-208.
This field can be implemented as RAZ/WI.
When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TEA, bit [37]

When FEAT_RAS is implemented:
Route synchronous External abort exceptions to EL2.
- 0b0 This control does not cause exceptions to be routed from EL0 and EL1 to EL2.
- 0b1 Route synchronous External abort exceptions from EL0 and EL1 to EL2, when EL2 is enabled in the current Security state, if not routed to EL3.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.
TERR, bit [36]

When FEAT_RAS is implemented:

Trap Error record accesses. Trap accesses to the RAS error registers from EL1 to EL2 as follows:

- If EL1 is using AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value 0x18:
  - ERRIDR_EL1, ERRSELR_EL1, ERXADDR_EL1, ERXCTLR_EL1, ERXF_R_EL1, ERXMISC0_EL1, ERXMISC1_EL1, and ERXSTATUS_EL1.
  - When FEAT_RASv1p1 is implemented, ERXMISC2_EL1, and ERXMISC3_EL1.

- If EL1 is using AArch32 state, MCR or MRC accesses are trapped to EL2, reported using EC syndrome value 0x03, MCRR or MRRC accesses are trapped to EL2, reported using EC syndrome value 0x04:
  - ERRIDR, ERRSELR, ERXADDR, ERXADDR2, ERXCTLR, ERXCTLR2, ERXF_R, ERXF_R2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, and ERXSTATUS.
  - When FEAT_RASv1p1 is implemented, ERXMISC4, ERXMISC5, ERXMISC6, and ERXMISC7.

0b0 This control does not cause any instructions to be trapped.
0b1 Accesses to the specified registers from EL1 generate a Trap exception to EL2, when EL2 is enabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLOR, bit [35]

When FEAT_LOR is implemented:

Trap LOR registers. Traps Non-secure EL1 accesses to LORSA_EL1, LOREA_EL1, LORN_EL1, LORC_EL1, and LORID_EL1 registers to EL2.

0b0 This control does not cause any instructions to be trapped.
0b1 Non-secure EL1 accesses to the LOR registers are trapped to EL2.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E2H, bit [34]

When FEAT_VHE is implemented:

EL2 Host. Enables a configuration where a Host Operating System is running in EL2, and the Host Operating System's applications are running in EL0.

0b0 The facilities to support a Host Operating System at EL2 are disabled.
0b1 The facilities to support a Host Operating System at EL2 are enabled.

For information on the behavior of this bit see Behavior of HCR_EL2.E2H on page D8-5168.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
ID, bit [33]
Stage 2 Instruction access cacheability disable. For the EL1&0 translation regime, when EL2 is enabled in the current Security state and HCR_EL2.VM==1, this control forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

0b0  This control has no effect on stage 2 of the EL1&0 translation regime.
0b1  Forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

This bit has no effect on the EL2, EL2&0, or EL3 translation regimes.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:
•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

CD, bit [32]
Stage 2 Data access cacheability disable. For the EL1&0 translation regime, when EL2 is enabled in the current Security state and HCR_EL2.VM==1, this control forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable.

0b0  This control has no effect on stage 2 of the EL1&0 translation regime for data accesses and translation table walks.
0b1  Forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable.

This bit has no effect on the EL2, EL2&0, or EL3 translation regimes.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:
•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

RW, bit [31]

When EL1 is capable of using AArch32:
Execution state control for lower Exception levels:

0b0  Lower levels are all AArch32.
0b1  The Execution state for EL1 is AArch64. The Execution state for EL0 is determined by the current value of PSTATE.nRW when executing at EL0.

In an implementation that includes EL3, when EL2 is not enabled in Secure state, the PE behaves as if this bit has the same value as the SCR_EL3.RW bit for all purposes other than a direct read or write access of HCR_EL2.

The RW bit is permitted to be cached in a TLB.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 1 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:
•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RAO/WI.
TRVM, bit [30]

Trap Reads of Virtual Memory controls. Traps EL1 reads of the virtual memory control registers to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64 state, the following registers are trapped to EL2 and reported using EC syndrome value 0x18.
  - SCTLR_EL1, TTBR0_EL1, TTBR1_EL1, TCR_EL1, ESR_EL1, FAR_EL1, AFSR0_EL1, AFSR1_EL1, MAIR_EL1, AMAIR_EL1, CONTEXTIDR_EL1.
- If EL1 is using AArch32 state, accesses using MRC to the following registers are trapped to EL2 and reported using EC syndrome value 0x03, accesses using MRRC are trapped to EL2 and reported using EC syndrome value 0x04:
  - SCTLR, TTBR0, TTBR1, TTBCR, TTBCR2, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR.

0b0  This control does not cause any instructions to be trapped.
0b1  EL1 read accesses to the specified Virtual Memory controls are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

——— Note ————

EL2 provides a second stage of address translation, that a hypervisor can use to remap the address map defined by a Guest OS. In addition, a hypervisor can trap attempts by a Guest OS to write to the registers that control the memory system. A hypervisor might use this trap as part of its virtualization of memory management.

——————

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

HCD, bit [29]

When EL3 is not implemented:

HVC instruction disable. Disables EL1 execution of HVC instructions, from both Execution states, when EL2 is enabled in the current Security state, reported using EC syndrome value 0x00.

0b0  HVC instruction execution is enabled at EL2 and EL1.
0b1  HVC instructions are UNDEFINED at EL2 and EL1. Any resulting exception is taken to the Exception level at which the HVC instruction is executed.

——— Note ————

HVC instructions are always UNDEFINED at EL0.

——————

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TDZ, bit [28]

Trap DC ZVA instructions. Traps EL0 and EL1 execution of DC ZVA instructions to EL2, when EL2 is enabled in the current Security state, from AArch64 state only, reported using EC syndrome value 0x18.

If FEAT_MTE is implemented, this trap also applies to DC GVA and DC GZVA.

0b0  This control does not cause any instructions to be trapped.
In AArch64 state, any attempt to execute an instruction this trap applies to at EL1, or at EL0 when the instruction is not UNDEFINED at EL0, is trapped to EL2 when EL2 is enabled in the current Security state.

Reading the DCZID_EL0 returns a value that indicates that the instructions this trap applies to are not supported.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is \{1, 1\}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**TGE, bit [27]**

Trap General Exceptions, from EL0.

- **0b0** This control has no effect on execution at EL0.
- **0b1** When EL2 is not enabled in the current Security state, this control has no effect on execution at EL0.

When EL2 is enabled in the current Security state, in all cases:

- All exceptions that would be routed to EL1 are routed to EL2.
- If EL1 is using AArch64, the SCTLR_EL1.M field is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR_EL1.
- If EL1 is using AArch32, the SCTLR.M field is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR.
- All virtual interrupts are disabled.
- Any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts are disabled.
- An exception return to EL1 is treated as an illegal exception return.
- The MDCR_EL2.{TDRA, TDOSA, TDA, TDE} fields are treated as being 1 for all purposes other than returning the result of a direct read of MDCR_EL2.

In addition, when EL2 is enabled in the current Security state, if:

- HCR_EL2.E2H is 0, the Effective values of the HCR_EL2.{FMO, IMO, AMO} fields are 1.
- HCR_EL2.E2H is 1, the Effective values of the HCR_EL2.{FMO, IMO, AMO} fields are 0.

For further information on the behavior of this bit when E2H is 1, see *Behavior of HCR_EL2.E2H* on page D8-5168.

HCR_EL2.TGE must not be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**TVM, bit [26]**

Trap Virtual Memory controls. Traps EL1 writes to the virtual memory control registers to EL2, when EL2 is enabled in the current Security state, as follows:

- If EL1 is using AArch64 state, the following registers are trapped to EL2 and reported using EC syndrome value 0x18:
  - SCTLR_EL1, TTBR0_EL1, TTBR1_EL1, TCR_EL1, ESR_EL1, FAR_EL1, AFSR0_EL1, AFSR1_EL1, MAIR_EL1, AMAIR_EL1, CONTEXTIDR_EL1.
• If EL1 is using AArch32 state, accesses using MCR to the following registers are trapped to EL2 and reported using EC syndrome value 0x03, accesses using MCRR are trapped to EL2 and reported using EC syndrome value 0x04:
  
  — SCTLR, TTBR0, TTBR1, TTBCR, TTBCR2, DACR, DFRS, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR.

0b0  This control does not cause any instructions to be trapped.

0b1  EL1 write accesses to the specified EL1 virtual memory control registers are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

TTLB, bit [25]

Trap TLB maintenance instructions. Traps EL1 execution of TLB maintenance instructions to EL2, when EL2 is enabled in the current Security state, as follows:

• When EL1 is using AArch64 state, the following instructions are trapped to EL2 and reported using EC syndrome value 0x18:
  
  — TLBI VMALLE1, TLBI VMALLE1NXS, TLBI VAE1, TLBI VAE1NXS, TLBI ASIDE1, TLBI ASIDE1NXS, TLBI VAAE1, TLBI VAAC1NXS, TLBI VALE1, TLBI VALE1NXS, TLBI VAAE1, TLBI VALE1NXS.

  — TLBI VMALLE1IS, TLBI VMALLE1ISNXS, TLBI VAEE1IS, TLBI VAEE1ISNXS, TLBI ASIDE1IS, TLBI ASIDE1ISNXS, TLBI VAAE1IS, TLBI VAAE1ISNXS, TLBI VAAE1IS, TLBI VAAE1ISNXS.

  — If FEAT_TLBIOS is implemented, this trap applies to TLBI VMALLE1OS, TLBI VMALLE1OSNXS, TLBI VAE1OS, TLBI VAE1OSNXS, TLBI ASIDE1OS, TLBI ASIDE1OSNXS, TLBI VAAE1OS, TLBI VAAE1OSNXS, TLBI VAAE1OS, TLBI VAAE1OSNXS.

  — If FEAT_TLBIRANGE is implemented, this trap applies to TLBI RVAE1, TLBI RVAE1NXS, TLBI RVAE1NXS, TLBI RVVAE1NXS, TLBI RVVAE1NXS, TLBI RVAAE1NXS, TLBI RVAAE1NXS, TLBI RVAAE1NXS, TLBI RVAAE1NXS.

  — If FEAT_TLBIRANGE is implemented, this trap applies to TLBI RVAAE1OS, TLBI RVAAE1OSNXS, TLBI RVAAE1OSNXS, TLBI RVAAE1OSNXS, TLBI RVAAE1OSNXS, TLBI RVAAE1OSNXS, TLBI RVAAE1OSNXS, TLBI RVAAE1OSNXS.

• When EL1 is using AArch32 state, the following instructions are trapped to EL2 and reported using EC syndrome value 0x03:
  
  — TLBIALLIS, TLBIMVAIS, TLBIASDIS, TLBIMVAAS, TLBIMVALIS, TLBIMVALLS.

  — TLBIALL, TLBIMVA, TLBIASID, TLBIMVAA, TLBIMVAL, TLBIMVALL.

  — ITLBIAII, ITLBIMVA, ITLBIAIS.

  — DTLBIAII, DTLBIMVA, DTLBIAIS.

0b0  This control does not cause any instructions to be trapped.

0b1  EL1 execution of the specified TLB maintenance instructions are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.
--- Note ---
The TLB maintenance instructions are UNDEFINED at EL0.

---

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TPU, bit [24]
Trap cache maintenance instructions that operate to the Point of Unification. Traps execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state as follows:

- If EL0 is using AArch64 state and the value of SCTLR_EL1.UCI is not 0, the following instructions are trapped to EL2 and reported with EC syndrome value 0x18:
  - IC IVAU, DC CVAU. If the value of SCTLR_EL1.UCI is 0 these instructions are UNDEFINED at EL0 and any resulting exception is higher priority than this trap to EL2.
- If EL1 is using AArch64 state, the following instructions are trapped to EL2 and reported with EC syndrome value 0x18:
  - IC IVAU, IC IALLU, IC IALLUIS, DC CVAU.
- If EL1 is using AArch32 state, the following instructions are trapped to EL2 and reported with EC syndrome value 0x18:
  - ICIMV AU, IC IALLU, IC IALLUIS, DCCMV AU.

--- Note ---
An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:

- IC IALLUIS and IC IALLU are always UNDEFINED at EL0 using AArch64.
- ICIMV AU, IC IALLU, IC IALLUIS, and DCCMV AU are always UNDEFINED at EL0 using AArch32.

0b0 This control does not cause any instructions to be trapped.
0b1 Execution of the specified instructions is trapped to EL2, when EL2 is enabled in the current Security state.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TPCP, bit [23]

When FEAT_DPB is implemented:

TPCP
Trap data or unified cache maintenance instructions that operate to the Point of Coherency or Persistence. Traps execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state as follows:

- If EL0 is using AArch64 state and the value of SCTLR_EL1.UCI is not 0, the following instructions are trapped to EL2 and reported using EC syndrome value 0x18:
  - DC CIVAC, DC CVAC, DC CVAP. If the value of SCTLR_EL1.UCI is 0 these instructions are UNDEFINED at EL0 and any resulting exception is higher priority than this trap to EL2.
- If EL1 is using AArch64 state, the following instructions are trapped to EL2 and reported using EC syndrome value 0x83:
  - DC IVAC, DC CIVAC, DC CVAC, DC CVAP.
- If EL1 is using AArch32 state, the following instructions are trapped to EL2 and reported using EC syndrome value 0x03:
  - DCIMVAC, DCCIMVAC, DCCMVAC.

If FEAT_DPB2 is implemented, this trap also applies to DC CVADP.
If FEAT_MTE is implemented, this trap also applies to DC CIGVAC, DC CIGDVAC, DC IGVAC, DC IGDVAC, DC CGVAC, DC CGDVAC, DC CGVAP and DC CGDVAP.
If FEAT_DPB2 and FEAT_MTE are implemented, this trap also applies to DC CGVADP and DC CGDVADP.

**Note**
- An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:
  - AArch64 instructions which invalidate by VA to the Point of Coherency are always UNDEFINED at EL0 using AArch64.
  - DCIMVAC, DCCIMVAC, and DCCMVAC are always UNDEFINED at EL0 using AArch32.
- In Armv8.0 and Armv8.1, this field is named TPC. From Armv8.2, it is named TPCP.

- **0b0** This control does not cause any instructions to be trapped.
- **0b1** Execution of the specified instructions is trapped to EL2, when EL2 is enabled in the current Security state.

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, invalidate, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 1.

If HCR_EL2.{E2H, TGE} is set to {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

**TPC**

Trap data or unified cache maintenance instructions that operate to the Point of Coherency. Traps execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state as follows:

- If EL0 is using AArch64 state and the value of SCTLR_EL1.UCI is not 0, accesses to the following registers are trapped and reported using EC syndrome value 0x18:
  - DC CIVAC, DC CVAC. However, if the value of SCTLR_EL1.UCI is 0 these instructions are UNDEFINED at EL0 and any resulting exception is higher priority than this trap to EL2.
- If EL1 is using AArch64 state, accesses to DC IVAC, DC CIVAC, DC CVAC are trapped and reported using EC syndrome value 0x18.
• When EL1 is using AArch32, accesses to DCIMVAC, DCCIMVAC, and DCCMVAC are trapped and reported using EC syndrome value 0x03.

——— Note ————
• An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:
  — AArch64 instructions which invalidate by VA to the Point of Coherency are always UNDEFINED at EL0 using AArch64.
  — DCIMVAC, DCCIMVAC, and DCCMVAC are always UNDEFINED at EL0 using AArch32.
• In Armv8.0 and Armv8.1, this field is named TPC. From Armv8.2, it is named TPCP.

0b0  This control does not cause any instructions to be trapped.
0b1  Execution of the specified instructions is trapped to EL2, when EL2 is enabled in the current Security state.

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, invalidate, or clean and invalidate instruction that operates by VA to the Point of coherency can be trapped when the value of this control is 1.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

TSW, bit [22]
Trap data or unified cache maintenance instructions that operate by Set/Way. Traps execution of those cache maintenance instructions at EL1 to EL2, when EL2 is enabled in the current Security state as follows:
• If EL1 is using AArch64 state, accesses to DCISW, DCCSW, DC CISW are trapped to EL2, reported using EC syndrome value 0x18.
• If EL1 is using AArch32 state, accesses to DCISW, DCCSW, DCCISW are trapped to EL2, reported using EC syndrome value 0x83.

If FEAT_MTE is implemented, this trap also applies to DC IGSW, DC IGDSW, DC CGSW, DC CGDSW, DC CIGSW, and DC CIGDSW.

——— Note ————
An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

0b0  This control does not cause any instructions to be trapped.
0b1  Execution of the specified instructions is trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

TACR, bit [21]
Trap Auxiliary Control Registers. Traps EL1 accesses to the Auxiliary Control Registers to EL2, when EL2 is enabled in the current Security state, as follows:
• If EL1 is using AArch64 state, accesses to ACTLR_EL1 to EL2, are trapped to EL2 and reported using EC syndrome value 0x18.
• If EL1 is using AArch32 state, accesses to ACTLR and, if implemented, ACTLR2 are trapped to EL2 and reported using EC syndrome value 0x03.

0b0  This control does not cause any instructions to be trapped.

0b1  EL1 accesses to the specified registers are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

——— Note ————
ACTLR_EL1 is not accessible at EL0.
ACTLR and ACTLR2 are not accessible at EL0.

The Auxiliary Control Registers are IMPLEMENTATION DEFINED registers that might implement global control bits for the PE.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

TIDCP, bit [20]

Trap IMPLEMENTATION DEFINED functionality. Traps EL1 accesses to the encodings reserved for IMPLEMENTATION DEFINED functionality to EL2, when EL2 is enabled in the current Security state as follows:

• In AArch64 state, access to any of the encodings in the following reserved encoding spaces are trapped and reported using EC syndrome 0x18:
  — IMPLEMENTATION DEFINED System instructions, which are accessed using SYS and SYSL, with CRn == {11, 15}.
  — IMPLEMENTATION DEFINED System registers, which are accessed using MRS and MSR with the S3_<op1>_<Cn>_<Cm>_<op2> register name.

• In AArch32 state, MCR and MRC access to instructions with the following encodings are trapped and reported using EC syndrome 0x03:
  — All coproc==p15, CRn==c9, opc1 == {0-7}, CRm == {c0-c2, c5-c8}, opc2 == {0-7}.
  — All coproc==p15, CRn==c10, opc1 =={0-7}, CRm == {c0, c1, c4, c8}, opc2 == {0-7}.
  — All coproc==p15, CRn==c11, opc1=={0-7}, CRm == {c0-c8, c15}, opc2 == {0-7}.

When this functionality is accessed from EL0:
• If FEAT_TIDCP1 is implemented and the Effective value of SCTLR_EL1.TIDCP is 1, any accesses from EL0 are trapped to EL1.
• Otherwise, if FEAT_TIDCP1 is implemented and the Effective value of SCTLR_EL2.TIDCP is 1, any accesses from EL0 are trapped to EL2.
• Otherwise:
  — If HCR_EL2.TIDCP is 1, it is IMPLEMENTATION DEFINED whether any accesses from EL0 are trapped to EL2.
  — If HCR_EL2.TIDCP is 0, any accesses from EL0 are UNDEFINED and generate an exception that is taken to EL1 or EL2.

0b0  This control does not cause any instructions to be trapped.

0b1  EL1 accesses to or execution of the specified encodings reserved for IMPLEMENTATION DEFINED functionality are trapped to EL2, when EL2 is enabled in the current Security state.

An implementation can also include IMPLEMENTATION DEFINED registers that provide additional controls, to give finer-grained control of the trapping of IMPLEMENTATION DEFINED features.
### Note

The trapping of accesses to these registers from EL1 is higher priority than an exception resulting from the register access being UNDEFINED.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### TSC, bit [19]

Trap SMC instructions. Traps EL1 execution of SMC instructions to EL2, when EL2 is enabled in the current Security state.

If execution is in AArch64 state, the trap is reported using EC syndrome value 0x17.

If execution is in AArch32 state, the trap is reported using EC syndrome value 0x13.

#### Note

HCR_EL2.TSC traps execution of the SMC instruction. It is not a routing control for the SMC exception. Trap exceptions and SMC exceptions have different preferred return addresses.

0b0 This control does not cause any instructions to be trapped.

0b1 If EL3 is implemented, then any attempt to execute an SMC instruction at EL1 is trapped to EL2, when EL2 is enabled in the current Security state, regardless of the value of SCR_EL3.SMD.

If EL3 is not implemented, FEAT_NV is implemented, and HCR_EL2.NV is 0, then any attempt to execute an SMC instruction at EL1 using AArch64 is trapped to EL2, when EL2 is enabled in the current Security state.

If EL3 is not implemented, and either FEAT_NV is not implemented or HCR_EL2.NV is 0, then it is IMPLEMENTATION DEFINED whether:
- Any attempt to execute an SMC instruction at EL1 is trapped to EL2, when EL2 is enabled in the current Security state.
- Any attempt to execute an SMC instruction is UNDEFINED.

In AArch32 state, the Armv8-A architecture permits, but does not require, this trap to apply to conditional SMC instructions that fail their condition code check, in the same way as with traps on other conditional instructions.

SMC instructions are UNDEFINED at EL0.

If EL3 is not implemented, and either FEAT_NV is not implemented or HCR_EL2.NV is 0, then it is IMPLEMENTATION DEFINED whether this bit is:
- RES0.
- Implemented with the functionality as described in HCR_EL2.TSC.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### TID3, bit [18]

Trap ID group 3. Traps EL1 reads of group 3 ID registers to EL2, when EL2 is enabled in the current Security state, as follows:

In AArch64 state:
- Reads of the following registers are trapped to EL2, reported using EC syndrome value 0x18:
  - ID_PFR0_EL1, ID_PFR1_EL1, ID_PFR2_EL1, ID_DFR0_EL1, ID_AFR0_EL1, ID_MMFRO_EL1, ID_MMFR1_EL1, ID_MMFR2_EL1, ID_MMFR3_EL1, ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, ID_ISAR5_EL1, MVFR0_EL1, MVFR1_EL1, MVFR2_EL1.
ID_AA64PFR0_EL1, ID_AA64PFR1_EL1, ID_AA64DFR0_EL1, ID_AA64DFR1_EL1, ID_AA64ISAR0_EL1, ID_AA64ISAR1_EL1, ID_AA64MMFR0_EL1, ID_AA64MMFR1_EL1, ID_AA64AFR0_EL1, ID_AA64AFR1_EL1.

If FEAT_FGT is implemented:

- ID_MMFR4_EL1 and ID_MMFR5_EL1 are trapped to EL2.
- ID_AA64MMFR2_EL1 and ID_ISAR6_EL1 are trapped to EL2.
- ID_DFR1_EL1 is trapped to EL2.
- ID_AA64ZFR0_EL1 is trapped to EL2.
- ID_AA64SMFR0_EL1 is trapped to EL2.
- ID_AA64ISAR2_EL1 is trapped to EL2.

This field traps all MRS accesses to registers in the following range that are not already mentioned in this field description: Op0 == 3, op1 == 0, CRn == c0, CRm == \{c1-c7\}, op2 == \{0-7\}.

If FEAT_FGT is not implemented:

- ID_MMFR4_EL1 and ID_MMFR5_EL1 are trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_MMFR4_EL1 or ID_MMFR5_EL1 are trapped to EL2.
- ID_AA64MMFR2_EL1 and ID_ISAR6_EL1 are trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID-AA64MMFR2_EL1 or ID_ISAR6_EL1 are trapped to EL2.
- ID_DFR1_EL1 is trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_DFR1_EL1 are trapped to EL2.
- ID_AA64ZFR0_EL1 is trapped to EL2, unless implemented as RAZ then it is IMPLEMENTATION DEFINED whether accesses to ID_AA64ZFR0_EL1 are trapped to EL2.
- ID_AA64SMFR0_EL1 is trapped to EL2, unless implemented as RAZ then it is IMPLEMENTATION DEFINED whether accesses to ID_AA64SMFR0_EL1 are trapped to EL2.
- ID_AA64ISAR2_EL1 is trapped to EL2, unless implemented as RAZ then it is IMPLEMENTATION DEFINED whether accesses to ID_AA64ISAR2_EL1 are trapped to EL2.

Otherwise, it is IMPLEMENTATION DEFINED whether this bit traps MRS accesses to registers in the following range that are not already mentioned in this field description: Op0 == 3, op1 == 0, CRn == c0, CRm == \{c1-c7\}, op2 == \{0-7\}.

In AArch32 state:

- VMRS access to MVFR0, MVFR1, and MVFR2, are trapped to EL2, reported using EC syndrome value 0x08, unless access is also trapped by HCPTR which takes priority.
- MRC access to the following registers are trapped to EL2, reported using EC syndrome value 0x03:
  - ID_PFR0, ID_PFR1, ID_PFR2, ID_DFR0, ID_AFR0, ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, ID_ISAR5.

  If FEAT_FGT is implemented:
  - ID_MMFR4 and ID_MMFR5 are trapped to EL2.
  - ID_ISAR6 is trapped to EL2.
  - ID_DFR1 is trapped to EL2.
— This field traps all MRC accesses to encodings in the following range that are
not already mentioned in this field description: coproc == p15, opc1 == 0, CRn
== c0, CRm == {c2-c7}, opc2 == {0-7}.

— If FEAT_FGT is not implemented:
  — ID_MMFR4 and ID_MMFR5 are trapped to EL2, unless implemented as RAZ,
    when it is IMPLEMENTATION DEFINED whether accesses to ID_MMFR4 or
    ID_MMFR5 are trapped.
  — ID_ISAR6 is trapped to EL2, unless implemented as RAZ, when it is
    IMPLEMENTATION DEFINED whether accesses to ID_ISAR6 are trapped to EL2.
  — ID_DFR1 is trapped to EL2, unless implemented as RAZ, when it is
    IMPLEMENTATION DEFINED whether accesses to ID_DFR1 are trapped to EL2.
  — Otherwise, it is IMPLEMENTATION DEFINED whether this bit traps all MRC
    accesses to registers in the following range not already mentioned in this field
    description with coproc == p15, opc1 == 0, CRn == c0, CRm == {c2-c7}, opc2
    == {0-7}.

0b0  This control does not cause any instructions to be trapped.
0b1  The specified EL1 read accesses to ID group 3 registers are trapped to EL2, when EL2
     is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct
read of this field.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

TID2, bit [17]
Trap ID group 2. Traps the following register accesses to EL2, when EL2 is enabled in the current
Security state, as follows:
• If EL1 is using AArch64, reads of CTR_EL0, CCSIDR_EL1, CCSIDR2_EL1, CLIDR_EL1,
  and CSSELR_EL1 are trapped to EL2, reported using EC syndrome value 0x18.
• If EL0 is using AArch64 and the value of SCTLR_EL1.UCT is not 0, reads of CTR_EL0 are
  trapped to EL2, reported using EC syndrome value 0x18. If the value of SCTLR_EL1.UCT
  is 0, then EL0 reads of CTR_EL0 are trapped to EL1 and the resulting exception takes
  precedence over this trap.
• If EL1 is using AArch64, writes to CSSELR_EL1 are trapped to EL2, reported using EC
  syndrome value 0x18.
• If EL1 is using AArch32, reads of CTR, CCSIDR, CCSIDR2, CLIDR, and CSSELR are
  trapped to EL2, reported using EC syndrome value 0x03.
• If EL1 is using AArch32, writes to CSSELR are trapped to EL2, reported using EC syndrome
  value 0x03.

0b0  This control does not cause any instructions to be trapped.
0b1  The specified EL1 and EL0 accesses to ID group 2 registers are trapped to EL2, when
     EL2 is enabled in the current Security state.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field
behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

TID1, bit [16]
Trap ID group 1. Traps EL1 reads of the following registers to EL2, when EL2 is enabled in the current
Security state as follows:
• In AArch64 state, accesses of REVIDR_EL1, AIDR_EL1, SMIDR_EL1, reported using EC
  syndrome value 0x18.
In AArch32 state, accesses of TCMTR, TLBTR, REVIDR, AIDR, reported using EC syndrome value 0x03.

0b0  This control does not cause any instructions to be trapped.
0b1  The specified EL1 read accesses to ID group 1 registers are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TID0, bit [15]

When AArch32 is supported:

Trap ID group 0. Traps the following register accesses to EL2:

- EL1 reads of the JIDR, reported using EC syndrome value 0x05.
- If the JIDR is RAZ from EL0, EL0 reads of the JIDR, reported using EC syndrome value 0x05.
- EL1 accesses using VMRS of the FPSID, reported using EC syndrome value 0x08.

--- Note ---

- It is IMPLEMENTATION DEFINED whether the JIDR is RAZ or UNDEFINED at EL0. If it is UNDEFINED at EL0, then any resulting exception takes precedence over this trap.
- The FPSID is not accessible at EL0 using AArch32.
- Writes to the FPSID are ignored, and not trapped by this control.

0b0  This control does not cause any instructions to be trapped.
0b1  The specified EL1 read accesses to ID group 0 registers are trapped to EL2, when EL2 is enabled in the current Security state.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TWE, bit [14]

Traps EL0 and EL1 execution of WFE instructions to EL2, when EL2 is enabled in the current Security state, from both Execution states, reported using EC syndrome value 0x01.

When FEAT_WFxT is implemented, this trap also applies to the WFET instruction.

0b0  This control does not cause any instructions to be trapped.
0b1  Any attempt to execute a WFE instruction at EL0 or EL1 is trapped to EL2, when EL2 is enabled in the current Security state, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWE or SCTLR_EL1.nTWE.

In AArch32 state, the attempted execution of a conditional WFE instruction is trapped only if the instruction passes its condition code check.
Note
Since a WFE can complete at any time, even without a Wakeup event, the traps on WFE are not guaranteed to be taken, even if the WFE is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is \{1, 1\}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

For more information about when WFE instructions can cause the PE to enter a low-power state, see Wait for Event on page D1-4676.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TWI, bit [13]
Traps EL0 and EL1 execution of WFI instructions to EL2, when EL2 is enabled in the current Security state, from both Execution states, reported using EC syndrome value 0x01.

When FEAT_WFxT is implemented, this trap also applies to the WFIT instruction.

0b0  This control does not cause any instructions to be trapped.

0b1  Any attempt to execute a WFI instruction at EL0 or EL1 is trapped to EL2, when EL2 is enabled in the current Security state, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWI or SCTLR_EL1.nTWI.

In AArch32 state, the attempted execution of a conditional WFI instruction is trapped only if the instruction passes its condition code check.

Note
Since a WFI can complete at any time, even without a Wakeup event, the traps on WFI are not guaranteed to be taken, even if the WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is \{1, 1\}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

For more information about when WFI instructions can cause the PE to enter a low-power state, see Wait for Interrupt mechanism on page D1-4678.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DC, bit [12]
Default Cacheability.

0b0  This control has no effect on the EL1&0 translation regime.

0b1  In any Security state:

- When EL1 is using AArch64, the PE behaves as if the value of the SCTLR_EL1.M field is 0 for all purposes other than returning the value of a direct read of SCTLR_EL1.

- When EL1 is using AArch32, the PE behaves as if the value of the SCTLR.M field is 0 for all purposes other than returning the value of a direct read of SCTLR.

- The PE behaves as if the value of the HCR_EL2.VM field is 1 for all purposes other than returning the value of a direct read of HCR_EL2.

- The memory type produced by stage 1 of the EL1&0 translation regime is Normal Non-Shareable, Inner Write-Back Read-Allocate Write-Allocate, Outer Write-Back Read-Allocate Write-Allocate.
This field has no effect on the EL2, EL2&0, and EL3 translation regimes.
This bit is permitted to be cached in a TLB.
When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is \{1, 1\}, this field behaves as 0 for all purposes other than a direct read of the value of this field.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**BSU, bits [11:10]**
Barrier Shareability upgrade. This field determines the minimum shareability domain that is applied to any barrier instruction executed from EL1 or EL0:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No effect.</td>
</tr>
<tr>
<td>01</td>
<td>Inner Shareable.</td>
</tr>
<tr>
<td>10</td>
<td>Outer Shareable.</td>
</tr>
<tr>
<td>11</td>
<td>Full system.</td>
</tr>
</tbody>
</table>

This value is combined with the specified level of the barrier held in its instruction, using the same principles as combining the shareability attributes from two stages of address translation.
When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is \{1, 1\}, this field behaves as 00 for all purposes other than a direct read of the value of this bit.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**FB, bit [9]**
Force broadcast. Causes the following instructions to be broadcast within the Inner Shareable domain when executed from EL1:

- AArch32: BPIALL, TLBIALL, TLBIMVA, TLBIASID, DTLBIALL, DTLBIMVA, DTLBIASID, ITLBIALL, ITLBIMVA, ITLBIASID, TLBIMVA, ICIALLU, TLBIMVAL, TLBIMVAAL.
- AArch64: TLB VMALLE1, TLB VMALLE1NXS, TLB VAE1, TLB VAE1NXS, TLB ASIDE1, TLB ASIDE1NXS, TLB VAAE1, TLB VAAE1NXS, TLB VALE1, TLB VALE1NXS, TLB VAAE1, TLB VAAE1NXS, TLB VAE1, TLB VAE1NXS, TLB VALE1, TLB VALE1NXS, TLB VAAE1NXS, TLB VAAE1NXS, TLB VAE1NXS, TLB VALE1NXS.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>This field has no effect on the operation of the specified instructions.</td>
</tr>
<tr>
<td>1</td>
<td>When one of the specified instruction is executed at EL1, the instruction is broadcast within the Inner Shareable shareability domain.</td>
</tr>
</tbody>
</table>

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**VSE, bit [8]**
Virtual SError interrupt.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>This mechanism is not making a virtual SError interrupt pending.</td>
</tr>
<tr>
<td>1</td>
<td>A virtual SError interrupt is pending because of this mechanism.</td>
</tr>
</tbody>
</table>

The virtual SError interrupt is enabled only when the value of HCR_EL2.{TGE, AMO} is \{0, 1\}.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**VI, bit [7]**
Virtual IRQ Interrupt.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>This mechanism is not making a virtual IRQ pending.</td>
</tr>
</tbody>
</table>

VF, bit [6]

Virtual FIQ Interrupt.

0b0  This mechanism is not making a virtual FIQ pending.

0b1  A virtual FIQ is pending because of this mechanism.

The virtual FIQ is enabled only when the value of HCR_EL2.{TGE, IMO} is {0, 1}.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

AMO, bit [5]

Physical SError interrupt routing.

0b0  When executing at Exception levels below EL2, and EL2 is enabled in the current Security state:
• When the value of HCR_EL2.TGE is 0, Physical SError interrupts are not taken to EL2.
• When the value of HCR_EL2.TGE is 1, Physical SError interrupts are taken to EL2 unless they are routed to EL3.
• Virtual SError interrupts are disabled.

0b1  When executing at any Exception level, and EL2 is enabled in the current Security state:
• Physical SError interrupts are taken to EL2, unless they are routed to EL3.
• When the value of HCR_EL2.TGE is 0, then virtual SError interrupts are enabled.

If EL2 is enabled in the current Security state and the value of HCR_EL2.TGE is 1:
• Regardless of the value of the AMO bit physical asynchronous External aborts and SError interrupts target EL2 unless they are routed to EL3.
• When FEAT_VHE is not implemented, or if HCR_EL2.E2H is 0, this field behaves as 1 for all purposes other than a direct read of the value of this bit.
• When FEAT_VHE is implemented and HCR_EL2.E2H is 1, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

For more information, see Establishing the target Exception level of an asynchronous exception on page D1-4657.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMO, bit [4]

Physical IRQ Routing.

0b0  When executing at Exception levels below EL2, and EL2 is enabled in the current Security state:
• When the value of HCR_EL2.TGE is 0, Physical IRQ interrupts are not taken to EL2.
• When the value of HCR_EL2.TGE is 1, Physical IRQ interrupts are taken to EL2 unless they are routed to EL3.
• Virtual IRQ interrupts are disabled.

0b1  When executing at any Exception level, and EL2 is enabled in the current Security state:
• Physical IRQ interrupts are taken to EL2, unless they are routed to EL3.

A virtual IRQ is pending because of this mechanism.
The virtual IRQ is enabled only when the value of HCR_EL2.{TGE, IMO} is {0, 1}.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
• When the value of HCR_EL2.TGE is 0, then Virtual IRQ interrupts are enabled.
If EL2 is enabled in the current Security state, and the value of HCR_EL2.TGE is 1:
• Regardless of the value of the IMO bit, physical IRQ Interrupts target EL2 unless they are routed to EL3.
• When FEAT_VHE is not implemented, or if HCR_EL2.E2H is 0, this field behaves as 1 for all purposes other than a direct read of the value of this bit.
• When FEAT_VHE is implemented and HCR_EL2.E2H is 1, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

For more information, see *Establishing the target Exception level of an asynchronous exception on page D1-4657.*

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

### FMO, bit [3]
Physical FIQ Routing.

- **0b0** When executing at Exception levels below EL2, and EL2 is enabled in the current Security state:
  • When the value of HCR_EL2.TGE is 0, Physical FIQ interrupts are not taken to EL2.
  • When the value of HCR_EL2.TGE is 1, Physical FIQ interrupts are taken to EL2 unless they are routed to EL3.
  • Virtual FIQ interrupts are disabled.
- **0b1** When executing at any Exception level, and EL2 is enabled in the current Security state:
  • Physical FIQ interrupts are taken to EL2, unless they are routed to EL3.
  • When HCR_EL2.TGE is 0, then Virtual FIQ interrupts are enabled.

If EL2 is enabled in the current Security state and the value of HCR_EL2.TGE is 1:
• Regardless of the value of the FMO bit, physical FIQ Interrupts target EL2 unless they are routed to EL3.
• When FEAT_VHE is not implemented, or if HCR_EL2.E2H is 0, this field behaves as 1 for all purposes other than a direct read of the value of this bit.
• When FEAT_VHE is implemented and HCR_EL2.E2H is 1, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

For more information, see *Establishing the target Exception level of an asynchronous exception on page D1-4657.*

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

### PTW, bit [2]
Protected Table Walk. In the EL1&0 translation regime, a translation table access made as part of a stage 1 translation table walk is subject to a stage 2 translation. The combining of the memory type attributes from the two stages of translation means the access might be made to a type of Device memory. If this occurs, then the value of this bit determines the behavior:

- **0b0** The translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be made speculatively.
- **0b1** The memory access generates a stage 2 Permission fault.

This bit is permitted to be cached in a TLB.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**SWIO, bit [1]**

Set/Way Invalidation Override. Causes EL1 execution of the data cache invalidate by set/way instructions to perform a data cache clean and invalidate by set/way:

- 0b0 This control has no effect on the operation of data cache invalidate by set/way instructions.
- 0b1 Data cache invalidate by set/way instructions perform a data cache clean and invalidate by set/way.

When the value of this bit is 1:
AArch32: DCISW performs the same invalidation as a DCCISW instruction.
AArch64: DCISW performs the same invalidation as a DC CISW instruction.
This bit can be implemented as RES1.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**VM, bit [0]**

Virtualization enable. Enables stage 2 address translation for the EL1&0 translation regime, when EL2 is enabled in the current Security state.

- 0b0 EL1&0 stage 2 address translation disabled.
- 0b1 EL1&0 stage 2 address translation enabled.

When the value of this bit is 1, data cache invalidate instructions executed at EL1 perform a data cache clean and invalidate. For the invalidate by set/way instruction this behavior applies regardless of the value of the HCR_EL2.SWIO bit.

This bit is permitted to be cached in a TLB.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing HCR_EL2**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, HCR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = NVMem[0x078];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    end
```
elsif PSTATE.EL == EL2 then
    X[t, 64] = HCR_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = HCR_EL2;

**MSR HCR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x078] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    endif
elsif PSTATE.EL == EL2 then
    HCR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    HCR_EL2 = X[t, 64];
D17.2.49  HCRX_EL2, Extended Hypervisor Configuration Register

The HCRX_EL2 characteristics are:

Purpose

Provides configuration controls for virtualization, including defining whether various operations are trapped to EL2.

Configurations

This register is present only when FEAT_HCX is implemented. Otherwise, direct accesses to HCRX_EL2 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

The bits in this register behave as if they are 0 for all purposes other than direct reads of the register if:

• EL2 is not enabled in the current Security state.
• SCR_EL3.HXEn is 0.

Attributes

HCRX_EL2 is a 64-bit register.

Field descriptions

Bits [63:12]

Reserved, RES0.

MSCEn, bit [11]

When FEAT_MOPS is implemented:

Memory Set and Memory Copy instructions Enable. Enables execution of the CPY*, SETG*, SETP*, SETM*, and SETE* instructions at EL1 or EL0.

0b0 Execution of the Memory Copy and Memory Set instructions is UNDEFINED at EL1 or EL0.

0b1 This control does not cause any instructions to be UNDEFINED.

This bit behaves as if it is 1 if any of the following are true:

• EL2 is not implemented or enabled.
• The value of HCR_EL2.{E2H, TGE} is {1, 1}.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
MCE2, bit [10]

*When FEAT_MOPS is implemented:*

Controls Memory Copy and Memory Set exceptions generated as part of attempting to execute the Memory Copy and Memory Set instructions from EL1.

0b0  Memory Copy and Memory Set exceptions generated from EL1 are taken to EL1.

0b1  Memory Copy and Memory Set exceptions generated from EL1 are taken to EL2.

When the value of `HCR_EL2.{E2H, TGE}` is {1, 1}, this control does not affect any exceptions due to the higher priority `SCTLR_EL2.MSCEn` control.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

*Otherwise:*

Reserved, RES0.

CMOW, bit [9]

*When FEAT_CMOW is implemented:*

Controls cache maintenance instruction permission for the following instructions executed at EL1 or EL0.

- IC IVAU, DC CIVAC, DC CIGDVAC and DC CIGVAC.
- ICIMVAC, DCCIMVAC.

0b0  These instructions executed at EL1 or EL0 with stage 2 read permission, but without stage 2 write permission do not generate a stage 2 permission fault.

0b1  These instructions executed at EL1 or EL0, if enabled as a result of `SCTLR_EL1.UCI==1`, with stage 2 read permission, but without stage 2 write permission generate a stage 2 permission fault.

For this control, stage 2 has write permission if S2AP[1] is 1 or DBM is 1 in the stage 2 descriptor. The instructions do not cause an update to the dirty state.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

*Otherwise:*

Reserved, RES0.

VFNMI, bit [8]

*When FEAT_NMI is implemented:*

Virtual FIQ Interrupt with Superpriority. Enables signaling of virtual FIQ interrupts with Superpriority.

0b0  When `HCR_EL2.VF` is 1, a signaled pending virtual FIQ interrupt does not have Superpriority.

0b1  When `HCR_EL2.VF` is 1, a signaled pending virtual FIQ interrupt has Superpriority.

When `HCR_EL2.VF` is 0, this bit has no effect.

The reset behavior of this field is:

- On a Warm reset, when EL3 is not implemented and EL2 is implemented, this field resets to 0.

*Otherwise:*

Reserved, RES0.
VINMI, bit [7]

When FEAT_NMI is implemented:

Virtual IRQ Interrupt with Superpriority. Enables signaling of virtual IRQ interrupts with Superpriority.

0b0 When HCR_EL2.VI is 1, a signaled pending virtual IRQ interrupt does not have Superpriority.

0b1 When HCR_EL2.VI is 1, a signaled pending virtual IRQ interrupt has Superpriority.

When HCR_EL2.VI is 0, this bit has no effect.

The reset behavior of this field is:

• On a Warm reset, when EL3 is not implemented and EL2 is implemented, this field resets to 0.

Otherwise:

Reserved, RES0.

TALLINT, bit [6]

When FEAT_NMI is implemented:

Trap MSR writes of ALLINT at EL1 using AArch64 to EL2, when EL2 is implemented and enabled in the current Security state, reported using EC syndrome value 0x18.

0b0 MSR writes of ALLINT are not trapped by this mechanism.

0b1 MSR writes of ALLINT at EL1 using AArch64 are trapped to EL2.

The reset behavior of this field is:

• On a Warm reset, when EL3 is not implemented and EL2 is implemented, this field resets to 0.

Otherwise:

Reserved, RES0.

SMPME, bit [5]

When FEAT_SME is implemented:

Streaming Mode Priority Mapping Enable.

Controls mapping of the value of SMPRI_EL1.Priority for streaming execution priority at EL0 or EL1.

0b0 The effective priority value is taken from SMPRI_EL1.Priority.

0b1 The effective priority value is:

• When the current Exception level is EL2 or EL3, the value of SMPRI_EL1.Priority.

• When the current Exception level is EL0 or EL1, the value of the SMPRIMAP_EL2 field corresponding to the value of SMPRI_EL1.Priority.

When SMIDR_EL1.SMPS is '0', this field is RES0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
FGTnXS, bit [4]

When FEAT_XS is implemented:
Determine if the fine-grained traps in HFGITR_EL2 that apply to each of the TLBI maintenance instructions that are accessible at EL1 also apply to the corresponding TLBI maintenance instructions with the nXS qualifier.

0b0  The fine-grained trap in the HFGITR_EL2 that applies to a TLBI maintenance instruction at EL1 also applies to the corresponding TLBI instruction with the nXS qualifier at EL1.

0b1  The fine-grained trap in the HFGITR_EL2 that applies to a TLBI maintenance instruction at EL1 does not apply to the corresponding TLBI instruction with the nXS qualifier at EL1.

The reset behavior of this field is:
•  On a Warm reset, when EL3 is not implemented and EL2 is implemented, this field resets to 0.

Otherwise:
Reserved, RES0.

FnXS, bit [3]

When FEAT_XS is implemented:
Determines the behavior of TLBI instructions affected by the XS attribute.
This control bit also determines whether an AArch64 DSB instruction behaves as a DSB instruction with an nXS qualifier when executed at EL0 and EL1.

0b0  This control does not have any effect on the behavior of the TLBI maintenance instructions.

0b1  A TLBI maintenance instruction without the nXS qualifier executed at EL1 behaves in the same way as the corresponding TLBI maintenance instruction with the nXS qualifier.
An AArch64 DSB instruction executed at EL1 or EL0 behaves in the same way as the corresponding DSB instruction with the nXS qualifier executed at EL1 or EL0.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:
•  On a Warm reset, when EL3 is not implemented and EL2 is implemented, this field resets to 0.

Otherwise:
Reserved, RES0.

EnASR, bit [2]

When FEAT_LS64_V is implemented:
When HCR_EL2.{E2H, TGE} !={1, 1}, traps execution of an ST64BV instruction at EL0 or EL1 to EL2.

0b0  Execution of an ST64BV instruction at EL0 is trapped to EL2 if the execution is not trapped by SCTLR_EL1.EnASR.
Execution of an ST64BV instruction at EL1 is trapped to EL2.

0b1  This control does not cause any instructions to be trapped.

A trap of an ST64BV instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000000.

The reset behavior of this field is:
•  On a Warm reset, when EL3 is not implemented and EL2 is implemented, this field resets to 0.
EnALS, bit [1]

When FEAT_LS64 is implemented:

When HCR_EL2.[E2H, TGE] != {1, 1}, traps execution of an LD64B or ST64B instruction at EL0 or EL1 to EL2.

0b0   Execution of an LD64B or ST64B instruction at EL0 is trapped to EL2 if the execution is not trapped by SCTLR_EL1.EnALS.

0b1   This control does not cause any instructions to be trapped.

A trap of an LD64B or ST64B instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000002.

The reset behavior of this field is:
- On a Warm reset, when EL3 is not implemented and EL2 is implemented, this field resets to 0.

EnAS0, bit [0]

When FEAT_LS64_ACCDATA is implemented:

When HCR_EL2.[E2H, TGE] != {1, 1}, traps execution of an ST64BV0 instruction at EL0 or EL1 to EL2.

0b0   Execution of an ST64BV0 instruction at EL0 is trapped to EL2 if the execution is not trapped by SCTLR_EL1.EnAS0.

0b1   This control does not cause any instructions to be trapped.

A trap of an ST64BV0 instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000001.

The reset behavior of this field is:
- On a Warm reset, when EL3 is not implemented and EL2 is implemented, this field resets to 0.

Accessing HCRX_EL2

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS } <Xt>, \text{ HCRX_EL2}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = NVMem[0xA0];
    elsif EL2Enabled() & HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.HXEn == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.HXEn == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = HCRX_EL2;
  end
elsif PSTATE.EL == EL3 then
  X[t, 64] = HCRX_EL2;
end

MSR HCRX_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    NVMem[0xA0] = X[t, 64];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.HXEn == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.HXEn == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    HCRX_EL2 = X[t, 64];
  end
elsif PSTATE.EL == EL3 then
  HCRX_EL2 = X[t, 64];
end
D17.2.50 HDFGRTR_EL2, Hypervisor Debug Fine-Grained Read Trap Register

The HDFGRTR_EL2 characteristics are:

**Purpose**

Provides controls for traps of MRS and MRC reads of debug, trace, PMU, and Statistical Profiling System registers.

**Configurations**

This register is present only when FEAT_FGT is implemented. Otherwise, direct accesses to HDFGRTR_EL2 are UNDEFINED.

**Attributes**

HDFGRTR_EL2 is a 64-bit register.

**Field descriptions**

When FEAT_SPE is implemented:

Traps MRS reads of PMBIDR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of PMBIDR_EL1 are not trapped by this mechanism.
If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMBIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

nPMSNEVFR_EL1, bit [62]

When FEAT_SPEv1p2 is implemented:

Trap MRS reads of PMSNEVFR_EL1 at EL1 using AArch64 to EL2.

If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMSNEVFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

nBRBDATA, bit [61]

When FEAT_BRBE is implemented:

Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:
- BRBINF<n>_EL1.
- BRBINFINJ_EL1.
- BRBSRC<n>_EL1.
- BRBSRCINJ_EL1.
- BRBTGT<n>_EL1.
- BRBTGTINJ_EL1.
- BRBTS_EL1.

If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

nBRBCTL, bit [60]

When FEAT_BRBE is implemented:

Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:
- BRBCR_EL1.
• **BRBFCR_EL1.**

0b0  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

0b1  MRS reads of the System registers listed above are not trapped by this mechanism.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**nBRBIDR, bit [59]**

*When FEAT_BRBE is implemented:*

Trap MRS reads of BRBIDR0_EL1 at EL1 using AArch64 to EL2.

0b0  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of BRBIDR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

0b1  MRS reads of BRBIDR0_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**PMCEIDn_EL0, bit [58]**

*When FEAT_PMUv3 is implemented:*

Trap MRS reads of PMCEID<n>_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMCEID<n> at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0  MRS reads of PMCEID<n>_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMCEID<n> at EL0 using AArch32 are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

• MRS reads of PMCEID<n>_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.

• MRC reads of PMCEID<n> at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**PMUSERENR_EL0, bit [57]**

*When FEAT_PMUv3 is implemented:*

Trap MRS reads of PMUSERENR_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMUSEREN at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0  MRS reads of PMUSERENR_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMUSEREN at EL0 using AArch32 are not trapped by this mechanism.
If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of PMUSERENR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of PMUSERENR at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x83.

The reset behavior of this field is:

- **If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:**
  - **MRS reads of PMUSERENR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.**
  - **MRC reads of PMUSERENR at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x83.**

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**TRBTRG_EL1, bit [56]**

*When FEAT_TRBE is implemented:*

- Trap MRS reads of TRBTRG_EL1 at EL1 using AArch64 to EL2.
  - **0b0** MRS reads of TRBTRG_EL1 are not trapped by this mechanism.
  - **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRBTRG_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**TRBSR_EL1, bit [55]**

*When FEAT_TRBE is implemented:*

- Trap MRS reads of TRBSR_EL1 at EL1 using AArch64 to EL2.
  - **0b0** MRS reads of TRBSR_EL1 are not trapped by this mechanism.
  - **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRBSR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**TRBPTR_EL1, bit [54]**

*When FEAT_TRBE is implemented:*

- Trap MRS reads of TRBPTR_EL1 at EL1 using AArch64 to EL2.
  - **0b0** MRS reads of TRBPTR_EL1 are not trapped by this mechanism.
  - **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRBPTR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.
Otherwise:

Reserved, RES0.

TRBMAR_EL1, bit [53]

When FEAT_TRBE is implemented:

Trap MRS reads of TRBMAR_EL1 at EL1 using AArch64 to EL2.

0\text{b}0 \ MRS reads of TRBMAR_EL1 are not trapped by this mechanism.

0\text{b}1 \ If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRBMAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

TRBLIMITR_EL1, bit [52]

When FEAT_TRBE is implemented:

Trap MRS reads of TRBLIMITR_EL1 at EL1 using AArch64 to EL2.

0\text{b}0 \ MRS reads of TRBLIMITR_EL1 are not trapped by this mechanism.

0\text{b}1 \ If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRBLIMITR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

TRBIDR_EL1, bit [51]

When FEAT_TRBE is implemented:

Trap MRS reads of TRBIDR_EL1 at EL1 using AArch64 to EL2.

0\text{b}0 \ MRS reads of TRBIDR_EL1 are not trapped by this mechanism.

0\text{b}1 \ If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRBIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

TRBBASER_EL1, bit [50]

When FEAT_TRBE is implemented:

Trap MRS reads of TRBBASER_EL1 at EL1 using AArch64 to EL2.

0\text{b}0 \ MRS reads of TRBBASER_EL1 are not trapped by this mechanism.

0\text{b}1 \ If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRBBASER_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**Bit [49]**

Reserved, RES0.

**TRCVICTLR, bit [48]**

*When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to
the trace unit registers is implemented):*

In an Armv9 implementation, trap MRS reads of TRCVICTLR at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MRS reads of ETM TRCVICTLR at EL1 using AArch64 to EL2.

0b0: MRS reads of TRCVICTLR are not trapped by this mechanism.

0b1: If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRCVICTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**TRCSTATR, bit [47]**

*When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to
the trace unit registers is implemented):*

In an Armv9 implementation, trap MRS reads of TRCSTATR at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MRS reads of ETM TRCSTATR at EL1 using AArch64 to EL2.

0b0: MRS reads of TRCSTATR are not trapped by this mechanism.

0b1: If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRCSTATR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**TRCSSCSRn, bit [46]**

*When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented, TRCSSCSR<n> are
implemented and System register access to the trace unit registers is implemented):*

In an Armv9 implementation, trap MRS reads of TRCSSCSR<n> at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MRS reads of ETM TRCSSCSR<n> at EL1 using AArch64 to EL2.

0b0: MRS reads of TRCSSCSR<n> are not trapped by this mechanism.

0b1: If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRCSSCSR<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

If Single-shot Comparator n is not implemented, a read of TRCSSCSR<n> is UNDEFINED.
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

TRCSEQSTR, bit [45]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented, TRCSEQSTR is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MRS reads of TRCSEQSTR at EL1 using AArch64 to EL2.
In an Armv8 implementation, trap MRS reads of ETM TRCSEQSTR at EL1 using AArch64 to EL2.

0b0 MRS reads of TRCSEQSTR are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRCSEQSTR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

TRCPRGCTLR, bit [44]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MRS reads of TRCPRGCTLR at EL1 using AArch64 to EL2.
In an Armv8 implementation, trap MRS reads of ETM TRCPRGCTLR at EL1 using AArch64 to EL2.

0b0 MRS reads of TRCPRGCTLR are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRCPRGCTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

TRCOSLSR, bit [43]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MRS reads of TRCOSLSR at EL1 using AArch64 to EL2.
In an Armv8 implementation, trap MRS reads of ETM TRCOSLSR at EL1 using AArch64 to EL2.

0b0 MRS reads of TRCOSLSR are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRCOSLSR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.
Bit [42]
Reserved, RES0.

TRCIMSPEC<n>, bit [41]

*When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):*

In an Armv9 implementation, trap \texttt{MRS} reads of TRCIMSPEC<n> at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap \texttt{MRS} reads of ETM TRCIMSPEC<n> at EL1 using AArch64 to EL2.

\[\begin{align*}
\texttt{0b0} & \quad \text{\texttt{MRS} reads of TRCIMSPEC<n> are not trapped by this mechanism.} \\
\texttt{0b1} & \quad \text{If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then \texttt{MRS} reads of TRCIMSPEC<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.}
\end{align*}\]

TRCIMSPEC<1-7> are optional. If TRCIMSPEC<n> is not implemented, a read of TRCIMSPEC<n> is \texttt{UNDEFINED}.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.

TRCID, bit [40]

*When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):*

Trap \texttt{MRS} reads of multiple System registers. Enables a trap on \texttt{MRS} reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- In an Armv9 implementation:
  - TRCDEVARCH.
  - TRCDEVVID.
  - All of the TRCIDR<n> registers.
- In an Armv8 implementation:
  - ETM TRCDEVARCH.
  - ETM TRCDEVVID.
  - All of the ETM TRCIDR<n> registers.

\[\begin{align*}
\texttt{0b0} & \quad \text{\texttt{MRS} reads of the System registers listed above are not trapped by this mechanism.} \\
\texttt{0b1} & \quad \text{If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then \texttt{MRS} reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.}
\end{align*}\]

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.

Bits [39:38]
Reserved, RES0.
TRCCNTVR<\text{n}>, bit [37]

When \text{FEAT\_ETE} is implemented or (\text{FEAT\_ETMv4} is implemented, \text{TRCCNTVR}<n> are implemented and System register access to the trace unit registers is implemented):

- In an Armv9 implementation, trap \texttt{MRS} reads of TRCCNTVR<n> at EL1 using AArch64 to EL2.
- In an Armv8 implementation, trap \texttt{MRS} reads of ETM TRCCNTVR<n> at EL1 using AArch64 to EL2.
  
\begin{tabular}{ll}
\hline
0b0 & \texttt{MRS} reads of TRCCNTVR<n> are not trapped by this mechanism. \\
0b1 & If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR\_EL3.FGTEn == 1, then \texttt{MRS} reads of TRCCNTVR<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
\end{tabular}

If Counter \text{n} is not implemented, a read of TRCCNTVR<n> is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

\textbf{Otherwise:}

Reserved, RES0.

TRCCLAIM, bit [36]

When \text{FEAT\_ETE} is implemented or (\text{FEAT\_ETMv4} is implemented and System register access to the trace unit registers is implemented):

- In an Armv9 implementation, trap \texttt{MRS} reads of TRCCLAIMCLR and TRCCLAIMSET at EL1 using AArch64 to EL2.
- In an Armv8 implementation, trap \texttt{MRS} reads of ETM TRCCLAIMCLR and ETM TRCCLAIMSET at EL1 using AArch64 to EL2.

\begin{tabular}{ll}
\hline
0b0 & \texttt{MRS} reads of the System registers listed above are not trapped by this mechanism. \\
0b1 & If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR\_EL3.FGTEn == 1, then \texttt{MRS} reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
\end{tabular}

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

\textbf{Otherwise:}

Reserved, RES0.

TRCAUXCTLR, bit [35]

When \text{FEAT\_ETE} is implemented or (\text{FEAT\_ETMv4} is implemented and System register access to the trace unit registers is implemented):

- In an Armv9 implementation, trap \texttt{MRS} reads of TRCAUXCTLR at EL1 using AArch64 to EL2.
- In an Armv8 implementation, trap \texttt{MRS} reads of ETM TRCAUXCTLR at EL1 using AArch64 to EL2.

\begin{tabular}{ll}
\hline
0b0 & \texttt{MRS} reads of TRCAUXCTLR are not trapped by this mechanism. \\
0b1 & If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR\_EL3.FGTEn == 1, then \texttt{MRS} reads of TRCAUXCTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
\end{tabular}

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

\textbf{Otherwise:}

Reserved, RES0.
TRCAUTHSTATUS, bit [34]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MRS reads of TRCAUTHSTATUS at EL1 using AArch64 to EL2.

0b0 MRS reads of TRCAUTHSTATUS are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TRCAUTHSTATUS at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

TRC, bit [33]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MRS reads of the following registers at EL1 using AArch64 to EL2:

- TRCACATR<n>.
- TRCACVR<n>.
- TRCBBCCTRLR.
- TRCCCCCTRLR.
- TRCCIDCCTRLR0.
- TRCCIDCCTRLR1.
- TRCCIDCVR<n>.
- TRCCNCTCTRL<n>.
- TRCCNTRLDVR<n>.
- TRCCONFIGR.
- TRCEVENTCTRL0R.
- TRCEVENTCTRL1R.
- TRCEXINSEL<n>.
- TRCQCTLR.
- TRCRSCTLR<n>.
- TRCRSR.
- TRCSEQEVR<n>.
- TRCSEQRSTEVXR.
- TRCSSCCCR<n>.
- TRCSSPCICR<n>.
- TRCSTALLCTRLR.
- TRCSYNCPR.
- TRCTRACEEDR.
- TRCTSCCTRLR.
- TRCVIIECTLR.
- TRCVIPCSSCTLR.
- TRCVISSCTLR.
In an Armv8 implementation, trap \texttt{MRS} reads of the following registers at EL1 using AArch64 to EL2:

- ETM TRCACATR<n>.
- ETM TRCACVR<n>.
- ETM TRCBBCTLR.
- ETM TRCCCTLR.
- ETM TRCCIDCCTL0.
- ETM TRCCIDCCTL1.
- ETM TRCCIDCVR<n>.
- ETM TRCCNTCTLR<n>.
- ETM TRCCNTRLDVR<n>.
- ETM TRCCONFIGR.
- ETM TRCEVENTCTL0R.
- ETM TRCEVENTCTL1R.
- ETM TRCEXINSELR.
- ETM TRCQCCTLR.
- ETM TRCRSCTLR<n>.
- ETM TRCSEQEVR<n>.
- ETM TRCSEQRSTEV.<
- ETM TRCSSC<n>.
- ETM TRCSSPCICR<n>.
- ETM TRCSTALLCTLR.
- ETM TRCSYNCR.
- ETM TRCTRACEIDR.
- ETM TRCTSTCTRL.
- ETM TRCVIIECTLR.
- ETM TRCVIPCSSCTLR.
- ETM TRCVISSCTLR.
- ETM TRCMIDCCTL0.
- ETM TRCMIDCCTL1.
- ETM TRCMIDCVR<n>.

\texttt{MRS} reads of the System registers listed above are not trapped by this mechanism.

If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then \texttt{MRS} reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

A read of an unimplemented register is UNDEFINED.

\texttt{TRCEXINSELR<n> and TRCRSR} are only implemented if FEAT_ETE is implemented.

\texttt{TRCEXINSELR} is only implemented if FEAT_ETE is not implemented and FEAT_ETMv4 is implemented.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.
Otherwise:
Reserved, RES0.

PMSLATFR_EL1, bit [32]

When FEAT_SPE is implemented:
Trap MSR reads of PMSLATFR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR reads of PMSLATFR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR reads of PMSLATFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

PMSIRR_EL1, bit [31]

When FEAT_SPE is implemented:
Trap MSR reads of PMSIRR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR reads of PMSIRR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR reads of PMSIRR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

PMSIDR_EL1, bit [30]

When FEAT_SPE is implemented:
Trap MSR reads of PMSIDR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR reads of PMSIDR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR reads of PMSIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

PMSICR_EL1, bit [29]

When FEAT_SPE is implemented:
Trap MSR reads of PMSICR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR reads of PMSICR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR reads of PMSICR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

PMSFCR_EL1, bit [28]

When FEAT_SPE is implemented:

- Trap MR5 reads of PMSFCR_EL1 at EL1 using AArch64 to EL2.
  - 0b0: MR5 reads of PMSFCR_EL1 are not trapped by this mechanism.
  - 0b1: If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MR5 reads of PMSFCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

PMSEVFR_EL1, bit [27]

When FEAT_SPE is implemented:

- Trap MR5 reads of PMSEVFR_EL1 at EL1 using AArch64 to EL2.
  - 0b0: MR5 reads of PMSEVFR_EL1 are not trapped by this mechanism.
  - 0b1: If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MR5 reads of PMSEVFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

PMSCR_EL1, bit [26]

When FEAT_SPE is implemented:

- Trap MR5 reads of PMSCR_EL1 at EL1 using AArch64 to EL2.
  - 0b0: MR5 reads of PMSCR_EL1 are not trapped by this mechanism.
  - 0b1: If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MR5 reads of PMSCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

PMBSR_EL1, bit [25]

When FEAT_SPE is implemented:

- Trap MR5 reads of PMBSR_EL1 at EL1 using AArch64 to EL2.
  - 0b0: MR5 reads of PMBSR_EL1 are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMBSR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

PMBPTR_EL1, bit [24]

When FEAT_SPE is implemented:
- Trap MRS reads of PMBPTR_EL1 at EL1 using AArch64 to EL2.
  0b0 MRS reads of PMBPTR_EL1 are not trapped by this mechanism.
  0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMBPTR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

PMBLIMITR_EL1, bit [23]

When FEAT_SPE is implemented:
- Trap MRS reads of PMBLIMITR_EL1 at EL1 using AArch64 to EL2.
  0b0 MRS reads of PMBLIMITR_EL1 are not trapped by this mechanism.
  0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMBLIMITR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

PMMIR_EL1, bit [22]

When FEAT_PMUv3 is implemented:
- Trap MRS reads of PMMIR_EL1 at EL1 using AArch64 to EL2.
  0b0 MRS reads of PMMIR_EL1 are not trapped by this mechanism.
  0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PMMIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

Bits [21:20]

Reserved, RES0.
**PMSEL_EL0, bit [19]**

When FEAT_PMUv3 is implemented:

Trap MRS reads of PMSEL_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMSEL at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0  MRS reads of PMSEL_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMSEL at EL0 using AArch32 are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, HCR_EL2.[E2H, TGE] != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of PMSEL_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of PMSEL at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**PMOVNS, bit [18]**

When FEAT_PMUv3 is implemented:

Trap MRS reads and MRC reads of multiple System registers.

Enables a trap to EL2 the following operations:

- At EL1 and EL0 using AArch64: MRS reads of PMOVSCLR_EL0 and PMOVSET_EL0.
- At EL0 using AArch32 when EL1 is using AArch64: MRC reads of PMOVSR and PMOVSET.

0b0  The operations listed above are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, HCR_EL2.[E2H, TGE] != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads at EL1 and EL0 using AArch64 of PMOVSCLR_EL0 and PMOVSET_EL0 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads at EL0 using AArch32 of PMOVSR and PMOVSET are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**PINTEN, bit [17]**

When FEAT_PMUv3 is implemented:

Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- PMINTENCLR_EL1.
- PMINTENSET_EL1.

0b0  MRS reads of the System registers listed above are not trapped by this mechanism.
If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

PMCNTEN, bit [16]

When FEAT_PMUv3 is implemented:

Traps MRS reads and MRC reads of multiple System registers.

Enables a trap to EL2 the following operations:
• At EL1 and EL0 using AArch64: MRS reads of PMCNTECLKR_EL0 and PMCNTESET_EL0.
• At EL0 using AArch32 when EL1 is using AArch64: MRC reads of PMCNTECLKR and PMCNTESET.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

PMCCNTR_EL0, bit [15]

When FEAT_PMUv3 is implemented:

Traps MRS reads of PMCCNTR_EL0 at EL1 and EL0 using AArch64 and MRC and MRRC reads of PMCCNTR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0 MRS reads of PMCCNTR_EL0 at EL1 and EL0 using AArch64 and MRC and MRRC reads of PMCCNTR at EL0 using AArch32 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:
• MRS reads at EL1 and EL0 using AArch64 of PMCNTECLKR_EL0 and PMCNTESET_EL0 are trapped to EL2 and reported with EC syndrome value 0x18.
• MRC reads at EL0 using AArch32 of PMCNTECLKR and PMCNTESET are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.
PMCCFILTR_EL0, bit [14]

When FEAT_PMUv3 is implemented:

Traps MRS reads of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMCCFILTR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0  MRS reads of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 and MRC reads of PMCCFILTR at EL0 using AArch32 are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current security state, HCR_EL2.{E2H, TGE} != \{1, 1\}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads of PMCCFILTR at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

PMCCFILTR_EL0 can also be accessed in AArch64 state using PMXEVTYPER_EL0 when PMSELR_EL0.SEL == 31, and PMCCFILTR_EL0 can also be accessed in AArch32 state using PMXEVTYPER when PMSELR.SEL == 31.

Setting this field to 1 has no effect on accesses to PMXEVTYPER_EL0 and PMXEVTYPER, regardless of the value of PMSELR_EL0.SEL or PMSELR.SEL.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMEVTYPERn_EL0, bit [13]

When FEAT_PMUv3 is implemented:

Traps MRS reads and MRC reads of multiple System registers.

Enables a trap to EL2 the following operations:

- At EL1 and EL0 using AArch64: MRS reads of PMEVTYPEP<P>EVTYPER_EL0 and PMXEVTYPER_EL0.
- At EL0 using AArch32 when EL1 is using AArch64: MRC reads of PMEVTYPEP<P>EVTYPER and PMXEVTYPER.

0b0  The operations listed above are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current security state, HCR_EL2.{E2H, TGE} != \{1, 1\}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

- MRS reads at EL1 and EL0 using AArch64 of PMEVTYPEP<P>EVTYPER_EL0 and PMXEVTYPER_EL0 are trapped to EL2 and reported with EC syndrome value 0x18.
- MRC reads at EL0 using AArch32 of PMEVTYPEP<P>EVTYPER and PMXEVTYPER are trapped to EL2 and reported with EC syndrome value 0x03.

Regardless of the value of this field, for each value n:

- If event counter n is not implemented, the following accesses are UNDEFINED:
  - In AArch64 state, a read of PMEVTYPEP<P>EVTYPER_EL0, or, if n is not 31, a read of PMXEVTYPER_EL0 when PMSELR_EL0.SEL == n.
  - In AArch32 state, a read of PMEVTYPEP<P>EVTYPER, or, if n is not 31, a read of PMXEVTYPER when PMSELR.SEL == n.
• If event counter n is implemented, n is greater-than-or-equal-to MDCR_EL2.HPMN, and EL2 is implemented and enabled in the current Security state, the following generate a Trap exception to EL2 from EL0 or EL1:
  — In AArch64 state, a read of PMEVTYPER<\(n\)>_EL0, or a read of PMXEVTYPER_EL0 when PMSELR_EL0.SEL == \(n\), reported with EC syndrome value 0x18.
  — In AArch32 state, a read of PMEVTYPER<\(n\)> , or a read of PMXEVTYPER when PMSELR.SEL == \(n\), reported with EC syndrome value 0x03.

See also HDFGRTR_EL2.PMCCFILTR_EL0.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**PMEVCNTRn_EL0, bit [12]**

_When FEAT_PMUv3 is implemented:_

Trap MRS reads and MRC reads of multiple System registers.

Enables a trap to EL2 the following operations:

• At EL1 and EL0 using AArch64: MRS reads of PMEVCNTR<\(n\)>_EL0 and PMXEVCNTR_EL0.

• At EL0 using AArch32 when EL1 is using AArch64: MRC reads of PMEVCNTR<\(n\)> and PMXEVCNTR.

0b0
The operations listed above are not trapped by this mechanism.

0b1
If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != \{1, 1\}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:

• MRS reads at EL1 and EL0 using AArch64 of PMEVCNTR<\(n\)>_EL0 and PMXEVCNTR_EL0 are trapped to EL2 and reported with EC syndrome value 0x18.

• MRC reads at EL0 using AArch32 of PMEVCNTR<\(n\)> and PMXEVCNTR are trapped to EL2 and reported with EC syndrome value 0x03.

Regardless of the value of this field, for each value \(n\):

• If event counter n is not implemented, the following accesses are UNDEFINED:
  — In AArch64 state, a read of PMEVCNTR<\(n\)>_EL0, or a read of PMXEVCNTR_EL0 when PMSELR_EL0.SEL == \(n\).
  — In AArch32 state, a read of PMEVCNTR<\(n\)> , or a read of PMXEVCNTR when PMSELR.SEL == \(n\).

• If event counter n is implemented, n is greater-than-or-equal-to MDCR_EL2.HPMN, and EL2 is implemented and enabled in the current Security state, the following generate a Trap exception to EL2 from EL0 or EL1:
  — In AArch64 state, a read of PMEVCNTR<\(n\)>_EL0, or a read of PMXEVCNTR_EL0 when PMSELR_EL0.SEL == \(n\), reported with EC syndrome value 0x18.
  — In AArch32 state, a read of PMEVCNTR<\(n\)> , or a read of PMXEVCNTR when PMSELR.SEL == \(n\), reported with EC syndrome value 0x03.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.
OSDLR_EL1, bit [11]

When FEAT_DoubleLock is implemented:

Trap MRS reads of OSDLR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of OSDLR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of OSDLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

OSECCR_EL1, bit [10]

Trap MRS reads of OSECCR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of OSECCR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of OSECCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

OSLSR_EL1, bit [9]

Trap MRS reads of OSLSR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of OSLSR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of OSLSR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Bit [8]

Reserved, RES0.

DBGPRCR_EL1, bit [7]

Trap MRS reads of DBGPRCR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of DBGPRCR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of DBGPRCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

DBGAUTHSTATUS_EL1, bit [6]

Trap MRS reads of DBGAUTHSTATUS_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of DBGAUTHSTATUS_EL1 are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of DBGAUTHSTATUS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**DBGCLAIM, bit [5]**

Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- DBGCLAIMCLR_EL1.
- DBGCLAIMSET_EL1.

0b0 MRS reads of the System registers listed above are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**MDSCR_EL1, bit [4]**

Trap MRS reads of MDSCR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of MDSCR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of MDSCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**DBGWVRn_EL1, bit [3]**

Trap MRS reads of DBGWVR<n>_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of DBGWVR<n>_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of DBGWVR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

If watchpoint n is not implemented, a read of DBGWVR<n>_EL1 is UNDEFINED.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**DBGWCRn_EL1, bit [2]**

Trap MRS reads of DBGWCR<n>_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of DBGWCR<n>_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of DBGWCR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

If watchpoint n is not implemented, a read of DBGWCR<n>_EL1 is UNDEFINED.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.
DBGBVR<sub>n</sub>_EL1, bit [1]  
Trap MRS reads of DBGBVR<sub>n</sub>_EL1 at EL1 using AArch64 to EL2.

- 0b0 MRS reads of DBGBVR<sub>n</sub>_EL1 are not trapped by this mechanism.
- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of DBGBVR<sub>n</sub>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

If breakpoint n is not implemented, a read of DBGBVR<sub>n</sub>_EL1 is undefined.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

DBGBCR<sub>n</sub>_EL1, bit [0]  
Trap MRS reads of DBGBCR<sub>n</sub>_EL1 at EL1 using AArch64 to EL2.

- 0b0 MRS reads of DBGBCR<sub>n</sub>_EL1 are not trapped by this mechanism.
- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of DBGBCR<sub>n</sub>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

If breakpoint n is not implemented, a read of DBGBCR<sub>n</sub>_EL1 is undefined.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Accessing HDFGRTR_EL2  
Accesses to this register use the following encodings in the System register encoding space:

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b100 & 0b0011 & 0b0001 & 0b100 \\
\end{array}
\]

if PSTATE.EL == EL0 then  
UNDEFINED;
elsif PSTATE.EL == EL1 then  
if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then  
X[t, 64] = NVMem[0x1D0];  
else  
if EL2Enabled() && HCR_EL2.NV == '1' then  
AArch64.SystemAccessTrap(EL2, 0x18);  
else  
UNDEFINED;
elsif PSTATE.EL == EL2 then  
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.FGTEn == '0' then  
UNDEFINED;  
elseif HaveEL(EL3) && SCR_EL3.FGTEn == '0' then  
if Halted() && EDSCR.SDD == '1' then  
UNDEFINED;  
ever  
AArch64.SystemAccessTrap(EL3, 0x18);  
ever  
X[t, 64] = HDFGRTR_EL2;  
elseif PSTATE.EL == EL3 then  
X[t, 64] = HDFGRTR_EL2;
MSR HDFGRTR_EL2, <Xt>  

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0011</td>
<td>0b0001</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then  
  UNDEFINED;  
elsif PSTATE.EL == EL1 then  
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then  
    NVMem[0x1D0] = X[t, 64];  
  elsif EL2Enabled() && HCR_EL2.NV == '1' then  
    AArch64.SystemAccessTrap(EL2, 0x18);  
  else  
    UNDEFINED;  
elsif PSTATE.EL == EL2 then  
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.FGTEn == '0' then  
    UNDEFINED;  
  elsif HaveEL(EL3) && SCR_EL3.FGTEn == '0' then  
    if Halted() && EDSCR.SDD == '1' then  
      UNDEFINED;  
    else  
      AArch64.SystemAccessTrap(EL3, 0x18);  
  else  
    HDFGRTR_EL2 = X[t, 64];  
elsif PSTATE.EL == EL3 then  
  HDFGRTR_EL2 = X[t, 64];
D17.2.51 HDFGWTR_EL2, Hypervisor Debug Fine-Grained Write Trap Register

The HDFGWTR_EL2 characteristics are:

**Purpose**

Provides controls for traps of MSR and MCR writes of debug, trace, PMU, and Statistical Profiling System registers.

**Configurations**

This register is present only when FEAT_FGT is implemented. Otherwise, direct accesses to HDFGWTR_EL2 are UNDEFINED.

**Attributes**

HDFGWTR_EL2 is a 64-bit register.

**Field descriptions**

Bit [63]

Reserved, RES0.
**nPMSNEVFR_EL1, bit [62]**

*When FEAT_SPEv1p2 is implemented:*

Trap MSR writes of PMSNEVFR_EL1 at EL1 using AArch64 to EL2.

0b0  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMSNEVFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

0b1  MSR writes of PMSNEVFR_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**nBRBDATA, bit [61]**

*When FEAT_BRBE is implemented:*

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- BRBINFINJ_EL1.
- BRBSRCINJ_EL1.
- BRBTGTINJ_EL1.
- BRBTS_EL1.

0b0  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

0b1  MSR writes of the System registers listed above are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**nBRBCTL, bit [60]**

*When FEAT_BRBE is implemented:*

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- BRBCR_EL1.
- BRBFRCR_EL1.

0b0  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

0b1  MSR writes of the System registers listed above are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Bits [59:58]**

Reserved, RES0.
PMUSERENR_EL0, bit [57]

When FEAT_PMUv3 is implemented:

Trap MSR writes of PMUSERENR_EL0 at EL1 using AArch64 to EL2.

0b0 MSR writes of PMUSERENR_EL0 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMUSERENR_EL0 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

TRBTRG_EL1, bit [56]

When FEAT_TRBE is implemented:

Trap MSR writes of TRBTRG_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of TRBTRG_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRBTRG_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

TRBSR_EL1, bit [55]

When FEAT_TRBE is implemented:

Trap MSR writes of TRBSR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of TRBSR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRBSR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

TRBPTR_EL1, bit [54]

When FEAT_TRBE is implemented:

Trap MSR writes of TRBPTR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of TRBPTR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRBPTR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.
TRBMAR_EL1, bit [53]

When FEAT_TRBE is implemented:

Trap MSR writes of TRBMAR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of TRBMAR_EL1 are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRBMAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

TRBLIMITR_EL1, bit [52]

When FEAT_TRBE is implemented:

Trap MSR writes of TRBLIMITR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of TRBLIMITR_EL1 are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRBLIMITR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

Bit [51]

Reserved, RES0.

TRBBASER_EL1, bit [50]

When FEAT_TRBE is implemented:

Trap MSR writes of TRBBASER_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of TRBBASER_EL1 are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRBBASER_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

TRFCR_EL1, bit [49]

When FEAT_TRF is implemented:

Trap MSR writes of TRFCR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of TRFCR_EL1 are not trapped by this mechanism.
If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRFCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**TRCVICTLR, bit [48]**

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MSR writes of TRCVICTLR at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MSR writes of ETM TRCVICTLR at EL1 using AArch64 to EL2.

0b0 MSR writes of TRCVICTLR are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRCVICTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**Bit [47]**

Reserved, RES0.

**TRCSSCSR<n>, bit [46]**

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented, TRCSSCSR<n> are implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MSR writes of TRCSSCSR<n> at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MSR writes of ETM TRCSSCSR<n> at EL1 using AArch64 to EL2.

0b0 MSR writes of TRCSSCSR<n> are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRCSSCSR<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

If Single-shot Comparator n is not implemented, a write of TRCSSCSR<n> is UNDEFINED.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**TRCSEQSTR, bit [45]**

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented, TRCSEQSTR is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MSR writes of TRCSEQSTR at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MSR writes of ETM TRCSEQSTR at EL1 using AArch64 to EL2.

0b0 MSR writes of TRCSEQSTR are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRCSEQSTR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.
If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRCSEQSTR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

TRCPRGCTLR, bit [44]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MSR writes of TRCPRGCTLR at EL1 using AArch64 to EL2.
In an Armv8 implementation, trap MSR writes of ETM TRCPRGCTLR at EL1 using AArch64 to EL2.

0b0  MSR writes of TRCPRGCTLR are not trapped by this mechanism.
0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRCPRGCTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

Bit [43]
Reserved, RES0.

TRCOSLAR, bit [42]

When System register access to the trace unit registers is implemented and FEAT_ETMv4 is implemented:

In an Armv8 implementation, trap MSR writes of ETM TRCOSLAR at EL1 using AArch64 to EL2.

0b0  MSR writes of TRCOSLAR are not trapped by this mechanism.
0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRCOSLAR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

TRCIMSPECn, bit [41]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MSR writes of TRCIMSPEC<n> at EL1 using AArch64 to EL2.
In an Armv8 implementation, trap MSR writes of ETM TRCIMSPEC<n> at EL1 using AArch64 to EL2.

0b0  MSR writes of TRCIMSPEC<n> are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRCIMSPEC<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value \(0x18\), unless the write generates a higher priority exception.

TRCIMSPEC<1-7> are optional. If TRCIMSPEC<n> is not implemented, a write of TRCIMSPEC<n> is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**Bits [40:38]**

Reserved, RES0.

**TRCCNTVRn, bit [37]**

*When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented, TRCCNTVR<n> are implemented and System register access to the trace unit registers is implemented):*

In an Armv9 implementation, trap MSR writes of TRCCNTVR<n> at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MSR writes of ETM TRCCNTVR<n> at EL1 using AArch64 to EL2.

- 0b0 MSR writes of TRCCNTVR<n> are not trapped by this mechanism.

- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRCCNTVR<n> at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value \(0x18\), unless the write generates a higher priority exception.

If Counter n is not implemented, a write of TRCCNTVR<n> is UNDEFINED.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**TRCCLAIM, bit [36]**

*When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):*

In an Armv9 implementation, trap MSR writes of TRCCLAIMCLR and TRCCLAIMSET at EL1 using AArch64 to EL2.

In an Armv8 implementation, trap MSR writes of ETM TRCCLAIMCLR and ETM TRCCLAIMSET at EL1 using AArch64 to EL2.

- 0b0 MSR writes of the System registers listed above are not trapped by this mechanism.

- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value \(0x18\), unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.
TRCAUXCTLR, bit [35]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MSR writes of TRCAUXCTLR at EL1 using AArch64 to EL2.
In an Armv8 implementation, trap MSR writes of ETM TRCAUXCTLR at EL1 using AArch64 to EL2.

0b0  MSR writes of TRCAUXCTLR are not trapped by this mechanism.
0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TRCAUXCTLR at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value \(0x18\), unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

Bit [34]
Reserved, RES0.

TRC, bit [33]

When FEAT_ETE is implemented or (FEAT_ETMv4 is implemented and System register access to the trace unit registers is implemented):

In an Armv9 implementation, trap MSR writes of the following registers at EL1 using AArch64 to EL2:

- TRCACATR<\(n\)>.
- TRCACVR<\(n\)>.
- TRCBBCCTRL.
- TRCCCTCTRL.
- TRCCIDCCTLR0.
- TRCCIDCCTLR1.
- TRCCIDCVR<\(n\)>.
- TRCCNTCTLR<\(n\)>.
- TRCCNTRLDVR<\(n\)>.
- TRCCONFIGR.
- TRCEVENTCTLR0R.
- TRCEVENTCTLR1R.
- TRCEXINTSELR<\(n\)>.
- TRCQCTRL.
- TRCRSCTRL<\(n\)>.
- TRCRSR.
- TRCSEQEVR<\(n\)>.
- TRCSEQRSRTEVR.
- TRCSSCCR<\(n\)>.
- TRCSSPCICR<\(n\)>.
- TRCSALLCTLR.
- TRCSYNCPR.
- TRCTRACEIDR.
• TRCTSCTLR.
• TRCVIIECTLR.
• TRCVIPCSSCTLR.
• TRCVISSCTLR.
• TRCVMIDCCTL0.
• TRCVMIDCCTL1.
• TRCVMIDCVR<n>.

In an Armv8 implementation, trap MSR writes of the following registers at EL1 using AArch64 to EL2:

• ETM TRCACATR<n>.
• ETM TRCACVR<n>.
• ETM TRCBBCTRL.
• ETM TRCCCCTL.
• ETM TRCCIDDCTL0.
• ETM TRCCIDDCTL1.
• ETM TRCCIDCVR<n>.
• ETM TRCCNTCTL<n>.
• ETM TRCCNTLDVR<n>.
• ETM TRCCONFIGR.
• ETM TRCEVENTCTL0R.
• ETM TRCEVENTCTL1R.
• ETM TRCEXTINSELR.
• ETM TRCQCTRL.
• ETM TRCRSCTL<n>.
• ETM TCSEQEVR<n>.
• ETM TRCSEQSTEV.<n>.
• ETM TRCSSCVR<n>.
• ETM TRCSSPCICR<n>.
• ETM TRCSSPCICR<n>.
• ETM TRCTALLCTL.<n>.
• ETM TRCSYNCPR.
• ETM TRCTRACEIDR.
• ETM TRCTSCTL.<n>.
• ETM TRCVIIECTL.<n>.
• ETM TRCVIPCSSCTL.<n>.
• ETM TRCVISSCTL.<n>.
• ETM TRCVMIDCCTL0.<n>.
• ETM TRCVMIDCCTL1.<n>.
• ETM TRCVMIDCVR<n>.

0b0 MSR writes of the System registers listed above are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

A write of an unimplemented register is UNDEFINED.

TRCEXTINSELR<n> and TRCRSR are only implemented if FEAT_ETE is implemented.
TRCEXTINSELR is only implemented if FEAT_ETE is not implemented and FEAT_ETMv4 is implemented.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

PMSLATFR_EL1, bit [32]
When FEAT_SPE is implemented:
Traps MSR writes of PMSLATFR_EL1 at EL1 using AArch64 to EL2.
- 0b0 MSR writes of PMSLATFR_EL1 are not trapped by this mechanism.
- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMSLATFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

PMSIRR_EL1, bit [31]
When FEAT_SPE is implemented:
Traps MSR writes of PMSIRR_EL1 at EL1 using AArch64 to EL2.
- 0b0 MSR writes of PMSIRR_EL1 are not trapped by this mechanism.
- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMSIRR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

Bit [30]
Reserved, RES0.

PMSICR_EL1, bit [29]
When FEAT_SPE is implemented:
Traps MSR writes of PMSICR_EL1 at EL1 using AArch64 to EL2.
- 0b0 MSR writes of PMSICR_EL1 are not trapped by this mechanism.
- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMSICR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.
PMSFCR_EL1, bit [28]

When FEAT_SPE is implemented:

Trap MSR writes of PMSFCR_EL1 at EL1 using AArch64 to EL2.

- **0b0** MSR writes of PMSFCR_EL1 are not trapped by this mechanism.
- **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMSFCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMSEVFR_EL1, bit [27]

When FEAT_SPE is implemented:

Trap MSR writes of PMSEVFR_EL1 at EL1 using AArch64 to EL2.

- **0b0** MSR writes of PMSEVFR_EL1 are not trapped by this mechanism.
- **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMSEVFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMSCR_EL1, bit [26]

When FEAT_SPE is implemented:

Trap MSR writes of PMSCR_EL1 at EL1 using AArch64 to EL2.

- **0b0** MSR writes of PMSCR_EL1 are not trapped by this mechanism.
- **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMSCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMBSR_EL1, bit [25]

When FEAT_SPE is implemented:

Trap MSR writes of PMBSR_EL1 at EL1 using AArch64 to EL2.

- **0b0** MSR writes of PMBSR_EL1 are not trapped by this mechanism.
- **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PMBSR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.
**PMBPTR_EL1, bit [24]**

*When FEAT_SPE is implemented:*

Trap MSR writes of `PMBPTR_EL1` at EL1 using AArch64 to EL2.

- **0b0** MSR writes of `PMBPTR_EL1` are not trapped by this mechanism.
- **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGTEn == 1`, then MSR writes of `PMBPTR_EL1` at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Reserved, RES0.**

**PMBLIMITR_EL1, bit [23]**

*When FEAT_SPE is implemented:*

Trap MSR writes of `PMBLIMITR_EL1` at EL1 using AArch64 to EL2.

- **0b0** MSR writes of `PMBLIMITR_EL1` are not trapped by this mechanism.
- **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGTEn == 1`, then MSR writes of `PMBLIMITR_EL1` at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Reserved, RES0.**

**Bit [22]**

Reserved, RES0.

**PMCR_EL0, bit [21]**

*When FEAT_PMUv3 is implemented:*

Trap MSR writes of `PMCR_EL0` at EL1 and EL0 using AArch64 and MCR writes of `PMCR` at EL0 using AArch32 when EL1 is using AArch64 to EL2.

- **0b0** MSR writes of `PMCR_EL0` at EL1 and EL0 using AArch64 and MCR writes of `PMCR` at EL0 using AArch32 are not trapped by this mechanism.
- **0b1** If EL2 is implemented and enabled in the current Security state, `HCR_EL2.{E2H, TGE} != {1, 1}`, EL1 is using AArch64, and either EL3 is not implemented or `SCR_EL3.FGTEn == 1`, then, unless the write generates a higher priority exception:
  - MSR writes of `PMCR_EL0` at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value `0x18`.
  - MCR writes of `PMCR` at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value `0x03`.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Reserved, RES0.**
PMSWINC_EL0, bit [20]

When FEAT_PMUv3 is implemented:
Trap MSR writes of PMSWINC_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMSWINC at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0  MSR writes of PMSWINC_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMSWINC at EL0 using AArch32 are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:
   •  MSR writes of PMSWINC_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
   •  MCR writes of PMSWINC at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:
•  On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

PMSELR_EL0, bit [19]

When FEAT_PMUv3 is implemented:
Trap MSR writes of PMSELR_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMSELR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0  MSR writes of PMSELR_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMSELR at EL0 using AArch32 are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:
   •  MSR writes of PMSELR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
   •  MCR writes of PMSELR at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:
•  On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

PMOVS, bit [18]

When FEAT_PMUv3 is implemented:
Trap MSR writes and MCR writes of multiple System registers.

Enables a trap to EL2 the following operations:
•  At EL1 and EL0 using AArch64: MSR writes of PMOVSCLR_EL0 and PMOVSSSET_EL0.
•  At EL0 using AArch32 when EL1 is using AArch64: MCR writes of PMOVSR and PMOVSSSET.

0b0  The operations listed above are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:
   •  MSR writes at EL1 and EL0 using AArch64 of PMOVSCLR_EL0 and PMOVSSSET_EL0 are trapped to EL2 and reported with EC syndrome value 0x18.
• MCR writes at EL0 using AArch32 of PMOVSR and PMOVSET are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

PMINTEN, bit [17]

When FEAT_PMUv3 is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

• PMINTENCL_EL1.
• PMINTENSET_EL1.

0b0 MSR writes of the System registers listed above are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

PMCNTEN, bit [16]

When FEAT_PMUv3 is implemented:

Trap MSR writes and MCR writes of multiple System registers.

Enables a trap to EL2 the following operations:

• At EL1 and EL0 using AArch64: MSR writes of PMCNTENCL_EL0 and PMCNTENSET_EL0.
• At EL0 using AArch32 when EL1 is using AArch64: MCR writes of PMCNTENCL and PMCNTENSET.

0b0 The operations listed above are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:

• MSR writes at EL1 and EL0 using AArch64 of PMCNTENCL_EL0 and PMCNTENSET_EL0 are trapped to EL2 and reported with EC syndrome value 0x18.
• MCR writes at EL0 using AArch32 of PMCNTENCL and PMCNTENSET are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.
PMCCNTR_EL0, bit [15]

When FEAT_PMUv3 is implemented:

Trap MSR writes of PMCCNTR_EL0 at EL1 and EL0 using AArch64 and MCR and MCRR writes of PMCCNTR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0  MSR writes of PMCCNTR_EL0 at EL1 and EL0 using AArch64 and MCR and MCRR writes of PMCCNTR at EL0 using AArch32 are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} = {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:

- MSR writes of PMCCNTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MCR and MCRR writes of PMCCNTR at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03 (for MCR) or 0x04 (for MCRR).

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMCCFILTR_EL0, bit [14]

When FEAT_PMUv3 is implemented:

Trap MSR writes of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMCCFILTR at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0  MSR writes of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 and MCR writes of PMCCFILTR at EL0 using AArch32 are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} = {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:

- MSR writes of PMCCFILTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MCR writes of PMCCFILTR at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

PMCCFILTR_EL0 can also be accessed in AArch64 state using PMXEVTYPER_EL0 when PMSELR_EL0.SEL == 31, and PMCCFILTR can also be accessed in AArch32 state using PMXEVTYPER when PMSEL R.SEL == 31.

Setting this field to 1 has no effect on accesses to PMXEVTYPER_EL0 and PMXEVTYPER, regardless of the value of PMSEL R_EL0.SEL or PMSEL R.SEL.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

PMEVTYPERn_EL0, bit [13]

When FEAT_PMUv3 is implemented:

Trap MSR writes and MCR writes of multiple System registers.

Enables a trap to EL2 the following operations:

- At EL1 and EL0 using AArch64: MSR writes of PMEVTYPER<n>_EL0 and PMXEVTYPER_EL0.
- At EL0 using AArch32 when EL1 is using AArch64: MCR writes of PMEVTYPER<n> and PMXEVTYPER.

0b0  The operations listed above are not trapped by this mechanism.
0b1  If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:
   • MSR writes at EL1 and EL0 using AArch64 of PMEVTYPER<ν>_EL0 and PMXEVTYPER_EL0 are trapped to EL2 and reported with EC syndrome value 0x18.
   • MCR writes at EL0 using AArch32 of PMEVTYPER<ν> and PMXEVTYPER are trapped to EL2 and reported with EC syndrome value 0x03.

Regardless of the value of this field, for each value ν:
   • If event counter ν is not implemented, the following accesses are UNDEFINED:
     — In AArch64 state, a write of PMEVTYPER<ν>_EL0, or if ν is not 31, a write of PMXEVTYPER_EL0 when PMSELR_EL0.SEL == ν.
     — In AArch32 state, a write of PMEVTYPER<ν>, or if ν is not 31, a write of PMXEVTYPER when PMSELR.SEL == ν.
   • If event counter ν is implemented, ν is greater-than-or-equal-to MDCR_EL2.HPMN, and EL2 is implemented and enabled in the current Security state, the following generate a Trap exception to EL2 from EL0 or EL1:
     — In AArch64 state, a write of PMEVTYPER<ν>_EL0, or a write of PMXEVTYPER_EL0 when PMSELR_EL0.SEL == ν, reported with EC syndrome value 0x18.
     — In AArch32 state, a write of PMEVTYPER<ν>, or a write of PMXEVTYPER when PMSELR.SEL == ν, reported with EC syndrome value 0x03.

See also HDFGWTR_EL2:PMCCFILTER_EL0.

The reset behavior of this field is:
   • On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

 Otherwise:
   Reserved, RES0.

PMEVCNTRn_EL0, bit [12]

When FEAT_PMUv3 is implemented:

Trap MSR writes and MCR writes of multiple System registers.

Enables a trap to EL2 the following operations:
   • At EL1 and EL0 using AArch64: MSR writes of PMEVCNTR<ν>_EL0 and PMXEVCNTR_EL0.
   • At EL0 using AArch32 when EL1 is using AArch64: MCR writes of PMEVCNTR<ν> and PMXEVCNTR.

0b0  The operations listed above are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:
   • MSR writes at EL1 and EL0 using AArch64 of PMEVCNTR<ν>_EL0 and PMXEVCNTR_EL0 are trapped to EL2 and reported with EC syndrome value 0x18.
   • MCR writes at EL0 using AArch32 of PMEVCNTR<ν> and PMXEVCNTR are trapped to EL2 and reported with EC syndrome value 0x03.

Regardless of the value of this field, for each value ν:
   • If event counter ν is not implemented, the following accesses are UNDEFINED:
     — In AArch64 state, a write of PMEVCNTR<ν>_EL0, or a write of PMXEVCNTR_EL0 when PMSELR_EL0.SEL == ν.
— In AArch32 state, a write of PMEVCNTR<\(n\)> , or a write of PMXEV CNTR when PMSELR.SEL \(\equiv n\).

• If event counter \(n\) is implemented, \(n\) is greater-than-or-equal-to MDCR_EL2.HPMN, and EL2 is implemented and enabled in the current Security state, the following generate a Trap exception to EL2 from EL0 or EL1:

— In AArch64 state, a write of PMEVCNTR<\(n\)> \_EL0, or a write of PMXEV CNTR\_EL0 when PMSELR\_EL0.SEL \(\equiv n\), reported with EC syndrome value \(0x18\).

— In AArch32 state, a write of PMEVCNTR<\(n\)> , or a write of PMXEV CNTR when PMSELR.SEL \(\equiv n\), reported with EC syndrome value \(0x03\).

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

\(\text{OSDLR\_EL1, bit [11]}\)

When \(\text{FEAT\_DoubleLock \text{is implemented}}\):

Trap MSR writes of OSDLR\_EL1 at EL1 using AArch64 to EL2.

\(0b0\) MSR writes of OSDLR\_EL1 are not trapped by this mechanism.

\(0b1\) If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR\_EL3.FGTEn \(\equiv 1\), then MSR writes of OSDLR\_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value \(0x18\), unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

\(\text{Otherwise:}\)

Reserved, RES0.

\(\text{OSECCR\_EL1, bit [10]}\)

Trap MSR writes of OSECCR\_EL1 at EL1 using AArch64 to EL2.

\(0b0\) MSR writes of OSECCR\_EL1 are not trapped by this mechanism.

\(0b1\) If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR\_EL3.FGTEn \(\equiv 1\), then MSR writes of OSECCR\_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value \(0x18\), unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

\(\text{Bit [9]}\)

Reserved, RES0.

\(\text{OSLAR\_EL1, bit [8]}\)

Trap MSR writes of OSLAR\_EL1 at EL1 using AArch64 to EL2.

\(0b0\) MSR writes of OSLAR\_EL1 are not trapped by this mechanism.

\(0b1\) If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR\_EL3.FGTEn \(\equiv 1\), then MSR writes of OSLAR\_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value \(0x18\), unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.
DBGPRCR_EL1, bit [7]

Trap MSR writes of DBGPRCR_EL1 at EL1 using AArch64 to EL2.

0b0  MSR writes of DBGPRCR_EL1 are not trapped by this mechanism.
0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of DBGPRCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Bit [6]

Reserved, RES0.

DBGCLAIM, bit [5]

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

• DBGCLAIMCLR_EL1.
• DBGCLAIMSET_EL1.

0b0  MSR writes of the System registers listed above are not trapped by this mechanism.
0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

MDSCR_EL1, bit [4]

Trap MSR writes of MDSCR_EL1 at EL1 using AArch64 to EL2.

0b0  MSR writes of MDSCR_EL1 are not trapped by this mechanism.
0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

DBGWVRn_EL1, bit [3]

Trap MSR writes of DBGWVR<n>_EL1 at EL1 using AArch64 to EL2.

0b0  MSR writes of DBGWVR<n>_EL1 are not trapped by this mechanism.
0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of DBGWVR<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

If watchpoint n is not implemented, a write of DBGWVR<n>_EL1 is UNDEFINED.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

DBGWCRn_EL1, bit [2]

Trap MSR writes of DBGWCR<n>_EL1 at EL1 using AArch64 to EL2.

0b0  MSR writes of DBGWCR<n>_EL1 are not trapped by this mechanism.
If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of DBGWCR<\text{n}>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

If watchpoint n is not implemented, a write of DBGWCR<\text{n}>_EL1 is UNDEFINED.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**DBGBVRn_EL1, bit [1]**

Trap MSR writes of DBGBVR<\text{n}>_EL1 at EL1 using AArch64 to EL2.

- 0b0: MSR writes of DBGBVR<\text{n}>_EL1 are not trapped by this mechanism.
- 0b1: If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of DBGBVR<\text{n}>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

If breakpoint n is not implemented, a write of DBGBVR<\text{n}>_EL1 is UNDEFINED.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**DBGBCRn_EL1, bit [0]**

Trap MSR writes of DBGBCR<\text{n}>_EL1 at EL1 using AArch64 to EL2.

- 0b0: MSR writes of DBGBCR<\text{n}>_EL1 are not trapped by this mechanism.
- 0b1: If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of DBGBCR<\text{n}>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

If breakpoint n is not implemented, a write of DBGBCR<\text{n}>_EL1 is UNDEFINED.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

### Accessing HDFGWTR_EL2

Accesses to this register use the following encodings in the System register encoding space:

**MRS <\text{Xt}>, HDFGWTR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0011</td>
<td>0b0001</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = NVMem[0x1D8];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.FGTEn == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.FGTEn == '0' then

D17-5912 Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. Non-Confidential
if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = HDFGWTR_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = HDFGWTR_EL2;

**MSR HDFGWTR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0011</td>
<td>0b0001</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x1D8] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && EL2Enabled() && EDSCR.SDD == '1' then boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.FGTEn == '0' then
        UNDEFINED;
    elsif Halted() && SCR_EL3.FGTEn == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            HDFGWTR_EL2 = X[t, 64];
    elsif PSTATE.EL == EL3 then
        HDFGWTR_EL2 = X[t, 64];
### D17.2.52 HFGITR_EL2, Hypervisor Fine-Grained Instruction Trap Register

The HFGITR_EL2 characteristics are:

**Purpose**

Provides instruction trap controls.

**Configurations**

This register is present only when FEAT_FGT is implemented. Otherwise, direct accesses to HFGITR_EL2 are UNDEFINED.

**Attributes**

HFGITR_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Positions</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-57</td>
<td>Reserved, RES0</td>
</tr>
</tbody>
</table>

**Bits [63:57]**

Reserved, RES0.
nBRBIALL, bit [56]

*When FEAT_BRBE is implemented:*

Trap execution of `BRB IALL` at EL1 using AArch64 to EL2.

0b0  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGTEn == 1`, then execution of `BRB IALL` at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

0b1  Execution of `BRB IALL` is not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

*Otherwise:*

Reserved, RES0.

nBRBINJ, bit [55]

*When FEAT_BRBE is implemented:*

Trap execution of `BRB INJ` at EL1 using AArch64 to EL2.

0b0  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGTEn == 1`, then execution of `BRB INJ` at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

0b1  Execution of `BRB INJ` is not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

*Otherwise:*

Reserved, RES0.

DCCVAC, bit [54]

Trap execution of multiple instructions. Enables a trap on execution at EL1 and EL0 using AArch64 of any of the following AArch64 instructions to EL2:

- `DC CVAC`.
- `DC CGVAC`, if `FEAT_MTE` is implemented.
- `DC CGDVAC`, if `FEAT_MTE` is implemented.

If the Point of Coherence is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of the affected instruction is trapped when the value of this control is 1.

0b0  Execution of the instructions listed above is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, `HCR_EL2.{E2H, TGE} != \{1, 1\}`, and either EL3 is not implemented or `SCR_EL3.FGTEn == 1`, then execution at EL1 and EL0 using AArch64 of any of the instructions listed above is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

SVC_EL1, bit [53]

Trap execution of `SVC` at EL1 using AArch64 to EL2.

0b0  Execution of `SVC` is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGTEn == 1`, then execution of `SVC` at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x15, unless the instruction generates a higher priority exception.
The reset behavior of this field is:
  • On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**SVC_EL0, bit [52]**

Trap execution of SVC at EL0 using AArch64 and execution of SVC at EL0 using AArch32 when EL1 is using AArch64 to EL2.

- **0b0** Execution of SVC at EL0 using AArch64 and execution of SVC at EL0 using AArch32 is not trapped by this mechanism.
- **0b1** If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != \{1, 1\}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the instruction generates a higher priority exception:
  - Execution of SVC at EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x15.
  - Execution of SVC at EL0 using AArch32 is trapped to EL2 and reported with EC syndrome value 0x11.

The reset behavior of this field is:
  • On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**ERET, bit [51]**

Trap execution of multiple instructions. Enables a trap on execution at EL1 using AArch64 of any of the following AArch64 instructions to EL2:

- ERET.
- ERETAA, if FEAT_PAuth is implemented.
- ERETAB, if FEAT_PAuth is implemented.

- **0b0** Execution of the instructions listed above is not trapped by this mechanism.
- **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution at EL1 using AArch64 of any of the instructions listed above is trapped to EL2 and reported with EC syndrome value 0x1A, unless the instruction generates a higher priority exception.

If EL2 is implemented and enabled in the current Security state, HCR_EL2.API == 0, and this field enables a fine-grained trap on the instruction, then execution at EL1 using AArch64 of ERETAA or ERETAB instructions is trapped to EL2 and reported with EC syndrome value 0x1A with its associated ISS field, as the fine-grained trap has higher priority than the trap enabled by HCR_EL2.API == 0.

The reset behavior of this field is:
  • On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**CPPRCTX, bit [50]**

*When FEAT_SPECRES is implemented:*

Trap execution of CPP RCTX at EL1 and EL0 using AArch64 and execution of CPPRCTX at EL0 using AArch32 when EL1 is using AArch64 to EL2.

- **0b0** Execution of CPP RCTX at EL1 and EL0 using AArch64 and execution of CPPRCTX at EL0 using AArch32 is not trapped by this mechanism.
- **0b1** If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != \{1, 1\}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the instruction generates a higher priority exception:
  - Execution of CPP RCTX at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18.
  - Execution of CPPRCTX at EL0 using AArch32 is trapped to EL2 and reported with EC syndrome value 0x03.
The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**DVPRCTX, bit [49]**

*When FEAT_SPECRES is implemented:*

Trap execution of `DVPRCTX` at EL1 and EL0 using AArch64 and execution of `DVPRCTX` at EL0 using AArch32 when EL1 is using AArch64 to EL2.

- **0b0** Execution of `DVPRCTX` at EL1 and EL0 using AArch64 and execution of `DVPRCTX` at EL0 using AArch32 is not trapped by this mechanism.

- **0b1** If EL2 is implemented and enabled in the current Security state, `HCR_EL2.{E2H, TGE} !={1, 1}`, EL1 is using AArch64, and either EL3 is not implemented or `SCR_EL3.FGTEn == 1`, then, unless the instruction generates a higher priority exception:
  - Execution of `DVPRCTX` at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18.
  - Execution of `DVPRCTX` at EL0 using AArch32 is trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**CFPRCTX, bit [48]**

*When FEAT_SPECRES is implemented:*

Trap execution of `CFPRCTX` at EL1 and EL0 using AArch64 and execution of `CFPRCTX` at EL0 using AArch32 when EL1 is using AArch64 to EL2.

- **0b0** Execution of `CFPRCTX` at EL1 and EL0 using AArch64 and execution of `CFPRCTX` at EL0 using AArch32 is not trapped by this mechanism.

- **0b1** If EL2 is implemented and enabled in the current Security state, `HCR_EL2.{E2H, TGE} !={1, 1}`, EL1 is using AArch64, and either EL3 is not implemented or `SCR_EL3.FGTEn == 1`, then, unless the instruction generates a higher priority exception:
  - Execution of `CFPRCTX` at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18.
  - Execution of `CFPRCTX` at EL0 using AArch32 is trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**TLBIV AALE1, bit [47]**

Trap execution of `TLBIV AALE1`, `TLBIV AALE1NXS` at EL1 using AArch64 to EL2.

If `FEAT_XS` is implemented and `HCRX_EL2.FGTnXS == 0`, this field also traps execution of `TLBIV AALE1NXS`.

- **0b0** Execution of `TLBIV AALE1`, `TLBIV AALE1NXS` is not trapped by this mechanism.
If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VALE1, TLBI VALE1NXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

TLBIVALE1, bit [46]  
Trap execution of TLBI VALE1, TLBI VALE1NXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTXS == 0, this field also traps execution of TLBI VALE1NXS.

0b0  Execution of TLBI VALE1, TLBI VALE1NXS is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VALE1, TLBI VALE1NXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

TLBIVAAE1, bit [45]  
Trap execution of TLBI VAAE1, TLBI VAAE1NXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTXS == 0, this field also traps execution of TLBI VAAE1NXS.

0b0  Execution of TLBI VAAE1, TLBI VAAE1NXS is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VAAE1, TLBI VAAE1NXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

TLBIASIDE1, bit [44]  
Trap execution of TLBI ASIDE1, TLBI ASIDE1NXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTXS == 0, this field also traps execution of TLBI ASIDE1NXS.

0b0  Execution of TLBI ASIDE1, TLBI ASIDE1NXS is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI ASIDE1, TLBI ASIDE1NXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

TLIBVAE1, bit [43]  
Trap execution of TLBI VAE1, TLBI VAE1NXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTXS == 0, this field also traps execution of TLBI VAE1NXS.

0b0  Execution of TLBI VAE1, TLBI VAE1NXS is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VAE1, TLBI VAE1NXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.
The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**TLBIVMALLE1, bit [42]**

Trap execution of TLBI VMALLE1, TLBI VMALLE1NXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VMALLE1NXS.

- 0b0: Execution of TLBI VMALLE1, TLBI VMALLE1NXS is not trapped by this mechanism.
- 0b1: If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VMALLE1, TLBI VMALLE1NXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**TLBIRVAALE1, bit [41]**

*When FEAT_TLBIRANGE is implemented:*

Trap execution of TLBI RVAALE1, TLBI RVAALE1NXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAALE1NXS.

- 0b0: Execution of TLBI RVAALE1, TLBI RVAALE1NXS is not trapped by this mechanism.
- 0b1: If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAALE1, TLBI RVAALE1NXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**TLBIRVALE1, bit [40]**

*When FEAT_TLBIRANGE is implemented:*

Trap execution of TLBI RVALE1, TLBI RVALE1NXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVALE1NXS.

- 0b0: Execution of TLBI RVALE1, TLBI RVALE1NXS is not trapped by this mechanism.
- 0b1: If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVALE1, TLBI RVALE1NXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**TLBIRVAAE1, bit [39]**

*When FEAT_TLBIRANGE is implemented:*

Trap execution of TLBI RVAAE1, TLBI RVAAE1NXS at EL1 using AArch64 to EL2.
If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAAE1NXS.

- **0b0**: Execution of TLBI RVAAE1, TLBI RVAAE1NXS is not trapped by this mechanism.
- **0b1**: If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAAE1, TLBI RVAAE1NXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**TLBIRVAE1, bit [38]**

*When FEAT_TLBIRANGE is implemented:*

 Trap execution of TLBI RVAE1, TLBI RVAE1NXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAE1NXS.

- **0b0**: Execution of TLBI RVAE1, TLBI RVAE1NXS is not trapped by this mechanism.
- **0b1**: If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAE1, TLBI RVAE1NXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**TLBIRVAALE1IS, bit [37]**

*When FEAT_TLBIRANGE is implemented:*

 Trap execution of TLBI RVAALE1IS, TLBI RVAALE1ISNXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAALE1ISNXS.

- **0b0**: Execution of TLBI RVAALE1IS, TLBI RVAALE1ISNXS is not trapped by this mechanism.
- **0b1**: If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAALE1IS, TLBI RVAALE1ISNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**TLBIRVALE1IS, bit [36]**

*When FEAT_TLBIRANGE is implemented:*

 Trap execution of TLBI RVALE1IS, TLBI RVALE1ISNXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVALE1ISNXS.

- **0b0**: Execution of TLBI RVALE1IS, TLBI RVALE1ISNXS is not trapped by this mechanism.
If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAE1IS, TLBI RVAE1ISNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

TLBIRVAE1IS, bit [35]

When FEAT_TLBIRANGE is implemented:
Trap execution of TLBI RVAE1IS, TLBI RVAE1ISNXS at EL1 using AArch64 to EL2.
If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAE1ISNXS.

0b0 Execution of TLBI RVAE1IS, TLBI RVAE1ISNXS is not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAE1IS, TLBI RVAE1ISNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

TLBIRVAE1IS, bit [34]

When FEAT_TLBIRANGE is implemented:
Trap execution of TLBI RVAE1IS, TLBI RVAE1ISNXS at EL1 using AArch64 to EL2.
If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAE1ISNXS.

0b0 Execution of TLBI RVAE1IS, TLBI RVAE1ISNXS is not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAE1IS, TLBI RVAE1ISNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

TLBIVAALE1IS, bit [33]

Trap execution of TLBI VAALE1IS, TLBI VAALE1ISNXS at EL1 using AArch64 to EL2.
If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VAALE1ISNXS.

0b0 Execution of TLBI VAALE1IS, TLBI VAALE1ISNXS is not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VAALE1IS, TLBI VAALE1ISNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.
The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**TLBIVALE11S, bit [32]**

Trap execution of TLBLI VALE11S, TLBLI VALE11SNXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBLI VALE11SNXS.

0b0 Execution of TLBLI VALE11S, TLBLI VALE11SNXS is not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBLI VALE11S, TLBLI VALE11SNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**TLBIVAAE11S, bit [31]**

Trap execution of TLBLI VAAE11S, TLBLI VAAE11SNXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBLI VAAE11SNXS.

0b0 Execution of TLBLI VAAE11S, TLBLI VAAE11SNXS is not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBLI VAAE11S, TLBLI VAAE11SNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**TLBIASIDE11S, bit [30]**

Trap execution of TLBLI ASIDE11S, TLBLI ASIDE11SNXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBLI ASIDE11SNXS.

0b0 Execution of TLBLI ASIDE11S, TLBLI ASIDE11SNXS is not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBLI ASIDE11S, TLBLI ASIDE11SNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**TLBIVAE11S, bit [29]**

Trap execution of TLBLI VAE11S, TLBLI VAE11SNXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBLI VAE11SNXS.

0b0 Execution of TLBLI VAE11S, TLBLI VAE11SNXS is not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBLI VAE11S, TLBLI VAE11SNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.
TLBIVMALLE11S, bit [28]

Trap execution of TLBI VMALLE11S, TLBI VMALLE11SNXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VMALLE11SNXS.

0b0  Execution of TLBI VMALLE11S, TLBI VMALLE11SNXS is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VMALLE11S, TLBI VMALLE11SNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

TLBIRVALEE10S, bit [27]

When FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented:

Trap execution of TLBI RVALEE10S, TLBI RVALEE10SNXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVALEE10SNXS.

0b0  Execution of TLBI RVALEE10S, TLBI RVALEE10SNXS is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVALEE10S, TLBI RVALEE10SNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

TLBIRVALEE10S, bit [26]

When FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented:

Trap execution of TLBI RVALEE10S, TLBI RVALEE10SNXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVALEE10SNXS.

0b0  Execution of TLBI RVALEE10S, TLBI RVALEE10SNXS is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVALEE10S, TLBI RVALEE10SNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

TLBIRVAEE10S, bit [25]

When FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented:

Trap execution of TLBI RVAEE10S, TLBI RVAEE10SNXS at EL1 using AArch64 to EL2.
If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAE1OSNXS.

0b0  Execution of TLBI RVAE1OS, TLBI RVAE1OSNXS is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAE1OS, TLBI RVAE1OSNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**TLBIRVAE1OS, bit [24]**

*When FEAT_TLBIRANGE is implemented and FEAT_TLB1OS is implemented:*

Trap execution of TLBI RVAE1OS, TLBI RVAE1OSNXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI RVAE1OSNXS.

0b0  Execution of TLBI RVAE1OS, TLBI RVAE1OSNXS is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI RVAE1OS, TLBI RVAE1OSNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**TLBIVAALE1OS, bit [23]**

*When FEAT_TLB1OS is implemented:*

Trap execution of TLBI VAALE1OS, TLBI VAALE1OSNXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VAALE1OSNXS.

0b0  Execution of TLBI VAALE1OS, TLBI VAALE1OSNXS is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VAALE1OS, TLBI VAALE1OSNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**TLBIVALE1OS, bit [22]**

*When FEAT_TLB1OS is implemented:*

Trap execution of TLBI VALE1OS, TLBI VALE1OSNXS at EL1 using AArch64 to EL2.

If FEAT_XS is implemented and HCRX_EL2.FGTnXS == 0, this field also traps execution of TLBI VALE1OSNXS.

0b0  Execution of TLBI VALE1OS, TLBI VALE1OSNXS is not trapped by this mechanism.
If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VAE1OS, TLBI VAE1OSNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

TLBIVAE1OS, bit [21]

When FEAT_TLBVIOS is implemented:
Trap execution of TLBI VAE1OS, TLBI VAE1OSNXS at EL1 using AArch64 to EL2.
If FEAT_XS is implemented and HCRX_EL2.FGTXS == 0, this field also traps execution of TLBI VAE1OSNXS.

0b0 Execution of TLBI VAE1OS, TLBI VAE1OSNXS is not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VAE1OS, TLBI VAE1OSNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

TLBIASIDE1OS, bit [20]

When FEAT_TLBIOs is implemented:
Trap execution of TLBI ASIDE1OS, TLBI ASIDE1OSNXS at EL1 using AArch64 to EL2.
If FEAT_XS is implemented and HCRX_EL2.FGTXS == 0, this field also traps execution of TLBI ASIDE1OSNXS.

0b0 Execution of TLBI ASIDE1OS, TLBI ASIDE1OSNXS is not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI ASIDE1OS, TLBI ASIDE1OSNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

TLBIVAE1OS, bit [19]

When FEAT_TLBVIOS is implemented:
Trap execution of TLBI VAE1OS, TLBI VAE1OSNXS at EL1 using AArch64 to EL2.
If FEAT_XS is implemented and HCRX_EL2.FGTXS == 0, this field also traps execution of TLBI VAE1OSNXS.

0b0 Execution of TLBI VAE1OS, TLBI VAE1OSNXS is not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of TLBI VAE1OS, TLBI VAE1OSNXS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.
The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**TLBIVMALLE1OS, bit [18]**

*When FEAT_TLB1OS is implemented:*

- Trap execution of `TLBI VMALE1OS, TLBI VMALE1OSNXS` at EL1 using AArch64 to EL2.

  If `FEAT_XS` is implemented and `HCRX_EL2.FGtNXS == 0`, this field also traps execution of `TLBI VMALE1OSNXS`.

  - **0b0** Execution of `TLBI VMALE1OS, TLBI VMALE1OSNXS` is not trapped by this mechanism.
  
  - **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGtEn == 1`, then execution of `TLBI VMALE1OS, TLBI VMALE1OSNXS` at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

  The reset behavior of this field is:

  - On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**ATS1E1WP, bit [17]**

*When FEAT_PAN2 is implemented:*

- Trap execution of `ATS1E1WP` at EL1 using AArch64 to EL2.

  - **0b0** Execution of `ATS1E1WP` is not trapped by this mechanism.
  
  - **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGtEn == 1`, then execution of `ATS1E1WP` at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

  The reset behavior of this field is:

  - On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**ATS1E1RP, bit [16]**

*When FEAT_PAN2 is implemented:*

- Trap execution of `ATS1E1RP` at EL1 using AArch64 to EL2.

  - **0b0** Execution of `ATS1E1RP` is not trapped by this mechanism.
  
  - **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGtEn == 1`, then execution of `ATS1E1RP` at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

  The reset behavior of this field is:

  - On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**ATS1E0W, bit [15]**

- Trap execution of `ATS1E0W` at EL1 using AArch64 to EL2.

  - **0b0** Execution of `ATS1E0W` is not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGTEn` == 1, then execution of `AT S1E0W` at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**ATS1E0R, bit [14]**

Trap execution of `AT S1E0R` at EL1 using AArch64 to EL2.

0b0 Execution of `AT S1E0R` is not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGTEn` == 1, then execution of `AT S1E0R` at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**ATS1E1W, bit [13]**

Trap execution of `AT S1E1W` at EL1 using AArch64 to EL2.

0b0 Execution of `AT S1E1W` is not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGTEn` == 1, then execution of `AT S1E1W` at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**ATS1E1R, bit [12]**

Trap execution of `AT S1E1R` at EL1 using AArch64 to EL2.

0b0 Execution of `AT S1E1R` is not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGTEn` == 1, then execution of `AT S1E1R` at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value `0x18`, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**DCZVA, bit [11]**

Trap execution of multiple instructions. Enables a trap on execution at EL1 and EL0 using AArch64 of any of the following AArch64 instructions to EL2:
- DC ZVA.
- DC GVA, if `FEAT_MTE` is implemented.
- DC GZVA, if `FEAT_MTE` is implemented.

Note
Unlike `HCR_EL2.TDZ`, this field has no effect on `DCZID_EL0.DZP`.

0b0 Execution of the instructions listed above is not trapped by this mechanism.
If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution at EL1 and EL0 using AArch64 of any of the instructions listed above is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**DCCIVAC, bit [10]**
Trap execution of multiple instructions. Enables a trap on execution at EL1 and EL0 using AArch64 of any of the following AArch64 instructions to EL2:
- DC CIVAC.
- DC CIGVAC, if FEAT_MTE is implemented.
- DC CIGDVAC, if FEAT_MTE is implemented.

If the Point of Coherence is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of the affected instruction is trapped when the value of this control is 1.

Execution of the instructions listed above is not trapped by this mechanism.

**DCCVADP, bit [9]**
*When FEAT_DPB2 is implemented:*
Trap execution of multiple instructions. Enables a trap on execution at EL1 and EL0 using AArch64 of any of the following AArch64 instructions to EL2:
- DC CVADP.
- DC CGVADP, if FEAT_MTE is implemented.
- DC CGDVADP, if FEAT_MTE is implemented.

If the Point of Deep Persistence is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of the affected instruction is trapped when the value of this control is 1.

Execution of the instructions listed above is not trapped by this mechanism.

**DCCVAP, bit [8]**
Trap execution of multiple instructions. Enables a trap on execution at EL1 and EL0 using AArch64 of any of the following AArch64 instructions to EL2:
- DC CVAP.
- DC CGVAP, if FEAT_MTE is implemented.
• DC CGDVAP, if FEAT_MTE is implemented.

If the Point of Persistence is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of the affected instruction is trapped when the value of this control is 1.

0b0  Execution of the instructions listed above is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution at EL1 and EL0 using AArch64 of any of the instructions listed above is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

DCCVAU, bit [7]

Trap execution of DC CV AU at EL1 and EL0 using AArch64 to EL2.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of the affected instruction is trapped when the value of this control is 1.

0b0  Execution of DC CV AU is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of DC CV AU at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

DCCISW, bit [6]

Trap execution of multiple instructions. Enables a trap on execution at EL1 using AArch64 of any of the following AArch64 instructions to EL2:
• DC CISW.
• DC CIGSW, if FEAT_MTE is implemented.
• DC CIGDSW, if FEAT_MTE is implemented.

0b0  Execution of the instructions listed above is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution at EL1 using AArch64 of any of the instructions listed above is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

DCCSW, bit [5]

Trap execution of multiple instructions. Enables a trap on execution at EL1 using AArch64 of any of the following AArch64 instructions to EL2:
• DC CSW.
• DC CGSW, if FEAT_MTE is implemented.
• DC CGDSW, if FEAT_MTE is implemented.

0b0  Execution of the instructions listed above is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution at EL1 using AArch64 of any of the instructions listed above is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.
The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

DCISW, bit [4]
Trap execution of multiple instructions. Enables a trap on execution at EL1 using AArch64 of any of the following AArch64 instructions to EL2:
- DC ISW.
- DC IGSW, if FEAT_MTE is implemented.
- DC IGDSW, if FEAT_MTE is implemented.

0b0 Execution of the instructions listed above is not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn \(\neq 1\), then execution at EL1 using AArch64 of any of the instructions listed above is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

DCIVAC, bit [3]
Trap execution of multiple instructions. Enables a trap on execution at EL1 using AArch64 of any of the following AArch64 instructions to EL2:
- DC IVAC.
- DC IGVAC, if FEAT_MTE is implemented.
- DC IGDVAC, if FEAT_MTE is implemented.

If the Point of Coherence is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of the affected instruction is trapped when the value of this control is 1.

0b0 Execution of the instructions listed above is not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn \(\neq 1\), then execution at EL1 using AArch64 of any of the instructions listed above is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

ICIVAU, bit [2]
Trap execution of IC IVAU at EL1 and EL0 using AArch64 to EL2.
If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of the affected instruction is trapped when the value of this control is 1.

0b0 Execution of IC IVAU is not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} \(\neq \{1, 1\}\), and either EL3 is not implemented or SCR_EL3.FGTEn \(\neq 1\), then execution of IC IVAU at EL1 and EL0 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

ICIALLU, bit [1]
Trap execution of IC IALLU at EL1 using AArch64 to EL2.
If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of the affected instruction is trapped when the value of this control is 1.

0b0 Execution of IC IALLU is not trapped by this mechanism.
0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of IC IALLU at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

ICIALLUIS, bit [0]

Trap execution of IC IALLUIS at EL1 using AArch64 to EL2.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of the affected instruction is trapped when the value of this control is 1.

0b0  Execution of IC IALLUIS is not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then execution of IC IALLUIS at EL1 using AArch64 is trapped to EL2 and reported with EC syndrome value 0x18, unless the instruction generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Accessing HFGITR_EL2

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, HFGITR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b11</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = NVMem[0x1C8];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.FGTEn == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.FGTEn == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = HFGITR_EL2;
    elseif PSTATE.EL == EL3 then
        X[t, 64] = HFGITR_EL2;
```
**MSR HFGITR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    NVMem[0x1C8] = X[t, 64];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" && SCR_EL3.FGTEn == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.FGTEn == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      HFGITR_EL2 = X[t, 64];
  else
    HFGITR_EL2 = X[t, 64];
elsif PSTATE_EL == EL3 then
  HFGITR_EL2 = X[t, 64];
D17.2.53 HFGTR_EL2, Hypervisor Fine-Grained Read Trap Register

The HFGTR_EL2 characteristics are:

**Purpose**

Provides controls for traps of \( \text{MRS} \) and \( \text{MRC} \) reads of System registers.

**Configurations**

This register is present only when FEAT_FGT is implemented. Otherwise, direct accesses to HFGTR_EL2 are **UNDEFINED**.

**Attributes**

HFGTR_EL2 is a 64-bit register.

**Field descriptions**

HFGTR_EL2 is a 64-bit register with the following field descriptions:

**Bits [63:56]**

Reserved, RES0.

**nTPIDR2_EL0, bit [55]**

*When FEAT_SME is implemented:*

Trap \( \text{MRS} \) reads of TPIDR_EL0 at EL1 and EL0 using AArch64 to EL2.

- **0b0**
  
  If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} \(!= \{1, 1\} \), and either EL3 is not implemented or SCR_EL3.FGTEn \(!= 1\) then \( \text{MRS} \) reads of TPIDR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1  
MRS reads of TPIDR2_EL0 are not trapped by this mechanism.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

nSMPRI_EL1, bit [54]

*When FEAT_SME is implemented:*

Trap MRS reads of SMPRI_EL1 at EL1 using AArch64 to EL2.

0b0  
If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of SMPRI_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

0b1  
MRS reads of SMPRI_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

Bits [53:51]

Reserved, RES0.

nACCDATA_EL1, bit [50]

*When FEAT_LS64_ACCDATA is implemented:*

Trap MRS reads of ACCDATA_EL1 at EL1 using AArch64 to EL2.

0b0  
If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ACCDATA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

0b1  
MRS reads of ACCDATA_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

ERXADDR_EL1, bit [49]

*When FEAT_RAS is implemented:*

Trap MRS reads of ERXADDR_EL1 at EL1 using AArch64 to EL2.

0b0  
MRS reads of ERXADDR_EL1 are not trapped by this mechanism.

0b1  
If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXADDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.
ERXPFGCDN_EL1, bit [48]

When FEAT_RASv1p1 is implemented:

Trap MRS reads of ERXPFGCDN_EL1 at EL1 using AArch64 to EL2.

0b0 \[\text{\textcopyright}\] MRS reads of ERXPFGCDN_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXPFGCDN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

\* On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

ERXPFGCCTL_EL1, bit [47]

When FEAT_RASv1p1 is implemented:

Trap MRS reads of ERXPFGCCTL_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of ERXPFGCCTL_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXPFGCCTL_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

\* On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

ERXPFGF_EL1, bit [46]

When FEAT_RAS is implemented:

Trap MRS reads of ERXPFGF_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of ERXPFGF_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXPFGF_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

\* On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

ERXMISCn_EL1, bit [45]

When FEAT_RAS is implemented:

Trap MRS reads of ERXMISC<n>_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of ERXMISC<n>_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXMISC<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

\* On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.
Otherwise:
   Reserved, RES0.

ERXSTATUS_EL1, bit [44]

When FEAT_RAS is implemented:
   Trap MRS reads of ERXSTATUS_EL1 at EL1 using AArch64 to EL2.
   0b0   MRS reads of ERXSTATUS_EL1 are not trapped by this mechanism.
   0b1   If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXSTATUS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
   • On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
   Reserved, RES0.

ERXCTLR_EL1, bit [43]

When FEAT_RAS is implemented:
   Trap MRS reads of ERXCTLR_EL1 at EL1 using AArch64 to EL2.
   0b0   MRS reads of ERXCTLR_EL1 are not trapped by this mechanism.
   0b1   If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXCTLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
   • On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
   Reserved, RES0.

ERXFR_EL1, bit [42]

When FEAT_RAS is implemented:
   Trap MRS reads of ERXFR_EL1 at EL1 using AArch64 to EL2.
   0b0   MRS reads of ERXFR_EL1 are not trapped by this mechanism.
   0b1   If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
   • On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
   Reserved, RES0.

ERRSELR_EL1, bit [41]

When FEAT_RAS is implemented:
   Trap MRS reads of ERRSELR_EL1 at EL1 using AArch64 to EL2.
   0b0   MRS reads of ERRSELR_EL1 are not trapped by this mechanism.
   0b1   If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERRSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**ERRIDR_EL1, bit [40]**

**When FEAT_RAS is implemented:**

- Trap MRS reads of ERRIDR_EL1 at EL1 using AArch64 to EL2.
- 0b0 MRS reads of ERRIDR_EL1 are not trapped by this mechanism.
- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERRIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**ICC_IGRPENn_EL1, bit [39]**

**When FEAT_GICv3 is implemented:**

- Trap MRS reads of ICC_IGRPEN<n>_EL1 at EL1 using AArch64 to EL2.
- 0b0 MRS reads of ICC_IGRPEN<n>_EL1 are not trapped by this mechanism.
- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ICC_IGRPEN<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**VBAR_EL1, bit [38]**

- Trap MRS reads of VBAR_EL1 at EL1 using AArch64 to EL2.
- 0b0 MRS reads of VBAR_EL1 are not trapped by this mechanism.
- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of VBAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**TTBR1_EL1, bit [37]**

- Trap MRS reads of TTBR1_EL1 at EL1 using AArch64 to EL2.
- 0b0 MRS reads of TTBR1_EL1 are not trapped by this mechanism.
- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TTBR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.
**TTBR0_EL1, bit [36]**

Trap MRS reads of TTBR0_EL1 at EL1 using AArch64 to EL2.

- **0b0** MRS reads of TTBR0_EL1 are not trapped by this mechanism.
- **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TTBR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**TPIDR_EL0, bit [35]**

Trap MRS reads of TPIDR_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURW at EL0 using AArch32 when EL1 is using AArch64 to EL2.

- **0b0** MRS reads of TPIDR_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURW at EL0 using AArch32 are not trapped by this mechanism.
- **0b1** If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:
  - MRS reads of TPIDR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
  - MRC reads of TPIDRURW at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**TPIDRRO_EL0, bit [34]**

Trap MRS reads of TPIDRRO_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURO at EL0 using AArch32 when EL1 is using AArch64 to EL2.

- **0b0** MRS reads of TPIDRRO_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURO at EL0 using AArch32 are not trapped by this mechanism.
- **0b1** If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception:
  - MRS reads of TPIDRRO_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
  - MRC reads of TPIDRURO at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**TPIDR_EL1, bit [33]**

Trap MRS reads of TPIDR_EL1 at EL1 using AArch64 to EL2.

- **0b0** MRS reads of TPIDR_EL1 are not trapped by this mechanism.
- **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TPIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.
TCR_EL1, bit [32]

Trap MRS reads of TCR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of TCR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

SCXTNUM_EL0, bit [31]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Trap MRS reads of SCXTNUM_EL0 at EL1 and EL0 using AArch64 to EL2.

0b0 MRS reads of SCXTNUM_EL0 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != \{1, 1\}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of SCXTNUM_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

SCXTNUM_EL1, bit [30]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Trap MRS reads of SCXTNUM_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of SCXTNUM_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of SCXTNUM_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

SCTLR_EL1, bit [29]

Trap MRS reads of SCTLR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of SCTLR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of SCTLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

REVIDR_EL1, bit [28]

Trap MRS reads of REVIDR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of REVIDR_EL1 are not trapped by this mechanism.
If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGTEn == 1`, then `MRS` reads of `REVIDR_EL1` at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**PAR_EL1, bit [27]**
Trap `MRS` reads of `PAR_EL1` at EL1 using AArch64 to EL2.
- 0b0 `MRS` reads of `PAR_EL1` are not trapped by this mechanism.
- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGTEn == 1`, then `MRS` reads of `PAR_EL1` at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**MPIDR_EL1, bit [26]**
Trap `MRS` reads of `MPIDR_EL1` at EL1 using AArch64 to EL2.
- 0b0 `MRS` reads of `MPIDR_EL1` are not trapped by this mechanism.
- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGTEn == 1`, then `MRS` reads of `MPIDR_EL1` at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**MIDR_EL1, bit [25]**
Trap `MRS` reads of `MIDR_EL1` at EL1 using AArch64 to EL2.
- 0b0 `MRS` reads of `MIDR_EL1` are not trapped by this mechanism.
- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGTEn == 1`, then `MRS` reads of `MIDR_EL1` at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**MAIR_EL1, bit [24]**
Trap `MRS` reads of `MAIR_EL1` at EL1 using AArch64 to EL2.
- 0b0 `MRS` reads of `MAIR_EL1` are not trapped by this mechanism.
- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or `SCR_EL3.FGTEn == 1`, then `MRS` reads of `MAIR_EL1` at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**LORSA_EL1, bit [23]**
*When FEAT_LOR is implemented:*
Trap `MRS` reads of `LORSA_EL1` at EL1 using AArch64 to EL2.
- 0b0 `MRS` reads of `LORSA_EL1` are not trapped by this mechanism.
If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LORSA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

* On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**LORN_EL1, bit [22]**

_When FEAT_LOR is implemented:_

Trap MRS reads of LORN_EL1 at EL1 using AArch64 to EL2.

0b0  MRS reads of LORN_EL1 are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LORN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

* On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**LORID_EL1, bit [21]**

_When FEAT_LOR is implemented:_

Trap MRS reads of LORID_EL1 at EL1 using AArch64 to EL2.

0b0  MRS reads of LORID_EL1 are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LORID_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

* On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**LOREA_EL1, bit [20]**

_When FEAT_LOR is implemented:_

Trap MRS reads of LOREA_EL1 at EL1 using AArch64 to EL2.

0b0  MRS reads of LOREA_EL1 are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LOREA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

* On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**

Reserved, RES0.
LORC_EL1, bit [19]

When FEAT_LOR is implemented:

Trap $MRS$ reads of LORC_EL1 at EL1 using AArch64 to EL2.

0b0  $MRS$ reads of LORC_EL1 are not trapped by this mechanism.
0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then $MRS$ reads of LORC_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

ISR_EL1, bit [18]

Trap $MRS$ reads of ISR_EL1 at EL1 using AArch64 to EL2.

0b0  $MRS$ reads of ISR_EL1 are not trapped by this mechanism.
0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then $MRS$ reads of ISR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

FAR_EL1, bit [17]

Trap $MRS$ reads of FAR_EL1 at EL1 using AArch64 to EL2.

0b0  $MRS$ reads of FAR_EL1 are not trapped by this mechanism.
0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then $MRS$ reads of FAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

ESR_EL1, bit [16]

Trap $MRS$ reads of ESR_EL1 at EL1 using AArch64 to EL2.

0b0  $MRS$ reads of ESR_EL1 are not trapped by this mechanism.
0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then $MRS$ reads of ESR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

DCZID_EL0, bit [15]

Trap $MRS$ reads of DCZID_EL0 at EL1 and EL0 using AArch64 to EL2.

0b0  $MRS$ reads of DCZID_EL0 are not trapped by this mechanism.
0b1  If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} !={1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then $MRS$ reads of DCZID_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**CTR_EL0, bit [14]**

Trap MRS reads of CTR_EL0 at EL1 and EL0 using AArch64 to EL2.

0b0 MRS reads of CTR_EL0 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**CSSELR_EL1, bit [13]**

Trap MRS reads of CSSELR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of CSSELR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CSSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**CPACR_EL1, bit [12]**

Trap MRS reads of CPACR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of CPACR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CPACR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**CONTEXTIDR_EL1, bit [11]**

Trap MRS reads of CONTEXTIDR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of CONTEXTIDR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CONTEXTIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**CLIDR_EL1, bit [10]**

Trap MRS reads of CLIDR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of CLIDR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CLIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

CCSIDR_EL1, bit [9]

Trap MRS reads of CCSIDR_EL1 at EL1 using AArch64 to EL2.
0b0 MRS reads of CCSIDR_EL1 are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CCSIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

APIBKey, bit [8]

When FEAT_PAuth is implemented:
Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:
• APIBKeyHi_EL1.
• APIBKeyLo_EL1.
0b0 MRS reads of the System registers listed above are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

APIAKey, bit [7]

When FEAT_PAuth is implemented:
Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:
• APIAKeyHi_EL1.
• APIAKeyLo_EL1.
0b0 MRS reads of the System registers listed above are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

APGAKey, bit [6]

When FEAT_PAuth is implemented:
Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:
• APGAKeyHi_EL1.
• APGAKeyLo_EL1.
  
 0b0  MSR reads of the System registers listed above are not trapped by this mechanism.
  0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

  The reset behavior of this field is:
  • On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

  Otherwise:
  Reserved, RES0.

APDBKey, bit [5]

When FEAT_PAuth is implemented:
  Trap MSR reads of multiple System registers. Enables a trap on MSR reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:
  • APDBKeyHi_EL1.
  • APDBKeyLo_EL1.
  
 0b0  MSR reads of the System registers listed above are not trapped by this mechanism.
  0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

  The reset behavior of this field is:
  • On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

  Otherwise:
  Reserved, RES0.

APDAKey, bit [4]

When FEAT_PAuth is implemented:
  Trap MSR reads of multiple System registers. Enables a trap on MSR reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:
  • APDAKeyHi_EL1.
  • APDAKeyLo_EL1.
  
 0b0  MSR reads of the System registers listed above are not trapped by this mechanism.
  0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

  The reset behavior of this field is:
  • On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

  Otherwise:
  Reserved, RES0.

AMAIR_EL1, bit [3]

Trap MSR reads of AMAIR_EL1 at EL1 using AArch64 to EL2.
  
 0b0  MSR reads of AMAIR_EL1 are not trapped by this mechanism.
  0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR reads of AMAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**AIDR_EL1, bit [2]**

Trap MRS reads of AIDR_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of AIDR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**AFSR1_EL1, bit [1]**

Trap MRS reads of AFSR1_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of AFSR1_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AFSR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**AFSR0_EL1, bit [0]**

Trap MRS reads of AFSR0_EL1 at EL1 using AArch64 to EL2.

0b0 MRS reads of AFSR0_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AFSR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Accessing HFGRTR_EL2**

Accesses to this register use the following encodings in the System register encoding space:

```plaintext
MRS <Xt>, HFGRTR_EL2
```

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV2,NV == '11' then
    X[t, 64] = NVMem[0x1B8]
  elsif EL2Enabled() && AArch64.SystemAccessTrap(EL2, 0x18)
    else
      UNDEFINED;
  elsif PSTATE_EL == EL2 then
    if Halted() && El3 & HaveEl(EL3) && EDS肥料.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority"
when SDD == '1' && SCR_EL3.FGTEn == '0' then
    UNDEFINED;
elsif HaveEL(EL3) && SCR_EL3.FGTEn == '0' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = HFGRTR_EL2;
else
    X[t, 64] = HFGRTR_EL2;

**MSR HFGRTR_EL2, <Xt>**

### MSR HFGRTR_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV2,NV> == '11' then
        NVMem[0x1B8] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.FGTEn == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.FGTEn == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            HFGRTR_EL2 = X[t, 64];
    elsif PSTATE.EL == EL3 then
        HFGRTR_EL2 = X[t, 64];
D17.2.54 HFGWTR_EL2, Hypervisor Fine-Grained Write Trap Register

The HFGWTR_EL2 characteristics are:

**Purpose**

Provides controls for traps of MSR and MCR writes of System registers.

**Configurations**

This register is present only when FEAT_FGT is implemented. Otherwise, direct accesses to HFGWTR_EL2 are UNDEFINED.

**Attributes**

HFGWTR_EL2 is a 64-bit register.

**Field descriptions**

The HFGWTR_EL2 register is a 64-bit register with specific fields described in detail.

**Bits [63:56]**

Reserved, RES0.

**nTPIDR2_EL0, bit [55]**

*When FEAT_SME is implemented:*

Trap MSR writes of TPIRDR_EL0 at EL1 and EL0 using AArch64 to EL2.

- **0b0** If EL2 is implemented and enabled in the current Security state, HCR_EL2.E2H, TGE = 1, and either EL3 is not implemented or SCR_EL3.FGTEn = 1, then MSR writes of TPID2_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

- **0b1** MSR writes of TPID2_EL0 are not trapped by this mechanism.
The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**nSMPRI_EL1, bit [54]**

When FEAT_SME is implemented:

- Trap MSR writes of SMPRI_EL1 at EL1 using AArch64 to EL2.

  - 0b0 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of SMPRI_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

  - 0b1 MSR writes of SMPRI_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**Bits [53:51]**
Reserved, RES0.

**nACCDATA_EL1, bit [50]**

When FEAT_LS64_ACCDATA is implemented:

- Trap MSR writes of ACCDATA_EL1 at EL1 using AArch64 to EL2.

  - 0b0 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ACCDATA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

  - 0b1 MSR writes of ACCDATA_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**ERXADDR_EL1, bit [49]**

When FEAT_RAS is implemented:

- Trap MSR writes of ERXADDR_EL1 at EL1 using AArch64 to EL2.

  - 0b0 MSR writes of ERXADDR_EL1 are not trapped by this mechanism.

  - 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXADDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.
ERXPFGCDN_EL1, bit [48]

*When FEAT_RASv1p1 is implemented:*

- Trap MSR writes of ERXPFGCDN_EL1 at EL1 using AArch64 to EL2.

  - **0b0** MSR writes of ERXPFGCDN_EL1 are not trapped by this mechanism.
  - **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXPFGCDN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

*Otherwise:*

Reserved, RES0.

ERXPFGCTEL1, bit [47]

*When FEAT_RASv1p1 is implemented:*

- Trap MSR writes of ERXPFGCTEL1 at EL1 using AArch64 to EL2.

  - **0b0** MSR writes of ERXPFGCTEL1 are not trapped by this mechanism.
  - **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXPFGCTEL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

*Otherwise:*

Reserved, RES0.

Bit [46]

Reserved, RES0.

ERXMISCn_EL1, bit [45]

*When FEAT_RAS is implemented:*

- Trap MSR writes of ERXMISCn_EL1 at EL1 using AArch64 to EL2.

  - **0b0** MSR writes of ERXMISCn_EL1 are not trapped by this mechanism.
  - **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXMISCn_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

*Otherwise:*

Reserved, RES0.

ERXSTATUS_EL1, bit [44]

*When FEAT_RAS is implemented:*

- Trap MSR writes of ERXSTATUS_EL1 at EL1 using AArch64 to EL2.

  - **0b0** MSR writes of ERXSTATUS_EL1 are not trapped by this mechanism.
  - **0b1** If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXSTATUS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
The reset behavior of this field is:
  • On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**ERXCTLR_EL1, bit [43]**

*When FEAT_RAS is implemented:*

- Trap MSR writes of ERXCTLR_EL1 at EL1 using AArch64 to EL2.
- 0b0 MSR writes of ERXCTLR_EL1 are not trapped by this mechanism.
- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXCTLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
  • On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**Bit [42]**

Reserved, RES0.

**ERRSELR_EL1, bit [41]**

*When FEAT_RAS is implemented:*

- Trap MSR writes of ERRSELR_EL1 at EL1 using AArch64 to EL2.
- 0b0 MSR writes of ERRSELR_EL1 are not trapped by this mechanism.
- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERRSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
  • On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**Bit [40]**

Reserved, RES0.

**ICC_IGRPEN<n>_EL1, bit [39]**

*When FEAT_GICv3 is implemented:*

- Trap MSR writes of ICC_IGRPEN<n>_EL1 at EL1 using AArch64 to EL2.
- 0b0 MSR writes of ICC_IGRPEN<n>_EL1 are not trapped by this mechanism.
- 0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ICC_IGRPEN<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
  • On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.
VBAR_EL1, bit [38]

Trap MSR writes of VBAR_EL1 at EL1 using AArch64 to EL2.

0b0  MSR writes of VBAR_EL1 are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of VBAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

TTBR1_EL1, bit [37]

Trap MSR writes of TTBR1_EL1 at EL1 using AArch64 to EL2.

0b0  MSR writes of TTBR1_EL1 are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TTBR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

TTBR0_EL1, bit [36]

Trap MSR writes of TTBR0_EL1 at EL1 using AArch64 to EL2.

0b0  MSR writes of TTBR0_EL1 are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TTBR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

TPIDR_EL0, bit [35]

Trap MSR writes of TPIDR_EL0 at EL1 and EL0 using AArch64 and MCR writes of TPIDRURW at EL0 using AArch32 when EL1 is using AArch64 to EL2.

0b0  MSR writes of TPIDR_EL0 at EL1 and EL0 using AArch64 and MCR writes of TPIDRURW at EL0 using AArch32 are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception:
- MSR writes of TPIDR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18.
- MCR writes of TPIDRURW at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x3.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

TPIDRRO_EL0, bit [34]

Trap MSR writes of TPIDRRO_EL0 at EL1 using AArch64 to EL2.

0b0  MSR writes of TPIDRRO_EL0 are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TPIDR_EL0 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

TPIDR_EL1, bit [33]
Trap MSR writes of TPIDR_EL1 at EL1 using AArch64 to EL2.
0b0 MSR writes of TPIDR_EL1 are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TPIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

TCR_EL1, bit [32]
Trap MSR writes of TCR_EL1 at EL1 using AArch64 to EL2.
0b0 MSR writes of TCR_EL1 are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TCR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

SCXTNUM_EL0, bit [31]
When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:
Trap MSR writes of SCXTNUM_EL0 at EL1 and EL0 using AArch64 to EL2.
0b0 MSR writes of SCXTNUM_EL0 are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of SCXTNUM_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.

SCXTNUM_EL1, bit [30]
When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:
Trap MSR writes of SCXTNUM_EL1 at EL1 using AArch64 to EL2.
0b0 MSR writes of SCXTNUM_EL1 are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of SCXTNUM_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.
Otherwise:
Reserved, RES0.

SCTLR_EL1, bit [29]
Trap MSR writes of SCTLR_EL1 at EL1 using AArch64 to EL2.
0b0 MSR writes of SCTLR_EL1 are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of SCTLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Bit [28]
Reserved, RES0.

PAR_EL1, bit [27]
Trap MSR writes of PAR_EL1 at EL1 using AArch64 to EL2.
0b0 MSR writes of PAR_EL1 are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Bits [26:25]
Reserved, RES0.

MAIR_EL1, bit [24]
Trap MSR writes of MAIR_EL1 at EL1 using AArch64 to EL2.
0b0 MSR writes of MAIR_EL1 are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of MAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

LORSA_EL1, bit [23]

When FEAT_LOR is implemented:
Trap MSR writes of LORSA_EL1 at EL1 using AArch64 to EL2.
0b0 MSR writes of LORSA_EL1 are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of LORSA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:
Reserved, RES0.
LORN_EL1, bit [22]

When FEAT_LOR is implemented:

- Trap MSR writes of LORN_EL1 at EL1 using AArch64 to EL2.
- 0b0  MSR writes of LORN_EL1 are not trapped by this mechanism.
- 0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of LORN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

Bit [21]

Reserved, RES0.

LOREA_EL1, bit [20]

When FEAT_LOR is implemented:

- Trap MSR writes of LOREA_EL1 at EL1 using AArch64 to EL2.
- 0b0  MSR writes of LOREA_EL1 are not trapped by this mechanism.
- 0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of LOREA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

LORC_EL1, bit [19]

When FEAT_LOR is implemented:

- Trap MSR writes of LORC_EL1 at EL1 using AArch64 to EL2.
- 0b0  MSR writes of LORC_EL1 are not trapped by this mechanism.
- 0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of LORC_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

Bit [18]

Reserved, RES0.

FAR_EL1, bit [17]

Trap MSR writes of FAR_EL1 at EL1 using AArch64 to EL2.

- 0b0  MSR writes of FAR_EL1 are not trapped by this mechanism.
If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of FAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

ESR_EL1, bit [16]
Trap MSR writes of ESR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of ESR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ESR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Bits [15:14]
Reserved, RES0.

CSSELR_EL1, bit [13]
Trap MSR writes of CSSELR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of CSSELR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of CSSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

CPACR_EL1, bit [12]
Trap MSR writes of CPACR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of CPACR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of CPACR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

CONTEXTIDR_EL1, bit [11]
Trap MSR writes of CONTEXTIDR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of CONTEXTIDR_EL1 are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of CONTEXTIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Bits [10:9]
Reserved, RES0.
APIBKey, bit [8]

When FEAT_PAuth is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIBKeyHi_EL1.
- APIBKeyLo_EL1.

0b0 MSR writes of the System registers listed above are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

APIAKey, bit [7]

When FEAT_PAuth is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIAKeyHi_EL1.
- APIAKeyLo_EL1.

0b0 MSR writes of the System registers listed above are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

APGAKey, bit [6]

When FEAT_PAuth is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APGAKeyHi_EL1.
- APGAKeyLo_EL1.

0b0 MSR writes of the System registers listed above are not trapped by this mechanism.

0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.
APDBKey, bit [5]

When FEAT_PAuth is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APDBKeyHi_EL1.
- APDBKeyLo_EL1.

0b0 MSR writes of the System registers listed above are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

APDAKey, bit [4]

When FEAT_PAuth is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APDAKeyHi_EL1.
- APDAKeyLo_EL1.

0b0 MSR writes of the System registers listed above are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Otherwise:

Reserved, RES0.

AMAIR_EL1, bit [3]

Trap MSR writes of AMAIR_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of AMAIR_EL1 are not trapped by this mechanism.
0b1 If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of AMAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Bit [2]

Reserved, RES0.

AFSR1_EL1, bit [1]

Trap MSR writes of AFSR1_EL1 at EL1 using AArch64 to EL2.

0b0 MSR writes of AFSR1_EL1 are not trapped by this mechanism.
If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of AFSR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**AFSR0_EL1, bit [0]**

Trap MSR writes of AFSR0_EL1 at EL1 using AArch64 to EL2.

0b0  MSR writes of AFSR0_EL1 are not trapped by this mechanism.

0b1  If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of AFSR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Accessing HFGWTR_EL2**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, HFGWTR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b101</td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = NVMem[0x1C0];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.FGTEn == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) & SCR_EL3.FGTEn == '0' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = HFGWTR_EL2;
elsif PSTATE_EL == EL3 then
  X[t, 64] = HFGWTR_EL2;
```

```c
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = NVMem[0x1C0];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.FGTEn == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) & SCR_EL3.FGTEn == '0' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = HFGWTR_EL2;
elsif PSTATE_EL == EL3 then
  X[t, 64] = HFGWTR_EL2;
```
if PSTATE.EZ == EL0 then
    UNDEFINED;
elsif PSTATE.EZ == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2, NV> == '11' then
        NVMem[0x1c0] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    end
elsif PSTATE.EZ == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' && SCR_EL3.FGTEn == '0'" then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.FGTEn == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        HFGWTR_EL2 = X[t, 64];
    end
elsif PSTATE.EZ == EL3 then
    HFGWTR_EL2 = X[t, 64];
else

---

### MSR HFGWTR_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b101</td>
</tr>
</tbody>
</table>
D17.2.55 HPFAR_EL2, Hypervisor IPA Fault Address Register

The HPFAR_EL2 characteristics are:

**Purpose**

Holds the faulting IPA for some aborts on a stage 2 translation taken to EL2.

**Configurations**

AArch64 System register HPFAR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HPFAR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

The HPFAR_EL2 is written for:

- Translation or Access faults in the second stage of translation.
- An abort in the second stage of translation performed during the translation table walk of a first stage translation, caused by a Translation fault, an Access flag fault, or a Permission fault.
- A stage 2 Address size fault.
- If FEAT_RME is implemented, a Granule Protection Check fault in the second stage of translation.

For all other exceptions taken to EL2, this register is **UNKNOWN**.

--- **Note**

The address held in this register is an address accessed by the instruction fetch or data access that caused the exception that gave rise to the Instruction Abort exception or Data Abort exception. It is the lowest address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

**Attributes**

HPFAR_EL2 is a 64-bit register.

**Field descriptions**

![Diagram of HPFAR_EL2 field descriptions]

Execution at EL1 or EL0 makes HPFAR_EL2 become **UNKNOWN**.

**NS, bit [63]**

*When FEAT_SEL2 is implemented:*

Faulting IPA address space.

- 0b0 Faulting IPA is from the Secure IPA space.
- 0b1 Faulting IPA is from the Non-secure IPA space.

For Data Abort exceptions or Instruction Abort exceptions taken to Non-secure EL2:

- This field is **RES0**.
- The address is from the Non-secure IPA space.
If FEAT_RME is implemented, for Data Abort exceptions or Instruction Abort exceptions taken to Realm EL2:

- This field is RES0.
- The address is from the Realm IPA space.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [62:44]

Reserved, RES0.

FIPA, bits [43:4]

Reserved, RES0.

FIPA, bits [39:0]


For implementations with fewer than 52 physical address bits, the corresponding upper bits in this field are RES0.

When FEAT_MOPS is implemented, the value presented in FIPA on a synchronous exception that set the HPFAR_EL2 from any of the Memory Copy and Memory Set instructions is within the address range of the current stage 2 translation granule, aligned to the size of the current stage 2 translation granule, of the address that generated the Data abort.

Bits[(n-1):0] of the value are UNKNOWN, where $2^n$ is the current stage 2 translation granule size in bytes.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**FIPA encoding for when FEAT_LPA is not implemented**

Bits [39:36]

Reserved, RES0.

FIPA, bits [35:0]

Reserved, RES0.
Bits\((n-1):0\) of the value are UNKNOWN, where \(2^n\) is the current stage 2 translation granule size in bytes.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits \([3:0]\)**

Reserved, RES0.

**Accessing HPFAR_EL2**

Accesses to this register use the following encodings in the System register encoding space:

**\(MRS \langle Xt\rangle, HPFAR_EL2\)**

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b100 & 0b0110 & 0b0000 & 0b100 \\
\end{array}
\]

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        X[t, 64] = HPFAR_EL2;
    elsif PSTATE.EL == EL3 then
        X[t, 64] = HPFAR_EL2;

**MSR HPFAR_EL2, \(\langle Xt\rangle\)**

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b100 & 0b0110 & 0b0000 & 0b100 \\
\end{array}
\]

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        HPFAR_EL2 = X[t, 64];
    elsif PSTATE.EL == EL3 then
        HPFAR_EL2 = X[t, 64];
**D17.2.56  HSTR_EL2, Hypervisor System Trap Register**

The HSTR_EL2 characteristics are:

**Purpose**

Controls trapping to EL2 of EL1 or lower AArch32 accesses to the System register in the coproc == 0b1111 encoding space, by the CRn value used to access the register using MCR or MRC instruction. When the register is accessible using an MCRR or MRRC instruction, this is the CRm value used to access the register.

**Configurations**

AArch64 System register HSTR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HSTR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

HSTR_EL2 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:

```
+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+
| 63                  | 62                  | 61                  | 60                  | 59                  | 58                  | 57                  | 56                  | 55                  | 54                  | 53                  | 52                  | 51                  | 50                  | 49                  | 48                  | 47                  | 46                  | 45                  | 44                  | 43                  | 42                  | 41                  | 40                  | 39                  | 38                  | 37                  | 36                  | 35                  | 34                  | 33                  | 32                  |
| 31                  | 30                  | 29                  | 28                  | 27                  | 26                  | 25                  | 24                  | 23                  | 22                  | 21                  | 20                  | 19                  | 18                  | 17                  | 16                  | 15                  | 14                  | 13                  | 12                  | 11                  | 10                  | 9                   | 8                   | 7                   | 6                   | 5                   | 4                   | 3                   | 2                   | 1                   | 0                   |
+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+
| RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | RES0               | T15               | T14               | T13               | T12               | T11               | T10               | T9                | T8                | T7                | T6                | T5                | T4                | T3                | T2                | T1                | T0                |
|                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |
```

**Bits [63:16, 14, 4]**

Reserved, RES0.

**T<n>, bit [n], for n = 15, 13 to 5, 3 to 0**

The remaining fields control whether EL0 and EL1 accesses, using MCR, MRC, MCRR, and MRRC instructions, to the System registers in the coproc == 0b1111 encoding space, are trapped to EL2 as follows:

- MCR or MRC accesses to these registers that are trapped to EL2 are reported using EC syndrome value 0x03, unless the access is UNDEFINED.
- MCRR or MRRC accesses to these registers that are trapped to EL2 are reported using EC syndrome value 0x04, unless the access is UNDEFINED.

**0b0**

This control has no effect on EL0 or EL1 accesses to System registers.

**0b1**

System registers in the coproc == 0b1111 encoding space and CRn == <n> or CRn == <n> where T<n> is the name of this field, are trapped as follows:

- An EL1 MCR or MRC access is trapped to EL2.
- An EL0 MCR or MRC access is trapped to EL2, if the access is not UNDEFINED when the value of this field is 0.
- An EL1 MCRR or MRRC access is trapped to EL2.
- An EL0 MCRR or MRRC access is trapped to EL2, if the access is not UNDEFINED when the value of this field is 0.
It is IMPLEMENTATION DEFINED whether an EL0 access using AArch32 is trapped to EL2, or is UNDEFINED.

If the access is UNDEFINED, and generates an exception that is taken to EL1 or EL2 using AArch64, this is reported with EC syndrome value 0x00.

--- Note ---

Arm expects that trapping to EL2 of EL0 accesses to these registers is unusual and used only when the hypervisor must virtualize EL0 operation. Arm recommends that, whenever possible, EL0 accesses to these registers behave as they would if the implementation did not include EL2. This means that, if the architecture does not support the EL0 access, then the register access instruction is treated as UNDEFINED and generates an exception that is taken to EL1.

For example, when HSTR_EL2.T7 is 1, for instructions executed at EL1:

- An MCR or MRC instruction with coproc set to 0b1111 and <CRn> set to c7 is trapped to EL2.
- An MCRR or MRRC instruction with coproc set to 0b1111 and <CRm> set to c7 is trapped to EL2.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

For example, when HSTR_EL2.T7 is 1, for instructions executed at EL1:

- An MCR or MRC instruction with coproc set to 0b1111 and <CRn> set to c7 is trapped to EL2.
- An MCRR or MRRC instruction with coproc set to 0b1111 and <CRm> set to c7 is trapped to EL2.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

--- Otherwise ---

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

### Accessing HSTR_EL2

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, HSTR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b011</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = NVMem[0x0800];
  elseif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
  end
elseif PSTATE.EL == EL2 then
  X[t, 64] = HSTR_EL2;
```

```
...```
elsif PSTATE.EL == EL3 then
    X[t, 64] = HSTR_EL2;

**MSR HSTR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x080] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        HSTR_EL2 = X[t, 64];
    elsif PSTATE.EL == EL3 then
        HSTR_EL2 = X[t, 64];
D17.2.57 ID_AA64AFR0_EL1, AArch64 Auxiliary Feature Register 0

The ID_AA64AFR0_EL1 characteristics are:

**Purpose**

Provides information about the IMPLEMENTATION DEFINED features of the PE in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

There are no configuration notes.

**Attributes**

ID_AA64AFR0_EL1 is a 64-bit register.

**Field descriptions**

Bits [63:32] Reserved, RES0.

**IMPLEMENTATION DEFINED, bits [31:28]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [27:24]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [23:20]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [19:16]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [15:12]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [11:8]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [7:4]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [3:0]**

IMPLEMENTATION DEFINED.
Accessing ID_AA64AFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[ MRS <Xt>, ID_AA64AFR0_EL1 \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0101</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TID3 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = ID_AA64AFR0_EL1;
elsif PSTATE.EL == EL2 then
  X[t, 64] = ID_AA64AFR0_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = ID_AA64AFR0_EL1;
D17.2.58  ID_AA64AFR1_EL1, AArch64 Auxiliary Feature Register 1

The ID_AA64AFR1_EL1 characteristics are:

**Purpose**

Reserved for future expansion of information about the IMPLEMENTATION DEFINED features of the
PE in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme
for fields in ID registers* on page D17-5551.

**Configurations**

There are no configuration notes.

**Attributes**

ID_AA64AFR1_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
</tr>
</tbody>
</table>

**Accessing ID_AA64AFR1_EL1**

Accesses to this register use the following encodings in the System register encoding space:

*MRS <Xt>, ID_AA64AFR1_EL1*

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0101</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TID3 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = ID_AA64AFR1_EL1;
elsif PSTATE.EL == EL2 then
  X[t, 64] = ID_AA64AFR1_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = ID_AA64AFR1_EL1;
D17.2.59  ID_AA64DFR0_EL1, AArch64 Debug Feature Register 0

The ID_AA64DFR0_EL1 characteristics are:

**Purpose**

Provides top level information about the debug system in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

The external register EDDFR gives information from this register.

**Attributes**

ID_AA64DFR0_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPMN0, bits [63:60]</td>
<td>Zero PMU event counters for a Guest operating system. Defined values are:</td>
<td>0b0000</td>
<td>Setting MDCR_EL2.HPMN to zero has <strong>CONSTRAINED UNPREDICTABLE</strong> behavior.</td>
<td>0b0001</td>
</tr>
<tr>
<td></td>
<td>All other values are reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If FEAT_PMUv3 is not implemented, FEAT_FGT is not implemented, or EL2 is not implemented, the only permitted value is 0b0000.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FEAT_HPMN0 implements the functionality identified by the value 0b0001.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>From Armv8.8, in an implementation that includes FEAT_PMUv3, FEAT_FGT, and EL2, the value 0b0000 is not permitted.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bits [59:56]</td>
<td>Reserved, RES0.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRBE, bits [55:52]</td>
<td>Branch Record Buffer Extension. Defined values are:</td>
<td>0b0000</td>
<td>Branch Record Buffer Extension not implemented.</td>
<td>0b0001</td>
</tr>
<tr>
<td></td>
<td>0b0010</td>
<td>As 0b0001, and adds support for branch recording at EL3.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>All other values are reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FEAT_BRBE implements the functionality identified by the value 0b0001.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FEAT_BRBEv1p1 implements the functionality identified by the value 0b010.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>From Armv9.3, if FEAT_BRBE is implemented, the value 0b0001 is not permitted.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### MTPMU, bits [51:48]

Multi-threaded PMU extension. Defined values are:

- **0b0000**: FEAT_MTPMU not implemented. 
  If FEAT_PMUv3 is implemented, it is IMPLEMENTATION DEFINED whether PMEVTYPER<\(n\)>_EL0.MT and PMEVTYPER<\(n\)>._MT are read/write or RES0.

- **0b0001**: FEAT_MTPMU and FEAT_PMUv3 implemented. 
  PMEVTYPER<\(n\)>_EL0.MT and PMEVTYPER<\(n\)>._MT are read/write. When FEAT_MTPMU is disabled, the Effective values of PMEVTYPER<\(n\)>_EL0.MT and PMEVTYPER<\(n\)>._MT are 0.

- **0b1111**: FEAT_MTPMU not implemented. 
  If FEAT_PMUv3 is implemented, PMEVTYPER<\(n\)>_EL0.MT and PMEVTYPER<\(n\)>._MT are RES0.

All other values are reserved.

FEAT_MTPMU implements the functionality identified by the value 0b0001.

From Armv8.6, in an implementation that includes FEAT_PMUv3, the value 0b0000 is not permitted.

In an implementation that does not include FEAT_PMUv3, the value 0b0001 is not permitted.

### TraceBuffer, bits [47:44]

Trace Buffer Extension. Defined values are:

- **0b0000**: Trace Buffer Extension not implemented.
- **0b0001**: Trace Buffer Extension implemented.

All other values are reserved.

FEAT_TRBE implements the functionality identified by the value 0b0001.

In any Armv9 implementation, if FEAT_ETE is implemented, the value 0b0000 is not permitted.

### TraceFilt, bits [43:40]

Armv8.4 Self-hosted Trace Extension version. Defined values are:

- **0b0000**: Armv8.4 Self-hosted Trace Extension not implemented.
- **0b0001**: Armv8.4 Self-hosted Trace Extension implemented.

All other values are reserved.

FEAT_TRF implements the functionality identified by the value 0b0001.

From Armv8.4, if an Embedded Trace Macrocell Architecture trace unit is implemented, the value 0b0000 is not permitted.

### DoubleLock, bits [39:36]

OS Double Lock implemented. Defined values are:

- **0b0000**: OS Double Lock implemented. OSDLR_EL1 is RW.
- **0b1111**: OS Double Lock not implemented. OSDLR_EL1 is RAZ/WI.

All other values are reserved.

FEAT_DoubleLock implements the functionality identified by the value 0b0000.

In Armv8.0, the only permitted value is 0b0000.

If FEAT_Debugv8p2 is implemented and FEAT_DoPD is not implemented, the permitted values are 0b0000 and 0b1111.

If FEAT_DoPD is implemented, the only permitted value is 0b1111.

### PMSVer, bits [35:32]

Statistical Profiling Extension version. Defined values are:

- **0b0000**: Statistical Profiling Extension not implemented.
- **0b0001**: Statistical Profiling Extension implemented.
0b0010  As 0b0001, and adds:
- Support for the Event packet Alignment flag.
- If FEAT_SVE is implemented, support for the Scalable Vector extensions to Statistical Profiling.

0b0011  As 0b0010, and adds:
- Discard mode.
- Extended event filtering, including the PMSNVFR_EL1 System register.
- Support for the OPTIONAL previous branch target Address packet.
- If FEAT_PMUv3 is implemented, controls to freeze the PMU event counters after an SPE buffer management event occurs.
- If FEAT_PMUv3 is implemented, the SAMPLE_FEED_BR, SAMPLE_FEED_EVENT, SAMPLE_FEED_LAT, SAMPLE_FEED_LD, SAMPLE_FEED_OP, and SAMPLE_FEED_ST PMU events.

0b0100  As 0b0011, and adds:
- If FEAT_MOPS is implemented, Operation Type packet encodings for Memory Copy and Set operations.
- If FEAT_MTE is implemented, Operation Type packet encodings for loads and stores of Allocation Tags.

All other values are reserved.

FEAT_SPE implements the functionality identified by the value 0b0001.
FEAT_SPEv1p1 implements the functionality identified by the value 0b0010.
FEAT_SPEv1p2 implements the functionality identified by the value 0b0011.
FEAT_SPEv1p3 implements the functionality identified by the value 0b100.

From Armv8.5, if FEAT_SPE is implemented, the value 0b0001 is not permitted.
From Armv8.7, if FEAT_SPE is implemented, the value 0b0010 is not permitted.
From Armv8.8, if FEAT_SPE is implemented, the value 0b0011 is not permitted.

CTX_CMPs, bits [31:28]
Number of breakpoints that are context-aware, minus 1. These are the highest numbered breakpoints.

Bits [27:24]
Reserved, RES0.

WRPs, bits [23:20]
Number of watchpoints, minus 1. The value of 0b0000 is reserved.

Bits [19:16]
Reserved, RES0.

BRPs, bits [15:12]
Number of breakpoints, minus 1. The value of 0b0000 is reserved.

PMUVer, bits [11:8]
Performance Monitors Extension version.
This field does not follow the standard ID scheme, but uses the alternative ID scheme described in Alternative ID scheme used for the Performance Monitors Extension version on page D17-5553

Defined values are:
0b0000  Performance Monitors Extension not implemented.
0b0001  Performance Monitors Extension, PMUv3 implemented.
PMUv3 for Armv8.1. As 0b0001, and adds support for:
- Extended 16-bit PMEVTYPEr<\alpha>_{EL0}.evtCount field.
- If EL2 is implemented, the MDCR_{EL2}.HPMD control.

0b0101 PMUv3 for Armv8.4. As 0b0100, and adds support for the PMMIR_EL1 register.

0b0110 PMUv3 for Armv8.5. As 0b0101, and adds support for:
- 64-bit event counters.
- If EL2 is implemented, the MDCR_{EL2}.HCCD control.
- If EL3 is implemented, the MDCR_{EL3}.SCCD control.

0b0111 PMUv3 for Armv8.7. As 0b0110, and adds support for:
- The PMCR_{EL0}.FZO and, if EL2 is implemented, MDCR_{EL2}.HPMFZO controls.
- If EL3 is implemented, the MDCR_{EL3}.\{MPMX,MCCD\} controls.

0b1000 PMUv3 for Armv8.8. As 0b0111, and:
- Extends the Common event number space to include 0x0040 to 0x08BF and 0x4040 to 0x40BF.
- Removes the CONSTRAINED UNPREDICTABLE behaviors if a reserved or unimplemented PMU event number is selected.

0b1111 IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported. Arm does not recommend this value for new implementations.

All other values are reserved.

FEAT_PMUv3 implements the functionality identified by the value 0b0001.
FEAT_PMUv3p1 implements the functionality identified by the value 0b0100.
FEAT_PMUv3p4 implements the functionality identified by the value 0b0101.
FEAT_PMUv3p5 implements the functionality identified by the value 0b0110.
FEAT_PMUv3p7 implements the functionality identified by the value 0b0111.
FEAT_PMUv3p8 implements the functionality identified by the value 0b1000.

From Armv8.1, if FEAT_PMUv3 is implemented, the value 0b0001 is not permitted.
From Armv8.4, if FEAT_PMUv3 is implemented, the value 0b0100 is not permitted.
From Armv8.5, if FEAT_PMUv3 is implemented, the value 0b0101 is not permitted.
From Armv8.7, if FEAT_PMUv3 is implemented, the value 0b0110 is not permitted.
From Armv8.8, if FEAT_PMUv3 is implemented, the value 0b0111 is not permitted.

TraceVer, bits [7:4]
Trace support. Indicates whether System register interface to a trace unit is implemented. Defined values are:
- 0b0000 Trace unit System registers not implemented.
- 0b0001 Trace unit System registers implemented.

All other values are reserved.

When trace unit System registers are implemented, see TRCIDR1 for tracing capabilities of the trace unit.

DebugVer, bits [3:0]
Debug architecture version. Indicates presence of Armv8 debug architecture. Defined values are:
- 0b0110 Armv8 debug architecture.
- 0b0111 Armv8 debug architecture with Virtualization Host Extensions.
- 0b1000 Armv8.2 debug architecture, FEAT_Debugv8p2.
- 0b1001 Armv8.4 debug architecture, FEAT_Debugv8p4.
0b1010  Armv8.8 debug architecture, FEAT_Debugv8p8.

All other values are reserved.

FEAT_VHE adds the functionality identified by the value 0b0111.
FEAT_Debugv8p2 adds the functionality identified by the value 0b1000.
FEAT_Debugv8p4 adds the functionality identified by the value 0b1001.
FEAT_Debugv8p8 adds the functionality identified by the value 0b1010.

From Armv8.1, when FEAT_VHE is implemented the value 0b0110 is not permitted.
From Armv8.2, the values 0b0110 and 0b1011 are not permitted.
From Armv8.4, the value 0b1000 is not permitted.
From Armv8.8, the value 0b1001 is not permitted.

Accessing ID_AA64DFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, ID_AA64DFR0_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0101</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    else
      UNDEFINED;
  elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID3 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      X[t, 64] = ID_AA64DFR0_EL1;
    elseif PSTATE_EL == EL2 then
      X[t, 64] = ID_AA64DFR0_EL1;
    elseif PSTATE_EL == EL3 then
      X[t, 64] = ID_AA64DFR0_EL1;
D17.2.60 ID_AA64DFR1_EL1, AArch64 Debug Feature Register 1

The ID_AA64DFR1_EL1 characteristics are:

Purpose

Reserved for future expansion of top level information about the debug system in AArch64 state.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers on page D17-5551.

Configurations

There are no configuration notes.

Attributes

ID_AA64DFR1_EL1 is a 64-bit register.

Field descriptions

Bits [63:0]

Reserved, RES0.

Accessing ID_AA64DFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, ID_AA64DFR1_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b0101</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TID3 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = ID_AA64DFR1_EL1;
elsif PSTATE.EL == EL2 then
  X[t, 64] = ID_AA64DFR1_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = ID_AA64DFR1_EL1;
D17.2.61 ID_AA64ISAR0_EL1, AArch64 Instruction Set Attribute Register 0

The ID_AA64ISAR0_EL1 characteristics are:

**Purpose**

Provides information about the instructions implemented in AArch64 state.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers on page D17-5551.

**Configurations**

There are no configuration notes.

**Attributes**

ID_AA64ISAR0_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNDR</td>
<td>63:60</td>
<td>Indicates support for Random Number instructions in AArch64 state.</td>
</tr>
<tr>
<td>TLB</td>
<td>59:56</td>
<td>Indicates support for Outer Shareable and TLB range maintenance instructions.</td>
</tr>
<tr>
<td>TS</td>
<td>55:52</td>
<td>Indicates support for flag manipulation instructions.</td>
</tr>
</tbody>
</table>

**RNDR, bits [63:60]**

Indicates support for Random Number instructions in AArch64 state.

When FEAT_RNG_TRAP is implemented, the value returned by a direct read of ID_AA64ISAR0_EL1.RNDR is further controlled by the value of SCR_EL3.TRNDR.

Defined values are:

- 0b0000: No Random Number instructions are implemented.
- 0b0001: RNDR and RNDRRS registers are implemented.
- All other values are reserved.

FEAT_RNG implements the functionality identified by the value 0b0001.

From Armv8.5, the permitted values are 0b0000 and 0b0001.

**TLB, bits [59:56]**

Indicates support for Outer Shareable and TLB range maintenance instructions. Defined values are:

- 0b0000: Outer Shareable and TLB range maintenance instructions are not implemented.
- 0b0001: Outer Shareable TLB maintenance instructions are implemented.
- 0b0010: Outer Shareable and TLB range maintenance instructions are implemented.

All other values are reserved.

FEAT_TLBBIOS implements the functionality identified by the values 0b0001 and 0b0010.

FEAT_TLBIRANGE implements the functionality identified by the value 0b0010.

From Armv8.4, the only permitted value is 0b0010.

**TS, bits [55:52]**

Indicates support for flag manipulation instructions. Defined values are:

- 0b0000: No flag manipulation instructions are implemented.
- 0b0001: CFINV, RMIF, SETF16, and SETF8 instructions are implemented.
- 0b0010: CFINV, RMIF, SETF16, SETF8, AXFLAG, and XAFLAG instructions are implemented.
All other values are reserved.

**FEAT_FlagM** implements the functionality identified by the value 0b0001.

In Armv8.2, the permitted values are 0b0000 and 0b0010.

In Armv8.4, the only permitted value is 0b0001.

From Armv8.5, the only permitted value is 0b0010.

**FHM, bits [51:48]**

Indicates support for FMLAL and FMLSL instructions. Defined values are:

- 0b0000  FMLAL and FMLSL instructions are not implemented.
- 0b0001  FMLAL and FMLSL instructions are implemented.

All other values are reserved.

**FEAT_FHM** implements the functionality identified by the value 0b0001.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

**DP, bits [47:44]**

Indicates support for Dot Product instructions in AArch64 state. Defined values are:

- 0b0000  No Dot Product instructions implemented.
- 0b0001  UDOT and SDOT instructions implemented.

All other values are reserved.

**FEAT_DotProd** implements the functionality identified by the value 0b0001.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

**SM4, bits [43:40]**

Indicates support for SM4 instructions in AArch64 state. Defined values are:

- 0b0000  No SM4 instructions implemented.
- 0b0001  SM4E and SM4EKEY instructions implemented.

All other values are reserved.

If **FEAT_SM4** is not implemented, the value 0b0001 is reserved.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

This field must have the same value as ID_AA64ISAR0_EL1.SM3.

**SM3, bits [39:36]**

Indicates support for SM3 instructions in AArch64 state. Defined values are:

- 0b0000  No SM3 instructions implemented.
- 0b0001  SM3SS1, SM3TT1A, SM3TT1B, SM3TT2A, SM3TT2B, SM3PARTW1, and SM3PARTW2 instructions implemented.

All other values are reserved.

If **FEAT_SM3** is not implemented, the value 0b0001 is reserved.

**FEAT_SM3** implements the functionality identified by the value 0b0001.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

This field must have the same value as ID_AA64ISAR0_EL1.SM4.

**SHA3, bits [35:32]**

Indicates support for SHA3 instructions in AArch64 state. Defined values are:

- 0b0000  No SHA3 instructions implemented.
- 0b0001  EOR3, RAX1, XAR, and BCAX instructions implemented.

All other values are reserved.
If FEAT_SHA3 is not implemented, the value 0b0001 is reserved.

FEAT_SHA3 implements the functionality identified by the value 0b0001.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

If the value of ID_AA64ISAR0_EL1.SHA1 is 0b0000, this field must have the value 0b0000.

If the value of this field is 0b0001, ID_AA64ISAR0_EL1.SHA2 must have the value 0b0010.

RDM, bits [31:28]
Indicates support for SQRDMLAH and SQRDMLSH instructions in AArch64 state. Defined values are:

0b0000  No RDMA instructions implemented.
0b0001  SQRDMLAH and SQRDMLSH instructions implemented.

All other values are reserved.

FEAT_RDM implements the functionality identified by the value 0b0001.

From Armv8.1, the only permitted value is 0b0001.

TME, bits [27:24]
Indicates support for TME instructions. Defined values are:

0b0000  TME instructions are not implemented.
0b0001  TCANCEL, TCOMMIT, TSTART, and TTEST instructions are implemented.

All other values are reserved.

Accessing this field has the following behavior:

•  RAZ/WI if all of the following are true:
   •  PSTATE.EL IN {EL2, EL1}
   •  SCR_EL3.TME == 0

•  RAZ/WI if all of the following are true:
   •  PSTATE.EL == EL1
   •  EL2Enabled()
   •  HCR_EL2.TME == 0

•  Otherwise, access to this field is RO

Atomic, bits [23:20]
Indicates support for Atomic instructions in AArch64 state. Defined values are:

0b0000  No Atomic instructions implemented.
0b010  LDADD, LDCLR, LDEOR, LDSET, LDSMAX, LDSMIN, LDUMAX, LDUMIN, CAS, CASP, and SWP instructions implemented.

All other values are reserved.

FEAT_LSE implements the functionality identified by the value 0b0010.

From Armv8.1, the only permitted value is 0b0010.

CRC32, bits [19:16]
Indicates support for CRC32 instructions in AArch64 state. Defined values are:

0b0000  No CRC32 instructions implemented.
0b0001  CRC32B, CRC32H, CRC32W, CRC32X, CRC32CB, CRC32CH, CRC32CW, and CRC32CX instructions implemented.

All other values are reserved.

In Armv8.0, the permitted values are 0b0000 and 0b0001.

From Armv8.1, the only permitted value is 0b0001.
SHA2, bits [15:12]
Indicates support for SHA2 instructions in AArch64 state. Defined values are:
0b0000 No SHA2 instructions implemented.
0b0001 Implements instructions: SHA256H, SHA256H2, SHA256SU0, and SHA256SU1.
0b0010 Implements instructions:
  • SHA256H, SHA256H2, SHA256SU0, and SHA256SU1.
  • SHA512H, SHA512H2, SHA512SU0, and SHA512SU1.
All other values are reserved.
FEAT_SHA256 implements the functionality identified by the value 0b0001.
FEAT_SHA512 implements the functionality identified by the value 0b0010.
In Armv8, the permitted values are 0b0000 and 0b0001.
From Armv8.2, the permitted values are 0b0000, 0b0001, and 0b0010.
If the value of ID_AA64ISAR0_EL1.SHA1 is 0b0000, this field must have the value 0b0000.
If the value of this field is 0b0010, ID_AA64ISAR0_EL1.SHA3 must have the value 0b0001.

SHA1, bits [11:8]
Indicates support for SHA1 instructions in AArch64 state. Defined values are:
0b0000 No SHA1 instructions implemented.
0b0001 SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 instructions implemented.
All other values are reserved.
FEAT_SHA1 implements the functionality identified by the value 0b0001.
From Armv8, the permitted values are 0b0000 and 0b0001.
If the value of ID_AA64ISAR0_EL1.SHA2 is 0b0000, this field must have the value 0b0000.

AES, bits [7:4]
Indicates support for AES instructions in AArch64 state. Defined values are:
0b0000 No AES instructions implemented.
0b0001 AESE, AESD, AESMC, and AESIMC instructions implemented.
0b0010 As for 0b0001, plus PMULL/PMULL2 instructions operating on 64-bit data quantities.
FEAT_AES implements the functionality identified by the value 0b0001.
FEAT_PMULL implements the functionality identified by the value 0b0010.
All other values are reserved.
From Armv8, the permitted values are 0b0000 and 0b0010.

Bits [3:0]
Reserved, RES0.

Accessing ID_AA64ISAR0_EL1
Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, ID_AA64ISAR0_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        else
            UNDEFINED;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && HCR_EL2.TID3 == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            X[t, 64] = ID_AA64ISAR0_EL1;
    elsif PSTATE.EL == EL2 then
        X[t, 64] = ID_AA64ISAR0_EL1;
    elsif PSTATE.EL == EL3 then
        X[t, 64] = ID_AA64ISAR0_EL1;
D17.2.62 ID_AA64ISAR1_EL1, AArch64 Instruction Set Attribute Register 1

The ID_AA64ISAR1_EL1 characteristics are:

**Purpose**

Provides information about the features and instructions implemented in AArch64 state.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers on page D17-5551.

**Configurations**

There are no configuration notes.

**Attributes**

ID_AA64ISAR1_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>LS64</th>
<th>XS</th>
<th>I8MM</th>
<th>DGH</th>
<th>BF16</th>
<th>SPECRES</th>
<th>SB</th>
<th>FRINTTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>60</td>
<td>59</td>
<td>56</td>
<td>55</td>
<td>52</td>
<td>51</td>
<td>48</td>
</tr>
<tr>
<td>47</td>
<td>44</td>
<td>43</td>
<td>40</td>
<td>39</td>
<td>36</td>
<td>35</td>
<td>32</td>
</tr>
<tr>
<td>31</td>
<td>28</td>
<td>27</td>
<td>24</td>
<td>23</td>
<td>20</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

**LS64, bits [63:60]**

Indicates support for LD64B and ST64B* instructions, and the ACCDATA_EL1 register. Defined values of this field are:

- **0b0000** The LD64B, ST64B, ST64BV, and ST64BV0 instructions, the ACCDATA_EL1 register, and associated traps are not supported.
- **0b0001** The LD64B and ST64B instructions are supported.
- **0b0010** The LD64B, ST64B, and ST64BV instructions, and their associated traps are supported.
- **0b0011** The LD64B, ST64B, ST64BV, and ST64BV0 instructions, the ACCDATA_EL1 register, and their associated traps are supported.

All other values are reserved.

FEAT_LS64 implements the functionality identified by 0b0001.

FEAT_LS64_V implements the functionality identified by 0b0010.

FEAT_LS64_ACCDATA implements the functionality identified by 0b0011.

From Armv8.7, the permitted values are 0b0000, 0b0001, 0b0010, and 0b0011.

**XS, bits [59:56]**

Indicates support for the XS attribute, the TLBI and DSB instructions with the nXS qualifier, and the HCRX_EL2.{FGTnXS, FnXS} fields in AArch64 state. Defined values are:

- **0b0000** The XS attribute, the TLBI and DSB instructions with the nXS qualifier, and the HCRX_EL2.{FGTnXS, FnXS} fields are not supported.
- **0b0001** The XS attribute, the TLBI and DSB instructions with the nXS qualifier, and the HCRX_EL2.{FGTnXS, FnXS} fields are supported.

All other values are reserved.

FEAT_XS implements the functionality identified by 0b0001.

From Armv8.7, the only permitted value is 0b0001.
I8MM, bits [55:52]
Indicates support for Advanced SIMD and Floating-point Int8 matrix multiplication instructions in AArch64 state. Defined values are:
- 0b0000  Int8 matrix multiplication instructions are not implemented.
- 0b0001  SMMLA, SUDOT, UMMMLA, USMMLA, and USDOT instructions are implemented.
All other values are reserved.
FEAT_I8MM implements the functionality identified by 0b0001.
When Advanced SIMD and SVE are both implemented, this field must return the same value as ID_AA64ZFR0_EL1.I8MM.
From Armv8.6, the only permitted value is 0b0001.

DGH, bits [51:48]
Indicates support for the Data Gathering Hint instruction. Defined values are:
- 0b0000  Data Gathering Hint is not implemented.
- 0b0001  Data Gathering Hint is implemented.
All other values are reserved.
FEAT_DGH implements the functionality identified by 0b0001.
From Armv8.0, the permitted values are 0b0000 and 0b0001.
If the DGH instruction has no effect in preventing the merging of memory accesses, the value of this field is 0b0000.

BF16, bits [47:44]
Indicates support for Advanced SIMD and Floating-point BFloat16 instructions in AArch64 state. Defined values are:
- 0b0000  BFloat16 instructions are not implemented.
- 0b0001  BFCVT, BFCVTN, BFCVTN2, BFDOT, BFMLALB, BFMLALT, and BFMMMLA instructions are implemented.
- 0b0010  As 0b0001, but the FPCR.EBF field is also supported.
All other values are reserved.
FEAT_BF16 implements the functionality identified by 0b0001.
FEAT_EBF16  implements the functionality identified by 0b0010.
When FEAT_SVE or FEAT_SME are implemented, this field must return the same value as ID_AA64ZFR0_EL1.BF16.
If FEAT_SME is implemented, the permitted values are 0b0001 and 0b0010.
Otherwise, from Armv8.6, the only permitted value is 0b0001.

SPECRES, bits [43:40]
Indicates support for prediction invalidation instructions in AArch64 state. Defined values are:
- 0b0000  CFP RCTX, DVP RCTX, and CPP RCTX instructions are not implemented.
- 0b0001  CFP RCTX, DVP RCTX, and CPP RCTX instructions are implemented.
All other values are reserved.
FEAT_SPECRES implements the functionality identified by 0b0001.
In Armv8.0, the permitted values are 0b0000 and 0b0001.
From Armv8.5, the only permitted value is 0b0001.

SB, bits [39:36]
Indicates support for SB instruction in AArch64 state. Defined values are:
- 0b0000  SB instruction is not implemented.
SB instruction is implemented.
All other values are reserved.

FEAT_SB implements the functionality identified by 0b0001.
In Armv8.0, the permitted values are 0b0000 and 0b0001.
From Armv8.5, the only permitted value is 0b0001.

FRINTTS, bits [35:32]
Indicates support for the FRINT32Z, FRINT32X, FRINT64Z, and FRINT64X instructions are implemented. Defined values are:
- 0b0000: FRINT32Z, FRINT32X, FRINT64Z, and FRINT64X instructions are not implemented.
- 0b0001: FRINT32Z, FRINT32X, FRINT64Z, and FRINT64X instructions are implemented.
All other values are reserved.

FEAT_FRINTTS implements the functionality identified by 0b0001.
From Armv8.5, the only permitted value is 0b0001.

GPI, bits [31:28]
Indicates support for an IMPLEMENTATION DEFINED algorithm is implemented in the PE for generic code authentication in AArch64 state. Defined values are:
- 0b0000: Generic Authentication using an IMPLEMENTATION DEFINED algorithm is not implemented.
- 0b0001: Generic Authentication using an IMPLEMENTATION DEFINED algorithm is implemented. This includes the PACGA instruction.
All other values are reserved.

FEAT_PACIMP implements the functionality identified by 0b0001.
From Armv8.3, the permitted values are 0b0000 and 0b0001.
If the value of ID_AA64ISAR1_EL1.GPA is non-zero, or the value of ID_AA64ISAR2_EL1.GPA3 is non-zero, this field must have the value 0b0000.

GPA, bits [27:24]
Indicates whether the QARMA5 algorithm is implemented in the PE for generic code authentication in AArch64 state. Defined values are:
- 0b0000: Generic Authentication using the QARMA5 algorithm is not implemented.
- 0b0001: Generic Authentication using the QARMA5 algorithm is implemented. This includes the PACGA instruction.
All other values are reserved.

FEAT_PACQARMA5 implements the functionality identified by 0b0001.
From Armv8.3, the permitted values are 0b0000 and 0b0001.
If the value of ID_AA64ISAR1_EL1.GPI is non-zero, or the value of ID_AA64ISAR2_EL1.GPA3 is non-zero, this field must have the value 0b0000.

LRCPC, bits [23:20]
Indicates support for weaker release consistency, RCpc, based model. Defined values are:
- 0b0000: The LDAPR*, LDAPUR*, and STLUR* instructions are not implemented.
- 0b0001: The LDAPR* instructions are implemented.
The LDAPUR*, and STLUR* instructions are not implemented.
- 0b0010: The LDAPR*, LDAPUR*, and STLUR* instructions are implemented.
All other values are reserved.

FEAT_LRCPC implements the functionality identified by the value 0b0001.
FEAT_LRCPC2 implements the functionality identified by the value 0b0010.
In Armv8.2, the permitted values are 0b0000, 0b0001, and 0b0010.
In Armv8.3, the permitted values are 0b0001 and 0b0010.
From Armv8.4, the only permitted value is 0b0010.

**FCMA, bits [19:16]**

Indicates support for complex number addition and multiplication, where numbers are stored in vectors. Defined values are:
- 0b0000: The FCMLA and FCADD instructions are not implemented.
- 0b0001: The FCMLA and FCADD instructions are implemented.

All other values are reserved.

FEAT_FCMA implements the functionality identified by the value 0b0001.
In Armv8.0, Armv8.1, and Armv8.2, the only permitted value is 0b0000.
From Armv8.3, if Advanced SIMD or Floating-point is implemented, the only permitted value is 0b0001.
From Armv8.3, if Advanced SIMD or Floating-point is not implemented, the only permitted value is 0b0000.

**JSCVT, bits [15:12]**

Indicates support for JavaScript conversion from double precision floating point values to integers in AArch64 state. Defined values are:
- 0b0000: The FJCVTZS instruction is not implemented.
- 0b0001: The FJCVTZS instruction is implemented.

All other values are reserved.

FEAT_JSCVT implements the functionality identified by 0b0001.
In Armv8.0, Armv8.1, and Armv8.2, the only permitted value is 0b0000.
From Armv8.3, if Advanced SIMD or Floating-point is implemented, the only permitted value is 0b0001.
From Armv8.3, if Advanced SIMD or Floating-point is not implemented, the only permitted value is 0b0000.

**API, bits [11:8]**

Indicates whether an IMPLEMENTATION DEFINED algorithm is implemented in the PE for address authentication, in AArch64 state. This applies to all Pointer Authentication instructions other than the PACGA instruction. Defined values are:
- 0b0000: Address Authentication using an IMPLEMENTATION DEFINED algorithm is not implemented.
- 0b0001: Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC() and HaveEnhancedPAC2() functions returning FALSE.
- 0b0010: Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC() function returning TRUE, and the HaveEnhancedPAC2() function returning FALSE.
- 0b0011: Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, and the HaveFPAC() function returning FALSE.
- 0b0100: Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning FALSE, and the HaveEnhancedPAC() function returning FALSE.
0b0101 Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning TRUE, and the HaveEnhancedPAC() function returning FALSE.

All other values are reserved.

FEAT_PAuth implements the functionality identified by 0b0001.
FEAT_EPAC implements the functionality identified by 0b0010.
FEAT_PAuth2 implements the functionality identified by 0b0011.
FEAT_FPAC implements the functionality identified by 0b0100.
FEAT_FPACCOMBINE implements the functionality identified by 0b0101.

When this field is non-zero, FEAT_PACIMP is implemented.

In Armv8.3, the permitted values are 0b0001, 0b0010, 0b0011, 0b0100, and 0b0101.
From Armv8.6, the permitted values are 0b0011, 0b0100, and 0b0101.

If the value of ID_AA64ISAR1_EL1.APA is non-zero, or the value of ID_AA64ISAR2_EL1.APA3 is non-zero, this field must have the value 0b0000.

APA, bits [7:4]

Indicates whether the QARMA5 algorithm is implemented in the PE for address authentication, in AArch64 state. This applies to all Pointer Authentication instructions other than the PACGA instruction. Defined values are:

0b0000 Address Authentication using the QARMA5 algorithm is not implemented.
0b0001 Address Authentication using the QARMA5 algorithm is implemented, with the HaveEnhancedPAC() and HaveEnhancedPAC2() functions returning FALSE.
0b0010 Address Authentication using the QARMA5 algorithm is implemented, with the HaveEnhancedPAC() function returning TRUE and the HaveEnhancedPAC2() function returning FALSE.
0b0011 Address Authentication using the QARMA5 algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning FALSE, the HaveFPACCombined() function returning FALSE, and the HaveEnhancedPAC() function returning FALSE.
0b0100 Address Authentication using the QARMA5 algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning FALSE, and the HaveEnhancedPAC() function returning FALSE.
0b0101 Address Authentication using the QARMA5 algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning TRUE, and the HaveEnhancedPAC() function returning FALSE.

All other values are reserved.

FEAT_PAuth implements the functionality identified by 0b0001.
FEAT_EPAC implements the functionality identified by 0b0010.
FEAT_PAuth2 implements the functionality identified by 0b0011.
FEAT_FPAC implements the functionality identified by 0b0100.
FEAT_FPACCOMBINE implements the functionality identified by 0b0101.

When this field is non-zero, FEAT_PACQARMA5 is implemented.

In Armv8.3, the permitted values are 0b0001, 0b0010, 0b0011, 0b0100, and 0b0101.
From Armv8.6, the permitted values are 0b0011, 0b0100, and 0b0101.

If the value of ID_AA64ISAR1_EL1.API is non-zero, or the value of ID_AA64ISAR2_EL1.APA3 is non-zero, this field must have the value 0b0000.
DPB, bits [3:0]

Data Persistence writeback. Indicates support for the DC CVAP and DC CVADP instructions in AArch64 state. Defined values are:

- 0b0000   DC CVAP not supported.
- 0b0001   DC CVAP supported.
- 0b0010   DC CVAP and DC CVADP supported.

All other values are reserved.

FEAT_DPB implements the functionality identified by the value 0b0001.

FEAT_DPB2 implements the functionality identified by the value 0b0010.

In Armv8.2, the permitted values are 0b0001 and 0b0010.
From Armv8.5, the only permitted value is 0b0010.

Accessing ID_AA64ISAR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[
MRS \langle Xt \rangle, ID_{AA64ISAR1\_EL1}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0110</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```
if PSTATE_EL == EL0 then
   if IsFeatureImplemented(FEAT_IDST) then
      if EL2Enabled() && HCR_EL2.TGE == '1' then
         AArch64.SystemAccessTrap(EL2, 0x18);
      else
         AArch64.SystemAccessTrap(EL1, 0x18);
      else
         UNDEFINED;
   else
      X[t, 64] = ID_AA64ISAR1_EL1;
else
   X[t, 64] = ID_AA64ISAR1_EL1;
```
D17.2.63 ID_AA64ISAR2_EL1, AArch64 Instruction Set Attribute Register 2

The ID_AA64ISAR2_EL1 characteristics are:

**Purpose**

Provides information about the features and instructions implemented in AArch64 state.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers on page D17-5551.

**Configurations**

--- Note ---

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

**Attributes**

ID_AA64ISAR2_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-28</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>27-24</td>
<td>PAC_frac</td>
<td></td>
</tr>
<tr>
<td>23-20</td>
<td>BC</td>
<td></td>
</tr>
<tr>
<td>19-16</td>
<td>MOPS</td>
<td></td>
</tr>
<tr>
<td>15-12</td>
<td>APA3</td>
<td></td>
</tr>
<tr>
<td>11-8</td>
<td>GPA3</td>
<td></td>
</tr>
<tr>
<td>7-4</td>
<td>RPRES</td>
<td></td>
</tr>
<tr>
<td>3-0</td>
<td>WFXT</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:28]**

Reserved, RES0.

**PAC_frac, bits [27:24]**

Indicates whether the ConstPACField() function used as part of the PAC addition returns FALSE or TRUE.

0b0000  ConstPACField() returns FALSE.
0b0001  ConstPACField() returns TRUE.

All other values are reserved.

FEAT_CONSTPACFIELD implements the functionality identified by 0b0001.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

**BC, bits [23:20]**

Indicates support for the BC instruction in AArch64 state. Defined values are:

0b0000  BC instruction is not implemented.
0b0001  BC instruction is implemented.

All other values are reserved.

FEAT_HBC implements the functionality identified by the value 0b0001.

From Armv8.8, the only permitted value is 0b0001.

**MOPS, bits [19:16]**

Indicates support for the Memory Copy and Memory Set instructions in AArch64 state.

0b0000  The Memory Copy and Memory Set instructions are not implemented in AArch64 state.
0b0001 The Memory Copy and Memory Set instructions are implemented in AArch64 state with the following exception. If FEAT_MTE is implemented, then SETGP*, SETGM* and SETGE* instructions are also supported.

All other values are reserved.

FEAT_MOPS implements the functionality identified by the value 0b0001.

From Armv8.8, the only permitted value is 0b0001.

APA3, bits [15:12]

Indicates whether the QARMA3 algorithm is implemented in the PE for address authentication in AArch64 state. This applies to all Pointer Authentication instructions other than the PACGA instruction. Defined values are:

- 0b0000 Address Authentication using the QARMA3 algorithm is not implemented.
- 0b0001 Address Authentication using the QARMA3 algorithm is implemented, with the HaveEnhancedPAC() and HaveEnhancedPAC2() functions returning FALSE.
- 0b0010 Address Authentication using the QARMA3 algorithm is implemented, with the HaveEnhancedPAC() function returning TRUE and the HaveEnhancedPAC2() function returning FALSE.
- 0b0011 Address Authentication using the QARMA3 algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning FALSE, the HaveFPACCombined() function returning FALSE, and the HaveEnhancedPAC() function returning FALSE.
- 0b0100 Address Authentication using the QARMA3 algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning FALSE, and the HaveEnhancedPAC() function returning FALSE.
- 0b0101 Address Authentication using the QARMA3 algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning TRUE, and the HaveEnhancedPAC() function returning FALSE.

All other values are reserved.

FEAT_PAuth implements the functionality identified by 0b0001.

FEAT_EPAC implements the functionality identified by 0b0010.

FEAT_PAuth2 implements the functionality identified by 0b0011.

FEAT_FPAC implements the functionality identified by 0b0100.

FEAT_FPACCOMBINE implements the functionality identified by 0b0101.

When this field is non-zero, FEAT_PACQARMA3 is implemented.

In Armv8.3, the permitted values are 0b0000, 0b0001, 0b0010, 0b0011, 0b0100, and 0b0101.

From Armv8.6, the permitted values are 0b0011, 0b0100, and 0b0101.

If the value of ID_AA64ISAR1_EL1.API is non-zero, or the value of ID_AA64ISAR1_EL1.APA is non-zero, this field must have the value 0b0000.

GPA3, bits [11:8]

Indicates whether the QARMA3 algorithm is implemented in the PE for generic code authentication in AArch64 state. Defined values are:

- 0b0000 Generic Authentication using the QARMA3 algorithm is not implemented.
- 0b0001 Generic Authentication using the QARMA3 algorithm is implemented. This includes the PACGA instruction.

All other values are reserved.

FEAT_PACQARMA3 implements the functionality identified by 0b0001.

From Armv8.3, the permitted values are 0b0000 and 0b0001.
If the value of `ID_AA64ISAR1_EL1.GPI` is non-zero, or the value of `ID_AA64ISAR1_EL1.GPA` is non-zero, this field must have the value `0b0000`.

**RPRES, bits [7:4]**

Indicates support for 12 bits of mantissa in reciprocal and reciprocal square root instructions in AArch64 state, when `FPCR.AH` is 1. Defined values are:

- **0b0000** When `FPCR.AH == 1`:
  Reciprocal and reciprocal square root estimates give 8 bits of mantissa, when `FPCR.AH` is 1.
- **0b0001** When `FPCR.AH == 1`:
  Reciprocal and reciprocal square root estimates give 12 bits of mantissa, when `FPCR.AH` is 1.

All other values are reserved.

**FEAT_RPRES** implements the functionality identified by the value `0b0001`.

From Armv8.7, if Advanced SIMD and floating-point is implemented, the only permitted value is `0b0001`.

**WFxT, bits [3:0]**

Indicates support for the `WFET` and `WFIT` instructions in AArch64 state. Defined values are:

- **0b0000** `WFET` and `WFIT` are not supported.
- **0b0010** `WFET` and `WFIT` are supported, and the register number is reported in the ESR_ELx on exceptions.

All other values are reserved.

**FEAT_WFxT** implements the functionality identified by the value `0b0010`.

From Armv8.7, the only permitted value is `0b0010`.

### Accessing `ID_AA64ISAR2_EL1`

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ID_AA64ISAR2_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0110</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(�FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    else
      UNDEFINED;
  elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (IsFeatureImplemented(�FEAT_FGT) || !IsZero(ID_AA64ISAR2_EL1) || boolean IMPLEMENTATION_DEFINED "ID_AA64ISAR2_EL1 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      X[t, 64] = ID_AA64ISAR2_EL1;
    elsif PSTATE.EL == EL2 then
      X[t, 64] = ID_AA64ISAR2_EL1;
    elsif PSTATE.EL == EL3 then
      X[t, 64] = ID_AA64ISAR2_EL1;
```
D17.2.64 ID_AA64MMFR0_EL1, AArch64 Memory Model Feature Register 0

The ID_AA64MMFR0_EL1 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

There are no configuration notes.

**Attributes**

ID_AA64MMFR0_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECV</td>
<td>[63:60]</td>
</tr>
<tr>
<td>FGT</td>
<td>[59:56]</td>
</tr>
<tr>
<td>RES0</td>
<td>[55:48]</td>
</tr>
<tr>
<td>ExS</td>
<td>[47:44]</td>
</tr>
<tr>
<td>TGran4_2</td>
<td>[43:40]</td>
</tr>
<tr>
<td>TGran64_2</td>
<td>[39:36]</td>
</tr>
<tr>
<td>TGran16_2</td>
<td>[35:32]</td>
</tr>
<tr>
<td>TGran4</td>
<td>[31:28]</td>
</tr>
<tr>
<td>TGran64</td>
<td>[27:24]</td>
</tr>
<tr>
<td>TGran16</td>
<td>[23:20]</td>
</tr>
<tr>
<td>SNSMem</td>
<td>[19:16]</td>
</tr>
<tr>
<td>BigEnd</td>
<td>[15:12]</td>
</tr>
<tr>
<td>ASIDBits</td>
<td>[11:8]</td>
</tr>
<tr>
<td>PARange</td>
<td>[7:4]</td>
</tr>
<tr>
<td>BigEndEL0</td>
<td>[3:0]</td>
</tr>
</tbody>
</table>

**ECV, bits [63:60]**

Indicates presence of Enhanced Counter Virtualization. Defined values are:

- **0b0000** Enhanced Counter Virtualization is not implemented.
- **0b0001** Enhanced Counter Virtualization is implemented. Supports CNTHCTL_EL2, {EL1ITVT, EL1TVCT, EL1NVPCT, EL1NVCVT, EVNTIS, CNTKCTL_EL1.EVNTIS, CNTPCTSS_EL0} counter views, and CNTVCTSS_EL0 counter views. Extends the PMSCR_EL1.PCT, PMSCR_EL2.PCT, TRFCR_EL1.TS, and TRFCR_EL2.TS fields.
- **0b0010** As 0b0001, and also includes support for CNTHCTL_EL2.ECV and CNTPOFF_EL2. All other values are reserved.

**FGT, bits [59:56]**

Indicates presence of the Fine-Grained Trap controls. Defined values are:

- **0b0000** Fine-grained trap controls are not implemented.
- **0b0001** Fine-grained trap controls are implemented. Supports:
  - If EL2 is implemented, the HAOFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGTR_EL2, HFGTR_EL2, and HFGWTR_EL2 registers, and their associated traps.
  - If EL2 is implemented, MDCR_EL2.TDCC.
  - If EL3 is implemented, MDCR_EL3.TDCC.
  - If both EL2 and EL3 are implemented, SCR_EL3.FGTEn.

All other values are reserved.

**FEAT_ECV** implements the functionality identified by the values 0b0001 and 0b0010.

From Armv8.6, the only permitted values are 0b0001 and 0b0010.
From Armv8.6, the value 0b0000 is not permitted.

**Bits [55:48]**

Reserved, RES0.

**ExS, bits [47:44]**

Indicates support for disabling context synchronizing exception entry and exit. Defined values are:

- 0b0000: All exception entries and exits are context synchronization events.
- 0b0001: Non-context synchronizing exception entry and exit are supported.

All other values are reserved.

FEAT_ExS implements the functionality identified by the value 0b0001.

**TGran4_2, bits [43:40]**

Indicates support for 4KB memory granule size at stage 2. Defined values are:

- 0b0000: Support for 4KB granule at stage 2 is identified in the ID_AA64MMFR0_EL1.TGran4 field.
- 0b0001: 4KB granule not supported at stage 2.
- 0b0010: 4KB granule supported at stage 2.
- 0b0011: When FEAT_LPA2 is implemented:
  4KB granule at stage 2 supports 52-bit input and output addresses.

All other values are reserved.

The 0b0000 value is deprecated.

--- Note ---

This field does not follow the standard ID scheme. See *Alternative ID scheme used for ID_AA64MMFR0_EL1 stage 2 granule sizes on page D17-5554* for more information.

**TGran64_2, bits [39:36]**

Indicates support for 64KB memory granule size at stage 2. Defined values are:

- 0b0000: Support for 64KB granule at stage 2 is identified in the ID_AA64MMFR0_EL1.TGran64 field.
- 0b0001: 64KB granule not supported at stage 2.
- 0b0010: 64KB granule supported at stage 2.

All other values are reserved.

The 0b0000 value is deprecated.

--- Note ---

This field does not follow the standard ID scheme. See *Alternative ID scheme used for ID_AA64MMFR0_EL1 stage 2 granule sizes on page D17-5554* for more information.

**TGran16_2, bits [35:32]**

Indicates support for 16KB memory granule size at stage 2. Defined values are:

- 0b0000: Support for 16KB granule at stage 2 is identified in the ID_AA64MMFR0_EL1.TGran16 field.
- 0b0001: 16KB granule not supported at stage 2.
- 0b0010: 16KB granule supported at stage 2.
- 0b0011: When FEAT_LPA2 is implemented:
  16KB granule at stage 2 supports 52-bit input and output addresses.

All other values are reserved.
The 0b0000 value is deprecated.

--- Note ---

This field does not follow the standard ID scheme. See Alternative ID scheme used for ID_AA64MMFR0_EL1 stage 2 granule sizes on page D17-5554 for more information.

**TGran4, bits [31:28]**
Indicates support for 4KB memory translation granule size. Defined values are:

- 0b0000 4KB granule supported.
- 0b0001 *When FEAT_LPA2 is implemented:* 4KB granule supports 52-bit input and output addresses.
- 0b1111 4KB granule not supported.

All other values are reserved.

**TGran64, bits [27:24]**
Indicates support for 64KB memory translation granule size. Defined values are:

- 0b0000 64KB granule supported.
- 0b1111 64KB granule not supported.

All other values are reserved.

**TGran16, bits [23:20]**
Indicates support for 16KB memory translation granule size. Defined values are:

- 0b0000 16KB granule not supported.
- 0b0001 16KB granule supported.
- 0b0010 *When FEAT_LPA2 is implemented:* 16KB granule supports 52-bit input and output addresses.

All other values are reserved.

**BigEndEL0, bits [19:16]**
Indicates support for mixed-endian at EL0 only. Defined values are:

- 0b0000 No mixed-endian support at EL0. The SCTLR_EL1.E0E bit has a fixed value.
- 0b0001 Mixed-endian support at EL0. The SCTLR_EL1.E0E bit can be configured.

All other values are reserved.

This field is invalid and is RES0 if ID_AA64MMFR0_EL1.BigEnd is not 0b0000.

**SNSMem, bits [15:12]**
Indicates support for a distinction between Secure and Non-secure Memory. Defined values are:

- 0b0000 Does not support a distinction between Secure and Non-secure Memory.
- 0b0001 Does support a distinction between Secure and Non-secure Memory.

--- Note ---

If EL3 is implemented, the value 0b0000 is not permitted.

All other values are reserved.

**BigEnd, bits [11:8]**
Indicates support for mixed-endian configuration. Defined values are:

- 0b0000 No mixed-endian support. The SCTLR_ELx.EE bits have a fixed value. See the BigEndEL0 field, bits[19:16], for whether EL0 supports mixed-endian.
0b0001  Mixed-endian support. The SCTLR_ELx.EE and SCTLR_EL1.E0E bits can be configured.

All other values are reserved.

**ASIDBits, bits [7:4]**

Number of ASID bits. Defined values are:

- 0b0000  8 bits.
- 0b0010  16 bits.

All other values are reserved.

**PARange, bits [3:0]**

Physical Address range supported. Defined values are:

- 0b0000  32 bits, 4GB.
- 0b0001  36 bits, 64GB.
- 0b0010  40 bits, 1TB.
- 0b0011  42 bits, 4TB.
- 0b0100  44 bits, 16TB.
- 0b0101  48 bits, 256TB.
- 0b0110  52 bits, 4PB.

All other values are reserved.

The value 0b0110 is permitted only if the implementation includes FEAT_LPA, otherwise it is reserved.

**Accessing ID_AA64MMFR0_EL1**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS}<Xt>, \text{ID-AA64MMFR0_EL1}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0111</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.TID3 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = ID_AA64MMFR0_EL1;
elsif PSTATE_EL == EL2 then
  X[t, 64] = ID_AA64MMFR0_EL1;
elsif PSTATE_EL == EL3 then
  X[t, 64] = ID_AA64MMFR0_EL1;
### D17.2.65 ID_AA64MMFR1_EL1, AArch64 Memory Model Feature Register 1

The ID_AA64MMFR1_EL1 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch64 state.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers on page D17-5551.

**Configurations**

There are no configuration notes.

**Attributes**

ID_AA64MMFR1_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-60</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>59-56</td>
<td>CMOW</td>
</tr>
<tr>
<td>55-52</td>
<td>TIDCP1</td>
</tr>
<tr>
<td>51-48</td>
<td>nTLBPA</td>
</tr>
<tr>
<td>47-44</td>
<td>AFP</td>
</tr>
<tr>
<td>43-40</td>
<td>HCX</td>
</tr>
<tr>
<td>39-36</td>
<td>ETS</td>
</tr>
<tr>
<td>35-32</td>
<td>TWED</td>
</tr>
<tr>
<td>31-28</td>
<td>XNX</td>
</tr>
<tr>
<td>27-24</td>
<td>SpecSEI</td>
</tr>
<tr>
<td>23-20</td>
<td>PAN</td>
</tr>
<tr>
<td>19-16</td>
<td>LO</td>
</tr>
<tr>
<td>15-12</td>
<td>HPDS</td>
</tr>
<tr>
<td>11-8</td>
<td>VH</td>
</tr>
<tr>
<td>7-4</td>
<td>VMIDBits</td>
</tr>
<tr>
<td>3-0</td>
<td>HAFDBS</td>
</tr>
</tbody>
</table>

**Bits [63:60]**

Reserved, RES0.

**CMOW, bits [59:56]**

Indicates support for cache maintenance instruction permission. Defined values are:

- **0b0000** SCTLRL1.CMOW, SCTLRL2.CMOW, and HCRX_EL2.CMOW bits are not implemented.
- **0b0001** SCTLRL1.CMOW is implemented. If EL2 is implemented, SCTLRL2.CMOW and HCRX_EL2.CMOW bits are implemented.

All other values are reserved.

FEAT_CMOW implements the functionality identified by the value 0b0001.

From Armv8.8, the only permitted value is 0b0001.

**TIDCP1, bits [55:52]**

Indicates whether SCTLRL1.TIDCP and SCTLRL2.TIDCP are implemented in AArch64 state. Defined values are:

- **0b0000** SCTLRL1.TIDCP and SCTLRL2.TIDCP bits are not implemented and are RES0.
- **0b0001** SCTLRL1.TIDCP bit is implemented. If EL2 is implemented, SCTLRL2.TIDCP bit is implemented.

All other values are reserved.

FEAT_TIDCP1 implements the functionality identified by the value 0b0001.

From Armv8.8, the only permitted value is 0b0001.

**nTLBPA, bits [51:48]**

Indicates support for intermediate caching of translation table walks. Defined values are:

- **0b0000** The intermediate caching of translation table walks might include non-coherent physical translation caches.
The intermediate caching of translation table walks does not include non-coherent physical translation caches.

Non-coherent physical translation caches are non-coherent caches of previous valid translation table entries since the last completed relevant TLBI applicable to the PE, where either:

- The caching is indexed by the physical address of the location holding the translation table entry.
- The caching is used for stage 1 translations and is indexed by the intermediate physical address of the location holding the translation table entry.

All other values are reserved.

FEAT_nTLBPA implements the functionality identified by the value 0b0001.

From Armv8.0, the permitted values are 0b0000 and 0b0001.

### AFP, bits [47:44]
Indicates support for FPCR.{AH, FIZ, NEP}. Defined values are:

- 0b0000 The FPCR.{AH, FIZ, NEP} fields are not supported.
- 0b0001 The FPCR.{AH, FIZ, NEP} fields are supported.

All other values are reserved.

FEAT_AFP implements the functionality identified by the value 0b0001.

From Armv8.7, if Advanced SIMD and floating-point is implemented, the only permitted value is 0b0001.

### HCX, bits [43:40]
Indicates support for HCRX_EL2 and its associated EL3 trap. Defined values are:

- 0b0000 HCRX_EL2 and its associated EL3 trap are not supported.
- 0b0001 HCRX_EL2 and its associated EL3 trap are supported.

All other values are reserved.

FEAT_HCX implements the functionality identified by the value 0b0001.

From Armv8.7, if EL2 is implemented, the only permitted value is 0b0001.

### ETS, bits [39:36]
Indicates support for Enhanced Translation Synchronization. Defined values are:

- 0b0000 Enhanced Translation Synchronization is not supported.
- 0b0001 Enhanced Translation Synchronization is supported.

All other values are reserved.

FEAT_ETS implements the functionality identified by the value 0b0001.

In Armv8.0, the permitted values are 0b0000 and 0b0001.

From Armv8.7, the only permitted value is 0b0001.

### TWED, bits [35:32]
Indicates support for the configurable delayed trapping of WFE. Defined values are:

- 0b0000 Configurable delayed trapping of WFE is not supported.
- 0b0001 Configurable delayed trapping of WFE is supported.

All other values are reserved.

FEAT_TWED implements the functionality identified by the value 0b0001.

From Armv8.6, the permitted values are 0b0000 and 0b0001.
XNX, bits [31:28]
Indicates support for execute-never control distinction by Exception level at stage 2. Defined values are:
0b0000 Distinction between EL0 and EL1 execute-never control at stage 2 not supported.
0b0001 Distinction between EL0 and EL1 execute-never control at stage 2 supported.
All other values are reserved.
FEAT_XNX implements the functionality identified by the value 0b0001.
From Armv8.2, the only permitted value is 0b0001.

SpecSEI, bits [27:24]
When FEAT_RAS is implemented:
Describes whether the PE can generate SError interrupt exceptions from speculative reads of memory, including speculative instruction fetches.
0b0000 The PE never generates an SError interrupt due to an External abort on a speculative read.
0b0001 The PE might generate an SError interrupt due to an External abort on a speculative read.
All other values are reserved.
Otherwise:
Reserved, RES0.

PAN, bits [23:20]
Privileged Access Never. Indicates support for the PAN bit in PSTATE, SPSR_EL1, SPSR_EL2, SPSR_EL3, and DSPSR_EL0. Defined values are:
0b0000 PAN not supported.
0b0001 PAN supported.
0b0010 PAN supported and AT S1E1RP and AT S1E1WP instructions supported.
0b0011 PAN supported, AT S1E1RP and AT S1E1WP instructions supported, and SCTLR_EL1.EPAN and SCTLR_EL2.EPAN bits supported.
All other values are reserved.
FEAT_PAN implements the functionality identified by the value 0b0001.
FEAT_PAN2 implements the functionality added by the value 0b0010.
FEAT_PAN3 implements the functionality added by the value 0b0011.
In Armv8.1, the permitted values are 0b0001, 0b0010, and 0b0011.
From Armv8.2, the permitted values are 0b0010 and 0b0011.
From Armv8.7, the only permitted value is 0b0011.

LO, bits [19:16]
LORegions. Indicates support for LORegions. Defined values are:
0b0000 LORegions not supported.
0b0001 LORegions supported.
All other values are reserved.
FEAT_LOR implements the functionality identified by the value 0b0001.
From Armv8.1, the only permitted value is 0b0001.

HPDS, bits [15:12]
Hierarchical Permission Disables. Indicates support for disabling hierarchical controls in translation tables. Defined values are:
0b0000 Disabling of hierarchical controls not supported.
Disabling of hierarchical controls supported with the TCR_EL1.{HPD1, HPD0}, TCR_EL2.HPD or TCR_EL2.{HPD1, HPD0}, and TCR_EL3.HPD bits.

As for value 0b0001, and adds possible hardware allocation of bits[62:59] of the Translation table descriptors from the final lookup level for IMPLEMENTATION DEFINED use.

All other values are reserved.

**FEAT_HPDS** implements the functionality identified by the value 0b0001.

**FEAT_HPDS2** implements the functionality identified by the value 0b0010.

From Armv8.1, the value 0b0000 is not permitted.

**VH**, bits [11:8]

Virtualization Host Extensions. Defined values are:

- 0b0000 Virtualization Host Extensions not supported.
- 0b0001 Virtualization Host Extensions supported.

All other values are reserved.

**FEAT_VHE** implements the functionality identified by the value 0b0001.

From Armv8.1, the only permitted value is 0b0001.

**VomidBits**, bits [7:4]

Number of VMID bits. Defined values are:

- 0b0000 8 bits
- 0b0010 16 bits

All other values are reserved.

**FEAT_VMID16** implements the functionality identified by the value 0b0010.

From Armv8.1, the permitted values are 0b0000 and 0b0010.

**HAFDBS**, bits [3:0]

Hardware updates to Access flag and Dirty state in translation tables. Defined values are:

- 0b0000 Hardware update of the Access flag and dirty state are not supported.
- 0b0001 Hardware update of the Access flag is supported.
- 0b0010 Hardware update of both the Access flag and dirty state is supported.

All other values are reserved.

**FEAT_HAFDBS** implements the functionality identified by the values 0b0001 and 0b0010.

From Armv8.1, the permitted values are 0b0000, 0b0001, and 0b0010.

**Accessing ID_AA64MMFR1_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ID_AA64MMFR1_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b0111</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
  else
AArch64.SystemAccessTrap(El1, 0x18); 
else
  UNDEFINED;
elsif PSTATE.EL == El1 then
  if EL2Enabled() && HCR_EL2.TID3 == '1' then
    AArch64.SystemAccessTrap(El2, 0x18);
  else
    X[t, 64] = ID_AA64MMFR1_EL1;
  endif
elsif PSTATE.EL == El2 then
  X[t, 64] = ID_AA64MMFR1_EL1;
elsif PSTATE.EL == El3 then
  X[t, 64] = ID_AA64MMFR1_EL1;
D17.2.66 ID_AA64MMFR2_EL1, AArch64 Memory Model Feature Register 2

The ID_AA64MMFR2_EL1 characteristics are:

### Purpose

Provides information about the implemented memory model and memory management support in AArch64 state.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers on page D17-5551.

### Configurations

**Note**

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

### Attributes

ID_AA64MMFR2_EL1 is a 64-bit register.

### Field descriptions

<table>
<thead>
<tr>
<th>E0PD</th>
<th>EVT</th>
<th>BBM</th>
<th>TTL</th>
<th>RES0</th>
<th>FWB</th>
<th>IDS</th>
<th>AT</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>60</td>
<td>59</td>
<td>56</td>
<td>55</td>
<td>52</td>
<td>48</td>
<td>47</td>
</tr>
<tr>
<td>44</td>
<td>43</td>
<td>40</td>
<td>39</td>
<td>36</td>
<td>35</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>23</td>
<td>20</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td>7</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV</td>
<td>CCIDX</td>
<td>VARange</td>
<td>IESB</td>
<td>LSM</td>
<td>UAO</td>
<td>CnP</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**E0PD**  
Indicates support for the E0PD mechanism. Defined values are:

- 0b0000  E0PDx mechanism is not implemented.
- 0b0001  E0PDx mechanism is implemented.

All other values are reserved.

**FEAT_E0PD** implements the functionality identified by the value 0b0001.

In Armv8.4, the permitted values are 0b0000 and 0b0001.

From Armv8.5, the only permitted value is 0b0001.

If FEAT_E0PD is implemented, FEAT_CSV3 must be implemented.

**EVT**  
Enhanced Virtualization Traps. If EL2 is implemented, indicates support for the HCR_EL2.{TTLBOS, TTLBIS, TOCU, TICAB, TID4} traps. Defined values are:

- 0b0000  HCR_EL2.{TTLBOS, TTLBIS, TOCU, TICAB, TID4} traps are not supported.
- 0b0001  HCR_EL2.{TOCU, TICAB, TID4} traps are supported. HCR_EL2.{TTLBOS, TTLBIS} traps are not supported.
- 0b0010  HCR_EL2.{TTLBOS, TTLBIS, TOCU, TICAB, TID4} traps are supported.

All other values are reserved.

**FEAT_EVT** implements the functionality identified by the values 0b0001 and 0b0010.

If EL2 is not implemented, the only permitted value is 0b0000.

In Armv8.2, the permitted values are 0b0000, 0b0001, and 0b0010.

From Armv8.5, the permitted values are:

- 0b0000 when EL2 is not implemented.
• 0b0010 when EL2 is implemented.

**BBM, bits [55:52]**

Allows identification of the requirements of the hardware to have break-before-make sequences when changing block size for a translation.

- 0b0000  Level 0 support for changing block size is supported.
- 0b0001  Level 1 support for changing block size is supported.
- 0b0010  Level 2 support for changing block size is supported.

All other values are reserved.

FEAT_BBM implements the functionality identified by the values 0b0000, 0b0001, and 0b0010.

From Armv8.4, the permitted values are 0b0000, 0b0001, and 0b0010.

**TTL, bits [51:48]**

Indicates support for TTL field in address operations. Defined values are:

- 0b0000  TLB maintenance instructions by address have bits[47:44] as RES0.
- 0b0001  TLB maintenance instructions by address have bits[47:44] holding the TTL field.

All other values are reserved.

FEAT_TTL implements the functionality identified by the value 0b0001.

This field affects TLBI IPAS2E1, TLBI IPAS2E1NXS, TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS, TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS, TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS, TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS, TLBI VAAEI, TLBI VAAEI1XS, TLBI VAAEI1IS, TLBI VAAEI1ISNXS, TLBI VAAEI1OS, TLBI VAAEI1OSNXS, TLBI VAAEI0S, TLBI VAAEI0SXS, TLBI VAAEI0SNXS, TLBI VAAEI0SNXSNXS, TLBI VAAE1, TLBI VAAE1NXS, TLBI VAAE1IS, TLBI VAAE1ISNXS, TLBI VAAE1OS, TLBI VAAE1OSNXS, TLBI VAAE1OSNXS, TLBI VAAE0S, TLBI VAAE0SNXS, TLBI VAAE0SNXSNXS, TLBI VAAE0S, TLBI VAAE0SNXS, TLBI VAAE0SNXSNXS, TLBI VAAE2, TLBI VAAE2NXS, TLBI VAAE2IS, TLBI VAAE2ISNXS, TLBI VAAE2OS, TLBI VAAE2OSNXS, TLBI VAAE2OSNXS, TLBI VAE1, TLBI VAE1NXS, TLBI VAE1IS, TLBI VAE1ISNXS, TLBI VAE1OS, TLBI VAE1OSNXS, TLBI VAE1OSNXS, TLBI VAE0S, TLBI VAE0SNXS, TLBI VAE0SNXSNXS, TLBI VAE0S, TLBI VAE0SNXS, TLBI VAE0SNXSNXS, TLBI VAE1, TLBI VAE1IS, TLBI VAE1ISNXS, TLBI VAE1OS, TLBI VAE1OSNXS, TLBI VAE2, TLBI VAE2NXS, TLBI VAE2IS, TLBI VAE2ISNXS, TLBI VAE2OS, TLBI VAE2OSNXS, TLBI VAE2OSNXS, TLBI VAE3, TLBI VAE3NXS, TLBI VAE3IS, TLBI VAE3ISNXS, TLBI VAE3OS, TLBI VAE3OSNXS, TLBI VAEC1, TLBI VAEC1NXS, TLBI VAEC1IS, TLBI VAEC1ISNXS, TLBI VAEC1OS, TLBI VAEC1OSNXS, TLBI VAEC2, TLBI VAEC2NXS, TLBI VAEC2IS, TLBI VAEC2ISNXS, TLBI VAEC2OS, TLBI VAEC2OSNXS, TLBI VAEC2OSNXS, TLBI VAEC3, TLBI VAEC3NXS, TLBI VAEC3IS, TLBI VAEC3ISNXS, TLBI VAEC3OS, TLBI VAEC3OSNXS, TLBI VAEC3OS, TLBI VAEC3OSNXS.

From Armv8.4, the only permitted value is 0b0001.

**Bits [47:44]**

Reserved, RES0.

**FWB, bits [43:40]**

Indicates support for HCR_EL2.FWB. Defined values are:

- 0b0000  HCR_EL2.FWB bit is not supported.
- 0b0001  HCR_EL2.FWB is supported.

All other values reserved.

FEAT_S2FWB implements the functionality identified by the value 0b0001.

From Armv8.4, the only permitted value is 0b0001.

**IDS, bits [39:36]**

Indicates the value of ESR_ELx.EC that reports an exception generated by a read access to the feature ID space. Defined values are:

- 0b0000  An exception which is generated by a read access to the feature ID space, other than a trap caused by HCR_EL2.TIDx, SCTLR_EL1.UCT, or SCTLR_EL2.UCT, is reported by ESR_ELx.EC == 0x0.
- 0b0001  All exceptions generated by an AArch64 read access to the feature ID space are reported by ESR_ELx.EC == 0x18.
All other values are reserved.
The Feature ID space is defined as the System register space in AArch64 with op0==3, op1==\{0, 1, 3\}, CRn==0, CRm==\{0-7\}, op2==\{0-7\}.

**FEAT_IDST** implements the functionality identified by the value \(0b0001\).

From Armv8.4, the only permitted value is \(0b0001\).

**AT, bits [35:32]**

Identifies support for unaligned single-copy atomicity and atomic functions. Defined values are:
- \(0b0000\) Unaligned single-copy atomicity and atomic functions are not supported.
- \(0b0001\) Unaligned single-copy atomicity and atomic functions with a 16-byte address range aligned to 16-bytes are supported.

All other values are reserved.

**FEAT_LSE2** implements the functionality identified by the value \(0b0001\).

In Armv8.2, the permitted values are \(0b0000\) and \(0b0001\).

From Armv8.4, the only permitted value is \(0b0001\).

**ST, bits [31:28]**

Identifies support for small translation tables. Defined values are:
- \(0b0000\) The maximum value of the TCR_ELx.\{T0SZ,T1SZ\} and VTCR_EL2.T0SZ fields is 39.
- \(0b0001\) The maximum value of the TCR_ELx.\{T0SZ,T1SZ\} and VTCR_EL2.T0SZ fields is 48 for 4KB and 16KB granules, and 47 for 64KB granules.

All other values are reserved.

**FEAT_TTST** implements the functionality identified by the value \(0b0001\).

If **FEAT_SEL2** is implemented, the only permitted value is \(0b0001\).

In an implementation which does not support **FEAT_SEL2**, the permitted values are \(0b0000\) and \(0b0001\).

**NV, bits [27:24]**

Nested Virtualization. If EL2 is implemented, indicates support for the use of nested virtualization. Defined values are:
- \(0b0000\) Nested virtualization is not supported.
- \(0b0001\) The HCR_EL2.\{AT, NV1, NV\} bits are implemented.
- \(0b0010\) The VNCR_EL2 register and the HCR_EL2.\{NV2, AT, NV1, NV\} bits are implemented.

All other values are reserved.

If EL2 is not implemented, the only permitted value is \(0b0000\).

**FEAT_NV** implements the functionality identified by the value \(0b0001\).

**FEAT_NV2** implements the functionality identified by the value \(0b0010\).

In Armv8.3, if EL2 is implemented, the permitted values are \(0b0000\) and \(0b0001\).

From Armv8.4, if EL2 is implemented, the permitted values are \(0b0000\), \(0b0001\), and \(0b0010\).

**CCIDX, bits [23:20]**

Support for the use of revised CCSIDR_EL1 register format. Defined values are:
- \(0b0000\) 32-bit format implemented for all levels of the CCSIDR_EL1.
- \(0b0001\) 64-bit format implemented for all levels of the CCSIDR_EL1.

All other values are reserved.

**FEAT_CCIDX** implements the functionality identified by the value \(0b0001\).

From Armv8.3, the permitted values are \(0b0000\) and \(0b0001\).
VARange, bits [19:16]
Indicates support for a larger virtual address. Defined values are:
0b0000  VMSAv8-64 supports 48-bit VAs.
0b0001  VMSAv8-64 supports 52-bit VAs when using the 64KB translation granule. The size for
other translation granules is not defined by this field.
All other values are reserved.
FEAT_LVA implements the functionality identified by the value 0b0001.
From Armv8.2, the permitted values are 0b0000 and 0b0001.

IESB, bits [15:12]
Indicates support for the IESB bit in the SCTLR_ELx registers. Defined values are:
0b0000  IESB bit in the SCTLR_ELx registers is not supported.
0b0001  IESB bit in the SCTLR_ELx registers is supported.
All other values are reserved.
FEAT_IESB implements the functionality identified by the value 0b0001.

LSM, bits [11:8]
Indicates support for LSMAOE and nTLSMD bits in SCTLR_EL1 and SCTLR_EL2. Defined
values are:
0b0000  LSMAOE and nTLSMD bits not supported.
0b0001  LSMAOE and nTLSMD bits supported.
All other values are reserved.
FEAT_LSMAOC implements the functionality identified by the value 0b0001.

UAO, bits [7:4]
User Access Override. Defined values are:
0b0000  UAO not supported.
0b0001  UAO supported.
All other values are reserved.
FEAT_UAO implements the functionality identified by the value 0b0001.
From Armv8.2, the only permitted value is 0b0001.

CnP, bits [3:0]
Indicates support for Common not Private translations. Defined values are:
0b0000  Common not Private translations not supported.
0b0001  Common not Private translations supported.
All other values are reserved.
FEAT_TTCNP implements the functionality identified by the value 0b0001.
From Armv8.2, the only permitted value is 0b0001.

Accessing ID_AA64MMFR2_EL1
Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, ID_AAA64MMFR2_EL1

if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        endif
    else
        UNDEFINED;
    endif
else if PSTATE.EL == EL1 then
    if EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_AAA64MMFR2_EL1) || boolean IMPLEMENTATION_DEFINED "ID_AAA64MMFR2_EL1 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AAA64MMFR2_EL1;
    endif
else if PSTATE.EL == EL2 then
    X[t, 64] = ID_AAA64MMFR2_EL1;
else if PSTATE.EL == EL3 then
    X[t, 64] = ID_AAA64MMFR2_EL1;
endif

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0111</td>
<td>0b010</td>
</tr>
</tbody>
</table>

```
**ID_AA64PFR0_EL1, AArch64 Processor Feature Register 0**

The ID_AA64PFR0_EL1 characteristics are:

**Purpose**

Provides additional information about implemented PE features in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

The external register EDPFR gives information from this register.

**Attributes**

ID_AA64PFR0_EL1 is a 64-bit register.

### Field descriptions

<table>
<thead>
<tr>
<th>Bit 63</th>
<th>Bit 60</th>
<th>Bit 59</th>
<th>Bit 56</th>
<th>Bit 55</th>
<th>Bit 52</th>
<th>Bit 51</th>
<th>Bit 48</th>
<th>Bit 47</th>
<th>Bit 44</th>
<th>Bit 43</th>
<th>Bit 40</th>
<th>Bit 39</th>
<th>Bit 36</th>
<th>Bit 35</th>
<th>Bit 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSV3</td>
<td>CSV2</td>
<td>RME</td>
<td>DIT</td>
<td>AMU</td>
<td>MPAM</td>
<td>SEL2</td>
<td>SVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAS</td>
<td>GIC</td>
<td>AdvSIMD</td>
<td>FP</td>
<td>EL3</td>
<td>EL2</td>
<td>EL1</td>
<td>EL0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CSV3, bits [63:60]**

Speculative use of faulting data. Defined values are:

- **0b0000** This PE does not disclose whether data loaded under speculation with a permission or domain fault can be used to form an address or generate condition codes or SVE predicate values to be used by other instructions in the speculative sequence.

- **0b0001** Data loaded under speculation with a permission or domain fault cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence. The execution timing of any other instructions in the speculative sequence is not a function of the data loaded under speculation.

All other values are reserved.

**FEAT_CSV3** implements the functionality identified by the value **0b0001**.

In Armv8.0, the permitted values are **0b0000** and **0b0001**.

From Armv8.5, the only permitted value is **0b0001**.

If **FEAT_E0PD** is implemented, **FEAT_CSV3** must be implemented.

**CSV2, bits [59:56]**

Speculative use of out of context branch targets. Defined values are:

- **0b0000** The implementation does not disclose whether **FEAT_CSV2** is implemented.

- **0b0001** **FEAT_CSV2** is implemented, but **FEAT_CSV2_2** and **FEAT_CSV2_3** are not implemented.

  **ID_AA64PFR1_EL1.CSV2_frac** determines whether either or both of **FEAT_CSV2_1p1** or **FEAT_CSV2_1p2** are implemented.

- **0b0010** **FEAT_CSV2_2** is implemented, but **FEAT_CSV2_3** is not implemented.

- **0b0011** **FEAT_CSV2_3** is implemented.

All other values are reserved.

**FEAT_CSV2** implements the functionality identified by the value **0b0001**.

**FEAT_CSV2_2** implements the functionality identified by the value **0b0010**.
FEAT_CSV2_3 implements the functionality identified by the feature 0b0011.
In Armv8.0, the permitted values are 0b0000, 0b0001, 0b0010, and 0b0011.
From Armv8.5, the permitted values are 0b0001, 0b0010, and 0b0011.

RME, bits [55:52]
Realm Management Extension (RME). Defined values are:

- 0b0000  Realm Management Extension not implemented.
- 0b0001  RMEv1 is implemented.

All other values are reserved.
FEAT_RME implements the functionality identified by the value 0b0001.

DIT, bits [51:48]
Data Independent Timing. Defined values are:

- 0b0000  AArch64 does not guarantee constant execution time of any instructions.
- 0b0001  AArch64 provides the PSTATE.DIT mechanism to guarantee constant execution time
of certain instructions.

All other values are reserved.
FEAT_DIT implements the functionality identified by the value 0b0001.
From Armv8.4, the only permitted value is 0b0001.

AMU, bits [47:44]
Indicates support for Activity Monitors Extension. Defined values are:

- 0b0000  Activity Monitors Extension is not implemented.
- 0b0001  FEAT_AMUv1 is implemented.
- 0b0010  FEAT_AMUv1p1 is implemented. As 0b0001 and adds support for virtualization of the
activity monitor event counters.

All other values are reserved.
FEAT_AMUv1 implements the functionality identified by the value 0b0001.
FEAT_AMUv1p1 implements the functionality identified by the value 0b0010.
In Armv8.0, the only permitted value is 0b0000.
In Armv8.4, the permitted values are 0b0000 and 0b0001.
From Armv8.6, the permitted values are 0b0000, 0b0001, and 0b0010.

MPAM, bits [43:40]
Indicates the major version number of support for the MPAM Extension.
Defined values are:

- 0b0000  The major version number of the MPAM extension is 0.
- 0b0001  The major version number of the MPAM extension is 1.

All other values are reserved.
When combined with the minor version number from ID_AA64PFR1_EL1.MPAM_frac, the
"major.minor" version is:

<table>
<thead>
<tr>
<th>MPAM Extension version</th>
<th>MPAM</th>
<th>MPAM_frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not implemented.</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>
SEL2, bits [39:36]

Secure EL2. Defined values are:
- 0b0000 Secure EL2 is not implemented.
- 0b0001 Secure EL2 is implemented.

All other values are reserved.

FEAT_SEL2 implements the functionality identified by the value 0b0001.

SVE, bits [35:32]

Scalable Vector Extension. Defined values are:
- 0b0000 SVE architectural state and programmers' model are not implemented.
- 0b0001 SVE architectural state and programmers' model are implemented.

All other values are reserved.

FEAT_SVE implements the functionality identified by the value 0b0001.

If implemented, refer to ID_AA64ZFR0_EL1 for information about which SVE instructions are available.

RAS, bits [31:28]

RAS Extension version. Defined values are:
- 0b0000 No RAS Extension.
- 0b0001 RAS Extension implemented.
- 0b0010 FEAT_RASv1p1 implemented and, if EL3 is implemented, FEAT_DoubleFault implemented. As 0b0001, and adds support for:
  - If EL3 is implemented, FEAT_DoubleFault.
  - Additional ERXMISCmn_EL1 System registers.
  - Additional System registers ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1, and the SCR_EL3.FIEN and HCR_EL2.FIEN trap controls, to support the optional RAS Common Fault Injection Model Extension.

Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS and support for the optional RAS Timestamp and RAS Common Fault Injection Model Extensions.

All other values are reserved.

FEAT_RAS implements the functionality identified by the value 0b0001.

FEAT_RASv1p1 and FEAT_DoubleFault implement the functionality identified by the value 0b0010.

In Armv8.0 and Armv8.1, the permitted values are 0b0000 and 0b0001.

In Armv8.2, the only permitted value is 0b0001.

From Armv8.4, if FEAT_DoubleFault is implemented, the only permitted value is 0b0010.

From Armv8.4, when FEAT_DoubleFault is not implemented, and ERRIDR_EL1 is 0, the permitted values are IMPLEMENTATION DEFINED 0b0001 or 0b0010.

---

For more information, see FEAT_MPAM.
Note

When the value of this field is 0b0001, ID_AA64PFR1_EL1.RAS_frac indicates whether FEAT_RASv1p1 is implemented.

GIC, bits [27:24]
System register GIC CPU interface. Defined values are:

- **0b0000**: GIC CPU interface system registers not implemented.
- **0b0001**: System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.
- **0b0011**: System register interface to version 4.1 of the GIC CPU interface is supported.

All other values are reserved.

AdvSIMD, bits [23:20]
Advanced SIMD. Defined values are:

- **0b0000**: Advanced SIMD is implemented, including support for the following SISD and SIMD operations:
  - Integer byte, halfword, word and doubleword element operations.
  - Single-precision and double-precision floating-point arithmetic.
  - Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.
- **0b0001**: As for 0b0000, and also includes support for half-precision floating-point arithmetic.
- **0b1111**: Advanced SIMD is not implemented.

All other values are reserved.

This field must have the same value as the FP field.

The permitted values are:

- **0b0000**: in an implementation with Advanced SIMD support that does not include the FEAT_FP16 extension.
- **0b0001**: in an implementation with Advanced SIMD support that includes the FEAT_FP16 extension.
- **0b1111**: in an implementation without Advanced SIMD support.

FP, bits [19:16]
Floating-point. Defined values are:

- **0b0000**: Floating-point is implemented, and includes support for:
  - Single-precision and double-precision floating-point types.
  - Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.
- **0b0001**: As for 0b0000, and also includes support for half-precision floating-point arithmetic.
- **0b1111**: Floating-point is not implemented.

All other values are reserved.

This field must have the same value as the AdvSIMD field.

The permitted values are:

- **0b0000**: in an implementation with floating-point support that does not include the FEAT_FP16 extension.
- **0b0001**: in an implementation with floating-point support that includes the FEAT_FP16 extension.
- **0b1111**: in an implementation without floating-point support.
EL3, bits [15:12]

EL3 Exception level handling. Defined values are:
- 0b0000  EL3 is not implemented.
- 0b0001  EL3 can be executed in AArch64 state only.
- 0b0010  EL3 can be executed in either AArch64 or AArch32 state.
All other values are reserved.

EL2, bits [11:8]

EL2 Exception level handling. Defined values are:
- 0b0000  EL2 is not implemented.
- 0b0001  EL2 can be executed in AArch64 state only.
- 0b0010  EL2 can be executed in either AArch64 or AArch32 state.
All other values are reserved.

EL1, bits [7:4]

EL1 Exception level handling. Defined values are:
- 0b0001  EL1 can be executed in AArch64 state only.
- 0b0010  EL1 can be executed in either AArch64 or AArch32 state.
All other values are reserved.

EL0, bits [3:0]

EL0 Exception level handling. Defined values are:
- 0b0001  EL0 can be executed in AArch64 state only.
- 0b0010  EL0 can be executed in either AArch64 or AArch32 state.
All other values are reserved.

Accessing ID_AA64PFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[ MRS <Xt>, ID_AA64PFR0_EL1 \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```
if PSTATE_EL == EL0 then
    if IsFeatureImplemented(Feat_IDST) then
        if EL2Enabled() & HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        else
            UNDEFINED;
    elsif PSTATE_EL == EL1 then
        if EL2Enabled() & HCR_EL2.TID3 == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            X[t, 64] = ID_AA64PFR0_EL1;
        else
            PSTATE_EL == EL2 then
                X[t, 64] = ID_AA64PFR0_EL1;
            elseif PSTATE_EL == EL3 then
                X[t, 64] = ID_AA64PFR0_EL1;
```
D17.2.68   ID_AA64PFR1_EL1, AArch64 Processor Feature Register 1

The ID_AA64PFR1_EL1 characteristics are:

**Purpose**

Reserved for future expansion of information about implemented PE features in AArch64 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

There are no configuration notes.

**Attributes**

ID_AA64PFR1_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:40]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[39:36]</td>
<td>Non-maskable Interrupt. Indicates support for Non-maskable interrupts. Defined values are:</td>
</tr>
<tr>
<td>0b0000</td>
<td>SCTLR_ELx.SPINTMASK, NMI} and PSTATE.ALLINT with its associated instructions are not supported.</td>
</tr>
<tr>
<td>0b0001</td>
<td>SCTLR_ELx.SPINTMASK, NMI} and PSTATE.ALLINT with its associated instructions are supported.</td>
</tr>
<tr>
<td>All other values are reserved.</td>
<td></td>
</tr>
<tr>
<td>FEAT_NMI implements the functionality identified by the value 0b0001.</td>
<td></td>
</tr>
<tr>
<td>From Armv8.8, the only permitted value is 0b0001.</td>
<td></td>
</tr>
<tr>
<td>[35:32]</td>
<td>CSV2 fractional field. Defined values are:</td>
</tr>
<tr>
<td>0b0000</td>
<td>Either ID_AA64PFR0_EL1.CSV2 is not 0b0002, or the implementation does not disclose whether FEAT_CSV2_1p1 is implemented.</td>
</tr>
<tr>
<td>0b0001</td>
<td>FEAT_CSV2_1p1 is implemented, but FEAT_CSV2_1p2 is not implemented.</td>
</tr>
<tr>
<td>0b0010</td>
<td>FEAT_CSV2_1p2 is implemented.</td>
</tr>
<tr>
<td>All other values are reserved.</td>
<td></td>
</tr>
<tr>
<td>FEAT_CSV2_1p1 implements the functionality identified by the value 0b0001.</td>
<td></td>
</tr>
<tr>
<td>FEAT_CSV2_1p2 implements the functionality identified by the value 0b0010.</td>
<td></td>
</tr>
<tr>
<td>From Armv8.0, the permitted values are 0b0000, 0b0001, and 0b0010.</td>
<td></td>
</tr>
<tr>
<td>The values 0b0001 and 0b0010 are permitted only when ID_AA64PFR0_EL1.CSV2 is 0b0001.</td>
<td></td>
</tr>
</tbody>
</table>
**RNDR_trap, bits [31:28]**

Random Number trap to EL3 field. Defined values are:

- **0b0000**  Trapping of RNDR and RNDRRS to EL3 is not supported.
- **0b0001**  Trapping of RNDR and RNDRRS to EL3 is supported. SCR_EL3.TRNDR is present.

All other values are reserved.

FEAT_RNG_TRAP implements the functionality identified by the value 0b0001.

**SME, bits [27:24]**

Scalable Matrix Extension. Defined values are:

- **0b0000**  SME architectural state and programmers' model are not implemented.
- **0b0001**  SME architectural state and programmers' model are implemented.

All other values are reserved.

FEAT_SME implements the functionality identified by the value 0b0001.

From Armv9.2, the permitted values are **0b0000** and **0b0001**.

If implemented, refer to ID_AA64SMFR0_EL1 and ID_AA64ZFR0_EL1 for information about which SME and SVE instructions are available.

**Bits [23:20]**

Reserved, RES0.

**MPAM_frac, bits [19:16]**

Indicates the minor version number of support for the MPAM Extension.

Defined values are:

- **0b0000**  The minor version number of the MPAM extension is 0.
- **0b0001**  The minor version number of the MPAM extension is 1.

All other values are reserved.

When combined with the major version number from ID_AA64PFR0_EL1.MPAM, The combined "major.minor" version is:

<table>
<thead>
<tr>
<th>MPAM Extension version</th>
<th>MPAM</th>
<th>MPAM_frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not implemented.</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
<tr>
<td>v0.1 is implemented.</td>
<td>0b0000</td>
<td>0b0001</td>
</tr>
<tr>
<td>v1.0 is implemented.</td>
<td>0b0001</td>
<td>0b0000</td>
</tr>
<tr>
<td>v1.1 is implemented.</td>
<td>0b0001</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

For more information, see *The Memory Partitioning and Monitoring (MPAM) Extension* on page A2-126.

**RAS_frac, bits [15:12]**

RAS Extension fractional field. Defined values are:

- **0b0000**  If ID_AA64PFR0_EL1.RAS == 0b0001, RAS Extension implemented.
- **0b0001**  If ID_AA64PFR0_EL1.RAS == 0b0001, as 0b0000 and adds support for:
  - Additional ERXMisc<m>_EL1 System registers.
  - Additional System registers ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1, and the SCR_EL3.FIEN and HCR_EL2.FIEN trap controls, to support the optional RAS Common Fault Injection Model Extension.
Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<\textit{n}>STATUS, and support for the optional RAS Timestamp and RAS Common Fault Injection Model Extensions.

All other values are reserved.

**FEAT\_RASv1p1** implements the functionality identified by the value \texttt{0b0001}.

This field is valid only if \texttt{ID\_AA64PFR0\_EL1.RAS} \(\equiv \texttt{0b0001}\).

### MTE, bits [11:8]

Support for the Memory Tagging Extension. Defined values are:

- \texttt{0b0000}: Memory Tagging Extension is not implemented.
- \texttt{0b0001}: Instruction-only Memory Tagging Extension is implemented.
- \texttt{0b0010}: Full Memory Tagging Extension is implemented.
- \texttt{0b0011}: Memory Tagging Extension is implemented with support for asymmetric Tag Check Fault handling.

All other values are reserved.

**FEAT\_MTE** implements the functionality identified by the value \texttt{0b0001}.

**FEAT\_MTE2** implements the functionality identified by the value \texttt{0b0010}.

**FEAT\_MTE3** implements the functionality identified by the value \texttt{0b0011}.

In Armv8.5, the permitted values are \texttt{0b0000}, \texttt{0b0001}, \texttt{0b0010}, and \texttt{0b0011}.

From Armv8.7, the value \texttt{0b0010} is not permitted.

### SSBS, bits [7:4]

Speculative Store Bypassing controls in AArch64 state. Defined values are:

- \texttt{0b0000}: AArch64 provides no mechanism to control the use of Speculative Store Bypassing.
- \texttt{0b0001}: AArch64 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypass Safe.
- \texttt{0b0010}: As \texttt{0b0001}, and adds the MSR and MRS instructions to directly read and write the PSTATE.SSBS field.

All other values are reserved.

**FEAT\_SSBS** implements the functionality identified by the value \texttt{0b0001}.

**FEAT\_SBSS2** implements the functionality identified by the value \texttt{0b0010}.

### BT, bits [3:0]

Branch Target Identification mechanism support in AArch64 state. Defined values are:

- \texttt{0b0000}: The Branch Target Identification mechanism is not implemented.
- \texttt{0b0001}: The Branch Target Identification mechanism is implemented.

All other values are reserved.

**FEAT\_BTI** implements the functionality identified by the value \texttt{0b0001}.

From Armv8.5, the only permitted value is \texttt{0b0001}.

### Accessing ID\_AA64PFR1\_EL1

Accesses to this register use the following encodings in the System register encoding space:
### MRS <Xt>, ID_AA64PFR1_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    else
      UNDEFINED;
  elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID3 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      X[t, 64] = ID_AA64PFR1_EL1;
  elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64PFR1_EL1;
  elsif PSTATE.EL == EL3 then
    X[t, 64] = ID_AA64PFR1_EL1;
ID_AA64SMFR0_EL1, SME Feature ID register 0

The ID_AA64SMFR0_EL1 characteristics are:

Purpose

Provides information about the implemented features of the AArch64 Scalable Matrix Extension.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers on page D17-5551.

Configurations

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

Attributes

ID_AA64SMFR0_EL1 is a 64-bit register.

Field descriptions

FA64, bit [63]

Indicates support for execution of the full A64 instruction set when the PE is in Streaming SVE mode. Defined values are:

0b0 Only those A64 instructions defined as being legal can be executed in Streaming SVE mode.

0b1 All implemented A64 instructions are legal for execution in Streaming SVE mode, when enabled at the current Exception level by SMCR_EL1.FA64, SMCR_EL2.FA64, and SMCR_EL3.FA64.

FEAT_SME_FA64 implements the functionality identified by the value 0b1.

Bits [62:60]

Reserved, RES0.

SMEver, bits [59:56]

When ID_AA64PFRI_EL1.SME != 0b0000:

Indicates support for SME instructions when FEAT_SME is implemented. Defined values are:

0b0000 The mandatory SME instructions are implemented.

All other values are reserved.

If FEAT_SME is implemented the only permitted value is 0b0000.

Otherwise:

Reserved, RES0.
I16I64, bits [55:52]
Indicates SME support for instructions that accumulate into 64-bit integer elements in the ZA array. Defined values are:

0b0000  Instructions that accumulate into 64-bit integer elements in the ZA array are not implemented.

0b1111  The variants of the ADDHA, ADDVA, SMOPA, SMOPS, SUMOPA, SUMOPS, UMOPA, UMOPS, USMOPA, and USMOPS instructions that accumulate into 64-bit integer tiles are implemented.

All other values are reserved.
FEAT_SME_I16I64 implements the functionality identified by the value 0b1111.
The only permitted values are 0b0000 and 0b1111.

Bits [51:49]
Reserved, RES0.

F64F64, bit [48]
Indicates SME support for instructions that accumulate into FP64 double-precision floating-point elements in the ZA array. Defined values are:

0b0  Instructions that accumulate into double-precision floating-point elements in the ZA array are not implemented.

0b1  The variants of the FMOPA and FMOPS instructions that accumulate into double-precision tiles are implemented.

FEAT_SME_F64F64 implements the functionality identified by the value 0b1.

Bits [47:40]
Reserved, RES0.

I8I32, bits [39:36]
Indicates SME support for instructions that accumulate 8-bit integer outer products into 32-bit integer tiles. Defined values are:

0b0000  Instructions that accumulate 8-bit outer products into 32-bit tiles are not implemented.

0b1111  The SMOPA, SMOPS, SUMOPA, SUMOPS, UMOPA, UMOPS, USMOPA, and USMOPS instructions that accumulate 8-bit outer products into 32-bit tiles are implemented.

All other values are reserved.
If FEAT_SME is implemented, the only permitted value is 0b1111.

F16F32, bit [35]
Indicates SME support for instructions that accumulate FP16 half-precision floating-point outer products into FP32 single-precision floating-point tiles. Defined values are:

0b0  Instructions that accumulate half-precision outer products into single-precision tiles are not implemented.

0b1  The FMOPA and FMOPS instructions that accumulate half-precision outer products into single-precision tiles are implemented.

If FEAT_SME is implemented, the only permitted value is 0b1.

B16F32, bit [34]
Indicates SME support for instructions that accumulate BFloat16 outer products into FP32 single-precision floating-point tiles. Defined values are:

0b0  Instructions that accumulate BFloat16 outer products into single-precision tiles are not implemented.
0b1  The BFMOPA and BFMOPS instructions that accumulate BFloat16 outer products into single-precision tiles are implemented.

If FEAT_SME is implemented, the only permitted value is 0b1.

**Bit [33]**

Reserved, RES0.

**F32F32, bit [32]**

Indicates SME support for instructions that accumulate FP32 single-precision floating-point outer products into single-precision floating-point tiles. Defined values are:

0b0  Instructions that accumulate single-precision outer products into single-precision tiles are not implemented.

0b1  The FMOPA and FMOPS instructions that accumulate single-precision outer products into single-precision tiles are implemented.

If FEAT_SME is implemented, the only permitted value is 0b1.

**Bits [31:0]**

Reserved, RES0.

### Accessing ID_AA64SMFR0_EL1

This register is read-only and can be accessed from EL1 and higher.

This register is only accessible from the AArch64 state.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ID_AA64SMFR0_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0100</td>
<td>0b101</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        else
            UNDEFINED;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && HCR_EL2.TID3 == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            X[t, 64] = ID_AA64SMFR0_EL1;
    elseif PSTATE.EL == EL2 then
        X[t, 64] = ID_AA64SMFR0_EL1;
    elseif PSTATE.EL == EL3 then
        X[t, 64] = ID_AA64SMFR0_EL1;
```
D17.2.70   ID_AA64ZFR0_EL1, SVE Feature ID register 0

The ID_AA64ZFR0_EL1 characteristics are:

Purpose

Provides additional information about the implemented features of the AArch64 Scalable Vector Extension instruction set, when one or more of FEAT_SVE and FEAT_SME is implemented.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers on page D17-5551.

Configurations

Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

If FEAT_SME is implemented and FEAT_SVE is not implemented, then SVE instructions can only be executed when the PE is in Streaming SVE mode and the instructions are legal to execute in Streaming SVE mode.

Attributes

ID_AA64ZFR0_EL1 is a 64-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>63</th>
<th>60</th>
<th>59</th>
<th>56</th>
<th>55</th>
<th>52</th>
<th>51</th>
<th>48</th>
<th>47</th>
<th>44</th>
<th>43</th>
<th>40</th>
<th>39</th>
<th>36</th>
<th>35</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>F64MM</td>
<td>F32MM</td>
<td>RES0</td>
<td>I8MM</td>
<td>SM4</td>
<td>RES0</td>
<td>SHA3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>28</td>
<td>23</td>
<td>20</td>
<td>19</td>
<td>16</td>
<td>15</td>
<td>8</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RES0</td>
<td>BF16</td>
<td>BitPerm</td>
<td>RES0</td>
<td>AES</td>
<td>SVEver</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits [63:60]

Reserved, RES0.

F64MM, bits [59:56]

Indicates support for SVE FP64 double-precision floating-point matrix multiplication instructions. Defined values are:

0b0000 Double-precision matrix multiplication and related instructions are not implemented.

0b0001 Double-precision variant of the FMMLA instruction, and the LD1RO* instructions are implemented. The 128-bit element variations of TRN1, TRN2, UZP1, UZP2, ZIP1, and ZIP2 are also implemented.

All other values are reserved.

FEAT_F64MM implements the functionality identified by 0b0001.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

F32MM, bits [55:52]

Indicates support for the SVE FP32 single-precision floating-point matrix multiplication instruction. Defined values are:

0b0000 Single-precision matrix multiplication instruction is not implemented.

0b0001 Single-precision variant of the FMMLA instruction is implemented.

All other values are reserved.

FEAT_F32MM implements the functionality identified by 0b0001.
From Arm v8.2, the permitted values are 0b0000 and 0b0001.

**Bits [51:48]**

Reserved, RES0.

**I8MM, bits [47:44]**

Indicates support for SVE Int8 matrix multiplication instructions. Defined values are:

- 0b0000  Int8 matrix multiplication instructions are not implemented.
- 0b0001  SMMLA, SUDOT, UMMLA, USMMLA, and USDOT instructions are implemented.

All other values are reserved.

FEAT_I8MM implements the functionality identified by 0b0001.

When Advanced SIMD and SVE are both implemented, this field must return the same value as ID_AA64ISAR1_EL1.I8MM.

From Armv8.6, the only permitted value is 0b0001.

**SM4, bits [43:40]**

Indicates support for SVE SM4 instructions. Defined values are:

- 0b0000  SVE SM4 instructions are not implemented.
- 0b0001  SVE SM4E and SM4EKEY instructions are implemented.

All other values are reserved.

FEAT_SVE_SM4 implements the functionality identified by 0b0001.

**Bits [39:36]**

Reserved, RES0.

**SHA3, bits [35:32]**

Indicates support for the SVE SHA3 instructions. Defined values are:

- 0b0000  SVE SHA3 instructions are not implemented.
- 0b0001  SVE RAX1 instruction is implemented.

All other values are reserved.

FEAT_SVE_SHA3 implements the functionality identified by 0b0001.

**Bits [31:24]**

Reserved, RES0.

**BF16, bits [23:20]**

Indicates support for SVE BFloat16 instructions. Defined values are:

- 0b0000  BFloat16 instructions are not implemented.
- 0b0001  BFCVT, BFCVTNT, BF DOT, BFMLALB, BFMLALT, and BFMM MLA instructions are implemented.
- 0b0010  As 0b0001, but the FPCR.EBF field is also supported.

All other values are reserved.

FEAT_BF16 implements the functionality identified by 0b0001.

FEAT_EBF16 implements the functionality identified by 0b0010.

This field must return the same value as ID_AA64ISAR1_EL1.BF16.

If FEAT_SME is implemented, the permitted values are 0b0001 and 0b0010.

Otherwise, from Armv8.6, the only permitted value is 0b0001.
BitPerm, bits [19:16]
Indicates support for SVE bit permute instructions. Defined values are:
0b0000  SVE bit permute instructions are not implemented.
0b0001  SVE BDEP, BEXT, and BGRP instructions are implemented.
All other values are reserved.
FEAT_SVE_BitPerm implements the functionality identified by 0b0001.

Bits [15:8]
Reserved, RES0.

AES, bits [7:4]
Indicates support for SVE AES instructions. Defined values are:
0b0000  SVE AES instructions are not implemented.
0b0001  SVE AESE, AESD, AESMC, and AESIMC instructions are implemented.
0b0010  As 0b0001, plus SVE PMULLB and PMULLT instructions with 64-bit source.
All other values are reserved.
FEAT_SVE_AES implements the functionality identified by the value 0b0001.
FEAT_SVE_PMULL128 implements the functionality identified by the value 0b0010.
The permitted values are 0b0000 and 0b0010.

SVEver, bits [3:0]
Indicates support for SVE instructions when one or more of FEAT_SME and FEAT_SVE is implemented. Defined values are:
0b0000  The SVE instructions are implemented.
0b0001  As 0b0000, and adds the mandatory SVE2 instructions.
All other values are reserved.
From Armv9, if this register is present, the value 0b0000 is not permitted.
FEAT_SVE2 adds the functionality identified by 0b0001 when the PE is not in Streaming SVE mode.
FEAT_SME adds the functionality identified by 0b0001 when the PE is in Streaming SVE mode.

Accessing ID_AA64ZFR0_EL1
Accesses to this register use the following encodings in the System register encoding space:

\[ MRS <Xt>, ID_AA64ZFR0_EL1 \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0100</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
else if PSTATE.EL == EL1 then
  if EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_AA64ZFR0_EL1) || boolean IMPLEMENTATION_DEFINED "ID_AA64ZFR0_EL1 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    X[t, 64] = ID_AA64ZFR0_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64ZFR0_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = ID_AA64ZFR0_EL1;
D17.2.71  ID_AFR0_EL1, AArch32 Auxiliary Feature Register 0

The ID_AFR0_EL1 characteristics are:

**Purpose**

Provides information about the IMPLEMENTATION DEFINED features of the PE in AArch32 state.

Must be interpreted with the Main ID Register, MIDR_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register ID_AFR0_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_AFR0[31:0].

**Attributes**

ID_AFR0_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>15-12</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>11-8</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>7-4</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>3-0</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

*Otherwise:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>15</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>11</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>8</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>4</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>0</td>
<td>UNKNOWN</td>
</tr>
</tbody>
</table>
Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_AFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[ \text{MRS} \langle Xt \rangle, \text{ID}_\text{AFR0}_\text{EL1} \]

<table>
<thead>
<tr>
<th>( \text{op0} )</th>
<th>( \text{op1} )</th>
<th>( \text{CRn} )</th>
<th>( \text{CRm} )</th>
<th>( \text{op2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if \( \text{PSTATE.EL} = \text{EL0} \) then
if \( \text{IsFeatureImplemented(\text{FEAT_IDST})} \) then
if \( \text{EL2Enabled()} \) \&\& \( \text{HCR}_\text{EL2}.TGE = '1' \) then
\( \text{AArch64.SystemAccessTrap(EL2, 0x18);} \)
else
\( \text{AArch64.SystemAccessTrap(EL1, 0x18);} \)
else
UNDEFINED;
elsif \( \text{PSTATE.EL} = \text{EL1} \) then
if \( \text{EL2Enabled()} \) \&\& \( \text{HCR}_\text{EL2}.TID3 = '1' \) then
\( \text{AArch64.SystemAccessTrap(EL2, 0x18);} \)
else
\( \text{X[t, 64]} = \text{ID}_\text{AFR0}_\text{EL1}; \)
elsif \( \text{PSTATE.EL} = \text{EL2} \) then
\( \text{X[t, 64]} = \text{ID}_\text{AFR0}_\text{EL1}; \)
elsif \( \text{PSTATE.EL} = \text{EL3} \) then
\( \text{X[t, 64]} = \text{ID}_\text{AFR0}_\text{EL1}; \)
D17.2.72  ID_DFR0_EL1, AArch32 Debug Feature Register 0

The ID_DFR0_EL1 characteristics are:

**Purpose**

Provides top level information about the debug system in AArch32 state.

Must be interpreted with the Main ID Register, MIDR_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register ID_DFR0_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_DFR0[31:0].

**Attributes**

ID_DFR0_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

![Field Descriptions Diagram](image)

**Bits [63:32]**

Reserved, RES0.

**TraceFilt, bits [31:28]**

Armv8.4 Self-hosted Trace Extension version. Defined values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>Armv8.4 Self-hosted Trace Extension not implemented.</td>
</tr>
<tr>
<td>0b0001</td>
<td>Armv8.4 Self-hosted Trace Extension implemented.</td>
</tr>
</tbody>
</table>

All other values are reserved.

FEAT_TRF implements the functionality added by the value 0b0001.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

**PerfMon, bits [27:24]**

Performance Monitors Extension version.

This field does not follow the standard ID scheme, but uses the alternative ID scheme described in *Alternative ID scheme used for the Performance Monitors Extension version* on page D17-5553

Defined values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>Performance Monitors Extension not implemented.</td>
</tr>
<tr>
<td>0b0001</td>
<td>Performance Monitors Extension, PMUv1 implemented.</td>
</tr>
<tr>
<td>0b0010</td>
<td>Performance Monitors Extension, PMUv2 implemented.</td>
</tr>
<tr>
<td>0b0011</td>
<td>Performance Monitors Extension, PMUv3 implemented.</td>
</tr>
<tr>
<td>0b0100</td>
<td>PMUv3 for Armv8.1. As 0b0011, and adds support for:</td>
</tr>
<tr>
<td></td>
<td>• Extended 16-bit PMEVTYPER&lt;n&gt;.evtCount field.</td>
</tr>
<tr>
<td></td>
<td>• If EL2 is implemented, the HDCR.HPMD control.</td>
</tr>
</tbody>
</table>
PMUv3 for Armv8.4. As 0b0100, and adds support for the PMMIR register.

PMUv3 for Armv8.5. As 0b0101, and adds support for:
- 64-bit event counters.
- If EL2 is implemented, the HDCR.HCCD control.
- If EL3 is implemented, the MDCR_EL3.SCCD control.

PMUv3 for Armv8.7. As 0b0110, and adds support for:
- The PMCR.FZO and, if EL2 is implemented, HDCR.HPMFZO controls.
- If EL3 is implemented, the MDCR_EL3.{MPMX,MCCD} controls.

PMUv3 for Armv8.8. As 0b0111, and:
- Extends the Common event number space to include 0x0040 to 0x00BF and 0x4040 to 0x40BF.
- Removes the CONSTRAINED UNPREDICTABLE behaviors if a reserved or unimplemented PMU event number is selected.

IMPLEMENATION DEFINED form of performance monitors supported, PMUv3 not supported. Arm does not recommend this value for new implementations.

All other values are reserved.

FEAT_PMUv3 implements the functionality identified by the value 0b0011.
FEAT_PMUv3p1 implements the functionality identified by the value 0b0100.
FEAT_PMUv3p4 implements the functionality identified by the value 0b0101.
FEAT_PMUv3p5 implements the functionality identified by the value 0b0110.
FEAT_PMUv3p7 implements the functionality identified by the value 0b0111.
FEAT_PMUv3p8 implements the functionality identified by the value 0b1000.

In any Armv8 implementation, the values 0b0001 and 0b0010 are not permitted.
From Armv8.1, if FEAT_PMUv3 is implemented, the value 0b0011 is not permitted.
From Armv8.4, if FEAT_PMUv3 is implemented, the value 0b0100 is not permitted.
From Armv8.5, if FEAT_PMUv3 is implemented, the value 0b0101 is not permitted.
From Armv8.7, if FEAT_PMUv3 is implemented, the value 0b0110 is not permitted.
From Armv8.8, if FEAT_PMUv3 is implemented, the value 0b0111 is not permitted.

--- Note ---
PMUv1 and PMUv2 are not permitted in an Armv8 implementation.

MProfDbg, bits [23:20]
M-profile Debug. Support for memory-mapped debug model for M-profile processors. Defined values are:
0b0000  Not supported.
0b0001  Support for M-profile Debug architecture, with memory-mapped access.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

MMapTrc, bits [19:16]
Memory-mapped Trace. Support for memory-mapped trace model. Defined values are:
0b0000  Not supported.
0b0001  Support for Arm trace architecture, with memory-mapped access.
All other values are reserved.
In Armv8-A, the permitted values are 0b0000 and 0b0001.
For more information, see the ARM® Embedded Trace Macrocell Architecture Specification, ETMv4 (ARM IHI 0064).

**CopTrc, bits [15:12]**
Support for System registers-based trace model, using registers in the coproc = 0b1110 encoding space. Defined values are:
- 0b0000 Not supported.
- 0b0001 Support for Arm trace architecture, with System registers access.
All other values are reserved.
In Armv8-A, the permitted values are 0b0000 and 0b0001.
For more information, see the ARM® Embedded Trace Macrocell Architecture Specification, ETMv4 (ARM IHI 0064).

**MMapDbg, bits [11:8]**
Memory-mapped Debug. Support for Armv7 memory-mapped debug model for A and R-profile processors. Defined values are:
- 0b0000 Not supported.
- 0b0100 Support for Armv7, v7 Debug architecture, with memory-mapped access.
- 0b0101 Support for Armv7, v7.1 Debug architecture, with memory-mapped access.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.
The optional memory map defined by Armv8 is not compatible with Armv7.

**CopSDbg, bits [7:4]**
Support for a System registers-based Secure debug model, using registers in the coproc = 0b1110 encoding space, for an A-profile processor that includes EL3.
If EL3 is not implemented and the implemented Security state is Non-secure state, this field is RES0. Otherwise, this field reads the same as bits [3:0].

**CopDbg, bits [3:0]**
Debug architecture version. Indicates presence of Armv8 debug architecture. Defined values are:
- 0b0000 Not supported.
- 0b0010 Armv6, v6 Debug architecture, with System registers access.
- 0b0011 Armv6, v6.1 Debug architecture, with System registers access.
- 0b0100 Armv7, v7 Debug architecture, with System registers access.
- 0b0101 Armv7, v7.1 Debug architecture, with System registers access.
- 0b0110 Armv8 debug architecture.
- 0b0111 Armv8 debug architecture with Virtualization Host Extensions.
- 0b1000 Armv8.2 debug architecture, FEAT_Debugv8p2.
- 0b1001 Armv8.4 debug architecture, FEAT_Debugv8p4.
- 0b1010 Armv8.8 debug architecture, FEAT_Debugv8p8.
All other values are reserved.
The values 0b0000, 0b0010, 0b0011, 0b0100, and 0b0101 are not permitted in Armv8.
FEAT_VHE adds the functionality identified by the value 0b0111.
FEAT_Debugv8p2 adds the functionality identified by the value 0b1000.
FEAT_Debugv8p4 adds the functionality identified by the value 0b1001.
FEAT_Debugv8p8 adds the functionality identified by the value 0b1010.
From Armv8.1, when FEAT_VHE is implemented the value 0b0110 is not permitted.
From Armv8.2, the values 0b0110 and 0b0111 are not permitted.
From Armv8.4, the value 0b1000 is not permitted.
From Armv8.8, the value 0b1001 is not permitted.

Otherwise:

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Reserved, UNKNOWN.</th>
</tr>
</thead>
</table>

**Accessing ID_DFR0_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ID_DFR0_EL1**

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    else
      UNDEFINED;
  elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TID3 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      X[t, 64] = ID_DFR0_EL1;
  elseif PSTATE.EL == EL2 then
    X[t, 64] = ID_DFR0_EL1;
  elseif PSTATE.EL == EL3 then
    X[t, 64] = ID_DFR0_EL1;
### D17.2.73 ID_DFR1_EL1, Debug Feature Register 1

The ID_DFR1_EL1 characteristics are:

**Purpose**

Provides top level information about the debug system in AArch32.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register ID_DFR1_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_DFR1[31:0].

**Note**

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

**Attributes**

ID_DFR1_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-8</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>7-4</td>
<td>HPMN0, Multi-threaded PMU extension</td>
<td>0b0000: Setting HDCR.HPMN to zero has CONSTRINED UNPREDICTABLE behavior. 0b0001: Setting HDCR.HPMN to zero has defined behavior. All other values are reserved.</td>
</tr>
<tr>
<td>3-0</td>
<td>MTPMU, Multi-threaded PMU</td>
<td>0b0000: FEAT_MTPMU not implemented. If FEAT_PMUv3 is implemented, it is IMPLEMENTATION DEFINED whether PMEVTYPE&lt;\n&gt;M. MT are read/write or RES0. 0b0001: FEAT_MTPMU and FEAT_PMUv3 implemented. PMEVTYPE&lt;\n&gt;M. MT are read/write. When FEAT_MTPMU is disabled, the Effective values of PMEVTYPE&lt;\n&gt;M. MT are 0.</td>
</tr>
</tbody>
</table>

Reserved, RES0.
0b1111 FEAT_MTPMU not implemented. If FEAT_PMUv3 is implemented, PMEVTPYPER<n>.MT are RES0.

All other values are reserved.

FEAT_MTPMU implements the functionality identified by the value 0b0001.

From Armv8.6, in an implementation that includes FEAT_PMUv3, the value 0b0000 is not permitted.

In an implementation that does not include FEAT_PMUv3, the value 0b0001 is not permitted.

**Otherwise:**

| 63 | 32 |
|------------------|
| UNKNOWN          |
|                  |

| 31 | 0 |
|------------------|
| UNKNOWN          |

**Bits [63:0]**

Reserved, UNKNOWN.

**Accessing ID_DFR1_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ID_DFR1_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0011</td>
<td>0b101</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
  elsif PSTATE_EL == EL1 then
    if EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || boolean IMPLEMENTATION_DEFINED "ID_DFR1_EL1 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      X[t, 64] = ID_DFR1_EL1;
  else
    X[t, 64] = ID_DFR1_EL1;
  elsif PSTATE_EL == EL2 then
    X[t, 64] = ID_DFR1_EL1;
  elsif PSTATE_EL == EL3 then
    X[t, 64] = ID_DFR1_EL1;
```
D17.2.74  ID_ISAR0_EL1, AArch32 Instruction Set Attribute Register 0

The ID_ISAR0_EL1 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register ID_ISAR0_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_ISAR0[31:0].

**Attributes**

ID_ISAR0_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th>BitField</th>
<th>BitCount</th>
<th>Swap</th>
<th>CmpBranch</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>Divide</td>
<td>Debug</td>
<td>Coproc</td>
</tr>
</tbody>
</table>

**Bits [63:28]**

Reserved, RES0.

**Divide, bits [27:24]**

Indicates the implemented Divide instructions. Defined values are:

- **0b0000** None implemented.
- **0b0001** Adds SDIV and UDIV in the T32 instruction set.
- **0b0010** As for 0b0001, and adds SDIV and UDIV in the A32 instruction set.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0010.

**Debug, bits [23:20]**

Indicates the implemented Debug instructions. Defined values are:

- **0b0000** None implemented.
- **0b0001** Adds BKPT.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

**Coproc, bits [19:16]**

Indicates the implemented System register access instructions. Defined values are:

- **0b0000** None implemented, except for instructions separately attributed by the architecture to provide access to AArch32 System registers and System instructions.
- **0b0001** Adds generic CDP, LDC, MCR, MRC, and STC.
0b0010   As for 0b0001, and adds generic CDP2, LDC2, MCR2, MRC2, and STC2.
0b0011   As for 0b0010, and adds generic MCRR and MRRC.
0b0100   As for 0b0011, and adds generic MCRR2 and MRRC2.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

CmpBranch, bits [15:12]
Indicates the implemented combined Compare and Branch instructions in the T32 instruction set.
Defined values are:
0b0000   None implemented.
0b0001   Adds CBNZ and CBZ.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

BitField, bits [11:8]
Indicates the implemented BitField instructions. Defined values are:
0b0000   None implemented.
0b0001   Adds BFC, BF1, SBFX, and UBFX.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

BitCount, bits [7:4]
Indicates the implemented Bit Counting instructions. Defined values are:
0b0000   None implemented.
0b0001   Adds CLZ.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

Swap, bits [3:0]
Indicates the implemented Swap instructions in the A32 instruction set. Defined values are:
0b0000   None implemented.
0b0001   Adds SWP and SWPB.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

Otherwise:

Bits [63:0]
Reserved, UNKNOWN.

Accessing ID_ISAR0_EL1
Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, ID_ISAR0_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        end if
    else
        UNDEFINED;
    end if
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_ISAR0_EL1;
    end if
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_ISAR0_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = ID_ISAR0_EL1;
D17.2.75  ID_ISAR1_EL1, AArch32 Instruction Set Attribute Register 1

The ID_ISAR1_EL1 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers on page D17-5551.*

**Configurations**

AArch64 System register ID_ISAR1_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_ISAR1[31:0].

**Attributes**

ID_ISAR1_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>Jazelle, bits [31:28]</td>
<td>Indicates the implemented Jazelle extension instructions. Defined values are:</td>
</tr>
<tr>
<td>0b0000</td>
<td>No support for Jazelle.</td>
</tr>
<tr>
<td>0b0001</td>
<td>Adds the BXJ instruction and the J bit in the PSR. This setting might indicate a trivial implementation of the Jazelle extension.</td>
</tr>
<tr>
<td>All other values are reserved.</td>
<td></td>
</tr>
<tr>
<td>In Armv8-A, the only permitted value is 0b0001.</td>
<td></td>
</tr>
<tr>
<td>Interwork, bits [27:24]</td>
<td>Indicates the implemented Interworking instructions. Defined values are:</td>
</tr>
<tr>
<td>0b0000</td>
<td>None implemented.</td>
</tr>
<tr>
<td>0b0001</td>
<td>Adds the BX instruction, and the T bit in the PSR.</td>
</tr>
<tr>
<td>0b0010</td>
<td>As for 0b0001, and adds the BLX instruction. PC loads have BX-like behavior.</td>
</tr>
<tr>
<td>0b0011</td>
<td>As for 0b0010, and guarantees that data-processing instructions in the A32 instruction set with the PC as the destination and the S bit clear have BX-like behavior.</td>
</tr>
<tr>
<td>All other values are reserved.</td>
<td></td>
</tr>
<tr>
<td>In Armv8-A, the only permitted value is 0b0011.</td>
<td></td>
</tr>
</tbody>
</table>
Immediate, bits [23:20]
Indicates the implemented data-processing instructions with long immediates. Defined values are:

- **0b0000** None implemented.
- **0b0001** Adds:
  - The MOVT instruction.
  - The MOV instruction encodings with zero-extended 16-bit immediates.
  - The T32 ADD and SUB instruction encodings with zero-extended 12-bit immediates, and the other ADD, ADR, and SUB encodings cross-referenced by the pseudocode for those encodings.

All other values are reserved.

In Armv8-A, the only permitted value is **0b0001**.

IfThen, bits [19:16]
Indicates the implemented If-Then instructions in the T32 instruction set. Defined values are:

- **0b0000** None implemented.
- **0b0001** Adds the IT instructions, and the IT bits in the PSRs.

All other values are reserved.

In Armv8-A, the only permitted value is **0b0001**.

Extend, bits [15:12]
Indicates the implemented Extend instructions. Defined values are:

- **0b0000** No scalar sign-extend or zero-extend instructions are implemented, where scalar instructions means non-Advanced SIMD instructions.
- **0b0001** Adds the SXTB, SXTH, UXTB, and UXTH instructions.
- **0b0100** As for **0b0001**, and adds the SXTB16, SXTAB, SXTAB16, SXTAH, UXTB16, UXTAB, UXTAB16, and UXTAH instructions.

All other values are reserved.

In Armv8-A, the only permitted value is **0b0100**.

Except_AR, bits [11:8]
Indicates the implemented A and R-profile exception-handling instructions. Defined values are:

- **0b0000** None implemented.
- **0b0001** Adds the SRS and RFE instructions, and the A and R-profile forms of the CPS instruction.

All other values are reserved.

In Armv8-A, the only permitted value is **0b0001**.

Except, bits [7:4]
Indicates the implemented exception-handling instructions in the A32 instruction set. Defined values are:

- **0b0000** Not implemented. This indicates that the User bank and Exception return forms of the LDM and STM instructions are not implemented.
- **0b0001** Adds the LDM (exception return), LDM (user registers), and STM (user registers) instruction versions.

All other values are reserved.

In Armv8-A, the only permitted value is **0b0001**.

Endian, bits [3:0]
Indicates the implemented Endian instructions. Defined values are:

- **0b0000** None implemented.
0b0001  Adds the SETEND instruction, and the E bit in the PSRs.
All other values are reserved.
In Armv8-A, the permitted values are 0b0000 and 0b0001.

Otherwise:

Bits [63:0]
Reserved, UNKNOWN.

Accessing ID_ISAR1_EL1
Accesses to this register use the following encodings in the System register encoding space:

|MRS <Xt>, ID_ISAR1_EL1|
|---|---|---|---|---|---|
|   |   |   |   |   |
|op0| op1| CRn| CRm| op2|
|0b1| 0b00| 0b0000| 0b0010| 0b001|

if PSTATE_EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    else
      UNDEFINED;
  elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID3 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      X[t, 64] = ID_ISAR1_EL1;
  elsif PSTATE_EL == EL2 then
    X[t, 64] = ID_ISAR1_EL1;
  elsif PSTATE_EL == EL3 then
    X[t, 64] = ID_ISAR1_EL1;
D17.2.76 ID_ISAR2_EL1, AArch32 Instruction Set Attribute Register 2

The ID_ISAR2_EL1 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see Principles of the ID scheme for fields in ID registers on page D17-5551.

**Configurations**

AArch64 System register ID_ISAR2_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_ISAR2[31:0].

**Attributes**

ID_ISAR2_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th>Position</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>RES0</td>
</tr>
<tr>
<td>31-28</td>
<td>Reversal</td>
</tr>
<tr>
<td>27-24</td>
<td>PSR_AR</td>
</tr>
<tr>
<td>23-20</td>
<td>MultU</td>
</tr>
<tr>
<td>19-16</td>
<td>MultS</td>
</tr>
<tr>
<td>15-12</td>
<td>Mult</td>
</tr>
<tr>
<td>11-8</td>
<td>MemHint</td>
</tr>
<tr>
<td>7-4</td>
<td>MultiAccessInt</td>
</tr>
<tr>
<td>3-0</td>
<td>LoadStore</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**Reversal, bits [31:28]**

Indicates the implemented Reversal instructions. Defined values are:

- 0b0000: None implemented.
- 0b0001: Adds the REV, REV16, and REVSH instructions.
- 0b0010: As for 0b0001, and adds the RBIT instruction.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0010.

**PSR_AR, bits [27:24]**

Indicates the implemented A and R-profile instructions to manipulate the PSR. Defined values are:

- 0b0000: None implemented.
- 0b0001: Adds the MRS and MSR instructions, and the exception return forms of data-processing instructions.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

The exception return forms of the data-processing instructions are:

- In the A32 instruction set, data-processing instructions with the PC as the destination and the S bit set. These instructions might be affected by the WithShifts attribute.
- In the T32 instruction set, the SUBS PC,LR,#N instruction.
MultU, bits [23:20]
Indicates the implemented advanced unsigned Multiply instructions. Defined values are:
- 0b0000: None implemented.
- 0b0001: Adds the UMULL and UMLAL instructions.
- 0b0010: As for 0b0001, and adds the UMAAL instruction.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0010.

MultS, bits [19:16]
Indicates the implemented advanced signed Multiply instructions. Defined values are:
- 0b0000: None implemented.
- 0b0001: Adds the SMULL and SMLAL instructions.
- 0b0010: As for 0b0001, and adds the SMLAB, SMLABT, SMLALBB, SMLALBT, SMLALTB, SMLALTT, SMLATT, SMLAWB, SMLAWT, SMULBB, SMULBT, SMULTB, SMULTT, SMULWB, and SMULWT instructions. Also adds the Q bit in the PSRs.
- 0b0011: As for 0b0010, and adds the SMLAD, SMLADX, SMLALD, SMLALDX, SMLSD, SMLSDX, SMLSLD, SMLSLDX, SMMLA, SMMLAR, SMMLS, SMMLS, SMMLSR, SMMLUL, SMMLUR, SMUAD, SMUADX, SMUSD, and SMUSDX instructions.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0011.

Mult, bits [15:12]
Indicates the implemented additional Multiply instructions. Defined values are:
- 0b0000: No additional instructions implemented. This means only MUL is implemented.
- 0b0001: Adds the MLA instruction.
- 0b0010: As for 0b0001, and adds the MLS instruction.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0010.

MultiAccessInt, bits [11:8]
Indicates the support for interruptible multi-access instructions. Defined values are:
- 0b0000: No support. This means the LDM and STM instructions are not interruptible.
- 0b0001: LDM and STM instructions are restartable.
- 0b0010: LDM and STM instructions are continuatable.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

MemHint, bits [7:4]
Indicates the implemented Memory Hint instructions. Defined values are:
- 0b0000: None implemented.
- 0b0001: Adds the PLD instruction.
- 0b0010: Adds the PLD instruction. (0b0001 and 0b0010 have identical effects.)
- 0b0011: As for 0b0001 (or 0b0010), and adds the PLI instruction.
- 0b0100: As for 0b0011, and adds the PLDW instruction.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0010.
LoadStore, bits [3:0]
Indicates the implemented additional load/store instructions. Defined values are:

- 0b0000: No additional load/store instructions implemented.
- 0b0001: Adds the LDRD and STRD instructions.
- 0b0010: As for 0b0001, and adds the Load Acquire (LDAB, LDAH, LDA, LDAEXB, LDAEXH, LDAEX, LDAEXD) and Store Release (STLB, STLH, STL, STLEXB, STLEXH, STLEX, STLEXD) instructions.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0010.

Otherwise:

Bits [63:0]
Reserved, UNKNOWN.

Accessing ID_ISAR2_EL1
Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS } \langle X_t \rangle, \text{ ID_ISAR2_EL1} \\
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b000 & 0b0000 & 0b0010 & 0b010
\end{array}
\]

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNEDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TID3 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    \text{X}[t, 64] = ID_ISAR2_EL1;
elsif PSTATE.EL == EL2 then
  \text{X}[t, 64] = ID_ISAR2_EL1;
elsif PSTATE.EL == EL3 then
  \text{X}[t, 64] = ID_ISAR2_EL1;
D17.2.77 ID_ISAR3_EL1, AArch32 Instruction Set Attribute Register 3

The ID_ISAR3_EL1 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register ID_ISAR3_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_ISAR3[31:0].

**Attributes**

ID_ISAR3_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>28</td>
</tr>
<tr>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**RES0**

Reserved, 0.

**T32EE, bits [31:28]**

Indicates the implemented T32EE instructions. Defined values are:

- 0b0000 None implemented.
- 0b0001 Adds the ENTERX and LEAVEX instructions, and modifies the load behavior to include null checking.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

**TrueNOP, bits [27:24]**

Indicates the implemented true NOP instructions. Defined values are:

- 0b0000 None implemented. This means there are no NOP instructions that do not have any register dependencies.
- 0b0001 Adds true NOP instructions in both the T32 and A32 instruction sets. This also permits additional NOP-compatible hints.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.
T32Copy, bits [23:20]
Indicates the support for T32 non flag-setting MOV instructions. Defined values are:

0b0000  Not supported. This means that in the T32 instruction set, encoding T1 of the MOV (register) instruction does not support a copy from a low register to a low register.

0b0001  Adds support for T32 instruction set encoding T1 of the MOV (register) instruction, copying from a low register to a low register.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

TabBranch, bits [19:16]
Indicates the implemented Table Branch instructions in the T32 instruction set. Defined values are:

0b0000  None implemented.

0b0001  Adds the TBB and TBH instructions.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

SynchPrim, bits [15:12]
Used in conjunction with ID_ISAR4.SynchPrim_frac to indicate the implemented Synchronization Primitive instructions. Defined values are:

0b0000  If SynchPrim_frac == 0b0000, no Synchronization Primitives implemented.

0b0001  If SynchPrim_frac == 0b0000, adds the LDREX and STREX instructions.

0b0010  If SynchPrim_frac == 0b0001, as for 0b0001, and also adds the LDREXD and STREXD instructions.

All other combinations of SynchPrim and SynchPrim_frac are reserved.
In Armv8-A, the only permitted value is 0b0010.

SVC, bits [11:8]
Indicates the implemented SVC instructions. Defined values are:

0b0000  Not implemented.

0b0001  Adds the SVC instruction.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

SIMD, bits [7:4]
Indicates the implemented SIMD instructions. Defined values are:

0b0000  None implemented.

0b0001  As for 0b0001, and adds the PKHBT, PKHTB, QADD16, QADD8, QASX, QSUB16, QSUB8, QSAX, QADD16, QADD8, QASX, SEL, SHADD16, SHADD8, SHASX, SHSUB16, SHSUB8, SHSAX, SSAT16, SSUB16, SSUB8, SSAX, SXTAB16, SXTB16, UADD16, UADD8, UASX, UHADD16, UHADD8, UHASX, UHASB16, UHASB8, UHASX, UQADD16, UQADD8, UQASX, UQSUB16, UQSUB8, UQSAX, USAD8, USADA8, USAT16, USUB16, USUB8, USAX, UXTAB16, and UXTB16 instructions. Also adds support for the GE[3:0] bits in the PSRs.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0011.
The SIMD field relates only to implemented instructions that perform SIMD operations on the general-purpose registers. In an implementation that supports Advanced SIMD and floating-point instructions, MVFR0, MVFR1, and MVFR2 give information about the implemented Advanced SIMD instructions.

**Saturate, bits [3:0]**

Indicates the implemented Saturate instructions. Defined values are:
- 0b0000 None implemented. This means no non-Advanced SIMD saturate instructions are implemented.
- 0b0001 Adds the QADD, QDADD, QDSUB, and QSUB instructions, and the Q bit in the PSRs. All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

**Otherwise:**

![Diagram](image-url)

**Bits [63:0]**

Reserved, UNKNOWN.

**Accessing ID_ISAR3_EL1**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS <Xt>, ID_ISAR3_EL1}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(Feat_IDST) then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TID3 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = ID_ISAR3_EL1;
elsif PSTATE.EL == EL2 then
  X[t, 64] = ID_ISAR3_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = ID_ISAR3_EL1;
### D17.2.78 ID_ISAR4_EL1, AArch32 Instruction Set Attribute Register 4

The ID_ISAR4_EL1 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32 state. Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register ID_ISAR4_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_ISAR4[31:0].

**Attributes**

ID_ISAR4_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

#### Bits [63:32]

Reserved, RES0.

#### SWP_frac, bits [31:28]

Indicates support for the memory system locking the bus for SWP or SWPB instructions. Defined values are:

- **0b0000**: SWP or SWPB instructions not implemented.
- **0b0001**: SWP or SWPB implemented but only in a uniprocessor context. SWP and SWPB do not guarantee whether memory accesses from other Requesters can come between the load memory access and the store memory access of the SWP or SWPB.

All other values are reserved. This field is valid only if ID_ISAR0.Swap is **0b0000**.

In Armv8-A, the only permitted value is **0b0000**.

#### PSR_M, bits [27:24]

Indicates the implemented M-profile instructions to modify the PSRs. Defined values are:

- **0b0000**: None implemented.
- **0b0001**: Adds the M-profile forms of the CPS, MRS, and MSR instructions.

All other values are reserved.

In Armv8-A, the only permitted value is **0b0000**.
**SynchPrim_frac, bits [23:20]**

Used in conjunction with ID_ISAR3.SynchPrim to indicate the implemented Synchronization Primitive instructions. Possible values are:

- **0b0000**  
  If SynchPrim == 0b0000, no Synchronization Primitives implemented. If SynchPrim == 0b0001, adds the LDREX and STREX instructions. If SynchPrim == 0b0001, also adds the CLREX, LDREXB, LDREXH, STREXB, STREXH, LDREXD, and STREXD instructions.

- **0b0011**  
  If SynchPrim == 0b0001, adds the LDREX, STREX, CLREX, LDREXB, LDREXH, STREXB, and STREXH instructions.

All other combinations of SynchPrim and SynchPrim_frac are reserved.

In Armv8-A, the only permitted value is 0b0000.

**Barrier, bits [19:16]**

Indicates the implemented Barrier instructions in the A32 and T32 instruction sets. Defined values are:

- **0b0000**  
  None implemented. Barrier operations are provided only as System instructions in the (coproc==0b1111) encoding space.

- **0b0001**  
  Adds the DMB, DSB, and ISB barrier instructions.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

**SMC, bits [15:12]**

Indicates the implemented SMC instructions. Defined values are:

- **0b0000**  
  None implemented.

- **0b0001**  
  Adds the SMC instruction.

All other values are reserved.

In Armv8-A, the permitted values are:

- If EL3 is implemented and EL1 can use AArch32, the only permitted value is 0b0001.
- If neither EL3 nor EL2 is implemented, the only permitted value is 0b0000.

If EL1 cannot use AArch32, this field has the value 0b0000.

**Writeback, bits [11:8]**

Indicates the support for Writeback addressing modes. Defined values are:

- **0b0000**  
  Basic support. Only the LDM, STM, PUSH, POP, SRS, and RFE instructions support writeback addressing modes. These instructions support all of their writeback addressing modes.

- **0b0001**  
  Adds support for all of the writeback addressing modes.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

**WithShifts, bits [7:4]**

Indicates the support for instructions with shifts. Defined values are:

- **0b0000**  
  Nonzero shifts supported only in MOV and shift instructions.

- **0b0001**  
  Adds support for shifts of loads and stores over the range LSL 0-3.

- **0b0011**  
  As for 0b0001, and adds support for other constant shift options, both on load/store and other instructions.

- **0b100**  
  As for 0b011, and adds support for register-controlled shift options.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0100.
Unpriv, bits [3:0]
Indicates the implemented unprivileged instructions. Defined values are:

- 0b0000: None implemented. No T variant instructions are implemented.
- 0b0001: Adds the LDRBT, LDRT, STRBT, and STRT instructions.
- 0b0010: As for 0b0001, and adds the LDRHT, LDRSBT, LDRSHT, and STRHT instructions.
All other values are reserved.

In Armv8-A, the only permitted value is 0b0010.

Otherwise:

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits [63:0]
Reserved, UNKNOWN.

Accessing ID_ISAR4_EL1
Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ID_ISAR4_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b100</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && HCR_EL2.TID3 == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            X[t, 64] = ID_ISAR4_EL1;
    elsif PSTATE.EL == EL2 then
        X[t, 64] = ID_ISAR4_EL1;
    elsif PSTATE.EL == EL3 then
        X[t, 64] = ID_ISAR4_EL1;
```
D17.2.79 ID_ISAR5_EL1, AArch32 Instruction Set Attribute Register 5

The ID_ISAR5_EL1 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, and ID_ISAR4_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register ID_ISAR5_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_ISAR5[31:0].

**Attributes**

ID_ISAR5_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>RES0</td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>31-28</td>
<td>VCMA</td>
</tr>
<tr>
<td>27-24</td>
<td>RDM</td>
</tr>
<tr>
<td>23-20</td>
<td>RES0</td>
</tr>
<tr>
<td>19-16</td>
<td>CRC32</td>
</tr>
<tr>
<td>15-12</td>
<td>SHA2</td>
</tr>
<tr>
<td>11-8</td>
<td>SHA1</td>
</tr>
<tr>
<td>7-4</td>
<td>AES</td>
</tr>
<tr>
<td>3-0</td>
<td>SEVL</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**VCMA, bits [31:28]**

Indicates AArch32 support for complex number addition and multiplication where numbers are stored in vectors. Defined values are:

- **0b0000**: The VCMLA and VCADD instructions are not implemented in AArch32.
- **0b0001**: The VCMLA and VCADD instructions are implemented in AArch32.

All other values are reserved.

**FEAT_FCMA** implements the functionality identified by **0b0001**.

In Armv8.0, Armv8.1, and Armv8.2, the only permitted value is **0b0000**.

From Armv8.3, the only permitted value is **0b0001**.

**RDM, bits [27:24]**

Indicates whether the VQRDMLAH and VQRDMLSH instructions are implemented in AArch32 state. Defined values are:

- **0b0000**: No VQRDMLAH and VQRDMLSH instructions implemented.
- **0b0001**: VQRDMLAH and VQRDMLSH instructions implemented.

All other values are reserved.

**FEAT_RDM** implements the functionality identified by the value **0b0001**.

In Armv8.0, the only permitted value is **0b0000**.

From Armv8.1, the only permitted value is **0b0001**.
Bits [23:20]
Reserved, RES0.

CRC32, bits [19:16]
Indicates whether the CRC32 instructions are implemented in AArch32 state.

- **0b0000**: No CRC32 instructions implemented.
- **0b0001**: CRC32B, CRC32H, CRC32W, CRC32CB, CRC32CH, and CRC32CW instructions implemented.

All other values are reserved.

In Armv8.0, the permitted values are 0b0000 and 0b0001.
From Armv8.1, the only permitted value is 0b0001.

SHA2, bits [15:12]
Indicates whether the SHA2 instructions are implemented in AArch32 state.

- **0b0000**: No SHA2 instructions implemented.
- **0b0001**: SHA256H, SHA256H2, SHA256SU0, and SHA256SU1 implemented.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0001.

SHA1, bits [11:8]
Indicates whether the SHA1 instructions are implemented in AArch32 state.

- **0b0000**: No SHA1 instructions implemented.
- **0b0001**: SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 implemented.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0001.

AES, bits [7:4]
Indicates whether the AES instructions are implemented in AArch32 state.

- **0b0000**: No AES instructions implemented.
- **0b0001**: AESE, AESD, AESMC, and AESIMC implemented.
- **0b0010**: As for 0b0001, plus VMULL (polynomial) instructions operating on 64-bit data quantities.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0010.

SEVL, bits [3:0]
Indicates whether the SEVL instruction is implemented in AArch32 state.

- **0b0000**: SEVL is implemented as a NOP.
- **0b0001**: SEVL is implemented as Send Event Local.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.
Otherwise:

Bits [63:0]
Reserved, UNKNOWN.

Accessing ID_ISAR5_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS } <X_t>, \text{ ID_ISAR5_EL1}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(_FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    else
      UNDEFINED;
  elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID3 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      X[t, 64] = ID_ISAR5_EL1;
    else
      PSTATE.EL == EL2 then
      X[t, 64] = ID_ISAR5_EL1;
    else
      PSTATE.EL == EL3 then
      X[t, 64] = ID_ISAR5_EL1;
D17.2.80 ID_ISAR6_EL1, AArch32 Instruction Set Attribute Register 6

The ID_ISAR6_EL1 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1 and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see Principles of the ID scheme for fields in ID registers on page D17-5551.

**Configurations**

AArch64 System register ID_ISAR6_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_ISAR6[31:0].

**Note**

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

**Attributes**

ID_ISAR6_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>RES0</th>
<th>I8MM</th>
<th>BF16</th>
<th>SPECRES</th>
<th>SB</th>
<th>FHM</th>
<th>DP</th>
<th>JSCVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>32</td>
<td>28</td>
<td>27</td>
<td>28</td>
<td>31</td>
<td>23</td>
<td>20</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>31</td>
<td>28</td>
<td>27</td>
<td>24</td>
<td>23</td>
<td>20</td>
<td>19</td>
<td>16</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

**Bits [63:28]**

Reserved, RES0.

**I8MM, bits [27:24]**

Indicates support for Advanced SIMD and floating-point Int8 matrix multiplication instructions in AArch32 state. Defined values of this field are:

0b0000  Int8 matrix multiplication instructions are not implemented.

0b0001  VSMLA, VSUDOT, VUMMLA, VUSMMLA, and VUSDOT instructions are implemented.

All other values are reserved.

**BF16, bits [23:20]**

Indicates support for Advanced SIMD and floating-point BFloat16 instructions in AArch32 state. Defined values are:

0b0000  BFloat16 instructions are not implemented.

0b0001  VCVT, VCVTB, VCVTT, VDOT, VFMA, VFMA, and VMMLA instructions with BFloat16 operand or result types are implemented.

All other values are reserved.

**FEAT-AA32BF16** implements the functionality identified by 0b0001.
**SPECRES, bits [19:16]**
Indicates support for Speculation invalidation instructions in AArch32 state. Defined values are:
- 0b0000  Prediction invalidation instructions are not implemented.
- 0b0001  CFPRCTX, DVPRCTX, and CPPRCTX instructions are implemented.
All other values are reserved.
**FEAT_SPECRES** implements the functionality identified by 0b0001.
From Armv8.5, the only permitted value is 0b0001.

**SB, bits [15:12]**
Indicates support for the SB instruction in AArch32 state. Defined values are:
- 0b0000  SB instruction is not implemented.
- 0b0001  SB instruction is implemented.
All other values are reserved.
**FEAT_SB** implements the functionality identified by 0b0001.
From Armv8.5, the only permitted value is 0b0001.

**FHM, bits [11:8]**
Indicates support for Advanced SIMD and floating-point VFMAL and VFMSL instructions in AArch32 state. Defined values are:
- 0b0000  VFMAL and VMFSL instructions are not implemented.
- 0b0001  VFMAL and VMFSL instructions are implemented.
All other values are reserved.
**FEAT_FHM** implements the functionality identified by 0b0001.
From Armv8.2, the permitted values are 0b0000 and 0b0001.

**DP, bits [7:4]**
Indicates support for dot product instructions in AArch32 state. Defined values are:
- 0b0000  Dot product instructions are not implemented.
- 0b0001  VUDOT and VSDOT instructions are implemented.
All other values are reserved.
**FEAT_DotProd** implements the functionality identified by 0b0001.
In Armv8.2, the permitted values are 0b0000 and 0b0001.
From Armv8.4, the only permitted value is 0b0001.

**JSCVT, bits [3:0]**
Indicates support for the VJCVT instruction in AArch32 state. Defined values are:
- 0b0000  The VJCVT instruction is not implemented.
- 0b0001  The VJCVT instruction is implemented.
All other values are reserved.
**FEAT_JSCVT** implements the functionality identified by 0b0001.
In Armv8.0, Armv8.1, and Armv8.2, the only permitted value is 0b0000.
From Armv8.3, if Advanced SIMD or Floating-point is implemented, the only permitted value is 0b0001.
From Armv8.3, if Advanced SIMD or Floating-point is not implemented, the only permitted value is 0b0000.
Otherwise:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Accessing ID_ISAR6_EL1</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:0]</td>
<td>Reserved, UNKNOWN.</td>
</tr>
</tbody>
</table>

Access to this register uses the following encodings in the System register encoding space:

**MRS <Xt>, ID_ISAR6_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b11</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_ISAR6_EL1) || boolean IMPLEMENTATION_DEFINED "ID_ISAR6_EL1 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = ID_ISAR6_EL1;
elsif PSTATE.EL == EL2 then
  X[t, 64] = ID_ISAR6_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = ID_ISAR6_EL1;
```
D17.2.81 ID_MMFR0_EL1, AArch32 Memory Model Feature Register 0

The ID_MMFR0_EL1 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register ID_MMFR0_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_MMFR0[31:0].

**Attributes**

ID_MMFR0_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>RES0</td>
</tr>
<tr>
<td>31</td>
<td>InnerShr</td>
</tr>
<tr>
<td>27</td>
<td>FCSE</td>
</tr>
<tr>
<td>23</td>
<td>AuxReg</td>
</tr>
<tr>
<td>19</td>
<td>TCM</td>
</tr>
<tr>
<td>15</td>
<td>ShareLvl</td>
</tr>
<tr>
<td>11</td>
<td>OuterShr</td>
</tr>
<tr>
<td>7</td>
<td>PMSA</td>
</tr>
<tr>
<td>3</td>
<td>VMSA</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**InnerShr, bits [31:28]**

Innermost Shareability. Indicates the innermost shareability domain implemented. Defined values are:

- 0b0000 Implemented as Non-cacheable.
- 0b0001 Implemented with hardware coherency support.
- 0b1111 Shareability ignored.

All other values are reserved.

From Armv8 the permitted values are 0b0000, 0b0001, and 0b1111.

This field is valid only if the implementation supports two levels of shareability, as indicated by ID_MMFR0_EL1.ShareLvl having the value 0b0001.

When ID_MMFR0_EL1.ShareLvl is zero, this field is UNKNOWN.

**FCSE, bits [27:24]**

Indicates whether the implementation includes the FCSE. Defined values are:

- 0b0000 Not supported.
- 0b0001 Support for FCSE.

All other values are reserved.

From Armv8 the only permitted value is 0b0000.

**AuxReg, bits [23:20]**

Auxiliary Registers. Indicates support for Auxiliary registers. Defined values are:

- 0b0000 None supported.
0b0001  Support for Auxiliary Control Register only.
0b0010  Support for Auxiliary Fault Status Registers (AIFSR and ADFSR) and Auxiliary Control Register.

All other values are reserved.
From Armv8 the only permitted value is 0b0010.

--- Note ---
Accesses to unimplemented Auxiliary registers are UNDEFINED.

TCM, bits [19:16]
Indicates support for TCMs and associated DMAs. Defined values are:
0b0000  Not supported.
0b0001  Support is IMPLEMENTATION DEFINED.
0b0010  Support for TCM only, Armv6 implementation.
0b0011  Support for TCM and DMA, Armv6 implementation.
All other values are reserved.
In Armv8-A the only permitted value is 0b0000.

ShareLvl, bits [15:12]
Shareability Levels. Indicates the number of shareability levels implemented. Defined values are:
0b0000  One level of shareability implemented.
0b0001  Two levels of shareability implemented.
All other values are reserved.
From Armv8 the only permitted value is 0b0001.

OuterShr, bits [11:8]
Outermost Shareability. Indicates the outermost shareability domain implemented. Defined values are:
0b0000  Implemented as Non-cacheable.
0b0001  Implemented with hardware coherency support.
0b1111  Shareability ignored.
All other values are reserved.
From Armv8 the permitted values are 0b0000, 0b0001, and 0b1111.

PMSA, bits [7:4]
Indicates support for a PMSA. Defined values are:
0b0000  Not supported.
0b0001  Support for IMPLEMENTATION DEFINED PMSA.
0b0010  Support for PMSAv6, with a Cache Type Register implemented.
0b0011  Support for PMSAv7, with support for memory subsections. Armv7-R profile.
All other values are reserved.
In Armv8-A the only permitted value is 0b0000.

VMSA, bits [3:0]
Indicates support for a VMSA. Defined values are:
0b0000  Not supported.
0b0001  Support for IMPLEMENTATION DEFINED VMSA.
0b0010  Support for VMSAv6, with Cache and TLB Type Registers implemented.
0b0011  Support for VMSAv7, with support for remapping and the Access flag. Armv7-A profile.

0b0100  As for 0b0011, and adds support for the PXN bit in the Short-descriptor translation table format descriptors.

0b0101  As for 0b0100, and adds support for the Long-descriptor translation table format.

All other values are reserved.

In Armv8-A the only permitted value is 0b0101.

**Otherwise:**

![Bits [63:0]

Reserved, UNKNOWN.

**Accessing ID_MMFR0_EL1**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS } \langle X_t \rangle, \text{ID_MMFR0_EL1}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b100</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
  elseif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID3 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      X[t, 64] = ID_MMFR0_EL1;
  elseif PSTATE_EL == EL2 then
    X[t, 64] = ID_MMFR0_EL1;
  elseif PSTATE_EL == EL3 then
    X[t, 64] = ID_MMFR0_EL1;
```

Otherwise
### D17.2.82 ID_MMFR1_EL1, AArch32 Memory Model Feature Register 1

The ID_MMFR1_EL1 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register ID_MMFR1_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_MMFR1[31:0].

**Attributes**

ID_MMFR1_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Reserved (RES0)</td>
</tr>
<tr>
<td>31</td>
<td>Branch Predictor (BPred) 0b0000 = No branch predictor, or no MMU present. Implies a fixed MPU configuration. 0b0001 = Branch predictor requires flushing on:  - Enabling or disabling a stage of address translation.  - Writing new data to instruction locations.  - Writing new mappings to the translation tables.  - Changes to the TTBR0, TTBR1, or TTBCR registers.  - Changes to the ContextID or ASID, or to the FCSE ProcessID if this is supported. 0b0010 = Branch predictor requires flushing on:  - Enabling or disabling a stage of address translation.  - Writing new data to instruction locations.  - Writing new mappings to the translation tables.  - Any change to the TTBR0, TTBR1, or TTBCR registers without a change to the corresponding ContextID or ASID, or FCSE ProcessID if this is supported. 0b0011 = Branch predictor requires flushing only on writing new data to instruction locations. 0b0100 = For execution correctness, branch predictor requires no flushing at any time. All other values are reserved.</td>
</tr>
<tr>
<td>28</td>
<td>L1TstCln</td>
</tr>
<tr>
<td>24</td>
<td>L1Uni</td>
</tr>
<tr>
<td>20</td>
<td>L1Hvd</td>
</tr>
<tr>
<td>16</td>
<td>L1UniSW</td>
</tr>
<tr>
<td>12</td>
<td>L1HvdSW</td>
</tr>
<tr>
<td>8</td>
<td>L1UniVA</td>
</tr>
<tr>
<td>4</td>
<td>L1HvdVA</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.
In Armv8-A, the permitted values are \texttt{0b0010}, \texttt{0b0011}, and \texttt{0b0100}. For values other than \texttt{0b0000} and \texttt{0b0100} the Arm Architecture Reference Manual, or the product documentation, might give more information about the required maintenance.

\textbf{L1TstCln, bits [27:24]}

Level 1 cache Test and Clean. Indicates the supported Level 1 data cache test and clean operations, for Harvard or unified cache implementations. Defined values are:

\begin{itemize}
\item \texttt{0b0000} None supported.
\item \texttt{0b0001} Supported Level 1 data cache test and clean operations are:
  \begin{itemize}
  \item Test and clean data cache.
  \end{itemize}
\item \texttt{0b0010} As for \texttt{0b0001}, and adds:
  \begin{itemize}
  \item Test, clean, and invalidate data cache.
  \end{itemize}
\end{itemize}

All other values are reserved.

In Armv8-A, the only permitted value is \texttt{0b0000}.

\textbf{L1Uni, bits [23:20]}

Level 1 Unified cache. Indicates the supported entire Level 1 cache maintenance operations for a unified cache implementation. Defined values are:

\begin{itemize}
\item \texttt{0b0000} None supported.
\item \texttt{0b0001} Supported entire Level 1 cache operations are:
  \begin{itemize}
  \item Invalidate cache, including branch predictor if appropriate.
  \item Invalidate branch predictor, if appropriate.
  \end{itemize}
\item \texttt{0b0010} As for \texttt{0b0001}, and adds:
  \begin{itemize}
  \item Clean cache, using a recursive model that uses the cache dirty status bit.
  \item Clean and invalidate cache, using a recursive model that uses the cache dirty status bit.
  \end{itemize}
\end{itemize}

All other values are reserved.

In Armv8-A, the only permitted value is \texttt{0b0000}.

\textbf{L1Hvd, bits [19:16]}

Level 1 Harvard cache. Indicates the supported entire Level 1 cache maintenance operations for a Harvard cache implementation. Defined values are:

\begin{itemize}
\item \texttt{0b0000} None supported.
\item \texttt{0b0001} Supported entire Level 1 cache operations are:
  \begin{itemize}
  \item Invalidate instruction cache, including branch predictor if appropriate.
  \item Invalidate branch predictor, if appropriate.
  \end{itemize}
\item \texttt{0b0010} As for \texttt{0b0001}, and adds:
  \begin{itemize}
  \item Invalidate data cache.
  \item Invalidate data cache and instruction cache, including branch predictor if appropriate.
  \end{itemize}
\item \texttt{0b0011} As for \texttt{0b0010}, and adds:
  \begin{itemize}
  \item Clean data cache, using a recursive model that uses the cache dirty status bit.
  \item Clean and invalidate data cache, using a recursive model that uses the cache dirty status bit.
  \end{itemize}
\end{itemize}

All other values are reserved.

In Armv8-A, the only permitted value is \texttt{0b0000}. 
L1UniSW, bits [15:12]

Level 1 Unified cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a unified cache implementation. Defined values are:

- **0b0000**: None supported.
- **0b0001**: Supported Level 1 unified cache line maintenance operations by set/way are:
  - Clean cache line by set/way.
- **0b0010**: As for 0b0001, and adds:
  - Clean and invalidate cache line by set/way.
- **0b0011**: As for 0b0010, and adds:
  - Invalidate cache line by set/way.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

L1HvdSW, bits [11:8]

Level 1 Harvard cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a Harvard cache implementation. Defined values are:

- **0b0000**: None supported.
- **0b0001**: Supported Level 1 Harvard cache line maintenance operations by set/way are:
  - Clean data cache line by set/way.
  - Clean and invalidate data cache line by set/way.
- **0b0010**: As for 0b0001, and adds:
  - Invalidate data cache line by set/way.
- **0b0011**: As for 0b0010, and adds:
  - Invalidate instruction cache line by set/way.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

L1UniVA, bits [7:4]

Level 1 Unified cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a unified cache implementation. Defined values are:

- **0b0000**: None supported.
- **0b0001**: Supported Level 1 unified cache line maintenance operations by VA are:
  - Clean cache line by VA.
  - Invalidate cache line by VA.
  - Clean and invalidate cache line by VA.
- **0b0010**: As for 0b0001, and adds:
  - Invalidate branch predictor by VA, if branch predictor is implemented.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

L1HvdVA, bits [3:0]

Level 1 Harvard cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a Harvard cache implementation. Defined values are:

- **0b0000**: None supported.
- **0b0001**: Supported Level 1 Harvard cache line maintenance operations by VA are:
  - Clean data cache line by VA.
  - Invalidate data cache line by VA.
• Clean and invalidate data cache line by VA.
• Clean instruction cache line by VA.

0b0010  As for 0b0001, and adds:
• Invalidate branch predictor by VA, if branch predictor is implemented.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

**Otherwise:**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>31-0</td>
<td>UNKNOWN</td>
</tr>
</tbody>
</table>

**Accessing ID_MMFR1_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ID_MMFR1_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
  elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TID3 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      UNDEFINED;
    else
      X[t, 64] = ID_MMFR1_EL1;
  elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_MMFR1_EL1;
  elsif PSTATE.EL == EL3 then
    X[t, 64] = ID_MMFR1_EL1;
D17.2.83  ID_MMFR2_EL1, AArch32 Memory Model Feature Register 2

The ID_MMFR2_EL1 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register ID_MMFR2_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_MMFR2[31:0].

**Attributes**

ID_MMFR2_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

```plaintext
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   | HWAccFlg | WFIStall | MemBarr | UniTLB | HvdTLB | L1HvdRng | L1HvdBG | L1HvdFG |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
     63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40
```

**Bits [63:32]**

Reserved, RES0.

**HWAccFlg, bits [31:28]**

Hardware Access Flag. In earlier versions of the Arm Architecture, this field indicates support for a Hardware Access flag, as part of the VMSAv7 implementation. Defined values are:

- 0b0000  Not supported.
- 0b0001  Support for VMSAv7 Access flag, updated in hardware.

All other values are reserved.

From Armv8, the only permitted value is 0b0000.

**WFIStall, bits [27:24]**

Wait For Interrupt Stall. Indicates the support for Wait For Interrupt (WFI) stalling. Defined values are:

- 0b0000  Not supported.
- 0b0001  Support for WFI stalling.

All other values are reserved.

From Armv8, the permitted values are 0b0000 and 0b0001.

**MemBarr, bits [23:20]**

Memory Barrier. Indicates the supported memory barrier System instructions in the (coproc==0b1111) encoding space:

- 0b0000  None supported.
- 0b0001  Supported memory barrier System instructions are:
  - Data Synchronization Barrier (DSB).
0b0010  As for 0b0001, and adds:
  • Instruction Synchronization Barrier (ISB).
  • Data Memory Barrier (DMB).
All other values are reserved.
From Armv8, the only permitted value is 0b0010.
Arm deprecates the use of these operations. ID_ISAR4.Barrier_instrs indicates the level of support
for the preferred barrier instructions.

UniTLB, bits [19:16]
Unified TLB. Indicates the supported TLB maintenance operations, for a unified TLB
implementation. Defined values are:
0b0000  Not supported.
0b0001  Supported unified TLB maintenance operations are:
  • Invalidate all entries in the TLB.
  • Invalidate TLB entry by VA.
0b0010  As for 0b0001, and adds:
  • Invalidate TLB entries by ASID match.
0b0011  As for 0b0010, and adds:
  • Invalidate instruction TLB and data TLB entries by All ASID. This is a
    shared unified TLB operation.
0b0100  As for 0b0011, and adds:
  • Invalidate Hyp mode unified TLB entry by VA.
  • Invalidate entire Non-secure PL1&0 unified TLB.
  • Invalidate entire Hyp mode unified TLB.
0b0101  As for 0b0100, and adds the following operations: TLBIMVALIS, TLBIMVAALIS,
    TLBIMVALIS, TLBIMVAL, TLBIMVAAL, TLBIMVAALH.
0b0110  As for 0b0101, and adds the following operations: TLBIPAS2IS, TLBIPAS2LIS,
    TLBIPAS2, TLBIPAS2L.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0110.

HvdTLB, bits [15:12]
If the Unified TLB field (UniTLB, bits [19:16]) is not 0000, then the meaning of this field is
IMPLEMENTATION DEFINED. Arm deprecates the use of this field by software.

LIHvdRng, bits [11:8]
Level 1 Harvard cache Range. Indicates the supported Level 1 cache maintenance range operations,
for a Harvard cache implementation. Defined values are:
0b0000  Not supported.
0b0001  Supported Level 1 Harvard cache maintenance range operations are:
  • Invalidate data cache range by VA.
  • Invalidate instruction cache range by VA.
  • Clean data cache range by VA.
  • Clean and invalidate data cache range by VA.
All other values are reserved.
From Armv8, the only permitted value is 0b0000.
L1HvdBG, bits [7:4]

Level 1 Harvard cache Background fetch. Indicates the supported Level 1 cache background fetch operations, for a Harvard cache implementation. When supported, background fetch operations are non-blocking operations. Defined values are:

0b0000 Not supported.
0b0001 Supported Level 1 Harvard cache background fetch operations are:
  • Fetch instruction cache range by VA.
  • Fetch data cache range by VA.

All other values are reserved.
From Armv8, the only permitted value is 0b0000.

L1HvdFG, bits [3:0]

Level 1 Harvard cache Foreground fetch. Indicates the supported Level 1 cache foreground fetch operations, for a Harvard cache implementation. When supported, foreground fetch operations are blocking operations. Defined values are:

0b0000 Not supported.
0b0001 Supported Level 1 Harvard cache foreground fetch operations are:
  • Fetch instruction cache range by VA.
  • Fetch data cache range by VA.

All other values are reserved.
From Armv8, the only permitted value is 0b0000.

Otherwise:

Bits [63:0]
Reserved, UNKNOWN.

Accessing ID_MMFR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, ID_MMFR2_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_MMFR2_EL1;
    end
elsif PSTATE_EL == EL2 then
    X[t, 64] = ID_MMFR2_EL1;
elsif PSTATE_EL == EL3 then
    X[t, 64] = ID_MMFR2_EL1;
D17.2.84  ID_MMFR3_EL1, AArch32 Memory Model Feature Register 3

The ID_MMFR3_EL1 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register ID_MMFR3_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_MMFR3[31:0].

**Attributes**

ID_MMFR3_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supersec</td>
<td>[31:28]</td>
</tr>
<tr>
<td>CMemSz</td>
<td>[27:24]</td>
</tr>
<tr>
<td>MaintBcst</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**Supersec, bits [31:28]**

Supersections. On a VMSA implementation, indicates whether Supersections are supported. Defined values are:

- 0b0000  Supersections supported.
- 0b1111  Supersections not supported.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b1111.

**CMemSz, bits [27:24]**

Cached Memory Size. Indicates the physical memory size supported by the caches. Defined values are:

- 0b0000  4GB, corresponding to a 32-bit physical address range.
- 0b0001  64GB, corresponding to a 36-bit physical address range.
- 0b0010  1TB or more, corresponding to a 40-bit or larger physical address range.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000, 0b0001, and 0b0010.
CohWalk, bits [23:20]

Coherent Walk. Indicates whether Translation table updates require a clean to the Point of Unification. Defined values are:

0b0000 Updates to the translation tables require a clean to the Point of Unification to ensure visibility by subsequent translation table walks.

0b0001 Updates to the translation tables do not require a clean to the Point of Unification to ensure visibility by subsequent translation table walks.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

PAN, bits [19:16]

Privileged Access Never. Indicates support for the PAN bit in CPSR, SPSR, and DSPSR in AArch32 state. Defined values are:

0b0000 PAN not supported.

0b0001 PAN supported.

0b0010 PAN supported and ATS1CPRP and ATS1CPWP instructions supported.

All other values are reserved.

FEAT_PAN implements the functionality identified by the value 0b0001.

FEAT_PAN2 implements the functionality added by the value 0b0010.

In Armv8.1, the value 0b0000 is not permitted.

From Armv8.2, the only permitted value is 0b0010.

MaintBest, bits [15:12]

Maintenance Broadcast. Indicates whether Cache, TLB, and branch predictor operations are broadcast. Defined values are:

0b0000 Cache, TLB, and branch predictor operations only affect local structures.

0b0001 Cache and branch predictor operations affect structures according to shareability and defined behavior of instructions. TLB operations only affect local structures.

0b0010 Cache, TLB, and branch predictor operations affect structures according to shareability and defined behavior of instructions.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0010.

BPMaint, bits [11:8]

Branch Predictor Maintenance. Indicates the supported branch predictor maintenance operations in an implementation with hierarchical cache maintenance operations. Defined values are:

0b0000 None supported.

0b0001 Supported branch predictor maintenance operations are:

• Invalidate all branch predictors.

0b0010 As for 0b0001, and adds:

• Invalidate branch predictors by VA.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0010.

CMaintSW, bits [7:4]

Cache Maintenance by Set/Way. Indicates the supported cache maintenance operations by set/way, in an implementation with hierarchical caches. Defined values are:

0b0000 None supported.
0b0001  Supported hierarchical cache maintenance instructions by set/way are:
• Invalidate data cache by set/way.
• Clean data cache by set/way.
• Clean and invalidate data cache by set/way.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.
In a unified cache implementation, the data cache maintenance operations apply to the unified caches.

**CMaintVA, bits [3:0]**
Cache Maintenance by Virtual Address. Indicates the supported cache maintenance operations by VA, in an implementation with hierarchical caches. Defined values are:
0b0000  None supported.
0b0001  Supported hierarchical cache maintenance operations by VA are:
• Invalidate data cache by VA.
• Clean data cache by VA.
• Clean and invalidate data cache by VA.
• Invalidate instruction cache by VA.
• Invalidate all instruction cache entries.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.
In a unified cache implementation, data cache maintenance operations apply to the unified caches, and the instruction cache maintenance instructions are not implemented.

**Otherwise:**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, UNKNOWN.</td>
</tr>
</tbody>
</table>

**Accessing ID_MMFR3_EL1**
Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ID_MMFR3_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b111</td>
</tr>
</tbody>
</table>

```
if PSTATE_EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            UNKNOWN
```

D17-6062  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  ARM DDI 0487I.a
Non-Confidential
ID081822
AArch64.SystemAccessTrap(EL1, 0x18);
else
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TID3 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = ID_MMFR3_EL1;
elsif PSTATE.EL == EL2 then
  X[t, 64] = ID_MMFR3_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = ID_MMFR3_EL1;
D17.2.85  ID_MMFR4_EL1, AArch32 Memory Model Feature Register 4

The ID_MMFR4_EL1 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register ID_MMFR4_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_MMFR4[31:0].

**Attributes**

ID_MMFR4_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th>Bits</th>
<th>Purpose Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:32]</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>[31:28]</td>
<td>EVT</td>
</tr>
<tr>
<td>[27:24]</td>
<td>CCIDX</td>
</tr>
<tr>
<td>[23:20]</td>
<td>LSM</td>
</tr>
<tr>
<td>[19:16]</td>
<td>HPDS</td>
</tr>
<tr>
<td>[15:12]</td>
<td>CnP</td>
</tr>
<tr>
<td>[11:8]</td>
<td>XNX</td>
</tr>
<tr>
<td>[7:4]</td>
<td>AC2</td>
</tr>
<tr>
<td>[3:0]</td>
<td>SpecSEI</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**EVT, bits [31:28]**

Enhanced Virtualization Traps. If EL2 is implemented, indicates support for the HCR2.{TTLBIS, TOCU, TICAB, TID4} traps. Defined values are:

- 0b0000  HCR2.{TTLBIS, TOCU, TICAB, TID4} traps are not supported.
- 0b0001  HCR2.{TOCU, TICAB, TID4} traps are supported. HCR2.TTLBIS trap is not supported.
- 0b0010  HCR2.{TTLBIS, TOCU, TICAB, TID4} traps are supported.

All other values are reserved.

**FEAT.EVT** implements the functionality identified by the values 0b0001 and 0b0010.

If EL2 is not implemented supporting AArch32, the only permitted value is 0b0000.

In Armv8.2, the permitted values are 0b0000, 0b0001, and 0b0010.

From Armv8.5, the permitted values are:

- 0b0000 when EL2 is not implemented or does not support AArch32.
- 0b0010 when EL2 is implemented and supports AArch32.

**CCIDX, bits [27:24]**

Support for use of the revised CCSIDR format and the presence of the CCSIDR2 is indicated. Defined values are:

- 0b0000  32-bit format implemented for all levels of the CCSIDR, and the CCSIDR2 register is not implemented.
- 0b0001  64-bit format implemented for all levels of the CCSIDR, and the CCSIDR2 register is implemented.
All other values are reserved.

**FEAT_CCIDX** implements the functionality identified by 0b0001.
From Armv8.3, the permitted values are 0b0000 and 0b0001.

**LSM, bits [23:20]**
Indicates support for LSMAOE and nTLSMD bits in HSCTLR and SCTLR. Defined values are:
0b0000  LSMAOE and nTLSMD bits not supported.
0b0001  LSMAOE and nTLSMD bits supported.
All other values are reserved.
**FEAT_LSMAOC** implements the functionality identified by the value 0b0001.
From Armv8.2, the permitted values are 0b0000 and 0b0001.

**HPDS, bits [19:16]**
Hierarchical permission disables bits in translation tables. Defined values are:
0b0000  Disabling of hierarchical controls not supported.
0b0001  Supports disabling of hierarchical controls using the TTBCR2.HPD0, TTBCR2.HPD1, and HTCR.HPD bits.
0b0010  As for value 0b0001, and adds possible hardware allocation of bits[62:59] of the Translation table descriptors from the final lookup level for IMPLEMENTATION DEFINED use.
All other values are reserved.
**FEAT_AA32HPD** implements the functionality identified by the value 0b0001.
**FEAT_HPDS2** implements the functionality added by the value 0b0010.

--- Note  
The value 0b0000 implies that the encoding for TTBCR2 is UNDEFINED.

**CnP, bits [15:12]**
Common not Private translations. Defined values are:
0b0000  Common not Private translations not supported.
0b0001  Common not Private translations supported.
All other values are reserved.
**FEAT_TTCNP** implements the functionality identified by the value 0b0001.
From Armv8.2 the only permitted value is 0b0001.

**XNX, bits [11:8]**
Support for execute-never control distinction by Exception level at stage 2. Defined values are:
0b0000  Distinction between EL0 and EL1 execute-never control at stage 2 not supported.
0b0001  Distinction between EL0 and EL1 execute-never control at stage 2 supported.
All other values are reserved.
**FEAT_XNX** implements the functionality identified by the value 0b0001.
When **FEAT_XNX** is implemented:
- If all of the following conditions are true, it is IMPLEMENTATION DEFINED whether the value of ID.MMFR4_EL1.XNX is 0b0000 or 0b0001:
  - ID.AA64MMFR1_EL1.XNX == 1.
  - EL2 cannot use AArch32.
  - EL1 can use AArch32.
- If EL2 can use AArch32 then the only permitted value is 0b0001.
AC2, bits [7:4]
Indicates the extension of the ACTLR and HACTLR registers using ACTLR2 and HACTLR2.
Defined values are:
- 0b0000 ACTLR2 and HACTLR2 are not implemented.
- 0b0001 ACTLR2 and HACTLR2 are implemented.
All other values are reserved.
In Armv8.0 and Armv8.1 the permitted values are 0b0000 and 0b0001.
From Armv8.2, the only permitted value is 0b0001.

SpecSEI, bits [3:0]
When FEAT_RAS is implemented:
Describes whether the PE can generate SError interrupt exceptions from speculative reads of memory, including speculative instruction fetches.
- 0b0000 The PE never generates an SError interrupt due to an External abort on a speculative read.
- 0b0001 The PE might generate an SError interrupt due to an External abort on a speculative read.
All other values are reserved.
Otherwise:
Reserved, RES0.

Otherwise:

Bits [63:0]
Reserved, UNKNOWN.

Accessing ID_MMFR4_EL1
Acceses to this register use the following encodings in the System register encoding space:

*MRS <Xt>, ID_MMFR4_EL1*

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_MMFR4_EL1) || boolean...
IMPLEMENTATION_DEFINED "ID_MMFR4_EL1 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
    AArch64.SystemAccessTrap(El2, 0x18);
else
    X[t, 64] = ID_MMFR4_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_MMFR4_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = ID_MMFR4_EL1;
D17.2.86 ID_MMFR5_EL1, AArch32 Memory Model Feature Register 5

The ID_MMFR5_EL1 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers on page D17-5551.

**Configurations**

AArch64 System register ID_MMFR5_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_MMFR5[31:0].

**Attributes**

ID_MMFR5_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:8</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>7:4</td>
<td>nTLBPA, bits [7:4]</td>
</tr>
<tr>
<td>3:0</td>
<td>ETS, bits [3:0]</td>
</tr>
</tbody>
</table>

- **nTLBPA, bits [7:4]**
  - Indicates support for intermediate caching of translation table walks. Defined values are:
    - 0b0000: The intermediate caching of translation table walks might include non-coherent physical translation caches.
    - 0b0001: The intermediate caching of translation table walks does not include non-coherent physical translation caches.

- **ETS, bits [3:0]**
  - Indicates support for Enhanced Translation Synchronization. Defined values are:
    - 0b0000: Enhanced Translation Synchronization is not supported.
    - 0b0001: Enhanced Translation Synchronization is supported.

All other values are reserved.

FEAT_nTLBPA implements the functionality identified by the value 0b0001.

From Armv8.0, the permitted values are 0b0000 and 0b0001.
From Armv8.0, the permitted values are 0b0000 and 0b0001.
From Armv8.7, the only permitted value is 0b0001.

Otherwise:

Bits [63:0]
Reserved, UNKNOWN.

Accessing ID_MMFR5_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS } X[t] \text{, ID_MMFR5_EL1}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0011</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_MMFR5_EL1) || boolean IMPLEMENTATION_DEFINED "ID_MMFR5_EL1 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = ID_MMFR5_EL1;
elsif PSTATE.EL == EL2 then
  X[t, 64] = ID_MMFR5_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = ID_MMFR5_EL1;
D17.2.87 ID_PFR0_EL1, AArch32 Processor Feature Register 0

The ID_PFR0_EL1 characteristics are:

**Purpose**

Gives top-level information about the instruction sets supported by the PE in AArch32 state.
Must be interpreted with ID_PFR1_EL1.
For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register ID_PFR0_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_PFR0[31:0].

**Attributes**

ID_PFR0_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th>Bits [63:32]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS, bits [31:28]</td>
<td>RAS Extension version. Defined values are:</td>
</tr>
<tr>
<td>0b0000</td>
<td>No RAS Extension.</td>
</tr>
<tr>
<td>0b0001</td>
<td>RAS Extension implemented.</td>
</tr>
<tr>
<td>0b0010</td>
<td>FEAT_RASv1p1 implemented. As 0b0001, and adds support for additional ERXMISC&lt;m&gt; System registers.</td>
</tr>
<tr>
<td></td>
<td>Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR&lt;n&gt;STATUS and support for the optional RAS Timestamp Extension.</td>
</tr>
<tr>
<td>All other values are reserved.</td>
<td></td>
</tr>
<tr>
<td>FEAT_RAS implements the functionality identified by the value 0b0001.</td>
<td></td>
</tr>
<tr>
<td>FEAT_RASv1p1 implements the functionality identified by the value 0b0010.</td>
<td></td>
</tr>
<tr>
<td>In Armv8.0 and Armv8.1, the permitted values are 0b0000 and 0b0001.</td>
<td></td>
</tr>
<tr>
<td>In Armv8.2, the only permitted value is 0b0001.</td>
<td></td>
</tr>
<tr>
<td>From Armv8.4, if FEAT_DoubleFault is implemented, the only permitted value is 0b0010.</td>
<td></td>
</tr>
<tr>
<td>From Armv8.4, when FEAT_DoubleFault is not implemented, and ERRIDR_EL1.NUM is 0, the permitted values are IMPLEMENTATION DEFINED 0b0001 or 0b0010.</td>
<td></td>
</tr>
</tbody>
</table>

**Note**

When the value of this field is 0b0001, ID_PFR2_EL1.RAS_frac indicates whether FEAT_RASv1p1 is implemented.
DIT, bits [27:24]

Data Independent Timing. Defined values are:

0b0000  AArch32 does not guarantee constant execution time of any instructions.
0b0001  AArch32 provides the PSTATE.DIT mechanism to guarantee constant execution time of certain instructions.

All other values are reserved.

FEAT_DIT implements the functionality identified by the value 0b0001.

From Armv8.4, the only permitted value is 0b0001.

AMU, bits [23:20]

Indicates support for Activity Monitors Extension. Defined values are:

0b0000  Activity Monitors Extension is not implemented.
0b0001  FEAT_AMUv1 is implemented.
0b0010  FEAT_AMUv1p1 is implemented. As 0b0001 and adds support for virtualization of the activity monitor event counters.

All other values are reserved.

FEAT_AMUv1 implements the functionality identified by the value 0b0001.

FEAT_AMUv1p1 implements the functionality identified by the value 0b0010.

In Armv8.0, the only permitted value is 0b0000.

In Armv8.4, the permitted values are 0b0000 and 0b0001.

From Armv8.6, the permitted values are 0b0000, 0b0001, and 0b0010.

CSV2, bits [19:16]

Speculative use of out of context branch targets. Defined values are:

0b0000  The implementation does not disclose whether FEAT_CSV2 is implemented.
0b0001  FEAT_CSV2 is implemented, but FEAT_CSV2_1p1 is not implemented.
0b0010  FEAT_CSV2_1p1 is implemented.

All other values are reserved.

FEAT_CSV2 implements the functionality identified by the value 0b0001.

FEAT_CSV2_1p1 implements the functionality identified by the value 0b0010.

From Armv8.5, the permitted values are 0b0001 and 0b0010.

State3, bits [15:12]

T32EE instruction set support. Defined values are:

0b0000  Not implemented.
0b0001  T32EE instruction set implemented.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

State2, bits [11:8]

Jazelle extension support. Defined values are:

0b0000  Not implemented.
0b0001  Jazelle extension implemented, without clearing of JOSCR.CV on exception entry.
0b0010  Jazelle extension implemented, with clearing of JOSCR.CV on exception entry.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.
State1, bits [7:4]

T32 instruction set support. Defined values are:

0b0000  T32 instruction set not implemented.
0b0001  T32 encodings before the introduction of Thumb-2 technology implemented:
        •  All instructions are 16-bit.
        •  A BL or BLX is a pair of 16-bit instructions.
        •  32-bit instructions other than BL and BLX cannot be encoded.
0b0011  T32 encodings after the introduction of Thumb-2 technology implemented, for all
        16-bit and 32-bit T32 basic instructions.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0011.

State0, bits [3:0]

A32 instruction set support. Defined values are:

0b0000  A32 instruction set not implemented.
0b0001  A32 instruction set implemented.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

Otherwise:

Bits [63:0]
Reserved, UNKNOWN.

Accessing ID_PFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
\hline
0b11 & 0b000 & 0b0000 & 0b0001 & 0b000 \\
\hline
\end{array}
\]

if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18); \
        else
            AArch64.SystemAccessTrap(EL1, 0x18); \
        end
    else
        UNDEFINED; \
    end
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18); \
    else
        UNDEFINED; \
    end
X[t, 64] = ID_PFR0_EL1;
elsif PSTATE_EL == EL2 then
  X[t, 64] = ID_PFR0_EL1;
elsif PSTATE_EL == EL3 then
  X[t, 64] = ID_PFR0_EL1;
### ID_PFR1_EL1, AArch32 Processor Feature Register 1

The ID_PFR1_EL1 characteristics are:

**Purpose**

Gives information about the AArch32 programmers' model.

Must be interpreted with ID_PFR0_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register ID_PFR1_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_PFR1[31:0].

**Attributes**

ID_PFR1_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>Reserved, RES0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:28</td>
<td>GIC</td>
<td></td>
<td>System register GIC CPU interface. Defined values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0000</td>
<td>GIC CPU interface system registers not implemented.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0001</td>
<td>System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0011</td>
<td>System register interface to version 4.1 of the GIC CPU interface is supported.</td>
</tr>
<tr>
<td>27:24</td>
<td>Virt_frac</td>
<td></td>
<td>Virtualization fractional field. When the Virtualization field is 0b0000, determines the support for Virtualization Extensions. Defined values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0000</td>
<td>No Virtualization Extensions are implemented.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0001</td>
<td>The following Virtualization Extensions are implemented:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The SCR.SIF bit, if EL3 is implemented.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The modifications to the SCR.AW and SCR.FW bits described in the Virtualization Extensions, if EL3 is implemented.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The MSR (banked register) and MRS (banked register) instructions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The ERET instruction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>All other values are reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0000</td>
<td>when EL2 is implemented.</td>
</tr>
</tbody>
</table>
• 0b0001 when EL2 is not implemented.

This field is valid only when the value of ID_PFR1_EL1.Virtualization is 0, otherwise it holds the value 0b0000.

——— Note ————

The ID_ISAR registers do not identify whether the instructions added by the Virtualization Extensions are implemented.

Sec_frac, bits [23:20]

Security fractional field. When the Security field is 0b0000, determines the support for Security Extensions. Defined values are:

0b0000 No Security Extensions are implemented.
0b0001 The following Security Extensions are implemented:
  • The VBAR register.
  • The TTBCR.PD0 and TTBCR.PD1 bits.
0b0010 As for 0b0001, plus the ability to access Secure or Non-secure physical memory is supported.

All other values are reserved.

In Armv8-A, the permitted values are:
• 0b0000 when EL3 is implemented.
• 0b0001 or 0b0010 when EL3 is not implemented.

This field is valid only when the value of ID_PFR1_EL1.Security is 0, otherwise it holds the value 0b0000.

GenTimer, bits [19:16]

Generic Timer support. Defined values are:
0b0000 Generic Timer is not implemented.
0b0001 Generic Timer is implemented.
0b0010 Generic Timer is implemented, and also includes support for CNTHCTL.EVTNTIS and CNTKCTL.EVTNTIS fields, and CNTPCNTSS and CNTVCTSS counter views.

All other values are reserved.

FEAT_ECV implements the functionality identified by the value 0b0010.

In Armv8.0, the only permitted value is 0b0001.
From Armv8.6, the only permitted value is 0b0010.

Virtualization, bits [15:12]

Virtualization support. Defined values are:
0b0000 EL2, Hyp mode, and the HVC instruction not implemented.
0b0001 EL2, Hyp mode, the HVC instruction, and all the features described by Virt_frac == 0b0001 implemented.

All other values are reserved.

In Armv8-A, the permitted values are:
• 0b0000 when EL2 is not implemented.
• 0b0001 when EL2 is implemented.

In an implementation that includes EL2, if EL2 cannot use AArch32 but EL1 can use AArch32 then this field has the value 0b0001.
If EL1 cannot use AArch32 then this field has the value 0b0000.
Note
The ID_ISARs do not identify whether the HVC instruction is implemented.

MProgMod, bits [11:8]
M-profile programmers' model support. Defined values are:
0b0000 Not supported.
0b0010 Support for two-stack programmers' model.
All other values are reserved.
In Armv8-A the only permitted value is 0b0000.

Security, bits [7:4]
Security support. Defined values are:
0b0000 EL3, Monitor mode, and the SMC instruction not implemented.
0b0001 EL3, Monitor mode, the SMC instruction, and all the features described by Sec_frac == 0b0001 implemented.
0b0010 As for 0b0001, and adds the ability to set the NSACR.RFR bit. Not permitted in Armv8 as the NSACR.RFR bit is RES0.
All other values are reserved.
In Armv8-A, the permitted values are:
• 0b0000 when EL3 is not implemented.
• 0b0001 when EL3 is implemented.
In an implementation that includes EL3, if EL3 cannot use AArch32 but EL1 can use AArch32 then this field has the value 0b0001.
If EL1 cannot use AArch32 then this field has the value 0b0000.

ProgMod, bits [3:0]
Support for the standard programmers' model for Armv4 and later. Model must support User, FIQ, IRQ, Supervisor, Abort, Undefined, and System modes. Defined values are:
0b0000 Not supported.
0b0001 Supported.
All other values are reserved.
In Armv8-A, the permitted values are 0b0001 and 0b0000.
If EL1 cannot use AArch32 then this field has the value 0b0000.

Otherwise:

Bits [63:0]
Reserved, UNKNOWN.

Accessing ID_PFR1_EL1
Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, ID_PFR1_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    else
      UNDEFINED;
  elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID3 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      X[t, 64] = ID_PFR1_EL1;
  elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_PFR1_EL1;
  elsif PSTATE.EL == EL3 then
    X[t, 64] = ID_PFR1_EL1;
D17.2.89 ID_PFR2_EL1, AArch32 Processor Feature Register 2

The ID_PFR2_EL1 characteristics are:

**Purpose**

Gives information about the AArch32 programmers' model.

Must be interpreted with ID_PFR0_EL1 and ID_PFR1_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register ID_PFR2_EL1 bits [31:0] are architecturally mapped to AArch32 System register ID_PFR2[31:0].

**Attributes**

ID_PFR2_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field Description</th>
<th>Value</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:12]</td>
<td>Reserved, RES0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[11:8]</td>
<td>RAS_frac</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[7:4]</td>
<td>SSBS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:12]**

Reserved, RES0.

**RAS_frac, bits [11:8]**

RAS Extension fractional field. Defined values are:

- **0b0000** If ID_PFR0_EL1.RAS == 0b0001, RAS Extension implemented.
- **0b0001** If ID_PFR0_EL1.RAS == 0b0001, as 0b0000 and adds support for additional ERXMISC<m> System registers.
  
  Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS and support for the optional RAS Timestamp Extension.

All other values are reserved.

This field is valid only if ID_PFR0_EL1.RAS == 0b0001.

**SSBS, bits [7:4]**

Speculative Store Bypassing controls in AArch64 state. Defined values are:

- **0b0000** AArch32 provides no mechanism to control the use of Speculative Store Bypassing.
- **0b0001** AArch32 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypass Safe.

In Armv8.0, the permitted values are 0b0000 and 0b0001.

From Armv8.5, the only permitted value is 0b0001.

All other values are reserved.
CSV3, bits [3:0]

Speculative use of faulting data. Defined values are:

0b0000  This PE does not disclose whether data loaded under speculation with a permission or domain fault can be used to form an address or generate condition codes or SVE predicate values to be used by other instructions in the speculative sequence.

0b0001  Data loaded under speculation with a permission or domain fault cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence. The execution timing of any other instructions in the speculative sequence is not a function of the data loaded under speculation.

All other values are reserved.

FEAT_CSV3 implements the functionality identified by the value 0b0001.

In Armv8.0, the permitted values are 0b0000 and 0b0001.

From Armv8.5, the only permitted value is 0b0001.

If FEAT_E0PD is implemented, FEAT_CSV3 must be implemented.

Otherwise:

Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_PFR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, ID_PFR2_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0011</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
  elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID3 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      X[t, 64] = ID_PFR2_EL1;
  elseif PSTATE.EL == EL2 then
    X[t, 64] = ID_PFR2_EL1;
elsif PSTATE_EL == EL3 then
X[t, 64] = ID_PPR2_EL1;
D17.2.90 IFSR32_EL2, Instruction Fault Status Register (EL2)

The IFSR32_EL2 characteristics are:

**Purpose**
Allows access to the AArch32 IFSR register from AArch64 state only. Its value has no effect on execution in AArch64 state.

**Configurations**
AArch64 System register IFSR32_EL2 bits [31:0] are architecturally mapped to AArch32 System register IFSR[31:0].
This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to IFSR32_EL2 are UNDEFINED.
If EL2 is not implemented but EL3 is implemented, and EL1 is capable of using AArch32, then this register is not RES0.

**Attributes**
IFSR32_EL2 is a 64-bit register.

**Field descriptions**

*When TTBCR.EAE == 0:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-17</td>
<td>Reserved, RES0</td>
</tr>
</tbody>
</table>
| 16   | FnV, bit [16] FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.  
0b0  IFAR is valid.  
0b1  IFAR is not valid, and holds an UNKNOWN value.  
This field is valid only for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.  
The reset behavior of this field is:  
• On a Warm reset, this field resets to an architecturally UNKNOWN value.  
| 15-13 | Reserved, RES0 |
| 12   | Ext, bit [12] External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.  
In an implementation that does not provide any classification of External aborts, this bit is RES0.  
For aborts other than External aborts this bit always returns 0.  
| 11-0  | Reserved, RES0 |
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally unknown value.

**Bit [11]**

Reserved, RES0.

**FS, bits [10, 3:0]**


- 0b00001: PC alignment fault.
- 0b00010: Debug exception.
- 0b00011: Access flag fault, level 1.
- 0b00101: Translation fault, level 1.
- 0b00110: Access flag fault, level 2.
- 0b00111: Translation fault, level 2.
- 0b01000: Synchronous External abort, not on translation table walk.
- 0b01001: Domain fault, level 1.
- 0b01011: Domain fault, level 2.
- 0b01100: Synchronous External abort, on translation table walk, level 1.
- 0b01101: Permission fault, level 1.
- 0b01110: Synchronous External abort, on translation table walk, level 2.
- 0b01111: Permission fault, level 2.
- 0b10000: TLB conflict abort.
- 0b10100: IMPLEMENTATION DEFINED fault (Lockdown fault).
- 0b11001: When FEAT_RAS is not implemented:
  Synchronous parity or ECC error on memory access, not on translation table walk.
- 0b11100: When FEAT_RAS is not implemented:
  Synchronous parity or ECC error on translation table walk, level 1.
- 0b11110: When FEAT_RAS is not implemented:
  Synchronous parity or ECC error on translation table walk, level 2.

All other values are reserved.

For more information about the lookup level associated with a fault, see The level associated with MMU faults on a Short-descriptor translation table lookup on page G5-9257.

The FS field is split as follows:

- FS[4] is IFSR32_EL2[10].
- FS[3:0] is IFSR32_EL2[3:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally unknown value.

**LPAE, bit [9]**

On taking a Data Abort exception, this bit is set as follows:

- 0b0: Using the Short-descriptor translation table formats.
- 0b1: Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally unknown value.
Bits [8:4]
Reserved, RES0.

When \texttt{TTBCR.EAE} == 1:

Bits [63:17]
Reserved, RES0.

\textbf{FnV, bit [16]}

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

\begin{itemize}
  \item \texttt{0b0} IFAR is valid.
  \item \texttt{0b1} IFAR is not valid, and holds an \texttt{UNKNOWN} value.
\end{itemize}

This field is valid only for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.

The reset behavior of this field is:

\begin{itemize}
  \item On a Warm reset, this field resets to an architecturally \texttt{UNKNOWN} value.
\end{itemize}

Bits [15:13]
Reserved, RES0.

\textbf{ExT, bit [12]}

External abort type. This bit can be used to provide an \texttt{IMPLEMENTATION DEFINED} classification of External aborts.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

The reset behavior of this field is:

\begin{itemize}
  \item On a Warm reset, this field resets to an architecturally \texttt{UNKNOWN} value.
\end{itemize}

Bits [11:10]
Reserved, RES0.

\textbf{LPAE, bit [9]}

On taking a Data Abort exception, this bit is set as follows:

\begin{itemize}
  \item \texttt{0b0} Using the Short-descriptor translation table formats.
  \item \texttt{0b1} Using the Long-descriptor translation table formats.
\end{itemize}

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

The reset behavior of this field is:

\begin{itemize}
  \item On a Warm reset, this field resets to an architecturally \texttt{UNKNOWN} value.
\end{itemize}

Bits [8:6]
Reserved, RES0.
STATUS, bits [5:0]

Fault status bits. Possible values of this field are:

- **0b000000** Address size fault in translation table base register.
- **0b000001** Address size fault, level 1.
- **0b000010** Address size fault, level 2.
- **0b000011** Address size fault, level 3.
- **0b000101** Translation fault, level 1.
- **0b000110** Translation fault, level 2.
- **0b000111** Translation fault, level 3.
- **0b001001** Access flag fault, level 1.
- **0b001010** Access flag fault, level 2.
- **0b001011** Access flag fault, level 3.
- **0b001101** Permission fault, level 1.
- **0b001110** Permission fault, level 2.
- **0b001111** Permission fault, level 3.
- **0b010000** Synchronous External abort, not on translation table walk.
- **0b010101** Synchronous External abort on translation table walk, level 1.
- **0b010110** Synchronous External abort on translation table walk, level 2.
- **0b010111** Synchronous External abort on translation table walk, level 3.
- **0b011000** When FEAT_RAS is not implemented:
  - Synchronous parity or ECC error on memory access, not on translation table walk.
- **0b011101** When FEAT_RAS is not implemented:
  - Synchronous parity or ECC error on memory access on translation table walk, level 1.
- **0b011110** When FEAT_RAS is not implemented:
  - Synchronous parity or ECC error on memory access on translation table walk, level 2.
- **0b011111** When FEAT_RAS is not implemented:
  - Synchronous parity or ECC error on memory access on translation table walk, level 3.
- **0b100001** PC alignment fault.
- **0b100010** Debug exception.
- **0b110000** TLB conflict abort.

All other values are reserved.

When FEAT_RAS is implemented, **0b011000**, **0b011101**, **0b011110**, and **0b011111** are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults on a Long-descriptor translation table lookup* on page G5-9259.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing IFSR32_EL2**

Accesses to this register use the following encodings in the System register encoding space:
**MRS <Xt>, IFSR32_EL2**

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = IFSR32_EL2;
elsif PSTATE.EL == EL3 then
  X[t, 64] = IFSR32_EL2;

**MSR IFSR32_EL2, <Xt>**

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  IFSR32_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
  IFSR32_EL2 = X[t, 64];
D17.2.91 ISR_EL1, Interrupt Status Register

The ISR_EL1 characteristics are:

**Purpose**

Shows the pending status of the IRQ, FIQ, or SError interrupt.

When executing at EL2, EL3 or Secure EL1 when SCR_EL3.EEL2 == 0b0, this shows the pending status of the physical IRQ, FIQ, or SError interrupts.

When executing at either Non-secure EL1 or at Secure EL1 when SCR_EL3.EEL2 == 0b1:

- If the HCR_EL2.{IMO,FMO,AMO} bit has a value of 1, the corresponding ISR_EL1.{I,F,A} bit shows the pending status of the virtual IRQ, FIQ, or SError.
- If the HCR_EL2.{IMO,FMO,AMO} bit has a value of 0, the corresponding ISR_EL1.{I,F,A} bit shows the pending status of the physical IRQ, FIQ, or SError.

**Configurations**

AArch64 System register ISR_EL1 bits [31:0] are architecturally mapped to AArch32 System register ISR[31:0].

**Attributes**

ISR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:11</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>10</td>
<td>IS, bit</td>
</tr>
<tr>
<td>9</td>
<td>FS, bit</td>
</tr>
</tbody>
</table>

**When FEAT_NMI is implemented:**

- ISR_EL1.I bit shows the pending status of the virtualIRQ, FIQ, and SError.
- ISR_EL1.F bit shows the pending status of the virtual IRQ, FIQ, and SError.
- ISR_EL1.A bit shows the pending status of the virtual IRQ, FIQ, and SError.

**Otherwise:**

- Reserved, RES0.
A, bit [8]
SError interrupt pending bit. Indicates whether an SError interrupt is pending.
0b0 No pending SError.
0b1 An SError interrupt is pending.
If the SError interrupt is edge-triggered, this field is cleared to zero when the physical SError interrupt is taken.

I, bit [7]
IRQ pending bit. Indicates whether an IRQ interrupt is pending.
0b0 No pending IRQ.
0b1 An IRQ interrupt is pending.

——— Note ————
This bit indicates the presence of a pending IRQ interrupt regardless of whether the interrupt has Superpriority.

F, bit [6]
FIQ pending bit. Indicates whether an FIQ interrupt is pending.
0b0 No pending FIQ.
0b1 An FIQ interrupt is pending.

——— Note ————
This bit indicates the presence of a pending FIQ interrupt regardless of whether the interrupt has Superpriority.

Bits [5:0]
Reserved, RES0.

Accessing ISR_EL1
Accesses to this register use the following encodings in the System register encoding space:

\[
\begin{array}{cccc}
\text{MRS <Xt>, ISR_EL1} \\
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b000 & 0b1100 & 0b0001 & 0b000 \\
\end{array}
\]

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCRTR_EL2.ISR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = ISR_EL1;
elsif PSTATE.EL == EL2 then
  X[t, 64] = ISR_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = ISR_EL1;
D17.2.92  LORC_EL1, LORegion Control (EL1)

The LORC_EL1 characteristics are:

**Purpose**

Enables and disables LORegions, and selects the current LORegion descriptor.

**Configurations**

This register is present only when FEAT_LOR is implemented. Otherwise, direct accesses to LORC_EL1 are UNDEFINED.

If no LORegion descriptors are supported by the PE, then this register is RES0.

**Attributes**

LORC_EL1 is a 64-bit register.

**Field descriptions**

Bits [63:10]

Reserved, RES0.

DS, bits [9:2]

Descriptor Select. Selects the current LORegion descriptor accessed by LORSA_EL1, LOREA_EL1, and LORN_EL1.

The number of LORegion descriptors in IMPLEMENTATION DEFINED. The maximum number of LORegion descriptors supported is 256. If the number is less than 256, then bits[63:M+2] are RES0, where M is Log2(Number of LORegion descriptors supported by the implementation).

If this field points to an LORegion descriptor that is not supported by an implementation, then the registers LORN_EL1, LOREA_EL1, and LORSA_EL1 are RES0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [1]

Reserved, RES0.

EN, bit [0]

Enable. Indicates whether LORegions are enabled.

0b0  Disabled. Memory accesses do not match any LORegions.

0b1  Enabled. Memory accesses may match a LORegion.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

• On a Warm reset, this field resets to 0.

**Accessing LORC_EL1**

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, LORC_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0100</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsiif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
    UNDEFINED;
elsiif SCR_EL3.NS == '0' then
    UNDEFINED;
elsiif EL2Enabled() && HCR_EL2.TLOR == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
elsiif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.LORC_EL1 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
elsiif HaveEL(EL3) && SCR_EL3.TLOR == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
else
        AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = LORC_EL1;
elsiif PSTATE.EL == EL2 then
  if SCR_EL3.NS == '0' then
    UNDEFINED;
elsiif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
    UNDEFINED;
elsiif HaveEL(EL3) && SCR_EL3.TLOR == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
else
      AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = LORC_EL1;
elsiif PSTATE.EL == EL3 then
  if SCR_EL3.NS == '0' then
    UNDEFINED;
else
  X[t, 64] = LORC_EL1;

MSR LORC_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0100</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsiif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
    UNDEFINED;
elsiif SCR_EL3.NS == '0' then
    UNDEFINED;
elsiif EL2Enabled() && HCR_EL2.TLOR == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
elsiif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.LORC_EL1 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEl(EL3) & SCR_EL3.TLOR == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end;
elsif PSTATE.EL == EL2 then
  if SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & SCR_EL3.TLOR == '1' then
    UNDEFINED;
  endif
elsif PSTATE.EL == EL3 then
  if SCR_EL3.NS == '0' then
    UNDEFINED;
  else
    LORC_EL1 = X[t, 64];
  end.
else
  LORC_EL1 = X[t, 64];
end.
D17.2.93 LOREA_EL1, LORegion End Address (EL1)

The LOREA_EL1 characteristics are:

**Purpose**

Holds the physical address of the end of the LORegion described in the current LORegion descriptor selected by LORC_EL1.DS.

**Configurations**

This register is present only when FEAT_LOR is implemented. Otherwise, direct accesses to LOREA_EL1 are UNDEFINED.

This register is res0 if any of the following apply:

- No LORegion descriptors are supported by the PE.
- LORC_EL1.DS points to a LORegion that is not supported by the PE.

**Attributes**

LOREA_EL1 is a 64-bit register.

**Field descriptions**

Any of the fields in this register are permitted to be cached in a TLB.

**Bits [63:52]**

Reserved, res0.

**EA[51:48], bits [51:48]**

*When FEAT_LPA is implemented:*

Extension to EA[47:16]. For more information, see EA[47:16].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

*Otherwise:*

Reserved, res0.

**EA[47:16], bits [47:16]**

Bits [47:16] of the end physical address of an LORegion described in the current LORegion descriptor selected by LORC_EL1.DS. Bits[15:0] of this address are 0xFFFF. For implementations with fewer than 48 bits, the upper bits of this field are res0.

When FEAT_LPA is implemented and 52-bit addresses are in use, EA[51:48] form bits [51:48] of the end physical address of the LORegion. Otherwise, when 52-bit addresses are not in use, EA[51:48] is res0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Bits [15:0]**

Reserved, res0.
Accessing LOREA_EL1

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, LOREA_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
    UNDEFINED;
  elsif SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.TLOR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCRTR_EL2.LOREA_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.TLOR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  end
else
  X[t, 64] = LOREA_EL1;
elsif PSTATE.EL == EL2 then
  if SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
    UNDEFINED;
  elsif Halted() && SCR_EL3.TLOR == '1' then
    if SCR_EL3.FGTEn == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  end
else
  if SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif SCR_EL3.TLOR == '1' then
    if EDSCR.SDD == '1' then
      UNDEFINED;
    else
      X[t, 64] = LOREA_EL1;
    end
  elsif PSTATE.EL == EL3 then
    if SCR_EL3.NS == '0' then
      UNDEFINED;
    else
      X[t, 64] = LOREA_EL1;
    end
  end

**MSR LOREA_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
    UNDEFINED;
elsif SCR_EL3.NS == '0' then
    UNDEFINED;
elsif EL2Enabled() && SCR_EL3.TLOR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.LOREA_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.TLOR == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        LOREA_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.TLOR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            LOREA_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    if SCR_EL3.NS == '0' then
        UNDEFINED;
else
    LOREA_EL1 = X[t, 64];
D17.2.94  LORID_EL1, LORegionID (EL1)

The LORID_EL1 characteristics are:

**Purpose**

Indicates the number of LORegions and LORegion descriptors supported by the PE.

**Configurations**

This register is present only when FEAT_LOR is implemented. Otherwise, direct accesses to LORID_EL1 are UNDEFINED.

If no LORegion descriptors are implemented, then the registers LORC_EL1, LORN_EL1, LOREA_EL1, and LORSA_EL1 are RES0.

**Attributes**

LORID_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:24]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[23:16]</td>
<td>LD, Number of LORegion descriptors supported by the PE. This is an 8-bit binary number.</td>
</tr>
<tr>
<td>[15:8]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[7:0]</td>
<td>LR, Number of LORegions supported by the PE. This is an 8-bit binary number.</td>
</tr>
</tbody>
</table>

**Note**

If LORID_EL1 indicates that no LORegions are implemented, then LoadLOAcquire and StoreLORelease will behave as LoadAcquire and StoreRelease.

**Accessing LORID_EL1**

Accesses to this register use the following encodings in the System register encoding space:

\[ MRS <Xt>, LORID_EL1 \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0100</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then

if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
  UNDEFINED;
elsif EL2Enabled() && HCR_EL2.TLOR == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.LORID_EL1 == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.TLOR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = LORID_EL1;
else
  X[t, 64] = LORID_EL1;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
    UNDEFINED;
elsif HaveEL(EL3) && SCR_EL3.TLOR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = LORID_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = LORID_EL1;
D17.2.95  **LORN_EL1, LORegion Number (EL1)**

The LORN_EL1 characteristics are:

**Purpose**

Holds the number of the LORegion described in the current LORegion descriptor selected by LORC_EL1.DS.

**Configurations**

This register is present only when FEAT_LOR is implemented. Otherwise, direct accesses to LORN_EL1 are **UNDEFINED**.

This register is **RES0** if any of the following apply:

- No LORegion descriptors are supported by the PE.
- LORC_EL1.DS points to a LORegion that is not supported by the PE.

**Attributes**

LORN_EL1 is a 64-bit register.

**Field descriptions**

Any of the fields in this register are permitted to be cached in a TLB.

**Bits [63:8]**

Reserved, **RES0**.

**Num, bits [7:0]**

Number of the LORegion described in the current LORegion descriptor selected by LORC_EL1.DS.

The maximum number of LORegions supported by the PE is 256. If the maximum number is less than 256, then bits[8:N] are **RES0**, where N is (Log2(Number of LORegions supported by the PE)).

If this field points to a LORegion that is not supported by the PE, then the current LORegion descriptor does not match any LORegion.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Accessing LORN_EL1**

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, LORN_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0100</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
    UNDEFINED;
  elsif SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.TLOR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.LORN_EL1 == '1' then
    UNundefined;
  elsif HaveEL(EL3) && SCR_EL3.TLOR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = LORN_EL1;
  end
elsif PSTATE_EL == EL2 then
  if SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
    UNDEFINED;
  elsif SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif EL2Enabled() && SCR_EL3.TLOR == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
 elsif HaveEL(EL3) && SCR_EL3.TLOR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = LORN_EL1;
  end
elsif PSTATE_EL == EL3 then
  if SCR_EL3.NS == '0' then
    UNDEFINED;
  else
    X[t, 64] = LORN_EL1;
  end

MSR LORN_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0100</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
    UNDEFINED;
  elsif SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.TLOR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.LORN_EL1 == '1' then
    UNDEFINED;
AArch64.SystemAccessTrap(EL2, 0x18);
elif HaveEl(EL3) && SCR_EL3.TLOR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    LORN_EL1 = X[t, 64];
  elsif PSTATE.EL == EL2 then
    if SCR_EL3.NS == '0' then
      UNDEFINED;
    elsif Halted() && HaveEl(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
      UNDEFINED;
    elsif HaveEl(EL3) && SCR_EL3.TLOR == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      else
        LORN_EL1 = X[t, 64];
    elsif PSTATE.EL == EL3 then
      if SCR_EL3.NS == '0' then
        UNDEFINED;
      else
        LORN_EL1 = X[t, 64];
    else
      LORN_EL1 = X[t, 64];
D17.2.96  LORSA_EL1, LORegion Start Address (EL1)

The LORSA_EL1 characteristics are:

**Purpose**

Indicates whether the current LORegion descriptor selected by LORC_EL1.DS is enabled, and holds the physical address of the start of the LORegion.

**Configurations**

This register is present only when FEAT_LOR is implemented. Otherwise, direct accesses to LORSA_EL1 are UNDEFINED.

This register is RES0 if any of the following apply:
- No LORegion descriptors are supported by the PE.
- LORC_EL1.DS points to a LORegion that is not supported by the PE.

**Attributes**

LORSA_EL1 is a 64-bit register.

**Field descriptions**

Any of the fields in this register are permitted to be cached in a TLB.

**Bits [63:52]**

Reserved, RES0.

**SA, bits [51:16]**

*SA encoding when FEAT_LPA is implemented*

**SA, bits [35:0]**

Bits [51:16] of the start physical address of the LORegion described in the current LORegion descriptor selected by LORC_EL1.DS.

Bits[15:0] of this address are 0x0000.

For implementations with fewer than 52 physical address bits, the corresponding upper bits of this field are RES0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
SA encoding when FEAT_LPA is not implemented

Bits [35:32]
  Reserved, RES0.

SA, bits [31:0]
  Bits [47:16] of the start physical address of the LORegion described in the current LORegion descriptor selected by LORC_EL1.DS.
  Bits[15:0] of this address are 0x0000.
  For implementations with fewer than 48 physical address bits, the corresponding upper bits of this field are RES0.
  The reset behavior of this field is:
    • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:1]
  Reserved, RES0.

Valid, bit [0]
  Indicates whether the current LORegion descriptor is enabled.
  0b0   LORegion descriptor is disabled.
  0b1   LORegion descriptor is enabled.
  The reset behavior of this field is:
    • On a Warm reset, this field resets to 0.

Accessing LORSA_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, LORSA_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
    UNDEFINED;
  elsif SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.TLOR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCR_TR.LORSA_EL1 == '1') then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.TLOR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      RES0
AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = LORSA_EL1;
elsif PSTATE.EL == EL2 then
  if SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.TLOR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = LORSA_EL1;
  end
elsif PSTATE.EL == EL3 then
  if SCR_EL3.NS == '0' then
    UNDEFINED;
  else
    X[t, 64] = LORSA_EL1;
else
  X[t, 64] = LORSA_EL1;
end

MSR LORSA_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
    UNDEFINED;
  elsif SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.TLOR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.LORSA_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.TLOR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    LORSA_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL2 then
  if SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TLOR == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.TLOR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    LORSA_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL3 then
  if SCR_EL3.NS == '0' then
    UNDEFINED;
else
    LORSA_EL1 = X[t, 64];
D17.2.97 MAIR_EL1, Memory Attribute Indirection Register (EL1)

The MAIR_EL1 characteristics are:

**Purpose**

Provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations at EL1.

**Configurations**

AArch64 System register MAIR_EL1 bits [31:0] are architecturally mapped to AArch32 System register PRRR[31:0] when TTBCR.EAE == 0.
AArch64 System register MAIR_EL1 bits [31:0] are architecturally mapped to AArch32 System register MAIR0[31:0] when TTBCR.EAE == 1.
AArch64 System register MAIR_EL1 bits [63:32] are architecturally mapped to AArch32 System register NMRR[31:0] when TTBCR.EAE == 0.
AArch64 System register MAIR_EL1 bits [63:32] are architecturally mapped to AArch32 System register MAIR1[31:0] when TTBCR.EAE == 1.

**Attributes**

MAIR_EL1 is a 64-bit register.

**Field descriptions**

MAIR_EL1 is permitted to be cached in a TLB.

**Attr<n>, bits [8n+7:8n], for n = 7 to 0**


Attr is encoded as follows:

<table>
<thead>
<tr>
<th>Attr</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000dd00</td>
<td>Device memory. See encoding of 'dd' for the type of Device memory.</td>
</tr>
<tr>
<td>0b0000dd01</td>
<td>If FEAT_XS is implemented: Device memory with the XS attribute set to 0.</td>
</tr>
<tr>
<td></td>
<td>See encoding of 'dd' for the type of Device memory.</td>
</tr>
<tr>
<td></td>
<td>Otherwise, UNPREDICTABLE.</td>
</tr>
<tr>
<td>0b0000dd1x</td>
<td>UNPREDICTABLE.</td>
</tr>
<tr>
<td>0booo0000 and iii!=0000</td>
<td>Normal memory. See encoding of 'oooo' and 'iiii' for the type of Normal</td>
</tr>
<tr>
<td></td>
<td>Memory.</td>
</tr>
<tr>
<td>0b01000000</td>
<td>If FEAT_XS is implemented: Normal Inner Non-cacheable, Outer Non-</td>
</tr>
<tr>
<td></td>
<td>cacheable memory with the XS attribute set to 0.</td>
</tr>
<tr>
<td></td>
<td>Otherwise, UNPREDICTABLE.</td>
</tr>
</tbody>
</table>
## AArch64 System Register Descriptions

### D17.2 General system control registers

<table>
<thead>
<tr>
<th>Attr</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10100000</td>
<td>If FEAT_XS is implemented: Normal Inner Write-through Cacheable, Outer Write-through Cacheable, Read-Allocate, No-Write Allocate, Non-transient memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.</td>
</tr>
<tr>
<td>0b11110000</td>
<td>If FEAT_MTE2 is implemented: Tagged Normal Inner Write-Back, Outer Write-Back, Read-Allocate, Write-Allocate Non-transient memory. Otherwise, UNPREDICTABLE.</td>
</tr>
<tr>
<td>0bxxxx0000</td>
<td>(xxxx != 0000, xxxx != 0100, xxxx != 1010, xxxx != 1111) UNPREDICTABLE.</td>
</tr>
</tbody>
</table>

'dd' is encoded as follows:

<table>
<thead>
<tr>
<th>dd</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>Device-nGnRnE memory</td>
</tr>
<tr>
<td>0b0001</td>
<td>Device-nGnRE memory</td>
</tr>
<tr>
<td>0b1000</td>
<td>Device-nGRE memory</td>
</tr>
<tr>
<td>0b1001</td>
<td>Device-GRE memory</td>
</tr>
</tbody>
</table>

'oooo' is encoded as follows:

<table>
<thead>
<tr>
<th>'oooo'</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>See encoding of Attr</td>
</tr>
<tr>
<td>0b0001</td>
<td>Normal memory, Outer Write-Through Transient</td>
</tr>
<tr>
<td>0b0100</td>
<td>Normal memory, Outer Non-cacheable</td>
</tr>
<tr>
<td>0b0101</td>
<td>Normal memory, Outer Write-Back Transient</td>
</tr>
<tr>
<td>0b1000</td>
<td>Normal memory, Outer Write-Through Non-transient</td>
</tr>
<tr>
<td>0b1001</td>
<td>Normal memory, Outer Write-Back Non-transient</td>
</tr>
</tbody>
</table>

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

'iiii' is encoded as follows:

<table>
<thead>
<tr>
<th>'iiii'</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>See encoding of Attr</td>
</tr>
<tr>
<td>0b0001</td>
<td>Normal memory, Inner Write-Through Transient</td>
</tr>
<tr>
<td>0b0100</td>
<td>Normal memory, Inner Non-cacheable</td>
</tr>
<tr>
<td>0b0101</td>
<td>Normal memory, Inner Write-Back Transient</td>
</tr>
<tr>
<td>0b1000</td>
<td>Normal memory, Inner Write-Through Non-transient</td>
</tr>
<tr>
<td>0b1001</td>
<td>Normal memory, Inner Write-Back Non-transient</td>
</tr>
</tbody>
</table>
R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.
The R and W bits in 'oooo' and 'iiii' fields have the following meanings:

<table>
<thead>
<tr>
<th>R or W</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>No Allocate</td>
</tr>
<tr>
<td>0b1</td>
<td>Allocate</td>
</tr>
</tbody>
</table>

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MAIR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic MAIR_EL1 or MAIR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, MAIR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TRVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.MAIR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x140];
  else
    X[t, 64] = MAIR_EL1;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = MAIR_EL2;
  else
    X[t, 64] = MAIR_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = MAIR_EL1;
else
  X[t, 64] = MAIR_EL1;
```

**MSR MAIR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.MAIR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x140];
  else
    X[t, 64] = MAIR_EL1;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = MAIR_EL2;
  else
    X[t, 64] = MAIR_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = MAIR_EL1;
else
  X[t, 64] = MAIR_EL1;
```
AArch64 System Register Descriptions

D17.2 General system control registers

AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEl(EL3) || SCR_EL3.FGTEn == '1') && HFWTR_EL2.MAIR_EL1 == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
  NVMem[0x140] = X[t, 64];
else
  MAIR_EL1 = X[t, 64];
elif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    MAIR_EL2 = X[t, 64];
  else
    MAIR_EL1 = X[t, 64];
elifs PSTATE.EL == EL3 then
  MAIR_EL1 = X[t, 64];

MRS <xt>, MAIR_EL12

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elifs PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    X[t, 64] = NVMem[0x140];
elifs EL2Enabled() && HCR_EL2.NV == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elifs PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = MAIR_EL1;
elifs PSTATE.EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    X[t, 64] = MAIR_EL1;
elifs PSTATE.EL == EL0 then
  UNDEFINED;
elifs PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    NVMem[0x140] = X[t, 64];
elifs EL2Enabled() && HCR_EL2.NV == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elifs PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    MAIR_EL1 = X[t, 64];
elifs PSTATE.EL == EL3 then
  if EL2Enabled() && HCR_EL2.E2H == '1' then
    MAIR_EL1 = X[t, 64];
elifs PSTATE.EL == EL0 then
  UNDEFINED;
elifs PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    NVMem[0x140] = X[t, 64];
elifs EL2Enabled() && HCR_EL2.NV == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elifs PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    MAIR_EL1 = X[t, 64];
elifs PSTATE.EL == EL3 then
  if EL2Enabled() && HCR_EL2.E2H == '1' then
    MAIR_EL1 = X[t, 64];

MSR MAIR_EL12, <xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>
UNDEFINED;
elsif PSTATE_EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    MAIR_EL1 = X[t, 64];
  else
    UNDEFINED;
D17.2.98 MAIR_EL2, Memory Attribute Indirection Register (EL2)

The MAIR_EL2 characteristics are:

**Purpose**

Provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations at EL2.

**Configurations**

AArch64 System register MAIR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HMAIR0[31:0].

AArch64 System register MAIR_EL2 bits [63:32] are architecturally mapped to AArch32 System register HMAIR1[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

MAIR_EL2 is a 64-bit register.

**Field descriptions**

MAIR_EL2 is permitted to be cached in a TLB.

**Attr<n>, bits [8n+7:8n], for n = 7 to 0**


Attr is encoded as follows:

<table>
<thead>
<tr>
<th>Attr</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000dd00</td>
<td>Device memory. See encoding of 'dd' for the type of Device memory.</td>
</tr>
<tr>
<td>0b0000dd01</td>
<td>If FEAT_XS is implemented: Device memory with the XS attribute set to 0. See encoding of 'dd' for the type of Device memory. Otherwise, UNPREDICTABLE.</td>
</tr>
<tr>
<td>0b0000dd1x</td>
<td>UNPREDICTABLE.</td>
</tr>
<tr>
<td>0b00000000</td>
<td>Normal memory. See encoding of 'oooo' and 'iiii' for the type of Normal Memory.</td>
</tr>
<tr>
<td>0b01000000</td>
<td>If FEAT_XS is implemented: Normal Inner Non-cacheable, Outer Non-cacheable memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.</td>
</tr>
</tbody>
</table>
'dd' is encoded as follows:

<table>
<thead>
<tr>
<th>Attr</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10100000</td>
<td>If FEAT_XS is implemented: Normal Inner Write-through Cacheable, Outer Write-through Cacheable, Read-Allocate, No-Write Allocate, Non-transient memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.</td>
</tr>
<tr>
<td>0b11110000</td>
<td>If FEAT_MTE2 is implemented: Tagged Normal Inner Write-Back, Outer Write-Back, Read-Allocate, Write-Allocate Non-transient memory. Otherwise, UNPREDICTABLE.</td>
</tr>
<tr>
<td>0bxxxx0000, (xxxx != 0000, xxxx != 0100, xxxx != 1010, xxxx != 1111)</td>
<td>UNPREDICTABLE.</td>
</tr>
</tbody>
</table>

'o000' is encoded as follows:

<table>
<thead>
<tr>
<th>'o000'</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>See encoding of Attr</td>
</tr>
<tr>
<td>0b00RW, RW not 0b00</td>
<td>Normal memory, Outer Write-Through Transient</td>
</tr>
<tr>
<td>0b0100</td>
<td>Normal memory, Outer Non-cacheable</td>
</tr>
<tr>
<td>0b01RW, RW not 0b00</td>
<td>Normal memory, Outer Write-Back Transient</td>
</tr>
<tr>
<td>0b10RW</td>
<td>Normal memory, Outer Write-Through Non-transient</td>
</tr>
<tr>
<td>0b11RW</td>
<td>Normal memory, Outer Write-Back Non-transient</td>
</tr>
</tbody>
</table>

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

'iii' is encoded as follows:

<table>
<thead>
<tr>
<th>'iii'</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>See encoding of Attr</td>
</tr>
<tr>
<td>0b00RW, RW not 0b00</td>
<td>Normal memory, Inner Write-Through Transient</td>
</tr>
<tr>
<td>0b0100</td>
<td>Normal memory, Inner Non-cacheable</td>
</tr>
<tr>
<td>0b01RW, RW not 0b00</td>
<td>Normal memory, Inner Write-Back Transient</td>
</tr>
<tr>
<td>0b10RW</td>
<td>Normal memory, Inner Write-Through Non-transient</td>
</tr>
<tr>
<td>0b11RW</td>
<td>Normal memory, Inner Write-Back Non-transient</td>
</tr>
</tbody>
</table>
R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in 'oooo' and 'iii' fields have the following meanings:

<table>
<thead>
<tr>
<th>R or W</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>No Allocate</td>
</tr>
<tr>
<td>0b1</td>
<td>Allocate</td>
</tr>
</tbody>
</table>

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing MAIR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic MAIR_EL2 or MAIR_EL1 is not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, MAIR_EL2**

```plaintext
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = MAIR_EL2;
elsif PSTATE.EL == EL3 then
  X[t, 64] = MAIR_EL2;
```

**MSR MAIR_EL2, <Xt>**

```plaintext
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  MAIR_EL2 = X[t, 64];
```
elsif PSTATE.EL == EL3
    MAIR_EL2 = X[t, 64];

MRS <Xt>, MAIR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') &
                HFCRTR_EL2.MAIR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x140];
    else
        X[t, 64] = MAIR_EL1;
    end;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        MAIR_EL2 = X[t, 64];
    else
        MAIR_EL1 = X[t, 64];
    end;
elsif PSTATE.EL == EL3 then
    MAIR_EL1 = X[t, 64];

MSR MAIR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') &
                HFCRTR_EL2.MAIR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x140] = X[t, 64];
    else
        MAIR_EL1 = X[t, 64];
    end;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        MAIR_EL2 = X[t, 64];
    else
        MAIR_EL1 = X[t, 64];
    end;
elsif PSTATE.EL == EL3 then
    MAIR_EL1 = X[t, 64];
D17.2.99 MAIR_EL3, Memory Attribute Indirection Register (EL3)

The MAIR_EL3 characteristics are:

**Purpose**

Provides the memory attribute encodings corresponding to the possible AttrIndx values in a
Long-descriptor format translation table entry for stage 1 translations at EL3.

**Configurations**

This register is present only when EL3 is implemented. Otherwise, direct accesses to MAIR_EL3
are UNDEFINED.

**Attributes**

MAIR_EL3 is a 64-bit register.

**Field descriptions**

MAIR_EL3 is permitted to be cached in a TLB.

**Attr<n>, bits [8n+7:8n], for n = 7 to 0**

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation
table entry, where AttrIndx[2:0] gives the value of <n> in Attr<\n>.

Attr is encoded as follows:

<table>
<thead>
<tr>
<th>Attr</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000dd00</td>
<td>Device memory. See encoding of 'dd' for the type of Device memory.</td>
</tr>
<tr>
<td>0b0000dd01</td>
<td>If FEAT_XS is implemented: Device memory with the XS attribute set to 0. See encoding of 'dd' for the type of Device memory. Otherwise, UNPREDICTABLE.</td>
</tr>
<tr>
<td>0b0000dd1x</td>
<td>UNPREDICTABLE.</td>
</tr>
<tr>
<td>0b00000000</td>
<td>Normal memory. See encoding of 'oooo' and 'iiii' for the type of Normal Memory.</td>
</tr>
<tr>
<td>0b10100000</td>
<td>If FEAT_XS is implemented: Normal Inner Write-back, Outer Non-transient memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.</td>
</tr>
<tr>
<td>0b11110000</td>
<td>If FEAT_MTE2 is implemented: Tagged Normal Inner Write-Back, Outer Non-transient memory. Otherwise, UNPREDICTABLE.</td>
</tr>
<tr>
<td>0bxxxx8000, (xxxx != 0000, xxxx != 0100, xxxx != 1010, xxxx != 1111)</td>
<td>UNPREDICTABLE.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attr</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0bxxxx8000, (xxxx != 0000, xxxx != 0100, xxxx != 1010, xxxx != 1111)</td>
<td>UNPREDICTABLE.</td>
</tr>
</tbody>
</table>
'dd' is encoded as follows:

<table>
<thead>
<tr>
<th>dd</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>Device-nGnRnE memory</td>
</tr>
<tr>
<td>0b01</td>
<td>Device-nGnRE memory</td>
</tr>
<tr>
<td>0b10</td>
<td>Device-nGRE memory</td>
</tr>
<tr>
<td>0b11</td>
<td>Device-GRE memory</td>
</tr>
</tbody>
</table>

'o000' is encoded as follows:

<table>
<thead>
<tr>
<th>'o000'</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>See encoding of Attr</td>
</tr>
<tr>
<td>0b00RW</td>
<td>Normal memory, Outer Write-Through Transient</td>
</tr>
<tr>
<td>0b0100</td>
<td>Normal memory, Outer Non-cacheable</td>
</tr>
<tr>
<td>0b01RW</td>
<td>Normal memory, Outer Write-Back Transient</td>
</tr>
<tr>
<td>0b10RW</td>
<td>Normal memory, Outer Write-Through Non-transient</td>
</tr>
<tr>
<td>0b11RW</td>
<td>Normal memory, Outer Write-Back Non-transient</td>
</tr>
</tbody>
</table>

'R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

'iii' is encoded as follows:

<table>
<thead>
<tr>
<th>'iii'</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>See encoding of Attr</td>
</tr>
<tr>
<td>0b00RW</td>
<td>Normal memory, Inner Write-Through Transient</td>
</tr>
<tr>
<td>0b0100</td>
<td>Normal memory, Inner Non-cacheable</td>
</tr>
<tr>
<td>0b01RW</td>
<td>Normal memory, Inner Write-Back Transient</td>
</tr>
<tr>
<td>0b10RW</td>
<td>Normal memory, Inner Write-Through Non-transient</td>
</tr>
<tr>
<td>0b11RW</td>
<td>Normal memory, Inner Write-Back Non-transient</td>
</tr>
</tbody>
</table>

'R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in 'o000' and 'iii' fields have the following meanings:

<table>
<thead>
<tr>
<th>R or W</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>No Allocate</td>
</tr>
<tr>
<td>0b1</td>
<td>Allocate</td>
</tr>
</tbody>
</table>

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Accessing MAIR_EL3

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, MAIR_EL3**

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>
```

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  X[t, 64] = MAIR_EL3;

**MSR MAIR_EL3, <Xt>**

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>
```

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  MAIR_EL3 = X[t, 64];
D17.2.100 MIDR_EL1, Main ID Register

The MIDR_EL1 characteristics are:

**Purpose**

Provides identification information for the PE, including an implementer code for the device and a device ID number.

**Configurations**

AArch64 System register MIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register MIDR[31:0].

AArch64 System register MIDR_EL1 bits [31:0] are architecturally mapped to External register MIDR_EL1[31:0].

**Attributes**

MIDR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:32]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
</table>

**Implementer, bits [31:24]**

The Implementer code. This field must hold an implementer code that has been assigned by Arm. Assigned codes include the following:

- 0x00: Reserved for software use.
- 0x41: Arm Limited.
- 0x42: Broadcom Corporation.
- 0x43: Cavium Inc.
- 0x44: Digital Equipment Corporation.
- 0x46: Fujitsu Ltd.
- 0x49: Infineon Technologies AG.
- 0x40: Motorola or Freescale Semiconductor Inc.
- 0x4E: NVIDIA Corporation.
- 0x50: Applied Micro Circuits Corporation.
- 0x51: Qualcomm Inc.
- 0x56: Marvell International Ltd.
- 0x69: Intel Corporation.
- 0xC0: Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.
Variant, bits [23:20]

Variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Architecture, bits [19:16]

Architecture version. Defined values are:

- 0b0001  Armv4.
- 0b0010  Armv4T.
- 0b0011  Armv5 (obsolete).
- 0b0100  Armv5T.
- 0b0101  Armv5TE.
- 0b0110  Armv5TEJ.
- 0b0111  Armv6.
- 0b1111  Architectural features are individually identified in the ID_* registers.

All other values are reserved.

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

PartNum, bits [15:4]

Primary Part Number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Revision, bits [3:0]

Revision number for the device.

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Accessing MIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS} \ <Xt>, \ \text{MIDR}\_\text{EL1}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        end if
    else
        UNDEFINED;
    end if
else if PSTATE.EL == EL1 then
if EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCRTR_EL2.MIDR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() then
    X[t, 64] = VPIDR_EL2;
else
    X[t, 64] = MIDR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MIDR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = MIDR_EL1;
### D17.2.101 MPIDR_EL1, Multiprocessor Affinity Register

The MPIDR_EL1 characteristics are:

**Purpose**

In a multiprocessor system, provides an additional PE identification mechanism for scheduling purposes.

**Configurations**

AArch64 System register MPIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register MPIDR[31:0].

In a uniprocessor system, Arm recommends that each Aff<n> field of this register returns a value of 0.

**Attributes**

MPIDR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-40</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>31</td>
<td>Reserved, RES1.</td>
</tr>
<tr>
<td>30</td>
<td>U, bit [30]</td>
</tr>
<tr>
<td>29-25</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>24</td>
<td>MT, bit [24]</td>
</tr>
</tbody>
</table>

- **Aff3, bits [39:32]**
  
  Affinity level 3. See the description of Aff0 for more information.
  
  Aff3 is not supported in AArch32 state.

- **Bit [31]**
  
  Reserved, RES1.

- **U, bit [30]**
  
  Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.
  
  0b0 Processor is part of a multiprocessor system.
  
  0b1 Processor is part of a uniprocessor system.

- **Bits [29:25]**
  
  Reserved, RES0.

- **MT, bit [24]**
  
  Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multithreading type approach. See the description of Aff0 for more information about affinity levels.
  
  0b0 Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
  
  0b1 Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.
Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

Aff0, bits [7:0]

Affinity level 0. This is the affinity level that is most significant for determining PE behavior. Higher affinity levels are increasingly less significant in determining PE behavior. The assigned value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

Accessing MPIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[ MRS <X>, MPIDR_EL1 \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        end
    else
        UNDEFINED;
    end
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.MPIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() then
        X[t, 64] = VMPIDR_EL2;
    else
        X[t, 64] = MPIDR_EL1;
    end
elsif PSTATE_EL == EL2 then
    X[t, 64] = MPIDR_EL1;
elsif PSTATE_EL == EL3 then
    X[t, 64] = MPIDR_EL1;
```


D17.2.102 MVFR0_EL1, AArch32 Media and VFP Feature Register 0

The MVFR0_EL1 characteristics are:

**Purpose**

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

Must be interpreted with MVFR1_EL1 and MVFR2_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register MVFR0_EL1 bits [31:0] are architecturally mapped to AArch32 System register MVFR0[31:0].

In an implementation where at least one Exception level supports execution in AArch32 state, but there is no support for Advanced SIMD and floating-point operation, this register is RAZ.

**Attributes**

MVFR0_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

![Register Diagram]

**Bits [63:32]**

Reserved, RES0.

**FPRound, bits [31:28]**

Floating-Point Rounding modes. Indicates whether the floating-point implementation provides support for rounding modes. Defined values are:

- **0b0000**: Not implemented, or only Round to Nearest mode supported, except that Round towards Zero mode is supported for VCVT instructions that always use that rounding mode regardless of the FPSCR setting.
- **0b0001**: All rounding modes supported.

All other values are reserved.

In Armv8-A the permitted values are **0b0000** and **0b0001**.

**FPShVec, bits [27:24]**

Short Vectors. Indicates whether the floating-point implementation provides support for the use of short vectors. Defined values are:

- **0b0000**: Short vectors not supported.
- **0b0001**: Short vector operation supported.

All other values are reserved.

In Armv8-A the only permitted value is **0b0000**.
FPSqrt, bits [23:20]
Square Root. Indicates whether the floating-point implementation provides support for the ARMv6 VFP square root operations. Defined values are:
0b0000 Not supported in hardware.
0b0001 Supported.
All other values are reserved.
In Armv8-A the permitted values are 0b0000 and 0b0001.
The VSQRT.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VSQRT.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

FPDivide, bits [19:16]
Indicates whether the floating-point implementation provides support for VFP divide operations. Defined values are:
0b0000 Not supported in hardware.
0b0001 Supported.
All other values are reserved.
In Armv8-A the permitted values are 0b0000 and 0b0001.
The VDIV.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VDIV.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

FPTrap, bits [15:12]
Floating Point Exception Trapping. Indicates whether the floating-point implementation provides support for exception trapping. Defined values are:
0b0000 Not supported.
0b0001 Supported.
All other values are reserved.
A value of 0b0001 indicates that, when the corresponding trap is enabled, a floating-point exception generates an exception.

FPDP, bits [11:8]
Double Precision. Indicates whether the floating-point implementation provides support for double-precision operations. Defined values are:
0b0000 Not supported in hardware.
0b0001 Supported, VFPv2.
0b0010 Supported, VFPv3, VFPv4, or Armv8. VFPv3 and Armv8 add an instruction to load a double-precision floating-point constant, and conversions between double-precision and fixed-point values.
All other values are reserved.
In Armv8-A the permitted values are 0b0000 and 0b0010.
A value of 0b0001 or 0b0010 indicates support for all VFP double-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:
• VSQRT.F64 is only available if the Square root field is 0b0001.
• VDIV.F64 is only available if the Divide field is 0b0001.
• Conversion between double-precision and single-precision is only available if the single-precision field is nonzero.
FPSP, bits [7:4]

Single Precision. Indicates whether the floating-point implementation provides support for single-precision operations. Defined values are:

- 0b0000: Not supported in hardware.
- 0b0001: Supported, VFPv2.
- 0b0010: Supported, VFPv3 or VFPv4. VFPv3 adds an instruction to load a single-precision floating-point constant, and conversions between single-precision and fixed-point values.

All other values are reserved.

In Armv8-A the permitted values are 0b0000 and 0b0010.

A value of 0b0001 or 0b0010 indicates support for all VFP single-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- VSQRT.F32 is only available if the Square root field is 0b0001.
- VDIV.F32 is only available if the Divide field is 0b0001.
- Conversion between double-precision and single-precision is only available if the double-precision field is nonzero.

SIMDReg, bits [3:0]

Advanced SIMD registers. Indicates whether the Advanced SIMD and floating-point implementation provides support for the Advanced SIMD and floating-point register bank. Defined values are:

- 0b0000: The implementation has no Advanced SIMD and floating-point support.
- 0b0001: The implementation includes floating-point support with 16 x 64-bit registers.
- 0b0010: The implementation includes Advanced SIMD and floating-point support with 32 x 64-bit registers.

All other values are reserved.

In Armv8-A the permitted values are 0b0000 and 0b0010.

Otherwise:

Bits [63:0]

Reserved, UNKNOWN.

Accessing MVFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, MVFR0_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if IsFeatureImplemented(Feat_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            UNDEFINED;
        end if;
    else
        UNDEFINED;
    end if;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t], 64 = MVFR0_EL1;
    end if;
elsif PSTATE.EL == EL2 then
    X[t], 64 = MVFR0_EL1;
elsif PSTATE.EL == EL3 then
    X[t], 64 = MVFR0_EL1;
D17.2.103 MVFR1_EL1, AArch32 Media and VFP Feature Register 1

The MVFR1_EL1 characteristics are:

**Purpose**

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

Must be interpreted with MVFR0_EL1 and MVFR2_EL1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

AArch64 System register MVFR1_EL1 bits [31:0] are architecturally mapped to AArch32 System register MVFR1[31:0].

In an implementation where at least one Exception level supports execution in AArch32 state, but there is no support for Advanced SIMD and floating-point operation, this register is RAZ.

**Attributes**

MVFR1_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
<th>Values</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:32]</td>
<td>Reserved, RES0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[31:28]</td>
<td>SIMDFMAC, Advanced SIMD Fused MAC</td>
<td>0b0000 Not implemented</td>
<td>In Armv8-A, the permitted values are 0b0000 and 0b0001.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0001 Implemented</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other values are reserved</td>
<td></td>
</tr>
<tr>
<td>[27:24]</td>
<td>FPHP, Floating Point Half Precision</td>
<td>0b0000 Not supported</td>
<td>The Advanced SIMD and floating-point implementations must provide the same level of support for these instructions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0001 Floating-point half-precision conversion instructions are supported for conversion between single-precision and half-precision.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0010 As for 0b0001, and adds instructions for conversion between double-precision and half-precision.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0011 As for 0b0010, and adds support for half-precision floating-point arithmetic.</td>
<td></td>
</tr>
</tbody>
</table>

RES0

SIMDFMAC

FPHP

SIMDHP

SIMDSP

SIMDInt

SIMDLS

FPDNaN

FPFtZ
All other values are reserved.

In Armv8-A, the permitted values are:

- 0b0000 in an implementation without floating-point support.
- 0b0001 in an implementation with floating-point support that does not include the FEAT_FP16 extension.
- 0b0011 in an implementation with floating-point support that includes the FEAT_FP16 extension.

The level of support indicated by this field must be equivalent to the level of support indicated by the SIMDHP field, meaning the permitted values are:

<table>
<thead>
<tr>
<th>Half Precision instructions supported</th>
<th>FPHP</th>
<th>SIMDHP</th>
</tr>
</thead>
<tbody>
<tr>
<td>No support</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
<tr>
<td>Conversions only</td>
<td>0b0010</td>
<td>0b0001</td>
</tr>
<tr>
<td>Conversions and arithmetic</td>
<td>0b0011</td>
<td>0b0010</td>
</tr>
</tbody>
</table>

**SIMDHP, bits [23:20]**

Advanced SIMD Half Precision. Indicates the level of half-precision floating-point support. Defined values are:

- 0b0000 Not supported.
- 0b0001 SIMD half-precision conversion instructions are supported for conversion between single-precision and half-precision.
- 0b0010 As for 0b0001, and adds support for half-precision floating-point arithmetic.

All other values are reserved.

In Armv8-A, the permitted values are:

- 0b0000 in an implementation without SIMD floating-point support.
- 0b0001 in an implementation with SIMD floating-point support that does not include the FEAT_FP16 extension.
- 0b0010 in an implementation with SIMD floating-point support that includes the FEAT_FP16 extension.

The level of support indicated by this field must be equivalent to the level of support indicated by the FPHP field, meaning the permitted values are:

<table>
<thead>
<tr>
<th>Half Precision instructions supported</th>
<th>FPHP</th>
<th>SIMDHP</th>
</tr>
</thead>
<tbody>
<tr>
<td>No support</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
<tr>
<td>Conversions only</td>
<td>0b0010</td>
<td>0b0001</td>
</tr>
<tr>
<td>Conversions and arithmetic</td>
<td>0b0011</td>
<td>0b0010</td>
</tr>
</tbody>
</table>

**SIMDSP, bits [19:16]**

Advanced SIMD Single Precision. Indicates whether the Advanced SIMD and floating-point implementation provides single-precision floating-point instructions. Defined values are:

- 0b0000 Not implemented.
- 0b0001 Implemented. This value is permitted only if the SIMDInt field is 0b0001.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0001.
SIMDInt, bits [15:12]
Advanced SIMD Integer. Indicates whether the Advanced SIMD and floating-point implementation provides integer instructions. Defined values are:

0b0000  Not implemented.
0b0001  Implemented.
All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0001.

SIMDLS, bits [11:8]
Advanced SIMD Load/Store. Indicates whether the Advanced SIMD and floating-point implementation provides load/store instructions. Defined values are:

0b0000  Not implemented.
0b0001  Implemented.
All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0001.

FPDNaN, bits [7:4]
Default NaN mode. Indicates whether the floating-point implementation provides support only for the Default NaN mode. Defined values are:

0b0000  Not implemented, or hardware supports only the Default NaN mode.
0b0001  Hardware supports propagation of NaN values.
All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0001.

FPFtZ, bits [3:0]
Flush to Zero mode. Indicates whether the floating-point implementation provides support only for the Flush-to-Zero mode of operation. Defined values are:

0b0000  Not implemented, or hardware supports only the Flush-to-Zero mode of operation.
0b0001  Hardware supports full denormalized number arithmetic.
All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0001.

Otherwise:

Bits [63:0]
Reserved, UNKNOWN.

Accessing MVFR1_EL1
Accesses to this register use the following encodings in the System register encoding space:
MRS <xt>, MVFR1_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    else
      UNDEFINED;
  elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID3 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      X[t, 64] = MVFR1_EL1;
  elsif PSTATE.EL == EL2 then
    X[t, 64] = MVFR1_EL1;
  elsif PSTATE.EL == EL3 then
    X[t, 64] = MVFR1_EL1;
D17.2.104 MVFR2_EL1, AArch32 Media and VFP Feature Register 2

The MVFR2_EL1 characteristics are:

**Purpose**

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

Must be interpreted with MVFR0_EL1 and MVFR1_EL1.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers on page D17-5551.

**Configurations**

AArch64 System register MVFR2_EL1 bits [31:0] are architecturally mapped to AArch32 System register MVFR2[31:0].

In an implementation where at least one Exception level supports execution in AArch32 state, but there is no support for Advanced SIMD and floating-point operation, this register is RAZ.

**Attributes**

MVFR2_EL1 is a 64-bit register.

**Field descriptions**

*When AArch32 is supported:*

```
<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>RES0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>8</td>
</tr>
<tr>
<td>RES0</td>
<td>FPMisc</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>SIMDMisc</td>
</tr>
</tbody>
</table>
```

**Bits [63:8]**

Reserved, RES0.

**FPMisc, bits [7:4]**

Indicates whether the floating-point implementation provides support for miscellaneous VFP features.

- 0b0000: Not implemented, or no support for miscellaneous features.
- 0b0001: Support for Floating-point selection.
- 0b0010: As 0b0001, and Floating-point Conversion to Integer with Directed Rounding modes.
- 0b0011: As 0b0010, and Floating-point Round to Integer Floating-point.
- 0b0100: As 0b0011, and Floating-point MaxNum and MinNum.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0100.

**SIMDMisc, bits [3:0]**

Indicates whether the Advanced SIMD implementation provides support for miscellaneous Advanced SIMD features.

- 0b0000: Not implemented, or no support for miscellaneous features.
- 0b0001: Floating-point Conversion to Integer with Directed Rounding modes.
- 0b0010: As 0b0001, and Floating-point Round to Integer Floating-point.
- 0b0011: As 0b0010, and Floating-point MaxNum and MinNum.
All other values are reserved.
In Armv8-A, the permitted values are \texttt{0b0000} and \texttt{0b0011}.

\textit{Otherwise}:

\begin{verbatim}
<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{UNKNOWN}</td>
<td>\texttt{UNKNOWN}</td>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{Bits} [63:0]
Reserved, \texttt{UNKNOWN}.

\textbf{Accessing MVFR2\_EL1}

Accesses to this register use the following encodings in the System register encoding space:

\textbf{MRS <\texttt{Xt}, MVFR2\_EL1}

\begin{verbatim}
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{0b11}</td>
<td>\texttt{0b000}</td>
<td>\texttt{0b0000}</td>
<td>\texttt{0b0011}</td>
<td>\texttt{0b010}</td>
</tr>
</tbody>
</table>
\end{verbatim}

if PSTATE.EL == EL0 then
    if IsFeatureImplemented(\texttt{FEAT\_IDST}) then
        if EL2Enabled() \&\& HCR\_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() \&\& HCR\_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = MVFR2\_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MVFR2\_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = MVFR2\_EL1;
D17.2.105 PAR_EL1, Physical Address Register

The PAR_EL1 characteristics are:

**Purpose**

Returns the output address (OA) from an Address translation instruction that executed successfully, or fault information if the instruction did not execute successfully.

**Configurations**

AArch64 System register PAR_EL1 bits [63:0] are architecturally mapped to AArch32 System register PAR[63:0].

**Attributes**

PAR_EL1 is a 64-bit register.

**Field descriptions**

*When PAR_EL1.F == 0:*

This section describes the register value returned by the successful execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

On a successful conversion, the PAR_EL1 can return a value that indicates the resulting attributes, rather than the values that appear in the Translation table descriptors. More precisely:

- The PAR_EL1.[ATTR, SH] fields are permitted to report the resulting attributes, as determined by any permitted implementation choices and any applicable configuration bits, instead of reporting the values that appear in the Translation table descriptors.
- See the PAR_EL1.NS bit description for constraints on the value it returns.

**ATTR, bits [63:56]**

Memory attributes for the returned output address. This field uses the same encoding as the Attr<\text{n}> fields in MAIR_EL1, MAIR_EL2, and MAIR_EL3.

The value returned in this field can be the resulting attribute that is actually implemented by the implementation, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

---

**Note**

The attributes presented are consistent with the stages of translation applied in the address translation instruction. If the instruction performed a stage 1 translation only, the attributes are from the stage 1 translation. If the instruction performed a stage 1 and stage 2 translation, the attributes are from the combined stage 1 and stage 2 translation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Bits [55:52]
Reserved, RES0.

PA[51:48], bits [51:48]

When FEAT_LPA is implemented:
Extension to PA[47:12]. For more information, see PA[47:12].
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

PA[47:12], bits [47:12]
Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits[47:12].
When FEAT_LPA is implemented and 52-bit addresses are in use, PA[51:48] forms the upper part of the address value. Otherwise, when 52-bit addresses are not in use, PA[51:48] is RES0.
For implementations with fewer than 48 physical address bits, the corresponding upper bits in this field are RES0.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSE, bit [11]

When FEAT_RME is implemented:
Reports the NSE attribute for a translation table entry from the EL3 translation regime.
For a description of the values derived by evaluating NS and NSE together, see PAR_EL1.NS.
For a result from a Secure, Non-secure, or Realm translation regime, this bit is UNKNOWN.

Otherwise:
Reserved, RES1.

IMPLEMENTATION DEFINED, bit [10]

IMPLEMENTATION DEFINED.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NS, bit [9]

When FEAT_RME is implemented:
Non-secure. The NS attribute for a translation table entry from a Secure translation regime, a Realm translation regime, and the EL3 translation regime.
For a result from an EL3 translation regime, NS and NSE are evaluated together to report the physical address space:

<table>
<thead>
<tr>
<th>NSE</th>
<th>NS</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>Secure.</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>Non-secure.</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>Root.</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>Realm.</td>
</tr>
</tbody>
</table>
For a result from a Secure translation regime, when SCR_EL3.EEL2 is 1, this bit reflects the Security state of the intermediate physical address space of the translation for the instructions:

- In AArch64 state: AT S1E1R, AT S1E1W, AT S1E1RP, AT S1E1WP, AT S1E0R, and AT S1E0W.
- In AArch32 state: ATS1CPR, ATS1CPW, ATS1CPRP, ATS1CPWP, ATS1CUR, and ATS1CUW.

Otherwise, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

For a result from an S1E1 or S1E0 operation on the Realm EL1&0 translation regime, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Non-secure. The NS attribute for a translation table entry from a Secure translation regime.

For a result from a Secure translation regime, when SCR_EL3.EEL2 is 1, this bit reflects the Security state of the intermediate physical address space of the translation for the instructions:

- In AArch64 state: AT S1E1R, AT S1E1W, AT S1E1RP, AT S1E1WP, AT S1E0R, and AT S1E0W.
- In AArch32 state: ATS1CPR, ATS1CPW, ATS1CPRP, ATS1CPWP, ATS1CUR, and ATS1CUW.

Otherwise, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**SH, bits [8:7]**

Shareability attribute, for the returned output address.

- **0b00** Non-shareable.
- **0b10** Outer Shareable.
- **0b11** Inner Shareable.

The value 0b01 is reserved.

**Note**

This field returns the value 0b10 for:

- Any type of Device memory.
- Normal memory with both Inner Non-cacheable and Outer Non-cacheable attributes.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [6:1]**

Reserved, RES0.
F, bit [0]
Indicates whether the instruction performed a successful address translation.

0b0 Address translation completed successfully.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When PAR_EL1.F == 1:

This section describes the register value returned by a fault on the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

IMPLEMENTATION DEFINED, bits [63:56]
IMPLEMENTATION DEFINED.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [55:52]
IMPLEMENTATION DEFINED.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [51:48]
IMPLEMENTATION DEFINED.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [47:12]
Reserved, RES0.

Bit [11]
Reserved, RES1.

Bit [10]
Reserved, RES0.

S, bit [9]
Indicates the translation stage at which the translation aborted:

0b0 Translation aborted because of a fault in the stage 1 translation.
0b1 Translation aborted because of a fault in the stage 2 translation.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
PTW, bit [8]

If this bit is set to 1, it indicates the translation aborted because of a stage 2 fault during a stage 1 translation table walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

FST, bits [6:1]

Fault status code, as shown in the Data Abort exception ESR encoding.

- 0b000000  Address size fault, level 0 of translation or translation table base register.
- 0b000001  Address size fault, level 1.
- 0b000010  Address size fault, level 2.
- 0b000011  Address size fault, level 3.
- 0b000100  Translation fault, level 0.
- 0b000101  Translation fault, level 1.
- 0b000110  Translation fault, level 2.
- 0b000111  Translation fault, level 3.
- 0b001000  Access flag fault, level 0.
- 0b001001  Access flag fault, level 1.
- 0b001010  Access flag fault, level 2.
- 0b001011  Access flag fault, level 3.
- 0b001100  When FEAT_LPA2 is implemented: Access flag fault, level 0.
- 0b001101  When FEAT_LPA2 is implemented: Permission fault, level 0.
- 0b001110  Permission fault, level 2.
- 0b001111  Permission fault, level 3.
- 0b010000  When FEAT_LPA2 is implemented: Synchronous External abort on translation table walk or hardware update of translation table, level -1.
- 0b010001  Synchronous External abort on translation table walk or hardware update of translation table, level 0.
- 0b010010  Synchronous External abort on translation table walk or hardware update of translation table, level 1.
- 0b010011  Synchronous External abort on translation table walk or hardware update of translation table, level 2.
- 0b010100  Synchronous External abort on translation table walk or hardware update of translation table, level 3.
- 0b010101  Synchronous External abort on translation table walk or hardware update of translation table, level -1.
- 0b010110  Synchronous External abort on translation table walk or hardware update of translation table, level 0.
- 0b010111  Synchronous External abort on translation table walk or hardware update of translation table, level 1.
- 0b011000  When FEAT_LPA2 is implemented and FEAT_RAS is not implemented: Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.
- 0b011001  When FEAT_RAS is not implemented: Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.
- 0b011010  When FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.

0b011110  *When FEAT_RAS is not implemented:*  
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.

0b011111  *When FEAT_RAS is not implemented:*  
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.

0b100011  *When FEAT_RME is implemented and FEAT_LPA2 is implemented:*  
Granule Protection Fault on translation table walk or hardware update of translation table, level -1.

0b100100  *When FEAT_RME is implemented:*  
Granule Protection Fault on translation table walk or hardware update of translation table, level 0.

0b100101  *When FEAT_RME is implemented:*  
Granule Protection Fault on translation table walk or hardware update of translation table, level 1.

0b100110  *When FEAT_RME is implemented:*  
Granule Protection Fault on translation table walk or hardware update of translation table, level 2.

0b100111  *When FEAT_RME is implemented:*  
Granule Protection Fault on translation table walk or hardware update of translation table, level 3.

0b101000  *When FEAT_RME is implemented:*  
Granule Protection Fault, not on translation table walk or hardware update of translation table.

0b101001  *When FEAT_LPA2 is implemented:*  
Address size fault, level -1.

0b101011  *When FEAT_LPA2 is implemented:*  
Translation fault, level -1.

0b110000  TLB conflict abort.

0b110001  *When FEAT_HAFDBS is implemented:*  
Unsupported atomic hardware update fault.

0b111101  *When EL1 is capable of using AArch32:*  
Section Domain fault, from an AArch32 stage 1 EL1&0 translation regime using Short-descriptor translation table format.

0b111110  *When EL1 is capable of using AArch32:*  
Page Domain fault, from an AArch32 stage 1 EL1&0 translation regime using Short-descriptor translation table format.

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**F, bit [0]**

Indicates whether the instruction performed a successful address translation.

0b1  Address translation aborted.

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
Accessing PAR_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[
MRS \langle X_t \rangle, \text{PAR\_EL1}
\]

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0\text{b11} & 0\text{b000} & 0\text{b011} & 0\text{b0100} & 0\text{b000}
\end{array}
\]

if PSTATE\_EL == EL0 then
  UNDEFINED;
elsif PSTATE\_EL == EL1 then
  if EL2Enabled() && (!HaveEL(EL3) || SCR\_EL3\_FGTEn == '1') && HFGRTR\_EL2\_PAR\_EL1 == '1' then
    AArch64\_SystemAccessTrap(EL2, 0x18);
  else
    \( X[t, 64] = \text{PAR\_EL1} \);
elsif PSTATE\_EL == EL2 then
  \( X[t, 64] = \text{PAR\_EL1} \);
elsif PSTATE\_EL == EL3 then
  \( X[t, 64] = \text{PAR\_EL1} \);

\[
MSR \text{PAR\_EL1}, \langle X_t \rangle
\]

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0\text{b11} & 0\text{b000} & 0\text{b011} & 0\text{b0100} & 0\text{b000}
\end{array}
\]

if PSTATE\_EL == EL0 then
  UNDEFINED;
elsif PSTATE\_EL == EL1 then
  if EL2Enabled() && (!HaveEL(EL3) || SCR\_EL3\_FGTEn == '1') && HFGWTR\_EL2\_PAR\_EL1 == '1' then
    AArch64\_SystemAccessTrap(EL2, 0x18);
  else
    \( \text{PAR\_EL1} = X[t, 64] \);
elsif PSTATE\_EL == EL2 then
  \( \text{PAR\_EL1} = X[t, 64] \);
elsif PSTATE\_EL == EL3 then
  \( \text{PAR\_EL1} = X[t, 64] \);
D17.2.106 REVIDR_EL1, Revision ID Register

The REVIDR_EL1 characteristics are:

**Purpose**
Provides implementation-specific minor revision information.

**Configurations**
AArch64 System register REVIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register REVIDR[31:0].

If REVIDR_EL1 has the same value as MIDR_EL1, then its contents have no significance.

**Attributes**
REVIDR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>31-0</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

**Accessing REVIDR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, REVIDR_EL1**

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b110</td>
</tr>
</tbody>
</table>
```

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TID1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.REVIDR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = REVIDR_EL1;
elsif PSTATE.EL == EL2 then
  X[t, 64] = REVIDR_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = REVIDR_EL1;
D17.2.107  **RGSR_EL1, Random Allocation Tag Seed Register.**

The RGSR_EL1 characteristics are:

**Purpose**

Random Allocation Tag Seed Register.

**Configurations**

This register is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to RGSR_EL1 are UNDEFINED.

When **GCR_EL1.RRND==0b1**, updates to RGSR_EL1 are implementation-specific.

**Attributes**

RGSR_EL1 is a 64-bit register.

**Field descriptions**

![Field descriptions diagram]

Bits [63:24]  
Reserved, RES0.

**SEED, bits [23:8]**
Seed register used for generating values returned by RandomAllocationTag().

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [7:4]  
Reserved, RES0.

**TAG, bits [3:0]**
Tag generated by the most recent IRG instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing RGSR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, RGSR_EL1**

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b000 & 0b0001 & 0b0000 & 0b101
\end{array}
\]

if PSTATE.EL == EL0 then

UNDEFINED;

elsif PSTATE.EL == EL1 then

if Halted() && HaveEl(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1' && SCR_EL3.ATA == '0' then
UNDEFINED;
elsif EL2Enabled() && HCR_EL2.ATA == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.ATA == '0' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = RGSR_EL1;
elsif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1' && SCR_EL3.ATA == '0' then
UNDEFINED;
elsif HaveEL(EL3) && SCR_EL3.ATA == '0' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = RGSR_EL1;
elsif PSTATE.EL == EL3 then
X[t, 64] = RGSR_EL1;
else
RGSR_EL1 = X[t, 64];
elif PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1' && SCR_EL3.ATA == '0' then
UNDEFINED;
elsif EL2Enabled() && HCR_EL2.ATA == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.ATA == '0' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
RGSR_EL1 = X[t, 64];
elif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1' && SCR_EL3.ATA == '0' then
UNDEFINED;
elsif HaveEL(EL3) && SCR_EL3.ATA == '0' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
RGSR_EL1 = X[t, 64];
elif PSTATE.EL == EL3 then
RGSR_EL1 = X[t, 64];

**MSR RGSR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1' && SCR_EL3.ATA == '0' then
UNDEFINED;
elsif EL2Enabled() && HCR_EL2.ATA == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.ATA == '0' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
RGSR_EL1 = X[t, 64];
elif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1' && SCR_EL3.ATA == '0' then
UNDEFINED;
elsif HaveEL(EL3) && SCR_EL3.ATA == '0' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
RGSR_EL1 = X[t, 64];
elif PSTATE.EL == EL3 then
RGSR_EL1 = X[t, 64];
D17.2.108 RMR_EL1, Reset Management Register (EL1)

The RMR_EL1 characteristics are:

**Purpose**
When this register is implemented:
- A write to the register at EL1 can request a Warm reset.
- If EL1 can use all Execution states, this register specifies the Execution state that the PE boots into on a Warm reset.

**Configurations**
AArch64 System register RMR_EL1 bits [31:0] are architecturally mapped to AArch32 System register RMR[31:0] when the highest implemented Exception level is EL1.

This register is present only when the highest implemented Exception level is EL1. Otherwise, direct accesses to RMR_EL1 are UNDEFINED.

When EL1 is the highest implemented Exception level:
- If EL1 can use all Execution states then this register must be implemented.
- If EL1 cannot use AArch32 then it is IMPLEMENTATION DEFINED whether the register is implemented.

**Attributes**
RMR_EL1 is a 64-bit register.

**Field descriptions**

![Field diagram]

**Bits [63:2]**
Reserved, RES0.

**RR, bit [1]**
Reset Request. Setting this bit to 1 requests a Warm reset.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**AA64, bit [0]**

*When EL1 is capable of using AArch32:*
- When EL1 can use AArch32, determines which Execution state the PE boots into after a Warm reset:
  - 0b0   AArch32.
  - 0b1   AArch64.

  On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.
- If EL1 can only use AArch64 state, this bit is RAO/WI.

  The reset behavior of this field is:
  - When implemented as a RW field, this field resets to 1 on a Cold reset.
Otherwise:
Reserved, RAO/WI.

Accessing RMR_EL1

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, RMR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL1 && IsHighestEL(EL1) then
  X[t, 64] = RMR_EL1;
else
  UNDEFINED;

**MSR RMR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL1 && IsHighestEL(EL1) then
  RMR_EL1 = X[t, 64];
else
  UNDEFINED;
D17.2.109 RMR_EL2, Reset Management Register (EL2)

The RMR_EL2 characteristics are:

**Purpose**

When this register is implemented:
- A write to the register at EL2 can request a Warm reset.
- If EL2 can use all Execution states, this register specifies the Execution state that the PE boots into on a Warm reset.

**Configurations**

AArch64 System register RMR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HRMR[31:0] when the highest implemented Exception level is EL2.

This register is present only when the highest implemented Exception level is EL2. Otherwise, direct accesses to RMR_EL2 are **UNDEFINED**.

When EL2 is the highest implemented Exception level:
- If EL2 can use all Execution states then this register must be implemented.
- If EL2 cannot use AArch32 then it is **IMPLEMENTATION DEFINED** whether the register is implemented.

**Attributes**

RMR_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:2</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>31:0</td>
<td>Reserved, RES0</td>
</tr>
</tbody>
</table>
| 1       | Reset Request. Setting this bit to 1 requests a Warm reset.  
|         | The reset behavior of this field is:  
|         | - On a Warm reset, this field resets to 0. |
| 0       | When EL2 is capable of using AArch32:  
|         | When EL2 can use AArch32, determines which Execution state the PE boots into after a Warm reset:  
|         | 0b0   AArch32.  
|         | 0b1   AArch64.  
|         | On coming out of the Warm reset, execution starts at the **IMPLEMENTATION DEFINED** reset vector address of the specified Execution state.  
|         | If EL2 can only use AArch64 state, this bit is RAO/WI.  
|         | The reset behavior of this field is:  
|         | - When implemented as a RW field, this field resets to 1 on a Cold reset. |
Otherwise:
Reserved, RAO/WI.

Accessing RMR_EL2

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, RMR_EL2**

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>
```

if PSTATE_EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && HCR_EL2.NV == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsiif PSTATE_EL == EL2 && IsHighestEL(EL2) then
  X[t, 64] = RMR_EL2;
else
  UNDEFINED;

**MSR RMR_EL2, <Xt>**

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>
```

if PSTATE_EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && HCR_EL2.NV == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsiif PSTATE_EL == EL2 && IsHighestEL(EL2) then
  RMR_EL2 = X[t, 64];
else
  UNDEFINED;
D17.2.110  RMR_EL3, Reset Management Register (EL3)

The RMR_EL3 characteristics are:

**Purpose**

If EL3 is the implemented and this register is implemented:

- A write to the register at EL3 can request a Warm reset.
- If EL3 can use all Execution states, this register specifies the Execution state that the PE boots into on a Warm reset.

**Configurations**

AArch64 System register RMR_EL3 bits [31:0] are architecturally mapped to AArch32 System register RMR[31:0] when EL3 is implemented.

This register is present only when EL3 is implemented. Otherwise, direct accesses to RMR_EL3 are UNDEFINED.

When EL3 is implemented:

- If EL3 can use all Execution states then this register must be implemented.
- If EL3 cannot use AArch32, then it is IMPLEMENTATION DEFINED whether the register is implemented.

Otherwise, direct accesses to RMR_EL3 are UNDEFINED.

**Attributes**

RMR_EL3 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:2]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[1]</td>
<td>RR, bit</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>[0]</td>
<td>AA64, bit</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The reset behavior of this field is:
- When implemented as a RW field, this field resets to 1 on a Cold reset.

**Otherwise:**
Reserved, RAO/W1.

**Accessing RMR_EL3**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, RMR_EL3**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL3 & IsHighestEL(EL3) then
X[t, 64] = RMR_EL3;
else
UNDEFINED;

**MSR RMR_EL3, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL3 & IsHighestEL(EL3) then
RMR_EL3 = X[t, 64];
else
UNDEFINED;
D17.2.111   RNDR, Random Number

The RNDR characteristics are:

**Purpose**

Random Number. Returns a 64-bit random number which is reseeded from the True Random Number source at an IMPLEMENTATION DEFINED rate.

If the hardware returns a genuine random number, PSTATE.NZCV is set to 0b0000.

If the instruction cannot return a genuine random number in a reasonable period of time, PSTATE.NZCV is set to 0b0100 and the data value returned is 0.

**Configurations**

This register is present only when FEAT_RNG is implemented or FEAT_RNG_TRAP is implemented. Otherwise, direct accesses to RNDR are UNDEFINED.

**Attributes**

RNDR is a 64-bit register.

**Field descriptions**

RNDR, bits [63:0]

Random Number. Returns a 64-bit Random Number which is reseeded from the True Random Number source at an IMPLEMENTATION DEFINED rate.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing RNDR**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, RNDR**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0010</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3.TRNDR == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  elsif !IsFeatureImplemented(FEAT_RNG) then
    UNDEFINED;
  else
    X[t, 64] = RNDR;
  endif
else
  if PSTATE_EL == EL1 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3.TRNDR == '1' then
      AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !IsFeatureImplemented(FEAT_RNG) then
      UNDEFINED;
    else
      RNDR
    endif
  endif
endif
X[t, 64] = RNDR;

elsif PSTATE_EL == EL2 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3.TRNDR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !IsFeatureImplemented(FEAT_RNG) then
        UNDEFINED;
    else
        X[t, 64] = RNDR;
    endif
else
    X[t, 64] = RNDR;
endif

elsif PSTATE_EL == EL3 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3.TRNDR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !IsFeatureImplemented(FEAT_RNG) then
        UNDEFINED;
    else
        X[t, 64] = RNDR;
    endif
else
    X[t, 64] = RNDR;
ENDIF
D17.2.112 RNDRRS, Reseeded Random Number

The RNDRRS characteristics are:

**Purpose**

Reseeded Random Number. Returns a 64-bit random number which is reseeded from the True Random Number source immediately before the read of the random number.

If the hardware returns a genuine random number, PSTATE.NZCV is set to 0b0000.

If the instruction cannot return a genuine random number in a reasonable period of time, PSTATE.NZCV is set to 0b0100 and the data value returned is 0.

When FEAT_RNG_TRAP is implemented and SCR_EL3.TRNDR is 1, reads of this register are trapped to EL3.

**Configurations**

This register is present only when FEAT_RNG is implemented or FEAT_RNG_TRAP is implemented. Otherwise, direct accesses to RNDRRS are UNDEFINED.

**Attributes**

RNDRRS is a 64-bit register.

**Field descriptions**

RNDRRS, bits [63:0]

Reseeded Random Number. Returns a 64-bit Random Number which is reseeded from the True Random Number source immediately before this read.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing RNDRRS**

Accesses to this register use the following encodings in the System register encoding space:

### MRS <Xt>, RNDRRS

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0010</td>
<td>0b0100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3.TRNDR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elseif !IsFeatureImplemented(FEAT_RNG) then
        UNDEFINED;
    else
        X[t, 64] = RNDRRS;
    endif
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3.TRNDR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    endif
elsif !IsFeatureImplemented(FEAT_RNG) then
    UNDEFINED;
else
    X[t, 64] = RNDRRS;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3.TRNDR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        if IsFeatureImplemented(FEAT_RNG) then
            UNDEFINED;
        else
            X[t, 64] = RNDRRS;
    endif
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3.TRNDR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        if IsFeatureImplemented(FEAT_RNG) then
            UNDEFINED;
        else
            X[t, 64] = RNDRRS;
    endif
D17.2.113 RVBAR_EL1, Reset Vector Base Address Register (if EL2 and EL3 not implemented)

The RVBAR_EL1 characteristics are:

**Purpose**

If EL1 is the highest Exception level implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch64 state.

**Configurations**

This register is present only when the highest implemented Exception level is EL1. Otherwise, direct accesses to RVBAR_EL1 are UNDEFINED.

**Attributes**

RVBAR_EL1 is a 64-bit register.

**Field descriptions**

![ResetAddress diagram]

**ResetAddress, bits [63:0]**

The IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 64-bit state. Bits[1:0] of this register are 00, as this address must be aligned, and the address must be within the physical address size supported by the PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Accessing RVBAR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, RVBAR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL1 && IsHighestEL(EL1) then
  X[t, 64] = RVBAR_EL1;
else
  UNDEFINED;
D17.2.114 RVBAR_EL2, Reset Vector Base Address Register (if EL3 not implemented)

The RVBAR_EL2 characteristics are:

**Purpose**

If EL2 is the highest Exception level implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch64 state.

**Configurations**

This register is present only when the highest implemented Exception level is EL2. Otherwise, direct accesses to RVBAR_EL2 are UNDEFINED.

**Attributes**

RVBAR_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResetAddress, bits [63:0]</td>
<td>The IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 64-bit state. Bits[1:0] of this register are 00, as this address must be aligned, and the address must be within the physical address size supported by the PE. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.</td>
</tr>
</tbody>
</table>

**Accessing RVBAR_EL2**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, RVBAR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && HCR_EL2.NV == '1' then
    AArch64_SystemAccessTrap(El2, 0x18);
elsif PSTATE_EL == EL2 && IsHighestEL(EL2) then
    X[t, 64] = RVBAR_EL2;
else
    UNDEFINED;
D17.2.115  RVBAR_EL3, Reset Vector Base Address Register (if EL3 implemented)

The RVBAR_EL3 characteristics are:

**Purpose**

If EL3 is the highest Exception level implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch64 state.

**Configurations**

This register is present only when EL3 is implemented. Otherwise, direct accesses to RVBAR_EL3 are UNDEFINED.

Only implemented if the highest Exception level implemented is EL3.

**Attributes**

RVBAR_EL3 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-0</td>
<td>ResetAddress</td>
</tr>
</tbody>
</table>

**ResetAddress, bits [63:0]**

The IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 64-bit state. Bits[1:0] of this register are 00, as this address must be aligned, and the address must be within the physical address size supported by the PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Accessing RVBAR_EL3**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, RVBAR_EL3**

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b11</td>
<td>0b11</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>
```

if PSTATE.EL == EL3 && IsHighestEL(EL3) then
    X[t, 64] = RVBAR_EL3;
else
    UNDEFINED;
D17.2.116 S3_<op1>_C<Cn>_C<Cm>_C<op2>, IMPLEMENTATION DEFINED registers

The S3_<op1>_C<Cn>_C<Cm>_C<op2> characteristics are:

**Purpose**

This area of the instruction set space is reserved for IMPLEMENTATION DEFINED registers.

**Configurations**

There are no configuration notes.

**Attributes**

S3_<op1>_C<Cn>_C<Cm>_C<op2> is a 64-bit register.

**Field descriptions**

![Field representation]

**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

**Accessing S3_<op1>_C<Cn>_C<Cm>_C<op2>**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, S3_<op1>_C<Cn>_C<Cm>_C<op2>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>op1[2:0]</td>
<td>0b1x11</td>
<td>Cm[3:0]</td>
<td>op2[2:0]</td>
</tr>
</tbody>
</table>

```assembly
if PSTATE.EL == EL0 then
    if !(EL2Enabled() & HCR_EL2.<E2H,TGE> == '11') & SCTLR_EL1.TIDCP == '1' then
        if EL2Enabled() & HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        AArch64.ImpDefSysRegRead(op0, op1, CRn, CRm, op2, t);
    endif
else
    AArch64.ImpDefSysRegRead(op0, op1, CRn, CRm, op2, t);
endif
```

```assembly
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & SCTLR_EL2.TIDCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.ImpDefSysRegRead(op0, op1, CRn, CRm, op2, t);
    endif
else
    AArch64.ImpDefSysRegRead(op0, op1, CRn, CRm, op2, t);
endif
```

```assembly
elsif PSTATE.EL == EL2 then
    AArch64.ImpDefSysRegRead(op0, op1, CRn, CRm, op2, t);
else
    AArch64.ImpDefSysRegRead(op0, op1, CRn, CRm, op2, t);
endif
```

```assembly
elsif PSTATE.EL == EL3 then
    AArch64.ImpDefSysRegRead(op0, op1, CRn, CRm, op2, t);
else
    AArch64.ImpDefSysRegRead(op0, op1, CRn, CRm, op2, t);
endif
```
### MSR S3_<op1>_<Cn>_C<Cm>_<op2>, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>op1[2:0]</td>
<td>0b1x11</td>
<td>Cm[3:0]</td>
<td>op2[2:0]</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.TIDCP == '1' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
      else
        AArch64.SystemAccessTrap(EL1, 0x18);
      endif
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.TIDCP == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.ImpDefSysRegWrite(op0, op1, CRn, CRm, op2, t);
    endif
  else
    PSTATE.EL == EL1 then
      if EL2Enabled() && HCR_EL2.TIDCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      else
        AArch64.ImpDefSysRegWrite(op0, op1, CRn, CRm, op2, t);
      endif
      elseif PSTATE.EL == EL2 then
        AArch64.ImpDefSysRegWrite(op0, op1, CRn, CRm, op2, t);
      elseif PSTATE.EL == EL3 then
        AArch64.ImpDefSysRegWrite(op0, op1, CRn, CRm, op2, t);
      endif
    else
      AArch64.ImpDefSysRegWrite(op0, op1, CRn, CRm, op2, t);
    endif
D17.2.117 SCR_EL3, Secure Configuration Register

The SCR_EL3 characteristics are:

**Purpose**

Defines the configuration of the current Security state. It specifies:

- The Security state of EL0, EL1, and EL2. The Security state is Secure, Non-secure, or Realm.
- The Execution state at lower Exception levels.
- WhetherIRQ, F IQ, SError interrupts, and External abort exceptions are taken to EL3.
- Whether various operations are trapped to EL3.

**Configurations**

This register is present only when EL3 is implemented. Otherwise, direct accesses to SCR_EL3 are UNDEFINED.

**Attributes**

SCR_EL3 is a 64-bit register.

**Field descriptions**

Bit [63]
Reserved, RES0.

NSE, bit [62]

*When FEAT_RME is implemented:*

This field, evaluated with SCR_EL3.NS, selects the Security state of EL2 and lower Exception levels.

For a description of the values derived by evaluating NS and NSE together, see SCR_EL3.NS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.
Bits [61:49]
Reserved, RES0.

GPF, bit [48]

When FEAT_RME is implemented:
Controls the reporting of Granule protection faults at EL0, EL1 and EL2.

0b0 This control does not cause exceptions to be routed from EL0, EL1 or EL2 to EL3.
0b1 GPFs at EL0, EL1 and EL2 are routed to EL3 and reported as Granule Protection Check exceptions.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bits [47:42]
Reserved, RES0.

EnTP2, bit [41]

When FEAT_SME is implemented:
Traps instructions executed at EL2, EL1, and EL0 that access TPIDR2_EL0 to EL3. The exception is reported using ESR_ELx.EC value 0x18.

0b0 This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b1 This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

TRNDR, bit [40]

When FEAT_RNG_TRAP is implemented:
Controls trapping of reads of RNDR and RNDRRS. The exception is reported using ESR_ELx.EC value 0x18.

0b0 This control does not cause RNDR and RNDRRS to be trapped.

When FEAT_RNG is implemented:
• ID_AA64ISAR0_EL1.RNDR returns the value 0b0001.
When FEAT_RNG is not implemented:
• ID_AA64ISAR0_EL1.RNDR returns the value 0b0000.
• MRS reads of RNDR and RNDRRS are UNDEFINED.

0b1 ID_AA64ISAR0_EL1.RNDR returns the value 0b0001.
Any attempt to read RNDR or RNDRRS is trapped to EL3.

When FEAT_RNG is not implemented, Arm recommends that SCR_EL3.TRNDR is initialized before entering Exception levels below EL3 and not subsequently changed.

The reset behavior of this field is:
• On a Warm reset, this field resets to 0.

Otherwise:
Reserved, RES0.

Bit [39]
Reserved, RES0.
HXEn, bit [38]

When FEAT_HCX is implemented:

Enables access to the HCRX_EL2 register at EL2 from EL3.

0b0  Accesses at EL2 to HCRX_EL2 are trapped to EL3. Indirect reads of HCRX_EL2 return 0.

0b1  This control does not cause any instructions to be trapped.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ADEn, bit [37]

When FEAT_LS64_ACCDATA is implemented:

Enables access to the ACCDATA_EL1 register at EL1 and EL2.

0b0  Accesses to ACCDATA_EL1 at EL1 and EL2 are trapped to EL3, unless the accesses are trapped to EL2 by the EL2 fine-grained trap.

0b1  This control does not cause accesses to ACCDATA_EL1 to be trapped.

If the HFGWTR_EL2.nACCDATA_EL1 or HFGRT_EL2.nACCDATA_EL1 traps are enabled, they take priority over this trap.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnAS0, bit [36]

When FEAT_LS64_ACCDATA is implemented:

Traps execution of an ST64BV0 instruction at EL0, EL1, or EL2 to EL3.

0b0  EL0 execution of an ST64BV0 instruction is trapped to EL3, unless it is trapped to EL1 by SCTLR_EL1.EnAS0, or to EL2 by either HCRX_EL2.EnAS0 or SCTLR_EL2.EnAS0.

EL1 execution of an ST64BV0 instruction is trapped to EL3, unless it is trapped to EL2 by HCRX_EL2.EnAS0.

EL2 execution of an ST64BV0 instruction is trapped to EL3.

0b1  This control does not cause any instructions to be trapped.

A trap of an ST64BV0 instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x8000001.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AMVOFFEN, bit [35]

When FEAT_AMUv1p1 is implemented:

Activity Monitors Virtual Offsets Enable.

0b0  Accesses to AMEVCTVOFF0<n>_EL2 and AMEVCTVOFF1<n>_EL2 at EL2 are trapped to EL3. Indirect reads of the virtual offset registers are zero.

0b1  Accesses to AMEVCTVOFF0<n>_EL2 and AMEVCTVOFF1<n>_EL2 are not affected by this field.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**TME, bit [34]**

*When FEAT_TME is implemented:*

Enables access to the TSTART, TCOMMIT, TTEST and TCANCEL instructions at EL0, EL1 and EL2.

- **0b0** EL0, EL1 and EL2 accesses to TSTART, TCOMMIT, TTEST and TCANCEL instructions are UNDEFINED.
- **0b1** This control does not cause any instruction to be UNDEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**TWEDEL, bits [33:30]**

*When FEAT_TWED is implemented:*

TWE Delay. A 4-bit unsigned number that, when SCR_EL3.TWEDEn is 1, encodes the minimum delay in taking a trap of WFE* caused by SCR_EL3.TWE as $2^{(TWEDEL + 8)}$ cycles.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**TWEDEn, bit [29]**

*When FEAT_TWED is implemented:*

TWE Delay Enable. Enables a configurable delayed trap of the WFE* instruction caused by SCR_EL3.TWE.

Traps are reported using an ESR_ELx.EC value of 0x01.

- **0b0** The delay for taking the trap is IMPLEMENTATION DEFINED.
- **0b1** The delay for taking the trap is at least the number of cycles defined in SCR_EL3.TWEDEL.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**ECVEn, bit [28]**

*When FEAT_ECV is implemented:*

ECV Enable. Enables access to the CNTPOFF_EL2 register.

- **0b0** EL2 accesses to CNTPOFF_EL2 are trapped to EL3, and the value of CNTPOFF_EL2 is treated as 0 for all purposes other than direct reads or writes to the register from EL3.
- **0b1** EL2 accesses to CNTPOFF_EL2 are not trapped to EL3 by this mechanism.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
D17.2 General system control registers

FGTEn, bit [27]

When FEAT_FGT is implemented:

Fine-Grained Traps Enable. When EL2 is implemented, enables the traps to EL2 controlled by HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGTR_EL2, HFGITR_EL2, and HFGWTR_EL2, and controls access to those registers.

____ Note _______

If EL2 is not implemented but EL3 is implemented, FEAT_FGT implements the MDCR_EL3.TDCC traps.

| 0b0 | EL2 accesses to HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGTR_EL2, HFGITR_EL2 and HFGWTR_EL2 registers are trapped to EL3, and the traps to EL2 controlled by those registers are disabled. |
| 0b1 | EL2 accesses to HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGTR_EL2, HFGITR_EL2 and HFGWTR_EL2 registers are not trapped to EL3 by this mechanism. |

Traps caused by accesses to the fine-grained trap registers are reported using an ESR_ELx.EC value of 0x18 and its associated ISS.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ATA, bit [26]

When FEAT_MTE2 is implemented:

Allocation Tag Access. Controls access to Allocation Tags, System registers for Memory tagging, and prevention of Tag checking, at EL2, EL1 and EL0.

| 0b0 | Access to Allocation Tags is prevented at EL2, EL1, and EL0. Accesses at EL1 and EL2 to GCR_EL1, RGSR_EL1, TFSR_EL1, TFSR_EL2 or TFSRE0_EL1 that are not UNDEFINED or trapped to a lower Exception level are trapped to EL3. Memory accesses at EL2, EL1, and EL0 are not subject to a Tag Check operation. |
| 0b1 | This control does not prevent access to Allocation Tags at EL2, EL1, and EL0. This control does not prevent Tag checking at EL2, EL1, and EL0. |

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnSCXT, bit [25]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Enable access to the SCXTNUM_EL2, SCXTNUM_EL1, and SCXTNUM_EL0 registers.

| 0b0 | Accesses at EL0, EL1 and EL2 to SCXTNUM_EL0, SCXTNUM_EL1, or SCXTNUM_EL2 registers are trapped to EL3 if they are not trapped by a higher priority exception, and the values of these registers are treated as 0. |
| 0b1 | This control does not cause any accesses to be trapped, or register values to be treated as 0. |
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bits [24:22]**

Reserved, RES0.

**FIEN, bit [21]**

*When FEAT_RASv1p1 is implemented:*

Fault Injection enable. Trap accesses to the registers ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1 from EL1 and EL2 to EL3, reported using an ESR_ELx.EC value of 0x18.

- **0b0** Accesses to the specified registers from EL1 and EL2 generate a Trap exception to EL3.
- **0b1** This control does not cause any instructions to be trapped.

If EL3 is not implemented, the Effective value of SCR_EL3.FIEN is 0b1.

If ERRIDR_EL1.NUM is zero, meaning no error records are implemented, or no error record accessible using System registers is owned by a node that implements the RAS Common Fault Injection Model Extension, then this bit might be RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**NMEA, bit [20]**

*When FEAT_DoubleFault is implemented:*

Non-maskable External Aborts. When SCR_EL3.EA == 1, controls whether PSTATE.A masks SError interrupts at EL3.

- **0b0** If SCR_EL3.EA == 1, asserted SError interrupts are not taken at EL3 if PSTATE.A == 1.
- **0b1** If SCR_EL3.EA == 1, asserted SError interrupts are taken at EL3 regardless of the value of PSTATE.A.

When SCR_EL3.EA == 0:

- Asserted SError interrupts are not taken at EL3 regardless of the value of PSTATE.A and this field.
- This field is ignored and its Effective value is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**EASE, bit [19]**

*When FEAT_DoubleFault is implemented:*

External aborts to SError interrupt vector.

- **0b0** Synchronous External abort exceptions taken to EL3 are taken to the appropriate synchronous exception vector offset from VBAR_EL3.
- **0b1** Synchronous External abort exceptions taken to EL3 are taken to the appropriate SError interrupt vector offset from VBAR_EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.
**EEL2, bit [18]**

*When FEAT_SEL2 is implemented:*

Secure EL2 Enable.

- **0b0**: All behaviors associated with Secure EL2 are disabled. All registers, including timer registers, defined by FEAT_SEL2 are **UNDEFINED**, and those timers are disabled.
- **0b1**: All behaviors associated with Secure EL2 are enabled.

When the value of this bit is 1, then:

- When SCR_EL3.NS == 0, the SCR_EL3.RW bit is treated as 1 for all purposes other than reading or writing the register.
- If Secure EL1 is using AArch32, then any of the following operations, executed in Secure EL1, is trapped to Secure EL2, using the EC value of `ESR_EL2.EC==0x3`:
  - A read or write of the SCR.
  - A read or write of the NSACR.
  - A read or write of the MVBAR.
  - A read or write of the SDCR.
  - Execution of an ATS12NSO** instruction.
- If Secure EL1 is using AArch32, then any of the following operations, executed in Secure EL1, is trapped to Secure EL2 using the EC value of `ESR_EL2.EC==0x0`:
  - Execution of an SRS instruction that uses R13_mon.
  - Execution of an MRS (Banked register) or MSR (Banked register) instruction that would access SPSR_mon, R13_mon, or R14_mon.

**Note**

If the Effective value of SCR_EL3.EEL2 is 0, then these operations executed in Secure EL1 using AArch32 are trapped to EL3.

A Secure only implementation that does not implement EL3 but implements EL2, behaves as if SCR_EL3.EEL2 == 1.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**API, bit [17]**

*When FEAT_SEL2 is implemented and FEAT_PAuth is implemented:*

Controls the use of the following instructions related to Pointer Authentication. Traps are reported using an ESR_ELx.EC value of `0x9`:

- PACGA, which is always enabled.
- AUTDA, AUTDB, AUTDZA, AUTDZB, AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZA, AUTIZB, PACDA, PACDB, PACDZA, PACDZB, PACIA, PACIA1716, PACIASP, PACIAZ, PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZA, PACIZB, RETAA, RETAB, BRAA, BRAB, BLRAA, BLRAB, BRAAZ, BRABZ, BLRAAZ, BLRABZ, ERETTA, ERETTAB, LDRRAA and LDRRAB when:
  - In EL0, when HCR_EL2.TGE == 0 or HCR_EL2.E2H == 0, and the associated SCTLR_EL1.En<<M> == 1.
— In EL0, when HCR_EL2.TGE == 1 and HCR_EL2.E2H == 1, and the associated SCTLR_EL2.En<N><M> == 1.
— In EL1, when the associated SCTLR_EL1.En<N><M> == 1.
— In EL2, when the associated SCTLR_EL2.En<N><M> == 1.

0b0 The use of any instruction related to pointer authentication in any Exception level except EL3 when the instructions are enabled are trapped to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.API bit.

0b1 This control does not cause any instructions to be trapped.

An instruction is trapped only if Pointer Authentication is enabled for that instruction, for more information, see Pointer authentication on page D8-5164.

—— Note ———

If FEAT_PAuth is implemented but EL3 is not implemented, the system behaves as if this bit is 1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SEL2 is not implemented and FEAT_PAuth is implemented:

Controls the use of instructions related to Pointer Authentication:
• PACGA.
• AUTDA, AUTDB, AUTDZA, AUTDZB, AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZA, AUTIZB, PACDA, PACDB, PACDZA, PACDZB, PACIA, PACIA1716, PACIASP, PACIAZ, PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZ, RETAA, RETAB, BRAA, BRAB, BLRAA, BLRAB, BRAAZ, BRABZ, BLRAAZ, BLRABZ, ERETTA, ERETTA, LDRAA and LDRAB when:
  — In Non-secure EL0, when HCR_EL2.TGE == 0 or HCR_EL2.E2H == 0, and the associated SCTLR_EL1.En<N><M> == 1.
  — In Non-secure EL0, when HCR_EL2.TGE == 1 and HCR_EL2.E2H == 1, and the associated SCTLR_EL2.En<N><M> == 1.
  — In Secure EL0, when the associated SCTLR_EL1.En<N><M> == 1.
  — In Secure or Non-secure EL1, when the associated SCTLR_EL1.En<N><M> == 1.
  — In EL2, when the associated SCTLR_EL2.En<N><M> == 1.

0b0 The use of any instruction related to pointer authentication in any Exception level except EL3 when the instructions are enabled are trapped to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.API bit.

0b1 This control does not cause any instructions to be trapped.

—— Note ———

If FEAT_PAuth is implemented but EL3 is not implemented, the system behaves as if this bit is 1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

APK, bit [16]

When FEAT_PAuth is implemented:

Trap registers holding "key" values for Pointer Authentication. Traps accesses to the following registers, using an ESR_ELx.EC value of 0x18, from EL1 or EL2 to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.APK bit or other traps:
• APIAKeyLo_EL1, APIAKeyHi_EL1, APIBKeyLo_EL1, APIBKeyHi_EL1.
• APDAKeyLo_EL1, APDAKeyHi_EL1, APDBKeyLo_EL1, APDBKeyHi_EL1.
• APGAKeyLo_EL1, and APGAKeyHi_EL1.

0b0  Access to the registers holding "key" values for pointer authentication from EL1 or EL2 are trapped to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.APK bit or other traps.

0b1  This control does not cause any instructions to be trapped.

For more information, see Pointer authentication on page D8-5164.

Note
If FEAT_PAuth is implemented but EL3 is not implemented, the system behaves as if this bit is 1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

TERR, bit [15]

When FEAT_RAS is implemented:
Trap Error record accesses. Accesses to the RAS ERR* and RAS ERX* registers from EL1 and EL2 to EL3 are trapped as follows:
• Accesses from EL1 and EL2 using AArch64 to the following registers are trapped and reported using an ESR_ELx.EC value of 0x18:
  — ERRIDR_EL1, ERRSEL_EL1, ERXADDR_EL1, ERXCTRL_EL1, ERXFR_EL1, ERXMISC0_EL1, ERXMISC1_EL1, and ERXSTATUS_EL1.
• If FEAT_RASv1p1 is implemented, accesses from EL1 and EL2 using AArch64 to ERXMISC2_EL1, and ERXMISC3_EL1, are trapped and reported using an ESR_ELx.EC value of 0x18.
• Accesses from EL1 and EL2 using AArch32, to the following registers are trapped and reported using an ESR_ELx.EC value of 0x03:
  — ERRIDR, ERRSEL, ERXADDR, ERXADDR2, ERXCTRL, ERXCTRL2, ERXFR, ERXFR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, and ERXSTATUS.
• If FEAT_RASv1p1 is implemented, accesses from EL1 and EL2 using AArch32 to the following registers are trapped and reported using an ESR_ELx.EC value of 0x03:
  — ERXMISC4, ERXMISC5, ERXMISC6, and ERXMISC7.

0b0  This control does not cause any instructions to be trapped.

0b1  Accesses to the specified registers from EL1 and EL2 generate a Trap exception to EL3.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

TLOR, bit [14]

When FEAT_LOR is implemented:
Trap LOR registers. Traps accesses to the LORSA_EL1, LOREA_EL1, LORN_EL1, LORC_EL1, and LORID_EL1 registers from EL1 and EL2 to EL3, unless the access has been trapped to EL2.

0b0  This control does not cause any instructions to be trapped.

0b1  EL1 and EL2 accesses to the LOR registers that are not UNDEFINED are trapped to EL3, unless it is trapped HCR_EL2.TLOR.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
**Otherwise:**

Reserved, RES0.

**TWE, bit [13]**

Traps EL2, EL1, and EL0 execution of WFE instructions to EL3, from any Security state and both Execution states, reported using an ESR_ELx.EC value of 0x01.

When `FEAT_WFxT` is implemented, this trap also applies to the WFET instruction.

- **0b0** This control does not cause any instructions to be trapped.
- **0b1** Any attempt to execute a WFE instruction at any Exception level lower than EL3 is trapped to EL3, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by `SCTLR.nTWE`, `HCR.TWE`, `SCTLR_EL1.nTWE`, `SCTLR_EL2.nTWE`, or `HCR_EL2.TWE`.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

---

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

---

For more information about when WFE instructions can cause the PE to enter a low-power state, see `Wait for Event` on page D1-4676.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**TWI, bit [12]**

Traps EL2, EL1, and EL0 execution of WFI instructions to EL3, from any Security state and both Execution states, reported using an ESR_ELx.EC value of 0x01.

When `FEAT_WFxT` is implemented, this trap also applies to the WFIT instruction.

- **0b0** This control does not cause any instructions to be trapped.
- **0b1** Any attempt to execute a WFI instruction at any Exception level lower than EL3 is trapped to EL3, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by `SCTLR.nTWI`, `HCR.TWII`, `SCTLR_EL1.nTWI`, `SCTLR_EL2.nTWI`, or `HCR_EL2.TWII`.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

---

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

---

For more information about when WFI instructions can cause the PE to enter a low-power state, see `Wait for Interrupt mechanism` on page D1-4678.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
### ST, bit [11]

Traps Secure EL1 accesses to the Counter-timer Physical Secure timer registers to EL3, from AArch64 state only, reported using an ESR_ELx.EC value of 0x18.

- **0b0**: Secure EL1 using AArch64 accesses to the `CNTPS.TVAL_EL1`, `CNTPS.CTL_EL1`, and `CNTPS.CV_AL_EL1` are trapped to EL3 when Secure EL2 is disabled. If Secure EL2 is enabled, the behavior is as if the value of this field was 0b1.

- **0b1**: This control does not cause any instructions to be trapped.

--- **Note** ---

Accesses to the Counter-timer Physical Secure timer registers are always enabled at EL3. These registers are not accessible at EL0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### RW, bit [10]

**When EL1 is capable of using AArch32 or EL2 is capable of using AArch32:**

Execution state control for lower Exception levels.

- **0b0**: Lower levels are all AArch32.
- **0b1**: The next lower level is AArch64.
  - If EL2 is present:
    - EL2 is AArch64.
    - EL2 controls EL1 and EL0 behaviors.
  - If EL2 is not present:
    - EL1 is AArch64.
    - EL0 is determined by the Execution state described in the current process state when executing at EL0.

If AArch32 state is supported by the implementation at EL1, `SCR_EL3.NS == 1` and AArch32 state is not supported by the implementation at EL2, the Effective value of this bit is 1.

If AArch32 state is supported by the implementation at EL1, `FEAT_SEL2` is implemented and `SCR_EL3.{EEL2, NS} == {1, 0}`, the Effective value of this bit is 1.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RAO/WI.

### SIF, bit [9]

Secure instruction fetch. When the PE is in Secure state, this bit disables instruction execution from memory marked in the first stage of translation as being Non-secure.

- **0b0**: Secure state instruction execution from memory marked in the first stage of translation as being Non-secure is permitted.
- **0b1**: Secure state instruction execution from memory marked in the first stage of translation as being Non-secure is not permitted.

When `FEAT_PAN3` is implemented, it is IMPLEMENTATION DEFINED whether `SCR_EL3.SIF` is also used to determine instruction access permission for the purpose of PAN.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
HCE, bit [8]

Hypervisor Call instruction enable. Enables HVC instructions at EL3 and, if EL2 is enabled in the current Security state, at EL2 and EL1, in both Execution states, reported using an ESR_ELx.EC value of 0x00.

0b0   HVC instructions are UNDEFINED.
0b1   HVC instructions are enabled at EL3, EL2, and EL1.

--- Note ---

HVC instructions are always UNDEFINED at EL0 and, if Secure EL2 is disabled, at Secure EL1. Any resulting exception is taken from the current Exception level to the current Exception level.

---

If EL2 is not implemented, this bit is RES0.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

SMD, bit [7]

Secure Monitor Call disable. Disables SMC instructions at EL1 and above, from any Security state and both Execution states, reported using an ESR_ELx.EC value of 0x00.

0b0   SMC instructions are enabled at EL3, EL2 and EL1.
0b1   SMC instructions are UNDEFINED.

--- Note ---

SMC instructions are always UNDEFINED at EL0. Any resulting exception is taken from the current Exception level to the current Exception level.

If HCR_EL2.TSC or HCR.TSC traps attempted EL1 execution of SMC instructions to EL2, that trap has priority over this disable.

---

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

Bits [5:4]

Reserved, RES1.

EA, bit [3]

External Abort and SError interrupt routing.

0b0   When executing at Exception levels below EL3, External aborts and SError interrupts are not taken to EL3.
      In addition, when executing at EL3:
      • SError interrupts are not taken.
      • External aborts are taken to EL3.

0b1   When executing at any Exception level, External aborts and SError interrupts are taken to EL3.

For more information, see Establishing the target Exception level of an asynchronous exception on page D1-4657.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
FIQ, bit [2]

Physical FIQ Routing.

0b0  When executing at Exception levels below EL3, physical FIQ interrupts are not taken to EL3.
When executing at EL3, physical FIQ interrupts are not taken.

0b1  When executing at any Exception level, physical FIQ interrupts are taken to EL3.

For more information, see Establishing the target Exception level of an asynchronous exception on page D1-4657.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IRQ, bit [1]

Physical IRQ Routing.

0b0  When executing at Exception levels below EL3, physical IRQ interrupts are not taken to EL3.
When executing at EL3, physical IRQ interrupts are not taken.

0b1  When executing at any Exception level, physical IRQ interrupts are taken to EL3.

For more information, see Establishing the target Exception level of an asynchronous exception on page D1-4657.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NS, bit [0]

When FEAT_RME is implemented:

Non-secure bit. This field is used in combination with SCR_EL3.NSE to select the Security state of EL2 and lower Exception levels.

<table>
<thead>
<tr>
<th>NSE</th>
<th>NS</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b0</td>
<td>Secure.</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>Non-secure.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>Realm.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Non-secure bit.

0b0  Indicates that EL0 and EL1 are in Secure state.
When FEAT_SEL2 is implemented and SCR_EL3.EEL2 == 1, then EL2 is using AArch64 and in Secure state.

0b1  Indicates that Exception levels lower than EL3 are in Non-secure state, so memory accesses from those Exception levels cannot access Secure memory.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Accessing SCR_EL3

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, SCR_EL3**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  UNDEFINED;
elsif PSTATE_EL == EL2 then
  UNDEFINED;
elsif PSTATE_EL == EL3 then
  X[t, 64] = SCR_EL3;

**MSR SCR_EL3, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  UNDEFINED;
elsif PSTATE_EL == EL2 then
  UNDEFINED;
elsif PSTATE_EL == EL3 then
  SCR_EL3 = X[t, 64];
D17.2.118  SCTLR_EL1, System Control Register (EL1)

The SCTLR_EL1 characteristics are:

### Purpose

Provides top level control of the system, including its memory system, at EL1 and EL0.

### Configurations

AArch64 System register SCTLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register SCTLR[31:0].

### Attributes

SCTLR_EL1 is a 64-bit register.

### Field descriptions

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>TIDCP</td>
</tr>
<tr>
<td>62</td>
<td>SPINTMASK</td>
</tr>
<tr>
<td>61</td>
<td>NMI</td>
</tr>
<tr>
<td>60</td>
<td>EnT2</td>
</tr>
<tr>
<td>59</td>
<td>EPAN</td>
</tr>
<tr>
<td>58</td>
<td>EnALS</td>
</tr>
<tr>
<td>57</td>
<td>EnASR</td>
</tr>
<tr>
<td>56</td>
<td>TME</td>
</tr>
<tr>
<td>55</td>
<td>TME0</td>
</tr>
<tr>
<td>54</td>
<td>LATA0</td>
</tr>
<tr>
<td>53</td>
<td>LATA1</td>
</tr>
<tr>
<td>52</td>
<td>TWEDEn</td>
</tr>
<tr>
<td>51</td>
<td>TWEDEL</td>
</tr>
<tr>
<td>50</td>
<td>TCF</td>
</tr>
<tr>
<td>49</td>
<td>TCF0</td>
</tr>
<tr>
<td>48</td>
<td>CMON</td>
</tr>
<tr>
<td>47</td>
<td>MSCEn</td>
</tr>
<tr>
<td>46</td>
<td>RES0</td>
</tr>
<tr>
<td>45</td>
<td>BT0</td>
</tr>
<tr>
<td>44</td>
<td>BT1</td>
</tr>
<tr>
<td>43</td>
<td>ITFSB</td>
</tr>
<tr>
<td>42</td>
<td>RES0</td>
</tr>
<tr>
<td>41</td>
<td>RES0</td>
</tr>
<tr>
<td>40</td>
<td>RES0</td>
</tr>
<tr>
<td>39</td>
<td>RES0</td>
</tr>
<tr>
<td>38</td>
<td>RES0</td>
</tr>
<tr>
<td>37</td>
<td>RES0</td>
</tr>
<tr>
<td>36</td>
<td>RES0</td>
</tr>
<tr>
<td>35</td>
<td>RES0</td>
</tr>
<tr>
<td>34</td>
<td>RES0</td>
</tr>
<tr>
<td>33</td>
<td>RES0</td>
</tr>
<tr>
<td>32</td>
<td>RES0</td>
</tr>
<tr>
<td>31</td>
<td>RES0</td>
</tr>
<tr>
<td>30</td>
<td>RES0</td>
</tr>
<tr>
<td>29</td>
<td>RES0</td>
</tr>
<tr>
<td>28</td>
<td>RES0</td>
</tr>
<tr>
<td>27</td>
<td>RES0</td>
</tr>
<tr>
<td>26</td>
<td>RES0</td>
</tr>
<tr>
<td>25</td>
<td>RES0</td>
</tr>
<tr>
<td>24</td>
<td>RES0</td>
</tr>
<tr>
<td>23</td>
<td>RES0</td>
</tr>
<tr>
<td>22</td>
<td>RES0</td>
</tr>
<tr>
<td>21</td>
<td>RES0</td>
</tr>
<tr>
<td>20</td>
<td>RES0</td>
</tr>
<tr>
<td>19</td>
<td>RES0</td>
</tr>
<tr>
<td>18</td>
<td>RES0</td>
</tr>
<tr>
<td>17</td>
<td>RES0</td>
</tr>
<tr>
<td>16</td>
<td>RES0</td>
</tr>
<tr>
<td>15</td>
<td>RES0</td>
</tr>
<tr>
<td>14</td>
<td>RES0</td>
</tr>
<tr>
<td>13</td>
<td>RES0</td>
</tr>
<tr>
<td>12</td>
<td>RES0</td>
</tr>
<tr>
<td>11</td>
<td>RES0</td>
</tr>
<tr>
<td>10</td>
<td>RES0</td>
</tr>
<tr>
<td>9</td>
<td>RES0</td>
</tr>
<tr>
<td>8</td>
<td>RES0</td>
</tr>
<tr>
<td>7</td>
<td>RES0</td>
</tr>
<tr>
<td>6</td>
<td>RES0</td>
</tr>
<tr>
<td>5</td>
<td>RES0</td>
</tr>
<tr>
<td>4</td>
<td>RES0</td>
</tr>
<tr>
<td>3</td>
<td>RES0</td>
</tr>
<tr>
<td>2</td>
<td>RES0</td>
</tr>
<tr>
<td>1</td>
<td>RES0</td>
</tr>
<tr>
<td>0</td>
<td>RES0</td>
</tr>
</tbody>
</table>

### TIDCP, bit [63]

**When FEAT_TIDCP1 is implemented:**

Trap IMPLEMENTATION DEFINED functionality. When HCR_EL2.{E2H, TGE} != \{1, 1\}, traps EL0 accesses to the encodings reserved for IMPLEMENTATION DEFINED functionality to EL1.

- **0b0**  No instructions accessing the System register or System instruction spaces are trapped by this mechanism.

- **0b1**  Instructions accessing the following System register or System instruction spaces are trapped to EL1 by this mechanism:
  - In AArch64 state, EL0 access to the encodings in the following reserved encoding spaces are trapped and reported using EC syndrome 0x18:
    - IMPLEMENTATION DEFINED System instructions, which are accessed using SYS and SYSL, with CRn = {11, 15}.
IMPLEMENTATION DEFINED System registers, which are accessed using 
MRS and MSR with the S3_<op1>_<Cn>_<Cm>_<op2> register name.

- In AArch32 state, EL0 MCR and MRC access to the following encodings are 
  trapped and reported using EC syndrome 0x03:
  - All coproc==p15, CRn==c9, opc1 == {0-7}, CRm == {c0-c2, c5-c8}, 
    opc2 == {0-7}.
  - All coproc==p15, CRn==c10, opc1 == {0-7}, CRm == {c0, c1, c4, c8}, 
    opc2 == {0-7}.
  - All coproc==p15, CRn==c11, opc1=={0-7}, CRm == {c0-c8, c15}, opc2 
    == {0-7}.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL1, this field resets to an 
  architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

SPINTMASK, bit [62]
When FEAT_NMI is implemented:
SP Interrupt Mask enable. When SCTLR_EL1.NMI is 1, controls whether PSTATE.SP acts as an 
interrupt mask, and controls the value of PSTATE.ALLINT on taking an exception to EL1.
0b0    Does not cause PSTATE.SP to mask interrupts. 
PSTATE.ALLINT is set to 1 on taking an exception to EL1.
0b1    When PSTATE.SP is 1 and execution is at EL1, an IRQ or FIQ interrupt that is targeted 
to EL1 is masked regardless of any denotation of Superpriority. 
PSTATE.ALLINT is set to 0 on taking an exception to EL1.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL1, this field resets to an 
  architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

NMI, bit [61]
When FEAT_NMI is implemented:
Non-maskable Interrupt enable.
0b0    This control does not affect interrupt masking behavior.
0b1    This control enables all of the following:
  - The use of the PSTATE.ALLINT interrupt mask.
  - IRQ and FIQ interrupts to have Superpriority as an additional attribute.
  - PSTATE.SP to be used as an interrupt mask.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL1, this field resets to 0.

Otherwise:
Reserved, RES0.
EnTP2, bit [60]

When FEAT_SME is implemented:

Traps instructions executed at EL0 that access TPIDR2_EL0 to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. The exception is reported using ESR_ELx.EC value 0x18.

0b0 This control causes execution of these instructions at EL0 to be trapped.
0b1 This control does not cause execution of any instructions to be trapped.

If FEAT_VHE is implemented, EL2 is implemented and enabled in the current Security state, and HCR_EL2.{E2H, TGE} == {1, 1}, this field has no effect on execution at EL0.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bits [59:58]
Reserved, RES0.

EPAN, bit [57]

When FEAT_PAN3 is implemented:

Enhanced Privileged Access Never. When PSTATE.PAN is 1, determines whether an EL1 data access to a page with stage 1 EL0 instruction access permission generates a Permission fault as a result of the Privileged Access Never mechanism.

0b0 No additional Permission faults are generated by this mechanism.
0b1 An EL1 data access to a page with stage 1 EL0 data access permission or stage 1 EL0 instruction access permission generates a Permission fault.

Any speculative data accesses that would generate a Permission fault if the accesses were not speculative will not cause an allocation into a cache.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EnALS, bit [56]

When FEAT_LS64 is implemented:

When HCR_EL2.{E2H, TGE} != {1, 1}, traps execution of an LD64B or ST64B instruction at EL0 to EL1.

0b0 Execution of an LD64B or ST64B instruction at EL0 is trapped to EL1.
0b1 This control does not cause any instructions to be trapped.

A trap of an LD64B or ST64B instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x00000002.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.
EnAS0, bit [55]

*When FEAT_LS64_ACCDATA is implemented:*

When HCR_EL2.{E2H, TGE} != {1, 1}, traps execution of an ST64BV0 instruction at EL0 to EL1.

- **0b0** Execution of an ST64BV0 instruction at EL0 is trapped to EL1.
- **0b1** This control does not cause any instructions to be trapped.

A trap of an ST64BV0 instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000001.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

EnASR, bit [54]

*When FEAT_LS64_V is implemented:*

When HCR_EL2.{E2H, TGE} != {1, 1}, traps execution of an ST64BV instruction at EL0 to EL1.

- **0b0** Execution of an ST64BV instruction at EL0 is trapped to EL1.
- **0b1** This control does not cause any instructions to be trapped.

A trap of an ST64BV instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000000.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

TME, bit [53]

*When FEAT_TME is implemented:*

Enables the Transactional Memory Extension at EL1.

- **0b0** Any attempt to execute a TSTART instruction at EL1 is trapped to EL1, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART instructions to be UNEDEFINED at EL1.
- **0b1** This control does not cause any TSTART instruction to be trapped.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

TME0, bit [52]

*When FEAT_TME is implemented:*

Enables the Transactional Memory Extension at EL0.

- **0b0** Any attempt to execute a TSTART instruction at EL0 is trapped to EL1, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART instructions to be UNEDEFINED at EL0.
- **0b1** This control does not cause any TSTART instruction to be trapped.

If FEAT_VHE is implemented, EL2 is implemented and enabled in the current Security state, and HCR_EL2.{E2H, TGE} == {1, 1}, this field has no effect on execution at EL0.
The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TMT, bit [51]

When FEAT_TME is implemented:

Forces a trivial implementation of the Transactional Memory Extension at EL1.

0b0       This control does not cause any TSTART instruction to fail.
0b1       When the TSTART instruction is executed at EL1, the transaction fails with a TRIVIAL failure cause.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TMT0, bit [50]

When FEAT_TME is implemented:

Forces a trivial implementation of the Transactional Memory Extension at EL0.

0b0       This control does not cause any TSTART instruction to fail.
0b1       When the TSTART instruction is executed at EL0, the transaction fails with a TRIVIAL failure cause.

If FEAT_VHE is implemented, EL2 is implemented and enabled in the current Security state, and

HCR_EL2.{E2H, TGE} == {1, 1}, this field has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TWEDEL, bits [49:46]

When FEAT_TWED is implemented:

TWE Delay. A 4-bit unsigned number that, when SCTLR_EL1.TWEDEn is 1, encodes the minimum delay in taking a trap of WFE* caused by SCTLR_EL1.nTWE as $2^{(TWEDEL + 8)}$ cycles.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TWEDEn, bit [45]

When FEAT_TWED is implemented:

TWE Delay Enable. Enables a configurable delayed trap of the WFE* instruction caused by SCTLR_EL1.nTWE.

0b0       The delay for taking the trap is IMPLEMENTATION DEFINED.
0b1       The delay for taking the trap is at least the number of cycles defined in SCTLR_EL1.TWEDEL.
The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**DSSBS, bit [44]**

*When FEAT_SSBS is implemented:*
Default PSTATE.SSBS value on Exception Entry.

- **0b0** PSTATE.SSBS is set to 0 on an exception to EL1.
- **0b1** PSTATE.SSBS is set to 1 on an exception to EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

**Otherwise:**
Reserved, RES0.

**ATA, bit [43]**

*When FEAT_MTE2 is implemented:*
Allocation Tag Access in EL1.

When SCR_EL3.ATA == 1 and HCR_EL2.ATA == 1, controls access to Allocation Tags and Tag Check operations in EL1.

- **0b0** Access to Allocation Tags is prevented at EL1.
  Memory accesses at EL1 are not subject to a Tag Check operation.
- **0b1** This control does not prevent access to Allocation Tags at EL1.
  Tag Checked memory accesses at EL1 are subject to a Tag Check operation.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**ATA0, bit [42]**

*When FEAT_MTE2 is implemented:*
Allocation Tag Access in EL0.

When SCR_EL3.ATA == 1, HCR_EL2.ATA == 1, and HCR_EL2.{E2H, TGE} != {1, 1}, controls access to Allocation Tags and Tag Check operations in EL0.

- **0b0** Access to Allocation Tags is prevented at EL0.
  Memory accesses at EL0 are not subject to a Tag Check operation.
- **0b1** This control does not prevent access to Allocation Tags at EL0.
  Tag Checked memory accesses at EL0 are subject to a Tag Check operation.

--- **Note** ---

Software may change this control bit on a context switch.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.
TCF, bits [41:40]

When \textit{FEAT\_MTE2} is implemented:

Tag Check Fault in EL1. Controls the effect of Tag Check Faults due to Loads and Stores in EL1.

If \textit{FEAT\_MTE3} is not implemented, the value 0b11 is reserved.

0b00 Tag Check Faults have no effect on the PE.
0b01 Tag Check Faults cause a synchronous exception.
0b10 Tag Check Faults are asynchronously accumulated.
0b11 \textit{When \textit{FEAT\_MTE3} is implemented:}

Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCF0, bits [39:38]

When \textit{FEAT\_MTE2} is implemented:

Tag Check Fault in EL0. When HCR\_EL2.E2H,TGE \neq \{1,1\}, controls the effect of Tag Check Faults due to Loads and Stores in EL0.

If \textit{FEAT\_MTE3} is not implemented, the value 0b11 is reserved.

\textbf{Note}

Software may change this control bit on a context switch.

0b00 Tag Check Faults have no effect on the PE.
0b01 Tag Check Faults cause a synchronous exception.
0b10 Tag Check Faults are asynchronously accumulated.
0b11 \textit{When \textit{FEAT\_MTE3} is implemented:}

Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ITFSB, bit [37]

When \textit{FEAT\_MTE2} is implemented:

When synchronous exceptions are not being generated by Tag Check Faults, this field controls whether on exception entry into EL1, all Tag Check Faults due to instructions executed before exception entry, that are reported asynchronously, are synchronized into TFSRE0\_EL1 and TFSR\_EL1 registers.

0b0 Tag Check Faults are not synchronized on entry to EL1.
0b1 Tag Check Faults are synchronized on entry to EL1.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.
Otherwise:
  Reserved, RES0.

BT1, bit [36]

When FEAT_BTI is implemented:

PAC Branch Type compatibility at EL1.

  0b0  When the PE is executing at EL1, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == 0b11.
  0b1  When the PE is executing at EL1, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == 0b11.

The reset behavior of this field is:
  • On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:
  Reserved, RES0.

BT0, bit [35]

When FEAT_BTI is implemented:

PAC Branch Type compatibility at EL0.

  0b0  When the PE is executing at EL0, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == 0b11.
  0b1  When the PE is executing at EL0, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == 0b11.

When the value of HCR_EL2.{E2H, TGE} is {1, 1}, the value of SCTLR_EL1.BT0 has no effect on execution at EL0.

The reset behavior of this field is:
  • On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:
  Reserved, RES0.

Bit [34]

Reserved, RES0.

MSCEn, bit [33]

When FEAT_MOPS is implemented and (HCR_EL2.E2H == 0 or HCR_EL2.TGE == 0):

Memory Copy and Memory Set instructions Enable. Enables execution of the Memory Copy and Memory Set instructions at EL0.

  0b0  Execution of the Memory Copy and Memory Set instructions is UNDEFINED at EL0.
  0b1  This control does not cause any instructions to be UNDEFINED.

When FEAT_MOPS is implemented and HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this bit is 0b1.

The reset behavior of this field is:
  • On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:
  Reserved, RES0.
CMOW, bit [32]

When *FEAT_CMOW* is implemented:

Controls cache maintenance instruction permission for the following instructions executed at EL0.

- **IC IV AU, DC CIVAC, DC CIGDVAC** and **DC CIGVAC**.

  0b0 These instructions executed at EL0 with stage 1 read permission, but without stage 1 write permission, do not generate a stage 1 permission fault.

  0b1 If enabled as a result of **SCTLR_EL1.UCI==1**, these instructions executed at EL0 with stage 1 read permission, but without stage 1 write permission, generate a stage 1 permission fault.

When **AArch64.HCR_EL2.{E2H, TGE}** is {1, 1}, this bit has no effect on execution at EL0.

For this control, stage 1 has write permission if all of the following apply:

- **AP[2]** is 0 or **DBM** is 1 in the stage 1 descriptor.
- Where **APTable** is in use, **APTable[1]** is 0 for all levels of the translation table.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally **UNKNOWN** value.

*Otherwise:*  
Reserved, **RES0**.

EnIA, bit [31]

When *FEAT_PAuth* is implemented:

Controls enabling of pointer authentication (using the **APIAKey_EL1** key) of instruction addresses in the EL1&0 translation regime.

For more information, see **Pointer authentication** on page D8-5164.

  0b0 Pointer authentication (using the **APIAKey_EL1** key) of instruction addresses is not enabled.

  0b1 Pointer authentication (using the **APIAKey_EL1** key) of instruction addresses is enabled.

--- **Note** ---

This field controls the behavior of the **AddPACIA** and **AuthIA** pseudocode functions. Specifically, when the field is 1, **AddPACIA** returns a copy of a pointer to which a pointer authentication code has been added, and **AuthIA** returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

---

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally **UNKNOWN** value.

*Otherwise:*  
Reserved, **RES0**.

EnIB, bit [30]

When *FEAT_PAuth* is implemented:

Controls enabling of pointer authentication (using the **APIBKey_EL1** key) of instruction addresses in the EL1&0 translation regime.

For more information, see **Pointer authentication** on page D8-5164.

  0b0 Pointer authentication (using the **APIBKey_EL1** key) of instruction addresses is not enabled.
0b1 Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is enabled.

__________ Note __________

This field controls the behavior of the AddPACIB and AuthIB pseudocode functions. Specifically, when the field is 1, AddPACIB returns a copy of a pointer to which a pointer authentication code has been added, and AuthIB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

LSMAOE, bit [29]

When FEAT_LSMAOC is implemented:

Load Multiple and Store Multiple Atomicity and Ordering Enable.

0b0 For all memory accesses at EL0, A32 and T32 Load Multiple and Store Multiple can have an interrupt taken during the sequence memory accesses, and the memory accesses are not required to be ordered.

0b1 The ordering and interrupt behavior of A32 and T32 Load Multiple and Store Multiple at EL0 is as defined for Armv8.0.

This bit is permitted to be cached in a TLB.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES1.

nTLSMD, bit [28]

When FEAT_LSMAOC is implemented:

No Trap Load Multiple and Store Multiple to Device-nGRE/Device-nGnRE/Device-nGnRnE memory.

0b0 All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are trapped and generate a stage 1 Alignment fault.

0b1 All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are not trapped.

This bit is permitted to be cached in a TLB.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES1.
EnDA, bit [27]

*When FEAT_PAuth is implemented:*

Controls enabling of pointer authentication (using the APDAKey_EL1 key) of instruction addresses in the EL1&0 translation regime.

For more information, see *Pointer authentication on page D8-5164.*

- **0b0** Pointer authentication (using the APDAKey_EL1 key) of data addresses is not enabled.
- **0b1** Pointer authentication (using the APDAKey_EL1 key) of data addresses is enabled.

--- **Note** ---

This field controls the behavior of the AddPACDA and AuthDA pseudocode functions. Specifically, when the field is 1, AddPACDA returns a copy of a pointer to which a pointer authentication code has been added, and AuthDA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

---

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

UCI, bit [26]

Traps EL0 execution of cache maintenance instructions, to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using an ESR_ELx.EC value of 0x18.

This applies to DC CVAU, DC CIVAC, DC CVAC, DC CVAP, and IC IVAU.

If FEAT_DPB2 is implemented, this trap also applies to DC CVADP.

If FEAT_MTE is implemented, this trap also applies to DC CIGVAC, DC CIGDVAC, DC CGVAC, DC CGDVAC, DC CGVAP, and DC CGDVAP.

If FEAT_DPB2 and FEAT_MTE are implemented, this trap also applies to DC CGVADP and DC CGDVADP.

- **0b0** Execution of the specified instructions at EL0 using AArch64 is trapped.
- **0b1** This control does not cause any instructions to be trapped.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 1.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.
EE, bit [25]

Endianness of data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime.

0b0  Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are little-endian.

0b1  Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception levels higher than EL0, this bit is RES1.

The EE bit is permitted to be cached in a TLB.

When FEAT_VHE is implemented, and the value of HCR_EL2.\{E2H, TGE\} is \{1, 1\}, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

E0E, bit [24]

Endianness of data accesses at EL0.

0b0  Explicit data accesses at EL0 are little-endian.

0b1  Explicit data accesses at EL0 are big-endian.

If an implementation only supports Little-endian accesses at EL0, then this bit is RES0. This option is not permitted when SCTLR_EL1.EE is RES1.

If an implementation only supports Big-endian accesses at EL0, then this bit is RES1. This option is not permitted when SCTLR_EL1.EE is RES0.

This bit has no effect on the endianness of LDTR, LDTRH, LDTRSH, LDTRSW, STTR, and STTRH instructions executed at EL1.

When FEAT_VHE is implemented, and the value of HCR_EL2.\{E2H, TGE\} is \{1, 1\}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

SPAN, bit [23]

When FEAT_PAN is implemented:

Set Privileged Access Never, on taking an exception to EL1.

0b0  PSTATE.PAN is set to 1 on taking an exception to EL1.

0b1  The value of PSTATE.PAN is left unchanged on taking an exception to EL1.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

EIS, bit [22]

When FEAT_ExS is implemented:

Exception Entry is Context Synchronizing.

0b0  The taking of an exception to EL1 is not a context synchronizing event.

0b1  The taking of an exception to EL1 is a context synchronizing event.
When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is \{1,1\}, this bit has no effect on execution at EL0.

If SCTLR_EL1.EIS is set to 0b0:
- Indirect writes to ESR_EL1, FAR_EL1, SPSR_EL1, ELR_EL1 are synchronized on exception entry to EL1, so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.
- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL1.EIS:
- Changes to the PSTATE information on entry to EL1.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores and data processing instructions.
- Exit from Debug state.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES1.

**IESB, bit [21]**

*When FEAT_IESB is implemented:*
Implicit Error Synchronization event enable. Possible values are:
- 0b0  Disabled.
- 0b1  An implicit error synchronization event is added:
  - At each exception taken to EL1.
  - Before the operational pseudocode of each ERET instruction executed at EL1.

When the PE is in Debug state, the effect of this field is CONSTRAINED UNPREDICTABLE, and its Effective value might be 0 or 1 regardless of the value of the field. If the Effective value of the field is 1, then an implicit error synchronization event is added after each DCPSx instruction taken to EL1 and before each DRPS instruction executed at EL1, in addition to the other cases where it is added.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**TSCXT, bit [20]**

*When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:*
Trap EL0 Access to the SCXTNUM_EL0 register, when EL0 is using AArch64.
- 0b0  EL0 access to SCXTNUM_EL0 is not disabled by this mechanism.
- 0b1  EL0 access to SCXTNUM_EL0 is disabled, causing an exception to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1.

The value of SCXTNUM_EL0 is treated as 0.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is \{1,1\}, this bit has no effect on execution at EL0.
The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES1.

**WXN, bit [19]**

Write permission implies XN (Execute-never). For the EL1&0 translation regime, this bit can force all memory regions that are writable to be treated as XN.

- 0b0 This control has no effect on memory access permissions.
- 0b1 Any region that is writable in the EL1&0 translation regime is forced to XN for accesses from software executing at EL1 or EL0.

This bit applies only when SCTLR_EL1.M bit is set.

The WXN bit is permitted to be cached in a TLB.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

**nTWE, bit [18]**

Traps EL0 execution of WFE instructions to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states, reported using an ESR_ELx.EC value of 0x01.

When FEAT_WFxT is implemented, this trap also applies to the WFET instruction.

- 0b0 Any attempt to execute a WFE instruction at EL0 is trapped, if the instruction would otherwise have caused the PE to enter a low-power state.
- 0b1 This control does not cause any instructions to be trapped.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

**Note**

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

**Bit [17]**

Reserved, RES0.

**nTWI, bit [16]**

Traps EL0 execution of WFI instructions to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states, reported using an ESR_ELx.EC value of 0x01.
When \texttt{FEAT\_WFxT} is implemented, this trap also applies to the WFIT instruction.

- **0b0** Any attempt to execute a WFI instruction at EL0 is trapped, if the instruction would otherwise have caused the PE to enter a low-power state.
- **0b1** This control does not cause any instructions to be trapped.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

--- **Note** ---

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

---

When \texttt{FEAT\_VHE} is implemented, and the value of \texttt{HCR\_EL2.\{E2H, TGE\}} is \{1, 1\}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

**UCT, bit [15]**

Traps EL0 accesses to the \texttt{CTR\_EL0} to EL1, or to EL2 when it is implemented and enabled for the current Security state and \texttt{HCR\_EL2.TGE} is 1, from AArch64 state only, reported using an ESR\_ELx.EC value of \(0x18\).

- **0b0** Accesses to the \texttt{CTR\_EL0} from EL0 using AArch64 are trapped.
- **0b1** This control does not cause any instructions to be trapped.

When \texttt{FEAT\_VHE} is implemented, and the value of \texttt{HCR\_EL2.\{E2H, TGE\}} is \{1, 1\}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

**DZE, bit [14]**

Traps EL0 execution of DC ZVA instructions to EL1, or to EL2 when it is implemented and enabled for the current Security state and \texttt{HCR\_EL2.TGE} is 1, from AArch64 state only, reported using an ESR\_ELx.EC value of \(0x18\).

If \texttt{FEAT\_MTE} is implemented, this trap also applies to DC GVA and DC GZVA.

- **0b0** Any attempt to execute an instruction that this trap applies to at EL0 using AArch64 is trapped.
  
  Reading \texttt{DCZID\_EL0.DZP} from EL0 returns 1, indicating that the instructions this trap applies to are not supported.
- **0b1** This control does not cause any instructions to be trapped.

When \texttt{FEAT\_VHE} is implemented, and the value of \texttt{HCR\_EL2.\{E2H, TGE\}} is \{1, 1\}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

**EnDB, bit [13]**

\textbf{When \texttt{FEAT\_PAuth} is implemented:}

Controls enabling of pointer authentication (using the \texttt{APDBKey\_EL1} key) of instruction addresses in the EL1&0 translation regime.
For more information, see *Pointer authentication* on page D8-5164.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Pointer authentication (using the APDBKey_EL1 key) of data addresses is not enabled.</td>
</tr>
<tr>
<td>0b1</td>
<td>Pointer authentication (using the APDBKey_EL1 key) of data addresses is enabled.</td>
</tr>
</tbody>
</table>

---

**Note**

This field controls the behavior of the AddPACDB and AuthDB pseudocode functions. Specifically, when the field is 1, AddPACDB returns a copy of a pointer to which a pointer authentication code has been added, and AuthDB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**

Reserved, RES0.

---

**I, bit [12]**

Stage 1 instruction access Cacheability control, for accesses at EL0 and EL1:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>All instruction access to Stage 1 Normal memory from EL0 and EL1 are Stage 1 Non-cacheable.</td>
</tr>
<tr>
<td></td>
<td>If the value of SCTLR_EL1.M is 0, instruction accesses from stage 1 of the EL1&amp;0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.</td>
</tr>
<tr>
<td>0b1</td>
<td>This control has no effect on the Stage 1 Cacheability of instruction access to Stage 1 Normal memory from EL0 and EL1.</td>
</tr>
<tr>
<td></td>
<td>If the value of SCTLR_EL1.M is 0, instruction accesses from stage 1 of the EL1&amp;0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.</td>
</tr>
</tbody>
</table>

When the value of the HCR_EL2.DC bit is 1, then instruction access to Normal memory from EL0 and EL1 are Cacheable regardless of the value of the SCTLR_EL1.I bit.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is \{1, 1\}, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to 0.

---

**EOS, bit [11]**

*When FEAT_ExS is implemented:*

Exception Exit is Context Synchronizing.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>An exception return from EL1 is not a context synchronizing event</td>
</tr>
<tr>
<td>0b1</td>
<td>An exception return from EL1 is a context synchronizing event</td>
</tr>
</tbody>
</table>

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is \{1,1\}, this bit has no effect on execution at EL0.

If SCTLR_EL1.EOS is set to 0b0:

- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL1.EOS:

- The indirect write of the PSTATE and PC values from SPSR_EL1 and ELR_EL1 on exception return is synchronized.
• Behavior of accessing the banked copies of the stack pointer using the SP register name for
loads, stores and data processing instructions.
• Exit from Debug state.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL1, this field resets to an
architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES1.

**EnRCTX, bit [10]**

**When FEAT_SPECRES is implemented:**
Enable EL0 Access to the following instructions:
• AArch32 CFPRCTX, DVPRCTX and CPPRCTX instructions.
• AArch64 CFP RCTX, DVP RCT and CPP RCTX instructions.

0b0 EL0 access to these instructions is disabled, and these instructions are trapped to EL1,
or to EL2 when it is implemented and enabled for the current Security state and
HCR_EL2.TGE is 1.
0b1 EL0 access to these instructions is enabled.

When FEAT_VHE is implemented, and the value of HCR_EL2.\{E2H, TGE\} is \{1,1\}, this bit has
no effect on execution at EL0.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL1, this field resets to an
architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**UMA, bit [9]**

User Mask Access. Traps EL0 execution of MSR and MRS instructions that access the PSTATE.\{D,
A, I, F\} masks to EL1, or to EL2 when it is implemented and enabled for the current Security state
and HCR_EL2.TGE is 1, from AArch64 state only, reported using an ESR_ELx.EC value of 0x18.

0b0 Any attempt at EL0 using AArch64 to execute an MRS, MSR\{register\}, or MSR\{immediate\}
instruction that accesses the DAIF is trapped.
0b1 This control does not cause any instructions to be trapped.

When FEAT_VHE is implemented, and the value of HCR_EL2.\{E2H, TGE\} is \{1, 1\}, this bit has
no effect on execution at EL0.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL1, this field resets to an
architecturally UNKNOWN value.

**SED, bit [8]**

**When EL0 is capable of using AArch32:**
SETEND instruction disable. Disables SETEND instructions at EL0 using AArch32.

0b0 SETEND instruction execution is enabled at EL0 using AArch32.
0b1 SETEND instructions are UNDEFINED at EL0 using AArch32 and any attempt at EL0 to
access a SETEND instruction generates an exception to EL1, or to EL2 when it is
implemented and enabled for the current Security state and HCR_EL2.TGE is 1,
reported using an ESR_ELx.EC value of 0x00.

If the implementation does not support mixed-endian operation at any Exception level, this bit is
RES1.
When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is \{1, 1\}, this bit has no effect on execution at EL0. The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES1.

**ITD, bit [7]**

**When EL0 is capable of using AArch32:**

IT Disable. Disables some uses of IT instructions at EL0 using AArch32.

- **0b0** All IT instruction functionality is enabled at EL0 using AArch32.
- **0b1** Any attempt at EL0 using AArch32 to execute any of the following is UNDEFINED and generates an exception, reported using an ESR_ELx.EC value of 0x0, to EL1 or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1:
  - All encodings of the IT instruction with hw1[3:0]!=1000.
  - All encodings of the subsequent instruction with the following values for hw1:
    - 0b1xxxxxxxxxxxx: All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.
    - 0b101xxxxxxxxxxxx: All instructions in Miscellaneous 16-bit instructions on page F3-7303.
    - 0b1010xxxxxxxxxxxx: ADD Rd, #imm
    - 0b0101xxxxxxxxxxxx: LDR Rd, [PC, #imm]
    - 0b01001xxxxxxxxxxxx: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers unpredictable cases with BLX Rn.
    - 0b010001xxxxxxxxxxxx: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers unpredictable cases with BLX Rn.

These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block. It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:

- A 16-bit instruction, that can only be followed by another 16-bit instruction.
- The first half of a 32-bit instruction.

This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED. An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If an instruction in an active IT block that would be disabled by this field sets this field to 1 then behavior is CONSTRAINED UNPREDICTABLE. For more information, see Changes to an ITD control by an instruction in an IT block on page E1-7138.

ITD is optional, but if it is implemented in the SCTLR_EL1 then it must also be implemented in the SCTLR_EL2, HSCTLR, and SCTLR.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is \{1, 1\}, this bit has no effect on execution at EL0. The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES1.
nAA, bit [6]

When FEAT_LSE2 is implemented:

Non-aligned access. This bit controls generation of Alignment faults at EL1 and EL0 under certain conditions.

The following instructions generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes for access:

- LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH.
- STLLR, STLLRH, STLR, STLRH, STLUR, and STLURH.

0b0 Unaligned accesses by the specified instructions generate an Alignment fault.
0b1 This control does not generate Alignment faults.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

CP15BEN, bit [5]

When EL0 is capable of using AArch32:

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==0b1111) encoding space from EL0:

0b0 EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED and generates an exception to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. The exception is reported using an ESR_ELx.EC value of 0x00.
0b1 EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the SCTLR_EL1 then it must also be implemented in the SCTLR_EL2, HSCTLR, and SCTLR.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

SA0, bit [4]

SP Alignment check enable for EL0. When set to 1, if a load or store instruction executed at EL0 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see SP alignment checking on page D1-4668.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.
SA, bit [3]
SP Alignment check enable. When set to 1, if a load or store instruction executed at EL1 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see SP alignment checking on page D1-4668.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

C, bit [2]
Stage 1 Cacheability control, for data accesses.

0b0 All data access to Stage 1 Normal memory from EL0 and EL1, and all Normal memory accesses from unified cache to the EL1&0 Stage 1 translation tables, are treated as Stage 1 Non-cacheable.

0b1 This control has no effect on the Stage 1 Cacheability of:
- Data access to Normal memory from EL0 and EL1.
- Normal memory accesses to the EL1&0 Stage 1 translation tables.

When the value of the HCR_EL2.DC bit is 1, the PE ignores SCTLR.C. This means that Non-secure EL0 and Non-secure EL1 data accesses to Normal memory are Cacheable.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL1, this field resets to 0.

A, bit [1]
Alignment check enable. This is the enable bit for Alignment fault checking at EL1 and EL0.

0b0 Alignment fault checking disabled when executing at EL1 or EL0. Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.

0b1 Alignment fault checking enabled when executing at EL1 or EL0. All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

If FEAT_MOPS is implemented, SETG* instructions have an alignment check regardless of the value of the A bit.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL1, this field resets to an architecturally UNKNOWN value.

M, bit [0]
MMU enable for EL1&0 stage 1 address translation.

0b0 EL1&0 stage 1 address translation disabled. See the SCTLR_EL1.I field for the behavior of instruction accesses to Normal memory.

0b1 EL1&0 stage 1 address translation enabled.
If the value of \texttt{HCR\_EL2.\{DC, TGE\}} is not \{0, 0\} then in Non-secure state the PE behaves as if the value of the \texttt{SCTLR\_EL1.M} field is 0 for all purposes other than returning the value of a direct read of the field.

When \texttt{FEAT\_VHE} is implemented, and the value of \texttt{HCR\_EL2.\{E2H, TGE\}} is \{1, 1\}, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL1, this field resets to 0.

### Accessing SCTLR\_EL1

When \texttt{HCR\_EL2.E2H} is 1, without explicit synchronization, access from EL3 using the mnemonic \texttt{SCTLR\_EL1} or \texttt{SCTLR\_EL12} are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <xt>, SCTLR\_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if \texttt{PSTATE.EL} == EL0 then
    UNDEFINED;
elsif \texttt{PSTATE.EL} == EL1 then
    if \texttt{EL2Enabled()} \&\& \texttt{HCR\_EL2.TVM == '1'} then
        \texttt{AArch64.SystemAccessTrap(EL2, 0x18)};
    elsif \texttt{EL2Enabled()} \&\& (\texttt{HaveEL(EL3)} \&\& \texttt{SCR\_EL3.FGTEn == '1')} \&\& \texttt{HFGTR\_EL2.SCTLR\_EL1 == '1'} then
        \texttt{AArch64.SystemAccessTrap(EL2, 0x18)};
    elsif \texttt{EL2Enabled()} \&\& \texttt{HCR\_EL2.<NV2,NV1,NV> == '111'} then
        \texttt{X[t, 64] = NVMem[0x110]};
    else
        \texttt{X[t, 64] = SCTLR\_EL1};
    end
elsif \texttt{PSTATE.EL} == EL2 then
    if \texttt{HCR\_EL2.E2H == '1'} then
        \texttt{X[t, 64] = SCTLR\_EL2};
    else
        \texttt{X[t, 64] = SCTLR\_EL1};
    end
elsif \texttt{PSTATE.EL} == EL3 then
    \texttt{X[t, 64] = SCTLR\_EL1};

**MSR SCTLR\_EL1, <xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if \texttt{PSTATE.EL} == EL0 then
    UNDEFINED;
elsif \texttt{PSTATE.EL} == EL1 then
    if \texttt{EL2Enabled()} \&\& \texttt{HCR\_EL2.TVM == '1'} then
        \texttt{AArch64.SystemAccessTrap(EL2, 0x18)};
    elsif \texttt{EL2Enabled()} \&\& (\texttt{HaveEL(EL3)} \&\& \texttt{SCR\_EL3.FGTEn == '1')} \&\& \texttt{HFGTR\_EL2.SCTLR\_EL1 == '1'} then
        \texttt{AArch64.SystemAccessTrap(EL2, 0x18)};
    elsif \texttt{EL2Enabled()} \&\& \texttt{HCR\_EL2.<NV2,NV1,NV> == '111'} then
        \texttt{NVMem[0x110] = X[t, 64]};
    else
        \texttt{SCTLR\_EL1 = X[t, 64]};
    end
elsif \texttt{PSTATE.EL} == EL2 then
if HCR_EL2.E2H == '1' then
    SCTLR_EL2 = X[t, 64];
else
    SCTLR_EL1 = X[t, 64];
elsif PSTATE_EL == EL3 then
    SCTLR_EL1 = X[t, 64];
end if

MRS <X>, SCTLR_EL12

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '101' then
        X[t, 64] = NVMem[0x110];
    elsif EL2Enabled() & HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    end if
elsif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' then
        X[t, 64] = SCTLR_EL1;
    else
        UNDEFINED;
    end if
elsif PSTATE_EL == EL3 then
    if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.E2H == '1' then
        X[t, 64] = SCTLR_EL1;
    else
        UNDEFINED;
    end if
end if

MSR SCTLR_EL12, <X>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x110] = X[t, 64];
    elsif EL2Enabled() & HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    end if
elsif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' then
        SCTLR_EL1 = X[t, 64];
    else
        UNDEFINED;
    end if
elsif PSTATE_EL == EL3 then
    if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.E2H == '1' then
        SCTLR_EL1 = X[t, 64];
    else
        UNDEFINED;
    end if
D17.2.119 SCTLR_EL2, System Control Register (EL2)

The SCTLR_EL2 characteristics are:

Purpose

Provides top level control of the system, including its memory system, at EL2.
When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is \{1, 1\}, these controls apply also to execution at EL0.

Configurations

AArch64 System register SCTLR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HSCTLR[31:0].
If EL2 is not implemented, this register is RES0 from EL3.
This register has no effect if EL2 is not enabled in the current Security state.

Attributes

SCTLR_EL2 is a 64-bit register.

Field descriptions

TIDCP, bit [63]

*When FEAT_TIDCP1 is implemented and HCR_EL2.E2H == 1:*

Trap IMPLEMENTATION DEFINED functionality. Traps EL0 accesses to the encodings reserved for IMPLEMENTATION DEFINED functionality to EL2.

0b0 No instructions accessing the System register or System instruction spaces are trapped by this mechanism.

0b1 If HCR_EL2.TGE==0, no instructions accessing the System register or System instruction spaces are trapped by this mechanism.
If HCR_EL2.TGE==1, instructions accessing the following System register or System instruction spaces are trapped to EL2 by this mechanism:

- In AArch64 state, EL0 access to the encodings in the following reserved encoding spaces are trapped and reported using EC syndrome 0x18:
  - IMPLEMENTATION DEFINED System instructions, which are accessed using SYS and SYSL, with CRn == {11, 15}.
  - IMPLEMENTATION DEFINED System registers, which are accessed using MRS and MSR with the S3_<op1>_<Cn>_<Cm>_<op2> register name.

- In AArch32 state, EL0 MCR and MRC access to the following encodings are trapped and reported using EC syndrome 0x03:
  - All coproc==p15, CRn==c9, opc1 == {0-7}, CRm == {c0-c2, c5-c8}, opc2 == {0-7}.
  - All coproc==p15, CRn==c10, opc1 == {0-7}, CRm == {c0, c1, c4, c8}, opc2 == {0-7}.
  - All coproc==p15, CRn==c11, opc1=={0-7}, CRm == {c0-c8, c15}, opc2 == {0-7}.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**SPINTMASK, bit [62]**

When FEAT_NMI is implemented:

SP Interrupt Mask enable. When SCTLR_EL2.NMI is 1, controls whether PSTATE.SP acts as an interrupt mask, and controls the value of PSTATE.ALLINT on taking an exception to EL2.

**0b0**
Does not cause PSTATE.SP to mask interrupts.
PSTATE.ALLINT is set to 1 on taking an exception to EL2.

**0b1**
When PSTATE.SP is 1 and execution is at EL2, an IRQ or FIQ interrupt that is targeted to EL2 is masked regardless of any denotation of Superpriority.
PSTATE.ALLINT is set to 0 on taking an exception to EL2.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**NMI, bit [61]**

When FEAT_NMI is implemented:

Non-maskable Interrupt enable.

**0b0**
This control does not affect interrupt masking behavior.

**0b1**
This control enables all of the following:
- The use of the PSTATE.ALLINT interrupt mask.
- IRQ and FIQ interrupts to have Superpriority as an additional attribute.
- PSTATE.SP to be used as an interrupt mask.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Otherwise:**
Reserved, RES0.
EnTP2, bit [60]

**When FEAT_SME is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:**

Traps instructions executed at EL0 that access TPIDR_EL0 to EL2 when EL2 is implemented and enabled for the current Security state. The exception is reported using ESR_ELx.EC value 0x18.

- **0b0** This control causes execution of these instructions at EL0 to be trapped.
- **0b1** This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

**When FEAT_SME is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:**

IGNORED.

**Otherwise:**

Reserved, RES0.

Bits [59:58]

Reserved, RES0.

EPAN, bit [57]

**When FEAT_PAN3 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:**

Enhanced Privileged Access Never. When PSTATE.PAN is 1, determines whether an EL2 data access to a page with EL0 instruction access permission generates a Permission fault as a result of the Privileged Access Never mechanism.

- **0b0** No additional Permission faults are generated by this mechanism.
- **0b1** An EL2 data access to a page with stage 1 EL0 data access permission or stage 1 EL0 instruction access permission generates a Permission fault.

Any speculative data accesses that would generate a Permission fault if the accesses were not speculative will not cause an allocation into a cache.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

**When FEAT_PAN3 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:**

IGNORED.

**Otherwise:**

Reserved, RES0.

EnALS, bit [56]

**When FEAT_LS64 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:**

Traps execution of an LD64B or ST64B instruction at EL0 to EL2.

- **0b0** Execution of an LD64B or ST64B instruction at EL0 is trapped to EL2.
- **0b1** This control does not cause any instructions to be trapped.

A trap of an LD64B or ST64B instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000002.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

**When FEAT_LS64 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:**

IGNORED.
Otherwise:
Reserved, RES0.

EnAS0, bit [55]
When FEAT_LS64_ACCDATA is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:
Traps execution of an ST64BV0 instruction at EL0 to EL2.
0b0 Execution of an ST64BV0 instruction at EL0 is trapped to EL2.
0b1 This control does not cause any instructions to be trapped.
A trap of an ST64BV0 instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000001.
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

When FEAT_LS64_ACCDATA is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:
IGNORED.
Otherwise:
Reserved, RES0.

EnASR, bit [54]
When FEAT_LS64_V is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:
Traps execution of an ST64BV instruction at EL0 to EL2.
0b0 Execution of an ST64BV instruction at EL0 is trapped to EL2.
0b1 This control does not cause any instructions to be trapped.
A trap of an ST64BV instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000000.
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

When FEAT_LS64_V is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:
IGNORED.
Otherwise:
Reserved, RES0.

TME, bit [53]
When FEAT_TME is implemented:
Enables the Transactional Memory Extension at EL2.
0b0 Any attempt to execute a TSTART instruction at EL2 is trapped, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART instructions to be UNDEFINED at EL2.
0b1 This control does not cause any TSTART instruction to be trapped.
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.
TME0, bit [52]

*When FEAT_TME is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:*

Enables the Transactional Memory Extension at EL0.

0b0  Any attempt to execute a TSTART instruction at EL0 is trapped to EL2, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART instructions to be UNDEFINED at EL0.

0b1  This control does not cause any TSTART instruction to be trapped.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

*When FEAT_TME is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:*

_IGNORED._

*Otherwise:*

Reserved, RES0.

TMT, bit [51]

*When FEAT_TME is implemented:*

Forces a trivial implementation of the Transactional Memory Extension at EL2.

0b0  This control does not cause any TSTART instruction to fail.

0b1  When the TSTART instruction is executed at EL2, the transaction fails with a TRIVIAL failure cause.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

TMT0, bit [50]

*When FEAT_TME is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:*

Forces a trivial implementation of the Transactional Memory Extension at EL0.

0b0  This control does not cause any TSTART instruction to fail.

0b1  When the TSTART instruction is executed at EL0, the transaction fails with a TRIVIAL failure cause.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

*When FEAT_TME is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:*

_IGNORED._

*Otherwise:*

Reserved, RES0.

TWEDEL, bits [49:46]

*When FEAT_TWED is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:*

TWE Delay. A 4-bit unsigned number that, when SCTLR_EL2.TWEDEn is 1, encodes the minimum delay in taking a trap of WFE caused by SCTLR_EL2.nTWE as $2^{(TWEDEL + 8)}$ cycles.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.
When FEAT_TWED is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:
IGNORED.

Otherwise:
Reserved, RES0.

TWEDEn, bit [45]

When FEAT_TWED is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:
TWE Delay Enable. Enables a configurable delayed trap of the WFE instruction caused by
SCTRL_EL2.nTWE.

0b0 The delay for taking a WFE trap is IMPLEMENTATION DEFINED.
0b1 The delay for taking a WFE trap is at least the number of cycles defined in
SCTRL_EL2.TWEDEL.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to an
architecturally UNKNOWN value.

When FEAT_TWED is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:
IGNORED.

Otherwise:
Reserved, RES0.

DSSBS, bit [44]

When FEAT_SSBS is implemented:
Default PSTATE.SSBS value on Exception Entry.

0b0 PSTATE.SSBS is set to 0 on an exception to EL2.
0b1 PSTATE.SSBS is set to 1 on an exception to EL2.

The reset behavior of this field is:
• On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:
Reserved, RES0.

ATA, bit [43]

When FEAT_MTE2 is implemented:
Allocation Tag Access in EL2.

When SCR_EL3.ATA is 1, controls access to Allocation Tags and Tag Check operations in EL2.

0b0 Access to Allocation Tags is prevented at EL2.
Memory accesses at EL2 are not subject to a Tag Check operation.
0b1 This control does not prevent access to Allocation Tags at EL2.
Tag Checked memory accesses at EL2 are subject to a Tag Check operation.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to an
architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

ATA0, bit [42]

When FEAT_MTE2 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:
Allocation Tag Access in EL0.
When SCR_EL3.ATA is 1, controls access to Allocation Tags and Tag Check operations in EL0.

0b0  Access to Allocation Tags is prevented at EL0.
      Memory accesses at EL0 are not subject to a Tag Check operation.

0b1  This control does not prevent access to Allocation Tags at EL0.
      Tag Checked memory accesses at EL0 are subject to a Tag Check operation.

--- Note ---
Software may change this control bit on a context switch.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

When FEAT_MTE2 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:
IGNORED.

Otherwise:
Reserved, RES0.

TCF, bits [41:40]

When FEAT_MTE2 is implemented:
Tag Check Fault in EL2. Controls the effect of Tag Check Faults due to Loads and Stores in EL2.

0b00  Tag Check Faults have no effect on the PE.
0b01  Tag Check Faults cause a synchronous exception.
0b10  Tag Check Faults are asynchronously accumulated.
0b11  When FEAT_MTE3 is implemented:
      Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.

If FEAT_MTE3 is not implemented, the value 0b11 is reserved.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

TCF0, bits [39:38]

When FEAT_MTE2 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:
Tag Check Fault in EL0. Controls the effect of Tag Check Faults due to Loads and Stores in EL0.

0b00  Tag Check Faults have no effect on the PE.
0b01  Tag Check Faults cause a synchronous exception.
0b10  Tag Check Faults are asynchronously accumulated.
0b11  When FEAT_MTE3 is implemented:
      Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.

If FEAT_MTE3 is not implemented, the value 0b11 is reserved.

--- Note ---
Software may change this control bit on a context switch.
The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

*When FEAT_MTE2 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:*

- IGNORED.

*Otherwise:*

  Reserved, RES0.

**ITFSB, bit [37]**

*When FEAT_MTE2 is implemented:*

  When synchronous exceptions are not being generated by Tag Check Faults, this field controls whether on exception entry into EL2, all Tag Check Faults due to instructions executed before exception entry, that are reported asynchronously, are synchronized into TFSRE0_EL1, TFSR_EL1 and TFSR_EL2 registers.

- **0b0**: Tag Check Faults are not synchronized on entry to EL2.
- **0b1**: Tag Check Faults are synchronized on entry to EL2.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

  Reserved, RES0.

**BT, bit [36]**

*When FEAT_BTI is implemented:*

  PAC Branch Type compatibility at EL2.

  When HCR_EL2.{E2H, TGE} == {1, 1}, this bit is named BT1.

- **0b0**: When the PE is executing at EL2, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == 0b11.
- **0b1**: When the PE is executing at EL2, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == 0b11.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

  Reserved, RES0.

**BT0, bit [35]**

*When FEAT_BTI is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:*

  PAC Branch Type compatibility at EL0.

- **0b0**: When the PE is executing at EL0, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == 0b11.
- **0b1**: When the PE is executing at EL0, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == 0b11.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

*When FEAT_BTI is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:*

  IGNORED.
**Otherwise:**

Reserved, RES0.

**Bit [34]**

Reserved, RES0.

**MSCEn, bit [33]**

When FEAT_MOPS is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:

Memory Copy and Memory Set instructions Enable. Enables execution of the Memory Copy and Memory Set instructions at EL0.

- **0b0** Execution of the Memory Copy and Memory Set instructions is UNDEFINED at EL0.
- **0b1** This control does not cause any instructions to be UNDEFINED.

When FEAT_MOPS is implemented and HCR_EL2.{E2H, TGE} is not {1, 1}, the Effective value of this bit is 0b1.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

When FEAT_MOPS is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:

IGNORED.

**Otherwise:**

Reserved, RES0.

**CMOW, bit [32]**

When FEAT_CMOW is implemented and HCR_EL2.E2H == 1:

Controls cache maintenance instruction permission for the following instructions executed at EL0.

- IC IVAU, DC CIVAC, DC CIGDVAC and DC CIGVAC.

- **0b0** These instructions executed at EL0 with stage 1 read permission, but without stage 1 write permission, do not generate a stage 1 permission fault.

- **0b1** If enabled as a result of SCTLR_EL2.UCI==1, these instructions executed at EL0 with stage 1 read permission, but without stage 1 write permission, generate a stage 1 permission fault.

When HCR_EL2.TGE is 0, this bit has no effect on execution at EL0.

For this control, stage 1 has write permission if all of the following apply:

- AP[2] is 0 or DBM is 1 in the stage 1 descriptor.
- Where APTable is in use, APTable[1] is 0 for all levels of the translation table.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**EnIA, bit [31]**

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APIAKey_EL1 key) of instruction addresses in the EL2 or EL2&0 translation regime.

For more information, see Pointer authentication on page D8-5164.

- **0b0** Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is not enabled.

Reserved, RES0.
0b1  Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is enabled.

——— Note ————
This field controls the behavior of the AddPACIA and AuthIA pseudocode functions. Specifically, when the field is 1, AddPACIA returns a copy of a pointer to which a pointer authentication code has been added, and AuthIA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

——— Note ————
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EnIB, bit [30]

When FEAT_PAuth is implemented:
Controls enabling of pointer authentication (using the APIBKey_EL1 key) of instruction addresses in the EL2 or EL2&0 translation regime.
For more information, see Pointer authentication on page D8-5164.
0b0  Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is not enabled.
0b1  Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is enabled.

——— Note ————
This field controls the behavior of the AddPACIB and AuthIB pseudocode functions. Specifically, when the field is 1, AddPACIB returns a copy of a pointer to which a pointer authentication code has been added, and AuthIB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

——— Note ————
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

LSMAOE, bit [29]

When FEAT_LSMAOC is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:
Load Multiple and Store Multiple Atomicity and Ordering Enable.
0b0  For all memory accesses at EL0, A32 and T32 Load Multiple and Store Multiple can have an interrupt taken during the sequence memory accesses, and the memory accesses are not required to be ordered.
0b1  The ordering and interrupt behavior of A32 and T32 Load Multiple and Store Multiple at EL0 is as defined for Armv8.0.
This bit is permitted to be cached in a TLB.
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

When FEAT_LSMAOC is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:
IGNORED.
Otherwise:
    Reserved, RES1.

nTLSMD, bit [28]
When FEAT_LSMAOC is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:

No Trap Load Multiple and Store Multiple to Device-nGRE/Device-nGnRE/Device-nGnRnE memory.

0b0    All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are trapped and generate a stage 1 Alignment fault.

0b1    All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are not trapped.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:
    • On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

When FEAT_LSMAOC is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:
    IGNORED.

Otherwise:
    Reserved, RES1.

EnDA, bit [27]
When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APDAKey_EL1 key) of instruction addresses in the EL2 or EL2&0 translation regime.

For more information, see Pointer authentication on page D8-5164.

0b0    Pointer authentication (using the APDAKey_EL1 key) of data addresses is not enabled.

0b1    Pointer authentication (using the APDAKey_EL1 key) of data addresses is enabled.

Note
This field controls the behavior of the AddPACDA and AuthDA pseudocode functions. Specifically, when the field is 1, AddPACDA returns a copy of a pointer to which a pointer authentication code has been added, and AuthDA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:
    • On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:
    Reserved, RES0.

UCI, bit [26]
When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:

Traps execution of cache maintenance instructions at EL0 to EL2, from AArch64 state only. This applies to DC CVAU, DC CIVAC, DC CVAC, DC CVAP, and IC IVAU.

If FEAT_DPB2 is implemented, this trap also applies to DC CVADP.

If FEAT_MTE is implemented, this trap also applies to DC CIGVAC, DC CIGDVAC, DC CGVAC, DC CGDVAC, DC CGVAP, and DC CGDVAP.
If FEAT_DPB2 and FEAT_MTE are implemented, this trap also applies to DC CGVADP and DC CGDVADP.

0b0 Any attempt to execute an instruction that this trap applies to at EL0 using AArch64 is trapped to EL2.

0b1 This control does not cause any instructions to be trapped.

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 1.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:

IGNORED.

Otherwise:

Reserved, RES0.

```
EE, bit [25]
```

Endianness of data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL1&0 translation regime.

0b0 Explicit data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL1&0 translation regime are little-endian.

0b1 Explicit data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL1&0 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception levels higher than EL0, this bit is RES1.

The EE bit is permitted to be cached in a TLB.

The reset behavior of this field is:
- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

```
E0E, bit [24]
```

Endianness of data accesses at EL0.

0b0 Explicit data accesses at EL0 are little-endian.

0b1 Explicit data accesses at EL0 are big-endian.

If an implementation only supports Little-endian accesses at EL0, then this bit is RES0. This option is not permitted when SCTLR_EL1.EE is RES1.

If an implementation only supports Big-endian accesses at EL0, then this bit is RES1. This option is not permitted when SCTLR_EL1.EE is RES0.

This bit has no effect on the endianness of LDTR, LDTRH, LDTRSH, LDTRSW, STTR, and STTRH instructions executed at EL1.
The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

**When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:**

IGNORED.

**Otherwise:**

Reserved, RES0.

**SPAN, bit [23]**

**When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:**

Set Privileged Access Never, on taking an exception to EL2.

- 0b0 PSTATE.PAN is set to 1 on taking an exception to EL2.
- 0b1 The value of PSTATE.PAN is left unchanged on taking an exception to EL2.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

**When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:**

IGNORED.

**Otherwise:**

Reserved, RES1.

**EIS, bit [22]**

**When FEAT_ExS is implemented:**

Exception entry is a context synchronization event.

- 0b0 The taking of an exception to EL2 is not a context synchronization event.
- 0b1 The taking of an exception to EL2 is a context synchronization event.

If SCTLR_EL2.EIS is set to 0b0:

- Indirect writes to ESR_EL2, FAR_EL2, SPSR_EL2, ELR_EL2, and HPFAR_EL2 are synchronized on exception entry to EL2, so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.

- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.

- Exception Catch debug events are synchronous debug events.

- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL2.EIS:

- Changes to the PSTATE information on entry to EL2.

- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores, and data processing instructions.

- Exit from Debug state.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES1.
IESB, bit [21]

When **FEAT_IESB is implemented**:
- Implicit Error Synchronization event enable.
  - 0b0: Disabled.
  - 0b1: An implicit error synchronization event is added:
    - At each exception taken to EL2.
    - Before the operational pseudocode of each **ERET** instruction executed at EL2.

When the PE is in Debug state, the effect of this field is **CONSTRAINED UNPREDICTABLE**, and its Effective value might be 0 or 1 regardless of the value of the field. If the Effective value of the field is 1, then an implicit error synchronization event is added after each **DCPSx** instruction taken to EL2 and before each **DRPS** instruction executed at EL2, in addition to the other cases where it is added.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**
- Reserved, RES0.

TSCXT, bit [20]

When (FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented), HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:
- Trap EL0 Access to the SCXTNUM_EL0 register, when EL0 is using AArch64.
  - 0b0: EL0 access to SCXTNUM_EL0 is not disabled by this mechanism.
  - 0b1: EL0 access to SCXTNUM_EL0 is disabled, causing an exception to EL2, and the SCXTNUM_EL0 value is treated as 0.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally **UNKNOWN** value.

When FEAT_CSV2_2 is not implemented, FEAT_CSV2_1p2 is not implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:
- Reserved, RES1.

When (FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented), HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:
- IGNORED.

**Otherwise:**
- Reserved, RES0.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL2 or EL2&0 translation regime, this bit can force all memory regions that are writable to be treated as XN.
- 0b0: This control has no effect on memory access permissions.
- 0b1: Any region that is writable in the EL2 or EL2&0 translation regime is forced to XN for accesses from software executing at EL2.

This bit applies only when SCTLR_EL2.M bit is set.

The WXN bit is permitted to be cached in a TLB.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally **UNKNOWN** value.
nTWE, bit [18]

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:

Traps execution of WFE instructions at EL0 to EL2, from both Execution states.

0b0 Any attempt to execute a WFE instruction at EL0 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state.

0b1 This control does not cause any instructions to be trapped.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

--- Note ---

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:

IGNORED.

Otherwise:

Reserved, RES1.

Bit [17]

Reserved, RES0.

nTWI, bit [16]

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:

Traps execution of WFI instructions at EL0 to EL2, from both Execution states.

0b0 Any attempt to execute a WFI instruction at EL0 is trapped EL2, if the instruction would otherwise have caused the PE to enter a low-power state.

0b1 This control does not cause any instructions to be trapped.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

--- Note ---

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:

IGNORED.

Otherwise:

Reserved, RES1.
UCT, bit [15]

When $HCR\_EL2.2H == 1$ and $HCR\_EL2.TGE == 1$:

Traps EL0 accesses to the CTR_EL0 to EL2, from AArch64 state only.

- $0b0$ Accesses to the CTR_EL0 from EL0 using AArch64 are trapped to EL2.
- $0b1$ This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

When $HCR\_EL2.2H == 1$ and $HCR\_EL2.TGE == 0$:

IGNORED.

Otherwise:

Reserved, RES0.

DZE, bit [14]

When $HCR\_EL2.2H == 1$ and $HCR\_EL2.TGE == 1$:

Traps execution of DC ZVA instructions at EL0 to EL2, from AArch64 state only.

If FEAT_MTE is implemented, this trap also applies to DC GVA and DC GZVA.

- $0b0$ Any attempt to execute an instruction that this trap applies to at EL0 using AArch64 is trapped to EL2. Reading DCZID_EL0.DZP from EL0 returns 1, indicating that the instructions that this trap applies to are not supported.
- $0b1$ This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

When $HCR\_EL2.2H == 1$ and $HCR\_EL2.TGE == 0$:

IGNORED.

Otherwise:

Reserved, RES0.

EnDB, bit [13]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APDBKey_EL1 key) of instruction addresses in the EL2 or EL2&0 translation regime.

For more information, see Pointer authentication on page D8-5164.

- $0b0$ Pointer authentication (using the APDBKey_EL1 key) of data addresses is not enabled.
- $0b1$ Pointer authentication (using the APDBKey_EL1 key) of data addresses is enabled.

Note

This field controls the behavior of the AddPACDB and AuthDB pseudocode functions. Specifically, when the field is 1, AddPACDB returns a copy of a pointer to which a pointer authentication code has been added, and AuthDB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
I, bit [12]

Instruction access Cacheability control, for accesses at EL2 and, when EL2 is enabled in the current Security state and \( \text{HCR\_EL2.\{E2H,TGE\}} = \{1,1\}, \text{EL0} \).

0b0  
All instruction accesses to Normal memory from EL2 are Non-cacheable for all levels of instruction and unified cache.
When EL2 is enabled in the current Security state and \( \text{HCR\_EL2.\{E2H, TGE\}} = \{1, 1\} \), all instruction accesses to Normal memory from EL0 are Non-cacheable for all levels of instruction and unified cache.
If SCTLR\_EL2.M is 0, instruction accesses from stage 1 of the EL2 or EL2&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

0b1  
This control has no effect on the Cacheability of instruction access to Normal memory from EL2 and, when EL2 is enabled in the current Security state and \( \text{HCR\_EL2.\{E2H, TGE\}} = \{1, 1\} \), instruction access to Normal memory from EL0.
If the value of SCTLR\_EL2.M is 0, instruction accesses from stage 1 of the EL2 or EL2&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the EL3 translation regime.
When EL2 is disabled in the current Security state or \( \text{HCR\_EL2.\{E2H, TGE\}} \neq \{1,1\} \), this bit has no effect on the EL1&0 translation regime.
The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

EOS, bit [11]

*When FEAT\_ExS is implemented:*

Exception exit is a context synchronization event.

0b0  
An exception return from EL2 is not a context synchronization event.

0b1  
An exception return from EL2 is a context synchronization event.

If SCTLR\_EL2.EOS is set to 0b0:
- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.
The following are not affected by the value of SCTLR\_EL2.EOS:
- The indirect write of the PSTATE and PC values from SPSR\_EL2 and ELR\_EL2 on exception return is synchronized.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores, and data processing instructions.
- Exit from Debug state.
The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES1.

EnRCTX, bit [10]

*When FEAT\_SPECRES is implemented, HCR\_EL2.E2H == 1 and HCR\_EL2.TGE == 1:*

Enable EL0 Access to the following instructions:

- AArch32 CFPRCTX, DVPRCTX and CPPRCTX instructions.
• AArch64 CFP RCTX, DVP RCT and CPP RCTX instructions.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>EL0 access to these instructions is disabled, and these instructions are trapped to EL1.</td>
</tr>
<tr>
<td>0b1</td>
<td>EL0 access to these instructions is enabled.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

When `FEAT_SPECRES` is implemented, `HCR_EL2.E2H == 1` and `HCR_EL2.TGE == 0`:
IGNORED.

Otherwise:
Reserved, RES0.

Bit [9]
Reserved, RES0.

SED, bit [8]

When EL0 is capable of using AArch32, `HCR_EL2.E2H == 1` and `HCR_EL2.TGE == 1`:
SETEND instruction disable. Disables SETEND instructions at EL0 using AArch32.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>SETEND instruction execution is enabled at EL0 using AArch32.</td>
</tr>
<tr>
<td>0b1</td>
<td>SETEND instructions are UNDEFINED at EL0 using AArch32.</td>
</tr>
</tbody>
</table>

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

When EL0 can only use AArch64, `HCR_EL2.E2H == 1` and `HCR_EL2.TGE == 1`:
Reserved, RES1.

When EL0 is capable of using AArch32, `HCR_EL2.E2H == 1` and `HCR_EL2.TGE == 0`:
IGNORED.

Otherwise:
Reserved, RES0.

ITD, bit [7]

When EL0 is capable of using AArch32, `HCR_EL2.E2H == 1` and `HCR_EL2.TGE == 1`:
IT Disable. Disables some uses of IT instructions at EL0 using AArch32.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>All IT instruction functionality is enabled at EL0 using AArch32.</td>
</tr>
<tr>
<td>0b1</td>
<td>Any attempt at EL0 using AArch32 to execute any of the following is UNDEFINED:</td>
</tr>
</tbody>
</table>
• All encodings of the IT instruction with hw1[3:0]!=1000. |
• All encodings of the subsequent instruction with the following values for hw1:
  — 0b11xxxxxxxxxxxxxx: All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM. |
  — 0b101xxxxxxxxxxxx: All instructions in Miscellaneous 16-bit instructions on page F3-7303. |
  — 0b10100xxxxxxxxxxx: ADD Rd, PC, #imm |
  — 0b01001xxxxxxxxxxx: LDR Rd, [PC, #imm] |
  — 0b01000xxxxx111111: ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC. |
  — 0b010001xx1xxx111111: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers UNPREDICTABLE cases with BLX Rn. |
These instructions are always **UNDEFINED**, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block. It is **IMPLEMENTATION DEFINED** whether the IT instruction is treated as:

- A 16-bit instruction, that can only be followed by another 16-bit instruction.
- The first half of a 32-bit instruction.

This means that, for the situations that are **UNDEFINED**, either the second 16-bit instruction or the 32-bit instruction is **UNDEFINED**.

An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If an instruction in an active IT block that would be disabled by this field sets this field to 1 then behavior is **CONSTRAINED UNPREDICTABLE**. For more information see Changes to an ITD control by an instruction in an IT block on page E1-7138.

ITD is optional, but if it is implemented in the SCTLR_EL2 then it must also be implemented in the SCTLR_EL1, HSCTLR, and SCTLR.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally **UNKNOWN** value.

When an implementation does not implement ITD, access to this field is RAZ/WI.

**When EL0 can only use AArch64, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:**

Reserved, RES1.

**When EL0 is capable of using AArch32, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:**

IGNORED.

**Otherwise:**

Reserved, RES0.

**nAA, bit [6]**

**When FEAT_LSE2 is implemented:**

Non-aligned access. This bit controls generation of Alignment faults under certain conditions at EL2, and, when EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}, EL0.

The following instructions generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes for access:

- LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH.
- STLLR, STLRRH, STLR, STLRH, STLUR, and STLURH

0b0 Unaligned accesses by the specified instructions generate an Alignment fault.

0b1 Unaligned accesses by the specified instructions do not generate an Alignment fault.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**

Reserved, RES0.

**CP15BEN, bit [5]**

**When EL0 is capable of using AArch32, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:**

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==0b1111) encoding space from EL0:

0b0 EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is **UNDEFINED**.
EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the SCTLR_EL2 then it must also be implemented in the SCTLR_EL1, HSCTLR, and SCTLR.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

When EL0 can only use AArch64, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:

Reserved, RES0.

When EL0 is capable of using AArch32, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0:

IGNORED.

Otherwise:

Reserved, RES1.

SA0, bit [4]

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:

SP Alignment check enable for EL0. When set to 1, if a load or store instruction executed at EL0 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see SP alignment checking on page D1-4668.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL2 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see SP alignment checking on page D1-4668.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Data access Cacheability control, for accesses at EL2 and, when EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}, EL0

0b0 The following are Non-cacheable for all levels of data and unified cache:

- Data accesses to Normal memory from EL2.
- When HCR_EL2.{E2H, TGE} != {1, 1}, Normal memory accesses to the EL2 translation tables.
- When EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}:
  — Data accesses to Normal memory from EL0.
  — Normal memory accesses to the EL2&0 translation tables.

0b1 This control has no effect on the Cacheability of:

- Data access to Normal memory from EL2.
- When HCR_EL2.{E2H, TGE} != {1, 1}, Normal memory accesses to the EL2 translation tables.
• When EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == \{1, 1\}:
  — Data accesses to Normal memory from EL0.
  — Normal memory accesses to the EL2&0 translation tables.

This bit has no effect on the EL3 translation regime.

When EL2 is disabled in the current Security state or HCR_EL2.{E2H, TGE} != \{1, 1\}, this bit has no effect on the EL1&0 translation regime.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL2 and, when EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == \{1, 1\}, EL0.

0b0 Alignment fault checking disabled when executing at EL2.

When EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == \{1, 1\}, alignment fault checking disabled when executing at EL0.

Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.

0b1 Alignment fault checking enabled when executing at EL2.

When EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == \{1, 1\}, alignment fault checking enabled when executing at EL0.

All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

If FEAT_MOPS is implemented, SETG* instructions have an alignment check regardless of the value of the A bit.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

M, bit [0]

MMU enable for EL2 or EL2&0 stage 1 address translation.

0b0 When HCR_EL2.{E2H, TGE} != \{1, 1\}, EL2 stage 1 address translation disabled.

When HCR_EL2.{E2H, TGE} == \{1, 1\}, EL2&0 stage 1 address translation disabled.

See the SCTLR_EL2.1 field for the behavior of instruction accesses to Normal memory.

0b1 When HCR_EL2.{E2H, TGE} != \{1, 1\}, EL2 stage 1 address translation enabled.

When HCR_EL2.{E2H, TGE} == \{1, 1\}, EL2&0 stage 1 address translation enabled.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

### Accessing SCTLR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic SCTLR_EL2 or SCTLR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:
### MRS <Xt>, SCTLR_EL2

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  X[t, 64] = SCTLR_EL2;
elsif PSTATE_EL == EL3 then
  X[t, 64] = SCTLR_EL2;

### MSR SCTLR_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  SCTLR_EL2 = X[t, 64];
elsif PSTATE_EL == EL3 then
  SCTLR_EL2 = X[t, 64];

### MRS <Xt>, SCTLR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.TRVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.SCTLR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x110];
  else
    X[t, 64] = SCTLR_EL1;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = SCTLR_EL2;
elsif PSTATE_EL == EL3 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = SCTLR_EL2;
else
  X[t, 64] = SCTLR_EL1;
X[t, 64] = SCTLR_EL1;
elsif PSTATE_EL == EL3 then
  X[t, 64] = SCTLR_EL1;

**MSR SCTLR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.TVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.SCTLR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    NVMem[0x110] = X[t, 64];
  else
    SCTLR_EL1 = X[t, 64];
  end
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    SCTLR_EL2 = X[t, 64];
  else
    SCTLR_EL1 = X[t, 64];
  end
elsif PSTATE_EL == EL3 then
  SCTLR_EL1 = X[t, 64];
D17.2.120  SCTLR_EL3, System Control Register (EL3)

The SCTLR_EL3 characteristics are:

Purpose

Provides top level control of the system, including its memory system, at EL3.

Configurations

This register is present only when EL3 is implemented. Otherwise, direct accesses to SCTLR_EL3 are UNDEFINED.

Attributes

SCTLR_EL3 is a 64-bit register.

Field descriptions

Bit [63]

Reserved, RES0.

SPINTMASK, bit [62]

When FEAT_NMI is implemented:

SP Interrupt Mask enable. When SCTLR_EL3.NMI is 1, controls whether PSTATE.SP acts as an interrupt mask, and controls the value of PSTATE.ALLINT on taking an exception to EL3.

0b0  Does not cause PSTATE.SP to mask interrupts.

PSTATE.ALLINT is set to 1 on taking an exception to EL3.

0b1  When PSTATE.SP is 1 and execution is at EL3, an IRQ or FIQ interrupt that is targeted to EL3 is masked regardless of any denotion of Superpriority.

PSTATE.ALLINT is set to 0 on taking an exception to EL3.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NMI, bit [61]

When FEAT_NMI is implemented:

Non-maskable Interrupt enable.

0b0  This control does not affect interrupt masking behavior.
This control enables all of the following:

- The use of the PSTATE.ALLINT interrupt mask.
- IRQ and FIQ interrupts to have Superpriority as an additional attribute.
- PSTATE.SP to be used as an interrupt mask.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**Bits [60:54]**

Reserved, RES0.

**TME, bit [53]**

*When FEAT_TME is implemented:*

Enables the Transactional Memory Extension at EL3.

- **0b0** Any attempt to execute a TSTART instruction at EL3 is trapped, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART instructions to be UNDEFINED at EL3.
- **0b1** This control does not cause any TSTART instruction to be trapped.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bit [52]**

Reserved, RES0.

**TMT, bit [51]**

*When FEAT_TME is implemented:*

Forces a trivial implementation of the Transactional Memory Extension at EL3.

- **0b0** This control does not cause any TSTART instruction to fail.
- **0b1** When the TSTART instruction is executed at EL3, the transaction fails with a TRIVIAL failure cause.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bits [50:45]**

Reserved, RES0.

**DSSBS, bit [44]**

*When FEAT_SSBS is implemented:*

Default PSTATE.SSBS value on Exception Entry.

- **0b0** PSTATE.SSBS is set to 0 on an exception to EL3.
- **0b1** PSTATE.SSBS is set to 1 on an exception to EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.
ATA, bit [43]

When FEAT_MTE2 is implemented:
Allocation Tag Access in EL3.
Controls access to Allocation Tags and Tag Check operations in EL3.
0b0 Access to Allocation Tags is prevented at EL3.
Memory accesses at EL3 are not subject to a Tag Check operation.
0b1 This control does not prevent access to Allocation Tags at EL3.
Tag Checked memory accesses at EL3 are subject to a Tag Check operation.
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bit [42]
Reserved, RES0.

TCF, bits [41:40]

When FEAT_MTE2 is implemented:
Tag Check Fault in EL3. Controls the effect of Tag Check Faults due to Loads and Stores in EL3.
If FEAT_MTE3 is not implemented, the value 0b11 is reserved.
0b0 Tag Check Faults have no effect on the PE.
0b1 Tag Check Faults cause a synchronous exception.
0b10 Tag Check Faults are asynchronously accumulated.
0b11 When FEAT_MTE3 is implemented:
Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bits [39:38]
Reserved, RES0.

ITFSB, bit [37]

When FEAT_MTE2 is implemented:
When synchronous exceptions are not being generated by Tag Check Faults, this field controls whether on exception entry into EL3, all Tag Check Faults due to instructions executed before exception entry, that are reported asynchronously, are synchronized into TFSRE0_EL1 and TFSR_ELx registers.
0b0 Tag Check Faults are not synchronized on entry to EL3.
0b1 Tag Check Faults are synchronized on entry to EL3.
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.
BT, bit [36]

When FEAT_BTI is implemented:

PAC Branch Type compatibility at EL3.

0b0  When the PE is executing at EL3, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == 0b11.

0b1  When the PE is executing at EL3, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == 0b11.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [35:32]

Reserved, RES0.

EnIA, bit [31]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APIAKey_EL1 key) of instruction addresses in the EL3 translation regime.

Possible values of this bit are:

0b0  Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is not enabled.

0b1  Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is enabled.

For more information, see Pointer authentication on page D8-5164.

——— Note ————

This field controls the behavior of the AddPACIA and AuthIA pseudocode functions. Specifically, when the field is 1, AddPACIA returns a copy of a pointer to which a pointer authentication code has been added, and AuthIA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

———

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnIB, bit [30]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APIBKey_EL1 key) of instruction addresses in the EL3 translation regime.

Possible values of this bit are:

0b0  Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is not enabled.

0b1  Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is enabled.
For more information, see Pointer authentication on page D8-5164.

Note
This field controls the behavior of the AddPACIB and AuthIB pseudocode functions. Specifically, when the field is 1, AddPACIB returns a copy of a pointer to which a pointer authentication code has been added, and AuthIB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bits [29:28]
Reserved, RES1.

EnDA, bit [27]
When FEAT_PAuth is implemented:
Controls enabling of pointer authentication (using the APDAKey_EL1 key) of instruction addresses in the EL3 translation regime.

0b0 Pointer authentication (using the APDAKey_EL1 key) of data addresses is not enabled.
0b1 Pointer authentication (using the APDAKey_EL1 key) of data addresses is enabled.

For more information, see Pointer authentication on page D8-5164.

Note
This field controls the behavior of the AddPACDA and AuthDA pseudocode functions. Specifically, when the field is 1, AddPACDA returns a copy of a pointer to which a pointer authentication code has been added, and AuthDA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bit [26]
Reserved, RES0.

EE, bit [25]
Endianness of data accesses at EL3, and stage 1 translation table walks in the EL3 translation regime.

0b0 Explicit data accesses at EL3, and stage 1 translation table walks in the EL3 translation regime are little-endian.
0b1 Explicit data accesses at EL3, and stage 1 translation table walks in the EL3 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception levels higher than EL0, this bit is RES0.
If an implementation does not provide Little-endian support at Exception levels higher than EL0, this bit is RES1.
The EE bit is permitted to be cached in a TLB.
The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

**Bit [24]**

Reserved, RES0.

**Bit [23]**

Reserved, RES1.

**EIS, bit [22]**

*When FEAT_ExS is implemented:*

Exception Entry is Context Synchronizing.

- **0b0** The taking of an exception to EL3 is not a context synchronizing event.
- **0b1** The taking of an exception to EL3 is a context synchronizing event.

If SCTLR_EL3.EIS is set to **0b0**:

- Indirect writes to ESR_EL3, FAR_EL3, SPSR_EL3, ELR_EL3 are synchronized on exception entry to EL3, so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.
- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL3.EIS:

- Changes to the PSTATE information on entry to EL3.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores and data processing instructions.
- Debug state exit.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES1.

**IESB, bit [21]**

*When FEAT_IESB is implemented:*

Implicit Error Synchronization event enable.

- **0b0** Disabled.
- **0b1** An implicit error synchronization event is added:
  - At each exception taken to EL3.
  - Before the operational pseudocode of each ERET instruction executed at EL3.

When the PE is in Debug state, the effect of this field is CONSTRAINED UNPREDICTABLE, and its Effective value might be 0 or 1 regardless of the value of the field and, if implemented, SCR_EL3.NMEA. If the Effective value of the field is 1, then an implicit error synchronization event is added after each DCPSx instruction taken to EL3 and before each DRPS instruction executed at EL3, in addition to the other cases where it is added.

When FEAT_DoubleFault is implemented, the PE is in Non-debug state, and the Effective value of SCR_EL3.NMEA is 1, this field is ignored and its Effective value is 1.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.
Otherweise:
Reserved, RES0.

Bit [20]
Reserved, RES0.

WXN, bit [19]
Write permission implies XN (Execute-never). For the EL3 translation regime, this bit can force all
memory regions that are writable to be treated as XN.
0b0 This control has no effect on memory access permissions.
0b1 Any region that is writable in the EL3 translation regime is forced to XN for accesses
from software executing at EL3.
This bit applies only when SCTLR_EL3.M bit is set.
The WXN bit is permitted to be cached in a TLB.
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL3, this field resets to an
architecturally UNKNOWN value.

Bit [18]
Reserved, RES1.

Bit [17]
Reserved, RES0.

Bit [16]
Reserved, RES1.

Bits [15:14]
Reserved, RES0.

EnDB, bit [13]
When FEAT_PAuth is implemented:
Controls enabling of pointer authentication (using the APDBKey_EL1 key) of instruction addresses
in the EL3 translation regime.
0b0 Pointer authentication (using the APDBKey_EL1 key) of data addresses is not enabled.
0b1 Pointer authentication (using the APDBKey_EL1 key) of data addresses is enabled.
For more information, see Pointer authentication on page D8-5164.

Note
This field controls the behavior of the AddPACDB and AuthDB pseudocode functions. Specifically,
when the field is 1, AddPACDB returns a copy of a pointer to which a pointer authentication code
has been added, and AuthDB returns an authenticated copy of a pointer. When the field is 0, both of
these functions are NOP.

The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL3, this field resets to an
architecturally UNKNOWN value.

Otherweise:
Reserved, RES0.
I, bit [12]

Instruction access Cacheability control, for accesses at EL3:

0b0  All instruction access to Normal memory from EL3 are Non-cacheable for all levels of instruction and unified cache.

If the value of SCTLR_EL3.M is 0, instruction accesses from stage 1 of the EL3 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

0b1  This control has no effect on the Cacheability of instruction access to Normal memory from EL3.

If the value of SCTLR_EL3.M is 0, instruction accesses from stage 1 of the EL3 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the EL1&0, EL2, or EL2&0 translation regimes.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

EOS, bit [11]

**When FEAT_ExS is implemented:**

Exception Exit is Context Synchronizing.

0b0  An exception return from EL3 is not a context synchronizing event

0b1  An exception return from EL3 is a context synchronizing event

If SCTLR_EL3.EOS is set to 0b0:

• Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.

• Exception Catch debug events are synchronous debug events.

• DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL3.EOS:

• The indirect write of the PSTATE and PC values from SPSR_EL3 and ELR_EL3 on exception return is synchronized.

• If the PE enters Debug state before the first instruction after an Exception return from EL3 to Non-secure state, any pending Halting debug event completes execution.

• The GIC behavior that allocates interrupts to FIQ or IRQ changes simultaneously with leaving the EL3 Exception level.

• Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores and data processing instructions.

• Exit from Debug state.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES1.

Bits [10:7]

Reserved, RES0.
nAA, bit [6]

When `FEAT_LSE2` is implemented:

Non-aligned access. This bit controls generation of Alignment faults at EL3 under certain conditions. The following instructions generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes for access:

- LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH.
- STLLR, STLLRH, STL, STLHR, STLUR, and STLURH

0b0 Unaligned accesses by the specified instructions generate an Alignment fault.
0b1 Unaligned accesses by the specified instructions do not generate an Alignment fault.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [5:4]

Reserved, RES1.

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL3 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then a SP alignment fault exception is generated. For more information, see SP alignment checking on page D1-4668.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Cacheability control, for data accesses.

0b0 All data access to Normal memory from EL3, and all Normal memory accesses to the EL3 translation tables, are Non-cacheable for all levels of data and unified cache.
0b1 This control has no effect on the Cacheability of:
  - Data access to Normal memory from EL3.
  - Normal memory accesses to the EL3 translation tables.

This bit has no effect on the EL1&0, EL2, or EL2&0 translation regimes.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL3.

0b0 Alignment fault checking disabled when executing at EL3. Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.
0b1 Alignment fault checking enabled when executing at EL3. All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.
If `FEAT_MOPS` is implemented, `SETG*` instructions have an alignment check regardless of the value of the A bit.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally **UNKNOWN** value.

**M, bit [0]**

MMU enable for EL3 stage 1 address translation. Possible values of this bit are:

- **0b0** EL3 stage 1 address translation disabled.
  - See the `SCTLR_EL3.I` field for the behavior of instruction accesses to Normal memory.
- **0b1** EL3 stage 1 address translation enabled.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

### Accessing SCTLR_EL3

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, SCTLR_EL3**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  X[t, 64] = SCTLR_EL3;
```

**MSR SCTLR_EL3, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  SCTLR_EL3 = X[t, 64];
```
D17.2.121 SCXTNUM_EL0, EL0 Read/Write Software Context Number

The SCXTNUM_EL0 characteristics are:

**Purpose**

Provides a number that can be used to separate out different context numbers with the EL0 exception level, for the purpose of protecting against side-channels using branch prediction and similar resources.

**Configurations**

This register is present only when FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented. Otherwise, direct accesses to SCXTNUM_EL0 are UNDEFINED.

**Attributes**

SCXTNUM_EL0 is a 64-bit register.

**Field descriptions**

- **Bits [63:0]**
  - Software Context Number. A number to identify the context within the EL0 exception level.
  - The reset behavior of this field is:
    - On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing SCXTNUM_EL0**

Accesses to this register use the following encodings in the System register encoding space:

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b011 & 0b1101 & 0b0000 & 0b111 \\
\end{array}
\]

```c
if PSTATE_EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && SCR_EL3.EnSCXT == '0' then
    UNDEFINED;
  elsif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.TSCXT == '1' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && HCR_EL2.EnSCXT == '0' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
      HFGRTR_EL2.SCXTNUM_EL0 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.TSCXT == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
```
if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = SCXTNUM_EL0;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnSCXT == '0' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.SCXTNUM_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = SCXTNUM_EL0;
        elsif PSTATE.EL == EL2 then
            if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnSCXT == '0' then
                UNDEFINED;
            elsif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
                if Halted() && EDSCR.SDD == '1' then
                    UNDEFINED;
                else
                    AArch64.SystemAccessTrap(EL3, 0x18);
                else
                    X[t, 64] = SCXTNUM_EL0;
        elsif PSTATE.EL == EL3 then
            X[t, 64] = SCXTNUM_EL0;
        elsif PSTATE.EL == EL0 then
            if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnSCXT == '0' then
                UNDEFINED;
            elsif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.TSCXT == '1' then
                if EL2Enabled() && HCR_EL2.TGE == '1' then
                    AArch64.SystemAccessTrap(EL2, 0x18);
                else
                    AArch64.SystemAccessTrap(EL1, 0x18);
                elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && HCR_EL2.EnSCXT == '0' then
                    AArch64.SystemAccessTrap(EL2, 0x18);
                elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && SCTLR_EL1.TSCXT == '1' then
                    AArch64.SystemAccessTrap(EL2, 0x18);
                elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.SCXTNUM_EL0 == '1' then
                    AArch64.SystemAccessTrap(EL2, 0x18);
                elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.TSCXT == '1' then
                    AArch64.SystemAccessTrap(EL2, 0x18);
                elsif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
                    if Halted() && EDSCR.SDD == '1' then
                        UNDEFINED;
                    else
                        AArch64.SystemAccessTrap(EL3, 0x18);
                    else
                        SCXTNUM_EL0 = X[t, 64];
elsif PSTATE_EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnSCXT == '0' then
UNDEFINED;
elsif EL2Enabled() && HCR_EL2.EnSCXT == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.SCXTNUM_EL0 == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
SCXTNUM_EL0 = X[t, 64];
elsif PSTATE_EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnSCXT == '0' then
UNDEFINED;
elsif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
SCXTNUM_EL0 = X[t, 64];
elsif PSTATE_EL == EL3 then
SCXTNUM_EL0 = X[t, 64];
D17.2.122 SCXTNUM_EL1, EL1 Read/Write Software Context Number

The SCXTNUM_EL1 characteristics are:

**Purpose**
Provides a number that can be used to separate out different context numbers with the EL1 exception level, for the purpose of protecting against side-channels using branch prediction and similar resources.

**Configurations**
This register is present only when FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented. Otherwise, direct accesses to SCXTNUM_EL1 are UNDEFINED.

**Attributes**
SCXTNUM_EL1 is a 64-bit register.

**Field descriptions**

```
+-------------------+-------------------+
| Bits [63:0]        | Software Context Number |
|-------------------+-------------------|
| 63 32             | Software Context Number |
| 31 0              |                   |
```

Bits [63:0]
Software Context Number. A number to identify the context within the EL1 exception level.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing SCXTNUM_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, SCXTNUM_EL1**

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b111</td>
</tr>
</tbody>
</table>
```

```
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" 
        && SCR_EL3.EnSCXT == '0' then
        UNDEFINED;
elseif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elseif EL2Enabled() && HCR_EL2.EnSCXT == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elseif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.SCXTNUM_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elseif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
```
elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
  X[t, 64] = NVMem[0x188];
else
  X[t, 64] = SCXTNUM_EL1;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnSCXT == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
      elsif HCR_EL2.E2H == '1' then
        SCXTNUM_EL2 = X[t, 64];
      else
        SCXTNUM_EL1 = X[t, 64];
      elsif PSTATE.EL == EL3 then
        SCXTNUM_EL1 = X[t, 64];
      endif
    endif
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  endif
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif PSTATE.EL == EL0 then
    UNDEFINED;
  elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnSCXT == '0' then
      UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
      AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
      SCXTNUM_EL2 = X[t, 64];
    else
      SCXTNUM_EL1 = X[t, 64];
    endif
  endif
endif

MSR SCXTNUM_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b00000</td>
<td>0b111</td>
</tr>
</tbody>
</table>
MRS <Xt>, SCXTNUM_EL12

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b11</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsf PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    \( X[t, 64] = \text{NVMem}[0x188] \);
elsf EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsf PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnSCXT == '0' then
      UNDEFINED;
elsf HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = SCXTNUM_EL1;
  else
    UNDEFINED;
elsf PSTATE.EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    X[t, 64] = SCXTNUM_EL1;
  else
    UNDEFINED;

MSR SCXTNUM_EL12, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b11</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsf PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    \( \text{NVMem}[0x188] = X[t, 64] \);
elsf EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsf PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnSCXT == '0' then
      UNDEFINED;
elsf HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
  else
    UNDEFINED;
SCXTNUM_EL1 = X[t, 64];
else
  UNDEFINED;
elsif PSTATE_EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    SCXTNUM_EL1 = X[t, 64];
  else
    UNDEFINED;
D17.2.123   SCXTNUM_EL2, EL2 Read/Write Software Context Number

The SCXTNUM_EL2 characteristics are:

**Purpose**

Provides a number that can be used to separate out different context numbers with the EL2 exception level, for the purpose of protecting against side-channels using branch prediction and similar resources.

**Configurations**

This register is present only when FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented. Otherwise, direct accesses to SCXTNUM_EL2 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

SCXTNUM_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Context Number</td>
<td>A number to identify the context within the EL2 exception level. The reset behavior of this field is:</td>
</tr>
<tr>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
<td></td>
</tr>
</tbody>
</table>

**Accessing SCXTNUM_EL2**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic SCXTNUM_EL2 or SCXTNUM_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, SCXTNUM_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1111</td>
<td>0b0000</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnSCXT == '0' then
    UNDEFINED;
elif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = SCXTNUM_EL2;
else if PSTATE_EL == EL3 then
  X[t, 64] = SCXTNUM_EL2;

**MSR SCXTNUM_EL2, <xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>b11</td>
<td>b100</td>
<td>b1101</td>
<td>b00000</td>
<td>b011</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
else if PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
else if PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnSCXT == '0' then
    UNDEFINED;
  elseif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      SCXTNUM_EL2 = X[t, 64];
else if PSTATE_EL == EL3 then
  SCXTNUM_EL2 = X[t, 64];

**MRS <xt>, SCXTNUM_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>b11</td>
<td>b0000</td>
<td>b1101</td>
<td>b00000</td>
<td>b011</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
else if PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnSCXT == '0' then
    UNDEFINED;
  elseif EL2Enabled() && HCR_EL2.NV == '011' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.EnSCXT == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && (HaveEL(EL3) | SCR_EL3.STRVy == '1') && HFCRTR_EL2.SCXTNUM_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x188];
else
    X[t, 64] = SCXTNUM_EL1;
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnSCXT == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        X[t, 64] = SCXTNUM_EL2;
    else
        X[t, 64] = SCXTNUM_EL1;
elsif PSTATE_EL == EL3 then
    X[t, 64] = SCXTNUM_EL1;

MSR SCXTNUM_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnSCXT == '0' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWR埃尔2.SCXTNUM_EL1 == '1') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x188] = X[t, 64];
    else
        SCXTNUM_EL1 = X[t, 64];
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnSCXT == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        SCXTNUM_EL2 = X[t, 64];
    else
        SCXTNUM_EL2 = X[t, 64];
elsif PSTATE_EL == EL3 then
    SCXTNUM_EL1 = X[t, 64];
**D17.2.124  SCXTNUM_EL3, EL3 Read/Write Software Context Number**

The SCXTNUM_EL3 characteristics are:

**Purpose**

Provides a number that can be used to separate out different context numbers with the EL3 exception level, for the purpose of protecting against side-channels using branch prediction and similar resources.

**Configurations**

This register is present only when EL3 is implemented and (FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented). Otherwise, direct accesses to SCXTNUM_EL3 are UNDEFINED.

**Attributes**

SCXTNUM_EL3 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Software Context Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Accessing SCXTNUM_EL3**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, SCXTNUM_EL3**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b111</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  UNDEFINED;
elsif PSTATE_EL == EL2 then
  UNDEFINED;
elsif PSTATE_EL == EL3 then
  X[t, 64] = SCXTNUM_EL3;
```
**MSR SCXTNUM_EL3, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b10</td>
<td>0b101</td>
<td>0b0000</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  SCXTNUM_EL3 = X[t, 64];
D17.2.125 SMCR_EL1, SME Control Register (EL1)

The SMCR_EL1 characteristics are:

**Purpose**

This register controls aspects of Streaming SVE that are visible at Exception levels EL1 and EL0.

**Configurations**

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMCR_EL1 are UNDEFINED.

This register has no effect if the PE is not in Streaming SVE mode.

When HCR_EL2.{E2H, TGE} == {1, 1} and EL2 is enabled in the current Security state, this register has no effect on execution at EL0 and EL1.

**Attributes**

SMCR_EL1 is a 64-bit register.

**Field descriptions**

**Bits [63:32]**

Reserved, RES0.

**FA64, bit [31]**

When FEAT_SME_FA64 is implemented:

Controls whether execution of an A64 instruction is considered legal when the PE is in Streaming SVE mode.

0b0 This control does not cause any instruction to be treated as legal in Streaming SVE mode.

0b1 This control causes all implemented A64 instructions to be treated as legal in Streaming SVE mode at EL1 and EL0, if they are treated as legal at more privileged Exception levels in the current Security state.

Arm recommends that portable SME software should not rely on this optional feature, and that operating systems should provide a means to test for compliance with this recommendation.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bits [30:9]**

Reserved, RES0.

**Bits [8:4]**

Reserved, RAZ/WI.
LEN, bits [3:0]
Requests an Effective Streaming SVE vector length (SVL) at EL1 of (LEN+1)*128 bits. This field also defines the Effective Streaming SVE vector length at EL0 when EL2 is not implemented, or EL2 is not enabled in the current Security state, or HCR_EL2.{E2H,TGE} is not {1,1}.

The Streaming SVE vector length can be any power of two from 128 bits to 2048 bits inclusive. An implementation can support any subset of the architecturally permitted lengths.

When the PE is in Streaming SVE mode, the Effective SVE vector length (VL) is equal to SVL. When FEAT_SVE is implemented, and the PE is not in Streaming SVE mode, VL is equal to the Effective Non-streaming SVE vector length. See ZCR_EL1.

For all purposes other than returning the result of a direct read of SMCR_EL1, the PE selects the Effective Streaming SVE vector length by performing checks in the following order:

1. If the requested length is less than the minimum implemented Streaming SVE vector length, then the Effective length is the minimum implemented Streaming SVE vector length.
2. If EL2 is implemented and enabled in the current Security state, and the requested length is greater than the Effective length at EL2, then the Effective length at EL2 is used.
3. If EL3 is implemented and the requested length is greater than the Effective length at EL3, then the Effective length at EL3 is used.
4. Otherwise, the Effective length is the highest supported Streaming SVE vector length that is less than or equal to the requested length.

An indirect read of SMCR_EL1.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMCR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic SMCR_EL1 or SMCR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b000 & 0b0001 & 0b0010 & 0b110 \\
\end{array}
\]
else
    X[t, 64] = SMCR_EL1;
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
elsif HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x1D);
elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x1D);
elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x1D);
    endif
elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x1D);
elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x1D);
    endif
else
    X[t, 64] = SMCR_EL1;
else
    if PSTATE_EL == EL0 then
        UNDEFINED;
    elsif PSTATE_EL == EL1 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && CPTR_EL3.ESM == '0' then
            UNDEFINED;
        elsif CPACR_EL1.SMEN == 'x0' then
            AArch64.SystemAccessTrap(EL1, 0x1D);
        elsif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x1D);
            endif
        elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
            NVMem[0x1F0] = X[t, 64];
        else
            SMCR_EL1 = X[t, 64];
        endif
    elsif PSTATE_EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && CPTR_EL3.ESM == '0' then
            UNDEFINED;
        elsif HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x1D);
            endif
        else
            if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && CPTR_EL3.ESM == '0' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x1D);
            endif
        endif
    endif
else
    X[t, 64] = SMCR_EL1;
end if

---

**MSR SMCR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif CPACR_EL1.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x1D);
    elsif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
        endif
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x1F0] = X[t, 64];
    else
        SMCR_EL1 = X[t, 64];
    endif
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
        endif
    else
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && CPTR_EL3.ESM == '0' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
        endif
    endif
else
    X[t, 64] = SMCR_EL1;
endif
AArch64.SystemAccessTrap(EL3, 0x1D);
elsif HCR_EL2.E2H == '1' then
  SMCR_EL2 = X[t, 64];
else
  SMCR_EL1 = X[t, 64];
elsif PSTATE_EL == EL3 then
  if CPTR_EL3.ESM == '0' then
    AArch64.SystemAccessTrap(EL3, 0x1D);
  else
    SMCR_EL1 = X[t, 64];
MRS <Xt>, SMCR_EL12

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV2,NV1,NV == '101' then
    X[t, 64] = NVMem[0x1F0];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
      UNDEFINED;
    elsif CPTR_EL2.SMEN == 'x0' then
      AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
      if Halted() && EDSCR.SOD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x1D);
      endif
    else
      X[t, 64] = SMCR_EL1;
    endif
  else
    X[t, 64] = SMCR_EL1;
  end
elsif PSTATE_EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    if CPTR_EL3.ESM == '0' then
      AArch64.SystemAccessTrap(EL3, 0x1D);
    else
      X[t, 64] = SMCR_EL1;
    endif
  else
    UNDEFINED;
  end
else
  X[t, 64] = SMCR_EL1;
endif

MSR SMCR_EL12, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then

if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    NVMem[0x1F0] = X[t, 64];
elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
            UNDEFINED;
        elsif CPTR_EL2.SMEN == 'x0' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
        end
    else
        SMCR_EL1 = X[t, 64];
    end
else
    UNDEFINED;
elsif PSTATE_EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if CPTR_EL3.ESM == '0' then
            AArch64.SystemAccessTrap(EL3, 0x1D);
        else
            SMCR_EL1 = X[t, 64];
        end
    else
        SMCR_EL1 = X[t, 64];
    end
else
    UNDEFINED;
D17.2.126  SMCR_EL2, SME Control Register (EL2)

The SMCR_EL2 characteristics are:

**Purpose**

This register controls aspects of Streaming SVE that are visible at Exception levels EL2, EL1, and EL0.

**Configurations**

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMCR_EL2 are UNDEFINED.

This register has no effect if the PE is not in Streaming SVE mode, or if EL2 is not enabled in the current Security state.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

SMCR_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:32]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits [30:9]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>Bits [8:4]</td>
<td>Reserved, RAZ/WI.</td>
</tr>
<tr>
<td>Bits [4:0]</td>
<td>LEN</td>
</tr>
</tbody>
</table>

**FA64, bit [31]**

*When FEAT_SME_FA64 is implemented:*

Controls whether execution of an A64 instruction is considered legal when the PE is in Streaming SVE mode.

0b0  This control does not cause any instruction to be treated as legal in Streaming SVE mode.

0b1  This control causes all implemented A64 instructions to be treated as legal in Streaming SVE mode at EL2, if they are treated as legal at EL3.

Arm recommends that portable SME software should not rely on this optional feature, and that operating systems should provide a means to test for compliance with this recommendation.

The reset behavior of this field is:

* On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.
LEN, bits [3:0]

Requests an Effective Streaming SVE vector length (SVL) at EL2 of \((LEN+1)\times 128\) bits. This field also defines the Effective Streaming SVE vector length at EL0 when EL2 is implemented and enabled in the current Security state, and HCR_EL2.E2H,TGE is \{1,1\}.

The Streaming SVE vector length can be any power of two from 128 bits to 2048 bits inclusive. An implementation can support any subset of the architecturally permitted lengths.

When the PE is in Streaming SVE mode, the Effective vector length (VL) is equal to SVL. When FEAT_SVE is implemented, and the PE is not in Streaming SVE mode, VL is equal to the Effective Non-streaming SVE vector length. See ZCR_EL2.

For all purposes other than returning the result of a direct read of SMCR_EL2, the PE selects the Effective Streaming SVE vector length by performing checks in the following order:

1. If the requested length is less than the minimum implemented Streaming SVE vector length, then the Effective length is the minimum implemented Streaming SVE vector length.
2. If EL3 is implemented and the requested length is greater than the Effective length at EL3, then the Effective length at EL3 is used.
3. Otherwise, the Effective length is the highest supported Streaming SVE vector length that is less than or equal to the requested length.

An indirect read of SMCR_EL2.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMCR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic SMCR_EL2 or SMCR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

### MRS <Xr>, SMCR_EL2

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b110</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() & HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & CPTR_EL3.ESM == '0'" then
    UNDEFINED;
  elsif HCR_EL2.E2H == '0' & CPTR_EL2.TSM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x1D);
  elsif HCR_EL2.E2H == '1' & CPTR_EL2.SMEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x1D);
  else
    AArch64.SystemAccessTrap(EL3, 0x1D);
  else
    AArch64.SystemAccessTrap(EL3, 0x1D);
```
X[t, 64] = SMCR_EL2;
elsif PSTATE_EL == EL3 then
  if CPTR_EL3.ESM == '0' then
    AArch64.SystemAccessTrap(EL3, 0x1D);
  else
    X[t, 64] = SMCR_EL2;

MSR SMCR_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
    UNDEFINED;
  elsif HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x1D);
  elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x1D);
  elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x1D);
    end
  else
    SMCR_EL2 = X[t, 64];
  endif
elsif PSTATE_EL == EL3 then
  if CPTR_EL3.ESM == '0' then
    AArch64.SystemAccessTrap(EL3, 0x1D);
  else
    SMCR_EL2 = X[t, 64];
  endif

MRS <Xt>, SMCR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
    UNDEFINED;
  elsif CPACR_EL1 SMEN == 'x0' then
    AArch64.SystemAccessTrap(EL1, 0x1D);
  elsif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x1D);
  elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x1D);
  elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
    if Halted() && CPTR_EL3.ESM == '0' then
      AArch64.SystemAccessTrap(EL3, 0x1D);
    else
      SMCR_EL2 = X[t, 64];
    endif
  else
    SMCR_EL2 = X[t, 64];
  endif

if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x1D);
elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x1F0];
else
    X[t, 64] = SMCR_EL1;
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        elsif CPTR_EL3.TSM == '0' then
            AArch64.SystemAccessTrap(EL3, 0x1D);
        else
            X[t, 64] = SMCR_EL1;
        endif
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x1F0] = X[t, 64];
    else
        SMCR_EL1 = X[t, 64];
    endif
    elsif PSTATE_EL == EL1 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
            UNDEFINED;
        elsif CPACR_EL1.SMEN == 'x0' then
            AArch64.SystemAccessTrap(EL1, 0x1D);
        elsif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            elsif HCR_EL2.E2H == '1' then
                X[t, 64] = SMCR_EL2;
            else
                X[t, 64] = SMCR_EL1;
            endif
        elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
            SMCR_EL1 = X[t, 64];
        else
            SMCR_EL1 = X[t, 64];
        endif
    elsif PSTATE_EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
            UNDEFINED;
        elsif HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        else
            X[t, 64] = SMCR_EL1;
        endif
    endif
endif

MSR SMCR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b110</td>
</tr>
</tbody>
</table>
elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x1D);
elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x1D);
elsif HCR_EL2.E2H == '1' then
    SMCR_EL2 = X[t, 64];
else
    SMCR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.ESM == '0' then
        AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        SMCR_EL1 = X[t, 64];
D17.2.127 SMCR_EL3, SME Control Register (EL3)

The SMCR_EL3 characteristics are:

Purpose

This register controls aspects of Streaming SVE that are visible at all Exception levels.

Configurations

This register is present only when FEAT_SME is implemented and EL3 is implemented. Otherwise, direct accesses to SMCR_EL3 are UNDEFINED.

This register has no effect if the PE is not in Streaming SVE mode.

Attributes

SMCR_EL3 is a 64-bit register.

Field descriptions

Bits [63:32]

Reserved, RES0.

FA64, bit [31]

When FEAT_SME_FA64 is implemented:

Controls whether execution of an A64 instruction is considered legal when the PE is in Streaming SVE mode.

0b0 This control does not cause any instruction to be treated as legal in Streaming SVE mode.

0b1 This control causes all implemented A64 instructions to be treated as legal in Streaming SVE mode at EL3.

Arm recommends that portable SME software should not rely on this optional feature, and that operating systems should provide a means to test for compliance with this recommendation.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [30:9]

Reserved, RES0.

Bits [8:4]

Reserved, RAZ/WI.

LEN, bits [3:0]

Requests an Effective Streaming SVE vector length (SVL) at EL3 of (LEN+1)*128 bits.

The Streaming SVE vector length can be any power of two from 128 bits to 2048 bits inclusive. An implementation can support any subset of the architecturally permitted lengths.
When the PE is in Streaming SVE mode, the Effective SVE vector length (VL) is equal to SVL. When FEAT_SVE is implemented, and the PE is not in Streaming SVE mode, VL is equal to the Effective Non-streaming SVE vector length. See ZCR_EL3.

For all purposes other than returning the result of a direct read of SMCR_EL3, the PE selects the Effective Streaming SVE vector length by performing checks in the following order:

1. If the requested length is less than the minimum implemented Streaming SVE vector length, then the Effective length is the minimum implemented Streaming SVE vector length.
2. Otherwise, the Effective length is the highest supported Streaming SVE vector length that is less than or equal to the requested length.

An indirect read of SMCR_EL3.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing SMCR_EL3

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, SMCR_EL3**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b110</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.ESM == '0' then
    AArch64.SystemAccessTrap(EL3, 0x1D);
  else
    X[t, 64] = SMCR_EL3;
end if
```

**MSR SMCR_EL3, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b110</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.ESM == '0' then
    AArch64.SystemAccessTrap(EL3, 0x1D);
  else
    SMCR_EL3 = X[t, 64];
end if
```
D17.2.128  SMIDR_EL1, Streaming Mode Identification Register

The SMIDR_EL1 characteristics are:

**Purpose**

Provides additional identification mechanisms for scheduling purposes, for a PE that supports Streaming SVE mode.

**Configurations**

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMIDR_EL1 are UNDEFINED.

**Attributes**

SMIDR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>31-24</td>
<td>Implementer code. This field must hold an implementer code that has been assigned by Arm. Assigned codes include the following:</td>
</tr>
<tr>
<td>23-16</td>
<td>Revision</td>
</tr>
<tr>
<td>15-12</td>
<td>RES0</td>
</tr>
<tr>
<td>11-0</td>
<td>Affinity</td>
</tr>
</tbody>
</table>

**Implementer, bits [31:24]**

The Implementer code. This field must hold an implementer code that has been assigned by Arm. Assigned codes include the following:

- 0x00  Reserved for software use.
- 0x41  Arm Limited.
- 0x42  Broadcom Corporation.
- 0x43  Cavium Inc.
- 0x44  Digital Equipment Corporation.
- 0x46  Fujitsu Ltd.
- 0x49  Infineon Technologies AG.
- 0x4D  Motorola or Freescale Semiconductor Inc.
- 0x50  NVIDIA Corporation.
- 0x51  Applied Micro Circuits Corporation.
- 0x56  Marvell International Ltd.
- 0x69  Intel Corporation.
- 0x5E  Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

It is not required that this value is the same as the value of MIDR_EL1.Implementer. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.
Revision, bits [23:16]

Revision number for the Streaming Mode Compute Unit (SMCU).
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

SMPS, bit [15]

Indicates support for Streaming SVE mode execution priority.
0b0 Priority control not supported.
0b1 Priority control supported.

Bits [14:12]

Reserved, RES0.

Affinity, bits [11:0]

The SMCU affinity of the accessing PE.
• A value of zero indicates that the PE's implementation of Streaming SVE mode is not shared
  with other PEs.
• Otherwise, the value identifies which SMCU is associated with this PE. The Affinity value
  associated with each SMCU is unique within the system as a whole.

Accessing SMIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS} \ <X_t>, \ SMIDR_EL1
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
  elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID1 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      X[t, 64] = SMIDR_EL1;
  elsif PSTATE_EL == EL2 then
    X[t, 64] = SMIDR_EL1;
  elsif PSTATE_EL == EL3 then
    X[t, 64] = SMIDR_EL1;
D17.2.129 SMPRIMAP_EL2, Streaming Mode Priority Mapping Register

The SMPRIMAP_EL2 characteristics are:

**Purpose**

Maps the value in SMPRI_EL1 to a streaming execution priority value for instructions executed at EL1 and EL0 in the same Security states as EL2.

**Configurations**

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMPRIMAP_EL2 are UNDEFINED.

When SMIDR_EL1.SMPS is '0', this register is RES0.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

SMPRIMAP_EL2 is a 64-bit register.

**Field descriptions**

When all of the following are true, the value in SMPRI_EL1 is mapped to a streaming execution priority using this register:

- The current Exception level is EL1 or EL0.
- EL2 is implemented and enabled in the current Security state.
- HCRX_EL2.SMPME is '1'.

Otherwise, SMPRI_EL1 holds the streaming execution priority value.

**P15, bits [63:60]**

Priority Mapping Entry 15. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '15'.

This value is the highest streaming execution priority.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**P14, bits [59:56]**

Priority Mapping Entry 14. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '14'.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**P13, bits [55:52]**

Priority Mapping Entry 13. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '13'.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
P12, bits [51:48]
Priority Mapping Entry 12. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '12'.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P11, bits [47:44]
Priority Mapping Entry 11. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '11'.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P10, bits [43:40]
Priority Mapping Entry 10. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '10'.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P9, bits [39:36]
Priority Mapping Entry 9. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '9'.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P8, bits [35:32]
Priority Mapping Entry 8. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '8'.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P7, bits [31:28]
Priority Mapping Entry 7. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '7'.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P6, bits [27:24]
Priority Mapping Entry 6. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '6'.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P5, bits [23:20]
Priority Mapping Entry 5. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '5'.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P4, bits [19:16]
Priority Mapping Entry 4. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '4'.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**P3, bits [15:12]**
Priority Mapping Entry 3. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘3’.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**P2, bits [11:8]**
Priority Mapping Entry 2. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘2’.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**P1, bits [7:4]**
Priority Mapping Entry 1. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘1’.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**P0, bits [3:0]**
Priority Mapping Entry 0. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘0’.
This value is the lowest streaming execution priority.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing SMPRIMAP_EL2**
Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS} <Xt>, \text{SMPRIMAP_EL2}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b101</td>
</tr>
</tbody>
</table>

```
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = NMem[0x1F8];
  elsif EL2Enabled() & HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.ESM == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) & CPTR_EL3.ESM == '0' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
```
else
    \[X[t, 64] = \text{SMPRIMAP\_EL2};\]
elsif PSTATE\_EL == EL3 then
    if CPTR\_EL3.ESM == '0' then
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      \[X[t, 64] = \text{SMPRIMAP\_EL2};\]

**MSR SMPRIMAP\_EL2, \(<Xt>\)**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE\_EL == EL0 then
  UNDEFINED;
elsif PSTATE\_EL == EL1 then
  if EL2Enabled() && HCR\_EL2.\(<\text{NV2,NV}>\) == '11' then
    NVMem[0x1F8] = \[X[t, 64];\]
  elsif EL2Enabled() && HCR\_EL2.\(<\text{NV}>\) == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE\_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.\(<\text{SDD}) == '1' && boolean IMPLEMENTATION\_DEFINED "EL3 trap priority when SDD == '1'" && CPTR\_EL3.ESM == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && CPTR\_EL3.ESM == '0' then
    if Halted() && EDSCR.\(<\text{SDD}) == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    SMPRIMAP\_EL2 = X[t, 64];
elsif PSTATE\_EL == EL3 then
  if CPTR\_EL3.ESM == '0' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    SMPRIMAP\_EL2 = X[t, 64];
D17.2.130 SMPRI_EL1, Streaming Mode Priority Register

The SMPRI_EL1 characteristics are:

**Purpose**

Configures the streaming execution priority for instructions executed on a shared Streaming Mode Compute Unit (SMCU) when the PE is in Streaming SVE mode at any Exception Level.

**Configurations**

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMPRI_EL1 are UNDEFINED.

When SMIDR_EL1.SMPS is '0', this register is RES0.

**Attributes**

SMPRI_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>31</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>30</td>
<td>Priority</td>
</tr>
</tbody>
</table>

**Bits [63:4]**

Reserved, RES0.

**Priority, bits [3:0]**

Streaming execution priority value.

Either this value is used directly, or it is mapped into an effective priority value using SMPRIMAP_EL2.

This value is used directly when any of the following are true:

- The current Exception level is EL3 or EL2.
- The current Exception level is EL1 or EL0, if EL2 is implemented and enabled in the current Security state and HCRX_EL2.SMPME is '0'.
- The current Exception level is EL1 or EL0, if EL2 is either not implemented or not enabled in the current Security state.

The precise meaning and behavior of each streaming execution priority value is IMPLEMENTATION DEFINED.

In an implementation that shares execution resources between PEs, higher priority values are allocated more processing resource than other PEs configured with lower priority values in the same Priority domain.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing SMPRI_EL1**

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, SMPRI_EL1

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CRTPR_EL3.EFM == '0' then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTE5 == '1') && HFGWTR_EL2.nSMPRI_EL1 == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CRTPR_EL3.EFM == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CRTPR_EL3.EFM == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && CRTPR_EL3.EFM == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  end
elsif PSTATE.EL == EL3 then
  if CRTPR_EL3.EFM == '0' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    SMPRI_EL1 = X[t, 64];
  end

MSR SMPRI_EL1, <Xt>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CRTPR_EL3.EFM == '0' then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTE5 == '1') && HFGWTR_EL2.nSMPRI_EL1 == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CRTPR_EL3.EFM == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CRTPR_EL3.EFM == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && CRTPR_EL3.EFM == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      SMPRI_EL1 = X[t, 64];
    end
  end
elsif PSTATE.EL == EL3 then
  if CRTPR_EL3.EFM == '0' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    SMPRI_EL1 = X[t, 64];
  end
when SDD == '1' && CPTR_EL3.ESM == '0' then
  UNDEFINED;
elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.ESM == '0' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    SMPRI_EL1 = X[t, 64];
else
  SMPRI_EL1 = X[t, 64];

D17.2.131 TCR_EL1, Translation Control Register (EL1)

The TCR_EL1 characteristics are:

**Purpose**

The control register for stage 1 of the EL1&0 translation regime.

**Configurations**

AArch64 System register TCR_EL1 bits [31:0] are architecturally mapped to AArch32 System register TTBCR[31:0].

AArch64 System register TCR_EL1 bits [63:32] are architecturally mapped to AArch32 System register TTBCR2[31:0].

**Attributes**

TCR_EL1 is a 64-bit register.

**Field descriptions**

Any of the bits in TCR_EL1, other than the A1 bit and the EPDx bits when they have the value 1, are permitted to be cached in a TLB.

**Bits [63:60]**

Reserved, RES0.

**DS, bit [59]**

When FEAT_LPA2 is implemented:

This field affects 52-bit output addressing when using 4KB and 16KB translation granules in stage 1 of the EL1&0 translation regime.

- **0b0** Bits[49:48] of translation descriptors are RES0.
  
  
  - The minimum value of the TCR_EL1.{T0SZ, T1SZ} fields is 16. Any memory access using a smaller value generates a stage 1 level 0 translation table fault.
  
  - Output address[51:48] is 0b0000.

- **0b1** Bits[49:48] of translation descriptors hold output address[49:48].
  
  - Bits[9:8] of Translation table descriptors hold output address[51:50].
The shareability information of Block and Page descriptors for cacheable locations is determined by:

- TCR_EL1.SH0 if the VA is translated using tables pointed to by TTBR0_EL1.
- TCR_EL1.SH1 if the VA is translated using tables pointed to by TTBR1_EL1.

The minimum value of the TCR_EL1.{T0SZ, T1SZ} fields is 12. Any memory access using a smaller value generates a stage 1 level 0 translation table fault.

All calculations of the stage 1 base address are modified for tables of fewer than 8 entries so that the table is aligned to 64 bytes.

Bits[5:2] of TTBR0_EL1 or TTBR1_EL1 are used to hold bits[51:48] of the output address in all cases.

--- Note ---

As FEAT_LVA must be implemented if TCR_EL1.DS == 1, the minimum value of the TCR_EL1.{T0SZ, T1SZ} fields is 12, as determined by that extension.

---

For the TLBI Range instructions affecting VA, the format of the argument is changed so that bits[36:0] hold BaseADDR[52:16]. For the 4KB translation granule, bits[15:12] of BaseADDR are treated as 0b0000. For the 16KB translation granule, bits[15:14] of BaseADDR are treated as 0b00.

--- Note ---

This forces alignment of the ranges used by the TLBI range instructions.

---

This field is RES0 for a 64KB translation granule.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCMA1, bit [58]

When FEAT_MTE2 is implemented:

Controls the generation of Unchecked accesses at EL1, and at EL0 if HCR_EL2.{E2H,TGE}!={1,1}, when address[59:55] = 0b11111.

0b0 This control has no effect on the generation of Unchecked accesses at EL1 or EL0.
0b1 All accesses at EL1 and EL0 are Unchecked.

--- Note ---

Software may change this control bit on a context switch.

---

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TCMA0, bit [57]

When FEAT_MTE2 is implemented:

Controls the generation of Unchecked accesses at EL1, and at EL0 if HCR_EL2.{E2H,TGE}!={1,1}, when address[59:55] = 0b00000.

0b0 This control has no effect on the generation of Unchecked accesses at EL1 or EL0.
0b1 All accesses at EL1 and EL0 are Unchecked.
— Note —

Software may change this control bit on a context switch.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E0PD1, bit [56]

When FEAT_E0PD is implemented:

Faulting control for Unprivileged access to any address translated by TTBR1_EL1.

0b0 Unprivileged access to any address translated by TTBR1_EL1 will not generate a fault by this mechanism.

0b1 Unprivileged access to any address translated by TTBR1_EL1 will generate a level 0 Translation fault.

Level 0 Translation faults generated as a result of this field are not counted as TLB misses for performance monitoring. The fault should take the same time to generate, whether the address is present in the TLB or not, to mitigate attacks that use fault timing.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E0PD0, bit [55]

When FEAT_E0PD is implemented:

Faulting control for Unprivileged access to any address translated by TTBR0_EL1.

0b0 Unprivileged access to any address translated by TTBR0_EL1 will not generate a fault by this mechanism.

0b1 Unprivileged access to any address translated by TTBR0_EL1 will generate a level 0 Translation fault.

Level 0 Translation faults generated as a result of this field are not counted as TLB misses for performance monitoring. The fault should take the same time to generate, whether the address is present in the TLB or not, to mitigate attacks that use fault timing.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NFD1, bit [54]

When FEAT_SVE is implemented or FEAT_TME is implemented:

Non-fault translation table walk disable for stage 1 translations using TTBR1_EL1.

This bit controls whether to perform a stage 1 translation table walk in response to a non-fault unprivileged access for a virtual address that is translated using TTBR1_EL1.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.
- Accesses due to an SVE first-fault gather load instruction that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.
- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.
For more information, see FEAT_SVE.

If FEAT_TME is implemented, the affected access types include all accesses generated by a load or store instruction in Transactional state.

0b0  Does not disable stage 1 translation table walks using TTBR1_EL1.
0b1  A TLB miss on a virtual address that is translated using TTBR1_EL1 due to the specified access types causes the access to fail without taking an exception. No stage 1 translation table walk is performed.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**NFD0, bit [53]**

*When FEAT_SVE is implemented or FEAT_TME is implemented:*

Non-fault translation table walk disable for stage 1 translations using TTBR0_EL1.

This bit controls whether to perform a stage 1 translation table walk in response to a non-fault unprivileged access for a virtual address that is translated using TTBR0_EL1.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.
- Accesses due to an SVE first-fault gather load instruction that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.
- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

For more information, see FEAT_SVE.

If FEAT_TME is implemented, the affected access types include all accesses generated by a load or store instruction in Transactional state.

0b0  Does not disable stage 1 translation table walks using TTBR0_EL1.
0b1  A TLB miss on a virtual address that is translated using TTBR0_EL1 due to the specified access types causes the access to fail without taking an exception. No stage 1 translation table walk is performed.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**TBID1, bit [52]**

*When FEAT_PAuth is implemented:*

Controls the use of the top byte of instruction addresses for address matching.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see Address tagging on page D8-5162.

0b0  TCR_EL1.TBI1 applies to Instruction and Data accesses.
0b1  TCR_EL1.TBI1 applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR1_EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.
TBID0, bit [51]

When FEAT_PAuth is implemented:

- Controls the use of the top byte of instruction addresses for address matching.
- For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.
- For more information, see Address tagging on page D8-5162.

- \( 0b0 \): TCR_EL1.TBI0 applies to Instruction and Data accesses.
- \( 0b1 \): TCR_EL1.TBI0 applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR0_EL1.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU162, bit [50]

When FEAT_HPDS2 is implemented:

- Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

- \( 0b0 \): For translations using TTBR1_EL1, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
- \( 0b1 \): For translations using TTBR1_EL1, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU161, bit [49]

When FEAT_HPDS2 is implemented:

- Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

- \( 0b0 \): For translations using TTBR1_EL1, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
- \( 0b1 \): For translations using TTBR1_EL1, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
HWU160, bit [48]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

0b0 For translations using TTBR1_EL1, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1 For translations using TTBR1_EL1, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU159, bit [47]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

0b0 For translations using TTBR1_EL1, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1 For translations using TTBR1_EL1, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU062, bit [46]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

0b0 For translations using TTBR0_EL1, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1 For translations using TTBR0_EL1, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
HWU061, bit [45]

When FEAT_HPDS2 is implemented:

   Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

   0b0  For translations using TTBR0_EL1, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

   0b1  For translations using TTBR0_EL1, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

   The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.

   The reset behavior of this field is:
   
   • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

   Reserved, RES0.

HWU060, bit [44]

When FEAT_HPDS2 is implemented:

   Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

   0b0  For translations using TTBR0_EL1, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

   0b1  For translations using TTBR0_EL1, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

   The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.

   The reset behavior of this field is:
   
   • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

   Reserved, RES0.

HWU059, bit [43]

When FEAT_HPDS2 is implemented:

   Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

   0b0  For translations using TTBR0_EL1, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

   0b1  For translations using TTBR0_EL1, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

   The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.

   The reset behavior of this field is:
   
   • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

   Reserved, RES0.
HPD1, bit [42]

When FEAT_HPDS is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR1_EL1.

0b0  Hierarchical permissions are enabled.
0b1  Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD0, bit [41]

When FEAT_HPDS is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR0_EL1.

0b0  Hierarchical permissions are enabled.
0b1  Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HD, bit [40]

When FEAT_HAFDBS is implemented:

Hardware management of dirty state in stage 1 translations from EL0 and EL1.

0b0  Stage 1 hardware management of dirty state disabled.
0b1  Stage 1 hardware management of dirty state enabled, only if the HA bit is also set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HA, bit [39]

When FEAT_HAFDBS is implemented:

Hardware Access flag update in stage 1 translations from EL0 and EL1.

0b0  Stage 1 Access flag update disabled.
0b1  Stage 1 Access flag update enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
TBI1, bit [38]

Top Byte ignored. Indicates whether the top byte of an address is used for address match for the TTBR1_EL1 region, or ignored and used for tagged addresses.

0b0 Top Byte used in the address calculation.
0b1 Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL1 using AArch64 where the address would be translated by tables pointed to by TTBR1_EL1. It has an effect whether the EL1&0 translation regime is enabled or not.

If FEAT_PAuth is implemented and TCR_EL1.TBID1 is 1, then this field only applies to Data accesses.

Otherwise, if the value of TBI1 is 1 and bit [55] of the target address to be stored to the PC is 1, then bits[63:56] of that target address are also set to 1 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

TBI0, bit [37]

Top Byte ignored. Indicates whether the top byte of an address is used for address match for the TTBR0_EL1 region, or ignored and used for tagged addresses.

0b0 Top Byte used in the address calculation.
0b1 Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL1 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL1. It has an effect whether the EL1&0 translation regime is enabled or not.

If FEAT_PAuth is implemented and TCR_EL1.TBID0 is 1, then this field only applies to Data accesses.

Otherwise, if the value of TBI0 is 1 and bit [55] of the target address to be stored to the PC is 0, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

AS, bit [36]

ASID Size.

0b0 8 bit - the upper 8 bits of TTBR0_EL1 and TTBR1_EL1 are ignored by hardware for every purpose except reading back the register, and are treated as if they are all zeros for when used for allocation and matching entries in the TLB.

0b1 16 bit - the upper 16 bits of TTBR0_EL1 and TTBR1_EL1 are used for allocation and matching in the TLB.

If the implementation has only 8 bits of ASID, this field is **RES0**.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
Bit [35]
Reserved, RES0.

IPS, bits [34:32]
Intermediate Physical Address Size.

- **0b000**: 32 bits, 4GB.
- **0b001**: 36 bits, 64GB.
- **0b010**: 40 bits, 1TB.
- **0b011**: 42 bits, 4TB.
- **0b100**: 44 bits, 16TB.
- **0b101**: 48 bits, 256TB.
- **0b110**: 52 bits, 4PB.
- All other values are reserved.

The reserved values behave in the same way as the **0b101** or **0b110** encoding, but software must not rely on this property as the behavior of the reserved values might change in a future revision of the architecture.

If the translation granule is not 64KB and **FEAT_LPA2** is not implemented, the value **0b110** is treated as reserved.

It is **IMPLEMENTATION DEFINED** whether an implementation that does not implement **FEAT_LPA** supports setting the value of **0b110** for the 64KB translation granule size or whether setting this value behaves as the **0b101** encoding.

In an implementation that supports 52-bit PAs, if the value of this field is not **0b110** or a value treated as **0b110**, then bits[51:48] of every translation table base address for the stage of translation controlled by TCR_EL1 are **0b0000**.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TG1, bits [31:30]
Granule size for the TTBR1_EL1.

- **0b01**: 16KB.
- **0b10**: 4KB.
- **0b11**: 64KB.
- Other values are reserved.

If the value is programmed to either a reserved value or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an **IMPLEMENTATION DEFINED** choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is **IMPLEMENTATION DEFINED** whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SH1, bits [29:28]
Shareability attribute for memory associated with translation table walks using TTBR1_EL1.

- **0b00**: Non-shareable.
- **0b10**: Outer Shareable.
- **0b11**: Inner Shareable.
- Other values are reserved. The effect of programming this field to a Reserved value is that behavior is **CONSTRAINED UNPREDICTABLE**.
The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ORGN1, bits [27:26]**

Outer cacheability attribute for memory associated with translation table walks using TTBR1_EL1.

- **0b00**: Normal memory, Outer Non-cacheable.
- **0b01**: Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
- **0b10**: Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
- **0b11**: Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IRGN1, bits [25:24]**

Inner cacheability attribute for memory associated with translation table walks using TTBR1_EL1.

- **0b00**: Normal memory, Inner Non-cacheable.
- **0b01**: Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
- **0b10**: Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
- **0b11**: Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**EPD1, bit [23]**

Translation table walk disable for translations using TTBR1_EL1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1_EL1. The encoding of this bit is:

- **0b0**: Perform translation table walks using TTBR1_EL1.
- **0b1**: A TLB miss on an address that is translated using TTBR1_EL1 generates a Translation fault. No translation table walk is performed.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**A1, bit [22]**

Selects whether TTBR0_EL1 or TTBR1_EL1 defines the ASID. The encoding of this bit is:

- **0b0**: TTBR0_EL1.ASID defines the ASID.
- **0b1**: TTBR1_EL1.ASID defines the ASID.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**T1SZ, bits [21:16]**

The size offset of the memory region addressed by TTBR1_EL1. The region size is $2^{(64-T1SZ)}$ bytes.

The maximum and minimum possible values for T1SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

--- Note ---

For the 4KB translation granule, if FEAT_LPA2 is implemented and this field is less than 16, the translation table walk begins with a level -1 initial lookup.

For the 16KB translation granule, if FEAT_LPA2 is implemented and this field is less than 17, the translation table walk begins with a level 0 initial lookup.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**TG0, bits [15:14]**

Granule size for the TTBR0_EL1.

- 0b00: 4KB
- 0b01: 64KB
- 0b10: 16KB

Other values are reserved.

If the value is programmed to either a reserved value or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**SH0, bits [13:12]**

Shareability attribute for memory associated with translation table walks using TTBR0_EL1.

- 0b00: Non-shareable
- 0b10: Outer Shareable
- 0b11: Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ORGN0, bits [11:10]**

Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL1.

- 0b00: Normal memory, Outer Non-cacheable.
- 0b01: Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
- 0b10: Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
- 0b11: Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IRGN0, bits [9:8]**

Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL1.

- 0b00: Normal memory, Inner Non-cacheable.
- 0b01: Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
- 0b10: Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
- 0b11: Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
EPD0, bit [7]
Translation table walk disable for translations using TTBR0_EL1. This bit controls whether a
translation table walk is performed on a TLB miss, for an address that is translated using
TTBR0_EL1. The encoding of this bit is:
0b0 Perform translation table walks using TTBR0_EL1.
0b1 A TLB miss on an address that is translated using TTBR0_EL1 generates a Translation
fault. No translation table walk is performed.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [6]
Reserved, RES0.

T0SZ, bits [5:0]
The size offset of the memory region addressed by TTBR0_EL1. The region size is \(2^{(64-T0SZ)}\) bytes.
The maximum and minimum possible values for T0SZ depend on the level of translation table and
the memory translation granule size, as described in the AArch64 Virtual Memory System
Architecture chapter.

Note
For the 4KB translation granule, if FEAT_LPA2 is implemented and this field is less than 16, the
translation table walk begins with a level -1 initial lookup.
For the 16KB translation granule, if FEAT_LPA2 is implemented and this field is less than 17, the
translation table walk begins with a level 0 initial lookup.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TCR_EL1
When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic TCR_EL1 or
TCR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b000 & 0b0010 & 0b0000 & 0b010 \\
\end{array}
\]

if PSTATE.EL == EL0 then
  UNDEFINED;
elsf PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TRVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsf EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.TCR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsf EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x120];
  else
    X[t, 64] = TCR_EL1;
elsf PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = TCR_EL2;
else
X[t, 64] = TCR_EL1;
elsif PSTATE.EL == EL3 then
X[t, 64] = TCR_EL1;

**MSR TCR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if EL2Enabled() && HCR_EL2.TVM == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.TCR_EL1 == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
NVMem[0x120] = X[t, 64];
else
TCR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
if HCR_EL2.E2H == '1' then
TCR_EL2 = X[t, 64];
else
TCR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
TCR_EL1 = X[t, 64];

**MRS <Xt>, TCR_EL12**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
X[t, 64] = NVMem[0x120];
elsif EL2Enabled() && HCR_EL2.NV == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
UNDEFINED;
elsif PSTATE.EL == EL2 then
if HCR_EL2.E2H == '1' then
X[t, 64] = TCR_EL1;
else
UNDEFINED;
elsif PSTATE.EL == EL3 then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
X[t, 64] = TCR_EL1;
else
UNDEFINED;
**MSR TCR_EL12, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NMem[0x120] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        TCR_EL1 = X[t, 64];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        TCR_EL1 = X[t, 64];
    else
        UNDEFINED;
D17.2.132  TCR_EL2, Translation Control Register (EL2)

The TCR_EL2 characteristics are:

### Purpose

The control register for stage 1 of the EL2, or EL2&0, translation regime:

- When the Effective value of HCR_EL2.E2H is 0, this register controls stage 1 of the EL2 translation regime, that supports a single VA range, translated using TTBR0_EL2.
- When the value of HCR_EL2.E2H is 1, this register controls stage 1 of the EL2&0 translation regime, that supports both:
  - A lower VA range, translated using TTBR0_EL2.
  - A higher VA range, translated using TTBR1_EL2.

### Configurations

AArch64 System register TCR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HTCR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

### Attributes

TCR_EL2 is a 64-bit register.

### Field descriptions

**When HCR_EL2.E2H == 0:**

```
| 63 | 62 | 61 | 60 | 59 | 58 | 57 | 56 | 55 | 54 | 53 | 52 | 51 | 50 | 49 | 48 | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9|  8|  7|  6|  5|  4|  3|  2|  1|  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| RES0 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9|  8|  7|  6|  5|  4|  3|  2|  1|  0 |
| RES1 | 63 | 62 | 61 | 60 | 59 | 58 | 57 | 56 | 55 | 54 | 53 | 52 | 51 | 50 | 49 | 48 | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9|  8|  7|  6|  5|  4|  3|  2|  1|  0 |
| TCMA | TBID | HWU62 | HWU61 | HD | HA | PS | TG0 | SH0 | RES0 | T0SZ | RES1 | TBI | HWU60 | HWU59 | RPD | HPD | RES1 | ORGN0 | IRGN0 | RES0 |
```

Any of the bits in TCR_EL2, other than the A1 bit and the EPDx bits when they have the value 1, are permitted to be cached in a TLB.

**Bits [63:33]**

- Reserved, RES0.

**DS, bit [32]**

*When FEAT_LPA2 is implemented:*

This field affects 52-bit output addressing when using 4KB and 16KB translation granules in stage 1 of the EL2 translation regime.

- **0b0**  Bits[49:48] of translation descriptors are RES0. Bits[9:8] in Block and Page descriptors encode shareability information in the SH[1:0] field. Bits[9:8] in table descriptors are ignored by hardware. The minimum value of TCR_EL2.T0SZ is 16. Any memory access using a smaller value generates a stage 1 level 0 translation table fault. Output address[51:48] is 0b0000.
0b1  Bits[49:48] of translation descriptors hold output address[49:48].
     Bits[9:8] of Translation table descriptors hold output address[51:50].
     The shareability information of Block and Page descriptors for cacheable locations is
determined by TCR_EL2.SH0.
     The minimum value of TCR_EL2.T0SZ is 12. Any memory access using a smaller
value generates a stage 1 level 0 translation table fault.
     All calculations of the stage 1 base address are modified for tables of fewer than 8
entries so that the table is aligned to 64 bytes.
     Bits[5:2] of TTBR0_EL2 are used to hold bits[51:48] of the output address in all cases.

Note
As FEAT_LVA must be implemented if TCR_EL2.DS == 1, the minimum value of the
TCR_EL2.T0SZ field is 12, as determined by that extension.

For the TLBI Range instructions affecting VA, the format of the argument is changed so
that bits[36:0] hold BaseADDR[52:16]. For the 4KB translation granule, bits[15:12] of
BaseADDR are treated as 0b0000. For the 16KB translation granule, bits[15:14] of
BaseADDR are treated as 0b00.

Note
This forces alignment of the ranges used by the TLBI range instructions.

This field is RES0 for a 64KB translation granule.
     The reset behavior of this field is:
     • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
     Reserved, RES0.

Bit [31]
     Reserved, RES1.

TCMA, bit [30]
When FEAT_MTE2 is implemented:
     Controls the generation of Unchecked accesses at EL2 when address [59:56] = 0b0000.
     0b0  This control has no effect on the generation of Unchecked accesses.
     0b1  All accesses are Unchecked.
     The reset behavior of this field is:
     • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
     Reserved, RES0.

TBID, bit [29]
When FEAT_PAuth is implemented:
     Controls the use of the top byte of instruction addresses for address matching.
     For the purpose of this field, all cache maintenance and address translation instructions that perform
address translation are treated as data accesses.
     For more information, see Address tagging on page D8-5162.
     0b0  TCR_EL2.TBI applies to Instruction and Data accesses.
     0b1  TCR_EL2.TBI applies to Data accesses only.
     This affects addresses where the address would be translated by tables pointed to by TTBR0_EL2.
     The reset behavior of this field is:
     • On a Warm reset, this field resets to an architecturally UNKNOWN value.
Otherwise:
Reserved, RES0.

HWU62, bit [28]

When FEAT_HPDS2 is implemented:
Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1
translation table Block or Page entry.
0b0 Bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware
for an IMPLEMENTATION DEFINED purpose.
0b1 Bit[62] of each stage 1 translation table Block or Page entry can be used by hardware
for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD is 1.
The Effective value of this field is 0 if the value of TCR_EL2.HPD is 0.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

HWU61, bit [27]

When FEAT_HPDS2 is implemented:
Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1
translation table Block or Page entry.
0b0 Bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware
for an IMPLEMENTATION DEFINED purpose.
0b1 Bit[61] of each stage 1 translation table Block or Page entry can be used by hardware
for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD is 1.
The Effective value of this field is 0 if the value of TCR_EL2.HPD is 0.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

HWU60, bit [26]

When FEAT_HPDS2 is implemented:
Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1
translation table Block or Page entry.
0b0 Bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware
for an IMPLEMENTATION DEFINED purpose.
0b1 Bit[60] of each stage 1 translation table Block or Page entry can be used by hardware
for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD is 1.
The Effective value of this field is 0 if the value of TCR_EL2.HPD is 0.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.
HWU59, bit [25]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry.

0b0  Bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1  Bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD is 0.

The reset behavior of this field is:

•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD, bit [24]

When FEAT_HPDS is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR0_EL2.

0b0  Hierarchical permissions are enabled.

0b1  Hierarchical permissions are disabled.

Note

In this case, bit[61] (APTable[0]) and bit[59] (PXNTable) of the next level descriptor attributes are required to be ignored by the PE and are no longer reserved, allowing them to be used by software.

When disabled, the permissions are treated as if the bits are zero.

The reset behavior of this field is:

•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES1.

HD, bit [22]

When FEAT_HAFDBS is implemented:

Hardware management of dirty state in stage 1 translations from EL2.

0b0  Stage 1 hardware management of dirty state disabled.

0b1  Stage 1 hardware management of dirty state enabled, only if the HA bit is also set to 1.

The reset behavior of this field is:

•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HA, bit [21]

When FEAT_HAFDBS is implemented:

Hardware Access flag update in stage 1 translations from EL2.

0b0  Stage 1 Access flag update disabled.

0b1  Stage 1 Access flag update enabled.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**TBI, bit [20]**

Top Byte Ignored. Indicates whether the top byte of an address is used for address match for the TTBR0_EL2 region, or ignored and used for tagged addresses.

For more information, see *Address tagging on page D8-5162*.

- 0b0: Top Byte used in the address calculation.
- 0b1: Top Byte ignored in the address calculation.

This affects addresses generated in EL2 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL2. It has an effect whether the EL2, or EL2&0, translation regime is enabled or not.

If FEAT_PAuth is implemented and TCR_EL2.TBID is 1, then this field only applies to Data accesses.

If the value of TBI is 1, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL2.
- An exception taken to EL2.
- An exception return to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [19]**

Reserved, RES0.

**PS, bits [18:16]**

Physical Address Size.

- 0b000: 32 bits, 4GB.
- 0b001: 36 bits, 64GB.
- 0b010: 40 bits, 1TB.
- 0b011: 42 bits, 4TB.
- 0b100: 44 bits, 16TB.
- 0b101: 48 bits, 256TB.
- 0b110: 52 bits, 4PB.

All other values are reserved.

The reserved values behave in the same way as the 0b101 or 0b110 encoding, but software must not rely on this property as the behavior of the reserved values might change in a future revision of the architecture.

If the translation granule is not 64KB and FEAT_LPA2 is not implemented, the value 0b110 is treated as reserved.

It is IMPLEMENTATION DEFINED whether an implementation that does not implement FEAT_LPA supports setting the value of 0b110 for the 64KB translation granule size or whether setting this value behaves as the 0b101 encoding.

In an implementation that supports 52-bit PAs, if the value of this field is not 0b110 or a value treated as 0b110, then bits[51:48] of every translation table base address for the stage of translation controlled by TCR_EL2 are 0b0000.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TG0, bits [15:14]
Granule size for the TTBR0_EL2.

0b00  4KB.
0b01  64KB.
0b10  16KB.

Other values are reserved.

If the value is programmed to either a reserved value or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SH0, bits [13:12]
Shareability attribute for memory associated with translation table walks using TTBR0_EL2.

0b00  Non-shareable.
0b10  Outer Shareable.
0b11  Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ORGN0, bits [11:10]
Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL2.

0b00  Normal memory, Outer Non-cacheable.
0b01  Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
0b10  Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11  Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IRGN0, bits [9:8]
Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL2.

0b00  Normal memory, Inner Non-cacheable.
0b01  Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
0b10  Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11  Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [7:6]
Reserved, RES0.
T0SZ, bits [5:0]

The size offset of the memory region addressed by TTBR0_EL2. The region size is 2^{64-T0SZ} bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

Note

For the 4KB translation granule, if FEAT_LPA2 is implemented and this field is less than 16, the translation table walk begins with a level -1 initial lookup.

For the 16KB translation granule, if FEAT_LPA2 is implemented and this field is less than 17, the translation table walk begins with a level 0 initial lookup.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_VHE is implemented and HCR_EL2.E2H == 1:

Any of the bits in TCR_EL2 are permitted to be cached in a TLB.

Bits [63:60]

Reserved, RES0.

DS, bit [59]

When FEAT_LPA2 is implemented:

This field affects 52-bit output addressing when using 4KB and 16KB translation granules in stage 1 of the EL2&0 translation regime.

0b0 Bits[49:48] of translation descriptors are RES0.


The minimum value of the TCR_EL2.{T0SZ, T1SZ} fields is 16. Any memory access using a smaller value generates a stage 1 level 0 translation table fault.

Output address[51:48] is 0b0000.

0b1 Bits[49:48] of translation descriptors hold output address[49:48].

Bits[9:8] of Translation table descriptors hold output address[51:50].
The shareability information of Block and Page descriptors for cacheable locations is determined by:

- TCR_EL2.SH0 if the VA is an address that is translated using tables pointed to by TTBR0_EL2.
- TCR_EL2.SH1 if the VA is an address that is translated using tables pointed to by TTBR1_EL2.

The minimum value of the TCR_EL2.{T0SZ, T1SZ} fields is 12. Any memory access using a smaller value generates a stage 1 level 0 translation table fault.

All calculations of the stage 1 base address are modified for tables of fewer than 16 entries so that the table is aligned to 64 bytes.

Bits[5:2] of TTBR0_EL2 or TTBR1_EL2 are used to hold bits[51:48] of the output address in all cases.

**Note**

As FEAT_LVA must be implemented if TCR_EL2.DS == 1, the minimum value of the TCR_EL2.{T0SZ, T1SZ} fields is 12, as determined by that extension.

For the TLBI Range instructions affecting VA, the format of the argument is changed so that bits[36:0] hold BaseADDR[52:16]. For the 4KB translation granule, bits[15:12] of BaseADDR are treated as 0b0000. For the 16KB translation granule, bits[15:14] of BaseADDR are treated as 0b00.

**Note**

This forces alignment of the ranges used by the TLBI range instructions.

This field is RES0 for a 64KB translation granule.

The reset behavior of this field is:

- **On a Warm reset, this field resets to an architecturally UNKNOWN value.**

**Otherwise:**

Reserved, RES0.

**TCMA1**, bit [58]

When **FEAT_MTE2** is implemented:

Controls the generation of Unchecked accesses at EL2, and at EL0 if HCR_EL2.TGE=1, when address[59:55] = \texttt{0b11111}.

\begin{itemize}
  \item \texttt{0b0} This control has no effect on the generation of Unchecked accesses at EL2 or EL0.
  \item \texttt{0b1} All accesses are Unchecked.
\end{itemize}

**Note**

Software may change this control bit on a context switch.

The reset behavior of this field is:

- **On a Warm reset, this field resets to an architecturally UNKNOWN value.**

**Otherwise:**

Reserved, RES0.

**TCMA0**, bit [57]

When **FEAT_MTE2** is implemented:

Controls the generation of Unchecked accesses at EL2, and at EL0 if HCR_EL2.TGE=1, when address[59:55] = \texttt{0b00000}.

\begin{itemize}
  \item \texttt{0b0} This control has no effect on the generation of Unchecked accesses at EL2 or EL0.
  \item \texttt{0b1} All accesses are Unchecked.
\end{itemize}
Note
Software may change this control bit on a context switch.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

E0PD1, bit [56]
When FEAT_E0PD is implemented:
Faulting control for Unprivileged access to any address translated by TTBR1_EL2.
0b0 Unprivileged access to any address translated by TTBR1_EL2 will not generate a fault by this mechanism.
0b1 Unprivileged access to any address translated by TTBR1_EL2 will generate a level 0 Translation fault.

Level 0 Translation faults generated as a result of this field are not counted as TLB misses for performance monitoring. The fault should take the same time to generate, whether the address is present in the TLB or not, to mitigate attacks that use fault timing.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

E0PD0, bit [55]
When FEAT_E0PD is implemented:
Faulting control for Unprivileged access to any address translated by TTBR0_EL2.
0b0 Unprivileged access to any address translated by TTBR0_EL2 will not generate a fault by this mechanism.
0b1 Unprivileged access to any address translated by TTBR0_EL2 will generate a level 0 Translation fault.

Level 0 Translation faults generated as a result of this field are not counted as TLB misses for performance monitoring. The fault should take the same time to generate, whether the address is present in the TLB or not, to mitigate attacks that use fault timing.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

NFD1, bit [54]
When FEAT_SVE is implemented or FEAT_TME is implemented:
Non-fault translation table walk disable for stage 1 translations using TTBR1_EL2.

This bit controls whether to perform a stage 1 translation table walk in response to a non-fault unprivileged access for a virtual address that is translated using TTBR1_EL2.

If SVE is implemented, the affected access types include:
• All accesses due to an SVE non-fault contiguous load instruction.
• Accesses due to an SVE first-fault gather load instruction that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.
• Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.
For more information, see FEAT_SVE.

If FEAT_TME is implemented, the affected access types include all accesses generated by a load or store instruction in Transactional state.

0b0  Does not disable stage 1 translation table walks using TTBR1_EL2.
0b1  A TLB miss on a virtual address that is translated using TTBR1_EL2 due to the specified access types causes the access to fail without taking an exception. No stage 1 translation table walk is performed.

The reset behavior of this field is:
•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

NFD0, bit [53]

When FEAT_SVE is implemented or FEAT_TME is implemented:

Non-fault translation table walk disable for stage 1 translations using TTBR0_EL2.

This bit controls whether to perform a stage 1 translation table walk in response to a non-fault unprivileged access for a virtual address that is translated using TTBR0_EL2.

If SVE is implemented, the affected access types include:
•  All accesses due to an SVE non-fault contiguous load instruction.
•  Accesses due to an SVE first-fault gather load instruction that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.
•  Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

For more information, see FEAT_SVE.

If FEAT_TME is implemented, the affected access types include all accesses generated by a load or store instruction in Transactional state.

0b0  Does not disable stage 1 translation table walks using TTBR0_EL2.
0b1  A TLB miss on a virtual address that is translated using TTBR0_EL2 due to the specified access types causes the access to fail without taking an exception. No stage 1 translation table walk is performed.

The reset behavior of this field is:
•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

TBID1, bit [52]

When FEAT_PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see Address tagging on page D8-5162.

0b0  TCR_EL2.TB1 applies to Instruction and Data accesses.
0b1  TCR_EL2.TB1 applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR1_EL2.

The reset behavior of this field is:
•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.
TBID0, bit [51]

When FEAT_PAuth is implemented:
Controls the use of the top byte of instruction addresses for address matching.
For more information, see Address tagging on page D8-5162.
0b0   TCR_EL2.TBI0 applies to Instruction and Data accesses.
0b1   TCR_EL2.TBI0 applies to Data accesses only.
This affects addresses where the address would be translated by tables pointed to by TTBR0_EL2.
The reset behavior of this field is:
•   On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

HWU162, bit [50]

When FEAT_HPDS2 is implemented:
Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1
translation table Block or Page entry for translations using TTBR1_EL2.
0b0   For translations using TTBR1_EL2, bit[62] of each stage 1 translation table Block or
       Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1   For translations using TTBR1_EL2, bit[62] of each stage 1 translation table Block or
       Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the
       value of TCR_EL2.HPD1 is 1.
The Effective value of this field is 0 if the value of TCR_EL2.HPD1 is 0.
The reset behavior of this field is:
•   On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

HWU161, bit [49]

When FEAT_HPDS2 is implemented:
Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1
translation table Block or Page entry for translations using TTBR1_EL2.
0b0   For translations using TTBR1_EL2, bit[61] of each stage 1 translation table Block or
       Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1   For translations using TTBR1_EL2, bit[61] of each stage 1 translation table Block or
       Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the
       value of TCR_EL2.HPD1 is 1.
The Effective value of this field is 0 if the value of TCR_EL2.HPD1 is 0.
The reset behavior of this field is:
•   On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

HWU160, bit [48]

When FEAT_HPDS2 is implemented:
Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1
translation table Block or Page entry for translations using TTBR1_EL2.
0b0   For translations using TTBR1_EL2, bit[60] of each stage 1 translation table Block or
       Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1 For translations using TTBR1_EL2, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD1 is 0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**HWU159, bit [47]**

*When FEAT_HPDS2 is implemented:*

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL2.

0b0 For translations using TTBR1_EL2, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1 For translations using TTBR1_EL2, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD1 is 0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**HWU062, bit [46]**

*When FEAT_HPDS2 is implemented:*

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

0b0 For translations using TTBR0_EL1, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1 For translations using TTBR0_EL1, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD0 is 0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**HWU061, bit [45]**

*When FEAT_HPDS2 is implemented:*

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

0b0 For translations using TTBR0_EL1, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1 For translations using TTBR0_EL1, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD0 is 0.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**HWU060, bit [44]**

*When FEAT_HPDS2 is implemented:*

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

- **0b0** For translations using TTBR0_EL1, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
- **0b1** For translations using TTBR0_EL1, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD0 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**HWU059, bit [43]**

*When FEAT_HPDS2 is implemented:*

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

- **0b0** For translations using TTBR0_EL1, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
- **0b1** For translations using TTBR0_EL1, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL2.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL2.HPD0 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**HPD1, bit [42]**

*When FEAT_HPDS is implemented:*

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR1_EL2.

- **0b0** Hierarchical permissions are enabled.
- **0b1** Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.
HPD0, bit [41]

*When FEAT_HPDS is implemented:*

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR0_EL2.

- 0b0: Hierarchical permissions are enabled.
- 0b1: Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

HD, bit [40]

*When FEAT_HAFDBS is implemented:*

Hardware management of dirty state in stage 1 translations from EL2.

- 0b0: Stage 1 hardware management of dirty state disabled.
- 0b1: Stage 1 hardware management of dirty state enabled, only if the HA bit is also set to 1.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

HA, bit [39]

*When FEAT_HAFDBS is implemented:*

Hardware Access flag update in stage 1 translations from EL2.

- 0b0: Stage 1 Access flag update disabled.
- 0b1: Stage 1 Access flag update enabled.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

TBI1, bit [38]

Top Byte Ignored. Indicates whether the top byte of an address is used for address match for the TTBR1_EL2 region, or ignored and used for tagged addresses.

For more information, see Address tagging on page D8-5162.

- 0b0: Top Byte used in the address calculation.
- 0b1: Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL2 using AArch64 where the address would be translated by tables pointed to by TTBR1_EL2. It has an effect whether the EL2, or EL2&0, translation regime is enabled or not.

If FEAT_PAuth is implemented and TCR_EL2.TBID1 is 1, then this field only applies to Data accesses.

If the value of TBI1 is 1 and bit [55] of the target address to be stored to the PC is 1, then bits[63:56] of that target address are also set to 1 before the address is stored in the PC, in the following cases:
- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**TBI0, bit [37]**

Top Byte Ignored. Indicates whether the top byte of an address is used for address match for the TTBR0_EL2 region, or ignored and used for tagged addresses.

For more information, see *Address tagging* on page D8-5162.

- 0b0  Top Byte used in the address calculation.
- 0b1  Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL2 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL2. It has an effect whether the EL2, or EL2&0, translation regime is enabled or not.

If FEAT_PAuth is implemented and TCR_EL2.TBID0 is 1, then this field only applies to Data accesses.

If the value of TBI0 is 1 and bit [55] of the target address to be stored to the PC is 0, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**AS, bit [36]**

ASID Size.

- 0b0  8 bit - the upper 8 bits of TTBR0_EL2 and TTBR1_EL2 are ignored by hardware for every purpose except reading back the register, and are treated as if they are all zeros for when used for allocation and matching entries in the TLB.
- 0b1  16 bit - the upper 16 bits of TTBR0_EL2 and TTBR1_EL2 are used for allocation and matching in the TLB.

If the implementation has only 8 bits of ASID, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [35]**

Reserved, RES0.

**IPS, bits [34:32]**

Intermediate Physical Address Size.

- 0b000  32 bits, 4GB.
- 0b001  36 bits, 64GB.
- 0b010  40 bits, 1TB.
- 0b011  42 bits, 4TB.
- 0b100  44 bits, 16TB.
- 0b101  48 bits, 256TB.
- 0b110  When FEAT_LPA is implemented:
  - 52 bits, 4PB.

All other values are reserved.
The reserved values behave in the same way as the 0b101 or 0b110 encoding, but software must not rely on this property as the behavior of the reserved values might change in a future revision of the architecture.

If the translation granule is not 64KB, the value 0b110 is treated as reserved.

It is IMPLEMENTATION DEFINED whether an implementation that does not implement FEAT_LPA supports setting the value of 0b110 for the 64KB translation granule size or whether setting this value behaves as the 0b101 encoding.

In an implementation that supports 52-bit PAs, if the value of this field is not 0b110 or a value treated as 0b110, then bits[51:48] of every translation table base address for the stage of translation controlled by TCR_EL2 are 0b0000.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**TG1, bits [31:30]**
Granule size for the TTBR1_EL2.
- 0b01 16KB.
- 0b10 4KB.
- 0b11 64KB.
Other values are reserved.
If the value is programmed to either a reserved value, or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.
It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**SH1, bits [29:28]**
Shareability attribute for memory associated with translation table walks using TTBR1_EL2.
- 0b00 Non-shareable.
- 0b10 Outer Shareable.
- 0b11 Inner Shareable.
Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ORGN1, bits [27:26]**
Outer cacheability attribute for memory associated with translation table walks using TTBR1_EL2.
- 0b00 Normal memory, Outer Non-cacheable.
- 0b01 Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
- 0b10 Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
- 0b11 Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IRGN1, bits [25:24]**
Inner cacheability attribute for memory associated with translation table walks using TTBR1_EL2.
- 0b00 Normal memory, Inner Non-cacheable.
0b01  Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
0b10  Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11  Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**EPD1, bit [23]**
Translation table walk disable for translations using TTBR1_EL2. This bit controls whether a
translation table walk is performed on a TLB miss, for an address that is translated using
TTBR1_EL2. The encoding of this bit is:
0b0  Perform translation table walks using TTBR1_EL2.
0b1  A TLB miss on an address that is translated using TTBR1_EL2 generates a Translation
fault. No translation table walk is performed.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**A1, bit [22]**
Selects whether TTBR0_EL2 or TTBR1_EL2 defines the ASID. The encoding of this bit is:
0b0  TTBR0_EL2.ASID defines the ASID.
0b1  TTBR1_EL2.ASID defines the ASID.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**T1SZ, bits [21:16]**
The size offset of the memory region addressed by TTBR1_EL2. The region size is $2^{(64-T1SZ)}$ bytes.
The maximum and minimum possible values for T1SZ depend on the level of translation table and
the memory translation granule size, as described in the AArch64 Virtual Memory System
Architecture chapter.

--- Note ---
For the 4KB translation granule, if FEAT_LPA2 is implemented and this field is less than 16, the
translation table walk begins with a level -1 initial lookup.
For the 16KB translation granule, if FEAT_LPA2 is implemented and this field is less than 17, the
translation table walk begins with a level 0 initial lookup.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**TG0, bits [15:14]**
Granule size for the TTBR0_EL2.
0b00  4KB.
0b01  64KB.
0b10  16KB.

Other values are reserved.
If the value is programmed to either a reserved value, or a size that has not been implemented, then
the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED
choice of the sizes that has been implemented for all purposes other than the value read back from
this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value
that corresponds to the size chosen.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**SH0, bits [13:12]**

Shareability attribute for memory associated with translation table walks using TTBR0_EL2.

- 0b00 Non-shareable.
- 0b10 Outer Shareable.
- 0b11 Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ORGN0, bits [11:10]**

Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL2.

- 0b00 Normal memory, Outer Non-cacheable.
- 0b01 Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
- 0b10 Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
- 0b11 Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IRGN0, bits [9:8]**

Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL2.

- 0b00 Normal memory, Inner Non-cacheable.
- 0b01 Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
- 0b10 Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
- 0b11 Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**EPD0, bit [7]**

Translation table walk disable for translations using TTBR0_EL2. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR0_EL2. The encoding of this bit is:

- 0b0 Perform translation table walks using TTBR0_EL2.
- 0b1 A TLB miss on an address that is translated using TTBR0_EL2 generates a Translation fault. No translation table walk is performed.

The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [6]**

Reserved, RES0.

**T0SZ, bits [5:0]**

The size offset of the memory region associated by TTBR0_EL2. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.
—— Note ———

For the 4KB translation granule, if FEAT_LPA2 is implemented and this field is less than 16, the translation table walk begins with a level -1 initial lookup.

For the 16KB translation granule, if FEAT_LPA2 is implemented and this field is less than 17, the translation table walk begins with a level 0 initial lookup.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TCR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic TCR_EL2 or TCR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TCR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elif PSTATE.EL == EL2 then
    X[t, 64] = TCR_EL2;
elif PSTATE.EL == EL3 then
    X[t, 64] = TCR_EL2;

**MSR TCR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elif PSTATE.EL == EL2 then
    TCR_EL2 = X[t, 64];
elif PSTATE.EL == EL3 then
    TCR_EL2 = X[t, 64];
MRS <Xt>, TCR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() &amp; HCR_EL2.TVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() &amp; (HaveEL(EL3) || SCR_EL3.FGTEn == '1') &amp; HFGWTR_EL2.TCR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() &amp; HCR_EL2.<NV2,NV1,NV> == '111' then
      NVMem[0x120] = X[t, 64];
    else
      X[t, 64] = TCR_EL1;
  endif;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    TCR_EL2 = X[t, 64];
  else
    TCR_EL1 = X[t, 64];
  endif;
elsif PSTATE.EL == EL3 then
  TCR_EL1 = X[t, 64];
endif;

MSR TCR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() &amp; HCR_EL2.TVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() &amp; (HaveEL(EL3) || SCR_EL3.FGTEn == '1') &amp; HFGWTR_EL2.TCR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() &amp; HCR_EL2.<NV2,NV1,NV> == '111' then
      NVMem[0x120] = X[t, 64];
    else
      TCR_EL1 = X[t, 64];
  endif;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    TCR_EL2 = X[t, 64];
  else
    TCR_EL1 = X[t, 64];
  endif;
elsif PSTATE.EL == EL3 then
  TCR_EL1 = X[t, 64];
endif;
D17.2.133 TCR_EL3, Translation Control Register (EL3)

The TCR_EL3 characteristics are:

Purpose

The control register for stage 1 of the EL3 translation regime.

Configurations

This register is present only when EL3 is implemented. Otherwise, direct accesses to TCR_EL3 are UNDEFINED.

Attributes

TCR_EL3 is a 64-bit register.

Field descriptions

Any of the bits in TCR_EL3 are permitted to be cached in a TLB.

Bits [63:33]

Reserved, RES0.

DS, bit [32]

Reserved, RES0.

When FEAT_LPA2 is implemented:

This field affects 52-bit output addressing when using 4KB and 16KB translation granules in stage 1 of the EL3 translation regime.

0b0

Bits[49:48] of translation descriptors are RES0.


The minimum value of TCR_EL3.T0SZ is 16. Any memory access using a smaller value generates a stage 1 level 0 translation table fault.

Output address[51:48] is 0b0000.

0b1

Bits[49:48] of translation descriptors hold output address[49:48].

Bits[9:8] of table translation descriptors hold output address[51:50].

The shareability information of Block and Page descriptors for cacheable locations is determined by TCR_EL3.SH0.

The minimum value of TCR_EL3.T0SZ is 12. Any memory access using a smaller value generates a stage 1 level 0 translation table fault.

All calculations of the stage 1 base address are modified for tables of fewer than 8 entries so that the table is aligned to 64 bytes.

Bits[5:2] of TTBR0_EL3 are used to hold bits[51:48] of the output address in all cases.
Note

As FEAT_LVA must be implemented if TCR_EL3.DS == 1, the minimum value of the TCR_EL3.T0SZ field is 12, as determined by that extension.

For the TLBI Range instructions affecting VA, the format of the argument is changed so that bits[36:0] hold BaseADDR[52:16]. For the 4KB translation granule, bits[15:12] of BaseADDR are treated as 0b0000. For the 16KB translation granule, bits[15:14] of BaseADDR are treated as 0b00.

Note

This forces alignment of the ranges used by the TLBI range instructions.

This field is RES0 for a 64KB translation granule.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [31]

Reserved, RES1.

TCMA, bit [30]

When FEAT_MTE2 is implemented:

Controls the generation of Unchecked accesses at EL3 when address [59:56] = 0b0000.

0b0  This control has no effect on the generation of Unchecked accesses.

0b1  All accesses are Unchecked.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBID, bit [29]

When FEAT_PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

0b0  TCR_EL3.TBI applies to Instruction and Data accesses.

0b1  TCR_EL3.TBI applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR0_EL3.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU62, bit [28]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry.

0b0  Bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
Bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL3.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL3.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**HWU61, bit [27]**

When **FEAT_HPDS2** is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry.

- 0b0: Bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
- 0b1: Bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL3.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL3.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**HWU60, bit [26]**

When **FEAT_HPDS2** is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry.

- 0b0: Bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
- 0b1: Bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL3.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL3.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**HWU59, bit [25]**

When **FEAT_HPDS2** is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry.

- 0b0: Bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
- 0b1: Bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL3.HPD is 1.

The Effective value of this field is 0 if the value of TCR_EL3.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.
HPD, bit [24]

When FEAT_HPDS is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR0_EL3.

- 0b0 Hierarchical permissions are enabled.
- 0b1 Hierarchical permissions are disabled.

--- Note ---

In this case, bit[61] (APTable[0]) and bit[59] (PXNTable) of the next level descriptor attributes are required to be ignored by the PE, and are no longer reserved, allowing them to be used by software.

When disabled, the permissions are treated as if the bits are zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES1.

HD, bit [22]

When FEAT_HAFDBS is implemented:

Hardware management of dirty state in stage 1 translations from EL3.

- 0b0 Stage 1 hardware management of dirty state disabled.
- 0b1 Stage 1 hardware management of dirty state enabled, only if the HA bit is also set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HA, bit [21]

When FEAT_HAFDBS is implemented:

Hardware Access flag update in stage 1 translations from EL3.

- 0b0 Stage 1 Access flag update disabled.
- 0b1 Stage 1 Access flag update enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBI, bit [20]

Top Byte Ignored. Indicates whether the top byte of an address is used for address match for the TTBR0_EL3 region, or ignored and used for tagged addresses.

- 0b0 Top Byte used in the address calculation.
- 0b1 Top Byte ignored in the address calculation.

This affects addresses generated in EL3 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL3. It has an effect whether the EL3 translation regime is enabled or not.
If FEAT_PAuth is implemented and TCR_EL3.TBID is 1, then this field only applies to Data accesses.

Otherwise, if the value of TBI is 1, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL3.
- A exception taken to EL3.
- An exception return to EL3.

For more information, see Address tagging on page D8-5162.

--- Note ---

This control determines the scope of address tagging. It never causes an exception to be generated.

---

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [19]

Reserved, RES0.

PS, bits [18:16]

Physical Address Size.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>32 bits, 4GB</td>
</tr>
<tr>
<td>0b01</td>
<td>36 bits, 64GB</td>
</tr>
<tr>
<td>0b10</td>
<td>40 bits, 1TB</td>
</tr>
<tr>
<td>0b11</td>
<td>42 bits, 4TB</td>
</tr>
<tr>
<td>0b10</td>
<td>44 bits, 16TB</td>
</tr>
<tr>
<td>0b10</td>
<td>48 bits, 256TB</td>
</tr>
<tr>
<td>0b10</td>
<td>52 bits, 4PB</td>
</tr>
</tbody>
</table>

All other values are reserved.

The reserved values behave in the same way as the 0b101 or 0b110 encoding, but software must not rely on this property as the behavior of the reserved values might change in a future revision of the architecture.

If the translation granule is not 64KB and FEAT_LPA2 is not implemented, the value 0b110 is treated as reserved.

It is IMPLEMENTATION DEFINED whether an implementation that does not implement FEAT_LPA supports setting the value of 0b110 for the 64KB translation granule size or whether setting this value behaves as the 0b101 encoding.

In an implementation that supports 52-bit PAs, if the value of this field is not 0b110 or a value treated as 0b110, then bits[51:48] of every translation table base address for the stage of translation controlled by TCR_EL3 are 0b0000.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TG0, bits [15:14]

Granule size for the TTBR0_EL3.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>4KB</td>
</tr>
<tr>
<td>0b1</td>
<td>64KB</td>
</tr>
<tr>
<td>0b10</td>
<td>16KB</td>
</tr>
</tbody>
</table>

Other values are reserved.
If the value is programmed to either a reserved value or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**SH0, bits [13:12]**

Shareability attribute for memory associated with translation table walks using TTBR0_EL3.

- 0b00  Non-shareable.
- 0b10  Outer Shareable.
- 0b11  Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ORGN0, bits [11:10]**

Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL3.

- 0b00  Normal memory, Outer Non-cacheable.
- 0b01  Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
- 0b10  Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
- 0b11  Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IRGN0, bits [9:8]**

Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL3.

- 0b00  Normal memory, Inner Non-cacheable.
- 0b01  Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
- 0b10  Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
- 0b11  Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [7:6]**

Reserved, RES0.

**T0SZ, bits [5:0]**

The size offset of the memory region addressed by TTBR0_EL3. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

--- Note ---

For the 4KB translation granule, if FEAT_LPA2 is implemented and this field is less than 16, the translation table walk begins with a level -1 initial lookup.
For the 16KB translation granule, if FEAT_LPA2 is implemented and this field is less than 17, the translation table walk begins with a level 0 initial lookup.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing TCR_EL3**

Accesses to this register use the following encodings in the System register encoding space:

\[ MRS <Xt>, TCR_EL3 \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

\[
\text{if} \ PSTATE.EL == \text{EL0 then} \\
\text{UNDEFINED;} \\
\text{elsif} \ PSTATE.EL == \text{EL1 then} \\
\text{UNDEFINED;} \\
\text{elsif} \ PSTATE.EL == \text{EL2 then} \\
\text{UNDEFINED;} \\
\text{elsif} \ PSTATE.EL == \text{EL3 then} \\
X[t, 64] = TCR_EL3;
\]

\[ MSR TCR_EL3, <Xt> \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

\[
\text{if} \ PSTATE.EL == \text{EL0 then} \\
\text{UNDEFINED;} \\
\text{elsif} \ PSTATE.EL == \text{EL1 then} \\
\text{UNDEFINED;} \\
\text{elsif} \ PSTATE.EL == \text{EL2 then} \\
\text{UNDEFINED;} \\
\text{elsif} \ PSTATE.EL == \text{EL3 then} \\
TCR_EL3 = X[t, 64];
\]
D17.2.134  TFSRE0_EL1, Tag Fault Status Register (EL0).

The TFSRE0_EL1 characteristics are:

**Purpose**

Holds accumulated Tag Check Faults occurring in EL0 that are not taken precisely.

**Configurations**

This register is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to TFSRE0_EL1 are UNDEFINED.

**Attributes**

TFSRE0_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>59-32</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>31</td>
<td>TF1, Tag Check Fault. Asynchronously set to 1</td>
</tr>
<tr>
<td></td>
<td>when a Tag Check Fault using a virtual address</td>
</tr>
<tr>
<td></td>
<td>with bit[55] == 0b1 occurs.</td>
</tr>
<tr>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On a Warm reset, this field resets to an</td>
</tr>
<tr>
<td></td>
<td>architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>30</td>
<td>TF0, Tag Check Fault. Asynchronously set to 1</td>
</tr>
<tr>
<td></td>
<td>when a Tag Check Fault using a virtual address</td>
</tr>
<tr>
<td></td>
<td>with bit[55] == 0b0 occurs.</td>
</tr>
<tr>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On a Warm reset, this field resets to an</td>
</tr>
<tr>
<td></td>
<td>architecturally UNKNOWN value.</td>
</tr>
</tbody>
</table>

**Accessing TFSRE0_EL1**

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TFSRE0_EL1
```

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0110</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.ATA == '0' then
    UNDEFINED;
elsif EL2Enabled() && HCR_EL2.ATA == '0' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.ATA == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TFSRE0_EL1;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.ATA == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.ATA == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TFSRE0_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.ATA == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.ATA == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TFSRE0_EL1 = X[t, 64];
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.ATA == '0' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.ATA == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.ATA == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TFSRE0_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.ATA == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.ATA == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TFSRE0_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
  TFSRE0_EL1 = X[t, 64];

### MSR TFSRE0_EL1, <Xt>###

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0110</td>
<td>0b001</td>
</tr>
</tbody>
</table>
D17.2.135   **TFSR_EL1, Tag Fault Status Register (EL1)**

The TFSR_EL1 characteristics are:

**Purpose**

Holds accumulated Tag Check Faults occurring in EL1 that are not taken precisely.

**Configurations**

This register is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to TFSR_EL1 are UNDEFINED.

**Attributes**

TFSR_EL1 is a 64-bit register.

**Field descriptions**

| Bits [63:2] | Reserved, RES0. |
| TF1, bit [1] | Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == 0b1 occurs.  
  The reset behavior of this field is:  
  • On a Warm reset, this field resets to an architecturally UNKNOWN value. |
| TF0, bit [0] | Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == 0b0 occurs.  
  The reset behavior of this field is:  
  • On a Warm reset, this field resets to an architecturally UNKNOWN value. |

**Accessing TFSR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS <Xt>, TFSR_EL1} \\
\begin{array}{cccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b000 & 0b010 & 0b0110 & 0b000 \\
\end{array}
\]

if PSTATE_EL == EL0 then  
UNDEFINED;  
elseif PSTATE_EL == EL1 then  
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' && SCR_EL3.ATA == '0' then  
  UNDEFINED;
elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> = '011' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & HCR_EL2.ATA = '0' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & SCR_EL3.ATA = '0' then
  if Halted() & EDSCR.SDD = '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  if EL2Enabled() & HCR_EL2.<NV2,NV1,NV> = '111' then
    X[t, 64] = NVMem[0x190];
  else
    X[t, 64] = TFSR_EL1;
elsif PSTATE_EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD = '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & SCR_EL3.ATA == '0' then
    UNDEFINED;
  elsif HCR_EL2.ATA == '0' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '011' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & HCR_EL2.ATA == '0' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & SCR_EL3.ATA == '0' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
elsif HCR_EL2.E2H == '1' then
  X[t, 64] = TFSR_EL2;
else
  X[t, 64] = TFSR_EL1;
elsif PSTATE_EL == EL3 then
  X[t, 64] = TFSR_EL1;

MSR TFSR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>
TFSR_EL2 = X[t, 64];
else
TFSR_EL1 = X[t, 64];
elseif PSTATE_EL == EL3 then
TFSR_EL1 = X[t, 64];

**MRS <X>, TFSR_EL12**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0101</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elseif PSTATE_EL == EL1 then
if EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '101' then
X[t, 64] = NVMem[0x190];
elseif EL2Enabled() & HCR_EL2.NV == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
UNDEFINED;
elseif PSTATE_EL == EL2 then
if HCR_EL2.E2H == '1' then
if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & SCR_EL3.ATA == '0' then
UNDEFINED;
elseif HaveEL(EL3) & SCR_EL3.ATA == '0' then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = TFSR_EL1;
else
UNDEFINED;
elseif PSTATE_EL == EL3 then
if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.E2H == '1' then
X[t, 64] = TFSR_EL1;
else
UNDEFINED;

**MSR TFSR_EL12, <X>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0101</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elseif PSTATE_EL == EL1 then
if EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '101' then
NVMem[0x190] = X[t, 64];
elseif EL2Enabled() & HCR_EL2.NV == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
UNDEFINED;
elseif PSTATE_EL == EL2 then
if HCR_EL2.E2H == '1' then
if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & SCR_EL3.ATA == '0' then
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.NV == '1' then
        if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & SCR_EL3.ATA == '0' then
            undefined;
        elsif EL2Enabled() & HCR_EL2.ATA == '0' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) & SCR_EL3.ATA == '0' then
            if Halted() & EDSCR.SDD == '1' then
                undefined;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end if
        else
            undefined;
        end if
    elsif EL2Enabled() & HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        undefined;
    end if
elsif PSTATE.EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & SCR_EL3.ATA == '0' then
        undefined;
    elsif HaveEL(EL3) & SCR_EL3.ATA == '0' then
        if Halted() & EDSCR.SDD == '1' then
            undefined;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end if
    else
        undefined;
    end if
else
    if PSTATE.EL == EL3 then
        undefined;
    else
        X[t, 64] = TFSR_EL2;
    end if
end if

---

MRS <Xt>, TFSR_EL2

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>
**MSR TFSR_EL2, <Xt>**

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() \&\& HCR_EL2.<NV,NV> == '11' then
    if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& SCR_EL3.ATA == '0' then
      UNDEFINED;
    elsif EL2Enabled() \&\& HCR_EL2.ATA == '0' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) \&\& SCR_EL3.ATA == '0' then
      if Halted() \&\& EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    else
      TFSR_EL2 = X[t, 64];
  end
elsif PSTATE.EL == EL2 then
  if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& SCR_EL3.ATA == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) \&\& SCR_EL3.ATA == '0' then
    if Halted() \&\& EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    TFSR_EL2 = X[t, 64];
  end
elsif PSTATE.EL == EL3 then
  TFSR_EL2 = X[t, 64];
else
  TFSR_EL2 = X[t, 64];

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>
D17.2.136  TFSR_EL2, Tag Fault Status Register (EL2)

The TFSR_EL2 characteristics are:

**Purpose**

Holds accumulated Tag Check Faults occurring in EL2 that are not taken precisely.

**Configurations**

This register is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to TFSR_EL2 are UNDEFINED.

**Attributes**

TFSR_EL2 is a 64-bit register.

**Field descriptions**

![Register Diagram]

**Bits [63:2]**

Reserved, RES0.

**TF1, bit [1]**

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == 0b1 occurs.

When HCR_EL2.E2H==0b0, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**TF0, bit [0]**

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == 0b0 occurs.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing TFSR_EL2**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic TFSR_EL2 or TFSR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.ATA == '0' then
      UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.ATA == '0' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3.ATA == '0' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
      if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
      if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.ATA == '0' then
        UNDEFINED;
      elseif HaveEL(EL3) && SCR_EL3.ATA == '0' then
        if Halted() && EDSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL3, 0x18);
        end
      elseif PSTATE.EL == EL3 then
        X[t, 64] = TFSR_EL2;
      end
MRS <Xt>, TFSR_EL2

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>

MSR TFSR_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>
TFSR_EL1 = X[t, 64];
elsif EL2Enabled() & HCR_EL2.NV == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elsif PSTATE_EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & SCR_EL3.ATA == '0' then
    UNDEFINED;
  elseif HaveEL(EL3) & SCR_EL3.ATA == '0' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  else
    TFSR_EL2 = X[t, 64];
  endif
elsif PSTATE_EL == EL3 then
  TFSR_EL2 = X[t, 64];
endif

MRS <xt>, TFSR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b010</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & SCR_EL3.ATA == '0' then
    UNDEFINED;
  elseif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '011' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() & HCR_EL2.ATA == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif HaveEL(EL3) & SCR_EL3.ATA == '0' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  else
    X[t, 64] = TFSR_EL1;
  endif
elsif PSTATE_EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & SCR_EL3.ATA == '0' then
    UNDEFINED;
  elseif HaveEL(EL3) & SCR_EL3.ATA == '0' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  else
    HCR_EL2.E2H == '1' then
      X[t, 64] = TFSR_EL2;
  else
    X[t, 64] = TFSR_EL1;
  endif
elsif PSTATE_EL == EL3 then
  X[t, 64] = TFSR_EL1;
MSR TFSR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.ATA == '0' then
        UNDEFINED;
    elseif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.ATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) && SCR_EL3.ATA == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        TFSR_EL1 = X[t, 64];
    end
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" && SCR_EL3.ATA == '0' then
        UNDEFINED;
    elseif HaveEL(EL3) && SCR_EL3.ATA == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    elseif HCR_EL2.E2H == '1' then
        TFSR_EL2 = X[t, 64];
    else
        TFSR_EL1 = X[t, 64];
    end
elsif PSTATE.EL == EL3 then
    TFSR_EL1 = X[t, 64];
else
    TFSR_EL1 = X[t, 64];
D17.2.137  **TFSR_EL3, Tag Fault Status Register (EL3)**

The TFSR_EL3 characteristics are:

**Purpose**

Holds accumulated Tag Check Faults occurring in EL3 that are not taken precisely.

**Configurations**

This register is present only when FEAT_MTE2 is implemented. Otherwise, direct accesses to TFSR_EL3 are UNDEFINED.

**Attributes**

TFSR_EL3 is a 64-bit register.

**Field descriptions**

```
+----+----+----+----+
| 63 | 32 | 31 |  0 |
+----+----+----+----+
|     | RES0|     |     |
|     |     |     |     |
|     |     |     | TF0 |
```

**Bits [63:1]**

Reserved, RES0.

**TF0, bit [0]**

Tag Check Fault. Asynchronously set to 1 when a Tag Check Fault using a virtual address with bit[55] == 0b0 occurs.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing TFSR_EL3**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TFSR_EL3**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b10</td>
<td>0b0101</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  UNDEFINED;
elsif PSTATE_EL == EL2 then
  UNDEFINED;
elsif PSTATE_EL == EL3 then
  X[t, 64] = TFSR_EL3;
```
**MSR TFSR_EL3, <Xt>**

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    TFSR_EL3 = X[t, 64];
```

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b10</td>
<td>0b010</td>
<td>0b011</td>
<td>0b000</td>
</tr>
</tbody>
</table>
D17.2.138  TPIDR2_EL0, EL0 Read/Write Software Thread ID Register 2

The TPIDR2_EL0 characteristics are:

**Purpose**

Provides a location where SME-aware software executing at EL0 can store thread identifying information, for context management purposes.

The PE makes no use of this register.

**Configurations**

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to TPIDR2_EL0 are UNDEFINED.

**Attributes**

TPIDR2_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Thread identifying information stored by software running at this Exception level.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On a Warm reset, this field resets to an architecturally <strong>UNKNOWN</strong> value.</td>
</tr>
</tbody>
</table>

**Accessing TPIDR2_EL0**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TPIDR2_EL0**

```plaintext
if PSTATE_EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
    UNDEFINED;
  elsif (!EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.EnTP2 == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    end;
  elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.EnTP2 == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.nTPIDR2_EL0 == '0' then
    if Halted() && EDSCR.SDD == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3.EnTP2 == '0' then
      if Halted() && EDSCR.SDD == '1' then
```
if PSTATE_EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
        UNDEFINED;
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && SCR_EL3.EnTP2 == '0' then
        UNDEFINED;
    else
        MSR TPIDR2_EL0, <Xt>
    end
else
    X[t, 64] = TPIDR2_EL0;
endif

if PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
        UNDEFINED;
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && SCR_EL3.EnTP2 == '0' then
        UNDEFINED;
    else
        MSR TPIDR2_EL0, <Xt>
    end
else
    X[t, 64] = TPIDR2_EL0;
endif

if PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
        UNDEFINED;
    elsif EL2Enabled() && SCR_EL3.EnTP2 == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            MSR TPIDR2_EL0, <Xt>
        endif
    elsif EL2Enabled() && HFGWTR_EL2.nTPIDR2_EL0 == '0' then
        UNDEFINED;
    else
        MSR TPIDR2_EL0, <Xt>
    end
else
    X[t, 64] = TPIDR2_EL0;
endif

if PSTATE_EL == EL3 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
        UNDEFINED;
    elsif EL2Enabled() && SCR_EL3.EnTP2 == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            MSR TPIDR2_EL0, <Xt>
        endif
    elsif EL2Enabled() && HFGWTR_EL2.nTPIDR2_EL0 == '0' then
        UNDEFINED;
    else
        MSR TPIDR2_EL0, <Xt>
    end
else
    X[t, 64] = TPIDR2_EL0;
endif

if PSTATE_EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL1.EnTP2 == '0' then
        MSR TPIDR2_EL0, <Xt>
    else
        MSR TPIDR2_EL0, <Xt>
    end
else
    X[t, 64] = TPIDR2_EL0;
endif

if PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.EnTP2 == '0' then
        MSR TPIDR2_EL0, <Xt>
    else
        MSR TPIDR2_EL0, <Xt>
    end
else
    X[t, 64] = TPIDR2_EL0;
endif

if PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.EnTP2 == '0' then
        MSR TPIDR2_EL0, <Xt>
    else
        MSR TPIDR2_EL0, <Xt>
    end
else
    X[t, 64] = TPIDR2_EL0;
endif

if PSTATE_EL == EL3 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.EnTP2 == '0' then
        MSR TPIDR2_EL0, <Xt>
    else
        MSR TPIDR2_EL0, <Xt>
    end
else
    X[t, 64] = TPIDR2_EL0;
endif
AArch64 System Register Descriptions
D17.2 General system control registers

AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & SCR_EL3.EnTP2 == '0' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  TPIDR2_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED “EL3 trap priority when SDD == '1'” & SCR_EL3.EnTP2 == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) & SCR_EL3.EnTP2 == '0' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TPIDR2_EL0 = X[t, 64];
elsif PSTATE.EL == EL3 then
  TPIDR2_EL0 = X[t, 64];
D17.2.139 TPIDR_EL0, EL0 Read/Write Software Thread ID Register

The TPIDR_EL0 characteristics are:

**Purpose**

Provides a location where software executing at EL0 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

**Configurations**

AArch64 System register TPIDR_EL0 bits [31:0] are architecturally mapped to AArch32 System register TPIDRURW[31:0].

**Attributes**

TPIDR_EL0 is a 64-bit register.

**Field descriptions**

```
+-------------+-------------+
| 63          | 32          |
| Thread ID   |             |
| [31:0]      |             |
```

**Bits [63:0]**

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing TPIDR_EL0**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <X>, TPIDR_EL0**

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>
```

```c
if PSTATE.EL == EL0 then
    if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        HFGRT_EL2.TPIDR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = TPIDR_EL0;
    elseif PSTATE.EL == EL1 then
        if EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRT_EL2.TPIDR_EL0 == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = TPIDR_EL0;
    elseif PSTATE.EL == EL2 then
        X[t, 64] = TPIDR_EL0;
    elseif PSTATE.EL == EL3 then
        X[t, 64] = TPIDR_EL0;
```

**MSR TPIDR_EL0, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &
  HFGWTR_EL2.TPIDR_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    TPIDR_EL0 = X[t, 64];
  endif
else PSTATE.EL == EL1 then
  if EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.TPIDR_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    TPIDR_EL0 = X[t, 64];
  endif
else PSTATE.EL == EL2 then
  TPIDR_EL0 = X[t, 64];
else PSTATE.EL == EL3 then
  TPIDR_EL0 = X[t, 64];
D17.2.140 TPIDR_EL1, EL1 Software Thread ID Register

The TPIDR_EL1 characteristics are:

**Purpose**

Provides a location where software executing at EL1 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

**Configurations**

AArch64 System register TPIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register TPIDRPRW[31:0].

**Attributes**

TPIDR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Thread ID</td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Thread ID</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:0]**

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing TPIDR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TPIDR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b100</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCRTR_EL2.TPIDR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = TPIDR_EL1;
    elsif PSTATE.EL == EL2 then
        X[t, 64] = TPIDR_EL1;
    elsif PSTATE.EL == EL3 then
        X[t, 64] = TPIDR_EL1;
```
**MSR TPIDR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.TPIDR_EL1 == '1') then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    TPIDR_EL1 = X[t, 64];
  endif
elsif PSTATE.EL == EL2 then
  TPIDR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
  TPIDR_EL1 = X[t, 64];
D17.2.141 TPIDR_EL2, EL2 Software Thread ID Register

The TPIDR_EL2 characteristics are:

**Purpose**

Provides a location where software executing at EL2 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

**Configurations**

AArch64 System register TPIDR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HTPIDR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

TPIDR_EL2 is a 64-bit register.

**Field descriptions**

| 63 | 62 | 61 | ... | 32
|----|----|----|-----|----
| Thread ID |
| 31 | 30 | 29 | ... | 0
| Thread ID |

**Bits [63:0]**

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing TPIDR_EL2**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TPIDR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = NVMem[0x090];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        X[t, 64] = TPIDR_EL2;
```
```plaintext
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x090] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    end
elsif PSTATE.EL == EL2 then
    TPIDR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    TPIDR_EL2 = X[t, 64];
```

### MSR TPIDR_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>
D17.2.142  TPIDR_EL3, EL3 Software Thread ID Register

The TPIDR_EL3 characteristics are:

**Purpose**

Provides a location where software executing at EL3 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

**Configurations**

This register is present only when EL3 is implemented. Otherwise, direct accesses to TPIDR_EL3 are UNDEFINED.

**Attributes**

TPIDR_EL3 is a 64-bit register.

**Field descriptions**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>62</td>
<td>61</td>
<td>60</td>
<td>59</td>
<td>58</td>
<td>57</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Thread ID</td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Thread ID</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:0]**

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing TPIDR_EL3**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRS <Xt>, TPIDR_EL3
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    X[t, 64] = TPIDR_EL3;
MSR TPIDR_EL3, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    UNDEFINED;
elsif PSTATE_EL == EL2 then
    UNDEFINED;
elsif PSTATE_EL == EL3 then
    TPIDR_EL3 = X[t, 64];
D17.2.143  TPIDRRO_EL0, EL0 Read-Only Software Thread ID Register

The TPIDRRO_EL0 characteristics are:

**Purpose**

Provides a location where software executing at EL1 or higher can store thread identifying information that is visible to software executing at EL0, for OS management purposes.

The PE makes no use of this register.

**Configurations**

AArch64 System register TPIDRRO_EL0 bits [31:0] are architecturally mapped to AArch32 System register TPIDRRO[31:0].

**Attributes**

TPIDRRO_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Thread ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td></td>
</tr>
</tbody>
</table>

**Accessing TPIDRRO_EL0**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TPIDRRO_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b011</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
    if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' &
        (HaveEL(EL3) || SCR_EL3.FGTEn == '1') &
        HGRTR_EL2.TPIDRRO_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = TPIDRRO_EL0;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &
        HGRTR_EL2.TPIDRRO_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = TPIDRRO_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = TPIDRRO_EL0;
elsif PSTATE.EL == EL3 then
    X[t, 64] = TPIDRRO_EL0;
```
### MSR TPIDRRO_EL0, <Xt>}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.TPIDRRO_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    TPIDRRO_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
  TPIDRRO_EL0 = X[t, 64];
elsif PSTATE.EL == EL3 then
  TPIDRRO_EL0 = X[t, 64];
D17.2.144  TTBR0_EL1, Translation Table Base Register 0 (EL1)

The TTBR0_EL1 characteristics are:

**Purpose**

Holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the lower VA range in the EL1&0 translation regime, and other information for this translation regime.

**Configurations**

AArch64 System register TTBR0_EL1 bits [63:0] are architecturally mapped to AArch32 System register TTBR0[63:0].

**Attributes**

TTBR0_EL1 is a 64-bit register.

**Field descriptions**

- **ASID**, bits [63:48]
  - An ASID for the translation table base address. The TCR_EL1.A1 field selects either TTBR0_EL1.ASID or TTBR1_EL1.ASID.
  - If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.
  - The reset behavior of this field is:
    - On a Warm reset, this field resets to an architecturally UNKNOWN value.

- **BADDR[47:1]**, bits [47:1]
  - Translation table base address:
    - Bits A[47:x] of the stage 1 translation table base address bits are in register bits[47:x].
    - Bits A[(x-1):0] of the stage 1 translation table base address are zero.
  - Address bit x is the minimum address bit required to align the translation table to the size of the table. The smallest permitted value of x is 6. The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL1.T0SZ, the translation stage, and the translation granule size.

  **Note**

  A translation table is required to be aligned to the size of the table. If a table contains fewer than eight entries, it must be aligned on a 64 byte address boundary.

  - If the value of TCR_EL1.IPS is not 0b110, then:
    - Register bits[(x-1):1] are RES0.
    - If the implementation supports 52-bit PAs and IPAs, then bits A[51:48] of the stage 1 translation table base address are 0b0000.
  - If FEAT_LPA is implemented and the value of TCR_EL1.IPS is 0b110, then:
    - Bits A[51:48] of the stage 1 translation table base address bits are in register bits[5:2].
    - Register bit[1] is RES0.
• When \( x > 6 \), register bits\([(x-1):6]\] are \texttt{RES0}.

---

**Note**

TCR_EL1.IPS==0b110 is permitted when:

- \texttt{FEAT_LPA} is implemented and the 64KB translation granule is used.
- \texttt{FEAT_LPA2} is implemented and the 4KB or 16KB translation granule is used.

When the value of \texttt{ID_AA64MMFR0_EL1.PARange} indicates that the implementation does not support a 52 bit PA size, if a translation table lookup uses this register when the Effective value of TCR_EL1.IPS is 0b110 and the value of register bits[5:2] is nonzero, an Address size fault is generated.

---

If any register bit[47:1] that is defined as \texttt{RES0} has the value 1 when a translation table walk is done using TTBR0_EL1, then the translation table base address might be misaligned, with effects that are 

\texttt{CONSTRAINED UNPREDICTABLE}, and must be one of the following:

- Bits \( A[(x-1):0] \) of the stage 1 translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally \texttt{UNKNOWN} value.

**CnP, bit [0]**

*When \texttt{FEAT_TTCNP} is implemented:*

Common not Private. This bit indicates whether each entry that is pointed to by TTBR0_EL1 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0_EL1.CnP is 1.

\begin{itemize}
  \item \texttt{0b0} The translation table entries pointed to by TTBR0_EL1, for the current translation regime and ASID, are permitted to differ from corresponding entries for TTBR0_EL1 for other PEs in the Inner Shareable domain. This is not affected by:
    \begin{itemize}
      \item The value of TTBR0_EL1.CnP on those other PEs.
      \item The value of the current ASID.
      \item If EL2 is implemented and enabled in the current Security state, the value of the current VMID.
    \end{itemize}
  \item \texttt{0b1} The translation table entries pointed to by TTBR0_EL1 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR0_EL1.CnP is 1 and all of the following apply:
    \begin{itemize}
      \item The translation table entries are pointed to by TTBR0_EL1.
      \item The translation tables relate to the same translation regime.
      \item The ASID is the same as the current ASID.
      \item If EL2 is implemented and enabled in the current Security state, the value of the current VMID.
    \end{itemize}
\end{itemize}

This bit is permitted to be cached in a TLB.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.
Note

If the value of the TTBR0_EL1.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR0_EL1s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONstrained UNpredictable, see CONstrained UNpredictable behaviors due to caching of control or data values on page K1-11585.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TTBR0_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic TTBR0_EL1 or TTBR0_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

### MRS <Xt>, TTBR0_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFCR_EL2.TTBR0_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x200];
  else
    X[t, 64] = TTBR0_EL1;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = TTBR0_EL2;
  else
    X[t, 64] = TTBR0_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = TTBR0_EL1;

### MSR TTBR0_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.TVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFCWTR_EL2.TTBR0_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = TTBR0_EL1;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = TTBR0_EL2;
  else
    X[t, 64] = TTBR0_EL1;
AArch64 System Register Descriptions

D17.2 General system control registers

```
AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    NVMem[0x200] = X[t, 64];
else
    TTBR0_EL1 = X[t, 64];
elif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' then
        TTBR0_EL2 = X[t, 64];
    else
        TTBR0_EL1 = X[t, 64];
elif PSTATE_EL == EL3 then
    TTBR0_EL1 = X[t, 64];

MRS <Xt>, TTBR0_EL12

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        X[t, 64] = NVMem[0x200];
elif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
e1se
    UNDEFINED;
elif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' then
        X[t, 64] = TTBR0_EL1;
e1se
    UNDEFINED;
elif PSTATE_EL == EL3 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        X[t, 64] = TTBR0_EL1;
e1se
    UNDEFINED;

MSR TTBR0_EL12, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x200] = X[t, 64];
elif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
e1se
    UNDEFINED;
elif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' then
        TTBR0_EL1 = X[t, 64];
e1se
    UNDEFINED;
elif PSTATE_EL == EL3 then
```
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    TTBR0_EL1 = X[t, 64];
else
    UNDEFINED;
D17.2.145   TTBR0_EL2, Translation Table Base Register 0 (EL2)

The TTBR0_EL2 characteristics are:

**Purpose**

When \( \text{HCR\_EL2.E2H} \) is 0, holds the base address of the translation table for the initial lookup for stage 1 of an address translation in the EL2 translation regime, and other information for this translation regime.

When \( \text{HCR\_EL2.E2H} \) is 1, holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the lower VA range in the EL2\&0 translation regime, and other information for this translation regime.

**Configurations**

AArch64 System register TTBR0_EL2 bits [47:1] are architecturally mapped to AArch32 System register HTTBR[47:1].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

TTBR0_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASID</td>
<td>Bits [63:48]</td>
</tr>
<tr>
<td>BADDR[47:1]</td>
<td>Translation table base address:</td>
</tr>
</tbody>
</table>

**ASID, bits [63:48]**

*When FEAT\_VHE is implemented:*

When \( \text{HCR\_EL2.E2H} \) is 0, this field is RES0.

When \( \text{HCR\_EL2.E2H} \) is 1, it holds an ASID for the translation table base address. The TCR\_EL2.A1 field selects either TTBR0_EL2.ASID or TTBR1_EL2.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

**BADDR[47:1], bits [47:1]**

Translation table base address:

- Bits A[47:x] of the stage 1 translation table base address bits are in register bits[47:x].
- Bits A[(x-1):0] of the stage 1 translation table base address are zero.

Address bit x is the minimum address bit required to align the translation table to the size of the table. The smallest permitted value of x is 6. The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR\_EL2.T0SZ, the translation stage, and the translation granule size.
A translation table is required to be aligned to the size of the table. If a table contains fewer than eight entries, it must be aligned on a 64 byte address boundary.

If the value of TCR_EL2.{I}PS is not 0b110, then:
- Register bits[(x-1):1] are RES0.
- If the implementation supports 52-bit PAs and IPAs, then bits A[51:48] of the stage 1 translation table base address are 0b0000.

If FEAT_LPA is implemented and the value of TCR_EL2.{I}PS is 0b110, then:
- Bits A[51:48] of the stage 1 translation table base address bits are in register bits[5:2].
- Register bit[1] is RES0.
- When x=6, register bits[(x-1):6] are RES0.

The OA size specified by TCR_EL2.{I}PS is determined as follows:
- The value of TCR_EL2.PS when the value of HCR_EL2.E2H is 0.
- The value of TCR_EL2.IPS when the value of HCR_EL2.E2H is 1.

TCR_EL2.{I}PS==0b110 is permitted when:
- FEAT_LPA is implemented and the 64KB translation granule is used.
- FEAT_LPA2 is implemented and the 4KB or 16KB translation granule is used.

When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation does not support a 52 bit PA size, if a translation table lookup uses this register when the Effective value of TCR_EL2.{I}PS is 0b110 and the value of register bits[5:2] is nonzero, an Address size fault is generated.

If any register bit[47:1] that is defined as RES0 has the value 1 when a translation table walk is done using TTBR0_EL2, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:
- Bits A[(x-1):0] of the stage 1 translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

When FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TTBR0_EL2 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0_EL2.CnP is 1.

0b0 The translation table entries pointed to by TTBR0_EL2 for the current translation regime, and ASID if applicable, are permitted to differ from corresponding entries for TTBR0_EL2 for other PEs in the Inner Shareable domain. This is not affected by:
- The value of TTBR0_EL2.CnP on those other PEs.
- When the current translation regime is the EL2&0 regime, the value of the current ASID.
The translation table entries pointed to by TTBR0_EL2 are the same as the translation
table entries for every other PE in the Inner Shareable domain for which the value of
TTBR0_EL2.CnP is 1 and all of the following apply:

• The translation table entries are pointed to by TTBR0_EL2.
• The translation tables relate to the same translation regime.
• If that translation regime is the EL2&0 regime, the ASID is the same as the
current ASID.

This bit is permitted to be cached in a TLB.

Note

If the value of the TTBR0_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain
and those TTBR0_EL2s do not point to the same translation table entries when the other conditions
specified for the case when the value of CnP is 1 apply, then the results of translations are
CONSTRAINED UNPREDICTABLE, see CONSTRAINED UNPREDICTABLE behaviors due to caching
of control or data values on page K1-11585.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TTBR0_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic TTBR0_EL2
or TTBR0_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS } <Xt>, \text{ TTBR0_EL2}
\]

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b1 & 0b100 & 0b0010 & 0b0000 & 0b0000
\end{array}
\]

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = TTBR0_EL2;
elsif PSTATE.EL == EL3 then
  X[t, 64] = TTBR0_EL2;

\[
\text{MSR TTBR0_EL2, } <Xt>
\]

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b1 & 0b100 & 0b0010 & 0b0000 & 0b0000
\end{array}
\]

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
    endif
endif PSTATE_EL == EL2 then
    TTBR0_EL2 = X[t, 64];
elsif PSTATE_EL == EL3 then
    TTBR0_EL2 = X[t, 64];

MRS <xt>, TTBR0_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    endif
    if EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.TTBR0_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x200];
    else
        X[t, 64] = TTBR0_EL1;
    endif
elsif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' then
        X[t, 64] = TTBR0_EL2;
    else
        TTBR0_EL1 = X[t, 64];
    endif
elsif PSTATE_EL == EL3 then
    TTBR0_EL1 = X[t, 64];

MSR TTBR0_EL1, <xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    endif
    if EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.TTBR0_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x200] = X[t, 64];
    else
        TTBR0_EL1 = X[t, 64];
    endif
elsif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' then
        TTBR0_EL2 = X[t, 64];
    else
        TTBR0_EL1 = X[t, 64];
    endif
elsif PSTATE_EL == EL3 then
    TTBR0_EL1 = X[t, 64];
D17.2.146 TTBR0_EL3, Translation Table Base Register 0 (EL3)

The TTBR0_EL3 characteristics are:

**Purpose**

Holds the base address of the translation table for the initial lookup for stage 1 of an address translation in the EL3 translation regime, and other information for this translation regime.

**Configurations**

This register is present only when EL3 is implemented. Otherwise, direct accesses to TTBR0_EL3 are UNDEFINED.

**Attributes**

TTBR0_EL3 is a 64-bit register.

**Field descriptions**

- **Bits [63:48]**
  - Reserved, RES0.

- **BADDR[47:1], bits [47:1]**
  - Translation table base address:
    - Bits A[47:x] of the stage 1 translation table base address bits are in register bits[47:x].
    - Bits A[(x-1):0] of the stage 1 translation table base address are zero.
  - Address bit x is the minimum address bit required to align the translation table to the size of the table. The smallest permitted value of x is 6. The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL3.T0SZ, the translation stage, and the translation granule size.

  ——— Note ———

  A translation table is required to be aligned to the size of the table. If a table contains fewer than eight entries, it must be aligned on a 64 byte address boundary.

  ——— Note ———

  If the value of TCR_EL3.PS is not 0b110, then:
  - Register bits[(x-1):1] are RES0.
  - If the implementation supports 52-bit PAs and IPAs, then bits A[51:48] of the stage 1 translation table base address are 0b0000.

  If FEAT_LPA is implemented and the value of TCR_EL3.PS is 0b110, then:
  - Bits A[51:48] of the stage 1 translation table base address bits are in register bits[5:2].
  - Register bit[1] is RES0.
  - When x>6, register bits[(x-1):6] are RES0.

  ——— Note ———

  TCR_EL3.PS==0b110 is permitted when:
  - FEAT_LPA is implemented and the 64KB translation granule is used.
• **FEAT_LPA2** is implemented and the 4KB or 16KB translation granule is used.

When the value of **ID_AA64MMFR0_EL1.PARange** indicates that the implementation does not support a 52 bit PA size, if a translation table lookup uses this register when the Effective value of **TCR_EL3.PS** is 0b110 and the value of register bits[5:2] is nonzero, an Address size fault is generated.

If any register bit[47:1] that is defined as RES0 has the value 1 when a translation table walk is done using TTBR0_EL3, then the translation table base address might be misaligned, with effects that are **CONSTRAINED UNPREDICTABLE**, and must be one of the following:

• Bits A[(x-1):0] of the stage 1 translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.

• The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**CnP, bit [0]**

*When FEAT_TTCNP is implemented:*

Common not Private. This bit indicates whether each entry that is pointed to by TTBR0_EL3 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0_EL3.CnP is 1.

0b0 The translation table entries pointed to by TTBR0_EL3, for the current translation regime, are permitted to differ from corresponding entries for TTBR0_EL3 for other PEs in the Inner Shareable domain. This is not affected by the value of TTBR0_EL3.CnP on those other PEs.

0b1 The translation table entries pointed to by TTBR0_EL3 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR0_EL3.CnP is 1 and the translation table entries are pointed to by TTBR0_EL3.

This bit is permitted to be cached in a TLB.

**Note**

If the value of the TTBR0_EL3.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR0_EL3s do not point to the same translation table entries the results of translations using TTBR0_EL3 are **CONSTRAINED UNPREDICTABLE**, see **CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values** on page K1-11585.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

*Otherwise:*

Reserved, RES0.

**Accessing TTBR0_EL3**

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, TTBR0_EL3

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b110 & 0b0010 & 0b0000 & 0b000 \\
\end{array}
\]

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  X[t, 64] = TTBR0_EL3;

MSR TTBR0_EL3, <Xt>

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b110 & 0b0010 & 0b0000 & 0b000 \\
\end{array}
\]

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  TTBR0_EL3 = X[t, 64];
TTBR1_EL1, Translation Table Base Register 1 (EL1)

The TTBR1_EL1 characteristics are:

**Purpose**

Holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the higher VA range in the EL1&0 stage 1 translation regime, and other information for this translation regime.

**Configurations**

AArch64 System register TTBR1_EL1 bits [63:0] are architecturally mapped to AArch32 System register TTBR1[63:0].

**Attributes**

TTBR1_EL1 is a 64-bit register.

**Field descriptions**

**ASID, bits [63:48]**

An ASID for the translation table base address. The TCR_EL1.A1 field selects either TTBR0_EL1.ASID or TTBR1_EL1.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**BADDR[47:1], bits [47:1]**

Translation table base address:

- Bits A[47:x] of the stage 1 translation table base address bits are in register bits[47:x].
- Bits A[(x-1):0] of the stage 1 translation table base address are zero.

Address bit x is the minimum address bit required to align the translation table to the size of the table. The smallest permitted value of x is 6. The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL1.T1SZ, the translation stage, and the translation granule size.

--- Note ---

A translation table is required to be aligned to the size of the table. If a table contains fewer than eight entries, it must be aligned on a 64 byte address boundary.

If the value of TCR_EL1.IPS is not 0b110, then:

- Register bits[(x-1):1] are RES0.
- If the implementation supports 52-bit PAs and IPAs, then bits A[51:48] of the stage 1 translation table base address are 0b0000.

If FEAT_LPA is implemented and the value of TCR_EL1.IPS is 0b110, then:

- Bits A[51:48] of the stage 1 translation table base address bits are in register bits[5:2].
- Register bit[1] is RES0.
• When x>6, register bits[(x-1):6] are RES0.

**Note**

TCR_EL1.IPS==0b110 is permitted when:

- FEAT_LPA is implemented and the 64KB translation granule is used.
- FEAT_LPA2 is implemented and the 4KB or 16KB translation granule is used.

When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation does not support a 52 bit PA size, if a translation table lookup uses this register when the Effective value of TCR_EL1.IPS is 0b110 and the value of register bits[5:2] is nonzero, an Address size fault is generated.

If any register bit[47:1] that is defined as RES0 has the value 1 when a translation table walk is done using TTBR1_EL1, then the translation table base address might be misaligned, with effects that are constrained unpredictable, and must be one of the following:

- Bits A[(x-1):0] of the stage 1 translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally unknown value.

**CnP, bit [0]**

*When FEAT_TTCNP is implemented:*

Common not Private. This bit indicates whether each entry that is pointed to by TTBR1_EL1 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR1_EL1.CnP is 1.

0b0 The translation table entries pointed to by TTBR1_EL1, for the current translation regime and ASID, are permitted to differ from corresponding entries for TTBR1_EL1 for other PEs in the Inner Shareable domain. This is not affected by:

- The value of TTBR1_EL1.CnP on those other PEs.
- The value of the current ASID.
- If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

0b1 The translation table entries pointed to by TTBR1_EL1 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR1_EL1.CnP is 1 and all of the following apply:

- The translation table entries are pointed to by TTBR1_EL1.
- The translation tables relate to the same translation regime.
- The ASID is the same as the current ASID.
- If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

This bit is permitted to be cached in a TLB.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.
--- Note ---

If the value of the TTBR1_EL1.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR1_EL1s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONstrained UNPREDICTABLE, see CONstrained UNPREDICTABLE behaviors due to caching of control or data values on page K1-11585.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Accessing TTBR1_EL1**

When `HCR_EL2.E2H` is 1, without explicit synchronization, access from EL3 using the mnemonic TTBR1_EL1 or TTBR1_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TTBR1_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() & (!HaveEL(EL3) | SCR_EL3.FGTEn == '1') & HFCRTR_EL2.TTBR1_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x210];
else
        X[t, 64] = TTBR1_EL1;
elif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        X[t, 64] = TTBR1_EL2;
else
        X[t, 64] = TTBR1_EL1;
elif PSTATE.EL == EL3 then
        X[t, 64] = TTBR1_EL1;

**MSR TTBR1_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() & (!HaveEL(EL3) | SCR_EL3.FGTEn == '1') & HFGWTR_EL2.TTBR1_EL1 == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
  NVMem[0x210] = X[t, 64];
else
  TTBR1_EL1 = X[t, 64];
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    TTBR1_EL2 = X[t, 64];
  else
    TTBR1_EL1 = X[t, 64];
elsif PSTATE_EL == EL3 then
  TTBR1_EL1 = X[t, 64];

MRS <Xt>, TTBR1_EL12

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    X[t, 64] = NVMem[0x210];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);  
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    TTBR1_EL1 = X[t, 64];
  else
    UNDEFINED;
elsif PSTATE_EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    X[t, 64] = TTBR1_EL1;
  else
    UNDEFINED;

MSR TTBR1_EL12, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    NVMem[0x210] = X[t, 64];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);  
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    TTBR1_EL2 = X[t, 64];
  else
    UNDEFINED;
elsif PSTATE_EL == EL3 then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
  TTBR1_EL1 = X[t, 64];
else
  UNDEFINED;
D17.2.148  TTBR1_EL2, Translation Table Base Register 1 (EL2)

The TTBR1_EL2 characteristics are:

**Purpose**

When HCR_EL2.E2H is 1, holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the higher VA range in the EL2&0 translation regime, and other information for this translation regime.

--- Note ---

When HCR_EL2.E2H is 0, the contents of this register are ignored by the PE, except for a direct read or write of the register.

**Configurations**

This register is present only when FEAT_VHE is implemented. Otherwise, direct accesses to TTBR1_EL2 are **UNDEFINED**.

If EL2 is not implemented, this register is **RES0** from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

TTBR1_EL2 is a 64-bit register.

**Field descriptions**

ASID, bits [63:48]

An ASID for the translation table base address. The TCR_EL2.A1 field selects either TTBR0_EL2.ASID or TTBR1_EL2.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are **RES0**.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

BADDR[47:1], bits [47:1]

Translation table base address:

- Bits A[47:x] of the stage 1 translation table base address bits are in register bits[47:x].
- Bits A[(x-1):0] of the stage 1 translation table base address are zero.

Address bit x is the minimum address bit required to align the translation table to the size of the table. The smallest permitted value of x is 6. The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL2.T1SZ, the translation stage, and the translation granule size.

--- Note ---

A translation table is required to be aligned to the size of the table. If a table contains fewer than eight entries, it must be aligned on a 64 byte address boundary.
If the value of `TCR_EL2.{i}PS` is not `0b110`, then:

- Register bits[(x-1):1] are RES0.
- If the implementation supports 52-bit PAs and IPAs, then bits A[51:48] of the stage 1 translation table base address are `0b0000`.

If `FEAT_LPA` is implemented and the value of `TCR_EL2.{i}PS` is `0b110`, then:

- Bits A[51:48] of the stage 1 translation table base address bits are in register bits[5:2].
- Register bit[1] is RES0.
- When x>6, register bits[(x-1):6] are RES0.

---

**Note**

The OA size specified by `TCR_EL2.{i}PS` is determined as follows:

- The value of `TCR_EL2.PS` when the value of `HCR_EL2.E2H` is 0.
- The value of `TCR_EL2.IPS` when the value of `HCR_EL2.E2H` is 1.

`TCR_EL2.{i}PS==0b110` is permitted when:

- `FEAT_LPA` is implemented and the 64KB translation granule is used.
- `FEAT_LPA2` is implemented and the 4KB or 16KB translation granule is used.

When the value of `ID_A64MMFR0_EL1.PARange` indicates that the implementation does not support a 52 bit PA size, if a translation table lookup uses this register when the Effective value of `TCR_EL2.{i}PS` is `0b110` and the value of register bits[5:2] is nonzero, an Address size fault is generated.

---

If any register bit[47:1] that is defined as RES0 has the value 1 when a translation table walk is done using TTBR1_EL2, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits A[(x-1):0] of the stage 1 translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

---

**CnP, bit [0]**

*When FEAT_TTCNP is implemented:*

Common not Private. This bit indicates whether each entry that is pointed to by TBR1_EL2 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR1_EL2.CnP is 1.

*0b0* The translation table entries pointed to by TTBR1_EL2 for the current ASID are permitted to differ from corresponding entries for TTBR1_EL2 for other PEs in the Inner Shareable domain. This is not affected by:

- The value of TTBR1_EL2.CnP on those other PEs.
- The value of the current ASID.

*0b1* The translation table entries pointed to by TTBR1_EL2 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR1_EL2.CnP is 1 and all of the following apply:

- The translation table entries are pointed to by TTBR1_EL2.
- The ASID is the same as the current ASID.

This bit is permitted to be cached in a TLB.
Note

- TTBR1_EL2 is accessible only when the value of HCR_EL2.E2H is 1, meaning the current translation regime is the EL2&0 regime.
- If the value of the TTBR1_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR1_EL2s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values on page K1-11585.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TTBR1_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic TTBR1_EL2 or TTBR1_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TTBR1_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = TTBR1_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = TTBR1_EL2;

**MSR TTBR1_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    TTBR1_EL2 = X[t, 64];
```c
elsif PSTATE.EL == EL3
    TTBR1_EL2 = X[t, 64];

MRS <Xt>, TTBR1_EL1

    op0 op1 CRn CRm op2
    0b11 0b000 0b0010 0b0000 0b001

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFCRTR_EL2.TTBR1_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x210];
    else
        X[t, 64] = TTBR1_EL1;
    end;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        TTBR1_EL2 = X[t, 64];
    else
        TTBR1_EL1 = X[t, 64];
    end;
elsif PSTATE.EL == EL3 then
    TTBR1_EL1 = X[t, 64];

MSR TTBR1_EL1, <Xt>

    op0 op1 CRn CRm op2
    0b11 0b000 0b0010 0b0000 0b001

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFCRTR_EL2.TTBR1_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x210] = X[t, 64];
    else
        TTBR1_EL1 = X[t, 64];
    end;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        TTBR1_EL2 = X[t, 64];
    else
        TTBR1_EL1 = X[t, 64];
    end;
elsif PSTATE.EL == EL3 then
    TTBR1_EL1 = X[t, 64];
```
D17.2.149 VBAR_EL1, Vector Base Address Register (EL1)

The VBAR_EL1 characteristics are:

**Purpose**

Holds the vector base address for any exception that is taken to EL1.

**Configurations**

AArch64 System register VBAR_EL1 bits [31:0] are architecturally mapped to AArch32 System register VBAR[31:0].

**Attributes**

VBAR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Vector Base Address</td>
</tr>
<tr>
<td>32</td>
<td>Vector Base Address</td>
</tr>
<tr>
<td>31</td>
<td>Vector Base Address</td>
</tr>
<tr>
<td>11</td>
<td>Vector Base Address</td>
</tr>
<tr>
<td>10</td>
<td>RES0</td>
</tr>
<tr>
<td>0</td>
<td>RES0</td>
</tr>
</tbody>
</table>

**Bits [63:11]**

Vector Base Address. Base address of the exception vectors for exceptions taken to EL1.

--- **Note** ---

If the implementation does not support FEAT_LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

If the implementation supports FEAT_LVA, then:

- If tagged addresses are being used, bits [55:52] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

---

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [10:0]**

Reserved, RES0.

**Accessing VBAR_EL1**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic VBAR_EL1 or VBAR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:
### MRS <Xt>, VBAR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.VBAR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = NVMem[0x250];
  end
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    VBAR_EL2 = X[t, 64];
  else
    VBAR_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL3 then
  VBAR_EL1 = X[t, 64];

### MSR VBAR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.VBAR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    NVMem[0x250] = X[t, 64];
  else
    VBAR_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    VBAR_EL2 = X[t, 64];
  else
    VBAR_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL3 then
  VBAR_EL1 = X[t, 64];
\textbf{MRS <Xt>, VBAR\_EL12}

\begin{center}
\begin{tabular}{cccccc}
    \textbf{op0} & \textbf{op1} & \textbf{CRn} & \textbf{CRm} & \textbf{op2} \\
    0b11 & 0b101 & 0b1100 & 0b0000 & 0b0000 \\
\end{tabular}
\end{center}

\begin{verbatim}
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR\_EL2.\<NV2,\NV1,\NV> == '101' then
        X[t, 64] = NVMem[0x250];
    elseif EL2Enabled() && HCR\_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR\_EL2.E2H == '1' then
        X[t, 64] = VBAR\_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR\_EL2.E2H == '1' then
        VBAR\_EL1 = X[t, 64];
    else
        UNDEFINED;
end if
\end{verbatim}

\textbf{MSR VBAR\_EL12, <Xt>}

\begin{center}
\begin{tabular}{cccccc}
    \textbf{op0} & \textbf{op1} & \textbf{CRn} & \textbf{CRm} & \textbf{op2} \\
    0b11 & 0b101 & 0b1100 & 0b0000 & 0b0000 \\
\end{tabular}
\end{center}

\begin{verbatim}
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR\_EL2.\<NV2,\NV1,\NV> == '101' then
        NVMem[0x250] = X[t, 64];
    elseif EL2Enabled() && HCR\_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR\_EL2.E2H == '1' then
        VBAR\_EL1 = X[t, 64];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR\_EL2.E2H == '1' then
        VBAR\_EL1 = X[t, 64];
    else
        UNDEFINED;
end if
\end{verbatim}
D17.2.150  VBAR_EL2, Vector Base Address Register (EL2)

The VBAR_EL2 characteristics are:

**Purpose**

Holds the vector base address for any exception that is taken to EL2.

**Configurations**

AArch64 System register VBAR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HVBAR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

VBAR_EL2 is a 64-bit register.

**Field descriptions**

---

**Bits [63:11]**

Vector Base Address. Base address of the exception vectors for exceptions taken to EL2.

---

**Note**

If FEAT_LVA is implemented:

- If HCR_EL2.E2H == 0b1:
  - If tagged addresses are being used, bits [55:52] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.
  - If tagged addresses are not being used, bits [63:52] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.

- If HCR_EL2.E2H == 0b0:
  - If tagged addresses are being used, bits [55:52] of VBAR_EL2 must be 0 or else the use of the vector address will result in a recursive exception.
  - If tagged addresses are not being used, bits [63:52] of VBAR_EL2 must be 0 or else the use of the vector address will result in a recursive exception.

If FEAT_LVA is not implemented:

- If HCR_EL2.E2H == 0b1:
  - If tagged addresses are being used, bits [55:48] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.
  - If tagged addresses are not being used, bits [63:48] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.

- If HCR_EL2.E2H == 0b0:
  - If tagged addresses are being used, bits [55:48] of VBAR_EL2 must be 0 or else the use of the vector address will result in a recursive exception.
  - If tagged addresses are not being used, bits [63:48] of VBAR_EL2 must be 0 or else the use of the vector address will result in a recursive exception.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Bits [10:0]**

Reserved, RES0.

### Accessing VBAR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic VBAR_EL2 or VBAR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, VBAR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = VBAR_EL2;
elsif PSTATE.EL == EL3 then
  X[t, 64] = VBAR_EL2;
```

**MSR VBAR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  VBAR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
  VBAR_EL2 = X[t, 64];
```
MRS <Xt>, VBAR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWRTR_EL2.VBAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x250];
    else
        X[t, 64] = VBAR_EL1;
    elsif PSTATE.EL == EL2 then
        if HCR_EL2.E2H == '1' then
            VBAR_EL2 = X[t, 64];
        else
            VBAR_EL1 = X[t, 64];
        elsif PSTATE.EL == EL3 then
            VBAR_EL1 = X[t, 64];
    MSR VBAR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWRTR_EL2.VBAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x250] = X[t, 64];
    else
        VBAR_EL1 = X[t, 64];
    elsif PSTATE.EL == EL2 then
        if HCR_EL2.E2H == '1' then
            VBAR_EL2 = X[t, 64];
        else
            VBAR_EL1 = X[t, 64];
    elsif PSTATE.EL == EL3 then
        VBAR_EL1 = X[t, 64];
D17.2.151 VBAR_EL3, Vector Base Address Register (EL3)

The VBAR_EL3 characteristics are:

**Purpose**

Holds the vector base address for any exception that is taken to EL3.

**Configurations**

This register is present only when EL3 is implemented. Otherwise, direct accesses to VBAR_EL3 are UNDEFINED.

**Attributes**

VBAR_EL3 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:11]</td>
<td>Vector Base Address</td>
</tr>
<tr>
<td>[10:0]</td>
<td>Reserved, RES0</td>
</tr>
</tbody>
</table>

**Notes**

If the implementation does not support FEAT_LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL3 must be 0 or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:48] of VBAR_EL3 must be 0 or else the use of the vector address will result in a recursive exception.

If the implementation supports FEAT_LVA, then:

- If tagged addresses are being used, bits [55:52] of VBAR_EL3 must be 0 or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL3 must be 0 or else the use of the vector address will result in a recursive exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing VBAR_EL3**

Accesses to this register use the following encodings in the System register encoding space:
**MRS <Xt>, VBAR_EL3**

```assembly
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    X[t, 64] = VBAR_EL3;
```

**MSR VBAR_EL3, <Xt>**

```assembly
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    VBAR_EL3 = X[t, 64];
```
D17.2.152  VMPIDR_EL2, Virtualization Multiprocessor ID Register

The VMPIDR_EL2 characteristics are:

**Purpose**

Holds the value of the Virtualization Multiprocessor ID. This is the value returned by EL1 reads of MPIDR_EL1.

**Configurations**

AArch64 System register VMPIDR_EL2 bits [31:0] are architecturally mapped to AArch32 System register VMPIDR[31:0].

If EL2 is not implemented, reads of this register return the value of the MPIDR_EL1 and writes to the register are ignored.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

VMPIDR_EL2 is a 64-bit register.

**Field descriptions**

![Register Layout]

**Bits [63:40]**

Reserved, RES0.

**Aff3, bits [39:32]**

Affinity level 3. See the description of VMPIDR_EL2.Aff0 for more information.

Aff3 is not supported in AArch32 state.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [31]**

Reserved, RES1.

**U, bit [30]**

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

0b0  Processor is part of a multiprocessor system.

0b1  Processor is part of a uniprocessor system.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [29:25]**

Reserved, RES0.
MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multithreading type approach. See the description of VMPIDR_EL2.Aff0 for more information about affinity levels.

- **0b0** Performance of PEs at the lowest affinity level is largely independent.
- **0b1** Performance of PEs at the lowest affinity level is very interdependent.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Aff2, bits [23:16]**

Affinity level 2. See the description of VMPIDR_EL2.Aff0 for more information.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Aff1, bits [15:8]**

Affinity level 1. See the description of VMPIDR_EL2.Aff0 for more information.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Aff0, bits [7:0]**

Affinity level 0. This is the affinity level that is most significant for determining PE behavior. Higher affinity levels are increasingly less significant in determining PE behavior.

The assigned value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing VMPIDR_EL2**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, VMPIDR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = NVMem[0x050];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = VMPIDR_EL2;
elsif PSTATE.EL == EL3 then
  if !HaveEL(EL2) then
    X[t, 64] = MPIDR_EL1;
  else
    X[t, 64] = VMPIDR_EL2;
```
**MSR VMPIDR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() AND HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x050] = X[t, 64];
    elsif EL2Enabled() AND HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VMPIDR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        return;
    else
        VMPIDR_EL2 = X[t, 64];

**MRS <Xt>, MPIDR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() AND HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        else
            UNDEFINED;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() AND (HaveEL(EL3) || SCR_EL3.FGTEn == '1') AND HFCRTR_EL2.MPIDR_EL1 == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() then
            X[t, 64] = VMPIDR_EL2;
        else
            X[t, 64] = MPIDR_EL1;
        elsif PSTATE.EL == EL2 then
            X[t, 64] = MPIDR_EL1;
        elsif PSTATE.EL == EL3 then
            X[t, 64] = MPIDR_EL1;
D17.2.153 VNCR_EL2, Virtual Nested Control Register

The VNCR_EL2 characteristics are:

**Purpose**

When FEAT_NV2 is implemented, holds the base address that is used to define the memory location that is accessed by transformed reads and writes of System registers.

**Configurations**

This register is present only when FEAT_NV2 is implemented. Otherwise, direct accesses to VNCR_EL2 are UNDEFINED.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

VNCR_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-53</td>
<td>RESS</td>
</tr>
<tr>
<td>52-12</td>
<td>BADDR</td>
</tr>
<tr>
<td>11-0</td>
<td>RES0</td>
</tr>
</tbody>
</table>

**RESS, bits [63:53]**

Reserved, Sign extended. If the bits marked as RESS do not all have the same value, then there is a CONSTRAINED UNPREDICTABLE choice between:

- Generating an EL2 translation regime Translation abort on use of the VNCR_EL2 register.
- Bits[63:49] of VNCR_EL2 are treated as the same value as bit[48] for all purposes other than reading back the register.
- Bits[63:49] of VNCR_EL2 are treated as the same value as bit[48] for all purposes.
- If the virtual address space for EL2 supports more than 48 bits, bits[63:53] of VNCR_EL2 are treated as the same value as bit[52] for all purposes other than reading back the register.
- If the virtual address space for EL2 supports more than 48 bits, bits[63:53] of VNCR_EL2 are treated as the same value as bit[52].

Where the EL2 translation regime has upper and lower address ranges, bit[52] is used to select between those address ranges to determine if the address space supports more than 48 bits.

**BADDR, bits [52:12]**

Base Address. If the virtual address space for EL2 does not support more than 48 bits, then bits [52:49] are RESS.

When a register read/write is transformed to be a Load or Store, the address of the load/store is to SignOffset(VNCR_EL2.BADDR:Offset<11:0>, 64).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [11:0]**

Reserved, RES0.

**Accessing VNCR_EL2**

Accesses to this register use the following encodings in the System register encoding space:
**MRS <Xt>, VNCR_EL2**

```
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = NVMem[0x0B0];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE_EL == EL2 then
    X[t, 64] = VNCR_EL2;
elsif PSTATE_EL == EL3 then
    X[t, 64] = VNCR_EL2;
```

**MSR VNCR_EL2, <Xt>**

```
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x0B0] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE_EL == EL2 then
    VNCR_EL2 = X[t, 64];
elsif PSTATE_EL == EL3 then
    VNCR_EL2 = X[t, 64];
```
D17.2.154  VPIDR_EL2, Virtualization Processor ID Register

The VPIDR_EL2 characteristics are:

**Purpose**

Holds the value of the Virtualization Processor ID. This is the value returned by EL1 reads of MIDR_EL1.

**Configurations**

AArch64 System register VPIDR_EL2 bits [31:0] are architecturally mapped to AArch32 System register VPIDR[31:0].

If EL2 is not implemented, reads of this register return the value of the MIDR_EL1 and writes to the register are ignored.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

VPIDR_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>RES0</td>
</tr>
<tr>
<td>23</td>
<td>Implementer</td>
</tr>
<tr>
<td>19</td>
<td>Variant</td>
</tr>
<tr>
<td>15</td>
<td>PartNum</td>
</tr>
<tr>
<td>0</td>
<td>Revision</td>
</tr>
</tbody>
</table>

Bits [63:32] Reserved, RES0.

**Implementer, bits [31:24]**

The Implementer code. This field must hold an implementer code that has been assigned by Arm. Assigned codes include the following:

- 0x00 Reserved for software use.
- 0x41 Arm Limited.
- 0x42 Broadcom Corporation.
- 0x43 Cavium Inc.
- 0x44 Digital Equipment Corporation.
- 0x46 Fujitsu Ltd.
- 0x49 Infineon Technologies AG.
- 0x4D Motorola or Freescale Semiconductor Inc.
- 0x4E NVIDIA Corporation.
- 0x50 Applied Micro Circuits Corporation.
- 0x51 Qualcomm Inc.
- 0x56 Marvell International Ltd.
- 0x69 Intel Corporation.
- 0xC0 Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Variant, bits [23:20]**

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Architecture, bits [19:16]**

Architecture version. Defined values are:

- 0b0001  Armv4.
- 0b0010  Armv4T.
- 0b0011  Armv5 (obsolete).
- 0b0100  Armv5T.
- 0b0101  Armv5TE.
- 0b0110  Armv5TEJ.
- 0b0111  Armv6.
- 0b1111  Architectural features are individually identified in the ID_* registers.

All other values are reserved.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**PartNum, bits [15:4]**

An IMPLEMENTATION DEFINED primary part number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Revision, bits [3:0]**

An IMPLEMENTATION DEFINED revision number for the device.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing VPIDR_EL2**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, VPIDR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
 UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
X[t, 64] = NVMem[0x088];
 elsif EL2Enabled() && HCR_EL2.NV == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = VPIDR_EL2;
elif PSTATE.EL == EL3 then
  if !HaveEL(EL2) then
    X[t, 64] = MIDR_EL1;
  else
    X[t, 64] = VPIDR_EL2;

MSR VPIDR_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.NV == '11' then
    NVMem[0x088] = X[t, 64];
elif EL2Enabled() & HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elif PSTATE.EL == EL2 then
  VPIDR_EL2 = X[t, 64];
elif PSTATE.EL == EL3 then
  if !HaveEL(EL2) then
    return;
elif PSTATE.EL == EL3 then
  VPIDR_EL2 = X[t, 64];

MRS <Xt>, MIDR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if IsFeatureImplemented(FEAT_IDST) then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    UNDEFINED;
elif PSTATE.EL == EL1 then
  if EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFCRTR_EL2.MIDR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() then
  X[t, 64] = VPIDR_EL2;
elif PSTATE.EL == EL2 then
  X[t, 64] = MIDR_EL1;
elif PSTATE.EL == EL3 then
  X[t, 64] = MIDR_EL1;
D17.2.155  VSTCR_EL2, Virtualization Secure Translation Control Register

The VSTCR_EL2 characteristics are:

**Purpose**

The control register for stage 2 of the Secure EL1&0 translation regime.

**Configurations**

This register is present only when FEAT_SEL2 is implemented. Otherwise, direct accesses to VSTCR_EL2 are UNDEFINED.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

VSTCR_EL2 is a 64-bit register.

**Field descriptions**

Any of the bits in VSTCR_EL2 are permitted to be cached in a TLB.

**Bits [63:34]**

Reserved, RES0.

**SL2, bit [33]**

When FEAT_LPA2 is implemented:

Starting level of the Secure stage 2 translation lookup controlled by VSTCR_EL2.

- If VTCR_EL2.DS == 1, then VSTCR_EL2.SL2, in combination with VSTCR_EL2.SL0, gives encodings for the Secure stage 2 translation table walk initial lookup level.
- If VTCR_EL2.DS == 0, then VSTCR_EL2.SL2 is RES0.

If the translation granule size is not 4KB, then this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bit [32]**

Reserved, RES0.

**Bit [31]**

Reserved, RES1.

**SA, bit [30]**

Secure stage 2 translation output address space.

- 0b0  All stage 2 translations for the Secure IPA space access the Secure PA space.
- 0b1  All stage 2 translations for the Secure IPA space access the Non-secure PA space.
When the value of VSTCR_EL2.SW is 1, this bit behaves as 1 for all purposes other than reading back the value of the bit.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**SW, bit [29]**

Secure stage 2 translation address space.

- 0b0: All stage 2 translation table walks for the Secure IPA space are to the Secure PA space.
- 0b1: All stage 2 translation table walks for the Secure IPA space are to the Non-secure PA space.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [28:16]**

Reserved, RES0.

**TG0, bits [15:14]**

Secure stage 2 granule size for VSTTBR_EL2.

- 0b00: 4KB.
- 0b01: 64KB.
- 0b10: 16KB.

Other values are reserved.

If FEAT_GTG is implemented, ID_AA64MMFR0_EL1.{TGran4_2, TGran16_2, TGran64_2} indicate which granule sizes are supported for stage 2 translation.

If FEAT_GTG is not implemented, ID_AA64MMFR0_EL1.{TGran4, TGran16, TGran64} indicate which granule sizes are supported.

If the value is programmed to either a reserved value, or a size that has not been implemented, then for all purposes other than read back from this register, the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [13:8]**

Reserved, RES0.

**SL0, bits [7:6]**

*When FEAT_TTST is implemented:*

Starting level of the Secure stage 2 translation lookup, controlled by VSTCR_EL2. The meaning of this field depends on the value of VSTCR_EL2.TG0.

- 0b00: If VSTCR_EL2.TG0 is 0b00 (4KB granule):
  - If FEAT_LPA2 is not implemented, start at level 0.
  - If FEAT_LPA2 is implemented and VSTCR_EL2.SL2 is 0b0, start at level 2.
  - If FEAT_LPA2 is implemented and VSTCR_EL2.SL2 is 0b1, start at level -1.
  - If VSTCR_EL2.TG0 is 0b10 (16KB granule) or 0b01 (64KB granule), start at level 3.

- 0b01: If VSTCR_EL2.TG0 is 0b00 (4KB granule):
  - If FEAT_LPA2 is not implemented, start at level 1.
  - If FEAT_LPA2 is implemented and VSTCR_EL2.SL2 is 0b0, start at level 1.
If FEAT_LPA2 is implemented, the combination of VSTCR_EL2.SL0 == 01 and VSTCR_EL2.SL2 == 1 is reserved.

If VSTCR_EL2.TG0 is 0b10 (16KB granule) or 0b01 (64KB granule), start at level 2.

0b10
- If VSTCR_EL2.TG0 is 0b00 (4KB granule):
  - If FEAT_LPA2 is not implemented, start at level 0.
  - If FEAT_LPA2 is implemented and VSTCR_EL2.SL2 is 0b0, start at level 0.
  - If FEAT_LPA2 is implemented, the combination of VSTCR_EL2.SL0 == 10 and VSTCR_EL2.SL2 == 1 is reserved.

If VSTCR_EL2.TG0 is 0b10 (16KB granule) or 0b01 (64KB granule), start at level 1.

0b11
- If VSTCR_EL2.TG0 is not implemented, start at level 3.
- If FEAT_LPA2 is implemented and VSTCR_EL2.SL2 is 0b0, start at level 3.
- If FEAT_LPA2 is implemented, the combination of VSTCR_EL2.SL0 == 11 and VSTCR_EL2.SL2 == 1 is reserved.

If VSTCR_EL2.TG0 is 0b10 (16KB granule) and FEAT_LPA2 is implemented, start at level 0.

If this field is programmed to a value that is not consistent with the programming of VSTCR_EL2.T0SZ, then a stage 2 level 0 Translation fault is generated.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Starting level of the Secure stage 2 translation lookup, controlled by VSTCR_EL2. The meaning of this field depends on the value of VSTCR_EL2.TG0.

0b00
- If VSTCR_EL2.TG0 is 0b00 (4KB granule), start at level 2. If VSTCR_EL2.TG0 is 0b10 (16KB granule) or 0b01 (64KB granule), start at level 3.

0b01
- If VSTCR_EL2.TG0 is 0b00 (4KB granule), start at level 1. If VSTCR_EL2.TG0 is 0b10 (16KB granule) or 0b01 (64KB granule), start at level 2.

0b10
- If VSTCR_EL2.TG0 is 0b00 (4KB granule), start at level 0. If VSTCR_EL2.TG0 is 0b10 (16KB granule) or 0b01 (64KB granule), start at level 1.

All other values are reserved. If this field is programmed to a reserved value, or to a value that is not consistent with the programming of VSTCR_EL2.T0SZ, then a stage 2 level 0 Translation fault is generated.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T0SZ, bits [5:0]

The size offset of the memory region addressed by VSTTBR_EL2. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for this field depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

If this field is programmed to a value that is not consistent with the programming of SL0, then a stage 2 level 0 Translation fault is generated.

--- Note ---

For the 4KB translation granule, if FEAT_LPA2 is implemented and this field is less than 16, the translation table walk begins with a level -1 initial lookup.

For the 16KB translation granule, if FEAT_LPA2 is implemented and this field is less than 17, the translation table walk begins with a level 0 initial lookup.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VSTCR_EL2
Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, VSTCR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0010</td>
<td>0b0110</td>
<td>0b010</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  else
    X[t, 64] = NVMem[0x048];
  end if;
else
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = NVMem[0x048];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
  end if;
else
  if PSTATE_EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
      UNDEFINED;
    else
      X[t, 64] = VSTCR_EL2;
    end if;
e elseif PSTATE_EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
      UNDEFINED;
    else
      X[t, 64] = VSTCR_EL2;
    end if;
  end if;
```

**MSR VSTCR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0010</td>
<td>0b0110</td>
<td>0b010</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  else
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
      NVMem[0x048] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      UNDEFINED;
    end if;
  end if;
e elseif PSTATE_EL == EL2 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  else
    VSTCR_EL2 = X[t, 64];
  end if;
e elseif PSTATE_EL == EL3 then
  if SCR_EL3.EEL2 == '0' then
    UNDEFINED;
  end if;
```

```
UNDEFINED;
else
   VSTCR_EL2 = X[t, 64];
D17.2.156 VSTTBR_EL2, Virtualization Secure Translation Table Base Register

The VSTTBR_EL2 characteristics are:

**Purpose**

The base register for stage 2 of the Secure EL1&0 translation regime. Holds the base address of the translation table for the initial lookup for stage 2 of an address translation in the Secure EL1&0 translation regime, and other information for this translation stage.

**Configurations**

This register is present only when FEAT_SEL2 is implemented. Otherwise, direct accesses to VSTTBR_EL2 are **UNDEFINED**.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

VSTTBR_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:48]</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
</tbody>
</table>

**Note**

A translation table must be aligned to the size of the table, except that when using a translation table base address larger than 48 bits the minimum alignment of a table containing fewer than eight entries is 64 bytes.

If the value of VTCR_EL2.PS is 0b110, then:

- Register bits[47:z] hold bits[47:z] of the stage 1 translation table base address, where z is determined as follows:
  - If x >= 6 then z=x.
  - Otherwise, z=6.
- When z>x register bits[(z-1):x] are RES0, and bits[(z-1):x] of the translation table base address are zero.
- When x>6 register bits[(x-1):6] are RES0.
- Register bit[1] is RES0.
- Bits[5:2] of the stage 1 translation table base address are zero.
Note

When the value of ID_AA64MMFR0_EL1.PARange indicates that the implementation does not support a 52-bit PA size, if a translation table lookup uses this register with the 64KB translation granule when the Effective value of VTCR_EL2.PS is 0b110 and the value of register bits[5:2] is nonzero, an Address size fault is generated.

If the Effective value of VTCR_EL2.PS is not 0b110, then:

- Register bits[47:x] hold bits[47:x] of the stage 1 translation table base address.
- Register bits[(x-1):1] are RES0.
- If the implementation supports 52-bit PAs and IPAs then bits[51:48] of the translation table base addresses used in this stage of translation are 0b0000.

If any VSTTBR_EL2[47:1] bit that is defined as RES0 has the value 1 when a translation table walk is performed using VSTTBR_EL2, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits[(x-1):0] of the translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of VSTCR_EL2.T0SZ, the stage of translation, and the translation granule size.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

Common not Private, for stage 2 of the Secure EL1&0 translation regime. In an implementation that includes FEAT_TTCNP, indicates whether each entry that is pointed to by VSTTBR_EL2 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of VSTTBR_EL2.CnP is 1.

0b0 The translation table entries pointed to by VSTTBR_EL2 are permitted to differ from the entries for VSTTBR_EL2 for other PEs in the Inner Shareable domain. This is not affected by the value of the current VMID.

0b1 The translation table entries pointed to by VSTTBR_EL2 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of VSTTBR_EL2.CnP is 1 and the VMID is the same as the current VMID.

This bit is permitted to be cached in a TLB.

Note

If the value of VSTTBR_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those VSTTBR_EL2s do not point to the same translation table entries when using the current VMID, then the results of translations using VSTTBR_EL2 are CONSTRAINED UNPREDICTABLE, see CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values on page K1-11585.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VSTTBR_EL2

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, VSTTBR_EL2

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0010</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = NVMem[0x030];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  else
    X[t, 64] = VSTTBR_EL2;
elsif PSTATE_EL == EL3 then
  if SCR_EL3.EEL2 == '0' then
    UNDEFINED;
  else
    X[t, 64] = VSTTBR_EL2;

MSR VSTTBR_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0010</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    NVMem[0x030] = X[t, 64];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  else
    VSTTBR_EL2 = X[t, 64];
elsif PSTATE_EL == EL3 then
  if SCR_EL3.EEL2 == '0' then
    UNDEFINED;
  else
    VSTTBR_EL2 = X[t, 64];
D17.2.157 VTCR_EL2, Virtualization Translation Control Register

The VTCR_EL2 characteristics are:

**Purpose**

The control register for stage 2 of the EL1&0 translation regime.

**Configurations**

AArch64 System register VTCR_EL2 bits [31:0] are architecturally mapped to AArch32 System register VTCR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

VTCR_EL2 is a 64-bit register.

**Field descriptions**

Any of the bits in VTCR_EL2 are permitted to be cached in a TLB.

**Bits [63:34]**

Reserved, RES0.

**SL2, bit [33]**

*When FEAT_LPA2 is implemented:*

Starting level of the stage 2 translation lookup controlled by VTCR_EL2.

If VTCR_EL2.DS == 1, then VTCR_EL2.SL2, in combination with VTCR_EL2.SL0, gives encodings for the stage 2 translation table walk initial lookup level.

If VTCR_EL2.DS == 0, then VTCR_EL2.SL2 is RES0.

If the translation granule size is not 4KB, then this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**DS, bit [32]**

*When FEAT_LPA2 is implemented:*

This field affects 52-bit output addressing when using 4KB and 16KB translation granules in stage 2 of the EL1&0 translation regime.

0b0: Bits[49:48] of translation descriptors are RES0.

The minimum value of VTCR_EL2.T0SZ is 16. Any memory access using a smaller value generates a stage 2 level 0 translation table fault.

The minimum value of VSTCR_EL2.T0SZ is 16. Any memory access using a smaller value generates a stage 2 level 0 translation table fault.

Output address[51:48] is 0000.

0b1 Bits[49:48] of translation descriptors hold output address[49:48].

Bits[9:8] in translation descriptors hold output address[50:48].

The shareability information of Block and Page descriptors for cacheable locations is determined by VTCR_EL2.SH0.

The minimum value of VTCR_EL2.T0SZ is 12. Any memory access using a smaller value generates a stage 2 level 0 translation table fault.

The minimum value of VSTCR_EL2.T0SZ is 12. Any memory access using a smaller value generates a stage 2 level 0 translation table fault.

--- Note ---

As FEAT_LPA must be implemented if VTCR_EL2.DS == 1, the minimum values of VTCR_EL2.T0SZ and VSTCR_EL2.T0SZ are 12, as determined by that extension.

--- Note ---

For the TLBI range instructions affecting IPA, the format of the argument is changed so that bits[36:0] hold BaseADDR[52:16]. For the 4KB translation granule, bits[15:12] of BaseADDR are treated as 0000. For the 16KB translation granule, bits[15:14] of BaseADDR are treated as 00.

--- Note ---

This forces alignment of the ranges used by the TLBI range instructions.

This field is RES0 for a 64KB translation granule.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [31]

Reserved, RES1.

NSA, bit [30]

When FEAT_SEL2 is implemented:

Non-secure stage 2 translation output address space for the Secure EL1&0 translation regime.

0b0 All stage 2 translations for the Non-secure IPA space of the Secure EL1&0 translation regime access the Secure PA space.

0b1 All stage 2 translations for the Non-secure IPA space of the Secure EL1&0 translation regime access the Non-secure PA space.

This bit behaves as 1 for all purposes other than reading back the value of the bit when one of the following is true:

- The value of VTCR_EL2.NSW is 1.
- The value of VSTCR_EL2.SA is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
NSW, bit [29]

When FEAT_SEL2 is implemented:

Non-secure stage 2 translation table address space for the Secure EL1&0 translation regime.

0b0  All stage 2 translation table walks for the Non-secure IPA space of the Secure EL1&0 translation regime are to the Secure PA space.

0b1  All stage 2 translation table walks for the Non-secure IPA space of the Secure EL1&0 translation regime are to the Non-secure PA space.

The reset behavior of this field is:

•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU62, bit [28]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 2 translation table Block or Page entry.

0b0  Bit[62] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1  Bit[62] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

The reset behavior of this field is:

•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU61, bit [27]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 2 translation table Block or Page entry.

0b0  Bit[61] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1  Bit[61] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

The reset behavior of this field is:

•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU60, bit [26]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 2 translation table Block or Page entry.

0b0  Bit[60] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1  Bit[60] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

The reset behavior of this field is:

•  On a Warm reset, this field resets to an architecturally UNKNOWN value.
**HWU59, bit [25]**

*When FEAT_HPDS2 is implemented:*

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 2 translation table Block or Page entry.

- **0b0**: Bit[59] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
- **0b1**: Bit[59] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bits [24:23]**

Reserved, RES0.

**HD, bit [22]**

*When FEAT_HAFDBS is implemented:*

Hardware management of dirty state in stage 2 translations when EL2 is enabled in the current Security state.

- **0b0**: Stage 2 hardware management of dirty state disabled.
- **0b1**: Stage 2 hardware management of dirty state enabled, only if the VTCR_EL2.HA bit is also set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**HA, bit [21]**

*When FEAT_HAFDBS is implemented:*

Hardware Access flag update in stage 2 translations when EL2 is enabled in the current Security state.

- **0b0**: Stage 2 Access flag update disabled.
- **0b1**: Stage 2 Access flag update enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bit [20]**

Reserved, RES0.

**VS, bit [19]**

*When FEAT_VMID16 is implemented:*

VMID Size.

- **0b0**: 8-bit VMID. The upper 8 bits of VTTBR_EL2 are ignored by the hardware, and treated as if they are all zeros, for every purpose except when reading back the register.
0b1 16-bit VMID. The upper 8 bits of VTTBR_EL2 are used for allocation and matching in the TLB.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

PS, bits [18:16]
Physical address Size for the second stage of translation.
- 0b000 32 bits, 4GB.
- 0b001 36 bits, 64GB.
- 0b010 40 bits, 1TB.
- 0b011 42 bits, 4TB.
- 0b100 44 bits, 16TB.
- 0b101 48 bits, 256TB.
- 0b110 52 bits, 4PB.
All other values are reserved.

The reserved values behave in the same way as the 0b101 or 0b110 encoding, but software must not rely on this property as the behavior of the reserved values might change in a future revision of the architecture.

If the translation granule is not 64KB and FEAT_LPA2 is not implemented, the value 0b110 is treated as reserved.

It is IMPLEMENTATION DEFINED whether an implementation that does not implement FEAT_LPA supports setting the value of 0b110 for the 64KB translation granule size or whether setting this value behaves as the 0b101 encoding.

In an implementation that supports 52-bit PAs, if the value of this field is not 0b110 or a value treated as 0b110, then bits[51:48] of every translation table base address for the stage of translation controlled by VTCR_EL2 are 0b0000.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TG0, bits [15:14]
Granule size for the VTTBR_EL2.
- 0b0 4KB.
- 0b1 64KB.
- 0b10 16KB.
Other values are reserved.

If FEAT_GTG is implemented, ID_AA64MMFR0_EL1.{TGran4_2, TGran16_2, TGran64_2} indicate which granule sizes are supported for stage 2 translation.

If FEAT_GTG is not implemented, ID_AA64MMFR0_EL1.{TGran4, TGran16, TGran64} indicate which granule sizes are supported.

If the value is programmed to either a reserved value or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using VTTBR_EL2 or VSTTBR_EL2.

- \texttt{0b00} Non-shareable.
- \texttt{0b10} Outer Shareable.
- \texttt{0b11} Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is \textit{CONSTRAINED UNPREDICTABLE}.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally \textit{UNKNOWN} value.

ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using VTTBR_EL2 or VSTTBR_EL2.

- \texttt{0b00} Normal memory, Outer Non-cacheable.
- \texttt{0b01} Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
- \texttt{0b10} Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
- \texttt{0b11} Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally \textit{UNKNOWN} value.

IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using VTTBR_EL2 or VSTTBR_EL2.

- \texttt{0b00} Normal memory, Inner Non-cacheable.
- \texttt{0b01} Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
- \texttt{0b10} Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
- \texttt{0b11} Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally \textit{UNKNOWN} value.

SL0, bits [7:6]

\textit{When FEAT\_TTST is implemented:}

Starting level of the stage 2 translation lookup, controlled by VTCR\_EL2. The meaning of this field depends on the value of VTCR\_EL2.TG0.

- \texttt{0b00} If VTCR\_EL2.TG0 is \texttt{0b00} (4KB granule):
  - If \texttt{FEAT\_LPA2} is not implemented, start at level 2.
  - If \texttt{FEAT\_LPA2} is implemented and VTCR\_EL2.SL2 is \texttt{0b0}, start at level 2.
  - If \texttt{FEAT\_LPA2} is implemented and VTCR\_EL2.SL2 is \texttt{0b1}, start at level -1.

- \texttt{0b01} If VTCR\_EL2.TG0 is \texttt{0b01} (4KB granule):
  - If \texttt{FEAT\_LPA2} is not implemented, start at level 1.
  - If \texttt{FEAT\_LPA2} is implemented and VTCR\_EL2.SL2 is \texttt{0b0}, start at level 1.
  - If \texttt{FEAT\_LPA2} is implemented, the combination of VTCR\_EL2.SL0 == 01 and VTCR\_EL2.SL2 == 1 is reserved.

- \texttt{0b10} If VTCR\_EL2.TG0 is \texttt{0b10} (16KB granule) or \texttt{0b11} (64KB granule), start at level 2.
• If FEAT_LPA2 is implemented and VTCR_EL2.SL2 is 0b0, start at level 0.
• If FEAT_LPA2 is implemented, the combination of VTCR_EL2.SL0 == 10 and VTCR_EL2.SL2 == 1 is reserved.
If VTCR_EL2.TG0 is 0b10 (16KB granule) or 0b01 (64KB granule), start at level 1.

If VTCR_EL2.TG0 is 0b11 (4KB granule):
• If FEAT_LPA2 is not implemented, start at level 3.
• If FEAT_LPA2 is implemented and VTCR_EL2.SL2 is 0b0, start at level 3.
• If FEAT_LPA2 is implemented, the combination of VTCR_EL2.SL0 == 11 and VTCR_EL2.SL2 == 1 is reserved.
If VTCR_EL2.TG0 is 0b10 (16KB granule) and FEAT_LPA2 is implemented, start at level 0.

If this field is programmed to a value that is not consistent with the programming of VTCR_EL2.T0SZ, then a stage 2 level 0 Translation fault is generated.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Starting level of the stage 2 translation lookup, controlled by VTCR_EL2. The meaning of this field depends on the value of VTCR_EL2.TG0.

If VTCR_EL2.TG0 is 0b00 (4KB granule), start at level 2. If VTCR_EL2.TG0 is 0b10 (16KB granule) or 0b01 (64KB granule), start at level 3.

• If FEAT_LPA2 is implemented and VTCR_EL2.SL2 is 0b0, start at level 3.
• If FEAT_LPA2 is implemented, the combination of VTCR_EL2.SL0 == 11 and VTCR_EL2.SL2 == 1 is reserved.
If VTCR_EL2.TG0 is 0b10 (16KB granule) and FEAT_LPA2 is implemented, start at level 0.

If this field is programmed to a value that is not consistent with the programming of VTCR_EL2.T0SZ, then a stage 2 level 0 Translation fault is generated.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

T0SZ, bits [5:0]
The size offset of the memory region addressed by VTTBR_EL2. The region size is $2^{(64-T0SZ)}$ bytes.
The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in Chapter D8 The AArch64 Virtual Memory System Architecture.

If this field is programmed to a value that is not consistent with the programming of SL0, then a stage 2 level 0 Translation fault is generated.

--- Note ---
For the 4KB translation granule, if FEAT_LPA2 is implemented and this field is less than 16, the translation table walk begins with a level -1 initial lookup.
For the 16KB translation granule, if FEAT_LPA2 is implemented and this field is less than 17, the translation table walk begins with a level 0 initial lookup.

---

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VTCR_EL2
Any of the bits in VTCR_EL2 are permitted to be cached in a TLB.

Accesses to this register use the following encodings in the System register encoding space:
**MRS <Xt>, VTCR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0010</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = NVMem[0x040];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = VTCR_EL2;
elsif PSTATE.EL == EL3 then
  X[t, 64] = VTCR_EL2;

**MSR VTCR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0010</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    NVMem[0x040] = X[t, 64];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  VTCR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
  VTCR_EL2 = X[t, 64];
D17.2.158 VTTBR_EL2, Virtualization Translation Table Base Register

The VTTBR_EL2 characteristics are:

**Purpose**

Holds the base address of the translation table for the initial lookup for stage 2 of an address translation in the EL1&0 translation regime, and other information for this translation regime.

**Configurations**

AArch64 System register VTTBR_EL2 bits [63:0] are architecturally mapped to AArch32 System register VTTBR[63:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

VTTBR_EL2 is a 64-bit register.

**Field descriptions**

**VMID, bits [63:48]**

VMID encoding when FEAT_VMID16 is implemented and VTCR_EL2.VS == 1

**VMID, bits [15:0]**

The VMID for the translation table.

If the implementation has an 8-bit VMID, bits [15:8] of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**VMID, bits [15:0]**

The VMID for the translation table.

If the implementation has an 8-bit VMID, bits [15:8] of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [15:8]**

Reserved, RES0.

**VMID, bits [7:0]**

The VMID for the translation table.

The VMID is 8 bits when any of the following are true:

- EL2 is using AArch32.
- VTCR_EL2.VS is 0.
- **FEAT_VMID16** is not implemented.
  The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**BADDR, bits [47:1]**

Translation table base address, A[47:x] or A[51:x], bits[47:1].

--- **Note**

A translation table must be aligned to the size of the table, except that when using a translation table base address larger than 48 bits the minimum alignment of a table containing fewer than eight entries is 64 bytes.

In an implementation that includes **FEAT_LPA**, if the value of **VTCR_EL2.PS** is 0b110, then:

- Register bits[47:z] hold bits[47:z] of the stage 1 translation table base address, where z is determined as follows:
  - If x >= 6 then z=x.
  - Otherwise, z=6.
- When z>x register bits[(z-1):x] are RES0, and bits[(z-1):x] of the translation table base address are zero.
- When x>6 register bits[(x-1):6] are RES0.
- Register bit[1] is RES0.
- Bits[5:2] of the stage 1 translation table base address are zero.
- In an implementation that includes **FEAT_TTCNP**, bit[0] of the stage 1 translation table base address is zero.

--- **Note**

When the value of **ID_AA64MMFR0_EL1.PARange** indicates that the implementation does not support a 52 bit PA size, if a translation table lookup uses this register when the **Effective value** of **VTCR_EL2.PS** is 0b110 and the value of register bits[5:2] is nonzero, an Address size fault is generated.

If the Effective value of **VTCR_EL2.PS** is not 0b110 then:

- Register bits[47:x] hold bits[47:x] of the stage 1 translation table base address.
- Register bits[(x-1):1] are RES0.
- If the implementation supports 52-bit PAs and IPAs then bits[51:48] of the translation table base addresses used in this stage of translation are 0b0000.

If any **VTTBR_EL2[47:0]** bit that is defined as RES0 has the value 1 when a translation table walk is performed using **VTTBR_EL2**, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits[x-1:0] of the translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of **VTCR_EL2.T0SZ**, the stage of translation, and the translation granule size.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
CnP, bit [0]

**When FEAT_TTCNP is implemented:**

Common not Private. This bit indicates whether each entry that is pointed to by VTTBR_EL2 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of VTTBR_EL2.CnP is 1.

0b0  The translation table entries pointed to by VTTBR_EL2 are permitted to differ from the entries for VTTBR_EL2 for other PEs in the Inner Shareable domain. This is not affected by the value of the current VMID.

0b1  The translation table entries pointed to by VTTBR_EL2 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of VTTBR_EL2.CnP is 1 and the VMID is the same as the current VMID.

This bit is permitted to be cached in a TLB.

--- **Note** ---

If the value of VTTBR_EL2.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those VTTBR_EL2s do not point to the same translation table entries when using the current VMID then the results of translations using VTTBR_EL2 are CONSTRAINED UNPREDICTABLE, see **CONSTRANED UNPREDICTABLE behaviors due to caching of control or data values on page K1-11585**.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

Accessing VTTBR_EL2

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, VTTBR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0010</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = NVMem[0x020];
  elsif EL2Enabled() & HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
  elsif PSTATE.EL == EL2 then
    X[t, 64] = VTTBR_EL2;
else
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  X[t, 64] = VTTBR_EL2;
```

---
### MSR VTTBR_EL2, < Xt >

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0010</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elself PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    NVMem[0x020] = X[t, 64];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
  elself PSTATE_EL == EL2 then
    VTTBR_EL2 = X[t, 64];
  elself PSTATE_EL == EL3 then
    VTTBR_EL2 = X[t, 64];
D17.2.159 ZCR_EL1, SVE Control Register (EL1)

The ZCR_EL1 characteristics are:

**Purpose**

This register controls aspects of SVE visible at Exception levels EL1 and EL0.

**Configurations**

This register is present only when FEAT_SVE is implemented. Otherwise, direct accesses to ZCR_EL1 are UNDEFINED.

This register has no effect when FEAT_SME is implemented and the PE is in Streaming SVE mode.

When HCR_EL2.{E2H, TGE} == {1, 1} and EL2 is enabled in the current Security state, this register has no effect on execution at EL0.

**Attributes**

ZCR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit 63</th>
<th>Bit 32</th>
<th>Bit 31</th>
<th>Bit 9</th>
<th>Bit 8</th>
<th>Bit 4</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>RES0</td>
<td>RAZ/WI</td>
<td>LEN</td>
<td>LEN</td>
<td>LEN</td>
<td>LEN</td>
</tr>
</tbody>
</table>

Bits [63:9]

Reserved, RES0.

Bits [8:4]

Reserved, RAZ/WI.

LEN, bits [3:0]

Requests an Effective Non-streaming SVE vector length at EL1 of (LEN+1)*128 bits. This field also defines the Effective Non-streaming SVE vector length at EL0 when EL2 is not implemented, or EL2 is not enabled in the current Security state, or HCR_EL2.{E2H,TGE} is not {1,1}.

The Non-streaming SVE vector length can be any multiple of 128 bits, from 128 bits to 2048 bits inclusive. An implementation can support a subset of the architecturally permitted lengths. An implementation is required to support all lengths that are powers of two, from 128 bits up to its maximum implemented Non-streaming SVE vector length.

When FEAT_SME is not implemented, or the PE is not in Streaming SVE mode, the Effective SVE vector length (VL) is equal to the Effective Non-streaming SVE vector length.

When FEAT_SME is implemented and the PE is in Streaming SVE mode, VL is equal to the Effective Streaming SVE vector length. See SMCR_EL1.

For all purposes other than returning the result of a direct read of ZCR_EL1, the PE selects the Effective Non-streaming SVE vector length by performing checks in the following order:

1. If EL2 is implemented and enabled in the current Security state, and the requested length is greater than the Effective length at EL2, then the Effective length at EL2 is used.
2. If EL3 is implemented and the requested length is greater than the Effective length at EL3, then the Effective length at EL3 is used.
3. Otherwise, the Effective length is the highest supported Non-streaming SVE vector length that is less than or equal to the requested length.

An indirect read of ZCR_EL1.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ZCR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic ZCR_EL1 or ZCR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

_MRS <Xt>, ZCR_EL1_

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.EZ == '0' then
    UNDEFINED;
  elsif CPACR_EL1.ZEN == 'x0' then
    AArch64.SystemAccessTrap(EL1, 0x19);
  elsif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TZ == '1' then
    AArch64.SystemAccessTrap(EL2, 0x19);
  elsif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.ZEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x19);
  elsif HCR_EL2.E2H == '0' && CPTR_EL3.EZ == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    elsif EL2Enabled() && CPTR_EL1.ZEN == 'x0' then
      AArch64.SystemAccessTrap(EL2, 0x19);
    elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL3.EZ == '0' then
      if HaveEL(EL3) && CPTR_EL3.EZ == '0' then
        if Halted() && EDSCR.SDD == '1' then
          UNDEFINED;
        elsif EL2Enabled() && CPTR_EL2.EZ == '0' then
          AArch64.SystemAccessTrap(EL2, 0x19);
        elsif CPACR_EL1.ZEN == '0' then
          AArch64.SystemAccessTrap(EL3, 0x19);
        elsif Halted() && CPTR_EL3.EZ == '0' then
          if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
          elsif EL2Enabled() && CPTR_EL1.EZ == '0' then
            AArch64.SystemAccessTrap(EL1, 0x19);
          elsif CPTR_EL3.EZ == '0' then
            AArch64.SystemAccessTrap(EL3, 0x19);
          else
            X[t, 64] = ZCR_EL1;
        else
          X[t, 64] = ZCR_EL1;
      else
        X[t, 64] = ZCR_EL1;
      else
        X[t, 64] = ZCR_EL1;
  else
    X[t, 64] = ZCR_EL1;
elsif PSTATE.EL == EL2 then
  if PSTATE.EL == EL0 then
    UNDEFINED;
  elsif PSTATE.EL == EL1 then
    if Halted() && HaveEl(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.EZ == '0' then
      UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TZ == '1' then
      AArch64.SystemAccessTrap(EL2, 0x19);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.ZEN == 'x0' then
      AArch64.SystemAccessTrap(EL2, 0x19);
    elsif HaveEL(EL3) && CPTR_EL3.EZ == '0' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      elsif EL2Enabled() && CPTR_EL1.ZEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x19);
      elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.ZEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x19);
      elsif Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      elsif CPACR_EL1.ZEN == '0' then
        AArch64.SystemAccessTrap(EL3, 0x19);
      elsif HCR_EL2.E2H == '1' && CPTR_EL3.EZ == '0' then
        if Halted() && EDSCR.SDD == '1' then
          UNDEFINED;
        elsif CPTR_EL3.EZ == '0' then
          AArch64.SystemAccessTrap(EL3, 0x19);
        else
          X[t, 64] = ZCR_EL1;
      else
        X[t, 64] = ZCR_EL1;
  else
    X[t, 64] = ZCR_EL1;
**MSR ZCR_EL1, <Xi>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.EZ == '0' then
    UNDEFINED;
  elsif CPACR_EL1.ZEN == 'x0' then
    AArch64.SystemAccessTrap(EL1, 0x19);
  elsif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TZ == '1' then
    AArch64.SystemAccessTrap(EL2, 0x19);
  elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.ZEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x19);
  elsif HaveEL(EL3) && CPTR_EL3.EZ == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x19);
    end if
  elsif HCR_EL2.E2H == '1' then
    ZCR_EL2 = X[t, 64];
  else
    ZCR_EL1 = X[t, 64];
  end if
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.EZ == '0' then
    UNDEFINED;
  elsif HCR_EL2.E2H == '0' && CPTR_EL2.TZ == '1' then
    AArch64.SystemAccessTrap(EL2, 0x19);
  elsif HCR_EL2.E2H == '1' && CPTR_EL2.ZEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x19);
  elsif HaveEL(EL3) && CPTR_EL3.EZ == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x19);
    end if
  else
    ZCR_EL1 = X[t, 64];
  end if
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.EZ == '0' then
    AArch64.SystemAccessTrap(EL3, 0x19);
  else
    ZCR_EL1 = X[t, 64];
  end if

**MRS <Xt>, ZCR_EL12**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    X[t, 64] = NVMem[0x1E0];
else
  ZCR_EL1 = X[t, 64];
  if PSTATE.EL == EL3 then
    if CPTR_EL3.EZ == '0' then
      AArch64.SystemAccessTrap(EL3, 0x19);
    else
      ZCR_EL1 = X[t, 64];
    end if
  end if
endif
endif
elsif EL2Enabled() && HCR_EL2.NV == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.EZ == '0' then
      UNDEFINED;
    elsif CPTR_EL2.ZEN == 'x0' then
      AArch64.SystemAccessTrap(EL2, 0x19);
    elsif HaveEL(EL3) && CPTR_EL3.EZ == '0' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x19);
      end else
        AArch64.SystemAccessTrap(EL2, 0x19);
      end else
        X[t, 64] = ZCR_EL1;
      else
        UNDEFINED;
else
  elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
      if CPTR_EL3.EZ == '0' then
        AArch64.SystemAccessTrap(EL3, 0x19);
      else
        X[t, 64] = ZCR_EL1;
      else
        ZCR_EL1 = X[t, 64];
      else
  endif
endif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    NMem[0x1E0] = X[t, 64];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.EZ == '0' then
      UNDEFINED;
    elsif CPTR_EL2.ZEN == 'x0' then
      AArch64.SystemAccessTrap(EL2, 0x19);
    elsif HaveEL(EL3) && CPTR_EL3.EZ == '0' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(El3, 0x19);
      else
        ZCR_EL1 = X[t, 64];
      else
  endif
endif PSTATE.EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    if CPTR_EL3.EZ == '0' then
      AArch64.SystemAccessTrap(EL3, 0x19);
    else

MISR ZCR_EL12, <X>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b10</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>
else
    ZCR_EL1 = X[t, 64];
else
    UNDEFINED;
D17.2.160  ZCR_EL2, SVE Control Register (EL2)

The ZCR_EL2 characteristics are:

Purpose

This register controls aspects of SVE visible at Exception levels EL2, EL1, and EL0.

Configurations

This register is present only when FEAT_SVE is implemented. Otherwise, direct accesses to ZCR_EL2 are UNDEFINED.

This register has no effect when EL2 is not enabled in the current Security state, or when FEAT_SME is implemented and the PE is in Streaming SVE mode.

Attributes

ZCR_EL2 is a 64-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:9</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>8:4</td>
<td>Reserved, RAZ/WI</td>
</tr>
<tr>
<td>3:0</td>
<td>LEN</td>
</tr>
</tbody>
</table>

Bits [63:9]

Reserved, RES0.

Bits [8:4]

Reserved, RAZ/WI.

LEN, bits [3:0]

Requests an Effective Non-streaming SVE vector length at EL2 of (LEN+1)*128 bits. This field also defines the Effective Non-streaming SVE vector length at EL0 when EL2 is implemented and enabled in the current Security state, and HCR_EL2.{E2H,TGE} is {1,1}.

The Non-streaming SVE vector length can be any multiple of 128 bits, from 128 bits to 2048 bits inclusive. An implementation can support a subset of the architecturally permitted lengths. An implementation is required to support all lengths that are powers of two, from 128 bits up to its maximum implemented Non-streaming SVE vector length.

When FEAT_SME is not implemented, or the PE is not in Streaming SVE mode, the Effective SVE vector length (VL) is equal to the Effective Non-streaming SVE vector length.

When FEAT_SME is implemented and the PE is in Streaming SVE mode, VL is equal to the Effective Streaming SVE vector length. See SMCR_EL2.

For all purposes other than returning the result of a direct read of ZCR_EL2, the PE selects the Effective Non-streaming SVE vector length by performing checks in the following order:

1. If EL3 is implemented and the requested length is greater than the Effective length at EL3, then the Effective length at EL3 is used.
2. Otherwise, the Effective length is the highest supported Non-streaming SVE vector length that is less than or equal to the requested length.

An indirect read of ZCR_EL2.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Accessing ZCR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic ZCR_EL2 or ZCR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ZCR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.EZ == '0' then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TZ == '1' then
        AArch64.SystemAccessTrap(EL2, 0x19);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.ZEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x19);
    elsif HaveEL(EL3) && CPTR_EL3.EZ == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x19);
        else
            X[t, 64] = ZCR_EL2;
    elsif PSTATE.EL == EL3 then
        if CPTR_EL3.EZ == '0' then
            AArch64.SystemAccessTrap(EL3, 0x19);
    else
        X[t, 64] = ZCR_EL2;

**MSR ZCR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.EZ == '0' then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TZ == '1' then
        AArch64.SystemAccessTrap(EL2, 0x19);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.ZEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x19);
elsif HCR_EL2.E2H == '1' && CPTR_EL2.ZEN == 'x0' then
  AArch64.SystemAccessTrap(EL2, 0x19);
elsif HaveEL(EL3) && CPTR_EL3.ZEN == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x19);
else
  ZCR_EL2 = X[t, 64];
elsif PSTATE_EL == EL3 then
  if CPTR_EL3.ZEN == '0' then
    AArch64.SystemAccessTrap(EL3, 0x19);
  else
    ZCR_EL2 = X[t, 64];

MRS <Xt>, ZCR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.ZEN == '0' then
    UNDEFINED;
  elsif CPACR_EL1.ZEN == 'x0' then
    AArch64.SystemAccessTrap(EL1, 0x19);
  elsif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TZ == '1' then
    AArch64.SystemAccessTrap(EL2, 0x19);
  elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.ZEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x19);
  elsif HaveEL(EL3) && CPTR_EL3.ZEN == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x19);
    elsif HCR_EL2.E2H == '1' then
      X[t, 64] = ZCR_EL2;
    else
      X[t, 64] = ZCR_EL1;
  elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.ZEN == '0' then
      UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TZ == '1' then
      AArch64.SystemAccessTrap(EL2, 0x19);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.ZEN == 'x0' then
      AArch64.SystemAccessTrap(EL2, 0x19);
    elsif HaveEL(EL3) && CPTR_EL3.ZEN == '0' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x19);
      elsif HCR_EL2.E2H == '1' then
        X[t, 64] = ZCR_EL2;
      else
        X[t, 64] = ZCR_EL1;
    elsif PSTATE_EL == EL3 then
      if CPTR_EL3.ZEN == '0' then
        AArch64.SystemAccessTrap(EL3, 0x19);
else
    X[t, 64] = ZCR_EL1;

**MSR ZCR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>
D17.2.161 ZCR_EL3, SVE Control Register (EL3)

The ZCR_EL3 characteristics are:

**Purpose**

This register controls aspects of SVE visible at all Exception levels.

**Configurations**

This register is present only when FEAT_SVE is implemented. Otherwise, direct accesses to ZCR_EL3 are **undefined**.

This register has no effect when FEAT_SME is implemented and the PE is in Streaming SVE mode.

**Attributes**

ZCR_EL3 is a 64-bit register.

**Field descriptions**

| Bits [63:9] | Reserved, RES0. |
| Bits [8:4] | Reserved, RAZ/WI. |
| LEN, bits [3:0] | Requests an Effective Non-streaming SVE vector length at EL3 of \((LEN+1) \times 128\) bits.  
The Non-streaming SVE vector length can be any multiple of 128 bits, from 128 bits to 2048 bits inclusive. An implementation can support a subset of the architecturally permitted lengths. An implementation is required to support all lengths that are powers of two, from 128 bits up to its maximum implemented Non-streaming SVE vector length.  
When FEAT_SME is not implemented, or the PE is not in Streaming SVE mode, the Effective SVE vector length (VL) is equal to the Effective Non-streaming SVE vector length.  
When FEAT_SME is implemented and the PE is in Streaming SVE mode, VL is equal to the Effective Streaming SVE vector length. See SMCR_EL3.  
For all purposes other than returning the result of a direct read of ZCR_EL3, the PE selects the highest supported Non-streaming SVE vector length that is less than or equal to the requested length.  
An indirect read of ZCR_EL3.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.  
The reset behavior of this field is:  
• On a Warm reset, this field resets to an architecturally **UNKNOWN** value. |

**Accessing ZCR_EL3**

Accesses to this register use the following encodings in the System register encoding space:
**MRS <Xt>, ZCR_EL3**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    undefined;
elseif PSTATE_EL == EL1 then
    undefined;
elseif PSTATE_EL == EL2 then
    undefined;
elseif PSTATE_EL == EL3 then
    if CPTR_EL3.EZ == '0' then
        AArch64.SystemAccessTrap(EL3, 0x19);
    else
        X[t, 64] = ZCR_EL3;

**MSR ZCR_EL3, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    undefined;
elseif PSTATE_EL == EL1 then
    undefined;
elseif PSTATE_EL == EL2 then
    undefined;
elseif PSTATE_EL == EL3 then
    if CPTR_EL3.EZ == '0' then
        AArch64.SystemAccessTrap(EL3, 0x19);
    else
        ZCR_EL3 = X[t, 64];
D17.3 Debug registers

This section lists the Debug System registers in AArch64 state, in alphabetic order:

- The principal encoding space for debug registers is \( \text{op0} = \text{0b10}, \text{op1} = \{0, 3, 4\} \). *Instructions for accessing debug System registers on page D16-5519* summarizes the registers in this encoding space and lists them in order of their encodings.

- In addition, the following registers in the \( \text{op0} = \text{0b11} \) encoding space are classified as Debug registers:
  - DLR_EL0.
  - DSPSR_EL0.
  - MDCR_EL2.
  - MDCR_EL3.
  - SDER32_EL3.
D17.3.1  DBGAUTHSTATUS_EL1, Debug Authentication Status register

The DBGAUTHSTATUS_EL1 characteristics are:

**Purpose**

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

**Configurations**

AArch64 System register DBGAUTHSTATUS_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGAUTHSTATUS[31:0].

AArch64 System register DBGAUTHSTATUS_EL1 bits [31:0] are architecturally mapped to External register DBGAUTHSTATUS_EL1[31:0].

**Attributes**

DBGAUTHSTATUS_EL1 is a 64-bit register.

**Field descriptions**

Bits [63:28]

Reserved, RES0.

RTNID, bits [27:26]

Root non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RTID.

RTID, bits [25:24]

Root invasive debug.

0b00  Not implemented.

0b10  Implemented and disabled.

`ExternalRootInvasiveDebugEnabled () == FALSE.`

0b11  Implemented and enabled.

`ExternalRootInvasiveDebugEnabled () == TRUE.`

All other values are reserved.

If FEAT_RME is not implemented, the only permitted value is 00.

Bits [23:16]

Reserved, RES0.

RLNID, bits [15:14]

Realm non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RLID.

RLID, bits [13:12]

Realm invasive debug.

0b00  Not implemented.
0b10  Implemented and disabled.
      ExternalRealmInvasiveDebugEnabled() == FALSE.
0b11  Implemented and enabled.
      ExternalRealmInvasiveDebugEnabled() == TRUE.
All other values are reserved.
If FEAT_RME is not implemented, the only permitted value is 00.

Bits [11:8]
      Reserved, RES0.

SNID, bits [7:6]

When FEAT_Debugv8p4 is implemented:
      Secure non-invasive debug.
      This field has the same value as DBGAUTHSTATUS_EL1.SID.
Otherwise:
      Secure non-invasive debug.
0b00  Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 1.
0b10  Implemented and disabled. ExternalSecureNoninvasiveDebugEnabled() == FALSE.
0b11  Implemented and enabled. ExternalSecureNoninvasiveDebugEnabled() == TRUE.
All other values are reserved.

SID, bits [5:4]
      Secure invasive debug.
0b00  Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 1.
0b10  Implemented and disabled. ExternalSecureInvasiveDebugEnabled() == FALSE.
0b11  Implemented and enabled. ExternalSecureInvasiveDebugEnabled() == TRUE.
All other values are reserved.

NSNID, bits [3:2]

When FEAT_Debugv8p4 is implemented:
      Non-secure non-invasive debug.
0b00  Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 0.
0b11  Implemented and enabled. EL3 is implemented or the Effective value of SCR_EL3.NS is 1.
All other values are reserved.
Otherwise:
      Non-secure non-invasive debug.
0b00  Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 0.
0b10  Implemented and disabled. ExternalNoninvasiveDebugEnabled() == FALSE.
0b11  Implemented and enabled. ExternalNoninvasiveDebugEnabled() == TRUE.
All other values are reserved.

NSID, bits [1:0]
      Non-secure invasive debug.
0b00  Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 0.
0b10  Implemented and disabled. ExternalInvasiveDebugEnabled() == FALSE.
0b11  Implemented and enabled. ExternalInvasiveDebugEnabled() == TRUE.
All other values are reserved.
Accessing DBGAUTHSTATUS_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[ MRS \langle Xt\rangle, \text{DBGAUTHSTATUS\_EL1} \]

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1110</td>
<td>0b110</td>
</tr>
</tbody>
</table>
```

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED “EL3 trap priority when SDD == '1'” && MDTCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && EDSCR_EL3.FGTEn == '1' && HDFGTR_EL2.DBGAUTHSTATUS_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif MDTCR_EL2.<TDE,TDA> != '00' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && MDTCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  else
    X[t, 64] = DBGAUTHSTATUS_EL1;
  endif
else
  if PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED “EL3 trap priority when SDD == '1'” && MDTCR_EL3.TDA == '1' then
      UNDEFINED;
    elsif HaveEL(EL3) && MDTCR_EL3.TDA == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      endif
    else
      X[t, 64] = DBGAUTHSTATUS_EL1;
    endif
  elseif PSTATE.EL == EL3 then
    X[t, 64] = DBGAUTHSTATUS_EL1;
  endif
```
D17.3.2  DBGBCR<\textit{n}>_EL1, Debug Breakpoint Control Registers, \(n = 0 \text{ - } 15\)

The DBGBCR<\textit{n}>_EL1 characteristics are:

### Purpose

Holds control information for a breakpoint. Forms breakpoint \(n\) together with value register DBGBVR<\textit{n}>_EL1.

### Configurations

AArch64 System register DBGBCR<\textit{n}>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGBCR<\textit{n}>[31:0].

AArch64 System register DBGBCR<\textit{n}>_EL1 bits [31:0] are architecturally mapped to External register DBGBCR<\textit{n}>_EL1[31:0].

If breakpoint \(n\) is not implemented, accesses to this register are UNDEFINED.

### Attributes

DBGBCR<\textit{n}>_EL1 is a 64-bit register.

### Field descriptions

#### Bits [63:30]

Reserved, RES0.

#### SSCE, bit [29]

*When FEAT_RME is implemented:*

Security State Control Extended.

The fields that indicate when the breakpoint can be generated are: HMC, PMC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

#### Bits [28:24]

Reserved, RES0.

#### BT, bits [23:20]

Breakpoint Type. Possible values are:

- **0b0000**: Unlinked instruction address match. DBGBVR<\textit{n}>_EL1 is the address of an instruction.

- **0b0001**: As 0b0000, but linked to a Context matching breakpoint.
0b0010  Unlinked Context ID match. When FEAT_VHE is implemented, EL2 is using AArch64, and the Effective value of HCR_EL2.E2H is 1, if either the PE is executing at EL0 with HCR_EL2.TGE set to 1 or the PE is executing at EL2, then DBGBVR<n>_EL1.ContextID must match the CONTEXTIDR_EL2 value. Otherwise, DBGBVR<n>_EL1.ContextID must match the CONTEXTIDR_EL1 value.

0b0011  As 0b0010, with linking enabled.

0b0110  Unlinked CONTEXTIDR_EL1 match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1.

0b0111  As 0b0110, with linking enabled.

0b1000  Unlinked VMID match. DBGBVR<n>_EL1.VMID is a VMID compared against VTTBR_EL2.VMID.

0b1001  As 0b1000, with linking enabled.

0b1010  Unlinked VMID and Context ID match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.VMID is a VMID compared against VTTBR_EL2.VMID.

0b1011  As 0b1010, with linking enabled.

0b1100  Unlinked CONTEXTIDR_EL2 match. DBGBVR<n>_EL1.ContextID2 is a Context ID compared against CONTEXTIDR_EL2.

0b1101  As 0b1100, with linking enabled.

0b1110  Unlinked Full Context ID match. DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.ContextID2 is compared against CONTEXTIDR_EL2.

0b1111  As 0b1110, with linking enabled.

All other values are reserved. Constraints on breakpoint programming mean other values are reserved under some conditions.

The fields that indicate when the breakpoint can be generated are: HMC, PMC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see Execution conditions for which a breakpoint generates Breakpoint exceptions on page D2-4717.

For more information on the effect of programming the fields to a reserved value, see Reserved DBGBCR<n>_EL1.BT values on page D2-4722.

The reset behavior of this field is:

•  On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked breakpoint number. For Linked address matching breakpoints, this specifies the index of the Context-matching breakpoint linked to.

For all other breakpoint types this field is ignored and reads of the register return an UNKNOWN value.

This field is ignored when the value of DBGBCR<n>_EL1.E is 0.

The reset behavior of this field is:

•  On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Breakpoint debug event for breakpoint n is generated.

The fields that indicate when the breakpoint can be generated are: HMC, PMC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.
For more information on the operation of these fields, see \textit{Execution conditions for which a breakpoint generates Breakpoint exceptions} on page D2-4717.

For more information on the effect of programming the fields to a reserved set of values, see \textit{Reserved DBGBCR\textless n\textgreater _EL1.(SSC, HMC, PMC) values} on page D2-4722.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally \textbf{UNKNOWN} value.

\textbf{HMC, bit [13]}

Higher mode control. Determines the debug perspective for deciding when a Breakpoint debug event for breakpoint \textit{n} is generated.

The fields that indicate when the breakpoint can be generated are: HMC, PMC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see \textit{Execution conditions for which a breakpoint generates Breakpoint exceptions} on page D2-4717.

For more information, see DBGBCR\textless n\textgreater _EL1.SSC.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally \textbf{UNKNOWN} value.

\textbf{Bits [12:9]}

Reserved, \textbf{RES0}.

\textbf{BAS, bits [8:5]}

\textit{When AArch32 is supported:}

Byte address select. Defines which half-words an address-matching breakpoint matches, regardless of the instruction set and Execution state.

The permitted values depend on the breakpoint type.

For Address match breakpoints, the permitted values are:

<table>
<thead>
<tr>
<th>BAS</th>
<th>Match instruction at</th>
<th>Constraint for debuggers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0011</td>
<td>DBGBVR\textless n\textgreater _EL1</td>
<td>Use for T32 instructions</td>
</tr>
<tr>
<td>0b100</td>
<td>DBGBVR\textless n\textgreater _EL1 + 2</td>
<td>Use for T32 instructions</td>
</tr>
<tr>
<td>0b1111</td>
<td>DBGBVR\textless n\textgreater _EL1</td>
<td>Use for A64 and A32 instructions</td>
</tr>
</tbody>
</table>

All other values are reserved. For more information, see \textit{Reserved DBGBCR\textless n\textgreater _EL1.BAS values} on page D2-4723.

For more information on using the BAS field in address match breakpoints, see \textit{Using the BAS field in Address Match breakpoints} on page G2-9067.

For Context matching breakpoints, this field is \textbf{RES1} and ignored.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally \textbf{UNKNOWN} value.

\textit{Otherwise:}

Reserved, \textbf{RES1}.

\textbf{Bits [4:3]}

Reserved, \textbf{RES0}. 

For more information on the operation of these fields, see \textit{Execution conditions for which a breakpoint generates Breakpoint exceptions} on page D2-4717.
PMC, bits [2:1]

Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated.

The fields that indicate when the breakpoint can be generated are: HMC, PMC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see Execution conditions for which a breakpoint generates Breakpoint exceptions on page D2-4717.

For more information, see DBGBCR<n>_EL1.SSC.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Enable breakpoint DBGBVR<n>_EL1.

0b0 Breakpoint disabled.

0b1 Breakpoint enabled.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGBCR<n>_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, DBGBCR<m>_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0000</td>
<td>m[3:0]</td>
<td>0b101</td>
</tr>
</tbody>
</table>

integer m = UInt(CRm<3:0>);

if m >= NUM_BREAKPOINTS then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.DBGBCRn_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    elsif OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
      Halt(DebugHalt_SoftwareAccess);
    else
      X[t, 64] = DBGBCR_EL1[m];
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
      UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
elsif OSLR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt.SoftwareAccess);
else
    X[t, 64] = DBGCR_EL1[m];
elsif PSTATE_EL == EL3 then
    if OSLR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt.SoftwareAccess);
    else
        X[t, 64] = DBGCR_EL1[m];
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean
        IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1'" && EDSCR.TDA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
elsif OSLR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt.SoftwareAccess);
else
    DBGCR_EL1[m] = X[t, 64];
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean
        IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1'" && MDCR_EL3.TDA == '1' then
        UNDEFINED;
elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
elsif OSLR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt.SoftwareAccess);
else
    DBGCR_EL1[m] = X[t, 64];
elsif PSTATE_EL == EL3 then
    if OSLR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt.SoftwareAccess);
else
    DBGCR_EL1[m] = X[t, 64];

MSR DBGCR<m>_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0000</td>
<td>m[3:0]</td>
<td>0b101</td>
</tr>
</tbody>
</table>
D17.3.3 DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15

The DBGBVR<n>_EL1 characteristics are:

**Purpose**

Holds a virtual address, or a VMID and/or a context ID, for use in breakpoint matching. Forms breakpoint n together with control register DBGBCR<n>_EL1.

**Configurations**

AArch64 System register DBGBVR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGBVR<n>[31:0].

If the breakpoint is context-aware and EL2 is implemented then AArch64 System register DBGVR<n>_EL1 bits [63:32] are architecturally mapped to AArch32 System register DBGBXVR<n>. Otherwise, there is no System register access to DBGBVR<n>_EL1[63:32] from AArch32 state.

AArch64 System register DBGBVR<n>_EL1 bits [63:0] are architecturally mapped to External register DBGBVR<n>_EL1[63:0].

How this register is interpreted depends on the value of DBGBCR<n>_EL1.BT.

- When DBGBCR<n>_EL1.BT is 0b000x, this register holds a virtual address.
- When DBGBCR<n>_EL1.BT is 0b001x, or 0b110x, this register holds a Context ID.
- When DBGBCR<n>_EL1.BT is 0b100x, this register holds a VMID.
- When DBGBCR<n>_EL1.BT is 0b111x, this register holds two Context ID values.

For other values of DBGBCR<n>_EL1.BT, this register is RES0.

If breakpoint n is not implemented then accesses to this register are UNDEFINED.

**Attributes**

DBGBVR<n>_EL1 is a 64-bit register.

**Field descriptions**

*When DBGBCR<n>_EL1.BT == 0b000x:*

<table>
<thead>
<tr>
<th>63</th>
<th>53</th>
<th>49</th>
<th>48</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RESS[14:4]</strong></td>
<td><strong>VA[48:2]</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RESS[14:4], bits [63:53]**

Reserved, Sign extended. Software must set all bits in this field to the same value as the most significant bit of the VA field. If all bits in this field are not the same value as the most significant bit of the VA field, then all of the following apply:

- It is CONSTRAINED UNPREDICTABLE whether the PE ignores this field when comparing an address.
- If the breakpoint is not context-aware, it is IMPLEMENTATION DEFINED whether the value read back in each bit of this field is a copy of the most significant bit of the VA field or the value written.
VA[52:49], bits [52:49]

When FEAT_LVA is implemented:

VA[52:49]
Extension to VA[48:2]. For more information, see VA[48:2].
The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RESS[3:0]
Extension to RESS[14:4]. For more information, see RESS[14:4].

VA[48:2], bits [48:2]

Bits[48:2] of the address value for comparison.
When FEAT_LVA is implemented, VA[52:49] forms the upper part of the address value. Otherwise, bits [52:49] are part of the RESS field.
The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]
Reserved, RES0.

When DBGBCR<n>_EL1.BT == 0b001x:

Bits [63:32]
Reserved, RES0.

ContextID, bits [31:0]
Context ID value for comparison.
The value is compared against CONTEXTIDR_EL2 when (FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented), HCR_EL2.E2H is 1, and either:
- The PE is executing at EL2.
- HCR_EL2.TGE is 1, the PE is executing at EL0, and EL2 is enabled in the current Security state.

Otherwise, the value is compared against CONTEXTIDR_EL1.
The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.
**When DBGBCR<\text{n}>_EL1.BT == 0b011x:**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
<th>Reset Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:32]</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>[31:0]</td>
<td>ContextID</td>
<td>Context ID value for comparison against CONTEXTIDR_EL1. On a Cold reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
</tbody>
</table>

**When DBGBCR<\text{n}>_EL1.BT == 0b100x and EL2 is implemented:**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
<th>Reset Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:48]</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>[47:40]</td>
<td>VMID[15:8]</td>
<td>Extension to VMID[7:0]. For more information, see DBGBVR&lt;\text{n}&gt;_EL1.VMID[7:0]. On a Cold reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>[39:32]</td>
<td>VMID[7:0]</td>
<td>VMID value for comparison. The VMID is 8 bits when any of the following are true: EL2 is using AArch32, VTCR_EL2.VS is 0, FEAT_VMID16 is not implemented. On a Cold reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>[31:0]</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
</tbody>
</table>
When DBGBCR<\textgreater n\textless>_EL1.BT == 0b101x and EL2 is implemented:

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:48</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>47:40</td>
<td>VMID[15:8]</td>
</tr>
<tr>
<td>39:32</td>
<td>VMID[7:0]</td>
</tr>
</tbody>
</table>

Bits [63:48]

Reserved, RES0.

VMID[15:8], bits [47:40]

When FEAT_VMID16 is implemented, VTCR_EL2.VS == 1 and EL2 is using AArch64:

Extension to VMID[7:0]. For more information, see DBGVR<\textgreater n\textless>_EL1.VMID[7:0].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID[7:0], bits [39:32]

VMID value for comparison.

The VMID is 8 bits when any of the following are true:

- EL2 is using AArch32.
- VTCR_EL2.VS is 0.
- FEAT_VMID16 is not implemented.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<\textgreater n\textless>_EL1.BT == 0b110x, EL2 is implemented and (FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented):

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>ContextID2</td>
</tr>
<tr>
<td>31:0</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

ContextID2, bits [63:32]

Context ID value for comparison against CONTEXTIDR_EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]

Reserved, RES0.
When DBGBCR\textsubscript{\textit{n}}\_EL1.BT == 0b111\textit{x}, EL2 is implemented and (FEAT\_VHE is implemented or FEAT\_Debugv8p2 is implemented):

**ContextID2**, bits [63:32]
Context ID value for comparison against CONTEXTIDR\_EL2.
The reset behavior of this field is:
* On a Cold reset, this field resets to an architecturally UNKNOWN value.

**ContextID**, bits [31:0]
Context ID value for comparison against CONTEXTIDR\_EL1.
The reset behavior of this field is:
* On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Accessing DBGBVR\textsubscript{\textit{n}}\_EL1**
Accesses to this register use the following encodings in the System register encoding space:

\textbf{MRS <\textit{Xt}>, DBGBVR\textsubscript{\textit{m}}\_EL1}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0000</td>
<td>m[3:0]</td>
<td>0b100</td>
</tr>
</tbody>
</table>

integer \(m = \text{UInt}(\text{CRm}\langle3:0\rangle)\);

if \(m \geq \text{NUM\_BREAKPOINTS}\) then
  UNDEFINED;
elsif PSTATE\_EL == EL0 then
  UNDEFINED;
elsif PSTATE\_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR\_SDD == '1' && boolean IMPLEMENTATION\_DEFINED "EL3 trap priority when SDD == '1'" && MDCR\_EL3\_TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR\_EL3\_FGTEn == '1') && HDFGRTR\_EL2\_DBGBVRn\_EL1 == '1' then
    AArch64\_SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR\_EL2\_TDE,TDA\_ != '00' then
    AArch64\_SystemAccessTrap(EL2, 0x17);
  elsif HaveEL(EL3) && MDCR\_EL3\_TDA == '1' then
    if Halted() && EDSCR\_SDD == '1' then
      UNDEFINED;
    else
      HALT(DebugHalt\_SoftwareAccess);
    elseif OSLSR\_EL1\_OSLK == '0' && HaltingAllowed() && EDSCR\_TDA == '1' then
      HALT(DebugHalt\_SoftwareAccess);
    else
      X[t, 64] = DBGBVR\_EL1[m];
    elsif PSTATE\_EL == EL2 then
      if Halted() && HaveEL(EL3) && EDSCR\_SDD == '1' && boolean IMPLEMENTATION\_DEFINED "EL3 trap priority when SDD == '1'" && MDCR\_EL3\_TDA == '1' then
        UNDEFINED;
      elsif HaveEL(EL3) && MDCR\_EL3\_TDA == '1' then
        AArch64\_SystemAccessTrap(EL3, 0x18);
      elsif OSLSR\_EL1\_OSLK == '0' && HaltingAllowed() && EDSCR\_TDA == '1' then
        HALT(DebugHalt\_SoftwareAccess);
      else
        X[t, 64] = DBGBVR\_EL1[m];
      endif
    endif
  endif
else
  AArch64\_SystemAccessTrap(EL3, 0x18);
endif
if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
elsif OSLR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
else
    X[t, 64] = DBGVR_EL1[m];
elsif PSTATE.EL == EL3 then
    if OSLR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X[t, 64] = DBGVR_EL1[m];
    end if;
end if;

integer m = UInt(CRm<3:0>);
if m >= NUM_BREAKPOINTS then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elseif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.DBGBVRn_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        elseif OSLR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            DBGBVR_EL1[m] = X[t, 64];
        end if;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end if;
else
    OSLR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    end if;
else
    DBGBVR_EL1[m] = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elseif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        elseif OSLR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
            Halt(DebugHalt_SoftwareAccess);
        else
            DBGBVR_EL1[m] = X[t, 64];
        end if;
    end if;
else
    DBGBVR_EL1[m] = X[t, 64];
elsif PSTATE.EL == EL3 then
    if OSLR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBVR_EL1[m] = X[t, 64];
end if;

---

MSR DBGBVR<m>_EL1, <Xt>
D17.3.4 DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register

The DBGCLAIMCLR_EL1 characteristics are:

**Purpose**

Used by software to read the values of the CLAIM tag bits, and to clear CLAIM tag bits to 0. The architecture does not define any functionality for the CLAIM tag bits.

--- Note

CLAIM tags are typically used for communication between the debugger and target software.

---

Used in conjunction with the DBGCLAIMSET_EL1 register.

**Configurations**

AArch64 System register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGCLAIMCLR[31:0].

AArch64 System register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to External register DBGCLAIMCLR_EL1[31:0].

An implementation must include eight CLAIM tag bits.

**Attributes**

DBGCLAIMCLR_EL1 is a 64-bit register.

**Field descriptions**

```
| 63 | 32 | RES0 |
| 31 | 8  | RAZ/WI |
| 7  | 0  | CLAIM |
```

**Bits [63:32]**

Reserved, RES0.

**Bits [31:8]**

Reserved, RAZ/WI.

**CLAIM, bits [7:0]**

Read or clear CLAIM tag bits. Reading this field returns the current value of the CLAIM tag bits. Writing a 1 to one of these bits clears the corresponding CLAIM tag bit to 0. This is an indirect write to the CLAIM tag bits. A single write operation can clear multiple CLAIM tag bits to 0.

Writing 0 to one of these bits has no effect.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.

**Accessing DBGCLAIMCLR_EL1**

Accesses to this register use the following encodings in the System register encoding space:
### MRS <Xt>, DBGCLAIMCLR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1001</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elifs PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elifs EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.DBGCLAIM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elifs EL2Enabled() && MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elifs HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      DBGCLAIMCLR_EL1 = X[t, 64];
elifs PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elifs HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  if PSTATE.EL == EL3 then
    X[t, 64] = DBGCLAIMCLR_EL3;
elifs PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elifs EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.DBGCLAIM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elifs EL2Enabled() && MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elifs HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      DBGCLAIMCLR_EL1 = X[t, 64];
elifs PSTATE.EL == EL2 then


### MSR DBGCLAIMCLR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1001</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elifs PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elifs EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.DBGCLAIM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elifs EL2Enabled() && MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elifs HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      DBGCLAIMCLR_EL1 = X[t, 64];
elifs PSTATE.EL == EL2 then


---

**AArch64 System Register Descriptions**

D17.3 Debug registers

---

ARM DDI 0487I.a  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.

ID081822  Non-Confidential

---

D17-6409
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
  UNDEFINED;
elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
elsif PSTATE.EL == EL3 then
  DBGCLAIMCLR_EL1 = X[t, 64];
else
  DBGCLAIMCLR_EL1 = X[t, 64];
D17.3.5 DBGCLAIMSET_EL1, Debug CLAIM Tag Set register

The DBGCLAIMSET_EL1 characteristics are:

**Purpose**

Used by software to set the CLAIM tag bits to 1.
The architecture does not define any functionality for the CLAIM tag bits.

--- **Note** ---

CLAIM tags are typically used for communication between the debugger and target software.

---

Used in conjunction with the DBGCLAIMCLR_EL1 register.

**Configurations**

AArch64 System register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGCLAIMSET[31:0].
AArch64 System register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to External register DBGCLAIMSET_EL1[31:0].
An implementation must include eight CLAIM tag bits.

**Attributes**

DBGCLAIMSET_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>31:8</td>
<td>Reserved, RAZ/WI.</td>
</tr>
<tr>
<td>7:0</td>
<td>CLAIM</td>
</tr>
</tbody>
</table>

Bits [63:32]

Reserved, RES0.

Bits [31:8]

Reserved, RAZ/WI.

CLAIM, bits [7:0]

Set CLAIM tag bits.
This field is RAO.
Writing a 1 to one of these bits sets the corresponding CLAIM tag bit to 1. This is an indirect write to the CLAIM tag bits. A single write operation can set multiple CLAIM tag bits to 1.
Writing 0 to one of these bits has no effect.

**Accessing DBGCLAIMSET_EL1**

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE.EL == EL0 then
UNDEFINED;
elsesif PSTATE.EL == EL1 then
if Halted() \& \& HaveEL(EL3) \& \& EDSCR.SDD == '1' \& \& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \& \& MDCR_EL3.TDA == '1' then
UNDEFINED;
elsesif EL2Enabled() \& \& !HaveEL(EL3) \& \& SCR_EL3.FGTEn == '1' \& \& HDFGRTR_EL2.DBGCLAIM == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsesif EL2Enabled() \& \& MDCR_EL2.<TDE,TDA> != '00' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsesif HaveEL(EL3) \& \& MDCR_EL3.TDA == '1' then
if Halted() \& \& EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = DBGCLAIMSET_EL1;
else
if Halted() \& \& HaveEL(EL3) \& \& EDSCR.SDD == '1' \& \& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \& \& MDCR_EL3.TDA == '1' then
UNDEFINED;
elsesif HaveEL(EL3) \& \& MDCR_EL3.TDA == '1' then
if Halted() \& \& EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = DBGCLAIMSET_EL1;
elesif PSTATE.EL == EL3 then
X[t, 64] = DBGCLAIMSET_EL1;
if PSTATE.EL == EL0 then
UNDEFINED;
elsesif PSTATE.EL == EL1 then
if Halted() \& \& HaveEL(EL3) \& \& EDSCR.SDD == '1' \& \& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \& \& MDCR_EL3.TDA == '1' then
UNDEFINED;
elsesif EL2Enabled() \& \& !HaveEL(EL3) \& \& SCR_EL3.FGTEn == '1' \& \& HDFGRTR_EL2.DBGCLAIM == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsesif EL2Enabled() \& \& MDCR_EL2.<TDE,TDA> != '00' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsesif HaveEL(EL3) \& \& MDCR_EL3.TDA == '1' then
if Halted() \& \& EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
DBGCLAIMSET_EL1 = X[t, 64];
elesif PSTATE.EL == EL2 then

---
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        if PSTATE.EL == EL3 then
            DBGCLAIMSET_EL1 = X[t, 64];
        else
            DBGCLAIMSET_EL1 = X[t, 64];
        end
    end
    elif PSTATE.EL == EL3 then
        DBGCLAIMSET_EL1 = X[t, 64];
    else
        DBGCLAIMSET_EL1 = X[t, 64];
    end
end
D17.3.6  **DBGDTR_EL0, Debug Data Transfer Register, half-duplex**

The DBGDTR_EL0 characteristics are:

**Purpose**

Transfers 64 bits of data between the PE and an external debugger. Can transfer both ways using only a single register.

**Configurations**

- AArch64 System register DBGDTR_EL0 bits [63:32] are architecturally mapped to AArch32 System register DBGDTRRXint[31:0] when written.
- AArch64 System register DBGDTR_EL0 bits [63:32] are architecturally mapped to External register DBGDTRRX_EL0[31:0] when written.
- AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0] when written.
- AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to AArch32 System register DBGDTRTXint[31:0] when written.
- AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to External register DBGDTRTX_EL0[31:0] when written.
- AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to AArch64 System register DBGDTRTX_EL0[31:0] when written.
- AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to External register DBGDTRRX_EL0[31:0] when read.
- AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0] when read.
- AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to External register DBGDTRRX_EL0[31:0] when read.
- AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0] when read.
- AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to External register DBGDTRRX_EL0[31:0] when read.

**Attributes**

DBGDTR_EL0 is a 64-bit register.

**Field descriptions**

![HighWord, bits [63:32]](image)

Writes to this register set DTRRX to the value in this field and do not change RXfull.

Reads of this register:

- If RXfull is set to 1, return the last value written to DTRTX.
- If RXfull is set to 0, return an UNKNOWN value.

After the read, RXfull is cleared to 0.
**LowWord, bits [31:0]**

Writes to this register set DTRTX to the value in this field and set TXfull to 1.

Reads of this register:
- If RXfull is set to 1, return the last value written to DTRRX.
- If RXfull is set to 0, return an UNKNOWN value.

After the read, RXfull is cleared to 0.

### Accessing DBGDTR_EL0

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, DBGDTR_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b011</td>
<td>0b0000</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```c
if Halted() then
    X[t, 64] = DBGDTR_EL0;
elsif PSTATE_EL == EL0 then
    if MDSCR_EL1.TDCC == '1' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        end if
    elsif EL2Enabled() && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = DBGDTR_EL0;
    end if
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = DBGDTR_EL0;
    end if
elsif PSTATE_EL == EL2 then
    if HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = DBGDTR_EL0;
    end if
elsif PSTATE_EL == EL3 then
    X[t, 64] = DBGDTR_EL0;
```
if Halted() then
  DBGDTR_EL0 = X[t, 64];
elsif PSTATE_EL == EL0 then
  if MSR_EL1.TDCC == '1' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    endif
  else
    if EL2Enabled() && MDCR_EL2.TDCC == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    endif
  endif
else
  if EL2Enabled() && HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if EL2Enabled() && MDCR_EL2.TDCC == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    endif
  endif
else
  if HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    if HaveEL(EL3) && MDCR_EL3.TDA == '1' then
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      DBGDTR_EL0 = X[t, 64];
    endif
  endif
else
  if PSTATE_EL == EL2 then
    if HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      if HaveEL(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
      else
        DBGDTR_EL0 = X[t, 64];
      endif
    endif
  else
    if PSTATE_EL == EL3 then
      DBGDTR_EL0 = X[t, 64];
    else
      DBGDTR_EL0 = X[t, 64];
    endif
  endif
endif

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b011</td>
<td>0b0000</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>
D17.3.7 DBGDTRRX_EL0, Debug Data Transfer Register, Receive

The DBGDTRRX_EL0 characteristics are:

**Purpose**

Transfers data from an external debugger to the PE. For example, it is used by a debugger transferring commands and data to a debug target. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communications Channel.

**Configurations**

AArch64 System register DBGDTRRX_EL0 bits [31:0] are architecturally mapped to AArch32 System register DBGDTRRXint[31:0].

AArch64 System register DBGDTRRX_EL0 bits [31:0] are architecturally mapped to External register DBGDTRRX_EL0[31:0].

**Attributes**

DBGDTRRX_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
<th>RES0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>Update DTRRX</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**Bits [31:0]**

Update DTRRX.

Reads of this register:
- If RXfull is set to 1, return the last value written to DTRRX.
- If RXfull is set to 0, return an UNKNOWN value.

After the read, RXfull is cleared to 0.

For the full behavior of the Debug Communications Channel, see Chapter H4 The Debug Communication Channel and Instruction Transfer Register.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Accessing DBGDTRRX_EL0**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS} <Xt>, \text{DBGDTRRX}_\text{EL0}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b011</td>
<td>0b0000</td>
<td>0b0101</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if Halted() then
X[t, 64] = DBGDTRRX_EL0;
elsif PSTATE_EL == EL0 then
    if MDSCR_EL1.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.TDA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = DBGDTRRX_EL0;
    elsif PSTATE_EL == EL1 then
        if EL2Enabled() && MDSCR_EL2.TDCC == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && MDCR_EL2.<TDE,TDA> != '00' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = DBGDTRRX_EL0;
        elsif PSTATE_EL == EL2 then
            if HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
                AArch64.SystemAccessTrap(EL3, 0x18);
            elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
                AArch64.SystemAccessTrap(EL3, 0x18);
            else
                X[t, 64] = DBGDTRRX_EL0;
        elsif PSTATE_EL == EL3 then
            X[t, 64] = DBGDTRRX_EL0;
**D17.3.8 DBGDTRTX_EL0, Debug Data Transfer Register, Transmit**

The DBGDTRTX_EL0 characteristics are:

**Purpose**

Transfers data from the PE to an external debugger. For example, it is used by a debug target to transfer data to the debugger. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communication Channel.

**Configurations**

AArch64 System register DBGDTRTX_EL0 bits [31:0] are architecturally mapped to AArch32 System register DBGDTRTXint[31:0].

AArch64 System register DBGDTRTX_EL0 bits [31:0] are architecturally mapped to External register DBGDTRTX_EL0[31:0].

**Attributes**

DBGDTRTX_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:32]</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>[31:0]</td>
<td>Return DTRTX</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**Bits [31:0]**

Return DTRTX.

Writes to this register:
- If TXfull is set to 1, set DTRRX and DTRTX to UNKNOWN.
- If TXfull is set to 0, update the value in DTRTX.

After the write, TXfull is set to 1.

For the full behavior of the Debug Communications Channel, see Chapter H4 The Debug Communication Channel and Instruction Transfer Register.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Accessing DBGDTRTX_EL0**

Accesses to this register use the following encodings in the System register encoding space:

**MSR DBGDTRTX_EL0, <Xt>**

<table>
<thead>
<tr>
<th></th>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b011</td>
<td>0b0000</td>
<td>0b0101</td>
<td>0b000</td>
<td></td>
</tr>
</tbody>
</table>

if Halted() then
DBGDTRTX_EL0 = X[t, 64];
elsif PSTATE_EL == EL0 then
    if MDSCR_EL1.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.SystemAccessTrap(EL1, 0x18);
    endif
elseif EL2Enabled() && MDCR_EL2.TDCC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elseif EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00') then
    AArch64.SystemAccessTrap(EL2, 0x18);
elseif HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
elseif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    DBGDTRTX_EL0 = X[t, 64];
elseif PSTATE_EL == EL1 then
    if EL2Enabled() && MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    endif
elseif HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
elseif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    DBGDTRTX_EL0 = X[t, 64];
elseif PSTATE_EL == EL2 then
    if HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elseif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        DBGDTRTX_EL0 = X[t, 64];
elseif PSTATE_EL == EL3 then
    DBGDTRTX_EL0 = X[t, 64];
D17.3.9 DBGPRCR_EL1, Debug Power Control Register

The DBGPRCR_EL1 characteristics are:

**Purpose**

Controls behavior of the PE on powerdown request.

**Configurations**

AArch64 System register DBGPRCR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGPRCR[31:0].

Bit [0] of this register is mapped to EDPRCR.CORENPDRQ, bit [0] of the external view of this register.

The other bits in these registers are not mapped to each other.

**Attributes**

DBGPRCR_EL1 is a 64-bit register.

**Field descriptions**

Bits [63:1]

Reserved, RES0.

CORENPDRQ, bit [0]

*When FEAT_DoPD is implemented:*

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

0b0 If the system responds to a powerdown request, it powers down Core power domain.

0b1 If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

In an implementation that includes the recommended external debug interface, this bit drives the DBGNOPWRDWN signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to its Cold reset value on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see Core power domain power states on page H6-10336.

**Note**

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:

- On a Cold reset, if the powerup request is implemented and the powerup request has been asserted, this field is set to an IMPLEMENTATION DEFINED choice of 0 or 1. If the powerup request is not asserted, this field is set to 0.
Otherwise:

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

- **0b0** If the system responds to a powerdown request, it powers down Core power domain.
- **0b1** If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

In an implementation that includes the recommended external debug interface, this bit drives the DBGNOPWRDWN signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the value of EDRCCR.COREPURQ on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see Core power domain power states on page H6-10336.

**Note**

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:

- On a Cold reset, this field resets to the value in EDRCCR.COREPURQ.

### Accessing DBGPRCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, DBGPRCR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0100</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsf PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDOSA == '1' then
        UNDEFINED;
elsf ELE2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.DBGPRCR_EL1 == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
elsf ELE2Enabled() && MDCR_EL2.<TDE,TDOSA> != '00' then
            AArch64.SystemAccessTrap(EL2, 0x18);
elsf HaveEL(EL3) && MDCR_EL3.TDOSA == '1' then
                if Halted() && EDSCR.SDD == '1' then
                    UNDEFINED;
elsf PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDOSA == '1' then
        UNDEFINED;
elsf HaveEL(EL3) && MDCR_EL3.TDOSA == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
elsf AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = DBGPRCR_EL1;
elesf PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDOSA == '1' then
        UNDEFINED;
elsf HaveEL(EL3) && MDCR_EL3.TDOSA == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
elsf AArch64.SystemAccessTrap(EL3, 0x18);
else
  \( X[t, 64] = \text{DBGPRCR_EL1}; \)
elsif PSTATE.EL == EL3 then
  \( X[t, 64] = \text{DBGPRCR_EL1}; \)

\textbf{MSR DBGPRCR\_EL1, \(<Xt>\)}

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0100</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDOSA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.DBGPRCR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.<TDE,TDOSA> != '00' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && MDCR_EL3.TDOSA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    DBGPRCR_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEl(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDOSA == '1' then
    UNDEFINED;
  elsif HaveEl(EL3) && MDCR_EL3.TDOSA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    DBGPRCR_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL3 then
  DBGPRCR_EL1 = X[t, 64];
D17.3.10  DBGVCR32_EL2, Debug Vector Catch Register

The DBGVCR32_EL2 characteristics are:

**Purpose**

Allows access to the AArch32 register DBGVCR from AArch64 state only. Its value has no effect on execution in AArch64 state.

**Configurations**

AArch64 System register DBGVCR32_EL2 bits [31:0] are architecturally mapped to AArch32 System register DBGVCR[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGVCR32_EL2 are UNDEFINED.

If EL2 is not implemented but EL3 is implemented, and EL1 is capable of using AArch32, then this register is not RES0.

**Attributes**

DBGVCR32_EL2 is a 64-bit register.

**Field descriptions**

*When EL3 is implemented:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Reserved, RES0</td>
<td></td>
<td>0x1c</td>
</tr>
<tr>
<td>31</td>
<td>NSF, bit [31]</td>
<td>RES0</td>
<td>0x18</td>
</tr>
<tr>
<td>30</td>
<td>NSI, bit [30]</td>
<td>RES0</td>
<td>0x10</td>
</tr>
<tr>
<td>29</td>
<td>Reserved, RES0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>NSD, bit [28]</td>
<td>RES0</td>
<td>0x0c</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**NSF, bit [31]**

FIQ vector catch enable in Non-secure state.

The exception vector offset is 0x1c.

The reset behavior of this field is:

* • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**NSI, bit [30]**

IRQ vector catch enable in Non-secure state.

The exception vector offset is 0x18.

The reset behavior of this field is:

* • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [29]**

Reserved, RES0.

**NSD, bit [28]**

Data Abort exception vector catch enable in Non-secure state.

The exception vector offset is 0x10.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**NSP, bit [27]**
Prefetch Abort vector catch enable in Non-secure state.
The exception vector offset is 0x0C.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**NSS, bit [26]**
Supervisor Call (SVC) vector catch enable in Non-secure state.
The exception vector offset is 0x08.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**NSU, bit [25]**
Undefined Instruction vector catch enable in Non-secure state.
The exception vector offset is 0x04.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [24:8]**
Reserved, RES0.

**SF, bit [7]**
FIQ vector catch enable in Secure state.
The exception vector offset is 0x1C.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**SI, bit [6]**
IRQ vector catch enable in Secure state.
The exception vector offset is 0x18.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [5]**
Reserved, RES0.

**SD, bit [4]**
Data Abort exception vector catch enable in Secure state.
The exception vector offset is 0x10.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**SP, bit [3]**
Prefetch Abort vector catch enable in Secure state.
The exception vector offset is 0x0C.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
SS, bit [2]
Supervisor Call (SVC) vector catch enable in Secure state.
The exception vector offset is 0x08.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

SU, bit [1]
Undefined Instruction vector catch enable in Secure state.
The exception vector offset is 0x04.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]
Reserved, RES0.

When EL3 is not implemented:

Bits [63:8]
Reserved, RES0.

F, bit [7]
FIQ vector catch enable.
The exception vector offset is 0x1C.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [6]
IRQ vector catch enable.
The exception vector offset is 0x18.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]
Reserved, RES0.

D, bit [4]
Data Abort exception vector catch enable.
The exception vector offset is 0x10.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P, bit [3]
Prefetch Abort vector catch enable.
The exception vector offset 0x0C.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S, bit [2]
Supervisor Call (SVC) vector catch enable.
The exception vector offset is 0x08.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [1]
Undefined Instruction vector catch enable.
The exception vector offset is 0x04.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]
Reserved, RES0.

**Accessing DBGVCR32_EL2**
Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, DBGVCR32_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b100</td>
<td>0b0000</td>
<td>0b0111</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif PSTATE.EL == EL3 then
        X[t, 64] = DBGVCR32_EL2;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = DBGVCR32_EL2;
```
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  DBGVCR32_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
  DBGVCR32_EL2 = X[t, 64];
D17.3.11  DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 15

The DBGWCR<n>_EL1 characteristics are:

Purpose

Holds control information for a watchpoint. Forms watchpoint n together with value register DBGWVR<n>_EL1.

Configurations

AArch64 System register DBGWCR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGWCR<n>[31:0].

AArch64 System register DBGWCR<n>_EL1 bits [31:0] are architecturally mapped to External register DBGWCR<n>_EL1[31:0].

If watchpoint n is not implemented then accesses to this register are UNDEFINED.

Attributes

DBGWCR<n>_EL1 is a 64-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>29</td>
<td>SSCE, bit [29]</td>
</tr>
</tbody>
</table>
| 28  | When FEAT_RME is implemented: Security State Control Extended.  
     Security State Control Extended.  
     The fields that indicate when the watchpoint can be generated are: HMC, PAC, SSC, and SSCE.  
     These fields must be considered in combination, and the values that are permitted for these fields are constrained.  
     The reset behavior of this field is:  
     * On a Cold reset, this field resets to an architecturally UNKNOWN value. |
| 24  | MASK, bits [28:24] |
| 28  | Address mask. Only objects up to 2GB can be watched using a single mask.  
     0b00000  No mask.  
     0b00001  Reserved.  
     0b00010  Reserved.  
     If programmed with a reserved value, a watchpoint must behave as if either:  
     * MASK has been programmed with a defined value, which might be 0 (no mask), other than for a direct read of DBGWCRn_EL1.  
     * The watchpoint is disabled. |
Software must not rely on this property because the behavior of reserved values might change in a future revision of the architecture.

Other values mask the corresponding number of address bits, from 0b00011 masking 3 address bits (0x00000007 mask for address) to 0b11111 masking 31 address bits (0x7FFFFFFF mask for address).

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Bits [23:21]**

Reserved, RES0.

**WT, bit [20]**

Watchpoint type. Possible values are:
- 0b0 Unlinked data address match.
- 0b1 Linked data address match.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**LBN, bits [19:16]**

Linked breakpoint number. For Linked data address watchpoints, this specifies the index of the Context-matching breakpoint linked to.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**SSC, bits [15:14]**

Security state control. Determines the Security states under which a Watchpoint debug event for watchpoint n is generated.

The fields that indicate when the watchpoint can be generated are: HMC, PAC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see Execution conditions for which a watchpoint generates Watchpoint exceptions on page D2-4728.

For more information on the effect of programming the fields to a reserved value, see Reserved DBGWCR<\a>_EL1.(SSC, HMC, PAC) values on page D2-4737.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**HMC, bit [13]**

Higher mode control. Determines the debug perspective for deciding when a Watchpoint debug event for watchpoint n is generated.

The fields that indicate when the watchpoint can be generated are: HMC, PAC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see Execution conditions for which a watchpoint generates Watchpoint exceptions on page D2-4728.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.
BAS, bits [12:5]

Byte address select. Each bit of this field selects whether a byte from within the word or double-word addressed by DBGWVR<n>_EL1 is being watched.

<table>
<thead>
<tr>
<th>BAS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxx1</td>
<td>Match byte at DBGWVR&lt;n&gt;_EL1</td>
</tr>
<tr>
<td>xxxx1x</td>
<td>Match byte at DBGWVR&lt;n&gt;_EL1 + 1</td>
</tr>
<tr>
<td>xxxx1xx</td>
<td>Match byte at DBGWVR&lt;n&gt;_EL1 + 2</td>
</tr>
<tr>
<td>xxxx1xxx</td>
<td>Match byte at DBGWVR&lt;n&gt;_EL1 + 3</td>
</tr>
</tbody>
</table>

In cases where DBGWVR<n>_EL1 addresses a double-word:

<table>
<thead>
<tr>
<th>BAS</th>
<th>Description, if DBGWVR&lt;n&gt;_EL1[2] == 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx1xxxx</td>
<td>Match byte at DBGWVR&lt;n&gt;_EL1 + 4</td>
</tr>
<tr>
<td>xx1xxxxxx</td>
<td>Match byte at DBGWVR&lt;n&gt;_EL1 + 5</td>
</tr>
<tr>
<td>x1xxxxxxx</td>
<td>Match byte at DBGWVR&lt;n&gt;_EL1 + 6</td>
</tr>
<tr>
<td>1xxxxxxxx</td>
<td>Match byte at DBGWVR&lt;n&gt;_EL1 + 7</td>
</tr>
</tbody>
</table>


The valid values for BAS are non-zero binary numbers all of whose set bits are contiguous. All other values are reserved and must not be used by software. See Reserved DBGWCR<n>_EL1.BAS values on page D2-4738.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LSC, bits [4:3]

Load/store control. This field enables watchpoint matching on the type of access being made. Possible values of this field are:
- 0b01 Match instructions that load from a watchpointed address.
- 0b10 Match instructions that store to a watchpointed address.
- 0b11 Match instructions that load from or store to a watchpointed address.

All other values are reserved, but must behave as if the watchpoint is disabled. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PAC, bits [2:1]

Privilege of access control. Determines the Exception level or levels at which a Watchpoint debug event for watchpoint n is generated.

The fields that indicate when the watchpoint can be generated are: HMC, PAC, SSC, and SSCE. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see Execution conditions for which a watchpoint generates Watchpoint exceptions on page D2-4728.
The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally **UNKNOWN** value.

**E, bit [0]**

Enable watchpoint n. Possible values are:

- 0b0  Watchpoint disabled.
- 0b1  Watchpoint enabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally **UNKNOWN** value.

**Accessing DBGWCR<n>_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, DBGWCR<m>_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0000</td>
<td>m[3:0]</td>
<td>0b11</td>
</tr>
</tbody>
</table>

```
integer m = UInt(CRm<3:0>);

if m > Num_Watchpoints then
  UNDEFINED;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR_SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGR_EL2DBGWCRn_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR_SDD == '1' then
      UNDEFINED;
    elsif OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
      Halt(DebugHalt_SoftwareAccess);
    else
      X[t, 64] = DBGWCR_EL1[m];
  elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR_SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
      UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR_SDD == '1' then
      UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  elsif OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
  else
    X[t, 64] = DBGWCR_EL1[m];
  elsif PSTATE_EL == EL3 then
    if OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
      Halt(DebugHalt_SoftwareAccess);
```
else
  \[X[t, 64] = \text{DBGWCR\_EL1}[m]\];

**MSR DBGWCR\_m\_EL1, \langle Xt\rangle**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0000</td>
<td>m[3:0]</td>
<td>0b111</td>
</tr>
</tbody>
</table>

integer \(m = \text{UInt}(\text{CRm}<3:0>)\);

if \(m \geq \text{NUM\_WATCHPOINTS}\) then
  UNDEFINED;
elseif \(\text{PSTATE\_EL} = \text{EL0}\) then
  UNDEFINED;
elseif \(\text{PSTATE\_EL} = \text{EL1}\) then
  if \(\text{Halted()} \&\& \text{HaveEL(EL3)} \&\& \text{EDSCR.SDD} = '1' \&\& \text{boolean IMPLEMENTATION\_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCR\_EL3.TDA == '1'}\) then
    UNDEFINED;
  elseif \(\text{EL2Enabled()} \&\& (\text{HaveEL(EL3)} || \text{SCR\_EL3.FGTEn} = '1') \&\& \text{HDGWTR\_EL2.DBGWCRn\_EL1} = '1'\) then
    \text{AArch64.SystemAccessTrap(EL2, 0x18)};
  elseif \(\text{EL2Enabled()} \&\& \text{MDCR\_EL2.<TDE,TDA>} != '00'\) then
    \text{AArch64.SystemAccessTrap(EL2, 0x18)};
  elseif \(\text{HaveEL(EL3)} \&\& \text{MDCR\_EL3.TDA} = '1'\) then
    if \(\text{Halted()} \&\& \text{EDSCR.SDD} = '1'\) then
      UNDEFINED;
    else
      \text{AArch64.SystemAccessTrap(EL3, 0x18)};
    endif
  elseif \(\text{OSLSR\_EL1.OSLK} = '0' \&\& \text{HaltingAllowed()} \&\& \text{EDSCR.TDA} = '1'\) then
    \text{Halt(DebugHalt\_SoftwareAccess)};
  else
    \[\text{DBGWCR\_EL1}[m] = X[t, 64]\];
  endif
elseif \(\text{PSTATE\_EL} = \text{EL2}\) then
  if \(\text{Halted()} \&\& \text{HaveEL(EL3)} \&\& \text{EDSCR.SDD} = '1' \&\& \text{boolean IMPLEMENTATION\_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCR\_EL3.TDA == '1'}\) then
    UNDEFINED;
  elseif \(\text{HaveEL(EL3)} \&\& \text{MDCR\_EL3.TDA} = '1'\) then
    if \(\text{Halted()} \&\& \text{EDSCR.SDD} = '1'\) then
      UNDEFINED;
    else
      \text{AArch64.SystemAccessTrap(EL3, 0x18)};
    endif
  elseif \(\text{OSLSR\_EL1.OSLK} = '0' \&\& \text{HaltingAllowed()} \&\& \text{EDSCR.TDA} = '1'\) then
    \text{Halt(DebugHalt\_SoftwareAccess)};
  else
    \[\text{DBGWCR\_EL1}[m] = X[t, 64]\];
  endif
elseif \(\text{PSTATE\_EL} = \text{EL3}\) then
  if \(\text{OSLSR\_EL1.OSLK} = '0' \&\& \text{HaltingAllowed()} \&\& \text{EDSCR.TDA} = '1'\) then
    \text{Halt(DebugHalt\_SoftwareAccess)};
  else
    \[\text{DBGWCR\_EL1}[m] = X[t, 64]\];
  endif
D17.3.12 DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 15

The DBGWVR<n>_EL1 characteristics are:

Purpose

Holds a data address value for use in watchpoint matching. Forms watchpoint n together with control register DBGWCR<n>_EL1.

Configurations

AArch64 System register DBGWVR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGWVR<n>[31:0].
AArch64 System register DBGWVR<n>_EL1 bits [63:0] are architecturally mapped to External register DBGWVR<n>_EL1[63:0].

If watchpoint n is not implemented then accesses to this register are UNDEFINED.

Attributes

DBGWVR<n>_EL1 is a 64-bit register.

Field descriptions

RESS[14:4], bits [63:53]

Reserved, Sign extended. Software must set all bits in this field to the same value as the most significant bit of the VA field. If all bits in this field are not the same value as the most significant bit of the VA field, then all of the following apply:

- It is CONSTRAINED UNPREDICTABLE whether the PE ignores this field when comparing an address.
- It is IMPLEMENTATION DEFINED whether the value read back in each bit of this field is a copy of the most significant bit of the VA field or the value written.

VA[52:49], bits [52:49]

When FEAT_LVA is implemented:

VA[52:49]

Extension to VA[48:2]. For more information, see VA[48:2].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RESS[3:0]

Extension to RESS[14:4]. For more information, see RESS[14:4].

VA[48:2], bits [48:2]

Bits[48:2] of the address value for comparison.

When FEAT_LVA is implemented, VA[52:49] forms the upper part of the address value. Otherwise, bits [52:49] are part of the RESS field.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Bits [1:0]**

Reserved, RES0.

**Accessing DBGWVR\(n\)_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS \(X_t\), DBGWVR\(m\)_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0000</td>
<td>m[3:0]</td>
<td>0b110</td>
</tr>
</tbody>
</table>

integer \(m = \text{UInt}(\text{CRm}3:0);\)

if \(m >= \text{NUM\_WATCHPOINTS}\) then
  UNDEFINED;
elsif PSTATE\_EL == EL0 then
  UNDEFINED;
elsif PSTATE\_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION\_DEFINED "EL3 trap priority when SDD == '1'" && MDCR\_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) \&\& SCR\_EL3.FGTEn == '1') && HDFGRTR\_EL2.DBGWVRn\_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR\_EL2.<TDE,TDA> != '00' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && MDCR\_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = DBGWVR\_EL1[m];
  end
elsif PSTATE\_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION\_DEFINED "EL3 trap priority when SDD == '1'" && MDCR\_EL3.TDA == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR\_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  elsif OSLSR\_EL1.OSLK == '0' && Halted() && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt\_SoftwareAccess);
  else
    X[t, 64] = DBGWVR\_EL1[m];
  end
elsif PSTATE\_EL == EL3 then
  if OSLSR\_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt\_SoftwareAccess);
  else
    X[t, 64] = DBGWVR\_EL1[m];
  end
### MSR DBGWVR<m>_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0000</td>
<td>m[3:0]</td>
<td>0b110</td>
</tr>
</tbody>
</table>

```plaintext
integer m = UInt(CRm<3:0>);

if m >= NUM_WATCHPOINTS then
  UNDEFINED;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDGWTR_EL2.DBGWVRn_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  elsif EL2Enabled() && MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elif EL3Enabled() && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  elsif OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
  else
    DBGWVR_EL1[m] = X[t, 64];
  endif
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  elseif OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
  else
    DBGWVR_EL1[m] = X[t, 64];
  endif
elsif PSTATE_EL == EL3 then
  if OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
  else
    DBGWVR_EL1[m] = X[t, 64];
  endif
```

---

**Non-Confidential**

D17-6436  
Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  
ARM DDI 0487I.a  
ID081822
D17.3.13  DLR_EL0, Debug Link Register

The DLR_EL0 characteristics are:

Purpose

In Debug state, holds the address to restart from.

Configurations

AArch64 System register DLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register DLR[31:0].

Attributes

DLR_EL0 is a 64-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Restart address</td>
</tr>
<tr>
<td>31</td>
<td>Restart address</td>
</tr>
</tbody>
</table>

Bits [63:0]

Restart address.

Accessing DLR_EL0

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, DLR_EL0**

```plaintext
if !Halted() then
UNDEFINED;
else
X[t, 64] = DLR_EL0;
```

**MSR DLR_EL0, <Xt>**

```plaintext
if !Halted() then
UNDEFINED;
else
DLR_EL0 = X[t, 64];
```
D17.3.14 DSPSR_EL0, Debug Saved Program Status Register

The DSPSR_EL0 characteristics are:

Purpose

Holds the saved process state for Debug state. On entering Debug state, PSTATE information is written to this register. On exiting Debug state, values are copied from this register to PSTATE.

Configurations

AArch64 System register DSPSR_EL0 bits [31:0] are architecturally mapped to AArch32 System register DSPSR[31:0].

Attributes

DSPSR_EL0 is a 64-bit register.

Field descriptions

When AArch32 is supported and exiting Debug state to AArch32 state:

Bits [63:32]
Reserved, RES0.

N, bit [31]
Negative Condition flag. Copied to PSTATE.N on exiting Debug state.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]
Zero Condition flag. Copied to PSTATE.Z on exiting Debug state.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]
Carry Condition flag. Copied to PSTATE.C on exiting Debug state.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]
Overflow Condition flag. Copied to PSTATE.V on exiting Debug state.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]
Overflow or saturation flag. Copied to PSTATE.Q on exiting Debug state.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IT, bits [15:10, 26:25]**
If-Then. Copied to PSTATE.IT on exiting Debug state.
DSPSR_EL0.IT must contain a value that is valid for the instruction being returned to.
The IT field is split as follows:
- IT[1:0] is DSPSR_EL0[26:25].
- IT[7:2] is DSPSR_EL0[15:10].
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**DIT, bit [24]**

*When FEAT_DIT is implemented:*
Data Independent Timing. Copied to PSTATE.DIT on exiting Debug state.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*
Reserved, RES0.

**SSBS, bit [23]**

*When FEAT_SSBS is implemented:*
Speculative Store Bypass. Copied to PSTATE.SSBS on exiting Debug state.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*
Reserved, RES0.

**PAN, bit [22]**

*When FEAT_PAN is implemented:*
Privileged Access Never. Copied to PSTATE.PAN on exiting Debug state.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*
Reserved, RES0.

**SS, bit [21]**
Software Step. Copied to PSTATE.SS on exiting Debug state.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IL, bit [20]**
Illegal Execution state. Copied to PSTATE.IL on exiting Debug state.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**GE, bits [19:16]**
Greater than or Equal flags. Copied to PSTATE.GE on exiting Debug state.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**E, bit [9]**
Endianness. Copied to PSTATE.E on exiting Debug state.
If the implementation does not support big-endian operation, DSPSR_EL0.E is RES0. If the implementation does not support little-endian operation, DSPSR_EL0.E is RES1. On exiting Debug state, if the implementation does not support big-endian operation at the Exception level being returned to, DSPSR_EL0.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, DSPSR_EL0.E is RES1.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**A, bit [8]**
SError interrupt mask. Copied to PSTATE.A on exiting Debug state.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**I, bit [7]**
IRQ interrupt mask. Copied to PSTATE.I on exiting Debug state.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**F, bit [6]**
FIQ interrupt mask. Copied to PSTATE.F on exiting Debug state.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**T, bit [5]**
T32 Instruction set state. Copied to PSTATE.T on exiting Debug state.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**M[4], bit [4]**
Execution state. Copied to PSTATE.nRW on exiting Debug state.
0b1 AArch32 execution state.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**M[3:0], bits [3:0]**
AArch32 Mode. Copied to PSTATE.M[3:0] on exiting Debug state.
0b0000 User.
0b0001 FIQ.
0b0010 IRQ.
0b0011 Supervisor.
0b0110 Monitor.
0b0111 Abort.
0b1010 Hyp.
0b1011 Undefined.
0b1111 System.
Other values are reserved. If DSPSR_EL0.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, exiting Debug state is an illegal return event, as described in *Illegal exception returns from AArch64 state* on page D1-4645.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*When AArch64 is supported and entering or exiting Debug state from or to AArch64 state:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Reserved</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>
| 31  | N        | Negative Condition flag. Set to the value of PSTATE.N on entering Debug state, and copied to PSTATE.N on exiting Debug state. The reset behavior of this field is:  
  - On a Warm reset, this field resets to an architecturally UNKNOWN value. |
| 30  | Z        | Zero Condition flag. Set to the value of PSTATE.Z on entering Debug state, and copied to PSTATE.Z on exiting Debug state. The reset behavior of this field is:  
  - On a Warm reset, this field resets to an architecturally UNKNOWN value. |
| 29  | C        | Carry Condition flag. Set to the value of PSTATE.C on entering Debug state, and copied to PSTATE.C on exiting Debug state. The reset behavior of this field is:  
  - On a Warm reset, this field resets to an architecturally UNKNOWN value. |
| 28  | V        | Overflow Condition flag. Set to the value of PSTATE.V on entering Debug state, and copied to PSTATE.V on exiting Debug state. The reset behavior of this field is:  
  - On a Warm reset, this field resets to an architecturally UNKNOWN value. |
| 27-26| Reserved | Reserved, RES0.                                                             |
| 25  | TCO      | Tag Check Override. Set to the value of PSTATE.TCO on entering Debug state, and copied to PSTATE.TCO on exiting Debug state.  
When FEAT_MTE is not implemented, it is CONSTRAINED UNPREDICTABLE whether this field is RES0 or behaves as if FEAT_MTE is implemented. |
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

DIT, bit [24]

When FEAT_DIT is implemented:
Data Independent Timing. Set to the value of PSTATE.DIT on entering Debug state, and copied to PSTATE.DIT on exiting Debug state.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

UAO, bit [23]

When FEAT_UAO is implemented:
User Access Override. Set to the value of PSTATE.UAO on entering Debug state, and copied to PSTATE.UAO on exiting Debug state.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:
Privileged Access Never. Set to the value of PSTATE.PAN on entering Debug state, and copied to PSTATE.PAN on exiting Debug state.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on entering Debug state, and conditionally copied to PSTATE.SS on exiting Debug state.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on entering Debug state, and copied to PSTATE.IL on exiting Debug state.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:14]
Reserved, RES0.
ALLINT, bit [13]

When FEAT_NMI is implemented:

All IRQ or FIQ interrupts mask. Set to the value of PSTATE.ALLINT on entering Debug state, and copied to PSTATE.ALLINT on exiting Debug state.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSBS, bit [12]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on entering Debug state, and copied to PSTATE.SSBS on exiting Debug state.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BTYPE, bits [11:10]

When FEAT_BTI is implemented:

Branch Type Indicator. Set to the value of PSTATE.BTYPE on entering Debug state, and copied to PSTATE.BTYPE on exiting Debug state.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on entering Debug state, and copied to PSTATE.D on exiting Debug state.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on entering Debug state, and copied to PSTATE.A on exiting Debug state.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on entering Debug state, and copied to PSTATE.I on exiting Debug state.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on entering Debug state, and copied to PSTATE.F on exiting Debug state.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Bit [5]
Reserved, RES0.

M[4], bit [4]
Execution state. Set to 0b0, the value of PSTATE.nRW, on entering Debug state from AArch64 state, and copied to PSTATE.nRW on exiting Debug state.
0b0 AArch64 execution state.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]
AArch64 Exception level and selected Stack Pointer.
0b0000 EL0t.
0b0100 EL1t.
0b0101 EL1h.
0b1000 EL2t.
0b1001 EL2h.
0b1100 EL3t.
0b1101 EL3h.
Other values are reserved. If DSPSR_EL0.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, exiting Debug state is an illegal return event, as described in Illegal exception returns from AArch64 state on page D1-4645.
The bits in this field are interpreted as follows:
• M[3:2] is set to the value of PSTATE.EL on entering Debug state and copied to PSTATE.EL on exiting Debug state.
• M[1] is unused and is 0 for all non-reserved values.
• M[0] is set to the value of PSTATE.SP on entering Debug state and copied to PSTATE.SP on exiting Debug state.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DSPSR_EL0
Accesses to this register use the following encodings in the System register encoding space:

\[
\text{\textbf{MRS} } \langle \text{Xt} \rangle, \text{DSPSR_EL0}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0101</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if !Halted() then
   UNDEFINED;
else
   X[t, 64] = DSPSR_EL0;
MSR DSPSR_EL0, <Xt>

if !Halted() then
    UNDEFINED;
else
    DSPSR_EL0 = X[t, 64];

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0101</td>
<td>0b000</td>
</tr>
</tbody>
</table>
D17.3.15 MDCCINT_EL1, Monitor DCC Interrupt Enable Register

The MDCCINT_EL1 characteristics are:

**Purpose**

Enables interrupt requests to be signaled based on the DCC status flags.

**Configurations**

AArch64 System register MDCCINT_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGDCCINT[31:0].

**Attributes**

MDCCINT_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field Description</th>
<th>Value</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-31</td>
<td>Reserved, RES0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>RX, bit</td>
<td>DCC interrupt request enable control for DTRRX. Enables a common COMMIRQ interrupt request to be signaled based on the DCC status flags.</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td>0b0</td>
<td>No interrupt request generated by DTRRX.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b1</td>
<td>Interrupt request will be generated on RXfull == 1.</td>
</tr>
<tr>
<td>29</td>
<td>TX, bit</td>
<td>DCC interrupt request enable control for DTRTX. Enables a common COMMIRQ interrupt request to be signaled based on the DCC status flags.</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>0b0</td>
<td>No interrupt request generated by DTRTX.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b1</td>
<td>Interrupt request will be generated on TXfull == 0.</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If legacy COMMRX and COMMTX signals are implemented, then these are not affected by the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

**Accessing MDCCINT_EL1**

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, MDCCINT_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
X[t, 64] = MDCCINT_EL1;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDCC == '1' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && MDCR_EL2.TDCC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.TDA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = MDCCINT_EL1;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDCC == '1' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = MDCCINT_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = MDCCINT_EL1;
MSR MDCCINT_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elseif Halted() && ConstraintUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
  MDCCINT_EL1 = X[t, 64];
elseif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'' & MDCC_EL3.TDCC == '1'' then
    UNDEFINED;
elseif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'' then
    UNDEFINED;
elseif EL2Enabled() && MDCC_EL2.TDCC == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
elseif EL2Enabled() && MDCC_EL2.TDCC != '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
elseif HaveEL(EL3) && MDCC_EL3.TDCC == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
elses
      AArch64.SystemAccessTrap(EL3, 0x18);
else
  MDCCINT_EL1 = X[t, 64];
elseif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'' & MDCC_EL3.TDCC == '1'' then
    UNDEFINED;
elses
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
elses
      AArch64.SystemAccessTrap(EL3, 0x18);
elseif HaveEL(EL3) && MDCC_EL3.TDCC == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
elses
      AArch64.SystemAccessTrap(EL3, 0x18);
else
  MDCCINT_EL1 = X[t, 64];
elseif PSTATE.EL == EL3 then
  MDCCINT_EL1 = X[t, 64];
D17.3.16 MDCCSR_EL0, Monitor DCC Status Register

The MDCCSR_EL0 characteristics are:

**Purpose**

Read-only register containing control status flags for the DCC.

**Configurations**

AArch64 System register MDCCSR_EL0 bits [30:29] are architecturally mapped to External register EDSCR[30:29].

AArch64 System register MDCCSR_EL0 bits [30:29] are architecturally mapped to AArch32 System register DBGDSCRint[30:29].

**Attributes**

MDCCSR_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:31]</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td></td>
<td>DTRRX full. Read-only view of the equivalent bit in the EDSCR.</td>
</tr>
<tr>
<td>[29]</td>
<td>TXfull, bit [29]</td>
</tr>
<tr>
<td></td>
<td>DTRTX full. Read-only view of the equivalent bit in the EDSCR.</td>
</tr>
<tr>
<td>[28:19]</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>[18:15]</td>
<td>Reserved, RAZ</td>
</tr>
<tr>
<td>[14:13]</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>[12]</td>
<td>Reserved, RAZ</td>
</tr>
<tr>
<td>[5:2]</td>
<td>Reserved, RAZ</td>
</tr>
<tr>
<td>[1:0]</td>
<td>Reserved, RES0</td>
</tr>
</tbody>
</table>
Accessing MDCCSR_EL0

Accesses to this register use the following encodings in the System register encoding space:

\[
MRS <Xt>, MDCCSR_EL0
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b011</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if Halted() \&\& ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    X[t, 64] = MDCCSR_EL0;
elsif PSTATE_EL == EL0 then
    if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCR_EL3.TDCC == '1' then
        UNDEFINED;
    elsif Halted() \&\& HaveEL(EL1) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elsif MDSCR_EL1.TDCC == '1' then
        if EL2Enabled() \&\& MDCR_EL2.TDCC == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        endif
    elsif MDSCR_EL1.TDA == '1' then
        if EL2Enabled() \&\& MDCR_EL2.TDA == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        endif
    elsif EL2Enabled() \&\& HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if EL2Enabled() \&\& (HCR_EL2.TGE == '1' \&\& MDCR_EL2.<TDE,TDA> != '00') then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        endif
    endif
else
    X[t, 64] = MDCCSR_EL0;
elsif PSTATE_EL == EL1 then
    if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCR_EL3.TDCC == '1' then
        UNDEFINED;
    elsif Halted() \&\& HaveEL(EL1) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elsif EL2Enabled() \&\& MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if EL2Enabled() \&\& MDCR_EL2.TDA == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        endif
    endif
else
    X[t, 64] = MDCCSR_EL0;
elsif PSTATE_EL == EL2 then
    if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCR_EL3.TDCC == '1' then
        UNDEFINED;
    elsif Halted() \&\& HaveEL(EL1) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elseif MDSCR_EL2.TDCC == '1' then
        if EL2Enabled() \&\& MDCR_EL2.TDCC == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        endif
    elseif MDSCR_EL2.TDA == '1' then
        if EL2Enabled() \&\& MDCR_EL2.TDA == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        endif
    elsif EL2Enabled() \&\& HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if EL2Enabled() \&\& (HCR_EL2.TGE == '1' \&\& MDCR_EL2.<TDE,TDA> != '00') then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        endif
    endif
else
    X[t, 64] = MDCCSR_EL0;
elsif PSTATE_EL == EL3 then
    if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCR_EL3.TDCC == '1' then
        UNDEFINED;
    elsif Halted() \&\& HaveEL(EL1) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elseif EL2Enabled() \&\& MDCR_EL2.TDCC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if EL2Enabled() \&\& MDCR_EL2.TDA == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        endif
    endif
else
    X[t, 64] = MDCCSR_EL0;
when SDD == '1' && MDCR_EL3.TDCC == '1' then
  UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
  UNDEFINED;
elsif HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end if;
elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end if;
else
  X[t, 64] = MDCCSR_EL0;
elsif PSTATE_EL == EL3 then
  X[t, 64] = MDCCSR_EL0;
### D17.3.17  MDCR_EL2, Monitor Debug Configuration Register (EL2)

The MDCR_EL2 characteristics are:

**Purpose**

Provides EL2 configuration options for self-hosted debug and the Performance Monitors Extension.

**Configurations**

AArch64 System register MDCR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HDCR[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

MDCR_EL2 is a 64-bit register.

**Field descriptions**

[Diagram of field descriptions]

**Bits [63:37]**

Reserved, RES0.

**HPMFZS, bit [36]**

*When FEAT_SPEv1p2 is implemented:*

Hyp Performance Monitors Freeze-on-SPE event. Stop counters when PMBLIMITR_EL1.{PMFZ, E} == {1, 1} and PMBSR_EL1.S == 1.

0b0 Do not freeze on Statistical Profiling Buffer Management event.

0b1 Event counters do not count following a Statistical Profiling Buffer Management event.

If MDCR_EL2.HPMN is less than PMCR_EL0.N, this field affects the operation of event counters in the range [MDCR_EL2.HPMN .. (PMCR_EL0.N-1)].

This field does not affect the operation of other event counters and PMCCNTR_EL0.

The operation of this field applies even when EL2 is disabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bits [35:30]**

Reserved, RES0.
HPMFZO, bit [29]

When FEAT_PMU\textsubscript{v3p7} is implemented:

Hyp Performance Monitors Freeze-on-overflow. Stop event counters on overflow.

- 0b0 \hspace{1em} Do not freeze on overflow.
- 0b1 \hspace{1em} Event counters do not count when \texttt{PMOVNSCLR\_EL0\_EL0.N-1\_MDCR\_EL2.HPMN} is nonzero.

If MDCR\_EL2.HPMN is less than PMCR\_EL0.N, this field affects the operation of event counters in the range [MDCR\_EL2.HPMN .. (PMCR\_EL0.N-1)].

This field does not affect the operation of other event counters and PMCCNTR\_EL0.

The operation of this field applies even when EL2 is disabled in the current Security state.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

MTPME, bit [28]

When FEAT_MTPMU is implemented and EL3 is not implemented:

Multi-threaded PMU Enable. Enables use of the \texttt{PMEVTYPE}\textsubscript{\textit{n}}\_EL0.MT bits.

- 0b0 \hspace{1em} FEAT_MTPMU is disabled. The Effective value of \texttt{PMEVTYPE}\textsubscript{\textit{n}}\_EL0.MT is zero.
- 0b1 \hspace{1em} \texttt{PMEVTYPE}\textsubscript{\textit{n}}\_EL0.MT bits not affected by this field.

If FEAT_MTPMU is disabled for any other PE in the system that has the same level 1 Affinity as the PE, it is IMPLEMENTATION DEFINED whether the PE behaves as if this field is 0.

The reset behavior of this field is:
- On a Cold reset, this field resets to 1.

Otherwise:

Reserved, RES0.

TDCC, bit [27]

When FEAT_FGT is implemented:

Trap DCC. Traps use of the Debug Comms Channel at EL1 and EL0 to EL2.

- 0b0 \hspace{1em} This control does not cause any register accesses to be trapped.
- 0b1 \hspace{1em} If EL2 is implemented and enabled in the current Security state, accesses to the DCC registers at EL1 and EL0 generate a Trap exception to EL2, unless the access also generates a higher priority exception.

Traps on the DCC data transfer registers are ignored when the PE is in Debug state.

The DCC registers trapped by this control are:

AArch64: OSDTRRX\_EL1, OSDTRTX\_EL1, MDCCSR\_EL0, MDCCINT\_EL1, and, when the PE is in Non-debug state, DBGDTR\_EL0, DBGDTRRX\_EL0, and DBGDTRTX\_EL0.

AArch32: DBGDTRRX\text{\textsubscript{ext}}, DBGDTRTX\text{\textsubscript{ext}}, DBGDSCR\text{\textsubscript{int}}, DBGDCCINT, and, when the PE is in Non-debug state, DBGDTR\text{\textsubscript{RX\text{\textsubscript{int}}} and DBGDTR\text{\textsubscript{TX\text{\textsubscript{int}}}}.

The traps are reported with EC syndrome value:

- 0x05 for trapped AArch32 \texttt{MRC} and \texttt{MCR} accesses with coproc == 0b1110.
- 0x06 for trapped AArch32 \texttt{LDC} to DBGDTR\text{\textsubscript{RX\text{\textsubscript{int}}} and \texttt{STC} from DBGDTR\text{\textsubscript{RX\text{\textsubscript{int}}}.
- 0x18 for trapped AArch64 \texttt{MRS} and \texttt{MSR} accesses.

When the PE is in Debug state, MDCR\_EL2.TDCC does not trap any accesses to:

AArch64: DBGDTR\_EL0, DBGDTRRX\_EL0, and DBGDTRTX\_EL0.
AArch32: DBGDTRRXint and DBGDTRTXint.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**HLP, bit [26]**

*When FEAT_PMUv3p5 is implemented:*

Hypervisor Long event counter enable. Determines when unsigned overflow is recorded by an event counter overflow bit.

- \(0b0\) Event counter overflow on increment that causes unsigned overflow of PMEVCNTR\(<n>\)_EL0[31:0].
- \(0b1\) Event counter overflow on increment that causes unsigned overflow of PMEVCNTR\(<n>\)_EL0[63:0].

If MDCR_EL2.HPMN is less than PMCR_EL0.N, this bit affects the operation of event counters in the range \([MDCR_EL2.HPMN..(PMCR_EL0.N-1)]\).

This field does not affect the operation of other event counters.

The operation of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**E2TB, bits [25:24]**

*When FEAT_TRBE is implemented:*

EL2 Trace Buffer.

If EL2 is implemented and enabled in the Trace Buffer owning Security state, controls the owning translation regime.

If EL2 is implemented and enabled in the current Security state, controls access to Trace Buffer control registers from EL1.

- \(0b00\) If EL2 is implemented and enabled in the Trace Buffer owning Security state, the Trace Buffer owning Exception level is EL2. Otherwise, the Trace Buffer owning Exception level is EL1 and, if TraceBufferEnabled() == TRUE, tracing is prohibited at EL2.
- \(0b10\) Trace Buffer owning Exception level is EL1. If TraceBufferEnabled() == TRUE, tracing is prohibited at EL2.
- \(0b11\) Trace Buffer owning Exception level is EL1. If TraceBufferEnabled() == TRUE, tracing is prohibited at EL2.

All other values are reserved.

The Trace Buffer control registers trapped by this control are: TRBBASER_EL1, TRBLIMITR_EL1, TRBMAR_EL1, TRBPTR_EL1, TRBSR_EL1, and TRBTRG_EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.
HCCD, bit [23]

*When FEAT_PMUv3p5 is implemented:*

Hypervisor Cycle Counter Disable. Prohibits PMCCNTR_EL0 from counting at EL2.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Cycle counting by PMCCNTR_EL0 is not affected by this mechanism.</td>
</tr>
<tr>
<td>0b1</td>
<td>Cycle counting by PMCCNTR_EL0 is prohibited at EL2.</td>
</tr>
</tbody>
</table>

This field does not affect the CPU_CYCLES event or any other event that counts cycles.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

*Otherwise:*

Reserved, RES0.

Bits [22:20]

Reserved, RES0.

TTRF, bit [19]

*When FEAT_TRF is implemented:*

Traps use of the Trace Filter Control registers at EL1 to EL2, as follows:

- Access to TRFCR_EL1 is trapped to EL2, reported using EC syndrome value 0x18.
- Access to TRFCR is trapped to EL2, reported using EC syndrome value 0x03.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Accesses to the specified registers at EL1 are not affected by this control.</td>
</tr>
<tr>
<td>0b1</td>
<td>Accesses to the specified registers at EL1 generate a trap exception to EL2 when EL2 is enabled in the current Security state.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

Bit [18]

Reserved, RES0.

HPMD, bit [17]

*When FEAT_PMUv3p1 is implemented and FEAT_Debugv8p2 is implemented:*

Guest Performance Monitors Disable. Controls event counting by some event counters at EL2.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Event counting and PMCCNTR_EL0 are not affected by this mechanism.</td>
</tr>
<tr>
<td>0b1</td>
<td>Event counting by some event counters is prohibited at EL2. If PMCR_EL0.DP is 1, PMCCNTR_EL0 is disabled at EL2. Otherwise, PMCCNTR_EL0 is not affected by this mechanism.</td>
</tr>
</tbody>
</table>

If MDCR_EL2.HPMN is not 0, this field affects the operation of event counters in the range [0 .. (MDCR_EL2.HPMN-1)].

This field does not affect the operation of other event counters.

If PMCR_EL0.DP is 1, this field affects PMCCNTR_EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

*When FEAT_PMUv3p3p1 is implemented:*

Guest Performance Monitors Disable. Controls event counting by some event counters at EL2.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Event counting and PMCCNTR_EL0 are not affected by this mechanism.</td>
</tr>
</tbody>
</table>
If `ExternalSecureNoninvasiveDebugEnabled()` is FALSE, event counting by some event counters is prohibited at EL2, and if `PMCR_EL0.DP` is 1, `PMCCNTR_EL0` is disabled at EL2.

If `ExternalSecureNoninvasiveDebugEnabled()` is TRUE, this field does not affect the event counters and does not affect `PMCCNTR_EL0`.

Otherwise:

- If `MDCR_EL2.HPMN` is not 0, this field affects the operation of event counters in the range `[0 .. (MDCR_EL2.HPMN-1)]`.
- This field does not affect the operation of other event counters.
- If `PMCR_EL0.DP` is 1, this field affects `PMCCNTR_EL0`.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**Bits [16:15]**

Reserved, RES0.

**TPMS, bit [14]**

*When FEAT_SPE is implemented:*

Trap Performance Monitor Sampling. If EL2 is implemented and enabled in the current Security state, controls access to Statistical Profiling control registers from EL1.

0b0 Do not trap Statistical Profiling controls to EL2.

0b1 If EL2 is implemented and enabled in the current Security state, accesses to Statistical Profiling control registers at EL1 generate a Trap exception to EL2.

The Statistical Profiling control registers trapped by this control are:

- `PMSCR_EL1, PMSEVFR_EL1, PMSFCR_EL1, PMSICR_EL1, PMSIDR_EL1, PMSIRR_EL1, and PMSLATFR_EL1`.
- If `FEAT_SPEv1p2` is implemented, `PMSNEVFR_EL1`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**E2PB, bits [13:12]**

*When FEAT_SPE is implemented:*

EL2 Profiling Buffer. If EL2 is implemented and enabled in the Profiling Buffer owning Security state, this field controls the owning translation regime. If EL2 is implemented and enabled in the current Security state, this field controls access to Profiling Buffer control registers from EL1.

0b00 If EL2 is implemented and enabled in the Profiling Buffer owning Security state, the Profiling Buffer uses the EL2 or EL2&0 stage 1 translation regime. Otherwise the Profiling Buffer uses the EL1&0 stage 1 translation regime.

If EL2 is implemented and enabled in the current Security state, accesses to Profiling Buffer control registers at EL1 generate a Trap exception to EL2.

0b10 Profiling Buffer uses the EL1&0 stage 1 translation regime. If EL2 is implemented and enabled in the current Security state, accesses to Profiling Buffer control registers at EL1 generate a Trap exception to EL2.

0b11 Profiling Buffer uses the EL1&0 stage 1 translation regime. Accesses to Profiling Buffer control registers at EL1 are not trapped to EL2.

All other values are reserved.
The Profiling Buffer control registers trapped by this control are: **PMBLIMITR_EL1**, **PMBPTR_EL1**, and **PMBSR_EL1**.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**

Reserved, **RES0**.

**TDRA, bit [11]**

Trap Debug ROM Address register access. Traps System register accesses to the Debug ROM registers to EL2 when EL2 is enabled in the current Security state as follows:

- If EL1 is using AArch64 state, accesses to **MDRAR_EL1** are trapped to EL2, reported using EC syndrome value **0x18**.
- If EL0 or EL1 is using AArch32 state, MRC or MCR accesses to the following registers are trapped to EL2, reported using EC syndrome value **0x05** and MRRC or MCRR accesses are trapped to EL2, reported using EC syndrome value **0x0C**:
  - **DBGDRAR**, **DBGDSAR**.

0b0 This control does not cause any instructions to be trapped.

0b1 EL0 and EL1 System register accesses to the Debug ROM registers are trapped to EL2 when EL2 is enabled in the current Security state, unless it is trapped by the following:

- **DBGDSCRTD** and **MDCD_MDLR_TDCC**.

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- **MDCR_EL2.TDE == 1**.
- **HCR_EL2.TGE == 1**.

---

**Note**

EL2 does not provide traps on debug register accesses through the optional memory-mapped external debug interfaces.

System register accesses to the debug registers might have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**TDOSA, bit [10]**

*When FEAT_DoubleLock is implemented:*

Trap debug OS-related register access. Traps EL1 System register accesses to the powerdown debug registers to EL2, from both Execution states as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value **0x18**:
  - **OSLR_EL1**, **OSLSR_EL1**, **OSDLR_EL1**, and **DBGPRCR_EL1**.
  - Any **IMPLEMENTATION DEFINED** register with similar functionality that the implementation specifies as trapped by this bit.
- In AArch32 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value **0x05**:
  - **DBGOSLSR**, **DBGOSLAR**, **DBGOSDLR**, and **DBGPRCR**.
  - Any **IMPLEMENTATION DEFINED** register with similar functionality that the implementation specifies as trapped by this bit.

0b0 This control does not cause any instructions to be trapped.
EL1 System register accesses to the powerdown debug registers are trapped to EL2 when EL2 is enabled in the current Security state.

--- Note ---

These registers are not accessible at EL0.

---

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- MDCR_EL2.TDE == 1.
- HCR_EL2.TGE == 1.

System register accesses to the debug registers might have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Trap debug OS-related register access. Traps EL1 System register accesses to the powerdown debug registers to EL2, from both Execution states as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value 0x18:
  - OSLAR_EL1, OSLSR_EL1, and DBGPRCR_EL1.
  - Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.
- In AArch32 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value 0x05:
  - DBGOSLSR, DBGOSLR, and DBGPRCR.
  - Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

It is IMPLEMENTATION DEFINED whether accesses to OSDLR_EL1 are trapped.

It is IMPLEMENTATION DEFINED whether accesses to DBGOSDLR are trapped.

This control does not cause any instructions to be trapped.

EL1 System register accesses to the powerdown debug registers are trapped to EL2 when EL2 is enabled in the current Security state.

--- Note ---

These registers are not accessible at EL0.

---

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- MDCR_EL2.TDE == 1.
- HCR_EL2.TGE == 1.

--- Note ---

EL2 does not provide traps on debug register accesses through the optional memory-mapped external debug interfaces.

---

System register accesses to the debug registers might have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
**TDA, bit [9]**

Trap Debug Access. Traps EL0 and EL1 System register accesses to debug System registers that are not trapped by MDCR_EL2.TDRA or MDCR_EL2.TDOSA, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2 reported using EC syndrome value 0x18:
  - MDCCSR_EL0, MDCCINT_EL1, OSDTRRX_EL1, OSDTRTX_EL1, OSECCR_EL1, DBGBVR<n>_EL1, DBGBCR<n>_EL1, DBGWVR<n>_EL1, DBGWCR<n>_EL1, DBGCLAIMSET_EL1, DBGCLAIMCLR_EL1, DBGAUTHSTATUS_EL1.
  - When not in Debug state, DBGDTR_EL0, DBGDTRRX_EL0, DBGDTRTX_EL0.

- In AArch32 state, MRC or MCR accesses to the following registers are trapped to EL2, reported using EC syndrome value 0x05.
  - DBGDIDR, DBGDCINT, DBGFWCR, DBGDCRext, DBGDTRTXext, DBGVCR, DBGDSCINT, DBGDTRRXext, DBGWCR<n>, DBGWVR<n>, DBGCLAIMSET, DBGCLAIMCLR, DBGAUTHSTATUS, DBGDEVID, DBGDEVID1, DBGDEVID2, DBGSECCR.
  - When not in Debug state, DBGDTRRXint and DBGDTRTXint.

- In AArch32 state, STC accesses to DBGDTRRXint and LDC accesses to DBGDTRTXint are trapped to EL2, reported using EC syndrome value 0x06.

- This control does not cause any instructions to be trapped.

- EL0 or EL1 System register accesses to the debug registers are trapped from both Execution states to EL2 when EL2 is enabled in the current Security state, unless the access generates a higher priority exception.

- Traps of AArch32 accesses to DBGDTRRXint and DBGDTRTXint are ignored in Debug state.

- Traps of AArch64 accesses to DBGDTR_EL0, DBGDTRRX_EL0, and DBGDTRTX_EL0 are ignored in Debug state.

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- MDCR_EL2.TDE == 1
- HCR_EL2.TGE == 1

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**TDE, bit [8]**

Trap Debug Exceptions. Controls routing of Debug exceptions, and defines the debug target Exception level, EL0.

- The debug target Exception level is EL1.

- If EL2 is enabled for the current Effective value of SCR_EL3.NS, the debug target Exception level is EL2, otherwise the debug target Exception level is EL1.

For more information, see *Routing debug exceptions on page D2-4697*.

This field is treated as being 1 for all purposes other than a direct read when HCR_EL2.TGE == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**HPME, bit [7]**

*When FEAT_PMUv3 is implemented:*

([MDCR_EL2.HPMN..(N-1)]) event counters enable.

- Event counters in the range [MDCR_EL2.HPMN..(PMCR_EL0.N-1)] are disabled.
0b1  Event counters in the range [MDCR_EL2.HPMN..(PMCR_EL0.N-1)] are enabled by PMCNTENSET_EL0.

If MDCR_EL2.HPMN is less than PMCR_EL0.N, this field affects the operation of event counters in the range [MDCR_EL2.HPMN..(PMCR_EL0.N-1)].

This field does not affect the operation of other event counters.

The operation of this field applies even when EL2 is disabled in the current Security state.

The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**

Reserved, RES0.

TPM, bit [6]

*When FEAT_PMUv3 is implemented:*

Trap Performance Monitors accesses. Traps EL0 and EL1 accesses to all Performance Monitor registers to EL2 when EL2 is enabled in the current Security state, from both Execution states, as follows:

• In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value \(0x18\):
  — PMCR_EL0, PMCNTENSET_EL0, PMCNTENCLR_EL0, PMOVVSLRR_EL0, PMSWINC_EL0, PMSELR_EL0, PMCEID0_EL0, PMCEID1_EL0, PMCCNTR_EL0, PMXEVTYPE_EL0, PMXEVCNTR_EL0, PMUSERENR_EL0, PMINTENSET_EL1, PMINTENCLR_EL1, PMOVSET_EL0, PMOVSCLR_EL0, PMEVCNTR<\text{n}>_EL0, PMEVTYPER<\text{n}>_EL0, PMCCFILTR_EL0.
  — If FEAT_PMUv3p4 is implemented, PMMIR_EL1

• In AArch32 state, MRC or MCR accesses to the following registers are trapped to EL2 and reported using EC syndrome value \(0x03\), MRRC or MCRR accesses are trapped to EL2 and reported using EC syndrome value \(0x04\):
  — PMCR, PMCNTENSET, PMCNTENCLR, PMOVSR, PMSWINC, PMSELR, PMCEID0, PMCEID1, PMCCNTR, PMXEVTYPE, PMXEVCNTR, PMUSERENR, PMINTENSET, PMINTENCLR, PMOVSET, PMOVSSET, PMEVCNTR<\text{n}>, PMEVTYPER<\text{n}>, PMCCFILTR.
  — If FEAT_PMUv3p1 is implemented, PMCEID2, and PMCEID3.
  — If FEAT_PMUv3p4 is implemented, PMMIR.

0b0  This control does not cause any instructions to be trapped.

0b1  EL0 and EL1 accesses to all Performance Monitor registers are trapped to EL2 when EL2 is enabled in the current Security state.

---

**Note**

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

---

The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**

Reserved, RES0.
TPMCR, bit [5]

**When FEAT_PMUv3 is implemented:**

Trap PMCR_EL0 or PMCR accesses. Traps EL0 and EL1 accesses to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to PMCR_EL0 are trapped to EL2, reported using EC syndrome value 0x18.
- In AArch32 state, accesses to PMCR are trapped to EL2, reported using EC syndrome value 0x03.

0b0  This control does not cause any instructions to be trapped.

0b1  EL0 and EL1 accesses to the specified registers are trapped to EL2 when EL2 is enabled in the current Security state, unless it is trapped by the following:

- PMUSERENR.EN.
- PMUSERENR_EL0.EN.

**Note**

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

HPMN, bits [4:0]

**When FEAT_PMUv3 is implemented:**

Defines the number of event counters that are accessible from EL3, EL2, EL1, and from EL0 if permitted.

If HPMN is not 0 and is less than PMCR_EL0.N, HPMN divides the event counters into a first range [0..(HPMN-1)], and a second range [HPMN..(PMCR_EL0.N-1)].

If FEAT_HPMN0 is implemented and this field is 0, all event counters are in the second range and none are in the first range.

If HPMN is equal to PMCR_EL0.N, all event counters are in the first range and none are in the second range.

For an event counter <n> in the first range:

- The counter is accessible from EL1, EL2, and EL3.
- The counter is accessible from EL0 if permitted by PMUSERENR_EL0 or PMUSERENR.
- If FEAT_PMUv3p5 is implemented, PMCR_EL0.LP or PMCR_LP determines whether the counter overflow flag is set on unsigned overflow of PMEVCNTR<n>_EL0[31:0] or PMEVCNTR<n>_EL0[63:0].
- PMCR_EL0.E and PMCNTENSET_EL0[n] enable the operation of event counter n.

For an event counter <n> in the second range:

- The counter is accessible from EL2 and EL3.
- If EL2 is disabled in the current Security state, the event counter is also accessible from EL1, and from EL0 if permitted by PMUSERENR_EL0.
- If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP determines whether the counter overflow flag is set on unsigned overflow of PMEVCNTR<n>_EL0[31:0] or PMEVCNTR<n>_EL0[63:0].
- MDCR_EL2.HPME and PMCNTENSET_EL0[n] enable the operation of event counter n.
If HPMN is larger than `PMCR_EL0.N`, or if `FEAT_HPMN0` is not implemented and HPMN is 0, the following CONSTRAINED UNPREDICTABLE behaviors apply:

- The value returned by a direct read of `MDCR_EL2.HPMN` is UNKNOWN.
- Either:
  - An UNKNOWN number of counters are reserved for EL2 and EL3 use. That is, the PE behaves as if `MDCR_EL2.HPMN` is set to an UNKNOWN non-zero value less than or equal to `PMCR_EL0.N`.
  - All counters are reserved for EL2 and EL3 use, meaning no counters are accessible from EL1 and EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to the value in `PMCR_EL0.N`.

**Otherwise:**

Reserved, RES0.

### Accessing MDCR_EL2

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, MDCR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = MDCR_EL2;
elsif PSTATE_EL == EL3 then
    X[t, 64] = MDCR_EL2;
```

**MSR MDCR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE_EL == EL2 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
...
AArch64.SystemAccessTrap(EL2, 0x18);
else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority" when SDD == '1'" && MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            MDCR_EL2 = X[t, 64];
        elsif PSTATE_EL == EL3 then
            MDCR_EL2 = X[t, 64];
**D17.3.18 MDCR_EL3, Monitor Debug Configuration Register (EL3)**

The MDCR_EL3 characteristics are:

**Purpose**

Provides EL3 configuration options for self-hosted debug and the Performance Monitors Extension.

**Configurations**

This register is present only when EL3 is implemented. Otherwise, direct accesses to MDCR_EL3 are UNDEFINED.

**Attributes**

MDCR_EL3 is a 64-bit register.

**Field descriptions**

**Bits [63:39]**

Reserved, RES0.

**E3BREC, bit [38]**

When FEAT_BRBEv1p1 is implemented:

Branch Record Buffer EL3 Cold Reset Enable. With MDCR_EL3.E3BRE, controls branch recording at EL3.

- **0b0** When MDCR_EL3.E3BRE == 0: Branch recording at EL3 is disabled.
- **0b1** When MDCR_EL3.E3BRE == 1: Branch recording at EL3 is enabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.

**Otherwise:**

Reserved, RES0.
E3BREW, bit [37]

*When FEAT_BRBEv1p1 is implemented:*

Branch Record Buffer EL3 Warm Reset Enable. With MDCR_EL3.E3BREC, controls branch recording at EL3.

For a description of the values derived by evaluating MDCR_EL3.E3BREC and MDCR_EL3.E3BREW together, see MDCR_EL3.E3BREC.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

*Otherwise:*

Reserved, RES0.

EnPMSN, bit [36]

*When FEAT_SPEv1p2 is implemented:*

Trap accesses to PMSNEVFR_EL1. Controls access to Statistical Profiling PMSNEVFR_EL1 System register from EL2 and EL1.

- 0b0: Accesses to PMSNEVFR_EL1 at EL2 and EL1 generate a Trap exception to EL3.
- 0b1: Do not trap PMSNEVFR_EL1 to EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

*Otherwise:*

Reserved, RES0.

MPMX, bit [35]

*When FEAT_PMUv3p7 is implemented:*

Monitor Performance Monitors Extended control. In conjunction with MDCR_EL3.SPME, controls when event counters are enabled at EL3 and in other Secure Exception levels.

- 0b0: Event counting and PMCCNTR_EL0 are not affected by this mechanism.
- 0b1: Event counting by some or all event counters is prohibited at EL3. If PMCR_EL0.DP is 1, PMCCNTR_EL0 is disabled at EL3. Otherwise, PMCCNTR_EL0 is not affected by this mechanism.

If EL2 is implemented, MDCR_EL3.SPME == 1, and MDCR_EL2.HPMN is less than PMCR_EL0.N then all the following are true:

- If MDCR_EL2.HPMN is not 0, this field affects the operation of event counters in the range [0..(MDCR_EL2.HPMN-1)] at EL3.
- This field does not affect the operation of other event counters.
- If PMCR_EL0.DP is 1, this field affects the operation of PMCCNTR_EL0 at EL3.

The operation of this field applies even when EL2 is disabled in the current Security state.

If EL2 is not implemented, MDCR_EL3.SPME == 0, or MDCR_EL2.HPMN is equal to PMCR_EL0.N then this field affects the operation of all event counters at EL3, and if PMCR_EL0.DP is 1, the operation of PMCCNTR_EL0 at EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

*Otherwise:*

Reserved, RES0.
MCCD, bit [34]

When FEAT_PMUV3p7 is implemented:

Monitor Cycle Counter Disable. Prohibits the Cycle Counter, PMCCNTR_EL0, from counting at EL3.

- 0b0 Cycle counting by PMCCNTR_EL0 is not affected by this mechanism.
- 0b1 Cycle counting by PMCCNTR_EL0 is prohibited at EL3.

This field does not affect the CPU_CYCLES event or any other event that counts cycles.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

Otherwise:

Reserved, RES0.

SBRBE, bits [33:32]

When FEAT_BRBVE is implemented:

Secure Branch Record Buffer Enable. Controls branch recording by the BRBE, and access to BRBE registers and instructions at EL2 and EL1.

- 0b00 Direct accesses to BRBE registers and instructions, except when in EL3, generate a Trap exception to EL3. EL0, EL1, and EL2 are prohibited regions.
- 0b01 Direct accesses to BRBE registers and instructions in Secure state, except when in EL3, generate a Trap exception to EL3. EL0, EL1, and EL2 in Secure state are prohibited regions. This control does not cause any direct accesses to BRBE registers when not in Secure state to be trapped, and does not cause any Exception levels when not in Secure state to be a prohibited region.
- 0b10 Direct accesses to BRBE registers and instructions, except when in EL3, generate a Trap exception to EL3. This control does not cause any Exception levels to be prohibited regions.
- 0b11 This control does not cause any direct accesses to BRBE registers or instruction to be trapped, and does not cause any Exception levels to be a prohibited region.

The Branch Record Buffer registers trapped by this control are: BRBCR_EL1, BRBCR_EL2, BRBCR_EL12, BRBCR_EL1, BRBIFDR0_EL1, BRBIFV<np>_EL1, BRBSTC<np>_EL1, BRBSTCINJ_EL1, BRBTGT<np>_EL1, BRBTGTINJ_EL1, and BRBSTS_EL1.

The Branch Record Buffer instructions trapped by this control are:
- BRB IALL.
- BRB INJ.

Note

If FEAT_BRBVE is not implemented, EL3 is a prohibited region.

Otherwise:

If EL3 is not implemented then the Effective value of this field is 0b11.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [31:29]

Reserved, RES0.
MTPME, bit [28]

When FEAT_MTPMU is implemented:
Multi-threaded PMU Enable. Enables use of the PMEVTPYPER<ν>_EL0.MT bits.

0b0  FEAT_MTPMU is disabled. The Effective value of PMEVTPYPER<ν>_EL0.MT is zero.

0b1  PMEVTPYPER<ν>_EL0.MT bits not affected by this field.

If FEAT_MTPMU is disabled for any other PE in the system that has the same level 1 Affinity as the PE, it is IMPLEMENTATION DEFINED whether the PE behaves as if this field is 0.

The reset behavior of this field is:
• On a Cold reset, this field resets to 1.

Otherwise:
Reserved, RES0.

TDCC, bit [27]

When FEAT_FGT is implemented:
Trap DCC. Traps use of the Debug Comms Channel at EL2, EL1, and EL0 to EL3.

0b0  This control does not cause any register accesses to be trapped.

0b1  Accesses to the DCC registers at EL2, EL1, and EL0 generate a Trap exception to EL3, unless the access also generates a higher priority exception.

Traps on the DCC data transfer registers are ignored when the PE is in Debug state.

The DCC registers trapped by this control are:
AArch64: OSDTRRX_EL1, OSDTRTX_EL1, MDCCSR_EL0, MDCCINT_EL1, and, when the PE is in Non-debug state, DBGDTR_EL0, DBGDTRRX_EL0, and DBGDTRTX_EL0.
AArch32: DBGDTRRXext, DBGDTRTXext, DBGDSCRint, DBGDCCINT, and, when the PE is in Non-debug state, DBGDTRRXint and DBGDTRTXint.

The traps are reported with EC syndrome value:
• 0x05 for trapped AArch32 MRC and MCR accesses with coproc == 0b1110.
• 0x06 for trapped AArch32 LDC to DBGDTRTXint and STC from DBGDTRRXint.
• 0x18 for trapped AArch64 MSR and MSR accesses.

When the PE is in Debug state, MDCR_EL3.TDCC does not trap any accesses to:
AArch64: DBGDTR_EL0, DBGDTRRX_EL0, and DBGDTRTX_EL0.
AArch32: DBGDTRRXint and DBGDTRTXint.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

NSTBE, bit [26]

When FEAT_TRBE is implemented and FEAT_RME is implemented:
Non-secure Trace Buffer Extended. Together with MDCR_EL3.NSTB, controls the owning translation regime and accesses to Trace Buffer control registers from EL2 and EL1.

For a description of the values derived by evaluating NSTB and NSTBE together, see MDCR_EL3.NSTB.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.
NSTB, bits [25:24]

When FEAT_TRBE is implemented and FEAT_RME is implemented:

Non-secure Trace Buffer. Together with MDCR_EL3.NSTBE, controls the owning translation regime and accesses to Trace Buffer control registers from EL2 and EL1.

0b00  When MDCR_EL3.NSTBE == 0b0:
Trace Buffer owning Security state is Secure state. If TraceBufferEnabled () == TRUE, tracing is prohibited in Non-secure and Realm state. Accesses to Trace Buffer control registers at EL2 and EL1 generate Trap exceptions to EL3.
When MDCR_EL3.NSTBE == 0b1: Reserved.

0b01  When MDCR_EL3.NSTBE == 0b0:
Trace Buffer owning Security state is Secure state. If TraceBufferEnabled () == TRUE, tracing is prohibited in Non-secure and Realm state. Accesses to Trace Buffer control registers at EL2 and EL1 in Non-secure and Realm state generate Trap exceptions to EL3.
When MDCR_EL3.NSTBE == 0b1: Reserved.

0b10  When MDCR_EL3.NSTBE == 0b0:
Trace Buffer owning Security state is Non-secure state. If TraceBufferEnabled () == TRUE, tracing is prohibited in Secure and Realm state. Accesses to Trace Buffer control registers at EL2 and EL1 generate Trap exceptions to EL3.
When MDCR_EL3.NSTBE == 0b1:
Trace Buffer owning Security state is Realm state. If TraceBufferEnabled () == TRUE, tracing is prohibited in Non-secure and Secure state. Accesses to Trace Buffer control registers at EL2 and EL1 generate Trap exceptions to EL3.

0b11  When MDCR_EL3.NSTBE == 0b0:
Trace Buffer owning Security state is Non-secure state. If TraceBufferEnabled () == TRUE, tracing is prohibited in Secure and Realm state. Accesses to Trace Buffer control registers at EL2 and EL1 in Secure and Realm state generate Trap exceptions to EL3.
When MDCR_EL3.NSTBE == 0b1:
Trace Buffer owning Security state is Realm state. If TraceBufferEnabled () == TRUE, tracing is prohibited in Non-secure and Secure state. Accesses to Trace Buffer control registers at EL2 and EL1 in Non-secure and Secure state generate Trap exceptions to EL3.

The Trace Buffer control registers trapped by this control are: TRBBASER_EL1, TRBLIMITR_EL1, TRBMAR_EL1, TRBPTR_EL1, TRBSR_EL1, and TRBTRG_EL1.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_TRBE is implemented and FEAT_RME is not implemented:

Non-secure Trace Buffer. Controls the owning translation regime and accesses to Trace Buffer control registers from EL2 and EL1.

0b00  Trace Buffer owning Security state is Secure state. If TraceBufferEnabled () == TRUE, tracing is prohibited in Non-secure state. Accesses to Trace Buffer control registers at EL2 and EL1 generate Trap exceptions to EL3.

0b01  Trace Buffer owning Security state is Secure state. If TraceBufferEnabled () == TRUE, tracing is prohibited in Non-secure state. Accesses to Trace Buffer control registers at EL2 and EL1 in Non-secure state generate Trap exceptions to EL3.

0b10  Trace Buffer owning Security state is Non-secure state. If TraceBufferEnabled () == TRUE, tracing is prohibited in Secure state. Accesses to Trace Buffer control registers at EL2 and EL1 generate Trap exceptions to EL3.

0b11  Trace Buffer owning Security state is Non-secure state. If TraceBufferEnabled () == TRUE, tracing is prohibited in Secure state. Accesses to Trace Buffer control registers at EL2 and EL1 in Secure state generate Trap exceptions to EL3.
The Trace Buffer control registers trapped by this control are: TRBBASER_EL1, TRBLIMITR_EL1, TRBMAR_EL1, TRBPTR_EL1, TRBSR_EL1, and TRBTRG_EL1.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 1, then the Effective value of this field is 0b1.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 0b0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

SCCD, bit [23]

When **FEAT_PMUv3p5** is implemented:
Secure Cycle Counter Disable. Prohibits PMCCNTR_EL0 from counting in Secure state.

- 0b0 Cycle counting by PMCCNTR_EL0 is not affected by this mechanism.
- 0b1 Cycle counting by PMCCNTR_EL0 is prohibited in Secure state.

This field does not affect the CPU_CYCLES event or any other event that counts cycles.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**Otherwise:**
Reserved, RES0.

ETAD, bit [22]

When **FEAT_RME** is implemented, external debugger access to the trace unit registers is implemented and **FEAT_TRBE** is implemented:
External Trace Access Disable. Together with MDCR_EL3.ETADE, controls access to trace unit registers by an external debugger.

<table>
<thead>
<tr>
<th>ETADE</th>
<th>ETAD</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b0</td>
<td>Access to trace unit registers by an external debugger is permitted.</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>Root and Secure access to trace unit registers by an external debugger is permitted. Realm and Non-secure access to trace unit registers by an external debugger is not permitted.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>Root and Realm access to trace unit registers by an external debugger is permitted. Secure and Non-secure access to trace unit registers by an external debugger is not permitted.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>Root access to trace unit registers by an external debugger is permitted. Secure, Non-secure, and Realm access to trace unit registers by an external debugger is not permitted.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

When **external debugger access to the trace unit registers is implemented and **FEAT_TRBE** is implemented:**
External Trace Access Disable. Controls Non-secure access to trace unit registers by an external debugger.

- 0b0 Non-secure accesses from an external debugger to trace unit are allowed.
- 0b1 Non-secure accesses from an external debugger to some trace unit registers are prohibited. See individual registers for the effect of this field.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 1.
The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**EPMAD, bit [21]**

*When FEAT_RME is implemented, FEAT_PMUv3 is implemented and the Performance Monitors Extension supports external debug interface accesses:*

External Performance Monitors Access Disable. Together with MDCR_EL3.EPMADE, controls access to Performance Monitor registers by an external debugger.

<table>
<thead>
<tr>
<th>EPMAD</th>
<th>EPMAD</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b0</td>
<td>Access to Performance Monitor registers by an external debugger is permitted.</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>Root and Secure access to Performance Monitor registers by an external debugger is permitted. Realm and Non-secure access to Performance Monitor registers by an external debugger is not permitted.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>Root and Realm access to Performance Monitor registers by an external debugger is permitted. Secure and Non-secure access to Performance Monitor registers by an external debugger is not permitted.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>Root access to Performance Monitor registers by an external debugger is permitted. Secure, Non-secure, and Realm access to Performance Monitor registers by an external debugger is not permitted.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

*When FEAT_Debugv8p4 is implemented, FEAT_PMUv3 is implemented and the Performance Monitors Extension supports external debug interface accesses:*

External Performance Monitors Non-secure Access Disable. Controls Non-secure access to Performance Monitor registers by an external debugger.

- 0b0: Non-secure access to Performance Monitor registers from external debugger is permitted.
- 0b1: Non-secure access to Performance Monitor registers from external debugger is not permitted.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0b0, then the Effective value of this bit is 0b1.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

*When FEAT_PMUv3 is implemented and the Performance Monitors Extension supports external debug interface accesses:*

External Performance Monitors Access Disable. Controls access to Performance Monitor registers by an external debugger.

- 0b0: Access to Performance Monitor registers from external debugger is permitted.
- 0b1: Access to Performance Monitor registers from external debugger is not permitted, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0b0, then the Effective value of this bit is 0b1.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.
Otherwise:

Reserved, RES0.

EDAD, bit [20]

When FEAT_RME is implemented:

External Debug Access Disable. Together with MDCR_EL3.EDADE, controls access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger.

<table>
<thead>
<tr>
<th>EDADE</th>
<th>EDAD</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b0</td>
<td>Access to Debug registers by an external debugger is permitted.</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>Root and Secure access to Debug registers by an external debugger is permitted. Realm and Non-secure access to Debug registers by an external debugger is not permitted.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>Root and Realm access to Debug registers by an external debugger is permitted. Secure and Non-secure access to Debug registers by an external debugger is not permitted.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>Root access to Debug registers by an external debugger is permitted. Secure, Non-secure, and Realm access to Debug registers by an external debugger is not permitted.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

When FEAT_Debugv8p4 is implemented:

External Debug Non-secure Access Disable. Controls Non-secure access to breakpoint, watchpoint, and OSLAR_EL1 registers by an external debugger.

0b0 Non-secure access to debug registers from external debugger is permitted.

0b1 Non-secure access to breakpoint and watchpoint registers, and OSLAR_EL1 from external debugger is not permitted.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0b0, then the Effective value of this field is 0b1.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

When FEAT_Debugv8p2 is implemented:

External Debug Access Disable. Controls access to breakpoint, watchpoint, and OSLAR_EL1 registers by an external debugger.

0b0 Access to debug registers, and to OSLAR_EL1 from external debugger is permitted.

0b1 Access to breakpoint and watchpoint registers, and to OSLAR_EL1 from external debugger is not permitted, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0b0, then the Effective value of this field is 0b1.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

Otherwise:

External Debug Access disable. Controls access to breakpoint, watchpoint, and optionally OSLAR_EL1 registers by an external debugger.

0b0 Access to debug registers from external debugger is permitted.

0b1 Access to breakpoint and watchpoint registers from an external debugger is not permitted, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

It is IMPLEMENTATION DEFINED whether access to the OSLAR_EL1 register from an external debugger is permitted or not permitted.
If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0b0, then the Effective value of this field is 0b1.

The reset behavior of this field is:
• On a Warm reset, this field resets to 0.

TTRF, bit [19]

When FEAT_TRF is implemented:
Trap Trace Filter controls. Traps use of the Trace Filter control registers at EL2 and EL1 to EL3.
The Trace Filter registers trapped by this control are:
• TRFCR_EL2, TRFCR_EL12, TRFCR_EL1, reported using EC syndrome value 0x18.
• HTRFCR and TRFCR, reported using EC syndrome value 0x03.
0b0 Accesses to Trace Filter registers at EL2 and EL1 are not affected by this bit.
0b1 Accesses to Trace Filter registers at EL2 and EL1 generate a Trap exception to EL3, unless the access generates a higher priority exception.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

STE, bit [18]

When FEAT_TRF is implemented:
Secure Trace enable. Enables tracing in Secure state.
0b0 Trace prohibited in Secure state unless overridden by the IMPLEMENTATION DEFINED authentication interface.
0b1 Trace in Secure state is not affected by this bit.

This bit also controls the level of authentication required by an external debugger to enable external tracing. See Register controls to enable self-hosted trace on page G3-9104.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0b0, the Effective value of this bit is 0b1.

The reset behavior of this field is:
• On a Warm reset, this field resets to 0.

Otherwise:
Reserved, RES0.

SPME, bit [17]

When FEAT_PMUv3 is implemented and FEAT_PMUv3p7 is implemented:
Secure Performance Monitors Enable. Controls event counting in Secure state and EL3.
0b0 When MDCR_EL3.MPMX == 0: Event counting is prohibited in Secure state. If PMCR_EL0.DP is 1, PMCCNTR_EL0 is disabled in Secure state. Otherwise, PMCCNTR_EL0 is not affected by this mechanism.
0b1 When MDCR_EL3.MPMX == 0: Event counting and PMCCNTR_EL0 are not affected by this mechanism.

When MDCR_EL3.MPMX is 0, this field affects the operation of all event counters in Secure state, and if PMCR_EL0.DP is 1, the operation of PMCCNTR_EL0 in Secure state.

When MDCR_EL3.MPMX is 1, this field affects the operation of event counters at EL3 only, and if PMCR_EL0.DP is 1, the operation of PMCCNTR_EL0 at EL3 only. See MDCR_EL3.MPMX for more information.

When PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.
If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**When FEAT_PMUv3 is implemented and FEAT_Debugv8p2 is implemented:**
Secure Performance Monitors Enable. Controls event counting in Secure state.
0b0 Event counting is prohibited in Secure state. If PMCR_EL0.DP is 1, PMCCNTR_EL0 is disabled in Secure state. Otherwise, PMCCNTR_EL0 is not affected by this mechanism.
0b1 Event counting and PMCCNTR_EL0 are not affected by this mechanism.
This field affects the operation of all event counters in Secure state, and if PMCR_EL0.DP is 1, the operation of PMCCNTR_EL0 in Secure state. When PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.
If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 1.
The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**When FEAT_PMUv3 is implemented:**
Secure Performance Monitors Enable. Controls event counting in Secure state.
0b0 If ExternalSecureNoninvasiveDebugEnabled() is FALSE, event counting is prohibited in Secure state, and if PMCR_EL0.DP is 1, PMCCNTR_EL0 is disabled in Secure state.
0b1 Event counting and PMCCNTR_EL0 are not affected by this mechanism.
If ExternalSecureNoninvasiveDebugEnabled() is TRUE, the event counters and PMCCNTR_EL0 are not affected by this field.
Otherwise, this field affects the operation of all event counters in Secure state, and if PMCR_EL0.DP is 1, the operation of PMCCNTR_EL0 in Secure state. When PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.
If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 1.
The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**SDD, bit [16]**
AArch64 Secure Self-hosted invasive debug disable. Disables Software debug exceptions in Secure state, other than Breakpoint Instruction exceptions.
0b0 Debug exceptions in Secure state are not affected by this bit.
0b1 Debug exceptions, other than Breakpoint Instruction exceptions, are disabled from all Exception levels in Secure state.
The SDD bit is ignored unless both of the following are true:
- The PE is in Secure state.
- The Effective value of SCR_EL3.RW is 0b1.
If Secure EL2 is implemented and enabled, and Secure EL1 is using AArch32, then:
- If debug exceptions from Secure EL1 are enabled, debug exceptions from Secure EL0 are also enabled.
- Otherwise, debug exceptions from Secure EL0 are enabled only if the value of SDER32_EL3.SUIDEN is 0b1.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**SPD32, bits [15:14]**

*When EL1 is capable of using AArch32:*

AArch32 Secure self-hosted privileged debug. Enables or disables debug exceptions from Secure EL1 using AArch32, other than Breakpoint Instruction exceptions.

- **0b00** Legacy mode. Debug exceptions from Secure EL1 are enabled by the IMPLEMENTATION DEFINED authentication interface.
- **0b10** Secure privileged debug disabled. Debug exceptions from Secure EL1 are disabled.
- **0b11** Secure privileged debug enabled. Debug exceptions from Secure EL1 are enabled.

Other values are reserved, and have the CONstrained UNPREDICTABLE behavior that they must have the same behavior as **0b00**. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

This field has no effect on Breakpoint Instruction exceptions. These are always enabled.

This field is ignored unless both of the following are true:
• The PE is in Secure state.
• The Effective value of SCR_EL3.RW is **0b0**.

If Secure EL1 is using AArch32, then:
• If debug exceptions from Secure EL1 are enabled, then debug exceptions from Secure EL0 are also enabled.
• Otherwise, debug exceptions from Secure EL0 are enabled only if the value of SDER32_EL3.SUIDEN is **0b1**.

If EL3 is not implemented and the Effective value of SCR_EL3.NS is **0b0**, then the Effective value of this field is **0b11**.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, **RES0**.

**NSPB, bits [13:12]**

*When FEAT_SPE is implemented and FEAT_RME is implemented:*

Non-secure Profiling Buffer. Together with MDCR_EL3.NSPBE, controls the owning translation regime and accesses to Statistical Profiling and Profiling Buffer control registers.

- **0b00** When MDCR_EL3.NSPBE == **0b0**:
  Profiling Buffer uses Secure Virtual Addresses. Statistical Profiling enabled in Secure state and disabled in Non-secure and Realm state. Accesses to Statistical Profiling and Profiling Buffer control registers at EL2 and EL1 in all Security states generate Trap exceptions to EL3.
  When MDCR_EL3.NSPBE == **0b1**: Reserved.

- **0b01** When MDCR_EL3.NSPBE == **0b0**:
  Profiling Buffer uses Secure Virtual Addresses. Statistical Profiling enabled in Secure state and disabled in Non-secure and Realm state. Accesses to Statistical Profiling and Profiling Buffer control registers at EL2 and EL1 in Non-secure and Realm states generate Trap exceptions to EL3.
  When MDCR_EL3.NSPBE == **0b1**: Reserved.

- **0b10** When MDCR_EL3.NSPBE == **0b0**:
Profiling Buffer uses Non-secure Virtual Addresses. Statistical Profiling enabled in Non-secure state and disabled in Secure and Realm state. Accesses to Statistical Profiling and Profiling Buffer control registers at EL2 and EL1 in all Security states generate Trap exceptions to EL3.

When MDCR_EL3.NSPBE == 0b1:
Profiling Buffer uses Realm Virtual Addresses. Statistical Profiling enabled in Realm state and disabled in Non-secure and Secure state. Accesses to Statistical Profiling and Profiling Buffer control registers at EL2 and EL1 in all Security states generate Trap exceptions to EL3.

When MDCR_EL3.NSPBE == 0b0:
Profiling Buffer uses Non-secure Virtual Addresses. Statistical Profiling enabled in Non-secure state and disabled in Secure and Realm state. Accesses to Statistical Profiling and Profiling Buffer control registers at EL2 and EL1 in Non-secure and Secure states generate Trap exceptions to EL3.

The Statistical Profiling and Profiling Buffer control registers trapped by this control are:

- PMBLIMITR_EL1, PMBPTR_EL1, PMBSR_EL1, PMSCR_EL1, PMSCR_EL2, PMSCR_EL12, PMSEVFR_EL1, PMSFCR_EL1, PMSICR_EL1, PMSIDR_EL1, PMSIRR_EL1, and PMSLATFR_EL1.
- If FEAT_SPEv1p2 is implemented, PMSNEVFR_EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SPE is implemented and FEAT_RME is not implemented:
Non-secure Profiling Buffer. Controls the owning translation regime and accesses to Statistical Profiling and Profiling Buffer control registers.

When EL3 is not implemented and the Effective value of SCR_EL3.NS is 1, then the Effective value of this field is 0b11.
If EL3 is not implemented and the Effective value of SCR_EL3.NS is 0, then the Effective value of this field is 0b01.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

NSPBE, bit [11]

When FEAT_RME is implemented:
Non-secure Profiling Buffer Extended. Together with MDCR_EL3.NSPB, controls the owning translation regime and accesses to Statistical Profiling and Profiling Buffer control registers.

For a description of the values derived by evaluating NSPB and NSPBE together, see MDCR_EL3.NSPB.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

TDOSA, bit [10]

When FEAT_DoubleLock is implemented:
Trap debug OS-related register access. Traps EL2 and EL1 System register accesses to the powerdown debug registers to EL3.

Accesses to the registers are trapped as follows:
• Accesses from AArch64 state, OSLAR_EL1, OSLSR_EL1, OSDLR_EL1, DBGPRCR_EL1, and any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit, are trapped to EL3 and reported using EC syndrome value 0x18.
• Accesses using MCR or MRC to DBGOSLAR, DBGOSLSR, DBGOSDLR, and DBGPRCR, are trapped to EL3 and reported using EC syndrome value 0x05.
• Accesses to any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

0b0 This control does not cause any instructions to be trapped.
0b1 EL2 and EL1 System register accesses to the powerdown debug registers are trapped to EL3, unless it is trapped by HDCR.TDOSA or MDCR_EL2.TDOSA.

Note

The powerdown debug registers are not accessible at EL0.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Trap debug OS-related register access. Traps EL2 and EL1 System register accesses to the powerdown debug registers to EL3.

The following registers are affected by this trap:
• AArch64: OSLAR_EL1, OSLSR_EL1, and DBGPRCR_EL1.
• AArch32: DBGOSLAR, DBGOSLSR, and DBGPRCR.
• AArch64 and AArch32: Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.
• It is IMPLEMENTATION DEFINED whether accesses to OSDLR_EL1 and DBGOSDLR are trapped.

0b0  This control does not cause any instructions to be trapped.
0b1  EL2 and EL1 System register accesses to the powerdown debug registers are trapped to EL3, unless it is trapped by HDCR.TDOSA or MDCR_EL2.TDOSA.

--- Note ---
The powerdown debug registers are not accessible at EL0.

---
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDA, bit [9]

Trap Debug Access. Traps EL2, EL1, and EL0 System register accesses to those debug System registers that cannot be trapped using the MDCR_EL3.TDOSA field.

Accesses to the debug registers are trapped as follows:
• In AArch64 state, the following registers are trapped to EL3 and reported using EC syndrome value 0x18:
  — DBGBVR<n>_EL1, DBGBCR<n>_EL1, DBGWVR<n>_EL1, DBGWCR<n>_EL1, DBGCLAIMSET_EL1, DBGCLAIMCLR_EL1, DBGAUTHSTATUS_EL1, DBGVCR32_EL2.
  — AArch64: MDCR_EL2, MDRAR_EL1, MDCCSR_EL0, MDCCINT_EL1, MDSCR_EL1, OSDTRRX_EL1, OSDTRTX_EL1, OSECCR_EL1.
• In AArch32 state, SDER is trapped to EL3 and reported using EC syndrome value 0x03.
• In AArch32 state, accesses using MCR or MRC to the following registers are reported using EC syndrome value 0x05, accesses using MCRR or MRRC are reported using EC syndrome value 0x0C:
  — HDCR, DBGDRAR, DBGDSAR, DBGIDIR, DBGDCINT, DBGWFAR, DBGVCR, DBGVR<n>, DBGBCR<n>, DBGXVR<n>, DBGWCR<n>, DBGWVR<n>.
  — DBGCLAIMSET, DBGCLAIMCLR, DBGAUTHSTATUS, DBGDEVID, DBGDEVID1, DBGDEVID2, DBGSECCR.
• In AArch32 state, STC accesses to DBGDTRRXint and LDC accesses to DBGDTRTXint are reported using EC syndrome value 0x06.
• When not in Debug state, the following registers are also trapped to EL3:
  — AArch64 accesses to DBGDTR_EL0, DBGDTRRX_EL0, and DBGDTRTX_EL0, reported using EC syndrome value 0x18.
  — AArch32 accesses using MCR or MRC to DBGDTRRXint and DBGDTRTXint, reported using EC syndrome value 0x05.

0b0  This control does not cause any instructions to be trapped.
0b1  EL0, EL1, and EL2 accesses to the debug registers, other than the registers that can be trapped by MDCR_EL3.TDOSA, are trapped to EL3, from any Security state and both Execution states, unless it is trapped by DBGDSCRx.UDCCdis, MDSCR_EL1.TDCC, HDCR.TDA or MDCR_EL2.TDA.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:7]

Reserved, RES0.
TPM, bit [6]

When FEAT_PMUv3 is implemented:

Trap Performance Monitor register accesses. Accesses to all Performance Monitor registers from EL0, EL1, and EL2 to EL3, from any Security state and both Execution states are trapped as follows:

- In AArch64 state, accesses to the following registers are trapped to EL3 and are reported using EC syndrome value 0x18:
  - PMCR_EL0, PMCINTENSET_EL0, PMCINTENCLR_EL0, PMOVSCCLR_EL0, PMSWINC_EL0, PMSEL_EL0, PMCEID0_EL0, PMCEID1_EL0, PMCCTR_EL0, PMXEVTYPER_EL0, PMXEVCTR_EL0, PMUSERENR_EL0, PMINTENSET_EL1, PMINTENCLR_EL1, PMINTENCLR_EL0, PMINTENSET_EL0.
  - If FEAT_PMUv3p4 is implemented, PMMIR_EL1.

- In AArch32 state, accesses using MCR or MRC to the following registers are reported using EC syndrome value 0x03, accesses using MCRR or MRRC are reported using EC syndrome value 0x04:
  - PMCR, PMCINTENSET, PMCINTENCLR, PMOVSR, PMSWINC, PMSEL, PMCEID0, PMCEID1, PMCCTR, PMXEVTYPER, PMXEVCTR, PMUSERENR, PMINTENSET, PMINTENCLR, PMOVSET, PMEVCNTR<n>, PMEVTYPER<n>, PMCCFILTR.
  - If FEAT_PMUv3p1 is implemented, PMCEID2, and PMCEID3.
  - If FEAT_PMUv3p4 is implemented, PMMIR.

0b0 This control does not cause any instructions to be trapped.
0b1 EL2, EL1, and EL0 System register accesses to all Performance Monitor registers are trapped to EL3, unless it is trapped by HDCR.TPM or MDCR_EL2.TPM.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [5]

Reserved, RES0.

EDADE, bit [4]

When FEAT_RME is implemented:

External Debug Access Disable Extended. Together with MDCR_EL3.EDAD, controls access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger.

For a description of the values derived by evaluating EDAD and EDADE together, see MDCR_EL3.EDAD.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

Otherwise:

Reserved, RES0.

ETADE, bit [3]

When FEAT_RME is implemented, external debugger access to the trace unit registers is implemented and FEAT_TRBE is implemented:

External Trace Access Disable Extended. Together with MDCR_EL3.ETAD, controls access to trace unit registers by an external debugger.

For a description of the values derived by evaluating ETAD and ETADE together, see MDCR_EL3.ETAD.
The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**EPMADE, bit [2]**

When FEAT_RME is implemented, FEAT_PMUv3 is implemented and the Performance Monitors Extension supports external debug interface accesses:
External Performance Monitors Access Disable Extended. Together with MDCR_EL3.EPMAD, controls access to Performance Monitor registers by an external debugger.
For a description of the values derived by evaluating EPMAD and EPMADE together, see MDCR_EL3.EPMAD.
The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**Bit [1]**
Reserved, RES0.

**RLTE, bit [0]**

When FEAT_RME is implemented and FEAT_TRF is implemented:
Realm Trace enable. Enables tracing in Realm state.

0b0 Trace prohibited in Realm state, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

0b1 Trace in Realm state is not affected by this bit.

This bit also controls the level of authentication that is required by an external debugger to enable external tracing.
The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**Accessing MDCR_EL3**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, MDCR_EL3**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0001</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  X[t, 64] = MDCR_EL3;
MSR MDCR_EL3, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b11</td>
<td>0b0001</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  MDCR_EL3 = X[t, 64];
D17.3.19 MDRAR_EL1, Monitor Debug ROM Address Register

The MDRAR_EL1 characteristics are:

Purpose

Defines the base physical address of a 4KB-aligned memory-mapped debug component, usually a ROM table that locates and describes the memory-mapped debug components in the system. Armv8 deprecates any use of this register.

Configurations

AArch64 System register MDRAR_EL1 bits [63:0] are architecturally mapped to AArch32 System register DBGDRAR[63:0].

Attributes

MDRAR_EL1 is a 64-bit register.

Field descriptions

Bits [63:52]

Reserved, RES0.

ROMADDR, bits [51:12]

ROMADDR encoding when FEAT_LPA is implemented and MDRAR_EL1.Valid != 0b00

ROMADDR, bits [39:0]

Bits [39:0] of the ROM table physical address.

Bits [51:12] of the ROM table physical address are zero.

For implementations with fewer than 52 physical address bits, the corresponding upper bits of this field are RES0.

In an implementation that includes EL3, ROMADDR is an address in Non-secure PA space. It is IMPLEMENTATION DEFINED whether the ROM table is also accessible in Secure PA space. If FEAT_RME is implemented, it is IMPLEMENTATION DEFINED whether the ROM table is also accessible in the Root or Realm PA spaces.

Arm strongly recommends that bits ROMADDR[PAsize-1]:32] are zero in any system where the implementation only supports execution in AArch32 state.
ROMADDR encoding when FEAT_LPA is not implemented and MDRAR_EL1.Valid != 0b00

Bits [39:36]
Reserved, RES0.

ROMADDR, bits [35:0]
Bits [39:12] of the ROM table physical address.
Bits [11:0] of the ROM table physical address are zero.
For implementations with fewer than 48 physical address bits, the corresponding upper bits of this field are RES0.
In an implementation that includes EL3, ROMADDR is an address in Non-secure PA space. It is IMPLEMENTATION DEFINED whether the ROM table is also accessible in Secure PA space. If FEAT_RME is implemented, it is IMPLEMENTATION DEFINED whether the ROM table is also accessible in Root or Realm PA spaces.
Arm strongly recommends that bits ROMADDR[(PAsize-1):32] are zero in any system where the implementation only supports execution in AArch32 state.

ROMADDR encoding when MDRAR_EL1.Valid == 0b00

Bits [39:0]
Reserved, UNKNOWN.

Bits [11:2]
Reserved, RES0.

Valid, bits [1:0]
This field indicates whether the ROM Table address is valid.
0b00 ROM Table address is not valid. Software must ignore ROMADDR.
0b11 ROM Table address is valid.
Other values are reserved.
Arm recommends implementations set this field to zero.

Accessing MDRAR_EL1

Accesses to this register use the following encodings in the System register encoding space:
MRS <x_t>, MDRAR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elsif EL2Enabled() && MDCR_EL2.<TDE,TDRA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end else
        X[t, 64] = MDRAR_EL1;
    elsif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
            UNDEFINED;
        elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end else
            X[t, 64] = MDRAR_EL1;
        elsif PSTATE.EL == EL3 then
            X[t, 64] = MDRAR_EL1;
        end if
    end if

D17.3.20 MDSCR_EL1, Monitor Debug System Control Register

The MDSCR_EL1 characteristics are:

**Purpose**

Main control register for the debug implementation.

**Configurations**

AArch64 System register MDSCR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGDSCRExt[31:0].

AArch64 System register MDSCR_EL1 bit [15] is architecturally mapped to AArch32 System register DBGDSCRInt[15].

AArch64 System register MDSCR_EL1 bit [12] is architecturally mapped to AArch32 System register DBGDSCRInt[12].

AArch64 System register MDSCR_EL1 bits [5:2] are architecturally mapped to AArch32 System register DBGDSCRInt[5:2].

**Attributes**

MDSCR_EL1 is a 64-bit register.

**Field descriptions**

Bits [63:32]

Reserved, RES0.

TFO, bit [31]

*When FEAT_TRF is implemented:*

Trace Filter override. Used for save/restore of EDSCR.TFO.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TFO. Reads and writes of this bit are indirect accesses to EDSCR.TFO.

Accessing this field has the following behavior:

* When OSLSR_EL1.OSLK == 1, access to this field is RW.
* When OSLSR_EL1.OSLK == 0, access to this field is RO.

**Otherwise:**

Reserved, RES0.

RXfull, bit [30]

Used for save/restore of EDSCR.RXfull.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.
When OSLR_EL1.OSLK == 1, this bit holds the value of EDSCR.RXfull. Reads and writes of this bit are indirect accesses to EDSCR.RXfull.

The reset behavior of this field is:
- The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:
- When OSLR_EL1.OSLK == 1, access to this field is RW.
- When OSLR_EL1.OSLK == 0, access to this field is RO.

TXfull, bit [29]

Used for save/restore of EDSCR.TXfull.

When OSLR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLR_EL1.OSLK == 1, this bit holds the value of EDSCR.TXfull. Reads and writes of this bit are indirect accesses to EDSCR.TXfull.

The reset behavior of this field is:
- The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:
- When OSLR_EL1.OSLK == 1, access to this field is RW.
- When OSLR_EL1.OSLK == 0, access to this field is RO.

Bit [28]

Reserved, RES0.

RXO, bit [27]

Used for save/restore of EDSCR.RXO.

When OSLR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLR_EL1.OSLK == 1, this bit holds the value of EDSCR.RXO. Reads and writes of this bit are indirect accesses to EDSCR.RXO.

When OSLR_EL1.OSLK == 1, if bits [27,6] of the value written to MDSCR_EL1 are \{1,0\}, that is, the RXO bit is 1 and the ERR bit is 0, the PE sets EDSCR.{RXO,ERR} to UNKNOWN values.

The reset behavior of this field is:
- The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:
- When OSLR_EL1.OSLK == 1, access to this field is RW.
- When OSLR_EL1.OSLK == 0, access to this field is RO.

TXU, bit [26]

Used for save/restore of EDSCR.TXU.

When OSLR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLR_EL1.OSLK == 1, this bit holds the value of EDSCR.TXU. Reads and writes of this bit are indirect accesses to EDSCR.TXU.

When OSLR_EL1.OSLK == 1, if bits [26,6] of the value written to MDSCR_EL1 are \{1,0\}, that is, the TXU bit is 1 and the ERR bit is 0, the PE sets EDSCR.{TXU,ERR} to UNKNOWN values.

The reset behavior of this field is:
- The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:
- When OSLR_EL1.OSLK == 1, access to this field is RW.
- When OSLR_EL1.OSLK == 0, access to this field is RO.

Bits [25:24]

Reserved, RES0.
INTdis, bits [23:22]
Used for save/restore of EDSCR.INTdis.
When OSLSR_EL1.OSLK == 0, and software must treat this bit as UNK/SBZP.
When OSLSR_EL1.OSLK == 1, this field holds the value of EDSCR.INTdis. Reads and writes of this field are indirect accesses to EDSCR.INTdis.
The reset behavior of this field is:
• The architected behavior of this field determines the value it returns after a reset.
Accessing this field has the following behavior:
• When OSLSR_EL1.OSLK == 1, access to this field is RW.
• When OSLSR_EL1.OSLK == 0, access to this field is RO.

TDA, bit [21]
Used for save/restore of EDSCR.TDA.
When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.
When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TDA. Reads and writes of this bit are indirect accesses to EDSCR.TDA.
The reset behavior of this field is:
• The architected behavior of this field determines the value it returns after a reset.
Accessing this field has the following behavior:
• When OSLSR_EL1.OSLK == 1, access to this field is RW.
• When OSLSR_EL1.OSLK == 0, access to this field is RO.

Bit [20]
Reserved, RES0.

SC2, bit [19]
When FEAT_PCSRv8 is implemented, FEAT_VHE is implemented and FEAT_PCSRv8p2 is not implemented:
Used for save/restore of EDSCR.SC2.
When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.
When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.SC2. Reads and writes of this bit are indirect accesses to EDSCR.SC2.
Accessing this field has the following behavior:
• When OSLSR_EL1.OSLK == 1, access to this field is RW.
• When OSLSR_EL1.OSLK == 0, access to this field is RO.
Otherwise:
Reserved, RES0.

Bits [18:16]
Reserved, RAZ/WI.
Hardware must implement this field as RAZ/WI. Software must not rely on the register reading as zero, and must use a read-modify-write sequence to write to the register.

MDE, bit [15]
Monitor debug events. Enable Breakpoint, Watchpoint, and Vector Catch exceptions.
0b0 Breakpoint, Watchpoint, and Vector Catch exceptions disabled.
0b1 Breakpoint, Watchpoint, and Vector Catch exceptions enabled.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
HDE, bit [14]

Used for save/restore of EDSCR.HDE.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.HDE. Reads and writes of this bit are indirect accesses to EDSCR.HDE.

The reset behavior of this field is:
- The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:
- When OSLSR_EL1.OSLK == 1, access to this field is RW.
- When OSLSR_EL1.OSLK == 0, access to this field is RO.

KDE, bit [13]

Local (kernel) debug enable. If EL_D is using AArch64, enable debug exceptions within EL_D.

Permitted values are:
- 0b0: Debug exceptions, other than Breakpoint Instruction exceptions, disabled within EL_D.
- 0b1: All debug exceptions enabled within EL_D.
- RES0 if EL_D is using AArch32.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDCC, bit [12]

Traps EL0 accesses to the Debug Communication Channel (DCC) registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states, as follows:
- In AArch64 state, MRS or MSR accesses to the following DCC registers are trapped, reported using EC syndrome value 0x18:
  - MDCCSR_EL0.
  - If not in Debug state, DBGDTR_EL0, DBGDTRTX_EL0, and DBGDTRRX_EL0.
- In AArch32 state, MRC or MCR accesses to the following registers are trapped, reported using EC syndrome value 0x85:
  - DBGDSCRint, DBGDIDR, DBGDSAR, DBGDRAR.
  - If not in Debug state, DBGDTRXint, and DBGDTRXXint.
- In AArch32 state, LDC access to DBGDTRXXint and STC access to DBGDTRXXint are trapped, reported using EC syndrome value 0x06.
- In AArch32 state, MRRC accesses to DBGDSAR and DBGDRAR are trapped, reported using EC syndrome value 0x0C.

0b0: This control does not cause any instructions to be trapped.
0b1: EL0 using AArch64: EL0 accesses to the AArch64 DCC registers are trapped.
EL0 using AArch32: EL0 accesses to the AArch32 DCC registers are trapped.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:7]

Reserved, RES0.

ERR, bit [6]

Used for save/restore of EDSCR.ERR.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.
When `OSLSR_EL1.OSLK == 1`, this bit holds the value of `EDSCR.ERR`. Reads and writes of this bit are indirect accesses to `EDSCR.ERR`.

The reset behavior of this field is:

- The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

- When `OSLSR_EL1.OSLK == 1`, access to this field is RW.
- When `OSLSR_EL1.OSLK == 0`, access to this field is RO.

**Bits [5:1]**

Reserved, RES0.

**SS, bit [0]**

Software step control bit. If EL_D is using AArch64, enable Software step. Permitted values are:

- 0b0 Software step disabled
- 0b1 Software step enabled.

RES0 if EL_D is using AArch32.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing MDSCR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, MDSCR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() & (HaveEL(EL3) | SCR_EL3.FGTEn == '1') & HDGRTR_EL2.MDSCR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) & MDCR_EL3.TDA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() & HCR_EL2.<NV2,NV> == '11' then
      X[t, 64] = NVMem[0x158];
    else
      X[t, 64] = MDSCR_EL1;
  elsif PSTATE.EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TDA == '1' then
      UNDEFINED;
    elsif HaveEL(EL3) & MDCR_EL3.TDA == '1' then
      if Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = MDSCR_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = MDSCR_EL1;

**MSR MDSCR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.MDSCR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    end
  end
elsif PSTATE.EL == EL3 then
  MDSCR_EL1 = X[t, 64];
D17.3.21 OSDLR_EL1, OS Double Lock Register

The OSDLR_EL1 characteristics are:

**Purpose**

Used to control the OS Double Lock.

**Configurations**

AArch64 System register OSDLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGOSDLR[31:0].

**Attributes**

OSDLR_EL1 is a 64-bit register.

**Field descriptions**

| Bits [63:1] | Reserved, RES0. |
| DLK, bit [0] | Reserved, RES0. |

**When FEAT_DoubleLock is implemented:**

OS Double Lock control bit.

- 0b0 OS Double Lock unlocked.
- 0b1 OS Double Lock locked, if DBGPRCR_EL1.CORENPDRQ (Core no powerdown request) bit is set to 0 and the PE is in Non-debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

**Otherwise:**

Reserved, RAZ/WI.

**Accessing OSDLR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, OSDLR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b001</td>
<td>0b011</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then

UNDEFINED;

elsif PSTATE_EL == EL1 then

if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD = '1'" && MDSCR_EL3.TDOSA == '1' && (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by MDSCR_EL3.TDOSA") then
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" && MDCR_EL3.TDOSA == '1' && (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by MDCR_EL3.TDOSA") then
    UNDEFINED;
elsif EL2Enabled() && MDCR_EL2.<TDE,TDOSA> != '00' && (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by MDCR_EL2.TDOSA") then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && MDCR_EL3.TDOSA == '1' && (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by MDCR_EL3.TDOSA") then
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = OSDLR_EL1;
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" && MDCR_EL3.TDOSA == '1' && (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by MDCR_EL3.TDOSA") then
    UNDEFINED;
elsif EL2Enabled() && MDCR_EL2.<TDE,TDOSA> != '00' && (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by MDCR_EL2.TDOSA") then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && MDCR_EL3.TDOSA == '1' && (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by MDCR_EL3.TDOSA") then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = OSDLR_EL1;
elsif PSTATE_EL == EL3 then
  X[t, 64] = OSDLR_EL1;

MSR OSDLR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0011</td>
<td>0b10</td>
</tr>
</tbody>
</table>
IMPLEMENTATION_DEFINED "Trapped by MDSCR_EL3.TDOSA") then
    if Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        OSDLR_EL1 = X[t, 64];
elsif PSTATE_EL == EL3 then
    OSDLR_EL1 = X[t, 64];
D17.3.22 OSDTRRX_EL1, OS Lock Data Transfer Register, Receive

The OSDTRRX_EL1 characteristics are:

**Purpose**

Used for save and restore of DBGDTRRX_EL0. It is a component of the Debug Communications Channel.

**Configurations**

AArch64 System register OSDTRRX_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGDTRRXext[31:0].

**Attributes**

OSDTRRX_EL1 is a 64-bit register.

**Field descriptions**

Bits [63:32]

Reserved, RES0.

Bits [31:0]

Update DTRRX without side-effect.

Updates to this register update the value in DTRRX and do not change RXfull.

Reads of this register return the last value written to DTRRX and do not change RXfull.

For the full behavior of the Debug Communications Channel, see Chapter H4 The Debug Communication Channel and Instruction Transfer Register.

**Accessing OSDTRRX_EL1**

Arm deprecates reads and writes of OSDTRRX_EL1 when the OS Lock is unlocked.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, OSDTRRX_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then

UNDEFINED;

elsif Halted() && ConstraintUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then

X[t, 64] = OSDTRRX_EL1;

elsif PSTATE_EL == EL1 then

if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDCC == '1' then

UNDEFINED;

elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then

...
UNDEFINED;
elsif EL2Enabled() && MDCR_EL2.TDCC == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && MDCR_EL2.<TDE,TDA> != '00' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
else
  X[t, 64] = OSDTRRX_EL1;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDCC == '1' then
    UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elsif HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
else
  X[t, 64] = OSDTRRX_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = OSDTRRX_EL1;
endif

MSR OSDTRRX_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
  OSDTRRX_EL1 = X[t, 64];
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDCC == '1' then
    UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elsif EL2Enabled() && MDCR_EL2.TDCC == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && MDCR_EL2.<TDE,TDA> != '00' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
else
  X[t, 64] = OSDTRRX_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = OSDTRRX_EL1;
endif
AArch64.SystemAccessTrap(EL3, 0x18);
elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    OSDTRRX_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'' & MDCR_EL3.TDCC == '1' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap
  priority when SDD == '1'' & MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TDDC == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      else
        OSDTRRX_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
  OSDTRRX_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
  OSDTRRX_EL1 = X[t, 64];
D17.3.23 **OSDTRTX_EL1, OS Lock Data Transfer Register, Transmit**

The OSDTRTX_EL1 characteristics are:

**Purpose**

Used for save/restore of DBGDTRTX_EL0. It is a component of the Debug Communications Channel.

**Configurations**

AArch64 System register OSDTRTX_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGDTRTXext[31:0].

**Attributes**

OSDTRTX_EL1 is a 64-bit register.

**Field descriptions**

| Bits [63:32] | Reserved, RES0. |
| Bits [31:0] | Return DTRTX without side-effect. |

Reads of this register return the value in DTRTX and do not change TXfull.
Writes of this register update the value in DTRTX and do not change TXfull.

For the full behavior of the Debug Communications Channel, see Chapter H4 *The Debug Communication Channel and Instruction Transfer Register*.

**Accessing OSDTRTX_EL1**

Arm deprecates reads and writes of OSDTRTX_EL1 when the OS Lock is unlocked.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, OSDTRTX_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif Halted() & ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
  X[t, 64] = OSDTRTX_EL1;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TDCC == '1' then
    UNDEFINED;
  elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TDA == '1' then

---

D17-6496  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  ARM DDI 0487I.a  Non-Confidential
UNDEFINED;
elif EL2Enabled() && MDCR_EL2.TDCC == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() && MDCR_EL2.<TDE,TDA> != '00' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elif HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
elif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = OSDTRTX_EL1;
elif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && MDCR_EL3.TDCC == '1' then
    UNDEFINED;
elif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap
  priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elif HaveEL(EL3) && MDCR_EL3.TDCC == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
elif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = OSDTRTX_EL1;
elif PSTATE.EL == EL3 then
  X[t, 64] = OSDTRTX_EL1;

**MSR OSDTRTX_EL1, <Xt>**

<table>
<thead>
<tr>
<th></th>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0011</td>
<td>0b010</td>
<td></td>
</tr>
</tbody>
</table>
AArch64.SystemAccessTrap(EL3, 0x18); 
else if HaveEL(EL3) && MDCR_EL3.TDA == '1' then 
  if Halted() && EDSCR.SDD == '1' then 
    UNDEFINED; 
  else 
    AArch64.SystemAccessTrap(EL3, 0x18); 
else 
  OSDTRTX_EL1 = X[t, 64]; 
elsif PSTATE_EL == EL2 then 
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDCC == '1' then 
    UNDEFINED; 
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then 
    UNDEFINED; 
  elsif HaveEL(EL3) && MDCR_EL3.TDCC == '1' then 
    if Halted() && EDSCR.SDD == '1' then 
      UNDEFINED; 
    else 
      AArch64.SystemAccessTrap(EL3, 0x18); 
  elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then 
    if Halted() && EDSCR.SDD == '1' then 
      UNDEFINED; 
    else 
      AArch64.SystemAccessTrap(EL3, 0x18); 
else 
  OSDTRTX_EL1 = X[t, 64]; 
elsif PSTATE_EL == EL3 then 
  OSDTRTX_EL1 = X[t, 64] then 
  OSDTRTX_EL1 = X[t, 64];
D17.3.24 OSECCR_EL1, OS Lock Exception Catch Control Register

The OSECCR_EL1 characteristics are:

**Purpose**

Provides a mechanism for an operating system to access the contents of EDECCR that are otherwise invisible to software, so it can save/restore the contents of EDECCR over powerdown on behalf of the external debugger.

**Configurations**

AArch64 System register OSECCR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGOSECCR[31:0].

AArch64 System register OSECCR_EL1 bits [31:0] are architecturally mapped to External register EDECCR[31:0].

If OSLSR_EL1.OSLK == 0, then OSECCR_EL1 returns an UNKNOWN value on reads and ignores writes.

**Attributes**

OSECCR_EL1 is a 64-bit register.

**Field descriptions**

*When OSLSR_EL1.OSLK == 1:*

![Field Diagram]

- **Bits [63:32]**
  - Reserved, RES0.

- **EDECCR, bits [31:0]**
  - Used for save/restore to EDECCR over powerdown.
  - Reads or writes to this field are indirect accesses to EDECCR.

**Accessing OSECCR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, OSECCR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b0110</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.OSECCR_EL1 == '1' then

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & MDCR_EL3.TDA == '1' then
    UNDEFINED;
  else
    MSR OSECCR_EL1, <Xt>
  end if
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & MDCR_EL3.TDA == '1' then
    UNDEFINED;
  else
    MSR OSECCR_EL1, <Xt>
  end if
elsif PSTATE.EL == EL3 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & MDCR_EL3.TDA == '1' then
    UNDEFINED;
  else
    MSR OSECCR_EL1, <Xt>
  end if
else
  MSR OSECCR_EL1, <Xt>
end if
AArch64.SystemAccessTrap(El3, 0x18);
elsif OSLR_EL1.OSLK == '0' then
  return;
else
  OSECCR_EL1 = X[t, 64];
elsif PSTATE_EL == EL3 then
  if OSLR_EL1.OSLK == '0' then
    return;
  else
    OSECCR_EL1 = X[t, 64];
end if;
D17.3.25 **OSLAR_EL1, OS Lock Access Register**

The OSLAR_EL1 characteristics are:

**Purpose**

Used to lock or unlock the OS Lock.

**Configurations**

AArch64 System register OSLAR_EL1 bits [31:0] are architecturally mapped to External register OSLAR_EL1[31:0].

The OS Lock can also be locked or unlocked using DBGOSLAR.

**Attributes**

OSLAR_EL1 is a 64-bit register.

**Field descriptions**

**Bits [63:1]**

Reserved, RES0.

**OSLK, bit [0]**

On writes to OSLAR_EL1, bit[0] is copied to the OS Lock.

Use OSLSR_EL1.OSLK to check the current status of the lock.

**Accessing OSLAR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MSR OSLAR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDOSA == '1' then
        UNDEFINED;
    elseif EL2Enabled() && !HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.OSLAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && MDCR_EL2.<TDE,TDOSA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) && MDCR_EL3.TDOSA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
else
  OSLAR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TDOSA == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TDOSA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    OSLAR_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL3 then
  OSLAR_EL1 = X[t, 64];
D17.3.26 OSLSR_EL1, OS Lock Status Register

The OSLSR_EL1 characteristics are:

**Purpose**

Provides the status of the OS Lock.

**Configurations**

AArch64 System register OSLSR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGOSLR[31:0].

**Attributes**

OSLR_EL1 is a 64-bit register.

**Field descriptions**

![OSLR_EL1 Diagram]

**Bits [63:4]**

Reserved, RES0.

**OSLM, bits [3, 0]**

OS Lock model implemented. Identifies the form of OS save and restore mechanism implemented.

- **0b00** OS Lock not implemented.
- **0b10** OS Lock implemented.

All other values are reserved. In an Armv8 implementation the value 0b00 is not permitted.

The OSLM field is split as follows:

- OSLM[1] is OSLSR_EL1[3].
- OSLM[0] is OSLSR_EL1[0].

**nTT, bit [2]**

Not 32-bit access. This bit is always RAZ. It indicates that a 32-bit access is needed to write the key to the OS Lock Access Register.

**OSLK, bit [1]**

OS Lock Status.

- **0b0** OS Lock unlocked.
- **0b1** OS Lock locked.

The OS Lock is locked and unlocked by writing to the OS Lock Access Register.

The reset behavior of this field is:

- On a Cold reset, this field resets to 1.

**Accessing OSLSR_EL1**

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, OSLSR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR_SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCTR_EL3 TDOSA == '1' then
        UNDEFINED;
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3 FGETn == '1') && HDFGTR_EL2 OSLSR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCTR_EL2 <TDE, TDOSA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCTR_EL3 TDOSA == '1' then
        if Halted() && EDSCR_SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end if
    end if
else
    X[t, 64] = OSLSR_EL1;
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR_SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCTR_EL3 TDOSA == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && MDCTR_EL3 TDOSA == '1' then
        if Halted() && EDSCR_SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end if
    end if
else
    X[t, 64] = OSLSR_EL1;
elsif PSTATE_EL == EL3 then
    X[t, 64] = OSLSR_EL1;
**SDER32_EL2, AArch32 Secure Debug Enable Register**

The SDER32_EL2 characteristics are:

**Purpose**

Allows access to the AArch32 register SDER from Secure EL2 and EL3 only.

**Configurations**

AArch64 System register SDER32_EL2 bits [63:0] are architecturally mapped to AArch64 System register SDER32_EL3[63:0] when EL3 is implemented.

AArch64 System register SDER32_EL2 bits [31:0] are architecturally mapped to AArch32 System register SDER[31:0].

This register is present only when EL2 is implemented, FEAT_SEL2 is implemented and EL1 is capable of using AArch32. Otherwise, direct accesses to SDER32_EL2 are UNDEFINED.

This register is ignored by the PE when one or more of the following are true:

- The PE is in Non-secure state.
- EL1 is using AArch64.

**Attributes**

SDER32_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:2</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>1</td>
<td>SUNIDEN, Secure User Non-Invasive Debug Enable</td>
</tr>
<tr>
<td>0</td>
<td>This bit has no effect on non-invasive debug.</td>
</tr>
<tr>
<td>1</td>
<td>Non-invasive debug is allowed in Secure EL0 using AArch32.</td>
</tr>
<tr>
<td>0</td>
<td>Secure EL1 is using AArch32, the forms of non-invasive debug affected by this control are:</td>
</tr>
<tr>
<td>0</td>
<td>• The PC Sample-based Profiling Extension. See About the PC Sample-based Profiling Extension.</td>
</tr>
<tr>
<td>1</td>
<td>• When SelfHostedTraceEnabled() == FALSE, processor trace.</td>
</tr>
<tr>
<td>0</td>
<td>• When EL3 is implemented, Performance Monitors.</td>
</tr>
<tr>
<td>0</td>
<td>Secure EL1 is using AArch64, this bit has no effect.</td>
</tr>
<tr>
<td>0</td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td>0</td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>0</td>
<td>SUIDEN, Secure User Invasive Debug Enable</td>
</tr>
<tr>
<td>0</td>
<td>0b0 This bit does not affect the generation of debug exceptions at Secure EL0.</td>
</tr>
</tbody>
</table>

When **EL3 is implemented:**

Secure User Invasive Debug Enable.

0b0 This bit does not affect the generation of debug exceptions at Secure EL0.
If EL1 is using AArch32, debug exceptions from Secure EL0 are enabled.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Accessing SDER32_EL2

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, SDER32_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = SDER32_EL2;
elsif PSTATE_EL == EL3 then
  if SCR_EL3.EEL2 == '0' then
    UNDEFINED;
  else
    X[t, 64] = SDER32_EL2;
```

**MSR SDER32_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TDA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = SDER32_EL2;
```
AArch64.SystemAccessTrap(El3, 0x18);
else
  SDER32_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
  if SCR_EL3.EEL2 == '0' then
    UNDEFINED;
  else
    SDER32_EL2 = X[t, 64];
D17.3.28   SDER32_EL3, AArch32 Secure Debug Enable Register

The SDER32_EL3 characteristics are:

**Purpose**

Allows access to the AArch32 register SDER from AArch64 state only. Its value has no effect on execution in AArch64 state.

**Configurations**

AArch64 System register SDER32_EL3 bits [63:0] are architecturally mapped to AArch64 System register SDER32_EL2[63:0] when EL2 is implemented and FEAT_SEL2 is implemented.

AArch64 System register SDER32_EL3 bits [31:0] are architecturally mapped to AArch32 System register SDER[31:0].

This register is present only when EL3 is implemented and EL1 is capable of using AArch32. Otherwise, direct accesses to SDER32_EL3 are UNDEFINED.

This register is ignored by the PE when one or more of the following are true:

- The PE is in Non-secure state.
- EL1 is using AArch64.

**Attributes**

SDER32_EL3 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>61</td>
<td>0b0 This bit has no effect on non-invasive debug.</td>
</tr>
<tr>
<td>60</td>
<td>0b1 Non-invasive debug is allowed in Secure EL0 using AArch32. When Secure EL1 is using AArch32, the forms of non-invasive debug affected by this control are:</td>
</tr>
<tr>
<td></td>
<td>• The PC Sample-based Profiling Extension. See About the PC Sample-based Profiling Extension.</td>
</tr>
<tr>
<td></td>
<td>• When SelfHostedTraceEnabled() == FALSE, processor trace.</td>
</tr>
<tr>
<td></td>
<td>• Performance Monitors. When Secure EL1 is using AArch64, this bit has no effect. The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>59</td>
<td>SUIDEN, bit [0] Secure User Invasive Debug Enable.</td>
</tr>
<tr>
<td>58</td>
<td>0b0 This bit does not affect the generation of debug exceptions at Secure EL0.</td>
</tr>
<tr>
<td>57</td>
<td>0b1 If EL1 is using AArch32, debug exceptions from Secure EL0 are enabled.</td>
</tr>
<tr>
<td>56</td>
<td></td>
</tr>
</tbody>
</table>
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing SDER32_EL3**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, SDER32_EL3**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elif PSTATE.EL == EL1 then
    UNDEFINED;
elif PSTATE.EL == EL2 then
    UNDEFINED;
elif PSTATE.EL == EL3 then
    X[t, 64] = SDER32_EL3;

**MSR SDER32_EL3, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elif PSTATE.EL == EL1 then
    UNDEFINED;
elif PSTATE.EL == EL2 then
    UNDEFINED;
elif PSTATE.EL == EL3 then
    SDER32_EL3 = X[t, 64];
D17.3.29 TRFCR_EL1, Trace Filter Control Register (EL1)

The TRFCR_EL1 characteristics are:

**Purpose**

Provides EL1 controls for Trace.

**Configurations**

AArch64 System register TRFCR_EL1 bits [31:0] are architecturally mapped to AArch32 System register TRFCR[31:0].

This register is present only when FEAT_TRF is implemented. Otherwise, direct accesses to TRFCR_EL1 are UNDEFINED.

**Attributes**

TRFCR_EL1 is a 64-bit register.

**Field descriptions**

Bits [63:7]

Reserved, RES0.

TS, bits [6:5]

Timestamp Control. Controls which timebase is used for trace timestamps.

- **0b01** Virtual timestamp. The traced timestamp is the physical counter value minus the value of CNTVOFF_EL2.
- **0b10** *When FEAT_ECV is implemented:* Guest physical timestamp. The traced timestamp is the physical counter value minus a physical offset. If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2:
  - SCR_EL3.ECVEn == 0.
  - CNTHCTL_EL2.ECV == 0.
- **0b11** Physical timestamp. The traced timestamp is the physical counter value.

All other values are reserved.

This field is ignored by the PE when any of the following are true:

- EL2 is implemented and TRFCR_EL2.TS != 0b00.
- SelfHostedTraceEnabled () == FALSE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [4:2]**

Reserved, RES0.
E1TRE, bit [1]

EL1 Trace Enable.

0b0 Trace is prohibited at EL1.

0b1 Trace is allowed at EL1.

This field is ignored if SelfHostedTraceEnabled() == FALSE.

The reset behavior of this field is:

• On a Warm reset, this field resets to 0.

E0TRE, bit [0]

EL0 Trace Enable.

0b0 Trace is prohibited at EL0.

0b1 Trace is allowed at EL0.

This field is ignored if any of the following are true:

• SelfHostedTraceEnabled() == FALSE.

• EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE == 1.

The reset behavior of this field is:

• On a Warm reset, this field resets to 0.

Accessing TRFCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, TRFCR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TTRF == '1' then
    UNDEFINED;
  elsif EL2Enabled() && MDCR_EL2.TTRF == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && MDCR_EL3.TTRF == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if
  else
    if Halted() && HCR_EL2.<NV2,NV1,NV> == '111' then
      X[t, 64] = NVMem[0x880];
    else
      X[t, 64] = TRFCR_EL1;
    end if
  endif
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TTRF == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TTRF == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if
  else
    if HCR_EL2.E2H == '1' then
      X[t, 64] = TRFCR_EL2;
    end if
  endif

else
    \(X[t, 64] = TRFCR_EL1\);
elsif \(PSTATE.EL = EL3\) then
    \(X[t, 64] = TRFCR_EL1\);

**MSR TRFCR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if \(PSTATE.EL = EL0\) then
    UNDEFINED;
elsif \(PSTATE.EL = EL1\) then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TTRF == '1' then
        UNDEFINED;
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRFCR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && MDCR_EL2.TTRF == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && MDCR_EL3.TTRF == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        \(NVMem[0x880] = X[t, 64]\);
    else
        TRFCR_EL1 = X[t, 64];
elsif\ PSTATE.EL == EL2\ then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TTRF == '1' then
        UNDEFINED;
elsif HaveEL(EL3) && MDCR_EL3.TTRF == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
elsif HCR_EL2.E2H == '1' then
    TRFCR_EL2 = X[t, 64];
else
    TRFCR_EL1 = X[t, 64];
elsif\ PSTATE.EL == EL3\ then
    TRFCR_EL1 = X[t, 64];

**MRS <Xt>, TRFCR_EL12**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if \(PSTATE.EL = EL0\) then
    UNDEFINED;
elsif \(PSTATE.EL = EL1\) then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        \(X[t, 64] = NVMem[0x880]\);
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.SystemAccessTrap(EL2, 0x18);
UNDEFINED;
eselif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TTRF == '1' then
            UNDEFINED;
        elsif HaveEL(EL3) && MDCR_EL3.TTRF == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        X[t, 64] = TRFCR_EL1;
    end
else
    UNDEFINED;
eselif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        TRFCR_EL1 = X[t, 64];
    else
        UNDEFINED;
    end
else
    if PSTATE.EL == EL0 then
        UNDEFINED;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
            NVMem[0x880] = X[t, 64];
        elsif EL2Enabled() && HCR_EL2.NV == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            UNDEFINED;
        end
    else
        TRFCR_EL1 = X[t, 64];
    end
else
    UNDEFINED;
eselif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TTRF == '1' then
            UNDEFINED;
        elsif HaveEL(EL3) && MDCR_EL3.TTRF == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end
        else
            TRFCR_EL1 = X[t, 64];
        end
    else
        UNDEFINED;
eselif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        TRFCR_EL1 = X[t, 64];
    else
        UNDEFINED;
end

---

**MSR TRFCR_EL12, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

---
D17.3.30 TRFCR_EL2, Trace Filter Control Register (EL2)

The TRFCR_EL2 characteristics are:

**Purpose**

Provides EL2 controls for Trace.

**Configurations**

AArch64 System register TRFCR_EL2 bits [31:0] are architecturally mapped to AArch32 System register HTRFCR[31:0].

This register is present only when FEAT_TRF is implemented. Otherwise, direct accesses to TRFCR_EL2 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

TRFCR_EL2 is a 64-bit register.

**Field descriptions**

Bits [63:7]

Reserved, RES0.

**TS, bits [6:5]**

Timestamp Control. Controls which timebase is used for trace timestamps.

- 0b00 Timestamp controlled by TRFCR_EL1.TS or TRFCR.TS.
- 0b01 Virtual timestamp. The traced timestamp is the physical counter value minus the value of CNTVOFF_EL2.
- 0b10 *When FEAT_ECV is implemented:* Guest physical timestamp. The traced timestamp is the physical counter value minus a physical offset. If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2:
  - SCR_EL3.ECVEn == 0.
  - CNTHCTL_EL2.ECV == 0.
- 0b11 Physical timestamp. The traced timestamp is the physical counter value.

This field is ignored by the PE when SelfHostedTraceEnabled () == FALSE.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

**Bit [4]**

Reserved, RES0.

**CX, bit [3]**

CONTEXTIDR_EL2 and VMID trace enable.

- 0b0 CONTEXTIDR_EL2 and VMID trace prohibited.
0b1 CONTEXTIDR_EL2 and VMID trace allowed.
This field is ignored if SelfHostedTraceEnabled() != FALSE.
The reset behavior of this field is:
• On a Warm reset, this field resets to 0.

Bit [2]
Reserved, RES0.

E2TRE, bit [1]
EL2 Trace Enable.
0b0 Trace is prohibited at EL2.
0b1 Trace is allowed at EL2.
This field is ignored if SelfHostedTraceEnabled() != FALSE.
The reset behavior of this field is:
• On a Warm reset, this field resets to 0.

E0HTRE, bit [0]
EL0 Trace Enable.
0b0 Trace is prohibited at EL0 when HCR_EL2.TGE == 1.
0b1 Trace is allowed at EL0 when HCR_EL2.TGE == 1.
This field is ignored if any of the following are true:
• SelfHostedTraceEnabled() != FALSE.
• EL2 is disabled in the current security state.
• HCR_EL2.TGE == 0.
The reset behavior of this field is:
• On a Warm reset, this field resets to 0.

Accessing TRFCR_EL2
Accesses to this register use the following encodings in the System register encoding space:

|MRS <Xt>, TRFCR_EL2|
|---|---|---|---|---|
|op0| op1| CRn| CRm| op2|
|0b1| 0b100| 0b0001| 0b0010| 0b001|

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & MDCR_EL3.TTRF == '1'" then
    UNDEFINED;
elsif HaveEL(EL3) && MDCR_EL3.TTRF == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRFCR_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = TRFCR_EL2;

MSR TRFCR_EL2, <Xt>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TTRF == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && MDCR_EL3.TTRF == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        endif
    else
        TRFCR_EL2 = X[t, 64];
    endif
elsif PSTATE.EL == EL3 then
    TRFCR_EL2 = X[t, 64];

MRS <Xt>, TRFCR_EL1

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TTRF == '1' then
        UNDEFINED;
    elsif EL2Enabled() && MDCR_EL2.TTRF == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.TTRF == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        endif
    else
        if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
            X[t, 64] = NVMem[0x880];
        else
            X[t, 64] = TRFCR_EL1;
        endif
    endif
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TTRF == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && MDCR_EL3.TTRF == '1' then
        UNDEFINED;
    endif
else
    TRFCR_EL2 = X[t, 64];
endif
if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
elsif HCR_EL2.E2H == '1' then
    X[t, 64] = TRFCR_EL2;
else
    X[t, 64] = TRFCR_EL1;
elsif PSTATE_EL == EL3 then
    X[t, 64] = TRFCR_EL1;

**MSR TRFCR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TTRF == '1' then
        UNDEFINED;
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRFCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.TTRF == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.TTRF == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        elsif HCR_EL2.<NV2,NV1,NV> == '111' then
            NVMem[0x880] = X[t, 64];
        else
            TRFCR_EL1 = X[t, 64];
    elsif PSTATE_EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TTRF == '1' then
            UNDEFINED;
        elsif HaveEL(EL3) && MDCR_EL3.TTRF == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
        elsif HCR_EL2.E2H == '1' then
            TRFCR_EL2 = X[t, 64];
        else
            TRFCR_EL1 = X[t, 64];
    elsif PSTATE_EL == EL3 then
        TRFCR_EL1 = X[t, 64];
else
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
elsif HCR_EL2.E2H == '1' then
    X[t, 64] = TRFCR_EL2;
else
    X[t, 64] = TRFCR_EL1;
D17.4 Trace registers

This section lists the Performance Monitoring registers in AArch64.
D17.4.1 TRBBASER_EL1, Trace Buffer Base Address Register

The TRBBASER_EL1 characteristics are:

**Purpose**
Defines the base address for the trace buffer.

**Configurations**
This register is present only when FEAT_TRBE is implemented. Otherwise, direct accesses to TRBBASER_EL1 are UNDEFINED.

**Attributes**
TRBBASER_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>BASE, bits [63:12]</th>
</tr>
</thead>
</table>
| Trace Buffer Base pointer address. (TRBBASER_EL1.BASE << 12) is the address of the first byte in the trace buffer. Bits [11:0] of the Base pointer address are always zero. If the smallest implemented translation granule is not 4KB, then TRBBASER_EL1[N-1:12] are RES0, where N is the IMPLEMENTATION DEFINED value Log2(smallest implemented translation granule). The reset behavior of this field is:
  - On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Bits [11:0]**
Reserved, RES0.

**Accessing TRBBASER_EL1**
The PE might ignore a direct write to TRBBASER_EL1 if TRBLIMITR_EL1.E == 1.

Accesses to this register use the following encodings in the System register encoding space:

### MRS <Xt>, TRBBASER_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
    (IsFeatureImplemented( FEATURE_RME ) && MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elseif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRBBASER_EL1 == '1') then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && MDCR_EL2.E2TB == 'x0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elseif HaveEL(EL3) && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
if Halted() && EDSCR.SDD == '1' then
  UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRBBASER_EL1;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elseif HaveEL(EL3) && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = TRBBASER_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = TRBBASER_EL1;

### MSR TRBBASER_EL1, <xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elseif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRBBASER_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && MDCR_EL2.E2TB == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif HaveEL(EL3) && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRBBASER_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elseif HaveEL(EL3) && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRBBASER_EL1 = X[t, 64];
elsif PSTATE_EL == EL3 then
    TRBBASER_EL1 = X[t, 64];
D17.4.2 TRBIDR_EL1, Trace Buffer ID Register

The TRBIDR_EL1 characteristics are:

**Purpose**

Describes constraints on using the Trace Buffer Unit to software, including whether the Trace Buffer Unit can be programmed at the current Exception level.

**Configurations**

This register is present only when FEAT_TRBE is implemented. Otherwise, direct accesses to TRBIDR_EL1 are UNDEFINED.

**Attributes**

TRBIDR_EL1 is a 64-bit register.

**Field descriptions**

| Bits [63:12] | Reserved, RES0. |
| Bits [7:6] | Reserved, RES0. |

**From Armv9.3:**

External Abort handling. Describes how the PE manages External aborts on writes made by the Trace Buffer Unit to the trace buffer.

- **0b0000**: Not described.
- **0b0001**: The PE ignores External aborts on writes made by the Trace Buffer Unit.
- **0b0010**: The External abort generates an SError interrupt at the PE.

All other values are reserved.

From Armv9.3, the value **0b0000** is not permitted.

TRBIDR_EL1.EA describes only External aborts generated by the write to memory. External aborts on a translation table walk made by the Trace Buffer Unit generate trace buffer management events reported as MMU faults using TRBSR_EL1.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Otherwise:**

Reserved, RES0.
Hardware management of the Access flag and dirty state for accesses made by the Trace Buffer Unit is controlled in the same way as explicit memory accesses in the trace buffer owning translation regime.

--- Note ---

If hardware management of the Access flag is disabled for a stage of translation, an access to a Page or Block with the Access flag bit not set in the descriptor will generate an Access Flag fault.

If hardware management of the dirty state is disabled for a stage of translation, an access to a Page or Block will ignore the Dirty Bit Modifier in the descriptor and might generate a Permission fault, depending on the values of the access permission bits in the descriptor.

From Armv9.3, the value 0 is not permitted.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

P, bit [4]

Programming not allowed. When read at EL3, this field reads as zero. Otherwise, indicates that the trace buffer is owned by a higher Exception level or another Security state. Defined values are:

- **0b0**  Programming is allowed.
- **0b1**  Programming not allowed.

The value read from this field depends on the current Exception level and the Effective values of MDCR_EL3.NSTB, MDCR_EL3.NSTBE, and MDCR_EL2.E2TB:

- If EL3 is implemented, and the owning Security state is Secure state, this field reads as one from:
  - Non-secure EL1 and Non-secure EL2.
  - If FEAT_RME is implemented, Realm EL1 and Realm EL2.
  - If Secure EL2 is implemented and enabled, and MDCR_EL2.E2TB is 0b00, Secure EL1.
- If EL3 is implemented, and the owning Security state is Non-secure state, this field reads as one from:
  - Secure EL1.
  - If Secure EL2 is implemented, Secure EL2.
  - If EL2 is implemented and MDCR_EL2.E2TB is 0b00, Non-secure EL1.
  - If FEAT_RME is implemented, Realm EL1 and Realm EL2.
- If FEAT_RME is implemented, and the owning Security state is Realm state, this field reads as one from:
  - Non-secure EL1 and Non-secure EL2.
  - Secure EL1 and Secure EL2.
  - If MDCR_EL2.E2TB is 0b00, Realm EL1.
- If EL3 is not implemented, EL2 is implemented, and MDCR_EL2.E2TB is 0b00, this field reads as one from EL1.
- Otherwise, this field reads as zero.

Align, bits [3:0]

Defines the minimum alignment constraint for writes to TRBPTR_EL1 and TRBTRG_EL1.

Defined values are:

- **0b0000**  Byte.
- **0b0001**  Halfword.
- **0b0010**  Word.
- **0b0011**  Doubleword.
Accessing TRBIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

$\text{MRS} \langle X_t \rangle, \text{TRBIDR}_\text{EL1}$

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b01001</td>
<td>0b01011</td>
<td>0b111</td>
</tr>
</tbody>
</table>

If PSTATE.EL == EL0 then
   UNDEFINED;
else if PSTATE.EL == EL1 then
   if EL2Enabled() \& (\!(HaveEL(EL3)) \& (SCR_EL3.FGTEn == \'1\') \& (HDFGTR_EL2.TRBIDR_EL1 == \'1\') then
      AArch64.SystemAccessTrap(EL2, 0x18);
   else
      X[t, 64] = TRBIDR_EL1;
else if PSTATE.EL == EL2 then
   X[t, 64] = TRBIDR_EL1;
else if PSTATE.EL == EL3 then
   X[t, 64] = TRBIDR_EL1;
D17.4.3 TRBLIMITR_EL1, Trace Buffer Limit Address Register

The TRBLIMITR_EL1 characteristics are:

Purpose

Defines the top address for the trace buffer, and controls the trace buffer modes and enable.

Configurations

This register is present only when FEAT_TRBE is implemented. Otherwise, direct accesses to TRBLIMITR_EL1 are UNDEFINED.

Attributes

TRBLIMITR_EL1 is a 64-bit register.

Field descriptions

LIMIT, bits [63:12]

Trace Buffer Limit pointer address. (TRBLIMITR_EL1.LIMIT << 12) is the address of the last byte in the trace buffer plus one. Bits [11:0] of the Limit pointer address are always zero. If the smallest implemented translation granule is not 4KB, then TRBLIMITR_EL1[N-1:12] are RES0, where N is the IMPLEMENTATION DEFINED value Log2(smallest implemented translation granule).

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [11:6]

Reserved, RES0.

nVM, bit [5]

Address mode.

0b0 The trace buffer pointers are virtual addresses.

0b1 The trace buffer pointers are:

- Physical address in the owning security state if the owning translation regime has no stage 2 translation.
- Intermediate physical addresses in the owning security state if the owning translation regime has stage 2 translations.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

TM, bits [4:3]

Trigger mode.

0b00 Stop on trigger. Flush then stop collection and raise maintenance interrupt on Trigger Event.

0b01 IRQ on trigger. Continue collection and raise maintenance interrupt on Trigger Event.

0b11 Ignore trigger. Continue collection and do not raise maintenance interrupt on Trigger Event.
All other values are reserved.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

**FM, bits [2:1]**
Trace buffer mode.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>Fill mode. Stop collection and raise maintenance interrupt on current write pointer wrap.</td>
</tr>
<tr>
<td>0b01</td>
<td>Wrap mode. Continue collection and raise maintenance interrupt on current write pointer wrap.</td>
</tr>
<tr>
<td>0b11</td>
<td>Circular Buffer mode. Continue collection and do not raise maintenance interrupt on current write pointer wrap.</td>
</tr>
</tbody>
</table>

All other values are reserved.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

**E, bit [0]**
Trace Buffer Unit enable.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Trace Buffer Unit disabled.</td>
</tr>
<tr>
<td>1</td>
<td>Trace Buffer Unit enabled by this control.</td>
</tr>
</tbody>
</table>

Regardless of the value of this field, the Trace Buffer Unit is disabled when `SelfHostedTraceEnabled()` == FALSE. All output is discarded by the Trace Buffer Unit when it is disabled.
The reset behavior of this field is:
• On a Warm reset, this field resets to 0.

**Accessing TRBLIMITR_EL1**
The PE might ignore a direct write to TRBLIMITR_EL1, other than a direct write that modifies TRBLIMITR_EL1.E, if TRBLIMITR_EL1.E == 1.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRBLIMITR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCCR_EL3.NSTB[0] == '0' && MDCCR_EL3.NSTB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCCR_EL3.NSTBE != SCR_EL3.NSE) then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRBLIMITR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCCR_EL2.E2TB == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && (MDCCR_EL3.NSTB[0] == '0' || MDCCR_EL3.NSTB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCCR_EL3.NSTBE != SCR_EL3.NSE) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRBLIMITR_EL1;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'":
        MSR TRBLIMITR_EL1, <Xt>
    elsif PSTATE.EL == EL0 then
        UNDEFINED;
    elsif PSTATE.EL == EL1 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'":
            MSR TRBLIMITR_EL1, <Xt>
    elsif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'
when SDD == '1':
        MSR TRBLIMITR_EL1, <Xt>
    elsif PSTATE.EL == EL3 then
        MSR TRBLIMITR_EL1, <Xt>
else
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1':
        MSR TRBLIMITR_EL1, <Xt>
    elsif PSTATE.EL == EL4 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1':
            MSR TRBLIMITR_EL1, <Xt>
    elsif PSTATE.EL == EL5 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1':
            MSR TRBLIMITR_EL1, <Xt>
    elsif PSTATE.EL == EL6 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1':
            MSR TRBLIMITR_EL1, <Xt>
    elsif PSTATE.EL == EL7 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1':
            MSR TRBLIMITR_EL1, <Xt>
D17.4.4 TRBMAR_EL1, Trace Buffer Memory Attribute Register

The TRBMAR_EL1 characteristics are:

**Purpose**

- Controls Trace Buffer Unit accesses to memory.
- If the trace buffer pointers specify virtual addresses, the address properties are defined by the translation tables and this register is ignored.

**Configurations**

- This register is present only when FEAT_TRBE is implemented. Otherwise, direct accesses to TRBMAR_EL1 are UNDEFINED.

**Attributes**

- TRBMAR_EL1 is a 64-bit register.

**Field descriptions**

Bits [63:10]

- Reserved, RES0.

SH, bits [9:8]

- Trace buffer shareability domain. Defines the shareability domain for Normal memory used by the trace buffer.
  - 0b00: Non-shareable.
  - 0b10: Outer Shareable.
  - 0b11: Inner Shareable.
- All other values are reserved.
- This field is ignored when TRBMAR_EL1.Attr specifies any of the following memory types:
  - Any Device memory type.
  - Normal memory, Inner Non-cacheable, Outer Non-cacheable.
- All Device and Normal Inner Non-cacheable Outer Non-cacheable memory regions are always treated as Outer Shareable.
- The reset behavior of this field is:
  - On a Warm reset, this field resets to an architecturally UNKNOWN value.

Attr[7:4], bits [7:4]

*When TRBMAR_EL1.Attr[3:0] \(\neq\) 0b0000:*

- Trace buffer memory type and attributes. Defines the memory type and, for Normal memory, the cacheability attributes, for memory addressed by the trace buffer.
- 0b0000: Device-nGnRnE memory.
- 0b0100: When FEAT_XS is implemented:
  - Normal memory, Inner Non-cacheable, Outer Non-cacheable with the XS attribute set to 0.
When FEAT_XS is implemented:
Normal memory, Inner Write-through Cacheable, Outer Write-through Cacheable,
Non-transient, Read-Allocate with the XS attribute set to 0.

When FEAT_MTE2 is implemented:
Tagged Normal memory, Outer Write-Back Non-transient, Read-allocate Write-allocation.

All other values are reserved.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Trace buffer memory type and attributes. Defines the memory type and, for Normal memory, the
Outer cacheability attributes, for memory addressed by the trace buffer.

0b0000 Device memory. The Device memory type is defined by TRBMAR_EL1.Attr[3:0].
0b0001 Normal memory, Outer Write-Through Transient, Write-allocate.
0b0010 Normal memory, Outer Write-Through Transient, Read-allocate.
0b0011 Normal memory, Outer Write-Through Transient, Read-allocate Write-allocate.
0b0100 Normal memory, Outer Non-cacheable.
0b0101 Normal memory, Outer Write-Back Transient, Write-allocate.
0b0110 Normal memory, Outer Write-Back Transient, Read-allocate.
0b0111 Normal memory, Outer Write-Back Transient, Read-allocate Write-allocate.
0b1000 Normal memory, Outer Write-Through Non-transient, No allocate.
0b1001 Normal memory, Outer Write-Through Non-transient, Write-allocate.
0b1010 Normal memory, Outer Write-Through Non-transient, Read-allocate.
0b1011 Normal memory, Outer Write-Through Non-transient, Read-allocate Write-allocate.
0b1100 Normal memory, Outer Write-Back Non-transient, No allocate.
0b1101 Normal memory, Outer Write-Back Non-transient, Write-allocate.
0b1110 Normal memory, Outer Write-Back Non-transient, Read-allocate.
0b1111 Normal memory, Outer Write-Back Non-transient, Read-allocate Write-allocate.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Attr[3:0], bits [3:0]
When TRBMAR_EL1.Attr[7:4] == 0b0000:
Trace buffer memory attributes. Defines the Device memory attributes for memory addressed by the
trace buffer.

0b0000 Device-nGnRnE memory.
0b0100 Device-nGnRE memory.
0b1000 Device-nGRE memory.
0b1100 Device-GRE memory.
0b0001 When FEAT_XS is implemented:
Device-nGnRnE memory with the XS attribute set to 0.
0b0101 When FEAT_XS is implemented:
Device-nGnRE memory with the XS attribute set to 0.
0b1001 When FEAT_XS is implemented:
Device-nGRE memory with the XS attribute set to 0.
Device-GRE memory with the XS attribute set to 0.
All other values are reserved.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Trace buffer memory attributes. Defines the Inner cacheability attributes for memory addressed by the trace buffer.

0b0000  *When FEAT_MTE2 is implemented or FEAT_XS is implemented:*
The memory type is defined by TRBMAR_EL1.Attr[7:4].

0b0001  Normal memory, Inner Write-Through Transient, Write-allocate.
0b0010  Normal memory, Inner Write-Through Transient, Read-allocate.
0b0011  Normal memory, Inner Write-Through Transient, Read-allocate Write-allocate.
0b0100  Normal memory, Inner Non-cacheable.
0b0101  Normal memory, Inner Write-Back Transient, Write-allocate.
0b0110  Normal memory, Inner Write-Back Transient, Read-allocate.
0b0111  Normal memory, Inner Write-Back Transient, Read-allocate Write-allocate.
0b1000  Normal memory, Inner Write-Through Non-transient, No allocate.
0b1001  Normal memory, Inner Write-Through Non-transient, Write-allocate.
0b1010  Normal memory, Inner Write-Through Non-transient, Read-allocate.
0b1011  Normal memory, Inner Write-Through Non-transient, Read-allocate Write-allocate.
0b1100  Normal memory, Inner Write-Back Non-transient, No allocate.
0b1101  Normal memory, Inner Write-Back Non-transient, Write-allocate.
0b1110  Normal memory, Inner Write-Back Non-transient, Read-allocate.
0b1111  Normal memory, Inner Write-Back Non-transient, Read-allocate Write-allocate.
All other values are reserved.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing TRBMAR_EL1**
The PE might ignore a direct write to TRBMAR_EL1 if TRBLIMITR_EL1.E == 1.
Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRBMAR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b10001</td>
<td>0b1011</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS) && (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSTBE != SCR_EL3.NSE) then
  UNDEFINED;
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRBMAR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.E2TB == 'x0' then

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'' && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'' && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    TRBMAR_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL3 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'' && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    TRBMAR_EL1 = X[t, 64];
  end

---

### MSR TRBMAR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1011</td>
<td>0b100</td>
</tr>
</tbody>
</table>

---
elsif PSTATE.EL == EL3 then
    TRMAR_EL1 = X[t, 64];
D17.4.5 TRBPTR_EL1, Trace Buffer Write Pointer Register

The TRBPTR_EL1 characteristics are:

**Purpose**

Defines the current write pointer for the trace buffer.

**Configurations**

This register is present only when FEAT_TRBE is implemented. Otherwise, direct accesses to TRBPTR_EL1 are UNDEFINED.

**Attributes**

TRBPTR_EL1 is a 64-bit register.

**Field descriptions**

PTR, bits [63:0]

Trace Buffer current write pointer address.

Defines the virtual address of the next entry to be written to the trace buffer.

The architecture places restrictions on the values that software can write to the pointer.

Note

As a result of the restrictions an implementation might treat some of PTR[M:0] as RES0, where M is defined by TRBIDR_EL1.Align.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Accessing TRBPTR_EL1**

The PE might ignore a direct write to TRBPTR_EL1 if TRBLIMITR_EL1.E == 1.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRBPTR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SODD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SODD == '1'" \&\& (MDCR_EL3.NSTB[0] == '0' \&\& MDCR_EL3.NSTB[1] != SCR_EL3.NS) \&\& (IsFeatureImplemented(FEAT_RME) \&\& MDCR_EL3.NSTBE != SCR_EL3.NSE) then
        UNDEFINED;
    elsif EL2Enabled() \&\& !(HaveEL(EL3) \&\& SCR_EL3.FGTEn == '1') \&\& HDFGTR_EL2.TRBPTR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && MDCR_EL2.E2TB == 'x0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
elsif PSTATE.EL == EL2 then
if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
UNDEFINED;
elsif HaveEL(EL3) && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
elsif PSTATE.EL == EL3 then
X[t, 64] = TRBPTR_EL1;
elsif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
UNDEFINED;
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGWTR_EL2.TRBPTR_EL1 == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && MDCR_EL2.E2TB == 'x0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
TRBPTR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
UNDEFINED;
elsif HaveEL(EL3) && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
TRBPTR_EL1 = X[t, 64];

**MSR TRBPTR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b0000</td>
<td>0b1001</td>
<td>0b1011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGWTR_EL2.TRBPTR_EL1 == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && MDCR_EL2.E2TB == 'x0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
TRBPTR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
UNDEFINED;
elsif HaveEL(EL3) && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
TRBPTR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    TRBPTR_EL1 = X[t, 64];
**D17.4.6 TRBSR_EL1, Trace Buffer Status/syndrome Register**

The TRBSR_EL1 characteristics are:

**Purpose**

Provides syndrome information to software for a trace buffer management event.

**Configurations**

This register is present only when FEAT_TRBE is implemented. Otherwise, direct accesses to TRBSR_EL1 are UNDEFINED.

**Attributes**

TRBSR_EL1 is a 64-bit register.

**Field descriptions**

Bits [63:32]

Reserved, RES0.

EC, bits [31:26]

Event class. Top-level description of the cause of the trace buffer management event.

- **0b000000** Other trace buffer management event. All trace buffer management events other than those described by the other defined Event class codes.

- **0b011110** *When FEAT_RME is implemented:* Granule Protection Check fault, other than GPF, on write to trace buffer.

- **0b011111** Buffer management event for IMPLEMENTATION DEFINED reason.

- **0b100100** Stage 1 Data Abort on write to trace buffer.

- **0b100101** Stage 2 Data Abort on write to trace buffer.

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [25:23]

Reserved, RES0.

IRQ, bit [22]

Maintenance interrupt status.

- **0b0** Maintenance interrupt is not asserted.

- **0b1** Maintenance interrupt is asserted.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
TRG, bit [21]

Triggered.
0b0  No Detected Trigger has been observed since this field was last cleared to zero.
0b1  A Detected Trigger has been observed since this field was last cleared to zero.

The reset behavior of this field is:
  •  On a Cold reset, this field resets to an architecturally UNKNOWN value.

WRAP, bit [20]

Wrapped.
0b0  The current write pointer has not wrapped since this field was last cleared to zero.
0b1  The current write pointer has wrapped since this field was last cleared to zero.

For each byte of trace the Trace Buffer Unit Accepts and writes to the trace buffer at the address in
the current write pointer, if the current write pointer is equal to the Limit pointer minus one, the
current write pointer is wrapped by setting it to the Base pointer, and this field is set to 1.

The reset behavior of this field is:
  •  On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bit [19]

Reserved, RES0.

EA, bit [18]

External Abort.
0b0  An External Abort has not been asserted.
0b1  An External Abort has been asserted and detected by the Trace Buffer Unit.

The reset behavior of this field is:
  •  On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:
  •  When the PE never sets this field as the result of an External Abort, access to this field is
     RES0.
  •  Otherwise, access to this field is RW

S, bit [17]

Stopped.
0b0  Collection has not been stopped.
0b1  Collection is stopped.

The reset behavior of this field is:
  •  On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bit [16]

Reserved, RES0.

MSS, bits [15:0]

Management Event Specific Syndrome. Contains syndrome specific to the management event.
The syndrome contents for each management event are described in the following sections.
The reset behavior of this field is:
  •  On a Cold reset, this field resets to an architecturally UNKNOWN value.
**MSS encoding for other trace buffer management events**

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BSC, bits [5:0]</td>
<td>Trace buffer status code.</td>
</tr>
<tr>
<td>0b000000</td>
<td>Collection not stopped.</td>
</tr>
<tr>
<td>0b000001</td>
<td>Trace buffer filled. Collection stopped because the current write pointer wrapped to the base pointer and the trace buffer mode is Fill mode.</td>
</tr>
<tr>
<td>0b000010</td>
<td>Trigger Event. Collection stopped because of a Trigger Event. See TRBTRG_EL1 for more information.</td>
</tr>
<tr>
<td>All other values are reserved.</td>
<td></td>
</tr>
</tbody>
</table>

**MSS encoding for Buffer management event for IMPLEMENTATION DEFINED reason**

<table>
<thead>
<tr>
<th>Bits [15:0]</th>
<th>IMPLEMENTATION DEFINED.</th>
</tr>
</thead>
</table>

**MSS encoding for Granule Protection Check fault**

| Bits [15:0] | Reserved, RES0. |

**MSS encoding for stage 1 or stage 2 Data Aborts on write to trace buffer**

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000000</td>
<td>Address size fault, level 0 of translation or translation table base register.</td>
</tr>
<tr>
<td>0b000001</td>
<td>Address size fault, level 1.</td>
</tr>
<tr>
<td>0b000010</td>
<td>Address size fault, level 2.</td>
</tr>
</tbody>
</table>
0b00011  Address size fault, level 3.
0b00010  Translation fault, level 0.
0b00011  Translation fault, level 1.
0b00110  Translation fault, level 2.
0b00111  Translation fault, level 3.
0b01011  Access flag fault, level 1.
0b01010  Access flag fault, level 2.
0b01001  Access flag fault, level 3.
0b01000  When FEAT_LPA2 is implemented:
          Access flag fault, level 0.
0b01100  When FEAT_LPA2 is implemented:
          Permission fault, level 0.
0b00110  Permission fault, level 1.
0b00111  Permission fault, level 2.
0b00101  Permission fault, level 3.
0b10000  Synchronous External abort, not on translation table walk or hardware update of translation table.
0b10001  Asynchronous External abort.
0b10011  When FEAT_LPA2 is implemented:
          Synchronous External abort on translation table walk or hardware update of translation table, level -1.
0b10010  Synchronous External abort on translation table walk or hardware update of translation table, level 0.
0b10011  Synchronous External abort on translation table walk or hardware update of translation table, level 1.
0b10010  Synchronous External abort on translation table walk or hardware update of translation table, level 2.
0b10011  Synchronous External abort on translation table walk or hardware update of translation table, level 3.
0b11011  When FEAT_LPA2 is implemented and FEAT_RAS is not implemented:
          Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.
0b10000  Alignment fault.
0b10011  When FEAT_RME is implemented and FEAT_LPA2 is implemented:
          Granule Protection Fault on translation table walk or hardware update of translation table, level -1.
0b10010  When FEAT_RME is implemented:
          Granule Protection Fault on translation table walk or hardware update of translation table, level 0.
0b10011  When FEAT_RME is implemented:
          Granule Protection Fault on translation table walk or hardware update of translation table, level 1.
0b10010  When FEAT_RME is implemented:
          Granule Protection Fault on translation table walk or hardware update of translation table, level 2.
0b10011  When FEAT_RME is implemented:
          Granule Protection Fault on translation table walk or hardware update of translation table, level 3.
0b101000  When FEAT_RME is implemented:
Granule Protection Fault, not on translation table walk or hardware update of translation table.

0b101001  When FEAT_LPA2 is implemented:
Address size fault, level -1.

0b101011  When FEAT_LPA2 is implemented:
Translation fault, level -1.

0b100000  TLB conflict abort.

0b100001  When FEAT_HAFDBS is implemented:
Unsupported atomic hardware update fault.

All other values are reserved.

Accessing TRBSR_EL1

The PE might ignore a direct write to TRBSR_EL1 if TRBLIMITR_EL1.E == 1.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRBSR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b00</td>
<td>0b1001</td>
<td>0b1011</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SOD != '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS) & (IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSTBE != SCR_EL3.NSE) then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGTR_EL2.TRBSR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.ETB == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS) & (IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSTBE != SCR_EL3.NSE) then
    if Halted() && EDSCR.SOD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    if PSTATE_EL == EL2 then
      if Halted() && HaveEL(EL3) && EDSCR.SOD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS) & (IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSTBE != SCR_EL3.NSE) then
        UNDEFINED;
      elsif HaveEL(EL3) && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS) & (IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSTBE != SCR_EL3.NSE) then
        if Halted() && EDSCR.SOD == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL3, 0x18);
        end
      else
        X[t, 64] = TRBSR_EL1;
      end
    elsif PSTATE_EL == EL3 then
      X[t, 64] = TRBSR_EL1;
    else
      X[t, 64] = TRBSR_EL1;
    end
  end
endif
**MSR TRBSR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1011</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && \boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && \( MDCCR_EL3.NSTB[0] == '0' \&\& MDCCR_EL3.NSTB[1] != SCR_EL3.NS \) && (IsFeatureImplemented(FEAT_RME) && MDCCR_EL3.NSTBE != SCR_EL3.NSE) then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRBSR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCCR_EL2.E2TB == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && (MDCCR_EL3.NSTB[0] == '0' || MDCCR_EL3.NSTB[1] != SCR_EL3.NS) && (IsFeatureImplemented(FEAT_RME) && MDCCR_EL3.NSTBE != SCR_EL3.NSE) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    TRBSR_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && \boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && \( MDCCR_EL3.NSTB[0] == '0' \&\& MDCCR_EL3.NSTB[1] != SCR_EL3.NS \) && (IsFeatureImplemented(FEAT_RME) && MDCCR_EL3.NSTBE != SCR_EL3.NSE) then
    UNDEFINED;
  elsif HaveEL(EL3) && (MDCCR_EL3.NSTB[0] == '0' || MDCCR_EL3.NSTB[1] != SCR_EL3.NS) && (IsFeatureImplemented(FEAT_RME) && MDCCR_EL3.NSTBE != SCR_EL3.NSE) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    TRBSR_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL3 then
  TRBSR_EL1 = X[t, 64];
D17.4.7 TRBTRG_EL1, Trace Buffer Trigger Counter Register

The TRBTRG_EL1 characteristics are:

**Purpose**

Specifies the number of bytes of trace to capture following a Detected Trigger before a Trigger Event.

**Configurations**

This register is present only when FEAT_TRBE is implemented. Otherwise, direct accesses to TRBTRG_EL1 are UNDEFINED.

**Attributes**

TRBTRG_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
<th>63 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRG</td>
<td>RES0</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**TRG, bits [31:0]**

Trigger count.

Specifies the number of bytes of trace to capture following a Detected Trigger before a Trigger Event.

TRBTRG_EL1 decrements by 1 for every byte of trace written to the trace buffer when all of the following are true:

- TRBTRG_EL1 is nonzero.
- TRBSR_EL1.TRG is 1.

The architecture places restrictions on the values that software can write to the counter.

--- **Note** ---

As a result of the restrictions an implementation might treat some of TRG[M:0] as RES0, where M is defined by TRBIDR_EL1.Align.

---

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Accessing TRBTRG_EL1**

The PE might ignore a direct write to TRBTRG_EL1 if TRBLIMITR_EL1.E == 1.

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS) then
    UNDEFINED;
  elseif EL2Enabled() && (HaveEL(EL3) && HDFGRTR_EL2.TRBTRG_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && MDCR_EL2.E2TB == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif HaveEL(EL3) && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end else
X[t, 64] = TRBTRG_EL1;
else
  X[t, 64] = TRBTRG_EL1;
end if

if PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS) then
    UNDEFINED;
  elseif EL2Enabled() && (HaveEL(EL3) && HDFGRTR_EL2.TRBTRG_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && MDCR_EL2.E2TB == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif HaveEL(EL3) && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end else
X[t, 64] = TRBTRG_EL1;
else
  X[t, 64] = TRBTRG_EL1;
end if

if PSTATE.EL == EL3 then
  X[t, 64] = TRBTRG_EL1;
end if

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS) then
    UNDEFINED;
  elseif EL2Enabled() && (HaveEL(EL3) && HDFGRTR_EL2.TRBTRG_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && MDCR_EL2.E2TB == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif HaveEL(EL3) && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end else
X[t, 64] = TRBTRG_EL1;
else
  X[t, 64] = TRBTRG_EL1;
end if
UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  TRBTRG_EL1 = X[t, 64];
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elsif HaveEL(EL3) && (MDCR_EL3.NSTB[0] == '0' || MDCR_EL3.NSTB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSTBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if
else
  TRBTRG_EL1 = X[t, 64];
elsif PSTATE_EL == EL3 then
  TRBTRG_EL1 = X[t, 64];
**D17.4.8 TRCACATR<n>, Address Comparator Access Type Register <n>, n = 0 - 15**

The TRCACATR<n> characteristics are:

**Purpose**

Defines the type of access for the corresponding TRCACVR<n> Register. This register configures the context type, Exception levels, alignment, masking that is applied by the Address Comparator, and how the Address Comparator behaves when it is one half of an Address Range Comparator.

**Configurations**

AArch64 System register TRCACATR<n> bits [63:0] are architecturally mapped to External register TRCACATR<n>[63:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR4.NUMACPAIRS) * 2 > n. Otherwise, direct accesses to TRCACATR<n> are UNDEFINED.

**Attributes**

TRCACATR<n> is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-19</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>18</td>
<td>EXLEVEL_RL_EL2, bit [18]</td>
</tr>
<tr>
<td></td>
<td><strong>When TRCIDR6.EXLEVEL_RL_EL2 == 1:</strong></td>
</tr>
<tr>
<td></td>
<td>Realm EL2 address comparison control. Controls whether a comparison can occur at EL2 in Realm state.</td>
</tr>
<tr>
<td></td>
<td>0b0 When TRCACATR&lt;n&gt;.EXLEVEL_NS_EL2 is 0 the Address Comparator performs comparisons in Realm EL2.</td>
</tr>
<tr>
<td></td>
<td>When TRCACATR&lt;n&gt;.EXLEVEL_NS_EL2 is 1 the Address Comparator does not perform comparisons in Realm EL2.</td>
</tr>
<tr>
<td></td>
<td>0b1 When TRCACATR&lt;n&gt;.EXLEVEL_NS_EL2 is 0 the Address Comparator does not perform comparisons in Realm EL2.</td>
</tr>
<tr>
<td></td>
<td>When TRCACATR&lt;n&gt;.EXLEVEL_NS_EL2 is 1 the Address Comparator performs comparisons in Realm EL2.</td>
</tr>
<tr>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td></td>
<td><strong>Otherwise:</strong></td>
</tr>
<tr>
<td></td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>
EXLEVEL_RL_EL1, bit [17]

When TRCIDR6.EXLEVEL_RL_EL1 == 1:

Realm EL1 address comparison control. Controls whether a comparison can occur at EL1 in Realm state.

0b0  When TRCACATR<n>.EXLEVEL_NS_EL1 is 0 the Address Comparator performs comparisons in Realm EL1.
    When TRCACATR<n>.EXLEVEL_NS_EL1 is 1 the Address Comparator does not perform comparisons in Realm EL1.

0b1  When TRCACATR<n>.EXLEVEL_NS_EL1 is 0 the Address Comparator does not perform comparisons in Realm EL1.
    When TRCACATR<n>.EXLEVEL_NS_EL1 is 1 the Address Comparator performs comparisons in Realm EL1.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_RL_EL0, bit [16]

When TRCIDR6.EXLEVEL_RL_EL0 == 1:

Realm EL0 address comparison control. Controls whether a comparison can occur at EL0 in Realm state.

0b0  When TRCACATR<n>.EXLEVEL_NS_EL0 is 0 the Address Comparator performs comparisons in Realm EL0.
    When TRCACATR<n>.EXLEVEL_NS_EL0 is 1 the Address Comparator does not perform comparisons in Realm EL0.

0b1  When TRCACATR<n>.EXLEVEL_NS_EL0 is 0 the Address Comparator does not perform comparisons in Realm EL0.
    When TRCACATR<n>.EXLEVEL_NS_EL0 is 1 the Address Comparator performs comparisons in Realm EL0.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [15]

Reserved, RES0.

EXLEVEL_NS_EL2, bit [14]

When Non-secure EL2 is implemented:

Non-secure EL2 address comparison control. Controls whether a comparison can occur at EL2 in Non-secure state.

0b0  The Address Comparator performs comparisons in Non-secure EL2.

0b1  The Address Comparator does not perform comparisons in Non-secure EL2.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
EXLEVEL_NS_EL1, bit [13]

When Non-secure EL1 is implemented:
Non-secure EL1 address comparison control. Controls whether a comparison can occur at EL1 in Non-secure state.
0b0 The Address Comparator performs comparisons in Non-secure EL1.
0b1 The Address Comparator does not perform comparisons in Non-secure EL1.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EXLEVEL_NS_EL0, bit [12]

When Non-secure EL0 is implemented:
Non-secure EL0 address comparison control. Controls whether a comparison can occur at EL0 in Non-secure state.
0b0 The Address Comparator performs comparisons in Non-secure EL0.
0b1 The Address Comparator does not perform comparisons in Non-secure EL0.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EXLEVEL_S_EL3, bit [11]

When EL3 is implemented:
EL3 address comparison control. Controls whether a comparison can occur at EL3.
0b0 The Address Comparator performs comparisons in EL3.
0b1 The Address Comparator does not perform comparisons in EL3.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EXLEVEL_S_EL2, bit [10]

When EL2 is implemented and FEAT_SEL2 is implemented:
Secure EL2 address comparison control. Controls whether a comparison can occur at EL2 in Secure state.
0b0 The Address Comparator performs comparisons in Secure EL2.
0b1 The Address Comparator does not perform comparisons in Secure EL2.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EXLEVEL_S_EL1, bit [9]

When Secure EL1 is implemented:
Secure EL1 address comparison control. Controls whether a comparison can occur at EL1 in Secure state.
0b0 The Address Comparator performs comparisons in Secure EL1.
The Address Comparator does not perform comparisons in Secure EL1.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**EXLEVEL__S_EL0, bit [8]**

*When Secure EL0 is implemented:*
Secure EL0 address comparison control. Controls whether a comparison can occur at EL0 in Secure state.

0b0  The Address Comparator performs comparisons in Secure EL0.
0b1  The Address Comparator does not perform comparisons in Secure EL0.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**Bit [7]**
Reserved, RES0.

**CONTEXT, bits [6:4]**

*When TRCIDR4.NUMCIDC != 0b0000 or TRCIDR4.NUMVMIDC != 0b0000:*
Selects a Context Identifier Comparator or Virtual Context Identifier Comparator:

0b000  Comparator 0.
0b001  When UInt(TRCIDR4.NUMCIDC) > 1 or UInt(TRCIDR4.NUMVMIDC) > 1:
        Comparator 1.
0b010  When UInt(TRCIDR4.NUMCIDC) > 2 or UInt(TRCIDR4.NUMVMIDC) > 2:
        Comparator 2.
0b011  When UInt(TRCIDR4.NUMCIDC) > 3 or UInt(TRCIDR4.NUMVMIDC) > 3:
        Comparator 3.
0b100  When UInt(TRCIDR4.NUMCIDC) > 4 or UInt(TRCIDR4.NUMVMIDC) > 4:
        Comparator 4.
0b101  When UInt(TRCIDR4.NUMCIDC) > 5 or UInt(TRCIDR4.NUMVMIDC) > 5:
        Comparator 5.
0b110  When UInt(TRCIDR4.NUMCIDC) > 6 or UInt(TRCIDR4.NUMVMIDC) > 6:
        Comparator 6.
0b111  When UInt(TRCIDR4.NUMCIDC) > 7 or UInt(TRCIDR4.NUMVMIDC) > 7:
        Comparator 7.

The width of this field is dependent on the maximum number of Context Identifier Comparators or Virtual Context Identifier Comparators implemented. Unimplemented bits are RES0.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.
CONTEXTTYPE, bits [3:2]

When TRCIDR4.NUMCIDC \neq 0b0000 or TRCIDR4.NUMVMIDC \neq 0b0000:

Controls whether the Address Comparator is dependent on a Context Identifier Comparator, a Virtual Context Identifier Comparator, or both comparisons.

- **0b00** When TRCIDR4.NUMCIDC \neq 0b0000: The Address Comparator is not dependent on the Context Identifier Comparators or Virtual Context Identifier Comparators.
- **0b01** When TRCIDR4.NUMVMIDC \neq 0b0000: The Address Comparator is dependent on the Context Identifier Comparator that TRCACATR<\text{n}>.CONTEXT specifies. The Address Comparator signals a match only if both the Context Identifier Comparator and the address comparison match.
- **0b10** When TRCIDR4.NUMVMIDC != 0b0000: The Address Comparator is dependent on the Virtual Context Identifier Comparator that TRCACATR<\text{n}>.CONTEXT specifies. The Address Comparator signals a match only if both the Virtual Context Identifier Comparator and the address comparison match.
- **0b11** When TRCIDR4.NUMCIDC \neq 0b0000 and TRCIDR4.NUMVMIDC \neq 0b0000: The Address Comparator is dependent on the Context Identifier Comparator and Virtual Context Identifier Comparator that TRCACATR<\text{n}>.CONTEXT specifies. The Address Comparator signals a match only if the Context Identifier Comparator, the Virtual Context Identifier Comparator, and address comparison all match.

If TRCIDR4.NUMCIDC == 0b0000, then bit [2] is RES0.
If TRCIDR4.NUMVMIDC == 0b0000, then bit [3] is RES0.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bits [1:0]
Reserved, RES0.

Accessing TRCACATR<\text{n}>

Must be programmed if any of the following are true:

- TRCBBCTLR.RANGE[\text{n}/2] == 1.
- TRCRSCCTR<\text{n}>.GROUP == 0b0100 and TRCRSCTLTR<\text{n}>.SAC[\text{n}] == 1.
- TRCRSCCTR<\text{n}>.GROUP == 0b0101 and TRCRSCTLTR<\text{n}>.ARC[\text{n}/2] == 1.
- TRCVIIIETCTR.EXCLUDE[\text{n}/2] == 1.
- TRCVIIIECTLTR.INCLUDE[\text{n}/2] == 1.
- TRCVISSCTLR.START[\text{n}] == 1.
- TRCVISSCTLR.STOP[\text{n}] == 1.
- TRCQCTLR.RANGE[\text{n}/2] == 1.
- TRCSSCCR<\text{>}_.ARC[\text{n}/2] == 1.
- TRCSSCCR<\text{>}_.SAC[\text{n}] == 1.
- TRCVIUETCTR.RANGE[\text{n}/2] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, TRCACATR<m>

integer m = UInt(op2<0>:CRm<3:1>);
if m >= NUM_TRACE_ADDRESS_COMPARATOR_PAIRS * 2 then
  UNDEFINED;
elsi // PSTATE.EL == EL0 then
  UNDEFINED;
elsi // PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsi // EL2Enabled() && CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsi // EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsi // HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = TRCACATR[m];
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
      UNDEFINED;
elsi // EL2Enabled() && CPTR_EL3.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
elsi // HaveEL(EL3) && CPTR_EL3.TTA == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      else
        X[t, 64] = TRCACATR[m];
  elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = TRCACATR[m];
    else
      MSR TRCACATR<m>, <Xt>

integer m = UInt(op2<0>:CRm<3:1>);
if m >= NUM_TRACE_ADDRESS_COMPARATOR_PAIRS * 2 then
  UNDEFINED;
elsi // PSTATE.EL == EL0 then
  UNDEFINED;
elsi // PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
   UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
   AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
   AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWT_EL2.TRC == '1') then
   AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
   if Halted() && CPACR_EL1.TTA == '1' then
      UNDEFINED;
   elseif EL2Enabled() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
      UNDEFINED;
   elsif CPACR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elseif CPTR_EL3.TTA == '1' then
      AArch64.SystemAccessTrap(EL3, 0x18);
   else
      TRCACATR[m] = X[t, 64];
   end
elsif PSTATE.EL == EL2 then
   if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
      UNDEFINED;
   elseif CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elseif CPTR_EL3.TTA == '1' then
      AArch64.SystemAccessTrap(EL3, 0x18);
   else
      TRCACATR[m] = X[t, 64];
   end
elsif PSTATE.EL == EL3 then
   if CPTR_EL3.TTA == '1' then
      AArch64.SystemAccessTrap(EL3, 0x18);
   else
      TRCACATR[m] = X[t, 64];
   end
D17.4.9 TRCACVR<n>, Address Comparator Value Register <n>, n = 0 - 15

The TRCACVR<n> characteristics are:

**Purpose**
Contains the address value.

**Configurations**
AArch64 System register TRCACVR<n> bits [63:0] are architecturally mapped to External register TRCACVR<n>[63:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR4.NUMACPAIRS) * 2 > n. Otherwise, direct accesses to TRCACVR<n> are UNDEFINED.

**Attributes**
TRCACVR<n> is a 64-bit register.

**Field descriptions**

ADDRESS, bits [63:0]
Address Value.

The Address Comparators can support implementations that use multiple address widths. When the trace unit compares the ADDRESS field with an address that has a width less than this field, then the address must be zero-extended to the ADDRESS field width. The trace unit then compares all implemented bits. For example, in a system that supports both 32-bit and 64-bit addresses, when the PE is in AArch32 state the comparator must zero-extend the 32-bit address and compare against the full 64 bits that are stored in the TRCACVR<n>. This requires that the trace analyzer always programs all implemented bits of the TRCACVR<n>.

The result of writing a value other than all zeros or all ones to ADDRESS at bits[63:P] is an UNKNOWN value, where P is defined as the maximum virtual address size supported by the PE.

The result of writing a value of all zeros or all ones to ADDRESS at bits[63:P] is the written value, and a read of the register returns the written value.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing TRCACVR<n>**
Must be programmed if any of the following are true:

- TRCBBCTLR.RANGE[n/2] == 1.
- TRCRSCTLR<n>.GROUP == 0b0100 and TRCRSCTLR<n>.SAC[n] == 1.
- TRCRSCTLR<n>.GROUP == 0b0101 and TRCRSCTLR<n>.ARC[n/2] == 1.
- TRCVIIECTL.RANGE[n/2] == 1.
- TRCVIIECTL.RANGE[n/2] == 1.
- TRCVISSCTLR.START[n] == 1.
• TRCVISSCTRLR.STOP[n] == 1.
• TRCSSCCR<->.ARC[n/2] == 1.
• TRCSSCCR<->.SAC[n] == 1.
• TRCQCTRLR.RANGE[n/2] == 1.

Writers are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS}< Xt>, \text{TRCACVR}<m> \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0010</td>
<td>m[2:0]:0b0</td>
<td>0b00:m[3]</td>
</tr>
</tbody>
</table>

integer m = UInt(op2<0>:CRm<3:1>);
if m >= NUM_TRACE_ADDRESS_COMPARATOR_PAIRS * 2 then
  UNDEFINED;
elsf if PSTATE.EL == EL0 then
  UNDEFINED;
elsf if PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsf if CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsf if EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsf if EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsf if HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
elsf if CPNR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
elsf else
    X[t, 64] = TRCACVR[m];
elsf if PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsf if CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsf if HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
elsf else
    AArch64.SystemAccessTrap(EL3, 0x18);
elsf else
    X[t, 64] = TRCACVR[m];
elsf if PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
elsf else
    X[t, 64] = TRCACVR[m];
integer m = UInt(op2<0>:CRm<3:1>);

if m >= NUM_TRACE_ADDRESS_COMPARATOR_PAIRS * 2 then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1') then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    TRCACVR[m] = X[t, 64];
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    TRCACVR[m] = X[t, 64];
  end
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCACVR[m] = X[t, 64];
D17.4.10 TRCAUTHSTATUS, Authentication Status Register

The TRCAUTHSTATUS characteristics are:

**Purpose**

 Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

 For additional information, see the CoreSight Architecture Specification.

**Configurations**

AArch64 System register TRCAUTHSTATUS bits [31:0] are architecturally mapped to External register TRCAUTHSTATUS[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCAUTHSTATUS are UNDEFINED.

**Attributes**

TRCAUTHSTATUS is a 64-bit register.

**Field descriptions**

```plaintext
<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>27-26</td>
<td>RTNID, Root non-invasive debug. This field has the same value as DBGAUTHSTATUS_EL1.RTNID.</td>
</tr>
<tr>
<td>25-24</td>
<td>RTID, Root invasive debug. 0b00 Not implemented.</td>
</tr>
<tr>
<td>15-14</td>
<td>RLNID, Realm non-invasive debug. This field has the same value as DBGAUTHSTATUS_EL1.RLNID.</td>
</tr>
<tr>
<td>13-12</td>
<td>RLID, Realm invasive debug. 0b00 Not implemented.</td>
</tr>
<tr>
<td>11-10</td>
<td>HNID, Hyp Non-invasive Debug. Indicates whether a separate enable control for EL2 non-invasive debug features is implemented and enabled. 0b00 Separate Hyp non-invasive debug enable not implemented, or EL2 non-invasive debug features not implemented.</td>
</tr>
</tbody>
</table>
```
0b10  Implemented and disabled.
0b11  Implemented and enabled.

All other values are reserved.

This field reads as 0b00.

HID, bits [9:8]

Hyp Invasive Debug. Indicates whether a separate enable control for EL2 invasive debug features is implemented and enabled.

0b00  Separate Hyp invasive debug enable not implemented, or EL2 invasive debug features not implemented.
0b10  Implemented and disabled.
0b11  Implemented and enabled.

All other values are reserved.

This field reads as 0b00.

SNID, bits [7:6]

Secure Non-invasive Debug. Indicates whether Secure non-invasive debug features are implemented and enabled.

0b00  Secure non-invasive debug features not implemented.
0b10  Implemented and disabled.
0b11  Implemented and enabled.

All other values are reserved.

When EL3 is implemented, this field takes the value 0b10 or 0b11 depending whether Secure non-invasive debug is enabled.

When EL3 is not implemented and the PE is Non-secure, this field reads as 0b00.

When EL3 is not implemented and the PE is Secure, this field takes the value 0b10 or 0b11 depending whether Secure non-invasive debug is enabled.

SID, bits [5:4]

Secure Invasive Debug. Indicates whether Secure invasive debug features are implemented and enabled.

0b00  Secure invasive debug features not implemented.
0b10  Implemented and disabled.
0b11  Implemented and enabled.

All other values are reserved.

This field reads as 0b00.

NSNID, bits [3:2]

Non-secure Non-invasive Debug. Indicates whether Non-secure non-invasive debug features are implemented and enabled.

0b00  Non-secure non-invasive debug features not implemented.
0b10  Implemented and disabled.
0b11  Implemented and enabled.

All other values are reserved.

When EL3 is implemented, this field reads as 0b11.

When EL3 is not implemented and the PE is Non-secure, this field reads as 0b11.

When EL3 is not implemented and the PE is Secure, this field reads as 0b00.
NSID, bits [1:0]

Non-secure Invasive Debug. Indicates whether Non-secure invasive debug features are implemented and enabled.

- 0b00: Non-secure invasive debug features not implemented.
- 0b10: Implemented and disabled.
- 0b11: Implemented and enabled.

All other values are reserved.
This field reads as 0b00.

**Accessing TRCAUTHSTATUS**

For implementations that support multiple access mechanisms, different access mechanisms can return different values for reads of TRCAUTHSTATUS if the authentication signals have changed and that change has not yet been synchronized by a Context synchronization event. This scenario can happen if, for example, the external debugger view is implemented separately from the system instruction view to allow for separate power domains, and so observes changes on the signals differently.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCAUTHSTATUS**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b011</td>
<td>0b110</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCAUTHSTATUS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  else
    X[t, 64] = TRCAUTHSTATUS;
  endif
else
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  else
    X[t, 64] = TRCAUTHSTATUS;
  endif
else
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = TRCAUTHSTATUS;
  endif
endif
else
  X[t, 64] = TRCAUTHSTATUS;
D17.4.11 TRCAUXCTLR, Auxiliary Control Register

The TRCAUXCTLR characteristics are:

Purpose

The function of this register is IMPLEMENTATION DEFINED.

Configurations

AArch64 System register TRCAUXCTLR bits [31:0] are architecturally mapped to External register TRCAUXCTLR[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCAUXCTLR are UNDEFINED.

Attributes

TRCAUXCTLR is a 64-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bits [63:32]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>IMPLEMENTATION DEFINED, bits [31:0]</th>
</tr>
</thead>
</table>

IMPLEMENTATION DEFINED. This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to 0.

Accessing TRCAUXCTLR

If this register is nonzero then it might cause the behavior of a trace unit to contradict this architecture specification. See the documentation of the specific implementation for information about the IMPLEMENTATION DEFINED support for this register.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCAUXCTLR**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then

UNDEFINED;

elsif PSTATE.EL == EL1 then

if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then

UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
  AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRCAUXCTLR == '1') then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end;
elsif PSTATE.EL == EL2 then
  if Halted() && CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      TRCAUXCTLR = X[t, 64];
    end;
  else
    TRCAUXCTLR = X[t, 64];
  end;
elsif PSTATE.EL == EL2 then
  if Halted() && CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCAUXCTLR = X[t, 64];
  end;
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCAUXCTLR = X[t, 64];
  end;
else
  TRCAUXCTLR = X[t, 64];
end;

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>
if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    TRCAUXCLTR = X[t, 64];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRCAUXCLTR = X[t, 64];
D17.4.12 TRCBBCTLR, Branch Broadcast Control Register

The TRCBBCTLR characteristics are:

**Purpose**

Controls the regions in the memory map where branch broadcasting is active.

**Configurations**

AArch64 System register TRCBBCTLR bits [31:0] are architecturally mapped to External register TRCBBCTLR[31:0].

This register is present only when FEAT_ETE is implemented, TRCIDR0.TRCBB == 1 and UInt(TRCIDR4.NUMACPAIRS) > 0. Otherwise, direct accesses to TRCBBCTLR are UNDEFINED.

**Attributes**

TRCBBCTLR is a 64-bit register.

**Field descriptions**

![Diagram of TRCBBCTLR register]

**Bits [63:9]**

Reserved, RES0.

**MODE, bit [8]**

Mode.

- **0b0** Exclude Mode.
  
  Branch broadcasting is not active for instructions in the address ranges defined by TRCBBCTLR.RANGE.
  
  If TRCBBCTLR.RANGE == 0x00 then branch broadcasting is active for all instructions.

- **0b1** Include Mode.
  
  Branch broadcasting is active for instructions in the address ranges defined by TRCBBCTLR.RANGE.
  
  If TRCBBCTLR.RANGE == 0x00 then the behavior of the trace unit is CONSTRAINED UNPREDICTABLE. That is, the trace unit might or might not consider any instructions to be in a branch broadcasting region.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**RANGE[<m>], bit [m], for m = 7 to 0**

Address range field.

Selects whether Address Range Comparator <m> is used with branch broadcasting.

- **0b0** The address range that Address Range Comparator <m> defines, is not selected.

- **0b1** The address range that Address Range Comparator <m> defines, is selected.
This bit is RES0 if m >= TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing TRCBBCTRLR**

Must be programmed if TRCCONFIGR.BB == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCBBCTRLR**

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b111</td>
<td>0b000</td>
</tr>
</tbody>
</table>
```

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
else
    X[t, 64] = TRCBBCTRLR;
else
    if PSTATE.EL == EL3 then
        if CPTR_EL3.TTA == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = TRCBBCTRLR;
        end
else
    X[t, 64] = TRCBBCTRLR;
```

```
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    TRCBBCTRL = X[t, 64];
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    TRCBBCTRL = X[t, 64];
  end
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCBBCTRL = X[t, 64];
D17.4.13 TRCCCCTLR, Cycle Count Control Register

The TRCCCCTLR characteristics are:

Purpose
Set the threshold value for cycle counting.

Configurations
AArch64 System register TRCCCCTLR bits [31:0] are architecturally mapped to External register TRCCCCTLR[31:0].

This register is present only when FEAT_ETE is implemented and TRCIDR0.TRCCCI == 1. Otherwise, direct accesses to TRCCCCTLR are UNDEFINED.

Attributes
TRCCCCTLR is a 64-bit register.

Field descriptions

Bits [63:12]
Reserved, RES0.

THRESHOLD, bits [11:0]
Sets the threshold value for instruction trace cycle counting.

The minimum threshold value that can be programmed into THRESHOLD is given in TRCIDR3.CCITMIN. If the THRESHOLD value is smaller than the value in TRCIDR3.CCITMIN then the behavior is CONSTRAINED UNPREDICTABLE. That is, cycle counts might or might not be included in the trace and the cycle count threshold is not known.

Writing a value of zero when TRCCONFIGR.CCI enables instruction trace cycle counting results in CONSTRAINED UNPREDICTABLE behavior. That is, cycle counts might or might not be included in the trace and the cycle count threshold is not known.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCCCTLR

Must be programmed if TRCCONFIGR.CCI == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, TRCCCTLR

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>op0</strong></td>
<td><strong>op1</strong></td>
<td><strong>CRn</strong></td>
<td><strong>CRm</strong></td>
</tr>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elseif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = TRCCCTLR;
  else
    if PSTATE.EL == EL2 then
      if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
      elseif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL3, 0x18);
        else
          X[t, 64] = TRCCCTLR;
      else
        if PSTATE.EL == EL3 then
          if CPTR_EL3.TTA == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
          else
            X[t, 64] = TRCCCTLR;
          else
            MSR TRCCCTLR, <Xt>
AArch64.SystemAccessTrap(EL2, 0x18);
else if HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  TRCCCCTL = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
else
  TRCCCCTL = X[t, 64];
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  TRCCCCTL = X[t, 64];
D17.4.14 TRCCIDCCTLR0, Context Identifier Comparator Control Register 0

The TRCCIDCCTLR0 characteristics are:

Purpose
Contains Context identifier mask values for the TRCCIDCVR<\text{n}> registers, for \text{n} = 0 to 3.

Configurations
AArch64 System register TRCCIDCCTLR0 bits [31:0] are architecturally mapped to External register TRCCIDCCTLR0[31:0].
This register is present only when FEAT_ETE is implemented, UInt(TRCIDR4.NUMCIDC) > 0x0 and UInt(TRCIDR2.CIDSIZE) > 0. Otherwise, direct accesses to TRCCIDCCTLR0 are UNDEFINED.

Attributes
TRCCIDCCTLR0 is a 64-bit register.

Field descriptions

Bits [63:32]
Reserved, RES0.

COMP3[\text{<m}>], bit [\text{m}+24], for \text{m} = 7 to 0
When UInt(TRCIDR4.NUMCIDC) > 3:
TRCCIDCVR3 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR3. Each bit in this field corresponds to a byte in TRCCIDCVR3.
0b0 The trace unit includes TRCCIDCVR3[(\text{m}×8+7):(\text{m}×8)] when it performs the Context identifier comparison.
0b1 The trace unit ignores TRCCIDCVR3[(\text{m}×8+7):(\text{m}×8)] when it performs the Context identifier comparison.
This bit is RES0 if \text{m} >= TRCIDR2.CIDSIZE.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

COMP2[\text{<m}>], bit [\text{m}+16], for \text{m} = 7 to 0
When UInt(TRCIDR4.NUMCIDC) > 2:
TRCCIDCVR2 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR2. Each bit in this field corresponds to a byte in TRCCIDCVR2.
0b0 The trace unit includes TRCCIDCVR2[(\text{m}×8+7):(\text{m}×8)] when it performs the Context identifier comparison.
0b1 The trace unit ignores TRCCIDCVR2[(\text{m}×8+7):(\text{m}×8)] when it performs the Context identifier comparison.
This bit is RES0 if \text{m} >= TRCIDR2.CIDSIZE.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
D17.4 Trace registers

COMP1[m], bit [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 1:

TRCCIDCVR1 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR1. Each bit in this field corresponds to a byte in TRCCIDCVR1.

- 0b0 The trace unit includes TRCCIDCVR1[(m×8+7):(m×8)] when it performs the Context identifier comparison.
- 0b1 The trace unit ignores TRCCIDCVR1[(m×8+7):(m×8)] when it performs the Context identifier comparison.

This bit is RES0 if m >= TRCIDR2.CIDSIZE.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

COMP0[m], bit [m], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 0:

TRCCIDCVR0 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR0. Each bit in this field corresponds to a byte in TRCCIDCVR0.

- 0b0 The trace unit includes TRCCIDCVR0[(m×8+7):(m×8)] when it performs the Context identifier comparison.
- 0b1 The trace unit ignores TRCCIDCVR0[(m×8+7):(m×8)] when it performs the Context identifier comparison.

This bit is RES0 if m >= TRCIDR2.CIDSIZE.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCIDCCTL0

If software uses the TRCCIDCVR<n> registers, for n = 0 to 3, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCCIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCCIDCVR<n> is not 0x00, the behavior of the Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, TRCCIDCCTL0

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
UNDEFINED;
elif CPACR_EL1.TTA == '1' then
AArch64.SystemAccessTrap(EL1, 0x18);
elif EL2Enabled() && CPTR_EL2.TTA == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGETn == '1') && HDFGRTR_EL2.TRC == '1' then

AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end if;
else
  X[t, 64] = TRCCIDCCTLR0;
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if;
  else
    TRCCIDCCTL0 = X[t, 64];
  end if;
elsif PSTATE_EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCCIDCCTL0 = X[t, 64];
  end if;
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if;
  else
    TRCCIDCCTL0 = X[t, 64];
  end if;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TR6 == '1') then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if;
  else
    TRCCIDCCTL0 = X[t, 64];
  end if;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
end if;


<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>
TRCCIDCCTRL0 = X[t, 64];
elsif PSTATE_EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRCCIDCCTRL0 = X[t, 64];
D17.4.15 TRCCIDCCTLRL1, Context Identifier Comparator Control Register 1

The TRCCIDCCTLRL1 characteristics are:

**Purpose**

Contains Context identifier mask values for the TRCCIDCVR<n> registers, for n = 4 to 7.

**Configurations**

AArch64 System register TRCCIDCCTLRL1 bits [31:0] are architecturally mapped to External register TRCCIDCCTLRL1[31:0].

This register is present only when FEAT_ETE is implemented, UInt(TRCIDR4.NUMCIDC) > 0x4 and UInt(TRCIDR2.CIDSIZE) > 0. Otherwise, direct accesses to TRCCIDCCTLRL1 are UNDEFINED.

**Attributes**

TRCCIDCCTLRL1 is a 64-bit register.

**Field descriptions**

```
<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
<th>RES0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>24</td>
<td>COMP7 [m]</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>COMP6 [m]</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>COMP5 [m]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>COMP4 [m]</td>
</tr>
</tbody>
</table>
```

**Bits [63:32]**

Reserved, RES0.

**COMP7[<m>], bit [m+24], for m = 7 to 0**

When UInt(TRCIDR4.NUMCIDC) > 7:

TRCCIDCVR7 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR7. Each bit in this field corresponds to a byte in TRCCIDCVR7.

0b0 The trace unit includes TRCCIDCVR7[(m×8+7):(m×8)] when it performs the Context identifier comparison.

0b1 The trace unit ignores TRCCIDCVR7[(m×8+7):(m×8)] when it performs the Context identifier comparison.

This bit is RES0 if m >= TRCIDR2.CIDSIZE.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**COMP6[<m>], bit [m+16], for m = 7 to 0**

When UInt(TRCIDR4.NUMCIDC) > 6:

TRCCIDCVR6 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR6. Each bit in this field corresponds to a byte in TRCCIDCVR6.

0b0 The trace unit includes TRCCIDCVR6[(m×8+7):(m×8)] when it performs the Context identifier comparison.

0b1 The trace unit ignores TRCCIDCVR6[(m×8+7):(m×8)] when it performs the Context identifier comparison.

This bit is RES0 if m >= TRCIDR2.CIDSIZE.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
COMP5[<m>], bit [m+8], for m = 7 to 0

When UInt( TRCIDR4.NUMCIDC ) > 5:

TRCCIDCVR5 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR5. Each bit in this field corresponds to a byte in TRCCIDCVR5.

0b0 The trace unit includes TRCCIDCVR5[(m×8+7):(m×8)] when it performs the Context identifier comparison.

0b1 The trace unit ignores TRCCIDCVR5[(m×8+7):(m×8)] when it performs the Context identifier comparison.

This bit is res0 if m >= TRCIDR2.CIDSIZE.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

COMP4[<m>], bit [m], for m = 7 to 0

When UInt( TRCIDR4.NUMCIDC ) > 4:

TRCCIDCVR4 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR4. Each bit in this field corresponds to a byte in TRCCIDCVR4.

0b0 The trace unit includes TRCCIDCVR4[(m×8+7):(m×8)] when it performs the Context identifier comparison.

0b1 The trace unit ignores TRCCIDCVR4[(m×8+7):(m×8)] when it performs the Context identifier comparison.

This bit is res0 if m >= TRCIDR2.CIDSIZE.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCIDCCTLR1

If software uses the TRCCIDCVR<n> registers, for n = 4 to 7, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCCIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCCIDCVR<n> is not 0x00, the behavior of the Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, TRCCIDCCTLR1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0011</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDGFRTR_EL2.TRC == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCCIDCCTLR1;
elsif PSTATE_EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCCIDCCTLR1;
elsif PSTATE_EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCCIDCCTLR1;

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() & CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGWTR_EL2.TRC == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  TRCCIDCCTLR1 = X[t, 64];
elsif PSTATE_EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else

---

**MSR TRCCIDCCTLR1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0011</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

---

ARM DDI 04871.a  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  D17-6575
ID081822  Non-Confidential
TRCIDCCTRL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCIDCCTRL1 = X[t, 64];
D17.4.16 TRCCIDCVR<n>, Context Identifier Comparator Value Registers <n>, n = 0 - 7

The TRCCIDCVR<n> characteristics are:

**Purpose**
Contains a Context identifier value.

**Configurations**
AArch64 System register TRCCIDCVR<n> bits [63:0] are architecturally mapped to External register TRCCIDCVR<n>[63:0].
This register is present only when FEAT_ETE is implemented and UInt(TRCIDR4.NUMCIDC) > n. Otherwise, direct accesses to TRCCIDCVR<n> are UNDEFINED.

**Attributes**
TRCCIDCVR<n> is a 64-bit register.

**Field descriptions**

![Field Descriptions Diagram]

**VALUE, bits [63:0]**
Context identifier value. The width of this field is indicated by TRCIDR2.CIDSIZE. Unimplemented bits are RES0. After a PE Reset, the trace unit assumes that the Context identifier is zero until the PE updates the Context identifier.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing TRCCIDCVR<n>**
Must be programmed if any of the following are true:
• TRCRSCTLR<n>.GROUP == 0b0110 and TRCRSCTLR<n>.CID[n] == 1.
• TRCACATR<n>.CONTEXTTYPE == 0b01 or 0b11 and TRCACATR<n>.CONTEXT == n.
Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.
Accesses to this register use the following encodings in the System register encoding space:

\[\text{MRS <Xt>, TRCCIDCVR<m>}\]

\[
\begin{array}{cccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
\text{0b10} & \text{0b001} & \text{0b0011} & \text{m[2:0]:0b0} & \text{0b000} \\
\end{array}
\]

integer \( m = \text{UInt}(\text{CRm}<3:1>) \);
if \( m >\) NUM_TRACE_CONTEXT_IDENTIFIER_COMPARATORS then UNDEFINED;
elsif PSTATE.EL == EL0 then UNDEFINED;
elsif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && ESCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
   UNDEFINED;
elsif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && ESCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
   UNDEFINED;
elsif PSTATE.EL == EL3 then
if CPTR_EL3.TTA == '1' then
   AArch64.SystemAccessTrap(EL3, 0x18);
else
   X[t, 64] = TRCCIDCVR[m];
else
   MSR TRCCIDCVR<m>, <Xt>

integer m = UInt(CRm<3:1>);
if m >= NUM_TRACE_CONTEXT_IDENTIFIER_COMPARATORS then
   UNDEFINED;
elsif PSTATE.EL == EL0 then
   UNDEFINED;
elsif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && ESCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
   UNDEFINED;
elsif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && ESCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
   UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
   AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
   AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
   if Halted() && ESCR.SDD == '1' then
      UNDEFINED;
   else
      AArch64.SystemAccessTrap(EL3, 0x18);
   else
      X[t, 64] = TRCCIDCVR[m];
else
   if Halted() && HaveEL(EL3) && ESCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
      UNDEFINED;
   elsif CPACR_EL1.TTA == '1' then
      AArch64.SystemAccessTrap(EL1, 0x18);
   elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
      if Halted() && ESCR.SDD == '1' then
         UNDEFINED;
      else
         AArch64.SystemAccessTrap(EL3, 0x18);
      end
   else
      X[t, 64] = TRCCIDCVR[m];
end

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0011</td>
<td>m[2:0]:0b0</td>
<td>0b000</td>
</tr>
</tbody>
</table>
TRCCIDCV[m] = X[t, 64];
elif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'' && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end else
            TRCCIDCV[m] = X[t, 64];
    end else
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRCCIDCV[m] = X[t, 64];
D17.4.17 TRCCLAIMCLR, Claim Tag Clear Register

The TRCCLAIMCLR characteristics are:

**Purpose**

In conjunction with TRCCLAIMSET, provides Claim Tag bits that can be separately set and cleared to indicate whether functionality is in use by a debug agent.

For additional information, see the CoreSight Architecture Specification.

**Configurations**

AArch64 System register TRCCLAIMCLR bits [31:0] are architecturally mapped to External register TRCCLAIMCLR[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCCLAIMCLR are UNDEFINED.

**Attributes**

TRCCLAIMCLR is a 64-bit register.

**Field descriptions**

Bits [63:32]

Reserved, RES0.

CLR[<m>], bit [m], for m = 31 to 0

Claim Tag Clear. Indicates the current status of Claim Tag bit <m>, and is used to clear Claim Tag bit <m> to 0.

0b0 On a read: Claim Tag bit <m> is not set.
On a write: Ignored.

0b1 On a read: Claim Tag bit <m> is set.
On a write: Clear Claim tag bit <m> to 0.

The number of Claim Tag bits implemented is indicated in TRCCLAIMSET.

This bit reads-as-zero and ignores writes if m > the number of Claim Tag bits.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to 0.
Access to this field is W1C.

**Accessing TRCCLAIMCLR**

Accesses to this register use the following encodings in the System register encoding space:

### MRS <Xt>, TRCCLAIMCLR

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0111</td>
<td>0b1001</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
   UNDEFINED;
elsif PSTATE.EL == EL1 then
   if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
      UNDEFINED;
   elsif CPACR_EL1.TTA == '1' then
      AArch64.SystemAccessTrap(EL1, 0x18);
   elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCCLAIM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
      if Halted() && EDSCR.SDD == '1' then
         UNDEFINED;
      else
         AArch64.SystemAccessTrap(EL3, 0x18);
      end if
   else
      X[t, 64] = TRCCLAIMCLR;
   end if
elsif PSTATE.EL == EL2 then
   if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
      UNDEFINED;
   elsif CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
      if Halted() && EDSCR.SDD == '1' then
         UNDEFINED;
      else
         AArch64.SystemAccessTrap(EL3, 0x18);
      end if
   else
      X[t, 64] = TRCCLAIMCLR;
   end if
elsif PSTATE.EL == EL3 then
   if CPTR_EL3.TTA == '1' then
      AArch64.SystemAccessTrap(EL3, 0x18);
   else
      X[t, 64] = TRCCLAIMCLR;
   end if
   MSR TRCCLAIMCLR, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0111</td>
<td>0b1001</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
   UNDEFINED;
elsif PSTATE.EL == EL1 then
   if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
      UNDEFINED;
   elsif CPACR_EL1.TTA == '1' then
      AArch64.SystemAccessTrap(EL1, 0x18);
   elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCCLAIM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
      if Halted() && EDSCR.SDD == '1' then
         UNDEFINED;
      else
         AArch64.SystemAccessTrap(EL3, 0x18);
      end if
   else
      X[t, 64] = TRCCLAIMCLR;
   end if
elsif PSTATE.EL == EL2 then
   if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
      UNDEFINED;
   elsif CPACR_EL1.TTA == '1' then
      AArch64.SystemAccessTrap(EL1, 0x18);
   elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCCLAIM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
      if Halted() && EDSCR.SDD == '1' then
         UNDEFINED;
      else
         AArch64.SystemAccessTrap(EL3, 0x18);
      end if
   else
      X[t, 64] = TRCCLAIMCLR;
   end if
elsif PSTATE.EL == EL3 then
   if CPTR_EL3.TTA == '1' then
      AArch64.SystemAccessTrap(EL3, 0x18);
   else
      X[t, 64] = TRCCLAIMCLR;
   end if
   MSR TRCCLAIMCLR, <Xt>
when SDD == '1' && CPTR_EL3.TTA == '1' then
  UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
  AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRCLAIM == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
else
  TRCLAIMCLR = X[t, 64];
endif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  else
    TRCLAIMCLR = X[t, 64];
  endif
else
  PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      TRCLAIMCLR = X[t, 64];
    endif
  else
    PSTATE.EL == EL1 then
      if CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      else
        TRCLAIMCLR = X[t, 64];
      endif
    else
      PSTATE.EL == EL0 then
        if CPTR_EL1.TTA == '1' then
          AArch64.SystemAccessTrap(EL1, 0x18);
        else
          TRCLAIMCLR = X[t, 64];
        endif
      else
        TRCLAIMCLR = X[t, 64];
      endif
    endif
  endif
else
  TRCLAIMCLR = X[t, 64];
endif
D17.4.18 TRCCLAIMSET, Claim Tag Set Register

The TRCCLAIMSET characteristics are:

Purpose

In conjunction with TRCCLAIMCLR, provides Claim Tag bits that can be separately set and cleared to indicate whether functionality is in use by a debug agent.

For additional information, see the CoreSight Architecture Specification.

Configurations

AArch64 System register TRCCLAIMSET bits [31:0] are architecturally mapped to External register TRCCLAIMSET[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCCLAIMSET are UNDEFINED.

The number of claim tag bits implemented is IMPLEMENTATION DEFINED. Arm recommends that implementations support a minimum of four claim tag bits, that is, SET[3:0] reads as 0b1111.

Attributes

TRCCLAIMSET is a 64-bit register.

Field descriptions

Bits [63:32]

Reserved, RES0.

SET[<m>], bit [m], for m = 31 to 0

Claim Tag Set. Indicates whether Claim Tag bit <m> is implemented, and is used to set Claim Tag bit <m> to 1.

0b0 On a read: Claim Tag bit <m> is not implemented.
     On a write: Ignored.

0b1 On a read: Claim Tag bit <m> is implemented.
     On a write: Set Claim Tag bit <m> to 1.

This bit reads-as-zero and ignores writes if m > the number of Claim Tag bits.

Access to this field is RAO/W1S.
Accessing TRCCLAIMSET

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCCLAIMSET**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() & CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & ((HaveEL(EL3) | SCR_EL3.FGTEn == '1') & HDFGRTR_EL2.TRCLAIM == '1') then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = TRCCLAIMSET;
  elsif PSTATE.EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
      UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
      if Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      else
        X[t, 64] = TRCCLAIMSET;
    elsif PSTATE.EL == EL3 then
      if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
      else
        X[t, 64] = TRCCLAIMSET;
  else
    MSR TRCCLAIMSET, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRCCLAIM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end if;
else
    TRCCLAIMSET = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end if;
    end if;
else
    TRCCLAIMSET = X[t, 64];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRCCLAIMSET = X[t, 64];
D17.4.19 TRCCNTCTLR<\text{n}>, Counter Control Register <\text{n}>, \text{n} = 0 - 3

The TRCCNTCTLR<\text{n}> characteristics are:

Purpose

Controls the operation of Counter <\text{n}>.

Configurations

AArch64 System register TRCCNTCTLR<\text{n}> bits [31:0] are architecturally mapped to External register TRCCNTCTLR<\text{n}>[31:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR5.NUMCNTR) > \text{n}. Otherwise, direct accesses to TRCCNTCTLR<\text{n}> are UNDEFINED.

Attributes

TRCCNTCTLR<\text{n}> is a 64-bit register.

Field descriptions

Bits [63:18]

Reserved, RES0.

CNTCHAIN, bit [17]

For TRCCNTCTLR3 and TRCCNTCTLR1, this field controls whether the Counter decrements when a reload event occurs for Counter <\text{n-1}>.

\[0b0\] The Counter does not decrement when a reload event for Counter <\text{n-1}> occurs.

\[0b1\] Counter <\text{n}> decrements when a reload event for Counter <\text{n-1}> occurs. This concatenates Counter <\text{n}> and Counter <\text{n-1}>, to provide a larger count value.

CNTCHAIN is not implemented for TRCCNTCTLR0 and TRCCNTCTLR2.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

RLDSELF, bit [16]

Controls whether a reload event occurs for the Counter, when the Counter reaches zero.

\[0b0\] Normal mode.

\[0b1\] Self-reload mode.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

RLDEVENT_TYPE, bit [15]

Chooses the type of Resource Selector.
Selects an event, that when it occurs causes a reload event for Counter \(<n>\).

0b0 A single Resource Selector.

\(\text{TRCCNTCTLR}<n>\).RLDEVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

0b1 A Boolean-combined pair of Resource Selectors.

\(\text{TRCCNTCTLR}<n>\).RLDEVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. \(\text{TRCCNTCTLR}<n>\).RLDEVENT.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [14:13]

Reserved, RES0.

**RLDEVENT_SEL, bits [12:8]**

Defines the selected Resource Selector or pair of Resource Selectors.

\(\text{TRCCNTCTLR}<n>\).RLDEVENT.TYPE controls whether \(\text{TRCCNTCTLR}<n>\).RLDEVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

Selects an event, that when it occurs causes a reload event for Counter \(<n>\).

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**CNTEVENT_TYPE, bit [7]**

Chooses the type of Resource Selector.

Selects an event, that when it occurs causes Counter \(<n>\) to decrement.

0b0 A single Resource Selector.

\(\text{TRCCNTCTLR}<n>\).CNTEVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

0b1 A Boolean-combined pair of Resource Selectors.

\(\text{TRCCNTCTLR}<n>\).CNTEVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. \(\text{TRCCNTCTLR}<n>\).CNTEVENT.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

**CNTEVENT_SEL, bits [4:0]**

Defines the selected Resource Selector or pair of Resource Selectors.

\(\text{TRCCNTCTLR}<n>\).CNTEVENT.TYPE controls whether \(\text{TRCCNTCTLR}<n>\).CNTEVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

Selects an event, that when it occurs causes Counter \(<n>\) to decrement.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.
Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

## Accessing TRCCNTCTLR\(n\)

Must be programmed if \(TRCRSCTLR\(n\).GROUP == 0b0010\) and \(TRCRSCTLR\(n\).COUNTERS\(n\) == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

### MRS \(<X_t, TRCCNTCTLR\(m\)>\

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b01:m[1:0]</td>
<td>0b101</td>
</tr>
</tbody>
</table>

integer\( m = \) UInt(CRm<1:0>);

if \( m >= \) NUM_TRACE_COUNTERS then

UNDEFINED;

elsif PSTATE\_.EL == EL0 then

UNDEFINED;

elsif PSTATE\_.EL == EL1 then

if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD != '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& CPTR_EL3.TTA == '1' then

UNDEFINED;

elsif CPACR_EL1.TTA == '1' then

AArch64.SystemAccessTrap(EL1, 0x18);

elsif EL2Enabled() \&\& CPTR_EL2.TTA == '1' then

AArch64.SystemAccessTrap(EL2, 0x18);

elsif EL2Enabled() \&\& (HaveEL(EL3) \&\& SCR_EL3.FGTEn == '1') \&\& HDFGRTR_EL2.TRC == '1' then

AArch64.SystemAccessTrap(EL2, 0x18);

elsif HaveEL(EL3) \&\& CPTR_EL3.TTA == '1' then

if Halted() \&\& EDSCR.SDD == '1' then

UNDEFINED;

else

AArch64.SystemAccessTrap(EL3, 0x18);

else

X[t, 64] = TRCCNTCTLR[m];

elsif PSTATE\_.EL == EL2 then

if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& CPTR_EL3.TTA == '1' then

UNDEFINED;

elsif CPTR_EL2.TTA == '1' then

AArch64.SystemAccessTrap(EL2, 0x18);

elsif CPTR_EL3.TTA == '1' then

if Halted() \&\& EDSCR.SDD == '1' then

UNDEFINED;

else

AArch64.SystemAccessTrap(EL3, 0x18);

else

X[t, 64] = TRCCNTCTLR[m];

elsif PSTATE\_.EL == EL3 then

if CPTR_EL3.TTA == '1' then

AArch64.SystemAccessTrap(EL3, 0x18);

else

X[t, 64] = TRCCNTCTLR[m];
integer \( m = \text{UInt}(\text{CRm}[1:0]) \);

if \( m \geq \text{NUM TRACE COUNTERS} \) then
UNDEFINED;
elsif PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    TRCCNTCTLR[m] = X[t, 64];
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    TRCCNTCTLR[m] = X[t, 64];
  end
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCCNTCTLR[m] = X[t, 64];
end
D17.4.20  TRCCNTRLDVR<n>, Counter Reload Value Register <n>, n = 0 - 3

The TRCCNTRLDVR<n> characteristics are:

Purpose

This sets or returns the reload count value for Counter <n>.

Configurations

AArch64 System register TRCCNTRLDVR<n> bits [31:0] are architecturally mapped to External register TRCCNTRLDVR<n>[31:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR5.NUMCNTR) > n. Otherwise, direct accesses to TRCCNTRLDVR<n> are UNDEFINED.

Attributes

TRCCNTRLDVR<n> is a 64-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>16</td>
</tr>
<tr>
<td>RES0</td>
<td>VALUE</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>

Bits [63:16]

Reserved, RES0.

VALUE, bits [15:0]

Contains the reload value for Counter <n>. When a reload event occurs for Counter <n> then the trace unit copies the VALUE<n> field into Counter <n>.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCNTRLDVR<n>

Must be programmed if TRCRSCTL<rn>.GROUP == 0b0010 and TRCRSCTL<rn>.COUNTERS[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCCNTRLDVR<m>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b00:m[1:0]</td>
<td>0b101</td>
</tr>
</tbody>
</table>

integer m = UInt(CRm<1:0>);

if m >= NUM_TRACE_COUNTERS then
UNDEFINED;
elsif PSTATE_EL == EL0 then
UNDEFINED;
elsif PSTATE_EL == EL1 then
if Halted() && HaveEL(EL3) && ESCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1' && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsf CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsf EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsf EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsf HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
X[t, 64] = TRCCNTRLDVR[m];
elsf PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
elsf CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsf HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
X[t, 64] = TRCCNTRLDVR[m];
elsf PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
X[t, 64] = TRCCNTRLDVR[m];
elsf PSTATE.EL == EL0 then
    UNDEFINED;
elsf PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
elsf CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsf EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsf EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsf HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
X[t, 64] = TRCCNTRLDVR[m];
elsf PSTATE.EL == EL2 then
X[t, 64] = TRCCNTRLDVR[m];

MSR TRCCNTRLDVR<m>, <xt>
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "El3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRCCNTRLDVR[m] = X[t, 64];
    elsif PSTATE.EL == EL3 then
        if CPTR_EL3.TTA == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            TRCCNTRLDVR[m] = X[t, 64];
D17.4.21 TRCCNTVR<n>, Counter Value Register <n>, n = 0 - 3

The TRCCNTVR<n> characteristics are:

**Purpose**

This sets or returns the value of Counter <n>.

**Configurations**

AArch64 System register TRCCNTVR<n> bits [31:0] are architecturally mapped to External register TRCCNTVR<n>[31:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR5.NUMCNTR) > n. Otherwise, direct accesses to TRCCNTVR<n> are UNDEFINED.

**Attributes**

TRCCNTVR<n> is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:16]</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>[15:0]</td>
<td>VALUE</td>
</tr>
</tbody>
</table>

- **VALUE, bits [15:0]**
  - Contains the count value of Counter.
  - The reset behavior of this field is:
    - On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing TRCCNTVR<n>**

Must be programmed if TRCRSCTL<n>.GROUP == 0b0010 and TRCRSCTL<n>.COUNTERS[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCCNTVR<m>**

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b10:m[1:0]</td>
<td>0b101</td>
</tr>
</tbody>
</table>
```

integer m = UInt(CRm<1:0>);

if m >= NUM_TRACE_COUNTERS then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
```
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCCNTVRn == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCCNTVR[m];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
else
    X[t, 64] = TRCCNTVR[m];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCCNTVR[m];
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRCCNTVRn == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
else
    TRCCNTVR[m] = X[t, 64];

integer m = UInt(CRm<1:0>);
if m >= NUM_TRACE_COUNTERS then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRCCNTVRn == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
else
    TRCCNTVR[m] = X[t, 64];

MSR TRCCNTVR<m>, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b10:m[1:0]</td>
<td>0b101</td>
</tr>
</tbody>
</table>
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        TRCCNTVR[m] = X[t, 64];
    else
        if PSTATE_EL == EL3 then
            if CPTR_EL3.TTA == '1' then
                AArch64.SystemAccessTrap(EL3, 0x18);
            else
                TRCCNTVR[m] = X[t, 64];
            end
        end
    end
D17.4.22 TRCCONFIGR, Trace Configuration Register

The TRCCONFIGR characteristics are:

**Purpose**

Controls the tracing options.

**Configurations**

AArch64 System register TRCCONFIGR bits [31:0] are architecturally mapped to External register TRCCONFIGR[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCCONFIGR are UNDEFINED.

**Attributes**

TRCCONFIGR is a 64-bit register.

**Field descriptions**

Bits [63:16]

Reserved, RES0.

VMIOPT, bit [15]

*When TRCIDR2.VMIDOPT == 0b01:*

Virtual context identifier selection control.
- **0b0** VTTBR_EL2.VMID is used as the Virtual context identifier.
- **0b1** CONTEXTIDR_EL2.PROCID is used as the Virtual context identifier.

*When TRCIDR2.VMIDOPT == 0b00:*

Reserved, RES0.

Virtual context identifier selection control.

VTTBR_EL2.VMID is used as the Virtual context identifier.

*When TRCIDR2.VMIDOPT == 0b10:*

Reserved, RES1.

Virtual context identifier selection control.

CONTEXTIDR_EL2.PROCID is used as the Virtual context identifier.

*Otherwise:*

Reserved, RES0.

QE, bits [14:13]

*When TRCIDR0.QSUPP == 0b01:*

Q element generation control.
- **0b00** Q elements are disabled.
- **0b01** Q elements with instruction counts are enabled.
Q elements without instruction counts are disabled.
All other values are reserved.

*When TRCIDR0.QSUPP == 0b10:*

- **0b0**: Q elements are disabled.
- **0b1**: Q elements with instruction counts are enabled.
  Q elements without instruction counts are enabled.

All other values are reserved.

*When TRCIDR0.QSUPP == 0b11:*

- **0b0**: Q elements are disabled.
- **0b1**: Q elements with instruction counts are enabled.
  Q elements without instruction counts are disabled.
- **0b11**: Q elements with instruction counts are enabled.
  Q elements without instruction counts are enabled.

All other values are reserved.

*Otherwise:*

Reserved, RES0.

**RS, bit [12]**

*When TRCIDR0.RETSTACK == 1:*

- **0b0**: Return stack is disabled.
- **0b1**: Return stack is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally **UNKNOWN** value.

*Otherwise:*

Reserved, RES0.

**TS, bit [11]**

*When TRCIDR0.TSIZE != 0b00000:*

- **0b0**: Global timestamp tracing is disabled.
- **0b1**: Global timestamp tracing is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally **UNKNOWN** value.

*Otherwise:*

Reserved, RES0.

**Bits [10:8]**

Reserved, RES0.

**VMID, bit [7]**

*When TRCIDR2.VMIDSIZE != 0b00000:*

- **0b0**: Virtual context identifier tracing is disabled.
- **0b1**: Virtual context identifier tracing is enabled.
The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**
Reserved, RES0.

**CID, bit [6]**

*When **TRCIDR2.CIDSIZE** != 0b00000:*
Context identifier tracing control.
- 0b0  Context identifier tracing is disabled.
- 0b1  Context identifier tracing is enabled.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**
Reserved, RES0.

**Bit [5]**
Reserved, RES0.

**CCI, bit [4]**

*When **TRCIDR0.TRCCCI** == 1:*
Cycle counting instruction tracing control.
- 0b0  Cycle counting instruction tracing is disabled.
- 0b1  Cycle counting instruction tracing is enabled.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**
Reserved, RES0.

**BB, bit [3]**

*When **TRCIDR0.TRCBB** == 1:*
Branch broadcasting control.
- 0b0  Branch broadcasting is disabled.
- 0b1  Branch broadcasting is enabled.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**
Reserved, RES0.

**Bits [2:1]**
Reserved, RES0.

**Bit [0]**
Reserved, RES1.

**Accessing TRCCONFIGR**

Must always be programmed.

TRCCONFIGR.QE must be set to 0b00 if TRCCONFIGR.BB is not 0.

Writes are **CONSTRAINED UNPREDICTABLE** if the trace unit is not in the Idle state.
Accesses to this register use the following encodings in the System register encoding space:

### MRS <Xt>, TRCCONFIGR

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = TRCCONFIGR;
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
      UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      else
        X[t, 64] = TRCCONFIGR;
    elsif PSTATE.EL == EL3 then
      if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
      else
        X[t, 64] = TRCCONFIGR;
  else
    X[t, 64] = TRCCONFIGR;

### MSR TRCCONFIGR, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        TRCCONFIGR = X[t, 64];
    end
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRCCONFIGR = X[t, 64];
    end
elsif PSTATE.EL == EL2 then
    if CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TRCCONFIGR = X[t, 64];
    end
D17.4.23 TRCDEVARCH, Device Architecture Register

The TRCDEVARCH characteristics are:

**Purpose**

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

**Configurations**

AArch64 System register TRCDEVARCH bits [31:0] are architecturally mapped to External register TRCDEVARCH[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCDEVARCH are UNDEFINED.

**Attributes**

TRCDEVARCH is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>31:21</td>
<td>ARCHITECT, bits [31:21]</td>
<td>0b01000111011</td>
</tr>
<tr>
<td>20</td>
<td>PRESENT, bit [20]</td>
<td>0b1</td>
</tr>
<tr>
<td>19:16</td>
<td>REVISION, bits [19:16]</td>
<td></td>
</tr>
<tr>
<td>15:12</td>
<td>ARCHVER, bits [15:12]</td>
<td></td>
</tr>
<tr>
<td>11:0</td>
<td>ARCHPART</td>
<td></td>
</tr>
</tbody>
</table>

Bits [63:32]

Reserved, RES0.

ARCHITECT, bits [31:21]

Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code.

0b01000111011 JEP106 continuation code 0x4, ID code 0x3B. Arm Limited.

Other values are defined by the JEDEC JEP106 standard.

This field reads as 0x23B.

PRESENT, bit [20]

DEVARCH Present. Defines that the DEVARCH register is present.

0b0  Device Architecture information not present.

0b1  Device Architecture information present.

This field reads as 1.

REVISION, bits [19:16]

Revision. Defines the architecture revision of the component.

0b00000  ETEv1.0, FEAT_ETE.

0b00010  ETEv1.1, FEAT_ETEv1p1.

0b00100  ETEv1.2, FEAT_ETEv1p2.

All other values are reserved.

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component.

0b01010  ETEv1.
ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as 0x5.

**ARCHPART, bits [11:0]**

Architecture Part. Defines the architecture of the component.

0xA13 Arm PE trace architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as 0xA13.

**Accessing TRCDEVARCH**

Accesses to this register use the following encodings in the System register encoding space:

*MRS <X>, TRCDEVARCH*

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0111</td>
<td>0b1111</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = TRCDEVARCH;
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = TRCDEVARCH;
  end
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = TRCDEVARCH;
D17.4.24 TRCDEVID, Device Configuration Register

The TRCDEVID characteristics are:

**Purpose**

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

**Configurations**

AArch64 System register TRCDEVID bits [31:0] are architecturally mapped to External register TRCDEVID[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCDEVID are UNDEFINED.

**Attributes**

TRCDEVID is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>32-31</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**Accessing TRCDEVID**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCDEVID**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b011</td>
<td>0b0010</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED “EL3 trap priority when SDD == '1'” && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() && ((HaveEL(EL3) && SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCID == '1') then
    AArch64.SystemAccessTrap(EL2, 0x18);
elif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
else
  AArch64.SystemAccessTrap(EL3, 0x18);
X[t, 64] = TRCDEVID;
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEl(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEl(EL3) & CPTR_EL3.TTA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = TRCDEVID;
  end
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = TRCDEVID;
  end
D17.4.25 TRCEVENTCTL0R, Event Control 0 Register

The TRCEVENTCTL0R characteristics are:

Purpose

Controls the generation of ETEEvents.

Configurations

AArch64 System register TRCEVENTCTL0R bits [31:0] are architecturally mapped to External register TRCEVENTCTL0R[31:0].

This register is present only when FEAT_ETE is implemented and TRCIDR4.NUMRSPAIR != 0b0000. Otherwise, direct accesses to TRCEVENTCTL0R are UNDEFINED.

Attributes

TRCEVENTCTL0R is a 64-bit register.

Field descriptions

Bits [63:32]

Reserved, RES0.

EVENT3_TYPE, bit [31]

When TRCIDR4.NUMRSPAIR != 0b0000 and UInt(TRCIDR0.NUMEVENT) >= 3:

Chooses the type of Resource Selector.

0b0 A single Resource Selector.

TRCEVENTCTL0R.EVENT3.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

0b1 A Boolean-combined pair of Resource Selectors.

TRCEVENTCTL0R.EVENT3.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT3.SEL[4] is RES0.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [30:29]

Reserved, RES0.

EVENT3_SEL, bits [28:24]

When TRCIDR4.NUMRSPAIR != 0b0000 and UInt(TRCIDR0.NUMEVENT) >= 3:

Defines the selected Resource Selector or pair of Resource Selectors.

TRCEVENTCTL0R.EVENT3.TYPE controls whether TRCEVENTCTL0R.EVENT3.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.
If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[3] == 1, then Event element 3 is generated in the instruction trace element stream.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EVENT2_TYPE, bit [23]

When TRCIDR4.NUMRSPAIR != 0b0000 and UInt(TRCIDR0.NUMEVENT) >= 2:

Chooses the type of Resource Selector.

0b0 A single Resource Selector.
 TRCEVENTCTL0R.EVENT2.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

0b1 A Boolean-combined pair of Resource Selectors.
 TRCEVENTCTL0R.EVENT2.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT2.SEL[4] is RES0.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [22:21]

Reserved, RES0.

EVENT2_SEL, bits [20:16]

When TRCIDR4.NUMRSPAIR != 0b0000 and UInt(TRCIDR0.NUMEVENT) >= 2:

Defines the selected Resource Selector or pair of Resource Selectors.

TRCEVENTCTL0R.EVENT2.TYPE controls whether TRCEVENTCTL0R.EVENT2.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[2] == 1, then Event element 2 is generated in the instruction trace element stream.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
EVENT1_TYPE, bit [15]

When TRCIDR4.NUMRSPAIR != 0b0000 and UInt(TRCIDR0.NUMEVENT) >= 1:

Chooses the type of Resource Selector.

0b0  A single Resource Selector.
    TRCEVENTCTL0R.EVENT1.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

0b1  A Boolean-combined pair of Resource Selectors.
    TRCEVENTCTL0R.EVENT1.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT1.SEL[4] is RES0.

The reset behavior of this field is:

*  On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [14:13]

Reserved, RES0.

EVENT1_SEL, bits [12:8]

When TRCIDR4.NUMRSPAIR != 0b0000 and UInt(TRCIDR0.NUMEVENT) >= 1:

Defines the selected Resource Selector or pair of Resource Selectors.

TRCEVENTCTL0R.EVENT1.TYPE controls whether TRCEVENTCTL0R.EVENT1.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[1] == 1, then Event element 1 is generated in the instruction trace element stream.

The reset behavior of this field is:

*  On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EVENT0_TYPE, bit [7]

When TRCIDR4.NUMRSPAIR != 0b0000:

Chooses the type of Resource Selector.

0b0  A single Resource Selector.
    TRCEVENTCTL0R.EVENT0.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

0b1  A Boolean-combined pair of Resource Selectors.
    TRCEVENTCTL0R.EVENT0.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT0.SEL[4] is RES0.

The reset behavior of this field is:

*  On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
Bits [6:5]

Reserved, RES0.

EVENT0_SEL, bits [4:0]

When TRCIDR4.NUMRSPAIR != 0b0000:

Defines the selected Resource Selector or pair of Resource Selectors.

TRCEVENTCTL0R.EVENT0.TYPE controls whether TRCEVENTCTL0R.EVENT0_SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[0] == 1, then Event element 0 is generated in the instruction trace element stream.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TRCEVENTCTL0R

Must be programmed if implemented.

Writes are CONSTRANDED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCEVENTCTL0R**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then

UNDEFINED;

elsif PSTATE_EL == EL1 then

  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" then

    UNDEFINED;

  elseif CPACR_EL1.TTA == '1' then

    AArch64.SystemAccessTrap(EL1, 0x18);

  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then

    AArch64.SystemAccessTrap(EL2, 0x18);

  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRC == '1' then

    AArch64.SystemAccessTrap(EL2, 0x18);

  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then

    if Halted() && EDSCR.SDD == '1' then

      UNDEFINED;

    else

      AArch64.SystemAccessTrap(EL3, 0x18);

    else

      X[t, 64] = TRCEVENTCTL0R;

  elsif PSTATE_EL == EL2 then

    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" then

      UNDEFINED;

    elseif CPTR_EL2.TTA == '1' then

      ...
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCEVENTCTL0R;
elsif PSTATE_EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = TRCEVENTCTL0R;
else
  if PSTATE_EL == EL0 then
    UNDEFINED;
  elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED “EL3 trap priority when SDD == '1'” && CPTR_EL3.TTA == '1' then
      UNDEFINED;
    elsif CPACR_EL1.TTA == '1' then
      AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      else
        TRCEVENTCTL0R = X[t, 64];
  elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
      UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
      TRCEVENTCTL0R = X[t, 64];
  elsif PSTATE_EL == EL3 then
    if CPTR_EL3.TTA == '1' then
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      TRCEVENTCTL0R = X[t, 64];
else
  TRCEVENTCTL0R = X[t, 64];
D17.4.26 TRCEVENTCTL1R, Event Control 1 Register

The TRCEVENTCTL1R characteristics are:

Purpose

Controls the behavior of the ETEEvents that TRCEVENTCTL0R selects.

Configurations

AArch64 System register TRCEVENTCTL1R bits [31:0] are architecturally mapped to External register TRCEVENTCTL1R[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCEVENTCTL1R are UNDEFINED.

Attributes

TRCEVENTCTL1R is a 64-bit register.

Field descriptions

Bits [63:13]

Reserved, RES0.

LPOVERRIDE, bit [12]

When TRCIDR5.LPOVERRIDE == 1:

Low-power Override Mode select.

0b0 Trace unit Low-power Override Mode is not enabled. That is, the trace unit is permitted to enter low-power state.

0b1 Trace unit Low-power Override Mode is enabled. That is, entry to a low-power state does not affect the trace unit resources or trace generation.

Otherwise:

Reserved, RES0.

ATB, bit [11]

When TRCIDR5.ATBTRIG == 1:

AMBA Trace Bus (ATB) trigger enable.

If a CoreSight ATB interface is implemented then when ETEEvent 0 occurs the trace unit sets:

- ATID == 0x7D.
- ATDATA to the value of TRCTRACEIDR.

If the width of ATDATA is greater than the width of TRCTRACEIDR.TRACEID then the trace unit zeros the upper ATDATA bits.

If ETEEvent 0 is programmed to occur based on program execution, such as an Address Comparator, the ATB trigger might not be inserted into the ATB stream at the same time as any trace generated by that program execution is output by the trace unit. Typically, the generated trace might be buffered in a trace unit which means that the ATB trigger would be output before the associated trace is output.
If ETEEvent 0 is asserted multiple times in close succession, the trace unit is required to generate an ATB trigger for the first assertion, but might ignore one or more of the subsequent assertions. Arm recommends that the window in which ETEEvent 0 is ignored is limited only by the time taken to output an ATB trigger.

<table>
<thead>
<tr>
<th>Value</th>
<th>ATB trigger status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>ATB trigger is disabled.</td>
</tr>
<tr>
<td>0b1</td>
<td>ATB trigger is enabled.</td>
</tr>
</tbody>
</table>

**Otherwise:**

Reserved, RES0.

**Bits [10:4]**

Reserved, RES0.

**INSTEN[<m>], bit [m], for m = 3 to 0**

Event element control.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>The trace unit does not generate an Event element &lt;m&gt;.</td>
</tr>
<tr>
<td>0b1</td>
<td>The trace unit generates an Event element &lt;m&gt;.</td>
</tr>
</tbody>
</table>

This bit is RES0 if m >= the number indicated by TRCIDR0.NUMEVENT.

**Accessing TRCEVENTCTL1R**

Must be programmed.

Writes are CONSTRANDED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCEVENTCTL1R**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGTR_EL2.TRC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        elsif EDSCR.SDD == '0' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = TRCEVENTCTL1R;
        end
    else
        X[t, 64] = TRCEVENTCTL1R;
    elsif Halted() && HaveEL(EL3) & EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        elsif EDSCR.SDD == '0' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = TRCEVENTCTL1R;
        end
    else
        X[t, 64] = TRCEVENTCTL1R;
    end

UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = TRCEVENTCTL1R;
elseif PSTATE.EL == EL3 then
if CPTR_EL3.TTA == '1' then
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = TRCEVENTCTL1R;

MSR TRCEVENTCTL1R, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elseif PSTATE.EL == EL1 then
if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" & CPTR_EL3.TTA == '1' then
UNDEFINED;
elseif CPACR_EL1.TTA == '1' then
AArch64.SystemAccessTrap(EL1, 0x18);
elseif EL2Enabled() & CPTR_EL2.TTA == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elseif EL2Enabled() & (!HaveEL(EL3) | | SCR_EL3.FGTEn == '1') & HDFGWTR_EL2.TRC == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elseif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
TRCEVENTCTL1R = X[t, 64];
elseif PSTATE.EL == EL2 then
if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" & CPTR_EL3.TTA == '1' then
UNDEFINED;
elseif CPTR_EL2.TTA == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elseif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
TRCEVENTCTL1R = X[t, 64];
elseif PSTATE.EL == EL3 then
if CPTR_EL3.TTA == '1' then
AArch64.SystemAccessTrap(EL3, 0x18);
else
TRCEVENTCTL1R = X[t, 64];
D17.4.27 TRCEXTINSELR<n>, External Input Select Register <n>, n = 0 - 3

The TRCEXTINSELR<n> characteristics are:

Purpose

Use this to set, or read, which External Inputs are resources to the trace unit.

The name TRCEXTINSELR is an alias of TRCEXTINSELR0.

Configurations

AArch64 System register TRCEXTINSELR<n> bits [31:0] are architecturally mapped to External register TRCEXTINSELR<n>[31:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR5.NUMEXTINSEL) > n. Otherwise, direct accesses to TRCEXTINSELR<n> are UNDEFINED.

Attributes

TRCEXTINSELR<n> is a 64-bit register.

Field descriptions

![Field diagram]

Bits [63:16]

Reserved, RES0.

evtCount, bits [15:0]

PMU event to select.

The event number as defined by the Arm ARM.

Software must program this field with a PMU event that is supported by the PE being programmed.

There are three ranges of PMU event numbers:

- PMU event numbers in the range 0x0000 to 0x003F are common architectural and microarchitectural events.
- PMU event numbers in the range 0x0040 to 0x00BF are Arm recommended common architectural and microarchitectural PMU events.
- PMU event numbers in the range 0x00C0 to 0x03FF are IMPLEMENTATION DEFINED PMU events.

If evtCount is programmed to a PMU event that is reserved or not supported by the PE, the behavior depends on the PMU event type:

- For the range 0x0000 to 0x003F, then the PMU event is not active, and the value returned by a direct or external read of the evtCount field is the value written to the field.
- For IMPLEMENTATION DEFINED PMU events, it is UNPREDICTABLE what PMU event, if any, is counted, and the value returned by a direct or external read of the evtCount field is UNKNOWN.

UNPREDICTABLE means the PMU event must not expose privileged information.

Arm recommends that the behavior across a family of implementations is defined such that if a given implementation does not include a PMU event from a set of common IMPLEMENTATION DEFINED PMU events, then no PMU event is counted and the value read back on evtCount is the value written.
The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally **UNKNOWN** value.

**Accessing TRCEXTINSELR<n>**

Must be programmed if any of the following is true: TRCRSCTLR<n>.GROUP == 0b0000 and TRCRSCTLR<n>.EXTIN[n] == 1.

Writes are **CONSTRAINED UNPREDICTABLE** if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCEXTINSELR<m>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b10:m[1:0]</td>
<td>0b100</td>
</tr>
</tbody>
</table>

```
integer m = UInt(CRm<1:0>);

if m >= NUM_TRACE_EXTERNAL_INPUT_SELECTOR_RESOURCES then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = TRCEXTINSELR[m];
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = TRCEXTINSELR[m];
  end
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = TRCEXTINSELR[m];
```
MSR TRCEXTINSEL[m], <Xt>

integer m = UInt(CRm<1:0>);

if m >= NUM_TRACE_EXTERNAL_INPUT_SELECTOR_RESOURCES then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) && SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      TRCEXTINSEL[m] = X[t, 64];
  elseif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
      UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      else
        TRCEXTINSEL[m] = X[t, 64];
    elseif PSTATE.EL == EL3 then
      if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
      else
        TRCEXTINSEL[m] = X[t, 64];
      else
        TRCEXTINSEL[m] = X[t, 64];
### D17.4.28 TRCIDR0, ID Register 0

The TRCIDR0 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

The AArch64 System register TRCIDR0 bits [31:0] are architecturally mapped to External register TRCIDR0[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR0 are UNDEFINED.

**Attributes**

TRCIDR0 is a 64-bit register.

**Field descriptions**

**Bits [63:31]**

Reserved, RES0.

**COMMTRANS, bit [30]**

Transaction Start element behavior.

0: Transaction Start elements are P0 elements.

1: Transaction Start elements are not P0 elements.

**COMMOPT, bit [29]**

Indicates the contents and encodings of Cycle count packets.

0: Commit mode 0.

1: Commit mode 1.

The Commit mode defines the contents and encodings of Cycle Count packets, in particular how Commit elements are indicated by these packets. See the descriptions of these packets for more details.

Accessing this field has the following behavior:

- RAO/WI if all of the following are true:
  - TRCIDR0.TRCCCI == 1
  - UInt(TRCIDR8.MAXSPEC) == 0x0
- When TRCIDR0.TRCCCI == 0, access to this field is RAZ/WI.
- Otherwise, access to this field is RO
TSSIZE, bits [28:24]
Indicates that the trace unit implements Global timestamping and the size of the timestamp value.
0b00000  Global timestamping not implemented.
0b01000  Global timestamping implemented with a 64-bit timestamp value.
All other values are reserved.
This field reads as 0b01000.

TSMARK, bit [23]
*When FEAT_ETEv1p1 is implemented:*
Indicates whether Timestamp Marker elements are generated.
0b0  Timestamp Marker elements are not generated.
0b1  Timestamp Marker elements are generated.
*Otherwise:*
Reserved, RES0.

Bits [22:18]
Reserved, RES0.

TRCEXDATA, bit [17]
*When TRCIDR0.TRCDATA != 0b00:*
Indicates if the trace unit implements tracing of data transfers for exceptions and exception returns.
Data tracing is not implemented in ETE and this field is reserved for other trace architectures.
Allocated in other trace architectures.
0b0  Tracing of data transfers for exceptions and exception returns not implemented.
0b1  Tracing of data transfers for exceptions and exception returns implemented.
*Otherwise:*
Reserved, RES0.

QSUPP, bits [16:15]
Indicates that the trace unit implements Q element support.
0b0  Q element support is not implemented.
0b1  Q element support is implemented, and only supports Q elements with instruction counts.
0b10 Q element support is implemented, and only supports Q elements without instruction counts.
0b11 Q element support is implemented, and supports:
•  Q elements with instruction counts.
•  Q elements without instruction counts.

QFILT, bit [14]
Indicates if the trace unit implements Q element filtering.
0b0  Q element filtering is not implemented.
0b1  Q element filtering is implemented.
If TRCIDR0.QSUPP == 0b0 then this field is 0.
CONDTYPE, bits [13:12]

When TRCIDR0.TRCCOND == 1:
Indicates how conditional instructions are traced. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

0b00  Conditional instructions are traced with an indication of whether they pass or fail their condition code check.
0b01  Conditional instructions are traced with an indication of the APSR condition flags.
All other values are reserved.

Otherwise:
Reserved, RES0.

NUMEVENT, bits [11:10]

When TRCIDR4.NUMRSPAIR == 0b0000:
Indicates the number of ETEEvents implemented.

0b00  The trace unit supports 0 ETEEvents.
All other values are reserved.

When TRCIDR4.NUMRSPAIR != 0b0000:
Indicates the number of ETEEvents implemented.

0b00  The trace unit supports 1 ETEEvent.
0b01  The trace unit supports 2 ETEEvents.
0b10  The trace unit supports 3 ETEEvents.
0b11  The trace unit supports 4 ETEEvents.

Otherwise:
Reserved, RES0.

RETSTACK, bit [9]
Indicates if the trace unit supports the return stack.

0b0  Return stack not implemented.
0b1  Return stack implemented.

Bit [8]
Reserved, RES0.

TRCCCI, bit [7]
Indicates if the trace unit implements cycle counting.

0b0  Cycle counting not implemented.
0b1  Cycle counting implemented.
This field reads as 1.

TRCCOND, bit [6]
Indicates if the trace unit implements conditional instruction tracing. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures.

0b0  Conditional instruction tracing not implemented.
0b1  Conditional instruction tracing implemented.
This field reads as 0.

TRCBB, bit [5]
Indicates if the trace unit implements branch broadcasting.

0b0  Branch broadcasting not implemented.
Branch broadcasting implemented.
This field reads as 1.

TRCDATA, bits [4:3]
Indicates if the trace unit implements data tracing. Data tracing is not implemented in ETE and this field is reserved for other trace architectures.

0b00  Data tracing not implemented.
0b11  Data tracing implemented.
All other values are reserved.
This field reads as 0b00.

INSTP0, bits [2:1]
Indicates if load and store instructions are P0 instructions. Load and store instructions as P0 instructions is not implemented in ETE and this field is reserved for other trace architectures.

0b00  Load and store instructions are not P0 instructions.
0b11  Load and store instructions are P0 instructions.
All other values are reserved.
This field reads as 0b00.

Bit [0]
Reserved, RES1.

Accessing TRCIDR0
Accesses to this register use the following encodings in the System register encoding space:

|MRS <Xt>, TRCIDR0|
|---|---|---|---|---|
| op0 | op1 | CRn | CRm | op2 |
| 0b10 | 0b001 | 0b0000 | 0b1000 | 0b111 |

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCID == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = TRCIDR0;
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
      UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
    if Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end;
else
    X[t, 64] = TRCIDR0;
eelsif PSTATE_EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = TRCIDR0;
D17.4.29 TRCIDR1, ID Register 1

The TRCIDR1 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configurations

AArch64 System register TRCIDR1 bits [31:0] are architecturally mapped to External register TRCIDR1[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR1 are UNDEFINED.

Attributes

TRCIDR1 is a 64-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:32]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[31:24]</td>
<td>DESIGNER, bits [31:24]</td>
</tr>
<tr>
<td>[23:16]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[3:0]</td>
<td>REVISION, bits [3:0]</td>
</tr>
</tbody>
</table>

Bits [63:32]

Reserved, RES0.

DESIGNER, bits [31:24]

Indicates which company designed the trace unit. The permitted values of this field are the same as MIDR_EL1.Implementer.

Bits [23:16]

Reserved, RES0.

Bits [15:12]

Reserved, RES1.

TRCARCHMAJ, bits [11:8]

Major architecture version.

0b1111 If both TRCARCHMAJ and TRCARCHMIN == 0xf then refer to TRCDEVARCH.

All other values are reserved.

This field reads as 0b1111.

TRCARCHMIN, bits [7:4]

Minor architecture version.

0b1111 If both TRCARCHMAJ and TRCARCHMIN == 0xf then refer to TRCDEVARCH.

All other values are reserved.

This field reads as 0b1111.

REVISION, bits [3:0]

Implementation revision.

Returns an IMPLEMENTATION DEFINED value that identifies the revision of the trace unit.
Accessing TRCIDR1

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCIDR1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1001</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elseif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() & CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGRTR_EL2.TRCID == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
        if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        X[t, 64] = TRCIDR1;
    end
elsif PSTATE.EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elseif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
        if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        X[t, 64] = TRCIDR1;
    end
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = TRCIDR1;
    end
else
    X[t, 64] = TRCIDR1;
end
D17.4.30 TRCIDR10, ID Register 10

The TRCIDR10 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

AArch64 System register TRCIDR10 bits [31:0] are architecturally mapped to External register TRCIDR10[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR10 are UNDEFINED.

**Attributes**

TRCIDR10 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:32]</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>[31:0]</td>
<td>NUMP1KEY</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**NUMP1KEY, bits [31:0]**

*When TRCIDR0.TRCDATA != 0b00:*

Indicates the number of P1 right-hand keys. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

*Otherwise:*

Reserved, RES0.

**Accessing TRCIDR10**

Accesses to this register use the following encodings in the System register encoding space:

*MRS <Xt>, TRCIDR10*

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b110</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCID == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = TRCIDR10;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCIDR10;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCIDR10;
D17.4.31 TRCIDR11, ID Register 11

The TRCIDR11 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

AArch64 System register TRCIDR11 bits [31:0] are architecturally mapped to External register TRCIDR11[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR11 are UNDEFINED.

**Attributes**

TRCIDR11 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:32]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMP1SPC, bits [31:0]</td>
<td>When TRCIDR0.TRCDATA != 0b00:</td>
</tr>
<tr>
<td></td>
<td>Indicates the number of special P1 right-hand keys. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.</td>
</tr>
<tr>
<td></td>
<td>Otherwise:</td>
</tr>
<tr>
<td></td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**Accessing TRCIDR11**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCIDR11**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0011</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elseif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCID == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCIDR11;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCIDR11;
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCIDR11;
D17.4.32  TRCIDR12, ID Register 12

The TRCIDR12 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

AArch64 System register TRCIDR12 bits [31:0] are architecturally mapped to External register TRCIDR12[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR12 are UNDEFINED.

**Attributes**

TRCIDR12 is a 64-bit register.

**Field descriptions**

```
+---+---+---+
|   |   |   |
| 31| 0  |   |
+---+---+---+
|   |   |   |
| RES0 | NUMCONDKEY |
```

**Bits [63:32]**

Reserved, RES0.

**NUMCONDKEY, bits [31:0]**

*When TRCIDR0.TRCCOND == 1:*

Indicates the number of conditional instruction right-hand keys. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

*Otherwise:*

Reserved, RES0.

**Accessing TRCIDR12**

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TRCIDR12
```

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0100</td>
<td>0b110</td>
</tr>
</tbody>
</table>

```
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEl(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
```
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCID == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end;
else
  X[t, 64] = TRCIDR12;
elsif PSTATE.EL == EL2 then
  X[t, 64] = TRCIDR12;
elsif PSTATE.EL == EL3 then
  X[t, 64] = TRCIDR12;
elsif PSTATE.EL == EL2 then
  X[t, 64] = TRCIDR12;
elsif PSTATE.EL == EL3 then
  X[t, 64] = TRCIDR12;
D17.4.33 TRCIDR13, ID Register 13

The TRCIDR13 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

AArch64 System register TRCIDR13 bits [31:0] are architecturally mapped to External register TRCIDR13[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR13 are UNDEFINED.

**Attributes**

TRCIDR13 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>31:0</td>
<td>NUMCONDSPC</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**NUMCONDSPC, bits [31:0]**

*When TRCIDR6.TRCCOND == 1:*

Indicates the number of special conditional instruction right-hand keys. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

*Otherwise:*

Reserved, RES0.

**Accessing TRCIDR13**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCIDR13**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0101</td>
<td>0b110</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(E13) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCID == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = TRCIDR13;
else if PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      if PSTATE_EL == EL3 then
        if CPTR_EL3.TTA == '1' then
          AArch64.SystemAccessTrap(EL3, 0x18);
        else
          X[t, 64] = TRCIDR13;
        end
      else
        X[t, 64] = TRCIDR13;
      end
    end
  else
    X[t, 64] = TRCIDR13;
  end
### D17.4.34 TRCIDR2, ID Register 2

The TRCIDR2 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

AArch64 System register TRCIDR2 bits [31:0] are architecturally mapped to External register TRCIDR2[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR2 are undefined.

**Attributes**

TRCIDR2 is a 64-bit register.

**Field descriptions**

![Field Diagram]

- **Bits [63:32]**
  - Reserved, RES0.

- **WFXMODE, bit [31]**
  - Indicates whether WFI, WFIT, WFE, and WFET instructions are classified as P0 instructions:
    - 0b0: WFI, WFIT, WFE, and WFET instructions are not classified as P0 instructions.
    - 0b1: WFI, WFIT, WFE, and WFET instructions are classified as P0 instructions.

- **VMIDOPT, bits [30:29]**
  - Indicates the options for Virtual context identifier selection.
    - 0b00: Virtual context identifier selection not supported. TRCCONFIGR.VMIDOPT is RES0.
    - 0b01: Virtual context identifier selection supported. TRCCONFIGR.VMIDOPT is implemented.
    - 0b10: Virtual context identifier selection not supported. TRCCONFIGR.VMIDOPT is RES1.
    - All other values are reserved.

- **CCSIZE, bits [28:25]**
  - Indicates the size of the cycle counter.
    - 0b0000: The cycle counter is 12 bits in length.
    - 0b0001: The cycle counter is 13 bits in length.
    - 0b0010: The cycle counter is 14 bits in length.
    - 0b0011: The cycle counter is 15 bits in length.
    - 0b0100: The cycle counter is 16 bits in length.
0b0101  The cycle counter is 17 bits in length.
0b0110  The cycle counter is 18 bits in length.
0b0111  The cycle counter is 19 bits in length.
0b1000  The cycle counter is 20 bits in length.
All other values are reserved.

Otherwise:
Reserved, RES0.

DVSIZE, bits [24:20]
When TRCIDR0.TRCDATA != 0b00:
Indicates the data value size in bytes. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.
0b00000  Data value tracing not implemented.
0b00100  Data value tracing has a maximum of 32-bit data values.
0b01000  Data value tracing has a maximum of 64-bit data values.
All other values are reserved.

Otherwise:
Reserved, RES0.

DASIZE, bits [19:15]
When TRCIDR0.TRCDATA != 0b00:
Indicates the data address size in bytes. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.
0b00000  Data address tracing not implemented.
0b00100  Data address tracing has a maximum of 32-bit data addresses.
0b01000  Data address tracing has a maximum of 64-bit data addresses.
All other values are reserved.

Otherwise:
Reserved, RES0.

VMIDSIZE, bits [14:10]
Indicates the trace unit Virtual context identifier size.
0b00000  Virtual context identifier tracing is not supported.
0b00001  8-bit Virtual context identifier size.
0b00010  16-bit Virtual context identifier size.
0b00100  32-bit Virtual context identifier size.
All other values are reserved.
If the PE does not implement EL2 then this field is 0b00000.
If the PE implements EL2 then this field is 0b00100.

CIDSIZE, bits [9:5]
Indicates the Context identifier size.
0b00000  Context identifier tracing is not supported.
0b00100  32-bit Context identifier size.
All other values are reserved.
This field reads as 0b00100.
IASIZE, bits [4:0]

Virtual instruction address size.

0b00100  Maximum of 32-bit instruction address size.
0b01000  Maximum of 64-bit instruction address size.
All other values are reserved.
This field reads as 0b01000.

Accessing TRCIDR2

Accesses to this register use the following encodings in the System register encoding space:

**MRS <x>, TRCIDR2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1010</td>
<td>0b11</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
   UNDEFINED;
elsif PSTATE.EL == EL1 then
   if Halted() & & HaveEl(EL3) & & EDSCR.SDD == '1' & & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & & CPTR_EL3.TTA == '1' then
      UNDEFINED;
   elsif CPACR_EL1.TTA == '1' then
      AArch64.SystemAccessTrap(EL1, 0x18);
   elsif EL2Enabled() & & CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif EL2Enabled() & & (!HaveEl(EL3) || SCR_EL3.FGTEn == '1') & & HDFGRTR_EL2.TRCD == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   elsif HaveEl(EL3) & & CPTR_EL3.TTA == '1' then
      if Halted() & & EDSCR.SDD == '1' then
         UNDEFINED;
      else
         AArch64.SystemAccessTrap(EL3, 0x18);
      else
         X[t, 64] = TRCIDR2;
   else
      PSTATE.EL == EL2 then
         if Halted() & & HaveEl(EL3) & & EDSCR.SDD == '1' & & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & & CPTR_EL3.TTA == '1' then
            UNDEFINED;
         elsif CPTR_EL2.TTA == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
         elsif HaveEl(EL3) & & CPTR_EL3.TTA == '1' then
            if Halted() & & EDSCR.SDD == '1' then
               UNDEFINED;
            else
               AArch64.SystemAccessTrap(EL3, 0x18);
            else
               X[t, 64] = TRCIDR2;
         elsif PSTATE.EL == EL3 then
            if CPTR_EL3.TTA == '1' then
               AArch64.SystemAccessTrap(EL3, 0x18);
            else
               X[t, 64] = TRCIDR2;
D17.4.35  TRCIDR3, ID Register 3

The TRCIDR3 characteristics are:

**Purpose**

Returns the base architecture of the trace unit.

**Configurations**

AArch64 System register TRCIDR3 bits [31:0] are architecturally mapped to External register TRCIDR3[31:0]. This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR3 are UNDEFINED.

**Attributes**

TRCIDR3 is a 64-bit register.

**Field descriptions**

Bits [63:32]  
Reserved, RES0.

**NOOVERFLOW, bit [31]**

Indicates if overflow prevention is implemented.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Overflow prevention is not implemented.</td>
</tr>
<tr>
<td>0b1</td>
<td>Overflow prevention is implemented.</td>
</tr>
</tbody>
</table>

If TRCIDR3.STALLCTL == 0 then this field is 0.

**NUMPROC, bits [13:12, 30:28]**

Indicates the number of PEs available for tracing.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00000</td>
<td>The trace unit can trace one PE.</td>
</tr>
</tbody>
</table>

This field reads as 0b00000.

The NUMPROC field is split as follows:

* NUMPROC[2:0] is TRCIDR3[30:28].
* NUMPROC[4:3] is TRCIDR3[13:12].

**SYSSTALL, bit [27]**

Indicates if stalling of the PE is permitted.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Stalling of the PE is not permitted.</td>
</tr>
<tr>
<td>0b1</td>
<td>Stalling of the PE is permitted.</td>
</tr>
</tbody>
</table>
The value of this field might be dynamic and change based on system conditions.
If TRCIDR3.STALLCTL == 0 then this field is 0.

STALLCTL, bit [26]
Indicates if trace unit implements stalling of the PE.
0b0 Stalling of the PE is not implemented.
0b1 Stalling of the PE is implemented.

SYNCPR, bit [25]
Indicates if an implementation has a fixed synchronization period.
0b0 TRCSYNPR is read/write so software can change the synchronization period.
0b1 TRCSYNPR is read-only so the synchronization period is fixed.
This field reads as 0.

TRCERR, bit [24]
Indicates forced tracing of System Error exceptions is implemented.
0b0 Forced tracing of System Error exceptions is not implemented.
0b1 Forced tracing of System Error exceptions is implemented.
This field reads as 1.

Bit [23]
Reserved, RES0.

EXLEVEL_NS_EL2, bit [22]
Indicates if Non-secure EL2 is implemented.
0b0 Non-secure EL2 is not implemented.
0b1 Non-secure EL2 is implemented.

EXLEVEL_NS_EL1, bit [21]
Indicates if Non-secure EL1 is implemented.
0b0 Non-secure EL1 is not implemented.
0b1 Non-secure EL1 is implemented.

EXLEVEL_NS_EL0, bit [20]
Indicates if Non-secure EL0 is implemented.
0b0 Non-secure EL0 is not implemented.
0b1 Non-secure EL0 is implemented.

EXLEVEL_S_EL3, bit [19]
Indicates if EL3 is implemented.
0b0 EL3 is not implemented.
0b1 EL3 is implemented.

EXLEVEL_S_EL2, bit [18]
Indicates if Secure EL2 is implemented.
0b0 Secure EL2 is not implemented.
0b1 Secure EL2 is implemented.

EXLEVEL_S_EL1, bit [17]
Indicates if Secure EL1 is implemented.
0b0 Secure EL1 is not implemented.
Secure EL1 is implemented.

EXLEVEL_S_EL0, bit [16]
Indicates if Secure EL0 is implemented.
0b0 Secure EL0 is not implemented.
0b1 Secure EL0 is implemented.

Bits [15:14]
Reserved, RES0.

CCITMIN, bits [11:0]
Indicates the minimum value that can be programmed in TRCCCTLR.THRESHOLD.
If TRCIDR0.TRCCCI == 1 then the minimum value of this field is 0x001.
If TRCIDR0.TRCCCI == 0 then this field is zero.

Accessing TRCIDR3
Accesses to this register use the following encodings in the System register encoding space:

|MRS <Xt>, TRCIDR3|
|---|---|---|---|---|
|op0|op1|CRn|CRm|op2|
|0b10|0b001|0b0000|0b1011|0b111|

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && &CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCID == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && &CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  end
  X[t, 64] = TRCIDR3;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && &CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  end
  X[t, 64] = TRCIDR3;
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCIDR3;
D17.4.36 TRCIDR4, ID Register 4

The TRCIDR4 characteristics are:

**Purpose**
Returns the tracing capabilities of the trace unit.

**Configurations**
AArch64 System register TRCIDR4 bits [31:0] are architecturally mapped to External register TRCIDR4[31:0].
This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR4 are UNDEFINED.

**Attributes**
TRCIDR4 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:32]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[31:28]</td>
<td>NUMVMIDC, bits [31:28]</td>
</tr>
<tr>
<td>[23:16]</td>
<td>NUMSSCC</td>
</tr>
<tr>
<td>[15:12]</td>
<td>NUMPC, bits [15:12]</td>
</tr>
<tr>
<td>[8:16]</td>
<td>NUMDVC, bits [8:16]</td>
</tr>
<tr>
<td>[19:16]</td>
<td>SUPPDAC</td>
</tr>
<tr>
<td>[11:8]</td>
<td>NUMRSPAIR</td>
</tr>
<tr>
<td>[30:30]</td>
<td>NUMACPAIRS</td>
</tr>
</tbody>
</table>

**Bits [63:32]**
Reserved, RES0.

**NUMVMIDC, bits [31:28]**
Indicates the number of Virtual Context Identifier Comparators that are available for tracing.
- 00000  No Virtual Context Identifier Comparators are available.
- 00001  The implementation has one Virtual Context Identifier Comparator.
- 00010  The implementation has two Virtual Context Identifier Comparators.
- 00011  The implementation has three Virtual Context Identifier Comparators.
- 00100  The implementation has four Virtual Context Identifier Comparators.
- 00101  The implementation has five Virtual Context Identifier Comparators.
- 00110  The implementation has six Virtual Context Identifier Comparators.
- 00111  The implementation has seven Virtual Context Identifier Comparators.
- 01000  The implementation has eight Virtual Context Identifier Comparators.
All other values are reserved.

**NUMCIDC, bits [27:24]**
Indicates the number of Context Identifier Comparators that are available for tracing.
- 00000  No Context Identifier Comparators are available.
- 00001  The implementation has one Context Identifier Comparator.
- 00010  The implementation has two Context Identifier Comparators.
- 00011  The implementation has three Context Identifier Comparators.
- 00100  The implementation has four Context Identifier Comparators.
- 00101  The implementation has five Context Identifier Comparators.
- 00110  The implementation has six Context Identifier Comparators.
0b0111 The implementation has seven Context Identifier Comparators.
0b1000 The implementation has eight Context Identifier Comparators.

All other values are reserved.

NUMSSCC, bits [23:20]
Indicates the number of Single-shot Comparator Controls that are available for tracing.
0b0000 No Single-shot Comparator Controls are available.
0b0001 The implementation has one Single-shot Comparator Control.
0b0010 The implementation has two Single-shot Comparator Controls.
0b0011 The implementation has three Single-shot Comparator Controls.
0b0100 The implementation has four Single-shot Comparator Controls.
0b0101 The implementation has five Single-shot Comparator Controls.
0b0110 The implementation has six Single-shot Comparator Controls.
0b0111 The implementation has seven Single-shot Comparator Controls.
0b1000 The implementation has eight Single-shot Comparator Controls.

All other values are reserved.

NUMRSPAIR, bits [19:16]
Indicates the number of resource selector pairs that are available for tracing.
0b0000 The implementation has zero resource selectors.
0b0001 The implementation has two resource selector pairs.
0b0010 The implementation has three resource selector pairs.
0b0011 The implementation has four resource selector pairs.
0b0100 The implementation has five resource selector pairs.
0b0101 The implementation has six resource selector pairs.
0b0110 The implementation has seven resource selector pairs.
0b0111 The implementation has eight resource selector pairs.
0b1000 The implementation has nine resource selector pairs.
0b1001 The implementation has ten resource selector pairs.
0b1010 The implementation has eleven resource selector pairs.
0b1011 The implementation has twelve resource selector pairs.
0b1100 The implementation has thirteen resource selector pairs.
0b1101 The implementation has fourteen resource selector pairs.
0b1110 The implementation has fifteen resource selector pairs.
0b1111 The implementation has sixteen resource selector pairs.

All other values are reserved.

NUMPC, bits [15:12]
Indicates the number of PE Comparator Inputs that are available for tracing.
0b0000 No PE Comparator Inputs are available.
0b0001 The implementation has one PE Comparator Input.
0b0010 The implementation has two PE Comparator Inputs.
0b0011 The implementation has three PE Comparator Inputs.
0b0100 The implementation has four PE Comparator Inputs.
0b0101 The implementation has five PE Comparator Inputs.
0b0110 The implementation has six PE Comparator Inputs.
0b0111  The implementation has seven PE Comparator Inputs.
0b1000  The implementation has eight PE Comparator Inputs.
All other values are reserved.

Bits [11:9]

Reserved, RES0.

SUPPDAC, bit [8]

When TRCIDR4.NUMACPAIRS != 0b0000:
Indicates whether data address comparisons are implemented. Data address comparisons are not
implemented in ETE and are reserved for other trace architectures. Allocated in other trace
architectures.
0b0  Data address comparisons not implemented.
0b1  Data address comparisons implemented.
This field reads as 0.

Otherwise:
Reserved, RES0.

NUMDVC, bits [7:4]
Indicates the number of data value comparators. Data value comparators are not implemented in
ETE and are reserved for other trace architectures. Allocated in other trace architectures.
0b0000  No data value comparators implemented.
0b0001  One data value comparator implemented.
0b0010  Two data value comparators implemented.
0b0011  Three data value comparators implemented.
0b0100  Four data value comparators implemented.
0b0101  Five data value comparators implemented.
0b0110  Six data value comparators implemented.
0b0111  Seven data value comparators implemented.
0b1000  Eight data value comparators implemented.
All other values are reserved.
This field reads as 0b0000.

NUMACPAIRS, bits [3:0]
Indicates the number of Address Comparator pairs that are available for tracing.
0b0000  No Address Comparator pairs are available.
0b0001  The implementation has one Address Comparator pair.
0b0010  The implementation has two Address Comparator pairs.
0b0011  The implementation has three Address Comparator pairs.
0b0100  The implementation has four Address Comparator pairs.
0b0101  The implementation has five Address Comparator pairs.
0b0110  The implementation has six Address Comparator pairs.
0b0111  The implementation has seven Address Comparator pairs.
0b1000  The implementation has eight Address Comparator pairs.
All other values are reserved.
Accessing TRCIDR4

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCIDR4**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1100</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() & CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & ((HaveEL(EL3) | SCR_EL3.FGTEn == '1') & HDFGRTR_EL2.TRCID == '1') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
        if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        X[t, 64] = TRCIDR4;
    end
elsif PSTATE.EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
        if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        X[t, 64] = TRCIDR4;
    end
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = TRCIDR4;
    end
D17.4.37   TRCIDR5, ID Register 5

The TRCIDR5 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

AArch64 System register TRCIDR5 bits [31:0] are architecturally mapped to External register TRCIDR5[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR5 are UNDEFINED.

**Attributes**

TRCIDR5 is a 64-bit register.

**Field descriptions**

**Bits [63:31]**

Reserved, RES0.

**NUMCNTR, bits [30:28]**

Indicates the number of Counters that are available for tracing.

- **0b000**  No Counters are available.
- **0b001**  One Counter implemented.
- **0b010**  Two Counters implemented.
- **0b011**  Three Counters implemented.
- **0b100**  Four Counters implemented.

All other values are reserved.

If TRCIDR4.NUMRSPAIR == 0b0000 then this field is 0b0000.

**NUMSEQSTATE, bits [27:25]**

Indicates if the Sequencer is implemented and the number of Sequencer states that are implemented.

- **0b000**  The Sequencer is not implemented.
- **0b100**  Four Sequencer states are implemented.

All other values are reserved.

If TRCIDR4.NUMRSPAIR == 0b0000 then this field is 0b0000.

**Bit [24]**

Reserved, RES0.
LPOVERRIDE, bit [23]
Indicates support for Low-power Override Mode.
0b0 The trace unit does not support Low-power Override Mode.
0b1 The trace unit supports Low-power Override Mode.

ATBTRIG, bit [22]
Indicates if the implementation can support ATB triggers.
0b0 The implementation does not support ATB triggers.
0b1 The implementation supports ATB triggers.
If TRCIDR4.NUMRSPAIR == 0b0000 then this field is 0.

TRACEIDSIZE, bits [21:16]
Indicates the trace ID width.
0b000000 The external trace interface is not implemented.
0b000111 The implementation supports a 7-bit trace ID.
All other values are reserved.
Note that AMBA ATB requires a 7-bit trace ID width.

Bits [15:12]
Reserved, RES0.

NUMEXTINSEL, bits [11:9]
Indicates how many External Input Selector resources are implemented.
0b000 No External Input Selector resources are available.
0b001 1 External Input Selector resource is available.
0b010 2 External Input Selector resources are available.
0b011 3 External Input Selector resources are available.
0b100 4 External Input Selector resources are available.
All other values are reserved.

NUMEXTIN, bits [8:0]
Indicates how many External Inputs are implemented.
0b1111111111 Unified PMU event selection.
All other values are reserved.

Accessing TRCIDR5
Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, TRCIDR5

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1101</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCID == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
        UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        X[t, 64] = TRCIDR5;
else
    X[t, 64] = TRCIDR5;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCIDR5;
D17.4.38 TRCIDR6, ID Register 6

The TRCIDR6 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

AArch64 System register TRCIDR6 bits [31:0] are architecturally mapped to External register TRCIDR6[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR6 are UNDEFINED.

**Attributes**

TRCIDR6 is a 64-bit register.

**Field descriptions**

![Diagram of TRCIDR6 register]

- **Bits [63:3]**
  - Reserved, RES0.
- **EXLEVEL_RL_EL2, bit [2]**
  - Indicates if Realm EL2 is implemented.
  - 0b0 Realm EL2 is not implemented.
  - 0b1 Realm EL2 is implemented.
- **EXLEVEL_RL_EL1, bit [1]**
  - Indicates if Realm EL1 is implemented.
  - 0b0 Realm EL1 is not implemented.
  - 0b1 Realm EL1 is implemented.
- **EXLEVEL_RL_EL0, bit [0]**
  - Indicates if Realm EL0 is implemented.
  - 0b0 Realm EL0 is not implemented.
  - 0b1 Realm EL0 is implemented.

**Accessing TRCIDR6**

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, TRCIDR6

<table>
<thead>
<tr>
<th></th>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1110</td>
<td>0b111</td>
<td></td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCId == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            X[t, 64] = TRCIDR6;
        end
    end
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            X[t, 64] = TRCIDR6;
        end
    end
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = TRCIDR6;
    end
D17.4.39  TRCIDR7, ID Register 7

The TRCIDR7 characteristics are:

**Purpose**
- Returns the tracing capabilities of the trace unit.

**Configurations**
- AArch64 System register TRCIDR7 bits [31:0] are architecturally mapped to External register TRCIDR7[31:0].
- This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR7 are UNDEFINED.

**Attributes**
- TRCIDR7 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>RES0</td>
</tr>
<tr>
<td>31-0</td>
<td>RES0</td>
</tr>
</tbody>
</table>

**Bits [63:0]**
- Reserved, RES0.

**Accessing TRCIDR7**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCIDR7**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1111</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCID == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = TRCIDR7;
  elsif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
  UNDEFINED;
elsif CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end if
else
  X[t, 64] = TRCIDR7;
end if
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = TRCIDR7;
  end if
D17.4.40 TRCIDR8, ID Register 8

The TRCIDR8 characteristics are:

**Purpose**

Returns the maximum speculation depth of the instruction trace element stream.

**Configurations**

AArch64 System register TRCIDR8 bits [31:0] are architecturally mapped to External register TRCIDR8[31:0].  
This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR8 are UNDEFINED.

**Attributes**

TRCIDR8 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>31:0</td>
<td>MAXSPEC</td>
</tr>
</tbody>
</table>

MAXSPEC, bits [31:0]  
Indicates the maximum speculation depth of the instruction trace element stream. This is the maximum number of P0 elements in the trace element stream that can be speculative at any time.

**Accessing TRCIDR8**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCIDR8**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b01</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then  
UNDEFINED;  
elseif PSTATE.EL == EL1 then  
if Halted() && HaveEl(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' && CPTR_EL3.TTA == '1' then  
UNDEFINED;  
elseif CPACR_EL1.TTA == '1' then  
AArch64.SystemAccessTrap(EL1, 0x18);  
elseif EL2Enabled() && CPTR_EL2.TTA == '1' then  
AArch64.SystemAccessTrap(EL2, 0x18);  
elseif EL2Enabled() && (HaveEl(EL3) || SCR_EL3.FGTEn == '1') && HDGFRTR_EL2.TRCID == '1' then  
AArch64.SystemAccessTrap(EL2, 0x18);  
elseif HaveEl(EL3) && CPTR_EL3.TTA == '1' then  
if Halted() && EDSCR.SDD == '1' then  
UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCIDR8;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = TRCIDR8;
  else
    if CPTR_EL3.TTA == '1' then
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = TRCIDR8;
D17.4.41 TRCIDR9, ID Register 9

The TRCIDR9 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

AArch64 System register TRCIDR9 bits [31:0] are architecturally mapped to External register TRCIDR9[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR9 are UNDEFINED.

**Attributes**

TRCIDR9 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:32]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMP0KEY, bits [31:0]</td>
<td>When TRCIDR0.TRCDATA != 0b00:</td>
</tr>
</tbody>
</table>

  Indicates the number of P0 right-hand keys. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

  Otherwise:

  Reserved, RES0.

**Accessing TRCIDR9**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCIDR9**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEl(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCID == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = TRCIDR9;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = TRCIDR9;
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = TRCIDR9;
D17.4.42 TRCIMSPEC0, IMP DEF Register 0

The TRCIMSPEC0 characteristics are:

**Purpose**

TRCIMSPEC0 shows the presence of any IMPLEMENTATION DEFINED features, and provides an interface to enable the features that are provided.

**Configurations**

AArch64 System register TRCIMSPEC0 bits [31:0] are architecturally mapped to External register TRCIMSPEC0[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIMSPEC0 are UNDEFINED.

**Attributes**

TRCIMSPEC0 is a 64-bit register.

**Field descriptions**

![Field descriptions diagram]

**Bits [63:8]**

Reserved, RES0.

**EN, bits [7:4]**

*When TRCIMSPEC0.SUPPORT != 0b0000:*

Enable. Controls whether the IMPLEMENTATION DEFINED features are enabled.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>The IMPLEMENTATION DEFINED features are not enabled. The trace unit must behave as if the IMPLEMENTATION DEFINED features are not supported.</td>
</tr>
<tr>
<td>0b0001</td>
<td>The trace unit behavior is IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0b0010</td>
<td>The trace unit behavior is IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0b0011</td>
<td>The trace unit behavior is IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0b0100</td>
<td>The trace unit behavior is IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0b0101</td>
<td>The trace unit behavior is IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0b0110</td>
<td>The trace unit behavior is IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0b0111</td>
<td>The trace unit behavior is IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0b1000</td>
<td>The trace unit behavior is IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0b1001</td>
<td>The trace unit behavior is IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0b1010</td>
<td>The trace unit behavior is IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0b1011</td>
<td>The trace unit behavior is IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0b1100</td>
<td>The trace unit behavior is IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0b1101</td>
<td>The trace unit behavior is IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0b1110</td>
<td>The trace unit behavior is IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0b1111</td>
<td>The trace unit behavior is IMPLEMENTATION DEFINED.</td>
</tr>
</tbody>
</table>
The reset behavior of this field is:

• On a Trace unit reset, this field resets to 0.

Otherwise:
Reserved, RES0.

SUPPORT, bits [3:0]
Indicates whether the implementation supports IMPLEMENTATION DEFINED features.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>No IMPLEMENTATION DEFINED features are supported.</td>
</tr>
<tr>
<td>0b0001</td>
<td>IMPLEMENTATION DEFINED features are supported.</td>
</tr>
<tr>
<td>0b0010</td>
<td>IMPLEMENTATION DEFINED features are supported.</td>
</tr>
<tr>
<td>0b0011</td>
<td>IMPLEMENTATION DEFINED features are supported.</td>
</tr>
<tr>
<td>0b0100</td>
<td>IMPLEMENTATION DEFINED features are supported.</td>
</tr>
<tr>
<td>0b0101</td>
<td>IMPLEMENTATION DEFINED features are supported.</td>
</tr>
<tr>
<td>0b0110</td>
<td>IMPLEMENTATION DEFINED features are supported.</td>
</tr>
<tr>
<td>0b0111</td>
<td>IMPLEMENTATION DEFINED features are supported.</td>
</tr>
<tr>
<td>0b1000</td>
<td>IMPLEMENTATION DEFINED features are supported.</td>
</tr>
<tr>
<td>0b1001</td>
<td>IMPLEMENTATION DEFINED features are supported.</td>
</tr>
<tr>
<td>0b1010</td>
<td>IMPLEMENTATION DEFINED features are supported.</td>
</tr>
<tr>
<td>0b1011</td>
<td>IMPLEMENTATION DEFINED features are supported.</td>
</tr>
<tr>
<td>0b1100</td>
<td>IMPLEMENTATION DEFINED features are supported.</td>
</tr>
<tr>
<td>0b1101</td>
<td>IMPLEMENTATION DEFINED features are supported.</td>
</tr>
<tr>
<td>0b1110</td>
<td>IMPLEMENTATION DEFINED features are supported.</td>
</tr>
<tr>
<td>0b1111</td>
<td>IMPLEMENTATION DEFINED features are supported.</td>
</tr>
</tbody>
</table>

Use of nonzero values requires written permission from Arm.
Access to this field is RO.

Accessing TRCIMSPEC0
Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, TRCIMSPEC0

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!(HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRNCMSPECn == '1') then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCIMSPEC0;
elsif PSTATE.EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1'" & CPTR.EL3.TTA == '1' then
        UNDEFINED;
    elsif CPTR.EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) & CPTR.EL3.TTA == '1' then
        if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = TRCIMSPEC0;
else
    if CPTR.EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCIMSPEC0;
elsif PSTATE.EL == EL3 then
    if CPTR.EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    TRCIMSPEC0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1'" & CPTR.EL3.TTA == '1' then
        UNDEFINED;
    elsif CPTR.EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) & CPTR.EL3.TTA == '1' then
        if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRCIMSPEC0 = X[t, 64];
elsif PSTATE.EL == EL3 then
    if CPTR.EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);

---

\[
\begin{array}{lllll}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b10 & 0b001 & 0b0000 & 0b0000 & 0b111
\end{array}
\]
else
    TRCIMSPEC0 = X[t, 64];
D17.4.43 TRCIMSPEC<n>, IMP DEF Register <n>, n = 1 - 7

The TRCIMSPEC<n> characteristics are:

**Purpose**

These registers might return information that is specific to an implementation, or enable features specific to an implementation to be programmed. The product Technical Reference Manual describes these registers.

**Configurations**

AArch64 System register TRCIMSPEC<n> bits [31:0] are architecturally mapped to External register TRCIMSPEC<n>[31:0].

This register is present only when the trace unit implements this OPTIONAL register and FEAT_ETE is implemented. Otherwise, direct accesses to TRCIMSPEC<n> are UNDEFINED.

**Attributes**

TRCIMSPEC<n> is a 64-bit register.

**Field descriptions**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>32</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>RES0</td>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

**Accessing TRCIMSPEC<n>**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCIMSPEC<m>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0:m[2:0]</td>
<td>0b111</td>
</tr>
</tbody>
</table>

integer m = UInt(CRn<2:0>);

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEl(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
  AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL3Enabled() && CPTR_EL3.TTA == '1' then
  AArch64.SystemAccessTrap(EL3, 0x18);
elsif EDSCR.SDD == '1' && CPTR_EL3.TTA == '1' then
  AArch64.SystemAccessTrap(EL3, 0x18);
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRCIMSPECn == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end if;
else
  X[t, 64] = TRCIMSPEC[m];
end if;

integer m = UInt(CRm<2:0>);
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRCIMSPECn == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if;
  else
    X[t, 64] = TRCIMSPEC[m];
  end if;
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = TRCIMSPEC[m];
  end if;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if;
  else
    TRCIMSPEC[m] = X[t, 64];
  end if;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if;
  else
    TRCIMSPEC[m] = X[t, 64];
  end if;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRCIMSPECn == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if;
  else
    TRCIMSPEC[m] = X[t, 64];
  end if;

MSR TRCIMSPEC<m>, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0:m&lt;2:0&gt;</td>
<td>0b111</td>
</tr>
</tbody>
</table>
UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    TRCIMSPEC[m] = X[t, 64];
else if PSTATE_EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRCIMSPEC[m] = X[t, 64];
D17.4.44 TRCOSLSR, Trace OS Lock Status Register

The TRCOSLSR characteristics are:

**Purpose**

Returns the status of the Trace OS Lock.

**Configurations**

AArch64 System register TRCOSLSR bits [31:0] are architecturally mapped to External register TRCOSLSR[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCOSLSR are UNDEFINED.

**Attributes**

TRCOSLSR is a 64-bit register.

**Field descriptions**

```
<table>
<thead>
<tr>
<th>Bit [63:5]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSLM, bits [4:3, 0]</td>
<td>OS Lock model.</td>
</tr>
<tr>
<td>0b000</td>
<td>Trace OS Lock is not implemented.</td>
</tr>
<tr>
<td>0b010</td>
<td>Trace OS Lock is implemented.</td>
</tr>
<tr>
<td>0b100</td>
<td>Trace OS Lock is not implemented, and the trace unit is controlled by the PE OS Lock. All other values are reserved.</td>
</tr>
<tr>
<td>This field reads as 0b100.</td>
<td></td>
</tr>
<tr>
<td>The OSLM field is split as follows:</td>
<td></td>
</tr>
<tr>
<td>• OSLM[2:1] is TRCOSLSR[4:3].</td>
<td></td>
</tr>
<tr>
<td>• OSLM[0] is TRCOSLSR[0].</td>
<td></td>
</tr>
<tr>
<td>Bit [2]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>0b0</td>
<td>The OS Lock is unlocked.</td>
</tr>
<tr>
<td>0b1</td>
<td>The OS Lock is locked.</td>
</tr>
<tr>
<td>Note that this field indicates the state of the PE OS Lock.</td>
<td></td>
</tr>
</tbody>
</table>
```

**Accessing TRCOSLSR**

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, TRCOSLSR

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCSLSR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = TRCOSLSR;
  end
else
  if PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && CPTR_EL3.TTA == '1' then
      UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    else
      X[t, 64] = TRCOSLSR;
    end
  elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = TRCOSLSR;
    end
  end

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b100</td>
</tr>
</tbody>
</table>
D17.4.45 TRCPRGCTLR, Programming Control Register

The TRCPRGCTLR characteristics are:

**Purpose**

Enables the trace unit.

**Configurations**

AArch64 System register TRCPRGCTLR bits [31:0] are architecturally mapped to External register TRCPRGCTLR[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCPRGCTLR are UNDEFINED.

**Attributes**

TRCPRGCTLR is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:31</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>31:1</td>
<td>EN, bit [0]</td>
</tr>
</tbody>
</table>

Trace unit enable.

- **0b0** The trace unit is disabled.
- **0b1** The trace unit is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to 0.

**Accessing TRCPRGCTLR**

Must be programmed.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCPRGCTLR**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
  AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() & CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGRTR_EL2.TRCPRCTLR == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end if;
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end if;
else
  X[t, 64] = TRCPRCTLR;
end if;
elsif PSTATE_EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  end if;
elsif CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end if;
elsif PSTATE_EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = TRCPRCTLR;
end if;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL1, 0x18);
  end if;
elsif EL2Enabled() & CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGWTR_EL2.TRCPRCTLR == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end if;
else
  TRCPRCTLR = X[t, 64];
elsif PSTATE_EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  end if;
elsif CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end if;
else
  TRCPRCTLR = X[t, 64];
end if;

**MSR TRCPRCTLR, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL1, 0x18);
  end if;
elsif EL2Enabled() & CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGWTR_EL2.TRCPRCTLR == '1' then
  AArch64.SystemAccess Trap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end if;
else
  TRCPRCTLR = X[t, 64];
elsif PSTATE_EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  end if;
elsif CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    TRCPRTGLR = X[t, 64];
elsif PSTATE_EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRCPRTGLR = X[t, 64];
D17.4.46 TRCQCTLR, Q Element Control Register

The TRCQCTLR characteristics are:

**Purpose**
Controls when Q elements are enabled.

**Configurations**
AArch64 System register TRCQCTLR bits [31:0] are architecturally mapped to External register TRCQCTLR[31:0].
This register is present only when FEAT_ETE is implemented and TRCIDR0.QFILT == 1.
Otherwise, direct accesses to TRCQCTLR are UNDEFINED.

**Attributes**
TRCQCTLR is a 64-bit register.

**Field descriptions**

![ register diagram ]

**Bits [63:9]**
Reserved, RES0.

**MODE, bit [8]**
Selects whether the Address Range Comparators selected by TRCQCTLR.RANGE indicate address ranges where the trace unit is permitted to generate Q elements or address ranges where the trace unit is not permitted to generate Q elements:

0b0 Exclude mode.
The Address Range Comparators selected by TRCQCTLR.RANGE indicate address ranges where the trace unit must not generate Q elements. If no ranges are selected, Q elements are permitted across the entire memory map.

0b1 Include Mode.
The Address Range Comparators selected by TRCQCTLR.RANGE indicate address ranges where the trace unit can generate Q elements. If all the implemented bits in RANGE are set to 0 then Q elements are disabled.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**RANGE[<m>], bit [m], for m = 7 to 0**
Specifies whether Address Range Comparator <m> controls Q elements.

0b0 The address range that Address Range Comparator <m> defines, is not selected.

0b1 The address range that Address Range Comparator <m> defines, is selected.

This bit is RES0 if m >= TRCIDR4.NUMACPAIRS.
The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing TRCQCTLR**

Must be programmed if TRCCONFIGR.QE != 0b00.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <X>, TRCQCTLR**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif CPACR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && CPTR_EL3.TTA == '1' then
      UNDEFINED;
    else
      UNDEFINED;
    end (
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  end else

if PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif haveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      UNDEFINED;
    end else
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end else

if PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = TRCQCTLR;
  end else

X[t, 64] = TRCQCTLR;
```

D17-6666  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  Non-Confidential
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && ((!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if
  else
    TRCQCTLR = X[t, 64];
  end if
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if
  else
    TRCQCTLR = X[t, 64];
  end if
elsif PSTATE_EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCQCTLR = X[t, 64];
  end if
D17.4.47  TRCRSCTRL<\textgreater n\textless>, Resource Selection Control Register <\textgreater n\textless>, n = 2 - 31

The TRCRSCTRL<\textgreater n\textless> characteristics are:

**Purpose**

Controls the selection of the resources in the trace unit.

**Configurations**

AArch64 System register TRCRSCTRL<\textgreater n\textless> bits [31:0] are architecturally mapped to External register TRCRSCTRL<\textgreater n\textless>[31:0].

This register is present only when FEAT_ETE is implemented and (UInt(TRCIDR4.NUMRSPAIR) + 1) * 2 > n. Otherwise, direct accesses to TRCRSCTRL<\textgreater n\textless> are UNDEFINED.

Resource selector 0 always returns FALSE.

Resource selector 1 always returns TRUE.

Resource selectors are implemented in pairs. Each odd numbered resource selector is part of a pair with the even numbered resource selector that is numbered as one less than it. For example, resource selectors 2 and 3 form a pair.

**Attributes**

TRCRSCTRL<\textgreater n\textless> is a 64-bit register.

**Field descriptions**

| Bits [63:22] | Reserved, RES0. |
| PAIRINV, bit [21] |

* **When n MOD 2 == 0:**

  Controls whether the combined result from a resource selector pair is inverted.

  0b0  Do not invert the combined output of the 2 resource selectors.

  0b1  Invert the combined output of the 2 resource selectors.

  If:

  * A is the register TRCRSCTRL<\textgreater n\textless>.
  * B is the register TRCRSCTRL<\textgreater n+1\textless>.

  Then the combined output of the 2 resource selectors A and B depends on the value of (A.PAIRINV, A.INV, B.INV) as follows:

  * 0b0000 -> A and B.
  * 0b0001 -> Reserved.
  * 0b0010 -> not(A) and B.
  * 0b0011 -> not(A) and not(B).
  * 0b0100 -> not(A) or not(B).
  * 0b0101 -> not(A) or B.
  * 0b0110 -> Reserved.
• 0b111 -> A or B.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

INV, bit [20]
Controls whether the resource, that TRCRSCTLR<n>.GROUP and TRCRSCTLR<n>.SELECT selects, is inverted.

0b0 Do not invert the output of this selector.
0b1 Invert the output of this selector.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

GROUP, bits [19:16]
Selects a group of resources.

0b0000 External Input Selectors.
0b0001 PE Comparator Inputs.
0b0010 Counters and Sequencer.
0b0011 Single-shot Comparator Controls.
0b0100 Single Address Comparators.
0b0101 Address Range Comparators.
0b0110 Context Identifier Comparators.
0b0111 Virtual Context Identifier Comparators.
All other values are reserved.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT, bits [15:0]
Resource Specific Controls. Contains the controls specific to the resource group selected by GROUP, described in the following sections.

SELECT encoding for External Input Selectors

Bits [15:4]
Reserved, RES0.

EXTIN[<m>], bit [m], for m = 3 to 0
Selects one or more External Inputs.

0b0 Ignore EXTIN <m>.
0b1 Select EXTIN <m>.
This bit is RES0 if m >= TRCIDR5.NUMEXTINSEL.
The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**SELECT encoding for PE Comparator Inputs**

![Diagram](image)

**Bits [15:8]**

Reserved, RES0.

**PECOMP[<m>], bit [m], for m = 7 to 0**

Selects one or more PE Comparator Inputs.

0b0 Ignored PE Comparator Input <m>.
0b1 Select PE Comparator Input <m>.

This bit is RES0 if m >= TRCIDR4.NUMPC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**SELECT encoding for Counters and Sequencer**

![Diagram](image)

**Bits [15:8]**

Reserved, RES0.

**SEQUENCER[<m>], bit [m+4], for m = 3 to 0**

Sequencer states.

0b0 Ignore Sequencer state <m>.
0b1 Select Sequencer state <m>.

This bit is RES0 if m >= TRCIDR5.NUMSEQSTATE.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**COUNTERS[<m>], bit [m], for m = 3 to 0**

Counters resources at zero.

0b0 Ignore Counter <m>.
0b1 Select Counter <m> is zero.

This bit is RES0 if m >= TRCIDR5.NUMCNTR.
The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**SELECT encoding for Single-shot Comparator Controls**

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 15| 14| 13| 12| 11| 10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| RES0|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SINGLE_SHOT[7]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SINGLE_SHOT[6]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SINGLE_SHOT[5]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SINGLE_SHOT[4]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SINGLE_SHOT[3]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SINGLE_SHOT[2]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SINGLE_SHOT[1]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SINGLE_SHOT[0]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
```

Bits [15:8]

Reserved, RES0.

**SINGLE_SHOT[<m>], bit [m], for m = 7 to 0**

Selects one or more Single-shot Comparator Controls.

- 0b0: Ignore Single-shot Comparator Control <m>.
- 0b1: Select Single-shot Comparator Control <m>.

This bit is RES0 if m >= TRCIDR4.NUMSSCC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**SELECT encoding for Single Address Comparators**

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 16| 15| 14| 13| 12| 11| 10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SAC[15]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SAC[14]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SAC[13]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SAC[12]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SAC[11]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SAC[10]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SAC[9]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SAC[8]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SAC[7]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SAC[6]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SAC[5]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SAC[4]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SAC[3]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SAC[2]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SAC[1]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| SAC[0]|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

**SAC[<m>], bit [m], for m = 15 to 0**

Selects one or more Single Address Comparators.

- 0b0: Ignore Single Address Comparator <m>.
- 0b1: Select Single Address Comparator <m>.

This bit is RES0 if m >= 2 × TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
SELECT encoding for Address Range Comparators

Bits [15:8]
Reserved, RES0.

ARC[<m>], bit [m], for m = 7 to 0
Selects one or more Address Range Comparators.
0b0 Ignore Address Range Comparator <m>.
0b1 Select Address Range Comparator <m>.
This bit is RES0 if m >= TRCIDR4.NUMACPAIRS.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Context Identifier Comparators

Bits [15:8]
Reserved, RES0.

CID[<m>], bit [m], for m = 7 to 0
Selects one or more Context Identifier Comparators.
0b0 Ignore Context Identifier Comparator <m>.
0b1 Select Context Identifier Comparator <m>.
This bit is RES0 if m >= TRCIDR4.NUMCIDC.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Virtual Context Identifier Comparators
Bits [15:8]

Reserved, RES0.

VMID[<m>], bit [m], for m = 7 to 0

Selects one or more Virtual Context Identifier Comparators.

0b0  Ignore Virtual Context Identifier Comparator <m>.
0b1  Select Virtual Context Identifier Comparator <m>.

This bit is RES0 if m >= TRCIDR4.NUMVMIDC.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TCRSCTLR<n>

Must be programmed if any of the following are true:

• TRCCNTCTLR<n>.RLDEVENT.TYPE == 0 and TRCCNTCTLR<n>.RLDEEVENT.SEL == n.
• TRCCNTCTLR<n>.RLDEVENT.TYPE == 1 and TRCCNTCTLR<n>.RLDEEVENT.SEL == n/2.
• TRCCNTCTLR<n>.CNTEVENT.TYPE == 0 and TRCCNTCTLR<n>.CNTEVENT.SEL == n.
• TRCCNTCTLR<n>.CNTEVENT.TYPE == 1 and TRCCNTCTLR<n>.CNTEVENT.SEL == n/2.
• TRCEVENTCTLR0.EVENT0.TYPE == 0 and TRCEVENTCTLR0.EVENT0.SEL == n.
• TRCEVENTCTLR0.EVENT0.TYPE == 1 and TRCEVENTCTLR0.EVENT0.SEL == n/2.
• TRCEVENTCTLR0.EVENT1.TYPE == 0 and TRCEVENTCTLR0.EVENT1.SEL == n.
• TRCEVENTCTLR0.EVENT1.TYPE == 1 and TRCEVENTCTLR0.EVENT1.SEL == n/2.
• TRCEVENTCTLR0.EVENT2.TYPE == 0 and TRCEVENTCTLR0.EVENT2.SEL == n.
• TRCEVENTCTLR0.EVENT2.TYPE == 1 and TRCEVENTCTLR0.EVENT2.SEL == n/2.
• TRCEVENTCTLR0.EVENT3.TYPE == 0 and TRCEVENTCTLR0.EVENT3.SEL == n.
• TRCEVENTCTLR0.EVENT3.TYPE == 1 and TRCEVENTCTLR0.EVENT3.SEL == n/2.
• TRCSEQEVR<n>.B.TYPE == 0 and TRCSEQEVR<n>.B.SEL = n.
• TRCSEQEVR<n>.B.TYPE == 1 and TRCSEQEVR<n>.B.SEL = n/2.
• TRCSEQEVR<n>.B.TYPE == 0 and TRCSEQEVR<n>.F.SEL = n.
• TRCSEQEVR<n>.F.TYPE == 1 and TRCSEQEVR<n>.F.SEL = n/2.
• TRCSEQEVR<n>.F.TYPE == 1 and TRCSEQEVR<n>.F.SEL = n/2.
• TRCSEQEVR<n>.RST.TYPE == 0 and TRCSEQEVR<n>.RST.SEL == n.
• TRCSEQEVR<n>.RST.TYPE == 1 and TRCSEQEVR<n>.RST.SEL == n/2.
• TRCTSCTLR.EVENT.TYPE == 0 and TRCTSCTLR.EVENT.SEL == n.
• TRCTSCTLR.EVENT.TYPE == 1 and TRCTSCTLR.EVENT.SEL == n/2.
• TRCVICTLR.EVENT.TYPE == 0 and TRCVICTLR.EVENT.SEL == n.
• TRCVICTLR.EVENT.TYPE == 1 and TRCVICTLR.EVENT.SEL == n/2.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:
MRS \(<X_t>, \text{TRCRSCTL}<m>\)

<table>
<thead>
<tr>
<th>(\text{op0})</th>
<th>(\text{op1})</th>
<th>(\text{CRn})</th>
<th>(\text{CRm})</th>
<th>(\text{op2})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0001</td>
<td>(m[3:0])</td>
<td>0b00:0m[4]</td>
</tr>
</tbody>
</table>

integer \(m = \text{UInt}(\text{op2<0>:CRm<3:0>});\)

if \(m \geq \text{NUM\_TRACE\_RESOURCE\_SELECTOR\_PAIRS} \times 2\) then
  UNDEFINED;
elsif \(\text{PSTATE.EL} = \text{EL0}\) then
  UNDEFINED;
elsif \(\text{PSTATE.EL} = \text{EL1}\) then
  if \(\text{Halted()} \&\& \text{haveEL(EL3)} \&\& \text{EDSCR.SDD} = \text{'1'} \&\& \text{boolean IMPLEMENTATION\_DEFINED "EL3 trap priority when SDD == '1'" \&\& CPTR\_EL3.TTA == '1' then}
    UNDEFINED;
  else
    \(\text{AArch64.SystemAccessTrap(EL1, 0x18)};\)
  endif
endif

MSR \(\text{TRCRSCTL}<m>, <X_t>\)

<table>
<thead>
<tr>
<th>(\text{op0})</th>
<th>(\text{op1})</th>
<th>(\text{CRn})</th>
<th>(\text{CRm})</th>
<th>(\text{op2})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0001</td>
<td>(m[3:0])</td>
<td>0b00:0m[4]</td>
</tr>
</tbody>
</table>

integer \(m = \text{UInt}(\text{op2<0>:CRm<3:0>});\)

if \(m \geq \text{NUM\_TRACE\_RESOURCE\_SELECTOR\_PAIRS} \times 2\) then
  UNDEFINED;
elsif \(\text{PSTATE.EL} = \text{EL0}\) then
  UNDEFINED;
elsif \(\text{PSTATE.EL} = \text{EL1}\) then
  if \(\text{Halted()} \&\& \text{haveEL(EL3)} \&\& \text{EDSCR.SDD} = \text{'1'} \&\& \text{boolean IMPLEMENTATION\_DEFINED "EL3 trap priority when SDD == '1'" \&\& CPTR\_EL3.TTA == '1' then}
    UNDEFINED;
  else
    \(\text{AArch64.SystemAccessTrap(EL1, 0x18)};\)
  endif
endif

\[\text{MRS } <X_t>, \text{TRCRSCTL}<m>\]
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && CPTR_EL3.TTA == '1' then
  UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
  AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCRSCTLR[m] = X[t, 64];
  endif
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  else
    TRCRSCTLR[m] = X[t, 64];
  endif
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCRSCTLR[m] = X[t, 64];
D17.4.48 TRCRSR, Resources Status Register

The TRCRSR characteristics are:

**Purpose**

Use this to set, or read, the status of the resources.

**Configurations**

AArch64 System register TRCRSR bits [31:0] are architecturally mapped to External register TRCRSR[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCRSR are UNDEFINED.

**Attributes**

TRCRSR is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
<th>Reset Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Reserved, RES0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>TA, Tracing active</td>
<td>0b0</td>
<td>Tracing is not active.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b1</td>
<td>Tracing is active.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>8</td>
<td>EVENT[3], Untraced status of ETEEvents</td>
<td>0b0</td>
<td>An ETEEvent &lt;m&gt; has not occurred.</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>0b1</td>
<td>An ETEEvent &lt;m&gt; has occurred while the resources were in the Paused state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This bit is res0 if TRCIDR4.NUMRSPAIR == 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>4</td>
<td>EVENT[2],</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>EVENT[1],</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>EVENT[0],</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:13]**

Reserved, RES0.

**TA, bit [12]**

Tracing active.

0b0 Tracing is not active.

0b1 Tracing is active.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**EVENT[<m>], bit [m+8], for m = 3 to 0**

Untraced status of ETEEvents.

0b0 An ETEEvent <m> has not occurred.

0b1 An ETEEvent <m> has occurred while the resources were in the Paused state.

This bit is res0 if TRCIDR4.NUMRSPAIR == 0 || m > TRCIDR0.NUMEVENT.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Bits [7:4]**

Reserved, RES0.

**EXTIN[<m>], bit [m], for m = 3 to 0**

The sticky status of the External Input Selectors.

0b0 An event selected by External Input Selector <m> has not occurred.
At least one event selected by External Input Selector <m> has occurred while the resources were in the Paused state.

This bit is RES0 if m >= TRCIDR5.NUMEXTINSEL.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

### Accessing TRCRSR

Must always be programmed.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCRSR**

<table>
<thead>
<tr>
<th>0b10</th>
<th>0b001</th>
<th>0b0000</th>
<th>0b1010</th>
<th>0b000</th>
</tr>
</thead>
</table>

```asm
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elseif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elseif EL2Enabled() && CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
        else
        X[t, 64] = TRCRSR;
    end
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elseif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
        else
        X[t, 64] = TRCRSR;
    end
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = TRCRSR;
    end
```

AIAAI
### MSR TRCRSR, <xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && !(HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL3Enabled() && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    TRCRSR = X[t, 64];
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    TRCRSR = X[t, 64];
  end
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCRSR = X[t, 64];
end
D17.4.49 TRCSEQEVR\(<n>\), Sequencer State Transition Control Register \(<n>\), \(n = 0 - 2\)

The TRCSEQEVR\(<n>\) characteristics are:

**Purpose**

Moves the Sequencer state:
- Backwards, from state \(n+1\) to state \(n\) when a programmed resource event occurs.
- Forwards, from state \(n\) to state \(n+1\) when a programmed resource event occurs.

**Configurations**

AArch64 System register TRCSEQEVR\(<n>\) bits [31:0] are architecturally mapped to External register TRCSEQEVR\(<n>\[31:0\].

This register is present only when FEAT_ETE is implemented and TRCIDR5.NUMSEQSTATE != 0b000. Otherwise, direct accesses to TRCSEQEVR\(<n>\) are UNDEFINED.

**Attributes**

TRCSEQEVR\(<n>\) is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>RES0</td>
</tr>
<tr>
<td>31</td>
<td>RES0</td>
</tr>
<tr>
<td>16</td>
<td>RES0, B_SEL</td>
</tr>
<tr>
<td>14</td>
<td>RES0, B_SEL</td>
</tr>
<tr>
<td>13</td>
<td>B_TYPE</td>
</tr>
<tr>
<td>12</td>
<td>F_TYPE</td>
</tr>
<tr>
<td>11</td>
<td>B_SEL</td>
</tr>
<tr>
<td>10</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>9</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>8</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>7</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>6</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>5</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>4</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>3</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>2</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>1</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>0</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**Bits [63:16]**

Reserved, RES0.

**B_TYPE, bit [15]**

Chooses the type of Resource Selector.

Backward field. Defines whether the backward resource event is a single Resource Selector or a Resource Selector pair. When the resource event occurs then the Sequencer state moves from state \(n+1\) to state \(n\). For example, if TRCSEQEVR2.B_SEL == 0x14 then when event 0x14 occurs, the Sequencer moves from state 3 to state 2.

- \(0b0\) A single Resource Selector.
  - TRCSEQEVR\(<n>\).B_SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

- \(0b1\) A Boolean-combined pair of Resource Selectors.
  - TRCSEQEVR\(<n>\).B_SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event.
  - TRCSEQEVR\(<n>\).B_SEL[4] is RES0.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Bits [14:13]**

Reserved, RES0.

**B_SEL, bits [12:8]**

Defines the selected Resource Selector or pair of Resource Selectors. TRCSEQEVR\(<n>\).B_TYPE controls whether TRCSEQEVR\(<n>\).B_SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.
Backward field. Selects the single Resource Selector or Resource Selector pair.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**F_TYPE, bit [7]**

Chooses the type of Resource Selector.

Backward field. Defines whether the forward resource event is a single Resource Selector or a Resource Selector pair. When the resource event occurs then the Sequencer state moves from state n to state n+1. For example, if TRCSEQEVR1.F.SEL == 0x12 then when event 0x12 occurs, the Sequencer moves from state 1 to state 2.

- 0b0 A single Resource Selector.
  TRCSEQEVR<n>.F.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

- 0b1 A Boolean-combined pair of Resource Selectors.
  TRCSEQEVR<n>.F.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event.
  TRCSEQEVR<n>.F.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Bits [6:5]**

Reserved, RES0.

**F_SEL, bits [4:0]**

Defines the selected Resource Selector or pair of Resource Selectors. TRCSEQEVR<n>.F.TYPE controls whether TRCSEQEVR<n>.F.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

Forward field. Selects the single Resource Selector or Resource Selector pair.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing TRCSEQEVR<n>**

Must be programmed if TRCRSCTRL<n>.GROUP == 0b0010 and TRCRSCTRL<n>.SEQUENCER != 0b0000.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, TRCSEQEVR<m>

integer m = UInt(CRm<1:0>);

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRCE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    elsif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
            UNDEFINED;
        elsif CPTR_EL2.TTA == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end
        else
            X[t, 64] = TRCSEQEVR[m];
        end
    elsif PSTATE.EL == EL3 then
        if CPTR_EL3.TTA == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = TRCSEQEVR[m];
        end
    else
        X[t, 64] = TRCSEQEVR[m];
    end

MSR TRCSEQEVR<m>, <Xt>

integer m = UInt(CRm<1:0>);

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    else
        X[t, 64] = TRCSEQEVR[m];
    end

AArch64 System Register Descriptions
D17.4 Trace registers

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b00:m[1:0]</td>
<td>0b100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b00:m[1:0]</td>
<td>0b100</td>
</tr>
</tbody>
</table>
AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    TRCSEQEVR[m] = X[t, 64];
  end if
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  else
    if CPT_AR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      if CPT_AR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
      else
        TRCSEQEVR[m] = X[t, 64];
      end if
    end if
  end if
else
  if EL3Enabled() && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if
  else
    TRCSEQEVR[m] = X[t, 64];
  end if
elsif PSTATE.EL == EL3 then
  if CPR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCSEQEVR[m] = X[t, 64];
  end if
D17.4.50 TRCSEQRSTEV, Sequencer Reset Control Register

The TRCSEQRSTEV characteristics are:

Purpose

Moves the Sequencer to state 0 when a programmed resource event occurs.

Configurations

AArch64 System register TRCSEQRSTEV bits [31:0] are architecturally mapped to External register TRCSEQRSTEV[31:0].

This register is present only when FEAT_ETE is implemented and TRCIDR5.NUMSEQSTATE != 0b000. Otherwise, direct accesses to TRCSEQRSTEV are UNDEFINED.

Attributes

TRCSEQRSTEV is a 64-bit register.

Field descriptions

Bits [63:8]

Reserved, RES0.

RST_TYPE, bit [7]

Chooses the type of Resource Selector.

0b0 A single Resource Selector.

TRCSEQRSTEV.RST_SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

0b1 A Boolean-combined pair of Resource Selectors.

TRCSEQRSTEV.RST_SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCSEQRSTEV.RST_SEL[4] is RES0.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

RST_SEL, bits [4:0]

Defines the selected Resource Selector or pair of Resource Selectors.

TRCSEQRSTEV.RST.TYPE controls whether TRCSEQRSTEV.RST_SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.
The reset behavior of this field is:
   • On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing TRCSEQRSTEV**

Must be programmed if TRCRSCTL<\text{n}>.GROUP = 0b0010 and TRCRSCTL<\text{n}>.SEQUENCER != 0b0000.

 Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

**MRS \text{<Xt>}, TRCSEQRSTEV**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0110</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif CPACR_EL1.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    else
      X[t, 64] = TRCSEQRSTEV;
  else
    AArch64.SystemAccessTrap(EL1, 0x18);
  endif
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = TRCSEQRSTEV;
  elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = TRCSEQRSTEV;
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    TRCSEQRSTEVR = X[t, 64];
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    TRCSEQRSTEVR = X[t, 64];
  end
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCSEQRSTEVR = X[t, 64];
end
D17.4.51 TRCSEQSTR, Sequencer State Register

The TRCSEQSTR characteristics are:

**Purpose**

Use this to set, or read, the Sequencer state.

**Configurations**

AArch64 System register TRCSEQSTR bits [31:0] are architecturally mapped to External register TRCSEQSTR[31:0].

This register is present only when FEAT_ETE is implemented and TRCIDR5.NUMSEQSTATE != 0b000. Otherwise, direct accesses to TRCSEQSTR are UNDEFINED.

**Attributes**

TRCSEQSTR is a 64-bit register.

**Field descriptions**

Bits [63:2]

Reserved, RES0.

**STATE, bits [1:0]**

Set or returns the state of the Sequencer.

0b00 State 0.

0b01 State 1.

0b10 State 2.

0b11 State 3.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing TRCSEQSTR**

Must be programmed if TRCRSCTRL<n>.GROUP == 0b0010 and TRCRSCTRL<n>.SEQUENCER != 0b0000.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

Accesses to this register use the following encodings in the System register encoding space:
### MRS <Xt>, TRCSEQSTR

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0111</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDENfined;
elIf PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & CPTR_EL3.TTA == '1' then
    UNDEFINED;
elIf CPACR_EL1.TTA == '1' then
      AArch64.SystemAccessTrap(EL1, 0x18);
elIf EL2Enabled() & CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
elIf EL2Enabled() & (((HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGWTR_EL2.TRCEQSTR == '1') then
      AArch64.SystemAccessTrap(EL2, 0x18);
elIf HaveEL(EL3) & CPTR_EL3.TTA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = TRCSEQSTR;
elIf PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & CPTR_EL3.TTA == '1' then
    UNDEFINED;
elIf CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
elIf HaveEL(EL3) & CPTR_EL3.TTA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = TRCSEQSTR;
elIf PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCSEQSTR;

### MSR TRCSEQSTR, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0111</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDENfined;
elIf PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & CPTR_EL3.TTA == '1' then
    UNDEFINED;
elIf CPACR_EL1.TTA == '1' then
      AArch64.SystemAccessTrap(EL1, 0x18);
elIf EL2Enabled() & CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
elIf EL2Enabled() & ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGWTR_EL2.TRCEQSTR == '1') then
      AArch64.SystemAccessTrap(EL2, 0x18);
elIf HaveEL(EL3) & CPTR_EL3.TTA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
else
      AArch64.SystemAccessTrap(EL3, 0x18);
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end if;
else
  TRCSEQSTR = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD = '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if;
  else
    TRCSEQSTR = X[t, 64];
  end if;
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCSEQSTR = X[t, 64];
  end if;
D17.4.52  TRCSSCCR<\text{n}>, Single-shot Comparator Control Register <\text{n}>, \text{n} = 0 - 7

The TRCSSCCR<\text{n}> characteristics are:

**Purpose**

Controls the corresponding Single-shot Comparator Control resource.

**Configurations**

AArch64 System register TRCSSCCR<\text{n}> bits [31:0] are architecturally mapped to External register TRCSSCCR<\text{n}>[31:0].

This register is present only when FEAT ETE is implemented and UInt(TRCIDR4.NUMSSCC) > \text{n}. Otherwise, direct accesses to TRCSSCCR<\text{n}> are UNDEFINED.

**Attributes**

TRCSSCCR<\text{n}> is a 64-bit register.

**Field descriptions**

Bits [63:25]

Reserved, RES0.

RST, bit [24]

Selects the Single-shot Comparator Control mode.

0b0  The Single-shot Comparator Control is in single-shot mode.

0b1  The Single-shot Comparator Control is in multi-shot mode.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

ARC\text{[<m>]}, bit [m+16], for m = 7 to 0

Selects one or more Address Range Comparators for Single-shot control.

0b0  The Address Range Comparator <\text{m}>, is not selected for Single-shot control.

0b1  The Address Range Comparator <\text{m}>, is selected for Single-shot control.

This bit is RES0 if \text{m} >= TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
SAC[<m>], bit [m], for m = 15 to 0

Selects one or more Single Address Comparators for Single-shot control.

0b0  The Single Address Comparator <m>, is not selected for Single-shot control.
0b1  The Single Address Comparator <m>, is selected for Single-shot control.

This bit is res0 if m >= 2 × TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:

•  On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSSCCR<n>

Must be programmed if any TRCRSCTL<n>.GROUP == 0b0011 and TRCRSCTL<n>.SINGLE_SHOT[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

\[
egin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b10 & 0b001 & 0b0001 & 0b0:m[2:0] & 0b010 \\
\end{array}
\]

integer m = UInt(CRm<2:0>);

if m >= NUM_TRACE_SINGLE_SHOT_COMPARATOR_CONTROLS then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
        UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
    AA64SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() & CPTR_EL2.TTA == '1' then
    AA64SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGRTR_EL2.TRCC == '1' then
    AA64SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
    if Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
else
    AA64SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCSSCCR[m];
elsif PSTATE.EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
        UNDEFINED;
elsif CPTR_EL2.TTA == '1' then
    AA64SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
    if Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
else
    AA64SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCSSCCR[m];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AA64SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCSSCCR[m];

**MSR TRCSSCCR<m>, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0001</td>
<td>0b0:m[2:0]</td>
<td>0b010</td>
</tr>
</tbody>
</table>

integer m = UInt(CRm<2:0>);
    if m >= NUM_TRACE_SINGLE_SHOT_COMPARATOR_CONTROLS then
        UNDEFINED;
    elsif PSTATE.EL == EL0 then
        UNDEFINED;
    elsif PSTATE.EL == EL1 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
            UNDEFINED;
        elsif CPACR_EL1.TTA == '1' then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
else
    TRCSSCCR[m] = X[t, 64];
else
    if PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
            UNDEFINED;
        elsif CPTR_EL2.TTA == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end
    else
        TRCSSCCR[m] = X[t, 64];
    end
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRCSSCCR[m] = X[t, 64];
    end
D17.4.53 TRCSSCSR<\(n\)>, Single-shot Comparator Control Status Register <\(n\)>, \(n = 0 - 7\)

The TRCSSCSR<\(n\)> characteristics are:

**Purpose**

Returns the status of the corresponding Single-shot Comparator Control.

**Configurations**

AArch64 System register TRCSSCSR<\(n\)> bits [31:0] are architecturally mapped to External register TRCSSCSR<\(n\)>[31:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR4.NUMSSCC) > \(n\). Otherwise, direct accesses to TRCSSCSR<\(n\)> are UNDEFINED.

**Attributes**

TRCSSCSR<\(n\)> is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits [63:32]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>STATUS, bit [31]</td>
<td>Single-shot Comparator Control status. Indicates if any of the comparators selected by this Single-shot Comparator control have matched. The selected comparators are defined by TRCSSCCR&lt;(n)&gt;.ARC, TRCSSCCR&lt;(n)&gt;.SAC, and TRCSSPCICR&lt;(n)&gt;.PC.</td>
</tr>
<tr>
<td>(0b0)</td>
<td>No match has occurred. When the first match occurs, this field takes a value of 1. It remains at 1 until explicitly modified by a write to this register.</td>
</tr>
<tr>
<td>(0b1)</td>
<td>One or more matches has occurred. If TRCSSCCR&lt;(n)&gt;.RST == 0 then:</td>
</tr>
<tr>
<td></td>
<td>• There is only one match and no more matches are possible.</td>
</tr>
<tr>
<td></td>
<td>• Software must reset this field to 0 to re-enable the Single-shot Comparator Control.</td>
</tr>
<tr>
<td>The reset behavior of this field is:</td>
<td></td>
</tr>
<tr>
<td>• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.</td>
<td></td>
</tr>
<tr>
<td>PENDING, bit [30]</td>
<td>Single-shot pending status. The Single-shot Comparator Control fired while the resources were in the Paused state.</td>
</tr>
<tr>
<td>(0b0)</td>
<td>No match has occurred.</td>
</tr>
<tr>
<td>(0b1)</td>
<td>One or more matches has occurred.</td>
</tr>
<tr>
<td>The reset behavior of this field is:</td>
<td></td>
</tr>
<tr>
<td>• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.</td>
<td></td>
</tr>
<tr>
<td>Bits [29:4]</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>
PC, bit [3]

PE Comparator Input support. Indicates if the Single-shot Comparator Control supports PE Comparator Inputs.

0b0  This Single-shot Comparator Control does not support PE Comparator Inputs. Selecting any PE Comparator Inputs using the associated TRCSSPCICR<n> results in CONstrained unpredictable behavior of the Single-shot Comparator Control resource. The Single-shot Comparator Control might match unexpectedly or might not match.

0b1  This Single-shot Comparator Control supports PE Comparator Inputs.

Access to this field is RO.

DV, bit [2]

Data value comparator support. Data value comparisons are not implemented in ETE and are reserved for other trace architectures. Allocated in other trace architectures.

0b0  This Single-shot Comparator Control does not support data value comparisons.

0b1  This Single-shot Comparator Control supports data value comparisons.

This field reads as 0.

Access to this field is RO.

DA, bit [1]

Data Address Comparator support. Data address comparisons are not implemented in ETE and are reserved for other trace architectures. Allocated in other trace architectures.

0b0  This Single-shot Comparator Control does not support data address comparisons.

0b1  This Single-shot Comparator Control supports data address comparisons.

This field reads as 0.

Access to this field is RO.

INST, bit [0]

Instruction Address Comparator support. Indicates if the Single-shot Comparator Control supports instruction address comparisons.

0b0  This Single-shot Comparator Control does not support instruction address comparisons.

0b1  This Single-shot Comparator Control supports instruction address comparisons.

This field reads as 1.

Access to this field is RO.

Accessing TRCSSCSR<n>

Must be programmed if TRCRSCTL<n>.GROUP == 0b0011 and TRCRSCTL<n>.SINGLE_SHOT[n] == 1.

Writes are CONstrained Unpredictable if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

Accesses to this register use the following encodings in the System register encoding space:

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b10 & 0b001 & 0b0001 & 0b1:m[2:0] & 0b010 \\
\end{array}
\]

\[\text{integer } m = \text{UInt}(\text{CRm}[2:0]);\]
MSR TRCSSCSR<\(m\)>, <\(X_t\)>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0001</td>
<td>0b1:m[2:0]</td>
<td>0b010</td>
</tr>
</tbody>
</table>

integer \(m\) = UInt(CRm\[2:0]\));

if \(m \geq\) NUM_TRACE_SINGLE_SHOT_COMPARATOR_CONTROLS then
UNDEFINED;

elsif PSTATE_EL == EL0 then
UNDEFINED;

elsif PSTATE_EL == EL1 then
    if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& CPTR_EL3.TTA == '1' then
UNDEFINED;

    elsif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() \&\& CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) \&\& CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    AArch64.SystemAccessTrap(EL3, 0x18);

    X[t, 64] = TRCSSCSR[m];

else PSTATE_EL == EL2 then
    if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& CPTR_EL3.TTA == '1' then
UNDEFINED;

    elsif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) \&\& CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    AArch64.SystemAccessTrap(EL3, 0x18);

    X[t, 64] = TRCSSCSR[m];

else PSTATE_EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCSSCSR[m];
UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRCSSCSR[m] = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    TRCSSCSR[m] = X[t, 64];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    TRCSSCSR[m] = X[t, 64];
D17.4.54 TRCSPCICR<n>, Single-shot Processing Element Comparator Input Control Register <n>, n = 0 - 7

The TRCSPCICR<n> characteristics are:

**Purpose**

Returns the status of the corresponding Single-shot Comparator Control.

**Configurations**

AArch64 System register TRCSPCICR<n> bits [31:0] are architecturally mapped to External register TRCSPCICR<n>[31:0].

This register is present only when FEAT_ETE is implemented, UInt(TRCIDR4.NUMSSCC) > n, UInt(TRCIDR4.NUMPC) > 0 and TRCSSCSR<n>.PC == 1. Otherwise, direct accesses to TRCSSPCICR<n> are UNDEFINED.

**Attributes**

TRCSSPCICR<n> is a 64-bit register.

**Field descriptions**

![Field descriptions diagram]

**Bits [63:8]**

Reserved, RES0.

**PC[<m>], bit [m], for m = 7 to 0**

Selects one or more PE Comparator Inputs for Single-shot control.

- 0b0: The single PE Comparator Input <m>, is not selected as for Single-shot control.
- 0b1: The single PE Comparator Input <m>, is selected as for Single-shot control.

This bit is RES0 if m >= TRCIDR4.NUMPC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing TRCSPCICR<n>**

Must be programmed if implemented and any TRCRSCTRL<n>.GROUP == 0b0011 and TRCRSCTRL<n>.SINGLE_SHOT[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, TRCSSPCICR<m>

integer m = UInt(CRm<2:0>);
if m >= NUM_TRACE_SINGLE_SHOT_COMPARATOR_CONTROLS then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() & HaveEL(EL3) & & ESCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & & CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif EL2Enabled() & & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & & HDFGTR_EL2.TRC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() & & !HaveEL(EL3) || SCR_EL3.FGTEn == '1' & & HDFGTR_EL2.TRC == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    else
        AArch64.SystemAccessTrap(EL1, 0x18);
    else
        X[t, 64] = TRCSSPCICR[m];
elsif PSTATE.EL == EL2 then
    if Halted() & HaveEL(EL3) & & ESCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & & CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) & & CPTR_EL3.TTA == '1' then
        if Halted() & & ESCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = TRCSSPCICR[m];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCSSPCICR[m];

MSR TRCSSPCICR<m>, <Xt>

integer m = UInt(CRm<2:0>);
if m >= NUM_TRACE_SINGLE_SHOT_COMPARATOR_CONTROLS then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRCSSPCICR[m] = X[t, 64];
    elsif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
            UNDEFINED;
        elsif CPTR_EL2.TTA == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            else
                TRCSSPCICR[m] = X[t, 64];
    elsif PSTATE.EL == EL3 then
        if CPTR_EL3.TTA == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            TRCSSPCICR[m] = X[t, 64];
D17.4.55 TRCSTALLCTRLR, Stall Control Register

The TRCSTALLCTRLR characteristics are:

**Purpose**

Enables trace unit functionality that prevents trace unit buffer overflows.

**Configurations**

AArch64 System register TRCSTALLCTRLR bits [31:0] are architecturally mapped to External register TRCSTALLCTRLR[31:0].

This register is present only when FEAT_ETE is implemented and TRCIDR3.STALLCTL == 1. Otherwise, direct accesses to TRCSTALLCTRLR are UNDEFINED.

**Attributes**

TRCSTALLCTRLR is a 64-bit register.

**Field descriptions**

| Bit [63:14] | Reserved, RES0. |

*When TRCIDR3.NOOVERFLOW == 1:*

- Trace overflow prevention.
  - 0b0: Trace unit buffer overflow prevention is disabled.
  - 0b1: Trace unit buffer overflow prevention is enabled.

Note that enabling this feature might cause a significant performance impact.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

| Bit [12:9] | Reserved, RES0. |

| Bit [8] | ISTALL, bit [8] |

Instruction stall control. Controls if a trace unit can stall the PE when the trace buffer space is less than LEVEL.

- 0b0: The trace unit must not stall the PE.
- 0b1: The trace unit can stall the PE.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

| Bit [7:4] | Reserved, RES0. |
LEVEL, bits [3:0]
Threshold level field. The field can support 16 monotonic levels from 0b0000 to 0b1111.
The value 0b0000 defines the Minimal invasion level. This setting has a greater risk of a trace unit buffer overflow.
The value 0b1111 defines the Maximum invasion level. This setting has a reduced risk of a trace unit buffer overflow.
Note that for some implementations, invasion might occur at the minimal invasion level.
One or more of the least significant bits of LEVEL are permitted to be RES0. Arm recommends that LEVEL[3:2] are fully implemented. Arm strongly recommends that LEVEL[3] is always implemented. If one or more bits are RES0 and are written with a non-zero value, the effective value of LEVEL is rounded down to the nearest power of 2 value which has the RES0 bits as zero. For example, if LEVEL[1:0] are RES0 and a value of 0b1110 is written to LEVEL, the effective value of LEVEL is 0b1100.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSTALLCTLR
Must be programmed if implemented.
Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.
Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, TRCSTALLCTLR

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = TRCSTALLCTLR;
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      X[t, 64] = TRCSTALLCTLR;
    end
  else
    X[t, 64] = TRCSTALLCTLR;
  end
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL_EL3 && EDSCR_SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3_TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1_TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  end
elsif CPACR_EL1_TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && CPTR_EL2_TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL_EL3 || SCR_EL3_FGTEn == '1') && HDFGWTR_EL2_TRC == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL_EL3 && CPTR_EL3_TTA == '1' then
  if Halted() && EDSCR_SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL_EL3 && EDSCR_SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3_TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2_TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  end
elsif HaveEL_EL3 && CPTR_EL3_TTA == '1' then
  if Halted() && EDSCR_SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
elsif PSTATE_EL == EL3 then
  if CPTR_EL3_TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCSTALLCTRL = X[t, 64];
else
  TRCSTALLCTRL = X[t, 64];
end

### MSR TRCSTALLCTRL, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1011</td>
<td>0b000</td>
</tr>
</tbody>
</table>
D17.4.56 TRCSTATR, Trace Status Register

The TRCSTATR characteristics are:

Purpose

Returns the trace unit status.

Configurations

AArch64 System register TRCSTATR bits [31:0] are architecturally mapped to External register TRCSTATR[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCSTATR are UNDEFINED.

Attributes

TRCSTATR is a 64-bit register.

Field descriptions

Bits [63:2]

Reserved, RES0.

PMSTABLE, bit [1]

Programmers' model stable.

- 0b0 The programmers' model is not stable.
- 0b1 The programmers' model is stable.

Accessing this field has the following behavior:

- When the trace unit is enabled, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO

IDLE, bit [0]

Idle status.

- 0b0 The trace unit is not idle.
- 0b1 The trace unit is idle.

Accessing TRCSTATR

Accesses to this register use the following encodings in the System register encoding space:
### MRS <Xt>, TRCSTATR

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRSTATR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
elif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
elif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
elif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
elif X[t, 64] = TRCSTATR;
D17.4.57 TRCSYNCPR, Synchronization Period Register

The TRCSYNCPR characteristics are:

**Purpose**
Controls how often trace protocol synchronization requests occur.

**Configurations**
AArch64 System register TRCSYNCPR bits [31:0] are architecturally mapped to External register TRCSYNCPR[31:0].
This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCSYNCPR are UNDEFINED.

**Attributes**
TRCSYNCPR is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:5</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>31:0</td>
<td>PERIOD, bits [4:0]</td>
</tr>
</tbody>
</table>

**Bits [63:5]**
Reserved, RES0.

**PERIOD, bits [4:0]**
Defines the number of bytes of trace between each periodic trace protocol synchronization request.

- 0b00000: Trace protocol synchronization is disabled.
- 0b01000: Trace protocol synchronization request occurs after 2^8 bytes of trace.
- 0b01001: Trace protocol synchronization request occurs after 2^9 bytes of trace.
- 0b01010: Trace protocol synchronization request occurs after 2^10 bytes of trace.
- 0b01011: Trace protocol synchronization request occurs after 2^11 bytes of trace.
- 0b01100: Trace protocol synchronization request occurs after 2^12 bytes of trace.
- 0b01101: Trace protocol synchronization request occurs after 2^13 bytes of trace.
- 0b01110: Trace protocol synchronization request occurs after 2^14 bytes of trace.
- 0b01111: Trace protocol synchronization request occurs after 2^15 bytes of trace.
- 0b10000: Trace protocol synchronization request occurs after 2^16 bytes of trace.
- 0b10001: Trace protocol synchronization request occurs after 2^17 bytes of trace.
- 0b10010: Trace protocol synchronization request occurs after 2^18 bytes of trace.
- 0b10011: Trace protocol synchronization request occurs after 2^19 bytes of trace.
- 0b10100: Trace protocol synchronization request occurs after 2^20 bytes of trace.

Other values are reserved. If a reserved value is programmed into PERIOD, then the behavior of the synchronization period counter is CONSTRAINED UNPREDICTABLE and one of the following behaviors occurs:

- No trace protocol synchronization requests are generated by this counter.
- Trace protocol synchronization requests occur at the specified period.
- Trace protocol synchronization requests occur at some other UNKNOWN period which can vary.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSYNCPR

Must be programmed if TRCIDR3.SYNCPR == 0.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <X>, TRCSYNCPR**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1101</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED “EL3 trap priority
when SDD == '1'” && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCSYNCPR;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED “EL3 trap priority
when SDD == '1'” && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCSYNCPR;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCSYNCPR;
### MSR TRCSYNCPR, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1101</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TR == '1') then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if
  else
    TRCSYNCPR = X[t, 64];
  end if
else
  PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
      UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end if
    else
      TRCSYNCPR = X[t, 64];
    end if
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCSYNCPR = X[t, 64];
  end if
D17.4.58 TRCTRACEIDR, Trace ID Register

The TRCTRACEIDR characteristics are:

**Purpose**
Sets the trace ID for instruction trace.

**Configurations**
AArch64 System register TRCTRACEIDR bits [31:0] are architecturally mapped to External register TRCTRACEIDR[31:0].
This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCTRACEIDR are UNDEFINED.

**Attributes**
TRCTRACEIDR is a 64-bit register.

**Field descriptions**

![Diagram of field descriptions]

**Bits [63:7]**
Reserved, RES0.

**TRACEID, bits [6:0]**
Trace ID field. Sets the trace ID value for instruction trace. The width of the field is indicated by the value of TRCIDR5.TRACEIDSIZE. Unimplemented bits are RES0.
If an implementation supports AMBA ATB, then:
- The width of the field is 7 bits.
- Writing a reserved trace ID value does not affect behavior of the trace unit but it might cause UNPREDICTABLE behavior of the trace capture infrastructure.
See the AMBA ATB Protocol Specification for information about which ATID values are reserved.
The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing TRCTRACEIDR**
Must be programmed if implemented.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, TRCTRACEIDR

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if;
else
  X[t, 64] = TRCTRACEIDR;
else if PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if;
else
  X[t, 64] = TRCTRACEIDR;
else if PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCTRACEIDR;

MSR TRCTRACEIDR, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then

D17-6708 Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. Non-Confidential
AArch64.SystemAccessTrap(EL2, 0x18);
elif HaveEl(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  TRCTRACEIDR = X[t, 64];
elif PSTATE.EL == EL2 then
  if Halted() && HaveEl(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elif HaveEl(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  TRCTRACEIDR = X[t, 64];
elif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  TRCTRACEIDR = X[t, 64];
D17.4.59 TRCTSCTLR, Timestamp Control Register

The TRCTSCTLR characteristics are:

**Purpose**

Controls the insertion of global timestamps in the trace stream.

**Configurations**

AArch64 System register TRCTSCTLR bits [31:0] are architecturally mapped to External register TRCTSCTLR[31:0].

This register is present only when FEAT_ETE is implemented and TRCIDR0.TSSIZE != 0b00000. Otherwise, direct accesses to TRCTSCTLR are UNDEFINED.

**Attributes**

TRCTSCTLR is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:8]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[7]</td>
<td>EVENT_TYPE, bit</td>
</tr>
<tr>
<td>When TRCIDR4.NUMRSPAIR != 0b0000:</td>
<td>Chooses the type of Resource Selector.</td>
</tr>
<tr>
<td>0b0</td>
<td>A single Resource Selector.</td>
</tr>
<tr>
<td>TRCTSCTLR.EVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.</td>
<td></td>
</tr>
<tr>
<td>0b1</td>
<td>A Boolean-combined pair of Resource Selectors.</td>
</tr>
<tr>
<td>TRCTSCTLR.EVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCTSCTLR.EVENT.SEL[4] is RES0.</td>
<td></td>
</tr>
<tr>
<td>The reset behavior of this field is:</td>
<td></td>
</tr>
<tr>
<td>•</td>
<td>On a Trace unit reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>Otherwise:</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[6:5]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[4:0]</td>
<td>EVENT_SEL, bits</td>
</tr>
<tr>
<td>When TRCIDR4.NUMRSPAIR != 0b0000:</td>
<td>Defines the selected Resource Selector or pair of Resource Selectors. TRCTSCTLR.EVENT.TYPE controls whether TRCTSCTLR.EVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.</td>
</tr>
</tbody>
</table>
If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is **UNPREDICTABLE**, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is **UNPREDICTABLE**, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**

Reserved, **RES0**.

### Accessing TRCTSCCTRL

Must be programmed if `TRCCONFIGR.TS == 1`.

Writes are **CONSTRAINED UNPREDICTABLE** if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCTSCCTRL**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elseif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = TRCTSCCTRL;
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elseif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = TRCTSCCTRL;
  end
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = TRCTSCTLR;

**MSR TRCTSCTLR, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b1100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& CPTR_EL3.TTA == '1' then
        UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() \&\& CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() \&\& (HaveEL(EL3) \| SCR_EL3.FGTEn == '1') \&\& HDFGWTR_EL2.TR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) \&\& CPTR_EL3.TTA == '1' then
    if Halted() \&\& EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRCTSCTLR = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& CPTR_EL3.TTA == '1' then
        UNDEFINED;
elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) \&\& CPTR_EL3.TTA == '1' then
    if Halted() \&\& EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    TRCTSCTLR = X[t, 64];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    TRCTSCTLR = X[t, 64];
D17.4.60 TRCVICTLR, ViewInst Main Control Register

The TRCVICTLR characteristics are:

**Purpose**
Controls instruction trace filtering.

**Configurations**
AArch64 System register TRCVICTLR bits [31:0] are architecturally mapped to External register TRCVICTLR[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCVICTLR are UNDEFINED.

**Attributes**
TRCVICTLR is a 64-bit register.

**Field descriptions**

Bits [63:27]
Reserved, RES0.

EXLEVEL_RL_EL2, bit [26]

*When TRCIDR6.EXLEVEL_RL_EL2 == 1:*

Filter instruction trace for EL2 in Realm state.

0b0 When TRCVICTLR.EXLEVEL_NS_EL2 is 0 the trace unit generates instruction trace for EL2 in Realm state.

When TRCVICTLR.EXLEVEL_NS_EL2 is 1 the trace unit does not generate instruction trace for EL2 in Realm state.

0b1 When TRCVICTLR.EXLEVEL_NS_EL2 is 0 the trace unit does not generate instruction trace for EL2 in Realm state.

When TRCVICTLR.EXLEVEL_NS_EL2 is 1 the trace unit generates instruction trace for EL2 in Realm state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.
EXLEVEL_RL_EL1, bit [25]

When TRCIDR6.EXLEVEL_RL_EL1 == 1:

Filter instruction trace for EL1 in Realm state.

0b0 When TRCVICTLR.EXLEVEL_NS_EL1 is 0 the trace unit generates instruction trace for EL1 in Realm state.

When TRCVICTLR.EXLEVEL_NS_EL1 is 1 the trace unit does not generate instruction trace for EL1 in Realm state.

0b1 When TRCVICTLR.EXLEVEL_NS_EL1 is 0 the trace unit does not generate instruction trace for EL1 in Realm state.

When TRCVICTLR.EXLEVEL_NS_EL1 is 1 the trace unit generates instruction trace for EL1 in Realm state.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_RL_EL0, bit [24]

When TRCIDR6.EXLEVEL_RL_EL0 == 1:

Filter instruction trace for EL0 in Realm state.

0b0 When TRCVICTLR.EXLEVEL_NS_EL0 is 0 the trace unit generates instruction trace for EL0 in Realm state.

When TRCVICTLR.EXLEVEL_NS_EL0 is 1 the trace unit does not generate instruction trace for EL0 in Realm state.

0b1 When TRCVICTLR.EXLEVEL_NS_EL0 is 0 the trace unit does not generate instruction trace for EL0 in Realm state.

When TRCVICTLR.EXLEVEL_NS_EL0 is 1 the trace unit generates instruction trace for EL0 in Realm state.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES0.

EXLEVEL_NS_EL2, bit [22]

When Non-secure EL2 is implemented:

Filter instruction trace for EL2 in Non-secure state.

0b0 The trace unit generates instruction trace for EL2 in Non-secure state.

0b1 The trace unit does not generate instruction trace for EL2 in Non-secure state.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_NS_EL1, bit [21]

When Non-secure EL1 is implemented:

Filter instruction trace for EL1 in Non-secure state.

0b0 The trace unit generates instruction trace for EL1 in Non-secure state.
0b1 The trace unit does not generate instruction trace for EL1 in Non-secure state.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EXLEVEL_NS_EL0, bit [20]
When Non-secure EL0 is implemented:
Filter instruction trace for EL0 in Non-secure state.
0b0 The trace unit generates instruction trace for EL0 in Non-secure state.
0b1 The trace unit does not generate instruction trace for EL0 in Non-secure state.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EXLEVEL_S_EL3, bit [19]
When EL3 is implemented:
Filter instruction trace for EL3.
0b0 The trace unit generates instruction trace for EL3.
0b1 The trace unit does not generate instruction trace for EL3.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EXLEVEL_S_EL2, bit [18]
When EL2 is implemented and FEAT_SEL2 is implemented:
Filter instruction trace for EL2 in Secure state.
0b0 The trace unit generates instruction trace for EL2 in Secure state.
0b1 The trace unit does not generate instruction trace for EL2 in Secure state.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EXLEVEL_S_EL1, bit [17]
When Secure EL1 is implemented:
Filter instruction trace for EL1 in Secure state.
0b0 The trace unit generates instruction trace for EL1 in Secure state.
0b1 The trace unit does not generate instruction trace for EL1 in Secure state.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.
EXLEVEL_S_EL0, bit [16]

When Secure EL0 is implemented:

Filter instruction trace for EL0 in Secure state.

0b0 The trace unit generates instruction trace for EL0 in Secure state.
0b1 The trace unit does not generate instruction trace for EL0 in Secure state.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bits [15:12]
Reserved, RES0.

TRCERR, bit [11]

When TRCIDR3.TRCERR == 1:
Controls the forced tracing of System Error exceptions.

0b0 Forced tracing of System Error exceptions is disabled.
0b1 Forced tracing of System Error exceptions is enabled.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

TRCRESET, bit [10]
Controls the forced tracing of PE Resets.

0b0 Forced tracing of PE Resets is disabled.
0b1 Forced tracing of PE Resets is enabled.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SSSTATUS, bit [9]
ViewInst start/stop function status.

0b0 Stopped State.
The ViewInst start/stop function is in the stopped state.

0b1 Started State.
The ViewInst start/stop function is in the started state.

Before software enables the trace unit, it must write to this field to set the initial state of the ViewInst start/stop function. If the ViewInst start/stop function is not used then set this field to 1. Arm recommends that the value of this field is set before each trace session begins.

If the trace unit becomes disabled while a start point or stop point is still speculative, then the value of TRCVICTLR.SSSTATUS is UNKNOWN and might represent the result of a speculative start point or stop point.

If software which is running on the PE being traced disables the trace unit, either by clearing TRCPRGCTLR.EN or locking the OS Lock, Arm recommends that a DSB and an ISB instruction are executed before disabling the trace unit to prevent any start points or stop points being speculative at the point of disabling the trace unit. This procedure assumes that all start points or stop points occur before the barrier instructions are executed. The procedure does not guarantee that there are no speculative start points or stop points when disabling, although it helps minimize the probability.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:
• RES1 if all of the following are true:
  — TRCIDR4.NUMACPAIRS == 0b0000
  — TRCIDR4.NUMPC == 0b0000
• Otherwise, access to this field is RW

**Bit [8]**

Reserved, RES0.

**EVENT TYPE, bit [7]**

*When TRCIDR4.NUMRSPAIR != 0b0000:*

Chooses the type of Resource Selector.

0b0  A single Resource Selector.
    TRCVICTLR.EVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

0b1  A Boolean-combined pair of Resource Selectors.
    TRCVICTLR.EVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCVICTLR.EVENT.SEL[4] is RES0.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

**Bits [6:5]**

Reserved, RES0.

**EVENT_SEL, bits [4:0]**

*When TRCIDR4.NUMRSPAIR != 0b0000:*

**EVENT_SEL**

Defines the selected Resource Selector or pair of Resource Selectors. TRCVICTLR.EVENT.TYPE controls whether TRCVICTLR.EVENT_SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

*When TRCIDR4.NUMRSPAIR == 0b0000:*

**Reserved**

This field is reserved:
• Bits [4:1] are RES0.
• Bit [0] is RES1.

*Otherwise:*

Reserved, RES0.
Accessing TRCVICTLR

Must be programmed.

Reads from this register might return an UNDEFINED value if the trace unit is not in either of the Idle or Stable states.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCVICTLR**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGTR_EL2.TRVCICTLR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    elsif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
            UNDEFINED;
        elsif CPTR_EL2.TTA == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    elsif PSTATE.EL == EL3 then
        if CPTR_EL3.TTA == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = TRCVICTLR;
        end
    else
        X[t, 64] = TRCVICTLR;
    end
    if PSTATE.EL == EL0 then
        UNDEFINED;
    elsif PSTATE.EL == EL1 then
        X[t, 64] = TRCVICTLR;
    else
        if CPTR_EL3.TTA == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = TRCVICTLR;
        end
    end

**MSR TRCVICTLR, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>
if Halted() && HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() & CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGWTR_EL2.TRCVICTLR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
    if Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRCVICTLR = X[t, 64];
    elsif PSTATE.EL == EL2 then
        if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
            UNDEFINED;
        elsif CPTR_EL2.TTA == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
            if Halted() & EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end
        else
            TRCVICTLR = X[t, 64];
        end
    elsif PSTATE.EL == EL3 then
        if CPTR_EL3.TTA == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            TRCVICTLR = X[t, 64];
        end
D17.4.61 TRCVIIECTLR, ViewInst Include/Exclude Control Register

The TRCVIIECTLR characteristics are:

**Purpose**

Use this to select, or read, the Address Range Comparators for the ViewInst include/exclude function.

**Configurations**

AArch64 System register TRCVIIECTLR bits [31:0] are architecturally mapped to External register TRCVIIECTLR[31:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR4.NUMACPAIRS) > 0. Otherwise, direct accesses to TRCVIIECTLR are UNDEFINED.

**Attributes**

TRCVIIECTLR is a 64-bit register.

**Field descriptions**

![Field description diagram]

**Bits [63:24]**

Reserved, RES0.

**EXCLUDE[<m>], bit [m+16], for m = 7 to 0**

Exclude Address Range Comparator <m>. Selects whether Address Range Comparator <m> is in use with the ViewInst exclude function.

0b0 The address range that Address Range Comparator <m> defines, is not selected for the ViewInst exclude function.

0b1 The address range that Address Range Comparator <m> defines, is selected for the ViewInst exclude function.

This bit is RES0 if m >= TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Bits [15:8]**

Reserved, RES0.

**INCLUDE[<m>], bit [m], for m = 7 to 0**

Include Address Range Comparator <m>.

Selects whether Address Range Comparator <m> is in use with the ViewInst include function.
Selecting no comparators for the ViewInst include function indicates that all instructions are included by default.

The ViewInst exclude function then indicates which ranges are excluded.

0b0  The address range that Address Range Comparator <m> defines, is not selected for the ViewInst include function.

0b1  The address range that Address Range Comparator <m> defines, is selected for the ViewInst include function.

This bit is RES0 if m >= TRCIDR4[NUMACPAIRS].

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

### Accessing TRCVIIECTLR

Must be programmed if TRCIDR4[NUMACPAIRS] > 0b0000.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, TRCVIIECTLR**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elseif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(E1, 0x18);
  elseif EL2Enabled() & CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(E2, 0x18);
  elseif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGRTR_EL2.TRC == '1' then
    AArch64.SystemAccessTrap(E2, 0x18);
  elseif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(E3, 0x18);
    else
      X[t, 64] = TRCVIIECTLR;
  elsif PSTATE.EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
      UNDEFINED;
    elseif CPTR_EL2.TTA == '1' then
      AArch64.SystemAccessTrap(E2, 0x18);
    elseif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
      if Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(E3, 0x18);
      else
        X[t, 64] = TRCVIIECTLR;
  elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
      AArch64.SystemAccessTrap(E3, 0x18);
else
  X[t, 64] = TRCVIIECTLR;

### MSR TRCVIIECTLR, <xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
  AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() & CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGWTR_EL2.TRC == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    TRCVIIECTLR = X[t, 64];
  endif
endif
else
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  TRCVIIECTLR = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif CPTR_EL3.TTA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    TRCVIIECTLR = X[t, 64];
  endif
endif
else
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  TRCVIIECTLR = X[t, 64];
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  TRCVIIECTLR = X[t, 64];
D17.4.62 TRCVIPCSSCTLR, ViewInst Start/Stop PE Comparator Control Register

The TRCVIPCSSCTLR characteristics are:

**Purpose**

Use this to select, or read, which PE Comparator Inputs can control the ViewInst start/stop function.

**Configurations**

AArch64 System register TRCVIPCSSCTLR bits [31:0] are architecturally mapped to External register TRCVIPCSSCTLR[31:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR4.NUMPC) > 0. Otherwise, direct accesses to TRCVIPCSSCTLR are UNDEFINED.

**Attributes**

TRCVIPCSSCTLR is a 64-bit register.

**Field descriptions**

![Field diagram]

**Bits [63:24]**

Reserved, RES0.

**STOP[<m>], bit [m+16], for m = 7 to 0**

Selects whether PE Comparator Input <m> is in use with ViewInst start/stop function, for the purpose of stopping trace.

- **0b0** The PE Comparator Input <m>, is not selected as a stop resource.
- **0b1** The PE Comparator Input <m>, is selected as a stop resource.

This bit is RES0 if m >= TRCIDR4.NUMPC.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Bits [15:8]**

Reserved, RES0.

**START[<m>], bit [m], for m = 7 to 0**

Selects whether PE Comparator Input <m> is in use with ViewInst start/stop function, for the purpose of starting trace.

- **0b0** The PE Comparator Input <m>, is not selected as a start resource.
- **0b1** The PE Comparator Input <m>, is selected as a start resource.

This bit is RES0 if m >= TRCIDR4.NUMPC.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
**Accessing TRCVIPCSSCTLR**

Must be programmed if TRCIDR4.NUMPC != 0b0000.

Writes are **CONSTRAINED UNPREDICTABLE** if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

### MRS <Xt>, TRCVIPCSSCTLR

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = TRCVIPCSSCTLR;
    end;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = TRCVIPCSSCTLR;
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = TRCVIPCSSCTLR;
    end;
```

### MSR TRCVIPCSSCTLR, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.TRC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = TRCVIPCSSCTLR;
    end;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = TRCVIPCSSCTLR;
    end;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = TRCVIPCSSCTLR;
    end;
```
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
  UNDEFINED;
elsf CPACR_EL1.TTA == '1' then
  AArch64.SystemAccessTrap(EL1, 0x18);
elsf EL2Enabled() && CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsf EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsf HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
elsf TRCVIPCSSCTLR = X[t, 64];
elsf PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsf CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsf HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
elsf TRCVIPCSSCTLR = X[t, 64];
elsf PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
elsf TRCVIPCSSCTLR = X[t, 64];
D17.4.63  TRCVISSCTLR, ViewInst Start/Stop Control Register

The TRCVISSCTLR characteristics are:

**Purpose**

Use this to select, or read, the Single Address Comparators for the ViewInst start/stop function.

**Configurations**

AArch64 System register TRCVISSCTRL bits [31:0] are architecturally mapped to External register TRCVISSCTRL[31:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR4.NUMACPAIRS) > 0. Otherwise, direct accesses to TRCVISSCTRL are UNDEFINED.

**Attributes**

TRCVISSCTRL is a 64-bit register.

**Field descriptions**

| Bits [63:32] | Reserved, RES0. |

**STOP[<m>], bit [m+16], for m = 15 to 0**

Selects whether Single Address Comparator <m> is used with the ViewInst start/stop function, for the purpose of stopping trace.

- 0b0  The Single Address Comparator <m>, is not selected as a stop resource.
- 0b1  The Single Address Comparator <m>, is selected as a stop resource.

This bit is RES0 if m >= 2 × TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
START[<m>], bit [m], for m = 15 to 0
Selects whether Single Address Comparator <m> is used with the ViewInst start/stop function, for
the purpose of starting trace.
0b0 The Single Address Comparator <m>, is not selected as a start resource.
0b1 The Single Address Comparator <m>, is selected as a start resource.
This bit is res0 if m >= 2 × TRCIDR4.NUMACPAIRS.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCVISSCTRL
Must be programmed if TRCIDR4.NUMACPAIRS > 0b0000.
For any 2 comparators selected for the ViewInst start/stop function, the comparator containing the lower address
must be a lower numbered comparator.
Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

\[ \text{MRS } <Xt>, \text{ TRCVISSCTRL} \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>


if PSTATE.EL == EL0 then
UNDONE;
elif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1′ && CPTR_EL3.TTA == '1′ then
    UNDONE;
elif EL2Enabled() && CPTR_EL2.TTA == '1′ then
    AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1′) && HDFGRTR_EL2.TRC == '1′ then
    AArch64.SystemAccessTrap(EL2, 0x18);
elif HaveEL(EL3) && CPTR_EL3.TTA == '1′ then
    if Halted() && EDSCR.SDD == '1′ then
        UNDONE;
elif CPACR_EL1.TTA == '1′ then
    AArch64.SystemAccessTrap(EL1, 0x18);
elif EL2Enabled() && CPTR_EL2.TTA == '1′ then
    AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1′) && HDFGRTR_EL2.TRC == '1′ then
    AArch64.SystemAccessTrap(EL2, 0x18);
elif HaveEL(EL3) && CPTR_EL3.TTA == '1′ then
    if Halted() && EDSCR.SDD == '1′ then
        UNDONE;
elif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1′ && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1′ && CPTR_EL3.TTA == '1′ then
    UNDONE;
elif CPTR_EL2.TTA == '1′ then
    AArch64.SystemAccessTrap(EL2, 0x18);
elif HaveEL(EL3) && CPTR_EL3.TTA == '1′ then
    if Halted() && EDSCR.SDD == '1′ then
        UNDONE;
elif EL2Enabled() && CPTR_EL2.TTA == '1′ then
    AArch64.SystemAccessTrap(EL2, 0x18);
elif HaveEL(EL3) && CPTR_EL3.TTA == '1′ then
    if Halted() && EDSCR.SDD == '1′ then
        UNDONE;
elif CPACR_EL1.TTA == '1′ then
    AArch64.SystemAccessTrap(EL1, 0x18);
elif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1′ then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCVISSCTRL;
elif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1′ && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1′ && CPTR_EL3.TTA == '1′ then
    UNDONE;
elif CPTR_EL2.TTA == '1′ then
    AArch64.SystemAccessTrap(EL2, 0x18);
elif HaveEL(EL3) && CPTR_EL3.TTA == '1′ then
    if Halted() && EDSCR.SDD == '1′ then
        UNDONE;
elif EL2Enabled() && CPTR_EL2.TTA == '1′ then
    AArch64.SystemAccessTrap(EL2, 0x18);
elif HaveEL(EL3) && CPTR_EL3.TTA == '1′ then
    if Halted() && EDSCR.SDD == '1′ then
        UNDONE;
elif CPACR_EL1.TTA == '1′ then
    AArch64.SystemAccessTrap(EL1, 0x18);
elif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1′ then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCVISSCTL;

**MSR TRCVISSCTL, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
  AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRC == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  TRCVISSCTL = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  TRCVISSCTL = X[t, 64];
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  TRCVISSCTL = X[t, 64];
D17.4.64 TRCVMIDCCTRLR0, Virtual Context Identifier Comparator Control Register 0

The TRCVMIDCCTRLR0 characteristics are:

**Purpose**

Virtual Context Identifier Comparator mask values for the TRCVMIDCVR<\text{n}> registers, where \text{n}=0-3.

**Configurations**

AArch64 System register TRCVMIDCCTRLR0 bits [31:0] are architecturally mapped to External register TRCVMIDCCTRLR0[31:0].

This register is present only when FEAT_ETE is implemented, UInt(TRCIDR4.NUMVMIDC) > 0x0 and UInt(TRCIDR2.VMIDSIZE) > 0. Otherwise, direct accesses to TRCVMIDCCTRLR0 are UNDEFINED.

**Attributes**

TRCVMIDCCTRLR0 is a 64-bit register.

**Field descriptions**

![Register Diagram]

- **Bits [63:32]**
  - Reserved, RES0.

- **COMP3\[<m>\]**, bit [m+24], for m = 7 to 0
  
  *When UInt(TRCIDR4.NUMVMIDC) > 3:*
  
  TRCVMIDCVR3 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR3. Each bit in this field corresponds to a byte in TRCVMIDCVR3.

  - 0b0: The trace unit includes TRCVMIDCVR3[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
  - 0b1: The trace unit ignores TRCVMIDCVR3[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

  This bit is RES0 if m >= TRCIDR2.VMIDSIZE.

  The reset behavior of this field is:
  
  - On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

- **COMP2\[<m>\]**, bit [m+16], for m = 7 to 0
  
  *When UInt(TRCIDR4.NUMVMIDC) > 2:*
  
  TRCVMIDCVR2 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR2. Each bit in this field corresponds to a byte in TRCVMIDCVR2.

  - 0b0: The trace unit includes TRCVMIDCVR2[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
  - 0b1: The trace unit ignores TRCVMIDCVR2[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

  This bit is RES0 if m >= TRCIDR2.VMIDSIZE.

  The reset behavior of this field is:
  
  - On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
COMP1[<m>], bit [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 1:

TRCVmIDCVR1 mask control. Specifies the mask value that the trace unit applies to TRCVmIDCVR1. Each bit in this field corresponds to a byte in TRCVmIDCVR1.

0b0 The trace unit includes TRCVmIDCVR1[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

0b1 The trace unit ignores TRCVmIDCVR1[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

This bit is RES0 if m >= TRCIDR2.VMIDSIZE.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

COMP0[<m>], bit [m], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 0:

TRCVmIDCVR0 mask control. Specifies the mask value that the trace unit applies to TRCVmIDCVR0. Each bit in this field corresponds to a byte in TRCVmIDCVR0.

0b0 The trace unit includes TRCVmIDCVR0[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

0b1 The trace unit ignores TRCVmIDCVR0[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

This bit is RES0 if m >= TRCIDR2.VMIDSIZE.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCVmIDCCTLR0

If software uses the TRCVmIDCVR<n> registers, where n=0-3, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCVmIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCVmIDCVR<n> is not 0x00, the behavior of the Virtual Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

<table>
<thead>
<tr>
<th>MRS &lt;Xt&gt;, TRCVmIDCCTLR0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
</tr>
<tr>
<td>0b10</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD = '1'" & CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() & CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDGRTR_EL2.TRC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    endif
endif
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCVMIDCCTL0;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  else if CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = TRCVMIDCCTL0;
else
  X[t, 64] = TRCVMIDCCTL0;
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCVMIDCCTL0;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  else if CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRCE == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCVMIDCCTL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  else if CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
else

MSR TRCVMIDCCTL0, <Xt>  

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0011</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  else if CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.TRCE == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCVMIDCCTL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  else if CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
else

ARM DDI 0487I.a  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  D17-6731
ID081822  Non-Confidential
TRCVMIDCCTL0 = X[t, 64];
elsif PSTATE_EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCVMIDCCTL0 = X[t, 64];

D17.4.65 TRCVMIDCCTLR1, Virtual Context Identifier Comparator Control Register 1

The TRCVMIDCCTLR1 characteristics are:

**Purpose**

Virtual Context Identifier Comparator mask values for the TRCVMIDCVR<\textit{n}> registers, where \textit{n}=4-7.

**Configurations**

AArch64 System register TRCVMIDCCTLR1 bits [31:0] are architecturally mapped to External register TRCVMIDCCTLR1[31:0].

This register is present only when FEAT_ETE is implemented, UInt(TRCIDR4.NUMVMIDC) > 0x4 and UInt(TRCIDR2.VMIDSIZE) > 0. Otherwise, direct accesses to TRCVMIDCCTLR1 are UNDEFINED.

**Attributes**

TRCVMIDCCTLR1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:32]</th>
<th>RES0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits [31:24]</td>
<td>COMP7[&lt;m&gt;]</td>
</tr>
<tr>
<td>Bits [15:8]</td>
<td>COMP6[&lt;m&gt;]</td>
</tr>
<tr>
<td>Bits [7:0]</td>
<td>COMP5[&lt;m&gt;]</td>
</tr>
<tr>
<td>Bits [0]</td>
<td>COMP4[&lt;m&gt;]</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**COMP7\[<m>\]**, bit [m+24], for m = 7 to 0

*When UInt(TRCIDR4.NUMVMIDC) > 7:*

TRCVMIDCVR7 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR7. Each bit in this field corresponds to a byte in TRCVMIDCVR7.

0b0 The trace unit includes TRCVMIDCVR7[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

0b1 The trace unit ignores TRCVMIDCVR7[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

This bit is RES0 if m >= TRCIDR2.VMIDSIZE.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**COMP6\[<m>\]**, bit [m+16], for m = 7 to 0

*When UInt(TRCIDR4.NUMVMIDC) > 6:*

TRCVMIDCVR6 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR6. Each bit in this field corresponds to a byte in TRCVMIDCVR6.

0b0 The trace unit includes TRCVMIDCVR6[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

0b1 The trace unit ignores TRCVMIDCVR6[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

This bit is RES0 if m >= TRCIDR2.VMIDSIZE.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
COMP5[<m>], bit [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 5:

TRCVMIDCVR5 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR5. Each bit in this field corresponds to a byte in TRCVMIDCVR5.

0b0 The trace unit includes TRCVMIDCVR5[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

0b1 The trace unit ignores TRCVMIDCVR5[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

This bit is RES0 if m >= TRCIDR2.VMIDSIZE.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

COMP4[<m>], bit [m], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 4:

TRCVMIDCVR4 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR4. Each bit in this field corresponds to a byte in TRCVMIDCVR4.

0b0 The trace unit includes TRCVMIDCVR4[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

0b1 The trace unit ignores TRCVMIDCVR4[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

This bit is RES0 if m >= TRCIDR2.VMIDSIZE.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCVMIDCCTLR1

If software uses the TRCVMIDCVR<n> registers, where n=4-7, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCVMIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCVMIDCVR<n> is not 0x00, the behavior of the Virtual Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, TRCVMIDCCTLR1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0011</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() & & HaveEL(EL3) & & EDSCR.SDD == '1' & & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & & CPTR_EL3.TTA == '1' then
        UNDEFINED;
    elsif CPACR_EL1.TTA == '1' then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() & & CPTR_EL2.TTA == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & & HDFGTR_EL2.TRC == '1' then...
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCVMDCCRL1;
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end;
else
  X[t, 64] = TRCVMDCCRL1;
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCVMDCCRL1;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() & CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & ((HaveEL(EL3)) || SCR_EL3.FGTEn == '1') & HDFGWTR_EL2.TRSC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end;
else
  TRCVMDCCRL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) & CPTR_EL3.TTA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
end;

---

### MSR TRCVMDCCRL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b001</td>
<td>0b001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

---
TRCMIDCCTRL1 = X[t, 64];
elsif PSTATE_EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    TRCMIDCCTRL1 = X[t, 64];
D17.4.66 TRCVMIDCVR\(<n>\), Virtual Context Identifier Comparator Value Register \(<n>\), \(n = 0 - 7\)

The TRCVMIDCVR\(<n>\) characteristics are:

**Purpose**
Contains the Virtual Context Identifier Comparator value.

**Configurations**
AArch64 System register TRCVMIDCVR\(<n>\) bits [63:0] are architecturally mapped to External register TRCVMIDCVR\(<n>\)[63:0].
This register is present only when FEAT_ETE is implemented and UInt(TRCIDR4.NUMVMIDC) > n. Otherwise, direct accesses to TRCVMIDCVR\(<n>\) are UNDEFINED.

**Attributes**
TRCVMIDCVR\(<n>\) is a 64-bit register.

**Field descriptions**

![Field descriptions diagram]

VALUE, bits [63:0]
Virtual context identifier value. The width of this field is indicated by TRCIDR2.VMIDSIZE.
Unimplemented bits are RES0. After a PE Reset, the trace unit assumes that the Virtual context identifier is zero until the PE updates the Virtual context identifier.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing TRCVMIDCVR\(<n>\)**
Must be programmed if any of the following are true:
• TRCRSCTL\(<n>\).GROUP == 0b0111 and TRCRSCTL\(<n>\).VMID[n] == 1.
• TRCACATR\(<n>\).CONTEXTTYPE == 0b10 or 0b11 and TRCACATR\(<n>\).CONTEXT == n.
Accesses to this register use the following encodings in the System register encoding space:

\[ MRS \ <Xt>, \ TRCVMIDCVR<m> \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b0011</td>
<td>m[2:0]:0b0</td>
<td>0b001</td>
</tr>
</tbody>
</table>

integer \(m = \text{UInt}(\text{CRm}<3:1>)\);
if \(m \geq \text{NUM_TRACE_VIRTUAL_CONTEXT_IDENTIFIER_COMPARATORS}\) then
UNDEFINED;
elsif PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1' && CPTR_EL3.TTA == '1' then
  UNDEFINED;
elsif CPACR_EL1.TTA == '1' then
  AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGTR_EL2.TRC == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCVMIDCVR[m];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGTR_EL2.TRC == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
else
  TRCVMIDCVR[m] = X[t, 64];
elsif PSTATE.EL == EL2 then
  integer m = UInt(CRm<3:1>);
if m >= NUM_TRACE_VIRTUAL_CONTEXT_IDENTIFIER_COMPARATORS then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
  elsif CPACR_EL1.TTA == '1' then
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TTA == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGTR_EL2.TRC == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
else
  TRCVMIDCVR[m] = X[t, 64];
elsif PSTATE.EL == EL3 then
  if CPTR_EL3.TTA == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = TRCVMIDCVR[m];
else
  TRCVMIDCVR[m] = X[t, 64];

MSR TRCVMIDCVR<m>, <Xt>
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TTA == '1' then
    UNDEFINED;
elsif CPTR_EL2.TTA == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TTA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRCVMIDCVR[m] = X[t, 64];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TTA == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TRCVMIDCVR[m] = X[t, 64];
D17.5 Performance Monitors registers

This section lists the Performance Monitoring registers in AArch64.
D17.5.1 PMCCFILTR_EL0, Performance Monitors Cycle Count Filter Register

The PMCCFILTR_EL0 characteristics are:

**Purpose**
Determines the modes in which the Cycle Counter, PMCCNTR_EL0, increments.

**Configurations**
AArch64 System register PMCCFILTR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCCFILTR[31:0].
AArch64 System register PMCCFILTR_EL0 bits [31:0] are architecturally mapped to External register PMCCFILTR_EL0[31:0].
This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCCFILTR_EL0 are UNDEFINED.

**Attributes**
PMCCFILTR_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>31</td>
<td>P, bit [31]</td>
</tr>
<tr>
<td>30</td>
<td>U, bit [30]</td>
</tr>
<tr>
<td>32</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>29</td>
<td>NSK</td>
</tr>
<tr>
<td>28</td>
<td>NSU</td>
</tr>
<tr>
<td>27</td>
<td>NSH</td>
</tr>
<tr>
<td>26</td>
<td>RLH</td>
</tr>
<tr>
<td>25</td>
<td>RLU</td>
</tr>
<tr>
<td>24</td>
<td>RLK</td>
</tr>
<tr>
<td>23</td>
<td>T</td>
</tr>
<tr>
<td>22</td>
<td>M</td>
</tr>
<tr>
<td>21</td>
<td>SH</td>
</tr>
<tr>
<td>20</td>
<td>SH</td>
</tr>
<tr>
<td>19</td>
<td>RES0</td>
</tr>
<tr>
<td>0</td>
<td>RES0</td>
</tr>
</tbody>
</table>

**Bits [63:32]**
Reserved, RES0.

**P, bit [31]**
Privileged filtering bit. Controls counting in EL1.
If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the PMCCFILTR_EL0.NSK bit.
If FEAT_RME is implemented, then counting in Realm EL1 is further controlled by the PMCCFILTR_EL0.RLK bit.

| 0b0  | Count cycles in EL1. |
| 0b1  | Do not count cycles in EL1. |

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**U, bit [30]**
User filtering bit. Controls counting in EL0.
If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the PMCCFILTR_EL0.NSU bit.
If FEAT_RME is implemented, then counting in Realm EL0 is further controlled by the PMCCFILTR_EL0.RLU bit.

| 0b0  | Count cycles in EL0. |
| 0b1  | Do not count cycles in EL0. |
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSK, bit [29]

When EL3 is implemented:
Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1.
If the value of this bit is equal to the value of the PMCCFILTR_EL0.P bit, cycles in Non-secure EL1 are counted.
Otherwise, cycles in Non-secure EL1 are not counted.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:
Non-secure EL0 (Unprivileged) filtering bit. Controls counting in Non-secure EL0.
If the value of this bit is equal to the value of the PMCCFILTR_EL0.U bit, cycles in Non-secure EL0 are counted.
Otherwise, cycles in Non-secure EL0 are not counted.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:
EL2 (Hypervisor) filtering bit. Controls counting in EL2.
If Secure EL2 is implemented, and EL3 is implemented, counting in Secure EL2 is further controlled by the PMCCFILTR_EL0.SH bit.
If FEAT_RME is implemented, then counting in Realm EL2 is further controlled by the PMCCFILTR_EL0.RLH bit.
0b0 Do not count cycles in EL2.
0b1 Count cycles in EL2.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

M, bit [26]

When EL3 is implemented:
Secure EL3 filtering bit.
If the value of this bit is equal to the value of the PMCCFILTR_EL0.P bit, cycles in Secure EL3 are counted.
Otherwise, cycles in Secure EL3 are not counted.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Otherwise:
Reserved, RES0.

Bit [25]
Reserved, RES0.

SH, bit [24]

When FEAT_SEL2 is implemented and EL3 is implemented:
Secure EL2 filtering.
If the value of this bit is not equal to the value of the PMCCFILTR_EL0.NSH bit, cycles in Secure
EL2 are counted.
Otherwise, cycles in Secure EL2 are not counted.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

T, bit [23]

When FEAT_TME is implemented:
Non-transactional state filtering bit.
0b0 This bit has no effect on filtering of cycles.
0b1 Do not count cycles in Non-transactional state.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

RLK, bit [22]

When FEAT_RME is implemented:
Realm EL1 (kernel) filtering bit. Controls counting in Realm EL1.
If the value of this bit is equal to the value of the PMCCFILTR_EL0.P bit, cycles in Realm EL1 are
counted.
Otherwise, cycles in Realm EL1 are not counted.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

RLU, bit [21]

When FEAT_RME is implemented:
Realm EL0 (unprivileged) filtering bit. Controls counting in Realm EL0.
If the value of this bit is equal to the value of the PMCCFILTR_EL0.U bit, cycles in Realm EL0 are
counted.
Otherwise, cycles in Realm EL0 are not counted.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.
RLH, bit [20]

When FEAT_RME is implemented:

Realm EL2 filtering bit. Controls counting in Realm EL2.

If the value of this bit is not equal to the value of the PMCCFILTR_EL0.NSH bit, cycles in Realm EL2 are counted.

Otherwise, cycles in Realm EL2 are not counted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [19:0]

Reserved, RES0.

Accessing PMCCFILTR_EL0

PMCCFILTR_EL0 can also be accessed by using PMXEVTYPER_EL0 with PMSELR_EL0.SEL set to 0b11111.

Accesses to this register use the following encodings in the System register encoding space:

\[ MRS <Xt>, \text{PMCCFILTR_EL0} \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b1111</td>
<td>0b11</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    end
  elseif EL2Enabled() && SCR_EL3.FGTEn == '1' && HDFGRTR_EL2.PMCCFILTR_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsifHaveEL(EL3) && HDFGRTR_EL2.PMCCFILTR_EL0 == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  elsif EL2Enabled() && SCR_EL3.FGTEn == '1' && HDFGRTR_EL2.PMCCFILTR_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && HDFGRTR_EL2.PMCCFILTR_EL0 == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = PMCCFILTR_EL0;
  end
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCCFILTR_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HDFGRTR_EL2.PMCCFILTR_EL0 == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = PMCCFILTR_EL0;
else
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
  elseif EL2Enabled() && HDFGRTR_EL2.PMCCFILTR_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = PMCCFILTR_EL0;
end


```c
AArch64.SystemAccessTrap(EL3, 0x18); 
else
  X[t, 64] = PMCCFILTR_EL0;
elsif PSTATE.EL == EL2 then 
  if Halted() & HaveEL(EL3) & EDSR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then 
    UNDEFINED;
elsif HaveEL(EL3) & MDCR_EL3.TPM == '1' then 
  if Halted() & EDSR.SDD == '1' then 
    UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = PMCCFILTR_EL0;
elsif PSTATE.EL == EL3 then 
  X[t, 64] = PMCCFILTR_EL0;
else
  PMCCFILTR_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then 
  if Halted() & HaveEL(EL3) & EDSR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then 
    UNDEFINED;
elsif EL2Enabled() & HCR_EL2.TGE == '1' then 
  if PMUSERENR_EL0.EN == '0' then 
    AArch64.SystemAccessTrap(EL1, 0x18);
else
  AArch64.SystemAccessTrap(EL1, 0x18);
else
  PMCCFILTR_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then 
  if Halted() & HaveEL(EL3) & EDSR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then 
    UNDEFINED;
elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCCFILTR_EL0 == '1' then 
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  PMCCFILTR_EL0 = X[t, 64];
```
if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    PMCCFILTR_EL0 = X[t, 64];
elsif PSTATE_EL == EL3 then
    PMCCFILTR_EL0 = X[t, 64];
D17.5.2 PMCCNTR_EL0, Performance Monitors Cycle Count Register

The PMCCNTR_EL0 characteristics are:

**Purpose**

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. See *Time as measured by the Performance Monitors cycle counter on page D11-5246* for more information. PMCCFILTR_EL0 determines the modes and states in which the PMCCNTR_EL0 can increment.

**Configurations**

AArch64 System register PMCCNTR_EL0 bits [63:0] are architecturally mapped to AArch32 System register PMCCNTR[63:0].

AArch64 System register PMCCNTR_EL0 bits [63:0] are architecturally mapped to External register PMCCNTR_EL0[63:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCCNTR_EL0 are UNDEFINED.

All counters are subject to any changes in clock frequency, including clock stopping caused by the WFI and WFE instructions. This means that it is CONSTRAINED UNPREDICTABLE whether or not PMCCNTR_EL0 continues to increment when clocks are stopped by WFI and WFE instructions.

**Attributes**

PMCCNTR_EL0 is a 64-bit register.

**Field descriptions**

![Field descriptions diagram]

**CCNT, bits [63:0]**

Cycle count. Depending on the values of PMCR_EL0.\{LC,D\}, this field increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to PMCR_EL0.C sets this field to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing PMCCNTR_EL0**

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, PMCCNTR_EL0

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1101</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif PMUSERENR_EL0.<CR,EN> == '00' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    end ifEl2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCCNTR_EL0 == '1' then
      AArch64.SystemAccessTrap(EL1, 0x18);
    else
      if EL2Enabled() && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      else
        if HaveEL(EL3) && MDCR_EL3.TPM == '1' then
          if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
          else
            AArch64.SystemAccessTrap(EL3, 0x18);
          end if
        else
          X[t, 64] = PMCCNTR_EL0;
        end if
      end if
    end if
  end if
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCCNTR_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if
  else
    X[t, 64] = PMCCNTR_EL0;
  end if
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if
  else
    X[t, 64] = PMCCNTR_EL0;
  end if
elsif PSTATE_EL == EL3 then
  X[t, 64] = PMCCNTR_EL0;
MSR PMCCNTR_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1101</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elseif PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCCNTR_EL0 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && MDCR_EL2.TPM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      else
        PMCCNTR_EL0 = X[t, 64];
  elseif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elseif EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && MDCR_EL2.TPM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
      PMCCNTR_EL0 = X[t, 64];
  elseif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elseif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
      PMCCNTR_EL0 = X[t, 64];
  elseif PSTATE_EL == EL3 then
    PMCCNTR_EL0 = X[t, 64];
D17.5.3  PMCEID0_EL0, Performance Monitors Common Event Identification register 0

The PMCEID0_EL0 characteristics are:

**Purpose**

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the ranges 0x0000 to 0x001F and 0x4000 to 0x401F.

For more information about the Common events and the use of the PMCEID<n>_EL0 registers see *The PMU event number space and common events* on page D11-5275.

**Configurations**

AArch64 System register PMCEID0_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCEID0[31:0].

AArch64 System register PMCEID0_EL0 bits [63:32] are architecturally mapped to AArch32 System register PMCEID2[31:0].

AArch64 System register PMCEID0_EL0 bits [31:0] are architecturally mapped to External register PMCEID0[31:0].

AArch64 System register PMCEID0_EL0 bits [63:32] are architecturally mapped to External register PMCEID2[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCEID0_EL0 are UNDEFINED.

**Attributes**

PMCEID0_EL0 is a 64-bit register.

**Field descriptions**

IDhi<n>, bit [n+32], for n = 31 to 0

*When FEAT_PMUv3p1 is implemented:*

IDhi[n] corresponds to Common event (0x4000 + n).

For each bit:

0b0  The Common event is not implemented, or not counted.

0b1  The Common event is implemented.
When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

--- Note ---
Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

---

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

--- Note ---
Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n>_EL0 registers of that earlier version of the PMU architecture.

---

**ID<n>, bit [n], for n = 31 to 0**

ID[n] corresponds to Common event n.

For each bit:

- **0b0** The Common event is not implemented, or not counted.
- **0b1** The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

--- Note ---
Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

---

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

--- Note ---
Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n>_EL0 registers of that earlier version of the PMU architecture.

---

**Accessing PMCEID0_EL0**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRS \langle Xt \rangle, \text{PMCEID0_EL0}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b110</td>
</tr>
</tbody>
</table>
elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL3.TPM == '1' then
    AArch64.SystemAccessTrap(EL3, 0x18);
elsif PSTATE.EL == EL3 then
  X[t, 64] = PMCEID0_EL0;
elsif PSTATE.EL == EL2 then
  if Halted() && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  end
elsif PSTATE.EL == EL3 then
  X[t, 64] = PMCEID0_EL0;
elsif PSTATE.EL == EL3 then
  X[t, 64] = PMCEID0_EL0;
D17.5.4  PMCEID1_EL0, Performance Monitors Common Event Identification register 1

The PMCEID1_EL0 characteristics are:

**Purpose**

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the ranges 0x0020 to 0x003F and 0x4020 to 0x403F.

For more information about the Common events and the use of the PMCEID<n>_EL0 registers see The PMU event number space and common events on page D11-5275.

**Configurations**

AArch64 System register PMCEID1_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCEID1[31:0].

AArch64 System register PMCEID1_EL0 bits [63:32] are architecturally mapped to AArch32 System register PMCEID3[31:0].

AArch64 System register PMCEID1_EL0 bits [31:0] are architecturally mapped to External register PMCEID1[31:0].

AArch64 System register PMCEID1_EL0 bits [63:32] are architecturally mapped to External register PMCEID3[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCEID1_EL0 are UNDEFINED.

**Attributes**

PMCEID1_EL0 is a 64-bit register.

**Field descriptions**

When FEAT_PMUv3p1 is implemented:

IDhi[n] corresponds to Common event (0x4020 + n).

For each bit:

- 0b0: The Common event is not implemented, or not counted.
- 0b1: The Common event is implemented.
When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

--- Note ---
Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

--- Note ---
Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n>_EL0 registers of that earlier version of the PMU architecture.

**ID<n>, bit [n], for n = 31 to 0**

ID[n] corresponds to Common event (0x0020 + n).

For each bit:

- **0b0**: The Common event is not implemented, or not counted.
- **0b1**: The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

--- Note ---
Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

--- Note ---
Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n>_EL0 registers of that earlier version of the PMU architecture.

### Accessing PMCEID1_EL0

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, PMCEID1_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b111</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
  if Halted() & HaveEL(EL3) & EDSR.SDD == '1' & boolean IMPLEMENTATION DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    endif
  else
    if EL2Enabled() & HCR_EL2.<E2H,TGE> != '11' & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCEIDn_EL0 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      if HDFGRTR_EL2.PMCEIDn_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      endif
    endif
  endif
endif
```
elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
else
    X[t, 64] = PMCEID1_EL0;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && MDCR_EL3.TPM == '1' then
        UNDEFINED;
elsif EL2Enabled() && EDSCR.SDD == '1' && HDFGRTR_EL2.PMCEIDn_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = PMCEID1_EL0;
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && MDCR_EL3.TPM == '1' then
        UNDEFINED;
elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = PMCEID1_EL0;
elsif PSTATE_EL == EL3 then
    X[t, 64] = PMCEID1_EL0;
### D17.5.5 PMCNTENCLR_EL0, Performance Monitors Count Enable Clear register

The PMCNTENCLR_EL0 characteristics are:

**Purpose**

Disables the Cycle Count Register, PMCCNTR_EL0, and any implemented event counters PMEVCNTR<\(n\)>_EL0. Reading this register shows which counters are enabled.

**Configurations**

AArch64 System register PMCNTENCLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCNTENCLR[31:0].

AArch64 System register PMCNTENCLR_EL0 bits [31:0] are architecturally mapped to External register PMCNTENCLR_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCNTENCLR_EL0 are UNDEFINED.

**Attributes**

PMCNTENCLR_EL0 is a 64-bit register.

#### Field descriptions

**Bits [63:32]**

Reserved, RES0.

**C, bit [31]**

PMCCNTR_EL0 disable bit. Disables the cycle counter register. Possible values are:

- 0b0 When read, means the cycle counter is disabled. When written, has no effect.
- 0b1 When read, means the cycle counter is enabled. When written, disables the cycle counter.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**P<\(n\)>, bit [n], for \(n = 30 \text{ to } 0\)**

Event counter disable bit for PMEVCNTR<\(n\)>_EL0.

If \(N\) is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, \(N\) is the value in MDCR_EL2.HPMN. Otherwise, \(N\) is the value in PMCR_EL0.N.

- 0b0 When read, means that PMEVCNTR<\(n\)>_EL0 is disabled. When written, has no effect.
0b1 When read, means that PMEVCNTR<n>_EL0 is enabled. When written, disables PMEVCNTR<n>_EL0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing PMCNTENCLR_EL0**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <X>, PMCNTENCLR_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b01</td>
<td>0b1000</td>
<td>0b1100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    end
  elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGTR_EL2.PMCNTEN == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
else
  X[t, 64] = PMCNTENCLR_EL0;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = PMCNTENCLR_EL0;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = PMCNTENCLR_EL0;
elsif PSTATE.EL == EL3 then
  X[t, 64] = PMCNTENCLR_EL0;

**MSR PMCNTENCLR_EL0, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif PMUSEREN_EL0.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCNTEN == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    PMCNTENCLR_EL0 = X[t, 64];
  elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCNTEN == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
      PMCNTENCLR_EL0 = X[t, 64];
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
      PMCNTENCLR_EL0 = X[t, 64];
  elsif PSTATE.EL == EL3 then
    PMCNTENCLR_EL0 = X[t, 64];
D17.5.6 PMCNTENSET_EL0, Performance Monitors Count Enable Set register

The PMCNTENSET_EL0 characteristics are:

**Purpose**

Enables the Cycle Count Register, PMCCNTR_EL0, and any implemented event counters PMEVCNTR<\(n\)>_EL0. Reading this register shows which counters are enabled.

**Configurations**

AArch64 System register PMCNTENSET_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCNTENSET[31:0].

AArch64 System register PMCNTENSET_EL0 bits [31:0] are architecturally mapped to External register PMCNTENSET_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCNTENSET_EL0 are UNDEFINED.

**Attributes**

PMCNTENSET_EL0 is a 64-bit register.

**Field descriptions**

Bits [63:32]

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 enable bit. Enables the cycle counter register. Possible values are:

0b0     When read, means the cycle counter is disabled. When written, has no effect.

0b1     When read, means the cycle counter is enabled. When written, enables the cycle counter.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<\(n\)> , bit [n], for n = 30 to 0

Event counter enable bit for PMEVCNTR<\(n\)> EL0.

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR_EL0.N.

0b0     When read, means that PMEVCNTR<\(n\)> EL0 is disabled. When written, has no effect.
When read, means that PMEVCNTR<\text{n}> EL0 event counter is enabled. When written, enables PMEVCNTR<\text{n}> EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing PMCNTENSET_EL0**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <\text{X}>, PMCNTENSET_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    end
  else
    if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCNTEN == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      if EL2Enabled() && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      elseif EL2Enabled() && MDCR_EL3.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      elseif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL3, 0x18);
        end
      else
        X[t, 64] = PMCNTENSET_EL0;
      end
    end
  end
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elseif (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCNTEN == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = PMCNTENSET_EL0;
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elseif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = PMCNTENSET_EL0;
end
```
AArch64 System Register Descriptions
D17.5 Performance Monitors registers

elsif PSTATE.EL == EL3 then
X[t, 64] = PMCNTENSET_EL0;

**MSR PMCNTENSET_EL0, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if Halted() & HaveEL(EL3) & ESDCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
UNDEFINED;
elif PMUSEREN_EL0.EN == '0' then
if EL2Enabled() & HCR_EL2.TGE == '1' then
AArch64.SystemAccessTrap(EL2, 0x18); else
AArch64.SystemAccessTrap(EL1, 0x18); elsif EL2Enabled() & HCR_EL2.<E2H,TGE> != '11' & HaveEL(EL3) || SCR_EL3.FGTEn == '1' & HDFGWTR_EL2.PMCNTEN == '1' then
AArch64.SystemAccessTrap(EL2, 0x18); elsif EL2Enabled() & MDCR_EL2.TPM == '1' then
AArch64.SystemAccessTrap(EL2, 0x18); elsif HaveEL(EL3) & MDCR_EL3.TPM == '1' then
if Halted() & ESDCR.SDD == '1' then
UNDEFINED; else
AArch64.SystemAccessTrap(EL3, 0x18); elsif PSTATE.EL == EL1 then
if Halted() & HaveEL(EL3) & ESDCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
UNDEFINED;
elif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGWTR_EL2.PMCNTEN == '1' then
AArch64.SystemAccessTrap(EL2, 0x18); elsif EL2Enabled() & MDCR_EL2.TPM == '1' then
AArch64.SystemAccessTrap(EL2, 0x18); elsif HaveEL(EL3) & MDCR_EL3.TPM == '1' then
if Halted() & ESDCR.SDD == '1' then
UNDEFINED; else
AArch64.SystemAccessTrap(EL3, 0x18); elsif PSTATE.EL == EL2 then
if Halted() & HaveEL(EL3) & ESDCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
UNDEFINED;
elif HaveEL(EL3) & MDCR_EL3.TPM == '1' then
if Halted() & ESDCR.SDD == '1' then
UNDEFINED; else
AArch64.SystemAccessTrap(EL3, 0x18); else
PMCNTENSET_EL0 = X[t, 64]; elsif PSTATE.EL == EL3 then
PMCNTENSET_EL0 = X[t, 64];
D17.5.7   PMCR_EL0, Performance Monitors Control Register

The PMCR_EL0 characteristics are:

**Purpose**

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

**Configurations**

AArch64 System register PMCR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCR[31:0].

AArch64 System register PMCR_EL0 bits [7:0] are architecturally mapped to External register PMCR_EL0[7:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCR_EL0 are UNDEFINED.

**Attributes**

PMCR_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>32</td>
<td>FZS</td>
</tr>
<tr>
<td>31-24</td>
<td>IMP, IDCODE</td>
</tr>
<tr>
<td>15-11</td>
<td>N</td>
</tr>
<tr>
<td>10-8</td>
<td>LP, LD, DP, X, D, C, P, E</td>
</tr>
<tr>
<td>9-8</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>7</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>6</td>
<td>FZO</td>
</tr>
</tbody>
</table>

**Bits [63:33]**

Reserved, RES0.

**FZS, bit [32]**

*When FEAT_SPEv1p2 is implemented:*

Freeze-on-SPE event. Stop counters when PMBLIMITR_EL1.{PMFZ,E} == {1,1} and PMBSR_EL1.S == 1.

In the description of this field:

- If EL2 is implemented and is using AArch32, PMN is HDCR.HPMN.
- If EL2 is implemented and is using AArch64, PMN is MDCR_EL2.HPMN.
- If EL2 is not implemented, PMN is PMCR_EL0.N.

0b0: Do not freeze on Statistical Profiling Buffer Management event.

0b1: Event counter PMEVCNTR<n>_EL0 does not count following a Statistical Profiling Buffer Management event if n is in the range of affected event counters.

If PMN is not 0, this field affects the operation of event counters in the range [0 .. (PMN-1)].

This field does not affect the operation of other event counters and PMCCNTR_EL0.

The operation of this field applies even when EL2 is disabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset:
  - When AArch32 is supported, this field resets to 0.
When the implementation only supports execution in AArch64 state, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

IMP, bits [31:24]

When FEAT_PMUv3p7 is not implemented:
Implementer code.
If this field is zero, then PMCR_EL0.IDCODE is RES0 and software must use MIDR_EL1 to identify the PE.
Otherwise, this field and PMCR_EL0.IDCODE identify the PMU implementation to software. The implementer codes are allocated by Arm. A non-zero value has the same interpretation as MIDR_EL1.Implementer.
Use of this field is deprecated.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Otherwise:
Reserved, RAZ.

IDCODE, bits [23:16]

When PMCR_EL0.IMP != 0b0000000000:
Identification code. Use of this field is deprecated.
Each implementer must maintain a list of identification codes that are specific to the implementer. A specific implementation is identified by the combination of the implementer code and the identification code.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Otherwise:
Reserved, RES0.

N, bits [15:11]
Indicates the number of event counters implemented. This value is in the range of 0b00000 - 0b11111.
If the value is 0b00000, then only PMCCNTR_EL0 is implemented. If the value is 0b11111, then PMCCNTR_EL0 and 31 event counters are implemented.
When EL2 is implemented and enabled for the current Security state, reads of this field from EL1 and EL0 return the value of MDCR_EL2.HPMN.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Bit [10]
Reserved, RES0.

FZO, bit [9]

When FEAT_PMUv3p7 is implemented:
Freeze-on-overflow. Stop event counters on overflow.
In the description of this field:
• If EL2 is implemented and is using AArch32, PMN is HDCR.HPMN.
• If EL2 is implemented and is using AArch64, PMN is MDCR_EL2.HPMN.
• If EL2 is not implemented, PMN is PMCR_EL0.N.
0b0 Do not freeze on overflow.
0b1 Event counter PMEVCNTR<\text{n}>_EL0 does not count when PMOVSSLR_EL0[(PMN-1):0] is nonzero and n is in the range of affected event counters.

If PMN is not 0, this field affects the operation of event counters in the range [0 .. (PMN-1)]. This field does not affect the operation of other event counters and PMCCNTR_EL0.

The operation of this field applies even when EL2 is disabled in the current Security state.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

\textbf{Otherwise:}
Reserved, RES0.

\textbf{Bit [8]}
Reserved, RES0.

\textbf{LP, bit [7]}

\textbf{When FEAT\_PMUv3p5 is implemented:}
Long event counter enable. Determines when unsigned overflow is recorded by an event counter overflow bit.

In the description of this field:
• If EL2 is implemented and is using AArch32, PMN is HDCCR\_HPMN.
• If EL2 is implemented and is using AArch64, PMN is MDCR\_EL2\_HPMN.
• If EL2 is not implemented, PMN is PMCR\_EL0\_N.

0b0 Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<\text{n}>_EL0[31:0].

0b1 Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<\text{n}>_EL0[63:0].

If PMN is not 0, this field affects the operation of event counters in the range [0 .. (PMN-1)]. This field does not affect the operation of other event counters and PMCCNTR_EL0.

The operation of this field applies even when EL2 is disabled in the current Security state.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

\textbf{Otherwise:}
Reserved, RES0.

\textbf{LC, bit [6]}

\textbf{When AArch32 is supported:}
Long cycle counter enable. Determines when unsigned overflow is recorded by the cycle counter overflow bit.

0b0 Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[31:0].

0b1 Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[63:0].

Arm deprecates use of PMCR\_EL0\_LC = 0.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

\textbf{Otherwise:}
Reserved, RES1.
DP, bit [5]

*When EL3 is implemented or (FEAT_PMUv3p1 is implemented and EL2 is implemented):*

Disable cycle counter when event counting is prohibited.

- **0b0** Cycle counting by PMCCNTR_EL0 is not affected by this mechanism.
- **0b1** Cycle counting by PMCCNTR_EL0 is disabled in prohibited regions and when event counting is frozen:
  - If FEAT_PMUv3p1 is implemented, EL2 is implemented, and MDCR_EL2.HPMD is 1, then cycle counting by PMCCNTR_EL0 is disabled at EL2.
  - If FEAT_PMUv3p7 is implemented, EL3 is implemented and using AArch64, and MDCR_EL3.MPMX is 1, then cycle counting by PMCCNTR_EL0 is disabled at EL3.
  - If FEAT_PMUv3p7 is implemented and event counting is frozen by PMCR_EL0.FZO, then cycle counting by PMCCNTR_EL0 is disabled.
  - If EL3 is implemented, MDCR_EL3.SPME is 0, and either FEAT_PMUv3p7 is not implemented or MDCR_EL3.MPMX is 0, then cycle counting by PMCCNTR_EL0 is disabled at EL3 and in Secure state.

If MDCR_EL2.HPMN is not 0, this is when event counting by event counters in the range [0..(MDCR_EL2.HPMN-1)] is prohibited or frozen.

For more information, see *Prohibiting event and cycle counting on page D11-5256.*

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

X, bit [4]

*When the implementation includes a PMU event export bus:*

Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

- **0b0** Do not export events.
- **0b1** Export events where not prohibited.

This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device, for example to an OPTIONAL trace unit.

No events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RAZ/WI.

D, bit [3]

*When AArch32 is supported:*

Clock divider.

- **0b0** When enabled, PMCCNTR_EL0 counts every clock cycle.
- **0b1** When enabled, PMCCNTR_EL0 counts once every 64 clock cycles.

If PMCR_EL0.LC == 1, this bit is ignored and the cycle counter counts every clock cycle.

Arm deprecates use of PMCR_EL0.D = 1.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**

Reserved, RES0.

**C, bit [2]**

Cycle counter reset. The effects of writing to this bit are:

- 0b0: No action.
- 0b1: Reset PMCCNTR_EL0 to zero.

--- **Note** ---

Resetting PMCCNTR_EL0 does not change the cycle counter overflow bit. If FEAT_PMUv3p5 is implemented, the value of PMCR_EL0.LC is ignored, and bits [63:0] of the cycle counter are reset.

---

Access to this field is WO/RAZ.

**P, bit [1]**

Event counter reset.

In the description of this field:

- If EL2 is implemented and is using AArch32, PMN is HDCR.HPMN.
- If EL2 is implemented and is using AArch64, PMN is MDCR_EL2.HPMN.
- If EL2 is not implemented, PMN is PMCR_EL0.N.

- 0b0: No action.
- 0b1: If n is in the range of affected event counters, resets each event counter PMEVCNTR<\n>_EL0 to zero.

The effects of writing to this bit are:

- If EL2 is implemented and enabled in the current Security state, in EL0 and EL1, if PMN is not 0, a write of 1 to this bit resets event counters in the range [0 .. (PMN-1)].
- If EL2 is disabled in the current Security state, a write of 1 to this bit resets all the event counters.
- In EL2 and EL3, a write of 1 to this bit resets all the event counters.
- This field does not affect the operation of other event counters and PMCCNTR_EL0.

--- **Note** ---

Resetting the event counters does not change the event counter overflow bits. If FEAT_PMUv3p5 is implemented, the values of MDCR_EL2.HLP and PMCR_EL0.LP are ignored, and bits [63:0] of all affected event counters are reset.

---

Access to this field is WO/RAZ.

**E, bit [0]**

Enable.

If EL2 is implemented and is using AArch32, PMN is HDCR.HPMN.

If EL2 is implemented and is using AArch64, PMN is MDCR_EL2.HPMN.

If EL2 is not implemented, PMN is PMCR_EL0.N.

- 0b0: PMCCNTR_EL0 is disabled and event counters PMEVCNTR<\n>_EL0, where n is in the range of affected event counters, are disabled.
- 0b1: PMCCNTR_EL0 and event counters PMEVCNTR<\n>_EL0, where n is in the range of affected event counters, are enabled by PMCNTENSET_EL0.

If PMN is not 0, this field affects the operation of event counters in the range [0 .. (PMN-1)].
This field does not affect the operation of other event counters.
The operation of this field applies even when EL2 is disabled in the current Security state.
The reset behavior of this field is:
• On a Warm reset, this field resets to 0.

Accessing PMCR_EL0

Accesses to this register use the following encodings in the System register encoding space:

\[
\begin{array}{cccc}
op0 & op1 & CRn & CRm & op2 \\
0b11 & 0b011 & 0b1001 & 0b1100 & 0b000
\end{array}
\]

if PSTATE.EL == EL0 then
  if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" \&\& MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elseif PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() \&\& HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  elseif EL2Enabled() \&\& MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() \&\& MDCR_EL2.TPMCR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif HaveEL(EL3) \&\& MDCR_EL3.TPM == '1' then
    if Halted() \&\& EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = PMCR_EL0;
  elseif PSTATE.EL == EL1 then
    if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1'" \&\& MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elseif EL2Enabled() \&\& MDCR_EL2.TPM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() \&\& MDCR_EL2.TPMCR == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) \&\& MDCR_EL3.TPM == '1' then
      if Halted() \&\& EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = PMCR_EL0;
  elseif PSTATE.EL == EL2 then
    if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1'" \&\& MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elseif HaveEL(EL3) \&\& MDCR_EL3.TPM == '1' then
      if Halted() \&\& EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = PMCR_EL0;
elsif PSTATE.EL == EL3 then
    X[t, 64] = PMCR_EL0;

**MSR PMCR_EL0, <xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        end
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && MDCR_EL2.TPMCR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        PMCR_EL0 = X[t, 64];
    end
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elseif EL2Enabled() && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && MDCR_EL2.TPMCR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        PMCR_EL0 = X[t, 64];
    end
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elseif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        PMCR_EL0 = X[t, 64];
    end
elsif PSTATE.EL == EL3 then
    PMCR_EL0 = X[t, 64];
D17.5.8 PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30

The PMEVCNTR<n>_EL0 characteristics are:

**Purpose**
Holds event counter n, which counts events, where n is 0 to 30.

**Configurations**
AArch64 System register PMEVCNTR<n>_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMEVCNTR<n>[31:0].
AArch64 System register PMEVCNTR<n>_EL0 bits [31:0] are architecturally mapped to External register PMEVCNTR<n>_EL0[31:0].
This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMEVCNTR<n>_EL0 are UNDEFINED.

**Attributes**
PMEVCNTR<n>_EL0 is a 64-bit register.

**Field descriptions**

*When FEAT_PMUv3p5 is implemented:*

![Event counter n diagram](image)

**Bits [63:0]**
Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

![Event counter n diagram](image)

**Bits [63:32]**
Reserved, RES0.

**Bits [31:0]**
Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Accessing PMEVCNTR<n>_EL0

PMEVCNTR<n>_EL0 can also be accessed by using PMXEVCNTR_EL0 with PMSELR_EL0.SEL set to the value of <n>.

If FEAT_FGT is implemented and <n> is greater than or equal to the number of accessible event counters, then the behavior of permitted reads and writes of PMEVCNTR<n>_EL0 is as follows:

- If <n> is an unimplemented event counter, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If FEAT_FGT is not implemented and <n> is greater than or equal to the number of accessible event counters, then reads and writes of PMEVCNTR<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if <n> is an UNKNOWN value less-than-or-equal-to the index of the highest accessible event counter.
- If EL2 is implemented and enabled in the current Security state, and <n> is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR_EL0.{ER,EN}.

If EL2 is implemented and enabled in the current Security state, in EL1 and EL0, MDCR_EL2.HPMN identifies the number of accessible event counters. Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see MDCR_EL2.HPMN.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, PMEVCNTR<m>_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b10:m[4:3]</td>
<td>m[2:0]</td>
</tr>
</tbody>
</table>

integer m = UInt(CRm<1:0>:op2<2:0>);

if m >= NUM_PMU_COUNTERS then
  if IsFeatureImplemented(FEAT_FGT) then
    UNDEFINED;
  else
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
  endif
elsif PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED “EL3 trap priority when SDD == ‘1’” && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  else
    PMUSERENR_EL0.<ER,EN> == '00' then
      if EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      else
        AArch64.SystemAccessTrap(EL1, 0x18);
      endif
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMEVCNTRn_EL0 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    endif
  endif
endif
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && m >= AArch64.GetNumEventCountersAccessible() then
AArch64.SystemAccessTrap(EL2, 0x18);
elseif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = PMEVCNTR_EL0[m];
elseif PSTATE_EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
UNDEFINED;
elseif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMEVCNTRn_EL0 == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elseif EL2Enabled() & m >= AArch64.GetNumEventCountersAccessible() then
AArch64.SystemAccessTrap(EL2, 0x18);
elseif EL2Enabled() && MDCR_EL2.TPM == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elseif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = PMEVCNTR_EL0[m];
elseif PSTATE_EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
UNDEFINED;
elseif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = PMEVCNTR_EL0[m];
eelseif PSTATE_EL == EL3 then
X[t, 64] = PMEVCNTR_EL0[m];

**MSR PMEVCNTR<op0>_<op1>_EL0, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b10:m[4:3]</td>
<td>m[2:0]</td>
</tr>
</tbody>
</table>

integer m = UInt(CRm<1:0>:op2<2:0>);
if m >= NUM_PMU_COUNTERS then
if IsFeatureImplemented(FEAT_FGT) then
UNDEFINED;
else
ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elseif PSTATE_EL == EL0 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
UNDEFINED;
elseif PMUSERENR_EL0.EN == '0' then
if EL2Enabled() && HCR_EL2.TGE == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
AArch64.SystemAccessTrap(EL1, 0x18);
elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
HDFGWR_EL2.PMEVCNTRn_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && m == AArch64.GetNumEventCountersAccessible() then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
elsif PMEVCNTR_EL0[m] == X[t, 64];
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD = '1'" && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD = '1'" && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
elsif PSTATE.EL == EL3 then
    PMEVCNTR_EL0[m] = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD = '1'" && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
elsif PSTATE.EL == EL3 then
    PMEVCNTR_EL0[m] = X[t, 64];
D17.5.9 PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n>_EL0 characteristics are:

**Purpose**

Configures event counter n, where n is 0 to 30.

**Configurations**

 AA hire System register PMEVTYPER<n>_EL0 bits [31:0] are architecturally mapped to AA hire System register PMEVTYPER<n>[31:0].

 AA hire System register PMEVTYPER<n>_EL0 bits [63:0] are architecturally mapped to External register PMEVTYPER<n>_EL0[63:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMEVTYPER<n>_EL0 are UNDEFINED.

**Attributes**

PMEVTYPER<n>_EL0 is a 64-bit register.

**Field descriptions**

TC, bits [63:61]

When FEAT_PMUv3_TH is implemented:

Threshold Control. Defines the threshold function. In the description of this field, the value V is the value the event specified by PMEVTYPER<n>_EL0 would increment the counter by on a processor cycle if the threshold function is disabled. Comparisons treat V and PMEVTYPER<n>_EL0.TH as unsigned integer values.

- **0b000**: Not-equal. The counter increments by V on each processor cycle when V is not equal to PMEVTYPER<n>_EL0.TH. If PMEVTYPER<n>_EL0.TH is zero, the threshold function is disabled.
- **0b001**: Not-equal, count. The counter increments by 1 on each processor cycle when V is not equal to PMEVTYPER<n>_EL0.TH.
- **0b010**: Equals. The counter increments by V on each processor cycle when V is equal to PMEVTYPER<n>_EL0.TH.
- **0b011**: Equals, count. The counter increments by 1 on each processor cycle when V is equal to PMEVTYPER<n>_EL0.TH.
- **0b100**: Greater-than-or-equal. The counter increments by V on each processor cycle when V is PMEVTYPER<n>_EL0.TH or more.
- **0b101**: Greater-than-or-equal, count. The counter increments by 1 on each processor cycle when V is PMEVTYPER<n>_EL0.TH or more.
- **0b110**: Less-than. The counter increments by V on each processor cycle when V is less than PMEVTYPER<n>_EL0.TH.
- **0b111**: Less-than, count. The counter increments by 1 on each processor cycle when V is less than PMEVTYPER<n>_EL0.TH.
The reset behavior of this field is:

- On a Warm reset:
  - When AArch32 is supported, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

**Bits [60:44]**
Reserved, RES0.

**TH, bits [43:32]**

*When FEAT_PMUv3_TH is implemented:*
Threshold value. Provides the unsigned value for the threshold function defined by PMEVTYPER<\texttt{n}>_EL0.TC.

If PMEVTYPER<\texttt{n}>_EL0.TC is \texttt{0b000} and PMEVTYPER<\texttt{n}>_EL0.TH is zero, then the threshold function is disabled.

If PMMIR_EL1.THWIDTH is less than 12, then bits PMEVTYPER<\texttt{n}>_EL0.TH[11:PMMIR_EL1.THWIDTH] are RES0. This accounts for the behavior when writing a value greater-than-or-equal-to 2^{PMMIR_EL1.THWIDTH}.

The reset behavior of this field is:

- On a Warm reset:
  - When AArch32 is supported, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

**P, bit [31]**
Privileged filtering bit. Controls counting in EL1.

If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the PMEVTYPER<\texttt{n}>_EL0.NSK bit.

If FEAT_RME is implemented, then counting in Realm EL1 is further controlled by the PMEVTYPER<\texttt{n}>_EL0.RLK bit.

\texttt{0b0} Count events in EL1.
\texttt{0b1} Do not count events in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**U, bit [30]**
User filtering bit. Controls counting in EL0.

If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the PMEVTYPER<\texttt{n}>_EL0.NSU bit.

If FEAT_RME is implemented, then counting in Realm EL0 is further controlled by the PMEVTYPER<\texttt{n}>_EL0.RLU bit.

\texttt{0b0} Count events in EL0.
\texttt{0b1} Do not count events in EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
NSK, bit [29]

When EL3 is implemented:
Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1.
If the value of this bit is equal to the value of the PMEVTYPER<n>_EL0.P bit, events in Non-secure EL1 are counted.
Otherwise, events in Non-secure EL1 are not counted.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:
Non-secure EL0 (Unprivileged) filtering bit. Controls counting in Non-secure EL0.
If the value of this bit is equal to the value of the PMEVTYPER<n>_EL0.U bit, events in Non-secure EL0 are counted.
Otherwise, events in Non-secure EL0 are not counted.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:
EL2 (Hypervisor) filtering bit. Controls counting in EL2.
If Secure EL2 is implemented, and EL3 is implemented, counting in Secure EL2 is further controlled by the PMEVTYPER<n>_EL0.SH bit.
If FEAT_RME is implemented, then counting in Realm EL2 is further controlled by the PMEVTYPER<n>_EL0.RLH bit.
0b0 Do not count events in EL2.
0b1 Count events in EL2.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

M, bit [26]

When EL3 is implemented:
EL3 filtering bit.
If the value of this bit is equal to the value of the PMEVTYPER<n>_EL0.P bit, events in EL3 are counted.
Otherwise, events in EL3 are not counted.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.
MT, bit [25]

When FEAT_MTPMU is implemented or an IMPLEMENTATION DEFINED multi-threaded PMU extension is implemented:

Multithreading.

- 0b0: Count events only on controlling PE.
- 0b1: Count events from any PE with the same affinity at level 1 and above as this PE.

From Armv8.6, the IMPLEMENTATION DEFINED multi-threaded PMU extension is not permitted, meaning if FEAT_MTPMU is not implemented, this field is RES0. See ID_AA64DFR0_EL1.MTPMU.

This field is ignored by the PE and treated as zero when FEAT_MTPMU is implemented and Disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SH, bit [24]

When FEAT_SEL2 is implemented and EL3 is implemented:

Secure EL2 filtering.

If the value of this bit is not equal to the value of the PMEVTYPER<n>_EL0.NSH bit, events in Secure EL2 are counted.

Otherwise, events in Secure EL2 are not counted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

T, bit [23]

When FEAT_TME is implemented:

Transactional state filtering bit. Controls counting in Transactional state.

- 0b0: This bit has no effect on the filtering of events.
- 0b1: Do not count events in Transactional state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLK, bit [22]

When FEAT_RME is implemented:

Realm EL1 (kernel) filtering bit. Controls counting in Realm EL1.

If the value of this bit is equal to the value of the PMEVTYPER<n>_EL0.P bit, events in Realm EL1 are counted.

Otherwise, events in Realm EL1 are not counted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
RLU, bit [21]

*When FEAT_RME is implemented:*

Realm EL0 (unprivileged) filtering bit. Controls counting in Realm EL0.

- If the value of this bit is equal to the value of the PMEVTYPER<\(n\)>_EL0.U bit, events in Realm EL0 are counted.
- Otherwise, events in Realm EL0 are not counted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

RLH, bit [20]

*When FEAT_RME is implemented:*

Realm EL2 filtering bit. Controls counting in Realm EL2.

- If the value of this bit is not equal to the value of the PMEVTYPER<\(n\)>_EL0.NSH bit, events in Realm EL2 are counted.
- Otherwise, events in Realm EL2 are not counted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

Bits [19:16]

Reserved, RES0.

evtCount[15:10], bits [15:10]

*When FEAT_PMUv3p1 is implemented:*

Extension to evtCount[9:0]. For more information, see evtCount[9:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

evtCount[9:0], bits [9:0]

Event to count.

The event number of the event that is counted by event counter PMEVCNTR<\(n\)>_EL0.

The ranges of event numbers allocated to each type of event are shown in Table D11-7 on page D11-5275.

If FEAT_PMUv3p8 is implemented and PMEVTYPER<\(n\)>_EL0.evtCount is programmed to an event that is reserved or not supported by the PE, no events are counted and the value returned by a direct or external read of the PMEVTYPER<\(n\)>_EL0.evtCount field is the value written to the field.

--- Note ---

Arm recommends this behavior for all implementations of FEAT_PMUv3.

Otherwise, if PMEVTYPER<\(n\)>_EL0.evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the value written:

- For the range 0x0000 to 0x003F, no events are counted and the value returned by a direct or external read of the PMEVTYPER<\(n\)>_EL0.evtCount field is the value written to the field.
• If FEAT_PMUv3p1 is implemented, for the range 0x4000 to 0x403F, no events are counted and the value returned by a direct or external read of the PMEVTYPER<\text{n}>_EL0.evtCount field is the value written to the field.

• For other values, it is UNPREDICTABLE what event, if any, is counted and the value returned by a direct or external read of the PMEVTYPER<\text{n}>_EL0.evtCount field is UNKNOWN.

Note

UNPREDICTABLE means the event must not expose privileged information.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMEVTYPER<\text{n}>_EL0

PMEVTYPER<\text{n}>_EL0 can also be accessed by using PMXEVTYPER_EL0 with PMSELR_EL0.SEL set to n.

If FEAT_FGT is implemented and <\text{n}> is greater than or equal to the number of accessible event counters, then the behavior of permitted reads and writes of PMEVTYPER<\text{n}>_EL0 is as follows:

• If <\text{n}> is an unimplemented event counter, the access is UNDEFINED.

• Otherwise, the access is trapped to EL2.

If FEAT_FGT is not implemented and <\text{n}> is greater than or equal to the number of accessible event counters, then reads and writes of PMEVTYPER<\text{n}>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

• Accesses to the register are UNDEFINED.

• Accesses to the register behave as RAZ/WI.

• Accesses to the register execute as a NOP.

• Accesses to the register behave as if <\text{n}> is an UNKNOWN value less-than-or-equal-to the index of the highest accessible event counter.

• If EL2 is implemented and enabled in the current Security state, and <\text{n}> is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR_EL0.EN.

If EL2 is implemented and enabled in the current Security state, in EL1 and EL0, MDCR_EL2.HPMN identifies the number of accessible event counters. Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see MDCR_EL2.HPMN.

Accesses to this register use the following encodings in the System register encoding space:

\textit{MRS <Xt>, PMEVTYPER<m>_EL0}

\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{op0} & \textbf{op1} & \textbf{CRn} & \textbf{CRm} & \textbf{op2} \\
\hline
0b11 & 0b011 & 0b1110 & 0b11:m[4:3] & m[2:0] \\
\hline
\end{tabular}

integer \texttt{m} = \texttt{UInt}(\texttt{CRm<1:0>:op2<2:0>});

if \texttt{m} >= \texttt{NUM_PMU_COUNTERS} then

if IsFeatureImplemented(FEAT_FGT) then

UNDEFINED;
else
  ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  endif
else EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGTR_EL2.PMEVTYPERn_EL0 == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && m >= AArch64.GetNumEventCountersAccessible() then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && HDFGTR_EL2.PMEVTYPERn_EL0 == '1' then
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = PMEVTYPER_EL0[m];
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGTR_EL2.PMEVTYPERn_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && m >= AArch64.GetNumEventCountersAccessible() then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = PMEVTYPER_EL0[m];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = PMEVTYPER_EL0[m];
elsif PSTATE.EL == EL3 then
  X[t, 64] = PMEVTYPER_EL0[m];
integer m = UInt(CRm<1:0>:op2<2:0>);
if m >= NUM_PMU_COUNTERS then
  if IsFeatureImplemented(FEAT_FGT) then
    UNDEFINED;
  else
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE_EL == EL0 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
    UNDEFINED;
elsif PMUSERENR_EL0.EN == '0' then
  if EL2Enabled() & HCR_EL2.TGE == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() & HCR_EL2.<E2H,TGE> != '11' & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGWTR_EL2.PMEVTYPERn_EL0 == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & MDCR_EL2.TPM == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & m == AArch64.GetNumEventCountersAccessible() then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & MDCR_EL3.TPM == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  PMEVTYPER_EL0[m] = X[t, 64];
elsif PSTATE_EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
    UNDEFINED;
elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGWTR_EL2.PMEVTYPERn_EL0 == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & MDCR_EL2.TPM == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & m == AArch64.GetNumEventCountersAccessible() then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & MDCR_EL3.TPM == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  PMEVTYPER_EL0[m] = X[t, 64];
elsif PSTATE_EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
    UNDEFINED;
elsif HaveEL(EL3) & MDCR_EL3.TPM == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  PMEVTYPER_EL0[m] = X[t, 64];
elsif PSTATE_EL == EL3 then
    PMEVTYPE_EL0[m] = X[t, 64];
D17.5.10 PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear register

The PMINTENCLR_EL1 characteristics are:

Purpose

Disables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR_EL0, and the event counters PMEVCNTR<\text{\textless}n\text{\textgreater}>_EL0. Reading the register shows which overflow interrupt requests are enabled.

Configurations

AArch64 System register PMINTENCLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register PMINTENCLR[31:0].

AArch64 System register PMINTENCLR_EL1 bits [31:0] are architecturally mapped to External register PMINTENCLR_EL1[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMINTENCLR_EL1 are UNDEFINED.

Attributes

PMINTENCLR_EL1 is a 64-bit register.

Field descriptions

Bits [63:32] Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 overflow interrupt request disable bit. Possible values are:

0b0 When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

0b1 When read, means the cycle counter overflow interrupt request is enabled. When written, disables the cycle count overflow interrupt request.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<\text{\textless}n\text{\textgreater}>, bit [n], for n = 30 to 0

Event counter overflow interrupt request disable bit for PMEVCNTR<\text{\textless}n\text{\textgreater}>_EL0.
If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR_EL0.N.

0b0 When read, means that the PMEVCNTR<n>_EL0 event counter interrupt request is disabled. When written, has no effect.

0b1 When read, means that the PMEVCNTR<n>_EL0 event counter interrupt request is enabled. When written, disables the PMEVCNTR<n>_EL0 interrupt request.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing PMINTENCLR_EL1

Accesses to this register use the following encodings in the System register encoding space:

#### MRS <xt>, PMINTENCLR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b010</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMINTEN == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    else
      X[t, 64] = PMINTENCLR_EL1;
    end
  elsif PSTATE_EL == EL3 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    end
  else
    X[t, 64] = PMINTENCLR_EL1;
  end
elsif PSTATE_EL == EL3 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      X[t, 64] = PMINTENCLR_EL1;
    end
  else
    X[t, 64] = PMINTENCLR_EL1;
  end
```
MSR PMINTENCLR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMINTEN == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end;
  else
    PMINTENCLR_EL1 = X[t, 64];
  end;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end;
  else
    PMINTENCLR_EL1 = X[t, 64];
  end;
elsif PSTATE.EL == EL3 then
  PMINTENCLR_EL1 = X[t, 64];
D17.5.11 PMINTENSET_EL1, Performance Monitors Interrupt Enable Set register

The PMINTENSET_EL1 characteristics are:

**Purpose**

Enables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR_EL0, and the event counters PMEVCNTR<\text{n}>_EL0. Reading the register shows which overflow interrupt requests are enabled.

**Configurations**

AArch64 System register PMINTENSET_EL1 bits [31:0] are architecturally mapped to AArch32 System register PMINTENSET[31:0].

AArch64 System register PMINTENSET_EL1 bits [31:0] are architecturally mapped to External register PMINTENSET_EL1[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMINTENSET_EL1 are UNDEFINED.

**Attributes**

PMINTENSET_EL1 is a 64-bit register.

**Field descriptions**

PMINTENSET_EL1 is a 64-bit register.

**Bits [63:32]**

Reserved, RES0.

**C, bit [31]**

PMCCNTR_EL0 overflow interrupt request enable bit. Possible values are:

0b0 When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

0b1 When read, means the cycle counter overflow interrupt request is enabled. When written, enables the cycle count overflow interrupt request.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**P<\text{n}>, bit [n], for \text{n} = 30 to 0**

Event counter overflow interrupt request enable bit for PMEVCNTR<\text{n}>_EL0.
If \( N \) is less than 31, then bits \([30:N]\) are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1, \( N \) is the value in MDCR_EL2.HPMN. Otherwise, \( N \) is the value in PMCR_EL0.N.

\[ \text{0b0} \]

When read, means that the PMEVCNTR<\(n\)_EL0 event counter interrupt request is disabled. When written, has no effect.

\[ \text{0b1} \]

When read, means that the PMEVCNTR<\(n\)_EL0 event counter interrupt request is enabled. When written, enables the PMEVCNTR<\(n\>_EL0 interrupt request.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing PMINTENSET_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <\(X_t\), PMINTENSET_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if \( \text{PSTATE.EL == EL0} \) then
  \text{UNDEFINED};
elsif \( \text{PSTATE.EL == EL1} \) then
  if \( \text{Halted()} \) \&\& \( \text{HaveEL(EL3)} \) \&\& \( \text{EDSCR.SDD == '1'} \) \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCHR_EL3.TPM == '1' then
    \text{UNDEFINED};
  else
    \text{AArch64.SystemAccessTrap(EL2, 0x18)};
  end
elsif \( \text{EL2Enabled()} \) \&\& \( \text{MDCHR_EL2.TPM == '1'} \) then
  \text{AArch64.SystemAccessTrap(EL2, 0x18)};
elsif \( \text{HaveEL(EL3)} \) \&\& \( \text{MDCHR_EL3.TPM == '1'} \) then
  if \( \text{Halted()} \) \&\& \( \text{EDSCR.SDD == '1'} \) then
    \text{UNDEFINED};
  else
    X[\(t, 64\)] = PMINTENSET_EL1;
  end
else
  \( X[\(t, 64\)] = \text{PMINTENSET_EL1} \);
elsif \( \text{PSTATE.EL == EL2} \) then
  if \( \text{Halted()} \) \&\& \( \text{HaveEL(EL3)} \) \&\& \( \text{EDSCR.SDD == '1'} \) \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCHR_EL3.TPM == '1' then
    \text{UNDEFINED};
  elsif \( \text{HaveEL(EL3)} \) \&\& \( \text{MDCHR_EL3.TPM == '1'} \) then
    if \( \text{Halted()} \) \&\& \( \text{EDSCR.SDD == '1'} \) then
      \text{UNDEFINED};
    else
      \text{AArch64.SystemAccessTrap(EL3, 0x18)};
    end
  else
    \( X[\(t, 64\)] = \text{PMINTENSET_EL1} \);
  end
elsif \( \text{PSTATE.EL == EL3} \) then
  \( X[\(t, 64\)] = \text{PMINTENSET_EL1} \);
MSR PMINTENSEL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMINTEN == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        PMINTENSEL1 = X[t, 64];
    end
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        PMINTENSEL1 = X[t, 64];
    end
elsif PSTATE_EL == EL3 then
    PMINTENSEL1 = X[t, 64];
D17.5.12 PMMIR_EL1, Performance Monitors Machine Identification Register

The PMMIR_EL1 characteristics are:

**Purpose**

Describes Performance Monitors parameters specific to the implementation to software.

**Configurations**

This register is present only when FEAT_PMUv3p4 is implemented. Otherwise, direct accesses to PMMIR_EL1 are UNDEFINED.

**Attributes**

PMMIR_EL1 is a 64-bit register.

**Field descriptions**

**Bits [63:24]**

Reserved, RES0.

**THWIDTH, bits [23:20]**

PMEVTYPER<\(n\)>_EL0.TH width. Indicates implementation of the FEAT_PMUv3_TH feature, and, if implemented, the size of the PMEVTYPER<\(n\)>_EL0.TH field.

- 0b0000 FEAT_PMUv3_TH is not implemented.
- 0b0001 1 bit. PMEVTYPER<\(n\)>_EL0.TH[11:1] are RES0.
- 0b0010 2 bits. PMEVTYPER<\(n\)>_EL0.TH[11:2] are RES0.
- 0b0011 3 bits. PMEVTYPER<\(n\)>_EL0.TH[11:3] are RES0.
- 0b0100 4 bits. PMEVTYPER<\(n\)>_EL0.TH[11:4] are RES0.
- 0b0101 5 bits. PMEVTYPER<\(n\)>_EL0.TH[11:5] are RES0.
- 0b0110 6 bits. PMEVTYPER<\(n\)>_EL0.TH[11:6] are RES0.
- 0b0111 7 bits. PMEVTYPER<\(n\)>_EL0.TH[11:7] are RES0.
- 0b1000 8 bits. PMEVTYPER<\(n\)>_EL0.TH[11:8] are RES0.
- 0b1001 9 bits. PMEVTYPER<\(n\)>_EL0.TH[11:9] are RES0.
- 0b1010 10 bits. PMEVTYPER<\(n\)>_EL0.TH[11:10] are RES0.
- 0b1011 11 bits. PMEVTYPER<\(n\)>_EL0.TH[11] is RES0.
- 0b1100 12 bits.

All other values are reserved.

If FEAT_PMUv3_TH is not implemented, this field is zero. Otherwise, the largest value that can be written to PMEVTYPER<\(n\)>_EL0.TH is 2^(PMMIR_EL1.THWIDTH) minus one.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.
BUS_WIDTH, bits [19:16]

Bus width. Indicates the number of bytes each BUS_ACCESS event relates to. Encoded as Log2(number of bytes), plus one.

- 0b0000: The information is not available.
- 0b0011: Four bytes.
- 0b0100: 8 bytes.
- 0b0101: 16 bytes.
- 0b0110: 32 bytes.
- 0b0111: 64 bytes.
- 0b1000: 128 bytes.
- 0b1001: 256 bytes.
- 0b1010: 512 bytes.
- 0b1011: 1024 bytes.
- 0b1100: 2048 bytes.

All other values are reserved.

Each transfer is up to this number of bytes. An access might be smaller than the bus width.

When this field is nonzero, each access counted by BUS_ACCESS is at most BUS_WIDTH bytes. An implementation might treat a wide bus as multiple narrower buses, such that a wide access on the bus increments the BUS_ACCESS counter by more than one.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

BUS_SLOTS, bits [15:8]

Bus count. The largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle.

When this field is nonzero, the largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle is BUS_SLOTS.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

SLOTS, bits [7:0]

Operation width. The largest value by which the STALL_SLOT event might increment in a single cycle. If the STALL SLOT event is not implemented, this field might read as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMMIR_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, PMMIR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b101</td>
<td>0b1110</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1' && MDCR_EL3.TPM == '1' then
  UNDEFINED;
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMMIR_EL1 == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = PMMIR_EL1;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority"
  when SDD == '1' && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = PMMIR_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = PMMIR_EL1;
D17.5.13 PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear Register

The PMOVSCLR_EL0 characteristics are:

**Purpose**

Contains the state of the overflow bit for the Cycle Count Register, PMCCNTR_EL0, and each of the implemented event counters PMEVCNTR-<n>_EL0. Writing to this register clears these bits.

**Configurations**

AArch64 System register PMOVSCLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMOVSR[31:0].

AArch64 System register PMOVSCLR_EL0 bits [31:0] are architecturally mapped to External register PMOVSCLR_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMOVSCLR_EL0 are UNDEFINED.

**Attributes**

PMOVSCLR_EL0 is a 64-bit register.

**Field descriptions**

**Bits [63:32]**

Reserved, RES0.

**C, bit [31]**

Cycle counter overflow clear bit.

0b0 When read, means the cycle counter has not overflowed since this bit was last cleared. When written, has no effect.

0b1 When read, means the cycle counter has overflowed since this bit was last cleared. When written, clears the cycle counter overflow bit to 0.

**PMCR_EL0.LC** controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**P<n>, bit [n], for n = 30 to 0**

Event counter overflow clear bit for PMEVCNTR-<n>_EL0.
If \( N \) is less than 31, then bits [30:0] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, \( N \) is the value in MDCR_EL2.HPMN. Otherwise, \( N \) is the value in PMCR_EL0.N.

- 0b0  When read, means that PMEVCNTR\( \langle n \rangle \_EL0 \) has not overflowed since this bit was last cleared. When written, has no effect.
- 0b1  When read, means that PMEVCNTR\( \langle n \rangle \_EL0 \) has overflowed since this bit was last cleared. When written, clears the PMEVCNTR\( \langle n \rangle \_EL0 \) overflow bit to 0.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR\( \langle n \rangle \_EL0[31:0] \) or unsigned overflow of PMEVCNTR\( \langle n \rangle \_EL0[63:0] \).

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing PMOVSCLR_EL0

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, PMOVSCLR_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b011</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' \\&\& MDSCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        end if
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMOV = '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) && MDSCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = PMOVSCLR_EL0;
        end if
    elsif PSTATE.EL == EL1 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' \\&\& MDSCR_EL3.TPM == '1' then
            UNDEFINED;
        elsif EL2Enabled() && MDSCR_EL2.TPM == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) && MDSCR_EL3.TPM == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end if
        else
            X[t, 64] = PMOVSCLR_EL0;
        end if
    elsif PSTATE.EL == EL2 then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMOVSCLR_EL0;
    end if
```
if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCR_EL3.TPM == '1' then
  UNDEFINED;
elsif HaveEL(EL3) \&\& MDCR_EL3.TPM == '1' then
  if Halted() \&\& EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = PMOVSLR_EL0;
elsif PSTATE.EL == EL3 then
  X[t, 64] = PMOVSLR_EL0;

**MSR PMOVSLR_EL0, <xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCR_EL3.TPM == '1' then
    UNDEFINED;
elsif PMUSERENR_EL0.EN == '0' then
  if EL2Enabled() \&\& HCR_EL2.TGE == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL0, 0x18);
elsif EL2Enabled() \&\& HCR_EL2.<E2H,TGE> != '11' \&\& (!HaveEL(EL3) \&\& SCRE.L3.FGTEn == '1') \&\& HDFGWTR_EL2.PMOV == '1' then
  if EL2Enabled() \&\& MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL1, 0x18);
elseif PSTATE.EL == EL1 then
  if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCR_EL3.TPM == '1' then
    UNDEFINED;
elsif EL2Enabled() \&\& HCR_EL2.<E2H,TGE> != '11' \&\& (!HaveEL(EL3) \&\& SCRE.L3.FGTEn == '1') \&\& HDFGWTR_EL2.PMOV == '1' then
  if EL2Enabled() \&\& MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif PSTATE.EL == EL2 then
  if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCR_EL3.TPM == '1' then
    UNDEFINED;
elsif EL2Enabled() \&\& HCR_EL2.<E2H,TGE> != '11' \&\& (!HaveEL(EL3) \&\& SCRE.L3.FGTEn == '1') \&\& HDFGWTR_EL2.PMOV == '1' then
  if EL2Enabled() \&\& MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif PSTATE.EL == EL3 then
  if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& MDCR_EL3.TPM == '1' then
    UNDEFINED;
elsif EL2Enabled() \&\& HCR_EL2.<E2H,TGE> != '11' \&\& (!HaveEL(EL3) \&\& SCRE.L3.FGTEn == '1') \&\& HDFGWTR_EL2.PMOV == '1' then
  if EL2Enabled() \&\& MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL1, 0x18);
else
    PMOVCLR_EL0 = X[t, 64];
elsif PSTATE.EL == EL3 then
    PMOVCLR_EL0 = X[t, 64];
D17.5.14 PMOVSET_EL0, Performance Monitors Overflow Flag Status Set register

The PMOVSET_EL0 characteristics are:

**Purpose**

Sets the state of the overflow bit for the Cycle Count Register, PMCCNTR_EL0, and each of the implemented event counters PMEVCNTR<n>_EL0.

**Configurations**

AArch64 System register PMOVSET_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMOVSET[31:0].

AArch64 System register PMOVSET_EL0 bits [31:0] are architecturally mapped to External register PMOVSET_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMOVSET_EL0 are UNDEFINED.

**Attributes**

PMOVSET_EL0 is a 64-bit register.

**Field descriptions**

```
+-------------------+-------------------+
| 63                | 32                |
| RES0              | C                 |
| P30               | P29               |
| P28               | P27               |
| P26               | P25               |
| P24               | P23               |
| P22               | P21               |
| P19               | P18               |
| P14               | P13               |
| P17               | P16               |
| P11               | P10               |
| P8                | P7                |
| P5                | P4                |
| P2                | P1                |
| P0                |                  |
|                  |                  |
```

**Bits [63:32]**

Reserved, RES0.

**C, bit [31]**

Cycle counter overflow set bit.

0b0 When read, means the cycle counter has not overflowed since this bit was last cleared. When written, has no effect.

0b1 When read, means the cycle counter has overflowed since this bit was last cleared. When written, sets the cycle counter overflow bit to 1.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**P<n>, bit [n], for n = 30 to 0**

Event counter overflow set bit for PMEVCNTR<n>_EL0.
If \( N \) is less than 31, then bits [30:31] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, \( N \) is the value in MDCR_EL2.HPMN. Otherwise, \( N \) is the value in PMCR_EL0.HPMN.

- **0b0** When read, means that PMEVCNTR\(<n>_EL0 has not overflowed since this bit was last cleared. When written, has no effect.
- **0b1** When read, means that PMEVCNTR\(<n>_EL0 has overflowed since this bit was last cleared. When written, sets the PMEVCNTR\(<n>_EL0 overflow bit to 1.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR\(<n>_EL0[31:0] or unsigned overflow of PMEVCNTR\(<n>_EL0[63:0].

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing PMOVSSET_EL0

Accesses to this register use the following encodings in the System register encoding space:

### MRS <Xt>, PMOVSSET_EL0

<table>
<thead>
<tr>
<th>( \text{op0} )</th>
<th>( \text{op1} )</th>
<th>( \text{CRn} )</th>
<th>( \text{CRm} )</th>
<th>( \text{op2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b011</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD \( = \) '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD \( = \) '1' \&\& MDCR_EL3.TPM \( = \) '1' then
    UNDEFINED;
  elsif PMUSERENR_EL0.EN \( = \) '0' then
    if EL2Enabled() \&\& HCR_EL2.TGE \( = \) '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    end
  else
    if EL2Enabled() \&\& HCR_EL2.<E2H,TGE> \( = \) '11' \&\& edge(EL3) \&\& SCR_EL3.FGTEn \( = \) '1') \&\& HDFGRT_EL2.PMOVS \( = \) '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() \&\& MDCR_EL2.TPM \( = \) '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) \&\& MDCR_EL3.TPM \( = \) '1' then
      if Halted() \&\& EDSCR.SDD \( = \) '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
else
  X[t, 64] = PMOVSSET_EL0;
elsif PSTATE.EL == EL1 then
  if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD \( = \) '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD \( = \) '1' \&\& MDCR_EL3.TPM \( = \) '1' then
    UNDEFINED;
  elsif EL2Enabled() \&\& ((HaveEL(EL3) \&\& SCR_EL3.FGTEn \( = \) '1') \&\& HDFGRT_EL2.PMOVS \( = \) '1') then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() \&\& MDCR_EL2.TPM \( = \) '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) \&\& MDCR_EL3.TPM \( = \) '1' then
    if Halted() \&\& EDSCR.SDD \( = \) '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
else
  X[t, 64] = PMOVSSET_EL0;
elsif PSTATE.EL == EL2 then

D17-6796

Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential

ARM DDI 04871.a

ID081822
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then UNDEFINED;
elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
else
  X[t, 64] = PMOVSSET_EL0;
endif PSTATE.EL == EL3 then
  X[t, 64] = PMOVSSET_EL0;
end

MSR PMOVSSET_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then UNDEFINED;
  elsif PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      if EL2Enabled() && HDFGWTR_EL2.PMOVS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      else
        PMOVSSET_EL0 = X[t, 64];
      end
    end
  elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    PMOVSSET_EL0 = X[t, 64];
  end
endif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then UNDEFINED;
  elsif EL2Enabled() && HDFGWTR_EL2.PMOVS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if EL2Enabled() && HDFGWTR_EL2.PMOVS == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      PMOVSSET_EL0 = X[t, 64];
    end
  end
endif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
end
else
  PMOVSET_EL0 = X[t, 64];
elsif PSTATE.EL == EL3 then
  PMOVSET_EL0 = X[t, 64];
D17.5.15 PMSEL_EL0, Performance Monitors Event Counter Selection Register

The PMSEL_EL0 characteristics are:

**Purpose**

Selects the current event counter PMEVCNTR\(<n>\)_EL0 or the cycle counter, CCNT.

PMSEL_EL0 is used in conjunction with PMXEVTYPER_EL0 to determine the event that increments a selected event counter, and the modes and states in which the selected counter increments.

It is also used in conjunction with PMXEVCNTR_EL0, to determine the value of a selected event counter.

**Configurations**

AArch64 System register PMSEL_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMSEL[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMSEL_EL0 are UNDEFINED.

**Attributes**

PMSEL_EL0 is a 64-bit register.

**Field descriptions**

![Field descriptions diagram]

**Bits [63:5]**

Reserved, RES0.

**SEL, bits [4:0]**

Selects event counter, PMEVCNTR\(<n>\)_EL0, where n is the value held in this field. This value identifies which event counter is accessed when a subsequent access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 occurs.

This field can take any value from 0 (0b00000) to (PMCR.N)-1, or 31 (0b11111).

When PMSEL_EL0.SEL is 0b11111, it selects the cycle counter and:

- A read of the PMXEVTYPER_EL0 returns the value of PMCCFILTR_EL0.
- A write of the PMXEVTYPER_EL0 writes to PMCCFILTR_EL0.
- A read or write of PMXEVCNTR_EL0 has CONSTRAINED UNPREDICTABLE effects. For more information, see PMXEVCNTR_EL0.

For more information about the results of accesses to the event counters, see PMXEVTYPER_EL0 and PMXEVCNTR_EL0.

For more information about the number of counters accessible at each Exception level, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing PMSEL_EL0**

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
    UNDEFINED;
  elsif PMUSERENR_EL0.<ER,EN> == '00' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
      elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '1' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMSELR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL3, 0x18);
      else
        X[t, 64] = PMSELR_EL0;
    elsif PSTATE.EL == EL1 then
      if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
        UNDEFINED;
      elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMSELR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL3, 0x18);
      else
        X[t, 64] = PMSELR_EL0;
    elsif PSTATE.EL == EL2 then
      if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
        UNDEFINED;
      elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL3, 0x18);
      else
        X[t, 64] = PMSELR_EL0;
    elsif PSTATE.EL == EL3 then
      X[t, 64] = PMSELR_EL0;
 MSR PMSEL_EL0, <Xt>  

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif PMUSERENR_EL0.<ER,EN> == '00' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    end
  elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWR_EL2.PMSEL_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    PMSEL_EL0 = X[t, 64];
  end
else
  if PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elsif EL2Enabled() && (((HaveEL(EL3)) || SCR_EL3.FGTEn == '1') && HDFGWR_EL2.PMSEL_EL0 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    else
      PMSEL_EL0 = X[t, 64];
    end
  elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      end
    end
  else
    PMSEL_EL0 = X[t, 64];
  end
else
  if PSTATE_EL == EL3 then
    PMSEL_EL0 = X[t, 64];
  end
D17.5.16 PMSWINC_EL0, Performance Monitors Software Increment register

The PMSWINC_EL0 characteristics are:

**Purpose**

Increments a counter that is configured to count the Software increment event, event 0x00. For more information, see SW_INCR on page D11-5307.

**Configurations**

AArch64 System register PMSWINC_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMSWINC[31:0].

AArch64 System register PMSWINC_EL0 bits [31:0] are architecturally mapped to External register PMSWINC_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMSWINC_EL0 are UNDEFINED.

**Attributes**

PMSWINC_EL0 is a 64-bit register.

**Field descriptions**

**Bits [63:31]**

Reserved, RES0.

**P<n>, bit [n], for n = 30 to 0**

Event counter software increment bit for PMEVCNTR<n>_EL0.

If N is less than 31, then bits [30:N] are WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, N is the value in MDCR_EL2.HPMN. Otherwise, N is the value in PMCR_EL0.N.

0b0 No action. The write to this bit is ignored.

0b1 If PMEVCNTR<n>_EL0 is enabled and configured to count the software increment event, increments PMEVCNTR<n>_EL0 by 1. If PMEVCNTR<n>_EL0 is disabled, or not configured to count the software increment event, the write to this bit is ignored.

**Accessing PMSWINC_EL0**

Accesses to this register use the following encodings in the System register encoding space:


**MSR PMSWINC_EL0, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif PMUSERENR_EL0.<SW,EN> == '00' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    end if;
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMSWINC_EL0 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL2, 0x18);
    end if;
    elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end if;
    else
      PMSWINC_EL0 = X[t, 64];
    elsif PSTATE_EL == EL1 then
      if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
        UNDEFINED;
      elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMSWINC_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      else
        AArch64.SystemAccessTrap(EL2, 0x18);
      end if;
      elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL3, 0x18);
        end if;
      else
        PMSWINC_EL0 = X[t, 64];
      elsif PSTATE_EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
          UNDEFINED;
        elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
          if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
          else
            AArch64.SystemAccessTrap(EL3, 0x18);
          end if;
        else
          PMSWINC_EL0 = X[t, 64];
        end if;
      elsif PSTATE_EL == EL3 then
        PMSWINC_EL0 = X[t, 64];
      end if;
    else
      PMSWINC_EL0 = X[t, 64];
  end if;
else
  PMSWINC_EL0 = X[t, 64];
end if;
D17.5.17  PMUSERENR_EL0, Performance Monitors User Enable Register

The PMUSERENR_EL0 characteristics are:

**Purpose**

Enables or disables EL0 access to the Performance Monitors.

**Configurations**

AArch64 System register PMUSERENR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMUSERENR[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMUSERENR_EL0 are **UNDEFINED**.

**Attributes**

PMUSERENR_EL0 is a 64-bit register.

**Field descriptions**

| Bits [63:4] | Reserved, RES0. |
| Bits [3:0] | ER, bit [3] |
| Event counter Read. Traps EL0 access to event counters to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. |
| In AArch64 state, trapped accesses are reported using EC syndrome value 0x18. |
| In AArch32 state, trapped accesses are reported using EC syndrome value 0x03. |
| **0b0** EL0 using AArch64: EL0 reads of the PMXEVCNTR_EL0 and PMEVCNTR<0>_EL0, and EL0 read/write accesses to the PMSELR_EL0, are trapped if PMUSERENR_EL0.EN is also 0. |
| EL0 using AArch32: EL0 reads of the PMXEVCNTR and PMEVCNTR<0>, and EL0 read/write accesses to the PMSELR, are trapped if PMUSERENR_EL0.EN is also 0. |
| **0b1** Overrides PMUSERENR_EL0.EN and enables: |
| * RO access to PMXEVCNTR_EL0 and PMEVCNTR<0>_EL0 at EL0. |
| * RW access to PMSELR_EL0 at EL0. |
| * RW access to PMSELR at EL0. |

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

| Cycle counter Read. Traps EL0 access to cycle counter reads to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. |
| In AArch64 state, trapped accesses are reported using EC syndrome value 0x18. |
In AArch32 state, trapped MRC accesses are reported using EC syndrome value 0x03, trapped MRRC accesses are reported using EC syndrome value 0x04.

**0b0** EL0 using AArch64: EL0 read accesses to the PMCCNTR_EL0 are trapped if PMUSERENR_EL0.EN is also 0.
EL0 using AArch32: EL0 read accesses to the PMCCNTR are trapped if PMUSERENR_EL0.EN is also 0.

**0b1** Overrides PMUSERENR_EL0.EN and enables access to:
- PMCCNTR_EL0 at EL0.
- PMCCNTR at EL0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**SW, bit [1]**

Traps Software Increment writes to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1.

In AArch64 state, trapped accesses are reported using EC syndrome value 0x18.
In AArch32 state, trapped accesses are reported using EC syndrome value 0x03.

**0b0** EL0 using AArch64: EL0 writes to the PMSWINC_EL0 are trapped if PMUSERENR_EL0.EN is also 0.
EL0 using AArch32: EL0 writes to the PMSWINC are trapped if PMUSERENR_EL0.EN is also 0.

**0b1** Overrides PMUSERENR_EL0.EN and enables access to:
- PMSWINC_EL0 at EL0.
- PMSWINC at EL0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**EN, bit [0]**

Traps EL0 accesses to the Performance Monitor registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states as follows:

- In AArch64 state, MRS or MSR accesses to the following registers are reported using EC syndrome value 0x18:
  - PMCR_EL0, PMOVSCR_EL0, PMSELR_EL0, PMCEID0_EL0, PMCEID1_EL0, PMCCNTR_EL0, PMXEVTYPER_EL0, PMXEVCTRL_EL0, PMCNTSET_EL0, PMCNTENCLR_EL0, PMOVSSET_EL0, PMEVCNTR<n>_<n>_<n>_EL0, PMEVTYPER<n>_<n>_EL0, PMCCFILTR_EL0, PMSWINC_EL0.
  - If FEAT_PMUv3p4 is implemented, PMMIR_EL1.
- In AArch32 state, MRC or MCR accesses to the following registers are reported using EC syndrome value 0x03:
  - PMCR, PMOVSR, PMSELR, PMCEID0, PMCEID1, PMCCNTR, PMXEVTYPER, PMXEVCTRL, PMCNTSET, PMCNTENCLR, PMOVSSET, PMEVCNTR<n>, PMEVTYPER<n>, PMCCFILTR, PMSWINC.
  - If FEAT_PMUv3p4 is implemented, PMMIR.
  - If FEAT_PMUv3p1 is implemented, in AArch32 state, PMCEID2, and PMCEID3.
- In AArch32 state, MRRC or MCRR accesses to PMCCNTR are reported using EC syndrome value 0x04.

**0b0** While at EL0, accesses to the specified registers at EL0 are trapped, unless overridden by one of PMUSERENR_EL0.{ER, CR, SW}. 

While at EL0, software can access all of the specified registers. The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing PMUSERENR_EL0

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, PMUSERENR_EL0**

```
<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b000</td>
</tr>
</tbody>
</table>
```

```c
if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && HDFGRTR_EL2.PMUSERENR_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && HDFGRTR_EL2.PMUSERENR_EL0 == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elsif EL2Enabled() && HDFGRTR_EL2.PMUSERENR_EL0 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && HDFGRTR_EL2.PMUSERENR_EL0 == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    else
      X[t, 64] = PMUSERENR_EL0;
    end
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elsif EL2Enabled() && HDFGRTR_EL2.PMUSERENR_EL0 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && HDFGRTR_EL2.PMUSERENR_EL0 == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    else
      X[t, 64] = PMUSERENR_EL0;
    end
  elsif PSTATE.EL == EL3 then
    X[t, 64] = PMUSERENR_EL0;
  else
    X[t, 64] = PMUSERENR_EL0;
  end
```

```c
if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && HDFGRTR_EL2.PMUSERENR_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && HDFGRTR_EL2.PMUSERENR_EL0 == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elsif EL2Enabled() && HDFGRTR_EL2.PMUSERENR_EL0 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && HDFGRTR_EL2.PMUSERENR_EL0 == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    else
      X[t, 64] = PMUSERENR_EL0;
    end
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elsif EL2Enabled() && HDFGRTR_EL2.PMUSERENR_EL0 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && HDFGRTR_EL2.PMUSERENR_EL0 == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    else
      X[t, 64] = PMUSERENR_EL0;
    end
  elsif PSTATE.EL == EL3 then
    X[t, 64] = PMUSERENR_EL0;
  else
    X[t, 64] = PMUSERENR_EL0;
  end
```
MSR PMUSERENR_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMUSERENR_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
else
    PMUSERENR_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
else
    PMUSERENR_EL0 = X[t, 64];
elsif PSTATE.EL == EL3 then
    PMUSERENR_EL0 = X[t, 64];
D17.5.18 PMXEVCNTR_EL0, Performance Monitors Selected Event Count Register

The PMXEVCNTR_EL0 characteristics are:

**Purpose**
Reads or writes the value of the selected event counter, PMEVCNTR<n>_EL0. PMSELR_EL0.SEL determines which event counter is selected.

**Configurations**
AArch64 System register PMXEVCNTR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMXEVCTR[31:0].
This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMXEVCNTR_EL0 are UNDEFINED.

**Attributes**
PMXEVCNTR_EL0 is a 64-bit register.

**Field descriptions**

*When FEAT_PMUv3p5 is implemented:*

\[
\begin{array}{c}
63 \\
32 \\
31 \\
0 \\
\end{array}
\]

PMEVCNTR<n>, bits [63:0]
Value of the selected event counter, PMEVCNTR<n>_EL0, where n is the value stored in PMSELR_EL0.SEL.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

\[
\begin{array}{c}
63 \\
32 \\
31 \\
0 \\
\end{array}
\]

Bits [63:32]
Reserved, RES0.

PMEVCNTR<n>, bits [31:0]
Value of the selected event counter, PMEVCNTR<n>_EL0, where n is the value stored in PMSELR_EL0.SEL.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Accessing PMXEVCNTR_EL0

If `FEAT_FGT` is implemented and `PMSELR_EL0.SEL` is greater than or equal to the number of accessible event counters, then the behavior of permitted reads and writes of `PMXEVCNTR_EL0` is as follows:

- If `PMSELR_EL0.SEL` selects an unimplemented event counter, the access is **UNDEFINED**.
- Otherwise, the access is trapped to EL2.

If `FEAT_FGT` is not implemented and `PMSELR_EL0.SEL` is greater than or equal to the number of accessible event counters, then reads and writes of `PMXEVCNTR_EL0` are **CONSTRAINED UNPREDICTABLE**, and the following behaviors are permitted:

- Accesses to the register are **UNDEFINED**.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOOP.
- Accesses to the register behave as if `PMSELR_EL0.SEL` has an **UNKNOWN** value less than the number of counters accessible at the current Exception level and Security state.
- If EL2 is implemented and enabled in the current Security state, and `PMSELR_EL0.SEL` is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by `PMUSERENR_EL0.{ER,EN}`.

If EL2 is implemented and enabled in the current Security state, in EL1 and EL0, `MDCR_EL2.HPMN` identifies the number of accessible event counters. Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see `MDCR_EL2.HPMN`.

Accesses to this register use the following encodings in the System register encoding space:

### MRS <xt>, PMXEVCNTR_EL0

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1101</td>
<td>0b010</td>
</tr>
</tbody>
</table>

```plaintext
if UInt(PMSELR_EL0.SEL) >= NUM_PMU_COUNTERS then
    if IsFeatureImplemented(FEAT_FGT) then
        UNDEFINED;
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    endif
elsiif PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' " && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsiif PMUSERENR_EL0.<ER,EN> == '00' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        endsiif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFQTR_EL2.PMVECNTNn_EL0 == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsiif EL2Enabled() && MDCR_EL2.TPM == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsiif EL2Enabled() && UInt(PMSELR_EL0.SEL) >= AArch64.GetNumEventCountersAccessible() then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsiif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
```
if Halted() && EDSCR.SDD == '1' then
  UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = PMEVCNTR_EL0[UInt(PMSELR_EL0.SEL)];
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elseif EL2Enabled() && PMEVCNTR_EL0[UInt(PMSELR_EL0.SEL)] == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HDFGWTR_EL2.PMEVCNTRn_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = PMEVCNTR_EL0[UInt(PMSELR_EL0.SEL)];
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elseif HaveEL(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      else
        X[t, 64] = PMEVCNTR_EL0[UInt(PMSELR_EL0.SEL)];
  elsif PSTATE.EL == EL3 then
    X[t, 64] = PMEVCNTR_EL0[UInt(PMSELR_EL0.SEL)];
else
  X[t, 64] = PMEVCNTR_EL0[UInt(PMSELR_EL0.SEL)];

MSR PMXEVCNTR_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1101</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if UInt(PMSELR_EL0.SEL) >= NUM_PMU_COUNTERS then
  if IsFeatureImplemented(FEAT_FGT) then
    UNDEFINED;
  else
    ConstrainImplementedProcedure(Unpredictable_PMU_EVENT_COUNTER);
else
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elseif PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && HDFGWTR_EL2.PMEVCNTRn_EL0 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HDFGWTR_EL2.PMEVCNTRn_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HDFGWTR_EL2.PMEVCNTRn_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HDFGWTR_EL2.PMEVCNTRn_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & MDCR_EL3.TPM == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
elsif PSTATE_EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
    UNDEFINED;
elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGWTR_EL2.PMEVCNTRn_EL0 == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & MDCR_EL2.TPM == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & UInt(PMSELR_EL0.SEL) >= AArch64.GetNumEventCountersAccessible() then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & MDCR_EL3.TPM == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
elsif PSTATE_EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
    UNDEFINED;
elsif HaveEL(EL3) & MDCR_EL3.TPM == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
elsif PSTATE_EL == EL3 then
  PMEVCNTR_EL0[UInt(PMSELR_EL0.SEL)] = X[t, 64];
elsif PSTATE_EL == EL3 then
  PMEVCNTR_EL0[UInt(PMSELR_EL0.SEL)] = X[t, 64];
else
  PMEVCNTR_EL0[UInt(PMSELR_EL0.SEL)] = X[t, 64];
elsif PSTATE_EL == EL3 then
  PMEVCNTR_EL0[UInt(PMSELR_EL0.SEL)] = X[t, 64];
D17.5.19 PMXEVTYPER_EL0, Performance Monitors Selected Event Type Register

The PMXEVTYPER_EL0 characteristics are:

**Purpose**
When PMSELR_EL0.SEL selects an event counter, this accesses a PMEVTYPER<n>_EL0 register. When PMSELR_EL0.SEL selects the cycle counter, this accesses PMCCFILTR_EL0.

**Configurations**
AArch64 System register PMXEVTYPER_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMXEVTYPER[31:0].
This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMXEVTYPER_EL0 are UNDEFINED.

**Attributes**
PMXEVTYPER_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Event type register or PMCCFILTR_EL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>Event type register or PMCCFILTR_EL0</td>
</tr>
<tr>
<td>31</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

**Bits [63:0]**
When PMSELR_EL0.SEL == 31, this register accesses PMCCFILTR_EL0.
Otherwise, this register accesses PMEVTYPER<n>_EL0 where n is the value in PMSELR_EL0.SEL.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing PMXEVTYPER_EL0**
If FEAT_FGT is implemented, and PMSELR_EL0.SEL is not 31 and is greater than or equal to the number of accessible event counters, then the behavior of permitted reads and writes of PMXEVTYPER_EL0 is as follows:
- If PMSELR_EL0.SEL selects an unimplemented event counter, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If FEAT_FGT is not implemented, and PMSELR_EL0.SEL is not 31 and is greater than or equal to the number of accessible event counters, then reads and writes of PMXEVTYPER_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:
- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if PMSELR_EL0.SEL has an UNKNOWN value less than the number of event counters accessible at the current Exception level and Security state.
- Accesses to the register behave as if PMSELR_EL0.SEL is 31.
• If EL2 is implemented and enabled in the current Security state, PMSELR_EL0 is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR_EL0.EN.

If EL2 is implemented and enabled in the current Security state, in EL1 and EL0, MDCR_EL2.HPMN identifies the number of accessible event counters. Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see MDCR_EL2.HPMN.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, PMXEVTPER_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1101</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if UInt(PMSELR_EL0.SEL) != 31 && UInt(PMSELR_EL0.SEL) >= NUM_PMU_COUNTERS then
if IsFeatureImplemented(FeaT_FGT) then
  UNDEFINED;
else
  ConstrainImplementedProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elif PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    endif
  endelse
else
  AArch64.GetNumEventCountersAccessible() then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
endif
X[t, 64] = PMCCFILTR_EL0;
else
  X[t, 64] = PMEVTYPER_EL0[UInt(PMSELR_EL0.SEL)];
endif
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMEVTYPERn_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  endif
else
  AArch64.GetNumEventCountersAccessible() then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
endif
X[t, 64] = PMEVTYPER_EL0[UInt(PMSELR_EL0.SEL)];
UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  elsif UInt(PMSEL_EL0.SEL) == 31 then
    X[t, 64] = PMCCFILTR_EL0;
  else
    X[t, 64] = PMEVTYPER_EL0(UInt(PMSEL_EL0.SEL));
  endif
  if PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
      UNDEFINED;
  elsif HaveEL(EL3) & MDCR_EL3.TPM == '1'
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
  elsifUInt(PMSEL_EL0.SEL) == 31 then
    X[t, 64] = PMCCFILTR_EL0;
  else
    X[t, 64] = PMEVTYPER_EL0(UInt(PMSEL_EL0.SEL));
  endif
  elsif PSTATE_EL == EL3 then
    if UInt(PMSEL_EL0.SEL) == 31 then
      X[t, 64] = PMCCFILTR_EL0;
  else
    X[t, 64] = PMEVTYPER_EL0(UInt(PMSEL_EL0.SEL));
  endif
  elsif PSTATE_EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.TPM == '1' then
      UNDEFINED;
  elsif PMUSERENR_EL0.EN == '0'
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() & HCR_EL2.<E2H,TGE> != '1' & !(HaveEL(EL3) || SCR_EL3.FGTEn == '1') &
    HDFGWTR_EL2.PMEVTYPERn_EL0 == '1'
    then
      AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & MDCR_EL2.TPM == '1'
    then
      AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & UInt(PMSEL_EL0.SEL) != 31 & UInt(PMSEL_EL0.SEL) >= AArch64.GetNumEventCountersAccessible() then
      AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) & MDCR_EL3.TPM == '1'
    then
      if Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  elsif UInt(PMSEL_EL0.SEL) == 31 then
    PMCCFILTR_EL0 = X[t, 64];
  else
    PMEVTYPER_EL0(UInt(PMSEL_EL0.SEL)) = X[t, 64];
  endif
  end
end

MSR PMXEVTYPER_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1001</td>
<td>0b1101</td>
<td>0b001</td>
</tr>
</tbody>
</table>
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
    UNDEFINED;
elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMEVTYPERn_EL0 == '1'
    then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && MDCR_EL2.TPM == '1'
    then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && UInt(PMSELR_EL0.SEL) == 31
    then
    PMCCFILTR_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1'
        then
        UNDEFINED;
    elsif HaveEL(EL3) && MDCR_EL3.TPM == '1'
        then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        elsif UInt(PMSELR_EL0.SEL) == 31
            then
                PMEVTYPER_EL0[UInt(PMSELR_EL0.SEL)] = X[t, 64];
    elseif PSTATE.EL == EL3 then
        if UInt(PMSELR_EL0.SEL) == 31
            then
                PMCCFILTR_EL0 = X[t, 64];
elsif PSTATE.EL == EL3 then
    if UInt(PMSELR_EL0.SEL) == 31
        then
            PMCCFILTR_EL0 = X[t, 64];
D17.6 Activity Monitors registers

This section lists the Activity Monitors registers in AArch64.
D17.6.1 AMCFGR_EL0, Activity Monitors Configuration Register

The AMCFGR_EL0 characteristics are:

Purpose

Global configuration register for the activity monitors.
Provides information on supported features, the number of counter groups implemented, the total number of activity monitor event counters implemented, and the size of the counters.
AMCFGR_EL0 is applicable to both the architected and the auxiliary counter groups.

Configurations

AArch64 System register AMCFGR_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMCFGR[31:0].
AArch64 System register AMCFGR_EL0 bits [31:0] are architecturally mapped to External register AMCFGR[31:0].
This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCFGR_EL0 are UNDEFINED.

Attributes

AMCFGR_EL0 is a 64-bit register.

Field descriptions

Bits [63:32]
Reserved, RES0.

NCG, bits [31:28]
Defines the number of counter groups.
The number of implemented counter groups is [AMCFGR_EL0.NCG + 1].
If the number of implemented auxiliary activity monitor event counters is zero, this field has a value of 0b0000. Otherwise, this field has a value of 0b0001.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Bits [27:25]
Reserved, RES0.

HDBG, bit [24]
Halt-on-debug supported.
This feature must be supported, and so this bit is 0b1.
0b0 AMCR_EL0.HDBG is RES0.
0b1 AMCR_EL0.HDBG is read/write.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.
Bits [23:14]

Reserved, RAZ.

SIZE, bits [13:8]

Defines the size of activity monitor event counters.
The size of the activity monitor event counters implemented by the activity monitors Extension is
[AMCFGR_EL0.SIZE + 1].

--- Note ---

Software also uses this field to determine the spacing of counters in the memory-map. From Armv8,
the counters are at doubleword-aligned addresses.

---

Reads as 0b111111.
Access to this field is RO.

N, bits [7:0]

Defines the number of activity monitor event counters.
The total number of counters implemented in all groups by the Activity Monitors Extension is
[AMCFGR_EL0.N + 1].
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Accessing AMCFGR_EL0

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, AMCFGR_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1101</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif AMUSEREN_EL0.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    end if;
  elsif EL2Enabled() && CPTR_EL3.TAM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if;
  else
    X[<t, 64] = AMCFGR_EL0;
  end if;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if;
  elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[<t, 64] = AMCFGR_EL0;
  end if;
UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = AMCFGR_EL0;
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && CPTR_EL3.TAM == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        X[t, 64] = AMCFGR_EL0;
    end
elsif PSTATE_EL == EL3 then
    X[t, 64] = AMCFGR_EL0;
### D17.6.2 AMCG1IDR_EL0, Activity Monitors Counter Group 1 Identification Register

The AMCG1IDR_EL0 characteristics are:

**Purpose**

Defines which auxiliary counters are implemented, and which of them have a corresponding virtual offset register, AMEVCNTVOFF1<n>_EL2 implemented.

**Configurations**

This register is present only when FEAT_AMUv1p1 is implemented. Otherwise, direct accesses to AMCG1IDR_EL0 are UNDEFINED.

**Attributes**

AMCG1IDR_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:32]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, RES0</td>
<td></td>
</tr>
</tbody>
</table>

**AMEVCNTOFF1<n>_EL2, bit [n+16], for n = 15 to 0**

Indicates which implemented auxiliary counters have a corresponding virtual offset register, AMEVCNTVOFF1<n>_EL2 implemented.

- **0b0** When read, mean that AMEVCNTR1<n>_EL0 does not have an offset, or is not implemented.
- **0b1** When read, means the offset AMEVCNTVOFF1<n>_EL2 is implemented for AMEVCNTR1<n>_EL0.

**AMEVCNTR1<n>_EL0, bit [n], for n = 15 to 0**

Indicates which auxiliary counters AMEVCNTR1<n>_EL0 are implemented.

- **0b0** When read, means that AMEVCNTR1<n>_EL0 is not implemented.
- **0b1** When read, means that AMEVCNTR1<n>_EL0 is implemented.
Accessing AMCG1IDR_EL0

Accesses to this register use the following encodings in the System register encoding space:

\[ \text{MRS \( \langle Xt \rangle \), AMCG1IDR_EL0} \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1101</td>
<td>0b0010</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif AMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    if EL2Enabled() && CPTR_EL2.TAM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    else
      X[t, 64] = AMCG1IDR_EL0;
    end
  end
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = AMCG1IDR_EL0;
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = AMCG1IDR_EL0;
  end
elsif PSTATE.EL == EL3 then
  X[t, 64] = AMCG1IDR_EL0;
else
  X[t, 64] = AMCG1IDR_EL0;
end
### D17.6.3 AMCGCR_EL0, Activity Monitors Counter Group Configuration Register

The AMCGCR_EL0 characteristics are:

**Purpose**

Provides information on the number of activity monitor event counters implemented within each counter group.

**Configurations**

AArch64 System register AMCGCR_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMCGCR[31:0].

AArch64 System register AMCGCR_EL0 bits [31:0] are architecturally mapped to External register AMCGCR[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCGCR_EL0 are UNDEFINED.

**Attributes**

AMCGCR_EL0 is a 64-bit register.

**Field descriptions**

**Bits [63:16]**

Reserved, RES0.

**CG1NC, bits [15:8]**

Counter Group 1 Number of Counters. The number of counters in the auxiliary counter group.

In an implementation that includes FEAT_AMUv1, the permitted range of values is 0x0 to 0x10.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**CG0NC, bits [7:0]**

Counter Group 0 Number of Counters. The number of counters in the architected counter group.

Reads as 0x04.

Access to this field is RO.

**Accessing AMCGCR_EL0**

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, AMCGCR_EL0

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1101</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif AMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    end
    elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    else
      X[t, 64] = AMCGCR_EL0;
    end
  elif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
      UNDEFINED;
    elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
    elsif PSTATE_EL == EL2 then
      if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
        UNDEFINED;
      elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
        if Halted() && EDSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL3, 0x18);
        end
      else
        X[t, 64] = AMCGCR_EL0;
      end
    else
      PSTATE_EL == EL3 then
        X[t, 64] = AMCGCR_EL0;
      else
        PSTATE_EL == EL3 then
          X[t, 64] = AMCGCR_EL0;
D17.6.4 AMCNTENCLR0_EL0, Activity Monitors Count Enable Clear Register 0

The AMCNTENCLR0_EL0 characteristics are:

**Purpose**

Disable control bits for the architected activity monitors event counters, AMEVCNTR0<n>_EL0.

**Configurations**

AArch64 System register AMCNTENCLR0_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENCLR0[31:0].

AArch64 System register AMCNTENCLR0_EL0 bits [31:0] are architecturally mapped to External register AMCNTENCLR0[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCNTENCLR0_EL0 are UNDEFINED.

**Attributes**

AMCNTENCLR0_EL0 is a 64-bit register.

**Field descriptions**

Bits [63:16]

Reserved, RES0.

Bits [15:4]

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

P<n>, bit [n], for n = 3 to 0

Activity monitor event counter disable bit for AMEVCNTR0<n>_EL0.

**Note**

AMCGCR_EL0.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

0b0 When read, means that AMEVCNTR0<n>_EL0 is disabled. When written, has no effect.

0b1 When read, means that AMEVCNTR0<n>_EL0 is enabled. When written, disables AMEVCNTR0<n>_EL0.

The reset behavior of this field is:

- On an AMU reset, this field resets to 0.

**Accessing AMCNTENCLR0_EL0**

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE.EL == EL0 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && CPTR_EL3.TAM == '1' then
UNDEFINED;
elsif AMUSERENR_EL0.EN == '0' then
if EL2Enabled() && HCR_EL2.TGE == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && SCR_EL3.FGTEn == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = AMCNTENCLR0_EL0;
elsif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && CPTR_EL3.TAM == '1' then
UNDEFINED;
elsif EL2Enabled() && CPTR_EL3.TAM == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && SCR_EL3.FGTEn == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = AMCNTENCLR0_EL0;
elsif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && CPTR_EL3.TAM == '1' then
UNDEFINED;
elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = AMCNTENCLR0_EL0;
elsif PSTATE.EL == EL3 then
X[t, 64] = AMCNTENCLR0_EL0;
MSR AMCNTENCLR0_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRM</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1101</td>
<td>0b0010</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if IsHighestEL(PSTATE.EL) then
  AMCNTENCLR0_EL0 = X[t, 64];
else
  UNDEFINED;
D17.6.5  AMCNTENCLR1_EL0, Activity Monitors Count Enable Clear Register 1

The AMCNTENCLR1_EL0 characteristics are:

**Purpose**

Disable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>_EL0.

**Configurations**

AArch64 System register AMCNTENCLR1_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENCLR1[31:0].

AArch64 System register AMCNTENCLR1_EL0 bits [31:0] are architecturally mapped to External register AMCNTENCLR1[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCNTENCLR1_EL0 are UNDEFINED.

**Attributes**

AMCNTENCLR1_EL0 is a 64-bit register.

**Field descriptions**

| Bits [63:16] | Reserved, RES0. |
| P<n>, bit [n], for n = 15 to 0 |

Activity monitor event counter disable bit for AMEVCNTR1<n>_EL0.

When N is less than 16, bits [15:N] are RAZ/WI, where N is the value in AMCGCR_EL0.CG1NC.

Possible values of each bit are:

- **0b0**  When read, means that AMEVCNTR1<n>_EL0 is disabled. When written, has no effect.

- **0b1**  When read, means that AMEVCNTR1<n>_EL0 is enabled. When written, disables AMEVCNTR1<n>_EL0.

The reset behavior of this field is:

- On an AMU reset, this field resets to 0.

**Accessing AMCNTENCLR1_EL0**

If the number of auxiliary activity monitor event counters implemented is zero, reads and writes of AMCNTENCLR1_EL0 are UNDEFINED.

**Note**

The number of auxiliary activity monitor event counters implemented is zero exactly when AMCFGR_EL0.NCG == 0b0000.

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE_EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SOCK == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif AMUSEREN_EL0.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    endif
  else
    if EL2Enabled() && CPTR_EL2.TAM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMCNTEN1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      else
        if Halted() && EDSCR.SOCK == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL3, 0x18);
        endif
      endif
    endif
  endif
elseif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SOCK == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif AMUSEREN_EL0.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      if EL2Enabled() && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      else
        if EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMCNTEN1 == '1' then
          AArch64.SystemAccessTrap(EL2, 0x18);
        else
          if Halted() && EDSCR.SOCK == '1' then
            UNDEFINED;
          else
            AArch64.SystemAccessTrap(EL3, 0x18);
          endif
        endif
      endif
    endif
  else
    if EL2Enabled() && CPTR_EL2.TAM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMCNTEN1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      else
        if Halted() && EDSCR.SOCK == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL3, 0x18);
        endif
      endif
    endif
  endif
else
  X[t, 64] = AMCNTENCLR1_EL0;
endif
MSR AMCNTENCLR1_EL0, <Xt>

if IsHighestEL(PSTATE.EL) then
    AMCNTENCLR1_EL0 = X[t, 64];
else
    UNDEFINED;

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1101</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>
D17.6.6  AMCNTENSET0_EL0, Activity Monitors Count Enable Set Register 0

The AMCNTENSET0_EL0 characteristics are:

**Purpose**

Enable control bits for the architected activity monitors event counters, AMEVCNTR0<n>_EL0.

**Configurations**

AArch64 System register AMCNTENSET0_EL0 bits [31:0] are architecturally mapped to
AArch32 System register AMCNTENSET0[31:0].

AArch64 System register AMCNTENSET0_EL0 bits [31:0] are architecturally mapped to External
register AMCNTENSET0[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to
AMCNTENSET0_EL0 are UNDEFINED.

**Attributes**

AMCNTENSET0_EL0 is a 64-bit register.

**Field descriptions**

```
+---+---+---+---+---+---+---+---+
| 63| 32| 31| 16| 15| 14| 13| 12|
|    | RES0|    |    | RAZ/WI|    |    |    |
| 11| 10|  9|  8|  7|  6|  5|  4|
|     | RES0|     |      |      |    |    |    |
|  3|  2|  1|  0|    |  P3|  P2|  P1|  P0|
```

**Bits [63:16]**

Reserved, RES0.

**Bits [15:4]**

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might
define in a future version of the Activity Monitors architecture.

**P<n>, bit [n], for n = 3 to 0**

Activity monitor event counter enable bit for AMEVCNTR0<n>_EL0.

--- **Note** ---

AMCGCR_EL0.CG0NC identifies the number of architected activity monitor event counters. In an
implementation that includes FEAT_AMUv1, the number of architected activity monitor event
counters is 4.

Possible values of each bit are:

- **0b0** When read, means that AMEVCNTR0<n>_EL0 is disabled. When written, has no
effect.

- **0b1** When read, means that AMEVCNTR0<n>_EL0 is enabled. When written, enables
AMEVCNTR0<n>_EL0.

The reset behavior of this field is:

- On an AMU reset, this field resets to 0.

**Accessing AMCNTENSET0_EL0**

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, AMCNTENSETO_EL0

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1101</td>
<td>0b0010</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif AMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    endif
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  endif
else
  if EL2Enabled() && CPTR_EL2.TAM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMCNTEN0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  else
    X[t, 64] = AMCNTENSETO_EL0;
  endif
endif
else
  if PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
      UNDEFINED;
    elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMCNTEN0 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      endif
    else
      X[t, 64] = AMCNTENSETO_EL0;
    endif
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
      UNDEFINED;
    elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      endif
    else
      X[t, 64] = AMCNTENSETO_EL0;
    endif
  elsif PSTATE.EL == EL3 then
    X[t, 64] = AMCNTENSETO_EL0;
  endif
else
  X[t, 64] = AMCNTENSETO_EL0;
endif
if IsHighestEL(PSTATE.EL) then
  AMCNTENSET0_EL0 = X[t, 64];
else
  UNDEFINED;
D17.6.7 AMCNTENSET1_EL0, Activity Monitors Count Enable Set Register 1

The AMCNTENSET1_EL0 characteristics are:

**Purpose**

Enable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<\text{n}>_EL0.

**Configurations**

AArch64 System register AMCNTENSET1_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENSET1[31:0].

AArch64 System register AMCNTENSET1_EL0 bits [31:0] are architecturally mapped to External register AMCNTENSET1[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCNTENSET1_EL0 are UNDEFINED.

**Attributes**

AMCNTENSET1_EL0 is a 64-bit register.

**Field descriptions**

![Field diagram]

Bits [63:16]

Reserved, RES0.

P<\text{n}>, bit \text{n}, for \text{n} = 15 to 0

Activity monitor event counter enable bit for AMEVCNTR1<\text{n}>_EL0.

When \text{N} is less than 16, bits [15:\text{N}] are RAZ/WI, where \text{N} is the value in AMCGCR_EL0.CG1NC.

Possible values of each bit are:

0b0 When read, means that AMEVCNTR1<\text{n}>_EL0 is disabled. When written, has no effect.

0b1 When read, means that AMEVCNTR1<\text{n}>_EL0 is enabled. When written, enables AMEVCNTR1<\text{n}>_EL0.

The reset behavior of this field is:

- On an AMU reset, this field resets to 0.

**Accessing AMCNTENSET1_EL0**

If the number of auxiliary activity monitor event counters implemented is zero, reads and writes of AMCNTENSET1_EL0 are UNDEFINED.

**Note**

The number of auxiliary activity monitor counters implemented is zero when AMCFGCR_EL0.NCG == 0b0000.

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE_EL == EL0 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" & CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif AMUSERENR_EL0.EN == '0' then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      if EL2Enabled() & CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      elsif EL2Enabled() & HCR_EL2.<E2H,TGE> != '11' & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &
        HAFGRTR_EL2.AMCNTEN1 == '1' then
          AArch64.SystemAccessTrap(EL2, 0x18);
      elsif HaveEL(EL3) & CPTR_EL3.TAM == '1' then
        if Halted() & EDSCR.SDD == '1' then
          UNDEFINED;
        elsif Halted() & EDSCR.SDD == '1' then
          if EL2Enabled() & HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
          else
            if EL2Enabled() & CPTR_EL2.TAM == '1' then
              AArch64.SystemAccessTrap(EL2, 0x18);
            elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &
              HAFGRTR_EL2.AMCNTEN1 == '1' then
                AArch64.SystemAccessTrap(EL2, 0x18);
            elsif HaveEL(EL3) & CPTR_EL3.TAM == '1' then
              if Halted() & EDSCR.SDD == '1' then
                UNDEFINED;
            end
          end
        end
      elsif CPTR_EL3.TAM == '1' then
        if Halted() & EDSCR.SDD == '1' then
          UNDEFINED;
        elsif Halted() & EDSCR.SDD == '1' then
          if EL2Enabled() & HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
          else
            if EL2Enabled() & CPTR_EL2.TAM == '1' then
              AArch64.SystemAccessTrap(EL2, 0x18);
            elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &
              HAFGRTR_EL2.AMCNTEN1 == '1' then
                AArch64.SystemAccessTrap(EL2, 0x18);
            elsif HaveEL(EL3) & CPTR_EL3.TAM == '1' then
              if Halted() & EDSCR.SDD == '1' then
                UNDEFINED;
            end
          end
        end
      end
  else
    X[t, 64] = AMCNTENSET1_EL0;
  end
end

if PSTATE_EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" & CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif EL2Enabled() & CPTR_EL2.TAM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &
    HAFGRTR_EL2.AMCNTEN1 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) & CPTR_EL3.TAM == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    elsif CPTR_EL3.TAM == '1' then
      if Halted() & EDSCR.SDD == '1' then
        if EL2Enabled() & HCR_EL2.TGE == '1' then
          AArch64.SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() & CPTR_EL2.TAM == '1' then
          AArch64.SystemAccessTrap(EL2, 0x18);
        elsif HaveEL(EL3) & CPTR_EL3.TAM == '1' then
          if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        end
      end
    end
  elsif CPTR_EL3.TAM == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
  end
end

if PSTATE_EL == EL3 then
  X[t, 64] = AMCNTENSET1_EL0;
else
  X[t, 64] = AMCNTENSET1_EL0;
end
**MSR AMCNTENSET1_EL0, <Xt>**

\[
\text{if IsHighestEL(PSTATE.EL) then} \\
\quad \text{AMCNTENSET1_EL0} = X[t, 64]; \\
\text{else} \\
\quad \text{UNDEFINED;
}\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1101</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>
D17.6.8 AMCR_EL0, Activity Monitors Control Register

The AMCR_EL0 characteristics are:

**Purpose**
Global control register for the activity monitors implementation. AMCR_EL0 is applicable to both the architected and the auxiliary counter groups.

**Configurations**
AArch64 System register AMCR_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMCR[31:0].
AArch64 System register AMCR_EL0 bits [31:0] are architecturally mapped to External register AMCR[31:0].
This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCR_EL0 are UNDEFINED.

**Attributes**
AMCR_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>RES0</td>
</tr>
<tr>
<td>62</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>RES0</td>
</tr>
<tr>
<td>60</td>
<td>RES0</td>
</tr>
<tr>
<td>59</td>
<td>RES0</td>
</tr>
<tr>
<td>58</td>
<td>RES0</td>
</tr>
<tr>
<td>57</td>
<td>RES0</td>
</tr>
<tr>
<td>56</td>
<td>CG1RZ</td>
</tr>
<tr>
<td>55</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>HDBG</td>
</tr>
<tr>
<td>53</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>CG1RZ</td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:18]**
Reserved, RES0.

**CG1RZ, bit [17]**

*When FEAT_AMUv1p1 is implemented:*
Counter Group 1 Read Zero.

0b0 System register reads of AMEVCNTR1<n>_EL0 return the event count at all implemented and enabled Exception levels.

0b1 If the current Exception level is the highest implemented Exception level, system register reads of AMEVCNTR1<n>_EL0 return the event count. Otherwise, reads of AMEVCNTR1<n>_EL0 return a zero value.

**Note**
Reads from the memory-mapped view are unaffected by this field.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**Bits [16:11]**
Reserved, RES0.
HDBG, bit [10]

This bit controls whether activity monitor counting is halted when the PE is halted in Debug state.

- **0b0**: Activity monitors do not halt counting when the PE is halted in Debug state.
- **0b1**: Activity monitors halt counting when the PE is halted in Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Bits [9:0]**

Reserved, RES0.

**Accessing AMCR_EL0**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, AMCR_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1101</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'' && CPTR_EL3.TAM == '1' then
UNDEFINED;
elsif AMUSERENR_EL0.EN == '0' then
if EL2Enabled() && HCR_EL2.TGE == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif haveEL(EL3) && CPTR_EL3.TAM == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = AMCR_EL0;
elsif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'' && CPTR_EL3.TAM == '1' then
UNDEFINED;
elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = AMCR_EL0;
elsif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'' && CPTR_EL3.TAM == '1' then
UNDEFINED;
elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else

X[t, 64] = AMCR_EL0;
elsif PSTATE.EL == EL3 then
    X[t, 64] = AMCR_EL0;

**MSR AMCR_EL0, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1101</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if IsHighestEL(PSTATE.EL) then
    AMCR_EL0 = X[t, 64];
else
    UNDEFINED;
D17.6.9 AMEVCNTR0<n>_EL0, Activity Monitors Event Counter Registers 0, n = 0 - 3

The AMEVCNTR0<n>_EL0 characteristics are:

**Purpose**

Provides access to the architected activity monitor event counters.

**Configurations**

AArch64 System register AMEVCNTR0<n>_EL0 bits [63:0] are architecturally mapped to
AArch32 System register AMEVCNTR0<n>[63:0].

AArch64 System register AMEVCNTR0<n>_EL0 bits [63:0] are architecturally mapped to
External register AMEVCNTR0<n>[63:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to
AMEVCNTR0<n>_EL0 are UNDEFINED.

**Attributes**

AMEVCNTR0<n>_EL0 is a 64-bit register.

**Field descriptions**

```
+--------+--------+
| 63     | 32     |
| 31     | 0      |

ACNT
```

**ACNT, bits [63:0]**

Architected activity monitor event counter n.

Value of architected activity monitor event counter n, where n is the number of this register and is
a number from 0 to 3.

If FEAT_AMUv1p1 is implemented, HCR_EL2.AMVOFFEN is 1, SCR_EL3.AMVOFFEN is 1,
HCR_EL2.{E2H, TGE} is not {1,1}, and EL2 is implemented in the current Security state, access
to these registers at EL0 or EL1 return (PCount<63:0> - AMEVCNTRVOFF0<n>_EL2<63:0>).

PCount is the physical count returned when AMEVCNTR0<n>_EL0 is read from EL2 or EL3.

If the counter is enabled, writes to this register have UNPREDICTABLE results.

The reset behavior of this field is:

- On an AMU reset, this field resets to 0.

**Accessing AMEVCNTR0<n>_EL0**

If <n> is greater than or equal to the number of architected activity monitor event counters, reads and writes of
AMEVCNTR0<n>_EL0 are UNDEFINED.

--- Note ---

AMECGCR_EL0.CG0NC identifies the number of architected activity monitor event counters.

Accesses to this register use the following encodings in the System register encoding space:
integer m = UInt(CRm<0>:op2<2:0>);

if m >= 4 then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TAM == '1' then
        UNDEFINED;
elsif AMUSERENR_EL0.EN == '0' then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
else
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() & CPTR_EL2.TAM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & HCR_EL2.<E2H,TGE> != '11' & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMEVCNTR0<El0> == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TAM == '1' then
    if Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
endif;
else
    X[t, 64] = AMEVCNTR0_EL0[m];
elsif PSTATE.EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TAM == '1' then
        UNDEFINED;
elsif EL2Enabled() & CPTR_EL2.TAM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HAFGRTR_EL2.AMEVCNTR0<El0> == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & CPTR_EL3.TAM == '1' then
    if Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = AMEVCNTR0_EL0[m];
elsif PSTATE.EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & CPTR_EL3.TAM == '1' then
        UNDEFINED;
elsif HaveEL(EL3) & CPTR_EL3.TAM == '1' then
    if Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = AMEVCNTR0_EL0[m];
elsif PSTATE.EL == EL3 then
    X[t, 64] = AMEVCNTR0_EL0[m];
integer m = UInt(CRn<0>:op2<2:0>);
if m >= 4 then
    UNDEFINED;
elsif IsHighestEL(PSTATE.EL) then
    AMEVCNTR0_EL0[m] = X[t, 64];
else
    UNDEFINED;
D17.6.10  AMEVCNTR1<n>_EL0, Activity Monitors Event Counter Registers 1, n = 0 - 15

The AMEVCNTR1<n>_EL0 characteristics are:

**Purpose**

Provides access to the auxiliary activity monitor event counters.

**Configurations**

AArch64 System register AMEVCNTR1<n>_EL0 bits [63:0] are architecturally mapped to AArch32 System register AMEVCNTR1<n>[63:0].

AArch64 System register AMEVCNTR1<n>_EL0 bits [63:0] are architecturally mapped to External register AMEVCNTR1<n>[63:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMEVCNTR1<n>_EL0 are UNDEFINED.

**Attributes**

AMEVCNTR1<n>_EL0 is a 64-bit register.

**Field descriptions**

ACNT, bits [63:0]

Auxiliary activity monitor event counter n.

Value of auxiliary activity monitor event counter n, where n is the number of this register and is a number from 0 to 15.

If FEAT_AMUv1p1 is implemented, HCR_EL2.AMVOFFEN is 1, SCR_EL3.AMVOFFEN is 1, HCR_EL2.{E2H, TGE} is not {1,1}, EL2 is implemented in the current Security state, and AMCR_EL0.CG1RZ is 0, reads to these registers at EL0 or EL1 return (PCount<63:0> - AMEVCNTR1<n>_EL2<63:0>).

PCount is the physical count returned when AMEVCNTR1<n>_EL0 is read from EL2 or EL3.

If the counter is enabled, writes to this register have UNPREDICTABLE results.

The reset behavior of this field is:

- On an AMU reset, this field resets to 0.

**Accessing AMEVCNTR1<n>_EL0**

If <n> is greater than or equal to the number of auxiliary activity monitor event counters, reads and writes of AMEVCNTR1<n>_EL0 are UNDEFINED.

--- Note ---

AMECGCR_EL0.CG1NC identifies the number of auxiliary activity monitor event counters.

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, AMEVCNTR1<m>_EL0

integer m = UInt(CRn<0>:op2<2:0>);
if m >= NUM_AMU_CG1_MONITORS then
  UNDEFINED;
elsif !IsG1ActivityMonitorImplemented(m) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif AMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    end
  end
  elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  end
  elsif HAFGRTR_EL2.AMEVCNTR1<m>_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if AMCR_EL0.CG1RZ == '1' then
      X[t, 64] = Zeros(64);
    else
      X[t, 64] = AMEVCNTR1_EL0[m];
    end
  end
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  end
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HAFGRTR_EL2.AMEVCNTR1<m>_EL0 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if AMCR_EL0.CG1RZ == '1' then
      X[t, 64] = Zeros(64);
    else
      X[t, 64] = AMEVCNTR1_EL0[m];
    end
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  end
  elsif !IsHighestEL(PSTATE.EL) && AMCR_EL0.CG1RZ == '1' then
    X[t, 64] = Zeros(64);
  else
    if AMCR_EL0.CG1RZ == '1' then
      X[t, 64] = Zeros(64);
    else
      X[t, 64] = AMEVCNTR1_EL0[m];
    end
  end
else
  if AMCR_EL0.CG1RZ == '1' then
    X[t, 64] = Zeros(64);
  else
    X[t, 64] = AMEVCNTR1_EL0[m];
  end
\[
X[t, 64] = \text{AMEVCNTR1}_\text{EL0}[m]; \\
\text{elsif PSTATE.EL == EL3 then} \\
X[t, 64] = \text{AMEVCNTR1}_\text{EL0}[m]; \\
\]

**MSR AMEVCNTR1<_m>_EL0, <Xt>**

\[
\begin{array}{cccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b011 & 0b1101 & 0b110:m[3] & m[2:0] \\
\end{array}
\]

integer \( m = \text{UInt}(\text{CRm<0>:op2<2:0>}); \)

if \( m \geq \text{NUM_AMU_CG1_MONITORS} \) then
UNDEFINED;
elsif !IsG1ActivityMonitorImplemented(\( m \)) then
UNDEFINED;
elsif IsHighestEL(PSTATE.EL) then
\( \text{AMEVCNTR1}_\text{EL0}[m] = X[t, 64]; \)
else
UNDEFINED;
The AMEVCNTVOFF0<n>_EL2 characteristics are:

**Purpose**

Holds the 64-bit virtual offset for architectured activity monitor events.

**Configurations**

This register is present only when FEAT_AMUv1p1 is implemented. Otherwise, direct accesses to AMEVCNTVOFF0<n>_EL2 are UNDEFINED.

**Attributes**

AMEVCNTVOFF0<n>_EL2 is a 64-bit register.

**Field descriptions**

**Bits [63:0]**

Virtual offset.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing AMEVCNTVOFF0<n>_EL2**

If <n> is not 0, 2 or 3, reads and writes of AMEVCNTVOFF0<n>_EL2 are UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, AMEVCNTVOFF0<m>_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1101</td>
<td>0b100:m[3]</td>
<td>m[2:0]</td>
</tr>
</tbody>
</table>

integer m = UInt(CRm<0>:op2<2:0>);

if m >= 4 then
    UNDEFINED;
elsif m != 0 && m != 2 && m ! = 3 then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = NVMem[0xA00 + (8 * m)];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
  UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.AMOVFFN == '0' then
  UNDEFINED;
elsif HaveEL(EL3) && SCR_EL3.AMOVFFN == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  elseif HaveEL(EL3) && SCR_EL3.AMOVFFN == '0' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  elseif HaveEL(EL3) && SCR_EL3.AMOVFFN == '0' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
else
  X[t, 64] = AMEVCNTVFOFF0_EL2[m];
elsif PSTATE.EL == EL3 then
  X[t, 64] = AMEVCNTVFOFF0_EL2[m];

MSR AMEVCNTVFOFF0<e> EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1101</td>
<td>0b100:m[3]</td>
<td>m[2:0]</td>
</tr>
</tbody>
</table>

integer m = UInt(CRm<0>:op2<2:0>);
if m >= 4 then
  UNDEFINED;
elifs m != 0 && m != 2 && m != 3 then
  UNDEFINED;
elifs PSTATE.EL == EL0 then
  UNDEFINED;
elifs PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    NVMem[0xA00 + (8 * m)] = X[t, 64];
elifs EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  end
  UNDEFINED;
elifs PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
    UNDEFINED;
elifs Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.AMOVFFN == '0' then
    UNDEFINED;
elifs HaveEL(EL3) && SCR_EL3.AMOVFFN == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  elseif HaveEL(EL3) && SCR_EL3.AMOVFFN == '0' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
else
  AMEVCNTVFOFF0_EL2[m] = X[t, 64];
elifs PSTATE.EL == EL3 then
  AMEVCNTVFOFF0_EL2[m] = X[t, 64];
D17.6.12 AMEVCNTVOFF1<n>_EL2, Activity Monitors Event Counter Virtual Offset Registers 1, n = 0 - 15

The AMEVCNTVOFF1<n>_EL2 characteristics are:

**Purpose**

Holds the 64-bit virtual offset for auxiliary activity monitor events.

**Configurations**

This register is present only when FEAT_AMUv1p1 is implemented. Otherwise, direct accesses to AMEVCNTVOFF1<n>_EL2 are UNDEFINED.

**Attributes**

AMEVCNTVOFF1<n>_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Virtual offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Accessing AMEVCNTVOFF1<n>_EL2**

--- Note ---

AMCG1IDR_EL0 identifies which auxiliary activity monitor event counters have a corresponding virtual offset implemented.

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS} \ <Xt>, \ \text{AMEVCNTVOFF1}<m>_\text{EL2}
\]

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b100 & 0b1101 & 0b101:m[3] & m[2:0] \\
\end{array}
\]

\[
\text{integer} \ m = \text{UInt(CRm<0>:op2<2:0> '');}
\]

\[
\text{if } m \geq \text{NUM_AMU_CG1_MONITORS} \text{ then UNDEFINED;}
\]

\[
\text{elsif !IsGIActivityMonitorOffsetImplemented(m) then UNDEFINED;}
\]

\[
\text{elsif PSTATE.EL == EL0 then UNDEFINED;}
\]

\[
\text{elsif PSTATE.EL == EL1 then}
\]

\[
\text{if EL2Enabled()} \&\& \text{HCR_EL2.<NV2,NV> == '11' then}
\]

\[
\begin{align*}
X[t, 64] &= \text{NVMem}[0xA80 + (8 \times m)]; \\
\text{elsif } \text{EL2Enabled()} \&\& \text{HCR_EL2.NV} &= '1' \text{ then}
\end{align*}
\]
AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elseif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.AMVOFFEN == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.AMVOFFEN == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  elseif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  else
    X[t, 64] = AMEVCNTVOFF1_EL2[m];
  endif
elsif PSTATE.EL == EL3 then
  X[t, 64] = AMEVCNTVOFF1_EL2[m];
else
  integer m = UInt(CRm<0>:op2<2:0>);
  if m >= NUM_AMU_CG1_MONITORS then
    UNDEFINED;
  elseif !IsG1ActivityMonitorOffsetImplemented(m) then
    UNDEFINED;
  elsif PSTATE.EL == EL0 then
    UNDEFINED;
  elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
      NVMem[0xA80 + (8 * m)] = X[t, 64];
    elseif EL2Enabled() && HCR_EL2.NV == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      UNDEFINED;
    endif
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
      UNDEFINED;
    elseif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.AMVOFFEN == '0' then
      UNDEFINED;
    elseif HaveEL(EL3) && SCR_EL3.AMVOFFEN == '0' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      endif
    elseif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      endif
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  endif
else
  UNDEFINED;
endif

---

**MSR AMEVCNTVOFF1<	exttt{m}>, EL2, \texttt{<Xt>}**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b101</td>
<td>0b101:m[3]</td>
<td>m[2:0]</td>
</tr>
</tbody>
</table>
AMEVCNT0FF1_EL2[m] = X[t, 64];
elseif PSTATE_EL == EL3 then
  AMEVCNT0FF1_EL2[m] = X[t, 64];
D17.6.13 AMEVTYPER0<n>_EL0, Activity Monitors Event Type Registers 0, n = 0 - 3

The AMEVTYPER0<n>_EL0 characteristics are:

**Purpose**

Provides information on the events that an architected activity monitor event counter AMEVCNTR0<n>_EL0 counts.

**Configurations**

AArch64 System register AMEVTYPER0<n>_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMEVTYPER0<n>[31:0].

AArch64 System register AMEVTYPER0<n>_EL0 bits [31:0] are architecturally mapped to External register AMEVTYPER0<n>[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMEVTYPER0<n>_EL0 are UNDEFINED.

**Attributes**

AMEVTYPER0<n>_EL0 is a 64-bit register.

**Field descriptions**

![Field diagram]

Bits [63:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the architected activity monitor event counter AMEVCNTR0<n>_EL0. The value of this field is architecturally mandated for each architected counter.

The following table shows the mapping between required event numbers and the corresponding counters:

<table>
<thead>
<tr>
<th>Event Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0011</td>
<td>When n == 0:</td>
</tr>
<tr>
<td></td>
<td>Processor frequency cycles</td>
</tr>
<tr>
<td>0x4004</td>
<td>When n == 1:</td>
</tr>
<tr>
<td></td>
<td>Constant frequency cycles</td>
</tr>
<tr>
<td>0x0008</td>
<td>When n == 2:</td>
</tr>
<tr>
<td></td>
<td>Instructions retired</td>
</tr>
<tr>
<td>0x4005</td>
<td>When n == 3:</td>
</tr>
<tr>
<td></td>
<td>Memory stall cycles</td>
</tr>
</tbody>
</table>

**Accessing AMEVTYPER0<n>_EL0**

If <n> is greater than or equal to the number of architected activity monitor event counters, reads and writes of AMEVTYPER0<n>_EL0 are UNDEFINED.

--- Note ---

AMCGCR_EL0.CG0NC identifies the number of architected activity monitor event counters.
Accesses to this register use the following encodings in the System register encoding space:

\[
MRS <Xt>, AMEVTYPER0<m>\_EL0
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1101</td>
<td>0b011:m[3]</td>
<td>m[2:0]</td>
</tr>
</tbody>
</table>

integer \(m = \text{UInt}(\text{CRn}<0>:\text{op2}<2:0>)\);

if \(m >= 4\) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
        UNDEFINED;
elsif AMUSEREN_EL0.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && CPTR_EL3.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = AMEVTYPER0_EL0[m];
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
        UNDEFINED;
elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    AArch64.SystemAccessTrap(EL3, 0x18);
    if Halted() && HaveEL(EL3) && CPTR_EL3.TAM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = AMEVTYPER0_EL0[m];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
        UNDEFINED;
elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = AMEVTYPER0_EL0[m];
elsif PSTATE.EL == EL3 then
    X[t, 64] = AMEVTYPER0_EL0[m];
D17.6.14 AMEVTYPE1<n>_EL0, Activity Monitors Event Type Registers 1, n = 0 - 15

The AMEVTYPE1<n>_EL0 characteristics are:

**Purpose**
Provides information on the events that an auxiliary activity monitor event counter AMEVCNTR1<n>_EL0 counts.

**Configurations**
AArch64 System register AMEVTYPE1<n>_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMEVTYPE1<n>[31:0].
AArch64 System register AMEVTYPE1<n>_EL0 bits [31:0] are architecturally mapped to External register AMEVTYPE1<n>[31:0].
This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMEVTYPE1<n>_EL0 are UNDEFINED.

**Attributes**
AMEVTYPE1<n>_EL0 is a 64-bit register.

**Field descriptions**

**Bits [63:16]**
Reserved, RES0.

**evtCount, bits [15:0]**
Event to count. The event number of the event that is counted by the auxiliary activity monitor event counter AMEVCNTR1<n>_EL0.
It is IMPLEMENTATION DEFINED what values are supported by each counter.
If software writes a value to this field which is not supported by the corresponding counter AMEVCNTR1<n>_EL0, then:

- It is UNPREDICTABLE which event will be counted.
- The value read back is UNKNOWN.

The event counted by AMEVCNTR1<n>_EL0 might be fixed at implementation. In this case, the field is read-only and writes are UNDEFINED.

If the corresponding counter AMEVCNTR1<n>_EL0 is enabled, writes to this register have UNPREDICTABLE results.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing AMEVTYPE1<n>_EL0**
If <n> is greater than or equal to the number of auxiliary activity monitor event counters, reads and writes of AMEVTYPE1<n>_EL0 are UNDEFINED.

---
**Note**
AMCGCR_EL0.CG1NC identifies the number of auxiliary activity monitor event counters.
Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, AMEVTYPER1<\text{\textless}m\text{\textgreater}\_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b11</td>
<td>0b11</td>
<td>0b11</td>
<td>m[3] m[2:0]</td>
</tr>
</tbody>
</table>

integer m = UInt(CRm\&0b:op2<2:0>);

if m >= NUM_AMU_CG1_MONITORS then
UNDEFINED;
elsif !IsGIActivityMonitorImplemented(m) then
UNDEFINED;
elsif PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
    UNDEFINED;
elsif AMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && CR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    end
  elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL0, 0x18);
  end
  if HaveEL(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
    X[t, 64] = AMEVTYPER1_EL0[m];
  elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
      UNDEFINED;
elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      elsif EL2Enabled() && (HaveEL(EL3) || EDSCR.FGTEn == '1') && HAFGRTR_EL2.AMEVTYPER1<\text{\textless}m\text{\textgreater}\_EL0 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
        if Halted() && EDSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL3, 0x18);
        end
        X[t, 64] = AMEVTYPER1_EL0[m];
      elsif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
          UNDEFINED;
elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
            if Halted() && EDSCR.SDD == '1' then
              UNDEFINED;
            else
              AArch64.SystemAccessTrap(EL3, 0x18);
            end
            X[t, 64] = AMEVTYPER1_EL0[m];
      elsif PSTATE.EL == EL3 then
        X[t, 64] = AMEVTYPER1_EL0[m];
    elsif PSTATE.EL == EL0 then
    X[t, 64] = AMEVTYPER1_EL0[m];
MSR AMEVTYPE1<\textit{m}>_EL0, \textit{Xt}

\[
\begin{array}{cccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b011 & 0b1101 & 0b111: & m[3] & m[2:0]
\end{array}
\]

integer \( m = \text{UInt}(\text{CRm}<0>:\text{op2}<2:0>) \);

if \( m \geq \text{NUM_AMU_CLG1_MONITORS} \) then
  UNDEFINED;
elsif !\text{IsGIActivityMonitorImplemented}(m) then
  UNDEFINED;
elsif \text{IsHighestEL}(\text{PSTATE.EL}) \& \& !\text{boolean IMPLEMENTATION_DEFINED "AMEVCNTR1_EL0[m] is fixed"} \) then
  \text{AMEVTYPER1_EL0}[m] = X[t, 64];
else
  UNDEFINED;
D17.6.15  AMUSERENR_EL0, Activity Monitors User Enable Register

The AMUSERENR_EL0 characteristics are:

**Purpose**

Global user enable register for the activity monitors. Enables or disables EL0 access to the activity monitors. AMUSERENR_EL0 is applicable to both the architected and the auxiliary counter groups.

**Configurations**

AArch64 System register AMUSERENR_EL0 bits [31:0] are architecturally mapped to AArch32 System register AMUSEREN[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMUSERENR_EL0 are **UNDEFINED**.

**Attributes**

AMUSERENR_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>31:1</td>
<td>EN, bit [0]</td>
</tr>
</tbody>
</table>

Traps EL0 accesses to the activity monitors registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to the following registers are trapped, reported using EC syndrome value 0x18:
  - AMCFGR_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0, AMCR_EL0, AMEVCNTR0<n>_EL0, AMEVCNTR1<n>_EL0, AMEVTYPER0<n>_EL0, and AMEVTYPER1<n>_EL0.

- In AArch32 state, MRC and MCR accesses to the following registers are trapped and reported using EC syndrome value 0x03, MRCC and MCRR accesses are trapped and reported using EC syndrome value 0x04:
  - AMCFGR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVCNTR0<n>, AMEVCNTR1<n>, AMEVTYPER0<n>, and AMEVTYPER1<n>.

0b0  EL0 accesses to the activity monitors registers are trapped.

0b1  This control does not cause any instructions to be trapped. Software can access all activity monitor registers at EL0.

**Note**

AMUSERENR_EL0 can always be read at EL0 and is not governed by this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
Accessing AMUSERENR_EL0

Accesses to this register use the following encodings in the System register encoding space:

### MRS <Xt>, AMUSERENR_EL0

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1101</td>
<td>0b0010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
        UNDEFINED;
    elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = AMUSERENR_EL0;
    endif
else if PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
        UNDEFINED;
    elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = AMUSERENR_EL0;
    endif
else if PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = AMUSERENR_EL0;
    endif
else if PSTATE_EL == EL3 then
    X[t, 64] = AMUSERENR_EL0;
else
    X[t, 64] = AMUSERENR_EL0;
endif
```

### MSR AMUSERENR_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1101</td>
<td>0b0010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = AMUSERENR_EL0;
    endif
else if PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = AMUSERENR_EL0;
    endif
else if PSTATE_EL == EL3 then
    X[t, 64] = AMUSERENR_EL0;
else
    X[t, 64] = AMUSERENR_EL0;
endif
```
when SDD == '1' && CPTR_EL3.TAM == '1' then
    UNDEFINED;
elsif EL2Enabled() && CPTR_EL2.TAM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        AMUSERENR_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && CPTR_EL3.TAM == '1' then
        UNDEFINED;
elsif HaveEL(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
else
        AArch64.SystemAccessTrap(EL3, 0x18);
else
    AMUSERENR_EL0 = X[t, 64];
elsif PSTATE.EL == EL3 then
    AMUSERENR_EL0 = X[t, 64];
D17.7  Statistical Profiling Extension registers

This section lists the Statistical Profiling Extension registers in AArch64.
D17.7.1 PMBIDR_EL1, Profiling Buffer ID Register

The PMBIDR_EL1 characteristics are:

**Purpose**

Provides information to software as to whether the buffer can be programmed at the current Exception level.

**Configurations**

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMBIDR_EL1 are UNDEFINED.

**Attributes**

PMBIDR_EL1 is a 64-bit register.

**Field descriptions**

Bits [63:12]

Reserved, RES0.

EA, bits [11:8]

External Abort handling. Describes how the PE manages External aborts on writes made by the Statistical Profiling Unit to the Profiling Buffer.

- 0b0000 Not described.
- 0b0001 The PE ignores External aborts on writes made by the Statistical Profiling Unit.
- 0b0010 The External abort generates an SError interrupt at the PE.

All other values are reserved.

From Armv8.8, the value 0b0000 is not permitted.

PMBIDR_EL1.EA describes only External aborts generated by the write to memory. External aborts on a translation table walk made by the Statistical Profiling Unit generate Profiling Buffer management events reported as MMU faults using PMBSR_EL1.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bits [7:6]

Reserved, RES0.

**F, bit [5]**

Flag updates. Describes how address translations performed by the Statistical Profiling Unit manage the Access flag and dirty state.

- 0b0 Hardware management of the Access flag and dirty state for accesses made by the Statistical Profiling Unit is always disabled for all translation stages.
- 0b1 Hardware management of the Access flag and dirty state for accesses made by the Statistical Profiling Unit is controlled in the same way as explicit memory accesses in the Profiling Buffer owning translation regime.
Note

If hardware management of the Access flag is disabled for a stage of translation, an access to a Page or Block with the Access flag bit not set in the descriptor will generate an Access Flag fault.

If hardware management of the dirty state is disabled for a stage of translation, an access to a Page or Block will ignore the Dirty Bit Modifier in the descriptor and might generate a Permission fault, depending on the values of the access permission bits in the descriptor.

From Armv8.8, the value 0 is not permitted.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

P, bit [4]

Programming not allowed. When read at EL3, this field reads as zero. Otherwise, indicates that the Profiling Buffer is owned by a higher Exception level or another Security state. Defined values are:

0b0 Programming is allowed.
0b1 Programming not allowed.

The value read from this field depends on the current Exception level and the Effective values of MDCR_EL3.NSPB, MDCR_EL3.NSPBE, and MDCR_EL2.E2PB:

- If EL3 is implemented, and the owning Security state is Secure state, this field reads as one from:
  - Non-secure EL1 and Non-secure EL2.
  - If FEAT_RME is implemented, Realm EL1 and Realm EL2.
  - If Secure EL2 is implemented and enabled, and MDCR_EL2.E2PB is 0b00, Secure EL1.

- If EL3 is implemented, and the owning Security state is Non-secure state, this field reads as one from:
  - Secure EL1.
  - If Secure EL2 is implemented, Secure EL2.
  - If EL2 is implemented and MDCR_EL2.E2PB is 0b00, Non-secure EL1.
  - If FEAT_RME is implemented, Realm EL1 and Realm EL2.

- If FEAT_RME is implemented, and the owning Security state is Realm state, this field reads as one from:
  - Non-secure EL1 and Non-secure EL2.
  - Secure EL1 and Secure EL2.
  - If MDCR_EL2.E2PB is 0b00, Realm EL1.

- If EL3 is not implemented, EL2 is implemented, and MDCR_EL2.E2PB is 0b00, this field reads as one from EL1.

- Otherwise, this field reads as zero.

Align, bits [3:0]

Defines the minimum alignment constraint for writes to PMBPTR_EL1. Defined values are:

0b0000 Byte.
0b0001 Halfword.
0b0010 Word.
0b0011 Doubleword.
0b0100 16 bytes.
0b0101 32 bytes.
0b0110 64 bytes.
Accessing PMBIDR_EL1

Access to this field is RO.

For more information, see Restrictions on the current write pointer on page D13-5458.

This field has an IMPLEMENTATION DEFINED value.

### Accessing PMBIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

#### MRS <Xt>, PMBIDR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1010</td>
<td>0b111</td>
</tr>
</tbody>
</table>

If this field is non-zero, then every record is a multiple of this size.

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMBIDR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = PMBIDR_EL1;
elsif PSTATE.EL == EL2 then
  X[t, 64] = PMBIDR_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = PMBIDR_EL1;
D17.7.2 PMBLIMITR_EL1, Profiling Buffer Limit Address Register

The PMBLIMITR_EL1 characteristics are:

**Purpose**

Defines the upper limit for the profiling buffer, and enables the profiling buffer

**Configurations**

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMBLIMITR_EL1 are UNDEFINED.

**Attributes**

PMBLIMITR_EL1 is a 64-bit register.

**Field descriptions**

LIMIT, bits [63:12]

Limit address. PMBLIMITR_EL1.LIMIT:Zeros(12) is the address of the first byte in memory after the last byte in the profiling buffer. If the smallest implemented translation granule is not 4KB, then bits[N-1:12] are RES0, where N is the IMPLEMENTATION DEFINED value, Log2(smallest implemented translation granule).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:6]

Reserved, RES0.

PMFZ, bit [5]

*When FEAT_SPEv1p2 is implemented:*

Freeze PMU on SPE event. Stop PMU event counters when PMBSR_EL1.S == 1.

0b0 Do not freeze PMU event counters on Statistical Profiling Buffer Management event.

0b1 Freeze PMU event counters on Statistical Profiling Buffer Management event.

The PMU event counters affected by this control is controlled by PMCR_EL0.FZS and, if EL2 is implemented, MDCR_EL2.HPMFZS. See the descriptions of these control bits for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [4:3]

Reserved, RES0.
**FM, bits [2:1]**

Fill mode.

- **0b00** Fill mode. Stop collection and raise maintenance interrupt on buffer fill.
- **0b10** When FEAT_SPEv1p2 is implemented: Discard mode. All output is discarded.

All other values are reserved.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**E, bit [0]**

Profiling Buffer enable

- **0b0** All output is discarded.
- **0b1** Profiling buffer enabled.

The reset behavior of this field is:
- On a Warm reset, this field resets to **0**.

### Accessing PMBLIMITR_EL1

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, PMBLIMITR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & MDCC_EL3.NSPB[0] == '0' || MDCC_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(ATTRcombine) && MDCC_EL3.NSPBE != SCR_EL3.NSE)) then
        UNDEFINED;
    elsif EL2Enabled() & ((HaveEl(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMBLIMITR_EL1 == '1') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & MDCC_EL2.E2PB == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEl(EL3) & (MDCC_EL3.NSPB[0] == '0' || MDCC_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(ATTRcombine) && MDCC_EL3.NSPBE != SCR_EL3.NSE)) then
        if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        endif
    elseif EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = NVMem[0x800];
    else
        X[t, 64] = PMBLIMITR_EL1;
    endif
elsif PSTATE.EL == EL2 then
    if Halted() & HaveEl(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & MDCC_EL3.NSPB[0] == '0' || MDCC_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(ATTRcombine) && MDCC_EL3.NSPBE != SCR_EL3.NSE)) then
        UNDEFINED;
    elsif HaveEl(EL3) & (MDCC_EL3.NSPB[0] == '0' || MDCC_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(ATTRcombine) && MDCC_EL3.NSPBE != SCR_EL3.NSE)) then
        if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        endif
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    endif
else
    AArch64.SystemAccessTrap(EL3, 0x18);
endif
```

AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = PMBLIMITR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = PMBLIMITR_EL1;

MSR PMBLIMITR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDSCR.EL3.NSPB[0] == '0' || MDSCR.EL3.NSPB[1] != SCR_EL3.NS ||
    (IsFeatureImplemented(FeaT_RME) && MDSCR.EL3.NSPBE != SCR_EL3.NSE) then
        UNDEFINED;
    elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HCFGWTR_EL2.PMBLIMITR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDSCR.EL2.E2PB == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDSCR.EL3.NSPB[0] == '0' || MDSCR.EL3.NSPB[1] != SCR_EL3.NS ||
    (IsFeatureImplemented(FeaT_RME) && MDSCR.EL3.NSPBE != SCR_EL3.NSE) then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NMem[0x800] = X[t, 64];
    else
        PMBLIMITR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDSCR.EL3.NSPB[0] == '0' || MDSCR.EL3.NSPB[1] != SCR_EL3.NS ||
    (IsFeatureImplemented(FeaT_RME) && MDSCR.EL3.NSPBE != SCR_EL3.NSE) then
        UNDEFINED;
    elsif HaveEL(EL3) && MDSCR.EL3.NSPB[0] == '0' || MDSCR.EL3.NSPB[1] != SCR_EL3.NS ||
    (IsFeatureImplemented(FeaT_RME) && MDSCR.EL3.NSPBE != SCR_EL3.NSE) then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMBLIMITR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    PMBLIMITR_EL1 = X[t, 64];
D17.7.3 PMBPTR_EL1, Profiling Buffer Write Pointer Register

The PMBPTR_EL1 characteristics are:

**Purpose**

Defines the current write pointer for the profiling buffer.

**Configurations**

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMBPTR_EL1 are UNDEFINED.

**Attributes**

PMBPTR_EL1 is a 64-bit register.

**Field descriptions**

![Field diagram]

**PTR, bits [63:0]**

Current write address. Defines the virtual address of the next entry to be written to the buffer.

The architecture places restrictions on the values software can write to the pointer. For more information see *Restrictions on the current write pointer* on page D13-5458.

**Note**

As a result, an implementation might treat some of bits[M:0], where M is defined by PMBIDR_EL1.Align, as RES0.

On a management interrupt, PMBPTR_EL1 is frozen.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing PMBPTR_EL1

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, PMBPTR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD = '1'" && MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE) then
    UNDEFINED;
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRT_El2.PMBPTR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
```
elsif EL2Enabled() && MDCR_EL2.E2PB == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
    (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
elsif EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = NVMem[0x810];
else
    X[t, 64] = PMBPTR_EL1;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
        (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
        UNDEFINED;
    elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
        (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = PMBPTR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = PMBPTR_EL1;

**MSR PMBPTR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
        (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
        UNDEFINED;
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMBPTR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.E2PB == 'x0' then
        AArch64.SystemAccess Trap(EL2, 0x18);
    elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
        (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x810] = X[t, 64];
    else
        PMBPTR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
        (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
        UNDEFINED;
    elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
        (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
else
    AArch64.SystemAccessTrap(El3, 0x18);
else
    PMBPTR_EL1 = X[t, 64];
elseif PSTATE.EL == EL3 then
    PMBPTR_EL1 = X[t, 64];
D17.7.4 PMBSR_EL1, Profiling Buffer Status/syndrome Register

The PMBSR_EL1 characteristics are:

**Purpose**

Provides syndrome information to software when the buffer is disabled because the management interrupt has been raised.

**Configurations**

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMBSR_EL1 are UNDEFINED.

**Attributes**

PMBSR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>31:26</td>
<td>EC, Event class. Top-level description of the cause of the buffer management event.</td>
</tr>
<tr>
<td>25:20</td>
<td>RES0</td>
</tr>
<tr>
<td>19:18</td>
<td>DLEA, S</td>
</tr>
<tr>
<td>17:16</td>
<td>MSS</td>
</tr>
<tr>
<td>15:0</td>
<td>COLL</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**EC, bits [31:26]**

Event class. Top-level description of the cause of the buffer management event.

**EC == 0b000000**

Other buffer management event. All buffer management events other than those described by other defined Event class codes.

See MSS encoding for other buffer management events.

**EC == 0b011110**

*When FEAT_RME is implemented:*

Granule Protection Check fault, other than GPF, on write to Profiling Buffer.

See MSS encoding for Granule Protection Check fault.

**EC == 0b011111**

Buffer management event for an IMPLEMENTATION DEFINED reason.

See MSS encoding for a buffer management event for an IMPLEMENTATION DEFINED reason.

**EC == 0b100100**

Stage 1 Data Abort on write to Profiling Buffer.

See MSS encoding for stage 1 or stage 2 Data Aborts on write to buffer.

**EC == 0b100101**

Stage 2 Data Abort on write to Profiling Buffer.

See MSS encoding for stage 1 or stage 2 Data Aborts on write to buffer.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Writing a reserved value to this field will make the value of this field UNKNOWN. Values that are not supported act as reserved values when writing to this register.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [25:20]**

Reserved, RES0.

**DL, bit [19]**

Partial record lost.

Following a buffer management event other than an asynchronous External abort, indicates whether the last record written to the Profiling Buffer is complete.

- 0\(b\)0  PMBPTR_EL1 points to the first byte after the last complete record written to the Profiling Buffer.
- 0\(b\)1  Part of a record was lost because of a buffer management event or synchronous External abort. PMBPTR_EL1 might not point to the first byte after the last complete record written to the buffer, and so restarting collection might result in a data record stream that software cannot parse. All records prior to the last record have been written to the buffer.

When the buffer management event was because of an asynchronous External abort, this bit is set to 1 and software must not assume that any valid data has been written to the Profiling Buffer.

This bit is RES0 if the PE never sets this bit as a result of a buffer management event caused by an asynchronous External abort.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**EA, bit [18]**

External abort.

- 0\(b\)0  An External abort has not been asserted.
- 0\(b\)1  An External abort has been asserted and detected by the Statistical Profiling Unit.

This bit is RES0 if the PE never sets this bit as the result of an External abort.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**S, bit [17]**

Service

- 0\(b\)0  PMBIRQ is not asserted.
- 0\(b\)1  PMBIRQ is asserted. All profiling data has either been written to the buffer or discarded.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**COLL, bit [16]**

Collision detected.

- 0\(b\)0  No collision events detected.
- 0\(b\)1  At least one collision event was recorded.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**MSS, bits [15:0]**

Management Event Specific Syndrome.

Contains syndrome specific to the management event.

The syndrome contents for each management event are described in the following sections.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**MSS encoding for stage 1 or stage 2 Data Aborts on write to buffer**

![](image)

**Bits [15:6]**
- Reserved, RES0.

**FSC, bits [5:0]**
- Fault status code
  - 0b000000 Address size fault, level 0 of translation or translation table base register.
  - 0b000001 Address size fault, level 1.
  - 0b000010 Address size fault, level 2.
  - 0b000011 Address size fault, level 3.
  - 0b000100 Translation fault, level 0.
  - 0b000101 Translation fault, level 1.
  - 0b000110 Translation fault, level 2.
  - 0b000111 Translation fault, level 3.
  - 0b001001 Access flag fault, level 1.
  - 0b001010 Access flag fault, level 2.
  - 0b001011 Access flag fault, level 3.
  - 0b010000 When FEAT_LPA2 is implemented:
    - Access flag fault, level 0.
  - 0b010100 When FEAT_LPA2 is implemented:
    - Synchronous External abort on translation table walk or hardware update of translation table, level -1.
  - 0b010101 Synchronous External abort on translation table walk or hardware update of translation table, level -1.
  - 0b010110 Synchronous External abort on translation table walk or hardware update of translation table, level 0.
  - 0b010111 Synchronous External abort on translation table walk or hardware update of translation table, level 1.
  - 0b011100 Synchronous External abort on translation table walk or hardware update of translation table, level 2.
  - 0b011110 Synchronous External abort on translation table walk or hardware update of translation table, level 3.
  - 0b011111 When FEAT_LPA2 is implemented and FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.

- **0b100001**: Alignment fault.
- **0b100011**: When FEAT_RME is implemented and FEAT_LPA2 is implemented:
  Granule Protection Fault on translation table walk or hardware update of translation table, level -1.
- **0b100100**: When FEAT_RME is implemented:
  Granule Protection Fault on translation table walk or hardware update of translation table, level 0.
- **0b100101**: When FEAT_RME is implemented:
  Granule Protection Fault on translation table walk or hardware update of translation table, level 1.
- **0b100110**: When FEAT_RME is implemented:
  Granule Protection Fault on translation table walk or hardware update of translation table, level 2.
- **0b100111**: When FEAT_RME is implemented:
  Granule Protection Fault on translation table walk or hardware update of translation table, level 3.
- **0b101000**: When FEAT_LPA2 is implemented:
  Granule Protection Fault, not on translation table walk or hardware update of translation table.
- **0b101001**: When FEAT_LPA2 is implemented:
  Address size fault, level -1.
- **0b101011**: When FEAT_LPA2 is implemented:
  Translation fault, level -1.
- **0b110000**: TLB conflict abort.
- **0b110001**: When FEAT_HAFDBS is implemented:
  Unsupported atomic hardware update fault.

All other values are reserved.

It is IMPLEMENTATION DEFINED whether each of the Access Flag fault, asynchronous External abort and synchronous External abort, Alignment fault, and TLB Conflict abort values can be generated by the PE. For more information see Faults and watchpoints on page D13-5464.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### MSS encoding for other buffer management events

![MSS diagram]

**Bits [15:6]**
- Reserved, RES0.

**BSC, bits [5:0]**
- Buffer status code
  - **0b000000**: Buffer not filled
  - **0b000001**: Buffer filled

All other values are reserved. Reserved values might be defined in a future version of the architecture.
Writing a reserved value to this field will make the value of this field UNKNOWN. Values that are not supported act as reserved values when writing to this register.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**MSS encoding for Granule Protection Check fault**

<table>
<thead>
<tr>
<th>Bits [15:0]</th>
<th>Reserved, RES0</th>
</tr>
</thead>
</table>

**MSS encoding for a buffer management event for an IMPLEMENTATION DEFINED reason**

<table>
<thead>
<tr>
<th>Bits [15:0]</th>
<th>IMPLEMENTATION DEFINED</th>
</tr>
</thead>
</table>

**IMPLEMENTATION DEFINED, bits [15:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing PMBSR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xr>, PMBSR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elseif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) && SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMBSR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.E2PB == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.E2PB == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
      X[t, 64] = NVMem[0x820];
    else
      AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  end
else
  AArch64.SystemAccessTrap(EL2, 0x18);
end
X[t, 64] = PMBSR_EL1;
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" & (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elsif HaveEL(EL3) & (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
    (IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
      X[t, 64] = PMBSR_EL1;
  else
    if PSTATE.EL == EL3 then
      X[t, 64] = PMBSR_EL1;
    elsif PSTATE.EL == EL3 then
      MSR PMBSR_EL1, <Xt>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" & (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elseif EL2Enabled() & ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HDFGWR_EL2.PMBSR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
      PMBSR_EL1 = X[t, 64];
  elsif PSTATE.EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1'" & (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
    (IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
      UNDEFINED;
    elseif HaveEL(EL3) & (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
      (IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
      if Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      else
        PMBSR_EL1 = X[t, 64];
      elsif PSTATE.EL == EL3 then
        PMBSR_EL1 = X[t, 64];

D17.7.5 PMSCR_EL1, Statistical Profiling Control Register (EL1)

The PMSCR_EL1 characteristics are:

**Purpose**

Provides EL1 controls for Statistical Profiling.

**Configurations**

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMSCR_EL1 are undefined.

**Attributes**

PMSCR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:8]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[7:6]</td>
<td>PCT, when EL2 is implemented:</td>
</tr>
<tr>
<td>00</td>
<td>Virtual timestamp. The collected timestamp is the physical counter minus the value of CNTVOFF_EL2.</td>
</tr>
<tr>
<td>01</td>
<td>Physical timestamp. The collected timestamp is the physical counter.</td>
</tr>
<tr>
<td>11</td>
<td>When FEAT_ECV is implemented:</td>
</tr>
<tr>
<td></td>
<td>Guest physical timestamp. The collected timestamp is the physical counter minus a physical offset. If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2:</td>
</tr>
<tr>
<td></td>
<td>• SCR_EL3.ECVEn == 0.</td>
</tr>
<tr>
<td></td>
<td>• CNTHCTL_EL2.ECV == 0.</td>
</tr>
</tbody>
</table>

If EL2 is enabled in the current Security state, then the value of PMSCR_EL2.PCT might override or modify the meaning of this field.

This field is ignored by the PE when the Profiling Buffer owning Exception level is EL2.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally unknown value.

Otherwise:

Physical Timestamp. Reserved. This field reads as 0b01 and ignores writes. Software should treat this field as UNK/SBZP.

When EL2 is not implemented, the Effective values of CNTVOFF_EL2 and CNTPOFF_EL2 are zero, meaning the virtual counter and physical counter have the same value.
**TS, bit [5]**

Timestamp enable.

0b0  Timestamp sampling disabled.
0b1  Timestamp sampling enabled.

This bit is ignored by the PE if EL2 is implemented and the Profiling Buffer is owned by EL2. For more information, see *Controlling the data that is collected* on page D13-5455.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**PA, bit [4]**

Physical Address sample enable.

0b0  Physical addresses are not collected.
0b1  Physical addresses are collected.

If EL2 is implemented:

- If the Profiling Buffer is owned by EL1, this bit is combined with PMSCR_EL2.PA to determine which address is collected. For more information, see *Controlling the data that is collected* on page D13-5455.
- If the Profiling Buffer is owned by EL2, this bit is ignored by the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**CX, bit [3]**

CONTEXTIDR_EL1 sample enable.

0b0  CONTEXTIDR_EL1 is not collected.
0b1  CONTEXTIDR_EL1 is collected.

If EL2 is implemented and enabled in the current Security state when an operation is sampled:

- If the PE is at EL2, this bit is ignored by the PE.
- If HCR_EL2.TGE == 1, this bit is ignored by the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [2]**

Reserved, RES0.

**E1SPE, bit [1]**

EL1 Statistical Profiling Enable.

0b0  Sampling disabled at EL1.
0b1  Sampling enabled at EL1.

If EL2 is implemented and enabled in the current Security state, this bit is ignored by the PE when HCR_EL2.TGE == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**E0SPE, bit [0]**

EL0 Statistical Profiling Enable. Controls sampling at EL0 when HCR_EL2.TGE == 0 or if EL2 is disabled or not implemented.

0b0  Sampling disabled at EL0.
0b1  Sampling enabled at EL0.
If EL2 is implemented and enabled in the current Security state, this bit is ignored by the PE when HCR_EL2.TGE = 1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMSCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[ MRS <X>, PMSCR_EL1 \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elseif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMSCR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && MDCR_EL2.TPMS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  elseif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x828];
  else
    X[t, 64] = PMSCR_EL1;
elseif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elseif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  elseif HCR_EL2.E2H == '1' then
    X[t, 64] = PMSCR_EL2;
  else
    X[t, 64] = PMSCR_EL1;
elseif PSTATE.EL == EL3 then
  X[t, 64] = PMSCR_EL1;
MSR PMSCR_EL1,  <xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDSCR_EL3.NSPB[0] == '0' || MDSCR_EL3.NSPB[1] != SCRS_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDSCR_EL3.NSPBE != SCRS_EL3.NSE) then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) || SCRS_EL3.FGTEn == '1') && HDFGWTR_EL2.PMSCR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDSCR_EL2.TPMS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && (MDSCR_EL3.NSPB[0] == '0' || MDSCR_EL3.NSPB[1] != SCRS_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDSCR_EL3.NSPBE != SCRS_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  elsif HCR_EL2.<NV2,NV1,NV> == '111' then
    NVMem[0x828] = X[t, 64];
  else
    PMSCR_EL1 = X[t, 64];
  endif
else
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDSCR_EL3.NSPB[0] == '0' || MDSCR_EL3.NSPB[1] != SCRS_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDSCR_EL3.NSPBE != SCRS_EL3.NSE) then
    UNDEFINED;
  elsif HaveEL(EL3) && (MDSCR_EL3.NSPB[0] == '0' || MDSCR_EL3.NSPB[1] != SCRS_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDSCR_EL3.NSPBE != SCRS_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  elsif HCR_EL2.E2H == '1' then
    PMSCR_EL2 = X[t, 64];
  else
    PMSCR_EL1 = X[t, 64];
  endif
else
  if PSTATE.EL == EL3 then
    PMSCR_EL1 = X[t, 64];
  else
    PMSCR_EL1 = X[t, 64];
  endif
endif

MRS <xt>, PMSCR_EL12

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1001</td>
<td>0b1001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    X[t, 64] = NVMem[0x828];
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
  endif
else
  if PSTATE.EL == EL2 then
    ..
  endif
endif
if HCR_EL2.E2H == '1' then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented( FEAT_RME ) && MDCR_EL3.NSPBE != SCR_EL3.NSE) then UNDEFINED;
elsif HaveEL(EL3) && MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented( FEAT_RME ) && MDCR_EL3.NSPBE != SCR_EL3.NSE) then
  if Halted() && EDSCR.SDD == '1' then UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = PMSCR_EL1;
else
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  if EL2Enabled() && &HCR_EL2.E2H == '1' then
    X[t, 64] = PMSCR_EL1;
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented( FEAT_RME ) && MDCR_EL3.NSPBE != SCR_EL3.NSE) then UNDEFINED;
elsif HaveEL(EL3) && MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented( FEAT_RME ) && MDCR_EL3.NSPBE != SCR_EL3.NSE) then
  if Halted() && EDSCR.SDD == '1' then UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  PMSCR_EL1 = X[t, 64];
else
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  if EL2Enabled() && &HCR_EL2.E2H == '1' then
    PMSCR_EL1 = X[t, 64];
  else
    UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    NVMem[0x828] = X[t, 64];
  elif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented( FEAT_RME ) && MDCR_EL3.NSPBE != SCR_EL3.NSE) then UNDEFINED;
elsif HaveEL(EL3) && MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented( FEAT_RME ) && MDCR_EL3.NSPBE != SCR_EL3.NSE) then
  if Halted() && EDSCR.SDD == '1' then UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  PMSCR_EL1 = X[t, 64];
else
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  if EL2Enabled() && &HCR_EL2.E2H == '1' then
    PMSCR_EL1 = X[t, 64];
  else
    UNDEFINED;
D17.7.6 PMSCR_EL2, Statistical Profiling Control Register (EL2)

The PMSCR_EL2 characteristics are:

**Purpose**

Provides EL2 controls for Statistical Profiling.

**Configurations**

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMSCR_EL2 are UNDEFINED.

**Attributes**

PMSCR_EL2 is a 64-bit register.

**Field descriptions**

Bits [63:8]

Reserved, RES0.

PCT, bits [7:6]

Physical Timestamp. If timestamp sampling is enabled, determines which counter is collected. The behavior depends on the Profiling Buffer owning Exception level.

If FEAT_ECV is implemented, this is a two-bit field as shown. Otherwise, bit[7] is RES0.

0b00 Virtual timestamp. The collected timestamp is the physical counter minus a virtual offset. If any of the following are true, the virtual offset is zero, otherwise the virtual offset is the value of CNTVOFF_EL2:

- The sampled operation executed at EL2 and HCR_EL2.E2H == 1.
- The sampled operation executed at EL0 and HCR_EL2.{E2H,TGE} == {1,1}.

  **Note**

  If the Profiling Buffer owning Exception level is EL1, the virtual offset is always CNTVOFF_EL2.

0b01 If the Profiling Buffer owning Exception level is EL1, then the timestamp value is selected by PMSCR_EL1.PCT.

Otherwise, physical timestamp. The collected timestamp is the physical counter.

0b11 **When FEAT_ECV is implemented:**

If the Profiling Buffer owning Exception level is EL1 and PMSCR_EL1.PCT == 0b00, then guest virtual timestamp. The collected timestamp is the physical counter minus the value of CNTVOFF_EL2.

Otherwise, guest physical timestamp. The collected timestamp is the physical counter minus a physical offset. If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2:

- SCR_EL3.ECVEn == 0.
- CNTHCTL_EL2.ECV == 0.
All other values are reserved.

If EL2 is not implemented or EL2 is disabled in the current Security state, then the Effective value of this field is 0b01, other than for a direct read of the register.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**TS, bit [5]**

Timestamp Enable.

0b0 Timestamp sampling disabled.
0b1 Timestamp sampling enabled.

This bit is ignored by the PE when any of the following are true:
• The Profiling Buffer owning Exception level is EL1.
• In Secure state, and either FEAT_SEL2 is not implemented or Secure EL2 is disabled.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**PA, bit [4]**

Physical Address Sample Enable.

0b0 Physical addresses are not collected.
0b1 Physical addresses are collected.

If the Profiling Buffer owning Exception level is EL1, and EL2 is enabled in the current Security state, this bit is combined with PMSCR_EL1.PA to determine which address is collected.

If EL2 is not implemented or EL2 is disabled in the current Security state, the PE ignores the value of this bit and behaves as if this bit is set to 1, other than for a direct read of the register.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**CX, bit [3]**

CONTEXTIDR_EL2 Sample Enable.

0b0 CONTEXTIDR_EL2 is not collected.
0b1 CONTEXTIDR_EL2 is collected.

If EL2 is not implemented or EL2 is disabled in the current Security state, the PE ignores the value of this bit.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [2]**

Reserved, RES0.

**E2SPE, bit [1]**

EL2 Statistical Profiling Enable.

0b0 Sampling disabled at EL2.
0b1 Sampling enabled at EL2.

This bit is RES0 if MDCR_EL2.E2PB != 0b00.

If EL2 is disabled in the current Security state, this bit is ignored by the PE.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
E0HSPE, bit [0]

EL0 Statistical Profiling Enable.

0b0  Sampling disabled at EL0.
0b1  Sampling enabled at EL0.

If MDCR_EL2.E2PB != 0b00, this bit is RES0.

If EL2 is implemented and enabled in the current Security state, this bit is ignored by the PE when HCR_EL2.TGE == 0.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMSCR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, PMSCR_EL2

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b1001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

MSR PMSCR_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b1001</td>
<td>0b000</td>
</tr>
</tbody>
</table>
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    PMSCR_EL2 = X[t, 64];
  end
elsif PSTATE_EL == EL3 then
  PMSCR_EL2 = X[t, 64];

MRS <Xt>, PMSCR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elsif EL2Enabled() && (SCF_EL3.FGTEn == '1') && HDFGRTR_EL2.PMSCR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.TPMS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x828];
  else
    X[t, 64] = PMSCR_EL1;
  end
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  elsif HCR_EL2.E2H == '1' then
    X[t, 64] = PMSCR_EL2;
  else
    X[t, 64] = PMSCR_EL1;
  end
elsif PSTATE_EL == EL3 then
  X[t, 64] = PMSCR_EL1;
### MSR PMSCR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMSCR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    PMSCR_EL1 = X[t, 64];
  endif
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    NVMem[0x828] = X[t, 64];
  else
    PMSCR_EL1 = X[t, 64];
  endif
elsif PSTATE_EL == EL3 then
  PMSCR_EL1 = X[t, 64];
else
  PMSCR_EL1 = X[t, 64];
"
D17.7.7 PMSEVFR_EL1, Sampling Event Filter Register

The PMSEVFR_EL1 characteristics are:

**Purpose**

Controls sample filtering by events. The overall filter is the logical AND of these filters. For example, if PMSEVFR_EL1.E[3] and PMSEVFR_EL1.E[5] are both set to 1, only samples that have both event 3 (Level 1 unified or data cache refill) and event 5 (TLB walk) set to 1 are recorded.

**Configurations**

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMSEVFR_EL1 are UNDEFINED.

**Attributes**

PMSEVFR_EL1 is a 64-bit register.

**Field descriptions**

E[<x>], bit [x], for x = 63 to 48, 31 to 24, 15 to 12

E[<x>] is the event filter for event <x>. If event <x> is not implemented, or filtering on event <x> is not supported, the corresponding bit is RAZ/WI.

0b0  Event <x> is ignored.
0b1  Do not record samples that have event <x> == 0.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, if the corresponding bits of PMSEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event. This field is ignored by the PE when PMSFCR_EL1.FE == 0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [47:32]**

Reserved, RAZ/WI.
Bits [23:19]  
Reserved, RAZ/WI.

E[18], bit [18]  
When FEAT_SPEv1p1 is implemented and FEAT_SVE is implemented:
Empty predicate.  
0b0 Empty predicate event is ignored.  
0b1 Do not record samples that have the Empty predicate event == 0.  
This bit is ignored by the PE when PMSFCR_EL1.FE == 0.  
The reset behavior of this field is:  
• On a Warm reset, this field resets to an architecturally UNKNOWN value.  
Otherwise:  
Reserved, RAZ/WI.

E[17], bit [17]  
When FEAT_SPEv1p1 is implemented and FEAT_SVE is implemented:
Partial predicate.  
0b0 Partial predicate event is ignored.  
0b1 Do not record samples that have the Partial predicate event == 0.  
This bit is ignored by the PE when PMSFCR_EL1.FE == 0.  
The reset behavior of this field is:  
• On a Warm reset, this field resets to an architecturally UNKNOWN value.  
Otherwise:  
Reserved, RAZ/WI.

E[16], bit [16]  
When FEAT_TME is implemented:  
Transactional  
0b0 Transactional event is ignored.  
0b1 Do not record samples that have the Transactional event == 0.  
This bit is ignored by the PE when PMSFCR_EL1.FE == 0.  
The reset behavior of this field is:  
• On a Warm reset, this field resets to an architecturally UNKNOWN value.  
Otherwise:  
Reserved, RAZ/WI.

When FEAT_SPEv1p1 is implemented:  
Alignment.  
0b0 Alignment event is ignored.  
0b1 Do not record samples that have the Alignment event == 0.  
This bit is ignored by the PE when PMSFCR_EL1.FE == 0.  
The reset behavior of this field is:  
• On a Warm reset, this field resets to an architecturally UNKNOWN value.  
Otherwise:  
Reserved, RAZ/WI.
Bits [10:8]
Reserved, RAZ/WI.

E[7], bit [7]
Mispredicted.
0b0 Mispredicted event is ignored.
0b1 Do not record samples that have the Mispredicted event == 0.
This bit is ignored by the PE when PMSFCR_EL1.FE == 0.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

E[6], bit [6]
When FEAT_SPEv1p2 is implemented:
Not taken.
0b0 Not taken event is ignored.
0b1 Do not record samples that have the Not taken event == 0.
This field is ignored by the PE when PMSFCR_EL1.FE == 0.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Otherwise:
Reserved, RAZ/WI.

E[5], bit [5]
TLB walk.
0b0 TLB walk event is ignored.
0b1 Do not record samples that have the TLB walk event == 0.
This bit is ignored by the PE when PMSFCR_EL1.FE == 0.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [4]
Reserved, RAZ/WI.

E[3], bit [3]
Level 1 data or unified cache refill.
0b0 Level 1 data or unified cache refill event is ignored.
0b1 Do not record samples that have the Level 1 data or unified cache refill event == 0.
This bit is ignored by the PE when PMSFCR_EL1.FE == 0.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [2]
Reserved, RAZ/WI.

E[1], bit [1]
When the PE supports sampling of speculative instructions:
Architecturally executed.
When the PE supports sampling of speculative instructions:
0b0 Architecturally executed event is ignored.
0b1  Do not record samples that have the Architecturally executed event == 0.
This bit is ignored by the PE when PMSFCR_EL1.FE == 0.

If the PE does not support the sampling of speculative instructions, or always discards the sample
record for speculative instructions, this bit reads as an UNKNOWN value and the PE ignores its value.
The reset behavior of this field is:
•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, UNKNOWN.

**Bit [0]**
Reserved, RAZ/WI.

**Accessing PMSEVFR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS \( \langle X_t \rangle, \text{PMSEVFR_EL1} \)**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1001</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
    (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  else
    if Halted() && HaveEL(EL3) && SCR_EL3.FGTEn == '1' && HDFGRTR_EL2.PMSEVFR_EL1 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif Halted() && MDCR_EL2.TPMS == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
      (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        X[t, 64] = PMSEVFR_EL1;
      end if
    else
    end if
  end if
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
    (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  else
    if HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
      (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end if
    else
    end if
  end if
elsif PSTATE.EL == EL3 then
  X[t, 64] = PMSEVFR_EL1;
else
  X[t, 64] = PMSEVFR_EL1;
end if
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMSEVFR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && MDCR_EL2.TPMS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV> == '1' then
      NVMem[0x830] = X[t, 64];
    else
      PMSEVFR_EL1 = X[t, 64];
  else
    PMSEVFR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
  else
    PMSEVFR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
  PMSEVFR_EL1 = X[t, 64];
D17.7.8 PMSFCR_EL1, Sampling Filter Control Register

The PMSFCR_EL1 characteristics are:

**Purpose**

Controls sample filtering. The filter is the logical AND of the FL, FT and FE bits. For example, if FE == 1 and FT == 1 only samples including the selected operation types and the selected events will be recorded.

**Configurations**

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMSFCR_EL1 are **UNDEFINED**.

**Attributes**

PMSFCR_EL1 is a 64-bit register.

**Field descriptions**

![Field Diagram]

**Bits [63:19]**

Reserved, RES0.

**ST, bit [18]**

Store filter enable

- **0b0** Do not record store operations
- **0b1** Record all store operations, including vector stores and all atomic operations

This bit is ignored by the PE when PMSFCR_EL1.FT == 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**LD, bit [17]**

Load filter enable

- **0b0** Do not record load operations
- **0b1** Record all load operations, including vector loads and atomic operations that return data

This bit is ignored by the PE when PMSFCR_EL1.FT == 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**B, bit [16]**

Branch filter enable

- **0b0** Do not record branch and exception return operations
- **0b1** Record all branch and exception return operations

This bit is ignored by the PE when PMSFCR_EL1.FT == 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
Bits [15:4]
Reserved, RES0.

FnE, bit [3]

When FEAT_SPEv1p2 is implemented:
Filter by event, inverted.
0b0 Inverted event filtering disabled.
0b1 Inverted event filtering enabled. Samples including the events selected by PMSNEVFR_EL1 will not be recorded.

If any of the following are true, it is CONSTRAINED UNPREDICTABLE whether no samples are recorded or the PE behaves as if PMSFCR_EL1.FnE == 0:
• PMSFCR_EL1.FnE == 1 and PMSNEVFR_EL1 is zero.
• PMSFCR_EL1.FnE == 1, PMSFCR_EL1.FE == 1, and there exists a value x such that PMSFELV_EL1.E[x] == 1 and PMSNEVFR_EL1.E[x] == 1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

FL, bit [2]
Filter by latency
0b0 Latency filtering disabled
0b1 Latency filtering enabled. Samples with a total latency less than PMSLATFR_EL1.MINLAT will not be recorded

If this field is set to 1 and PMSLATFR_EL1.MINLAT is set to zero, it is CONSTRAINED UNPREDICTABLE whether no samples are recorded or the PE behaves as if PMSFCR_EL1.FL is set to 0

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

FT, bit [1]
Filter by operation type. The filter is the logical OR of the ST, LD and B bits. For example, if LD and ST are both set, both load and store operations are recorded
0b0 Type filtering disabled
0b1 Type filtering enabled. Samples not one of the selected operation types will not be recorded

If this field is set to 1 and the PMSFCR_EL1.{ST, LD, B} bits are all set to zero, it is CONSTRAINED UNPREDICTABLE whether no samples are recorded or the PE behaves as if PMSFCR_EL1.FT is set to 0

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

FE, bit [0]
Filter by event.
0b0 Event filtering disabled.
0b1 Event filtering enabled. Samples not including the events selected by PMSEVFR_EL1 will not be recorded.

If any of the following are true, it is CONSTRAINED UNPREDICTABLE whether no samples are recorded or the PE behaves as if PMSFCR_EL1.FE == 0:
• PMSFCR_EL1.FE == 1 and PMSEVFR_EL1 is zero.
• FEAT_SPEv1p2 is implemented, PMSEVR_EL1.FnE = 1, PMSEVR_EL1.FE = 1, and there exists a value \( x \) such that PMSEVF_EL1.E[\( x \)] = 1 and PMSEVF_EL1.E[\( x \)] = 1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMSFCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

**MRS <xt>, PMSFCR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1001</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
        (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE) then
        UNDEFINED;
    elsif EL2Enabled() && (!HaveEL(EL3) || HDFGRTR_EL2.PMSFCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
        (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE) then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[\( t, 64 \)] = PMSEVF_EL1;
    elsif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
            (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE) then
            UNDEFINED;
        elsif HaveEL(EL3) && MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
            (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE) then
                if Halted() && EDSCR.SDD == '1' then
                    UNDEFINED;
                else
                    AArch64.SystemAccessTrap(EL3, 0x18);
                else
                    X[\( t, 64 \)] = PMSEVF_EL1;
        elsif PSTATE.EL == EL3 then
            X[\( t, 64 \)] = PMSEVF_EL1;

**MSR PMSEFR_EL1, <xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1001</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
        UNDEFINED;
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMSFCR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        if PSTATE.EL == EL1 then
            if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end
        else
            if PSTATE.EL == EL2 then
                if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
                    UNDEFINED;
                else
                    AArch64.SystemAccessTrap(EL3, 0x18);
                end
            else
                if PSTATE.EL == EL3 then
                    PMSFCR_EL1 = X[t, 64];
                else
                    if PSTATE.EL == EL2 then
                        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
                            UNDEFINED;
                        else
                            AArch64.SystemAccessTrap(EL3, 0x18);
                        end
                    else
                        PMSFCR_EL1 = X[t, 64];
                    end
                else
                    PMSFCR_EL1 = X[t, 64];
                end
            end
        end
    end
end
### D17.7.9 PMSICR_EL1, Sampling Interval Counter Register

The PMSICR_EL1 characteristics are:

**Purpose**

Software must write zero to PMSICR_EL1 before enabling sample profiling for a sampling session. Software must then treat PMSICR_EL1 as an opaque, 64-bit, read/write register used for context switches only.

**Configurations**

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMSICR_EL1 are **undefined**.

The value of PMSICR_EL1 does not change whilst profiling is disabled.

**Attributes**

PMSICR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-56</td>
<td>ECOUNT</td>
</tr>
<tr>
<td>55-32</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>31-0</td>
<td>COUNT</td>
</tr>
</tbody>
</table>

**ECOUNT, bits [63:56]**

*When PMSIDR_EL1.ERnd == 1:*

Secondary sample interval counter.

This field provides the secondary counter used after the primary counter reaches zero. Whilst the secondary counter is nonzero and profiling is enabled, the secondary counter decrements by 1 for each member of the sample population. The primary counter also continues to decrement since it is also nonzero. When the secondary counter reaches zero, a member of the sampling population is selected for sampling.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**

Reserved, RES0.

**Bits [55:32]**

Reserved, RES0.

**COUNT, bits [31:0]**

Primary sample interval counter

Provides the primary counter used for sampling.

The primary counter is reloaded when the value of this register is zero and the PE moves from a state or Exception level where profiling is disabled to a state or Exception level where profiling is enabled. Whilst the primary counter is nonzero and sampling is enabled, the primary counter decrements by 1 for each member of the sample population.

When the counter reaches zero, the behavior depends on the values of PMSIDR_EL1.ERnd and PMSIRR_EL1.RND

*If PMSIRR_EL1.RND == 0 or PMSIDR_EL1.ERnd == 0:*

- A member of the sampling population is selected for sampling
The primary counter is reloaded

- If PMSIRR_EL1.RND == 1 and PMSIDR_EL1.ERnd == 1:
  - The secondary counter is set to a random or pseudorandom value in the range 0x00 to 0xFF
  - The primary counter is reloaded

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing PMSICR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, PMSICR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR_SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & (MDCR_EL3.NSPB[0] == '0' || (MDCR_EL3.NSPB[1] != SCR_EL3.NS || !IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elseif EL2Enabled() & !(HaveEL(EL3) || SCR_EL3.FGETn == '1') & HDIFGRTR_EL2.PMSICR_EL1 == '1' then
    AArch64_SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() & MDCR_EL2.TPMS == '1' then
    AArch64_SystemAccessTrap(EL2, 0x18);
  elseif HaveEL(EL3) & (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || !IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() & EDSCR_SDD == '1' then
      UNDEFINED;
    else
      AArch64_SystemAccessTrap(EL3, 0x18);
    end if
  elseif EL2Enabled() & HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = NVMem[0x838];
  else
    X[t, 64] = PMSICR_EL1;
  end if
elsif PSTATE_EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR_SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & (MDCR_EL3.NSPB[0] == '0' || (MDCR_EL3.NSPB[1] != SCR_EL3.NS || !IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elseif HaveEL(EL3) & (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || !IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() & EDSCR_SDD == '1' then
      UNDEFINED;
    else
      AArch64_SystemAccessTrap(EL3, 0x18);
    end if
  elseif PSTATE_EL == EL3 then
    X[t, 64] = PMSICR_EL1;
else
  X[t, 64] = PMSICR_EL1;
end if
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  else
    EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMSICR_EL1 == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && MDCR_EL2.TPMS == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    elseif EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
      NVMem[0x838] = X[t, 64];
    else
      PMSICR_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elseif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    PMSICR_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL3 then
  PMSICR_EL1 = X[t, 64];
D17.7.10 PMSIDR_EL1, Sampling Profiling ID Register

The PMSIDR_EL1 characteristics are:

**Purpose**
Describes the Statistical Profiling implementation to software

**Configurations**
This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMSIDR_EL1 are UNDEFINED.

**Attributes**
PMSIDR_EL1 is a 64-bit register.

**Field descriptions**

![Field diagram]

Bits [63:25]
Reserved, RES0.

PBT, bit [24]
Previous branch target Address packet. Defined values are:
- 0b0: Previous branch target Address packet not supported.
- 0b1: Previous branch target Address packet support implemented.

FEAT_SPEv1p2 adds the OPTIONAL functionality identified by the value 1.

Format, bits [23:20]

*From Armv8.7:*
Defines the format of the sample records. Defined values are:
- 0b0000: Format 0.

All other values are reserved.

*Otherwise:*
Reserved, RAZ.

CountSize, bits [19:16]
 Defines the size of the counters.
- 0b0010: 12-bit saturating counters.
- 0b0011: From Armv8.8:
  16-bit saturating counters.

All other values are reserved.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.
MaxSize, bits [15:12]

Defines the largest size for a single record, rounded up to a power-of-two. If this is the same as the minimum alignment (PMBIDR_EL1.Align), then each record is exactly this size. Defined values are:

- 0b0100 16 bytes.
- 0b0101 32 bytes.
- 0b0110 64 bytes.
- 0b0111 128 bytes.
- 0b1000 256 bytes.
- 0b1001 512 bytes.
- 0b1010 1KB.
- 0b1011 2KB.

All other values are reserved.

The values 0b0100 and 0b0101 are not permitted for an implementation.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Interval, bits [11:8]

Recommended minimum sampling interval. This provides guidance from the implementer to the smallest minimum sampling interval, N. Defined values are:

- 0b0000 256.
- 0b0010 512.
- 0b0011 768.
- 0b0100 1,024.
- 0b0101 1,536.
- 0b0110 2,048.
- 0b0111 3,072.
- 0b1000 4,096.

All other values are reserved.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [7]

Reserved, RES0.

FnE, bit [6]

Filtering by events, inverted. Defined values are:

- 0b0 PMSNEVFR_EL1 is not implemented and PMSFCR_EL1.FnE is RES0.
- 0b1 PMSNEVFR_EL1 and PMSFCR_EL1.FnE are implemented.

FEAT_SPEv1p2 adds the functionality identified by the value 1.

ERnd, bit [5]

Defines how the random number generator is used in determining the interval between samples, when enabled by PMSIRR_EL1.RND. Defined values are:

- 0b0 The random number is added at the start of the interval, and the sample is taken and a new interval started when the combined interval expires.
- 0b1 The random number is added and the new interval started after the interval programmed in PMSIRR_EL1.INTERVAL expires, and the sample is taken when the random interval expires.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

**LDS, bit [4]**

Data source indicator for sampled load instructions. Defined values are:

- 0b0  Loaded data source not implemented.
- 0b1  Loaded data source implemented.

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

**ArchInst, bit [3]**

Architectural instruction profiling. Defined values are:

- 0b0  Micro-op sampling implemented.
- 0b1  Architecture instruction sampling implemented.

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

**FL, bit [2]**

Filtering by latency. This bit is RAO.

**FT, bit [1]**

Filtering by operation type. This bit is RAO.

**FE, bit [0]**

Filtering by events. This bit is RAO.

### Accessing PMSIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, PMSIDR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1001</td>
<td>0b111</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
    (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMSIDR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.TPMS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
    (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = PMSIDR_EL1;
  end
elsif PSTATE.EL == EL2 then
```

D17-6898  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
  UNDEFINED;
else if HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    X[t, 64] = PMSIDR_EL1;
else if PSTATE.EL == EL3 then
  X[t, 64] = PMSIDR_EL1;
D17.7.11   PMSIRR_EL1, Sampling Interval Reload Register

The PMSIRR_EL1 characteristics are:

Purpose
Defines the interval between samples.

Configurations
This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMSIRR_EL1 are UNDEFINED.

Attributes
PMSIRR_EL1 is a 64-bit register.

Field descriptions

Bits [63:32]
Reserved, RES0.

INTERVAL, bits [31:8]
Bits [31:8] of the PMSICR_EL1 interval counter reload value. Software must set this to a non-zero value. If software sets this to zero, an UNKNOWN sampling interval is used. Software should set this to a value greater than the minimum indicated by PMSIDR_EL1.Interval.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [7:1]
Reserved, RES0.

RND, bit [0]
Controls randomization of the sampling interval.

0b0  Disable randomization of sampling interval.
0b1  Add (pseudo-)random jitter to sampling interval.

The random number generator is not architected.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMSIRR_EL1

Accesses to this register use the following encodings in the System register encoding space:
**MRS <Xt>, PMSIRR_EL1**

```plaintext
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGTR_EL2.PMSIRR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && MDCR_EL2.TPMS == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
  (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  elsif EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = NVMem[0x840];
  else
    X[t, 64] = PMSIRR_EL1;
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
    (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSBE != SCR_EL3.NSE)) then
      UNDEFINED;
    elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
    (IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSBE != SCR_EL3.NSE)) then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    elsif PSTATE.EL == EL3 then
      X[t, 64] = PMSIRR_EL1;
    else
      X[t, 64] = PMSIRR_EL1;
  elsif PSTATE.EL == EL3 then
    X[t, 64] = PMSIRR_EL1;
```
(IsFeatureImplemented(FEAT_RME) && MDCCR_EL3.NSPBE != SCR_EL3.NSE) then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  elsif EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
    NVMem[0x840] = X[t, 64];
  else
    PMSIRR_EL1 = X[t, 64];
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority"
      when SDD == '1'" && (MDCCR_EL3.NSPB[0] == '0' || MDCCR_EL3.NSPB[1] != SCR_EL3.NS ||
      (IsFeatureImplemented(FEAT_RME) && MDCCR_EL3.NSPBE != SCR_EL3.NSE) then
        UNDEFINED;
      elseif Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
      PMSIRR_EL1 = X[t, 64];
  elseif PSTATE.EL == EL3 then
    PMSIRR_EL1 = X[t, 64];
  else
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if;
  end if;
end if;
D17.7.12 PMSLATFR_EL1, Sampling Latency Filter Register

The PMSLATFR_EL1 characteristics are:

**Purpose**

Controls sample filtering by latency

**Configurations**

This register is present only when FEAT_SPE is implemented. Otherwise, direct accesses to PMSLATFR_EL1 are UNDEFINED.

**Attributes**

PMSLATFR_EL1 is a 64-bit register.

**Field descriptions**

**Bits [63:16]**

Reserved, RES0.

**MINLAT, bits [15:0]**

Minimum latency. When PMSFCR_EL1.FL is 1, defines the minimum total latency for filtered operations. Samples with a total latency less than PMSLATFR_EL1.MINLAT are not recorded.

If PMSIDR_EL1.CountSize is 0b0010, PMSLATFR_EL1.MINLAT[15:12] is RES0.

This field is ignored by the PE when PMSFCR_EL1.FL == 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing PMSLATFR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, PMSLATFR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1001</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(REALM) && MDCR_EL3.NSPB != SCR_EL3.NSE) then
        UNDEFINED;
    elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMSLATFR_EL1 == '1') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && MDCR_EL2.TPMS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
**AArch64 System Register Descriptions**

**D17.7 Statistical Profiling Extension registers**

```c
elsif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
else
  EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elseif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
else
  X[t, 64] = PMSLATFR_EL1;
else
  PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
      UNDEFINED;
    elseif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      endif
else
  X[t, 64] = PMSLATFR_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = PMSLATFR_EL1;
endif
```

**MSR PMSLATFR_EL1, <xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1001</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elseif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
else
  EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    UNDEFINED;
  elseif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
else
  PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
      UNDEFINED;
    elseif HaveEL(EL3) && (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS ||
(IsFeatureImplemented(FEAT_RME) && MDCR_EL3.NSPBE != SCR_EL3.NSE)) then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      endif
else
  X[t, 64] = PMSLATFR_EL1;
else
  PSTATE.EL == EL3 then
  X[t, 64] = PMSLATFR_EL1;
endif
else
    PMSLATFR_EL1 = X[t, 64];
elsif PSTATE_EL == EL3 then
    PMSLATFR_EL1 = X[t, 64];
D17.7.13 PMSNEVFR_EL1, Sampling Inverted Event Filter Register

The PMSNEVFR_EL1 characteristics are:

**Purpose**

Controls sample filtering by events. The overall inverted filter is the logical OR of these filters. For example, if PMSNEVFR_EL1.E[3] and PMSNEVFR_EL1.E[5] are both set to 1, samples that have either event 3 (Level 1 unified or data cache refill) or event 5 (TLB walk) set to 1 are not recorded.

**Configurations**

This register is present only when FEAT_SPEv1p2 is implemented. Otherwise, direct accesses to PMSNEVFR_EL1 are UNDEFINED.

**Attributes**

PMSNEVFR_EL1 is a 64-bit register.

**Field descriptions**

E[<x>], bit [x], for x = 63 to 48, 31 to 24, 15 to 12

E[<x>] is the event filter for IMPLEMENTATION DEFINED event <x>.

0b0 Event <x> is ignored.

0b1 Do not record samples that have event <x> == 1.

An IMPLEMENTATION DEFINED event might be recorded as a multi-bit field. In this case, the corresponding bits of PMSNEVFR_EL1 define an IMPLEMENTATION DEFINED filter for the event.

This bit is ignored by the PE when PMSFCR_EL1.FnE == 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When event <x> is not implemented, or filtering on event <x> is not supported, access to this field is RAZ/WI.
Bits [47:32]  
Reserved, RAZ/WI.

Bits [23:19]  
Reserved, RAZ/WI.

E[18], bit [18]  
*When FEAT_SVE is implemented and FEAT_SPEv1p1 is implemented:*
  
  Not empty predicate.
  
  *0b0*  
  Empty predicate event is ignored.
  
  *0b1*  
  Do not record samples that have the Empty predicate event == 1.
  
  This field is ignored by the PE when PMSFCR_EL1.FnE == 0.
  
  The reset behavior of this field is:
  
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RAZ/WI.

E[17], bit [17]  
*When FEAT_SVE is implemented and FEAT_SPEv1p1 is implemented:*
  
  Not partial predicate.
  
  *0b0*  
  Partial predicate event is ignored.
  
  *0b1*  
  Do not record samples that have the Partial predicate event == 1.
  
  This field is ignored by the PE when PMSFCR_EL1.FnE == 0.
  
  The reset behavior of this field is:
  
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RAZ/WI.

E[16], bit [16]  
*When FEAT_TME is implemented:*
  
  Not transactional.
  
  *0b0*  
  Transactional event is ignored.
  
  *0b1*  
  Do not record samples that have the Transactional event == 1.
  
  This field is ignored by the PE when PMSFCR_EL1.FnE == 0.
  
  The reset behavior of this field is:
  
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RAZ/WI.

*When FEAT_SPEv1p1 is implemented:*
  
  Aligned.
  
  *0b0*  
  Misalignment event is ignored.
  
  *0b1*  
  Do not record samples that have the Misalignment event == 1.
  
  This field is ignored by the PE when PMSFCR_EL1.FnE == 0.
  
  The reset behavior of this field is:
  
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.
**Otherwise:**

Reserved, RAZ/WI.

**Bits [10:8]**

Reserved, RAZ/WI.

**E[7], bit [7]**

Correctly predicted.

0b0 Mispredicted event is ignored.

0b1 Do not record samples that have the Mispredicted event == 1.

This field is ignored by the PE when PMSFCR_EL1.FnE == 0.

The reset behavior of this field is:

* On a Warm reset, this field resets to an architecturally UNKNOWN value.

**E[6], bit [6]**

Taken.

0b0 Not taken event is ignored.

0b1 Do not record samples that have the Not taken event == 1.

This field is ignored by the PE when PMSFCR_EL1.FnE == 0.

The reset behavior of this field is:

* On a Warm reset, this field resets to an architecturally UNKNOWN value.

**E[5], bit [5]**

TLB hit.

0b0 TLB walk event is ignored.

0b1 Do not record samples that have the TLB walk event == 1.

This field is ignored by the PE when PMSFCR_EL1.FnE == 0.

The reset behavior of this field is:

* On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [4]**

Reserved, RAZ/WI.

**E[3], bit [3]**

Level 1 data or unified cache hit.

0b0 Level 1 data or unified cache refill event is ignored.

0b1 Do not record samples that have the Level 1 data or unified cache refill event == 1.

This field is ignored by the PE when PMSFCR_EL1.FnE == 0.

The reset behavior of this field is:

* On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [2]**

Reserved, RAZ/WI.

**E[1], bit [1]**

When the PE supports sampling of speculative instructions:

Speculative.

0b0 Architecturally executed event is ignored.

0b1 Do not record samples that have the Architecturally executed event == 1.

This field is ignored by the PE when PMSFCR_EL1.FnE == 0.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RAZ/WI.

**Bit [0]**

Reserved, RAZ/WI.

**Accessing PMSNEVFR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, PMSNEVFR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```latex
\textbf{if} PSTATE.EL == EL0 \textbf{then} \\
\quad \textbf{UNDEFINED;}
\textbf{elsif} PSTATE.EL == EL1 \textbf{then} \\
\quad \textbf{if} Halted() \&\& HaveEL(EL3) \&\& EDSCR.SOD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' & MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) \&\& MDCR_EL3.NSPBE != SCR_EL3.NSE) \textbf{then} \\
\quad \quad \textbf{UNDEFINED;}
\textbf{elsif} Halted() \&\& HaveEL(EL3) \&\& EDSCR.SOD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' \& MDCR_EL3.EnPSN == '0' \textbf{then} \\
\quad \quad \textbf{UNDEFINED;}
\textbf{elsif} EL2Enabled() \&\& (HaveEL(EL3) || SCR_EL3.FGETn == '1') \&\& HDFGRTR_EL2.nPMSNEVFR_EL1 == '0' \textbf{then} \\
\quad \quad \text{AArch64.SystemAccessTrap(EL2, 0x18);}
\textbf{elsif} EL2Enabled() \&\& MDCR_EL2.TPMS == '1' \textbf{then} \\
\quad \quad \text{AArch64.SystemAccessTrap(EL2, 0x18);}
\textbf{elsif} HaveEL(EL3) \&\& (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) \&\& MDCR_EL3.NSPBE != SCR_EL3.NSE) \textbf{then} \\
\quad \quad \textbf{if} Halted() \&\& EDSCR.SOD == '1' \textbf{then} \\
\quad \quad \quad \textbf{UNDEFINED;}
\textbf{elsif} Halted() \&\& EL2Enabled() \&\& HCR_EL2.<NV2,NV> == '11' \textbf{then} \\
\quad \quad \quad \textbf{X[t, 64] = NVMem[0x850];}
\textbf{else} \\
\quad \quad \quad \text{AArch64.SystemAccessTrap(EL3, 0x18);}
\textbf{elsif} HaveEL(EL3) \&\& MDCR_EL3.EnPSN == '0' \textbf{then} \\
\quad \quad \textbf{UNDEFINED;}
\textbf{else} \\
\quad \quad \text{AArch64.SystemAccessTrap(EL3, 0x18);}
\textbf{elsif} EL2Enabled() \&\& EL3Enabled() \&\& MDCR_EL3.TPMS == '1' \textbf{then} \\
\quad \quad \textbf{AArch64.SystemAccessTrap(EL3, 0x18);}
\textbf{elsif} EL2Enabled() \&\& HCR_EL2.<NV2,NV> == '11' \textbf{then} \\
\quad \quad \textbf{X[t, 64] = PMSNEVFR_EL1;}
\textbf{else} \\
\quad \quad \text{PSTATE.EL == EL2 \textbf{then} \\
\quad \quad \textbf{if} Halted() \&\& HaveEL(EL3) \&\& EDSCR.SOD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' \& MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) \&\& MDCR_EL3.NSPBE != SCR_EL3.NSE) \textbf{then} \\
\quad \quad \quad \textbf{UNDEFINED;}
\textbf{elsif} Halted() \&\& HaveEL(EL3) \&\& EDSCR.SOD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1' \& MDCR_EL3.EnPSN == '0' \textbf{then} \\
\quad \quad \quad \textbf{UNDEFINED;}
\textbf{elsif} HaveEL(EL3) \&\& (MDCR_EL3.NSPB[0] == '0' || MDCR_EL3.NSPB[1] != SCR_EL3.NS || (IsFeatureImplemented(FEAT_RME) \&\& MDCR_EL3.NSPBE != SCR_EL3.NSE) \textbf{then} \\
\quad \quad \quad \textbf{if} Halted() \&\& EDSCR.SOD == '1' \textbf{then} \\
\quad \quad \quad \quad \textbf{UNDEFINED;}
\textbf{elsif} Halted() \&\& EL2Enabled() \&\& MDCR_EL3.EnPSN == '0' \textbf{then} \\
\quad \quad \quad \quad \textbf{AArch64.SystemAccessTrap(EL3, 0x18);}
\textbf{elsif} HaveEL(EL3) \&\& MDCR_EL3.EnPSN == '0' \textbf{then} \\
\quad \quad \quad \quad \textbf{UNDEFINED;}
\textbf{else} \\
\quad \quad \quad \quad \text{AArch64.SystemAccessTrap(EL3, 0x18);}
\textbf{elsif} HaveEL(EL3) \&\& MDCR_EL3.EnPSN == '0' \textbf{then} \\
```

AArch64 System Register Descriptions
D17.7 Statistical Profiling Extension registers

Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential
if Halted() & EDSCR.SDD == '1' then
  UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = PMSNEVFR_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = PMSNEVFR_EL1;

**MSR PMSNEVFR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.NSPB[0] == '0' & MDCR_EL3.NSPB[1] != SCR_EL3.NS & (IsFeatureImplemented(FEAT_RME) & MDCR_EL3.NSPBE != SCR_EL3.NSE) then UNDEFINED;
elsif HaveEL(EL3) & MDHR_EL1.FRTE == '1' & HDFGWTR_EL2.nPMSNEVFR_EL1 == '0' then AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & MDHR_EL1.TMPS == '1' then AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & MDHR_EL1.FGTE == '1' & HDFGWTR_EL2.nPMSNEVFR_EL1 == '0' & MDHR_EL1.FGTE == '1' then AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & MDHR_EL1.FGTE == '1' & HDFGWTR_EL2.nPMSNEVFR_EL1 == '0' then AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) & MDHR_EL1.FGTE == '1' then HDFGWTR_EL2.nPMSNEVFR_EL1 == '0' & MDHR_EL1.FGTE == '1' then AArch64.SystemAccessTrap(EL2, 0x18);
else if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
elsif HaveEL(EL3) & MDHR_EL3.EnPMSE == '0' then if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
elsif HaveEL(EL3) & MDHR_EL3.EnPMSE == '0' then if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
elsif HaveEL(EL3) & MDHR_EL3.EnPMSE == '0' then if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
PMSNEVFR_EL1 = X[t, 64];
elsif PSTATE_EL == EL3 then
  PMSNEVFR_EL1 = X[t, 64];
D17.8 Branch Record Buffer Extension registers

This section lists the Branch Record Buffer registers in AArch64.
D17.8.1   BRBCR_EL1, Branch Record Buffer Control Register (EL1)

The BRBCR_EL1 characteristics are:

Purpose

Controls the Branch Record Buffer.

Configurations

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBCR_EL1 are UNDEFINED.

Attributes

BRBCR_EL1 is a 64-bit register.

Field descriptions

Bits [63:24]  
Reserved, RES0.

EXCEPTION, bit [23]  
Enable the recording of entry to EL1 via an exception.

0b0    Disable the recording of Branch records for exceptions when taken to EL1.
0b1    Enable the recording of Branch records for exceptions when taken to EL1.

The reset behavior of this field is:

• On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
• On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

ERTN, bit [22]  
Allow the recording Branch records for exception return instructions from EL1.

0b0    Disable the recording Branch records for exception return instructions from EL1.
0b1    Enable the recording Branch records for exception return instructions from EL1.

The reset behavior of this field is:

• On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
• On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

Bits [21:9]  
Reserved, RES0.
FZP, bit [8]

*When FEAT_PMUv3 is implemented:*

Freeze BRBE on PMU overflow.

- **0b0**: Branch recording is not affected by this control.
- **0b1**: A BRBE freeze event occurs when a PMU overflow occurs.

The reset behavior of this field is:

- On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally **UNKNOWN** value.
- On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally **UNKNOWN** value.

*Otherwise:*

Reserved, RES0.

Bit [7]

Reserved, RES0.

TS, bits [6:5]

Timestamp Control.

- **0b01**: Virtual timestamp. The BRBE recorded timestamp is the physical counter value, minus the value of `CNTVOFF_EL2`.
- **0b10**: *When FEAT_ECV is implemented:*
  
  Guest physical timestamp. The BRBE recorded timestamp is the physical counter value minus a physical offset. If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of `CNTPOFF_EL2`:
  
  - EL3 is implemented and `SCR_EL3.ECVEn == 0`.
  - EL2 is implemented and `CNDTHTL_EL2.ECV == 0`.
- **0b11**: Physical timestamp. The BRBE recorded timestamp is the physical counter value.

All other values are reserved.

This field is ignored by the PE when EL2 is implemented and `BRBCR_EL2.TS != 0b00`.

The reset behavior of this field is:

- On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally **UNKNOWN** value.
- On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally **UNKNOWN** value.

MPRED, bit [4]

Mask the recording of mispredicts.

- **0b0**: Disable the recording of mispredict information.
- **0b1**: Allow the recording of mispredict information.

The reset behavior of this field is:

- On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally **UNKNOWN** value.
- On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally **UNKNOWN** value.

CC, bit [3]

Enable the recording of cycle count information.

- **0b0**: Disable the recording of cycle count information.
- **0b1**: Allow the recording of cycle count information.
The reset behavior of this field is:

- On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

Bit [2]
Reserved, RES0.

E1BRE, bit [1]

EL1 Branch recording enable.

0b0 Branch recording prohibited at EL1.

0b1 Branch recording enabled at EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

E0BRE, bit [0]

EL0 Branch recording enable.

0b0 Branch recording prohibited at EL0.

0b1 Branch recording enabled at EL0.

This field is ignored by the PE when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

Accessing BRBCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS <Xt>, BRBCR_EL1}
\]

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b10 & 0b001 & 0b1001 & 0b0000 & 0b000
\end{array}
\]
elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
  X[t, 64] = NVMem[0x8E0];
else
  X[t, 64] = BRBCR_EL1;
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if
  elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if
  elsif HCR_EL2.E2H == '1' then
    X[t, 64] = BRBCR_EL2;
  else
    X[t, 64] = BRBCR_EL1;
elsif PSTATE_EL == EL3 then
  X[t, 64] = BRBCR_EL1;
MRS <xt>, BRBCR_EL12

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b101</td>
<td>0b1001</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '101' then
    X[t, 64] = NVMem[0x8E0];
  elsif EL2Enabled() & HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
  end if
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
      UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
      UNDEFINED;
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end if
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end if
    else
      X[t, 64] = BRBCR_EL1;
    end if
  else
    X[t, 64] = BRBCR_EL1;
UNDEFINED;
elsif PSTATE_EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        X[t, 64] = BRBCR_EL1;
    else
        UNDEFINED;
end if;

MSR BRBCR_EL1, <Xt>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'' && MDCR_EL3.SBBRE != '0' then
        UNDEFINED;
    elseif Halted() && HaveEL(EL3) && EDSCR.SDD == '0' && SCR_EL3.NS == '1' then
        UNDEFINED;
    endif;
    elsif EL2Enabled() && HaveEL(EL3) && MDCR_EL3.SBBRE == '11' && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        BRBCR_EL1 = X[t, 64];
    endif;
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'' && MDCR_EL3.SBBRE != '0' then
        UNDEFINED;
    elseif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && SCR_EL3.FGTEn == '11' then
        UNDEFINED;
    elseif Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    elseif Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        BRBCR_EL2 = X[t, 64];
    endif;
elsif PSTATE_EL == EL3 then
    BRBCR_EL1 = X[t, 64];
else
    BRBCR_EL1 = X[t, 64];
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x8E0] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
            UNDEFINED;
        elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
            UNDEFINED;
        elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end
        elseif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end
        end
    else
        BRBCR_EL1 = X[t, 64];
    end
else
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        BRBCR_EL1 = X[t, 64];
    else
        UNDEFINED;
    end
D17.8.2 BRBCR_EL2, Branch Record Buffer Control Register (EL2)

The BRBCR_EL2 characteristics are:

**Purpose**

Controls the Branch Record Buffer.

**Configurations**

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBCR_EL2 are UNDEFINED.

**Attributes**

BRBCR_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:24</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>23</td>
<td><strong>EXCEPTION</strong></td>
</tr>
<tr>
<td>0</td>
<td>Disable the recording of Branch records for exceptions when taken to EL2.</td>
</tr>
<tr>
<td>1</td>
<td>Enable the recording of Branch records for exceptions when taken to EL2.</td>
</tr>
<tr>
<td>22</td>
<td><strong>ERTN</strong></td>
</tr>
<tr>
<td>0</td>
<td>Disable the recording Branch records for exception return instructions from EL2.</td>
</tr>
<tr>
<td>1</td>
<td>Enable the recording Branch records for exception return instructions from EL2.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:

- On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>21:9</td>
<td>Reserved, RES0</td>
</tr>
</tbody>
</table>
FZP, bit [8]

*When FEAT_PMUv3 is implemented:*

Freeze BRBE on PMU overflow.

- **0b0** Branch recording is not affected by this control.
- **0b1** A BRBE freeze event occurs when a PMU overflow occurs.

The reset behavior of this field is:

- On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

Bit [7]

Reserved, RES0.

TS, bits [6:5]

Timestamp Control.

- **0b00** Timestamp controlled by BRBCR_EL1.TS.
- **0b01** Virtual timestamp. The BRBE recorded timestamp is the physical counter value, minus the value of CNTVOFF_EL2.
- **0b10** *When FEAT_ECV is implemented:* Guest physical timestamp. The BRBE recorded timestamp is the physical counter value minus a physical offset. If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2:
  - EL3 is implemented and SCR_EL3.ECVEn == 0.
  - EL2 is implemented and CNTHCTL_EL2.ECV == 0.
- **0b11** Physical timestamp. The BRBE recorded timestamp is the physical counter value.

The reset behavior of this field is:

- On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

MPRED, bit [4]

Mask the recording of mispredicts.

- **0b0** Disable the recording of mispredict information.
- **0b1** Allow the recording of mispredict information.

If EL2 is not implemented, then the Effective value of this field is 1, other than for a direct read of the register.

The reset behavior of this field is:

- On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

CC, bit [3]

Enable the recording of cycle count information.

- **0b0** Disable the recording of cycle count information.
0b1  Allow the recording of cycle count information.
If EL2 is not implemented, then the Effective value of this field is 1, other than for a direct read of
the register.

The reset behavior of this field is:
• On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an
architecturally UNKNOWN value.
• On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an
architecturally UNKNOWN value.

Bit [2]
Reserved, RES0.

E2BRE, bit [1]
EL2 Branch recording enable.
0b0  Branch recording prohibited at EL2.
0b1  Branch recording enabled at EL2.

The reset behavior of this field is:
• On a Warm reset, this field resets to 0.

E0HBRE, bit [0]
EL0 Branch recording enable.
0b0  Branch recording prohibited at EL0 when HCR_EL2.TGE == 1.
0b1  Branch recording enabled at EL0 when HCR_EL2.TGE == 1.

This field is ignored by the PE when any of the following are true:
• HCR_EL2.TGE == 0.
• EL2 is disabled in the current Security state.

The reset behavior of this field is:
• On a Warm reset, this field resets to 0.

Accessing BRBCR_EL2
When HCR_EL2.E2H is 1, without explicit synchronization, accesses from EL2 using the register name BRBCR_EL2
or BRBCR_EL1 are not guaranteed to be ordered with respect to accesses using the other register name.

Accesses to this register use the following encodings in the System register encoding space:

|MRS <Xt>, BRBCR_EL1|
|---|---|---|---|---|
|op0| op1| CRn| CRm| op2|
|0b1| 0b001| 0b1001| 0b0000| 0b000|

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap
  priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    UNDEFINED;
  elseif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDGRTR_EL2.mBRBCTL == '0') then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
  X[t, 64] = NVMem[0x8E0];
else
  X[t, 64] = BRBCR_EL1;
elsef PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    UNDEFINED;
  elself Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    UNDEFINED;
  elself HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
elsif PSTATE.EL == EL3 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    UNDEFINED;
  elself Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    UNDEFINED;
  elself HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
elsif PSTATE.EL == EL2 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    UNDEFINED;
  elself Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    UNDEFINED;
  elself HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elsif PSTATE_EL == EL0 then
  if PSTATE.EL == EL0 then
    UNDEFINED;
  elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      UNDEFINED;
    endif
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    UNDEFINED;
  elself Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    UNDEFINED;
  elself HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
elsif PSTATE.EL == EL3 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    UNDEFINED;
  elself Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    UNDEFINED;
  elself HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    UNDEFINED;
if Halted() & EDSCR.SDD == '1' then
  UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = BRBCR_EL2;
elsif PSTATE.EL == EL3 then
  X[t, 64] = BRBCR_EL2;

MSR BRBCR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b1001</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.SBRBE != '1' & SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.SBRBE != '11' & SCR_EL3.NS == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) & MDCR_EL3.SBRBE != '11' & SCR_EL3.NS == '0' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  elsif HaveEL(EL3) & MDCR_EL3.SBRBE != '0' & SCR_EL3.NS == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
    NVMem[0x8E0] = X[t, 64];
  else
    BRBCR_EL1 = X[t, 64];
  endif
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.SBRBE != '1' & SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & MDCR_EL3.SBRBE != '11' & SCR_EL3.NS == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) & MDCR_EL3.SBRBE != '11' & SCR_EL3.NS == '0' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  elsif HaveEL(EL3) & MDCR_EL3.SBRBE != '0' & SCR_EL3.NS == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  elsif HCR_EL2.E2H == '1' then
    BRBCR_EL2 = X[t, 64];
  else
    BRBCR_EL1 = X[t, 64];
  endif
elsif PSTATE.EL == EL3 then
  BRBCR_EL1 = X[t, 64];
MSR BRBCR_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b100</td>
<td>0b1001</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
        if boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '0' then
            UNDEFINED;
        elsif Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        endif
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE == '11' && SCR_EL3.NS == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        endif
    else
        BRBCR_EL2 = X[t, 64];
    endif
elsif PSTATE.EL == EL3 then
    BRBCR_EL2 = X[t, 64];

D17.8.3 BRBFCR_EL1, Branch Record Buffer Function Control Register

The BRBFCR_EL1 characteristics are:

**Purpose**

Functional controls for the Branch Record Buffer.

**Configurations**

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBFCR_EL1 are UNDEFINED.

**Attributes**

BRBFCR_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:30]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANK, bits [29:28]</td>
<td>Branch record buffer bank access control.</td>
</tr>
<tr>
<td>0b0</td>
<td>Select branch records 0 to 31.</td>
</tr>
<tr>
<td>0b1</td>
<td>Select branch records 32 to 63.</td>
</tr>
<tr>
<td>All other values are reserved.</td>
<td></td>
</tr>
<tr>
<td>The reset behavior of this field is:</td>
<td></td>
</tr>
<tr>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
<td></td>
</tr>
</tbody>
</table>

| Bits [27:23] | Reserved, RES0. |
| CONDDIR, bit [22] | Match on conditional direct branch instructions. |
| 0b0 | Do not match on conditional direct branch instructions. |
| 0b1 | Match on conditional direct branch instructions. |
| The reset behavior of this field is: |
| • On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value. |
| • On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value. |

| 0b0 | Do not match on direct branch with link instructions. |
0b1 Match on direct branch with link instructions.

The reset behavior of this field is:

- On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

**INDCALL, bit [20]**

Match on indirect branch with link instructions.

0b0 Do not match on indirect branch with link instructions.
0b1 Match on indirect branch with link instructions.

The reset behavior of this field is:

- On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

**RTN, bit [19]**

Match on function return instructions.

0b0 Do not match on function return instructions.
0b1 Match on function return instructions.

The reset behavior of this field is:

- On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

**INDIRECT, bit [18]**

Match on indirect branch instructions.

0b0 Do not match on indirect branch instructions.
0b1 Match on indirect branch instructions.

The reset behavior of this field is:

- On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

**DIRECT, bit [17]**

Match on unconditional direct branch instructions.

0b0 Do not match on unconditional direct branch instructions.
0b1 Match on unconditional direct branch instructions.

The reset behavior of this field is:

- On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

**EnI, bit [16]**

Include or exclude matches.

0b0 Include records for matches, and exclude records for non-matches.
0b1     Exclude records for matches, and include records for non-matches.

The reset behavior of this field is:

- On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

Bits [15:8]
Reserved, RES0.

PAUSED, bit [7]
Branch recording Paused status.

0b0     Branch recording is not Paused.
0b1     Branch recording is Paused.

The reset behavior of this field is:

- On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

LASTFAILED, bit [6]

When FEAT_TME is implemented:
Indicates transaction failure or cancellation.

0b0     Indicates that no transactions in a non-prohibited region have failed or been canceled since the last Branch record was generated.
0b1     Indicates that at least one transaction in a non-prohibited region has failed or been canceled since the last Branch record was generated.

The reset behavior of this field is:

- On a Cold reset, when FEAT_BRBEv1p1 is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset, when FEAT_BRBEv1p1 is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bits [5:0]
Reserved, RES0.

Accessing BRBFCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b10 & 0b001 & 0b1001 & 0b0000 & 0b001 \\
\end{array}
\]

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1' && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then 
UNDEFINED; 
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then 
UNDEFINED; 
elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then 
if Halted() && EDSCR.SDD == '1' then 
UNDEFINED; 
else 
AArch64.SystemAccessTrap(EL3, 0x18); 
endif; 
elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then 
if Halted() && EDSCR.SDD == '1' then 
UNDEFINED; 
else 
AArch64.SystemAccessTrap(EL3, 0x18); 
endif; 
else 
X[t, 64] = BRBFCR_EL1; 
endif; 
else 
if PSTATE.EL == EL0 then 
UNDEFINED; 
elsif PSTATE.EL == EL1 then 
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then 
UNDEFINED; 
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then 
UNDEFINED; 
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.mBRBCTL == '0' then 
AArch64.SystemAccessTrap(EL2, 0x18); 
elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then 
if Halted() && EDSCR.SDD == '1' then 
UNDEFINED; 
else 
AArch64.SystemAccessTrap(EL3, 0x18); 
endif; 
elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then 
if Halted() && EDSCR.SDD == '1' then 
UNDEFINED; 
else 
AArch64.SystemAccessTrap(EL3, 0x18); 
endif; 
else 
X[t, 64] = BRBFCR_EL1; 
endif; 
else 
if PSTATE.EL == EL3 then 
X[t, 64] = BRBFCR_EL1; 
endif; 
endif; 

**MSR BRBFCR_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b1001</td>
<td>0b0000</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then 
UNDEFINED; 
elsif PSTATE.EL == EL1 then 
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then 
UNDEFINED; 
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then 
UNDEFINED; 
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.mBRBCTL == '0' then 
AArch64.SystemAccessTrap(EL2, 0x18); 
elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then 
if Halted() && EDSCR.SDD == '1' then 
UNDEFINED; 
else 
AArch64.SystemAccessTrap(EL3, 0x18); 
endif; 
elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then 
if Halted() && EDSCR.SDD == '1' then 
UNDEFINED; 
else 
AArch64.SystemAccessTrap(EL3, 0x18); 
endif; 
else 
X[t, 64] = BRBFCR_EL1; 
endif; 
else 
if PSTATE.EL == EL3 then 
X[t, 64] = BRBFCR_EL1; 
endif; 
endif;
if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    BRBFCR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif Halted() && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        BRBFCR_EL3 = X[t, 64];
elsif PSTATE.EL == EL3 then
    BRBFCR_EL3 = X[t, 64];
D17.8.4  BRBIDR0_EL1, Branch Record Buffer ID0 Register

The BRBIDR0_EL1 characteristics are:

**Purpose**

Indicates the features of the branch buffer unit.

**Configurations**

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBIDR0_EL1 are UNDEFINED.

**Attributes**

BRBIDR0_EL1 is a 64-bit register.

**Field descriptions**

```
+-----------------+-----------------+-----------------+-----------------|
| Reserved, RES0  | CC              | FORMAT          | NUMREC          |
+-----------------+-----------------+-----------------+-----------------|
```

**Bits [63:16]**

Reserved, RES0.

**CC, bits [15:12]**

Cycle counter support. Defined values are:
- `0b0101` 20-bit cycle counter implemented.
- All other values are reserved.

**FORMAT, bits [11:8]**

Data format of records of the Branch record buffer. Defined values are:
- `0b0000` Format 0.
- All other values are reserved.

**NUMREC, bits [7:0]**

Number of records supported. Defined values are:
- `0x08` 8 branch records implemented.
- `0x10` 16 branch records implemented.
- `0x20` 32 branch records implemented.
- `0x40` 64 branch records implemented.
- All other values are reserved.

**Accessing BRBIDR0_EL1**

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, BRBIDR0_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b1001</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
        UNDEFINED;
    elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.nBRBIDR == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE == '11' && SCR_EL3.NS == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        elseif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            else
                X[t, 64] = BRBIDR0_EL1;
            elseif PSTATE_EL == EL2 then
                if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
                    UNDEFINED;
                elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
                    UNDEFINED;
                elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
                    if Halted() && EDSCR.SDD == '1' then
                        UNDEFINED;
                    else
                        AArch64.SystemAccessTrap(EL3, 0x18);
                    elseif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
                        if Halted() && EDSCR.SDD == '1' then
                            UNDEFINED;
                        else
                            AArch64.SystemAccessTrap(EL3, 0x18);
                        else
                            X[t, 64] = BRBIDR0_EL1;
                    elseif PSTATE_EL == EL3 then
                        X[t, 64] = BRBIDR0_EL1;
D17.8.5 BRBINF<n>_EL1, Branch Record Buffer Information Register <n>, n = 0 - 31

The BRBINF<n>_EL1 characteristics are:

**Purpose**

The information for Branch record n + (BRBFCR_EL1.BANK × 32).

**Configurations**

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBINF<n>_EL1 are UNDEFINED.

**Attributes**

BRBINF<n>_EL1 is a 64-bit register.

**Field descriptions**

![Field Diagram]

**Bits [63:47]**

Reserved, RES0.

**CCU, bit [46]**

The number of PE clock cycles since the last Branch record entry is UNKNOWN.

- 0b0 Indicates that the number of PE clock cycles since the last Branch record is indicated by BRBINF<n>_EL1.CC.
- 0b1 Indicates that the number of PE clock cycles since the last Branch record is UNKNOWN.

The value in this field is only valid when BRBINF<n>_EL1.VALID != 0b00.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When BRBINF<n>_EL1.VALID == 0b00, access to this field is RES0.
- Otherwise, access to this field is RO

**CC, bits [45:32]**

The number of PE clock cycles since the last Branch record entry.

The format of this field uses a mantissa and exponent to express the cycle count value, as follows:

- CC bits[7:0] indicate the mantissa M.
- CC bits[13:8] indicate the exponent E.

The cycle count is expressed using the following function:

\[
\text{if IsZero(E) then } \text{UInt}(M) \text{ else } \text{UInt}(1':M:Zeros(UInt(E)-1))
\]

If required, the cycle count is rounded to a multiple of \(2^{(E-1)}\) towards zero before being encoded.

A value of all ones in both the mantissa and exponent indicates the cycle count value exceeded the size of the cycle counter.

The value in this field is only valid when BRBINF<n>_EL1.VALID != 0b00.
The reset behavior of this field is:
  • On a Cold reset, this field resets to an architecturally **UNKNOWN** value.
Accessing this field has the following behavior:
  • RES0 if any of the following are true:
    — BRBINF<n>_EL1.CCU == 1
    — BRBINF<n>_EL1.VALID == 0b00
  • Otherwise, access to this field is RO

**Bits [31:18]**
Reserved, RES0.

**LASTFAILED, bit [17]**

*When FEAT_TME is implemented:*
Indicates transaction failure or cancellation.
  0b0 Indicates that no transactions in a non-prohibited region have failed or been canceled between the previous Branch record and this Branch record.
  0b1 Indicates that at least one transaction in a non-prohibited region has failed or been canceled between the previous Branch record and this Branch record.

The value in this field is only valid when BRBINF<n>_EL1.VALID != 0b00.
The reset behavior of this field is:
  • On a Cold reset, this field resets to an architecturally **UNKNOWN** value.
Accessing this field has the following behavior:
  • When BRBINF<n>_EL1.VALID == 0b00, access to this field is RES0.
  • Otherwise, access to this field is RO

*Otherwise:*
Reserved, RES0.

**T, bit [16]**

*When FEAT_TME is implemented:*
Transactional state.
  0b0 The branch or exception was not executed in Transactional state.
  0b1 The branch or exception was executed in Transactional state.

The value in this field is only valid when BRBINF<n>_EL1.VALID == 0b10 or BRBINF<n>_EL1.VALID == 0b11.
The reset behavior of this field is:
  • On a Cold reset, this field resets to an architecturally **UNKNOWN** value.
Accessing this field has the following behavior:
  • RES0 if any of the following are true:
    — BRBINF<n>_EL1.VALID == 0b00
    — BRBINF<n>_EL1.VALID == 0b01
  • Otherwise, access to this field is RO

*Otherwise:*
Reserved, RES0.

**Bits [15:14]**
Reserved, RES0.

**TYPE, bits [13:8]**
Branch type.
  0b000000 Unconditional direct branch, excluding Branch with link.
0b000001  Indirect branch, excluding Branch with link, Return from subroutine, and Exception return.
0b000010  Direct Branch with link.
0b000011  Indirect Branch with link.
0b000101  Return from subroutine.
0b000111  Exception return.
0b001000  Conditional direct branch.
0b100001  Debug halt.
0b100010  Call.
0b100011  Trap.
0b101000  SError.
0b101010  Instruction debug.
0b101011  Data debug.
0b101100  Alignment.
0b101110  Inst Fault.
0b101101  Data Fault.
0b101100  IRQ.
0b101111  FIQ.
0b111001  Debug State Exit.
All other values are reserved.
The value in this field is only valid when BRBINF<n>_EL1.VALID != 0b00.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.
Accessing this field has the following behavior:
• When BRBINF<n>_EL1.VALID == 0b00, access to this field is RES0.
• Otherwise, access to this field is RO

EL, bits [7:6]
The Exception Level at the target address.
0b0  EL0.
0b1  EL1.
0b10 EL2.
0b11 When FEAT_BRBEv1p1 is implemented:
   EL3.
The value in this field is only valid when BRBINF<n>_EL1.VALID == 0b11 or
BRBINF<n>_EL1.VALID == 0b01.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.
Accessing this field has the following behavior:
• RES0 if any of the following are true:
   — BRBINF<n>_EL1.VALID == 0b00
   — BRBINF<n>_EL1.VALID == 0b10
• Otherwise, access to this field is RO

MPRED, bit [5]
Branch mispredict.
0b0 Branch was correctly predicted or the result of the prediction was not captured.
0b1 Branch was incorrectly predicted.

The value in this field is only valid when \( \text{BRBINF}_n.\_\text{EL1.VALID} = 0b11 \) or \( \text{BRBINF}_n.\_\text{EL1.VALID} = 0b10 \).

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally \texttt{UNKNOWN} value.

Accessing this field has the following behavior:
- \texttt{RES0} if any of the following are true:
  - \( \text{BRBINF}_n.\_\text{EL1.VALID} = 0b00 \)
  - \( \text{BRBINF}_n.\_\text{EL1.VALID} = 0b01 \)
  - \( \text{BRBINF}_n.\_\text{EL1.TYPE}[5] = 1 \)
- Otherwise, access to this field is RO

Bits [4:2]
Reserved, \texttt{RES0}.

VALID, bits [1:0]
The Branch record is valid.

\( 0b00 \) This Branch record is not valid.
The values of following fields are not valid:
- \( \text{BRBTGT}_n.\_\text{EL1.ADDRESS} \)
- \( \text{BRBSRC}_n.\_\text{EL1.ADDRESS} \)
- \( \text{BRBINF}_n.\_\text{EL1.MPRED} \)
- \( \text{BRBINF}_n.\_\text{EL1.LASTFAILED} \)
- \( \text{BRBINF}_n.\_\text{EL1.T} \)
- \( \text{BRBINF}_n.\_\text{EL1.EL} \)
- \( \text{BRBINF}_n.\_\text{EL1.TYPE} \)
- \( \text{BRBINF}_n.\_\text{EL1.CC} \)
- \( \text{BRBINF}_n.\_\text{EL1.CCU} \)

\( 0b01 \) This Branch record is valid.
The values of following fields are not valid:
- \( \text{BRBSRC}_n.\_\text{EL1.ADDRESS} \)
- \( \text{BRBINF}_n.\_\text{EL1.T} \)
- \( \text{BRBINF}_n.\_\text{EL1.MPRED} \)

\( 0b10 \) This Branch record is valid.
The values of following fields are not valid:
- \( \text{BRBTGT}_n.\_\text{EL1.ADDRESS} \)
- \( \text{BRBINF}_n.\_\text{EL1.EL} \)

\( 0b11 \) This Branch record is valid.
The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally \texttt{UNKNOWN} value.

**Accessing \text{BRBINF}_n.\_\text{EL1}**

\( \text{BRBINF}_n.\_\text{EL1} \) reads-as-zero if \( n + (\text{BRBFCR.EL1.BANK} \times 32) \geq \text{BRBIDR0.EL1.NUMREC} \).

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, BRBINF<\textless m\textrangle> \_EL1

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
    op0 & op1 & CRn & CRm & op2 \\
\hline
    0b10 & 0b001 & 0b1000 & m[3:0] & m[4]:0b00 \\
\hline
\end{tabular}
\end{table}

integer m = UInt(op2<2>::CRm<3:0>);

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.nBRBDATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end if
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
        if Halted() && EDSCR.SDD == '0' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end if
    end if
elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end if
elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '1' then
    if Halted() && EDSCR.SDD == '0' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end if
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
        UNDEFINED;
    elsif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.nBRBDATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end if
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
        if Halted() && EDSCR.SDD == '0' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end if
    end if
elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end if
elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '1' then
    if Halted() && EDSCR.SDD == '0' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end if
elsif m + (UInt(BRBFCR_EL1.BANK) * 32) >= NUM_BRBE_RECORDS then
    X[t, 64] = Zeros(64);
else
    X[t, 64] = BRBINF_EL1[m];
elsif PSTATE.EL == EL3 then
    if m + (UInt(BRBFCR_EL1.BANK) * 32) >= NUM_BRBE_RECORDS then
        X[t, 64] = Zeros(64);
else
    X[t, 64] = BRBINF_EL1[m];
end if
end if
D17.8.6  BRBINFINJ_EL1, Branch Record Buffer Information Injection Register

The BRBINFINJ_EL1 characteristics are:

**Purpose**

The information of a Branch record for injection.

**Configurations**

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBINFINJ_EL1 are UNDEFINED.

**Attributes**

BRBINFINJ_EL1 is a 64-bit register.

**Field descriptions**

![Field Descriptions Diagram]

**Bits [63:47]**

Reserved, RES0.

**CCU, bit [46]**

The number of PE clock cycles since the last Branch record entry is UNKNOWN.

- 0b0  Indicates that the number of PE clock cycles since the last Branch record is indicated by BRBINFINJ_EL1.CC.
- 0b1  Indicates that the number of PE clock cycles since the last Branch record is UNKNOWN.

The value in this field is only valid when BRBINFINJ_EL1.VALID != 0b00.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:
- When BRBINFINJ_EL1.VALID == 0b00, access to this field is RES0.
- Otherwise, access to this field is RW.

**CC, bits [45:32]**

The number of PE clock cycles since the last Branch record entry.

The format of this field uses a mantissa and exponent to express the cycle count value, as follows:
- CC bits[7:0] indicate the mantissa M.
- CC bits[13:8] indicate the exponent E.

The cycle count is expressed using the following function:

\[
\text{if } \text{IsZero}(E) \text{ then } \text{UInt}(M) \text{ else } \text{UInt}(\text{UInt}(E)-1)\]

If required, the cycle count is rounded to a multiple of \(2^{(E-1)} \) towards zero before being encoded.

A value of all ones in both the mantissa and exponent indicates the cycle count value exceeded the size of the cycle counter.

The value in this field is only valid when BRBINFINJ_EL1.VALID != 0b00.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:
- RES0 if any of the following are true:
  - BRBINFINJ_EL1.CCU == 1
  - BRBINFINJ_EL1.VALID == 0b00
- Otherwise, access to this field is RW

### Bits [31:18]

Reserved, RES0.

### LASTFAILED, bit [17]

When FEAT_TME is implemented:
Indicates transaction failure or cancellation.
- 0b0 Indicates that no transactions in a non-prohibited region have failed or been canceled between the previous Branch record and this Branch record.
- 0b1 Indicates that at least one transaction in a non-prohibited region has failed or been canceled between the previous Branch record and this Branch record.

The value in this field is only valid when BRBINFINJ_EL1.VALID != 0b00.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:
- When BRBINFINJ_EL1.VALID == 0b00, access to this field is RES0.
- Otherwise, access to this field is RW

Otherwise:
Reserved, RES0.

### T, bit [16]

When FEAT_TME is implemented:
Transactional state.
- 0b0 The branch or exception was not executed in Transactional state.
- 0b1 The branch or exception was executed in Transactional state.

The value in this field is only valid when BRBINFINJ_EL1.VALID == 0b10 or BRBINFINJ_EL1.VALID == 0b11.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:
- RES0 if any of the following are true:
  - BRBINFINJ_EL1.VALID == 0b00
  - BRBINFINJ_EL1.VALID == 0b01
- Otherwise, access to this field is RW

Otherwise:
Reserved, RES0.

### Bits [15:14]

Reserved, RES0.

### TYPE, bits [13:8]

Branch type.
- 0b000000 Unconditional direct branch, excluding Branch with link.
0b000001 Indirect branch, excluding Branch with link, Return from subroutine, and Exception return.
0b000010 Direct Branch with link.
0b000011 Indirect Branch with link.
0b000101 Return from subroutine.
0b000111 Exception return.
0b001000 Conditional direct branch.
0b100001 Debug halt.
0b100010 Call.
0b100011 Trap.
0b100100 SError.
0b100110 Instruction debug.
0b100111 Data debug.
0b101010 Alignment.
0b101011 Inst Fault.
0b101100 Data Fault.
0b101110 IRQ.
0b101111 FIQ.
0b110001 Debug State Exit.

All other values are reserved.
The value in this field is only valid when BRBINFINJ_EL1.VALID != 0b00.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Accessing this field has the following behavior:
• When BRBINFINJ_EL1.VALID == 0b00, access to this field is RES0.
• Otherwise, access to this field is RW

EL, bits [7:6]
The Exception Level at the target address.
0b00 EL0.
0b01 EL1.
0b10 EL2.
0b11 When FEAT_BRBEv1p1 is implemented:
EL3.
The value in this field is only valid when BRBINFINJ_EL1.VALID == 0b11 or BRBINFINJ_EL1.VALID == 0b01.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Accessing this field has the following behavior:
• RES0 if any of the following are true:
  — BRBINFINJ_EL1.VALID == 0b00
  — BRBINFINJ_EL1.VALID == 0b10
• Otherwise, access to this field is RW

MPRED, bit [5]
Branch mispredict.
0b0 Branch was correctly predicted or the result of the prediction was not captured.
0b1  Branch was incorrectly predicted.
The value in this field is only valid when BRBINFINJ_EL1.VALID == 0b11 or
BRBINFINJ_EL1.VALID == 0b10.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Accessing this field has the following behavior:
• RES0 if any of the following are true:
  — BRBINFINJ_EL1.VALID == 0b00
  — BRBINFINJ_EL1.VALID == 0b01
  — BRBINFINJ_EL1.TYPE[5] == 1
• Otherwise, access to this field is RW

Bits [4:2]
Reserved, RES0.

VALID, bits [1:0]
The Branch record is valid.
0b00  This Branch record is not valid.
The values of following fields are not valid:
• BRBTGTINJ_EL1.ADDRESS.
• BRBSRCINJ_EL1.ADDRESS.
• BRBINFINJ_EL1.MPRED.
• BRBINFINJ_EL1.LASTFAILED.
• BRBINFINJ_EL1.T.
• BRBINFINJ_EL1.EL.
• BRBINFINJ_EL1.TYPE.
• BRBINFINJ_EL1.CC.
• BRBINFINJ_EL1.CCU.
0b01  This Branch record is valid.
The values of following fields are not valid:
• BRBSRCINJ_EL1.ADDRESS.
• BRBINFINJ_EL1.T.
• BRBINFINJ_EL1.MPRED.
0b10  This Branch record is valid.
The values of following fields are not valid:
• BRBTGTINJ_EL1.ADDRESS.
• BRBINFINJ_EL1.MPRED.
0b11  This Branch record is valid.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing BRBINFINJ_EL1
Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, BRBINFINJ_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
UNDEFINED;
elif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
UNDEFINED;
elif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
elif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = BRBINFINJ_EL1;
else
PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
UNDEFINED;
elif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
UNDEFINED;
elif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
elif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = BRBINFINJ_EL1;
else
PSTATE.EL == EL3 then
X[t, 64] = BRBINFINJ_EL1;

MSR BRBINFINJ_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
  UNDEFINED;
elseif Halted() && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap
  priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
  UNDEFINED;
elseif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap
  priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() && SCR_EL3.FGTEn == '11' && HDFGWTR_EL2.nBRBDATA == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  endif
elseif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap
  priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
  UNDEFINED;
elseif PSTATE.EL == EL3 then
  if Halted() && SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  else
    BRBINFINJ_EL1 = X[t, 64];
  endif
elseif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap
  priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) && HDFGWTR_EL2.nBRBDATA == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  endif
elseif PSTATE.EL == EL3 then
  if Halted() && SCR_EL3.NS == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    endif
  endif
else
  BRBINFINJ_EL1 = X[t, 64];
endif
elseif PSTATE.EL == EL3 then
  BRBINFINJ_EL1 = X[t, 64];
else
  BRBINFINJ_EL1 = X[t, 64];
D17.8.7 BRBSRC\(<n>_EL1, Branch Record Buffer Source Address Register \(<n>, n = 0 - 31\)

The BRBSRC\(<n>_EL1 characteristics are:

**Purpose**

The source address of Branch record \(n + (\text{BRBFCR}_\text{EL1.BANK} \times 32)\).

**Configurations**

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBSRC\(<n>_EL1 are UNDEFINED.

**Attributes**

BRBSRC\(<n>_EL1 is a 64-bit register.

**Field descriptions**

ADDRESS, bits [63:0]

Source virtual address of the Branch record.

When an indirect write occurs with a value with ADDRESS bits [63:P] being other than all zeroes or all ones, an UNKNOWN value which is not all zeroes or all ones is written to bits [63:P]. P is defined as the virtual address size supported by the PE, as returned by VAMax(). The value in bits [P-1:0] are the value written.

When an indirect write occurs with a value with ADDRESS bits [63:P] being all zeroes or all ones, the written value is written to bits [63:0], and a read of the register returns the written value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- RES0 if any of the following are true:
  - \(\text{BRBINF}_\text{<n>_EL1.VALID == 0b00}\)
  - \(\text{BRBINF}_\text{<n>_EL1.VALID == 0b01}\)
- Otherwise, access to this field is RO

Accessing BRBSRC\(<n>_EL1

BRBSRC\(<n>_EL1 is RES0 if \(n + (\text{BRBFCR}_\text{EL1.BANK} \times 32) >= \text{BRBIDR0}_\text{EL1.NUMREC}\).

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, BRBSRC\(<m>_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b01</td>
<td>0b1000</td>
<td>m[3:0]</td>
<td>m[4]:0b01</td>
</tr>
</tbody>
</table>

\[\text{integer } m = \text{UInt}(\text{op2<2>:CRn<3:0>});\]

\[\text{if } \text{PSTATE.EL} == \text{EL0 then} \]
UNDEFINED;
elseif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL1) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
UNDEFINED;
elseif Halted() && HaveEL(EL1) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
UNDEFINED;
elseif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGTR_EL2.mBRBDATA == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elseif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
elseif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else m + (UInt(BRBFCR_EL1.BANK) * 32) >= NUM_BRBE_RECORDS then
X[t, 64] = Zeros(64);
elself PSTATE.EL == EL2 then
if Halted() && HaveEL(EL2) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
UNDEFINED;
elseif Halted() && HaveEL(EL2) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
UNDEFINED;
elseif HaveEL(EL2) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL2, 0x18);
elseif HaveEL(EL2) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL2, 0x18);
else m + (UInt(BRBFCR_EL1.BANK) * 32) >= NUM_BRBE_RECORDS then
X[t, 64] = Zeros(64);
elself PSTATE.EL == EL3 then
if m + (UInt(BRBFCR_EL1.BANK) * 32) >= NUM_BRBE_RECORDS then
X[t, 64] = Zeros(64);
elself PSTATE.EL == EL3 then
if m + (UInt(BRBFCR_EL1.BANK) * 32) >= NUM_BRBE_RECORDS then
X[t, 64] = Zeros(64);
elself PSTATE.EL == EL3 then
if m + (UInt(BRBFCR_EL1.BANK) * 32) >= NUM_BRBE_RECORDS then
X[t, 64] = Zeros(64);
D17.8.8 BRBSRCINJ_EL1, Branch Record Buffer Source Address Injection Register

The BRBSRCINJ_EL1 characteristics are:

Purpose

The source address of a Branch record for injection.

Configurations

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBSRCINJ_EL1 are UNDEFINED.

Attributes

BRBSRCINJ_EL1 is a 64-bit register.

Field descriptions

ADDRESS, bits [63:0]

Source virtual address of the Branch record.

When a direct write occurs with a value with ADDRESS bits [63:P] being other than all zeroes or all ones, an UNKNOWN value which is not all zeroes or all ones is written to bits [63:P]. P is defined as the virtual address size supported by the PE, as returned by VAMax(). The value in bits [P-1:0] are the value written.

When a direct write occurs with a value with ADDRESS bits [63:P] being all zeroes or all ones, the written value is written to bits [63:0], and a read of the register returns the written value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- RES0 if any of the following are true:
  - BRBINFINJ_EL1.VALID == 0b00
  - BRBINFINJ_EL1.VALID == 0b01

- Otherwise, access to this field is RW

Accessing BRBSRCINJ_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[ MRS <Xt>, BRBSRCINJ_EL1 \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
else if PSTATE_EL == EL1 then
  if Halted() && HaveEl(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
UNDEFINED:
  elsif Halted() && HaveEL(EL3) && EdSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == '11' && SCR_EL3.NS == '1' then
  UNDEFINED;
  elsif EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.mBBRDATA == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '0' && SCR_EL3.NS == '0' then
    if Halted() && EdSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  elsif HaveEL(EL3) && MDCR_EL3.SBRBE == '1' && SCR_EL3.NS == '1' then
    if Halted() && EdSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = BRBSRCINJ_EL1;
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EdSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
      UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EdSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
      if Halted() && EdSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
      if Halted() && EdSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    else
      X[t, 64] = BRBSRCINJ_EL1;
    elsif PSTATE.EL == EL3 then
      X[t, 64] = BRBSRCINJ_EL1;
  end

MSR BRBSRCINJ_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EdSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EdSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    if Halted() && EdSCR.SDD == '1' then
      UNDEFINED;
    else
      EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.mBBRDATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '0' && SCR_EL3.NS == '0' then
        if Halted() && EdSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL2, 0x18);
        end
      elsif HaveEL(EL3) && MDCR_EL3.SBRBE == '1' && SCR_EL3.NS == '1' then
        if Halted() && EdSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL2, 0x18);
        end
      else
        X[t, 64] = BRBSRCINJ_EL1;
      else
        X[t, 64] = BRBSRCINJ_EL1;
    end
end

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EdSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EdSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    if Halted() && EdSCR.SDD == '1' then
      UNDEFINED;
    else
      EL2Enabled() && ((HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.mBBRDATA == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '0' && SCR_EL3.NS == '0' then
        if Halted() && EdSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL2, 0x18);
        end
      elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
        if Halted() && EdSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.SystemAccessTrap(EL2, 0x18);
        end
      else
        X[t, 64] = BRBSRCINJ_EL1;
      else
        X[t, 64] = BRBSRCINJ_EL1;
    end
end

X[t, 64] = BRBSRCINJ_EL1;
else
  X[t, 64] = BRBSRCINJ_EL1;
end
UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  BRBSRCINJ_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    BRBSRCINJ_EL1 = X[t, 64];
  elsif PSTATE.EL == EL3 then
    BRBSRCINJ_EL1 = X[t, 64];
D17.8.9 BRBTGT<n>_EL1, Branch Record Buffer Target Address Register <n>, n = 0 - 31

The BRBTGT<n>_EL1 characteristics are:

**Purpose**

The target address of Branch record \( n + (\text{BRBFCR_EL1.BANK} \times 32) \).

**Configurations**

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBTGT<n>_EL1 are UNDEFINED.

**Attributes**

BRBTGT<n>_EL1 is a 64-bit register.

**Field descriptions**

ADDRESS, bits [63:0]

Target virtual address of the Branch record.

When an indirect write occurs with a value with ADDRESS bits [63:P] being other than all zeroes or all ones, an **UNKNOWN** value which is not all zeroes or all ones is written to bits [63:P]. P is defined as the virtual address size supported by the PE, as returned by VAMMax(). The value in bits [P-1:0] are the value written.

When an indirect write occurs with a value with ADDRESS bits [63:P] being all zeroes or all ones, the written value is written to bits [63:0], and a read of the register returns the written value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally **UNKNOWN** value.

Accessing this field has the following behavior:

- **RES0** if any of the following are true:
  - \( \text{BRBINF<n>_EL1.VALID} == 0b00 \)
  - \( \text{BRBINF<n>_EL1.VALID} == 0b10 \)

- Otherwise, access to this field is **RO**.

**Accessing BRBTGT<n>_EL1**

BRBTGT<n>_EL1 is **RES0** if \( n + (\text{BRBFCR_EL1.BANK} \times 32) >= \text{BRBIDR0_EL1.NUMREC} \).

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, BRBTGT<m>_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b1000</td>
<td>m[3:0]</td>
<td>m[4]:0b10</td>
</tr>
</tbody>
</table>

integer \( m = \text{UInt}(\text{op2}<2>:\text{CRm}<3:0>) \);

if PSTATE.EL == EL0 then
UNDEFINED;
eslif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSR.SOO == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '1' && SCR_EL3.NS == '0' then
        UNDEFINED;
eslif Halted() && HaveEL(EL3) && EDSR.SOO == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
        UNDEFINED;
eslif HaveEL(EL3) && MDCR_EL3.SBRBE != '1' && SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SOO == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
eslif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    if Halted() && EDSCR.SOO == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
eslif m + (UInt(BRBFCR_EL1.BANK) * 32) >= NUM_BRBE_RECORDS then
    X[t, 64] = Zeros(64);
else
    X[t, 64] = BRBTGT_EL1[m];
eslif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSR.SOO == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '1' && SCR_EL3.NS == '0' then
        UNDEFINED;
eslif Halted() && HaveEL(EL3) && EDSR.SOO == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
        UNDEFINED;
eslif HaveEL(EL3) && MDCR_EL3.SBRBE != '1' && SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SOO == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
eslif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    if Halted() && EDSCR.SOO == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
eslif m + (UInt(BRBFCR_EL1.BANK) * 32) >= NUM_BRBE_RECORDS then
    X[t, 64] = Zeros(64);
else
    X[t, 64] = BRBTGT_EL1[m];
eslif PSTATE.EL == EL3 then
    if m + (UInt(BRBFCR_EL1.BANK) * 32) >= NUM_BRBE_RECORDS then
    X[t, 64] = Zeros(64);
else
    X[t, 64] = BRBTGT_EL1[m];
D17.8.10 BRBTGTINJ_EL1, Branch Record Buffer Target Address Injection Register

The BRBTGTINJ_EL1 characteristics are:

**Purpose**

The target address of a Branch record for injection.

**Configurations**

This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBTGTINJ_EL1 are UNDEFINED.

**Attributes**

BRBTGTINJ_EL1 is a 64-bit register.

**Field descriptions**

ADDRESS, bits [63:0]

Target virtual address of the Branch record.

When a direct write occurs with a value with ADDRESS bits [63:P] being other than all zeroes or all ones, an UNKNOWN value which is not all zeroes or all ones is written to bits [63:P]. P is defined as the virtual address size supported by the PE, as returned by VAMax(). The value in bits [P-1:0] are the value written.

When a direct write occurs with a value with ADDRESS bits [63:P] being all zeroes or all ones, the written value is written to bits [63:0], and a read of the register returns the written value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- RES0 if any of the following are true:
  - BRBINFINJ_EL1.VALID == 0b00
  - BRBINFINJ_EL1.VALID == 0b10
- Otherwise, access to this field is RW

Accessing BRBTGTINJ_EL1

Accesses to this register use the following encodings in the System register encoding space:

\[ MRS <Xt>, BRBTGTINJ_EL1 \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then

D17-6950 Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. Non-Confidential
UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDO == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == '11' && SCR_EL3.NS == '1' then
    UNDEFINED;
elsif EL2Enabled() && ((HaveEL(EL3)) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.mBRBDATA == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SDO == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    endif
    elseif HaveEL(EL3) && MDCR_EL3.SBRBE == '0' && SCR_EL3.NS == '1' then
        if Halted() && EDSCR.SDO == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        endif
    else
        X[t, 64] = BRBTGTINJ_EL1;
    end
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDO == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
        UNDEFINED;
    elseif Halted() && EDSCR.SDO == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    endif
elsif HaveEL(EL3) && MDCR_EL3.SBRBE == '1' &&SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SDO == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    endif
else
    X[t, 64] = BRBTGTINJ_EL1;
endif PSTATE.EL == EL3 then
X[t, 64] = BRBTGTINJ_EL1;

**MSR BRBTGTINJ_EL1, <xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b1001</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDO == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
        UNDEFINED;
    elseif Halted() && EDSCR.SDO == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL2, 0x18);
    endif
elsif HaveEL(EL3) && MDCR_EL3.SBRBE == '0' && SCR_EL3.NS == '1' then
    if Halted() && EDSCR.SDO == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    endif
elsif HaveEL(EL3) && MDCR_EL3.SBRBE == '1' && SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SDO == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    endif
elsif HaveEL(EL3) && MDCR_EL3.SBRBE == '1' && SCR_EL3.NS == '1' then
    if Halted() && EDSCR.SDO == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    endif
else
    X[t, 64] = BRBTGTINJ_EL1;
endif PSTATE.EL == EL3 then
X[t, 64] = BRBTGTINJ_EL1;
UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  BRBTGTINJ_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if
  elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end if
  else
    BRBTGTINJ_EL1 = X[t, 64];
  endif
elsif PSTATE.EL == EL3 then
  BRBTGTINJ_EL1 = X[t, 64];
D17.8.11 BRBTS_EL1, Branch Record Buffer Timestamp Register

The BRBTS_EL1 characteristics are:

**Purpose**
Captures the Timestamp value on a BRBE freeze event.

**Configurations**
This register is present only when FEAT_BRBE is implemented. Otherwise, direct accesses to BRBTS_EL1 are UNDEFINED.

**Attributes**
BRBTS_EL1 is a 64-bit register.

**Field descriptions**

![Field Diagram]

TS, bits [63:0]
Timestamp value at the time of a BRBE freeze event.
The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**Accessing BRBTS_EL1**
Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS <Xt>, BRBTS_EL1}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b1001</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
    UNDEFINED;
elseif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    UNDEFINED;
elseif EL2Enabled() && (HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.mBRBDATA == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elseif HaveEL(EL3) && MDCR_EL3.SBRBE != '11' && SCR_EL3.NS == '0' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
elelse
    AArch64.SystemAccessTrap(EL3, 0x18);
elseif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  \(X[t, 64] = BRBTS\_EL1\);
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'
  & MDCR.EL3.SBRBE != '11' & SCR.EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'
  & MDCR.EL3.SBRBE == 'x0' & SCR.EL3.NS == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) & MDCR.EL3.SBRBE != '11' & SCR.EL3.NS == '0' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
elsif HaveEL(EL3) & MDCR.EL3.SBRBE == 'x0' & SCR.EL3.NS == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
else
  \(X[t, 64] = BRBTS\_EL1\);
elsif PSTATE.EL == EL3 then
  \(X[t, 64] = BRBTS\_EL1\);

**MSR BRBTS_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0b001</td>
<td>0b1001</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'
  & MDCR.EL3.SBRBE != '11' & SCR.EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'
  & MDCR.EL3.SBRBE == 'x0' & SCR.EL3.NS == '1' then
    UNDEFINED;
  elsif EL2Enabled() & ((HaveEL(EL3) || SCR.EL3.FGETn == '1') & HDFGWTR.EL2.mRRBDATA == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) & MDCR.EL3.SBRBE != '11' & SCR.EL3.NS == '0' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
elsif HaveEL(EL3) & MDCR.EL3.SBRBE == 'x0' & SCR.EL3.NS == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  endif
else
  BRBTS_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'
  & MDCR.EL3.SBRBE != '11' & SCR.EL3.NS == '0' then
    UNDEFINED;
  elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'
  & MDCR.EL3.SBRBE == 'x0' & SCR.EL3.NS == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) & MDCR.EL3.SBRBE != '11' & SCR.EL3.NS == '0' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
elsif HaveEL(EL3) && MDCR_EL3.SBRBE == 'x0' && SCR_EL3.NS == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        BRBTS_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    BRBTS_EL1 = X[t, 64];
D17.9 RAS registers

This section lists The Reliability, Availability, and Serviceability Extension registers in AArch64.
D17.9.1 DISR_EL1, Deferred Interrupt Status Register

The DISR_EL1 characteristics are:

**Purpose**

Records that an SError interrupt has been consumed by an ESB instruction.

**Configurations**

AArch64 System register DISR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DISR[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to DISR_EL1 are UNDEFINED.

**Attributes**

DISR_EL1 is a 64-bit register.

**Field descriptions**

*When DISR_EL1.IDS == 0:*

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>13</th>
<th>12</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RES0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>RES0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IDS</td>
<td></td>
<td></td>
<td></td>
<td>AET</td>
<td>EA</td>
<td>RES0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**A, bit [31]**

Set to 1 when an ESB instruction defers an asynchronous SError interrupt. If the implementation does not include any sources of SError interrupt that can be synchronized by an Error Synchronization Barrier, then this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Bits [30:25]**

Reserved, RES0.

**IDS, bit [24]**

Indicates the deferred SError interrupt type.

- **0b0**: Deferred error uses architecturally-defined format.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Bits [23:13]**

Reserved, RES0.

**AET, bits [12:10]**

Asynchronous Error Type. See the description of ESR_ELx.AET for an SError interrupt.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
EA, bit [9]
External abort Type. See the description of ESR_ELx.EA for an SError interrupt.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:6]
Reserved, RES0.

DFSC, bits [5:0]
Fault Status Code. See the description of ESR_ELx.DFSC for an SError interrupt.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When DISR_EL1.IDS == 1:

Bits [63:32]
Reserved, RES0.

A, bit [31]
Set to 1 when an ESB instruction defers an asynchronous SError interrupt. If the implementation does not include any sources of SError interrupt that can be synchronized by an Error Synchronization Barrier, then this bit is RES0.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:25]
Reserved, RES0.

IDS, bit [24]
Indicates the deferred SError interrupt type.
0b1 Deferred error uses IMPLEMENTATION DEFINED format.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS, bits [23:0]
IMPLEMENTATION DEFINED syndrome. See the description of ESR_ELx[23:0] for an SError interrupt.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DISR_EL1
An indirect write to DISR_EL1 made by an ESB instruction does not require an explicit synchronization operation for the value that is written to be observed by a direct read of DISR_EL1 occurring in program order after the ESB instruction.
DISR_EL1 is RAZ/WI if EL3 is implemented, the PE is in Non-debug state, SCR_EL3.EA == 1, and any of the following apply:

- At EL2.
- At EL1 and ((SCR_EL3.NS == 0 && SCR_EL3.EEL2 == 0) || HCR_EL2.AMO == 0).

Accesses to this register use the following encodings in the System register encoding space:

### MRS <Xt>, DISR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.AMO == '1' then
        X[t, 64] = VDISR_EL2;
    elsif HaveEL(EL3) && !Halted() && SCR_EL3.EA == '1' then
        X[t, 64] = Zeros(64);
    else
        X[t, 64] = DISR_EL1;

elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !Halted() && SCR_EL3.EA == '1' then
        X[t, 64] = Zeros(64);
    else
        DISR_EL1 = X[t, 64];

elsif PSTATE.EL == EL3 then
    DISR_EL1 = X[t, 64];
```

### MSR DISR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.AMO == '1' then
        VDISR_EL2 = X[t, 64];
    elsif HaveEL(EL3) && !Halted() && SCR_EL3.EA == '1' then
        return;
    else
        DISR_EL1 = X[t, 64];

elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !Halted() && SCR_EL3.EA == '1' then
        return;
    else
        DISR_EL1 = X[t, 64];

elsif PSTATE.EL == EL3 then
    DISR_EL1 = X[t, 64];
```
### D17.9.2 ERRIDR_EL1, Error Record ID Register

The ERRIDR_EL1 characteristics are:

**Purpose**
Defines the highest numbered index of the error records that can be accessed through the Error Record System registers.

**Configurations**
AArch64 System register ERRIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERRIDR[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERRIDR_EL1 are UNDEFINED.

**Attributes**
ERRIDR_EL1 is a 64-bit register.

**Field descriptions**

- **Bits [63:16]**
  Reserved, RES0.

- **NUM, bits [15:0]**
  Highest numbered index of the records that can be accessed through the Error Record System registers plus one. Zero indicates no records can be accessed through the Error Record System registers.

  Each implemented record is owned by a node. A node might own multiple records.

**Accessing ERRIDR_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ERRIDR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>
if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    X[t, 64] = ERRIDR_EL1;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = ERRIDR_EL1;
    elsif PSTATE.EL == EL3 then
        X[t, 64] = ERRIDR_EL1;
D17.9.3 ERRSELR_EL1, Error Record Select Register

The ERRSELR_EL1 characteristics are:

**Purpose**

Selects an error record to be accessed through the Error Record System registers.

**Configurations**

AArch64 System register ERRSELR_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERRSELR[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERRSELR_EL1 are UNDEFINED.

If ERRIDR_EL1 indicates that zero error records are implemented, then it is IMPLEMENTATION DEFINED whether ERRSELR_EL1 is UNDEFINED or RES0.

**Attributes**

ERRSELR_EL1 is a 64-bit register.

**Field descriptions**

![Field descriptions diagram]

Bits [63:16]

Reserved, RES0.

SEL, bits [15:0]

Selects the error record accessed through the ERX registers.

For example, if ERRSELR_EL1.SEL is 0x0004, then direct reads and writes of ERXSTATUS_EL1 access ERR4STATUS.

If ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then all of the following apply:

- The value read back from ERRSELR_EL1.SEL is UNKNOWN.
- One of the following occurs:
  - An UNKNOWN error record is selected.
  - The ERX*_EL1 registers are RAZ/WI.
  - ERX*_EL1 register reads and writes are NOPs.
  - ERX*_EL1 register reads and writes are UNDEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing ERRSELR_EL1**

Accesses to this register use the following encodings in the System register encoding space:
\textit{MRS \textlangle X\rangle, ERRSEL_EL1}

\begin{center}
\begin{tabular}{c c c c c}
\hline
op0 & op1 & CRn & CRm & op2 \\
\hline
0b11 & 0b000 & 0b0101 & 0b0011 & 0b001 \\
\end{tabular}
\end{center}

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.TERR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.ERRSEL_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = ERRSEL_EL1;
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = ERRSEL_EL1;
  end
elsif PSTATE.EL == EL3 then
  X[t, 64] = ERRSEL_EL1;
\end{verbatim}

\textit{MSR ERRSEL_EL1, \textlangle X\rangle}

\begin{center}
\begin{tabular}{c c c c c}
\hline
op0 & op1 & CRn & CRm & op2 \\
\hline
0b11 & 0b000 & 0b0101 & 0b0011 & 0b001 \\
\end{tabular}
\end{center}

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.TERR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.ERRSEL_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    ERRSEL_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
else
    ERRSELR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    ERRSELR_EL1 = X[t, 64];
D17.9.4 ERXADDR_EL1, Selected Error Record Address Register

The ERXADDR_EL1 characteristics are:

Purpose

Accesses ERR<\text{n}>ADDR for the error record <n> selected by ERRSELR_EL1.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

Configurations

AArch64 System register ERXADDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERXADDR[31:0].

AArch64 System register ERXADDR_EL1 bits [63:32] are architecturally mapped to AArch32 System register ERXADDR2[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXADDR_EL1 are UNDEFINED.

Attributes

ERXADDR_EL1 is a 64-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR&lt;\text{n}&gt;ADDR</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR&lt;\text{n}&gt;ADDR</td>
<td></td>
</tr>
</tbody>
</table>

Bits [63:0]

ERXADDR_EL1 accesses ERR<\text{n} ADDR, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXADDR_EL1

If ERRIDR_EL1.NUM is 0x0000 or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXADDR_EL1 is RAZ/WI.
- Direct reads and writes of ERXADDR_EL1 are NOPs.
- Direct reads and writes of ERXADDR_EL1 are UNDEFINED.

ERR<\text{n}>ADDR describes additional constraints that also apply when ERR<\text{n}>ADDR is accessed through ERXADDR_EL1.

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, ERXADDR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.TERR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) | SCR_EL3.FGTEn == '1') && HFGWTR_EL2.ERXADDR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  end
  X[t, 64] = ERXADDR_EL1;
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  end
  XR[t, 64] = ERXADDR_EL1;
elsif PSTATE_EL == EL3 then
  XR[t, 64] = ERXADDR_EL1;

MSR ERXADDR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.TERR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (HaveEL(EL3) | SCR_EL3.FGTEn == '1') && HFGWTR_EL2.ERXADDR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  end
  ERXADDR_EL1 = XR[t, 64];
elsif PSTATE_EL == EL2 then

if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
  UNDEFINED;
elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    ERXADDR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
  ERXADDR_EL1 = X[t, 64];
### ERXCTLR_EL1, Selected Error Record Control Register

The ERXCTLR_EL1 characteristics are:

**Purpose**

Accesses ERR<\text{n}>CTRL for the error record \text{n} selected by ERRSELR_EL1.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

AArch64 System register ERXCTLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERXCTRL[31:0].

AAarch64 System register ERXCTRL_EL1 bits [63:32] are architecturally mapped to AArch32 System register ERXCTRL2[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXCTLR_EL1 are UNDEFINED.

**Attributes**

ERXCTRL_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR&lt;\text{n}&gt;CTRL</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR&lt;\text{n}&gt;CTRL</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:0]**

ERXCTRL_EL1 accesses ERR<\text{n}>CTRL, where \text{n} is the value in ERRSELR_EL1.SEL.

**Accessing ERXCTRL_EL1**

If ERRIDR_EL1.NUM is 0x0000 or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An **UNKNOWN** error record is selected.
- ERXCTRL_EL1 is RAZ/WI.
- Direct reads and writes of ERXCTRL_EL1 are NOPs.
- Direct reads and writes of ERXCTRL_EL1 are UNDEFINED.

If ERRSELR_EL1.SEL is not the index of the first error record owned by a node, then ERR<\text{n}>CTRL is not present, meaning reads and writes of ERXCTRL_EL1 are **RES0**.

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, ERXCTLR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsf PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
        UNDEFINED;
elsf EL2Enabled() && HCR_EL2.TERR == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
elsf EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.ERXCTLR_EL1 == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
elsf HaveEL(EL3) && SCR_EL3.TERR == '1' then
                if Halted() && EDSCR.SDD == '1' then
                    UNDEFINED;
elsf EL2Enabled() && EL2Enabled() && SCR_EL3.TERR == '1' then
                    if Halted() && EDSCR.SDD == '1' then
                        UNDEFINED;
elsf EL2Enabled() && EL2Enabled() && SCR_EL3.TERR == '1' then
                        if Halted() && EDSCR.SDD == '1' then
                            UNDEFINED;
elsf HaveEL(EL3) && SCR_EL3.TERR == '1' then
                            if Halted() && EDSCR.SDD == '1' then
                                AArch64.SystemAccessTrap(EL3, 0x18);
elsf PSTATE.EL == EL2 then
    if HaveEL(EL3) && SCR_EL3.TERR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
elsf HaveEL(EL3) && SCR_EL3.TERR == '1' then
            if Halted() && EDSCR.SDD == '1' then
                AArch64.SystemAccessTrap(EL3, 0x18);
elsf PSTATE.EL == EL3 then
    X[t, 64] = ERXCTLR_EL1;
elsf PSTATE.EL == EL1 then
    X[t, 64] = ERXCTLR_EL1;
elsf PSTATE.EL == EL2 then
    X[t, 64] = ERXCTLR_EL1;
elsf PSTATE.EL == EL3 then
    X[t, 64] = ERXCTLR_EL1;

MSR ERXCTLR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsf PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
        UNDEFINED;
elsf EL2Enabled() && HCR_EL2.TERR == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
elsf EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.ERXCTLR_EL1 == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
elsf HaveEL(EL3) && SCR_EL3.TERR == '1' then
                if Halted() && EDSCR.SDD == '1' then
                    UNDEFINED;
elsf EL2Enabled() && EL2Enabled() && SCR_EL3.TERR == '1' then
                    if Halted() && EDSCR.SDD == '1' then
                        UNDEFINED;
elsf EL2Enabled() && EL2Enabled() && SCR_EL3.TERR == '1' then
                        if Halted() && EDSCR.SDD == '1' then
                            UNDEFINED;
elsf HaveEL(EL3) && SCR_EL3.TERR == '1' then
                            if Halted() && EDSCR.SDD == '1' then
                                AArch64.SystemAccessTrap(EL3, 0x18);
elsf PSTATE.EL == EL2 then
    else
        ERXCTLR_EL1 = X[t, 64];
elsf PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && SCR_EL3.TERR == '1' then
  UNDEFINED;
elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
else
  ERXCTR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
  ERXCTR_EL1 = X[t, 64];
D17.9.6 ERXFR_EL1, Selected Error Record Feature Register

The ERXFR_EL1 characteristics are:

**Purpose**

Accesses ERR<\text{n}>FR for the error record <\text{n}> selected by ERRSEL_EL1.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

AArch64 System register ERXFR_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERXFR[31:0].

AArch64 System register ERXFR_EL1 bits [63:32] are architecturally mapped to AArch32 System register ERXFR2[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXFR_EL1 are UNDEFINED.

**Attributes**

ERXFR_EL1 is a 64-bit register.

**Field descriptions**

![Field Description Diagram](image)

**Bits [63:0]**

ERXFR_EL1 accesses ERR<\text{n}>FR, where <\text{n}> is the value in ERRSEL_EL1.SEL.

**Accessing ERXFR_EL1**

If ERRIDR_EL1.NUM is 0x0000 or ERRSEL_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An **UNKNOWN** error record is selected.
- ERXFR_EL1 is RAZ.
- Direct reads of ERXFR_EL1 are NOPs.
- Direct reads of ERXFR_EL1 are **UNDEFINED**.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ERXFR_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then

if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
elsif EL2Enabled() && HCR_EL2.TERR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.ERXFR_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        X[t, 64] = ERXFR_EL1;
    end
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        X[t, 64] = ERXFR_EL1;
    end
elsif PSTATE.EL == EL3 then
    X[t, 64] = ERXFR_EL1;
D17.9.7  ERXMISC0_EL1, Selected Error Record Miscellaneous Register 0

The ERXMISC0_EL1 characteristics are:

Purpose

Accesses ERR<n>MISC0 for the error record <n> selected by ERRSELR_EL1.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

Configurations

AArch64 System register ERXMISC0_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERXMISC0[31:0].

AArch64 System register ERXMISC0_EL1 bits [63:32] are architecturally mapped to AArch32 System register ERXMISC1[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXMISC0_EL1 are UNDEFINED.

Attributes

ERXMISC0_EL1 is a 64-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR&lt;n&gt;MISC0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR&lt;n&gt;MISC0</td>
<td></td>
</tr>
</tbody>
</table>

Bits [63:0]

ERXMISC0_EL1 accesses ERR<n>MISC0, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXMISC0_EL1

If ERRIDR_EL1.NUM is 0x0000 or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC0_EL1 is RAZ/WI.
- Direct reads and writes of ERXMISC0_EL1 are NOPs.
- Direct reads and writes of ERXMISC0_EL1 are UNDEFINED.

ERR<n>MISC0 describes additional constraints that also apply when ERR<n>MISC0 is accessed through ERXMISC0_EL1.

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, ERXMISCO_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.TERR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.ERXMIScn_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
else
  X[t, 64] = ERXMISCO_EL1;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = ERXMISCO_EL1;
  end
elsif PSTATE.EL == EL3 then
  X[t, 64] = ERXMISCO_EL1;

MSR ERXMISCO_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.TERR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.ERXMIScn_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    ERXMISCO_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL2 then
if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & SCR_EL3.TERR == '1' then
  UNDEFINED;
elsif HaveEL(EL3) & SCR_EL3.TERR == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    ERXMISC0_EL1 = X[t, 64];
  elseif PSTATE.EL == EL3 then
    ERXMISC0_EL1 = X[t, 64];
D17.9.8 ERXMISC1_EL1, Selected Error Record Miscellaneous Register 1

The ERXMISC1_EL1 characteristics are:

**Purpose**

Accesses ERR<\(n\)>MISC1 for the error record \(<n\>\) selected by ERRSELR_EL1.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

AArch64 System register ERXMISC1_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERXMISC2[31:0].

AArch64 System register ERXMISC1_EL1 bits [63:32] are architecturally mapped to AArch32 System register ERXMISC3[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXMISC1_EL1 are UNDEFINED.

**Attributes**

ERXMISC1_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>ERR&lt;(n)&gt;MISC1</td>
</tr>
<tr>
<td>31</td>
<td>ERR&lt;(n)&gt;MISC1</td>
</tr>
</tbody>
</table>

**Bits [63:0]**

ERXMISC1_EL1 accesses ERR<\(n\)>MISC1, where \(<n\>\) is the value in ERRSELR_EL1.SEL.

**Accessing ERXMISC1_EL1**

If ERRIDR_EL1.NUM is 0x0000 or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An **UNKNOWN** error record is selected.
- ERXMISC1_EL1 is RAZ/WI.
- Direct reads and writes of ERXMISC1_EL1 are NOPs.
- Direct reads and writes of ERXMISC1_EL1 are UNDEFINED.

ERR<\(n\)>MISC1 describes additional constraints that also apply when ERR<\(n\)>MISC1 is accessed through ERXMISC1_EL1.

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, ERXMISC1_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
UNDEFINED;
elsif EL2Enabled() && HCR_EL2.TERR == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.ERXMISCn_EL1 == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = ERXMISC1_EL1;
elsif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
UNDEFINED;
elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
X[t, 64] = ERXMISC1_EL1;
elsif PSTATE.EL == EL3 then
X[t, 64] = ERXMISC1_EL1;

MSR ERXMISC1_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
UNDEFINED;
elsif EL2Enabled() && HCR_EL2.TERR == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.ERXMISCn_EL1 == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.SystemAccessTrap(EL3, 0x18);
else
ERXMISC1_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end
else
    if PSTATE.EL == EL3 then
        ERXMISC1_EL1 = X[t, 64];
    elsif PSTATE.EL == EL3 then
        ERXMISC1_EL1 = X[t, 64];
D17.9.9 ERXMISC2_EL1, Selected Error Record Miscellaneous Register 2

The ERXMISC2_EL1 characteristics are:

**Purpose**

Accesses ERR<\text{n}>&MISC2 for the error record <\text{n}> selected by ERRSELIR_EL1.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

AArch64 System register ERXMISC2_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERXMISC4[31:0].

AArch64 System register ERXMISC2_EL1 bits [63:32] are architecturally mapped to AArch32 System register ERXMISC5[31:0].

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXMISC2_EL1 are UNDEFINED.

**Attributes**

ERXMISC2_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR&lt;\text{n}&gt;MISC2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR&lt;\text{n}&gt;MISC2</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:0]**

ERXMISC2_EL1 accesses ERR<\text{n}>MISC2, where <\text{n}> is the value in ERRSELIR_EL1.SEL.

**Accessing ERXMISC2_EL1**

If ERRIDR_EL1.NUM is 0x00000 or ERRSELIR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC2_EL1 is RAZ/WI.
- Direct reads and writes of ERXMISC2_EL1 are NOPs.
- Direct reads and writes of ERXMISC2_EL1 are UNDEFINED.

ERR<\text{n}>MISC2 describes additional constraints that also apply when ERR<\text{n}>MISC2 is accessed through ERXMISC2_EL1.

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, ERXMISC2_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.TERR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.ERXMISCn_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = ERXMISC2_EL1;
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = ERXMISC2_EL1;
  end
elsif PSTATE.EL == EL3 then
  X[t, 64] = ERXMISC2_EL1;

MSR ERXMISC2_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.TERR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.ERXMISCn_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    ERXMISC2_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL2 then

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.TERR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.ERXMISCn_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    ERXMISC2_EL1 = X[t, 64];
  end
elsif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
    end if
else
    ERXMISC2_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    ERXMISC2_EL1 = X[t, 64];
else
    AArch64.SystemAccessTrap(EL3, 0x18);
D17.9.10 ERXMISC3_EL1, Selected Error Record Miscellaneous Register 3

The ERXMISC3_EL1 characteristics are:

**Purpose**

Accesses ERR<\text{n}>MISC3 for the error record <\text{n}> selected by ERRSELR_EL1.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

AArch64 System register ERXMISC3_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERXMISC6[31:0].

AArch64 System register ERXMISC3_EL1 bits [63:32] are architecturally mapped to AArch32 System register ERXMISC7[31:0].

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXMISC3_EL1 are UNDEFINED.

**Attributes**

ERXMISC3_EL1 is a 64-bit register.

**Field descriptions**

![Field Description Diagram]

Bits [63:0]

ERXMISC3_EL1 accesses ERR<\text{n}>MISC3, where <\text{n}> is the value in ERRSELR_EL1.SEL.

**Accessing ERXMISC3_EL1**

If ERRIDR_EL1.NUM is 0x0000 or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC3_EL1 is RAZ/WI.
- Direct reads and writes of ERXMISC3_EL1 are NOPs.
- Direct reads and writes of ERXMISC3_EL1 are UNDEFINED.

ERR<\text{n}>MISC3 describes additional constraints that also apply when ERR<\text{n}>MISC3 is accessed through ERXMISC3_EL1.

Accesses to this register use the following encodings in the System register encoding space:
MRS \(<X_t>, \text{ERXMISC3_EL1}\)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.TERR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGTR_EL2.ERXMISCn_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
    else
      X[t, 64] = ERXMISC3_EL1;
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
      UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.SystemAccessTrap(EL3, 0x18);
      end
    else
      X[t, 64] = ERXMISC3_EL1;
  elsif PSTATE.EL == EL3 then
    X[t, 64] = ERXMISC3_EL1;

MSR \(\text{ERXMISC3_EL1, } <X_t>\)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.TERR == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGTR_EL2.ERXMISCn_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    ERXMISC3_EL1 = X[t, 64];
  elsif PSTATE.EL == EL2 then

if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
  UNDEFINED;
elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  else
    ERX_MISC3_EL1 = X[t, 64];
  else if PSTATE.EL == EL3 then
    ERX_MISC3_EL1 = X[t, 64];
D17.9.11 ERXPFGCDN_EL1, Selected Pseudo-fault Generation Countdown register

The ERXPFGCDN_EL1 characteristics are:

Purpose

Accesses ERR<\textless n\textgreater>PFGCDN for the error record <\textless n\textgreater> selected by ERRSEL_EL1.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

Configurations

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXPFGCDN_EL1 are UNDEFINED.

Attributes

ERXPFGCDN_EL1 is a 64-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>ERR&lt;\textless n\textgreater&gt;PFGCDN</td>
</tr>
<tr>
<td>31-0</td>
<td>ERR&lt;\textless n\textgreater&gt;PFGCDN</td>
</tr>
</tbody>
</table>

Bits [63:0]

ERXPFGCDN_EL1 accesses ERR<\textless n\textgreater>PFGCDN, where <\textless n\textgreater> is the value in ERRSEL_EL1.SEL.

Accessing ERXPFGCDN_EL1

If ERRIDR_EL1.NUM is 0x0000 or ERRSEL_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXPFGCDN_EL1 is RAZ/WI.
- Direct reads and writes of ERXPFGCDN_EL1 are NOPs.
- Direct reads and writes of ERXPFGCDN_EL1 are UNDEFINED.

If ERRSEL_EL1.SEL selects an error record owned by a node that does not implement the Common Fault Injection Model Extension, then one of the following occurs:

- ERXPFGCDN_EL1 is RAZ/WI.
- Direct reads and writes of ERXPFGCDN_EL1 are NOPs.
- Direct reads and writes of ERXPFGCDN_EL1 are UNDEFINED.

Note

A node does not implement the Common Fault Injection Model Extension if ERR<\textless n\textgreater>FR.INJ reads as 0b00. <\textless q\textgreater> is the index of the first error record owned by the same node as error record <\textless n\textgreater>, where <\textless n\textgreater> is the value in ERRSEL_EL1.SEL. If the node owns a single record, then q = n.

If ERRSEL_EL1.SEL is not the index of the first error record owned by a node, then ERR<\textless n\textgreater>PFGCDN is not present, meaning reads and writes of ERXPFGCDN_EL1 are RES0.
ERR<n>PFGCDN describes additional constraints that also apply when ERR<n>PFGCDN is accessed through ERXPFGCDN_EL1.

Accesses to this register use the following encodings in the System register encoding space:

### MRS <Xt>, ERXPFGCDN_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & SCR_EL3.FIEN == '0' then
        UNDEFINED;
    elsif EL2Enabled() & HCR_EL2.FIEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFCWTR_EL2.ERXPFGCDN_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) & SCR_EL3.FIEN == '0' then
        if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end if
    elsif PSTATE.EL == EL2 then
        if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & SCR_EL3.FIEN == '0' then
            UNDEFINED;
        elsif HaveEL(EL3) & SCR_EL3.FIEN == '0' then
            if Halted() & EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end if
        elseif PSTATE.EL == EL3 then
            X[t, 64] = ERXPFGCDN_EL1;
        end if
    elseif PSTATE.EL == EL3 then
        X[t, 64] = ERXPFGCDN_EL1;
    end if

### MSR ERXPFGCDN_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & SCR_EL3.FIEN == '0' then
        UNDEFINED;
    elsif EL2Enabled() & HCR_EL2.FIEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFCWTR_EL2.ERXPFGCDN_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) & SCR_EL3.FIEN == '0' then
        if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end if
    elseif PSTATE.EL == EL2 then
        if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & SCR_EL3.FIEN == '0' then
            UNDEFINED;
        elsif HaveEL(EL3) & SCR_EL3.FIEN == '0' then
            if Halted() & EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end if
        elseif PSTATE.EL == EL3 then
            X[t, 64] = ERXPFGCDN_EL1;
        end if
    elseif PSTATE.EL == EL3 then
        X[t, 64] = ERXPFGCDN_EL1;
    end if
UNDEFINED;
else
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  ERXPFGCDN_EL1 = X[t, 64];
else if PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.FIEN == '0' then
    UNDEFINED;
else if HaveEL(EL3) && SCR_EL3.FIEN == '0' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
else
  ERXPFGCDN_EL1 = X[t, 64];
else if PSTATE_EL == EL3 then
  ERXPFGCDN_EL1 = X[t, 64];
### ERXPFGCTL_EL1, Selected Pseudo-fault Generation Control register

The ERXPFGCTL_EL1 characteristics are:

#### Purpose

Accesses ERR<\text{n}>PFGCTL for the error record \(<\text{n}>\) selected by ERRSELR_EL1.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

#### Configurations

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXPFGCTL_EL1 are UNDEFINED.

#### Attributes

ERXPFGCTL_EL1 is a 64-bit register.

#### Field descriptions

![Field descriptions diagram]

#### Accessing ERXPFGCTL_EL1

If ERRIDR_EL1.NUM is 0x0000 or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An \text{UNKNOWN} error record is selected.
- ERXPFGCTL_EL1 is RAZ/WI.
- Direct reads and writes of ERXPFGCTL_EL1 are NOPs.
- Direct reads and writes of ERXPFGCTL_EL1 are UNDEFINED.

If ERRSELR_EL1.SEL selects an error record owned by a node that does not implement the Common Fault Injection Model Extension, then one of the following occurs:

- ERXPFGCTL_EL1 is RAZ/WI.
- Direct reads and writes of ERXPFGCTL_EL1 are NOPs.
- Direct reads and writes of ERXPFGCTL_EL1 are UNDEFINED.

#### Note

A node does not implement the Common Fault Injection Model Extension if \text{ERR<\text{n}>FR.INJ} reads as 0b00. \(<\text{q}>\) is the index of the first error record owned by the same node as error record \(<\text{n}>\), where \(<\text{n}>\) is the value in ERRSELR_EL1.SEL. If the node owns a single record, then \(q = n\).

If ERRSELR_EL1.SEL is not the index of the first error record owned by a node, then \text{ERR<\text{n}>PFGCTL} is not present, meaning reads and writes of ERXPFGCTL_EL1 are RES0.
ERR<n>PFGCTL describes additional constraints that also apply when ERR<n>PFGCTL is accessed through ERXPFGCTL_EL1.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, ERXPFGCTL_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & SCR_EL3.FIEN == '0' then
    UNDEFINED;
  elsif EL2Enabled() & HCR_EL2.FIEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFGWTR_EL2.ERXPFGCTL_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) & SCR_EL3.FIEN == '0' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = ERXPFGCTL_EL1;
  end
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & SCR_EL3.FIEN == '0' then
    UNDEFINED;
  elsif HaveEL(EL3) & SCR_EL3.FIEN == '0' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.SystemAccessTrap(EL3, 0x18);
    end
  else
    X[t, 64] = ERXPFGCTL_EL1;
  end
elsif PSTATE.EL == EL3 then
  X[t, 64] = ERXPFGCTL_EL1;

**MSR ERXPFGCTL_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & SCR_EL3.FIEN == '0' then
    UNDEFINED;
  elsif EL2Enabled() & HCR_EL2.FIEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & (HaveEL(EL3) || SCR_EL3.FGTEn == '1') & HFGWTR_EL2.ERXPFGCTL_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL3) & SCR_EL3.FIEN == '0' then
    if Halted() & EDSCR.SDD == '1' then

UNDEFINED;
else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
    ERXPFGCTL_EL1 = X[t, 64];
elif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' &&
        boolean IMPLEMENTATION_DEFINED "EL3 trap priority
        when SDD == '1'" &&
        SCR_EL3.FIEN == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.FIEN == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        UNDEFINED;
    else
        ERXPFGCTL_EL1 = X[t, 64];
elif PSTATE_EL == EL3 then
    ERXPFGCTL_EL1 = X[t, 64];
### ERXPFGF_EL1, Selected Pseudo-fault Generation Feature register

The ERXPFGF_EL1 characteristics are:

#### Purpose

Accesses ERR<\(n\)>PFGF for the error record \(n\) selected by ERRSELR_EL1.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

#### Configurations

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXPFGF_EL1 are UNDEFINED.

#### Attributes

ERXPFGF_EL1 is a 64-bit register.

#### Field descriptions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>ERXPFGF_EL1 accesses ERR&lt;(n)&gt;PFGF, where (n) is the value in ERRSELR_EL1.SEL.</td>
</tr>
<tr>
<td>32-31</td>
<td>ERR&lt;(n)&gt;PFGF</td>
</tr>
<tr>
<td>0</td>
<td>ERR&lt;(n)&gt;PFGF</td>
</tr>
</tbody>
</table>

#### Accessing ERXPFGF_EL1

If ERRIDR_EL1.NUM is \(0x0000\) or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXPFGF_EL1 is RAZ.
- Direct reads of ERXPFGF_EL1 are NOPs.
- Direct reads of ERXPFGF_EL1 are UNDEFINED.

If ERRSELR_EL1.SEL selects an error record owned by a node that does not implement the Common Fault Injection Model Extension, then one of the following occurs:

- ERXPFGF_EL1 is RAZ.
- Direct reads of ERXPFGF_EL1 are NOPs.
- Direct reads of ERXPFGF_EL1 are UNDEFINED.

#### Note

A node does not implement the Common Fault Injection Model Extension if ERR<\(n\)>FR.INJ reads as \(0b00\). \(q\) is the index of the first error record owned by the same node as error record \(n\), where \(n\) is the value in ERRSELR_EL1.SEL. If the node owns a single record, then \(q = n\).

If ERRSELR_EL1.SEL is not the index of the first error record owned by a node, then ERR<\(n\)>PFGF is not present, meaning reads of ERXPFGF_EL1 are RES0.
ERR<\texttt{PFGF} describes additional constraints that also apply when ERR<\texttt{PFGF} is accessed through ERXPFGF_EL1.

Accesses to this register use the following encodings in the System register encoding space:

\textbf{MRS <Xt>, ERXPFGF_EL1}

<table>
<thead>
<tr>
<th>\textbf{op0}</th>
<th>\textbf{op1}</th>
<th>\textbf{CRn}</th>
<th>\textbf{CRm}</th>
<th>\textbf{op2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b100</td>
</tr>
</tbody>
</table>

\begin{verbatim}
i if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.FIEN == '0' then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.FIEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && (!(HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFCR_EL2.ERXPFGF_EL1 == '1') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3.FIEN == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        end
    else
        X[t, 64] = ERXPFGF_EL1;
    elsif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.FIEN == '0' then
            UNDEFINED;
        elsif HaveEL(EL3) && SCR_EL3.FIEN == '0' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
            end
        else
            X[t, 64] = ERXPFGF_EL1;
        end
    elsif PSTATE.EL == EL3 then
        X[t, 64] = ERXPFGF_EL1;
    end
\end{verbatim}
D17.9.14 ERXSTATUS_EL1, Selected Error Record Primary Status Register

The ERXSTATUS_EL1 characteristics are:

Purpose

Accesses ERR<n>STATUS for the error record <n> selected by ERRSELR_EL1.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

Configurations

AArch64 System register ERXSTATUS_EL1 bits [31:0] are architecturally mapped to AArch32 System register ERXSTATUS[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXSTATUS_EL1 are UNDEFINED.

Attributes

ERXSTATUS_EL1 is a 64-bit register.

Field descriptions

Bits [63:0]

ERXSTATUS_EL1 accesses ERR<n>STATUS, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXSTATUS_EL1

If ERRIDR_EL1.NUM is 0x0000 or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

• An UNKNOWN error record is selected.
• ERXSTATUS_EL1 is RAZ/WI.
• Direct reads and writes of ERXSTATUS_EL1 are NOPs.
• Direct reads and writes of ERXSTATUS_EL1 are UNDEFINED.

ERR<n>STATUS describes additional constraints that also apply when ERR<n>STATUS is accessed through ERXSTATUS_EL1.

Accesses to this register use the following encodings in the System register encoding space:

\[ \text{MRS} \ <\Xt>, \text{ERXSTATUS_EL1} \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
  UNDEFINED;
elsif EL2Enabled() && HCR_EL2.TERR == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && ((HaveEL(EL3)) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.ERXSTATUS_EL1 == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif HaveEL(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
  end
else
  X[t, 64] = ERXSTATUS_EL1;
else PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.TERR == '1' then
    UNDEFINED;
elsif EL2Enabled() && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = ERXSTATUS_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = ERXSTATUS_EL1;
else
  MSR ERXSTATUS_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b010</td>
</tr>
</tbody>
</table>
elsif PSTATE.EL == EL3 then
    ERXSTATUS_EL1 = X[t, 64];
D17.9.15 VDISR_EL2, Virtual Deferred Interrupt Status Register

The VDISR_EL2 characteristics are:

**Purpose**
Records that a virtual SError interrupt has been consumed by an E58 instruction executed at EL1.
An indirect write to VDISR_EL2 made by an E58 instruction does not require an explicit synchronization operation for the value written to be observed by a direct read of one of the following registers occurring in program order after the E58 instruction:
- DISR_EL1.
- DISR.

**Configurations**
AArch64 System register VDISR_EL2 bits [31:0] are architecturally mapped to AArch32 System register VDISR[31:0].
This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to VDISR_EL2 are UNDEFINED.
If EL2 is not implemented, this register is RES0 from EL3.
This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**
VDISR_EL2 is a 64-bit register.

**Field descriptions**

*When EL1 is using AArch64:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Reset Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>RES0</td>
<td><strong>Reserved, RES0.</strong></td>
</tr>
</tbody>
</table>
| 31  | A, bit [31]                  | Set to 1 when an E58 instruction defers a virtual SError interrupt.  
The reset behavior of this field is:  
  - On a Warm reset, this field resets to an architecturally **UNKNOWN** value. |
| 30  | RES0                         | **Reserved, RES0.**                                                           |
| 24  | IDS, bit [24]                | The value copied from VSESR_EL2.IDS.  
The reset behavior of this field is:  
  - On a Warm reset, this field resets to an architecturally **UNKNOWN** value. |
| 23  | ISS, bits [23:0]             | The value copied from VSESR_EL2.ISS.                                          |
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**When EL1 is using AArch32 and VDISR_EL2.LPAE == 0:**

Bits [63:32]

Reserved, RES0.

A, bit [31]

Set to 1 when an E58 instruction defers a virtual SError interrupt.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:16]

Reserved, RES0.

AET, bits [15:14]

The value copied from VSESR_EL2.AET.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

Ext, bit [12]

The value copied from VSESR_EL2.ExT.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [11]

Reserved, RES0.

FS, bits [10, 3:0]

Fault status code. Set to \(0b10110\) when an E58 instruction defers a virtual SError interrupt.

\(0b10110\)  Asynchronous SError interrupt.

All other values are reserved.

The FS field is split as follows:

- FS[4] is VDISR_EL2[10].
- FS[3:0] is VDISR_EL2[3:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
LPAE, bit [9]

Format.
Set to TTBCR.EAE when an E58 instruction defers a virtual SError interrupt.

0b0 Using the Short-descriptor translation table format.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:4]

Reserved, RES0.

When EL1 is using AArch32 and VDISR_EL2.LPAE == 1:

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits [63:32]

Reserved, RES0.

A, bit [31]

Set to 1 when an E58 instruction defers a virtual SError interrupt.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:16]

Reserved, RES0.

AET, bits [15:14]

The value copied from VSESR_EL2.AET.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

ExT, bit [12]

The value copied from VSESR_EL2.ExT.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

LPAE, bit [9]

Format.
Set to TTBCR.EAE when an E58 instruction defers a virtual SError interrupt.

0b1 Using the Long-descriptor translation table format.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [8:6]**
Reserved, RES0.

**STATUS, bits [5:0]**
Fault status code. Set to 0b010001 when an ESB instruction defers a virtual SError interrupt.
0b010001 Asynchronous SError interrupt.
All other values are reserved.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing VDISR_EL2**
An indirect write to VDISR_EL2 made by an ESB instruction does not require an explicit synchronization operation for the value that is written to be observed by a direct read of one of the following registers occurring in program order after the ESB instruction:
- DISR_EL1.
- DISR.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, VDISR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1100</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```
if PSTATE_EL == EL0 then 
    UNDEFINED;
elsif PSTATE_EL == EL1 then 
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then 
        X[t, 64] = NVMem[0x500];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then 
        AArch64.SystemAccessTrap(EL2, 0x18);
    else 
        UNDEFINED;
    elsif PSTATE_EL == EL2 then 
        X[t, 64] = VDISR_EL2;
    elsif PSTATE_EL == EL3 then 
        X[t, 64] = VDISR_EL2;

MSR VDISR_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1100</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>
```

```
elsif EL2Enabled() && HCR_EL2.NV == '1' then
AArch64.SystemAccessTrap(EL2, @0x10);
else
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  VDISR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
  VDISR_EL2 = X[t, 64];

MRS <Xt>, DISR_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.AMO == '1' then
    X[t, 64] = VDISR_EL2;
  elsif HaveEL(EL3) && !Halted() && SCR_EL3.EA == '1' then
    X[t, 64] = Zeros(64);
  else
    X[t, 64] = DISR_EL1;
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && !Halted() && SCR_EL3.EA == '1' then
    X[t, 64] = Zeros(64);
  else
    X[t, 64] = DISR_EL1;
elsif PSTATE.EL == EL3 then
  X[t, 64] = DISR_EL1;

MSR DISR_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.AMO == '1' then
    VDISR_EL2 = X[t, 64];
  elsif HaveEL(EL3) && !Halted() && SCR_EL3.EA == '1' then
    return;
  else
    DISR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && !Halted() && SCR_EL3.EA == '1' then
    return;
  else
    DISR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
  DISR_EL1 = X[t, 64];
D17.9.16  VSESR_EL2, Virtual SError Exception Syndrome Register

The VSESR_EL2 characteristics are:

**Purpose**

Provides the syndrome value reported to software on taking a virtual SError interrupt exception to EL1, or on executing an ESB instruction at EL1.

When the virtual SError interrupt injected using HCR_EL2.VSE is taken to EL1 using AArch64, then the syndrome value is reported in ESR_EL1.

When the virtual SError interrupt injected using HCR_EL2.VSE is taken to EL1 using AArch32, then the syndrome value is reported in DFSR.\{AET, ExT\} and the remainder of DFSR is set as defined by VMSAv8-32. For more information, see Chapter G5 The AArch32 Virtual Memory System Architecture.

When the virtual SError interrupt injected using HCR_EL2.VSE is deferred by an ESB instruction, then the syndrome value is written to VDISR_EL2.

**Configurations**

AArch64 System register VSESR_EL2 bits [31:0] are architecturally mapped to AArch32 System register VDFSR[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to VSESR_EL2 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

VSESR_EL2 is a 64-bit register.

**Field descriptions**

*When EL1 is using AArch32:*

![Field Diagram]

**Bits [63:16]**

Reserved, RES0.

**AET, bits [15:14]**

When a virtual SError interrupt is taken to EL1 using AArch32, DFSR[15:14] is set to VSESR_EL2.AET.

When a virtual SError interrupt is deferred by an ESB instruction, VDISR_EL2[15:14] is set to VSESR_EL2.AET.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [13]**

Reserved, RES0.
ExT, bit [12]

When a virtual SError interrupt is taken to EL1 using AArch32, DFSR[12] is set to VSESR_EL2.ExT.

When a virtual SError interrupt is deferred by an ES$ instruction, VDISR_EL2[12] is set to VSESR_EL2.ExT.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:0]

Reserved, RES0.

When EL1 is using AArch64:

Bits [63:25]

Reserved, RES0.

IDS, bit [24]

When a virtual SError interrupt is taken to EL1 using AArch64, ESR_EL1[24] is set to VSESR_EL2.IDS.

When a virtual SError interrupt is deferred by an ES$ instruction, VDISR_EL2[24] is set to VSESR_EL2.IDS.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS, bits [23:0]

When a virtual SError interrupt is taken to EL1 using AArch64, ESR_EL1[23:0] is set to VSESR_EL2.ISS.

When a virtual SError interrupt is deferred by an ES$ instruction, VDISR_EL2[23:0] is set to VSESR_EL2.ISS.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VSESR_EL2

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, VSESR_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
   X[t, 64] = NVMem[0x508];
elsif EL2Enabled() && HCR_EL2.NV == '1' then
   AArch64.SystemAccessTrap(EL2, 0x18);
else
   UNDEFINED;
elsif PSTATE.EL == EL2 then
   X[t, 64] = VSESR_EL2;
elsif PSTATE.EL == EL3 then
   X[t, 64] = VSESR_EL2;

**MSR VSESR_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
   UNDEFINED;
elsif PSTATE.EL == EL1 then
   if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
      NVMem[0x508] = X[t, 64];
   elsif EL2Enabled() && HCR_EL2.NV == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
   else
      UNDEFINED;
   elsif PSTATE.EL == EL2 then
      VSESR_EL2 = X[t, 64];
   else if PSTATE.EL == EL3 then
      VSESR_EL2 = X[t, 64];
D17.10  Realm Management Extension registers

This section lists the Realm Management Extension registers in AArch64.
D17.10.1 GPCCR_EL3, Granule Protection Check Control Register (EL3)

The GPCCR_EL3 characteristics are:

**Purpose**

The control register for Granule Protection Checks.

**Configurations**

This register is present only when FEAT_RME is implemented. Otherwise, direct accesses to GPCCR_EL3 are UNDEFINED.

**Attributes**

GPCCR_EL3 is a 64-bit register.

**Field descriptions**

![Register Diagram]

**Bits [63:24]**

Reserved, RES0.

**L0GPTSZ, bits [23:20]**

Level 0 GPT entry size.

This field advertises the number of least-significant address bits protected by each entry in the level 0 GPT.

- **0b0000**: 30-bits. Each entry covers 1GB of address space.
- **0b0100**: 34-bits. Each entry covers 16GB of address space.
- **0b0110**: 36-bits. Each entry covers 64GB of address space.
- **0b1001**: 39-bits. Each entry covers 512GB of address space.

All other values are reserved.

Access to this field is RO.

**Bits [19:18]**

Reserved, RES0.

**GPCP, bit [17]**

Granule Protection Check Priority.

This control governs behavior of granule protection checks on fetches of stage 2 Table descriptors.

- **0b0**: GPC faults are all reported with a priority that is consistent with the GPC being performed on any access to physical address space.
- **0b1**: A GPC fault for the fetch of a Table descriptor for a stage 2 translation table walk might not be generated or reported.

All other GPC faults are reported with a priority consistent with the GPC being performed on all accesses to physical address spaces.

This bit is permitted to be cached in a TLB.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GPC, bit [16]
Granule Protection Check Enable.
0b0 Granule protection checks are disabled. Accesses are not prevented by this mechanism.
0b1 All accesses to physical address spaces are subject to granule protection checks, except
for fetches of GPT information and accesses governed by the GPCCR_EL3.GPCP
control.

If any stage of translation is enabled, this bit is permitted to be cached in a TLB.
The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

PGS, bits [15:14]
Physical Granule size.
0b00 4KB.
0b01 64KB.
0b10 16KB.
All other values are reserved.
The value of this field is permitted to be cached in a TLB.
Granule sizes not supported for stage 1 and not supported for stage 2, as defined in
ID_AA64MMFR0_EL1, are reserved. For example, if ID_AA64MMFR0_EL1.TGran16 == 0b0000
and ID_AA64MMFR0_EL1.TGran16_2 == 0b0001, then the PGS encoding 0b10 is reserved.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SH, bits [13:12]
GPT fetch Shareability attribute
0b00 Non-shareable.
0b10 Outer Shareable.
0b11 Inner Shareable.
All other values are reserved.
Fetches of GPT information are made with the Shareability attribute that is configured in this field.
If both ORGN and IRGN are configured with Non-cacheable attributes, it is invalid to configure this
field to any value other than 0b10.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ORGN, bits [11:10]
GPT fetch Outer cacheability attribute.
0b00 Normal memory, Outer Non-cacheable.
0b01 Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
0b10 Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11 Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.
Fetches of GPT information are made with the Outer cacheability attributes configured in this field.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
IRGN, bits [9:8]

GPT fetch Inner cacheability attribute.

0b00  Normal memory, Inner Non-cacheable.
0b01  Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
0b10  Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11  Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

Fetches of GPT information are made with the Inner cacheability attributes configured in this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

Bits [7:3]

Reserved, RES0.

PPS, bits [2:0]

Protected Physical Address Size.

The size of the memory region protected by GPTBR_EL3, in terms of the number of least-significant address bits.

0b000  32 bits, 4GB protected address space.
0b001  36 bits, 64GB protected address space.
0b010  40 bits, 1TB protected address space.
0b011  42 bits, 4TB protected address space.
0b100  44 bits, 16TB protected address space.
0b101  48 bits, 256TB protected address space.
0b110  52 bits, 4PB protected address space.

All other values are reserved.

Configuration of this field to a value exceeding the implemented physical address size is invalid.

The value of this field is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Accessing GPCCR_EL3**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRS <Xt>, GPCCR_EL3
\]

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b110 & 0b0010 & 0b0001 & 0b110 \\
\end{array}
\]

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    X[t, 64] = GPCCR_EL3;
**MSR GPCCR_EL3, \textless Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0010</td>
<td>0b0001</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  GPCCR_EL3 = X[t, 64];
D17.10.2 GPTBR_EL3, Granule Protection Table Base Register

The GPTBR_EL3 characteristics are:

Purpose

The control register for Granule Protection Table base address.

Configurations

This register is present only when FEAT_RME is implemented. Otherwise, direct accesses to GPTBR_EL3 are UNDEFINED.

Attributes

GPTBR_EL3 is a 64-bit register.

Field descriptions

Bits [63:40]

Reserved, RES0.

BADDR, bits [39:0]

Base address for the level 0 GPT.

This field represents bits [51:12] of the level 0 GPT base address.

The level 0 GPT is aligned in memory to the greater of:

• The size of the level 0 GPT in bytes.
• 4KB.

Bits [x:0] of the base address are treated as zero, where:

• \( x = \max(pps - 10gptsz + 2, 11) \)
• pps is derived from GPCCR_EL3.PPS as follows:

<table>
<thead>
<tr>
<th>GPCCR_EL3.PPS</th>
<th>pps</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000</td>
<td>32</td>
</tr>
<tr>
<td>0b001</td>
<td>36</td>
</tr>
<tr>
<td>0b010</td>
<td>40</td>
</tr>
<tr>
<td>0b011</td>
<td>42</td>
</tr>
<tr>
<td>0b100</td>
<td>44</td>
</tr>
<tr>
<td>0b101</td>
<td>48</td>
</tr>
<tr>
<td>0b110</td>
<td>52</td>
</tr>
</tbody>
</table>

• l0gptsz is derived from GPCCR_EL3.L0GPTSZ as follows:

<table>
<thead>
<tr>
<th>GPCCR_EL3.L0GPTSZ</th>
<th>l0gptsz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>30</td>
</tr>
<tr>
<td>0b0100</td>
<td>34</td>
</tr>
<tr>
<td>0b0110</td>
<td>36</td>
</tr>
<tr>
<td>0b1001</td>
<td>39</td>
</tr>
</tbody>
</table>

If x is greater than 11, then BADDR[x - 12:0] are RES0.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing GPTBR_EL3**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, GPTBR_EL3**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b110</td>
<td>0b0010</td>
<td>0b0001</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    X[t, 64] = GPTBR_EL3;
### MSR GPTBR_EL3, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b10</td>
<td>0b0010</td>
<td>0b0001</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  GPTBR_EL3 = X[t, 64];
D17.10.3 MFAR_EL3, PA Fault Address Register

The MFAR_EL3 characteristics are:

**Purpose**

Holds the faulting physical address for Granule Protection Check exceptions taken to EL3.

**Configurations**

This register is present only when FEAT_RME is implemented. Otherwise, direct accesses to MFAR_EL3 are UNDEFINED.

**Attributes**

MFAR_EL3 is a 64-bit register.

**Field descriptions**

An exception return at EL3 makes MFAR_EL3 UNKNOWN.

**NS, bit [63]**

Together with the NSE field, reports the physical address space of the access that triggered the Granule Protection Check exception.

<table>
<thead>
<tr>
<th>NSE</th>
<th>NS</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b0</td>
<td>Secure.</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>Non-secure.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>Root.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>Realm.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**NSE, bit [62]**

Together with the NS field, reports the physical address space of the access that triggered the Granule Protection Check exception.

For a description of the values derived by evaluating NS and NSE together, see MFAR_EL3.NS.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [61:52]**

Reserved, RES0.

**FPA[51:48], bits [51:48]**

*When FEAT_LPA is implemented:*

When FEAT_LPA is implemented, extension to FPA[47:12].
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**FPA[47:12], bits [47:12]**

Bits [47:12] of the faulting physical address.

For implementations with fewer than 48 physical address bits, the corresponding upper bits in this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [11:0]**

Reserved, RES0.

**Accessing MFAR_EL3**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, MFAR_EL3**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b11</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  X[t, 64] = MFAR_EL3;
```

**MSR MFAR_EL3, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b11</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  MFAR_EL3 = X[t, 64];
```
D17.11  Generic Timer registers

This section lists the Generic Timer registers in AArch64.
D17.11 CNTFRQ_EL0, Counter-timer Frequency register

The CNTFRQ_EL0 characteristics are:

**Purpose**

This register is provided so that software can discover the frequency of the system counter. It must be programmed with this value as part of system initialization. The value of the register is not interpreted by hardware.

**Configurations**

AArch64 System register CNTFRQ_EL0 bits [31:0] are architecturally mapped to AArch32 System register CNTFRQ[31:0].

**Attributes**

CNTFRQ_EL0 is a 64-bit register.

**Field descriptions**

**Bits [63:32]**

Reserved, RES0.

**Bits [31:0]**

Clock frequency. Indicates the system counter clock frequency, in Hz.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTFRQ_EL0**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, CNTFRQ_EL0**

```
if PSTATE.EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.<EL0PCTEN,EL0VCTEN> == '00' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL2.<EL0PCTEN,EL0VCTEN> == '00' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    X[t, 64] = CNTFRQ_EL0;
  elsif PSTATE.EL == EL1 then
    X[t, 64] = CNTFRQ_EL0;
  elsif PSTATE.EL == EL2 then
```

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>
X[t, 64] = CNTFRQ_EL0;
elsif PSTATE.EL == EL3 then
  X[t, 64] = CNTFRQ_EL0;

**MSR CNTFRQ_EL0, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>
D17.11.2 CNTHCTL_EL2, Counter-timer Hypervisor Control register

The CNTHCTL_EL2 characteristics are:

**Purpose**

Controls the generation of an event stream from the physical counter, and access from EL1 to the physical counter and the EL1 physical timer.

**Configurations**

AArch64 System register CNTHCTL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHCTL[31:0].

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

**Attributes**

CNTHCTL_EL2 is a 64-bit register.

**Field descriptions**

*When FEAT_VHE is implemented and HCR_EL2.E2H == 1:*

```
<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>61</th>
<th>60</th>
<th>59</th>
<th>58</th>
<th>57</th>
<th>56</th>
<th>55</th>
<th>54</th>
<th>53</th>
<th>52</th>
<th>51</th>
<th>50</th>
<th>49</th>
<th>48</th>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 51    | 50  | 49  | 48  | 47  | 46  | 45  | 44  | 43  | 42  | 41  | 40  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| CNTPMASK | CNTVMASK | EVNTIS | ELINNVCT | ELINVPCT | EL1TVCT | ECV | LVPTEN | LVPTEN | LVPTEN | LVPTEN | LVPTEN | EVNTEN | EVNTEN | EVNTEN | EVNTEN | EVNTEN | EVNTEN | EVNTEN | EVNTEN | EVNTEN | EVNTEN | EVNTEN | EVNTEN |

Bits [63:20]

Reserved, RES0.

**CNTPMASK, bit [19]**

*When FEAT_RME is implemented:*

| 0b0 | This control has no affect on CNTP_CTL_EL0.IMASK. |
| 0b1 | CNTP_CTL_EL0.IMASK behaves as if set to 1 for all purposes other than a direct read of the field. |

This bit is RES0 in Non-secure and Secure state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**CNTVMASK, bit [18]**

*When FEAT_RME is implemented:*

| 0b0 | This control has no affect on CNTV_CTL_EL0.IMASK. |
| 0b1 | CNTV_CTL_EL0.IMASK behaves as if set to 1 for all purposes other than a direct read of the field. |
This bit is RES0 in Non-secure and Secure state.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**  
Reserved, RES0.

**EVNTIS, bit [17]**

**When FEAT_ECV is implemented:**  
Controls the scale of the generation of the event stream.

- **0b0** The CNTHCTL_EL2.EVNTI field applies to CNTPCT_EL0[15:0].
- **0b1** The CNTHCTL_EL2.EVNTI field applies to CNTPCT_EL0[23:8].

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**  
Reserved, RES0.

**EL1NVVCT, bit [16]**

**When FEAT_ECV is implemented:**  
Traps EL1 accesses to the specified EL1 virtual timer registers using the EL02 descriptors to EL2, when EL2 is enabled for the current Security state.

- **0b0** This control does not cause any instructions to be trapped.
- **0b1** If ((HCR_EL2.E2H==1 && HCR_EL2.TGE==1) || HCR_EL2.NV2==0 || HCR_EL2.NV1==1 || HCR_EL2.NV==0), this control does not cause any instructions to be trapped.
  
If ((HCR_EL2.E2H==0 || HCR_EL2.NV2==1 && HCR_EL2.NV1==0 || HCR_EL2.NV==1), then EL1 accesses to CNTV_CTL_EL02 and CNTV_CVAL_EL02 are trapped to EL2.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**  
Reserved, RES0.

**EL1NVPCT, bit [15]**

**When FEAT_ECV is implemented:**  
Traps EL1 accesses to the specified EL1 physical timer registers using the EL02 descriptors to EL2, when EL2 is enabled for the current Security state.

- **0b0** This control does not cause any instructions to be trapped.
- **0b1** If ((HCR_EL2.E2H==1 && HCR_EL2.TGE==1) || HCR_EL2.NV2==0 || HCR_EL2.NV1==1 || HCR_EL2.NV==0), this control does not cause any instructions to be trapped.
  
If (HCR_EL2.E2H==0 || HCR_EL2.TGE==0) && HCR_EL2.NV2==1 && HCR_EL2.NV1==0 && HCR_EL2.NV==1, then EL1 accesses to CNTP_CTL_EL02 and CNTP_CVAL_EL02 are trapped to EL2.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.
This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**EL1TVCT, bit [14]**

*When FEAT_ECV is implemented:*
Traps EL0 and EL1 accesses to the EL1 virtual counter registers to EL2, when EL2 is enabled for the current Security state.

0b0 This control does not cause any instructions to be trapped.

0b1 If HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

If HCR_EL2.E2H is 0 or HCR_EL2.TGE is 0, then:
- In AArch64 state, traps EL0 and EL1 accesses to CNTVCT_EL0 to EL2, unless they are trapped by CNTKCTL_EL1.EL0VCTEN.
- In AArch32 state, traps EL0 and EL1 accesses to CNTVCT to EL2, unless they are trapped by CNTKCTL_EL1.EL0VCTEN or CNTKCTL.PL0VCTEN.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.
This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**EL1TVT, bit [13]**

*When FEAT_ECV is implemented:*
Traps EL0 and EL1 accesses to the EL1 virtual timer registers to EL2, when EL2 is enabled for the current Security state.

0b0 This control does not cause any instructions to be trapped.

0b1 If HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

If HCR_EL2.E2H is 0 or HCR_EL2.TGE is 0, then:
- In AArch64 state, traps EL0 and EL1 accesses to CNTV_CTL_EL0, CNTV_CVAL_EL0, and CNTV_TVAL_EL0 to EL2, unless they are trapped by CNTKCTL_EL1.EL0VTEN.
- In AArch32 state, traps EL0 and EL1 accesses to CNTV_CTL, CNTV_CVAL, and CNTV_TVAL to EL2, unless they are trapped by CNTKCTL_EL1.EL0VTEN or CNTKCTL.PL0VTEN.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.
This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.
ECV, bit [12]

When FEAT_ECV is implemented:

- Enables the Enhanced Counter Virtualization functionality registers.
- **0b0** Enhanced Counter Virtualization functionality is disabled.
- **0b1** When HCR_EL2.E2H, TGE == {1, 1} or SCR_EL3.NS, EEL2 == {0, 0}, then Enhanced Counter Virtualization functionality is disabled.

When SCR_EL3.NS or SCR_EL3.EEL2 are 1, and HCR_EL2.E2H or HCR_EL2.TGE are 0, then Enhanced Counter Virtualization functionality is enabled when EL2 is enabled for the current Security state. This means that:

- An MRS to CNTPCT_EL0 from either EL0 or EL1 that is not trapped will return the value (PCount<63:0> - CNTPOFF_EL2<63:0>).
- The EL1 physical timer interrupt is triggered when 
  \((\text{PCount}<63:0>-\text{CNTPOFF}_\text{EL2}<63:0>-\text{PCVal}<63:0>)\) is greater than or equal to 0. 
  PCount<63:0> is the physical count returned when CNTPCT_EL0 is read from EL2 or EL3. PCVal<63:0> is the EL1 physical timer compare value for this timer.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

- Reserved, RES0.

EL1PTEN, bit [11]

When HCR_EL2.TGE is 0, traps EL0 and EL1 accesses to the EL1 physical timer registers to EL2 when EL2 is enabled in the current Security state.

- **0b0** From AArch64 state: EL0 and EL1 accesses to the CNTP_CTL_EL0, CNTP_CVAL_EL0, and CNTP_TVAL_EL0 are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by CNTKCTL_EL1.EL0PTEN.
  From AArch32 state: EL0 and EL1 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by CNTKCTL_EL1.EL0PTEN or CNTKCTL.PL0PTEN.
- **0b1** This control does not cause any instructions to be trapped.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL1PCTEN, bit [10]

When HCR_EL2.TGE is 0, traps EL0 and EL1 accesses to the EL1 physical counter register to EL2 when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to CNTPCT_EL0 are trapped to EL2, reported using EC syndrome value 0x18.
- In AArch32 state, MRRC or MCRR accesses to CNTPCT are trapped to EL2, reported using EC syndrome value 0x04.

- **0b0** From AArch64 state: EL0 and EL1 accesses to the CNTPCT_EL0 are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by CNTKCTL_EL1.EL0PCTEN.
  From AArch32 state: EL0 and EL1 accesses to the CNTPCT are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by CNTKCTL_EL1.EL0PCTEN or CNTKCTL.PL0PCTEN.
- **0b1** This control does not cause any instructions to be trapped.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
EL0PTEN, bit [9]
When HCR_EL2.TGE is 0, this control does not cause any instructions to be trapped.
When HCR_EL2.TGE is 1, traps EL0 accesses to the physical timer registers to EL2.

0b0  EL0 using AArch64: EL0 accesses to the CNTP_CTL_EL0, CNTP_CVAL_EL0, and CNTP_TVAL_EL0 registers are trapped to EL2.
     EL0 using AArch32: EL0 accesses to the CNTP_CTL, CNTP_CVAL and CNTP_TVAL registers are trapped to EL2.
0b1  This control does not cause any instructions to be trapped.
The reset behavior of this field is:
  •  On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL0VTEN, bit [8]
When HCR_EL2.TGE is 0, this control does not cause any instructions to be trapped.
When HCR_EL2.TGE is 1, traps EL0 accesses to the virtual timer registers to EL2.

0b0  EL0 using AArch64: EL0 accesses to the CNTV_CTL_EL0, CNTV_CVAL_EL0, and CNTV_TVAL_EL0 registers are trapped to EL2.
     EL0 using AArch32: EL0 accesses to the CNTV_CTL, CNTV_CVAL, and CNTV_TVAL registers are trapped to EL2.
0b1  This control does not cause any instructions to be trapped.
The reset behavior of this field is:
  •  On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTI, bits [7:4]
Selects which bit of CNTPCT_EL0, as seen from EL2, is the trigger for the event stream generated from that counter when that stream is enabled.

If FEAT_ECV is implemented, and CNTHCTL_EL2.EVNTIS is 1, this field selects a trigger bit in the range 8 to 23 of CNTPCT_EL0.
Otherwise, this field selects a trigger bit in the range 0 to 15 of CNTPCT_EL0.
The reset behavior of this field is:
  •  On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTDIR, bit [3]
Controls which transition of the CNTPCT_EL0 trigger bit, as seen from EL2 and defined by EVNTI, generates an event when the event stream is enabled.

0b0  A 0 to 1 transition of the trigger bit triggers an event.
0b1  A 1 to 0 transition of the trigger bit triggers an event.
The reset behavior of this field is:
  •  On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTEN, bit [2]
Enables the generation of an event stream from CNTPCT_EL0 as seen from EL2.

0b0  Disables the event stream.
0b1  Enables the event stream.
The reset behavior of this field is:
  •  On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL0VCTEN, bit [1]
When HCR_EL2.TGE is 0, this control does not cause any instructions to be trapped.
When HCR_EL2.TGE is 1, traps EL0 accesses to the frequency register and virtual counter register to EL2.

0b0  EL0 using AArch64: EL0 accesses to the CNTVCT_EL0 are trapped to EL2.
    EL0 using AArch64: EL0 accesses to the CNTFRQ_EL0 register are trapped to EL2, if CNTHCTL_EL2.EL0PCTEN is also 0.
    EL0 using AArch32: EL0 accesses to the CNTVCT are trapped to EL2.
    EL0 using AArch32: EL0 accesses to the CNTFRQ register are trapped to EL2, if CNTHCTL_EL2.EL0PCTEN is also 0.

0b1  This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**EL0PCTEN, bit [0]**

When HCR_EL2.TGE is 0, this control does not cause any instructions to be trapped.

When HCR_EL2.TGE is 1, traps EL0 accesses to the frequency register and physical counter register to EL2.

0b0  EL0 using AArch64: EL0 accesses to the CNTPCT_EL0 are trapped to EL2.
    EL0 using AArch64: EL0 accesses to the CNTFRQ_EL0 register are trapped to EL2, if CNTHCTL_EL2.EL0VCTEN is also 0.
    EL0 using AArch32: EL0 accesses to the CNTPCT are trapped to EL2.
    EL0 using AArch32: EL0 accesses to the CNTFRQ and register are trapped to EL2, if CNTHCTL_EL2.EL0VCTEN is also 0.

0b1  This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

This format applies in all Armv8.0 implementations.

It also contains a description of the behavior when EL3 is implemented and EL2 is not implemented.

**Bits [63:20]**

Reserved, RES0.

**CNTPMASK, bit [19]**

*When FEAT_RME is implemented:*

0b0  This control has no affect on CNTP_CTL_EL0.IMASK.

0b1  CNTP_CTL_EL0.IMASK behaves as if set to 1 for all purposes other than a direct read of the field.

This bit is RES0 in Non-secure and Secure state.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**CNTVMASK, bit [18]**

When **FEAT_RME** is implemented:

- 0b0 This control has no affect on CNTV_CTL_EL0.IMASK.
- 0b1 CNTV_CTL_EL0.IMASK behaves as if set to 1 for all purposes other than a direct read of the field.

This bit is RES0 in Non-secure and Secure state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**EVNTIS, bit [17]**

When **FEAT_ECV** is implemented:

- Controls the scale of the generation of the event stream.
- 0b0 The CNTHCTL_EL2.EVNTI field applies to CNTPCT_EL0[15:0].
- 0b1 The CNTHCTL_EL2.EVNTI field applies to CNTPCT_EL0[23:8].

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**EL1NVVCT, bit [16]**

When **FEAT_ECV** is implemented:

- Traps EL1 accesses to the specified EL1 virtual timer registers using the EL02 descriptors to EL2, when EL2 is enabled for the current Security state.
- 0b0 This control does not cause any instructions to be trapped.
- 0b1 If ((HCR_EL2.E2H==1 && HCR_EL2.TGE==1) || HCR_EL2.NV2==0 || HCR_EL2.NV1==1 || HCR_EL2.NV==0), this control does not cause any instructions to be trapped.

If ((HCR_EL2.E2H==0 || HCR_EL2.TGE==0) && HCR_EL2.NV2==1 && HCR_EL2.NV1==0 && HCR_EL2.NV==1), then EL1 accesses to CNTV_CTL_EL02 and CNTV_CVAL_EL02 are trapped to EL2.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.
EL1NVPCT, bit [15]

When FEAT_ECV is implemented:
Traps EL1 accesses to the specified EL1 physical timer registers using the EL02 descriptors to EL2, when EL2 is enabled for the current Security state.

0b0  This control does not cause any instructions to be trapped.
0b1  If (HCR_EL2.E2H==1 & HCR_EL2.TGE==1) || HCR_EL2.NV2==0 ||
     HCR_EL2.NV1==1 || HCR_EL2.NV==0, this control does not cause any instructions
     to be trapped.

If (HCR_EL2.E2H==0 || HCR_EL2.TGE==0) && HCR_EL2.NV2==1 &&
     HCR_EL2.NV1==0 && HCR_EL2.NV==1, then EL1 accesses to CNTP_CTL_EL02
and CNTP_CVAL_EL02, are trapped to EL2.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EL1TVCT, bit [14]

When FEAT_ECV is implemented:
Traps EL0 and EL1 accesses to the EL1 virtual counter registers to EL2, when EL2 is enabled for the current Security state.

0b0  This control does not cause any instructions to be trapped.
0b1  If HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

If HCR_EL2.E2H is 0 or HCR_EL2.TGE is 0, then:
In AArch64 state, traps EL0 and EL1 accesses to CNTVCT_EL0 to EL2, unless they
are trapped by CNTKCTL_EL1.EL0VCTEN. In AArch32 state, traps EL0 and EL1
accesses to CNTVCT to EL2, unless they are trapped by CNTKCTL_EL1.EL0VCTEN
or CNTKCTL.PL0VCTEN.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EL1TVT, bit [13]

When FEAT_ECV is implemented:
Traps EL0 and EL1 accesses to the EL1 virtual timer registers to EL2, when EL2 is enabled for the current Security state.

0b0  This control does not cause any instructions to be trapped.
0b1  If HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

If HCR_EL2.E2H is 0 or HCR_EL2.TGE is 0, then:
• In AArch64 state, traps EL0 and EL1 accesses to CNTV_CTL_EL0,
  CNTV_CVAL_EL0, and CNTV_TVVAL_EL0 to EL2, unless they are trapped by
  CNTKCTL_EL1.EL0VTEN.
In AArch32 state, traps EL0 and EL1 accesses to CNTV_CTL, CNTV_CVAL, and CNTV_TVAL to EL2, unless they are trapped by CNTKCTL_EL1.EL0VTEN or CNTKCTL.PL0VTEN.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 0 other than for the purpose of a direct read.

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:

* On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

**ECV, bit [12]**

*When FEAT_ECV is implemented:*

Enables the Enhanced Counter Virtualization functionality registers.

0b0  Enhanced Counter Virtualization functionality is disabled.

0b1  When HCR_EL2.{E2H, TGE} == {1, 1} or SCR_EL3.{NS, EEL2} == {0, 0}, then Enhanced Counter Virtualization functionality is disabled.

When SCR_EL3.NS or SCR_EL3.EEL2 are 1, and HCR_EL2.E2H or HCR_EL2.TGE are 0, then Enhanced Counter Virtualization functionality is enabled when EL2 is enabled for the current Security state. This means that:

* An MRS to CNTPCT_EL0 from either EL0 or EL1 that is not trapped will return the value (PCount<63:0> - CNTPOFF_EL2<63:0>).

* The EL1 physical timer interrupt is triggered when ((PCount<63:0> - CNTPOFF_EL2<63:0> - PCVal<63:0>) is greater than or equal to 0. PCount is the physical count returned when CNTPCT_EL0 is read from EL2 or EL3. PCVal<63:0> is the EL1 physical timer compare value for this timer.

The reset behavior of this field is:

* On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

**Bits [11:8]**

Reserved, RES0.

**EVNTI, bits [7:4]**

Selects which bit of CNTPCT_EL0, as seen from EL2, is the trigger for the event stream generated from that counter when that stream is enabled.

If FEAT_ECV is implemented, and CNTHCTL_EL2_EVNTIS is 1, this field selects a trigger bit in the range 8 to 23 of CNTPCT_EL0.

Otherwise, this field selects a trigger bit in the range 0 to 15 of CNTPCT_EL0.

The reset behavior of this field is:

* On a Warm reset, this field resets to an architecturally UNKNOWN value.

**EVNTDIR, bit [3]**

Controls which transition of the CNTPCT_EL0 trigger bit, as seen from EL2 and defined by EVNTI, generates an event when the event stream is enabled.

0b0  A 0 to 1 transition of the trigger bit triggers an event.

0b1  A 1 to 0 transition of the trigger bit triggers an event.

The reset behavior of this field is:

* On a Warm reset, this field resets to an architecturally UNKNOWN value.
EVNTEN, bit [2]

Enables the generation of an event stream from CNTPCT_EL0 as seen from EL2.

0\text{b}0 \quad \text{Disables the event stream.}
0\text{b}1 \quad \text{Enables the event stream.}

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

EL1PCEN, bit [1]

Traps EL0 and EL1 accesses to the EL1 physical timer registers to EL2 when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to 
  \text{CNTP\_CTL\_EL0, CNTP\_CVAL\_EL0, CNTP\_TVAL\_EL0}
  are trapped to EL2, reported using EC syndrome value \text{0x18}.
- In AArch32 state, MRC or MCR accesses to the following registers are trapped to EL2 reported using EC syndrome value \text{0x3} and MRRC and MCRR accesses are trapped to EL2, reported using EC syndrome value \text{0x04}:
  \begin{itemize}
  \item \text{CNTP\_CTL, CNTP\_CVAL, CNTP\_TVAL}.
  \end{itemize}

0\text{b}0 \quad \text{From AArch64 state: EL0 and EL1 accesses to the CNTP\_CTL\_EL0, CNTP\_CVAL\_EL0, and CNTP\_TVAL\_EL0 are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by CNTKCTL\_EL1.EL0PTEN. From AArch32 state: EL0 and EL1 accesses to the CNTP\_CTL, CNTP\_CVAL, and CNTP\_TVAL are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by CNTKCTL\_EL1.EL0PTEN or CNTKCTL\_PL0PTEN.}
0\text{b}1 \quad \text{This control does not cause any instructions to be trapped.}

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

EL1PCTEN, bit [0]

Traps EL0 and EL1 accesses to the EL1 physical counter register to EL2 when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to \text{CNTPCT\_EL0} are trapped to EL2, reported using EC syndrome value \text{0x18}.
- In AArch32 state, MRRC or MCRR accesses to \text{CNTPCT} are trapped to EL2, reported using EC syndrome value \text{0x04}.

0\text{b}0 \quad \text{From AArch64 state: EL0 and EL1 accesses to the CNTPCT\_EL0 are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by CNTKCTL\_EL1.EL0PCTEN. From AArch32 state: EL0 and EL1 accesses to the CNTPCT are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by CNTKCTL\_EL1.EL0PCTEN or CNTKCTL\_PL0PCTEN.}
0\text{b}1 \quad \text{This control does not cause any instructions to be trapped.}

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
Accessing CNTHCTL_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHCTL_EL2 or CNTKCTL_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, CNTHCTL_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = CNTHCTL_EL2;
elsif PSTATE.EL == EL3 then
  X[t, 64] = CNTHCTL_EL2;

**MSR CNTHCTL_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  CNTHCTL_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
  CNTHCTL_EL2 = X[t, 64];

**MRS <Xt>, CNTKCTL_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  X[t, 64] = CNTKCTL_EL1;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
X[t, 64] = CNTHCTL_EL2;
else
    X[t, 64] = CNTKCTL_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = CNTKCTL_EL1;

**MSR CNTKCTL_EL1, <Xt>**

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    CNTKCTL_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CNTHCTL_EL2 = X[t, 64];
    else
        CNTKCTL_EL1 = X[t, 64];
    elsif PSTATE.EL == EL3 then
        CNTKCTL_EL1 = X[t, 64];

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b000</td>
<td>0b111</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    CNTKCTL_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CNTHCTL_EL2 = X[t, 64];
    else
        CNTKCTL_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    CNTKCTL_EL1 = X[t, 64];
D17.11.3 CNTHP_CTL_EL2, Counter-timer Hypervisor Physical Timer Control register

The CNTHP_CTL_EL2 characteristics are:

**Purpose**
Control register for the EL2 physical timer.

**Configurations**
AArch64 System register CNTHP_CTL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHP_CTL[31:0].

This register is present only when EL3 is implemented or (EL3 is not implemented, EL2 is implemented and FEAT_SEL2 is not implemented). Otherwise, direct accesses to CNTHP_CTL_EL2 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**
CNTHP_CTL_EL2 is a 64-bit register.

**Field descriptions**

![Field Diagram]

**Bits [63:3]**
Reserved, RES0.

**ISTATUS, bit [2]**
The status of the timer. This bit indicates whether the timer condition is met:
- **0b0**  Timer condition is not met.
- **0b1**  Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

**IMASK, bit [1]**
Timer interrupt mask bit. Permitted values are:
- **0b0**  Timer interrupt is not masked by the IMASK bit.
- **0b1**  Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
ENABLE, bit [0]

Enables the timer. Permitted values are:

0b0       Timer disabled.
0b1       Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHP_TVAL_EL2 continues to count down.

--- Note ---

Disabling the output signal might be a power-saving option.

---

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHP_CTL_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHP_CTL_EL2 or CNTP_CTL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, CNTHP_CTL_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b01</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE_EL == EL2 then
    X[t, 64] = CNTHP_CTL_EL2;
elsif PSTATE_EL == EL3 then
    X[t, 64] = CNTHP_CTL_EL2;
```

**MSR CNTHP_CTL_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE_EL == EL2 then
    CNTHP_CTL_EL2 = X[t, 64];
```
elsif PSTATE.EL == EL3 then
    CNTHP_CTL_EL2 = X[t, 64];

MRS <Xt>, CNTP_CTL_EL0

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if ((EL2Enabled() & HCR_EL2.<E2H,TGE> == '11') & CNTKCTL_EL1.EL0PTEN == '0') then
        if EL2Enabled() & HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        endif
    else
        if EL2Enabled() & HCR_EL2.E2H == '0' & CNTHTCTL_EL2.EL1PCEN == '0' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        endif
    endif
endif

if PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.E2H == '0' & CNTHTCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if EL2Enabled() & HCR_EL2.E2H == '1' & CNTHTCTL_EL2.EL1PTEN == '0' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            if EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
                X[t, 64] = NVMem[0x180];
            else
                X[t, 64] = CNTP_CTL_EL0;
            endif
        endif
    endif
endif

else if PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' & SCR_EL3.NS == '0' & IsFeatureImplemented(FEAT_SEL2) then
        X[t, 64] = CNTHPS_CTL_EL2;
    else if HCR_EL2.E2H == '1' & SCR_EL3.NS == '1' then
        X[t, 64] = CNTHP_CTL_EL2;
    else
        X[t, 64] = CNTP_CTL_EL0;
    endif
endif

else if PSTATE.EL == EL3 then
    if HCR_EL2.E2H == '1' & SCR_EL3.NS == '0' & IsFeatureImplemented(FEAT_SEL2) then
        X[t, 64] = CNTHPS_CTL_EL2;
    else if HCR_EL2.E2H == '1' & SCR_EL3.NS == '1' then
        X[t, 64] = CNTHP_CTL_EL2;
    else
        X[t, 64] = CNTP_CTL_EL0;
    endif
endif

MSR CNTP_CTL_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if ((EL2Enabled() & HCR_EL2.<E2H,TGE> == '11') & CNTKCTL_EL1.EL0PTEN == '0') then
        if EL2Enabled() & HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        endif
    else
        if EL2Enabled() & HCR_EL2.E2H == '0' & CNTHTCTL_EL2.EL1PCEN == '0' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        endif
    endif
endif

AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
  IsFeatureImplemented(FEAT_SEL2) then
  CNTHPS_CTL_EL2 = X\[t, 64\];
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
  CNTHP_CTL_EL2 = X\[t, 64\];
else
  CNTP_CTL_EL0 = X\[t, 64\];
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    NVMem[0x18] = X\[t, 64\];
  else
    CNTP_CTL_EL0 = X\[t, 64\];
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
    CNTHPS_CTL_EL2 = X\[t, 64\];
  elseif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
    CNTHP_CTL_EL2 = X\[t, 64\];
else
    CNTP_CTL_EL0 = X\[t, 64\];
elsif PSTATE.EL == EL3 then
  CNTP_CTL_EL0 = X\[t, 64\];
D17.11.4 CNTHP_CVAL_EL2, Counter-timer Physical Timer CompareValue register (EL2)

The CNTHP_CVAL_EL2 characteristics are:

**Purpose**

Holds the compare value for the EL2 physical timer.

**Configurations**

AArch64 System register CNTHP_CVAL_EL2 bits [63:0] are architecturally mapped to AArch32 System register CNTHP_CVAL[63:0].

This register is present only when EL3 is implemented or (EL3 is not implemented, EL2 is implemented and FEAT_SEL2 is not implemented). Otherwise, direct accesses to CNTHP_CVAL_EL2 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

CNTHP_CVAL_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CompareValue</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>CompareValue</td>
</tr>
</tbody>
</table>

**CompareValue, bits [63:0]**

Holds the EL2 physical timer CompareValue.

When CNTHP_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHP_CTL_EL2.ISTATUS is set to 1.
- If CNTHP_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHP_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTHP_CVAL_EL2**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHP_CVAL_EL2 or CNTP_CVAL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:
**MRS <Xt>, CNTHP_CVAL_EL2**

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
\hline
0b11 & 0b100 & 0b1110 & 0b0010 & 0b010 \\
\hline
\end{array}
\]

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = CNTHP_CVAL_EL2;
elsif PSTATE.EL == EL3 then
  X[t, 64] = CNTHP_CVAL_EL2;

**MSR CNTHP_CVAL_EL2, <Xt>**

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
\hline
0b11 & 0b100 & 0b1110 & 0b0010 & 0b010 \\
\hline
\end{array}
\]

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  CNTHP_CVAL_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
  CNTHP_CVAL_EL2 = X[t, 64];

**MRS <Xt>, CNTP_CVAL_EL0**

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
\hline
0b11 & 0b011 & 0b1110 & 0b0010 & 0b010 \\
\hline
\end{array}
\]

if PSTATE.EL == EL0 then
  if (!EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  elseif EL2Enabled() && HCR_EL2.E2H == '0' && CNTKCTL_EL2.E1PTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if EL2Enabled() && HCR_EL2.<E2H,TGE> == '10' && CNTKCTL_EL2.E1PTEN == '0' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL2.E0PTEN == '0' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
      IsFeatureImplemented(FEAT_SEL2) then
      X[t, 64] = CNTHPS_CVAL_EL2;
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
    X[t, 64] = CNTHP_CVAL_EL2;
else
    X[t, 64] = CNTP_CVAL_EL0;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.SystemAccessTrap(EL1, 0x18);
    endif
elsif EL2Enabled() && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x178];
else
    X[t, 64] = CNTP_CVAL_EL0;
elsif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
        X[t, 64] = CNTHPS_CVAL_EL2;
    elseif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        X[t, 64] = CNTHP_CVAL_EL2;
    else
        X[t, 64] = CNTP_CVAL_EL0;
    endif
elsif PSTATE_EL == EL3 then
    X[t, 64] = CNTP_CVAL_EL0;

MSR CNTP_CVAL_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>
elsif PSTATE_EL == EL3 then
  CNTP_CVAL_EL0 = X[t, 64];
D17.11.5 CNTHP_TVAL_EL2, Counter-timer Physical Timer TimerValue register (EL2)

The CNTHP_TVAL_EL2 characteristics are:

**Purpose**

Holds the timer value for the EL2 physical timer.

**Configurations**

AArch64 System register CNTHP_TVAL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHP_TVAL[31:0].

This register is present only when EL3 is implemented or (EL3 is not implemented, EL2 is implemented and FEAT_SEL2 is not implemented). Otherwise, direct accesses to CNTHP_TVAL_EL2 are **undefined**.

If EL2 is not implemented, this register is **RES0** from EL3.

**Attributes**

CNTHP_TVAL_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>RES0</td>
</tr>
<tr>
<td>31-0</td>
<td>TimerValue</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, **RES0**.

**TimerValue, bits [31:0]**

The TimerValue view of the EL2 physical timer.

On a read of this register:
- If CNTHP_CTL_EL2.ENABLE is 0, the value returned is **UNKNOWN**.
- If CNTHP_CTL_EL2.ENABLE is 1, the value returned is (CNTHP_CVVAL_EL2 - CNTPCT_EL0).

On a write of this register, CNTHP_CVVAL_EL2 is set to (CNTPCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHP_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CNTHP_CVVAL_EL2) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:
- CNTHP_CTL_EL2.ISTATUS is set to 1.
- If CNTHP_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHP_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Accessing CNTHP_TVAL_EL2**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHP_TVAL_EL2 or CNTP_TVAL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.
Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, CNTHP_TVAL_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if EL2Enabled() && HCR_EL2.NV == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
UNDEFINED;
elsif PSTATE.EL == EL2 then
if CNTHP_CTL_EL2.ENABLE == '0' then
X[t, 64] = bits(64) UNKNOWN;
else
X[t, 64] = CNTHP_CVAL_EL2 - PhysicalCountInt();
elsif PSTATE.EL == EL3 then
if CNTHP_CTL_EL2.ENABLE == '0' then
X[t, 64] = bits(64) UNKNOWN;
else
X[t, 64] = CNTHP_CVAL_EL2 - PhysicalCountInt();

**MSR CNTHP_TVAL_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if EL2Enabled() && HCR_EL2.NV == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
UNDEFINED;
elsif PSTATE.EL == EL2 then
if CNTHP_CTL_EL2.ENABLE == '0' then
X[t, 64] = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
elsif PSTATE.EL == EL3 then
if CNTHP_CTL_EL2.ENABLE == '0' then
X[t, 64] = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();

**MRS <Xt>, CNTP_TVAL_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0' then
if EL2Enabled() && HCR_EL2.TGE == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
  IsFeatureImplemented(FEAT_SEL2) then
  if CNTHPS_CTL_EL2.ENABLE == '0' then
    X[t, 64] = bits(64) UNKNOWN;
  else
    X[t, 64] = CNTHPS_CVAL_EL2 - PhysicalCountInt();
  end
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
  if CNTHP_CTL_EL2.ENABLE == '0' then
    X[t, 64] = bits(64) UNKNOWN;
  else
    X[t, 64] = CNTHP_CVAL_EL2 - PhysicalCountInt();
  end
elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && SCR_EL3.ECVEn == '1' &&
  CNTHCTL_EL2.ECV == '1' && HCR_EL2.<E2H,TGE> != '11' then
  if CNTP_CTL_EL0.ENABLE == '0' then
    X[t, 64] = bits(64) UNKNOWN;
  else
    X[t, 64] = CNTP_CVAL_EL0 - (PhysicalCountInt() - CNTPOFF_EL2);
  else
    if CNTP_CTL_EL0.ENABLE == '0' then
      X[t, 64] = bits(64) UNKNOWN;
    else
      X[t, 64] = CNTP_CVAL_EL0 - PhysicalCountInt();
    end
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && SCR_EL3.ECVEn == '1' &&
    CNTHCTL_EL2.ECV == '1' && HCR_EL2.<E2H,TGE> != '11' then
    if CNTP_CTL_EL0.ENABLE == '0' then
      X[t, 64] = bits(64) UNKNOWN;
    else
      X[t, 64] = CNTP_CVAL_EL0 - (PhysicalCountInt() - CNTPOFF_EL2);
    else
      if CNTP_CTL_EL0.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
      else
        X[t, 64] = CNTP_CVAL_EL0 - PhysicalCountInt();
      end
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' &&
    IsFeatureImplemented(FEAT_SEL2) then
    if CNTHPS_CTL_EL2.ENABLE == '0' then
      X[t, 64] = bits(64) UNKNOWN;
    else
      X[t, 64] = CNTHPS_CVAL_EL2 - PhysicalCountInt();
  elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
    if CNTHP_CTL_EL2.ENABLE == '0' then
      X[t, 64] = bits(64) UNKNOWN;
    else
      X[t, 64] = CNTHP_CVAL_EL2 - PhysicalCountInt();
  else
    if CNTP_CTL_EL0.ENABLE == '0' then
      X[t, 64] = bits(64) UNKNOWN;
    else
      X[t, 64] = CNTP_CVAL_EL0 - PhysicalCountInt();
  elsif PSTATE.EL == EL3 then
    if CNTP_CTL_EL0.ENABLE == '0' then
      X[t, 64] = bits(64) UNKNOWN;
    else
      X[t, 64] = CNTP_CVAL_EL0 - PhysicalCountInt();
else
  X[t, 64] = CNTP_CVAL_EL0 - PhysicalCountInt();
D17.11 Generic Timer registers

**MSR CNTP_TVAL_EL0, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !(EL2Enabled() & HCR_EL2.E2H, TGE) == '11') & CNTKCTL_EL1.EL0PTEN == '0' then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    EL2Enabled() & HCR_EL2.E2H == '0' & CNTHCTL_EL2.EL1PCEN == '0' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.E2H == '1' & CNTHCTL_EL2.EL1PTEN == '0' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    end if
  end if
  IsFeatureImplemented(FEAT_SEL2) then
    CNTHP_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
  else
    CNTP_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTPOFF_EL2;
  end if
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.E2H == '0' & CNTHCTL_EL2.EL1PCEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & HCR_EL2.E2H == '1' & CNTHCTL_EL2.EL1PTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  end if
  IsFeatureImplemented(FEAT_ECV) & EL2Enabled() & SCR_EL3.ECVEn == '1' & CNTHCTL_EL2.ECV == '1' & HCR_EL2.E2H == '1' then
    CNTP_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTPOFF_EL2;
else
  CNTP_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' & SCR_EL3.NS == '0' & IsFeatureImplemented(FEAT_SEL2) then
    CNTHPS_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
  else
    CNTP_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTPOFF_EL2;
  end if
elsif PSTATE.EL == EL3 then
  CNTP_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
else
  CNTP_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
D17.11.6 CNTHPS_CTL_EL2, Counter-timer Secure Physical Timer Control register (EL2)

The CNTHPS_CTL_EL2 characteristics are:

**Purpose**
Control register for the Secure EL2 physical timer.

**Configurations**
AArch64 System register CNTHPS_CTL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHPS_CTL[31:0].

This register is present only when FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHPS_CTL_EL2 are UNDEFINEd.

**Attributes**

CNTHPS_CTL_EL2 is a 64-bit register.

**Field descriptions**

![Register Diagram]

**Bits [63:3]**
Reserved, RES0.

**ISTATUS, bit [2]**
The status of the timer. This bit indicates whether the timer condition is met:
0b0 Timer condition is not met.
0b1 Timer condition is met.

When the value of the CNTHPS_CTL_EL2.ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the CNTHPS_CTL_EL2.ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

**IMASK, bit [1]**
Timer interrupt mask bit. Permitted values are:
0b0 Timer interrupt is not masked by the IMASK bit.
0b1 Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
**ENABLE, bit [0]**

Enables the timer. Permitted values are:

- **0b0**: Timer disabled.
- **0b1**: Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from `CNTHPS_TVVAL_EL2` continues to count down.

--- **Note** ---

Disabling the output signal might be a power-saving option.

---

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Accessing CNTHPS_CTL_EL2**

Accesses to this register use the following encodings in the System register encoding space:

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b100 & 0b1110 & 0b0101 & 0b001 \\
\end{array}
\]

```c
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  else
    X[t, 64] = CNTHPS_CTL_EL2;
elsif PSTATE.EL == EL3 then
  if SCR_EL3.EEL2 == '0' then
    UNDEFINED;
  else
    X[t, 64] = CNTHPS_CTL_EL2;
else
  UNDEFINED;
\]

\[
\begin{array}{cccccc}
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
0b11 & 0b100 & 0b1110 & 0b0101 & 0b001 \\
\end{array}
\]

```
else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        UNDEFINED;
    else
        CNTHPS_CTL_EL2 = X[t, 64];
    elsif PSTATE.EL == EL3 then
        if SCR_EL3.EEL2 == '0' then
            UNDEFINED;
        else
            CNTHPS_CTL_EL2 = X[t, 64];
        end if
    end if
end if

MRS <Xt>, CNTP_CTL_EL0

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if !(EL2Enabled() & HCR_EL2.<E2H,TGE> == '11') & CNTHCTL_EL1.EL0PTEN == '0' then
        if EL2Enabled() & HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        end if
    elsif EL2Enabled() & HCR_EL2.E2H == '0' & CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '10' & CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & CNTHCTL_EL2.EL0PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & SCR_EL3.NS == '0' & IsFeatureImplemented(FEAT_SEL2) then
        X[t, 64] = CNTHPS_CTL_EL2;
    elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & SCR_EL3.NS == '1' then
        X[t, 64] = CNTHP_CTL_EL2;
    else
        X[t, 64] = CNTP_CTL_EL0;
    end if
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & HCR_EL2.E2H == '0' & CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.E2H == '1' & CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x180];
    else
        X[t, 64] = CNTP_CTL_EL0;
    end if
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' & SCR_EL3.NS == '0' & IsFeatureImplemented(FEAT_SEL2) then
        X[t, 64] = CNTHPS_CTL_EL2;
    elsif HCR_EL2.E2H == '1' & SCR_EL3.NS == '1' then
        X[t, 64] = CNTHP_CTL_EL2;
    else
        X[t, 64] = CNTP_CTL_EL0;
    end if
elsif PSTATE.EL == EL3 then
    X[t, 64] = CNTP_CTL_EL0;
**MSR CNTP_CTL_EL0, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        end elsif
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
        IsFeatureImplemented( FEAT_SEL2) then
        CNTHPS_CTL_EL2 = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CTL_EL2 = X[t, 64];
    else
        CNTP_CTL_EL0 = X[t, 64];
    end elsif
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL2.EL1PTEN == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        end elsif
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
        IsFeatureImplemented( FEAT_SEL2) then
        CNTHPS_CTL_EL2 = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CTL_EL2 = X[t, 64];
    else
        CNTP_CTL_EL0 = X[t, 64];
    end elsif
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<E2H,TGE> == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented( FEAT_SEL2) then
        CNTHPS_CTL_EL2 = X[t, 64];
    elsif HCR_EL2.<E2H,TGE> == '1' && SCR_EL3.NS == '1' then
        CNTHP_CTL_EL2 = X[t, 64];
    else
        CNTP_CTL_EL0 = X[t, 64];
    end elsif
elsif PSTATE.EL == EL3 then
    CNTP_CTL_EL0 = X[t, 64];
D17.11.7  CNTHPS_CVAL_EL2, Counter-timer Secure Physical Timer CompareValue register (EL2)

The CNTHPS_CVAL_EL2 characteristics are:

**Purpose**

Holds the compare value for the Secure EL2 physical timer.

**Configurations**

AArch64 System register CNTHPS_CVAL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHPS_CVAL[31:0].

This register is present only when EL2 is implemented and FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHPS_CVAL_EL2 are UNDEFINED.

**Attributes**

CNTHPS_CVAL_EL2 is a 64-bit register.

**Field descriptions**

\[
\begin{array}{c}
\text{CompareValue} \\
63 & 32 \\
31 & 0 \\
\end{array}
\]

**CompareValue, bits [63:0]**

Holds the EL2 physical timer CompareValue.

When CNTHPS_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHPS_CTL_EL2.ISTATUS is set to 1.
- If CNTHPS_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHPS_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTHPS_CVAL_EL2**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, CNTHPS_CVAL_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0101</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  

ARM DDI 0487I.a  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. ID081822 Non-Confidential
if !IsCurrentSecurityState(SS_Secure) then
  UNDEFINED;
elsif EL2Enabled() && HCR_EL2.NV == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
      UNDEFINED;
    else
      X[t, 64] = CNTHPS_CVAL_EL2;
    end
  else
    X[t, 64] = CNTHPS_CVAL_EL2;
end

**MSR CNTHPS_CVAL_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0101</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
  elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
      UNDEFINED;
    else
      CNTHPS_CVAL_EL2 = X[t, 64];
    end
  elseif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
      UNDEFINED;
    else
      CNTHPS_CVAL_EL2 = X[t, 64];
    end
  end

**MRS <Xt>, CNTP_CVAL_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTHCTL_EL1.EL0PTEN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    end
  elsif EL2Enabled() && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  end

AArch64.SystemAccessTrap(EL2, 0x18);
elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FEAT_SEL2) then
X[t, 64] = CNTHPS_CVAL_EL2;
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
X[t, 64] = CNTHP_CVAL_EL2;
else
X[t, 64] = CNTP_CVAL_EL0;
elsif PSTATE_EL == EL1 then
if EL2Enabled() && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
X[t, 64] = NVMem[0x178];
else
X[t, 64] = CNTP_CVAL_EL0;
elsif PSTATE_EL == EL2 then
if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
X[t, 64] = CNTHPS_CVAL_EL2;
elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
X[t, 64] = CNTHP_CVAL_EL2;
else
X[t, 64] = CNTP_CVAL_EL0;
elsif PSTATE_EL == EL3 then
X[t, 64] = CNTP_CVAL_EL0;

MSR CNTP_CVAL_EL0, <t>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b110</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTHCTL_EL1.EL0PTEN == '0' then
if EL2Enabled() && HCR_EL2.TGE == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL0PTEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FEAT_SEL2) then
CNTHPS_CVAL_EL2 = X[t, 64];
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
CNTHP_CVAL_EL2 = X[t, 64];
else
CNTP_CVAL_EL0 = X[t, 64];
elsif PSTATE_EL == EL1 then
if EL2Enabled() && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
NVMem[0x178] = X[t, 64];
else
CNTP_CVAL_EL0 = X[t, 64];
elsif PSTATE_EL == EL2 then
if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
CNTHPS_CVAL_EL2 = X[t, 64];
elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
    CNTHP_CVAL_EL2 = X[t, 64];
else
    CNT_P_CVAL_EL0 = X[t, 64];
elsif PSTATE_EL == EL3 then
    CNT_P_CVAL_EL0 = X[t, 64];
D17.11.8 CNTHPS_TVAL_EL2, Counter-timer Secure Physical Timer TimerValue register (EL2)

The CNTHPS_TVAL_EL2 characteristics are:

**Purpose**

Holds the timer value for the Secure EL2 physical timer.

**Configurations**

AArch64 System register CNTHPS_TVAL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHPS_TVAL[31:0].

This register is present only when EL2 is implemented and FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHPS_TVAL_EL2 are UNDEFINED.

**Attributes**

CNTHPS_TVAL_EL2 is a 64-bit register.

**Field descriptions**

[Diagram of field descriptions]

**Bits [63:32]**

Reserved, RES0.

**TimerValue, bits [31:0]**

The TimerValue view of the EL2 physical timer.

On a read of this register:

- If CNTHPS_CTL_EL2.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHPS_CTL_EL2.ENABLE is 1, the value returned is (CNTHPS_CVAL_EL2 - CNTPCT_EL0).

On a write of this register, CNTHPS_CVAL_EL2 is set to (CNTPCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHPS_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CNTHPS_CVAL_EL2) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHPS_CTL_EL2.ISTATUS is set to 1.
- If CNTHPS_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHPS_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTHPS_TVAL_EL2**

Accesses to this register use the following encodings in the System register encoding space:


\textbf{MRS <Xt>, CNTHPS_TVAL_EL2}

\begin{center}
\begin{tabular}{cccccc}
\hline
op0 & op1 & CRn & CRm & op2 \\
\hline
0b11 & 0b100 & 0b1110 & 0b0101 & 0b000 \\
\hline
\end{tabular}
\end{center}

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  else
    if CNTHPS_CTL_EL2.ENABLE == '0' then
      X[t, 64] = bits(64) UNKNOWN;
    else
      X[t, 64] = CNTHPS_CVAL_EL2 - PhysicalCountInt();
  endif
elsif PSTATE.EL == EL3 then
  if SCR_EL3.EEL2 == '0' then
    UNDEFINED;
  else
    if CNTHPS_CTL_EL2.ENABLE == '0' then
      X[t, 64] = bits(64) UNKNOWN;
    else
      X[t, 64] = CNTHPS_CVAL_EL2 - PhysicalCountInt();
  endif
endif

\textbf{MSR CNTHPS_TVAL_EL2, <Xt>}

\begin{center}
\begin{tabular}{cccccc}
\hline
op0 & op1 & CRn & CRm & op2 \\
\hline
0b11 & 0b100 & 0b1110 & 0b0101 & 0b000 \\
\hline
\end{tabular}
\end{center}

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  else
    CNTHPS_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
  endif
elsif PSTATE.EL == EL3 then
  if SCR_EL3.EEL2 == '0' then
    UNDEFINED;
  else
    CNTHPS_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
  endif
MRS \langle X_t \rangle, \text{CNTP\_TV\_VAL\_EL0} \n
\begin{tabular}{cccccc}
\hline
op0 & op1 & CRn & CRm & op2 \\
\hline
0b11 & 0b011 & 0b1110 & 0b0010 & 0b0000 \\
\end{tabular}

if PSTATE.EL == EL0 then
if (EL2Enabled() && HCR.EL2.<E2H,TGE> == '11') && CNTKCTL.EL1.EL0PTEN == '0' then
if EL2Enabled() && HCR.EL2.TGE == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && HCR.EL2.<E2H,TGE> == '10' && CNTKCTL.EL2.EL0PTEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR.EL2.<E2H,TGE> == '11' && CNTKCTL.EL2.EL0PTEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR.EL2.<E2H,TGE> == '11' && SCR.EL3.NS == '0' &&
IsFeatureImplemented(FEAT_SEL2) then
if CNTHPS_CTL.EL2.ENABLE == '0' then
X[t, 64] = bits(64) UNKNOWN;
else
X[t, 64] = CNTHPS.CVAL.EL2 - PhysicalCountInt();
elsif EL2Enabled() && HCR.EL2.<E2H,TGE> == '11' && SCR.EL3.NS == '1' then
if CNTHP_CTL.EL2.ENABLE == '0' then
X[t, 64] = bits(64) UNKNOWN;
else
X[t, 64] = CNTHP.CVAL.EL2 - PhysicalCountInt();
elsif IsFeatureImplemented(FEAT_ECV) & EL2Enabled() && SCR.EL3.ECVEn == '1' & CNTHCTL.EL2.ECV == '1' & HCR.EL2.<E2H,TGE> != '11' then
if CNTP_CTL.EL1.ENABLE == '0' then
X[t, 64] = bits(64) UNKNOWN;
else
X[t, 64] = CNTP.CVAL.EL1 - (PhysicalCountInt() - CNTPOFF.EL2);
elsif PSTATE.EL == EL1 then
if EL2Enabled() && HCR.EL2.E2H == '0' & CNTHCTL.EL2.E2PCEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR.EL2.E2H == '1' & CNTHCTL.EL2.EL1PTEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif IsFeatureImplemented(FEAT_ECV) & EL2Enabled() && SCR.EL3.ECVEn == '1' & CNTHCTL.EL2.ECV == '1' & SCR.EL3.NS == '0' then
if CNTP_CTL.EL2.ENABLE == '0' then
X[t, 64] = bits(64) UNKNOWN;
else
X[t, 64] = CNTP.CVAL.EL2 - PhysicalCountInt();
elsif PSTATE.EL == EL2 then
if HCR.EL2.E2H == '1' & SCR.EL3.NS == '0' & IsFeatureImplemented(FEAT_SEL2) then
if CNTHPS_CTL.EL2.ENABLE == '0' then
X[t, 64] = bits(64) UNKNOWN;
else
X[t, 64] = CNTHPS.CVAL.EL2 - PhysicalCountInt();
elsif HCR.EL2.E2H == '1' & SCR.EL3.NS == '1' then
if CNTP_CTL.EL2.ENABLE == '0' then
X[t, 64] = bits(64) UNKNOWN;
else
    X[t, 64] = CNTHP_CVAL_EL2 - PhysicalCountInt();
else
    if CNTP_CTL_EL0.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = CNTP_CVAL_EL0 - PhysicalCountInt();
elsif PSTATE.EL == EL3 then
    if CNTP_CTL_EL0.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = CNTP_CVAL_EL0 - PhysicalCountInt();

MSR CNTP_TVAL_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if (!EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTHCTL_EL1.EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL0PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && SCR_EL3.ECVEn == '1' && CNTHCTL_EL2.ECV == '1' && HCR_EL2.<E2H,TGE> == '11' then
        CNTP_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTPOFF_EL2;
    else
        CNTP_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.E2H == '0' && CNTHCTL_EL2.E1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.E2H == '1' && CNTHCTL_EL2.E1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && SCR_EL3.ECVEn == '1' && CNTHCTL_EL2.ECV == '1' then
        CNTP_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTPOFF_EL2;
    else
        CNTP_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHP_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    else
        CNTP_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
elsif PSTATE.EL == EL3 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' then
        CNTHPS_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHP_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    else
        CNTP_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
D17.11.9 CNTHV_CTL_EL2, Counter-timer Virtual Timer Control register (EL2)

The CNTHV_CTL_EL2 characteristics are:

Purpose
Control register for the EL2 virtual timer.

Configurations
AArch64 System register CNTHV_CTL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHV_CTL[31:0].

This register is present only when FEAT_VHE is implemented and (EL3 is implemented or (EL3 is not implemented and FEAT_SEL2 is not implemented)). Otherwise, direct accesses to CNTHV_CTL_EL2 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes
CNTHV_CTL_EL2 is a 64-bit register.

Field descriptions

Bits [63:3]
Reserved, RES0.

ISTATUS, bit [2]
The status of the timer. This bit indicates whether the timer condition is met:

0b0 Timer condition is not met.
0b1 Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Access to this field is RO.

IMASK, bit [1]
Timer interrupt mask bit. Permitted values are:

0b0 Timer interrupt is not masked by the IMASK bit.
0b1 Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
ENABLE, bit [0]

Enables the timer. Permitted values are:

- **0b0**  Timer disabled.
- **0b1**  Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHV_TVAL_EL2 continues to count down.

**Note**

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHV_CTL_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHV_CTL_EL2 or CNTV_CTL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, CNTHV_CTL_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = CNTHV_CTL_EL2;
elsif PSTATE.EL == EL3 then
  X[t, 64] = CNTHV_CTL_EL2;

**MSR CNTHV_CTL_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  CNTHV_CTL_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    CNTHV_CTL_EL2 = X[t, 64];

MRS <Xt>, CNTV_CTL_EL0

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL2.EL0VTEN == '0' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTKCTL_EL2.EL1VTEN == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
                IsFeatureImplemented(FEAT_SEL2) then
                X[t, 64] = CNTHV_CTL_EL2;
            elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
                X[t, 64] = CNTHV_CTL_EL2;
            else
                X[t, 64] = CNTV_CTL_EL0;
            end

else
    if PSTATE.EL == EL1 then
        if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL2.EL1VTEN == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL2.EL1VTEN == '1' then
            X[t, 64] = CNTHV_CTL_EL2;
        else
            X[t, 64] = CNTV_CTL_EL0;
        end
    elseif PSTATE.EL == EL2 then
        if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
            X[t, 64] = CNTHV_CTL_EL2;
        elseif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
            X[t, 64] = CNTHV_CTL_EL2;
        else
            if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
                IsFeatureImplemented(FEAT_SEL2) then
                X[t, 64] = CNTHV_CTL_EL2;
            elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
                X[t, 64] = CNTHV_CTL_EL2;
            else
                X[t, 64] = CNTHV_CTL_EL2;
        end
    elseif PSTATE.EL == EL3 then
        X[t, 64] = CNTV_CTL_EL0;
    end

MSR CNTV_CTL_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
        elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL2.EL0VTEN == '0' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTKCTL_EL2.EL1VTEN == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
                IsFeatureImplemented(FEAT_SEL2) then
                X[t, 64] = CNTHV_CTL_EL2;
            elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
                X[t, 64] = CNTHV_CTL_EL2;
            else
                X[t, 64] = CNTV_CTL_EL0;
            end

else
    if PSTATE.EL == EL1 then
        if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL2.EL1VTEN == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL2.EL1VTEN == '1' then
            X[t, 64] = CNTHV_CTL_EL2;
        else
            X[t, 64] = CNTV_CTL_EL0;
        end
    elseif PSTATE.EL == EL2 then
        if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
            X[t, 64] = CNTHV_CTL_EL2;
        elseif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
            X[t, 64] = CNTHV_CTL_EL2;
        else
            if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
                IsFeatureImplemented(FEAT_SEL2) then
                X[t, 64] = CNTHV_CTL_EL2;
            elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
                X[t, 64] = CNTHV_CTL_EL2;
            else
                X[t, 64] = CNTV_CTL_EL0;
            end
    elseif PSTATE.EL == EL3 then
        X[t, 64] = CNTV_CTL_EL0;
    end

MSR CNTV_CTL_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>
IsFeatureImplemented(FEAT_SEL2) then
    CNTHVS_CTL_EL2 = X[t, 64];
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
    CNTHV_CTL_EL2 = X[t, 64];
else
    CNTV_CTL_EL0 = X[t, 64];
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && CNTHCTL_EL2.E1TVT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x170] = X[t, 64];
    else
        CNTV_CTL_EL0 = X[t, 64];
elsif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CTL_EL2 = X[t, 64];
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        CNTHV_CTL_EL2 = X[t, 64];
    else
        CNTV_CTL_EL0 = X[t, 64];
elsif PSTATE_EL == EL3 then
    CNTV_CTL_EL0 = X[t, 64];
D17.11.10  CNTHV_CVAL_EL2, Counter-timer Virtual Timer CompareValue register (EL2)

The CNTHV_CVAL_EL2 characteristics are:

**Purpose**

Holds the compare value for the EL2 virtual timer.

**Configurations**

AArch64 System register CNTHV_CVAL_EL2 bits [63:0] are architecturally mapped to AArch32 System register CNTHV_CVAL[63:0].

This register is present only when FEAT_VHE is implemented and (EL3 is implemented or (EL3 is not implemented and FEAT_SEL2 is not implemented)). Otherwise, direct accesses to CNTHV_CVAL_EL2 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

CNTHV_CVAL_EL2 is a 64-bit register.

**Field descriptions**

CompareValue, bits [63:0]

Holds the EL2 virtual timer CompareValue.

When CNTHV_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTVCT_EL0 - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHV_CTL_EL2.ISTATUS is set to 1.
- If CNTHV_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHV_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTVCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTHV_CVAL_EL2**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHV_CVAL_EL2 or CNTV_CVAL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:
### MRS <Xt>, CNTHV_CVAL_EL2

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = CNTHV_CVAL_EL2;
elsif PSTATE.EL == EL3 then
  X[t, 64] = CNTHV_CVAL_EL2;

### MSR CNTHV_CVAL_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  CNTHV_CVAL_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
  CNTHV_CVAL_EL2 = X[t, 64];

### MRS <Xt>, CNTV_CVAL_EL0

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && SCR_EL3.NS == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && IsFeatureImplemented(FEAT_SEL2) then
      X[t, 64] = CNTHVS_CVAL_EL2;
  elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
    X[t, 64] = CNTV_CVAL_EL2;


else
  X[t, 64] = CNTV_CVAL_EL0;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && CNTHCTL_EL2.EL1VT == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    X[t, 64] = NVMem[0x168];
  else
    X[t, 64] = CNTV_CVAL_EL0;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
    X[t, 64] = CNTHVS_CVAL_EL2;
  elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
    X[t, 64] = CNTHV_CVAL_EL2;
  else
    X[t, 64] = CNTV_CVAL_EL0;
elsif PSTATE.EL == EL3 then
  X[t, 64] = CNTV_CVAL_EL0;

MSR CNTV_CVAL_EL0, <X>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTHKCTL_EL1.EL0VTEN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHKCTL_EL2.EL0VTEN == '0' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHKCTL_EL2.EL1VTEN == '0' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
      IsFeatureImplemented(FEAT_SEL2) then
      CNTHVS_CVAL_EL2 = X[t, 64];
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
      CNTHV_CVAL_EL2 = X[t, 64];
    else
      CNTV_CVAL_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && CNTHKCTL_EL2.EL1VTEN == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    NVMem[0x168] = X[t, 64];
  else
    CNTV_CVAL_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
    CNTHVS_CVAL_EL2 = X[t, 64];
  elseif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
    CNTHV_CVAL_EL2 = X[t, 64];
  else
    CNTV_CVAL_EL0 = X[t, 64];
elsif PSTATE.EL == EL3 then
  CNTV_CVAL_EL0 = X[t, 64];
### D17.11.1 CNTHV_TVAL_EL2, Counter-timer Virtual Timer TimerValue Register (EL2)

The CNTHV_TVAL_EL2 characteristics are:

**Purpose**

Holds the timer value for the EL2 virtual timer.

**Configurations**

AArch64 System register CNTHV_TVAL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHV_TV[31:0].

This register is present only when FEAT_VHE is implemented and (EL3 is implemented or (EL3 is not implemented and FEAT_SEL2 is not implemented)). Otherwise, direct accesses to CNTHV_TVAL_EL2 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

CNTHV_TVAL_EL2 is a 64-bit register.

**Field descriptions**

![Field Descriptions Diagram]

**Bits [63:32]**

Reserved, RES0.

**TimerValue, bits [31:0]**

The TimerValue view of the EL2 virtual timer.

On a read of this register:

- If CNTHV_CTL_EL2.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHV_CTL_EL2.ENABLE is 1, the value returned is (CNTHV_CVAL_EL2 - CNTVCT_EL0).

On a write of this register, CNTHV_CVAL_EL2 is set to (CNTVCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHV_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTVCT_EL0 - CNTHV_CVAL_EL2) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHV_CTL_EL2.ISTATUS is set to 1.
- If CNTHV_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHV_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTVCT_EL0 continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTHV_TVAL_EL2**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CNTHV_TVAL_EL2 or CNTV_TVAL_EL0 are not guaranteed to be ordered with respect to accesses using the other mnemonic.
Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, CNTHV_TVAL_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE_EL == EL2 then
    if CNTHV_CTL_EL2.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = CNTHV_CVAL_EL2 - PhysicalCountInt();
elsif PSTATE_EL == EL3 then
    if CNTHV_CTL_EL2.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = CNTHV_CVAL_EL2 - PhysicalCountInt();

**MSR CNTHV_TVAL_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE_EL == EL2 then
    CNTHV_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
elsif PSTATE_EL == EL3 then
    CNTHV_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();

**MRS <Xt>, CNTV_TVAL_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
    if CNTHVS_CTL_EL2.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = CNTHVS_CVAL_EL2 - PhysicalCountInt();
    endif
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
    if CNTH_CTL_EL2.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = CNTHV_CVAL_EL2 - PhysicalCountInt();
    endif
    elsif HaveEL(EL2) && (!EL2Enabled() || HCR_EL2.<E2H,TGE> != '11') then
    if CNTV_CTL_EL0.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = CNTV_CVAL_EL0 - (PhysicalCountInt() - CNTVOFF_EL2);
    endif
    elsif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL_EL2.EL1TVT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
elif HaveEL(EL2) then
    if CNTV_CTL_EL0.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = CNTV_CVAL_EL0 - (PhysicalCountInt() - CNTVOFF_EL2);
    endif
    elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
    if CNTHVS_CTL_EL2.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = CNTHVS_CVAL_EL2 - PhysicalCountInt();
    endif
    elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
    if CNTHV_CTL_EL2.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = CNTHV_CVAL_EL2 - PhysicalCountInt();
    endif
    elsif HCR_EL2.E2H == '0' then
    if CNTV_CTL_EL0.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = CNTV_CVAL_EL0 - (PhysicalCountInt() - CNTVOFF_EL2);
    endif
    elsif PSTATE.EL == EL3 then
    if CNTV_CTL_EL0.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    elsif HaveEL(EL2) && !ELUsingAArch32(EL2) then
        X[t, 64] = CNTV_CVAL_EL0 - (PhysicalCountInt() - CNTVOFF_EL2);
elif HaveEL(EL2) && ELUsingAArch32(EL2) then
        X[t, 64] = CNTV_CVAL_EL0 - (PhysicalCountInt() - CNTVOFF);
else
    X[t, 64] = CNTV_CVAL_EL0 - PhysicalCountInt();

**MSR CNTV_TVAL_EL0, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if (EL2Enabled() & HCR_EL2.<E2H,TGE> == '11') & CNTKCTL_EL1.EL0VTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & CNTKCTL_EL2.EL0VTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.<E2H,TGE> != '11' & CNTHCTL_EL2.EL1TVT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & SCR_EL3.NS == '0' & IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & SCR_EL3.NS == '1' then
        CNTHV_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    elsif HaveEL(EL2) & (!EL2Enabled() || HCR_EL2.<E2H,TGE> != '11') then
        CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF_EL2;
    else
        CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & CNTHCTL_EL2.EL1TVT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL2) then
        CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF_EL2;
    else
        CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' & SCR_EL3.NS == '0' & IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    elsif HCR_EL2.E2H == '1' & SCR_EL3.NS == '1' then
        CNTHV_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    elsif HCR_EL2.E2H == '0' then
        CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF_EL2;
    else
        CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
elsif PSTATE.EL == EL3 then
    if HaveEL(EL2) & !ELUsingAArch32(EL2) then
        CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF_EL2;
    elsif HaveEL(EL2) & ELUsingAArch32(EL2) then
        CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF;
    else
        CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt());
D17.11.12 CNTHVS_CTL_EL2, Counter-timer Secure Virtual Timer Control register (EL2)

The CNTHVS_CTL_EL2 characteristics are:

**Purpose**

Control register for the Secure EL2 virtual timer.

**Configurations**

AArch64 System register CNTHVS_CTL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHVS_CTL[31:0]. This register is present only when FEAT_SEL2 is implemented and FEAT_VHE is implemented. Otherwise, direct accesses to CNTHVS_CTL_EL2 are UNDEFINED.

**Attributes**

CNTHVS_CTL_EL2 is a 64-bit register.

**Field descriptions**

Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

- 0b0 Timer condition is not met.
- 0b1 Timer condition is met.

When the value of the CNTHVS_CTL_EL2.ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

- 0b0 Timer interrupt is not masked by the IMASK bit.
- 0b1 Timer interrupt is masked by the IMASK bit.

For more information, see the description of the CNTHVS_CTL_EL2.ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
ENABLE, bit [0]

Enables the timer. Permitted values are:

- 0b0  Timer disabled.
- 0b1  Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHVS_TVAL_EL2 continues to count down.

--- Note ---

Disabling the output signal might be a power-saving option.

---

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHVS_CTL_EL2

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, CNTHVS_CTL_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  else
    X[t, 64] = CNTHVS_CTL_EL2;
elsif PSTATE.EL == EL3 then
  if SCR_EL3.EEL2 == '0' then
    UNDEFINED;
  else
    X[t, 64] = CNTHVS_CTL_EL2;
else
  X[t, 64] = CNTHVS_CTL_EL2;
```

**MSR CNTHVS_CTL_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
  X[t, 64] = CNTHVS_CTL_EL2;
```
else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
      UNDEFINED;
    else
      CNTHVS_CTL_EL2 = X[t, 64];
    end if
elsif PSTATE_EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
      UNDEFINED;
    else
      CNTHVS_CTL_EL2 = X[t, 64];
    end if
MRS <Xt>, CNTV_CTL_EL0

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '1') && CNTKCTL_EL1.EL0VTEN == '0' then
      if EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      end if
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    end if
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '1' && CNTKCTL_EL2.EL0VTEN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    end if
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '1' && CNTHCTL_EL2.EL1TVT == '1' then
    if EL2Enabled() && HCR_EL2.<E2H,TGE> == '1' && SCR_EL3.NS == '0' &&
    IsFeatureImplemented(FEAT_SEL2) then
      X[t, 64] = CNTHVS_CTL_EL2;
    else
      X[t, 64] = CNTV_CTL_EL0;
    end if
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '1' && SCR_EL3.NS == '1' then
    X[t, 64] = CNTHV_CTL_EL2;
else
    X[t, 64] = CNTV_CTL_EL0;
else
    if PSTATE_EL == EL1 then
      if EL2Enabled() && CNTHKCTL_EL2.E1VTVT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
      else
        X[t, 64] = NVMem[0x170];
      end if
    else
      X[t, 64] = CNTV_CTL_EL0;
    end if
elsif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
      X[t, 64] = CNTHVS_CTL_EL2;
    else
      if HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        X[t, 64] = CNTHV_CTL_EL2;
      else
        X[t, 64] = CNTV_CTL_EL0;
      end if
    end if
elsif PSTATE_EL == EL3 then
    X[t, 64] = CNTV_CTL_EL0;
MSR CNTV_CTL_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !(EL2Enabled() & HCR_EL2.<E2H,TGE> == '11') & CNTKCTL_EL1.EL0VTEN == '0' then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & CNTKCTL_EL1.EL0VTEN == '0' then
  AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() & HCR_EL2.<E2H,TGE> != '11' & CNTHCTL_EL2.EL1TVT == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & SCR_EL3.NS == '0' &
  IsFeatureImplemented(FEAT_SEL2) then
  CNTHVS_CTL_EL2 = X[t, 64];
elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & SCR_EL3.NS == '1' then
  CNTHV_CTL_EL2 = X[t, 64];
else
  CNTV_CTL_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & CNTHCTL_EL2.EL1TVT == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '111' then
  NVMem[0x170] = X[t, 64];
else
  CNTV_CTL_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' & SCR_EL3.NS == '0' &
    IsFeatureImplemented(FEAT_SEL2) then
    CNTHVS_CTL_EL2 = X[t, 64];
elsif HCR_EL2.E2H == '1' & SCR_EL3.NS == '1' then
  CNTHV_CTL_EL2 = X[t, 64];
else
  CNTV_CTL_EL0 = X[t, 64];
elsif PSTATE.EL == EL3 then
  CNTV_CTL_EL0 = X[t, 64];
D17.11.13 CNTHVS_CVAL_EL2, Counter-timer Secure Virtual Timer CompareValue register (EL2)

The CNTHVS_CVAL_EL2 characteristics are:

**Purpose**

Holds the compare value for the Secure EL2 virtual timer.

**Configurations**

AArch64 System register CNTHVS_CVAL_EL2 bits [63:0] are architecturally mapped to AArch32 System register CNTHVS_CV[63:0].

This register is present only when FEAT_SEL2 is implemented and FEAT_VHE is implemented. Otherwise, direct accesses to CNTHVS_CVAL_EL2 are UNDEFINED.

**Attributes**

CNTHVS_CVAL_EL2 is a 64-bit register.

**Field descriptions**

CompareValue, bits [63:0]

Holds the Secure EL2 virtual timer CompareValue.

When CNTHVS_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTVCT_EL0 - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHVS_CTL_EL2.ISTATUS is set to 1.
- If CNTHVS_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHVS_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTVCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTHVS_CVAL_EL2**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS} <Xt>, \text{CNTHVS_CVAL_EL2}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0100</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if !IsCurrentSecurityState(SS_Secure) then
  UNDEFINED;
elsif EL2Enabled() & HCR_EL2.NV == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  UNDEFINED;
elsif PSTATE_EL == EL2 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  else
    X[t, 64] = CNTHVS_CVAL_EL2;
elsif PSTATE_EL == EL3 then
  if SCR_EL3.EEL2 == '0' then
    UNDEFINED;
  else
    X[t, 64] = CNTHVS_CVAL_EL2;

**MSR CNTHVS_CVAL_EL2, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0100</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  elsif EL2Enabled() & HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if !IsCurrentSecurityState(SS_Secure) then
    UNDEFINED;
  else
    CNTHVS_CVAL_EL2 = X[t, 64];
elsif PSTATE_EL == EL3 then
  if SCR_EL3.EEL2 == '0' then
    UNDEFINED;
  else
    CNTHVS_CVAL_EL2 = X[t, 64];

**MRS <Xt>, CNTV_CVAL_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b01</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if !(EL2Enabled() & HCR_EL2.<E2H,TGE> == '11') & CNTHCTL_EL1.EL0VTEN == '0' then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    elseif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & CNTHCTL_EL2.EL0VTEN == '0' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() & HCR_EL2.<E2H,TGE> != '11' & CNTHCTL_EL2.EL1VT == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & SCR_EL3.NS == '0' &

IsFeatureImplemented(FEAT_SEL2) then
X[t, 64] = CNTHVS_CVAL_EL2;
elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & SCR_EL3.NS == '1' then
X[t, 64] = CNTHV_CVAL_EL2;
else
X[t, 64] = CNTV_CVAL_EL0;
elsif PSTATE_EL == EL1 then
if EL2Enabled() & CNTHCTL_EL2.EL1TVT == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' then
X[t, 64] = CNTHVS_CVAL_EL2;
else
X[t, 64] = CNTV_CVAL_EL0;
elsif PSTATE_EL == EL2 then
if HCR_EL2.E2H == '1' & SCR_EL3.NS == '0' & IsFeatureImplemented(FEAT_SEL2) then
X[t, 64] = CNTHV_CVAL_EL2;
elsif HCR_EL2.E2H == '11' & SCR_EL3.NS == '1' then
X[t, 64] = CNTHV_CVAL_EL2;
else
X[t, 64] = CNTV_CVAL_EL0;
elsif PSTATE_EL == EL3 then
X[t, 64] = CNTV_CVAL_EL0;

MSR CNTV_CVAL_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
if !(EL2Enabled() & HCR_EL2.<E2H,TGE> == '11') & CNTHCTL_EL1.EL0VTEN == '0' then
if EL2Enabled() & HCR_EL2.TGE == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & CNTHCTL_EL2.EL0VTEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & CNTHCTL_EL2.EL1TVT == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & SCR_EL3.NS == '0' &
IsFeatureImplemented(FEAT_SEL2) then
CNTHVS_CVAL_EL2 = X[t, 64];
elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & SCR_EL3.NS == '1' then
CNTHV_CVAL_EL2 = X[t, 64];
else
CNTV_CVAL_EL0 = X[t, 64];
elsif PSTATE_EL == EL1 then
if EL2Enabled() & CNTHCTL_EL2.EL1TVT == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' then
NVMem[0x168] = X[t, 64];
else
CNTV_CVAL_EL0 = X[t, 64];
elsif PSTATE_EL == EL2 then
if HCR_EL2.E2H == '1' & SCR_EL3.NS == '0' & IsFeatureImplemented(FEAT_SEL2) then
CNTHVS_CVAL_EL2 = X[t, 64];
elsif HCR_EL2.E2H == '1' & SCR_EL3.NS == '1' then
CNTHV_CVAL_EL2 = X[t, 64];
else
CNTV_CVAL_EL0 = X[t, 64];
elsif PSTATE_EL == EL3 then
CNTV_CVAL_EL0 = X[t, 64];
D17.11.14   CNTHVS_TVAL_EL2, Counter-timer Secure Virtual Timer TimerValue register (EL2)

The CNTHVS_TVAL_EL2 characteristics are:

**Purpose**

Holds the timer value for the Secure EL2 virtual timer.

**Configurations**

AArch64 System register CNTHVS_TVAL_EL2 bits [31:0] are architecturally mapped to AArch32 System register CNTHVS_TVAL[31:0].

This register is present only when FEAT_SEL2 is implemented and FEAT_VHE is implemented. Otherwise, direct accesses to CNTHVS_TVAL_EL2 are UNDEFINED.

**Attributes**

CNTHVS_TVAL_EL2 is a 64-bit register.

**Field descriptions**

![Diagram of register fields]

**Bits [63:32]**

Reserved, RES0.

**TimerValue, bits [31:0]**

The TimerValue view of the EL2 virtual timer.

On a read of this register:

- If CNTHVS_CTL_EL2.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHVS_CTL_EL2.ENABLE is 1, the value returned is (CNTHVS_CVAL_EL2 - CNTVCT_EL0).

On a write of this register, CNTHVS_CVAL_EL2 is set to (CNTVCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHVS_CTL_EL2.ENABLE is 1, the timer condition is met when ((CNTVCT_EL0 - CNTHVS_CVAL_EL2) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHVS_CTL_EL2.ISTATUS is set to 1.
- If CNTHVS_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHVS_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTVCT_EL0 continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTHVS_TVAL_EL2**

Accesses to this register use the following encodings in the System register encoding space:
**MRS <Xt>, CNTHVS_TVAL_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        UNDEFINED;
    else
        if CNTHVS_CTL_EL2.ENABLE == '0' then
            X[t, 64] = bits(64) UNKNOWN;
        else
            X[t, 64] = CNTHVS_CVAL_EL2 - PhysicalCountInt();
        end if
    endif
    elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        if CNTHVS_CTL_EL2.ENABLE == '0' then
            X[t, 64] = bits(64) UNKNOWN;
        else
            X[t, 64] = CNTHVS_CVAL_EL2 - PhysicalCountInt();
        end if
    endif
else
    MSR CNTHVS_TVAL_EL2, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if !IsCurrentSecurityState(SS_Secure) then
        UNDEFINED;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if !IsCurrentSecurityState(SS_Secure) then
        UNDEFINED;
    else
        CNTHVS_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    endif
elsif PSTATE.EL == EL3 then
    if SCR_EL3.EEL2 == '0' then
        UNDEFINED;
    else
        CNTHVS_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    endif

MRS <Xt>, CNTV_TVAL_EL0

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>b1110</td>
<td>b0011</td>
<td>b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !(EL2Enabled() & HCR_EL2.<E2H,TGE> == '11') & CNTKCTL_EL1.EL0VTEN == '0' then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & CNTKCTL_EL1.EL0VTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & HCR_EL2.<E2H,TGE> != '11' & CNTKCTL_EL1.EL1VTEN == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & SCR_EL3.NS == '0' &
    IsFeatureImplemented(FEAT_SEL2) then
    if CNTKVS_CTL_EL2.ENABLE == '0' then
      X[t, 64] = bits(64) UNKNOWN;
    else
      X[t, 64] = CNTKVS_CVAL_EL2 - PhysicalCountInt();
    end
  elseif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & SCR_EL3.NS == '1' then
    if CNTKVL_CVAL_EL2.ENABLE == '0' then
      X[t, 64] = bits(64) UNKNOWN;
    else
      X[t, 64] = CNTKVL_CVAL_EL2 - PhysicalCountInt();
    end
  elsif HaveEL(EL2) & (!EL2Enabled() || HCR_EL2.<E2H,TGE> != '11') then
    if CNTKVT_CTL_EL0.ENABLE == '0' then
      X[t, 64] = bits(64) UNKNOWN;
    else
      X[t, 64] = CNTKVT_CVAL_EL0 - (PhysicalCountInt() - CNTKVOFF_EL2);
    end
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.<E2H,TGE> == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif HaveEL(EL2) & (!EL2Enabled() || HCR_EL2.<E2H,TGE> != '11') then
    if CNTKVT_CTL_EL0.ENABLE == '0' then
      X[t, 64] = bits(64) UNKNOWN;
    else
      X[t, 64] = CNTKVT_CVAL_EL0 - (PhysicalCountInt() - CNTKVOFF_EL2);
    end
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' & SCR_EL3.NS == '0' &
    IsFeatureImplemented(FEAT_SEL2) then
    if CNTKVS_CTL_EL2.ENABLE == '0' then
      X[t, 64] = bits(64) UNKNOWN;
    else
      X[t, 64] = CNTKVS_CVAL_EL2 - PhysicalCountInt();
  elsif HCR_EL2.E2H == '1' & SCR_EL3.NS == '1' then
    if CNTKVL_CVAL_EL2.ENABLE == '0' then
      X[t, 64] = bits(64) UNKNOWN;
    else
      X[t, 64] = CNTKVL_CVAL_EL2 - PhysicalCountInt();
  elsif HCR_EL2.E2H == '0' then
    if CNTKVT_CTL_EL0.ENABLE == '0' then
      X[t, 64] = bits(64) UNKNOWN;
    else
      X[t, 64] = CNTKVT_CVAL_EL0 - PhysicalCountInt();
  end
X[t, 64] = CNTV_CVAL_EL0 - (PhysicalCountInt() - CNTVOFF_EL2);
else
  if CNTV_CTL_EL0.ENABLE == '0' then
    X[t, 64] = bits(64) UNKNOWN;
  else
    X[t, 64] = CNTV_CVAL_EL0 - PhysicalCountInt();
  end if
elsif PSTATE_EL == EL3 then
  if CNTV_CTL_EL0.ENABLE == '0' then
    X[t, 64] = bits(64) UNKNOWN;
  elsif HaveEL(EL2) && !ELUsingAArch32(EL2) then
    X[t, 64] = CNTV_CVAL_EL0 - (PhysicalCountInt() - CNTVOFF_EL2);
  elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
    X[t, 64] = CNTV_CVAL_EL0 - (PhysicalCountInt() - CNTVOFF);
  else
    X[t, 64] = CNTV_CVAL_EL0 - PhysicalCountInt();
  end if
endif;

MSR CNTV_TVAL_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if (EL2Enabled() & HCR_EL2.E2H,TGE) == '11' && CNTKCTL_EL1.EL0VTEN == '0' then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    endif
  elsif EL2Enabled() & HCR_EL2.E2H,TGE) == '1' && CNTKCTL_EL1.EL0VTEN == '0' then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    endif
  endif
else
  if HaveEL(EL2) && (!EL2Enabled() || HCR_EL2.E2H,TGE) == '11' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elseif HaveEL(EL2) then
    CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF_EL2;
  else
    CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
  endif
endif;
else
  if PSTATE_EL == EL1 then
    if EL2Enabled() & CNTKCTL_EL1.EL1VTEN == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    endif
  elseif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
      CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    else
      CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    endif
  elseif PSTATE_EL == EL3 then
    if HaveEL(EL2) && !ELUsingAArch32(EL2) then
      CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF_EL2;
    elseif HaveEL(EL2) && ELUsingAArch32(EL2) then
      CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF;
    else
      CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    endif
  endif
endif;
else
    CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
D17.11.15 CNTKCTL_EL1, Counter-timer Kernel Control register

The CNTKCTL_EL1 characteristics are:

Purpose

When FEAT_VHE is implemented and HCR_EL2.{E2H, TGE} is \{1, 1\}, this register does not cause any event stream from the virtual counter to be generated, and does not control access to the counters and timers. The access to counters and timers at EL0 is controlled by CNTKCTL_EL2.

When FEAT_VHE is not implemented, or when HCR_EL2.{E2H, TGE} is not \{1, 1\}, this register controls the generation of an event stream from the virtual counter, and access from EL0 to the physical counter, virtual counter, EL1 physical timers, and the virtual timer.

Configurations

AArch64 System register CNTKCTL_EL1 bits [31:0] are architecturally mapped to AArch32 System register CNTKCTL[31:0].

Attributes

CNTKCTL_EL1 is a 64-bit register.

Field descriptions

Bits [63:18]

Reserved, RES0.

EVNTIS, bit [17]

When FEAT_ECV is implemented:

Controls the scale of the generation of the event stream.

0b0  The CNTKCTL_EL1.EVNTI field applies to CNTVCT_EL0[15:0].
0b1  The CNTKCTL_EL1.EVNTI field applies to CNTVCT_EL0[23:8].

This control applies regardless of the value of the CNTKCTL_EL2.ECV bit.

The reset behavior of this field is:

•  On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [16:10]

Reserved, RES0.

EL0PTEN, bit [9]

Traps EL0 accesses to the physical timer registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

•  In AArch64 state, the following registers are trapped, reported using EC syndrome value 0x18:

  —  CNTP_CTL_EL0, CNTP_CVAL_EL0, and CNTP_TVAL_EL0.

RES0

RES0

RES0

RES0

EVNTIS

EL0PTEN

EVNTI

EL0VCTEN

EVNDIR

EVNTEN
In AArch32 state, MRC and MCR accesses to the following registers are trapped, reported using EC syndrome value 0x03, MRRC and MCRR accesses are trapped, reported using EC syndrome value 0x04:

- CNTP_CTL, CNTP_CVAL, CNTP_TV AL.

0b0: EL0 accesses to the physical timer registers are trapped to EL1.

0b1: This control does not cause any instructions to be trapped.

When FEAT_VHE is implemented and HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**EL0VTEN, bit [8]**

Traps EL0 accesses to the virtual timer registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to the following registers are trapped, reported using EC syndrome value 0x18:
  - CNTV_CTL_EL0, CNTV_CVAL_EL0, and CNTV_TV AL_EL0.

- In AArch32 state, MRC and MCR accesses to the following registers are trapped and reported using EC syndrome value 0x03, MRRC and MCRR accesses are trapped using EC syndrome value 0x04:
  - CNTV_CTL, CNTV_CVAL, and CNTV_TV AL.

0b0: EL0 accesses to the virtual timer registers are trapped.

0b1: This control does not cause any instructions to be trapped.

When FEAT_VHE is implemented and HCR_EL2.{E2H, TGE} is {1, 1}, this control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**EVNTI, bits [7:4]**

Selects which bit of CNTVCT_EL0, as seen from EL1, is the trigger for the event stream generated from that counter when that stream is enabled.

If FEAT_ECV is implemented, and CNTKCTL_EL1.EVNTIS is 1, this field selects a trigger bit in the range 8 to 23 of CNTVCT_EL0.

Otherwise, this field selects a trigger bit in the range 0 to 15 of CNTVCT_EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**EVNTDIR, bit [3]**

Controls which transition of the CNTVCT_EL0 trigger bit, as seen from EL1 and defined by EVNTI, generates an event when the event stream is enabled.

0b0: A 0 to 1 transition of the trigger bit triggers an event.

0b1: A 1 to 0 transition of the trigger bit triggers an event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**EVNTEN, bit [2]**

When FEAT_VHE is not implemented, or when HCR_EL2.{E2H, TGE} is not {1, 1}, enables the generation of an event stream from CNTVCT_EL0 as seen from EL1.

0b0: Disables the event stream.

0b1: Enables the event stream.
When $\text{FEAT\_VHE}$ is implemented and $\text{HCR\_EL2.\{E2H, TGE\}}$ is $\{1, 1\}$, this control does not enable the event stream.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**EL0VCTEN, bit [1]**

Traps EL0 accesses to the frequency register and virtual counter register to EL1, or to EL2 when it is implemented and enabled for the current Security state and $\text{HCR\_EL2.TGE}$ is 1, as follows:

- In AArch64 state, accesses to the following registers are trapped and reported using EC syndrome value $0\times18$:
  - $\text{CNTVCT\_EL0}$ and if $\text{CNTKCTL\_EL1.EL0PCTEN}$ is 0, $\text{CNTFRQ\_EL0}$.
- In AArch32 state, MRC and MCR accesses to the following registers are trapped and reported using EC syndrome value $0\times83$, MRR and MCRR accesses are trapped and reported using EC syndrome value $0\times04$:
  - $\text{CNTVCT}$ and if $\text{CNTKCTL\_EL1.EL0PCTEN}$ is 0, $\text{CNTFRQ}$.

0b0 EL0 accesses to the frequency register and virtual counter registers are trapped.

0b1 This control does not cause any instructions to be trapped.

When $\text{FEAT\_VHE}$ is implemented and $\text{HCR\_EL2.\{E2H, TGE\}}$ is $\{1, 1\}$, this control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**EL0PCTEN, bit [0]**

Traps EL0 accesses to the frequency register and physical counter register to EL1, or to EL2 when it is implemented and enabled for the current Security state and $\text{HCR\_EL2.TGE}$ is 1, as follows:

- In AArch64 state, the following registers are trapped, reported using EC syndrome value $0\times18$:
  - $\text{CNTPCT\_EL0}$ and if $\text{CNTKCTL\_EL1.EL0VCTEN}$ is 0, $\text{CNTFRQ\_EL0}$.
- In AArch32 state, MCR or MRC accesses the following registers are trapped, reported using EC syndrome value $0\times83$, MCRR or MRRC accesses are trapped and reported using EC syndrome value $0\times04$:
  - $\text{CNTPCT}$ and if $\text{CNTKCTL\_EL1.EL0VCTEN}$ is 0, $\text{CNTFRQ}$.

0b0 EL0 accesses to the frequency register and physical counter register are trapped.

0b1 This control does not cause any instructions to be trapped.

When $\text{FEAT\_VHE}$ is implemented and $\text{HCR\_EL2.\{E2H, TGE\}}$ is $\{1, 1\}$, this control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing CNTKCTL\_EL1

When $\text{HCR\_EL2.E2H}$ is 1, without explicit synchronization, access from EL3 using the mnemonic $\text{CNTKCTL\_EL1}$ or $\text{CNTKCTL\_EL12}$ are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, CNTKCTL_EL1

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b00</td>
<td>0b1110</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elif PSTATE_EL == EL1 then
X[t, 64] = CNTKCTL_EL1;
elif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = CNTKCTL_EL2;
  else
    X[t, 64] = CNTKCTL_EL1;
elif PSTATE_EL == EL3 then
X[t, 64] = CNTKCTL_EL1;

MSR CNTKCTL_EL1, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b00</td>
<td>0b1110</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elif PSTATE_EL == EL1 then
  CNTKCTL_EL1 = X[t, 64];
elif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    CNTKCTL_EL2 = X[t, 64];
  else
    CNTKCTL_EL1 = X[t, 64];
elif PSTATE_EL == EL3 then
  CNTKCTL_EL1 = X[t, 64];

MRS <Xt>, CNTKCTL_EL12

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b10</td>
<td>0b1110</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = CNTKCTL_EL1;
  else
    UNDEFINED;
elif PSTATE_EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    X[t, 64] = CNTKCTL_EL1;
else
    UNDEFINED;

**MSR CNTKCTL_EL12, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1110</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CNTKCTL_EL1 = X[t, 64];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        CNTKCTL_EL1 = X[t, 64];
    else
        UNDEFINED;
D17.11.16  CNTP_CTL_EL0, Counter-timer Physical Timer Control register

The CNTP_CTL_EL0 characteristics are:

**Purpose**
Control register for the EL1 physical timer.

**Configurations**
AArch64 System register CNTP_CTL_EL0 bits [31:0] are architecturally mapped to AArch32 System register CNTP_CTL[31:0].

**Attributes**
CNTP_CTL_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>62</td>
<td>ISTATUS, bit [2]</td>
</tr>
<tr>
<td>61</td>
<td>IMASK, bit [1]</td>
</tr>
<tr>
<td>60</td>
<td>ENABLE, bit [0]</td>
</tr>
</tbody>
</table>

**ISTATUS, bit [2]**
The status of the timer. This bit indicates whether the timer condition is met:
- 0b0 Timer condition is not met.
- 0b1 Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is **UNKNOWN**.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

Access to this field is RO.

**IMASK, bit [1]**
Timer interrupt mask bit. Permitted values are:
- 0b0 Timer interrupt is not masked by the IMASK bit.
- 0b1 Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**ENABLE, bit [0]**
Enables the timer. Permitted values are:
- 0b0 Timer disabled.
- 0b1 Timer enabled.
Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTP_TVVAL_EL0 continues to count down.

--- Note ---
Disabling the output signal might be a power-saving option.

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing CNTP_CTL_EL0

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CNTP_CTL_EL0 or CNTP_CTL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTP_CTL_EL0
```

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrapEl2(0x18);
      else
        AArch64.SystemAccessTrapEl1(0x18);
    elseif EL2Enabled() && HCR_EL2.E2H == '0' then
      X[t, 64] = CNTHPS_CTL_EL2;
    elseif EL2Enabled() && HCR_EL2.E2H == '1' then
      X[t, 64] = CNTHP_CTL_EL2;
    elseif IsFeatureImplemented(FEAT_SEL2) then
      X[t, 64] = NVMem[0x180];
      else
        X[t, 64] = CNTP_CTL_EL0;
  else
    X[t, 64] = CNTP_CTL_EL0;

elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.E2H == '0' then
    AArch64.SystemAccessTrapEl2(0x18);
  elseif EL2Enabled() && HCR_EL2.E2H == '1' then
    AArch64.SystemAccessTrapEl1(0x18);
  elseif IsFeatureImplemented(FEAT_SEL2) then
    X[t, 64] = CNTHPS_CTL_EL2;
  elseif IsFeatureImplemented(FEAT_SEL2) then
    X[t, 64] = CNTHP_CTL_EL2;
  else
    X[t, 64] = CNTP_CTL_EL0;
  else
    X[t, 64] = CNTP_CTL_EL0;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = CNTHPS_CTL_EL2;
  elseif HCR_EL2.E2H == '1' then
    X[t, 64] = CNTHP_CTL_EL2;
  else
    X[t, 64] = CNTP_CTL_EL0;
elsif PSTATE_EL == EL3 then
  X[t, 64] = CNTP_CTL_EL0;
```
### MSR CNTP_CTL_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```java
if PSTATE.EL == EL0 then
    if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTHCTL_EL1.EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
      IsFeatureImplemented(�T_SEL2) then
        CNTHPS_CTL_EL2 = X[t, 64];
    elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CTL_EL2 = X[t, 64];
    else
        CNTP_CTL_EL0 = X[t, 64];
    elseif PSTATE.EL == EL1 then
        if EL2Enabled() && HCR_EL2.<E2H,TGE> == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        elseif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' then
            NVMem[0x180] = X[t, 64];
        else
            CNTP_CTL_EL0 = X[t, 64];
    elseif PSTATE.EL == EL2 then
        if HCR_EL2.<E2H, TGE> == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(�T_SEL2) then
            CNTHPS_CTL_EL2 = X[t, 64];
        elseif HCR_EL2.<E2H, TGE> == '1' && SCR_EL3.NS == '1' then
            CNTHP_CTL_EL2 = X[t, 64];
        else
            CNTP_CTL_EL0 = X[t, 64];
    elseif PSTATE.EL == EL3 then
        CNTP_CTL_EL0 = X[t, 64];
```

### MRS <Xt>, CNTP_CTL_EL02

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```java
if PSTATE.EL == EL0 then
    UNDEFINED;
else if PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1NVPCT == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            X[t, 64] = NVMem[0x180];
    elseif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        AArch64.SystemAccessTrap(EL2, 0x18);
else if PSTATE.EL == EL3 then
    CNTP_CTL_EL0 = X[t, 64];
```

---

**AArch64 System Register Descriptions**

**D17.11 Generic Timer registers**

**Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.**

**Non-Confidential**
UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = CNTP_CTL_EL0;
  else
    UNDEFINED;
elsif PSTATE.EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    X[t, 64] = CNTP_CTL_EL0;
  else
    UNDEFINED;

**MSR CNTP_CTL_EL02, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b10</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.<NV2,NV1,NV> == '101' then
    if EL2Enabled() & HCR_EL2.<E2H,TGE> != '11' & CNTHCTL_EL2.EL1NPCT == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      NVMem[0x180] = X[t, 64];
    elsif EL2Enabled() & HCR_EL2.NV == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    CNTP_CTL_EL0 = X[t, 64];
  else
    UNDEFINED;
elsif PSTATE.EL == EL3 then
  if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.E2H == '1' then
    CNTP_CTL_EL0 = X[t, 64];
  else
    UNDEFINED;
D17.11.17  CNTP_CVAL_EL0, Counter-timer Physical Timer CompareValue register

The CNTP_CVAL_EL0 characteristics are:

**Purpose**

Holds the compare value for the EL1 physical timer.

**Configurations**

AArch64 System register CNTP_CVAL_EL0 bits [63:0] are architecturally mapped to AArch32 System register CNTP_CVAL[63:0].

**Attributes**

CNTP_CVAL_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompareValue</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompareValue</td>
<td></td>
</tr>
</tbody>
</table>

**CompareValue, bits [63:0]**

Holds the EL1 physical timer CompareValue.

When CNTP_CTL_EL0.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTP_CTL_EL0.ISTATUS is set to 1.
- If CNTP_CTL_EL0.IMASK is 0, an interrupt is generated.

When CNTP_CTL_EL0.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTP_CVAL_EL0**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CNTP_CVAL_EL0 or CNTP_CVAL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, CNTP_CVAL_EL0

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else if EL2Enabled() && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else if EL2Enabled() && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
    IsFeatureImplemented(FEAT_SEL2) then
        X[t, 64] = CNTHPS_CVAL_EL2;
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        X[t, 64] = CNTHP_CVAL_EL2;
    else
        X[t, 64] = CNTP_CVAL_EL0;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else if EL2Enabled() && HCR_EL2.<E2H,TGE> == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
        IsFeatureImplemented(FEAT_SEL2) then
            X[t, 64] = CNTHPS_CVAL_EL2;
        elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
            X[t, 64] = CNTHP_CVAL_EL2;
        else
            X[t, 64] = CNTP_CVAL_EL0;
    elsif PSTATE.EL == EL2 then
        if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
            X[t, 64] = CNTHPS_CVAL_EL2;
        elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
            X[t, 64] = CNTHP_CVAL_EL2;
        else
            X[t, 64] = CNTP_CVAL_EL0;
    elsif PSTATE.EL == EL3 then
        X[t, 64] = CNTP_CVAL_EL0;

MSR CNTP_CVAL_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else if EL2Enabled() && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else if EL2Enabled() && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
    IsFeatureImplemented(FEAT_SEL2) then
        X[t, 64] = CNTHPS_CVAL_EL2;
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        X[t, 64] = CNTHP_CVAL_EL2;
    else
        X[t, 64] = CNTP_CVAL_EL0;
```
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FEAT_SEL2) then
  CNTHPS_CVAL_EL2 = X[t, 64];
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
  CNTHP_CVAL_EL2 = X[t, 64];
else
  CNTP_CVAL_EL0 = X[t, 64];
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
    NVMem[0x178] = X[t, 64];
  else
    CNTP_CVAL_EL0 = X[t, 64];
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
    CNTHPS_CVAL_EL2 = X[t, 64];
  elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
    CNTHP_CVAL_EL2 = X[t, 64];
  else
    CNTP_CVAL_EL0 = X[t, 64];
elsif PSTATE_EL == EL3 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = CNTP_CVAL_EL0;
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = CNTP_CVAL_EL0;
  else
    UNDEFINED;
elsif PSTATE_EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    X[t, 64] = CNTP_CVAL_EL0;
else
  UNDEFINED;
```

### MRS <Xt>, CNTP_CVAL_EL02

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1NVPCT == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      X[t, 64] = NVMem[0x178];
    endif
  elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  if HCR_EL2.E2H == '1' then
    X[t, 64] = CNTP_CVAL_EL0;
  else
    UNDEFINED;
elsif PSTATE_EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    X[t, 64] = CNTP_CVAL_EL0;
else
  UNDEFINED;
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1NVPCT == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            NVMem[0x178] = X[t, 64];
        endif
    endif
elsif EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CNTP_CVAL_EL0 = X[t, 64];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        CNTP_CVAL_EL0 = X[t, 64];
    else
        UNDEFINED;
else
    UNDEFINED;
end
D17.11.18 CNTP_TVAL_EL0, Counter-timer Physical Timer TimerValue register

The CNTP_TVAL_EL0 characteristics are:

**Purpose**

Holds the timer value for the EL1 physical timer.

**Configurations**

AArch64 System register CNTP_TVAL_EL0 bits [31:0] are architecturally mapped to AArch32 System register CNTP_TVAL[31:0].

**Attributes**

CNTP_TVAL_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>31-0</td>
<td>TimerValue</td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**TimerValue, bits [31:0]**

The TimerValue view of the EL1 physical timer.

On a read of this register:
- If CNTP_CTL_EL0.ENABLE is 0, the value returned is UNKNOWN.
- If CNTP_CTL_EL0.ENABLE is 1, the value returned is (CNTP_CVAL_EL0 - CNTPCT_EL0).

On a write of this register, CNTP_CVAL_EL0 is set to (CNTPCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTP_CTL_EL0.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CNTP_CVAL_EL0) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:
- CNTP_CTL_EL0.ISTATUS is set to 1.
- If CNTP_CTL_EL0.IMASK is 0, an interrupt is generated.

When CNTP_CTL_EL0.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count, so the TimerValue view appears to continue to count down.

--- Note ---

The value of CNTPCT_EL0 used in these calculations is the value seen at the Exception Level that the CNTPCT_EL0 register is being read or written from.

---

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTP_TVAL_EL0**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CNTP_TVAL_EL0 or CNTP_TVAL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.
Accesses to this register use the following encodings in the System register encoding space:

\[ \text{MRS} \ <Xt>, \ \text{CNTP\_TVAL\_EL0} \]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if (EL2Enabled() && HCR_EL2.E2H.TGE == '11') && CNTHCTL_EL1.EL0PTEN == '0' then
if EL2Enabled() && HCR_EL2.TGE == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
AArch64.SystemAccessTrap(EL1, 0x18);
elseif EL2Enabled() && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elseif EL2Enabled() && HCR_EL2.E2H == '10' && CNTHCTL_EL2.EL1PTEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elseif EL2Enabled() && HCR_EL2.E2H == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elseif EL2Enabled() && HCR_EL2.E2H == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FEATURE_SEL2) then
if CNTHPS_CTL_EL2.ENABLE == '0' then
X[t, 64] = bits(64) UNKNOWN;
else
X[t, 64] = CNTHPS_CVAL_EL2 - PhysicalCountInt();
elseif EL2Enabled() && HCR_EL2.E2H == '11' && SCR_EL3.NS == '1' then
X[t, 64] = CNTHP_CVAL_EL2 - PhysicalCountInt();
elseif IsFeatureImplemented(FEATURE_ECV) && EL2Enabled() && SCR_EL3.ECVEn == '1' && CNTHCTL_EL2.ECV ==
'1' && HCR_EL2.E2H.TGE == '11' then
if CNTP_CTL_EL0.ENABLE == '0' then
X[t, 64] = bits(64) UNKNOWN;
else
X[t, 64] = CNTP_CVAL_EL0 - (PhysicalCountInt() - CNTPOFF_EL2);
elseif CNTP_CTL_EL0.ENABLE == '0' then
X[t, 64] = bits(64) UNKNOWN;
else
X[t, 64] = CNTP_CVAL_EL0 - PhysicalCountInt();
elseif PSTATE.EL == EL1 then
if EL2Enabled() && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elseif EL2Enabled() && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elseif IsFeatureImplemented(FEATURE_ECV) && EL2Enabled() && SCR_EL3.ECVEn == '1' && CNTHCTL_EL2.ECV ==
'1' then
if CNTP_CTL_EL0.ENABLE == '0' then
X[t, 64] = bits(64) UNKNOWN;
else
X[t, 64] = CNTP_CVAL_EL0 - (PhysicalCountInt() - CNTPOFF_EL2);
elseif CNTP_CTL_EL0.ENABLE == '0' then
X[t, 64] = bits(64) UNKNOWN;
else
X[t, 64] = CNTP_CVAL_EL0 - PhysicalCountInt();
elseif PSTATE.EL == EL2 then
if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FEATURE_SEL2) then
if CNTHPS_CTL_EL2.ENABLE == '0' then
X[t, 64] = bits(64) UNKNOWN;
else
X[t, 64] = CNTHPS_CVAL_EL2 - PhysicalCountInt();
elseif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
X[t, 64] = CNTHP_CVAL_EL2 - PhysicalCountInt();
elseif SCR_EL3.NS == '1' then
X[t, 64] = CNTP_CVAL_EL0 - PhysicalCountInt();
elseif HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' then
X[t, 64] = CNTP_CVAL_EL0 - (PhysicalCountInt() - CNTPOFF_EL2);
else
X[t, 64] = CNTP_CVAL_EL0 - PhysicalCountInt();
elseif SCR_EL3.NS == '1' then
X[t, 64] = CNTP_CVAL_EL0 - PhysicalCountInt();
else
X[t, 64] = CNTP_CVAL_EL0 - PhysicalCountInt();
if CNTHP_CTL_EL2.ENABLE == '0' then
    X[t, 64] = bits(64) UNKNOWN;
else
    X[t, 64] = CNTHP_CVAL_EL2 - PhysicalCountInt();
else
    if CNTP_CTL_EL0.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = CNTP_CVAL_EL0 - PhysicalCountInt();
    endif
endif PSTATE.EL == EL3 then
    if CNTP_CTL_EL0.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = CNTP_CVAL_EL0 - PhysicalCountInt();
    endif
endif

MSR CNTP_TVAL_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if !(EL2Enabled() && HCR_EL2.E2H,TGE == '11') && CNTHCTL_EL1.EL0PTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            if SCR_EL3.NS == '0' && IsFeature Implemented(FEAT_SEL2) then
                CNTHPS_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
            endif
            if IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && SCR_EL3.ECVEn == '1' then
                CNTP_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTPOFF_EL2;
            else
                CNTP_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
            endif
        else
            CNTP_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
        endif
    endif
else
    CNTP_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
endif
MRS \(<Xt>, \text{CNTP\_TVAL\_EL02}\)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    if CNTP_CTL_EL0.ENABLE == '0' then
      \(X[t, 64] = \text{bits}(64) \text{ UNKNOWN}\);
    else
      \(X[t, 64] = \text{CNTP\_CVAL\_EL0} - \text{PhysicalCountInt()};\)
    else
      UNDEFINED;
  elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
      if CNTP_CTL_EL0.ENABLE == '0' then
        \(X[t, 64] = \text{bits}(64) \text{ UNKNOWN}\);
      else
        \(X[t, 64] = \text{CNTP\_CVAL\_EL0} - \text{PhysicalCountInt()};\)
      else
        UNDEFINED;
    else
      UNDEFINED;
MRS \(<Xt>, \text{CNTP\_TVAL\_EL02}\>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' then
    CNTP_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
  else
    UNDEFINED;
elsif PSTATE.EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
    CNTP_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
  else
    UNDEFINED;
D17.11.19 CNTPCTSS_EL0, Counter-timer Self-Synchronized Physical Count register

The CNTPCTSS_EL0 characteristics are:

**Purpose**

Holds the self-synchronized view of the 64-bit physical count value.

**Configurations**

AArch64 System register CNTPCTSS_EL0 bits [63:0] are architecturally mapped to AArch32 System register CNTPCTSS[63:0].

This register is present only when FEAT_ECV is implemented. Otherwise, direct accesses to CNTPCTSS_EL0 are UNDEFINED.

All reads to the CNTPCTSS_EL0 occur in program order relative to reads to CNTPCT_EL0 or CNTPCTSS_EL0.

This register is a self-synchronised view of the CNTPCT_EL0 counter, and cannot be read speculatively.

**Attributes**

CNTPCTSS_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self-synchronized physical count value</td>
</tr>
</tbody>
</table>

**Accessing CNTPCTSS_EL0**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, CNTPCTSS_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTHCTL_EL1.EL0PCTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    AArch64.SystemAccessTrap(EL1, 0x18);
elif EL2Enabled() && HCR_EL2.TGE == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elisf EL2Enabled() && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PCTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elisf EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PCTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
AArch64.SystemAccessTrap(El2, 0x18);
else
  if IsFeatureImplemented(FEAT_ECV) && El2Enabled() && SCR_EL3.ECVEn == '1' && CNTHCTL_EL2.ECV == '1' && HCR_EL2.<E2H,TGE> != '11' then
    X[t, 64] = PhysicalCountInt() - CNTPOFF_EL2;
  else
    X[t, 64] = PhysicalCountInt();
elsif PSTATE.EL == EL1 then
  if El2Enabled() && CNTHCTL_EL2.EL1PCTEN == '0' then
    AArch64.SystemAccessTrap(El2, 0x18);
  else
    if IsFeatureImplemented(FEAT_ECV) && El2Enabled() && SCR_EL3.ECVEn == '1' && CNTHCTL_EL2.ECV == '1' then
      X[t, 64] = PhysicalCountInt() - CNTPOFF_EL2;
    else
      X[t, 64] = PhysicalCountInt();
    elsif PSTATE.EL == EL2 then
      X[t, 64] = PhysicalCountInt();
    elsif PSTATE.EL == EL3 then
      X[t, 64] = PhysicalCountInt();
  else
    X[t, 64] = PhysicalCountInt();
### D17.11.20 CNTPCT_EL0, Counter-timer Physical Count register

The CNTPCT_EL0 characteristics are:

**Purpose**

Holds the 64-bit physical count value.

**Configurations**

AArch64 System register CNTPCT_EL0 bits [63:0] are architecturally mapped to AArch32 System register CNTPCT[63:0].

All reads to the CNTPCT_EL0 occur in program order relative to reads to CNTPCTSS_EL0 or CNTPCT_EL0.

**Attributes**

CNTPCT_EL0 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical count value</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical count value</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:0]**

Physical count value.

Reads of CNTPCT_EL0 from EL0 or EL1 return \((\text{PhysicalCountInt}_{63:0} - \text{CNTPOFF}_{EL2}_{63:0})\) if the access is not trapped, and all of the following are true:

- \(\text{CNTHCTL}_{EL2}.ECV\) is 1.
- \(\text{HCR}_{EL2}.\{E2H, TGE\}\) is not \(\{1, 1\}\).

Where \(\text{PhysicalCountInt}_{63:0}\) is the physical count returned when \(\text{CNTPCT}_{EL0}\) is read from EL2 or EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally \textit{UNKNOWN} value.

**Accessing CNTPCT_EL0**

Accesses to this register use the following encodings in the System register encoding space:

**MRS \(<Xt>, CNTPCT_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Crn</th>
<th>Crm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```bash
if PSTATE.EL == EL0 then
    if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PCTEN == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elseif EL2Enabled() && HCR_EL2.E2H == '0' && CNTKCTL_EL2.EL1PCTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PCTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PCTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    if IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && SCR_EL3.ECVEn == '1' && CNTHCTL_EL2.ECV ==
        '1' && HCR_EL2.<E2H,TGE> != '11' then
        X[t, 64] = PhysicalCountInt() - CNTPOFF_EL2;
    else
        X[t, 64] = PhysicalCountInt();
    end
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL_EL2.EL1PCTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && SCR_EL3.ECVEn == '1' && CNTHCTL_EL2.ECV ==
            '1' then
            X[t, 64] = PhysicalCountInt() - CNTPOFF_EL2;
        else
            X[t, 64] = PhysicalCountInt();
        end
    end
elsif PSTATE.EL == EL2 then
    X[t, 64] = PhysicalCountInt();
elsif PSTATE.EL == EL3 then
    X[t, 64] = PhysicalCountInt();
D17.11.21  CNTPS_CTL_EL1, Counter-timer Physical Secure Timer Control register

The CNTPS_CTL_EL1 characteristics are:

**Purpose**

Control register for the secure physical timer, usually accessible at EL3 but configurably accessible at EL1 in Secure state.

**Configurations**

There are no configuration notes.

**Attributes**

CNTPS_CTL_EL1 is a 64-bit register.

**Field descriptions**

```
+-------------------+-------------------+-------------------+
| Reserved, RES0    | Reserved, RES0    | Reserved, RES0    |
| The status of the timer. This bit indicates whether the timer condition is met: | Timer interrupt mask bit. Permitted values are: | Enables the timer. Permitted values are: |
| 0b0 Timer condition is not met. | 0b0 Timer interrupt is not masked by the IMASK bit. | 0b0 Timer disabled. |
| 0b1 Timer condition is met. | 0b1 Timer interrupt is masked by the IMASK bit. | 0b1 Timer enabled. |
```

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.
Setting this bit to 0 disables the timer output signal, but the timer value accessible from $\text{CNTPS\_TVAL\_EL1}$ continues to count down.

--- Note ---
Disabling the output signal might be a power-saving option.

---

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally \text{UNKNOWN} value.

**Accessing \text{CNTPS\_CTL\_EL1}**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS <Xt>, CNTPS\_CTL\_EL1}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b11</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

\[
\text{MSR CNTPS\_CTL\_EL1, <Xt>}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b11</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>
elsif PSTATE.EL == EL3 then
    CNTPS_CTL_EL1 = X[t, 64];
### D17.11.22 CNTPOFF_EL2, Counter-timer Physical Offset register

The CNTPOFF_EL2 characteristics are:

**Purpose**

Holds the 64-bit physical offset. This is the offset for the AArch64 physical timers and counters when Enhanced Counter Virtualization is enabled.

**Configurations**

This register is present only when FEAT_ECV is implemented. Otherwise, direct accesses to CNTPOFF_EL2 are UNDEFINED.

The CNTPOFF_EL2 offset applies to:

- Direct reads of the physical counter from EL0 or EL1.
- Indirect reads of the physical counter by the EL1 physical timer.

When EL2 is implemented and enabled in the current Security state, the physical counter uses a fixed physical offset of zero if any of the following are true:

- CNTHCTL_EL2.ECV is 0.
- SCR_EL3.ECVEn is 0.
- HCR_EL2.{E2H, TGE} is {1, 1}.

**Attributes**

CNTPOFF_EL2 is a 64-bit register.

**Field descriptions**

#### Bits [63:0]

Physical offset.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTPOFF_EL2**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, CNTPOFF_EL2**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0000</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = NVMem[0x1A8];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        Physical offset

32

Physical offset

11

Physical offset

0
AArch64.SystemAccessTrap(EL2, 0x18);
else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1'" && SCR_EL3.ECVEn == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.ECVEn == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = CNTPOFF_EL2;
    else
        X[t, 64] = CNTPOFF_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = CNTPOFF_EL2;
else
    CNTPOFF_EL2 = X[t, 64];
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x1A8] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1'" && SCR_EL3.ECVEn == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.ECVEn == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        CNTPOFF_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    CNTPOFF_EL2 = X[t, 64];

---

**MSR CNTPOFF_EL2, <Xt>**

<table>
<thead>
<tr>
<th></th>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0b11</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0000</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x1A8] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1'" && SCR_EL3.ECVEn == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.ECVEn == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        CNTPOFF_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    CNTPOFF_EL2 = X[t, 64];
D17.11.23  **CNTPS_CVAL_EL1, Counter-timer Physical Secure Timer CompareValue register**

The CNTPS_CVAL_EL1 characteristics are:

**Purpose**

Holds the compare value for the secure physical timer, usually accessible at EL3 but configurably accessible at EL1 in Secure state.

**Configurations**

There are no configuration notes.

**Attributes**

CNTPS_CVAL_EL1 is a 64-bit register.

**Field descriptions**

**CompareValue, bits [63:0]**

Holds the secure physical timer CompareValue.

When CNTPS_CTL_EL1.ENABLE is 1, the timer condition is met when \( \text{CNTPCT_EL0} - \text{CompareValue} \) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTPS_CTL_EL1.ISTATUS is set to 1.
- If CNTPS_CTL_EL1.IMASK is 0, an interrupt is generated.

When CNTPS_CTL_EL1.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTPS_CVAL_EL1**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, CNTPS_CVAL_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b111</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if HaveEL(EL3) && SCR_EL3.NS == '0' then
    if SCR_EL3.EEL2 == '1' then
UNDEFINED;
elsif SCU_EL3.ST == '0' then
  AArch64.SystemAccessTrap(EL3, 0x18);
else
  X[t, 64] = CNTPS_CVAL_EL1;
else
  UNDEFINED;
elsif PSTATE_EL == EL2 then
  UNDEFINED;
elsif PSTATE_EL == EL3 then
  X[t, 64] = CNTPS_CVAL_EL1;

**MSR CNTPS_CVAL_EL1, <xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b111</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if HaveEl(EL3) && SCU_EL3.NS == '0' then
    if SCU_EL3.EEL2 == '1' then
      UNDEFINED;
    elsif SCU_EL3.ST == '0' then
      AArch64.SystemAccessTrap(EL3, 0x18);
    else
      CNTPS_CVAL_EL1 = X[t, 64];
    else
      UNDEFINED;
  elsif PSTATE_EL == EL2 then
    UNDEFINED;
eif PSTATE_EL == EL3 then
  CNTPS_CVAL_EL1 = X[t, 64];
D17.11.24  CNTPS_TVAL_EL1, Counter-timer Physical Secure Timer TimerValue register

The CNTPS_TVAL_EL1 characteristics are:

**Purpose**

Holds the timer value for the secure physical timer, usually accessible at EL3 but configurably accessible at EL1 in Secure state.

**Configurations**

There are no configuration notes.

**Attributes**

CNTPS_TVAL_EL1 is a 64-bit register.

**Field descriptions**

![Register Diagram]

**Bits [63:32]**

Reserved, RES0.

**TimerValue, bits [31:0]**

The TimerValue view of the secure physical timer.

On a read of this register:
- If CNTPS_CTL_EL1.ENABLE is 0, the value returned is UNKNOWN.
- If CNTPS_CTL_EL1.ENABLE is 1, the value returned is (CNTPS_CVAL_EL1 - CNTPCT_EL0).

On a write of this register, CNTPS_CVAL_EL1 is set to (CNTPCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTPS_CTL_EL1.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CNTPS_CVAL_EL1) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:
- CNTPS_CTL_EL1.ISTATUS is set to 1.
- If CNTPS_CTL_EL1.IMASK is 0, an interrupt is generated.

When CNTPS_CTL_EL1.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTPS_TVAL_EL1**

Accesses to this register use the following encodings in the System register encoding space:
**MRS <Xt>, CNTPS_TVAL_EL1**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b11</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if HaveEL(EL3) && SCR_EL3.NS == '0' then
    if SCR_EL3.EEL2 == '1' then
      UNDEFINED;
    elsif SCR_EL3.ST == '0' then
      AArch64.SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && SCR_EL3.ECVEn == '1' && CNTHCTL_EL2.ECV == '1' then
      if CNTPS_CTL_EL1.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
      else
        X[t, 64] = CNTPS_CVAL_EL1 - (PhysicalCountInt() - CNTPOFF_EL2);
      endif
    else
      UNDEFINED;
    endif
  elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
  if CNTPS_CTL_EL1.ENABLE == '0' then
    X[t, 64] = bits(64) UNKNOWN;
  else
    X[t, 64] = CNTPS_CVAL_EL1 - PhysicalCountInt();
  endif
endif

**MSR CNTPS_TVAL_EL1, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b11</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if HaveEL(EL3) && SCR_EL3.NS == '0' then
    if SCR_EL3.EEL2 == '1' then
      UNDEFINED;
    elsif SCR_EL3.ST == '0' then
      AArch64.SystemAccessTrap(EL3, 0x18);
    elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && SCR_EL3.ECVEn == '1' && CNTHCTL_EL2.ECV == '1' then
      CNTPS_CVAL_EL1 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTPOFF_EL2;
    else
      CNTPS_CVAL_EL1 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    endif
  else
    UNDEFINED;
  endif
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  CNTPS_CVAL_EL1 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();

D17.11.25 CNTV_CTL_EL0, Counter-timer Virtual Timer Control register

The CNTV_CTL_EL0 characteristics are:

**Purpose**
Control register for the virtual timer.

**Configurations**
AArch64 System register CNTV_CTL_EL0 bits [31:0] are architecturally mapped to AArch32 System register CNTV_CTL[31:0].

**Attributes**
CNTV_CTL_EL0 is a 64-bit register.

**Field descriptions**

```
  63  32
     |   RES0
  31  24
     |   RES0
  30  23
     |   ISTATUS
  29  22
     |   IMASK
  28  21
     |   ENABLE
  27  20
     |   ENABLE
  26  19
     |   ENABLE
  25  18
     |   ENABLE
  24  17
     |   ENABLE
  23  16
     |   ENABLE
  22  15
     |   ENABLE
  21  14
     |   ENABLE
  20  13
     |   ENABLE
  19  12
     |   ENABLE
  18  11
     |   ENABLE
  17  10
     |   ENABLE
  16   9
     |   ENABLE
  15   8
     |   ENABLE
  14   7
     |   ENABLE
  13   6
     |   ENABLE
  12   5
     |   ENABLE
  11   4
     |   ENABLE
  10   3
     |   ENABLE
   9   2
     |   ENABLE
   8   1
     |   ENABLE
   7   0
     |   ENABLE
```

**Bits [63:3]**
Reserved, RES0.

**ISTATUS, bit [2]**
The status of the timer. This bit indicates whether the timer condition is met:

- 0b0 Timer condition is not met.
- 0b1 Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

**IMASK, bit [1]**
Timer interrupt mask bit. Permitted values are:

- 0b0 Timer interrupt is not masked by the IMASK bit.
- 0b1 Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ENABLE, bit [0]**
Enables the timer. Permitted values are:

- 0b0 Timer disabled.
- 0b1 Timer enabled.
D17.11 Generic Timer registers

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTV_TVAL_EL0 continues to count down.

--- Note ---

Disabling the output signal might be a power-saving option.

---

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing CNTV_CTL_EL0

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CNTV_CTL_EL0 or CNTV_CTL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

### MRS <Xt>, CNTV_CTL_EL0

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
    endif
  else
    AArch64.SystemAccessTrap(EL1, 0x18);
  endif
else
  if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL0.EL0VTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1VTEN == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
        X[t, 64] = CNTHVS_CTL_EL2;
      else
        X[t, 64] = CNTV_CTL_EL0;
      endif
    endif
  endif
else
  if PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL_EL2.EL1VTEN == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      if EL2Enabled() && HCR_EL2.<E2H,TGE> == '111' then
        X[t, 64] = NVMem[0x170];
      else
        X[t, 64] = CNTV_CTL_EL0;
      endif
    endif
  else
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
      X[t, 64] = CNTHVS_CTL_EL2;
    else
      if HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        X[t, 64] = CNTHV_CTL_EL2;
      else
        X[t, 64] = CNTV_CTL_EL0;
      endif
    endif
  endif
else
  if PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
      X[t, 64] = CNTHVS_CTL_EL2;
    else
      if HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        X[t, 64] = CNTHV_CTL_EL2;
      else
        X[t, 64] = CNTV_CTL_EL0;
      endif
    endif
  else
    if PSTATE.EL == EL3 then
      X[t, 64] = CNTV_CTL_EL0;
    endif
  endif
```
**MSR CNTV_CTL_EL0, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0' then
if EL2Enabled() && HCR_EL2.TGE == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL1.EL0VTEN == '0' then
AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTKCTL_EL1.EL1VT == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' & IsFeatureImplemented(FEAT_SEL2) then
CNTVS_CTL_EL2 = X[t, 64];
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
CNTV_CTL_EL2 = X[t, 64];
else
CNTV_CTL_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
if EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL2.EL0VTEN == '0' then
AArch64.SystemAccessTrap(EL2, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL2.EL1VT == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
CNTV_CTL_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' & IsFeatureImplemented(FEAT_SEL2) then
CNTVS_CTL_EL2 = X[t, 64];
elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
CNTV_CTL_EL2 = X[t, 64];
else
CNTV_CTL_EL0 = X[t, 64];
elsif PSTATE.EL == EL3 then
CNTV_CTL_EL0 = X[t, 64];

**MRS <Xt>, CNTV_CTL_EL02**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
if EL2Enabled() && HCR_EL2.<E2H,EGE> != '11' && CNTKCTL_EL2.EL0VT == '1' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
X[t, 64] = NVMem[0x170];
elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
AArch64.SystemAccessTrap(EL2, 0x18);
else
UNDEFINED;
elsif PSTATE.EL == EL2 then
if HCR_EL2.E2H == '1' then
X[t, 64] = CNTV_CTL_EL0;
else
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        X[t, 64] = CNTV_CTL_EL0;
    else
        UNDEFINED;

    MSR CNTV_CTL_EL0, <Xt>

    if PSTATE.EL == EL0 then
        UNDEFINED;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
            if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1NVVCT == '1' then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                NVMem[0x170] = X[t, 64];
            elsif EL2Enabled() && HCR_EL2.NV == '1' then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                UNDEFINED;
        elsif PSTATE.EL == EL2 then
            if HCR_EL2.E2H == '1' then
                CNTV_CTL_EL0 = X[t, 64];
            else
                UNDEFINED;
        elsif PSTATE.EL == EL3 then
            if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
                CNTV_CTL_EL0 = X[t, 64];
            else
                UNDEFINED;

    op0 op1 CRn CRm op2
    0b11 0b101 0b1110 0b0011 0b001
D17.11.26 CNTV_CVAL_EL0, Counter-timer Virtual Timer CompareValue register

The CNTV_CVAL_EL0 characteristics are:

**Purpose**
Holds the compare value for the virtual timer.

**Configurations**
AArch64 System register CNTV_CVAL_EL0 bits [63:0] are architecturally mapped to AArch32 System register CNTV_CVAL[63:0].

**Attributes**
CNTV_CVAL_EL0 is a 64-bit register.

**Field descriptions**

```
<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CompareValue</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**CompareValue, bits [63:0]**

Holds the EL1 virtual timer CompareValue.

When CNTV_CTL_EL0.ENABLE is 1, the timer condition is met when (CNTVCT_EL0 - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTV_CTL_EL0.ISTATUS is set to 1.
- If CNTV_CTL_EL0.IMASK is 0, an interrupt is generated.

When CNTV_CTL_EL0.ENABLE is 0, the timer condition is not met, but CNTVCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTV_CVAL_EL0**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CNTV_CVAL_EL0 or CNTV_CVAL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:
MRS <Xt>, CNTV\_CVAL\_EL0

if PSTATE.EL == EL0 then
  if !(EL2Enabled() && HCR\_EL2.\(<E2H, TGE>=’11’\)) && CNTKCTL\_EL1.EL0VTEN == ’0’ then
    if EL2Enabled() && HCR\_EL2.TGE == ’1’ then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  endif
endif

MSR CNTV\_CVAL\_EL0, <Xt>

if PSTATE.EL == EL0 then
  if !(EL2Enabled() && HCR\_EL2.\(<E2H, TGE>=’11’\)) && CNTKCTL\_EL1.EL0VTEN == ’0’ then
    if EL2Enabled() && HCR\_EL2.TGE == ’1’ then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
  endif
endif
else
  CNTV_CVAL_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && CNTHCTL_EL2.EL1TVD == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled() && SCR_EL2.<NV2,NV1,NV> == '111' then
    NVMem[0x168] = X[t, 64];
  else
    CNTV_CVAL_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '1' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
    CNTHVS_CVAL_EL2 = X[t, 64];
  elsif HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
    CNTHV_CVAL_EL2 = X[t, 64];
  else
    CNTV_CVAL_EL0 = X[t, 64];
elsif PSTATE.EL == EL3 then
  CNTV_CVAL_EL0 = X[t, 64];

MRS <Xt>, CNTV_CVAL_EL02

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1NVVCT == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      X[t, 64] = NVMem[0x168];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      UNDEFINED;
  elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
      X[t, 64] = CNTV_CVAL_EL0;
    else
      UNDEFINED;
  elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
      X[t, 64] = CNTV_CVAL_EL0;
    else
      UNDEFINED;

MSR CNTV_CVAL_EL02, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
    if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1NVVCT == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
else
    NVMem[0x168] = X[t, 64];
elsif EL2Enabled() & HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CNTV_CVAL_EL0 = X[t, 64];
    else
        UNDEFINED;
elsif PSTATE_EL == EL3 then
    if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.E2H == '1' then
        CNTV_CVAL_EL0 = X[t, 64];
    else
        UNDEFINED;
D17.11.27  CNTV_TVAL_EL0, Counter-timer Virtual Timer TimerValue register

The CNTV_TVAL_EL0 characteristics are:

**Purpose**
Holds the timer value for the EL1 virtual timer.

**Configurations**
AArch64 System register CNTV_TVAL_EL0 bits [31:0] are architecturally mapped to AArch32 System register CNTV_TVAL[31:0].

**Attributes**
CNTV_TVAL_EL0 is a 64-bit register.

**Field descriptions**

![Field diagram]

**Bits [63:32]**
Reserved, RES0.

**TimerValue, bits [31:0]**
The TimerValue view of the EL1 virtual timer.

On a read of this register:
- If CNTV_CTL_EL0.ENABLE is 0, the value returned is UNKNOWN.
- If CNTV_CTL_EL0.ENABLE is 1, the value returned is (CNTV_CVAL_EL0 - CNTVCT_EL0).

On a write of this register, CNTV_CVAL_EL0 is set to (CNTVCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTV_CTL_EL0.ENABLE is 1, the timer condition is met when (CNTVCT_EL0 - CNTV_CVAL_EL0) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:
- CNTV_CTL_EL0.ISTATUS is set to 1.
- If CNTV_CTL_EL0.IMASK is 0, an interrupt is generated.

When CNTV_CTL_EL0.ENABLE is 0, the timer condition is not met, but CNTVCT_EL0 continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTV_TVAL_EL0**

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CNTV_TVAL_EL0 or CNTV_TVAL_EL02 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:
MRS \text{<Xt>}, CNTV\_TVAL\_EL0

\begin{tabular}{|c|c|c|c|c|}
\hline
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
\hline
0b11 & 0b011 & 0b1110 & 0b0011 & 0b000 \\
\hline
\end{tabular}

if \text{PSTATE.EL} == \text{EL0} then
\begin{align*}
\text{if} & \ (\text{EL2Enabled()} \&\& \text{HCR\_EL2.\text{<E2H,TGE>} == '11'} \&\& \text{CNTKCTL\_EL1.EL0VTEN == '0'}) \text{ then} \\
& \text{if EL2Enabled()} \&\& \text{HCR\_EL2.\text{TGE} == '1'} \text{ then} \\
& \quad \AArch64.SystemAccessTrap(\text{EL2}, 0x18); \\
& \text{else} \\
& \quad \AArch64.SystemAccessTrap(\text{EL1}, 0x18); \\
& \text{elsif EL2Enabled()} \&\& \text{HCR\_EL2.\text{<E2H,TGE>} == '11'} \&\& \text{CNTKCTL\_EL2.EL0VTEN == '0'} \text{ then} \\
& \quad \AArch64.SystemAccessTrap(\text{EL2}, 0x18); \\
& \text{elsif EL2Enabled()} \&\& \text{HCR\_EL2.\text{<E2H,TGE>} != '11'} \&\& \text{CNTKCTL\_EL2.EL1VTEN == '1'} \text{ then} \\
& \quad \AArch64.SystemAccessTrap(\text{EL1}, 0x18); \\
& \text{elsif EL2Enabled()} \&\& \text{HCR\_EL2.\text{<E2H,TGE>} == '11'} \&\& \text{SCR\_EL3.NS == '0'} \&\& \\
& \quad \text{IsFeatureImplemented(\text{FEAT\_SEL2}) then} \\
& \quad \text{if CNTHSV\_CTL\_EL2.ENABLE == '0'} \text{ then} \\
& \quad \quad \text{X[t, 64]} = \text{bits(64)} \ \text{UNKNOWN}; \\
& \quad \text{else} \\
& \quad \quad \text{X[t, 64]} = \text{CNTHSV\_CVAL\_EL2} - \text{PhysicalCountInt();} \\
& \text{elsif EL2Enabled()} \&\& \text{HCR\_EL2.\text{<E2H,TGE>} == '11'} \&\& \text{SCR\_EL3.NS == '1'} \text{ then} \\
& \quad \text{if CNTHSV\_CTL\_EL2.ENABLE == '0'} \text{ then} \\
& \quad \quad \text{X[t, 64]} = \text{bits(64)} \ \text{UNKNOWN}; \\
& \text{elsif EL2Enabled()} \&\& \text{HCR\_EL2.\text{<E2H,TGE>} != '11'} \&\& \text{SCR\_EL3.NS == '0'} \&\& \\
& \quad \text{IsFeatureImplemented(\text{FEAT\_SEL2}) then} \\
& \quad \text{if CNTHSV\_CTL\_EL2.ENABLE == '0'} \text{ then} \\
& \quad \quad \text{X[t, 64]} = \text{bits(64)} \ \text{UNKNOWN}; \\
& \text{else} \\
& \quad \quad \text{X[t, 64]} = \text{CNTHSV\_CVAL\_EL2} - \text{PhysicalCountInt();} \\
& \text{elsif PSTATE.EL == \text{EL1} then}
\end{align*}

\begin{align*}
\text{if} & \ (\text{EL2Enabled()} \&\& \text{CNTKCTL\_EL2.EL1VTEN == '0'}) \text{ then} \\
& \text{if EL2Enabled()} \&\& \text{HCR\_EL2.\text{TGE} == '1'} \text{ then} \\
& \quad \AArch64.SystemAccessTrap(\text{EL2}, 0x18); \\
& \text{else} \\
& \quad \AArch64.SystemAccessTrap(\text{EL1}, 0x18); \\
& \text{elsif EL2Enabled()} \&\& \text{HCR\_EL2.\text{<E2H,TGE>} == '11'} \&\& \text{CNTKCTL\_EL2.EL0VTEN == '0'} \text{ then} \\
& \quad \AArch64.SystemAccessTrap(\text{EL2}, 0x18); \\
& \text{elsif EL2Enabled()} \&\& \text{HCR\_EL2.\text{<E2H,TGE>} != '11'} \&\& \text{CNTKCTL\_EL2.EL1VTEN == '1'} \text{ then} \\
& \quad \AArch64.SystemAccessTrap(\text{EL1}, 0x18); \\
& \text{elsif EL2Enabled()} \&\& \text{HCR\_EL2.\text{<E2H,TGE>} == '11'} \&\& \text{SCR\_EL3.NS == '0'} \&\& \\
& \quad \text{IsFeatureImplemented(\text{FEAT\_SEL2}) then} \\
& \quad \text{if CNTTV\_CTL\_EL0.ENABLE == '0'} \text{ then} \\
& \quad \quad \text{X[t, 64]} = \text{bits(64)} \ \text{UNKNOWN}; \\
& \text{else} \\
& \quad \quad \text{X[t, 64]} = \text{CNTTV\_CVAL\_EL0} - \text{PhysicalCountInt() - CNTVOFF\_EL2}; \\
& \text{elsif PSTATE.EL == \text{EL2} then}
\end{align*}

\begin{align*}
\text{if} & \ (\text{EL2Enabled()} \&\& \text{CNTKCTL\_EL2.EL1VTEN == '1'}) \text{ then} \\
& \text{if EL2Enabled()} \&\& \text{HCR\_EL2.\text{TGE} == '1'} \text{ then} \\
& \quad \AArch64.SystemAccessTrap(\text{EL2}, 0x18); \\
& \text{else} \\
& \quad \AArch64.SystemAccessTrap(\text{EL1}, 0x18); \\
& \text{elsif HaveEL(\text{EL2}) \&\& EL2Enabled() \&\& \text{HCR\_EL2.\text{<E2H,TGE>} == '11'}} \text{ then} \\
& \quad \text{if CNTTV\_CTL\_EL0.ENABLE == '0'} \text{ then} \\
& \quad \quad \text{X[t, 64]} = \text{bits(64)} \ \text{UNKNOWN}; \\
& \text{elsif PSTATE.EL == \text{EL1} then} \\
\end{align*}

\begin{align*}
\text{if} & \ (\text{HCR\_EL2.\text{<E2H,TGE>} == '1'} \&\& \text{SCR\_EL3.NS == '0'}) \&\& \text{IsFeatureImplemented(\text{FEAT\_SEL2}) then} \\
& \text{if CNTTV\_CTL\_EL2.ENABLE == '0'} \text{ then} \\
& \quad \text{X[t, 64]} = \text{bits(64)} \ \text{UNKNOWN}; \\
& \text{else} \\
& \quad \text{X[t, 64]} = \text{CNTTV\_CVAL\_EL2} - \text{PhysicalCountInt();} \\
& \text{elsif PSTATE.EL == \text{EL2} then} \\
\end{align*}

\begin{align*}
\text{if} & \ (\text{HCR\_EL2.\text{<E2H,TGE>} == '1'} \&\& \text{SCR\_EL3.NS == '1'}) \&\& \text{IsFeatureImplemented(\text{FEAT\_SEL2}) then} \\
& \text{if CNTTV\_CTL\_EL2.ENABLE == '0'} \text{ then} \\
& \quad \text{X[t, 64]} = \text{bits(64)} \ \text{UNKNOWN}; \\
& \text{else} \\
& \quad \text{X[t, 64]} = \text{CNTTV\_CVAL\_EL2} - \text{PhysicalCountInt();} \\
& \text{elsif PSTATE.EL == \text{EL1} then} \\
\end{align*}

\begin{align*}
\text{if} & \ (\text{HCR\_EL2.\text{<E2H,TGE>} == '0'}) \&\& \text{IsFeatureImplemented(\text{FEAT\_SEL2}) then} \\
& \text{if CNTTV\_CTL\_EL2.ENABLE == '0'} \text{ then} \\
& \quad \text{X[t, 64]} = \text{bits(64)} \ \text{UNKNOWN}; \\
& \text{else} \\
& \quad \text{X[t, 64]} = \text{CNTTV\_CVAL\_EL2} - \text{PhysicalCountInt();} \\
& \text{elsif PSTATE.EL == \text{EL2} then} \\
\end{align*}
X[t, 64] = CNTV_CVAL_EL0 - (PhysicalCountInt() - CNTVOFF_EL2);
else
    if CNTV_CTL_EL0.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = CNTV_CVAL_EL0 - PhysicalCountInt();
elsif PSTATE_EL == EL3 then
    if CNTV_CTL_EL0.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    elsif HaveEL(EL2) && !ELUsingAArch32(EL2) then
        X[t, 64] = CNTV_CVAL_EL0 - (PhysicalCountInt() - CNTVOFF_EL2);
    elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
        X[t, 64] = CNTV_CVAL_EL0 - (PhysicalCountInt() - CNTVOFF);
    else
        X[t, 64] = CNTV_CVAL_EL0 - PhysicalCountInt();

MSR CNTV_TVAL_EL0, <Xt>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    end
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL0.EL0VTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && SCR_EL3 NS == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3 NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3 NS == '1' then
        CNTHV_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    elsif EL2Enabled() && ((EL2Enabled() || HCR_EL2.<E2H,TGE> == '11') then
        CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF_EL2;
    else
        CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && CNTHCTL_EL2.EL1VT == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL2) then
        CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF_EL2;
    else
        CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
elsif PSTATE_EL == EL2 then
    if !EL2Enabled() then
        CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    elsif HCR_EL2.E2H == '1' && SCR_EL3 NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
        CNTHVS_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    elsif HCR_EL2.E2H == '1' && SCR_EL3 NS == '1' then
        CNTHV_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
    elsif HCR_EL2.E2H == '0' then
        CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF_EL2;
    else
        CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
elsif PSTATE_EL == EL3 then
    if HaveEL(EL2) && !ELUsingAArch32(EL2) then
        CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF_EL2;
    elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
        CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF;
else
    CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();

**MRS <Xt>, CNTV_TVAL_EL02**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if CNTV_CTL_EL0.ENABLE == '0' then
            X[t, 64] = bits(64) UNKNOWN;
        else
            X[t, 64] = CNTV_CVAL_EL0 - (PhysicalCountInt() - CNTVOFF_EL2);
        else
            UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if CNTV_CTL_EL0.ENABLE == '0' then
            X[t, 64] = bits(64) UNKNOWN;
        else
            X[t, 64] = CNTV_CVAL_EL0 - (PhysicalCountInt() - CNTVOFF_EL2);
        else
            UNDEFINED;

**MSR CNTV_TVAL_EL02, <Xt>**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b101</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF_EL2;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF_EL2;
    else
        UNDEFINED;
D17.11.28 CNTVCTSS_EL0, Counter-timer Self-Synchronized Virtual Count register

The CNTVCTSS_EL0 characteristics are:

**Purpose**

Holds the 64-bit virtual count value. The virtual count value is equal to the physical count value visible in CNTPCT_EL0 minus the virtual offset visible in CNTVOFF_EL2.

**Configurations**

AArch64 System register CNTVCTSS_EL0 bits [63:0] are architecturally mapped to AArch32 System register CNTVCTSS[63:0].

This register is present only when FEAT_ECV is implemented. Otherwise, direct accesses to CNTVCTSS_EL0 are UNDEFINED.

All reads to the CNTVCTSS_EL0 occur in program order relative to reads to CNTVCT_EL0 or CNTVCTSS_EL0.

This register is a self-synchronised view of the CNTVCT_EL0 counter, and cannot be read speculatively.

**Attributes**

CNTVCTSS_EL0 is a 64-bit register.

**Field descriptions**

![Field descriptions diagram](image)

**Bits [63:0]**

Self-synchronized virtual count value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTVCTSS_EL0**

Accesses to this register use the following encodings in the System register encoding space:

**MRS <Xt>, CNTVCTSS_EL0**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0000</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VCTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL1.EL0VCTEN == '0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else if EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && CNTKCTL_EL1.EL1TVCT == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL2, 0x18);
else
  if HaveEl(El2) && (!EL2Enabled() || HCR_EL2.<E2H,TGE> != '1') then
    X[t, 64] = PhysicalCountInt() - CNTVOFF_EL2;
  else
    X[t, 64] = PhysicalCountInt();
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && CNTHCTL_EL2.EL1TVCT == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if HaveEl(El2) then
      X[t, 64] = PhysicalCountInt() - CNTVOFF_EL2;
    else
      X[t, 64] = PhysicalCountInt();
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '0' then
    X[t, 64] = PhysicalCountInt() - CNTVOFF_EL2;
  else
    X[t, 64] = PhysicalCountInt();
elsif PSTATE.EL == EL3 then
  if HaveEl(El2) && !ELUsingAArch32(El2) then
    X[t, 64] = PhysicalCountInt() - CNTVOFF_EL2;
  elsif HaveEl(El2) && ELUsingAArch32(El2) then
    X[t, 64] = PhysicalCountInt() - CNTVOFF;
  else
    X[t, 64] = PhysicalCountInt();
D17.11.29 CNTVCT_EL0, Counter-timer Virtual Count register

The CNTVCT_EL0 characteristics are:

**Purpose**

Holds the 64-bit virtual count value. The virtual count value is equal to the physical count value minus the virtual offset visible in CNTVOFF_EL2.

**Configurations**

AArch64 System register CNTVCT_EL0 bits [63:0] are architecturally mapped to AArch32 System register CNTVCT[63:0].

The value of this register is the same as the value of CNTPCT_EL0 in the following conditions:

- When EL2 is not implemented.
- When EL2 is implemented, HCR_EL2.E2H is 1, and this register is read from EL2.
- When EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} is {1, 1}, and this register is read from EL0 or EL2.

All reads to the CNTVCT_EL0 occur in program order relative to reads to CNTVCTSS_EL0 or CNTVCT_EL0.

**Attributes**

CNTVCT_EL0 is a 64-bit register.

**Field descriptions**

![Virtual count value](image)

**Bits [63:0]**

Virtual count value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTVCT_EL0**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS } <Xt>, \text{ CNTVCT_EL0}
\]

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11</td>
<td>0b011</td>
<td>0b1110</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if !(EL2Enabled() & HCR_EL2.<E2H,TGE> == '11') & CNTKCTL_EL1.EL0VCTEN == 0' then
    if EL2Enabled() & HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
    else
      AArch64.SystemAccessTrap(EL1, 0x18);
  elsif EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & CNTHCTL_EL2.EL0VCTEN == 0' then
    AArch64.SystemAccessTrap(EL2, 0x18);
else
  AArch64.SystemAccessTrap(EL1, 0x18);
elsif EL2Enabled() && HCR_EL2.<E2H, TGE> != '11' && CNTHCTL_EL2.EL1TVCT == '1' then
  AArch64.SystemAccessTrap(EL2, 0x18);
else
  if HaveEL(EL2) && (EL2Enabled() || HCR_EL2.<E2H, TGE> != '11') then
    X[t, 64] = PhysicalCountInt() - CNTVOFF_EL2;
  else
    X[t, 64] = PhysicalCountInt();
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && CNTHCTL_EL2.EL1TVCT == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    if HaveEL(EL2) then
      X[t, 64] = PhysicalCountInt() - CNTVOFF_EL2;
    else
      X[t, 64] = PhysicalCountInt();
elsif PSTATE.EL == EL2 then
  if HCR_EL2.E2H == '0' then
    X[t, 64] = PhysicalCountInt() - CNTVOFF_EL2;
  else
    X[t, 64] = PhysicalCountInt();
elsif PSTATE.EL == EL3 then
  if HaveEL(EL2) && !ELUsingAArch32(EL2) then
    X[t, 64] = PhysicalCountInt() - CNTVOFF_EL2;
elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
  X[t, 64] = PhysicalCountInt() - CNTVOFF;
else
  X[t, 64] = PhysicalCountInt();
D17.11.30 CNTVOFF_EL2, Counter-timer Virtual Offset register

The CNTVOFF_EL2 characteristics are:

**Purpose**

Holds the 64-bit virtual offset. This is the offset between the physical count value visible in CNTPCT_EL0 and the virtual count value visible in CNTVCT_EL0.

**Configurations**

AArch64 System register CNTVOFF_EL2 bits [63:0] are architecturally mapped to AArch32 System register CNTVOFF[63:0].

If EL2 is not implemented, this register is RES0 from EL3 and the virtual counter uses a fixed virtual offset of zero.

--- Note ---

When EL2 is implemented and enabled in the current Security state, and is using AArch64, the virtual counter uses a fixed virtual offset of zero in the following situations:

- HCR_EL2.E2H is 1, and CNTVCT_EL0 is read from EL2.
- HCR_EL2.{E2H, TGE} is {1, 1}, and either:
  - CNTVCT_EL0 is read from EL0 or EL2.
  - CNTVCT is read from EL0.

---

**Attributes**

CNTVOFF_EL2 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:0</td>
<td>Virtual offset</td>
</tr>
</tbody>
</table>

The value of this field is treated as zero-extended in all counter calculations. The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTVOFF_EL2**

Accesses to this register use the following encodings in the System register encoding space:
MRS \( <X_t>, CNTVOFF_EL2 \)

\[
\begin{array}{c|c|c|c|c}
\hline
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
\hline
0b11 & 0b100 & 0b1110 & 0b0000 & 0b011 \\
\hline
\end{array}
\]

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.<NV2,NV> == '11' then
    X[t, 64] = NVMem[0x060];
  elsif EL2Enabled() & HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  X[t, 64] = CNTVOFF_EL2;
elsif PSTATE.EL == EL3 then
  X[t, 64] = CNTVOFF_EL2;

MSR CNTVOFF_EL2, \( <X_t> \)

\[
\begin{array}{c|c|c|c|c}
\hline
\text{op0} & \text{op1} & \text{CRn} & \text{CRm} & \text{op2} \\
\hline
0b11 & 0b100 & 0b1110 & 0b0000 & 0b011 \\
\hline
\end{array}
\]

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & HCR_EL2.<NV2,NV> == '11' then
    NVMem[0x060] = X[t, 64];
  elsif EL2Enabled() & HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  CNTVOFF_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
  CNTVOFF_EL2 = X[t, 64];
Part E
The AArch32 Application Level Architecture
Chapter E1
The AArch32 Application Level Programmers’ Model

This chapter gives an Application level description of the programmers’ model for software executing in AArch32 state. This means it describes execution in EL0 when EL0 is using AArch32. It contains the following sections:

- About the Application level programmers’ model on page E1-7128.
- The Application level programmers’ model in AArch32 state on page E1-7129.
- Advanced SIMD and floating-point instructions on page E1-7140.
- About the AArch32 System register interface on page E1-7158.
- Exceptions on page E1-7159.
E1.1 About the Application level programmers’ model

This chapter contains the programmers’ model information required for the development of applications that will execute in AArch32 state.

The information in this chapter is distinct from the system information required to service and support application execution under an operating system, or higher level of system software. However, some knowledge of that system information is needed to put the Application level programmers’ model into context.

Depending on the implementation, the architecture supports multiple levels of execution privilege. These privilege levels are indicated by different Exception levels that number upwards from EL0, where EL0 corresponds to the lowest privilege level and is often described as unprivileged. The Application level programmers’ model is the programmers’ model for software executing at EL0. For more information, see Arm architectural concepts on page A1-39.

System software determines the Exception level, and therefore the level of privilege, at which application software runs. When an operating system supports execution at both EL1 and EL0, an application usually runs unprivileged. This has the following effects:

• It means that the operating system can allocate system resources to an application in a unique or shared manner.

• It provides a degree of protection from other processes, and so helps protect the operating system from malfunctioning software.

This chapter indicates where some System level understanding is helpful, and if appropriate it gives a reference to the System level description.

Application level software is generally unaware of its Security state, and of any virtualization. For more information, see The Armv8-A security model on page G1-8905 and The effect of implementing EL2 on the Exception model on page G1-8910.

--- Note ---

• When an implementation includes EL3, application and operating system software normally executes in Non-secure state.

• Older documentation, describing implementations or architecture versions that support only two privilege levels, often refers to execution at EL1 as privileged execution.

• In this manual, the terms CONSTRAINED UNPREDICTABLE, IMPLEMENTATION DEFINED, OPTIONAL, RES0, RES1, UNDEFINED, UNKNOWN, and UNPREDICTABLE have Arm-specific meanings, as defined in the Glossary. In body text, these terms are shown in SMALL CAPS, for example IMPLEMENTATION DEFINED.
The AArch32 Application Level Programmers’ Model

E1.2 The Application level programmers’ model in AArch32 state

The following sections give more information about the application level programmers’ model in AArch32 state:

- Instruction sets, arithmetic operations, and register files.
- Core data types and arithmetic in AArch32 state.
- The general-purpose registers, and the PC, in AArch32 state on page E1-7131.
- Process state, PSTATE on page E1-7133.
- Jazelle support on page E1-7139.

E1.2.1 Instruction sets, arithmetic operations, and register files

The A32 and T32 instruction sets both provide a wide range of integer arithmetic and logical operations, that operate on a register file of sixteen 32-bit registers, that are comprised of the AArch32 general-purpose registers and the PC. As described in The general-purpose registers, and the PC, in AArch32 state on page E1-7131, these registers include the registers SP (R13) and LR (R14), which have specialized uses. Core data types and arithmetic in AArch32 state gives more information about these operations.

In addition, an implementation that implements the T32 and A32 instruction sets includes both:

- Scalar floating-point instructions.
- The Advanced SIMD vector instructions.

Floating-point and vector instructions operate on a separate common register file, described in The SIMD and floating-point register file on page E1-7140. Advanced SIMD and floating-point instructions on page E1-7140 gives more information about these instructions.

E1.2.2 Core data types and arithmetic in AArch32 state

When executing in AArch32 state, a PE supports the following data types in memory:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>8 bits.</td>
</tr>
<tr>
<td>Halfword</td>
<td>16 bits.</td>
</tr>
<tr>
<td>Word</td>
<td>32 bits.</td>
</tr>
<tr>
<td>Doubleword</td>
<td>64 bits.</td>
</tr>
</tbody>
</table>

PE registers are 32 bits in size. The instruction sets provide instructions that use the following data types for data held in registers:

- 32-bit pointers.
- Unsigned or signed 32-bit integers.
- Unsigned 16-bit or 8-bit integers, held in zero-extended form.
- Signed 16-bit or 8-bit integers, held in sign-extended form.
- Two 16-bit integers packed into a register.
- Four 8-bit integers packed into a register.
- Unsigned or signed 64-bit integers held in two registers.

Load and store operations can transfer bytes, halfwords, or words to and from memory. Loads of bytes or halfwords zero-extend or sign-extend the data as it is loaded, as specified in the appropriate load instruction.

The instruction sets include load and store operations that transfer two or more words to and from memory. Software can load and store doublewords using these instructions.

Note

For information about the atomicity of memory accesses, see Atomicity in the Arm architecture on page E2-7164.

When any of the data types is described as unsigned, the N-bit data value represents a non-negative integer in the range 0 to \(2^{N-1}\), using normal binary format.

When any of these types is described as signed, the N-bit data value represents an integer in the range \(-2^{(N-1)}\) to \(+2^{(N-1)}-1\), using two's complement format.
The instructions that operate on packed halfwords or bytes include some multiply instructions that use only one of two halfwords, and SIMD instructions that perform parallel addition or subtraction on all of the halfwords or bytes.

--- Note ---
These SIMD instructions operate on values held in the general-purpose registers, and must not be confused with the Advanced SIMD instructions that operate on a separate register file that provides registers of up to 128 bits.

Direct instruction support for 64-bit integers is limited, and most 64-bit operations require sequences of two or more instructions to synthesize them.

### Integer arithmetic

The instruction set provides a wide range of operations on the values in registers, including bitwise logical operations, shifts, additions, subtractions, multiplications, and divisions. The pseudocode described in Appendix K16 Arm Pseudocode Definition defines these operations, usually in one of three ways:

- By direct use of the pseudocode operators and built-in functions defined in Operators on page K16-11816.
- By use of pseudocode helper functions defined in the main text.
- By a sequence of the form:
  1. Use of the $\text{Int}()$, $\text{UInt}()$, and $\text{Int}()$ built-in functions defined in Converting bitstrings to integers on page K16-11828 to convert the bitstring contents of the instruction operands to the unbounded integers that they represent as two's complement or unsigned integers.
  2. Use of mathematical operators, built-in functions and helper functions on those unbounded integers to calculate other such integers.
  3. Use of either the bitstring extraction operator defined in Bitstring concatenation and slicing on page K16-11817 or of the saturation helper functions described in Pseudocode description of saturation on page E1-7131 to convert an unbounded integer result into a bitstring result that can be written to a register.

### Shift and rotate operations

The following types of shift and rotate operations are used in instructions:

#### Logical Shift Left

The $\text{LSL()}$ pseudocode function moves each bit of a bitstring left by a specified number of bits. Zeros are shifted in at the right end of the bitstring. Bits that are shifted off the left end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

#### Logical Shift Right

The $\text{LSR()}$ pseudocode function moves each bit of a bitstring right by a specified number of bits. Zeros are shifted in at the left end of the bitstring. Bits that are shifted off the right end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

#### Arithmetic Shift Right

The $\text{ASR()}$ pseudocode function moves each bit of a bitstring right by a specified number of bits. Copies of the leftmost bit are shifted in at the left end of the bitstring. Bits that are shifted off the right end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

#### Rotate Right

The $\text{ROR()}$ pseudocode function moves each bit of a bitstring right by a specified number of bits. Each bit that is shifted off the right end of the bitstring is re-introduced at the left end. The last bit shifted off the right end of the bitstring can be produced as a carry output.

#### Rotate Right with Extend

The $\text{RRX()}$ pseudocode function moves each bit of a bitstring right by one bit. A carry input is shifted in at the left end of the bitstring. The bit shifted off the right end of the bitstring can be produced as a carry output.
Pseudocode description of addition and subtraction

In pseudocode, addition and subtraction can be performed on any combination of unbounded integers and bitstrings, provided that if they are performed on two bitstrings, the bitstrings must be identical in length. The result is another unbounded integer if both operands are unbounded integers, and a bitstring of the same length as the bitstring operand or operands otherwise. For the definition of these operations, see Addition and subtraction on page K16-11818.

The main addition and subtraction instructions can produce status information about both unsigned carry and signed overflow conditions. When necessary, multi-word additions and subtractions can be synthesized from this status information. In pseudocode, the AddWithCarry() function provides an addition with a carry input and a set of output Condition flags including carry output and overflow:

An important property of the AddWithCarry() function is that if:

\[(\text{result}, \text{nzcv}) = \text{AddWithCarry}(x, \text{NOT}(y), \text{carry\_in})\]

Then:

- If \(\text{carry\_in} == '1'\), then \(\text{result} == x-y\) with:
  - \(\text{nzcv}<0> == '1'\) if signed overflow occurred during the subtraction.
  - \(\text{nzcv}<1> == '1'\) if unsigned borrow did not occur during the subtraction, that is, if \(x>y\).
- If \(\text{carry\_in} == '0'\), then \(\text{result} == x-y-1\) with:
  - \(\text{nzcv}<0> == '1'\) if signed overflow occurred during the subtraction.
  - \(\text{nzcv}<1> == '1'\) if unsigned borrow did not occur during the subtraction, that is, if \(x>y\).

Taken together, this means that the \(\text{carry\_in}\) and \(\text{nzcv}<1>\) output in AddWithCarry() calls can act as NOT borrow flags for subtractions as well as carry flags for additions.

Pseudocode description of saturation

Some instructions perform saturating arithmetic, that is, if the result of the arithmetic overflows the destination signed or unsigned N-bit integer range, the result produced is the largest or smallest value in that range, rather than wrapping around modulo \(2^N\). This is supported in pseudocode by:

- The SignedSatQ() and UnsignedSatQ() functions when an operation requires, in addition to the saturated result, a Boolean argument that indicates whether saturation occurred.
- The SignedSat() and UnsignedSat() functions when only the saturated result is required.

\(\text{SatQ}(i, N, \text{unsigned})\) returns either \(\text{UnsignedSatQ}(i, N)\) or \(\text{SignedSatQ}(i, N)\) depending on the value of its third argument, and \(\text{Sat}(i, N, \text{unsigned})\) returns either \(\text{UnsignedSat}(i, N)\) or \(\text{SignedSat}(i, N)\) depending on the value of its third argument.

E1.2.3 The general-purpose registers, and the PC, in AArch32 state

In the AArch32 Application level view, a PE has:

- Fifteen general-purpose 32-bit registers, R0 to R14, of which R13 and R14 have alternative names reflecting how they are, or can be, used:
  - R13 is usually identified as SP.
  - R14 is usually identified as LR.
- The PC (program counter), which can be described as R15.

The specialized uses of the SP (R13), LR (R14), and PC (R15) are:

SP, the stack pointer

The PE uses SP as a pointer to the active stack.

In the T32 instruction set, some instructions cannot access SP. Instructions that can access SP can use SP as a general-purpose register.
The A32 instruction set provides more general access to SP, and it can be used as a general-purpose register.

--- Note ---

Using SP for any purpose other than as a stack pointer might break the requirements of operating systems, debuggers, and other software systems, causing them to malfunction.

---

Software can refer to SP as R13.

LR, the link register

The link register can be used to hold return link information, and some cases described in this manual require this use of the LR. When software does not require the LR for linking, it can use it for other purposes. Software can refer to LR as R14.

PC, the program counter

- When executing an A32 instruction, PC reads as the address of the current instruction plus 8.
- When executing a T32 instruction, PC reads as the address of the current instruction plus 4.
- Writing an address to PC causes a branch to that address.

Most T32 instructions cannot access PC.

The A32 instruction set provides more general access to the PC, and many A32 instructions can use the PC as a general-purpose register. However, Arm deprecates the use of PC for any purpose other than as the program counter. See Writing to the PC for more information.

Software can refer to PC as R15.

See AArch32 general-purpose registers, the PC, and the Special-purpose registers on page G1-8917 for the system level view of these registers.

--- Note ---

In general, Arm strongly recommends using the names SP, LR, and PC instead of R13, R14, and R15. However, sometimes it is simpler to use the R13-R15 names when referring to a group of registers. For example, it is simpler to refer to registers R8 to R15, rather than to registers R8 to R12, the SP, LR, and PC. These two descriptions of the group of registers have exactly the same meaning.

---

Writing to the PC

In the A32 and T32 instruction sets, many data-processing instructions can write to the PC. Writes to the PC are handled as follows:

- In T32 state, the following 16-bit T32 instruction encodings branch to the value written to the PC:
  - Encoding T2 of ADD, ADDS (register) on page F5-7456.
  - Encoding T1 of MOV, MOVS (register) on page F5-7719.
  The value written to the PC is forced to be halfword-aligned by ignoring its least significant bit, treating that bit as being 0.

- The B, BL, CBNZ, CBZ, CHKA, HB, HBL, HBLP, HBP, TBB, and TBH instructions remain in the same instruction set state and branch to the value written to the PC.
  The definition of each of these instructions ensures that the value written to the PC is correctly aligned for the current instruction set state.

- The BLX (immediate) instruction switches between A32 and T32 states and branches to the value written to the PC. Its definition ensures that the value written to the PC is correctly aligned for the new instruction set state.

- The following instructions write a value to the PC, treating that value as an interworking address to branch to, with low-order bits that determine the new instruction set state:
  - BLX (register), BX, and BXJ.
— LDR instructions with <Rt> equal to the PC.
— POP and all forms of LDM except LDM (exception return), when the register list includes the PC.
— In A32 state only, ADC, ADD, ADR, AND, ASR (immediate), BIC, EOR, LSL (immediate), LSR (immediate), MOV, MVN, ORR, ROR (immediate), RRS, RSB, RSC, SBC, and SUB instructions with <Rd> equal to the PC and without flag-setting specified.

For details of how an interworking address specifies the new instruction set state and instruction address, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC.

——— Note ————

The register-shifted register instructions, which are available only in the A32 instruction set and are summarized in Data-processing register (register shift) on page F4-7386, are CONstrained UNPREDICTABLE if they attempt to write to the PC, see Using R15 by instruction on page K1-11563.

——— Note ————

• Some instructions are treated as exception return instructions, and write both the PC and the CPSR. For more information, including which instructions are exception return instructions, see Exception return to an Exception level using AArch32 on page G1-8951.
• Some instructions cause an exception, and the exception handler address is written to the PC as part of the exception entry.

Pseudocode description of operations on the AArch32 general-purpose registers and the PC

In pseudocode, the uses of the R[n] function, with an index parameter n, are:
• Reading or writing R0-R12, SP, and LR, using n = 0-12, 13, and 14 respectively.
• Reading the PC, using n = 15.

Pseudocode description of general-purpose register and PC operations on page G1-8919 describes accesses to these registers.

Descriptions of A32 store instructions that store the PC value use the PCStoreValue() pseudocode function to specify the PC value stored by the instruction.

Writing an address to the PC causes either a simple branch to that address or an interworking branch that also selects the instruction set to execute after the branch. A simple branch is performed by the BranchWritePC() function.

An interworking branch is performed by the BXWritePC() function.

The LoadWritePC() and ALUWritePC() functions are used for two cases where the behavior was systematically modified between architecture versions.

E1.2.4 Process state, PSTATE

Process state or PSTATE is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

——— Note ————

In this chapter, references to PSTATE link to the more appropriate of:
• The Application-level view of PSTATE given in this section.
• The System-level description in Process state, PSTATE on page G1-8921.
The following PSTATE information is accessible at EL0:

**The Condition flags**

Flag-setting instructions set these. They are:

- **N** Negative Condition flag. If the result of the instruction is regarded as a two's complement signed integer, the PE sets this to:
  - 1 if the result is negative.
  - 0 if the result is positive or zero.

- **Z** Zero Condition flag. Set to:
  - 1 if the result of the instruction is zero.
  - 0 otherwise.
  
  A result of zero often indicates an equal result from a comparison.

- **C** Carry Condition flag. Set to:
  - 1 if the instruction results in a carry condition, for example an unsigned overflow that is the result of an addition.
  - 0 otherwise.

- **V** Overflow Condition flag. Set to:
  - 1 if the instruction results in an overflow condition, for example a signed overflow that is the result of an addition.
  - 0 otherwise.

Conditional instructions test the N, Z, C, and V Condition flags, combining them with the *Condition code* for the instruction, to determine whether the instruction must be executed. In this way, execution of the instruction is conditional on the result of a previous operation. For more information about conditional execution, see *Conditional execution on page F1-7229*.

**The overflow or saturation flag**

- **Q** Some instructions can set this. For those instructions that can, the PE:
  - Sets it to 1 if the instruction indicates overflow or saturation.
  - Leaves it unchanged otherwise.

  For more information, see *Pseudocode description of saturation on page E1-7131*.

**The greater than or equal flags**

- **GE[3:0]** The instructions described in *Parallel addition and subtraction instructions on page F2-7266* update these to indicate the results from individual bytes or halfwords of the operation. These flags can control a later SEL instruction. For more information, see *SEL on page F5-7880*.

PSTATE also contains PE state controls. There is no direct access to these from application level instructions, but they can be changed by side-effects of application level instructions. They are:

**Instruction set state**

- **J, T** The current instruction set state, as shown in Table E1-1. The J bit is RES0, see the Note in this section.

<table>
<thead>
<tr>
<th>J</th>
<th>T</th>
<th>Instruction set state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>A32</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>T32</td>
</tr>
</tbody>
</table>

A32 The PE is executing the A32 instruction set, summarized in *Chapter F4.A32 Instruction Set Encoding*. 
The PE is executing the T32 instruction set, summarized in Chapter F3 T32 Instruction Set Encoding.

--- Note ---

Encoding with J==1 before Armv8, Jazelle, and T32EE states

In previous versions of the Arm architecture, the encoding \{1, 0\} selected Jazelle state, and encoding \{1, 1\} selected T32EE state. From the introduction of Armv8, the architecture does not support either of these states, and these are encodings for unimplemented instruction set states, see Unimplemented instruction sets on page G1-8927. AArch32 state requires a Trivial Jazelle implementation, see Trivial implementation of the Jazelle extension on page G1-8927.

The IT block state

\[ \text{IT}[7:0] \]  

The If-Then controls for the T32 IT instruction, which applies to the IT block of instructions that immediately follow the IT instruction. See IT on page F5-7580 for a description of the IT instruction and its associated IT block.

For more information about the use of \text{PSTATE.IT}, see Use of \text{PSTATE.IT} on page E1-7137.

Endianness mapping

\[ E \]  

For data accesses, controls the endianness:

- 0 Little-endian.
- 1 Big-endian.

If an implementation does not provide:

- Big-endian support for data accesses, this bit is \text{RES0}.
- Little-endian support for data accesses, this bit is \text{RES1}.

Instruction fetches are always little-endian, and ignore \text{PSTATE.E}.

Timing control bits

\[ \text{DIT} \]  

Data Independent Timing (DIT) bit. For more information, see About the DIT bit on page E1-7139.

This bit is implemented only when \text{FEAT_DIT} is implemented.

On a reset to AArch32 state, this bit is set to 0.

Accessing PSTATE fields at EL0

The following sections describe which PSTATE fields can be directly accessed at EL0, and how they can be accessed:

- The Application Program Status Register, APSR.
- The SETEND instruction on page E1-7136.

The Application Program Status Register, APSR

At EL0, some PSTATE fields can be accessed using the Special-purpose Application Program Status Register (APSR). The APSR can be directly read using the MRS instruction, and directly written using the MSR (register) and MSR (immediate) instructions.
The APSR bit assignments are:

The APSR bit assignments are:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>24</th>
<th>23</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Z</td>
<td>C</td>
<td>V</td>
<td>Q</td>
<td>RES0</td>
<td>RES0</td>
<td>GE[3:0]</td>
<td>RES0</td>
<td>RES0</td>
<td>RES1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **N, Z, C, V, bits [31:28]**
  - The PSTATE Condition flags.

- **Q, bit [27]**
  - The PSTATE overflow or saturation flag.

- **Bits[26:24]**
  - Reserved, RES0. Software can use MSR instructions that write the top byte of the APSR without using a read-modify-write sequence. If it does this, it must write zeros to bits[26:24].

- **Bits[23:20, 15:0]**
  - Reserved bits that are allocated to system features, or are available for future expansion.

Unprivileged execution ignores writes to fields that are accessible only at EL1 or higher. However, application level software that writes to the APSR must treat reserved bits as **Do-Not-Modify** (DNM) bits. For more information about the reserved bits, see *The Current Program Status Register, CPSR on page G1-8923*.

These bits are **UNKNOWN** on a Read, and it is permitted that, on a read of APSR:

- Bit[22] returns the value of PSTATE.PAN.
- Bit[9] returns the value of PSTATE.E.
- Bits[4:0] return the value of PSTATE.M[4:0]. Bit[4] is RES1 indicating that the PE is in AArch32 state.

--- **Note**

This is an exception to the general rule that an **UNKNOWN** field must not return information that cannot be obtained, at the current Privilege level, by an architected mechanism.

- **GE[3:0], bits [19:16]**
  - The PSTATE greater than or equal flags.

The other PSTATE fields cannot be accessed by using the APSR.

The system level alias for the APSR is the CPSR. The CPSR is a superset of the APSR. See *The Current Program Status Register, CPSR on page G1-8923*.

Writes to the PSTATE fields have side-effects on various aspects of PE operation. All of these side-effects, except side-effects on memory accesses associated with fetching instructions, are synchronous to the APSR write. This means they are guaranteed:

- Not to be visible to earlier instructions in the execution stream.
- To be visible to later instructions in the execution stream.

**The SETEND instruction**

The A32 and T32 instruction sets both include an instruction to manipulate PSTATE.E:

- **SETEND BE** Sets PSTATE.E to 1, for big-endian operation.
- **SETEND LE** Sets PSTATE.E to 0, for little-endian operation.

The SETEND instruction is unconditional. For more information, see *SETEND on page F5-7882*. Arm deprecates use of the SETEND instruction.
Use of PSTATE.IT

PSTATE.IT provides the If-Then controls for the T32 IT instruction, which applies to the IT block of instructions that immediately follow the IT instruction.

PSTATE.IT divides into two subfields:

- IT[7:5]  
  Holds the base condition for the current IT block. The base condition is the top three bits of the Condition code specified by the <firstcond> field of the IT instruction.

- IT[4:0]  
  Encodes:
  - Implicitly, the size of the IT block. This is the number of instructions that are to be conditionally executed. The size of the block is indicated by the position of the least significant 1 in this field, as shown in Table E1-2.
  - For each instruction in the IT block, the least significant bit of the Condition code. This is encoded in the IT block entries that Table E1-2 shows as Nx.

  Note
  Changing the least significant bit of a Condition code from 0 to 1 has the effect of inverting the Condition code.

Both subfields are all zeros when no IT block is active.

When an IT instruction is executed, PSTATE.IT is set according to the <firstcond> field of the instruction and the Then and Else (T and E) parameters in the instruction, see IT on page F5-7580. This means that, on executing an IT instruction, the initial state of PSTATE.IT depends on the number of instructions in the IT block, as Table E1-2 shows:

<table>
<thead>
<tr>
<th>Number of instructions in IT block</th>
<th>PSTATE.IT bits</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>cond_base</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>cond_base</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>cond_base</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>cond_base</td>
<td></td>
</tr>
<tr>
<td>Not executing an IT instruction</td>
<td>000</td>
<td>No IT block is active</td>
</tr>
</tbody>
</table>

In Table E1-2, N1 refers to the first instruction in the IT block, and N2, N3, and N4 refer to the second, third, and fourth instructions in the IT block if they are present.

When permitted, an instruction in an IT block is conditional, see Conditional instructions on page F2-7257 and Conditional execution on page F1-7229. The Condition code used is the current value of IT[7:4]. When an instruction in an IT block completes its execution normally, PSTATE.IT[4:0] is left-shifted by one bit, so that PSTATE[4] always relates to the next instruction to be executed.
Table E1-3 shows how PSTATE.IT during the execution of an IT instruction with four instructions in the IT block.

Table E1-3 Updates to PSTATE.IT when executing an IT instruction with a four-instruction IT block

<table>
<thead>
<tr>
<th>IT block instruction being executed</th>
<th>PSTATE.IT bits</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>[7:5] cond_base [4] N1 [3] N2 [2] N3 [1] N4 [0] 1</td>
<td>-</td>
</tr>
<tr>
<td>Second</td>
<td>[7:5] cond_base [4] N2 [3] N3 [2] N4 [1] 1 0</td>
<td>-</td>
</tr>
<tr>
<td>Third</td>
<td>[7:5] cond_base [4] N3 [3] N4 [2] 1 0 0</td>
<td>-</td>
</tr>
<tr>
<td>Fourth</td>
<td>[7:5] cond_base [4] N4 [3] 1 0 0 0</td>
<td>-</td>
</tr>
<tr>
<td>Not executing an IT instruction</td>
<td>[7:5] 0 0 [4] 0 [3] 0 [2] 0</td>
<td>No IT block is active</td>
</tr>
</tbody>
</table>

A few instructions, for example BKPT, cannot be conditional and therefore are always executed ignoring the current value of PSTATE.IT.

For details of what happens if an instruction in an IT block takes an exception, see Overview of exception entry on page G1-8936.

An instruction that might complete its normal execution by branching is only permitted in an IT block as the last instruction in the block. This means that normal execution of the instruction always results in PSTATE.IT advancing to execution where no IT block is active.

Implementations can provide a set of ITD control fields, SCTLR.ITD, SCTLR_EL1.ITD, and HSCTLR.ITD, to disable use of IT for some instructions, making them UNDEFINED. When an implementation includes ITD control fields, Changes to an ITD control by an instruction in an IT block describes the permitted CONSTRAINED UNPREDICTABLE behaviors if an instruction in an IT block changes the value of an ITD control to disable the use of the IT instruction.

On a branch or an exception return, if PSTATE.IT is set to a value that is not consistent with the instruction stream being branched to or returned to, then instruction execution is CONSTRAINED UNPREDICTABLE.

PSTATE.IT affects instruction execution only in T32 state. In A32 state, PSTATE.IT must be 0b00000000, otherwise the behavior is CONSTRAINED UNPREDICTABLE.

For more information, see CONSTRAINED UNPREDICTABLE behavior associated with IT instructions and PSTATE.IT on page K1-11564.

Changes to an ITD control by an instruction in an IT block

In an implementation that includes SCTLR.ITD, SCTLR_EL1.ITD, and HSCTLR.ITD controls, if an instruction in an IT block changes an ITD control so that the IT instruction using the IT block would be disabled, then one of the following behaviors applies:

- The change to the ITD field, once synchronized, has no effect on the execution of instructions in the current IT block, but applies only to any subsequent execution of an IT instruction to which the control applies.
- Synchronizing the change to the ITD field guarantees that all bits of PSTATE.IT are cleared to 0.

In addition, after the change to the ITD field has been synchronized, any remaining instructions in the IT block that would be made UNDEFINED by the new value of ITD are either:

- Executed normally.
- Treated as UNDEFINED.

The choice between the options described in this section is determined by the implementation, and any choice can vary between different changes to an ITD control by an instruction in an IT block.
Pseudocode description of PSTATE PE state fields

The pseudocode function `CurrentInstrSet()` returns the current instruction set. The pseudocode function `SelectInstrSet()` selects a new instruction set.

PSTATE.IT advances after normal execution of an IT block instruction. This is described by the `AArch32.ITAdvance()` pseudocode function.

The pseudocode function `InITBlock()` tests whether the current instruction is in an IT block. The pseudocode function `LastInITBlock()` tests whether the current instruction is the last instruction in an IT block.

The `BigEndian()` pseudocode function tests whether big-endian data memory accesses are currently selected.

### E1.2.5 About the DIT bit

When the value of CPSR.DIT is 1:

- The instructions listed in CPSR are required to have:
  - Timing which is independent of the values of the data supplied in any of its registers, and the values of the NZCV flags.
  - Responses to asynchronous exceptions which do not vary based on the values supplied in any of their registers, or the values of the NZCV flags.
  - All loads and stores have their timing insensitive to the value of the data being loaded or stored.

/player/cpsr

Note

When the value of CPSR.DIT is 0, the architecture makes no statement about the timing properties of any instructions.

A corresponding DIT bit is added to PSTATE in AArch64 state, and to CPSR in AArch32 state.

When an exception is taken from AArch32 state to AArch32 state, CPSR.DIT is copied to SPSR.DIT.

When an exception is taken from AArch32 state to AArch64 state, CPSR.DIT is copied to SPSR_ELx.DIT.

When an exception returns to AArch32 state from AArch32 state, SPSR.DIT is copied to CPSR.DIT.

When an exception returns to AArch32 state from AArch64 state, SPSR_ELx.DIT is copied to CPSR.DIT.

CPSR.DIT bit can be written using an MSR instruction at any Exception Level in AArch32 state, and read using an MRS instruction at any Exception Level.

### E1.2.6 Jazelle support

The architecture requires AArch32 state to include a trivial implementation of the Jazelle extension, as described in `Trivial implementation of the Jazelle extension` on page G1-8927.
E1.3 Advanced SIMD and floating-point instructions

In general, the architecture requires implementation of Advanced SIMD and floating-point instructions in the T32 and A32 instruction sets, but see Implications of not including Advanced SIMD and floating-point support on page E1-7153.

The Advanced SIMD instructions perform packed Single Instruction Multiple Data (SIMD) operations, either integer or single-precision floating-point. The floating-point instructions perform single-precision or double-precision scalar floating-point operations. When FEAT_FP16 is implemented, half-precision floating-point can also be used for data processing.

These instructions permit floating-point exceptions, such as Overflow or Divide by Zero, to be handled without trapping. When handled in this way, a floating-point exception causes a cumulative status register bit to be set to 1 and a default result to be produced by the operation. The architecture also optionally supports the trapping of floating-point exceptions. For more information about floating-point exceptions, see Floating-point exceptions and exception traps on page E1-7148.

The Advanced SIMD and floating-point instructions also provide the following conversion functions:

- Between half-precision floating-point and single-precision floating point, in both directions.
- From double-precision, floating-point to single-precision floating point or integer.
- When FEAT_AA32BF16 is implemented, between single-precision floating-point and BFloat16 floating-point.

Some Advanced SIMD instructions support polynomial arithmetic over \{0, 1\}, as described in Polynomial arithmetic over \{0, 1\} on page A1-54.

For system level information about the Advanced SIMD and Floating-point implementation, see Advanced SIMD and floating-point support on page G1-8996.

The following sections give more information about the Advanced SIMD and floating-point instructions:

- The SIMD and floating-point register file.
- Data types supported by the Advanced SIMD implementation on page E1-7142.
- Advanced SIMD and floating-point System registers on page E1-7142.
- Floating-point data types and arithmetic on page E1-7142.
- Flushing denormalized numbers to zero on page E1-7143.
- Floating-point exceptions and exception traps on page E1-7148.
- Controls of Advanced SIMD operation that do not apply to floating-point operation on page E1-7153.
- Implications of not including Advanced SIMD and floating-point support on page E1-7153.
- Pseudocode description of floating-point operations on page E1-7153.

E1.3.1 The SIMD and floating-point register file

The Advanced SIMD and floating-point instructions use the same register file, which comprises 32 registers. This is distinct from the register file that holds the general-purpose registers and the PC.

The Advanced SIMD and floating-point views of the register file are different. The following sections describe these different views. Figure E1-1 on page E1-7141 shows the views of the register file, and the way the word, doubleword, and quadword registers overlap.

Advanced SIMD views of the register file

Advanced SIMD can view this register file as:

- Sixteen 128-bit quadword registers, Q0-Q15.
- Thirty-two 64-bit doubleword registers, D0-D31.

These views can be used simultaneously. For example, a program might hold 64-bit vectors in Q0 and Q1 and a 128-bit vector in Q1.
Floating-point views of the register file

The Advanced SIMD and floating-point register file consists of thirty-two doubleword registers, which can be viewed as:

- Thirty-two 64-bit doubleword registers, D0-D31. This view is also available to Advanced SIMD instructions.
- Thirty-two 32-bit single word registers, S0-S31. Only half of the set is accessible in this view.

**Note**

In AArch32 state, half-precision floating point values are always represented using the bottom 16 bits of a single word register, S0-S31. When a half-precision value is written to a single word register, the top 16 bits of that register are set to 0.

The two views can be used simultaneously.

SIMD and Floating-point register file mapping onto registers

Figure E1-1 shows the different views of the SIMD and floating-point register file, and the relationship between them.

![Figure E1-1 SIMD and floating-point register file, AArch32 operation](image-url)

The mapping between the registers is as follows:

- S<2n> maps to the least significant half of D<n>.
- S<2n+1> maps to the most significant half of D<n>.
- D<2n> maps to the least significant half of Q<n>.
- D<2n+1> maps to the most significant half of Q<n>.

For example, software can access the least significant half of the elements of a vector in Q6 by referring to D12, and the most significant half of the elements by referring to D13.
Pseudocode description of the SIMD and Floating-point register file

The functions \_Dclone, S[], and D[] provide the S0-S31, D0-D31, and Q0-Q15 views of the Advanced SIMD and floating-point registers:

The \texttt{Din[]} function returns a doubleword register from the \_Dclone copy of the SIMD and Floating-point register file, and the Qin[] function returns a quadword register from that register file.

--- Note ---

The \texttt{CheckAdvSIMDEnabled()} function copies the D[] register file to \_Dclone[], see \textit{Pseudocode description of enabling SIMD and floating-point functionality on page G1-9035}.

---

E1.3.2 Data types supported by the Advanced SIMD implementation

Advanced SIMD instructions can operate on integer and floating-point data, and the implementation defines a set of data types that support the required data formats. \textit{Advanced SIMD vector formats in AArch32 state on page A1-44} describes these formats.

Advanced SIMD vectors

In an implementation that includes support for Advanced SIMD operation, a register can hold one or more packed elements, all of the same size and type. The combination of a register and a data type describes a vector of elements. The vector is considered to be an array of elements of the data type specified in the instruction. The number of elements in the vector is implied by the size of the data elements and the size of the register.

Vector indices are in the range 0 to (number of elements – 1). An index of 0 refers to the least significant end of the vector. In \textit{Advanced SIMD vector formats in AArch32 state on page A1-44, Figure A1-3 on page A1-46 shows the Advanced SIMD vector formats}.

\textit{Pseudocode description of Advanced SIMD vectors}

The pseudocode function \texttt{Elem[]} accesses the element of a specified index and size in a vector.

---

E1.3.3 Advanced SIMD and floating-point System registers

The Advanced SIMD and floating-point instructions have a shared register space for System registers. The only register in this space that is accessible at the Application level is the FPSCR.

Writes to the FPSCR can have side-effects on various aspects of PE operation. All of these side-effects are synchronous to the FPSCR write. This means they are guaranteed not to be visible to earlier instructions in the execution stream, and they are guaranteed to be visible to later instructions in the execution stream.

See \textit{Advanced SIMD and floating-point System registers on page G1-8998} for the system level view of the registers.

These registers can be described as the \textit{SIMD and floating-point System registers}.

---

E1.3.4 Floating-point data types and arithmetic

The T32 and A32 floating-point instructions support single-precision (32-bit) and double-precision (64-bit) data types and arithmetic as defined by the IEEE 754 floating-point standard. They also support the half-precision (16-bit) floating-point data type for data storage, by supporting conversions between single-precision and half-precision data types. When \texttt{FEAT\_FP16} is implemented, it also supports the half-precision floating-point data type for data processing operations. When \texttt{FEAT\_AA32BF16} is implemented, it also supports the BFloat16 floating-point storage format.

\textit{Arm standard floating-point arithmetic} means IEEE 754 floating-point arithmetic with the restrictions described in \textit{Floating-point support on page A1-55}, including supporting only the input and output values described in \textit{Arm standard floating-point input and output values on page A1-57}.

The AArch32 Advanced SIMD instructions support single-precision and, when \texttt{FEAT\_FP16} is implemented, half-precision Arm standard floating-point arithmetic.
The following sections describe the Advanced SIMD and floating-point formats:

- **Double-precision floating-point format** on page A1-51.
- **BFloat16 floating-point format** on page A1-52.

The following sections describe features of Advanced SIMD and floating-point processing:

- **Flushing denormalized numbers to zero.**
- **NaN handling and the Default NaN** on page A1-60.

### E1.3.5 Flushing denormalized numbers to zero

Calculations involving denormalized numbers and Underflow exceptions can reduce the performance of floating-point processing. For many algorithms, replacing the denormalized operands and Intermediate results with zeros can recover this performance, without significantly affecting the accuracy of the final result. Arm floating-point implementations allow denormalized numbers to be flushed to zero to permit this optimization. If a number satisfies the condition \(0 < \text{Abs}(<result>) < \text{MinNorm}\), it is treated as a denormalized number.

\(\text{MinNorm}\) is defined as follows:

- For half-precision numbers, \(\text{MinNorm} = 2^{-14}\).
- For single-precision and BFloat16 numbers, \(\text{MinNorm} = 2^{-126}\).
- For double-precision numbers, \(\text{MinNorm} = 2^{-1022}\).

Flushing denormals to zero is incompatible with the IEEE 754 standard, and must not be used when IEEE 754 compatibility is a requirement. Enabling flushing of denormals to zero must be done with care. Although it can improve performance on some algorithms, there are significant limitations on its use. These are application-dependent:

- On many algorithms, it has no noticeable effect, because the algorithm does not usually process denormalized numbers.
- On other algorithms, it can cause exceptions to occur and can seriously reduce the accuracy of the results of the algorithm.

#### Flushing denormalized inputs to zero

If flushing denormalized inputs to zero is enabled for an instruction and a data type, and an input to that instruction is a denormalized number of that data type, the input operand is flushed to zero, and its sign bit is not changed. If a floating-point operation has an input denormalized number that is flushed to zero, for all purposes within the instruction other than calculating Input Denormal floating-point exceptions, all inputs that are denormalized numbers are treated as though they were zero with the same sign as the input.

For Advanced SIMD and floating-point instructions, other than \textit{FABS} and \textit{FNEG}, that process half-precision inputs, flushing denormalized inputs to zero can be controlled as follows:

- If \text{FPSCR.FZ16} is 0, denormalized half-precision inputs are not flushed to zero.
- If \text{FPSCR.FZ16} is 1, flushing denormalized inputs to zero occurs as follows:
  - If an instruction does not convert a half-precision input to a higher precision output, all input denormalized numbers are flushed to zero.
  - If an instruction converts a half-precision input to a higher precision output, input denormalized numbers are not flushed to zero.

For Advanced SIMD and scalar floating-point instructions, other than \textit{FABS} and \textit{FNEG}, that process single-precision, or double-precision inputs, flushing denormalized inputs to zero can be controlled as follows:

- If \text{FPSCR.FZ} is 0, flushing denormalized inputs to zero occurs as follows:
  - For Advanced SIMD floating-point instructions, all single-precision and double-precision inputs that are denormalized numbers are flushed to zero.
For scalar floating-point instructions, single-precision and double-precision inputs that are denormalized numbers are not flushed to zero.

- If FPSCR.FZ is 1, for all A32, and T32 instructions, single-precision, and double-precision inputs that are denormalized numbers are flushed to zero.

If FEAT_AA32BF16 is implemented, for Advanced SIMD and scalar floating-point instructions, other than FABS and FNNEG, that process BF16 inputs, flushing denormalized inputs to zero is treated as follows:

- Instructions that convert from single-precision floating-point values to BF16 format flush denormalized inputs to zero.
- For any value of FPSCR.FZ, VDOT (vector), VDOT (by element), and VMMLA instructions flush all BF16 inputs that are denormalized numbers to zero.

**Flushing to zero of denormalized numbers as Intermediate results of some BF16 instructions**

BF16 arithmetic instructions VDOT (by element), VDOT (vector), and VMMLA, convert BF16 input values to IEEE single-precision format, and calculate N-way dot-products, accumulating the products in single-precision accumulators.

If FEAT_AA32BF16 is implemented, for Advanced SIMD and floating-point instructions, if a BF16 arithmetic instruction processes an Intermediate result that is a single-precision denormalized number, the Intermediate result is unconditionally flushed to zero.

**Flushing denormalized outputs to zero**

If flushing denormalized outputs to zero is enabled for an instruction and a data type, and an output from that instruction is a denormalized number of that data type, the output operand is flushed to zero, and its sign bit is not changed.

If a floating-point operation has an output denormalized number that is flushed to zero, for all purposes within the instruction other than calculating floating-point exceptions, all outputs that are denormalized numbers are treated as though they were zero with the same sign as the output.

For Advanced SIMD and floating-point instructions, other than FABS and FNNEG, that generate half-precision outputs, flushing denormalized outputs to zero can be controlled as follows:

- If FPSCR.FZ16 is 0, denormalized half-precision outputs are not flushed to zero.
- If FPSCR.FZ16 is 1, flushing denormalized outputs to zero occurs as follows:
  - If the instruction does not convert a half-precision input to a higher precision output, all output denormalized numbers are flushed to zero.
  - If the instruction converts a half-precision input to a higher precision output, output denormalized numbers are not flushed to zero.

For Advanced SIMD and scalar floating-point instructions, other than FABS and FNNEG, that process single-precision, or double-precision inputs, flushing denormalized outputs to zero can be controlled as follows:

- If FPSCR.FZ is 0, flushing denormalized outputs to zero occurs as follows:
  - For Advanced SIMD floating-point instructions, all single-precision and double-precision outputs that are denormalized numbers are flushed to zero.
  - For scalar floating-point instructions, single-precision and double-precision outputs that are denormalized numbers are not flushed to zero.
- If FPSCR.FZ is 1, for all A32, and T32 instructions, single-precision, and double-precision outputs that are denormalized numbers are flushed to zero.

If FEAT_AA32BF16 is implemented, for Advanced SIMD and scalar floating-point instructions, other than FABS and FNNEG, that generate BF16 outputs, flushing denormalized outputs to zero can be controlled as follows:

- BF16 arithmetic instructions flush denormalized outputs to zero.
• If FPSCR.FZinstructions that convert from single-precision floating-point values to BF16 format flush
denormalized outputs to zero.
• VDOT (vector), VDOT (by element), and VMMLA instructions flush all BF16 outputs that are denormalized
numbers to zero regardless of the value of FPSCR.FZ.

E1.3.6 NaN handling and the Default NaN

The IEEE 754 standard defines a NaN as a number with all exponent bits set to 1 and a nonzero number in the
mantissa. The Arm architecture also defines a Default NaN which does not follow this format.

The IEEE 754 standard specifies that the sign bit of a NaN has no significance.

For a quiet NaN output derived from a signaling NaN operand, the most significant fraction bit is set to 1.

A PE is forbidden to generate a NaN whose value is strongly correlated to the values of non-NaN inputs as a
speculative result of a floating-point calculation not involving NaN inputs.

The Default NaN

The Default NaN is encoded as described in Table E1-4 on page E1-7145.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign bit</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Exponent</td>
<td>0x1F</td>
<td>0xFF</td>
<td>0x7FF</td>
<td>0xFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bits[21:0] == 0</td>
<td>bits[50:0] == 0</td>
<td>bits[5:0] == 0</td>
</tr>
</tbody>
</table>

If FPSCR.DN is 1, for Advanced SIMD and floating-point instructions other than FABS, FMAX*, FMIN* and FNEG, if any
input to a floating-point operation performed by the instruction is a NaN, the output of the floating-point operation
is the Default NaN.

For FABS, FNEG, FMAX*, and FMIN*, Default NaN behavior is explained in the instruction description.

If FPSCR.DN is 0, for floating-point processing the Default NaN is not used for NaN propagation.

If VDOT (vector), VDOT (by element), and VMMLA instructions generate a NaN, the NaN is the default NaN.
regardless of the setting of FPSCR.DN.

If a floating-point instruction performs a floating-point operation, and that instruction generates an untrapped
Invalid Operation floating-point exception for a reason other than one of the inputs being a signaling NaN, the
output is the Default NaN.

NaN handling

The IEE 754 standard does not specify which input NaN is used as the output NaN. Therefore, where the Arm
architecture specifies which input NaN to use, this is an addition to the requirements in the IEEE 754 standard.

Depending on the operation, the exact value of a derived quiet NaN output might have both a different sign and a
different number of fraction bits from its source. See instruction descriptions for details.

NaN propagation

If an output NaN is derived from one of the operands, how the input NaN propagates to the output depends on the
instruction and the number of operands.
If an output NaN is derived from an input NaN and if the size of the output format is the same as the input format, then all of the following apply:

• If the input NaN is a quiet NaN, the output NaN is the same as the input NaN.

• If the input NaN is a signaling NaN, the output NaN is derived as follows:
  — If the handling of a signaling NaN by the instruction generates an Invalid Operation exception, the output NaN is the quieted version of the input NaN.
  — If the handling of a signaling NaN by the instruction does not generate an Invalid Operation exception, the output NaN is the same as the input NaN. This case applies for FABS, FNEG, and FTSSEL instructions.

If an output NaN is derived from an input NaN and if the size of the output format is larger than the input format, all of the following apply:

• If the input NaN is a quiet NaN, the output NaN is the same as the input NaN except that the mantissa is zero-extended in the low-order bit to fit the output format, and the exponent field is set to all ones.

• If the input NaN is a signaling NaN, the output NaN is the quieted version of the input NaN, except that the mantissa is zero-extended in the low-order bits and the exponent field is set to all ones.

If an output NaN is derived from an input NaN and if the size of the output format is smaller than the input format, all of the following apply:

• If the input NaN is a quiet NaN, the output NaN is the same as the input NaN except that the mantissa is truncated in the lower-order bits to fit the output format, and the exponent field is set to all ones.

• If the input NaN is a signaling NaN, the output NaN is the quieted version of the input NaN except that the mantissa is truncated in the lower-order bits to fit the output format, and the exponent field is set to all ones.

For the following descriptions, when an operand is described as *first* this relates to the left-to-right ordering of the arguments of the pseudocode function that describes the operation.

If FPSCR.DN is 0, for Advanced SIMD, floating-point, or BF16 instructions that perform a floating-point operation, other than FABS, FNEG, FMAX*, and FMIN*, NaN outputs that derive from NaN inputs are derived as follows:

• If all of the following apply, an instruction outputs a quiet NaN derived from the first signaling NaN operand:
  — At least one operand is a signaling NaN.
  — The instruction is not trapped.

• If all of the following apply, an instruction outputs a quiet NaN derived from the first NaN operand:
  — At least one operand is a NaN, but none of the operands is a signaling NaN.
  — The instruction is not trapped.

If an output NaN is derived from an input NaN, the pseudocode functions FPAbs(), and FPNeg() can change the sign of the NaN,

### E1.3.7 Rounding

The rounding mode specifies how the exact result of a floating-point operation is rounded to a value in the destination format.

The rounding mode is either determined by the rounding mode control field FPSCR.RMode or by the instruction.

The rounding mode control field FPSCR.RMode can select the following rounding modes:

• Round to Nearest (RN) mode.
• Round towards Plus Infinity (RP) mode.
• Round towards Minus Infinity (RM) mode.
• Round towards Zero (RZ) mode.
The following two additional rounding modes are not selected by FPSCR.RMode, but are used by some instructions:

- Round to Odd mode.
- Round to Nearest with ties to away mode.

**Round to Nearest mode**

Round to Nearest rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format as follows:

- If the value before rounding has an absolute value that is too large to represent in the output format, the rounded value is an Infinity. The sign of the rounded value is the same as the sign of the value before rounding.
- If the value before rounding has an absolute value that is not too large to represent in the output format, the result is calculated as follows:
  - If the two nearest floating-point numbers bracketing the value before rounding are equally near, the result is the number with an even least significant digit.
  - If the two nearest floating-point numbers bracketing the value before rounding are not equally near, the result is the floating-point number nearest to the value before rounding.

Advanced SIMD arithmetic always uses the Round to Nearest setting, regardless of the value of the RMode bits.

**Round towards Plus Infinity mode**

Round towards Plus Infinity rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the floating-point number in the output format that is closest to and not less than the value before rounding. The result can be plus infinity.

**Round towards Minus Infinity mode**

Round towards Minus Infinity rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the number in the output format that is closest to and not greater than the value before rounding. The result can be minus infinity.

**Round towards Zero mode**

Round towards Zero rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the floating-point number in the output format that is closest to and not greater in absolute value than the value before rounding.

**Round to Nearest with Ties to Away**

Round to Nearest with Ties to Away rounding mode is used by the VCVTA (Advanced SIMD), VCVTA (floating-point), VRINTA (Advanced SIMD), and VRINTA (floating-point) instructions.

Round to Nearest with Ties to Away rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format as follows:

- If the value before rounding has an absolute value that is too large to represent in the output format, the rounded value is an Infinity, the sign of the rounded value is the same as the sign of the value before rounding.
- If the value before rounding has an absolute value that is not too large to represent in the output format, the result is calculated as follows:
  - If the two nearest floating-point numbers bracketing the value before rounding are equally near, the result is the larger number.
  - If the two nearest floating-point numbers bracketing the value before rounding are not equally near, the result is the floating-point number nearest to the value before rounding.
Round to Odd mode

Round to Odd mode is not defined by IEEE 754.

For BF16 instructions, if an intermediate format has at least two more bits of precision than the result format, Round to Odd mode is used and operates as follows:

- If the rounded value is inexact, the least significant bit of the fraction is set to 1.
- If the value is too large to represent in the single-precision format, the rounded value is a single-precision Infinity, the sign of the rounded value is the same as the sign of the value before rounding.

E1.3.8 Floating-point exceptions and exception traps

Execution of a floating-point instruction, or execution of an Advanced SIMD instruction that performs floating-point operations, can generate an exceptional condition, called a floating-point exception.

Note

An Advanced SIMD instruction that operates on floating-point values can perform multiple floating-point operations. Therefore, this section describes the handling of a floating-point exception on an operation, rather than on an instruction.

The architecture does not support asynchronous reporting of floating-point exceptions.

For each of the following floating-point exceptions, it is IMPLEMENTATION DEFINED whether an implementation includes synchronous exception generation:

- Input Denormal.
- Inexact.
- Underflow.
- Overflow.
- Divide by Zero.
- Invalid Operation.

If an implementation does not support synchronous exception generation from a floating-point exception, then that synchronous exception is never generated, and all statements about synchronous exception generation from that floating-point exception do not apply to the implementation.

If an implementation supports synchronous exception generation for a floating-point exception, then the registers that are presented to the exception handler are consistent with the state of the PE immediately before the instruction that caused the exception.

On return from a synchronous floating-point exception, software might not restore the cumulative exception flags.

Trapped floating-point exceptions are taken to the following levels:

- If a trapped floating-point exception occurs at EL0, the exception level it is taken to is as follows:
  - If EL2 is using AArch32 and HCR.TGE is 1, the exception is taken to EL2.
  - If EL2 is using AArch64 and HCR_EL2.TGE is 1, the exception is taken to EL2
  - Otherwise, the exception is taken to EL1
- If a trapped floating-point exception occurs at EL1, it is taken to EL1.
- If a trapped floating-point exception occurs at EL2, it is taken to EL2.
- If a trapped floating-point exception occurs at EL3, it is taken to EL3.

If a trapped floating-point exception is taken to an Exception level that is using AArch64, then it is reported in the ELR_ELx for the target Exception level, as described in Exception entry on page D1-4641.
If the exception is taken to an Exception level that is using AArch32, then it is taken as an Undefined Instruction exception, see *Undefined Instruction exception* on page G1-8964. The FPEXC identifies the floating-point exceptions that occurred since the corresponding status bits in that register were last set to 0.

**Input Denormal exceptions**

The cumulative floating-point exception bit FPSCR.IDC, and the trap enable bit FPSCR.IDE both relate to Input Denormal exceptions.

If a single-precision or double-precision floating-point input is flushed to zero, an Input Denormal exception is generated.

If a half-precision floating-point value is flushed to zero, an Input Denormal exception is not generated.

**Inexact exceptions**

The cumulative floating-point exception bit FPSCR.IXC, and the trap enable bit FPSCR.IXE both relate to Inexact exceptions.

If a denormalized output is flushed to zero, an Inexact exception is not generated.

If a result is not flushed to zero, and the result does not equal the result computed with unbounded exponent range and unbounded precision, then an Inexact exception is generated.

**Underflow exceptions**

The cumulative floating-point exception bit FPSR.UFC, and the trap enable bit FPSCR.UFE both relate to Underflow exceptions.

For the purpose of underflow floating-point exception generation, a denormalized number is detected before rounding is applied.

If the result of a floating-point operation is a denormalized number that is not flushed to zero, then the underflow exception is generated as follows:

- If FPSCR.UFE is 0, and the result is inexact, then the underflow floating-point exception is generated.
- If FPSCR.UFE is 1, for both exact and inexact results, the underflow floating-point exception is generated.

If the result of a floating-point operation is a denormalized number that is flushed to zero, then the Underflow floating-point exception is generated. The Underflow exception is not trapped regardless of the value of FPSCR.UFE.

**Overflow exceptions**

The cumulative floating-point exception bit FPSCR.OFC, and the trap enable bit FPSCR.OFE both relate to Overflow exceptions.

If the output of an instruction rounded with an unbounded exponent is greater than the maximum normalized number for the output precision, an overflow exception is generated.

If an untrapped Overflow exception is generated, the result is determined by the rounding mode and the sign of the result before rounding as follows:

- Round to Nearest carries all overflows to infinity with the sign of the result before rounding.
- Round towards Plus Infinity carries negative overflows to the most negative finite number of the output precision, and carries positive overflows to plus infinity.
- Round towards Minus Infinity carries positive overflows to the largest finite number of the output precision, and carries negative overflows to minus infinity.
- Round towards Zero carries all overflows to the output precision’s largest finite number with the sign of the result before rounding.
Divide by Zero exceptions

The cumulative floating-point exception bit FPSCR.DZC, and the trap enable bit FPSCR.DZE both relate to Divide by Zero exceptions.

If a floating-point operation divides a finite nonzero number by zero, a Divide by Zero exception is generated.

For the purpose of Divide by Zero exception generation, testing for zero occurs after flushing of denormalized numbers to zero.

A denormalized dividend that is flushed to zero is treated as zero and prevents Divide by Zero from occurring.

If the dividend is a finite nonzero, normalized number, and the divisor is a denormalized number, the divisor is treated as zero and causes Divide by Zero to occur.

For the reciprocal and reciprocal square root estimate functions, the dividend is assumed to be $+1.0$. This means that a zero or denormalized operand to these functions causes generation of a Divide by Zero floating-point exception.

If a floating-point operation divides a finite nonzero number by zero, and the Divide by Zero exception is untrapped, the result is a correctly signed infinity.

Invalid Operation exceptions

The cumulative floating-point exception bit FPSCR.IOC, and the trap enable bit FPSCR.IOE both relate to Invalid Operation exceptions.

For any floating-point instruction that performs a floating-point operation, if any of the following apply, the instruction generates an Invalid Operation exception:

- At least one operand is a signaling NaN, and the instruction is not FABS or FNEG.
- Magnitude subtraction of infinities.
- Multiplying a zero by an infinity.
- Dividing a zero by a zero.
- Dividing an infinity by an infinity.
- Square root of an operand that is less than zero.

For the purpose of Invalid Operation Exception generation, testing for zero occurs after flushing of denormalized numbers to zero. So a denormalized input that is flushed to zero is treated as zero.

If the input is one of: a quiet NaN, an infinity, or a number that overflows the values that can be represented in the output format, and if another exception is not generated to signal the condition, then a conversion from floating-point to either integer or fixed-point format, generates an Invalid Operation exception.

For the signaling compare instructions FCMPE and FCCMPE, if either of the source operands is any type of NaN, the instruction generates an Invalid Operation floating-point exception.

Floating-point exception traps

For Advanced SIMD instructions, and for floating-point instructions when floating-point exception trapping is not supported, these are non-trapping exceptions and the data-processing instructions do not generate any trapped exceptions.

For floating-point instructions when floating-point exception trapping is supported:

- The floating-point exceptions can be trapped, by setting trap enable bits in the FPSCR, see Floating-point exceptions and exception traps on page E1-7148, and:
  - When a trap is not enabled the corresponding floating-point exception updates the corresponding FPSCR cumulative bit, but does not generate an exception.
  - When a trap is enabled the corresponding floating-point exception does not update the FPSCR, but generates an exception. In this case, bits in the FPEXC indicate which floating-point exceptions have occurred.
• The definition of the Underflow floating-point exception is different in the trapped and cumulative exception cases. In the trapped case, the definition is:
  — The trapped Underflow floating-point exception occurs if the absolute value of the result of an operation, produced before rounding, is less than the minimum positive normalized number for the destination precision, regardless of whether the rounded result is inexact.
• As with cumulative exceptions, higher priority trapped exceptions can prevent lower priority exceptions from occurring, as described in Combinations of floating-point exceptions.
• For Invalid Operation floating-point exceptions, for details of which quiet NaN is produced as the default result, see NaN handling and the Default NaN on page A1-60.
• For Overflow floating-point exceptions, the sign bit of the default result is determined normally for the overflowing operation.
• For Divide by Zero floating-point exceptions, the sign bit of the default result is determined normally for a division. This means it is the exclusive OR of the sign bits of the two operands.

Table E1-5 shows the results of untrapped floating-point exceptions. That table uses the following abbreviations:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxNorm</td>
<td>The maximum normalized number of the destination precision.</td>
</tr>
<tr>
<td>RM</td>
<td>Round towards Minus Infinity mode, as defined in the IEEE 754 standard.</td>
</tr>
<tr>
<td>RN</td>
<td>Round to Nearest mode, as defined in the IEEE 754 standard.</td>
</tr>
<tr>
<td>RP</td>
<td>Round towards Plus Infinity mode, as defined in the IEEE 754 standard.</td>
</tr>
<tr>
<td>RZ</td>
<td>Round towards Zero mode, as defined in the IEEE 754 standard.</td>
</tr>
</tbody>
</table>

For more information about the IEEE 754 descriptions of the rounding modes, see Floating-point standards, and terminology on page A1-57.

### Table E1-5 Results of untrapped floating-point exceptions

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Default result for positive sign</th>
<th>Default result for negative sign</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOC, Invalid Operation</td>
<td>Quiet NaN</td>
<td>Quiet NaN</td>
</tr>
<tr>
<td>DZC, Divide by Zero</td>
<td>+infinity</td>
<td>-infinity</td>
</tr>
<tr>
<td>OFC, Overflow</td>
<td>RN, RP: +infinity</td>
<td>RN, RM: -infinity</td>
</tr>
<tr>
<td></td>
<td>RM, RZ: +MaxNorm</td>
<td>RP, RZ: -MaxNorm</td>
</tr>
<tr>
<td>UFC, Underflow</td>
<td>Normal rounded result</td>
<td>Normal rounded result</td>
</tr>
<tr>
<td>IXC, Inexact</td>
<td>Normal rounded result</td>
<td>Normal rounded result</td>
</tr>
<tr>
<td>IDC, Input Denormal</td>
<td>Normal rounded result</td>
<td>Normal rounded result</td>
</tr>
</tbody>
</table>

### Combinations of floating-point exceptions

Many pseudocode functions perform floating-point operations, including FixedToFP(), FPAdd(), FCompare(), FCompareEQ(), FCompareGE(), FCompareGT(), FDiv(), FMax(), FMin(), FMul(), FMulAdd(), FRecipEstimate(), FRecipStep(), FPRSQRTEstimate(), FPRSQRTStep(), FPSqrt(), FPSub(), and FPToFixed(). All of these operations can generate floating-point exceptions.

**Note**

FPAbs() and FPMeg() are not classified as floating-point operations because:
• They cannot generate floating-point exceptions.
• The floating-point operation behavior described in the following sections does not apply to them:
  — Flushing denormalized numbers to zero on page A1-58.
  — NaN handling and the Default NaN on page A1-60.
More than one exception can occur on the same operation. The only combinations of floating-point exceptions that can occur are:

- Overflow with Inexact.
- Underflow with Inexact.
- Input Denormal with other floating-point exceptions.

The priority order of these floating-point exceptions is that the Inexact exception is treated as lowest priority, and the Input Denormal exception is treated as highest priority.

When none of the floating-point exceptions caused by an operation is trapped, any floating-point exception that occurs causes the associated cumulative bit in the FPSCR to be set.

When one or more floating-point exceptions caused by an operation is trapped, the behavior of the instruction depends on the priority of the exceptions:

- If the higher priority floating-point exception is trapped, its trap handler is called. It is IMPLEMENTATION DEFINED whether any information about the lower priority floating-point exception is provided.

  Note

  Information about the lower priority floating-point exception might be provided in:

  — The FPEXC, if the exception generated by the trap is taken to an Exception level that is using AArch32.
  — The ESR_ELx.ISS field, if the exception generated by the trap is taken to an Exception level that is using AArch64.

  However, information might be provided in another IMPLEMENTATION DEFINED way, for example using an IMPLEMENTATION DEFINED register.

  Apart from this, the lower priority floating-point exception is ignored in this case.

- If the higher priority floating-point exception is untrapped, its cumulative bit is set to 1 and its default result is evaluated. Then the lower priority floating-point exception is handled normally, using this default result.

Some floating-point instructions specify more than one floating-point operation, as indicated by the pseudocode descriptions of the instruction. In such cases, a floating-point exception on one operation is treated as higher priority than a floating-point exception on another operation if the occurrence of the second floating-point exception depends on the result of the first operation. Otherwise, it is CONSTRAINED UNPREDICTABLE which floating-point exception is treated as higher priority.

For example, a VMLA.F32 instruction specifies a floating-point multiplication followed by a floating-point addition. The addition can generate Overflow, Underflow and Inexact floating-point exceptions, all of which depend on both operands to the addition and so are treated as lower priority than any floating-point exception on the multiplication. The same applies to Invalid Operation floating-point exceptions on the addition caused by adding opposite-signed infinities. The addition can also generate an Input Denormal floating-point exception, caused by the addend being a denormalized number while in Flush-to-zero mode. It is CONSTRAINED UNPREDICTABLE which of an Input Denormal floating-point exception on the addition and a floating-point exception on the multiplication is treated as higher priority, because the occurrence of the Input Denormal floating-point exception does not depend on the result of the multiplication. The same applies to an Invalid Operation floating-point exception on the addition caused by the addend being a signaling NaN.

  Note

The VFMA instruction performs a vector addition and a vector multiplication as a single operation. The VFMS instruction performs a vector subtraction and a vector multiplication as a single operation.
E1.3.9 Controls of Advanced SIMD operation that do not apply to floating-point operation

Earlier architectures permitted implementation of either, both, or neither of the Advanced SIMD and floating-point additions to the base instruction set, and provided some controls that applied to the Advanced SIMD functionality but not to the floating-point functionality. From the introduction of Armv8, Advanced SIMD functionality cannot be separated from floating-point functionality, but in AArch32 state these controls function as they did in earlier architectures. This means they apply only to the following instructions and instruction encodings:

- All instructions with encodings defined in:
  - Advanced SIMD data-processing on page F3-7314, for the T32 instruction set.
  - Advanced SIMD data-processing on page F4-7422, for the A32 instruction set.
- All instructions with encodings defined in:
  - Advanced SIMD element or structure load/store on page F3-7350, for the T32 instruction set.
  - Advanced SIMD element or structure load/store on page F4-7434, for the A32 instruction set.
- The form of the VDUP instruction described in VDUP (general-purpose register) on page F6-8370.
- The byte and halfword forms of the VMOV instructions described in each of:
  - VMOV (general-purpose register to scalar) on page F6-8550.
  - VMOV (scalar to general-purpose register) on page F6-8554.

The controls of this functionality are:

- The CPACR.ASEDIS field.
- The HCPTR.TASE field.

In an implementation that supports Advanced SIMD functionality, support for each of these controls is optional:

- If the CPACR.ASEDIS control is not supported then the CPACR.ASEDIS field is RAZ/WI. This is equivalent to the control permitting the execution of Advanced SIMD instructions at EL1 and EL0.
- If the HCPTR.TASE control is not supported then the HCPTR.TASE field is RAZ/WI. This means the HCPTR does not provide a control that can trap Non-secure execution of Advanced SIMD instructions to Hyp mode.

E1.3.10 Implications of not including Advanced SIMD and floating-point support

In general, the architecture requires the inclusion of the Advanced SIMD and floating-point instructions in all instruction sets. Exceptionally, for implementation targeting specialized markets, Arm might produce or license an Armv8-A implementation that does not provide any support for Advanced SIMD and floating-point instructions. In such an implementation, in AArch32 state:

- Each of the CPACR.[cp10, cp11] fields is RES0.
- The CPACR.ASEDIS bit is RES0.
- Each of the HCPTR.[TASE, TCP10, TCP11] fields is RES1.
- Each of the NSACR.[NSASEDIS, cp10, cp11] fields is RES0.
- The FPEXC register is UNDEFINED.

E1.3.11 Pseudocode description of floating-point operations

The following subsections contain pseudocode definitions of the floating-point functionality supported by the architecture:

- Generation of specific floating-point values on page E1-7154.
- Floating-point negation and absolute value on page E1-7154.
- Floating-point value unpacking on page E1-7154.
- Floating-point exception and NaN handling on page E1-7154.
- Floating-point rounding on page E1-7154.
- Selection of Arm standard floating-point arithmetic on page E1-7154.
- Floating-point comparisons on page E1-7155.
• Floating-point maximum and minimum on page E1-7155.
• Floating-point addition and subtraction on page E1-7155.
• Floating-point multiplication and division on page E1-7155.
• Floating-point fused multiply-add on page E1-7155.
• Floating-point reciprocal estimate and step on page E1-7155.
• Floating-point square root on page E1-7156.
• Floating-point reciprocal square root estimate and step on page E1-7156.
• Floating-point conversions on page E1-7157.

Generation of specific floating-point values

The following pseudocode functions generate specific floating-point values. The sign argument is '0' for the positive version and '1' for the negative version:
• FPInfinity().
• FPMaxNormal().
• FPZero().
• FPTwo().
• FPThree().
• FPDefaultNaN().

Floating-point negation and absolute value

The floating-point negation and absolute value operations affect only the sign bit. They do not treat NaN operands specially, nor denormalized number operands when flush-to-zero is selected.

The floating-point negation operation is described by the pseudocode function FPNeg(). The floating-point absolute value operation is described by the pseudocode function FPAbs().

Floating-point value unpacking

The FPUnpack() function determines the type of a floating-point number, defined by FPType{}, and its numerical value. It also does flush-to-zero processing on input operands.

Floating-point exception and NaN handling

The FPProcessException() procedure checks whether a floating-point exception is trapped, and handles it accordingly. The floating-point exception types are defined by FPExc{}.

The FPProcessNaN() function processes a NaN operand, producing the correct result value and generating an Invalid Operation floating-point exception if necessary. The FPProcessNaNs() function performs the standard NaN processing for a two-operand operation. The FPProcessNaNs3() function performs the standard NaN processing for a three-operand operation.

Floating-point rounding

The FPRound() function rounds and encodes a floating-point result to a specified destination format. This includes processing Overflow, Underflow and Inexact floating-point exceptions and performing flush-to-zero processing on result values.

Selection of Arm standard floating-point arithmetic

The StandardFPSCRValue() function returns the FPSCR value that selects Arm standard floating-point arithmetic. Most of the arithmetic functions have a Boolean fpscr_controlled argument that is TRUE for Floating-point operations and FALSE for Advanced SIMD operations, and that selects between using the real FPSCR value and this value.
Floating-point comparisons

The FPCompare() function compares two floating-point numbers, producing a \{N, Z, C, V\} Condition flags result as shown in Table E1-6:

<table>
<thead>
<tr>
<th>Comparison result</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Less than</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Greater than</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Unordered</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

This result defines the operation of the V_CMP floating-point instruction. The V_CMP instruction writes these flag values in the FPSR. After using a VMRS instruction to transfer them to the APSR, they can control conditional execution as shown in Table F1-1 on page F1-7229.

The FPCompareEQ(), FPCompareGE(), and FPCompareGT() functions describe the operation of Advanced SIMD instructions that perform floating-point comparisons.

Floating-point maximum and minimum

The FPMAX() function returns the maximum of two floating-point numbers. The FPMIN() function returns the minimum of two floating-point numbers.

Floating-point addition and subtraction

The FPADD() function adds two floating-point numbers. The FPSUB() function subtracts one floating-point number from another floating-point number.

Floating-point multiplication and division

The FPMUL() function multiplies two floating-point numbers. The FPDIV() function divides one floating-point number by another floating-point number.

Floating-point fused multiply-add

The FPMULADD() function performs a floating-point fused multiply-add.

Floating-point reciprocal estimate and step

The Advanced SIMD implementation includes instructions that support Newton-Raphson calculation of the reciprocal of a number.

The VRECPE instruction produces the initial estimate of the reciprocal. It uses the pseudocode functions:

- FPREcipEstimate().
- UnsignedRecipEstimate().
Table E1-7 shows the results where input values are out of range.

Table E1-7 VRECPE results for out of range inputs

<table>
<thead>
<tr>
<th>Number type</th>
<th>Input Vm[i]</th>
<th>Result Vd[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>&lt;= 0x7FFFFFFF</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>Floating-point</td>
<td>NaN</td>
<td>Default NaN</td>
</tr>
<tr>
<td>Floating-point</td>
<td>±0 or denormalized number</td>
<td>±infinity a</td>
</tr>
<tr>
<td>Floating-point</td>
<td>±infinity</td>
<td>±0</td>
</tr>
<tr>
<td>Floating-point</td>
<td>Absolute value &gt;= 2(^{126})</td>
<td>±0</td>
</tr>
</tbody>
</table>

a. FPSCR.DZC is set to 1

The Newton-Raphson iteration:

\[ x_{n+1} = x_n \left(2-\frac{dx_n}{x_0}\right) \]

converges to \((1/d)\) if \(x_0\) is the result of VRECPE applied to \(d\).

The VRECP instruction performs a \((2 - \text{op1} \times \text{op2})\) calculation and can be used with a multiplication to perform a step of this iteration. The functionality of this instruction is defined by the FPRecipStep() pseudocode function.

Table E1-8 shows the results where input values are out of range.

Table E1-8 VRECP results for out of range inputs

<table>
<thead>
<tr>
<th>Input Vn[i]</th>
<th>Input Vm[i]</th>
<th>Result Vd[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any NaN</td>
<td>-</td>
<td>Default NaN</td>
</tr>
<tr>
<td>±0.0 or denormalized number</td>
<td>±infinity</td>
<td>2.0</td>
</tr>
<tr>
<td>±infinity</td>
<td>±0.0 or denormalized number</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Floating-point square root

The FPSqrt() function returns the square root of a floating-point number.

Floating-point reciprocal square root estimate and step

The Advanced SIMD implementation includes instructions that support Newton-Raphson calculation of the reciprocal of the square root of a number.

The VRSQRTE instruction produces the initial estimate of the reciprocal of the square root. It uses the pseudocode functions:

- FPSqrtEstimate().
- UnsignedRSqrtEstimate().
Table E1-9 shows the results where input values are out of range.

### Table E1-9 VRSQRTE results for out of range inputs

<table>
<thead>
<tr>
<th>Number type</th>
<th>Input Vm[i]</th>
<th>Result Vd[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>&lt;= 0x3FFFFFFF</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>Floating-point</td>
<td>NaN, -(normalized number), -infinity</td>
<td>Default NaN</td>
</tr>
<tr>
<td>Floating-point</td>
<td>-0 or -(denormalized number)</td>
<td>- infinity</td>
</tr>
<tr>
<td>Floating-point</td>
<td>+0 or +(denormalized number)</td>
<td>+infinity</td>
</tr>
<tr>
<td>Floating-point</td>
<td>+infinity</td>
<td>+0</td>
</tr>
</tbody>
</table>

a. FPSCR.DZC is set to 1.

The Newton-Raphson iteration:

\[ x_{n+1} = x_n \frac{(3-dx_n^2)}{2} \]

converges to \( \frac{1}{\sqrt{d}} \) if \( x_0 \) is the result of VRSQRT applied to \( d \).

The VRSQRTS instruction performs a \( (3 – op1 \times op2)/2 \) calculation and can be used with two multiplications to perform a step of this iteration. The functionality of this instruction is defined by the FPRSqrtStep() pseudocode function.

Table E1-10 shows the results where input values are out of range.

### Table E1-10 VRSQRTS results for out of range inputs

<table>
<thead>
<tr>
<th>Input Vn[i]</th>
<th>Input Vm[i]</th>
<th>Result Vd[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any NaN</td>
<td>-</td>
<td>Default NaN</td>
</tr>
<tr>
<td>-</td>
<td>Any NaN</td>
<td>Default NaN</td>
</tr>
<tr>
<td>±0.0 or denormalized number</td>
<td>±infinity</td>
<td>1.5</td>
</tr>
<tr>
<td>±infinity</td>
<td>±0.0 or denormalized number</td>
<td>1.5</td>
</tr>
</tbody>
</table>

FPRSqrtStep() calls the FPHalvedSub() pseudocode function.

### Floating-point conversions

The FPConvert() pseudocode function performs conversions between half-precision, single-precision, and double-precision floating-point numbers.

The FPToFixed() and FixedToFP() functions perform conversions between floating-point numbers and integers or fixed-point numbers.
E1.4 About the AArch32 System register interface

AArch32 state provides a System register encoding space, which is indexed by the parameter set \{coproc, opc1, CRn, CRm, opc2\}, and a set of System register access instructions. This encoding space is used for:

- System registers.
- System instructions, for:
  - Cache and branch predictor maintenance.
  - Address translation.
  - TLB maintenance.

This encoding space uses only the coproc values 0b111x.

Note

The encoding space with coproc values 0b101x is redefined to provide Advanced SIMD and floating-point functionality.

The coproc encodings provide access to System register encoding space as follows:

- The (coproc==0b1111) encodings provide system control functionality, by providing access to System registers and System instructions. This includes architecture and feature identification, as well as control, status information and configuration support.

  The following sections give a general description of these encodings:
  - About the System registers for VMSAv8-32 on page G5-9281.
  - AArch32 VMSA organization of registers in the (coproc==0b1111) encoding space on page G7-9304.
  - Functional grouping of VMSAv8-32 System registers on page G5-9286.

  These encodings also provide:
  - The Performance Monitor registers. For more information, see Chapter D11 The Performance Monitors Extension.
  - The Activity Monitor registers. For more information, see Chapter D12 The Activity Monitors Extension.

  The (coproc==0b1110) encodings provide access to additional registers, which support:
  - Debug, see Chapter G2 AArch32 Self-hosted Debug.
  - The Jazelle identification registers, see Jazelle support on page E1-7139.

UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for AArch32 System register accesses on page G8-9323 gives information more information about permitted accesses to the System registers in AArch32 state.

Most functionality in the (coproc==0b111x) encoding space cannot be accessed by software executing at EL0. This manual clearly identifies those functions that can be accessed at EL0.

For more information:

- About this encoding space, including the naming of the parameters that index the space, see The AArch32 System register interface on page G1-8993.
- About the System interface access instructions, see System register access instructions on page F2-7277.
E1.5 Exceptions

The Arm architecture uses the following terms to describe various types of exceptional condition:

**Exceptions**

In the Arm architecture, an exception causes entry to EL1, EL2, or EL3. If the Exception level that is entered is using AArch32, it also causes entry to the PE mode in which the exception must be taken. A software handler for the exception is then executed.

--- Note ---

The term *floating-point exception* does not use this meaning of *exception*. This term is described later in this list.

---

Exceptions include:

- Reset.
- Interrupts.
- Memory system aborts.
- Undefined instructions.
- Supervisor calls (SVCs), Secure Monitor calls (SMCs), and Hypervisor calls (HVCs).
- Debug exceptions.

Most details of exception handling are not visible to application level software, and are described in *Handling exceptions that are taken to an Exception level using AArch32 on page G1-8929*. In an implementation that includes all the Exception levels, aspects that are visible to application level software are:

- The `SVC` instruction causes a Supervisor Call exception. This provides a mechanism for unprivileged software to make a call to the operating system, or other system component that is accessible only at EL1.
- The `SMC` instruction causes a Secure Monitor Call exception, but only if software execution is at EL1 or higher. Unprivileged software can only cause a Secure Monitor Call exception by methods defined by the operating system, or by another component of the software system that executes at EL1 or higher.
- The `HVC` instruction causes a Hypervisor Call exception, but only if software execution is at EL1 or higher. Unprivileged software can only cause a Hypervisor Call exception by methods defined by the hypervisor, or by another component of the software system that executes at EL1 or higher.
- The `BKPT` instruction causes a Breakpoint Instruction exception, which is taken as a Prefetch Abort exception. This provides a mechanism for a debugger to insert breakpoints into unprivileged software, or for unprivileged software to make a call into a debugger that is accessible at EL1.
- The `WFI` (Wait for Interrupt) instruction provides a hint that nothing needs to be done until an interrupt or another WFI wakeup event occurs, see *Wait For Interrupt on page G1-8991*. This means the hardware might enter a low-power state until the wakeup event occurs.
- The `WFE` (Wait for Event) instruction provides a hint that nothing needs to be done until either an SEV instruction generates an event, or another WFE wakeup event occurs, see *Wait For Event and Send Event on page G1-8988*. This means the hardware might enter a low-power state until the wakeup event occurs.

**Floating-point exceptions**

These relate to exceptional conditions encountered during floating-point arithmetic, such as Divide by Zero or Overflow. For more information, see:

- *Floating-point exceptions and exception traps on page E1-7148.*
- The `FPEXC` and `FPSCR` register descriptions.
Chapter E2
The AArch32 Application Level Memory Model

This chapter gives an application level description of the memory model for software executing in AArch32 state. This means it describes the memory model for execution in EL0 when EL0 is using AArch32 in the following sections:

• About the Arm memory model on page E2-7162.
• Atomicity in the Arm architecture on page E2-7164.
• Definition of the memory model on page E2-7168.
• Ordering of translation table walks on page E2-7187.
• Caches and memory hierarchy on page E2-7188.
• Alignment support on page E2-7193.
• Endian support on page E2-7195.
• Memory types and attributes on page E2-7198.
• Mismatched memory attributes on page E2-7208.
• Synchronization and semaphores on page E2-7211

Note

In this chapter, System register names usually link to the description of the register in Chapter G8 AArch32 System Register Descriptions, for example SCTLR.
E2.1 About the Arm memory model

The Arm architecture is a weakly ordered memory architecture that permits the observation and completion of memory accesses in a different order from the program order. The following sections of this chapter provide the complete definition of the memory model, this introduction is not intended to contradict the definition found in those sections. In general, the basic principles of the memory model are:

- To provide a memory model that has similar weaknesses to those found in the memory models used by high-level programming languages such as C or Java. For example, by permitting independent memory accesses to be reordered as seen by other observers.
- To avoid the requirement for multi-copy atomicity in the majority of memory types.
- The provision of instructions and memory barriers to compensate for the lack of multi-copy atomicity in the cases where it would be needed.
- The use of address, data, and control dependencies in the creation of order so as to avoid having excessive numbers of barriers or other explicit instructions in common situations where some order is required by the programmer or the compiler.

This section contains:
- Address space.
- Memory type overview.

E2.1.1 Address space

Address calculations are performed using 32-bit registers. Supervisory software determines the valid address range. Attempting to access an address that is not valid generates an MMU fault.

Address calculations are performed modulo $2^{32}$.

The result of an address calculation is UNKNOWN if it overflows or underflows the 32-bit address range $A[31:0]$.

Memory accesses use the MemA[], MemO[], MemU[], and MemU_unpriv[] pseudocode functions:
- The MemA[] function makes an aligned access of the required type.
- The MemO[] function makes an ordered access of the required type.
- The MemU[] function makes an unaligned access of the required type.
- The MemU_unpriv[] function makes an unaligned, unprivileged access of the required type.

Each of these functions calls Mem_with_type[] function, which specifies the required access. This calls AArch32.MemSingle[], which performs an atomic, little-endian read of size bytes.

The AccType enumeration defines the different access types.

Note

- Chapter G4 The AArch32 System Level Memory Model and Chapter G5 The AArch32 Virtual Memory System Architecture include descriptions of memory system features that are transparent to the application, including memory access, address translation, memory maintenance instructions, and alignment checking and the associated fault handling. These chapters also reference pseudocode descriptions of these operations.
- For references to the pseudocode that relates to memory accesses, see Basic memory access on page G4-9142, Unaligned memory access on page G4-9142, and Aligned memory access on page G4-9142.

E2.1.2 Memory type overview

The architecture provides the following mutually-exclusive memory types:

- **Normal** This is generally used for bulk memory operations, both read/write and read-only operations.
- **Device** The Arm architecture forbids speculative reads of any type of Device memory. This means Device memory types are suitable attributes for read-sensitive locations.
Locations of the memory map that are assigned to peripherals are usually assigned the Device memory attribute.

Device memory has additional attributes that have the following effects:

- They prevent aggregation of reads and writes, maintaining the number and size of the specified memory accesses. See Gathering on page E2-7204.
- They preserve the access order and synchronization requirements, both for accesses to a single peripheral and where there is a synchronization requirement on the observability of one or more memory write and read accesses. See Reordering on page E2-7205.
- They indicate whether a write can be acknowledged other than at the end point. See Early Write Acknowledgement on page E2-7205.

For more information on Normal memory and Device memory, see Memory types and attributes on page E2-7198.

——— Note ————

Earlier versions of the Arm architecture defined a single Device memory type and a Strongly-ordered memory type. A Note in Device memory on page E2-7202 describes how these memory types map onto the memory types used from the introduction of Armv8.
E2.2 Atomicity in the Arm architecture

Atomicity is a feature of memory accesses, described as atomic accesses. The Arm architecture description refers to two types of atomicity, single-copy atomicity and multi-copy atomicity. The atomicity requirements for memory accesses depend on the memory type, and whether the access is explicit or implicit. For more information, see:

- Requirements for single-copy atomicity.
- Properties of single-copy atomic accesses on page E2-7165.
- Multi-copy atomicity on page E2-7165.
- Requirements for multi-copy atomicity on page E2-7166.
- Concurrent modification and execution of instructions on page E2-7166.

For more information about the memory types, see Memory type overview on page E2-7162.

E2.2.1 Requirements for single-copy atomicity

In AArch32 state, the single-copy atomic PE accesses are:

- All byte accesses.
- All halfword accesses to halfword-aligned locations.
- All word accesses to word-aligned locations.
- Memory accesses caused by LDREXD and STREXD instructions to doubleword-aligned locations.

LDM, LDC, LDRD, STM, STC, STRD, PUSH, POP, RFE, SRS, VLDM, VLDR, VSTM, and VSTR instructions are executed as a sequence of word-aligned word accesses. Each 32-bit word access is guaranteed to be single-copy atomic. The architecture does not require subsequences of two or more word accesses from the sequence to be single-copy atomic.

LDRD and STRD accesses to 64-bit aligned locations are 64-bit single-copy atomic as seen by translation table walks and accesses to translation tables.

Note

This requirement has been added to avoid the need for complex measures to avoid atomicity issues when changing translation table entries, without creating a requirement that all locations in the memory system are 64-bit single-copy atomic. This addition means:

- The system designer must ensure that all writable memory locations that might be used to hold translations, such as bulk SDRAM, can be accessed with 64-bit single-copy atomicity.
- Software must ensure that translation tables are not held in memory locations that cannot meet this atomicity requirement, such as peripherals that are typically accessed using a narrow bus.

This requirement places no burden on read-only memory locations for which reads have no side effects, since it is impossible to detect the size of memory accesses to such locations.

Advanced SIMD element and structure loads and stores are executed as a sequence of accesses of the element or structure size. The architecture requires the element accesses to be single-copy atomic if and only if both:

- The element size is 32 bits, or smaller.
- The elements are naturally-aligned.

Accesses to 64-bit elements or structures that are 32-bit aligned are executed as a sequence of 32-bit accesses, each of which is single-copy atomic. The architecture does not require subsequences of two or more 32-bit accesses from the sequence to be single-copy atomic.

When an access is not single-copy atomic by the rules described in this section, it is executed as a sequence of one or more accesses that aggregate to the size of the original access. Each of the accesses in this sequence is single-copy atomic, at least at the byte level.
In this section, the terms before the write operation and after the write operation mean before or after the write operation has had its effect on the coherence order of the bytes of the memory location accessed by the write operation.

If, according to these rules, an instruction is executed as a sequence of accesses, a synchronous Data Abort exception or Debug state entry can be taken during that sequence. This causes execution of the instruction to be abandoned. See Data Abort exception on page G1-8975 and, when FEAT_LSMAOC is implemented, Taking an interrupt or other exception during a multiple-register load or store on page G1-8963.

If the synchronous Data Abort exception is returned from using the preferred return address, the instruction that generated the sequence of accesses is re-executed and so any access that was performed before the exception was taken is repeated. This also applies to an exit from Debug state.

Note
The exception behavior for these multiple access instructions means they are not suitable for use for writes to memory for the purpose of software synchronization.

For implicit accesses:
- Cache linefills and evictions have no effect on the single-copy atomicity of explicit transactions or instruction fetches.
- Instruction fetches are single-copy atomic:
  - At 32-bit granularity in A32 state.
  - At 16-bit granularity in T32 state.
- Translation table walks are performed using accesses that are single-copy atomic:
  - At 32-bit granularity when using Short-descriptor format translation tables.
  - At 64-bit granularity when using Long-descriptor format translation tables.

**E2.2.2 Properties of single-copy atomic accesses**

A memory access instruction that is single-copy atomic has the following properties:

1. For a pair of overlapping single-copy atomic store instructions, all of the overlapping writes generated by one of the stores are Coherence-after the corresponding overlapping writes generated by the other store.

2. For a single-copy atomic load instruction \( L_1 \) that overlaps a single-copy atomic store instruction \( S_2 \), if one of the overlapping reads generated by \( L_1 \) Reads-from one of the overlapping writes generated by \( S_2 \), then none of the overlapping writes generated by \( S_2 \) are Coherence-after the corresponding overlapping reads generated by \( L_1 \).

For more information, see Definition of the memory model on page E2-7168.

**E2.2.3 Multi-copy atomicity**

In a multiprocessing system, writes to a memory location are multi-copy atomic if the following conditions are both true:

- All writes to the same location are serialized, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes.
- A read of a location does not return the value of a write until all observers observe that write.

Note
Writes that are not coherent are not multi-copy atomic.
E2.2.4 Requirements for multi-copy atomicity

For Normal memory, writes are not required to be multi-copy atomic.

For Device memory, writes are not required to be multi-copy atomic.

The memory model is Other-multi-copy atomic. For more information, see Ordering constraints on page E2-7174.

E2.2.5 Concurrent modification and execution of instructions

The architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization.

Concurrent modification and execution of instructions can lead to the resulting instruction performing any behavior that can be achieved by executing any sequence of instructions that can be executed from the same Exception level, except where the instruction before modification and the instruction after modification are:

• When executing the A32 instruction set, a B, BKPT, BL, HVC, ISB, NOP, SMC, or SVC instruction.
• When executing the T32 instruction set. a 16-bit B, BKPT, BLX, BX, NOP, or SVC instruction.

In addition, for the 32-bit T32 instructions, for which Instruction encodings on page F1-7224 describes the meaning of \{hw1, hw2\}:

• hw1 of a 32-bit BL (immediate) instruction can be concurrently modified to hw1 of another BL (immediate) instruction:
  — This means that some of the most significant bits of the immediate value can be modified.
• hw1 of a 32-bit BLX (immediate) instruction can be concurrently modified to hw1 of another BLX immediate instruction:
  — This means that some of the most significant bits of the immediate value can be modified.
• hw1 of a 32-bit BL (immediate) or BLX (immediate) instruction can be concurrently modified to a T32 16-bit B, BX, BLX, BKPT, or SVC instruction. This modification also works in reverse.
• hw2 of a 32-bit BL (immediate) instruction can be concurrently modified to hw2 of another BL (immediate) instruction with a different immediate:
  — This means that some bits of the immediate value, including the least significant bits, can be modified.
• hw2 of a 32-bit BLX (immediate) instruction can be concurrently modified to hw2 of another BLX (immediate) instruction with a different immediate:
  — This means that some bits of the immediate value, including the least significant bits, can be modified.
• hw2 of a 32-bit B (immediate) instruction with a condition field can be concurrently modified to hw2 of another 32-bit B (immediate) instruction with a condition field with a different immediate:
  — This means that some bits of the immediate value, including the least significant bits, can be modified.
• hw2 of a 32-bit B (immediate) instruction without a condition field can be concurrently modified to hw2 of another 32-bit B (immediate) instruction without a condition field:
  — This means that some bits of the immediate value, including the least significant bits, can be modified.

   — Note ————

• In the T32 instruction set:
  — The only encodings of BKPT and SVC are 16-bit.
  — The only encoding of BL is 32-bit.
• The ISB instruction can be concurrently modified and executed in the A32 and A64 instruction sets, but not in the T32 instruction set.

For the instructions explicitly identified in this section, the architecture guarantees that, after modification of the instruction, behavior is consistent with execution of either:

• The instruction originally fetched.
• A fetch of the modified instruction.

The instructions to which this applies are the B, BL, NOP, BKPT, SVC, HVC, and SMC instructions.
For both instruction sets, if one thread of execution changes a conditional branch instruction to another conditional branch instruction, and the change affects both the condition field and the branch target, execution of the changed instruction by another thread of execution before the change is synchronized can lead to either:

- The old condition being associated with the new target address.
- The new condition being associated with the old target address.

These possibilities apply regardless of whether the condition, either before or after the change to the branch instruction, is the always condition.

For all other instructions, to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior, instruction modifications must be explicitly synchronized before they are executed. The required synchronization is as follows:

1. No PE must be executing an instruction when another PE is modifying that instruction.
2. To ensure that the modified instructions are observable, a PE that is writing the instructions must issue the following sequence of instructions and operations:
   - Coherency example for self-modifying code
   - Enter this code with <Rt> containing a new 32-bit instruction, to be held in Cacheable space at a location pointed to by Rn. Use STRH in the first line instead of STR for a 16-bit instruction.

\[
\begin{align*}
\text{STR} \ <\text{Rt}>, \ [\text{Rn}] & \quad \text{Clean data cache by MVA to point of unification (PoU)} \\
\text{DCCMVAU} \ Rn & \quad \text{Clean data cache by MVA to point of unification (PoU)} \\
\text{DSB} & \quad \text{Ensure visibility of the data stored} \\
\text{ICIMVAU} \ Rn & \quad \text{Invalidate instruction cache by VA to PoU} \\
\text{BPIMVA} \ Rn & \quad \text{Invalidate branch predictor by MVA to PoU} \\
\text{DSB} & \quad \text{Ensure visibility of the data stored}
\end{align*}
\]

**Note**

- The DCCMVAU operation is not required if the area of memory is either Non-cacheable or Write-Through Cacheable.
- If the contents of physical memory differ between the mappings, changing the mapping of VAs to PAs can cause the instructions to be concurrently modified by one PE and executed by another PE. If the modifications affect instructions other than those listed as being acceptable for modification, synchronization must be used to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.

3. In a multiprocessor system, the ICIMVAU and BPIMVA are broadcast to all PEs within the Inner Shareable domain of the PE running this sequence. However, once the modified instructions are observable, each PE that is executing the modified instructions must issue the following instruction to ensure execution of the modified instructions:

\[
\text{ISB} \quad ; \text{Synchronize fetched instruction stream}
\]

For more information about the required synchronization operation, see *Synchronization and coherency issues between data and instruction accesses* on page E2-7190.

For information about memory accesses caused by instruction fetches, see *Ordering constraints* on page E2-7174.
E2.3 Definition of the memory model

This section describes observation and ordering in the memory model. It contains the following subsections:

- Basic definitions.
- Dependency definitions on page E2-7170.
- Ordering relations on page E2-7172.
- Ordering constraints on page E2-7174.
- Internal visibility requirement on page E2-7174.
- External ordering constraints on page E2-7174.
- Completion and endpoint ordering on page E2-7176.
- Ordering of instruction fetches on page E2-7178.
- Restrictions on the effects of speculation on page E2-7179.
- Memory barriers on page E2-7181.

For more information on endpoint ordering of memory accesses, see Reordering on page E2-7205.

In the memory model, the Shareability memory attribute indicates the degree to which hardware must ensure memory coherency between a set of observers, see Memory types and attributes on page E2-7198.

The architecture defines additional memory attributes and associated behaviors, which are defined in the system level section of this manual. See:

- Chapter G4 The AArch32 System Level Memory Model.
- Chapter G5 The AArch32 Virtual Memory System Architecture.

See also Mismatched memory attributes on page E2-7208.

E2.3.1 Basic definitions

The memory model provides a set of definitions that are used to construct conditions on the permitted sequences of accesses to memory.

Observer

An Observer refers to a processing element or mechanism in the system, such as a peripheral device, that can generate reads from, or writes to, memory.

Common Shareability Domain

For the purpose of this section, all Observers are assumed to belong to a Common Shareability Domain. All read and write effects access only Normal memory locations in a Common Shareability Domain, and excludes the situations described in Mismatched memory attributes on page E2-7208.

Location

A Location is a byte that is associated with an address in the physical address space.

Note

It is expected that an operating system will present the illusion to the application programmer that is consistent with a location also being considered as a byte that is associated with an address in the virtual address space.

Effects

The Effects of an instruction can be:

- Register effects.
- Memory effects.
- Barrier effects.
- Branching effects.
The effects of an instruction $I_1$ are said to appear in program order before the effects of an instruction $I_2$ if and only if $I_1$ occurs before $I_2$ in the order specified by the program. Each effect generated by an instruction has a unique identifier, which characterizes it amongst the events generated by the same instruction.

**Register effect**

The *Register effects* of an instruction are register reads or register writes of that instruction. For an instruction that accesses registers, a register read effect is generated for each register read by the instruction and a register write effect is generated for each register written by the instruction. An instruction may generate both read and write register Register effects.

**Memory effect**

The *Memory effects* of an instruction are the memory reads or writes generated by that instruction. For an instruction that accesses memory, a memory read effect is generated for each Location read by the instruction and a memory write effect is generated for each Location written by the instruction. An instruction may generate both read and write Memory effects.

**Branching effect**

The *Branching effects* of an instruction are effects which correspond to a branching decision being taken.

---

**Note**

Conditional and compare-and-swap instructions do not create Branching effects.

---

**Intrinsic order**

There is a per-instruction *Intrinsic order* relation that provides a partial order over the effects of that instruction, according to the operation of that instruction.

The operation of an instruction is defined by the pseudocode in Chapter F5 T32 and A32 Base Instruction Set Instruction Descriptions.

**Reads-from-register**

The *Reads-from-register* relation couples register read and write effects to the same register such that each register read effect is paired with exactly one register write effect in the execution of a program. A register read effect $R_2$ Reads-from-register a register write effect $W_1$ to the same register if and only if $R_2$ takes its data from $W_1$. By construction $W_1$ must be in program order before $R_2$ and there must be no intervening write to the same register in program order between $W_1$ and $R_2$.

**Reads-from**

The *Reads-from* relation couples memory read and write effects to the same Location such that each memory read effect is paired with exactly one memory write effect in the execution of a program. A memory read effect $R_2$ from a Location Reads-from a memory write effect $W_1$ to the same Location if and only if $R_2$ takes its data from $W_1$.

**Coherence order**

There is a per-location *Coherence order* relation that provides a total order over all memory write effects from all coherent Observers to that Location, starting with a notional memory write effect of the initial value. The Coherence order of a Location represents the order in which memory write effects to the Location arrive at memory.

**Local read successor**

A memory read effect $R_2$ of a Location is the *Local read successor* of a memory write effect $W_1$ from the same Observer to the same Location if and only if $W_1$ appears in program order before $R_2$ and there is not a memory write effect $W_3$ from the same Observer to the same Location appearing in program order between $W_1$ and $R_2$. 
Local write successor

A memory write effect \( W_2 \) of a Location is a *Local write successor* of a memory read or write effect \( RW_1 \) from the same Observer to the same Location if and only if \( RW_1 \) appears in program order before \( W_2 \).

Coherence-after

A memory write effect \( W_2 \) to a Location is *Coherence-after* another memory write effect \( W_1 \) to the same Location if and only if \( W_2 \) is sequenced after \( W_1 \) in the Coherence order of the Location.

A memory write effect \( W_2 \) to a Location is Coherence-after a memory read effect \( R_1 \) of the same location if and only if \( R_1 \) Reads-from a memory write effect \( W_3 \) to the same Location and \( W_2 \) is Coherence-after \( W_3 \).

Observed-by

A memory read or write effect \( RW_1 \) from an Observer is *Observed-by* a memory write effect \( W_2 \) from a different Observer if and only if \( W_2 \) is coherence-after \( RW_1 \).

A memory write effect \( W_1 \) from an Observer is Observed-by a memory read effect \( R_2 \) from a different Observer if and only if \( R_2 \) Reads-from \( W_1 \).

---

**Note**
The Observed-by relation only relates Memory effects generated by different Observers.

---

Overlapping accesses

Two Memory effects overlap if and only if they access the same Location. Two instructions overlap if and only if one or more of their generated Memory effects overlap.

Single-copy-atomic-ordered-before

A memory read effect \( R_1 \) is *Single-copy-atomic-ordered-before* another memory read effect \( R_2 \) if and only if all of the following statements are true:

- \( R_1 \) and \( R_2 \) are memory read effects generated by the same instruction.
- \( R_1 \) is not a Local read successor of a memory write effect.
- \( R_2 \) is a Local read successor of a memory write effect.

DMB FULL

A DMB FULL is a DMB with neither the LD or the ST qualifier.

Where this section refers to DMB without any qualification, then it is referring to all types of DMB. Unless a specific shareability domain is defined, a DMB applies to the Common Shareability Domain.

All properties that apply to DMB also apply to the corresponding DSB.

Context synchronization instruction

A *Context synchronization instruction* is one of the following:

- An ISB instruction.
- An instruction that generates a synchronous exception.
- An exception return instruction.
- A DCPS or DRPS instruction.

E2.3.2 Dependency definitions

**Dependency through registers**

A *Dependency through registers* from a first effect \( E_1 \) to a second effect \( E_2 \) exists within a PE if and only if at least one of the following applies:

- \( E_1 \) is a register write effect \( W_1 \) which has not been generated by a Store Exclusive, \( E_2 \) is a register read effect \( R_2 \) and \( R_2 \) Reads-from-register \( W_1 \).
• E₁ and E₂ have been generated by the same instruction and E₁ is before E₂ in the Intrinsic order of that instruction.
• There is a Dependency through registers from E₁ to a third effect E₃, and there is a Dependency through registers from E₃ to E₂.

Address dependency

An Address dependency from a memory read effect R₁ to a Memory effect RW₂ exists if and only if there is a Dependency through registers from R₁ to a Register effect E₃ generated by RW₂, and E₃ affects the address part of RW₂, and either:
• RW₂ is a memory write effect W₂.
• RW₂ is a memory read effect R₂ and there is no Branching effect D₄ such that there is a Dependency through registers from R₁ to D₄ and from D₄ to R₂.

Data dependency

A Data dependency from a memory read effect R₁ to a memory write effect W₂ exists if and only if there is a Dependency through registers from R₁ to a Register effect E₃ generated by W₂, and E₃ affects the data part of W₂.

Control dependency

A Control dependency from a memory read effect R₁ to a subsequent Memory effect RW₂ exists if and only if either:
• There is a Dependency through registers from R₁ to a Branching effect B₃ and B₃ is in program order before RW₂.
• There is a Dependency through registers from R₁ to the determination of a synchronous exception on an instruction generating an effect RW₃, and RW₂ appears in program order after RW₃.

Note

This notion is under review. Arm’s intent is that a branch instruction between a read and a write, where the branch condition is dependent on the read, will provide order, regardless of whether the branch is taken. This applies only to branch instructions and not to conditional selection or other conditional data processing instructions. A formal definition of this change will be issued soon as an erratum to the Architecture Reference Manual.

Pick Basic dependency

A Pick Basic dependency from a read Register effect or read Memory effect R₁ to a Register effect or Memory effect E₂ exists if and only if one of the following applies:
• There is a Dependency through registers from R₁ to E₂.
• There is an Intrinsic Control dependency from R₁ to E₂.
• There is a Pick Basic dependency from R₁ to an Effect E₃ and there is a Pick Basic dependency from E₃ to E₂.

Pick Address dependency

A Pick Address dependency from a read Register effect or read Memory effect R₁ to a read or write Memory effect RW₂ exists if and only if all of the following apply:
• There is a Pick Basic dependency from R₁ to a Register effect E₃.
• There is an Intrinsic data dependency from the Register effect E₃ to RW₂.
• The Register effect E₃ affects the address of the location accessed by RW₂.

Pick Data dependency

A Pick Data dependency from a read Register effect or read Memory effect R₁ to a write Memory effect W₂ exists if and only if all of the following apply:
• There is a Pick Basic dependency from R₁ to a Register effect E₃.
• There is an Intrinsic data dependency from the Register effect \( E_3 \) to \( RW_2 \).
• The Register effect \( E_3 \) affects the value written by \( W_2 \).

### Pick Control dependency

A Pick Control dependency from a read Register effect or read Memory effect \( R_1 \) to an effect \( E_2 \) exists if and only if all of the following apply:

- There is a Pick Basic dependency from \( R_1 \) to a Branching effect \( BR_3 \).
- The Branching effect \( BR_3 \) is in program order before \( E_2 \).

### Pick dependency

A Pick dependency from a read Register effect or read Memory effect \( R_1 \) to an effect \( E_2 \) exists if and only if one of the following applies:

- There is a Pick Basic dependency from \( R_1 \) to \( E_2 \).
- There is a Pick Address dependency from \( R_1 \) to \( E_2 \).
- There is a Pick Data dependency from \( R_1 \) to \( E_2 \).
- There is a Pick Control dependency from \( R_1 \) to \( E_2 \).

### E2.3.3 Ordering relations

#### Dependency-ordered-before

A dependency creates externally-visible order between a memory read effect and another Memory effect generated by the same Observer. A memory read effect \( R_1 \) is Dependency-ordered-before a memory read or write effect \( RW_2 \) from the same Observer if and only if \( R_1 \) appears in program order before \( RW_2 \) and any of the following cases apply:

- There is an Address dependency or a Data dependency from \( R_1 \) to \( RW_2 \).
- \( RW_2 \) is a memory write effect \( W_2 \) and there is a Control dependency from \( R_1 \) to \( W_2 \).
- \( RW_2 \) is a memory read effect \( R_2 \) generated by an instruction appearing in program order after an instruction that generates a Context synchronization event \( E_3 \), and there is a Control dependency from \( R_1 \) to \( E_3 \).
- \( RW_2 \) is a memory write effect \( W_2 \) appearing in program order after a memory read or write effect \( RW_3 \) and there is an Address dependency from \( R_1 \) to \( RW_3 \).
- \( RW_2 \) is a Local read successor \( R_2 \) of a memory write effect \( W_3 \) and there is an Address dependency or a Data dependency from \( R_1 \) to \( W_3 \).

#### Pick-ordered-before

A read Register effect or read Memory effect \( R_1 \) is Pick-ordered-before a read or write Memory effect \( RW_2 \) from the same Observer if and only if \( R_1 \) appears in program order before \( RW_2 \) and any of the following cases apply:

- \( RW_2 \) is a write Memory effect \( W_2 \) and there is a Pick dependency from \( R_1 \) to \( W_2 \).
- \( RW_2 \) is a read Memory effect \( R_2 \) generated by an instruction appearing in program order after an instruction that generates a Context synchronization event \( E_3 \), and there is a Pick Control dependency from \( R_1 \) to \( E_3 \).
- \( RW_2 \) is a Memory effect generated by an instruction appearing in program order after an instruction that generates a Context synchronization event \( E_3 \), there is a Pick Address dependency from \( R_1 \) to an effect \( E_4 \) and \( E_4 \) is in program order before \( E_3 \).
- \( RW_2 \) is a write Memory effect \( W_2 \), there is a Pick Address dependency from \( R_1 \) to a read or write Memory effect \( RW_3 \) and \( W_2 \) is program-order-after \( RW_3 \).
Atomic-ordered-before

Load-Exclusive and Store-Exclusive instructions provide some ordering guarantees, even in the absence of dependencies. A read or a write Memory effect RW1 is Atomic-ordered-before a read or a write Memory effect RW2 from the same Observer if and only if RW1 appears in program order before RW2 and either of the following cases apply:

- RW1 is a read Memory effect R1 and RW2 is a write Memory effect W2 such that R1 and W2 are generated by an atomic instruction or a successful Load-Exclusive/Store-Exclusive instruction pair to the same Location.
- RW1 is a read Memory effect R1 generated by an atomic instruction (resp. a Load-Exclusive instruction) and RW2 is a read Memory effect R2 generated by an instruction with Acquire or AcquirePC semantics such that R2 is a Local read successor of the write Memory effect W3 generated by the same atomic instruction as R1 (resp. the successful Store-Exclusive instruction paired with the Load-Exclusive instruction that generated R1).

For more information, see Synchronization and semaphores on page E2-7211.

Barrier-ordered-before

Barrier instructions order prior Memory effects before subsequent Memory effects generated by the same Observer. A memory read or write effect RW1 is Barrier-ordered-before a memory read or write effect RW2 from the same Observer if and only if RW1 appears in program order before RW2 and any of the following cases apply:

- RW1 appears in program order before a DMB FULL that appears in program order before RW2.
- RW1 is a memory write effect W1 and is generated by an atomic instruction with both Acquire and Release semantics.
- RW1 is a memory write effect W1 generated by an instruction with Release semantics and RW2 is a memory read effect R2 generated by an instruction with Acquire semantics.
- RW1 is a memory read effect R1 and appears in program order before a DMB LD that appears in program order before RW2.
- RW1 is a memory read effect R1 and is generated by an instruction with Acquire or AcquirePC semantics.
- RW1 is a memory write effect W1 and RW2 is a memory write effect W2 appearing in program order before a DMB ST that appears in program order before W2.
- RW2 is a memory write effect W2 and is generated by an instruction with Release semantics.

Locally-ordered-before

Dependencies, Local write successor, load/store-exclusive, atomic and barrier instructions can be composed within an Observer to create externally-visible order. A memory read or write effect RW1 is Locally-ordered-before a memory read or write effect RW2 from the same Observer if and only if any of the following apply:

- RW1 is a memory write effect W1 and RW2 is a memory write effect W2 that is equal to or generated by the same instruction as a Local write successor of RW1.
- RW1 is Pick-ordered-before RW2.
- RW1 is Dependency-ordered-before RW2.
- RW1 is Atomic-ordered-before RW2.
- RW1 is Barrier-ordered-before RW2.
- RW1 is Locally-ordered-before a memory read or write effect that is Locally-ordered-before RW2.

Pick-locally-ordered-before

A read or a write Memory effect RW1 is Pick-locally-ordered-before a read or a write Memory effect RW2 from the same Observer if and only if all of the following apply:

- There is a Pick Basic dependency from RW1 to an effect E3.
E2.3 Definition of the memory model

E2.3.4 Ordering constraints

The memory model is described as being Other-multi-copy atomic. The definition of Other-multi-copy atomic is as follows:

**Other-multi-copy atomic**

In an Other-multi-copy atomic system, it is required that a memory write effect from an Observer, if observed by a different Observer, is then observed by all other Observers that access the Location coherently. It is, however, permitted for an Observer to observe its own writes prior to making them visible to other observers in the system.

The Other-multi-copy atomic property of the memory model is enforced by placing constraints on the possible executions of a program. Those executions that meet the constraints given by the ordering model are said to be Architecturally well-formed. An implementation that is executing a program is only permitted to exhibit behavior consistent with an Architecturally well-formed execution.

**Architecturally well-formed**

An Architecturally well-formed execution must satisfy both the Internal visibility requirement and any of the three alternative External ordering constraints.

E2.3.5 Internal visibility requirement

For a memory read or write effect RW₁ that appears in program order before a memory read or write effect RW₂ to the same Location, the **Internal visibility requirement** requires that exactly one of the following statements is true:

- RW₂ is a write Memory effect W₂.
- E₃ is Locally-ordered-before W₂.

E2.3.6 External ordering constraints

The memory model offers the following three alternative representations of the **External ordering constraint**:

- **External visibility requirement**.
- **External completion requirement**.
- **External global completion requirement**.

An Architecturally well-formed execution must satisfy both the Internal visibility requirement and one of the three alternative representations in the External ordering constraints.

**External visibility requirement**

Ordered-before

An arbitrary pair of Memory effects is ordered if it can be linked by a chain of ordered accesses consistent with external observation. A memory read or write effect RW₁ is Ordered-before a memory read or write effect RW₂ if and only if any of the following cases apply:

- RW₁ is Observed-by a memory read or write effect RW₃ that is generated by the same instruction as RW₂.
• RW₁ is Locally-ordered-before RW₂.
• RW₁ is Pick-locally-ordered-before RW₂.
• RW₁ is Ordered-before a memory read or write effect that is Ordered-before RW₂.

For a memory read or write effect RW₁ from an Observer that is Ordered-before a memory read or write effect RW₂ from a different Observer, the External visibility requirement requires that RW₂ is not Observed-by RW₁. This means that an Architecturally well-formed execution must not exhibit a cycle in the Ordered-before relation.

Informally, if a Memory effect M₁ from an Observer is Ordered-before another Memory effect M₂ from a different Observer, then M₁ will be seen to occur before M₂ by all Observers in the system.

Completes-before order

The Completes-before order is a total order that corresponds to the order in which Memory effects complete within the system. The following effects constitute a single entry in the Completes-before order:
• Writes from the same instruction.
• Reads from the same instruction which read from external writes.
• Reads from the same instruction which read from the same internal write.

All other reads constitute distinct entries in the Completes-before order.

Completes-before

A memory read or write effect RW₁ Completes-before a memory read or write effect RW₂ if and only if RW₁ appears in the Completes-before order before RW₂.

Deriving Reads-from and Coherence order from the Completes-before order

The Completes-before order can be used to resolve the Reads-from and Coherence order relations for every memory access in the system as follows:
• For a memory read effect R₁ of a memory location by an Observer, then:
  — If there is a memory write effect W₂ to the same Location from the same Observer and all of the following are true:
    — W₂ appears in program order before R₁.
    — R₁ Completes-before W₂.
    — There are no writes to the Location appearing in program order between W₂ and R₁ then R₁ Reads-from W₂.
  — Otherwise, R₁ Reads-from its closest preceding write in the Completes-before order to the same Location. If no such write exists, then R₁ Reads-from the initial value of the memory location.
• The Coherence order of writes to a memory location is the order in which those writes appear in the Completes-before order. The final value of each memory location is therefore determined by the final write to each Location in the Completes-before order. If no such write exists for a given Location, the final value is the initial value of that Location.

External completion requirement

A memory read or write effect RW₁ Globally-completes-before a memory read or write effect RW₂ if and only if any of the following statements are true:
• RW₁ is Locally-ordered-before RW₂.
• RW₁ is Pick-locally-ordered-before RW₂.
• RW₁ is a memory read effect R₁ and RW₂ is a memory read effect R₂ and R₁ is Single-copy-atomic-ordered-before R₂.
**Globally-completes-before order**

The *Globally-completes-before order* is a total order that corresponds to the order in which Memory effects globally-complete within the system. The following effects constitute a single entry in the Globally-completes-before order:

- Writes from the same instruction.
- Reads from the same instruction which read from external writes.
- Reads from the same instruction which read from the same internal write.

All other reads constitute distinct entries in the Globally-completes-before order.

**Globally-completes-before**

A memory read or write effect RW₁ *globally-completes-before* a memory read or write effect RW₂ if and only if RW₁ appears in the Globally-completes-before order before RW₂.

**Deriving Reads-from and Coherence order from the Globally-completes-before order**

The Globally-completes-before order can be used to resolve the Reads-from and Coherence order relations for every memory access in the system as follows:

- A memory read effect R₁ of a memory location by an Observer Reads-from its closest preceding write in the Globally-completes-before order to the same Location. If no such write exists, then R₁ Reads-from the initial value of the memory location.
- The Coherence order of writes to a memory location is the order in which those writes appear in the Globally-completes-before order. The final value of each memory location is therefore determined by the final write to each Location in the Globally-completes-before order. If no such write exists for a given Location, the final value is the initial value of that Location.

**External global completion requirement**

The *External global completion requirement* requires that a memory read or write effect RW₁ Globally-completes-before a memory read or write effect RW₂ if and only if any of the following statements are true:

- RW₁ is Locally-ordered-before RW₂ and either:
  - RW₁ is a memory write effect.
  - RW₁ is a memory read effect R₁ and either:
    - R₁ is not a Local read successor of a memory write effect.
    - R₁ is a Local read successor of a memory write effect that is Locally-ordered-before RW₂.
    - R₁ is a Local read successor of a memory write effect that is Pick-locally-ordered-before RW₂.
- RW₁ is a memory read effect R₁ and RW₂ is a memory read effect R₂ and R₁ is Single-copy-atomic-ordered-before R₂.

**E2.3.7 Completion and endpoint ordering**

Interaction between Observers in a system is not restricted to communication via shared variables in coherent memory. For example, an Observer could configure an interrupt controller to raise an interrupt on another Observer as a form of message passing. These interactions typically involve an additional agent, which defines the instruction sequence that is required to establish communication links between different Observers. When these forms of interaction are used in conjunction with shared variables, a DSB instruction can be used to enforce ordering between them.

For all memory, the completion rules are defined as:

- A memory read effect R₁ to a Location is complete for a shareability domain when all of the following are true:
  - Any write to the same Location by an Observer within the shareability domain will be Coherence-after R₁.
Any translation table walks associated with R1 are complete for that shareability domain.

- A memory write effect W1 to a Location is complete for a shareability domain when all of the following are true:
  - Any write to the same Location by an Observer within the shareability domain will be Coherence-after W1.
  - Any read to the same Location by an Observer within the shareability domain will either Reads-from W1 or Reads-from a memory write effect that is Coherence-after W1.
  - Any translation table walks associated with the write are complete for that shareability domain.

- A translation table walk is complete for a shareability domain when the memory accesses, including the updates to translation table entries, associated with the translation table walk are complete for that shareability domain, and the TLB is updated.

- A cache or branch predictor maintenance instruction is complete for a shareability domain when the memory effects of the instruction are complete for that shareability domain, and any translation table walks that arise from the instruction are complete for that shareability domain.

- A TLB invalidate instruction is complete when all memory accesses using the TLB entries that have been invalidated are complete.

The completion of any cache, branch predictor, or TLB maintenance instruction includes its completion on all PEs that are affected by both the instruction and the DSB operation that is required to guarantee visibility of the maintenance instruction.

**Note**

These completion rules mean that, for example, a cache maintenance instruction that operates by VA to the PoC completes only after memory at the PoC has been updated.

Additionally, for Device-nGnRnE memory, a read or write of a Location in a Memory-mapped peripheral that exhibits side-effects is complete only when the read or write both:

- Can begin to affect the state of the Memory-mapped peripheral.
- Can trigger all associated side-effects, whether they affect other peripheral devices, PEs, or memory.

**Note**

This requirement for Device-nGnRnE memory is consistent with the memory access having reached the peripheral endpoint.

### Peripherals

This section defines a Memory-mapped peripheral and the total order of reads and write to a peripheral which is defined as the Peripheral coherence order:

**Memory-mapped peripheral**

A Memory-mapped peripheral occupies a memory region of IMPLEMENTATION DEFINED size and can be accessed using load and store instructions. Memory effects to a Memory-mapped peripheral can have side-effects, such as causing the peripheral to perform an action. Values that are read from addresses within a Memory-mapped peripheral might not correspond to the last data value written to those addresses. As such, Memory effects to a Memory-mapped peripheral might not appear in the Reads-from or Coherence order relations.

**Peripheral coherence order**

The Peripheral coherence order of a Memory-mapped peripheral is a total order on all reads and writes to that peripheral.

**Note**

The Peripheral coherence order for a Memory-mapped peripheral signifies the order in which accesses arrive at the endpoint.
For a memory read or write effect RW₁ and a memory read or write effect RW₂ to the same peripheral, then RW₁ will appear in the Peripheral coherence order for the peripheral before RW₂ if either of the following cases apply:

- RW₁ and RW₂ are accesses using Non-cacheable or Device attributes and RW₁ is Ordered-before RW₂.
- RW₁ and RW₂ are accesses using Device-nGnRE or Device-nGnRnE attributes and RW₁ appears in program order before RW₂.

**Out-of-band-ordered-before**

A memory read or write effect RW₁ is Out-of-band-ordered-before a memory read or write effect RW₂ if and only if either of the following cases apply:

- RW₁ appears in program order before a DSB instruction that begins an IMPLEMENTATION DEFINED instruction sequence indirectly leading to the generation of RW₂.
- RW₁ is Ordered-before a memory read or write effect RW₃ and RW₃ is Out-of-band-ordered-before RW₂.

If a Memory effect M₁ is Out-of-band-ordered-before a memory read or write effect M₂, then M₁ is seen to occur before M₂ by all Observers.

### E2.3.8 Ordering of instruction fetches

For two memory locations A and B, if A has been written to with an updated value and been made coherent with the instruction fetches of the shareability domain before B has been written to with an updated value by an observer in the same shareability domain, then where, for an observer in the shareability domain, an instruction read from B appears in program order before an instruction fetched from A, if the instruction read from B contains the updated value of B then the instruction read from A appearing later in program order will contain the updated value of A.

A write has been made coherent with an instruction fetch of a shareability domain when:

- **CTR.{DIC, IDC} == {0, 0}**
  - The location written to has been cleaned to the Point of unification (PoU) from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

- **CTR.{DIC, IDC} == {1, 0}**
  - The location written to has been cleaned to the Point of unification (PoU) from the data cache, and that clean is complete for the shareability domain.

- **CTR.{DIC, IDC} == {0, 1}**
  - The write is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

- **CTR.{DIC, IDC} == {1, 1}**
  - The write is complete for the shareability domain.

--- **Note** ---

Microarchitecturally, this means that these situations cannot both be true in an implementation:

- After delays in fetching from memory, the instruction queue can have entries written into it out of order.
- For an implementation:
  - When **CTR.DIC == 0**, if there is an outstanding entry in the instruction queue, then later entries in the instruction queue are not impacted by the ICDMAU instructions of a different core.
 — When CTR.DIC == 1, if there is a write to the location that is held in the queue when there is an outstanding entry in the instruction queue for an older entry, then the instruction queue does not have entries invalidated from it.

### E2.3.9 Restrictions on the effects of speculation

The Arm architecture places certain restrictions on the effects of speculation. These are:

- Each load from a location using a particular VA after an exception return that is a Context synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception exit.
- Each load from a location using a particular VA after an exception entry that is a Context synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception entry.
- Any load from a location using a particular VA before an exception entry that is a Context synchronization event will not speculatively read data from a store to the same location using the same VA after the exception entry.
- Any load from a location using a particular VA before an exception return that is a Context synchronization event will not speculatively read an entry from earlier in the coherence order for the location being loaded from than the entry generated by the latest store to that location using the same VA before the exception exit.
- When data is loaded under speculation with a Translation fault, it cannot be used to form an address or generate condition codes to be used by other instructions in the speculative sequence.
- When data is loaded under speculation from a location without a Translation fault for the translation regime being speculated in, the data cannot be used to form an address or generate condition codes to be used by other instructions in the speculative sequence.
- When data is loaded as a result of speculative accesses made after TLBI + DSB + ERET using a translation that was invalidated by the TLBI, the data cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence. The execution timing of any other instructions in the speculative sequence is not a function of the data loaded.
- Changes to System registers must not occur speculatively in a way that can affect a speculative memory access that can cause a change to the micro-architectural state.
- Changes to Special-purpose registers can occur speculatively.
- Execute-never controls apply to speculative instruction fetching. See Access permissions for instruction execution on page G5-9196.
- When writing new instructions to memory, there is no requirement for an SB instruction to prevent speculative execution of the old code. See Instruction cache maintenance instructions on page K13-11773.

--- **Note**

The prohibition of using data loaded under speculation with faults to form addresses, condition codes or SVE predicate values does not prohibit the use of value predicted data from such locations for such purposes, so long as the training of the data value prediction was from the hardware defined context that is using the prediction. A consequence of this is that training of value prediction cannot be based on data loaded under speculation with a translation or Permission fault.

--- **Speculative Store Bypass Safe (SSBS)**

When FEAT_SSBS is implemented, CPSR.SSBS is a control that can be set by software to indicate whether hardware is use, in a manner that is potentially speculatively exploitable, a speculative value in a register that has been loaded from memory using a load instruction that speculatively read an entry for the location being loaded from, where the entry that is speculatively read is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

A speculative value in a register is used in a potentially speculatively exploitable manner if it is used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence or if the execution timing of any other instructions in the speculative sequence is a function of the data loaded under speculation.
When the value of CPSR.SSBS is 0, hardware is not permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

When the value of CPSR.SSBS is 1, hardware is permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

--- **Note** ---

- If speculation is permitted, then cache timing side channels can lead to addresses being derived using reads of address values that have been speculatively loaded from memory to a register.

- Software written for architectures from Armv8.0 to Armv8.4 will set SPSR.SSBS to 0. This means that CPSR.SSBS will not set, so hardware will not be permitted to use speculative loads with outstanding memory disambiguation issues for any subsequent speculative memory accesses if there is any possibility of those subsequent memory accesses creating a cache timing side channel.

---

**Further restrictions on the effects of speculation from Armv8.5**

From Armv8.5, there are some further restrictions on the effects of speculation in addition to those in Armv8.0:

- Data loaded under speculation with a Permission or Domain fault cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence. The execution timing of any other instructions in the speculative sequence is not a function of the data loaded under speculation.

- Any System register read under speculation to a register that is not architecturally accessible from the current Exception level cannot be used to form an address or to generate condition codes to be used by other instructions in the speculative sequence.

--- **Note** ---

As the effects of speculation are not architecturally visible, this restriction level requires that the effect of any speculation cannot give rise to side channels that will leak the values of memory locations, System registers, or Special-purpose registers to a level of privilege that would otherwise not be able to determine those values.

---

- For all execution prediction resources that predict address or register values, speculative execution at one hardware defined context should be separated in a hard-to-determine manner from control by a different hardware defined context. In the case of this definition, the hardware defined context is determined by:
  
  — The Exception level.
  — The Security state.
  — When executing at EL1 and when EL2 is enabled in the current Security state, the VMID.
  — When executing at EL0 and using the EL1&0 translation regime, the ASID and, when EL2 is enabled in the current Security state, the VMID.
  — When executing at EL0 and using the EL2&0 translation regime, the ASID.

--- **Note** ---

— The definition of “hard-to-determine manner” is left open to implementations. Examples could include the complete separation of prediction resources, or the isolation of the predictions using a cryptographic or pseudo-random mechanism to separate each context.

— The architecture does not require that prediction resources that simply predict the direction of a branch are separated in this way.

---

- Changes to System registers must not occur speculatively in a way that can affect a speculative memory access that can cause a change to the micro-architectural state.

- Changes to Special-purpose registers can occur speculatively.
E2.3.10 Memory barriers

The Arm architecture is a weakly ordered memory architecture that supports out of order completion. Memory barrier is the general term applied to an instruction, or sequence of instructions, that forces synchronization events by a PE with respect to retiring load/store instructions. The memory barriers defined by the architecture provide a range of functionality, including:

- Ordering of load/store instructions.
- Completion of load/store instructions.
- Context synchronization.

The following subsections describe the memory barrier instructions:

- **Instruction Synchronization Barrier (ISB).**
- **Data Memory Barrier (DMB) on page E2-7182.**
- **Data Synchronization Barrier (DSB) on page E2-7182.**
- **Speculation Barrier (SB) on page E2-7183.**
- **Consumption of Speculative Data Barrier (CSDB) on page E2-7183.**
- **Speculative Store Bypass Barrier (SSBB) on page E2-7183.**
- **Physical Speculative Store Bypass Barrier (PSSBB) on page E2-7184.**
- **Trace Synchronization Barrier (TSB CSYNC) on page E2-7184.**
- **Shareability and access limitations on the data barrier operations on page E2-7185.**
- **Load-Acquire, Store-Release on page E2-7186.**

--- Note ---

Depending on the required synchronization, a program might use memory barriers on their own, or it might use them in conjunction with cache maintenance and memory management instructions that in general are available only when software execution is at EL1 or higher.

The DMB and DSB memory barriers affect reads and writes to the memory system generated by load/store instructions and data or unified cache maintenance instructions being executed by the PE.

AArch32 state also supports the legacy barrier instructions CP15DMB, CP15DSB, and CP15ISB. These instructions are executed as MCRs using the appropriate encoding, and are accessible from EL0. However, for performance reasons Arm deprecates any use of these operations, and strongly recommends that software uses the DMB, DSB, and ISB instructions described in this section instead. Optionally, an implementation can support a CP15BEN control that supervisory software can use to disable use of these instructions, meaning the corresponding MCR encodings are UNDEFINED. When the CP15BEN control is supported, setting one of the following CP15BEN fields to 0 makes execution of CP15DMB, CP15DSB, and CP15ISB UNDEFINED:

- SCTLR_EL1.CP15BEN, for execution of these instructions at EL0 using AArch32 when EL1 is using AArch64.
- SCTLR.CP15BEN, for execution of these instructions at EL0 or EL1 when EL1 is using AArch32.
- HSCTLR.CP15BEN, for execution of these instructions at EL2 when EL2 is using AArch32.

**Instruction Synchronization Barrier (ISB)**

An ISB instruction ensures that all instructions that come after the ISB instruction in program order are fetched from the cache or memory after the ISB instruction has completed. Using an ISB ensures that the effects of context-changing operations executed before the ISB are visible to the instructions fetched after the ISB instruction. Examples of context-changing operations that require the insertion of an ISB instruction to ensure the effects of the operation are visible to instructions fetched after the ISB instruction are:

- Completed cache and TLB maintenance instructions.
- Changes to System registers.

Any context-changing operations appearing in program order after the ISB instruction take effect only after the ISB has been executed.

The pseudocode function for the operation of an ISB is `InstructionSynchronizationBarrier()`.
See also Memory barriers on page G4-9144.

**Data Memory Barrier (DMB)**

The DMB instruction is a memory barrier instruction that ensures the relative order of memory accesses before the barrier with memory accesses after the barrier. The DMB instruction does not ensure the completion of any of the memory accesses for which it ensures relative order.

The full definition of the DMB instruction is covered formally in the Definition of the memory model on page E2-7168 and this introduction to the DMB instruction is not intended to contradict that section.

The basic principle of a DMB instruction is to introduce order between memory accesses that are specified to be affected by the DMB options supplied as arguments to the DMB instruction. The DMB instruction ensures that all affected memory accesses by the PE executing the DMB instruction that appear in program order before the DMB instruction and those which originate from a different PE, to the extent required by the DMB options, which have been Observed-by the PE before the DMB instruction is executed, are Observed-by each PE, to the extent required by the DMB options, before any affected memory accesses that appear in program order after the DMB instruction are Observed-by that PE.

The use of a DMB instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The pseudocode function for the operation of a DMB instruction is DataMemoryBarrier().

**Data Synchronization Barrier (DSB)**

A DSB instruction is a memory barrier that ensures that memory accesses that occur before the DSB instruction have completed before the completion of the DSB instruction. In doing this, it acts as a stronger barrier than a DMB and all ordering that is created by a DMB with specific options is also generated by a DSB with the same options.

Execution of a DSB at EL2 ensures that any memory accesses caused by speculative translation table walks from the Non-secure PL1&0 translation regime have been observed.

For more information, see Use of out-of-context translation regimes on page G5-9152.

A DSB executed by a PE, PEE, completes when all of the following apply:

- All explicit memory effects of the required access types appearing in program order before the DSB are complete for the set of observers in the required shareability domain.
- If the required access types of the DSB is reads and writes, the following instructions issued by PEE before the DSB are complete for the required shareability domain:
  - All cache maintenance instructions.
  - All AArch32 TLB maintenance instructions.
  - All PSB CYNC instructions.
- If the required access types of the DSB is reads and writes, completion of a DSB instruction executed by PEE ensures that:
  - All previous TLB maintenance operations generated by AArch32 TLB maintenance instructions executed at EL1 by PEE when HCRX_EL2.FnXS is 1 are finished for all PEs in the shareability domain of the DSB instruction.
  - All previous TLB maintenance operations generated by AArch32 TLB maintenance instructions are finished for all PEs in the shareability domain of the DSB instruction.

In addition, no instruction that appears in program order after the DSB instruction can alter any state of the system or perform any part of its functionality until the DSB completes, other than:

- Being fetched from memory and decoded.
- Reading the general-purpose, SIMD and floating-point, Special-purpose, or System registers that are directly or indirectly read without causing side-effects.
- If FEAT_ETS is not implemented, having any virtual addresses of loads and stores translated.

The pseudocode function for the operation of a DSB is DataSynchronizationBarrier().

See also Memory barrier instructions on page G4-9141 and Memory barriers on page G4-9144.
Speculation Barrier (SB)

An SB is a memory barrier that prevents speculative execution of instructions until after the barrier has completed when those instructions could be observed through side-channels.

Until the barrier completes, the speculative execution of any instruction appearing later in the program order than the barrier:

- Cannot be performed to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be performed when predicting that an instruction that could generate an exception does not generate an exception.

Speculative execution of an SB instruction:

- Cannot be as a result of control flow speculation.
- Cannot be as a result of data value speculation.
- Can be as a result of predicting that an instruction that could generate an exception does not generate an exception.

An SB instruction can complete when:

- It is known that it is not speculative.
- All the predicted data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

Note

The SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched, so long as the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after an uncompleted SB instruction.

Consumption of Speculative Data Barrier (CSDB)

The CSDB instruction is a memory barrier instruction that controls speculative execution and data value prediction. This includes:

- Data value predictions of any instructions.
- PSTATE.\{N,Z,C,V\} predictions of any instructions other than conditional branch instructions appearing in program order before the CSDB that have not been architecturally resolved.
- Predictions of SVE prediction state for any SVE instructions.

For purposes of the definition of CSDB, PSTATE.\{N,Z,C,V\} is not considered a data value. This definition permits:

- Control flow speculation before and after the CSDB.
- Speculative execution of conditional data processing instructions after the CSDB, unless they use the results of data value or PSTATE.\{N,Z,C,V\} predictions of instructions appearing in program order before the CSDB that have not been architecturally resolved.

Speculative Store Bypass Barrier (SSBB)

The SSBB instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the SSBB instruction.
• When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  — The store is to the same location as the load.
  — The store uses the same virtual address as the load.
  — The store appears in program order after the SSBB instruction.

**Physical Speculative Store Bypass Barrier (PSSBB)**

The PSSBB instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same physical address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

• When a load to a location appears in program order after the PSSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  — The store is to the same location as the load.
  — The store appears in program order before the PSSBB instruction.

• When a load to a location appears in program order before the PSSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  — The store is to the same location as the load.
  — The store appears in program order after the PSSBB instruction.

--- Note ---
The effect of this barrier applies to accesses to the same location even if they are accessed with different virtual addresses and from different Exception levels.

--- Trace Synchronization Barrier (TSB CSYNC) ---

The TSB CSYNC instruction is a barrier instruction that preserves the relative order of accesses to System registers due to trace operations and other accesses to the same registers.

A trace operation is an operation of the trace unit generating trace for an instruction when FEAT_TRF is implemented and enabled.

A TSB CSYNC is not required to execute in program order with respect to other instructions. This includes being reordered with respect to other trace instructions. One or more Context synchronization events are required to ensure that TSB CSYNC is executed in the necessary order.

If trace is generated between a Context synchronization event and a TSB CSYNC operation, these trace operations may be reordered with respect to the TSB CSYNC operation, and therefore may not be synchronized.

The following situations are synchronized using a TSB CSYNC:

• A direct write B to a System register is ordered after an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all of the following are true:
  — A is executed in program order before a Context synchronization event C.
  — C is in program order before a TSB CSYNC operation T.
  — B is executed in program order after T.

• A direct read B of a System register is ordered after an indirect write to the same register by a trace operation of a traced instruction A if all the following are true:
  — A is executed in program order before a Context synchronization event C1.
  — C1 is in program order before TSB CSYNC operation T.
  — T is executed in program order before a second Context synchronization event C2.
  — B is executed in program order after C2.
A TSB CSYNC is not needed when a direct write B to a System register is ordered before an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all the following are true:
• A is executed in program order after a Context synchronization event C.
• B is executed in program order before C.

The pseudocode function for the operation of a TSB CSYNC is TraceSynchronizationBarrier().

**Shareability and access limitations on the data barrier operations**

The DMB and DSB instructions can each take an optional limitation argument that specifies:
• The shareability domain over which the instruction must operate. This is one of:
  — Full system.
  — Outer Shareable.
  — Inner Shareable.
  — Non-shareable.

Full system applies to all the observers in the system and, as such, encompasses the Inner and Outer Shareable domains of the processor.

—— Note ———
The distinction between Full system and Outer Shareable is applicable only for Normal Non-cacheable memory accesses and Device memory accesses.

——— Note ————
This form of a DMB or DSB instruction can be described as a load-load/store barrier.

For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) on page E2-7182 or Data Synchronization Barrier (DSB) on page E2-7182.

Table E2-1 on page E2-7185 shows how these options are encoded in the <option> field of the instruction.

**Table E2-1 Encoding of the DMB and DSB <option> parameter**

<table>
<thead>
<tr>
<th>Accesses</th>
<th>Shareability domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before the barrier</td>
<td>After the barrier</td>
</tr>
<tr>
<td>Reads and writes</td>
<td>Reads and writes</td>
</tr>
<tr>
<td>Writes</td>
<td>Writes</td>
</tr>
<tr>
<td>Reads</td>
<td>Reads and writes</td>
</tr>
</tbody>
</table>

If no <option> is specified then the instruction operates for read and write accesses, over the full system, meaning the operation is the same as for the SY option. See the instruction descriptions for more information:
• DMB on page F5-7555.
• DSB on page F5-7558.

——— Note ————
ISB also supports an optional limitation argument that can contain only one value that corresponds to full system operation, see ISB on page F5-7578.
Load-Acquire, Store-Release

The architecture provides a set of instructions with Acquire semantics for loads, and Release semantics for stores.

The full definition of the Load-Acquire instruction is covered formally in the Definition of the memory model on page E2-7168 and this introduction to the Load-Acquire instruction is not intended to contradict that section.

The basic principle of a Load-Acquire instruction is to introduce order between the memory access generated by the Load-Acquire instruction and the memory accesses appearing in program order after the Load-Acquire instruction, such that the memory access generated by the Load-Acquire instruction is Observed-by each PE, to the extent that PE is required to observe the access coherently, before any of the memory accesses appearing in program order after the Load-Acquire instruction are Observed-by that PE, to the extent that the PE is required to observe the accesses coherently.

The use of a Load-Acquire instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The full definition of the Store-Release instruction is covered formally in the Definition of the memory model on page E2-7168 and this introduction to the Store-Release instruction is not intended to contradict that section.

The basic principle of a Store-Release instruction is to introduce order between the memory accesses generated by the PE executing the Store-Release instruction, together with those which originate from a different PE, to the extent that the PE executing the Store-Release instruction is required to observe them coherently, Observed-by the PE before executing the Store-release.

The use of a Store-Release instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

In addition, the use of a Load-Acquire or a Store-Release instruction on accesses to a Memory-mapped peripheral introduces order between the Memory effects of the instructions that access that peripheral, as described in the definition of Peripheral coherence order.

Load-Acquire and Store-Release, other than LDAEXD and STLEXD, access only a single data element. This access is single-copy atomic. The address of the data object must be aligned to the size of the data element being accessed, otherwise the access generates an Alignment fault.

LDAEXD and STLEXD access two data elements. The address supplied to the instructions must be doubleword-aligned, otherwise the access generates an Alignment fault.

A Store-Release Exclusive instruction has the release semantics only if the store is successful.

Note

- Each Load-Acquire Exclusive and Store-Release Exclusive instruction is essentially a variant of the equivalent Load-Exclusive or Store-Exclusive instruction. All usage restrictions and single-copy atomicity properties:
  - That apply to the Load-Exclusive instructions also apply to the Load-Acquire Exclusive instructions.
  - That apply to the Store-Exclusive instructions also apply to the Store-Release Exclusive instructions.

- The Load-Acquire/Store-Release instructions can remove the requirement to use the explicit DMB memory barrier instruction.

Table E2-2 on page E2-7186 summarizes the Load-Acquire/Store-release instructions.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Load-Acquire</th>
<th>Store-Release</th>
<th>Load-Acquire Exclusive</th>
<th>Store-Release Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit word</td>
<td>LDA</td>
<td>STL</td>
<td>LDAEX</td>
<td>STLEX</td>
</tr>
<tr>
<td>16-bit halfword</td>
<td>LDAH</td>
<td>STLH</td>
<td>LDAEXH</td>
<td>STLEXH</td>
</tr>
<tr>
<td>8-bit byte</td>
<td>LDAB</td>
<td>STLB</td>
<td>LDAEXB</td>
<td>STLEXB</td>
</tr>
<tr>
<td>64-bit doubleword</td>
<td>-</td>
<td>-</td>
<td>LDAEXD</td>
<td>STLEXD</td>
</tr>
</tbody>
</table>
E2.4 Ordering of translation table walks

If FEAT_ETS is implemented, and a memory access RW₁ is Ordered-before a second memory access RW₂, then RW₁ is also Ordered-before any translation table walk generated by RW₂ that generates any of the following:
   • A Translation fault.
   • An Address size fault.
   • An Access flag fault.
**E2.5 Caches and memory hierarchy**

The implementation of a memory system depends heavily on the microarchitecture and therefore many details of the memory system are IMPLEMENTATION DEFINED. The architecture defines the application level interface to the memory system, including a hierarchical memory system with multiple levels of cache. This section describes an application level view of this system. It contains the subsections:

- Introduction to caches.
- Memory hierarchy.
- Implication of caches for the application programmer on page E2-7189.
- Preloading caches on page E2-7191.

### E2.5.1 Introduction to caches

A cache is a block of high-speed memory that contains a number of entries, each consisting of:

- Main memory address information, commonly known as a tag.
- The associated data.

Caches increase the average speed of a memory access and take account of two principles of locality:

**Spatial locality**

An access to one location is likely to be followed by accesses to adjacent locations. Examples of this principle are:

- Sequential instruction execution.
- Accessing a data structure.

**Temporal locality**

An access to an area of memory is likely to be repeated in a short time period. An example of this principle is the execution of a software loop.

To minimize the quantity of control information stored, the spatial locality property groups several locations together under the same tag. This logical block is commonly known as a cache line. When data is loaded into a cache, access times for subsequent loads and stores are reduced, resulting in overall performance benefits. An access to information already in a cache is known as a cache hit, and other accesses are called cache misses.

Normally, caches are self-managing, with the updates occurring automatically. Whenever the PE accesses a cacheable memory location, the cache is checked. If the access is a cache hit, the access occurs in the cache. Otherwise, the access is made to memory. Typically, when making this access, a cache location is allocated and the cache line loaded from memory. The architecture permits different cache topologies and access policies, provided they comply with the memory coherency model described in this manual.

Caches introduce a number of potential problems, mainly because:

- Memory accesses can occur at times other than when the programmer would expect them.
- A data item can be held in multiple physical locations.

### E2.5.2 Memory hierarchy

Typically memory close to a PE has very low latency, but is limited in size and expensive to implement. Further from the PE it is common to implement larger blocks of memory but these have increased latency. To optimize overall performance, a memory system can include multiple levels of cache in a hierarchical memory system that exploits this trade-off between size and latency. Figure E2-1 on page E2-7189 shows an example of such a system in a system that supports virtual addressing.
The AArch32 Application Level Memory Model

E2.5 Caches and memory hierarchy

Figure E2-1 Multiple levels of cache in a memory hierarchy

--- Note ---
In this manual, in a hierarchical memory system, Level 1 refers to the level closest to the PE, as shown in Figure E2-1.

--- Note ---
Instructions and data can be held in separate caches or in a unified cache. A cache hierarchy can have one or more levels of separate instruction and data caches, with one or more unified caches located at the levels closest to the main memory. Memory coherency for cache topologies can be defined using the conceptual points Point of Unification (PoU) and Point of Coherency (PoC). For more information, including the definitions of PoU and PoC, see About cache maintenance in AArch32 state on page G4-9119.

--- Note ---
FEAT_DPB adds architectural support for an additional conceptual point, Point of Persistence, but this support is provided only in AArch64 state. For more information, see About cache maintenance in AArch64 state on page D7-5050.

The Cacheability and Shareability memory attributes

Cacheability and Shareability are two attributes that describe the memory hierarchy in a multiprocessing system:

**Cacheability** This term defines whether memory locations are allowed to be allocated into a cache or not. Cacheability is defined independently for Inner and Outer Cacheability locations.

**Shareability** This term defines whether memory locations are shareable between different agents in a system. Marking a memory location as shareable for a particular domain requires hardware to ensure that the location is coherent for all agents in that domain. Shareability is defined independently for Inner and Outer Shareability domains.

For more information about the Cacheability and Shareability attributes, see Memory types and attributes on page E2-7198.

**E2.5.3 Implication of caches for the application programmer**

In normal operation, the caches are largely invisible to the application programmer. However they can become visible when there is a breakdown in the coherency of the caches. Such a breakdown can occur:

- When memory locations are updated by other agents in the system that do not use hardware management of coherency.
- When memory updates made from the application software must be made visible to other agents in the system, without the use of hardware management of coherency.
For example:

- In the absence of hardware management of coherency of DMA accesses, in a system with a DMA controller that reads memory locations that are held in the data cache of a PE, a breakdown of coherency occurs when the PE has written new data in the data cache, but the DMA controller reads the old data held in memory.

- In a Harvard cache implementation, where there are separate instruction and data caches, a breakdown of coherency occurs when new instruction data has been written into the data cache, but the instruction cache still contains the old instruction data.

### Data coherency issues

Software can ensure the data coherency of caches in the following ways:

- By not using the caches in situations where coherency issues can arise. This can be achieved by:
  - Using Non-cacheable or, in some cases, Write-Through Cacheable memory.
  - Not enabling caches in the system.
- By using system calls to functions using cache maintenance instructions that execute at a higher Exception level.
- By using hardware coherency mechanisms to ensure the coherency of data accesses to memory for cacheable locations by observers within the different shareability domains, see Non-shareable Normal memory on page E2-7200 and Shareable, Inner Shareable, and Outer Shareable Normal memory on page E2-7199.

---

**Note**

The performance of these hardware coherency mechanisms is highly implementation-specific. In some implementations the mechanism suppresses the ability to cache shareable locations. In other implementations, cache coherency hardware can hold data in caches while managing coherency between observers within the shareability domains.

### Synchronization and coherency issues between data and instruction accesses

How far ahead of the current point of execution instructions are fetched from is IMPLEMENTATION DEFINED. Such prefetching can be either a fixed or a dynamically varying number of instructions, and can follow any or all possible future execution paths. For all types of memory:

- The PE might have fetched the instructions from memory at any time since the last Context synchronization event on that PE.
- Any instructions fetched in this way might be executed multiple times, if this is required by the execution of the program, without being re-fetched from memory.

The Arm architecture does not require the hardware to ensure coherency between instruction caches and memory, even for locations of shared memory.

If software requires coherency between instruction execution and memory, it must manage this coherency using Context synchronization events and cache maintenance instructions. These can be accessed only from an Exception level that is higher than EL0, and therefore require a system call, see Exception-generating and exception-handling instructions on page F2-7275. The following code sequence can be used for this purpose:

```assembly
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Rt> containing a new 32-bit instruction, to be held in Cacheable space at a location pointed to by Rn. Use STRH in the first line instead of STR for a 16-bit instruction.
STR Rt, [Rn]
DCCMVAU Rn ; Clean data cache by MVA to point of unification (PoU)
DSB ; Ensure visibility of the data cleaned from cache
ICIMVAU Rn ; Invalidate instruction cache by MVA to PoU
BPIMVA Rn ; Invalidate branch predictor by MVA to PoU
DSB ; Ensure completion of the invalidations
ISB ; Synchronize the fetched instruction stream
```
A write has been made coherent with an instruction fetch of a shareability domain when:

\( \text{CTR.\{DIC, IDC\}} = \{0, 0\} \)

The location written to has been cleaned to the Point of unification (PoU) from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

\( \text{CTR.\{DIC, IDC\}} = \{1, 0\} \)

The location written to has been cleaned to the Point of unification (PoU) from the data cache, and that clean is complete for the shareability domain.

\( \text{CTR.\{DIC, IDC\}} = \{0, 1\} \)

The write is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

\( \text{CTR.\{DIC, IDC\}} = \{1, 1\} \)

The write is complete for the shareability domain.

--- Note ---

- For accesses that are Non-cacheable or Write-Through, the clean data cache instruction is not required. For accesses that are Non-cacheable, the invalidate instruction cache is not required, because in AArch32 state these accesses are not permitted to be held in an instruction cache.

- This code can be used when the thread of execution modifying the code is the same thread of execution that is executing the code. The architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization. See Concurrent modification and execution of instructions on page E2-7166.

## E2.5.4 Preloading caches

The Arm architecture provides the memory system hints PLD (Preload Data), PLDW (Preload Data With Intent To Write) and PLI (Preload Instruction) that software can use to communicate the expected use of memory locations to the hardware. The memory system can respond by taking actions that are expected to speed up the memory accesses if they occur. The effect of these memory system hints is IMPLEMENTATION DEFINED. Typically, implementations use this information to bring data or instruction locations into caches.

The Preload instructions are hints, and so implementations can treat them as \texttt{NOP}s without affecting the functional behavior of the device. The instructions cannot generate synchronous Data Abort exceptions, but the resulting memory system operations might, under exceptional circumstances, generate an asynchronous External abort, which is reported using an SError interrupt and taken using an asynchronous Data Abort exception. For more information, see Data Abort exception on page G1-8975.

A PLD, PLDW, or PLI instruction can only cause allocation to software-visible caching structures such caches or TLBs for memory locations that can be accessed, according to the permissions defined by the current translation regime or a translation regime for a higher Exception level in the current Security state, by any of:

- Reads.
- Writes.
- Instruction fetches.

A PLD, PLDW, or PLI instruction can access any memory location in Normal memory that can be accessed, according to the permissions defined by the current translation regime or a translation regime for a higher Exception level in the current Security state, by any of:

- Reads.
- Writes.
- Instruction fetches.
Note

In each case, the entire list applies to each of PLD, PLDW, and PLI.

A PLD, PLDW, or PLI instruction is guaranteed not to access any type of Device memory.

A PLI instruction must not perform any access that cannot be performed by a speculative instruction fetch by the processor. Therefore in a VMSA implementation, if all associated MMUs are disabled, a PLI instruction cannot access any memory location that cannot be accessed by instruction fetches.

The pseudocode enumeration `PrefetchHint` defines the prefetch hint types.

The `Hint_Prefetch()` pseudocode function signals to the memory system that memory accesses of the type hint to or from the specified address are likely to occur in the near future. The memory system might take some action to speed up the memory accesses when they do occur, such as preloading the specified address into one or more caches as indicated by the innermost cache level target and non-temporal hint stream.

For more information on PLD, PLI, and PLDW, see:

- **PLD, PLDW (immediate)** on page F5-7776.
- **PLD (literal)** on page F5-7779.
- **PLD, PLDW (register)** on page F5-7781.
- **PLI (immediate, literal)** on page F5-7784.
- **PLI (register)** on page F5-7787.
E2.6 Alignment support

This section describes alignment support. It contains the following subsections:

- Instruction alignment.
- Unaligned data access.
- Cases where unaligned accesses are CONSTRAINED UNPREDICTABLE on page E2-7194.
- Unaligned data access restrictions on page E2-7194.
- Generation of Alignment faults by load/store multiple accesses to Device memory on page E2-7194.

For more information about Alignment faults, see Alignment faults on page G5-9247.

E2.6.1 Instruction alignment

A32 instructions are word-aligned.

T32 instructions are halfword-aligned.

E2.6.2 Unaligned data access

An A-profile implementation must support unaligned data accesses to Normal memory by some load and store instructions. As Table E2-3 on page E2-7193 shows, software can control whether a misaligned access to Normal memory by one of these instructions causes an Alignment fault Data Abort exception:

- By setting SCTLR.A, for unaligned accesses from any mode other than Hyp mode.
- By setting HSCTLR.A, for unaligned accesses from Hyp mode.

Table E2-3 Alignment requirements of load/store instructions

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Alignment check</th>
<th>Result if check fails when:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SCTLR.A or HSCTLR.A is 0</td>
</tr>
<tr>
<td>LDRB, LDREXB, LDRBT, LDRSB, LDRSBT, STRB, STREXB, STRBT, TBB</td>
<td>None</td>
<td>-</td>
</tr>
<tr>
<td>LDRH, LDRHT, LDRSH, LDRSHT, STRH, STRHT, TBB</td>
<td>Halfword</td>
<td>Unaligned access</td>
</tr>
<tr>
<td>LDREXH, STREXH, LDAH, STLH, LDASEXH, STLEXH</td>
<td>Halfword</td>
<td>Alignment fault</td>
</tr>
<tr>
<td>LDR, LDRT, STR, STRT</td>
<td>Word</td>
<td>Unaligned access</td>
</tr>
<tr>
<td>PUSH, encodings T3 and A2 only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POP, encodings T3 and A2 only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDEX, STREX, LDA, STL, LDAEX, STLEX</td>
<td>Word</td>
<td>Alignment fault</td>
</tr>
<tr>
<td>LDREXD, STREXD, LDAEXD, STLEXD</td>
<td>Doubleword</td>
<td>Alignment fault</td>
</tr>
<tr>
<td>All forms of LDM and STM, LDRD, RFE, SRS, STRD</td>
<td>Word</td>
<td>Alignment fault</td>
</tr>
<tr>
<td>LDC, STC</td>
<td>Word</td>
<td>Alignment fault</td>
</tr>
<tr>
<td>VLDM, VPOR, VPUSH, VSTM</td>
<td>Word</td>
<td>Alignment fault</td>
</tr>
<tr>
<td>VLDR, VSTR - single-precision scalar and double-precision scalar</td>
<td>Word</td>
<td>Alignment fault</td>
</tr>
<tr>
<td>VLDR, VSTR - half-precision scalar</td>
<td>Halfword</td>
<td>Alignment fault</td>
</tr>
<tr>
<td>VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4, all with standard alignment</td>
<td>Element size</td>
<td>Unaligned access</td>
</tr>
<tr>
<td>VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4, all with :&lt;align&gt; specifieda</td>
<td>As specified by :&lt;align&gt;</td>
<td>Alignment fault</td>
</tr>
</tbody>
</table>

a As specified by :<align>
E2.6.3  Cases where unaligned accesses are CONSTRAINED UNPREDICTABLE

Any load instruction that is not faulted by the alignment restrictions shown in Table E2-3 on page E2-7193 and that loads the PC has CONSTRAINED UNPREDICTABLE behavior if the address it loads from is not word-aligned, see Loads and Stores to unaligned locations on page K1-11564. This overrules any permitted load/store behavior shown in Table E2-3 on page E2-7193.

E2.6.4  Unaligned data access restrictions

The following points apply to unaligned data accesses:

- Accesses are not guaranteed to be single-copy atomic except at the byte access level, see Atomicity in the Arm architecture on page E2-7164.
- Unaligned accesses typically take a number of additional cycles to complete compared to a naturally-aligned access.
- An operation that performs an unaligned access can abort on any memory access that it makes, and can abort on more than one access. This means that an unaligned access that occurs across a page boundary can generate an abort on either side of the boundary.

E2.6.5  Generation of Alignment faults by load/store multiple accesses to Device memory

When FEAT_LSMAOC is implemented and the value of the applicable nTLSMD field is 0, any memory access by an AArch32 Load Multiple or Store Multiple instruction to an address that the stage 1 translation assigns as Device-nGRE, Device-nGnRE, or Device-nGnRnE generates an Alignment fault.

The applicable nTLSMD field is the field in the SCTLR_EL1, SCTLR_EL2, HSCTLR, or SCTLR register that applies to the Exception level and Security state at which the LDM or STM instruction is executed.
E2.7 Endian support

*General description of endianness in the Arm architecture* describes the relationship between endianness and memory addressing in the Arm architecture.

The following subsections then describe the endianness schemes supported by the architecture:
- *Instruction endianness.*
- *Data endianness on page E2-7196.*
- *Endianness of memory-mapped peripherals on page E2-7197.*

**E2.7.1 General description of endianness in the Arm architecture**

This section only describes memory addressing and the effects of endianness for data elements up to doubleword of 64 bits. However, this description can be extended to apply to larger data elements.

For an address A, *Figure E2-2* shows, for big-endian and little-endian memory systems, the relationship between:
- The doubleword at address A.
- The words at addresses A and A+4.
- The halfwords at addresses A, A+2, A+4, and A+6.
- The bytes at addresses A, A+1, A+2, A+3, A+4, A+5, A+6, and A+7.

The terms in *Figure E2-2* have the following definitions:
- **MSByte**  Most significant byte.
- **LSByte**  Least significant byte.

In this figure, *Byte, A+1* is an abbreviation for *Byte at address A+1.*

**E2.7.2 Instruction endianness**

The mapping of instruction memory is always little-endian.
E2.7.3 Data endianness

The size of the data value that is loaded or stored is the size that is used for the purpose of endian conversion for floating-point, Advanced SIMD, and general-purpose register loads and stores.

Table E2-4 on page E2-7196 shows the element sizes of all the load/store instructions, for all instruction sets.

Table E2-4 Element size of load/store instructions

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Element size</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDRB, LDXRB, LDRBT, LDRSB, LDRSBT, STRB, STREXB, STRBT, TBB</td>
<td>Byte</td>
</tr>
<tr>
<td>LDRH, LDXRH, LDRHT, LDRSH, LDRSHT, STRH, STREXH, STRHT, TBH</td>
<td>Halfword</td>
</tr>
<tr>
<td>LDR, LDRT, LDREX, STR, STRT, STREX</td>
<td>Word</td>
</tr>
<tr>
<td>LDRD, LDREXD, STRD, STREXD</td>
<td>Word</td>
</tr>
<tr>
<td>All forms of LDM, PUSH, POP, RFE, SRS, all forms of STM</td>
<td>Word</td>
</tr>
<tr>
<td>LDC, STC</td>
<td>Word</td>
</tr>
<tr>
<td>Forms of VLDM, VLDR, VPUSH, VPOP, VSTM, VSTR that transfer 32-bit Si registers</td>
<td>Word</td>
</tr>
<tr>
<td>Forms of VLDM, VLDR, VPUSH, VPOP, VSTM, VSTR that transfer 64-bit Di registers</td>
<td>Doubleword</td>
</tr>
<tr>
<td>VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4</td>
<td>Element size of the Advanced SIMD access</td>
</tr>
</tbody>
</table>

CPSR.E determines the data endianness.

The data size used for endianness conversions:

• Is the size of the data value that is loaded or stored for Advanced SIMD and floating-point register and general-purpose register loads and stores.

• Is the size of the data element that is loaded or stored for Advanced SIMD element and data structure loads and stores. For more information, see Endianness in Advanced SIMD on page E2-7197.

Instructions to reverse bytes in registers

An application or device driver might have to interface to memory-mapped peripheral registers or shared memory structures that are not the same endianness as the internal data structures. Similarly, the endianness of the operating system might not match that of the peripheral registers or shared memory. In these cases, the PE requires an efficient method to transform explicitly the endianness of the data.

Table E2-5 on page E2-7196 shows the instructions that provide this functionality in the A32 and T32 instruction sets.

Table E2-5 Byte reversal instructions

<table>
<thead>
<tr>
<th>Function</th>
<th>T32/A32 instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reverse bytes in whole register</td>
<td>REV</td>
<td>For use with general purpose registers</td>
</tr>
<tr>
<td>Reverse bytes in 16-bit halfwords</td>
<td>REV16</td>
<td>For use with general purpose registers</td>
</tr>
<tr>
<td>Reverse bytes in halfword and sign-extend</td>
<td>REVSH</td>
<td>For use with general purpose registers</td>
</tr>
<tr>
<td>Reverse elements in doublewords, vector</td>
<td>VREV64</td>
<td>For use with registers in the SIMD and floating-point register file</td>
</tr>
<tr>
<td>Reverse elements in words, vector</td>
<td>VREV32</td>
<td>For use with registers in the SIMD and floating-point register file</td>
</tr>
<tr>
<td>Reverse elements in halfwords, vector</td>
<td>VREV16</td>
<td>For use with registers in the SIMD and floating-point register file</td>
</tr>
</tbody>
</table>
Endianness in Advanced SIMD

Advanced SIMD element load/store instructions transfer vectors of elements between memory and the SIMD and floating-point register file. An instruction specifies both the length of the transfer and the size of the data elements being transferred. This information is used by the PE to load and store data correctly in both big-endian and little-endian systems.

Consider, for example, the A32 or T32 instruction:

\[
\text{VLD1.16 } \{D0\}, [R1]
\]

This loads a 64-bit register with four 16-bit values. The four elements appear in the register in array order, with the lowest indexed element fetched from the lowest address. The order of bytes in the elements depends on the endianness configuration, as shown in Figure E2-3. Therefore, the order of the elements in the registers is the same regardless of the endianness configuration.

For information about the alignment of Advanced SIMD instructions, see Alignment support on page E2-7193.

The `BigEndian()` pseudocode function determines the current endianness of the data.

The `BigEndianReverse()` pseudocode function reverses the endianness of a bitstring.

The `BigEndian()` and `BigEndianReverse()` functions are defined in Chapter J1 Armv8 Pseudocode.

E2.7.4 Endianness of memory-mapped peripherals

All memory-mapped peripherals defined in the Arm architecture must be little-endian.

Peripherals to which this requirement applies include:

- Memory-mapped register interfaces to a debugger, or to a cross-trigger interface, see Chapter H8 About the External Debug Registers.
- The memory-mapped register interface to the system level implementation of the Generic Timer, see Chapter I2 System Level Implementation of the Generic Timer.
- A memory-mapped register interface to the Performance Monitors, see Chapter I3 Recommended External Interface to the Performance Monitors.
- A memory-mapped register interface to the Activity Monitors, see Chapter I4 Recommended External Interface to the Activity Monitors.
- Memory-mapped register interfaces to an Arm Generic Interface Controller, see the ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0.
- The memory-mapped register interface to an Arm trace component. See, for example, the ARM® Embedded Trace Macrocell Architecture Specification, ETMv4.
E2.8 Memory types and attributes

The ordering of accesses for addresses in memory, referred to as the memory order model, is defined by the memory attributes. The following sections describe this model:

- Normal memory.
- Device memory on page E2-7202.
- Memory access restrictions on page E2-7207.

E2.8.1 Normal memory

The Normal memory type attribute applies to most memory in a system. It indicates that the hardware is permitted by the architecture to perform Speculative data read accesses to these locations, regardless of the access permissions for these locations.

The Normal memory type has the following properties:

- A write to a memory location with the Normal attribute completes in finite time.
- Writes to a memory location with the Normal memory type that is either Non-cacheable or Write-Through cacheable for both the Inner and Outer cacheability must reach the endpoint for that location in the memory system in finite time. Two writes to the same location, where at least one is using the Normal memory type, might be merged before they reach the endpoint unless there is an ordered-before relationship between the two writes.
- Unaligned memory accesses can access Normal memory if the system is configured to generate such accesses.
- There is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register load/store instructions. See Multi-register loads and stores that access Normal memory on page E2-7202.

Note

- The Normal memory attribute is appropriate for locations of memory that are idempotent, meaning that they exhibit all of the following properties:
  - Read accesses can be repeated with no side-effects.
  - Repeated read accesses return the last value written to the resource being read.
  - Read accesses can fetch additional memory locations with no side-effects.
  - Write accesses can be repeated with no side-effects if the contents of the location accessed are unchanged between the repeated writes or as the result of an exception, as described in this section.
  - Unaligned accesses can be supported.
  - Accesses can be merged before accessing the target memory system.
- Normal memory allows speculative reads and may be affected by intermediate buffering and forwarding of data. If non-idempotent memory locations are mapped as Normal memory, the following may occur:
  - Memory accesses return UNKNOWN values.
  - UNPREDICTABLE effects on memory-mapped peripherals.
- An instruction that generates a sequence of accesses as described in Atomicity in the Arm architecture on page E2-7164 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

The following sections describe the other attributes for Normal memory:

- Shareable Normal memory on page E2-7199.
- Non-shareable Normal memory on page E2-7200.
- Cacheability attributes for Normal memory on page E2-7200.
See also:

- *Multi-register loads and stores that access Normal memory* on page E2-7202.
- *Atomicity in the Arm architecture* on page E2-7164.
- *Memory barriers* on page E2-7181. For accesses to Normal memory, a `DMB` instruction is required to ensure the required ordering.
- *Concurrent modification and execution of instructions* on page E2-7166.

**Shareable Normal memory**

A Normal memory location has a Shareability attribute that is defined as one of:

- Inner Shareable.
- Outer Shareable.
- Non-shareable.

The shareability attributes define the data coherency requirements of the location, which hardware must enforce. They do not affect the coherency requirements of instruction fetches, see *Synchronization and coherency issues between data and instruction accesses* on page E2-7190.

---

**Note**

- System designers can use the Shareability attribute to specify the locations in Normal memory for which coherency must be maintained. However, software developers must not assume that specifying a memory location as Non-shareable permits software to make assumptions about the incoherence of the location between different PEs in a shared memory system. Such assumptions are not portable between different multiprocessing implementations that might use the Shareability attribute. Any multiprocessing implementation might implement caches that are shared, inherently, between different PEs.

- This architecture assumes that all PEs that use the same operating system or hypervisor are in the same Inner Shareable shareability domain.

---

**Shareable, Inner Shareable, and Outer Shareable Normal memory**

The Arm architecture abstracts the system as a series of Inner and Outer Shareability domains.

Each Inner Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Inner Shareable attribute made by any member of that set.

Each Outer Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Outer Shareable attribute made by any member of that set.

The following properties also hold:

- Each observer is only a member of a single Inner Shareability domain.
- Each observer is only a member of a single Outer Shareability domain.
- All observers in an Inner Shareability domain are always members of the same Outer Shareability domain. This means that an Inner Shareability domain is a subset of an Outer Shareability domain, although it is not required to be a proper subset.

---

**Note**

- Because all data accesses to Non-cacheable locations are data coherent to all observers, Non-cacheable locations are always treated as Outer Shareable.

- The Inner Shareable domain is expected to be the set of PEs controlled by a single hypervisor or operating system.

---

The details of the use of the Shareability attributes are system-specific. *Example E2-1 on page E2-7200* shows how they might be used.
Example E2-1 Use of shareability attributes

In an implementation, a particular subsystem with two clusters of PEs has the requirement that:

- In each cluster, the data caches or unified caches of the PEs in the cluster are transparent for all data accesses to memory locations with the Inner Shareable attribute.
- However, between the two clusters, the caches:
  - Are not required to be coherent for data accesses that have only the Inner Shareable attribute.
  - Are coherent for data accesses that have the Outer Shareable attribute.

In this system, each cluster is in a different Shareability domain for the Inner Shareable attribute, but all components of the subsystem are in the same Shareability domain for the Outer Shareable attribute.

A system might implement two such subsystems. If the data caches or unified caches of one subsystem are not transparent to the accesses from the other subsystem, this system has two Outer Shareable Shareability domains.

Having two levels of shareability means system designers can reduce the performance and power overhead for shared memory locations that do not need to be part of the Outer Shareable Shareability domain.

For Shareable Normal memory, the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer in the same Shareability domain.

**Non-shareable Normal memory**

For Normal memory locations, the Non-shareable attribute identifies Normal memory that is likely to be accessed only by a single PE.

A location in Normal memory with the Non-shareable attribute does not require the hardware to make data accesses by different observers coherent, unless the memory is Non-cacheable. For a Non-shareable location, if other observers share the memory system, software must use cache maintenance instructions, if the presence of caches might lead to coherency issues when communicating between the observers. This cache maintenance requirement is in addition to the barrier operations that are required to ensure memory ordering.

For Non-shareable Normal memory, it is IMPLEMENTATION DEFINED whether the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer.

**Cacheability attributes for Normal memory**

In addition to being Outer Shareable, Inner Shareable or Non-shareable, each region of Normal memory is assigned a Cacheability attribute that is one of:

- Write-Through Cacheable.
- Write-Back Cacheable.
- Non-cacheable.

Also, for Write-Through Cacheable and Write-Back Cacheable Normal memory regions:

- A region might be assigned cache allocation hints for read and write accesses.
- It is IMPLEMENTATION DEFINED whether the cache allocation hints can have an additional attribute of Transient or Non-transient.

For more information, see *Cacheability, cache allocation hints, and cache transient hints on page G4-9116.*

A memory location can be marked as having different cacheability attributes, for example when using aliases in a VA to PA mapping:

- If the attributes differ only in the cache allocation hint this does not affect the behavior of accesses to that location.
- For other cases, see *Mismatched memory attributes on page E2-7208.*
The cacheability attributes provide a mechanism of coherency control with observers that lie outside the Shareability domain of a region of memory. In some cases, the use of Write-Through Cacheable or Non-cacheable regions of memory might provide a better mechanism for controlling coherency than the use of hardware coherency mechanisms or the use of cache maintenance routines. To this end, the architecture requires the following properties for Non-cacheable or Write-Through Cacheable memory:

• A completed write to a memory location that is Non-cacheable or Write-Through Cacheable for a level of cache made by an observer accessing the memory system inside the level of cache is visible to all observers accessing the memory system outside the level of cache without the need of explicit cache maintenance.

• A completed write to a memory location that is Non-cacheable for a level of cache made by an observer accessing the memory system outside the level of cache is visible to all observers accessing the memory system inside the level of cache without the need of explicit cache maintenance.

--- Note ---
Implementations can use the cache allocation hints to indicate a probable performance benefit of caching. For example, a programmer might know that a piece of memory is not going to be accessed again and would be better treated as Non-cacheable. The distinction between memory regions with attributes that differ only in the cache allocation hints exists only as a hint for performance.

For Normal memory, the Arm architecture provides cacheability attributes that are defined independently for each of two conceptual levels of cache, the inner and the outer cache. The relationship between these conceptual levels of cache and the implemented physical levels of cache is IMPLEMENTATION DEFINED, and can differ from the boundaries between the Inner and Outer Shareability domains. However:

• Inner refers to the innermost caches, meaning the caches that are closest to the PE, and always includes the lowest level of cache.

• No cache that is controlled by the Inner cacheability attributes can lie outside a cache that is controlled by the Outer cacheability attributes.

• An implementation might not have any outer cache.

Example E2-2, Example E2-3, and Example E2-4 on page E2-7202 describe the possible ways of implementing a system with three levels of cache, level 1 (L1) to level 3 (L3).

--- Note ---
• L1 cache is the level closest to the PE, see Memory hierarchy on page E2-7188.

• When managing coherency, system designs must consider both the inner and outer cacheability attributes, as well as the Shareability attributes. This is because hardware might have to manage the coherency of caches at one conceptual level, even when another conceptual level has the Non-cacheable attribute.

Example E2-2 Implementation with two inner and one outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

• The Inner cacheability attribute applied to L1 and L2 cache.

• The Outer cacheability attribute applied to L3 cache.

--- Example E2-3 Implementation with three inner and no outer cache levels ---

Implement the three levels of cache in the system, L1 to L3, with the Inner cacheability attribute applied to L1, L2, and L3 cache. Do not use the Outer cacheability attribute.
Implement the three levels of cache in the system, L1 to L3, with:

- The Inner cacheability attribute applied to L1 cache.
- The Outer cacheability attribute applied to L2 and L3 cache.

Multi-register loads and stores that access Normal memory

For all instructions that load or store more than one general-purpose register from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register from an Exception level the order in which the registers are accessed is not defined by the architecture.

For all instructions that load or store one or more registers from the SIMD and floating-point register file from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load or store instructions.

E2.8.2 Device memory

The Device memory type attributes define memory locations where an access to the location can cause side-effects, or where the value returned for a load can vary depending on the number of loads performed. Typically, the Device memory attributes are used for memory-mapped peripherals and similar locations.

The attributes for Device memory are:

- **Gathering**  Identified as G or nG, see *Gathering* on page E2-7204.
- **Reordering** Identified as R or nR, see *Reordering* on page E2-7205.
- **Early Write Acknowledgement** Identified as E or nE, see *Early Write Acknowledgement* on page E2-7205.

The Device memory types are:

- **Device-nGnRnE**  Device non-Gathering, non-Reordering, No Early Write Acknowledgement. Equivalent to the Strongly-ordered memory type in earlier versions of the architecture.
- **Device-nGnRE**  Device non-Gathering, non-Reordering, Early Write Acknowledgement. Equivalent to the Device memory type in earlier versions of the architecture.
- **Device-nGRE**  Device non-Gathering, Reordering, Early Write Acknowledgement. Armv8 added this memory type to the translation table formats found in earlier versions of the architecture. The use of barriers is required to order accesses to Device-nGRE memory. The Device-nGRE memory type is introduced into the AArch32 translation table formats when the PE is using the Long Descriptor Translation Table format.
- **Device-GRE**  Device Gathering, Reordering, Early Write Acknowledgement. Armv8 added this memory type to the translation table formats found in earlier versions of the architecture. Device-GRE memory has the fewest constraints. It behaves similar to Normal memory, with the restriction that speculative accesses to Device-GRE memory is forbidden. The Device-GRE memory type is introduced into the AArch32 translation table formats when the PE is using the Long Descriptor Translation Table format.

Collectively these are referred to as *any Device memory type*. Going down the list, the memory types are described as getting *weaker*; conversely the going up the list the memory types are described as getting *stronger*.
Note

- As the list of types shows, these additional attributes are hierarchical. For example, a memory location that permits Gathering must also permit Reordering and Early Write Acknowledgement.

- The architecture does not require an implementation to distinguish between each of these memory types and Arm recognizes that not all implementations will do so. The subsection that describes each of the attributes, describes the implementation rules for the attribute.

- Earlier versions of the Arm architecture defined the following memory types:
  - Strongly-ordered memory. This is the equivalent of the Device-nGnRnE memory type.
  - Device memory. This is the equivalent of the Device-nGnRE memory type.

All of these memory types have the following properties:

- Speculative data accesses are not permitted to any memory location with any Device memory attribute. This means that each memory access to any Device memory type must be one that would be generated by a simple sequential execution of the program.
  
  An exception to this applies:
  - Reads generated by the Advanced SIMD and floating-point instructions can access bytes that are not explicitly accessed by the instruction if the bytes accessed are in a 16-byte window, aligned to 16-bytes, that contains at least one byte that is explicitly accessed by the instruction.

  Note

  - An instruction that generates a sequence of accesses as described in Atomicity in the Arm architecture on page E2-7164 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated accesses to a location where the program defines only a single access. For this reason, Arm strongly recommends that no accesses to Device memory are performed from a single instruction that spans the boundary of a translation granule or which in some other way could lead to some of the accesses being aborted.
  - Write speculation that is visible to other observers is prohibited for all memory types.

- A write to a memory location with any Device memory type completes in finite time.

- If a value that would be returned from a read of a memory location with the Device memory type changes without an explicit memory write effect by an observer, this change must also be globally observed for all observers in the system in finite time. Such a change might occur in a peripheral location that holds status information.

- Data accesses to memory locations are coherent for all observers in the system, and correspondingly are treated as being Outer Shareable.

- A memory location with any Device memory attribute cannot be allocated into a cache.

- Writes to a memory location with any Device memory attribute must reach the endpoint for that address in the memory system in finite time. Two writes of Device memory type to the same location might be merged before they reach the endpoint, unless both writes have the non-Gathering attribute or there is an ordered-before relationship between the two writes.

- If a memory location is not capable of supporting unaligned memory accesses, then an unaligned access to that memory location generates an Alignment fault at the first stage of translation that defined the location as being Device.

- If a memory location is capable of supporting unaligned memory accesses, and such a memory location is marked as Device, then it is IMPLEMENTATION DEFINED whether an unaligned access to that memory location generates an Alignment fault at the first stage of translation that defined the location as being Device.

- Hardware does not prevent speculative instruction fetches from a memory location with any of the Device memory attributes unless the memory location is also marked as execute-never for all Exception levels.
This means that to prevent speculative instruction fetches from memory locations with Device memory attributes, any location that is assigned any Device memory type must also be marked as execute-never for all Exception levels. Failure to mark a memory location with any Device memory attribute as execute-never for all Exception levels is a programming error.

In the Non-secure PL1&0 translation regime in systems where HCR.TGE=1 and HCR.DC==0, any Alignment fault that results from the fact that all locations are treated as Device is a fault at the first stage of translation. This causes the value of HSR.ISS.[24] to be 0.

See also Memory access restrictions on page E2-7207.

The memory types for translation table walks cannot be defined as any Device memory type within the TCR. For the Non-secure EL1&0 translation regime, the memory accesses made during a stage 1 translation table walk are subject to a stage 2 translation, and as a result of this second stage of translation, the accesses from the first stage translation table walk might be made to memory locations with any Device memory type. These accesses might be made speculatively. When the value of the HCR.PTW bit is 1, a stage 2 Permission fault is generated if a first stage translation table walk is made to any Device memory type.

For an instruction fetch from a memory location with the Device attribute that is not marked as execute-never for the current Exception level, an implementation can either:

- Treat the instruction fetch as if it were to a memory location with the Normal Non-cacheable attribute.
- Take a Permission fault.

Gathering

In the Device memory attribute:

G Indicates that the location has the Gathering attribute.

nG Indicates that the location does not have the Gathering attribute, meaning it is non-Gathering.

The Gathering attribute determines whether it is permissible for either:

- Multiple memory accesses of the same type, read or write, to the same memory location to be merged into a single transaction.
- Multiple memory accesses of the same type, read or write, to different memory locations to be merged into a single memory transaction on an interconnect.

For memory types with the Gathering attribute, either of these behaviors is permitted, provided that the ordering and coherency rules of the memory location are followed.

For memory types with the non-Gathering attribute, neither of these behaviors is permitted. As a result:

- The number of memory accesses that are made corresponds to the number that would be generated by a simple sequential execution of the program.
- All access occur at their programmed size, except that there is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register load/store instructions. See Multi-register loads and stores that access Device memory on page E2-7206.

Gathering between memory accesses separated by a memory barrier that affects those memory accesses is not permitted.

Gathering between two memory accesses generated by a Load-Acquire/Store-Release is not permitted.

A read from a memory location with the non-Gathering attribute cannot come from a cache or a buffer, but must come from the endpoint for that address in the memory system. Typically this is a peripheral or physical memory.
--- Note ---

• A read from a memory location with the Gathering attribute can come from intermediate buffering of a previous write, provided that:
  — The accesses are not separated by a DMB or DSB barrier that affects both of the accesses.
  — The accesses are not separated by other ordering constructions that require that the accesses are in order. Such a construction might be a combination of Load-Acquire and Store-Release.
  — The accesses are not generated by a Store-Release instruction.

• The Arm architecture defines only programmer visible behavior. Therefore, gathering can be performed if a programmer cannot tell whether gathering has occurred.

An implementation is permitted to perform an access with the Gathering attribute in a manner consistent with the requirements specified by the non-Gathering attribute.

An implementation is not permitted to perform an access with the non-Gathering attribute in a manner consistent with the relaxations allowed by the Gathering attribute.

Reordering

In the Device memory attribute:

R Indicates that the location has the Reordering attribute.

nR Indicates that the location does not have the Reordering attribute, meaning it is non-Reordering.

For all memory types with the non-Reordering attribute, the order of memory accesses arriving at a single peripheral of IMPLEMENTATION DEFINED size, as defined by the peripheral, must be the same order that occurs in a simple sequential execution of the program. That is, the accesses appear in program order. This ordering applies to all accesses using any of the memory types with the non-Reordering attribute. As a result, if there is a mixture of Device-nGnRE and Device-nGnRnE accesses to the same peripheral, these occur in program order. If the memory accesses are not to a peripheral, then this attribute imposes no restrictions.

--- Note ---

• The IMPLEMENTATION DEFINED size of the single peripheral is the same as applies for the ordering guarantee provided by the DMB instruction.

• The Arm architecture defines only programmer visible behavior. Therefore, reordering can be performed if a programmer cannot tell whether reordering has occurred.

An implementation is permitted to perform an access with the Reordering attribute in a manner consistent with the requirements specified by the non-Reordering attribute.

An additional relaxation is that an implementation is not permitted to perform an access with the non-Reordering attribute in a manner consistent with the relaxations allowed by the Reordering attribute.

The non-Reordering attribute does not require any additional ordering, other than that which applies to Normal memory, between:

• Accesses to one physical address with the non-Reordering attribute and accesses to a different physical address with the Reordering attribute.
• Access to one physical address with the non-Reordering attribute and access to a different physical address to Normal memory.
• Accesses with the non-Reordering attribute and accesses to different peripherals of IMPLEMENTATION DEFINED size.

Early Write Acknowledgement

In the Device memory attribute:

E Indicates that the location has the Early Write Acknowledgement attribute.

nE Indicates that the location has the No Early Write Acknowledgement attribute.
If the No Early Write Acknowledgement attribute is assigned for a Device memory location:

- For memory system endpoints where the system architecture in which the PE is operating requires that acknowledgment of a write comes from the endpoint, it is guaranteed that:
  - Only the endpoint of the write access returns a write acknowledgement of the access.
  - No earlier point in the memory system returns a write acknowledgement.

- For memory system endpoints where the system architecture in which the PE is operating does not require that acknowledgment of a write comes from the endpoint, the acknowledgement of a write is not required to come from the endpoint.

--- Note ---

A write with the No Early Write Acknowledgement attribute assigned for a Device memory location is not expected to generate an abort in any situation where the equivalent write to the same location without the No Early Write Acknowledgement attribute assigned does not generate an abort.

---

This means that a DSB barrier instruction, executed by the PE that performed the write to the No Early Write Acknowledgement Location, completes only after the write has reached its endpoint in the memory system if that is required by the system architecture.

Peripherals are an example of system endpoints that require that the acknowledgment of a write comes from the endpoint.

--- Note ---

- The Early Write Acknowledgement attribute affects only where the endpoint acknowledgment is returned from, and does not affect the ordering of arrival at the endpoint between accesses, which is determined by either the Device Reordering attribute, or the use of barriers to create order.

- The areas of the physical memory map for which write acknowledgment from the endpoint is required is outside the scope of the Arm Architecture definition and must be defined as part of the system architecture in which the PE is operating. In particular, regions of memory handled as PCIe configuration writes are expected to support write acknowledgment from the endpoint.

- Arm recognizes that not all areas of a physical memory map will be capable of supporting write acknowledgment from the endpoint. In particular, Arm expects that regions of memory handled as posted writes under PCIe will not support write acknowledgment from the endpoint.

- For maximum software compatibility, Arm strongly recommends that all peripherals for which standard software drivers expect that the use of a DSB instruction will determine that a write has reached its endpoint are placed in areas of the physical memory map that support write acknowledgment from the endpoint.

---

Multi-register loads and stores that access Device memory

For all instructions that load or store more than one general-purpose register there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load and store instructions.

For all instructions that load or store one or more registers from the SIMD and floating-point register file there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load and store instructions.

The architecture permits that the non-speculative execution of an instruction that loads or stores more than one general-purpose or SIMD and floating-point register might result in repeated accesses to the same address.

For an LDRD, STRD, or LDM instruction with a register list that includes the PC, or an STM instruction with a register list that includes the PC, the order in which the registers are accessed is not defined by the architecture.

For a load or store of an Advanced SIMD element or structure, the order in which the registers are accessed is not defined by the architecture.
For a \texttt{VLDM} and \texttt{VSTM} instruction with a register list that does not include the PC, all registers are accessed in ascending address order for accesses to Device memory with the non-Reordering attribute.

For a \texttt{LDM} or \texttt{STM} instruction with a register list that does not include the PC:

- When \texttt{FEAT\_LSMAOC} is not implemented, and when \texttt{FEAT\_LSMAOC} is implemented and the value of the applicable LSMAOE field is 1, all registers are accessed in ascending address order for accesses to Device memory with the non-Reordering attribute.

- When \texttt{FEAT\_LSMAOC} is implemented and the value of the applicable LSMAOE field is 0, no memory accesses are required to be ordered.

- When \texttt{FEAT\_LSMAOC} is implemented and the value of the applicable nTLSMD field is 0, any memory access to an address that the stage 1 translation assigns as Device-nGRE, Device-nGnRE, or Device-nGnRnE generates an Alignment fault.

The applicable LSMAOE or nTLSMD field is the field in the \texttt{SCTLR\_EL1}, \texttt{SCTLR\_EL2}, \texttt{HSCTLR}, or \texttt{SCTLR} register that applies to the Exception level and Security state at which the \texttt{LDM} or \texttt{STM} instruction is executed.

Armv8.2 deprecates software relying on accesses to Device memory made by a single \texttt{LDM} or \texttt{STM} instruction not being reordered.

\section*{E2.8.3 Memory access restrictions}

The following restrictions apply to memory accesses:

- For two explicit memory read effects to any two adjacent bytes in memory, $p$ and $p+1$, generated by the same instruction, and for two explicit memory write effects to any two adjacent bytes in memory, $p$ and $p+1$, that are generated by the same instruction:
  - The bytes $p$ and $p+1$ must have the same memory type and Shareability attributes. otherwise the results are \texttt{CONSTRAINED UNPREDICTABLE}. For example, an \texttt{LDC}, \texttt{LDM}, \texttt{LDRD} \texttt{STC}, \texttt{STM} or \texttt{STRD} instruction, or an unaligned load or store that spans the boundary between Normal memory and Device memory is \texttt{CONSTRAINED UNPREDICTABLE}.
  - Except for possible differences in the cache allocation hints, Arm deprecates having different cacheability attributes for bytes $p$ and $p+1$.

For the permitted \texttt{CONSTRAINED UNPREDICTABLE} behavior, see \textit{Crossing a page boundary with different memory types or Shareability attributes on page K1-11573}.

- If the accesses of an instruction that causes multiple accesses to any type of Device memory cross a 4KB address boundary then behavior is \texttt{CONSTRAINED UNPREDICTABLE} and \textit{Crossing a 4KB boundary with a Device access on page K1-11573} describes the permitted behaviors.

\begin{itemize}
  \item \textbf{Note}
  \begin{itemize}
    \item The boundary referred to is between two Device memory regions that are both of 4KB and aligned to 4KB.
    \item This restriction means it is important that an access to a volatile memory device is not made using a single instruction that crosses a 4KB address boundary.
    \item Arm expects this restriction to constrain the placing of volatile memory devices in the system memory map, rather than expecting a compiler to be aware of the alignment of memory accesses.
  \end{itemize}
\end{itemize}
E2.9 Mismatched memory attributes

Mismatched memory attributes are controlled by privileged software. For more information, see Chapter G5 The AArch32 Virtual Memory System Architecture.

Physical memory Locations are accessed with mismatched attributes if all accesses to the Location do not use a common definition of all of the following attributes of that Location:

• Memory type: Device-nGnRnE, Device-nGnRE, Device-nGRE, Device-GRE or Normal.
• Shareability.
• Cacheability, for the same level of the inner or outer cache, but excluding any cache allocation hints.

Collectively these are referred to as memory attributes.

— Note —

In this document, the terms location and memory location refer to any byte within the current coherency granule and are used interchangeably.

When a memory Location is accessed with mismatched attributes the only software visible effects are one or more of the following:

• Uniprocessor semantics for reads and writes to that memory Location might be lost. This means:
  — A read of the memory Location by one agent might not return the value most recently written to that memory Location by the same agent.
  — Multiple writes to the memory Location by one agent with different memory attributes might not be ordered in program order.
• There might be a loss of coherency when multiple agents attempt to access a memory Location.
• There might be a loss of properties derived from the memory type, as described in later bullets in this section.
• If all Load-Exclusive/Store-Exclusive instructions executed across all threads to access a given memory Location do not use consistent memory attributes, the Exclusives monitor state becomes UNKNOWN.
• Bytes written without the Write-Back cacheable attribute within the same Write-Back granule as bytes written with the Write-Back cacheable attribute might have their values reverted to the old values as a result of cache Write-Back.

The loss of properties associated with mismatched memory type attributes refers only to the following properties of Device memory that are additional to the properties of Normal memory:

• Prohibition of speculative read accesses.
• Prohibition on Gathering.
• Prohibition on Reordering.

For the following situations, when a physical memory Location is accessed with mismatched attributes, a more restrictive set of behaviors applies. The description of each situation also describes the behaviors that apply:

1. Any agent that reads that memory Location using the same common definition of the Memory type, Shareability and Cacheability attributes is guaranteed to access it coherently, to the extent required by that common definition of the memory attributes, only if all the following conditions are met:
   • All writes are performed to an alias of the memory Location that uses the same definition of the Memory type, Shareability and Cacheability attributes.
   • Either:
     — In the Non-secure PL1&0 translation regime, HCR2.MIOCNCE has a value of 0.
     — All aliases with write permission have the Inner Cacheability attribute the same as the Outer Cacheability attribute.
   • Either:
     — All writes are performed to an alias of the memory Location that has Inner Cacheability and Outer Cacheability attributes both as Non-cacheable.
     — All aliases to a memory Location use a definition of the Shareability attributes that encompasses all the agents with permission to access the Location.
2. The possible software-visible effects caused by mismatched attributes for a memory Location are defined more precisely if all of the mismatched attributes define the memory Location as one of:
   • Any Device memory type.
   • Normal Inner Non-cacheable, Outer Non-cacheable memory.

In these cases, the only permitted software-visible effects of the mismatched attributes are one or more of the following:
   • Possible loss of properties derived from the memory type when multiple agents attempt to access the memory Location.
   • Possible reordering of memory transactions to the same memory Location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Any possible loss of coherency or uniprocessor semantics can be avoided by inserting DMB barrier instructions between accesses to the same memory Location that might use different attributes.

Where there is a loss of the uniprocessor semantics, ordering, or coherency, the following approaches can be used:

1. If the mismatched attributes for a Location all assign the same Shareability attribute to a Location that has a cacheable attribute, any loss of uniprocessor semantics, ordering, or coherency within a Shareability domain can be avoided by use of software cache management. To do so, software must use the techniques that are required for the software management of the ordering or coherency of cacheable Locations between agents in different shareability domains. This means:
   • Before writing to a cacheable Location not using the Write-Back attribute, software must invalidate, or clean, a Location from the caches if any agent might have written to the Location with the Write-Back attribute. This avoids the possibility of overwriting the Location with stale data.
   • After writing to a cacheable Location with the Write-Back attribute, software must clean the Location from the caches, to make the write visible to external memory.
   • Before reading the Location with a cacheable attribute, software must invalidate, or clean and invalidate, the Location from the caches, to ensure that any value held in the caches reflects the last value made visible in external memory.
   • Executing a DMB barrier instruction, with scope that applies to the common Shareability of the accesses, between any accesses to the same cacheable Location that use different attributes.

   ——— Note ———
   In AArch32 state, cache maintenance instructions can be accessed only from an Exception level that is higher than EL0, and therefore require a system call. For information on system calls, see Exception-generating and exception-handling instructions on page F2-7275. For information about the AArch32 cache maintenance instructions, see AArch32 cache and branch predictor support on page G4-9113.

   In all cases:
   • Location refers to any byte within the current coherency granule.
   • A clean and invalidate instruction can be used instead of a clean instruction, or instead of an invalidate instruction.
   • In the sequences outlined in this section, all cache maintenance instructions and memory transactions must be completed, or ordered by the use of barrier operations, if they are not naturally ordered by the use of a common address, see Ordering of cache and branch predictor maintenance instructions on page G4-9132.

   ——— Note ———
   With software management of coherency, race conditions can cause loss of data. A race condition occurs when different agents write simultaneously to bytes that are in the same Location, and the invalidate, write, clean sequence of one agent overlaps with the equivalent sequence of another agent. A race condition also occurs if the first operation of either sequence is a clean, rather than an invalidate.

2. If the mismatched attributes for a Location mean that multiple cacheable accesses to the Location might be made with different Shareability attributes, then ordering and coherency are guaranteed only if:
   • Software running on a PE cleans and invalidates a Location from cache before and after each read or write to that Location by that PE.
• A DMB barrier with scope that covers the full Shareability of the accesses is placed between any accesses to the same memory Location that use different attributes.

      --- Note ---
      The Note in rule 1 of this list, about possible race conditions, also applies to this rule.

In addition, if multiple agents attempt to use Load-Exclusive or Store-Exclusive instructions to access a Location, and the accesses from the different agents have different memory attributes associated with the Location, the Exclusives monitor state becomes UNKNOWN.

Arm strongly recommends that software does not use mismatched attributes for aliases of the same Location. An implementation might not optimize the performance of a system that uses mismatched aliases.

      --- Note ---
      As described in Non-cacheable accesses and instruction caches on page D7-5049, a non-cacheable access is permitted to be cached in an instruction cache, despite the fact that a non-cacheable access is not permitted to be cached in a unified cache. Despite this, when cacheable and non-cacheable aliases exist for memory which is executable, these must be treated as mismatched aliases to avoid coherency issues from the data or unified caches that might hold entries that will be brought into the instruction caches.
### E2.10 Synchronization and semaphores

The architecture provides non-blocking synchronization of shared memory, using **synchronization primitives**. The information in this section about memory accesses by synchronization primitives applies to accesses to both Normal and Device memory.

--- **Note**

Use of the synchronization primitives scales for multiprocessing system designs.

Table E2-6 on page E2-7211 shows the synchronization primitives and the associated CLREX instruction.

#### Table E2-6 Synchronization primitives and associated instruction, T32 and A32 instruction sets

<table>
<thead>
<tr>
<th>Transaction size</th>
<th>Additional semantics</th>
<th>Load-Exclusive&lt;sup&gt;a&lt;/sup&gt;</th>
<th>Store-Exclusive&lt;sup&gt;a&lt;/sup&gt;</th>
<th>Other&lt;sup&gt;a&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>-</td>
<td>LDREXB</td>
<td>STREXB</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Load-Acquire/Store-Release</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LDAXB</td>
<td>STLEXB</td>
<td>-</td>
</tr>
<tr>
<td>Halfword</td>
<td>-</td>
<td>LDREXH</td>
<td>STREXH</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Load-Acquire/Store-Release</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LDAXH</td>
<td>STLEXH</td>
<td>-</td>
</tr>
<tr>
<td>Word</td>
<td>-</td>
<td>LDREX</td>
<td>STREX</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Load-Acquire/Store-Release</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LDAX</td>
<td>STLEX</td>
<td>-</td>
</tr>
<tr>
<td>Doubleword</td>
<td>-</td>
<td>LDREXD</td>
<td>STREXD</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Load-Acquire/Store-Release</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LDAXD</td>
<td>STLEXD</td>
<td>-</td>
</tr>
<tr>
<td>None</td>
<td>Clear-Exclusive</td>
<td>-</td>
<td>-</td>
<td>CLREX</td>
</tr>
</tbody>
</table>

<sup>a</sup> Instruction in the T32 and A32 instruction sets.

Except for the row showing the CLREX instruction, the two instructions in a single row are a Load-Exclusive/Store-Exclusive instruction pair. The model for the use of a Load-Exclusive/Store-Exclusive instruction pair accessing a non-aborting memory address \( x \) is:

- The Load-Exclusive instruction reads a value from memory address \( x \).
- The corresponding Store-Exclusive instruction succeeds in writing back to memory address \( x \) only if no other observer, process, or thread has performed a more recent store to address \( x \). The Store-Exclusive instruction returns a status bit that indicates whether the memory write succeeded.

A Load-Exclusive instruction marks a small block of memory for exclusive access. The size of the marked block is **IMPLEMENTATION DEFINED**, see *Marking and the size of the marked memory block* on page E2-7217. A Store-Exclusive instruction to any address in the marked block clears the marking.

--- **Note**

In this section, the term PE includes any observer that can generate a Load-Exclusive or a Store-Exclusive instruction.

The following sections give more information:

- [Exclusive access instructions and Non-shareable memory locations](#)
- [Exclusive access instructions and shareable memory locations](#)
- [Marking and the size of the marked memory block](#)
- [Context switch support](#)
- [Load-Exclusive and Store-Exclusive instruction usage restrictions](#)
- [Use of WFE and SEV instructions by spin-locks](#)
E2.10.1 Exclusive access instructions and Non-shareable memory locations

For memory locations for which the Shareability attribute is Non-shareable, the exclusive access instructions rely on a local Exclusives monitor, or local monitor, that marks any address from which the PE executes a Load-Exclusive instruction. Any non-aborted attempt by the same PE to use a Store-Exclusive instruction to modify any address is guaranteed to clear the marking.

A Load-Exclusive instruction performs a load from memory, and:
- The executing PE marks the physical memory address for exclusive access.
- The local monitor of the executing PE transitions to the Exclusive Access state.

A Store-Exclusive instruction performs a conditional store to memory that depends on the state of the local monitor:

If the local monitor is in the Exclusive Access state

- If the address of the Store-Exclusive instruction is the same as the address that has been marked in the monitor by an earlier Load-Exclusive instruction, then the store occurs. Otherwise, it is IMPLEMENTATION DEFINED whether the store occurs.
- A status value is returned to a register:
  — If the store took place the status value is 0.
  — Otherwise, the status value is 1.
- The local monitor of the executing PE transitions to the Open Access state.

When an Exclusives monitor is in the Exclusive Access state the monitor is set.

If the local monitor is in the Open Access state

- No store takes place.
- A status value of 1 is returned to a register.
- The local monitor remains in the Open Access state.

When an Exclusives monitor is in the Exclusive Access state the monitor is clear.

The Store-Exclusive instruction defines the register to which the status value is returned.

When a PE writes using any instruction other than a Store-Exclusive instruction:

- If the write is to a PA that is not marked as Exclusive Access by its local monitor and that local monitor is in the Exclusive Access state it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.
- If the write is to a PA that is marked as Exclusive Access by its local monitor it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

It is IMPLEMENTATION DEFINED whether a store to a marked PA causes a mark in the local monitor to be cleared if that store is by an observer other than the one that caused the PA to be marked.

Figure E2-4 on page E2-7213 shows the state machine for the local monitor and the effect of each of the operations shown in the figure.
For the local monitor state machine, as shown in Figure E2-4:

- The IMPLEMENTATION DEFINED options for the local monitor are consistent with the local monitor being constructed so that it does not hold any PA, but instead treats any access as matching the address of the previous Load-Exclusive instruction.
- A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive instructions from other PEs.
- The architecture does not require a load instruction, by another PE, that is not a Load-Exclusive instruction, to have any effect on the local monitor.
- It is IMPLEMENTATION DEFINED whether the transition from Exclusive Access to Open Access state occurs when the Store or StoreExcl is from another observer.

Changes to the local monitor state resulting from speculative execution

The architecture permits a local monitor to transition to the Open Access state as a result of speculation, or from some other cause. This is in addition to the transitions to Open Access state caused by the architectural execution of an operation shown in Figure E2-4.

An implementation must ensure that:

- The local monitor cannot be seen to transition to the Exclusive Access state except as a result of the architectural execution of one of the operations shown in Figure E2-4.
- Any transition of the local monitor to the Open Access state not caused by the architectural execution of an operation shown in Figure E2-4 must not indefinitely delay forward progress of execution.

E2.10.2 Exclusive access instructions and shareable memory locations

In the context of this section, a shareable memory location is a memory location that has, or is treated as if it has, a Shareability attribute of Inner Shareable or Outer Shareable.
For shareable memory locations, exclusive access instructions rely on:

- A local monitor for each PE in the system, which marks any address from which the PE executes a Load-Exclusive. The local monitor operates as described in Exclusive access instructions and Non-shareable memory locations on page E2-7212, except that for shareable memory any Store-Exclusive is then subject to checking by the global monitor if it is described in that section as doing at least one of the following:
  - Updating memory.
  - Returning a status value of 0.
  The local monitor can ignore accesses from other PEs in the system.

- A global monitor that marks a PA as exclusive access for a particular PE. This marking is used later to determine whether a Store-Exclusive to that address that has not been failed by the local monitor can occur. Any successful write to the marked block by any other observer in the Shareability domain of the memory location is guaranteed to clear the marking. For each PE in the system, the global monitor:
  - Can hold at least one marked block.
  - Maintains a state machine for each marked block it can hold.

Note
For each PE, the architecture only requires global monitor support for a single marked address. Any situation that might benefit from the use of multiple marked addresses on a single PE is constrained unpredictable, see Load-Exclusive and Store-Exclusive instruction usage restrictions on page E2-7217.

Note
The global monitor can either reside in a block that is part of the hardware on which the PE executes or exist as a secondary monitor at the memory interfaces. The implementation defined aspects of the monitors mean that the global monitor and local monitor can be combined into a single unit, provided that the unit performs the global monitor and local monitor functions defined in this manual.

For shareable memory locations, in some implementations and for some memory types, the properties of the global monitor require functionality outside the PE. Some system implementations might not implement this functionality for all locations of memory. In particular, this can apply to:

- Any type of memory in the system implementation that does not support hardware cache coherency.
- Non-cacheable memory, or memory treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such a system, it is defined by the system:

- Whether the global monitor is implemented.
- If the global monitor is implemented, which address ranges or memory types it monitors.

Note
To support the use of the Load-Exclusive/Store-Exclusive mechanism when address translation is disabled, a system might define at least one location of memory, of at least the size of the translation granularity, in the system memory map to support the global monitor for all PEs within a common Inner Shareable domain. However, this is not an architectural requirement. Therefore, architecturally-compliant software that requires mutual exclusion must not rely on using the Load-Exclusive/Store-Exclusive mechanism, and must instead use a software algorithm such as Lamport’s Bakery algorithm to achieve mutual exclusion.

Because implementations can choose which memory types are treated as Non-cacheable, the only memory types for which it is architecturally guaranteed that a global Exclusives monitor is implemented are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hint and Write allocation hint and not transient.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hint and Write allocation hints and not transient.

If the global monitor is not implemented for an address range or memory type, then performing a Load-Exclusive/Store-Exclusive instruction to such a location has one or more of the following effects:

- The instruction generates an External abort.
The AArch32 Application Level Memory Model

E2.10 Synchronization and semaphores

- The instruction generates an IMPLEMENTATION DEFINED MMU fault. This is reported using the Fault status code of:
  - DFSR.STATUS = 0b110101 when using the Long-descriptor translation table format. The fault can also be reported in the HSR.ISS[5:0] field for exceptions to Hyp mode.
  - DFSR.FS = 0b10101 when using the Short-descriptor translation table format.

If the IMPLEMENTATION DEFINED MMU fault is generated for the Non-secure PL1&0 translation regime then:
- If the fault is generated because of the memory type defined in the first stage of translation, or if the second stage of translation is disabled, then this is a first stage fault and the exception is taken to EL1.
- Otherwise, the fault is a second stage fault and the exception is taken to EL2.

The priority of this fault is IMPLEMENTATION DEFINED.

- The instruction is treated as a NOP.
- The Load-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The Store-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The value held in the result register of the Store-Exclusive instruction becomes UNKNOWN.

In addition, for write transactions generated by non-PE observers that do not implement exclusive accesses or other atomic access mechanisms, the effect that writes have on the global and local monitors used by an Arm PE is IMPLEMENTATION DEFINED. The writes might not clear the global monitors of other PEs for:
- Some address ranges.
- Some memory types.

**Operation of the global Exclusives monitor**

A Load-Exclusive instruction from shareable memory performs a load from memory, and causes the PA of the access to be marked as exclusive access for the requesting PE. This access can also cause the exclusive access mark to be removed from any other PA that has been marked by the requesting PE.

--- **Note** ---

The global monitor only supports a single outstanding exclusive access to shareable memory for each PE.

---

A Load-Exclusive instruction by one PE has no effect on the global monitor state for any other PE.

A Store-Exclusive instruction performs a conditional store to memory:
- The store is guaranteed to succeed only if the PA accessed is marked as exclusive access for the requesting PE and both the local monitor and the global monitor state machines for the requesting PE are in the Exclusive Access state. In this case:
  - A status value of 0 is returned to a register to acknowledge the successful store.
  - The final state of the global monitor state machine for the requesting PE is IMPLEMENTATION DEFINED.
  - If the address accessed is marked for exclusive access in the global monitor state machine for any other PE then that state machine transitions to Open Access state.
- If no address is marked as exclusive access for the requesting PE, the store does not succeed:
  - A status value of 1 is returned to a register to indicate that the store failed.
  - The global monitor is not affected and remains in Open Access state for the requesting PE.
- If a different PA is marked as exclusive access for the requesting PE, it is IMPLEMENTATION DEFINED whether the store succeeds or not:
  - If the store succeeds a status value of 0 is returned to a register, otherwise a value of 1 is returned.
  - If the global monitor state machine for the PE was in the Exclusive Access state before the Store-Exclusive instruction it is IMPLEMENTATION DEFINED whether that state machine transitions to the Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.
In a shared memory system, the global monitor implements a separate state machine for each PE in the system. The state machine for accesses to shareable memory by PE(n) can respond to all the shareable memory accesses visible to it. This means it responds to:

- Accesses generated by PE(n).
- Accesses generated by the other observers in the Shareability domain of the memory location. These accesses are identified as (!n).

In a shared memory system, the global monitor implements a separate state machine for each observer that can generate a Load-Exclusive or a Store-Exclusive instruction in the system.

A global monitor:

- In the Exclusive Access state is set.
- In the Open Access state is clear.

**Clear global monitor event**

Whenever the global monitor state for a PE changes from Exclusive access to Open access, an event is generated and held in the Event register for that PE. This register is used by the Wait for Event mechanism, see Wait For Event and Send Event on page G1-8988.

Figure E2-5 shows the state machine for PE(n) in a global monitor.

For more information about marking, see Marking and the size of the marked memory block on page E2-7217.

---

**Note**

For the global monitor state machine, as shown in Figure E2-5:

- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the global monitor.
- Whether a Store-Exclusive instruction successfully updates memory or not depends on whether the address accessed matches the marked shareable memory address for the PE issuing the Store-Exclusive instruction, and whether the local and global monitors are in the exclusive state. For this reason, Figure E2-5 shows only how the operations by (!n) cause state transitions of the state machine for PE(n).
- A Load-Exclusive instruction can update only the marked shareable memory address for the PE issuing the Load-Exclusive instruction.
• When the global monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether a CLREX instruction causes the global monitor to transition from Exclusive Access to Open Access state.

• It is IMPLEMENTATION DEFINED:
  — Whether a modification to a Non-shareable memory location can cause a global monitor to transition from Exclusive Access to Open Access state.
  — Whether a Load-Exclusive instruction to a Non-shareable memory location can cause a global monitor to transition from Open Access to Exclusive Access state.

E2.10.3 Marking and the size of the marked memory block

When a Load-Exclusive instruction is executed, the resulting marked block ignores the least significant bits of the 64-bit memory address.

When a Load-Exclusive instruction is executed, a marked block of size $2^a$ bytes is created by ignoring the least significant bits of the memory address. A marked address is any address within this marked block. The size of the marked memory block is called the Exclusives reservation granule. The Exclusives reservation granule is IMPLEMENTATION DEFINED in the range 4 - 512 words.

Note

This definition means that the Exclusives reservation granule is:

• 4 words in an implementation where $a$ is 4.
• 512 words in an implementation where $a$ is 11.

For example, in an implementation where $a$ is 4, a successful LDREXB of address 0x341B4 defines a marked block using bits[47:4] of the address. This means that the four words of memory from 0x341B0 to 0x341BF are marked for exclusive access.

In some implementations the CTR identifies the Exclusives reservation granule, see CTR. Otherwise, software must assume that the maximum Exclusives reservation granule, 512 words, is implemented.

E2.10.4 Context switch support

An exception return clears the local monitor. As a result, performing a CLREX instruction as part of a context switch is not required in most situations.

Note

Context switching is not an application level operation. However, this information is included here to complete the description of the exclusive operations.

E2.10.5 Load-Exclusive and Store-Exclusive instruction usage restrictions

The Load-Exclusive and Store-Exclusive instructions are intended to work together as a pair, for example a LDREX/STREX pair or a LDREXB/STREXB pair. To support different implementations of these functions, software must follow the notes and restrictions given in this subsection.

The following notes describe use of a Load-Exclusive/Store-Exclusive instruction pair, LoadExcl/StoreExcl, to indicate the use of any of the Load-Exclusive/Store-Exclusive instruction pairs shown in Table E2-6 on page E2-7211. In this context, a LoadExcl/StoreExcl pair comprises two instructions in the same thread of execution:

• The exclusives support a single outstanding exclusive access for each PE thread that is executed. The architecture makes use of this by not requiring an address or size check as part of the IsExclusiveLocal() function. If the target VA of a StoreExcl is different from the VA of the preceding LoadExcl instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  — The StoreExcl either passes or fails, the status value returned by the StoreExcl is UNKNOWN, and the states of the local and global monitors for that PE are UNKNOWN.
Note

This means the StoreExcl might pass for some instances of a LoadExcl/StoreExcl pair with mismatched addresses, and fail for other instances of a LoadExcl/StoreExcl pair with mismatched addresses.

— The data at the address accessed by the LoadExcl, and at the address accessed by the StoreExcl, is UNKNOWN.

This means software can rely on a LoadExcl/StoreExcl pair to eventually succeed only if the LoadExcl and the StoreExcl are executed with the same VA.

- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the transaction size of a StoreExcl instruction is the same as the transaction size of the preceding LoadExcl instruction executed in that thread. If the transaction size of a StoreExcl instruction is different from the preceding LoadExcl instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  — The StoreExcl either passes or fails, and the status value returned by the StoreExcl is UNKNOWN.

Note

This means the StoreExcl might pass for some instances of a LoadExcl/StoreExcl pair with mismatched transaction sizes, and fail for other instances of a LoadExcl/StoreExcl pair with mismatched transaction sizes.

— The block of data of the size of the larger of the transaction sizes used by the LoadExcl/StoreExcl pair at the address accessed by the LoadExcl/StoreExcl pair, is UNKNOWN.

This means software can rely on a LoadExcl/StoreExcl pair to eventually succeed only if the LoadExcl and the StoreExcl have the same transaction size.

- LoadExcl/StoreExcl loops are guaranteed to make forward progress only if, for any LoadExcl/StoreExcl loop within a single thread of execution, the software meets all of the following conditions:
  1. Between the Load-Exclusive and the Store-Exclusive, there are no explicit memory effects, preloads, direct or indirect System register writes, address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, ISB barriers, or indirect branches.
  2. Between the Store-Exclusive returning a failing result and the retry of the corresponding Load-Exclusive:
     • There are no stores or PLDW instructions to any address within the Exclusives reservation granule accessed by the Store-Exclusive.
     • There are no loads or preloads to any address within the Exclusives reservation granule accessed by the Store-Exclusive that use a different VA alias to that address.
     • There are no direct or indirect System register writes, other than changes to the flag fields in the CPSR or FPSCR, caused by data processing or comparison instructions.
     • There are no direct or indirect address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, or indirect branches.
     • All loads and stores are to a block of contiguous virtual memory of not more than 512 bytes in size.

The Exclusives monitor can be cleared at any time without an application-related cause, provided that such clearing is not systematically repeated so as to prevent the forward progress in finite time of at least one of the threads that is accessing the Exclusives monitor. However, it is permissible for the LoadExcl/StoreExcl loop not to make forward progress if a different thread is repeatedly doing any of the following in a tight loop:
  — Performing stores to a PA covered by the Exclusives monitor.
  — Prefetching with intent to write to a PA covered by the Exclusives monitor.
  — Executing data cache clean, data cache invalidate, or data cache clean and invalidate instructions to a PA covered by the Exclusives monitor.
  — Executing instruction cache invalidate all instructions.
  — Executing instruction cache invalidate by VA instructions to a PA covered by the Exclusives monitor.
• Implementations can benefit from keeping the LoadExcl and StoreExcl operations close together in a single thread of execution. This minimizes the likelihood of the Exclusives monitor state being cleared between the LoadExcl instruction and the StoreExcl instruction. Therefore, for best performance, Arm strongly recommends a limit of 128 bytes between LoadExcl and StoreExcl instructions in a single thread of execution.

• The architecture sets an upper limit of 2048 bytes on the Exclusives reservation granule that can be marked as exclusive. For performance reasons, Arm recommends that objects that are accessed by exclusive accesses are separated by the size of the Exclusives reservation granule. This is a performance guideline rather than a functional requirement.

• After taking a Data Abort exception, the state of the Exclusives monitors is UNKNOWN.

• For the memory location accessed by a LoadExcl/StoreExcl pair, if the memory attributes for a StoreExcl instruction are different from the memory attributes for the preceding LoadExcl instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE. Where this occurs because the translation of the accessed address changes between the LoadExcl instruction and the StoreExcl instruction, the CONSTRAINED UNPREDICTABLE behavior is as follows:
  — The StoreExcl either passes or fails, and the status value returned by the StoreExcl is UNKNOWN.

     Note

This means the StoreExcl might pass for some instances of a LoadExcl/StoreExcl pair with changed memory attributes, and fail for other instances of a LoadExcl/StoreExcl pair with changed memory attributes.

— The data at the address accessed by the StoreExcl is UNKNOWN.

     Note

Another bullet point in this list covers the case where the memory attributes of a LoadExcl/StoreExcl pair differ as a result of using different VAs with different attributes that point to the same PA.

• The effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE, and the instruction might clear the monitor, or it might leave it in the Exclusive Access state. For address-based maintenance instructions, this also applies to the monitors of other PEs in the same Shareability domain as the PE executing the cache maintenance instruction, as determined by the Shareability domain of the address being maintained.

     Note

Arm strongly recommends that implementations ensure that the use of such maintenance instructions by a PE in the Non-secure state cannot cause a denial of service on a PE in the Secure state.

• If the mapping of the VA to PA is changed between the LoadExcl instruction and the StoreExcl instruction, and the change is performed using a break-before-make sequence as described in Using break-before-make when updating translation table entries on page G5-9221, if the StoreExcl is performed after another write to the same PA as the StoreExcl, and that other write was performed after the old translation was properly invalidated and that invalidation was properly synchronized, then the StoreExcl will not pass its monitor check.

     Note

Arm expects that, in many implementations, either:
  — The TLB invalidation will clear either the local or global monitor.
  — The PA will be checked between the LoadExcl and StoreExcl.

• The Exclusive Access state for an address accessed by a PE can be lost as a result of a PLDW instruction to the same PA executed by another PE. This means that a very high rate of repeated PLDW accesses to a memory location might impede the forward progress of another PE.
Note
In the event of repeatedly-contending LoadExcl/StoreExcl instruction sequences from multiple PEs, an implementation must ensure that forward progress is made by at least one PE.

E2.10.6 Use of WFE and SEV instructions by spin-locks

The architecture provides Wait For Event, Send Event, and Send Event Local instructions, WFE, SEV, SEVL, that can assist with reducing power consumption and bus contention caused by PEs repeatedly attempting to obtain a spin-lock. These instructions can be used at the application level, but a complete understanding of what they do depends on a system level understanding of exceptions. They are described in Wait For Event and Send Event on page G1-8988. However, when the global monitor for a PE changes from Exclusive Access state to Open Access state, an event is generated.

Note
This is equivalent to issuing an SEVL instruction on the PE for which the monitor state has changed. It removes the need for spinlock code to include an SEV instruction after clearing a spinlock.
Part F

The AArch32 Instruction Sets
Chapter F1
About the T32 and A32 Instruction Descriptions

This chapter describes each instruction. It contains the following sections:

- Format of instruction descriptions on page F1-7224.
- Standard assembler syntax fields on page F1-7228.
- Conditional execution on page F1-7229.
- Shifts applied to a register on page F1-7231.
- Memory accesses on page F1-7233.
- Encoding of lists of general-purpose registers and the PC on page F1-7234.
- General information about the T32 and A32 instruction descriptions on page F1-7235.
- Additional pseudocode support for instruction descriptions on page F1-7248.
- Additional information about Advanced SIMD and floating-point instructions on page F1-7249.
F1.1 Format of instruction descriptions

The instruction descriptions in Chapter F5 T32 and A32 Base Instruction Set Instruction Descriptions and Chapter F6 T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions normally use the following format:

• Instruction section title.
• Introduction to the instruction.
• A description of each encoding of the instruction.
• Assembler syntax.
• Pseudocode describing how the instruction operates.
• Notes, if applicable.

Each of these items is described in more detail in the following subsections.

F1.1.1 Instruction section title

The instruction section title gives the base mnemonic for the instruction or instructions described in the section. When one mnemonic has multiple forms described in separate instruction sections, this is followed by a short description of the form in parentheses. The most common use of this is to distinguish between forms of an instruction in which one of the operands is an immediate value and forms in which it is a register.

F1.1.2 Introduction to the instruction

The introduction to the instruction briefly describes the main features of the instruction. This description is not necessarily complete and is not definitive. If there is any conflict between it and the more detailed information that follows, the latter takes priority.

F1.1.3 Instruction encodings

This is a list of one or more instruction encodings. Each instruction encoding is labeled as:

• A1, A2, A3 … for the first, second, third, and any additional A32 encodings.
• T1, T2, T3 … for the first, second, third, and any additional T32 encodings.

Each instruction encoding description consists of:

• An assembly syntax that ensures that the assembler selects the encoding in preference to any other encoding. Sometimes, multiple syntax variants are given. These are written in a typewriter font using the conventions described in Assembler syntax prototype line conventions on page F1-7226. The correct one to use can be indicated by:
  — A subheading that identifies the encodings that correspond to the syntax. See, for example, the subheading Flag setting, rotate right with extend variant in the description of the A1 encoding of the ADC, ADCS (register) instructions in A1 on page F5-7446.
  — An annotation to the syntax, such as Inside IT block or Outside IT block. See, for example, the syntax descriptions of the T1 encoding of the ADC, ADCS (register) instructions in T1 on page F5-7447.

In other cases, the correct one to use can be determined by looking at the assembler syntax description and using it to determine which syntax corresponds to the instruction being disassembled.

There is usually more than one syntax variant that ensures re-assembly to any particular encoding, and the exact set of syntaxes that do so usually depends on the register numbers, immediate constants, and other operands to the instruction. For example, when assembling to the T32 instruction set, the syntax AND R0, R0, R8 ensures selection of a 32-bit encoding but AND R0, R0, R1 selects a 16-bit encoding.
For each instruction encoding belonging to a target instruction set, an assembler can use this information to determine whether it can use that encoding to encode the instruction requested by the UAL source. If multiple encodings can encode the instruction, then:

- If both a 16-bit encoding and a 32-bit encoding can encode the instruction, the architecture prefers the 16-bit encoding. This means the assembler must use the 16-bit encoding rather than the 32-bit encoding.

  Software can use the .W and .N qualifiers to specify the required encoding width, see Standard assembler syntax fields on page F1-7228.

- If multiple encodings of the same length can encode the instruction, the Assembler syntax subsection says which encoding is preferred, and how software can, instead, select the other encodings.

Each encoding also documents UAL syntax that selects it in preference to any other encoding.

If no encodings of the target instruction set can encode the instruction requested by the UAL source, normally the assembler generates an error saying that the instruction is not available in that instruction set.

--- Note ---

In some cases, an instruction is available in one instruction set but not in another. The Assembler syntax subsection identifies many of these cases. For example, the A32 instructions with bits<31:28> == 0b1111 described in Branch, branch with link, and block data transfer on page F4-7400, System register access, Advanced SIMD, floating-point, and Supervisor call on page F4-7402, and Unconditional instructions on page F4-7420 cannot have a Condition code, but the equivalent T32 instructions often can, and this usually appears in the Assembler syntax subsection as a statement that the A32 instruction cannot be conditional. However, some such cases are too complex to describe in the available space, so the definitive test of whether an instruction is available in a given instruction set is whether there is an available encoding for it in that instruction set.

The assembly syntax given for an encoding is therefore a suitable one for a disassembler to disassemble that encoding to. However, disassemblers might wish to use simpler syntaxes when they are suitable for the operand combination, to produce more readable disassembled code.

- An encoding diagram, where:
  - For a 32-bit A32 encoding diagram, the bits are numbered from 31-0.
  - For a 16-bit T32 encoding diagram, the bits are numbered from 15-0.

    This halfword can be described as hw1 of the instruction.

  - For a 32-bit T32 encoding diagram, the bits are numbered from 15-0 for each halfword, as a reminder that a 32-bit T32 instruction consists of two consecutive halfwords rather than a word.

    In this case, the left-hand halfword in the diagram is identified as hw1, and the right-hand halfword is identified as hw2.

Where instructions are stored using the standard little-endian instruction endianness:

- The encoding diagram for an A32 instruction at address A shows, from left to right, the bytes at addresses A+3, A+2, A+1, A.

- The encoding diagram for a 32-bit T32 instruction shows bytes in the order A+1, A for hw1, followed by bytes A+3, A+2 for hw2.

- Encoding-specific pseudocode. This is pseudocode that translates the encoding-specific instruction fields into inputs to the encoding-independent pseudocode in the Operation subsection, and that picks out any special cases in the encoding. For a detailed description of the pseudocode used and of the relationship between the encoding diagram, the encoding-specific pseudocode and the encoding-independent pseudocode, see Appendix K16 Arm Pseudocode Definition.

F1.1.4 Assembler symbols

The Assembly symbols describe the standard UAL syntax for the instruction.

Each syntax description consists of the following elements:

- Descriptions of all variable or optional fields of the syntax.
Some syntax fields are standardized across all or most instructions. Standard assembler syntax fields on page F1-7228 describes these fields.

By default, syntax fields that specify registers, such as <Rd>, <Rn>, or <Rt>, can be any of R0-R12 or LR in T32 instructions, and any of R0-R12, SP, or LR in A32 instructions. These require that the encoding-specific pseudocode set the corresponding integer variable (such as d, n, or t) to the corresponding register number, using 0-12 for R0-R12, 13 for SP, or 14 for LR:

- Normally, software can do this by setting the corresponding field in the instruction, typically named Rd, Rn, Rt, to the binary encoding of that number.
- In the case of 16-bit T32 encodings, the field is normally of length 3, and so the encoding is available only when the assembler syntax specifies one of R0-R7. Such encodings often use a register field name like Rdn. This indicates that the encoding is available only if <Rd> and <Rn> specify the same register, and that the register number of that register is encoded in the field if they do.

The description of a syntax field that specifies a register sometimes extends or restricts the permitted range of registers or documents other differences from the default rules for such fields. Examples of extensions are permitting the use of the SP in a T32 instruction, or permitting the use of the PC, identified using register number 15.

- Where appropriate, text that briefly describes changes from the pre-UAL assembler syntax. Where present, this usually consists of an alternative pre-UAL form of the assembler mnemonic. The pre-UAL assembler syntax does not conflict with UAL. Arm recommends that it is supported, as an optional extension to UAL, so that pre-UAL assembler source files can be assembled.

**Assembler syntax prototype line conventions**

The following conventions are used in assembler syntax prototype lines and their subfields:

- Any item bracketed by < and > is a short description of a type of value to be supplied by the user in that position. A longer description of the item is normally supplied by subsequent text. Such items often correspond to a similarly named field in an encoding diagram for an instruction. When the correspondence requires only the binary encoding of an integer value or register number to be substituted into the instruction encoding, it is not described explicitly. For example, if the assembler syntax for an instruction contains an item <Rn> and the instruction encoding diagram contains a 4-bit field named Rn, the number of the register specified in the assembler syntax is encoded in binary in the instruction field.

- If the correspondence between the assembler syntax item and the instruction encoding is more complex than simple binary encoding of an integer or register number, the item description indicates how it is encoded. This is often done by specifying a required output from the encoding-specific pseudocode, such as add = TRUE. The assembler must use only encodings that produce that output.

- Any item bracketed by { and } is optional. A description of the item and of how its presence or absence is encoded in the instruction is normally supplied by subsequent text. Many instructions have an optional destination register. Unless otherwise stated, if such a destination register is omitted, it is the same as the immediately following source register in the instruction syntax.

- In the assembler syntax, numeric constants are normally preceded by a #. Some UAL instruction syntax descriptions explicitly show this # as optional. Any UAL assembler:
  - Must treat the # as optional where an instruction syntax description shows it as optional.
  - Can treat the # either as mandatory or as optional where an instruction syntax description does not show it as optional.

  ______ Note _______

Arm recommends that UAL assemblers treat all uses of # shown in this manual as optional.

______

**spaces**

Single spaces are used for clarity, to separate items. When a space is obligatory in the assembler syntax, two or more consecutive spaces are used.
This indicates an optional + or - sign. If neither is coded, + is assumed.

All other characters must be encoded precisely as they appear in the assembler syntax. Apart from { and }, the special characters described above do not appear in the basic forms of assembler instructions documented in this manual. In a few places, the { and } characters must be encoded as part of a variable item. When this happens, the long description of the variable item indicates how they must be used.

**F1.1.5 Pseudocode describing how the instruction operates**

The *Operation for all classes* subsection contains encoding-independent pseudocode that describes the main operation of the instruction. For a detailed description of the pseudocode used and of the relationship between the encoding diagram, the encoding-specific pseudocode and the encoding-independent pseudocode, see Appendix K16 *Arm Pseudocode Definition*. 
F1.2 Standard assembler syntax fields

The following assembler syntax fields are standard across all or most instructions:

<>  Is an optional field. It specifies the condition under which the instruction is executed. See Conditional execution on page F1-7229 for the range of available conditions and their encoding. If <> is omitted, it defaults to always (AL).

<>  Specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

.N  Meaning narrow, specifies that the assembler must select a 16-bit encoding for the instruction. If this is not possible, an assembler error is produced.

.W  Meaning wide, specifies that the assembler must select a 32-bit encoding for the instruction. If this is not possible, an assembler error is produced.

If neither .W nor .N is specified, the assembler can select either 16-bit or 32-bit encodings. If both are available, it must select a 16-bit encoding. In a few cases, more than one encoding of the same length can be available for an instruction. The rules for selecting between such encodings are instruction-specific and are part of the instruction description. The assembler syntax includes a mandatory .W qualifier, along with a note describing the cases in which it applies, where this qualifier is required to select a particular encoding for an instruction. Additional assembler syntax will describe the syntax when the conditions are not met.

--- Note ---

When assembling to the A32 instruction set, the .N qualifier produces an assembler error and the .W qualifier has no effect.
F1.3 Conditional execution

Most T32 and A32 instructions can be executed conditionally, based on the values of the APSR Condition flags. Table F1-1 on page F1-7229 lists the available conditions.

Table F1-1 Condition codes

<table>
<thead>
<tr>
<th>cond</th>
<th>Mnemonic extension</th>
<th>Meaning (integer)</th>
<th>Meaning (floating-point)</th>
<th>Condition flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>Equal</td>
<td>Equal</td>
<td>Z == 1</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>Not equal</td>
<td>Not equal, or unordered</td>
<td>Z == 0</td>
</tr>
<tr>
<td>0010</td>
<td>CS (^b)</td>
<td>Carry set</td>
<td>Greater than, equal, or unordered</td>
<td>C == 1</td>
</tr>
<tr>
<td>0011</td>
<td>CC (^c)</td>
<td>Carry clear</td>
<td>Less than</td>
<td>C == 0</td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
<td>Minus, negative</td>
<td>Less than</td>
<td>N == 1</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td>Plus, positive or zero</td>
<td>Greater than, equal, or unordered</td>
<td>N == 0</td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
<td>Overflow</td>
<td>Unordered</td>
<td>V == 1</td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
<td>No overflow</td>
<td>Not unordered</td>
<td>V == 0</td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
<td>Unsigned higher</td>
<td>Greater than, or unordered</td>
<td>C == 1 and Z == 0</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td>Unsigned lower or same</td>
<td>Less than or equal</td>
<td>C == 0 or Z == 1</td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
<td>Signed greater than or equal</td>
<td>Greater than or equal</td>
<td>N == V</td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td>Signed less than</td>
<td>Less than, or unordered</td>
<td>N != V</td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
<td>Signed greater than</td>
<td>Greater than</td>
<td>Z == 0 and N == V</td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
<td>Signed less than or equal</td>
<td>Less than, equal, or unordered</td>
<td>Z == 1 or N != V</td>
</tr>
<tr>
<td>1110</td>
<td>None (AL) (^d)</td>
<td>Always (unconditional)</td>
<td>Always (unconditional)</td>
<td>Any</td>
</tr>
</tbody>
</table>

a. Unordered means at least one NaN operand.
b. HS (unsigned higher or same) is a synonym for CS.
c. LO (unsigned lower) is a synonym for CC.
d. AL is an optional mnemonic extension for always, except in IT instructions. For details, see IT on page F5-7580.

In T32 instructions, the condition, if it is not AL, is normally encoded in a preceding IT instruction. For more information, see Conditional instructions on page F2-7257 and IT on page F5-7580. Some conditional branch instructions do not require a preceding IT instruction, because they include a Condition code in their encoding.

Implementations can provide a set of ITD control fields, SCTLR.ITD, SCTLR_EL1.ITD, and HSCTLR.ITD, to disable use of IT for some instructions, making them UNDEFINED. For more information, see:

- Disabling or enabling PL0 and PL1 use of AArch32 optional functionality on page G1-9004.
- Disabling or enabling EL2 use of AArch32 optional functionality on page G1-9013.

In A32 instructions, bits[31:28] of the instruction contain either:

- The Condition code, see The Condition code field in A32 instruction encodings.
- 0b1111 for some A32 instructions that can only be executed unconditionally.

F1.3.1 The Condition code field in A32 instruction encodings

Every conditional A32 instruction contains a 4-bit Condition code field, the cond field, in bits 31-28:

<table>
<thead>
<tr>
<th>cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

\[^b\] HS (unsigned higher or same) is a synonym for CS.
\[^c\] LO (unsigned lower) is a synonym for CC.
\[^d\] AL is an optional mnemonic extension for always, except in IT instructions. For details, see IT on page F5-7580.
This field contains one of the values 0b0000-0b1110, as shown in Table F1-1 on page F1-7229. Most instruction mnemonics can be extended with the letters defined in the Mnemonic extension on page F1-7229 column of that table.

If the always (AL) condition is specified, the instruction is executed irrespective of the value of the Condition flags. The absence of a Condition code on an instruction mnemonic implies the AL Condition code.

**F1.3.2 Pseudocode description of conditional execution**

The AArch32.CurrentCond() function returns a 4-bit condition specifier as follows:

- For A32 instructions, it returns bits[31:28] of the instruction.
- For the T1 and T3 encodings of the Branch instruction (see B on page F5-7491), it returns the 4-bit cond field of the encoding.
- For all other T32 instructions:
  - If PSTATE.IT<3:0> != '0000' it returns PSTATE.IT<7:4>.
  - If PSTATE.IT<7:0> == '00000000' it returns '1110'.
  - Otherwise, execution of the instruction is CONSTRAINED UNPREDICTABLE.

For more information, see Process state, PSTATE on page E1-7133.

The ConditionPassed() function uses this condition specifier and the Condition flags to determine whether the instruction must be executed, by calling the ConditionHolds() function.

Chapter J1 Armv8 Pseudocode includes the definitions of these functions.

Undefined Instruction exception on page G1-8964 describes the handling of conditional instructions that are UNDEFINED, UNPREDICTABLE, or CONSTRAINED UNPREDICTABLE. The pseudocode in the manual, as a sequential description of the instructions, has limitations in this respect. For more information, see Limitations of the instruction pseudocode on page K16-11810.
F1.4 Shifts applied to a register

A32 register offset load/store word and unsigned byte instructions can apply a wide range of different constant shifts to the offset register. Both T32 and A32 data-processing instructions can apply the same range of different constant shifts to the second operand register. For details, see Constant shifts.

A32 data-processing instructions can apply a register-controlled shift to the second operand register.

F1.4.1 Constant shifts

These are the same in T32 and A32 instructions, except that the input bits come from different positions. <shift> is an optional shift to be applied to <Rm>. It can be any one of:

(omitted) No shift.

LSL <n> Logical shift left <n> bits. 1 <= <n> <= 31.

LSR <n> Logical shift right <n> bits. 1 <= <n> <= 32.

ASR <n> Arithmetic shift right <n> bits. 1 <= <n> <= 32.

ROR <n> Rotate right <n> bits. 1 <= <n> <= 31.

RRX Rotate right one bit, with extend. Bit[0] is written to shifter_carry_out, bits[31:1] are shifted right one bit, and the Carry flag is shifted into bit[31].

—— Note ———

Assemblers can permit the use of some or all of ASR #0, LSL #0, LSR #0, and ROR #0 to specify that no shift is to be performed. This is not standard UAL, and the encoding selected for T32 instructions might vary between UAL assemblers if it is used. To ensure disassembled code assembles to the original instructions, disassemblers must omit the shift specifier when the instruction specifies no shift.

Similarly, assemblers can permit the use of #0 in the immediate forms of ASR, LSL, LSR, and ROR instructions to specify that no shift is to be performed, that is, that a MOV (register) instruction is wanted. Again, this is not standard UAL, and the encoding selected for T32 instructions might vary between UAL assemblers if it is used. To ensure disassembled code assembles to the original instructions, disassemblers must use the MOV (register) syntax when the instruction specifies no shift.

Encoding

The assembler encodes <shift> into two type bits and five immediate bits, as follows:

(omitted) type = 0b00, immediate = 0.

LSL <n> type = 0b00, immediate = <n>.

LSR <n> type = 0b01.

If <n> < 32, immediate = <n>.

If <n> == 32, immediate = 0.

ASR <n> type = 0b10.

If <n> < 32, immediate = <n>.

If <n> == 32, immediate = 0.

ROR <n> type = 0b11, immediate = <n>.

RRX type = 0b11, immediate = 0.
F1.4.2 Register controlled shifts

These are available only in A32 instructions.

<type> is the type of shift to apply to the value read from <Rm>. It must be one of:

- **ASR**: Arithmetic shift right, encoded as type = 0b10.
- **LSL**: Logical shift left, encoded as type = 0b00.
- **LSR**: Logical shift right, encoded as type = 0b01.
- **ROR**: Rotate right, encoded as type = 0b11.

The bottom byte of <Rs> contains the shift amount.

F1.4.3 Pseudocode description of instruction-specified shifts and rotates

The pseudocode enumeration `SRType()` defines the shift types. Shift and rotate instruction decode is described by the pseudocode function:

- `DecodeImmShift()` for a constant shift.
- `DecodeRegShift()` for a register controlled shift.

Shift and rotate operations are made by the pseudocode function `Shift()`.
F1.5 Memory accesses

Commonly, the following addressing modes are permitted for memory access instructions:

Offset addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access. The value of the base register is unchanged.

The assembly language syntax for this mode is:

\[ [<Rn>, <offset>] \]

Pre-indexed addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access, and written back into the base register.

The assembly language syntax for this mode is:

\[ [<Rn>, <offset>]! \]

Post-indexed addressing

The address obtained from the base register is used, unchanged, as the address for the memory access. The offset value is applied to the address, and written back into the base register.

The assembly language syntax for this mode is:

\[ [<Rm>, <offset>] \]

In each case, \(<Rn>\) is the base register. \(<offset>\) can be:

- An immediate constant, such as \(<imm8>\) or \(<imm12>\).
- An index register, \(<Rm>\).
- A shifted index register, such as \(<Rm>, \text{LSL} #<shift>\).

For information about unaligned access, endianness, and exclusive access, see:

- Alignment support on page E2-7193.
- Endian support on page E2-7195.
- Synchronization and semaphores on page E2-7211.
F1.6 Encoding of lists of general-purpose registers and the PC

A number of instructions operate on lists of general-purpose registers. For some load instructions, the list of registers to be loaded can include the PC. For these instructions, the assembler syntax includes a `{registers}` field, which provides a list of the registers to be operated on, with list entries separated by commas.

The registers list is encoded in the instruction encoding. Most often, this is done using an 8-bit, 13-bit, or 16-bit register_list field. This section gives more information about these and other possible register list encodings.

In a register_list field, each bit corresponds to a single register, and if the `{registers}` field of the assembler instruction includes Rt then register_list<tt> is set to 1, otherwise it is set to 0.

The full rules for the encoding of lists of general-purpose registers, and possibly the PC, are:

- Except for the cases listed here, 16-bit T32 encodings use an 8-bit register list, and can access only registers R0-R7.
  
  The exceptions to this rule are:
  
  — The T1 encoding of POP uses an 8-bit register list, and an additional bit, P, that corresponds to the PC. This means it can access any of R0-R7 and the PC.
  
  — The T1 encoding of PUSH uses an 8-bit register list, and an additional bit, M, that corresponds to the LR. This means it can access any of R0-R7 and the LR.

- 32-bit T32 encodings of load operations use a 13-bit register list, and two additional bits, M, corresponding to the LR, and P, corresponding to the PC. This means these instructions can access any of R0-R12 and the LR and PC.

- 32-bit T32 encodings of store operations use a 13-bit register list, and one additional bit, M, corresponding to the LR. This means these instructions can access any of R0-R12 and the LR.

- Except for the case listed here, A32 encodings use a 16-bit register list. This means these instructions can access any of R0-R12 and the SP, LR, and PC.
  
  The exception to this rule is:
  
  — The System instructions LDM (exception return) and LDM (User registers) use a 15-bit register list. This means these instructions can access any of R0-R12 and the SP and LR.

- The T3 and A2 encodings of POP, and the T3 and A2 encodings of PUSH, access a single register from the set of registers {R0-R12, LR, PC} and encode the register number in the Rt field.

--- Note

POp is a load operation, and PUSH is a store operation.

In every case, the encoding-specific pseudocode converts the register list into a 32-bit variable, registers, with a bit corresponding to each of the registers R0-R12, SP, LR, and PC.

--- Note

Some Advanced SIMD and floating-point instructions operate on lists of SIMD and floating-point registers. The assembler syntax of these instructions includes a `{list}` field that specifies the registers to be operated on, and the description of the instruction in *Alphabetical list of T32 and A32 base instruction set instructions* on page F5-7442 defines the use and encoding of this field.
F1.7 General information about the T32 and A32 instruction descriptions

Chapter F3 T32 Instruction Set Encoding describes the T32 instruction encodings, and Chapter F4 A32 Instruction Set Encoding describes the A32 instruction encodings. The following subsections give more information about the descriptions of these instructions and their encodings:

- Execution of instructions in debug state.
- Fixed values in AArch32 instruction and System register descriptions.
- UNDEFINED, UNPREDICTABLE, and CONstrained UNPREDICTABLE instruction set space on page F1-7236.
- T32 and A32 Advanced SIMD and floating-point instruction encodings on page F1-7237.
- The PC and the use of 0b1111 as a register specifier in T32 and A32 instructions on page F1-7241.
- The SP and the use of 0b1101 as a register specifier in T32 and A32 instructions on page F1-7242.
- Modified immediate constants in T32 and A32 instructions on page F1-7243.

F1.7.1 Execution of instructions in debug state

In general, except for the instructions described in Debug state on page F2-7276, the T32 instruction descriptions do not indicate any differences in the behavior of the instruction if it is executed in Debug state. For this information, see Executing instructions in Debug state on page H2-10243.

Note

- A32 instructions cannot be executed in Debug state.
- For many T32 instructions, execution is unchanged in Debug state. Executing instructions in Debug state on page H2-10243 identifies these instructions.

F1.7.2 Fixed values in AArch32 instruction and System register descriptions

This section summarizes the terms used to describe fixed values in AArch64 register and instruction descriptions. The Glossary gives full descriptions of these terms, and each entry in this section includes a link to the corresponding Glossary entry.

Note

In register descriptions, the meaning of some bits depends on the PE state. This affects the definitions of RES0 and RES1, as shown in the Glossary.

The following terms are used to describe bits or fields with fixed values:

RAZ      Read-As-Zero. See Read-As-Zero (RAZ).

In diagrams, a RAZ bit can be shown as 0.

(0), RES0  Reserved. Should-Be-Zero (SBZ) or RES0.

In instruction encoding diagrams, and sometimes in other descriptions, (0) indicates an SBZ bit. If the bit is set to 1, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if the value of the bit was 0.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either (0)(0)(0) or as (000).

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES0. For more information, see the Glossary definition of RES0.

Note

Some of the System instruction descriptions in this chapter are based on the field description of the input value for the instruction. These are register descriptions and therefore can include RES0 fields,
The (0) and RES0 descriptions can be applied to bits or bitfields that are read-only, or are write-only. The Glossary definitions cover these cases.

RAO
Read-As-One. See Read-As-One (RAO).
In diagrams, a RAO bit can be shown as 1.

(1), RES1
Reserved. Should-Be-One (SBO) or RES1.
In instruction encoding diagrams, and sometimes in other descriptions, (1) indicates an SBO bit. If the bit is set to 0, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if the value of the bit was 1.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either (1)(1)(1) or as (111).
In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES1. For more information, see the Glossary definition of RES1.

Note
Some of the System instruction descriptions in this chapter are based on the field description of the input value for the instruction. These are register descriptions and therefore can include RES1 fields.

The (1) and RES1 descriptions can be applied to bits or bitfields that are read-only, or are write-only. The Glossary definitions cover these cases.

Note
In register diagrams, (0) is a synonym for RES0, and (1) is a synonym for RES1, where RES0 and RES1 are defined in the Glossary. However, when used in an instruction encoding diagram, (0) and (1) have the narrower definition that behavior is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE if either:

- A bit marked as (0) has the value 1.
- A bit marked as (1) has the value 0.

F1.7.3 UNDEFINED, UNPREDICTABLE, and CONSTRAINED UNPREDICTABLE instruction set space

An attempt to execute an unallocated instruction results in either:

- Unpredictable behavior. The instruction is described as UNPREDICTABLE or CONSTRAINED UNPREDICTABLE.
  
  From the introduction of Armv8-A, the architecturally UNPREDICTABLE behavior in AArch32 state is greatly reduced. Most cases that earlier versions of the architecture describe as UNPREDICTABLE become either:
  
  — CONSTRAINED UNPREDICTABLE, meaning the architecture defines a limited range of permitted behaviors.
  
  — Fully predictable.
  
  For more information, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

- An Undefined Instruction exception. The instruction is described as UNDEFINED.

An instruction is UNDEFINED if it is declared as UNDEFINED in an instruction description, or in Chapter F3 T32 Instruction Set Encoding or Chapter F4 A32 Instruction Set Encoding.

An instruction is UNPREDICTABLE only if:

- It is declared as UNPREDICTABLE in an instruction description or in Chapter F3 or Chapter F4, and Appendix K1 does not redefine the behavior as CONSTRAINED UNPREDICTABLE.

  The pseudocode for that encoding does not indicate that a different special case applies, and a bit marked (0) or (1) in the encoding diagram of an instruction is not 0 or 1 respectively. In most cases, Armv8 makes these cases CONSTRAINED UNPREDICTABLE, as described in SBZ or SBO fields T32 and A32 in instructions on page K1-11566.
Unless otherwise specified, T32 and A32 instructions provided as part of an architectural extension, or by an optional feature of the architecture, are UNDEFINED in an implementation that does not include that extension or feature.

--- Note ---

Examples of where this rule applies are:

- The instructions provided by the Cryptographic Extension.
- The System instructions that provide access to the System registers of the OPTIONAL Performance Monitors Extension.
- The Advanced SIMD and floating-point instructions.

---

For more information about UNDEFINED, UNPREDICTABLE, and CONSTRAINED UNPREDICTABLE instruction behavior, see *Undefined Instruction exception* on page G1-8964.

### F1.7.4 T32 and A32 Advanced SIMD and floating-point instruction encodings

The T32 and A32 encodings of Advanced SIMD and floating-point instructions are described in Chapter F3 *T32 Instruction Set Encoding* and in Chapter F4 *A32 Instruction Set Encoding*. This means:

- The instruction groups, and the set of instructions in each group, are identical for T32 and A32.
- For each instruction:
  - Each T32 encoding is exactly equivalent to an A32 encoding.
  - There is no T32 encoding without an equivalent A32 encoding, and no A32 encoding without an equivalent T32 encoding.

---

- In the T32 instruction sets, the Advanced SIMD and floating-point instructions have 32-bit encodings.
- In the base instruction sets, some instructions are common to the T32 and A32 instruction sets, whereas other instructions have equivalent but not identical functionality in the two instruction sets.

32-bit T32 encodings are described as two contiguous halfwords, \(\{\text{hw1, hw2}\}\), as described in *Instruction encodings* on page F1-7224. In general:

- \(\text{hw1}\) of a T32 encoding maps onto bits\([31:16]\) of an equivalent A32 encoding.
- \(\text{hw2}\) of a T32 encoding maps onto bits\([15:0]\) of an equivalent A32 encoding.

However, the different structures of the T32 instruction encoding space and the A32 instruction encoding space mean that:

- For a given Advanced SIMD and floating-point instruction group:
  - The positions of the fields that identify the instruction, or instruction encoding, within the instruction group might differ between the T32 encodings and the A32 encodings.
  - However, the field values that identify the instruction of instruction encoding are identical for the T32 encoding and the A32 encoding.

The remainder of this section describes the equivalence of the T32 and A32 encodings for each of the Advanced SIMD and floating-point instruction groups.

### Advanced SIMD data-processing

The T32 encoding of the Advanced SIMD data-processing group is:
The A32 encoding of the Advanced SIMD data-processing group is:

\[
\begin{array}{ccccccccccc}
| 31 | 25 | 24 | 23 | 22 | 5 | 4 | 3 | 0 | \\
\hline
1111001 \\
\hline
\end{array}
\]

The encodings in this group are identified by:

- \( \text{hw1}[15:13] \) of the T32 encoding is equivalent to bits\([27:25]\) of the A32 encoding, and:
  - Has the value \( 0b111 \) in the T32 encoding.
  - Has the value \( 0b001 \) in the A32 encoding.

- \( \text{hw1}[11:8] \) of the T32 encoding is equivalent to bits\([31:28]\) of the A32 encoding, and has the value \( 0b111 \).

This table shows the equivalence of the fields that identify the instructions, or instruction encodings, within this group:

<table>
<thead>
<tr>
<th>T32 encoding</th>
<th>A32 encoding</th>
<th>Field size</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0:op1</td>
<td>op0</td>
<td>2 bits</td>
</tr>
<tr>
<td>op2</td>
<td>op1</td>
<td>15 bits</td>
</tr>
<tr>
<td>op3</td>
<td>op2</td>
<td>1 bit</td>
</tr>
<tr>
<td>op4</td>
<td>op3</td>
<td>1 bit</td>
</tr>
</tbody>
</table>

**Advanced SIMD element or structure load/store**

The T32 encoding of the Advanced SIMD element or structure load/store group is:

\[
\begin{array}{cccccccccccc}
| 15 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 12 | 11 | 10 | 9 | 0 | \\
\hline
1111001 \\
\hline
\end{array}
\]

The A32 encoding of the Advanced SIMD element or structure load/store group is:

\[
\begin{array}{cccccccccccc}
| 31 | 24 | 23 | 22 | 21 | 20 | 19 | 12 | 11 | 10 | 9 | 0 | \\
\hline
1110100 \\
\hline
\end{array}
\]

The encodings in this group are identified by:

- \( \text{hw1}[15:12] \) of the T32 encoding is equivalent to bits\([31:28]\) of the A32 encoding, and has the value \( 0b1111 \).
• hw1[11:8] of the T32 encoding is equivalent to bits[27:24] of the A32 encoding, and:
  — Has the value 0b1001 in the T32 encoding.
  — Has the value 0b0100 in the A32 encoding.
• hw1[4] of the T32 encoding is equivalent to bit[20] of the A32 encoding, and has the value 0b0.

op0, op1, and op2 are the fields that identify the instructions, or instruction encodings, within this group, and they are in equivalent positions in the T32 and A32 encodings.

**Advanced SIMD and floating-point load/store and 64-bit register moves**

The T32 encoding of the Advanced SIMD and floating-point load/store and 64-bit register moves group is:

<table>
<thead>
<tr>
<th>15</th>
<th>8</th>
<th>5 4</th>
<th>0</th>
<th>15</th>
<th>11</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110110</td>
<td>op0</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The A32 encoding of the Advanced SIMD and floating-point load/store and 64-bit register moves group is:

<table>
<thead>
<tr>
<th>31</th>
<th>27 24</th>
<th>21 20</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>#=1111 110</td>
<td>op0</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The encodings in the group are identified by:

• hw1[15:12] of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and:
  — Has the value 0b1110 in the T32 encoding.
  — Can have any value other than 0b1111 in the A32 encoding.
    This range of values is required because A32 instructions in this group can be executed conditionally, see *Conditional execution* on page F1-7229.
• hw1[11:9] of the T32 encoding is equivalent to bits[27:25] of the A32 encoding, and has the value 0b110.
• hw2[11:9] of the T32 encoding is equivalent to bits[23:21] of the A32 encoding, and has the value 0b110.

op0 is the field that identifies the instructions, or instruction encodings, within this group, and is in equivalent positions in the T32 and A32 encodings.

**Advanced SIMD and floating-point 32-bit register moves**

The T32 encoding of the Advanced SIMD and floating-point 32-bit register moves group is:

<table>
<thead>
<tr>
<th>15</th>
<th>7</th>
<th>5 4</th>
<th>0</th>
<th>15</th>
<th>12</th>
<th>8</th>
<th>7 5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11101110</td>
<td>op0</td>
<td>101</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The A32 encoding of the Advanced SIMD 32-bit register moves group is:

<table>
<thead>
<tr>
<th>31</th>
<th>27 23</th>
<th>21 20</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7 5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>#=1111 1110</td>
<td>op0</td>
<td>101</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The encodings in this group are identified by:

- \( \text{hw1}[15:12] \) of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and:
  - Has the value 0b1110 in the T32 encoding.
  - Can have any value other than 0b1111 in the A32 encoding.
  
  This range of values is required because A32 instructions in this group can be executed conditionally, see Conditional execution on page F1-7229.

- \( \text{hw1}[11:8] \) of the T32 encoding is equivalent to bits[27:24] of the A32 encoding, and has the value 0b1110.

- \( \text{hw2}[4] \) of the T32 encoding is equivalent to bit[4] of the A32 encoding, and has the value 0b1.

\( \text{op0} \) is the field that identifies the instructions, or instruction encodings, within this group, and is in equivalent positions in the T32 and A32 encodings.

**Floating-point data-processing**

The T32 encoding of the Floating-point data-processing group is:

```
| 15 | 8 | 7 | 4 | 3 | 0 | 15 | 12 | 11 | 10 | 9 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|---|---|---|---|---|----|----|----|----|----|---|---|---|---|---|---|---|
| 11101110 | op0 | 10 | | | | 0 |
```  

The A32 encoding of the Floating-point data-processing group is:

```
| 31 | 28 | 27 | 24 | 23 | 20 | 19 | 12 | 11 | 10 | 9 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|
| !=1111 | 1110 | op0 | 10 | | | 0 |
```

The encodings in this group are identified by:

- \( \text{hw1}[15:12] \) of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and:
  - In the T32 encoding, \( \text{hw1}[15:13] \) has the value 0b111, and \( \text{hw1}[12] \) is the \( \text{op0} \) parameter used in identifying instruction encodings within this group.
  - In the A32 encoding, is the \( \text{cond} \) field and also implies the value of bit[28] of some A32 instruction encodings within this group, as the following table shows:

<table>
<thead>
<tr>
<th>\text{cond}</th>
<th>\text{Significance of bit}[28] in A32 encodings</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=0b1111</td>
<td>Part of the ( \text{cond} ) field.</td>
</tr>
<tr>
<td>0b1111</td>
<td>Has fixed value of 1.</td>
</tr>
</tbody>
</table>

The range of \( \text{cond} \) values other than 0b1111 is required because A32 instructions in this group can be executed conditionally, see Conditional execution on page F1-7229.

- \( \text{hw1}[11:8] \) of the T32 encoding is equivalent to bits[27:24] of the A32 encoding, and has the value 0b1110.

- \( \text{hw2}[4] \) of the T32 encoding is equivalent to bit[4] of the A32 encoding, and has the value 0b0.
This table shows the equivalence of the fields that identify the instructions, or instruction encodings, within this group:

<table>
<thead>
<tr>
<th>T32 encoding</th>
<th>A32 encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>Bit[28] of the instruction encoding is 1 when cond is 0b1111.</td>
</tr>
<tr>
<td>op1</td>
<td>op0</td>
</tr>
<tr>
<td>op2</td>
<td>op1</td>
</tr>
<tr>
<td>op3</td>
<td>op2</td>
</tr>
</tbody>
</table>

### F1.7.5 The PC and the use of 0b1111 as a register specifier in T32 and A32 instructions

Restrictions on the use of PC or 0b1111 as a register specifier differ between the T32 and the A32 instruction sets, as described in:

- **T32 restrictions on the use of the PC, and use of 0b1111 as a register specifier.**
- **A32 restrictions on the use of PC or 0b1111 as a register specifier** on page F1-7242.

#### T32 restrictions on the use of the PC, and use of 0b1111 as a register specifier

The use of 0b1111 as a register specifier is not normally permitted in T32 instructions. When a value of 0b1111 is permitted, a variety of meanings is possible. For register reads, these meanings include:

- Read the PC value, that is, the address of the current instruction + 4. The base register of the table branch instructions TBB and TBH can be the PC. This means branch tables can be placed in memory immediately after the instruction.

  ____________ Note ____________
  Arm deprecates use of the PC as the base register in the STC instruction.

- Read the word-aligned PC value, that is, the address of the current instruction + 4, with bits[1:0] forced to zero. The base register of LDC, LDR, LDRB, LDRD (pre-indexed, no writeback), LDRH, LDRSB, and LDRSH instructions can be the word-aligned PC. This provides PC-relative data addressing. In addition, some encodings of the ADD and SUB instructions permit their source registers to be 0b1111 for the same purpose.

- Read zero. This is done in some cases when one instruction is a special case of another, more general instruction, but with one operand zero. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page.

For register writes, these meanings include:

- The PC can be specified as the destination register of an LDR instruction. This is done by encoding Rt as 0b1111. The loaded value is treated as an address, and the effect of execution is a branch to that address. Bit[0] of the loaded value selects whether to execute A32 or T32 instructions after the branch.

- Some other instructions write the PC in similar ways. An instruction can specify that the PC is written:
  - Implicitly, for example, branch instructions.
  - Explicitly by a register specifier of 0b1111, for example 16-bit MOV (register) instructions.
  - Explicitly by using a register mask, for example LDM instructions.

  The address to branch to can be:
  - A loaded value, for example, RFE.
  - A register value, for example, BX.
  - The result of a calculation, for example, TBB or TBH.
The method of choosing the instruction set used after the branch can be:

- Similar to the LDR case, for example, LDM or BX.
- A fixed instruction set other than the one currently being used, for example, the immediate form of BLX.
- Unchanged, for example, branch instructions or 16-bit MOV (register) instructions.
- Set from the SPSR.T bit, for RFE and SUBS PC, LR, #imm8.

- Discard the result of a calculation. This is done in some cases when one instruction is a special case of another, more general instruction, but with the result discarded. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page.

- If the destination register specifier of an LDRB, LDRH, LDRSB, or LDRSH instruction is 0b1111, the instruction is a memory hint instead of a load operation.

- If the destination register specifier of an MRC instruction is 0b1111, bits[31:28] of the value transferred from the System register are written to the N, Z, C, and V condition flags in the APSR, and bits[27:0] are discarded.

### A32 restrictions on the use of PC or 0b1111 as a register specifier

In A32 instructions, the use of 0b1111 as a register specifier specifies the PC.

Many instructions are CONSTRAINED UNPREDICTABLE if they use 0b1111 as a register specifier. This is specified by pseudocode in the instruction description. Armv8-A constrains the resulting CONSTRAINED UNPREDICTABLE behavior, see Using R15 by instruction on page K1-11563.

**Note**

Arm deprecates use of the PC as the base register in any store instruction.

### F1.7.6 The SP and the use of 0b1101 as a register specifier in T32 and A32 instructions

In the T32 and A32 instruction sets, Arm recommends that the use of 0b1101 as a register specifier specifies the SP.

**Note**

- The recommendation that the register specifier 0b1101 is used only to specify the SP applies to both the T32 and the A32 instruction sets.

- Despite this recommendation, T32 instructions that can access R13, or the SP, behave predictably from the introduction of Armv8.

### F1.7.7 Modified immediate constants in T32 and A32 instructions

The following sections describe the encoding of modified immediate constants:

- Modified immediate constants in T32 instructions.
- Modified immediate constants in A32 instructions.

The encoding of a modified immediate constant in a 32-bit T32 instruction is:

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>imm3</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Modified immediate constants in T32 and A32 Advanced SIMD instructions.
- Modified immediate constants in T32 and A32 floating-point instructions.
Table F1-2 shows the range of modified immediate constants available in T32 data-processing instructions, and their encoding in the a, b, c, d, e, f, g, h, and i bits, and the imm3 field, in the instruction.

Table F1-2 Encoding of modified immediates in T32 data-processing instructions

<table>
<thead>
<tr>
<th>i:imm3:a</th>
<th>&lt;const&gt; a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000x</td>
<td>00000000 00000000 00000000 abcdefgh</td>
</tr>
<tr>
<td>0001x</td>
<td>00000000 abcdefgh 00000000 abcdefgh</td>
</tr>
<tr>
<td>0010x</td>
<td>abcdefgh 00000000 abcdefgh 00000000</td>
</tr>
<tr>
<td>0011x</td>
<td>abcdefgh abcdefgh abcdefgh abcdefgh</td>
</tr>
<tr>
<td>01000</td>
<td>1bcdefgh 00000000 00000000 00000000</td>
</tr>
<tr>
<td>01001</td>
<td>01bcdefgh 00000000 00000000 00000000</td>
</tr>
<tr>
<td>01010</td>
<td>001bcdefgh 00000000 00000000 00000000</td>
</tr>
<tr>
<td>01011</td>
<td>0001bcdefgh 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

a. This table shows the immediate constant value in binary form, to relate abcdedgh to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).

b. Arm deprecates using a modified immediate with abcdedgh == 00000000, and these cases are CONSTRAINED UNPREDICTABLE, see UNPREDICTABLE cases in immediate constants in T32 data-processing instructions on page K1-11566.

c. Not available in A32 instructions if h == 1.

Note

As the footnotes to Table F1-2 show, the range of values available in T32 modified immediate constants is slightly different from the range of values available in A32 instructions. See Modified immediate constants in A32 instructions on page F1-7244 for the A32 values.

Carry out

A logical instruction with i:imm3:a == '00xxx' does not affect the Carry flag. Otherwise, a logical flag-setting instruction sets the Carry flag to the value of bit[31] of the modified immediate constant.

Operation of modified immediate constants, T32 instructions

For a T32 data-processing instruction, the T32ExpandImm() pseudocode function returns the value of the 32-bit immediate constant, calling T32ExpandImm_C() to evaluate the constant.
Modified immediate constants in A32 instructions

The encoding of a modified immediate constant in an A32 instruction is:

```
rotation | a b c d e f g h
-----------|---------------------
0000      | 00000000 00000000 00000000 00000000
0001      | gh000000 00000000 00000000 00000000
0010      | efg0000 00000000 00000000 00000000
0011      | cdefg00 00000000 00000000 00000000
0100      | abcd0000 00000000 00000000 00000000
0101      | 00000000 00000000 00000000 00000000
0110      | 00000000 00000000 00000000 00000000
0111      | 00000000 00000000 00000000 00000000
```

Table F1-3 shows the range of modified immediate constants available in A32 data-processing instructions, and their encoding in the a, b, c, d, e, f, g, and h bits and the rotation field in the instruction.

### Table F1-3 Encoding of modified immediates in A32 processing instructions

<table>
<thead>
<tr>
<th>rotation</th>
<th>&lt;const&gt; a</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00000000 00000000 00000000 abcdefgh</td>
</tr>
<tr>
<td>00010000</td>
<td>gh000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>00100000</td>
<td>efg0000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>00110000</td>
<td>cdefg00 00000000 00000000 00000000</td>
</tr>
<tr>
<td>01000000</td>
<td>abcd0000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>01010000</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>01100000</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>01110000</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>10000000</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>10010000</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>11000000</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>11100000</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>11110000</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

Note

The range of values available in A32 modified immediate constants is slightly different from the range of values available in 32-bit T32 instructions. See Modified immediate constants in T32 instructions on page F1-7242.

### Carry out

A logical instruction with the rotation field set to 0b0000 does not affect APSR.C. Otherwise, a logical flag-setting instruction sets APSR.C to the value of bit[31] of the modified immediate constant.

### Constants with multiple encodings

Some constant values have multiple possible encodings. In this case, a UAL assembler must select the encoding with the lowest unsigned value of the rotation field. This is the encoding that appears first in Table F1-3. For example, the constant #3 must be encoded with (rotation, abcd0000) == (0b0000, 0b00000001), not (0b0001, 0b00001100), (0b0010, 0b00110000), or (0b0011, 0b11000000).
In particular, this means that all constants in the range 0-255 are encoded with rotation == 0b0000, and permitted constants outside that range are encoded with rotation != 0b0000. A flag-setting logical instruction with a modified immediate constant therefore leaves APSR.C unchanged if the constant is in the range 0-255 and sets it to the most significant bit of the constant otherwise. This matches the behavior of T32 modified immediate constants for all constants that are permitted in both the A32 and T32 instruction sets.

An alternative syntax is available for a modified immediate constant that permits the programmer to specify the encoding directly. In this syntax, #<const> is instead written as #<byte>, #<rot>, where:

<byte> Is the numeric value of abcdefgh, in the range 0-255.

<rot> Is twice the numeric value of rotation, an even number in the range 0-30.

This syntax permits all A32 data-processing instructions with modified immediate constants to be disassembled to assembler syntax that assembles to the original instruction.

This syntax also makes it possible to write variants of some flag-setting logical instructions that have different effects on APSR.C to those obtained with the normal #<const> syntax. For example, ANDS R1, R2, #12, #2 has the same behavior as ANDS R1, R2, #3 except that it sets APSR.C to 0 instead of leaving it unchanged. Such variants of flag-setting logical instructions do not have equivalents in the T32 instruction set, and Arm deprecates their use.

**Operation of modified immediate constants, A32 instructions**

For an A32 data-processing instruction, the A32ExpandImm() pseudocode function returns the value of the 32-bit immediate constant, calling A32ExpandImm_C() to evaluate the constant.

**Modified immediate constants in T32 and A32 Advanced SIMD instructions**

Table F1-4 on page F1-7245 shows the modified immediate constants available with Advanced SIMD instructions, and how they are encoded.

### Table F1-4 Modified immediate values for Advanced SIMD instructions

<table>
<thead>
<tr>
<th>op</th>
<th>cmode</th>
<th>Constant&lt;sup&gt;a&lt;/sup&gt;</th>
<th>&lt;dt&gt;&lt;sup&gt;b&lt;/sup&gt;</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>000x</td>
<td>00000000 00000000 00000000 00000000 00000000 00000000 abcdefgh</td>
<td>I32</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>001x</td>
<td>00000000 00000000 abcdefgh 00000000 00000000 00000000 abcdefgh 00000000</td>
<td>I32</td>
<td>c, d</td>
</tr>
<tr>
<td></td>
<td>010x</td>
<td>00000000 abcdefgh 00000000 00000000 00000000 abcdefgh 00000000 00000000</td>
<td>I32</td>
<td>c, d</td>
</tr>
<tr>
<td></td>
<td>011x</td>
<td>abcdefgh 00000000 00000000 00000000 abcdefgh 00000000 00000000 00000000</td>
<td>I32</td>
<td>c, d</td>
</tr>
<tr>
<td></td>
<td>100x</td>
<td>00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh</td>
<td>I16</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>101x</td>
<td>abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000</td>
<td>I16</td>
<td>c, d</td>
</tr>
<tr>
<td></td>
<td>1100</td>
<td>00000000 00000000 abcdefgh 11111111 00000000 00000000 abcdefgh 11111111</td>
<td>I32</td>
<td>d, e</td>
</tr>
<tr>
<td></td>
<td>1101</td>
<td>00000000 abcdefgh 11111111 11111111 00000000 abcdefgh 11111111 11111111</td>
<td>I32</td>
<td>d, e</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh</td>
<td>I8</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td>1111</td>
<td>aBbbbbbc defgh00 00000000 00000000 aBbbbbbc defgh00 00000000 00000000</td>
<td>F32</td>
<td>f, g</td>
</tr>
<tr>
<td>1</td>
<td>1110</td>
<td>aaaaaaaaa bbbbbbb bccccccc cddddddd eeeeeee ffffffff ggggggg hhhhhh</td>
<td>I64</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td>1111</td>
<td>UNDEFINED</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

a. In this table, the immediate value is shown in binary form, to relate abcdefgh to the encoding diagram. In assembler syntax, the constant is specified by a data type and a value of that type. That value is specified in the normal way (a decimal number by default) and is replicated enough times to fill the 64-bit immediate. For example, a data type of I32 and a value of 10 specify the 64-bit constant 0x000000000000000A.
Operation of modified immediate constants, Advanced SIMD instructions

For a T32 or A32 Advanced SIMD instruction that uses a modified immediate constant, the operation described by the \texttt{AdvSIMDExpandImm()} pseudocode function returns the value of the 64-bit immediate constant.

Modified immediate constants in T32 and A32 floating-point instructions

Table \texttt{F1-5} on page F1-7246 shows the immediate constants available in the \texttt{VMOV} (immediate) floating-point instruction, and Table \texttt{F1-6} on page F1-7246 shows the resulting floating-point values.

Table \texttt{F1-5} Floating-point modified immediate constants

<table>
<thead>
<tr>
<th>Data type</th>
<th>imm4H</th>
<th>imm4L</th>
<th>Constant a</th>
</tr>
</thead>
<tbody>
<tr>
<td>F16</td>
<td>abcd</td>
<td>efgH</td>
<td>aBbbcdefgh000000</td>
</tr>
<tr>
<td>F32</td>
<td>abcd</td>
<td>efgH</td>
<td>aBBBBBcdefgh000000000000</td>
</tr>
<tr>
<td>F64</td>
<td>abcd</td>
<td>efgH</td>
<td>aBBBBBBBBbccccdefgh00000000000000</td>
</tr>
</tbody>
</table>

a. In this column, \( B = \text{NOT}(b) \). The bit pattern represents the floating-point number \((-1)^{S} \times 2^{\text{exp}} \times \text{mantissa}, \) where \( S = \text{UInt}(a), \text{exp} = \text{UInt}(\text{NOT}(b):c:d)-3 \) and \( \text{mantissa} = (16+\text{UInt}(e:f:g:h))/16 \).

Table \texttt{F1-6} Floating-point constant values

<table>
<thead>
<tr>
<th>\texttt{efgh}</th>
<th>bcd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0001 010 011 100 101 110 111</td>
</tr>
<tr>
<td>2.0 4.0 8.0 16.0 0.125 0.25 0.5 1.0</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>2.125 4.25 8.5 17.0 0.1328125 0.265625 0.53125 1.0625</td>
</tr>
<tr>
<td>0010</td>
<td>2.25 4.5 9.0 18.0 0.140625 0.28125 0.5625 1.125</td>
</tr>
<tr>
<td>0011</td>
<td>2.375 4.75 9.5 19.0 0.1484375 0.296875 0.59375 1.1875</td>
</tr>
<tr>
<td>0100</td>
<td>2.5 5.0 10.0 20.0 0.15625 0.3125 0.625 1.25</td>
</tr>
<tr>
<td>0101</td>
<td>2.625 5.25 10.5 21.0 0.1640625 0.328125 0.65625 1.3125</td>
</tr>
<tr>
<td>0110</td>
<td>2.75 5.5 11.0 22.0 0.171875 0.34375 0.6875 1.375</td>
</tr>
<tr>
<td>0111</td>
<td>2.875 5.75 11.5 23.0 0.1796875 0.359375 0.71875 1.4375</td>
</tr>
<tr>
<td>1000</td>
<td>3.0 6.0 12.0 24.0 0.1875 0.375 0.75 1.5</td>
</tr>
</tbody>
</table>
Table F1-6 Floating-point constant values (continued)

<table>
<thead>
<tr>
<th>efgh</th>
<th>bcd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000</td>
</tr>
<tr>
<td>1001</td>
<td>3.125</td>
</tr>
<tr>
<td>1010</td>
<td>3.25</td>
</tr>
<tr>
<td>1011</td>
<td>3.375</td>
</tr>
<tr>
<td>1100</td>
<td>3.5</td>
</tr>
<tr>
<td>1101</td>
<td>3.625</td>
</tr>
<tr>
<td>1110</td>
<td>3.75</td>
</tr>
<tr>
<td>1111</td>
<td>3.875</td>
</tr>
</tbody>
</table>

**Operation of modified immediate constants, floating-point instructions**

For a T32 or A32 floating-point instruction that uses a modified immediate constant, the operation described by the `VFPExpandImm()` pseudocode function returns the value of the immediate constant.
F1.8 Additional pseudocode support for instruction descriptions

Earlier sections of this chapter include pseudocode that describes features of the execution of A32 and T32 instructions, see:

- Pseudocode description of conditional execution on page F1-7230.
- Pseudocode description of instruction-specified shifts and rotates on page F1-7232

The following subsection gives additional pseudocode support functions for some of the instructions described in Alphabetical list of T32 and A32 base instruction set instructions on page F5-7442. See also Pseudocode support for the banked register transfer instructions on page F5-8163.

F1.8.1 Pseudocode description of operations for System register access instructions

The AArch32.SysRegRead() function obtains the word for an MRC instruction from the System register.

The AArch32.SysRegRead64() function obtains the two words for an MRRC instruction from the System register.

—— Note ———

The relative significance of the two words returned is IMPLEMENTATION DEFINED, but all uses within this manual present the two words in the order (most significant, least significant).

—— ———

The AArch32.SysRegWrite() procedure sends the word for an MCR instruction to the System register.

The AArch32.SysRegWrite64() procedure sends the two words for an MCRR instruction to the System register.

—— Note ———

The relative significance of word2 and word1 is IMPLEMENTATION DEFINED, but all uses within this manual treat word2 as more significant than word1.

F1.8.2 Pseudocode details of system calls

The AArch32.CallSupervisor() pseudocode function generates a Supervisor Call exception. Valid execution of the SVC instruction calls this function.

The AArch32.CallHypervisor() pseudocode function generates an HVC exception. Valid execution of the HVC instruction calls this function.
F1.9 Additional information about Advanced SIMD and floating-point instructions

The following subsections give additional information about the Advanced SIMD and floating-point instructions:

- Advanced SIMD and floating-point instruction syntax.
- The Advanced SIMD addressing mode.
- Advanced SIMD instruction modifiers on page F1-7250.
- Advanced SIMD operand shapes on page F1-7250.
- Data type specifiers on page F1-7251.
- Register specifiers on page F1-7252.
- Register lists on page F1-7253.
- Register encoding on page F1-7253.
- Advanced SIMD scalars on page F1-7254.

——— Note ————

The Advanced SIMD architecture, its associated implementations, and supporting software, are commonly referred to as NEON™ technology.

F1.9.1 Advanced SIMD and floating-point instruction syntax

Advanced SIMD and floating-point instructions use the general conventions of the T32 and A32 instruction sets.

Advanced SIMD and floating-point data-processing instructions use the following general format:

\[ V\{<\text{modifier}>\}\text{<operation>}{\langle\text{shape}\rangle}\{<\text{c}>\}{<\text{q}>}\{.<\text{dt}>\}{<\text{dest}>}, \text{<src1>}, \text{<src2>} \]

All Advanced SIMD and floating-point instructions begin with a \( V \). This distinguishes Advanced SIMD vector and floating-point instructions from scalar instructions.

The main operation is specified in the \( \text{<operation>} \) field. It is usually a three letter mnemonic the same as or similar to the corresponding scalar integer instruction.

The \( \text{<c>} \) and \( \text{<q>} \) fields are standard assembler syntax fields. For details, see Standard assembler syntax fields on page F1-7228.

F1.9.2 The Advanced SIMD addressing mode

All the element and structure load/store instructions use this addressing mode. There is a choice of three formats:

\[ [<Rn>{:<align}>] \]

The address is contained in general-purpose register \( Rn \).

\( Rn \) is not updated by this instruction.

Encoded as \( Rm = \text{0b1111} \).

If \( Rn \) is encoded as \( \text{0b1111} \), the instruction is CONSTRAINED UNPREDICTABLE.

\[ [<Rn>{:<align}>]! \]

The address is contained in general-purpose register \( Rn \).

\( Rn \) is updated by this instruction: \( Rn = Rn + \text{transfer\_size} \)

Encoded as \( Rm = \text{0b1101} \).

\( \text{transfer\_size} \) is the number of bytes transferred by the instruction. This means that, after the instruction is executed, \( Rn \) points to the address in memory immediately following the last address loaded from or stored to.

If \( Rn \) is encoded as \( \text{0b1111} \), the instruction is CONSTRAINED UNPREDICTABLE.

This addressing mode can also be written as:

\[ [<Rn>{:align}], \#<\text{transfer\_size}> \]

However, disassembly produces the \([<Rn>:<align>]!\) form.
The address is contained in general-purpose register \(<Rn>\).

\(Rn\) is updated by this instruction:

\(Rn = Rn + Rm\)

Encoded as \(Rm = Rm\). \(Rm\) must not be encoded as \(0b1111\) or \(0b1101\), the PC or the SP.

If \(Rn\) is encoded as \(0b1111\), the instruction is CONSTRAINED UNPREDICTABLE.

The CONSTRAINED UNPREDICTABLE behavior of encodings where \(Rn\) is \(0b1111\) is described in the section: Using R15 by instruction on page K1-11563.

In all cases, \(<align>\) specifies an alignment, as specified by the individual instruction descriptions.

Previous versions of the manual used the \(@\) character for alignment. So, for example, the first format in this section was shown as \([<Rn>@<align>]\). Both \(@\) and : are supported. However, to ensure portability of code to assemblers that treat \(@\) as a comment character, : is preferred.

### F1.9.3 Advanced SIMD instruction modifiers

The \(<modifier>\) field provides additional variants of some instructions. Table F1-7 on page F1-7250 provides definitions of the modifiers. Modifiers are not available for every instruction.

<table>
<thead>
<tr>
<th>(&lt;modifier&gt;)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>The operation uses saturating arithmetic.</td>
</tr>
<tr>
<td>R</td>
<td>The operation performs rounding.</td>
</tr>
<tr>
<td>D</td>
<td>The operation doubles the result (before accumulation, if any).</td>
</tr>
<tr>
<td>H</td>
<td>The operation halves the result.</td>
</tr>
</tbody>
</table>

### F1.9.4 Advanced SIMD operand shapes

The \(<shape>\) field provides additional variants of some instructions. Table F1-8 on page F1-7250 provides definitions of the shapes. Operand shapes are not available for every instruction.

<table>
<thead>
<tr>
<th>(&lt;shape&gt;)</th>
<th>Meaning</th>
<th>Typical register shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>(none)</td>
<td>The operands and result are all the same width.</td>
<td>Dd, Dn, Dm, Qd, Qn, Qm</td>
</tr>
<tr>
<td>L</td>
<td>Long operation - result is twice the width of both operands</td>
<td>Qd, Dn, Dm</td>
</tr>
<tr>
<td>N</td>
<td>Narrow operation - result is half the width of both operands</td>
<td>Dd, Qn, Dm</td>
</tr>
<tr>
<td>W</td>
<td>Wide operation - result and first operand are twice the width of the second operand</td>
<td>Qd, Qn, Qm</td>
</tr>
</tbody>
</table>

#### Note

- Some assemblers support a Q shape specifier, which requires all operands to be Q registers. An example of using this specifier is \(VADDQ.S32 q0, q1, q2\). This is not standard UAL, and Arm recommends that programmers do not use a Q shape specifier.

- A disassembler must not generate any shape specifier not shown in Table F1-8 on page F1-7250.
F1.9.5 Data type specifiers

The `<dt>` field normally contains one data type specifier. Unless the assembler syntax description for the instruction indicates otherwise, this indicates the data type contained in:

- The second operand, if any.
- The operand, if there is no second operand.
- The result, if there are no operand registers.

The data types of the other operand and result are implied by the `<dt>` field combined with the instruction shape. For information about data type formats, see *Data types supported by the Advanced SIMD implementation* on page E1-7142.

In the instruction syntax descriptions in Chapter F1 *About the T32 and A32 Instruction Descriptions*, the `<dt>` field is usually specified as a single field. However, where more convenient, it is sometimes specified as a concatenation of two fields, `<type><size>`.

Syntax flexibility

There is some flexibility in the data type specifier syntax:

- Software can specify three data types, specifying the result and both operand data types. For example:
  
  ```
  VSUBW.I16.I16.S8 Q3, Q5, D0
  ```

- Software can specify two data types, specifying the data types of the two operands. The data type of the result is implied by the instruction shape. For example:
  
  ```
  VSUBW.I16.S8 Q3, Q5, D0 instead of VSUBW.S8 Q3, Q5, D0
  ```

- Software can specify two data types, specifying the data types of the single operand and the result. For example:
  
  ```
  VMOVN.I16.I32 D0, Q1
  ```

- Where an instruction requires a less specific data type, software can instead specify a more specific type, as shown in *Table F1-9 on page F1-7251*.

- Where an instruction does not require a data type, software can provide one.

- The F32 data type can be abbreviated to F.

- The F64 data type can be abbreviated to D.

In all cases, if software provides additional information, the additional information must match the instruction shape. Disassembly does not regenerate this additional information.

<table>
<thead>
<tr>
<th>Specified data type</th>
<th>Permitted more specific data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Any</td>
</tr>
<tr>
<td>.I&lt;size&gt;</td>
<td>-</td>
</tr>
<tr>
<td>.S&lt;size&gt;</td>
<td>.U&lt;size&gt;</td>
</tr>
<tr>
<td>.32</td>
<td>.I32 .S32 .U32 - .F32 or .F</td>
</tr>
<tr>
<td>.64</td>
<td>.I64 .S64 .U64 - .F64 or .D</td>
</tr>
</tbody>
</table>
**F1.9.6 Register specifiers**

The `<dest>`, `<src1>`, and `<src2>` fields contain register specifiers, or in some cases scalar specifiers or register lists. Table F1-10 on page F1-7252 shows the register and scalar specifier formats that appear in the instruction descriptions.

If `<dest>` is omitted, it is the same as `<src1>`.

<table>
<thead>
<tr>
<th><code>&lt;specifier&gt;</code></th>
<th>Usual meaning a</th>
<th>Used in</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;Qd&gt;</code></td>
<td>A quadword destination register for the result vector.</td>
<td>Advanced SIMD</td>
</tr>
<tr>
<td><code>&lt;Qn&gt;</code></td>
<td>A quadword source register for the first operand vector.</td>
<td>Advanced SIMD</td>
</tr>
<tr>
<td><code>&lt;Qm&gt;</code></td>
<td>A quadword source register for the second operand vector.</td>
<td>Advanced SIMD</td>
</tr>
<tr>
<td><code>&lt;Dd&gt;</code></td>
<td>A doubleword destination register for the result vector.</td>
<td>Both</td>
</tr>
<tr>
<td><code>&lt;Dn&gt;</code></td>
<td>A doubleword source register for the first operand vector.</td>
<td>Both</td>
</tr>
<tr>
<td><code>&lt;Dm&gt;</code></td>
<td>A doubleword source register for the second operand vector.</td>
<td>Both</td>
</tr>
<tr>
<td><code>&lt;Sd&gt;</code></td>
<td>A singleword destination register for the result vector.</td>
<td>Floating-point</td>
</tr>
<tr>
<td><code>&lt;Sn&gt;</code></td>
<td>A singleword source register for the first operand vector.</td>
<td>Floating-point</td>
</tr>
<tr>
<td><code>&lt;Sm&gt;</code></td>
<td>A singleword source register for the second operand vector.</td>
<td>Floating-point</td>
</tr>
<tr>
<td><code>&lt;Dd[x]&gt;</code></td>
<td>A destination scalar for the result. Element x of vector <code>&lt;Dd&gt;</code>.</td>
<td>Advanced SIMD</td>
</tr>
<tr>
<td><code>&lt;Dn[x]&gt;</code></td>
<td>A source scalar for the first operand. Element x of vector <code>&lt;Dn&gt;</code>.</td>
<td>Both b</td>
</tr>
<tr>
<td><code>&lt;Dm[x]&gt;</code></td>
<td>A source scalar for the second operand. Element x of vector <code>&lt;Dm&gt;</code>.</td>
<td>Advanced SIMD</td>
</tr>
<tr>
<td><code>&lt;Rt&gt;</code></td>
<td>A general-purpose register, used for a source or destination address.</td>
<td>Both</td>
</tr>
<tr>
<td><code>&lt;Rt2&gt;</code></td>
<td>A general-purpose register, used for a source or destination address.</td>
<td>Both</td>
</tr>
<tr>
<td><code>&lt;Rn&gt;</code></td>
<td>A general-purpose register, used as a load or store base address.</td>
<td>Both</td>
</tr>
<tr>
<td><code>&lt;Rm&gt;</code></td>
<td>A general-purpose register, used as a post-indexed address source.</td>
<td>Both</td>
</tr>
</tbody>
</table>

a. In some instructions the roles of registers are different.

b. In the floating-point instructions, `<Dn[x]>` is used only in VMOV (scalar to general-purpose register), see *VMOV (scalar to general-purpose register)* on page F6-8554.
**F1.9.7 Register lists**

A register list is a list of register specifiers separated by commas and enclosed in brackets `{` and `}`. There are restrictions on what registers can appear in a register list. These restrictions are described in the individual instruction descriptions. Table F1-11 on page F1-7253 shows some register list formats, with examples of actual register lists corresponding to those formats.

--- **Note** ---

Register lists must not wrap around the end of the register bank.

**Syntax flexibility**

There is some flexibility in the register list syntax:

- Where a register list contains consecutive registers, they can be specified as a range, instead of listing every register, for example `{D0-D3}` instead of `{D0, D1, D2, D3}`.

- Where a register list contains an even number of consecutive doubleword registers starting with an even-numbered register, it can be written as a list of quadword registers instead, for example `{Q1, Q2}` instead of `{D2-D5}`.

- Where a register list contains only one register, the enclosing braces can be omitted, for example `VLD1.8 D0, [R0]` instead of `VLD1.8 {D0}, [R0]`.

**Table F1-11 Example register lists**

<table>
<thead>
<tr>
<th>Format</th>
<th>Example</th>
<th>Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{&lt;Dd&gt;}</code></td>
<td>{D3}</td>
<td>D3</td>
</tr>
<tr>
<td><code>{&lt;Dd&gt;, &lt;Dd+1&gt;, &lt;Dd+2&gt;}</code></td>
<td>{D3, D4, D5}</td>
<td>{D3-D5}</td>
</tr>
<tr>
<td><code>{&lt;Dd[x]&gt;, &lt;Dd+2[x]}</code></td>
<td>{D0[3], D2[3]}</td>
<td>-</td>
</tr>
<tr>
<td><code>{&lt;Dd[]&gt;</code></td>
<td>{D7[0]}</td>
<td>D7[0]</td>
</tr>
</tbody>
</table>

**F1.9.8 Register encoding**

An Advanced SIMD register is either:

- **Quadword**, meaning it is 128 bits wide.
- **Doubleword**, meaning it is 64 bits wide.

Some instructions have options for either doubleword or quadword registers. This is normally encoded in Q, bit[6], as Q = 0 for doubleword operations, or Q = 1 for quadword operations.

A floating-point register is either:

- Double-precision, meaning it is 64 bits wide.
- Single-precision, meaning it is 32 bits wide.

This is encoded in the sz field, bit[8], as sz = 1 for double-precision operations, or sz = 0 for single-precision operations.

The T32 instruction encoding of Advanced SIMD or floating-point registers is:

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Vn</td>
<td>Vd</td>
<td>sz</td>
<td>N</td>
<td>G</td>
<td>M</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The A32 instruction encoding of Advanced SIMD or floating-point registers is:

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Vn</td>
<td>Vd</td>
<td>sz</td>
<td>N</td>
<td>G</td>
<td>M</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Some instructions use only one or two registers, and use the unused register fields as additional opcode bits.

Table F1-12 on page F1-7254 shows the encodings for the registers.

### Table F1-12 Encoding of register numbers

<table>
<thead>
<tr>
<th>Register mnemonic</th>
<th>Usual usage</th>
<th>Register number encoded in</th>
<th>Notes</th>
<th>Used in</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Qd&gt;</td>
<td>Destination (quadword)</td>
<td>D, Vd (bits[22, 15:13])</td>
<td>bit[12] == 0 b</td>
<td>Advanced SIMD</td>
</tr>
<tr>
<td>&lt;Qn&gt;</td>
<td>First operand (quadword)</td>
<td>N, Vn (bits[7, 19:17])</td>
<td>bit[16] == 0 b</td>
<td>Advanced SIMD</td>
</tr>
<tr>
<td>&lt;Qm&gt;</td>
<td>Second operand (quadword)</td>
<td>M, Vm (bits[5, 3:1])</td>
<td>bit[0] == 0 b</td>
<td>Advanced SIMD</td>
</tr>
<tr>
<td>&lt;Dd&gt;</td>
<td>Destination (doubleword)</td>
<td>D, Vd (bits[22, 15:12])</td>
<td>-</td>
<td>Both</td>
</tr>
<tr>
<td>&lt;Dn&gt;</td>
<td>First operand (doubleword)</td>
<td>N, Vn (bits[7, 19:16])</td>
<td>-</td>
<td>Both</td>
</tr>
<tr>
<td>&lt;Dm&gt;</td>
<td>Second operand (doubleword)</td>
<td>M, Vm (bits[5, 3:0])</td>
<td>-</td>
<td>Both</td>
</tr>
<tr>
<td>&lt;Sd&gt;</td>
<td>Destination (single-precision)</td>
<td>Vd, D (bits[15:12, 22])</td>
<td>-</td>
<td>Floating-point</td>
</tr>
<tr>
<td>&lt;Sn&gt;</td>
<td>First operand (single-precision)</td>
<td>Vn, N (bits[19:16, 7])</td>
<td>-</td>
<td>Floating-point</td>
</tr>
<tr>
<td>&lt;Sm&gt;</td>
<td>Second operand (single-precision)</td>
<td>Vm, M (bits[3:0, 5])</td>
<td>-</td>
<td>Floating-point</td>
</tr>
</tbody>
</table>

a. Bit numbers given for the A32 instruction encoding. See the figures in this section for the equivalent bits in the T32 encoding.
b. If this bit is 1, the instruction is UNDEFINED.

### F1.9.9 Advanced SIMD scalars

Advanced SIMD scalars can be 8-bit, 16-bit, 32-bit, or 64-bit. Instructions other than multiply instructions can access any element in the register set. The instruction syntax refers to the scalars using an index into a doubleword vector. The descriptions of the individual instructions contain details of the encodings.

Table F1-13 on page F1-7254 shows the form of encoding for scalars used in multiply instructions. These instructions cannot access scalars in some registers. The descriptions of the individual instructions contain cross references to this section where appropriate.

32-bit Advanced SIMD scalars, when used as single-precision floating-point numbers, are equivalent to Floating-point single-precision registers. That is, \( D_m[x] \) in a 32-bit context (\( 0 <= m <= 15, 0 <= x <= 1 \)) is equivalent to \( S[2m + x] \).

### Table F1-13 Encoding of scalars in multiply instructions

<table>
<thead>
<tr>
<th>Scalar mnemonic</th>
<th>Usual usage</th>
<th>Scalar size</th>
<th>Register specifier</th>
<th>Index specifier</th>
<th>Accessible registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>32-bit</td>
<td>Vm[3:0]</td>
<td>M</td>
<td>D0-D15</td>
</tr>
</tbody>
</table>
Chapter F2
The AArch32 Instruction Sets Overview

This chapter describes the T32 and A32 instruction sets. It contains the following sections:

- Support for instructions in different versions of the Arm architecture on page F2-7256.
- Unified Assembler Language on page F2-7257.
- Branch instructions on page F2-7259.
- Data-processing instructions on page F2-7260.
- PSTATE and banked register access instructions on page F2-7268.
- Load/store instructions on page F2-7269.
- Load/store multiple instructions on page F2-7272.
- Miscellaneous instructions on page F2-7273.
- Exception-generating and exception-handling instructions on page F2-7275.
- System register access instructions on page F2-7277.
- Advanced SIMD and floating-point load/store instructions on page F2-7278.
- Advanced SIMD and floating-point register transfer instructions on page F2-7280.
- Advanced SIMD data-processing instructions on page F2-7281.
- Floating-point data-processing instructions on page F2-7292.
F2.1 Support for instructions in different versions of the Arm architecture

This manual describes the T32 and A32 instruction sets for the architecture. Therefore, it indicates how any options or extensions in the architecture affect the available instructions.
F2.2 Unified Assembler Language

This manual uses the Arm Unified Assembler Language (UAL). This assembly language syntax provides a canonical form for all T32 and A32 instructions.

UAL describes the syntax for the mnemonic and the operands of each instruction. In addition, it assumes that instructions and data items can be given labels. It does not specify the syntax to be used for labels, nor what assembler directives and options are available. See your assembler documentation for these details.

Most earlier Arm assembly language mnemonics are still supported as synonyms, as described in the instruction details.

Note

Most earlier T32 assembly language mnemonics are not supported.

UAL includes instruction selection rules that specify which instruction encoding is selected when more than one can provide the required functionality. For example, both 16-bit and 32-bit encodings exist for an ADD R0, R1, R2 instruction. The most common instruction selection rule is that when both a 16-bit encoding and a 32-bit encoding are available, the 16-bit encoding is selected, to optimize code density.

Syntax options exist to override the normal instruction selection rules and ensure that a particular encoding is selected. These are useful when disassembling code, to ensure that subsequent assembly produces the original code, and in some other situations.

F2.2.1 Conditional instructions

For maximum portability of UAL assembly language between the T32 and A32 instruction sets, Arm recommends that:

- IT instructions are written before conditional instructions in the correct way for the T32 instruction set.
- When assembling to the A32 instruction set, assemblers check that any IT instructions are correct, but do not generate any code for them.

Although other T32 instructions are unconditional, all instructions that are made conditional by an IT instruction must be written with a condition. These conditions must match the conditions imposed by the IT instruction. For example, an ITTEE EQ instruction imposes the EQ condition on the first two following instructions, and the NE condition on the next two. Those four instructions must be written with EQ, EQ, NE and NE conditions respectively.

Some instructions cannot be made conditional by an IT instruction. Some instructions can be conditional if they are the last instruction in the IT block, but not otherwise.

The branch instruction encodings that include a Condition code field cannot be made conditional by an IT instruction. If the assembler syntax indicates a conditional branch that correctly matches a preceding IT instruction, it is assembled using a branch instruction encoding that does not include a Condition code field.

F2.2.2 Use of labels in UAL instruction syntax

The UAL syntax for some instructions includes the label of an instruction or a literal data item that is at a fixed offset from the instruction being specified. The assembler must:

1. Calculate the PC or Align(PC, 4) value of the instruction. The PC value of an instruction is its address plus 4 for a T32 instruction, or plus 8 for an A32 instruction. The Align(PC, 4) value of an instruction is its PC value ANDed with 0xFFFFFFFC to force it to be word-aligned. There is no difference between the PC and Align(PC, 4) values for an A32 instruction, but there can be for a T32 instruction.

2. Calculate the offset from the PC or Align(PC, 4) value of the instruction to the address of the labeled instruction or literal data item.

3. Assemble a PC-relative encoding of the instruction, that is, one that reads its PC or Align(PC, 4) value and adds the calculated offset to form the required address.
Note

For instructions that can encode a subtraction operation, if the instruction cannot encode the calculated offset but can encode minus the calculated offset, the instruction encoding specifies a subtraction of minus the calculated offset.

The syntax of the following instructions includes a label:

- B, BL, and BLX (immediate). The assembler syntax for these instructions always specifies the label of the instruction that they branch to. Their encodings specify a sign-extended immediate offset that is added to the PC value of the instruction to form the target address of the branch.

- CBNZ and CBZ. The assembler syntax for these instructions always specifies the label of the instruction that they branch to. Their encodings specify a zero-extended immediate offset that is added to the PC value of the instruction to form the target address of the branch. They do not support backward branches.

- LDR, LDRB, LDRD, LDRH, LDRSB, LDRSH, PLD, PLDW, PLI, and VLDR. The normal assembler syntax of these load instructions can specify the label of a literal data item that is to be loaded. The encodings of these instructions specify a zero-extended immediate offset that is either added to or subtracted from the Align(PC, 4) value of the instruction to form the address of the data item. A few such encodings perform a fixed addition or a fixed subtraction and must only be used when that operation is required, but most contain a bit that specifies whether the offset is to be added or subtracted.

  When the assembler calculates an offset of 0 for the normal syntax of these instructions, it must assemble an encoding that adds 0 to the Align(PC, 4) value of the instruction. Encodings that subtract 0 from the Align(PC, 4) value cannot be specified by the normal syntax.

  There is an alternative syntax for these instructions that specifies the addition or subtraction and the immediate offset explicitly. In this syntax, the label is replaced by [PC, #+/-<imm>], where:

  +/ - Is + or omitted to specify that the immediate offset is to be added to the Align(PC, 4) value, or if it is to be subtracted.

  <imm> Is the immediate offset.

  This alternative syntax makes it possible to assemble the encodings that subtract 0 from the Align(PC, 4) value, and to disassemble them to a syntax that can be re-assembled correctly.

- ADR. The normal assembler syntax for this instruction can specify the label of an instruction or literal data item whose address is to be calculated. Its encoding specifies a zero-extended immediate offset that is either added to or subtracted from the Align(PC, 4) value of the instruction to form the address of the data item, and some opcode bits that determine whether it is an addition or subtraction.

  When the assembler calculates an offset of 0 for the normal syntax of this instruction, it must assemble the encoding that adds 0 to the Align(PC, 4) value of the instruction. The encoding that subtracts 0 from the Align(PC, 4) value cannot be specified by the normal syntax.

  There is an alternative syntax for this instruction that specifies the addition or subtraction and the immediate value explicitly, by writing them as additions ADD <Rd>, PC, #<imm> or subtractions SUB <Rd>, PC, #<imm>. This alternative syntax makes it possible to assemble the encoding that subtracts 0 from the Align(PC, 4) value, and to disassemble it to a syntax that can be re-assembled correctly.

Note

Arm recommends that where possible, software avoids using:

- The alternative syntax for the ADR, LDC, LDR, LDRD, LDRH, LDRSB, LDRSH, PLD, PLI, PLDW, and VLDR instructions.

- The encodings of these instructions that subtract 0 from the Align(PC, 4) value.
### Branch instructions

Table F2-1 on page F2-7259 summarizes the branch instructions in the T32 and A32 instruction sets. In addition to providing for changes in the flow of execution, some branch instructions can change instruction set.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
<th>Range, T32</th>
<th>Range, A32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch to target address</td>
<td>B on page F5-7491</td>
<td>±16MB</td>
<td>±32MB</td>
</tr>
<tr>
<td>Compare and Branch on Nonzero</td>
<td>CBNZ, CBZ on page F5-7517</td>
<td>0-126 bytes</td>
<td>-(^a)</td>
</tr>
<tr>
<td>Compare and Branch on Zero</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call a subroutine</td>
<td>BL, BLX (immediate) on page F5-7509</td>
<td>±16MB</td>
<td>±32MB</td>
</tr>
<tr>
<td>Call a subroutine, change instruction set(^b)</td>
<td></td>
<td>±16MB</td>
<td>±32MB</td>
</tr>
<tr>
<td>Call a subroutine, optionally change instruction set</td>
<td>BLX (register) on page F5-7512</td>
<td>Any</td>
<td>Any</td>
</tr>
<tr>
<td>Branch to target address, change instruction set</td>
<td>BX on page F5-7514</td>
<td>Any</td>
<td>Any</td>
</tr>
<tr>
<td>Change to Jazelle state</td>
<td>BXJ on page F5-7516</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Table Branch (byte offsets)</td>
<td>TBB, TBH on page F5-8069</td>
<td>0-510 bytes</td>
<td>-(^a)</td>
</tr>
<tr>
<td>Table Branch (halfword offsets)</td>
<td></td>
<td>0-131070 bytes</td>
<td></td>
</tr>
</tbody>
</table>

\(^a\) These instructions do not exist in the A32 instruction set.

\(^b\) The range is determined by the instruction set of the BLX instruction, not of the instruction it branches to.

Branches to loaded and calculated addresses can be performed by LDR, LD\# and data-processing instructions. For details, see Load/store instructions on page F2-7269, Load/store multiple instructions on page F2-7272, Standard data-processing instructions on page F2-7260, and Shift instructions on page F2-7262.

In addition to the branch instructions shown in Table F2-1 on page F2-7259:

- In the A32 instruction set, a data-processing instruction that targets the PC behaves as a branch instruction. For more information, see Data-processing instructions on page F2-7260.

- In the T32 and A32 instruction sets, a load instruction that targets the PC behaves as a branch instruction. For more information, see Load/store instructions on page F2-7269.
F2.4 Data-processing instructions

Core data-processing instructions belong to one of the following groups:

- Standard data-processing instructions.
- Shift instructions on page F2-7262.
- Multiply instructions on page F2-7262.
- Saturating instructions on page F2-7264.
- Saturating addition and subtraction instructions on page F2-7264.
- Packing and unpacking instructions on page F2-7265.
- Parallel addition and subtraction instructions on page F2-7266.
- Divide instructions on page F2-7267.
- Miscellaneous data-processing instructions on page F2-7267.

For related Advanced SIMD and floating-point instructions see Advanced SIMD data-processing instructions on page F2-7281 and Floating-point data-processing instructions on page F2-7292.

F2.4.1 Standard data-processing instructions

These instructions perform basic data-processing operations, and share a common format with some variations.

These instructions generally have a destination register Rd, a first operand register Rn, and a second operand. The second operand can be another register Rm, or an immediate constant.

If the second operand is an immediate constant, it can be:

- Encoded directly in the instruction.
- A modified immediate constant that uses 12 bits of the instruction to encode a range of constants. T32 and A32 instructions have slightly different ranges of modified immediate constants. For more information, see Modified immediate constants in T32 instructions on page F1-7242 and Modified immediate constants in A32 instructions on page F1-7244.

If the second operand is another register, it can optionally be shifted in any of the following ways:

- LSL: Logical Shift Left by 1-31 bits.
- LSR: Logical Shift Right by 1-32 bits.
- ASR: Arithmetic Shift Right by 1-32 bits.
- ROR: Rotate Right by 1-31 bits.
- RRX: Rotate Right with Extend. For details, see Shift and rotate operations on page E1-7130.

In T32 code, the amount to shift by is always a constant encoded in the instruction. In A32 code, the amount to shift by is either a constant encoded in the instruction, or the value of a register, Rs.

For instructions other than CMN, CMP, TEQ, and TST, the result of the data-processing operation is placed in the destination register. In the A32 instruction set, the destination register can be the PC, causing the result to be treated as a branch address. In the T32 instruction set, this is only permitted for some 16-bit forms of the ADD and MOV instructions.

These instructions can optionally set the Condition flags, according to the result of the operation. If they do not set the flags, existing flag settings from a previous instruction are preserved.

Table F2-2 on page F2-7261 summarizes the main data-processing instructions in the T32 and A32 instruction sets. Generally, each of these instructions is described in three sections in Chapter F1 About the T32 and A32 Instruction Descriptions, one section for each of the following:

- INSTRUCTION (immediate) where the second operand is a modified immediate constant.
- INSTRUCTION (register) where the second operand is a register, or a register shifted by a constant.
- INSTRUCTION (register-shifted register) where the second operand is a register shifted by a value obtained from another register. These are only available in the A32 instruction set.
### Table F2-2 Standard data-processing instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mnemonic</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add with Carry</td>
<td>ADC</td>
<td>T32 instruction set permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.</td>
</tr>
<tr>
<td>Add</td>
<td>ADD</td>
<td>T32 instruction set permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.</td>
</tr>
<tr>
<td>Form PC-relative Address</td>
<td>ADR</td>
<td>First operand is the PC. Second operand is an immediate constant. T32 instruction set uses a zero-extended 12-bit immediate constant. Operation is an addition or a subtraction.</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>AND</td>
<td>-</td>
</tr>
<tr>
<td>Bitwise Bit Clear</td>
<td>BIC</td>
<td>-</td>
</tr>
<tr>
<td>Compare Negative</td>
<td>CMN</td>
<td>Sets flags. Like ADD but with no destination register.</td>
</tr>
<tr>
<td>Compare</td>
<td>CMP</td>
<td>Sets flags. Like SUB but with no destination register.</td>
</tr>
<tr>
<td>Bitwise Exclusive OR</td>
<td>EOR</td>
<td>-</td>
</tr>
<tr>
<td>Copy operand to destination</td>
<td>MOV</td>
<td>Has only one operand, with the same options as the second operand in most of these instructions. If the operand is a shifted register, the instruction is an LSL, LSR, ASR, or ROR instruction instead. For details, see Shift instructions on page F2-7262. The T32 and A32 instruction sets permit use of a modified immediate constant or a zero-extended 16-bit immediate constant.</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td>MVN</td>
<td>Has only one operand, with the same options as the second operand in most of these instructions.</td>
</tr>
<tr>
<td>Bitwise OR NOT</td>
<td>ORN</td>
<td>Not available in the A32 instruction set.</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td>ORR</td>
<td>-</td>
</tr>
<tr>
<td>Reverse Subtract</td>
<td>RSB</td>
<td>Subtracts first operand from second operand. This permits subtraction from constants and shifted registers.</td>
</tr>
<tr>
<td>Reverse Subtract with Carry</td>
<td>RSC</td>
<td>Not available in the T32 instruction set.</td>
</tr>
<tr>
<td>Subtract with Carry</td>
<td>SBC</td>
<td>-</td>
</tr>
<tr>
<td>Subtract</td>
<td>SUB</td>
<td>T32 instruction set permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.</td>
</tr>
<tr>
<td>Test Equivalence</td>
<td>TEQ</td>
<td>Sets flags. Like EOR but with no destination register.</td>
</tr>
<tr>
<td>Test</td>
<td>TST</td>
<td>Sets flags. Like AND but with no destination register.</td>
</tr>
</tbody>
</table>
F2.4.2 Shift instructions

Table F2-3 on page F2-7262 lists the shift instructions in the T32 and A32 instruction sets.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic Shift Right</td>
<td><em>ASR (immediate) on page F5-7483</em></td>
</tr>
<tr>
<td></td>
<td><em>ASR (register) on page F5-7485</em></td>
</tr>
<tr>
<td></td>
<td><em>ASRS (immediate) on page F5-7487</em></td>
</tr>
<tr>
<td></td>
<td><em>ASRS (register) on page F5-7489</em></td>
</tr>
<tr>
<td>Logical Shift Left</td>
<td><em>LSL (immediate) on page F5-7691</em></td>
</tr>
<tr>
<td></td>
<td><em>LSL (register) on page F5-7693</em></td>
</tr>
<tr>
<td></td>
<td><em>LSLS (immediate) on page F5-7695</em></td>
</tr>
<tr>
<td></td>
<td><em>LSLS (register) on page F5-7697</em></td>
</tr>
<tr>
<td>Logical Shift Right</td>
<td><em>LSR (immediate) on page F5-7699</em></td>
</tr>
<tr>
<td></td>
<td><em>LSR (register) on page F5-7701</em></td>
</tr>
<tr>
<td></td>
<td><em>LSRS (immediate) on page F5-7703</em></td>
</tr>
<tr>
<td></td>
<td><em>LSRS (register) on page F5-7705</em></td>
</tr>
<tr>
<td>Rotate Right</td>
<td><em>ROR (immediate) on page F5-7833</em></td>
</tr>
<tr>
<td></td>
<td><em>ROR (register) on page F5-7835</em></td>
</tr>
<tr>
<td></td>
<td><em>RORS (immediate) on page F5-7837</em></td>
</tr>
<tr>
<td></td>
<td><em>RORS (register) on page F5-7839</em></td>
</tr>
<tr>
<td>Rotate Right with Extend</td>
<td><em>RRX on page F5-7841</em></td>
</tr>
<tr>
<td></td>
<td><em>RRXS on page F5-7843</em></td>
</tr>
</tbody>
</table>

In the A32 instruction set only, the destination register of these instructions can be the PC, causing the result to be treated as an address to branch to.

F2.4.3 Multiply instructions

These instructions can operate on signed or unsigned quantities. In some types of operation, the results are the same whether the operands are signed or unsigned.

- Table F2-4 on page F2-7262 summarizes the multiply instructions where there is no distinction between signed and unsigned quantities.
  The least significant 32 bits of the result are used. More significant bits are discarded.
- Table F2-5 on page F2-7263 summarizes the signed multiply instructions.
- Table F2-6 on page F2-7263 summarizes the unsigned multiply instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
<th>Operation (number of bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply Accumulate</td>
<td><em>MLA, MLAS on page F5-7711</em></td>
<td>32 = 32 + 32 × 32</td>
</tr>
<tr>
<td>Multiply and Subtract</td>
<td><em>MLS on page F5-7713</em></td>
<td>32 = 32 – 32 × 32</td>
</tr>
<tr>
<td>Multiply</td>
<td><em>MUL, MULS on page F5-7749</em></td>
<td>32 = 32 × 32</td>
</tr>
</tbody>
</table>
### Table F2-5 Signed multiply instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
<th>Operation (number of bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed Multiply Accumulate (halfwords)</td>
<td>SMLABB, SMLABT, SMLATB, SMLATT on page F5-7902</td>
<td>32 = 32 + 16 × 16</td>
</tr>
<tr>
<td>Signed Multiply Accumulate Dual</td>
<td>SMLAD, SMLADX on page F5-7904</td>
<td>32 = 32 + 16 × 16 + 16 × 16</td>
</tr>
<tr>
<td>Signed Multiply Accumulate Long</td>
<td>SMLAL, SMLALS on page F5-7906</td>
<td>64 = 64 + 32 × 32</td>
</tr>
<tr>
<td>Signed Multiply Accumulate Long (halfwords)</td>
<td>SMLALBB, SMLALBT, SMLALTB, SMLALTT on page F5-7908</td>
<td>64 = 64 + 16 × 16</td>
</tr>
<tr>
<td>Signed Multiply Accumulate Long Dual</td>
<td>SMLALD, SMLALDX on page F5-7911</td>
<td>64 = 64 + 16 × 16 + 16 × 16</td>
</tr>
<tr>
<td>Signed Multiply Accumulate (word by halfword)</td>
<td>SMLAWB, SMLAWT on page F5-7914</td>
<td>32 = 32 + 32 × 16</td>
</tr>
<tr>
<td>Signed Multiply Subtract Dual</td>
<td>SMLSD, SMLSDX on page F5-7916</td>
<td>32 = 32 + 16 × 16 – 16 × 16</td>
</tr>
<tr>
<td>Signed Multiply Subtract Long Dual</td>
<td>SMLSLD, SMLSLDX on page F5-7918</td>
<td>64 = 64 + 16 × 16 – 16 × 16</td>
</tr>
<tr>
<td>Signed Most Significant Word Multiply Accumulate</td>
<td>SMMLA, SMMLAR on page F5-7920</td>
<td>32 = 32 + 32 × 32</td>
</tr>
<tr>
<td>Signed Most Significant Word Multiply Subtract</td>
<td>SMMLS, SMMLSR on page F5-7922</td>
<td>32 = 32 – 32 × 32</td>
</tr>
<tr>
<td>Signed Most Significant Word Multiply</td>
<td>SMMUL, SMMULR on page F5-7924</td>
<td>32 = 32 × 32</td>
</tr>
<tr>
<td>Signed Dual Multiply Add</td>
<td>SMUAD, SMUADX on page F5-7926</td>
<td>32 = 16 × 16 + 16 × 16</td>
</tr>
<tr>
<td>Signed Multiply (halfwords)</td>
<td>SMULBB, SMULBT, SMULTB, SMULTT on page F5-7928</td>
<td>32 = 16 × 16</td>
</tr>
<tr>
<td>Signed Multiply Long</td>
<td>SMULL, SMULLS on page F5-7930</td>
<td>64 = 32 × 32</td>
</tr>
<tr>
<td>Signed Multiply (word by halfword)</td>
<td>SMULWB, SMULWT on page F5-7932</td>
<td>32 = 32 × 16</td>
</tr>
<tr>
<td>Signed Dual Multiply Subtract</td>
<td>SMUSD, SMUSDX on page F5-7934</td>
<td>32 = 16 × 16 – 16 × 16</td>
</tr>
</tbody>
</table>

*The most significant 32 bits of the 48-bit product are used. Less significant bits are discarded.*

### Table F2-6 Unsigned multiply instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
<th>Operation (number of bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned Multiply Accumulate Accumulate Long</td>
<td>UMAAL on page F5-8110</td>
<td>64 = 32 + 32 + 32 × 32</td>
</tr>
<tr>
<td>Unsigned Multiply Accumulate Long</td>
<td>UMLAL, UMLALS on page F5-8112</td>
<td>64 = 64 + 32 × 32</td>
</tr>
<tr>
<td>Unsigned Multiply Long</td>
<td>UMULL, UMULLS on page F5-8114</td>
<td>64 = 32 × 32</td>
</tr>
</tbody>
</table>
F2.4.4 Saturating instructions

Table F2-7 on page F2-7264 lists the saturating instructions in the T32 and A32 instruction sets. For more information, see Pseudocode description of saturation on page E1-7131.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed Saturate</td>
<td>SSAT on page F5-7940</td>
<td>Saturates optionally shifted 32-bit value to selected range</td>
</tr>
<tr>
<td>Signed Saturate 16</td>
<td>SSAT16 on page F5-7942</td>
<td>Saturates two 16-bit values to selected range</td>
</tr>
<tr>
<td>Unsigned Saturate</td>
<td>USAT on page F5-8132</td>
<td>Saturates optionally shifted 32-bit value to selected range</td>
</tr>
<tr>
<td>Unsigned Saturate 16</td>
<td>USAT16 on page F5-8134</td>
<td>Saturates two 16-bit values to selected range</td>
</tr>
</tbody>
</table>

F2.4.5 Saturating addition and subtraction instructions

Table F2-8 on page F2-7264 lists the saturating addition and subtraction instructions in the T32 and A32 instruction sets. For more information, see Pseudocode description of saturation on page E1-7131.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saturating Add</td>
<td>QADD on page F5-7802</td>
<td>Add, saturating result to the 32-bit signed integer range</td>
</tr>
<tr>
<td>Saturating Subtract</td>
<td>QSUB on page F5-7816</td>
<td>Subtract, saturating result to the 32-bit signed integer range</td>
</tr>
<tr>
<td>Saturating Double and Add</td>
<td>QADD on page F5-7802</td>
<td>Doubles one value and adds a second value, saturating the doubling and the addition to the 32-bit signed integer range</td>
</tr>
<tr>
<td>Saturating Double and Subtract</td>
<td>QDSUB on page F5-7812</td>
<td>Doubles one value and subtracts the result from a second value, saturating the doubling and the subtraction to the 32-bit signed integer range</td>
</tr>
</tbody>
</table>
### F2.4.6 Packing and unpacking instructions

Table F2-9 on page F2-7265 lists the packing and unpacking instructions in the T32 and A32 instruction sets.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pack Halfword</td>
<td>PKHBT, PKHTB on page F5-7773</td>
<td>Combine halfwords</td>
</tr>
<tr>
<td>Signed Extend and Add Byte</td>
<td>SXTAB on page F5-8057</td>
<td>Extend 8 bits to 32 and add</td>
</tr>
<tr>
<td>Signed Extend and Add Byte 16</td>
<td>SXTAB16 on page F5-8059</td>
<td>Dual extend 8 bits to 16 and add</td>
</tr>
<tr>
<td>Signed Extend and Add Halfword</td>
<td>SXTAH on page F5-8061</td>
<td>Extend 16 bits to 32 and add</td>
</tr>
<tr>
<td>Signed Extend Byte</td>
<td>SXTB on page F5-8063</td>
<td>Extend 8 bits to 32</td>
</tr>
<tr>
<td>Signed Extend Byte 16</td>
<td>SXTB16 on page F5-8065</td>
<td>Dual extend 8 bits to 16</td>
</tr>
<tr>
<td>Signed Extend Halfword</td>
<td>SXTH on page F5-8067</td>
<td>Extend 16 bits to 32</td>
</tr>
<tr>
<td>Unsigned Extend and Add Byte</td>
<td>UXTAB on page F5-8142</td>
<td>Extend 8 bits to 32 and add</td>
</tr>
<tr>
<td>Unsigned Extend and Add Byte 16</td>
<td>UXTAB16 on page F5-8144</td>
<td>Dual extend 8 bits to 16 and add</td>
</tr>
<tr>
<td>Unsigned Extend and Add Halfword</td>
<td>UXTAH on page F5-8146</td>
<td>Extend 16 bits to 32 and add</td>
</tr>
<tr>
<td>Unsigned Extend Byte</td>
<td>UXTB on page F5-8148</td>
<td>Extend 8 bits to 32</td>
</tr>
<tr>
<td>Unsigned Extend Byte 16</td>
<td>UXTB16 on page F5-8150</td>
<td>Dual extend 8 bits to 16</td>
</tr>
<tr>
<td>Unsigned Extend Halfword</td>
<td>UXTH on page F5-8152</td>
<td>Extend 16 bits to 32</td>
</tr>
</tbody>
</table>
**F2.4.7 Parallel addition and subtraction instructions**

These instructions perform additions and subtractions on the values of two registers and write the result to a destination register, treating the register values as sets of two halfwords or four bytes. That is, they perform SIMD additions or subtractions on the general-purpose registers.

These instructions consist of a prefix followed by a main instruction mnemonic. The prefixes are as follows:
- *S* Signed arithmetic modulo $2^8$ or $2^{16}$.
- *Q* Signed saturating arithmetic.
- *SH* Signed arithmetic, halving the results.
- *U* Unsigned arithmetic modulo $2^8$ or $2^{16}$.
- *UQ* Unsigned saturating arithmetic.
- *UH* Unsigned arithmetic, halving the results.

The main instruction mnemonics are as follows:

- **ADD16** Adds the top halfwords of two operands to form the top halfword of the result, and the bottom halfwords of the same two operands to form the bottom halfword of the result.
- **ASX** Exchanges halfwords of the second operand, and then adds top halfwords and subtracts bottom halfwords.
- **SAX** Exchanges halfwords of the second operand, and then subtracts top halfwords and adds bottom halfwords.
- **SUB16** Subtracts each halfword of the second operand from the corresponding halfword of the first operand to form the corresponding halfword of the result.
- **ADD8** Adds each byte of the second operand to the corresponding byte of the first operand to form the corresponding byte of the result.
- **SUB8** Subtracts each byte of the second operand from the corresponding byte of the first operand to form the corresponding byte of the result.

The instruction set permits all 36 combinations of prefix and main instruction operand, as Table F2-10 on page F2-7266 shows.

See also *Advanced SIMD parallel addition and subtraction* on page F2-7282.

<table>
<thead>
<tr>
<th>Main instruction</th>
<th>Signed</th>
<th>Saturating</th>
<th>Signed halving</th>
<th>Unsigned</th>
<th>Unsigned saturating</th>
<th>Unsigned halving</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD16, add, two halfwords</td>
<td>SADD16</td>
<td>QADD16</td>
<td>SHAADD16</td>
<td>UADD16</td>
<td>UQADD16</td>
<td>UHADD16</td>
</tr>
<tr>
<td>ASX, add and subtract with exchange</td>
<td>SASX</td>
<td>QASX</td>
<td>SHAASX</td>
<td>USAX</td>
<td>UQASX</td>
<td>UHASX</td>
</tr>
<tr>
<td>SAX, subtract and add with exchange</td>
<td>SSAX</td>
<td>QSAX</td>
<td>SHSAX</td>
<td>USAX</td>
<td>UQSAX</td>
<td>UHSAX</td>
</tr>
<tr>
<td>SUB16, subtract, two halfwords</td>
<td>SSUB16</td>
<td>QSUB16</td>
<td>SHSUB16</td>
<td>SUB16</td>
<td>UQSUB16</td>
<td>UHSUB16</td>
</tr>
<tr>
<td>ADD8, add, four bytes</td>
<td>SADD8</td>
<td>QADD8</td>
<td>SHAADD8</td>
<td>UADD8</td>
<td>UQADD8</td>
<td>UHADD8</td>
</tr>
<tr>
<td>SUB8, subtract, four bytes</td>
<td>SSUB8</td>
<td>QSUB8</td>
<td>SHSUB8</td>
<td>SUB8</td>
<td>UQSUB8</td>
<td>UHSUB8</td>
</tr>
</tbody>
</table>
F2.4.8 Divide instructions

Signed and unsigned integer divide instructions are included in both the T32 instruction set and the A32 instruction set.

For descriptions of the instructions see:
• SDIV on page F5-7878.
• UDIV on page F5-8096.

For the SDIV and UDIV instructions, division by zero always returns a zero result.

The ID_ISAR0.Divide_instrs field indicates the level of support for these instructions. The field value of 0b0010 indicates they are implemented in both the T32 and A32 instruction sets.

F2.4.9 Miscellaneous data-processing instructions

Table F2-11 on page F2-7267 lists the miscellaneous data-processing instructions in the T32 and A32 instruction sets. Immediate values in these instructions are simple binary numbers.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitField Clear</td>
<td>BFC on page F5-7494</td>
<td>-</td>
</tr>
<tr>
<td>BitField Insert</td>
<td>BFI on page F5-7496</td>
<td>-</td>
</tr>
<tr>
<td>Count Leading Zeros</td>
<td>CLZ on page F5-7519</td>
<td>-</td>
</tr>
<tr>
<td>Move Top</td>
<td>MOVT on page F5-7728</td>
<td>Moves 16-bit immediate value to top halfword. Bottom halfword unchanged.</td>
</tr>
<tr>
<td>Reverse Bits</td>
<td>RBIT on page F5-7822</td>
<td>-</td>
</tr>
<tr>
<td>Byte-Reverse Word</td>
<td>REV on page F5-7824</td>
<td>-</td>
</tr>
<tr>
<td>Byte-Reverse Packed Halfword</td>
<td>REV16 on page F5-7826</td>
<td>-</td>
</tr>
<tr>
<td>Byte-Reverse Signed Halfword</td>
<td>REVSH on page F5-7828</td>
<td>-</td>
</tr>
<tr>
<td>Signed BitField Extract</td>
<td>SBFX on page F5-7876</td>
<td>-</td>
</tr>
<tr>
<td>Select Bytes using GE flags</td>
<td>SEL on page F5-7880</td>
<td>-</td>
</tr>
<tr>
<td>Unsigned BitField Extract</td>
<td>UBFX on page F5-8092</td>
<td>-</td>
</tr>
<tr>
<td>Unsigned Sum of Absolute Differences</td>
<td>USAD8 on page F5-8128</td>
<td>-</td>
</tr>
<tr>
<td>Unsigned Sum of Absolute Differences and Accumulate</td>
<td>USADA8 on page F5-8130</td>
<td>-</td>
</tr>
</tbody>
</table>
F2.5 PSTATE and banked register access instructions

These instructions transfer PE state information to or from a general-purpose register.

F2.5.1 PSTATE access instructions

PSTATE holds process state information, see Process state, PSTATE on page E1-7133. In AArch32 state:

- At EL1 or higher, PSTATE is accessible using the Current Program Status Register (CPSR).
- At EL0, a subset of the CPSR is accessible as the Application Program Status Register (APSR).
- On taking an exception, the contents of the CPSR are copied to the Saved Program Status Register (SPSR) of the mode from which the exception is taken.

The MRS and MSR instructions move the contents of the CPSR, APSR, or the SPSR of the current mode to or from a general-purpose register, see:

- MRS on page F5-7734.
- MSR (immediate) on page F5-7744.
- MSR (register) on page F5-7746.

When executed at EL0, MRS and MSR instructions can only access the APSR.

The PSTATE Condition flags, PSTATE.\{N, Z, C, V\} are set by the execution of data-processing instructions, and can control the execution of conditional instructions. However, software can set the Condition flags explicitly using the MSR instruction, and can read the current state of the Condition flags explicitly using the MRS instruction.

In addition, at EL1 or higher, software can use the CPS instruction to change the PSTATE.M field and the PSTATE.\{A, I, F\} interrupt mask bits, see CPS, CPSID, CPSIE on page F5-7535.

F2.5.2 Banked register access instructions

At EL1 or higher, the MRS (banked register) and MSR (banked register) instructions move the contents of a banked general-purpose register, the SPSR, or the ELR_hyp, to or from a general-purpose register. See:

- MRS (Banked register) on page F5-7736.
- MSR (Banked register) on page F5-7740.
F2.6 Load/store instructions

Table F2-12 on page F2-7269 summarizes the general-purpose register load/store instructions in the T32 and A32 instruction sets. Some of these instructions can also operate on the PC. See also:

- *Load/store multiple instructions* on page F2-7272.
- *Synchronization and semaphores* on page E2-7211, for more information about the Load-Exclusive and Store-Exclusive instructions.
- *Advanced SIMD and floating-point load/store instructions* on page F2-7278.

Load/store instructions have several options for addressing memory. For more information, see *Addressing modes* on page F2-7270.

### Table F2-12 Load/store instructions

<table>
<thead>
<tr>
<th>Data type</th>
<th>Load</th>
<th>Store</th>
<th>Unprivileged</th>
<th>Exclusive</th>
<th>Load-</th>
<th>Store-</th>
<th>Load-</th>
<th>Store-</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Unprivileged</td>
<td>Exclusive</td>
<td>Acquire</td>
<td>Release</td>
<td>Acquire</td>
<td>Release</td>
</tr>
<tr>
<td>32-bit word</td>
<td>LDR</td>
<td>STR</td>
<td>LDRRT</td>
<td>STRT</td>
<td>LDREX</td>
<td>STREX</td>
<td>LDA</td>
<td>STL</td>
</tr>
<tr>
<td>16-bit halfword</td>
<td>-</td>
<td>STRH</td>
<td>-</td>
<td>STRHT</td>
<td>-</td>
<td>STREXH</td>
<td>LDAH</td>
<td>STLH</td>
</tr>
<tr>
<td>16-bit unsigned</td>
<td>LDRH</td>
<td>-</td>
<td>LDRHT</td>
<td>-</td>
<td>LDREXH</td>
<td>-</td>
<td>LDAH</td>
<td>STLH</td>
</tr>
<tr>
<td>16-bit signed</td>
<td>LDRSH</td>
<td>-</td>
<td>LDRSHT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8-bit byte</td>
<td>-</td>
<td>STRB</td>
<td>-</td>
<td>STRBT</td>
<td>-</td>
<td>STREXB</td>
<td>LDAB</td>
<td>STLB</td>
</tr>
<tr>
<td>8-bit unsigned</td>
<td>LDRB</td>
<td>-</td>
<td>LDRBT</td>
<td>-</td>
<td>LDREXB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8-bit signed</td>
<td>LDRSB</td>
<td>-</td>
<td>LDRSBT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Two 32-bit words</td>
<td>LDRD</td>
<td>STRD</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>64-bit doubleword</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>LDRXDP</td>
<td>STREXD</td>
<td>-</td>
<td>LDAEXD</td>
<td>STLEXD</td>
</tr>
</tbody>
</table>

#### F2.6.1 Loads to the PC

The LDR instruction can load a value into the PC. The value loaded is treated as an interworking address, as described by the `LoadWritePC()` pseudocode function in *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-7133.

#### F2.6.2 Halfword and byte loads and stores

Halfword and byte stores store the least significant halfword or byte from the register, to 16 or 8 bits of memory respectively. There is no distinction between signed and unsigned stores.

Halfword and byte loads load 16 or 8 bits from memory into the least significant halfword or byte of a register. Unsigned loads zero-extend the loaded value to 32 bits, and signed loads sign-extend the value to 32 bits.
F2.6.3 Load unprivileged and Store unprivileged

When executing at EL0, a Load unprivileged or Store unprivileged instruction operates in exactly the same way as the corresponding ordinary load or store instruction. For example, an \texttt{LDRT} instruction executes in exactly the same way as the equivalent \texttt{LDR} instruction. When executed at PL1, Load unprivileged and Store unprivileged instructions behave as they would if they were executed at EL0. For example, an \texttt{LDRT} instruction executes in exactly the way that the equivalent \texttt{LDR} instruction would execute at EL0. In particular, the instructions make unprivileged memory accesses.

--- Note ---

As described in \textit{Security state, Exception levels, and AArch32 execution privilege on page G1-8908}, execution at PL1 describes all of the following:

- Execution at Non-secure EL1 using AArch32.
- Execution at Secure EL1 using AArch32 when EL3 is not implemented.
- Execution at Secure EL1 using AArch32 when EL3 is implemented and is using AArch64.
- Execution at Secure EL3 when EL3 is implemented and is using AArch32.

The Load unprivileged and Store unprivileged instructions are \textbf{CONSTRAINED UNPREDICTABLE} if executed at EL2.

For more information about execution privilege, see \textit{About access permissions on page G5-9192}.

F2.6.4 Load-Exclusive and Store-Exclusive

Load-Exclusive and Store-Exclusive instructions provide shared memory synchronization. For more information, see \textit{Synchronization and semaphores on page E2-7211}.

F2.6.5 Load-Acquire and Store-Release

Load-Acquire and Store-Release instructions provide memory barriers. Load-Acquire Exclusive and Store-Release Exclusive instructions provide memory barriers with shared memory synchronization. For more information, see \textit{Load-Acquire, Store-Release on page E2-7186}.

F2.6.6 Addressing modes

The address for a load or store is formed from two parts: a value from a base register, and an offset.

The base register can be any one of the general-purpose registers R0-R12, SP, or LR.

For loads, the base register can be the PC. This provides PC-relative addressing for position-independent code. Instructions marked (literal) in their title in Chapter F1 \textit{About the T32 and A32 Instruction Descriptions} are PC-relative loads.

The offset takes one of three formats:

- **Immediate**: The offset is an unsigned number that can be added to or subtracted from the base register value. Immediate offset addressing is useful for accessing data elements that are a fixed distance from the start of the data object, such as structure fields, stack offsets, and input/output registers.

- **Register**: The offset is a value from a general-purpose register. The value can be added to, or subtracted from, the base register value. Register offsets are useful for accessing arrays or blocks of data.

- **Scaled register**: The offset is a general-purpose register, shifted by an immediate value, then added to or subtracted from the base register. This means an array index can be scaled by the size of each array element.

The offset and base register can be used in three different ways to form the memory address. The addressing modes are described as follows:

- **Offset**: The offset is added to or subtracted from the base register to form the memory address.
Pre-indexed  The offset is added to or subtracted from the base register to form the memory address. The base register is then updated with this new address, to permit automatic indexing through an array or memory block.

Post-indexed  The value of the base register alone is used as the memory address. The offset is then added to or subtracted from the base register. The result is stored back in the base register, to permit automatic indexing through an array or memory block.

Note  Not every variant is available for every instruction, and the range of permitted immediate values and the options for scaled registers vary from instruction to instruction. See Chapter F1 About the T32 and A32 Instruction Descriptions for full details for each instruction.
F2.7 Load/store multiple instructions

Load Multiple instructions load from memory a subset, or possibly all, of the general-purpose registers and the PC.

Store Multiple instructions store to memory a subset, or possibly all, of the general-purpose registers.

The memory locations are consecutive word-aligned words. The addresses used are obtained from a base register, and can be either above or below the value in the base register. The base register can optionally be updated by the total size of the data transferred.

Table F2-13 on page F2-7272 summarizes the load/store multiple instructions in the T32 and A32 instruction sets.

### Table F2-13 Load/store multiple instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Multiple, Increment After or Full Descending</td>
<td>LDM, LDMIA, LDMFD on page F5-7600</td>
</tr>
<tr>
<td>Load Multiple, Decrement After or Full Ascending</td>
<td>LDMDA, LDMEF on page F5-7608</td>
</tr>
<tr>
<td>Load Multiple, Decrement Before or Empty Ascending</td>
<td>LDMDB, LDMEA on page F5-7610</td>
</tr>
<tr>
<td>Load Multiple, Increment Before or Empty Descending</td>
<td>LDMIB, LDMEB on page F5-7613</td>
</tr>
<tr>
<td>Pop multiple registers off the stack</td>
<td>POP on page F5-7789</td>
</tr>
<tr>
<td>Push multiple registers onto the stack</td>
<td>PUSH on page F5-7797</td>
</tr>
<tr>
<td>Store Multiple, Increment After or Empty Ascending</td>
<td>STM, STMIA, STMEA on page F5-7972</td>
</tr>
<tr>
<td>Store Multiple, Decrement After or Empty Descending</td>
<td>STMDB, STMED on page F5-7980</td>
</tr>
<tr>
<td>Store Multiple, Decrement Before or Full Descending</td>
<td>STMIB, STMF on page F5-7983</td>
</tr>
</tbody>
</table>

a. Not available in the T32 instruction set.
b. This instruction is equivalent to an LDM instruction with the SP as base register, and base register updating.
c. This instruction is equivalent to an STMDB instruction with the SP as base register, and base register updating.

When executing at EL1, variants of the LDM and STM instructions load and store User mode registers. Another system level variant of the LDM instruction performs an exception return.

F2.7.1 Loads to the PC

The LDM, LDMDA, LDMDB, LDMIB, and POP instructions can load a value into the PC. The value loaded is treated as an interworking address, as described by the LoadWritePC() pseudocode function in Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
F2.8 Miscellaneous instructions

Table F2-14 on page F2-7273 summarizes the miscellaneous instructions in the T32 and A32 instruction sets.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear-Exclusive</td>
<td>CLREX on page F5-7518</td>
</tr>
<tr>
<td>Data Memory Barrier</td>
<td>DMB on page F5-7555</td>
</tr>
<tr>
<td>Data Synchronization Barrier</td>
<td>DSB on page F5-7558</td>
</tr>
<tr>
<td>Error Synchronization Barrier</td>
<td>ESB on page F5-7572</td>
</tr>
<tr>
<td>Instruction Synchronization Barrier</td>
<td>ISB on page F5-7578</td>
</tr>
<tr>
<td>If-Then</td>
<td>IT on page F5-7580</td>
</tr>
<tr>
<td>No Operation</td>
<td>NOP on page F5-7758</td>
</tr>
<tr>
<td>Preload Data</td>
<td>PLD, PLDW (immediate) on page F5-7776</td>
</tr>
<tr>
<td></td>
<td>PLD (literal) on page F5-7779</td>
</tr>
<tr>
<td></td>
<td>PLD, PLDW (register) on page F5-7781</td>
</tr>
<tr>
<td>Preload Instruction</td>
<td>PLI (immediate, literal) on page F5-7784</td>
</tr>
<tr>
<td></td>
<td>PLI (register) on page F5-7787</td>
</tr>
<tr>
<td>Speculation Barrier</td>
<td>SB on page F5-7865</td>
</tr>
<tr>
<td>Set Endianness</td>
<td>SETEND on page F5-7882a</td>
</tr>
<tr>
<td>Set Privileged Access Never</td>
<td>SETPAN on page F5-7883</td>
</tr>
<tr>
<td>Send Event</td>
<td>SEV on page F5-7884</td>
</tr>
<tr>
<td>Send Event Local</td>
<td>SEVL on page F5-7886</td>
</tr>
<tr>
<td>Wait For Event</td>
<td>WFE on page F5-8154</td>
</tr>
<tr>
<td>Wait For Interrupt</td>
<td>WFI on page F5-8156</td>
</tr>
<tr>
<td>Yield</td>
<td>YIELD on page F5-8158b</td>
</tr>
</tbody>
</table>

Note

Previous versions of the architecture defined the DBG instruction, that could provide a hint to the debug system, in this group. From the introduction of Armv8, this instruction executes as a NOP. Arm deprecates any use of the DBG instruction.

F2.8.1 The Yield instruction

In a Symmetric Multithreading (SMT) design, a thread can use the YIELD instruction to give a hint to the PE that it is running on. The YIELD hint indicates that whatever the thread is currently doing is of low importance, and so could yield. For example, the thread might be sitting in a spin-lock. A similar use might be in modifying the arbitration priority of the snoop bus in a multiprocessor (MP) system. Defining such an instruction permits binary compatibility between SMT and SMP systems.

AArch32 state defines a YIELD instruction as a specific NOP (No Operation) hint instruction.
The `YIELD` instruction has no effect in a single-threaded system, but developers of such systems can use the instruction to flag its intended use on migration to a multiprocessor or multithreading system. Operating systems can use `YIELD` in places where a yield hint is wanted, knowing that it will be treated as a `NOP` if there is no implementation benefit.
F2.9 Exception-generating and exception-handling instructions

The following instructions are intended specifically to cause a synchronous exception to occur:

- The SVC instruction generates a Supervisor Call exception. For more information, see Supervisor Call (SVC) exception on page G1-8968.
- The Breakpoint instruction BKPT provides software breakpoints. For more information, see Breakpoint Instruction exceptions on page G2-9051.
- In an implementation that includes EL3 the SMC instruction generates a Secure Monitor Call exception. For more information, see Secure Monitor Call (SMC) exception on page G1-8969.
- In an implementation that includes EL2 the HVC instruction generates a Hypervisor Call exception. For more information, see Hypervisor Call (HVC) exception on page G1-8970.

Debug state on page F2-7276 summarizes the Debug state instructions.

For an exception taken to an EL1 mode:

- The system level variants of the SUBS and LDM instructions can perform a return from an exception.

  Note
  The variants of SUBS include MOV. See the references to Subtract (exception return) on page F2-7275, Move (exception return) on page F2-7275, and Load Multiple (exception return) on page F2-7275 in Table F2-15 on page F2-7275 for more information.

- The SRS instruction can be used near the start of the handler, to store return information. The RFE instruction can then perform a return from the exception using the stored return information.

In an implementation that includes EL2, the ERET instruction performs a return from an exception taken to Hyp mode.

For more information, see Exception return to an Exception level using AArch32 on page G1-8951.

Table F2-15 on page F2-7275 summarizes the instructions, in the T32 and A32 instruction sets, for generating or handling an exception. Except for BKPT and SVC, these are system level instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervisor Call</td>
<td>SVC on page F5-8055</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>BKPT on page F5-7507</td>
</tr>
<tr>
<td>Secure Monitor Call</td>
<td>SMC on page F5-7900</td>
</tr>
<tr>
<td>Return From Exception</td>
<td>RFE, RFEDA, RFEDB, RFEIA, RFEIB on page F5-7830</td>
</tr>
<tr>
<td>Subtract (exception return)a</td>
<td>SUB, SUBS (immediate) on page F5-8039a</td>
</tr>
<tr>
<td>Move (exception return)a</td>
<td>MOV, MOVX (register) on page F5-7719a</td>
</tr>
<tr>
<td>Hypervisor Call</td>
<td>HVC on page F5-7576</td>
</tr>
<tr>
<td>Exception Return</td>
<td>ERET on page F5-7570</td>
</tr>
<tr>
<td>Load Multiple (exception return)</td>
<td>LDM (exception return) on page F5-7604</td>
</tr>
<tr>
<td>Store Return State</td>
<td>SRS, SRSDA, SRSDB, SRSIA, SRSIB on page F5-7936</td>
</tr>
</tbody>
</table>

a. The A32 instruction set includes other instruction forms that can be used for an exception return, that have previously been described as variants of SUBS PC, LR. Arm deprecates any use of these instruction forms.
## F2.9.1 Debug state

Table F2-16 on page F2-7276 shows the Debug state instructions that are implemented in the T32 instruction set:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCPSn</td>
<td>Debug switch to ELn</td>
<td><a href="#">DCPS1 on page F5-7549</a> <a href="#">DCPS2 on page F5-7551</a> <a href="#">DCPS3 on page F5-7553</a></td>
<td>-</td>
</tr>
<tr>
<td>ERET</td>
<td>Debug restore PE state (DRPS)</td>
<td><a href="#">ERET on page F5-7570</a></td>
<td>When executed in Debug state, the T1 encoding of ERET performs the DRPS operation</td>
</tr>
</tbody>
</table>

Table F2-16 T32 Debug state instructions
F2.10 System register access instructions

The System register encoding space is indexed using the parameters {coproc, opc1, CRn, CRm, opc2}, see The AArch32 System register interface on page G1-8993. This encoding space provides System registers and System instructions. The only permitted values of coproc are 0b1110 and 0b1111, and the following instructions give access to this encoding space:

- Instructions that transfer data between general-purpose registers and System registers. See:
  - MCR on page F5-7707.
  - MCRR on page F5-7709.
  - MRC on page F5-7730.
  - MRRC on page F5-7732.

- Instructions that load or store from memory to a System register. See:
  - LDC (immediate) on page F5-7596.
  - LDC (literal) on page F5-7598.
  - STC on page F5-7952.

Note

The System register encoding space with coproc==0b101x is redefined to provide some of the Advanced SIMD and floating-point functionality. That is, to:

- Initiate a floating-point data-processing operation, see Floating-point data-processing instructions on page F2-7292.
- Transfer data between general-purpose registers and the Advanced SIMD and floating-point registers, see Advanced SIMD and floating-point register transfer instructions on page F2-7280.
- Load or store data to the Advanced SIMD and floating-point registers, see Advanced SIMD and floating-point load/store instructions on page F2-7278.

System register access instructions are part of the instruction stream executed by the PE, and therefore any System register access instruction that cannot be executed by the implementation causes an Undefined Instruction exception. In the A-profile and the R-profile architectures, the instruction encodings in the System register access instruction encoding space are unallocated, and generate Undefined Instruction exceptions, except for:

- The instructions summarized in this section that access the coproc==0b111x encoding space.
- The instructions in the coproc==0b101x encoding space that are redefined to provide Advanced SIMD and floating-point functionality, as summarized in the Note in this section.
F2.11 Advanced SIMD and floating-point load/store instructions

Table F2-17 on page F2-7278 summarizes the SIMD and floating-point register file load/store instructions in the Advanced SIMD and floating-point instruction sets.

Advanced SIMD also provides instructions for loading and storing multiple elements, or structures of elements, see *Element and structure load/store instructions*.

### Table F2-17 SIMD and floating-point register file load/store instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Load Multiple</td>
<td>VLDM, VLDMDM, VLDMIA on page F6-8476</td>
<td>Load 1-16 consecutive 64-bit registers, Advanced SIMD and floating-point.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Load 1-16 consecutive 32-bit registers, floating-point only.</td>
</tr>
<tr>
<td>Vector Load Register</td>
<td>VLDR (immediate) on page F6-8481</td>
<td>Load one 64-bit register, Advanced SIMD and floating-point.</td>
</tr>
<tr>
<td></td>
<td>VLDR (literal) on page F6-8484</td>
<td>Load one 32-bit register, floating-point only.</td>
</tr>
<tr>
<td>Vector Store Multiple</td>
<td>VSTM, VSTMDB, VSTMIA on page F6-8842</td>
<td>Store 1-16 consecutive 64-bit registers, Advanced SIMD and floating-point.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Store 1-16 consecutive 32-bit registers, floating-point only.</td>
</tr>
<tr>
<td>Vector Store Register</td>
<td>VSTR on page F6-8847</td>
<td>Store one 64-bit register, Advanced SIMD and floating-point.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Store one 32-bit register, floating-point only.</td>
</tr>
</tbody>
</table>

F2.11.1 Element and structure load/store instructions

Table F2-18 on page F2-7278 shows the element and structure load/store instructions available in the Advanced SIMD instruction set. Loading and storing structures of more than one element automatically de-interleaves or interleaves the elements, see *Figure F2-1 on page F2-7279* for an example of de-interleaving. Interleaving is the inverse process.

### Table F2-18 Element and structure load/store instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load single element</td>
<td></td>
</tr>
<tr>
<td>Multiple elements</td>
<td>VLD1 (multiple single elements) on page F6-8429</td>
</tr>
<tr>
<td>To one lane</td>
<td>VLD1 (single element to one lane) on page F6-8421</td>
</tr>
<tr>
<td>To all lanes</td>
<td>VLD1 (single element to all lanes) on page F6-8426</td>
</tr>
<tr>
<td>Load 2-element structure</td>
<td></td>
</tr>
<tr>
<td>Multiple structures</td>
<td>VLD2 (multiple 2-element structures) on page F6-8445</td>
</tr>
<tr>
<td>To one lane</td>
<td>VLD2 (single 2-element structure to one lane) on page F6-8436</td>
</tr>
<tr>
<td>To all lanes</td>
<td>VLD2 (single 2-element structure to all lanes) on page F6-8442</td>
</tr>
<tr>
<td>Load 3-element structure</td>
<td></td>
</tr>
<tr>
<td>Multiple structures</td>
<td>VLD3 (multiple 3-element structures) on page F6-8459</td>
</tr>
<tr>
<td>To one lane</td>
<td>VLD3 (single 3-element structure to one lane) on page F6-8450</td>
</tr>
<tr>
<td>To all lanes</td>
<td>VLD3 (single 3-element structure to all lanes) on page F6-8456</td>
</tr>
</tbody>
</table>
### Table F2-18 Element and structure load/store instructions (continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load 4-element structure</td>
<td></td>
</tr>
<tr>
<td>Multiple structures</td>
<td><em>VLD4 (multiple 4-element structures)</em> on page F6-8472</td>
</tr>
<tr>
<td>To one lane</td>
<td><em>VLD4 (single 4-element structure to one lane)</em> on page F6-8462</td>
</tr>
<tr>
<td>To all lanes</td>
<td><em>VLD4 (single 4-element structure to all lanes)</em> on page F6-8468</td>
</tr>
<tr>
<td>Store single element</td>
<td></td>
</tr>
<tr>
<td>Multiple elements</td>
<td><em>VST1 (multiple single elements)</em> on page F6-8803</td>
</tr>
<tr>
<td>From one lane</td>
<td><em>VST1 (single element from one lane)</em> on page F6-8798</td>
</tr>
<tr>
<td>Store 2-element structure</td>
<td></td>
</tr>
<tr>
<td>Multiple structures</td>
<td><em>VST2 (multiple 2-element structures)</em> on page F6-8817</td>
</tr>
<tr>
<td>From one lane</td>
<td><em>VST2 (single 2-element structure from one lane)</em> on page F6-8811</td>
</tr>
<tr>
<td>Store 3-element structure</td>
<td></td>
</tr>
<tr>
<td>Multiple structures</td>
<td><em>VST3 (multiple 3-element structures)</em> on page F6-8828</td>
</tr>
<tr>
<td>From one lane</td>
<td><em>VST3 (single 3-element structure from one lane)</em> on page F6-8822</td>
</tr>
<tr>
<td>Store 4-element structure</td>
<td></td>
</tr>
<tr>
<td>Multiple structures</td>
<td><em>VST4 (multiple 4-element structures)</em> on page F6-8838</td>
</tr>
<tr>
<td>From one lane</td>
<td><em>VST4 (single 4-element structure from one lane)</em> on page F6-8831</td>
</tr>
</tbody>
</table>

Figure F2-1 shows the de-interleaving of a VLD3.16 (multiple 3-element structures) instruction:

![De-interleaving diagram](image)

**Figure F2-1 De-interleaving an array of 3-element structures**

Figure F2-1 shows the VLD3.16 instruction operating to three 64-bit registers that comprise four 16-bit elements:

- Different instructions in this group would produce similar figures, but operate on different numbers of registers. For example, VLD4 and VST4 instructions operate on four registers.
- Different element sizes would produce similar figures but with 8-bit or 32-bit elements.
- These instructions operate only on doubleword (64-bit) registers.
F2.12 Advanced SIMD and floating-point register transfer instructions

Table F2-19 on page F2-7280 summarizes the SIMD and floating-point register file transfer instructions in the Advanced SIMD and floating-point instruction sets. These instructions transfer data between the general-purpose registers and the registers in the SIMD and floating-point register file.

Advanced SIMD vectors, and single-precision and double-precision floating-point registers, are all views of the same register file. For details, see The SIMD and floating-point register file on page E1-7140.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy element from general-purpose register to every element of an Advanced SIMD vector</td>
<td>VDUP (general-purpose register) on page F6-8370</td>
</tr>
<tr>
<td>Copy byte, halfword, or word from general-purpose register to a register in the SIMD and floating-point register file</td>
<td>VMOV (general-purpose register to scalar) on page F6-8550</td>
</tr>
<tr>
<td>Copy byte, halfword, or word from a register in the SIMD and floating-point register file to a general-purpose register</td>
<td>VMOV (scalar to general-purpose register) on page F6-8554</td>
</tr>
<tr>
<td>Copy from half-precision floating-point register to general-purpose register, or from general-purpose register to half-precision floating-point register Only supported if FEAT_FP16 is implemented</td>
<td>VMOV (between general-purpose register and half-precision) on page F6-8537</td>
</tr>
<tr>
<td>Copy from single-precision floating-point register to general-purpose register, or from general-purpose register to single-precision floating-point register</td>
<td>VMOV (between general-purpose register and single-precision) on page F6-8552</td>
</tr>
<tr>
<td>Copy two words from general-purpose registers to consecutive single-precision floating-point registers, or from consecutive single-precision floating-point registers to general-purpose registers</td>
<td>VMOV (between two general-purpose registers and two single-precision registers) on page F6-8556</td>
</tr>
<tr>
<td>Copy two words from general-purpose registers to a doubleword register in the SIMD and floating-point register file, or from a doubleword register in the SIMD and floating-point register file to general-purpose registers</td>
<td>VMOV (between two general-purpose registers and a doubleword floating-point register) on page F6-8535</td>
</tr>
<tr>
<td>Copy from an Advanced SIMD and floating-point System Register to a general-purpose register</td>
<td>VMRS on page F6-8565</td>
</tr>
<tr>
<td>Copy from a general-purpose register to an Advanced SIMD and floating-point System Register</td>
<td>VMSR on page F6-8568</td>
</tr>
</tbody>
</table>
F2.13 Advanced SIMD data-processing instructions

Advanced SIMD data-processing instructions process registers containing vectors of elements of the same type packed together, enabling the same operation to be performed on multiple items in parallel.

Instructions operate on vectors held in 64-bit or 128-bit registers. Figure F2-2 shows an operation on two 64-bit operand vectors, generating a 64-bit vector result.

Note

Figure F2-2 and other similar figures show 64-bit vectors that consist of four 16-bit elements, and 128-bit vectors that consist of four 32-bit elements. Other element sizes produce similar figures, but with 1, 2, 8, or 16 operations performed in parallel instead of 4.

Many Advanced SIMD instructions have variants that produce vectors of elements double the size of the inputs. In this case, the number of elements in the result vector is the same as the number of elements in the operand vectors, but each element, and the whole vector, is double the size.

Figure F2-3 shows an example of an Advanced SIMD instruction operating on 64-bit registers, and generating a 128-bit result.

There are also Advanced SIMD instructions that have variants that produce vectors containing elements half the size of the inputs. Figure F2-4 on page F2-7282 shows an example of an Advanced SIMD instruction operating on one 128-bit register, and generating a 64-bit result.
Some Advanced SIMD instructions do not conform to these standard patterns. Their operation patterns are described in the individual instruction descriptions.

Advanced SIMD instructions that perform floating-point arithmetic use the Arm standard floating-point arithmetic defined in Floating-point support on page A1-55.

The following sections summarize the Advanced SIMD data-processing instructions:

- Advanced SIMD parallel addition and subtraction.
- Bitwise Advanced SIMD data-processing instructions on page F2-7283.
- Advanced SIMD comparison instructions on page F2-7284.
- Advanced SIMD shift instructions on page F2-7285.
- Advanced SIMD multiply instructions on page F2-7286.
- Advanced SIMD dot product instructions on page F2-7287.
- Miscellaneous Advanced SIMD data-processing instructions on page F2-7289.
- Advanced SIMD BFloat16 instructions on page F2-7288.
- Advanced SIMD matrix multiply instructions on page F2-7288.
- The Cryptographic Extension in AArch32 state on page F2-7290.

### F2.13.1 Advanced SIMD parallel addition and subtraction

Table F2-20 on page F2-7282 shows the Advanced SIMD parallel add and subtract instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
</table>
| Vector Add                                       | VADD (integer) on page F6-8231  
                                                    | VADD (floating-point) on page F6-8227 |
| Vector Add and Narrow, returning High Half       | VADDHN on page F6-8233          |
| Vector Add Long                                  | VADDL on page F6-8235           |
| Vector Add Wide                                  | VADDW on page F6-8237           |
| Vector Halving Add                               | VHADD on page F6-8411           |
| Vector Halving Subtract                          | VHSUB on page F6-8414           |
| Vector Pairwise Add and Accumulate Long          | VPADAL on page F6-8615          |
| Vector Pairwise Add                              | VPADD (integer) on page F6-8620 |
| Vector Pairwise Add Long                         | VPADD (floating-point) on page F6-8618 |
| Vector Pairwise Add Long                         | VPADDL on page F6-8622          |
| Vector Rounding Add and Narrow, returning High Half | VRADDHN on page F6-8694      |

Figure F2-4 Advanced SIMD instruction producing narrower result


### Table F2-20 Advanced SIMD parallel add and subtract instructions (continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Rounding Halving Add</td>
<td>VRHADD on page F6-8709</td>
</tr>
<tr>
<td>Vector Rounding Subtract and Narrow, returning High Half</td>
<td>VRSUBHN on page F6-8757</td>
</tr>
<tr>
<td>Vector Saturating Add</td>
<td>VQADD on page F6-8639</td>
</tr>
<tr>
<td>Vector Saturating Subtract</td>
<td>VQSUB on page F6-8692</td>
</tr>
<tr>
<td>Vector Subtract</td>
<td>VSUB (integer) on page F6-8854</td>
</tr>
<tr>
<td></td>
<td>VSUB (floating-point) on page F6-8850</td>
</tr>
<tr>
<td>Vector Subtract and Narrow, returning High Half</td>
<td>VSUBHN on page F6-8856</td>
</tr>
<tr>
<td>Vector Subtract Long</td>
<td>VSUBL on page F6-8858</td>
</tr>
<tr>
<td>Vector Subtract Wide</td>
<td>VSUBW on page F6-8860</td>
</tr>
</tbody>
</table>

### F2.13.2 Bitwise Advanced SIMD data-processing instructions

Table F2-21 on page F2-7283 shows bitwise Advanced SIMD data-processing instructions. These operate on the doubleword (64-bit) or quadword (128-bit) registers in the SIMD and floating-point register file, and there is no division into vector elements.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Bitwise AND</td>
<td>VAND (register) on page F6-8242</td>
</tr>
<tr>
<td>Vector Bitwise Bit Clear (AND complement)</td>
<td>VBIC (immediate) on page F6-8244</td>
</tr>
<tr>
<td></td>
<td>VBIC (register) on page F6-8247</td>
</tr>
<tr>
<td>Vector Bitwise Exclusive OR</td>
<td>VEOR on page F6-8374</td>
</tr>
<tr>
<td>Vector Bitwise Insert if False</td>
<td>VBIT on page F6-8251</td>
</tr>
<tr>
<td>Vector Bitwise Insert if True</td>
<td>VBIT on page F6-8251</td>
</tr>
<tr>
<td>Vector Bitwise Move</td>
<td>VMOV (immediate) on page F6-8539</td>
</tr>
<tr>
<td></td>
<td>VMOV (register) on page F6-8546</td>
</tr>
<tr>
<td>Vector Bitwise NOT</td>
<td>VMVN (immediate) on page F6-8586</td>
</tr>
<tr>
<td></td>
<td>VMVN (register) on page F6-8590</td>
</tr>
<tr>
<td>Vector Bitwise OR</td>
<td>VORR (immediate) on page F6-8610</td>
</tr>
<tr>
<td></td>
<td>VORR (register) on page F6-8613</td>
</tr>
<tr>
<td>Vector Bitwise OR NOT</td>
<td>VORN (register) on page F6-8608</td>
</tr>
<tr>
<td>Vector Bitwise Select</td>
<td>VBSL on page F6-8253</td>
</tr>
</tbody>
</table>
### Advanced SIMD comparison instructions

Table F2-22 on page F2-7284 shows Advanced SIMD comparison instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Absolute Compare Greater Than or Equal</td>
<td>$VACGE$ on page F6-8217</td>
</tr>
<tr>
<td>Vector Absolute Compare Greater Than</td>
<td>$VACGT$ on page F6-8222</td>
</tr>
<tr>
<td>Vector Compare Equal</td>
<td>$VCEQ$ (register) on page F6-8260</td>
</tr>
<tr>
<td>Vector Compare Equal to Zero</td>
<td>$VCEQ$ (immediate #0) on page F6-8258</td>
</tr>
<tr>
<td>Vector Compare Greater Than or Equal</td>
<td>$VCGE$ (register) on page F6-8267</td>
</tr>
<tr>
<td>Vector Compare Greater Than or Equal to Zero</td>
<td>$VCGE$ (immediate #0) on page F6-8264</td>
</tr>
<tr>
<td>Vector Compare Greater Than</td>
<td>$VCGT$ (register) on page F6-8274</td>
</tr>
<tr>
<td>Vector Compare Greater Than Zero</td>
<td>$VCGT$ (immediate #0) on page F6-8271</td>
</tr>
<tr>
<td>Vector Compare Less Than or Equal to Zero</td>
<td>$VCLE$ (immediate #0) on page F6-8278</td>
</tr>
<tr>
<td>Vector Compare Less Than Zero</td>
<td>$VCLT$ (immediate #0) on page F6-8286</td>
</tr>
<tr>
<td>Vector Test Bits</td>
<td>$VTST$ on page F6-8872</td>
</tr>
</tbody>
</table>
### Advanced SIMD shift instructions

Table F2-23 on page F2-7285 lists the shift instructions in the Advanced SIMD instruction set.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Saturating Rounding Shift Left</td>
<td>VQRSHL on page F6-8670</td>
</tr>
<tr>
<td>Vector Saturating Rounding Shift Right and Narrow</td>
<td>VQRSHRN, VQRSHRUN on page F6-8675</td>
</tr>
<tr>
<td>Vector Saturating Shift Left</td>
<td>VQSHL (register) on page F6-8683</td>
</tr>
<tr>
<td></td>
<td>VQSHL, VQSHLU (immediate) on page F6-8680</td>
</tr>
<tr>
<td>Vector Saturating Shift Right and Narrow</td>
<td>VQSHRN, VQSHRUN on page F6-8687</td>
</tr>
<tr>
<td>Vector Rounding Shift Left</td>
<td>VRSHL on page F6-8738</td>
</tr>
<tr>
<td>Vector Rounding Shift Right</td>
<td>VRSHR on page F6-8741</td>
</tr>
<tr>
<td>Vector Rounding Shift Right and Accumulate</td>
<td>VRSRA on page F6-8754</td>
</tr>
<tr>
<td>Vector Rounding Shift Right and Narrow</td>
<td>VRSHRN on page F6-8746</td>
</tr>
<tr>
<td>Vector Shift Left</td>
<td>VSHL (immediate) on page F6-8767</td>
</tr>
<tr>
<td></td>
<td>VSHL (register) on page F6-8770</td>
</tr>
<tr>
<td>Vector Shift Left Long</td>
<td>VSHLL on page F6-8773</td>
</tr>
<tr>
<td>Vector Shift Right</td>
<td>VSHR on page F6-8776</td>
</tr>
<tr>
<td>Vector Shift Right and Narrow</td>
<td>VSHRN on page F6-8776</td>
</tr>
<tr>
<td>Vector Shift Left and Insert</td>
<td>VSLI on page F6-8785</td>
</tr>
<tr>
<td>Vector Shift Right and Accumulate</td>
<td>VSRRA on page F6-8792</td>
</tr>
<tr>
<td>Vector Shift Right and Insert</td>
<td>VSRI on page F6-8795</td>
</tr>
</tbody>
</table>
F2.13.5 Advanced SIMD multiply instructions

Table F2-24 on page F2-7286 shows the Advanced SIMD multiply instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply Accumulate</td>
<td>VMLA (integer) on page F6-8511</td>
</tr>
<tr>
<td></td>
<td>VMLA (floating-point) on page F6-8507</td>
</tr>
<tr>
<td></td>
<td>VMLA (by scalar) on page F6-8513</td>
</tr>
<tr>
<td>Vector Multiply Accumulate Long</td>
<td>VMLAL (integer) on page F6-8516</td>
</tr>
<tr>
<td></td>
<td>VMLAL (by scalar) on page F6-8518</td>
</tr>
<tr>
<td>Vector Multiply Subtract</td>
<td>VMLS (integer) on page F6-8524</td>
</tr>
<tr>
<td></td>
<td>VMLS (floating-point) on page F6-8520</td>
</tr>
<tr>
<td></td>
<td>VMLS (by scalar) on page F6-8526</td>
</tr>
<tr>
<td>Vector Multiply Subtract Long</td>
<td>VMLSL (integer) on page F6-8529</td>
</tr>
<tr>
<td></td>
<td>VMLSL (by scalar) on page F6-8531</td>
</tr>
<tr>
<td>Vector Multiply Long</td>
<td>VMUL (integer and polynomial) on page F6-8575</td>
</tr>
<tr>
<td></td>
<td>VMUL (floating-point) on page F6-8571</td>
</tr>
<tr>
<td></td>
<td>VMUL (by scalar) on page F6-8578</td>
</tr>
<tr>
<td>Vector Multiply Long</td>
<td>VMULL (integer and polynomial) on page F6-8581</td>
</tr>
<tr>
<td></td>
<td>VMULL (by scalar) on page F6-8584</td>
</tr>
<tr>
<td>Vector Fused Multiply Accumulate</td>
<td>VFMA on page F6-8381</td>
</tr>
<tr>
<td>Vector Floating-Point Multiply-Add Long</td>
<td>VFMAL (vector) on page F6-8389</td>
</tr>
<tr>
<td></td>
<td>VFMAL (by scalar) on page F6-8392</td>
</tr>
<tr>
<td>Vector Fused Multiply Subtract</td>
<td>VFMS on page F6-8395</td>
</tr>
<tr>
<td>Vector Floating-Point Multiply-Subtract Long</td>
<td>VFMSL (vector) on page F6-8399</td>
</tr>
<tr>
<td></td>
<td>VFMSL (by scalar) on page F6-8402</td>
</tr>
<tr>
<td>Vector Saturating Doubling Multiply Accumulate Long</td>
<td>VQDMLAL on page F6-8641</td>
</tr>
<tr>
<td>Vector Saturating Doubling Multiply Subtract Long</td>
<td>VQDMLSL on page F6-8644</td>
</tr>
<tr>
<td>Vector Saturating Doubling Multiply Returning High Half</td>
<td>VQDMULH on page F6-8647</td>
</tr>
<tr>
<td>Vector Saturating Doubling Multiply Long</td>
<td>VQDMULL on page F6-8651</td>
</tr>
<tr>
<td>Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half</td>
<td>VQRDMLAH on page F6-8658</td>
</tr>
<tr>
<td>Vector Saturating Rounding Doubling Multiply Subtract Returning High Half</td>
<td>VQRDMLSH on page F6-8662</td>
</tr>
<tr>
<td>Vector Saturating Rounding Doubling Multiply Returning High Half</td>
<td>VQRDMULH on page F6-8666</td>
</tr>
</tbody>
</table>

Advanced SIMD multiply instructions can operate on vectors of:
- 8-bit, 16-bit, or 32-bit unsigned integers.
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit polynomials over \{0, 1\}. VMUL and VMULL are the only instructions that operate on polynomials. VMULL produces a 16-bit polynomial over \{0, 1\}.
- Single-precision (32-bit) or half-precision (16-bit) floating-point numbers.
They can also act on one vector and one scalar.

Long instructions have doubleword (64-bit) operands, and produce quadword (128-bit) results. Other Advanced SIMD multiply instructions can have either doubleword or quadword operands, and produce results of the same size.

Floating-point multiply instructions can operate on:

- Half-precision (16-bit) floating-point numbers.
- Single-precision (32-bit) floating-point numbers.
- Double-precision (64-bit) floating-point numbers.

### F2.13.6 Advanced SIMD dot product instructions

**FEAT_DotProd** provides SIMD instructions that perform the dot product of the four 8-bit subelements of the 32-bit elements of one vector with the four 8-bit subelements of a second vector. It provides two forms of the instructions, each with signed and unsigned versions:

**Vector form**

The dot product is calculated for each element of the first vector with the corresponding element of the second vector.

**Indexed form**

The dot product is calculated for each element of the first vector with the element of the second vector that is indicated by the index argument to the instruction.

---

**Note**

That is, a single element from the second vector is used, and the dot product is calculated between each element of the first vector and this single element from the second vector.

Table F2-25 on page F2-7287 shows the Advanced SIMD dot product instructions.

#### Table F2-25 Advanced SIMD dot product instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSDOT</td>
<td>Signed dot product (vector form)</td>
<td><em>VSDOT (vector)</em> on page F6-8761</td>
</tr>
<tr>
<td>VUDOT</td>
<td>Unsigned dot product (vector form)</td>
<td><em>VUDOT (vector)</em> on page F6-8876</td>
</tr>
<tr>
<td>VSDOT</td>
<td>Signed dot product (indexed form)</td>
<td><em>VSDOT (by element)</em> on page F6-8759</td>
</tr>
<tr>
<td>VSUDOT</td>
<td>Mixed sign integer dot product by indexed quadruplet(^a)</td>
<td><em>VSUDOT (by element)</em> on page F6-8862</td>
</tr>
<tr>
<td>VUDOT</td>
<td>Unsigned dot product (indexed form)</td>
<td><em>VUDOT (by element)</em> on page F6-8874</td>
</tr>
<tr>
<td>VUSDOT</td>
<td>Mixed sign integer dot product (vector format)(^a)</td>
<td><em>VUSDOT (vector)</em> on page F6-8882</td>
</tr>
<tr>
<td></td>
<td>Mixed sign integer dot product by indexed quadruplet(^a)</td>
<td><em>VUSDOT (by element)</em> on page F6-8880</td>
</tr>
</tbody>
</table>

\(^a\) This instruction is only supported when FEAT_AA32I8MM is implemented.

---

### F2.13.7 Advanced SIMD complex number arithmetic instructions

**FEAT_FCMA** provides AArch32 Advanced SIMD instructions that perform arithmetic on complex numbers held in element pairs in vector registers, where the less significant element of the pair contains the real component and the more significant element contains the imaginary component.

These instructions provide single-precision versions. If **FEAT_FP16** is implemented they also provide half-precision versions, otherwise the half-precision encodings are **UNDEFINED**.
Table F2-26 on page F2-7288 shows the FEAT_FCMA AArch32 Advanced SIMD instructions.

**Table F2-26 Advanced SIMD complex number arithmetic instructions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCADD</td>
<td>Floating-point complex add</td>
<td>VCADD on page F6-8255</td>
</tr>
<tr>
<td>VCMILA</td>
<td>Floating-point complex multiply accumulate (vector form)</td>
<td>VCMILA on page F6-8294</td>
</tr>
<tr>
<td>VCMIL</td>
<td>Floating-point complex multiply accumulate (indexed form)</td>
<td>VCMILA (by element) on page F6-8297</td>
</tr>
</tbody>
</table>

A pair of VCMILA instructions can be used to perform a complex number multiplication. In Complex multiplication on page K12-11746, this is demonstrated for the similar AArch64 instruction FCMLA. The usage of VCMILA in this manner is identical.

### F2.13.8 Advanced SIMD BFloat16 instructions

When FEAT_AA32BF16 is implemented, BFloat16 instructions are available in AArch32 state.

Table F2-27 on page F2-7288 shows the Advanced SIMD BFloat16 instructions.

**Table F2-27 BFloat16 Advanced SIMD instructions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDOT</td>
<td>BFloat16 floating-point vector dot product (vector and by scalar formats)</td>
<td>VDOT (vector) on page F6-8366 VDOT (by element) on page F6-8368</td>
</tr>
<tr>
<td>VMMLA</td>
<td>BFloat16 floating-point matrix multiply-accumulate</td>
<td>VMMLA on page F6-8533</td>
</tr>
<tr>
<td>VFMAB, VFMAT</td>
<td>BFloat16 floating-point widening multiply-add long (vector and by scalar formats)</td>
<td>VFMAB, VFMAT (BFloat16, vector) on page F6-8385 VFMAB, VFMAT (BFloat16, by scalar) on page F6-8387</td>
</tr>
<tr>
<td>VCVT</td>
<td>BFloat16 convert from single-precision to BF16 format</td>
<td>VCVT (from single-precision to BFloat16, Advanced SIMD) on page F6-8310</td>
</tr>
</tbody>
</table>

### F2.13.9 Advanced SIMD matrix multiply instructions

When FEAT_AA32I8MM is implemented, these instructions are available in AArch32 state. They include integer and mixed sign dot product instructions.

The matrix multiply-accumulate instructions delimit source and destination vectors into segments. Within each segment:
- The first source vector matrix is organized in row-by-row order.
- The second source vector matrix elements are organized in a column-by-column order.
- The destination vector matrix is organized in row-by-row order.

One matrix multiplication is performed per segment.

Table F2-28 on page F2-7288 shows the Advanced SIMD matrix multiply instructions.

**Table F2-28 Matrix multiply Advanced SIMD instructions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSMLLA</td>
<td>Widening 8-bit signed integer matrix multiply-accumulate into 2x2 matrix</td>
<td>VSMLLA on page F6-8788</td>
</tr>
<tr>
<td>VUMMLA</td>
<td>Widening 8-bit unsigned integer matrix multiply-accumulate into 2x2 matrix</td>
<td>VUMMLA on page F6-8878</td>
</tr>
<tr>
<td>VUSMLLA</td>
<td>Widening 8-bit mixed sign integer matrix multiply-accumulate into 2x2 matrix</td>
<td>VUSMLLA on page F6-8884</td>
</tr>
</tbody>
</table>
### F2.13.10 Miscellaneous Advanced SIMD data-processing instructions

Table F2-29 on page F2-7289 shows miscellaneous Advanced SIMD data-processing instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Absolute Difference and Accumulate</td>
<td>\textit{VABA} on page F6-8201</td>
</tr>
<tr>
<td>Vector Absolute Difference and Accumulate Long</td>
<td>\textit{VABAL} on page F6-8204</td>
</tr>
<tr>
<td>Vector Absolute Difference</td>
<td>\textit{VABD (integer)} on page F6-8208</td>
</tr>
<tr>
<td></td>
<td>\textit{VABD (floating-point)} on page F6-8206</td>
</tr>
<tr>
<td>Vector Absolute Difference Long</td>
<td>\textit{VABDL (integer)} on page F6-8211</td>
</tr>
<tr>
<td>Vector Absolute</td>
<td>\textit{VABS} on page F6-8213</td>
</tr>
<tr>
<td>Vector Convert between floating-point and fixed point</td>
<td>\textit{VCVT (between floating-point and fixed-point, Advanced SIMD)} on page F6-8326</td>
</tr>
<tr>
<td>Vector Convert between floating-point and integer</td>
<td>\textit{VCVT (between floating-point and integer, Advanced SIMD)} on page F6-8316</td>
</tr>
<tr>
<td>Vector Convert between half-precision and single-precision</td>
<td>\textit{VCVT (between half-precision and single-precision, Advanced SIMD)} on page F6-8314</td>
</tr>
<tr>
<td>Vector Count Leading Sign Bits</td>
<td>\textit{VCLS} on page F6-8284</td>
</tr>
<tr>
<td>Vector Count Leading Zeros</td>
<td>\textit{VCLZ} on page F6-8292</td>
</tr>
<tr>
<td>Vector Count Set Bits</td>
<td>\textit{VCNT} on page F6-8308</td>
</tr>
<tr>
<td>Vector Duplicate scalar</td>
<td>\textit{VDUP (scalar)} on page F6-8372</td>
</tr>
<tr>
<td>Vector Extract</td>
<td>\textit{VEXT (byte elements)} on page F6-8376</td>
</tr>
<tr>
<td>Vector move Insertion</td>
<td>\textit{VINS} on page F6-8417</td>
</tr>
<tr>
<td>Vector Move and Narrow</td>
<td>\textit{VMOVN} on page F6-8561</td>
</tr>
<tr>
<td>Vector Move Long</td>
<td>\textit{VMOVL} on page F6-8559</td>
</tr>
<tr>
<td>Vector Move extraction</td>
<td>\textit{VMOVX} on page F6-8563</td>
</tr>
<tr>
<td>Vector Maximum</td>
<td>\textit{VMAX (integer)} on page F6-8490</td>
</tr>
<tr>
<td></td>
<td>\textit{VMAX (floating-point)} on page F6-8487</td>
</tr>
<tr>
<td>Vector Minimum</td>
<td>\textit{VMIN (integer)} on page F6-8500</td>
</tr>
<tr>
<td></td>
<td>\textit{VMIN (floating-point)} on page F6-8497</td>
</tr>
<tr>
<td>Vector Negate</td>
<td>\textit{VNEG} on page F6-8592</td>
</tr>
<tr>
<td>Vector Pairwise Maximum</td>
<td>\textit{VPMAX (integer)} on page F6-8627</td>
</tr>
<tr>
<td></td>
<td>\textit{VPMAX (floating-point)} on page F6-8625</td>
</tr>
<tr>
<td>Vector Pairwise Minimum</td>
<td>\textit{VPMIN (integer)} on page F6-8631</td>
</tr>
<tr>
<td></td>
<td>\textit{VPMIN (floating-point)} on page F6-8629</td>
</tr>
<tr>
<td>Vector Reciprocal Estimate</td>
<td>\textit{VRECPE} on page F6-8696</td>
</tr>
<tr>
<td>Vector Reciprocal Step</td>
<td>\textit{VRECPS} on page F6-8698</td>
</tr>
<tr>
<td>Vector Reciprocal Square Root Estimate</td>
<td>\textit{VRSQRTE} on page F6-8750</td>
</tr>
</tbody>
</table>
The AArch32 Instruction Sets Overview
F2.13 Advanced SIMD data-processing instructions

Table F2-29 Miscellaneous Advanced SIMD data-processing instructions (continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Reciprocal Square Root Step</td>
<td>VRSQRTS on page F6-8752</td>
</tr>
<tr>
<td>Vector Reverse in halfwords</td>
<td>VREV16 on page F6-8700</td>
</tr>
<tr>
<td>Vector Reverse in words</td>
<td>VREV32 on page F6-8703</td>
</tr>
<tr>
<td>Vector Reverse in doublewords</td>
<td>VREV64 on page F6-8706</td>
</tr>
<tr>
<td>Vector Saturating Absolute</td>
<td>VQABS on page F6-8637</td>
</tr>
<tr>
<td>Vector Saturating Move and Narrow</td>
<td>VQMOVN, VQMOVUN on page F6-8654</td>
</tr>
<tr>
<td>Vector Saturating Negate</td>
<td>VQNEG on page F6-8656</td>
</tr>
<tr>
<td>Vector Swap</td>
<td>VSWP on page F6-8864</td>
</tr>
<tr>
<td>Vector Table Lookup</td>
<td>VTBL, VTBX on page F6-8866</td>
</tr>
<tr>
<td>Vector Transpose</td>
<td>VTRN on page F6-8869</td>
</tr>
<tr>
<td>Vector Unzip</td>
<td>VUZP on page F6-8886</td>
</tr>
<tr>
<td>Vector Zip</td>
<td>VZIP on page F6-8890</td>
</tr>
</tbody>
</table>

F2.13.11 The Cryptographic Extension in AArch32 state

The instructions provided by the optional Cryptographic Extension use the Advanced SIMD and floating-point register file. For more information about the functions they provide see:
- Announcing the Advanced Encryption Standard.
- The Galois/Counter Mode of Operation.
- Announcing the Secure Hash Standard.

Table F2-30 on page F2-7290 shows the AArch32 Cryptographic Extension instructions.

Table F2-30 AArch32 Cryptographic Extension instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>AESD</td>
<td>AES single round decryption</td>
<td>AESD on page F6-8167</td>
</tr>
<tr>
<td>AESE</td>
<td>AES single round encryption</td>
<td>AESE on page F6-8169</td>
</tr>
<tr>
<td>AESIMC</td>
<td>AES inverse mix columns</td>
<td>AESIMC on page F6-8171</td>
</tr>
<tr>
<td>AESMC</td>
<td>AES mix columns</td>
<td>AESMC on page F6-8173</td>
</tr>
<tr>
<td>VMULL</td>
<td>Polynomial multiply long</td>
<td>VMULL (integer and polynomial) on page F6-8581a</td>
</tr>
<tr>
<td>SHA1C</td>
<td>SHA1 hash update (choose)</td>
<td>SHA1C on page F6-8181</td>
</tr>
<tr>
<td>SHA1H</td>
<td>SHA1 fixed rotate</td>
<td>SHA1H on page F6-8183</td>
</tr>
<tr>
<td>SHA1M</td>
<td>SHA1 hash update (majority)</td>
<td>SHA1M on page F6-8185</td>
</tr>
<tr>
<td>SHA1P</td>
<td>SHA1 hash update (parity)</td>
<td>SHA1P on page F6-8187</td>
</tr>
<tr>
<td>SHA1SU0</td>
<td>SHA1 schedule update 0</td>
<td>SHA1SU0 on page F6-8189</td>
</tr>
<tr>
<td>SHA1SU1</td>
<td>SHA1 schedule update 1</td>
<td>SHA1SU1 on page F6-8191</td>
</tr>
<tr>
<td>SHA256H</td>
<td>SHA256 hash update (part 1)</td>
<td>SHA256H on page F6-8193</td>
</tr>
</tbody>
</table>
### Table F2-30 AArch32 Cryptographic Extension instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA256H2</td>
<td>SHA256 hash update (part 2)</td>
<td>SHA256H2 on page F6-8195</td>
</tr>
<tr>
<td>SHA256SU0</td>
<td>SHA256 schedule update 0</td>
<td>SHA256SU0 on page F6-8197</td>
</tr>
<tr>
<td>SHA256SU1</td>
<td>SHA256 schedule update 1</td>
<td>SHA256SU1 on page F6-8199</td>
</tr>
</tbody>
</table>

a. The Cryptographic Extension adds the variant of the instruction that operates on two 64-bit polynomials.

See *The Armv8 Cryptographic Extension on page A2-80* for information about the permitted implementation options for the Cryptographic Extension.
F2.14 Floating-point data-processing instructions

Table F2-31 on page F2-7292 summarizes the data-processing instructions in the floating-point instruction set. In this table, floating-point register means a register in the SIMD and floating-point register file. The BFloat16 floating-point instructions are provided by FEAT_AA32BF16.

For details of the floating-point arithmetic used by floating-point instructions, see Floating-point support on page A1-55.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFloat16 convert from single-precision to BF16 format writing to bottom half of single-precision register</td>
<td>VCVTB (BFloat16) on page F6-8340</td>
</tr>
<tr>
<td>BFloat16 convert from single-precision to BF16 format writing to top half of single-precision register</td>
<td>VCVTT (BFloat16) on page F6-8361</td>
</tr>
<tr>
<td>Convert between double-precision and single-precision</td>
<td>VCVT (between double-precision and single-precision) on page F6-8312</td>
</tr>
<tr>
<td>Convert between floating-point and fixed-point</td>
<td>VCVT (between floating-point and fixed-point, floating-point) on page F6-8329</td>
</tr>
<tr>
<td>Convert between half-precision and single-precision, writing to bottom half of single-precision register</td>
<td>VCVTB on page F6-8337</td>
</tr>
<tr>
<td>Convert between half-precision and single-precision, writing to top half of single-precision register</td>
<td>VCVTT on page F6-8358</td>
</tr>
<tr>
<td>Convert from floating-point to integer</td>
<td>VCVT (floating-point to integer, floating-point) on page F6-8319</td>
</tr>
<tr>
<td>Convert from floating-point to integer using FPSCR rounding mode</td>
<td>VCVTR on page F6-8354</td>
</tr>
<tr>
<td>Convert from integer to floating-point</td>
<td>VCVT (integer to floating-point, floating-point) on page F6-8323</td>
</tr>
<tr>
<td>Floating-point Javascript convert to signed fixed-point, rounding toward zero</td>
<td>VJCVT on page F6-8419</td>
</tr>
<tr>
<td>Copy from one floating-point register to another</td>
<td>VMOV (register) on page F6-8546</td>
</tr>
<tr>
<td>Divide</td>
<td>VDIV on page F6-8363</td>
</tr>
<tr>
<td>Move immediate value to a floating-point register</td>
<td>VMOV (immediate) on page F6-8539</td>
</tr>
<tr>
<td>Square Root</td>
<td>VSQRT on page F6-8790</td>
</tr>
<tr>
<td>Vector Absolute value</td>
<td>VABS on page F6-8213</td>
</tr>
<tr>
<td>Vector Add</td>
<td>VADD (floating-point) on page F6-8227</td>
</tr>
<tr>
<td>Vector Compare with exceptions disabled</td>
<td>VCMPE on page F6-8304</td>
</tr>
<tr>
<td>Vector Compare with exceptions enabled</td>
<td>VCMP on page F6-8300</td>
</tr>
<tr>
<td>Vector Fused Multiply Accumulate</td>
<td>VFMA on page F6-8381</td>
</tr>
<tr>
<td>Vector Fused Multiply Subtract</td>
<td>VFMS on page F6-8395</td>
</tr>
<tr>
<td>Vector Fused Negate Multiply Accumulate</td>
<td>VFNMA on page F6-8405</td>
</tr>
<tr>
<td>Vector Fused Negate Multiply Subtract</td>
<td>VFNMS on page F6-8408</td>
</tr>
</tbody>
</table>
### Table F2-31 Floating-point data-processing instructions (continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply</td>
<td><em>VMUL (floating-point)</em> on page F6-8571</td>
</tr>
<tr>
<td>Vector Multiply Accumulate</td>
<td><em>VMLA (floating-point)</em> on page F6-8507</td>
</tr>
<tr>
<td>Vector Multiply Subtract</td>
<td><em>VMLS (floating-point)</em> on page F6-8520</td>
</tr>
<tr>
<td>Vector Negate Multiply</td>
<td><em>VNMUL</em> on page F6-8602</td>
</tr>
<tr>
<td>Vector Negate Multiply Accumulate</td>
<td><em>VNMLA</em> on page F6-8596</td>
</tr>
<tr>
<td>Vector Negate Multiply Subtract</td>
<td><em>VNMLS</em> on page F6-8599</td>
</tr>
<tr>
<td>Vector Negate, by inverting the sign bit</td>
<td><em>VNEG</em> on page F6-8592</td>
</tr>
<tr>
<td>Vector Subtract</td>
<td><em>VSUB (floating-point)</em> on page F6-8850</td>
</tr>
</tbody>
</table>
The AArch32 Instruction Sets Overview
F2.14 Floating-point data-processing instructions
Chapter F3
T32 Instruction Set Encoding

This chapter describes the encoding of the T32 instruction set. It contains the following sections:

- *T32 instruction set encoding* on page F3-7296.
- *About the T32 Advanced SIMD and floating-point instructions and their encoding* on page F3-7371.

In this chapter:

- In the decode tables, an entry of - for a field value means the value of the field does not affect the decoding.
- In the decode diagrams, a shaded field indicates that the bits in that field are not used in that level of decode.
F3.1 T32 instruction set encoding

The T32 instruction stream is a sequence of halfword-aligned halfwords. Each T32 instruction is either a single 16-bit halfword in that stream, or a 32-bit instruction consisting of two consecutive halfwords in that stream.

If the value of bits[15:11] of the halfword being decoded is one of the following, the halfword is the first halfword of a 32-bit instruction:
- 0b11101.
- 0b11110.
- 0b11111.

Otherwise, the halfword is a 16-bit instruction.

The T32 instruction encoding is:

<table>
<thead>
<tr>
<th>15 13 12</th>
<th>11 10</th>
<th>0 15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table F3-1 Main encoding table for the T32 instruction set

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>!= 111</td>
<td>16-bit on page F3-7296</td>
</tr>
<tr>
<td>111 00</td>
<td>B- T2 variant</td>
</tr>
<tr>
<td>111 1= 00</td>
<td>32-bit on page F3-7307</td>
</tr>
</tbody>
</table>

F3.1.1 16-bit

This section describes the encoding of the 16-bit group. The encodings in this section are decoded from T32 instruction set encoding.

<table>
<thead>
<tr>
<th>15 10 9</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
</tbody>
</table>

This decode also imposes the constraint:
- op0<5:3> != 111.

Table F3-2 Encoding table for the 16-bit group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>00xxxx</td>
<td>Shift (immediate), add, subtract, move, and compare on page F3-7300</td>
</tr>
<tr>
<td>010000</td>
<td>Data-processing (two low registers) on page F3-7297</td>
</tr>
<tr>
<td>010001</td>
<td>Special data instructions and branch and exchange on page F3-7302</td>
</tr>
<tr>
<td>0100lx</td>
<td>LDR (literal)- T1 variant</td>
</tr>
<tr>
<td>010xx</td>
<td>Load/store (register offset) on page F3-7298</td>
</tr>
</tbody>
</table>
Table F3-2 Encoding table for the 16-bit group (continued)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
</tbody>
</table>

| 011xxx        | Load/store word/byte (immediate offset) on page F3-7298 |
| 1000xx        | Load/store halfword (immediate offset) on page F3-7299 |
| 1001xx        | Load/store (SP-relative) on page F3-7299 |
| 1010xx        | Add PC/SP (immediate) on page F3-7299 |
| 1011xx        | Miscellaneous 16-bit instructions on page F3-7303 |
| 1100xx        | Load/store multiple on page F3-7300 |
| 1101xx        | Conditional branch, and Supervisor Call on page F3-7306 |

Data-processing (two low registers)

This section describes the encoding of the Data-processing (two low registers) instruction class. The encodings in this section are decoded from 16-bit on page F3-7296.

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 6 5 3 2 0</th>
<th>op</th>
<th>Rs</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
</tr>
</tbody>
</table>

| 0000          | AND, ANDS (register) |
| 0001          | EOR, EORS (register) |
| 0010          | MOV, MOVS (register-shifted register)- Logical shift left variant |
| 0011          | MOV, MOVS (register-shifted register)- Logical shift right variant |
| 0100          | MOV, MOVS (register-shifted register)- Arithmetic shift right variant |
| 0101          | ADC, ADCS (register) |
| 0110          | SBC, SBCS (register) |
| 0111          | MOV, MOVS (register-shifted register)- Rotate right variant |
| 1000          | TST (register) |
| 1001          | RSB, RSBS (immediate) |
| 1010          | CMP (register) |
| 1011          | CMN (register) |
| 1100          | ORR, ORRS (register) |
Load/store (register offset)

This section describes the encoding of the Load/store (register offset) instruction class. The encodings in this section are decoded from 16-bit on page F3-7296.

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>6 5</th>
<th>3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1</td>
<td>L</td>
<td>B</td>
<td>H</td>
</tr>
</tbody>
</table>

Load/store word/byte (immediate offset)

This section describes the encoding of the Load/store word/byte (immediate offset) instruction class. The encodings in this section are decoded from 16-bit on page F3-7296.

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10</th>
<th>6 5</th>
<th>3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1</td>
<td>B</td>
<td>L</td>
<td>imm5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>B</td>
</tr>
<tr>
<td>0 0 0</td>
<td>STR (register)</td>
</tr>
<tr>
<td>0 0 1</td>
<td>STRH (register)</td>
</tr>
<tr>
<td>0 1 0</td>
<td>STRB (register)</td>
</tr>
<tr>
<td>0 1 1</td>
<td>LDRSB (register)</td>
</tr>
<tr>
<td>1 0 0</td>
<td>LDR (register)</td>
</tr>
<tr>
<td>1 0 1</td>
<td>LDRH (register)</td>
</tr>
<tr>
<td>1 1 0</td>
<td>LDRB (register)</td>
</tr>
<tr>
<td>1 1 1</td>
<td>LDRSH (register)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>L</td>
</tr>
<tr>
<td>0 0</td>
<td>STR (immediate)</td>
</tr>
</tbody>
</table>
Load/store halfword (immediate offset)

This section describes the encoding of the Load/store halfword (immediate offset) instruction class. The encodings in this section are decoded from 16-bit on page F3-7296.

```
| 15 14 13 12| 11 10  6  5  3  2  0 |
| 1 0 0 0 | L | imm5 | Rn | Rt |
```

Load/store (SP-relative)

This section describes the encoding of the Load/store (SP-relative) instruction class. The encodings in this section are decoded from 16-bit on page F3-7296.

```
| 15 14 13 12| 11 10  8  7  0 |
| 1 0 0 1 | L | Rt | imm8 |
```

Add PC/SP (immediate)

This section describes the encoding of the Add PC/SP (immediate) instruction class. The encodings in this section are decoded from 16-bit on page F3-7296.
Load/store multiple

This section describes the encoding of the Load/store multiple instruction class. The encodings in this section are decoded from 16-bit on page F3-7296.

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10  8  7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0</td>
<td>SP</td>
<td>Rd</td>
</tr>
<tr>
<td></td>
<td>imm8</td>
<td></td>
</tr>
</tbody>
</table>
```

### Decode fields

<table>
<thead>
<tr>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

F3.1.2 Shift (immediate), add, subtract, move, and compare

This section describes the encoding of the Shift (immediate), add, subtract, move, and compare group. The encodings in this section are decoded from 16-bit on page F3-7296.

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10  9</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0</td>
<td>L</td>
<td>Rn</td>
</tr>
<tr>
<td></td>
<td>register_list</td>
<td></td>
</tr>
</tbody>
</table>
```

### Decode fields

<table>
<thead>
<tr>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

### Table F3-3 Encoding table for the Shift (immediate), add, subtract, move, and compare group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2</td>
<td></td>
</tr>
<tr>
<td>0 11 0</td>
<td>Add, subtract (three low registers)</td>
</tr>
<tr>
<td>0 11 1</td>
<td>Add, subtract (two low registers and immediate) on page F3-7301</td>
</tr>
<tr>
<td>0 != 11 -</td>
<td>MOV, MOVS (register)- T2 variant</td>
</tr>
<tr>
<td>1 - -</td>
<td>Add, subtract, compare, move (one low register and immediate) on page F3-7301</td>
</tr>
</tbody>
</table>
Add, subtract (three low registers)

This section describes the encoding of the Add, subtract (three low registers) instruction class. The encodings in this section are decoded from Shift (immediate), add, subtract, move, and compare on page F3-7300.

```
| 15 14 13 12|11 10 9 8 | 6 5 | 3 2 0 |
| 0 0 0 1 1 0 | S | Rm | Rn | Rd |
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>ADD, ADDS (register)</td>
</tr>
<tr>
<td>1</td>
<td>SUB, SUBS (register)</td>
</tr>
</tbody>
</table>

Add, subtract (two low registers and immediate)

This section describes the encoding of the Add, subtract (two low registers and immediate) instruction class. The encodings in this section are decoded from Shift (immediate), add, subtract, move, and compare on page F3-7300.

```
| 15 14 13 12|11 10 9 8 | 6 5 | 3 2 0 |
| 0 0 0 1 1 1 | S | imm3 | Rn | Rd |
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>ADD, ADDS (immediate)</td>
</tr>
<tr>
<td>1</td>
<td>SUB, SUBS (immediate)</td>
</tr>
</tbody>
</table>

Add, subtract, compare, move (one low register and immediate)

This section describes the encoding of the Add, subtract, compare, move (one low register and immediate) instruction class. The encodings in this section are decoded from Shift (immediate), add, subtract, move, and compare on page F3-7300.

```
| 15 14 13 12|11 10 8 7 | 0 |
| 0 0 1 | op | Rd | imm8 |
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>MOV, MOVS (immediate)</td>
</tr>
</tbody>
</table>
F3.1.3 Special data instructions and branch and exchange

This section describes the encoding of the Special data instructions and branch and exchange group. The encodings in this section are decoded from 16-bit on page F3-7296.

<table>
<thead>
<tr>
<th>15</th>
<th>10 9 8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>op0</td>
<td></td>
</tr>
</tbody>
</table>

Table F3-4 Encoding table for the Special data instructions and branch and exchange group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Branch and exchange on page F3-7302</td>
</tr>
<tr>
<td>!= 11</td>
<td>Add, subtract, compare, move (two high registers) on page F3-7302</td>
</tr>
</tbody>
</table>

Branch and exchange

This section describes the encoding of the Branch and exchange instruction class. The encodings in this section are decoded from Special data instructions and branch and exchange.

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 1</td>
<td>L Rm</td>
<td>(0) (0) (0)</td>
</tr>
</tbody>
</table>

Add, subtract, compare, move (two high registers)

This section describes the encoding of the Add, subtract, compare, move (two high registers) instruction class. The encodings in this section are decoded from Special data instructions and branch and exchange.
F3.1.4 Miscellaneous 16-bit instructions

This section describes the encoding of the Miscellaneous 16-bit instructions group. The encodings in this section are decoded from 16-bit on page F3-7296.

Table F3-5 Encoding table for the Miscellaneous 16-bit instructions group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2 op3</td>
<td>Adjust SP (immediate) on page F3-7304</td>
<td>-</td>
</tr>
<tr>
<td>0000 - - -</td>
<td>Extend on page F3-7304</td>
<td>-</td>
</tr>
<tr>
<td>0110 00 0 -</td>
<td>SETPAN</td>
<td>FEAT_PAN</td>
</tr>
<tr>
<td>0110 00 1 -</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0110 01 - -</td>
<td>Change Processor State on page F3-7304</td>
<td>-</td>
</tr>
<tr>
<td>0110 1x - -</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0111 - - -</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1000 - - -</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1010 10 - -</td>
<td>HLT</td>
<td>-</td>
</tr>
<tr>
<td>1010 != 10 - -</td>
<td>Reverse bytes on page F3-7305</td>
<td>-</td>
</tr>
<tr>
<td>1110 - - -</td>
<td>BKPT</td>
<td>-</td>
</tr>
<tr>
<td>1111 - - 0000</td>
<td>Hints on page F3-7305</td>
<td>-</td>
</tr>
</tbody>
</table>
Adjust SP (immediate)

This section describes the encoding of the Adjust SP (immediate) instruction class. The encodings in this section are decoded from Miscellaneous 16-bit instructions on page F3-7303.

<table>
<thead>
<tr>
<th>[15 14 13 12]</th>
<th>[11 10 9 8]</th>
<th>[7 6]</th>
<th>[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 0 0</td>
<td>S</td>
<td>imm7</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>ADD, ADDS (SP plus immediate)</td>
</tr>
<tr>
<td></td>
<td>SUB, SUBS (SP minus immediate)</td>
</tr>
</tbody>
</table>

Extend

This section describes the encoding of the Extend instruction class. The encodings in this section are decoded from Miscellaneous 16-bit instructions on page F3-7303.

<table>
<thead>
<tr>
<th>[15 14 13 12]</th>
<th>[11 10 9 8]</th>
<th>[7 5]</th>
<th>[3 2 0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 1 0 0</td>
<td>U</td>
<td>B</td>
<td>Rm</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>SXTH</td>
</tr>
<tr>
<td>B</td>
<td>SXTB</td>
</tr>
<tr>
<td></td>
<td>UXTH</td>
</tr>
<tr>
<td></td>
<td>UXTB</td>
</tr>
</tbody>
</table>

Change Processor State

This section describes the encoding of the Change Processor State instruction class. The encodings in this section are decoded from Miscellaneous 16-bit instructions on page F3-7303.
Reverse bytes

This section describes the encoding of the Reverse bytes instruction class. The encodings in this section are decoded from Miscellaneous 16-bit instructions on page F3-7303.

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 0 1 0 0 1</td>
<td>op</td>
<td>flags</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op flags</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SETEND</td>
</tr>
<tr>
<td>1 0xxxx</td>
<td>CPS, CPSID, CPSIE- Interrupt enable variant</td>
</tr>
<tr>
<td>1 1xxxx</td>
<td>CPS, CPSID, CPSIE- Interrupt disable variant</td>
</tr>
</tbody>
</table>

Hints

This section describes the encoding of the Hints instruction class. The encodings in this section are decoded from Miscellaneous 16-bit instructions on page F3-7303.

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5</th>
<th>3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 0 1 0 0 1</td>
<td>op</td>
<td>Rm</td>
<td>Rd</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>REV</td>
</tr>
<tr>
<td>01</td>
<td>REV16</td>
</tr>
<tr>
<td>11</td>
<td>REVSH</td>
</tr>
</tbody>
</table>

| 15 14 13 12|11 10 9 8 | 7 4 3 2 1 0 |
|-------------|---------|-------|-------|
| 1 0 1 1 1 1 1 1 | hint | 0 0 0 0 |

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>hint</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>NOP</td>
</tr>
<tr>
<td>0001</td>
<td>YIELD</td>
</tr>
<tr>
<td>0010</td>
<td>WFE</td>
</tr>
<tr>
<td>0011</td>
<td>WFI</td>
</tr>
</tbody>
</table>
Push and Pop

This section describes the encoding of the Push and Pop instruction class. The encodings in this section are decoded from Miscellaneous 16-bit instructions on page F3-7303.

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1</td>
<td>L</td>
<td>1 0</td>
<td>P</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>hint</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>SEV</td>
</tr>
<tr>
<td>0101</td>
<td>SEVL</td>
</tr>
<tr>
<td>011x</td>
<td>Reserved hint, behaves as NOP.</td>
</tr>
<tr>
<td>1xxx</td>
<td>Reserved hint, behaves as NOP.</td>
</tr>
</tbody>
</table>

F3.1.5 Conditional branch, and Supervisor Call

This section describes the encoding of the Conditional branch, and Supervisor Call group. The encodings in this section are decoded from 16-bit on page F3-7296.

<table>
<thead>
<tr>
<th>15 12</th>
<th>11 8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1101</td>
<td>op0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>PUSH</td>
</tr>
<tr>
<td>1</td>
<td>POP</td>
</tr>
</tbody>
</table>

Table F3-6 Encoding table for the Conditional branch, and Supervisor Call group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>011x</td>
<td>Exception generation on page F3-7306</td>
</tr>
<tr>
<td>!= 011x</td>
<td>B- T1 variant</td>
</tr>
</tbody>
</table>

Exception generation

This section describes the encoding of the Exception generation instruction class. The encodings in this section are decoded from Conditional branch, and Supervisor Call.
This section describes the encoding of the 32-bit group. The encodings in this section are decoded from T32 instruction set encoding on page F3-7296.

This decode also imposes the constraint:

- $\text{op0<3:2>} \neq 00$.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>UDF</td>
</tr>
<tr>
<td>1</td>
<td>SVC</td>
</tr>
</tbody>
</table>

### Table F3-7 Encoding table for the 32-bit group

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>xx0xx</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>xx1xx</td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10xx</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>10x0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>10x1</td>
<td>xxx0</td>
<td>0</td>
</tr>
<tr>
<td>10x1</td>
<td>xxxx1</td>
<td>0</td>
</tr>
<tr>
<td>1100</td>
<td>1xxx0</td>
<td>-</td>
</tr>
<tr>
<td>1100</td>
<td>!= 1xxx0</td>
<td>-</td>
</tr>
<tr>
<td>1101</td>
<td>0xxx</td>
<td>-</td>
</tr>
<tr>
<td>1101</td>
<td>10xxx</td>
<td>-</td>
</tr>
<tr>
<td>1101</td>
<td>11xxx</td>
<td>-</td>
</tr>
</tbody>
</table>
Load/store multiple

This section describes the encoding of the Load/store multiple instruction class. The encodings in this section are decoded from 32-bit on page F3-7307.

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 3 | 0 | 15 14 13 | | | | 0 |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 1 1 1 0 1 0 0 | opc | W | L | Rn | P | M | register_list |

Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

Data-processing (shifted register)

This section describes the encoding of the Data-processing (shifted register) instruction class. The encodings in this section are decoded from 32-bit on page F3-7307.

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>5 4 3</th>
<th>0</th>
<th>15 14 12</th>
<th>11 8 7 6 5 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 0 1</td>
<td>op1</td>
<td>S</td>
<td>Rn</td>
<td>[0]</td>
<td>imm3</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Decode fields

<table>
<thead>
<tr>
<th>op1</th>
<th>S</th>
<th>Rn</th>
<th>imm3:imm2:stype</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0 - 0000011</td>
<td>-</td>
<td>AND, ANDS (register)- AND, shift or rotate by value variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0 - 0000011</td>
<td>-</td>
<td>AND, ANDS (register)- AND, rotate right with extend variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 1 - 0000011</td>
<td>1111</td>
<td>AND, ANDS (register)- ANDS, shift or rotate by value variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 1 - 0000011</td>
<td>1111</td>
<td>TST (register)- Shift or rotate by value variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 1 - 0000011</td>
<td>1111</td>
<td>AND, ANDS (register)- ANDS, rotate right with extend variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 1 - 0000011</td>
<td>1111</td>
<td>TST (register)- Rotate right with extend variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001 0 0000011</td>
<td>-</td>
<td>BIC, BICS (register)- BICS, shift or rotate by value variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001 0 0000011</td>
<td>-</td>
<td>BIC, BICS (register)- BICS, rotate right with extend variant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 0 0000011</td>
<td>-</td>
<td>ORR, ORRS (register)- ORR, shift or rotate by value variant</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Decode fields

<table>
<thead>
<tr>
<th>op1</th>
<th>S</th>
<th>Rn</th>
<th>imm3:imm2:stype</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>0</td>
<td>!= 1111</td>
<td>0000011</td>
<td>-</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>1111</td>
<td>!= 0000011</td>
<td>-</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>1111</td>
<td>0000011</td>
<td>-</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td>!= 1111</td>
<td>!= 0000011</td>
<td>-</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td>1111</td>
<td>!= 0000011</td>
<td>-</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td>1111</td>
<td>0000011</td>
<td>-</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>!= 1111</td>
<td>!= 0000011</td>
<td>-</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>!= 1111</td>
<td>0000011</td>
<td>-</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>1111</td>
<td>!= 0000011</td>
<td>-</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
<td>!= 1111</td>
<td>!= 0000011</td>
<td>-</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
<td>!= 1111</td>
<td>= 0000011</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
<td>!= 1111</td>
<td>= 0000011</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
<td>0000011</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td>!= 1111</td>
<td>= 0000011</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td>1111</td>
<td>!= 0000011</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td>1111</td>
<td>0000011</td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>-</td>
<td>-</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>-</td>
<td>xxxxx00</td>
<td>-</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>-</td>
<td>xxxxx01</td>
<td>-</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>-</td>
<td>xxxxx10</td>
<td>-</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>-</td>
<td>xxxxx11</td>
<td>-</td>
</tr>
<tr>
<td>0111</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>!= 1101</td>
<td>!= 0000011</td>
<td>-</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>!= 1101</td>
<td>0000011</td>
<td>-</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>1101</td>
<td>!= 0000011</td>
<td>-</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>1101</td>
<td>0000011</td>
<td>-</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>-</td>
<td>!= 0000011</td>
<td>1111</td>
</tr>
</tbody>
</table>

### Instruction page

- **ORR, ORRS (register)- ORR, rotate right with extend variant**
- **MOV, MOVS (register)- MOV, shift or rotate by value variant**
- **MOV, MOVS (register)- MOV, rotate right with extend variant**
- **ORR, ORRS (register)- ORRS, shift or rotate by value variant**
- **ORR, ORRS (register)- ORRS, rotate right with extend variant**
- **MOV, MOVS (register)- MOVS, shift or rotate by value variant**
- **MOV, MOVS (register)- MOVS, rotate right with extend variant**
- **ORN, ORNS (register)- ORN, shift or rotate by value variant**
- **ORN, ORNS (register)- ORN, rotate right with extend variant**
- **MVN, MVNS (register)- MVN, shift or rotate by value variant**
- **MVN, MVNS (register)- MVN, rotate right with extend variant**
- **ORN, ORNS (register)- ORNS, shift or rotate by value variant**
- **ORN, ORNS (register)- ORNS, rotate right with extend variant**
- **MVN, MVNS (register)- MVNS, shift or rotate by value variant**
- **MVN, MVNS (register)- MVNS, rotate right with extend variant**
- **EOR, EORS (register)- EOR, shift or rotate by value variant**
- **EOR, EORS (register)- EOR, rotate right with extend variant**
- **EOR, EORS (register)- EORS, shift or rotate by value variant**
- **EOR, EORS (register)- EORS, rotate right with extend variant**
- **TEQ (register)- Shift or rotate by value variant**
- **TEQ (register)- Rotate right with extend variant**
- **PKHBT, PKHTB- PKHBT variant**
- **PKHBT, PKHTB- PKHTB variant**
- **Unallocated.**
- **Unallocated.**
- **Unallocated.**
- **ADD, ADDS (register)- ADD, shift or rotate by value variant**
- **ADD, ADDS (register)- ADD, rotate right with extend variant**
- **ADD, ADDS (SP plus register)- ADD, shift or rotate by value variant**
- **ADD, ADDS (SP plus register)- ADD, rotate right with extend variant**
- **CMN (register)- Shift or rotate by value variant**
## Data-processing (modified immediate)

This section describes the encoding of the Data-processing (modified immediate) instruction class. The encodings in this section are decoded from 32-bit on page F3-7307.

<table>
<thead>
<tr>
<th>op1</th>
<th>S</th>
<th>Rn</th>
<th>imm3:imm2:stype</th>
<th>Rd</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>!= 1101</td>
<td>!= 000011</td>
<td>!= 1111</td>
<td>ADD, ADDS (register)- ADDS, shift or rotate by value variant</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>!= 1101</td>
<td>000011</td>
<td>!= 1111</td>
<td>ADD, ADDS (register)- ADDS, rotate right with extend variant</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>-</td>
<td>000011</td>
<td>1111</td>
<td>CMN (register)- Rotate right with extend variant</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>1101</td>
<td>!= 000011</td>
<td>!= 1111</td>
<td>ADD, ADDS (SP plus register)- ADDS, shift or rotate by value variant</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>1101</td>
<td>000011</td>
<td>!= 1111</td>
<td>ADD, ADDS (SP plus register)- ADDS, rotate right with extend variant</td>
</tr>
<tr>
<td>1001</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1010</td>
<td>-</td>
<td>-</td>
<td>!= 000011</td>
<td>-</td>
<td>ADC, ADCS (register)- ADCS, shift or rotate by value variant</td>
</tr>
<tr>
<td>1010</td>
<td>-</td>
<td>-</td>
<td>000011</td>
<td>-</td>
<td>ADC, ADCS (register)- ADCS, rotate right with extend variant</td>
</tr>
<tr>
<td>1011</td>
<td>-</td>
<td>-</td>
<td>!= 000011</td>
<td>-</td>
<td>SBC, SBCS (register)- SBCS, shift or rotate by value variant</td>
</tr>
<tr>
<td>1011</td>
<td>-</td>
<td>-</td>
<td>000011</td>
<td>-</td>
<td>SBC, SBCS (register)- SBCS, rotate right with extend variant</td>
</tr>
<tr>
<td>1100</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td>!= 1101</td>
<td>!= 000011</td>
<td>-</td>
<td>SUB, SUBS (register)- SUB, shift or rotate by value variant</td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td>!= 1101</td>
<td>000011</td>
<td>-</td>
<td>SUB, SUBS (register)- SUB, rotate right with extend variant</td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td>1101</td>
<td>!= 000011</td>
<td>-</td>
<td>SUB, SUBS (SP minus register)- SUB, shift or rotate by value variant</td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td>1101</td>
<td>000011</td>
<td>-</td>
<td>SUB, SUBS (SP minus register)- SUB, rotate right with extend variant</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>-</td>
<td>!= 000011</td>
<td>1111</td>
<td>CMP (register)- Shift or rotate by value variant</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>!= 1101</td>
<td>!= 000011</td>
<td>!= 1111</td>
<td>SUB, SUBS (register)- SUBS, shift or rotate by value variant</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>!= 1101</td>
<td>000011</td>
<td>!= 1111</td>
<td>SUB, SUBS (register)- SUBS, rotate right with extend variant</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>-</td>
<td>000011</td>
<td>1111</td>
<td>CMP (register)- Rotate right with extend variant</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1101</td>
<td>!= 000011</td>
<td>!= 1111</td>
<td>SUB, SUBS (SP minus register)- SUBS, shift or rotate by value variant</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1101</td>
<td>000011</td>
<td>!= 1111</td>
<td>SUB, SUBS (SP minus register)- SUBS, rotate right with extend variant</td>
</tr>
<tr>
<td>1110</td>
<td>-</td>
<td>-</td>
<td>!= 000011</td>
<td>-</td>
<td>RSB, RSBS (register)- RSBS, shift or rotate by value variant</td>
</tr>
<tr>
<td>1110</td>
<td>-</td>
<td>-</td>
<td>000011</td>
<td>-</td>
<td>RSB, RSBS (register)- RSBS, rotate right with extend variant</td>
</tr>
<tr>
<td>1111</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
## Decode fields

<table>
<thead>
<tr>
<th>op1</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>AND, ANDS (immediate)- AND variant</td>
</tr>
<tr>
<td>0000</td>
<td>1</td>
<td>-</td>
<td>!= 1111</td>
<td>AND, ANDS (immediate)- ANDS variant</td>
</tr>
<tr>
<td>0000</td>
<td>1</td>
<td>-</td>
<td>1111</td>
<td>TST (immediate)</td>
</tr>
<tr>
<td>0011</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BIC, BICS (immediate)</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>!= 1111</td>
<td>-</td>
<td>ORR, ORRS (immediate)- ORR variant</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>1111</td>
<td>-</td>
<td>MOV, MOVS (immediate)- MOV variant</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td>!= 1111</td>
<td>-</td>
<td>ORR, ORRS (immediate)- ORRS variant</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td>1111</td>
<td>-</td>
<td>MOV, MOVS (immediate)- MOVS variant</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>!= 1111</td>
<td>-</td>
<td>ORN, ORNS (immediate)- Not flag setting variant</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>1111</td>
<td>-</td>
<td>MVN, MVNS (immediate)- MVN variant</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
<td>!= 1111</td>
<td>-</td>
<td>ORN, ORNS (immediate)- Flag setting variant</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
<td>1111</td>
<td>-</td>
<td>MVN, MVNS (immediate)- MVNS variant</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>EOR, EORS (immediate)- EOR variant</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td>-</td>
<td>!= 1111</td>
<td>EOR, EORS (immediate)- EORS variant</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td>1111</td>
<td>TEQ (immediate)</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>!= 1101</td>
<td>ADD, ADDS (immediate)- ADD variant</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>1101</td>
<td>ADD, ADDS (SP plus immediate)- ADD variant</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>!= 1101</td>
<td>!= 1111</td>
<td>ADD, ADDS (immediate)- ADDS variant</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>1101</td>
<td>!= 1111</td>
<td>ADD, ADDS (SP plus immediate)- ADDS variant</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>-</td>
<td>1111</td>
<td>CMN (immediate)</td>
</tr>
<tr>
<td>1001</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1010</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>ADC, ADCS (immediate)</td>
</tr>
<tr>
<td>1011</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SBC, SBCS (immediate)</td>
</tr>
<tr>
<td>1100</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td>!= 1101</td>
<td>SUB, SUBS (immediate)- SUB variant</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td>1101</td>
<td>SUB, SUBS (SP minus immediate)- SUB variant</td>
<td></td>
</tr>
</tbody>
</table>
## Long multiply and divide

This section describes the encoding of the Long multiply and divide instruction class. The encodings in this section are decoded from 32-bit on page F3-7307.

| 15 14 13 12 11 10 9 8 7 6 4 3 0 | 1 1 1 1 0 1 1 1 | op1 | Rn | RdLo | RdHi | op2 | Rm |

### Decode fields

<table>
<thead>
<tr>
<th>op1</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1101</td>
<td>1</td>
<td>!= 1101</td>
<td>!= 1111</td>
<td>SUB, SUBS (immediate)- SUBS variant</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1101</td>
<td>!= 1111</td>
<td>SUB, SUBS (SP minus immediate)- SUBS variant</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>-</td>
<td>1111</td>
<td>CMP (immediate)</td>
</tr>
<tr>
<td>1110</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>RSB, RSBS (immediate)</td>
</tr>
<tr>
<td>1111</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>!= 0000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>000</td>
<td>0000</td>
<td>SMULL, SMULLS</td>
</tr>
<tr>
<td>001</td>
<td>!= 1111</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>001</td>
<td>1111</td>
<td>SDIV</td>
</tr>
<tr>
<td>010</td>
<td>!= 0000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>010</td>
<td>0000</td>
<td>UMULL, UMULLS</td>
</tr>
<tr>
<td>011</td>
<td>!= 1111</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011</td>
<td>1111</td>
<td>UDIV</td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>SMLAL, SMLALS</td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100</td>
<td>001x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100</td>
<td>01xx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>SMLALBB, SMLALBT, SMLALTB, SMLALTT- SMLALBB variant</td>
</tr>
<tr>
<td>100</td>
<td>1001</td>
<td>SMLALBB, SMLALBT, SMLALTB, SMLALTT- SMLALBT variant</td>
</tr>
<tr>
<td>100</td>
<td>1010</td>
<td>SMLALBB, SMLALBT, SMLALTB, SMLALTT- SMLALTB variant</td>
</tr>
<tr>
<td>100</td>
<td>1011</td>
<td>SMLALBB, SMLALBT, SMLALTB, SMLALTT- SMLALTT variant</td>
</tr>
<tr>
<td>100</td>
<td>1100</td>
<td>SMLALD, SMLALDX- SMLALD variant</td>
</tr>
<tr>
<td>100</td>
<td>1101</td>
<td>SMLALD, SMLALDX- SMLALDX variant</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>111x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>101</td>
<td>0xx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>101</td>
<td>10xx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>101</td>
<td>1100</td>
<td>SMLSLD, SMLSLDX - SMLSLD variant</td>
</tr>
<tr>
<td>101</td>
<td>1101</td>
<td>SMLSLD, SMLSLDX - SMLSLDX variant</td>
</tr>
<tr>
<td>101</td>
<td>111x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>110</td>
<td>0000</td>
<td>UMLAL, UMLALS</td>
</tr>
<tr>
<td>110</td>
<td>0001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>110</td>
<td>001x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>110</td>
<td>010x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>110</td>
<td>0110</td>
<td>UMAAL</td>
</tr>
<tr>
<td>110</td>
<td>0111</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>110</td>
<td>1xxx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>111</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

### F3.1.7 System register access, Advanced SIMD, and floating-point

This section describes the encoding of the System register access, Advanced SIMD, and floating-point group. The encodings in this section are decoded from 32-bit on page F3-7307.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x</td>
<td>0x</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>0x</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>11</td>
<td>-</td>
<td>-</td>
<td>Advanced SIMD data-processing on page F3-7314</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>1x</td>
<td>-</td>
<td>Advanced SIMD and System register load/store and 64-bit move on page F3-7323</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1x</td>
<td>1</td>
<td>Advanced SIMD and System register 32-bit move on page F3-7327</td>
</tr>
</tbody>
</table>
F3.1 T32 instruction set encoding

F3.1.8 Advanced SIMD data-processing

This section describes the encoding of the Advanced SIMD data-processing group. The encodings in this section are decoded from System register access, Advanced SIMD, and floating-point on page F3-7313.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the T32 Advanced SIMD and floating-point instructions and their encoding on page F3-7371

```
<table>
<thead>
<tr>
<th>15</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>7</th>
<th>6</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>1111</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

op0 op1

Table F3-9 Encoding table for the Advanced SIMD data-processing group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>Advanced SIMD three registers of the same length</td>
</tr>
<tr>
<td>1 0</td>
<td>Advanced SIMD two registers, or three registers of different lengths on page F3-7317</td>
</tr>
<tr>
<td>1 1</td>
<td>Advanced SIMD shifts and immediate generation on page F3-7322</td>
</tr>
</tbody>
</table>

Advanced SIMD three registers of the same length

This section describes the encoding of the Advanced SIMD three registers of the same length instruction class. The encodings in this section are decoded from Advanced SIMD data-processing.

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>size</td>
<td>Vn</td>
<td>Vd</td>
<td>opc</td>
<td>N</td>
<td>Q</td>
<td>M</td>
<td>o1</td>
</tr>
</tbody>
</table>
```

Table F3-8 Encoding table for the System register access, Advanced SIMD, and floating-point group (continued)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2 op3</td>
<td></td>
</tr>
<tr>
<td>0 10 10 0</td>
<td>Floating-point data-processing on page F3-7329</td>
</tr>
<tr>
<td>0 10 11 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1 != 11 1x -</td>
<td>Additional Advanced SIMD and floating-point instructions on page F3-7334</td>
</tr>
</tbody>
</table>
## T32 Instruction Set Encoding

### F3.1 T32 instruction set encoding

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>size</td>
<td>opc</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>1110</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>1111</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>1111</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0000</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>0001</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0000</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0001</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>1100</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0010</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0001</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0010</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0011</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0011</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>1100</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>1100</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>1101</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>1101</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>1110</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>1111</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>1111</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0100</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>1000</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0001</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>1000</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0100</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>1001</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0101</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0101</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>1011</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1100</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>1011</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0110</td>
</tr>
<tr>
<td>U</td>
<td>size</td>
<td>opc</td>
</tr>
<tr>
<td>----</td>
<td>------</td>
<td>-----</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>0001</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0110</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0111</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0111</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>1100</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>1101</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>1101</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>1110</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>1110</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>1111</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>1111</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0001</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>1001</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1100</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>1010</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>0001</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>1010</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>1010</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1100</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>1101</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>1110</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>1110</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>1111</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>1111</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0001</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1001</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1011</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1100</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1011</td>
</tr>
</tbody>
</table>
F3.1 T32 instruction set encoding

F3.1.9 Advanced SIMD two registers, or three registers of different lengths

This section describes the encoding of the Advanced SIMD two registers, or three registers of different lengths group. The encodings in this section are decoded from Advanced SIMD data-processing on page F3-7314.

Table F3-10 Encoding table for the Advanced SIMD two registers, or three registers of different lengths group

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opc</th>
<th>Q</th>
<th>o1</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>0001</td>
<td>-</td>
<td>1</td>
<td>VBIF</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1100</td>
<td>-</td>
<td>1</td>
<td>VQRDMLSH FEAT_RDM</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1111</td>
<td>1</td>
<td>0</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

Decide group or instruction page

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>-</td>
<td>-</td>
<td>VEXT (byte elements)</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0x</td>
<td>-</td>
<td>Advanced SIMD two registers misc</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>10</td>
<td>-</td>
<td>VTBL, VTBX</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>11</td>
<td>-</td>
<td>Advanced SIMD duplicate (scalar) on page F3-7319</td>
</tr>
<tr>
<td>-</td>
<td>!= 11</td>
<td>-</td>
<td>0</td>
<td>Advanced SIMD three registers of different lengths on page F3-7320</td>
</tr>
<tr>
<td>-</td>
<td>!= 11</td>
<td>-</td>
<td>1</td>
<td>Advanced SIMD two registers and a scalar on page F3-7321</td>
</tr>
</tbody>
</table>

Advanced SIMD two registers misc

This section describes the encoding of the Advanced SIMD two registers misc instruction class. The encodings in this section are decoded from Advanced SIMD two registers, or three registers of different lengths.

Table F3-10 Encoding table for the Advanced SIMD two registers, or three registers of different lengths group

<table>
<thead>
<tr>
<th>size</th>
<th>opc1</th>
<th>opc2</th>
<th>Q</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0000</td>
<td>-</td>
<td>-</td>
<td>VREV64</td>
</tr>
<tr>
<td>-</td>
<td>0001</td>
<td>-</td>
<td>-</td>
<td>VREV32</td>
</tr>
<tr>
<td>-</td>
<td>0010</td>
<td>-</td>
<td>-</td>
<td>VREV16</td>
</tr>
</tbody>
</table>

ARM DDI 0487I.a
Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential
### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>opc1</th>
<th>opc2</th>
<th>Q</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>00</td>
<td>0011</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>00</td>
<td>010x</td>
<td>-</td>
<td>VPADDL</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>00</td>
<td>0110</td>
<td>0</td>
<td>AESE</td>
<td>FEAT_AES</td>
</tr>
<tr>
<td>-</td>
<td>00</td>
<td>0110</td>
<td>1</td>
<td>AESD</td>
<td>FEAT_AES</td>
</tr>
<tr>
<td>-</td>
<td>00</td>
<td>0111</td>
<td>0</td>
<td>AESMC</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>00</td>
<td>0111</td>
<td>1</td>
<td>AESIMC</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>00</td>
<td>1000</td>
<td>-</td>
<td>VCLS</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>0000</td>
<td>-</td>
<td>VSWP</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>00</td>
<td>1001</td>
<td>-</td>
<td>VCLZ</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>00</td>
<td>1010</td>
<td>-</td>
<td>VCNT</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>00</td>
<td>1011</td>
<td>-</td>
<td>VMVN (register)</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>1100</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>00</td>
<td>110x</td>
<td>-</td>
<td>VPADAL</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>00</td>
<td>1110</td>
<td>-</td>
<td>VQABS</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>00</td>
<td>1111</td>
<td>-</td>
<td>VQNEG</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>01</td>
<td>x000</td>
<td>-</td>
<td>VCGT (immediate #0)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>01</td>
<td>x001</td>
<td>-</td>
<td>VCGE (immediate #0)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>01</td>
<td>x010</td>
<td>-</td>
<td>VCEQ (immediate #0)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>01</td>
<td>x011</td>
<td>-</td>
<td>VCLE (immediate #0)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>01</td>
<td>x100</td>
<td>-</td>
<td>VCLT (immediate #0)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>01</td>
<td>x110</td>
<td>-</td>
<td>VABS</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>01</td>
<td>x111</td>
<td>-</td>
<td>VNEG</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>01</td>
<td>0101</td>
<td>1</td>
<td>SHA1H</td>
<td>FEAT_SHA1</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>1100</td>
<td>1</td>
<td>VCVT (from single-precision to BFloat16, Advanced SIMD)</td>
<td>FEAT_AA32BF16</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>0001</td>
<td>-</td>
<td>VTRN</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>0010</td>
<td>-</td>
<td>VUZP</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>0011</td>
<td>-</td>
<td>VZIP</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>0100</td>
<td>0</td>
<td>VMOVN</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>0100</td>
<td>1</td>
<td>VQMOVN, VQMOVUN- Unsigned result variant</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>0101</td>
<td>-</td>
<td>VQMOVN, VQMOVUN- Signed result variant</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>0110</td>
<td>0</td>
<td>VSHLL</td>
<td>-</td>
</tr>
</tbody>
</table>
## Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>opc1</th>
<th>opc2</th>
<th>Q</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>10</td>
<td>0111</td>
<td>0</td>
<td>SHA1SU1</td>
<td>FEAT_SHA1</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>0111</td>
<td>1</td>
<td>SHA256SU0</td>
<td>FEAT_SHA256</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>1000</td>
<td>-</td>
<td>VRINTN (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>1001</td>
<td>-</td>
<td>VRINTX (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>1010</td>
<td>-</td>
<td>VRINTA (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>1011</td>
<td>-</td>
<td>VRINTZ (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1100</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>1100</td>
<td>0</td>
<td>VCVT (between half-precision and single-precision, Advanced SIMD)- Single-precision to half-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>1101</td>
<td>-</td>
<td>VRINTM (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>1110</td>
<td>0</td>
<td>VCVT (between half-precision and single-precision, Advanced SIMD)- Half-precision to single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>1110</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
<td>1111</td>
<td>-</td>
<td>VRINTP (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>11</td>
<td>000x</td>
<td>-</td>
<td>VCVTA (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>11</td>
<td>001x</td>
<td>-</td>
<td>VCVTN (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>11</td>
<td>010x</td>
<td>-</td>
<td>VCVTP (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>11</td>
<td>011x</td>
<td>-</td>
<td>VCVTM (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>11</td>
<td>10x0</td>
<td>-</td>
<td>VRECPE</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>11</td>
<td>10x1</td>
<td>-</td>
<td>VRSQRTE</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>1100</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>11</td>
<td>11xx</td>
<td>-</td>
<td>VCVT (between floating-point and integer, Advanced SIMD)</td>
<td>-</td>
</tr>
</tbody>
</table>

### Advanced SIMD duplicate (scalar)

This section describes the encoding of the Advanced SIMD duplicate (scalar) instruction class. The encodings in this section are decoded from *Advanced SIMD two registers, or three registers of different lengths* on page F3-7317.
Advanced SIMD three registers of different lengths

This section describes the encoding of the Advanced SIMD three registers of different lengths instruction class. The encodings in this section are decoded from Advanced SIMD two registers, or three registers of different lengths on page F3-7317.

| 15 14 13 12|11 10 9 |8 7 6 5 4 |3 0 |15 12|11 10 9 |7 6 5 4 |3 0 |
|---|---|---|---|---|---|---|---|---|
|1 |1 |1 |1 |1 |1 |1 |D |1 |1 |

**opc**

- **000** VDUP (scalar)
- **001** Unallocated.
- **01x** Unallocated.
- **1xx** Unallocated.

**U**

- **0000** VADDL
- **0001** VADDDW
- **0010** VSUBL
- **0100** VADDHN
- **0011** VSUBW
- **0110** VSUBHN
- **0101** VQDMLAL
- **0111** VQDMLSL
- **1010** VQDMLSL
- **0111** VQDMLSL
- **1000** VQDMLSL
- **1010** VQDMLSL
- **1000** VRADDHN
- **0110** VRSUBHN
### Advanced SIMD two registers and a scalar

This section describes the encoding of the Advanced SIMD two registers and a scalar instruction class. The encodings in this section are decoded from *Advanced SIMD two registers, or three registers of different lengths* on page F3-7317.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>U  opc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-  11x0</td>
<td>VMULL (integer and polynomial)</td>
<td></td>
</tr>
<tr>
<td>1  1001</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>1  1011</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>1  1101</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>-  1111</td>
<td>Unallocated.</td>
<td></td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>Q  opc</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>-  000x</td>
<td>VMLA (by scalar)</td>
<td></td>
</tr>
<tr>
<td>0  0011</td>
<td>VQDMLAL</td>
<td></td>
</tr>
<tr>
<td>-  0010</td>
<td>VMLAL (by scalar)</td>
<td></td>
</tr>
<tr>
<td>0  0111</td>
<td>VQDMLSL</td>
<td></td>
</tr>
<tr>
<td>-  010x</td>
<td>VMLS (by scalar)</td>
<td></td>
</tr>
<tr>
<td>0  1011</td>
<td>VQDMULL</td>
<td></td>
</tr>
<tr>
<td>-  0110</td>
<td>VMLSL (by scalar)</td>
<td></td>
</tr>
<tr>
<td>-  100x</td>
<td>VMUL (by scalar)</td>
<td></td>
</tr>
<tr>
<td>1  0011</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>-  1010</td>
<td>VMULL (by scalar)</td>
<td></td>
</tr>
<tr>
<td>1  0111</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>-  1100</td>
<td>VQDMULH</td>
<td></td>
</tr>
<tr>
<td>-  1101</td>
<td>VQRDMULH</td>
<td></td>
</tr>
<tr>
<td>1  1011</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>-  1110</td>
<td>VQRDMLAH FEAT_RDM</td>
<td></td>
</tr>
<tr>
<td>-  1111</td>
<td>VQRDMLSH FEAT_RDM</td>
<td></td>
</tr>
</tbody>
</table>
F3.1.10 Advanced SIMD shifts and immediate generation

This section describes the encoding of the Advanced SIMD shifts and immediate generation group. The encodings in this section are decoded from Advanced SIMD data-processing on page F3-7314.

![Table F3-11 Encoding table for the Advanced SIMD shifts and immediate generation group](image)

### Advanced SIMD one register and modified immediate

This section describes the encoding of the Advanced SIMD one register and modified immediate instruction class. The encodings in this section are decoded from Advanced SIMD shifts and immediate generation.

![Table showing decoding fields and instruction page](image)
Advanced SIMD two registers and shift amount

This section describes the encoding of the Advanced SIMD two registers and shift amount instruction class. The encodings in this section are decoded from *Advanced SIMD shifts and immediate generation* on page F3-7322.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>imm3H</td>
<td>imm3L</td>
<td>Vd</td>
<td>opc</td>
<td>L</td>
<td>Q</td>
<td>M</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>imm3H:L</th>
<th>imm3L</th>
<th>opc</th>
<th>Q</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>!= 0000</td>
<td>-</td>
<td>000</td>
<td>-</td>
<td>VSHR</td>
</tr>
<tr>
<td>-</td>
<td>!= 0000</td>
<td>-</td>
<td>001</td>
<td>-</td>
<td>VSRA</td>
</tr>
<tr>
<td>-</td>
<td>!= 0000</td>
<td>000</td>
<td>1010</td>
<td>0</td>
<td>VMOVl</td>
</tr>
<tr>
<td>-</td>
<td>!= 0000</td>
<td>-</td>
<td>0110</td>
<td>-</td>
<td>VRSHR</td>
</tr>
<tr>
<td>-</td>
<td>!= 0000</td>
<td>-</td>
<td>0111</td>
<td>-</td>
<td>VRSRA</td>
</tr>
<tr>
<td>-</td>
<td>!= 0000</td>
<td>-</td>
<td>0111</td>
<td>-</td>
<td>VQSHL, VQSHLU (immediate)- 128-bit SIMD vector, signed result variant</td>
</tr>
<tr>
<td>-</td>
<td>!= 0000</td>
<td>-</td>
<td>1001</td>
<td>0</td>
<td>VQSHRN, VQSHRUN- Signed result variant</td>
</tr>
<tr>
<td>-</td>
<td>!= 0000</td>
<td>-</td>
<td>1001</td>
<td>1</td>
<td>VQSHRN, VQSHRUN- Signed result variant</td>
</tr>
<tr>
<td>-</td>
<td>!= 0000</td>
<td>-</td>
<td>11xx</td>
<td>-</td>
<td>VSHLL</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>-</td>
<td>0101</td>
<td>-</td>
<td>VSHL (immediate)</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>-</td>
<td>1000</td>
<td>0</td>
<td>VSHRN</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>-</td>
<td>1000</td>
<td>1</td>
<td>VRSHRN</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>-</td>
<td>0100</td>
<td>-</td>
<td>VSRI</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>-</td>
<td>0101</td>
<td>-</td>
<td>VSLI</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>-</td>
<td>0110</td>
<td>-</td>
<td>VQSHL, VQSHLU (immediate)- 128-bit SIMD vector, unsigned result variant</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>-</td>
<td>1000</td>
<td>0</td>
<td>VQSHRN, VQSHRUN- Unsigned result variant</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>-</td>
<td>1000</td>
<td>1</td>
<td>VQSHRN, VQSHRUN- Unsigned result variant</td>
</tr>
</tbody>
</table>

### F3.1.11 Advanced SIMD and System register load/store and 64-bit move

This section describes the encoding of the Advanced SIMD and System register load/store and 64-bit move group. The encodings in this section are decoded from *System register access, Advanced SIMD, and floating-point* on page F3-7313.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see *About the T32 Advanced SIMD and floating-point instructions and their encoding* on page F3-7371.
This section describes the encoding of the Advanced SIMD and floating-point 64-bit move instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move on page F3-7323.

### Advanced SIMD and floating-point 64-bit move

This section describes the encoding of the Advanced SIMD and floating-point 64-bit move instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move on page F3-7323.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D  op size opc2 o3</td>
<td>VMOV (between two general-purpose registers and two single-precision registers)- From general-purpose registers variant</td>
</tr>
<tr>
<td>1 0 10 00 1</td>
<td>VMOV (between two general-purpose registers and a doubleword floating-point register)- From general-purpose registers variant</td>
</tr>
<tr>
<td>1 1 10 00 1</td>
<td>VMOV (between two general-purpose registers and two single-precision registers)- To general-purpose registers variant</td>
</tr>
<tr>
<td>1 1 11 00 1</td>
<td>VMOV (between two general-purpose registers and a doubleword floating-point register)- To general-purpose registers variant</td>
</tr>
</tbody>
</table>
System register 64-bit move

This section describes the encoding of the System register 64-bit move instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move on page F3-7323.

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 0</th>
<th>1 1 1 0 1 1 0 0 D 0 L Rt2</th>
<th>Rt1</th>
<th>1 1 1 opc1 CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>cp15</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Decode fields

Instruction page

<table>
<thead>
<tr>
<th>D</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Advanced SIMD and floating-point load/store

This section describes the encoding of the Advanced SIMD and floating-point load/store instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move on page F3-7323.

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 0</th>
<th>1 1 1 0 1 1 0 1 P U D W L Rn Vd 1 0 size imm8</th>
</tr>
</thead>
</table>

Decode fields

Instruction page

Feature

<table>
<thead>
<tr>
<th>P</th>
<th>U</th>
<th>W</th>
<th>L</th>
<th>Rn</th>
<th>size</th>
<th>imm8</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>10</td>
<td>VSTM, VSTMDB, VSTMIA- Increment After variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>11</td>
<td>xxxxx</td>
<td>VSTM, VSTMDB, VSTMIA- Increment After variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>11</td>
<td>xxxxxx1</td>
<td>FSTMDBX, FSTMIAX- Increment After variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>10</td>
<td>VLD, VLDMD, VLDMA- Increment After variant</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>11</td>
<td>xxxxx</td>
<td>VLD, VLDMD, VLDMA- Increment After variant</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>11</td>
<td>xxxxxx1</td>
<td>FLDX* (FLDMDBX, FLDMIAX)- Increment After variant</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>VSTR- Half-precision scalar variant</td>
<td>FEAT_FP1 6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>VSTR- Single-precision scalar variant</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>VSTR- Double-precision scalar variant</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
This section describes the encoding of the System register Load/Store instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move on page F3-7323.

The decode fields are as follows:

- **P**: Program status register (PSR) update
- **U**: User mode
- **W**: Word size
- **L**: Length
- **Rn**: Register number
- **siz**: Size
- **imm8**: Immediate 8-bit value
- **Instruction page**: Description of the instruction
- **Feature**: Optional feature

### Decode fields

<table>
<thead>
<tr>
<th>P</th>
<th>U</th>
<th>W</th>
<th>L</th>
<th>Rn</th>
<th>siz</th>
<th>imm8</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>!111</td>
<td>01</td>
<td>VLDR (immediate)- Half-precision scalar variant</td>
<td>FEAT_FP1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>!111</td>
<td>10</td>
<td>VLDR (immediate)- Single-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>!111</td>
<td>11</td>
<td>VLDR (immediate)- Double-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0x</td>
<td></td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>10</td>
<td></td>
<td>VSTM, VSTMDB, VSTMIA- Decrement Before variant</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>11</td>
<td>xxxxxx0</td>
<td>VSTM, VSTMDB, VSTMIA- Decrement Before variant</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>11</td>
<td>xxxxxx1</td>
<td>FSTMDBX, FSTMIAX- Decrement Before variant</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>10</td>
<td></td>
<td>VLM, VLMDB, VLMIA- Decrement Before variant</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>11</td>
<td>xxxxxx0</td>
<td>VLM, VLMDB, VLMIA- Decrement Before variant</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>11</td>
<td>xxxxxx1</td>
<td>FLDM*X (FLDMDBX, FLDMIAX)- Decrement Before variant</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1111</td>
<td>01</td>
<td>VLDR (literal)- Half-precision scalar variant</td>
<td>FEAT_FP1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1111</td>
<td>10</td>
<td>VLDR (literal)- Single-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1111</td>
<td>11</td>
<td>VLDR (literal)- Double-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td>Unallocated.</td>
<td></td>
</tr>
</tbody>
</table>

### System register Load/Store

This section describes the encoding of the System register Load/Store instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move on page F3-7323.

The decode fields are as follows:

- **P**: Program status register (PSR) update
- **U**: User mode
- **D**: Data size
- **L**: Length
- **Rn**: Register number
- **CRd**: Condition code register (CCR) destination
- **cp15**: Condition code register (CCR) update
- **imm8**: Immediate 8-bit value
- **Instruction page**: Description of the instruction

### Decode fields

<table>
<thead>
<tr>
<th>P:U:D</th>
<th>L</th>
<th>Rn</th>
<th>CRd</th>
<th>cp15</th>
<th>imm8</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>!000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td>Unallocated.</td>
</tr>
<tr>
<td>!000</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>0101</td>
<td>0</td>
<td>LDC (literal)</td>
</tr>
<tr>
<td>!000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td>Unallocated.</td>
</tr>
<tr>
<td>!000</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0101</td>
<td>0</td>
<td>STC- Post-indexed variant</td>
</tr>
<tr>
<td>0x1</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0101</td>
<td>0</td>
<td>LDC (immediate)- Post-indexed variant</td>
</tr>
</tbody>
</table>
### F3.1.12 Advanced SIMD and System register 32-bit move

This section describes the encoding of the Advanced SIMD and System register 32-bit move group. The encodings in this section are decoded from *System register access, Advanced SIMD, and floating-point* on page F3-7313.

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>7</th>
<th>5</th>
<th>4</th>
<th>0</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>8</th>
<th>7</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11101110</td>
<td>op0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table F3-13 Encoding table for the Advanced SIMD and System register 32-bit move group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td>VMOV (between general-purpose register and half-precision)</td>
</tr>
<tr>
<td>000</td>
<td>010</td>
<td>VMOV (between general-purpose register and single-precision)</td>
</tr>
<tr>
<td>001</td>
<td>010</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01x</td>
<td>010</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10x</td>
<td>010</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>110</td>
<td>010</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>111</td>
<td>010</td>
<td>Floating-point move special register on page F3-7327</td>
</tr>
<tr>
<td>-</td>
<td>011</td>
<td>Advanced SIMD 8/16/32-bit element move/duplicate</td>
</tr>
<tr>
<td>-</td>
<td>10x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>11x</td>
<td>System register 32-bit move on page F3-7328</td>
</tr>
</tbody>
</table>

### Floating-point move special register

This section describes the encoding of the Floating-point move special register instruction class. The encodings in this section are decoded from *Advanced SIMD and System register 32-bit move.*
Advanced SIMD 8/16/32-bit element move/duplicate

This section describes the encoding of the Advanced SIMD 8/16/32-bit element move/duplicate instruction class. The encodings in this section are decoded from "Advanced SIMD and System register 32-bit move on page F3-7327."

System register 32-bit move

This section describes the encoding of the System register 32-bit move instruction class. The encodings in this section are decoded from "Advanced SIMD and System register 32-bit move on page F3-7327."
F3.1.13 Floating-point data-processing

This section describes the encoding of the Floating-point data-processing group. The encodings in this section are decoded from System register access, Advanced SIMD, and floating-point on page F3-7313.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the T32 Advanced SIMD and floating-point instructions and their encoding on page F3-7371.

Floating-point data-processing (two registers)

This section describes the encoding of the Floating-point data-processing (two registers) instruction class. The encodings in this section are decoded from Floating-point data-processing.

Table F3-14 Encoding table for the Floating-point data-processing group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td>1x11 1 Floating-point data-processing (two registers)</td>
</tr>
<tr>
<td></td>
<td>1x11 0 Floating-point move immediate on page F3-7332</td>
</tr>
<tr>
<td></td>
<td>!= 1x11 - Floating-point data-processing (three registers) on page F3-7332</td>
</tr>
</tbody>
</table>

Floating-point data-processing (two registers)

This section describes the encoding of the Floating-point data-processing (two registers) instruction class. The encodings in this section are decoded from Floating-point data-processing.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>o 1 opc 2 size 3</td>
<td>Un_allocated.</td>
<td>-</td>
</tr>
<tr>
<td>0 000 01 0 Un_allocated.</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 000 01 1 VABS- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0 000 10 0 VMOV (register)- Single-precision scalar variant</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 000 10 1 VABS- Single-precision scalar variant</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 000 11 0 VMOV (register)- Double-precision scalar variant</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 000 11 1 VABS- Double-precision scalar variant</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 001 01 0 VNEG- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0 001 01 1 VSQRT- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0 001 10 0 VNEG- Single-precision scalar variant</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>opc1</td>
<td>opc2</td>
<td>size</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>Decode fields</td>
<td>Instruction page</td>
<td>Feature</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------</td>
<td>---------</td>
</tr>
<tr>
<td>0 110 11 0</td>
<td>VRINTR- Double-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>0 110 11 1</td>
<td>VRINTZ (floating-point)- Double-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>0 111 01 0</td>
<td>VRINTX (floating-point)- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 111 01 1</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0 111 10 0</td>
<td>VRINTX (floating-point)- Single-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>0 111 10 1</td>
<td>VCVT (between double-precision and single-precision)- Single-precision to double-precision variant</td>
<td></td>
</tr>
<tr>
<td>0 111 11 0</td>
<td>VRINTX (floating-point)- Double-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>0 111 11 1</td>
<td>VCVT (between double-precision and single-precision)- Double-precision to single-precision variant</td>
<td></td>
</tr>
<tr>
<td>1 000 01 -</td>
<td>VCVT (integer to floating-point, floating-point)- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 000 10 -</td>
<td>VCVT (integer to floating-point, floating-point)- Single-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>1 000 11 -</td>
<td>VCVT (integer to floating-point, floating-point)- Double-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>1 001 01 -</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>1 001 10 -</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>1 001 11 0</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>1 001 11 1</td>
<td>VJCVT</td>
<td>FEAT_JSCVT</td>
</tr>
<tr>
<td>1 01x 01 -</td>
<td>VCVT (between floating-point and fixed-point, floating-point)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 01x 10 -</td>
<td>VCVT (between floating-point and fixed-point, floating-point)</td>
<td></td>
</tr>
<tr>
<td>1 01x 11 -</td>
<td>VCVT (between floating-point and fixed-point, floating-point)</td>
<td></td>
</tr>
<tr>
<td>1 100 01 0</td>
<td>VCVTR</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 100 01 1</td>
<td>VCVT (floating-point to integer, floating-point)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 100 10 0</td>
<td>VCVTR</td>
<td></td>
</tr>
<tr>
<td>1 100 10 1</td>
<td>VCVT (floating-point to integer, floating-point)</td>
<td></td>
</tr>
<tr>
<td>1 100 11 0</td>
<td>VCVTR</td>
<td></td>
</tr>
<tr>
<td>1 100 11 1</td>
<td>VCVT (floating-point to integer, floating-point)</td>
<td></td>
</tr>
<tr>
<td>1 101 01 0</td>
<td>VCVTR</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 101 01 1</td>
<td>VCVT (floating-point to integer, floating-point)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 101 10 0</td>
<td>VCVTR</td>
<td></td>
</tr>
<tr>
<td>1 101 10 1</td>
<td>VCVT (floating-point to integer, floating-point)</td>
<td></td>
</tr>
<tr>
<td>1 101 11 0</td>
<td>VCVTR</td>
<td></td>
</tr>
<tr>
<td>1 101 11 1</td>
<td>VCVT (floating-point to integer, floating-point)</td>
<td></td>
</tr>
</tbody>
</table>
Floating-point move immediate

This section describes the encoding of the Floating-point move immediate instruction class. The encodings in this section are decoded from Floating-point data-processing on page F3-7329.

```
<table>
<thead>
<tr>
<th>opcode</th>
<th>size</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td></td>
<td>VCVT (between floating-point and fixed-point, floating-point) FEAT_FP16</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>VCVT (between floating-point and fixed-point, floating-point) -</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>VCVT (between floating-point and fixed-point, floating-point) -</td>
</tr>
</tbody>
</table>
```

Floating-point data-processing (three registers)

This section describes the encoding of the Floating-point data-processing (three registers) instruction class. The encodings in this section are decoded from Floating-point data-processing on page F3-7329.

```
<table>
<thead>
<tr>
<th>size</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Unallocated. -</td>
</tr>
<tr>
<td>01</td>
<td>VMOV (immediate)- Half-precision scalar variant FEAT_FP16</td>
</tr>
<tr>
<td>10</td>
<td>VMOV (immediate)- Single-precision scalar variant -</td>
</tr>
<tr>
<td>11</td>
<td>VMOV (immediate)- Double-precision scalar variant -</td>
</tr>
</tbody>
</table>
```

Decode fields

```
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>o0:o1</td>
<td>size</td>
<td>o2</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>o0:o1 size o2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 11 1</td>
<td>VMLS (floating-point)- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>001 01 0</td>
<td>VNMLS- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>001 01 1</td>
<td>VNMLA- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>001 10 0</td>
<td>VNMLS- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>001 10 1</td>
<td>VNMLA- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>001 11 0</td>
<td>VNMLS- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>001 11 1</td>
<td>VNMLA- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>010 01 0</td>
<td>VMUL (floating-point)- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>010 01 1</td>
<td>VNMUL- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>010 10 0</td>
<td>VMUL (floating-point)- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>010 10 1</td>
<td>VNMUL- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>010 11 0</td>
<td>VMUL (floating-point)- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>010 11 1</td>
<td>VNMUL- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>011 01 0</td>
<td>VADD (floating-point)- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>011 01 1</td>
<td>VSUB (floating-point)- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>011 10 0</td>
<td>VADD (floating-point)- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>011 10 1</td>
<td>VSUB (floating-point)- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>011 11 0</td>
<td>VADD (floating-point)- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>011 11 1</td>
<td>VSUB (floating-point)- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>100 01 0</td>
<td>VDIV- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>100 10 0</td>
<td>VDIV- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>100 11 0</td>
<td>VDIV- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>101 01 0</td>
<td>VFNMNS- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>101 01 1</td>
<td>VFNMA- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>101 10 0</td>
<td>VFNMNS- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>101 10 1</td>
<td>VFNMA- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>101 11 0</td>
<td>VFNMNS- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>101 11 1</td>
<td>VFNMA- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>110 01 0</td>
<td>VFMA- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>110 01 1</td>
<td>VFMS- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>110 10 0</td>
<td>VFMA- Single-precision scalar variant</td>
<td>-</td>
</tr>
</tbody>
</table>
## F3.1.14 Additional Advanced SIMD and floating-point instructions

This section describes the encoding of the Additional Advanced SIMD and floating-point instructions group. The encodings in this section are decoded from System register access, Advanced SIMD, and floating-point instructions on page F3-7313.

### Decode fields

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>o0:o1 size o2</td>
<td>VFMS- Single-precision scalar variant -</td>
</tr>
</tbody>
</table>

Advanced SIMD three registers of the same length extension

This section describes the encoding of the Advanced SIMD three registers of the same length extension instruction class. The encodings in this section are decoded from Additional Advanced SIMD and floating-point instructions.

### Decode group or instruction page

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
<th>op5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xx</td>
<td>-</td>
<td>-</td>
<td>0x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>0</td>
<td>!= 00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>00xxxx</td>
<td>0</td>
<td>!= 00</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>110000</td>
<td>0</td>
<td>!= 00</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>111xxx</td>
<td>0</td>
<td>!= 00</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10x</td>
<td>-</td>
<td>0</td>
<td>00</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10x</td>
<td>-</td>
<td>1</td>
<td>0x</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
### T32 Instruction Set Encoding

**F3.1 T32 instruction set encoding**

---

**Decode fields**

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
<th>Q</th>
<th>U</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>0x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>VCADD- 64-bit SIMD vector variant</td>
<td>FEAT_FCMA</td>
</tr>
<tr>
<td>x1</td>
<td>0x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td>0x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>VCADD- 128-bit SIMD vector variant</td>
<td>FEAT_FCMA</td>
</tr>
<tr>
<td>x1</td>
<td>0x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0x</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0x</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>VMMLA</td>
<td>FEAT_AA32BF16</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VDOT (vector)- 64-bit SIMD vector variant</td>
<td>FEAT_AA32BF16</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VDOT (vector)- 128-bit SIMD vector variant</td>
<td>FEAT_AA32BF16</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>VFMAL (vector)</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>VSMMLA</td>
<td>FEAT_AA3218MM</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>VUMMLA</td>
<td>FEAT_AA3218MM</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VSDOT (vector)- 64-bit SIMD vector variant</td>
<td>FEAT_DotProd</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>VUDOT (vector)- 64-bit SIMD vector variant</td>
<td>FEAT_DotProd</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VSDOT (vector)- 128-bit SIMD vector variant</td>
<td>FEAT_DotProd</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>VUDOT (vector)- 128-bit SIMD vector variant</td>
<td>FEAT_DotProd</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>VFMAB, VFMAT (BFloat16, vector)</td>
<td>FEAT_AA32BF16</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td></td>
</tr>
</tbody>
</table>

---

**ID081822**

---

**Non-Confidential**
Floating-point conditional select

This section describes the encoding of the Floating-point conditional select instruction class. The encodings in this section are decoded from Additional Advanced SIMD and floating-point instructions on page F3-7334.

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
<th>Q</th>
<th>U</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>VFMSL (vector)</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>VUSMMLA</td>
<td>FEAT_AA32I8MM</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VUSDOT (vector)- 64-bit SIMD vector variant</td>
<td>FEAT_AA32I8MM</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>VUSDOT (vector)- 128-bit SIMD vector variant</td>
<td>FEAT_AA32I8MM</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>1x</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>VCMLA</td>
<td>FEAT_FCMA</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
Floating-point minNum/maxNum

This section describes the encoding of the Floating-point minNum/maxNum instruction class. The encodings in this section are decoded from Additional Advanced SIMD and floating-point instructions on page F3-7334.

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 0 1</td>
<td>D 0 0</td>
<td>Vn 1 0</td>
<td>!=00</td>
<td>N 0</td>
<td>M 0</td>
<td>Vm</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>size  op</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 0</td>
<td>VMAXNM- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>01 1</td>
<td>VMINNM- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>10 0</td>
<td>VMAXNM- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>10 1</td>
<td>VMINNM- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>11 0</td>
<td>VMAXNM- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>11 1</td>
<td>VMINNM- Double-precision scalar variant</td>
<td>-</td>
</tr>
</tbody>
</table>

Floating-point extraction and insertion

This section describes the encoding of the Floating-point extraction and insertion instruction class. The encodings in this section are decoded from Additional Advanced SIMD and floating-point instructions on page F3-7334.

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
<th>12 11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 0 1</td>
<td>D 1 0</td>
<td>0 0 0</td>
<td>0</td>
<td>Vd 1 0</td>
<td>!=00</td>
<td>op 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>size  op</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10 0</td>
<td>VMOVX</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>10 1</td>
<td>VINS</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>11</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

Floating-point directed convert to integer

This section describes the encoding of the Floating-point directed convert to integer instruction class. The encodings in this section are decoded from Additional Advanced SIMD and floating-point instructions on page F3-7334.
### Advanced SIMD and floating-point multiply with accumulate

This section describes the encoding of the Advanced SIMD and floating-point multiply with accumulate instruction class. The encodings in this section are decoded from *Additional Advanced SIMD and floating-point instructions on page F3-7334*.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>o1</strong> RM size op</td>
<td>VRINTA (floating-point)- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 - != 00 1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0 00 01 0</td>
<td>VRINTA (floating-point)- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>0 00 10 0</td>
<td>VRINTA (floating-point)- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>0 00 11 0</td>
<td>VRINTN (floating-point)- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 01 01 0</td>
<td>VRINTN (floating-point)- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>0 01 11 0</td>
<td>VRINTN (floating-point)- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>0 10 01 0</td>
<td>VRINTP (floating-point)- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 10 10 0</td>
<td>VRINTP (floating-point)- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>0 10 11 0</td>
<td>VRINTP (floating-point)- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>0 11 01 0</td>
<td>VRINTM (floating-point)- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 11 10 0</td>
<td>VRINTM (floating-point)- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>0 11 11 0</td>
<td>VRINTM (floating-point)- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>1 00 01 -</td>
<td>VCVTA (floating-point)- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 00 10 -</td>
<td>VCVTA (floating-point)- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>1 00 11 -</td>
<td>VCVTA (floating-point)- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>1 01 01 -</td>
<td>VCVTN (floating-point)- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 01 10 -</td>
<td>VCVTN (floating-point)- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>1 01 11 -</td>
<td>VCVTN (floating-point)- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>1 10 01 -</td>
<td>VCVTP (floating-point)- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 10 10 -</td>
<td>VCVTP (floating-point)- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>1 10 11 -</td>
<td>VCVTP (floating-point)- Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>1 11 01 -</td>
<td>VCVTM (floating-point)- Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 11 10 -</td>
<td>VCVTM (floating-point)- Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>1 11 11 -</td>
<td>VCVTM (floating-point)- Double-precision scalar variant</td>
<td>-</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>Q</th>
<th>U</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>VCMLA (by element)- 128-bit SIMD vector variant of half-precision floating-point</td>
<td>FEAT_FCMA</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>1</td>
<td>-</td>
<td>VFMAL (by scalar)</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>1</td>
<td>-</td>
<td>VFMSL (by scalar)</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>1</td>
<td>-</td>
<td>VFMAB, VFMAT (BFloat16, by scalar)</td>
<td>FEAT_AA32BF16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>VCMLA (by element)- 64-bit SIMD vector variant</td>
<td>FEAT_FCMA</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>VCMLA (by element)- 128-bit SIMD vector variant</td>
<td>FEAT_FCMA</td>
</tr>
</tbody>
</table>

### Advanced SIMD and floating-point dot product

This section describes the encoding of the Advanced SIMD and floating-point dot product instruction class. The encodings in this section are decoded from Additional Advanced SIMD and floating-point instructions on page F3-7334.

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>op4</th>
<th>Q</th>
<th>U</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VDOT (by element)- 64-bit SIMD vector variant</td>
<td>FEAT_AA32BF16</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VDOT (by element)- 128-bit SIMD vector variant</td>
<td>FEAT_AA32BF16</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VSDOT (by element)- 64-bit SIMD vector variant</td>
<td>FEAT_DotProd</td>
</tr>
</tbody>
</table>
F3.1.15 Load/store dual, load/store exclusive, load-acquire/store-release, and table branch

This section describes the encoding of the Load/store dual, load/store exclusive, load-acquire/store-release, and table branch group. The encodings in this section are decoded from 32-bit on page F3-7307.

This decode also imposes the constraint:

• op0<1> == 1.

Table F3-16 Encoding table for the Load/store dual, load/store exclusive, load-acquire/store-release, and table branch group

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>op4</th>
<th>Q</th>
<th>U</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>VUDOT (by element)- 64-bit SIMD vector variant</td>
<td>FEAT_DotProd</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VSDOT (by element)- 128-bit SIMD vector variant</td>
<td>FEAT_DotProd</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>VUDOT (by element)- 128-bit SIMD vector variant</td>
<td>FEAT_DotProd</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VUSDOT (by element)- 64-bit SIMD vector variant</td>
<td>FEAT_AA32I8MM</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>VUSDOT (by element)- 64-bit SIMD vector variant</td>
<td>FEAT_AA32I8MM</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VUSDOT (by element)- 128-bit SIMD vector variant</td>
<td>FEAT_AA32I8MM</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>VUSDOT (by element)- 128-bit SIMD vector variant</td>
<td>FEAT_AA32I8MM</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

F3-7340
Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential
Load/store exclusive

This section describes the encoding of the Load/store exclusive instruction class. The encodings in this section are decoded from Load/store dual, load/store exclusive, load-acquire/store-release, and table branch on page F3-7340.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x10</td>
<td>-</td>
<td>!= 1111</td>
<td>-</td>
</tr>
<tr>
<td>1x11</td>
<td>-</td>
<td>!= 1111</td>
<td>-</td>
</tr>
<tr>
<td>!= 0xx0</td>
<td>-</td>
<td>1111</td>
<td>-</td>
</tr>
</tbody>
</table>

Load/store dual (immediate) on page F3-7343
Load/store dual (immediate, pre-indexed) on page F3-7343
LDRD (literal)

Load/store exclusive byte/half/dual

This section describes the encoding of the Load/store exclusive byte/half/dual instruction class. The encodings in this section are decoded from Load/store dual, load/store exclusive, load-acquire/store-release, and table branch on page F3-7340.

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | | 0 | 12 | 11 | 8 | 7 | | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | L | Rn | Rt | Rd | imm8 |

Decide fields

<table>
<thead>
<tr>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Decide fields

<table>
<thead>
<tr>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>L sz</td>
</tr>
<tr>
<td>0 00</td>
</tr>
<tr>
<td>0 01</td>
</tr>
<tr>
<td>0 10</td>
</tr>
<tr>
<td>0 11</td>
</tr>
<tr>
<td>1 00</td>
</tr>
</tbody>
</table>
Load-acquire / Store-release

This section describes the encoding of the Load-acquire / Store-release instruction class. The encodings in this section are decoded from *Load/store dual, load/store exclusive, load-acquire/store-release, and table branch* on page F3-7340.

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 | 12 | 11 | 8 7 6 5 4 3 0 |
| 1 1 1 0 1 0 0 1 1 0 | L | Rn | Rt | Rt2 | op | sz | Rd |

| 1 | 01 | LDREXH |
| 1 | 10 | Unallocated. |
| 1 | 11 | LDREXD |

Load/store dual (immediate, post-indexed)

This section describes the encoding of the Load/store dual (immediate, post-indexed) instruction class. The encodings in this section are decoded from *Load/store dual, load/store exclusive, load-acquire/store-release, and table branch* on page F3-7340.

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 | 12 | 11 | 8 7 6 5 4 3 0 |
| 1 1 1 0 1 0 0 1 1 0 | L | Rn | Rt | Rt2 | op | sz | Rd |

| 0 0 0 00 | STLB |
| 0 0 0 01 | STLH |
| 0 0 0 10 | STL |
| 0 0 0 11 | Unallocated. |
| 0 1 0 00 | STLEXB |
| 0 1 0 01 | STLEXH |
| 0 1 0 10 | STLEX |
| 0 1 0 11 | STLEXD |
| 1 0 0 00 | LDAB |
| 1 0 0 01 | LDAH |
| 1 0 0 10 | LDA |
| 1 0 0 11 | Unallocated. |
| 1 1 0 00 | LDAEXB |
| 1 1 0 01 | LDAEXH |
| 1 1 0 10 | LDAEX |
| 1 1 0 11 | LDAEXD |
### Load/store dual (immediate)

This section describes the encoding of the Load/store dual (immediate) instruction class. The encodings in this section are decoded from *Load/store dual, load/store exclusive, load-acquire/store-release, and table branch* on page F3-7340.

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4 3</th>
<th>0</th>
<th>15 12</th>
<th>11 8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 0 0 0</td>
<td>U</td>
<td>1 1 L</td>
<td>l=1111</td>
<td>Rt</td>
<td>Rt2</td>
<td>imm8</td>
</tr>
</tbody>
</table>

### Load/store dual (immediate, pre-indexed)

This section describes the encoding of the Load/store dual (immediate, pre-indexed) instruction class. The encodings in this section are decoded from *Load/store dual, load/store exclusive, load-acquire/store-release, and table branch* on page F3-7340.

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4 3</th>
<th>0</th>
<th>15 12</th>
<th>11 8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 0 0 1</td>
<td>U</td>
<td>1 0 L</td>
<td>l=1111</td>
<td>Rt</td>
<td>Rt2</td>
<td>imm8</td>
</tr>
</tbody>
</table>

### F3.1.16 Branches and miscellaneous control

This section describes the encoding of the Branches and miscellaneous control group. The encodings in this section are decoded from *32-bit* on page F3-7307.
Hints

This section describes the encoding of the Hints instruction class. The encodings in this section are decoded from Branches and miscellaneous control on page F3-7343.
Change processor state

This section describes the encoding of the Change processor state instruction class. The encodings in this section are decoded from *Branches and miscellaneous control* on page F3-7343.
This section describes the encoding of the Miscellaneous system instruction class. The encodings in this section are decoded from Branches and miscellaneous control on page F3-7343.

### Miscellaneous system

This section describes the encoding of the Miscellaneous system instruction class. The encodings in this section are decoded from Branches and miscellaneous control on page F3-7343.

### Exception return

This section describes the encoding of the Exception return instruction class. The encodings in this section are decoded from Branches and miscellaneous control on page F3-7343.
T32 Instruction Set Encoding
F3.1 T32 instruction set encoding

This section describes the encoding of the DCPS instruction class. The encodings in this section are decoded from Branches and miscellaneous control on page F3-7343.

```plaintext
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>imm8</td>
<td></td>
</tr>
</tbody>
</table>
```

### DCPS

This section describes the encoding of the DCPS instruction class. The encodings in this section are decoded from Branches and miscellaneous control on page F3-7343.

```plaintext
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm10</td>
<td>opt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

#### Decode fields

**Rn:imm8**

- 111100000000 SUB, SUBS (immediate)
- 111100000000 ERET

### Exception generation

This section describes the encoding of the Exception generation instruction class. The encodings in this section are decoded from Branches and miscellaneous control on page F3-7343.

```plaintext
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>imm10</td>
<td>opt</td>
<td></td>
</tr>
</tbody>
</table>
```

#### Decode fields

**imm4**  **imm10**  **opt**

- 1111 - - Unallocated.
- 1111 00000000 00 Unallocated.
- 1111 00000000 01 DCPS1
- 1111 00000000 10 DCPS2
- 1111 00000000 11 DCPS3
This section describes the encoding of the Data-processing (plain binary immediate) group. The encodings in this section are decoded from 32-bit on page F3-7307.

Table F3-18 Encoding table for the Data-processing (plain binary immediate) group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>Data-processing (simple immediate)</td>
</tr>
<tr>
<td>0</td>
<td>Data-processing (simple immediate)</td>
</tr>
<tr>
<td>0 0x</td>
<td>Move Wide (16-bit immediate) on page F3-7349</td>
</tr>
<tr>
<td>0 10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0 11</td>
<td>Saturate, Bitfield on page F3-7349</td>
</tr>
</tbody>
</table>

Data-processing (simple immediate)

This section describes the encoding of the Data-processing (simple immediate) instruction class. The encodings in this section are decoded from Data-processing (plain binary immediate).
Move Wide (16-bit immediate)

This section describes the encoding of the Move Wide (16-bit immediate) instruction class. The encodings in this section are decoded from Data-processing (plain binary immediate) on page F3-7348.

Saturate, Bitfield

This section describes the encoding of the Saturate, Bitfield instruction class. The encodings in this section are decoded from Data-processing (plain binary immediate) on page F3-7348.
This section describes the encoding of the Advanced SIMD element or structure load/store group. The encodings in this section are decoded from 32-bit on page F3-7307.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the T32 Advanced SIMD and floating-point instructions and their encoding on page F3-7371.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Rn</th>
<th>imm3:imm2</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SSAT- Logical shift left variant</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>!0</td>
<td>00000</td>
<td>SSAT- Arithmetic shift right variant</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>0000</td>
<td>SSAT16</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>-</td>
<td>SBFX</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>!= 1111</td>
<td>-</td>
<td>BFI</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>1111</td>
<td>-</td>
<td>BFC</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>-</td>
<td>USAT- Logical shift left variant</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>!= 0000</td>
<td>-</td>
<td>USAT- Arithmetic shift right variant</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>-</td>
<td>0000</td>
<td>USAT16</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>-</td>
<td>-</td>
<td>UBFX</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td></td>
</tr>
</tbody>
</table>

### Table F3-19 Encoding table for the Advanced SIMD element or structure load/store group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>!= 11</td>
</tr>
</tbody>
</table>
Advanced SIMD load/store multiple structures

This section describes the encoding of the Advanced SIMD load/store multiple structures instruction class. The encodings in this section are decoded from Advanced SIMD element or structure load/store on page F3-7350.

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
<td>0 0 1 0</td>
<td>D</td>
<td>L</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>itype</td>
<td>size</td>
<td>align</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L</th>
<th>itype</th>
<th>Rm</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000x</td>
<td>!= 11x1</td>
<td>VST4 (multiple 4-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>000x</td>
<td>1101</td>
<td>VST4 (multiple 4-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>000x</td>
<td>1111</td>
<td>VST4 (multiple 4-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>!= 11x1</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>1101</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>1111</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>!= 11x1</td>
<td>VST2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>1101</td>
<td>VST2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>1111</td>
<td>VST2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>010x</td>
<td>!= 11x1</td>
<td>VST3 (multiple 3-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>010x</td>
<td>1101</td>
<td>VST3 (multiple 3-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>010x</td>
<td>1111</td>
<td>VST3 (multiple 3-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>!= 11x1</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>1101</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>1111</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>!= 11x1</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>1101</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>1111</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>100x</td>
<td>!= 11x1</td>
<td>VST2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>100x</td>
<td>1101</td>
<td>VST2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>100x</td>
<td>1111</td>
<td>VST2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td>!= 11x1</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td>1101</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td>1111</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>1</td>
<td>000x</td>
<td>!= 11x1</td>
<td>VLD4 (multiple 4-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>000x</td>
<td>1101</td>
<td>VLD4 (multiple 4-element structures)</td>
</tr>
</tbody>
</table>
```
### Advanced SIMD load single structure to all lanes

This section describes the encoding of the Advanced SIMD load single structure to all lanes instruction class. The encodings in this section are decoded from *Advanced SIMD element or structure load/store* on page F3-7350.

<table>
<thead>
<tr>
<th>L</th>
<th>itype</th>
<th>Rm</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>000x</td>
<td>1111</td>
<td>VLD4 (multiple 4-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>!= 11x1</td>
<td>VLD1 (multiple single elements)</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>1101</td>
<td>VLD1 (multiple single elements)</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>1111</td>
<td>VLD1 (multiple single elements)</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>!= 11x1</td>
<td>VLD2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>1101</td>
<td>VLD2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>1111</td>
<td>VLD2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>010x</td>
<td>!= 11x1</td>
<td>VLD3 (multiple 3-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>010x</td>
<td>1101</td>
<td>VLD3 (multiple 3-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>010x</td>
<td>1111</td>
<td>VLD3 (multiple 3-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
<td>!= 11x1</td>
<td>VLD1 (multiple single elements)</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
<td>1101</td>
<td>VLD1 (multiple single elements)</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
<td>1111</td>
<td>VLD1 (multiple single elements)</td>
</tr>
<tr>
<td>1</td>
<td>0111</td>
<td>!= 11x1</td>
<td>VLD1 (multiple single elements)</td>
</tr>
<tr>
<td>1</td>
<td>0111</td>
<td>1101</td>
<td>VLD1 (multiple single elements)</td>
</tr>
<tr>
<td>1</td>
<td>0111</td>
<td>1111</td>
<td>VLD1 (multiple single elements)</td>
</tr>
<tr>
<td>1</td>
<td>100x</td>
<td>!= 11x1</td>
<td>VLD2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>100x</td>
<td>1101</td>
<td>VLD2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>100x</td>
<td>1111</td>
<td>VLD2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
<td>!= 11x1</td>
<td>VLD1 (multiple single elements)</td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
<td>1101</td>
<td>VLD1 (multiple single elements)</td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
<td>1111</td>
<td>VLD1 (multiple single elements)</td>
</tr>
</tbody>
</table>

*Unallocated.*
Advanced SIMD load/store single structure to one lane

This section describes the encoding of the Advanced SIMD load/store single structure to one lane instruction class. The encodings in this section are decoded from Advanced SIMD element or structure load/store on page F3-7350.

**Decode fields**

<table>
<thead>
<tr>
<th>L</th>
<th>N</th>
<th>a</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>!= 11x1</td>
<td>VLD1 (single element to all lanes)</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>1101</td>
<td>VLD1 (single element to all lanes)</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>1111</td>
<td>VLD1 (single element to all lanes)</td>
</tr>
<tr>
<td>101</td>
<td>-</td>
<td>!= 11x1</td>
<td>VLD2 (single 2-element structure to all lanes)</td>
</tr>
<tr>
<td>101</td>
<td>-</td>
<td>1101</td>
<td>VLD2 (single 2-element structure to all lanes)</td>
</tr>
<tr>
<td>101</td>
<td>-</td>
<td>1111</td>
<td>VLD2 (single 2-element structure to all lanes)</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>!= 11x1</td>
<td>VLD3 (single 3-element structure to all lanes)</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>1101</td>
<td>VLD3 (single 3-element structure to all lanes)</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>1111</td>
<td>VLD3 (single 3-element structure to all lanes)</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>111</td>
<td>-</td>
<td>!= 11x1</td>
<td>VLD4 (single 4-element structure to all lanes)</td>
</tr>
<tr>
<td>111</td>
<td>-</td>
<td>1101</td>
<td>VLD4 (single 4-element structure to all lanes)</td>
</tr>
<tr>
<td>111</td>
<td>-</td>
<td>1111</td>
<td>VLD4 (single 4-element structure to all lanes)</td>
</tr>
</tbody>
</table>

**Instruction page**

- VST1 (single element from one lane)
- VST2 (single 2-element structure from one lane)
### T32 Instruction Set Encoding

#### F3.1 T32 instruction set encoding

<table>
<thead>
<tr>
<th>L</th>
<th>size</th>
<th>N</th>
<th>Rm</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>01</td>
<td>1111</td>
<td>VST2 (single 2-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>10</td>
<td>!= 11x1</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>10</td>
<td>1101</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>10</td>
<td>1111</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>11</td>
<td>!= 11x1</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>11</td>
<td>1101</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>11</td>
<td>1111</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>00</td>
<td>!= 11x1</td>
<td>VST1 (single element from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>00</td>
<td>1101</td>
<td>VST1 (single element from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>00</td>
<td>1111</td>
<td>VST1 (single element from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01</td>
<td>!= 11x1</td>
<td>VST2 (single 2-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01</td>
<td>1101</td>
<td>VST2 (single 2-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01</td>
<td>1111</td>
<td>VST2 (single 2-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>10</td>
<td>!= 11x1</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>10</td>
<td>1101</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>10</td>
<td>1111</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11</td>
<td>!= 11x1</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11</td>
<td>1101</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11</td>
<td>1111</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>00</td>
<td>!= 11x1</td>
<td>VST1 (single element from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>00</td>
<td>1101</td>
<td>VST1 (single element from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>00</td>
<td>1111</td>
<td>VST1 (single element from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01</td>
<td>!= 11x1</td>
<td>VST2 (single 2-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01</td>
<td>1101</td>
<td>VST2 (single 2-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01</td>
<td>1111</td>
<td>VST2 (single 2-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
<td>!= 11x1</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
<td>1101</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
<td>1111</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
<td>!= 11x1</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
<td>1101</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
<td>1111</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>00</td>
<td>!= 11x1</td>
<td>VLD1 (single element to one lane)</td>
</tr>
</tbody>
</table>
## T32 Instruction Set Encoding

**F3.1 T32 instruction set encoding**

### Decode fields

<table>
<thead>
<tr>
<th>L</th>
<th>size</th>
<th>N</th>
<th>Rm</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>00</td>
<td>1101</td>
<td>VLD1 (single element to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>00</td>
<td>1111</td>
<td>VLD1 (single element to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>!= 11x1</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>1101</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>1111</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>10</td>
<td>!= 11x1</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>10</td>
<td>1101</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>10</td>
<td>1111</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>11</td>
<td>!= 11x1</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>11</td>
<td>1101</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>11</td>
<td>1111</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>00</td>
<td>!= 11x1</td>
<td>VLD1 (single element to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>00</td>
<td>1101</td>
<td>VLD1 (single element to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>00</td>
<td>1111</td>
<td>VLD1 (single element to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>01</td>
<td>!= 11x1</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>01</td>
<td>1101</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>01</td>
<td>1111</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>10</td>
<td>!= 11x1</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>10</td>
<td>1101</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>10</td>
<td>1111</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11</td>
<td>!= 11x1</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11</td>
<td>1101</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11</td>
<td>1111</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>00</td>
<td>!= 11x1</td>
<td>VLD1 (single element to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>00</td>
<td>1101</td>
<td>VLD1 (single element to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>01</td>
<td>!= 11x1</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>01</td>
<td>1101</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>01</td>
<td>1111</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>!= 11x1</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>1101</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>1111</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
</tbody>
</table>
### F3.1.19 Load/store single

This section describes the encoding of the Load/store single group. The encodings in this section are decoded from 32-bit on page F3-7307.

<table>
<thead>
<tr>
<th>L</th>
<th>size</th>
<th>N</th>
<th>Rm</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td>!= 111</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td>1101</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td>1111</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
</tbody>
</table>

This decode also imposes the constraint:

- \( \text{op0<1>:op1 } \neq 10 \).

### Table F3-20 Encoding table for the Load/store single group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2 op3</td>
<td></td>
</tr>
<tr>
<td>00 - != 1111 000000</td>
<td>Load/store, unsigned (register offset) on page F3-7357</td>
</tr>
<tr>
<td>00 - != 1111 000001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00 - != 1111 00001x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00 - != 1111 0001xx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00 - != 1111 001xxx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00 - != 1111 01xxxx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00 - != 1111 1000xx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00 - != 1111 1011xx</td>
<td>Load/store, unsigned (immediate, post-indexed) on page F3-7357</td>
</tr>
<tr>
<td>00 - != 1111 1100xx</td>
<td>Load/store, unsigned (negative immediate) on page F3-7358</td>
</tr>
<tr>
<td>00 - != 1111 1110xx</td>
<td>Load/store, unsigned (unprivileged) on page F3-7358</td>
</tr>
<tr>
<td>00 - != 1111 1111xx</td>
<td>Load/store, unsigned (immediate, pre-indexed) on page F3-7359</td>
</tr>
<tr>
<td>01 - != 1111 -</td>
<td>Load/store, unsigned (positive immediate) on page F3-7359</td>
</tr>
<tr>
<td>0x - 1111 -</td>
<td>Load, unsigned (literal) on page F3-7360</td>
</tr>
<tr>
<td>10 1 != 1111 000000</td>
<td>Load/store, signed (register offset) on page F3-7360</td>
</tr>
<tr>
<td>10 1 != 1111 000001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10 1 != 1111 00001x</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
Load/store, unsigned (register offset)

This section describes the encoding of the Load/store, unsigned (register offset) instruction class. The encodings in this section are decoded from Load/store single on page F3-7356.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
</tr>
</tbody>
</table>

Load/store, unsigned (immediate, post-indexed)

This section describes the encoding of the Load/store, unsigned (immediate, post-indexed) instruction class. The encodings in this section are decoded from Load/store single on page F3-7356.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>L</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
</tr>
</tbody>
</table>
Load/store, unsigned (negative immediate)

This section describes the encoding of the Load/store, unsigned (negative immediate) instruction class. The encodings in this section are decoded from *Load/store single on page F3-7356*.

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0 15</th>
<th>12 11 10 9 8</th>
<th>7 0 1111</th>
<th>Rt 1 0</th>
<th>U 1</th>
<th>imm8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>size L</td>
<td></td>
</tr>
<tr>
<td>00 0</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00 1</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>01 0</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01 1</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>10 0</td>
<td>STR (immediate)</td>
</tr>
<tr>
<td>10 1</td>
<td>LDR (immediate)</td>
</tr>
<tr>
<td>11 -</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

Load/store, unsigned (unprivileged)

This section describes the encoding of the Load/store, unsigned (unprivileged) instruction class. The encodings in this section are decoded from *Load/store single on page F3-7356*.

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0 15</th>
<th>12 11 10 9 8</th>
<th>7 0 1111</th>
<th>Rt 1 1 0 0</th>
<th>imm8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>size L Rt</td>
<td></td>
</tr>
<tr>
<td>00 0</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00 1 != 1111</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00 1 1111</td>
<td>PLD, PLDW (immediate)- Preload read variant</td>
</tr>
<tr>
<td>01 0</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01 1 != 1111</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>01 1 1111</td>
<td>PLD, PLDW (immediate)- Preload write variant</td>
</tr>
<tr>
<td>10 0</td>
<td>STR (immediate)</td>
</tr>
<tr>
<td>10 1 -</td>
<td>LDR (immediate)</td>
</tr>
<tr>
<td>11 - -</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
Load/store, unsigned (immediate, pre-indexed)

This section describes the encoding of the Load/store, unsigned (immediate, pre-indexed) instruction class. The encodings in this section are decoded from *Load/store single on page F3-7356.*

Load/store, unsigned (positive immediate)

This section describes the encoding of the Load/store, unsigned (positive immediate) instruction class. The encodings in this section are decoded from *Load/store single on page F3-7356.*
Load, unsigned (literal)

This section describes the encoding of the Load, unsigned (literal) instruction class. The encodings in this section are decoded from *Load/store single* on page F3-7356.

<table>
<thead>
<tr>
<th>size</th>
<th>L</th>
<th>Rt</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>-</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>!111</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1111</td>
<td>PLD, PLDW (immediate)- Preload read variant</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>-</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>!111</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1111</td>
<td>PLD, PLDW (immediate)- Preload write variant</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>-</td>
<td>STR (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>-</td>
<td>LDR (immediate)</td>
</tr>
</tbody>
</table>

Load/store, signed (register offset)

This section describes the encoding of the Load/store, signed (register offset) instruction class. The encodings in this section are decoded from *Load/store single* on page F3-7356.

<table>
<thead>
<tr>
<th>size</th>
<th>L</th>
<th>Rt</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>1</td>
<td>1111</td>
<td>PLD (literal)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>!111</td>
<td>LDRB (literal)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>!111</td>
<td>LDRH (literal)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>-</td>
<td>LDR (literal)</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
Load/store, signed (immediate, post-indexed)

This section describes the encoding of the Load/store, signed (immediate, post-indexed) instruction class. The encodings in this section are decoded from Load/store single on page F3-7356.

Load/store, signed (negative immediate)

This section describes the encoding of the Load/store, signed (negative immediate) instruction class. The encodings in this section are decoded from Load/store single on page F3-7356.
Load/store, signed (unprivileged)

This section describes the encoding of the Load/store, signed (unprivileged) instruction class. The encodings in this section are decoded from Load/store single on page F3-7356.

Load/store, signed (immediate, pre-indexed)

This section describes the encoding of the Load/store, signed (immediate, pre-indexed) instruction class. The encodings in this section are decoded from Load/store single on page F3-7356.
Load/store, signed (positive immediate)

This section describes the encoding of the Load/store, signed (positive immediate) instruction class. The encodings in this section are decoded from Load/store single on page F3-7356.

```
| 15 14 13 12|11 10 9 8 7| 6 5 4 3 0 |15 12|11 10 9 8 7 | 0 |
| 1 1 1 1 1 0 0 1 0 size | 1 | !=1111 | Rt | 1 1 U | 1 | imm8 |
```

Decode fields

<table>
<thead>
<tr>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
</tr>
<tr>
<td>00               LDRSB (immediate)</td>
</tr>
<tr>
<td>01               LDRSH (immediate)</td>
</tr>
<tr>
<td>1x               Unallocated.</td>
</tr>
</tbody>
</table>

Load, signed (literal)

This section describes the encoding of the Load, signed (literal) instruction class. The encodings in this section are decoded from Load/store single on page F3-7356.

```
| 15 14 13 12|11 10 9 8 7| 6 5 4 3 0 |15 12|11 10 9 8 7 | 0 |
| 1 1 1 1 1 0 0 1 1 size | 1 | !=1111 | Rt | 1 1 1 1 1 | imm12 |
```

Decode fields

<table>
<thead>
<tr>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>size  Rt</td>
</tr>
<tr>
<td>00 1111         LDRSB (literal)</td>
</tr>
<tr>
<td>00 1111         PLI (immediate, literal)</td>
</tr>
<tr>
<td>01 1111         LDRSH (immediate)</td>
</tr>
<tr>
<td>01 1111         Reserved hint, behaves as NOP.</td>
</tr>
</tbody>
</table>

Load, signed (literal)
F3.1.20 Data-processing (register)

This section describes the encoding of the Data-processing (register) group. The encodings in this section are decoded from 32-bit on page F3-7307.

<table>
<thead>
<tr>
<th>15</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>0</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111010</td>
<td>op1</td>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Decode fields

<table>
<thead>
<tr>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV, MOVS (register-shifted register)- Flag setting variant</td>
</tr>
<tr>
<td>Reserved hint, behaves as NOP.</td>
</tr>
<tr>
<td>Unallocated.</td>
</tr>
<tr>
<td>Unallocated.</td>
</tr>
<tr>
<td>Unallocated.</td>
</tr>
<tr>
<td>Register extends on page F3-7364</td>
</tr>
<tr>
<td>Parallel add-subtract on page F3-7365</td>
</tr>
<tr>
<td>Data-processing (two source registers) on page F3-7367</td>
</tr>
<tr>
<td>Unallocated.</td>
</tr>
<tr>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

Register extends

This section describes the encoding of the Register extends instruction class. The encodings in this section are decoded from Data-processing (register).
### Parallel add-subtract

This section describes the encoding of the Parallel add-subtract instruction class. The encodings in this section are decoded from *Data-processing (register)* on page F3-7364.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>U</td>
<td>Rn</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>op1</th>
<th>U</th>
<th>Rn</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>!= 1111 SXTAH</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1111 SXTH</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>!= 1111 UXTAH</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1111 UXTH</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>!= 1111 SXTAB16</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1111 SXTB16</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>!= 1111 UXTAB16</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1111 UXTB16</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>!= 1111 SXTAB</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1111 SXTB</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>!= 1111 UXTAB</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1111 UXTB</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>op1</th>
<th>U</th>
<th>H</th>
<th>S</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>UHADD8</td>
</tr>
<tr>
<td>000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SADD16</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>QADD16</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SHADD16</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>UADD16</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>UQADD16</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>UHADD16</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SASX</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>QASX</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SHAUX</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>UASX</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>UQASX</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>UHASX</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SSUB8</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>QSUB8</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SHSUB8</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>USUB8</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>UQSUB8</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>UHSUB8</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SSUB16</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>QSUB16</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SHSUB16</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>USUB16</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>UQSUB16</td>
</tr>
</tbody>
</table>
Data-processing (two source registers)

This section describes the encoding of the Data-processing (two source registers) instruction class. The encodings in this section are decoded from *Data-processing (register)* on page F3-7364.

<table>
<thead>
<tr>
<th>op1</th>
<th>U</th>
<th>H</th>
<th>S</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>UHSUB16</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SSAX</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>QSAX</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SHSAX</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>USAX</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>UQSAX</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>UHSAX</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>111</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1</td>
<td>op2</td>
<td>QADD</td>
</tr>
<tr>
<td>000</td>
<td>00</td>
<td>QADD</td>
</tr>
<tr>
<td>000</td>
<td>01</td>
<td>QDADD</td>
</tr>
<tr>
<td>000</td>
<td>10</td>
<td>QSUB</td>
</tr>
<tr>
<td>000</td>
<td>11</td>
<td>QDSUB</td>
</tr>
<tr>
<td>001</td>
<td>00</td>
<td>REV</td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td>REV16</td>
</tr>
<tr>
<td>001</td>
<td>10</td>
<td>RBIT</td>
</tr>
<tr>
<td>001</td>
<td>11</td>
<td>REVSH</td>
</tr>
<tr>
<td>010</td>
<td>00</td>
<td>SEL</td>
</tr>
<tr>
<td>010</td>
<td>01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>010</td>
<td>1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>CLZ</td>
</tr>
</tbody>
</table>
The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings on page K1-11574.

F3.1.21 Multiply, multiply accumulate, and absolute difference

This section describes the encoding of the Multiply, multiply accumulate, and absolute difference group. The encodings in this section are decoded from 32-bit on page F3-7307.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1 op2</td>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>011 01</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>011 1x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>100 00</td>
<td>CRC32-CRC32B variant</td>
<td>FEAT_CRC32</td>
</tr>
<tr>
<td>100 01</td>
<td>CRC32-CRC32H variant</td>
<td>FEAT_CRC32</td>
</tr>
<tr>
<td>100 10</td>
<td>CRC32-CRC32W variant</td>
<td>FEAT_CRC32</td>
</tr>
<tr>
<td>100 11</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>-</td>
</tr>
<tr>
<td>101 00</td>
<td>CRC32C-CRC32CB variant</td>
<td>FEAT_CRC32</td>
</tr>
<tr>
<td>101 01</td>
<td>CRC32C-CRC32CH variant</td>
<td>FEAT_CRC32</td>
</tr>
<tr>
<td>101 10</td>
<td>CRC32C-CRC32CW variant</td>
<td>FEAT_CRC32</td>
</tr>
<tr>
<td>101 11</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>-</td>
</tr>
<tr>
<td>11x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

Table F3-22 Encoding table for the Multiply, multiply accumulate, and absolute difference group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>Multiply and absolute difference on page F3-7368</td>
</tr>
<tr>
<td>01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1x</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

Multiply and absolute difference

This section describes the encoding of the Multiply and absolute difference instruction class. The encodings in this section are decoded from Multiply, multiply accumulate, and absolute difference.
### T32 Instruction Set Encoding

#### F3.1 T32 instruction set encoding

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1 Ra op2</td>
<td>MLA, MLAS</td>
</tr>
<tr>
<td>000 1111 00</td>
<td></td>
</tr>
<tr>
<td>000 1111 01</td>
<td>MLS</td>
</tr>
<tr>
<td>000 1111 1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>000 1111 00</td>
<td>MUL, MULS</td>
</tr>
<tr>
<td>000 1111 01</td>
<td>SMLABB, SMLABT, SMLATB, SMLATT- SMLABB variant</td>
</tr>
<tr>
<td>000 1111 10</td>
<td>SMLABB, SMLABT, SMLATB, SMLATT- SMLABT variant</td>
</tr>
<tr>
<td>000 1111 11</td>
<td>SMLABB, SMLABT, SMLATB, SMLATT- SMLATB variant</td>
</tr>
<tr>
<td>001 1111 00</td>
<td>SMULBB, SMULBT, SMULTB, SMULTT- SMULBB variant</td>
</tr>
<tr>
<td>001 1111 01</td>
<td>SMULBB, SMULBT, SMULTB, SMULTT- SMULBT variant</td>
</tr>
<tr>
<td>001 1111 10</td>
<td>SMULBB, SMULBT, SMULTB, SMULTT- SMULTB variant</td>
</tr>
<tr>
<td>001 1111 11</td>
<td>SMULBB, SMULBT, SMULTB, SMULTT- SMULTT variant</td>
</tr>
<tr>
<td>010 1111 00</td>
<td>SMLAD, SMLADX- SMLAD variant</td>
</tr>
<tr>
<td>010 1111 01</td>
<td>SMLAD, SMLADX- SMLADX variant</td>
</tr>
<tr>
<td>010 1111 1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>010 1111 00</td>
<td>SMUAD, SMUADX- SMUAD variant</td>
</tr>
<tr>
<td>010 1111 01</td>
<td>SMUAD, SMUADX- SMUADX variant</td>
</tr>
<tr>
<td>011 1111 00</td>
<td>SMLAWB, SMLAWT- SMLAWB variant</td>
</tr>
<tr>
<td>011 1111 01</td>
<td>SMLAWB, SMLAWT- SMLAWT variant</td>
</tr>
<tr>
<td>011 1111 1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011 1111 00</td>
<td>SMULWB, SMULWT- SMULWB variant</td>
</tr>
<tr>
<td>011 1111 01</td>
<td>SMULWB, SMULWT- SMULWT variant</td>
</tr>
<tr>
<td>100 1111 00</td>
<td>SMLSD, SMLSDX- SMLSD variant</td>
</tr>
<tr>
<td>100 1111 01</td>
<td>SMLSD, SMLSDX- SMLSDX variant</td>
</tr>
<tr>
<td>100 1111 1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100 1111 00</td>
<td>SMUSD, SMUSDX- SMUSD variant</td>
</tr>
<tr>
<td>100 1111 01</td>
<td>SMUSD, SMUSDX- SMUSDX variant</td>
</tr>
<tr>
<td>101 1111 00</td>
<td>SMMLA, SMMLAR- SMMLA variant</td>
</tr>
</tbody>
</table>
## Decode fields

<table>
<thead>
<tr>
<th>op1</th>
<th>Ra</th>
<th>op2</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>!= 1111</td>
<td>01</td>
<td>SMMLA, SMMLAR- SMMLAR variant</td>
</tr>
<tr>
<td>101</td>
<td>-</td>
<td>1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>101</td>
<td>1111</td>
<td>00</td>
<td>SMMUL, SMMULR- SMMUL variant</td>
</tr>
<tr>
<td>101</td>
<td>1111</td>
<td>01</td>
<td>SMMUL, SMMULR- SMMULR variant</td>
</tr>
<tr>
<td>110</td>
<td>-</td>
<td>00</td>
<td>SMMLS, SMMLSR- SMMLS variant</td>
</tr>
<tr>
<td>110</td>
<td>-</td>
<td>01</td>
<td>SMMLS, SMMLSR- SMMLSR variant</td>
</tr>
<tr>
<td>110</td>
<td>-</td>
<td>1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>111</td>
<td>!= 1111</td>
<td>00</td>
<td>USADA8</td>
</tr>
<tr>
<td>111</td>
<td>-</td>
<td>01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>111</td>
<td>-</td>
<td>1x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>111</td>
<td>1111</td>
<td>00</td>
<td>USAD8</td>
</tr>
</tbody>
</table>
F3.2 About the T32 Advanced SIMD and floating-point instructions and their encoding

The Advanced SIMD and floating-point instructions are common to the T32 and A32 instruction sets. These instructions perform Advanced SIMD and floating-point operations on a common register file, the SIMD&FP register file. This means:

- In general, the instructions that load or store registers in this file, or move data between general-purpose registers and this register file, are common to the Advanced SIMD and floating-point instructions.
- There are distinct Advanced SIMD data-processing instructions and floating-point data-processing instructions.

All T32 Advanced SIMD and floating-point instructions have 32-bit encodings. Different groups of these instructions are decoded from different points in the 32-bit T32 instruction decode structure. Table F3-23 shows these instruction groups, and where each group is decoded from the overall T32 decode structure:

<table>
<thead>
<tr>
<th>Advanced SIMD and floating-point instruction group</th>
<th>T32 decode is from</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced SIMD and System register load/store and 64-bit move on page F3-7323</td>
<td>System register access, Advanced SIMD, and floating-point on page F3-7313</td>
</tr>
<tr>
<td>Floating-point data-processing on page F3-7329</td>
<td>System register access, Advanced SIMD, and floating-point on page F3-7313</td>
</tr>
<tr>
<td>Advanced SIMD and System register 32-bit move on page F3-7327</td>
<td>System register access, Advanced SIMD, and floating-point on page F3-7313</td>
</tr>
<tr>
<td>Advanced SIMD data-processing on page F3-7314</td>
<td>System register access, Advanced SIMD, and floating-point on page F3-7313</td>
</tr>
<tr>
<td>Advanced SIMD element or structure load/store on page F3-7350</td>
<td>32-bit on page F3-7307</td>
</tr>
</tbody>
</table>
F3.2 About the T32 Advanced SIMD and floating-point instructions and their encoding
Chapter F4
A32 Instruction SetEncoding

This chapter describes the encoding of the A32 instruction set. It contains the following sections:

- *A32 instruction set encoding* on page F4-7374.
- *About the A32 Advanced SIMD and floating-point instructions and their encoding* on page F4-7440.

In this chapter:

- In the decode tables, an entry of - for a field value means the value of the field does not affect the decoding.
- In the decode diagrams, a shaded field indicates that the bits in that field are not used in that level of decode.
F4.1 A32 instruction set encoding

The A32 instruction stream is a sequence of word-aligned words. Each A32 instruction is either a single 32-bit word in that stream.

Most A32 instructions can be conditional, with a condition determined by bits[31:28] of the instruction, the cond field. For more information see The Condition code field in A32 instruction encodings on page F1-7229. This applies to all instructions except those with the cond field equal to 0b111.

The behavior of an attempt to execute an unallocated instruction is described in UNDEFINED, UNPREDICTABLE, and CONSTRAINED UNPREDICTABLE instruction set space on page F1-7236.

For more information on A32 instruction encodings see Chapter F1 About the T32 and A32 Instruction Descriptions.

The A32 instruction encoding is:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>28</td>
<td>27</td>
<td>25</td>
<td>24</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table F4-1 Main encoding table for the A32 instruction set

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>op0</td>
</tr>
<tr>
<td>!111</td>
<td>00x</td>
</tr>
<tr>
<td>!111</td>
<td>010</td>
</tr>
<tr>
<td>!111</td>
<td>011 0</td>
</tr>
<tr>
<td>!111</td>
<td>011 1</td>
</tr>
<tr>
<td>-</td>
<td>10x</td>
</tr>
<tr>
<td>-</td>
<td>11x</td>
</tr>
<tr>
<td>1111</td>
<td>0xx</td>
</tr>
</tbody>
</table>

F4.1.1 Data-processing and miscellaneous instructions

This section describes the encoding of the Data-processing and miscellaneous instructions group. The encodings in this section are decoded from A32 instruction set encoding.
Multiply and Accumulate

This section describes the encoding of the Multiply and Accumulate instruction class. The encodings in this section are decoded from Data-processing and miscellaneous instructions on page F4-7374.

[31 28 27 26 25 24] [20 19] [16 15] [12 11] [8 7 6 5 4] [3 0] 

\(!=1111\) 0 0 0 0 opc S RdHi RdLo Rm 1 0 0 1 Rn 

cond

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2 op3 op4</td>
<td></td>
</tr>
<tr>
<td>0 - 1 != 00 1</td>
<td>Extra load/store on page F4-7376</td>
</tr>
<tr>
<td>0 0xxxx 1 00 1</td>
<td>Multiply and Accumulate on page F4-7375</td>
</tr>
<tr>
<td>0 1xxxx 1 00 1</td>
<td>Synchronization primitives and Load-Acquire/Store-Release on page F4-7379</td>
</tr>
<tr>
<td>0 10xx0 0 - -</td>
<td>Miscellaneous on page F4-7380</td>
</tr>
<tr>
<td>0 10xx0 1 - 0</td>
<td>Halfword Multiply and Accumulate on page F4-7376</td>
</tr>
<tr>
<td>0 != 10xx0 - - 0</td>
<td>Data-processing register (immediate shift) on page F4-7383</td>
</tr>
<tr>
<td>0 != 10xx0 0 - 1</td>
<td>Data-processing register (register shift) on page F4-7386</td>
</tr>
<tr>
<td>1 - - - -</td>
<td>Data-processing immediate on page F4-7387</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc S</td>
<td></td>
</tr>
<tr>
<td>000 -</td>
<td>MUL, MULS</td>
</tr>
<tr>
<td>001 -</td>
<td>MLA, MLAS</td>
</tr>
<tr>
<td>010 0</td>
<td>UMAAL</td>
</tr>
<tr>
<td>010 1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011 0</td>
<td>MLS</td>
</tr>
<tr>
<td>011 1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100 -</td>
<td>UMULL, UMULLS</td>
</tr>
</tbody>
</table>
### Halfword Multiply and Accumulate

This section describes the encoding of the Halfword Multiply and Accumulate instruction class. The encodings in this section are decoded from Data-processing and miscellaneous instructions on page F4-7374.

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>UMLAL, UMLALS</td>
</tr>
<tr>
<td>110</td>
<td>SMULL, SMULLS</td>
</tr>
<tr>
<td>111</td>
<td>SMLAL, SMLALS</td>
</tr>
</tbody>
</table>

#### F4.1.2 Extra load/store

This section describes the encoding of the Extra load/store group. The encodings in this section are decoded from Data-processing and miscellaneous instructions on page F4-7374.

<table>
<thead>
<tr>
<th>op0</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Load/Store Dual, Half, Signed Byte (register) on page F4-7377</td>
</tr>
<tr>
<td>1</td>
<td>Load/Store Dual, Half, Signed Byte (immediate, literal) on page F4-7377</td>
</tr>
</tbody>
</table>
## Load/Store Dual, Half, Signed Byte (register)

This section describes the encoding of the Load/Store Dual, Half, Signed Byte (register) instruction class. The encodings in this section are decoded from *Extra load/store on page F4-7376.*

<table>
<thead>
<tr>
<th>Cond</th>
<th>P</th>
<th>W</th>
<th>o1</th>
<th>op2</th>
<th>Rn</th>
<th>Rt</th>
<th>(0)</th>
<th>(0)</th>
<th>(0)</th>
<th>(0)</th>
<th>1</th>
<th>I=00</th>
<th>1</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td></td>
<td></td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>P</th>
<th>W</th>
<th>o1</th>
<th>op2</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>STRH (register) - Post-indexed variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>LDRD (register) - Post-indexed variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>STRD (register) - Post-indexed variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>LDRH (register) - Post-indexed variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>LDRSB (register) - Post-indexed variant</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>LDRSH (register) - Post-indexed variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>STRHT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>01</td>
<td>LDRHT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>LDRSBT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>LDRSHT</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>01</td>
<td>STRH (register) - Pre-indexed variant</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>10</td>
<td>LDRD (register) - Pre-indexed variant</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>11</td>
<td>STRD (register) - Pre-indexed variant</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1</td>
<td>01</td>
<td>LDRH (register) - Pre-indexed variant</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1</td>
<td>10</td>
<td>LDRSB (register) - Pre-indexed variant</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1</td>
<td>11</td>
<td>LDRSH (register) - Pre-indexed variant</td>
</tr>
</tbody>
</table>

## Load/Store Dual, Half, Signed Byte (immediate, literal)

This section describes the encoding of the Load/Store Dual, Half, Signed Byte (immediate, literal) instruction class. The encodings in this section are decoded from *Extra load/store on page F4-7376.*
## A32 Instruction Set Encoding

### F4.1 A32 instruction set encoding

<table>
<thead>
<tr>
<th>cond</th>
<th>P</th>
<th>U</th>
<th>W</th>
<th>o1</th>
<th>Rn</th>
<th>Rt</th>
<th>imm4H</th>
<th>1</th>
<th>1</th>
<th>imm4L</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>P:W</th>
<th>o1</th>
<th>Rn</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0</td>
<td>1111</td>
<td>10</td>
</tr>
<tr>
<td>!= 1</td>
<td>1</td>
<td>1111</td>
<td>01</td>
</tr>
<tr>
<td>!= 1</td>
<td>1</td>
<td>1111</td>
<td>10</td>
</tr>
<tr>
<td>!= 1</td>
<td>1</td>
<td>1111</td>
<td>11</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1111</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>-</td>
<td>01</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>-</td>
<td>11</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>!= 1</td>
<td>01</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>!= 1</td>
<td>10</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>!= 1</td>
<td>11</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>!= 1</td>
<td>1111</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>-</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>-</td>
<td>11</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>-</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>-</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>!= 1</td>
<td>1111</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>-</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>-</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>!= 1</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>!= 1</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>!= 1</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>!= 1</td>
<td>1111</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>-</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>-</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>!= 1</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>!= 1</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>!= 1</td>
<td>11</td>
</tr>
</tbody>
</table>

### Instruction page

- **LDRD** (literal)
- **LDRH** (literal)
- **LDRSB** (literal)
- **LDRSH** (literal)
- **LDRD** (immediate) - Post-indexed variant
- **STRH** (immediate) - Post-indexed variant
- **STRD** (immediate) - Post-indexed variant
- **LDRH** (immediate) - Post-indexed variant
- **LDRSB** (immediate) - Post-indexed variant
- **LDRSH** (immediate) - Post-indexed variant
- **LDRD** (immediate) - Offset variant
- **STRH** (immediate) - Offset variant
- **STRD** (immediate) - Offset variant
- **LDRH** (immediate) - Offset variant
- **LDRSB** (immediate) - Offset variant
- **LDRSH** (immediate) - Offset variant
- **LDRD** (immediate) - Pre-indexed variant
- **STRH** (immediate) - Pre-indexed variant
- **STRD** (immediate) - Pre-indexed variant
- **LDRH** (immediate) - Pre-indexed variant
- **LDRSB** (immediate) - Pre-indexed variant
- **LDRSH** (immediate) - Pre-indexed variant
F4.1.3 Synchronization primitives and Load-Acquire/Store-Release

This section describes the encoding of the Synchronization primitives and Load-Acquire/Store-Release group. The encodings in this section are decoded from Data-processing and miscellaneous instructions on page F4-7374.

Table F4-4 Encoding table for the Synchronization primitives and Load-Acquire/Store-Release group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>Load/Store Exclusive and Load-Acquire/Store-Release on page F4-7379</td>
</tr>
</tbody>
</table>

Load/Store Exclusive and Load-Acquire/Store-Release

This section describes the encoding of the Load/Store Exclusive and Load-Acquire/Store-Release instruction class. The encodings in this section are decoded from Synchronization primitives and Load-Acquire/Store-Release.

Table F4-4 Encoding table for the Synchronization primitives and Load-Acquire/Store-Release group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>size L ex ord</td>
<td></td>
</tr>
<tr>
<td>00 0 0 0</td>
<td>STL</td>
</tr>
<tr>
<td>00 0 0 1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00 0 1 0</td>
<td>STLEX</td>
</tr>
<tr>
<td>00 0 1 1</td>
<td>STREX</td>
</tr>
<tr>
<td>00 1 0 0</td>
<td>LDA</td>
</tr>
<tr>
<td>00 1 0 1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>00 1 1 0</td>
<td>LDAEX</td>
</tr>
<tr>
<td>00 1 1 1</td>
<td>LDREX</td>
</tr>
<tr>
<td>01 0 0 -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01 0 1 0</td>
<td>STLEXD</td>
</tr>
<tr>
<td>01 0 1 1</td>
<td>STREXD</td>
</tr>
<tr>
<td>01 1 0 -</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01 1 1 0</td>
<td>LDAEXD</td>
</tr>
</tbody>
</table>
### F4.1.4 Miscellaneous

This section describes the encoding of the Miscellaneous group. The encodings in this section are decoded from *Data-processing and miscellaneous instructions* on page F4-7374.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>size  L  ex  ord</td>
<td></td>
</tr>
<tr>
<td>01  1  1  1</td>
<td>LDREXD</td>
</tr>
<tr>
<td>10  0  0  0</td>
<td>STLBD</td>
</tr>
<tr>
<td>10  0  0  1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10  0  1  0</td>
<td>STLEXBD</td>
</tr>
<tr>
<td>10  0  1  1</td>
<td>STREXB</td>
</tr>
<tr>
<td>10  1  0  0</td>
<td>LDAB</td>
</tr>
<tr>
<td>10  1  0  1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10  1  1  0</td>
<td>LDAEXB</td>
</tr>
<tr>
<td>10  1  1  1</td>
<td>LDREXB</td>
</tr>
<tr>
<td>11  0  0  0</td>
<td>STLH</td>
</tr>
<tr>
<td>11  0  0  1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11  0  1  0</td>
<td>STLEXH</td>
</tr>
<tr>
<td>11  0  1  1</td>
<td>STREXH</td>
</tr>
<tr>
<td>11  1  0  0</td>
<td>LDAH</td>
</tr>
<tr>
<td>11  1  0  1</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11  1  1  0</td>
<td>LDAEXH</td>
</tr>
<tr>
<td>11  1  1  1</td>
<td>LDREXH</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table F4-5 Encoding table for the Miscellaneous group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode fields</td>
</tr>
<tr>
<td>op0  op1</td>
</tr>
<tr>
<td>00  001</td>
</tr>
<tr>
<td>00  010</td>
</tr>
<tr>
<td>00  011</td>
</tr>
<tr>
<td>00  110</td>
</tr>
<tr>
<td>01  001</td>
</tr>
</tbody>
</table>
Exception Generation

This section describes the encoding of the Exception Generation instruction class. The encodings in this section are decoded from Miscellaneous on page F4-7380.

<table>
<thead>
<tr>
<th>_decode_fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 010</td>
<td>BXJ</td>
</tr>
<tr>
<td>01 011</td>
<td>BLX (register)</td>
</tr>
<tr>
<td>01 110</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10 001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10 010</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10 011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>10 110</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11 001</td>
<td>CLZ</td>
</tr>
<tr>
<td>11 010</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11 011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11 110</td>
<td>ERET</td>
</tr>
<tr>
<td>- 111</td>
<td>Exception Generation</td>
</tr>
<tr>
<td>- 000</td>
<td>Move special register (register)</td>
</tr>
<tr>
<td>- 100</td>
<td>Cyclic Redundancy Check on page F4-7382</td>
</tr>
<tr>
<td>- 101</td>
<td>Integer Saturating Arithmetic on page F4-7382</td>
</tr>
</tbody>
</table>

Move special register (register)

This section describes the encoding of the Move special register (register) instruction class. The encodings in this section are decoded from Miscellaneous on page F4-7380.

<table>
<thead>
<tr>
<th>_decode_fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>HLT</td>
</tr>
<tr>
<td>01</td>
<td>BKPT</td>
</tr>
<tr>
<td>10</td>
<td>HVC</td>
</tr>
<tr>
<td>11</td>
<td>SMC</td>
</tr>
</tbody>
</table>
A32 Instruction Set Encoding
F4.1 A32 instruction set encoding

This section describes the encoding of the Cyclic Redundancy Check instruction class. The encodings in this section are decoded from Miscellaneous on page F4-7380.

### Cyclic Redundancy Check

This section describes the encoding of the Cyclic Redundancy Check instruction class. The encodings in this section are decoded from Miscellaneous on page F4-7380.

| [31] | [28] | [26] | [25] | [24] | [23] | [22] | [21] | [20] | [19] | [16] | [15] | [12] | [11] | [10] | [9] | [8] | [7] | [6] | [5] | [4] | [3] | [0] |
|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| !=1111 | 0 | 0 | 0 | 1 | 0 | opc | 0 | mask | Rd | (0) (0) | B | m | 0 | 0 | 0 | 0 | Rn |
| cond |

#### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>0</td>
</tr>
<tr>
<td>x0</td>
<td>1</td>
</tr>
<tr>
<td>x1</td>
<td>0</td>
</tr>
<tr>
<td>x1</td>
<td>1</td>
</tr>
</tbody>
</table>

| [31] | [28] | [26] | [25] | [24] | [23] | [22] | [21] | [20] | [19] | [16] | [15] | [12] | [11] | [10] | [9] | [8] | [7] | [6] | [5] | [4] | [3] | [0] |
|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| !=1111 | 0 | 0 | 0 | 1 | 0 | sz | 0 | Rn | Rd | (0) (0) | C | (0) | 0 | 1 | 0 | 0 | Rm |
| cond |

#### Decode fields

<table>
<thead>
<tr>
<th>sz</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
</tr>
</tbody>
</table>

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings on page K1-11574.

### Integer Saturating Arithmetic

This section describes the encoding of the Integer Saturating Arithmetic instruction class. The encodings in this section are decoded from Miscellaneous on page F4-7380.
F4.1.5 Data-processing register (immediate shift)

This section describes the encoding of the Data-processing register (immediate shift) group. The encodings in this section are decoded from Data-processing and miscellaneous instructions on page F4-7374.

```
|31|28|27|26|25|24|23|22|21|20|19|16|15|12|11|10| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|≠1111|0|0|0|1|0|opc|0|Rn|Rd|0|0|0|0|0|0|1|0|1|Rm|
```

This decode also imposes the constraint:

- $\text{op0:op1 } \neq 100$.

### Table F4-6 Encoding table for the Data-processing register (immediate shift) group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>0x</td>
<td>Integer Data Processing (three register, immediate shift) on page F4-7383</td>
</tr>
<tr>
<td>10 1</td>
<td>Integer Test and Compare (two register, immediate shift) on page F4-7385</td>
</tr>
<tr>
<td>11 -</td>
<td>Logical Arithmetic (three register, immediate shift) on page F4-7385</td>
</tr>
</tbody>
</table>

### Integer Data Processing (three register, immediate shift)

This section describes the encoding of the Integer Data Processing (three register, immediate shift) instruction class. The encodings in this section are decoded from Data-processing register (immediate shift).
### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>S</th>
<th>Rn</th>
<th>imm5:stype</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>-</td>
<td>-</td>
<td>!= 0000011</td>
<td>AND, ANDS (register) - ANDS, shift or rotate by value variant</td>
</tr>
<tr>
<td>000</td>
<td>-</td>
<td>-</td>
<td>0000011</td>
<td>AND, ANDS (register) - ANDS, rotate right with extend variant</td>
</tr>
<tr>
<td>001</td>
<td>-</td>
<td>-</td>
<td>!= 0000011</td>
<td>EOR, EORS (register) - EORS, shift or rotate by value variant</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>1101</td>
<td>!= 0000011</td>
<td>SUB, SUBS (register) - SUB, shift or rotate by value variant</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>1101</td>
<td>0000011</td>
<td>SUB, SUBS (register) - SUB, rotate right with extend variant</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>1101</td>
<td>!= 0000011</td>
<td>SUB, SUBS (SP minus register) - SUB, shift or rotate by value variant</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>1101</td>
<td>0000011</td>
<td>SUB, SUBS (SP minus register) - SUB, rotate right with extend variant</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>1101</td>
<td>!= 0000011</td>
<td>SUB, SUBS (register) - SUBS, shift or rotate by value variant</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>1101</td>
<td>0000011</td>
<td>SUB, SUBS (register) - SUBS, rotate right with extend variant</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>1101</td>
<td>!= 0000011</td>
<td>SUB, SUBS (SP minus register) - SUBS, shift or rotate by value variant</td>
</tr>
<tr>
<td>011</td>
<td>-</td>
<td>-</td>
<td>!= 0000011</td>
<td>RSB, RSBS (register) - RSBS, shift or rotate by value variant</td>
</tr>
<tr>
<td>011</td>
<td>-</td>
<td>-</td>
<td>0000011</td>
<td>RSB, RSBS (register) - RSBS, rotate right with extend variant</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1101</td>
<td>!= 0000011</td>
<td>ADD, ADDS (register) - ADD, shift or rotate by value variant</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1101</td>
<td>0000011</td>
<td>ADD, ADDS (register) - ADD, rotate right with extend variant</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1101</td>
<td>!= 0000011</td>
<td>ADD, ADDS (SP plus register) - ADD, shift or rotate by value variant</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1101</td>
<td>0000011</td>
<td>ADD, ADDS (SP plus register) - ADD, rotate right with extend variant</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1101</td>
<td>!= 0000011</td>
<td>ADD, ADDS (register) - ADDS, shift or rotate by value variant</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1101</td>
<td>0000011</td>
<td>ADD, ADDS (register) - ADDS, rotate right with extend variant</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1101</td>
<td>!= 0000011</td>
<td>ADD, ADDS (SP plus register) - ADDS, shift or rotate by value variant</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1101</td>
<td>0000011</td>
<td>ADD, ADDS (SP plus register) - ADDS, rotate right with extend variant</td>
</tr>
<tr>
<td>101</td>
<td>-</td>
<td>-</td>
<td>!= 0000011</td>
<td>ADC, ADCS (register) - ADCS, shift or rotate by value variant</td>
</tr>
<tr>
<td>101</td>
<td>-</td>
<td>-</td>
<td>0000011</td>
<td>ADC, ADCS (register) - ADCS, rotate right with extend variant</td>
</tr>
<tr>
<td>110</td>
<td>-</td>
<td>-</td>
<td>!= 0000011</td>
<td>SBC, SBCS (register) - SBCS, shift or rotate by value variant</td>
</tr>
<tr>
<td>110</td>
<td>-</td>
<td>-</td>
<td>0000011</td>
<td>SBC, SBCS (register) - SBCS, rotate right with extend variant</td>
</tr>
<tr>
<td>111</td>
<td>-</td>
<td>-</td>
<td>!= 0000011</td>
<td>RSC, RSCS (register) - RSCS, shift or rotate by value variant</td>
</tr>
<tr>
<td>111</td>
<td>-</td>
<td>-</td>
<td>0000011</td>
<td>RSC, RSCS (register) - RSCS, rotate right with extend variant</td>
</tr>
</tbody>
</table>
### Integer Test and Compare (two register, immediate shift)

This section describes the encoding of the Integer Test and Compare (two register, immediate shift) instruction class. The encodings in this section are decoded from *Data-processing register (immediate shift)* on page F4-7383.

<table>
<thead>
<tr>
<th><strong>Decode fields</strong></th>
<th><strong>Instruction page</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>opc imm5:stype</strong></td>
<td></td>
</tr>
<tr>
<td>00 ! = 0000011</td>
<td>TST (register) - Shift or rotate by value variant</td>
</tr>
<tr>
<td>00 0000011</td>
<td>TST (register) - Rotate right with extend variant</td>
</tr>
<tr>
<td>01 ! = 0000011</td>
<td>TEQ (register) - Shift or rotate by value variant</td>
</tr>
<tr>
<td>01 0000011</td>
<td>TEQ (register) - Rotate right with extend variant</td>
</tr>
<tr>
<td>10 ! = 0000011</td>
<td>CMP (register) - Shift or rotate by value variant</td>
</tr>
<tr>
<td>10 0000011</td>
<td>CMP (register) - Rotate right with extend variant</td>
</tr>
<tr>
<td>11 ! = 0000011</td>
<td>CMN (register) - Shift or rotate by value variant</td>
</tr>
<tr>
<td>11 0000011</td>
<td>CMN (register) - Rotate right with extend variant</td>
</tr>
</tbody>
</table>

### Logical Arithmetic (three register, immediate shift)

This section describes the encoding of the Logical Arithmetic (three register, immediate shift) instruction class. The encodings in this section are decoded from *Data-processing register (immediate shift)* on page F4-7383.

<table>
<thead>
<tr>
<th><strong>Decode fields</strong></th>
<th><strong>Instruction page</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>opc imm5:stype</strong></td>
<td></td>
</tr>
<tr>
<td>00 ! = 0000011</td>
<td>ORR, ORRS (register) - ORRS, shift or rotate by value variant</td>
</tr>
<tr>
<td>00 0000011</td>
<td>ORR, ORRS (register) - ORRS, rotate right with extend variant</td>
</tr>
<tr>
<td>01 ! = 0000011</td>
<td>MOV, MOVS (register) - MOVS, shift or rotate by value variant</td>
</tr>
<tr>
<td>01 0000011</td>
<td>MOV, MOVS (register) - MOVS, rotate right with extend variant</td>
</tr>
<tr>
<td>10 ! = 0000011</td>
<td>BIC, BICS (register) - BICS, shift or rotate by value variant</td>
</tr>
<tr>
<td>10 0000011</td>
<td>BIC, BICS (register) - BICS, rotate right with extend variant</td>
</tr>
<tr>
<td>11 ! = 0000011</td>
<td>MVN, MVNS (register) - MVNS, shift or rotate by value variant</td>
</tr>
<tr>
<td>11 0000011</td>
<td>MVN, MVNS (register) - MVNS, rotate right with extend variant</td>
</tr>
</tbody>
</table>
F4.1.6 Data-processing register (register shift)

This section describes the encoding of the Data-processing register (register shift) group. The encodings in this section are decoded from Data-processing and miscellaneous instructions on page F4-7374.

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111</td>
<td>000</td>
<td>op0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This decode also imposes the constraint:

- \( op0:op1 \neq 100 \).

Table F4-7 Encoding table for the Data-processing register (register shift) group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>0x</td>
<td>Integer Data Processing (three register, register shift) on page F4-7386</td>
</tr>
<tr>
<td>10</td>
<td>Integer Test and Compare (two register, register shift) on page F4-7386</td>
</tr>
<tr>
<td>11</td>
<td>Logical Arithmetic (three register, register shift) on page F4-7387</td>
</tr>
</tbody>
</table>

Integer Data Processing (three register, register shift)

This section describes the encoding of the Integer Data Processing (three register, register shift) instruction class. The encodings in this section are decoded from Data-processing register (register shift).

| 31 | 28|27 | 25 |24 |23 |22 |21 |20 |19 |16 |15 |12|11 | 8 |7  |6  |5  |4  |3  |0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| l=1111 | 0 0 0 0 opc | S | Rn | Rd | Rs | 0 | stype | 1 | Rm |

Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AND, ANDS (register-shifted register)</td>
</tr>
<tr>
<td>001</td>
<td>EOR, EORS (register-shifted register)</td>
</tr>
<tr>
<td>010</td>
<td>SUB, SUBS (register-shifted register)</td>
</tr>
<tr>
<td>011</td>
<td>RSB, RSBS (register-shifted register)</td>
</tr>
<tr>
<td>100</td>
<td>ADD, ADDS (register-shifted register)</td>
</tr>
<tr>
<td>101</td>
<td>ADC, ADCS (register-shifted register)</td>
</tr>
<tr>
<td>110</td>
<td>SBC, SBCS (register-shifted register)</td>
</tr>
<tr>
<td>111</td>
<td>RSC, RSCS (register-shifted register)</td>
</tr>
</tbody>
</table>

Integer Test and Compare (two register, register shift)

This section describes the encoding of the Integer Test and Compare (two register, register shift) instruction class. The encodings in this section are decoded from Data-processing register (register shift).
Logical Arithmetic (three register, register shift)

This section describes the encoding of the Logical Arithmetic (three register, register shift) instruction class. The encodings in this section are decoded from *Data-processing register (register shift)* on page F4-7386.

![Instruction Set Encoding Diagram]

F4.1.7  Data-processing immediate

This section describes the encoding of the Data-processing immediate group. The encodings in this section are decoded from *Data-processing and miscellaneous instructions* on page F4-7374.

---

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>TST (register-shifted register)</td>
</tr>
<tr>
<td>01</td>
<td>TEQ (register-shifted register)</td>
</tr>
<tr>
<td>10</td>
<td>CMP (register-shifted register)</td>
</tr>
<tr>
<td>11</td>
<td>CMN (register-shifted register)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>ORR, ORRS (register-shifted register)</td>
</tr>
<tr>
<td>01</td>
<td>MOV, MOVS (register-shifted register)</td>
</tr>
<tr>
<td>10</td>
<td>BIC, BICS (register-shifted register)</td>
</tr>
<tr>
<td>11</td>
<td>MVN, MVNS (register-shifted register)</td>
</tr>
</tbody>
</table>
### Integer Data Processing (two register and immediate)

This section describes the encoding of the Integer Data Processing (two register and immediate) instruction class. The encodings in this section are decoded from the Data-processing immediate group on page F4-7387.

#### Table F4-8 Encoding table for the Data-processing immediate group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
</tr>
<tr>
<td>0x</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>x1</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
</tr>
</tbody>
</table>

#### Integer Data Processing (two register and immediate)

This section describes the encoding of the Integer Data Processing (two register and immediate) instruction class. The encodings in this section are decoded from the Data-processing immediate on page F4-7387.

<table>
<thead>
<tr>
<th>[31 28 27 25 24 23 22 21 20 19 16 15 12 11 0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=1111</td>
</tr>
<tr>
<td>cond</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>S</th>
<th>Rn</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>-</td>
<td>-</td>
<td>AND, ANDS (immediate)</td>
</tr>
<tr>
<td>001</td>
<td>-</td>
<td>-</td>
<td>EOR, EORS (immediate)</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>!=11x1</td>
<td>SUB, SUBS (immediate) - SUB variant</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>1101</td>
<td>SUB, SUBS (SP minus immediate) - SUB variant</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>1111</td>
<td>ADR - A2 on page F5-7471</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>!=1101</td>
<td>SUB, SUBS (immediate) - SUBS variant</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>1101</td>
<td>SUB, SUBS (SP minus immediate) - SUBS variant</td>
</tr>
<tr>
<td>011</td>
<td>-</td>
<td>-</td>
<td>RSB, RSBS (immediate)</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>!=11x1</td>
<td>ADD, ADDS (immediate) - ADD variant</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1101</td>
<td>ADD, ADDS (SP plus immediate) - ADD variant</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1111</td>
<td>ADR - A1 on page F5-7471</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>!=1101</td>
<td>ADD, ADDS (immediate) - ADDS variant</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1101</td>
<td>ADD, ADDS (SP plus immediate) - ADDS variant</td>
</tr>
</tbody>
</table>
**Move Halfword (immediate)**

This section describes the encoding of the Move Halfword (immediate) instruction class. The encodings in this section are decoded from *Data-processing immediate* on page F4-7387.

```
<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=111</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>H</td>
<td>0</td>
<td>0</td>
<td>imm4</td>
<td>Rd</td>
<td>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Move Special Register and Hints (immediate)**

This section describes the encoding of the Move Special Register and Hints (immediate) instruction class. The encodings in this section are decoded from *Data-processing immediate* on page F4-7387.

```
<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=111</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>0</td>
<td>imm4</td>
<td>[1][1][1][1]</td>
<td>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Decode fields**

<table>
<thead>
<tr>
<th>opc</th>
<th>S</th>
<th>Rn</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>-</td>
<td>-</td>
<td>ADC, ADCS (immediate)</td>
</tr>
<tr>
<td>110</td>
<td>-</td>
<td>-</td>
<td>SBC, SBCS (immediate)</td>
</tr>
<tr>
<td>111</td>
<td>-</td>
<td>-</td>
<td>RSC, RSCS (immediate)</td>
</tr>
</tbody>
</table>

**Decode fields**

<table>
<thead>
<tr>
<th>H</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MOV, MOVS (immediate)</td>
</tr>
<tr>
<td>1</td>
<td>MOVT</td>
</tr>
</tbody>
</table>

**Decode fields**

<table>
<thead>
<tr>
<th>R:imm4</th>
<th>imm12</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=0000</td>
<td>-</td>
<td>MSR (immediate)</td>
<td>-</td>
</tr>
<tr>
<td>00000</td>
<td>xxxx00000000</td>
<td>NOP</td>
<td>-</td>
</tr>
<tr>
<td>00000</td>
<td>xxxx00000001</td>
<td>YIELD</td>
<td>-</td>
</tr>
<tr>
<td>00000</td>
<td>xxxx00000010</td>
<td>WFE</td>
<td>-</td>
</tr>
<tr>
<td>00000</td>
<td>xxxx00000111</td>
<td>WFI</td>
<td>-</td>
</tr>
<tr>
<td>00000</td>
<td>xxxx00001000</td>
<td>SEV</td>
<td>-</td>
</tr>
<tr>
<td>00000</td>
<td>xxxx00010101</td>
<td>SEVL</td>
<td>-</td>
</tr>
<tr>
<td>00000</td>
<td>xxxx000011x</td>
<td>Reserved hint, behaves as NOP.</td>
<td>-</td>
</tr>
<tr>
<td>00000</td>
<td>xxxx0001xxx</td>
<td>Reserved hint, behaves as NOP.</td>
<td>-</td>
</tr>
</tbody>
</table>
Integer Test and Compare (one register and immediate)

This section describes the encoding of the Integer Test and Compare (one register and immediate) instruction class. The encodings in this section are decoded from Data-processing immediate on page F4-7387.

<table>
<thead>
<tr>
<th>R:imm4</th>
<th>imm12</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>xxx0100000</td>
<td>ESB</td>
<td>FEAT_RAS</td>
</tr>
<tr>
<td>00000</td>
<td>xxx0100100</td>
<td>Reserved hint, behaves as NOP.</td>
<td>-</td>
</tr>
<tr>
<td>00000</td>
<td>xxx0100101</td>
<td>Reserved hint, behaves as NOP.</td>
<td>-</td>
</tr>
<tr>
<td>00000</td>
<td>xxx0100110</td>
<td>Reserved hint, behaves as NOP.</td>
<td>-</td>
</tr>
<tr>
<td>00000</td>
<td>xxx01011xx</td>
<td>Reserved hint, behaves as NOP.</td>
<td>-</td>
</tr>
<tr>
<td>00000</td>
<td>xxx1110xxx</td>
<td>Reserved hint, behaves as NOP.</td>
<td>-</td>
</tr>
<tr>
<td>00000</td>
<td>xxx1111xxx</td>
<td>Reserved hint, behaves as NOP.</td>
<td>-</td>
</tr>
<tr>
<td>00000</td>
<td>xxx1111xxx</td>
<td>DBG</td>
<td>-</td>
</tr>
</tbody>
</table>

Logical Arithmetic (two register and immediate)

This section describes the encoding of the Logical Arithmetic (two register and immediate) instruction class. The encodings in this section are decoded from Data-processing immediate on page F4-7387.

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>TST (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>TEQ (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>CMP (immediate)</td>
</tr>
<tr>
<td>11</td>
<td>CMN (immediate)</td>
</tr>
</tbody>
</table>
F4.1.8 Load/Store Word, Unsigned Byte (immediate, literal)

This section describes the encoding of the Load/Store Word, Unsigned Byte (immediate, literal) instruction class. The encodings in this section are decoded from *A32 instruction set encoding* on page F4-7374.

```
<table>
<thead>
<tr>
<th>31</th>
<th>28 27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16 15</th>
<th>12 11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1=1111</td>
<td>0 0 1 1 opc S Rn Rd imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Decode fields

### Instruction page

<table>
<thead>
<tr>
<th>opc</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ORR, ORRS (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>MOV, MOVS (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>BIC, BICS (immediate)</td>
</tr>
<tr>
<td>11</td>
<td>MVN, MVNS (immediate)</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>31</th>
<th>28 27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16 15</th>
<th>12 11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1=1111</td>
<td>0 1 0 P U o2 W o1 Rn Rt imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Decode fields

### Instruction page

<table>
<thead>
<tr>
<th>P:W o2 o1 Rn</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 01 0 1 1111</td>
<td>LDR (literal)</td>
</tr>
<tr>
<td>!= 01 1 1 1111</td>
<td>LDRB (literal)</td>
</tr>
<tr>
<td>00 0 0 -</td>
<td>STR (immediate) - Post-indexed variant</td>
</tr>
<tr>
<td>00 0 1 != 1111</td>
<td>LDR (immediate) - Post-indexed variant</td>
</tr>
<tr>
<td>00 1 0 -</td>
<td>STRB (immediate) - Post-indexed variant</td>
</tr>
<tr>
<td>00 1 1 != 1111</td>
<td>LDRB (immediate) - Post-indexed variant</td>
</tr>
<tr>
<td>01 0 0 -</td>
<td>STRT</td>
</tr>
<tr>
<td>01 0 1 -</td>
<td>LDRT</td>
</tr>
<tr>
<td>01 1 0 -</td>
<td>STRBT</td>
</tr>
<tr>
<td>01 1 1 -</td>
<td>LDRBT</td>
</tr>
<tr>
<td>10 0 0 -</td>
<td>STR (immediate) - Offset variant</td>
</tr>
<tr>
<td>10 0 1 != 1111</td>
<td>LDR (immediate) - Offset variant</td>
</tr>
<tr>
<td>10 1 0 -</td>
<td>STRB (immediate) - Offset variant</td>
</tr>
<tr>
<td>10 1 1 != 1111</td>
<td>LDRB (immediate) - Offset variant</td>
</tr>
<tr>
<td>11 0 0 -</td>
<td>STR (immediate) - Pre-indexed variant</td>
</tr>
</tbody>
</table>
F4.1 A32 instruction set encoding

### F4.1.9 Load/Store Word, Unsigned Byte (register)

This section describes the encoding of the Load/Store Word, Unsigned Byte (register) instruction class. The encodings in this section are decoded from *A32 instruction set encoding* on page F4-7374.

```
<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111</td>
<td>0</td>
<td>1</td>
<td>P</td>
<td>U</td>
<td>o2</td>
<td>W</td>
<td>o1</td>
<td>Rn</td>
<td>Rt</td>
<td>imm5</td>
<td>stype</td>
<td>0</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Decode fields**

<table>
<thead>
<tr>
<th>P</th>
<th>o2</th>
<th>o1</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>!= 1111 LDR (immediate) - Pre-indexed variant</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>STRB (immediate) - Pre-indexed variant</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>!= 1111 LDRB (immediate) - Pre-indexed variant</td>
</tr>
</tbody>
</table>

### F4.1.10 Media instructions

This section describes the encoding of the Media instructions group. The encodings in this section are decoded from *A32 instruction set encoding* on page F4-7374.
### Table F4-9 Encoding table for the Media instructions group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>00000</td>
<td>-  Parallel Arithmetic on page F4-7394</td>
</tr>
<tr>
<td>01000 101</td>
<td>SEL</td>
</tr>
<tr>
<td>01000 001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01000 xx0</td>
<td>PKHBT, PKHTB</td>
</tr>
<tr>
<td>01001 x01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01001 xx0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0110x x01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>0110x xx0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011x0 001</td>
<td>Saturate 16-bit on page F4-7396</td>
</tr>
<tr>
<td>011x0 101</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011x1 x01</td>
<td>Reverse Bit/Byte on page F4-7396</td>
</tr>
<tr>
<td>011x1 xx0</td>
<td>Saturate 32-bit on page F4-7396</td>
</tr>
<tr>
<td>01xxx 111</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01xxx 011</td>
<td>Extend and Add on page F4-7397</td>
</tr>
<tr>
<td>10xxx</td>
<td>Signed multiply. Divide on page F4-7397</td>
</tr>
<tr>
<td>11000 000</td>
<td>Unsigned Sum of Absolute Differences on page F4-7398</td>
</tr>
<tr>
<td>11000 100</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11001 x00</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1101x x00</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>110xx 111</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1110x 111</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1110x x00</td>
<td>Bitfield Insert on page F4-7399</td>
</tr>
<tr>
<td>11111 111</td>
<td>Permanently UNDEFINED on page F4-7399</td>
</tr>
<tr>
<td>1111x x00</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11x0x x10</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
Parallel Arithmetic

This section describes the encoding of the Parallel Arithmetic instruction class. The encodings in this section are decoded from Media instructions on page F4-7392.

```
Table F4-9 Encoding table for the Media instructions group (continued)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>11x1x x10</td>
<td>Bitfield Extract on page F4-7400</td>
</tr>
<tr>
<td>11xxx 011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11xxx x01</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

```

```
Parallel Arithmetic

This section describes the encoding of the Parallel Arithmetic instruction class. The encodings in this section are decoded from Media instructions on page F4-7392.

```

```
<table>
<thead>
<tr>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1 B op2</td>
</tr>
<tr>
<td>Unallocated.</td>
</tr>
<tr>
<td>SADD16</td>
</tr>
<tr>
<td>SASX</td>
</tr>
<tr>
<td>SSAX</td>
</tr>
<tr>
<td>SSUB16</td>
</tr>
<tr>
<td>SADD8</td>
</tr>
<tr>
<td>Unallocated.</td>
</tr>
<tr>
<td>Unallocated.</td>
</tr>
<tr>
<td>SSUB8</td>
</tr>
<tr>
<td>QADD16</td>
</tr>
<tr>
<td>QASX</td>
</tr>
<tr>
<td>QSAX</td>
</tr>
<tr>
<td>QSUB16</td>
</tr>
<tr>
<td>QADD8</td>
</tr>
<tr>
<td>Unallocated.</td>
</tr>
<tr>
<td>QSUB8</td>
</tr>
<tr>
<td>SHADD16</td>
</tr>
<tr>
<td>SHASX</td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th>Table F4-9 Encoding table for the Media instructions group (continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>op1</th>
<th>B</th>
<th>op2</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>011</td>
<td>0</td>
<td>10</td>
<td>SHSAX</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>11</td>
<td>SHSUB16</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>00</td>
<td>SHADD8</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>11</td>
<td>SHSUB8</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td>Unallocated.</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>00</td>
<td>UADD16</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>01</td>
<td>UASX</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>10</td>
<td>USAX</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>11</td>
<td>USUB16</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>00</td>
<td>UADD8</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>11</td>
<td>USUB8</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>00</td>
<td>UQADD16</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>01</td>
<td>UQASX</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>10</td>
<td>UQSAX</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>11</td>
<td>UQSUB16</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>00</td>
<td>UQADD8</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>11</td>
<td>UQSUB8</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
<td>00</td>
<td>UHADD16</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
<td>01</td>
<td>UHASX</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
<td>10</td>
<td>UHSAX</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
<td>11</td>
<td>UHSUB16</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>00</td>
<td>UHADD8</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>01</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>10</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>11</td>
<td>UHSUB8</td>
</tr>
</tbody>
</table>
**Saturate 16-bit**

This section describes the encoding of the Saturate 16-bit instruction class. The encodings in this section are decoded from *Media instructions* on page F4-7392.

```
[31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4 3 0 ]
  !=1111 0 1 1 0 1 U 1 0 sat_imm Rd [1](1)(1)(1)[1] 0 0 1 1 Rn
cond
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SSAT16</td>
</tr>
<tr>
<td>1</td>
<td>USAT16</td>
</tr>
</tbody>
</table>

**Reverse Bit/Byte**

This section describes the encoding of the Reverse Bit/Byte instruction class. The encodings in this section are decoded from *Media instructions* on page F4-7392.

```
[31 28|27 26 25 24|23 22 21 20|19 18 17 16|15 12|11 10 9 8 7 6 5 4 3 0 ]
  !=1111 0 1 1 0 1 o1 1 1 1(1)(1)(1) Rd [1](1)(1)(1)[02] 0 1 1 Rm
cond
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1 o2</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>REV</td>
</tr>
<tr>
<td>0 1</td>
<td>REV16</td>
</tr>
<tr>
<td>1 0</td>
<td>RBIT</td>
</tr>
<tr>
<td>1 1</td>
<td>REVSH</td>
</tr>
</tbody>
</table>

**Saturate 32-bit**

This section describes the encoding of the Saturate 32-bit instruction class. The encodings in this section are decoded from *Media instructions* on page F4-7392.

```
[31 28|27 26 25 24|23 22 21 20|19 18 17 16|15 12|11 10 9 8 7 6 5 4 3 0 ]
  !=1111 0 1 1 0 1 U 1 1 sat_imm Rd [1](1)(1)(1)[1] imm5 sh 0 1 Rn
cond
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SSAT</td>
</tr>
<tr>
<td>1</td>
<td>USAT</td>
</tr>
</tbody>
</table>
Extend and Add

This section describes the encoding of the Extend and Add instruction class. The encodings in this section are decoded from Media instructions on page F4-7392.

```
| 31  | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=1111 | 0 | 1 | 1 | 0 | 1 | U | op | Rn | Rd | rotate(0) | (0) | 0 | 1 | 1 | 1 | Rm |
| cond |
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>U op Rn</td>
<td></td>
</tr>
<tr>
<td>0 00 != 1111</td>
<td>SXTAB16</td>
</tr>
<tr>
<td>0 00 1111</td>
<td>SXTB16</td>
</tr>
<tr>
<td>0 10 != 1111</td>
<td>SXTAB</td>
</tr>
<tr>
<td>0 10 1111</td>
<td>SXTB</td>
</tr>
<tr>
<td>0 11 != 1111</td>
<td>SXTAH</td>
</tr>
<tr>
<td>0 11 1111</td>
<td>SXTH</td>
</tr>
<tr>
<td>1 00 != 1111</td>
<td>UXTAB16</td>
</tr>
<tr>
<td>1 00 1111</td>
<td>UXTB16</td>
</tr>
<tr>
<td>1 10 != 1111</td>
<td>UXTAB</td>
</tr>
<tr>
<td>1 10 1111</td>
<td>UXTB</td>
</tr>
<tr>
<td>1 11 != 1111</td>
<td>UXTAH</td>
</tr>
<tr>
<td>1 11 1111</td>
<td>UXTH</td>
</tr>
</tbody>
</table>

Signed multiply, Divide

This section describes the encoding of the Signed multiply, Divide instruction class. The encodings in this section are decoded from Media instructions on page F4-7392.

```
| 31  | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=1111 | 0 | 1 | 1 | 0 | 1 | 0 | op1 | Rd | Ra | Rm | op2 | 1 | Rn |
| cond |
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1 Ra op2</td>
<td></td>
</tr>
<tr>
<td>000 != 1111</td>
<td>SMLAD, SMLADX - SMLAD variant</td>
</tr>
<tr>
<td>000 != 1111</td>
<td>SMLAD, SMLADX - SMLADX variant</td>
</tr>
<tr>
<td>000 != 1111</td>
<td>SMLSD, SMLSDX - SMLSD variant</td>
</tr>
<tr>
<td>000 != 1111</td>
<td>SMLSD, SMLSDX - SMLSDX variant</td>
</tr>
<tr>
<td>000 - 1xx</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
## Untitled

<table>
<thead>
<tr>
<th>op1</th>
<th>Ra</th>
<th>op2</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>1111</td>
<td>000</td>
<td>SMUAD, SMUADX - SMUAD variant</td>
</tr>
<tr>
<td>000</td>
<td>1111</td>
<td>001</td>
<td>SMUAD, SMUADX - SMUADX variant</td>
</tr>
<tr>
<td>000</td>
<td>1111</td>
<td>010</td>
<td>SMUSD, SMUSDX - SMUSD variant</td>
</tr>
<tr>
<td>000</td>
<td>1111</td>
<td>011</td>
<td>SMUSD, SMUSDX - SMUSDX variant</td>
</tr>
<tr>
<td>001</td>
<td>-</td>
<td>000</td>
<td>SDIV</td>
</tr>
<tr>
<td>001</td>
<td>-</td>
<td>!= 000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>010</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>011</td>
<td>-</td>
<td>000</td>
<td>UDIV</td>
</tr>
<tr>
<td>011</td>
<td>-</td>
<td>!= 000</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>000</td>
<td>SMLALD, SMLALDX - SMLALD variant</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>001</td>
<td>SMLALD, SMLALDX - SMLALDX variant</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>010</td>
<td>SMLSLD, SMLSLDX - SMLSLD variant</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>011</td>
<td>SMLSLD, SMLSLDX - SMLSLDX variant</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>1xx</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>101</td>
<td>!= 1111</td>
<td>000</td>
<td>SMMLA, SMMLAR - SMMLA variant</td>
</tr>
<tr>
<td>101</td>
<td>!= 1111</td>
<td>001</td>
<td>SMMLA, SMMLAR - SMMLAR variant</td>
</tr>
<tr>
<td>101</td>
<td>-</td>
<td>01x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>101</td>
<td>-</td>
<td>10x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>101</td>
<td>-</td>
<td>110</td>
<td>SMMLS, SMMLSR - SMMLS variant</td>
</tr>
<tr>
<td>101</td>
<td>-</td>
<td>111</td>
<td>SMMLS, SMMLSR - SMMLSR variant</td>
</tr>
<tr>
<td>101</td>
<td>1111</td>
<td>000</td>
<td>SMMUL, SMMULR - SMMUL variant</td>
</tr>
<tr>
<td>101</td>
<td>1111</td>
<td>001</td>
<td>SMMUL, SMMULR - SMMULR variant</td>
</tr>
<tr>
<td>11x</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

### Unsigned Sum of Absolute Differences

This section describes the encoding of the Unsigned Sum of Absolute Differences instruction class. The encodings in this section are decoded from Media instructions on page F4-7392.
Bitfield Insert

This section describes the encoding of the Bitfield Insert instruction class. The encodings in this section are decoded from Media instructions on page F4-7392.

Permanently UNDEFINED

This section describes the encoding of the Permanently UNDEFINED instruction class. The encodings in this section are decoded from Media instructions on page F4-7392.
Bitfield Extract

This section describes the encoding of the Bitfield Extract instruction class. The encodings in this section are decoded from Media instructions on page F4-7392.

```
| 0  | 1  | 1  | 1  | U  | 1  | widthm1 | Rd  | lsb | 1  | 0  | Rn |
-    |    |    |    |    |    |         |     |     |    |    |    |
cond
```

Decode fields | Instruction page
--- | ---
U | SBFX
1 | UBFX

F4.1.11  Branch, branch with link, and block data transfer

This section describes the encoding of the Branch, branch with link, and block data transfer group. The encodings in this section are decoded from A32 instruction set encoding on page F4-7374.

```
| 0  | 28| 27| 26| 25| 24| 16| 15| 12| 11| 7 | 6 | 5 | 4 | 3 | 0 |
-    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
cond op0
```

Table F4-10 Encoding table for the Branch, branch with link, and block data transfer group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond op0</td>
<td></td>
</tr>
<tr>
<td>1111 0</td>
<td>Exception Save/Restore on page F4-7400</td>
</tr>
<tr>
<td>1111 0</td>
<td>Load/Store Multiple on page F4-7401</td>
</tr>
<tr>
<td>- 1</td>
<td>Branch (immediate) on page F4-7402</td>
</tr>
</tbody>
</table>

Exception Save/Restore

This section describes the encoding of the Exception Save/Restore instruction class. The encodings in this section are decoded from Branch, branch with link, and block data transfer.
Load/Store Multiple

This section describes the encoding of the Load/Store Multiple instruction class. The encodings in this section are decoded from Branch, branch with link, and block data transfer on page F4-7400.

<table>
<thead>
<tr>
<th>31 28 27 26 25 24 23 22 21 20 19 16 15</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111 1 0 0 P U op W L Rn register_list</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Decode fields**

<table>
<thead>
<tr>
<th>P U op L register_list</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 -</td>
</tr>
<tr>
<td>0 0 0 1 -</td>
</tr>
<tr>
<td>0 1 0 0 -</td>
</tr>
<tr>
<td>0 1 0 1 -</td>
</tr>
<tr>
<td>- - 1 0 -</td>
</tr>
<tr>
<td>1 0 0 0 -</td>
</tr>
<tr>
<td>1 0 0 1 -</td>
</tr>
<tr>
<td>- - 1 1 0</td>
</tr>
</tbody>
</table>

**Instruction page**

- STMDA, STMED
- LDMDA, LDMFA
- STM, STMIA, STMEA
- LDM, LDMIA, LDMFD
- STM (User registers)
- STMDB, STMFD
- LDMDB, LDMEA
- LDM (User registers)
Branch (immediate)

This section describes the encoding of the Branch (immediate) instruction class. The encodings in this section are decoded from Branch, branch with link, and block data transfer on page F4-7400.

<table>
<thead>
<tr>
<th>cond</th>
<th>1 0 1</th>
<th>H</th>
<th>imm24</th>
</tr>
</thead>
</table>

F4.1.12 System register access, Advanced SIMD, floating-point, and Supervisor call

This section describes the encoding of the System register access, Advanced SIMD, floating-point, and Supervisor call group. The encodings in this section are decoded from A32 instruction set encoding on page F4-7374.

<table>
<thead>
<tr>
<th>cond</th>
<th>11</th>
<th>op0</th>
<th>12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>3 0</th>
</tr>
</thead>
</table>

Table F4-11 Encoding table for the System register access, Advanced SIMD, floating-point, and Supervisor call group

<table>
<thead>
<tr>
<th>cond</th>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0x</td>
<td>0x</td>
<td>0x</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>0x</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0x</td>
<td>0x</td>
<td>0x</td>
<td>Supervisor call on page F4-7403</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1x</td>
<td>Unconditional Advanced SIMD and floating-point instructions on page F4-7403</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>0x</td>
<td>1x</td>
<td>Advanced SIMD and System register load/store and 64-bit move on page F4-7410</td>
<td></td>
</tr>
</tbody>
</table>
F4.13 Supervisor call

This section describes the encoding of the Supervisor call group. The encodings in this section are decoded from System register access, Advanced SIMD, floating-point, and Supervisor call on page F4-7402.

![Supervisor call encoding diagram](image)

Table F4-12 Encoding table for the Supervisor call group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1111</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>!= 1111</td>
<td>SVC</td>
</tr>
</tbody>
</table>

F4.14 Unconditional Advanced SIMD and floating-point instructions

This section describes the encoding of the Unconditional Advanced SIMD and floating-point instructions group. The encodings in this section are decoded from System register access, Advanced SIMD, floating-point, and Supervisor call on page F4-7402.

![Unconditional Advanced SIMD and floating-point instructions encoding diagram](image)
This decode also imposes the constraint:

- \( \text{op0}<2:1> \neq 11 \).

### Table F4-13 Encoding table for the Unconditional Advanced SIMD and floating-point instructions group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xx</td>
<td>Advanced SIMD three registers of the same length extension on page F4-7404</td>
</tr>
<tr>
<td>100</td>
<td>Floating-point conditional select on page F4-7406</td>
</tr>
<tr>
<td>101 00xxxx</td>
<td>Floating-point minNum/maxNum on page F4-7406</td>
</tr>
<tr>
<td>101 110000</td>
<td>Floating-point extraction and insertion on page F4-7407</td>
</tr>
<tr>
<td>101 111xxx</td>
<td>Floating-point directed convert to integer on page F4-7407</td>
</tr>
<tr>
<td>10x</td>
<td>Advanced SIMD and floating-point multiply with accumulate on page F4-7408</td>
</tr>
<tr>
<td>10x</td>
<td>Advanced SIMD and floating-point dot product on page F4-7409</td>
</tr>
</tbody>
</table>

### Advanced SIMD three registers of the same length extension

This section describes the encoding of the Advanced SIMD three registers of the same length extension instruction class. The encodings in this section are decoded from *Unconditional Advanced SIMD and floating-point instructions* on page F4-7403.

```
<table>
<thead>
<tr>
<th>[31 30 29 28 27 26 25 24 23 22 21 20 19]</th>
<th>16 15 12 11 10  9  8  7  6  5  4  3  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 op1</td>
<td>D</td>
</tr>
<tr>
<td>op4</td>
<td>op3</td>
</tr>
</tbody>
</table>
```

### Decode fields

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
<th>Q</th>
<th>U</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>0x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>VCADD - 64-bit SIMD vector variant</td>
<td>FEAT_FCMA</td>
</tr>
<tr>
<td>x1</td>
<td>0x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>x1</td>
<td>0x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>VCADD - 128-bit SIMD vector variant</td>
<td>FEAT_FCMA</td>
</tr>
<tr>
<td>x1</td>
<td>0x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0x</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0x</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>VMMLA</td>
<td>FEAT_AA32BF16</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VDOT (vector) - 64-bit SIMD vector variant</td>
<td>FEAT_AA32BF16</td>
</tr>
<tr>
<td>op1</td>
<td>op2</td>
<td>op3</td>
<td>op4</td>
<td>Q</td>
<td>U</td>
<td>Instruction page</td>
<td>Feature</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>----</td>
<td>----</td>
<td>------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VDOT (vector) - 128-bit SIMD vector variant</td>
<td>FEAT_AA32BF16</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>VFMA (vector)</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>VSMMLA</td>
<td>FEAT_AA32I8MM</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>VUMMLA</td>
<td>FEAT_AA32I8MM</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VSDOT (vector) - 64-bit SIMD vector variant</td>
<td>FEAT.DotProd</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>VUDOT (vector) - 64-bit SIMD vector variant</td>
<td>FEAT.DotProd</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VSDOT (vector) - 128-bit SIMD vector variant</td>
<td>FEAT.DotProd</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>VUDOT (vector) - 128-bit SIMD vector variant</td>
<td>FEAT.DotProd</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>VFMAB, VFMAT (BFloat16, vector)</td>
<td>FEAT_AA32BF16</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>VUSDOT (vector) - 64-bit SIMD vector variant</td>
<td>FEAT_AA32I8MM</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>VUSDOT (vector) - 64-bit SIMD vector variant</td>
<td>FEAT_AA32I8MM</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VUSDOT (vector) - 128-bit SIMD vector variant</td>
<td>FEAT_AA32I8MM</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>VUSDOT (vector) - 128-bit SIMD vector variant</td>
<td>FEAT_AA32I8MM</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VUSDOT (vector) - 128-bit SIMD vector variant</td>
<td>FEAT_AA32I8MM</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1x</td>
<td>1x</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>VCMLA</td>
<td>FEAT_FCMA</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>
### Floating-point conditional select

This section describes the encoding of the Floating-point conditional select instruction class. The encodings in this section are decoded from *Unconditional Advanced SIMD and floating-point instructions on page F4-7403*.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1 op2 op3 op4 Q U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 11 1 1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11 11 0 1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11 11 1 0</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11 11 1 1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

### Floating-point minNum/maxNum

This section describes the encoding of the Floating-point minNum/maxNum instruction class. The encodings in this section are decoded from *Unconditional Advanced SIMD and floating-point instructions on page F4-7403*.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>VSELEQ, VSELGE, VSELGT, VSELVS - Greater than, half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>01</td>
<td>VSELEQ, VSELGE, VSELGT, VSELVS - Greater than, single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>VSELEQ, VSELGE, VSELGT, VSELVS - Greater than, double-precision scalar variant</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>size op</td>
<td>VMAXNM - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>01 0</td>
<td>VMINNM - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>01 1</td>
<td>VMAXNM - Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>10 0</td>
<td>VMINNM - Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>10 1</td>
<td>VMAXNM - Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>11 0</td>
<td>VMINNM - Double-precision scalar variant</td>
<td>-</td>
</tr>
</tbody>
</table>
### Floating-point extraction and insertion

This section describes the encoding of the Floating-point extraction and insertion instruction class. The encodings in this section are decoded from Unconditional Advanced SIMD and floating-point instructions on page F4-7403.

![Floating-point extraction and insertion instruction encoding](image)

### Floating-point directed convert to integer

This section describes the encoding of the Floating-point directed convert to integer instruction class. The encodings in this section are decoded from Unconditional Advanced SIMD and floating-point instructions on page F4-7403.

![Floating-point directed convert to integer instruction encoding](image)
### Advanced SIMD and floating-point multiply with accumulate

This section describes the encoding of the Advanced SIMD and floating-point multiply with accumulate instruction class. The encodings in this section are decoded from Unconditional Advanced SIMD and floating-point instructions on page F4-7403.

![Instruction page Feature](image)

#### Advanced SIMD and floating-point multiply with accumulate

**Unallocated.**

![Instruction page Feature](image)

## Decode fields

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>Q</th>
<th>U</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>VCMLA (by element) - 128-bit SIMD vector of half-precision floating-point variant</td>
<td>FEAT_FCMA</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>-</td>
<td>1</td>
<td>VFMLA (by scalar)</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>-</td>
<td>1</td>
<td>VFMSL (by scalar)</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>-</td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>-</td>
<td>1</td>
<td>VFMAB, VFMAT (BFloat16, by scalar)</td>
<td>FEAT_AA32BF1</td>
</tr>
</tbody>
</table>

---

---
This section describes the encoding of the Advanced SIMD and floating-point dot product instruction class. The encodings in this section are decoded from *Unconditional Advanced SIMD and floating-point instructions* on page F4-7403.

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 16 15 12|11 10  9| 8| 7  6  5  4| 3  0]
  1 1 1 1 1 1 1 0 | D  op2 | Vn | Vd | 1 1 0 | N | Q | M | U | Vm
  op1                                        | op4
```

### Advanced SIMD and floating-point dot product

- **VCMLA (by element)** - 64-bit SIMD vector of single-precision floating-point variant
  - FEAT_FCMA
- **VCMLA (by element)** - 128-bit SIMD vector of single-precision floating-point variant
  - FEAT_FCMA

---

### Decode fields

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>Q</th>
<th>U</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>VCMLA (by element) - 64-bit SIMD vector of single-precision floating-point variant</td>
<td>FEAT_FCMA</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td></td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>VCMLA (by element) - 128-bit SIMD vector of single-precision floating-point variant</td>
<td>FEAT_FCMA</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>op4</th>
<th>Q</th>
<th>U</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VDOT (by element) - 64-bit SIMD vector variant</td>
<td>FEAT_AA32BF16</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>1</td>
<td></td>
<td>1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VDOT (by element) - 128-bit SIMD vector variant</td>
<td>FEAT_AA32BF16</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VSDOT (by element) - 64-bit SIMD vector variant</td>
<td>FEAT_DotProd</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td></td>
<td>1</td>
<td>VUDOT (by element) - 64-bit SIMD vector variant</td>
<td>FEAT_DotProd</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VSDOT (by element) - 128-bit SIMD vector variant</td>
<td>FEAT_DotProd</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>VUDOT (by element) - 128-bit SIMD vector variant</td>
<td>FEAT_DotProd</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>VUSDOT (by element) - 64-bit SIMD vector variant</td>
<td>FEAT_AA32I8MM</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td></td>
<td>1</td>
<td>VUSDOT (by element) - 64-bit SIMD vector variant</td>
<td>FEAT_AA32I8MM</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>VUSDOT (by element) - 128-bit SIMD vector variant</td>
<td>FEAT_AA32I8MM</td>
</tr>
</tbody>
</table>
F4.1.15 Advanced SIMD and System register load/store and 64-bit move

This section describes the encoding of the Advanced SIMD and System register load/store and 64-bit move group. The encodings in this section are decoded from System register access, Advanced SIMD, floating-point, and Supervisor call on page F4-7402.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the A32 Advanced SIMD and floating-point instructions and their encoding on page F4-7440.

| [31 28 27 25 24 | 21 20 | 12|11 10 9 8 | | 0 | |
| !=1111 | 110 | op0 | | 1 | op1 |

**Table F4-14 Encoding table for the Advanced SIMD and System register load/store and 64-bit move group**

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>00x0 0x</td>
<td>Advanced SIMD and floating-point 64-bit move</td>
</tr>
<tr>
<td>00x0 11</td>
<td>System register 64-bit move on page F4-7411</td>
</tr>
<tr>
<td>!= 00x0 0x</td>
<td>Advanced SIMD and floating-point load/store on page F4-7411</td>
</tr>
<tr>
<td>!= 00x0 11</td>
<td>System register load/store on page F4-7413</td>
</tr>
<tr>
<td>- 10</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

**Advanced SIMD and floating-point 64-bit move**

This section describes the encoding of the Advanced SIMD and floating-point 64-bit move instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move.
### System register 64-bit move

This section describes the encoding of the System register 64-bit move instruction class. The encodings in this section are decoded from *Advanced SIMD and System register load/store and 64-bit move on page F4-7410*.

![Instruction page](image)

#### Decode fields

<table>
<thead>
<tr>
<th>D</th>
<th>op</th>
<th>size</th>
<th>opc2</th>
<th>o3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0x</td>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>01</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1x</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
<td>00</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Advanced SIMD and floating-point load/store

This section describes the encoding of the Advanced SIMD and floating-point load/store instruction class. The encodings in this section are decoded from *Advanced SIMD and System register load/store and 64-bit move on page F4-7410*.

![Instruction page](image)

#### Decode fields

<table>
<thead>
<tr>
<th>D</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>P</th>
<th>U</th>
<th>W</th>
<th>L</th>
<th>Rn</th>
<th>size</th>
<th>imm8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0x</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>11 xxxxxxx0</td>
<td>VSTM, VSTMDB, VSTMIA - Increment After variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>11 xxxxxxx1</td>
<td>FSTMDBX, FSTMIAX - Increment After variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>11 xxxxxxx0</td>
<td>VLDM, VLDMDB, VLMIA - Increment After variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>11 xxxxxxx1</td>
<td>FLDM*X (FLDMDBX, FLDMIAX) - Increment After variant</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>01</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>!= 1111</td>
<td>01</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>!= 1111</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>!= 1111</td>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0x</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>11 xxxxxxx0</td>
<td>VSTM, VSTMDB, VSTMIA - Decrement Before variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>11 xxxxxxx1</td>
<td>FSTMDBX, FSTMIAX - Decrement Before variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>11 xxxxxxx0</td>
<td>VLDM, VLDMDB, VLMIA - Decrement Before variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>11 xxxxxxx1</td>
<td>FLDM*X (FLDMDBX, FLDMIAX) - Decrement Before variant</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>01</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1111</td>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

---

F4-7412 Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. Non-Confidential ARM DDI 0487I.a ID081822
System register load/store

This section describes the encoding of the System register load/store instruction class. The encodings in this section are decoded from *Advanced SIMD and System register load/store and 64-bit move on page F4-7410*.

The encoding for the System register load/store instruction class is:

| !|= 000 | 0 | - | - | !|= 010 | 0 | Unallocated. |
| !|= 000 | 0 | 1 | 1111 | 010 | 0 | LDC (literal) |
| !|= 000 | - | - | - | - | 1 | Unallocated. |
| !|= 000 | 1 | - | - | 010 | 0 | Unallocated. |
| 0x1 | 0 | 0 | - | 010 | 0 | STC - Post-indexed variant |
| 0x1 | 0 | 1 | !|= 1111 | 010 | 0 | LDC (immediate) - Post-indexed variant |
| 010 | 0 | 0 | - | 010 | 0 | STC - Unindexed variant |
| 010 | 0 | 1 | !|= 1111 | 010 | 0 | LDC (immediate) - Unindexed variant |
| 1x0 | 0 | 0 | - | 010 | 0 | STC - Offset variant |
| 1x0 | 0 | 1 | !|= 1111 | 010 | 0 | LDC (immediate) - Offset variant |
| 1x1 | 0 | 0 | - | 010 | 0 | STC - Pre-indexed variant |
| 1x1 | 0 | 1 | !|= 1111 | 010 | 0 | LDC (immediate) - Pre-indexed variant |

F4.1.16 Advanced SIMD and System register 32-bit move

This section describes the encoding of the Advanced SIMD and System register 32-bit move group. The encodings in this section are decoded from *System register access, Advanced SIMD, floating-point, and Supervisor call on page F4-7402*.

The encoding for the Advanced SIMD and System register 32-bit move group is:

| !|= 000 | 0 | - | - | !|= 010 | 0 | Unallocated. |
| !|= 000 | 0 | 1 | 1111 | 010 | 0 | LDC (literal) |
| !|= 000 | - | - | - | - | 1 | Unallocated. |
| !|= 000 | 1 | - | - | 010 | 0 | Unallocated. |
| 0x1 | 0 | 0 | - | 010 | 0 | STC - Post-indexed variant |
| 0x1 | 0 | 1 | !|= 1111 | 010 | 0 | LDC (immediate) - Post-indexed variant |
| 010 | 0 | 0 | - | 010 | 0 | STC - Unindexed variant |
| 010 | 0 | 1 | !|= 1111 | 010 | 0 | LDC (immediate) - Unindexed variant |
| 1x0 | 0 | 0 | - | 010 | 0 | STC - Offset variant |
| 1x0 | 0 | 1 | !|= 1111 | 010 | 0 | LDC (immediate) - Offset variant |
| 1x1 | 0 | 0 | - | 010 | 0 | STC - Pre-indexed variant |
| 1x1 | 0 | 1 | !|= 1111 | 010 | 0 | LDC (immediate) - Pre-indexed variant |

Table F4-15 Encoding table for the Advanced SIMD and System register 32-bit move group

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Decode group or instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td>VMOV (between general-purpose register and half-precision)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>000</td>
<td>010</td>
<td>VMOV (between general-purpose register and single-precision)</td>
<td>-</td>
</tr>
</tbody>
</table>
Floating-point move special register

This section describes the encoding of the Floating-point move special register instruction class. The encodings in this section are decoded from Advanced SIMD and System register 32-bit move on page F4-7413.

```
| 31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8|7 6 5 4|3 2 1 0 |
| L=1111 | 1 | 1 | 1 | 0 | 1 | 1 | L | reg | Rt | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
```

```
cond
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>VMSR</td>
</tr>
<tr>
<td>1</td>
<td>VMRS</td>
</tr>
</tbody>
</table>

Advanced SIMD 8/16/32-bit element move/duplicate

This section describes the encoding of the Advanced SIMD 8/16/32-bit element move/duplicate instruction class. The encodings in this section are decoded from Advanced SIMD and System register 32-bit move on page F4-7413.

```
| 31 28|27 26 25 24|23 21 20|19 16|15 12|11 10 9 8|7 6 5 4|3 2 1 0 |
| L=1111 | 1 | 1 | 1 | 0 | opc1 | L | Vn | Rt | 1 | 1 | 0 | 1 | N | opc2 | 1 | 0 | 0 | 0 | 0 |
```

```
cond
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc1 L opc2</td>
<td>VMOV (general-purpose register to scalar)</td>
</tr>
<tr>
<td>0xx 0 -</td>
<td></td>
</tr>
</tbody>
</table>
System register 32-bit move

This section describes the encoding of the System register 32-bit move instruction class. The encodings in this section are decoded from Advanced SIMD and System register 32-bit move on page F4-7413.

<table>
<thead>
<tr>
<th>[31]</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>8</th>
<th>7</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=111</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>opc1</td>
<td>L</td>
<td>CRn</td>
<td>Rt</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>opc2</td>
<td>1</td>
<td>CRm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unallocated.

F4.1.17 Floating-point data-processing

This section describes the encoding of the Floating-point data-processing group. The encodings in this section are decoded from System register access, Advanced SIMD, floating-point, and Supervisor call on page F4-7402.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the A32 Advanced SIMD and floating-point instructions and their encoding on page F4-7440.

<table>
<thead>
<tr>
<th>[31]</th>
<th>28</th>
<th>27</th>
<th>24</th>
<th>23</th>
<th>20</th>
<th>19</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=111</td>
<td>1110</td>
<td>op0</td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>op1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table F4-16 Encoding table for the Floating-point data-processing group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
</tr>
</tbody>
</table>

- 1x11: Floating-point data-processing (two registers) on page F4-7416
- 1x11: Floating-point move immediate on page F4-7418
- != 1x11: Floating-point data-processing (three registers) on page F4-7419
Floating-point data-processing (two registers)

This section describes the encoding of the Floating-point data-processing (two registers) instruction class. The encodings in this section are decoded from Floating-point data-processing on page F4-7415.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1 opc2 size</td>
<td>o3</td>
<td></td>
</tr>
<tr>
<td>- - 00 0</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0 000 01 0</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0 000 01 1</td>
<td>VABS - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 000 10 0</td>
<td>VMOV (register) - Single-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>0 000 10 1</td>
<td>VABS - Single-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>0 000 11 0</td>
<td>VMOV (register) - Double-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>0 000 11 1</td>
<td>VABS - Double-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>0 001 01 0</td>
<td>VNEG - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 001 01 1</td>
<td>VSQRT - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 001 10 0</td>
<td>VNEG - Single-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>0 001 10 1</td>
<td>VSQRT - Single-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>0 001 11 0</td>
<td>VNEG - Double-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>0 001 11 1</td>
<td>VSQRT - Double-precision scalar variant</td>
<td></td>
</tr>
<tr>
<td>0 010 01 0</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>0 010 10 0</td>
<td>VCVTB - Half-precision to single-precision variant</td>
<td></td>
</tr>
<tr>
<td>0 010 10 1</td>
<td>VCVTT - Half-precision to single-precision variant</td>
<td></td>
</tr>
<tr>
<td>0 010 11 0</td>
<td>VCVTB - Half-precision to double-precision variant</td>
<td></td>
</tr>
<tr>
<td>0 010 11 1</td>
<td>VCVTT - Half-precision to double-precision variant</td>
<td></td>
</tr>
<tr>
<td>0 011 01 0</td>
<td>VCVTB (BFloat16)</td>
<td>FEAT_AA32BF16</td>
</tr>
<tr>
<td>0 011 01 1</td>
<td>VCVTT (BFloat16)</td>
<td>FEAT_AA32BF16</td>
</tr>
<tr>
<td>0 011 10 0</td>
<td>VCVTB - Single-precision to half-precision variant</td>
<td></td>
</tr>
<tr>
<td>0 011 10 1</td>
<td>VCVTT - Single-precision to half-precision variant</td>
<td></td>
</tr>
<tr>
<td>0 011 11 0</td>
<td>VCVTB - Double-precision to half-precision variant</td>
<td></td>
</tr>
<tr>
<td>0 011 11 1</td>
<td>VCVTT - Double-precision to half-precision variant</td>
<td></td>
</tr>
<tr>
<td>0 100 01 0</td>
<td>VCMP</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>o1</td>
<td>opc2</td>
<td>size</td>
</tr>
<tr>
<td>----</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>11</td>
</tr>
</tbody>
</table>
Floating-point move immediate

This section describes the encoding of the Floating-point move immediate instruction class. The encodings in this section are decoded from Floating-point data-processing on page F4-7415.

```
| [31] 28|27 26 25 24|23 22 21 20|19 16|15 |12|11 10 9 8 7 6 5 4 |3 |0 |
| 1=1111 l cond | 1 1 1 0 | 1 | D | 1 | 1 | imm4H | Vd | 1 | 0 | size | 0 | 0 | 0 | imm4L |
```

### Decode fields

<table>
<thead>
<tr>
<th>o1</th>
<th>opc2</th>
<th>size</th>
<th>o3</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01x</td>
<td>01</td>
<td>-</td>
<td>VCVT (between floating-point and fixed-point, floating-point)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>01x</td>
<td>10</td>
<td>-</td>
<td>VCVT (between floating-point and fixed-point, floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>01x</td>
<td>11</td>
<td>-</td>
<td>VCVT (between floating-point and fixed-point, floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>01</td>
<td>0</td>
<td>VCVTR</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>01</td>
<td>1</td>
<td>VCVT (floating-point to integer, floating-point)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>10</td>
<td>0</td>
<td>VCVTR</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>10</td>
<td>1</td>
<td>VCVT (floating-point to integer, floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>11</td>
<td>0</td>
<td>VCVTR</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>11</td>
<td>1</td>
<td>VCVT (floating-point to integer, floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>01</td>
<td>0</td>
<td>VCVTR</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>01</td>
<td>1</td>
<td>VCVT (floating-point to integer, floating-point)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>10</td>
<td>0</td>
<td>VCVTR</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>10</td>
<td>1</td>
<td>VCVT (floating-point to integer, floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>11</td>
<td>0</td>
<td>VCVTR</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>11</td>
<td>1</td>
<td>VCVT (floating-point to integer, floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11x</td>
<td>01</td>
<td>-</td>
<td>VCVT (between floating-point and fixed-point, floating-point)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>11x</td>
<td>10</td>
<td>-</td>
<td>VCVT (between floating-point and fixed-point, floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11x</td>
<td>11</td>
<td>-</td>
<td>VCVT (between floating-point and fixed-point, floating-point)</td>
<td>-</td>
</tr>
</tbody>
</table>

### Floating-point move immediate

[0x0]11101 11 1 0 1 D 1 1 imm4H | Vd 1 0 size 0 0 0 imm4L cond

00 Unallocated.
01 VMOV (immediate) - Half-precision scalar variant FEAT_FP16
10 VMOV (immediate) - Single-precision scalar variant -
11 VMOV (immediate) - Double-precision scalar variant -
Floating-point data-processing (three registers)

This section describes the encoding of the Floating-point data-processing (three registers) instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page F4-7415.

![Decoder Table]

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>o0:o1 size o2</td>
<td>Vn Vd N o2 M Vm</td>
<td></td>
</tr>
<tr>
<td>111 00 -</td>
<td>Unallocated. -</td>
<td>-</td>
</tr>
<tr>
<td>000 01 0</td>
<td>VMLA (floating-point) - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>000 01 1</td>
<td>VMLS (floating-point) - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>000 10 0</td>
<td>VMLA (floating-point) - Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>000 10 1</td>
<td>VMLS (floating-point) - Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>000 11 0</td>
<td>VMLA (floating-point) - Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>000 11 1</td>
<td>VMLS (floating-point) - Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>001 01 0</td>
<td>VNMLS - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>001 01 1</td>
<td>VNMLA - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>001 10 0</td>
<td>VNMLS - Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>001 10 1</td>
<td>VNMLA - Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>001 11 0</td>
<td>VNMLS - Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>001 11 1</td>
<td>VNMLA - Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>010 01 0</td>
<td>VMUL (floating-point) - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>010 01 1</td>
<td>VMUL (floating-point) - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>010 10 0</td>
<td>VMUL (floating-point) - Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>010 10 1</td>
<td>VMUL (floating-point) - Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>010 11 0</td>
<td>VMUL (floating-point) - Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>010 11 1</td>
<td>VMUL (floating-point) - Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>011 01 0</td>
<td>VADD (floating-point) - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>011 01 1</td>
<td>VADD (floating-point) - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>011 10 0</td>
<td>VADD (floating-point) - Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>011 10 1</td>
<td>VADD (floating-point) - Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>011 11 0</td>
<td>VADD (floating-point) - Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>011 11 1</td>
<td>VADD (floating-point) - Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>100 01 0</td>
<td>VDIV - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
</tbody>
</table>
F4.1.18 Unconditional instructions

This section describes the encoding of the Unconditional instructions group. The encodings in this section are decoded from A32 instruction set encoding on page F4-7374.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 10 0</td>
<td>VDIV - Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>100 11 0</td>
<td>VDIV - Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>101 01 0</td>
<td>VFNMS - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>101 01 1</td>
<td>VFNMA - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>101 10 0</td>
<td>VFNS - Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>101 10 1</td>
<td>VFNS - Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>101 11 0</td>
<td>VFMS - Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>101 11 1</td>
<td>VFNS - Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>110 01 0</td>
<td>VFVA - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>110 01 1</td>
<td>VFMS - Half-precision scalar variant</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>110 10 0</td>
<td>VFMS - Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>110 10 1</td>
<td>VFMS - Single-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>110 11 0</td>
<td>VFMS - Double-precision scalar variant</td>
<td>-</td>
</tr>
<tr>
<td>110 11 1</td>
<td>VFMS - Double-precision scalar variant</td>
<td>-</td>
</tr>
</tbody>
</table>

Table F4-17 Encoding table for the Unconditional instructions group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>Miscellaneous on page F4-7421</td>
</tr>
<tr>
<td>01x</td>
<td>Advanced SIMD data-processing on page F4-7422</td>
</tr>
<tr>
<td>10x</td>
<td>Memory hints and barriers on page F4-7431</td>
</tr>
<tr>
<td>100 0</td>
<td>Advanced SIMD element or structure load/store on page F4-7434</td>
</tr>
<tr>
<td>101 0</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>11x 0</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
F4.1.19    Miscellaneous

This section describes the encoding of the Miscellaneous group. The encodings in this section are decoded from Unconditional instructions on page F4-7420.

<table>
<thead>
<tr>
<th>31</th>
<th>25 24</th>
<th>20</th>
<th>19</th>
<th></th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>111000</td>
<td>op0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table F4-18 Encoding table for the Miscellaneous group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xxxx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10000 xx0x</td>
<td>Change Process State</td>
<td>-</td>
</tr>
<tr>
<td>10001 1000</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10001 x100</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10001 xx01</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10001 0000</td>
<td>SETPAN</td>
<td>FEAT_PAN</td>
</tr>
<tr>
<td>10000 0111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10010 0111</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>-</td>
</tr>
<tr>
<td>10011 0111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10011 xx0x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>100xx 0011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>100xx 0x10</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>100xx 1x1x</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>101xx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>11xxx</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
</tbody>
</table>

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings on page K1-11574.

Change Process State

This section describes the encoding of the Change Process State instruction class. The encodings in this section are decoded from Miscellaneous.
F4.1.20 Advanced SIMD data-processing

This section describes the encoding of the Advanced SIMD data-processing group. The encodings in this section are decoded from Unconditional instructions on page F4-7420.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the A32 Advanced SIMD and floating-point instructions and their encoding on page F4-7440.

F4-7422

Advanced SIMD three registers of the same length

This section describes the encoding of the Advanced SIMD three registers of the same length instruction class. The encodings in this section are decoded from Advanced SIMD data-processing.
Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opc</th>
<th>Q</th>
<th>o1</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1100</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>VFMA</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1101</td>
<td>-</td>
<td>0</td>
<td></td>
<td>VADD (floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1101</td>
<td>-</td>
<td>1</td>
<td></td>
<td>VMLA (floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>-</td>
<td>0</td>
<td></td>
<td>VCEQ (register) - A2 on page F6-8260</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>-</td>
<td>0</td>
<td></td>
<td>VMAX (floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>-</td>
<td>1</td>
<td></td>
<td>VRECPS</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0000</td>
<td>-</td>
<td>0</td>
<td></td>
<td>VHADD</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>-</td>
<td>1</td>
<td></td>
<td>VAND (register)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0000</td>
<td>-</td>
<td>1</td>
<td></td>
<td>VQADD</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0001</td>
<td>-</td>
<td>0</td>
<td></td>
<td>VRHADD</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1100</td>
<td>-</td>
<td>0</td>
<td></td>
<td>SHA1C</td>
<td>FEAT_SHA1</td>
</tr>
<tr>
<td>-</td>
<td>0010</td>
<td>-</td>
<td>0</td>
<td></td>
<td>VHSUB</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>-</td>
<td>1</td>
<td></td>
<td>VBIC (register)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0010</td>
<td>-</td>
<td>1</td>
<td></td>
<td>VQSUB</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0011</td>
<td>-</td>
<td>0</td>
<td></td>
<td>VCGT (register)  - A1 on page F6-8274</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0011</td>
<td>-</td>
<td>1</td>
<td></td>
<td>VCGE (register)  - A1 on page F6-8267</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1100</td>
<td>-</td>
<td>0</td>
<td></td>
<td>SHA1P</td>
<td>FEAT_SHA1</td>
</tr>
<tr>
<td>0</td>
<td>1101</td>
<td>-</td>
<td>0</td>
<td></td>
<td>VSUB (floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>-</td>
<td>0</td>
<td></td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>-</td>
<td>0</td>
<td></td>
<td>VMIN (floating-point)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>-</td>
<td>1</td>
<td></td>
<td>VRSQRTS</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0100</td>
<td>-</td>
<td>0</td>
<td></td>
<td>VSHL (register)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>-</td>
<td>0</td>
<td></td>
<td>VADD (integer)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>-</td>
<td>1</td>
<td></td>
<td>VORR (register)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>-</td>
<td>1</td>
<td></td>
<td>VTST</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0100</td>
<td>-</td>
<td>1</td>
<td></td>
<td>VQSHL (register)</td>
<td>-</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opc</th>
<th>Q</th>
<th>o1</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>1001</td>
<td>-</td>
<td>0</td>
<td>VMLA (integer)</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0101</td>
<td>-</td>
<td>0</td>
<td>VRSHL</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0101</td>
<td>-</td>
<td>1</td>
<td>VQRSHL</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>1011</td>
<td>-</td>
<td>0</td>
<td>VQDMULH</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1100</td>
<td>-</td>
<td>0</td>
<td>SHA1M</td>
<td>FEAT_SHA1</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>1011</td>
<td>-</td>
<td>1</td>
<td>VPADD (integer)</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0110</td>
<td>-</td>
<td>0</td>
<td>VMAX (integer)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>0001</td>
<td>-</td>
<td>1</td>
<td>VORN (register)</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0110</td>
<td>-</td>
<td>1</td>
<td>VMIN (integer)</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0111</td>
<td>-</td>
<td>0</td>
<td>VABD (integer)</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0111</td>
<td>-</td>
<td>1</td>
<td>VABA</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>1100</td>
<td>-</td>
<td>0</td>
<td>SHA1SU0</td>
<td>FEAT_SHA1</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>1101</td>
<td>-</td>
<td>0</td>
<td>VPADD (floating-point)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>1101</td>
<td>-</td>
<td>1</td>
<td>VMUL (floating-point)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>1110</td>
<td>-</td>
<td>0</td>
<td>VCGE (register) - A2 on page F6-8267</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>1110</td>
<td>-</td>
<td>1</td>
<td>VACGE</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>1111</td>
<td>0</td>
<td>0</td>
<td>VPMAX (floating-point)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>1111</td>
<td>-</td>
<td>1</td>
<td>VMAXNM</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0001</td>
<td>-</td>
<td>1</td>
<td>VEOR</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>1001</td>
<td>-</td>
<td>1</td>
<td>VMUL (integer and polynomial)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1100</td>
<td>-</td>
<td>0</td>
<td>SHA256H</td>
<td>FEAT_SHA256</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>1010</td>
<td>0</td>
<td>0</td>
<td>VPMAX (integer)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>0001</td>
<td>-</td>
<td>1</td>
<td>VBSL</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>1010</td>
<td>0</td>
<td>1</td>
<td>VPMIN (integer)</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>1010</td>
<td>1</td>
<td>-</td>
<td>Unallocated.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1100</td>
<td>-</td>
<td>0</td>
<td>SHA256H2</td>
<td>FEAT_SHA256</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>1101</td>
<td>-</td>
<td>0</td>
<td>VABD (floating-point)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>1110</td>
<td>-</td>
<td>0</td>
<td>VCGT (register) - A2 on page F6-8274</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>1110</td>
<td>-</td>
<td>1</td>
<td>VACGT</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>1111</td>
<td>0</td>
<td>0</td>
<td>VPMIN (floating-point)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>1111</td>
<td>-</td>
<td>1</td>
<td>VMINNNM</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1000</td>
<td>-</td>
<td>0</td>
<td>VSUB (integer)</td>
<td></td>
</tr>
</tbody>
</table>
F4.1.21 Advanced SIMD two registers, or three registers of different lengths

This section describes the encoding of the Advanced SIMD two registers, or three registers of different lengths group. The encodings in this section are decoded from Advanced SIMD data-processing on page F4-7422.

Table F4-20 Encoding table for the Advanced SIMD two registers, or three registers of different lengths group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1 op2 op3</td>
<td>VEXT (byte elements)</td>
</tr>
<tr>
<td>0 11 - -</td>
<td>Advanced SIMD two registers misc on page F4-7425</td>
</tr>
<tr>
<td>1 11 0x -</td>
<td>VTBL, VTBX</td>
</tr>
<tr>
<td>1 11 11 -</td>
<td>Advanced SIMD duplicate (scalar) on page F4-7427</td>
</tr>
<tr>
<td>!= 11 - 0</td>
<td>Advanced SIMD three registers of different lengths on page F4-7428</td>
</tr>
<tr>
<td>!= 11 - 1</td>
<td>Advanced SIMD two registers and a scalar on page F4-7429</td>
</tr>
</tbody>
</table>

Advanced SIMD two registers misc

This section describes the encoding of the Advanced SIMD two registers misc instruction class. The encodings in this section are decoded from Advanced SIMD two registers, or three registers of different lengths.
<table>
<thead>
<tr>
<th>Decoded fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc1</td>
<td>opc2</td>
</tr>
<tr>
<td>00</td>
<td>00000</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00010</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00100</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00111</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>010x</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01100</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01101</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01110</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>01111</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10001</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10100</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>110x</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>11100</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>x0000</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>x0010</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>x0100</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>x0110</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>x1000</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>x1100</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>x1110</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>01010</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>11000</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>00010</td>
<td></td>
</tr>
<tr>
<td>Decode fields</td>
<td>Instruction page</td>
<td>Feature</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------</td>
<td>---------</td>
</tr>
<tr>
<td>size opc1 opc2 Q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 0010 -</td>
<td>VUZP</td>
<td>-</td>
</tr>
<tr>
<td>10 0011 -</td>
<td>VZIP</td>
<td>-</td>
</tr>
<tr>
<td>10 0100 0</td>
<td>VMOVN</td>
<td>-</td>
</tr>
<tr>
<td>10 0100 1</td>
<td>VQMOVN, VQMOVUN - Unsigned result variant</td>
<td>-</td>
</tr>
<tr>
<td>10 0101 -</td>
<td>VQMOVN, VQMOVUN - Signed result variant</td>
<td>-</td>
</tr>
<tr>
<td>10 0110 0</td>
<td>VSHLL</td>
<td>-</td>
</tr>
<tr>
<td>10 0111 0</td>
<td>SHAISU1</td>
<td>FEAT_SHA1</td>
</tr>
<tr>
<td>10 0111 1</td>
<td>SHA256SU0</td>
<td>FEAT_SHA256</td>
</tr>
<tr>
<td>10 1000 -</td>
<td>VRINTN (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>10 1001 -</td>
<td>VRINTX (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>10 1010 -</td>
<td>VRINTA (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>10 1011 -</td>
<td>VRINTZ (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>10 1100 1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10 1100 0</td>
<td>VCVT (between half-precision and single-precision, Advanced SIMD) - Single-precision to half-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>10 1101 -</td>
<td>VRINTM (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>10 1110 0</td>
<td>VCVT (between half-precision and single-precision, Advanced SIMD) - Half-precision to single-precision variant</td>
<td>-</td>
</tr>
<tr>
<td>10 1110 1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>10 1111 -</td>
<td>VRINTP (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>11 000x -</td>
<td>VCVTA (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>11 001x -</td>
<td>VCVTN (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>11 010x -</td>
<td>VCVTP (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>11 011x -</td>
<td>VCVTM (Advanced SIMD)</td>
<td>-</td>
</tr>
<tr>
<td>11 10x0 -</td>
<td>VRECPE</td>
<td>-</td>
</tr>
<tr>
<td>11 10x1 -</td>
<td>VRSQRTE</td>
<td>-</td>
</tr>
<tr>
<td>11 1100 1</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>11 11xx -</td>
<td>VCVT (between floating-point and integer, Advanced SIMD)</td>
<td>-</td>
</tr>
</tbody>
</table>

**Advanced SIMD duplicate (scalar)**

This section describes the encoding of the Advanced SIMD duplicate (scalar) instruction class. The encodings in this section are decoded from *Advanced SIMD two registers, or three registers of different lengths* on page F4-7425.
This section describes the encoding of the Advanced SIMD three registers of different lengths instruction class. The encodings in this section are decoded from Advanced SIMD two registers, or three registers of different lengths on page F4-7425.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 12</th>
<th>11 10 9</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1 1 D 1 1</td>
<td>imm4</td>
<td>Vd</td>
<td>1 1</td>
<td>opc</td>
<td>Q</td>
<td>M</td>
<td>0</td>
</tr>
</tbody>
</table>

**Advanced SIMD three registers of different lengths**

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>VDUP (scalar)</td>
</tr>
<tr>
<td>001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>01x</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1xx</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 12</th>
<th>11 10 9</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 0 1</td>
<td>U</td>
<td>1</td>
<td>D</td>
<td>!=11</td>
<td>Vn</td>
<td>Vd</td>
<td>opc</td>
</tr>
</tbody>
</table>

**Decode fields**

<table>
<thead>
<tr>
<th>U</th>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0000</td>
<td>VADDL</td>
</tr>
<tr>
<td>-</td>
<td>0001</td>
<td>VADDW</td>
</tr>
<tr>
<td>-</td>
<td>0010</td>
<td>VSUBL</td>
</tr>
<tr>
<td>0</td>
<td>0100</td>
<td>VADDDN</td>
</tr>
<tr>
<td>-</td>
<td>0011</td>
<td>VSUBW</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>VSUBHN</td>
</tr>
<tr>
<td>0</td>
<td>1001</td>
<td>VQDMLAL</td>
</tr>
<tr>
<td>-</td>
<td>0101</td>
<td>VABAL</td>
</tr>
<tr>
<td>0</td>
<td>1011</td>
<td>VQDMLSL</td>
</tr>
<tr>
<td>0</td>
<td>1101</td>
<td>VQDMULL</td>
</tr>
<tr>
<td>-</td>
<td>0111</td>
<td>VABDL (integer)</td>
</tr>
<tr>
<td>-</td>
<td>1000</td>
<td>VMLAL (integer)</td>
</tr>
<tr>
<td>-</td>
<td>1010</td>
<td>VMLSL (integer)</td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
<td>VRADDDN</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
<td>VRSUBHN</td>
</tr>
</tbody>
</table>
Advanced SIMD two registers and a scalar

This section describes the encoding of the Advanced SIMD two registers and a scalar instruction class. The encodings in this section are decoded from Advanced SIMD two registers, or three registers of different lengths on page F4-7425.

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 8 7 6 5 4 3 0 |
| 1 1 1 0 0 1 | Q 1 | D != 1 | Vn | Vd | opc | N | f | M | 0 | Vm |
```

### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>opc</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>11x0</td>
<td>VMULL (integer and polynomial)</td>
</tr>
<tr>
<td>1</td>
<td>1001</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1011</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>1101</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>-</td>
<td>1111</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>Q</th>
<th>opc</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>000x</td>
<td>VMLA (by scalar)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>VQDMLAL</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0010</td>
<td>VMLAL (by scalar)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>VQDMLSL</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>010x</td>
<td>VMLS (by scalar)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1011</td>
<td>VQDMULL</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0110</td>
<td>VMULS (by scalar)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>100x</td>
<td>VMUL (by scalar)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>1010</td>
<td>VMULL (by scalar)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0111</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>1100</td>
<td>VQDMULH</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>1101</td>
<td>VQRDMULH</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1011</td>
<td>Unallocated.</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>1110</td>
<td>VQRDMLAH</td>
<td>FEAT_RDM</td>
</tr>
<tr>
<td>-</td>
<td>1111</td>
<td>VQRDMLSH</td>
<td>FEAT_RDM</td>
</tr>
</tbody>
</table>
F4.1.22 Advanced SIMD shifts and immediate generation

This section describes the encoding of the Advanced SIMD shifts and immediate generation group. The encodings in this section are decoded from Advanced SIMD data-processing on page F4-7422.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>25 24 23 22 21</td>
<td>7 6 5 4 3 0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>op0</td>
<td></td>
</tr>
</tbody>
</table>

Table F4-21 Encoding table for the Advanced SIMD shifts and immediate generation group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>000xxxxxxxxxxx0</td>
<td>Advanced SIMD one register and modified immediate on page F4-7430</td>
</tr>
<tr>
<td>!= 000xxxxxxxxxxx0</td>
<td>Advanced SIMD two registers and shift amount on page F4-7431</td>
</tr>
</tbody>
</table>

Advanced SIMD one register and modified immediate

This section describes the encoding of the Advanced SIMD one register and modified immediate instruction class. The encodings in this section are decoded from Advanced SIMD shifts and immediate generation.

<table>
<thead>
<tr>
<th></th>
<th>25 24 23 22 21</th>
<th>12 11 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30 29 28</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 1 0 1</td>
<td>Vd  cmode 0 op 1 imm4</td>
</tr>
</tbody>
</table>

Table F4-21 Encoding table for the Advanced SIMD shifts and immediate generation group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmode op</td>
<td></td>
</tr>
<tr>
<td>0xx0</td>
<td>VMOV (immediate) - A1 on page F6-8539</td>
</tr>
<tr>
<td>0xx0</td>
<td>VMVN (immediate) - A1 on page F6-8586</td>
</tr>
<tr>
<td>0xx1</td>
<td>VORR (immediate) - A1 on page F6-8610</td>
</tr>
<tr>
<td>0xx1</td>
<td>VBIC (immediate) - A1 on page F6-8244</td>
</tr>
<tr>
<td>10x0</td>
<td>VMOV (immediate) - A3 on page F6-8540</td>
</tr>
<tr>
<td>10x0</td>
<td>VMVN (immediate) - A2 on page F6-8586</td>
</tr>
<tr>
<td>10x1</td>
<td>VORR (immediate) - A2 on page F6-8610</td>
</tr>
<tr>
<td>10x1</td>
<td>VBIC (immediate) - A2 on page F6-8244</td>
</tr>
<tr>
<td>11xx</td>
<td>VMOV (immediate) - A4 on page F6-8540</td>
</tr>
<tr>
<td>110x</td>
<td>VMVN (immediate) - A3 on page F6-8587</td>
</tr>
<tr>
<td>1110</td>
<td>VMOV (immediate) - A5 on page F6-8541</td>
</tr>
<tr>
<td>1111</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>
Advanced SIMD two registers and shift amount

This section describes the encoding of the Advanced SIMD two registers and shift amount instruction class. The encodings in this section are decoded from Advanced SIMD shifts and immediate generation on page F4-7430.

F4.1.23 Memory hints and barriers

This section describes the encoding of the Memory hints and barriers group. The encodings in this section are decoded from Unconditional instructions on page F4-7420.
The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings on page K1-11574.

Barriers

This section describes the encoding of the Barriers instruction class. The encodings in this section are decoded from Memory hints and barriers on page F4-7431.

Table F4-22 Encoding table for the Memory hints and barriers group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>00xx1</td>
<td>CONSTRAINED UNPREDICTABLE</td>
</tr>
<tr>
<td>01001</td>
<td>CONSTRAINED UNPREDICTABLE</td>
</tr>
<tr>
<td>01011</td>
<td>Barriers</td>
</tr>
<tr>
<td>011x1</td>
<td>CONSTRAINED UNPREDICTABLE</td>
</tr>
<tr>
<td>0xxx0</td>
<td>Preload (immediate) on page F4-7433</td>
</tr>
<tr>
<td>1xxx0</td>
<td>Preload (register) on page F4-7433</td>
</tr>
<tr>
<td>1xxx1</td>
<td>CONSTRAINED UNPREDICTABLE</td>
</tr>
<tr>
<td>1xxx</td>
<td>Unallocated.</td>
</tr>
</tbody>
</table>

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings on page K1-11574.

Barriers

This section describes the encoding of the Barriers instruction class. The encodings in this section are decoded from Memory hints and barriers on page F4-7431.

Table F4-22 Encoding table for the Memory hints and barriers group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode option</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>001x</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>!= 0x00</td>
<td>DSB</td>
</tr>
<tr>
<td>0100</td>
<td>0000</td>
<td>SSBB</td>
</tr>
<tr>
<td>0100</td>
<td>0100</td>
<td>PSSBB</td>
</tr>
<tr>
<td>0101</td>
<td>-</td>
<td>DMB</td>
</tr>
</tbody>
</table>
The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings on page K1-11574.

Preload (immediate)

This section describes the encoding of the Preload (immediate) instruction class. The encodings in this section are decoded from Memory hints and barriers on page F4-7431.

```
[31 30 29 28 27 26 25 24 23 22 21 20 19] [16 15 14 13 12 11] [7 6 5 4 3 0]  Rn (1) (1) (1) (1) (1) imm12
0 1 1 1 0 1 0 D U R 0 1                     
```

Preload (register)

This section describes the encoding of the Preload (register) instruction class. The encodings in this section are decoded from Memory hints and barriers on page F4-7431.

```
[31 30 29 28 27 26 25 24 23 22 21 20 19] [16 15 14 13 12 11] [7 6 5 4 3 0]  Rm
1 1 1 0 1 1 D U o2 0 1                     
```

---

**Decode fields**

<table>
<thead>
<tr>
<th>opcode</th>
<th>option</th>
<th>Instruction page</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110</td>
<td>-</td>
<td>ISB</td>
<td>-</td>
</tr>
<tr>
<td>0111</td>
<td>-</td>
<td>SB</td>
<td>FEAT_SB</td>
</tr>
<tr>
<td>1xxx</td>
<td>-</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>-</td>
</tr>
</tbody>
</table>

---

---

**Decode fields**

<table>
<thead>
<tr>
<th>D</th>
<th>R</th>
<th>Rn</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>Reserved hint, behaves as NOP.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>PLI (immediate, literal)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1111</td>
<td>PLD (literal)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 1111</td>
<td>PLD, PLDW (immediate) - Preload write variant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>PLD, PLDW (immediate) - Preload read variant</td>
</tr>
</tbody>
</table>

---

**Decode fields**

<table>
<thead>
<tr>
<th>D</th>
<th>o2</th>
<th>imm5:stype</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>Reserved hint, behaves as NOP.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>![= 0000011]</td>
<td>PLI (register) - Shift or rotate by value variant</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0000011</td>
<td>PLI (register) - Rotate right with extend variant</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>![= 0000011]</td>
<td>PLD, PLDW (register) - Preload write, optional shift or rotate variant</td>
</tr>
</tbody>
</table>
F4.1.24 Advanced SIMD element or structure load/store

This section describes the encoding of the Advanced SIMD element or structure load/store group. The encodings in this section are decoded from Unconditional instructions on page F4-7420.

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the A32 Advanced SIMD and floating-point instructions and their encoding on page F4-7440.

Table F4-23 Encoding table for the Advanced SIMD element or structure load/store group

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D o2 imm5:stype</td>
<td>PLD, PLDW (register) - Preload write, rotate right with extend variant</td>
</tr>
<tr>
<td>1 0 000011</td>
<td></td>
</tr>
<tr>
<td>1 1 != 000011</td>
<td>PLD, PLDW (register) - Preload read, optional shift or rotate variant</td>
</tr>
<tr>
<td>1 1 000011</td>
<td>PLD, PLDW (register) - Preload read, rotate right with extend variant</td>
</tr>
</tbody>
</table>

Advanced SIMD load/store multiple structures

This section describes the encoding of the Advanced SIMD load/store multiple structures instruction class. The encodings in this section are decoded from Advanced SIMD element or structure load/store.

Table F4-24 Encoding table for the Advanced SIMD load/store multiple structures on page F4-7434

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Decode group or instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>0 -</td>
<td>Advanced SIMD load/store multiple structures on page F4-7434</td>
</tr>
<tr>
<td>1 11</td>
<td>Advanced SIMD load single structure to all lanes on page F4-7436</td>
</tr>
<tr>
<td>1 != 11</td>
<td>Advanced SIMD load/store single structure to one lane on page F4-7437</td>
</tr>
</tbody>
</table>

Advanced SIMD load/store single structure to all lanes
<table>
<thead>
<tr>
<th>L</th>
<th>itype</th>
<th>Rm</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>1101</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>1111</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>!= 11x1</td>
<td>VST2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>1101</td>
<td>VST2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>1111</td>
<td>VST2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>010x</td>
<td>!= 11x1</td>
<td>VST3 (multiple 3-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>010x</td>
<td>1101</td>
<td>VST3 (multiple 3-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>010x</td>
<td>1111</td>
<td>VST3 (multiple 3-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>!= 11x1</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>1101</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>1111</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>!= 11x1</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>1101</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>1111</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>100x</td>
<td>!= 11x1</td>
<td>VST2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>100x</td>
<td>1101</td>
<td>VST2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>100x</td>
<td>1111</td>
<td>VST2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td>!= 11x1</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td>1101</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td>1111</td>
<td>VST1 (multiple single elements)</td>
</tr>
<tr>
<td>1</td>
<td>000x</td>
<td>!= 11x1</td>
<td>VLD4 (multiple 4-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>000x</td>
<td>1101</td>
<td>VLD4 (multiple 4-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>000x</td>
<td>1111</td>
<td>VLD4 (multiple 4-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>!= 11x1</td>
<td>VLD1 (multiple single elements)</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>1101</td>
<td>VLD1 (multiple single elements)</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>1111</td>
<td>VLD1 (multiple single elements)</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>!= 11x1</td>
<td>VLD2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>1101</td>
<td>VLD2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>1111</td>
<td>VLD2 (multiple 2-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>010x</td>
<td>!= 11x1</td>
<td>VLD3 (multiple 3-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>010x</td>
<td>1101</td>
<td>VLD3 (multiple 3-element structures)</td>
</tr>
<tr>
<td>1</td>
<td>010x</td>
<td>1111</td>
<td>VLD3 (multiple 3-element structures)</td>
</tr>
</tbody>
</table>
### Advanced SIMD load single structure to all lanes

This section describes the encoding of the Advanced SIMD load single structure to all lanes instruction class. The encodings in this section are decoded from *Advanced SIMD element or structure load/store on page F4-7434.*

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>iotype</td>
</tr>
<tr>
<td>0</td>
<td>1011</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
</tr>
<tr>
<td>1</td>
<td>0111</td>
</tr>
<tr>
<td>1</td>
<td>0111</td>
</tr>
<tr>
<td>1</td>
<td>0111</td>
</tr>
<tr>
<td>-</td>
<td>11xx</td>
</tr>
<tr>
<td>1</td>
<td>100x</td>
</tr>
<tr>
<td>1</td>
<td>100x</td>
</tr>
<tr>
<td>1</td>
<td>100x</td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
</tr>
</tbody>
</table>

---

### Decode fields

<table>
<thead>
<tr>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
### Advanced SIMD load/store single structure to one lane

This section describes the encoding of the Advanced SIMD load/store single structure to one lane instruction class. The encodings in this section are decoded from *Advanced SIMD element or structure load/store* on page F4-7434.

#### Decode fields

<table>
<thead>
<tr>
<th>L</th>
<th>N</th>
<th>a</th>
<th>Rm</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
<td>1111</td>
<td>VLD3 (single 3-element structure to all lanes)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>-</td>
<td>Unallocated.</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>-</td>
<td>!= 11x1</td>
<td>VLD4 (single 4-element structure to all lanes)</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1101</td>
<td>VLD4 (single 4-element structure to all lanes)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1111</td>
<td>VLD4 (single 4-element structure to all lanes)</td>
<td></td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>L</th>
<th>size</th>
<th>N</th>
<th>Rm</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>!= 11x1</td>
<td>VST1 (single element from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>1101</td>
<td>VST1 (single element from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>1111</td>
<td>VST1 (single element from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>01</td>
<td>!= 11x1</td>
<td>VST2 (single 2-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>01</td>
<td>1101</td>
<td>VST2 (single 2-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>01</td>
<td>1111</td>
<td>VST2 (single 2-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>10</td>
<td>!= 11x1</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>10</td>
<td>1101</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>10</td>
<td>1111</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>11</td>
<td>!= 11x1</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>11</td>
<td>1101</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>11</td>
<td>1111</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>00</td>
<td>!= 11x1</td>
<td>VST1 (single element from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>00</td>
<td>1101</td>
<td>VST1 (single element from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>00</td>
<td>1111</td>
<td>VST1 (single element from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01</td>
<td>!= 11x1</td>
<td>VST2 (single 2-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01</td>
<td>1101</td>
<td>VST2 (single 2-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01</td>
<td>1111</td>
<td>VST2 (single 2-element structure from one lane)</td>
</tr>
<tr>
<td>L</td>
<td>size</td>
<td>N</td>
<td>Rm</td>
<td>Instruction page</td>
</tr>
<tr>
<td>---</td>
<td>------</td>
<td>---</td>
<td>-----</td>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>10</td>
<td>!= 11x1</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>10</td>
<td>1101</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>10</td>
<td>1111</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11</td>
<td>!= 11x1</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11</td>
<td>1101</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11</td>
<td>1111</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>00</td>
<td>!= 11x1</td>
<td>VST1 (single element from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>00</td>
<td>1101</td>
<td>VST1 (single element from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>00</td>
<td>1111</td>
<td>VST1 (single element from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01</td>
<td>!= 11x1</td>
<td>VST2 (single 2-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01</td>
<td>1101</td>
<td>VST2 (single 2-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01</td>
<td>1111</td>
<td>VST2 (single 2-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
<td>!= 11x1</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
<td>1101</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
<td>1111</td>
<td>VST3 (single 3-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
<td>!= 11x1</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
<td>1101</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
<td>1111</td>
<td>VST4 (single 4-element structure from one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>00</td>
<td>!= 11x1</td>
<td>VLD1 (single element to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>00</td>
<td>1101</td>
<td>VLD1 (single element to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>00</td>
<td>1111</td>
<td>VLD1 (single element to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>!= 11x1</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>1101</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>1111</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>10</td>
<td>!= 11x1</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>10</td>
<td>1101</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>10</td>
<td>1111</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>11</td>
<td>!= 11x1</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>11</td>
<td>1101</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>11</td>
<td>1111</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>00</td>
<td>!= 11x1</td>
<td>VLD1 (single element to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>00</td>
<td>1101</td>
<td>VLD1 (single element to one lane)</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th></th>
<th>size</th>
<th>N</th>
<th>Rm</th>
<th>Instruction page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>0</td>
<td>1111</td>
<td>VLD1 (single element to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>01</td>
<td>!= 11x1</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>01</td>
<td>1101</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>01</td>
<td>1111</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>10</td>
<td>!= 11x1</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>10</td>
<td>1101</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>10</td>
<td>1111</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11</td>
<td>11x1</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11</td>
<td>1101</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11</td>
<td>1111</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>!= 11x1</td>
<td>VLD1 (single element to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>00</td>
<td>1101</td>
<td>VLD1 (single element to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>00</td>
<td>1111</td>
<td>VLD1 (single element to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>01</td>
<td>!= 11x1</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>01</td>
<td>1101</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>01</td>
<td>1111</td>
<td>VLD2 (single 2-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>!= 11x1</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>1101</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>1111</td>
<td>VLD3 (single 3-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td>!= 11x1</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td>1101</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td>1111</td>
<td>VLD4 (single 4-element structure to one lane)</td>
</tr>
</tbody>
</table>
F4.2 About the A32 Advanced SIMD and floating-point instructions and their encoding

The Advanced SIMD and floating-point instructions are common to the T32 and A32 instruction sets. These instructions perform Advanced SIMD and floating-point operations on a common register file, the SIMD&FP register file. This means:

- In general, the instructions that load or store registers in this file, or move data between general-purpose registers and this register file, are common to the Advanced SIMD and floating-point instructions.
- There are distinct Advanced SIMD data-processing instructions and floating-point data-processing instructions.

All A32 Advanced SIMD and floating-point instructions have 32-bit encodings. Different groups of these instructions are decoded from different points in the 32-bit A32 instruction decode structure. Table F4-24 shows these instruction groups, and where each group is decoded from the overall A32 decode structure:

<table>
<thead>
<tr>
<th>Advanced SIMD and floating-point instruction group</th>
<th>A32 decode is from</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced SIMD and System register load/store and 64-bit move on page F4-7410</td>
<td>System register access, Advanced SIMD, floating-point, and Supervisor call on page F4-7402</td>
</tr>
<tr>
<td>Floating-point data-processing on page F4-7415</td>
<td>System register access, Advanced SIMD, floating-point, and Supervisor call on page F4-7402</td>
</tr>
<tr>
<td>Advanced SIMD and System register 32-bit move on page F4-7413</td>
<td>System register access, Advanced SIMD, floating-point, and Supervisor call on page F4-7402</td>
</tr>
<tr>
<td>Advanced SIMD data-processing on page F4-7422</td>
<td>Unconditional instructions on page F4-7420</td>
</tr>
<tr>
<td>Advanced SIMD element or structure load/store on page F4-7434</td>
<td>Unconditional instructions on page F4-7420</td>
</tr>
</tbody>
</table>
Chapter F5
T32 and A32 Base Instruction Set Instruction Descriptions

This chapter describes each instruction. It contains the following sections:

- *Alphabetical list of T32 and A32 base instruction set instructions* on page F5-7442.
- *Encoding and use of banked register transfer instructions* on page F5-8160.
F5.1  Alphabetic list of T32 and A32 base instruction set instructions

This section lists every instruction in the T32 and A32 base instruction sets. For details of the format used see Format of instruction descriptions on page F1-7224.

This section is formatted so that each instruction description starts on a new page.
F5.1.1 ADC, ADCS (immediate)

Add with Carry (immediate) adds an immediate value and the Carry flag value to a register value, and writes the result to the destination register.

If the destination register is not the PC, the ADCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- The ADCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

```
<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1=1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**ADC variant**

Applies when S == 0.

ADC{<c>}{<q>} {<Rd>,} <Rn>, #<const>

**ADCS variant**

Applies when S == 1.

ADCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

**Decode for all variants of this encoding**

\[ d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ \text{setflags} = (S == '1'); \ \text{imm32} = A32ExpandImm(imm12); \]

T1

```
|15|14|13|12|11|10|9|8|7|6|5|4|3|0|15|14|12|11|8|7|0|0|
|1|1|1|0|1|0|1|0|1|0|S|Rn|0|imm3|Rd|imm8|
```

**ADC variant**

Applies when S == 0.

ADC{<c>}{<q>} {<Rd>,} <Rn>, #<const>

**ADCS variant**

Applies when S == 1.
ADCS{<c>}{<q>}{<Rd>,}<Rn>,<const>

Decode for all variants of this encoding

```plaintext
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 || n == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- `<c>`: See *Standard assembler syntax fields on page F1-7228.*
- `<q>`: See *Standard assembler syntax fields on page F1-7228.*
- `<Rd>`: For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:
  - For the ADC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.*
  - For the ADCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.
- `<Rn>`: For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
- `<const>`: For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions on page F1-7244* for the range of values.

Operation for all encodings

```plaintext
if ConditionPassed() then
  EncodingSpecificOperations();
  (result, nzcv) = AddWithCarry(R[n], imm32, PSTATE.C);
  if d == 15 then          // Can only occur for A32 encoding
    if setflags then
      ALUExceptionReturn(result);
    else
      ALUWritePC(result);
  else
    R[d] = result;
    if setflags then
      PSTATE.<N,Z,C,V> = nzcv;
```
Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.2 ADC, ADCS (register)

Add with Carry (register) adds a register value, the Carry flag value, and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ADCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-7133.
- The ADCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

| 31 28 27 26 25 24| 23 22 21 20| 19 16| 15| 12| 11 | 7 6 5 4 3 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| l=1111| 0 0 0 0| 1 0 1| S| Rn| Rd| imm5| stype| 0 | Rm |

**ADC, rotate right with extend variant**

Applies when S == 0 & imm5 == 00000 & stype == 11.

ADC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

**ADC, shift or rotate by value variant**

Applies when S == 0 & !(imm5 == 00000 & stype == 11).

ADC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

**ADCS, rotate right with extend variant**

Applies when S == 1 & imm5 == 00000 & stype == 11.

ADCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

**ADCS, shift or rotate by value variant**

Applies when S == 1 & !(imm5 == 00000 & stype == 11).

ADCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

**Decode for all variants of this encoding**

\[ d = \text{UInt}(Rd); n = \text{UInt}(Rn); m = \text{UInt}(Rm); \text{setflags} = (S == '1'); \]
\[ (\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{imm5}); \]
T1

T1 variant

ADC<{q}> {<Rdn>,} <Rdn>, <Rm> // Inside IT block
ADCS<{q}> {<Rdn>,} <Rdn>, <Rm> // Outside IT block

Decode for this encoding

d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);

T2

ADC, rotate right with extend variant

Applies when \( S == 0 \) \&\& \( \text{imm3} == 000 \) \&\& \( \text{imm2} == 00 \) \&\& \( \text{stype} == 11 \).

ADC<{c}>{q}> {<Rd>,} <Rn>, <Rm>, RRX

ADC, shift or rotate by value variant

Applies when \( S == 0 \) \&\& \( !(\text{imm3} == 000 \) \&\& \( \text{imm2} == 00 \) \&\& \( \text{stype} == 11 \).

ADC.W {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ADCS<{c}>{q}> {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

ADCS, rotate right with extend variant

Applies when \( S == 1 \) \&\& \( \text{imm3} == 000 \) \&\& \( \text{imm2} == 00 \) \&\& \( \text{stype} == 11 \).

ADCS<{c}>{q}> {<Rd>,} <Rn>, <Rm>, RRX

ADCS, shift or rotate by value variant

Applies when \( S == 1 \) \&\& \( !(\text{imm3} == 000 \) \&\& \( \text{imm2} == 00 \) \&\& \( \text{stype} == 11 \).

ADCS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ADCS<{c}>{q}> {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<c>
See Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

<Rdn>
Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd>
For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ADC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- For the ADCS variant, the instruction performs an exception return, that restores PSTATE from SPSR.<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn>
For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm>
For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift>
Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- LSL when stype = 00
- LSR when stype = 01
- ASR when stype = 10
- ROR when stype = 11

<amount>
For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

In T32 assembly:

- Outside an IT block, if ADCS <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ADCS <Rd>, <Rn> had been written.

- Inside an IT block, if ADC<< <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ADC<< <Rd>, <Rn> had been written.

To prevent either of these happening, use the .W qualifier.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
  (result, nzcv) = AddWithCarry(R[n], shifted, PSTATE.C);
  if d == 15 then          // Can only occur for A32 encoding
    if setflags then
      ALUExceptionReturn(result);
    else
      ALUWritePC(result);
else
  R[d] = result;
  if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.3 ADC, ADCS (register-shifted register)

Add with Carry (register-shifted register) adds a register value, the Carry flag value, and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

A1

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
| !\equiv 1111 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | S | Rn | Rd | Rs | 0 | sftype | 1 | Rm |
| cond |

Flag setting variant

Applies when \( S = 1 \).

\[
\text{ADCS}\{<c>\}{<q>} \{<Rd>,} <Rn>, <Rm>, <\text{shift}> <Rs>
\]

Not flag setting variant

Applies when \( S = 0 \).

\[
\text{ADC}\{<c>\}{<q>} \{<Rd>,} <Rn>, <Rm>, <\text{shift}> <Rs>
\]

Decode for all variants of this encoding

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \ s = \text{UInt}(Rs);
\setflags = (S == '1'); \ \text{shift}_t = \text{DecodeRegShift}(\text{stype});
\text{if} \ d == 15 || n == 15 || m == 15 || s == 15 \text{ then UNPREDICTABLE;}
\]

Notes for all encodings

For more information about the CONstrained UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\)

See Standard assembler syntax fields on page F1-7228.

\(<q>\)

See Standard assembler syntax fields on page F1-7228.

\(<Rd>\)

Is the general-purpose destination register, encoded in the "Rd" field.

\(<Rn>\)

Is the first general-purpose source register, encoded in the "Rn" field.

\(<Rm>\)

Is the second general-purpose source register, encoded in the "Rm" field.

\(<\text{shift}>\)

Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

<table>
<thead>
<tr>
<th>Shift</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL</td>
<td>when stype = 00</td>
</tr>
<tr>
<td>LSR</td>
<td>when stype = 01</td>
</tr>
<tr>
<td>ASR</td>
<td>when stype = 10</td>
</tr>
<tr>
<td>ROR</td>
<td>when stype = 11</td>
</tr>
</tbody>
</table>

\(<Rs>\)

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.4 ADD, ADDS (immediate)

Add (immediate) adds an immediate value to a register value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.

- The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

<table>
<thead>
<tr>
<th>31 28 27 26 25 24 23 22 21 20 19 16 15 12 11</th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Rn</td>
<td>Rd</td>
<td>imm12</td>
</tr>
</tbody>
</table>

ADD variant

Applies when S == 0 && Rn != 11x1.

ADD{c}{q} {<Rd>,} <Rn>, #<const>

ADDS variant

Applies when S == 1 && Rn != 1101.

ADDS{c}{q} {<Rd>,} <Rn>, #<const>

Decode for all variants of this encoding

if Rn == '1111' && S == '0' then SEE "ADR";
if Rn == '1101' then SEE "ADD (SP plus immediate)";
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = A32ExpandImm(imm12);

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>6 5</th>
<th>3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 1 0</td>
<td>imm3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rd</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T1 variant

ADD{c}{q} <Rd>, <Rn>, #<imm3> // Inside IT block
ADDS{c}{q} <Rd>, <Rn>, #<imm3> // Outside IT block
Decode for this encoding

d = UInt(Rd); n = UInt(Rn); setflags = !InITBlock(); imm32 = ZeroExtend(imm3, 32);

T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rd</td>
<td>imm8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T2 variant

ADD<c>{<q>} <Rdn>, #<imm8> // Inside IT block, and <Rdn>, <imm8> can be represented in T1
ADD<c>{<q>} {<Rdn>,} <Rdn>, #<imm8> // Inside IT block, and <Rdn>, <imm8> cannot be represented in T1
ADDS{<q>} <Rdn>, #<imm8> // Outside IT block, and <Rdn>, <imm8> can be represented in T1
ADDS{<q>} {<Rdn>,} <Rdn>, #<imm8> // Outside IT block, and <Rdn>, <imm8> cannot be represented in T1

Decode for this encoding

d = UInt(Rd); n = UInt(Rd); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32);

T3

| 15 | 14 | 13 | 12| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 14 | 12| 11 | 8 | 7 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | S | 1=1101 | 0  | imm3 | Rd | imm8 |

ADD variant

Applies when S == 0.

ADD<c>.W {<Rd>,} <Rn>, #<const> // Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2
ADD<{<q}>{<q}> {<Rd>,} <Rn>, #<const>

ADDS variant

Applies when S == 1 && Rd != 1111.

ADDS.W {<Rd>,} <Rn>, #<const> // Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2
ADDS{<q}>{<q}> {<Rd>,} <Rn>, #<const>

Decode for all variants of this encoding

if Rd == '1111' && S == '1' then SEE "CMN (immediate)";
if Rn == '1101' then SEE "ADD (SP plus immediate)"

if d == UInt(Rd); n = UInt(Rn); setflags = {S == '1'}; imm32 = T32ExpandImm(i:imm3:imm8);
if (d == IS && !setflags) || n == IS then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

T4

| 15 | 14 | 13 | 12| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 14 | 12| 11 | 8 | 7 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1=11x1 | 0  | imm3 | Rd | imm8 |

Rn
T4 variant

ADD{<c>}{<q>} {<Rd>,} <Rn>, #<imm12> // <imm12> cannot be represented in T1, T2, or T3
ADDW{<c>}{<q>} {<Rd>,} <Rn>, #<imm12> // <imm12> can be represented in T1, T2, or T3

Decode for this encoding

if Rn == '1111' then SEE "ADR";
if Rn == '1101' then SEE "ADD (SP plus immediate)";

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad \text{setflags} = \text{FALSE}; \quad \text{imm32} = \text{ZeroExtend}(i:imm3:imm8, 32);
\]
if d == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<\>
See Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

<Rdn>
Is the general-purpose source and destination register, encoded in the "Rdn" field.

<imm8>
Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

<Rd>
For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. If the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of this instruction.

For encoding T1, T3 and T4: is the general-purpose destination register, encoded in the "Rd" field.
If omitted, this register is the same as <Rn>.

<Rn>
For encoding A1 and T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see ADD, ADDS (SP plus immediate). If the PC is used, see ADR.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.
For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see ADD, ADDS (SP plus immediate).

<imm3>
Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field.

<imm12>
Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

<const>
For encoding A1: an immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values.
For encoding T3: an immediate value. See Modified immediate constants in T32 instructions on page F1-7242 for the range of values.

When multiple encodings of the same length are available for an instruction, encoding T3 is preferred to encoding T4 (if encoding T4 is required, use the ADDW syntax). Encoding T1 is preferred to encoding T2 if <Rd> is specified and encoding T2 is preferred to encoding T1 if <Rd> is omitted.
Operation for all encodings

if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], imm32, '0');
    if d == 15 then          // Can only occur for A32 encoding
      if setflags then
        ALUExceptionReturn(result);
      else
        ALUWritePC(result);
    else
      R[d] = result;
      if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
  else
    if ConditionPassed() then
      EncodingSpecificOperations();
      (result, nzcv) = AddWithCarry(R[n], imm32, '0');
      R[d] = result;
      if setflags then
        PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.5  ADD, ADDS (register)

Add (register) adds a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

• The ADD variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.

• The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  — The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  — The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  — The instruction is UNDEFINED in Hyp mode.
  — The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

```
<table>
<thead>
<tr>
<th>Stype</th>
<th>imm5</th>
<th>Rd</th>
<th>Rm</th>
<th>shift</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

ADD, rotate right with extend variant

Applies when S == 0 && imm5 == 00000 && stype == 11.

ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

ADD, shift or rotate by value variant

Applies when S == 0 && !(imm5 == 00000 && stype == 11).

ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

ADDS, rotate right with extend variant

Applies when S == 1 && imm5 == 00000 && stype == 11.

ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

ADDS, shift or rotate by value variant

Applies when S == 1 && !(imm5 == 00000 && stype == 11).

ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

if Rn == '1101' then SEE "ADD (SP plus register)";

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```
T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>6 5</th>
<th>3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 0 0</td>
<td>Rm</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

**T1 variant**

ADD<>{<q>} <Rd>, <Rn>, <Rm> // Inside IT block
ADDS{<q>} <Rd>, <Rn>, <Rm> // Outside IT block

**Decode for this encoding**

\[d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{setflags} = \neg \text{InITBlock}();\]
\[(\text{shift}_t, \text{shift}_n) = (\text{SRType}_{LSL}, 0);\]

T2

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6</th>
<th>3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>!=1101</td>
<td>Rdn</td>
<td></td>
</tr>
</tbody>
</table>

**T2 variant**

Applies when \(\neg (DN == 1 \&\& Rdn == 101)\).

ADD<>{<q}> <Rdn>, <Rm> // Preferred syntax, Inside IT block
ADD{{<q>}} <Rdn>, <Rdn>, <Rm>

**Decode for this encoding**

\[\text{if} \ (DN:Rdn) == '1101' \text{ or } Rm == '1101' \text{ then SEE "ADD (SP plus register)"};\]
\[d = \text{UInt}(DN:Rdn); \quad n = d; \quad m = \text{UInt}(Rm); \quad \text{setflags} = \text{FALSE}; \quad (\text{shift}_t, \text{shift}_n) = (\text{SRType}_{LSL}, 0);\]
\[\text{if} \ n == 15 \&\& m == 15 \text{ then UNPREDICTABLE};\]
\[\text{if} \ d == 15 \&\& \text{InITBlock()} \&\& \neg \text{LastInITBlock()} \text{ then UNPREDICTABLE};\]

T3

| 15 14 13 12| 11 10 9 8 | 7 6 5 4 | 3 0 | 15 14 12| 11 8 7 6 5 4 | 3 0 |
|-------------|----------|---------|-----|--------|----------|--------|-----|
| 1 1 1 0 1 0 1 1 1 0 0 0 | S | !=1101 | (0) | imm3 | Rd | imm2 | stype | Rm |

**ADD, rotate right with extend variant**

Applies when \(S == 0 \&\& \text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11\).

ADD{{<q>}} {{<q>}} <Rd>, <Rn>, RRX

**ADD, shift or rotate by value variant**

Applies when \(S == 0 \&\& \neg (\text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11)\).

ADD{{<q>}}.W <Rd>, <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ADD{{<q>}}.W <Rd>, <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T2
ADD{{<q>}}.W <Rd>, <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T2
ADD{{<q>}}.W <Rd>, <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T2
**ADDS, rotate right with extend variant**

Applies when $S = 1$ && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11.

ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

**ADDS, shift or rotate by value variant**

Applies when $S = 1$ && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111.

ADDS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1 or T2

ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

**Decode for all variants of this encoding**

if Rd == '1111' && S == '1' then SEE "CMN (register)";
if Rn == '1101' then SEE "ADD (SP plus register)"

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

<
See Standard assembler syntax fields on page F1-7228.

<
See Standard assembler syntax fields on page F1-7228.

<Rdn>
Is the general-purpose source and destination register, encoded in the "DN:Rdn" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.

The assembler language allows <Rdn> to be specified once or twice in the assembler syntax. When used inside an IT block, and <Rd> and <Rm> are in the range R0 to R7, <Rdn> must be specified so that encoding T2 is preferred to encoding T1. In all other cases there is no difference in behavior when <Rdn> is specified once or twice.

<Rd>
For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. If the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of this instruction.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. When used inside an IT block, <Rd> must be specified. When used outside an IT block, <Rd> is optional, and:

- If omitted, this register is the same as <Rn>.
- If present, encoding T1 is preferred to encoding T2.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn>
For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used. If the SP is used, see ADD, ADDS (SP plus register).
For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.
For encoding T3: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see ADD, ADDS (SP plus register).

<\text{Rm}> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
For encoding T1 and T3: is the second general-purpose source register, encoded in the "Rm" field.
For encoding T2: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

<\text{shift}> Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
- LSL when \text{stype} = 00
- LSR when \text{stype} = 01
- ASR when \text{stype} = 10
- ROR when \text{stype} = 11

<\text{amount}> For encoding A1: is the shift amount, in the range 1 to 31 (when <\text{shift}> = LSL or ROR) or 1 to 32 (when <\text{shift}> = LSR or ASR) encoded in the "imm5" field as <\text{amount}> modulo 32.
For encoding T3: is the shift amount, in the range 1 to 31 (when <\text{shift}> = LSL or ROR) or 1 to 32 (when <\text{shift}> = LSR or ASR), encoded in the "imm3:imm2" field as <\text{amount}> modulo 32.

Inside an IT block, if \text{ADD(c)} <\text{Rd}, <\text{Rn}, <\text{Rd}> cannot be assembled using encoding T1, it is assembled using encoding T2 as though \text{ADD(c)} <\text{Rd}, <\text{Rn}> had been written. To prevent this happening, use the .W qualifier.

**Operation for all encodings**

if ConditionPassed() then
   EncodingSpecificOperations();
   shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
   (result, nzcv) = AddWithCarry(R[n], shifted, '0');
   if d == 15 then
      if setflags then
         ALUExceptionReturn(result);
      else
         ALUWritePC(result);
      end
   else
      R[d] = result;
      if setflags then
         PSTATE.<N,Z,C,V> = nzcv;
   end

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.6   ADD, ADDS (register-shifted register)

Add (register-shifted register) adds a register value and a register-shifted register value. It writes the result to the
destination register, and can optionally update the condition flags based on the result.

A1

```
|31| 28|27|26|25|24|23|22|21|20|19|16|15| 12| 11|  8|  7|  6|  5|  4|  3|  0|
|1=111| 0| 0| 0| 0| 1| 0| 0| 1| 0| 0| 0| 0| 0| 0| 0| S| Rn| Rd| Rs| 0| stype| 1| Rm|
```

Flag setting variant

Applies when $S == 1$.

ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>

Not flag setting variant

Applies when $S == 0$.

ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural
Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- `<c>`   See Standard assembler syntax fields on page F1-7228.
- `<q>`   See Standard assembler syntax fields on page F1-7228.
- `<Rd>`  Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>`  Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>`  Is the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can
  have the following values:
  - LSL when stype = 00
  - LSR when stype = 01
  - ASR when stype = 10
  - ROR when stype = 11
- `<Rs>`  Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the
  "Rs" field.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, '0');
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.7 ADD, ADDS (SP plus immediate)

Add to SP (immediate) adds an immediate value to the SP value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. However, when the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.

- The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

```

ADD variant
Applies when S == 0.
ADD{<c>}{<q>} {<Rd>,} SP, #<const>

ADDS variant
Applies when S == 1.
ADDS{<c>}{<q>} {<Rd>,} SP, #<const>

Decode for all variants of this encoding
\[ d = \text{UInt}(Rd); \text{setflags} = (S == '1'); \text{imm32} = \text{A32ExpandImm}(\text{imm12}); \]

T1

```

```

T1 variant
ADD{<c>}{<q>} <Rd>, SP, #<imm8>

Decode for this encoding
\[ d = \text{UInt}(Rd); \text{setflags} = \text{FALSE}; \text{imm32} = \text{ZeroExtend}(\text{imm8}:'00', 32); \]
```
T2

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 0 0 0 0</td>
<td>imm7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**T2 variant**

ADD{<c>}{<q>} {SP,} SP, #<imm7>

*Decode for this encoding*

\[ d = 13; \text{ setflags} = \text{FALSE}; \text{ imm32} = \text{ZeroExtend}(\text{imm7}:'00', 32); \]

T3

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15 14</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0</td>
<td>1 0 1 0 0 0</td>
<td>S</td>
<td>1 1 0 1 0</td>
<td>imm3</td>
<td>Rd</td>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ADD variant**

Applies when \( S = 0 \).

ADD{<c>}.W {<Rd>,} SP, #<const> // <Rd>, <const> can be represented in T1 or T2
ADD{<c>}{<q>} {<Rd>,} SP, #<const>

**ADDS variant**

Applies when \( S = 1 \land Rd \neq 1111 \).

ADDS{<c>}{<q>} {<Rd>,} SP, #<const>

*Decode for all variants of this encoding*

\[
\text{if } Rd = '1111' \land S = '1' \text{ then SEE "CMN (immediate)";}
\]
\[ d = \text{UInt}(Rd); \text{ setflags} = (S = '1'); \text{ imm32} = T32ExpandImm(i:imm3:imm8);
\]
\[ \text{if } d = 15 \land \lnot \text{ setflags} \text{ then UNPREDICTABLE;}
\]

T4

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15 14</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0</td>
<td>1 1 0 0 0 0 0 0 1 1 0 1 0</td>
<td>imm3</td>
<td>Rd</td>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**T4 variant**

ADD{<c>}{<q>} {<Rd>,} SP, #<imm12> // <imm12> cannot be represented in T1, T2, or T3
ADDW{<c>}{<q>} {<Rd>,} SP, #<imm12> // <imm12> can be represented in T1, T2, or T3

*Decode for this encoding*

\[ d = \text{UInt}(Rd); \text{ setflags} = \text{FALSE}; \text{ imm32} = \text{ZeroExtend}(i:imm3:imm8, 32);
\]
\[ \text{if } d = 15 \text{ then UNPREDICTABLE;}
\]
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

SP, is the stack pointer.

<imm7> Is the unsigned immediate, a multiple of 4, in the range 0 to 508, encoded in the "imm7" field as <imm7>/4.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field.

For encoding T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

<imm8> Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4.

<imm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

<const> For encoding A1: an immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values.

For encoding T3: an immediate value. See Modified immediate constants in T32 instructions on page F1-7242 for the range of values.

Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations();
   (result, nzcv) = AddWithCarry(R[13], imm32, '0');
   if d == 15 then // Can only occur for A32 encoding
      if setflags then
         ALUExceptionReturn(result);
      else
         ALUWritePC(result);
   else
      R[d] = result;
      if setflags then
         PSTATE.<N,Z,C,V> = nzcv;
}
F5.1.8   ADD, ADDS (SP plus register)

Add to SP (register) adds an optionally-shifted register value to the SP value, and writes the result to the destination register.

If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for \(<Rd>\) identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- The ADDS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_\(<current\_mode>\).
  - The PE checks SPSR_\(<current\_mode>\) for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

```
<table>
<thead>
<tr>
<th>31 28 27 26 25 24 23 22 21 20 19 18 17 16 15</th>
<th>12 11</th>
<th>7  6  5  4  3  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>(l=1111)</td>
<td>(Rd)</td>
<td>imm5</td>
</tr>
</tbody>
</table>
| cond
```

ADD, rotate right with extend variant

Applies when \(S == 0 \&\& \text{imm5} == 00000 \&\& \text{stype} == 11\).

ADD\{<c>\}{<q>} \{<Rd>,\} SP, \(<Rm>\), RRX

ADD, shift or rotate by value variant

Applies when \(S == 0 \&\& !(\text{imm5} == 00000 \&\& \text{stype} == 11)\).

ADD\{<c>\}{<q>} \{<Rd>,\} SP, \(<Rm>\), \{, \(<shift>\) \#<amount>\}

ADDS, rotate right with extend variant

Applies when \(S == 1 \&\& \text{imm5} == 00000 \&\& \text{stype} == 11\).

ADDS\{<c>\}{<q>} \{<Rd>,\} SP, \(<Rm>\), RRX

ADDS, shift or rotate by value variant

Applies when \(S == 1 \&\& !(\text{imm5} == 00000 \&\& \text{stype} == 11)\).

ADDS\{<c>\}{<q>} \{<Rd>,\} SP, \(<Rm>\), \{, \(<shift>\) \#<amount>\}

Decode for all variants of this encoding

\(d = \text{UInt}(Rd); m = \text{UInt}(Rm); \text{setflags} = (S == '1'); (\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{imm5});\)
T1

```
| 15 14 13 12| 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
| 0 1 0 0 0 0 | 1 0 1 | 1 1 1 1 |
```

**T1 variant**

ADD\{<c>\}{<q>} {<Rdm>}, SP, <Rdm>

**Decode for this encoding**

\[d = \text{UInt}(\text{DM:Rdm}); m = \text{UInt}(\text{DM:Rdm}); \text{setflags} = \text{FALSE}; (\text{shift}_t, \text{shift}_n) = (\text{SRType}_{LSL}, 0);\]
\[\text{if } d = 15 \&\& \text{InITBlock()} \&\& \text{LastInITBlock()} \text{ then UNPREDICTABLE; }\]

T2

```
| 15 14 13 12| 11 10 9 8 | 7 6 3 2 1 0 |
| 0 1 0 0 0 0 | 1 0 1 | 1 1 1 1 |
```

**T2 variant**

ADD\{<c>\}{<q>} {SP,} SP, <Rm>

**Decode for this encoding**

\[\text{if } Rm = '1101' \text{ then SEE "encoding T1"; }\]
\[d = 13; m = \text{UInt}(Rm); \text{setflags} = \text{FALSE}; (\text{shift}_t, \text{shift}_n) = (\text{SRType}_{LSL}, 0);\]

T3

```
| 15 14 13 12| 11 10 9 8 | 7 6 5 4 | 3 2 1 0 | 15 14 12| 11 8 7 6 5 4 | 3 0 |
| 1 1 1 0 1 0 1 1 0 0 0 | S 1 1 0 1 | 0 | imm3 | Rd | imm2 | stype | Rm |
```

**ADD, rotate right with extend variant**

Applies when \(S = 0\) \&\& \(\text{imm}3 = 000\) \&\& \(\text{imm}2 = 00 \&\& \text{stype} = 11\).

ADD\{<c>\}{<q>} {<Rd>}, SP, <Rm>, RRX

**ADD, shift or rotate by value variant**

Applies when \(S = 0 \&\& !(\text{imm}3 = 000 \&\& \text{imm}2 = 00 \&\& \text{stype} = 11)\).

ADD\{<c>\}.W {<Rd>}, SP, <Rm> // {<Rd>}, <Rm> can be represented in T1 or T2

ADD\{<c>\}{<q>} {<Rd>}, SP, <Rm> {<Rd> \&\& shift \#amount}

**ADDS, rotate right with extend variant**

Applies when \(S = 1\) \&\& \(\text{imm}3 = 000\) \&\& \(\text{Rd} = 1111\) \&\& \(\text{imm}2 = 00 \&\& \text{stype} = 11\).

ADDS\{<c>\}{<q>} {<Rd>}, SP, <Rm>, RRX
**ADDS, shift or rotate by value variant**

Applies when $S = 1$ && $(\text{imm3} == 000$ && $\text{imm2} == 00$ && $\text{stype} == 11)$ && $Rd != 1111$.

`ADDS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}`

**Decode for all variants of this encoding**

$$
\begin{align*}
\text{if } Rd == '1111' \&\& S == '1' \text{ then SEE "CMN (register)";} \\
d = \text{UInt}(Rd); \ m = \text{UInt}(Rm); \ setflags = (S == '1'); \\
(\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{imm3:imm2}); \\
\text{if } (d == 15 \&\& \neg setflags) \text{ || } m == 15 \text{ then UNPREDICTABLE; } // \text{Armv8-A removes UNPREDICTABLE for R13}
\end{align*}
$$

**Notes for all encodings**

For more information about the CONstrained UNPREDICTable behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>`: See Standard assembler syntax fields on page F1-7228.
- `<q>`: See Standard assembler syntax fields on page F1-7228.
- `SP`: Is the stack pointer.
- `<Rdm>`: Is the general-purpose destination and second source register, encoded in the "Rdm" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- `<Rd>`: For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used:
  - For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
  - For the ADDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

- `<Rm>`: For encoding A1 and T2: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T3: is the second general-purpose source register, encoded in the "Rm" field.

- `<shift>`: Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
  - LSL: when stype = 00
  - LSR: when stype = 01
  - ASR: when stype = 10
  - ROR: when stype = 11

- `<amount>`: For encoding A1: is the shift amount, in the range 1 to 31 (when `<shift>` = LSL or ROR) or 1 to 32 (when `<shift>` = LSR or ASR) encoded in the "imm5" field as `<amount>` modulo 32.

For encoding T3: is the shift amount, in the range 1 to 31 (when `<shift>` = LSL or ROR) or 1 to 32 (when `<shift>` = LSR or ASR), encoded in the "imm3:imm2" field as `<amount>` modulo 32.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[13], shifted, '0');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
F5.1.9  ADD (immediate, to PC)

Add to PC adds an immediate value to the Align(PC, 4) value to form a PC-relative address, and writes the result

to the destination register. Arm recommends that, where possible, software avoids using this alias

This instruction is a pseudo-instruction of the ADR instruction. This means that:

- The encodings in this description are named to match the encodings of ADR.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of ADR gives the operational pseudocode for this instruction.

A1

\[
\begin{array}{ccccccccccccccc}
\end{array}
\]

A1 variant

ADD\{<c>\}{<q>} <Rd>, PC, #<const>

is equivalent to

ADR\{<c>\}{<q>} <Rd>, <label>

and is never the preferred disassembly.

T1

\[
\begin{array}{ccccccccccccccc}
1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 10 & 11 & 12 & 13 & 14 & 15 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & \text{imm8} & \text{Rd} & \text{cond} & 31 & 28 & 27 & 26 & 25 & 24 & 23 & 22 & 21 & 20 & 19 & 18 & 17 & 16 & 15 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & \text{imm8} & \text{Rd} & \text{cond} & 31 & 28 & 27 & 26 & 25 & 24 & 23 & 22 & 21 & 20 & 19 & 18 & 17 & 16 & 15 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & \text{imm8} & \text{Rd} & \text{cond}
\end{array}
\]

T1 variant

ADD\{<c>\}{<q>} <Rd>, PC, #<imm8>

is equivalent to

ADR\{<c>\}{<q>} <Rd>, <label>

and is never the preferred disassembly.

T3

\[
\begin{array}{ccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 10 & 11 & 12 & 13 & 14 & 15 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & \text{imm8} & \text{Rd} & \text{cond} & 31 & 28 & 27 & 26 & 25 & 24 & 23 & 22 & 21 & 20 & 19 & 18 & 17 & 16 & 15 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & \text{imm8} & \text{Rd} & \text{cond} & 31 & 28 & 27 & 26 & 25 & 24 & 23 & 22 & 21 & 20 & 19 & 18 & 17 & 16 & 15 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & \text{imm8} & \text{Rd} & \text{cond}
\end{array}
\]

T3 variant

ADD\{<c>\}{<q>} <Rd>, PC, #<imm12> // <Rd>, <imm12> can be represented in T1

is equivalent to

ADR\{<c>\}{<q>} <Rd>, <label>

and is never the preferred disassembly.
ADD{<c>}{<q>} <Rd>, PC, #<imm12>

is equivalent to

ADR{<c>}{<q>} <Rd>, <label>

and is never the preferred disassembly.

**Assembler symbols**

- **<c>**  
  See [Standard assembler syntax fields on page F1-7228](#).

- **<q>**  
  See [Standard assembler syntax fields on page F1-7228](#).

- **<Rd>**  
  For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133](#).

  For encoding T1 and T3: is the general-purpose destination register, encoded in the "Rd" field.

- **<label>**  
  For encoding A1: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label.

  If the offset is zero or positive, encoding A1 is used, with imm32 equal to the offset.

  If the offset is negative, encoding A2 is used, with imm32 equal to the size of the offset. That is, the use of encoding A2 indicates that the required offset is minus the value of imm32.

  Permitted values of the size of the offset are any of the constants described in [Modified immediate constants in A32 instructions on page F1-7244](#).

  For encoding T1: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.

  For encoding T3: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label.

  If the offset is zero or positive, encoding T3 is used, with imm32 equal to the offset.

  If the offset is negative, encoding T2 is used, with imm32 equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of imm32.

  Permitted values of the size of the offset are 0-4095.

- **<imm8>**  
  Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4.

- **<imm12>**  
  Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

- **<const>**  
  An immediate value. See [Modified immediate constants in A32 instructions on page F1-7244](#) for the range of values.

**Operation for all encodings**

The description of ADR gives the operational pseudocode for this instruction.
F5.1.10   ADR

Form PC-relative address adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.

This instruction is used by the pseudo-instructions ADD (immediate, to PC) and SUB (immediate, from PC). The pseudo-instruction is never the preferred disassembly.

A1

```
<table>
<thead>
<tr>
<th>31 28</th>
<th>26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 12</th>
<th>11</th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111</td>
<td>0 0 1 0 1 0 0 1 1 1 1</td>
<td>Rd</td>
<td>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**A1 variant**

ADR{(c)}{q} <Rd>, <label>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad \text{imm32} = \text{A32ExpandImm}(\text{imm12}); \quad \text{add} = \text{TRUE};
\]

A2

```
<table>
<thead>
<tr>
<th>31 28</th>
<th>26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 12</th>
<th>11</th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111</td>
<td>0 0 1 0 0 1 0 0 1 1 1 1</td>
<td>Rd</td>
<td>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**A2 variant**

ADR{(c)}{q} <Rd>, <label>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad \text{imm32} = \text{A32ExpandImm}(\text{imm12}); \quad \text{add} = \text{FALSE};
\]

T1

```
<table>
<thead>
<tr>
<th>15 14 13</th>
<th>12</th>
<th>11 10</th>
<th>8 7</th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0</td>
<td>Rd</td>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**T1 variant**

ADR{(c)}{q} <Rd>, <label>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad \text{imm32} = \text{ZeroExtend}(\text{imm8}:'00', 32); \quad \text{add} = \text{TRUE};
\]
**T32 and A32 Base Instruction Set Instruction Descriptions**

**F5.1 Alphabetical list of T32 and A32 base instruction set instructions**

---

### T2

![Instruction Format](image)

**T2 variant**

ADR{<c>}{<q>} <Rd>, <label>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad \text{imm32} = \text{ZeroExtend}(i:imm3:imm8, 32); \quad \text{add} = \text{FALSE};
\]

if \( d == 15 \) then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

### T3

![Instruction Format](image)

**T3 variant**

ADR{<c>}.W <Rd>, <label> // <Rd>, <label> can be presented in T1

ADR{<c>}{<q>} <Rd>, <label>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad \text{imm32} = \text{ZeroExtend}(i:imm3:imm8, 32); \quad \text{add} = \text{TRUE};
\]

if \( d == 15 \) then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

### Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

### Alias conditions

<table>
<thead>
<tr>
<th>Alias or pseudo-instruction</th>
<th>of variant</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD (immediate, to PC)</td>
<td>-</td>
<td>Never</td>
</tr>
<tr>
<td>SUB (immediate, from PC)</td>
<td>T2</td>
<td>imm3:imm8 == '000000000000'</td>
</tr>
<tr>
<td>SUB (immediate, from PC)</td>
<td>A2</td>
<td>imm12 == '000000000000'</td>
</tr>
</tbody>
</table>

### Assembler symbols

- `<c>` See *Standard assembler syntax fields* on page F1-7228.
- `<q>` See *Standard assembler syntax fields* on page F1-7228.
- `<Rd>` For encoding A1 and A2: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-7133.
For encoding T1, T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

For encoding A1 and A2: the label of an instruction or literal data item whose address is to be loaded into \(<Rd>\). The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label.

If the offset is zero or positive, encoding A1 is used, with imm32 equal to the offset.

If the offset is negative, encoding A2 is used, with imm32 equal to the size of the offset. That is, the use of encoding A2 indicates that the required offset is minus the value of imm32.

Permitted values of the size of the offset are any of the constants described in Modified immediate constants in A32 instructions on page F1-7244.

For encoding T1: the label of an instruction or literal data item whose address is to be loaded into \(<Rd>\). The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.

For encoding T2 and T3: the label of an instruction or literal data item whose address is to be loaded into \(<Rd>\). The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label.

If the offset is zero or positive, encoding T3 is used, with imm32 equal to the offset.

If the offset is negative, encoding T2 is used, with imm32 equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of imm32.

Permitted values of the size of the offset are 0-4095.

The instruction aliases permit the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax on page F2-7257.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    result = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    if d == 15 then          // Can only occur for A32 encodings
        ALUWritePC(result);
    else
        R[d] = result;
```
F5.1.11   **AND, ANDS (immediate)**

Bitwise AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the ANDS variant of the instruction updates the condition flags based on the result.

The field descriptions for \(<Rd>\) identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The AND variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-7133.
- The ANDS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_\(_{<current\_mode>}\).
  - The PE checks SPSR_\(_{<current\_mode>}\) for an illegal return event. See *Illegal return events from AArch32 state* on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

**A1**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1=111</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**AND variant**

Applies when \(S == 0\).

\(\text{AND}\{<c>\}{<q>} \{<Rd>,\} <Rn>, \#<\text{const}>\)

**ANDS variant**

Applies when \(S == 1\).

\(\text{ANDS}\{<c>\}{<q>} \{<Rd>,\} <Rn>, \#<\text{const}>\)

**Decode for all variants of this encoding**

\[d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad \text{setflags} = (S == '1');\]

\[(\text{imm32, carry}) = A32\text{ExpandImm}\_C(\text{imm12, PSTATE.C});\]

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 14 | 12 | 11 | 8 | 7 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | S | Rn | 0 | imm3 | Rd | imm8 |

**AND variant**

Applies when \(S == 0\).

\(\text{AND}\{<c>\}{<q>} \{<Rd>,\} <Rn>, \#<\text{const}>\)
**ANDS variant**
Applies when \( S = 1 \) \&\& \( Rd \neq 1111 \).

\[ \text{ANDS\{<c>\}{<q>} \{<Rd>,\} <Rn>, #<const>} \]

**Decode for all variants of this encoding**

\[
\text{if Rd == '1111' \&\& S == '1' then SEE "TST (immediate)";} \\
\text{d = \text{UInt}(Rd); n = \text{UInt}(Rn); \text{setflags} = (S == '1');} \\
\text{(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);} \\
\text{if (d == 15 \&\& !setflags) || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13} \\
\]

**Notes for all encodings**
For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

\(<c>\) See Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<Rd>\) For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as \(<Rn>\). Arm deprecates using the PC as the destination register, but if the PC is used:

- For the AND variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- For the ANDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_\(<\text{current}\_\text{mode})>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as \(<Rn>\).

\(<Rn>\) For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

\(<\text{const}>\) For encoding A1: an immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values.
For encoding T1: an immediate value. See Modified immediate constants in T32 instructions on page F1-7242 for the range of values.

**Operation for all encodings**

\[
\text{if ConditionPassed() then} \\
\text{EncodingSpecificOperations();} \\
\text{result = R[n] AND imm32;} \\
\text{if d == 15 then} \quad /\quad \text{Can only occur for A32 encoding} \\
\text{if setflags then} \\
\text{ALUExceptionReturn(result);} \\
\text{else} \\
\text{ALUWritePC(result);} \\
\text{else} \\
\text{R[d] = result;} \\
\text{if setflags then} \\
\text{PSTATE.N = result<31>;} \\
\text{PSTATE.Z = IsZeroBit(result);} \\
\]

ARM DD1 04871.a Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. F5-7475
ID081822 Non-Confidential
PSTATE.C = carry;
// PSTATE.V unchanged

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.12   AND, ANDS (register)

Bitwise AND (register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ANDS variant of the instruction updates the condition flags based on the result.

The field descriptions for \(<Rd>\) identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The AND variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.*
- The ANDS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state on page G1-8952.*
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**AND, rotate right with extend variant**

Applies when $S == 0$ 
and $imm5 == 00000$ 
and $stype == 11$.

**AND{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX**

**AND, shift or rotate by value variant**

Applies when $S == 0$ 
and $(imm5 == 00000$ 
and $stype == 11$).

**AND{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}**

**ANDS, rotate right with extend variant**

Applies when $S == 1$ 
and $imm5 == 00000$ 
and $stype == 11$.

**ANDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX**

**ANDS, shift or rotate by value variant**

Applies when $S == 1$ 
and $(imm5 == 00000$ 
and $stype == 11$).

**ANDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}**

**Decode for all variants of this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{setflags} = (S == '1');
\]

\[
(shift\_t, shift\_n) = \text{DecodeImmShift}(stype, imm5);
\]
T1

AND{<q>}{<Rdn>}, <Rdn>, <Rm> // Inside IT block
ANDS{<q>}{<Rdn>}, <Rdn>, <Rm> // Outside IT block

Decode for this encoding

d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);

T2

AND, rotate right with extend variant

Applies when S == 0 && imm3 == 000 && imm2 == 00 && stype == 11.

AND{<c>}{<q>}{<Rd>,} <Rn>, <Rm>, RRX

AND, shift or rotate by value variant

Applies when S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11).

AND{<c>}{<q>}{<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

ANDS, rotate right with extend variant

Applies when S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11.

ANDS{<c>}{<q>}{<Rd>,} <Rn>, <Rm>, RRX

ANDS, shift or rotate by value variant

Applies when S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111.

ANDS.W{<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ANDS{<c>}{<q>}{<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

if Rd == '1111' && S == '1' then SEE "TST (register)";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

<Rdn>
Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd>
For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the AND variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- For the ANDS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn>
For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm>
For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift>
Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- LSL when stype = 00
- LSR when stype = 01
- ASR when stype = 10
- ROR when stype = 11

<amount>
For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

In T32 assembly:

- Outside an IT block, if ANDS <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ANDS <Rd>, <Rn> had been written.

- Inside an IT block, if AND<< <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though AND<< <Rd>, <Rn> had been written.

To prevent either of these happening, use the .W qualifier.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    if d == 15 then  // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
```c
else
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.13 AND, ANDS (register-shifted register)

Bitwise AND (register-shifted register) performs a bitwise AND of a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

A1

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 8  | 7  | 6  | 5  | 4  | 3  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| S  | Rn | Rd | Rs | 0  | 0  | 0  | 0  | 1111 |

Flag setting variant

Applies when \( S = 1 \).

\[
\text{ANDS\{<c>\}{<q>} {<Rd>,} <Rn>, <Rm>, <\text{shift}> <Rs>}
\]

Not flag setting variant

Applies when \( S = 0 \).

\[
\text{AND\{<c>\}{<q>} {<Rd>,} <Rn>, <Rm>, <\text{shift}> <Rs>}
\]

Decode for all variants of this encoding

\[
\begin{align*}
d &= \text{UInt}(Rd); \\
n &= \text{UInt}(Rn); \\
m &= \text{UInt}(Rm); \\
s &= \text{UInt}(Rs);
\end{align*}
\]

\[
\text{setflags} = (S == '1'); \\
\text{shift_t} = \text{DecodeRegShift(stype)};
\]

\[
\text{if } d == 15 \text{ || } n == 15 \text{ || } m == 15 \text{ || } s == 15 \text{ then UNPREDICTABLE;}
\]

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\)  See Standard assembler syntax fields on page F1-7228.

\(<q>\)  See Standard assembler syntax fields on page F1-7228.

\(<Rd>\)  Is the general-purpose destination register, encoded in the "Rd" field.

\(<Rn>\)  Is the first general-purpose source register, encoded in the "Rn" field.

\(<Rm>\)  Is the second general-purpose source register, encoded in the "Rm" field.

\(<\text{shift}>\)  Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

\[
\begin{align*}
\text{LSL} & \quad \text{when stype} = 00 \\
\text{LSR} & \quad \text{when stype} = 01 \\
\text{ASR} & \quad \text{when stype} = 10 \\
\text{ROR} & \quad \text{when stype} = 11
\end{align*}
\]

\(<Rs>\)  Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        /* PSTATE.V unchanged */

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.14   ASR (immediate)

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

A1

\[
\begin{array}{cccccccccccc}
\hline
\text{cond} & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
S & Rd & imm5 & Rm & Rd & \hline
\end{array}
\]

**MOV, shift or rotate by value variant**

\[
\text{ASR\{}<c>\text{\{}<q>\text{\{}<Rd>,} <Rm>, #<imm>}
\]

is equivalent to

\[
\text{MOV\{}<c>\text{\{}<q>\text{\{}<Rd>,} <Rm>, ASR #<imm>}
\]

and is always the preferred disassembly.

T2

\[
\begin{array}{cccccccccccc}
\hline
\text{op} & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
Rm & Rd & imm5 & Rd & \hline
\end{array}
\]

**T2 variant**

\[
\text{ASR}\{}<c>\text{\{}<p>\text{\{}<Rd>,} <Rm>, #<imm> // Inside IT block}
\]

is equivalent to

\[
\text{MOV}\{}<c>\text{\{}<p>\text{\{}<Rd>,} <Rm>, ASR #<imm>}
\]

and is the preferred disassembly when InITBlock().

T3

\[
\begin{array}{cccccccccccc}
\hline
\text{S} & Rd & imm3 & imm2 & 1 & 0 & Rm & \hline
\end{array}
\]

**MOV, shift or rotate by value variant**

\[
\text{ASR}\{}<c>\text{\{}W\text{\{}<Rd>,} <Rm>, #<imm> // Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2}
\]

is equivalent to

\[
\text{MOV}\{}<c>\text{\{}<p>\text{\{}<Rd>,} <Rm>, ASR #<imm>}
\]

and is always the preferred disassembly.

\[
\text{ASR}\{}<c>\text{\{}<p>\text{\{}<Rd>,} <Rm>, #<imm>}
\]
is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>

and is always the preferred disassembly.

**Assembler symbols**

**<c>** See *Standard assembler syntax fields* on page F1-7228.

**<q>** See *Standard assembler syntax fields* on page F1-7228.

**<Rd>** For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-7133.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

**<Rm>** For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

**<imm>** For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.

For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

**Operation for all encodings**

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.
F5.1.15 ASR (register)

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

**A1**

Not flag setting variant

ASR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>

and is always the preferred disassembly.

**T1**

Arithmetic shift right variant

ASR{<c>}{<q>} {<Rdm>,} <Rdm>, <Rs> // Inside IT block

is equivalent to

MOV{<c>}{<q>} <Rdm>, <Rdm>, ASR <Rs>

and is the preferred disassembly when InITBlock().

**T2**

Not flag setting variant

ASR{<c>}{<W>} {<Rd>,} <Rm>, <Rs> // Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
and is always the preferred disassembly.

\[ \text{ASR} \{<c>\} \{<q>\} \{<Rd>,\} \{<Rm>,\} \{<Rs>\} \]

is equivalent to

\[ \text{MOV} \{<c>\} \{<q>\} \{<Rd>,\} \{<Rm>,\} \{\text{ASR} <Rs>\} \]

and is always the preferred disassembly.

**Assembler symbols**

\(<c>\) See [Standard assembler syntax fields on page F1-7228](#).

\(<q>\) See [Standard assembler syntax fields on page F1-7228](#).

\(<Rdm>\) Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

\(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.

\(<Rm>\) Is the first general-purpose source register, encoded in the "Rm" field.

\(<Rs>\) Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

**Operation for all encodings**

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.
F5.1.16 ASRS (immediate)

Arithmetic Shift Right, setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for \(<Rd>\) identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

A1

\[
\begin{array}{cccccccccccc}
\end{array}
\]

\begin{array}{ccccccccc}
| conds & S & & & & & & & & Rd & imm5 & 1 & 0 & 0 & Rm |
\end{array}

MOV, shift or rotate by value variant

\(\text{ASRS}\{<c>\}\{<q>\} \{<Rd>,\} \{<Rm>, \#<imm>\}

is equivalent to

\(\text{MOVS}\{<c>\}\{<q>\} \{<Rd>, \{<Rm>, \ ASR \#<imm>\}

and is always the preferred disassembly.

T2

\[
\begin{array}{cccccccccccc}
| & 15 & 14 & 13 & 12 | 11 & 10 | 6 & 5 & 3 & 2 & 0 |
\end{array}
\]

\[ \begin{array}{cccc}
0 & 0 & 0 & 1 & 0 & \text{imm5} & Rm & Rd \\
\end{array} \]

T2 variant

\(\text{ASRS}\{<q>\} \{<Rd>,\} \{<Rm>, \#<imm> / / \text{Outside IT block}

is equivalent to

\(\text{MOVS}\{<q>\} \{<Rd>, \{<Rm>, \ ASR \#<imm>\}

and is the preferred disassembly when !\text{InITBlock}().\)
T3

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>15 14 12 11</th>
<th>8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 0 1</td>
<td>0 0 1 0 1 1 1 1</td>
<td>[0]</td>
<td>imm3</td>
</tr>
</tbody>
</table>

**MOVS, shift or rotate by value variant**

ASRS.W {<Rd>,} <Rm>, #<imm> // Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>

and is always the preferred disassembly.

ASRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>

and is always the preferred disassembly.

**Assembler symbols**

- `<><>` See *Standard assembler syntax fields on page F1-7228*.
- `<><>` See *Standard assembler syntax fields on page F1-7228*.
- `<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.
  
  For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
  
  For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.
- `<imm>` For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as `<imm>` modulo 32.
  
  For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as `<imm>` modulo 32.

**Operation for all encodings**

The description of **MOV, MOVS (register)** gives the operational pseudocode for this instruction.
F5.1.17 ASRS (register)

Arithmetic Shift Right, setting flags (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

A1

Flag setting variant

ASRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>  
is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>  
and is always the preferred disassembly.

T1

Arithmetic shift right variant

ASRS{<q>} {<Rdm>,} <Rdm>, <Rs>  // Outside IT block  
is equivalent to

MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs>  
and is the preferred disassembly when !InITBlock().

T2

Flag setting variant

ASRS.W {<Rd>,} <Rm>, <Rs>  // Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1  
is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
and is always the preferred disassembly.

\[ \text{ASRS} \{<c>\} \{<q>\} \{<Rd>,\} \{<Rm>,\} \{<Rs>\} \]

is equivalent to

\[ \text{MOVS} \{<c>\} \{<q>\} \{<Rd>,\} \{<Rm>,\} \text{ ASR } <Rs> \]

and is always the preferred disassembly.

**Assembler symbols**

\(<c>\) See *Standard assembler syntax fields* on page F1-7228.
\(<q>\) See *Standard assembler syntax fields* on page F1-7228.
\(<Rdm>\) Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
\(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.
\(<Rm>\) Is the first general-purpose source register, encoded in the "Rm" field.
\(<Rs>\) Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

**Operation for all encodings**

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.
Branch causes a branch to a target address.

**A1**

```
+----------+----------+----------+----------+----------+----------+----------+----------+------------------------------------------+
| 31       | 28 27    | 26 25    | 24 23    | 22 21    | 20 19    | 18 17    | 16 15    | imm32 = SignExtend(imm24:'00', 32);     |
| !=1111    | 1 0      | 1 0      |          |          |          |          |          |                                          |
| cond      |          |          |          |          |          |          |          |                                          |
```

**A1 variant**

`B{<c>}{<q>} <label>` // Not permitted in IT block

**Decode for this encoding**

- if cond == '1110' then SEE "UDF";
- if cond == '1111' then SEE "SVC";
- `imm32 = SignExtend(imm8:'0', 32);
- if InITBlock() then UNPREDICTABLE;

**T1**

```
+----------+----------+----------+----------+----------+----------+----------+----------+------------------------------------------+
| 15 14 13 12 | 11 8 7   | 0        |
| 1 1 0      | !=111x   |imm8      |
| cond       |          |          |          |          |          |          |          |                                          |
```

**T1 variant**

`B{<c>}{<q>} <label>` // Outside or last in IT block

**Decode for this encoding**

- if cond == '1110' then SEE "UDF";
- if cond == '1111' then SEE "SVC";
- `imm32 = SignExtend(imm11:'0', 32);
- if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
T3

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>6 5</th>
<th>0</th>
<th>15 14 13 12</th>
<th>11 10</th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0</td>
<td>S</td>
<td>!=111x</td>
<td>imm6</td>
<td>1 0</td>
<td>J1</td>
<td>0</td>
<td>J2</td>
</tr>
</tbody>
</table>

**cond**

**T3 variant**

B<c>.W <label> // Not permitted in IT block, and <label> can be represented in T1
B<c>{<q>} <label> // Not permitted in IT block

**Decode for this encoding**

if cond<3:1> == '111' then SEE "Related encodings";
imm32 = SignExtend(S:J2:J1:imm6:imm11:'0', 32);
if InITBlock() then UNPREDICTABLE;

T4

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th></th>
<th>0</th>
<th>15 14 13 12</th>
<th>11 10</th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0</td>
<td>S</td>
<td>imm10</td>
<td>1 0</td>
<td>J1</td>
<td>1</td>
<td>J2</td>
<td>imm11</td>
</tr>
</tbody>
</table>

**T4 variant**

B{<c>}.W <label> // <label> can be represented in T2
B{<c>{<q>} <label>

**Decode for this encoding**

I1 = NOT(J1 EOR S); I2 = NOT(J2 EOR S); imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Related encodings: Branches and miscellaneous control on page F3-7343.

**Assembler symbols**

**<c>**

For encoding A1, T2 and T4: see Standard assembler syntax fields on page F1-7228.
For encoding T1: see Standard assembler syntax fields on page F1-7228. Must not be AL or omitted.
For encoding T3: see Standard assembler syntax fields on page F1-7228. <c> must not be AL or omitted.

**<q>**

See Standard assembler syntax fields on page F1-7228.

**<label>**

For encoding A1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the 8 instruction to this label, then selects an encoding that sets imm32 to that offset.
Permitted offsets are multiples of 4 in the range –33554432 to 33554428.
For encoding T1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the 8 instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range –256 to 254.
For encoding T2: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the \texttt{B} instruction to this label, then selects an encoding that sets \texttt{imm32} to that offset. Permitted offsets are even numbers in the range –2048 to 2046.

For encoding T3: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the \texttt{B} instruction to this label, then selects an encoding that sets \texttt{imm32} to that offset. Permitted offsets are even numbers in the range –1048576 to 1048574.

For encoding T4: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the \texttt{B} instruction to this label, then selects an encoding that sets \texttt{imm32} to that offset. Permitted offsets are even numbers in the range –16777216 to 16777214.

\textbf{Operation for all encodings}

\begin{verbatim}
if ConditionPassed() then
    EncodingSpecificOperations();
    BranchWritePC(PC + imm32, BranchType_DIR);
\end{verbatim}
F5.1.19 BFC

Bit Field Clear clears any number of adjacent bits at any position in a register, without affecting the other bits in the register.

A1

\[
\begin{array}{cccccccccccc}
| & | & | & | & | & | & | & | & | & | & | \\
|31|28|27|26|25|24|23|22|21|20|15|12|11|10|9|8|7|6|5|4|3|2|1|0| \\
| 0 | 1 | 1 | 1 | 1 | 0 | msb & Rd & lsb & 0 & 0 & 1 & 1 & 1 & 1 |
\end{array}
\]

**A1 variant**

\[
BFC\{c\}\{q\} <Rd>, #<lsb>, #<width>
\]

**Decode for this encoding**

\[
d = UInt(Rd); \ms = UInt(msb); \ls = UInt(lsb);
if \(d = 15\) then UNPREDICTABLE;
if \(\ms < \ls\) then UNPREDICTABLE;
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \(\ms < \ls\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as \texttt{NOP}.
- The value in the destination register is UNKNOWN.

T1

\[
\begin{array}{cccccccccccc}
|15|14|13|12|11|10|9|8|7|6|5|4|3|2|1|0|15|14|12|11|8|7|6|5|4|0| \\
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | imm3 & Rd & imm2 & 0 & msb |
\end{array}
\]

**T1 variant**

\[
BFC\{c\}\{q\} <Rd>, #<lsb>, #<width>
\]

**Decode for this encoding**

\[
d = UInt(Rd); \ms = UInt(msb); \ls = UInt(imm3:imm2);
if \(d = 15\) then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
if \(\ms < \ls\) then UNPREDICTABLE;
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \(\ms < \ls\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as \texttt{NOP}.
- The value in the destination register is UNKNOWN.
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

>
See Standard assembler syntax fields on page F1-7228.

Rd
Is the general-purpose destination register, encoded in the "Rd" field.

<lsb>
For encoding A1: is the least significant bit to be cleared, in the range 0 to 31, encoded in the "lsb" field.
For encoding T1: is the least significant bit that is to be cleared, in the range 0 to 31, encoded in the "imm3:imm2" field.

<width>
Is the number of bits to be cleared, in the range 1 to 32-<lsb>, encoded in the "msb" field as <lsb>+(<width>-1).

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  R[d]<msb:lsbit> = Replicate('0', (msb-<lsbit>);+1);
  // Other bits of R[d] are unchanged

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.20   BFI

Bit Field Insert copies any number of low order bits from a register into the same number of adjacent bits at any position in the destination register.

A1

| [31 28|27 26 25 24|23 22 21 20] | 16|15 | 12|11 | 7 6 5 4 | 3 | 0 |
|-----------------|--------|--------|--------|--------|
| !=1111 0 1 1 1 1 0 | msb | Rd | lsb | 0 0 1 | !=1111 |
| cond | Rd |

A1 variant

BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

Decode for this encoding

if Rn == '1111' then SEE "BFC";
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsb = UInt(imm);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if msbit < lsb then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If msbit < lsb, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The value in the destination register is UNKNOWN.

T1

<table>
<thead>
<tr>
<th>[15 14 13 12</th>
<th>11 10 9</th>
<th>8 7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0</td>
<td>[0]</td>
<td>1 1 0 1 0</td>
<td>!=1111</td>
<td>0</td>
</tr>
<tr>
<td>Rd</td>
<td>imm2</td>
<td>msb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T1 variant

BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

Decode for this encoding

if Rn == '1111' then SEE "BFC";
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsb = UInt(imm3:imm2);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if msbit < lsb then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If msbit < lsb, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The value in the destination register is UNKNOWN.
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

<
See Standard assembler syntax fields on page F1-7228.

<Rd>
Is the general-purpose destination register, encoded in the "Rd" field.

<Rn>
Is the general-purpose source register, encoded in the "Rn" field.

<lsb>
For encoding A1: is the least significant destination bit, in the range 0 to 31, encoded in the "lsb" field.
For encoding T1: is the least significant destination bit, in the range 0 to 31, encoded in the "imm3:imm2" field.

<width>
Is the number of bits to be copied, in the range 1 to 32-<lsb>, encoded in the "msb" field as <lsb>+<width>-1.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<msbit:lsbit> = R[n]<msbit-1:0>;
    // Other bits of R[d] are unchanged

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.21   BIC, BICS (immediate)

Bitwise Bit Clear (immediate) performs a bitwise AND of a register value and the complement of an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the BICS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The BIC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.*
- The BICS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state on page G1-8952.*
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

```
| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | | | 0 |
-----
S   Rn  Rd  imm12
cond
```

**BIC variant**

Applies when S == 0.

BIC{<c>}{<q>} {<Rd>,} <Rn>, #<const>

**BICS variant**

Applies when S == 1.

BICS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

**Decode for all variants of this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad \text{setflags} = (S == '1');
\]

\[
(\text{imm32}, \text{carry}) = \text{A32ExpandImm_C}(\text{imm12}, \text{PSTATE}.C);
\]

T1

```
| 15 | 14 | 13 | 12 | 11 | 10 | 0 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 14 | 12 | 11 | 8 | 7 | | 0 |
----
S   Rn  immm3  Rd  imm8
```

**BIC variant**

Applies when S == 0.

BIC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
**BICS variant**

Applies when $S = 1$.

BICS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

**Decode for all variants of this encoding**

$$d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ \text{setflags} = (S == '1');$$

$$(\text{imm32}, \text{carry}) = \text{T32ExpandImm}_{C}(i:imm3:imm8, \text{PSTATE}.C);$  

if $d == 15$ || $n == 15$ then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:
  - For the BIC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
  - For the BICS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

- `<Rn>` For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
  For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

- `<const>` For encoding A1: an immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values.
  For encoding T1: an immediate value. See Modified immediate constants in T32 instructions on page F1-7242 for the range of values.

**Operation for all encodings**

$$\text{if ConditionPassed()} \text{ then}$$

$$\text{EncodingSpecificOperations();}$$

$$\text{result} = R[n] \text{ AND NOT}(\text{imm32});$$

if $d == 15$ then  // Can only occur for A32 encoding

$$\text{if setflags then}$$

$$\text{ALUExceptionReturn(result);}$$

$$\text{else}$$

$$\text{ALUWritePC(result);}$$

$$\text{else}$$

$$R[d] = \text{result};$$

$$\text{if setflags then}$$

$$\text{PSTATE.N = result<31>;}$$

$$\text{PSTATE.Z = IsZeroBit(result);}$$
PSTATE.C = carry;
// PSTATE.V unchanged

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.22   BIC, BICS (register)

Bitwise Bit Clear (register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the BICS variant of the instruction updates the condition flags based on the result.

The field descriptions for \(<Rd>\) identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The BIC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- The BICS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

<table>
<thead>
<tr>
<th>31 28 27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16 15 12 11</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=111</td>
<td>0 0 0 1 1 0 S</td>
<td>Rn</td>
<td>Rd</td>
<td>imm5</td>
<td>stype</td>
</tr>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**BIC, rotate right with extend variant**

Applies when S == 0 && imm5 == 00000 && stype == 11.

\[ \text{BIC}\{<c>\}{<q>} \{<Rd>,} <Rn>, <Rm>, \text{RRX} \]

**BIC, shift or rotate by value variant**

Applies when S == 0 && !(imm5 == 00000 && stype == 11).

\[ \text{BIC}\{<c>\}{<q>} \{<Rd>,} <Rn>, <Rm> \{, <shift> \#<amount>\} \]

**BICS, rotate right with extend variant**

Applies when S == 1 && imm5 == 00000 && stype == 11.

\[ \text{BICS}\{<c>\}{<q>} \{<Rd>,} <Rn>, <Rm>, \text{RRX} \]

**BICS, shift or rotate by value variant**

Applies when S == 1 && !(imm5 == 00000 && stype == 11).

\[ \text{BICS}\{<c>\}{<q>} \{<Rd>,} <Rn>, <Rm> \{, <shift> \#<amount>\} \]

**Decode for all variants of this encoding**

\[ d = \text{UInt}(Rd); \; n = \text{UInt}(Rn); \; m = \text{UInt}(Rm); \; \text{setflags} = (S == '1'); \]
\[ (shift\_t, shift\_n) = \text{DecodeImmShift(stype, imm5)}; \]
T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5</th>
<th>3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 0</td>
<td>1 1 1 0</td>
<td>Rm</td>
<td>Rdn</td>
</tr>
</tbody>
</table>

**T1 variant**

BIC<>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
BICS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block

**Decode for this encoding**

d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);

T2

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
<th>15 14 12</th>
<th>11 8 7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 0 1 0 0 0 1</td>
<td>S</td>
<td>Rn</td>
<td>(0)</td>
<td>imm3</td>
<td>Rd</td>
<td>imm2</td>
</tr>
</tbody>
</table>

**BIC, rotate right with extend variant**

Applies when \( S == 0 \&\& \text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11 \).

BIC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

**BIC, shift or rotate by value variant**

Applies when \( S == 0 \&\& \neg (\text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11) \).

BICW{<c>}{<q>} {<Rd>,} <Rn>, <Rm> // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1

**BICS, rotate right with extend variant**

Applies when \( S == 1 \&\& \text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11 \).

BICS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

**BICS, shift or rotate by value variant**

Applies when \( S == 1 \&\& \neg (\text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11) \).

BICS.W{<c>}{<q>} {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1

**Decode for all variants of this encoding**

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<c>  
See *Standard assembler syntax fields* on page F1-7228.

<q>  
See *Standard assembler syntax fields* on page F1-7228.

<Rdn>  
Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd>  
For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the BIC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-7133.
- For the BICS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn>  
For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm>  
For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift>  
Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- LSL  when stype = 00
- LSR  when stype = 01
- ASR  when stype = 10
- ROR  when stype = 11

<amount>  
For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND NOT(shifted);
    if d == 15 then  // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
        // PSTATE.V unchanged
```
Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.23 **BIC, BICS (register-shifted register)**

Bitwise Bit Clear (register-shifted register) performs a bitwise AND of a register value and the complement of a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

**A1**

```
| 31 28|27 26 25 24|23 22 21 20|19 16|15 12|11  8 | 7 6 5 4 | 3 | 0 |
|-----|-----|-----|-----|----|----|----|----|---|---|
|   !=1111| 0 0 0 1 | 1 1 |   S | Rn | Rd | Rs | 0 | stype | 1 | Rm |
```

Flag setting variant

Applies when $S = 1$.

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting variant

Applies when $S = 0$.

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

**Decode for all variants of this encoding**

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
  - LSL when stype = 00
  - LSR when stype = 01
  - ASR when stype = 10
  - ROR when stype = 11
- `<Rs>` Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND NOT(shifted);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.24   BKPT

Breakpoint causes a Breakpoint Instruction exception.

Breakpoint is always unconditional, even when inside an IT block.

A1

\[
\begin{array}{cccccccccc}
31 & 28 & 27 & 26 & 25 & 24 & 23 & 22 & 21 & 20 & 19 & | & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1=1111 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & | & \text{imm12} & & & 0 & 1 & 1 & 1 & \text{imm4} \\
\text{cond} & & & & & & & & & & \end{array}
\]

A1 variant

\[
\text{BKPT}\{<q>\} \{#\}<\text{imm}>
\]

Decode for this encoding

\[
\text{imm16} = \text{imm12:imm4}; \\
\text{if cond} \neq \text{"1110" then UNPREDICTABLE; } // \text{BKPT must be encoded with AL condition}
\]

CONSTRAINED UNPREDICTABLE behavior

If cond \neq \text{"1110"}, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes unconditionally.
• The instruction executes conditionally.

T1

\[
\begin{array}{cccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & | & 0 \\
1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & | \text{imm8} \\
\end{array}
\]

T1 variant

\[
\text{BKPT}\{<q>\} \{#\}<\text{imm}>
\]

Decode for this encoding

\[
\text{imm16} = \text{ZeroExtend(imm8, 16)};
\]

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\[
<q> \quad \text{See Standard assembler syntax fields on page F1-7228. A BKPT instruction must be unconditional.}
\]
&lt;imm&gt; For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm12:imm4” field. This value:
  • Is recorded in the Comment field of ESR_ELx.ISS if the Software Breakpoint Instruction exception is taken to an exception level that is using AArch64.
  • Is ignored otherwise.
For encoding T1: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. This value:
  • Is recorded in the Comment field of ESR_ELx.ISS if the Software Breakpoint Instruction exception is taken to an exception level that is using AArch64.
  • Is ignored otherwise.

**Operation for all encodings**

EncodingSpecificOperations();
AArch32.SoftwareBreakpoint(imm16);
F5.1.25 BL, BLX (immediate)

Branch with Link calls a subroutine at a PC-relative address, and setting LR to the return address.

Branch with Link and Exchange Instruction Sets (immediate) calls a subroutine at a PC-relative address, setting LR to the return address, and changes the instruction set from A32 to T32, or from T32 to A32.

A1

```
| 31 | 28|27 26 25 24|23 | | | | | 0 |
|-------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| cond              | imm24           |
```

**Condition**

1=1111
0=1011
1=0101

** imm24 = SignExtend(imm24:'00', 32); targetInstrSet = InstrSet_A32; 

A2

```
| 31 | 28|27 26 25 24|23 | | | | | 0 |
|-------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| cond              | imm24           |
```

**Condition**

1=1111
0=101
H=001

** imm32 = SignExtend(imm24:H:'0', 32); targetInstrSet = InstrSet_T32; 

T1

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th></th>
<th>0</th>
<th>15 14 13 12</th>
<th>11 10</th>
<th></th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>imm10</td>
<td>1</td>
<td>1</td>
<td>J1</td>
<td>1</td>
<td>J2</td>
<td>imm11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Condition**

S=1111
J1=0111
J2=1101

** I1 = NOT(J1 EOR S); I2 = NOT(J2 EOR S); imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32); targetInstrSet = InstrSet_T32; if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
T2

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>0</th>
<th>15 14 13 12</th>
<th>11 10</th>
<th>0</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>S</td>
<td>imm10H</td>
<td>1</td>
</tr>
</tbody>
</table>

**T2 variant**

BLX{<c>}{<q>} <label>

**Decode for this encoding**

if H == '1' then UNDEFINED;
I1 = NOT(J1 EOR S); I2 = NOT(J2 EOR S);
imm32 = SignExtend(S:I1:I2:imm10H:imm10L:'00', 32);
targetInstrSet = InstrSet_A32;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

<c>
For encoding A1, T1 and T2: see Standard assembler syntax fields on page F1-7228.
For encoding A2: see Standard assembler syntax fields on page F1-7228. <c> must be A1 or omitted.

<q>
See Standard assembler syntax fields on page F1-7228.

<label>
For encoding A1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BL instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are multiples of 4 in the range –33554432 to 33554428.
For encoding A2: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BLX instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are even numbers in the range –33554432 to 33554430.
For encoding T1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BL instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are even numbers in the range –16777216 to 16777214.
For encoding T2: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the Align(PC, 4) value of the BLX instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are multiples of 4 in the range –16777216 to 16777212.

**Operation for all encodings**

if ConditionPassed() then
EncodingSpecificOperations();
if CurrentInstrSet() == InstrSet_A32 then
 LR = PC - 4;
else
 LR = PC<<31:1>: '1';
bites(32) targetAddress;
if targetInstrSet == InstrSet_A32 then


targetAddress = Align(PC,4) + imm32;
else
  targetAddress = PC + imm32;
SelectInstrSet(targetInstrSet);
BranchWritePC(targetAddress, BranchType_DIRECALL);
F5.1.26   BLX (register)

Branch with Link and Exchange (register) calls a subroutine at an address specified in the register, and if necessary changes to the instruction set indicated by bit[0] of the register value. If the value in bit[0] is 0, the instruction set after the branch will be A32. If the value in bit[0] is 1, the instruction set after the branch will be T32.

A1

<table>
<thead>
<tr>
<th>31 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=1111</td>
<td>0 0 0 1</td>
<td>0 0 1 0</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
</tbody>
</table>

A1 variant

BLX(<c>{<q>}{<Rm>}

Decode for this encoding

m = UInt(Rm);
if m == 15 then UNPREDICTABLE;

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 1 1</td>
<td>Rm</td>
<td>(0)(0)(0)</td>
<td></td>
</tr>
</tbody>
</table>

T1 variant

BLX(<c>{<q>}{<Rm>}

Decode for this encoding

m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.
<q> See Standard assembler syntax fields on page F1-7228.
<Rm> Is the general-purpose register holding the address to be branched to, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    target = R[m];
    bits(32) next_instr_addr;
    if CurrentInstrSet() == InstrSet_A32 then
next_instr_addr = PC - 4;
LR = next_instr_addr;
else
    next_instr_addr = PC - 2;
    LR = next_instr_addr[31:1] : '1';
BXWritePC(target, BranchType_INDCALL);
F5.1.27 BX

Branch and Exchange causes a branch to an address and instruction set specified by a register.

A1

| 31 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8 |7 6 5 4 |3 0 |
|-----|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| !=1111 | 0 0 0 1 0 0 1 | (1)(1)(1)(1)(1)(1)(1)(1)(1)(1)| 0 0 0 1 |

cond

A1 variant

BX{<c>}{<q>} <Rm>

Decode for this encoding

m = UInt(Rm);

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 1 1 0</td>
<td>Rm</td>
<td>(0)(0)(0)</td>
<td></td>
</tr>
</tbody>
</table>

T1 variant

BX{<c>}{<q>} <Rm>

Decode for this encoding

m = UInt(Rm);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

<
See Standard assembler syntax fields on page F1-7228.

<Rm>
For encoding A1: is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The PC can be used.

For encoding T1: is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The PC can be used.

--- Note ---

If <Rm> is the PC at a non word-aligned address, it results in UNPREDICTABLE behavior because the address passed to the BXWritePC() pseudocode function has bits<1:0> = '10'.

<table>
<thead>
<tr>
<th>0x0</th>
<th>00010010</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0001</td>
</tr>
<tr>
<td>0x0</td>
<td>00001110</td>
</tr>
<tr>
<td>0x0</td>
<td>00001</td>
</tr>
</tbody>
</table>
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R[m], BranchType_INDIR);
F5.1.28   BXJ

Branch and Exchange, previously Branch and Exchange Jazelle.

BXJ behaves as a BX instruction, see BX. This means it causes a branch to an address and instruction set specified by a register.

A1

```
|31|  28|  27|  26|  25|  24| 23|  22|  21|  20|  19|  18|  17|  16|  15|  14|  13|  12|  11|  10|  9|  8|  7|  6|  5|  4|  3|  0|
| l=1111 | 0| 0| 0| 1| 0| 0|1|1|1|1|1|1|1|1|1|1|1|1|0|0|1|0|Rm|
```

**A1 variant**

```
BXJ{<c>}{<q>} <Rm>
```

**Decode for this encoding**

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
```

T1

```
|15|  14|  13|  12| 11|  10|  9|  8|  7|  6|  5|  4|  3|  0| 15|  14|  13|  12| 11|  10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0|
| 1| 1| 1| 1| 0| 0| 1| 1| 1| 0| 0|Rm| 1| 0| 0|1|1|1|0|0|0|0|0|0|0|0|0|0|
```

**T1 variant**

```
BXJ{<c>}{<q>} <Rm>
```

**Decode for this encoding**

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

**Notes for all encodings**

For more information about the CONstrained UNPREDicTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDicTABLE Behaviors.

**Assembler symbols**

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rm>` Is the general-purpose register holding the address to be branched to, encoded in the "Rm" field.

**Operation for all encodings**

```
if ConditionPassed() then
  EncodingSpecificOperations();
  BXWritePC(R[m], BranchType_INDIR);
```
F5.1.29  CBNZ, CBZ

Compare and Branch on Nonzero and Compare and Branch on Zero compare the value in a register with zero, and conditionally branch forward a constant value. They do not affect the condition flags.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7</th>
<th>3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1</td>
<td>0 1</td>
<td>1</td>
</tr>
</tbody>
</table>

**CBNZ variant**

Applies when op == 1.

CBNZ{<q>} <Rn>, <label>

**CBZ variant**

Applies when op == 0.

CBZ{<q>} <Rn>, <label>

**Decode for all variants of this encoding**

```
n = UInt(Rn);  imm32 = ZeroExtend(i:imm5:'0', 32);  nonzero = (op == '1');
if InITBlock() then UNPREDICTABLE;
```

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

<q>  See Standard assembler syntax fields on page F1-7228.

<Rn>  Is the general-purpose register to be tested, encoded in the "Rn" field.

<label>  Is the program label to be conditionally branched to. Its offset from the PC, a multiple of 2 and in the range 0 to 126, is encoded as "i:imm5" times 2.

**Operation**

```
EncodingSpecificOperations();
if nonzero != IsZero(R[n]) then
    CBWritePC(PC + imm32);
```
F5.1.30  CLREX

Clear-Exclusive clears the local monitor of the executing PE.

A1

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8 |7 6 5 4 |3 2 1 0 ]
1 1 1 1 0 1 0 1 1 1 [1](1)(1)(1)(1)(1)(1)(1)(1)(1)(1)(1)(1)(1)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)
```

*A1 variant*

CLREX{<c>}{<q>}

*Decode for this encoding*

// No additional decoding required

T1

```
[15 14 13 12|11 10 9 8 |7 6 5 4 |3 2 1 0 |15 14 13 12|11 10 9 8 |7 6 5 4 |3 2 1 0 ]
```

*T1 variant*

CLREX{<c>}{<q>}

*Decode for this encoding*

// No additional decoding required

*Notes for all encodings*

For more information about the CONSTRANGED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

*Assembler symbols*

For encoding A1: see Standard assembler syntax fields on page F1-7228. Must be Al or omitted.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

*<c>* See Standard assembler syntax fields on page F1-7228.

*Operation for all encodings*

if ConditionPassed() then
    EncodingSpecificOperations();
    ClearExclusiveLocal(ProcessorID());
F5.1.31  CLZ

Count Leading Zeros returns the number of binary zero bits before the first binary one bit in a value.

A1

```
| [31] 28|27 26 25 24|23 22 21 20|19 18 17 16|15 12|11 10 9 8 |7 6 5 4 |3 |0 |
|------|------------------|-------------------|-------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| !=1111| 0 |0 |0 |1 |0 |1 |0 |0 |1 |1 |1 |Rd |1 |0 |0 |1 |0 |0 |1 |0 |0 |Rm |
```

**A1 variant**

CLZ{<c>}{<q>} <Rd>, <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(\text{Rd}); \quad m = \text{UInt}(\text{Rm});
\]

\[
\text{if } d == 15 || m == 15 \text{ then UNPREDICTABLE;}
\]

T1

```
<table>
<thead>
<tr>
<th>[15 14 13 12]</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**T1 variant**

CLZ{<c>}{<q>} <Rd>, <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(\text{Rd}); \quad m = \text{UInt}(\text{Rm}); \quad n = \text{UInt}(\text{Rn});
\]

\[
\text{if } m != n || d == 15 || m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( m != n \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes with the additional decode: \( m = \text{UInt}(\text{Rn}); \).
- The value in the destination register is UNKNOWN.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields* on page F1-7228.
- `<q>` See *Standard assembler syntax fields* on page F1-7228.
<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field.
For encoding T1: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    result = CountLeadingZeroBits(R[m]);
    R[d] = result<31:0>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.32  CMN (immediate)

Compare Negative (immediate) adds a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

A1

\[
\begin{array}{ccccccccccccc}
\hline
! & 1111 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & Rn & 0 & 0 & 0 & 0 & \hline
\end{array}
\]

\text{cond}

A1 variant

\text{CMN}\{<c>\}{<q>} <Rn>, #<const>

\text{Decode for this encoding}

\[n = \text{UInt}(Rn); \text{ imm32 } = \text{A32ExpandImm(imm12)};\]

T1

\[
\begin{array}{ccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 14 & 12 & 11 & 10 & 9 & 8 & 7 & 0 \\
\hline
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & Rn & 0 & \text{imm3} & 1 & 1 & 1 & \text{imm8} & \hline
\end{array}
\]

T1 variant

\text{CMN}\{<c>\}{<q>} <Rn>, #<const>

\text{Decode for this encoding}

\[n = \text{UInt}(Rn); \text{ imm32 } = \text{T32ExpandImm(i:imm3:imm8)};\]

\text{if } n = 15 \text{ then UNPREDICTABLE;}

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see \textit{Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors}.

Assembler symbols

\(<c>\)  See \textit{Standard assembler syntax fields on page F1-7228}.

\(<q>\)  See \textit{Standard assembler syntax fields on page F1-7228}.

\(<\text{Rn}>\)  For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

\(<\text{const}>\)  For encoding A1: an immediate value. See \textit{Modified immediate constants in A32 instructions on page F1-7244} for the range of values.

For encoding T1: an immediate value. See \textit{Modified immediate constants in T32 instructions on page F1-7242} for the range of values.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], imm32, '0');
    PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.33 CMN (register)

Compare Negative (register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

A1

\[
\begin{array}{cccccccccc}
\end{array}
\]

\(l=1111\) 0 0 0 1 0 1 1 0

\textbf{Rotate right with extend variant}

Applies when \(\text{imm5} == 00000 \&\& \text{stype} == 11\).

\texttt{CMN\{}<c>\}\{<q>\} <Rn>, <Rm>, RRX

\textbf{Shift or rotate by value variant}

Applies when \(! (\text{imm5} == 00000 \&\& \text{stype} == 11)\).

\texttt{CMN\{}<c>\}\{<q>\} <Rn>, <Rm> \{, <shift> #<amount>\}

\textit{Decode for all variants of this encoding}

\(n = \text{UInt}(Rn); \ m = \text{UInt}(Rm);\)
\((\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift(stype, imm5)};\)

T1

\[
\begin{array}{ccccccccc}
|15&14&13&12|11&10&9&8&7&6&5&3&2&0|
\end{array}
\]

0 1 0 0 0 0 1 0 1 1 0

\textbf{T1 variant}

\texttt{CMN\{}<c>\}\{<q>\} <Rn>, <Rm>

\textit{Decode for this encoding}

\(n = \text{UInt}(Rn); \ m = \text{UInt}(Rm);\)
\((\text{shift}_t, \text{shift}_n) = (\text{SRType}_{\text{LSL}}, 0);\)

T2

\[
\begin{array}{cccccccccccc}
|15&14&13&12|11&10&9&8&7&6&5&4&3&0|
\end{array}
\]

1 1 1 0 1 0 1 1 0 0 0 1

\textbf{Rotate right with extend variant}

Applies when \(\text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11\).

\texttt{CMN\{}<c>\}\{<q>\} <Rn>, <Rm>, RRX
**Shift or rotate by value variant**

Applies when !((imm3 == 000 && imm2 == 00 && stype == 11)).

CMN{<c>}.W <Rn>, <Rm> // <Rn>, <Rm> can be represented in T1
CMN{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

**Decode for all variants of this encoding**

n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONstrained UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

<\> See Standard assembler syntax fields on page F1-7228.

<\> See Standard assembler syntax fields on page F1-7228.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
For encoding T1 and T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<\> Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- LSL when stype = 00
- LSR when stype = 01
- ASR when stype = 10
- ROR when stype = 11

<\> For encoding A1: is the shift amount, in the range 1 to 31 (when <\> = LSL or ROR) or 1 to 32 (when <\> = LSR or ASR) encoded in the "imm5" field as <\> modulo 32.
For encoding T2: is the shift amount, in the range 1 to 31 (when <\> = LSL or ROR) or 1 to 32 (when <\> = LSR or ASR), encoded in the "imm3:imm2" field as <\> modulo 32.

**Operation for all encodings**

if ConditionPassed() then
  EncodingSpecificOperations();
  shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
  (result, nzcv) = AddWithCarry(R[n], shifted, '0');
  PSTATE.<N,Z,C,V> = nzcv;
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.34 CMN (register-shifted register)

Compare Negative (register-shifted register) adds a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

A1

```
<table>
<thead>
<tr>
<th>31 28 27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 14 13 12</th>
<th>11 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>=1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

A1 variant

CMN(<c>){<q>} <Rn>, <Rm>, <type> <Rs>

**Decode for this encoding**

\[ n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad s = \text{UInt}(Rs); \]
\[ \text{shift}_t = \text{DecodeRegShift}(stype); \]
\[ \text{if } n == 15 || m == 15 || s == 15 \text{ then UNPREDICTABLE;} \]

**Notes for all encodings**

For more information about the CONSTRANGED UNPREDICTABLE behavior, see *Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields* on page F1-7228.
- `<q>` See *Standard assembler syntax fields* on page F1-7228.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
- `<type>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
  - LSL when stype = 00
  - LSR when stype = 01
  - ASR when stype = 10
  - ROR when stype = 11
- `<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

**Operation**

```pseudo
if ConditionPassed() then
  EncodingSpecificOperations();
  shift_n = UInt(R[s]<7:0>);
  shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
  (result, nzcv) = AddWithCarry(R[n], shifted, '0');
  PSTATE.<N,Z,C,V> = nzcv;
```
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.35   CMP (immediate)

Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

**A1**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=1111</td>
<td>0 0 1 0 0 1 0 1</td>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*cond*

**A1 variant**

CMP{<c>}{<q>} <Rn>, #<const>

**Decode for this encoding**

n = UInt(Rn); imm32 = A32ExpandImm(imm12);

**T1**

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 1</td>
<td>Rn</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**T1 variant**

CMP{<c>}{<q>} <Rn>, #imm8

**Decode for this encoding**

n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);

**T2**

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3</th>
<th>0</th>
<th>15 14 12</th>
<th>11 10 9 8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0</td>
<td>i</td>
<td>0 1 1 0 1</td>
<td>Rn</td>
<td>0</td>
<td>imm3</td>
</tr>
</tbody>
</table>

**T2 variant**

CMP{<c>}.W <Rn>, #<const> // <Rd>, <const> can be represented in T1

CMP{<c>}{<q>} <Rn>, #<const>

**Decode for this encoding**

n = UInt(Rn); imm32 = T32ExpandImm(i:imm3:imm8);

if n == 15 then UNPREDICTABLE;

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<Rn> For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
For encoding T1: is a general-purpose source register, encoded in the "Rn" field.
For encoding T2: is the general-purpose source register, encoded in the "Rn" field.

<imm8> Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.
<const> For encoding A1: an immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values.
For encoding T2: an immediate value. See Modified immediate constants in T32 instructions on page F1-7242 for the range of values.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], NOT(imm32), '1');
    PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.36  **CMP (register)**

Compare (register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

### A1

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>[0][0][0][0]</td>
<td>imm5</td>
<td>stype</td>
<td>0</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Rotate right with extend variant**

Applies when imm5 == 00000 && stype == 11.

```
CMP{<c>}{<q>} <Rn>, <Rm>, RRX
```

**Shift or rotate by value variant**

Applies when !(imm5 == 00000 && stype == 11).

```
CMP{<c>}{<q>} <Rn>, <Rm> {}, <shift> #<amount>
```

**Decode for all variants of this encoding**

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

**T1**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**T1 variant**

```
CMP{<c>}{<q>} <Rn>, <Rm> // <Rn> and <Rm> both from R0-R7
```

**Decode for this encoding**

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

**T2**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>N</td>
<td>Rm</td>
<td>Rn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**T2 variant**

```
CMP{<c>}{<q>} <Rn>, <Rm> // <Rn> and <Rm> not both from R0-R7
```
Decode for this encoding

\[
n = \text{UInt}(N:Rn); \quad m = \text{UInt}(Rm);
\]
\[
(shift_t, shift_n) = (\text{SRType} \text{\_LSL}, 0);
\]
\[
\text{if } n < 8 \text{ \&\& } m < 8 \text{ then UNPREDICTABLE;}
\]
\[
\text{if } n == 15 \text{ \&\& } m == 15 \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \(n < 8 \text{ \&\& } m < 8\), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The condition flags become **UNKNOWN**.

### T3

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 0</th>
<th>15 14</th>
<th>12</th>
<th>11 10 9 8 7 6 5 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 0</td>
<td>1 1 0 1</td>
<td>Rn</td>
<td>[0]</td>
<td>imm3</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

**Rotate right with extend variant**

Applies when \(\text{imm3} == 000 \text{ \&\& } \text{imm2} == 00 \text{ \&\& } \text{stype} == 11\).

\[
\text{CMP}\{<c>\}\{<q>\} <Rn>, <Rm>, \text{RRX}
\]

**Shift or rotate by value variant**

Applies when \(!(\text{imm3} == 000 \text{ \&\& } \text{imm2} == 00 \text{ \&\& } \text{stype} == 11)\).

\[
\text{CMP}\{<c>\}.W <Rn>, <Rm> \text{ // } <Rn>, <Rm> \text{ can be represented in T1 or T2}
\]
\[
\text{CMP}\{<c>\}\{<q>\} <Rn>, <Rm>, \text{shift} \# <\text{amount}>
\]

**Decode for all variants of this encoding**

\[
n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]
\[
(shift_t, shift_n) = \text{DecodeImmShift(stype, imm3:imm2)};
\]
\[
\text{if } n == 15 \text{ \&\& } m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

### Notes for all encodings

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Appendix K1](#) *Architectural Constraints on UNPREDICTABLE Behaviors*.

### Assembler symbols

- **<c>** See *Standard assembler syntax fields on page F1-7228*.
- **<q>** See *Standard assembler syntax fields on page F1-7228*.
- **<Rn>** For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
  For encoding T1 and T3: is the first general-purpose source register, encoded in the "Rn" field.
  For encoding T2: is the first general-purpose source register, encoded in the "N:Rn" field.
<Rm>
For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
For encoding T1, T2 and T3: is the second general-purpose source register, encoded in the "Rm" field.

<shift>
Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
- LSL when stype = 00
- LSR when stype = 01
- ASR when stype = 10
- ROR when stype = 11

<amount>
For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.
For encoding T3: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    PSTATE.<N,Z,C,V> = nzcv;

Operational information
If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.37  CMP (register-shifted register)

Compare (register-shifted register) subtracts a register-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

A1

![Instruction Set Format]

A1 variant

CMP{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>

Decode for this encoding

\[ n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad s = \text{UInt}(Rs); \]
\[ \text{shift}_t = \text{DecodeRegShift}(\text{stype}); \]
\[ \text{if } n == 15 \quad \text{or} \quad m == 15 \quad \text{or} \quad s == 15 \quad \text{then UNPREDICTABLE;} \]

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\) See Standard assembler syntax fields on page F1-7228.
\(<q>\) See Standard assembler syntax fields on page F1-7228.
\(<Rn>\) Is the first general-purpose source register, encoded in the "Rn" field.
\(<Rm>\) Is the second general-purpose source register, encoded in the "Rm" field.
\(<\text{type}>\) Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
- LSL when stype = 00
- LSR when stype = 01
- ASR when stype = 10
- ROR when stype = 11
\(<Rs>\) Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  shift_n = UInt(R[s]<7:0>);
  shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
  (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
  PSTATE.<N,Z,C,V> = nzcv;
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.38  CPS, CPSID, CPSIE

Change PE State changes one or more of the PSTATE.{A, I, F} interrupt mask bits and, optionally, the PSTATE.M mode field, without changing any other PSTATE bits.

CPS is treated as NOP if executed in User mode unless it is defined as being CONSTRAINED UNPREDICTABLE elsewhere in this section.

The PE checks whether the value being written to PSTATE.M is legal. See Illegal changes to PSTATE.M on page G1-8925.

A1

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 0 1 0 0 0 0</td>
<td>imod</td>
<td>M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>I</td>
<td>F</td>
</tr>
</tbody>
</table>

Change mode variant

Applies when imod == 00 && M == 1.

CPS{<q>} #<mode> // Cannot be conditional

Interrupt disable variant

Applies when imod == 11 && M == 0.

CPSID{<q>} <iflags> // Cannot be conditional

Interrupt disable and change mode variant

Applies when imod == 11 && M == 1.

CPSID{<q>} <iflags> , #<mode> // Cannot be conditional

Interrupt enable variant

Applies when imod == 10 && M == 0.

CPSIE{<q>} <iflags> // Cannot be conditional

Interrupt enable and change mode variant

Applies when imod == 10 && M == 1.

CPSIE{<q>} <iflags> , #<mode> // Cannot be conditional

Decode for all variants of this encoding

if mode != '00000' && M == '0' then UNPREDICTABLE;
if (imod<1> == '1' && A:I:F == '000') || (imod<1> == '0' && A:I:F != '000') then UNPREDICTABLE;
enable = (imod == '10'); disable = (imod == '11'); changemode = (M == '1');
affectA = (A == '1'); affectI = (I == '1'); affectF = (F == '1');
if (imod == '00' && M == '0') || imod == '01' then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If imod == '01', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
If \( \text{imod} = '00' \) \&\& \( M = '0' \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If \( \text{mode} \neq '00000' \) \&\& \( M = '0' \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: \( \text{changemode} = \text{TRUE} \).
- The instruction executes as described, and the value specified by mode is ignored. There are no additional side-effects.

If \( \text{imod}<1> = '1' \) \&\& \( \text{A:I:F} = '000' \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if \( \text{imod}<1> = '0' \).
- The instruction behaves as if \( \text{A:I:F} \) has an UNKNOWN nonzero value.

If \( \text{imod}<1> = '0' \) \&\& \( \text{A:I:F} \neq '000' \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if \( \text{imod}<1> = '1' \).
- The instruction behaves as if \( \text{A:I:F} = '000' \).

\textbf{T1}

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 0 1 0 0 1 1</td>
</tr>
</tbody>
</table>

\textit{Interrupt disable variant}

Applies when \( \text{im} = 1 \).

\texttt{CPSID\{<q>\} <iflags>} // Not permitted in IT block

\textit{Interrupt enable variant}

Applies when \( \text{im} = 0 \).

\texttt{CPSIE\{<q>\} <iflags>} // Not permitted in IT block

\textit{Decode for all variants of this encoding}

\begin{verbatim}
if A:I:F == '000' then UNPREDICTABLE;
enable = (im == '0'); disable = (im == '1'); changemode = FALSE;
affectA = (A == '1'); affectI = (I == '1'); affectF = (F == '1');
if InITBlock() then UNPREDICTABLE;
\end{verbatim}
**CONSTRANGED UNPREDICTABLE behavior**

If $A:I:F == '000'$, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.

**T2**

```
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | (1)(1)(1)| 1 | 0 | (0) | 0 | (0) | imod | M | A | I | F | mode |
```

**Change mode variant**

Applies when $imod == 00$ && $M == 1$.

`CPS{<q>} #<mode>` // Not permitted in IT block

**Interrupt disable variant**

Applies when $imod == 11$ && $M == 0$.

`CPSID.W <iflags>` // Not permitted in IT block

**Interrupt disable and change mode variant**

Applies when $imod == 11$ && $M == 1$.

`CPSID{<q>} <iflags>, #<mode>` // Not permitted in IT block

**Interrupt enable variant**

Applies when $imod == 10$ && $M == 0$.

`CPSIE.W <iflags>` // Not permitted in IT block

**Interrupt enable and change mode variant**

Applies when $imod == 10$ && $M == 1$.

`CPSIE{<q>} <iflags>, #<mode>` // Not permitted in IT block

**Decode for all variants of this encoding**

if $imod == '00'$ && $M == '0'$ then SEE "Hint instructions";
if mode != '00000' && $M == '0'$ then UNPREDICTABLE;
if (imod<1> == '1' && $A:I:F == '000') || (imod<1> == '0' && $A:I:F != '000') then UNPREDICTABLE;
if imod == '01' || InITBlock() then UNPREDICTABLE;

**CONSTRANGED UNPREDICTABLE behavior**

If $imod == '01'$, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.

If mode != '00000' && $M == '0'$, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
• The instruction executes as NOP.
• The instruction executes with the additional decode: changemode = TRUE.
• The instruction executes as described, and the value specified by mode is ignored. There are no additional side-effects.

If imod<1> == '1' && A:I:F == '000', then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction behaves as if imod<1> == '0'.
• The instruction behaves as if A:I:F has an UNKNOWN nonzero value.

If imod<1> == '0' && A:I:F != '000', then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction behaves as if imod<1> == '1'.
• The instruction behaves as if A:I:F == '000'.

Notes for all encodings

Hint instructions: In encoding T2, if the imod field is 00 and the M bit is 0, a hint instruction is encoded. To determine which hint instruction, see Branches and miscellaneous control on page F3-7343.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.

<iflags> Is a sequence of one or more of the following, specifying which interrupt mask bits are affected:
    a Sets the A bit in the instruction, causing the specified effect on PSTATE.A, the SError interrupt mask bit.
    i Sets the I bit in the instruction, causing the specified effect on PSTATE.I, the IRQ interrupt mask bit.
    f Sets the F bit in the instruction, causing the specified effect on PSTATE.F, the FIQ interrupt mask bit.

<mode> Is the number of the mode to change to, in the range 0 to 31, encoded in the "mode" field.

Operation for all encodings

if CurrentInstrSet() == InstrSet_A32 then
    EncodingSpecificOperations();
    if PSTATE.EL != EL0 then
        if enable then
            if affectA then PSTATE.A = '0';
            if affectI then PSTATE.I = '0';
            if affectF then PSTATE.F = '0';
        if disable then
            if affectA then PSTATE.A = '1';
            if affectI then PSTATE.I = '1';
            if affectF then PSTATE.F = '1';
        if changemode then
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
AArch32.WriteModeByInstr(mode);

else

EncodingSpecificOperations();
if PSTATE.EL != EL0 then
   if enable then
      if affectA then PSTATE.A = '0';
      if affectI then PSTATE.I = '0';
      if affectF then PSTATE.F = '0';
   if disable then
      if affectA then PSTATE.A = '1';
      if affectI then PSTATE.I = '1';
      if affectF then PSTATE.F = '1';
   if changemode then
      // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
      AArch32.WriteModeByInstr(mode);
F5.1.39   CRC32

CRC32 performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes
an input CRC value in the first source operand, performs a CRC on the input value in the second source operand,
and returns the output CRC value. The second source operand can be 8, 16, or 32 bits. To align with common usage,
the bit order of the values is reversed as part of the operation, and the polynomial \(0x04C11DB7\) is used for the CRC
calculation.

In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all
implementations to implement this instruction.

Note

ID_ISAR5.CRC32 indicates whether this instruction is supported in the T32 and A32 instruction sets.

A1

(FEAT_CRC32)

| 31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 6 5 4 3 0 |
| l=111           | 0 | 0 | 0 | 1 | 0 | sz | 0 | Rd | [0] | 0 | 0 | 0 | 0 | Rn | [0] | 0 | 0 | 0 | 0 | Rm |

cond C

CRC32B variant

Applies when sz == 00.

CRC32B{<q>} <Rd>, <Rn>, <Rm>

CRC32H variant

Applies when sz == 01.

CRC32H{<q>} <Rd>, <Rn>, <Rm>

CRC32W variant

Applies when sz == 10.

CRC32W{<q>} <Rd>, <Rn>, <Rm>

Decode for all variants of this encoding

\[
\begin{align*}
\text{if } & \text{ ! HaveCRCExt() then UNDEFINED;} \\
& d = \text{UInt(Rd); } n = \text{UInt(Rn); } m = \text{UInt(Rm);} \\
& \text{size} = 8 \ll \text{UInt(sz);} \\
& \text{crc32c} = (C == '1'); \\
& \text{if } d == 15 \text{ } | | \text{ } n == 15 \text{ } | | \text{ } m == 15 \text{ then UNPREDICTABLE;} \\
& \text{if } \text{size} == 64 \text{ then UNPREDICTABLE;} \\
& \text{if } \text{cond} != '1110' \text{ then UNPREDICTABLE;} \\
\end{align*}
\]

CONSTRANDED UNPREDICTABLE behavior

If size == 64, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: size = 32;
If cond != '1111', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

**T1**

(FEAT_CRC32)

CRC32B variant

Applies when sz == 00.

CRC32B{<q>} <Rd>, <Rn>, <Rm>

CRC32H variant

Applies when sz == 01.

CRC32H{<q>} <Rd>, <Rn>, <Rm>

CRC32W variant

Applies when sz == 10.

CRC32W{<q>} <Rd>, <Rn>, <Rm>

Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If size == 64, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: size = 32;

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

$q$  See Standard assembler syntax fields on page F1-7228. A CRC32 instruction must be unconditional.

$<Rd>$ Is the general-purpose accumulator output register, encoded in the "Rd" field.

$<Rn>$ Is the general-purpose accumulator input register, encoded in the "Rn" field.

$<Rm>$ Is the general-purpose data source register, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();

  acc = R[n];  // accumulator
  val = R[m]<size-1:0>;  // input value
  poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;
  tempacc = BitReverse(acc):Zeros(size);
  tempval = BitReverse(val):Zeros(32);
  // Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
  R[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.40 CRC32C

CRC32C performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, or 32 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x1EDC6F41 is used for the CRC calculation.

In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note ID_ISAR5.CRC32 indicates whether this instruction is supported in the T32 and A32 instruction sets.

A1

(FEAT_CRC32)

CRC32CB variant
Applies when sz == 00.
CRC32CB{<q>} <Rd>, <Rn>, <Rm>

CRC32CH variant
Applies when sz == 01.
CRC32CH{<q>} <Rd>, <Rn>, <Rm>

CRC32CW variant
Applies when sz == 10.
CRC32CW{<q>} <Rd>, <Rn>, <Rm>

Decode for all variants of this encoding

if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
if cond != '1110' then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior
If size == 64, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: size = 32;
If cond \(!= '1110'\), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction executes unconditionally.
- The instruction executes conditionally.

**T1**

(FEAT_CRC32)

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3</th>
<th>0</th>
<th>15 14 13 12</th>
<th>11 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 0 1 0 1</td>
<td>Rn 1 1 1 1</td>
<td>Rd 1 0 sz</td>
<td>Rm</td>
<td></td>
</tr>
</tbody>
</table>

**CRC32CB variant**

Applies when \(sz == 00\).

\[\text{CRC32CB}\{<q>\} \ <Rd>, \ <Rn>, \ <Rm>\]

**CRC32CH variant**

Applies when \(sz == 01\).

\[\text{CRC32CH}\{<q>\} \ <Rd>, \ <Rn>, \ <Rm>\]

**CRC32CW variant**

Applies when \(sz == 10\).

\[\text{CRC32CW}\{<q>\} \ <Rd>, \ <Rn>, \ <Rm>\]

**Decode for all variants of this encoding**

\[
\begin{align*}
\text{if InITBlock()} \text{ then UNPREDICTABLE; } \\
\text{if ! HaveCRCExt()} \text{ then UNDEFINED; } \\
d \equiv \text{ UInt}(Rd); \ n \equiv \text{ UInt}(Rn); \ m \equiv \text{ UInt}(Rm); \\
\text{size} \equiv 8 \ll \text{ UInt}(sz); \\
\text{crc32c} = (C \equiv '1'); \\
\text{if } d \equiv 15 \ | | \ n \equiv 15 \ | | \ m \equiv 15 \text{ then UNPREDICTABLE; } \\
\text{if } \text{size} \equiv 64 \text{ then UNPREDICTABLE; }
\end{align*}
\]

**CONSTRANGED UNPREDICTABLE behavior**

If \(\text{size} == 64\), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction executes with the additional decode: \(\text{size} = 32\).

**Notes for all encodings**

For more information about the CONSTRANGED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors.*
Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228. A CRC32C instruction must be unconditional.

<Rd> Is the general-purpose accumulator output register, encoded in the "Rd" field.

<Rn> Is the general-purpose accumulator input register, encoded in the "Rn" field.

<Rm> Is the general-purpose data source register, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();

    acc = R[n];              // accumulator
    val = R[m]<size-1:0>;    // input value
    poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;
    tempacc = BitReverse(acc):Zeros(size);
    tempval = BitReverse(val):Zeros(32);
    // Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
    R[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.41 CSDB

Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution and data value prediction.

No instruction other than branch instructions and instructions that write to the PC appearing in program order after the CSDB can be speculatively executed using the results of any:

- Data value predictions of any instructions.
- PSTATE.\{N,Z,C,V\} predictions of any instructions other than conditional branch instructions and conditional instructions that write to the PC appearing in program order before the CSDB that have not been architecturally resolved.

Note

For purposes of the definition of CSDB, PSTATE.\{N,Z,C,V\} is not considered a data value. This definition permits:

- Control flow speculation before and after the CSDB.
- Speculative execution of conditional data processing instructions after the CSDB, unless they use the results of data value or PSTATE.\{N,Z,C,V\} predictions of instructions appearing in program order before the CSDB that have not been architecturally resolved.

A1

```
cond  !='1110' then UNPREDICTABLE;      // CSDB must be encoded with AL condition
```

A1 variant

CSDB\{<c>\}{<q>}

Decode for this encoding

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

```
cond  !='1110' then UNPREDICTABLE;      // CSDB must be encoded with AL condition
```

T1 variant

CSDB\{<c>\}{<q>}
Decode for this encoding

if InITBlock() then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

Notes for all encodings

For more information about the CONSTRANGED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

See Standard assembler syntax fields on page F1-7228.

Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations();
   ConsumptionOfSpeculativeDataBarrier();
F5.1.42  DBG

DBG executes as a NOP. Arm deprecates any use of the DBG instruction.

### A1

<table>
<thead>
<tr>
<th>31 28 27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=1111</td>
<td>0 0 1 1 0</td>
<td>0 1 0 0 0</td>
<td>0 0 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>option</td>
</tr>
</tbody>
</table>

**A1 variant**

DBG{<c>}{<q>} #<option>

**Decode for this encoding**

// DBG executes as a NOP. The 'option' field is ignored

### T1

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 | 0 |
|-------------|-----------|-----------|-----------|
| 1 1 1 1 0   | 1 1 0 1 0 | 0 0 1 1 1 | option    |

**T1 variant**

DBG{<c>}{<q>} #<option>

**Decode for this encoding**

// DBG executes as a NOP. The 'option' field is ignored

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>`  
  See Standard assembler syntax fields on page F1-7228.

- `<q>`  
  See Standard assembler syntax fields on page F1-7228.

- `<option>`  
  Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "option" field.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations();
F5.1.43  DCPS1

Debug Change PE State to EL1 allows the debugger to move the PE into EL1 from EL0 or to a specific mode at the current Exception level.

DCPS1 is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL2 is implemented, EL2 is implemented and enabled in the current Security state, and any of:
  - EL2 is using AArch32 and HCR.TGE is set to 1.
  - EL2 is using AArch64 and HCR_EL2.TGE is set to 1.

When the PE executes DCPS1 at EL0, EL1 or EL3:

- If EL3 or EL1 is using AArch32, the PE enters SVC mode and LR_svc, SPSR_svc, DLR, and DSPSR become UNKNOWN. If DCPS1 is executed in Monitor mode, SCR.NS is cleared to 0.
- If EL1 is using AArch64, the PE enters EL1 using AArch64, selects SP_EL1, and ELR_EL1, ESR_EL1, SPSR_EL1, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

When the PE executes DCPS1 at EL2 the PE does not change mode, and ELR_hyp, HSR, SPSR_hyp, DLR and DSPSR become UNKNOWN.

For more information on the operation of the DCPS<n> instructions, see DCPS<n> on page H2-10262.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 1 1 1 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T1 variant

DCPS1

Decode for this encoding

// No additional decoding required.

Operation

if !Halted() then UNDEFINED;

if EL2Enabled() && PSTATE_EL == EL0 then
tge = if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE;
if tge == '1' then UNDEFINED;

if PSTATE_EL != EL0 || ELUsingAArch32(EL1) then
  if PSTATE.M == M32_Monitor then SCR.NS = '0';
  if PSTATE_EL != EL2 then
    AArch32.WriteMode(M32_Svc);
    PSTATE.E = SCTLR.EE;
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    LR_svc = bits(32) UNKNOWN;
    SPSR_svc = bits(32) UNKNOWN;
  else
    PSTATE.E = HSCTLR.EE;
    ELR_hyp = bits(32) UNKNOWN;
    HSR = bits(32) UNKNOWN;
    SPSR_hyp = bits(32) UNKNOWN;
DLR = bits(32) UNKNOWN;
DSPSR = bits(32) UNKNOWN;
else // Targeting EL1 using AArch64
AArch64.MaybeZeroRegisterUppers();
MaybeZeroSVEUppers(EL1);
PSTATE.nRW = '0';
PSTATE.SP = '1';
PSTATE.EL = EL1;
if HavePANExt() && SCTLR_EL1.SPAN == '0' then PSTATE.PAN = '1';
if HaveUAOExt() then PSTATE.UAO = '0';

ELR_EL1 = bits(64) UNKNOWN;
ESR_EL1 = bits(64) UNKNOWN;
SPSR_EL1 = bits(64) UNKNOWN;

if HaveIESB() && SCTLR_EL1.IESB == '1' && !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
    SynchronizeErrors();
UpdateEDSCRFields(); // Update EDSCR PE state flags
F5.1.44  DCPS2

Debug Change PE State to EL2 allows the debugger to move the PE into EL2 from a lower Exception level.

DCPS2 is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL2 is not implemented.
- The PE is in Secure state and any of:
  - Secure EL2 is not implemented.
  - Secure EL2 is implemented and Secure EL2 is disabled.

When the PE executes DCPS2:

- If EL2 is using AArch32, the PE enters Hyp mode and ELR_hyp, HSR, SPSR_hyp, DLR and DSPSR become UNKNOWN.
- If EL2 is using AArch64, the PE enters EL2 using AArch64, selects SP_EL2, and ELR_EL2, ESR_EL2, SPSR_EL2, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

For more information on the operation of the DCPS<n> instructions, see DCPS<n> on page H2-10262.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 1 1 1 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T1 variant

DCPS2

Decode for this encoding

if !HaveEL(EL2) then UNDEFINED;

Operation

if !Halted() || !EL2Enabled() then UNDEFINED;

if ELUsingAArch32(EL2) then
   AArch32.WriteMode(M32_Hyp);
   PSTATE.E = HSCTRL.EE;

   ELR_hyp = bits(32) UNKNOWN;
   HSR = bits(32) UNKNOWN;
   SPSR_hyp = bits(32) UNKNOWN;

   DLR = bits(32) UNKNOWN;
   DSPSR = bits(32) UNKNOWN;
else                                        // Targeting EL2 using AArch64
   AArch64.MaybeZeroRegisterUppers();
   MaybeZeroSVEUppers(EL2);
   PSTATE.mRW = '0';
   PSTATE.SP = '1';
   PSTATE.EL = EL2;
   if HavePANExt() && SCTLR_EL2.SPAN == '0' && HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' then
      PSTATE.PAN = '1';
   if HaveUAOExt() then PSTATE.UAO = '0';

   else // Targeting EL2 using AArch64
      AArch64.MaybeZeroRegisterUppers();
      MaybeZeroSVEUppers(EL2);
      PSTATE.mRW = '0';
      PSTATE.SP = '1';
      PSTATE.EL = EL2;
      if HavePANExt() && SCTLR_EL2.SPAN == '0' && HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' then
         PSTATE.PAN = '1';
      if HaveUAOExt() then PSTATE.UAO = '0';
ELR_EL2 = bits(64) UNKNOWN;
ESR_EL2 = bits(64) UNKNOWN;
SPSR_EL2 = bits(64) UNKNOWN;

DLR_EL0 = bits(64) UNKNOWN;
DSPSR_EL0 = bits(64) UNKNOWN;

// SCTLR_EL2.IESB might be ignored in Debug state.
if HaveIESB() && SCTLR_EL2.IESB == '1' && !ConstrainUnpredictableBool(Unpredictable_IESBinDebug)
then
    SynchronizeErrors();

UpdateEDSCRFields(); // Update EDSCR PE state flags
F5.1.45   DCPS3

Debug Change PE State to EL3 allows the debugger to move the PE into EL3 from a lower Exception level or to a specific mode at the current Exception level.

DCPS3 is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL3 is not implemented.
- EDSCR.SDD is set to 1.

When the PE executes DCPS3:

- If EL3 is using AArch32, the PE enters Monitor mode and LR_mon, SPSR_mon, DLR and DSPSR become UNKNOWN. If DCPS3 is executed in Monitor mode, SCR.NS is cleared to 0.
- If EL3 is using AArch64, the PE enters EL3 using AArch64, selects SP_EL3, and ELR_EL3, ESR_EL3, SPSR_EL3, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

For more information on the operation of the DCPS<n> instructions, see DCPS<n> on page H2-10262.

T1

<table>
<thead>
<tr>
<th>[15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>1 1 1 1 0 1 1 1 0</th>
<th>0 0 0 0</th>
<th>0 0 0 0</th>
<th>0 0 0 0</th>
<th>1 1</th>
</tr>
</thead>
</table>

T1 variant

DCPS3

Decode for this encoding

if !HaveEL(EL3) then UNDEFINED;

Operation

if !Halted() || EDSCR.SDD == '1' then UNDEFINED;

if ELUsingAArch32(EL3) then

  from_secure = CurrentSecurityState() == SS_Secure;
  if PSTATE.E == M32_Monitor then SCR.NS = '0';
  AArch32.WriteMode(M32_Monitor);
  if HavePANExt() then
    if !from_secure then
      PSTATE.PAN = '0';
    elsif SCTLR.SPAN == '0' then
      PSTATE.PAN = '1';
      PSTATE.E = SCTLR.EE;
    PSTATE.E = SCTLR.EE;
  LR_mon = bits(32) UNKNOWN;
  SPSR_mon = bits(32) UNKNOWN;
  DLR = bits(32) UNKNOWN;
  DSPSR = bits(32) UNKNOWN;
else // Targeting EL3 using AArch64
  AArch64.MaybeZeroRegisterUppers();
  MaybeZeroSVEUppers(EL3);
  PSTATE.mRW = '0';
  PSTATE.SP = '1';
  PSTATE.EL = EL3;
if HaveUAOExt() then PSTATE.UAO = '0';

ELR_EL3 = bits(64) UNKNOWN;
ESR_EL3 = bits(64) UNKNOWN;
SPSR_EL3 = bits(64) UNKNOWN;

DLR_EL0 = bits(64) UNKNOWN;
DSPSR_EL0 = bits(64) UNKNOWN;

sync_errors = HaveIESB() && SCTLR_EL3.IESB == '1';
if HaveDoubleFaultExt() && SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' then
    sync_errors = TRUE;
// SCTLR_EL3.IESB might be ignored in Debug state.
if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
    sync_errors = FALSE;
if sync_errors then SynchronizeErrors();

UpdateEDSCRFields();                        // Update EDSCR PE state flags
F5.1.46 DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see Data Memory Barrier (DMB) on page E2-7182.

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1</td>
</tr>
</tbody>
</table>

A1 variant

DMB{<c>}{<q>} {<option>}

Decode for this encoding

// No additional decoding required

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1</td>
</tr>
</tbody>
</table>

T1 variant

DMB{<c>}{<q>} {<option>}

Decode for this encoding

// No additional decoding required

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<c> For encoding A1: see Standard assembler syntax fields on page F1-7228. Must be AL or omitted.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<option> Specifies an optional limitation on the barrier operation. Values are:

SY Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Can be omitted. This option is referred to as the full system barrier. Encoded as option = 0b1111.

ST Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. SYST is a synonym for ST. Encoded as option = 0b1110.

LD Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b1101.
ISH  Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = 0b1011.

ISHST  Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 0b1010.

ISHLD  Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b1001.

NSH  Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as option = 0b0111.

NSHST  Non-shareable is the required shareability domain, writes are the required access type both before and after the barrier instruction. Encoded as option = 0b0110.

NSHLD  Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b0101.

OSH  Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = 0b0011.

OSHST  Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 0b0010.

OSHLD  Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b0001.

For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) on page E2-7182. All other encodings of option are reserved. All unsupported and reserved options must execute as a full system DMB operation, but software must not rely on this behavior.

--- Note ---

The instruction supports the following alternative <option> values, but Arm recommends that software does not use these alternative values:

- SH as an alias for ISH.
- SHST as an alias for ISHST.
- UN as an alias for NSH.
- UNST as an alias for NSHST.

---

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  MBReqDomain domain;
  MBReqTypes types;
  case option of
    when '0001'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Reads;
    when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Writes;
    when '0011'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_All;
    when '0101'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Reads;
    when '0110'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Writes;
    when '0111'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_All;
    when '1001'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Reads;
    when '1010'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Writes;
    when '1011'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_All;
    when '1101'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Reads;
    when '1110'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Writes;
    otherwise    domain = MBReqDomain_FullSystem;      types = MBReqTypes_All;
  
if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
  if HCR.BSU == '11' then
domain = MBReqDomain_FullSystem;
if HCR.BSU == '10' && domain != MBReqDomain_FullSystem then
domain = MBReqDomain_OuterShareable;
if HCR.BSU == '01' && domain == MBReqDomain_Nonshareable then
domain = MBReqDomain_InnerShareable;

DataMemoryBarrier(domain, types);
F5.1.47   DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see *Data Synchronization Barrier (DSB)* on page E2-7182.

An AArch32 DSB instruction does not require the completion of any AArch64 TLB maintenance instructions, regardless of the nXS qualifier, appearing in program order before the AArch32 DSB.

A1

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0]
```

```
1 1 1 1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
```

// No additional decoding required

T1

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0]
```

```
1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

// No additional decoding required

**Notes for all encodings**

For more information about the *CONSTRAINED UNPREDICTABLE* behavior, see *Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

- `<c>`  For encoding A1: see *Standard assembler syntax fields on page F1-7228*. Must be AL or omitted.
  
  For encoding T1: see *Standard assembler syntax fields on page F1-7228*.

- `<q>`  See *Standard assembler syntax fields on page F1-7228*.

- `<option>`  Specifies an optional limitation on the barrier operation. Values are:
  
  SY  Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Can be omitted. This option is referred to as the full system barrier. Encoded as option = 0b1111.

  ST  Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. SYST is a synonym for ST. Encoded as option = 0b1110.
LD  Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b1101.

ISH  Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = 0b1011.

ISHST  Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 0b1010.

ISHLD  Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b1001.

NSH  Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as option = 0b0111.

NSHST  Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 0b0110.

NSHLD  Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b0101.

OSH  Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = 0b0011.

OSHST  Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 0b0010.

OSHLD  Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b0001.

For more information on whether an access is before or after a barrier instruction, see Data Synchronization Barrier (DSB) on page E2-7182. All other encodings of option are reserved, other than the values 0b0000 and 0b0100. All unsupported and reserved options must execute as a full system DSB operation, but software must not rely on this behavior.

--- Note ---

The value 0b0000 is used to encode SSBB and the value 0b0100 is used to encode PSSBB.

The instruction supports the following alternative <option> values, but Arm recommends that software does not use these alternative values:

- S as an alias for ISH.
- SHST as an alias for ISHST.
- UN as an alias for NSH.
- UNST as an alias for NSHST.

---

Operation for all encodings

```cpp
if ConditionPassed() then
    EncodingSpecificOperations();
    boolean nXS;
    if HaveFeatXS() && HaveFeatHCX() then
        nXS = (PSTATE.EL IN {EL0, EL1} && !ELUsingAArch32(EL2) && IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1');
    else
        nXS = FALSE;
    @MBReqDomain domain;
    @MBReqTypes types;
    case option of
        when '0001' domain = MBReqDomain_OuterShareable; types = MBReqTypes_Reads;
        when '0010' domain = MBReqDomain_OuterShareable; types = MBReqTypes_Writes;
        when '0011' domain = MBReqDomain_OuterShareable; types = MBReqTypes_All;
        when '0101' domain = MBReqDomain_Nonshareable; types = MBReqTypes_Reads;
```
when '0110'  domain = MBReqDomain_Nonshareable; types = MBReqTypes_Writes;
when '0111'  domain = MBReqDomain_Nonshareable; types = MBReqTypes_All;
when '1001'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Reads;
when '1010'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Writes;
when '1011'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_All;
when '1101'  domain = MBReqDomain_FullSystem;  types = MBReqTypes_Reads;
when '1110'  domain = MBReqDomain_FullSystem;  types = MBReqTypes_Writes;
otherwise
    assert !(option IN {'0x00'});
    domain = MBReqDomain_FullSystem;  types = MBReqTypes_All;

if PSTATE.EL IN {EL0, EL1} & EL2Enabled() then
    if HCR.BSU == '11' then
domain = MBReqDomain_FullSystem;
    if HCR.BSU == '10' && domain != MBReqDomain_FullSystem then
domain = MBReqDomain_OuterShareable;
    if HCR.BSU == '01' && domain == MBReqDomain_Nonshareable then
domain = MBReqDomain_InnerShareable;

DataSynchronizationBarrier(domain, types, nXS);
F5.1.48 EOR, EORS (immediate)

Bitwise Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the EORS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The EOR variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- The EORS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

| 31 | 28 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | | | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | S | Rn | Rd | | | | imm12 |
| cond |

**EOR variant**

Applies when $S == 0$.

EOR{<c>}{<q>} {<Rd>,} <Rn>, #<const>

**EORS variant**

Applies when $S == 1$.

EORS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

**Decode for all variants of this encoding**

$$d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ \text{setflags} = (S == '1');$$

$$\text{imm32, carry} = A32\text{ExpandImm}_C(\text{imm12}, \text{PSTATE.C});$$

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>S</td>
<td>Rn</td>
<td>0</td>
<td>imm3</td>
<td>Rd</td>
<td></td>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**EOR variant**

Applies when $S == 0$.

EOR{<c>}{<q>} {<Rd>,} <Rn>, #<const>
**EORS variant**

Applies when \( S = 1 \) \&\& \( Rd \neq 1111 \).

\[
\text{EORS}\{\langle c \rangle}\{\langle q \rangle}\{\langle Rd \rangle,}\{\langle Rn \rangle,\#\langle \text{const} \rangle}
\]

**Decode for all variants of this encoding**

\[
\text{if } \text{Rd} = '1111' \&\& \text{S} = '1' \text{ then SEE "TEQ (immediate)";}
\]
\[
d = \text{UInt(Rd)}; \quad n = \text{UInt(Rn)}; \quad \text{setflags} = (\text{S} = '1');
\]
\[
(\text{imm32}, \text{carry}) = \text{T32ExpandImm}_C(\langle \text{imm3:imm8}, \text{PSTATE.C} \rangle);
\]
\[
\text{if (d == 15 \&\& !setflags) \| n == 15 then UNPREDICTABLE;}
\]
\[
// \text{Armv8-A removes UNPREDICTABLE for R13}
\]

**Notes for all encodings**

For more information about the CONstrained UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

\(<c>\)  See Standard assembler syntax fields on page F1-7228.

\(<q>\)  See Standard assembler syntax fields on page F1-7228.

\(<\text{Rd}>\)  For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as \(<\text{Rn}>\). Arm deprecates using the PC as the destination register, but if the PC is used:

- For the EOR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.

- For the EORS variant, the instruction performs an exception return, that restores PSTATE from SPSR_\langle current\_mode\rangle.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as \(<\text{Rn}>\).

\(<\text{Rn}>\)  For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

\(<\text{const}>\)  For encoding A1: an immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values.

For encoding T1: an immediate value. See Modified immediate constants in T32 instructions on page F1-7242 for the range of values.

**Operation for all encodings**

\[
\text{if ConditionPassed() then}
\]
\[
\text{EncodingSpecificOperations();}
\]
\[
\text{result} = \text{R[n]} \text{ EOR imm32};
\]
\[
\text{if d == 15 then} \quad \text{// Can only occur for A32 encoding}
\]
\[
\text{if setflags then}
\]
\[
\text{ALUExceptionReturn(result);}\]
\[
\text{else}
\]
\[
\text{ALUWritePC(result);}\]
\[
}\]
\[
\text{else}
\]
\[
\text{R[d] = result;}\]
\[
\text{if setflags then}
\]
\[
\text{PSTATE.N = result<31>;}\]
\[
\text{PSTATE.Z = IsZeroBit(result);}\]
PSTATE.C = carry;
// PSTATE.V unchanged

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.49  **EOR, EORS (register)**

Bitwise Exclusive OR (register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the EORS variant of the instruction updates the condition flags based on the result.

The field descriptions for \(<Rd>\) identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The EOR variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-7133.
- The EORS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state* on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

### A1

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| \(=1111\) | 0 | 0 | 0 | 0 | 0 | 1 | S | Rn | Rd | \(\text{imm5}\) | stype | 0 | Rm |
| **cond**

**EOR, rotate right with extend variant**

Applies when \(S == 0 \&\& \text{imm5} == 00000 \&\& \text{stype} == 11\).

\[
\text{EOR}\{<c>\}\{<q>\} \{<Rd>,\} \{<Rn>, <Rm>, \text{RRX}\}
\]

**EOR, shift or rotate by value variant**

Applies when \(S == 0 \&\& !(\text{imm5} == 00000 \&\& \text{stype} == 11)\).

\[
\text{EOR}\{<c>\}\{<q>\} \{<Rd>,\} \{<Rn>, <Rm> \{, \text{<shift> #<amount>}\}
\]

**EORS, rotate right with extend variant**

Applies when \(S == 1 \&\& \text{imm5} == 00000 \&\& \text{stype} == 11\).

\[
\text{EORS}\{<c>\}\{<q>\} \{<Rd>,\} \{<Rn>, <Rm>, \text{RRX}\}
\]

**EORS, shift or rotate by value variant**

Applies when \(S == 1 \&\& !(\text{imm5} == 00000 \&\& \text{stype} == 11)\).

\[
\text{EORS}\{<c>\}\{<q>\} \{<Rd>,\} \{<Rn>, <Rm> \{, \text{<shift> #<amount>}\}
\]

**Decode for all variants of this encoding**

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \ \text{setflags} = (S == '1'); \ (\text{shift_t, shift_n}) = \text{DecodeImmShift}(\text{stype, imm5});
\]
T1

\[ \begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c} & & & & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ \hline & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & Rm & Rdn \end{array} \]

**T1 variant**

EOR\{<q>\} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
EORS\{<q>\} {<Rdn>,} <Rdn>, <Rm> // Outside IT block

**Decode for this encoding**

\[
\begin{align*}
& d = \text{UInt}(\text{Rdn}); \ n = \text{UInt}(\text{Rdn}); \ m = \text{UInt}(\text{Rm}); \ \text{setflags} = !\text{InITBlock}(); \\
& (\text{shift}_t, \ \text{shift}_n) = (\text{SRType}_{\text{LSL}}, 0);
\end{align*}
\]

T2

\[ \begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c} & & & & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ \hline & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & S & Rn & (0) & \text{imm3} & Rd & \text{imm2} & \text{stype} & Rm \end{array} \]

**EOR, rotate right with extend variant**

Applies when \( S = 0 \) \&\& \( \text{imm3} = 000 \) \&\& \( \text{imm2} = 00 \) \&\& \( \text{stype} = 11 \).

EOR\{<c>\}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

**EOR, shift or rotate by value variant**

Applies when \( S = 0 \) \&\& \( (!\text{imm3} = 000 \) \&\& \( \text{imm2} = 00 \) \&\& \( \text{stype} = 11 \)).

EOR\{<c>\}.W {<Rd>,} <Rn> // Inside IT block, and \( <Rd>, <Rn>, <Rm> \) can be represented in T1
EOR\{<c>\}{<q>} {<Rd>,} <Rn>, <Rm> \{, <shift> \#<amount>\}

**EORS, rotate right with extend variant**

Applies when \( S = 1 \) \&\& \( \text{imm3} = 000 \) \&\& \( \text{Rd} != 1111 \) \&\& \( \text{imm2} = 00 \) \&\& \( \text{stype} = 11 \).

EORS\{<c>\}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

**EORS, shift or rotate by value variant**

Applies when \( S = 1 \) \&\& \( (!\text{imm3} = 000 \) \&\& \( \text{imm2} = 00 \) \&\& \( \text{stype} = 11 \)) \&\& \( \text{Rd} != 1111 \).

EORS\{<c>\}.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and \( <Rd>, <Rn>, <Rm> \) can be represented in T1
EORS\{<c>\}{<q>} {<Rd>,} <Rn>, <Rm> \{, <shift> \#<amount>\}

**Decode for all variants of this encoding**

\[
\begin{align*}
& \text{if Rd} = \text{'1111'} \&\& \text{S} = \text{'1'} \text{ then SEE "TEQ (register)";}
& d = \text{UInt}(\text{Rd}); \ n = \text{UInt}(\text{Rn}); \ m = \text{UInt}(\text{Rm}); \ \text{setflags} = (\text{S} = \text{'1'}); \\
& (\text{shift}_t, \ \text{shift}_n) = \text{DecideImmShift}(\text{stype}, \text{imm3};\text{imm2}); \\
& \text{if } (d = 15 \&\& \text{!setflags}) \| \ n = 15 \| \ m = 15 \text{ then UNPREDICTABLE;}
\end{align*}
\]

// Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:
- For the EOR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- For the EORS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
- LSL when stype = 00
- LSR when stype = 01
- ASR when stype = 10
- ROR when stype = 11

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.
For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

In T32 assembly:
- Outside an IT block, if EORS <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though EORS <Rd>, <Rn> had been written
- Inside an IT block, if EOR<c> <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though EOR<c> <Rd>, <Rn> had been written.
To prevent either of these happening, use the .W qualifier.

Operation for all encodings

if ConditionPassed() then
EncodingSpecificOperations();
(shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
result = R[n] EOR shifted;
if d == 15 then // Can only occur for A32 encoding
if setflags then
  ALUExceptionReturn(result);
else
  ALUWritePC(result);
else
  R[d] = result;
  if setflags then
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.50   EOR, EORS (register-shifted register)

Bitwise Exclusive OR (register-shifted register) performs a bitwise Exclusive OR of a register value and a
register-shifted register value. It writes the result to the destination register, and can optionally update the condition
flags based on the result.

A1

\[
\begin{array}{cccccccc}
| & | & | & | & | & | & | & | \\
\text{cond} & ! & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & S & Rn & Rd & Rs & 0 & stype & 1 & Rm
\end{array}
\]

Flag setting variant

Applies when \( S == 1 \).

\[\text{EORS}\{<c>\}{<q>} \{<Rd>,} \{<Rn>, \langle Rm>, <\text{shift} \rangle <Rs}\]

Not flag setting variant

Applies when \( S == 0 \).

\[\text{EOR}\{<c>\}{<q>} \{<Rd>,} \{<Rn>, \langle Rm>, <\text{shift} \rangle <Rs}\]

Decode for all variants of this encoding

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \ s = \text{UInt}(Rs);
\text{setflags} = (S == '1'); \ \text{shift}_t = \text{DecodeRegShift}(stype);
\text{if} \ d == 15 || n == 15 || m == 15 || s == 15 \text{ then UNPREDICTABLE;}
\]

Notes for all encodings

For more information about the \text{CONSTRAINED UNPREDICTABLE} behavior, see Appendix K1 \text{Architectural Constraints on UNPREDICTABLE Behaviors}.

Assembler symbols

\(<c>\) \quad \text{See \textit{Standard assembler syntax fields} on page F1-7228.}
\(<q>\) \quad \text{See \textit{Standard assembler syntax fields} on page F1-7228.}
\(<Rd>\) \quad \text{Is the general-purpose destination register, encoded in the "Rd" field.}
\(<Rn>\) \quad \text{Is the first general-purpose source register, encoded in the "Rn" field.}
\(<Rm>\) \quad \text{Is the second general-purpose source register, encoded in the "Rm" field.}
\(<\text{shift}>\) \quad \text{Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:}
\text{LSL} \quad \text{when stype} = 00
\text{LSR} \quad \text{when stype} = 01
\text{ASR} \quad \text{when stype} = 10
\text{ROR} \quad \text{when stype} = 11
\(<Rs>\) \quad \text{Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.}
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] EOR shifted;
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
**F5.1.51 ERET**

Exception Return.

The PE branches to the address held in the register holding the preferred return address, and restores PSTATE from SPSR_<current_mode>.

The register holding the preferred return address is:

- ELR_hyp, when executing in Hyp mode.
- LR, when executing in a mode other than Hyp mode, User mode, or System mode.

The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.

Exception Return is CONSTRAINED UNPREDICTABLE in User mode and System mode.

In Debug state, the T1 encoding of ERET executes the DRPS operation.

### A1

| 31 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8 |7 6 5 4 |3 2 1 0 |1=1111 0 0 0 1 0 1 1 0 |
| cond | 0 0 0 |0 |0 |0 |0 |0 |0 |0 |1 1 0 (1) (1) (1) |

**A1 variant**

ERET{<c>}{<q>}

**Decode for this encoding**

// No additional decoding required

### T1

| 15 14 13 12|11 10 9 8 |7 6 5 4 |3 2 1 0 |15 14 13 12|11 10 9 8 |7 6 5 4 |3 2 1 0 |
| 1 1 1 0 0 1 1 1 0 1 1 1 0 1 0 0 0 1 0 (1) (1) (1) |

**T1 variant**

ERET{<c>}{<q>}

**Decode for this encoding**

if InITBlock() && !LastInITBlock() then UNPREDICTABLE;

### Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

### Assembler symbols

- <c> See Standard assembler syntax fields on page F1-7228.
- <q> See Standard assembler syntax fields on page F1-7228.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
if !Halted() then
    if PSTATE.M IN {M32_User, M32_System} then
        UNPREDICTABLE; // UNDEFINED or NOP
    else
        new_pc_value = if PSTATE.EL == EL2 then ELR_hyp else R[14];
        AArch32.ExceptionReturn(new_pc_value, SPSR[]);
    else
        // Perform DRPS operation in Debug state
        if PSTATE.M == M32_User then
            UNDEFINED;
        elsif PSTATE.M == M32_System then
            UNPREDICTABLE; // UNDEFINED or NOP
        else
            SynchronizeContext();
            DebugRestorePSR();

CONSTRANGED UNPREDICTABLE behavior

If PSTATE.M IN {M32_User, M32_System}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
F5.1.52  ESB

Error Synchronization Barrier is an error synchronization event that might also update DISR and VDISR. This instruction can be used at all Exception levels and in Debug state.

In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. See Error Synchronization Barrier in the ARM(R) Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for Armv8-A architecture profile.

If the RAS Extension is not implemented, this instruction executes as a NOP.

A1

(FEAT_RAS)

```
0 0 1 1 0 0 0 0 1 1 1 0 1 0 1 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
```

A1 variant

ESB{<c>}{<q>}

Decode for this encoding

if !HaveRASExt() then EndOfInstruction();  // Instruction executes as NOP
if cond != '1110' then UNPREDICTABLE;      // ESB must be encoded with AL condition

CONstrained UNPREDICTABLE behavior

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

(FEAT_RAS)

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

T1 variant

ESB{<c>}{<q>}

Decode for this encoding

if !HaveRASExt() then EndOfInstruction();  // Instruction executes as NOP
if InITBlock() then UNPREDICTABLE;
**CONSTRAINED UNPREDICTABLE behavior**

If InITBlock(), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction executes unconditionally.
- The instruction executes conditionally.

**Assembler symbols**

<> See *Standard assembler syntax fields on page F1-7228*.

<q> See *Standard assembler syntax fields on page F1-7228*.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    SynchronizeErrors();
    AArch32.ESB0peration();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch32.vESB0peration();
    TakeUnmaskedSErrorInterrupts();
```
F5.1.53  HLT

Halting breakpoint causes a software breakpoint to occur.

Halting breakpoint is always unconditional, even inside an IT block.

A1

\[
\begin{array}{ccccccccc}
\end{array}
\]

\[
=1111 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\text{cond} & \text{imm12} & 0 & 1 & 1 & 1 & \text{imm4}
\]

\textbf{A1 variant}

\texttt{HLT}{}{<q>} {#}<imm>

\textbf{Decode for this encoding}

\begin{verbatim}
if EDSRC.HDE == '0' || !HaltingAllowed() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE; // HLT must be encoded with AL condition
\end{verbatim}

\textbf{CONSTRAINED UNPREDICTABLE behavior}

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as \texttt{NOP}.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

\[
\begin{array}{cccccccccc}
| & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & | & 0 |
\end{array}
\]

\[
1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & \text{imm6}
\]

\textbf{T1 variant}

\texttt{HLT}{}{<q>} {#}<imm>

\textbf{Decode for this encoding}

\begin{verbatim}
if EDSRC.HDE == '0' || !HaltingAllowed() then UNDEFINED;
\end{verbatim}

\textbf{Notes for all encodings}

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see \textit{Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors}.

\textbf{Assembler symbols}

\texttt{<q>} See \textit{Standard assembler syntax fields} on page F1-7228. An HLT instruction must be unconditional.
For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value is for assembly and disassembly only. It is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.

For encoding T1: is a 6-bit unsigned immediate, in the range 0 to 63, encoded in the "imm6" field. This value is for assembly and disassembly only. It is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.

**Operation for all encodings**

```
EncodingSpecificOperations();
boolean is_async = FALSE;
Halt(DebugHalt_HaltInstruction, is_async);
```
F5.1.54   HVC

Hypervisor Call causes a Hypervisor Call exception. For more information, see Hypervisor Call (HVC) exception on page G1-8970. Software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is UNDEFINED:

- When EL3 is implemented and using AArch64, and SCR_EL3.HCE is set to 0.
- In Non-secure EL1 modes when EL3 is implemented and using AArch32, and SCR.HCE is set to 0.
- When EL3 is not implemented and either HCR_EL2.HCD is set to 1 or HCR.HCD is set to 1.
- When EL2 is not implemented.
- In Secure state, if EL2 is not enabled in the current Security state.
- In User mode.

The HVC instruction is CONSTRAINED UNPREDICTABLE in Hyp mode when EL3 is implemented and using AArch32, and SCR.HCE is set to 0.

On executing an HVC instruction, the HSR reports the exception as a Hypervisor Call exception, using the EC value 0x12, and captures the value of the immediate argument, see Use of the HSR on page G5-9265.

A1

```
[31  28 27 26 25 24 23 22 21 20 19] | | 8 7 6 5 4 3 0 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111 0 0 1 0 1 0 0</td>
<td>imm12</td>
<td>0 1 1 1</td>
</tr>
</tbody>
</table>
```

```
cond
```

A1 variant

HVC{<q>} {#}<imm16>

Decode for this encoding

if cond != '1110' then UNPREDICTABLE;
imm16 = imm12:imm4;

CONSTRANGED UNPREDICTABLE behavior

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

```
[15 14 13 12 11 10 9 8 7 6 5 4 3 0] | | 0 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 1 1 1 1 1 0</td>
<td>imm4</td>
<td>1 0</td>
</tr>
</tbody>
</table>
```

T1 variant

HVC{<q>} {#}<imm16>
Decode for this encoding

\[ \text{imm16} = \text{imm4:imm12}; \]
\[ \text{if InITBlock()} \text{ then UNPREDICTABLE;} \]

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<q>
See Standard assembler syntax fields on page F1-7228. An HVC instruction must be unconditional.

<imm16>
For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value is for assembly and disassembly only. It is reported in the HSR but otherwise is ignored by hardware. An HVC handler might interpret imm16, for example to determine the required service.

For encoding T1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. This value is for assembly and disassembly only. It is reported in the HSR but otherwise is ignored by hardware. An HVC handler might interpret imm16, for example to determine the required service.

Operation for all encodings

EncodingSpecificOperations();
\[ \text{if PSTATE.EL IN \{EL0, EL3\} || !EL2Enabled()} \text{ then UNDEFINED;} \]

\[ \text{bit hvc_enable;} \]
\[ \text{if HaveEL(EL3) then} \]
\[ \text{if ELUsingAArch32(EL3) && SCR.HCE == '0' && PSTATE.EL == EL2 then} \]
\[ \text{UNPREDICTABLE;} \]
\[ \text{else} \]
\[ \text{hvc_enable = SCR_GEN[].HCE;} \]
\[ \text{else} \]
\[ \text{hvc_enable = if ELUsingAAArch32(EL2) then NOT(HCR.HCD) else NOT(HCR_EL2.HCD);} \]
\[ \text{if hvc_enable == '0' then} \]
\[ \text{UNDEFINED;} \]
\[ \text{else} \]
\[ \text{AArch32.CallHypervisor(imm16);} \]

CONSTRAINED UNPREDICTABLE behavior

If ELUsingAAArch32(EL3) && SCR.HCE == '0' && PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
### F5.1.55 ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see *Instruction Synchronization Barrier (ISB)* on page E2-7181.

#### A1

**Decode for this encoding**

// No additional decoding required

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8 |7 6 5 4 |3 0 |]
  1 1 1 1 0 1 0 1 1 1 1 (1)(1)(1)(1)(1)(1)(1)(1)(0)(0)(0)(0) 0 1 1 0 0 option
```

**A1 variant**

ISB{<c>}{<q>}{<option>}

**Decode for this encoding**

// No additional decoding required

```
[15 14 13 12|11 10 9 8 |7 6 5 4 |3 0 |]
  1 1 1 1 0 1 1 1 0 1 1 (1)(1)(1)(1)(0)(0)(0)(1)(1)(1) 0 1 1 0 0 option
```

**T1 variant**

ISB{<c>}{<q>}{<option>}

**Decode for this encoding**

// No additional decoding required

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

- `<c>`
  - For encoding A1: see *Standard assembler syntax fields on page F1-7228*. Must be AL or omitted.
  - For encoding T1: see *Standard assembler syntax fields on page F1-7228*.

- `<q>`
  - See *Standard assembler syntax fields on page F1-7228*.

- `<option>`
  - Specifies an optional limitation on the barrier operation. Values are:
    - **SY**
      - Full system barrier operation, encoded as option = 0b1111. Can be omitted.
    - All other encodings of option are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    InstructionSynchronizationBarrier();
F5.1.56   IT

If-Then makes up to four following instructions (the IT block) conditional. The conditions for the instructions in the IT block are the same as, or the inverse of, the condition the IT instruction specifies for the first instruction in the block.

The IT instruction itself does not affect the condition flags, but the execution of the instructions in the IT block can change the condition flags.

16-bit instructions in the IT block, other than CMP, CMN and TST, do not set the condition flags. An IT instruction with the AL condition can change the behavior without conditional execution.

The architecture permits exception return to an instruction in the IT block only if the restoration of the CPSR restores PSTATE.IT to a state consistent with the conditions specified by the IT instruction. Any other exception return to an instruction in an IT block is UNPREDICTABLE. Any branch to a target instruction in an IT block is not permitted, and if such a branch is made it is UNPREDICTABLE what condition is used when executing that target instruction and any subsequent instruction in the IT block.

Many uses of the IT instruction are deprecated for performance reasons, and an implementation might include ITD controls that can disable those uses of IT, making them UNDEFINED.

For more information see Conditional execution on page F1-7229 and Conditional instructions on page F2-7257. The first of these sections includes more information about the ITD controls.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1 1</td>
<td>1 1 1 0</td>
<td>l=0000</td>
</tr>
</tbody>
</table>

T1 variant

IT{<>{<y>{<z>}}}{<q>} <cond>

Decode for this encoding

if mask == '0000' then SEE "Related encodings";
if firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1) then UNPREDICTABLE;
if InITBlock() then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The '1111' condition is treated as being the same as the '1110' condition, meaning always, and the ITSTATE state machine is progressed in the same way as for any other cond_base value.

Notes for all encodings

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Related encodings: Miscellaneous 16-bit instructions on page F3-7303.
Assembler symbols

<x> The condition for the second instruction in the IT block. If omitted, the "mask" field is set to 0b1000. If present it is encoded in the "mask[3]" field:

\[
\begin{align*}
\top & \text{ firstcond[0]} \\
\bot & \text{ NOT firstcond[0]}
\end{align*}
\]

<y> The condition for the third instruction in the IT block. If omitted and <x> is present, the "mask[2:0]" field is set to 0b100. If <y> is present it is encoded in the "mask[2]" field:

\[
\begin{align*}
\top & \text{ firstcond[0]} \\
\bot & \text{ NOT firstcond[0]}
\end{align*}
\]

<z> The condition for the fourth instruction in the IT block. If omitted and <y> is present, the "mask[1:0]" field is set to 0b10. If <z> is present, the "mask[0]" field is set to 1, and it is encoded in the "mask[1]" field:

\[
\begin{align*}
\top & \text{ firstcond[0]} \\
\bot & \text{ NOT firstcond[0]}
\end{align*}
\]

<q> See Standard assembler syntax fields on page F1-7228.

<cond> The condition for the first instruction in the IT block, encoded in the "firstcond" field. See Table F1-1 on page F1-7229 for the range of conditions available, and the encodings.

The conditions specified in an IT instruction must match those specified in the syntax of the instructions in its IT block. When assembling to A32 code, assemblers check IT instruction syntax for validity but do not generate assembled instructions for them. See Conditional instructions on page F2-7257.

Operation

EncodingSpecificOperations();
AArch32.CheckITEnabled(mask);
PSTATE.IT<7:0> = firstcond:mask;
ShouldAdvanceIT = FALSE;
F5.1.57 LDA

Load-Acquire Word loads a word from memory and writes it to a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release on page E2-7186.

For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For information about memory accesses see Memory accesses on page F1-7233.

A1

\[
\begin{array}{cccccccccccccccccc}
\end{array}
\]

\[
| 1=1111 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & \text{Rn} & \text{Rt} & (1)(1)(1) & 0 & 0 & 1 & 0 & 1 |
\]

\text{cond}

A1 variant

LDA{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding
\[
t = \text{UInt}(\text{Rt}); \quad n = \text{UInt}(\text{Rn});
\]
if \( t == 15 \) \( \text{||} \) \( n == 15 \) then UNPREDICTABLE;

T1

\[
\begin{array}{cccccccccccccccccc}
| & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 16 & 15 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 |
\end{array}
\]

\[
| 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & \text{Rn} & \text{Rt} & (1)(1)(1)(1) & 1 & 0 & 1 & 0 & (1)(1)(1)(1) |
\]

T1 variant

LDA{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding
\[
t = \text{UInt}(\text{Rt}); \quad n = \text{UInt}(\text{Rn});
\]
if \( t == 15 \) \( \text{||} \) \( n == 15 \) then UNPREDICTABLE;

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\) See Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<\text{Rt}>\) Is the general-purpose register to be transferred, encoded in the "Rt" field.

\(<\text{Rn}>\) Is the general-purpose base register, encoded in the "Rn" field.
Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations();
   address = R[n];
   R[t] = MemO[address, 4];

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.58   LDAB

Load-Acquire Byte loads a byte from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page E2-7186.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-7211. For information about memory accesses see *Memory accesses on page F1-7233*.

**A1**

```plaintext
[31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 6 5 4 3 2 1 0]
```

<table>
<thead>
<tr>
<th>t</th>
<th>0 0 1 1 0 1</th>
<th>Rn</th>
<th>Rt</th>
<th>1 0 0 1 1 1 1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>UNPREDICTABLE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**A1 variant**

LDAB{<c>}{<q>} <Rt>, [<Rn>]

**Decode for this encoding**

```plaintext
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

**T1**

```plaintext
[15 14 13 12 11 10 9 8 7 6 5 4 3 0 15 12 11 10 9 8 7 6 5 4 3 2 1 0]
```

<table>
<thead>
<tr>
<th>t</th>
<th>1 1 0 0 1 1 0 1</th>
<th>Rn</th>
<th>Rt</th>
<th>1 0 0 0 1 1 1 1 1 1</th>
</tr>
</thead>
</table>

**T1 variant**

LDAB{<c>}{<q>} <Rt>, [<Rn>]

**Decode for this encoding**

```plaintext
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields* on page F1-7228.
- `<q>` See *Standard assembler syntax fields* on page F1-7228.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    R[t] = ZeroExtend(MemO[address, 1], 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.59   LDAEX

Load-Acquire Exclusive Word loads a word from memory, writes it to a register and:

• If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.

• Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release on page E2-7186.

For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For information about memory accesses see Memory accesses on page F1-7233.

A1

```
| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| l=1111 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | Rn | Rn |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
```

**A1 variant**

LDAEX(<c>{<q}> <Rt>, [<Rn>])

**Decode for this encoding**

\[ t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \]
\[ \text{if } t = 15 \quad \text{||} \quad n = 15 \text{ then UNPREDICTABLE;} \]

T1

```
| 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  0 | 15 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | Rn | Rn |     |     |     |     |     |     |     |     |     |     |     |     |     |
```

**T1 variant**

LDAEX(<c>{<q}> <Rt>, [<Rn>])

**Decode for this encoding**

\[ t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \]
\[ \text{if } t = 15 \quad \text{||} \quad n = 15 \text{ then UNPREDICTABLE;} \]

**Notes for all encodings**

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 4);
    R[t] = MemO[address, 4];

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.60   LDAEXB

Load-Acquire Exclusive Byte loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

• If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.

• Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release on page E2-7186.

For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For information about memory accesses see Memory accesses on page F1-7233.

A1

```
[31  28 27  26  25  24 23  22  21  20 19 16 15 12 11 10  9  8  7  6  5  4  3  2  1  0 ]
```

```
| !=1111 0 0 0 1 1 1 0 1 | Rn | Rt | (1)(1)(1)(1) |
```

cond

A1 variant

LDAEXB{<c>}{<q>} <Rt>, [Rn]

Decode for this encoding

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn);
\]

if \( t == 15 \) \( || \) \( n == 15 \) then UNPREDICTABLE;

T1

```
[15 14 13 12 11 10  9  8  7  6  5  4  3  0  15 12 11 10  9  8  7  6  5  4  3  2  1  0 ]
```

```
| 1 1 1 0 1 0 0 1 1 0 1 | Rn | Rt | (1)(1)(1)(1) |
```

T1 variant

LDAEXB{<c>}{<q>} <Rt>, [Rn]

Decode for this encoding

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn);
\]

if \( t == 15 \) \( || \) \( n == 15 \) then UNPREDICTABLE;

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\) See Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<Rt>\) Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn> Is the general-purpose base register, encoded in the "Rn" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 1);
    R[t] = ZeroExtend(MemO[address, 1], 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.61 LDAEXD

Load-Acquire Exclusive Doubleword loads a doubleword from memory, writes it to two registers and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also acts as a barrier instruction with the ordering requirements described in Load-Acquire, Store-Release on page E2-7186.

For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For information about memory accesses see Memory accesses on page F1-7233.

A1

| 31 28|27 26 25 24|23 22 21 20|19 16|15 11 10 9 8 7 6 5 4 3 2 1 0 |
| l=1111 | 0 0 0 1 1 | 0 1 1 | Rn | Rt | (1)|1|1|0 0 1 0 1 1 | (1)|1|1|1|1|1|

cond

**A1 variant**

LDAEXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]

**Decode for this encoding**

\[
\begin{align*}
t &= \text{UInt}(Rt); \\
t2 &= t + 1; \\
n &= \text{UInt}(Rn);
\end{align*}
\]

if \(Rt<0> == '1'\) \(\text{or} \ t2 == 15 \text{ or} \ n == 15\) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \(Rt<0> == '1'\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: \(t<0> = '0'\).
- The instruction executes with the additional decode: \(t2 = t\).
- The instruction executes as described, with no change to its behavior and no additional side effects.

If \(Rt == '1110'\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction is handled as described in Using R15 by instruction on page K1-11563.

T1

| 15 14 13 12|11 10 9 8 7 6 5 4 3 0 |15 12 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 1 1 0 1 0 0 0 1 0 1 0 1 | Rn | Rt | Rn2 | (1)|1|1|1|1|1|1|

**T1 variant**

LDAEXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]
Decode for this encoding

\[ t = \text{UInt}(Rt); \quad t2 = \text{UInt}(Rt2); \quad n = \text{UInt}(Rn); \]
\[ \text{if } t == 15 \quad || \quad t2 == 15 \quad || \quad t == t2 \quad || \quad n == 15 \quad \text{then UNPREDICTABLE}; \]

**CONSTRAINED UNPREDICTABLE behavior**

If \( t == t2 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The load instruction executes but the destination register takes an UNKNOWN value.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields on page F1-7228*.
- `<q>` See *Standard assembler syntax fields on page F1-7228*.
- `<Rt>` For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. <Rt> must be even-numbered and not R14.
- `<Rt>` For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rt2>` For encoding A1: is the second general-purpose register to be transferred. <Rt2> must be \( R(t+1) \).
- `<Rt2>` For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

**Operation for all encodings**

\[ \text{if ConditionPassed()} \text{ then} \]
\[ \quad \text{EncodingSpecificOperations();} \]
\[ \quad \text{address} = R[n]; \]
\[ \quad \text{AArch32.SetExclusiveMonitors(address, 8);} \]
\[ \quad \text{value} = \text{MemO}[\text{address}, 8]; \]
\[ \quad // \text{Extract words from 64-bit loaded value such that } R[t] \text{ is} \]
\[ \quad // \text{loaded from address and } R[t2] \text{ from address+4.} \]
\[ \quad R[t] = \text{if BigEndian(AccType.ORDERED) then value<63:32> else value<31:0>;} \]
\[ \quad R[t2] = \text{if BigEndian(AccType.ORDERED) then value<31:0> else value<63:32>;} \]

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.62 LDAEXH

Load-Acquire Exclusive Halfword loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.

- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release on page E2-7186*.

For more information about support for shared memory see *Synchronization and semaphores on page E2-7211*. For information about memory accesses see *Memory accesses on page F1-7233*.

### A1

![Instruction Format](image1)

### A1 variant

LDAEXH(<c>{<q>} <Rt>, [<Rn>]

**Decode for this encoding**

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn);
\]

\[
\text{if } t == 15 \land n == 15 \text{ then UNPREDICTABLE;}
\]

### T1

![Instruction Format](image2)

### T1 variant

LDAEXH(<c>{<q>} <Rt>, [<Rn>]

**Decode for this encoding**

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn);
\]

\[
\text{if } t == 15 \land n == 15 \text{ then UNPREDICTABLE;}
\]

### Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors*.

### Assembler symbols

- `<c>` See *Standard assembler syntax fields on page F1-7228.*

- `<q>` See *Standard assembler syntax fields on page F1-7228.*

- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn> Is the general-purpose base register, encoded in the "Rn" field.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 2);
    R[t] = ZeroExtend(MemO[address, 2], 32);

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.63   LDAH

Load-Acquire Halfword loads a halfword from memory, zero-extends it to form a 32-bit word and writes it to a
register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on
page E2-7186.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-7211. For
information about memory accesses see *Memory accesses* on page F1-7233.

A1

```
|31|28|27|26|25|24|23|22|21|20|19|16|15|12|11|10|9|8|7|6|5|4|3|2|1|0 |
|l=1111|0|0|0|1|1|1|1|Rn|Rt|1|1|1|1|0|0|0|1|1|0|1|0|0|1|1|1|1|1|
```

**A1 variant**

LDAH{<c>}{<q>} <Rt>, [<Rn>]

**Decode for this encoding**

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

```
|15|14|13|12|11|10|9|8|7|6|5|4|3|0|15|12|11|10|9|8|7|6|5|4|3|2|1|0 |
|1|1|1|0|1|0|0|1|1|0|1|Rn|Rt|1|1|1|1|1|1|1|1|1|1|1|1|
```

**T1 variant**

LDAH{<c>}{<q>} <Rt>, [<Rn>]

**Decode for this encoding**

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Appendix K1 Architectural
Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields* on page F1-7228.
- `<q>` See *Standard assembler syntax fields* on page F1-7228.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    R[t] = ZeroExtend(MemO[address, 2], 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
### F5.1.64 LDC (immediate)

Load data to System register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to the `DBGDTRTXintSystem` register. It can use offset, post-indexed, pre-indexed, or unindexed addressing. For information about memory accesses see Memory accesses on page F1-7233.

In an implementation that includes EL2, the permitted LDC access to `DBGDTRTXint` can be trapped to Hyp mode, meaning that an attempt to execute an LDC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see Trapping general Non-secure System register accesses to debug registers on page G1-9027.

For simplicity, the LDC pseudocode does not show this possible trap to Hyp mode.

#### A1

```
<table>
<thead>
<tr>
<th>31 28 27 26 25 24</th>
<th>23 22 21 20</th>
<th>19</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9 8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=111</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>P</td>
<td>U</td>
<td>W</td>
</tr>
<tr>
<td>imm8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>cond</td>
<td>Rn</td>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

#### Offset variant

Applies when `P == 1 && W == 0`.

```
LDC{<c>}{<q>} p14, c5, [<Rn>{, #{+/-}<imm>}]
```

#### Post-indexed variant

Applies when `P == 0 && W == 1`.

```
LDC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>
```

#### Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDC{<c>}{<q>} p14, c5, [<Rn>,#{+/-}imm]
```

#### Unindexed variant

Applies when `P == 0 && U == 1 && W == 0`.

```
LDC{<c>}{<q>} p14, c5, [<Rn>,<option>]
```

#### Decode for all variants of this encoding

If `Rn == '1111'` then see "LDC (literal)";
If `P == '0'` & `U == '0'` & `W == '0'` then UNDEFINED;

```
n = UInt(Rn);  cp = 14;
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

#### T1

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4 3</th>
<th>0</th>
<th>15 14 13 12</th>
<th>11 10 9 8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>l=111</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Rn</td>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

#### Offset variant

Applies when `P == 1 && W == 0`. 
LDC{<c>}{<q>} p14, c5, [<Rn>{, #{+/-}<imm>}]  

Post-indexed variant
Applies when P == 0 && W == 1.
LDC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>  

Pre-indexed variant
Applies when P == 1 && W == 1.
LDC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>!  

Unindexed variant
Applies when P == 0 && U == 1 && W == 0.
LDC{<c>}{<q>} p14, c5, [<Rn>], <option>  

Decode for all variants of this encoding
if Rn == '1111' then SEE "LDC (literal)";  
if P == '0' && U == '0' && W == '0' then UNDEFINED;  
n = Uint(Rn);  cp = 14;  
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');  

Assembler symbols
<c>          See Standard assembler syntax fields on page F1-7228.
<q>          See Standard assembler syntax fields on page F1-7228.
<Rn>         Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see LDC (literal).
<option>     Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field. The value
              of this field is ignored when executing this instruction.
+/-          Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and
              encoded in the "U" field. It can have the following values:
              - when U == 0
              + when U == 1
<imm>        Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting
              to 0 and encoded in the "imm8" field, as <imm>/4.

Operation for all encodings
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];  

    // System register write to DBGDTRTXint.
    AArch32.SysRegWriteM(cp, ThisInstr(), address);  
    if wback then R[n] = offset_addr;  

Operational information
If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.65  LDC (literal)

Load data to System register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to the DBGDTRTXintSystem register. For information about memory accesses see Memory accesses on page F1-7233.

In an implementation that includes EL2, the permitted LDC access to DBGDTRXintcan be trapped to Hyp mode, meaning that an attempt to execute an LDC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see Trapping general Non-secure System register accesses to debug registers on page G1-9027.

For simplicity, the LDC pseudocode does not show this possible trap to Hyp mode.

A1

![A1 variant](image)

**A1 variant**

Applies when !(P == 0 && U == 0 && W == 0).

LDC{<c>}{<q>} p14, c5, <label>
LDC{<c>}{<q>} p14, c5, [PC, #{+/-}<imm>]
LDC{<c>}{<q>} p14, c5, [PC], <option>

**Decode for this encoding**

if P == '0' && U == '0' && W == '0' then UNDEFINED;
index = (P == '1'); add = (U == '1'); cp = 14; imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If W == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

![T1 variant](image)

**T1 variant**

Applies when !(P == 0 && U == 0 && W == 0).

LDC{<c>}{<q>} p14, c5, <label>
LDC{<c>}{<q>} p14, c5, [PC, #{+/-}<imm>]
Decode for this encoding

\[
\text{if } P == '0' \&\& U == '0' \&\& W == '0' \text{ then UNDEFINED;} \\
\text{index = (P == '1'); add = (U == '1'); cp = 14; imm32 = ZeroExtend(imm8:'00', 32);} \\
\text{if W == '1' || (P == '0' \&\& CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;} \\
\]

CONSTRANGED UNPREDICTABLE behavior

If W == '1' || P == '0', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes as LDC (immediate) with writeback to the PC. The instruction is handled as described in Using R15 by instruction on page K1-11563.

Assembler symbols

- See Standard assembler syntax fields on page F1-7228.
- See Standard assembler syntax fields on page F1-7228.
- Is an 8-bit immediate, in the range 0 to 255 enclosed in {}, encoded in the "imm8" field. The value of this field is ignored when executing this instruction.
- The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020.
- If the offset is zero or positive, imm32 is equal to the offset and add == TRUE (encoded as U == 1).
- If the offset is negative, imm32 is equal to minus the offset and add == FALSE (encoded as U == 0).
- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
  - when U = 0
  - when U = 1
- Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm" field, as <imm>/4.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax on page F2-7257.

Operation for all encodings

\[
\text{if ConditionPassed()} \text{ then} \\
\quad \text{EncodingSpecificOperations();} \\
\quad \text{offset_addr = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);} \\
\quad \text{address = if index then offset_addr else Align(PC,4);} \\
\quad // System register write to DBGDTRXint. \\
\quad AArch32.SysRegWriteM(cp, ThisInstr(), address);
\]

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.66   LDM, LDMIA, LDMFD

Load Multiple (Increment After, Full Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the highest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC on page F1-7234.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133. Related system instructions are LDM (User registers) and LDM (exception return).

This instruction is used by the alias POP (multiple registers). See Alias conditions on page F5-7602 for details of when each alias is preferred.

### A1

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>W</td>
<td>1</td>
<td>Rn</td>
<td>register_list</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### A1 variant

LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMFD{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Descending stack

#### Decode for this encoding

\[ n = \text{UInt}(Rn); \text{ registers } = \text{register_list}; \text{ wback } = (W == '1'); \]
\[ \text{ if } n == 15 \text{ } || \text{ BitCount(registers)} < 1 \text{ then UNPREDICTABLE; } \]
\[ \text{ if wback } \&\& \text{ registers}<n> == '1' \text{ then UNPREDICTABLE; } \]

#### CONSTRAINED UNPREDICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If wback \&\& registers<o> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.
T1

\[
\begin{array}{cccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 8 & 7 & | & 0 \\
1 & 1 & 0 & 0 & 1 & & Rn & & & & register\_list
\end{array}
\]

T1 variant

\[
\text{LDM(IA)\{<c>\}{<q>} <Rn>\{!\}, <registers> // Preferred syntax}
\]

\[
\text{LDMFD(IA)\{<c>\}{<q>} <Rn>\{!\}, <registers> // Alternate syntax, Full Descending stack}
\]

Decode for this encoding

\[
n = \text{UInt}(Rn); \quad \text{registers} = '00000000':\text{register\_list}; \quad \text{wback} = (\text{registers}<m> == '0');
\]

\[
\text{if BitCount(registers) < 1 then UNPREDICTABLE;}
\]

CONSTRINED UNPREDICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

T2

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & | & | & | & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & W & 1 & & Rn & P & M & register\_list
\end{array}
\]

T2 variant

\[
\text{LDM(IA)\{<c>\}.W <Rn>\{!\}, <registers> // Preferred syntax, if <Rn>, '! and <registers> can be represented in T1}
\]

\[
\text{LDMFD(IA)\{<c>\}.W <Rn>\{!\}, <registers> // Alternate syntax, Full Descending stack, if <Rn>, '! and <registers> can be represented in T1}
\]

\[
\text{LDM(IA)\{<c>\}{<q>} <Rn>\{!\}, <registers> // Preferred syntax}
\]

\[
\text{LDMFD(IA)\{<c>\}{<q>} <Rn>\{!\}, <registers> // Alternate syntax, Full Descending stack}
\]

Decode for this encoding

\[
n = \text{UInt}(Rn); \quad \text{registers} = P:M:register\_list; \quad \text{wback} = (W == '1');
\]

\[
\text{if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;}
\]

\[
\text{if wback && registers}<m> == '1' then UNPREDICTABLE;}
\]

\[
\text{if registers}<13> == '1' then UNPREDICTABLE;}
\]

\[
\text{if registers}<15> == '1' && \text{InITBlock()} && !\text{LastInITBlock()} then UNPREDICTABLE;}
\]

CONSTRINED UNPREDICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
• The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If $\text{wback} \land \text{registers}<n> == '1'$, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If $\text{BitCount(registers)} == 1$, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction loads a single register using the specified addressing modes.
• The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If $\text{registers}<13> == '1'$, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction performs all of the loads using the specified addressing mode, but R13 is UNKNOWN.

If $\text{P} == '1' \land \text{M} == '1'$, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>of variant</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>POP (multiple registers)</td>
<td>T2</td>
<td>$\text{w} == '1' \land \text{Rn} == '1101' \land \text{BitCount(P:M:register_list)} &gt; 1</td>
</tr>
<tr>
<td>POP (multiple registers)</td>
<td>A1</td>
<td>$\text{w} == '1' \land \text{Rn} == '1101' \land \text{BitCount(register_list)} &gt; 1</td>
</tr>
</tbody>
</table>

Assembler symbols

IA
Is an optional suffix for the Increment After form.

<
See Standard assembler syntax fields on page F1-7228.

<
See Standard assembler syntax fields on page F1-7228.

<Rn>
Is the general-purpose base register, encoded in the "Rn" field.
For encoding A1 and T2: the address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

For encoding T1: the address adjusted by the size of the data loaded is written back to the base register. It is omitted if <Rn> is included in <registers>, otherwise it must be present.

<registers> For encoding A1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

For encoding T1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field.

For encoding T2: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0.

If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    for i = 0 to 14
        if registers[i] == '1' then
            R[i] = MemS[address,4];  address = address + 4;
    end
    if registers[15] == '1' then
        LoadWritePC(MemS[address,4]);
    end
    if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
end
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDM (exception return)

Load Multiple (exception return) loads multiple registers from consecutive memory locations using an address from a base register. The SPSR of the current mode is copied to the CPSR. An address adjusted by the size of the data loaded can optionally be written back to the base register.

The registers loaded include the PC. The word loaded for the PC is treated as an address and a branch occurs to that address.

The PE checks the encoding that is copied to the CPSR for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.

Load Multiple (exception return) is:

• UNDEFINED in Hyp mode.
• UNPREDICTABLE in debug state, and in User mode and System mode.

A1

A1 variant

LDM{<amode>}{<c>}{<q>} <Rn>{!}, <registers_with_pc>^

Decode for this encoding

n = UInt(Rn);  registers = register_list;
wbback = (W == '1');  increment = (U == '1');  wordhigher = (P == U);
if n == 15 then UNPREDICTABLE;
if wbback && registers<n> == '1' then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If wback && registers<n> == '1', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction performs all the loads using the specified addressing mode and the content of the register being written back is UNKNOWN. In addition, if an exception occurs during the execution of this instruction, the base address might be corrupted so that the instruction cannot be repeated.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<amode> is one of:

DA Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.
FA Full Ascending. For this instruction, a synonym for DA.
DB Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.
Empty Ascending. For this instruction, a synonym for DB.
Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.
Full Descending. For this instruction, a synonym for IA.
Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.
Empty Descending. For this instruction, a synonym for IB.

See Standard assembler syntax fields on page F1-7228.

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must be specified in the register list, and the instruction causes a branch to the address (data) loaded into the PC. See also Encoding of lists of general-purpose registers and the PC on page F1-7234.

Instructions with similar syntax but without the PC included in the registers list are described in LDM (User registers).

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then
        UNPREDICTABLE;       // UNDEFINED or NOP
    else
        length = 4*BitCount(registers) + 4;
        address = if increment then R[n] else R[n]-length;
        if wordhigher then address = address + 4;
        for i = 0 to 14
            if registers<i> == '1' then
                R[i] = MemS[address,4];  address = address + 4;
                new_pc_value = MemS[address,4];
            if wback && registers<n> == '0' then R[n] = if increment then R[n]+length else R[n]-length;
            if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
            AArch32.ExceptionReturn(new_pc_value, SPSR[]);

CONstrained UNPREDICTABLE behavior

If PSTATE.M IN {M32_User,M32_System}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.68  LDM (User registers)

In an EL1 mode other than System mode, Load Multiple (User registers) loads multiple User mode registers from consecutive memory locations using an address from a base register. The registers loaded cannot include the PC. The PE reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

Load Multiple (User registers) is UNDEFINED in Hyp mode, and UNPREDICTABLE in User and System modes.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC.

A1

<table>
<thead>
<tr>
<th>31 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 14</th>
<th></th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111 1 0 0</td>
<td>P</td>
<td>U</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>0</td>
<td>register_list</td>
</tr>
</tbody>
</table>

**A1 variant**

LDM{<amode>}{<c>}{<q>} <Rn>, <registers_without_pc>\^a

**Decode for this encoding**

n = UInt(Rn); registers = register_list; increment = (U == '1'); wordhigher = (P == U);
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

<amode> is one of:

- **DA** Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.
- **FA** Full Ascending. For this instruction, a synonym for DA.
- **DB** Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.
- **EA** Empty Ascending. For this instruction, a synonym for DB.
- **IA** Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.
- **FD** Full Descending. For this instruction, a synonym for IA.
- **IB** Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.
ED Empty Descending. For this instruction, a synonym for IB.

<c> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<registers_without_pc>
Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded by the LDM instruction. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must not be in the register list. See also Encoding of lists of general-purpose registers and the PC on page F1-7234.

Instructions with similar syntax but with the PC included in <registers_without_pc> are described in LDM (exception return).

Operation
if ConditionPassed() then
   EncodingSpecificOperations();
   if PSTATE.EL == EL2 then UNDEFINED;
   elsif PSTATE.M IN {M32_User, M32_System} then UNPREDICTABLE;
   else
      length = 4*BitCount(registers);
      address = if increment then R[n] else R[n]-length;
      if wordhigher then address = address+4;
      for i = 0 to 14
         if registers<i> == '1' then // Load User mode register
            Rmode[i, M32_User] = MemS[address,4]; address = address + 4;
      endfor
   end

CONstrained UNpredicable behavior
If PSTATE.M IN {M32_User, M32_System}, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

Operational information
If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Load Multiple Decrement After (Full Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC on page F1-7234.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133. Related system instructions are LDM (User registers) and LDM (exception return).

A1

```
31 28 27 26 25 24 23 22 21 20 19 16 15  |  |  |  0
1=111 0 0 0 0 0 W 1  Rn  |  register_list
```

**A1 variant**

LDMDA{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMFA{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Ascending stack

**Decode for this encoding**

```
n = Uint(Rn); registers = register_list; wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

**CONSTRAINED UNPREDICTABLE behavior**

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If wback && registers<n> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers> Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.
The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers) + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemS[address,4]; address = address + 4;
        if registers<15> == '1' then
            LoadWritePC(MemS[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.70 LDMDB, LDMEA

Load Multiple Decrement Before (Empty Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC on page F1-7234.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133. Related system instructions are LDM (User registers) and LDM (exception return).

A1

\[
\begin{array}{cccccccccc}
| & | & & & & & & & & & \\
\hline
1 & =111 & & & & & & & & & 1 & 0 & 0 & 1 & 0 & 0 & W & 1 & Rn & \text{register_list} \\
\end{array}
\]

**cond**

\[
A1 \text{ variant}
\]

LDMDB\{<c>\}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMEA\{<c>\}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Ascending stack

**Decode for this encoding**

\[
n = \text{UInt}(Rn); \quad \text{registers} = \text{register_list}; \quad \text{wback} = (W' = '1');
\]

\[
\text{if } n = 15 \text{ || BitCount(registers) < 1 then UNPREDICTABLE;}
\]

\[
\text{if } \text{wback} \&\& \text{registers}<n> = '1' \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( \text{wback} \&\& \text{registers}<n> = '1' \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

T1

\[
\begin{array}{cccccccccc}
| & | & & & & & & & & & \\
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 14 & 13 & | & | & | & 0 \\
\hline
1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & W & 1 & Rn & P & M & \text{register_list} \\
\end{array}
\]
**T1 variant**

`LDMDB{<c>}{<q>} <Rn>{!}, <registers>` // Preferred syntax

`LDMEA{<c>}{<q>} <Rn>{!}, <registers>` // Alternate syntax, Empty Ascending stack

**Decode for this encoding**

\[
\begin{align*}
n &= \text{UInt}(Rn); \quad \text{registers} = P:M:register\_list; \quad \text{wback} = (W == '1'); \\
&\text{if } n = 15 || \text{BitCount(registers)} < 2 || (P == '1' &\& M == '1') \text{ then UNPREDICTABLE;} \\
&\text{if } \text{wback} &\& \text{registers}<n> == '1' \text{ then UNPREDICTABLE;} \\
&\text{if } \text{registers}<13> == '1' \text{ then UNPREDICTABLE;} \\
&\text{if } \text{registers}<15> == '1' &\& \text{InITBlock}() &\& !\text{LastInITBlock}() \text{ then UNPREDICTABLE;}
\end{align*}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If `wback &\& registers<15> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode, but R13 is UNKNOWN.

If `P == '1' &\& M == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<c>  See Standard assembler syntax fields on page F1-7228.

<i>  See Standard assembler syntax fields on page F1-7228.

<Rn>  Is the general-purpose base register, encoded in the "Rn" field.

!  The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers>  For encoding A1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

For encoding T1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0.

If the PC is in the list:

• The LR must not be in the list.
• The instruction must be either outside any IT block, or the last instruction in an IT block.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  address = R[n] - 4*BitCount(registers);
  for i = 0 to 14
    if registers<i> == '1' then
      R[i] = MemS[address,4];  address = address + 4;
    if registers<15> == '1' then
      LoadWritePC(MemS[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.71  LDMIB, LDMED

Load Multiple Increment Before (Empty Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC on page F1-7234.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133. Related system instructions are LDM (User registers) and LDM (exception return).

A1

A1 variant

LDMIB{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMED{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Descending stack

Decode for this encoding

\( n = \text{UInt}(Rn); \text{ registers} = \text{register\_list}; \text{ wback} = (W == '1'); \)
if \( n = 15 \mid \text{ BitCount}\text{ registers} < 1 \) then UNPREDICTABLE;
if \( \text{wback} \& \text{ registers}<< = '1' \) then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If \( \text{BitCount}\text{ registers} < 1 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If \( \text{wback} \& \text{ registers}<< = '1' \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

Notes for all encodings

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers> Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.
The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemS[address,4]; address = address + 4;
        if registers<15> == '1' then
            LoadWritePC(MemS[address,4]);
        if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
        if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.72   LDR (immediate)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses on page F1-7233.

This instruction is used by the alias POP (single register). See Alias conditions on page F5-7617 for details of when each alias is preferred.

A1

<table>
<thead>
<tr>
<th>31 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 12</th>
<th>11</th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1=1111</td>
<td>0</td>
<td>1</td>
<td>P</td>
<td>U</td>
<td>0</td>
<td>W</td>
<td>1</td>
</tr>
<tr>
<td>cond</td>
<td>Rt</td>
<td>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Offset variant

Applies when \( P = 1 \) \&\& \( W = 0 \).

LDR\{<c>\}{<q>} \{<Rt>, [<Rn> {, #{+/-}<imm>}]

Post-indexed variant

Applies when \( P = 0 \) \&\& \( W = 0 \).

LDR\{<c>\}{<q>} \{<Rt>, [<Rn>], #{+/-}<imm} |

Pre-indexed variant

Applies when \( P = 1 \) \&\& \( W = 1 \).

LDR\{<c>\}{<q>} \{<Rt>, [<Rn>], #{+/-}<imm} |

Decode for all variants of this encoding

- If \( Rn = '1111' \) then SEE "LDR (literal)";
- If \( P = '0' \) \&\& \( W = '1' \) then SEE "LDRT";
- \( t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32); index = (P = '1'); add = (U = '1'); wback = (P = '0') \| (W = '1');
- If wback \&\& n = t then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If wback \&\& n = t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10</th>
<th>6 5</th>
<th>3 2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1</td>
<td>0 1</td>
<td>imm5</td>
<td>Rn</td>
<td>Rt</td>
</tr>
</tbody>
</table>
**T1 variant**

\[
\text{LDR}\{<c>\}\{<q>\} \ <Rt>, \ [<Rn> \ \{, \ #{+}<imm>\}]
\]

**Decode for this encoding**

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad \text{imm32} = \text{ZeroExtend}(\text{imm5}:'00', \ 32);
\]

\[
\text{index} = \text{TRUE}; \quad \text{add} = \text{TRUE}; \quad \text{wback} = \text{FALSE};
\]

**T2**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
Rt  imm8
```

**T2 variant**

\[
\text{LDR}\{<c>\}\{<q>\} \ <Rt>, \ [SP\{, \ #{+}<imm>\}]
\]

**Decode for this encoding**

\[
t = \text{UInt}(Rt); \quad n = 13; \quad \text{imm32} = \text{ZeroExtend}(\text{imm8}:'00', \ 32);
\]

\[
\text{index} = \text{TRUE}; \quad \text{add} = \text{TRUE}; \quad \text{wback} = \text{FALSE};
\]

**T3**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 12 | 11 | | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  |     |     |     |     |     | l=1111 | Rt | imm12 | |     |     |

```
Rn
```

**T3 variant**

\[
\text{LDR}\{<c>\}.W \ <Rt>, \ [<Rn> \ \{, \ #{+}<imm>\}] \ // \ <Rt>, \ <Rn>, \ <imm> \ can \ be \ represented \ in \ T1 \ or \ T2
\]

\[
\text{LDR}\{<c>\}\{<q>\} \ <Rt>, \ [<Rn> \ \{, \ #{+}<imm>\}]
\]

**Decode for this encoding**

\[
\text{if} \ Rn == '1111' \ \text{then} \ \text{SEE} \ "LDR \ (\text{literal})";
\]

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad \text{imm32} = \text{ZeroExtend}(\text{imm12}, \ 32); \quad \text{index} = \text{TRUE}; \quad \text{add} = \text{TRUE}; \quad \text{wback} = \text{FALSE}; \quad \text{if} \ t == 15 \ \&\& \ \text{InITBlock() \ \&\& \ !LastInITBlock()} \ \text{then} \ \text{UNPREDICTABLE};
\]

**T4**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>l=1111</td>
<td>Rt</td>
<td>1</td>
<td>P</td>
<td>U</td>
<td>W</td>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
Rn
```

**Offset variant**

Applies when \( P == 1 \ \&\& \ U == 0 \ \&\& \ W == 0. \)

\[
\text{LDR}\{<c>\}\{<q>\} \ <Rt>, \ [<Rn> \ \{, \ #-<imm>\}]
\]

**Post-indexed variant**

Applies when \( P == 0 \ \&\& \ W == 1. \)
LDR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>

Pre-indexed variant
Applies when \( P = 1 \) \&\& \( W = 1 \).
LDR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>!

Decode for all variants of this encoding

\[
\begin{align*}
\text{if } Rn &= '1111' \text{ then SEE "LDR (literal)"; } \\
\text{if } P &= '1' \&\& U = '1' \&\& W = '0' \text{ then SEE "LDRT"; } \\
\text{if } P &= '0' \&\& W = '0' \text{ then UNDEFINED; } \\
t &= UInt(Rt); \quad n = UInt(Rn); \\
imm32 &= ZeroExtend(imm8, 32); \quad \text{index} = (P = '1'); \quad \text{add} = (U = '1'); \quad \text{wback} = (W = '1'); \\
\text{if } (\text{wback} \&\& n = t) || (t = 15 \&\& \text{InITBlock()} \&\& \text{!LastInITBlock()}) \text{ then UNPREDICTABLE;}
\end{align*}
\]

CONSTRUANED UNPREDICTABLE behavior
If \( wback \&\& n = t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

Notes for all encodings
For more information about the CONSTRUANED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Alias conditions

<table>
<thead>
<tr>
<th>Alias of variant</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>POP (single register) A1 (post-indexed)</td>
<td>( P = '0' &amp;&amp; U = '1' &amp;&amp; W = '0' &amp;&amp; Rn = '1101' &amp;&amp; imm12 = '000000000100'</td>
</tr>
<tr>
<td>POP (single register) T4 (post-indexed)</td>
<td>( Rn = '1101' &amp;&amp; P = '0' &amp;&amp; U = '1' &amp;&amp; W = '1' &amp;&amp; imm8 = '00000100'</td>
</tr>
</tbody>
</table>

Assembler symbols

- See Standard assembler syntax fields on page F1-7228.
- See Standard assembler syntax fields on page F1-7228.
- For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- For encoding T1 and T2: is the general-purpose register to be transferred, encoded in the "Rt" field.
- For encoding T3 and T4: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
<Rn> For encoding A1, T3 and T4: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDR (literal).
For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

+- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
- when $U = 0$
+ when $U = 1$

+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.
For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 124, defaulting to 0 and encoded in the "imm5" field as <imm>/4.
For encoding T2: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <imm>/4.
For encoding T3: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
For encoding T4: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

**Operation for all encodings**

```c
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        data = MemU[address,4];
        if wback then R[n] = offset_addr;
        if t == 15 then
            if address<1:0> == '00' then
                LoadWritePC(data);
            else
                UNPREDICTABLE;
            else
                R[t] = data;
    else
data = MemU[R[n],4];
        if wback then R[n] = offset_addr;
        if t == 15 then
            if address<1:0> == '00' then
                LoadWritePC(data);
            else
                UNPREDICTABLE;
        else
            R[t] = data;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.73   **LDR (literal)**

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses see *Memory accesses on page F1-7233*.

**A1**

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 12 | 11 | | | 0 |
| 1=1111 | 0 | 1 | 0 | P | U | 0 | W | 1 | 1 | 1 | 1 | Rt | | | imm12 |

**A1 variant**

Applies when !(P == 0 && W == 1).

LDR{<c>}{<q>} <Rt>, <label> // Normal form
LDR{<c>}{<q>} <Rt>, [PC, #{+/-}imm] // Alternative form

**Decode for this encoding**

if P == '0' && W == '1' then SEE "LDRT";
t = UInt(Rt); imm32 = ZeroExtend(imm12, 32);
add = (U == '1'); wback = (P == '0') || (W == '1');
if wback then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE;
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDR (immediate). The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15 by instruction on page K1-11563.

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 8 | 7 | | | | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 | Rt | | imm8 |

**T1 variant**

LDR{<c>}{<q>} <Rt>, <label> // Normal form

**Decode for this encoding**

t = UInt(Rt); imm32 = ZeroExtend(imm8:'00', 32); add = TRUE;
T2

|15|14|13|12|11|10|9|8|7|6|5|4|3|2|1|0|15|12|11| | | |0|
|1 |1 |1 |1 |1 |0 |0 |0 |U |1 |0 |1 |1 |1 |1 |Rt | | |imm12|

T2 variant

LDR{<c>}.W <Rt>, <label> // Preferred syntax, and <Rt>, <label> can be represented in T1
LDR{<c>}{<q>} <Rt>, <label> // Preferred syntax
LDR{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative syntax

Decode for this encoding

t = UInt(Rt); imm32 = ZeroExtend(imm12, 32); add = (U == '1');
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

>
See Standard assembler syntax fields on page F1-7228.

<Rt>
For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.

For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

For encoding T2: is the general-purpose register to be transferred, encoded in the "Rt" field. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.

<label>
For encoding A1 and T2: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1.
If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are Multiples of four in the range 0 to 1020.

+/-
Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when U = 0
+ when U = 1

<imm>
For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.
For encoding T2: if a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax on page F2-7257.

**Operation for all encodings**

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC, 4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address, 4];
    if t == 15 then
        if address<1:0> == '00' then
            LoadWritePC(data);
        else
            UNPREDICTABLE;
    else
        R[t] = data;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.74   LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses, see Memory accesses on page F1-7233.

The T32 form of LDR (register) does not support register writeback.

A1

<table>
<thead>
<tr>
<th>31 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 12</th>
<th>11</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1=1111</td>
<td>0 1 1</td>
<td>P U 0 W</td>
<td>1</td>
<td>Rn</td>
<td>Rt</td>
<td>immS</td>
<td>stype</td>
</tr>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Offset variant**

Applies when \( P == 1 \land W == 0 \).

\[ \text{LDR\{}<c>\}{<q>} \text{ <Rt>, \{<Rn>, {+/-}<Rm}\{, <shift>\}} \]

**Post-indexed variant**

Applies when \( P == 0 \land W == 0 \).

\[ \text{LDR\{}<c>\}{<q>} \text{ <Rt>, \{<Rn>\}, {+/-}<Rm}\{, <shift>\}} \]

**Pre-indexed variant**

Applies when \( P == 1 \land W == 1 \).

\[ \text{LDR\{}<c>\}{<q>} \text{ <Rt>, \{<Rn>, {+/-}<Rm}\{, <shift>\}}! \]

**Decode for all variants of this encoding**

\[
\begin{align*}
&\text{if } P == '0' \land W == '1' \text{ then SEE "LDRT";} \\
&t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \\
&\text{index} = (P == '1'); \quad \text{add} = (U == '1'); \quad \text{wback} = (P == '0') \land (W == '1'); \\
&(\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{immS}); \\
&\text{if } m == 15 \text{ then UNPREDICTABLE;} \\
&\text{if wback} \land (n == 15 || n == t) \text{ then UNPREDICTABLE;}
\end{align*}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( \text{wback} \land n == t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>6 5</th>
<th>3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1</td>
<td>1 0</td>
<td>0</td>
<td>Rm</td>
</tr>
</tbody>
</table>
**T1 variant**

LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

**Decode for this encoding**

t = Uint(Rt);  n = Uint(Rn);  m = Uint(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);

**T2**

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0 15</th>
<th>12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 0 0</td>
<td>1 0 1</td>
<td>t=1111</td>
<td>Rt</td>
<td>0 0 0 0 0 0</td>
<td>imm2</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**T2 variant**

LDR{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

**Decode for this encoding**

if Rn == '1111' then SEE "LDR (Literal)"

t = Uint(Rt);  n = Uint(Rn);  m = Uint(Rm);
(shift_t, shift_n) = (SRType_LSL, Uint(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1
Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

**<c>**

See Standard assembler syntax fields on page F1-7228.

**<q>**

See Standard assembler syntax fields on page F1-7228.

**<Rt>**

For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This branch is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.

For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

For encoding T2: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.

**<Rn>**

For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

**+-**

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when U = 0
+ when \( U = 1 \)

+ Specifies the index register is added to the base register.

\(<Rm>\)
Is the general-purpose index register, encoded in the "Rm" field.

\(<\text{shift}>\)
The shift to apply to the value read from \(<Rm>\). If absent, no shift is applied. Otherwise, see Shifts applied to a register on page F1-7231.

\(<\text{imm}>\)
If present, the size of the left shift to apply to the value from \(<Rm>\), in the range 1-3. \(<\text{imm}>\) is encoded in \text{imm}2. If absent, no shift is specified and \text{imm}2 is encoded as 0b00.

### Operation for all encodings

```
if \( \text{CurrentInstrSet}() = \text{InstrSet.A32} \) then
  if \( \text{ConditionPassed}() \) then
    \( \text{EncodingSpecificOperations}(); \)
    \( \) offset = \( \text{Shift}(R[m], \text{shift}_t, \text{shift}_n, \text{PSTATE.C}) \);
    \( \) offset_addr = if add then \( (R[n] + \text{offset}) \) else \( (R[n] - \text{offset}) \);
    \( \) address = if index then offset_addr else \( R[n] \);
    \( \) data = \( \text{MemU}[\text{address}, 4] \);
    \( \) if wback then \( R[n] = \) offset_addr;
    \( \) if \( t = 15 \) then
      if \( \text{address}[1:0] = '00' \) then
        \( \) \( \text{LoadWritePC}(\text{data}); \)
      else
        \text{UNPREDICTABLE};
    else
      \( R[t] = \) data;
  else
    if \( \text{ConditionPassed}() \) then
      \( \text{EncodingSpecificOperations}(); \)
      \( \) offset = \( \text{Shift}(R[m], \text{shift}_t, \text{shift}_n, \text{PSTATE.C}) \);
      \( \) offset_addr = \( \text{offset_addr} \);
      \( \) address = offset_addr;
      \( \) data = \( \text{MemU}[\text{address}, 4] \);
      \( \) if \( t = 15 \) then
        if \( \text{address}[1:0] = '00' \) then
          \( \) \( \text{LoadWritePC}(\text{data}); \)
        else
          \text{UNPREDICTABLE};
      else
        \( R[t] = \) data;
```

### Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.75  LDRB (immediate)

Load Register Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses on page F1-7233.

A1

Offset variant
Applies when \( P = 1 \) \&\& \( W = 0 \).
LDRB\{<c>\}<q> \{<Rt>, \[<Rn> {, #{+/-}<imm>}\]

Post-indexed variant
Applies when \( P = 0 \) \&\& \( W = 0 \).
LDRB\{<c>\}<q> \{<Rt>, \[<Rn>\], #{+/-}<imm>

Pre-indexed variant
Applies when \( P = 1 \) \&\& \( W = 1 \).
LDRB\{<c>\}<q> \{<Rt>, \[<Rn>, {#+/-}imm\]

Decode for all variants of this encoding

If \( \text{Rn} = '1111' \) then SEE "LDRB (literal)";
if \( P = '0' \) \&\& \( W = '1' \) then SEE "LDRBT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback \&\& n == t) then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior
If wback \&\& n == t, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

T1 variant
LDRB\{<c>\}<q> \{<Rt>, \[<Rn> \{, #{+/-}imm\]}

T32 and A32 Base Instruction Set Instruction Descriptions
F5.1 Alphabetical list of T32 and A32 base instruction set instructions

ARM DDI 0487I.a Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
ID081822 Non-Confidential
Decode for this encoding

t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;

T2

\[
\begin{array}{cccccccccc}
| & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 12 & 11 & 0 |
\end{array}
\]

<table>
<thead>
<tr>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T2 variant

LDRB{<c>}.W <Rt>, [Rn] {, #{+}<imm>} // <Rt>, <Rn>, <imm> can be represented in T1
LDRB{<c>{<q>} <Rt>, [Rn] {, #{+}<imm>}}

Decode for this encoding

if Rt == '1111' then SEE "PLD";
if Rn == '1111' then SEE "LDRB (literal)";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm32, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13

T3

\[
\begin{array}{cccccccccc}
| & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 12 & 11 & 0 |
\end{array}
\]

<table>
<thead>
<tr>
<th>Rn</th>
<th>Rt</th>
<th>P</th>
<th>U</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Offset variant

Applies when Rt != 1111 && P == 1 && U == 0 && W == 0.
LDRB{<c>}{<q>} <Rt>, [Rn] {, #{+/-<imm>}!}

Post-indexed variant

Applies when P == 0 && W == 1.
LDRB{<c>}{<q>} <Rt>, [Rn], #{+/-<imm>}

Pre-indexed variant

Applies when P == 1 && W == 1.
LDRB{<c>}{<q>} <Rt>, [Rn], #{+/-<imm>!}

Decode for all variants of this encoding

if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLD, PLDW (immediate)";
if P == '1' && U == '1' && W == '0' then SEE "LDRBT";
if Rn == '1111' then SEE "LDRB (literal)";
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
\textbf{CONSTRAINED UNPREDICTABLE behavior}

If \( \text{wback} \land n = t \), then one of the following behaviors must occur:

- The instruction is \textbf{UNDEFINED}.
- The instruction executes as \texttt{NOP}.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is \textit{UNKNOWN}. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

\textbf{Notes for all encodings}

For more information about the \textbf{CONSTRAINED UNPREDICTABLE behavior} of this instruction, see \textit{Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors}.

\textbf{Assembler symbols}

- \(<c>\) See \textit{Standard assembler syntax fields on page F1-7228}.
- \(<q>\) See \textit{Standard assembler syntax fields on page F1-7228}.
- \(<Rt>\) Is the general-purpose register to be transferred, encoded in the "Rt" field.
- \(<Rn>\) For encoding A1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see \texttt{LDRB (literal)}.
  - For encoding T1: is the general-purpose base register, encoded in the "Rn" field.
- \(+/-\) Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
  - - when \( U = 0 \)
  - + when \( U = 1 \)
  - + Specifies the offset is added to the base register.
- \(<\text{imm}>\)
  - For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.
  - For encoding T1: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.
  - For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
  - For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

\textbf{Operation for all encodings}

\begin{verbatim}
if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
  else
    if ConditionPassed() then
      EncodingSpecificOperations();
      offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
      address = if index then offset_addr else R[n];

\end{verbatim}
R[t] = ZeroExtend(MemU[address,1], 32);
if wbback then R[n] = offset_addr;

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.76 LDRB (literal)

Load Register Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses on page F1-7233.

A1

\[
\begin{array}{cccc|cc|c}
\hline
0 & 1 & 0 & P & U & 1 & W & 1 & 1 & 1 & 1 & Rt & \text{imm12} \hline
\end{array}
\]

\text{cond}

A1 variant

Applies when \(! (P == 0 \&\& W == 1)\).

LDRB\{<c>\}{<q>} <Rt>, <label> // Normal form
LDRB\{<c>\}{<q>} <Rt>, [PC, \#{+/-}\text{imm}] // Alternative form

Decode for this encoding

\[
\begin{align*}
\text{if } P &= '0' \&\& W == '1' \text{ then see "LDRBT";} \\
\text{t} &= \text{UInt}(Rt); \text{ imm32} = \text{ZeroExtend}(\text{imm12}, 32); \\
\text{add} &= (U == '1'); \text{ wback} = (P == '0') || (W == '1'); \\
\text{if } t &= 15 \text{ } \text{ } \text{ wback then UNPREDICTABLE;}
\end{align*}
\]

CONSTRAINED UNPREDICTABLE behavior

If wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE;
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRB (immediate). The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15 by instruction on page K1-11563.

T1

\[
\begin{array}{cccc|cc|c}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & \hline
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & \hline
\text{l=1111} & \text{imm12} \hline
Rt
\end{array}
\]

T1 variant

LDRB\{<c>\}{<q>} <Rt>, <label> // Preferred syntax
LDRB\{<c>\}{<q>} <Rt>, [PC, \#{+/-}\text{imm}] // Alternative syntax

Decode for this encoding

\[
\begin{align*}
\text{if } Rt &= '1111' \text{ then see "PLD";} \\
\text{t} &= \text{UInt}(Rt); \text{ imm32} = \text{ZeroExtend}(\text{imm12}, 32); \text{ add} = (U == '1'); \\
&\text{ // Armv8-A removes UNPREDICTABLE for R13}
\end{align*}
\]
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<> See Standard assembler syntax fields on page F1-7228.
<q> See Standard assembler syntax fields on page F1-7228.
<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
<label> The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095.
    If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1.
    If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.
+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
    - when U = 0
    + when U = 1
<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.
    For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax on page F2-7257.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC, 4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = ZeroExtend(MemU[address,1], 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.77 LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses see Memory accesses on page F1-7233.

A1

|31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 7 6 5 4 3 0 |
|---|---|---|---|---|---|---|---|
|!|0 1 1|P|U|1|W|1|Rn|Rt|imm5|stype|0|Rm|

**Offset variant**

Applies when \( P = 1 \) && \( W = 0 \).

\[
\text{LDRB}\{<c>\}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}] 
\]

**Post-indexed variant**

Applies when \( P = 0 \) && \( W = 0 \).

\[
\text{LDRB}\{<c>\}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
\]

**Pre-indexed variant**

Applies when \( P = 1 \) && \( W = 1 \).

\[
\text{LDRB}\{<c>\}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift}>]!
\]

**Decode for all variants of this encoding**

if \( P = '0' \) && \( W = '1' \) then SEE "LDRBT";

\[ t = \text{UInt}(Rt); \] \[ n = \text{UInt}(Rn); \] \[ m = \text{UInt}(Rm); \]

\[ \text{index} = (P == '1'); \]

\[ \text{add} = (U == '1'); \]

\[ \text{wback} = (P == '0') || (W == '1'); \]

\[ (\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(stype, \text{imm5}); \]

\[ \text{if} t == 15 \] \[ || \] \[ m == 15 \] \[ \text{then UNPREDICTABLE; } \]

\[ \text{if} \text{wback} \] && \[ n == 15 \] \[ || \] \[ n == t \] \[ \text{then UNPREDICTABLE; } \]

**CONstrained UNPREDICTable behavior**

If \( \text{wback} \) && \( n == t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

**T1**

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>6 5</th>
<th>3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1</td>
<td>1 1 0</td>
<td>Rm</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**T1 variant**

\[
\text{LDRB}\{<c>\}{<q>} <Rt>, [<Rn>, {+}<Rm}]
\]
**Decode for this encoding**

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]
\[
\text{index} = \text{TRUE}; \quad \text{add} = \text{TRUE}; \quad \text{wback} = \text{FALSE};
\]
\[
(shift_t, shift_n) = (\text{SRType}_{\text{LSL}}, 0);
\]

**T2**

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>15 12 11 10 9</th>
<th>8 7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 0 0 0 1</td>
<td>l=1111</td>
<td>0 0 0 0 0 0 0</td>
<td>imm2</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**T2 variant**

LDRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

**Decode for this encoding**

if Rt == '1111' then SEE "PLD";
if Rn == '1111' then SEE "LDRB (literal)"
\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]
\[
\text{index} = \text{TRUE}; \quad \text{add} = \text{TRUE}; \quad \text{wback} = \text{FALSE};
\]
\[
(shift_t, shift_n) = (\text{SRType}_{\text{LSL}}, \text{UInt}(imm2));
\]
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.
  For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.
- `+/-` Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
  - `-` when U = 0
  - `+` when U = 1
- `<Rm>` Is the general-purpose index register, encoded in the "Rm" field.
- `<shift>` The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see Shifts applied to a register on page F1-7231.
- `<imm>` If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 000.
Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations();
   offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
   offset_addr = if add then (R[n] + offset) else (R[n] - offset);
   address = if index then offset_addr else R[n];
   R[t] = ZeroExtend(MemU[address,1],32);
   if wback then R[n] = offset_addr;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.78   LDRBT

Load Register Byte Unprivileged loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses on page F1-7233.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

A1

A1 variant

LDRBT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}

Decode for this encoding

t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;

CONstrained UNpredictable behavior

If n == 15, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15 by instruction on page K1-11563.
• The instruction uses immediate offset addressing with the base register as PC, without writeback.

If n == t && n != 15, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.
A2

LDRBT{<c>}{<q>} <Rt>, [<Rn>], {+-}<Rm>{, <shift>}

Decode for this encoding

\[ t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{postindex} = \text{TRUE}; \quad \text{add} = (U == '1'); \]

\[ \text{register\_form} = \text{TRUE}; \quad (\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(stype, \text{imm5}); \]

\[ \text{if } t == 15 \| n == 15 \| n == t \| m == 15 \text{ then UNPREDICTABLE}; \]

### CONSTRAINED UNPREDICTABLE behavior

If \( n == t \&\& \text{!} n != 15 \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is **UNKNOWN**. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

LDRBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

Decode for this encoding

\[ \text{if } Rn == '1111' \text{ then SEE "LDRB (literal)";} \]

\[ t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad \text{postindex} = \text{FALSE}; \quad \text{add} = \text{TRUE}; \]

\[ \text{register\_form} = \text{FALSE}; \quad \text{imm32} = \text{ZeroExtend}(\text{imm8}, 32); \]

\[ \text{if } t == 15 \text{ then UNPREDICTABLE}; \quad // \text{Armv8-A removes UNPREDICTABLE for R13} \]

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

Assembler symbols

\(<c>\) See *Standard assembler syntax fields* on page F1-7228.

\(<q>\) See *Standard assembler syntax fields* on page F1-7228.

\(<Rt>\) For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.
For encoding A2 and T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Rn>\)

Is the general-purpose base register, encoded in the "Rn" field.

\(<imm>\)

For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
  offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
  offset_addr = if add then (R[n] + offset) else (R[n] - offset);
  address = if postindex then R[n] else offset_addr;
  R[t] = ZeroExtend(MemU_unpriv[address,1],32);
  if postindex then R[n] = offset_addr;

CONSTRANDED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDRB (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.79   LDRD (immediate)

Load Register Dual (immediate) calculates an address from a base register value and an immediate offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses on page F1-7233.

### A1

```
| 31 | 28| 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 |  8 |  7 |  6 |  5 |  4 |  3 |  0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=1111 |  0 |  0 |  0 | P | U |  1 | W |  0 | !=1111 | Rt | imm4H |  1 |  1 |  0 |  1 | imm4L |

cond  Rn
```

**Offset variant**

Applies when \( P = 1 \) \&\& \( W = 0 \).

\( \text{LDRD}\{<c>\}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}] \)

**Post-indexed variant**

Applies when \( P = 0 \) \&\& \( W = 0 \).

\( \text{LDRD}\{<c>\}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm> \)

**Pre-indexed variant**

Applies when \( P = 1 \) \&\& \( W = 1 \).

\( \text{LDRD}\{<c>\}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]! \)

**Decode for all variants of this encoding**

- If \( Rn == '1111' \) then see "LDRD (literal)";
- If \( Rt<0> == '1' \) then UNPREDICTABLE;
- \( t = \text{UInt}(Rt); \; t2 = t+1; \; n = \text{UInt}(Rn); \; \text{imm32} = \text{ZeroExtend}(\text{imm4H}:\text{imm4L}, 32); \)
- \( \text{index} = (P == '1'); \; \text{add} = (U == '1'); \; \text{wback} = (P == '0') || (W == '1'); \)
- If \( P == '0' \) \&\& \( W == '1' \) then UNPREDICTABLE;
- If \( \text{wback} \) \&\& \( (n == t || n == t2) \) then UNPREDICTABLE;
- If \( t2 == 15 \) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \( \text{wback} \) \&\& \( (n == t || n == t2) \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If \( P == '0' \) \&\& \( W == '1' \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as a LDRD using one of offset, post-indexed, or pre-indexed addressing.

If \( Rt<0> == '1' \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
• The instruction executes as NOOP.
• The instruction executes with the additional decode: t<0> = '0'.
• The instruction executes with the additional decode: t2 = t.
• The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == '1111'.

**T1**

 Offset variant
Applies when \( P == 1 \) \&\& \( W == 0 \).
LDRD\{<c>\}{<q>} <Rt>, <Rt2>, [{<Rn> \{, #{+/-}<imm>\}}]

 Post-indexed variant
Applies when \( P == 0 \) \&\& \( W == 1 \).
LDRD\{<c>\}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>

 Pre-indexed variant
Applies when \( P == 1 \) \&\& \( W == 1 \).
LDRD\{<c>\}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!

 Decode for all variants of this encoding

 if \( P == '0' \) \&\& \( W == '0' \) then SEE "Related encodings";
 if \( Rn == '1111' \) then SEE "LDRD (literal)";
 \( t = \) UInt(Rt); \( t2 = \) UInt(Rt2); \( n = \) UInt(Rn); \( imm32 = \) ZeroExtend(imm8:'00', 32);
 index = (\( P == '1' \)); \( add = (U == '1') \); \( wback = (W == '1') \);
 if wback \&\& (n == t || n == t2) then UNPREDICTABLE;
 if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

 **CONSTRANDED UNPREDICTABLE behavior**
If wback \&\& (n == t || n == t2), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOOP.
• The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If \( t == t2 \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOOP.
• The load instruction executes but the destination register takes an UNKNOWN value.
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.


Assembler symbols

<\> See Standard assembler syntax fields on page F1-7228.

<\> See Standard assembler syntax fields on page F1-7228.

<Rt> For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

<Rt2> For encoding A1: is the second general-purpose register to be transferred. This register must be <R(t+1)>

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRD (literal).

+- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when U = 0

+ when U = 1

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm32" field.

For encoding T1: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 if omitted, and encoded in the "imm8" field as <imm>/4.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        data = MemA[address,8];
        if BigEndian(AccType_ATOMIC) then
            R[t] = data<63:32>;
            R[t2] = data<31:0>;
        else
            R[t] = data<31:0>;
            R[t2] = data<63:32>;
        else
            R[t] = MemA[address,4];
            R[t2] = MemA[address+4,4];
            if wback then R[n] = offset_addr;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.80   LDRD (literal)

Load Register Dual (literal) calculates an address from the PC value and an immediate offset, loads two words from memory, and writes them to two registers. For information about memory accesses see Memory accesses on page F1-7233.

A1

| 31 28|27 26 25 24|23 22 21 20|19 18 17 16|15|12|11| 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| !|=1111| 0 | 0 | 0 | (1) | U | 1 | 1 | 0 | 1 | 1 | 1 | Rt | imm4H | 1 | 1 | 0 | 1 | imm4L |
| cond |

A1 variant

LDRD{<c>}{<q>} <Rt>, <Rt2>, <label> // Normal form
LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, #{+/-}<imm>] // Alternative form

Decode for this encoding

if Rt<0> == '1' then UNPREDICTABLE;

if P == '0' || W == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: t<0> = '0';
- The instruction executes with the additional decode: t2 = t;
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == '1111'.

If P == '0' || W == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as if P == 1 and W == 0.'

T1

| 15| 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 12|11| 8 | 7 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | P | U | 1 | W | 1 | 1 | 1 | 1 | Rt | Rt2 | imm8 |

T1 variant

Applies when !(P == 0 & & W == 0).

LDRD{<c>}{<q>} <Rt>, <Rt2>, <label> // Normal form
LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, #{+/-}<imm>] // Alternative form
Decode for this encoding

if P == '0' && W == '0' then SEE "Related encodings";
t = UInt(Rt);  t2 = UInt(Rt2);
imm32 = ZeroExtend(imm8:'00', 32);  add = (U == '1');
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if W == '1' then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If t == t2, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The load instruction executes but the destination register takes an UNKNOWN value.

If W == '1', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes without writeback of the base address.
• The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15 by instruction on page K1-11563.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.


Assembler symbols

See Standard assembler syntax fields on page F1-7228.
See Standard assembler syntax fields on page F1-7228.

For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.
For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

For encoding A1: is the second general-purpose register to be transferred. This register must be <R(t+1)>
For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

For encoding A1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Any value in the range -255 to 255 is permitted.
If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020.
If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.
+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
- when \( U = 0 \)
+ when \( U = 1 \)

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.
For encoding T1: is the optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax on page F2-7257.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    if address == Align(address, 8) then
        data = MemA[address,8];
        if BigEndian(AccType_ATOMIC) then
            R[t] = data<63:32>;
            R[t2] = data<31:0>;
        else
            R[t] = data<31:0>;
            R[t2] = data<63:32>;
    else
        R[t] = MemA[address,4];
        R[t2] = MemA[address+4,4];

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.81 LDRD (register)

Load Register Dual (register) calculates an address from a base register value and a register offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses on page F1-7233.

A1

![Instruction Format]

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>![cond]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Offset variant**

Applies when $P == 1 \land W == 0$.

$LDRD\{<c>\}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]$

**Post-indexed variant**

Applies when $P == 0 \land W == 0$.

$LDRD\{<c>\}{<q>} <Rt>, <Rt2>, [<Rn>], {+/-}<Rm>$

**Pre-indexed variant**

Applies when $P == 1 \land W == 1$.

$LDRD\{<c>\}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]$!

**Decode for all variants of this encoding**

if $Rt<0> == '1'$ then UNPREDICTABLE;

$t = \text{UInt}(Rt);  \ t2 = t+1;  \ n = \text{UInt}(Rn);  \ m = \text{UInt}(Rm);$

$\text{index} = (P == '1');  \ \text{add} = (U == '1');  \ \text{wback} = (P == '0') \ || \ (W == '1');$

if $P == '0' \land W == '1'$ then UNPREDICTABLE;

if $t2 == 15 \ || \ m == 15 \ || \ m == t \ || \ m = t2$ then UNPREDICTABLE;

if wback $\land (n == t \ || \ n == t2)$ then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If $wback \land (n == t \ || \ n == t2)$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

If $P == '0' \land W == '1'$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as an LDRD using one of offset, post-indexed, or pre-indexed addressing.

If $m == t \ || \ m == t2$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
The instruction loads register Rm with an \textit{UNKNOWN} value.

If Rt<0> == '1', then one of the following behaviors must occur:

- The instruction is \textit{UNDEFINED}.
- The instruction executes as \texttt{NOP}.
- The instruction executes with the additional decode: t<0> = '0'.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == '1111'.

**Notes for all encodings**

For more information about the \textit{CONSTRAINED UNPREDICTABLE} behavior of this instruction, see Appendix K1 \textit{Architectural Constraints on UNPREDICTABLE Behaviors}.

**Assembler symbols**

\(<c>\) See \textit{Standard assembler syntax fields} on page F1-7228.

\(<d>\) See \textit{Standard assembler syntax fields} on page F1-7228.

\(<Rt>\) Is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

\(<Rt2>\) Is the second general-purpose register to be transferred. This register must be <R(t+1)>.

\(<Rn>\) Is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

\(+/-\) Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when U = 0
- + when U = 1

\(<Rm>\) Is the general-purpose index register, encoded in the "Rm" field.

**Operation**

\[
\text{if } \text{ConditionPassed()} \text{ then}
\]

\[
\text{EncodingSpecificOperations();}
\]

\[
\text{offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);}
\]

\[
\text{address = if index then offset_addr else R[n];}
\]

\[
\text{if address == Align(address, 8) then}
\]

\[
\text{data = MemA[address,8];}
\]

\[
\text{if BigEndian(AccType_ATOMIC) then}
\]

\[
\text{R[t] = data<63:32>;}\]

\[
\text{R[t2] = data<31:0>;}\]

\[
\text{else}
\]

\[
\text{R[t] = data<31:0>;}\]

\[
\text{R[t2] = data<63:32>;}\]

\[
\text{else}
\]

\[
\text{R[t] = MemA[address,4];}
\]

\[
\text{R[t2] = MemA[address+4,4];}
\]

\[
\text{if wback then R[n] = offset_addr;}
\]
Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.82  LDREX

Load Register Exclusive calculates an address from a base register value and an immediate offset, loads a word from memory, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For information about memory accesses see Memory accesses on page F1-7233.

A1 variant

LDREX{<c>}{<q>} <Rt>, [<Rn> {, {#}<imm>}]

Decode for this encoding

t = UInt(Rt);  n = UInt(Rn);  imm32 = Zeros(32); // Zero offset
if t == 15 || n == 15 then UNPREDICTABLE;

T1 variant

LDREX{<c>}{<q>} <Rt>, [<Rn> {, #<imm>}]

Decode for this encoding

t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- <>  See Standard assembler syntax fields on page F1-7228.
- <>  See Standard assembler syntax fields on page F1-7228.
- <Rt>  Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn>  Is the general-purpose base register, encoded in the "Rn" field.
For encoding A1: the immediate offset added to the value of \(<Rn>\) to calculate the address. \(<imm>\) can only be 0 or omitted.

For encoding T1: the immediate offset added to the value of \(<Rn>\) to calculate the address. \(<imm>\) can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    AArch32.SetExclusiveMonitors(address,4);
    R[t] = MemA[address,4];
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Load Register Exclusive Byte derives an address from a base register value, loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For information about memory accesses see Memory accesses on page F1-7233.

**A1 variant**

LDREXB{<c>}{<q>} <Rt>, [<Rn>]

**Decode for this encoding**

\[ t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \]
\[ \text{if } t == 15 \text{ or } n == 15 \text{ then UNPREDICTABLE; } \]

**T1 variant**

LDREXB{<c>}{<q>} <Rt>, [<Rn>]

**Decode for this encoding**

\[ t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \]
\[ \text{if } t == 15 \text{ or } n == 15 \text{ then UNPREDICTABLE; } // \text{Armv8-A removes UNPREDICTABLE for R13} \]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address,1);
    R[t] = ZeroExtend(MemA[address,1], 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.84 LDREXD

Load Register Exclusive Doubleword derives an address from a base register value, loads a 64-bit doubleword from memory, writes it to two registers and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For information about memory accesses see Memory accesses on page F1-7233.

A1

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| l=1111 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| cond |

A1 variant

LDREXD(<c>{<q>}<Rt>, <Rt2>, [<Rn>])

Decode for this encoding

\[ t = \text{UInt}(Rt); \quad t2 = t + 1; \quad n = \text{UInt}(Rn); \]
\[ \text{if } Rt_{<0>} = '1' || t2 == 15 || n == 15 \text{ then UNPREDICTABLE; } \]

CONSTRANDED UNPREDICTABLE behavior

If \(Rt_{<0>} = '1'\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: \(t_{<0>} = '0'\).
- The instruction executes with the additional decode: \(t2 = t\).
- The instruction executes as described, with no change to its behavior and no additional side effects.

If \(Rt = '1110'\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction is handled as described in Using R15 by instruction on page K1-11563.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| cond |

T1 variant

LDREXD(<c>{<q>}<Rt>, <Rt2>, [<Rn>])
Decode for this encoding

t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if t == 15 || t2 == 15 || t == t2 || n == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13

CONSTRAINED UNPREDICTABLE behavior

If t == t2, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The load instruction executes but the destination register takes an UNKNOWN value.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<*> See Standard assembler syntax fields on page F1-7228.
<#> See Standard assembler syntax fields on page F1-7228.
<Rt> For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field.
For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.
<Rt2> For encoding A1: is the second general-purpose register to be transferred.
For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Rn> Is the general-purpose base register, encoded in the "Rn" field.

Operation for all encodings

if ConditionPassed() then
EncodingSpecificOperations();
address = R[n];
AArch32.SetExclusiveMonitors(address,8);
value = MemA[address,8];
// Extract words from 64-bit loaded value such that R[t] is
// loaded from address and R[t2] from address+4.
R[t] = if BigEndian(AccType_ATOMIC) then value<63:32> else value<31:0>;
R[t2] = if BigEndian(AccType_ATOMIC) then value<31:0> else value<63:32>;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.85    LDREXH

Load Register Exclusive Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For information about memory accesses see Memory accesses on page F1-7233.

**A1**

![Instruction Format](image)

**A1 variant**

LDREXH{<c>}{<q>} <Rt>, [<Rn>]

**Decode for this encoding**

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn);
\]

\[
\text{if } t == 15 \text{ || } n == 15 \text{ then UNPREDICTABLE;}
\]

**T1**

![Instruction Format](image)

**T1 variant**

LDREXH{<c>}{<q>} <Rt>, [<Rn>]

**Decode for this encoding**

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn);
\]

\[
\text{if } t == 15 \text{ || } n == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- <> See Standard assembler syntax fields on page F1-7228.
- <> See Standard assembler syntax fields on page F1-7228.
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address,2);
    R[t] = ZeroExtend(MemA[address,2], 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.86  LDRH (immediate)

Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses on page F1-7233.

A1

LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]

Offset variant

Applies when P == 1 && W == 0.
LDRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>

Post-indexed variant

Applies when P == 0 && W == 0.
LDRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>

Pre-indexed variant

Applies when P == 1 && W == 1.
LDRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>!

Decode for all variants of this encoding

if Rt == '1111' then SEE "LDRH (literal)";
if P == '0' && W == '1' then SEE "LDRHT";

index = (P == '1');
add = (U == '1');
wback = (P == '0') || (W == '1');

if t == 15 || (wback && n == t) then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

LDRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
Decode for this encoding

t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;

T2

![Encoding](#)

**T2 variant**

LDRH{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}]  // <Rt>, <Rn>, <imm> can be represented in T1
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

Decode for this encoding

if Rt == '1111' then SEE "PLD (immediate)"
if Rn == '1111' then SEE "LDRH (literal)"
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13

T3

![Encoding](#)

**Offset variant**

Applies when Rt != 1111 && P == 1 && U == 0 && W == 0.
LDRH{<c>}{<q>} <Rt>, [Rn] {, #<+/-><imm>}

**Post-indexed variant**

Applies when P == 0 && W == 1.
LDRH{<c>}{<q>} <Rt>, [Rn], #<+/-><imm>

**Pre-indexed variant**

Applies when P == 1 && W == 1.
LDRH{<c>}{<q>} <Rt>, [Rn], #<+/-><imm>!

Decode for all variants of this encoding

if Rn == '1111' then SEE "LDRH (literal)"
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLDW (immediate)"
if P == '1' && U == '1' && W == '0' then SEE "LDRHT"
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
**CONSTRAINED UNPREDICTABLE behavior**

If \( w_{back} \land n = t \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is **UNKNOWN**. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

**Notes for all encodings**

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see Appendix K1 Architectural Constraints on **UNPREDICTABLE** Behaviors.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields* on page F1-7228.
- `<q>` See *Standard assembler syntax fields* on page F1-7228.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` For encoding A1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRH (literal).
  
  For encoding T1: is the general-purpose base register, encoded in the "Rn" field.
  
  +/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
  
  - when \( U = 0 \)
  - when \( U = 1 \)
  - when \( U = 1 \)

- `<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.
  
  For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 2, in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as \(<\text{imm}>/2\).
  
  For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
  
  For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

**Operation for all encodings**

```assembly
if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
  else
    if ConditionPassed() then
      EncodingSpecificOperations();
      offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
      address = if index then offset_addr else R[n];
      data = MemU[address,2];
```
if wback then R[n] = offset_addr;
R[t] = ZeroExtend(data, 12);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.87  LDRH (literal)

Load Register Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses on page F1-7233.

A1

LDRH{<c>}{<q>} <Rt>, <label> // Normal form
LDRH{<c>}{<q>} <Rt>, [PC, #{+/-}imm] // Alternative form

Decode for this encoding
if P == '0' && W == '1' then SEE "LDRHT";
t = UInt(Rt);  imm32 = ZeroExtend(imm4H:imm4L, 32);
add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;

CONSTRUANCED UNPREDICTABLE behavior
If wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE;
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRH (immediate). The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15 by instruction on page K1-11563.

T1

LDRH{<c>}{<q>} <Rt>, <label> // Preferred syntax
LDRH{<c>}{<q>} <Rt>, [PC, #{+/-}imm] // Alternative syntax

Decode for this encoding
if Rt == '1111' then SEE "PLD (literal)";
t = UInt(Rt);  imm32 = ZeroExtend(imm22, 32);  add = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<label> For encoding A1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Any value in the range -255 to 255 is permitted.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1.

If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when U = 0
  + when U = 1

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax on page F2-7257.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  base = Align(PC,4);
  address = if add then (base + imm32) else (base - imm32);
  data = MemU[address,2];
  R[t] = ZeroExtend(data, 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**5.1.88 LDRH (register)**

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses on page F1-7233*.

**Offset variant**
Applies when $P == 1 \&\& W == 0$.

$LDRH\{<c>\}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]$

**Post-indexed variant**
Applies when $P == 0 \&\& W == 0$.

$LDRH\{<c>\}{<q>} <Rt>, [<Rn>], {+/-}<Rm>$

**Pre-indexed variant**
Applies when $P == 1 \&\& W == 1$.

$LDRH\{<c>\}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]$

**Decode for all variants of this encoding**

```
if P == '0' \&\& W == '1' then SEE "LDRHT";
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = (P == '1'); add = (U == '1'); wback = (P == '0') \|| (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

**CONSTRAINED UNPREDICTABLE behavior**

If $wback \&\& n == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

**T1**

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
T1 variant

LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

Decode for this encoding

t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);

T2

```
  15 14 13 12|11 10 9 8|7 6 5 4|3 0 | 12 11 10 9 8|7 6 5 4|3 0 |
  1 1 1 1 1 0 0 0 0 0 1 1 | 1 =1111 | 1 =1111 | 0 0 0 0 0 0 |imm2|Rm
```

T2 variant

LDRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

Decode for this encoding

if Rn == '1111' then SEE "LDRH (literal)";
if Rt == '1111' then SEE "PLDW (register)"

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

>
See Standard assembler syntax fields on page F1-7228.

<Rt>
Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn>
For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.
For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

+/
Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
- when U = 0
+ when U = 1

+ Specifies the index register is added to the base register.

<Rm>
Is the general-purpose index register, encoded in the "Rm" field.

<imm>
If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.89 LDRHT

Load Register Halfword Unprivileged loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses on page F1-7233.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

A1

A1 variant

LDRHT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}

Decode for this encoding

\[ t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad \text{postindex} = \text{TRUE}; \quad \text{add} = (U == '1'); \]
\[ \text{register} = \text{FALSE}; \quad \text{imm32} = \text{ZeroExtend}(\text{imm4H:imm4L}, 32); \]
\[ \text{if } t == 15 || n == 15 || n == t \text{ then UNPREDICTABLE; } \]

CONstrained UNPREDICTABLE behavior

If \( n = 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15 by instruction on page K1-11563.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If \( n == t \) \&\& \( n != 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.
A2

LDRHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>

Decode for this encoding

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If n == t && n != 15, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

LDRHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}] 

Decode for this encoding

if Rn == '1111' then SEE "LDRH (literal)";
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>  Is the general-purpose base register, encoded in the "Rn" field.

+/-  For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
    - when U = 0
    + when U = 1

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
    - when U = 0
    + when U = 1

<Rm>  Is the general-purpose index register, encoded in the "Rm" field.

+  Specifies the offset is added to the base register.

<imm>  For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.
For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE_EL == EL2 then UNPREDICTABLE;       // Hyp mode
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,2];
    if postindex then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);

CONSTRANGED UNPREDICTABLE behavior

If PSTATE_EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDRH (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.90 LDRSB (immediate)

Load Register Signed Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses on page F1-7233.

A1

\[
\begin{array}{ccccccccccc}
\hline
l=1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & P & U & 1 & W & 1 & ! & l=1 & 1 & 1 & 1 & Rrt & imm4H & 1 & 1 & 0 & 1 & imm4L \\
\hline
\text{cond} & \text{Rn} \\
\end{array}
\]

Offset variant

Applies when \( P == 1 \land W == 0 \).

\[
\text{LDRSB} \{<c>\} \{<q>\} \text{ Rt}, [\text{Rn} \{, #{+/-}\text{imm}\}]
\]

Post-indexed variant

Applies when \( P == 0 \land W == 0 \).

\[
\text{LDRSB} \{<c>\} \{<q>\} \text{ Rt}, [\text{Rn}], #{+/-}\text{imm}
\]

Pre-indexed variant

Applies when \( P == 1 \land W == 1 \).

\[
\text{LDRSB} \{<c>\} \{<q>\} \text{ Rt}, [\text{Rn}, \{+/-\text{imm}\}]!
\]

Decode for all variants of this encoding

if \( \text{Rn} == '1111' \) then SEE "LDRSB (literal)";
if \( P == '0' \land W == '1' \) then SEE "LDRSBT";
\( t = \text{UInt}(\text{Rt}) \); \( n = \text{UInt}(\text{Rn}) \); \( \text{imm32} = \text{ZeroExtend}(\text{imm4H:imm4L}, 32) \);
\( \text{index} = (P == '1') \); \( \text{add} = (U == '1') \); \( \text{wback} = (P == '0') \land (W == '1') \);
if \( t == 15 \land (\text{wback} \land n == t) \) then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If \( \text{wback} \land n == t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

\[
\begin{array}{ccccccccccc}
| & 15 & 14 & 13 & 12 | 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 | \\
\hline
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & l=1 & 1 & 1 & 1 & \text{imm12} & \\
\hline
\text{Rn} & \text{Rt} \\
\end{array}
\]

T1 variant

\[
\text{LDRSB} \{<c>\} \{<q>\} \text{ Rt}, [\text{Rn} \{, \{+/-\text{imm}\}]
\]
Decode for this encoding

```cpp
if Rt == '1111' then SEE "PLI";
if Rn == '1111' then SEE "LDRSB (literal)";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

### T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>!</td>
<td>1111</td>
<td>Rn</td>
<td>1</td>
<td>P</td>
<td>U</td>
<td>W</td>
<td>imm8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Offset variant**

Applies when Rt != 1111 && P == 1 && U == 0 && W == 0.

LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]!

**Post-indexed variant**

Applies when P == 0 && W == 1.

LDRSB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>

**Pre-indexed variant**

Applies when P == 1 && W == 1.

LDRSB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>!

**Decode for all variants of this encoding**

```cpp
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLI";
if Rn == '1111' then SEE "LDRSB (literal)";
if P == '1' && U == '1' && W == '0' then SEE "LDRSBT";
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

**CONSTRAINED UNPREDICTABLE behavior**

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

\(<c>\)
See Standard assembler syntax fields on page F1-7228.

\(<q>\)
See Standard assembler syntax fields on page F1-7228.

\(<R_t>\)
Is the general-purpose register to be transferred, encoded in the "Rt" field.

\(<R_n>\)
Is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRSB (literal).

\(+/-\)
Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when \( U = 0 \)
+ when \( U = 1 \)

\(+\)
Specifies the offset is added to the base register.

\(<imm>\)
For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  offset_addr = if add then \( R[n] + \text{imm32} \) else \( R[n] - \text{imm32} \);
  address = if index then offset_addr else \( R[n] \);
  \( R[t] = \text{SignExt}(\text{MemU}[address,1], 32) \);
  if wback then \( R[n] = \text{offset_addr} \);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.91 LDRSB (literal)

Load Register Signed Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses on page F1-7233.

A1

```
|31 28|27 26 25 24|23 22 21 20|19 18 17 16|15 12|11 8 7 6 5 4 3 0 |
|---|---|---|---|---|---|---|---|
| 0 0 0 | P | U | 1 | W | 1 | 1 | 1 | 1 | Rt | imm4H | 1 | 1 | 0 | 1 | imm4L |
```

**A1 variant**

Applies when !(P == 0 && W == 1).

LDRSB{<c>}{<q>} <Rt>, <label> // Normal form
LDRSB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative form

**Decode for this encoding**

if P == '0' && W == '1' then SEE "LDRSBT";

\[ t = \text{UInt}(Rt); \ imm32 = \text{ZeroExtend}(\text{imm4H};\text{imm4L}, 32); \]
\[ \text{add} = (U == '1'); \ \text{wback} = (P == '0') || (W == '1'); \]
if \( t == 15 \) \&\& wback then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE;
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRSB (immediate). The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15 by instruction on page K1-11563.

T1

```
|15 14 13 12|11 10 9 8 7 6 5 4 3 2 1 0 |15 12|11 |
|---|---|---|---|---|
| 1 1 1 1 | 0 0 | 1 | 1 | 1 | 1 | 1 | 0 0 | 1 | 1 | 1 | 1 | 0 |
```

**T1 variant**

LDRSB{<c>}{<q>} <Rt>, <label> // Preferred syntax
LDRSB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative syntax

**Decode for this encoding**

if Rt == '1111' then SEE "PLI";

\[ t = \text{UInt}(Rt); \ imm32 = \text{ZeroExtend}(\text{imm2};\text{imm32}, 32); \]
\[ \text{add} = (U == '1'); \]
// Armv8-A removes UNPREDICTABLE for R13
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
>  See Standard assembler syntax fields on page F1-7228.

<q>  See Standard assembler syntax fields on page F1-7228.

<Rt>  Is the general-purpose register to be transferred, encoded in the "Rt" field.

<label>  For encoding A1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Any value in the range -255 to 255 is permitted.

If the offset is zero or positive, \( \text{imm32} \) is equal to the offset and \( add = \text{TRUE} \), encoded as \( U = 1 \). If the offset is negative, \( \text{imm32} \) is equal to minus the offset and \( add = \text{FALSE} \), encoded as \( U = 0 \).

For encoding T1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, \( \text{imm32} \) is equal to the offset and \( add = \text{TRUE} \), encoded as \( U = 1 \).

If the offset is negative, \( \text{imm32} \) is equal to minus the offset and \( add = \text{FALSE} \), encoded as \( U = 0 \).

+-  Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

-  when \( U = 0 \)

+  when \( U = 1 \)

<imm>  For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax on page F2-7257.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  base = Align(PC,4);
  address = if add then (base + imm32) else (base - imm32);
  R[t] = SignExtend(MemU[address,1], 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.92 LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see Memory accesses on page F1-7233.

A1

| 31 28 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 6 5 4 3 0 |
|-----------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| !=1111 | 0 0 0 | P | U | 0 | W | 1 | Rn | 1 | Rt | 0 0 0 | 0 1 0 1 | 1 | 0 | 1 | Rm |

cond

Offset variant

Applies when P == 1 && W == 0.
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]

Post-indexed variant

Applies when P == 0 && W == 0.
LDRSB{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>

Pre-indexed variant

Applies when P == 1 && W == 1.
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]

Decode for all variants of this encoding

if P == '0' && W == '1' then SEE "LDRSBT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

| 15 14 13 12 11 10 9 8 6 5 3 2 0 |
|---------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 0 0 1 1 1 1 | Rm | Rn | 0 1 |

T1 variant

LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

Decode for this encoding

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRTType_LSL, 0);

T2

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3</th>
<th>0 15 12 11 10 9 8 7 6 5 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 0 0 0 1 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>imm2</td>
</tr>
<tr>
<td>Rn</td>
<td>Rt</td>
<td></td>
</tr>
</tbody>
</table>

T2 variant

LDRSB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

Decode for this encoding

if Rt == '1111' then SEE "PLI";
if Rn == '1111' then SEE "LDRSB (literal)";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRTType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<{}> See Standard assembler syntax fields on page F1-7228.

<{}> See Standard assembler syntax fields on page F1-7228.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

+/− Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when \( U = 0 \)
+ when \( U = 1 \)

+ Specifies the index register is added to the base register.

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<imm> If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address, 1], 32);
    if wback then R[n] = offset_addr;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.93   LDRSBT

Load Register Signed Byte Unprivileged loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses on page F1-7233.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRSBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

A1

A1 variant

LDRSBT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}

Decode for this encoding

\[ t = \text{UInt}(Rt); \ n = \text{UInt}(Rn); \ \text{postindex} = \text{TRUE}; \ \text{add} = (U == '1'); \ \text{register}._{\text{form}} = \text{FALSE}; \ \text{imm32} = \text{ZeroExtend}(\text{imm4H:imm4L}, 32); \text{if } t == 15 \ || \ n == 15 \ || \ n == t \text{ then UNPREDICTABLE; } \]

CONSTRARED UNPREDICTABLE behavior

If \( n == 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15 by instruction on page K1-11563.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If \( n == t \ && \ n != 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.
A2

LDRSBT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>

Decode for this encoding

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If n == t && n != 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

LDRSBT{<c>}{<q>} <Rt>, [Rn] {, #<imm>}

Decode for this encoding

if Rn == '1111' then SEE "LDRSB (literal)";
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

See Standard assembler syntax fields on page F1-7228.

See Standard assembler syntax fields on page F1-7228.

(Rt) Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn> Is the general-purpose base register, encoded in the "Rn" field.

+/− For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when U = 0
+ when U = 1

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when U = 0
+ when U = 1

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    R[t] = SignExtend(MemU_unpriv[address,1], 32);
    if postindex then R[n] = offset_addr;

CONSTRUCTED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes as LDRSB (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.94 LDRSH (immediate)

Load Register Signed Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses on page F1-7233.

A1

| 31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 8 7 6 5 4 3 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| !=1111 0 0 0 | P | U | I | W | 1 | !=1111 | Rt | imm4H | 1 | 1 | 1 | 1 | imm4L |
| cond | Rn |

Offset variant

Applies when \( P = 1 \) \&\& \( W = 0 \).

\[
\text{LDRSH}<\{<c>}{<q>}\ <Rt>, \ [<Rn> \{, \#\{+/-\}<imm>\}]
\]

Post-indexed variant

Applies when \( P = 0 \) \&\& \( W = 0 \).

\[
\text{LDRSH}<\{<c>}{<q>\ <Rt>, \ [<Rn>], \#\{+/-\}<imm>
\]

Pre-indexed variant

Applies when \( P = 1 \) \&\& \( W = 1 \).

\[
\text{LDRSH}<\{<c>}{<q}> <Rt>, \ [<Rn>, \#\{+/-\}<imm>]
\]

Decode for all variants of this encoding

If \( Rn = '1111' \) then SEE "LDRSH (literal)";
if \( P = '0' \) \&\& \( W = '1' \) then SEE "LDRSHT";
\( t = \text{UInt}(Rt); n = \text{UInt}(Rn); \text{imm32} = \text{ZeroExtend}(\text{imm4H:imm4L}, 32); \)
\( \text{index} = (P = '1'); \text{add} = (U = '1'); \text{wback} = (P = '0') || (W = '1'); \)
\( \text{if } t = 15 \langle\text{wback }\&\& n = t \rangle \text{ then UNPREDICTABLE;} \)

CONSTRUINED UNPREDICTABLE behavior

If \( \text{wback }\&\& n = t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

| 15 14 13 12|11 10 9 8 7 6 5 4 3 0 |15 12|11 | 0 |
|---|---|---|---|---|---|
| 1 1 1 1 0 0 1 1 0 1 1 | !=1111 | !=1111 | imm12 |
| Rn | Rt |

T1 variant

\[
\text{LDRSH}<\{<c>}{<q}> <Rt>, \ [<Rn> \{, \#\{+\}<imm>\}]
\]
Decode for this encoding

if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' then SEE "Related instructions";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13

T2

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 0</th>
<th>15 12</th>
<th>11 10 9 8 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 0</td>
<td>0 1 1</td>
<td>1</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Offset variant
Applies when Rt != 1111 && P == 1 && U == 0 && W == 0.
LDLSH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]!

Post-indexed variant
Applies when P == 0 && W == 1.
LDLSH{<c>}{<q>} <Rt>, [Rn], #{+/-}imm

Pre-indexed variant
Applies when P == 1 && W == 1.
LDLSH{<c>}{<q>} <Rt>, [Rn], #{+/-}imm!

Decode for all variants of this encoding

if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' then SEE "Related instructions";
if P == '1' && U == '0' && W == '0' then SEE "LDRSHT";
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13

CONSTRAINED UNPREDICTABLE behavior
If wback && n == t, then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

Notes for all encodings
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Related instructions: Load/store single on page F3-7356.
Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

<l> See Standard assembler syntax fields on page F1-7228.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRSH (literal).

<+/-> Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
- when U = 0
+ when U = 1

+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.
For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.95   **LDRSH (literal)**

Load Register Signed Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page F1-7233.

**A1**

![Table](https://example.com/table.png)

**A1 variant**

Applies when !(P == 0 && W == 1).

LDRSH{<c>}{<q>} <Rt>, <label> // Normal form
LDRSH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative form

**Decode for this encoding**

if P == '0' && W == '1' then SEE "LDRSHT";

\[ t = \text{UInt}(Rt); \text{imm32} = \text{ZeroExtend}(\text{imm4H:imm4L}, 32); \]
\[ \text{add} = (U == '1'); \text{wback} = (P == '0') || (W == '1'); \]
\[ \text{if} \ t == 15 || \text{wback then UNPREDICTABLE;} \]

**CONSTRAINED UNPREDICTABLE behavior**

If wback, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction executes with the additional decode: wback = FALSE;
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in **LDRSH (immediate)**. The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in *Using R15 by instruction* on page K1-11563.

**T1**

![Table](https://example.com/table.png)

**T1 variant**

LDRSH{<c>}{<q>} <Rt>, <label> // Preferred syntax
LDRSH{<c>}{<q>} <Rt>, [PC, #{+/-}< imm>] // Alternative syntax

**Decode for this encoding**

if Rt == '1111' then SEE "Related instructions";

\[ t = \text{UInt}(Rt); \text{imm32} = \text{ZeroExtend}(\text{imm2}, 32); \]
\[ \text{add} = (U == '1'); \]
// Armv8-A removes UNPREDICTABLE for R13
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Related instructions: Load, signed (literal) on page F3-7363.

Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

>
See Standard assembler syntax fields on page F1-7228.

\<Rt>\>
Is the general-purpose register to be transferred, encoded in the "Rt" field.

\<label>\>
For encoding A1: the label of the literal data item that is to be loaded into \<Rt>. The assembler calculates the required value of the offset from the Align(\PC, 4) value of the instruction to this label. Any value in the range -255 to 255 is permitted.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into \<Rt>. The assembler calculates the required value of the offset from the Align(\PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

\+-\>
Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when U = 0
  + when U = 1

\<imm>\>
For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax on page F2-7257.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  base = Align(\PC, 4);
  address = if add then (base + imm32) else (base - imm32);
  data = MemU[address, 2];
  \R[t]\ = SignExtend(data, 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see Memory accesses on page F1-7233.

Offset variant
Applies when P == 1 && W == 0.

\[
\text{LDRSH}\{<c>\}{<q>} \text{ <Rt>}, [\text{<Rn>}, \{+/-\}<Rm>] 
\]

Post-indexed variant
Applies when P == 0 && W == 0.

\[
\text{LDRSH}\{<c>\}{<q>} \text{ <Rt>}, [\text{<Rn>}], \{+/-\}<Rm> 
\]

Pre-indexed variant
Applies when P == 1 && W == 1.

\[
\text{LDRSH}\{<c>\}{<q>} \text{ <Rt>}, [\text{<Rn>}, \{+/-\}<Rm>]! 
\]

Decode for all variants of this encoding
if P == '0' && W == '1' then SEE "LDRSHT";
\[ t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \]
\[ \text{index} = (P == '1'); \quad \text{add} = (U == '1'); \quad \text{wback} = (P == '0') \ || \ (W == '1'); \]
\[ (\text{shift}_t, \text{shift}_n) = (\text{SRType\_LSL}, 0); \]
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;

CONstrained UNPREDIcTABLE behavior
If wback && n == t, then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is \(<\text{arm-defined-word}=\text{unknown}\>\). In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1
T1 variant

LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

Decode for this encoding

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRTtype_LSL, 0);

T2

T2 variant

LDRSH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

Decode for this encoding

if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' then SEE "Related instructions";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRTtype_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1
Architectural Constraints on UNPREDICTABLE Behaviors.

Related instructions: Load/store, signed (register offset) on page F3-7360.

Assembler symbols

<< See Standard assembler syntax fields on page F1-7228.

<< See Standard assembler syntax fields on page F1-7228.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used
in the offset variant.
For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

/+ Specify the index register is added to or subtracted from the base register, defaulting to + if omitted
and encoded in the "U" field. It can have the following values:
- when U = 0
+ when U = 1

+ Specifies the index register is added to the base register.

<Rm> Is the general-purpose index register, encoded in the "Rm" field.
<imm>     If present, the size of the left shift to apply to the value from \( R_m \), in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

**Operation for all encodings**

\[
\text{if } \text{ConditionPassed()} \text{ then}
\quad \text{EncodingSpecificOperations();}
\quad \text{offset} = \text{Shift}(R[m], \text{shift}_t, \text{shift}_n, \text{PSTATE.C});
\quad \text{offset_addr} = \text{if add then } (R[n] + \text{offset}) \text{ else } (R[n] - \text{offset});
\quad \text{address} = \text{if index then } \text{offset_addr} \text{ else } R[n];
\quad \text{data} = \text{MemU}[\text{address}, 2];
\quad \text{if wback then } R[n] = \text{offset_addr};
\quad R[t] = \text{SignExtend(data, } 32);\
\]

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.97 LDRSHT

Load Register Signed Halfword Unprivileged loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see Memory accesses on page F1-7233.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRSHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

A1

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>U</td>
<td>1</td>
<td>1</td>
<td>n</td>
<td>Rt</td>
<td>Rn</td>
<td>imm4H</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>imm4L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

cond

A1 variant

LDRSHT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}

Decode for this encoding

t = UInt(Rt); n = UInt(Rn); postindex = TRUE; add = (U == '1');

register_form = FALSE; imm32 = ZeroExtend(imm4H:imm4L, 32);

if t == 15 || n == 15 || n == t then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15 by instruction on page K1-11563.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If n == t & n != 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.
A2

LDRSHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>

Decode for this encoding

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If n == t & n != 15, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction performs all of the loads using the specified addressing mode and the content of the register
  that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base
  address might be corrupted so that the instruction cannot be repeated.

T1

LDRSHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}] 

Decode for this encoding

if Rn == '1111' then SEE "LDRSH (literal)";
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Appendix K.1
Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

See Standard assembler syntax fields on page F1-7228.
<Rn> Is the general-purpose base register, encoded in the "Rn" field.

+/− For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when $U = 0$
+ when $U = 1$

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when $U = 0$
+ when $U = 1$

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.
For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,2];
    if postindex then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);

CONstrained UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes as LDRSH (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.98   LDRT

Load Register Unprivileged loads a word from memory, and writes it to a register. For information about memory accesses see Memory accesses on page F1-7233.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

LDRT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

A1

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>U</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
</tr>
</tbody>
</table>

A1 variant

LDRT{<c>}{<q>} <Rt>, [<Rn>] {, #(+/-)<imm>}

Decode for this encoding

t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If \( n = 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15 by instruction on page K1-11563.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If \( n = t \&\& n != 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.
A2

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1=1111 | 0 | 1 | 1 | 0 | U | 0 | 1 | 1 | Rn | Rt | imm5 |stype | 0 | Rm |

A2 variant

LDRT{<c>}{<q>} <Rt>, [<Rn>], {+/−}<Rm>{, <shift>}

Decode for this encoding

t = UInt(Rt); n = UInt(Rn); m = UInt(Rm); postindex = TRUE; add = (U == '1');
register_form = TRUE; (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If n == t & n != 15, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1=1111 | Rt | 1 | 1 | 1 | 0 | imm8 |

T1 variant

LDRT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

Decode for this encoding

if Rn == '1111' then SEE "LDR (literal)";
t = UInt(Rt); n = UInt(Rn); postindex = FALSE; add = TRUE;
register_form = FALSE; imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<Rt> For encoding A1: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, but this is deprecated.
For encoding A2 and T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Rn>\) Is the general-purpose base register, encoded in the "Rn" field.

\(+/-\) For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when \(U = 0\)
- when \(U = 1\)

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when \(U = 0\)
- when \(U = 1\)

\(<Rm>\) Is the general-purpose index register, encoded in the "Rm" field.

\(<\text{shift}>\) The shift to apply to the value read from \(<Rm>\). If absent, no shift is applied. Otherwise, see \textit{Shifts applied to a register} on page F1-7231.

\(+\) Specifies the offset is added to the base register.

\(<\text{imm}>\) For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

### Operation for all encodings

\[
\text{if } \text{ConditionPassed}() \text{ then } \\
\text{EncodingSpecificOperations();} \\
\text{if PSTATE.EL == EL2 then UNPREDICTABLE;} \quad \text{// Hyp mode} \\
\text{offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;} \\
\text{offset_addr = if add then (R[n] + offset) else (R[n] - offset);} \\
\text{address = if postindex then R[n] else offset_addr;} \\
\text{data = MemU_unpriv[address,4];} \\
\text{if postindex then R[n] = offset_addr;} \\
\text{R[t] = data;}
\]

### CONSTRAINED UNPREDICTABLE behavior

If \(\text{PSTATE.EL} = \text{EL2}\), then one of the following behaviors must occur:

- The instruction is \textit{UNDEFINED}.
- The instruction executes as \textit{NOP}.
- The instruction executes as \textit{LDR} (immediate).

### Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.99  LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

• The encodings in this description are named to match the encodings of MOV, MOVS (register).
• The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

A1

\[
|31|28|27|26|25|24|23|22|21|20|19|18|17|16|15|12|11|7|6|5|4|3|0|
\]

\[
|\text{cond}|S|\text{imm}|\text{stype}|
\]

\[
|1111|0|0|0|1|1|0|1|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|
\]

\[
|\text{Rd}|=00000|\text{Rm}|=00000|
\]

**MOV, shift or rotate by value variant**

\[
\text{LSL}(<c>{<q>}) (<Rd>,) (<Rm>,) \#<imm>
\]

is equivalent to

\[
\text{MOV}(<c>{<q>}) (<Rd>, <Rm>, \text{LSL} \#<imm>)
\]

and is always the preferred disassembly.

T2

\[
|15|14|13|12|11|10|6|5|3|2|0|
\]

\[
|0|0|0|0|0|0|0|0|0|0|0|0|
\]

\[
|\text{op}|\text{imm}|\text{Rd}|\text{Rm}|
\]

**T2 variant**

\[
\text{LSL}<c>{<q>} (<Rd>,) (<Rm>,) \#<imm> // Inside IT block
\]

is equivalent to

\[
\text{MOV}<c>{<q>} (<Rd>, <Rm>, \text{LSL} \#<imm>)
\]

and is the preferred disassembly when InITBlock().

T3

\[
|15|14|13|12|11|10|9|8|7|6|5|4|3|2|1|0|15|14|12|11|8|7|6|5|4|3|0|
\]

\[
|1|1|1|0|1|0|1|0|0|1|0|0|1|1|1|1|0|0|1|1|1|1|0|0|0|0|1|0|0|0|
\]

\[
|\text{S}|\text{imm}|\text{stype}|
\]

\[
|\text{Rd}|\text{imm}|\text{Rm}|\text{Rm}|
\]

**MOV, shift or rotate by value variant**

\[
\text{LSL}<c>{.W} (<Rd>,) (<Rm>,) \#<imm> // Inside IT block, and \langle Rd\rangle, \langle Rm\rangle, \langle imm\rangle can be represented in T2
\]

is equivalent to

\[
\text{MOV}<c>{.W} (<Rd>, <Rm>, \text{LSL} \#<imm>)
\]

and is always the preferred disassembly.

\[
\text{LSL}<c>{<q>} (<Rd>,) (<Rm>,) \#<imm>
\]
is equivalent to

\[
\text{MOV}\{<c>\}\{<q>\} \ <Rd>, \ <Rm>, \ LSL \ #\langle\text{imm}\rangle
\]

and is always the preferred disassembly.

**Assembler symbols**

- \(<c>\) See *Standard assembler syntax fields on page F1-7228.*
- \(<q>\) See *Standard assembler syntax fields on page F1-7228.*
- \(<Rd>\) For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm depreciates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.* For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
- \(<Rm>\) For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is depreciated. For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.
- \(<\text{imm}>\) For encoding A1: is the shift amount, in the range 0 to 31, encoded in the "imm5" field as \(<\text{imm}>\) modulo 32. For encoding T2: is the shift amount, in the range 1 to 31, encoded in the "imm5" field as \(<\text{amount}>\) modulo 32. For encoding T3: is the shift amount, in the range 0 to 31, encoded in the "imm3:imm2" field as \(<\text{imm}>\) modulo 32.

**Operation for all encodings**

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.
Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

### A1

```
| 31  | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 12 | 11 |  8 |  7 |  6 |  5 |  4 |  3 |  0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| l=1111 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | (0) | (0) | (0) | (0) | Rd | Rs | 0 | 0 | 0 | 1 | Rm |
```

**Not flag setting variant**

$$\text{LSL}\{\langle c\rangle\}\{\langle q\rangle\} \{\langle Rdm\rangle,\} \langle Rm\rangle, \langle Rs\rangle$$

is equivalent to

$$\text{MOV}\{\langle c\rangle\}\{\langle q\rangle\} \langle Rdm\rangle, \langle Rm\rangle, \text{LSL} \langle Rs\rangle$$

and is always the preferred disassembly.

### T1

```
| 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | Rs | Rdm |
```

**Logical shift left variant**

$$\text{LSL}\{\langle q\rangle\} \{\langle Rdm\rangle,\} \langle Rdm\rangle, \langle Rm\rangle, \langle Rs\rangle \quad \text{// Inside IT block}$$

is equivalent to

$$\text{MOV}\{\langle q\rangle\} \langle Rdm\rangle, \langle Rdm\rangle, \text{LSL} \langle Rs\rangle$$

and is the preferred disassembly when InITBlock().

### T2

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rd</td>
</tr>
</tbody>
</table>
```

**Not flag setting variant**

$$\text{LSL}\{\langle c\rangle,\langle W\rangle\} \{\langle Rm\rangle,\} \langle Rm\rangle, \langle Rs\rangle \quad \text{// Inside IT block, and} \langle Rd\rangle, \langle Rm\rangle, \langle \text{shift}\rangle, \langle Rs\rangle \text{ can be represented in } T1$$

is equivalent to

$$\text{MOV}\{\langle c\rangle\}\{\langle q\rangle\} \langle Rdm\rangle, \langle Rm\rangle, \text{LSL} \langle Rs\rangle$$

and is always the preferred disassembly.
LSL{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
is equivalent to
MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
and is always the preferred disassembly.

**Assembler symbols**

- `<c>`: See *Standard assembler syntax fields on page F1-7228*.
- `<q>`: See *Standard assembler syntax fields on page F1-7228*.
- `<Rdm>`: Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
- `<Rd>`: Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>`: Is the first general-purpose source register, encoded in the "Rm" field.
- `<Rs>`: Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

**Operation for all encodings**

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.
F5.1.101  LSLS (immediate)

Logical Shift Left, setting flags (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

**A1**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>I=1111</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td>S</td>
<td>imm5</td>
<td>stype</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MOV$\,$ shift or rotate by value variant**

LSLS{<c>{<q>}}{<Rd>,} <Rm>, #$<imm>

is equivalent to

MOV$\,${<c>{<q>}} <Rd>, <Rm>, LSL #$<imm>

and is always the preferred disassembly.

**T2**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>op</td>
<td>imm5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**T2 variant**

LSLS{<q>}{<Rd>,} <Rm>, #$<imm> // Outside IT block

is equivalent to

MOV$\,${<q>}{<Rd>,} <Rm>, LSL #$<imm>

and is the preferred disassembly when !InITBlock().
T3

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1  | 1  | 1  | 0  | 0  | 0  | 0 | 1 | 1 | 1 | 1 | (0)| imm3| Rd | imm2| 0 | 0 | Rm |

**MOVS, shift or rotate by value variant**

LSLS.W {<Rd>,} <Rm>, #<imm> // Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>

and is always the preferred disassembly.

LSLS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>

and is always the preferred disassembly.

**Assembler symbols**

<

See *Standard assembler syntax fields on page F1-7228.*

<

See *Standard assembler syntax fields on page F1-7228.*

<Rd>

For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_{current_mode}.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

<Rm>

For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

<imm>

For encoding A1: is the shift amount, in the range 0 to 31, encoded in the "imm5" field as <imm> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31, encoded in the "imm5" field as <amount> modulo 32.

For encoding T3: is the shift amount, in the range 0 to 31, encoded in the "imm3:imm2" field as <imm> modulo 32.

**Operation for all encodings**

The description of MOV, MOVs (register) gives the operational pseudocode for this instruction.
F5.1.102  LSLS (register)

Logical Shift Left, setting flags (register) shifts a register value left by a variable number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVs (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVs (register-shifted register).
- The description of MOV, MOVs (register-shifted register) gives the operational pseudocode for this instruction.

**A1**

<table>
<thead>
<tr>
<th>31 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12</th>
<th>11 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111</td>
<td>Rd</td>
</tr>
<tr>
<td>cond</td>
<td>Rs</td>
</tr>
<tr>
<td>S</td>
<td>stype</td>
</tr>
</tbody>
</table>

**Flag setting variant**

LSLS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>

and is always the preferred disassembly.

**T1**

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>6 5</th>
<th>3 2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 0 0 0 1 0</td>
<td>Rs</td>
<td>Rdm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Logical shift left variant**

LSLS{<q>} {<Rdm>,} <Rdm>, <Rs> // Outside IT block

is equivalent to

MOV{<q>} <Rdm>, <Rdm>, LSL <Rs>

and is the preferred disassembly when !InITBlock().

**T2**

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 0 1 0 0 0 0 1</td>
<td>Rm</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>stype</td>
<td>S</td>
<td></td>
</tr>
</tbody>
</table>

**Flag setting variant**

LSLS.W {<Rd>,} <Rm>, <Rs> // Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
and is always the preferred disassembly.

$LSL\{<c>\}{<q>\} \{<Rd>,\} \ <Rm>, \ <Rs>$

is equivalent to

$MOVS\{<c>\}{<q>\} \ <Rd>, \ <Rm>, \ LSL \ <Rs>$

and is always the preferred disassembly.

**Assembler symbols**

\(<c>\) See Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<Rdm>\) Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

\(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.

\(<Rm>\) Is the first general-purpose source register, encoded in the "Rm" field.

\(<Rs>\) Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

**Operation for all encodings**

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.
F5.1.103  LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

A1

```
MOV, shift or rotate by value variant
LSR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

T2

```
MOV, shift or rotate by value variant
LSR<c>.W {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV<c>.W <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly when InITBlock().
is equivalent to

\[ \text{MOV}\{<c>\}\{<q>\} \text{ } <Rd>, \text{ } <Rm>, \text{ } \text{LSR } #<\text{imm}> \]

and is always the preferred disassembly.

**Assembler symbols**

- \(<c>\) See *Standard assembler syntax fields* on page F1-7228.
- \(<q>\) See *Standard assembler syntax fields* on page F1-7228.
- \(<Rd>\) For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-7133. For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
- \(<Rm>\) For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.
- \(<\text{imm}>\) For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as \(<\text{imm}>\) modulo 32. For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as \(<\text{imm}>\) modulo 32.

**Operation for all encodings**

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.
F5.1.104 LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVs (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVs (register-shifted register).
- The description of MOV, MOVs (register-shifted register) gives the operational pseudocode for this instruction.

### A1

<table>
<thead>
<tr>
<th>[31]</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Not flag setting variant

LSR<>{<q>} {<Rm>,} {<Rd>,} <Rs>

is equivalent to

MOV<>{<q>} <Rd>, <Rm>, LSR <Rs>

and is always the preferred disassembly.

### T1

<table>
<thead>
<tr>
<th>[15]</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rs</td>
<td>Rdm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Logical shift right variant

LSR<>{<q>} {<Rd>,} {<Rm>,} <Rs> // Inside IT block

is equivalent to

MOV<>{<q>} <Rd>, <Rm>, LSR <Rs>

and is the preferred disassembly when InITBlock().

### T2

<table>
<thead>
<tr>
<th>[15]</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>Rd</td>
</tr>
<tr>
<td>stype</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Not flag setting variant

LSR<>{<q>}W {<Rd>,} <Rs> // Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1

is equivalent to

MOV<>{<q>} <Rd>, <Rm>, LSR <Rs>

and is always the preferred disassembly.
LSR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
is equivalent to
MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
and is always the preferred disassembly.

**Assembler symbols**

<{}> See *Standard assembler syntax fields on page F1-7228.*

<q> See *Standard assembler syntax fields on page F1-7228.*

<Rdm> Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> Is the first general-purpose source register, encoded in the "Rm" field.

<Rs> Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

**Operation for all encodings**

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.
F5.1.105   LSRS (immediate)

Logical Shift Right, setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for \(<Rd>\) identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_\(<current\_mode>\).
- The PE checks SPSR_\(<current\_mode>\) for an illegal return event. See **Illegal return events from AArch32 state** on page G1-8952.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

### A1

<table>
<thead>
<tr>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
</tr>
<tr>
<td>27</td>
</tr>
<tr>
<td>26</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

**MOV, shift or rotate by value variant**

\(\text{LSRS\{} \langle c \rangle \{ \langle q \rangle \} \{ \langle Rd \rangle , \} \langle Rm \rangle , \# \langle \text{imm} \rangle \)**

is equivalent to

\(\text{MOVS\{} \langle c \rangle \{ \langle q \rangle \} \langle Rd \rangle , \langle Rm \rangle , \text{LSR} \# \langle \text{imm} \rangle \)**

and is always the preferred disassembly.

### T2

<table>
<thead>
<tr>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

**T2 variant**

\(\text{LSRS\{} \langle q \rangle \} \{ \langle Rd \rangle , \} \langle Rm \rangle , \# \langle \text{imm} \rangle \) // Outside IT block

is equivalent to

\(\text{MOVS\{} \langle q \rangle \} \langle Rd \rangle , \langle Rm \rangle , \text{LSR} \# \langle \text{imm} \rangle \)**

and is the preferred disassembly when !\text{InITBlock}().
T3

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>15 14 12 11 8 7 6 5 4 3 0</th>
<th>S stype</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 0 1 0 1 1 1 1</td>
<td>0 1</td>
<td>Rd imm3 Rm</td>
</tr>
</tbody>
</table>

**MOVS, shift or rotate by value variant**

LSRS.W {<Rd>,} <Rm>, #<imm> // Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>

and is always the preferred disassembly.

LSRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>

and is always the preferred disassembly.

**Assembler symbols**

- `<c>`: See *Standard assembler syntax fields on page F1-7228*.
- `<q>`: See *Standard assembler syntax fields on page F1-7228*.
- `<Rd>`: For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.
  
  For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

- `<Rm>`: For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
  
  For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

- `<imm>`: For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as `<imm>` modulo 32.
  
  For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as `<imm>` modulo 32.

**Operation for all encodings**

The description of MOV, MOVs (register) gives the operational pseudocode for this instruction.
F5.1.106 LSRS (register)

Logical Shift Right, setting flags (register) shifts a register value right by an immediate number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

A1

Flag setting variant

LSRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>

and is always the preferred disassembly.

Logical shift right variant

LSRS{<q>} {<Rdm>,} <Rdm>, <Rs> // Outside IT block

is equivalent to

MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs>

and is the preferred disassembly when !InITBlock().

Flag setting variant

LSRS.W {<Rd>,} <Rm>, <Rs> // Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
and is always the preferred disassembly.

\texttt{LSRS\{<c>\}\{<q>\} \{<Rd>,\} <Rm>, <Rs>}

is equivalent to

\texttt{MOVS\{<c>\}\{<q>\} <Rd>, <Rm>, LSR <Rs>}

and is always the preferred disassembly.

\textbf{Assembler symbols}

\(<c>\quad\text{See Standard assembler syntax fields on page F1-7228.}\)

\(<q>\quad\text{See Standard assembler syntax fields on page F1-7228.}\)

\(<Rdm>\quad\text{Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.}\)

\(<Rd>\quad\text{Is the general-purpose destination register, encoded in the "Rd" field.}\)

\(<Rm>\quad\text{Is the first general-purpose source register, encoded in the "Rm" field.}\)

\(<Rs>\quad\text{Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.}\)

\textbf{Operation for all encodings}

The description of \texttt{MOV, MOVS} (register-shifted register) gives the operational pseudocode for this instruction.
F5.1.107 MCR

Move to System register from general-purpose register or execute a System instruction. This instruction copies the value of a general-purpose register to a System register, or executes a System instruction.

The System register and System instruction descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see About the AArch32 System register interface on page E1-7158 and General behavior of System registers on page G8-9322.

In an implementation that includes EL2, MCR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCR instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see EL2 configurable controls on page G1-9010.

Because of the range of possible traps to Hyp mode, the MCR pseudocode does not show these possible traps.

A1

\[
\begin{array}{c|cccccccccccc}
\text{cond} & 31 & 28 & 27 & 26 & 25 & 24 & 23 & 21 & 20 & 19 & 16 & 15 & 12 & 11 & 9 & 8 & 7 & 5 & 4 & 3 & 0 \\
\hline
1 & 1 & 1 & 1 & 0 & \text{opc1} & 0 & \text{CRn} & \text{Rt} & 1 & 1 & 1 & \text{opc2} & 1 & \text{CRm} & 0
\end{array}
\]

coproc<0>

coproc<3:1>

A1 variant

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

Decode for this encoding

\[
t = \text{UInt}(Rt); \quad \text{cp} = \text{if coproc<0> == '0' then 14 else 15}; \\
\text{if } t == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

T1

\[
\begin{array}{c|cccccccccccc}
\text{cond} & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 5 & 4 & 3 & 0 & 15 & 12 & 11 & 9 & 8 & 7 & 5 & 4 & 3 & 0 \\
\hline
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & \text{opc1} & 0 & \text{CRn} & \text{Rt} & 1 & 1 & 1 & \text{opc2} & 1 & \text{CRm} & 0
\end{array}
\]

coproc<0>

coproc<3:1>

T1 variant

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

Decode for this encoding

\[
t = \text{UInt}(Rt); \quad \text{cp} = \text{if coproc<0> == '0' then 14 else 15}; \\
\text{if } t == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<coproc> Is the System register encoding space, encoded in the "coproc<0>" field. It can have the following values:
  p14 when coproc<0> = 0
  p15 when coproc<0> = 1

<opc1> Is the opc1 parameter within the System register encoding space, in the range 0 to 7, encoded in the "opc1" field.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<CRn> Is the CRn parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRn" field.

<CRm> Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.

<opc2> Is the opc2 parameter within the System register encoding space, in the range 0 to 7, encoded in the "opc2" field.

The possible values of { <coproc>, <opc1>, <CRn>, <CRm>, <opc2> } encode the entire System register and System instruction encoding space. Not all of this space is allocated, and the System register and System instruction descriptions identify the allocated encodings.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  AArch32.SysRegWrite(cp, ThisInstr(), t);
F5.1.108 MCRR

Move to System register from two general-purpose registers. This instruction copies the values of two general-purpose registers to a System register.

The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see About the AArch32 System register interface on page E1-7158 and General behavior of System registers on page G8-9322.

In an implementation that includes EL2, MCRR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCRR instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see EL2 configurable controls on page G1-9010.

Because of the range of possible traps to Hyp mode, the MCRR pseudocode does not show these possible traps.

A1

```
| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 9 | 8 | 7 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=1111 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | Rt2 | Rt | 1 | 1 | 1 | opc1 | CRm |
```

**cond**

**coproc<0>**

**coproc<3:1>**

**A1 variant**

MCRR{c}{q} <coproc>, {#<opc1>}, <Rt>, <Rt2>, <CRm>

**Decode for this encoding**

\[
t = \text{UInt(Rt)}; \quad t2 = \text{UInt(Rt2)}; \quad cp = \text{if coproc<0> == '0' then 14 else 15}; \\
\text{if } t == 15 || t2 == 15 \text{ then UNPREDICTABLE}; \\
\text{// Armv8-A removes UNPREDICTABLE for R13}
\]

T1

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rt2</td>
<td>Rt</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>opc1</td>
<td>CRm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**coproc<0>**

**coproc<3:1>**

**T1 variant**

MCRR{c}{q} <coproc>, {#<opc1>}, <Rt>, <Rt2>, <CRm>

**Decode for this encoding**

\[
t = \text{UInt(Rt)}; \quad t2 = \text{UInt(Rt2)}; \quad cp = \text{if coproc<0> == '0' then 14 else 15}; \\
\text{if } t == 15 || t2 == 15 \text{ then UNPREDICTABLE}; \\
\text{// Armv8-A removes UNPREDICTABLE for R13}
\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<<c>> See Standard assembler syntax fields on page F1-7228.

<<q>> See Standard assembler syntax fields on page F1-7228.

<coproce> Is the System register encoding space, encoded in the "coproc<0>" field. It can have the following values:
 p14 when coproc<0> = 0
 p15 when coproc<0> = 1

<opc1e> Is the opc1 parameter within the System register encoding space, in the range 0 to 15, encoded in the "opc1" field.

<Re> Is the first general-purpose register that is transferred into, encoded in the "Rt" field.

<Rt2e> Is the second general-purpose register that is transferred into, encoded in the "Rt2" field.

<CRme> Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.

The possible values of {<coproc>, <opc1>, <CRm>} encode the entire System register encoding space. Not all of this space is allocated, and the System register descriptions identify the allocated encodings.

For the permitted uses of these instructions, as described in this manual, <Rt2> transfers bits[63:32] of the selected System register, while <Rt> transfers bits[31:0].

Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations();
   AArch32.SysRegWrite64(cp, ThisInstr(), t, t2);
F5.1.109  MLA, MLAS

Multiply Accumulate multiplies two register values, and adds a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

In an A32 instruction, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

### Flag setting variant
Applies when \( S = 1 \).

\[
\text{MLAS}\{<c>\}\{<q>\} <Rd>, <Rn>, <Rm>, <Ra>
\]

### Not flag setting variant
Applies when \( S = 0 \).

\[
\text{MLA}\{<c>\}\{<q>\} <Rd>, <Rn>, <Rm>, <Ra>
\]

#### Decode for all variants of this encoding
\[
d = \text{UInt}(Rd); n = \text{UInt}(Rn); m = \text{UInt}(Rm); a = \text{UInt}(Ra); \text{setflags} = (S == '1');
\]

\[
\text{if } d == 15 || n == 15 || m == 15 || a == 15 \text{ then UNPREDICTABLE;}
\]

### T1

#### T1 variant
\[
\text{MLA}\{<c>\}\{<q>\} <Rd>, <Rn>, <Rm>, <Ra>
\]

#### Decode for this encoding
\[
\text{if } Ra == '1111' \text{ then SEE "MUL";}
\]

\[
d = \text{UInt}(Rd); n = \text{UInt}(Rn); m = \text{UInt}(Rm); a = \text{UInt}(Ra); \text{setflags} = \text{FALSE};
\]

\[
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

#### Notes for all encodings
For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

#### Assembler symbols

\(<c>\) See Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.
<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = operand1 * operand2 + addend;
    R[d] = result<31:0>;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result<31:0>);
        // PSTATE.C, PSTATE.V unchanged
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.110   MLS

Multiply and Subtract multiplies two register values, and subtracts the product from a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

A1

| 31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 8 7 6 5 4 3 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|  !=1111 0 0 0 0 0 1 1 0 | Rd | Ra | Rm | 1 0 0 1 | Rn |

**A1 variant**

MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

**Decode for this encoding**

\[
\begin{align*}
  d &= \text{UInt}(Rd); \\
  n &= \text{UInt}(Rn); \\
  m &= \text{UInt}(Rm); \\
  a &= \text{UInt}(Ra); \\
  \text{if } d == 15 || n == 15 || m == 15 || a == 15 \text{ then UNPREDICTABLE};
\end{align*}
\]

T1

| 15 14 13 12|11 10 9 8 7 6 5 4 3 | 0 | 15 12|11 8 7 6 5 4 3 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 1 1 1 0 1 1 0 0 0 0 | Rn | Ra | Rd | 0 0 0 1 | Rm |

**T1 variant**

MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

**Decode for this encoding**

\[
\begin{align*}
  d &= \text{UInt}(Rd); \\
  n &= \text{UInt}(Rn); \\
  m &= \text{UInt}(Rm); \\
  a &= \text{UInt}(Ra); \\
  \text{if } d == 15 || n == 15 || m == 15 || a == 15 \text{ then UNPREDICTABLE}; \\
  \text{// Armv8-A removes UNPREDICTABLE for R13}
\end{align*}
\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>`  
  See *Standard assembler syntax fields on page F1-7228*.

- `<q>`  
  See *Standard assembler syntax fields on page F1-7228*.

- `<Rd>`  
  Is the general-purpose destination register, encoded in the "Rd" field.

- `<Rn>`  
  Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

- `<Rm>`  
  Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

- `<Ra>`  
  Is the third general-purpose source register holding the minuend, encoded in the "Ra" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = addend - operand1 * operand2;
    R[d] = result<31:0>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.111  MOV, MOVS (immediate)

Move (immediate) writes an immediate value to the destination register.

If the destination register is not the PC, the MOVS variant of the instruction updates the condition flags based on the result.

The field descriptions for \(<Rd>\) identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The MOV variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- The MOVS variant of the instruction performs an exception return without the use of the stack. In this case:
  — The PE branches to the address written to the PC, and restores PSTATE from SPSR_\(<current\_mode>\).
  — The PE checks SPSR_\(<current\_mode>\) for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  — The instruction is UNDEFINED in Hyp mode.
  — The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

### A1

\[
\begin{array}{ccccccccccccc}
1=1111 & 0 & 0 & 1 & 1 & 0 & 1 & S & 0 & 0 & 0 & 0 & 0 & 0 & imm12 \\
\end{array}
\]

\(\text{cond}\)

**MOV variant**

Applies when \(S == 0\).

\[
\text{MOV}\{<c>\}{<q>} \ <Rd>, \ #<const>
\]

**MOVS variant**

Applies when \(S == 1\).

\[
\text{MOVS}\{<c>\}{<q>} \ <Rd>, \ #<const>
\]

**Decode for all variants of this encoding**

\[
d = \text{UInt}(Rd); \ \text{setflags} = (S == '1'); \ (\text{imm32}, \text{carry}) = \text{A32ExpandImm}_C(\text{imm12}, \text{PSTATE}.C);
\]

### A2

\[
\begin{array}{cccccccccccc}
1=1111 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & imm4 & Rd & imm12 \\
\end{array}
\]

\(\text{cond}\)

**A2 variant**

\[
\text{MOV}\{<c>\}{<q>} \ <Rd>, \ #<imm16> \ // <imm16> \text{ can not be represented in A1}
\]

\[
\text{MOIW}\{<c>\}{<q>} \ <Rd>, \ #<imm16> \ // <imm16> \text{ can be represented in A1}
\]

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \ \text{setflags} = \text{FALSE}; \ \text{imm32} = \text{ZeroExtend}(\text{imm4:imm12}, 32);
\]

\(\text{if } d == 15 \text{ then UNPREDICTABLE; }\)
T1

```
| 15 14 13 12|11 10  8 7  |  0 |
|0 0 1 0 0| Rd | imm8 |
```

### T1 variant

MOV<c>{<q>} <Rd>, #<imm8> // Inside IT block
MOVS{<q>} <Rd>, #<imm8> // Outside IT block

### Decode for this encoding

\[
d = \text{UInt}(Rd); \text{ setflags} = \neg \text{InITBlock}(); \text{ imm32} = \text{ZeroExtend}(\text{imm8}, 32); \text{ carry} = \text{PSTATE.C};
\]

T2

```
| 15 14 13 12|11 10  8 7  |  0 |
|1 1 1 1 0 | 0 0 0 1 0| S 1 1 1 0 | imm3 | Rd | imm8 |
```

### MOV variant

Applies when \( S = 0 \).

MOV<c>.W <Rd>, #<const> // Inside IT block, and <Rd>, <const> can be represented in T1
MOV{<c>}{<q>} <Rd>, #<const>

### MOVS variant

Applies when \( S = 1 \).

MOVS.W <Rd>, #<const> // Outside IT block, and <Rd>, <const> can be represented in T1
MOVS{<c>}{<q>} <Rd>, #<const>

### Decode for all variants of this encoding

\[
d = \text{UInt}(Rd); \text{ setflags} = (S == '1'); \text{ (imm32, carry)} = \text{T32ExpandImm.C(i:imm3:imm8, PSTATE.C)};\]
\[
\text{if } d == 15 \text{ then UNPREDICTABLE; } // \text{Armv8-A removes UNPREDICTABLE for R13}
\]

T3

```
| 15 14 13 12|11 10  8 7  |  0 |
|1 1 1 1 0 | 1 0 0 1 0 | imm4 | 0 | imm3 | Rd | imm8 |
```

### T3 variant

MOV{<c>}{<q>} <Rd>, #<imm16> // <imm16> cannot be represented in T1 or T2
MOVW{<c>}{<q>} <Rd>, #<imm16> // <imm16> can be represented in T1 or T2

### Decode for this encoding

\[
d = \text{UInt}(Rd); \text{ setflags} = \text{FALSE}; \text{ imm32} = \text{ZeroExtend}(\text{imm4:i:imm3:imm8}, 32);\]
\[
\text{if } d == 15 \text{ then UNPREDICTABLE; } // \text{Armv8-A removes UNPREDICTABLE for R13}
\]
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

>
See Standard assembler syntax fields on page F1-7228.

<Rd>
For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used:
- For the MOV variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- For the MOVS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding A2, T1, T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

<imm8>
Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

<imm16>
For encoding A2: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field.

For encoding T3: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:i:imm3:imm8" field.

<const>
For encoding A1: an immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values.

For encoding T2: an immediate value. See Modified immediate constants in T32 instructions on page F1-7242 for the range of values.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  result = imm32;
  if d == 15 then          // Can only occur for encoding A1
    if setflags then
      ALUExceptionReturn(result);
    else
      ALUWritePC(result);
  else
    R[d] = result;
    if setflags then
      PSTATE.N = result<31>;
      PSTATE.Z = IsZeroBit(result);
      PSTATE.C = carry;
      // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.112 MOV, MOVS (register)

Move (register) copies a value from a register to the destination register.

If the destination register is not the PC, the MOVS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The MOV variant of the instruction is a branch. In the T32 instruction set (encoding T1) this is a simple branch, and in the A32 instruction set it is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.

- The MOVS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is used by the aliases ASRS (immediate), ASR (immediate), LSLS (immediate), LSL (immediate), LSRS (immediate), LSR (immediate), RORS (immediate), ROR (immediate), RRXS, and RRX. See Alias conditions on page F5-7721 for details of when each alias is preferred.

A1

```
31 28 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 12 | 11 | 7 6 5 4 | 3 | 0 |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
</tr>
<tr>
<td>0 0 0 0 0 1 0 1</td>
</tr>
<tr>
<td>cond</td>
</tr>
</tbody>
</table>
```

**MOV, rotate right with extend variant**

Applies when $S = 0 \&\& imm5 = 00000 \&\& stype = 11$.

```
MOV{<c>}{<q>} <Rd>, <Rm>, RX
```

**MOV, shift or rotate by value variant**

Applies when $S = 0 \&\& !(imm5 = 00000 \&\& stype = 11)$.

```
MOV{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

**MOVS, rotate right with extend variant**

Applies when $S = 1 \&\& imm5 = 00000 \&\& stype = 11$.

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RX
```

**MOVS, shift or rotate by value variant**

Applies when $S = 1 \&\& !(imm5 = 00000 \&\& stype = 11)$.

```
MOVS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

**Decode for all variants of this encoding**

```
d = UInt(Rd);  m = UInt(Rm);  setflags = ($ == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```
T1

\[ \begin{array}{cccccc|cc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 3 & 2 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & D & Rm & Rd
\end{array} \]

**T1 variant**

`MOV{<c>}{<q>} <Rd>, <Rm>`

*Decode for this encoding*

\[
d = \text{UInt}(D:Rd); \quad m = \text{UInt}(Rm); \quad \text{setflags = FALSE};
\]

\[ (\text{shift}_t, \text{shift}_n) = (\text{SRType}_{LSL}, 0); \]

\[ \text{if } d == 15 \text{ } \&\& \text{ } \text{InITBlock()} \text{ } \&\& \text{ } \text{!LastInITBlock()} \text{ then UNPREDICTABLE; } \]

T2

\[ \begin{array}{cccccc|cc}
15 & 14 & 13 & 12 & 11 & 10 & 6 & 5 & 3 & 2 & 0 \\
0 & 0 & 0 & I=11 & \text{imm}5 & Rm & Rd
\end{array} \]

**T2 variant**

`MOV{<c>}{<q>} <Rd>, <Rm>` \{, \text{<shift>} \#\text{<amount>}\} // Inside IT block

`MOVS{<q>} <Rd>, <Rm>` \{, \text{<shift>} \#\text{<amount>}\} // Outside IT block

*Decode for this encoding*

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad \text{setflags = !InITBlock();}
\]

\[ (\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{op}, \text{imm}5); \]

\[ \text{if } \text{op} == '00' \text{ } \&\& \text{ } \text{imm}5 == '00000' \text{ } \&\& \text{ } \text{InITBlock()} \text{ then UNPREDICTABLE; } \]

**CONSTRAINED UNPREDICTABLE behavior**

If \(\text{op} == '00' \text{ } \&\& \text{ } \text{imm}5 == '00000' \text{ } \&\& \text{ } \text{InITBlock()}

then one of the following behaviors must occur:

- The instruction is \text{UNDEFINED}.
- The instruction executes as if it passed its condition code check.
- The instruction executes as \text{NOP}, as if it failed its condition code check.
- The instruction executes as \text{MOV} \text{Rd}, \text{Rm}.

T3

\[ \begin{array}{cccccc|cccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 14 & 12 & 11 & 8 & 7 & 6 & 5 & 4 & 3 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & S & 1 & 1 & 1 & 1 & (0) & \text{imm}3 & \text{Rd} & \text{imm}2 & \text{stype} & \text{Rm}
\end{array} \]

**MOV, rotate right with extend variant**

Applies when \( S = 0 \text{ } \&\& \text{imm}3 = 000 \text{ } \&\& \text{imm}2 = 00 \text{ } \&\& \text{stype} = 11. \)

`MOV{<c>}{<q>} <Rd>, <Rm}, RRX`
MOV, shift or rotate by value variant

Applies when \( S = 0 \) \&\& \((\text{imm3} = 000 \&\& \text{imm2} = 00 \&\& \text{stype} = 11)\).

\[
\text{MOV}(<c>).W <Rd>, <Rm> \{, \text{LSL} #0\} // <Rd>, <Rm> \text{ can be represented in T1}
\]

\[
\text{MOV}(<c>).W <Rd>, <Rm> \{, <\text{shift}> #<\text{amount}>\} // \text{Inside IT block, and <Rd>, <Rm>, <\text{shift}>, <\text{amount}> \text{ can be represented in T2}
\]

\[
\text{MOV}(<c>)\{<q>\} <Rd>, <Rm> \{, <\text{shift}> #<\text{amount}>\}
\]

MOV, rotate right with extend variant

Applies when \( S = 1 \) \&\& \text{imm3} = 000 \&\& \text{imm2} = 00 \&\& \text{stype} = 11.

\[
\text{MOVS}(<c>)\{<q>\} <Rd>, <Rm>, \text{RRX}
\]

MOV, shift or rotate by value variant

Applies when \( S = 1 \) \&\& \((\text{imm3} = 000 \&\& \text{imm2} = 00 \&\& \text{stype} = 11)\).

\[
\text{MOVS}.W <Rd>, <Rm> \{, <\text{shift}> #<\text{amount}>\} // \text{Outside IT block, and <Rd>, <Rm>, <\text{shift}>, <\text{amount}> \text{ can be represented in T1 or T2}
\]

\[
\text{MOVS}<c>{<q>} <Rd>, <Rm> \{, <\text{shift}> #<\text{amount}>\}
\]

Decode for all variants of this encoding

\[
d = \text{UInt}(\text{Rd}); \ m = \text{UInt}(\text{Rm}); \ \text{setflags} = (S = '1');
\]

\[
(\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{imm3:imm2});
\]

\[
\text{if} \ d = 15 || m = 15 \text{ then UNPREDICTABLE}; \ // \text{Armv8-A removes UNPREDICTABLE for R13}
\]

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>of variant</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASRS (immediate)</td>
<td>T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)</td>
<td>( S = '1' &amp;&amp; \text{stype} = '10' )</td>
</tr>
<tr>
<td>ASRS (immediate)</td>
<td>T2</td>
<td>( \text{op} = '10' &amp;&amp; !\text{InITBlock()} )</td>
</tr>
<tr>
<td>ASR (immediate)</td>
<td>T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)</td>
<td>( S = '0' &amp;&amp; \text{stype} = '10' )</td>
</tr>
<tr>
<td>ASR (immediate)</td>
<td>T2</td>
<td>( \text{op} = '10' &amp;&amp; \text{InITBlock()} )</td>
</tr>
<tr>
<td>LSLS (immediate)</td>
<td>T3 (MOV, shift or rotate by value)</td>
<td>( S = '1' &amp;&amp; \text{imm3:Rd:imm2} != '000xxxx00' &amp;&amp; \text{stype} = '00' )</td>
</tr>
<tr>
<td>LSLS (immediate)</td>
<td>A1 (MOV, shift or rotate by value)</td>
<td>( S = '1' &amp;&amp; \text{imm5} != '000000' &amp;&amp; \text{stype} = '00' )</td>
</tr>
<tr>
<td>LSLS (immediate)</td>
<td>T2</td>
<td>( \text{op} = '00' &amp;&amp; \text{imm5} != '000000' &amp;&amp; !\text{InITBlock()} )</td>
</tr>
<tr>
<td>LSL (immediate)</td>
<td>T3 (MOV, shift or rotate by value)</td>
<td>( S = '0' &amp;&amp; \text{imm3:Rd:imm2} != '000xxxx00' &amp;&amp; \text{stype} = '00' )</td>
</tr>
<tr>
<td>LSL (immediate)</td>
<td>A1 (MOV, shift or rotate by value)</td>
<td>( S = '0' &amp;&amp; \text{imm5} != '000000' &amp;&amp; \text{stype} = '00' )</td>
</tr>
<tr>
<td>LSL (immediate)</td>
<td>T2</td>
<td>( \text{op} = '00' &amp;&amp; \text{imm5} != '000000' &amp;&amp; \text{InITBlock()} )</td>
</tr>
<tr>
<td>LSRS (immediate)</td>
<td>T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)</td>
<td>( S = '1' &amp;&amp; \text{stype} = '01' )</td>
</tr>
</tbody>
</table>
### Assembler symbols

<table>
<thead>
<tr>
<th>Alias</th>
<th>of variant</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSRS(IMM)</td>
<td>T2</td>
<td>$\text{op} = '01' &amp; &amp; \text{InITBlock()}$</td>
</tr>
<tr>
<td>LSR(IMM)</td>
<td>T3(MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)</td>
<td>$S = '0' &amp; &amp; \text{stype} = '01'$</td>
</tr>
<tr>
<td>LSR(IMM)</td>
<td>T2</td>
<td>$\text{op} = '01' &amp; &amp; \text{InITBlock}()$</td>
</tr>
<tr>
<td>RORS(IMM)</td>
<td>T3(MOVS, shift or rotate by value)</td>
<td>$S = '1' &amp; &amp; \text{imm3:Rd:imm2} != '000xxxx00' &amp; &amp; \text{stype} = '11'$</td>
</tr>
<tr>
<td>RORS(IMM)</td>
<td>A1(MOVS, shift or rotate by value)</td>
<td>$S = '1' &amp; &amp; \text{imm5} != '00000' &amp; &amp; \text{stype} = '11'$</td>
</tr>
<tr>
<td>ROR(IMM)</td>
<td>T3(MOV, shift or rotate by value)</td>
<td>$S = '0' &amp; &amp; \text{imm3:Rd:imm2} != '000xxxx00' &amp; &amp; \text{stype} = '11'$</td>
</tr>
<tr>
<td>ROR(IMM)</td>
<td>A1(MOV, shift or rotate by value)</td>
<td>$S = '0' &amp; &amp; \text{imm5} != '00000' &amp; &amp; \text{stype} = '11'$</td>
</tr>
<tr>
<td>RRXS</td>
<td>T3(MOVS, rotate right with extend)</td>
<td>$S = '1' &amp; &amp; \text{imm3} = '000' &amp; &amp; \text{imm2} = '00' &amp; &amp; \text{stype} = '11'$</td>
</tr>
<tr>
<td>RRXS</td>
<td>A1(MOVS, rotate right with extend)</td>
<td>$S = '1' &amp; &amp;\text{imm5} = '00000' &amp; &amp; \text{stype} = '11'$</td>
</tr>
<tr>
<td>RRX</td>
<td>T3(MOV, rotate right with extend)</td>
<td>$S = '0' &amp; &amp; \text{imm3} = '000' &amp; &amp; \text{imm2} = '00' &amp; &amp; \text{stype} = '11'$</td>
</tr>
<tr>
<td>RRX</td>
<td>A1(MOV, rotate right with extend)</td>
<td>$S = '0' &amp; &amp;\text{imm5} = '00000' &amp; &amp; \text{stype} = '11'$</td>
</tr>
</tbody>
</table>

See **Standard assembler syntax fields** on page F1-7228.

For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used:

- For the MOV variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-7133. Arm deprecates use of the instruction if $<Rn>$ is the PC.

- For the MOVS variant, the instruction performs an exception return, that restores PSTATE from SPSR_{current_mode}. Arm deprecates use of the instruction if $<Rn>$ is not the LR, or if the optional shift or RRX argument is specified.

For encoding T1: is the general-purpose destination register, encoded in the "D:Rd" field. If the PC is used:

- The instruction causes a branch to the address moved to the PC. This is a simple branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC* on page E1-7133.

- The instruction must either be outside an IT block or the last instruction of an IT block.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used. Arm deprecates use of the instruction if $<Rd>$ is the PC.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

For encoding A1 and T3: is the type of shift to be applied to the source register, encoded in the "stype" field. It can have the following values:

- **LSL** when stype = 00
- **LSR** when stype = 01
- **ASR** when stype = 10
- **ROR** when stype = 11
For encoding T2: is the type of shift to be applied to the source register, encoded in the "op" field. It can have the following values:

- **LSL** when \( \text{op} = 00 \)
- **LSR** when \( \text{op} = 01 \)
- **ASR** when \( \text{op} = 10 \)

For encoding A1: is the shift amount, in the range 0 to 31 (when \(<\text{shift}>\) = LSL), or 1 to 31 (when \(<\text{shift}>\) = ROR) or 1 to 32 (when \(<\text{shift}>\) = LSR or ASR), encoded in the "imm5" field as \(<\text{amount}>\) modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when \(<\text{shift}>\) = LSL or ROR) or 1 to 32 (when \(<\text{shift}>\) = LSR or ASR), encoded in the "imm5" field as \(<\text{amount}>\) modulo 32.

For encoding T3: is the shift amount, in the range 0 to 31 (when \(<\text{shift}>\) = LSL) or 1 to 31 (when \(<\text{shift}>\) = ROR), or 1 to 32 (when \(<\text{shift}>\) = LSR or ASR), encoded in the "imm3:imm2" field as \(<\text{amount}>\) modulo 32.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = shifted;
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.113  MOV, MOVS (register-shifted register)

Move (register-shifted register) copies a register-shifted register value to the destination register. It can optionally update the condition flags based on the value.

This instruction is used by the aliases ASRS (register), ASR (register), LSLS (register), LSL (register), LSRS (register), LSR (register), RORS (register), and ROR (register). See Alias conditions on page F5-7726 for details of when each alias is preferred.

Flag setting variant
Applies when $S == 1$.

MOVS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>

Not flag setting variant
Applies when $S == 0$.

MOV{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

d = UInt(Rd);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE;

Arithmetic shift right variant
Applies when op == 0100.

MOV{<c>}{<q>} <Rdm>, <Rdm>, ASR <Rs> // Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs> // Outside IT block

Logical shift left variant
Applies when op == 0010.

MOV{<c>}{<q>} <Rdm>, <Rdm>, LSL <Rs> // Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs> // Outside IT block

Logical shift right variant
Applies when op == 0011.

MOV{<c>}{<q>} <Rdm>, <Rdm>, LSR <Rs> // Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs> // Outside IT block
**Rotate right variant**

Applies when `op == 0111`.

\[
\text{MOV<c>{<q}> <Rdm>, <Rdm>, ROR <Rs> // Inside IT block} \\
\text{MOVS{<q}> <Rdm>, <Rdm>, ROR <Rs> // Outside IT block}
\]

**Decode for all variants of this encoding**

\[
\text{if (!op IN ('0010', '0011', '0100', '0111')) then SEE "Related encodings";}
\text{d = UInt(Rdm); m = UInt(Rdm); s = UInt(Rs);}
\text{setflags = !InITBlock(); shift_t = DecodeRegShift(op<2>:op<0>);}
\]

**T2**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 14 | 13 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|---|---|---|---|---|---|---|---|---|
| 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|     |     |     |     |     |     |     |     | |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

**Flag setting variant**

Applies when `S == 1`.

\[
\text{MOVS.W <Rd>, <Rm>, <shift> <Rs> // Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1} \\
\text{MOV{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>}
\]

**Not flag setting variant**

Applies when `S == 0`.

\[
\text{MOV{<c>}.W <Rd>, <Rm>, <shift> <Rs> // Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1} \\
\text{MOV{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>}
\]

**Decode for all variants of this encoding**

\[
d = \text{UInt(Rd)}; m = \text{UInt(Rm)}; s = \text{UInt(Rs)};
\text{setflags = (S == '1'); shift_t = DecodeRegShift(stype);}
\text{if d == 15 || m == 15 || s == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

**Notes for all encodings**

Related encodings: In encoding T1, for an `op` field value that is not described above, see *Data-processing (two low registers)* on page F3-7297.

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see *Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.*
### Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>of variant</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASRS (register)</td>
<td>A1 (flag setting)</td>
<td>$S == '1'$ &amp;&amp; stype == '10'</td>
</tr>
<tr>
<td>ASRS (register)</td>
<td>T1 (arithmetic shift right)</td>
<td>op == '0100' &amp;&amp; !InITBlock()</td>
</tr>
<tr>
<td>ASRS (register)</td>
<td>T2 (flag setting)</td>
<td>stype == '10' &amp;&amp; $S == '1'$</td>
</tr>
<tr>
<td>ASR (register)</td>
<td>A1 (not flag setting)</td>
<td>$S == '0'$ &amp;&amp; stype == '10'</td>
</tr>
<tr>
<td>ASR (register)</td>
<td>T1 (arithmetic shift right)</td>
<td>op == '0100' &amp;&amp; InITBlock()</td>
</tr>
<tr>
<td>ASR (register)</td>
<td>T2 (not flag setting)</td>
<td>stype == '10' &amp;&amp; $S == '0'$</td>
</tr>
<tr>
<td>LSLS (register)</td>
<td>A1 (flag setting)</td>
<td>$S == '1'$ &amp;&amp; stype == '00'</td>
</tr>
<tr>
<td>LSLS (register)</td>
<td>T1 (logical shift left)</td>
<td>op == '0010' &amp;&amp; !InITBlock()</td>
</tr>
<tr>
<td>LSLS (register)</td>
<td>T2 (flag setting)</td>
<td>stype == '00' &amp;&amp; $S == '1'$</td>
</tr>
<tr>
<td>LSL (register)</td>
<td>A1 (not flag setting)</td>
<td>$S == '0'$ &amp;&amp; stype == '00'</td>
</tr>
<tr>
<td>LSL (register)</td>
<td>T1 (logical shift left)</td>
<td>op == '0010' &amp;&amp; InITBlock()</td>
</tr>
<tr>
<td>LSL (register)</td>
<td>T2 (not flag setting)</td>
<td>stype == '00' &amp;&amp; $S == '0'$</td>
</tr>
<tr>
<td>LSRS (register)</td>
<td>A1 (flag setting)</td>
<td>$S == '1'$ &amp;&amp; stype == '01'</td>
</tr>
<tr>
<td>LSRS (register)</td>
<td>T1 (logical shift right)</td>
<td>op == '0011' &amp;&amp; !InITBlock()</td>
</tr>
<tr>
<td>LSRS (register)</td>
<td>T2 (not flag setting)</td>
<td>stype == '01' &amp;&amp; $S == '0'$</td>
</tr>
<tr>
<td>LSR (register)</td>
<td>A1 (not flag setting)</td>
<td>$S == '0'$ &amp;&amp; stype == '01'</td>
</tr>
<tr>
<td>LSR (register)</td>
<td>T1 (logical shift right)</td>
<td>op == '0011' &amp;&amp; InITBlock()</td>
</tr>
<tr>
<td>LSR (register)</td>
<td>T2 (not flag setting)</td>
<td>stype == '01' &amp;&amp; $S == '0'$</td>
</tr>
<tr>
<td>RORS (register)</td>
<td>A1 (flag setting)</td>
<td>$S == '1'$ &amp;&amp; stype == '11'</td>
</tr>
<tr>
<td>RORS (register)</td>
<td>T1 (rotate right)</td>
<td>op == '0111' &amp;&amp; !InITBlock()</td>
</tr>
<tr>
<td>RORS (register)</td>
<td>T2 (flag setting)</td>
<td>stype == '11' &amp;&amp; $S == '1'$</td>
</tr>
<tr>
<td>ROR (register)</td>
<td>A1 (not flag setting)</td>
<td>$S == '0'$ &amp;&amp; stype == '11'</td>
</tr>
<tr>
<td>ROR (register)</td>
<td>T1 (rotate right)</td>
<td>op == '0111' &amp;&amp; InITBlock()</td>
</tr>
<tr>
<td>ROR (register)</td>
<td>T2 (not flag setting)</td>
<td>stype == '11' &amp;&amp; $S == '0'$</td>
</tr>
</tbody>
</table>

### Assembler symbols

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rdm>` Is the general-purpose source register and the destination register, encoded in the "Rdm" field.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<rm>` Is the general-purpose source register, encoded in the "Rm" field.
<shift> Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
- LSL when stype = 00
- LSR when stype = 01
- ASR when stype = 10
- ROR when stype = 11

<Rs> Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (result, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- **The execution time of this instruction is independent of:**
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- **The response of this instruction to asynchronous exceptions does not vary based on:**
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
### F5.1.114 MOVT

Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

#### A1

Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

#### A1 variant

MOVT{<c>}{<q>} <Rd>, #<imm16>

**Decode for this encoding**

\[
\begin{align*}
\text{d} &= \text{UInt}(\text{Rd}); \\
\text{imm16} &= \text{imm4:imm12}; \\
\text{if } d &= 15 \text{ then UNPREDICTABLE;}
\end{align*}
\]

#### T1

Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

#### T1 variant

MOVT{<c>}{<q>} <Rd>, #<imm16>

**Decode for this encoding**

\[
\begin{align*}
\text{d} &= \text{UInt}(\text{Rd}); \\
\text{imm16} &= \text{imm4:i:imm3:imm8}; \\
\text{if } d &= 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\end{align*}
\]

#### Notes for all encodings

For more information about the CONSTRUANED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

#### Assembler symbols

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<imm16>` For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field.
  - For encoding T1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:i:imm3:imm8" field.
Operation for all encodings

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<31:16> = imm16;
    // R[d]<15:0> unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.115  MRC

Move to general-purpose register from System register. This instruction copies the value of a System register to a
general-purpose register.

The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For
more information see About the AArch32 System register interface on page E1-7158and General behavior of System
registers on page G8-9322.

In an implementation that includes EL2, MRC accesses to system control registers can be trapped to Hyp mode,
meaning that an attempt to execute an MRC instruction in a Non-secure mode other than Hyp mode, that would be
permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see EL2
configurable controls on page G1-9010.

Because of the range of possible traps to Hyp mode, the MRC pseudocode does not show these possible traps.

A1

\[
\begin{array}{ccccccccccccccccccc}
\text{cond} & 31 & 28 & 27 & 26 & 25 & 24 & 23 & 21 & 20 & 19 & 16 & 15 & 12 & 11 & 9 & 8 & 7 & 5 & 4 & 3 & 0 \\
1=1111 & 1 & 1 & 1 & 0 & \text{opc1} & 1 & \text{CRn} & \text{Rt} & 1 & 1 & 1 & \text{opc2} & 1 & \text{CRm} \\
\end{array}
\]

\text{A1 variant}

\text{MRC(}<c>){<q>}> <\text{coproc}>, {#}<\text{opc1}>, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>{{#}<\text{opc2}}>}

\text{Decode for this encoding}

\[
t = \text{UInt}(\text{Rt}); \quad \text{cp} = \text{if coproc}<0> = '0' \text{ then } 14 \text{ else } 15;
\]/
// Armv8-A removes UNPREDICTABLE for R13

T1

\[
\begin{array}{ccccccccccccccccccc}
\text{cond} & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 5 & 4 & 3 & 0 & 15 & 12 & 11 & 9 & 8 & 7 & 5 & 4 & 3 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & \text{opc1} & 1 & \text{CRn} & \text{Rt} & 1 & 1 & 1 & \text{opc2} & 1 & \text{CRm} \\
\end{array}
\]

\text{T1 variant}

\text{MRC(}<c>){<q>}> <\text{coproc}>, {#}<\text{opc1}>, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>{{#}<\text{opc2}}>}

\text{Decode for this encoding}

\[
t = \text{UInt}(\text{Rt}); \quad \text{cp} = \text{if coproc}<0> = '0' \text{ then } 14 \text{ else } 15;
\]/
// Armv8-A removes UNPREDICTABLE for R13

\text{Assembler symbols}

\text{<c> See Standard assembler syntax fields on page F1-7228.}

\text{<q> See Standard assembler syntax fields on page F1-7228.}
<coproc> Is the System register encoding space, encoded in the "coproc<0>" field. It can have the following values:

- p14 when coproc<0> = 0
- p15 when coproc<0> = 1

<opc1> Is the opc1 parameter within the System register encoding space, in the range 0 to 7, encoded in the "opc1" field.

<Rt> Is the general-purpose register to be transferred or APSR_nzcv (encoded as 0b1111), encoded in the "Rt" field. If APSR_nzcv is used, bits [31:28] of the transferred value are written to the PSTATE condition flags.

<CRn> Is the CRn parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRn" field.

<CRm> Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.

<opc2> Is the opc2 parameter within the System register encoding space, in the range 0 to 7, encoded in the "opc2" field.

The possible values of \{<coproc>, <opc1>, <CRn>, <CRm>, <opc2>\} encode the entire System register and System instruction encoding space. Not all of this space is allocated, and the System register and System instruction descriptions identify the allocated encodings.

**Operation for all encodings**

if ConditionPassed() then
   EncodingSpecificOperations();

   if t != 15 || AArch32.SysRegReadWriteAPSR(cp, ThisInstr()) then
      AArch32.SysRegReadWrite(cp, ThisInstr(), t);
   else
      UNPREDICTABLE;
F5.1.116   MRRC

Move to two general-purpose registers from System register. This instruction copies the value of a System register to two general-purpose registers.

The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see About the AArch32 System register interface on page E1-7158 and General behavior of System registers on page G8-9322.

In an implementation that includes EL2, MRRC accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MRRC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see EL2 configurable controls on page G1-9010.

Because of the range of possible traps to Hyp mode, the MRRC pseudocode does not show these possible traps.

A1

```
\[31\quad 28\|27\quad 26\quad 25\quad 24\|23\quad 22\quad 21\quad 20\|19\quad 16\|15\quad 12\|11\quad 9\|8\|7\|4\|3\|0\]
\begin{array}{|c|c|c|c|c|}
\hline
\text{cond} & 1 & 1 & 0 & 0 & 1 & 0 & 1 & \text{Rt2} & \text{Rt} & 1 & 1 & 1 & \text{opc1} & \text{CRm} \\
\hline
\end{array}
```

A1 variant

MRRC{<c>}{<q>} <coproc>, {@opc1}, <Rt>, <Rt2>, <CRm>

Decode for this encoding

\[t = \text{UInt}(\text{Rt}); \quad t2 = \text{UInt}(\text{Rt2}); \quad cp = \text{if coproc}<0,3:1> == '0' \text{ then } 14 \text{ else } 15; \]
\[\text{if } t == 15 || t2 == 15 || t == t2 \text{ then UNPREDICTABLE; } // \text{ Armv8-A removes UNPREDICTABLE for } R13\]

CONstrained UNPREDICTABLE behavior

If \( t == t2 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

```
\[15\quad 14\quad 13\quad 12\|11\quad 10\quad 9\|8\|7\|6\|5\|4\|3\|0\]
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & \text{Rt2} & \text{Rt} & 1 & 1 & 1 & \text{opc1} & \text{CRm} \\
\hline
\end{array}
```

T1 variant

MRRC{<c>}{<q>} <coproc>, {@opc1}, <Rt>, <Rt2>, <CRm>
**Decode for this encoding**

\[ t = \text{UInt}(Rt); \quad t2 = \text{UInt}(Rt2); \quad cp = \text{if coproc<0> == '0' then 14 else 15}; \]
\[ \text{if } t == 15 || t2 == 15 || t == t2 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13} \]

**CONSTRAINED UNPREDICTABLE behavior**

If \( t == t2 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<coproc>` Is the System register encoding space, encoded in the "coproc<0>" field. It can have the following values:
  - p14 when coproc<0> = 0
  - p15 when coproc<0> = 1
- `<opc1>` Is the opc1 parameter within the System register encoding space, in the range 0 to 15, encoded in the "opc1" field.
- `<Rt>` Is the first general-purpose register that is transferred into, encoded in the "Rt" field.
- `<Rt2>` Is the second general-purpose register that is transferred into, encoded in the "Rt2" field.
- `<CRm>` Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.

The possible values of \{ <coproc>, <opc1>, <CRm> \} encode the entire System register encoding space. Not all of this space is allocated, and the System register descriptions identify the allocated encodings.

For the permitted uses of these instructions, as described in this manual, <t2> transfers bits[63:32] of the selected System register, while <Rt> transfers bits[31:0].

**Operation for all encodings**

\[
\text{if ConditionPassed() then}
\text{EncodingSpecificOperations();}
\text{AArch32.SysRegRead64(cp, ThisInstr(), t, t2);}\]
F5.1.117 MRS

Move Special register to general-purpose register moves the value of the The Application Program Status Register, APSR on page E1-7135, CPSR, or SPSR_{<current_mode>} into a general-purpose register.

Arm recommends the APSR form when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see The Application Program Status Register, APSR on page E1-7135.

An MRS that accesses the SPSR is UNPREDICTABLE if executed in User mode or System mode.

An MRS that is executed in User mode and accesses the CPSR returns an UNKNOWN value for the CPSR. {E, A, I, F, M} fields.

A1

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| l=111 | 0 | 0 | 0 | 1 | 0 | R | 0 | 0 | (1)(1)(1)(1) | Rd | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| cond |

A1 variant

MRS{<c>}{<q>} <Rd>, <spec_reg>

Decode for this encoding

d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | R | (1)(1)(1)(1) | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

T1 variant

MRS{<c>}{<q>} <Rd>, <spec_reg>

Decode for this encoding

d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<c>     See Standard assembler syntax fields on page F1-7228.

<q>     See Standard assembler syntax fields on page F1-7228.

<Rd>    Is the general-purpose destination register, encoded in the "Rd" field.
<spec_reg> Is the special register to be accessed, encoded in the "R" field. It can have the following values:

- CPSR|APSR when R = 0
- SPSR when R = 1

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations();
if read_spsr then
    if PSTATE.M IN {M32_User, M32_System} then
        UNPREDICTABLE;
    else
        R[d] = SPSR[];
else
    // CPSR has same bit assignments as SPSR, but with the IT, J, SS, IL, and T bits masked out.
    bits(32) mask = '11110000 11101111 00000011 11011111';
    psr_val = GetPSRFromPSTATE(AArch32_NonDebugState, 32) AND mask;
    if PSTATE.EL == EL0 then
        // If accessed from User mode return UNKNOWN values for E, A, I, F bits, bits<9:6>,
        // and for the M field, bits<4:0>
        psr_val<22> = bits(1) UNKNOWN;
        psr_val<9:6> = bits(4) UNKNOWN;
        psr_val<4:0> = bits(5) UNKNOWN;
        R[d] = psr_val;
```

**CONSTRAINED UNPREDICTABLE behavior**

If PSTATE.M IN (M32_User, M32_System) && read_spsr, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
F5.1.118   MRS (Banked register)

Move to Register from Banked or Special register moves the value from the Banked general-purpose register or SPSR of the specified mode, or the value of ELR_hyp on page G1-8920, to a general-purpose register.

\textit{MRS} (Banked register) is UNPREDICTABLE if executed in User mode.

When EL3 is using AArch64, if an \textit{MRS} (Banked register) instruction that is executed in a Secure EL1 mode would access SPSR\textsubscript{mon}, SP\textsubscript{mon}, or LR\textsubscript{mon}, it is trapped to EL3.

The effect of using an \textit{MRS} (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see Usage restrictions on the banked register transfer instructions on page F5-8161.

A1

\begin{verbatim}
|31|28|26|25|24|23|22|21|20|19|16|15|14|13|12|11|10|9|8|7|6|5|4|3|2|1|0|
|   1111  | 0| 0| 0| 1| 0| R| 0| 0| M1| Rd| 0| 0| 1| M| 0| 0| 0| 0| 0| 0| 0| 0|

cond
\end{verbatim}

\textbf{A1 variant}

\textit{MRS}\{<c>\}{<q>} <Rd>, <banked_reg>

\textbf{Decode for this encoding}

\begin{verbatim}
d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE;
SYSm = M:M1;
\end{verbatim}

T1

\begin{verbatim}
|15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 0|15|14|13|12|11| 8| 7| 6| 5| 4| 3| 2| 1|0|
|1| 1| 1| 1| 0| 0| 1| 1| 1| 1| R| M1| 1| 0| 0| Rd| 0|0|1| M| 0|0|0|0|
\end{verbatim}

\textbf{T1 variant}

\textit{MRS}\{<c>\}{<q>} <Rd>, <banked_reg>

\textbf{Decode for this encoding}

\begin{verbatim}
d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
SYSm = M:M1;
\end{verbatim}

\textbf{Notes for all encodings}

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

\textbf{Assembler symbols}

\begin{verbatim}
<c> See Standard assembler syntax fields on page F1-7228.
<q> See Standard assembler syntax fields on page F1-7228.
<Rd> Is the general-purpose destination register, encoded in the "Rd" field.
\end{verbatim}
<banked_reg> Is the name of the banked register to be transferred to or from, encoded in the "R:M:M1" field. It can have the following values:

- R8_usr when R = 0, M = 0, M1 = 0000
- R9_usr when R = 0, M = 0, M1 = 0001
- R10_usr when R = 0, M = 0, M1 = 0010
- R11_usr when R = 0, M = 0, M1 = 0011
- R12_usr when R = 0, M = 0, M1 = 0100
- SP_usr when R = 0, M = 0, M1 = 0101
- LR_usr when R = 0, M = 0, M1 = 0110
- R8_fiq when R = 0, M = 0, M1 = 1000
- R9_fiq when R = 0, M = 0, M1 = 1001
- R10_fiq when R = 0, M = 0, M1 = 1010
- R11_fiq when R = 0, M = 0, M1 = 1011
- R12_fiq when R = 0, M = 0, M1 = 1100
- SP_fiq when R = 0, M = 0, M1 = 1101
- LR_fiq when R = 0, M = 0, M1 = 1110
- LR_irq when R = 0, M = 1, M1 = 0000
- SP_irq when R = 0, M = 1, M1 = 0001
- LR_svc when R = 0, M = 1, M1 = 0010
- SP_svc when R = 0, M = 1, M1 = 0011
- LR_abt when R = 0, M = 1, M1 = 0100
- SP_abt when R = 0, M = 1, M1 = 0101
- LR_und when R = 0, M = 1, M1 = 0110
- SP_und when R = 0, M = 1, M1 = 0111
- LR_mon when R = 0, M = 1, M1 = 1100
- SP_mon when R = 0, M = 1, M1 = 1101
- ELR_hyp when R = 0, M = 1, M1 = 1110
- SP_hyp when R = 0, M = 1, M1 = 1111
- SPSR_fiq when R = 1, M = 0, M1 = 1110
- SPSR_irq when R = 1, M = 0, M1 = 1111
- SPSR_svc when R = 1, M = 0, M1 = 0000
- SPSR_abt when R = 1, M = 0, M1 = 0010
- SPSR_und when R = 1, M = 0, M1 = 0110
- SPSR_mon when R = 1, M = 0, M1 = 1100
- SPSR_hyp when R = 1, M = 0, M1 = 1110

The following encodings are UNPREDICTABLE:

- R = 0, M = 0, M1 = 0111.
- R = 0, M = 0, M1 = 1111.
- R = 0, M = 1, M1 = 10xx.
- R = 1, M = 0, M1 = 0xxx.
- R = 1, M = 0, M1 = 10xx.
- R = 1, M = 0, M1 = 110x.
- R = 1, M = 0, M1 = 1111.
- R = 1, M = 1, M1 = 0001.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
else if PSTATE.EL == EL0 then
    UNPREDICTABLE;
    mode = PSTATE.M;
    if read_spsr then
        SPSRaccessValid(SYSm, mode); // Check for UNPREDICTABLE cases
    case SYSm of
        when '01110' R[d] = SPSR_fiq<31:0>;
        when '10000' R[d] = SPSR_irq<31:0>;
        when '10010' R[d] = SPSR_svc<31:0>;
        when '10100' R[d] = SPSR_abt<31:0>;
        when '10110' R[d] = SPSR_und<31:0>;
        when '11100' if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
            R[d] = SPSR_mon;
        when '11110' R[d] = SPSR_hyp<31:0>;
    else
        integer m;
        BankedRegisterAccessValid(SYSm, mode); // Check for UNPREDICTABLE cases
        case SYSm of
            when '00xxx' // Access the User mode registers
                m = UInt(SYSm<2:0>) + 8;
                R[d] = Rmode[m,M32_User];
            when '01xxx' // Access the FIQ mode registers
                m = UInt(SYSm<2:0>) + 8;
                R[d] = Rmode[m,M32_FIQ];
            when '1000x' // Access the IRQ mode registers
                m = 14 - UInt(SYSm<0>); // LR when SYSm<0> == 0, otherwise SP
                R[d] = Rmode[m,M32_IRQ];
            when '1001x' // Access the Supervisor mode registers
                m = 14 - UInt(SYSm<0>); // LR when SYSm<0> == 0, otherwise SP
                R[d] = Rmode[m,M32_Svc];
            when '1010x' // Access the Abort mode registers
                m = 14 - UInt(SYSm<0>); // LR when SYSm<0> == 0, otherwise SP
                R[d] = Rmode[m,M32_Abort];
            when '1011x' // Access the Undefined mode registers
                m = 14 - UInt(SYSm<0>); // LR when SYSm<0> == 0, otherwise SP
                R[d] = Rmode[m,M32_Undef];
            when '1110x' // Access Monitor registers
                if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                m = 14 - UInt(SYSm<0>); // LR when SYSm<0> == 0, otherwise SP
                R[d] = Rmode[m,M32_Monitor];
            when '11110' // Access ELR_hyp register
                R[d] = ELR_hyp;
            when '11111' // Access SP_hyp register
                R[d] = Rmode[13,M32_Hyp];
CONSTRUANED UNPREDICTABLE behavior
If PSTATE.EL = EL0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
F5.1.119  MSR (Banked register)

Move to Banked or Special register from general-purpose register moves the value of a general-purpose register to the Banked general-purpose register or SPSR of the specified mode, or to ELR_hyp on page G1-8920.

MSR (Banked register) is UNPREDICTABLE if executed in User mode.

When EL3 is using AArch64, if an MSR (Banked register) instruction that is executed in a Secure EL1 mode would access SPSR_mon, SP_mon, or LR_mon, it is trapped to EL3.

The effect of using an MSR (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see Usage restrictions on the banked register transfer instructions on page F5-8161.

A1

```assembly
MSR{<c>}{<q>} <banked_reg>, <Rn>
```

Decode for this encoding

\[
n = \text{UInt}(Rn);\  \text{write_spsr} = (R == '1');\n\]

if \( n == 15 \) then UNPREDICTABLE;

\[\text{SYSm} = M:M1;\]

T1

```assembly
MSR{<c>}{<q>} <banked_reg>, <Rn>
```

Decode for this encoding

\[
n = \text{UInt}(Rn);\  \text{write_spsr} = (R == '1');\n\]

if \( n == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

\[\text{SYSm} = M:M1;\]

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\)  See Standard assembler syntax fields on page F1-7228.

\(<q>\)  See Standard assembler syntax fields on page F1-7228.
Is the name of the banked register to be transferred to or from, encoded in the "R:M:M1" field. It can have the following values:

- **R8_usr** when \( R = 0, M = 0, M1 = 0000 \)
- **R9_usr** when \( R = 0, M = 0, M1 = 0001 \)
- **R10_usr** when \( R = 0, M = 0, M1 = 0010 \)
- **R11_usr** when \( R = 0, M = 0, M1 = 0011 \)
- **R12_usr** when \( R = 0, M = 0, M1 = 0100 \)
- **SP_usr** when \( R = 0, M = 0, M1 = 0101 \)
- **LR_usr** when \( R = 0, M = 0, M1 = 0110 \)
- **R8_fiq** when \( R = 0, M = 0, M1 = 1000 \)
- **R9_fiq** when \( R = 0, M = 0, M1 = 1001 \)
- **R10_fiq** when \( R = 0, M = 0, M1 = 1010 \)
- **R11_fiq** when \( R = 0, M = 0, M1 = 1011 \)
- **R12_fiq** when \( R = 0, M = 0, M1 = 1100 \)
- **SP_fiq** when \( R = 0, M = 0, M1 = 1101 \)
- **LR_fiq** when \( R = 0, M = 0, M1 = 1110 \)
- **LR_irq** when \( R = 0, M = 1, M1 = 0000 \)
- **SP_irq** when \( R = 0, M = 1, M1 = 0001 \)
- **LR_svc** when \( R = 0, M = 1, M1 = 0010 \)
- **SP_svc** when \( R = 0, M = 1, M1 = 0011 \)
- **LR_abt** when \( R = 0, M = 1, M1 = 0100 \)
- **SP_abt** when \( R = 0, M = 1, M1 = 0101 \)
- **LR_und** when \( R = 0, M = 1, M1 = 0110 \)
- **SP_und** when \( R = 0, M = 1, M1 = 0111 \)
- **LR_mon** when \( R = 0, M = 1, M1 = 1100 \)
- **SP_mon** when \( R = 0, M = 1, M1 = 1101 \)
- **ELR_hyp** when \( R = 0, M = 1, M1 = 1110 \)
- **SP_hyp** when \( R = 0, M = 1, M1 = 1111 \)
- **SPSR_fiq** when \( R = 1, M = 0, M1 = 1110 \)
- **SPSR_irq** when \( R = 1, M = 0, M1 = 1111 \)
- **SPSR_svc** when \( R = 1, M = 0, M1 = 0000 \)
- **SPSR_abt** when \( R = 1, M = 0, M1 = 0001 \)
- **SPSR_und** when \( R = 1, M = 0, M1 = 0010 \)
- **SPSR_mon** when \( R = 1, M = 0, M1 = 1100 \)
- **SPSR_hyp** when \( R = 1, M = 0, M1 = 1110 \)

The following encodings are UNPREDICTABLE:

- \( R = 0, M = 0, M1 = 0111 \)
- \( R = 0, M = 0, M1 = 1111 \)
- \( R = 0, M = 1, M1 = 10xx \)
- \( R = 1, M = 0, M1 = 0xxx \)
- \( R = 1, M = 0, M1 = 10xx \)
- \( R = 1, M = 0, M1 = 110x \)
- \( R = 1, M = 0, M1 = 1111 \)
- \( R = 1, M = 1, M1 = 0001 \).
\begin{itemize}
  \item \( R = 1, M = 1, M1 = 0011 \).
  \item \( R = 1, M = 1, M1 = 0101 \).
  \item \( R = 1, M = 1, M1 = 0111 \).
  \item \( R = 1, M = 1, M1 = 10xx \).
  \item \( R = 1, M = 1, M1 = 1101 \).
  \item \( R = 1, M = 1, M1 = 1111 \).
\end{itemize}

\texttt{<Rn>}

Is the general-purpose source register, encoded in the "Rn" field.

\section*{Operation for all encodings}

\begin{verbatim}
if ConditionPassed() then
  EncodingSpecificOperations();
if PSTATE.EL == EL0 then
  UNPREDICTABLE;
else
  mode = PSTATE.M;
  if write_spsr then
    SPSRaccessValid(SYSm, mode); // Check for UNPREDICTABLE cases
    case SYSm of
      when '01110'  SPSR_fiq<31:0> = R[n];
      when '10000'  SPSR_irq<31:0> = R[n];
      when '10010'  SPSR_svc<31:0> = R[n];
      when '10100'  SPSR_abt<31:0> = R[n];
      when '10110'  SPSR_und<31:0> = R[n];
      when '11100'
        if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
        SPSR_mon<31:0> = R[n];
      when '11110'  SPSR_hyp<31:0> = R[n];
    else
      integer m;
      BankedRegisterAccessValid(SYSm, mode); // Check for UNPREDICTABLE cases
      case SYSm of
        when '00xxx'                       // Access the User mode registers
          m = UInt(SYSm<2:0>) + 8;
          Rmode[m,M32_User] = R[n];
        when '01xxx'                       // Access the FIQ mode registers
          m = UInt(SYSm<2:0>) + 8;
          Rmode[m,M32_FIQ] = R[n];
        when '1000x'                       // Access the IRQ mode registers
          m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
          Rmode[m,M32_IRQ] = R[n];
        when '1001x'                       // Access the Supervisor mode registers
          m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
          Rmode[m,M32_Svc] = R[n];
        when '1010x'                       // Access the Abort mode registers
          m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
          Rmode[m,M32_Abort] = R[n];
        when '1011x'                       // Access the Undefined mode registers
          m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
          Rmode[m,M32_Undef] = R[n];
        when '1110x'                       // Access Monitor registers
          if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
          m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
          Rmode[m,M32_Monitor] = R[n];
        when '11110'  ELR_hyp = R[n];
        when '11111'  Rmode[13,M32_Hyp] = R[n];
    \end{verbatim}
CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL0, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
F5.1.120  MSR (immediate)

Move immediate value to Special register moves selected bits of an immediate value to the corresponding bits in the Application Program Status Register, APSR on page E1-7135, CPSR, or SPSR_<current_mode>.

Because of the Do-Not-Modify nature of its reserved bits, the immediate form of MSR is normally only useful at the Application level for writing to APSR_nzcvq (CPSR_f).

If an MSR (immediate) moves selected bits of an immediate value to the CPSR, the PE checks whether the value being written to PSTATE.M is legal. See Illegal changes to PSTATE.M on page G1-8925.

An MSR (immediate) executed in User mode:

- Is CONSTRAINED UNPREDICTABLE if it attempts to update the SPSR.
- Otherwise, does not update any CPSR field that is accessible only at EL1 or higher.

An MSR (immediate) executed in System mode is CONSTRAINED UNPREDICTABLE if it attempts to update the SPSR.

The CPSR.E bit is writable from any mode using an MSR instruction. Arm deprecates using this to change its value.

A1

<table>
<thead>
<tr>
<th>31 28 27 26 25 24 23 22 21 20 19 16 15 13 12 11</th>
<th>0 0 1 1</th>
<th>mask</th>
<th>1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=1111</td>
<td>0 O R 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td>imm12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 variant

Applies when !(R == 0 & mask == 0000).

MSR<spec_reg>, #<imm>

Decode for this encoding

\[
\text{if mask} = '0000' \&\& R = '0' \text{ then SEE "Related encodings";}
\]

\[
\text{imm32} = \text{A32ExpandImm}(\text{imm12}); \quad \text{write_spsr} = (R = '1');
\]

\[
\text{if mask} = '0000' \text{ then UNPREDICTABLE;}
\]

CONSTRAINED UNPREDICTABLE behavior

If mask == '0000' & R == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Related encodings: Move Special Register and Hints (immediate) on page F4-7389.

Assembler symbols

<>

See Standard assembler syntax fields on page F1-7228.

<q>

See Standard assembler syntax fields on page F1-7228.

<spec_reg>

Is one of:

- APSR_bbits.
• CPSR_<fields>.
• SPSR_<fields>.

For CPSR and SPSR, <fields> is a sequence of one or more of the following:
- c mask<0> = '1' to enable writing of bits<7:0> of the destination PSR.
- x mask<1> = '1' to enable writing of bits<15:8> of the destination PSR.
- s mask<2> = '1' to enable writing of bits<23:16> of the destination PSR.
- f mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

For APSR, <bits> is one of nzcvq, g, or nzcvqg. These map to the following CPSR_<fields> values:
- APSR_nzcvq is the same as CPSR_f (mask== '1000').
- APSR_g is the same as CPSR_s (mask == '0100').
- APSR_nzcvqg is the same as CPSR_fs (mask == '1100').

Arm recommends the APSR_<bits> forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see The Application Program Status Register, APSR on page E1-7135.

<imm> Is an immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
if write_spsr then
    if PSTATE.M IN {M32_User, M32_System} then
        UNPREDICTABLE;
    else
        SPSRWriteByInstr(imm32, mask);
else
    // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
    CPSRWriteByInstr(imm32, mask);

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.M IN {M32_User, M32_System} && write_spsr, then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as NOP.
F5.1.121  MSR (register)

Move general-purpose register to Special register moves selected bits of a general-purpose register to the The Application Program Status Register, APSR on page E1-7135, CPSR or SPSR_<current_mode>.

Because of the Do-Not-Modify nature of its reserved bits, a read-modify-write sequence is normally required when the MSR instruction is being used at Application level and its destination is not APSR_nzcvq (CPSR_f).

If an MSR (register) moves selected bits of an immediate value to the CPSR, the PE checks whether the value being written to PSTATE.M is legal. See Illegal changes to PSTATE.M on page G1-8925.

An MSR (register) executed in User mode:

- Is UNPREDICTABLE if it attempts to update the SPSR.
- Otherwise, does not update any CPSR field that is accessible only at EL1 or higher.

An MSR (register) executed in System mode is UNPREDICTABLE if it attempts to update the SPSR.

The CPSR.E bit is writable from any mode using an MSR instruction. Arm deprecates using this to change its value.

A1

```
|31 28|26 25 24|23 22 21 20|19 16|15 14 13 12|11 10 |9 8 |7 6 5 4 |3 0 |
|---|---|---|---|---|---|---|---|---|---|
|1=1111|0 0 0 |1 0 |R |1 0 |mask |(1) |(1) |(1) |(0)|0 |0 |0 |0 |0 |Rn |
```

cond

A1 variant

MSR{<c>}{<q>} <spec_reg>, <Rn>

Decode for this encoding

```
n = UInt(Rn);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If mask == '0000', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

T1

```
|15 14 13 12|11 10 |9 8 |7 6 5 4 |3 0 |
|---|---|---|---|---|---|
|1 1 1 1 |0 0 1 0 |R |1 0 |
mask |0 |0 |0 |0 |0 |0 |
```

T1 variant

MSR{<c>}{<q>} <spec_reg>, <Rn>

Decode for this encoding

```
n = UInt(Rn);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```
CONSTRAINED UNPREDICTABLE behavior

If mask == '0000', then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as NOP.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

<spec_reg>
Is one of:
- APSR_<bits>.
- CPSR_<fields>.
- SPSR_<fields>.

For CPSR and SPSR, <fields> is a sequence of one or more of the following:
- c mask<0> = '1' to enable writing of bits<7:0> of the destination PSR.
- x mask<1> = '1' to enable writing of bits<15:8> of the destination PSR.
- s mask<2> = '1' to enable writing of bits<23:16> of the destination PSR.
- f mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

For APSR, <bits> is one of nzcvq, g, or nzcvqg. These map to the following CPSR_<fields> values:
- APSR_nzcvq is the same as CPSR_f (mask == '1000').
- APSR_g is the same as CPSR_s (mask == '0100').
- APSR_nzcvqg is the same as CPSR_fs (mask == '1100').

Arm recommends the APSR_<bits> forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see The Application Program Status Register, APSR on page E1-7135.

<Rn>
Is the general-purpose source register, encoded in the "Rn" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
if write_spsr then
    if PSTATE.M IN {M32_User,M32_System} then
        UNPREDICTABLE;
    else
        SPSRWriteByInstr(R[n], mask);
else
    // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
    CPSRWriteByInstr(R[n], mask);

CONSTRAINED UNPREDICTABLE behavior

If write_spsr & PSTATE.M IN {M32_User,M32_System}, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
• The instruction executes as NOP.
F5.1.122 MUL, MULS

Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

Optionally, it can update the condition flags based on the result. In the T32 instruction set, this option is limited to only a few forms of the instruction. Use of this option adversely affects performance on many implementations.

A1

Flag setting variant

Applies when $S = 1$.

MULS{<q>} <Rd>, <Rn>{, <Rm>}

Not flag setting variant

Applies when $S = 0$.

MUL{<q>} <Rd>, <Rn>{, <Rm>}

Decode for all variants of this encoding

\[
\begin{align*}
    d &= \text{UInt}(Rd); \\
    n &= \text{UInt}(Rn); \\
    m &= \text{UInt}(Rm); \\
    \text{setflags} &= (S == '1'); \\
    \text{if } d == 15 \text{ || } n == 15 \text{ || } m == 15 \text{ then UNPREDICTABLE;} \\
\end{align*}
\]

T1

T1 variant

MUL{<q>} <Rdm>, <Rn>{, <Rdm>} // Inside IT block
MULS{<q>} <Rdm>, <Rn>{, <Rdm>} // Outside IT block

Decode for this encoding

\[
\begin{align*}
    d &= \text{UInt}(Rdm); \\
    n &= \text{UInt}(Rn); \\
    m &= \text{UInt}(Rdm); \\
    \text{setflags} &= !\text{InITBlock}(); \\
\end{align*}
\]

T2

T2 variant

MUL{<q>W} <Rd>, <Rn>{, <Rm>} // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
MUL{<q>} <Rd>, <Rn>{, <Rm>}

\[
\begin{align*}
\end{align*}
\]
**Decode for this encoding**

\[
\begin{align*}
d &= \text{UInt}(Rd); & n &= \text{UInt}(Rn); & m &= \text{UInt}(Rm); & \text{setflags} &= \text{FALSE}; \\
\text{if } d &= 15 \text{ || } n &= 15 \text{ || } m &= 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\end{align*}
\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

\(<c>\) \(\text{See Standard assembler syntax fields on page F1-7228.}\)

\(<q>\) \(\text{See Standard assembler syntax fields on page F1-7228.}\)

\(<Rdm>\) \(\text{Is the second general-purpose source register holding the multiplier and the destination register, encoded in the "Rdm" field.}\)

\(<Rd>\) \(\text{Is the general-purpose destination register, encoded in the "Rd" field.}\)

\(<Rn>\) \(\text{Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.}\)

\(<Rm>\) \(\text{Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. If omitted, <Rd> is used.}\)

**Operation for all encodings**

if \(\text{ConditionPassed()}\) then

EncodingSpecificOperations();

\(\text{operand1} = \text{SInt}(R[n]);\) \(\text{// operand1 = UInt(R[n]) produces the same final results}\)

\(\text{operand2} = \text{SInt}(R[m]);\) \(\text{// operand2 = UInt(R[m]) produces the same final results}\)

\(\text{result} = \text{operand1} \times \text{operand2;}\)

\(\text{R[d]} = \text{result}<31:0>;\)

if \(\text{setflags}\) then

\(\text{PSTATE.N} = \text{result}<31>;\)

\(\text{PSTATE.Z} = \text{IsZeroBit(result}<31:0>);\)

\(\text{// PSTATE.C, PSTATE.V unchanged}\)

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.123 MVN, MVNS (immediate)

Bitwise NOT (immediate) writes the bitwise inverse of an immediate value to the destination register.

If the destination register is not the PC, the MVNS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The MVN variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- The MVNS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

\[
\begin{array}{cccccccccccccccc}
\hline
l=1111 & 0 & 0 & 1 & 1 & 1 & S & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 | \\
cond
\end{array}
\]

**MVN variant**

Applies when S == 0.

\[
MVN\{<c>\}{<q>} \quad <Rd>, \quad \#<const>
\]

**MVNS variant**

Applies when S == 1.

\[
MVNS\{<c>\}{<q>} \quad <Rd>, \quad \#<const>
\]

**Decode for all variants of this encoding**

\[
d = \text{UInt}(Rd); \quad \text{setflags} = (S == '1'); \\
(iimm32, carry) = \text{A32ExpandImm_C}(iimm8, \text{PSTATE}.C);
\]

T1

\[
\begin{array}{cccccccccccccccc}
|15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 | \\
\hline
1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & S & 1 & 1 & 1 & 0 | \\
imm3 & Rd & imm8
\end{array}
\]

**MVN variant**

Applies when S == 0.

\[
MVN\{<c>\}{<q>} \quad <Rd>, \quad \#<const>
\]

**MVNS variant**

Applies when S == 1.
MVNS{<c>}{<q>} <Rd>, #<const>

**Decode for all variants of this encoding**

\[
d = \text{UInt}(Rd); \quad \text{setflags} = (S == '1');
\]

\[
(\text{imm32}, \text{carry}) = \text{T32ExpandImm}_{C}(i:imm3:imm8, \text{PSTATE.C});
\]

if \(d == 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used:
  - For the MVN variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
  - For the MVNS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.
- `<const>` For encoding A1: an immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values.
  - For encoding T1: an immediate value. See Modified immediate constants in T32 instructions on page F1-7242 for the range of values.

**Operation for all encodings**

\[
\text{if ConditionPassed()} \text{ then}
\]

\[
\text{EncodingSpecificOperations();}
\]

\[
\text{result} = \text{NOT}(\text{imm32});
\]

\[
\text{if } d == 15 \text{ then} \quad \text{// Can only occur for A32 encoding}
\]

\[
\text{if } \text{setflags} \text{ then}
\]

\[
\text{ALUExceptionReturn(result)};
\]

\[
\text{else}
\]

\[
\text{ALUWritePC(result)};
\]

\[
\text{else}
\]

\[
\text{R}[d] = \text{result};
\]

\[
\text{if } \text{setflags} \text{ then}
\]

\[
\text{PSTATE.N} = \text{result<31>};
\]

\[
\text{PSTATE.Z} = \text{IsZeroBit(result)};
\]

\[
\text{PSTATE.C} = \text{carry};
\]

\[
\text{// PSTATE.V unchanged}
\]
F5.1.124  MVN, MVNS (register)

Bitwise NOT (register) writes the bitwise inverse of a register value to the destination register.

If the destination register is not the PC, the MVNS variant of the instruction updates the condition flags based on the result.

The field descriptions for \(<Rd>\) identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The MVN variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- The MVNS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_{current_mode}.
  - The PE checks SPSR_{current_mode} for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

### A1

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 12 | 11 |  7 |  6 |  5 |  4 |  3 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| l=1110 | 0 | 0 | 0 | 1 | 1 | 1 | S | [0](0)[0](0) | Rd | imm5 | stype | 0 | Rm |

**MVN, rotate right with extend variant**

Applies when \(S = 0 \land \text{imm5} = 00000 \land \text{stype} = 11\).

\[
\text{MVN} \{<c>\} \{<q>\} <Rd>, <Rm>, RRX
\]

**MVN, shift or rotate by value variant**

Applies when \(S = 0 \land \text{imm5} = 00000 \land \text{stype} = 11\).

\[
\text{MVN} \{<c>\} \{<q>\} <Rd>, <Rm> \{, <shift> \#<amount>\}
\]

**MVNS, rotate right with extend variant**

Applies when \(S = 1 \land \text{imm5} = 00000 \land \text{stype} = 11\).

\[
\text{MVNS} \{<c>\} \{<q>\} <Rd>, <Rm>, RRX
\]

**MVNS, shift or rotate by value variant**

Applies when \(S = 1 \land \text{imm5} = 00000 \land \text{stype} = 11\).

\[
\text{MVNS} \{<c>\} \{<q>\} <Rd>, <Rm> \{, <shift> \#<amount>\}
\]

**Decode for all variants of this encoding**

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad \text{setflags} = (S = '1'); \quad \text{(shift_t, shift_n)} = \text{DecodeImmShift(stype, imm5)};
\]
### T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5</th>
<th>3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 0 0 0 0</td>
<td>1 1 1 1</td>
<td>Rm</td>
</tr>
</tbody>
</table>

#### T1 variant

MVN{<c>{<q>} <Rd>, <Rm> // Inside IT block
MVNS{<q>} <Rd>, <Rm> // Outside IT block

**Decode for this encoding**

\[d = \text{UInt}(Rd); \; m = \text{UInt}(Rm); \; \text{setflags} = \neg \text{InITBlock};\]
\[(\text{shift}_t, \text{shift}_n) = (\text{SRType}_{LSL}, 0);\]

### T2

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15 14 12 11</th>
<th>8 7 6 5</th>
<th>4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 1 1 1 0 1 0 1</td>
<td>0 0 1 1</td>
<td>S 1 1 1 1</td>
<td>(0)</td>
<td>imm3</td>
<td>Rd</td>
</tr>
</tbody>
</table>

#### MVN, rotate right with extend variant

Applies when \(S = 0 \land \text{imm3} = 000 \land \text{imm2} = 000 \land \text{stype} = 11\).

MVN{<c>{<q>} <Rd>, <Rm>, RRX

#### MVN, shift or rotate by value variant

Applies when \(S = 0 \land \neg(\text{imm3} = 000 \land \text{imm2} = 000 \land \text{stype} = 11)\).

MVN{<c>{<q>} <Rd>, <Rm> can be represented in T1
MVN{<c>{<q>} <Rd>, <Rm> {, <shift> #<amount>}

#### MVNS, rotate right with extend variant

Applies when \(S = 1 \land \text{imm3} = 000 \land \text{imm2} = 000 \land \text{stype} = 11\).

MVNS{<q>} <Rd>, <Rm>, RRX

#### MVNS, shift or rotate by value variant

Applies when \(S = 1 \land \neg(\text{imm3} = 000 \land \text{imm2} = 000 \land \text{stype} = 11)\).

MVNS.W <Rd>, <Rm> can be represented in T1
MVNS{<c>{<q>} <Rd>, <Rm> {, <shift> #<amount>}

**Decode for all variants of this encoding**

\[d = \text{UInt}(Rd); \; m = \text{UInt}(Rm); \; \text{setflags} = (S \neq '1');\]
\[(\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{imm3} : \text{imm2});\]
\[\text{if } d = 15 \land m = 15 \text{ then UNPREDICTABLE}; \quad \text{Armv8-A removes UNPREDICTABLE for R13}\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<c>
See Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

<Rd>
For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the MVN variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- For the MVNS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1 and T2: is the general-purpose destination register, encoded in the "Rd" field.

<Rm>
For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the general-purpose source register, encoded in the "Rm" field.

<shift>
Is the type of shift to be applied to the source register, encoded in the "stype" field. It can have the following values:

- **LSL** when stype = 00
- **LSR** when stype = 01
- **ASR** when stype = 10
- **ROR** when stype = 11

<amount>
For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = NOT(shifted);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
        // PSTATE.V unchanged
```
F5.1.125 MVN, MVNS (register-shifted register)

Bitwise NOT (register-shifted register) writes the bitwise inverse of a register-shifted register value to the destination register. It can optionally update the condition flags based on the result.

A1

```
[31 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 8 7 6 5 4 3 2 1 0]
  !=1111 0 0 0 1 1 1 1 S(0)(0)(0) Rd Rs 0 stype 1 Rm
  cond
```

Flag setting variant

Applies when $S == 1$.

$MVNS{<c>}{<q}> <Rd>, <Rm>, <shift> <Rs>$

Not flag setting variant

Applies when $S == 0$.

$MVN{<c>}{<q}> <Rd>, <Rm>, <shift> <Rs>$

Decode for all variants of this encoding

```
d = UInt(Rd);  m = UInt(Rm);  s = UInt(Rs);
setflags = ($S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

$<c>$ See Standard assembler syntax fields on page F1-7228.

$<q>$ See Standard assembler syntax fields on page F1-7228.

$<Rd>$ Is the general-purpose destination register, encoded in the "Rd" field.

$<Rm>$ Is the general-purpose source register, encoded in the "Rm" field.

$<shift>$ Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

<table>
<thead>
<tr>
<th>Shift Type</th>
<th>Type of Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL</td>
<td>when stype = 00</td>
</tr>
<tr>
<td>LSR</td>
<td>when stype = 01</td>
</tr>
<tr>
<td>ASR</td>
<td>when stype = 10</td>
</tr>
<tr>
<td>ROR</td>
<td>when stype = 11</td>
</tr>
</tbody>
</table>

$<Rs>$ Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  shift_n = UInt(R[s]<7:0>);
```
(shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
result = NOT(shifted);
R[d] = result;
if setflags then
  PSTATE.N = result<31>;
PSTATE.Z = IsZeroBit(result);
PSTATE.C = carry;
// PSTATE.V unchanged
F5.1.126   NOP

No Operation does nothing. This instruction can be used for instruction alignment purposes.

Note

The timing effects of including a NOP instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, NOP instructions are not suitable for timing loops.

A1

```
|31|28|27|26|25|24|23|22|21|20|19|18|17|16|15|14|13|12|11|10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0|
|!|1|1|1|1|0|0|1|1|0|0|0|0|0|0|1|1|1|1|1|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|
```

A1 variant

NOP{<c>{<q>}

Decode for this encoding

// No additional decoding required

T1

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

T1 variant

NOP{<c>{<q>}

Decode for this encoding

// No additional decoding required

T2

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

T2 variant

NOP{<c>}{.W

Decode for this encoding

// No additional decoding required
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

<
See Standard assembler syntax fields on page F1-7228.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  // Do nothing

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.127  ORN, ORNS (immediate)

Bitwise OR NOT (immediate) performs a bitwise (inclusive) OR of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 14 | 12 | 11 | 8 | 7 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1  | 1  | 1  | 0 | 0  | 0  | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | S | 0  | 1 | 0 | imm3 | Rd | imm8 | Rn |

Flag setting variant

Applies when S == 1.

ORNS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

Not flag setting variant

Applies when S == 0.

ORN{<c>}{<q>} {<Rd>,} <Rn>, #<const>

Decode for all variants of this encoding

if Rn == '1111' then SEE "MVN (immediate)";
\[ d = \text{UInt}(Rd); n = \text{UInt}(Rn); 
\text{setflags} = (S == '1'); 
(\text{imm32}, \text{carry}) = \text{T32ExpandImm_C}(i:imm3:imm8, \text{PSTATE.C}); 
\text{if } d == 15 \text{ then UNPREDICTABLE}; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<\>  See Standard assembler syntax fields on page F1-7228.

<q>  See Standard assembler syntax fields on page F1-7228.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> Is the general-purpose source register, encoded in the "Rn" field.

<\const> An immediate value. See Modified immediate constants in T32 instructions on page F1-7242 for the range of values.

Operation

if ConditionPassed() then
\text{EncodingSpecificOperations();} 
result = R[n] OR NOT(imm32); 
R[d] = result; 
if setflags then 
\text{PSTATE.N} = \text{result}<31>; 
\text{PSTATE.Z} = \text{IsZeroBit(result)};
PSTATE.C = carry;
// PSTATE.V unchanged

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.128   ORN, ORNS (register)

Bitwise OR NOT (register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd</td>
<td>imm3</td>
<td>S</td>
<td>imm2</td>
</tr>
</tbody>
</table>

ORN, rotate right with extend variant

Applies when \( S == 0 && \text{imm3} == 000 \) \&\& \( \text{imm2} == 00 \) \&\& \( \text{stype} == 11 \).

ORN{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

ORN, shift or rotate by value variant

Applies when \( S == 0 && \neg(\text{imm3} == 000 \) \&\& \( \text{imm2} == 00 \) \&\& \( \text{stype} == 11 \).

ORN{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

ORNS, rotate right with extend variant

Applies when \( S == 1 \) \&\& \( \text{imm3} == 000 \) \&\& \( \text{imm2} == 00 \) \&\& \( \text{stype} == 11 \).

ORNS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

ORNS, shift or rotate by value variant

Applies when \( S == 1 \) \&\& \( \neg(\text{imm3} == 000 \) \&\& \( \text{imm2} == 00 \) \&\& \( \text{stype} == 11 \).

ORNS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

**Decode for all variants of this encoding**

if \( \text{Rn} == '1111' \) then SEE "MVN (register)";
\[ d = \text{UInt}(\text{Rd}); \text{ } n = \text{UInt}(\text{Rn}); \text{ } m = \text{UInt}(\text{Rm}); \text{ } \text{setflags} = (S == '1'); \]
\[ (\text{shift}_t, \text{shift}_n) = \text{DecodelImmShift}(\text{stype}, \text{imm3:imm2}); \]
if \( d == 15 \) \&\& \( m == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

\(<c>\) See Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.

\(<Rn>\) Is the first general-purpose source register, encoded in the "Rn" field.

\(<Rm>\) Is the second general-purpose source register, encoded in the "Rm" field.
<shift> Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

<table>
<thead>
<tr>
<th>&lt;shift&gt;</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL</td>
<td>when stype = 00</td>
</tr>
<tr>
<td>LSR</td>
<td>when stype = 01</td>
</tr>
<tr>
<td>ASR</td>
<td>when stype = 10</td>
</tr>
<tr>
<td>ROR</td>
<td>when stype = 11</td>
</tr>
</tbody>
</table>

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation

if ConditionPassed() then
   EncodingSpecificOperations();
   (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
   result = R[n] OR NOT(shifted);
   R[d] = result;
   if setflags then
      PSTATE.N = result<31>;
      PSTATE.Z = IsZeroBit(result);
      PSTATE.C = carry;
      // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.129   ORR, ORRS (immediate)

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the ORRS variant of the instruction updates the condition flags based on the result.

The field descriptions for \(<Rd>\) identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ORR variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- The ORRS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=1111</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ORR variant**

Applies when $S == 0$.

ORR{<c>}{<q>} {<Rd>,} <Rn>, #<const>

**ORRS variant**

Applies when $S == 1$.

ORRS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

**Decode for all variants of this encoding**

\[ d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ \text{setflags} = (S == '1'); \]
\[ (\text{imm32}, \text{carry}) = \text{A32ExpandImm_C}(\text{imm12}, \text{PSTATE}.C); \]

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | | 0 | 15 | 14 | 12 | 11 | 8 | 7 | | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|----|----|----|----|----|----|----|---|
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | S | i=1111 | 0 | imm3 | Rd | imm8 |
| Rn |

**ORR variant**

Applies when $S == 0$.

ORR{<c>}{<q>} {<Rd>,} <Rn>, #<const>
**ORRS variant**

Applies when \( S = 1 \).

\[
\text{ORRS}\{<c>\}{<q>} \{<Rd>,\} <Rn>, \#<constant>
\]

**Decode for all variants of this encoding**

\[
\text{if } Rn == \text{'}1111\text{'} \text{ then SEE } \text{"MOV (immediate)"};
\]

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ \text{setflags} = (S == \text{'}1\text{'});
\]

\[
(\text{imm32}, \text{carry}) = T32\text{ExpandImm}_C(i:imm3:imm8, \text{PSTATE.C});
\]

\[
\text{if } d == 15 \text{ then UNPREDICTABLE; } // \text{Armv8-A removes UNPREDICTABLE for R13}
\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- \(<c>\)
  
  See *Standard assembler syntax fields on page F1-7228.*

- \(<q>\)
  
  See *Standard assembler syntax fields on page F1-7228.*

- \(<Rd>\)
  
  For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as \(<Rn>\). Arm deprecates using the PC as the destination register, but if the PC is used:
  
  - For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.*
  
  - For the ORRS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

  For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as \(<Rn>\).

- \(<Rn>\)
  
  For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

  For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

- \(<constant>\)
  
  For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions on page F1-7244* for the range of values.

  For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions on page F1-7242* for the range of values.

**Operation for all encodings**

\[
\text{if } \text{ConditionPassed()} \text{ then}
\]

\[
\text{EncodingSpecificOperations();}
\]

\[
\text{result} = R[n] \text{ OR imm32;}
\]

\[
\text{if } d == 15 \text{ then } \quad // \text{Can only occur for A32 encoding}
\]

\[
\text{if setflags then}
\]

\[
\text{ALUExceptionReturn(result);}\]

\[
\text{else}
\]

\[
\text{ALUWritePC(result);}\]

\[
\text{else}
\]

\[
R[d] = \text{result;}
\]

\[
\text{if setflags then}
\]

\[
PSTATE.N = \text{result} <31>;
\]

\[
PSTATE.Z = \text{IsZeroBit(result);}\]
PSTATE.C = carry;
// PSTATE.V unchanged

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
### F5.1.130 ORR, ORRS (register)

Bitwise OR (register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the ORRS variant of the instruction updates the condition flags based on the result.

The field descriptions for `<Rd>` identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ORR variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133](#).
- The ORRS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores `PSTATE` from `SPSR_{<current_mode>}`.
  - The PE checks `SPSR_{<current_mode>}` for an illegal return event. See [Illegal return events from AArch32 state on page G1-8952](#).
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

### A1

<table>
<thead>
<tr>
<th>[31] 28 27 26 25 24 23 22 21 20 19 16 15 12 11</th>
<th>7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>(l=1111)</td>
<td></td>
</tr>
<tr>
<td>(0) (0) (0) (1) (0) (0) (S)</td>
<td>(Rn) (Rd) (imm5) (stype) (0)</td>
</tr>
<tr>
<td><strong>cond</strong></td>
<td></td>
</tr>
</tbody>
</table>

#### ORR, rotate right with extend variant

Applies when \(S = 0 \&\& \text{imm5} = 00000 \&\& \text{stype} = 11\).

\[
\text{ORR}\{<c>\}{<q>} \{<Rd>,\} \langle Rn\rangle, \langle Rm\rangle, \text{RRX}
\]

#### ORR, shift or rotate by value variant

Applies when \(S = 0 \&\& \neg(\text{imm5} = 00000 \&\& \text{stype} = 11)\).

\[
\text{ORR}\{<c>\}{<q>} \{<Rd>,\} \langle Rn\rangle, \langle Rm\rangle \{, <shift> \#<amount>\}
\]

#### ORRS, rotate right with extend variant

Applies when \(S = 1 \&\& \text{imm5} = 00000 \&\& \text{stype} = 11\).

\[
\text{ORRS}\{<c>\}{<q>} \{<Rd>,\} \langle Rn\rangle, \langle Rm\rangle, \text{RRX}
\]

#### ORRS, shift or rotate by value variant

Applies when \(S = 1 \&\& \neg(\text{imm5} = 00000 \&\& \text{stype} = 11)\).

\[
\text{ORRS}\{<c>\}{<q>} \{<Rd>,\} \langle Rn\rangle, \langle Rm\rangle \{, <shift> \#<amount>\}
\]

#### Decode for all variants of this encoding

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{setflags} = (S == '1');
\]

\[
\text{(shift}_t, \text{shift}_n) = \text{DecodeImmShift(stype, imm5)};
\]
T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5</th>
<th>3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 0</td>
<td>1 1 0 0</td>
<td>Rm</td>
<td>Rdn</td>
</tr>
</tbody>
</table>

**T1 variant**

ORR,<c>{<q>} {<Rdn>}, <Rdn>, <Rm> // Inside IT block
ORRS,<c>{<q>} {<Rdn>}, <Rdn>, <Rm> // Outside IT block

**Decode for this encoding**

d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);

T2

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5</th>
<th>4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 0 1 0 0 1 0 0 1</td>
<td>S = 1111</td>
<td>0 0</td>
<td>imm3</td>
<td>Rd</td>
</tr>
</tbody>
</table>

**ORR, rotate right with extend variant**

Applies when \( S = 0 \) && \( \text{imm3} = 000 \) && \( \text{imm2} = 00 \) && \( \text{stype} = 11 \).

**ORR,<c>{<q>} {<Rd>}, <Rn>, <Rm>, RRX**

**ORR, shift or rotate by value variant**

Applies when \( S = 0 \) && !\((\text{imm3} = 000 \) && \( \text{imm2} = 00 \) && \( \text{stype} = 11 \))

**ORRS, rotate right with extend variant**

Applies when \( S = 1 \) && \( \text{imm3} = 000 \) && \( \text{imm2} = 00 \) && \( \text{stype} = 11 \).

**ORRS,<c>{<q>} {<Rd>}, <Rn>, <Rm>, RRX**

**ORRS, shift or rotate by value variant**

Applies when \( S = 1 \) && !\((\text{imm3} = 000 \) && \( \text{imm2} = 00 \) && \( \text{stype} = 11 \))

**Decode for all variants of this encoding**

if \( \text{Rn} = '1111' \) then SEE "Related encodings";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 \&\& m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Related encodings: Data-processing (shifted register) on page F3-7308

Assembler symbols

<<> See Standard assembler syntax fields on page F1-7228.

<<q> See Standard assembler syntax fields on page F1-7228.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:
   • For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
   • For the ORRS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
   LSL when stype = 00
   LSR when stype = 01
   ASR when stype = 10
   ROR when stype = 11

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

In T32 assembly:
   • Outside an IT block, if ORRS <Rd>, <Rn>, <Rd> is written with <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ORRS <Rd>, <Rn> had been written.
   • Inside an IT block, if ORR<< <Rd>, <Rn>, <Rd> is written with <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ORR<< <Rd>, <Rn> had been written.

To prevent either of these happening, use the .W qualifier.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[n], shift_t, shift_n, PSTATE.C);
    result = R[n] OR shifted;
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
ALUExceptionReturn(result);
else
    ALUWritePC(result);
else
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.131 ORR, ORRS (register-shifted register)

Bitwise OR (register-shifted register) performs a bitwise (inclusive) OR of a register value and a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

```
| 31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 8 7 6 5 4 3 0 |
|---|---|---|---|---|---|---|---|---|---|
| !|=1111 0 0 0 1 1 0 0 S | Rn | Rd | Rs 0 | stype | 1 | Rm |
```

Flag setting variant

Applies when $S = 1$.

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting variant

Applies when $S = 0$.

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

```
<c>  See Standard assembler syntax fields on page F1-7228.
<q>  See Standard assembler syntax fields on page F1-7228.
<Rd> Is the general-purpose destination register, encoded in the "Rd" field.
<Rn> Is the first general-purpose source register, encoded in the "Rn" field.
<Rm> Is the second general-purpose source register, encoded in the "Rm" field.
<shift>  Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
  LSL   when stype = 00
  LSR   when stype = 01
  ASR   when stype = 10
  ROR   when stype = 11
<Rs>  Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.
```
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] OR shifted;
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.132  PKHBT, PKHTB

Pack Halfword combines one halfword of its first operand with the other halfword of its shifted second operand.

A1

PKHBT variant

Applies when \( \text{tb} = 0 \).

\[
\text{PKHBT}\{\langle c\rangle\}{\langle q\rangle}\ {\langle R_{\text{d}},\rangle} {\langle R_{\text{n}}, \langle R_{\text{m}}\rangle}, {\langle \text{LSL #imm}\rangle}}
\]

PKHTB variant

Applies when \( \text{tb} = 1 \).

\[
\text{PKHTB}\{\langle c\rangle\}{\langle q\rangle} {\langle R_{\text{d}},\rangle} {\langle R_{\text{n}}, \langle R_{\text{m}}\rangle}, {\langle \text{ASR #imm}\rangle}}
\]

Decode for all variants of this encoding

\[
d = \text{UInt}(\text{Rd}); \ n = \text{UInt}(\text{Rn}); \ m = \text{UInt}(\text{Rm}); \ \text{tbform} = (\text{tb} == '1');
\]

\[
(\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{tb}:'0', \text{imm5});
\]

\[
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE;}
\]

T1

PKHBT variant

Applies when \( \text{tb} = 0 \).

\[
\text{PKHBT}\{\langle c\rangle\}{\langle q\rangle} {\langle R_{\text{d}},\rangle} {\langle R_{\text{n}}, \langle R_{\text{m}}\rangle}, {\langle \text{LSL #imm}\rangle}} \quad \text{\textbackslash{}text{tbform} == \text{FALSE}}
\]

PKHTB variant

Applies when \( \text{tb} = 1 \).

\[
\text{PKHTB}\{\langle c\rangle\}{\langle q\rangle} {\langle R_{\text{d}},\rangle} {\langle R_{\text{n}}, \langle R_{\text{m}}\rangle}, {\langle \text{ASR #imm}\rangle}} \quad \text{\textbackslash{}text{tbform} == \text{TRUE}}
\]

Decode for all variants of this encoding

\[
\text{if } S == '1' || T == '1' \text{ then UNDEFINED;}
\]

\[
d = \text{UInt}(\text{Rd}); \ n = \text{UInt}(\text{Rn}); \ m = \text{UInt}(\text{Rm}); \ \text{tbform} = (\text{tb} == '1');
\]

\[
(\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{tb}:'0', \text{imm3:imm2});
\]

\[
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

Notes for all encodings

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

<Rd>
Is the general-purpose destination register, encoded in the "Rd" field.

<Rn>
Is the first general-purpose source register, encoded in the "Rn" field.

<Rm>
Is the second general-purpose source register, encoded in the "Rm" field.

<imm>
For encoding A1: the shift to apply to the value read from <Rm>, encoded in the "imm5" field.
For PKHBT, it is one of:

<table>
<thead>
<tr>
<th>Shift Type</th>
<th>Encoded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>omitted</td>
<td>0b00000</td>
</tr>
<tr>
<td>1-31</td>
<td>Left shift by specified number of bits, encoded as a binary number.</td>
</tr>
</tbody>
</table>

For PKHTB, it is one of:

<table>
<thead>
<tr>
<th>Shift Type</th>
<th>Encoded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>omitted</td>
<td>Instruction is a pseudo-instruction and is assembled as though PKHTB{&lt;c&gt;}{&lt;q&gt;} &lt;Rd&gt;, &lt;Rm&gt;, &lt;Rn&gt; had been written.</td>
</tr>
<tr>
<td>1-32</td>
<td>Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as 0b000000. Other shift amounts are encoded as binary numbers.</td>
</tr>
</tbody>
</table>

--- Note ---
An assembler can permit <imm> = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  operand2 = Shift(R[m], shift_t, shift_n, PSTATE.C); // PSTATE.C ignored
  R[d]<15:0> = if tbform then operand2<15:0> else R[n]<15:0>;
  R[d]<31:16> = if tbform then R[n]<31:16> else operand2<31:16>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
F5.1.133 PLD, PLDW (immediate)

Preload Data (immediate) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The PLD instruction signals that the likely memory access is a read, and the PLDW instruction signals that it is a write.

The effect of a PLD or PLDW instruction is IMPLEMENTATION DEFINED. For more information, see Preloading caches on page E2-7191.

A1

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19</th>
<th>16 15 14 13 12 11</th>
<th>1 0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 0 1 U R 0 1</td>
<td>1=1111</td>
<td>1 1 1</td>
<td>imm12</td>
</tr>
</tbody>
</table>
```

Preload read variant

Applies when $R = 1$.

$\text{PLD}(<c><q>)[<Rn>\{, #{/}-<imm>\}]$

Preload write variant

Applies when $R = 0$.

$\text{PLDW}(<c><q>)[<Rn>\{, #{/}-<imm>\}]$

Decode for all variants of this encoding

if $Rn == '1111'$ then SEE "PLD (literal)";
$n = \text{UInt}(Rn); \text{imm32} = \text{ZeroExtend(imm12, 32)}; \text{add} = (U == '1'); \text{is_pldw} = (R == '0');$

T1

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3</th>
<th>0</th>
<th>15 14 13 12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 0 1 0 W 1</td>
<td>1=1111</td>
<td>1 1 1 1</td>
<td>imm12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Preload read variant

Applies when $W = 0$.

$\text{PLD}(<c><q>)[<Rn>\{, #/+<imm>\}]$

Preload write variant

Applies when $W = 1$.

$\text{PLDW}(<c><q>)[<Rn>\{, #/+<imm>\}]$

Decode for all variants of this encoding

if $Rn == '1111'$ then SEE "PLD (literal)";
$n = \text{UInt}(Rn); \text{imm32} = \text{ZeroExtend(imm12, 32)}; \text{add} = \text{TRUE}; \text{is_pldw} = (W == '1');$
T2

Preload read variant
Applies when W == 0.
PLD{<c>}{<q>} [<Rn> {, #-<imm>}]

Preload write variant
Applies when W == 1.
PLDW{<c>}{<q>} [<Rn> {, #-<imm>}]

Decode for all variants of this encoding
if Rn == '1111' then SEE "PLD (literal)";
n = UInt(Rn); imm32 = ZeroExtend(imm8, 32); add = FALSE; is_pldw = (W == '1');

Notes for all encodings
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

{<c>}
For encoding A1: see Standard assembler syntax fields on page F1-7228. Must be AL or omitted.
For encoding T1 and T2: see Standard assembler syntax fields on page F1-7228.

{<q>}
See Standard assembler syntax fields on page F1-7228.

{<Rn>}
Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see PLD (literal).

{+-}
Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
- when U = 0
+ when U = 1

{+}
Specifies the offset is added to the base register.

{<imm>}
For encoding A1: is the optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation for all encodings
if ConditionPassed() then
   EncodingSpecificOperations();
   address = if add then (R[n] + imm32) else (R[n] - imm32);
   if is_pldw then
      Hint_PreloadDataForWrite(address);
else
  Hint_PreloadData(address);
F5.1.134  PLD (literal)

Preload Data (literal) signals the memory system that data memory accesses from a specified address are likely in
the near future. The memory system can respond by taking actions that are expected to speed up the memory
accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The effect of a PLD instruction is IMPLEMENTATION DEFINED. For more information, see Preloading caches on
page E2-7191.

A1

A1 variant

PLD{<c>}{<q>} <label> // Normal form
PLD{<c>}{<q>} [PC, #{+/-}<imm>] // Alternative form

Decode for this encoding

\[ imm32 = \text{ZeroExtend}(imm12, 32); \text{ add } = (U == \text{'1'}); \]

T1

T1 variant

PLD{<c>}{<q>} <label> // Preferred syntax
PLD{<c>}{<q>} [PC, #{+/-}<imm>] // Alternative syntax

Decode for this encoding

\[ imm32 = \text{ZeroExtend}(imm12, 32); \text{ add } = (U == \text{'1'}); \]

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1
Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\)  
For encoding A1: see Standard assembler syntax fields on page F1-7228. Must be AL or omitted.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

\(<q>\)  
See Standard assembler syntax fields on page F1-7228.

\(<label>\)  
The label of the literal data item that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. The offset must be in the range –4095 to 4095.
If the offset is zero or positive, \(imm32\) is equal to the offset and \(add\) == TRUE.
If the offset is negative, \(imm32\) is equal to minus the offset and \(add\) == FALSE.
+/-  Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
-  when U = 0
+  when U = 1

<imm>  For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.
        For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax on page F2-7257.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    Hint_PreloadData(address);
```
F5.1.135  PLD, PLDW (register)

Preload Data (register) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The PLD instruction signals that the likely memory access is a read, and the PLDW instruction signals that it is a write.

The effect of a PLD or PLDW instruction is IMPLEMENTATION DEFINED. For more information, see Preloading caches on page E2-7191.

A1

\[
\begin{array}{ccccccccc}
1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & U & R & 0 & 1 & Rn & (1)(1)(1)(1) & imm5 & stype & 0 & Rm
\end{array}
\]

**Preload read, optional shift or rotate variant**

Applies when \( R == 1 \) \&\& \(! (imm5 == 00000 \&\& stype == 11)\).

\[
\text{PLD}{<c}>{<q>}[<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
\]

**Preload read, rotate right with extend variant**

Applies when \( R == 1 \) \&\& \( imm5 == 00000 \&\& stype == 11 \).

\[
\text{PLD}{<c}>{<q>}[<Rn>, {+/-}<Rm> , RRX]
\]

**Preload write, optional shift or rotate variant**

Applies when \( R == 0 \) \&\& \(! (imm5 == 00000 \&\& stype == 11)\).

\[
\text{PLDW}{<c}>{<q>}[<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
\]

**Preload write, rotate right with extend variant**

Applies when \( R == 0 \) \&\& \( imm5 == 00000 \&\& stype == 11 \).

\[
\text{PLDW}{<c}>{<q>}[<Rn>, {+/-}<Rm> , RRX]
\]

**Decode for all variants of this encoding**

\[
n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad add = (U == '1'); \quad \text{is_pldw} = (R == '0');
\]

\[
(shift_t, shift_n) = \text{DecodeImmShift}(stype, imm5);
\]

\[
\text{if } m == 15 || (n == 15 \&\& \text{is_pldw}) \text{ then UNPREDICTABLE;}
\]

T1

\[
\begin{array}{ccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & imm2 & Rm
\end{array}
\]

**Preload read variant**

Applies when \( W == 0 \).

\[
\text{PLD}{<c}>{<q>}[<Rn>, {+}<Rm> {, LSL #<amount>}]\]
Preload write variant

Applies when \( W == 1 \).

\[ \text{PLDW}\{\langle c\rangle\}\{\langle q\rangle\} \{<Rn>, \{+\}<Rm> \{, \text{LSL } \langle\text{amount}\rangle\} \]  

Decode for all variants of this encoding

if \( Rn == '1111' \) then SEE "PLD (literal)";

\[ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \ add = \text{TRUE}; \ is\_pldw = (W == '1'); \]
\[ (\text{shift}_t, \text{shift}_n) = (\text{SRType}_\text{LSL}, \text{UInt}(\text{imm}_2)); \]
\[ \text{if } m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13} \]

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<\text{c}>\) For encoding A1: see Standard assembler syntax fields on page F1-7228. \(<\text{c}>\) must be \text{AT} or omitted.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

\(<\text{q}>\) See Standard assembler syntax fields on page F1-7228.

\(<\text{Rn}>\) For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used.

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

\(+/-\) Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- \( \text{when } U == 0 \)
- \( \text{when } U == 1 \)

\(+\) Specifies the index register is added to the base register.

\(<\text{Rm}>\) Is the general-purpose index register, encoded in the "Rm" field.

\(<\text{shift}>\) Is the type of shift to be applied to the index register, encoded in the "stype" field. It can have the following values:

- \( \text{LSL when } \text{stype} == \text{00} \)
- \( \text{LSR when } \text{stype} == \text{01} \)
- \( \text{ASR when } \text{stype} == \text{10} \)
- \( \text{ROR when } \text{stype} == \text{11} \)

\(<\text{amount}>\) For encoding A1: is the shift amount, in the range 1 to 31 (when \(<\text{shift}> == \text{LSL or ROR}\) or 1 to 32 (when \(<\text{shift}> == \text{LSR or ASR}\) encoded in the "imm5" field as \(<\text{amount}> \mod 32\).

For encoding T1: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the "imm2" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
else
    Hint_PreloadData(address);
F5.1.136   PLI (immediate, literal)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

The effect of a PLI instruction is IMPLEMENTATION DEFINED. For more information, see Preloading caches on page E2-7191.

A1

\[
\begin{array}{c|c|c}
1 & 1 & 1 & 0 & 1 & 0 & 0 & U & 1 & 0 & 1 & | & 1 & 1 & 1 & \hline
\end{array}
\]

A1 variant

\[
\text{PLI}\{<c>\}{<q>} \{<Rn> \, , \, #{+/-}<imm>\}
\]
\[
\text{PLI}\{<c>\}{<q>} <label> // Normal form
\]
\[
\text{PLI}\{<c>\}{<q>} [PC, #{+/-}<imm>] // Alternative form
\]

**Decode for this encoding**

\[
n = \text{UInt}(Rn); \quad \text{imm32} = \text{ZeroExtend}(\text{imm12}, 32); \quad \text{add} = (U = '1');
\]

T1

\[
\begin{array}{c|c|c}
| & | & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & | & 15 & 14 & 13 & 12 & 11 & | & 0 & | \hline
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & | & 1 & 1 & 1 & \hline
\end{array}
\]

T1 variant

PLI\{<c>\}{<q>} \{<Rn> \, , \, #{+}<imm>\}

**Decode for this encoding**

\[
\text{if } Rn \text{ == '1111' then SEE "encoding T3";}
\]
\[
n = \text{UInt}(Rn); \quad \text{imm32} = \text{ZeroExtend}(\text{imm12}, 32); \quad \text{add} = \text{TRUE};
\]

T2

\[
\begin{array}{c|c|c}
| & | & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & | & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & | & 0 & | \hline
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & \hline
\end{array}
\]

T2 variant

PLI\{<c>\}{<q>} \{<Rn> \, , \, #{-}<imm>\}

**Decode for this encoding**

\[
\text{if } Rn \text{ == '1111' then SEE "encoding T3";}
\]
\[
n = \text{UInt}(Rn); \quad \text{imm32} = \text{ZeroExtend}(\text{imm8}, 32); \quad \text{add} = \text{FALSE};
\]
T3

| 15 14 13 12| 11 10 9 8 | 7 6 5 4 | 3 2 1 0 | 15 14 13 12| 11 | 0 |
| 1 1 1 1 1 0 0 | 1 1 1 1 1 | 1 1 1 1 | imm12 |

**T3 variant**

PLI{<c>}{<q>} <label> // Preferred syntax
PLI{<c>}{<q>} [PC, #<+/-><imm>] // Alternative syntax

**Decode for this encoding**

\[ n = 15; \text{imm32} = \text{ZeroExtend}(\text{imm12}, 32); \text{add} = (U == '1'); \]

**Notes for all encodings**

For more information about the CONSTRANGED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

\(<c>\) For encoding A1: see Standard assembler syntax fields on page F1-7228. Must be At or omitted.

For encoding T1, T2 and T3: see Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<label>\) The label of the instruction that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the A1ign(PC, 4) value of the instruction to this label. The offset must be in the range -4095 to 4095.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE.

If the offset is negative, imm32 is equal to minus the offset and add == FALSE.

\(<Rn>\) Is the general-purpose base register, encoded in the "Rn" field.

\(+/-\) Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when \( U = 0 \)
+ when \( U = 1 \)

\(+\) Specifies the offset is added to the base register.

\(<imm>\) For encoding A1: is the optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

For encoding T3: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

For the literal forms of the instruction, encoding T3 is used, or Rn is encoded as 0b1111 in encoding A1, to indicate that the PC is the base register.

The alternative literal syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax on page F2-7257.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    Hint_PreloadInstr(address);
F5.1.137  PLI (register)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

The effect of a PLI instruction is IMPLEMENTATION DEFINED. For more information, see Preloading caches on page E2-7191.

A1

\[
\begin{array}{ccccccccccc}
\end{array}
\]

<table>
<thead>
<tr>
<th>Rn</th>
<th>imm5</th>
<th>stype</th>
<th>m</th>
<th>Rm</th>
</tr>
</thead>
</table>

Rotate right with extend variant

Applies when \( \text{imm5} == 00000 \) && \( \text{stype} == 11 \).

\[ \text{PLI}\{<c>\}{<q>} [\text{<Rn>}, {+/-}\text{<Rm> }, \text{RRX}] \]

Shift or rotate by value variant

Applies when \((\text{imm5} == 00000 \) && \( \text{stype} == 11 \)).

\[ \text{PLI}\{<c>\}{<q>} [\text{<Rn>}, {+/-}\text{<Rm>}, \text{<shift> #<amount>}] \]

Decode for all variants of this encoding

\[
n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \ \text{add} = (\text{U} == '1');
\]

\[
(\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{imm5});
\]

if \( m == 15 \) then UNPREDICTABLE;

T1

\[
\begin{array}{ccccccccccc}
| 15 & 14 & 13 & 12 | 11 & 10 & 9 & 8 | 7 & 6 & 5 & 4 & 3 & 0 | 15 & 14 & 13 & 12 | 11 & 10 & 9 & 8 | 7 & 6 & 5 & 4 & 3 & 0 |
\end{array}
\]

<table>
<thead>
<tr>
<th>Rn</th>
<th>imm2</th>
<th>Rm</th>
</tr>
</thead>
</table>

T1 variant

\[ \text{PLI}\{<c>\}{<q>} [\text{<Rn>}, {+}<\text{Rm}>, \text{LSL} #<\text{amount}>] \]

Decode for this encoding

if \( Rn == '1111' \) then SEE "PLI (immediate, literal)"

\[
n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \ \text{add} = \text{TRUE};
\]

\[
(\text{shift}_t, \text{shift}_n) = (\text{SRType}_{LSL}, \text{UInt}(\text{imm2}));
\]

if \( m == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<> 
For encoding A1: see Standard assembler syntax fields on page F1-7228. <> must be AL or omitted.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<> 
See Standard assembler syntax fields on page F1-7228.

<\n> 
Is the general-purpose base register, encoded in the "Rn" field.

+/- 
Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
- when U = 0
+ when U = 1

+ 
Specifies the index register is added to the base register.

<\m> 
Is the general-purpose index register, encoded in the "Rm" field.

<\shift> 
Is the type of shift to be applied to the index register, encoded in the "stype" field. It can have the following values:
- LSL when stype = 00
- LSR when stype = 01
- ASR when stype = 10
- ROR when stype = 11

<\amount> 
For encoding A1: is the shift amount, in the range 1 to 31 (when <\shift> = LSL or ROR) or 1 to 32 (when <\shift> = LSR or ASR) encoded in the "imm5" field as <\amount> modulo 32.
For encoding T1: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the "imm2" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    Hint_PreloadInstr(address);
F5.1.138 POP

Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC on page F1-7234.

The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1 1 0</td>
<td>P</td>
<td>register_list</td>
</tr>
</tbody>
</table>

T1 variant

POP{<c>}{<q>} <registers> // Preferred syntax
LDM{<c>}{<q>} SP!, <registers> // Alternate syntax

Decode for this encoding

register = P:"0000000":register_list; UnalignedAllowed = FALSE;
if BitCount(registers) < 1 then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<i> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<registers> Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the PC. If the PC is in the list, the "P" field is set to 1, otherwise this field defaults to 0.

If the PC is in the list, the instruction must be either outside any IT block, or the last instruction in an IT block.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[13];
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = if UnalignedAllowed then MemU[address,4] else MemA[address,4];
            address = address + 4;
        if registers<15> == '1' then
            if UnalignedAllowed then
                if address<1:0> == '00' then
                    LoadWritePC(MemU[address,4]);
                else
                    UNPREDICTABLE;
            else
                LoadWritePC(MemA[address,4]);
            if registers<13> == '0' then R[13] = R[13] + 4\BitCount(registers);
            if registers<13> == '1' then R[13] = bits(32) UNKNOWN;
F5.1.139   POP (multiple registers)

Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data.

This instruction is an alias of the LDM, LDMIA, LDMFD instruction. This means that:

- The encodings in this description are named to match the encodings of LDM, LDMIA, LDMFD.

- The description of LDM, LDMIA, LDMFD gives the operational pseudocode for this instruction.

**A1**

![A1 encoding](image)

**A1 variant**

`POP{<c>}{<q>} <registers>`

is equivalent to

`LDM{<c>}{<q>} SP!, <registers>`

and is the preferred disassembly when `BitCount(register_list) > 1`.

**T2**

![T2 encoding](image)

**T2 variant**

`POP{<c>}W <registers>` // All registers in R0-R7, PC

is equivalent to

`LDM{<c>}SP!, <registers>`

and is the preferred disassembly when `BitCount(P:M:register_list) > 1`.

**Assembler symbols**

- `<c>`  
  See *Standard assembler syntax fields* on page F1-7228.

- `<q>`  
  See *Standard assembler syntax fields* on page F1-7228.

- `<registers>`  
  For encoding A1: is a list of two or more registers to be loaded, separated by commas and surrounded by `{ and }`. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F1-7234.

  If the SP is in the list, the value of the SP after such an instruction is UNKNOWN.
The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.

Arm deprecates the use of this instruction with both the LR and the PC in the list.

For encoding T2: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC on page F1-7234.

The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0.

The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133. If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

**Operation for all encodings**

The description of LDM, LDMIA, LDMFD gives the operational pseudocode for this instruction.
F5.1.140   POP (single register)

Pop Single Register from Stack loads a single general-purpose register from the stack, loading from the address in SP, and updates SP to point just above the loaded data.

This instruction is an alias of the LDR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of LDR (immediate).
- The description of LDR (immediate) gives the operational pseudocode for this instruction.

### A1

| 31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| l=1111 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

**Post-indexed variant**

POP{<c>}{<q>} <single_register_list>

is equivalent to

LDR{<c>}{<q>} <Rt>, [SP], #4

and is always the preferred disassembly.

### T4

| 15 14 13 12|11 10 9 8|7 6 5 4|3 | 0 |15 |12|11 10 9 8|7 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Post-indexed variant**

POP{<c>}{<q>} <single_register_list>

is equivalent to

LDR{<c>}{<q>} <Rt>, [SP], #4

and is always the preferred disassembly.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields on page F1-7228.*
- `<q>` See *Standard assembler syntax fields on page F1-7228.*
- `<single_register_list>`
  
  Is the general-purpose register `<Rt>` to be loaded surrounded by `{` and `}`.
- `<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.*
For encoding T4: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.

**Operation for all encodings**

The description of LDR (immediate) gives the operational pseudocode for this instruction.
F5.1.141  PSSBB

Physical Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same physical address.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the PSSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the PSSBB.

- When a load to a location appears in program order before the PSSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the PSSBB.

A1

<table>
<thead>
<tr>
<th>[31 30 29 28]</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 1 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 variant

PSSBB{<q>}

Decode for this encoding

// No additional decoding required

T1

<table>
<thead>
<tr>
<th>[15 14 13 12]</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>[15 14 13 12]</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1 1 0 1 1 1 1 1 0 1 1 1 1 0 1 0 0 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T1 variant

PSSBB{<q>}

Decode for this encoding

if InITBlock() then UNPREDICTABLE;

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    SpeculativeStoreBypassBarrierToPA();
F5.1.142 PUSH

Push Multiple Registers to Stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data.

The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also Encoding of lists of general-purpose registers and the PC on page F1-7234.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 0 1 0 M</td>
<td>register_list</td>
<td></td>
</tr>
</tbody>
</table>

T1 variant

PUSH{<c>}{<q>} <registers> // Preferred syntax
STMDB{<c>}{<q>} SP!, <registers> // Alternate syntax

Decode for this encoding

registers = '0':M:'000000':register_list; UnalignedAllowed = FALSE;
if BitCount(registers) < 1 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

Notes for all encodings

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<>{<q>} See Standard assembler syntax fields on page F1-7228.

<>{<q>} See Standard assembler syntax fields on page F1-7228.

<registers> Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the LR. If the LR is in the list, the "M" field is set to 1, otherwise this field defaults to 0.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[13] - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            if i == 13 && i != LowestSetBit(registers) then // Only possible for encoding A1
MemA[address,4] = bits(32) UNKNOWN;
else
  if UnalignedAllowed then
    MemU[address,4] = R[i];
  else
    MemA[address,4] = R[i];
  address = address + 4;
if registers<15> == '1' then  // Only possible for encoding A1 or A2
  if UnalignedAllowed then
    MemU[address,4] = PCStoreValue();
  else
    MemA[address,4] = PCStoreValue();
F5.1.143  **PUSH (multiple registers)**

Push multiple registers to Stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data.

This instruction is an alias of the STMDB, STMFD instruction. This means that:

- The encodings in this description are named to match the encodings of STMDB, STMFD.
- The description of STMDB, STMFD gives the operational pseudocode for this instruction.

**A1**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=1111</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>register_list</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**cond**  **W**  **Rn**

**A1 variant**

PUSH{<c>}{<q>} <registers>

is equivalent to

STMDB{<c>}{<q>} SP!, <registers>

and is the preferred disassembly when BitCount(register_list) > 1.

**T1**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>M</th>
<th>register_list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**cond**  **W**  **Rn**  **P**

**T1 variant**

PUSH{<c>}.W <registers> // All registers in R0-R7, LR

is equivalent to

STMDB{<c>}{<q>} SP!, <registers>

and is the preferred disassembly when BitCount(M:register_list) > 1.

PUSH{<c>}{<q>} <registers>

is equivalent to

STMDB{<c>}{<q>} SP!, <registers>

and is the preferred disassembly when BitCount(M:register_list) > 1.

**Assembler symbols**

- **<c>**  See *Standard assembler syntax fields* on page F1-7228.
- **<q>**  See *Standard assembler syntax fields* on page F1-7228.
- **<registers>**  For encoding A1: is a list of two or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F1-7234.
The SP and PC can be in the list. However:

- Arm deprecates the use of instructions that include the PC in the list.
- If the SP is in the list, and it is not the lowest-numbered register in the list, the instruction stores an UNKNOWN value for the SP.

For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also Encoding of lists of general-purpose registers and the PC on page F1-7234.

The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

**Operation for all encodings**

The description of STMDB, STMFD gives the operational pseudocode for this instruction.
F5.1.144   PUSH (single register)

Push Single Register to Stack stores a single general-purpose register to the stack, storing to the 32-bit word below
the address in SP, and updates SP to point to the start of the stored data

This instruction is an alias of the STR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of STR (immediate).
- The description of STR (immediate) gives the operational pseudocode for this instruction.

A1

```
| 31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 |  |  | 0 |
|---|---|---|---|---|---|---|---|
| !=1111 0 1 0 1 0 1 1 0 1 1 0 | Rt 0 0 0 0 0 0 0 0 0 |
| cond P U W Rn imm12 |
```

**Pre-indexed variant**

PUSH{<c>}{<q>} <single_register_list>

is equivalent to

STR{<c>}{<q>} <Rt>, [SP, #-4]!

and is always the preferred disassembly.

T4

```
| 15 14 13 12|11 10 9 8 7 6 5 4 3 0 15 12|11 10 9 8 7 6 5 4 3 0 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 1 1 1 1 0 0 0 1 0 1 0 1 1 0 | Rt 1 1 0 1 0 0 0 0 0 0 |
| Rn P U W imm8 |
```

**Pre-indexed variant**

PUSH{<c>}{<q>} <single_register_list> // Standard syntax

is equivalent to

STR{<c>}{<q>} <Rt>, [SP, #-4]!

and is always the preferred disassembly.

**Assembler symbols**

- `<c>`  See Standard assembler syntax fields on page F1-7228.
- `<q>`  See Standard assembler syntax fields on page F1-7228.
- `<single_register_list>`
  
  Is the general-purpose register <Rt> to be stored surrounded by { and }.

- `<Rt>`
  
  For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.
  
  For encoding T4: is the general-purpose register to be transferred, encoded in the "Rt" field.

**Operation for all encodings**

The description of STR (immediate) gives the operational pseudocode for this instruction.
F5.1.145  QADD

Saturating Add adds two register values, saturates the result to the 32-bit signed integer range \(-2^{31}\) to \((2^{31} - 1)\), and writes the result to the destination register. If saturation occurs, it sets PSTATE.Q to 1.

A1

\[
\begin{array}{ccccccccccccccccccc}
\hline
! = 1111 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & Rn & Rd & \langle 0 \rangle & \langle 0 \rangle & \langle 0 \rangle & 0 & 1 & 0 & 1 & \text{cond} & Rm
\end{array}
\]

**A1 variant**

QADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

\[
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE;}
\]

T1

\[
\begin{array}{ccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 14 & 13 & 12 & 11 & 8 & 7 & 6 & 5 & 4 & 3 & 0 \\
\hline
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & Rn & 1 & 1 & 1 & 1 & Rd & 1 & 0 & 0 & 0 & Rm
\end{array}
\]

**T1 variant**

QADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

\[
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE; } // \text{Armv8-A removes UNPREDICTABLE for R13}
\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>`  See Standard assembler syntax fields on page F1-7228.
- `<q>`  See Standard assembler syntax fields on page F1-7228.
- `<Rd>`  Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>`  Is the first general-purpose source register, encoded in the "Rm" field.
- `<Rn>`  Is the second general-purpose source register, encoded in the "Rn" field.

**Operation for all encodings**

if ConditionPassed() then

EncodingSpecificOperations();

boolean sat;
(R[d], sat) = SignedSatQ(SInt(R[m]) + SInt(R[n]), 32);
if sat then
    PSTATE.Q = '1';
F5.1.146   QADD16

Saturating Add 16 performs two 16-bit integer additions, saturates the results to the 16-bit signed integer range \([-2^{15} \leq x \leq 2^{15} - 1]\), and writes the results to the destination register.

A1

![Instruction Format](image)

_A1 variant_

QADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

_Conversion from mnemonic to machine code:_

\[
\begin{array}{cccccccccccccccccc}
\hline
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & \text{Rn} & & \text{Rd} & [1][1][1][1][0] & 0 & 0 & 1 & \text{Rm} \\
\end{array}
\]

A1 variant

QADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

_Decode for this encoding_

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

if \(d == 15 \lor n == 15 \lor m == 15\) then UNPREDICTABLE;

T1

![Instruction Format](image)

_T1 variant_

QADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

_Decode for this encoding_

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

if \(d == 15 \lor n == 15 \lor m == 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

<
See Standard assembler syntax fields on page F1-7228.

<Rd>
Is the general-purpose destination register, encoded in the "Rd" field.

<Rn>
Is the first general-purpose source register, encoded in the "Rn" field.

<Rm>
Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

if \(\text{ConditionPassed}()\) then

\[
\text{EncodingSpecificOperations();}
\]

\[
\text{sum1} = \text{SInt}(R[n]<15:0>) + \text{SInt}(R[m]<15:0>);
\]
sum2 = \texttt{SInt(R[n]<31:16>) + SInt(R[m]<31:16>)};
R[d]<15:0> = \texttt{SignedSat(sum1, 16)};
R[d]<31:16> = \texttt{SignedSat(sum2, 16)};
F5.1.147  QADD8

Saturating Add 8 performs four 8-bit integer additions, saturates the results to the 8-bit signed integer range \(-2^7 \leq x \leq 2^7 - 1\), and writes the results to the destination register.

A1

QADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

QADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
   See Standard assembler syntax fields on page F1-7228.

<q
   See Standard assembler syntax fields on page F1-7228.

<Rd
   Is the general-purpose destination register, encoded in the "Rd" field.

<Rn
   Is the first general-purpose source register, encoded in the "Rn" field.

<Rm
   Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
sum2 = $\text{SInt}(R[n]<15:8>) + \text{SInt}(R[m]<15:8>)$
sum3 = $\text{SInt}(R[n]<23:16>) + \text{SInt}(R[m]<23:16>)$
sum4 = $\text{SInt}(R[n]<31:24>) + \text{SInt}(R[m]<31:24>)$
R[d]<7:0> = \text{SignedSat}(\text{sum1}, 8)$
R[d]<15:8> = \text{SignedSat}(\text{sum2}, 8)$
R[d]<23:16> = \text{SignedSat}(\text{sum3}, 8)$
R[d]<31:24> = \text{SignedSat}(\text{sum4}, 8)$
### F5.1.148 QASX

Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, saturates the results to the 16-bit signed integer range \(-2^{15} \leq x \leq 2^{15} - 1\), and writes the results to the destination register.

**A1**

```
| 31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4 |3 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| !=1111 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Rn | Rd | [1][1][1][1] | 0 | 0 | 1 | 1 | Rm |
```

**A1 variant**

QASX\(<c>\){<q>}{<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

\[
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE;}
\]

**T1**

```
| 15 14 13 12|11 10 9 8 7 6 5 4 |3 0 |15 14 13 12|11 8 7 6 5 4 |3 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | Rn | 1 | 1 | 1 | Rd | 0 | 0 | 1 | Rm |
```

**T1 variant**

QASX\(<c>\){<q>}{<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

\[
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields* on page F1-7228.
- `<q>` See *Standard assembler syntax fields* on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0> = SignedSat(diff, 16);
    R[d]<31:16> = SignedSat(sum, 16);
F5.1.149 QDADD

Saturating Double and Add adds a doubled register value to another register value, and writes the result to the destination register. Both the doubling and the addition have their results saturated to the 32-bit signed integer range \(-2^{31} \leq x \leq 2^{31} - 1\). If saturation occurs in either operation, it sets \(\text{PSTATE.Q}\) to 1.

A1

\[
\begin{array}{ccccccccccccccccc}
1 & = & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & Rn & Rd & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \text{cond}
\end{array}
\]

A1 variant

QDADD{<c>}{<q>}{<Rd>,} <Rm>, <Rn>

Decode for this encoding

\[
d = \text{UInt}(Rd); 
 n = \text{UInt}(Rn); 
 m = \text{UInt}(Rm);
\]

if \(d == 15 || n == 15 || m == 15\) then UNPREDICTABLE;

T1

\[
\begin{array}{cccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 14 & 13 & 12 & 11 & 8 & 7 & 6 & 5 & 4 & 3 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & Rn & Rn & Rn & Rn & Rn & 1 & 0 & 1 & 0 & 1 & \text{cond}
\end{array}
\]

T1 variant

QDADD{<c>}{<q>}{<Rd>,} <Rm>, <Rn>

Decode for this encoding

\[
d = \text{UInt}(Rd); 
 n = \text{UInt}(Rn); 
 m = \text{UInt}(Rm);
\]

if \(d == 15 || n == 15 || m == 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- \(<c>\) See Standard assembler syntax fields on page F1-7228.
- \(<q>\) See Standard assembler syntax fields on page F1-7228.
- \(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.
- \(<Rm>\) Is the first general-purpose source register, encoded in the "Rm" field.
- \(<Rn>\) Is the second general-purpose source register, encoded in the "Rn" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
    boolean sat2;
    (R[d], sat2)  = SignedSatQ(SInt(R[m]) + SInt(doubled), 32);
    if sat1 || sat2 then
        PSTATE.Q = '1';
F5.1.150 QDSUB

Saturating Double and Subtract subtracts a doubled register value from another register value, and writes the result to the destination register. Both the doubling and the subtraction have their results saturated to the 32-bit signed integer range \(-2^{31} \leq x \leq 2^{31} - 1\). If saturation occurs in either operation, it sets PSTATE.Q to 1.

A1

| [31] | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=1111 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | Rn | | Rd | (0) | (0) | (0) | 0 | 1 | 0 | 1 | Rm |

cond

A1 variant

QDSUB\(<c>{<q>} {<Rd>,} <Rm>, <Rn>\)

Decode for this encoding

\(d = \text{UInt}(Rd); n = \text{UInt}(Rn); m = \text{UInt}(Rm);\)
\(\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE;}\)

T1

<table>
<thead>
<tr>
<th>[15]</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
<th>[15]</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rd</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T1 variant

QDSUB\(<c>{<q>} {<Rd>,} <Rm>, <Rn>\)

Decode for this encoding

\(d = \text{UInt}(Rd); n = \text{UInt}(Rn); m = \text{UInt}(Rm);\)
\(\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE; } // \text{Armv8-A removes UNPREDICTABLE for R13}\)

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\) See Standard assembler syntax fields on page F1-7228.
\(<q>\) See Standard assembler syntax fields on page F1-7228.
\(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.
\(<Rm>\) Is the first general-purpose source register, encoded in the "Rm" field.
\(<Rn>\) Is the second general-purpose source register, encoded in the "Rn" field.
Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations();
   (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
   boolean sat2;
   (R[d], sat2) = SignedSatQ(SInt(R[m]) - SInt(doubled), 32);
   if sat1 || sat2 then
      PSTATE.Q = '1';
F5.1.151  QSAX

Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, saturates the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$, and writes the results to the destination register.

**A1**

\[
\begin{array}{cccccccccccc}
=1111 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & | & Rn & | & Rd & | & 1 & | & 1 & | & 1 & | & 0 & 1 & 0 & 1 & | & Rm
\end{array}
\]

**A1 variant**

QSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

if \( d == 15 \) || \( n == 15 \) || \( m == 15 \) then UNPREDICTABLE;

**T1**

\[
\begin{array}{cccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 | & 15 & 14 & 13 & 12 & 11 & 8 & 7 & 6 & 5 & 4 & 3 & 0
\end{array}
\]

**T1 variant**

QSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

if \( d == 15 \) || \( n == 15 \) || \( m == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- **<>** See Standard assembler syntax fields on page F1-7228.
- **<q>** See Standard assembler syntax fields on page F1-7228.
- **<Rd>** Is the general-purpose destination register, encoded in the "Rd" field.
- **<Rn>** Is the first general-purpose source register, encoded in the "Rn" field.
- **<Rm>** Is the second general-purpose source register, encoded in the "Rm" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    sum = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    R[d]<15:0> = SignedSat(sum, 16);
    R[d]<31:16> = SignedSat(diff, 16);
F5.1.152  QSUB

Saturating Subtract subtracts one register value from another register value, saturates the result to the 32-bit signed integer range \(-2^{31} \leq x \leq 2^{31} - 1\), and writes the result to the destination register. If saturation occurs, it sets PSTATE.Q to 1.

A1

\[
\begin{array}{cccccccccccccccc}
\hline
\text{cond} & 1 \text{=} 1 1 1 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & \text{Rn} & \text{Rd} & (0) & (0) & (0) & 0 & 1 & 0 & 1 & \text{Rm} \\
\end{array}
\]

A1 variant

QSUB\{<c>\}{<q>} \{<Rd>, \} <Rm>, <Rn>

Decode for this encoding

\[
d = \text{UInt}(\text{Rd}); \quad n = \text{UInt}(\text{Rn}); \quad m = \text{UInt}(\text{Rm});
\]

if \(d == 15 || n == 15 || m == 15\) then UNPREDICTABLE;

T1

\[
\begin{array}{cccccccccccccccc}
| & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 14 & 13 & 12 & 11 & 8 & 7 & 6 & 5 & 4 & 3 & 0 |
\hline
\text{cond} & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & \text{Rn} & 1 & 1 & 1 & 1 & \text{Rd} & 1 & 0 & 1 & 0 & \text{Rm} \\
\end{array}
\]

T1 variant

QSUB\{<c>\}{<q>} \{<Rd>, \} <Rm>, <Rn>

Decode for this encoding

\[
d = \text{UInt}(\text{Rd}); \quad n = \text{UInt}(\text{Rn}); \quad m = \text{UInt}(\text{Rm});
\]

if \(d == 15 || n == 15 || m == 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\)  See Standard assembler syntax fields on page F1-7228.

\(<q>\)  See Standard assembler syntax fields on page F1-7228.

\(<Rd>\)  Is the general-purpose destination register, encoded in the "Rd" field.

\(<Rm>\)  Is the first general-purpose source register, encoded in the "Rm" field.

\(<Rn>\)  Is the second general-purpose source register, encoded in the "Rn" field.
Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  boolean sat;
  (R[d], sat) = SignedSatQ(SInt(R[m]) - SInt(R[n]), 32);
  if sat then
    PSTATE.Q = '1';
F5.1.153 QSUB16

Saturating Subtract 16 performs two 16-bit integer subtractions, saturates the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$, and writes the results to the destination register.

**A1 variant**

QSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

if \( d = 15 \) \( || \) \( n = 15 \) \( || \) \( m = 15 \) then UNPREDICTABLE;

**T1 variant**

QSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

if \( d = 15 \) \( || \) \( n = 15 \) \( || \) \( m = 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- <c> See Standard assembler syntax fields on page F1-7228.
- <q> See Standard assembler syntax fields on page F1-7228.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
\[\text{diff2} = \text{SInt}(R[n]<31:16>) - \text{SInt}(R[m]<31:16>);\]
\[R[d]<15:0> = \text{SignedSat}(\text{diff1}, 16);\]
\[R[d]<31:16> = \text{SignedSat}(\text{diff2}, 16);\]
F5.1.154  QSUB8

Saturating Subtract 8 performs four 8-bit integer subtractions, saturates the results to the 8-bit signed integer range -27 <= x <= 27 - 1, and writes the results to the destination register.

A1

| 31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4|3 |0 |
|-----|------|--------|---|----|-----|----|----|---|---|---|---|---|
| !0 1 1 0 0 0 1 0 | Rn  | Rd  | (1)(1)(1)|1|1|1|1 | Rm |

**A1 variant**

QSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

\[
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE;}
\]

T1

| 15 14 13 12|11 10 9 8 7 6 5 4|3 |0 |15 14 13 12|11 8 7 6 5 4 3 |0 |
|---|---|---|---|---|---|---|---|---|
| 1 1 1 1 1 0 1 0 1 1 0 0 | Rn  | 1 1 1 1 | Rd | 0 0 0 1 | Rm |

**T1 variant**

QSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

\[
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>`  See *Standard assembler syntax fields* on page F1-7228.
- `<q>`  See *Standard assembler syntax fields* on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

**Operation for all encodings**

\[
\text{if ConditionPassed() then}
\]

\[
\text{EncodingSpecificOperations();}
\]

\[
diff1 = \text{SInt}(R[n]<7:0>) - \text{SInt}(R[m]<7:0>);
\]

F5-7820  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  ARM DDI 0487I.a  Non-Confidential  ID081822
diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
R[d]<7:0>   = SignedSat(diff1, 8);
R[d]<15:8>  = SignedSat(diff2, 8);
R[d]<23:16> = SignedSat(diff3, 8);
R[d]<31:24> = SignedSat(diff4, 8);
F5.1.155   RBIT

Reverse Bits reverses the bit order in a 32-bit register.

A1

| [31] 28 26 25 24|23 22 21 20|19 18 17 16|15 12|11|10|9|8|7|6|5|4|3 |0 |
| !=1111 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | Rd | (1) | (1) | (1) | (1) | 0 | 0 | 1 | 1 | Rm |

A1 variant

RBIT{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;

T1

| [15 14 13 12]|11|10|9|8|7|6|5|4|3 |0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | Rn | 1 | 1 | 1 | 1 | Rd | 1 | 1 | 0 | 0 | Rm |

T1 variant

RBIT{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

d = UInt(Rd);  m = UInt(Rm);  n = UInt(Rn);
if m != n || d == 15 || m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

CONSTRUINED UNPREDICTABLE behavior

If m != n, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: m = UInt(Rn);
- The instruction executes with the additional decode: m = UInt(Rm);
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRUINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- See Standard assembler syntax fields on page F1-7228.
<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field.
For encoding T1: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    for i = 0 to 31
        result<31-i> = R[m]<i>;
    R[d] = result;
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.156   REV

Byte-Reverse Word reverses the byte order in a 32-bit register.

**A1**

 REV{<c>}{<q>} <Rd>, <Rm>

<table>
<thead>
<tr>
<th>cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=1111, 0 1 1 0 1 0 1 1</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**A1 variant**

REV{<c>}{<q>} <Rd>, <Rm>

**Decode for this encoding**

\[ d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \]
\[ \text{if } d == 15 \quad \text{or } m == 15 \quad \text{then UNPREDICTABLE}; \]

**T1**

 REV{<c>}{<q>} <Rd>, <Rm>

<table>
<thead>
<tr>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1 0 1 0 0</td>
</tr>
<tr>
<td>Rm</td>
</tr>
</tbody>
</table>

**T1 variant**

REV{<c>}{<q>} <Rd>, <Rm>

**Decode for this encoding**

\[ d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \]

**T2**

 REV{<c>}{<q>} <Rd>, <Rm>

<table>
<thead>
<tr>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 0 1 0 0 1</td>
</tr>
<tr>
<td>Rn</td>
</tr>
<tr>
<td>1 1 1 1</td>
</tr>
<tr>
<td>Rm</td>
</tr>
</tbody>
</table>

**T2 variant**

REV{<c>}{<q>} <Rd>, <Rm>

REV{<c>}{<q>} <Rd>, <Rm> can be represented in T1

**Decode for this encoding**

\[ d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad n = \text{UInt}(Rn); \]
\[ \text{if } m != n \quad \text{or } d == 15 \quad \text{or } m == 15 \quad \text{then UNPREDICTABLE}; \quad \text{Armv8-A removes UNPREDICTABLE for R13} \]

**CONSTRAINED UNPREDICTABLE behavior**

If \( m != n \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes with the additional decode: m = UInt(Rn);
• The instruction executes with the additional decode: m = UInt(Rm);
• The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
  See Standard assembler syntax fields on page F1-7228.

\<
  See Standard assembler syntax fields on page F1-7228.

<Rd>
  Is the general-purpose destination register, encoded in the "Rd" field.

<Rm>
  For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field.
  For encoding T2: is the general-purpose source register, encoded in the "Rm" field. It must be
  encoded with an identical value in the "Rn" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  bits(32) result;
  result<31:24> = R[m]<7:0>;
  result<23:16> = R[m]<15:8>;
  result<15:8>  = R[m]<23:16>;
  result<7:0>   = R[m]<31:24>;
  R[d] = result;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.157  REV16

Byte-Reverse Packed Halfword reverses the byte order in each 16-bit halfword of a 32-bit register.

A1

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=1111 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | (1) | (1) | (1) | Rd | (1) | (1) | (1) | 1 | 0 | 1 | 1 | Rm |

cond

A1 variant

REV16{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

\[ d = \text{UInt}(Rd); \; m = \text{UInt}(Rm); \]
\[ \text{if } d == 15 \; \| \; m == 15 \; \text{then UNPREDICTABLE}; \]

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T1 variant

REV16{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

\[ d = \text{UInt}(Rd); \; m = \text{UInt}(Rm); \]

T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

| 15 | 14 | 13 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 0 | 0 | 1 | Rm |

T2 variant

REV16{<c>}.{<q>} <Rd>, <Rm> // <Rd>, <Rm> can be represented in T1
REV16{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

\[ d = \text{UInt}(Rd); \; m = \text{UInt}(Rm); \; n = \text{UInt}(Rn); \]
\[ \text{if } m != n \; \| \; d == 15 \; \| \; m == 15 \; \text{then UNPREDICTABLE}; \; // \text{Armv8-A removes UNPREDICTABLE for R13} \]

CONSTRAINED UNPREDICTABLE behavior

If \( m \neq n \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
• The instruction executes as **NOP**.
• The instruction executes with the additional decode: \( m = \text{U} \text{nt}(Rn) \).
• The instruction executes with the additional decode: \( m = \text{U} \text{nt}(Rm) \).
• The value in the destination register is **UNKNOWN**.

**Notes for all encodings**

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see Appendix K1

**Architectural Constraints on UNPREDICTABLE Behaviors.**

**Assembler symbols**

\(<c>\) See [Standard assembler syntax fields on page F1-7228](#).

\(<q>\) See [Standard assembler syntax fields on page F1-7228](#).

\(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.

\(<Rm>\) For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field.

For encoding T2: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

**Operation for all encodings**

```plaintext
if \text{ConditionPassed()} \text{ then}
    \text{EncodingSpecificOperations();}
    \text{bits(32) result;}
    \text{result}<31:24> = R[m]<23:16>;
    \text{result}<23:16> = R[m]<31:24>;
    \text{result}<15:8> = R[m]<7:0>;
    \text{result}<7:0> = R[m]<15:8>;
    R[d] = result;
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.158  REVSH

Byte-Reverse Signed Halfword reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign-extends the result to 32 bits.

A1

A1 variant

REVSH{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm);
\]
\[
\text{if } d == 15 \text{ || } m == 15 \text{ then UNPREDICTABLE;}
\]

T1

T1 variant

REVSH{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm);
\]

T2

T2 variant

REVSH{<c>}{<q>} <Rd>, <Rm> // <Rd>, <Rm> can be represented in T1

REVSH{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad n = \text{UInt}(Rn);
\]
\[
\text{if } m != n \text{ || } d == 15 \text{ || } m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

CONSTRAINED UNPREDICTABLE behavior

If \( m != n \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes with the additional decode: m = UInt(Rn);
• The instruction executes with the additional decode: m = UInt(Rm);
• The value in the destination register is UNKNOWN.

Notes for all encodings
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1
Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols
<e> See Standard assembler syntax fields on page F1-7228.
<q> See Standard assembler syntax fields on page F1-7228.
<Rd> Is the general-purpose destination register, encoded in the "Rd" field.
<Rm> For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field.
For encoding T2: is the general-purpose source register, encoded in the "Rm" field. It must be
encoded with an identical value in the "Rn" field.

Operation for all encodings
if ConditionPassed() then
  EncodingSpecificOperations();
  bits(32) result;
  result<31:8> = SignExtend(R[m]<7:0>, 24);
  result<7:0> = R[m]<15:8>;
  R[d] = result;

Operational information
If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source
or destination:
• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.159  **RFE, RFEDA, RFEDB, RFEIA, RFEIB**

Return From Exception loads two consecutive memory locations using an address in a base register:

- The word loaded from the lower address is treated as an instruction address. The PE branches to it.
- The word loaded from the higher address is used to restore PSTATE. This word must be in the format of an SPSR.

An address adjusted by the size of the data loaded can optionally be written back to the base register.

The PE checks the value of the word loaded from the higher address for an illegal return event. See *Illegal return events from AArch32 state* on page G1-8952.

RFE is UNDEFINED in Hyp mode and CONSTRAINED UNPREDICTABLE in User mode.

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
**T1 variant**

RFEDB{<c>}{<q>} <Rn>{!} // Outside or last in IT block, preferred syntax  
RFEFA{<c>}{<q>} <Rn>{!} // Outside or last in IT block, alternate syntax, Full Ascending stack

**Decode for this encoding**

```plaintext
n = UInt(Rn);  wback = (W == '1');  increment = FALSE;  wordhigher = FALSE;  
if n == 15 then UNPREDICTABLE;  
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

**T2**

```
|15| 14| 13| 12| 11| 10|  9|  8|  7|  6|  5|  4|  3|  0|15| 14| 13| 12| 11| 10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0|
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | W | 1 | Rn | (1)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0)| (0) |
```

**T2 variant**

RFE{IA}{<c>}{<q>} <Rn>{!} // Outside or last in IT block, preferred syntax  
RFEFD{<c>}{<q>} <Rn>{!} // Outside or last in IT block, alternate syntax, Full Descending stack

**Decode for this encoding**

```plaintext
n = UInt(Rn);  wback = (W == '1');  increment = TRUE;  wordhigher = FALSE;  
if n == 15 then UNPREDICTABLE;  
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

**Notes for all encodings**

For more information about the **CONstrained UNPREDICTABLE** behavior of this instruction, see Appendix K1, *Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

IA  
For encoding A1: is an optional suffix to indicate the Increment After variant.  
For encoding T2: is an optional suffix for the Increment After form.

{<c>}  
For encoding A1: see *Standard assembler syntax fields* on page F1-7228.  
{<c>} must be AL or omitted.  
For encoding T1 and T2: see *Standard assembler syntax fields* on page F1-7228.

{<q>}  
See *Standard assembler syntax fields* on page F1-7228.

{<Rn>}  
Is the general-purpose base register, encoded in the "Rn" field.

!  
The address adjusted by the size of the data loaded is written back to the base register.  
If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

RFEFA, RFEEA, RFEDD, and RFEED are pseudo-instructions for RFEDA, RFEDB, RFEA, and RFEIB respectively, referring to their use for popping data from Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

**Operation for all encodings**

```plaintext
if ConditionPassed() then  
    EncodingSpecificOperations();  
    if PSTATE.EL == EL2 then  
        UNDEFINED;  
    elsif PSTATE.EL == EL0 then  
        UNPREDICTABLE;  // UNDEFINED or NOP
```
else
    address = if increment then $R[n]$ else $R[n]$-8;
    if wordhigher then address = address+4;
    new_pc_value = MemA[address,4];
    spsr = MemA[address+4,4];
    if wback then $R[n]$ = if increment then $R[n]$+8 else $R[n]$-8;
    AArch32.ExceptionReturn(new_pc_value, spsr);

CONSTRAINED UNPREDICTABLE behavior

If $PSTATE_EL == EL0$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
F5.1.160  ROR (immediate)

Rotate Right (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

**A1**

```
MOV, shift or rotate by value variant

ROR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

**T3**

```
MOV, shift or rotate by value variant

Apply when !(imm3 == 000 && imm2 == 00).

ROR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

**Assembler symbols**

- `<>`  See Standard assembler syntax fields on page F1-7228.
- `<q>`  See Standard assembler syntax fields on page F1-7228.
- `<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133. For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For encoding T3: is the general-purpose source register, encoded in the "Rm" field.
For encoding A1: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.
For encoding T3: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field.

**Operation for all encodings**

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.
F5.1.161  ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

A1

ROR<>{<c>}{<q>} {<Rd>,} <Rm>, <Rs>

is equivalent to

MOV<>{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>

and is always the preferred disassembly.

T1

ROR<>{<c>}{<q>} {<Rdm>,} <Rdm>, <Rs>

is equivalent to

MOV<>{<c>}{<q>} <Rdm>, <Rdm>, ROR <Rs>

and is the preferred disassembly when InITBlock().

T2

Not flag setting variant

ROR<>{<c>}{<q>} W {<Rd>,} <Rm>, <Rs> // Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1

is equivalent to

MOV<>{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>

Not flag setting variant

ROR<>{<c>}{<q>} {<Rd>,} <Rm>, <Rs> // Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1

is equivalent to

MOV<>{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
and is always the preferred disassembly.

\[ \text{ROR} \{<c>\} \{<q>\} \{<Rd>,\} \ <Rm>, \ <Rs> \]

is equivalent to

\[ \text{MOV} \{<c>\} \{<q>\} \ <Rd>, \ <Rm>, \ \text{ROR} \ <Rs> \]

and is always the preferred disassembly.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields* on page F1-7228.
- `<q>` See *Standard assembler syntax fields* on page F1-7228.
- `<Rdm>` Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>` Is the first general-purpose source register, encoded in the "Rm" field.
- `<Rs>` Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field.

**Operation for all encodings**

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.
F5.1.162   RORS (immediate)

Rotate Right, setting flags (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

A1

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cond</td>
<td>S</td>
<td>imm5</td>
<td>stype</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MOV5{<c>{<q>}{<Rd>,} <Rm>, #<imm>

is equivalent to

MOV5{<c>{<q>}{<Rd>,} <Rm>, ROR #<imm>

and is always the preferred disassembly.

T3

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S | imm3 | Rd | imm2 | 1 | 1 | Rm |

MOV5, shift or rotate by value variant

Applies when !(imm3 == 000 && imm2 == 00).

RORS{<c>{<q>}{<Rd>,} <Rm>, #<imm>

is equivalent to

MOV5{<c>{<q>}{<Rd>,} <Rm>, ROR #<imm>

and is always the preferred disassembly.

Assembler symbols

See Standard assembler syntax fields on page F1-7228.
<q> See *Standard assembler syntax fields on page F1-7228.*

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T3: is the general-purpose source register, encoded in the "Rm" field.

<imm> For encoding A1: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.

For encoding T3: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field.

**Operation for all encodings**

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.
F5.1.163  RORS (register)

Rotate Right, setting flags (register) provides the value of the contents of a register rotated by a variable number of bits, and updates the condition flags based on the result. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

**A1**

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
\hline
l=1111 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
\hline
cond & S & Rd & Rs & Rm & stype \\
\hline
\end{array}
\]

**Flag setting variant**

RORS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>

and is always the preferred disassembly.

**T1**

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 3 & 2 & 0 \\
\hline
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
\hline
op & Rs & Rdm \\
\hline
\end{array}
\]

**Rotate right variant**

RORS{<q>} {<Rdm>,} <Rdm>, <Rs> // Outside IT block

is equivalent to

MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs>

and is the preferred disassembly when !InITBlock().

**T2**

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 \\
\hline
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
stype & S & Rd & Rs \\
\hline
\end{array}
\]

**Flag setting variant**

RORS.W {<Rd>,} <Rm>, <Rs> // Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
and is always the preferred disassembly.

RORS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>

and is always the preferred disassembly.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields on page F1-7228*.
- `<q>` See *Standard assembler syntax fields on page F1-7228*.
- `<Rdm>` Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>` Is the first general-purpose source register, encoded in the "Rm" field.
- `<Rs>` Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field.

**Operation for all encodings**

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.
F5.1.164 RRX

Rotate Right with Extend provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31].

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

A1

<table>
<thead>
<tr>
<th>[31] 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12</th>
<th>11</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111</td>
<td>0 0 0 1 1 0 1 0</td>
<td>0</td>
<td>0 0 0 1 1 0</td>
<td>0</td>
</tr>
<tr>
<td>cond</td>
<td>S</td>
<td>Rd</td>
<td>imm5</td>
<td>stype</td>
</tr>
</tbody>
</table>

**MOV, rotate right with extend variant**

RRX{<c>}{<q>} {<Rd>,} <Rm>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, RRX

and is always the preferred disassembly.

T3

<table>
<thead>
<tr>
<th>[15] 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15 14 12</th>
<th>11</th>
<th>8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 0 0 1</td>
<td>0</td>
<td>0 1 1 1 0</td>
<td>0 0 0</td>
<td>Rd</td>
<td>0 0 1 1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>imm3</td>
<td>imm2</td>
<td>stype</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MOV, rotate right with extend variant**

RRX{<c>}{<q>} {<Rd>,} <Rm>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, RRX

and is always the preferred disassembly.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields on page F1-7228*.
- `<q>` See *Standard assembler syntax fields on page F1-7228*.
- `<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
  For encoding T3: is the general-purpose destination register, encoded in the "Rm" field.
- `<Rm>` For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
  For encoding T3: is the general-purpose source register, encoded in the "Rm" field.
**Operation for all encodings**

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.
### F5.1.165 RRXS

Rotate Right with Extend, setting flags provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31].

If the destination register is not the PC, this instruction updates the condition flags based on the result, and bit[0] is shifted into the Carry flag.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is an alias of the MOV, MOVS (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

#### A1

<table>
<thead>
<tr>
<th>[31] 28 27 26 25 24</th>
<th>[23 22 21 20</th>
<th>[19 18 17 16</th>
<th>[15 12</th>
<th>11</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111</td>
<td>0 0 0 1 1</td>
<td>0 1 1</td>
<td>[0] [0] [0]</td>
<td></td>
<td>Rd 0 0 0 0</td>
<td>1 1 0</td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td>S</td>
<td>imm5</td>
<td>stype</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### MOVES, rotate right with extend variant

RRXS{<c>}{<q>}{<Rd>,}<Rm>

is equivalent to

MOVS{<c>}{<q>}{<Rd>,}<Rm>, RRX

and is always the preferred disassembly.

#### T3

| [15 14 13 12 | [11 10 9 | 8 7 6 5 | 4 | 3 2 1 0 | [15 14 | [12 | 11 | 8 7 6 5 | 4 | 3 | 0 |
|---------------|----------|----------|----------|-------|----------|----------|-------|----------|----------|-------|----------|----------|-------|------|
| 1 1 1 0 1 0 1 | 0 1 0 0 1 | 0 0 1 1 1 | 1 | 1 1 1 1 | [0] | 0 0 | Rd 0 0 0 1 | 1 |        |
| S              | imm3     | imm2     | stype    |

### MOVES, rotate right with extend variant

RRXS{<c>}{<q>}{<Rd>,}<Rm>

is equivalent to

MOVS{<c>}{<q>}{<Rd>,}<Rm>, RRX

and is always the preferred disassembly.

### Assembler symbols

<

See Standard assembler syntax fields on page F1-7228.
See *Standard assembler syntax fields* on page F1-7228.

<\texttt{Rd}>

For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores \texttt{PSTATE} from \texttt{SPSR_{<current\_mode>}}.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.

<\texttt{Rm}>

For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T3: is the general-purpose source register, encoded in the "Rm" field.

**Operation for all encodings**

The description of \texttt{MOV, MOVS (register)} gives the operational pseudocode for this instruction.
F5.1.166 RSB, RSBS (immediate)

Reverse Subtract (immediate) subtracts a register value from an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the RSBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSB variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- The RSBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_{current_mode}.
  - The PE checks SPSR_{current_mode} for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

<table>
<thead>
<tr>
<th>[31] 28 27 26 25 24 23 22 21 20 19 16 15 12 11</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1=1111</td>
<td>S</td>
<td>Rn</td>
</tr>
<tr>
<td>cond</td>
<td>Rd</td>
<td>imm12</td>
</tr>
</tbody>
</table>

**RSB variant**

Applies when $S == 0$.

$\text{RSB}{\{c\}}{\{q\}}{\{<Rd>,\}}{<Rn>, \#<\text{const}>}$

**RSBS variant**

Applies when $S == 1$.

$\text{RSBS}{\{c\}}{\{q\}}{\{<Rd>,\}}{<Rn>, \#<\text{const}>}$

**Decode for all variants of this encoding**

$d = \text{UInt}(Rd); n = \text{UInt}(Rn); \text{setflags} = (S == '1'); \text{imm32} = \text{A32ExpandImm}(\text{imm}12);$

**T1**

<table>
<thead>
<tr>
<th>[15] 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5</th>
<th>3 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 0 0 1 0 0 1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
</tr>
</tbody>
</table>

**T1 variant**

$\text{RSB}<c>{\{q\}}{\{<Rd>,\}}{<Rn>, \#0}$ // Inside IT block

$\text{RSBS}<q>{\{<Rd>,\}}{<Rn>, \#0}$ // Outside IT block

**Decode for this encoding**

$d = \text{UInt}(Rd); n = \text{UInt}(Rn); \text{setflags} = !\text{InITBlock}(); \text{imm32} = \text{Zeros}(32); // \text{immediate} = \#0$
T2

| 15  | 14  | 13  | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 14 | 12 | 11 | 8 | 7 | 0 |
|-----|-----|-----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1   | 1   | 1   | 0 | 1  | 1  | 0 | S | Rn | 0 | imm3 | Rd | imm8 |

**RSB variant**

Applies when \( S = 0 \).

\[
\text{RSB}\{<c>\}{<q>} \{<Rd>,\} <Rn>, #<const>
\]

**RSBS variant**

Applies when \( S = 1 \).

\[
\text{RSBS}\{<c>\}{<q>} \{<Rd>,\} <Rn>, #<const>
\]

**Decode for all variants of this encoding**

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ \text{setflags} = (S == '1'); \ \text{imm32} = \text{T32ExpandImm}(i;imm3;imm8);
\]

if \( d == 15 \) || \( n == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:
  - For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
  - For the RSBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.
- `<Rn>` For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
- `<const>` For encoding A1: an immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values.

For encoding T2: an immediate value. See Modified immediate constants in T32 instructions on page F1-7242 for the range of values.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    \( (\text{result}, \text{ncsv}) = \text{AddWithCarry}(\text{NOT}(R[n]), \text{imm32}, '1'); \)
    if \( d == 15 \) then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        \( R[d] = \text{result}; \)
        if setflags then
            PSTATE.<N,Z,C,V> = ncsv;

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.167 RSB, RSBS (register)

Reverse Subtract (register) subtracts a register value from an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the RSBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSB variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- The RSBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

RSB, rotate right with extend variant

Applies when S == 0 && imm5 == 00000 && stype == 11.

RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

RSB, shift or rotate by value variant

Applies when S == 0 && !(imm5 == 00000 && stype == 11).

RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

RSBS, rotate right with extend variant

Applies when S == 1 && imm5 == 00000 && stype == 11.

RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

RSBS, shift or rotate by value variant

Applies when S == 1 && !(imm5 == 00000 && stype == 11).

RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

\[ d = \text{UInt}(Rd); \] \[ n = \text{UInt}(Rn); \] \[ m = \text{UInt}(Rm); \] \[ \text{setflags} = (S == '1'); \] \[ (\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{imm5}); \]
T1

```
| 15 14 13 12|11 10 9 8| 7 6 5 4| 3 0 |
| 0 15 14 12|11 8 7 6 5 4| 3 | 0 |

S    Rn    _[0]_    imm3    Rd    imm2    stype    Rm
```

### RSB, rotate right with extend variant

Applies when $S == 0 \&\& \text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11$.

```assembly
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

### RSB, shift or rotate by value variant

Applies when $S == 0 \&\& !(\text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11)$.

```assembly
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

### RSBS, rotate right with extend variant

Applies when $S == 1 \&\& \text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11$.

```assembly
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

### RSBS, shift or rotate by value variant

Applies when $S == 1 \&\& !(\text{imm3} == 000 \&\& \text{imm2} == 00 \&\& \text{stype} == 11)$.

```assembly
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

### Decode for all variants of this encoding

```c
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

### Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

### Assembler symbols

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`. Arm deprecates using the PC as the destination register, but if the PC is used:
  - For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
  - For the RSBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.

- `<Rn>` For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.

For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
- LSL when stype = 00
- LSR when stype = 01
- ASR when stype = 10
- ROR when stype = 11

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.
For encoding T1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        else
            R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.168   RSB, RSBS (register-shifted register)

Reverse Subtract (register-shifted register) subtracts a register value from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

**A1**

```
 31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 8 |7 6 5 4 |3 1 0
 00 0|0 0 0 0|0 1 1 S| Rn Rd Rs 0 stype |1 Rm

Flag setting variant

Applies when S == 1.

```

RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>

**Not flag setting variant**

Applies when S == 0.

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

```
<
See Standard assembler syntax fields on page F1-7228.
```

```
<q
See Standard assembler syntax fields on page F1-7228.
```

```
<Rd>
Is the general-purpose destination register, encoded in the "Rd" field.
```

```
<Rn>
Is the first general-purpose source register, encoded in the "Rn" field.
```

```
<Rm>
Is the second general-purpose source register, encoded in the "Rm" field.
```

```
<shift>
Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

<table>
<thead>
<tr>
<th>Shift Type</th>
<th>stype</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL</td>
<td>00</td>
</tr>
<tr>
<td>LSR</td>
<td>01</td>
</tr>
<tr>
<td>ASR</td>
<td>10</td>
</tr>
<tr>
<td>ROR</td>
<td>11</td>
</tr>
</tbody>
</table>
```

```
<Rs>
Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.
```
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, '1');
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.169   RSC, RSCS (immediate)

Reverse Subtract with Carry (immediate) subtracts a register value and the value of NOT (Carry flag) from an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the RSCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- The RSCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

\[
\begin{array}{cccccccccccc}
\hline
1=111 & 0 & 0 & 0 & 1 & 1 & 1 & S & Rn & Rd & | & | & imm12 \\
\end{array}
\]

**RSC variant**

Applies when S == 0.

RSC{<c>}{<q>} {<Rd>,} <Rn>, #<const>

**RSCS variant**

Applies when S == 1.

RSCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

**Decode for all variants of this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad \text{setflags} = (S == '1'); \quad \text{imm32} = \text{A32ExpandImm}(\text{imm12});
\]

**Assembler symbols**

- See Standard assembler syntax fields on page F1-7228.
- See Standard assembler syntax fields on page F1-7228.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:
  - For the RSC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
  - For the RSCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.
- <Rn> Is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
<const> An immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values.

**Operation**

if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(NOT(R[n]), imm32, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.170   RSC, RSCS (register)

Reverse Subtract with Carry (register) subtracts a register value and the value of NOT (Carry flag) from an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the RSCS variant of the instruction updates the condition flags based on the result.

The field descriptions for \texttt{<Rd>} identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.

- The RSCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR\textunderscore_<current\_mode>.
  - The PE checks SPSR\textunderscore_<current\_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

\begin{verbatim}
A1
\end{verbatim}

\begin{verbatim}
| 31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 | 7 6 5 4 | 3 0 |
- 1=111 0 0 0 1 1 1 S Rn Rd imm5 stype 0 Rm
\end{verbatim}

\textbf{RSC, rotate right with extend variant}

Applies when $S = 0 \&\& \text{imm5} = 00000 \&\& \text{stype} = 11$.

\texttt{RSC\{<c>\}{<q>\} \{<Rd>,\} <Rn>, <Rm>, RRX}

\textbf{RSC, shift or rotate by value variant}

Applies when $S = 0 \&\! (\text{imm5} = 00000 \&\& \text{stype} = 11)$.

\texttt{RSC\{<c>\}{<q>\} \{<Rd>,\} <Rn>, <Rm> \{, <shift> #<amount>\}}

\textbf{RSCS, rotate right with extend variant}

Applies when $S = 1 \&\& \text{imm5} = 00000 \&\& \text{stype} = 11$.

\texttt{RSCS\{<c>\}{<q>\} \{<Rd>,\} <Rn>, <Rm>, RRX}

\textbf{RSCS, shift or rotate by value variant}

Applies when $S = 1 \&\! (\text{imm5} = 00000 \&\& \text{stype} = 11)$.

\texttt{RSCS\{<c>\}{<q>\} \{<Rd>,\} <Rn>, <Rm> \{, <shift> #<amount>\}}

\textbf{Decode for all variants of this encoding}

\begin{verbatim}
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1'); (shift_t, shift_n) = DecodeImmShift(stype, imm5);
\end{verbatim}

\textbf{Assembler symbols}

\begin{verbatim}<c>\end{verbatim} See Standard assembler syntax fields on page F1-7228.
<q>See Standard assembler syntax fields on page F1-7228.</q>

<r>
Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the RSC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- For the RSCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

<r>
Is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

<r>
Is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

<shift>
Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- LSL when stype = 00
- LSR when stype = 01
- ASR when stype = 10
- ROR when stype = 11

<amount>
Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.171  RSC, RSCS (register-shifted register)

Reverse Subtract (register-shifted register) subtracts a register value and the value of NOT (Carry flag) from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

```
| 31  | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=1111 | 0  | 0  | 0  | 1  | 1  | S  | Rn | Rd | Rs | 0  | stype | 1  | Rm |
```

**Flag setting variant**

Applies when $S == 1$.

RSCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>

**Not flag setting variant**

Applies when $S == 0$.

RSC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>

**Decode for all variants of this encoding**

- $d = \text{UInt}(Rd)$; $n = \text{UInt}(Rn)$; $m = \text{UInt}(Rm)$; $s = \text{UInt}(Rs)$;
- setflags = ($S == '1'$); $\text{shift}_t = \text{DecodeRegShift}(\text{stype})$;
- if $d == 15$ || $n == 15$ || $m == 15$ || $s == 15$ then UNPREDICTABLE;

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>`  See Standard assembler syntax fields on page F1-7228.
- `{<c>`  See Standard assembler syntax fields on page F1-7228.
- `<Rd>`  Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>`  Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>`  Is the second general-purpose source register, encoded in the "Rm" field.
- `<shift>`  Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
  - LSL  when $\text{stype} = 00$
  - LSR  when $\text{stype} = 01$
  - ASR  when $\text{stype} = 10$
  - ROR  when $\text{stype} = 11$
- `<Rs>`  Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.
**Operation**

if `ConditionPassed()` then
   EncodingSpecificOperations();
   shift\_n = UInt(R[s]<7:0>);
   shifted = Shift(R[m], shift\_t, shift\_n, PSTATE.C);
   (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, PSTATE.C);
   R[d] = result;
   if setflags then
      PSTATE.<N,Z,C,V> = nzcv;

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.172  SADD16

Signed Add 16 performs two 16-bit signed integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

A1

```
|  31  |  28 |  27 |  26 |  25 |  24 |  23 | 22 | 21 | 20 |  16 |  15 |  12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  0 |
|-------|-----|-----|-----|-----|-----|-----|----|----|----|-----|-----|-----|----|----|----|----|----|----|----|----|----|----|----|----|
|   111 |  0  |  1  |  1  |  0  |  0  |  0  |  1  | Rn |   | Rd | [1][1][1][1] | 0  |  0  |  0  |  1  | Rm |
| cond  |
```

A1 variant

SADD16(<c>{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rd</td>
<td>0</td>
</tr>
</tbody>
</table>
```

T1 variant

SADD16(<c>{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- `<c>`  See Standard assembler syntax fields on page F1-7228.
- `<q>`  See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
```
```plaintext
sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
R[d]<15:0>  = sum1<15:0>;
R[d]<31:16> = sum2<15:0>;
PSTATE.GE<1:0> = if sum1 >= 0 then '11' else '00';
PSTATE.GE<3:2> = if sum2 >= 0 then '11' else '00';
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.173  SADD8

Signed Add 8 performs four 8-bit signed integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

A1

```
|31 28|27|26|25|24|23|22|21|20|19|16|15|12|11|10| 9| 8| 7| 6| 5| 4| 3| 0 |
-|--------------------------|--------------------------|
| 1|1111| 0| 1| 1| 0| 0| 0| 1| | Rn | | Rd | 0| 1|1|1|1|1|0|0|0|1| | Rm |
```

A1 variant

SADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

*Decode for this encoding*

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

\[
\text{if } d == 15 \quad || \quad n == 15 \quad || \quad m == 15 \quad \text{then UNPREDICTABLE};
\]

T1

```
|15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 0|15|14|13|12|11| 8| 7| 6| 5| 4| 3| 0 |
-|--------------------------|--------------------------|
| 1| 1| 1| 1| 1| 0| 1| 0| 0| 0| | Rn | 1| 1|1|1|0|0|0|1| | Rm |
```

T1 variant

SADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

*Decode for this encoding*

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

\[
\text{if } d == 15 \quad || \quad n == 15 \quad || \quad m == 15 \quad \text{then UNPREDICTABLE}; \quad // \text{Armv8-A removes UNPREDICTABLE for R13}
\]

*Notes for all encodings*

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors.*

*Assembler symbols*

- `<c>`  See *Standard assembler syntax fields on page F1-7228*.
- `<q>`  See *Standard assembler syntax fields on page F1-7228*.
- `<Rd>`  Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>`  Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>`  Is the second general-purpose source register, encoded in the "Rm" field.

*Operation for all encodings*

\[
\text{if ConditionPassed() then}
\]

\[
\text{EncodingSpecificOperations();}
\]

\[
\text{sum1} = \text{SInt}[R[n]<7:0>] + \text{SInt}[R[m]<7:0>];
\]
sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
R[d]<7:0>   = sum1<7:0>;
R[d]<15:8>  = sum2<7:0>;
R[d]<23:16> = sum3<7:0>;
R[d]<31:24> = sum4<7:0>;
PSTATE.GE<0> = if sum1 >= 0 then '1' else '0';
PSTATE.GE<1> = if sum2 >= 0 then '1' else '0';
PSTATE.GE<2> = if sum3 >= 0 then '1' else '0';
PSTATE.GE<3> = if sum4 >= 0 then '1' else '0';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.174   SASX

Signed Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, and writes the results to the destination register. It sets PSTATE.GE according to the results.

A1

\[ |31| 28|27|26|25|24|23|22|21|20|19|16|15|12|11|10|9|8|7|6|5|4|3|0 | \]

\[ \text{cond} 1111 \]

\[ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ Rn \]

\[ Rd \ (1) \ 1 | 1 | 1 | 1 | 0 \ 0 \ 1 \ 0 \]

\[ !=1111 \ 01100 \ 001 \ Rm \]

**A1 variant**

SASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

*Decode for this encoding*

\[ d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \]

\[ \text{if } d == 15 \ | \ n == 15 \ | \ m == 15 \ \text{then UNPREDICTABLE;} \]

T1

\[ |15| 14|13|12|11|10|9|8|7|6|5|4|3|0 | \]

\[ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ Rn \]

\[ 1 \ 1 \ 1 \ 1 \ Rd \ 0 \ 0 \ 0 \ 0 \]

\[ Rm \]

**T1 variant**

SASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

*Decode for this encoding*

\[ d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \]

\[ \text{if } d == 15 \ | \ n == 15 \ | \ m == 15 \ \text{then UNPREDICTABLE;} \ \text{// Armv8-A removes UNPREDICTABLE for R13} \]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

- \(<c>\)  See *Standard assembler syntax fields* on page F1-7228.
- \(<q>\)  See *Standard assembler syntax fields* on page F1-7228.
- \(<Rd>\)  Is the general-purpose destination register, encoded in the "Rd" field.
- \(<Rn>\)  Is the first general-purpose source register, encoded in the "Rn" field.
- \(<Rm>\)  Is the second general-purpose source register, encoded in the "Rm" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SIInt(R[n]<15:0>) - SIInt(R[m]<31:16>);
    sum  = SIInt(R[n]<31:16>) + SIInt(R[m]<15:0>);
    R[d]<15:0>  = diff<15:0>;
    R[d]<31:16> = sum<15:0>;
    PSTATE.GE<1:0> = if diff >= 0 then '11' else '00';
    PSTATE.GE<3:2> = if sum  >= 0 then '11' else '00';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.175  SB

Speculation Barrier is a barrier that controls speculation.

The semantics of the Speculation Barrier are that the execution, until the barrier completes, of any instruction that appears later in the program order than the barrier:

- Cannot be performed speculatively to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception.

In particular, any instruction that appears later in the program order than the barrier cannot cause a speculative allocation into any caching structure where the allocation of that entry could be indicative of any data value present in memory or in the registers.

The SB instruction:

- Cannot be speculatively executed as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception. The potentially exception generating instruction can complete once it is known not to be speculative, and all data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

When the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after an uncompleted SB instruction, the SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched.

A1

(FEAT_SB)

\[\begin{array}{cccccccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & (1) & (1) & (1) & (1) & (1) & (1) & (0) & (0) & (0) & (0) & 0 & 1 & 1 & (0) & (0) & (0)
\end{array}\]

A1 variant

SB{<q>}

Decode for this encoding

// No additional decoding required

T1

(FEAT_SB)

\[\begin{array}{cccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & (1) & (1) & (1) & (1) & (1) & (1) & (0) & (0) & (0) & (0) & 0 & 1 & 1 & (0) & (0) & (0)
\end{array}\]

T1 variant

SB{<q>}

---

T32 and A32 Base Instruction Set Instruction Descriptions
F5.1 Alphabetical list of T32 and A32 base instruction set instructions

Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Decode for this encoding

if InITBlock() then UNPREDICTABLE;

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    SpeculationBarrier();
F5.1.176   SBC, SBCS (immediate)

Subtract with Carry (immediate) subtracts an immediate value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SBCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

•   The SBC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.

•   The SBCS variant of the instruction performs an exception return without the use of the stack. In this case:
  —   The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  —   The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  —   The instruction is UNDEFINED in Hyp mode.
  —   The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

| 31  | 28 | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 16  | 15  | 12  | 11  | | | 0 |
|-----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|---|---|
| 1=1111 | 0  | 0   | 0   | 1   | 1   | 0   | S   | Rn  | Rd  | imm12 |  |

**SBC variant**

Applies when \( S = 0 \).

\[ \text{SBC} \{<c>\}\{<q>\} \{<Rd>,\} <Rn>, \#<\text{const}> \]

**SBCS variant**

Applies when \( S = 1 \).

\[ \text{SBCS} \{<c>\}\{<q>\} \{<Rd>,\} <Rn>, \#<\text{const}> \]

**Decode for all variants of this encoding**

\[ d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad \text{setflags} = (S == '1'); \quad \text{imm32} = \text{A32ExpandImm}(\text{imm12}); \]

T1

| 15  | 14  | 13  | 12 | 11  | 10  | 9  | 8  | 7  | 6  | 5  | 4 | 3 | | | 15 | | 12 | 11 | 8  | 7  | | 0  |
|-----|-----|-----|----|-----|-----|----|---|---|---|---|---|---|---|---|----|----|----|----|---|---|---|
| 1   | 1   | 1   | 0  | 1   | 0   | 1   | 1 | 0 | S | Rn | 0 | imm3 | Rd | imm8 |

**SBC variant**

Applies when \( S = 0 \).

\[ \text{SBC} \{<c>\}\{<q>\} \{<Rd>,\} <Rn>, \#<\text{const}> \]

**SBCS variant**

Applies when \( S = 1 \).
SBCS\{<c>\}{<q>\} {<Rd>,} <Rn>, #<const>

**Decode for all variants of this encoding**

\[d = \text{UInt}(Rd);\ \ n = \text{UInt}(Rn);\ \ \text{setflags} = (S == '1');\ \ \text{imm32} = \text{T32ExpandImm}(i:imm3:imm8);
\]

\[\text{if } d == 15 || n == 15 \text{ then UNPREDICTABLE;} // \text{Armv8-A removes UNPREDICTABLE for R13}\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- **<c>** See Standard assembler syntax fields on page F1-7228.
- **<q>** See Standard assembler syntax fields on page F1-7228.
- **<Rd>** For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:
  - For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
  - For the SBCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

- **<Rn>** For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

- **<const>** For encoding A1: an immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values.

For encoding T1: an immediate value. See Modified immediate constants in T32 instructions on page F1-7242 for the range of values.

**Operation for all encodings**

\[\text{if } \text{ConditionPassed}() \text{ then}\]
\[\text{EncodingSpecificOperations();}\]
\[(\text{result}, \text{nzcv}) = \text{AddWithCarry}(R[n], \text{NOT(imm32), PSTATE.C});\]
\[\text{if } d == 15 \text{ then} \quad // \text{Can only occur for A32 encoding}\]
\[\quad \text{if } \text{setflags} \text{ then}\]
\[\quad \text{ALUExceptionReturn}(\text{result});\]
\[\quad \text{else}\]
\[\quad \text{ALUWritePC}(\text{result});\]
\[\text{else}\]
\[\quad R[d] = \text{result};\]
\[\quad \text{if } \text{setflags} \text{ then}\]
\[\quad \text{PSTATE.<N,Z,C,V>} = \text{nzcv};\]
Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.177   SBC, SBCS (register)

Subtract with Carry (register) subtracts an optionally-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SBCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The SBC variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- The SBCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>imm5</td>
<td>stype</td>
<td>0</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

cond

SBC, rotate right with extend variant

Applies when S == 0 && imm5 == 00000 && stype == 11.

SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

SBC, shift or rotate by value variant

Applies when S == 0 && !(imm5 == 00000 && stype == 11).

SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

SBCS, rotate right with extend variant

Applies when S == 1 && imm5 == 00000 && stype == 11.

SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

SBCS, shift or rotate by value variant

Applies when S == 1 && !(imm5 == 00000 && stype == 11).

SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
T1

T1 variant

SBC(0)(q) {Rdn}, <Rm> // Inside IT block
SBCS(0)(q) {Rdn}, <Rm> // Outside IT block

Decode for this encoding

d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);

T2

SBC, rotate right with extend variant

Applies when \( S == 0 \) && \( \text{imm3} == 000 \) && \( \text{imm2} == 00 \) && \( \text{stype} == 11 \).

SBC(0)(q) {Rd}, <Rn>, <Rm>, RRX

SBC, shift or rotate by value variant

Applies when \( S == 0 \) && \( \neg(\text{imm3} == 000 \) && \( \text{imm2} == 00 \) && \( \text{stype} == 11 \).

SBC(0)(q) {Rd}, <Rn>, <Rm> {, <shift> #<amount>}

SBCS, rotate right with extend variant

Applies when \( S == 1 \) && \( \text{imm3} == 000 \) && \( \text{imm2} == 00 \) && \( \text{stype} == 11 \).

SBCS(0)(q) {Rd}, <Rn>, <Rm>, RRX

SBCS, shift or rotate by value variant

Applies when \( S == 1 \) && \( \neg(\text{imm3} == 000 \) && \( \text{imm2} == 00 \) && \( \text{stype} == 11 \).

SBCS(0)(q) {Rd}, <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 \| n == 15 \| m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<c>
See Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

<Rdn>
Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd>
For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- For the SBCS variant, the instruction performs an exception return, that restores PSTATE from SPSR_.current_mode.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn>
For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm>
For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift>
Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

- LSL when stype = 00
- LSR when stype = 01
- ASR when stype = 10
- ROR when stype = 11

<amount>
For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
  (result, nzcv) = AddWithCarry(R[n], NOT(shifted), PSTATE.C);
  if d == 15 then  // Can only occur for A32 encoding
    if setflags then
      ALUExceptionReturn(result);
    else
      ALUWritePC(result);
  else
    R[d] = result;
    if setflags then
      PSTATE.<N,Z,C,V> = nzcv;
**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
### F5.1.178 SBC, SBCS (register-shifted register)

Subtract with Carry (register-shifted register) subtracts a register-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### A1

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|
| !e111 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | S | Rn | Rd | Rs | 0 | stype | 1 | Rm |

#### Flag setting variant

Applies when $S == 1$.

SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>

#### Not flag setting variant

Applies when $S == 0$.

SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>

#### Decode for all variants of this encoding

```plaintext
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

#### Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

#### Assembler symbols

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
  - LSL when stype = 00
  - LSR when stype = 01
  - ASR when stype = 10
  - ROR when stype = 11
- `<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.179 SBFX

Signed Bit Field Extract extracts any number of adjacent bits at any position from a register, sign-extends them to 32 bits, and writes the result to the destination register.

A1

\[
\begin{array}{cccccccccc}
|31 & 28|27 26 25 24|23 22 21 20| & 16|15 & 12|11 & |7 6 5 4 | 3 & 0 |
\end{array}
\]

cond

\[
=1111 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad \text{widthm1} \quad \text{Rd} \quad \text{lsb} \quad 1 \quad 0 \quad 1 \quad \text{Rn}
\]

A1 variant

\[
\text{SBFX}\{<c>\}{<q>} \text{ <Rd>, <Rn>, #<lsb>, #<width>}
\]

Decode for this encoding

\[
d = \text{UInt}(\text{Rd}); \quad n = \text{UInt}(\text{Rn});
\]
\[
\text{lsbit} = \text{UInt}(\text{lsb}); \quad \text{widthminus1} = \text{UInt}(\text{widthm1});
\]
\[
\text{msbit} = \text{lsbit} + \text{widthminus1};
\]
\[
\text{if } d == 15 || n == 15 \text{ then UNPREDICTABLE;}
\]
\[
\text{if msbit > 31 then UNPREDICTABLE;}
\]

CONSTRAINED UNPREDICTABLE behavior

If msbit > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

\[
\begin{array}{cccccccccc}
|15 \quad 14 \quad 13 \quad 12|11 \quad 10 \quad 9 \quad 8 \quad 7 \quad 6 \quad 5 \quad 4 \quad 3 \quad 0 |
\end{array}
\]

\[
|1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 |1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 | \quad \text{Rn} \quad 0 \quad \text{imm3} \quad \text{Rd} \quad \text{imm2}{0} \quad \text{widthm1}
\]

T1 variant

\[
\text{SBFX}\{<c>\}{<q>} \text{ <Rd>, <Rn>, #<lsb>, #<width>}
\]

Decode for this encoding

\[
d = \text{UInt}(\text{Rd}); \quad n = \text{UInt}(\text{Rn});
\]
\[
\text{lsbit} = \text{UInt}(\text{imm3:imm2}); \quad \text{widthminus1} = \text{UInt}(\text{widthm1});
\]
\[
\text{msbit} = \text{lsbit} + \text{widthminus1};
\]
\[
\text{if } d == 15 || n == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]
\[
\text{if msbit > 31 then UNPREDICTABLE;}
\]

CONSTRAINED UNPREDICTABLE behavior

If msbit > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
The value in the destination register is UNKNOWN.

Notes for all encodings
For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols
• \(<c>\) See Standard assembler syntax fields on page F1-7228.
• \(<q>\) See Standard assembler syntax fields on page F1-7228.
• \(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.
• \(<Rn>\) Is the general-purpose source register, encoded in the "Rn" field.
• \(<lsb>\) For encoding A1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "lsb" field.
  For encoding T1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field.
• \(<width>\) Is the width of the field, in the range 1 to 32-\(<lsb>\), encoded in the "widthm1" field as \(<width>-1.

Operation for all encodings
if ConditionPassed() then
  EncodingSpecificOperations();
  R[d] = SignExtend(R[n]<msbit:lsbit>, 32);

Operational information
If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.180  SDIV

Signed Divide divides a 32-bit signed integer register value by a 32-bit signed integer register value, and writes the result to the destination register. The condition flags are not affected.

A1

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
\hline
\text{cond} & !=1111 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & \text{Rd} & (1)(1)(1)(1) & \text{Rm} & 0 & 0 & 0 & 1 & \text{Rn} & \text{Ra} \\
\hline
\end{array}
\]

A1 variant

SDIV\{<c>\}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

\[
d = \text{UInt(Rd)}; \ n = \text{UInt(Rn)}; \ m = \text{UInt(Rm)}; \ a = \text{UInt(Ra)};
if \ d == 15 \ || \ n == 15 \ || \ m == 15 \ || \ a != 15 \ then \ \text{UNPREDICTABLE};
\]

CONSTRAINED UNPREDICTABLE behavior

If Ra != '1111', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes as described, and the register specified by Ra becomes UNKNOWN.

T1

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
\hline
1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & \text{Rn} & (1)(1)(1)(1) & \text{Rd} & 1 & 1 & 1 & 1 & \text{Rm} & \text{Ra} \\
\hline
\end{array}
\]

T1 variant

SDIV\{<c>\}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

\[
d = \text{UInt(Rd)}; \ n = \text{UInt(Rn)}; \ m = \text{UInt(Rm)}; \ a = \text{UInt(Ra)};
if \ d == 15 \ || \ n == 15 \ || \ m == 15 \ || \ a != 15 \ then \ \text{UNPREDICTABLE}; \ // \text{Armv8-A removes UNPREDICTABLE for R13}
\]

CONSTRAINED UNPREDICTABLE behavior

If Ra != '1111', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes as described, and the register specified by Ra becomes UNKNOWN.
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register holding the dividend, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register holding the divisor, encoded in the "Rm" field.

Overflow

If the signed integer division \(0x80000000 / 0xFFFFFFFF\) is performed, the pseudocode produces the intermediate integer result \(+2^{31}\), that overflows the 32-bit signed integer range. No indication of this overflow case is produced, and the 32-bit result written to \(<Rd>\) must be the bottom 32 bits of the binary representation of \(+2^{31}\). So the result of the division is \(0x80000000\).

Operation for all encodings

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    integer result;
    if SInt(R[m]) == 0 then
        result = 0;
    else
        result = RoundTowardsZero(Real(SInt(R[n])) / Real(SInt(R[m])));
    R[d] = result<31:0>;
```
F5.1.181 SEL

Select Bytes selects each byte of its result from either its first operand or its second operand, according to the values of the PSTATE.GE flags.

A1

A1 variant

SEL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

if \( d == 15 \) || \( n == 15 \) || \( m == 15 \) then UNPREDICTABLE;

T1

T1 variant

SEL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

if \( d == 15 \) || \( n == 15 \) || \( m == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- \(<c>\) See Standard assembler syntax fields on page F1-7228.
- \(<q>\) See Standard assembler syntax fields on page F1-7228.
- \(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.
- \(<Rn>\) Is the first general-purpose source register, encoded in the "Rn" field.
- \(<Rm>\) Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    \( R[d]<7:0> = \) if PSTATE.GE<0> == '1' then \( R[n]<7:0> \) else \( R[m]<7:0> \);
R[d]<15:8> = if PSTATE.GE<1> == '1' then R[n]<15:8> else R[m]<15:8>;
R[d]<23:16> = if PSTATE.GE<2> == '1' then R[n]<23:16> else R[m]<23:16>;
R[d]<31:24> = if PSTATE.GE<3> == '1' then R[n]<31:24> else R[m]<31:24>;
F5.1.182   SETEND

Set Endianness writes a new value to PSTATE.E.

A1

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8 |7 6 5 4 |3 2 1 0 |
|---|---|---|---|---|---|---|---|---|
|1 1 1 0 0 0 1 0 0 0 0 |(0)(0)(0)(0)|1 0 0 0 |0 0 0 0 0 0 E (0)(0)(0)|
```

A1 variant

SETEND(<q>) <endian_specifier> // Cannot be conditional

**Decode for this encoding**

```cpp
set_bigend = (E == '1');
```

T1

```
|15 14 13 12|11 10 9 8 |7 6 5 4 |3 2 1 0 |
|---|---|---|---|---|
|1 0 1 1 0 1 1 0 0 1 0 1 0 1 0 0 (0)(0)(0) (0)(0)(0)|
```

T1 variant

SETEND(<q>) <endian_specifier> // Not permitted in IT block

**Decode for this encoding**

```cpp
set_bigend = (E == '1');
if InITBlock() then UNPREDICTABLE;
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<endian_specifier>` Is the endianness to be selected, and the value to be set in PSTATE.E, encoded in the "E" field.
  - It can have the following values:
    - LE when E = 0
    - BE when E = 1

Operation for all encodings

EncodingSpecificOperations();
AArch32.CheckSETENDEnabled();
PSTATE.E = if set_bigend then '1' else '0';
F5.1.183 SETPAN

Set Privileged Access Never writes a new value to PSTATE.PAN.

This instruction is available only in privileged mode and it is a NOP when executed in User mode.

A1

(FEAT_PAN)

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8|7 6 5 4|3 2 1 0 |
|---|---|---|---|---|---|---|---|---|
|1|1|1|0|0|0|1|0|0|0|1|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|
```

**A1 variant**

SETPAN{<q>} #<imm> // Cannot be conditional

**Decode for this encoding**

```c
if !HavePANExt() then UNDEFINED;
value = imm1;
```

T1

(FEAT_PAN)

```
|15 14 13 12|11 10 9 8|7 6 5 4|3 2 1 0 |
|---|---|---|---|---|
|1|0|1|1|0|1|1|0|0|0|0|0|0|0|0|0|
```

**T1 variant**

SETPAN{<q>} #<imm> // Not permitted in IT block

**Decode for this encoding**

```c
if InITBlock() then UNPREDICTABLE;
if !HavePANExt() then UNDEFINED;
value = imm1;
```

**Assembler symbols**

- `<q>` See *Standard assembler syntax fields on page F1-7228.*
- `<imm>` Is the unsigned immediate 0 or 1, encoded in the "imm1" field.

**Operation for all encodings**

```c
EncodingSpecificOperations();
if PSTATE.EL != EL0 then
    PSTATE.PAN = value;
```
F5.1.184   SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see *Wait For Event and Send Event* on page G1-8988.

A1

\[
\begin{array}{c}
|31| 28|27|26|25|24|23|22|21|20|19|18|17|16|15|14|13|12|11|10|9|8|7|6|5|4|3|2|1|0|
\end{array}
\]

\[!=111 0 0 1 1 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0\]

cond

**A1 variant**

SEV{<c>}{<q>}

*Decode for this encoding*

// No additional decoding required

T1

\[
\begin{array}{c}
|15|14|13|12|11|10|9|8|7|6|5|4|3|2|1|0|
\end{array}
\]

\[1 0 1 1 1 1 0 0 0 0 0 0\]

**T1 variant**

SEV{<c>}{<q>}

*Decode for this encoding*

// No additional decoding required

T2

\[
\begin{array}{c}
|15|14|13|12|11|10|9|8|7|6|5|4|3|2|1|0|
\end{array}
\]

\[1 1 1 0 0 1 1 0 1 0 [1][1][1] 1 0 0 0 0 0 0 0 0 0 0 0 0\]

**T2 variant**

SEV{<c>}.W

*Decode for this encoding*

// No additional decoding required

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors*. 
Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

&q> See Standard assembler syntax fields on page F1-7228.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    SendEvent();
F5.1.185   SEVL

Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a WFE instruction.

A1

\[
\begin{bmatrix}
\end{bmatrix}
\]

\text{cond} = 1111

A1 variant

SEVL{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T1

\[
\begin{bmatrix}
\end{bmatrix}
\]

T1 variant

SEVL{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T2

\[
\begin{bmatrix}
\end{bmatrix}
\]

T2 variant

SEVL{<c>}.W

Decode for this encoding

// No additional decoding required

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<

See Standard assembler syntax fields on page F1-7228.

>

See Standard assembler syntax fields on page F1-7228.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  SendEventLocal();
SHADD16

Signed Halving Add 16 performs two signed 16-bit integer additions, halves the results, and writes the results to the destination register.

A1

```
|31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4 3 0 |
|----|-----------|-------------|-----|----|----|----|----|----|----|----|
|1=1111 0 1 1 0 0 0 1 1 | Rn | Rd {1}{1}{1} |0 0 0 1 | Rm |
```

A1 variant

SHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

```c
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

```
|15 14 13 12|11 10 9 8 7 6 5 4 3 0 |
|-----------|---------------------|----|----|----|----|----|----|----|----|----|
|1 1 1 1 1 0 1 0 1 0 1 | Rn | Rd 0 0 1 0 | Rm |
```

T1 variant

SHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

```c
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
```

sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
R[d]<15:0> = sum1<16:1>;
R[d]<31:16> = sum2<16:1>;

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.187  SHADD8

Signed Halving Add 8 performs four signed 8-bit integer additions, halves the results, and writes the results to the destination register.

A1

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
\hline
\text{cond} & 1111 & 0 & 1 & 0 & 0 & 1 & 1 & \text{Rn} & \text{Rd} & (1) & (1) & (1) & 1 & 0 & 0 & 1 & \text{Rm} \\
\hline
\end{array}
\]

A1 variant

SHADD8{<c>}{<q>}{<Rd>,} {<Rn>,} <Rm>

Decode for this encoding

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm);
\]

if \( d == 15 \) || \( n == 15 \) || \( m == 15 \) then UNPREDICTABLE;

T1

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 14 & 13 & 12 & 11 & 8 & 7 & 6 & 5 & 4 & 3 & 0 \\
\hline
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & \text{Rn} & \text{Rd} & 1 & 1 & 1 & 0 & 0 & 1 & \text{Rm} \\
\hline
\end{array}
\]

T1 variant

SHADD8{<c>}{<q>}{<Rd>,} {<Rn>,} <Rm>

Decode for this encoding

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm);
\]

if \( d == 15 \) || \( n == 15 \) || \( m == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\)  See Standard assembler syntax fields on page F1-7228.

\(<q>\)  See Standard assembler syntax fields on page F1-7228.

\(<Rd>\)  Is the general-purpose destination register, encoded in the "Rd" field.

\(<Rn>\)  Is the first general-purpose source register, encoded in the "Rn" field.

\(<Rm>\)  Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    \[
    \text{sum1} = \text{SInt}(R[n]<7:0>) + \text{SInt}(R[m]<7:0>);
    \]
sum2 = \text{SInt}(R[n]<15:8>) + \text{SInt}(R[m]<15:8>);
sum3 = \text{SInt}(R[n]<23:16>) + \text{SInt}(R[m]<23:16>);
sum4 = \text{SInt}(R[n]<31:24>) + \text{SInt}(R[m]<31:24>);
R[d]<7:0> = \text{sum1}<8:1>;
R[d]<15:8> = \text{sum2}<8:1>;
R[d]<23:16> = \text{sum3}<8:1>;
R[d]<31:24> = \text{sum4}<8:1>;

\text{Operational information}

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.188 SHASX

Signed Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer addition and one signed 16-bit subtraction, halves the results, and writes the results to the destination register.

A1

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|------------------|
| !=1111           |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | Rd |       |
| cond             |
|                 |

A1 variant

SHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

\[
d = \text{U} \text{Int}(Rd); \quad n = \text{U} \text{Int}(Rn); \quad m = \text{U} \text{Int}(Rm);
\]

if \( d == 15 \) || \( n == 15 \) || \( m == 15 \) then UNPREDICTABLE;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 14 | 13 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|------------------|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | Rd | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rm |

T1 variant

SHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

\[
d = \text{U} \text{Int}(Rd); \quad n = \text{U} \text{Int}(Rn); \quad m = \text{U} \text{Int}(Rm);
\]

if \( d == 15 \) || \( n == 15 \) || \( m == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<{}> See Standard assembler syntax fields on page F1-7228.

<{}> See Standard assembler syntax fields on page F1-7228.

<{}> Is the general-purpose destination register, encoded in the "Rd" field.

<{}> Is the first general-purpose source register, encoded in the "Rn" field.

<{}> Is the second general-purpose source register, encoded in the "Rm" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0> = diff<16:1>;
    R[d]<31:16> = sum<16:1>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.189   SHSAX

Signed Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer subtraction and one signed 16-bit addition, halves the results, and writes the results to the destination register.

A1

\[
\begin{array}{c|cccccccccccc|cccccccc}
\hline
! & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & Rn & Rd & (1) & (1) & (1) & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
\hline
\end{array}
\]

**A1 variant**

SHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

\[\text{if } d == 15 \text{ || } n == 15 \text{ || } m == 15 \text{ then UNPREDICTABLE; }
\]

T1

\[
\begin{array}{c|cccccccccccc|cccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 14 & 13 & 12 & 11 & 8 & 7 & 6 & 5 & 4 & 3 & 0 \\
\hline
1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & Rn & 1 & 1 & 1 & 1 & Rd & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
\hline
\end{array}
\]

**T1 variant**

SHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

\[\text{if } d == 15 \text{ || } n == 15 \text{ || } m == 15 \text{ then UNPREDICTABLE; } // \text{Armv8-A removes UNPREDICTABLE for R13}
\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>`  See Standard assembler syntax fields on page F1-7228.
- `<q>`  See Standard assembler syntax fields on page F1-7228.
- `<Rd>`  Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>`  Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>`  Is the second general-purpose source register, encoded in the "Rm" field.
**Operation for all encodings**

```plaintext
define if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    R[d]<15:0> = sum<16:1>;
    R[d]<31:16> = diff<16:1>;
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.190  SHSUB16

Signed Halving Subtract 16 performs two signed 16-bit integer subtractions, halves the results, and writes the results to the destination register.

A1

```
|31 28 27 26 25 24|23 22 21 20|19|16|15|12|11 10 9|8|7|6|5|4|3|0|
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|1=1111| 0| 1| 1| 0| 0| 1| 1| Rd| Rm|
cond
```

A1 variant

```sh
SHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>Rm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

T1 variant

```sh
SHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
```
diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
R[d]<15:0> = diff1<16:1>;
R[d]<31:16> = diff2<16:1>;

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.191    SHSUB8

Signed Halving Subtract 8 performs four signed 8-bit integer subtractions, halves the results, and writes the results to the destination register.

**A1**

\[
\begin{array}{cccccccccc}
| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
\hline
| 1111 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | Rn | Rd | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
\end{array}
\]

**A1 variant**

SHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

*Decode for this encoding*

\[
d = \text{UInt}(Rd);\quad n = \text{UInt}(Rn);\quad m = \text{UInt}(Rm);
\]

if \( d == 15 \) \| \( n == 15 \) \| \( m == 15 \) then UNPREDICTABLE;

**T1**

\[
\begin{array}{cccccccccc}
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
\hline
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | Rn | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | Rd |
| 0 | 0 | 0 | 0 | 0 | 0 | Rm |
\end{array}
\]

**T1 variant**

SHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

*Decode for this encoding*

\[
d = \text{UInt}(Rd);\quad n = \text{UInt}(Rn);\quad m = \text{UInt}(Rm);
\]

if \( d == 15 \) \| \( n == 15 \) \| \( m == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations();
    \[
    \text{diff1} = \text{SInt}(R[n]<7:0>) - \text{SInt}(R[m]<7:0>);
    \]
diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
R[d]<7:0> = diff1<8:1>;
R[d]<15:8> = diff2<8:1>;
R[d]<23:16> = diff3<8:1>;
R[d]<31:24> = diff4<8:1>;

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.192   SMC

Secure Monitor Call causes a Secure Monitor Call exception. For more information see Secure Monitor Call (SMC) exception on page G1-8969.

SMC is available only for software executing at EL1 or higher. It is UNDEFINED in User mode.

If the values of HCR.TSC and SCR.SCD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception that is taken to EL3. When EL3 is using AArch32 this exception is taken to Monitor mode. When EL3 is using AArch64, it is the SCR_EL3.SMD bit, rather than the SCR.SCD bit, that can change the effect of executing an SMC instruction.

If the value of HCR.TSC is 1, execution of an SMC instruction in a Non-secure EL1 mode generates an exception that is taken to EL2, regardless of the value of SCR.SCD. When EL2 is using AArch32, this is a Hyp Trap exception that is taken to Hyp mode. For more information see Traps to Hyp mode of Non-secure EL1 execution of SMC instructions on page G1-9017.

If the value of HCR.TSC is 0 and the value of SCR.SCD is 1, the SMC instruction is:

- UNDEFINED in Non-secure state.
- CONSTRAINED UNPREDICTABLE if executed in Secure state at EL1 or higher.

### A1

| [31] 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 | 0 |
|-------|
| 1=1111 | 0 0 0 1 0 1 1 0 (0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)| 0 1 1 1 | imm4 |

**cond**

#### A1 variant

SMC{<c>}{<q>} {#}<imm4>

**Decode for this encoding**

// imm4 is for assembly/disassembly only and is ignored by hardware

### T1

| [15] 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 | 0 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 | 2 | 1 | 0 |
|--------------|
| 1 1 1 1 0 1 1 1 1 1 | 1 | 1 | imm4 | 1 0 | 0 (0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)(0)|

#### T1 variant

SMC{<c>}{<q>} {#}<imm4>

**Decode for this encoding**

// imm4 is for assembly/disassembly only and is ignored by hardware
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;

#### Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

\(<c>\)  
See *Standard assembler syntax fields* on page F1-7228.

\(<q>\)  
See *Standard assembler syntax fields* on page F1-7228.

\(<\text{imm}4>\)  
Is a 4-bit unsigned immediate value, in the range 0 to 15, encoded in the "imm4" field. This is ignored by the PE. The Secure Monitor Call exception handler (Secure Monitor code) can use this value to determine what service is being requested, but Arm does not recommend this.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();

    AArch32.CheckForSMCUndefOrTrap();

    if !ELUsingAArch32(EL3) then
        if SCR_EL3.SMD == '1' then
            // SMC disabled.
            UNDEFINED;
        else
            if SCR.SCD == '1' then
                // SMC disabled
                if CurrentSecurityState() == SS_Secure then
                    // Executes either as a NOP or UNALLOCATED.
                    c = ConstrainUnpredictable(Unpredictable_SMD);
                    assert c IN {Constraint_NOP, Constraint_UNDEF};
                    if c == Constraint_NOP then EndOfInstruction();
                    UNDEFINED;
                else
                    AArch32.CheckForSMCUndefOrTrap();
                endif
            else
                if !ELUsingAArch32(EL3) then
                    AArch64.CallSecureMonitor(Zeros(16));
                else
                    AArch32.TakeSMCException();
                endif
            endif
        endif
    endif
endif
```

**CONSTRAINED UNPREDICTABLE behavior**

If SCR.SCD == '1' && CurrentSecurityState() == SS_Secure, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
F5.1.193  **SMLABB, SMLABT, SMLATB, SMLATT**

Signed Multiply Accumulate (halfwords) performs a signed multiply accumulate operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is added to a 32-bit accumulate value and the result is written to the destination register.

If overflow occurs during the addition of the accumulate value, the instruction sets $PSTATE.Q$ to 1. It is not possible for overflow to occur during the multiplication.

### A1

```
<table>
<thead>
<tr>
<th>31 28 27 26 25 24 23 22 21 20 19 16 15</th>
<th>12 11  8  7  6  5  4  3  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111</td>
<td>0 0 0 1 0 0 0 0</td>
</tr>
</tbody>
</table>
```

#### SMLABB variant

Applies when $M == 0$ && $N == 0$.

SMLABB$\{<c>\}{<q>} <Rd>, <Rn>, <Rm>, <Ra>$

#### SMLABT variant

Applies when $M == 1$ && $N == 0$.

SMLABT$\{<c>\}{<q>} <Rd>, <Rn>, <Rm>, <Ra>$

#### SMLATB variant

Applies when $M == 0$ && $N == 1$.

SMLATB$\{<c>\}{<q>} <Rd>, <Rn>, <Rm>, <Ra>$

#### SMLATT variant

Applies when $M == 1$ && $N == 1$.

SMLATT$\{<c>\}{<q>} <Rd>, <Rn>, <Rm>, <Ra>$

#### Decode for all variants of this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

### T1

```
<table>
<thead>
<tr>
<th>15 14 13 12 11 10  9  8  7  6  5  4  3  0  15 12 11</th>
<th>8  7  6  5  4  3  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1</td>
<td>0 1 0 0 0 1</td>
</tr>
</tbody>
</table>
```

#### SMLABB variant

Applies when $N == 0$ && $M == 0$.

SMLABB$\{<c>\}{<q>} <Rd>, <Rn>, <Rm>, <Ra>$
SMLABT variant
Applies when N == 0 && M == 1.
SMLABT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLATB variant
Applies when N == 1 && M == 0.
SMLATB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLATT variant
Applies when N == 1 && M == 1.
SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

if Ra == '1111' then SEE "SMULBB, SMULBT, SMULTB, SMULTT";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols
<>
See Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

<Rd>
Is the general-purpose destination register, encoded in the "Rd" field.

<Rn>
Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field.

<Rm>
Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.

<Ra>
Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation for all encodings
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2) + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        PSTATE.Q = '1';
F5.1.194 SMLAD, SMLADX

Signed Multiply Accumulate Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets PSTATE.Q to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications.

A1

<table>
<thead>
<tr>
<th>31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 8 7 6 5 4 3 0</th>
<th>Rd 0 1 1 1 0 0 0</th>
<th>Rn 0 0 0 M 1</th>
<th>Rm 0 1111 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>Ra</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SMLAD variant

Applies when M == 0.

SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLADX variant

Applies when M == 1.

SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

if Ra == '1111' then SEE "SMUAD";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra);
m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 | Rd 0 1 1 1 0 0 0 | Rn 1111 1 | Rm 0 0 0 M |
|-------------------|------------------|----------------|
| Ra                |                  |

SMLAD variant

Applies when M == 0.

SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLADX variant

Applies when M == 1.

SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

if Ra == '1111' then SEE "SMUAD";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra);
m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<\c> See Standard assembler syntax fields on page F1-7228.

<\q> See Standard assembler syntax fields on page F1-7228.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  operand2 = if m_swap then ROR(R[m],16) else R[m];
  product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
  product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
  result = product1 + product2 + SInt(R[a]);
  R[d] = result<31:0>;
  if result != SInt(result<31:0>) then // Signed overflow
    PSTATE.Q = '1';
F5.1.195 SMLAL, SMLALS

Signed Multiply Accumulate Long multiplies two signed 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

A1

```plaintext
| 31 28 27 26 25 24 23 22 21 20 19 16 15 12 11  8 | 7 6 5 4 3 |
|---------------------------------------------------|
|        1=1111 0 0 0 1 1 1 | S | RdHi | RdLo | Rm | 1 0 0 1 | Rn |
```

Flag setting variant

Applies when \( S = 1 \).

SMLALS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Not flag setting variant

Applies when \( S = 0 \).

SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

\[ dLo = \text{UInt}(RdLo); \quad dHi = \text{UInt}(RdHi); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{setflags} = (S == '1'); \]
\[ \text{if } dLo == 15 || dHi == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE}; \]
\[ \text{if } dHi == dLo \text{ then UNPREDICTABLE}; \]

CONstrained UNPREDICTABLE behavior

If \( dHi == dLo \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

```plaintext
| 15 14 13 12 11  9 8 | 7 6 5 4 3 |
|----------------------|
|          1 1 1 1 1 0 1 1 1 1 0 0 | Rn | RdLo | RdHi | 0 0 0 0 | Rm |
```

T1 variant

SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for this encoding

\[ dLo = \text{UInt}(RdLo); \quad dHi = \text{UInt}(RdHi); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{setflags} = \text{FALSE}; \]
\[ \text{if } dLo == 15 || dHi == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE}; \]
\[ \text{// Armv8-A removes UNPREDICTABLE for R13} \]
\[ \text{if } dHi == dLo \text{ then UNPREDICTABLE}; \]
**CONSTRAINED UNPREDICTABLE behavior**

If \( d_{Hi} = d_{Lo} \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The value in the destination register is **UNKNOWN**.

**Notes for all encodings**

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see Appendix K1 **Architectural Constraints on UNPREDICTABLE Behaviors**.

**Assembler symbols**

- \(<c>\) See *Standard assembler syntax fields* on page F1-7228.
- \(<q>\) See *Standard assembler syntax fields* on page F1-7228.
- \(<RdLo>\) Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
- \(<RdHi>\) Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
- \(<Rn>\) Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- \(<Rm>\) Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        PSTATE.N = result<63>;
        PSTATE.Z = IsZeroBit(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.196  SMLALBB, SMLALBT, SMLALTB, SMLALTT

Signed Multiply Accumulate Long (halfwords) multiplies two signed 16-bit values to produce a 32-bit value, and
accumulates this with a 64-bit value. The multiply acts on two signed 16-bit quantities, taken from either the bottom
or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit
product is sign-extended and accumulated with a 64-bit accumulate value.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected
if it occurs. Instead, the result wraps around modulo 2^64.

SMLALBB variant

Applies when M == 0 && N == 0.
SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALBT variant

Applies when M == 1 && N == 0.
SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALTB variant

Applies when M == 0 && N == 1.
SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALTT variant

Applies when M == 1 && N == 1.
SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.
T1

| 15 14 13 12 | 11 10 9 8 7 6 5 4 3 0 | 12 | 11 | 8 7 6 5 4 3 0 |
| 1 1 1 1 0 1 1 0 0 | Rn | RdLo | RdHi | 1 0 | N | M | Rm |

SMLALBB variant

Applies when N == 0 && M == 0.
SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALBT variant

Applies when N == 0 && M == 1.
SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALTB variant

Applies when N == 1 && M == 0.
SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALTT variant

Applies when N == 1 && M == 1.
SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

{<c>} See Standard assembler syntax fields on page F1-7228.
{<q>} See Standard assembler syntax fields on page F1-7228.
<RdLo> Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi> Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

<Rn> For encoding A1: is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field.
   For encoding T1: is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <x>), encoded in the "Rm" field.
   For encoding T1: is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <x>), encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations();
   operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
   operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
   result = SInt(operand1) * SInt(operand2) + SInt(R[dHi]:R[dLo]);
   R[dHi] = result<63:32>;
   R[dLo] = result<31:0>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.197   SMLALD, SMLALDX

Signed Multiply Accumulate Long Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

A1

SMLALD variant

Applies when M == 0.

SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALDX variant

Applies when M == 1.

SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

\[ \text{dLo} = \text{UInt}(\text{RdLo}); \quad \text{dHi} = \text{UInt}(\text{RdHi}); \quad \text{n} = \text{UInt}(\text{Rn}); \quad \text{m} = \text{UInt}(\text{Rm}); \quad \text{m\_swap} = (M == '1'); \]
\[ \text{if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE}; \]
\[ \text{if dHi == dLo then UNPREDICTABLE}; \]

CONSTRAINED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The value in the destination register is UNKNOWN.

T1

SMLALD variant

Applies when M == 0.

SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALDX variant

Applies when M == 1.
SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

\[
dLo = \text{UInt}(RdLo); \quad dHi = \text{UInt}(RdHi); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad m\_swap = (M == '1');
\]

if \( dLo == 15 \) || \( dHi == 15 \) \( n == 15 \) \( m == 15 \) then UNPREDICTABLE;

// Armv8-A removes UNPREDICTABLE for R13
if \( dHi == dLo \) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \( dHi == dLo \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

- \(<c>\) See *Standard assembler syntax fields on page F1-7228*.
- \(<q>\) See *Standard assembler syntax fields on page F1-7228*.
- \(<RdLo>\) Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
- \(<RdHi>\) Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
- \(<Rn>\) Is the first general-purpose source register, encoded in the "Rn" field.
- \(<Rm>\) Is the second general-purpose source register, encoded in the "Rm" field.

**Operation for all encodings**

if ConditionPassed() then

EncodingSpecificOperations();

operand2 = if \( m\_swap \) then XOR(R[m],16) else R[m];

product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);

product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);

result = product1 + product2 + SInt(R[dHi]:R[dLo]);

R[dHi] = result<63:32>;

R[dLo] = result<31:0>;

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.198  SMLAWB, SMLAWT

Signed Multiply Accumulate (word by halfword) performs a signed multiply accumulate operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are added to a 32-bit accumulate value and the result is written to the destination register. The bottom 16 bits of the 48-bit product are ignored.

If overflow occurs during the addition of the accumulate value, the instruction sets PSTATE.Q to 1. No overflow can occur during the multiplication.

A1

| 31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 8 7 6 5 4 3 0 |
|---------------|---------------|---------------|---------------|
| 1=1111        | 0 0 0 1 0 1 0 | Rd  Ra  Rm  1 | M 0 0  Rn    |
| cond          |

SMLAWB variant

Applies when M == 0.
SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLAWT variant

Applies when M == 1.
SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 15 12 11 8 7 6 5 4 3 0 |
|-----------------|---------------|---------------|---------------|
| 1 1 1 1 1 0 1 1 0 1 1 0 | Rd  Rm  !=1111 | Ra          |

SMLAWB variant

Applies when M == 0.
SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLAWT variant

Applies when M == 1.
SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

if Ra == '1111' then SEE "SMULWB, SMULWT";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
  See Standard assembler syntax fields on page F1-7228.

$q$
  See Standard assembler syntax fields on page F1-7228.

<Rd>
  Is the general-purpose destination register, encoded in the "Rd" field.

<Rn>
  Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm>
  Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by $<>$), encoded in the "Rm" field.

<Ra>
  Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  operand2 = if $m_{high}$ then $R[m]<31:16>$ else $R[m]<15:0>$;
  result = SInt($R[n]$) * SInt(operand2) + (SInt($R[a]$) << 16);
  $R[d]$ = result<47:16>;
  if (result >> 16) != SInt($R[d]$) then  // Signed overflow
    PSTATE.Q = '1';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.199  SMLSD, SMLSDX

Signed Multiply Subtract Dual performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets PSTATE.Q to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications or subtraction.

A1

\[
\begin{array}{cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

<
See Standard assembler syntax fields on page F1-7228.

<Rd>
Is the general-purpose destination register, encoded in the "Rd" field.

<Rn>
Is the first general-purpose source register, encoded in the "Rn" field.

<Rm>
Is the second general-purpose source register, encoded in the "Rm" field.

<Ra>
Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = (product1 - product2) + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then // Signed overflow
        PSTATE.Q = '1';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.200 SMLSLD, SMLSLDX

Signed Multiply Subtract Long Dual performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^64.

A1

| 31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 8 7 6 5 4 3 0 |
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| !=111             | 0 1 1 1 0 1 0 0   | RdHi             | RdLo             | Rm 0 1 M 1        | Rn                |
| cond               |

**SMLSLD variant**

Applies when \( M = 0 \).

\[ \text{SMLSLD} \{ <c> \} \{ <q> \} <RdLo>, <RdHi>, <Rn>, <Rm> \]

**SMLSLDX variant**

Applies when \( M = 1 \).

\[ \text{SMLSLDX} \{ <c> \} \{ <q> \} <RdLo>, <RdHi>, <Rn>, <Rm> \]

**Decode for all variants of this encoding**

\[ dLo = \text{UInt}(RdLo); \quad dHi = \text{UInt}(RdHi); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad m\_swap = (M == '1'); \]

- if \( dLo == 15 \) \| \( dHi == 15 \) \| \( n == 15 \) \| \( m == 15 \) then UNPREDICTABLE;
- if \( dHi == dLo \) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \( dHi == dLo \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 15 12 11 8 7 6 5 4 3 0 |
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| 1 1 1 1 0 1 1 1 0 1 | RdLo             | RdHi             | 1 1 0 M           | Rm                |

**SMLSLD variant**

Applies when \( M = 0 \).

\[ \text{SMLSLD} \{ <c> \} \{ <q> \} <RdLo>, <RdHi>, <Rn>, <Rm> \]

**SMLSLDX variant**

Applies when \( M = 1 \).
SMLSLDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

\[ \text{dLo} = \text{UInt}(\text{RdLo}); \quad \text{dHi} = \text{UInt}(\text{RdHi}); \quad n = \text{UInt}(\text{Rn}); \quad m = \text{UInt}(\text{Rm}); \quad m\_swap = (M == '1'); \]
\[ \text{if } \text{dLo} == 15 \quad \text{or} \quad \text{dHi} == 15 \quad \text{or} \quad n == 15 \quad \text{or} \quad m == 15 \quad \text{then UNPREDICTABLE}; \]
// Armv8-A removes UNPREDICTABLE for R13
\[ \text{if } \text{dHi} == \text{dLo} \quad \text{then UNPREDICTABLE}; \]

**CONSTRUINED UNPREDICTABLE behavior**

If \( dHi = dLo \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

**Notes for all encodings**

For more information about the CONSTRUINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- \(<c>\):
  - See *Standard assembler syntax fields on page F1-7228*.
- \(<q>\):
  - See *Standard assembler syntax fields on page F1-7228*.
- \(<RdLo>\):
  - Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
- \(<RdHi>\):
  - Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
- \(<Rn>\):
  - Is the first general-purpose source register, encoded in the "Rn" field.
- \(<Rm>\):
  - Is the second general-purpose source register, encoded in the "Rm" field.

**Operation for all encodings**

\[ \text{if ConditionPassed()} \text{ then} \]
\[ \text{EncodingSpecificOperations();} \]
\[ \text{operand2} = \text{if } m\_swap \text{ then } \text{ROTR}(R[m],16) \text{ else } R[m]; \]
\[ \text{product1} = S\text{Int}(R[n]<15:0>) \times S\text{Int}(\text{operand2}<15:0>); \]
\[ \text{product2} = S\text{Int}(R[n]<31:16>) \times S\text{Int}(\text{operand2}<31:16>); \]
\[ \text{result} = (\text{product1} - \text{product2}) + S\text{Int}(R[dHi]:R[dLo]); \]
\[ R[dHi] = \text{result}<63:32>; \]
\[ R[dLo] = \text{result}<31:0>; \]
F5.1.201 SMMLA, SMMLAR

Signed Most Significant Word Multiply Accumulate multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and adds an accumulate value.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant 0x80000000 is added to the product before the high word is extracted.

A1

```
[31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 8 7 6 5 4 3 0]
| Rd | l=1111 | Rm | 0 0 | R 1 | Rn |
```

**SMMLA variant**
Applies when \( R = 0 \).

\[ \text{SMMLA} \{<c>}\{<q>\} \quad \text{<Rd>, <Rn>, <Rm>, <Ra>} \]

**SMMLAR variant**
Applies when \( R = 1 \).

\[ \text{SMMLAR} \{<c>\}\{<q>\} \quad \text{<Rd>, <Rn>, <Rm>, <Ra>} \]

**Decode for all variants of this encoding**

- If Ra == '1111' then SEE "SMMUL";
- \( d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad a = \text{UInt}(Ra); \quad \text{round} = (R == '1'); \)

- If \( d == 15 \) \&\& \( n == 15 \) \&\& \( m == 15 \) then UNPREDICTABLE;

T1

```
[15 14 13 12 11 10 9 8 7 6 5 4 3 0 15 12 11 8 7 6 5 4 3 0]
| Rn | l=1111 | Rd | 0 0 | R 1 | Rm |
```

**SMMLA variant**
Applies when \( R = 0 \).

\[ \text{SMMLA} \{<c>\}\{<q>\} \quad \text{<Rd>, <Rn>, <Rm>, <Ra>} \]

**SMMLAR variant**
Applies when \( R = 1 \).

\[ \text{SMMLAR} \{<c>\}\{<q>\} \quad \text{<Rd>, <Rn>, <Rm>, <Ra>} \]

**Decode for all variants of this encoding**

- If Ra == '1111' then SEE "SMMUL";
- \( d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad a = \text{UInt}(Ra); \quad \text{round} = (R == '1'); \)
- If \( d == 15 \) \&\& \( n == 15 \) \&\& \( m == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    result = (SInt(R[a]) << 32) + SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
**F5.1.202 SMMLS, SMMLSR**

Signed Most Significant Word Multiply Subtract multiplies two signed 32-bit values, subtracts the result from a 32-bit accumulate value that is shifted left by 32 bits, and extracts the most significant 32 bits of the result of that subtraction.

Optionally, the instruction can specify that the result of the instruction is rounded instead of being truncated. In this case, the constant 0x80000000 is added to the result of the subtraction before the high word is extracted.

**A1**

```
|31| 28| 27| 26| 25| 24|23| 22| 21| 20|19|16|15|12|11| 8| 7| 6| 5| 4| 3| 0 |
|l=1111| 0| 1| 1| 1| 0| 1| 0| 1| Rd| Ra| Rm| 1| 1| R| 1| Rn|
```

**SMMLLS variant**

Applies when \( R == 0 \).

\text{SMMLLS\{\<c>\}\{\<q>\} <Rd>, <Rn>, <Rm>, <Ra>}

**SMMLSR variant**

Applies when \( R == 1 \).

\text{SMMLSR\{\<c>\}\{\<q>\} <Rd>, <Rn>, <Rm>, <Ra>}

**Decode for all variants of this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad a = \text{UInt}(Ra); \quad \text{round} = (R == '1');
\]

\[
\text{if } d == 15 \quad \text{||} \quad n == 15 \quad \text{||} \quad m == 15 \quad \text{||} \quad a == 15 \text{ then UNPREDICTABLE;}
\]

**T1**

```
|15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3|0  |15|12|11| 8| 7| 6| 5| 4| 3| 0 |
|1 | 1 | 1 | 1 | 0| 1| 1| 0| 1| Rd| Ra| Rm| 0 | 0 | 0 | R| Rn|
```

**SMMLLS variant**

Applies when \( R == 0 \).

\text{SMMLLS\{\<c>\}\{\<q>\} <Rd>, <Rn>, <Rm>, <Ra>}

**SMMLSR variant**

Applies when \( R == 1 \).

\text{SMMLSR\{\<c>\}\{\<q>\} <Rd>, <Rn>, <Rm>, <Ra>}

**Decode for all variants of this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad a = \text{UInt}(Ra); \quad \text{round} = (R == '1');
\]

\[
\text{if } d == 15 \quad \text{||} \quad n == 15 \quad \text{||} \quad m == 15 \quad \text{||} \quad a == 15 \text{ then UNPREDICTABLE;}
\]

// Armv8-A removes UNPREDICTABLE for R13
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

?q>
See Standard assembler syntax fields on page F1-7228.

<Rd>
Is the general-purpose destination register, encoded in the "Rd" field.

<Rn>
Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm>
Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Ra>
Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  result = (SInt(R[a]) << 32) - SInt(R[n]) * SInt(R[m]);
  if round then result = result + 0x80000000;
  R[d] = result<63:32>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.203   SMMUL, SMMULR

Signed Most Significant Word Multiply multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and writes those bits to the destination register.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant \(0x80000000\) is added to the product before the high word is extracted.

**A1**

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
|   | 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 14 | 13 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
\hline
| l=1111 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | Rd | 1 | 1 | 1 | 1 | Rm | 0 | 0 | R | 1 | Rn |
\hline
\end{array}
\]

**SMMUL variant**

Applies when \(R == 0\).

\[\text{SMMUL}\{\text{<c>}}\{\text{<q>}}\ \{\text{<Rd>}}, \text{<Rn>}, \text{<Rm}>\]

**SMMULR variant**

Applies when \(R == 1\).

\[\text{SMMULR}\{\text{<c>}}\{\text{<q>}}\ \{\text{<Rd>}}, \text{<Rn>}, \text{<Rm}>\]

**Decode for all variants of this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{round} = (R == '1');
\]

if \(d == 15 \lor n == 15 \lor m == 15\) then UNPREDICTABLE;

**T1**

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
|   | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 14 | 13 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
\hline
|   | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | Rd | 1 | 1 | 1 | 1 | Rm | 0 | 0 | R | 1 | Rn |
\hline
\end{array}
\]

**SMMUL variant**

Applies when \(R == 0\).

\[\text{SMMUL}\{\text{<c>}}\{\text{<q>}}\ \{\text{<Rd>}}, \text{<Rn>}, \text{<Rm}>\]

**SMMULR variant**

Applies when \(R == 1\).

\[\text{SMMULR}\{\text{<c>}}\{\text{<q>}}\ \{\text{<Rd>}}, \text{<Rn>}, \text{<Rm}>\]

**Decode for all variants of this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{round} = (R == '1');
\]

if \(d == 15 \lor n == 15 \lor m == 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<
> See Standard assembler syntax fields on page F1-7228.

<q>
> See Standard assembler syntax fields on page F1-7228.

<Rd>
Is the general-purpose destination register, encoded in the "Rd" field.

<Rn>
Is the first general-purpose source register, encoded in the "Rn" field.

<Rm>
Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.204  SMUAD, SMUADX

Signed Dual Multiply Add performs two signed 16 x 16-bit multiplications. It adds the products together, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets PSTATE.Q to 1 if the addition overflows. The multiplications cannot overflow.

A1

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 14 | 13 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| l=1111 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | Rd | 1 | 1 | 1 | 1 | Rm | 0 | 0 | M | 1 | Rn |

SMUAD variant
Applies when \( M == 0 \).

SMUAD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

SMUADX variant
Applies when \( M == 1 \).

SMUADX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for all variants of this encoding

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad m\text{\_swap} = (M == '1');
\]
if \( d == 15 \) || \( n == 15 \) || \( m == 15 \) then UNPREDICTABLE;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rd | 1 | 1 | 1 | 1 | Rn | 0 | 0 | 0 | M | Rm |

SMUAD variant
Applies when \( M == 0 \).

SMUAD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

SMUADX variant
Applies when \( M == 1 \).

SMUADX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for all variants of this encoding

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad m\text{\_swap} = (M == '1');
\]
if \( d == 15 \) || \( n == 15 \) || \( m == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
>  See Standard assembler syntax fields on page F1-7228.

<q>
>  See Standard assembler syntax fields on page F1-7228.

<Rd>
>  Is the general-purpose destination register, encoded in the "Rd" field.

<Rn>
>  Is the first general-purpose source register, encoded in the "Rn" field.

<Rm>
>  Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2;
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        PSTATE.Q = '1';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.205 \textbf{SMULBB, SMULBT, SMULTB, SMULTT}

Signed Multiply (halfwords) multiplies two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is written to the destination register. No overflow is possible during this instruction.

\textbf{A1}

\begin{verbatim}
|31 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12|11 8|7 6 5 4|3 0 |
|-|=1111 0 0 0 1 0 1 1 0| Rd |{0}{0}{0}{0} | Rm | 1 | M | N | 0 | Rn
|cond|
\end{verbatim}

\textit{SMULBB variant}

Applies when \(M == 0 \&\& N == 0\).

\texttt{SMULBB\{<c>{<q>}{<Rd>,} <Rn>, <Rm>}

\textit{SMULBT variant}

Applies when \(M == 1 \&\& N == 0\).

\texttt{SMULBT\{<c>{<q>}{<Rd>,} <Rn>, <Rm>}

\textit{SMULTB variant}

Applies when \(M == 0 \&\& N == 1\).

\texttt{SMULTB\{<c>{<q>}{<Rd>,} <Rn>, <Rm>}

\textit{SMULTT variant}

Applies when \(M == 1 \&\& N == 1\).

\texttt{SMULTT\{<c>{<q>}{<Rd>,} <Rn>, <Rm>}

\textit{Decode for all variants of this encoding}

\begin{verbatim}
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
n\_high = (N == '1');  m\_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
\end{verbatim}

\textbf{T1}

\begin{verbatim}
|15 14 13 12|11 10 9 8|7 6 5 4|3 0 |15 14 13 12|11 8 7 6 5 4|3 0 |
|1 1 1 1 1 0 1 1 0 0 0 1 | Rn | 1 1 1 1 | Rd | 0 0 | M | Rm |
\end{verbatim}

\textit{SMULBB variant}

Applies when \(N == 0 \&\& M == 0\).

\texttt{SMULBB\{<c>{<q>}{<Rd>,} <Rn>, <Rm>}

\textit{SMULBT variant}

Applies when \(N == 0 \&\& M == 1\).

\texttt{SMULBT\{<c>{<q>}{<Rd>,} <Rn>, <Rm>}

\textbf{F5-7928}  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential  ARM DDI 0487I.a  ID081822
**SMULTB variant**

Applies when \(N == 1 \land M == 0\).

\[
\text{SMULTB} {<c>}{<q}> {<Rd>,} <Rn>, <Rm>
\]

**SMULTT variant**

Applies when \(N == 1 \land M == 1\).

\[
\text{SMULTT} {<c>}{<q}> {<Rd>,} <Rn>, <Rm>
\]

**Decode for all variants of this encoding**

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm);
\]

\[
n_{\text{high}} = (N == '1'); \ m_{\text{high}} = (M == '1');
\]

\[
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

- \(<c>\) See *Standard assembler syntax fields on page F1-7228*.
- \(<q>\) See *Standard assembler syntax fields on page F1-7228*.
- \(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.
- \(<Rn>\) Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by \(<x>\)), encoded in the "Rn" field.
- \(<Rm>\) Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by \(<y>\)), encoded in the "Rm" field.

**Operation for all encodings**

\[
\text{if ConditionPassed()} \text{ then}
\]

\[
\text{EncodingSpecificOperations();}
\]

\[
\text{operand1} = \text{if } n_{\text{high}} \text{ then } R[n]<31:16> \text{ else } R[n]<15:0>;
\]

\[
\text{operand2} = \text{if } m_{\text{high}} \text{ then } R[m]<31:16> \text{ else } R[m]<15:0>;
\]

\[
\text{result} = \text{SInt}(\text{operand1}) \times \text{SInt}(\text{operand2});
\]

\[
R[d] = \text{result}<31:0>;
\]

// Signed overflow cannot occur

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.206  SMULL, SMULLS

Signed Multiply Long multiplies two 32-bit signed values to produce a 64-bit result.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

A1

| 31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 8 7 6 5 4 3 0 |
|---|---|---|---|---|---|---|---|---|---|
| 1=1111 | 0 0 0 | 1 1 0 | S | RdHi | RdLo | Rm | 1 0 0 1 | Rn |
| **cond** |

Flag setting variant

Applies when $S = 1$.

SMULL{<c>}{<q}> <RdLo>, <RdHi>, <Rn>, <Rm>

Not flag setting variant

Applies when $S = 0$.

SMULL{<c>}{<q}> <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```plaintext
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If $dHi = dLo$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 15 12 11 8 7 6 5 4 3 0 |
| 1 1 1 1 1 0 1 1 0 0 0 | Rn | RdLo | RdHi | 0 0 0 0 | Rm |

T1 variant

SMULL{<c>}{<q}> <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for this encoding

```plaintext
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```
CONSTRANDED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<\> See Standard assembler syntax fields on page F1-7228.
<\> See Standard assembler syntax fields on page F1-7228.
<RdLo> Is the general-purpose destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi> Is the general-purpose destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations();
   result = SInt(R[n]) * SInt(R[m]);
   R[dHi] = result<63:32>;
   R[dLo] = result<31:0>;
   if setflags then
      PSTATE.N = result<63>;
      PSTATE.Z = IsZeroBit(result<63:0>);
      // PSTATE.C, PSTATE.V unchanged

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.207 SMULWB, SMULWT

Signed Multiply (word by halfword) multiplies a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are written to the destination register. The bottom 16 bits of the 48-bit product are ignored. No overflow is possible during this instruction.

A1

| 31 28|27 26 25 24|23 22 21 20|19 16|15 14 13 12|11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|---|---|---|---|----|---|---|---|---|---|---|---|---|---|---|
| l=1|111| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | Rd | (0) | (0) | (0) | Rm | 1 | M | 1 | 0 | Rn |

SMULWB variant
Applies when \( M = 0 \).

\[
\text{SMULWB}\{<c>\}{<q>} \{<Rd>,\} <Rn>, <Rm>
\]

SMULWT variant
Applies when \( M = 1 \).

\[
\text{SMULWT}\{<c>\}{<q>} \{<Rd>,\} <Rn>, <Rm>
\]

Decode for all variants of this encoding
\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \ m\text{high} = (M = \textquoteleft 1\textquoteright); \\
\text{if} \ d = 15 \ | \ n = 15 \ | \ m = 15 \ \text{then UNPREDICTABLE;}
\]

T1

| 15 14 13 12|11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | Rn | 1 | 1 | 1 | Rd | 0 | 0 | 0 | M | Rm |

SMULWB variant
Applies when \( M = 0 \).

\[
\text{SMULWB}\{<c>\}{<q>} \{<Rd>,\} <Rn>, <Rm>
\]

SMULWT variant
Applies when \( M = 1 \).

\[
\text{SMULWT}\{<c>\}{<q>} \{<Rd>,\} <Rn>, <Rm>
\]

Decode for all variants of this encoding
\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \ m\text{high} = (M = \textquoteleft 1\textquoteright); \\
\text{if} \ d = 15 \ | \ n = 15 \ | \ m = 15 \ \text{then UNPREDICTABLE;} \ // \text{Armv8-A removes UNPREDICTABLE for R13}
\]

Notes for all encodings
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

<
See Standard assembler syntax fields on page F1-7228.

<Rd>
Is the general-purpose destination register, encoded in the "Rd" field.

<Rn>
Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm>
Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    product = SInt(R[n]) * SInt(operand2);
    R[d] = product<47:16>;
    // Signed overflow cannot occur

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.208   SMUSD, SMUSDX

Signed Multiply Subtract Dual performs two signed 16 x 16-bit multiplications. It subtracts one of the products from
the other, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This
produces top x bottom and bottom x top multiplication.

Overflow cannot occur.

A1

|cond| 31 | 28|27|26|25|24|23|22|21|20|19|16|15|14|13|12|11| 8| 7| 6| 5| 4| 3| 0 | Rd | M | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | Rm |

**SMUSD variant**

Applies when M == 0.

SMUSD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**SMUSDX variant**

Applies when M == 1.

SMUSDX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for all variants of this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad m\_\text{swap} = (M == '1');
\]

if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

|cond| 31 | 28|27|26|25|24|23|22|21|20|19|16|15|14|13|12|11| 8| 7| 6| 5| 4| 3| 0 | Rd | M | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | Rn |

**SMUSD variant**

Applies when M == 0.

SMUSD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**SMUSDX variant**

Applies when M == 1.

SMUSDX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for all variants of this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad m\_\text{swap} = (M == '1');
\]

if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
Notes for all encodings

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations();
   operand2 = if m_swap then ROR(R[m],16) else R[m];
   product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
   product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
   result = product1 - product2;
   R[d] = result<31:0>;
   // Signed overflow cannot occur
F5.1.209  **SRS, SRSDA, SRSDB, SRSIA, SRSIB**

Store Return State stores the LR_<current_mode> and SPSR_<current_mode> to the stack of a specified mode. For information about memory accesses see *Memory accesses* on page F1-7233.

SRS is **UNDEFINED** in Hyp mode.

SRS is **CONSTRAINED UNPREDICTABLE** if it is executed in User or System mode, or if the specified mode is any of the following:

- Not implemented.
- A mode that Table G1-5 on page G1-8912 does not show.
- Hyp mode.
- Monitor mode, if the SRS instruction is executed in Non-secure state.

If EL3 is using AArch64 and an SRS instruction that is executed in a Secure EL1 mode specifies Monitor mode, it is trapped to EL3.

**A1**

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 9 8 |7 6 5 4 |0 |
| 1 1 1 1 |0 |P |U |1 |W |0 |1 |(1) |0 |(0) |(0) |(0) |(0) |(0) |(0) |(0) |(0) |(0) |(0) |(0) |mode |
```

**Decrement After variant**

Applies when \(P == 0 \&\& U == 0\).

SRSDA{<c}>{<q>} SP{!}, #<mode>

**Decrement Before variant**

Applies when \(P == 1 \&\& U == 0\).

SRSDB{<c}>{<q>} SP{!}, #<mode>

**Increment After variant**

Applies when \(P == 0 \&\& U == 1\).

SRS{IA}{<c}>{<q>} SP{!}, #<mode>

**Increment Before variant**

Applies when \(P == 1 \&\& U == 1\).

SRSIB{<c}>{<q>} SP{!}, #<mode>

**Decode for all variants of this encoding**

\[wback = (W == '1'); \text{ increment } = (U == '1'); \text{ wordhigher } = (P == U);\]

**T1**

```
| 15 14 13 12|11 10 9 8 |7 6 5 4 |3 2 1 0 |15 14 13 12|11 10 9 8 |7 6 5 4 |0 |
| 1 1 1 0 |1 |0 |0 |0 |W |0 |1 |(1) |0 |(0) |(1) |(1) |(0) |(0) |(0) |(0) |(0) |(0) |(0) |(0) |mode |
```
T1 variant
SRSDB[^c][^q] SP[^!], #^<mode>

Decode for this encoding
wback = (W == '1'); increment = FALSE; wordhigher = FALSE;

T2

\begin{verbatim}
1110100 110 0 0 1 1 0 0 1 1 0 1 1 1 0 9 8 7 6 5 4 1 0 9 8 7 6 5 4
\end{verbatim}

T2 variant
SRS[^c][^q] SP[^!], #^<mode>

Decode for this encoding
wback = (W == '1'); increment = TRUE; wordhigher = FALSE;

Notes for all encodings
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly SRS (T32) on page K1-11575 and SRS (A32) on page K1-11575.

Assembler symbols
IA For encoding A1: is an optional suffix to indicate the Increment After variant. For encoding T2: is an optional suffix for the Increment After form.
<< For encoding A1: see Standard assembler syntax fields on page F1-7228. << must be AL or omitted.
For encoding T1 and T2: see Standard assembler syntax fields on page F1-7228.
<< For encoding A1: see Standard assembler syntax fields on page F1-7228.
! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<< For encoding A1: see Standard assembler syntax fields on page F1-7228.

Operation for all encodings
\begin{verbatim}
if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then // UNDEFINED at EL2
      UNDEFINED;
    // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
    // to be security holes
    if PSTATE.M IN {M32_User,M32_System} then
\end{verbatim}
UNPREDICTABLE;
elsif mode == M32_Hyp then // Check for attempt to access Hyp mode SP
UNPREDICTABLE;
elsif mode == M32_Monitor then // Check for attempt to access Monitor mode SP
if !HaveEL(EL3) || CurrentSecurityState() != SS_Secure then
UNPREDICTABLE;
elsif !ELUsingAArch32(EL3) then
AArch64.MonitorModeTrap();
elsif BadMode(mode) then
UNPREDICTABLE;

base = Rmode[13,mode];
address = if increment then base else base-8;
if wordhigher then address = address+4;
MemA[address,4] = LR;
MemA[address+4,4] = SPSR[];
if wback then Rmode[13,mode] = if increment then base+8 else base-8;
else
if ConditionPassed() then
EncodingSpecificOperations();
if PSTATE.EL == EL2 then // UNDEFINED at EL2
UNDEFINED;

// Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
// to be security holes
if PSTATE.M IN {M32_User,M32_System} then
UNPREDICTABLE;
elsif mode == M32_Hyp then // Check for attempt to access Hyp mode SP
UNPREDICTABLE;
elsif mode == M32_Monitor then // Check for attempt to access Monitor mode SP
if !HaveEL(EL3) || CurrentSecurityState() != SS_Secure then
UNPREDICTABLE;
elsif !ELUsingAArch32(EL3) then
AArch64.MonitorModeTrap();
selseif BadMode(mode) then
UNPREDICTABLE;

base = Rmode[13,mode];
address = if increment then base else base-8;
if wordhigher then address = address+4;
MemA[address,4] = LR;
MemA[address+4,4] = SPSR[];
if wback then Rmode[13,mode] = if increment then base+8 else base-8;

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.M IN {M32_User,M32_System}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If mode == M32_Hyp, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If mode == M32_Monitor && (!HaveEL(EL3) || CurrentSecurityState() != SS_Secure), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
If BadMode(mode), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction stores to the stack of the mode in which it is executed.
- The instruction stores to an UNKNOWN address, and if the instruction specifies writeback then any general-purpose register that can be accessed from the current Exception level without a privilege violation becomes UNKNOWN.
F5.1.210  SSAT

Signed Saturate saturates an optionally-shifted signed value to a selectable signed range.

This instruction sets PSTATE.Q to 1 if the operation saturates.

A1

```
<table>
<thead>
<tr>
<th>31 28 27 26 25 24 23 22 21 20</th>
<th>16 15</th>
<th>12 11</th>
<th>7  6  5  4</th>
<th>3  0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1=111</td>
<td>0 1 0 1 0 1</td>
<td>sat_imm</td>
<td>Rd</td>
</tr>
</tbody>
</table>
```

*cond*

**Arithmetic shift right variant**

Applies when sh == 1.

SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>

**Logical shift left variant**

Applies when sh == 0.

SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

**Decode for all variants of this encoding**

```
d = UInt(Rd); n = UInt(Rn); saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm5);
if d == 15 || n == 15 then UNPREDICTABLE;
```

T1

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3</th>
<th>0 15 14 12</th>
<th>11 8 7 6 5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 1 1 1 0 (0) 1 1 0</td>
<td>sh 0</td>
<td>Rn 0</td>
<td>imm3</td>
</tr>
<tr>
<td></td>
<td>Rd</td>
<td>imm2 [0]</td>
<td>sat_imm</td>
<td></td>
</tr>
</tbody>
</table>
```

**Arithmetic shift right variant**

Applies when sh == 1 && !(imm3 == 000 & imm2 == 0).

SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>

**Logical shift left variant**

Applies when sh == 0.

SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

**Decode for all variants of this encoding**

```
if sh == '1' && (imm3:imm2) == '00000' then SEE "SSAT16";
d = UInt(Rd); n = UInt(Rn); saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

**Notes for all encodings**

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1

Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<imm> Is the bit position for saturation, in the range 1 to 32, encoded in the "sat_imm" field as <imm>-1.

<Rn> Is the general-purpose source register, encoded in the "Rn" field.

<amount> For encoding A1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.

For encoding A1: is the shift amount, in the range 1 to 32 encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field.

For encoding T1: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  operand = Shift(R[n], shift_t, shift_n, PSTATE.C); // PSTATE.C ignored
  (result, sat) = SignedSatQ(SInt(operand), saturate_to);
  R[d] = SignExtend(result, 32);
  if sat then
    PSTATE.Q = '1';
F5.1.211   SSAT16

Signed Saturate 16 saturates two signed 16-bit values to a selected signed range.

This instruction sets PSTATE.Q to 1 if the operation saturates.

A1

\[
\begin{array}{cccccccccccccccccccccc}
|31|28|27|26|25|24|23|22|21|20|19|16|15|12|11|10|9|8|7|6|5|4|3|0|
\end{array}
\]

\[
1=1111\ \ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 0\ \text{sat_imm}\ \ Rd\ \ (1)|(1)|(1)|0\ 0\ 1\ 1\ \ Rn\ \ cond\
\]

A1 variant

SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

Decode for this encoding

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ \text{saturate_to} = \text{UInt(sat_imm)}+1;
\]

if \(d == 15 \| n == 15\) then UNPREDICTABLE;

T1

\[
\begin{array}{cccccccccccccccccccccc}
|15|14|13|12|11|10|9|8|7|6|5|4|3|0|15|14|13|12|11|8|7|6|5|4|3|0|
\end{array}
\]

\[
1\ 1\ 1\ 1\ 0\ (0)\ 1\ 1\ 0\ 0\ 1\ 0\ \ Rn\ \ 0\ 0\ 0\ 0\ \ Rd\ \ 0\ 0\ (0)(0)\ \text{sat_imm}
\]

T1 variant

SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

Decode for this encoding

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ \text{saturate_to} = \text{UInt(sat_imm)}+1;
\]

if \(d == 15 \| n == 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

<q
See Standard assembler syntax fields on page F1-7228.

<Rd
Is the general-purpose destination register, encoded in the "Rd" field.

<imm
Is the bit position for saturation, in the range 1 to 16, encoded in the "sat_imm" field as <imm>-1.

<Rn
Is the general-purpose source register, encoded in the "Rn" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    (result1, sat1) = SignedSatQ(SInt(R[n]<15:0>), saturate_to);
    (result2, sat2) = SignedSatQ(SInt(R[n]<31:16>), saturate_to);
    R[d]<15:0> = SignExtend(result1, 16);
    R[d]<31:16> = SignExtend(result2, 16);
    if sat1 || sat2 then
        PSTATE.Q = '1';
F5.1.212   SSAX

Signed Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, and writes the results to the destination register. It sets \texttt{PSTATE.GE} according to the results.

**A1**

\[
\begin{array}{cccccccccccccccc}
& 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & \text{Rn} & \text{Rd} & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & \text{Rm} & 0
\end{array}
\]

**A1 variant**

SSAX\{\text{<c>},\} {\text{<q>},} {\text{<Rd>},} \text{<Rn>}, \text{<Rm>}

*Decode for this encoding*

\[
d = \text{UInt}(\text{Rd}); \ n = \text{UInt}(\text{Rn}); \ m = \text{UInt}(\text{Rm});
\]

if \(d = 15 \text{ or } n = 15 \text{ or } m = 15\) then UNPREDICTABLE;

**T1**

\[
\begin{array}{cccccccccccccccc}
\text{cond} & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 14 & 13 & 12 & 11 & 8 & 7 & 6 & 5 & 4 & 3 & 0 \\
& 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & \text{Rn} & \text{Rd} & 1 & 1 & 1 & 0 & 0 & 0 & 0 & \text{Rm} & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

**T1 variant**

SSAX\{\text{<c>},\} {\text{<q>},} {\text{<Rd>},} \text{<Rn>}, \text{<Rm>}

*Decode for this encoding*

\[
d = \text{UInt}(\text{Rd}); \ n = \text{UInt}(\text{Rn}); \ m = \text{UInt}(\text{Rm});
\]

if \(d = 15 \text{ or } n = 15 \text{ or } m = 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the \texttt{CONSTRAINED UNPREDICTABLE} behavior of this instruction, see Appendix K1 \textit{Architectural Constraints on UNPREDICTABLE Behaviors}.

**Assembler symbols**

\begin{itemize}
\item \texttt{<c>}  See \textit{Standard assembler syntax fields on page F1-7228}.
\item \texttt{<q>}  See \textit{Standard assembler syntax fields on page F1-7228}.
\item \texttt{<Rd>}  Is the general-purpose destination register, encoded in the "Rd" field.
\item \texttt{<Rn>}  Is the first general-purpose source register, encoded in the "Rn" field.
\item \texttt{<Rm>}  Is the second general-purpose source register, encoded in the "Rm" field.
\end{itemize}
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    sum = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    R[d]<15:0> = sum<15:0>;
    R[d]<31:16> = diff<15:0>;
    PSTATE.GE<1:0> = if sum >= 0 then '11' else '00';
    PSTATE.GE<3:2> = if diff >= 0 then '11' else '00';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.213 SSBB

Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the SSBB.

- When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order after the SSBB.

A1

A1 variant

SSBB{<q>}

Decode for this encoding

// No additional decoding required

T1

T1 variant

SSBB{<q>}

Decode for this encoding

if InITBlock() then UNPREDICTABLE;

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

See Standard assembler syntax fields on page F1-7228.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    SpeculativeStoreBypassBarrierToVA();
F5.1.214 SSUB16

Signed Subtract 16 performs two 16-bit signed integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

A1

\[
\begin{array}{ccccccccccccc}
\end{array}
\]

\[
\begin{array}{ccccccccccccc}
in=1111 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & \text{Rn} & \text{Rd} & \text{Rm} & \text{cond}
\end{array}
\]

\text{A1 variant}

SSUB16\{\text{<c}>\}{\text{<q}>} \{\text{<Rd>}, \} \text{<Rn>}, \text{<Rm>}

\text{Decode for this encoding}

\[
d = \text{UInt(Rd)}; \quad n = \text{UInt(Rn)}; \quad m = \text{UInt(Rm)};
\]

\[
\text{if} \quad d == 15 \quad \text{||} \quad n == 15 \quad \text{||} \quad m == 15 \quad \text{then UNPREDICTABLE};
\]

T1

\[
\begin{array}{ccccccccccccc}
| & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 | & 15 & 14 & 13 & 12 & 8 & 7 & 6 & 5 & 4 & 3 & 0 |
\end{array}
\]

\[
\begin{array}{ccccccccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & \text{Rn} & 1 & 1 & 1 & \text{Rd} & 0 & 0 & 0 & \text{Rm}
\end{array}
\]

\text{T1 variant}

SSUB16\{\text{<c}>\}{\text{<q}>} \{\text{<Rd>}, \} \text{<Rn>}, \text{<Rm>}

\text{Decode for this encoding}

\[
d = \text{UInt(Rd)}; \quad n = \text{UInt(Rn)}; \quad m = \text{UInt(Rm)};
\]

\[
\text{if} \quad d == 15 \quad \text{||} \quad n == 15 \quad \text{||} \quad m == 15 \quad \text{then UNPREDICTABLE}; \quad // \text{Armv8-A removes UNPREDICTABLE for R13}
\]

\text{Notes for all encodings}

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

\text{Assembler symbols}

\text{<c>} \quad \text{See Standard assembler syntax fields on page F1-7228.}

\text{<q>} \quad \text{See Standard assembler syntax fields on page F1-7228.}

\text{<Rd>} \quad \text{Is the general-purpose destination register, encoded in the "Rd" field.}

\text{<Rn>} \quad \text{Is the first general-purpose source register, encoded in the "Rn" field.}

\text{<Rm>} \quad \text{Is the second general-purpose source register, encoded in the "Rm" field.}

\text{Operation for all encodings}

\[
\text{if ConditionPassed() then}
\]

\[
\text{EncodingSpecificOperations();}
\]

\[
\text{diff}1 = \text{SInt(R[n]<15:0>) - SInt(R[m]<15:0>)};
\]
diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
R[d]<15:0> = diff1<15:0>;
R[d]<31:16> = diff2<15:0>;
PSTATE.GE<1:0> = if diff1 >= 0 then '11' else '00';
PSTATE.GE<3:2> = if diff2 >= 0 then '11' else '00';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.215  **SSUB8**

Signed Subtract 8 performs four 8-bit signed integer subtractions, and writes the results to the destination register. It sets **PSTATE.GE** according to the results of the subtractions.

**A1**

```
<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

cond
```

**A1 variant**

SSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

\[
d = \text{UInteger}(Rd); \quad n = \text{UInteger}(Rn); \quad m = \text{UInteger}(Rm);
\]

\[
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE;}
\]

**T1**

```
|  15 |  14 |  13 |  12 |  11 |  10 |  9  |  8  |  7  |  6  |  5  |  4  |  3  |  0  |  15 |  14 |  13 |  12 |  11 |  8  |  7  |  6  |  5  |  4  |  3  |  0  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | 1   | 1   | 1   | 0   | 1   | 0   | 1   | 0   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Rd  | 1   | 1   | 1   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

**T1 variant**

SSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

\[
d = \text{UInteger}(Rd); \quad n = \text{UInteger}(Rn); \quad m = \text{UInteger}(Rm);
\]

\[
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

**Notes for all encodings**

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- **<>**  See Standard assembler syntax fields on page F1-7228.
- **<>**  See Standard assembler syntax fields on page F1-7228.
- **<Rd>**  Is the general-purpose destination register, encoded in the "Rd" field.
- **<Rn>**  Is the first general-purpose source register, encoded in the "Rn" field.
- **<Rm>**  Is the second general-purpose source register, encoded in the "Rm" field.

**Operation for all encodings**

\[
\text{if } \text{ConditionPassed()} \text{ then}
\]

\[
\text{EncodingSpecificOperations();}
\]

\[
diff1 = \text{SInt}(R[n]<7:0>) - \text{SInt}(R[m]<7:0>);
\]
diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
R[d]<7:0>   = diff1<7:0>;
R[d]<15:8>  = diff2<7:0>;
R[d]<23:16> = diff3<7:0>;
R[d]<31:24> = diff4<7:0>;
PSTATE.GE<0>  = if diff1 >= 0 then '1' else '0';
PSTATE.GE<1>  = if diff2 >= 0 then '1' else '0';
PSTATE.GE<2>  = if diff3 >= 0 then '1' else '0';
PSTATE.GE<3>  = if diff4 >= 0 then '1' else '0';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.216   STC

Store data to System register calculates an address from a base register value and an immediate offset, and stores a word from the DBGDTRRXint System register to memory. It can use offset, post-indexed, pre-indexed, or unindexed addressing. For information about memory accesses, see Memory accesses on page F1-7233.

In an implementation that includes EL2, the permitted STC access to DBGDTRRXint can be trapped to Hyp mode, meaning that an attempt to execute an STC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see Trapping general Non-secure System register accesses to debug registers on page G1-9027.

For simplicity, the STC pseudocode does not show this possible trap to Hyp mode.

A1

```
<table>
<thead>
<tr>
<th>31 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=111 1 1 0</td>
<td>P</td>
<td>U</td>
<td>W</td>
<td>0</td>
<td>Rn</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**Offset variant**
Applies when \( P == 1 \) \&\& \( W == 0 \).

STC{<c>}{<q>} p14, c5, [<Rn>{, #{+/-}<imm>}]  

**Post-indexed variant**
Applies when \( P == 0 \) \&\& \( W == 1 \).

STC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>  

**Pre-indexed variant**
Applies when \( P == 1 \) \&\& \( W == 1 \).

STC{<c>}{<q>} p14, c5, [<Rn>, #{+/-}<imm>]  

**Unindexed variant**
Applies when \( P == 0 \) \&\& \( U == 1 \) \&\& \( W == 0 \).

STC{<c>}{<q>} p14, c5, [<Rn>], <option>  

**Decode for all variants of this encoding**

if \( P == '0' \) \&\& \( U == '0' \) \&\& \( W == '0' \) then UNDEFINED;  
n = UInt(Rn);  
cp = 14;  
imm32 = ZeroExtend(imm8:'00', 32);  
index = (P == '1');  
add = (U == '1');  
wbback = (W == '1');  
if n == 15 \&\& (wbback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;  

**CONSTRANGED UNPREDICTABLE behavior**
If \( n == 15 \) \&\& wbback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in Using R15 by instruction on page K1-11563.
T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 0</th>
<th>15 14 13 12</th>
<th>11 10 9 8 7 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 1 0</td>
<td>P U 0 W 0</td>
<td>Rn 0 1 0 1 1 1 0</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Offset variant**
Applies when \( P = 1 \) \&\& \( W = 0 \).

\[ \text{STC} \{<c>\}{<q>} \text{p14, c5, [<Rn>\{, #{+//-}<imm>\}} \]

**Post-indexed variant**
Applies when \( P = 0 \) \&\& \( W = 1 \).

\[ \text{STC} \{<c>\}{<q>} \text{p14, c5, [<Rn>\], #{+//-}<imm>} \]

**Pre-indexed variant**
Applies when \( P = 1 \) \&\& \( W = 1 \).

\[ \text{STC} \{<c>\}{<q>} \text{p14, c5, [<Rn>\], #{+//-}<imm>}! \]

**Unindexed variant**
Applies when \( P = 0 \) \&\& \( U = 1 \) \&\& \( W = 0 \).

\[ \text{STC} \{<c>\}{<q>} \text{p14, c5, [<Rn>\], <option> \]

**Decode for all variants of this encoding**

if \( P = '0' \) \&\& \( U = '0' \) \&\& \( W = '0' \) then UNDEFINED;

\[ n = \text{UInt}(Rn); \; \; cp = 14; \]

\[ \text{imm32} = \text{ZeroExtend}(\text{imm8}:'00', 32); \; \; \text{index} = (P == '1'); \; \; \text{add} = (U == '1'); \; \; \text{wback} = (W == '1'); \]

if \( n == 15 \) \&\& (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**
If \( n == 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in *Using R15 by instruction* on page K1-11563.

**Notes for all encodings**
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

<><
See *Standard assembler syntax fields* on page F1-7228.

<q>
See *Standard assembler syntax fields* on page F1-7228.

<Rn>
For the offset or unindexed variant: is the general-purpose base register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
For the offset, post-indexed or pre-indexed variant: is the general-purpose base register, encoded in the "Rn" field.

<option> Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field. The value of this field is ignored when executing this instruction.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when U = 0
+ when U = 1

<imm> Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];

    // System register read from DBGDTRXint.
    MemA[address,4] = AArch32.SysRegRead(cp, ThisInstr());
    if wback then R[n] = offset_addr;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.217   STL

Store-Release Word stores a word from a register to memory. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release on page E2-7186*.

For more information about support for shared memory see *Synchronization and semaphores on page E2-7211*. For information about memory accesses see *Memory accesses on page F1-7233*.

**A1**

<table>
<thead>
<tr>
<th>31 28 27 26 25 24</th>
<th>23 22 21 20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>(1) (1) (1) (1)</td>
</tr>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**A1 variant**

STL{<c>}{<q>}<Rt>, [<Rn>]

*Decode for this encoding*

\[
t = \text{UInt}(Rt); n = \text{UInt}(Rn);
if t == 15 \text{ or } n == 15 \text{ then UNPREDICTABLE;}
\]

**T1**

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3</th>
<th>0</th>
<th>15</th>
<th>12</th>
<th>11 10 9 8 7 6 5 4</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**T1 variant**

STL{<c>}{<q>}<Rt>, [<Rn>]

*Decode for this encoding*

\[
t = \text{UInt}(Rt); n = \text{UInt}(Rn);
if t == 15 \text{ or } n == 15 \text{ then UNPREDICTABLE;}
\]

**Notes for all encodings**

For more information about the CONSTRUANCED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

- <c> See *Standard assembler syntax fields on page F1-7228*.
- <q> See *Standard assembler syntax fields on page F1-7228*.
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    Mem0[address, 4] = R[t];

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.218  STLB

Store-Release Byte stores a byte from a register to memory. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release on page E2-7186.

For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For information about memory accesses see Memory accesses on page F1-7233.

A1

\[
| 31 \ 28 \ 27 \ 26 \ 25 \ 24 | 23 \ 22 \ 21 \ 20 | 19 \ 16 | 15 \ 14 \ 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
\]

\[ t=1111 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad Rn \quad (1)(1)(1)(1)(1)(1)(1)(1) 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad Rt \]

cond

A1 variant

STLB{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn);
\]

if \( t == 15 \) || \( n == 15 \) then UNPREDICTABLE;

T1

\[
| 15 \ 14 \ 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
\]

\[ 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad Rn \quad Rt \quad (1)(1)(1)(1)(1)(1)(1)(1) \]

T1 variant

STLB{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn);
\]

if \( t == 15 \) || \( n == 15 \) then UNPREDICTABLE;

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\) See Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<\text{Rt}>\) Is the general-purpose register to be transferred, encoded in the "Rt" field.

\(<\text{Rn}>\) Is the general-purpose base register, encoded in the "Rn" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    MemO[address, 1] = R[t]<7:0>;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.219   STLEX

Store-Release Exclusive Word stores a word from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release on page E2-7186. For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For information about memory accesses see Memory accesses on page F1-7233.

A1

```
|31|28|27|26|25|24|23|22|21|20|19|16|15|12|11|10|9|8|7|6|5|4|3|0|
|-------------|---------------|---------------|---------------|---------------|---------------|
|l=1111       |0              |0              |1              |1              |0              |0              |Rn         |Rd             |1       |1             |0 |0              |0 |1              |
```

cond

**A1 variant**

STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

*Decode for this encoding*

\[
d = \text{UInt}(Rd); \quad t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn);
\]
\[
\text{if } d == 15 || t == 15 || n == 15 \text{ then UNPREDICTABLE;}
\]
\[
\text{if } d == n || d == t \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( d == t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \( d == n \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

```
|15|14|13|12|11|10|9|8|7|6|5|4|3|0|15|11|10|9|8|7|6|5|4|3|0|
|-------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|李先生
**CONSTRAINED UNPREDICTABLE behavior**

If \( d == t \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The store instruction executes but the value stored is **UNKNOWN**.

If \( d == n \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction performs the store to an **UNKNOWN** address.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:
  - 0  If the operation updates memory.
  - 1  If the operation fails to update memory.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

**Aborts and alignment**

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns **TRUE**, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns **FALSE** and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
  EncodingSpecificOperations();
  address = R[n];
  if AArch32.ExclusiveMonitorsPass(address, 4) then
    Mem0[address, 4] = R[t];
    R[d] = ZeroExtend('0', 32);
```

else
    R[d] = ZeroExtend('1', 32);

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.220  STLEXB

Store-Release Exclusive Byte stores a byte from a register to memory if the executing PE has exclusive access to
the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was
performed.

The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release on page E2-7186.
For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For
information about memory accesses see Memory accesses on page F1-7233.

A1

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
| 1=1111 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | Rn | Rd | (1)(1) | 1 | 0 | 1 | 0 | 0 | 1 | Rt |

**A1 variant**

STLEXB{<c>}{<q>} <Rd>, <Rt>, [Rn]

**Decode for this encoding**

d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If d == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If d == n, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | Rn | Rt | (1)(1)(1) | 1 | 1 | 0 | 0 | Rd |

**T1 variant**

STLEXB{<c>}{<q>} <Rd>, <Rt>, [Rn]

**Decode for this encoding**

d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
**CONSTRAINED UNPREDICTABLE behavior**

If \( d == t \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The store instruction executes but the value stored is **UNKNOWN**.

If \( d == n \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction performs the store to an **UNKNOWN** address.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields* on page F1-7228.
- `<q>` See *Standard assembler syntax fields* on page F1-7228.
- `<Rd>` Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:
  - 0 If the operation updates memory.
  - 1 If the operation fails to update memory.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

**Aborts**

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Rd>` is not updated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,1) then
        MemO[address, 1] = R[t]<7:0>;
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('1', 32);
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.221  STLEXD

Store-Release Exclusive Doubleword stores a doubleword from two registers to memory if the executing PE has
exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if
no store was performed.

The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release on page E2-7186.
For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For
information about memory accesses see Memory accesses on page F1-7233.

A1

\[ \begin{array}{cccccccccccc}
\hline
!=111 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & Rn & Rd & (1)(1) & 1 & 0 & 1 & 0 & 0 & 1 & \text{Rt} \\
\end{array} \]

A1 variant

STLEXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]

Decode for this encoding

\[
d = \text{UInt}(Rd); \quad t = \text{UInt}(Rt); \quad t2 = t+1; \quad n = \text{UInt}(Rn);
\text{if } d == 15 \quad \text{||} \quad Rt<0> == \text{'}1\text{'} \quad \text{||} \quad t2 == 15 \quad \text{||} \quad n == 15 \quad \text{then UNPREDICTABLE};
\text{if } d == n \quad \text{||} \quad d == t \quad \text{||} \quad d == t2 \quad \text{then UNPREDICTABLE};
\]

CONSTRAINED UNPREDICTABLE behavior

If \( d == t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \( d == n \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

If \( Rt<0> == \text{'}1\text{'} \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: \( Rt<0> = \text{'}0\text{'} \).
- The instruction executes with the additional decode: \( t2 = t \).
- The instruction executes as described, with no change to its behavior and no additional side effects.

If \( Rt == \text{'}1110\text{'} \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction is handled as described in Using R15 by instruction on page K1-11563.
T1

|15 14 13 12|11 10 9 8 7 6 5 4 3 0|15 12|11 8 7 6 5 4 3 0|1 0 0 1 1 0 0|1 1 1|1 0 0 1 1 0 0|

T1 variant

STLEXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad t = \text{UInt}(Rt); \quad t2 = \text{UInt}(Rt2); \quad n = \text{UInt}(Rn);
\]

if \(d == 15\) || \(t == 15\) || \(t2 == 15\) || \(n == 15\) then UNPREDICTABLE;

if \(d == n\) || \(d == t\) || \(d == t2\) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \(d == t\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \(d == n\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

\(<c>\) \[See Standard assembler syntax fields on page F1-7228.\]

\(<q>\) \[See Standard assembler syntax fields on page F1-7228.\]

\(<Rd>\) Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

\(<Rt>\) For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. \(<Rt>\) must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

\(<Rt2>\) For encoding A1: is the second general-purpose register to be transferred. \(<Rt2>\) must be \(<R(t+1)>\).

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

\(<Rn>\) Is the general-purpose base register, encoded in the "Rn" field.
If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    value = if BigEndian(AccType_ORDERED) then R[t]:R[t2] else R[t2]:R[t];
    if AArch32.ExclusiveMonitorsPass(address, 8) then
        MemO[address, 8] = value;
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('1', 32);
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.222   STLEXH

Store-Release Exclusive Halfword stores a halfword from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release on page E2-7186.

For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For information about memory accesses see Memory accesses on page F1-7233.

A1

| 31  | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|
|     | 0  | 0  | 1  | 1  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |
| cond|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |

A1 variant

STLEXH(<c>){<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

\[
d = \text{UInt}(Rd); \quad t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn);
\]
\[
\text{if } d == 15 \quad \text{||} \quad t == 15 \quad \text{||} \quad n == 15 \text{ then UNPREDICTABLE;}
\]
\[
\text{if } d == n \quad \text{||} \quad d == t \text{ then UNPREDICTABLE;}
\]

CONSTRANGED UNPREDICTABLE behavior

If \(d == t\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \(d == n\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

T1 variant

STLEXH(<c>){<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

\[
d = \text{UInt}(Rd); \quad t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn);
\]
\[
\text{if } d == 15 \quad \text{||} \quad t == 15 \quad \text{||} \quad n == 15 \text{ then UNPREDICTABLE;}
\]
\[
\text{if } d == n \quad \text{||} \quad d == t \text{ then UNPREDICTABLE;}
\]
**CONSTRAINED UNPREDICTABLE behavior**

If d == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If d == n, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- <c> See Standard assembler syntax fields on page F1-7228.
- <q> See Standard assembler syntax fields on page F1-7228.
- <Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:
  - 0 If the operation updates memory.
  - 1 If the operation fails to update memory.
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.

**Aborts and alignment**

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address, 2) then
        Mem0[address, 2] = R[t]<15:0>;
        R[d] = ZeroExtend('0', 32);
```
else
  R[d] = ZeroExtend('1', 32);

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.223  STLH

Store-Release Halfword stores a halfword from a register to memory. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release on page E2-7186.

For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For information about memory accesses see Memory accesses on page F1-7233.

A1

```
|31| 28|27|26|25|24|23|22|21|20|19|16|15|14|13|12|11| 9| 8| 7| 6| 5| 4| 3| 0|
+l=1111 |0|0|0|1|1|1|0| Rn |(1)(1)(1)(1)(1) |0|0|1|0|0|1 |Rt |
    cond
```

**A1 variant**

STLH{<c>}{<q>} <Rt>, [<Rn>]

**Decode for this encoding**

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

```
|15|14|13|12|11| 9| 8| 7| 6| 5| 4| 3 |0|15|12|11| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
|1|1|1|0|1|0|0|1|1|0|0|Rn |Rt |(1)(1)(1)(1) |1|0|0|1 |(1)(1)(1) |
```

**T1 variant**

STLH{<c>}{<q>} <Rt>, [<Rn>]

**Decode for this encoding**

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>`  See Standard assembler syntax fields on page F1-7228.
- `<q>`  See Standard assembler syntax fields on page F1-7228.
- `<Rt>` Is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    Mem0[address, 2] = R[t]<15:0>;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.224   STM, STMIA, STMEA

Store Multiple (Increment After, Empty Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC on page F1-7234.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. For details of related system instructions see STM (User registers).

A1

A1 variant

STM{IA}{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
STM{IA}{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Ascending stack

Decode for this encoding

n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If n == 15 && wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in Using R15 by instruction on page K1-11563.

T1

<table>
<thead>
<tr>
<th>31 28 27 26 25 24 23 22 21 20 19 16 15</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>register_list</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=1111</td>
<td>1 0 0 0 1 0 W 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
T1 variant

STM{IA}{<c>}{<q>} <Rn>!, <registers> // Preferred syntax
STMEA{<c>}{<q>} <Rn>!, <registers> // Alternate syntax, Empty Ascending stack

Decode for this encoding

\[ n \text{ = UInt}(Rn); \text{ registers} = '00000000':\text{register\_list}; \text{ wback} = \text{TRUE}; \]
if \( \text{BitCount}(\text{registers}) < 1 \) then UNPREDICTABLE;

CONstrained UNPRedictable behavior

If \( \text{BitCount}(\text{registers}) < 1 \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If \( n = 15 \) && \( \text{wback} \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes without writeback of the base address.
• The instruction executes with writeback to the PC. The instruction is handled as described in Using R15 by instruction on page K1-11563.

T2

| I15 | I14 | I13 | I12 | I11 | I10 | I9 | I8 | I7 | I6 | I5 | I4 | I3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | 11  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | W  | 0  | Rn | 0 | M | register\_list |

T2 variant

STM{IA}{<c>}.W <Rn>!, <registers> // Preferred syntax, if \( \text{<Rn>} \), '!' and <registers> can be represented in T1
STMEA{<c>}.W <Rn>!, <registers> // Alternate syntax, Empty Ascending stack, if \( \text{<Rn>} \), '!' and <registers> can be represented in T1
STM{IA}{<c>}{<q>}.W <Rn>!, <registers> // Preferred syntax
STMEA{<c>}{<q>}.W <Rn>!, <registers> // Alternate syntax, Empty Ascending stack

Decode for this encoding

\[ n \text{ = UInt}(Rn); \text{ registers} = \text{P:M:register\_list}; \text{ wback} = (W = '1'); \]
if \( n = 15 \) || \( \text{BitCount}(\text{registers}) < 2 \) then UNPREDICTABLE;
if \( \text{wback} \) && \( \text{registers}<n> = '1' \) then UNPREDICTABLE;
if \( \text{registers}<13> = '1' \) then UNPREDICTABLE;
if \( \text{registers}<15> = '1' \) then UNPREDICTABLE;

CONstrained UNPRedictable behavior

If \( \text{BitCount}(\text{registers}) < 1 \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as \texttt{NOP}.
• The instruction operates as an \texttt{STM} with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If $\text{BitCount}(\text{registers}) = 1$, then one of the following behaviors must occur:
• The instruction is \texttt{UNDEFINED}.
• The instruction executes as \texttt{NOP}.
• The instruction executes as described, with no change to its behavior and no additional side effects.
• The instruction operates as an \texttt{STM} with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If $\text{wback} \&\& \text{registers}<n> == '1'$, then one of the following behaviors must occur:
• The instruction is \texttt{UNDEFINED}.
• The instruction executes as \texttt{NOP}.
• The store instruction executes but the value stored for the base register is \texttt{UNKNOWN}.

If $\text{registers}<13> == '1'$, then one of the following behaviors must occur:
• The instruction is \texttt{UNDEFINED}.
• The instruction executes as \texttt{NOP}.
• The store instruction performs all of the stores using the specified addressing mode but the value of R13 is \texttt{UNKNOWN}.

If $\text{registers}<15> == '1'$, then one of the following behaviors must occur:
• The instruction is \texttt{UNDEFINED}.
• The instruction executes as \texttt{NOP}.
• The store instruction performs all of the stores using the specified addressing mode but the value of R15 is \texttt{UNKNOWN}.

If $n == 15 \&\& \text{wback}$, then one of the following behaviors must occur:
• The instruction is \texttt{UNDEFINED}.
• The instruction executes as \texttt{NOP}.
• The instruction executes without writeback of the base address.
• The instruction executes with writeback to the PC. The instruction is handled as described in \textit{Using R15 by instruction on page K1-11563}.

Notes for all encodings
For more information about the \texttt{CONSTRAINED UNPREDICTABLE} behavior of this instruction, see Appendix K1 \textit{Architectural Constraints on UNPREDICTABLE Behaviors}.

Assembler symbols

\texttt{IA} is an optional suffix for the Increment After form.

$<>$ See \textit{Standard assembler syntax fields} on page F1-7228.

$q$ See \textit{Standard assembler syntax fields} on page F1-7228.
<Rn> Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers> For encoding A1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.

If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field. If the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For encoding T2: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS[address,4] = bits(32) UNKNOWN; // Only possible for encodings T1 and A1
            else
                MemS[address,4] = R[i];
                address = address + 4;
        if registers<i> == '1' then // Only possible for encoding A1
            MemS[address,4] = PCStoreValue();
        if wback then R[n] = R[n] + 4*BitCount(registers);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.225   STM (User registers)

In an EL1 mode other than System mode, Store Multiple (User registers) stores multiple User mode registers to consecutive memory locations using an address from a base register. The PE reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

Store Multiple (User registers) is UNDEFINED in Hyp mode, and CONSTRAINED UNPREDICTABLE in User or System modes.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC.

**A1**

```
<table>
<thead>
<tr>
<th>31 28 27 26 25 24 23 22 21 20 19 16 15</th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=1111</td>
<td>1 0 0</td>
<td>P</td>
<td>U</td>
</tr>
<tr>
<td>Rn</td>
<td>register_list</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**A1 variant**

STM{<amode>}{<c>}{<q>} <Rn>, <registers>^  

**Decode for this encoding**

```plaintext
n = UInt(Rn);  registers = register_list;  increment = (U == '1');  wordhigher = (P == U);  
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

**CONSTRAINED UNPREDICTABLE behavior**

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<amode>` is one of:
  - **DA**: Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.
  - **ED**: Empty Descending. For this instruction, a synonym for DA.
  - **DB**: Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.
  - **FD**: Full Descending. For this instruction, a synonym for DB.
  - **IA**: Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.
  - **EA**: Empty Ascending. For this instruction, a synonym for IA.
Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as \( P = 1, U = 1 \).

Full Ascending. For this instruction, a synonym for \( \text{IB} \).

See *Standard assembler syntax fields* on page F1-7228.

Yesterday

See *Standard assembler syntax fields* on page F1-7228.

Is the general-purpose base register, encoded in the "Rn" field.

See *Standard assembler syntax fields* on page F1-7228.

Is a list of one or more registers, separated by commas and surrounded by \{ and \}. It specifies the set of registers to be stored by the \( \text{STM} \) instruction. The registers are stored with the lowest-numbered register to the lowest memory address, through to the highest-numbered register to the highest memory address. See also *Encoding of lists of general-purpose registers and the PC* on page F1-7234.

**Operation**

\[
\text{if } \text{ConditionPassed()} \text{ then}\n\quad \text{EncodingSpecificOperations();}\n\quad \text{if } \text{PSTATE.EL} = \text{EL2} \text{ then}\n\quad \quad \text{UNDEFINED;}\n\quad \text{elsif } \text{PSTATE.M} \in \{\text{M32_User, M32_System}\} \text{ then}\n\quad \quad \text{UNPREDICTABLE;}\n\quad \text{else}\n\quad \quad \text{length} = 4 \times \text{BitCount}(\text{registers});\n\quad \quad \text{address} = \text{if increment then} \ R[n] \text{ else} \ R[n]-\text{length};\n\quad \quad \text{if wordhigher then address = address + 4;}\n\quad \quad \text{for } i = 0 \text{ to 14}\n\quad \quad \quad \text{if registers} < i > = '1' \text{ then} \ // \text{Store User mode register}\n\quad \quad \quad \quad \text{MemS}[\text{address}, 4] = R\text{mode}[i, M32_\text{User}];\n\quad \quad \quad \quad \text{address} = \text{address} + 4;\n\quad \quad \quad \text{if registers} < 15 > = '1' \text{ then}\n\quad \quad \quad \quad \text{MemS}[\text{address}, 4] = \text{PCStoreValue();}\n\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( \text{PSTATE.M} \in \{\text{M32_User, M32_System}\} \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an \( \text{STM} \) with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.226  STMDA, STMED

Store Multiple Decrement After (Empty Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC on page F1-7234.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. For details of related system instructions see STM (User registers).

A1

A1 variant

STMDA{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
STMED{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Descending stack

Decode for this encoding

n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If n == 15 && wback, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes without writeback of the base address.
• The instruction uses the addressing mode described in the equivalent immediate offset instruction.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<>{ See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.
<Rn> Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers> Is a list of one or more registers to be stored, separated by commas and surrounded by { and }.
The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.
If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

Operation

if ConditionPassed() then
  EncodingSpecificOperations();
  address = R[n] - 4*BitCount(registers) + 4;
  for i = 0 to 14
    if registers<i> == '1' then
      if i == n && wback && i != LowestSetBit(registers) then
        MemS[address,4] = bits(32) UNKNOWN;
      else
        MemS[address,4] = R[i];
        address = address + 4;
    if registers<15> == '1' then
      MemS[address,4] = PCStoreValue();
    if wback then R[n] = R[n] - 4*BitCount(registers);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.227 STMDB, STMFD

Store Multiple Decrement Before (Full Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC on page F1-7234.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. For details of related system instructions see STM (User registers).

This instruction is used by the alias PUSH (multiple registers). See Alias conditions on page F5-7982 for details of when each alias is preferred.

**A1**

```
|31 28 27 26 25 24 23 22 21 20 19 16 15 | 1 0 0 1 0 0 W 0 | Rn | register_list |
```

**A1 variant**

STMDB{<c>{<q>}}<Rn>{!}, <registers> // Preferred syntax
STMFD{<c>{<q>}}<Rn>{!}, <registers> // Alternate syntax, Full Descending stack

**Decode for this encoding**

n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;

**CONSERVED UNPREDICTABLE behavior**

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

**T1**

```
|15 14 13 12|11 10 9 8 7 6 5 4 3 | 0 |15 14 13 | 1 0 0 1 0 0 W 0 | Rn | register_list |
```

**T1 variant**

STMDB{<c>{<q>}}<Rn>{!}, <registers> // Preferred syntax
STMFD{<c>{<q>}}<Rn>{!}, <registers> // Alternate syntax, Full Descending stack

**Decode for this encoding**

n = UInt(Rn);  registers = P:M:register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<0> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '1' then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If wback && registers<n> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored for the base register is UNKNOWN.

If BitCount(registers) == 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If registers<13> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The store instruction performs all of the stores using the specified addressing mode but the value of R13 is UNKNOWN.

If registers<15> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs all of the stores using the specified addressing mode but the value of R15 is UNKNOWN.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>of variant</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH (multiple registers)</td>
<td>T1</td>
<td>W == '1' &amp; Rn == '1101' &amp; BitCount(M:register_list) &gt; 1</td>
</tr>
<tr>
<td>PUSH (multiple registers)</td>
<td>A1</td>
<td>W == '1' &amp; Rn == '1101' &amp; BitCount(register_list) &gt; 1</td>
</tr>
</tbody>
</table>

Assembler symbols

<> See Standard assembler syntax fields on page F1-7228.

<?, See Standard assembler syntax fields on page F1-7228.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers> For encoding A1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.

If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations();
   address = R[n] - 4*BitCount(registers);
   for i = 0 to 14
      if registers<i> == '1' then
         if i == n & wback & i != LowestSetBit(registers) then
            MemS[address,4] = bits(32) UNKNOWN;  // Only possible for encoding A1
         else
            MemS[address,4] = R[i];
            address = address + 4;
         if registers<i> == '1' then  // Only possible for encoding A1
            MemS[address,4] = PCStoreValue();
      if wback then R[n] = R[n] - 4*BitCount(registers);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.228  STMIB, STMFA

Store Multiple Increment Before (Full Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also Encoding of lists of general-purpose registers and the PC on page F1-7234.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see FEAT_LSMAOC. For details of related system instructions see STM (User registers).

A1

<table>
<thead>
<tr>
<th>31 28 27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16 15</th>
<th>1 0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>W</th>
<th>0</th>
<th>Rn</th>
<th>register_list</th>
</tr>
</thead>
<tbody>
<tr>
<td>=1111</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>W</td>
<td>0</td>
<td>Rn</td>
<td>register_list</td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 variant

STMIB{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
STMFA{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Ascending stack

Decode for this encoding

n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If n == 15 && wback, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes without writeback of the base address.
• The instruction uses the addressing mode described in the equivalent immediate offset instruction.

Notes for all encodings

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.
<Rn> Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers> Is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.

If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

**Operation**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS[address,4] = bits(32) UNKNOWN;
            else
                MemS[address,4] = R[i];
                address = address + 4;
            if registers<15> == '1' then
                MemS[address,4] = PCStoreValue();
            if wback then R[n] = R[n] + 4*BitCount(registers);
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.229 STR (immediate)

Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses on page F1-7233.

This instruction is used by the alias PUSH (single register). See Alias conditions on page F5-7988 for details of when each alias is preferred.

A1

<table>
<thead>
<tr>
<th>31 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>t=1111</td>
<td>0 1 0</td>
<td>P</td>
<td>U</td>
<td>0</td>
<td>W</td>
<td>0</td>
</tr>
</tbody>
</table>

**Offset variant**

Applies when \( P == 1 \) \&\& \( W == 0 \).

\[ \text{STR}\{<c>}{<q}> <Rt>, [<Rn> {, #{+/-}<imm}>] \]

**Post-indexed variant**

Applies when \( P == 0 \) \&\& \( W == 0 \).

\[ \text{STR}\{<c>}{<q}> <Rt>, [<Rn>], #{+/-}<imm> \]

**Pre-indexed variant**

Applies when \( P == 1 \) \&\& \( W == 1 \).

\[ \text{STR}\{<c>}{<q}> <Rt>, [<Rn>], #{+/-}<imm>! \]

**Decode for all variants of this encoding**

\[ \text{if } P == '0' \&\& W == '1' \text{ then SEE "STRT";} \]
\[ t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad \text{imm32} = \text{ZeroExtend}(\text{imm12}, 32); \]
\[ \text{index} = (P == '1'); \quad \text{add} = (U == '1'); \quad \text{wback} = (P == '0') \&\& (W == '1'); \]
\[ \text{if wback} \&\& (n == t) \text{ then UNPREDICTABLE;} \]

**CONSTRAINED UNPREDICTABLE behavior**

If \( \text{wback} \&\& n == t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \( \text{wback} \&\& n == 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.
**T1**

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10</th>
<th>8 7</th>
<th>6 5</th>
<th>3 2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0</td>
<td></td>
<td>imm5</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
</tr>
</tbody>
</table>
```

**T1 variant**

STR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

**Decode for this encoding**

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad \text{imm32} = \text{ZeroExtend}(\text{imm5}'00', 32);
\]

index = TRUE; add = TRUE; wback = FALSE;

**T2**

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10</th>
<th>8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1 0</td>
<td></td>
<td></td>
<td>imm8</td>
</tr>
</tbody>
</table>
```

**T2 variant**

STR{<c>}{<q>} <Rt>, [SP{, #{+}<imm>}]

**Decode for this encoding**

\[
t = \text{UInt}(Rt); \quad n = 13; \quad \text{imm32} = \text{ZeroExtend}(\text{imm8}'00', 32);
\]

index = TRUE; add = TRUE; wback = FALSE;

**T3**

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10</th>
<th>9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 0 1 1 0 0</td>
<td>f=1111</td>
<td>Rt</td>
<td></td>
<td>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**T3 variant**

STR{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // <Rt>, <Rn>, <imm> can be represented in T1 or T2

STR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm}>]

**Decode for this encoding**

\[
\text{if } Rn == '1111' \text{ then UNDEFINED;}
\]

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad \text{imm32} = \text{ZeroExtend}(\text{imm12}, 32);
\]

index = TRUE; add = TRUE; wback = FALSE;

\[
\text{if } t == 15 \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( t == 15 \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

**T4**

```
| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 0 | 15 12 11 10 9 8 7 | 0 |
| 1 1 1 1 0 0 0 1 0 0 | P | U | W | imm8 |
```

**Offset variant**

Applies when \( P == 1 \) \&\& \( U == 0 \) \&\& \( W == 0 \).

\[
\text{STR}\{<c>}{<q>} \ <Rt>, \ [<Rn> \ {, #-<imm>}]
\]

**Post-indexed variant**

Applies when \( P == 0 \) \&\& \( W == 1 \).

\[
\text{STR}\{<c>}{<q>} \ <Rt>, \ [<Rn>], \ #{+/-}<imm>
\]

**Pre-indexed variant**

Applies when \( P == 1 \) \&\& \( W == 1 \).

\[
\text{STR}\{<c>}{<q>} \ <Rt>, \ [<Rn>, \ #{+/-}<imm>]
\]

**Decode for all variants of this encoding**

- If \( \text{Rn} == '1111' \) \&\& \( \text{P} == '0' \) \&\& \( \text{W} == '0' \) then SEE "STRT";
- If \( \text{Rn} == '1111' \) \&\& \( \text{P} == '0' \) \&\& \( \text{W} == '0' \) then UNDEFINED;
- \( t = \text{UInt(Rt)}; \ n = \text{UInt(Rn)}; \ \text{imm32} = \text{ZeroExtend(imm8, 32)}; \)
- \( \text{index} = (P == '1'); \ \text{add} = (U == '1'); \ \text{wback} = (W == '1'); \)
- If \( t == 15 \) \&\& \( \text{wback} \&\& n == t \) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \( \text{wback} \&\& n == t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \( t == 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>of variant</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH (single register)</td>
<td>A1 (pre-indexed)</td>
<td>P == '1' &amp; U == '0' &amp; W == '1' &amp; Rn == '1101' &amp; imm12 == '000000000100'</td>
</tr>
<tr>
<td>PUSH (single register)</td>
<td>T4 (pre-indexed)</td>
<td>Rn == '1101' &amp; P == '1' &amp; U == '0' &amp; W == '1' &amp; imm8 == '00000100'</td>
</tr>
</tbody>
</table>

Assembler symbols

<
See *Standard assembler syntax fields* on page F1-7228.

<q>
See *Standard assembler syntax fields* on page F1-7228.

<Rt>
For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.
For encoding T1, T2, T3 and T4: is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn>
For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.
For encoding T1, T3 and T4: is the general-purpose base register, encoded in the "Rn" field.

+/-
Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when U = 0
+ when U = 1

+ Specifies the offset is added to the base register.

<imm>
For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.
For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 124, defaulting to 0 and encoded in the "imm5" field as <imm>/4.
For encoding T2: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <imm>/4.
For encoding T3: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
For encoding T4: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation for all encodings

```c
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,4] = if t == 15 then PCStoreValue() else R[t];
        if wback then R[n] = offset_addr;
    else
        if ConditionPassed() then
            EncodingSpecificOperations();
            offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
            address = if index then offset_addr else R[n];
            MemU[address,4] = R[t];
            if wback then R[n] = offset_addr;
```

Alias of variant is preferred when

PUSH (single register) A1 (pre-indexed)
P == '1' & U == '0' & W == '1' & Rn == '1101' & imm12 == '000000000100'
PUSH (single register) T4 (pre-indexed)
Rn == '1101' & P == '1' & U == '0' & W == '1' & imm8 == '00000100'
Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.230  STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see Memory accesses on page F1-7233.

A1

| [31] 28|27 26 25 24|23 22 21 20|19 16|15 12|11 | 7 6 5 4 | 3 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| !=1111 | 0 | 1 | P | U | 0 | W | 0 | Rn | Rt | imm5 | stype | 0 | Rm |

**Offset variant**

Applies when \( P = 1 \) && \( W = 0 \).

\[
\text{STR}\{<c>\}{<q>} \text{Rt}, [<Rn>, {+/-}\text{Rm}\{, <shift>\}]
\]

**Post-indexed variant**

Applies when \( P = 0 \) && \( W = 0 \).

\[
\text{STR}\{<c>\}{<q>} \text{Rt}, [<Rn>], {+/-}\text{Rm}\{, <shift>\}
\]

**Pre-indexed variant**

Applies when \( P = 1 \) && \( W = 1 \).

\[
\text{STR}\{<c>\}{<q>} \text{Rt}, [<Rn>, {+/-}\text{Rm}\{, <shift>\}]
\]

**Decode for all variants of this encoding**

if \( P = '0' \) && \( W = '1' \) then SEE "STR";

\[
t = \text{UInt}(\text{Rt}); \quad n = \text{UInt}(<Rn>); \quad \text{m} = \text{UInt}(\text{Rm});
\]

\[
\text{index} = (P = '1'); \quad \text{add} = (U = '1'); \quad \text{wback} = (P = '0') || (W = '1');
\]

\[
(\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{imm5});
\]

if \( \text{m} = 15 \) then UNPREDICTABLE;

if \( \text{wback} && (n = 15 || n = t) \) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \( \text{wback} \) && \( n = t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \( \text{wback} \) && \( n = 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.
T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>6 5</th>
<th>3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 0</td>
<td>0 0</td>
<td>Rm</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**T1 variant**

STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

**Decode for this encoding**

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

\[
\text{index} = \text{TRUE}; \quad \text{add} = \text{TRUE}; \quad \text{wback} = \text{FALSE};
\]

\[
(shift_t, shift_n) = (\text{SRType}_{\text{LSL}}, 0);
\]

T2

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 0 0</td>
<td>1 0</td>
<td>1=1111</td>
<td>Rt</td>
</tr>
</tbody>
</table>

**T2 variant**

STR{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1

STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>], LSL #<imm>]

**Decode for this encoding**

\[
\text{if } \text{Rn} == '1111' \text{ then UNDEFINED};
\]

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

\[
\text{index} = \text{TRUE}; \quad \text{add} = \text{TRUE}; \quad \text{wback} = \text{FALSE};
\]

\[
(shift_t, shift_n) = (\text{SRType}_{\text{LSL}}, \text{UInt}(\text{imm2}))
\]

\[
\text{if } t == 15 || m == 15 \text{ then UNPREDICTABLE}; // Armv8-A removes UNPREDICTABLE for R13
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( t == 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1

Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

\(<\cdot>\) See *Standard assembler syntax fields* on page F1-7228.

\(<\cdot>\) See *Standard assembler syntax fields* on page F1-7228.
<Rt> For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.
For encoding T1 and T2: is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.
For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

+/− Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
−      when U = 0
+      when U = 1

+ Specifies the index register is added to the base register.

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<shift> The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see Shifts applied to a register on page F1-7231.

<imm> If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
  offset_addr = if add then (R[n] + offset) else (R[n] - offset);
  address = if index then offset_addr else R[n];
  bits(32) data;
  if t == 15 then  // Only possible for encoding A1
    data = PCStoreValue();
  else
    data = R[t];
  MemU[address,4] = data;
  if wback then R[n] = offset_addr;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.231   STRB (immediate)

Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page F1-7233.

A1

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=&quot;1111</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>P</td>
<td>U</td>
<td>1</td>
<td>W</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td>Rt</td>
<td>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Offset variant**

Applies when \( P == 1 \) & \( W == 0 \).

\[ \text{STRB}\{\langle c\rangle}\{\langle q\rangle\} \text{ <Rt>}, [\text{<Rn>} \{, #\{+/->\text{imm}\}] \]

**Post-indexed variant**

Applies when \( P == 0 \) & \( W == 0 \).

\[ \text{STRB}\{\langle c\rangle}\{\langle q\rangle\} \text{ <Rt>}, [\text{<Rn>}], #\{+/->\text{imm}\} \]

**Pre-indexed variant**

Applies when \( P == 1 \) & \( W == 1 \).

\[ \text{STRB}\{\langle c\rangle}\{\langle q\rangle\} \text{ <Rt>}, [\text{<Rn>}, #\{+/->\text{imm}\}] \]

**Decode for all variants of this encoding**

if \( P == '0' \) & \( W == '1' \) then SEE "STRBT";

\( t = \text{UInt} (Rt); \) \( n = \text{UInt} (Rn); \) \( \text{imm32} = \text{ZeroExtend} (\text{imm12}, 32); \)

\( \text{index} = (P == '1'); \) \( \text{add} = (U == '1'); \) \( \text{wback} = (P == '0') || (W == '1'); \)

if \( t == 15 \) then \text{UNPREDICTABLE};

if \( \text{wback} \&\& (n == 15 || n == t) \) then \text{UNPREDICTABLE};

**CONSTRAINED UNPREDICTABLE behavior**

If \( t == 15 \), then one of the following behaviors must occur:

- The instruction is \text{UNDEFINED}.
- The instruction executes as \text{NOP}.
- The store instruction performs the store using the specified addressing mode but the value corresponding to \( R15 \) is \text{UNKNOWN}.

If \( \text{wback} \&\& n == t \), then one of the following behaviors must occur:

- The instruction is \text{UNDEFINED}.
- The instruction executes as \text{NOP}.
- The store instruction executes but the value stored is \text{UNKNOWN}.

If \( \text{wback} \&\& n == 15 \), then one of the following behaviors must occur:

- The instruction is \text{UNDEFINED}.
- The instruction executes as \text{NOP}.
- The instruction executes without writeback of the base address.
• The instruction uses the addressing mode described in the equivalent immediate offset instruction.

**T1**

| [15 14 13 12]| 11 10 | 6 5 | 3 2 | 0 |
| 0 1 1 1 | 0 | imm5 | Rn | Rt |

**T1 variant**

STRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

**Decode for this encoding**

\[ t = \text{UInt}(Rt) \; ; \; n = \text{UInt}(Rn) \; ; \; \text{imm32} = \text{ZeroExtend}(\text{imm5}, 32) \; ; \\
\quad \text{index} = \text{TRUE} \; ; \; \text{add} = \text{TRUE} \; ; \; \text{wback} = \text{FALSE} \; ; \]

**T2**

| [15 14 13 12]| 11 10 9 | 8 | 7 6 5 4 | 3 | 0 | 15 12 | 11 | 0 |
| 1 1 1 1 | 0 | 0 | 0 | 0 | 1 | 1 1 1 1 | Rn |

**T2 variant**

STRB{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // <Rt>, <Rn>, <imm> can be represented in T1

STRB{<c>}{<q>} <Rt>, [<Rn> {, #<{imm}>}]  

**Decode for this encoding**

\[ \text{if } Rn == '1111' \text{ then UNDEFINED} ; \\
\quad t = \text{UInt}(Rt) \; ; \; n = \text{UInt}(Rn) \; ; \; \text{imm32} = \text{ZeroExtend}(\text{imm12}, 32) \; ; \\
\quad \text{index} = \text{TRUE} \; ; \; \text{add} = \text{TRUE} \; ; \; \text{wback} = \text{FALSE} \; ; \\
\quad \text{if } t == 15 \text{ then UNPREDICTABLE} ; \] // Armv8-A removes UNPREDICTABLE for R13

**CONSTRAINED UNPREDICTABLE behavior**

If \( t == 15 \), then one of the following behaviors must occur:

• The instruction is **UNDEFINED**.

• The instruction executes as **NOP**.

• The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is **UNKNOWN**.

**T3**

| [15 14 13 12]| 11 10 9 | 8 | 7 6 5 4 | 3 | 0 | 15 12 | 11 9 8 7 | 0 |
| 1 1 1 1 | 0 | 0 | 0 | 0 | 1 | P U W | imm8 |

**Offset variant**

Applies when \( P == 1 \) \&\& \( U == 0 \) \&\& \( W == 0 \).

STRB{<c>}{<q>} <Rt>, [<Rn> {, #-{<imm}>}]
Post-indexed variant

Applies when \( P = 0 \) \&\& \( W = 1 \).

\[
\text{STRB}\{<c>\}{<q>} <Rt>, [<Rn>], \#\{+/-\}<imm>
\]

Pre-indexed variant

Applies when \( P = 1 \) \&\& \( W = 1 \).

\[
\text{STRB}\{<c>\}{<q>} <Rt>, [<Rn>, \#\{+/-\}<imm>]!
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{if } P &= '1' \&\& U = '1' \&\& W = '0' \text{ then SEE "STRBT";} \\
\text{if } Rn &= '1111' \&\& (P = '0' \&\& W = '0') \text{ then UNDEFINED;} \\
t &= \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad \text{imm32} = \text{ZeroExtend}(\text{imm8}, 32); \\
\text{index} &= (P = '1'); \quad \text{add} = (U = '1'); \quad \text{wback} = (W = '1'); \\
\text{if } t &= 15 \&\& (\text{wback} \&\& n = t) \text{ then UNPREDICTABLE; } // \text{Armv8-A removes UNPREDICTABLE for R13}
\end{align*}
\]

CONSTRAINED UNPREDICTABLE behavior

If \( t = 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If \( \text{wback} \&\& n = t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- \( <c> \) See Standard assembler syntax fields on page F1-7228.
- \( <q> \) See Standard assembler syntax fields on page F1-7228.
- \( <Rt> \) Is the general-purpose register to be transferred, encoded in the "Rt" field.
- \( <Rn> \) For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.
  For encoding T1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field.
- \( +/\- \) Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
  - \(- \) when \( U = 0 \)
  - \( + \) when \( U = 1 \)
- \(+ \) Specifies the offset is added to the base register.
For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.

For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

**Operation for all encodings**

```plaintext
if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
  else
    if ConditionPassed() then
      EncodingSpecificOperations();
      offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
      address = if index then offset_addr else R[n];
      MemU[address,1] = R[t]<7:0>;
      if wback then R[n] = offset_addr;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.232 STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see Memory accesses on page F1-7233.

A1

|31| 28|27|26|25|24|23|22|21|20|19|16|15|12|11| 7| 6| 5| 4| 3| 0 |
|  | !|=1111| 0| 1| 1| P| U| 1| W| 0| Rn| Rt| imm5| stype| 0| Rm |

**Offset variant**

Applies when \( P = 1 \) \&\& \( W = 0 \).

\[
\text{STRB}\{<c>\}{<q>}\ <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}] 
\]

**Post-indexed variant**

Applies when \( P = 0 \) \&\& \( W = 0 \).

\[
\text{STRB}\{<c>\}{<q>}\ <Rt>, [<Rn>], {+/-}<Rm>{, <shift>} 
\]

**Pre-indexed variant**

Applies when \( P = 1 \) \&\& \( W = 1 \).

\[
\text{STRB}\{<c>\}{<q>}\ <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}! 
\]

**Decode for all variants of this encoding**

\[
\begin{align*}
\text{if } P &= '0' \ &\& W = '1' \text{ then SEE "STRBT";} \\
\text{t} &= \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \\
\text{index} &= (P = '1'); \quad \text{add} = (U = '1'); \quad \text{wback} = (P = '0') \ || \ (W = '1'); \\
(shift_t, shift_n) &= \text{DecodeImmShift(stype, imm5)}; \\
\text{if } t &= 15 \ || \ m = 15 \text{ then UNPREDICTABLE;} \\
\text{if } \text{wback} \ &\& (n = 15 \ || \ n = t) \text{ then UNPREDICTABLE;}
\end{align*}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( t = 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to \( R15 \) is UNKNOWN.

If \( \text{wback} \ &\& n = t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \( \text{wback} \ &\& n = 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
• The instruction executes without writeback of the base address.
• The instruction uses the addressing mode described in the equivalent immediate offset instruction.

**T1**

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 6 5 3 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 0 1 0 1 0</td>
</tr>
</tbody>
</table>

_T1 variant_

STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

**Decode for this encoding**

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

index = TRUE;  add = TRUE;  wback = FALSE;

(shift_t, shift_n) = (SRType_LSL, 0);

**T2**

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 6 5 4 3 0 15 12 11 10 9 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

_T2 variant_

STRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1

STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>, LSL #<imm>]}

**Decode for this encoding**

if Rn == '1111' then UNDEFINED;
\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

index = TRUE;  add = TRUE;  wback = FALSE;

(shift_t, shift_n) = (SRType_LSL, UInt(imm2));

if t == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**CONSTRAINED UNPREDICTABLE behavior**

If t == 15, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 _Architectural Constraints on UNPREDICTABLE Behaviors_.

**Assembler symbols**

See _Standard assembler syntax fields_ on page F1-7228.


See Standard assembler syntax fields on page F1-7228.

<\( \text{Rt} \)>

Is the general-purpose register to be transferred, encoded in the "Rt" field.

<\( \text{Rn} \)>

For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

\( +/- \)

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when \( U = 0 \)
- when \( U = 1 \)

<\( \text{Rm} \)>

Is the general-purpose index register, encoded in the "Rm" field.

<\( \text{shift} \)>

The shift to apply to the value read from <\( \text{Rm} \)>. If absent, no shift is applied. Otherwise, see Shifts applied to a register on page F1-7231.

<\( \text{imm} \)>

If present, the size of the left shift to apply to the value from <\( \text{Rm} \)>, in the range 1-3. <\( \text{imm} \) is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.233  STRBT

Store Register Byte Unprivileged stores a byte from a register to memory. For information about memory accesses see Memory accesses on page F1-7233.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

A1

```
| 31 28 27 26 25 24 | 23 22 21 20 | 19 16 | 15 | 12 | 11 | 0 |
```

\[ l=1111 \ 0 \ 1 \ 0 \ U \ 1 \ 1 \ 0 \ \text{Rn} \ \text{Rt} \ \text{imm12} \]

**A1 variant**

\[ \text{STRBT}\{<c>\}{<q>} \ <Rt>, \ [<Rn>] \ \{, \ #{+/-}<imm>\} \]

**Decode for this encoding**

\[ t = \text{UInt}(Rt); \ n = \text{UInt}(Rn); \ \text{postindex} = \text{TRUE}; \ \text{add} = (U == '1'); \]
\[ \text{register_form} = \text{FALSE}; \ \text{imm32} = \text{ZeroExtend}(<\text{imm12}, 32); \]
\[ \text{if } t == 15 || n == 15 || n == t \ \text{then UNPREDICTABLE;} \]

**CONSTRAINED UNPREDICTABLE behavior**

If \( t == 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If \( n == t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \( n == 15 \), then one of the following behaviors must occur:

- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

• The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15 by instruction on page K1-11563.

- The instruction uses immediate offset addressing with the base register as PC, without writeback.
A2

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| l=1111 | 0 | 1 | 1 | 0 | U | 1 | 1 | 0 | Rn | Rt | imm5 | stype | 0 | Rm |
| cond |

**A2 variant**

\[
\text{STRBT}\{<c>\}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
\]

**Decode for this encoding**

\[
t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{postindex} = \text{TRUE}; \quad \text{add} = (U == '1'); \\
\text{register\_form} = \text{TRUE}; \quad (\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(stype, \text{imm}5); \\
\text{if } t == 15 || n == 15 || n == t || m == 15 \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \(t == 15\), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The store instruction performs the store using the specified addressing mode but the value corresponding to \(R15\) is **UNKNOWN**.

If \(n == t\), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The store instruction executes but the value stored is **UNKNOWN**.

If \(n == 15\), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15 by instruction on page K1-11563.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | l=1111 | Rt | 1 | 1 | 1 | 0 | imm8 |
| Rn |

**T1 variant**

\[
\text{STRBT}\{<c>\}{<q>} <Rt>, [<Rn> \{, #{+}<imm>\}]
\]
Decode for this encoding

```plaintext
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
```

**CONSTRAINED UNPREDICTABLE behavior**

If t == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields on page F1-7228.*
- `<q>` See *Standard assembler syntax fields on page F1-7228.*
- `<Rt>` For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.
  - For encoding A2 and T1: is the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.
- `+-` For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
  - when U = 0
  - when U = 1
  - For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
    - when U = 0
    - when U = 1
- `<Rm>` Is the general-purpose index register, encoded in the "Rm" field.
- `<shift>` The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see *Shifts applied to a register on page F1-7231.*
- `+` Specifies the offset is added to the base register.
- `<imm>` For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.
  - For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    MemU_unpriv[address,1] = R[t]<7:0>;
    if postindex then R[n] = offset_addr;

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes as STRB (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.234 STRD (immediate)

Store Register Dual (immediate) calculates an address from a base register value and an immediate offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses on page F1-7233.

A1

Store Register Dual (immediate) calculates an address from a base register value and an immediate offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses on page F1-7233.

Offset variant
Applies when \( P == 1 && W == 0 \).

\[ \text{STRD} \{<c>\}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}] \]

Post-indexed variant
Applies when \( P == 0 && W == 0 \).

\[ \text{STRD} \{<c>\}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm> \]

Pre-indexed variant
Applies when \( P == 1 && W == 1 \).

\[ \text{STRD} \{<c>\}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]! \]

Decode for all variants of this encoding

if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt); t2 = t+1; n = UInt(Rn); imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if wback && (n == t || n == t2) then UNPREDICTABLE;
if t2 == 15 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If \( t == 15 || t2 == 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If wback && (n == t || n == t2), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If wback && n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
• The instruction executes without writeback of the base address.
• The instruction uses the addressing mode described in the equivalent immediate offset instruction.

If \( \text{Rt}<\#> = '1' \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes with the additional decode: \( t2 = t \).
• The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when \( \text{Rt} == '1111' \).

If \( P == '0' \) \&\& \( W == '1' \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes as an LDRD using one of offset, post-indexed, or pre-indexed addressing.

### T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 15 12 11 8 7 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 1 1 0 1 0 0 P U 1 W 0 t=1111 Rt Rt2 imm8 |

#### Offset variant
Applies when \( P == 1 \) \&\& \( W == 0 \).

\[
\text{STRD}\{\text{c}\}\{\text{q}\} \text{<Rt>, <Rt2>, [<Rn> \{, #{+/-}<imm>\}]

#### Post-indexed variant
Applies when \( P == 0 \) \&\& \( W == 1 \).

\[
\text{STRD}\{\text{c}\}\{\text{q}\} \text{<Rt>, <Rt2>, [<Rn>], #{+/-}<imm>}

#### Pre-indexed variant
Applies when \( P == 1 \) \&\& \( W == 1 \).

\[
\text{STRD}\{\text{c}\}\{\text{q}\} \text{<Rt>, <Rt2>, [<Rn>, #{+/-}<imm>!}

#### Decode for all variants of this encoding
if \( P == '0' \) \&\& \( W == '0' \) then SEE "Related encodings";
\( t = \text{UInt(Rt)}; \ t2 = \text{UInt(Rt2)}; \ n = \text{UInt(Rn)}; \ \text{imm32} = \text{ZeroExtend(imm8:'00', 32)}; \)
\( \text{index} = (P == '1'); \ \text{add} = (U == '1'); \ \text{wback} = (W == '1'); \)
if wback \&\& (n == t \| n == t2) then UNPREDICTABLE;
if n == 15 \| t == 15 \| t2 == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

#### CONSTRAINED UNPREDICTABLE behavior
If \( t == 15 \| t2 == 15 \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If \( \text{wback} \land (n = t \lor n = t2) \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction executes but the value stored is UNKNOWN.

If \( \text{wback} \land n = 15 \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes without writeback of the base address.
• The instruction uses the addressing mode described in the equivalent immediate offset instruction.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.


**Assembler symbols**

- `<>` See *Standard assembler syntax fields* on page F1-7228.
- `<q>` See *Standard assembler syntax fields* on page F1-7228.
- `<Rt>` For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.
  
  For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Rt2>` For encoding A1: is the second general-purpose register to be transferred. This register must be \( R(t+1) \).
  
  For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Rn>` For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.
  
  For encoding T1: is the general-purpose base register, encoded in the "Rn" field.
- `+/` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
  
  - when \( U = 0 \)
  + when \( U = 1 \)
- `<imm>` For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.
  
  For encoding T1: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 if omitted, and encoded in the "imm8" field as `<imm>/4`.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
```
address = if index then offset_addr else R[n];
if address == Align(address, 8) then
    bits(64) data;
    if BigEndian(AccType_ATOMIC) then
        data<63:32> = R[t];
        data<31:0> = R[t2];
    else
        data<31:0> = R[t];
        data<63:32> = R[t2];
    MemA[address,8] = data;
else
    MemA[address,4] = R[t];
    MemA[address+4,4] = R[t2];
if wback then R[n] = offset_addr;

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.235 STRD (register)

Store Register Dual (register) calculates an address from a base register value and a register offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses on page F1-7233.

A1

|31| 28|27|26|25|24|23|22|21|20|19|16|15|12|11|10| 9 |8 |7 |6 |5 |4 |3 |0 |
|   | 0 | 0 | 0 |P | U | 0 |W | 0 | Rn | Rt | 0 |0 |0 |0 |1 |1 |1 |1 | Rm |
|cond|

**Offset variant**

Applies when \(P == 1 \&\& W == 0\).

\[\text{STRD}\{<c>\}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]\]

**Post-indexed variant**

Applies when \(P == 0 \&\& W == 0\).

\[\text{STRD}\{<c>\}{<q>} <Rt>, <Rt2>, [<Rn>], {+/-}<Rm}\]

**Pre-indexed variant**

Applies when \(P == 1 \&\& W == 1\).

\[\text{STRD}\{<c>\}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]!\]

**Decode for all variants of this encoding**

\[
\text{if } Rt<0> == '1' \text{ then UNPREDICTABLE;}
\text{t} = \text{UInt}(Rt); \text{t2} = \text{t}+1; \text{n} = \text{UInt}(Rn); \text{m} = \text{UInt}(Rm);
\text{index} = (P == '1'); \text{add} = (U == '1'); \text{wback} = (P == '0') || (W == '1');
\text{if } P == '0' \&\& W == '1' \text{ then UNPREDICTABLE;}
\text{if } t2 == 15 || m == 15 \text{ then UNPREDICTABLE;}
\text{if wback \&\& (n == 15 || n == t || n == t2) then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \(t == 15 \&\& t2 == 15\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If \(\text{wback} \&\& (n == t \&\& n == t2)\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \(\text{wback} \&\& n == 15\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
• The instruction executes without writeback of the base address.
• The instruction uses the addressing mode described in the equivalent immediate offset instruction.

If $Rt<0> = '1'$, then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes with the additional decode: $t<0> = '0'$.
• The instruction executes with the additional decode: $t2 = t$.
• The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when $Rt == '1111'$.

If $P == '0'$ \&\& $W == '1'$, then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes with the additional decode: $P = '1'; W = '0'$.
• The instruction executes with the additional decode: $P = '1'; W = '1'$.
• The instruction executes with the additional decode: $P = '0'; W = '0'$.

Notes for all encodings
For more information about the CONSTRANGED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\) See Standard assembler syntax fields on page F1-7228.
\(<q>\) See Standard assembler syntax fields on page F1-7228.
\(<Rt>\) Is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.
\(<Rt2>\) Is the second general-purpose register to be transferred. This register must be $R(t+1)$.

\(<Rn>\) Is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

\(+/-\) Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
- when $U = 0$
+ when $U = 1$

\(<Rm>\) Is the general-purpose index register, encoded in the "Rm" field.

Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        bits(64) data;
    if BigEndian(AccType_ATOMIC) then
data<63:32> = R[t];
data<31:0> = R[t2];
else
  data<31:0> = R[t];
data<63:32> = R[t2];
  MemA[address,8] = data;
else
  MemA[address,4] = R[t];
  MemA[address+4,4] = R[t2];
  if wback then R[n] = offset_addr;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.236 STREX

Store Register Exclusive calculates an address from a base register value and an immediate offset, stores a word from a register to the calculated address if the PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For information about memory accesses see Memory accesses on page F1-7233.

A1

```
|31| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 16| 15| 12| 11| 10| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
 !l=1111 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | Rn | Rd | (1) | (1) | 1 | 1 | 0 | 0 | 1 | Rt |

cond
```

**A1 variant**

STREX{<c>}{<q>} <Rd>, <Rt>, [{<Rn} {, {#}<imm}>}

**Decode for this encoding**

```
d = Uint(Rd);  t = Uint(Rt);  n = Uint(Rn);  imm32 = Zeros(32); // Zero offset
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

**CONSTRAINED UNPREDICTABLE behavior**

If d == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If d == n, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1

```
|15| 14| 13| 12| 11| 10| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15| 12| 11| 8 | 7 | | 0 |
 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | Rn | Rt | Rd | imm8 |
```

**T1 variant**

STREX{<c>}{<q>} <Rd>, <Rt>, [{<Rn} {, {#}<imm}>}

**Decode for this encoding**

```
d = Uint(Rd);  t = Uint(Rt);  n = Uint(Rn);  imm32 = ZeroExtend(imm8:’00’, 32);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t then UNPREDICTABLE;
```
CONSTRAINED UNPREDICTABLE behavior

If \( d == t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as \texttt{NOP}.
- The store instruction executes but the value stored is UNKNOWN.

If \( d == n \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as \texttt{NOP}.
- The instruction performs the store to an UNKNOWN address.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\) See Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<Rd>\) Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

\begin{align*}
0 & \quad \text{If the operation updates memory.} \\
1 & \quad \text{If the operation fails to update memory.}
\end{align*}

\(<Rt>\) Is the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Rn>\) Is the general-purpose base register, encoded in the "Rn" field.

\(<\text{imm}>\) For encoding A1: the immediate offset added to the value of \(<Rn>\) to calculate the address. \(<\text{imm}>\) can only be 0 or omitted.

For encoding T1: the immediate offset added to the value of \(<Rn>\) to calculate the address. \(<\text{imm}>\) can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- \(<Rd>\) is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns \texttt{TRUE}, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns \texttt{FALSE} and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    if AArch32.ExclusiveMonitorsPass(address, 4) then
        MemA[address, 4] = R[t];
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('1', 32);

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.237   STREXB

Store Register Exclusive Byte derives an address from a base register value, stores a byte from a register to the
derived address if the executing PE has exclusive access to the memory at that address, and returns a status value
of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For
information about memory accesses see Memory accesses on page F1-7233.

A1

\[
\begin{array}{cccccccccccccc}
\end{array}
\]

\[
\text{cond} = \begin{array}{ccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \end{array}
\]

A1 variant

STREXB\{<c>\}{<q>} <Rd>, <Rt>, [<Rn]>

Decode for this encoding

\[
d = \text{UInt}(Rd); \ t = \text{UInt}(Rt); \ n = \text{UInt}(Rn);
\]

\[
\text{if } d = 15 \ || \ t = 15 \ || \ n = 15 \text{ then UNPREDICTABLE;}
\]

\[
\text{if } d = n \ || \ d = t \text{ then UNPREDICTABLE;}
\]

CONSTRANGED UNPREDICTABLE behavior

If \(d = t\), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction executes but the value stored is UNKNOWN.

If \(d = n\), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction performs the store to an UNKNOWN address.

T1

\[
\begin{array}{ccccccccccccccc}
| & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 |
\end{array}
\]

\[
\begin{array}{ccccccccccccccc}
1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \end{array}
\]

T1 variant

STREXB\{<c>\}{<q>} <Rd>, <Rt>, [<Rn]>

Decode for this encoding

\[
d = \text{UInt}(Rd); \ t = \text{UInt}(Rt); \ n = \text{UInt}(Rn);
\]

\[
\text{if } d = 15 \ || \ t = 15 \ || \ n = 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

\[
\text{if } d = n \ || \ d = t \text{ then UNPREDICTABLE;}
\]
**CONSTRUDED UNPREDICTABLE behavior**

If d == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If d == n, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

**Notes for all encodings**

For more information about the CONSTRUDED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

<

See Standard assembler syntax fields on page F1-7228.

<

See Standard assembler syntax fields on page F1-7228.

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

- 0 if the operation updates memory.
- 1 if the operation fails to update memory.

<Rt>

Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn>

Is the general-purpose base register, encoded in the "Rn" field.

**Aborts**

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address, 1) then
        MemA[address, 1] = R[t]<7:0>;
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('1', 32);
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.238  STREXD

Store Register Exclusive Doubleword derives an address from a base register value, stores a 64-bit doubleword from two registers to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see Synchronization and semaphores on page E2-7211. For information about memory accesses see Memory accesses on page F1-7233.

A1

```
| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| l=1111 | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  |
```

A1 variant

STREXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]

Decode for this encoding

\[
\begin{align*}
d &= \text{UInt}(Rd); \\
t &= \text{UInt}(Rt); \\
t2 &= t+1; \\
n &= \text{UInt}(Rn);
\end{align*}
\]

if \(d = 15 \text{ or } Rt<0> = '1' \text{ or } t2 = 15 \text{ or } n = 15\) then UNPREDICTABLE;

if \(d = n \text{ or } d = t \text{ or } d = t2\) then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If \(d = t\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \(d = n\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

If \(Rt<0> = '1'\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: \(Rt<0> = '0'\).
- The instruction executes with the additional decode: \(t2 = t\).
- The instruction executes as described, with no change to its behavior and no additional side effects.

If \(Rt = '1110'\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction is handled as described in Using R15 by instruction on page K1-11563.
T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 12 11 8 7 6 5 4 3 0 |
| 1 1 1 0 1 0 0 1 1 0 0 Rn | Rt | Rt2 | 0 1 1 1 Rd |

**T1 variant**

STREXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]

**Decode for this encoding**

\[
\begin{align*}
d &= \text{UInt}(Rd); \\
t &= \text{UInt}(Rt); \\
t2 &= \text{UInt}(Rt2); \\
n &= \text{UInt}(Rn);
\end{align*}
\]

if \(d = 15\) || \(t = 15\) || \(t2 = 15\) || \(n = 15\) then UNPREDICTABLE;

// Armv8-A removes UNPREDICTABLE for R13
if \(d = n\) || \(d = t\) || \(d = t2\) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \(d = t\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \(d = n\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

\(<c>\) See Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<\text{Rd}>\) Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:
\[
\begin{align*}
0 & \quad \text{If the operation updates memory.} \\
1 & \quad \text{If the operation fails to update memory.}
\end{align*}
\]

\(<\text{Rd}>\) must not be the same as \(<\text{Rn}>\), \(<\text{Rt}>\), or \(<\text{Rt2}>\).

\(<\text{Rt}>\) For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. \(<\text{Rt}>\) must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

\(<\text{Rt2}>\) For encoding A1: is the second general-purpose register to be transferred. \(<\text{Rt2}>\) must be \(R(t+1)\).

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

\(<\text{Rn}>\) Is the general-purpose base register, encoded in the "Rn" field.
Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- \(<Rd>\) is not updated.

A non doubleword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If \(\text{AArch32.ExclusiveMonitorsPass()}\) returns \(\text{TRUE}\), the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If \(\text{AArch32.ExclusiveMonitorsPass()}\) returns \(\text{FALSE}\) and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation for all encodings

\[
\text{if } \text{ConditionPassed()} \text{ then}
\begin{align*}
\text{if } \text{BigEndian(AccType_ATOMIC)} \text{ then } R[t]:R[t2] \text{ else } R[t2]:R[t];
\text{if } \text{AArch32.ExclusiveMonitorsPass(address,8)} \text{ then }
\text{MemA}[address,8] = value; \quad R[d] = \text{ZeroExtend('0', 32)};
\text{else }
R[d] = \text{ZeroExtend('1', 32)};
\end{align*}
\]

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.239   **STREXH**

Store Register Exclusive Halfword derives an address from a base register value, stores a halfword from a register to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see *Synchronization and semaphores* on page E2-7211. For information about memory accesses see *Memory accesses* on page F1-7233.

### A1

```
[31  28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 |7 6 5 4 |3 0 |
   l=1111 0 0 0 1 1 1 0 | Rn | Rd | (1)(1) | 1 1 0 0 1 | Rt |
```

**A1 variant**

STREXH{<c>}{<q>} <Rd>, <Rt>, [Rn]

**Decode for this encoding**

\[
\begin{align*}
d &= \text{UInt}(Rd); \\
t &= \text{UInt}(Rt); \\
n &= \text{UInt}(Rn);
\end{align*}
\]
\[
\text{if } d == 15 || t == 15 || n == 15 \text{ then UNPREDICTABLE;}
\]
\[
\text{if } d == n || d == t \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( d == t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \( d == n \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

### T1

```
[15 14 13 12|11 10 9 8 |7 6 5 4 |3 0 |
   1 1 1 0 | 1 0 0 0 1 1 0 0 | Rn | Rt | (1)(1)(1) | 0 1 0 1 | Rd |
```

**T1 variant**

STREXH{<c>}{<q>} <Rd>, <Rt>, [Rn]

**Decode for this encoding**

\[
\begin{align*}
d &= \text{UInt}(Rd); \\
t &= \text{UInt}(Rt); \\
n &= \text{UInt}(Rn);
\end{align*}
\]
\[
\text{if } d == 15 || t == 15 || n == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]
\[
\text{if } d == n || d == t \text{ then UNPREDICTABLE;}
\]
**CONSTRAINED UNPREDICTABLE behavior**

If \( d == t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \( d == n \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- \(<c>\) See Standard assembler syntax fields on page F1-7228.
- \(<q>\) See Standard assembler syntax fields on page F1-7228.
- \(<Rd>\) Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:
  - 0 If the operation updates memory.
  - 1 If the operation fails to update memory.
- \(<Rt>\) Is the general-purpose register to be transferred, encoded in the "Rt" field.
- \(<Rn>\) Is the general-purpose base register, encoded in the "Rn" field.

**Aborts and alignment**

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- \(<Rd>\) is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address, 2) then
        MemA[address, 2] = R[t]<15:0>;
        R[d] = ZeroExtend('0', 32);
```

else
    R[d] = ZeroExtend('1', 32);

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.240 STRH (immediate)

Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see Memory accesses on page F1-7233.

A1

| [31] 28 27 26 25 24 23 22 21 20 19 16 15 12 11 8 7 6 5 4 3 0 |
|----------------|----------------|----------------|----------------|
| !==1111 0 0 0 | P U 1 | W 0 | Rn | Rt | imm4H 1 | 0 | 1 | 1 | imm4L |
| cond |

**Offset variant**

Applies when \( P == 1 \&\& W == 0 \).

\[
\text{STRH}\{<c>}{<q}> <Rt>, [<Rn> {, #\{+/-\}<imm}>]
\]

**Post-indexed variant**

Applies when \( P == 0 \&\& W == 0 \).

\[
\text{STRH}\{<c>}{<q}> <Rt>, [<Rn>], #\{+/-\}<imm>
\]

**Pre-indexed variant**

Applies when \( P == 1 \&\& W == 1 \).

\[
\text{STRH}\{<c>}{<q}> <Rt>, [<Rn>], #\{+/-\}<imm>!
\]

**Decode for all variants of this encoding**

if \( P == '0' \&\& W == '1' \) then SEE "STRHT";

\[
t = \text{UInt}(Rt); \ n = \text{UInt}(Rn); \ \text{imm32} = \text{ZeroExtend}(\text{imm4H:imm4L}, 32);
\]

\[
\text{index} = (P == '1'); \ \text{add} = (U == '1'); \ \text{wback} = (P == '0') || (W == '1');
\]

if \( t == 15 \) then UNPREDICTABLE;

if \( \text{wback} \&\& (n == 15 || n == t) \) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \( t == 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If \( \text{wback} \&\& n == t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \( \text{wback} \&\& n == 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
• The instruction uses the addressing mode described in the equivalent immediate offset instruction.

### T1

![T1 encoding]

**T1 variant**

`STRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]`

**Decode for this encoding**

\[
t = \text{UInt}(Rt);
\]
\[
n = \text{UInt}(Rn);
\]
\[
\text{imm32} = \text{ZeroExtend}(\text{imm}5:0', 32);
\]
\[
\text{index} = \text{TRUE}; \quad \text{add} = \text{TRUE}; \quad \text{wback} = \text{FALSE};
\]

### T2

![T2 encoding]

**T2 variant**

`STRH{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}]` // <Rt>, <Rn>, <imm> can be represented in T1

`STRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]`

**Decode for this encoding**

\[
\text{if } Rn == '1111' \text{ then UNDEFINED};
\]
\[
t = \text{UInt}(Rt);
\]
\[
n = \text{UInt}(Rn);
\]
\[
\text{imm32} = \text{ZeroExtend}(\text{imm}12, 32);
\]
\[
\text{index} = \text{TRUE}; \quad \text{add} = \text{TRUE}; \quad \text{wback} = \text{FALSE};
\]
\[
\text{if } t == 15 \text{ then UNPREDICTABLE}; \quad // \text{Armv8-A removes UNPREDICTABLE for R13}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \(t == 15\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

### T3

![T3 encoding]

**Offset variant**

Applies when \(P == 1 \&\& U == 0 \&\& W == 0\).

`STRH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]]`
Post-indexed variant

Applies when \( P = 0 \) \&\& \( W = 1 \).

\[
\text{STRH}\{<c>\}{<q>}\ <Rt>, \ [<Rn>], \ #{+/-}<imm>
\]

Pre-indexed variant

Applies when \( P = 1 \) \&\& \( W = 1 \).

\[
\text{STRH}\{<c>\}{<q>}\ <Rt>, \ [<Rn>, \ #{+/-}<imm>]
\]

Decode for all variants of this encoding

\[
\text{if } P = '1' \&\& U = '1' \&\& W = '0' \text{ then see } "\text{STRHT}";
\]
\[
\text{if } Rn = '1111' || (P = '0' \&\& W = '0') \text{ then UNDEFINED};
\]
\[
t = \text{UInt}(Rt); \ n = \text{UInt}(Rn); \ \text{imm32} = \text{ZeroExtend}(\text{imm8}, 32);
\]
\[
\text{index} = (P = '1'); \ \text{add} = (U = '1'); \ \text{wback} = (W = '1');
\]
\[
\text{if } t = 15 || (\text{wback} \&\& n = t) \text{ then UNPREDICTABLE}; \ // \text{Armv8-A removes UNPREDICTABLE for R13}
\]

CONSTRUANCED UNPREDICTABLE behavior

If \( t = 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If \( wback \&\& n = t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

Notes for all encodings

For more information about the CONSTRUANCED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\) See Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<Rt>\) Is the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Rn>\) For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding A1, T1, T2, T3: is the general-purpose base register, encoded in the "Rn" field.

\(+/-\) Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when \( U = 0 \)
- when \( U = 1 \)

+ Specifies the offset is added to the base register.
For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 2, in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as <imm>/2.

For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

**Operation for all encodings**

```c
if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,2] = R[t]<15:0>;
    if wback then R[n] = offset_addr;
  else
    if ConditionPassed() then
      EncodingSpecificOperations();
      offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
      address = if index then offset_addr else R[n];
      MemU[address,2] = R[t]<15:0>;
      if wback then R[n] = offset_addr;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.241  STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and
stores a halfword from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For
information about memory accesses see Memory accesses on page F1-7233.

A1

| 31 | 28 | 27 | 26 | 25 | 24 | 22 | 21 | 20 | 19 | 16 | 15 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|
| ![1111] | 0 | 0 | 0 | P | U | 0 | W | 0 | Rn | Rt | (0) | (0) | (0) | 1 | 0 | 1 | 1 | Rm |

**Offset variant**

Applies when P == 1 && W == 0.

`STRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]`

**Post-indexed variant**

Applies when P == 0 && W == 0.

`STRH{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>`

**Pre-indexed variant**

Applies when P == 1 && W == 1.

`STRH{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>`!

**Decode for all variants of this encoding**

if P == '0' && W == '1' then SEE "STRHT";

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If t == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to
  R15 is UNKNOWN.

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If wback && n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
The instruction executes without writeback of the base address.

The instruction uses the addressing mode described in the equivalent immediate offset instruction.

### T1

| 15 14 13 12| 11 10 9 8 | 6 5 | 3 2 0 |
| 0 1 0 1 0 0 1 | Rm | Rn | Rt |

**T1 variant**

STRH\{<c>\}{<q>} <Rt>, [<Rn>, {+}<Rm>]

**Decode for this encoding**

\[t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]
\[\text{index} = \text{true}; \quad \text{add} = \text{true}; \quad \text{wback} = \text{false};
\]
\[(\text{shift}_t, \text{shift}_n) = (\text{SRType_LSL}, 0);
\]

### T2

| 15 14 13 12| 11 10 9 8 | 7 6 5 | 4 3 | 0 | 15 | 12| 11 10 9 8 | 7 6 5 | 4 3 | 0 |
| 1 1 1 1 1 0 0 0 0 0 1 0 | \text{f=1111} | Rt | 0 0 0 0 0 0 0 \text{imm2} | Rm |

**T2 variant**

STRH\{<c>\}.W <Rt>, [<Rn>, {+}<Rm>] // <Rt>, <Rn>, <Rm> can be represented in T1

STRH\{<c>\}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

**Decode for this encoding**

\[\text{if } Rn == '1111' \text{ then UNDEFINED;}
\]
\[t = \text{UInt}(Rt); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]
\[\text{index} = \text{true}; \quad \text{add} = \text{true}; \quad \text{wback} = \text{false};
\]
\[(\text{shift}_t, \text{shift}_n) = (\text{SRType_LSL}, \text{UInt}(\text{imm2}));
\]
\[\text{if } t == 15 \text{ || } m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \[t == 15\], then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

\(<c>\) \quad \text{See Standard assembler syntax fields on page F1-7228.}
<q>
See *Standard assembler syntax fields on page F1-7228.*
</q>

<Rt>
Is the general-purpose register to be transferred, encoded in the "Rt" field.
</Rt>

<Rn>
For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.
</Rn>

+/-
Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when \( U = 0 \)
+ when \( U = 1 \)

<imm>
If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

+<Rm>
Is the general-purpose index register, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    MemU[address,2] = R[t]<15:0>;
    if wback then R[n] = offset_addr;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.242   STRHT

Store Register Halfword Unprivileged stores a halfword from a register to memory. For information about memory accesses see Memory accesses on page F1-7233.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

A1

A1 variant

STRHT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}

Decode for this encoding

t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If t == 15, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If n == t, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction executes but the value stored is UNKNOWN.

If n == 15, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15 by instruction on page K1-11563.
• The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.
A2

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| cond |

A2 variant

STRHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>

Decode for this encoding

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If t == 15, then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If n == t, then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction executes but the value stored is UNKNOWN.

If n == 15, then one of the following behaviors must occur:
• The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 12 | 11 | 10 | 9 | 8 | 7 |   |   |   |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  |   |   |   |   |   | 1  | 1  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |

T1 variant

STRHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<imm>
Decode for this encoding

if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

CONSTRAINED UNPREDICTABLE behavior

If t == 15, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<->  See Standard assembler syntax fields on page F1-7228.
<q>  See Standard assembler syntax fields on page F1-7228.
<Rt>  Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>  Is the general-purpose base register, encoded in the "Rn" field.
+-  For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
- when U = 0
+ when U = 1
For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
- when U = 0
+ when U = 1

<Rm>  Is the general-purpose index register, encoded in the "Rm" field.
+
  Specifies the offset is added to the base register.

<imm>  For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.
For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
offset = if register_form then R[m] else imm32;
offset_addr = if add then (R[n] + offset) else (R[n] - offset);
address = if postindex then R[n] else offset_addr;
MemU_unpriv[address,2] = R[t]<15:0>;
if postindex then R[n] = offset_addr;

**CONSTRAINED UNPREDICTABLE behavior**

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as STRH (immediate).

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.243   STRT

Store Register Unprivileged stores a word from a register to memory. For information about memory accesses see Memory accesses on page F1-7233.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

A1

<table>
<thead>
<tr>
<th>31 28 27 26 25 24 23 22 21 20 19 16 15 12 11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111 0 1 0 0 U 0 1 0</td>
<td>Rn</td>
</tr>
<tr>
<td>Rt</td>
<td>imm12</td>
</tr>
</tbody>
</table>

A1 variant

STRT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}

Decode for this encoding

t = UInt(Rt); n = UInt(Rn); postindex = TRUE; add = (U == '1');
register_form = FALSE; imm32 = ZeroExtend(imm12, 32);
if n == 15 || n == t then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15 by instruction on page K1-11563.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

A2

<table>
<thead>
<tr>
<th>31 28 27 26 25 24 23 22 21 20 19 16 15 12 11</th>
<th>7 6 5 4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111 0 1 1 0 U 0 1 0</td>
<td>Rn</td>
<td></td>
</tr>
<tr>
<td>Rt</td>
<td>imm5</td>
<td>stype 0</td>
</tr>
</tbody>
</table>
**A2 variant**

STRT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}

**Decode for this encoding**

\[
\begin{align*}
& t = \text{UInt}(Rt); \\
& n = \text{UInt}(Rn); \\
& m = \text{UInt}(Rm); \\
& \text{postindex} = \text{TRUE}; \\
& \text{add} = (U == '1'); \\
& \text{register\_form} = \text{TRUE}; \\
& (\text{shift\_t}, \text{shift\_n}) = \text{DecodeImmShift(stype, imm5)}; \\
& \text{if } n == 15 \text{ || } n == t \text{ || } m == 15 \text{ then UNPREDICTABLE; }
\end{align*}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( n == t \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If \( n == 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15 by instruction* on page K1-11563.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

**T1**

\[
\begin{array}{cccccccccccccccc}
|15|14|13|12|11|10|9|8|7|6|5|4|3|0|15|12|11|10|9|8|7|0|\hline
1|1|1|1|1|0|0|0|1|0|0|!=1111|Rt|1|1|1|0|imm8
\end{array}
\]

**T1 variant**

STRT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]}

**Decode for this encoding**

\[
\begin{align*}
& \text{if } Rn == '1111' \text{ then UNDEFINED; } \\
& t = \text{UInt}(Rt); \\
& n = \text{UInt}(Rn); \\
& \text{postindex} = \text{FALSE}; \\
& \text{add} = \text{TRUE}; \\
& \text{register\_form} = \text{FALSE}; \\
& \text{imm32} = \text{ZeroExtend}(\text{imm8}, 32); \\
& \text{if } t == 15 \text{ then UNPREDICTABLE; } \text{// Armv8-A removes UNPREDICTABLE for R13}
\end{align*}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( t == 15 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\)
See Standard assembler syntax fields on page F1-7228.

\(<q>\)
See Standard assembler syntax fields on page F1-7228.

\(<Rt>\)
For encoding A1 and A2: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.
For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Rn>\)
Is the general-purpose base register, encoded in the "Rn" field.

\(+/-\)
For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
- when U = 0
+ when U = 1
For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
- when U = 0
+ when U = 1

\(<Rm>\)
Is the general-purpose index register, encoded in the "Rm" field.

\(<\text{shift}>\)
The shift to apply to the value read from \(<Rm>\). If absent, no shift is applied. Otherwise, see Shifts applied to a register on page F1-7231.

\(+\)
Specifies the offset is added to the base register.

\(<\text{imm}>\)
For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.
For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
  offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
  offset_addr = if add then (R[n] + offset) else (R[n] - offset);
  address = if postindex then R[n] else offset_addr;
  bits(32) data;
  if t == 15 then // Only possible for encodings A1 and A2
    data = PCStoreValue();
  else
    data = R[t];
  MemU_unpriv[address,4] = data;
  if postindex then R[n] = offset_addr;

CONSTRANGED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes as STR (immediate).

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F5.1.244    SUB (immediate, from PC)

Subtract from PC subtracts an immediate value from the Align(PC, 4) value to form a PC-relative address, and
writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias
This instruction is an alias of the ADR instruction. This means that:

- The encodings in this description are named to match the encodings of ADR.
- The description of ADR gives the operational pseudocode for this instruction.

A2

```
| 31 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 |   |   |
| 1=1111 | 0 0 1 0 0 1 0 0 1 1 1 1 | Rd | imm12 |
|         cond |
```

A2 variant

SUB\{<c>\}{<q>} <Rd>, PC, #<const>

is equivalent to

ADR\{<c>\}{<q>} <Rd>, <label>

and is the preferred disassembly when imm12 == '000000000000'.

T2

```
| 15 14 13 12 | 11 10 9 8 7 6 5 4 3 2 1 0 | 15 14 | 12 11 | 8 7 |   |   |
| 1 1 1 1 0 | 1 1 0 1 0 1 0 1 1 1 1 0 | imm3 | Rd | imm8 |
```

T2 variant

SUB\{<c>\}{<q>} <Rd>, PC, #imm12

is equivalent to

ADR\{<c>\}{<q>} <Rd>, <label>

and is the preferred disassembly when i:imm3:imm8 == '000000000000'.

Assembler symbols

- \(<c>\) See Standard assembler syntax fields on page F1-7228.
- \(<q>\) See Standard assembler syntax fields on page F1-7228.
- \(<Rd>\) For encoding A2: is the general-purpose destination register, encoded in the "Rd" field. If the PC is
  used, the instruction is a branch to the address calculated by the operation. This is an interworking
  branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the
  PC on page E1-7133.
  For encoding T2: is the general-purpose destination register, encoded in the "Rd" field.
- \(<label>\) For encoding A2: the label of an instruction or literal data item whose address is to be loaded into
  \(<Rd>\). The assembler calculates the required value of the offset from the Align(PC, 4) value of the
  ADR instruction to this label.
  If the offset is zero or positive, encoding A1 is used, with imm32 equal to the offset.
If the offset is negative, encoding A2 is used, with imm32 equal to the size of the offset. That is, the use of encoding A2 indicates that the required offset is minus the value of imm32.

Permitted values of the size of the offset are any of the constants described in Modified immediate constants in A32 instructions on page F1-7244.

For encoding T2: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the A1ign(PC, 4) value of the ADR instruction to this label.

If the offset is zero or positive, encoding T3 is used, with imm32 equal to the offset.

If the offset is negative, encoding T2 is used, with imm32 equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of imm32.

Permitted values of the size of the offset are 0-4095.

<imm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

<const> An immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values.

Operation for all encodings

The description of ADR gives the operational pseudocode for this instruction.
F5.1.245 SUB, SUBS (immediate)

Subtract (immediate) subtracts an immediate value from a register value, and writes the result to the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- The SUBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode, except for encoding T5 with <imm8> set to zero, which is the encoding for the ERET instruction, see ERET.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

\[
\begin{array}{cccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccc}
\text{cond} & \text{imm12} & \text{Rd} & \text{Rn} & S & 0 & 1 & 0 & 0 & 1 & 0
\end{array}
\]

**SUB variant**

Applies when S == 0 && Rn != 11x1.

\[
\text{SUB\{}<c>{<q}\{}{<Rd>,} <Rn>, #<const>}
\]

**SUBS variant**

Applies when S == 1 && Rn != 1101.

\[
\text{SUBS\{}<c>{<q}\{}{<Rd>,} <Rn>, #<const>}
\]

**Decode for all variants of this encoding**

- if Rn == '1111' && S == '0' then SEE "ADR";
- if Rn == '1101' then SEE "SUB (SP minus immediate)";
- d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = A32ExpandImm(imm12);

T1

\[
\begin{array}{cccccccccccc}
| 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 6 & 5 & 3 & 2 & 0 |
\end{array}
\]

\[
\begin{array}{cccccccccccc}
0 & 0 & 0 & 1 & 1 & 1 & 1 & \text{imm3} & \text{Rd} & \text{Rn}
\end{array}
\]

**T1 variant**

\[
\text{SUB\{}<c>{<q}\{}{<Rd>,} <Rn>, #<imm3> // Inside IT block}
\]

\[
\text{SUBS\{}<c>{<q}\{}{<Rd>,} <Rn>, #<imm3> // Outside IT block}
\]
Decode for this encoding

d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);

T2

\[
\begin{array}{cccccccc}
| & | & 15 & 14 & 13 & 12 & 11 & 10 & 8 & 7 & | & 0 & | \\
0 & 0 & 1 & 1 & 1 & Rdn & | & imm8 & | & & & |
\end{array}
\]

T2 variant

SUB<c>{<q>} <Rdn>, #<imm8> // Inside IT block, and <Rdn>, <imm8> can be represented in T1
SUB<c>{<q>} {<Rdn>,} <Rdn>, #<imm8> // Inside IT block, and <Rdn>, <imm8> cannot be represented in T1
SUBS{<q>} <Rdn>, #<imm8> // Outside IT block, and <Rdn>, <imm8> can be represented in T1
SUBS{<q>} {<Rdn>,} <Rdn>, #<imm8> // Outside IT block, and <Rdn>, <imm8> cannot be represented in T1

Decode for this encoding

d = UInt(Rd);  n = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);

T3

\[
\begin{array}{cccccccccccccccc}
| & | & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 14 & 12 & 11 & 8 & 7 & | & 0 & | \\
1 & 1 & 1 & 1 & 0 & | & 0 & 1 & 1 & 0 & 1 & S & | & 1101 & 0 & | & imm3 & Rd & | & imm8 & | & Rn
\end{array}
\]

SUB variant

Applies when \( S == 0 \).

SUB<.W {<Rd>,} <Rn>, #<const> // Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2
SUB{<c>}{<q>} {<Rd>,} <Rn>, #<const>

SUBS variant

Applies when \( S == 1 \) && \( Rd != 1111 \).

SUBS.W {<Rd>,} <Rn>, #<const> // Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2
SUBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

Decode for all variants of this encoding

if \( Rd == '1111' \) && \( S == '1' \) then SEE "CMP (immediate)";
if \( Rn == '11001' \) then SEE "SUB (SP minus immediate)";
if \( d == '11101' \) then SEE "SUB (SP minus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

T4

\[
\begin{array}{cccccccccccccccc}
| & | & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 14 & 12 & 11 & 8 & 7 & | & 0 & | \\
1 & 1 & 1 & 1 & 0 & | & 1 & 0 & 1 & 0 & 1 & 0 & 1 & i11x1 & 0 & | & imm3 & Rd & | & imm8 & | & Rn
\end{array}
\]
T4 variant

SUB{<c>}{<q>} {<Rd>,} <Rn>, #=>imm12> // <imm12> cannot be represented in T1, T2, or T3
SUBW{<c>}{<q>} {<Rd>,} <Rn>, #=>imm12> // <imm12> can be represented in T1, T2, or T3

Decode for this encoding

if Rn == '1111' then SEE "ADR";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

T5

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1 1 1 0 1</td>
<td>Rn</td>
<td>1 0</td>
<td>0 (1)(1)(1)(1)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

T5 variant

Applies when !(Rn == 1110 && imm8 == 00000000).

SUBS{<c>}{<q>} PC, LR, #=>imm8>

Decode for this encoding

if Rn == '1110' && IsZero(imm8) then SEE "ERET";
d = 15;  n = UInt(Rn);  setflags = TRUE;  imm32 = ZeroExtend(imm8, 32);
if n != 14 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly SUBS PC, LR and related instructions (A32) on page K1-11576 and SUBS PC, LR and related instructions (T32) on page K1-11575.

Assembler symbols

<>
See Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

<r> Is the general-purpose source and destination register, encoded in the "Rdn" field.

<imm8> For encoding T2: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.
For encoding T5: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.
If <Rn> is the LR, and zero is used, see ERET.

If omitted, this register is the same as <r>. If the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation.
- This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.

- For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of this instruction unless <Rn> is
the LR.
For encoding T1, T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

For encoding A1 and T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see SUB, SUBS (SP minus immediate). If the PC is used, see ADR.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see SUB, SUBS (SP minus immediate).

<iimm3> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field.

<iimm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

<const> For encoding A1: an immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values.

For encoding T3: an immediate value. See Modified immediate constants in T32 instructions on page F1-7242 for the range of values.

In the T32 instruction set, MOVS{<c>}{<q>} PC, LR is a pseudo-instruction for SUB{<c>}{<q>} PC, LR, #0.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], NOT(imm32), '1');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.246  SUB, SUBS (register)

Subtract (register) subtracts an optionally-shifted register value from a register value, and writes the result to the
destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the
result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. However,
when the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see Pseudocode description of operations on
  the AArch32 general-purpose registers and the PC on page E1-7133.

- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm
deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from
    AArch32 state on page G1-8952.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>S</td>
<td>1101</td>
<td>Rn</td>
<td>imm5</td>
<td>stype</td>
<td>0</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SUB, rotate right with extend variant

Applies when S == 0 && imm5 == 00000 && stype == 11.

SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

SUB, shift or rotate by value variant

Applies when S == 0 && !(imm5 == 00000 && stype == 11).

SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

SUBS, rotate right with extend variant

Applies when S == 1 && imm5 == 00000 && stype == 11.

SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

SUBS, shift or rotate by value variant

Applies when S == 1 && !(imm5 == 00000 && stype == 11).

SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

if Rn == '1101' then SEE "SUB (SP minus register)";
  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
  (shift_t, shift_n) = DecodeImmShift(stype, imm5);
T1

```
T1 variant

SUB{<c>}{<q>} <Rd>, <Rn>, <Rm> // Inside IT block
SUBS{<q>} {<Rd>,} <Rn>, <Rm> // Outside IT block

Decode for this encoding

d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

```
T2 variant

SUB, rotate right with extend variant
Applies when S == 0 && imm3 == 000 && imm2 == 00 && stype == 11.
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

SUB, shift or rotate by value variant
Applies when S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11).
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

SUBS, rotate right with extend variant
Applies when S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11.
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX

SUBS, shift or rotate by value variant
Applies when S == 1 && !(imm3 == 000 && stype == 11) && Rd != 1111.
SUBS.W {<Rd>,} <Rn>, <Rm> // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

if Rd == '1111' && S == '1' then SEE "CMP (register)";
if Rn == '1101' then SEE "SUB (SP minus register)"
if d == '1010' then SEE "SUB (SP minus register)"
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

<
See Standard assembler syntax fields on page F1-7228.

<
For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:
• For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
• For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>.

For encoding T1 and T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<
For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. If the SP is used, see SUB, SUBS (SP minus register).
For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.
For encoding T2: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see SUB, SUBS (SP minus register).

<
For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<
Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
- LSL when stype = 00
- LSR when stype = 01
- ASR when stype = 10
- ROR when stype = 11

<
For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.
For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
  (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
  if d == 15 then // Can only occur for A32 encoding
    if setflags then
      ALUExceptionReturn(result);
    else
      ALUWritePC(result);
  else
    R[d] = result;
if setflags then
    PSTATE.<N,Z,C,V> = nzcv;

**Operational information**

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.247   SUB, SUBS (register-shifted register)

Subtract (register-shifted register) subtracts a register-shifted register value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

```
| 31 | 28|27|26|25|24|23|22|21|20|19|16|15|12|11| 8 | 7 | 6 | 5 | 4 | 3 | 0 |
| ! = 1111 | 0 | 0 | 0 | 0 | 1 | 0 | S | Rn | Rd | Rs | 0 |stype| 1 | Rm |
```

Flag setting variant
Applies when $S == 1$.

```
SUBS{<c>}{<q>}{<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting variant
Applies when $S == 0$.

```
SUB{<c>}{<q>}{<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding
```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Notes for all encodings
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
  - LSL when stype = 00
  - LSR when stype = 01
  - ASR when stype = 10
  - ROR when stype = 11
- `<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.
Operation

if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
   — The values of the data supplied in any of its registers.
   — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
   — The values of the data supplied in any of its registers.
   — The values of the NZCV flags.
F5.1.248  SUB, SUBS (SP minus immediate)

Subtract from SP (immediate) subtracts an immediate value from the SP value, and writes the result to the
destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the
result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the
destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see Pseudocode description of operations on
  the AArch32 general-purpose registers and the PC on page E1-7133.
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm
  deprecates use of this instruction. However, in this case:
  — The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  — The PE checks SPSR_ <current_mode> for an illegal return event. See Illegal return events from
    AArch32 state on page G1-8952.
  — The instruction is UNDEFINED in Hyp mode.
  — The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

\[
\begin{array}{cccccccccc}
\hline
1=111 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & S & 1 & 1 & 0 & 1 & Rd & \text{imm12} & |
\end{array}
\]

SUB variant
Applies when \( S = 0 \).

\[
\text{SUB}\{<c>\}{<q>} \{<Rd>,\} \ SP, \#\text{const}
\]

SUBS variant
Applies when \( S = 1 \).

\[
\text{SUBS}\{<c>\}{<q>} \{<Rd>,\} \ SP, \#\text{const}
\]

Decode for all variants of this encoding
\[
d = \text{UInt}(Rd); \ \text{setflags} = (S == '1'); \ \text{imm32} = \text{A32ExpandImm(imm12)};
\]

T1

\[
\begin{array}{cccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & | & 0 \\
\hline
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & \text{imm7}
\end{array}
\]

T1 variant

\[
\text{SUB}\{<c>\}{<q>} \{SP,\} \ SP, \#\text{imm7}
\]

Decode for this encoding
\[
d = 13; \ \text{setflags} = \text{FALSE}; \ \text{imm32} = \text{ZeroExtend(imm7:'00', 32)};
\]
T2

SUB variant
Applies when S == 0.

SUB{<c>}.W {<Rd>,} SP, #<const> // <Rd>, <const> can be represented in T1
SUB{<c>}{<q>} {<Rd>,} SP, #<const>

SUBS variant
Applies when S == 1 && Rd != 1111.

SUBS{<c>}{<q>} {<Rd>,} SP, #<const>

Decode for all variants of this encoding
if Rd == '1111' && S == '1' then SEE "CMP (immediate)";
d = UInt(Rd); setflags = (S == '1'); imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 && !setflags then UNPREDICTABLE;

T3

T3 variant
SUB{<c>}{<q>} {<Rd>,} SP, #<imm12> // <imm12> cannot be represented in T1, T2, or T3
SUBW{<c>}{<q>} {<Rd>,} SP, #<imm12> // <imm12> can be represented in T1, T2, or T3

Decode for this encoding

d = UInt(Rd); setflags = FALSE; imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;

Notes for all encodings
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

SP, Is the stack pointer.

<imm7> Is the unsigned immediate, a multiple of 4, in the range 0 to 508, encoded in the "imm7" field as <imm7>/4.
<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. If the PC is used:
- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_<current_mode>. Arm deprecates use of this instruction unless <Rn> is the LR.
For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

<imm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

<const> For encoding A1: an immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values.
For encoding T2: an immediate value. See Modified immediate constants in T32 instructions on page F1-7242 for the range of values.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    \( (result, nzcv) = \text{AddWithCarry}(R[13], \text{NOT}(\text{imm32}), '1') \);
    if d == 15 then  // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
F5.1.249   SUB, SUBS (SP minus register)

Subtract from SP (register) subtracts an optionally-shifted register value from the SP value, and writes the result to
the destination register.

If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the
result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the
destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see Pseudocode description of operations on
  the AArch32 general-purpose registers and the PC on page E1-7133.
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm
deprecates use of this instruction. However, in this case:
  — The PE branches to the address written to the PC, and restores PSTATE from SPSR_<current_mode>.
  — The PE checks SPSR_<current_mode> for an illegal return event. See Illegal return events from
    AArch32 state on page G1-8952.
  — The instruction is UNDEFINED in Hyp mode.
  — The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1

| 31 | 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15 | 12| 11 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| l=1111 | 0 | 0 | 0 | 0 | 1 | 0 | S | 1 | 1 | 0 | 1 | Rd | imm5 | stype | 0 | Rm |
| cond |

SUB, rotate right with extend variant

Applies when S == 0 && imm5 == 00000 && stype == 11.
SUB{<c>}{<q>} {<Rd>,} SP, <Rm> , RRX

SUB, shift or rotate by value variant

Applies when S == 0 && !(imm5 == 00000 && stype == 11).
SUB{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}

SUBS, rotate right with extend variant

Applies when S == 1 && imm5 == 00000 && stype == 11.
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm> , RRX

SUBS, shift or rotate by value variant

Applies when S == 1 && !(imm5 == 00000 && stype == 11).
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
T32 and A32 Base Instruction Set Instruction Descriptions

F5.1 Alphabetical list of T32 and A32 base instruction set instructions

T1

SUB, rotate right with extend variant
Applies when S == 0 \&\& imm3 == 000 \&\& imm2 == 00 \&\& stype == 11.
SUB{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX

SUB, shift or rotate by value variant
Applies when S == 0 \&\& !(imm3 == 000 \&\& imm2 == 00 \&\& stype == 11).
SUB{<c>}'W {<Rd>,} SP, <Rm> // <Rd>, <Rm> can be represented in T1 or T2
SUB{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}

SUBS, rotate right with extend variant
Applies when S == 1 \&\& imm3 == 000 \&\& Rd != 1111 \&\& imm2 == 00 \&\& stype == 11.
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX

SUBS, shift or rotate by value variant
Applies when S == 1 \&\& !(imm3 == 000 \&\& imm2 == 00 \&\& stype == 11) \&\& Rd != 1111.
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding
if Rd == '1111' \&\& S == '1' then SEE "CMP (register)"
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings
For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<<> See Standard assembler syntax fields on page F1-7228.
<q> See Standard assembler syntax fields on page F1-7228.
<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used:
- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see Pseudocode description of operations on the AArch32 general-purpose registers and the PC on page E1-7133.
- For the SUBS variant, the instruction performs an exception return, that restores PSTATE from SPSR_{<current_mode>}. For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.
<Rm>       For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
       For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.

<shift>    Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
    LSL     when stype = 00
    LSR     when stype = 01
    ASR     when stype = 10
    ROR     when stype = 11

<amount>   For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.
       For encoding T1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

**Operation for all encodings**

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[13], NOT(shifted), '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```
### F5.1.250 SVC

Supervisor Call causes a Supervisor Call exception. For more information, see *Supervisor Call (SVC) exception* on page G1-8968.

--- **Note** ---

SVC was previously called SWI, Software Interrupt, and this name is still found in some documentation.

Software can use this instruction as a call to an operating system to provide a service.

In the following cases, the Supervisor Call exception generated by the SVC instruction is taken to Hyp mode:

- If the SVC is executed in Hyp mode.
- If HCR.TGE is set to 1, and the SVC is executed in Non-secure User mode. For more information, see *Supervisor Call exception, when the value of HCR.TGE is 1* on page G1-8945.

In these cases, the HSR identifies that the exception entry was caused by a Supervisor Call exception, EC value 0x11, see *Use of the HSR* on page G5-9265. The immediate field in the HSR:

- If the SVC is unconditional:
  - For the T32 instruction, is the zero-extended value of the imm8 field.
  - For the A32 instruction, is the least-significant 16 bits the imm24 field.
- If the SVC is conditional, is UNKNOWN.

#### A1

\[
\begin{array}{cccccccc}
\text{cond} & 28 & 27 & 26 & 25 & 24 & 23 & \text{imm24} \\
\text{1111} & 1 & 1 & 1 & 1 & & & \\
\end{array}
\]

**A1 variant**

SVC{<c>}{<q>} {#}<imm>

**Decode for this encoding**

\[
\text{imm32} = \text{ZeroExtend}(\text{imm24}, 32);
\]

#### T1

\[
\begin{array}{cccccccc}
\text{imm8} & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & & & & \\
\end{array}
\]

**T1 variant**

SVC{<c>}{<q>} {#}<imm>

**Decode for this encoding**

\[
\text{imm32} = \text{ZeroExtend}(\text{imm8}, 32);
\]
Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

>
See Standard assembler syntax fields on page F1-7228.

<imm>
For encoding A1: is a 24-bit unsigned immediate, in the range 0 to 16777215, encoded in the "imm24" field. This value is for assembly and disassembly only. SVC handlers in some systems interpret imm24 in software, for example to determine the required service.

For encoding T1: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. This value is for assembly and disassembly only. SVC handlers in some systems interpret imm8 in software, for example to determine the required service.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32.CheckForSVCTrap(imm32<15:0>);
    AArch32.CallSupervisor(imm32<15:0>);
F5.1.251   SXTAB

Signed Extend and Add Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

A1

```
|31| 28|27|26|25|24|23|22|21|20|19| 16|15| 12|11|10| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
```

cond  Rd  rotate(0)  0  1  1  1  Rm

A1 variant

SXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

if Rn == '1111' then SEE "SXTB";

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');

T1

```
|15|14|13|12|11|10| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
```

cond  Rd  rotate(0)  0  1  1  1  Rm

T1 variant

SXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

if Rn == '1111' then SEE "SXTB";

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- `<c>`  See Standard assembler syntax fields on page F1-7228.
- `<q>`  See Standard assembler syntax fields on page F1-7228.
- `<Rd>`  Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>`  Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>`  Is the second general-purpose source register, encoded in the "Rm" field.
- `<amount>`  Is the rotate amount, encoded in the "rotate" field. It can have the following values:

(omitted) when rotate = 00
8     when rotate = 01
16    when rotate = 10
24    when rotate = 11

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<7:0>, 32);
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.252 SXTAB16

Signed Extend and Add Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

A1

\[
\begin{array}{cccccccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
\text{cond} & \text{Rd} & \text{rotate} & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & Rm
\end{array}
\]

**A1 variant**

SXTAB16{<c>}{<q>}{<Rd>,} <Rn>, <Rm> {, ROR #<amount>}

**Decode for this encoding**

if Rn == '1111' then SEE "SXTB16'';

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');

T1

\[
\begin{array}{cccccccccccccccc}
| 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 |
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & Rn
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
| 15 & 14 & 13 & 12 & 11 & 8 & 7 & 6 & 5 & 4 & 3 & 0 |
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & Rm
\end{array}
\]

**T1 variant**

SXTAB16{<c>}{<q>}{<Rd>,} <Rn>, <Rm> {, ROR #<amount>}

**Decode for this encoding**

if Rn == '1111' then SEE "SXTB16'';

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');

if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRANGED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>`  See *Standard assembler syntax fields on page F1-7228.*
- `<q>`  See *Standard assembler syntax fields on page F1-7228.*
- `<Rd>`  Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>`  Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>`  Is the second general-purpose source register, encoded in the "Rm" field.
- `<amount>`  Is the rotate amount, encoded in the "rotate" field. It can have the following values:

```plaintext
(omitted) when rotate = 00
```
8 \quad \text{when rotate} = 01
16 \quad \text{when rotate} = 10
24 \quad \text{when rotate} = 11

**Operation for all encodings**

if ConditionPassed() then
  EncodingSpecificOperations();
  rotated = ROR(R[m], rotation);
  R[d]<15:0> = R[n]<15:0> + SignExtend(rotated<7:0>, 16);
  R[d]<31:16> = R[n]<31:16> + SignExtend(rotated<23:16>, 16);

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.253  SXTAH

Signed Extend and Add Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

A1

```
|31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4|3 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|!=1111 0 1 1 0 1 0 1 1 !|=1111| Rd rotate(0) 0 1 1 1| Rn |
```

A1 variant

$\text{SXTAH}(<c>){<q>}{<Rd>,} <Rn>, <Rm> {, ROR #<amount>}$

Decode for this encoding

if $\text{Rn} == '1111'$ then SEE "SXTH";

$d = \text{UInt}(\text{Rd}); n = \text{UInt}(\text{Rn}); m = \text{UInt}(\text{Rm}); \text{rotation} = \text{UInt}(\text{rotate:'000'});$

if $d == 15 || m == 15$ then UNPREDICTABLE;

T1

```
|15 14 13 12|11 10 9 8 7 6 5 4|3 0 |15 14 13 12|11 8 7 6 5 4|3 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|1 1 1 1 1 1 0 0 0|0 0 0 !|=1111 1 1 1| Rd 1 [0]rotate| Rm |
```

T1 variant

$\text{SXTAH}(<c>){<q>}{<Rd>,} <Rn>, <Rm> {, ROR #<amount>}$

Decode for this encoding

if $\text{Rn} == '1111'$ then SEE "SXTH";

$d = \text{UInt}(\text{Rd}); n = \text{UInt}(\text{Rn}); m = \text{UInt}(\text{Rm}); \text{rotation} = \text{UInt}(\text{rotate:'000'});$

if $d == 15 || m == 15$ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

$<c>$ See Standard assembler syntax fields on page F1-7228.

$<q>$ See Standard assembler syntax fields on page F1-7228.

$<Rd>$ Is the general-purpose destination register, encoded in the "Rd" field.

$<Rn>$ Is the first general-purpose source register, encoded in the "Rn" field.

$<Rm>$ Is the second general-purpose source register, encoded in the "Rm" field.

$<amount>$ Is the rotate amount, encoded in the "rotate" field. It can have the following values:

(omitted) when rotate = 00
8       when rotate = 01
16      when rotate = 10
24      when rotate = 11

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<15:0>, 32);

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.254 SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

A1

\[
\begin{array}{c|cccccccccccc}
& & & & & & & & & & & & \\
& & & & & & & & & & & & \\
\text{cond} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & \text{Rd} & \text{rotate}(0)(0) & 0 & 1 & 1 & 1 & \text{Rm} \\
\end{array}
\]

A1 variant

SXTB{<c>}{<q>}{<Rd>,}<Rm>{, ROR #<amount>}

Decode for this encoding

\[
d = \text{UInt}(\text{Rd}); \hspace{1em} m = \text{UInt}(\text{Rm}); \hspace{1em} \text{rotation} = \text{UInt}('000'); \\
\text{if} \ d == 15 || m == 15 \text{ then UNPREDICTABLE;}
\]

T1

\[
\begin{array}{c|cccccccccccc}
& & & & & & & & & & & & \\
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 3 & 2 & 0 \\
& & & & & & & & & & & & \\
1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & \text{Rm} & \text{Rd} \\
\end{array}
\]

T1 variant

SXTB{<c>}{<q>}{<Rd>,}<Rm>

Decode for this encoding

\[
d = \text{UInt}(\text{Rd}); \hspace{1em} m = \text{UInt}(\text{Rm}); \hspace{1em} \text{rotation} = 0;
\]

T2

\[
\begin{array}{c|cccccccccccc}
& & & & & & & & & & & & \\
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 14 & 13 & 12 & 11 & 8 & 7 & 6 & 5 & 4 & 3 & 0 \\
& & & & & & & & & & & & \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \text{Rd} & 1 & 0 & \text{rotate} & \text{Rm} \\
\end{array}
\]

T2 variant

SXTB{<c>}{<q>}{<Rd>,}<Rm> / / <Rd>,<Rm> can be represented in T1
SXTB{<c>}{<q>}{<Rd>,}<Rm>{, ROR #<amount>}

Decode for this encoding

\[
d = \text{UInt}(\text{Rd}); \hspace{1em} m = \text{UInt}(\text{Rm}); \hspace{1em} \text{rotation} = \text{UInt}('000'); \\
\text{if} \ d == 15 || m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1

Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<
> See Standard assembler syntax fields on page F1-7228.

<q>
> See Standard assembler syntax fields on page F1-7228.

<Rd>
Is the general-purpose destination register, encoded in the "Rd" field.

<Rm>
Is the general-purpose source register, encoded in the "Rm" field.

<amount>
Is the rotate amount, encoded in the "rotate" field. It can have the following values:

- omitted when rotate = 00
- 8 when rotate = 01
- 16 when rotate = 10
- 24 when rotate = 11

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<7:0>, 32);

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.255  **SXTB16**

Signed Extend Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

**A1**

```
|31| 28|27|26|25|24|23|22|21|20|19|18|17|16|15|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0 |
|==1111| 0| 1| 0| 1| 0| 0| 1| 1| 1| 1| Rd| rotate|0|0|0|0|1|1|1| Rm|
```

**A1 variant**

SXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad \text{rotation} = \text{UInt}(\text{rotate}:'000'); \\
\text{if } d == 15 \quad \text{||} \quad m == 15 \quad \text{then UNPREDICTABLE;}
\]

**T1**

```
|15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|15|14|13|12|11| 8| 7| 6| 5| 4| 3| 2| 1| 0| Rm|
|1|1|1|1|1|0|0|0|1|0|0|1|0|1|1|1|1|1|1|1|1|1|0|1|0|1|0|1|0| rotate| Rd |
```

**T1 variant**

SXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad \text{rotation} = \text{UInt}(\text{rotate}:'000'); \\
\text{if } d == 15 \quad \text{||} \quad m == 15 \quad \text{then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields on page F1-7228*.
- `<q>` See *Standard assembler syntax fields on page F1-7228*.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rm>` Is the general-purpose source register, encoded in the "Rm" field.
- `<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:
  - (omitted) when rotate = 00
  - 8 when rotate = 01
  - 16 when rotate = 10
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0> = SignExtend(rotated<7:0>, 16);
    R[d]<31:16> = SignExtend(rotated<23:16>, 16);

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.256  SXTH

Signed Extend Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

A1

\[
\begin{array}{cccccccccc}
! = 1111 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & R d & \text{rotate} & 0 & 0 & 1 & 1 & R m & \\
\end{array}
\]

**cond**

\[\text{A1 variant}\]

\[\text{SXTH}\{<c>\}{<q>\} \{<Rd>,\} <Rm> \{, \text{ROR} \#\text{<amount>}\}\]

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad \text{rotation} = \text{UInt}(\text{rotate}:'000');
\]

\[\text{if} \quad d == 15 || m == 15 \text{ then UNPREDICTABLE;}
\]

T1

\[
\begin{array}{cccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 3 & 2 & 0 & \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & R m & R d & \\
\end{array}
\]

**T1 variant**

\[\text{SXTH}\{<c>\}{<q>\} \{<Rd>,\} <Rm>\]

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad \text{rotation} = 0;
\]

T2

\[
\begin{array}{cccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & R d & 1 & 0 & \text{rotate} & R m & \\
\end{array}
\]

**T2 variant**

\[\text{SXTH}\{<c>\}.W \{<Rd>,\} <Rm> // <Rd>, <Rm> \text{ can be represented in } T1\]

\[\text{SXTH}\{<c>\}{<q>\} \{<Rd>,\} <Rm> \{, \text{ROR} \#\text{<amount>}\}\]

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad \text{rotation} = \text{UInt}(\text{rotate}:'000');
\]

\[\text{if} \quad d == 15 || m == 15 \text{ then UNPREDICTABLE;} // \text{Armv8-A removes UNPREDICTABLE for R13}\]
Notes for all encodings

For more information about the CONstrained UNPredICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<c>  
See Standard assembler syntax fields on page F1-7228.

<q>  
See Standard assembler syntax fields on page F1-7228.

<Rd>  
Is the general-purpose destination register, encoded in the "Rd" field.

<Rm>  
Is the general-purpose source register, encoded in the "Rm" field.

<amount>  
Is the rotate amount, encoded in the "rotate" field. It can have the following values:

- (omitted) when rotate = 00
- 8 when rotate = 01
- 16 when rotate = 10
- 24 when rotate = 11

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<15:0>, 32);

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.257   TBB, TBH

Table Branch Byte or Halfword causes a PC-relative forward branch using a table of single byte or halfword offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value returned from the table.

T1

\[
\begin{array}{cccccccccccccccccccc}
0 & 1 & 5 & 1 & 4 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & Rn & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & H & Rm
\end{array}
\]

**Byte variant**

Applies when \( H = 0 \).

\[ \text{TBB} \{\text{<c>}}\{\text{<q>}} \ [\text{<Rn>}, \text{<Rm>}} ] \] // Outside or last in IT block

**Halfword variant**

Applies when \( H = 1 \).

\[ \text{TBH} \{\text{<c>}}\{\text{<q>}} \ [\text{<Rn>}, \text{<Rm>}, \text{LSL} \#1] \] // Outside or last in IT block

**Decode for all variants of this encoding**

\[
n = \text{UInt}(\text{Rn}); \quad m = \text{UInt}(\text{Rm}); \quad \text{is_tbh} = (H == '1');
\]

if \( m = 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

if InITBlock() && !LastInITBlock() then UNPREDICTABLE;

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

\(<\text{c}>\) See Standard assembler syntax fields on page F1-7228.

\(<\text{q}>\) See Standard assembler syntax fields on page F1-7228.

\(<\text{Rn}>\) Is the general-purpose base register holding the address of the table of branch lengths, encoded in the "Rn" field. The PC can be used. If it is, the table immediately follows this instruction.

\(<\text{Rm}>\) For the byte variant: is the general-purpose index register, encoded in the "Rm" field. This register contains an integer pointing to a single byte in the table. The offset in the table is the value of the index.

For the halfword variant: is the general-purpose index register, encoded in the "Rm" field. This register contains an integer pointing to a halfword in the table. The offset in the table is twice the value of the index.

**Operation**

if \( \text{ConditionPassed()} \) then

\[
\begin{align*}
\text{EncodingSpecificOperations();} \\
\text{integer halfwords;} \\
\text{if is_tbh then} \\
\quad \text{halfwords} = \text{UInt(MemU}[R[n]+\text{LSL}(R[m],1)], 2)); \\
\text{else}
\end{align*}
\]
halfwords = UInt(MemU[R[n]+R[m], 1]);
BranchWritePC(PC + 2*halfwords, BranchType_INDIR);
TEQ (immediate)

Test Equivalence (immediate) performs a bitwise exclusive OR operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

**A1**

![A1 Instruction Format]

**A1 variant**

TEQ{<c>}{<q}> <Rn>, #<const>

*Decode for this encoding*

\[
n = \text{UInt}(Rn); \\
\text{(imm32, carry)} = A32ExpandImm_C(imm12, PSTATE.C);
\]

**T1**

![T1 Instruction Format]

**T1 variant**

TEQ{<c>}{<q}> <Rn>, #<const>

*Decode for this encoding*

\[
n = \text{UInt}(Rn); \\
\text{(imm32, carry)} = T32ExpandImm_C(i:imm3:imm8, PSTATE.C); \\
\text{if } n == 15 \text{ then UNPREDICTABLE; } // \text{Armv8-A removes UNPREDICTABLE for R13}
\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>`: See Standard assembler syntax fields on page F1-7228.
- `<q>`: See Standard assembler syntax fields on page F1-7228.
- `<Rn>`: For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For encoding T1: is the general-purpose source register, encoded in the "Rn" field.
- `<const>`: For encoding A1: an immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values. For encoding T1: an immediate value. See Modified immediate constants in T32 instructions on page F1-7242 for the range of values.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.259   TEQ (register)

Test Equivalence (register) performs a bitwise exclusive OR operation on a register value and an optionally-shifted
register value. It updates the condition flags based on the result, and discards the result.

A1

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>[0]</td>
<td>[0]</td>
<td>[0]</td>
<td>imm5</td>
<td>stype</td>
<td>0</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Rotate right with extend variant**

Applies when \( \text{imm5} == 00000 \) & \( \text{stype} == 11 \).

TEQ\{<c>{<q>}\} <Rn>, <Rm>, RRX

**Shift or rotate by value variant**

Applies when \( \neg(\text{imm5} == 00000 \) & \( \text{stype} == 11 \).

TEQ\{<c>{<q>}\} <Rn>, <Rm> {:, <shift> #<amount>}

**Decode for all variants of this encoding**

\[ n = \text{UInt}(\text{Rn}); \quad m = \text{UInt}(\text{Rm}); \]
\[ (\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{imm5}); \]

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | Rn | [0] | imm3 | 1 | 1 | 1 | imm2 | stype | Rm |

**Rotate right with extend variant**

Applies when \( \text{imm3} == 000 \) & \( \text{imm2} == 00 \) & \( \text{stype} == 11 \).

TEQ\{<c>{<q>}\} <Rn>, <Rm>, RRX

**Shift or rotate by value variant**

Applies when \( \neg(\text{imm3} == 000 \) & \( \text{imm2} == 00 \) & \( \text{stype} == 11 \).

TEQ\{<c>{<q>}\} <Rn>, <Rm> {:, <shift> #<amount>}

**Decode for all variants of this encoding**

\[ n = \text{UInt}(\text{Rn}); \quad m = \text{UInt}(\text{Rm}); \]
\[ (\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{imm3}:\text{imm2}); \]
\[ \text{if } n == 15 \text{ || } m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13} \]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1
Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<c>
See Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

<Rn>
For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.

<Rm>
For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.

<shift>
Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
- LSL when stype = 00
- LSR when stype = 01
- ASR when stype = 10
- ROR when stype = 11

<amount>
For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.
For encoding T1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] EOR shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.260   TEQ (register-shifted register)

Test Equivalence (register-shifted register) performs a bitwise exclusive OR operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

A1

| [31] | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 14 | 13 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=1111 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | Rn | 0 | 0 | 0 | 0 | Rs | 0 | stype | 1 | Rm |

A1 variant

TEQ{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>

Decode for this encoding

\[
\begin{align*}
  n &= \text{UInt}(Rn); \\
  m &= \text{UInt}(Rm); \\
  s &= \text{UInt}(Rs); \\
  \text{shift}_t &= \text{DecodeRegShift}(\text{stype}); \\
  \text{if } n == 15 || m == 15 || s == 15 \text{ then UNPREDICTABLE;}
\end{align*}
\]

Notes for all encodings

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- \(<c>\) See Standard assembler syntax fields on page F1-7228.
- \(<q>\) See Standard assembler syntax fields on page F1-7228.
- \(<Rn>\) Is the first general-purpose source register, encoded in the "Rn" field.
- \(<Rm>\) Is the second general-purpose source register, encoded in the "Rm" field.
- \(<\text{type}>\) Is the type of shift to be applied to the second source register, encoded in the "sttype" field. It can have the following values:
  - LSL when \(\text{stype} = 00\)
  - LSR when \(\text{stype} = 01\)
  - ASR when \(\text{stype} = 10\)
  - ROR when \(\text{stype} = 11\)
- \(<Rs>\) Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

\[
\begin{align*}
  \text{if } \text{ConditionPassed()} \text{ then} \\
  \text{EncodingSpecificOperations();} \\
  \text{shift}_n &= \text{UInt}(R[s]<7:0>); \\
  (\text{shifted, carry}) &= \text{Shift}_C(R[n], \text{shift}_t, \text{shift}_n, \text{PSTATE.C}); \\
  \text{result} &= R[n] \text{ EOR shifted;} \\
  \text{PSTATE.N} &= \text{result}<31>; \\
  \text{PSTATE.Z} &= \text{IsZeroBit(result)}; \\
  \text{PSTATE.C} &= \text{carry}; \\
  // \text{PSTATE.V} \text{ unchanged}
\end{align*}
\]
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.261   TSB CSYNC

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions. If FEAT_TRF is not implemented, this instruction executes as a NOP.

A1

(FEAT_TRF)

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| l=1111 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

cond

A1 variant

TSB{<c>}{<q>} CSYNC

Decode for this encoding

if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
if cond != '1110' then UNPREDICTABLE;              // TSB must be encoded with AL condition

CONSTRANDED UNPREDICTABLE behavior

If cond != '1110', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes unconditionally.
• The instruction executes conditionally.

T1

(FEAT_TRF)

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

T1 variant

TSB{<c>}{<q>} CSYNC

Decode for this encoding

if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
if InITBlock() then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The instruction executes unconditionally.
• The instruction executes conditionally.

Assembler symbols

<\> See Standard assembler syntax fields on page F1-7228.

<\&> See Standard assembler syntax fields on page F1-7228.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    TraceSynchronizationBarrier();
F5.1.262   TST (immediate)

Test (immediate) performs a bitwise AND operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

A1

\[
\begin{array}{cccccccccccccccc}
| & | & | & | \hline
|\text{!=1111} & 0 & 0 & 1 & 1 & 0 & 0 & 1 & \text{Rn} & \text{imm12} & | & | & 0 & | \\
\text{cond} & & & & & & & & & & & & & & & \\
\end{array}
\]

A1 variant

\[
\text{TST\{<c>\}{<q>} <Rn>, #<const>}
\]

Decode for this encoding

\[
n = \text{UInt}(\text{Rn});
\]
\[
(\text{imm32}, \text{carry}) = \text{A32ExpandImm}\_C(\text{imm12}, \text{PSTATE.C});
\]

T1

\[
\begin{array}{cccccccccccccccc}
| & | & | & | \hline
|1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & \text{Rn} & 0 & \text{imm3} & 1 & 1 & 1 & \text{imm8} & | & | & 0 & | \\
\end{array}
\]

T1 variant

\[
\text{TST\{<c>\}{<q>} <Rn>, #<const>}
\]

Decode for this encoding

\[
n = \text{UInt}(\text{Rn});
\]
\[
(\text{imm32}, \text{carry}) = \text{T32ExpandImm}\_C(\text{i:imm3:imm8}, \text{PSTATE.C});
\]
\[
\text{if } n = 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\) See Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<\text{Rn}\)> For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

\(<\text{const}\)> For encoding A1: an immediate value. See Modified immediate constants in A32 instructions on page F1-7244 for the range of values.

For encoding T1: an immediate value. See Modified immediate constants in T32 instructions on page F1-7242 for the range of values.
Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  result = R[n] AND imm32;
  PSTATE.N = result<31>;
  PSTATE.Z = IsZeroBit(result);
  PSTATE.C = carry;
  // PSTATE.V unchanged

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.263  TST (register)

Test (register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

### A1

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1=111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm5</td>
<td>stype</td>
<td>0</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Rotate right with extend variant**

Applies when imm5 == 00000 & stype == 11.

TST{<c>}{<q>} <Rn>, <Rm>, RRX

**Shift or rotate by value variant**

Applies when !(imm5 == 00000 & stype == 11).

TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

**Decode for all variants of this encoding**

\[ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \]
\[ (\text{shift}_t, \text{shift}_n) = \text{DecodeImmShift}(\text{stype}, \text{imm5}); \]

### T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rm</td>
<td>Rn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**T1 variant**

TST{<c>}{<q>} <Rn>, <Rm>

**Decode for this encoding**

\[ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \]
\[ (\text{shift}_t, \text{shift}_n) = (\text{SRType}_{LSL}, 0); \]

### T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>0</td>
</tr>
</tbody>
</table>

**Rotate right with extend variant**

Applies when imm3 == 000 & imm2 == 00 & stype == 11.

TST{<c>}{<q>} <Rn>, <Rm>, RRX
Shift or rotate by value variant

Applies when !(imm3 == 000 && imm2 == 00 && stype == 11).

TST{<c>}.W <Rn>, <Rm> // <Rn>, <Rm> can be represented in T1
TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

>
See Standard assembler syntax fields on page F1-7228.

<Rn>
For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
For encoding T1 and T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm>
For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift>
Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:

<table>
<thead>
<tr>
<th>Shift Type</th>
<th>stype</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL</td>
<td>00</td>
</tr>
<tr>
<td>LSR</td>
<td>01</td>
</tr>
<tr>
<td>ASR</td>
<td>10</td>
</tr>
<tr>
<td>ROR</td>
<td>11</td>
</tr>
</tbody>
</table>

<amount>
For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.
For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.264 TST (register-shifted register)

Test (register-shifted register) performs a bitwise AND operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

A1

| 31  | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 14 | 13 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| ![cond] | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | Rn | 0 | 0 | 0 | 0 | Rs | 0 | stype | 1 | Rm |

A1 variant

TST{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>

Decode for this encoding

\[
n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad s = \text{UInt}(Rs);
\]
\[
\text{shift}_t = \text{DecodeRegShift(stype)};
\]
\[
\text{if } n == 15 || m == 15 || s == 15 \text{ then UNPREDICTABLE};
\]

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
- `<type>` Is the type of shift to be applied to the second source register, encoded in the "stype" field. It can have the following values:
  - LSL when stype = 00
  - LSR when stype = 01
  - ASR when stype = 10
  - ROR when stype = 11
- `<Rs>` Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

if ConditionPassed() then

EncodingSpecificOperations();

shift_n = UInt(R[s]<7:0>);
(shifted, carry) = Shift_C(R[n], shift_t, shift_n, PSTATE.C);
result = R[n] AND shifted;
PSTATE.N = result<31>;
PSTATE.Z = IsZeroBit(result);
PSTATE.C = carry;
// PSTATE.V unchanged
**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.265   UADD16

Unsigned Add 16 performs two 16-bit unsigned integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

A1

|31 28|27 26 25|24|23|22|21|20|19|16|15|12|11|10|9|8|7|6|5|4|3|0 |
| 1=1111 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | Rn | Rd | (1)(1)(1)(1) | 0 | 0 | 0 | 1 | Rm |

**A1 variant**

UADD16{<c>}{<q>}{<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

|15|14|13|12|11|10|9|8|7|6|5|4|3|0|15|14|13|12|11|8|7|6|5|4|3|0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | Rn | 1 | 1 | 1 | 1 | Rd | 0 | 1 | 0 | 0 | Rm |

**T1 variant**

UADD16{<c>}{<q>}{<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

<
See Standard assembler syntax fields on page F1-7228.

<
See Standard assembler syntax fields on page F1-7228.

<Rd>
Is the general-purpose destination register, encoded in the "Rd" field.

<Rn>
Is the first general-purpose source register, encoded in the "Rn" field.

<Rm>
Is the second general-purpose source register, encoded in the "Rm" field.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
\[
\text{sum2} = \text{UInt}(R[n]<31:16>) + \text{UInt}(R[m]<31:16>);
\]
\[
R[d]<15:0> = \text{sum1}<15:0>;
\]
\[
R[d]<31:16> = \text{sum2}<15:0>;
\]
\[
PSTATE.GE<1:0> = \text{if sum1} >= 0x10000 \text{ then '11' else '00'};
\]
\[
PSTATE.GE<3:2> = \text{if sum2} >= 0x10000 \text{ then '11' else '00'};
\]

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.266 UADD8

Unsigned Add 8 performs four unsigned 8-bit integer additions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the additions.

A1

A1 variant

UADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

if \( d = 15 \) || \( n = 15 \) || \( m = 15 \) then UNPREDICTABLE;

T1

T1 variant

UADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

if \( d = 15 \) || \( n = 15 \) || \( m = 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\) See Standard assembler syntax fields on page F1-7228.
\(<q>\) See Standard assembler syntax fields on page F1-7228.
\(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.
\(<Rn>\) Is the first general-purpose source register, encoded in the "Rn" field.
\(<Rm>\) Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

\[
\text{if \text{ConditionPassed}() then}
\quad \text{EncodingSpecificOperations();}
\quad \text{sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);}
\]
sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
R[d]<7:0> = sum1<7:0>;
R[d]<15:8> = sum2<7:0>;
R[d]<23:16> = sum3<7:0>;
R[d]<31:24> = sum4<7:0>;
PSTATE.GE<0>  = if sum1 >= 0x100 then '1' else '0';
PSTATE.GE<1>  = if sum2 >= 0x100 then '1' else '0';
PSTATE.GE<2>  = if sum3 >= 0x100 then '1' else '0';
PSTATE.GE<3>  = if sum4 >= 0x100 then '1' else '0';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.267   **UASX**

Unsigned Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, and writes the results to the destination register. It sets PSTATE.GE according to the results.

**A1**

| 31  | 28|27 26 25 24|23 22 21 20|19  |16|15  |12|11  |10  |9  |8  |7  |6  |5  |4  |3  |0  |
|-----|----|-------------|-------------|----|---|----|---|----|----|---|---|---|---|---|---|---|---|---|
| !=111 | 0  |1  |0  |0  |1  |0  |Rn |----|----|----|----|----|----|----|----|----|----|
| cond  | Rd |----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|

**A1 variant**

UASX{<c>}{<q>}{<Rd>,} <Rn>, <Rm>

*Decode for this encoding*

\[ d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \]
\[ \text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE; } \]

**T1**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**T1 variant**

UASX{<c>}{<q>}{<Rd>,} <Rn>, <Rm>

*Decode for this encoding*

\[ d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm); \]
\[ \text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE; } // \text{ Armv8-A removes UNPREDICTABLE for R13} \]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields on page F1-7228.*
- `<q>` See *Standard assembler syntax fields on page F1-7228.*
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d]<15:0>  = diff<15:0>;
    R[d]<31:16> = sum<15:0>;
    PSTATE.GE<1:0> = if diff >= 0 then '11' else '00';
    PSTATE.GE<3:2> = if sum  >= 0x10000 then '11' else '00';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.268   UBFX

Unsigned Bit Field Extract extracts any number of adjacent bits at any position from a register, zero-extends them to 32 bits, and writes the result to the destination register.

A1

| 31 28|27 26 25 24|23 22 21 20| 16|15 | 12|11 |  7 6 5 4 |  3 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| !|=1111| 0 1|1 1|1|1|1| widthm1 | Rd | lsb | 1 0 |1 | Rn |
| cond

**A1 variant**

UBFX(<c>{<q>}:<Rd>, <Rn>, #<lsb>, #<width>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn);
\]
\[
lsbit = \text{UInt}(\text{imm3:imm2}); \quad \text{widthminus1} = \text{UInt}(\text{widthm1});
\]
\[
msbit = lsbit + widthminus1;
\]
\[
\text{if } d == 15 || n == 15 \text{ then UNPREDICTABLE;}
\]
\[
\text{if } msbit > 31 \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \(msbit > 31\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

| 15 14 13 12|11 10 | 9 8 | 7 6 5 4 | 3 | 0 | 15 14 12|11 | 8 7 6 5 4 | 0 |
| 1 1 1 1 0 | 0 | 1 1 1 1 | 0 | 0 | Rn | 0 | imm3 | Rd | imm2|0 | widthm1 |

**T1 variant**

UBFX(<c>{<q>}:<Rd>, <Rn>, #<lsb>, #<width>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn);
\]
\[
lsbit = \text{UInt}(\text{imm3:imm2}); \quad \text{widthminus1} = \text{UInt}(\text{widthm1});
\]
\[
msbit = lsbit + widthminus1;
\]
\[
\text{if } d == 15 || n == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]
\[
\text{if } msbit > 31 \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \(msbit > 31\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
• The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<\_c\_> See Standard assembler syntax fields on page F1-7228.

<\_q\_> See Standard assembler syntax fields on page F1-7228.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the general-purpose source register, encoded in the "Rn" field.

<lsb> For encoding A1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "lsb" field.

For encoding T1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field.

<width> Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    R[d] = ZeroExtend(R[n]<msbit:lsbit>, 32);

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.269   UDF

Permanently Undefined generates an Undefined Instruction exception.

The encodings for UDF used in this section are defined as permanently UNDEFINED. However:

- With the T32 instruction set, Arm deprecates using the UDF instruction in an IT block.
- In the A32 instruction set, UDF is not conditional.

A1

| 31  28|27 26 25 24|23 22 21 20|19 | | | 0 |
| 1  1 0 0 1 1 1 1 1 1 imm12 | 1 1 1 imm4 |
| cond |

A1 variant

UDF{<c>}{<q>} {#}<imm>

Decode for this encoding

\[
\text{imm32} = \text{ZeroExtend}(\text{imm12}:\text{imm4}, 32);
\]

// imm32 is for assembly and disassembly only, and is ignored by hardware.

T1

| 15 14 13 12|11 10 9 8 7 | 0 |
| 1 1 0 1 1 1 1 0 imm8 |

T1 variant

UDF{<c>}{<q>} {#}<imm>

Decode for this encoding

\[
\text{imm32} = \text{ZeroExtend}(\text{imm8}, 32);
\]

// imm32 is for assembly and disassembly only, and is ignored by hardware.

T2

| 15 14 13 12|11 10 9 8 7 6 5 4 3 0 |15 14 13 12|11| | 0 |
| 1 1 1 0 1 1 1 1 1 1 imm4 | 1 0 1 0 imm12 |

T2 variant

UDF{<c>}.W {#}<imm> // <imm> can be represented in T1
UDF{<c>}{<q>} {#}<imm>

Decode for this encoding

\[
\text{imm32} = \text{ZeroExtend}(\text{imm4}:\text{imm12}, 32);
\]

// imm32 is for assembly and disassembly only, and is ignored by hardware.
Assembler symbols

<
> For encoding A1: see Standard assembler syntax fields on page F1-7228. < must be AL or omitted.
For encoding T1 and T2: see Standard assembler syntax fields on page F1-7228. Arm deprecates using any < value other than AL.

<
> See Standard assembler syntax fields on page F1-7228.

<imm>
For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the"imm12:imm4" field. The PE ignores the value of this constant.
For encoding T1: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. The PE ignores the value of this constant.
For encoding T2: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. The PE ignores the value of this constant.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    UNDEFINED;
F5.1.270   UDIV

Unsigned Divide divides a 32-bit unsigned integer register value by a 32-bit unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

See Divide instructions on page F2-7267 for more information about this instruction.

A1

<table>
<thead>
<tr>
<th>cond</th>
<th>Rd</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>1=1111</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>15</td>
<td>12</td>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>

A1 variant

UDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If Ra != '1111', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction performs a divide and the register specified by Ra becomes UNKNOWN.

T1

<table>
<thead>
<tr>
<th>cond</th>
<th>Rd</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>111110</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

T1 variant

UDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

CONSTRAINED UNPREDICTABLE behavior

If Ra != '1111', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction performs a divide and the register specified by Ra becomes UNKNOWN.
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register holding the dividend, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the divisor, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    integer result;
    if UInt(R[m]) == 0 then
        result = 0;
    else
        result = RoundTowardsZero(Real(UInt(R[n])) / Real(UInt(R[m])));
    R[d] = result<31:0>;
F5.1.271   UHADD16

Unsigned Halving Add 16 performs two unsigned 16-bit integer additions, halves the results, and writes the results to the destination register.

A1

A1 variant

UHADD16{<c>}{<q}> {<Rd>,} <Rn>, <Rm>

Decode for this encoding

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

T1 variant

UHADD16{<c>}{<q}> {<Rd>,} <Rn>, <Rm>

Decode for this encoding

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRANGED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

< >   See Standard assembler syntax fields on page F1-7228.
<q>   See Standard assembler syntax fields on page F1-7228.
<Rd>  Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>  Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>  Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
R[d]<15:0>  = sum1<16:1>;
R[d]<31:16> = sum2<16:1>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.272   UHADD8

Unsigned Halving Add 8 performs four unsigned 8-bit integer additions, halves the results, and writes the results to the destination register.

A1

\[
\begin{array}{|c|}
\hline
\text{[31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 6 5 4 3 0]}
\hline
1=1111 0 1 1 0 0 1 1 1 & \text{cond} & \hline
\text{Rn} & \text{Rd} & (1)(1)(1)(1) & 1 & 0 & 0 & 1 \ \text{Rm}
\hline
\end{array}
\]

A1 variant

UHADD8{<c>{<q>}{<Rd>,} <Rn>, <Rm>}

Decode for this encoding

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
\]

T1

\[
\begin{array}{|c|}
\hline
\text{[15 14 13 12 11 10 9 8 7 6 5 4 3 0]}
\hline
1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & \text{cond} & 0 & \text{Rn} & 1 & 1 & 1 & \text{Rd} & 0 & 1 & 1 & \text{Rm}
\hline
\end{array}
\]

T1 variant

UHADD8{<c>{<q>}{<Rd>,} <Rn>, <Rm>}

Decode for this encoding

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
\]

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- \(<c>\) See Standard assembler syntax fields on page F1-7228.
- \(<q>\) See Standard assembler syntax fields on page F1-7228.
- \(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.
- \(<Rn>\) Is the first general-purpose source register, encoded in the "Rn" field.
- \(<Rm>\) Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

\[
\text{if ConditionPassed() then}
\begin{array}{l}
\text{EncodingSpecificOperations();}
\text{sum1} = \text{UInt}(R[n]<7:0>) + \text{UInt}(R[m]<7:0>);
\end{array}
\]
sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
R[d]<7:0> = sum1<8:1>;
R[d]<15:8> = sum2<8:1>;
R[d]<23:16> = sum3<8:1>;
R[d]<31:24> = sum4<8:1>;

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.273  UHASX

Unsigned Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, halves the results, and writes the results to the destination register.

A1

\[
\begin{array}{cccccccccccccccccccc}
!1111 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
\end{array}
\]

**A1 variant**

UHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

if \( d == 15 \) \( || \) \( n == 15 \) \( || \) \( m == 15 \) then UNPREDICTABLE;

T1

\[
\begin{array}{cccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 14 & 13 & 12 & 11 & 8 & 7 & 6 & 5 & 4 & 3 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & Rn & 1 & 1 & 1 & 0 & Rd & 0 & 1 & 1 & 0 & Rm \\
\end{array}
\]

**T1 variant**

UHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

if \( d == 15 \) \( || \) \( n == 15 \) \( || \) \( m == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

\(<c>\)  See Standard assembler syntax fields on page F1-7228.

\(<q>\)  See Standard assembler syntax fields on page F1-7228.

\(<\text{Rd}>\)  Is the general-purpose destination register, encoded in the "Rd" field.

\(<\text{Rn}>\)  Is the first general-purpose source register, encoded in the "Rn" field.

\(<\text{Rm}>\)  Is the second general-purpose source register, encoded in the "Rm" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d]<15:0> = diff<16:1>;
    R[d]<31:16> = sum<16:1>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.274   UHSAX

Unsigned Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, halves the results, and writes the results to the destination register.

**A1**

```
| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 1 1 1 | 0 | 1 | 1 | 0 | 0 | 1 1 | Rn | Rd | 1 1 | 1 1 | 0 | 1 0 | Rm |
```

**A1 variant**

UHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \\
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE;}
\]

**T1**

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>1 1 1 1</td>
<td>Rd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**T1 variant**

UHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \\
\text{if } d == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` : See Standard assembler syntax fields on page F1-7228.
- `<q>` : See Standard assembler syntax fields on page F1-7228.
- `<Rd>` : Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` : Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` : Is the second general-purpose source register, encoded in the "Rm" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    sum = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    R[d]<15:0> = sum<16:1>;
    R[d]<31:16> = diff<16:1>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.275  **UHSUB16**

Unsigned Halving Subtract 16 performs two unsigned 16-bit integer subtractions, halves the results, and writes the results to the destination register.

### A1

```
<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>![cond]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**A1 variant**

UHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

```c
// d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

### T1

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rd</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**T1 variant**

UHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

```c
// d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

**Notes for all encodings**

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>`  See *Standard assembler syntax fields on page F1-7228*.
- `<q>`  See *Standard assembler syntax fields on page F1-7228*.
- `<Rd>`  Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>`  Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>`  Is the second general-purpose source register, encoded in the "Rm" field.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
```
diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
R[d]<15:0> = diff1<16:1>;
R[d]<31:16> = diff2<16:1>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.276   UHSUB8

Unsigned Halving Subtract 8 performs four unsigned 8-bit integer subtractions, halves the results, and writes the results to the destination register.

A1

```
|31| 28|27|26|25|24|23|22|21|20|19|16|15|12|11|10| 9| 8| 7| 6| 5| 4| 3| 0|
|   | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | | | | | | | | | | | | | |
```

**A1 variant**

UHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

```c
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

```
|15| 14| 13| 12|11|10| 9| 8| 7| 6| 5| 4| 3| 0|15|14|13|12|11| 8| 7| 6| 5| 4| 3| 0|
|1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | | | | | | | | | | | | | |
```

**T1 variant**

UHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

```c
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

**Notes for all encodings**

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
```
\[
\text{diff2} = \text{UInt}(R[n]<15:8>) - \text{UInt}(R[m]<15:8>);
\text{diff3} = \text{UInt}(R[n]<23:16>) - \text{UInt}(R[m]<23:16>);
\text{diff4} = \text{UInt}(R[n]<31:24>) - \text{UInt}(R[m]<31:24>);
\text{R}[d]<7:0> = \text{diff1}<8:1>;
\text{R}[d]<15:8> = \text{diff2}<8:1>;
\text{R}[d]<23:16> = \text{diff3}<8:1>;
\text{R}[d]<31:24> = \text{diff4}<8:1>;
\]

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**F5.1.277 UMAAL**

Unsigned Multiply Accumulate Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, adds two unsigned 32-bit values, and writes the 64-bit result to two registers.

**A1**

\[
\begin{array}{cccccccccccc}
| 31 & 28 & 27 & 26 & 25 & 24 & 23 & 22 & 21 & 20 & 19 & 16 | 15 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
\end{array}
\]

\[
\begin{array}{cccccccccc}
! = 1111 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \text{RdHi} & \text{RdLo} & \text{Rm} & 1 & 0 & 0 & 1 & \text{Rn}
\end{array}
\]

**A1 variant**

UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

**Decode for this encoding**

\[
dLo = \text{UInt}(\text{RdLo}); \quad dhI = \text{UInt}(\text{RdHi}); \quad n = \text{UInt}(\text{Rn}); \quad m = \text{UInt}(\text{Rm});
\]

\[
\text{if } dLo == 15 || dhI == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( dHi = dLo \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The value in the destination register is **UNKNOWN**.

**T1**

\[
\begin{array}{cccccccccccc}
| 15 & 14 & 13 & 12 | 11 & 10 & 9 & 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 & 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
\end{array}
\]

\[
\begin{array}{cccccccccc}
1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & \text{Rn} & \text{RdLo} & \text{RdHi} & 0 & 1 & 1 & 0 & \text{Rm}
\end{array}
\]

**T1 variant**

UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

**Decode for this encoding**

\[
dLo = \text{UInt}(\text{RdLo}); \quad dhI = \text{UInt}(\text{RdHi}); \quad n = \text{UInt}(\text{Rn}); \quad m = \text{UInt}(\text{Rm});
\]

\[
\text{if } dLo == 15 || dhI == 15 || n == 15 || m == 15 \text{ then UNPREDICTABLE;}
\]

// Armv8-A removes UNPREDICTABLE for R13
\[
\text{if } dhI == dLo \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( dHi = dLo \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The value in the destination register is **UNKNOWN**.
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<RdLo> Is the general-purpose source register holding the first addend and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

<RdHi> Is the general-purpose source register holding the second addend and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]) + UInt(R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.278  **UMLAL, UMLALS**

Unsigned Multiply Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

### A1

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3 1</td>
<td>2 8</td>
<td>2 7</td>
<td>2 6</td>
<td>2 5</td>
<td>2 4</td>
<td>2 3</td>
<td>2 2</td>
<td>2 1</td>
<td>2 0</td>
<td>1 9</td>
<td>1 8</td>
<td>1 7</td>
<td>1 6</td>
<td>1 5</td>
<td>1 2</td>
<td>1 1</td>
<td>1 8</td>
<td>7 6</td>
<td>5 4</td>
<td>3 0</td>
</tr>
<tr>
<td>1=1111</td>
<td>0 0 0 0</td>
<td>1 0 1</td>
<td>S</td>
<td>RdHi</td>
<td>RdLo</td>
<td>Rm</td>
<td>1 0 0 1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Flag setting variant**

Applies when $S = 1$.

UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

**Not flag setting variant**

Applies when $S = 0$.

UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

**Decode for all variants of this encoding**

\[
dLo = \text{UInt}(RdLo); \quad dHi = \text{UInt}(RdHi); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{setflags} = (S == '1');
\]

if $dLo == 15 || dHi == 15 || n == 15 || m == 15$ then UNPREDICTABLE;

if $dHi == dLo$ then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If $dHi == dLo$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 12 | 11 |  8 |  7 |  6 |  5 |  4 |  3 |  0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|----|----|----|---|---|---|---|---|---|---|---|
| 1  | 1  | 1  | 1  | 1  | 1  | 0 | 1 | 1 | 1 | 1 | 0 | Rn | RdLo | RdHi | 0 0 0 0 | Rm |

**T1 variant**

UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

**Decode for this encoding**

\[
dLo = \text{UInt}(RdLo); \quad dHi = \text{UInt}(RdHi); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{setflags} = \text{FALSE};
\]

if $dLo == 15 || dHi == 15 || n == 15 || m == 15$ then UNPREDICTABLE;

// Armv8-A removes UNPREDICTABLE for R13

if $dHi == dLo$ then UNPREDICTABLE;
**CONSTRAINED UNPREDICTABLE behavior**

If \( d_{Hi} = d_{Lo} \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The value in the destination register is **UNKNOWN**.

**Notes for all encodings**

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields* on page F1-7228.
- `<q>` See *Standard assembler syntax fields* on page F1-7228.
- `<RdLo>` Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
- `<RdHi>` Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
- `<Rn>` Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        PSTATE.N = result<63>;
        PSTATE.Z = IsZeroBit(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.279 UMULL, UMULLS

Unsigned Multiply Long multiplies two 32-bit unsigned values to produce a 64-bit result.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

A1

| 31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 8 7 6 5 4 3 0 |
| 1111 | 0 0 0 1 0 0 | S | RdHi | RdLo | Rm | 1 0 0 1 | Rn |

Flag setting variant

Applies when \( S = 1 \).

\( \text{UMULL}\{<c>\}{<q>} \ <RdLo>, \ <RdHi>, \ <Rn>, \ <Rm> \)

Not flag setting variant

Applies when \( S = 0 \).

\( \text{UMULL}\{<c>\}{<q>} \ <RdLo>, \ <RdHi>, \ <Rn>, \ <Rm> \)

Decode for all variants of this encoding

\[
dLo = \text{UInt}(\text{RdLo}); \quad dHi = \text{UInt}(\text{RdHi}); \quad n = \text{UInt}(\text{Rn}); \quad m = \text{UInt}(\text{Rm}); \quad \text{setflags} = (S == '1');
\]

if \( dLo == 15 || dHi == 15 || n == 15 || m == 15 \) then UNPREDICTABLE;

if \( dHi == dLo \) then UNPREDICTABLE;

CONstrained UNPREDICTABLE behavior

If \( dHi = dLo \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 15 12 11 8 7 6 5 4 3 0 |
| 1 1 1 1 1 0 1 1 1 0 1 0 | Rn | RdLo | RdHi | 0 0 0 0 | Rm |

T1 variant

\( \text{UMULL}\{<c>\}{<q>} \ <RdLo>, \ <RdHi>, \ <Rn>, \ <Rm> \)

Decode for this encoding

\[
dLo = \text{UInt}(\text{RdLo}); \quad dHi = \text{UInt}(\text{RdHi}); \quad n = \text{UInt}(\text{Rn}); \quad m = \text{UInt}(\text{Rm}); \quad \text{setflags} = \text{FALSE};
\]

if \( dLo == 15 || dHi == 15 || n == 15 || m == 15 \) then UNPREDICTABLE;

// Armv8-A removes UNPREDICTABLE for R13

if \( dHi == dLo \) then UNPREDICTABLE;
CONSTRANGED UNPREDICTABLE behavior

If \( d_{Hi} = d_{Lo} \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRANGED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- \(<c>\) See Standard assembler syntax fields on page F1-7228.
- \(<q>\) See Standard assembler syntax fields on page F1-7228.
- \(<RdLo>\) Is the general-purpose destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
- \(<RdHi>\) Is the general-purpose destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
- \(<Rn>\) Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- \(<Rm>\) Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation for all encodings

```c
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        PSTATE.N = result<63>;
        PSTATE.Z = IsZeroBit(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.280  UQADD16

Unsigned Saturating Add 16 performs two unsigned 16-bit integer additions, saturates the results to the 16-bit unsigned integer range 0 <= x <= 2^{16} - 1, and writes the results to the destination register.

A1

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1=1111</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>Rd</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

-cond

A1 variant

UQAD016{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 14 | 13 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | Rn | 1 | 1 | 1 | 1 | Rd | 0 | 1 | 0 | 1 | Rm |

T1 variant

UQAD016{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- <c> See Standard assembler syntax fields on page F1-7228.
- <q> See Standard assembler syntax fields on page F1-7228.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
R[d]<15:0> = UnsignedSat(sum1, 16);
R[d]<31:16> = UnsignedSat(sum2, 16);
F5.1.281  UQADD8

Unsigned Saturating Add 8 performs four unsigned 8-bit integer additions, saturates the results to the 8-bit unsigned integer range $0 \leq x \leq 2^8 - 1$, and writes the results to the destination register.

A1

```
[31  28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4 3 0 ]
  !=1111 0 1 1 0 0 1 0  Rn  Rd  (1)(1)(1)(1) 1 0 0 1  Rm
cond
```

A1 variant

UQADD8(<<c>>{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

$d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;

T1

```
[15 14 13 12|11 10 9 8 7 6 5 4 3 0 ]
  1 1 1 1 0 1 0 0 1 0 0 0  Rn  1 1 1 1  Rd  0 1 0 1  Rm
```

T1 variant

UQADD8(<<c>>{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

$d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<<c>>  See Standard assembler syntax fields on page F1-7228.

<<q>>  See Standard assembler syntax fields on page F1-7228.

<Rd>  Is the general-purpose destination register, encoded in the "Rd" field.

<Rn>  Is the first general-purpose source register, encoded in the "Rn" field.

<Rm>  Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
R[d]<7:0> = UnsignedSat(sum1, 8);
R[d]<15:8> = UnsignedSat(sum2, 8);
R[d]<23:16> = UnsignedSat(sum3, 8);
R[d]<31:24> = UnsignedSat(sum4, 8);
F5.1.282  UQASX

Unsigned Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, saturates the results to the 16-bit unsigned integer range \(0 \leq x \leq 2^{16} - 1\), and writes the results to the destination register.

### A1

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td>Rd</td>
<td></td>
<td>(1)</td>
<td></td>
<td>(1)</td>
<td></td>
<td>(1)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**cond**

#### A1 variant

\[
\text{UQASX}\{<c>}\{<q>\} \{<Rd>,\} \langle Rn\rangle, \langle Rm\rangle
\]

**Decode for this encoding**

\[
\text{d} = \text{UInt}(\text{Rd}); \quad \text{n} = \text{UInt}(\text{Rn}); \quad \text{m} = \text{UInt}(\text{Rm});
\]

\[
\text{if d} = 15 \quad \land \quad \text{n} = 15 \quad \land \quad \text{m} = 15 \quad \text{then UNPREDICTABLE;}
\]

#### T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 14 | 13 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | Rn | 1  | 1  | 1  | 1 | Rd | 0 | 1 | 0 | 1 | Rm |

#### T1 variant

\[
\text{UQASX}\{<c>\}{<q>} \langle <Rd>, \rangle \langle Rn\rangle, \langle Rm\rangle
\]

**Decode for this encoding**

\[
\text{d} = \text{UInt}(\text{Rd}); \quad \text{n} = \text{UInt}(\text{Rn}); \quad \text{m} = \text{UInt}(\text{Rm});
\]

\[
\text{if d} = 15 \quad \land \quad \text{n} = 15 \quad \land \quad \text{m} = 15 \quad \text{then UNPREDICTABLE; \quad Armv8-A removes UNPREDICTABLE for R13}
\]

### Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

### Assembler symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;c&gt;</td>
<td>See Standard assembler syntax fields on page F1-7228.</td>
</tr>
<tr>
<td>&lt;q&gt;</td>
<td>See Standard assembler syntax fields on page F1-7228.</td>
</tr>
<tr>
<td>&lt;Rd&gt;</td>
<td>Is the general-purpose destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>&lt;Rn&gt;</td>
<td>Is the first general-purpose source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>&lt;Rm&gt;</td>
<td>Is the second general-purpose source register, encoded in the &quot;Rm&quot; field.</td>
</tr>
</tbody>
</table>
Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations();
   diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
   sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
   R[d]<15:0>  = UnsignedSat(diff, 16);
   R[d]<31:16> = UnsignedSat(sum, 16);
F5.1.283 UQSAX

Unsigned Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$, and writes the results to the destination register.

A1

![Instruction Format]

**A1 variant**

UQSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

$$d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);$$

if $d == 15 || n == 15 || m == 15$ then UNPREDICTABLE;

T1

![Instruction Format]

**T1 variant**

UQSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

$$d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);$$

if $d == 15 || n == 15 || m == 15$ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields on page F1-7228*.
- `<q>` See *Standard assembler syntax fields on page F1-7228*.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    R[d]<15:0> = UnsignedSat(sum, 16);
    R[d]<31:16> = UnsignedSat(diff, 16);
F5.1.284 UQSUB16

Unsigned Saturating Subtract 16 performs two unsigned 16-bit integer subtractions, saturates the results to the 16-bit unsigned integer range 0 <= x <= 2^16 - 1, and writes the results to the destination register.

A1

```
| 31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 |7 6 5 4 |3 0 |
|-----|-------|-------|   |-----|-------|-------|-----|
| !=1111 | 0 1 1 0 0 1 0 1 | Rn | Rd | (1)(1)(1)(1) | 0 1 1 1 | Rm |
cond |
```

A1 variant

UQSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

```python
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

```
| 15 14 13 12|11 10 9 8 |7 6 5 4 |3 0 |15 14 13 12|11 8 7 6 5 4 |3 0 |
|-----|-------|-------|-----|-----|-------|-------|-----|
| 1 1 1 1 1 0 1 0 1 1 0 1 | Rn | 1 1 1 1 | Rd | 0 1 0 1 | Rm |
```

T1 variant

UQSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

```python
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
R[d]<15:0> = UnsignedSat(diff1, 16);
R[d]<31:16> = UnsignedSat(diff2, 16);
F5.1.285   UQSUB8

Unsigned Saturating Subtract 8 performs four unsigned 8-bit integer subtractions, saturates the results to the 8-bit unsigned integer range $0 \leq x \leq 2^8 - 1$, and writes the results to the destination register.

A1

```
| 31  | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=1111 | 0 | 1 | 0 | 0 | 1 | 0 | Rn | Rd | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | Rm |
cond
```

A1 variant

UQSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

```
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 0 | 1 | 0 | 1 | 0 | 0 | Rn | 1 | 1 | 1 | 1 | Rd | 0 | 1 | 0 | 1 | Rm |
```

T1 variant

UQSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

**Decode for this encoding**

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
```
$\text{diff2} = \text{UInt}(R[n]<15:8>) - \text{UInt}(R[m]<15:8>);$  
$\text{diff3} = \text{UInt}(R[n]<23:16>) - \text{UInt}(R[m]<23:16>);$  
$\text{diff4} = \text{UInt}(R[n]<31:24>) - \text{UInt}(R[m]<31:24>);$  
$R[d]<7:0> = \text{UnsignedSat}(\text{diff1}, 8);$  
$R[d]<15:8> = \text{UnsignedSat}(\text{diff2}, 8);$  
$R[d]<23:16> = \text{UnsignedSat}(\text{diff3}, 8);$  
$R[d]<31:24> = \text{UnsignedSat}(\text{diff4}, 8);$
F5.1.286   USAD8

Unsigned Sum of Absolute Differences performs four unsigned 8-bit subtractions, and adds the absolute values of the differences together.

A1

|31 | 28|27|26|25|24|23|22|21|20|19|16|15|14|13|12|11| 8| 7| 6| 5| 4| 3| 0 |

| Rd | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

cond

**A1 variant**

USAD8{<c>}{<q>}{<Rd>,} <Rn>, <Rm>

*Decode for this encoding*

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

if \( d == 15 \) || \( n == 15 \) || \( m == 15 \) then UNPREDICTABLE;

T1

|15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 0 |15|14|13|12|11| 8| 7| 6| 5| 4| 3| 0 |

| Rd | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**T1 variant**

USAD8{<c>}{<q>}{<Rd>,} <Rn>, <Rm>

*Decode for this encoding*

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

if \( d == 15 \) || \( n == 15 \) || \( m == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

*<>*  See Standard assembler syntax fields on page F1-7228.

*<q>*  See Standard assembler syntax fields on page F1-7228.

*Rd*  Is the general-purpose destination register, encoded in the "Rd" field.

*Rn*  Is the first general-purpose source register, encoded in the "Rn" field.

*Rm*  Is the second general-purpose source register, encoded in the "Rm" field.

**Operation for all encodings**

if \( \text{ConditionPassed()} \) then

\[
\text{EncodingSpecificOperations();}
\]

\[
\text{absdiff1} = \text{Abs}((\text{UInt}(R[n]<7:0>)) - \text{UInt}(R[m]<7:0>));
\]
absdiff2 = Abs(UInt(R[n]<15:8>) - UInt(R[m]<15:8>));
absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
result = absdiff1 + absdiff2 + absdiff3 + absdiff4;
R[d] = result<31:0>;
F5.1.287   USADA8

Unsigned Sum of Absolute Differences and Accumulate performs four unsigned 8-bit subtractions, and adds the absolute values of the differences to a 32-bit accumulate operand.

A1

```
| 31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 8 7 6 5 4 3 0 |
|-------------------------|---------------------|---------------------|
| !=1111                  | Rd                  | !!=1111             |
| cond                    | Ra                  | Rm                  |
|                         | !0 0 1 1 1 1 0 0 0   |
```

A1 variant

USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for this encoding

```
if Ra == '1111' then SEE "USAD8";
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 0 15 12</th>
<th>11 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 1 0 1 1 1</td>
<td>Rd</td>
<td>!!=1111</td>
</tr>
<tr>
<td></td>
<td>Rn</td>
<td>Rm</td>
</tr>
<tr>
<td></td>
<td>!0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
```

T1 variant

USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for this encoding

```
if Ra == '1111' then SEE "USAD8";
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Notes for all encodings

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
- `<Ra>` Is the third general-purpose source register holding the addend, encoded in the "Ra" field.
Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  absdiff1 = Abs(UInt(R[n]<7:0>)   - UInt(R[m]<7:0>));
  absdiff2 = Abs(UInt(R[n]<15:8>)  - UInt(R[m]<15:8>));
  absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
  absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
  result = UInt(R[a]) + absdiff1 + absdiff2 + absdiff3 + absdiff4;
  R[d] = result<31:0>;

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.288 USAT

Unsigned Saturate saturates an optionally-shifted signed value to a selected unsigned range.

This instruction sets PSTATE.Q to 1 if the operation saturates.

A1

Arithmetic shift right variant

Applies when sh == 1.

USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>

Logical shift left variant

Applies when sh == 0.

USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

Decode for all variants of this encoding

if sh == '1' && (imm3:imm2) == '00000' then SEE “USAT16”;

T1

Arithmetic shift right variant

Applies when sh == 1 && !(imm3 == 000 && imm2 == 00).

Logical shift left variant

Applies when sh == 0.

Decode for all variants of this encoding

if sh == '1' && (imm3:imm2) == '00000' then SEE “USAT16”;

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<
See Standard assembler syntax fields on page F1-7228.

>
See Standard assembler syntax fields on page F1-7228.

<Rd>
Is the general-purpose destination register, encoded in the "Rd" field.

<imm>
Is the bit position for saturation, in the range 0 to 31, encoded in the "sat_imm" field.

<Rn>
Is the general-purpose source register, encoded in the "Rn" field.

<amount>
For encoding A1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.

For encoding A1: is the shift amount, in the range 1 to 32 encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field.

For encoding T1: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, PSTATE.C);  // PSTATE.C ignored
    (result, sat) = UnsignedSatQ(SInt(operand), saturate_to);
    R[d] = ZeroExtend(result, 32);
    if sat then
        PSTATE.Q = '1';
F5.1.289   USAT16

Unsigned Saturate 16 saturates two signed 16-bit values to a selected unsigned range.
This instruction sets \texttt{PSTATE.Q} to 1 if the operation saturates.

A1

\begin{verbatim}
| 31  28 | 27  26  25 | 24  23  22  21  20| 19 | 16| 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|------------------|-----------|
| l=1111           | 0 | 1 | 1 | 0 | 1 | 1 | 0 | sat_imm | Rd | (1) | (1) | (1) | 0 | 0 | 1 | 1 | Rn |
\end{verbatim}

\textbf{A1 variant}

\texttt{USAT16\{<c>\}{<q>} <Rd>, #<imm>, <Rn>}

\textit{Decode for this encoding}

\begin{verbatim}
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
if d == 15 || n == 15 then UNPREDICTABLE;
\end{verbatim}

T1

\begin{verbatim}
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 14 | 13 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|------------------|-----------|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | Rn | 0 | 0 | 0 | Rd | 0 | 0 | 0 | 0 | sat_imm |
\end{verbatim}

\textbf{T1 variant}

\texttt{USAT16\{<c>\}{<q>} <Rd>, #<imm>, <Rn>}

\textit{Decode for this encoding}

\begin{verbatim}
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
\end{verbatim}

\textbf{Notes for all encodings}

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see \textit{Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors}.

\textbf{Assembler symbols}

- \texttt{<c>} \quad See \textit{Standard assembler syntax fields} on page F1-7228.
- \texttt{<q>} \quad See \textit{Standard assembler syntax fields} on page F1-7228.
- \texttt{<Rd>} \quad Is the general-purpose destination register, encoded in the "Rd" field.
- \texttt{<imm>} \quad Is the bit position for saturation, in the range 0 to 15, encoded in the "sat_imm" field.
- \texttt{<Rn>} \quad Is the general-purpose source register, encoded in the "Rn" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    (result1, sat1) = UnsignedSatQ(SInt(R[n]<15:0>), saturate_to);
    (result2, sat2) = UnsignedSatQ(SInt(R[n]<31:16>), saturate_to);
    R[d]<15:0> = ZeroExtend(result1, 16);
    R[d]<31:16> = ZeroExtend(result2, 16);
    if sat1 || sat2 then
        PSTATE.Q = '1';
F5.1.290 USAX

Unsigned Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, and writes the results to the destination register. It sets PSTATE.GE according to the results.

**A1**

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
| !=1111 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | Rn | Rd | (1) | (1) | (1) | 0 | 1 | 0 | 1 | Rm |

**A1 variant**

USAX(<c>{<q>} {<Rd>,} <Rn>, <Rm>)

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm);
\]

if \(d == 15 \text{ || } n == 15 \text{ || } m == 15\) then UNPREDICTABLE;

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 14 | 13 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | Rn | 1 | 1 | 1 | 1 | Rd | 0 | 1 | 0 | 0 | Rm |

**T1 variant**

USAX(<c>{<q>} {<Rd>,} <Rn>, <Rm>)

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \ n = \text{UInt}(Rn); \ m = \text{UInt}(Rm);
\]

if \(d == 15 \text{ || } n == 15 \text{ || } m == 15\) then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

<

See Standard assembler syntax fields on page F1-7228.

<

See Standard assembler syntax fields on page F1-7228.

<Rd>

Is the general-purpose destination register, encoded in the "Rd" field.

<Rn>

Is the first general-purpose source register, encoded in the "Rn" field.

<Rm>

Is the second general-purpose source register, encoded in the "Rm" field.
Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();
  sum  = Uint(R[n]<15:0>) + Uint(R[m]<31:16>);
  diff = Uint(R[n]<31:16>) - Uint(R[m]<15:0>);
  R[d]<15:0> = sum<15:0>;
  R[d]<31:16> = diff<15:0>;
  PSTATE.GE<1:0> = if sum  >= 0x10000 then '11' else '00';
  PSTATE.GE<3:2> = if diff >= 0 then '11' else '00';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.291 USUB16

Unsigned Subtract 16 performs two 16-bit unsigned integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

A1

\[
\begin{array}{cccccccccccccccccccccc}
\hline
1=1111 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & \text{cond} & Rn & \text{Rd} & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & Rm
\end{array}
\]

A1 variant

USUB16({<c>}{<q>}{<Rd>,} {<Rn>,} <Rm>)

Decode for this encoding

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

if \(d == 15 || n == 15 || m == 15\) then UNPREDICTABLE;

T1

\[
\begin{array}{cccccccccccccccccccccc}
|15| & 14| & 13| & 12| & 11| & 10| & 9| & 8| & 7| & 6| & 5| & 4| & 3| & 0| & 15| & 14| & 13| & 12| & 11| & 8| & 7| & 6| & 5| & 4| & 3| & 0| \\
\hline
1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & Rn & 1 & 1 & 1 & 1 & Rd & 0 & 1 & 0 & 0 & Rm
\end{array}
\]

T1 variant

USUB16({<c>}{<q>}{<Rd>,} {<Rn>,} <Rm>)

Decode for this encoding

\[
d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

if \(d == 15 || n == 15 || m == 15\) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\) See Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<Rd>\) Is the general-purpose destination register, encoded in the "Rd" field.

\(<Rn>\) Is the first general-purpose source register, encoded in the "Rn" field.

\(<Rm>\) Is the second general-purpose source register, encoded in the "Rm" field.

Operation for all encodings

\[
\text{if } \text{ConditionPassed()} \text{ then}
\]

\[
\text{EncodingSpecificOperations();}
\]

\[
diff1 = \text{UInt}(R[n<15:0>]) - \text{UInt}(R[m<15:0>]);
\]
diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
R[d]<15:0> = diff1<15:0>;
R[d]<31:16> = diff2<15:0>;
PSTATE.GE<1:0> = if diff1 >= 0 then '11' else '00';
PSTATE.GE<3:2> = if diff2 >= 0 then '11' else '00';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.292  USUB8

Unsigned Subtract 8 performs four 8-bit unsigned integer subtractions, and writes the results to the destination register. It sets PSTATE.GE according to the results of the subtractions.

A1

\[
\begin{array}{cccccccccccc}
\end{array}
\]

\begin{array}{cccccccccccc}
| !=1111 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & \text{Rn} & \text{Rd} & (1) & (1) & (1) & 1 & 1 & 1 & 1 & \text{Rm} \\
\end{array}

\text{cond}

A1 variant

USUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

\[
d = \text{UInt}(\text{Rd}); \quad n = \text{UInt}(\text{Rn}); \quad m = \text{UInt}(\text{Rm});
\]

if \( d \neq 15 \) \&\& \( n \neq 15 \) \&\& \( m \neq 15 \) then UNPREDICTABLE;

T1

\[
\begin{array}{cccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 14 & 13 & 12 & 11 & 8 & 7 & 6 & 5 & 4 & 3 & 0 \\
\end{array}
\]

\begin{array}{cccccccccccc}
| 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 | \text{Rn} & 1 & 1 & 1 & \text{Rd} & 0 & 1 & 0 & 0 & \text{Rm} \\
\end{array}

T1 variant

USUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

Decode for this encoding

\[
d = \text{UInt}(\text{Rd}); \quad n = \text{UInt}(\text{Rn}); \quad m = \text{UInt}(\text{Rm});
\]

if \( d \neq 15 \) \&\& \( n \neq 15 \) \&\& \( m \neq 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\) \hspace{1cm} \text{See Standard assembler syntax fields on page F1-7228.}

\(<q>\) \hspace{1cm} \text{See Standard assembler syntax fields on page F1-7228.}

\(<\text{Rd}>\) \hspace{1cm} \text{Is the general-purpose destination register, encoded in the "Rd" field.}

\(<\text{Rn}>\) \hspace{1cm} \text{Is the first general-purpose source register, encoded in the "Rn" field.}

\(<\text{Rm}>\) \hspace{1cm} \text{Is the second general-purpose source register, encoded in the "Rm" field.}

Operation for all encodings

if \text{ConditionPassed}() \text{ then}

\text{EncodingSpecificOperations();}

diff1 = \text{UInt}(R[n]<7:0>) - \text{UInt}(R[m]<7:0>);
diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
R[d]<7:0> = diff1<7:0>;
R[d]<15:8> = diff2<7:0>;
R[d]<23:16> = diff3<7:0>;
R[d]<31:24> = diff4<7:0>;
PSTATE.GE<0> = if diff1 >= 0 then '1' else '0';
PSTATE.GE<1> = if diff2 >= 0 then '1' else '0';
PSTATE.GE<2> = if diff3 >= 0 then '1' else '0';
PSTATE.GE<3> = if diff4 >= 0 then '1' else '0';

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.293   UXTAB

Unsigned Extend and Add Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

**A1**

A1 variant

UXTAB{<c>}{<q>}{<Rd>,} <Rn>, <Rm> {, ROR #<amount>}

*Decode for this encoding*

if Rn == '1111' then SEE "UXTB";

\[ d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{rotation} = \text{UInt}(\text{rotate:'000'}); \]

if d == 15 || m == 15 then UNPREDICTABLE;

**T1**

T1 variant

UXTAB{<c>}{<q>}{<Rd>,} <Rn>, <Rm> {, ROR #<amount>}

*Decode for this encoding*

if Rn == '1111' then SEE "UXTB";

\[ d = \text{UInt}(Rd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \quad \text{rotation} = \text{UInt}(\text{rotate:'000'}); \]

if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**Notes for all encodings**

For more information about the constrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
- `<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:
  (omitted) when rotate = 00
8     when rotate = 01
16    when rotate = 10
24    when rotate = 11

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<7:0>, 32);
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Extend and Add Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

### A1

| 31  | 28 | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 16  | 15  | 12 | 11  | 10  |  9 |  8  |  7 |  6 |  5 |  4 |  3 |  0 |
|-----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| ![1111] |  0 |  1 |  1 |  0 |  1 |  0 |  0 | ![1111] | Rd | ![rotate(0)] | 0 |  1 |  1 | Rm |
| cond |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

#### A1 variant

UXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}

#### Decode for this encoding

if Rn == '1111' then SEE "UXTB16";

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');

if d == 15 || m == 15 then UNPREDICTABLE;

### T1

| 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  0 | 15 | 14 | 13 | 12 | 11 |  8 |  7 |  6 |  5 |  4 |  3 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 1  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

#### T1 variant

UXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}

#### Decode for this encoding

if Rn == '1111' then SEE "UXTB16";

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');

if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

### Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

### Assembler symbols

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
- `<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:
  - (omitted) when rotate = 00
8     when rotate = 01
16    when rotate = 10
24    when rotate = 11

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0> = R[n]<15:0> + ZeroExtend(rotated<7:0>, 16);
    R[d]<31:16> = R[n]<31:16> + ZeroExtend(rotated<23:16>, 16);
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.295   UXTAH

Unsigned Extend and Add Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

A1

```
| 31  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 16  | 15  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| !=1111 | 0  | 1  | 0  | 1  | 1  | !=1111 | Rd  | rotate | 0  | 0  | 1  | 1  | Rm |
```

cond Rn

A1 variant

UXTAH(<c>{<q}> {<Rd>}, <Rn>, <Rm> {, ROR #<amount>})

Decode for this encoding

if Rn == '1111' then SEE "UXTH";

\[
d = \text{UInt}(Rd);  
n = \text{UInt}(Rn);  
m = \text{UInt}(Rm);  
\text{rotation} = \text{UInt}(\text{rotate}:'000');
\]

if d == 15 || m == 15 then UNPREDICTABLE;

T1

```
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 14 | 13 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|----|----|----|----|----|---|---|---|---|---|---|---|----|
| 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | !=1111 | 1  | 1  | 1  | 1  | Rm |
```

Rn

T1 variant

UXTAH(<c>{<q}> {<Rd>}, <Rn>, <Rm> {, ROR #<amount>})

Decode for this encoding

if Rn == '1111' then SEE "UXTH";

\[
d = \text{UInt}(Rd);  
n = \text{UInt}(Rn);  
m = \text{UInt}(Rm);  
\text{rotation} = \text{UInt}(\text{rotate}:'000');
\]

if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<Rn>` Is the first general-purpose source register, encoded in the "Rn" field.
- `<Rm>` Is the second general-purpose source register, encoded in the "Rm" field.
- `<amount>` Is the rotate amount, encoded in the "rotate" field. It can have the following values:

  (omitted) when rotate = 00
8 when rotate = 01
16 when rotate = 10
24 when rotate = 11

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<15:0>, 32);
```

**Operational information**

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.296    UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

A1

\[
\begin{array}{cccccccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & Rd & rotate(0) & (0) & 0 & 1 & 1 & 1 & Rm
\end{array}
\]

cond

**A1 variant**

UXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad \text{rotation} = \text{UInt}(\text{rotate:'000'});
\]

\[
\text{if } d == 15 \text{ || } m == 15 \text{ then UNPREDICTABLE;}
\]

T1

\[
\begin{array}{cccccccccccccccc}
| 15 & 14 & 13 & 12 | 11 & 10 & 9 & 8 & 7 & 6 & 5 & 3 & 2 & 0 |
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & Rm & Rd
\end{array}
\]

**T1 variant**

UXTB{<c>}{<q>} {<Rd>,} <Rm>

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad \text{rotation} = 0;
\]

T2

\[
\begin{array}{cccccccccccccccc}
| 15 & 14 & 13 & 12 | 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 | 15 & 14 & 13 & 12 | 11 & 8 & 7 & 6 & 5 & 4 & 3 & 0 |
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & Rd & 1 & (0) & rotate & Rm
\end{array}
\]

**T2 variant**

UXTB{<c>}.W {<Rd>,} <Rm> // <Rd>, <Rm> can be represented in T1

UXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

**Decode for this encoding**

\[
d = \text{UInt}(Rd); \quad m = \text{UInt}(Rm); \quad \text{rotation} = \text{UInt}(\text{rotate:'000'});
\]

\[
\text{if } d == 15 \text{ || } m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> Is the general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in the "rotate" field. It can have the following values:

- (omitted) when rotate = 00
- 8 when rotate = 01
- 16 when rotate = 10
- 24 when rotate = 11

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<7:0>, 32);

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.297  UXTB16

Unsigned Extend Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

A1

| 31 28 27 26 25 24|23 22 21 20|19 18 17 16|15 12 11 10 9 8 7 6 5 4 |3 0 |
|-----------------|-----------------|-----------------|-----------------|
| !=1111| 0 1 1 0 1 1 0 1 | 1 1 1 1 1 | Rd | rotate[0][0] | 0 1 1 1 | Rm | cond |

A1 variant

UXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

Decode for this encoding

d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;

T1

| 15 14 13 12|11 10 9 8 7 6 5 4 |3 2 1 0 |15 14 13 12|11 8 7 6 5 4 |3 0 |
|-------------|-----------------|-----------------|-----------------|
| 1 1 1 1 1 0 0 0 | 1 1 1 1 1 | 1 1 1 1 | Rd | rotate | 1 0 | Rm |

T1 variant

UXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

Decode for this encoding

d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

< >  See Standard assembler syntax fields on page F1-7228.
<q>  See Standard assembler syntax fields on page F1-7228.
<Rd>  Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>  For encoding A1: is the general-purpose source register, encoded in the "Rm" field.
For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.
<amount>  Is the rotate amount, encoded in the "rotate" field. It can have the following values:
(omitted)  when rotate = 00
8  when rotate = 01
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0> = ZeroExtend(rotated<7:0>, 16);
    R[d]<31:16> = ZeroExtend(rotated<23:16>, 16);

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F5.1.298  UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

A1

    ![1111 0 1 1 0 1 1 1 1 1 1 | Rd  rotate(0)| 0 1 1 1 | Rm]

A1 variant
UXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

Decode for this encoding

\[
d = \text{UInt}(Rd); \ m = \text{UInt}(Rm); \ rotation = \text{UInt}(\text{rotate:'000'});
\]

\[
\text{if } d == 15 || m == 15 \text{ then UNPREDICTABLE;}
\]

T1

    ![15 14 13 12|11 10 9 8 7 6 5 |3 2 0 |]

T1 variant
UXTH{<c>}{<q>} {<Rd>,} <Rm>

Decode for this encoding

\[
d = \text{UInt}(Rd); \ m = \text{UInt}(Rm); \ rotation = 0;
\]

T2

    ![15 14 13 12|11 10 9 8 7 6 5 4 |3 2 1 0 |15 14 13 12|11 8 7 6 5 4 |3 0 |]

T2 variant
UXTH{<c>}.W {<Rd>,} <Rm> // <Rd>, <Rm> can be represented in T1
UXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

Decode for this encoding

\[
d = \text{UInt}(Rd); \ m = \text{UInt}(Rm); \ rotation = \text{UInt}(\text{rotate:'000'});
\]

\[
\text{if } d == 15 || m == 15 \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<\c> See Standard assembler syntax fields on page F1-7228.

<\q> See Standard assembler syntax fields on page F1-7228.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> Is the general-purpose source register, encoded in the "Rm" field.

<\amount> Is the rotate amount, encoded in the "rotate" field. It can have the following values:

- (omitted) when rotate = 00
- 8 when rotate = 01
- 16 when rotate = 10
- 24 when rotate = 11

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<15:0>, 32);

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F5.1.299    WFE

Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a
wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any
PE in the multiprocessor system. For more information, see Wait For Event and Send Event on page G1-8988.

As described in Wait For Event and Send Event on page G1-8988, the execution of a WFE instruction that would
otherwise cause entry to a low-power state can be trapped to a higher Exception level, see:

- Traps to Undefined mode of EL0 execution of WFE and WFI instructions on page G1-9004.
- Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions on page G1-9020.
- Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode on
  page G1-9032.

A1

![Instruction Format](image)

**A1 variant**

WFE{<c>}{<q>}

**Decode for this encoding**

// No additional decoding required

T1

![Instruction Format](image)

**T1 variant**

WFE{<c>}{<q>}

**Decode for this encoding**

// No additional decoding required

T2

![Instruction Format](image)

**T2 variant**

WFE{<c>}.W
Decode for this encoding

// No additional decoding required

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 "Architectural Constraints on UNPREDICTABLE Behaviors."

Assembler symbols

<> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    if IsEventRegisterSet() then
        ClearEventRegister();
    else
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch32.CheckForWxTrap(EL1, WxType_WFE);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch32.CheckForWxTrap(EL2, WxType_WFE);
        if HaveEL(EL3) && PSTATE.M != M32_Monitor then
            // Check for traps described by the Secure Monitor.
            AArch32.CheckForWxTrap(EL3, WxType_WFE);
        integer localtimeout = 1 << 64;  // No local timeout event is generated
        WaitForEvent(localtimeout);
F5.1.300  WFI

Wait For Interrupt is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see Wait For Interrupt on page G1-8991.

As described in Wait For Interrupt on page G1-8991, the execution of a WFI instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level, see:

- Traps to Undefined mode of EL0 execution of WFE and WFI instructions on page G1-9004.
- Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions on page G1-9020.
- Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode on page G1-9032.

A1

\[
\begin{array}{c}
|31|27|26|25|24|23|22|21|20|19|18|17|16|15|14|13|12|11|10|9|8|7|6|5|4|3|2|1|0| \\
\end{array}
\]

cond

WFI[{<c>}]{<q>}

Decode for this encoding

// No additional decoding required

T1

\[
\begin{array}{c}
|15|14|13|12|11|10|9|8|7|6|5|4|3|2|1|0| \\
\end{array}
\]

WFI[{<c>}]{<q>}

Decode for this encoding

// No additional decoding required

T2

\[
\begin{array}{c}
|15|14|13|12|11|10|9|8|7|6|5|4|3|2|1|0| \\
\end{array}
\]

WFI[{<c>}.W
Decode for this encoding

// No additional decoding required

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<> See Standard assembler syntax fields on page F1-7228.

<< See Standard assembler syntax fields on page F1-7228.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    if !InterruptPending() then
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch32.CheckForWfxTrap(EL1, WfxType_WFI);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch32.CheckForWfxTrap(EL2, WfxType_WFI);
        if HaveEL(EL3) && PSTATE.M != M32_Monitor then
            // Check for traps described by the Secure Monitor.
            AArch32.CheckForWfxTrap(EL3, WfxType_WFI);
        integer localtimeout = 1 << 64;  // No local timeout event is generated
        WaitForInterrupt(localtimeout);
F5.1.301 YIELD

YIELD is a hint instruction. Software with a multithreading capability can use a YIELD instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction see The Yield instruction on page F2-7273.

A1

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
    | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | (1) | (1) | (1) | (0) | (0) | (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
    cond

A1 variant

YIELD{<c}{{q}

Decode for this encoding

// No additional decoding required

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
    | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

T1 variant

YIELD{<c}{{q}

Decode for this encoding

// No additional decoding required

T2

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
    | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | (1) | (1) | (1) | (0) | (0) | (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

T2 variant

YIELD{<c}.

Decode for this encoding

// No additional decoding required

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.
Assembler symbols

<\(c\)> See Standard assembler syntax fields on page F1-7228.

<\(q\)> See Standard assembler syntax fields on page F1-7228.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Yield();
F5.2 Encoding and use of banked register transfer instructions

Software executing at EL1 or higher can use the MRS (banked register) and MSR (banked register) instructions to transfer values between the general-purpose registers and Special-purpose registers. One particular use of these instructions is for a hypervisor to save or restore the register values of a Guest OS. The following sections give more information about these instructions:

- Register arguments in the banked register transfer instructions.
- Usage restrictions on the banked register transfer instructions on page F5-8161.
- Encoding the register argument in the banked register transfer instructions on page F5-8162.
- Pseudocode support for the banked register transfer instructions on page F5-8163.

For descriptions of the instructions see MRS (Banked register) on page F5-7736 and MSR (Banked register) on page F5-7740.

F5.2.1 Register arguments in the banked register transfer instructions

Figure F5-1 shows the banked general-purpose registers and Special-purpose registers:

Software using an MRS (banked register) or MSR (banked register) instruction specifies one of these registers using a name shown in Figure F5-1, or an alternative name for SP or LR. These registers can be grouped as follows:

R8-R12 Each of these registers has two banked copies, _usr and _fiq, for example R8_usr and R8_fiq.
SP  There is a banked copy of SP for every mode except System mode. For example, SP_svc is the SP for Supervisor mode.

LR  There is a banked copy of LR for every mode except System mode and Hyp mode. For example, LR_svc is the LR for Supervisor mode.

SPSR  There is a banked copy of SPSR for every mode except System mode and User mode.

ELR_hyp  Except for the operations provided by MRS (banked register) and MSR (banked register), ELR_hyp is accessible only from Hyp mode. It is not banked.

F5.2.2  Usage restrictions on the banked register transfer instructions

MRS (banked register) and MSR (banked register) instructions are CONSTRAINED UNPREDICTABLE if any of the following applies:

- The instruction is executed in User mode.
- The instruction accesses a banked register that is not implemented, or that either:
  - Is not accessible from the current Privilege level and Security state.
  - Can be accessed from the current mode by using a different instruction.

MSR (banked register) and MRS (banked register) on page K1-11582 describes the permitted CONSTRAINED UNPREDICTABLE behavior.

An MRS (banked register) instruction or an MSR (banked register) instruction executed:

- At Non-secure EL1 cannot access any Hyp mode banked registers.
- At Non-secure EL1 or EL2 cannot access any Monitor mode banked registers.
- In a Secure mode other than Monitor mode cannot access any Hyp banked registers.

This means that the banked registers that MRS (banked register) and MSR (banked register) instructions cannot access are:

From Monitor mode
- The current mode registers R8_usr-R12_usr, SP_mon, LR_mon, and SPSR_mon.

From Hyp mode
- The Monitor mode registers SP_mon, LR_mon, and SPSR_mon.
- The current mode registers R8_usr-R12_usr, SP_hyp, LR_usr, and SPSR_hyp.

**Note**

MRS (banked register) and MSR (banked register) instructions can access the current mode register ELR_hyp.

From FIQ mode
- From Non-secure EL1, the Monitor mode registers SP_mon, LR_mon, and SPSR_mon.
- The Hyp mode registers SP_hyp, SPSR_hyp, and ELR_hyp.
- The current mode registers R8_fiq-R12_fiq, SP_fiq, LR_fiq, and SPSR_fiq.

From System mode
- From Non-secure EL1, the Monitor mode registers SP_mon, LR_mon, and SPSR_mon.
- The Hyp mode registers SP_hyp, SPSR_hyp, and ELR_hyp.
- The current mode registers R8_usr-R12_usr, SP_usr, and LR_usr.

From Supervisor mode, Abort mode, Undefined mode, and IRQ mode
- From Non-secure EL1, the Monitor mode registers SP_mon, LR_mon, and SPSR_mon.
- The Hyp mode registers SP_hyp, SPSR_hyp, and ELR_hyp.
- The current mode registers R8_usr-R12_usr, SP_<current_mode>, LR_<current_mode>, and SPSR_<current_mode>.
If EL3 is using AArch64, all MRS (banked register) and MSR (banked register) accesses to the Monitor mode registers from Secure EL1 modes are trapped to EL3.

For more information, see:

- **Encoding the register argument in the banked register transfer instructions**.
- **Pseudocode support for the banked register transfer instructions on page F5-8163**.
- **MRS (Banked register) on page F5-7736**.
- **MSR (Banked register) on page F5-7740**.

--- **Note**

CONSTRAINED UNPREDICTABLE behavior must not give access to registers that are not accessible from the current Privilege level and Security state.

### F5.2.3 Encoding the register argument in the banked register transfer instructions

The MRS (banked register) and MSR (banked register) instructions include a 5-bit field, SYSm, and an R bit, that together encode the register argument for the instruction.

When the R bit is set to 0, the argument is a register other than a banked copy of the SPSR, and Table F5-1 on page F5-8162 shows how the SYSm field defines the required register argument. In this table, CONST.

<table>
<thead>
<tr>
<th>SYSm&lt;2:0&gt;</th>
<th>SYSm&lt;4:3&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>0b01</td>
</tr>
<tr>
<td>0b000</td>
<td>R8_usr</td>
</tr>
<tr>
<td>0b001</td>
<td>R9_usr</td>
</tr>
<tr>
<td>0b010</td>
<td>R10_usr</td>
</tr>
<tr>
<td>0b011</td>
<td>R11_usr</td>
</tr>
<tr>
<td>0b100</td>
<td>R12_usr</td>
</tr>
<tr>
<td>0b101</td>
<td>SP_usr</td>
</tr>
<tr>
<td>0b110</td>
<td>LR_usr</td>
</tr>
<tr>
<td>0b111</td>
<td>CONSTRAINED UNPREDICTABLE</td>
</tr>
</tbody>
</table>
When the R bit is set to 1, the argument is a banked copy of the SPSR, and Table F5-2 on page F5-8163 shows how the SYSm field defines the required register argument. In this table, CONST. UNPREDICTABLE indicates that behavior is CONSTRAINED UNPREDICTABLE.

**Table F5-2 Banked register encodings when R==1**

<table>
<thead>
<tr>
<th>SYSm&lt;2:0&gt;</th>
<th>SYSm&lt;4:3&gt;</th>
<th>0b00</th>
<th>0b01</th>
<th>0b10</th>
<th>0b11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>SPSR_irq</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td></td>
</tr>
<tr>
<td>0b001</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td></td>
</tr>
<tr>
<td>0b010</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>SPSR_svc</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td></td>
</tr>
<tr>
<td>0b011</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td></td>
</tr>
<tr>
<td>0b100</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>SPSR_abt</td>
<td>SPSR_mon</td>
<td></td>
</tr>
<tr>
<td>0b101</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td></td>
</tr>
<tr>
<td>0b110</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>SPSR_fiq</td>
<td>SPSR_irq</td>
<td>SPSR_hyp</td>
<td></td>
</tr>
<tr>
<td>0b111</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td>CONSTRAINED UNPREDICTABLE</td>
<td></td>
</tr>
</tbody>
</table>

**F5.2.4 Pseudocode support for the banked register transfer instructions**

The pseudocode functions `BankedRegisterAccessValid()` and `SPSRAccessValid()` check the validity of MRS (banked register) and MSR (banked register) accesses. That is, they filter the accesses that are CONSTRAINED UNPREDICTABLE either because:

- They attempt to access a register that Usage restrictions on the banked register transfer instructions on page F5-8161 shows is not accessible.
- They use an SYSm<4:0> encoding that Encoding the register argument in the banked register transfer instructions on page F5-8162 shows as CONSTRAINED UNPREDICTABLE.

`BankedRegisterAccessValid()` applies to accesses to the banked general-purpose registers, or to ELR_hyp, and `SPSRAccessValid()` applies to accesses to the SPSRs.
Chapter F6
T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions

This chapter describes each instruction. It contains the following sections:

- Alphabetical list of Advanced SIMD and floating-point instructions on page F6-8166.

Note

Some headings in this chapter use the term floating-point register. This is an abbreviated description, and means a register in the Advanced SIMD and floating-point register file.
F6.1 Alphabetical list of Advanced SIMD and floating-point instructions

This section lists every Advanced SIMD and floating-point instruction in the T32 and A32 instruction sets. For details of the format used see Format of instruction descriptions on page F1-7224.

This section is formatted so that each instruction description starts on a new page.
F6.1.1 AESD

AES single round decryption.

**A1**

(FEAT_AES)

\[
\begin{array}{c|cccccccccccccccc}
\hline
\text{size} & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & D & 1 & 1 & 0 & 0 & Vd & 0 & 0 & 1 & 1 & 0 & 1 & M & 0 & Vm
\end{array}
\]

**A1 variant**

AESD.<dt> <Qd>, <Qm>

*Decode for this encoding*

\[
\text{if !HaveAESExt()} \text{ then UNDEFINED;}
\text{if size !=} '00' \text{ then UNDEFINED;}
\text{if Vd<0> == '1' || Vm<0> == '1' \text{ then UNDEFINED;}}
\text{d = UInt(D:Vd); m = UInt(M:Vm);}
\]

**T1**

(FEAT_AES)

\[
\begin{array}{c|cccccccccccccccc}
| & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 |
\hline
\text{size} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & D & 1 & 1 & 0 & 0 & Vd & 0 & 0 & 1 & 1 & 0 & 1 & M & 0 & Vm
\end{array}
\]

**T1 variant**

AESD.<dt> <Qd>, <Qm>

*Decode for this encoding*

\[
\text{if InITBlock()} \text{ then UNPREDICTABLE;}
\text{if !HaveAESExt()} \text{ then UNDEFINED;}
\text{if size !=} '00' \text{ then UNDEFINED;}
\text{if Vd<0> == '1' || Vm<0> == '1' \text{ then UNDEFINED;}}
\text{d = UInt(D:Vd); m = UInt(M:Vm);}
\]

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

<dt> Is the data type, encoded in the "size" field. It can have the following values:

8 when size = 00

The following encodings are reserved:

- size = 01.
- size = 1x.
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    op1 = Q[d>>1]; op2 = Q[m>>1];
    Q[d>>1] = AESInvSubBytes(AESInvShiftRows(op1 EOR op2));

**Operational information**

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.2 AESE

AES single round encryption.

**A1**

(FEAT_AES)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
</table>
| 1 1 1 1 0 0 1 1 1 | D 1 1 | size 0 0 | Vd 0 0 1 1 0 | M 0 | Vm
```

**A1 variant**

AESE.<dt> <Qd>, <Qm>

*Decode for this encoding*

if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);

**T1**

(FEAT_AES)

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
</table>
| 1 1 1 1 1 1 1 1 | D 1 1 | size 0 0 | Vd 0 0 1 1 0 | M 0 | Vm
```

**T1 variant**

AESE.<dt> <Qd>, <Qm>

*Decode for this encoding*

if InITBlock() then UNPREDICTABLE;
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

<dt> Is the data type, encoded in the "size" field. It can have the following values:

- 8 when size = 00

The following encodings are reserved:

- size = 01.
- size = 1x.
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**Operation for all encodings**

if ConditionPassed() then
   EncodingSpecificOperations(); CheckCryptoEnabled32();
   op1 = Q[d>>1]; op2 = Q[m>>1];
   Q[d>>1] = AESSubBytes(AESShiftRows(op1 EOR op2));

**Operational information**

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.3   AESIMC

AES inverse mix columns.

A1

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 12|11 10 9 8|7 6 5 4|3 0 |
|-----------|-----------|-----------|-----------|--------|-----------|-----------|-----------|-----------|
|           |           |           |           |        |           |           |           |           |
| 1 1 1 1 0 | 0 1 1 1   | D 1 1     | size 0 0  | Vd 0   | 0 1 1 1   | 1 1 M 0   | Vm        |
```

A1 variant

AESIMC.<dt> <Qd>, <Qm>

**Decode for this encoding**

if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);

T1

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 1 1</td>
<td>1 1 1 1</td>
<td>D 1 1</td>
<td>size 0 0</td>
<td>Vd 0</td>
<td>0 1 1 1</td>
<td>1 1 M 0</td>
<td>Vm</td>
</tr>
</tbody>
</table>
```

T1 variant

AESIMC.<dt> <Qd>, <Qm>

**Decode for this encoding**

if InITBlock() then UNPREDICTABLE;
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);

**Assembler symbols**

<dt> Is the data type, encoded in the "size" field. It can have the following values:

- 8 when size = 00

The following encodings are reserved:

- size = 01.
- size = 1x.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q[d>>1] = AESInvMixColumns(Q[m>>1]);

Operational information

If CPSR.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.4 AESMC

AES mix columns.

A1

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & D & 1 & 1 & size & 0 & 0 & Vd & 0 & 0 & 1 & 1 & 1 & 0 & M & 0 & Vm
\end{array}
\]

A1 variant

AESMC.<dt> <Qd>, <Qm>

Decode for this encoding

\[
\text{if } \text{!HaveAESExt() then UNDEFINED;}
\]
\[
\text{if size != '00' then UNDEFINED;}
\]
\[
\text{if Vd<0> == '1' ||Vm<0> == '1' then UNDEFINED;}
\]
\[
d = \text{UInt}(D:Vd); m = \text{UInt}(M:Vm);
\]

T1

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & D & 1 & 1 & size & 0 & 0 & Vd & 0 & 0 & 1 & 1 & 1 & 0 & M & 0 & Vm
\end{array}
\]

T1 variant

AESMC.<dt> <Qd>, <Qm>

Decode for this encoding

\[
\text{if InITBlock() then UNPREDICTABLE;}
\]
\[
\text{if !HaveAESExt() then UNDEFINED;}
\]
\[
\text{if size != '00' then UNDEFINED;}
\]
\[
\text{if Vd<0> == '1' ||Vm<0> == '1' then UNDEFINED;}
\]
\[
d = \text{UInt}(D:Vd); m = \text{UInt}(M:Vm);
\]

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<dt> Is the data type, encoded in the "size" field. It can have the following values:

8 when size = '00'

The following encodings are reserved:

• size = '01'.
• size = '1x'.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    \(Q[d>>1] = AESMixColumns(Q[m>>1])\);

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
### F6.1.5 FLDM\*X (FLDMDBX, FLDMIAX)

FLDMDBX is the Decrement Before variant of this instruction, and FLDMIAX is the Increment After variant. FLDM\*X loads multiple SIMD\&FP registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register.

Arm deprecates use of FLDMDBX and FLDMIAX, except for disassembly purposes, and reassembly of disassembled code.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

#### A1

\[
\begin{array}{c|cccccccc|c|}
\text{cond} & 1 & 1 & 0 & P & U & D & W & 1 & \text{imm8}{<7:1>} & 1 \\
\hline
\text{Decrement Before variant} & \text{Applies when } P == 1 && U == 0 && W == 1. \\
\text{FLDMDBX}{<c>}{<q>} <Rn>!, <dreglist> & \\
\text{Increment After variant} & \text{Applies when } P == 0 && U == 1. \\
\text{FLDMIAX}{<c>}{<q>} <Rn>{!}, <dreglist> & \\
\end{array}
\]

#### Decode for all variants of this encoding

if \( P == '0' && U == '0' \) then SEE "Related encodings";
if \( P == '1' && U == '0' \) then SEE "VLDR";
if \( P == U && U == '1' \) then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE; add = (U == '1'); wback = (W == '1');
d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2; // If UInt(imm8) is odd, see "FLDM\*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;

#### CONSTRAINED UNPREDICTABLE behavior

If \( \text{regs} == 0 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If \( \text{regs} > 16 \) || (d+regs) > 16, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.
### T1

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
<th>15 12 11 10 9 8</th>
<th>7</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 1 0</td>
<td>P</td>
<td>U</td>
<td>D</td>
<td>W</td>
<td>Rn</td>
<td>Vd</td>
</tr>
</tbody>
</table>
```

**Decrement Before variant**

Applies when \( P = 1 \land U = 0 \land W = 1 \).

\[
\text{FLDMDBX\{<c>\}{<q>} <Rn>!, <dreglist>}
\]

**Increment After variant**

Applies when \( P = 0 \land U = 1 \).

\[
\text{FLDMIAX\{<c>\}{<q>} <Rn>{!}, <dreglist>}
\]

**Decode for all variants of this encoding**

- if \( P = '0' \land U = '0' \land W = '0' \) then SEE "Related encodings";
- if \( P = '1' \land U = '0' \land W = '0' \) then SEE "VLDR";
- if \( P = U \land W = '1' \) then UNDEFINED;

// Remaining combinations are \( PW = 010 \) (IA without !), \( 011 \) (IA with !), \( 101 \) (DB with !)

- \( \text{singleRegs} = \text{FALSE}; \text{add} = (U = '1'); \text{wback} = (W = '1'); \)
- \( d = \text{UInt}(0:D); n = \text{UInt}(Rn); \text{imm32} = \text{ZeroExtend}(\text{imm8}'00', 32); \)
- \( \text{regs} = \text{UInt}(\text{imm8}) \text{ DIV 2}; \)

// If UInt(imm8) is odd, see "FLDM*X".

- if \( n = 15 \land (\text{wback} \land \text{CurrentInstrSet()} \neq \text{InstrSet_A32}) \) then UNPREDICTABLE;
- if \( \text{regs} = 0 \lor (\text{regs} > 16 \land (d+\text{regs}) > 32 \) then UNPREDICTABLE;
- if \( \text{imm8<0>} = '1' \land (d+\text{regs}) > 16 \) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \( \text{regs} = 0 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as \text{NOP}.
- The instruction operates as a \text{VLDM} with the same addressing mode but loads no registers.

If \( \text{regs} > 16 \lor (d+\text{regs}) > 16 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as \text{NOP}.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Related encodings: See Advanced SIMD and floating-point 64-bit move on page F3-7324 for the T32 instruction set, or Advanced SIMD and floating-point 64-bit move on page F4-7410 for the A32 instruction set.

**Assembler symbols**

\(<c>\) See Standard assembler syntax fields on page F1-7228.
See *Standard assembler syntax fields* on page F1-7228.

**Rn**

Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.

! Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.

**dreglist**

Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            S[d+r] = MemA[address,4];  address = address+4;
        else
            word1 = MemA[address,4];  word2 = MemA[address+4,4];  address = address+8;
            // Combine the word-aligned words in the correct order for current endianness.
            D[d+r] = if BigEndian(AccType_ATOMIC) then word1:word2 else word2:word1;
        if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.6 FSTMDBX, FSTMIAX

FSTMX stores multiple SIMD&FP registers from the Advanced SIMD and floating-point register file to consecutive locations in using an address from a general-purpose register.

Arm deprecates use of FSTMDBX and FSTMIAX, except for disassembly purposes, and reassembly of disassembled code.

Depending on settings in the CPACR, NSACR, HCPTL, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

Decrement Before variant

Applies when \( P = 1 \) \&\& \( U = 0 \) \&\& \( W = 1 \).

\[
\text{FSTMDBX}\{<c>\}{<q>} <Rn>!, <dreglist>
\]

Increment After variant

Applies when \( P = 0 \) \&\& \( U = 1 \).

\[
\text{FSTMIAX}\{<c>\}{<q>} <Rn>!, <dreglist>
\]

Decode for all variants of this encoding

```c
if P == '0' \&\& U == '0' \&\& W == '0' then SEE "Related encodings";
if P == '1' \&\& W == '0' then SEE "VSTR";
if P == U \&\& W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
singleRegs = FALSE; add = (U == '1'); wback = (W == '1');
d = UInt(D : Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2; // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 \&\& (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' \&\& (d+regs) > 16 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If \( \text{regs} = 0 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If \( \text{regs} > 16 || (\text{d+regs}) > 16 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.
Decrement Before variant

Applies when \( P == 1 \) \&\& \( U == 0 \) \&\& \( W == 1 \).

\[
\text{FSTMDBX}\{<c>}{<q>}\ <Rn>!, \ <dreglist>
\]

Increment After variant

Applies when \( P == 0 \) \&\& \( U == 1 \).

\[
\text{FSTMIAX}\{<c>}{<q>\} \ <Rn>{!}, \ <dreglist>
\]

Decode for all variants of this encoding

- if \( P == '0' \) \&\& \( U == '0' \) \&\& \( W == '0' \) then SEE "Related encodings";
- if \( P == '1' \) \&\& \( W == '0' \) then SEE "VSTR";
- if \( P == U \) \&\& \( W == '1' \) then UNDEFINED;

\[
\begin{array}{cccccccccc}
\text{P} & \text{U} & \text{D} & \text{W} & \text{Rn} & \text{Vd} & \text{0} & \text{1} & \text{1} & \text{0} & \text{1} & \text{imm8<7:1>}
\end{array}
\]

imm8<0>

CONSTRANDED UNPREDICTABLE behavior

If \( \text{regs} == 0 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If \( \text{regs} > 16 \) || \( \text{d+regs} > 16 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Notes for all encodings

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Related encodings: See Advanced SIMD and floating-point 64-bit move on page F3-7324 for the T32 instruction set, or Advanced SIMD and floating-point 64-bit move on page F4-7410 for the A32 instruction set.

Assembler symbols

\(<c>\) See Standard assembler syntax fields on page F1-7228.
See Standard assembler syntax fields on page F1-7228.

<Rn> Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used. However, Arm deprecates use of the PC.

! Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.

<dreglist> Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            MemA[address,4] = S[d+r];  address = address+4;
        else
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4] = if BigEndian(AccType_ATOMIC) then D[d+r]<63:32> else D[d+r]<31:0>;
            MemA[address+4,4] = if BigEndian(AccType_ATOMIC) then D[d+r]<31:0> else D[d+r]<63:32>;
            address = address+8;
        if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.7 SHA1C

SHA1 hash update (choose).

A1

(FEAT_SHA1)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 6 5 4 3 0 | |
|-----------------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 1 1 1 1 0 0 1 0 0 D 0 0 Vn Vd 1 1 0 0 N Q M 0 Vm |

A1 variant

SHA1C.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1

(FEAT_SHA1)

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 15 12 11 10 9 8 7 6 5 4 3 0 | |
|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| 1 1 1 0 1 1 1 1 0 D 0 0 Vn Vd 1 1 0 0 N Q M 0 Vm |

T1 variant

SHA1C.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    x = Q[d>>1];
    y = Q[n>>1]<31:0>; // Note: 32 bits wide
    w = Q[m>>1];
    for e = 0 to 3
        t = SHAchoose(x<63:32>, x<95:64>, x<127:96>);
        y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
        x<63:32> = ROL(x<63:32>, 30);
        <y, x> = ROL(y:x, 32);
    Q[d>>1] = x;

Operational information

If CPSR.DIT is 1:

• The execution time of this instruction is independent of:

  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:

  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.8 SHA1H

SHA1 fixed rotate.

A1

(FEAT_SHA1)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 | 15 | 12 | 11 10 9 8 7 6 5 4 3 | 0 |
|-----------------------------------------------|---|-----------|-----------------|
| 1 1 1 1 0 0 1 1 1 | D | 1 1 | size | 0 | 1 | Vd | 0 | 0 | 1 | 0 | 1 | 1 | M | 0 | Vm |

A1 variant

SHA1H.32 <Qd>, <Qm>

*Decode for this encoding*

if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);

T1

(FEAT_SHA1)

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 |
|------------------------------------------|---|
| 1 1 1 1 1 1 1 1 | D | 1 1 | size | 0 | 1 | Vd | 0 | 0 | 1 | 0 | 1 | 1 | M | 0 | Vm |

T1 variant

SHA1H.32 <Qd>, <Qm>

*Decode for this encoding*

if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q[d>>1] = ZeroExtend(ROL(Q[m>>1]<31:0>, 30), 128);
```

**Operational information**

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.9 SHA1M

SHA1 hash update (majority).

A1

(FEAT_SHA1)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 16 15</th>
<th>12 11 10 9 8 7 6 5 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 0 0 D 1 0 Vn Vd 1 1 0 0 N</td>
<td>Q M 0 Vm</td>
<td></td>
</tr>
</tbody>
</table>

A1 variant

SHA1M.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<> == '1' || Vn<> == '1' || Vm<> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1

(FEAT_SHA1)

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 1 1 0 D 1 0 Vn Vd 1 1 0 0 N</td>
<td>Q M 0 Vm</td>
<td></td>
</tr>
</tbody>
</table>

T1 variant

SHA1M.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<> == '1' || Vn<> == '1' || Vm<> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

Notes for all encodings

For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    x = Q[d>>1];
    y = Q[n>>1]<31:0>; // Note: 32 bits wide
    w = Q[m>>1];
    for e = 0 to 3
        t = SHAmajority(x<63:32>, x<95:64>, x<127:96>);
        y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
        x<63:32> = ROL(x<63:32>, 30);
        <y, x> = ROL(y:x, 32);
    Q[d>>1] = x;

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.10 SHA1P

SHA1 hash update (parity).

A1

(FEAT_SHA1)

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 10 9 8 7 6 5 4 3 0]
[ 0 0 0 0 0 D 0 1 1 1 1 1 0 0 1 Vn Vd 1 1 0 1 N Q M 0 Vm]
```

**A1 variant**

SHA1P.32 <Qd>, <Qn>, <Qm>

**Decode for this encoding**

```
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<> == '1' || Vn<> == '1' || Vm<> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1

(FEAT_SHA1)

```
[15 14 13 12 11 10 9 8 7 6 5 4 3 0 15 12 11 10 9 8 7 6 5 4 3 0]
[ 0 0 0 0 0 D 0 1 1 1 1 0 0 1 Vn Vd 1 1 0 1 N Q M 0 Vm]
```

**T1 variant**

SHA1P.32 <Qd>, <Qn>, <Qm>

**Decode for this encoding**

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<> == '1' || Vn<> == '1' || Vm<> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

**Notes for all encodings**

For more information about the CONSTRANGED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- `<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- `<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
Operation for all encodings

if ConditionPassed() then
   EncodingspecificOperations(); CheckCryptoEnabled32();
   x = Q[d>>1];
   y = Q[n>>1]<31:0>; // Note: 32 bits wide
   w = Q[m>>1];
   for e = 0 to 3
      t = SHAparity(x<63:32>, x<95:64>, x<127:96>);
      y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
      x<63:32> = ROL(x<63:32>, 30);
      <y, x> = ROL(y:x, 32);
   Q[d>>1] = x;

Operational information

If CPSR.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
### F6.1.11 SHA1SU0

SHA1 schedule update 0.

#### A1

(FEAT_SHA1)

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 |7 6 5 4 |3 0 |
| 1 1 1 0 | 0 | 0 | 0 | 0 | D | 1 | 1 | Vn | Vd | 1 1 | 0 | 0 | N | Q | M | 0 | Vm |
```

#### A1 variant

SHA1SU0.32 <Qd>, <Qn>, <Qm>

**Decode for this encoding**

```
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

#### T1

(FEAT_SHA1)

```
| 15 14 13 12|11 10 9 8 |7 6 5 4 |3 0 |15 12|11 10 9 8 |7 6 5 4 |3 0 |
| 1 1 1 | 0 | 1 1 1 0 | 0 | D | 1 | 1 | Vn | Vd | 1 1 | 0 | 0 | N | Q | M | 0 | Vm |
```

#### T1 variant

SHA1SU0.32 <Qd>, <Qn>, <Qm>

**Decode for this encoding**

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

#### Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

#### Assembler symbols

- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.
- `<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as `<Qn>*2.
- `<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    op1 = Q[d>>1]; op2 = Q[n>>1]; op3 = Q[m>>1];
    op2 = op2<63:0> : op1<127:64>;
    Q[d>>1] = op1 EOR op2 EOR op3;

Operational information

If CPSR.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.12 SHA1SU1

SHA1 schedule update 1.

A1

(FEAT_SHA1)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 10 9 8 7 6 5 4 3 0 |
|------------------|------------------|
| 1 1 1 1 0 0 1 1 1 D 1 1 | size 1 0 | Vd 0 0 1 1 1 0 M 0 | Vm |

A1 variant

SHA1SU1.32 <Qd>, <Qm>

Decode for this encoding

if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);

T1

(FEAT_SHA1)

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 D 1 1</td>
</tr>
</tbody>
</table>

T1 variant

SHA1SU1.32 <Qd>, <Qm>

Decode for this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[m>>1];
    T = X EOR LSR(Y, 32);
    W0 = ROL(T<31:0>, 1);
    W1 = ROL(T<63:32>, 1);
    W2 = ROL(T<95:64>, 1);
    W3 = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);

Operational information

If CPSR.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.13 SHA256H

SHA256 hash update part 1.

A1

(FEAT_SHA256)

```
| 0 1 6 1 1 1 0 0 1 1 0 | D | 0 0 | Vn | Vd | 1 1 0 0 | N | Q | M | 0 | Vm |
```

A1 variant

SHA256H.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1

(FEAT_SHA256)

```
| 0 1 5 1 4 1 3 1 2 1 1 1 0 | 9 8 7 6 5 4 | 3 | 0 1 | 5 1 1 1 1 1 0 | D | 0 0 | Vn | Vd | 1 1 0 0 | N | Q | M | 0 | Vm |
```

T1 variant

SHA256H.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[n>>1]; W = Q[m>>1]; part1 = TRUE;
    Q[d>>1] = SHA256hash(X, Y, W, part1);

Operational information

If CPSR.DIT is 1:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.14 SHA256H2

SHA256 hash update part 2.

A1

(FEAT_SHA256)

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 16</th>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1</td>
<td>0 D 0 1 Vn</td>
<td>Vd</td>
<td>1 1 0 0 N Q M 0</td>
<td>Vm</td>
</tr>
</tbody>
</table>
```

**A1 variant**

SHA256H2.32 <Qd>, <Qn>, <Qm>

**Decode for this encoding**

- if !HaveSHA256Ext() then UNDEFINED;
- if Q != '1' then UNDEFINED;
- if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
- d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1

(FEAT_SHA256)

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4</th>
<th>3 0</th>
<th>15 12</th>
<th>11 10 9 8 7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
<td>1 1 1 1 0 D 0 1</td>
<td>Vn</td>
<td>Vd</td>
<td>1 1 0 0 N Q M 0</td>
<td>Vm</td>
</tr>
</tbody>
</table>
```

**T1 variant**

SHA256H2.32 <Qd>, <Qn>, <Qm>

**Decode for this encoding**

- if InITBlock() then UNPREDICTABLE;
- if !HaveSHA256Ext() then UNDEFINED;
- if Q != '1' then UNDEFINED;
- if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
- d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

**Notes for all encodings**

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see *Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.
- `<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as `<Qn>*2.
- `<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.
Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations(); CheckCryptoEnabled32();
  X = Q[m>>1]; Y = Q[d>>1]; W = Q[m>>1]; part1 = FALSE;
  Q[d>>1] = SHA256hash(X, Y, W, part1);

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.15 SHA256SU0

SHA256 schedule update 0.

**A1**

**(FEAT_SHA256)**

```
|31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 10 9 8 7 6 5 4 3 0 |
|1 |1 |1 |1 |0 |0 |1 |1 |1 |D |1 |1 |size |1 |0 |Vd |0 |0 |1 |1 |1 |M |0 |Vm |
```

**A1 variant**

SHA256SU0.32 <Qd>, <Qm>

**Decode for this encoding**

if !HaveSHA256Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<> == '1' || Vm<> == '1' then UNDEFINED;
d = UInt(D\:Vd); m = UInt(M\:Vm);

**T1**

**(FEAT_SHA256)**

```
|15 14 13 12|11 10 9 8 7 6 5 4 3 2 1 0|15 12|11 10 9 8 7 6 5 4 3 0 |
|1 |1 |1 |1 |1 |1 |1 |1 |D |1 |1 |size |1 |0 |Vd |0 |0 |1 |1 |1 |M |0 |Vm |
```

**T1 variant**

SHA256SU0.32 <Qd>, <Qm>

**Decode for this encoding**

if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<> == '1' || Vm<> == '1' then UNDEFINED;
d = UInt(D\:Vd); m = UInt(M\:Vm);

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see **Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors**.

**Assembler symbols**

- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D\:Vd" field as <Qd>*2.
- <Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M\:Vm" field as <Qm>*2.
### Operation for all encodings

```c
if ConditionPassed() then
    bits(128) result;
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    x = Q[d>>1]; y = Q[m>>1];
    t = y<31:0> : x<127:32>;
    for e = 0 to 3
        elt = Elem[t, e, 32];
        elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
        Elem[result, e, 32] = elt + Elem[x, e, 32];
    Q[d>>1] = result;
```

### Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.16   SHA256SU1

SHA256 schedule update 1.

A1

(FEAT_SHA256)

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & D & 1 & 0 & Vn & Vd & 1 & 1 & 0 & N & Q & M & 0 & Vm
\end{array}
\]

A1 variant

SHA256SU1.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1

(FEAT_SHA256)

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & D & 1 & 0 & Vn & Vd & 1 & 1 & 0 & N & Q & M & 0 & Vm
\end{array}
\]

T1 variant

SHA256SU1.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    bits(32) elt;
    bits(128) result;
    x = Q[d>>1]; y = Q[n>>1]; z = Q[m>>1];
    T0 = z<31:0> : y<127:32>;

    T1 = z<127:64>;
    for e = 0 to 1
        elt = Elem[T1, e, 32];
        elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
        elt = elt + Elem[x, e, 32] + Elem[T0, e, 32];
        Elem[result, e, 32] = elt;

    T1 = result<63:0>;
    for e = 2 to 3
        elt = Elem[T1, e - 2, 32];
        elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
        elt = elt + Elem[x, e, 32] + Elem[T0, e, 32];
        Elem[result, e, 32] = elt;

    Q[d>>1] = result;

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.17  VABA

Vector Absolute Difference and Accumulate subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

Operand and result elements are all integers of the same length.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VABA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VABA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when Q == 0.

VABA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VABA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**Assembler symbols**

<\> For encoding A1: see *Standard assembler syntax fields* on page F1-7228. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields* on page F1-7228.

<\> See *Standard assembler syntax fields* on page F1-7228.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:
- S8 when U = 0, size = 00
- S16 when U = 0, size = 01
- S32 when U = 0, size = 10
- U8 when U = 1, size = 00
- U16 when U = 1, size = 01
- U32 when U = 1, size = 10

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[Din[n+r],e,esize];
        op2 = Elem[Din[m+r],e,esize];
        absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
        if long_destination then
            Elem[D[Qd>>1],e,2*esize] = Elem[Din[Qd>>1],e,2*esize] + absdiff;
        else
            Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + absdiff;

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.
F6.1.18 VABAL

Vector Absolute Difference and Accumulate Long subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

Operand elements are all integers of the same length, and the result elements are double the length of the operands. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

```
[31 30 29 28][27 26 25 24][23 22 21 20]19 16 15 12 11 10 9 8 7 6 5 4 3 0
1 1 1 0 1 | U | 1 | D | != 11 | Vn | Vd | 0 1 0 1 | N | 0 | M | 0 | Vm
```

A1 variant

VABAL{<c>{<q>},{dt}<Qd>, <Dn>, <Dm>}

Decode for this encoding

- if size == '11' then SEE "Related encodings";
- if Vd<0> == '1' then UNDEFINED;
- unsigned = (U == '1'); long_destination = TRUE;
- esize = 8 << UInt(size); elements = 64 DIV esize;
- d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;

T1

```
[15 14 13 12][11 10 9 8 7 6 5 4 3 0][15 12][11 10 9 8 7 6 5 4 3 0]
1 1 1 U 1 1 1 1 | D | != 11 | Vn | Vd | 0 1 0 1 | N | 0 | M | 0 | Vm
```

T1 variant

VABAL{<c>{<q>},{dt}<Qd>, <Dn>, <Dm>}

Decode for this encoding

- if size == '11' then SEE "Related encodings";
- if Vd<0> == '1' then UNDEFINED;
- unsigned = (U == '1'); long_destination = TRUE;
- esize = 8 << UInt(size); elements = 64 DIV esize;
- d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;

Notes for all encodings

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

Assembler symbols

<

For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions
F6.1 Alphabetical list of Advanced SIMD and floating-point instructions

For encoding T1: see *Standard assembler syntax fields* on page F1-7228.

See *Standard assembler syntax fields* on page F1-7228.

Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

- **S8** when \( U = 0 \), \( size = 00 \)
- **S16** when \( U = 0 \), \( size = 01 \)
- **S32** when \( U = 0 \), \( size = 10 \)
- **U8** when \( U = 1 \), \( size = 00 \)
- **U16** when \( U = 1 \), \( size = 01 \)
- **U32** when \( U = 1 \), \( size = 10 \)

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<Qd>*2\).

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```c
if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = Elem[Din[n+r],e,esize];
      op2 = Elem[Din[m+r],e,esize];
      absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
      if long_destination then
        Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + absdiff;
      else
        Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + absdiff;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.19  VABD (floating-point)

Vector Absolute Difference (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are floating-point numbers of the same size.

Depending on settings in the CPACR, NSACR, and HCPR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 6 5 4 3 0 |
|-----------------|----------------|-----------------|-----------------|----------------|
| 1 1 1 0 1 1 0 D 1 sz | Vn | Vd | 1 1 0 1 N Q M 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 |
|----------------|-----------------|----------------|----------------|
| 1 1 1 1 1 1 1 0 D 1 sz | Vn | Vd | 1 1 0 1 N Q M 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRANDED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<o> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
    For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<d> Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:
    F32 when sz = 0
    F16 when sz = 1

<qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <qd>*2.

<q> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <q>*2.

<qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <qm>*2.

<d> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<n> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<m> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize]; op2 = Elem[D[m+r],e,esize];
            Elem[D[d+r],e,esize] = FPAbs(FPSub(op1,op2,StandardFPSCRValue()));
F6.1.20  VABD (integer)

Vector Absolute Difference (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are all integers of the same length.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 |7 6 5 4 |3 0 |
|-----------|-----------|-----------|---|---|-------|---|---|---|
| 1 1 1 0 0 1 |U|0|D|size|Vn|Vd|0 1 1 1 |N|Q|M|0|Vm|
```

**64-bit SIMD vector variant**

Applies when Q == 0.

VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

**128-bit SIMD vector variant**

Applies when Q == 1.

VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

**Decode for all variants of this encoding**

```c
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

```
| 15 14 13 12|11 10 9 8 |7 6 5 4 |3 0 |
|-----------|-------|---|---|---|
| 1 1 1 1 |U|1|1|1|0|D|size|Vn|Vd|0 1 1 1 |N|Q|M|0|Vm|
```

**64-bit SIMD vector variant**

Applies when Q == 0.

VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

**128-bit SIMD vector variant**

Applies when Q == 1.

VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

**Decode for all variants of this encoding**

```c
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
```
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Assembler symbols

<>  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
   For encoding T1: see Standard assembler syntax fields on page F1-7228.

<>  See Standard assembler syntax fields on page F1-7228.

<dt>  Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

   S8  when U = 0, size = 00
   S16 when U = 0, size = 01
   S32 when U = 0, size = 10
   U8  when U = 1, size = 00
   U16 when U = 1, size = 01
   U32 when U = 1, size = 10

<qd>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<qn>  Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<qm>  Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>  Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>  Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations();  CheckAdvSIMDEnabled();
   for r = 0 to regs-1
      for e = 0 to elements-1
         op1 = Elem[Din[n+r],e,esize];
         op2 = Elem[Din[m+r],e,esize];
         absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
         if long_destination then
            Elem[Q[d>>1],e,2*esize] = absdiff<2*esize-1:0>;
         else
            Elem[D[d+r],e,esize] = absdiff<esize-1:0>;
   Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.
F6.1.21 VABDL (integer)

Vector Absolute Difference Long (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand elements are all integers of the same length, and the result elements are double the length of the operands. Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 6 5 4 3 0 |
|-----------------------------|-----------------------------|-----------------------------|-----------------------------|-----------------------------|
| 1 1 1 1 0 0 1 1 1 D l=11 Vn | Vd 0 1 1 1 N 0 M 0 Vm       |

A1 variant

VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if Vd<> == '1' then UNDEFINED;
unsigned = (U == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 15 12 11 10 9 8 7 6 5 4 3 0 |
|-----------------------------|-----------------------------|-----------------------------|-----------------------------|-----------------------------|
| 1 1 1 1 1 1 1 1 1 D l=11 Vn | Vd 0 1 1 1 N 0 M 0 Vm       |

T1 variant

VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if Vd<> == '1' then UNDEFINED;
unsigned = (U == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;

Notes for all encodings

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

Assembler symbols

<c>
For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.
See Standard assembler syntax fields on page F1-7228.

Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

- **S8** when $U = 0$, size = 00
- **S16** when $U = 0$, size = 01
- **S32** when $U = 0$, size = 10
- **U8** when $U = 1$, size = 00
- **U16** when $U = 1$, size = 01
- **U32** when $U = 1$, size = 10

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as $<Qd>*2$.

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for $r = 0$ to regs-1
        for $e = 0$ to elements-1
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = absdiff<2*esize-1:0>;
            else
                Elem[D[d+r],e,esize] = absdiff<esize-1:0>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.22   VABS

Vector Absolute takes the absolute value of each element in a vector, and places the results in a second vector. The floating-point version only clears the sign bit.

Depending on settings in the CPACR, NSACR, HCPtr, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 12|11 10 9 8|7 6 5 4|3 0 |
|---|---|---|---|---|---|---|---|---|
| 1 1 1 1 0 0 1 1 | D | 1 1 | size | 0 | 1 | Vd | 0 | F | 1 1 0 | Q | M | 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VABS{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VABS{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

A2

| 31 28|27 26 25 24|23 22 21 20|19 18 17 16|15 12|11 10 9 8|7 6 5 4|3 0 |
|---|---|---|---|---|---|---|---|---|
| 1=1111 1 1 1 0 1 | D | 1 1 | 0 0 0 | Vd | 1 0 | size | 1 1 | M | 0 | Vm |

Half-precision scalar variant

Applies when size == 01.

VABS{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar variant

Applies when size == 10.

VABS{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar variant

Applies when size == 11.

VABS{<c>}{<q>}.F64 <Dd>, <Dm>
Decode for all variants of this encoding

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' then UNDEFINED;
if size == '01' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15</th>
<th>12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1</td>
<td>1 1</td>
<td>size</td>
<td>0 1</td>
<td>Vd</td>
<td>0</td>
<td>F</td>
<td>1 1 0</td>
<td>Q</td>
<td>M</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VABS{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VABS{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE; floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 1;

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
T2

| 15 14 13 12| 11 10 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 12| 11 10 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | D | 1 | 1 | 0 | 0 | 0 | Vd | 1 | 0 | size | 1 | 1 | M | 0 | Vm |

**Half-precision scalar variant**
Applies when size == 01.

\[
VABS\{<c>\}{<q>}.F16 <Sd>, <Sm>
\]

**Single-precision scalar variant**
Applies when size == 10.

\[
VABS\{<c>\}{<q>}.F32 <Sd>, <Sm>
\]

**Double-precision scalar variant**
Applies when size == 11.

\[
VABS\{<c>\}{<q>}.F64 <Dd>, <Dm>
\]

**Decode for all variants of this encoding**

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
advsimd = FALSE;
integer esize;
d = UInt(Vd:D);
m = UInt(Vm:M);
case size of
  when '01' esize = 16;
  when '10' esize = 32;
  when '11' esize = 64;
  case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**
If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

\(<c>\) For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding A2, T1 and T2: see Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<dt>\) Is the data type for the elements of the vectors, encoded in the "F:size" field. It can have the following values:

- S8 when F = 0, size = 00
- S16 when F = 0, size = 01
- S32 when F = 0, size = 10
F16 when \( F = 1 \), size = 01
F32 when \( F = 1 \), size = 10

\(<Q_d>\) is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<Q_d>*2.\)

\(<Q_m>\) is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as \(<Q_m>*2.\)

\(<D_d>\) is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<D_m>\) is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

\(<S_d>\) is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

\(<S_m>\) is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

**Operation for all encodings**

if ConditionPassed() then
   EncodingSpecificOperations(); CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
   if advsimd then // Advanced SIMD instruction
      for \( r = 0 \) to \( \text{regs-1} \)
         for \( e = 0 \) to \( \text{elements-1} \)
            if floating_point then
               Elem[D[d+r],e,esize] = FPAbs(Elem[D[m+r],e,esize]);
            else
               result = Abs(SInt(Elem[D[m+r],e,esize]));
               Elem[D[d+r],e,esize] = result<esize-1:0>;
         else // VFP instruction
            case esize of
               when 16 \( S[d] = \text{Zeros}(16) \) : FPAbs(S[m]<15:0>);
               when 32 \( S[d] = \text{FPAbs}(S[m]) \);
               when 64 \( D[d] = \text{FPAbs}(D[m]) \);

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

* The execution time of this instruction is independent of:
  * The values of the data supplied in any of its registers.
  * The values of the NZCV flags.

* The response of this instruction to asynchronous exceptions does not vary based on:
  * The values of the data supplied in any of its registers.
  * The values of the NZCV flags.
F6.1.23   VACGE

Vector Absolute Compare Greater Than or Equal takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operands and result can be quadword or doubleword vectors. They must all be the same size.

The operand vector elements are floating-point numbers. The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

This instruction is used by the pseudo-instruction VACLE. The pseudo-instruction is never the preferred disassembly.

A1

64-bit SIMD vector variant

 Applies when \( Q = 0 \).

\[
\text{VACGE}\{<c>\}{<q>}.<dt> \{<Dd>, }<Dn>, <Dm>
\]

128-bit SIMD vector variant

 Applies when \( Q = 1 \).

\[
\text{VACGE}\{<c>\}{<q>}.<dt> \{<Qd>, }<Qn>, <Qm>
\]

Decode for all variants of this encoding

if \( Q = '1' \) && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
or_equal = (op == '0');
integer esize, integer elements;
\begin{align*}
\text{case sz of} \\
\text{when '0' esize = 32; elements = 2;}
\text{when '1' esize = 16; elements = 4;}
\end{align*}

\( d = \text{UInt}(D:Vd); \ n = \text{UInt}(N:Vn); \ m = \text{UInt}(M:Vm); \) regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

 Applies when \( Q = 0 \).

\[
\text{VACGE}\{<c>\}{<q>}.<dt> \{<Dd>, }<Dn>, <Dm>
\]
128-bit SIMD vector variant

Applies when Q == 1.

VACGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' & !HaveFP16Ext() then UNDEFINED;
if sz == '1' & InITBlock() then UNPREDICTABLE;

or_equal = (op == '0');
integer esize;
integer elements;

case sz of
     when '0' esize = 32; elements = 2;
     when '1' esize = 16; elements = 4;
   d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' & InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:
    F32 when sz = 0
    F16 when sz = 1

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = FPAbs(Elem[D][n+r],e,esize));  op2 = FPAbs(Elem[D][m+r],e,esize));
            boolean test_passed;
if or_equal then
test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());
else
test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());
Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
F6.1.24  VACLE

Vector Absolute Compare Less Than or Equal takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is less than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This instruction is a pseudo-instruction of the VACGE instruction. This means that:

- The encodings in this description are named to match the encodings of VACGE.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VACGE gives the operational pseudocode for this instruction.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VACLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VACGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when Q == 1.

VACLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VACGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

and is never the preferred disassembly.

T1

64-bit SIMD vector variant

Applies when Q == 0.

VACLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VACGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

and is never the preferred disassembly.
128-bit SIMD vector variant

Applies when Q == 1.

VACLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VACGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

and is never the preferred disassembly.

Assembler symbols

<dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<c> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

F32 when sz = 0
F16 when sz = 1

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Operation for all encodings

The description of VACGE gives the operational pseudocode for this instruction.

Operational information

The description of VACGE gives the operational pseudocode for this instruction.
F6.1.25  VACGT

Vector Absolute Compare Greater Than takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operands and result can be quadword or doubleword vectors. They must all be the same size.

The operand vector elements are floating-point numbers. The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

This instruction is used by the pseudo-instruction VACLT. The pseudo-instruction is never the preferred disassembly.

64-bit SIMD vector variant

Applies when Q == 0.

VACGT{<c>}{<q>}{<op>}{<dt>}{<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VACGT{<c>}{<q>}{<op>}{<dt>}{<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
or_equal = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

64-bit SIMD vector variant

Applies when Q == 0.

VACGT{<c>}{<op>}{<dt>}{<Dd>, }<Dn>, <Dm>
128-bit SIMD vector variant

Applies when Q == 1.

\[ VACGT\{<c>\}{<q>}{<dt> \{<Qd>, }<Qn>, <Qm> \]  

Decode for all variants of this encoding

\[
\begin{align*}
\text{if } Q & = '1' \text{ } \&\& \text{ } (Vd<0> == '1' \text{ } \mid \text{ } Vn<0> == '1' \text{ } \mid \text{ } Vm<0> == '1') \text{ then UNDEFINED;} \\
\text{if } sz & = '1' \text{ } \&\& \text{ } !\text{HaveFP16Ext()} \text{ then UNDEFINED;} \\
\text{if } sz & = '1' \text{ } \&\& \text{ } \text{InITBlock()} \text{ then UNPREDICTABLE;} \\
or\text{_equal} & = (op == '0'); \\
\end{align*}
\]

integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
\]

d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRANDED UNPREDICTABLE behavior

If sz == '1' \&\& InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

\(<c>\) For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<dt>\) Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

- F32 when sz = 0
- F16 when sz = 1

\(<Qd>\) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

\(<Qn>\) Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

\(<Qm>\) Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

\(<Dd>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<Dn>\) Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

\(<Dm>\) Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

\[
\begin{align*}
\text{if } ConditionPassed() \text{ then } \\
\text{EncodingSpecificOperations(); CheckAdvSIMDEnabled();} \\
\text{for } r = 0 \text{ to regs-1} \\
\text{for } e = 0 \text{ to elements-1} \\
\text{op1} = \text{FPAbs}(\text{Elem}[D[n+r],e,esize]); \text{ op2} = \text{FPAbs}(\text{Elem}[D[m+r],e,esize]); \\
\text{boolean test\_passed;}
\end{align*}
\]
if or_equal then
    test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());
else
    test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());
Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
F6.1.26  VACLT

Vector Absolute Compare Less Than takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is less than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This instruction is a pseudo-instruction of the VACGT instruction. This means that:

- The encodings in this description are named to match the encodings of VACGT.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VACGT gives the operational pseudocode for this instruction.

**A1**

$$\begin{array}{cccccccccc}
|31|30|29|28|27|26|25|24|23|22|21|20|19|16|15|12|11|10| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 11110011 |
\end{array}$$

**64-bit SIMD vector variant**

Applies when $Q == 0$.

VACLT{$<c>$}{$<q>$}{$<dt>$} {$<Dd>$}, {$<Dm>$}, {$<Dn>$}

is equivalent to

VACGT{$<c>$}{$<q>$}{$<dt>$} {$<Dd>$}, {$<Dm>$}, {$<Dn>$}

and is never the preferred disassembly.

**128-bit SIMD vector variant**

Applies when $Q == 1$.

VACLT{$<c>$}{$<q>$}{$<dt>$} {$<Qd>$}, {$<Qm>$}, {$<Qn>$}

is equivalent to

VACGT{$<c>$}{$<q>$}{$<dt>$} {$<Qd>$}, {$<Qm>$}, {$<Qn>$}

and is never the preferred disassembly.

**T1**

$$\begin{array}{cccccccccc}
|15|14|13|12|11|10| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15|14|13|12|11|10| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 11111111 |
\end{array}$$

**64-bit SIMD vector variant**

Applies when $Q == 0$.

VACLT{$<c>$}{$<q>$}{$<dt>$} {$<Dd>$}, {$<Dm>$}, {$<Dn>$}

is equivalent to

VACGT{$<c>$}{$<q>$}{$<dt>$} {$<Dd>$}, {$<Dm>$}, {$<Dn>$}

and is never the preferred disassembly.
128-bit SIMD vector variant

Applies when \( Q == 1 \).

\[
\text{VACLT}\{<c>\}{<q>}.\{\text{dt}\} \{<Qd>\}, \{<Qn>\}, \{<Qm>\}
\]
is equivalent to

\[
\text{VACGT}\{<c>\}{<q>}.\text{dt} \{<Qd>\}, \{<Qm>\}, \{<Qn>\}
\]
and is never the preferred disassembly.

Assembler symbols

\(<Dm>\)
Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

\(<Dn>\)
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

\(<Qm>\)
Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as \(<Qm>\times 2\).

\(<Qn>\)
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as \(<Qn>\times 2\).

\(<c>\)
For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

\(<q>\)
See Standard assembler syntax fields on page F1-7228.

\(<\text{dt}>\)
Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

\[\begin{array}{ll}
\text{F32} & \text{when } sz = 0 \\
\text{F16} & \text{when } sz = 1
\end{array}\]

\(<Qd>\)
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<Qd>\times 2\).

\(<Dd>\)
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Operation for all encodings

The description of VACGE gives the operational pseudocode for this instruction.

Operational information

The description of VACGE gives the operational pseudocode for this instruction.
F6.1.27 VADD (floating-point)

Vector Add (floating-point) adds corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, HCPT, and FPX registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>31 28 27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16 15</th>
<th>12 11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 0</td>
<td>D 0 sz</td>
<td>Vn</td>
<td>Vd</td>
<td>1 1 0 1</td>
<td>N Q M 0</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
integer esize;
integer elements;
case sz of
when '0' esize = 32; elements = 2;
when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

A2

<table>
<thead>
<tr>
<th>31 28 27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16 15</th>
<th>12 11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=1111</td>
<td>1 1 1 0 0</td>
<td>D 1 1</td>
<td>Vn</td>
<td>Vd</td>
<td>1 0</td>
</tr>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Half-precision scalar variant

Applies when size == 01.

VADD{<c>}{<q>}.F16 {<Sd>, }<Sn>, <Sm>

Single-precision scalar variant

Applies when size == 10.

VADD{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>

Double-precision scalar variant

Applies when size == 11.
VADD{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>

**Decode for all variants of this encoding**

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
integer esize;
integer d;
integer n;
integer m;
case size of
when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

<table>
<thead>
<tr>
<th>[15 14 13 12]</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
<th>[15 14 13 12]</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0</td>
<td>1 1 1 1 0</td>
<td>D 0</td>
<td>sz</td>
<td>Vn</td>
<td>Vd</td>
<td>1 1 0 1</td>
<td>N Q M 0</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector variant**

Applies when Q == 0.
VADD{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>

**128-bit SIMD vector variant**

Applies when Q == 1.
VADD{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>

**Decode for all variants of this encoding**

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE;
integer esize;
integer elements;
case sz of
when '0' esize = 32; elements = 2;
when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**CONSTRAINED UNPREDICTABLE behavior**

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

\[
\begin{array}{cccccccccccccccc}
| 15 & 14 & 13 & 12 | 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 | \\
\hline
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & D & 1 & 1 & Vn & Vd & 1 & 0 & s & i & z & e & N & 0 & M & 0 & Vm \\
\end{array}
\]

**Half-precision scalar variant**

Applies when size == 01.

\[
\text{VADD}\{<c>\}{<q>}.F16 \{<5d>,\} <Sn>, <Sm>
\]

**Single-precision scalar variant**

Applies when size == 10.

\[
\text{VADD}\{<c>\}{<q>}.F32 \{<5d>,\} <Sn>, <Sm>
\]

**Double-precision scalar variant**

Applies when size == 11.

\[
\text{VADD}\{<c>\}{<q>}.F64 \{<5d>,\} <Dn>, <Dm>
\]

**Decode for all variants of this encoding**

\[
\begin{align*}
\text{if FPSCR.Len} \neq '000' || \text{FPSCR.Stride} \neq '00' \text{ then UNDEFINED;}
\text{if size} \neq '00' || (\text{size} = '01' \&\& \text{!HaveFP16Ext()}) \text{ then UNDEFINED;}
\text{if size} = '01' \&\& \text{InITBlock()} \text{ then UNPREDICTABLE;}
\text{advsimd} = \text{FALSE;}
\text{integer esize;}
\text{integer d;}
\text{integer n;}
\text{integer m;}
\text{case size of}
\begin{align*}
\text{when '01' esize} = 16; d = \text{UInt(Vd:D); n} = \text{UInt(Vn:N); m} = \text{UInt(Vm:M);} \\
\text{when '10' esize} = 32; d = \text{UInt(Vd:D); n} = \text{UInt(Vn:N); m} = \text{UInt(Vm:M);} \\
\text{when '11' esize} = 64; d = \text{UInt(D:Vd); n} = \text{UInt(N:Vn); m} = \text{UInt(M:Vm);} \\
\end{align*}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' \&\& InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

\(<c>\) For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding A2, T1 and T2: see Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:
F32 when sz = 0
F16 when sz = 1

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

**Operation for all encodings**

if ConditionPassed() then
   EncodingSpecificOperations(); CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
   if advsimd then // Advanced SIMD instruction
      for r = 0 to regs-1
         for e = 0 to elements-1
            Elem[D[d+r],e,esize] = FPAdd(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRValue());
   else // VFP instruction
      case esize of
         when 16
            S[d] = Zeros(16) : FPAdd(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
         when 32
            S[d] = FPAdd(S[n], S[m], FPSCR[]);
         when 64
            D[d] = FPAdd(D[n], D[m], FPSCR[]);
F6.1.28  VADD (integer)

Vector Add (integer) adds corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size); elements = 64 DIV esize;

d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when Q == 0.

VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size); elements = 64 DIV esize;

d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
Assembler symbols

\(<c>\) For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<dt>\) Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

- I8 when size = 00
- I16 when size = 01
- I32 when size = 10
- I64 when size = 11

\(<qd>\) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

\(<qn>\) Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

\(<qm>\) Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

\(<Dd>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<Dn>\) Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

\(<Dm>\) Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      Elem[D[d+\(r\)],e,esize] = Elem[D[n+r],e,esize] + Elem[D[m+r],e,esize];

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.29 VADDHN

Vector Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are truncated. For rounded results, see VRADDHN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 | 12 | 11 10 9 8 7 6 5 4 3 | 0 |
|-------------|-------------|-------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | D | !=11 | Vn | Vd | 0 | 1 | 0 | 0 | N | 0 | M | 0 | Vm |

size

A1 variant

VADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D;Vd); n = UInt(N;Vn); m = UInt(M;Vm);

T1

| 15 14 13 | 12 | 11 10 9 8 7 6 5 4 3 | 0 | 15 | 12 | 11 10 9 8 7 6 5 4 3 | 0 |
|-----------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | D | !=11 | Vn | Vd | 0 | 1 | 0 | 0 | N | 0 | M | 0 | Vm |

size

T1 variant

VADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D;Vd); n = UInt(N;Vn); m = UInt(M;Vm);

Notes for all encodings

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

Assembler symbols

<c>

For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.
See Standard assembler syntax fields on page F1-7228.

Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

- I16  when size = 00
- I32  when size = 01
- I64  when size = 10

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] + Elem[Qin[m>>1],e,2*esize];
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
### F6.1.30 VADDL

Vector Add Long adds corresponding elements in two doubleword vectors, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of both operands.

Depending on settings in the CPACR, NSACR, and HCPR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

#### A1

```
[31 30 29 28] [27 26 25 24] [23 22 21 20] [19 16 15 12] [11 10 9 8] [7 6 5 4] [3 0]
  1 1 1 0 0 1 1 1
```

**A1 variant**

```
VADDL{<c>}{<q>}{<dt> <Qd>, <Dn>, <Dm>}
```

**Decode for this encoding**

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' & Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize; is_vaddw = (op == '1');
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

#### T1

```
[15 14 13 12] [11 10 9 8] [7 6 5 4] [3 0]
  1 1 1 1 1
```

**T1 variant**

```
VADDL{<c>}{<q>}{<dt> <Qd>, <Dn>, <Dm>}
```

**Decode for this encoding**

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' & Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize; is_vaddw = (op == '1');
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

### Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-7314 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-7422 for the A32 instruction set.

### Assembler symbols

```
<c>
```

For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields on page F1-7228*.

```
<q>
```

See *Standard assembler syntax fields on page F1-7228*. 

---

ARM DDI 04871.a Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. F6-8235
ID081822 Non-Confidential
<dt> Is the data type for the elements of the second operand vector, encoded in the "U:size" field. It can have the following values:
- S8 when U = 0, size = 00
- S16 when U = 0, size = 01
- S32 when U = 0, size = 10
- U8 when U = 1, size = 00
- U16 when U = 1, size = 01
- U32 when U = 1, size = 10

<d> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <d>*2.

<D> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<M> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        integer op1;
        if is_vaddw then
            op1 = Int(Elem[Q[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[D[n],e,esize], unsigned);
        result = op1 + Int(Elem[M[e,esize],unsigned];
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.31   VADDW

Vector Add Wide adds corresponding elements in one quadword and one doubleword vector, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of the doubleword operand.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

```
[31 30 29 28][27 26 25 24][23 22 21 20][19 16][15 12][11 10 9 8][7 6 5 4][3 0]
1 1 1 0 0 1 | U | 1 | D | l=11 | Vn | Vd | 0 0 0 1 | N | 0 | M | 0 | Vm
```

**A1 variant**

VADDW{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>

**Decode for this encoding**

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vaddw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

```
[15 14 13][12][11 10 9][8][7 6 5 4][3 0]
1 1 1 | U | 1 | 1 | 1 | 1 | D | l=11 | Vn | Vd | 0 0 0 1 | N | 0 | M | 0 | Vm
```

**T1 variant**

VADDW{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>

**Decode for this encoding**

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vaddw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

**Notes for all encodings**

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

**Assembler symbols**

- `<c>` For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
  
  For encoding T1: see Standard assembler syntax fields on page F1-7228.

- `<q>` See Standard assembler syntax fields on page F1-7228.
Is the data type for the elements of the second operand vector, encoded in the "U:size" field. It can have the following values:

- **S8** when $U = 0, size = 00$
- **S16** when $U = 0, size = 01$
- **S32** when $U = 0, size = 10$
- **U8** when $U = 1, size = 00$
- **U16** when $U = 1, size = 01$
- **U32** when $U = 1, size = 10$

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>`*2.

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as `<Qn>`*2.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        integer op1;
        if is_vaddw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
            result = op1 + Int(Elem[Din[m],e,esize],unsigned);
            Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;

```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.32  **VAND (immediate)**

Vector Bitwise AND (immediate) performs a bitwise AND between a register value and an immediate value, and returns the result into the destination vector.

This instruction is a pseudo-instruction of the VBIC (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of VBIC (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VBIC (immediate) gives the operational pseudocode for this instruction.

**64-bit SIMD vector variant**

Applies when \( Q == 0 \).

VAND{\(<c>\)\}{\(<q>\)}.I16 {\(<Dd>\),} <Dd>, #\(<imm>\)

is equivalent to

VBIC{\(<c>\)\}{\(<q>\)}.I16 <Dd>, #~\(<imm>\)

and is never the preferred disassembly.

**128-bit SIMD vector variant**

Applies when \( Q == 1 \).

VAND{\(<c>\)\}{\(<q>\)}.I16 {\(<Qd>\),} <Qd>, #\(<imm>\)

is equivalent to

VBIC{\(<c>\)\}{\(<q>\)}.I16 <Qd>, #~\(<imm>\)

and is never the preferred disassembly.

**64-bit SIMD vector variant**

Applies when \( Q == 0 \).

VAND{\(<c>\)\}{\(<q>\)}.I32 {\(<Dd>\),} <Dd>, #\(<imm>\)

is equivalent to

VBIC{\(<c>\)\}{\(<q>\)}.I32 <Dd>, #~\(<imm>\)

and is never the preferred disassembly.
**128-bit SIMD vector variant**

Applies when \( Q = 1 \).

\[
\text{VAND}\{<c>\}{<q>}.I32 \{<Qd>,\} <Qd>, \#<imm>
\]

is equivalent to

\[
\text{VBIC}\{<c>\}{<q>}.I32 <Qd>, \#<imm>
\]

and is never the preferred disassembly.

**T1**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 0 | 15 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
| 1  | 1  | 1  | 1  | 1  | 1  | 1 | D | 0 | 0 | 0 | imm3 | Vd | 0 | x | x | 1 | 0 | Q | 1 | 1 | imm4 |

**64-bit SIMD vector variant**

Applies when \( Q = 0 \).

\[
\text{VAND}\{<c>\}{<q>}.I16 \{<Dd>,\} <Dd>, \#<imm>
\]

is equivalent to

\[
\text{VBIC}\{<c>\}{<q>}.I16 <Dd>, \#<imm>
\]

and is never the preferred disassembly.

**128-bit SIMD vector variant**

Applies when \( Q = 1 \).

\[
\text{VAND}\{<c>\}{<q>}.I16 \{<Qd>,\} <Qd>, \#<imm>
\]

is equivalent to

\[
\text{VBIC}\{<c>\}{<q>}.I16 <Qd>, \#<imm>
\]

and is never the preferred disassembly.

**T2**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 0 | 15 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
| 1  | 1  | 1  | 1  | 1  | 1  | 1 | D | 0 | 0 | 0 | imm3 | Vd | 1 | 0 | x | 1 | 0 | Q | 1 | 1 | imm4 |

**64-bit SIMD vector variant**

Applies when \( Q = 0 \).

\[
\text{VAND}\{<c>\}{<q>}.I32 \{<Dd>,\} <Dd>, \#<imm>
\]

is equivalent to

\[
\text{VBIC}\{<c>\}{<q>}.I32 <Dd>, \#<imm>
\]

and is never the preferred disassembly.

**128-bit SIMD vector variant**

Applies when \( Q = 1 \).
VAND{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I32 <Qd>, #~<imm>

and is never the preferred disassembly.

**Assembler symbols**

- `<c>` For encoding A1 and A2: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.
  
  For encoding T1 and T2: see *Standard assembler syntax fields on page F1-7228*.

- `<q>` See *Standard assembler syntax fields on page F1-7228*.

- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- `<imm>` Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<imm>`, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions on page F1-7245*.

**Operation for all encodings**

The description of VBIC (immediate) gives the operational pseudocode for this instruction.
F6.1.33 VAND (register)

Vector Bitwise AND (register) performs a bitwise AND operation between two registers, and places the result in the destination register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 |12|11 10 9 8 |7 6 5 4 |3 0 |
|1 1 1 0 1 0 0 |Vn| Vd|0 0 1 |N |Q |M |1 |Vm|

64-bit SIMD vector variant

Applies when Q == 0.

VAND{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VAND{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

\[ d = \text{UInt}(D:Vd); \quad n = \text{UInt}(N:Vn); \quad m = \text{UInt}(M:Vm); \quad \text{regs} = \text{if Q == '0' then 1 else 2}; \]

T1

| 15 14 13 12|11 10 9 8 |7 6 5 4 |3 0 |15 |12|11 10 9 8 |7 6 5 4 |3 0 |
|1 1 |0 1 1 1 0 |D |0 0 |Vn| Vd|0 0 1 |N |Q |M |1 |Vm|

64-bit SIMD vector variant

Applies when Q == 0.

VAND{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VAND{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

\[ d = \text{UInt}(D:Vd); \quad n = \text{UInt}(N:Vn); \quad m = \text{UInt}(M:Vm); \quad \text{regs} = \text{if Q == '0' then 1 else 2}; \]

Assembler symbols

<...> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

See Standard assembler syntax fields on page F1-7228.

An optional data type. It is ignored by assemblers, and does not affect the encoding.

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] AND D[m+r];
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.34   VBIC (immediate)

Vector Bitwise Bit Clear (immediate) performs a bitwise AND between a register value and the complement of an immediate value, and returns the result into the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

This instruction is used by the pseudo-instruction VAND (immediate). The pseudo-instruction is never the preferred disassembly.

A1

64-bit SIMD vector variant
Applies when Q == 0.

VBIC{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>

128-bit SIMD vector variant
Applies when Q == 1.

VBIC{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>

Decode for all variants of this encoding
if cmode<0> = '0' || cmode<3:2> = '11' then SEE "Related encodings";
if Q == '1' && Vd<0> = '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;

A2

64-bit SIMD vector variant
Applies when Q == 0.

VBIC{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>

128-bit SIMD vector variant
Applies when Q == 1.

VBIC{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
**Decode for all variants of this encoding**

if \texttt{cmode<0>} == '0' || \texttt{cmode<3:2>} == '11' then SEE "Related encodings";
if \texttt{Q} == '1' \&\& \texttt{Vd<0>} == '1' then UNDEFINED;
imm64 = \texttt{AdvSIMDExpandImm('1', cmode, i:imm3:imm4)};
d = UInt(D:Vd);  regs = if \texttt{Q} == '0' then 1 else 2;

T1

```
| 15 14 13 12 | 11 10 9 8 7 6 5 4 3 2 0 | 12 11 8 7 6 5 4 3 0 |
| 1 1 1 | 1 1 1 1 | D 0 0 0 | imm3 | Vd 0 x x 1 0 Q 1 1 | imm4 |
```

64-bit SIMD vector variant

Applies when \texttt{Q} == 0.

\texttt{VBIC}<c>{<q>}.I32 \{<Dd>,} <Dd>, \#\texttt{imm}

128-bit SIMD vector variant

Applies when \texttt{Q} == 1.

\texttt{VBIC}<c>{<q>}.I32 \{<Qd>,} <Qd>, \#\texttt{imm}

**Decode for all variants of this encoding**

if \texttt{cmode<0>} == '0' || \texttt{cmode<3:2>} == '11' then SEE "Related encodings";
if \texttt{Q} == '1' \&\& \texttt{Vd<0>} == '1' then UNDEFINED;
imm64 = \texttt{AdvSIMDExpandImm('1', cmode, i:imm3:imm4)};
d = UInt(D:Vd);  regs = if \texttt{Q} == '0' then 1 else 2;

T2

```
| 15 14 13 12 | 11 10 9 8 7 6 5 4 3 2 0 | 12 11 8 7 6 5 4 3 0 |
| 1 1 1 | 1 1 1 1 | D 0 0 0 | imm3 | Vd 1 0 x 1 0 Q 1 1 | imm4 |
```

64-bit SIMD vector variant

Applies when \texttt{Q} == 0.

\texttt{VBIC}<c>{<q>}.I16 \{<Dd>,} <Dd>, \#\texttt{imm}

128-bit SIMD vector variant

Applies when \texttt{Q} == 1.

\texttt{VBIC}<c>{<q>}.I16 \{<Qd>,} <Qd>, \#\texttt{imm}

**Decode for all variants of this encoding**

if \texttt{cmode<0>} == '0' || \texttt{cmode<3:2>} == '11' then SEE "Related encodings";
if \texttt{Q} == '1' \&\& \texttt{Vd<0>} == '1' then UNDEFINED;
imm64 = \texttt{AdvSIMDExpandImm('1', cmode, i:imm3:imm4)};
d = UInt(D:Vd);  regs = if \texttt{Q} == '0' then 1 else 2;
Notes for all encodings

Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.

Assembler symbols

<<c>>  For encoding A1 and A2: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1 and T2: see Standard assembler syntax fields on page F1-7228.

<<q>>  See Standard assembler syntax fields on page F1-7228.

<Qd>>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<int>>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<imm>>  Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see Modified immediate constants in T32 and A32 Advanced SIMD instructions on page F1-7245.

The I8, I64, and F32 data types are permitted as pseudo-instructions, if the immediate can be represented by this instruction, and are encoded using a permitted encoding of the I16 or I32 data type.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    D[d+r] = D[d+r] AND NOT(imm64);

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.35   VBIC (register)

Vector Bitwise Bit Clear (register) performs a bitwise AND between a register value and the complement of a register value, and places the result in the destination register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

A1

64-bit SIMD vector variant
Applies when \( Q = 0 \).

\[
\text{VBIC}\{<c>\}{<q>}{<.dt>} \{<Dd>,} <Dn>, <Dm>
\]

128-bit SIMD vector variant
Applies when \( Q = 1 \).

\[
\text{VBIC}\{<c>\}{<q>}{<.dt>} \{<Qd>,} <Qn>, <Qm>
\]

**Decode for all variants of this encoding**

if \( Q = '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') \) then UNDEFINED;

\[
d = \text{UInt}(D:Vd); \quad n = \text{UInt}(N:Vn); \quad m = \text{UInt}(M:Vm); \quad \text{regs} = \text{if } Q == '0' \text{ then 1 else 2};
\]

T1

64-bit SIMD vector variant
Applies when \( Q = 0 \).

\[
\text{VBIC}\{<c>\}{<q>}{<.dt>} \{<Dd>,} <Dn>, <Dm>
\]

128-bit SIMD vector variant
Applies when \( Q = 1 \).

\[
\text{VBIC}\{<c>\}{<q>}{<.dt>} \{<Qd>,} <Qn>, <Qm>
\]

**Decode for all variants of this encoding**

if \( Q = '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') \) then UNDEFINED;

\[
d = \text{UInt}(D:Vd); \quad n = \text{UInt}(N:Vn); \quad m = \text{UInt}(M:Vm); \quad \text{regs} = \text{if } Q == '0' \text{ then 1 else 2};
\]

**Assembler symbols**

\(<c>\)

For encoding A1: see *Standard assembler syntax fields* on page F1-7228. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields on page F1-7228.*

- An optional data type. It is ignored by assemblers, and does not affect the encoding.

- Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.

- Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as `<Qn>*2.

- Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.

- Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

- Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] AND NOT(D[m+r]);
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.36    VBIF

Vector Bitwise Insert if False inserts each bit from the first source register into the destination register if the corresponding bit of the second source register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

|31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8|7 6 5 4|3 0 |
|---|---|---|---|---|---|---|---|---|
|1 1 1 1 0 0 1 1|0 D 1 1|Vn|Vd|0 0 0 1|N Q M 1|Vm|

64-bit SIMD vector variant

Applies when $Q == 0$.

$\text{VBIF} <c> <q> {<dt>} <Dd>, <Dn>, <Dm>$

128-bit SIMD vector variant

Applies when $Q == 1$.

$\text{VBIF} <c> <q> {<dt>} <Qd>, <Qn>, <Qm>$

Decode for all variants of this encoding

if $Q == '1'$ && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;

$\text{d} = \text{UInt(D:Vd)}; \quad \text{n} = \text{UInt(N:Vn)}; \quad \text{m} = \text{UInt(M:Vm)}; \quad \text{regs} = \text{if Q == '0' then 1 else 2};$

T1

|15 14 13 12|11 10 9 8|7 6 5 4|3 0|15 12|11 10 9 8|7 6 5 4|3 0 |
|---|---|---|---|---|---|---|---|---|
|1 1 1 1|1 1 1 1|0|D 1 1|Vn|Vd|0 0 0 1|N Q M 1|Vm|

64-bit SIMD vector variant

Applies when $Q == 0$.

$\text{VBIF} <c> <q> {<dt>} <Dd>, <Dn>, <Dm>$

128-bit SIMD vector variant

Applies when $Q == 1$.

$\text{VBIF} <c> <q> {<dt>} <Qd>, <Qn>, <Qm>$

Decode for all variants of this encoding

if $Q == '1'$ && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Assembler symbols

<q> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<dt> See Standard assembler syntax fields on page F1-7228.

<dt> An optional data type. It is ignored by assemblers, and does not affect the encoding.

<qdb> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        case operation of
            when VBitOps_VBIF  D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]));
            when VBitOps_VBIT  D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]));
            when VBitOps_VBSL  D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]));

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.37  VBIT

Vector Bitwise Insert if True inserts each bit from the first source register into the destination register if the corresponding bit of the second source register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 16 | 15 12 | 11 10 9 | 8 7 6 5 4 | 3 0 |
|-------------|-------------|-------------|-----|-----|------|------|------|-----|
| 1 1 1 1 0 | 0 1 1 0 0 | 1 0 | Vn | Vd | 0 0 0 1 | N Q M 1 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VBIT{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VBIT{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 | 11 10 9 | 8 7 6 5 4 | 3 0 | 12 | 11 10 9 | 8 7 6 5 4 | 3 0 |
|-------------|------|------|------|-----|------|------|------|-----|
| 1 1 1 1 0 | 1 1 1 1 0 | 1 0 | Vn | Vd | 0 0 0 1 | N Q M 1 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VBIT{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VBIT{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Assembler symbols

<< For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<< See Standard assembler syntax fields on page F1-7228.

dt An optional data type. It is ignored by assemblers, and does not affect the encoding.

<q> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<n> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<m> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<d> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<n> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<m> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        case operation of
            when VBitOps_VBIF  D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]));
            when VBitOps_VBIT  D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]));
            when VBitOps_VBSL  D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]));

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.38 VBSL

Vector Bitwise Select sets each bit in the destination to the corresponding bit from the first source operand when the original destination bit was 1, otherwise from the second source operand.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 12 11 10 9 8 7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1 0</td>
<td>D</td>
<td>0</td>
<td>1</td>
<td>Vn</td>
<td>Vd</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VBSL{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VBSL{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4</th>
<th>3 0</th>
<th>15 12 11 10 9 8 7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 0</td>
<td>D</td>
<td>0</td>
<td>1</td>
<td>Vn</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VBSL{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VBSL{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
    d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Assembler symbols
<>
For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.
<q>
See Standard assembler syntax fields on page F1-7228.
<dt>
An optional data type. It is ignored by assemblers, and does not affect the encoding.
<d>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<n>
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<m>
Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<db>
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<dn>
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<dm>
Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
    case operation of
        when VBitOps_VBIF  D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]));
        when VBitOps_VBIT  D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]));
        when VBitOps_VBSL  D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]));

Operational information
If CPSR.DIT is 1 and this instruction passes its condition execution check:
• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.39 VCADD

Vector Complex Add.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 90 or 270 degrees.
- The rotated complex number is added to the complex number from the first source register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

(FEAT_FCMA)

```
[31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 ]
1 1 1 1 1 1 1 0 | rot 1 | D | O | S | Vn | Vd | 1 | 0 | 0 | 0 | N | Q | M | 0 | Vm
```

64-bit SIMD vector variant

Applies when Q == 0.

```
VCADD{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>
```

128-bit SIMD vector variant

Applies when Q == 1.

```
VCADD{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>
```

Decode for all variants of this encoding

```
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
```

T1

(FEAT_FCMA)

```
[15 14 13 12 | 11 10 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 ]
1 1 1 1 1 1 1 0 | rot 1 | D | O | S | Vn | Vd | 1 | 0 | 0 | 0 | N | Q | M | 0 | Vm
```

64-bit SIMD vector variant

Applies when Q == 0.

```
VCADD{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>
```
128-bit SIMD vector variant

Applies when Q == 1.

VCADD{<q>;<dt>} <Qd>, <Qn>, <Qm>, #<rotate>

Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
reg = if Q == '0' then 1 else 2;

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the vectors, encoded in the "S" field. It can have the following values:
   F16 when S = 0
   F32 when S = 1

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<rotate> Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in the "rot" field. It can have the following values:
   90 when rot = 0
   270 when rot = 1

Operation for all encodings

EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1
   operand1 = D[n+r];
   operand2 = D[m+r];
for e = 0 to (elements DIV 2)-1
   bits(size) element1;
   bits(size) element3;
   case rot of
      when '0'
         element1 = FPNeg(Elem[operand2,e*2+1,esize]);
         element3 = Elem[operand2,e*2,esize];
      when '1'
         element1 = Elem[operand2,e*2+1,esize];
         element3 = FPNeg(Elem[operand2,e*2,esize]);
   result1 = FPAdd(Elem[operand1,e*2,esize],element1,StandardFPSCRValue());
   result2 = FPAdd(Elem[operand1,e*2+1,esize],element3,StandardFPSCRValue());
\texttt{Elem[D+r],e=2,esize} = \text{result1};
\texttt{Elem[D+r],e=2+1,esize} = \text{result2};
F6.1.40   VCEQ (immediate #0)

Vector Compare Equal to Zero takes each element in a vector, and compares it with zero. If it is equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 | 12 | 11 10 9 8 7 6 5 4 3 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | D | 1 | 1 | size | 0 | 1 | Vd | 0 | F | 0 | 1 | 0 | Q | M | 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VCEQ{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0

128-bit SIMD vector variant

Applies when Q == 1.

VCEQ{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 | 11 10 9 8 7 6 5 4 3 2 1 0 | 15 | 12 | 11 10 9 8 7 6 5 4 3 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | D | 1 | 1 | size | 0 | 1 | Vd | 0 | F | 0 | 1 | 0 | Q | M | 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VCEQ{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0

128-bit SIMD vector variant

Applies when Q == 1.

VCEQ{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRANDED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<ct> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

<ct> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the elements of the operands, encoded in the "F:size" field. It can have the following values:

I8 when F = 0, size = 00
I16 when F = 0, size = 01
I32 when F = 0, size = 10
F16 when F = 1, size = 01
F32 when F = 1, size = 10

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
EncodingSpecificOperations();  CheckAdvSIMDEnabled();
for r = 0 to regs-1
for e = 0 to elements-1
  boolean test_passed;
  if floating_point then
    bits(esize) zero = FPZero('0', esize);
    test_passed = FPCompareEQ(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
  else
    test_passed = (Elem[D[m+r],e,esize] == Zeros(esize));
    Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);

F6.1.41   VCEQ (register)

Vector Compare Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If they are equal, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support on page G1-8996*.

A1

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 | 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|-------------|-------------|-------------|---|----|----|----|--------|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | D | size | Vn | Vd | 1 | 0 | 0 | 0 | N | Q | M | 1 | Vm |

**64-bit SIMD vector variant**

Applies when Q == 0.

VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

**128-bit SIMD vector variant**

Applies when Q == 1.

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

**Decode for all variants of this encoding**

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
int_operation = TRUE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

A2

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 | 12 | 11 10 9 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|-------------|-------------|-------------|---|----|----|----|--------|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | D | sz | Vn | Vd | 1 | 1 | 0 | N | Q | M | 0 | Vm |

**64-bit SIMD vector variant**

Applies when Q == 0.

VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

**128-bit SIMD vector variant**

Applies when Q == 1.

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
### Decode for all variants of this encoding

```c
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
int_operation = FALSE;
g = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

**T1**

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8</th>
<th>7 6 5 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1 0 1 0 0 0 N Q M 1</td>
<td>Vm</td>
<td></td>
</tr>
</tbody>
</table>

#### 64-bit SIMD vector variant

Applies when Q == 0.

VCEQ{<c>}{<q>}.<dt> {<Db>, }<Db>, <Db> <Qd>, <Qd>, <Qd>

#### 128-bit SIMD vector variant

Applies when Q == 1.

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qd>, <Qd>, <Qd>

### Decode for all variants of this encoding

```c
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
int_operation = TRUE; esize = 8 << UInt(sz); elements = 64 DIV esize;
g = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

**T2**

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8</th>
<th>7 6 5 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 1 1 1 0 1 0 0 N Q M 0</td>
<td>Vm</td>
<td></td>
</tr>
</tbody>
</table>

#### 64-bit SIMD vector variant

Applies when Q == 0.

VCEQ{<c>}{<q>}.<dt> {<Db>, }<Db>, <Db> <Qd>, <Qd>, <Qd>

#### 128-bit SIMD vector variant

Applies when Q == 1.

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qd>, <Qd>, <Qd>

### Decode for all variants of this encoding

```c
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
```
int_operation = FALSE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRANDED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<> For encoding A1 and A2: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1 and T2: see Standard assembler syntax fields on page F1-7228.
<>> See Standard assembler syntax fields on page F1-7228.
<dt> For encoding A1 and T1: is the data type for the elements of the vectors, encoded in the "size" field.
It can have the following values:
I8 when size = 00
I16 when size = 01
I32 when size = 10
For encoding A2 and T2: is the data type for the elements of the vectors, encoded in the "sz" field.
It can have the following values:
F32 when sz = 0
F16 when sz = 1
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = Elem[D[n+r],e,esize]; op2 = Elem[D[m+r],e,esize];
      boolean test_passed;
      if int_operation then
        test_passed = (op1 == op2);
      else
test_passed = FPCompareEQ(op1, op2, StandardFPSCRValue());
Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
F6.1.42   VCGE (immediate #0)

Vector Compare Greater Than or Equal to Zero takes each element in a vector, and compares it with zero. If it is
greater than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is
set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector
elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which
the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For
more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VCGE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0

128-bit SIMD vector variant

Applies when Q == 1.

VCGE{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when Q == 0.

VCGE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0

128-bit SIMD vector variant

Applies when Q == 1.

VCGE{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if F == '1' && (size == '01' && HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONstrained UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<q>  For encoding T1: see Standard assembler syntax fields on page F1-7228.

<dt>  Is the data type for the elements of the operands, encoded in the "F:size" field. It can have the following values:
S8  when F = 0, size = 00
S16 when F = 0, size = 01
S32 when F = 0, size = 10
F16 when F = 1, size = 01
F32 when F = 1, size = 10

<Qd>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>  Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<dm>  Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      boolean test_passed;
      if floating_point then
        bits(esize) zero = FPZero('0', esize);
        test_passed = FPCompareGE(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
      else
        test_passed = (SInt(Elem[D[m+r],e,esize]) >= 0);
      Elem[D[r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.43 VCGE (register)

Vector Compare Greater Than or Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers, unsigned integers, or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

This instruction is used by the pseudo-instruction VCLE (register). The pseudo-instruction is never the preferred disassembly.

A1

| [31 30 29 28]27 26 25 24|23 22 21 20 |19 16|15 12|11 10 9 8 |7 6 5 4 |3 0 |1 1 1 1 0 1 |U 0 |D |size |Vn |Vd |0 0 1 1 |N |Q |M |1 |Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
vtype = if U == '1' then VCGEType_unsigned else VCGEType_signed;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

A2

| [31 30 29 28]27 26 25 24|23 22 21 20 |19 16|15 12|11 10 9 8 |7 6 5 4 |3 0 |1 1 1 1 0 1 |1 0 |D |sz |Vn |Vd |1 1 1 0 |N |Q |M |0 |Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
Decode for all variants of this encoding

if \( Q = '1' \) \&\& (\( Vd<0> = '1' \) \|\| \( Vn<0> = '1' \) \|\| \( Vm<0> = '1' \)) then UNDEFINED;
if \( sz = '1' \) \&\& !HaveFP16Ext() then UNDEFINED;
vtype = VCGEType_fp;
integer esize;
integer elements;
case \( sz \) of
when '0' esize = 32; elements = 2;
when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if \( Q = '0' \) then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3</th>
<th>0</th>
<th>15 12 11 10 9 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 0 D size Vn Vd</td>
<td>0 0 1 1 N Q M 1 Vm</td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant
Applies when \( Q = 0 \).
\( \text{VCGE\{<c>\}{<q>}.<dt> \{<Dd>, }<Dn>, <Dm} \)

128-bit SIMD vector variant
Applies when \( Q = 1 \).
\( \text{VCGE\{<c>\}{<q>}.<dt> \{<Qd>, }<Qn>, <Qm} \)

Decode for all variants of this encoding

if \( Q = '1' \) \&\& (\( Vd<0> = '1' \) \|\| \( Vn<0> = '1' \) \|\| \( Vm<0> = '1' \)) then UNDEFINED;
if \( sz = '11' \) then UNDEFINED;
vtype = if \( U = '1' \) then VCGEType_unsigned else VCGEType_signed;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if \( Q = '0' \) then 1 else 2;

T2

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3</th>
<th>0</th>
<th>15 12 11 10 9 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 0 D 0 sz Vn Vd</td>
<td>1 1 1 0 N Q M 0 Vm</td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant
Applies when \( Q = 0 \).
\( \text{VCGE\{<c>\}{<q>}.<dt> \{<Dd>, }<Dn>, <Dm} \)

128-bit SIMD vector variant
Applies when \( Q = 1 \).
\( \text{VCGE\{<c>\}{<q>}.<dt> \{<Qd>, }<Qn>, <Qm} \)
**Decode for all variants of this encoding**

if Q == '1' && (Vd<> == '1' || Vn<> == '1' || Vm<> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;

vtype = VCGEType_fp;
integer esize;
integer elements;

```plaintext
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;

d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

**CONSTRAINED UNPREDICTABLE behavior**

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

<>

For encoding A1 and A2: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields on page F1-7228*.

&lt;

See *Standard assembler syntax fields on page F1-7228*.

&lt;d>

For encoding A1 and T1: is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

- S8 when U = 0, size = 00
- S16 when U = 0, size = 01
- S32 when U = 0, size = 10
- U8 when U = 1, size = 00
- U16 when U = 1, size = 01
- U32 when U = 1, size = 10

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

- F32 when sz = 0
- F16 when sz = 1

<qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <qd>*2.

<qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <qn>*2.

<qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <qm>*2.

<dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            boolean test_passed;
            case vtype of
                when VCGEType_signed    test_passed = (SInt(op1) >= SInt(op2));
                when VCGEType_unsigned  test_passed = (UInt(op1) >= UInt(op2));
                when VCGEType_fp        test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.44 VCGT (immediate #0)

Vector Compare Greater Than Zero takes each element in a vector, and compares it with zero. If it is greater than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

64-bit SIMD vector variant

Applies when Q == 0.

\[
\text{VCGT} \langle c \rangle \langle q \rangle \langle d \rangle \langle m \rangle \#0
\]

128-bit SIMD vector variant

Applies when Q == 1.

\[
\text{VCGT} \langle c \rangle \langle q \rangle \langle d \rangle \langle m \rangle \#0
\]

Decode for all variants of this encoding

- if size == '11' then UNDEFINED;
- if F == '1' && (size == '01' || size == '00') then UNDEFINED;
- if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
- floating_point = (F == '1');
- esize = 8 << UInt(size);
- elements = 64 DIV esize;
- d = UInt(D:Vd);
- m = UInt(M:Vm);
- regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when Q == 0.

\[
\text{VCGT} \langle c \rangle \langle q \rangle \langle d \rangle \langle m \rangle \#0
\]

128-bit SIMD vector variant

Applies when Q == 1.

\[
\text{VCGT} \langle c \rangle \langle q \rangle \langle d \rangle \langle m \rangle \#0
\]
Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<c> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
   
   For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the elements of the operands, encoded in the "F:size" field. It can have the following values:
   
   S8      when F = 0, size = 00
   S16     when F = 0, size = 01
   S32     when F = 0, size = 10
   F16     when F = 1, size = 01
   F32     when F = 1, size = 10

<qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <qd>*2.

<qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <qm>*2.

<dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations(); CheckAdvSIMDEnabled();
   for r = 0 to regs-1
      for e = 0 to elements-1
         boolean test_passed;
         if floating_point then
            bits(esize) zero = FPZero('0', esize);
            test_passed = FPCompareGT(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
         else
            test_passed = (SInt(Elem[D[m+r],e,esize]) > 0);
         Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.45  **VCGT (register)**

Vector Compare Greater Than takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers, unsigned integers, or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

This instruction is used by the pseudo-instruction VCLT (register). The pseudo-instruction is never the preferred disassembly.

### A1

64-bit SIMD vector variant

Applies when Q == 0.

\[ \text{VCGT}(\{c\}\{q\}.<dt> \{<Dd>, }<Dn>, <Dm]\]

128-bit SIMD vector variant

Applies when Q == 1.

\[ \text{VCGT}(\{c\}\{q\}.<dt> \{<Qd>, }<Qn>, <Qm}\]

Decode for all variants of this encoding

\[
\text{if } Q == '1' \&\& (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') \text{ then UNDEFINED;}
\]

\[
\text{if } \text{size} == '11' \text{ then UNDEFINED;}
\]

\[
\text{vtype} = \text{if } U == '1' \text{ then VCGTtype_unsigned else VCGTtype_signed;}
\]

\[
\text{esize} = 8 \times \text{UInt(size); elements} = 64 \text{ DIV esize;}
\]

\[
\text{d} = \text{UInt(D:Vd); } n = \text{UInt(N:Vn); } m = \text{UInt(M:Vm); } \text{regs} = \text{if } Q == '0' \text{ then 1 else 2;}
\]

### A2

64-bit SIMD vector variant

Applies when Q == 0.

\[ \text{VCGT}(\{c\}\{q\}.<dt> \{<Dd>, }<Dn>, <Dm}\]

128-bit SIMD vector variant

Applies when Q == 1.

\[ \text{VCGT}(\{c\}\{q\}.<dt> \{<Qd>, }<Qn>, <Qm}\]

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | U  | 0  | D  | size | Vn | Vd | 0  | 0  | 1  | N  | Q  | M  | 0  | Vm |

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | D  | sz | Vn | Vd | 1  | 1  | 0  | N  | Q  | M  | 0  | Vm |
Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
vtype = VCGTtype_fp;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); reg = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13</th>
<th>12 11 10</th>
<th>9 8</th>
<th>7 6 5</th>
<th>4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
<td>U 1 1 1 0</td>
<td>D</td>
<td>size</td>
<td>Vn</td>
<td>Vd</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant
Applies when Q == 0.
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant
Applies when Q == 1.
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '11' then UNDEFINED;
vtype = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); reg = if Q == '0' then 1 else 2;

T2

<table>
<thead>
<tr>
<th>15 14 13</th>
<th>12 11 10</th>
<th>9 8</th>
<th>7 6 5</th>
<th>4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
<td>1 1 1 1 0</td>
<td>D</td>
<td>1</td>
<td>sz</td>
<td>Vn</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant
Applies when Q == 0.
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant
Applies when Q == 1.
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
vtype = VCGEType_fp;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<> For encoding A1 and A2: see Standard assembler syntax fields on page F1-7228. This encoding
must be unconditional.

For encoding T1 and T2: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<dt> For encoding A1 and T1: is the data type for the elements of the operands, encoded in the "U:size"
field. It can have the following values:

S8 when U = 0, size = 00
S16 when U = 0, size = 01
S32 when U = 0, size = 10
U8 when U = 1, size = 00
U16 when U = 1, size = 01
U32 when U = 1, size = 10

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in the "sz" field.
It can have the following values:

F32 when sz = 0
F16 when sz = 1

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as
<Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize]; op2 = Elem[D[m+r],e,esize];
            boolean test_passed;
            case vtype of
                when VCGTtype_signed    test_passed = (SInt(op1) > SInt(op2));
                when VCGTtype_unsigned  test_passed = (UInt(op1) > UInt(op2));
                when VCGTtype_fp        test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.46 **VCLE (immediate #0)**

Vector Compare Less Than or Equal to Zero takes each element in a vector, and compares it with zero. If it is less than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

A1

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 12 11 10 9 8 7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 1</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>size</td>
<td>0</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector variant**

Applies when Q == 0.

VCLE{<c>}{<q>}.{dt} {<Dd>,} <Dm>, #0

**128-bit SIMD vector variant**

Applies when Q == 1.

VCLE{<c>}{<q>}.{dt} {<Qd>,} <Qm>, #0

**Decode for all variants of this encoding**

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);
elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4</th>
<th>3 2 1 0</th>
<th>15 12</th>
<th>11 10 9 8 7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>size</td>
<td>0</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector variant**

Applies when Q == 0.

VCLE{<c>}{<q>}.{dt} {<Dd>,} <Dm>, #0

**128-bit SIMD vector variant**

Applies when Q == 1.

VCLE{<c>}{<q>}.{dt} {<Qd>,} <Qm>, #0
Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<
For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

<dt>
Is the data type for the elements of the operands, encoded in the "F:size" field. It can have the following values:

S8 when F = 0, size = 00
S16 when F = 0, size = 01
S32 when F = 0, size = 10
F16 when F = 1, size = 01
F32 when F = 1, size = 10

<qd>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <qd>*2.

<qm>
Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <qm>*2.

<dd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<dm>
Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      boolean test_passed;
      if floating_point then
        bits(esize) zero = FPZero('0', esize);
        test_passed = FPCompareGE(zero, Elem[D[m+r],e,esize], StandardFPSCRValue());
      else
        test_passed = (SInt(Elem[D[m+r],e,esize]) <= 0);
        Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.47  VCLE (register)

Vector Compare Less Than or Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is less than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This instruction is a pseudo-instruction of the VCGE (register) instruction. This means that:

- The encodings in this description are named to match the encodings of VCGE (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VCGE (register) gives the operational pseudocode for this instruction.

A1

- **64-bit SIMD vector variant**
  - Applies when Q == 0.
  - VCLE\{<c>\}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>
  - is equivalent to
  - VCGE\{<c>\}{<q>}.<dt> <Dd>, <Dm>, <Dn>
  - and is never the preferred disassembly.

- **128-bit SIMD vector variant**
  - Applies when Q == 1.
  - VCLE\{<c>\}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>
  - is equivalent to
  - VCGE\{<c>\}{<q>}.<dt> <Qd>, <Qm>, <Qn>
  - and is never the preferred disassembly.

A2

- **64-bit SIMD vector variant**
  - Applies when Q == 0.
  - VCLE\{<c>\}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>
  - is equivalent to
  - VCGE\{<c>\}{<q>}.<dt> <Dd>, <Dm>, <Dn>
  - and is never the preferred disassembly.
**128-bit SIMD vector variant**

Applies when \( Q = 1 \).

\[
\text{VCLE}\{<c>\}{<q>}.\{dt\} \{<Qd>, <Qn>, <Qm> \}
\]

is equivalent to

\[
\text{VCGE}\{<c>\}{<q>}.\{dt\} <Qd>, <Qm>, <Qn>
\]

and is never the preferred disassembly.

**T1**

```
| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 | 0 | 15 | 12 | 11 10 9 8 | 7 6 5 4 | 3 | 0 |
|---------------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 1 1 1 U       | 1 1 1 1 0 | D sz     | Vn       | Vd       | 0 0 1 1 N | Q M 1    | Vm       |
```

**64-bit SIMD vector variant**

Applies when \( Q = 0 \).

\[
\text{VCLE}\{<c>\}{<q>}.\{dt\} \{<Dd>, <Dn>, <Dm> \}
\]

is equivalent to

\[
\text{VCGE}\{<c>\}{<q>}.\{dt\} <Dd>, <Dm>, <Dn>
\]

and is never the preferred disassembly.

**128-bit SIMD vector variant**

Applies when \( Q = 1 \).

\[
\text{VCLE}\{<c>\}{<q>}.\{dt\} \{<Qd>, <Qn>, <Qm> \}
\]

is equivalent to

\[
\text{VCGE}\{<c>\}{<q>}.\{dt\} <Qd>, <Qm>, <Qn>
\]

and is never the preferred disassembly.

**T2**

```
| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 | 0 | 15 | 12 | 11 10 9 8 | 7 6 5 4 | 3 | 0 |
|---------------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 1 1 1 U       | 1 1 1 1 0 | D sz     | Vn       | Vd       | 1 1 1 0 N | Q M 0    | Vm       |
```

**64-bit SIMD vector variant**

Applies when \( Q = 0 \).

\[
\text{VCLE}\{<c>\}{<q>}.\{dt\} \{<Dd>, <Dn>, <Dm> \}
\]

is equivalent to

\[
\text{VCGE}\{<c>\}{<q>}.\{dt\} <Dd>, <Dm>, <Dn>
\]

and is never the preferred disassembly.

**128-bit SIMD vector variant**

Applies when \( Q = 1 \).
VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VCGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

and is never the preferred disassembly.

**Assembler symbols**

- `<Dm>` is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
- `<Dn>` is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- `<Qm>` is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2`.
- `<Qn>` is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as `<Qn>*2`.
- `<c>` For encoding A1 and A2: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
- For encoding T1 and T2: see Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<dt>` For encoding A1 and T1: is the data type for the elements of the operands, encoded in the "U:sz" field. It can have the following values:
  - S8 when U = 0, size = 00
  - S16 when U = 0, size = 01
  - S32 when U = 0, size = 10
  - U8 when U = 1, size = 00
  - U16 when U = 1, size = 01
  - U32 when U = 1, size = 10
- For encoding A2 and T2: is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:
  - F32 when sz = 0
  - F16 when sz = 1

- `<Qd>` is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2`.
- `<Dd>` is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

**Operation for all encodings**

The description of VCGE (register) gives the operational pseudocode for this instruction.

**Operational information**

The description of VCGE (register) gives the operational pseudocode for this instruction.
F6.1.48 VCLS

Vector Count Leading Sign Bits counts the number of consecutive bits following the topmost bit, that are the same as the topmost bit, in each element in a vector, and places the results in a second vector. The count does not include the topmost bit itself.

The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit signed integers.

The result vector elements are the same data type as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VCLS{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VCLS{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when Q == 0.

VCLS{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VCLS{<c>}{<q>}.<dt> <Qd>, <Qm>
Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Assembler symbols

<⟩ For encoding A1: see Standard assemble r syntax fields on page F1-7228. This encoding must be unconditional.
   For encoding T1: see Standard assemble r syntax fields on page F1-7228.
<α⟩ See Standard assemble r syntax fields on page F1-7228.
<dt⟩ Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:
S8 when size = 00
S16 when size = 01
S32 when size = 10
   The encoding size = 11 is reserved.
<Qd⟩ Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm⟩ Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd⟩ Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm⟩ Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations(); CheckAdvSIMDEnabled();
   for r = 0 to regs-1
      for e = 0 to elements-1
         Elem[D[d+r],e,esize] = CountLeadingSignBits(Elem[D[m+r],e,esize])<esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
   — The values of the data supplied in any of its registers.
   — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
   — The values of the data supplied in any of its registers.
   — The values of the NZCV flags.
F6.1.49  VCLT (immediate #0)

Vector Compare Less Than Zero takes each element in a vector, and compares it with zero. If it is less than zero, 
the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector 
elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPtr registers, and the Security state and PE mode in which 
the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For 
more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

A1

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 | 12 | 11 10 9 | 8 | 7 6 5 4 | 3 | 0 |
|-------------|---------------|---------------|---------------|---------------|---------------|---------------|---|---------------|---|---------------|---|---------------|---|---------------|
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | D | 1 | 1 | size | 0 | 1 | Vd | 0 | F | 1 | 0 | 0 | Q | M | 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VCLT{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0

128-bit SIMD vector variant

Applies when Q == 1.

VCLT{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 | 11 10 9 | 8 | 7 6 5 4 | 3 | 2 | 1 | 0 | 15 | 12 | 11 10 9 | 8 | 7 6 5 4 | 3 | 0 |
|---------------|---------------|---|---------------|---|---------------|---|---------------|---|---------------|---|---------------|---|---------------|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | D | 1 | 1 | size | 0 | 1 | Vd | 0 | F | 1 | 0 | 0 | Q | M | 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VCLT{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0

128-bit SIMD vector variant

Applies when Q == 1.

VCLT{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;

floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRANGED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<=> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the elements of the operands, encoded in the "F:size" field. It can have the following values:
S8 when F = 0, size = 00
S16 when F = 0, size = 01
S32 when F = 0, size = 10
F16 when F = 1, size = 01
F32 when F = 1, size = 10

<qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <qd>*2.

<qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <qm>*2.

<dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      boolean test_passed;
      if floating_point then
        bits(esize) zero = FPZero('0', esize);
        test_passed = FPCompareGT(zero, Elem[D[m+r],e,esize], StandardFPSCRValue());
      else
        test_passed = (SInt(Elem[D[m+r],e,esize]) < 0);
        Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.50  VCLT (register)

Vector Compare Less Than takes each element in a vector, and compares it with the corresponding element of a
second vector. If the first is less than the second, the corresponding element in the destination vector is set to all
ones. Otherwise, it is set to all zeros.

This instruction is a pseudo-instruction of the VCGT (register) instruction. This means that:

- The encodings in this description are named to match the encodings of VCGT (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VCGT (register) gives the operational pseudocode for this instruction.

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 6 5 4 3 0 |
|------------------|------------------|------------------|
| 1 1 1 1 0 0 1 U 0 D size | Vn | Vd | 0 0 1 1 N Q M 0 Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VCLT{<c>}{<q>}.dt {<Dd>, }<Dm>, <Dn>

is equivalent to

VCGT{<c>}{<q>}.dt <Dd>, <Dm>, <Dn>

and is never the preferred disassembly.

128-bit SIMD vector variant

Applies when Q == 1.

VCLT{<c>}{<q>}.dt {<Qd>, }<Qm>, <Qn>

is equivalent to

VCGT{<c>}{<q>}.dt <Qd>, <Qm>, <Qn>

and is never the preferred disassembly.

A2

| 31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 6 5 4 3 0 |
|------------------|------------------|------------------|
| 1 1 1 1 0 0 1 U 0 D 1 sz | Vn | Vd | 1 1 1 0 N Q M 0 Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VCLT{<c>}{<q>}.dt {<Dd>, }<Dm>, <Dn>

is equivalent to

VCGT{<c>}{<q>}.dt <Dd>, <Dm>, <Dn>

and is never the preferred disassembly.
**128-bit SIMD vector variant**

Applies when $Q = 1$.

$VCLT\{<c>{<q>}.<dt> \{<Qd>, }<Qn>, <Qm>$

is equivalent to

$VCGT\{<c>{<q>}.<dt> <Qd>, <Qm>, <Qn>$

and is never the preferred disassembly.

**T1**

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
<th>15 12 11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0</td>
<td>D</td>
<td>size</td>
<td>Vn</td>
<td>Vd</td>
<td>0 1 1</td>
<td>N Q M 0</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector variant**

Applies when $Q = 0$.

$VCLT\{<c>{<q>}.<dt> \{<Dd>, }<Dn>, <Dm>$

is equivalent to

$VCGT\{<c>{<q>}.<dt> <Dd>, <Dm>, <Dn>$

and is never the preferred disassembly.

**128-bit SIMD vector variant**

Applies when $Q = 1$.

$VCLT\{<c>{<q>}.<dt> \{<Qd>, }<Qn>, <Qm>$

is equivalent to

$VCGT\{<c>{<q>}.<dt> <Qd>, <Qm>, <Qn>$

and is never the preferred disassembly.

**T2**

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
<th>15 12 11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
<td>D</td>
<td>sz</td>
<td>Vn</td>
<td>Vd</td>
<td>1 1 0</td>
<td>N Q M 0</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector variant**

Applies when $Q = 0$.

$VCLT\{<c>{<q>}.<dt> \{<Dd>, }<Dn>, <Dm>$

is equivalent to

$VCGT\{<c>{<q>}.<dt> <Dd>, <Dm>, <Dn>$

and is never the preferred disassembly.

**128-bit SIMD vector variant**

Applies when $Q = 1$. 

---

F6-8290  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  ARM DDI 0487I.a  Non-Confidential  ID081822
VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

and is never the preferred disassembly.

**Assembler symbols**

- **<Dm>**
  - Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

- **<Dn>**
  - Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

- **<Qm>**
  - Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

- **<Qn>**
  - Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

- **<<**
  - For encoding A1 and A2: see *Standard assembler syntax fields* on page F1-7228. This encoding must be unconditional.
  - For encoding T1 and T2: see *Standard assembler syntax fields* on page F1-7228.

- **<q>**
  - See *Standard assembler syntax fields* on page F1-7228.

- **<dt>**
  - For encoding A1 and T1: is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:
    - S8     when U = 0, size = 00
    - S16    when U = 0, size = 01
    - S32    when U = 0, size = 10
    - U8     when U = 1, size = 00
    - U16    when U = 1, size = 01
    - U32    when U = 1, size = 10
  - For encoding A2 and T2: is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:
    - F32    when sz = 0
    - F16    when sz = 1

- **<Qd>**
  - Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

- **<Dd>**
  - Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

**Operation for all encodings**

The description of VCGT (register) gives the operational pseudocode for this instruction.

**Operational information**

The description of VCGT (register) gives the operational pseudocode for this instruction.
F6.1.51  VCLZ

Vector Count Leading Zeros counts the number of consecutive zeros, starting from the most significant bit, in each element in a vector, and places the results in a second vector.

The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

The result vector elements are the same data type as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

**A1**

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c}
\text{[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15|12|11 10 9 8|7 6 5 4|3|0]}
\text{[0 1 1 1 1 0 0 1 1 1 |D 1 1 |size 0 0 |Vd 0 1 0 0 1 |Q |M 0 |Vm]}
\end{array}
\]

**64-bit SIMD vector variant**

Applies when \( Q == 0 \).

\[ \text{VCLZ}{<c>}{<q>}.<dt> <Dd>, <Dm> \]

**128-bit SIMD vector variant**

Applies when \( Q == 1 \).

\[ \text{VCLZ}{<c>}{<q>}.<dt> <Qd>, <Qm> \]

**Decode for all variants of this encoding**

- If \( \text{size} == '11' \) then UNDEFINED;
- If \( Q == '1' \) && \( (Vd<0> == '1' || Vm<0> == '1') \) then UNDEFINED;
- \( \text{esize} = 8 << \text{UInt(size)} \); elements = 64 DIV esize;
- \( d = \text{UInt}(D:Vd); \quad m = \text{UInt}(M:Vm); \quad \text{regs} = \text{if } Q == '0' \text{ then 1 else 2}; \)

**T1**

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c}
\text{[15 14 13 12|11 10 9 8|7 6 5 4|3 2 1 0|15|12|11 10 9 8|7 6 5 4|3|0]}
\text{[0 1 1 1 1 1 1 1 1 |D 1 1 |size 0 0 |Vd 0 1 0 0 1 |Q |M 0 |Vm]}
\end{array}
\]

**64-bit SIMD vector variant**

Applies when \( Q == 0 \).

\[ \text{VCLZ}{<c>}{<q>}.<dt> <Dd>, <Dm> \]

**128-bit SIMD vector variant**

Applies when \( Q == 1 \).

\[ \text{VCLZ}{<c>}{<q>}.<dt> <Qd>, <Qm> \]
Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Assembler symbols

<> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:
I8 when size = 00
I16 when size = 01
I32 when size = 10
The encoding size = 11 is reserved.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      Elem[D[d+r],e,esize] = CountLeadingZeroBits(Elem[D[m+r],e,esize])<esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Vector Complex Multiply Accumulate.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers and the destination register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

(FEAT_FCMA)

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8|7 6 5 4|3 |0 |
1 1 1 1 1 1 0 rot D 1 S Vn Vd 1 0 0 0 N Q M 0 Vm
```

64-bit SIMD vector variant

Applies when Q == 0.

VCMLA{<q>}..<dt> <Dd>, <Dn>, <Dm>, #<rotate>

128-bit SIMD vector variant

Applies when Q == 1.

VCMLA{<q>}..<dt> <Qd>, <Qn>, <Qm>, #<rotate>

Decode for all variants of this encoding

if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;

T1

(FEAT_FCMA)
64-bit SIMD vector variant
Applies when Q == 0.

VCMLA{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>

128-bit SIMD vector variant
Applies when Q == 1.

VCMLA{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>

Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the vectors, encoded in the "S" field. It can have the following values:
F16 when S = 0
F32 when S = 1

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<rotate> Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in the "rot" field. It can have the following values:
0 when rot = 00
90 when rot = 01
180 when rot = 10
270 when rot = 11
Operation for all encodings

EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1
  operand1 = D[n+r];
  operand2 = D[m+r];
  operand3 = D[d+r];
  for e = 0 to (elements DIV 2)-1
    bits(esize) element1;
    bits(esize) element2;
    bits(esize) element3;
    bits(esize) element4;
    case rot of
      when '00'
        element1 = Elem[operand2,e*2,esize];
        element2 = Elem[operand1,e*2,esize];
        element3 = Elem[operand2,e*2+1,esize];
        element4 = Elem[operand1,e*2,esize];
      when '01'
        element1 = FPNeg(Elem[operand2,e*2+1,esize]);
        element2 = Elem[operand1,e*2+1,esize];
        element3 = Elem[operand2,e*2,esize];
        element4 = Elem[operand1,e*2+1,esize];
      when '10'
        element1 = FPNeg(Elem[operand2,e*2,esize]);
        element2 = Elem[operand1,e*2,esize];
        element3 = FPNeg(Elem[operand2,e*2+1,esize]);
        element4 = Elem[operand1,e*2,esize];
      when '11'
        element1 = Elem[operand2,e*2+1,esize];
        element2 = Elem[operand1,e*2+1,esize];
        element3 = FPNeg(Elem[operand2,e*2,esize]);
        element4 = Elem[operand1,e*2+1,esize];
    result1 = FPMulAdd(Elem[operand3,e*2,esize],element2,element1, StandardFPSCRValue());
    result2 = FPMulAdd(Elem[operand3,e*2+1,esize],element4,element3, StandardFPSCRValue());
    Elem[D[d+r],e*2,esize] = result1;
    Elem[D[d+r],e*2+1,esize] = result2;
F6.1.53 VCMLA (by element)

Vector Complex Multiply Accumulate (by element).

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on complex numbers from the first source register and the destination register with the specified complex number from the second source register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

(FEAT_FCMA)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>S</td>
<td>D</td>
<td>rot</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

64-bit SIMD vector of half-precision floating-point variant

Applies when S == 0 & Q == 0.

VCMLA(<q>, .F16 <Dd>, <Dn>, <Dm>[<index>], #<rotate>

64-bit SIMD vector of single-precision floating-point variant

Applies when S == 1 & Q == 0.

VCMLA(<q>, .F32 <Dd>, <Dn>, <Dm>[0], #<rotate>

128-bit SIMD vector of half-precision floating-point variant

Applies when S == 0 & Q == 1.

VCMLA(<q>, .F16 <Qd>, <Qn>, <Qm>[<index>], #<rotate>

128-bit SIMD vector of single-precision floating-point variant

Applies when S == 1 & Q == 1.

VCMLA(<q>, .F32 <Qd>, <Qn>, <Qm>[0], #<rotate>
**Decode for all variants of this encoding**

if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn);
m = if S=='1' then UInt(M:Vm) else UInt(Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
index = if S=='1' then 0 else UInt(M);

**T1**

(FEAT_FCMA)

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 0</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 0 S</td>
<td>D</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector of half-precision floating-point variant**

Applies when S == 0 && Q == 0.

VCMLA{<q>}.F16 <Dd>, <Dn>, <Dm>[<index>], #<rotate>

**64-bit SIMD vector of single-precision floating-point variant**

Applies when S == 1 && Q == 0.

VCMLA{<q>}.F32 <Dd>, <Dn>, <Dm>[0], #<rotate>

**128-bit SIMD vector of half-precision floating-point variant**

Applies when S == 0 && Q == 1.

VCMLA{<q>}.F16 <Qd>, <Qn>, <Dm>[<index>], #<rotate>

**128-bit SIMD vector of single-precision floating-point variant**

Applies when S == 1 && Q == 1.

VCMLA{<q>}.F32 <Qd>, <Qn>, <Dm>[0], #<rotate>

**Decode for all variants of this encoding**

if InITBlock() then UNPREDICTABLE;
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn);
m = if S=='1' then UInt(M:Vm) else UInt(Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
index = if S=='1' then 0 else UInt(M);

**Assembler symbols**

| <q> | See **Standard assembler syntax fields on page F1-7228**. |
| <Qd> | Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2. |
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> For the half-precision scalar variant: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.

For the single-precision scalar variant: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<index> Is the element index in the range 0 to 1, encoded in the "M" field.

<rotate> Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in the "rot" field. It can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Rotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>90</td>
</tr>
<tr>
<td>10</td>
<td>180</td>
</tr>
<tr>
<td>11</td>
<td>270</td>
</tr>
</tbody>
</table>

Operation for all encodings

EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1
  operand1 = D[n+r];
  operand2 = D[m];
  operand3 = D[d+r];
  for e = 0 to (elements DIV 2)-1
    bits(esize) element1;
    bits(esize) element2;
    bits(esize) element3;
    bits(esize) element4;
    case rot of
      when '00'
        element1 = Elem[operand2, index*2, esize];
        element2 = Elem[operand1, e*2+1, esize];
        element3 = Elem[operand2, index*2+1, esize];
        element4 = Elem[operand1, e*2, esize];
      when '01'
        element1 = FPNeg(Elem[operand2, index*2+1, esize]);
        element2 = Elem[operand1, e*2+1, esize];
        element3 = Elem[operand2, index*2, esize];
        element4 = Elem[operand1, e*2+1, esize];
      when '10'
        element1 = FPNeg(Elem[operand2, index*2, esize]);
        element2 = Elem[operand1, e*2, esize];
        element3 = FPNeg(Elem[operand2, index*2+1, esize]);
        element4 = Elem[operand1, e*2+1, esize];
      when '11'
        element1 = Elem[operand2, index*2+1, esize];
        element2 = Elem[operand1, e*2+1, esize];
        element3 = FPNeg(Elem[operand2, index*2, esize]);
        element4 = Elem[operand1, e*2, esize];
    end case
    result1 = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, StandardFPSCRValue());
    result2 = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, StandardFPSCRValue());
    Elem[D[d+r], e*2+1, esize] = result1;
    Elem[D[d+r], e*2+1, esize] = result2;
F6.1.54   VCMP

Vector Compare compares two floating-point registers, or one floating-point register and zero. It writes the result to the FPSCR flags. These are normally transferred to the PSTATE.{N, Z, C, V} Condition flags by a subsequent VMRS instruction.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>31 28 27 26 25 24 23 22 21 20 19 18 17 16 15</th>
<th>12 11 10 9 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=111 1 1 1 0 1 D 1 1 0 1 0 0</td>
<td>Vd 1 0 size 0 1 M 0 Vm</td>
</tr>
<tr>
<td>cond E</td>
<td></td>
</tr>
</tbody>
</table>

**Half-precision scalar variant**

Applies when size == 01.

VCMP{<c>}{<q>}.F16 <Sd>, <Sm>

**Single-precision scalar variant**

Applies when size == 10.

VCMP{<c>}{<q>}.F32 <Sd>, <Sm>

**Double-precision scalar variant**

Applies when size == 11.

VCMP{<c>}{<q>}.F64 <Dd>, <Dm>

**Decode for all variants of this encoding**

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quaint_nan_exc = (E == '1'); with_zero = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
A2

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=111</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Half-precision scalar variant
Applies when size == 01.
VCMP{<c>}{<q>}.F16 <Sd>, #0.0

Single-precision scalar variant
Applies when size == 10.
VCMP{<c>}{<q>}.F32 <Sd>, #0.0

Double-precision scalar variant
Applies when size == 11.
VCMP{<c>}{<q>}.F64 <Dd>, #0.0

Decode for all variants of this encoding
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = TRUE;
integer esize;
integer d;
case size of
when '01' esize = 16; d = UInt(Vd:D);
when '10' esize = 32; d = UInt(Vd:D);
when '11' esize = 64; d = UInt(D:Vd);

CONSTRAINED UNPREDICTABLE behavior
If size == '01' && cond != '1110', then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>0</td>
<td>1</td>
<td>M</td>
<td>0</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
VCMP{<c>}{<q>}.F32 <Sd>, <Sm>  

**Double-precision scalar variant**  
Applies when size == 11.

VCMP{<c>}{<q>}.F64 <Dd>, <Dm>

**Decode for all variants of this encoding**

```plaintext
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Half-precision scalar variant**

Applies when size == 01.

VCMP{<c>}{<q>}.F16 <Sd>, #0.0

**Single-precision scalar variant**

Applies when size == 10.

VCMP{<c>}{<q>}.F32 <Sd>, #0.0

**Double-precision scalar variant**

Applies when size == 11.

VCMP{<c>}{<q>}.F64 <Dd>, #0.0

**Decode for all variants of this encoding**

```plaintext
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D);
```
when '10' esize = 32; d = UInt(Vd:D);
when '11' esize = 64; d = UInt(D:Vd);

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

\(<c>\) See Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<Sd>\) Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

\(<Sm>\) Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

\(<Dd>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<Dm>\) Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    bits(4) nzcv;
    case esize of
        when 16
            bits(16) op16 = if with_zero then FPZero('0', 16) else S[m]<15:0>;
            nzcv = FPCompare(S[d]<15:0>, op16, quiet_nan_exc, FPSCR[]);
        when 32
            bits(32) op32 = if with_zero then FPZero('0', 32) else S[m];
            nzcv = FPCompare(S[d], op32, quiet_nan_exc, FPSCR[]);
        when 64
            bits(64) op64 = if with_zero then FPZero('0', 64) else D[m];
            nzcv = FPCompare(D[d], op64, quiet_nan_exc, FPSCR[]);
    FPSCR<31:28> = nzcv; // FPSCR.<N,Z,C,V> set to nzcv
```

**Operational information**

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the FPSCR condition flags to N=0, Z=0, C=1, and V=1.
F6.1.55  VCMPE

Vector Compare, raising Invalid Operation on NaN compares two floating-point registers, or one floating-point
register and zero. It writes the result to the FPSCR flags. These are normally transferred to the PSTATE.{N, Z, C,
V} Condition flags by a subsequent VMRS instruction.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of
NaN.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which
the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For
more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 28| 26 25 24|23 22 21 20|19 18 17 16|15 12|11 10 9 8| 7 6 5 4 | 3 0 |
|---|---|---|---|---|---|---|---|---|
| !=111 | 1 1 1 0 1 | D | 1 | 0 | 0 | 0 | Vd | 1 0 | size | 1 1 | M | 0 | Vm |

**Half-precision scalar variant**

Applies when size == 01.

VCMPE{<c>}{<q>}.F16 <Sd>, <Sm>

**Single-precision scalar variant**

Applies when size == 10.

VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>

**Double-precision scalar variant**

Applies when size == 11.

VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>

**Decode for all variants of this encoding**

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
A2

Half-precision scalar variant
Applies when size == 01.
VCMPE{<c>}{<q>}.F16 <Sd>, #0.0

Single-precision scalar variant
Applies when size == 10.
VCMPE{<c>}{<q>}.F32 <Sd>, #0.0

Double-precision scalar variant
Applies when size == 11.
VCMPE{<c>}{<q>}.F64 <Dd>, #0.0

Decode for all variants of this encoding
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = TRUE;
integer esize;
integer d;
case size of
  when '01' esize = 16; d = UInt(Vd:D);
  when '10' esize = 32; d = UInt(Vd:D);
  when '11' esize = 64; d = UInt(D:Vd);
CONSTRAINED UNPREDICTABLE behavior
If size == '01' && cond != '1110', then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1
VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>

**Double-precision scalar variant**
Applies when size == 11.

VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>

**Decode for all variants of this encoding**
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**
If size == '01' && InITBlock(), then one of the following behaviors must occur:
  \* The instruction is UNDEFINED.
  \* The instruction executes as if it passes the Condition code check.
  \* The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0</td>
<td>1 1 0 1</td>
<td>D 1 1 0</td>
<td>1 0 1</td>
<td>Vd 1 0</td>
<td>size 1</td>
<td>1 0 (0) 0</td>
<td>(0) (0) (0)</td>
</tr>
</tbody>
</table>

E

**Half-precision scalar variant**
Applies when size == 01.
VCMPE{<c>}{<q>}.F16 <Sd>, #0.0

**Single-precision scalar variant**
Applies when size == 10.
VCMPE{<c>}{<q>}.F32 <Sd>, #0.0

**Double-precision scalar variant**
Applies when size == 11.
VCMPE{<c>}{<q>}.F64 <Dd>, #0.0

**Decode for all variants of this encoding**
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = TRUE;
integer esize;
integer d;
case size of
  when '01' esize = 16; d = UInt(Vd:D);
when '10' esize = 32; d = UInt(Vd:D);
when '11' esize = 64; d = UInt(D:Vd);

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

<

See *Standard assembler syntax fields* on page F1-7228.

<

See *Standard assembler syntax fields* on page F1-7228.

<

Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<

Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
  bits(4) nzcv;
  case esize of
    when 16
      bits(16) op16 = if with_zero then FPZero('0', 16) else S[m]<15:0>;
      nzcv = FPCompare(S[d]<15:0>, op16, quiet_nan_exc, FPSCR[]);
    when 32
      bits(32) op32 = if with_zero then FPZero('0', 32) else S[m];
      nzcv = FPCompare(S[d], op32, quiet_nan_exc, FPSCR[]);
    when 64
      bits(64) op64 = if with_zero then FPZero('0', 64) else D[m];
      nzcv = FPCompare(D[d], op64, quiet_nan_exc, FPSCR[]);
  FPSCR<31:28> = nzcv; // FPSCR.<N,Z,C,V> set to nzcv

**Operational information**

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the FPSCR condition flags to N=0, Z=0, C=1, and V=1.
F6.1.56  VCNT

Vector Count Set Bits counts the number of bits that are one in each element in a vector, and places the results in a second vector.

The operand vector elements must be 8-bit fields.

The result vector elements are 8-bit integers.

Depending on settings in the CPACR, NSACR, and HCPR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VCNT{<c>}{<q>}.8 <Dd>, <Dm> // Encoded as Q = 0

128-bit SIMD vector variant

Applies when Q == 1.

VCNT{<c>}{<q>}.8 <Qd>, <Qm> // Encoded as Q = 1

Decode for all variants of this encoding

if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
e size = 8;  elements = 8;
d = Uint(D:Vd);  m = Uint(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when Q == 0.

VCNT{<c>}{<q>}.8 <Dd>, <Dm> // Encoded as Q = 0

128-bit SIMD vector variant

Applies when Q == 1.

VCNT{<c>}{<q>}.8 <Qd>, <Qm> // Encoded as Q = 1
Decode for all variants of this encoding

if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8;  elements = 8;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Assembler symbols

<q> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
   For encoding T1: see Standard assembler syntax fields on page F1-7228.

<Qd> See Standard assembler syntax fields on page F1-7228.

<Qm> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      Elem[D[r],e,esize] = BitCount(Elem[D[r],e,esize]<esize-1:0>);

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.57 VCVT (from single-precision to BFloat16, Advanced SIMD)

Vector Convert from single-precision to BFloat16 converts each 32-bit element in a vector from single-precision floating-point to BFloat16 format, and writes the result into a second vector. The result vector elements are half the width of the source vector elements.

Unlike the BFloat16 multiplication instructions, this instruction uses the Round to Nearest rounding mode, and can generate a floating-point exception that causes cumulative exception bits in the FPSCR to be set.

A1

(FEAT_AA32BF16)

```
| 31 30 29 28| 27 26 25 24| 23 22 21 20| 19 18 17 16| 15 | 12| 11 10 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|--------|--------|--------|--------|----|----|--------|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | Vd | 0 | 1 | 1 | 0 | 0 | 1 | M | 0 | Vm |
```

A1 variant

VCVT{<c>}{<q>}.BF16.F32 <Dd>, <Qm>

**Decode for this encoding**

- if !HaveAArch32BF16Ext() then UNDEFINED;
- if Vm<0> == '1' then UNDEFINED;
- integer d = UInt(D:Vd);
- integer m = UInt(M:Vm);

T1

(FEAT_AA32BF16)

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>1</th>
<th>0</th>
<th>15</th>
<th>12</th>
<th>11 10 9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Vd</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

T1 variant

VCVT{<c>}{<q>}.BF16.F32 <Dd>, <Qm>

**Decode for this encoding**

- if !HaveAArch32BF16Ext() then UNDEFINED;
- if Vm<0> == '1' then UNDEFINED;
- integer d = UInt(D:Vd);
- integer m = UInt(M:Vm);

**Assembler symbols**

- `<c>` For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional. For encoding T1: see Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.`
Operation for all encodings

bits(128) operand;
bits(64) result;

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

operand = Q[m>>1];
for e = 0 to 3
    bits(32) op = Elem[operand, e, 32];
    Elem[result, e, 16] = FPConvertBF(op, StandardFPSCRValue());
D[d] = result;
F6.1.58   VCVT (between double-precision and single-precision)

Convert between double-precision and single-precision does one of the following:

- Converts the value in a double-precision register to single-precision and writes the result to a single-precision register.
- Converts the value in a single-precision register to double-precision and writes the result to a double-precision register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

Single-precision to double-precision variant

Applies when size == 10.

VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>

Double-precision to single-precision variant

Applies when size == 11.

VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>

Decode for all variants of this encoding

double_to_single = (size == '11');
d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);
m = if double_to_single then UInt(M:Vm) else UInt(Vm:M);

T1

Single-precision to double-precision variant

Applies when size == 10.

VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>

Double-precision to single-precision variant

Applies when size == 11.

VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>
Decode for all variants of this encoding

double_to_single = (size == '11');
d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);
m = if double_to_single then UInt(M:Vm) else UInt(Vm:M);

Assembler symbols

<e> See Standard assembler syntax fields on page F1-7228.
<q> See Standard assembler syntax fields on page F1-7228.
<5d> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<6m> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
<6d> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<5m> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    if double_to_single then
        S[d] = FPConvert(D[m], FPSCR[], 32);
    else
        D[d] = FPConvert(S[m], FPSCR[], 64);
F6.1.59  VCVT (between half-precision and single-precision, Advanced SIMD)

Vector Convert between half-precision and single-precision converts each element in a vector from single-precision to half-precision floating-point, or from half-precision to single-precision, and places the results in a second vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 12|11 10 9 |8 7 6 5 |4 3 |0 |
|-------------|-------------|-------------|-------------|-------------|
| 1 1 1 1 0 0 1 1 |1 |size 1 0 |Vd 0 1 1 |op 0 0 |M 0 |Vm |

Half-precision to single-precision variant

Applies when \( \text{op} = 1 \).

\[ \text{VCVT}\{\text{<c>}}\{\text{<q>}}\}.F32.F16 <\text{Qd}>, <\text{Dm}> \] // Encoded as \( \text{op} = 1 \)

Single-precision to half-precision variant

Applies when \( \text{op} = 0 \).

\[ \text{VCVT}\{\text{<c>}}\{\text{<q>}}\}.F16.F32 <\text{Dd}>, <\text{Qm}> \] // Encoded as \( \text{op} = 0 \)

Decode for all variants of this encoding

\[
\text{if size} != '01' \text{ then UNDEFINED}; \\
\text{half_to_single} = (\text{op} == '1'); \\
\text{if half_to_single} \&\& \text{Vd}<0> == '1' \text{ then UNDEFINED}; \\
\text{if !half_to_single} \&\& \text{Vm}<0> == '1' \text{ then UNDEFINED}; \\
\text{esize} = 16; \text{ elements} = 4; \\
\text{m} = \text{UInt(M:Vm)}; \text{ d} = \text{UInt(D:Vd)};
\]

T1

| 15 14 13 12|11 10 9 |8 7 6 5 |4 3 2 1 0|15 12|11 10 9 |8 7 6 5 |4 3 |0 |
|-------------|-------------|-------------|-------------|-------------|
| 1 1 1 1 1 1 1 1 |1 |size 1 0 |Vd 0 1 1 |op 0 0 |M 0 |Vm |

Half-precision to single-precision variant

Applies when \( \text{op} = 1 \).

\[ \text{VCVT}\{\text{<c>}}\{\text{<q>}}\}.F32.F16 <\text{Qd}>, <\text{Dm}> \] // Encoded as \( \text{op} = 1 \)

Single-precision to half-precision variant

Applies when \( \text{op} = 0 \).

\[ \text{VCVT}\{\text{<c>}}\{\text{<q>}}\}.F16.F32 <\text{Dd}>, <\text{Qm}> \] // Encoded as \( \text{op} = 0 \)

Decode for all variants of this encoding

\[
\text{if size} != '01' \text{ then UNDEFINED}; \\
\text{half_to_single} = (\text{op} == '1'); \\
\text{if half_to_single} \&\& \text{Vd}<0> == '1' \text{ then UNDEFINED}; \\
\text{if !half_to_single} \&\& \text{Vm}<0> == '1' \text{ then UNDEFINED};
\]
esize = 16;  elements = 4;
m = UInt(M:Vm);  d = UInt(D:Vd);

**Assembler symbols**

<q> For encoding A1: see *Standard assembler syntax fields* on page F1-7228. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F1-7228.

<q> See *Standard assembler syntax fields* on page F1-7228.

<qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <qd>*2.

<dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <qm>*2.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if half_to_single then
            Elem[q[d>>1],e,32] = FPConvert(Elem[din[m],e,16], StandardFPSCRValue(), 32);
        else
            Elem[d,d],e,16] = FPConvert(Elem[qin[m>>1],e,32], StandardFPSCRValue(), 16);
F6.1.60 VCVT (between floating-point and integer, Advanced SIMD)

Vector Convert between floating-point and integer converts each element in a vector from floating-point to integer, or from integer to floating-point, and places the results in a second vector.

The vector elements are the same type, and are floating-point numbers or integers. Signed and unsigned integers are distinct.

The floating-point to integer operation uses the Round towards Zero rounding mode. The integer to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 | 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | D | size | Vd | 0 1 1 | op | Q | M | 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
to_integer = (op<1> == '1'); unsigned = (op<0> == '1');
to_integer = (op<1> == '1'); unsigned = (op<0> == '1');

integer esize;
integer elements;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 12 11 10 9 8 7 6 5 4 3 0 | D | size | Vd | 0 1 1 | op | Q | M | 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.
VCVT{<c>}{<q>}{<dt1>}{<dt2>}{<Qd>}{<Qm>}

**Decode for all variants of this encoding**

```c
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
to_integer = (op<1> == '1'); unsigned = (op<0> == '1');
integer esize;
integer elements;
case size of
when '01' esize = 16; elements = 4;
when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

**CONSTRAINED UNPREDICTABLE behavior**

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

- `<c>` For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.
  - For encoding T1: see *Standard assembler syntax fields on page F1-7228*.

- `<q>` See *Standard assembler syntax fields on page F1-7228*.

- `<dt1>` Is the data type for the elements of the destination vector, encoded in the "size:op" field. It can have the following values:
  - F16 when size = 01, op = 0x
  - S16 when size = 01, op = 10
  - U16 when size = 01, op = 11
  - F32 when size = 10, op = 0x
  - S32 when size = 10, op = 10
  - U32 when size = 10, op = 11

- `<dt2>` Is the data type for the elements of the source vector, encoded in the "size:op" field. It can have the following values:
  - S16 when size = 01, op = 00
  - U16 when size = 01, op = 01
  - F16 when size = 01, op = 1x
  - S32 when size = 10, op = 00
  - U32 when size = 10, op = 01
  - F32 when size = 10, op = 1x

- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.`

- `<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.`

- `<Qd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- `<Qm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(esize) result;
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[m+r],e,esize];
            if to_integer then
                result = FPToFixed(op1, 0, unsigned, StandardFPSCRValue(), FPRounding_ZERO, esize);
            else
                result = FixedToFP(op1, 0, unsigned, StandardFPSCRValue(), FPRounding_TIEEVEN, esize);
            Elem[D[d+r],e,esize] = result;
F6.1.61  VCVT (floating-point to integer, floating-point)

Convert floating-point to integer with Round towards Zero converts a value in a register from floating-point to a 32-bit integer, using the Round towards Zero rounding mode, and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the CPACR, NSACR, HCPtr, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

Half-precision scalar variant

Applies when opc2 == 100 && size == 01.

VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>

Half-precision scalar variant

Applies when opc2 == 101 && size == 01.

VCVT{<c>}{<q>}.S32.F16 <Sd>, <Sm>

Single-precision scalar variant

Applies when opc2 == 100 && size == 10.

VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>

Single-precision scalar variant

Applies when opc2 == 101 && size == 10.

VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>

Double-precision scalar variant

Applies when opc2 == 100 && size == 11.

VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>

Double-precision scalar variant

Applies when opc2 == 101 && size == 11.

VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>

Decode for all variants of this encoding

if opc2 != '000' && !(opc2 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
integer d;
integer esize;
integer m;
boolean unsigned;
FPRounding rounding;
to_integer = (opc2<2> == '1');
if to_integer then
  unsigned = (opc2<0> == '0');
  rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
  d = UInt(Vd:D);
  case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);
else
  unsigned = (op == '0');
  rounding = FPRoundingMode(FPSCR[]);
  m = UInt(Vm:M);
  case size of
  when '01' esize = 16; d = UInt(Vd:D);
  when '10' esize = 32; d = UInt(Vd:D);
  when '11' esize = 64; d = UInt(D:Vd);
endcase

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 0</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  1  1  0  1  1  0  1</td>
<td>D  1  1  1  1  0  x</td>
<td>Vd</td>
<td>1  0</td>
<td>size</td>
<td>1  1</td>
<td>M  0</td>
<td>Vm</td>
</tr>
</tbody>
</table>

opc2 | op

Half-precision scalar variant

Applies when opc2 == 100 && size == 01.

VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>

Half-precision scalar variant

Applies when opc2 == 101 && size == 01.

VCVT{<c>}{<q>}.S32.F16 <Sd>, <Sm>

Single-precision scalar variant

Applies when opc2 == 100 && size == 10.

VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>

Single-precision scalar variant

Applies when opc2 == 101 && size == 10.

VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>

Double-precision scalar variant

Applies when opc2 == 100 && size == 11.

VCVT{<c>}{<q>}.U32.F64 <Sd>, <Sm>
**Double-precision scalar variant**

Applies when opc2 == 101 && size == 11.

`VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>`

**Decode for all variants of this encoding**

```plaintext
if opc2 != '000' && !(opc2 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
integer esize;
integer m;
integer d;
boolean unsigned;
FPRounding rounding;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
    else
        unsigned = (op == '0');
        rounding = FPRoundingMode(FPSCR[]);
        m = UInt(Vm:M);
        case size of
            when '01' esize = 16; d = UInt(Vd:D);
            when '10' esize = 32; d = UInt(Vd:D);
            when '11' esize = 64; d = UInt(D:Vd);
```

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Notes for all encodings**

Related encodings: See *Floating-point data-processing* on page F3-7329 for the T32 instruction set, or *Floating-point data-processing* on page F4-7415 for the A32 instruction set.

**Assembler symbols**

- `<c>` See *Standard assembler syntax fields* on page F1-7228.
- `<q>` See *Standard assembler syntax fields* on page F1-7228.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
  if to_integer then
    case esize of
      when 16
        S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding, 32);
      when 32
        S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding, 32);
      when 64
        S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding, 32);
    else
      case esize of
        when 16
          bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 16);
          S[d] = Zeros(16):fp16;
        when 32
          S[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 32);
        when 64
          D[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 64);
  else
F6.1.62 VCVT (integer to floating-point, floating-point)

Convert integer to floating-point converts a 32-bit integer to floating-point using the rounding mode specified by the FPSCR, and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the CPACR, NSACR, HCPtr, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

```
| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 1  | 1  | 1  | 0  | 1  | D  | 1  | 1  | 0  | 0  | Vd | 1  | 0  | size | op  | 1  | M | 0  | Vm |
```

**Half-precision scalar variant**

Applies when size == 01.

VCVT{<c>}{<q>}.F16.<dt> <$d>, <$m>

**Single-precision scalar variant**

Applies when size == 10.

VCVT{<c>}{<q>}.F32.<dt> <$d>, <$m>

**Double-precision scalar variant**

Applies when size == 11.

VCVT{<c>}{<q>}.F64.<dt> <$d>, <$m>

**Decode for all variants of this encoding**

```
if opc2 != '000' && (opc2 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
integer d;
integer esize;
integer m;
boolean unsigned;
FPRounding rounding;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
    else
        unsigned = (op == '0');
        rounding = FPRoundingMode(FPSCR[]);
        m = UInt(Vm:M);
        case size of
            when '01' esize = 16; d = UInt(Vd:D);
            when '10' esize = 32; d = UInt(Vd:D);
            when '11' esize = 64; d = UInt(D:Vd);
```
**CONSTRAINED UNPREDICTABLE behavior**

If \( \text{size} == '01' \&\& \text{cond} != '1110' \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**T1**

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9  8 7 6 5 4 3 2 0 15</th>
<th>11 10 9 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Vd</td>
</tr>
<tr>
<td>opc2</td>
<td>size</td>
<td>op</td>
</tr>
</tbody>
</table>
```

**Half-precision scalar variant**

Applies when \( \text{size} == 01 \).

\[
\text{VCVT}\{<c>\}{<q>}.F16.<dt> <Sd>, <Sm>
\]

**Single-precision scalar variant**

Applies when \( \text{size} == 10 \).

\[
\text{VCVT}\{<c>\}{<q>}.F32.<dt> <Sd>, <Sm>
\]

**Double-precision scalar variant**

Applies when \( \text{size} == 11 \).

\[
\text{VCVT}\{<c>\}{<q>}.F64.<dt> <Dd>, <Sm>
\]

**Decode for all variants of this encoding**

if \( \text{opc2} == '000' \&\& (\text{opc2} \in \{10x\}) \) then SEE "Related encodings";
if \( \text{size} == '00' || (\text{size} == '01' \&\& \text{HaveFP16Ext()}) \) then **UNDEFINED**;
if \( \text{size} == '01' \&\& \text{InITBlock()} \) then **UNPREDICTABLE**;
integer esize;
integer m;
integer d;
boolean unsigned;
FPRounding rounding;
to_integer = (opc2<2> == '1');
if to_integer then
  unsigned = (opc2<6> == '0');
  rounding = if \( \text{op} == '1' \) then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
  d = UInt(Vd:D);
  case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);
  else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR[]);
    m = UInt(Vm:M);
  case size of
  when '01' esize = 16; d = UInt(Vd:D);
  when '10' esize = 32; d = UInt(Vd:D);
  when '11' esize = 64; d = UInt(D:Vd);
CONSTRANDED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See Floating-point data-processing on page F3-7329 for the T32 instruction set, or Floating-point data-processing on page F4-7415 for the A32 instruction set.

Assembler symbols

<< See Standard assembler syntax fields on page F1-7228.
<q> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the operand, encoded in the "op" field. It can have the following values:
  U32 when op = 0
  S32 when op = 1
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<DD> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
if to_integer then
  case esize of
    when 16
      S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding, 32);
    when 32
      S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding, 32);
    when 64
      S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding, 32);
  else
    case esize of
      when 16
        bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 16);
        S[d] = Zeros(16):fp16;
      when 32
        S[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 32);
      when 64
        D[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 64);
F6.1.63 VCVT (between floating-point and fixed-point, Advanced SIMD)

Vector Convert between floating-point and fixed-point converts each element in a vector from floating-point to fixed-point, or from fixed-point to floating-point, and places the results in a second vector.

The vector elements are the same type, and are floating-point numbers or integers. Signed and unsigned integers are distinct.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

1111001

64-bit SIMD vector variant

Applies when imm6 != 000xxx & Q == 0.

VCVT{<c>}{<q>}{<dt1>}{<dt2>} <Dd>, <Dm>, #<fbits>

128-bit SIMD vector variant

Applies when imm6 != 000xxx & Q == 1.

VCVT{<c>}{<q>}{<dt1>}{<dt2>} <Qd>, <Qm>, #<fbits>

Decode for all variants of this encoding

if imm6 IN {000xxx} then SEE "Related encodings";
if op<1> == '0' & !HaveFP16Ext() then UNDEFINED;
if op<1> == '0' & imm6 IN {10xxxx} then UNDEFINED;
if imm6 IN {0xxxxx} then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
to_fixed = (op<0> == '1');  frac_bits = 64 - UInt(imm6);
unsigned = (U == '1');
integer esize;
integer elements;
case op<1> of
  when '0' esize = 16; elements = 4;
  when '1' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when imm6 != 000xxx & Q == 0.

VCVT{<c>}{<q>}{<dt1>}{<dt2>} <Dd>, <Dm>, #<fbits>
128-bit SIMD vector variant

Applies when imm6 != 000xxx && Q == 1.

VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>, #<fbits>

Decode for all variants of this encoding

if imm6 IN {'000xxx'} then SEE "Related encodings";
if op<1> == '0' && !HaveFP16Ext() then UNDEFINED;
if op<1> == '0' && imm6 IN {'10xxxx'} then UNDEFINED;
if imm6 IN {'0xxxxx'} then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

to_fixed = (op<0> == '1'); frac_bits = 64 - UInt(imm6);
unsigned = (U == '1');
integer esize;
integer elements;
case op<1> of
  when '0' esize = 16; elements = 4;
  when '1' esize = 32; elements = 2;
  d = UInt(D:Vd); m = UInt(M:Vm);
  regs = if Q == '0' then 1 else 2;

Notes for all encodings

Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<dt1> Is the data type for the elements of the destination vector, encoded in the "op:U" field. It can have the following values:

F16 when op = 00, U = x
S16 when op = 01, U = 0
U16 when op = 01, U = 1
F32 when op = 10, U = x
S32 when op = 11, U = 0
U32 when op = 11, U = 1

<dt2> Is the data type for the elements of the source vector, encoded in the "op:U" field. It can have the following values:

S16 when op = 00, U = 0
U16 when op = 00, U = 1
F16 when op = 01, U = x
S32 when op = 10, U = 0
U32 when op = 10, U = 1
F32 when op = 11, U = x

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<fbits> The number of fraction bits in the fixed point number, in the range 1 to 32 for 32-bit elements, or in the range 1 to 16 for 16-bit elements:

- \((64 - \text{<fbits>})\) is encoded in imm6.

An assembler can permit an <fbits> value of 0. This is encoded as floating-point to integer or integer to floating-point instruction, see VCVT (between floating-point and integer, Advanced SIMD).

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    bits(esize) result;
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[m-r],e,esize];
            if to_fixed then
                result = FPtoFixed(op1, frac_bits, unsigned, StandardFPSCRValue(),
                    FPRounding_ZERO, esize);
            else
                result = FixedToFP(op1, frac_bits, unsigned, StandardFPSCRValue(),
                    FPRounding_TIEEVEN, esize);
            Elem[D[d+r],e,esize] = result;
```

F6.1.64   VCVT (between floating-point and fixed-point, floating-point)

Convert between floating-point and fixed-point converts a value in a register from floating-point to fixed-point, or from fixed-point to floating-point. Software can specify the fixed-point value as either signed or unsigned.

The fixed-point value can be 16-bit or 32-bit. Conversions from fixed-point values take their operand from the low-order bits of the source register and ignore any remaining bits. Signed conversions to fixed-point values sign-extend the result value to the destination register width. Unsigned conversions to fixed-point values zero-extend the result value to the destination register width.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the CPACR, NSACR, HCPRTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

\[
\begin{array}{c}
\begin{array}{cccccccccccccccc}
\end{array}
\end{array}
\]

**Half-precision scalar variant**

Applies when \( \text{op} = 0 \) \&\& \( \text{sf} = 01 \).

\[
\text{VCVT}\{<c>\}{<q>}.F16.<dt> <Sdm>, <Sdm>, #<fbits>
\]

**Half-precision scalar variant**

Applies when \( \text{op} = 1 \) \&\& \( \text{sf} = 01 \).

\[
\text{VCVT}\{<c>\}{<q>}.<dt>.F16 <Sdm>, <Sdm>, #<fbits>
\]

**Single-precision scalar variant**

Applies when \( \text{op} = 0 \) \&\& \( \text{sf} = 10 \).

\[
\text{VCVT}\{<c>\}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>
\]

**Single-precision scalar variant**

Applies when \( \text{op} = 1 \) \&\& \( \text{sf} = 10 \).

\[
\text{VCVT}\{<c>\}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>
\]

**Double-precision scalar variant**

Applies when \( \text{op} = 0 \) \&\& \( \text{sf} = 11 \).

\[
\text{VCVT}\{<c>\}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>
\]

**Double-precision scalar variant**

Applies when \( \text{op} = 1 \) \&\& \( \text{sf} = 11 \).

\[
\text{VCVT}\{<c>\}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>
\]

**Decode for all variants of this encoding**

\[
\text{if } \text{sf} = '00' \text{ || (sf = '01' \&\& !HaveFP16Ext()) then UNDEFINED;}
\text{if } \text{sf} = '01' \&\& \text{cond != '1110' then UNPREDICTABLE;}
\text{to_fixed = (op == '1'); unsigned = (U == '1');}
\]
size = if sx == '0' then 16 else 32;
frac_bits = size - UInt(imm4:i);
integer fp_size;
integer d;
case sf of
  when '01' fp_size = 16; d = UInt(Vd:D);
  when '10' fp_size = 32; d = UInt(Vd:D);
  when '11' fp_size = 64; d = UInt(D:Vd);
if frac_bits < 0 then UNPREDICTABLE;

**CONSTRANDED UNPREDICTABLE behavior**

If frac_bits < 0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

**T1**

```
|15 14 13 12|11 10 9 8 |7 6 5 4 |3 2 1 0|15|12|11 10 9 8 |7 6 5 4 |3 0 |
|1 1 1 0 |1 1 0 1|D 1 1 1|op 1 |U| Vd |1 0 |sf| sx |1 |0 |imm4|
```

**Half-precision scalar variant**

Applies when op == 0 && sf == 01.

VCVT{<c>}{<q>}.F16.<dt> <Sdm>, <Sdm>, #<fbits>

**Half-precision scalar variant**

Applies when op == 1 && sf == 01.

VCVT{<c>}{<q>}.<dt>.F16 <Sdm>, <Sdm>, #<fbits>

**Single-precision scalar variant**

Applies when op == 0 && sf == 10.

VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>

**Single-precision scalar variant**

Applies when op == 1 && sf == 10.

VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>

**Double-precision scalar variant**

Applies when op == 0 && sf == 11.

VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>

**Double-precision scalar variant**

Applies when op == 1 && sf == 11.

VCVT{<c>}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>
Decode for all variants of this encoding

\[
\begin{align*}
\text{if } sf &= '00' \text{ || } (sf == '01' \text{ && } \text{HaveFP16Ext()} \text{ then UNDEFINED;} \\
\text{if } sf &= '01' \text{ && } \text{InITBlock()} \text{ then UNPREDICTABLE;}
\end{align*}
\]

\[
\begin{align*}
to\_fixed &= (op == '1'); \text{ unsigned } = (U == '1'); \\
size &= \text{if } sx == '0' \text{ then 16 else 32;}
\end{align*}
\]

\[
\begin{align*}
\text{frac\_bits} &= \text{size} - \text{UInt}(\text{imm4:i}); \\
\text{integer } fp\_size; \text{ integer } d; \\
\text{case } sf \text{ of}
\end{align*}
\]

\[
\begin{align*}
\text{when } '01' \text{ } fp\_size &= 16; d = \text{UInt}(\text{Vd:D}); \\
\text{when } '10' \text{ } fp\_size &= 32; d = \text{UInt}(\text{Vd:D}); \\
\text{when } '11' \text{ } fp\_size &= 64; d = \text{UInt}(\text{D:Vd}); \\
\end{align*}
\]

\[
\begin{align*}
\text{if } \text{frac\_bits} < 0 \text{ then UNPREDICTABLE;}
\end{align*}
\]

CONSTRAINED UNPREDICTABLE behavior

If \( \text{frac\_bits} < 0 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VCVT (between floating-point and fixed-point) on page K1-11577.

Assembler symbols

- <> See Standard assembler syntax fields on page F1-7228.
- <> See Standard assembler syntax fields on page F1-7228.
- <dt> Is the data type for the fixed-point number, encoded in the "U:sx" field. It can have the following values:
  - S16 when \( U = 0, sx = 0 \)
  - S32 when \( U = 0, sx = 1 \)
  - U16 when \( U = 1, sx = 0 \)
  - U32 when \( U = 1, sx = 1 \)
- <Sdm> Is the 32-bit name of the SIMD&FP destination and source register, encoded in the "Vd:D" field.
- <Ddm> Is the 64-bit name of the SIMD&FP destination and source register, encoded in the "D:Vd" field.
- <fbits> The number of fraction bits in the fixed-point number:
  - If <dt> is S16 or U16, <fbits> must be in the range 0-16. (16 - <fbits>) is encoded in [imm4, i]
  - If <dt> is S32 or U32, <fbits> must be in the range 1-32. (32 - <fbits>) is encoded in [imm4, i].

Operation for all encodings

\[
\begin{align*}
\text{if } \text{ConditionPassed()} \text{ then} \\
\text{EncodingSpecificOperations(); \ CheckVFPEnabled(TRUE);} \\
\text{if } \text{to\_fixed} \text{ then} \\
\text{bits(size) result;} \\
\text{case } fp\_size \text{ of} \\
\text{when 16}
\end{align*}
\]
result = FPToFixed(S[d]<15:0>, frac_bits, unsigned, FPSCR[], FPRounding_ZERO, size);
S[d] = Extend(result, 32, unsigned);

when 32
  result = FPToFixed(S[d], frac_bits, unsigned, FPSCR[], FPRounding_ZERO, size);
  S[d] = Extend(result, 32, unsigned);

when 64
  result = FPToFixed(D[d], frac_bits, unsigned, FPSCR[], FPRounding_ZERO, size);
  D[d] = Extend(result, 64, unsigned);

else
  case fp_size of
    when 16
      bits(16) fp16 = FixedToFP(S[d]<size-1:0>, frac_bits, unsigned, FPSCR[],
        FPRounding_TIEEVEN, 16);
      S[d] = Zeros(16):fp16;
    when 32
      S[d] = FixedToFP(S[d]<size-1:0>, frac_bits, unsigned, FPSCR[], FPRounding_TIEEVEN, 32);
    when 64
      D[d] = FixedToFP(D[d]<size-1:0>, frac_bits, unsigned, FPSCR[], FPRounding_TIEEVEN, 64);
F6.1.65 VCVTA (Advanced SIMD)

Vector Convert floating-point to integer with Round to Nearest with Ties to Away converts each element in a vector from floating-point to integer using the Round to Nearest with Ties to Away rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 | 12 | 11 10 9 8 | 7 6 5 4 | 3 | 0 |
| 1 1 1 1 0 0 1 1 1 | D | 1 | 1 | size | 1 | 1 | Vd | 0 | 0 | 0 | 0 | op | Q | M | 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VCVTA{<q>}.<dt>.<dt2> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VCVTA{<q>}.<dt>.<dt2> <Qd>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<<0> == '1' || Vm<<0> == '1') then UNDEFINED;
if {size == '01' && !HaveFP16Ext()} || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
iinteger esize;
iinteger elements;
case size of
when '01' esize = 16; elements = 4;
when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 | 15 | 12 | 11 10 9 8 | 7 6 5 4 | 3 | 0 |
| 1 1 1 1 1 1 1 1 | D | 1 | 1 | size | 1 | 1 | Vd | 0 | 0 | 0 | 0 | op | Q | M | 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VCVTA{<q>}.<dt>.<dt2> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.
VCVTA{<q>}.<dt>.<dt2> <Qd>, <Qm>

**Decode for all variants of this encoding**

if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
integer esize;
integer elements;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

**CONSTRAINED UNPREDICTABLE behavior**

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

- `<q>`
  See *Standard assembler syntax fields on page F1-7228*.

- `<dt>`
  Is the data type for the elements of the destination, encoded in the "op:size" field. It can have the following values:
  - S16 when op = 0, size = 01
  - S32 when op = 0, size = 10
  - U16 when op = 1, size = 01
  - U32 when op = 1, size = 10

- `<dt2>`
  Is the data type for the elements of the source vector, encoded in the "size" field. It can have the following values:
  - F16 when size = 01
  - F32 when size = 10

- `<Qd>`
  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.

- `<Qm>`
  Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.

- `<Dd>`
  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- `<Dm>`
  Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
  for e = 0 to elements-1
    Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned,
    StandardFPSCRValue(), rounding, esize);
F6.1.66 \textbf{VCVTA (floating-point)}

Convert floating-point to integer with Round to Nearest with Ties to Away converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest with Ties to Away rounding mode, and places the result in a second register.

Depending on settings in the \texttt{CPACR}, \texttt{NSACR}, \texttt{HCPTR}, and \texttt{FPEXC} registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see \textit{Enabling Advanced SIMD and floating-point support} on page G1-8996.

A1

\begin{verbatim}
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 10  9  8  7  6  5  4  3  0 |
|   1 1 1 1 1 1 1 0 1 |D| 1 1 1 1 0 0 | Vd | 1 0 | l=00 | op | 1 | M | 0 | Vm |
RM       size
\end{verbatim}

\textbf{Half-precision scalar variant}

Applies when size == 01.

\texttt{VCVTA\{<q>.<dt>.F16\ <Sd>, <Sm>}

\textbf{Single-precision scalar variant}

Applies when size == 10.

\texttt{VCVTA\{<q>.<dt>.F32\ <Sd>, <Sm>}

\textbf{Double-precision scalar variant}

Applies when size == 11.

\texttt{VCVTA\{<q>.<dt>.F64\ <Sd>, <Sm>}

\textbf{Decode for all variants of this encoding}

\begin{verbatim}
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; m = UInt(M:Vm);
\end{verbatim}

T1

\begin{verbatim}
| 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 | 15 12 11 10  9  8  7  6  5  4  3  0 |
|   1 1 1 1 1 1 1 0 1 |D| 1 1 1 1 0 0 | Vd | 1 0 | l=00 | op | 1 | M | 0 | Vm |
RM       size
\end{verbatim}

\textbf{Half-precision scalar variant}

Applies when size == 01.

\texttt{VCVTA\{<q>.<dt>.F16\ <Sd>, <Sm>}

\end{verbatim}
Single-precision scalar variant
Applies when size == 10.

VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>

Double-precision scalar variant
Applies when size == 11.

VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>

Decode for all variants of this encoding
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior
If InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:
  U32 when op = 0
  S32 when op = 1

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSR[], rounding, 32);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, FPSR[], rounding, 32);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, FPSR[], rounding, 32);
F6.1.67 VCVTB

Convert to or from a half-precision value in the bottom half of a single-precision register does one of the following:

- Converts the half-precision value in the bottom half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the half-precision value in the bottom half of a single-precision register to double-precision and writes the result to a double-precision register.
- Converts the single-precision value in a single-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.
- Converts the double-precision value in a double-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| [31] | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| l=1111 | 1 | 1 | 1 | 0 | 1 | D | 1 | 1 | 0 | 0 | 1 | op | Vd | 1 | 0 | 1 | sz | 0 | 1 | M | 0 | Vm |

**Half-precision to single-precision variant**

Applies when op == 0 && sz == 0.

VCVTB{<c>}{<q>}.F32.F16 <Sd>, <Sm>

**Half-precision to double-precision variant**

Applies when op == 0 && sz == 1.

VCVTB{<c>}{<q>}.F64.F16 <Dd>, <Sm>

**Single-precision to half-precision variant**

Applies when op == 1 && sz == 0.

VCVTB{<c>}{<q>}.F16.F32 <Dd>, <Sm>

**Double-precision to half-precision variant**

Applies when op == 1 && sz == 1.

VCVTB{<c>}{<q>}.F16.F64 <Dd>, <Dm>

**Decode for all variants of this encoding**

uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
integer d;
integer m;
if uses_double then
  if convert_from_half then
    d = UINT(D:Vd); m = UINT(Vm:M);
  else
    d = UINT(Vd:D); m = UINT(M:Vm);
else
  d = UINT(Vd:D); m = UINT(Vm:M);
T1

<table>
<thead>
<tr>
<th>op</th>
<th>sz</th>
<th>Vd</th>
<th>D</th>
<th>M</th>
<th>Vm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Half-precision to single-precision variant**

Applies when \( \text{op} == 0 \land \text{sz} == 0 \).

\[
\text{VCVTB}\{<c>\}{<q>}.F32.F16 <Sd>, <Sm>
\]

**Half-precision to double-precision variant**

Applies when \( \text{op} == 0 \land \text{sz} == 1 \).

\[
\text{VCVTB}\{<c>\}{<q>}.F64.F16 <Dd>, <Sm>
\]

**Single-precision to half-precision variant**

Applies when \( \text{op} == 1 \land \text{sz} == 0 \).

\[
\text{VCVTB}\{<c>\}{<q>}.F16.F32 <Sd>, <Sm>
\]

**Double-precision to half-precision variant**

Applies when \( \text{op} == 1 \land \text{sz} == 1 \).

\[
\text{VCVTB}\{<c>\}{<q>}.F16.F64 <Sd>, <Sm>
\]

**Decode for all variants of this encoding**

\[
\text{uses\_double} = (\text{sz} == '1') \land \text{convert\_from\_half} = (\text{op} == '0') ;
\text{lowbit} = (\text{if } \text{T} == '1' \text{ then } 16 \text{ else } 0) ;
\text{integer } d ;
\text{integer } m ;
\text{if } \text{uses\_double} \text{ then}
\text{if } \text{convert\_from\_half} \text{ then}
\text{d} = \text{UInt(D:Vd)} ; \text{m} = \text{UInt(Vm:M)} ;
\text{else}
\text{d} = \text{UInt(Vd:D)} ; \text{m} = \text{UInt(M:Vm)} ;
\text{else}
\text{d} = \text{UInt(Vd:D)} ; \text{m} = \text{UInt(Vm:M)} ;
\]

**Assembler symbols**

\(<c>\)  See *Standard assembler syntax fields on page F1-7228.*

\(<q>\)  See *Standard assembler syntax fields on page F1-7228.*

\(<Sd>\)  Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

\(<Dd>\)  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<Sm>\)  Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

\(<Dm>\)  Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

\[
\text{if } \text{ConditionPassed()} \text{ then}
\text{EncodingSpecificOperations(); CheckVFPEnabled(TRUE);} \text{ bits(16) } \text{hp};
\]
if convert_from_half then
  hp = S[m]<lowbit+15:lowbit>;
  if uses_double then
    D[d] = FPConvert(hp, FPSCR[], 64);
  else
    S[d] = FPConvert(hp, FPSCR[], 32);
  else
  if uses_double then
    hp = FPConvert(D[m], FPSCR[], 16);
  else
    hp = FPConvert(S[m], FPSCR[], 16);
  S[d]<lowbit+15:lowbit> = hp;
F6.1.68  VCVTB (BFLOAT16)

Converts the single-precision value in a single-precision register to BFLOAT16 format and writes the result into the bottom half of a single precision register, preserving the top 16 bits of the destination register.

Unlike the BFLOAT16 multiplication instructions, this instruction honors all the control bits in the FPSCR that apply to single-precision arithmetic, including the rounding mode. This instruction can generate a floating-point exception which causes a cumulative exception bit in the FPSCR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPSCR.

A1

(FEAT_AA32BF16)

```
!1111 1 1 1 0 1 D 1 1 0 0 1 1 Vd 1 0 0 1 0 1 M 0 Vm
cond
```

**A1 variant**

VCVTB{<c>}{<q>}.BF16.F32 <Sd>, <Sm>

**Decode for this encoding**

if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
integer m = UInt(Vm:M);

T1

(FEAT_AA32BF16)

```
1 1 1 0 1 1 1 0 1 D 1 1 0 0 1 1 Vd 1 0 0 1 0 1 M 0 Vm
```

**T1 variant**

VCVTB{<c>}{<q>}.BF16.F32 <Sd>, <Sm>

**Decode for this encoding**

if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
integer m = UInt(Vm:M);

**Assembler symbols**

- `<c>`  See Standard assembler syntax fields on page F1-7228.
- `<q>`  See Standard assembler syntax fields on page F1-7228.
- `<Sd>`  Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Sm>`  Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);

    S[d]<15:0> = FPConvertBF(S[m], FPSCR[]);
VCVTM (Advanced SIMD)

Vector Convert floating-point to integer with Round towards -Infinity converts each element in a vector from floating-point to integer using the Round towards -Infinity rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VCVTM{<q>}.<dt>.<dt2> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VCVTM{<q>}.<dt>.<dt2> <Qd>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if {size == '01' && !HaveFP16Ext()} || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
i *** integer esize;
i *** integer elements;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when Q == 0.

VCVTM{<q>}.<dt>.<dt2> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VCVTM{<q>}.<dt>.<dt2> <Qd>, <Qm>
Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' || HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
integer esize;
integer elements;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONstrained UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the destination, encoded in the "op:size" field. It can have the following values:
S16 when op = 0, size = 01
S32 when op = 0, size = 10
U16 when op = 1, size = 01
U32 when op = 1, size = 10
<dt2> Is the data type for the elements of the source vector, encoded in the "size" field. It can have the following values:
F16 when size = 01
F32 when size = 10
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
  for e = 0 to elements-1
    Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned,
    StandardFPSCRValue(), rounding, esize);
F6.1.70  **VCVTM (floating-point)**

Convert floating-point to integer with Round towards -Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards -Infinity rounding mode, and places the result in a second register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

A1

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 12|11 10 9 8|7 6 5 4|3 0 ]
  | 1 1 1 1 1 1 0 1 |D| 1 1 1 1 1 1 1 1 | Vd | 1 0 | !=00 | op | M | 0 | Vm |
  RM    size
```

**Half-precision scalar variant**

Applies when size == 01.

VCVTM{<q>}.<dt>.F16 <Sd>, <Sm>

**Single-precision scalar variant**

Applies when size == 10.

VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>

**Double-precision scalar variant**

Applies when size == 11.

VCVTM{<q>}.<dt>.F64 <Sd>, <Sm>

**Decode for all variants of this encoding**

if size == '00' || (size == '01' & & !HaveFP16Ext()) then UNDEFINED;

r = FPDecodeRM(RM);  unsigned = (op == '0');

d = UInt(Vd:D);

integer esize;

ter integer m;

case size of

when '01' esize = 16; m = UInt(Vm:M);

when '10' esize = 32; m = UInt(Vm:M);

when '11' esize = 64; m = UInt(M:Vm);

```
T1

[15 14 13|12|11 10 9 8|7 6 5 4|3 2 1 0|15 12|11 10 9 8|7 6 5 4|3 0 ]
  | 1 1 1 1 1 1 0 1 |D| 1 1 1 1 1 1 1 1 | Vd | 1 0 | !=00 | op | M | 0 | Vm |
  RM    size
```

**Half-precision scalar variant**

Applies when size == 01.

VCVTM{<q>}.<dt>.F16 <Sd>, <Sm>

**Single-precision scalar variant**

Applies when size == 10.
VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>

**Double-precision scalar variant**

Applies when size == 11.

VCVTM{<q>}.<dt>.F64 <Sd>, <Dm>

**Decode for all variants of this encoding**

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

<q>  See *Standard assembler syntax fields* on page F1-7228.
<dt> Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:
  U32   when op = 0
  S32   when op = 1
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding, 32);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding, 32);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding, 32);
F6.1.71 VCVTN (Advanced SIMD)

Vector Convert floating-point to integer with Round to Nearest converts each element in a vector from floating-point to integer using the Round to Nearest rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 12|11 10 9 8|7 6 5 4|3 0 | 1 1 1 1 0 0 1 1 | D 1 1 | size 1 1 | Vd 0 0 | Q M 0 | Vm |

RM

64-bit SIMD vector variant

Applies when Q == 0.

VCVTN{<q>}.<dt>.<dt2> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VCVTN{<q>}.<dt>.<dt2> <Qd>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
integer esize;
integer elements;
case size of
when '01' esize = 16; elements = 4;
when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

[15 14 13|12|11 10 9 8|7 6 5 4|3 2 1 0|15 12|11 10 9 8|7 6 5 4|3 0 | 1 1 1 1 1 1 1 1 | D 1 1 | size 1 1 | Vd 0 0 | Q M 0 | Vm |

RM

64-bit SIMD vector variant

Applies when Q == 0.

VCVTN{<q>}.<dt>,<dt2> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VCVTN{<q>}.<dt>,<dt2> <Qd>, <Qm>
Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
integer esize;
integer elements;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<op> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the destination, encoded in the "op:size" field. It can have the following values:
  S16 when op = 0, size = 01
  S32 when op = 0, size = 10
  U16 when op = 1, size = 01
  U32 when op = 1, size = 10
<dt2> Is the data type for the elements of the source vector, encoded in the "size" field. It can have the following values:
  F16 when size = 01
  F32 when size = 10
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bias(esize) result;
for r = 0 to regs-1
  for e = 0 to elements-1
    \text{Elem}[D[d+r],e,esize] = \text{FPToFixed}(\text{Elem}[D[m+r],e,esize], 0, unsigned, StandardFPSCRValue(), rounding, esize);
F6.1.72  VCVTN (floating-point)

Convert floating-point to integer with Round to Nearest converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest rounding mode, and places the result in a second register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 12|11 10 9 8|7 6 5 4 |3 0 |
|----------|----------|----------|----------|---|----------|---|---|---|
| 1 1 1 1 1 1 1 1 1 0 1 | D 1 1 1 0 1 | Vd 1 0 | !=0 | op 1 | M 0 | Vm |
| RM | size |

**Half-precision scalar variant**

Applies when size == 01.

VCVTN{<q>}.<dt>.F16 <Sd>, <Sm>

**Single-precision scalar variant**

Applies when size == 10.

VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>

**Double-precision scalar variant**

Applies when size == 11.

VCVTN{<q>}.<dt>.F64 <Sd>, <Sm>

**Decode for all variants of this encoding**

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
case size of
when '01' esize = 16; m = UInt(Vm:M);
when '10' esize = 32; m = UInt(Vm:M);
when '11' esize = 64; m = UInt(M:Vm);

T1

| 15 14 13 12|11 10 9 8|7 6 5 4 |3 2 1 0|15 12|11 10 9 8|7 6 5 4 |3 0 |
|----------|----------|----------|----------|---|----------|---|---|---|
| 1 1 1 1 1 1 1 1 1 0 1 | D 1 1 1 0 1 | Vd 1 0 | !=0 | op 1 | M 0 | Vm |
| RM | size |

**Half-precision scalar variant**

Applies when size == 01.

VCVTN{<q>}.<dt>.F16 <Sd>, <Sm>

**Single-precision scalar variant**

Applies when size == 10.
VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>

**Double-precision scalar variant**

Applies when size == 11.

VCVTN{<q>}.<dt>.F64 <Sd>, <Dm>

**Decode for all variants of this encoding**

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; m = UInt(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

<q> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:

U32 when op = 0
S32 when op = 1

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding, 32);
    when 32
        S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding, 32);
    when 64
        S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding, 32);
F6.1.73  VCVTP (Advanced SIMD)

Vector Convert floating-point to integer with Round towards +Infinity converts each element in a vector from floating-point to integer using the Round towards +Infinity rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1</td>
<td>D 1 1</td>
<td>size 1 1</td>
<td>Vd 0 0 1 0</td>
<td>op Q</td>
<td>M</td>
<td>0</td>
<td>Vm</td>
</tr>
</tbody>
</table>
```

64-bit SIMD vector variant

Applies when Q = 0.

VCVTP{<q>}.<dt>.<dt2> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q = 1.

VCVTP{<q>}.<dt>.<dt2> <Dd>, <Dm>

Decode for all variants of this encoding

- if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
- if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
- rounding = FPDecodeRM(RM); unsigned = (op == '1');
- integer esize;
- integer elements;
- case size of
  - when '01' esize = 16; elements = 4;
  - when '10' esize = 32; elements = 2;
- d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>D 1 1</td>
<td>size 1 1</td>
<td>Vd 0 0 1 0</td>
<td>op Q</td>
<td>M</td>
<td>0</td>
<td>Vm</td>
</tr>
</tbody>
</table>
```

64-bit SIMD vector variant

Applies when Q = 0.

VCVTP{<q>}.<dt>.<dt2> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q = 1.

VCVTP{<q>}.<dt>.<dt2> <Dd>, <Dm>
Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<> == '1' || Vm<> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
i  nteger esize;
integer elements;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the destination, encoded in the "op:size" field. It can have the following values:
  S16 when op = 0, size = 01
  S32 when op = 0, size = 10
  U16 when op = 1, size = 01
  U32 when op = 1, size = 10
<dt2> Is the data type for the elements of the source vector, encoded in the "size" field. It can have the following values:
  F16 when size = 01
  F32 when size = 10
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
  for e = 0 to elements-1
    Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned,
      StandardFPSCRValue(), rounding, esize);
F6.1.74  **VCVTP (floating-point)**

Convert floating-point to integer with Round towards +Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards +Infinity rounding mode, and places the result in a second register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

**A1**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 10 9 8 7 6 5 4 3 0 |
|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>

**Double-precision scalar variant**

Applies when size == 11.

VCVTP{<q>}.<dt>.F64 <Sd>, <Dm>

**Decode for all variants of this encoding**

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UINT(Vd:D);
integer esize;
integer m;
case size of
  when '01' esize = 16; m = UINT(Vm:M);
  when '10' esize = 32; m = UINT(Vm:M);
  when '11' esize = 64; m = UINT(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

<q> See *Standard assembler syntax fields* on page F1-7228.
<dt> Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:
  U32 when op = 0
  S32 when op = 1
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

EncodingSpecificOperations(); CheckVFPEnable(TRUE);
case esize of
  when 16
    S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding, 32);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding, 32);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding, 32);
F6.1.75 VCVTR

Convert floating-point to integer converts a value in a register from floating-point to a 32-bit integer, using the rounding mode specified by the FPSCR and places the result in a second register.

VCVTR (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>cond</th>
<th>opc2</th>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>1 1 1 0 1</td>
<td>D 1 1 1 1 x</td>
</tr>
</tbody>
</table>

Half-precision scalar variant

Applies when opc2 == 100 && size == 01.

VCVTR{<c>}{<q>}.U32.F16 <Sd>, <Sm>

Half-precision scalar variant

Applies when opc2 == 101 && size == 01.

VCVTR{<c>}{<q>}.S32.F16 <Sd>, <Sm>

Single-precision scalar variant

Applies when opc2 == 100 && size == 10.

VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>

Single-precision scalar variant

Applies when opc2 == 101 && size == 10.

VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>

Double-precision scalar variant

Applies when opc2 == 100 && size == 11.

VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>

Double-precision scalar variant

Applies when opc2 == 101 && size == 11.

VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>

Decode for all variants of this encoding

if opc2 != '000' && !(opc2 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
integer d;
integer esize;
integer m;
boolean unsigned;
FPRounding rounding;
to_integer = (opc2<2> == '1');
if to_integer then
  unsigned = (opc2<0> == '0');
  rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
  d = UInt(Vd:D);
  case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; m = UInt(M:Vm);
  else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR[]);
    m = UInt(Vm:M);
    case size of
    when '01' esize = 16; d = UInt(Vd:D);
    when '10' esize = 32; d = UInt(Vd:D);
    when '11' esize = 64; d = UInt(D:Vd);
  end;
end;

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is **undefined**.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**T1**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>Vd</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>0</td>
<td>1</td>
<td>M</td>
<td>0</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Half-precision scalar variant**
Applies when opc2 == 100 && size == 01.

VCVTR{<c>}{<q>}.U32.F16 <Sd>, <Sm>

**Half-precision scalar variant**
Applies when opc2 == 101 && size == 01.

VCVTR{<c>}{<q>}.S32.F16 <Sd>, <Sm>

**Single-precision scalar variant**
Applies when opc2 == 100 && size == 10.

VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>

**Single-precision scalar variant**
Applies when opc2 == 101 && size == 10.

VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>

**Double-precision scalar variant**
Applies when opc2 == 100 && size == 11.

VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>
**Double-precision scalar variant**

Applies when opc2 == 101 && size == 11.

VCVTR\(<c>\){<q>}.S32.F64 <Sd>, <Dm>

**Decode for all variants of this encoding**

if opc2 != '000' && !(opc2 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
integer esize;
integer m;
integer d;
boolean unsigned;
FPRounding rounding;
to_integer = (opc2<2> == '1');
if to_integer then
unsigned = (opc2<0> == '0');
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
d = UInt(Vd:D);
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);
else
unsigned = (op == '0');
rounding = FPRoundingMode(FPSCR[]);
m = UInt(Vm:M);
case size of
  when '01' esize = 16; d = UInt(Vd:D);
  when '10' esize = 32; d = UInt(Vd:D);
  when '11' esize = 64; d = UInt(D:Vd);

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Notes for all encodings**

Related encodings: See [Floating-point data-processing on page F3-7329](#) for the T32 instruction set, or [Floating-point data-processing on page F4-7415](#) for the A32 instruction set.

**Assembler symbols**

\(<c>\) See [Standard assembler syntax fields on page F1-7228](#).

\(<q>\) See [Standard assembler syntax fields on page F1-7228](#).

\(<Sd>\) Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

\(<Sm>\) Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

\(<Dm>\) Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    if to_integer then
        case esize of
            when 16
                S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding, 32);
            when 32
                S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding, 32);
            when 64
                S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding, 32);
        else
            case esize of
                when 16
                    bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 16);
                    S[d] = Zeros(16):fp16;
                when 32
                    S[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 32);
                when 64
                    D[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 64);
F6.1.76   VCVTT

Convert to or from a half-precision value in the top half of a single-precision register does one of the following:

• Converts the half-precision value in the top half of a single-precision register to single-precision and writes the result to a single-precision register.

• Converts the half-precision value in the top half of a single-precision register to double-precision and writes the result to a double-precision register.

• Converts the single-precision value in a single-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the destination register.

• Converts the double-precision value in a double-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

Half-precision to single-precision variant

Applies when op == 0 && sz == 0.

VCVTT{<c>}{<q>}.F32.F16 <Sd>, <Sm>

Half-precision to double-precision variant

Applies when op == 0 && sz == 1.

VCVTT{<c>}{<q>}.F64.F16 <Dd>, <Sm>

Single-precision to half-precision variant

Applies when op == 1 && sz == 0.

VCVTT{<c>}{<q>}.F16.F32 <Sd>, <Sm>

Double-precision to half-precision variant

Applies when op == 1 && sz == 1.

VCVTT{<c>}{<q>}.F16.F64 <Sd>, <Dm>

Decode for all variants of this encoding

uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (T == '1' then 16 else 0);
integer d;
integer m;
if uses_double then
  if convert_from_half then
d = UInt(D:Vd); m = UInt(Vm:M);
else
d = UInt(Vd:D); m = UInt(M:Vm);
else
d = UInt(Vd:D); m = UInt(Vm:M);
Half-precision to single-precision variant
Applies when \( \text{op} == 0 \) \&\& \( \text{sz} == 0 \).
\[ \text{VCVTT}\{<c>\}{<q>}.F32.F16 \ <Sd>, \ <Sm> \]

Half-precision to double-precision variant
Applies when \( \text{op} == 0 \) \&\& \( \text{sz} == 1 \).
\[ \text{VCVTT}\{<c>\}{<q>}.F64.F16 \ <Dd>, \ <Sm> \]

Single-precision to half-precision variant
Applies when \( \text{op} == 1 \) \&\& \( \text{sz} == 0 \).
\[ \text{VCVTT}\{<c>\}{<q>}.F16.F32 \ <Sd>, \ <Sm> \]

Double-precision to half-precision variant
Applies when \( \text{op} == 1 \) \&\& \( \text{sz} == 1 \).
\[ \text{VCVTT}\{<c>\}{<q>}.F16.F64 \ <Sd>, \ <Dm> \]

Decode for all variants of this encoding
\[
\text{uses\_double} = (\text{sz} == '1'); \quad \text{convert\_from\_half} = (\text{op} == '0'); \\
\text{lowbit} = (\text{if} \ T == '1' \ \text{then} \ 16 \ \text{else} \ 0); \\
\quad \text{integer} \ d; \\
\quad \text{integer} \ m; \\
\quad \text{if} \ \text{uses\_double} \ \text{then} \\
\quad \quad \text{if} \ \text{convert\_from\_half} \ \text{then} \\
\quad \quad \quad \ d = \text{UInt}('D:Vd'); \quad m = \text{UInt}('M:Vm'); \\
\quad \quad \quad \ \text{else} \\
\quad \quad \quad \quad d = \text{UInt}('Vd:D'); \quad m = \text{UInt}('M:Vm'); \\
\quad \quad \quad \ \text{else} \\
\quad \quad \quad \quad d = \text{UInt}('Vd:D'); \quad m = \text{UInt}('Vm:M');
\]

Assembler symbols
\(<c>\) See \textit{Standard assembler syntax fields} on page F1-7228.
\(<q>\) See \textit{Standard assembler syntax fields} on page F1-7228.

\(<Sd>\) Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

\(<Dm>\) Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

\(<Db>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<Sm>\) Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings
\[
\text{if} \ \text{ConditionPassed()} \ \text{then} \\
\quad \text{EncodingSpecificOperations(); CheckVFPEnabled(TRUE);} \\
\quad \text{bits(16)} \ \text{hp};
\]
if convert_from_half then
    hp = S[m]<lowbit+15:lowbit>;
if uses_double then
    D[d] = FPConvert(hp, FPSCR[], 64);
else
    S[d] = FPConvert(hp, FPSCR[], 32);
else
    if uses_double then
        hp = FPConvert(D[m], FPSCR[], 16);
    else
        hp = FPConvert(S[m], FPSCR[], 16);
    S[d]<lowbit+15:lowbit> = hp;
F6.1.77 VCVTT (BFloat16)

Converts the single-precision value in a single-precision register to BFloat16 format and writes the result in the top half of a single-precision register, preserving the bottom 16 bits of the register.

Unlike the BFloat16 multiplication instructions, this instruction honors all the control bits in the FPSCR that apply to single-precision arithmetic, including the rounding mode. This instruction can generate a floating-point exception which causes a cumulative exception bit in the FPSCR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPSCR.

A1

(FEAT_AAA32BF16)

```
|31| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 12| 11| 10| 9| 8| 7| 6| 5| 4| 3| 0|
| 1| 1| 1| 0| 1| D| 1| 1| 0| 0| 1| 1| Vd| 1| 0| 0| 1| 1| 1| M| 0| Vm|
```

A1 variant

VCVTT{<c>}{<q>}.BF16.F32 <Sd>, <Sm>

Decode for this encoding

```cpp
if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
integer m = UInt(Vm:M);
```

T1

(FEAT_AAA32BF16)

```
|31| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 12| 11| 10| 9| 8| 7| 6| 5| 4| 3| 0|
| 1| 1| 1| 0| 1| 1| 1| 0| 1| D| 1| 1| 0| 0| 1| 1| Vd| 1| 0| 0| 1| 1| 1| M| 0| Vm|
```

T1 variant

VCVTT{<c>}{<q>}.BF16.F32 <Sd>, <Sm>

Decode for this encoding

```cpp
if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
integer m = UInt(Vm:M);
```

Assembler symbols

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);

    S[d]<31:16> = FPConvertBF(S[m], FPSCR[]);
F6.1.78 VDIV

Divide divides one floating-point value by another floating-point value and writes the result to a third floating-point register.

Depending on settings in the CPACR, NSACR, HCPT, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

Half-precision scalar variant
Applies when size == 01.

VDIV{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>

Single-precision scalar variant
Applies when size == 10.

VDIV{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>

Double-precision scalar variant
Applies when size == 11.

VDIV{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>

Decode for all variants of this encoding

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
integer esize;
integer d;
integer n;
integer m;
case size of
when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONstrained UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

| l=1111 | 1 | 1 | 1 | 0 | 1 | D | 0 | 0 | \( V_n \) | \( V_d \) | 1 | 0 | size | \( N \) | 0 | \( M \) | 0 | \( V_m \) | \( \text{cond} \) |
**T1**

```
|15 14 13 12|11 10 9 8|7 6 5 4|3 |2 |1 |0 |1 1 1 0 1 1 0 1 |D |0 0 |Vn |Vd |1 0 |size |N |0 |M |0 |Vm |
```

### Half-precision scalar variant
Applies when \( \text{size} == 01 \).

\[
\text{VDIV}\{<c>\}{<q>}.F16 \{<Sd>,\} <Sn>, <Sm>
\]

### Single-precision scalar variant
Applies when \( \text{size} == 10 \).

\[
\text{VDIV}\{<c>\}{<q>}.F32 \{<Sd>,\} <Sn>, <Sm>
\]

### Double-precision scalar variant
Applies when \( \text{size} == 11 \).

\[
\text{VDIV}\{<c>\}{<q>}.F64 \{<Dd>,\} <Dn>, <Dm>
\]

### Decode for all variants of this encoding

```c
if size == '01' && \text{InITBlock}() then UNPREDICTABLE;
if \text{FPSCR.Len} != '000' || \text{FPSCR.Stride} != '00' then UNDEFINED;
if size == '00' || (size == '01' && !\text{HaveFP16Ext}()) then UNDEFINED;
integer esize;
integer d;
integer n;
integer m;
case size of
when '01' esize = 16; d = \text{UInt}(Vd:D); n = \text{UInt}(Vn:N); m = \text{UInt}(Vm:M);
when '10' esize = 32; d = \text{UInt}(Vd:D); n = \text{UInt}(Vn:N); m = \text{UInt}(Vm:M);
when '11' esize = 64; d = \text{UInt}(D:Vd); n = \text{UInt}(N:Vn); m = \text{UInt}(M:Vm);
```
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    case esize of
        when 16
            S[d] = Zeros(16) : FPDiv(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
        when 32
            S[d] = FPDiv(S[n], S[m], FPSCR[]);
        when 64
            D[d] = FPDiv(D[n], D[m], FPSCR[]);
```
F6.1.79   VDOT (vector)

BFLOAT16 floating-point (BF16) dot product (vector). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Within each pair, the elements in the first source vector are multiplied by the corresponding elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element in the destination vector which aligns with the pair of BF16 values in the first source vector. The instruction does not update the FPSCR exception status.

A1

(FEAT_AA32BF16)

64-bit SIMD vector variant

Applies when \( Q = 0 \).

VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when \( Q = 1 \).

VDOT{<q>}.BF16 <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

\[
\text{if} \ \text{!HaveAArch32BF16Ext()} \ \text{then UNDEFINED};
\]
\[
\text{if} \ Q = '1' \ \text{and} \ (\text{Vd}<0> = '1' \ \text{or} \ \text{Vn}<0> = '1' \ \text{or} \ \text{Vm}<0> = '1') \ \text{then UNDEFINED};
\]
\[
\text{integer} \ d = \text{UInt}(D;Vd);
\]
\[
\text{integer} \ n = \text{UInt}(N;Vn);
\]
\[
\text{integer} \ m = \text{UInt}(M;Vm);
\]
\[
\text{integer} \ \text{regs} = \text{if} \ Q = '1' \ \text{then} \ 2 \ \text{else} \ 1;
\]

T1

(FEAT_AA32BF16)

64-bit SIMD vector variant

Applies when \( Q = 0 \).

VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when \( Q = 1 \).

VDOT{<q>}.BF16 <Qd>, <Qn>, <Qm>
Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = if Q == '1' then 2 else 1;

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

bits(64) operand1;
bits(64) operand2;
bits(64) result;

CheckAdvSIMDEnabled();

for r = 0 to regs-1
operand1 = Din[n+r];
operand2 = Din[m+r];
result = Din[d+r];
for e = 0 to 1
  bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
  bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
  bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
  bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
  bits(32) sum = FPAdd_BF16(BFMulH(elt1_a, elt2_a), BFMulH(elt1_b, elt2_b));
  Elem[result, e, 32] = FPAdd_BF16(Elem[result, e, 32], sum);
D[d+r] = result;
F6.1.80 VDOT (by element)

BFloat16 floating-point indexed dot product (vector, by element). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Each pair of elements in the first source vector is multiplied by the indexed pair of elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element in the destination vector which aligns with the pair of BFloat16 values in the first source vector. The instruction does not update the FPSCR exception status.

A1

(FEAT_AA32BF16)

|31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8|7 6 5 4|3 0 |
|---|---|---|---|---|---|---|---|---|
|1 1 1 1|1 1 0|0 D|0 0|Vn|Vd|1 1 0|1 N Q M 0|Vm|

64-bit SIMD vector variant

Applies when Q == 0.

VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>{<index>}

128-bit SIMD vector variant

Applies when Q == 1.

VDOT{<q>}.BF16 <Qd>, <Qn>, <Dm>{<index>}

Decode for all variants of this encoding

if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;

T1

(FEAT_AA32BF16)

<table>
<thead>
<tr>
<th>15 14 13</th>
<th>12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
<td>1 1 0</td>
<td>0 D</td>
<td>0 0</td>
<td>Vn</td>
<td>Vd</td>
<td>1 1 0</td>
<td>1 N Q M 0</td>
<td>Vm</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>{<index>}

128-bit SIMD vector variant

Applies when Q == 1.

VDOT{<q>}.BF16 <Qd>, <Qn>, <Dm>{<index>}


Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.

<index> Is the element index in the range 0 to 1, encoded in the "M" field.

Operation for all encodings

bits(64) operand1;
bits(64) operand2;
bits(64) result;

CheckAdvSIMDEnabled();

operand2 = Din[m];
for r = 0 to regs-1
  operand1 = Din[n+r];
  result = Din[d+r];
  for e = 0 to 1
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * i + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * i + 1, 16];
    bits(32) sum = FPAdd_BF16(BFMulH(elt1_a, elt2_a), BFMulH(elt1_b, elt2_b));
    Elem[result, e, 32] = FPAdd_BF16(Elem[result, e, 32], sum);
  D[d+r] = result;
F6.1.81 VDUP (general-purpose register)

Duplicate general-purpose register to vector duplicates an element from a general-purpose register into every element of the destination vector.

The destination vector elements can be 8-bit, 16-bit, or 32-bit fields. The source element is the least significant 8, 16, or 32 bits of the general-purpose register. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------|----------------|--------------|--------------|--------------|--------------|--------------|--------------|
| t=1111         | 1 1 1 0 1 B Q 0 | Vd           | Rt           | 1 0 1 1 D 0 E 1 (0)(0)(0)(0)|

A1 variant

VDUP[<c>]<q>.<size> <Qd>, <Rt> // Encoded as Q = 1
VDUP[<c>]<q>.<size> <Dd>, <Rt> // Encoded as Q = 0

Decode for this encoding

if Q == '1' && Vd<6> == '1' then UNDEFINED;
d = UInt(D:Vd); t = UInt(Rt); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
case B:E of
  when '00' esize = 32; elements = 2;
  when '01' esize = 16; elements = 4;
  when '10' esize = 8;  elements = 8;
  when '11' UNDEFINED;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 0</th>
<th>15 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 1 1 0 1 B Q 0</td>
<td>Vd</td>
<td>Rt</td>
</tr>
<tr>
<td>1 0 1 1 D 0 E 1 (0)(0)(0)(0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T1 variant

VDUP[<c>]<q>.<size> <Qd>, <Rt> // Encoded as Q = 1
VDUP[<c>]<q>.<size> <Dd>, <Rt> // Encoded as Q = 0

Decode for this encoding

if Q == '1' && Vd<6> == '1' then UNDEFINED;
d = UInt(D:Vd); t = UInt(Rt); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
case B:E of
  when '00' esize = 32; elements = 2;
  when '01' esize = 16; elements = 4;
  when '10' esize = 8;  elements = 8;
  when '11' UNDEFINED;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<\(c\)\> See Standard assembler syntax fields on page F1-7228. Arm strongly recommends that any \texttt{VDP} instruction is unconditional, see Conditional execution on page F1-7229.

<\(q\)\> See Standard assembler syntax fields on page F1-7228.

<\(\text{size}\)\> The data size for the elements of the destination vector. It must be one of:

- 8 Encoded as \([b, e] = 0b10\).
- 16 Encoded as \([b, e] = 0b01\).
- 32 Encoded as \([b, e] = 0b00\).

<\(\text{Qd}\)\> The destination vector for a quadword operation.

<\(\text{Dd}\)\> The destination vector for a doubleword operation.

<\(\text{Rt}\)\> The Arm source register.

Operation for all encodings

\[
\text{if } \text{ConditionPassed}() \text{ then}
\]
\[
\text{EncodingSpecificOperations}(); \quad \text{CheckAdvSIMDEnabled}();
\]
\[
\text{scalar} = R[\text{t}]<\text{size-1:0}>;
\]
\[
\text{for } r = 0 \text{ to } \text{regs-1}
\]
\[
\text{for } e = 0 \text{ to } \text{elements-1}
\]
\[
\text{Elem}[D[d+r],e,\text{size}] = \text{scalar};
\]

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.82 VDUP (scalar)

Duplicate vector element to vector duplicates a single element of a vector into every element of the destination vector.

The scalar, and the destination vector elements, can be any one of 8-bit, 16-bit, or 32-bit fields. There is no distinction between data types.

For more information about scalars see Advanced SIMD scalars on page F1-7254.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

|31 30 29 28|27 26 25 24|23 22 21 20|19 18 17|16 15|12|11 10 9 8 7 6 5 4 3|0|
|---|---|---|---|---|---|---|---|---|---|
|1|1|1|0|0|1|1|D|1|1|imm4|Vd|1|1|0|0|Q|M|0|Vm|

Encoding

Applies when Q == 0.

VDUP{<c>}{<q>}.<size> <Dd>, <Dm[x]>

W3

|31 30 29 28|27 26 25 24|23 22 21 20|19 18 17|16 15|12|11 10 9 8 7 6 5 4 3|0|
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|1|1|1|1|1|1|1|D|1|1|imm4|Vd|1|1|0|0|Q|M|0|Vm|

Decode for all variants of this encoding

if imm4 IN {'x000'} then UNDEFINED;
if Q == '1' & Vd<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer index;
case imm4 of
  when 'xxx1' esize = 8; elements = 8; index = UInt(imm4<3:1>);
  when 'xx10' esize = 16; elements = 4; index = UInt(imm4<3:2>);
  when 'x100' esize = 32; elements = 2; index = UInt(imm4<3>);
d = UInt(D:Vd); m = UInt(M:Vm); regS = if Q == '0' then 1 else 2;

T1

|15 14 13 12|11 10 9 8 7 6 5 4 3|0|15 14 13 12|11 10 9 8 7 6 5 4 3|0|
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|1|1|1|1|1|1|1|1|1|D|1|1|imm4|Vd|1|1|0|0|Q|M|0|Vm|

Encoding

Applies when Q == 0.

VDUP{<c>}{<q>}.<size> <Dd>, <Dm[x]>

Encoding

Applies when Q == 1.
VDUP{<c>}{<q>}.<size> <Qd>, <Dm[x]>

**Decode for all variants of this encoding**

if imm4 IN {'x000'} then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
integer esize;
native elements;
native index;
case imm4 of
  when 'xxx1'  esize = 8;  elements = 8;  index = UInt(imm4<3:1>);
  when 'xx10'  esize = 16;  elements = 4;  index = UInt(imm4<3:2>);
  when 'x100'  esize = 32;  elements = 2;  index = UInt(imm4<3>);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

**Assembler symbols**

<q> For encoding A1: see [Standard assembler syntax fields on page F1-7228](#). This encoding must be unconditional.

For encoding T1: see [Standard assembler syntax fields on page F1-7228](#).

<q> See [Standard assembler syntax fields on page F1-7228](#).

<size> The data size. It must be one of:
- 8   Encoded as imm4<0> = '1'. imm4<3:1> encodes the index[x] of the scalar.
- 16  Encoded as imm4<1:0> = '10'. imm4<3:2> encodes the index [x] of the scalar.
- 32  Encoded as imm4<2:0> = '100'. imm4<3> encodes the index [x] of the scalar.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm[x]> The scalar. For details of how [x] is encoded, see the description of <size>.

**Operation for all encodings**

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  scalar = Elem[D[m],index,esize];
  for r = 0 to regs-1
    for e = 0 to elements-1
      Elem[D[d+r],e,esize] = scalar;

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.83   VEOR

Vector Bitwise Exclusive OR performs a bitwise Exclusive OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VEOR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VEOR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when Q == 0.

VEOR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VEOR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
Assembler symbols

<c> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<dt> An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    D[d+r] = D[n+r] EOR D[m+r];

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.84  VEXT (byte elements)

Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector.

The elements of the vectors are treated as being 8-bit fields. There is no distinction between data types.

The following figure shows the operation of VEXT doubleword operation for imm = 3.

Depending on settings in the CPACR, NSACR, and HCPR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

This instruction is used by the pseudo-instruction VEXT (multibyte elements). The pseudo-instruction is never the preferred disassembly.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VEXT{<c>}{<q>}.8 {<Dd>,} <Dn>, <Dm>, #<imm>

128-bit SIMD vector variant

Applies when Q == 1.

VEXT{<c>}{<q>}.8 {<Qd>,} <Qn>, <Qm>, #<imm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') || Vm<0> == '1') then UNDEFINED;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
quadword_operation = (Q == '1'); position = 8 * UInt(imm4);
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1

64-bit SIMD vector variant

Applies when Q == 0.
VEXT{<c>}{<q>}.8 {<Dd>,} <Dn>, <Dm>, #<imm>

128-bit SIMD vector variant

Applies when Q == 1.

VEXT{<c>}{<q>}.8 {<Qd>,} <Qn>, <Qm>, #<imm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
quadword_operation = (Q == '1');  position = 8 * UInt(imm4);
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);

Assembler symbols

<c> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<imm> For the 64-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to 7, encoded in the "imm4" field.
For the 128-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to 15, encoded in the "imm4" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
if quadword_operation then
    Q[d>>1] = (Q[m>>1]:Q[n>>1])<position+127:position>;
else
    D[d] = (D[m]:D[n])<position+63:position>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.85  VEXT (multibyte elements)

Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector.

This instruction is a pseudo-instruction of the VEXT (byte elements) instruction. This means that:

- The encodings in this description are named to match the encodings of VEXT (byte elements).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VEXT (byte elements) gives the operational pseudocode for this instruction.

**64-bit SIMD vector variant**

Applies when Q == 0.

\[
\text{VEXT}\{<c>\}{<q>}.<\text{size}>\{<Dd>,\} <Dn>, <Qm>, \#<\text{imm}>
\]

is equivalent to

\[
\text{VEXT}\{<c>\}{<q>}.8\{<Dd>,\} <Dn>, <Qm>, \#<\text{imm}*(size/8)>
\]

and is never the preferred disassembly.

**128-bit SIMD vector variant**

Applies when Q == 1.

\[
\text{VEXT}\{<c>\}{<q>}.<\text{size}>\{<Qd>,\} <Qn>, <Qm>, \#<\text{imm}>
\]

is equivalent to

\[
\text{VEXT}\{<c>\}{<q>}.8\{<Qd>,\} <Qn>, <Qm>, \#<\text{imm}*(size/8)>
\]

and is never the preferred disassembly.

**T1**

\[
|15| 14| 13| 12| 11| 10| 9| 8| 7| 6| 5| 4| 3| 0|
\]

64-bit SIMD vector variant

Applies when Q == 0.

\[
\text{VEXT}\{<c>\}{<q>}.<\text{size}>\{<Dd>,\} <Dn>, <Qm>, \#<\text{imm}>
\]

is equivalent to

\[
\text{VEXT}\{<c>\}{<q>}.8\{<Dd>,\} <Dn>, <Qm>, \#<\text{imm}*(size/8)>
\]

and is never the preferred disassembly.
128-bit SIMD vector variant

Applies when Q == 1.

\texttt{VEXT\{c\}\{q\}\{size\} \{Qd\}, \{Qn\}, \{Qm\}, \#imm}\n
is equivalent to

\texttt{VEXT\{c\}\{q\}\{size\}8 \{Qd\}, \{Qn\}, \{Qm\}, \#imm*(size/8)}

and is never the preferred disassembly.

Assembler symbols

\(<c>\) For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<\text{size}>\) For the 64-bit SIMD vector variant: is the size of the operation, and can be one of 16 or 32.

For the 128-bit SIMD vector variant: is the size of the operation, and can be one of 16, 32 or 64.

\(<\text{Qd}>\) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<\text{Qd}>*2\).

\(<\text{Qn}>\) Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as \(<\text{Qn}>*2\).

\(<\text{Qm}>\) Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as \(<\text{Qm}>*2\).

\(<\text{Dd}>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<\text{Dn}>\) Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

\(<\text{Dm}>\) Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

\(<\text{imm}>\) For the 64-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to (128/<\text{size}>)-1.

For the 128-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to (64/<\text{size}>)-1.

Operation for all encodings

The description of \texttt{VEXT (byte elements)} gives the operational pseudocode for this instruction.
VFMA

Vector Fused Multiply Accumulate multiplies corresponding elements of two vectors, and accumulates the results into the elements of the destination vector. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

VFMA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE; op1_neg = (op == '1');
integer esize;
integer elements;
case sz of
when '0' esize = 32; elements = 2;
when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
regs = if Q == '0' then 1 else 2;

A2

VFMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Half-precision scalar variant

Applies when size == 01.

VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Single-precision scalar variant

Applies when size == 10.
Double-precision scalar variant
Applies when size == 11.

VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '03' && cond != '11110' then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior
If size == '01' && cond != '11110', then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

64-bit SIMD vector variant
Applies when Q == 0.

VFMA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant
Applies when Q == 1.

VFMA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' ||Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && !InITBlock() then UNPREDICTABLE;
advsimd = TRUE; op1_neg = (op == '1');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
        when '1' esize = 16; elements = 4;
    d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
    regs = if Q == '0' then 1 else 2;
**CONSTRAINED UNPREDICTABLE behavior**

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**T2**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>0</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Half-precision scalar variant**

Applies when `size == 01`.

`VFMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>`

**Single-precision scalar variant**

Applies when `size == 10`.

`VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>`

**Double-precision scalar variant**

Applies when `size == 11`.

`VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>`

**Decode for all variants of this encoding**

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

**CONSTRAINED UNPREDICTABLE behavior**

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

<

For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.
For encoding A2, T1 and T2: see *Standard assembler syntax fields* on page F1-7228.

<q>
See *Standard assembler syntax fields* on page F1-7228.
</q>

<dt>
Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:
- F32 when \( \text{sz} = 0 \)
- F16 when \( \text{sz} = 1 \)
</dt>

<qdb>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<Qd>\times 2\).
</qdb>

<qn>
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as \(<Qn>\times 2\).
</qn>

<qm>
Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as \(<Qm>\times 2\).
</qm>

<qdb>
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
</qdb>

<qn>
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
</qn>

<qm>
Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
</qm>

<qdn>
Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
</qdn>

<qdn>
Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
</qdn>

<qdn>
Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
</qdn>

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
if advsimd then // Advanced SIMD instruction
    for r = 0 to regs-1
        for e = 0 to elements-1
            bits(esize) op1 = Elem[D[n+r],e,esize];
            if op1_neg then op1 = FPNeg(op1);
            Elem[D[d+r],e,esize] = FPMulAdd(Elem[D[d+r],e,esize],
                op1, Elem[D[n+r],e,esize], StandardFPSCRValue());
    else // VFP instruction
        case esize of
            when 16
                op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
                S[d] = Zeros(16) : FPMulAdd(S[d]<15:0>, op16, S[m]<15:0>, FPSCR[]);
            when 32
                op32 = if op1_neg then FPNeg(S[n]) else S[n];
                S[d] = FPMulAdd(S[d], op32, S[m], FPSCR[]);
            when 64
                op64 = if op1_neg then FPNeg(D[n]) else D[n];
                D[d] = FPMulAdd(D[d], op64, D[m], FPSCR[]);
```

F6.1.87   VFMAB, VFMAT (BFloat16, vector)

The BFloat16 floating-point widening multiply-add long instruction widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first and second source vectors from BFloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

Unlike other BFloat16 multiplication instructions, this performs a fused multiply-add, without intermediate rounding that uses the Round to Nearest rounding mode and can generate a floating-point exception that causes cumulative exception bits in the FPSCR to be set.

A1

(FEAT_AA32BF16)

```
|1111110|00|D|Vn|Vd|1|1|0|0|1|1|
|1|1|1|1|1|1|1|0|0|0|0|1|1|Vd|Vn|=0
```

A1 variant

VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Qm>

Decode for this encoding

```c
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

T1

(FEAT_AA32BF16)

```
|1111110|00|D|Vn|Vd|1|1|0|0|1|1|
|1|1|1|1|1|1|1|0|0|0|0|1|1|Vd|Vn|=0
```

T1 variant

VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Qm>

Decode for this encoding

```c
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer elements = 128 DIV 32;
integer sel = UInt(Q);
```
Assembler symbols

<b>
Is the bottom or top element specifier, encoded in the "Q" field. It can have the following values:
B 
when Q = 0
T 
when Q = 1
</b>

<q>
See Standard assembler syntax fields on page F1-7228.
</q>

<q>d>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
</q>

<q>n>
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
</q>

<q>m>
Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) operand3 = Q[d>>1];
bits(128) result;

for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2*e + sel, 16] : Zeros(16);
    bits(32) element2 = Elem[operand2, 2*e + sel, 16] : Zeros(16);
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = FPMulAdd(addend, element1, element2,
                              StandardFPSCRValue());

Q[d>>1] = result;
VFMAB, VFMAT (BFloat16, by scalar)

The BFloat16 floating-point widening multiply-add long instruction widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first source vector, and an indexed element in the second source vector from BFloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

Unlike other BFloat16 multiplication instructions, this performs a fused multiply-add, without intermediate rounding that uses the Round to Nearest rounding mode and can generate a floating-point exception that causes cumulative exception bits in the FPSCR to be set.

A1

(Feat_AA32BF16)

VFMA{bt}{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]

Decode for this encoding

if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<2:0>);
integer i = UInt(M:Vm<3>);
integer elements = 128 DIV 32;
integer sel = UInt(Q);

T1

(Feat_AA32BF16)

VFMA{bt}{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]

Decode for this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<2:0>);
integer i = UInt(M:Vm<3>);
integer elements = 128 DIV 32;
integer sel = UInt(Q);
Assembler symbols

<bt>
Is the bottom or top element specifier, encoded in the "Q" field. It can have the following values:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>when Q = 0</td>
</tr>
<tr>
<td>T</td>
<td>when Q = 1</td>
</tr>
</tbody>
</table>

<q>
See Standard assembler syntax fields on page F1-7228.

<Qd>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dm>
Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field.

<index>
Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.

Operation for all encodings

CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(64) operand2 = D[m];
bits(128) operand3 = Q[d>>1];
bits(128) result;

bits(32) element2 = Elem[operand2, i, 16] : Zeros(16);

for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2 + e + sel, 16] : Zeros(16);
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = FPMulAdd(addend, element1, element2,
                                StandardFPSCRValue());

Q[d>>1] = result;
F6.1.89   VFMAL (vector)

Vector Floating-point Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding values in the vectors in the two source SIMD&FP registers, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

In Armv8.2 and Armv8.3, this is an an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note
ID_ISAR6.FHM indicates whether this instruction is supported.

A1

(FEAT_FHM)

64-bit SIMD vector variant

Applies when Q == 0.

VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>

128-bit SIMD vector variant

Applies when Q == 1.

VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>

Decode for all variants of this encoding

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' & Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = $=='1';

T1

(FEAT_FHM)
**64-bit SIMD vector variant**

Applies when Q == 0.

\[ \text{VFMAL}\{<q>\}.F16 \quad \text{<Dd>}, \quad \text{<Sn>}, \quad \text{<Sm>} \]

**128-bit SIMD vector variant**

Applies when Q == 1.

\[ \text{VFMAL}\{<q>\}.F16 \quad \text{<Qd>}, \quad \text{<Dn>}, \quad \text{<Dm>} \]

**Decode for all variants of this encoding**

if InITBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q==1' then 64 else 32;
boolean sub_op = S=='1';

**Assembler symbols**

\( <q> \) See Standard assembler syntax fields on page F1-7228.

\( <Qd> \) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \( <Qd>*2 \).

\( <Dn> \) Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

\( <Dm> \) Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

\( <Dd> \) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\( <Sn> \) Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

\( <Sm> \) Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

**Operation for all encodings**

CheckAdvSIMDEnabled();
bits(datasize) operand1;
bits(datasize) operand2;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

if Q=='0' then
    operand1 = S[n]<datasize-1:0>;
    operand2 = S[m]<datasize-1:0>;
else
    operand1 = D[n]<datasize-1:0>;
    operand2 = D[m]<datasize-1:0>;
for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        element2 = Elem[operand2, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2,
StandardFPSCRValue();
D[d+r] = result;
F6.1.90  **VFMAL (by scalar)**

Vector Floating-point Multiply-Add Long to accumulator (by scalar). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support on page G1-8996.*

In Armv8.2 and Armv8.3, this is an optional instruction. From Armv8.4 it is mandatory for all implementations to support it.

**Note**

ID_ISAR6.FHM indicates whether this instruction is supported.

**A1**

(FEAT_FHM)

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 | 12 | 11 10 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 1 | 0 0 0 | N | Q | M | 1 | Vm |
|-------------|-------------|-------------|-----|----|----|----|-----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 1 1 1 1 1 1 0 0 | D | 0 | 0 | Vn | Vd | 1 0 0 0 | N | Q | M | 1 | Vm |

**64-bit SIMD vector variant**

Applies when \( Q = 0 \).

\[ VFMAL\{<q>\}.F16 \ <D>, \ <S>, \ <M>[<index>] \]

**128-bit SIMD vector variant**

Applies when \( Q = 1 \).

\[ VFMAL\{<q>\}.F16 \ <Q>D, \ <Q>S, \ <Q>M[<index>] \]

**Decode for all variants of this encoding**

\[
\begin{align*}
&\text{if } \neg \text{HaveFP16NoRoundingToFP32Ext()} \text{ then UNDEFINED;} \\
&\text{if } Q = '1' \&\& Vd<0> = '1' \text{ then UNDEFINED;} \\
&\text{integer } d = \text{ UInt}(D:Vd); \\
&\text{integer } n = \text{ if } Q = '1' \text{ then UInt}(N:Vn) \text{ else UInt}(Vn:N); \\
&\text{integer } m = \text{ if } Q = '1' \text{ then UInt}(Vm<2:0>) \text{ else UInt}(Vm<2:0>:M); \\
&\text{integer } index = \text{ if } Q = '1' \text{ then UInt}(M:Vm<3>) \text{ else UInt}(Vm<3>); \\
&\text{integer } size = 32; \\
&\text{integer } regs = \text{ if } Q='1' \text{ then 2 else 1; } \\
&\text{integer } data = \text{ if } Q='1' \text{ then 64 else 32; } \\
&\text{boolean } sub_{op} = S=='1'; \\
\end{align*}
\]

**T1**

(FEAT_FHM)

| 15 14 13 12 | 11 10 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 12 | 11 10 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 1 | 0 0 0 | N | Q | M | 1 | Vm |
| 1 1 1 1 1 1 1 0 0 | D | 0 | 0 | Vn | Vd | 1 0 0 0 | N | Q | M | 1 | Vm |
64-bit SIMD vector variant
Applies when Q == 0.
VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]

128-bit SIMD vector variant
Applies when Q == 1.
VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]

Decode for all variants of this encoding
if InITBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);
integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';

Assembler symbols
&q
See Standard assembler syntax fields on page F1-7228.

<Qd>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn>
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>
Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>:M" field.

<Dd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Sn>
Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm>
Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>:M" field.

<index>
For the 64-bit SIMD vector variant: is the element index in the range 0 to 1, encoded in the "Vm<3>" field.
For the 128-bit SIMD vector variant: is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.

Operation for all encodings
CheckAdvSIMDEnabled();
bits(datasize) operand1 ;
bits(datasize) operand2 ;
bits(64) operand3;
bits(64) result;
b STM(DIV 2) element1;
b STM(DIV 2) element2;
if Q=='0' then
operand1 = S[n]<datasize-1:0>;
operand2 = S[m]<datasize-1:0>;
else
operand1 = D[n]<datasize-1:0>;
operand2 = D[m]<datasize-1:0>;

element2 = Elem[operand2, index, esize DIV 2];
for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r + e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2,
                                           StandardFPSCRValue());
    D[d+r] = result;
F6.1.91 VFMS

Vector Fused Multiply Subtract negates the elements of one vector and multiplies them with the corresponding elements of another vector, adds the products to the corresponding elements of the destination vector, and places the results in the destination vector. The instruction does not round the result of the multiply before the addition.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VFMS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VFMS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE; op1_neg = (op == '1');
integer esize;
integer elements;
case sz of
when '0' esize = 32; elements = 2;
when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
regs = if Q == '0' then 1 else 2;

A2

Half-precision scalar variant

Applies when size == 01.

VFMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar variant

Applies when size == 10.

VFMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
Double-precision scalar variant
Applies when size == 11.
VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior
If size == '01' && cond != '1110', then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

64-bit SIMD vector variant
Applies when Q == 0.
VFMS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant
Applies when Q == 1.
VFMS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE; op1_neg = (op == '1');
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
regs = if Q == '0' then 1 else 2;
**CONSTRAINED UNPREDICTABLE behavior**

If \( sz == '1' \) \&\& InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

### T2

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 3 | 0 | 15 12 11 10 9 8 | 7 6 5 4 3 | 0 |  
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | D | 1 | 0 | Vn | Vd | 1 | 0 | size | N | M | 0 | Vm |

#### Half-precision scalar variant

Applies when \( size == 01 \).

\[
VFMS\{<c>\}{<q>}.F16 <Sd>, <Sn>, <Sm>
\]

#### Single-precision scalar variant

Applies when \( size == 10 \).

\[
VFMS\{<c>\}{<q>}.F32 <Sd>, <Sn>, <Sm>
\]

#### Double-precision scalar variant

Applies when \( size == 11 \).

\[
VFMS\{<c>\}{<q>}.F64 <Dd>, <Dn>, <Dm>
\]

#### Decode for all variants of this encoding

```c
if FPSCR.Len != '000' \| FPSCR.Stride != '00' then UNDEFINED;
if size == '00' \| (size == '01' \&\& !HaveFP16Ext()) then UNDEFINED;
if size == '01' \&\& InITBlock() then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```  

**CONSTRAINED UNPREDICTABLE behavior**

If \( size == '01' \) \&\& InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

#### Assembler symbols

\(<c>\)

For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.
For encoding A2, T1 and T2: see Standard assembler syntax fields on page F1-7228.

See Standard assembler syntax fields on page F1-7228.

Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

- F32 when sz = 0
- F16 when sz = 1

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

### Operation for all encodings

```c
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
if advsimd then // Advanced SIMD instruction
    for r = 0 to regs-1
        for e = 0 to elements-1
            bits(esize) op1 = Elem[D[n+r],e,esize];
            if op1_neg then op1 = FPNeg(op1);
            Elem[D[d+r],e,esize] = FPMulAdd(Elem[D[d+r],e,esize],
                                            op1, Elem[D[m+r],e,esize], StandardFPSCRValue());
else // VFP instruction
    case esize of
        when 16
            op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
            S[d] = Zeros(16) : FPMulAdd(S[d]<15:0>, op16, S[m]<15:0>, FPSCR[]);
        when 32
            op32 = if op1_neg then FPNeg(S[n]) else S[n];
            S[d] = FPMulAdd(S[d],op32,S[m],FPSCR[]);
        when 64
            op64 = if op1_neg then FPNeg(D[n]) else D[n];
            D[d] = FPMulAdd(D[d],op64,D[m],FPSCR[]);
```
F6.1.92 VFMSL (vector)

Vector Floating-point Multiply-Subtract Long from accumulator (vector). This instruction negates the values in the vector of one SIMD&FP register, multiplies these with the corresponding values in another vector, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and Floating-point support on page G1-8996.

In Armv8.2 and Armv8.3, this is an optional instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

A1

(Feat_FHM)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 6 5 4 3 0 |
|----------------|----------------|----------------|----------------|
| 1 1 1 1 1 1 0 0 1 1 0| Vn | Vd | 1 | 0 | 0 | 0 | N | Q | M | 1 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.
VFMSL(<q>).F16 <Dd>, <Sn>, <Sm>

128-bit SIMD vector variant

Applies when Q == 1.
VFMSL(<q>).F16 <Qd>, <Dn>, <Dm>

Decode for all variants of this encoding

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = $=='1';

T1

(Feat_FHM)

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 15 12 11 10 9 8 7 6 5 4 3 0 |
|----------------|----------------|----------------|----------------|
| 1 1 1 1 1 1 1 0 0 1 1 0| Vn | Vd | 1 | 0 | 0 | 0 | N | Q | M | 1 | Vm |
64-bit SIMD vector variant

Applies when Q == 0.

VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>

128-bit SIMD vector variant

Applies when Q == 1.

VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>

Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

CheckAdvSIMDEnabled();
bits(datasize) operand1;
bits(datasize) operand2;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;
if Q=='0' then
  operand1 = S[n]<datasize-1:0>;
  operand2 = S[m]<datasize-1:0>;
else
  operand1 = D[n]<datasize-1:0>;
  operand2 = D[m]<datasize-1:0>;
for r = 0 to regs-1
  operand3 = D[d+r];
  for e = 0 to 1
    element1 = Elem[operand1, 2*r+e, esize DIV 2];
    element2 = Elem[operand2, 2*r+e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2,
StandardFPSCRValue();
D[d+r] = result;
F6.1.93  VFMSL (by scalar)

Vector Floating-point Multiply-Subtract Long from accumulator (by scalar). This instruction multiplies the negated vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

In Armv8.2 and Armv8.3, this is an optional instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

A1

(FEAT_FHM)

VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]

64-bit SIMD vector variant

Applies when Q == 0.

VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]

128-bit SIMD vector variant

Applies when Q == 1.

VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]

Decode for all variants of this encoding

if !HaveFP16WnLNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q==1' then 64 else 32;
boolean sub_op = S=='1';

T1

(FEAT_FHM)

VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]
64-bit SIMD vector variant

Applies when Q == 0.

VFMSL{<q>}.F16 <Dd>, <Sm>, <Sm>[<index>]

128-bit SIMD vector variant

Applies when Q == 1.

VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]

Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);
integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>:M" field.

<index> For the 64-bit SIMD vector variant: is the element index in the range 0 to 1, encoded in the "Vm<3>" field.
For the 128-bit SIMD vector variant: is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.

Operation for all encodings

CheckAdvSIMDEnabled();
bits(datasize) operand1;
bits(datasize) operand2;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;
if Q=='0' then
  operand1 = S[n]<datasize-1:0>;
  operand2 = S[m]<datasize-1:0>;
else
  operand1 = D[n]<datasize-1:0>;
  operand2 = D[m]<datasize-1:0>;
element2 = Elem[operand2, index, esize DIV 2];
for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2,
                                              StandardFPSCRValue());
        D[d+r] = result;
F6.1.94   VFNMA

Vector Fused Negate Multiply Accumulate negates one floating-point register value and multiplies it by another floating-point register value, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The instruction does not round the result of the multiply before the addition.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

A1

```
|31 28|27 25 24|23 22 21|20|19|16|15|12|11|10|9 |8 |7 |6 |5 |4 |3 |0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|l=1111|1|1|0|1|D|0|1|Vn|Vd|1|0|size|N|M|0|Vm|
```

**Half-precision scalar variant**

Applies when `size == 01`.

`VFNMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>`

**Single-precision scalar variant**

Applies when `size == 10`.

`VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>`

**Double-precision scalar variant**

Applies when `size == 11`.

`VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>`

**Decode for all variants of this encoding**

```c
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
op1_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

**CONSTRAINED UNPREDICTABLE behavior**

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
T1

Half-precision scalar variant
Applies when size == 01.
VFNMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar variant
Applies when size == 10.
VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar variant
Applies when size == 11.
VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
op1_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior
If size == '01' && InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols
<c> See Standard assembler syntax fields on page F1-7228.
<q> See Standard assembler syntax fields on page F1-7228.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    case esize of
        when 16
            op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
            S[d] = Zeros(16) : FPMulAdd(FPNeg(S[d]<15:0>), op16, S[m]<15:0>, FPSCR[]);
        when 32
            op32 = if op1_neg then FPNeg(S[n]) else S[n];
            S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], FPSCR[]);
        when 64
            op64 = if op1_neg then FPNeg(D[n]) else D[n];
            D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], FPSCR[]);
```
F6.1.95 VFNMS

Vector Fused Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The instruction does not round the result of the multiply before the addition.

Depending on settings in the CPACR, NSACR, HCPTT, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

```
|31| 28|26|25|24|23|22|21|20|19|16|15|12|11|10| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|l=1111|1|1|1|0|1|D|0|1|Vn|Vd|1|0|size|N|0|M|0|Vm|

cond op
```

**Half-precision scalar variant**

Applies when size == 01.

\[ VFNMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm> \]

**Single-precision scalar variant**

Applies when size == 10.

\[ VFNMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm> \]

**Double-precision scalar variant**

Applies when size == 11.

\[ VFNMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm> \]

**Decode for all variants of this encoding**

```c
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
op1_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
Half-precision scalar variant
Applies when size == 01.
VFNMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar variant
Applies when size == 10.
VFNMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar variant
Applies when size == 11.
VFNMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
op1_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m =UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior
If size == '01' && InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols
<op> See Standard assembler syntax fields on page F1-7228.
<q> See Standard assembler syntax fields on page F1-7228.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<op>
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```markdown
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    case esize of
        when 16
            op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
            S[d] = Zeros(16) : FPMulAdd(FPNeg(S[d]<15:0>), op16, S[m]<15:0>, FPSCR[ ]);
        when 32
            op32 = if op1_neg then FPNeg(S[n]) else S[n];
            S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], FPSCR[ ]);  
        when 64
            op64 = if op1_neg then FPNeg(D[n]) else D[n];
            D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], FPSCR[ ]);  
```

```markdown
```
F6.1.96   VHADD

Vector Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated. For rounded results, see VRHADD).

The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when Q == 0.

VHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Assembler symbols

<e>  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
    For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q>  See Standard assembler syntax fields on page F1-7228.

<dt>  Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:
    S8    when U = 0, size = 00
    S16   when U = 0, size = 01
    S32   when U = 0, size = 10
    U8    when U = 1, size = 00
    U16   when U = 1, size = 01
    U32   when U = 1, size = 10

<Qd>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>  Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>  Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>  Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>  Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = (if add then op1+op2 else op1-op2) >> 1;
            Elem[D[d+r],e,esize] = result<esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.97 VHSUB

Vector Halving Subtract subtracts the elements of the second operand from the corresponding elements of the first operand, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated. There is no rounding version.

The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>19 16 15 12 11 10 9 8 7 6 5 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1</td>
<td>U</td>
<td>0</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VHSUB{<c>}{<q>}{<dt>}{<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VHSUB{<c>}{<q>}{<dt>}{<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
<td>U</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VHSUB{<c>}{<q>}{<dt>}{<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VHSUB{<c>}{<q>}{<dt>}{<Qd>, }<Qn>, <Qm>
Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Assembler symbols

<>  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q>  See Standard assembler syntax fields on page F1-7228.

<dt>  Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:
S8  when U = 0, size = 00
S16 when U = 0, size = 01
S32 when U = 0, size = 10
U8  when U = 1, size = 00
U16 when U = 1, size = 01
U32 when U = 1, size = 10

<Qd>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>  Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>  Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>  Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>  Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = (if add then op1+op2 else op1-op2) >> 1;
            Elem[D[d+r],e,esize] = result<esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
Vector move Insertion. This instruction copies the lower 16 bits of the 32-bit source SIMD&FP register into the upper 16 bits of the 32-bit destination SIMD&FP register, while preserving the values in the remaining bits.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

(FEAT_FP16)

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & Vd & 1 & 0 & 1 & 0 & 1 & 1 & M & 0 & Vm
\end{array}
\]

A1 variant

VINS{<q>}.F16 <Sd>, <Sm>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); m = UInt(Vm:M);

T1

(FEAT_FP16)

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & Vd & 1 & 0 & 1 & 0 & 1 & 1 & M & 0 & Vm
\end{array}
\]

T1 variant

VINS{<q>}.F16 <Sd>, <Sm>

Decode for this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); m = UInt(Vm:M);

CONstrained UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Ss> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEncabled(TRUE);
    S[d] = S[m]<15:0> : S[d]<15:0>;


F6.1.99 VJCVT

Javascript Convert to signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD&FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register. If the result is too large to be accommodated as a signed 32-bit integer, then the result is the integer modulo 2^{32}, as held in a 32-bit signed integer.

This instruction can generate a floating-point exception. Depending on the settings in FPSCR, the exception results in either a flag being set or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps on page E1-7148.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

(FEAT_JSCVT)

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|------------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=1111           | 1  | 1  | 1  | 0  | 1  | D  | 1  | 1  | 0  | 0  | 1  | Vd | 1  | 0  | 1  | 1  | 1  | M  | 0  | Vm |
| cond             |

**A1 variant**

VJCVT{<q>}.S32.F64 <Sd>, <Dm>

**Decode for this encoding**

if !HaveFJCVTZSExt() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE;
d = UInt(Vd:D);  m = UInt(M:Vm);

T1

(FEAT_JSCVT)

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | D  | 1  | 1  | 0  | 0  | 1  | Vd | 1  | 0  | 1  | 1  | 1  | M  | 0  | Vm |

**T1 variant**

VJCVT{<q>}.S32.F64 <Sd>, <Dm>

**Decode for this encoding**

if !HaveFJCVTZSExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
d = UInt(Vd:D);  m = UInt(M:Vm);

**Assembler symbols**

<q> See Standard assembler syntax fields on page F1-7228.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
Operation for all encodings

EncodingSpecificOperations();
CheckVFPEnabled(TRUE);
bits(64) fltval = D[m];
bits(32) intval;
bit Z;
(intval, Z) = FPToFixedJS(fltval, FPSCR[], FALSE, 32);
FPSCR<31:28> = '0':Z:'00';
S[d] = intval;
F6.1.100 VLD1 (single element to one lane)

Load single 1-element structure to one lane of one register loads one element from memory into one element of a
register. Elements of the register that are not loaded are unchanged. For details of the addressing mode, see The
Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which
the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For
more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

Offset variant
Applies when Rm == 1111.
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}

Post-indexed variant
Applies when Rm == 1101.
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed variant
Applies when Rm != 1111.
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

Decode for all variants of this encoding

if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<0> != '0' then UNDEFINED;
  ebytes = 1;  index = UInt(index_align<3:1>);  alignment = 1;
  d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
  wback = (m != 15);  register_index = (m != 15 && m != 13);
  if n == 15 then UNPREDICTABLE;

A2

Offset variant
Applies when Rm == 1111.
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed variant
Applies when Rm == 1101.
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!
**Post-indexed variant**

Applies when \( Rm \neq 11x1 \).

\[
\text{VLDI}{<c>}{<q>}.<\text{size}> <\text{list}>, [<Rn>{:<\text{align}>}], <Rm>
\]

**Decode for all variants of this encoding**

if \( \text{size} == '11' \) then SEE "VLDI (single element to all lanes)";
if \( \text{index\_align} <2> != '0' \) then UNDEFINED;
\( e\text{bytes} = 2; \quad \text{index} = \text{UINT}(\text{index\_align}<3:2>); \)
\( \text{alignment} = \text{if} \ \text{index\_align}<0> == '0' \ \text{then} \ 1 \ \text{else} \ 2; \)
\( d = \text{UINT}(D:Vd); \quad n = \text{UINT}(Rn); \quad m = \text{UINT}(Rm); \)
\( \text{wback} = (m != 15); \quad \text{register\_index} = (m != 15 \& \& m != 13); \)
if \( n == 15 \) then UNPREDICTABLE;

**Offset variant**

Applies when \( Rm == 1111 \).

\[
\text{VLDI}{<c>}{<q>}.<\text{size}> <\text{list}>, [<Rn>{:<\text{align}>}]
\]

**Post-indexed variant**

Applies when \( Rm == 1101 \).

\[
\text{VLDI}{<c>}{<q>}.<\text{size}> <\text{list}>, [<Rn>{:<\text{align}>}]!
\]

**Post-indexed variant**

Applies when \( Rm != 11x1 \).

\[
\text{VLDI}{<c>}{<q>}.<\text{size}> <\text{list}>, [<Rn>{:<\text{align}>}], <Rm>
\]

**Decode for all variants of this encoding**

if \( \text{size} == '11' \) then SEE "VLDI (single element to all lanes)";
if \( \text{index\_align} <2> != '0' \) then UNDEFINED;
\( \text{alignment} = \text{if} \ \text{index\_align}<1:0> == '00' \ \text{then} \ 1 \ \text{else} \ 4; \)
\( \text{alignment} = \text{if} \ \text{index\_align}<1:0> == '00' \ \text{then} \ 1 \ \text{else} \ 4; \)
\( d = \text{UINT}(D:Vd); \quad n = \text{UINT}(Rn); \quad m = \text{UINT}(Rm); \)
\( \text{wback} = (m != 15); \quad \text{register\_index} = (m != 15 \& \& m != 13); \)
if \( n == 15 \) then UNPREDICTABLE;

**Offset variant**

Applies when \( Rm == 1111 \).
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}

**Post-indexed variant**
Applies when Rm == 1101.
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

**Post-indexed variant**
Applies when Rm != 11x1.
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

**Decode for all variants of this encoding**
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); alignment = 1;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;

T2

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>1</td>
</tr>
</tbody>
</table>

**Offset variant**
Applies when Rm == 1111.
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}

**Post-indexed variant**
Applies when Rm == 1101.
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

**Post-indexed variant**
Applies when Rm != 11x1.
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

**Decode for all variants of this encoding**
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 2; index = UInt(index_align<3:2>);
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
T3

Offset variant
Applies when \( Rm == 1111 \).

\[
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
\]

Post-indexed variant
Applies when \( Rm == 1101 \).

\[
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
\]

Post-indexed variant
Applies when \( Rm != 11x1 \).

\[
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>
\]

Decode for all variants of this encoding

if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;

\[
ebytes = 4;  \quad \text{index} = \text{UInt}(\text{index_align}<3>);
\]

alignment = if index_align<1:0> == '00' then 1 else 4;
\[d = \text{UInt}(D:Vd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]
\[wback = (m != 15); \quad \text{register_index} = (m != 15 && m != 13);
\]
if n == 15 then UNPREDICTABLE;

Notes for all encodings
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

\(<c>\) For encoding A1, A2 and A3: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1, T2 and T3: see Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<size>\) Is the data size, encoded in the "size" field. It can have the following values:

8 when size = 00
16 when size = 01
32 when size = 10

\(<list>\) Is a list containing the single 64-bit name of the SIMD&FP register holding the element.

The list must be \{ \(<Dd>\)[\{\<index>\}] \}.

The register \(<Dd>\) is encoded in the "D:Vd" field.

The permitted values and encoding of \(<index>\) depend on \(<size>\):

\(<size> == \&<index>\) is in the range 0 to 7, encoded in the "index_align<3:1>" field.
**Operation for all encodings**

```plaintext
if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  address = R[n];  iswrite = FALSE;
  - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
  Elem[D[d],index] = MemU[address,ebytes];
  if wback then
    if register_index then
      R[n] = R[n] + R[m];
    else
      R[n] = R[n] + ebytes;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.101   VLD1 (single element to all lanes)

Load single 1-element structure and replicate to all lanes of one register loads one element from memory into every element of one or two vectors. For details of the addressing mode, see The Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

```
[31 30 29 28 27 26 25 24 23 22 21 20 | 19 16 15 | 12 11 10 9 8 7 6 5 4 3 2 1 0 |
  1 1 1 1 0 1 0 0 1 1 0 | D | 1 | 0 | Rn | Vd | 1 1 0 0 | size | T | a | Rm |
```

**Offset variant**

Applies when Rm == 1111.
VLD1(<c>{<q>}.<size> <list>, [<Rn>{:<align}>])

**Post-indexed variant**

Applies when Rm == 1101.
VLD1(<c>{<q>}.<size> <list>, [<Rn>{:<align}>])!

**Post-indexed variant**

Applies when Rm != 11x1.
VLD1(<c>{<q>}.<size> <list>, [<Rn>{:<align}>]), Rm

**Decode for all variants of this encoding**

if size == '11' || (size == '00' && a == '1') then UNDEFINED;
ebytes = 1 << UInt(size);  regs = if T == '0' then 1 else 2;
alignment = if a == '0' then 1 else ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

```
[15 14 13 12 | 11 10 9 8 7 6 5 4 3 0 | 12 | 11 10 9 8 7 6 5 4 3 0 | 1 1 1 1 1 0 0 1 1 | D | 1 | 0 | Rn | Vd | 1 1 0 0 | size | T | a | Rm |
```
**Offset variant**

Applies when \( Rm = \text{1111} \).

\[
\text{VLDI}{<c>}{<q>}.<\text{size} > \text{<list>}, [\text{<Rn>:{<align>}}]}
\]

**Post-indexed variant**

Applies when \( Rm = \text{1101} \).

\[
\text{VLDI}{<c>}{<q>}.<\text{size} > \text{<list>}, [\text{<Rn>:{<align>}}]!
\]

**Post-indexed variant**

Applies when \( Rm \neq \text{11x1} \).

\[
\text{VLDI}{<c>}{<q>}.<\text{size} > \text{<list>}, [\text{<Rn>:{<align>}}], <Rm>
\]

**Decode for all variants of this encoding**

\[
\begin{align*}
\text{if } \text{size} &= \ '11' \quad | \ (\text{size} = \ '00' \quad \&\& \quad a = \ '1') \text{ then UNDEFINED;} \\
\text{ebytes} &= 1 \text{ << } \text{UInt(size);} \\
\text{regs} &= \text{if } T = \ '0' \text{ then } 1 \text{ else } 2; \\
\text{alignment} &= \text{if } a = \ '0' \text{ then } 1 \text{ else } \text{ebytes}; \\
\text{d} &= \text{UInt(D:Vd);} \\
\text{n} &= \text{UInt(Rn);} \\
\text{m} &= \text{UInt(Rm);} \\
\text{wback} &= (m != 15); \\
\text{register_index} &= (m != 15 \quad \&\& \quad m = 13); \\
\text{if } n &= \text{15} || \text{d+regs} > 32 \text{ then UNPREDICTABLE;}
\end{align*}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( d+\text{regs} > 32 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VLDI (single element to all lanes) on page K1-11577.

**Assembler symbols**

- \(<c>\)
  - For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
  - For encoding T1: see Standard assembler syntax fields on page F1-7228.

- \(<q>\)
  - See Standard assembler syntax fields on page F1-7228.

- \(<\text{size}>\)
  - Is the data size, encoded in the "size" field. It can have the following values:
    - 8 when \( \text{size} = \ '00' \)
    - 16 when \( \text{size} = \ '01' \)
    - 32 when \( \text{size} = \ '10' \)
  - The encoding \( \text{size} = \ '11' \) is reserved.

- \(<\text{list}>\)
  - Is a list containing the 64-bit names of the SIMD&FP registers.
  - The list must be one of:
    - \{ <Dd>{} \} Encoded in the "T" field as 0.
{ <Dd>[], <Dd+1>[] } Encoded in the "T" field as 1.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> When <size> == 8, <align> must be omitted, otherwise it is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see Unaligned data access on page E2-7193, and is encoded in the "a" field as 0.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<align> == 16<align> is 16, meaning 16-bit alignment, encoded in the "a" field as 1.
<align> == 32<align> is 32, meaning 32-bit alignment, encoded in the "a" field as 1.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see The Advanced SIMD addressing mode on page F1-7249.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see The Advanced SIMD addressing mode on page F1-7249.

**Operation for all encodings**

```c
if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  address = R[n]; iswrite = FALSE;
  - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
  constant integer esize = 8 * ebytes;
  bits(esize) element = MemU[address, ebytes];
  bits(64) replicated_element = Replicate(element, 64 DIV esize);
  for r = 0 to regs-1
    D[d+r] = replicated_element;
  if wback then
    if register_index then
      R[n] = R[n] + R[m];
    else
      R[n] = R[n] + ebytes;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.102  **VLD1** (multiple single elements)

Load multiple single 1-element structures to one, two, three, or four registers loads elements from memory into one, two, three, or four registers, without de-interleaving. Every element of each register is loaded. For details of the addressing mode, see *The Advanced SIMD addressing mode on page F1-7249.*

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see *Enabling Advanced SIMD and floating-point support on page G1-8996.*

| 31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 6 5 4 3 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | D | 1 | 0 | Rn | Vd | 0 | 1 | 1 | 1 | size | align | Rm |

**Offset variant**

Applies when Rm == 1111.

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}

**Post-indexed variant**

Applies when Rm == 1101.

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

**Post-indexed variant**

Applies when Rm != 11x1.

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

**Decode for all variants of this encoding**

\[
\begin{align*}
\text{regs} &= 1; \quad \text{if align} \neq '1' \text{ then UNDEFINED;} \\
\text{alignment} &= \text{if align == '00' then 1 else 4 \ll \text{UInt}}(\text{align}) \; \text{\ll INT(size); elements = 8 \text{DIV} ebytes}; \\
\text{d} &= \text{\text{UInt}}(D:Vd); \quad n = \text{\text{UInt}}(Rn); \quad m = \text{\text{UInt}}(Rm); \\
\text{wback} &= (m != 15); \quad \text{register_index} = (m != 15 \&\& m != 13); \quad \text{if n == 15 then UNPREDICTABLE};
\end{align*}
\]

A2

| 31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 6 5 4 3 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | D | 1 | 0 | Rn | Vd | 1 | 0 | 1 | 0 | size | align | Rm |

**Offset variant**

Applies when Rm == 1111.

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}

**Post-indexed variant**

Applies when Rm == 1101.

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
**Post-indexed variant**
Applies when Rm != 11x1.

\[ \text{VLDI}\{\langle c\rangle\}\{\langle q\rangle\}\langle\text{size}\rangle \langle\text{list}\rangle, [\langle\text{Rn}\rangle\{:<\text{align}\}\}], <\text{Rm}> \]

**Decode for all variants of this encoding**

\[ \text{regs} = 2; \text{if align == '11' then UNDEFINED}; \]
\[ \text{alignment = if align == '00' then 1 else 4 << UInt(align)}; \]
\[ \text{ebytes = 1 << UInt(size)}; \text{elements = 8 DIV ebytes}; \]
\[ \text{d = UInt(D:Vd)}; \text{n = UInt(Rn)}; \text{m = UInt(Rm)}; \]
\[ \text{wback = (m != 15); register_index = (m != 15 && m != 13)}; \]
\[ \text{if n == 15 || d+regs > 32 then UNPREDICTABLE}; \]

**CONstrained UNPREDICTABLE behavior**

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

**A3**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|------------|-----------|-----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 1 1 1 0 1 0 0 0 0 D | 1 0 | Rn | Vd | 0 | 1 | 1 | 0 | size | align | Rm |

**Offset variant**

Applies when Rm == 1111.

\[ \text{VLDI}\{\langle c\rangle\}\{\langle q\rangle\}\langle\text{size}\rangle \langle\text{list}\rangle, [\langle\text{Rn}\rangle\{:<\text{align}\}\}] \]

**Post-indexed variant**

Applies when Rm == 1101.

\[ \text{VLDI}\{\langle c\rangle\}\{\langle q\rangle\}\langle\text{size}\rangle \langle\text{list}\rangle, [\langle\text{Rn}\rangle\{:<\text{align}\}\}]! \]

**Post-indexed variant**

Applies when Rm != 11x1.

\[ \text{VLDI}\{\langle c\rangle\}\{\langle q\rangle\}\langle\text{size}\rangle \langle\text{list}\rangle, [\langle\text{Rn}\rangle\{:<\text{align}\}\}], <\text{Rm}> \]

**Decode for all variants of this encoding**

\[ \text{regs} = 3; \text{if align}\langle c\rangle == '1' then UNDEFINED}; \]
\[ \text{alignment = if align == '00' then 1 else 4 << UInt(align)}; \]
\[ \text{ebytes = 1 << UInt(size)}; \text{elements = 8 DIV ebytes}; \]
\[ \text{d = UInt(D:Vd)}; \text{n = UInt(Rn)}; \text{m = UInt(Rm)}; \]
\[ \text{wback = (m != 15); register_index = (m != 15 && m != 13)}; \]
\[ \text{if n == 15 || d+regs > 32 then UNPREDICTABLE}; \]
**CONSTRANDED UNPREDICTABLE behavior**

If \( d + \text{regs} > 32 \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- One or more of the SIMD and floating-point registers are **UNKNOWN**. If the instruction specifies writeback, the base register becomes **UNKNOWN**. This behavior does not affect any general-purpose registers.

### A4

| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4 3 | 0 |
|-------------|-------------|-------------|-----|-----|---------------------|--|-----|
| 1 1 1 1 0 1 0 0 0 | D 1 0 | Rn | Vd | 0 0 1 0 | size | align | Rm |

**Offset variant**

Applies when \( Rm == 1111 \).

\[ \text{VLDI}(<c>{<q>}.<size> <list>, [<Rn} {:<align}>] \]

**Post-indexed variant**

Applies when \( Rm == 1101 \).

\[ \text{VLDI}(<c>{<q>}.<size> <list>, [<Rn} {:<align}>]! \]

**Post-indexed variant**

Applies when \( Rm != 11x1 \).

\[ \text{VLDI}(<c>{<q>}.<size> <list>, [<Rn} {:<align}>], <Rm> \]

**Decode for all variants of this encoding**

\[
\begin{align*}
\text{regs} &= 4; \\
\text{alignment} &= \text{if align} == '00' \text{ then 1 else 4} \ll \text{UInt}(\text{align}); \\
\text{ebytes} &= 1 \ll \text{UInt}(\text{size}); \text{ elements } = \text{8 DIV ebytes}; \\
\text{d} &= \text{UInt}(D:Vd); \text{ n } = \text{UInt}(Rn); \text{ m } = \text{UInt}(Rm); \\
\text{wback} &= (m != 15); \text{ register_index } = (m != 15 \&\& m != 13); \\
\text{if n == 15 \|| d+regs > 32 then UNPREDICTABLE;}
\end{align*}
\]

### T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4</th>
<th>3 0</th>
<th>15</th>
<th>12</th>
<th>11 10 9 8 7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1</td>
<td>0 0 1 0</td>
<td>D 1 0</td>
<td>Rn</td>
<td>Vd</td>
<td>0 1 1 1</td>
<td>size</td>
<td>align</td>
</tr>
</tbody>
</table>

**CONSTRANDED UNPREDICTABLE behavior**

If \( d + \text{regs} > 32 \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- One or more of the SIMD and floating-point registers are **UNKNOWN**. If the instruction specifies writeback, the base register becomes **UNKNOWN**. This behavior does not affect any general-purpose registers.
Offset variant
Applies when \( Rm == 1111 \).
\[
\text{VLDI} \{ \text{c} \} \{ \text{q} \} . \text{<size>} \ \text{<list>}, \ [\text{<Rn>}{:\text{<align>}}] \]

Post-indexed variant
Applies when \( Rm == 1101 \).
\[
\text{VLDI} \{ \text{c} \} \{ \text{q} \} . \text{<size>} \ \text{<list>}, \ [\text{<Rn>}{:\text{<align>}}]!
\]

Post-indexed variant
Applies when \( Rm != 11x1 \).
\[
\text{VLDI} \{ \text{c} \} \{ \text{q} \} . \text{<size>} \ \text{<list>}, \ [\text{<Rn>}{:\text{<align>}}] \!, \text{<Rm>}
\]

Decode for all variants of this encoding
\[
\begin{align*}
\text{regs} &= 1; \quad \text{if align<1> == '1' then UNDEFINED;} \\
\text{alignment} &= \text{if align == '00' then 1 else } 4 \times \text{UInt(align)}; \\
\text{ebytes} &= 1 \times \text{UInt(size)}; \quad \text{elements} = 8 \text{ DIV ebytes}; \\
\text{d} &= \text{UInt(D:Vd)}; \quad \text{n} = \text{UInt(Rn)}; \quad \text{m} = \text{UInt(Rm)}; \\
\text{wback} &= \text{(m != 15)}; \quad \text{register_index} = \text{(m != 15 && m != 13)}; \\
\text{if n == 15 then UNPREDICTABLE;}
\end{align*}
\]

T2

| 15 14 13 12 | 11 10 9 8 7 6 5 4 | 3 0 | 12 11 10 9 8 7 6 5 4 | 3 0 |
| 1 1 1 1 0 0 1 0 | D 1 0 | Rn | Vd | 1 0 1 0 | size | align | Rm |

Offset variant
Applies when \( Rm == 1111 \).
\[
\text{VLDI} \{ \text{c} \} \{ \text{q} \} . \text{<size>} \ \text{<list>}, \ [\text{<Rn>}{:\text{<align>}}] \]

Post-indexed variant
Applies when \( Rm == 1101 \).
\[
\text{VLDI} \{ \text{c} \} \{ \text{q} \} . \text{<size>} \ \text{<list>}, \ [\text{<Rn>}{:\text{<align>}}]!
\]

Post-indexed variant
Applies when \( Rm != 11x1 \).
\[
\text{VLDI} \{ \text{c} \} \{ \text{q} \} . \text{<size>} \ \text{<list>}, \ [\text{<Rn>}{:\text{<align>}}] \!, \text{<Rm>}
\]

Decode for all variants of this encoding
\[
\begin{align*}
\text{regs} &= 2; \quad \text{if align == '11' then UNDEFINED;} \\
\text{alignment} &= \text{if align == '00' then 1 else } 4 \times \text{UInt(align)}; \\
\text{ebytes} &= 1 \times \text{UInt(size)}; \quad \text{elements} = 8 \text{ DIV ebytes}; \\
\text{d} &= \text{UInt(D:Vd)}; \quad \text{n} = \text{UInt(Rn)}; \quad \text{m} = \text{UInt(Rm)}; \\
\text{wback} &= \text{(m != 15)}; \quad \text{register_index} = \text{(m != 15 && m != 13)}; \\
\text{if n == 15 || d<regs > 32 then UNPREDICTABLE;}
\end{align*}
\]
**CONSTRANGED UNPREDICTABLE behavior**

If \( d + \text{regs} > 32 \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- One or more of the SIMD and floating-point registers are **UNKNOWN**. If the instruction specifies writeback, the base register becomes **UNKNOWN**. This behavior does not affect any general-purpose registers.

**T3**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 | 1 1 1 1 1 0 0 1 0 D 1 0 Rn Vd 0 1 1 0 size align Rm |

**Offset variant**

Applies when \( Rm == 1111 \).

\[
\text{VLDI}\{<c>\}{<q>}.<size> \text{ <list> }, [<Rn＞:<align>]]
\]

**Post-indexed variant**

Applies when \( Rm == 1101 \).

\[
\text{VLDI}\{<c>\}{<q>}.<size> \text{ <list> }, [<Rn＞:<align>]]!
\]

**Post-indexed variant**

Applies when \( Rm != 11x1 \).

\[
\text{VLDI}\{<c>\}{<q>}.<size> \text{ <list> }, [<Rn＞:<align>]], <Rm>
\]

**Decode for all variants of this encoding**

\[
\begin{align*}
\text{regs} = 3; & \text{ if align}<1> == '1' \text{ then UNDEFINED;} \\
\text{alignment} = & \text{ if align == '00' then 1 else 4} \times \text{ UInt(align);} \\
\text{ebytes} = & \text{ 1} \times \text{ UInt(size);} \quad \text{elements} = \text{ 8} \times \text{ ebytes;} \\
\text{d} = & \text{ UInt(D:Vd);} \quad \text{n} = \text{ UInt(Rn);} \\
\text{m} = & \text{ UInt(Rm);} \\
\text{wback} = & \text{ (m != 15);} \quad \text{register_index} = \text{ (m != 15 \&\& m != 13);} \\
\text{if n == 15 || d+regs > 32 then UNPREDICTABLE;}
\end{align*}
\]

**CONSTRANGED UNPREDICTABLE behavior**

If \( d + \text{regs} > 32 \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- One or more of the SIMD and floating-point registers are **UNKNOWN**. If the instruction specifies writeback, the base register becomes **UNKNOWN**. This behavior does not affect any general-purpose registers.

**T4**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 | 1 1 1 1 1 0 0 1 0 D 1 0 Rn Vd 0 0 1 0 size align Rm |
**Offset variant**

Applies when \( Rm = 1111 \).

\[
\text{VLDI}{<c>}{<q>}{<size> <list>, \ [<Rn>:{<align}>]}
\]

**Post-indexed variant**

Applies when \( Rm = 1101 \).

\[
\text{VLDI}{<c>}{<q>}{<size> <list>, \ [<Rn>:{<align}>]}!
\]

**Post-indexed variant**

Applies when \( Rm \neq 11x1 \).

\[
\text{VLDI}{<c>}{<q>}{<size> <list>, \ [<Rn>:{<align}>]}, \ <Rm>
\]

**Decode for all variants of this encoding**

\[
\text{regs} = 4; \\
\text{alignment} = \text{if align} == '00' \text{ then 1 else } 4 \ll \text{UInt(align)}; \\
\text{ebytes} = 1 \ll \text{UInt(size)}; \ \text{elements} = 8 \div \text{ebytes}; \\
\text{d} = \text{UInt(D:Vd)}; \ \text{n} = \text{UInt(Rn)}; \ \text{m} = \text{UInt(Rm)}; \\
\text{wback} = (m != 15); \ \text{register_index} = (m != 15 \&\& m != 13); \\
\text{if n == 15 || d+regs > 32 then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( d+regs > 32 \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- One or more of the SIMD and floating-point registers are **UNKNOWN**. If the instruction specifies writeback, the base register becomes **UNKNOWN**. This behavior does not affect any general-purpose registers.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VLDI (multiple single elements)* on page K1-11577.

Related encodings: See *Advanced SIMD element or structure load/store* on page F3-7350 for the T32 instruction set, or *Advanced SIMD element or structure load/store* on page F4-7434 for the A32 instruction set.

**Assembler symbols**

- `<c>`
  For encoding A1, A2, A3 and A4: see *Standard assembler syntax fields* on page F1-7228. This encoding must be unconditional.
  For encoding T1, T2, T3 and T4: see *Standard assembler syntax fields* on page F1-7228.

- `<q>`
  See *Standard assembler syntax fields* on page F1-7228.

- `<size>`
  Is the data size, encoded in the "size" field. It can have the following values:
  
  8 when size = 00  
  16 when size = 01  
  32 when size = 10  
  64 when size = 11  

- `<list>`
  Is a list containing the 64-bit names of the SIMD&FP registers.
The list must be one of:

{ <Dd> } Single register. Selects the A1 and T1 encodings of the instruction.
{ <Dd>, <Dd+1> } Two single-spaced registers. Selects the A2 and T2 encodings of the instruction.
{ <Dd>, <Dd+1>, <Dd+2> } Three single-spaced registers. Selects the A3 and T3 encodings of the instruction.
{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> } Four single-spaced registers. Selects the A4 and T4 encodings of the instruction.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see Unaligned data access on page E2-7193, and is encoded in the "align" field as 0b00.
Whenever <align> is present, the permitted values are:

- 64  64-bit alignment, encoded in the "align" field as 0b01.
- 128 128-bit alignment, encoded in the "align" field as 0b10. Available only if <list> contains two or four registers.
- 256 256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as 0<align>, see The Advanced SIMD addressing mode on page F1-7249.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see The Advanced SIMD addressing mode on page F1-7249.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to regs-1
        for e = 0 to elements-1
            if ebytes != 8 then
                data = MemU[address, ebytes];
            else
                - = AArch32.CheckAlignment(address, ebytes, AccType_NORMAL, iswrite);
                data|31:0| = if BigEndian(AccType_NORMAL) then MemU[address+4,4] else MemU[address,4];
                data|63:32| = if BigEndian(AccType_NORMAL) then MemU[address,4] else MemU[address+4,4];
                Elem[D[d+r],e] = data;
                address = address + ebytes;
            if wback then
                if register_index then
                    R[n] = R[n] + R[m];
                else
                    R[n] = R[n] + 8*regs;
            
**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.103  VLD2 (single 2-element structure to one lane)

Load single 2-element structure to one lane of two registers loads one 2-element structure from memory into corresponding elements of two registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode, see The Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

```
[31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 4 3 0 ]
  1 1 1 1 0 1 0 0 1 | D | 1 0 | Rn | Vd | 0 0 0 1 | index_align | Rm |
```

**Offset variant**

Applies when \( Rm = 1111 \).

\( \text{VLD2} \{<c>\}{<q>}.<size> \text{ <list>}, \ [<Rn>{:<align>}]} \)

**Post-indexed variant**

Applies when \( Rm = 1101 \).

\( \text{VLD2} \{<c>\}{<q>}.<size> \text{ <list>}, \ [<Rn>{:<align>}]}! \)

**Post-indexed variant**

Applies when \( Rm \neq 11x1 \).

\( \text{VLD2} \{<c>\}{<q>}.<size> \text{ <list>}, \ [<Rn>{:<align>}]}, \ Rm \)

**Decode for all variants of this encoding**

if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";

ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;

\( d = \text{UInt}(D:Vd);  d2 = d + inc;  n = \text{UInt}(Rn);  m = \text{UInt}(Rm); \)

\( wback = (m != 15);  register_index = (m != 15 & m != 13); \)

if \( n == 15 || d2 > 31 \) then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \( d2 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.
**Offset variant**
Applies when \( Rm = 1111 \).
\[ \text{VLD2} \{<c>\} \{<q>\} \{<size> \} <\text{list}>, \left[ <Rn> \{<align> \} \right] \]

**Post-indexed variant**
Applies when \( Rm = 1101 \).
\[ \text{VLD2} \{<c>\} \{<q>\} \{<size> \} <\text{list}>, \left[ <Rn> \{<align> \} \right]! \]

**Post-indexed variant**
Applies when \( Rm \neq 11x1 \).
\[ \text{VLD2} \{<c>\} \{<q>\} \{<size> \} <\text{list}>, \left[ <Rn> \{<align> \} \right], <Rm> \]

**Decode for all variants of this encoding**

if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
if n == 15 || d2 > 31 then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**
If \( d2 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

**A3**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>index_align</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Offset variant**
Applies when \( Rm = 1111 \).
\[ \text{VLD2} \{<c>\} \{<q>\} \{<size> \} <\text{list}>, \left[ <Rn> \{<align> \} \right] \]

**Post-indexed variant**
Applies when \( Rm = 1101 \).
\[ \text{VLD2} \{<c>\} \{<q>\} \{<size> \} <\text{list}>, \left[ <Rn> \{<align> \} \right]! \]

**Post-indexed variant**
Applies when \( Rm \neq 11x1 \).
\[ \text{VLD2} \{<c>\} \{<q>\} \{<size> \} <\text{list}>, \left[ <Rn> \{<align> \} \right], <Rm> \]
Decode for all variants of this encoding

if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
icc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 & m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

| 15 14 13 12|11 10 9 8|7 6 5 4|3 0 |15 12|11 10 9 8|7 4 3 0 |
| 1 1 1 1 0 0 1 1 | D 1 0 | Rn | Vd | 0 0 0 1 | index_align | Rm |

Offset variant

Applies when Rm == 1111.
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed variant

Applies when Rm == 1101.
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed variant

Applies when Rm != 11x1.
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

Decode for all variants of this encoding

if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
ebytes = 1; index = UInt(index_align<3>); inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 & m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• One or more of the SIMD and floating-point registers are **UNKNOWN**. If the instruction specifies writeback, the base register becomes **UNKNOWN**. This behavior does not affect any general-purpose registers.

**T2**

| 15 14 13 12 | 11 10 9 8 7 6 5 4 3 | 0 | 15 12 11 10 9 8 7 4 3 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | D | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | index_align | Rm |

**Offset variant**

Applies when \( Rm == 1111 \).

\( \text{VLD2}\{<c>\}{<q>}.<\text{size}>, \ [<\text{Rn}\{:<\text{align}>\}] \)

**Post-indexed variant**

Applies when \( Rm == 1101 \).

\( \text{VLD2}\{<c>\}{<q>}.<\text{size}>, \ [<\text{Rn}\{:<\text{align}>\}!] \)

**Post-indexed variant**

Applies when \( Rm != 11x1 \).

\( \text{VLD2}\{<c>\}{<q>}.<\text{size}>, \ [<\text{Rn}\{:<\text{align}>\}], \ <\text{Rm} \)

**Decode for all variants of this encoding**

if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)"

\( ebytes = 2; \ \text{index} = \text{UInt}(\text{index_align}<3:2>); \)
\( \text{inc} = \text{if index_align}<1> == '0' \text{ then } 1 \text{ else } 2; \)
\( \text{alignment} = \text{if index_align}<0> == '0' \text{ then } 1 \text{ else } 4; \)
\( d = \text{UInt}(D:Vd); \ \text{d2} = d + \text{inc}; \ \text{n} = \text{UInt}(\text{Rn}); \ \text{m} = \text{UInt}(\text{Rm}); \)
\( \text{wback} = (m != 15); \ \text{register_index} = (m != 15 \&\& m != 13); \)
\( \text{if n == 15 || d2 > 31 then UNPREDICTABLE}; \)

**CONSTRAINED UNPREDICTABLE behavior**

If \( d2 > 31 \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- One or more of the SIMD and floating-point registers are **UNKNOWN**. If the instruction specifies writeback, the base register becomes **UNKNOWN**. This behavior does not affect any general-purpose registers.

**T3**

| 15 14 13 12 | 11 10 9 8 7 6 5 4 3 | 0 | 15 12 11 10 9 8 7 4 3 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | D | 1 | 0 |
| 1 | 0 | 0 | 1 | index_align | Rm |

**Offset variant**

Applies when \( Rm == 1111 \).

\( \text{VLD2}\{<c>\}{<q>}.<\text{size}>, \ [<\text{Rn}\{:<\text{align}>\}] \)
**Post-indexed variant**

Applies when \( R_m = 1101 \).

\[
VLD2\langle c\rangle\langle q\rangle.\langle size\rangle \ <\text{list}>., \ [\langle Rn\rangle\langle:\langle align\rangle\rangle]!=
\]

**Post-indexed variant**

Applies when \( R_m \neq 11x1 \).

\[
VLD2\langle c\rangle\langle q\rangle.\langle size\rangle \ <\text{list}>., \ [\langle Rn\rangle\langle:\langle align\rangle\rangle], \ <Rm>
\]

**Decode for all variants of this encoding**

\[
\text{if size} = '11' \text{ then SEE "VLD2 (single 2-element structure to all lanes)";}
\]

\[
\text{if index}_{\text{align}<1>} != '0' \text{ then UNDEFINED;}
\]

\[
e_{\text{bytes}} = 4; \quad \text{index} = \text{UInt}(\text{index}_{\text{align}<3>});
\]

\[
\text{inc} = \text{if index}_{\text{align}<2>} == '0' \text{ then 1 else 2;}
\]

\[
\text{alignment} = \text{if index}_{\text{align}<0>} == '0' \text{ then 1 else 8;}
\]

\[
d = \text{UInt}(D:V_d); \quad d2 = d + \text{inc}; \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm);
\]

\[
\text{wback} = (m != 15); \quad \text{register}_{\text{index}} = (m != 15 \&\& m != 13);
\]

\[
\text{if n} == 15 \text{ || } \text{d2} > 31 \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( d2 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *VLD2 (single 2-element structure to one lane)* on page K1-11577.

**Assembler symbols**

\(<c>\) For encoding A1, A2 and A3: see *Standard assembler syntax fields* on page F1-7228. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields* on page F1-7228.

\(<q>\) See *Standard assembler syntax fields* on page F1-7228.

\(<\text{size}>\) Is the data size, encoded in the "size" field. It can have the following values:

- \(8\) when \( \text{size} = 00\)
- \(16\) when \( \text{size} = 01\)
- \(32\) when \( \text{size} = 10\)

\(<\text{list}>\) Is a list containing the 64-bit names of the two SIMD&FP registers holding the element.

The list must be one of:

- \{ \langle Dd\rangle[\langle index\rangle], \langle Dd\rangle[\langle index\rangle+1] \} Single-spaced registers, encoded as "spacing" = 0.
- \{ \langle Dd\rangle[\langle index\rangle], \langle Dd\rangle[\langle index\rangle+2] \} Double-spaced registers, encoded as "spacing" = 1. Not permitted when \( \text{size} = 8\).

The encoding of "spacing" depends on \(<\text{size}>\).

\(<\text{size}> = 16"spacing" is encoded in the "index_{align}<1>" field.
<size> == 32"spacing" is encoded in the "index_align<2>" field.
The register <Dd> is encoded in the "D:Vd" field.
The permitted values and encoding of <index> depend on <size>:
<size> == 8<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.
<size> == 16<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.
<size> == 32<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see Unaligned data access on page E2-7193, and the encoding depends on <size>:
<size> == 8Encoded in the "index_align<0>" field as 0.
<size> == 16Encoded in the "index_align<0>" field as 0.
<size> == 32Encoded in the "index_align<1:0>" field as 0b0.
Whenever <align> is present, the permitted values and encoding depend on <size>:
<size> == 8<align> is 16, meaning 16-bit alignment, encoded in the "index_align<0>" field as 1.
<size> == 16<align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.
<size> == 32<align> is 64, meaning 64-bit alignment, encoded in the "index_align<1:0>" field as 0b1.
: is the preferred separator before the <align> value, but the alignment can be specified as 0<align>, see The Advanced SIMD addressing mode on page F1-7249.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see The Advanced SIMD addressing mode on page F1-7249.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    Elem[D[d], index] = MemU[address,ebytes];
    Elem[D[d2],index] = MemU[address+ebytes,ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.104 VLD2 (single 2-element structure to all lanes)

Load single 2-element structure and replicate to all lanes of two registers loads one 2-element structure from memory into all lanes of two registers. For details of the addressing mode, see The Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

Offset variant
Applies when \( Rm = 1111 \).

\[
\text{VLD2}{{<c>}}{<q>}.<\text{size}> <\text{list}>, [<\text{Rn}>{:<\text{align}>}]
\]

Post-indexed variant
Applies when \( Rm = 1101 \).

\[
\text{VLD2}{{<c>}}{<q>}.<\text{size}> <\text{list}>, [<\text{Rn}>{:<\text{align}>}]!
\]

Post-indexed variant
Applies when \( Rm != 11x1 \).

\[
\text{VLD2}{{<c>}}{<q>}.<\text{size}> <\text{list}>, [<\text{Rn}>{:<\text{align}>}],<\text{Rm}>
\]

Decode for all variants of this encoding

\[
\text{if size } == '11' \text{ then UNDEFINED;}
\]
\[
ebytes = 1 << \text{UInt(size)};
\]
\[
\text{alignment } = \text{if a } == '0' \text{ then 1 else 2*ebytes;}
\]
\[
\text{inc } = \text{if T } == '0' \text{ then 1 else 2;}
\]
\[
d = \text{UInt(D:Vd)}; \quad d2 = d + \text{inc}; \quad n = \text{UInt(Rn)}; \quad m = \text{UInt(Rm)};
\]
\[
\text{wbback } = (m != 15); \quad \text{register_index } = (m != 15 \&\& m != 13);
\]
\[
\text{if n } == 15 || d2 > 31 \text{ then UNPREDICTABLE;}
\]

CONstrained UNPREDICTABLE behavior

If \( d2 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1
Offset variant

Applies when \( Rm = 1111 \).

\[
\text{VLD2}\{<c>\}{<q>}.<\text{size}> \text{ <list>}, [<Rn>{{:<align>}}]
\]

Post-indexed variant

Applies when \( Rm = 1101 \).

\[
\text{VLD2}\{<c>\}{<q>}.<\text{size}> \text{ <list>}, [<Rn>{{:<align>}}]!
\]

Post-indexed variant

Applies when \( Rm \neq 11x1 \).

\[
\text{VLD2}\{<c>\}{<q>}.<\text{size}> \text{ <list>}, [<Rn>{{:<align>}}], <Rm>
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{if size} & = '11' \text{ then UNDEFINED;} \\
\text{ebytes} & = 1 \ll \text{ UInt(size);} \\
\text{alignment} & = \text{ if a} = '0' \text{ then 1 else 2\times ebytes;} \\
\text{inc} & = \text{ if T} = '0' \text{ then 1 else 2;} \\
\text{d} & = \text{ UInt(D:Vd);} \\
\text{d2} & = \text{ d + inc;} \\
\text{n} & = \text{ UInt(Rn);} \\
\text{m} & = \text{ UInt(Rm);} \\
\text{wback} & = (m \neq 15); \\
\text{register_index} & = (m \neq 15 \&\& m \neq 13); \\
\text{if n} & = 15 \&\& \text{ d2 > 31 then UNPREDICTABLE;}
\end{align*}
\]

CONSTRANDED UNPREDICTABLE behavior

If \( d2 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VLD2 (single 2-element structure to all lanes) on page K1-11578.

Assembler symbols

- For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
- For encoding T1: see Standard assembler syntax fields on page F1-7228.
- See Standard assembler syntax fields on page F1-7228.
- Is the data size, encoded in the "size" field. It can have the following values:
  - 8 when size = 00
  - 16 when size = 01
  - 32 when size = 10
  - The encoding size = 11 is reserved.
- Is a list containing the 64-bit names of two SIMD&FP registers.
The list must be one of:
{ <Dd>[], <Dd+1>[] } Single-spaced registers, encoded in the "T" field as 0.
{ <Dd>[], <Dd+2>[] } Double-spaced registers, encoded in the "T" field as 1.
The register <Dd> is encoded in the "D:Vd" field.

<Rn>  Is the general-purpose base register, encoded in the "Rn" field.

<align>  Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see Unaligned data access on page E2-7193, and is encoded in the "a" field as 0.
Whenever <align> is present, the permitted values and encoding depend on <size>:
<size> == 8<align> is 16, meaning 16-bit alignment, encoded in the "a" field as 1.
<size> == 16<align> is 32, meaning 32-bit alignment, encoded in the "a" field as 1.
<size> == 32<align> is 64, meaning 64-bit alignment, encoded in the "a" field as 1.
:  is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see The Advanced SIMD addressing mode on page F1-7249.

<Rm>  Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see The Advanced SIMD addressing mode on page F1-7249.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    constant integer esize = 8 * ebytes;
    bits(esize) element1 = MemU[address, ebytes];
    bits(esize) element2 = MemU[address+ebytes, ebytes];
    D[d] = Replicate(element1, 64 DIV esize);
    D[d2] = Replicate(element2, 64 DIV esize);
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
    
Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.105  **VLD2 (multiple 2-element structures)**

Load multiple 2-element structures to two or four registers loads multiple 2-element structures from memory into two or four registers, with de-interleaving. For more information, see Element and structure load/store instructions on page F2-7278. Every element of each register is loaded. For details of the addressing mode, see The Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 16 15 12 11 8 7 6 5 4 3 0 |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| 1 1 1 1 0 1 0 0 0 | D 1 0 | Rn | Vd | 1 0 0 x | size | align | Rm |

**Offset variant**

Applies when Rm == 1111.

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

**Post-indexed variant**

Applies when Rm == 1101.

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

**Post-indexed variant**

Applies when Rm != 11x1.

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

**Decode for all variants of this encoding**

pairs = 1;  if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
ingc = if itype == '1001' then 2 else 1;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If d2+pairs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.
A2

| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9|8|7|6|5|4|3|0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | D | 1 | 0 | Rn | Vd |

**Offset variant**
Applies when Rm == 1111.

\[ \text{VLD2}\{<c>\}{<q>}.<\text{size}> \langle\text{list}\rangle, [\langle\text{Rn}\rangle{:<\text{align}}]\]  

**Post-indexed variant**
Applies when Rm == 1101.

\[ \text{VLD2}\{<c>\}{<q>}.<\text{size}> \langle\text{list}\rangle, [\langle\text{Rn}\rangle{:<\text{align}}]\]

**Post-indexed variant**
Applies when Rm != 11x1.

\[ \text{VLD2}\{<c>\}{<q>}.<\text{size}> \langle\text{list}\rangle, [\langle\text{Rn}\rangle{:<\text{align}}]\]

**Decode for all variants of this encoding**

- pairs = 2;  inc = 2;
- if size == '11' then UNDEFINED;
- alignment = if align == '00' then 1 else 4 \(<\text{UInt}(\text{align})\);
- ebytes = 1 \(<\text{UInt}(\text{size})\);
- elements = 8 \(<\text{DIV}\ ebytes);
- d = \text{UInt}(D:Vd);  d2 = d + inc;  n = \text{UInt}(\text{Rn});  m = \text{UInt}(\text{Rm});
- \text{wback} = (m != 15);  \text{register_index} = (m != 15 &amp;&amp; m != 13);
- if n == 15 || d2+pairs > 32 then UNPREDICTABLE;

**CONSTRINED UNPREDICTABLE behavior**
If d2+pairs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>D</td>
</tr>
</tbody>
</table>

**Offset variant**
Applies when Rm == 1111.

\[ \text{VLD2}\{<c>\}{<q>}.<\text{size}> \langle\text{list}\rangle, [\langle\text{Rn}\rangle{:<\text{align}}]\]  

**Post-indexed variant**
Applies when Rm == 1101.

\[ \text{VLD2}\{<c>\}{<q>}.<\text{size}> \langle\text{list}\rangle, [\langle\text{Rn}\rangle{:<\text{align}}]\]
Post-indexed variant

Applies when $Rm \neq 11x1$.

\[
\text{VLD2}\{<c>\}{<q>}.<\text{size}> \text{ <list>}, [<Rn\{:<\text{align}>\}], <Rm>
\]

Decode for all variants of this encoding

\[
\text{pairs} = 1; \quad \text{if align} = '11' \text{ then UNDEFINED;}
\]

\[
\text{if size} = '11' \text{ then UNDEFINED;}
\]

\[
\text{inc} = \text{if itype} = '1001' \text{ then 2 else 1;}
\]

\[
\text{alignment} = \text{if align} = '00' \text{ then 1 else 4} \ll \text{UInt}(align);
\]

\[
\text{ebytes} = 1 \ll \text{UInt}(\text{size}); \quad \text{elements} = 8 \div \text{ebytes};
\]

\[
\text{d} = \text{UInt}(D:Vd); \quad \text{d2} = \text{d} + \text{inc}; \quad \text{n} = \text{UInt}(Rn); \quad \text{m} = \text{UInt}(Rm);
\]

\[
\text{wback} = (m \neq 15); \quad \text{register_index} = (m \neq 15 \&\& n \neq 15);
\]

\[
\text{if n} = 15 || \text{d2+pairs} > 32 \text{ then UNPREDICTABLE;}
\]

CONstrained UNPREDICTABLE behavior

If d2+pairs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 | 0 | 15 12 11 10 9 8 7 6 5 4 | 3 | 0 |
| 1 1 1 1 1 0 0 1 0 | D | 1 0 | Rn | Vd | 0 0 1 1 | size | align | Rm |

Offset variant

Applies when $Rm = 1111$.

\[
\text{VLD2}\{<c>\}{<q>}.<\text{size}> \text{ <list>}, [<Rn\{:<\text{align}>\}]
\]

Post-indexed variant

Applies when $Rm = 1101$.

\[
\text{VLD2}\{<c>\}{<q>}.<\text{size}> \text{ <list>}, [<Rn\{:<\text{align}>\}]
\]

Post-indexed variant

Applies when $Rm \neq 11x1$.

\[
\text{VLD2}\{<c>\}{<q>}.<\text{size}> \text{ <list>}, [<Rn\{:<\text{align}>\}], <Rm>
\]

Decode for all variants of this encoding

\[
\text{pairs} = 2; \quad \text{inc} = 2;
\]

\[
\text{if size} = '11' \text{ then UNDEFINED;}
\]

\[
\text{alignment} = \text{if align} = '00' \text{ then 1 else 4} \ll \text{UInt}(align);
\]

\[
\text{ebytes} = 1 \ll \text{UInt}(\text{size}); \quad \text{elements} = 8 \div \text{ebytes};
\]

\[
\text{d} = \text{UInt}(D:Vd); \quad \text{d2} = \text{d} + \text{inc}; \quad \text{n} = \text{UInt}(Rn); \quad \text{m} = \text{UInt}(Rm);
\]

\[
\text{wback} = (m \neq 15); \quad \text{register_index} = (m \neq 15 \&\& m \neq 15);
\]

\[
\text{if n} = 15 || \text{d2+pairs} > 32 \text{ then UNPREDICTABLE;}
\]
**CONSTRAINED UNPREDICTABLE behavior**

If \(d_2+\text{pairs} > 32\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VLD2 (multiple 2-element structures) on page K1-11577.

Related encodings: See *Advanced SIMD element or structure load/store* on page F3-7350 for the T32 instruction set, or *Advanced SIMD element or structure load/store* on page F4-7434 for the A32 instruction set.

**Assembler symbols**

- `<c>` For encoding A1 and A2: see *Standard assembler syntax fields* on page F1-7228. This encoding must be unconditional.
  
  For encoding T1 and T2: see *Standard assembler syntax fields* on page F1-7228.

- `<q>` See *Standard assembler syntax fields* on page F1-7228.

- `<size>` Is the data size, encoded in the "size" field. It can have the following values:
  
  - 8 when size = 00
  - 16 when size = 01
  - 32 when size = 10

  The encoding size = 11 is reserved.

- `<list>` Is a list containing the 64-bit names of the SIMD&FP registers.

  The list must be one of:

  - \{<Dd>, <Dd+1>\} Two single-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as 0b1000.
  - \{<Dd>, <Dd+2>\} Two double-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as 0b1001.
  - \{<Dd>, <Dd+1>, <Dd+2>, <Dd+3>\} Three single-spaced registers. Selects the A2 and T2 encodings of the instruction.

  The register <Dd> is encoded in the "D:Vd" field.

- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field.

- `<align>` Is the optional alignment.

  Whenever `<align>` is omitted, the standard alignment is used, see *Unaligned data access on page E2-7193*, and is encoded in the "align" field as 0b00.

  Whenever `<align>` is present, the permitted values are:

  - 64 64-bit alignment, encoded in the "align" field as 0b01.
  - 128 128-bit alignment, encoded in the "align" field as 0b10.
  - 256 256-bit alignment, encoded in the "align" field as 0b11. Available only if `<list>` contains four registers.

  : is the preferred separator before the `<align>` value, but the alignment can be specified as @<align>, see *The Advanced SIMD addressing mode on page F1-7249*. 
<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode* on page F1-7249.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to pairs-1
        for e = 0 to elements-1
            Elem[D[d+r], e] = MemU[address,ebytes];
            Elem[D[d2+r], e] = MemU[address+ebytes,ebytes];
            address = address + 2*ebytes;
            if wback then
                if register_index then
                    R[n] = R[n] + R[m];
                else
                    R[n] = R[n] + 16*pairs;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.106  VLD3 (single 3-element structure to one lane)

Load single 3-element structure to one lane of three registers loads one 3-element structure from memory into corresponding elements of three registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode, see The Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

Offset variant

Applies when \( Rm = 1111 \).

\[ VLD3{<c>}{<q>}.<size> \text{ <list>, } [<Rn>] \]

Post-indexed variant

Applies when \( Rm = 1101 \).

\[ VLD3{<c>}{<q>}.<size> \text{ <list>, } [<Rn>!] \]

Post-indexed variant

Applies when \( Rm \neq 11x1 \).

\[ VLD3{<c>}{<q>}.<size> \text{ <list>, } [<Rn>], <Rm> \]

Decode for all variants of this encoding

if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m \&\& m != 13);
if d3 > 31 then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If \( d3 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2
Offset variant
Applies when \( Rm = 1111 \).
\[ \text{VLD3}{\{<c>}{<q>}.<size> <list>, [<Rn>]} \]

Post-indexed variant
Applies when \( Rm = 1101 \).
\[ \text{VLD3}{\{<c>}{<q>}.<size> <list>, [<Rn>]!} \]

Post-indexed variant
Applies when \( Rm \neq 1111 \).
\[ \text{VLD3}{\{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>} \]

Decode for all variants of this encoding
\[
\begin{align*}
\text{if size} &= '11' \text{ then see } "\text{VLD3 (single 3-element structure to all lanes)}"; \\
\text{if index\_align<0>} &= '0' \text{ then UNDEFINED;} \\
ebytes &= 2; \text{ index = UInt(index\_align<3:2>);} \\
\text{inc} &= \text{if index\_align<1>} = '0' \text{ then 1 else 2;} \\
d &= \text{UInt(D:Vd)}; \text{ d2 = d + inc; d3 = d2 + inc;} \text{ n = UInt(Rn); m = UInt(Rm);} \\
wback &= (m != 15); \text{ register\_index = (m != 15 && m != 13);} \\
\text{if n} &= 15 || d3 > 31 \text{ then UNPREDICTABLE};
\end{align*}
\]

CONSTRAINED UNPREDICTABLE behavior
If \( d3 > 31 \), then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | D  | 1  | 0  | Rn | Vd | 1  | 0  | 1  | 0  | index\_align | Rm |
| size |

Offset variant
Applies when \( Rm = 1111 \).
\[ \text{VLD3}{\{<c>}{<q>}.<size> <list>, [<Rm>]} \]

Post-indexed variant
Applies when \( Rm = 1101 \).
\[ \text{VLD3}{\{<c>}{<q>}.<size> <list>, [<Rn>]!} \]

Post-indexed variant
Applies when \( Rm \neq 1111 \).
\[ \text{VLD3}{\{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>} \]
Decode for all variants of this encoding

if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

|15 14 13 12|11 10 9 8 |7 6 5 4 |3 0 |15|12|11 10 9 8 |7 4 |3 0 |
|---|---|---|---|---|---|---|---|---|---|
|1 |1 |1 |1 |0 |0 |1 |1 |Rn |Vd |
|0 |1 |0 |index_align |Rm |

Offset variant

Applies when Rm == 1111.
VLD3{<c>}{<q>}.<size> <list>, [<Rn]>

Post-indexed variant

Applies when Rm == 1101.
VLD3{<c>}{<q>}.<size> <list>, [<Rn]!)

Post-indexed variant

Applies when Rm != 11x1.
VLD3{<c>}{<q>}.<size> <list>, [<Rn]!, <Rm>

Decode for all variants of this encoding

if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>);
inc = 1;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are $\texttt{UNKNOWN}$. If the instruction specifies writeback, the base register becomes $\texttt{UNKNOWN}$. This behavior does not affect any general-purpose registers.

### T2

- **Offset variant**
  - Applies when $\texttt{Rm} == 1111$.
  - $\texttt{VLD3}\{<c>\}{<q>}.<\texttt{size}> \texttt{<list>}, [<Rn>]$

- **Post-indexed variant**
  - Applies when $\texttt{Rm} == 1101$.
  - $\texttt{VLD3}\{<c>\}{<q>}.<\texttt{size}> \texttt{<list>}, [<Rn>]!$

- **Post-indexed variant**
  - Applies when $\texttt{Rm} != 11x1$.
  - $\texttt{VLD3}\{<c>\}{<q>}.<\texttt{size}> \texttt{<list>}, [<Rn>], <Rm>$

#### Decode for all variants of this encoding

```plaintext
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)"
if index_align<0> != '0' then UNDEFINED;
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If $d3 > 31$, then one of the following behaviors must occur:

- The instruction is $\texttt{UNDEFINED}$.
- The instruction executes as $\texttt{NOP}$.
- One or more of the SIMD and floating-point registers are $\texttt{UNKNOWN}$. If the instruction specifies writeback, the base register becomes $\texttt{UNKNOWN}$. This behavior does not affect any general-purpose registers.

### T3

- **Offset variant**
  - Applies when $\texttt{Rm} == 1111$.
  - $\texttt{VLD3}\{<c>\}{<q>}.<\texttt{size}> \texttt{<list>}, [<Rn>]$
**Post-indexed variant**

Applies when \( Rm = 1101 \).

\[
VLD3\{<c>\}\{<q>\}.<size>\ <list>, [\{<Rn>\}]
\]

**Post-indexed variant**

Applies when \( Rm \neq 11x1 \).

\[
VLD3\{<c>\}\{<q>\}.<size>\ <list>, [\{<Rn>\}], \langle Rm \rangle
\]

**Decode for all variants of this encoding**

if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)"";
if index_align<1:0> != '00' then UNDEFINED;
  ebytes = 4;  index = UInt(index_align<3>);
in1 = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  \( d2 = d + inc \);  \( d3 = d2 + inc \);  \( n = UInt(Rn) \);  \( m = UInt(Rm) \);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \( d3 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as \textit{NOP}.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VLD3 (single 3-element structure to one lane) on page K1-11578.

**Assembler symbols**

\(<c>\) For encoding A1, A2 and A3: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1, T2 and T3: see Standard assembler syntax fields on page F1-7228.

\(<q>\) See Standard assembler syntax fields on page F1-7228.

\(<size>\) Is the data size, encoded in the "size" field. It can have the following values:

- \( 8 \) when size = 00
- \( 16 \) when size = 01
- \( 32 \) when size = 10

\(<list>\) Is a list containing the 64-bit names of the three SIMD&FP registers holding the element.

The list must be one of:

\{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>] } Single-spaced registers, encoded as "spacing" = 0.

\{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>] } Double-spaced registers, encoded as "spacing" = 1. Not permitted when \(<size> == 8\).
The encoding of "spacing" depends on `<size>`:
- `<size> == 8"spacing" is encoded in the "index_align<0>" field.
- `<size> == 16"spacing" is encoded in the "index_align<1>" field, and "index_align<0>" is set to 0.
- `<size> == 32"spacing" is encoded in the "index_align<2>" field, and "index_align<1:0>" is set to 0b00.

The register `<Dd>` is encoded in the "D:Vd" field.

The permitted values and encoding of `<index>` depend on `<size>`:
- `<size> == 8`<index>` is in the range 0 to 7, encoded in the "index_align<3:1>" field.
- `<size> == 16`<index>` is in the range 0 to 3, encoded in the "index_align<3:2>" field.
- `<size> == 32`<index>` is 0 or 1, encoded in the "index_align<3>" field.

`<Rn>`
- Is the general-purpose base register, encoded in the "Rn" field.

`<Rm>`
- Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode on page F1-7249.*

Alignment

Standard alignment rules apply, see *Alignment support on page B2-189.*

**Operation for all encodings**

if `ConditionPassed()` then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  address = R[n];
  Elem[D[d], index] = MemU[address,ebytes];
  Elem[D[d2], index] = MemU[address+ebytes,ebytes];
  Elem[D[d3], index] = MemU[address+2*ebytes,ebytes];
  if wback then
    if register_index then
      R[n] = R[n] + R[m];
    else
      R[n] = R[n] + 3*ebytes;
  endif
endif

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.107  VLD3 (single 3-element structure to all lanes)

Load single 3-element structure and replicate to all lanes of three registers. For details of the addressing mode, see The Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8|7 6 5 4|3 0 |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | Rn | Vd | 1 | 1 | 0 | size | T | 0 | Rm |

**Offset variant**

Applies when \( Rm = 1111 \).

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

**Post-indexed variant**

Applies when \( Rm = 1101 \).

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

**Post-indexed variant**

Applies when \( Rm != 11x1 \).

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

**Decode for all variants of this encoding**

\[
\text{if size} = '11' || a = '1' \text{ then UNDEFINED;}
\]

\[
ebytes = 1 << \text{UInt(size);}
\]

\[
\text{inc} = \text{if T} = '0' \text{ then 1 else 2;}
\]

\[
d = \text{UInt(D:Vd)} ;
\]

\[
d2 = d + \text{inc};
\]

\[
d3 = d2 + \text{inc};
\]

\[
n = \text{UInt(Rn)} ;
\]

\[
m = \text{UInt(Rm)};
\]

\[
wback = (m != 15); \text{ register_index} = (m != 15 && m != 13);
\]

\[
\text{if n} = 15 \text{ || d3} > 31 \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( d3 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

[52x758]T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions

F6.1 Alphabetical list of Advanced SIMD and floating-point instructions

F6-8456  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  ARM DDI 0487I.a

Non-Confidential  ID081822
Offset variant

Applies when \( Rm == 1111 \).

\[ \text{VLD3}\{<c}\}{<q>\}.<\text{size}> <\text{list}>, [<Rn>] \]

Post-indexed variant

Applies when \( Rm == 1101 \).

\[ \text{VLD3}\{<c}\}{<q>\}.<\text{size}> <\text{list}>, [<Rn>]! \]

Post-indexed variant

Applies when \( Rm != 11x1 \).

\[ \text{VLD3}\{<c}\}{<q>\}.<\text{size}> <\text{list}>, [<Rn>], <Rm> \]

Decode for all variants of this encoding

\[
\begin{align*}
\text{if size} &= \ '11' || a == \ '1' \text{ then UNDEFINED;} \\
\text{ebytes} &= 1 <\text{ Uint}(\text{size}); \\
ninc &= \text{ if } T == \ '0' \text{ then } 1 \text{ else } 2; \\
d &= \text{ Uint}(D:Vd); \\
d2 &= d + ninc; \\
d3 &= d2 + ninc; \\
n &= \text{ Uint}(Rn); \\
m &= \text{ Uint}(Rm); \\
wback &= (m != 15); \\
\text{register_index} &= (m != 15 && m != 13); \\
\text{if } n == 15 \text{ || } d3 > 31 \text{ then UNPREDICTABLE;}
\end{align*}
\]

CONstrained unPredictable behavior

If \( d3 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONstrained unPredICtable behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly \( \text{VLD3 (single 3-element structure to all lanes)} \) on page K1-11578.

Assembler symbols

\(<c>\)  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

\(<q>\)  See Standard assembler syntax fields on page F1-7228.

\(<\text{size}>\)  Is the data size, encoded in the "size" field. It can have the following values:

\[
\begin{align*}
8 & \text{ when size} = 00 \\
16 & \text{ when size} = 01 \\
32 & \text{ when size} = 10 \\
\end{align*}
\]

The encoding size = 11 is reserved.

\(<\text{list}>\)  Is a list containing the 64-bit names of three SIMD&FP registers.

The list must be one of:

\[
\{ <\text{Dd}>[], <\text{Dd+1}>[], <\text{Dd+2}>[] \} \text{Single-spaced registers, encoded in the "T" field as 0.}
\]
{ <Dd>[], <Dd+2>[], <Dd+4>[] } Double-spaced registers, encoded in the "T" field as 1.
The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *The Advanced SIMD addressing mode on page F1-7249.*

Alignment

Standard alignment rules apply, see *Alignment support on page B2-189.*

**Operation for all encodings**

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  address = R[n];
  constant integer esize = ebytes * 8;
  bits(esize) element1 = MemU[address, ebytes];
  bits(esize) element2 = MemU[address+ebytes, ebytes];
  bits(esize) element3 = MemU[address+2*ebytes, ebytes];

  D[d] = Replicate(element1, 64 DIV esize);
  D[d2] = Replicate(element2, 64 DIV esize);
  D[d3] = Replicate(element3, 64 DIV esize);
  if wback then
    if register_index then
      R[n] = R[n] + R[m];
    else
      R[n] = R[n] + 3*ebytes;

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.108 VLD3 (multiple 3-element structures)

Load multiple 3-element structures to three registers loads multiple 3-element structures from memory into three registers, with de-interleaving. For more information, see Element and structure load/store instructions on page F2-7278. Every element of each register is loaded. For details of the addressing mode, see The Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPRR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 12</th>
<th>11 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 0 0 0</td>
<td>D</td>
<td>1 0</td>
<td>Rn</td>
<td>Vd</td>
<td>0 1 0</td>
<td>x</td>
<td>size</td>
</tr>
</tbody>
</table>

Offset variant

Applies when Rm == 1111.

VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]<Rm>

Post-indexed variant

Applies when Rm == 1101.

VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed variant

Applies when Rm != 11x1.

VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

Decode for all variants of this encoding

integer inc;
case itype of
  when '0100'
    inc = 1;
  when '0101'
    inc = 2;
  otherwise
    SEE "Related encodings";
if size == '11' || align<1> == '1' then UNDEFINED;
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.
Offset variant
Applies when \( Rm == 1111 \).

\[ \text{VLD3\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align>}] \} \]

Post-indexed variant
Applies when \( Rm == 1101 \).

\[ \text{VLD3\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align>}]!} \]

Post-indexed variant
Applies when \( Rm != 11x1 \).

\[ \text{VLD3\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm} \]

Decode for all variants of this encoding

```c
integer inc;
case itype of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
if size == '11' || align<1> == '1' then UNDEFINED;
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRANDED UNPREDICTABLE behavior

If \( d3 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VLD3 (multiple 3-element structures) on page K1-11578.

Related encodings: See Advanced SIMD element or structure load/store on page F3-7350 for the T32 instruction set, or Advanced SIMD element or structure load/store on page F4-7434 for the A32 instruction set.
Assembler symbols

&lt;c&gt; For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
   For encoding T1: see Standard assembler syntax fields on page F1-7228.

&lt;q&gt; See Standard assembler syntax fields on page F1-7228.

&lt;size&gt; Is the data size, encoded in the "size" field. It can have the following values:
   8 when size = 00
   16 when size = 01
   32 when size = 10
   The encoding size = 11 is reserved.

&lt;list&gt; Is a list containing the 64-bit names of the SIMD&FP registers.
   The list must be one of:
   { <Dd>, <Dd+1>, <Dd+2> }Single-spaced registers, encoded in the "itype" field as 0b0100.
   { <Dd>, <Dd+2>, <Dd+4> }Double-spaced registers, encoded in the "itype" field as 0b0101.
   The register &lt;Dd&gt; is encoded in the "D:Vd" field.

&lt;Rn&gt; Is the general-purpose base register, encoded in the "Rn" field.

&lt;align&gt; Is the optional alignment.
   Whenever &lt;align&gt; is omitted, the standard alignment is used, see Unaligned data access on page E2-7193, and is encoded in the "align" field as 0000.
   Whenever &lt;align&gt; is present, the only permitted values is 64, meaning 64-bit alignment, encoded in the "align" field as 0b01.
   : is the preferred separator before the &lt;align&gt; value, but the alignment can be specified as @&lt;align&gt;, see The Advanced SIMD addressing mode on page F1-7249.

&lt;Rm&gt; Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about &lt;Rn&gt;, !, and &lt;Rm&gt;, see The Advanced SIMD addressing mode on page F1-7249.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for e = 0 to elements-1
        Elem[D[d], e] = MemU[address,ebytes];
        Elem[D[d2],e] = MemU[address+ebytes,ebytes];
        Elem[D[d3],e] = MemU[address+2*ebytes,ebytes];
        address = address + 3*ebytes;
        if wback then
            if register_index then
                R[n] = R[n] + R[m];
            else
                R[n] = R[n] + 24;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.109 VLD4 (single 4-element structure to one lane)

Load single 4-element structure to one lane of four registers loads one 4-element structure from memory into corresponding elements of four registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode, see The Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19</th>
<th>16 15 12 11 10 9 8 7 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 0 0 1</td>
<td>D 1 0</td>
</tr>
</tbody>
</table>

Offset variant

Applies when Rm == 1111.

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}

Post-indexed variant

Applies when Rm == 1101.

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed variant

Applies when Rm != 11x1.

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

Decode for all variants of this encoding

if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";

ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);

wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19</th>
<th>16 15 12 11 10 9 8 7 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 0 0 1</td>
<td>D 1 0</td>
</tr>
</tbody>
</table>
Offset variant
Applies when \( Rm = 1111 \).
\[
\text{VLD4}\{<c>\}{<q>}.<size> \ <list>, \ [<Rn}>{:<align}>]
\]

Post-indexed variant
Applies when \( Rm = 1101 \).
\[
\text{VLD4}\{<c>\}{<q>}.<size> \ <list>, \ [<Rn}>{:<align}>]!
\]

Post-indexed variant
Applies when \( Rm \neq 11x1 \).
\[
\text{VLD4}\{<c>\}{<q>}.<size> \ <list>, \ [<Rn}>{:<align}>], \ <Rm>
\]

Decode for all variants of this encoding
\[
\begin{align*}
\text{if size} &= \text{7} \text{ then see } \text{"VLD4 (single 4-element structure to all lanes)";} \smallskip \\
\text{ebytes} &= 2; \ \text{index} = \text{UInt(index_align3:2);} \smallskip \\
\text{inc} &= \text{if index_align1} = \text{0' then 1 else 2;} \smallskip \\
\text{alignment} &= \text{if index_align0} = \text{0' then 1 else 8;} \smallskip \\
\text{d} &= \text{UInt(D:Vd);} \ \text{d2} = \text{d + inc;} \ \text{d3} = \text{d2 + inc;} \ \text{d4} = \text{d3 + inc;} \ \text{n} = \text{UInt(Rn);} \ \text{m} = \text{UInt(Rm)}; \smallskip \\
\text{wback} &= (m \neq 15); \ \text{register_index} = (m \neq 15 \&\& m \neq 13); \smallskip \\
\text{if n} &= \text{15 || d4 > 31 then UNPREDICTABLE;}
\end{align*}
\]

CONSTRAINED UNPREDICTABLE behavior
If \( d4 \ > \ 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3

\[
\begin{array}{ccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & D & 1 & 0 & Rn & Vd & 1 & 0 & 1 & 1 & \text{index_align} & Rm \\
\text{size}
\end{array}
\]

Offset variant
Applies when \( Rm = 1111 \).
\[
\text{VLD4}\{<c>\}{<q>}.<size> \ <list>, \ [<Rm}>{:<align}>]
\]

Post-indexed variant
Applies when \( Rm = 1101 \).
\[
\text{VLD4}\{<c>\}{<q>}.<size> \ <list>, \ [<Rm}>{:<align}>]!
\]

Post-indexed variant
Applies when \( Rm \neq 11x1 \).
\[
\text{VLD4}\{<c>\}{<q>}.<size> \ <list>, \ [<Rm}>{:<align}>], \ <Rm>
\]
Decode for all variants of this encoding

if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4
<< UInt(index_align<1:0>);
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn);
m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

Offset variant

Applies when Rm == 1111.
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]

Post-indexed variant

Applies when Rm == 1101.
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed variant

Applies when Rm != 11x1.
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

Decode for all variants of this encoding

if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
ebytes = 1; index = UInt(index_align<3:1>);
inc = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn);
m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes asNOP.
• One or more of the SIMD and floating-point registers are \texttt{UNKNOWN}. If the instruction specifies writeback, the base register becomes \texttt{UNKNOWN}. This behavior does not affect any general-purpose registers.

### T2

![Instruction Format](image)

**Offset variant**

Applies when \( Rm = 1111 \).

\[
\text{VLD4}\{<c>\}{<q>}.<\text{size}> <\text{list}>, \left[<Rn>\{:<\text{align}>\}\right]
\]

**Post-indexed variant**

Applies when \( Rm = 1101 \).

\[
\text{VLD4}\{<c>\}{<q>}.<\text{size}> <\text{list}>, \left[<Rn>\{:<\text{align}>\}\right]!
\]

**Post-indexed variant**

Applies when \( Rm \neq 11x1 \).

\[
\text{VLD4}\{<c>\}{<q>}.<\text{size}> <\text{list}>, \left[<Rn>\{:<\text{align}>\}\right], <Rm>
\]

**Decode for all variants of this encoding**

- if \( \text{size} = \text{'11'} \) then SEE "VLD4 (single 4-element structure to all lanes)"
- \( \text{ebytes} = 2; \text{ index} = \text{UInt(index_align<3:2>)} \)
- \( \text{inc} = \text{if index_align<1> == '0' then 1 else 2} \)
- \( \text{alignment} = \text{if index_align<0> == '0' then 1 else 8} \)
- \( d = \text{UInt(D:Vd)}; d2 = d + \text{inc}; d3 = d2 + \text{inc}; d4 = d3 + \text{inc}; n = \text{UInt(Rn)}; m = \text{UInt(Rm)}; wback = (m != 15); \text{register_index} = (m != 15 && m != 13); \)
- \( \text{if n == 15 || d4 > 31 then UNPREDICTABLE;} \)

**CONSTRAINED UNPREDICTABLE behavior**

If \( d4 > 31 \), then one of the following behaviors must occur:

- The instruction is \texttt{UNDEFINED}.
- The instruction executes as \texttt{NOP}.
- One or more of the SIMD and floating-point registers are \texttt{UNKNOWN}. If the instruction specifies writeback, the base register becomes \texttt{UNKNOWN}. This behavior does not affect any general-purpose registers.

### T3

![Instruction Format](image)

**Offset variant**

Applies when \( Rm = 1111 \).

\[
\text{VLD4}\{<c>\}{<q>}.<\text{size}> <\text{list}>, \left[<Rn>\{:<\text{align}>\}\right]
\]
Post-indexed variant
Applies when Rm == 1101.
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>]!

Post-indexed variant
Applies when Rm != 11x1.
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align}>], <Rm>

Decode for all variants of this encoding
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
ingc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wbac = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior
If d4 > 31, then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as NOP.
• One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VLD4 (single 4-element structure to one lane) on page K1-11578.

Assembler symbols
<<
For encoding A1, A2 and A3: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1, T2 and T3: see Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

<size>
Is the data size, encoded in the "size" field. It can have the following values:
8 when size = 00
16 when size = 01
32 when size = 10

<List>
Is a list containing the 64-bit names of the four SIMD&FP registers holding the element.
The list must be one of:
{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>] } Single-spaced registers, encoded as "spacing" = 0.
{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>] } Double-spaced registers, encoded as "spacing" = 1. Not permitted when size == 8.
The encoding of "spacing" depends on <size>:
- <size> == 16: "spacing" is encoded in the "index_align<1>" field.
- <size> == 32: "spacing" is encoded in the "index_align<2>" field.

The register <Dd> is encoded in the "D:Vd" field.

The permitted values and encoding of <index> depend on <size>:
- <size> == 8: <index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.
- <size> == 16: <index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.
- <size> == 32: <index> is 0 or 1, encoded in the "index_align<3>" field.

<rn> is the general-purpose base register, encoded in the "Rn" field.
<align> is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see Unaligned data access on page E2-7193, and the encoding depends on <size>:
- <size> == 8: <align> is encoded in the "index_align<0>" field as 0.
- <size> == 16: <align> is encoded in the "index_align<0>" field as 0b00.
- <size> == 32: <align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 0b0.

Whenever <align> is present, the permitted values and encoding depend on <size>:
- <size> == 8: <align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.
- <size> == 16: <align> is 64, meaning 64-bit alignment, encoded in the "index_align<0>" field as 1.
- <size> == 32: <align> can be 64 or 128. 64-bit alignment is encoded in the "index_align<1:0>" field as 0b01, and 128-bit alignment is encoded in the "index_align<1:0>" field as 0b10.

<rm> is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see The Advanced SIMD addressing mode on page F1-7249.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    Elem[D[d], index] = MemU[address, ebytes];
    Elem[D[d2], index] = MemU[address+ebytes, ebytes];
    Elem[D[d3], index] = MemU[address+2*ebytes, ebytes];
    Elem[D[d4], index] = MemU[address+3*ebytes, ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.110  VLD4 (single 4-element structure to all lanes)

Load single 4-element structure and replicate to all lanes of four registers loads one 4-element structure from memory into all lanes of four registers. For details of the addressing mode, see The Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | D  | 1  | 0  | Rn | Vd | 1  | 1  | 1  | 1  | size | T  | a  | Rm |
```

**Offset variant**

Applies when Rm == 1111.

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

**Post-indexed variant**

Applies when Rm == 1101.

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

**Post-indexed variant**

Applies when Rm != 11x1.

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}],<Rm>

**Decode for all variants of this encoding**

if size == '11' & & a == '0' then UNDEFINED;

integer ebytes;

integer alignment;

if size == '11' then
ebytes = 4;  alignment = 16;
else
ebytes = 1 << UInt(size);
if size == '10' then
alignment = if a == '0' then 1 else 8;
else
alignment = if a == '0' then 1 else 4*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);

wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.
T1

| 15 14 13 12| 11 10 9 | 8 | 7 6 5 4 | 3 | 0 8 | 12 | 11 10 9 | 8 | 7 6 5 4 | 3 | 0 |
|----------|--------|---|---------|---|------|--------|--------|---|---------|---|--------|---|------|
| 1 1 1 1 0 0 1 1 | D | 1 | 0 | Rn | Vd | 1 | 1 | 1 | size | T | a | Rm |

Offset variant
Applies when Rm == 1111.

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed variant
Applies when Rm == 1101.

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed variant
Applies when Rm != 11x1.

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

\[
\text{if size == '11' \&\& a == '0' then UNDEFINED;}
\]
\[
\text{integer ebytes;}
\]
\[
\text{integer alignment;}
\]
\[
\text{if size == '11' then}
\]
\[
\text{ebytes = 4; alignment = 16;}
\]
\[
\text{else}
\]
\[
\text{ebytes = 1 << UInt(size);}
\]
\[
\text{if size == '10' then}
\]
\[
\text{alignment = if a == '0' then 1 else 8;}
\]
\[
\text{else}
\]
\[
\text{alignment = if a == '0' then 1 else 4\times ebytes;}
\]
\[
\text{inc = if T == '0' then 1 else 2;}
\]
\[
\text{d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);}
\]
\[
\text{wback = (m != 15); register_index = (m != 15 \&\& m != 13);}
\]
\[
\text{if n == 15 || d4 > 31 then UNPREDICTABLE;}
\]

CONSTRANDED UNPREDICTABLE behavior
If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings
For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VLD4 (single 4-element structure to all lanes) on page K1-11579.
Assembler symbols

<c>
For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

<size>
Is the data size, encoded in the "size" field. It can have the following values:

- 8 when size = 00
- 16 when size = 01
- 32 when size = 1x

<List>
Is a list containing the 64-bit names of four SIMD&FP registers.
The list must be one of:

- { <Dd>[], <Dd+1>[], <Dd+2>[], <Dd+3>[]} Single-spaced registers, encoded in the "T" field as 0.
- { <Dd>[], <Dd+2>[], <Dd+4>[], <Dd+6>[]} Double-spaced registers, encoded in the "T" field as 1.
The register <Dd> is encoded in the "D:Vd" field.

<Rn>
Is the general-purpose base register, encoded in the "Rn" field.

<align>
Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see Unaligned data access on page E2-7193, and is encoded in the "a" field as 0.
Whenever <align> is present, the permitted values and encoding depend on <size>:

- <size> == 8<align> is 32, meaning 32-bit alignment, encoded in the "a" field as 1.
- <size> == 16<align> is 64, meaning 64-bit alignment, encoded in the "a" field as 1.
- <size> == 32<align> can be 64 or 128. 64-bit alignment is encoded in the "a:size<0>" field as 0b10, and 128-bit alignment is encoded in the "a:size<0>" field as 0b11.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see The Advanced SIMD addressing mode on page F1-7249.

<Rm>
Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see The Advanced SIMD addressing mode on page F1-7249.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    constant integer esize = ebytes * 8;
    bits(esize) element1 = MemU[address, ebytes];
    bits(esize) element2 = MemU[address+ebytes,ebytes];
    bits(esize) element3 = MemU[address+2*ebytes,ebytes];
    bits(esize) element4 = MemU[address+3*ebytes,ebytes];
    D[d] = Replicate(element1, 64 DIV esize);
    D[d2] = Replicate(element2, 64 DIV esize);
    D[d3] = Replicate(element3, 64 DIV esize);
    D[d4] = Replicate(element4, 64 DIV esize);
if wback then
    if register_index then
        R[n] = R[n] + R[m];
    else
        R[n] = R[n] + 4*ebytes;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.111  VLD4 (multiple 4-element structures)

Load multiple 4-element structures to four registers loads multiple 4-element structures from memory into four registers, with de-interleaving. For more information, see Element and structure load/store instructions on page F2-7278. Every element of each register is loaded. For details of the addressing mode, see The Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

Offset variant
Applies when \(Rm == 1111\).

\[\text{VLD4}\{<c>\}{<q>}.<\text{size}> \text{ <list>}, \{<\text{Rn}>{:<\text{align}>}\}\]

Post-indexed variant
Applies when \(Rm == 1101\).

\[\text{VLD4}\{<c>\}{<q>}.<\text{size}> \text{ <list>}, \{<\text{Rn}>{:<\text{align}>}\}!\]

Post-indexed variant
Applies when \(Rm != 11x1\).

\[\text{VLD4}\{<c>\}{<q>}.<\text{size}> \text{ <list>}, \{<\text{Rn}>{:<\text{align}>}\}, Rm\]

Decode for all variants of this encoding

integer inc;
case itype of
  when '0000'
    inc = 1;
  when '0001'
    inc = 2;
  otherwise
    \text{SEE "Related encodings";}
if size == '11' then UNDEFINED;
alignment = if align == '00' then 4 << UInt(align);
ebytes = 1 << UInt(size);
elements = 8 DIV ebytes;
d = UInt(D:Vd);
d2 = d + inc;
d3 = d2 + inc;
d4 = d3 + inc;
n = UInt(Rn);
m = UInt(Rm);
wback = (m != 15);
register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If \(d4 > 31\), then one of the following behaviors must occur:

\- The instruction is UNDEFINED.
\- The instruction executes as NOP.
\- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.
Offset variant
Applies when \( Rm = 1111 \).
\[
\text{VLD4}\{<c>\}{<q>}\cdot<size>\ <\text{list}>, \ [<Rn}\{:<align>\}]
\]

Post-indexed variant
Applies when \( Rm = 1101 \).
\[
\text{VLD4}\{<c>\}{<q>}\cdot<size>\ <\text{list}>, \ [<Rn}\{:<align>\}]!
\]

Post-indexed variant
Applies when \( Rm \neq 11x1 \).
\[
\text{VLD4}\{<c>\}{<q>}\cdot<size>\ <\text{list}>, \ [<Rn}\{:<align>\}], \ <Rm>
\]

Decode for all variants of this encoding

```c
int inc;
case itype of
  when '0000'
    inc = 1;
  when '0001'
    inc = 2;
  otherwise
    \text{SEE "Related encodings";}
  if size == '11' then UNDEFINED;
  alignment = if align == '00' then 1 else 4 << \text{UInt}(align);
  ebytes = 1 << \text{UInt}(size); \text{elements} = 8 \text{DIV} ebytes;
  d = \text{UInt}(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = \text{UInt}(Rn); m = \text{UInt}(Rm);
  wback = (m != 15);
  register_index = (m != 15 && m != 13);
  if n == 15 || d4 > 31 then UNPREDICTABLE;
```

**CONSTRAINED UNPREDICTABLE behavior**

If \( d4 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VLD4 (multiple 4-element structures) on page K1-11578.

Related encodings: See Advanced SIMD element or structure load/store on page F3-7350 for the T32 instruction set, or Advanced SIMD element or structure load/store on page F4-7434 for the A32 instruction set.
Assembler symbols

<\>  For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.
   For encoding T1: see *Standard assembler syntax fields on page F1-7228*.

<\>
   See *Standard assembler syntax fields on page F1-7228*.

<size>
   Is the data size, encoded in the "size" field. It can have the following values:
   8    when size = 00
   16   when size = 01
   32   when size = 10
   The encoding size = 11 is reserved.

/list
   Is a list containing the 64-bit names of the SIMD&FP registers.
   The list must be one of:
   \{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3>\} Single-spaced registers, encoded in the "itype" field as 0b0000.
   \{ <Dd>, <Dd+2>, <Dd+4>, <Dd+6>\} Double-spaced registers, encoded in the "itype" field as 0b0001.
   The register <Dd> is encoded in the "D:Vd" field.

<Rn>
   Is the general-purpose base register, encoded in the "Rn" field.

<align>
   Is the optional alignment.
   Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access on page E2-7193*, and is encoded in the "align" field as 0000.
   Whenever <align> is present, the permitted values are:
   64   64-bit alignment, encoded in the "align" field as 0b01.
   128  128-bit alignment, encoded in the "align" field as 0b10.
   256  256-bit alignment, encoded in the "align" field as 0b11.
   ; is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *The Advanced SIMD addressing mode on page F1-7249*.

<Rm>
   Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.
   For more information about the variants of this instruction, see *The Advanced SIMD addressing mode on page F1-7249*.

Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations(); CheckAdvSIMDEnabled();
   address = R[n]; iswrite = FALSE;
   - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
   for e = 0 to elements-1
      Elem[D[d], e] = MemU[address,ebytes];
      Elem[D[d+2], e] = MemU[address+ebytes,ebytes];
      Elem[D[d+4], e] = MemU[address+2*ebytes,ebytes];
      Elem[D[d+6], e] = MemU[address+3*ebytes,ebytes];
      address = address + 4*ebytes;
   if wback then
      if register_index then
         R[n] = R[n] + R[m];
      else
         R[n] = R[n] + 32;
Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.112   VLDM, VLDMDB, VLMIA

Load Multiple SIMD&FP registers loads multiple registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register.

 Depending on settings in the CPACR, NSACR, and HCPTF registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

This instruction is used by the alias VPOP. See Alias conditions on page F6-8479 for details of when each alias is preferred.

A1

<table>
<thead>
<tr>
<th>[31]</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=1111</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>P</td>
<td>U</td>
<td>D</td>
<td>W</td>
<td>1</td>
<td>Rn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>imm8&lt;7:1&gt;</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\textit{Decrement Before variant}

Applies when P == 1 && U == 0 && W == 1.

VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>

\textit{Increment After variant}

Applies when P == 0 && U == 1.

VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

\textit{Decode for all variants of this encoding}

if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && U == '0' then SEE "VLDR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE; add = (U == '1'); wback = (W == '1');
d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8<0:0>, 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 16 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;

\textbf{CONSTRANDED UNPREDICTABLE behavior}

If regs == 0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If regs > 16 || (d+regs) > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.
A2

\[
\begin{array}{ccccccccccc}
1=1111 & 1 & 1 & 0 & P & U & D & W & 1 & Rn & Vd & 1 & 0 & 1 & 0 & \text{imm8} \\
\end{array}
\]

cond

Decrement Before variant
Applies when \( P == 1 \&\& U == 0 \&\& W == 1 \).

\[
\text{VLDMDB}\{<c>\}{<q>\}{.<size>\} <Rn>!, \text{<sreglist>}
\]

Increment After variant
Applies when \( P == 0 \&\& U == 1 \).

\[
\text{VLDM}\{<c>\}{<q>\}{.<size>\} <Rn>\{!\}, \text{<sreglist>}
\]

\[
\text{VLDMIA}\{<c>\}{<q>\}{.<size>\} <Rn>\{!\}, \text{<sreglist>}
\]

Decode for all variants of this encoding
\[
\begin{align*}
\text{if } P & \text{ == '0' } \&\& U \text{ == '0' } \&\& W \text{ == '0' then SEE "Related encodings";} \\
\text{if } P & \text{ == '1' } \&\& W \text{ == '0' then SEE "VLDR";} \\
\text{if } P & \text{ == U } \&\& W \text{ == '1' then UNDEFINED;} \\
\end{align*}
\]

// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE; \text{ add} = (U == '1'); wback = (W == '1'); d = UInt(Vd:D); n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32); \text{regs} = UInt(imm8);
\[
\text{if } n \text{ == 15 } \&\& \text{(wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;}
\]

if \( \text{regs} == 0 \&\& (d+\text{regs}) > 32 \) then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior
If \( \text{regs} == 0 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If \( (d+\text{regs}) > 32 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1

\[
\begin{array}{ccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 12 & 11 & 10 & 9 & 8 & 7 & 1 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 0 & P & U & D & W & 1 & Rn & Vd & 1 & 0 & 1 & 1 & \text{imm8<7:1>} & 0 \\
\end{array}
\]

imm8<0>

Decrement Before variant
Applies when \( P == 1 \&\& U == 0 \&\& W == 1 \).

\[
\text{VLDMDB}\{<c>\}{<q>\}{.<size>\} <Rn>!, \text{<sreglist>}
\]
Increment After variant

Applies when \( P = 0 \) \&\& \( U = 1 \).

\[
\text{VLDM}{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
\]
\[
\text{VLDMIA}{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
\]

Decode for all variants of this encoding

if \( P = '0' \) \&\& \( U = '0' \) \&\& \( W = '0' \) then SEE "Related encodings";
if \( P = '1' \) \&\& \( W = '0' \) then SEE "VLDR";
if \( P = U \) \&\& \( W = '1' \) then UNDEFINED;

// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)

\[
single\_regs = \text{FALSE}; \quad \text{add} = (U == '1'); \quad \text{wback} = (W == '1');
\]
\[
d = \text{UInt}(D:Vd); \quad n = \text{UInt}(Rn); \quad \text{imm32} = \text{ZeroExtend}(\text{imm8}:'00', 32);
\]
\[
\text{regs} = \text{UInt}(\text{imm8}) \text{ DIV 2}; \quad \text{If UInt(imm8) is odd, see "FLDM*X".}
\]
if \( n == 15 \) \&\& \( \text{CurrentInstrSet() != InstrSet_A32} \) then UNPREDICTABLE;
if \( \text{regs} == 0 \) || \( \text{regs} > 16 \) || (d+regs) > 32 then UNPREDICTABLE;
if \( \text{imm8<0> == '1'} \) \&\& (d+regs) > 16 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If \( \text{regs} == 0 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If \( \text{regs} > 16 \) || (d+regs) > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2

\[
\begin{array}{ccccccccccccccccc}
11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 12 & 11 & 10 & 9 & 8 & 7 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 0 & P & U & D & W & 1 & Rn & Vd & 1 & 0 & 1 & 0 & \text{imm8}
\end{array}
\]

Decrement Before variant

Applies when \( P = 1 \) \&\& \( U = 0 \) \&\& \( W = 1 \).

\[
\text{VLDMDB}{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
\]

Increment After variant

Applies when \( P = 0 \) \&\& \( U = 1 \).

\[
\text{VLDM}{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
\]
\[
\text{VLDMIA}{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
\]

Decode for all variants of this encoding

if \( P == '0' \) \&\& \( U == '0' \) \&\& \( W == '0' \) then SEE "Related encodings";
if \( P == '1' \) \&\& \( W == '0' \) then SEE "VLDR";
if \( P == U \) \&\& \( W == '1' \) then UNDEFINED;

// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If regs == 0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If (d+regs) > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VLDM on page K1-11579.

Related encodings: See Advanced SIMD and floating-point 64-bit move on page F3-7324 for the T32 instruction set, or Advanced SIMD and floating-point 64-bit move on page F4-7410 for the A32 instruction set.

**Alias conditions**

<table>
<thead>
<tr>
<th>Alias is preferred when</th>
<th>VPOP P == '0' &amp;&amp; U == '1' &amp;&amp; W == '1' &amp;&amp; Rn == '1101'</th>
</tr>
</thead>
</table>

**Assembler symbols**

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<size>` An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.
- `<Rn>` Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.
- `!` Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.
- `<sreglist>` Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.
- `<dreglist>` Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    address = if add then R[n] else R[n].imm32;
    for r = 0 to regs-1
        if single_regs then
            S[d+r] = MemA[address,4];  address = address+4;
        else
            word1 = MemA[address,4];  word2 = MemA[address+4,4];  address = address+8;
            // Combine the word-aligned words in the correct order for current endianness.
            D[d+r] = if BigEndian(AccType_ATOMIC) then word1:word2 else word2:word1;
        if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.113   VLDR (immediate)

Load SIMD&FP register (immediate) loads a single register from the Advanced SIMD and floating-point register file, using an address from a general-purpose register, with an optional offset.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

Half-precision scalar variant

Applies when size == 01.

VLDR{<c>}{<q>}.16 <Sd>, [<Rn> {, #{+/-}<imm}>]

Single-precision scalar variant

Applies when size == 10.

VLDR{<c>}{<q>}{.32} <Sd>, [<Rn> {, #{+/-}<imm}>]

Double-precision scalar variant

Applies when size == 11.

VLDR{<c>}{<q>}{.64} <Dd>, [<Rn> {, #{+/-}<imm}>]

Decode for all variants of this encoding

if size == '00' || (size == '01' & & !HaveFP16Ext()) then UNDEFINED;
if size == '01' & & cond != '1110' then UNPREDICTABLE;
esize = 8 << UInt(size); add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
in(ger) d;
case size of
when '01' d = UInt(Vd:D);
when '10' d = UInt(Vd:D);
when '11' d = UInt(D:Vd);
n = UInt(Rn);

CONCONSTRAINED UNPREDICTABLE behavior

If size == '01' & & cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1
**Half-precision scalar variant**
Applies when size == 01.
\[VLDR{<c>}{<q>}.16 <Sd>, [<Rn> {, #{+/−}<imm>}]\]

**Single-precision scalar variant**
Applies when size == 10.
\[VLDR{<c>}{<q>}{.32} <Sd>, [<Rn> {, #{+/−}<imm>}]\]

**Double-precision scalar variant**
Applies when size == 11.
\[VLDR{<c>}{<q>}{.64} <Dd>, [<Rn> {, #{+/−}<imm>}]\]

**Decode for all variants of this encoding**

\[
\begin{align*}
    \text{if } \text{size} &= '00' \text{ || } (\text{size} = '01' \text{ \&\& \!HaveFP16Ext()} \text{ then UNDEFINED;)} \\
    \text{if } \text{size} &= '01' \text{ \&\& InITBlock()} \text{ then UNPREDICTABLE;} \\
    \text{esize} &= 8 \text{ << UInt(size); } \text{add} &= (U = '1'); \\
    \text{imm32} &= \text{if esize} = 16 \text{ then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);} \\
    \text{integer } d; \\
    \text{case size of} \\
    \text{when '01'} \text{ d} &= \text{UInt(Vd:D);} \\
    \text{when '10'} \text{ d} &= \text{UInt(Vd:D);} \\
    \text{when '11'} \text{ d} &= \text{UInt(D:Vd);} \\
    \text{n} &= \text{UInt(Rn);}
\end{align*}
\]

**CONSTRAINED UNPREDICTABLE behavior**
If size == '01' \&\& InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

- **<>** See Standard assembler syntax fields on page F1-7228.
- **<q>** See Standard assembler syntax fields on page F1-7228.
- **.64** Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
- **<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- **.32** Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
- **<Sd>** Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- **<Rn>** Is the general-purpose base register, encoded in the "Rn" field.
- **+/−** Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
  
  - when \(U = 0\)
  - when \(U = 1\)
<imm> For the single-precision scalar or double-precision scalar variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as <imm>/4.

For the half-precision scalar variant: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as <imm>/2.

**Operation for all encodings**

if ConditionPassed() then
   EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
   base = if n == 15 then Align(PC,4) else R[n];
   address = if add then (base + imm32) else (base - imm32);
   case esize of
      when 16
         S[d] = Zeros(16) : MemA[address,2];
      when 32
         S[d] = MemA[address,4];
      when 64
         word1 = MemA[address,4]; word2 = MemA[address+4,4];
         // Combine the word-aligned words in the correct order for current endianness.
         D[d] = if BigEndian(AccType_ATOMIC) then word1:word2 else word2:word1;

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.114   VLDR (literal)

Load SIMD&FP register (literal) loads a single register from the Advanced SIMD and floating-point register file, using an address from the PC value and an immediate offset.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

Half-precision scalar variant

Applies when size == 01.

VLDR{<c>}{<q>}.16 <Sd>, <label>
VLDR{<c>}{<q>}.16 <Sd>, [PC, #{+/-}<imm>]

Single-precision scalar variant

Applies when size == 10.

VLDR{<c>}{<q>}{.32} <Sd>, <label>
VLDR{<c>}{<q>}{.32} <Sd>, [PC, #{+/-}<imm>]

Double-precision scalar variant

Applies when size == 11.

VLDR{<c>}{<q>}{.64} <Dd>, <label>
VLDR{<c>}{<q>}{.64} <Dd>, [PC, #{+/-}<imm>]

Decode for all variants of this encoding

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
esize = 8 << UInt(size);  add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
integer d;
case size of
  when '01' d = UInt(Vd:D);
  when '10' d = UInt(Vd:D);
  when '11' d = UInt(D:Vd);
n = UInt(Rn);

CONSTRANED UNPREDITABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions
F6.1 Alphabetical list of Advanced SIMD and floating-point instructions

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 0 15 12 11 10 9 8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 1 0 1</td>
<td>U</td>
</tr>
</tbody>
</table>

Rn

**Half-precision scalar variant**

Applies when size == 01.

VLDR{<c>}{<q>}.16 <Sd>, <label>
VLDR{<c>}{<q>}.16 <Sd>, [PC, #{+/−}<imm>]

**Single-precision scalar variant**

Applies when size == 10.

VLDR{<c>}{<q>}{.32} <Sd>, <label>
VLDR{<c>}{<q>}{.32} <Sd>, [PC, #{+/−}<imm>]

**Double-precision scalar variant**

Applies when size == 11.

VLDR{<c>}{<q>}{.64} <Dd>, <label>
VLDR{<c>}{<q>}{.64} <Dd>, [PC, #{+/−}<imm>]

**Decode for all variants of this encoding**

if size == '00' || (size == '01' & & !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
esize = 8 << UInt(size); add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
integer d;
case size of
  when '01' d = UInt(Vd:D);
  when '10' d = UInt(Vd:D);
  when '11' d = UInt(D:Vd);
n = UInt(Rn);

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' & & InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

<
| See Standard assembler syntax fields on page F1-7228.

<q>

.64

Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

<0d>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

.32

Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<label> The label of the literal data item to be loaded.

For the single-precision scalar or double-precision scalar variants: the assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values are multiples of 4 in the range -1020 to 1020.

For the half-precision scalar variant: the assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values are multiples of 2 in the range -510 to 510.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE.
If the offset is negative, imm32 is equal to minus the offset and add == FALSE.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
- when U = 0
+ when U = 1

<imm> For the single-precision scalar or double-precision scalar variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as <imm>/4.
For the half-precision scalar variant: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as <imm>/2.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see Use of labels in UAL instruction syntax on page F2-7257.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
  base = if n == 15 then Align(PC,4) else R[n];
  address = if add then (base + imm32) else (base - imm32);
  case esize of
    when 16
      S[d] = Zeros(16) : MemA[address,2];
    when 32
      S[d] = MemA[address,4];
    when 64
      word1 = MemA[address,4]; word2 = MemA[address+4,4];
      // Combine the word-aligned words in the correct order for current endianness.
      D[d] = if BigEndian(AccType_ATOMIC) then word1:word2 else word2:word1;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.115 **VMAX (floating-point)**

Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements are floating-point numbers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

### 64-bit SIMD vector variant

Applies when \( Q = 0 \).

\[
\text{VMAX}\{\text{c}\}\{\text{q}\}.\text{dt} \{\text{Dd}, \text{Dn}, \text{Dm}\}
\]

### 128-bit SIMD vector variant

Applies when \( Q = 1 \).

\[
\text{VMAX}\{\text{c}\}\{\text{q}\}.\text{dt} \{\text{Qd}, \text{Qn}, \text{Qm}\}
\]

**Decode for all variants of this encoding**

if \( Q = '1' \&\& (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') \) then UNDEFINED;
if \( sz = '1' \&\& !\text{HaveFP16Ext}() \) then UNDEFINED;
maximum = (op == '0');
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = \text{UInt}(D:Vd); n = \text{UInt}(N:Vn); m = \text{UInt}(M:Vm); regs = if Q == '0' then 1 else 2;

**T1**

\[
\begin{array}{cccccccccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 0 & D & 0 & sz & Vn & Vd & 1 & 1 & 1 & N & Q & M & 0 & Vm \\
\end{array}
\]

**64-bit SIMD vector variant**

Applies when \( Q = 0 \).

\[
\text{VMAX}\{\text{c}\}\{\text{q}\}.\text{dt} \{\text{Dd}, \text{Dn}, \text{Dm}\}
\]

**128-bit SIMD vector variant**

Applies when \( Q = 1 \).

\[
\text{VMAX}\{\text{c}\}\{\text{q}\}.\text{dt} \{\text{Qd}, \text{Qn}, \text{Qm}\}
\]
Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
maximum = (op == '0');
integer esize;
integer elements;
case sz of
   when '0' esize = 32; elements = 2;
   when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:
   F32 when sz = 0
   F16 when sz = 1

<q> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <q>d.*2.
<q> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <q>n.*2.
<q> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <q>m.*2.
<q> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<q> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<q> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Floating-point maximum and minimum
• max(+0.0, -0.0) = +0.0
• If any input is a NaN, the corresponding result element is the default NaN.

Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations();  CheckAdvSIMDEnabled();
for r = 0 to regs-1
   for e = 0 to elements-1
      op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
      if maximum then
         Elem[D[d+r],e,esize] = FMax(op1, op2, StandardFPSCRValue());
else
    Elem[D[d+r],e,esize] = FPMin(op1, op2, StandardFPSCRValue());
F6.1.116 VMAX (integer)

Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VMAX{c}{q}.<dt> {Dd}, {Dn}, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VMAX{c}{q}.<dt> {Qd}, {Qn}, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<> == '1' || Vn<> == '1' || Vm<> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0'); unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when Q == 0.

VMAX{c}{q}.<dt> {<Dd>}, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VMAX{c}{q}.<dt> {<Qd>}, <Qn>, <Qm>
Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0'); unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Assembler symbols
<q> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:
S8 when U = 0, size = 00
S16 when U = 0, size = 01
S32 when U = 0, size = 10
U8 when U = 1, size = 00
U16 when U = 1, size = 01
U32 when U = 1, size = 10

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations(); CheckAdvSIMDEnabled();
   for r = 0 to regs-1
      for e = 0 to elements-1
         op1 = Int(Elem[D[n+r],e,esize], unsigned);
         op2 = Int(Elem[D[m+r],e,esize], unsigned);
         result = if maximum then Max(op1,op2) else Min(op1,op2);
         Elem[D[d+r],e,esize] = result<esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.117 VMAXNM

This instruction determines the floating-point maximum number.

It handles NaNs in consistence with the IEEE754-2008 specification. It returns the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to floating-point VMAX.

This instruction is not conditional.

A1

64-bit SIMD vector variant

Applies when \( Q = 0 \).

\[
\text{VMAXNM}\{<q>\}.<dt> <Dd>, <Dn>, <Dm>
\]

128-bit SIMD vector variant

Applies when \( Q = 1 \).

\[
\text{VMAXNM}\{<q>\}.<dt> <Qd>, <Qn>, <Qm>
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{if} \ Q &= '1' \ \&\& \ \text{(Vd}<0> = '1' \ \| \ \text{Vn}<0> = '1' \ \| \ \text{Vm}<0> = '1') \ \text{then UNDEFINED}; \\
\text{if} \ sz &= '1' \ \&\& \ \text{!HaveFP16Ext()} \ \text{then UNDEFINED}; \\
\text{maximum} &= (\text{op} == '0'); \\
\text{advsimd} &= \text{TRUE}; \\
\text{integer esize}; \\
\text{integer elements}; \\
\text{case} \ sz \ \text{of} \\
& \quad \text{when} \ '0' \ \text{esize} = 32; \ \text{elements} = 2; \\
& \quad \text{when} \ '1' \ \text{esize} = 16; \ \text{elements} = 4; \\
& \quad d = \text{UInt}(D:Vd); \ n = \text{UInt}(N:Vn); \ m = \text{UInt}(M:Vm); \ \text{regs} = \text{if} \ Q = '0' \ \text{then} \ 1 \ \text{else} \ 2;
\end{align*}
\]

A2

Half-precision scalar variant

Applies when size == 01.

\[
\text{VMAXNM}\{<q>\}.F16 <Sd>, <Sn>, <Sm> // Cannot be conditional
\]

Single-precision scalar variant

Applies when size == 10.

\[
\text{VMAXNM}\{<q>\}.F32 <Sd>, <Sn>, <Sm> // Cannot be conditional
\]

Double-precision scalar variant

Applies when size == 11.
VMAXNM{<q>}.F64 <Dd>, <Dn>, <Dm> // Cannot be conditional

**Decode for all variants of this encoding**

```plaintext
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
T1
```

**64-bit SIMD vector variant**

Applies when Q == 0.

VMAXNM{<q>}.<dt> <Dd>, <Dn>, <Dm>

**128-bit SIMD vector variant**

Applies when Q == 1.

VMAXNM{<q>}.<dt> <Qd>, <Qn>, <Qm>

**Decode for all variants of this encoding**

```plaintext
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

**CONSTRANDED UNPREDICTABLE behavior**

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
T2

```
T2

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4</th>
<th>3 0</th>
<th>12 11 10 9 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 0 1</td>
<td>D 0 0</td>
<td>Vn</td>
<td>Vd</td>
</tr>
</tbody>
</table>
```

**Half-precision scalar variant**

Applies when size == 01.

\[ \text{VMAXNM}\langle q\rangle.F16 <Sd>, <Sn>, <Sm> \] // Not permitted in IT block

**Single-precision scalar variant**

Applies when size == 10.

\[ \text{VMAXNM}\langle q\rangle.F32 <Sd>, <Sn>, <Sm> \] // Not permitted in IT block

**Double-precision scalar variant**

Applies when size == 11.

\[ \text{VMAXNM}\langle q\rangle.F64 <Dd>, <Dn>, <Dm> \] // Not permitted in IT block

**Decode for all variants of this encoding**

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
integer esize;
integer d;
integer n;
integer m;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

\(<q\rangle\) See Standard assembler syntax fields on page F1-7228.

\(<dt\)> Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

- F32 when sz = 0
- F16 when sz = 1
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

**Operation for all encodings**

EncodingSpecificOperations(); CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
if advsimd then // Advanced SIMD instruction
  for r = 0 to regs-1 
    for e = 0 to elements-1
      op1 = Elem[D[n+r], e, esize]; op2 = Elem[D[m+r], e, esize];
      if maximum then
        Elem[D[d+r], e, esize] = FPMaxNum(op1, op2, StandardFPSCRValue());
      else
        Elem[D[d+r], e, esize] = FPMinNum(op1, op2, StandardFPSCRValue());
  
else // VFP instruction
  case esize of
    when 16
      if maximum then
        S[d] = Zeros(16) : FPMaxNum(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
      else
        S[d] = Zeros(16) : FPMinNum(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
    when 32
      if maximum then
        S[d] = FPMaxNum(S[n], S[m], FPSCR[]);
      else
        S[d] = FPMinNum(S[n], S[m], FPSCR[]);
    when 64
      if maximum then
        D[d] = FPMaxNum(D[n], D[m], FPSCR[]);
      else
        D[d] = FPMinNum(D[n], D[m], FPSCR[]);
F6.1.118 \textbf{VMIN (floating-point)}

Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements are floating-point numbers.

Depending on settings in the \texttt{CPACR}, \texttt{NSACR}, and \texttt{HCPTR} registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see \textit{Enabling Advanced SIMD and floating-point support} on page G1-8996.

\textbf{A1}

\begin{verbatim}
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4 3 0 ]
| 1 1 1 0 0 1 0 0 D 1 sz Vn Vd 1 1 1 N Q M 0 Vm |
\end{verbatim}

64-bit SIMD vector variant

Applies when $Q == 0$.

\texttt{VMIN\{<c>\}{<q>}.<dt> \{<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when $Q == 1$.

\texttt{VMIN\{<c>\}{<q>}.<dt> \{<Qd>, }<Qn>, <Qm>

\textbf{Decode for all variants of this encoding}

if $Q == '1' \&\& (Vd<0> == '1' \&\& Vn<0> == '1' \&\& Vm<0> == '1')$ then UNDEFINED;
if $sz == '1' \&\& !HaveFP16Ext()$ then UNDEFINED;
maximum = (op == '0');
exteger esize;
exteger elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); reg = if $Q == '0'$ then 1 else 2;

\textbf{T1}

\begin{verbatim}
[15 14 13|12|11 10 9 8 7 6 5 4 3 0 ]
| 1 1 1 0 1 1 1 1 0 D 1 sz Vn Vd 1 1 1 N Q M 0 Vm |
\end{verbatim}

64-bit SIMD vector variant

Applies when $Q == 0$.

\texttt{VMIN\{<c>\}{<q>}.<dt> \{<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when $Q == 1$.

\texttt{VMIN\{<c>\}{<q>}.<dt> \{<Qd>, }<Qn>, <Qm>
**Decode for all variants of this encoding**

```c
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
maximum = (op == '0');
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
endcase
```

**CONSTRAINED UNPREDICTABLE behavior**

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

- `<c>` For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.
  
  For encoding T1: see *Standard assembler syntax fields on page F1-7228*.

- `<q>` See *Standard assembler syntax fields on page F1-7228*.

- `<dt>` Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:
  - F32 when sz = 0
  - F16 when sz = 1

- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

- `<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

- `<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Floating-point minimum**

- \( \min(+0.0, -0.0) = -0.0 \)
- If any input is a NaN, the corresponding result element is the default NaN.

**Operation for all encodings**

```c
if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
      if maximum then
        Elem[D[d+r],e,esize] = FPMax(op1, op2, StandardFPSCRValue());
```

---

F6-8498  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  ARM DDI 0487I.a  Non-Confidential  ID081822
else
    Elem[D[d+r],e,esize] = FPMin(op1, op2, StandardFPSCRVValue());
F6.1.119 **VMIN (integer)**

Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

A1

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8|7 6 5 4|3 0 |
|1 1 1 1 0 0 1 |U|0|D|size|Vn|Vd|0 1 1 0 |N|Q|M|1 |Vm |
```

**64-bit SIMD vector variant**

Applies when Q == 0.

`VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>`

**128-bit SIMD vector variant**

Applies when Q == 1.

`VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>`

**Decode for all variants of this encoding**

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0');
unsigned = (U == '1');
esize = 8 << UInt(size);
elements = 64 DIV esize;
d = UInt(D:Vd);
n = UInt(N:Vn);
m = UInt(M:Vm);
regs = if Q == '0' then 1 else 2;

T1

```
|15 14 13 12|11 10 9 8|7 6 5 4|3 0 |15 12|11 10 9 8|7 6 5 4|3 0 |
|1 1 1 |U|1|1|1|0 |D|size|Vn|Vd|0 1 1 0 |N|Q|M|1 |Vm |
```

**64-bit SIMD vector variant**

Applies when Q == 0.

`VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>`

**128-bit SIMD vector variant**

Applies when Q == 1.

`VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>`. 
F6.1 Alphabetical list of Advanced SIMD and floating-point instructions

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Assembler symbols

<q> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:
S8 when U = 0, size = 00
S16 when U = 0, size = 01
S32 when U = 0, size = 10
U8 when U = 1, size = 00
U16 when U = 1, size = 01
U32 when U = 1, size = 10

<Dd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Dd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations();  CheckAdvSIMDEnabled();
   for r = 0 to regs-1
      for e = 0 to elements-1
         op1 = Int(Elem[D[n+r],e,esize], unsigned);
         op2 = Int(Elem[D[m+r],e,esize], unsigned);
         result = if maximum then Max(op1,op2) else Min(op1,op2);
         Elem[D[d+r],e,esize] = result<esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.120 \textbf{VMINNM}

This instruction determines the floating point minimum number.

It handles NaNs in consistence with the IEEE754-2008 specification. It returns the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to floating-point \texttt{VMIN}.

This instruction is not conditional.

\textbf{A1}

\begin{verbatim}
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4 3 0 ]
  1 1 1 0 0 1 1 0 D 1 sz Vn Vd 1 1 1 1 N Q M 1 Vm
\end{verbatim}

\textbf{64-bit SIMD vector variant}

Applies when \(Q == 0\).

\texttt{VMINNM\{<q>\}.<dt> <Dd>, <Dn>, <Dm>}

\textbf{128-bit SIMD vector variant}

Applies when \(Q == 1\).

\texttt{VMINNM\{<q>\}.<dt> <Qd>, <Qn>, <Qm>}

\textbf{Decode for all variants of this encoding}

\begin{verbatim}
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
\end{verbatim}

\textbf{A2}

\begin{verbatim}
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4 3 0 ]
  1 1 1 1 1 0 1 1 0 D 0 0 Vn Vd 1 0 l=00 N 1 M 0 Vm
\end{verbatim}

\textbf{Half-precision scalar variant}

Applies when \(\text{size} == 01\).

\texttt{VMINNM\{<q>\}.F16 <Sd>, <Sn>, <Sm> // Cannot be conditional}

\textbf{Single-precision scalar variant}

Applies when \(\text{size} == 10\).

\texttt{VMINNM\{<q>\}.F32 <Sd>, <Sn>, <Sm> // Cannot be conditional}

\textbf{Double-precision scalar variant}

Applies when \(\text{size} == 11\).
Decode for all variants of this encoding

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

64-bit SIMD vector variant
Applies when Q == 0.

VMINNM{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant
Applies when Q == 1.

VMINNM{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior
If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
T2

Half-precision scalar variant
Applies when size == 01.
VMINNM{<q>}.F16 <Sd>, <Sn>, <Sm> // Not permitted in IT block

Single-precision scalar variant
Applies when size == 10.
VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block

Double-precision scalar variant
Applies when size == 11.
VMINNM{<q>}.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block

Decode for all variants of this encoding
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
integer esize;
integer d;
integer n;
integer m;
case size of
   when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
   when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
   when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
CONstrained UNPREDICTABLE behavior
If InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings
For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols
<q> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:
   F32 when sz = 0
   F16 when sz = 1
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

**Operation for all encodings**

```
EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
if advsimd then             // Advanced SIMD instruction
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = Elem[D[n+r], e, esize]; op2 = Elem[D[m+r], e, esize];
      if maximum then
        Elem[D[d+r], e, esize] = FPMaxNum(op1, op2, StandardFPSCRValue());
      else
        Elem[D[d+r], e, esize] = FPMinNum(op1, op2, StandardFPSCRValue());
  else                        // VFP instruction
    case esize of
      when 16
        if maximum then
          S[d] = Zeros(16) : FPMaxNum(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
        else
          S[d] = Zeros(16) : FPMinNum(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
      when 32
        if maximum then
          S[d] = FPMaxNum(S[n], S[m], FPSCR[]);
        else
          S[d] = FPMinNum(S[n], S[m], FPSCR[]);
      when 64
        if maximum then
          D[d] = FPMaxNum(D[n], D[m], FPSCR[]);
        else
          D[d] = FPMinNum(D[n], D[m], FPSCR[]);
```
F6.1.121 VMLA (floating-point)

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and accumulates the results into the elements of the destination vector.

Depending on settings in the CPACR, NSACR, HCPR, and FPSCR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 0</td>
<td>D</td>
<td>0</td>
<td>sz</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>N</td>
<td>Q</td>
<td>M</td>
<td>1</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE; add = (op == '0');
exteger esize;
exteger elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D;Vd); n = UInt(N;Vn); m = UInt(M;Vm); regs = if Q == '0' then 1 else 2;

A2

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 1 1 1 0 0 0 | D | 0 | 0 | Vn | Vd | 1 | 0 | sz | N | 0 | M | 0 | Vm |

Half-precision scalar variant

Applies when size == 01.

VMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar variant

Applies when size == 10.

VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar variant

Applies when size == 11.
VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

**Decode for all variants of this encoding**

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**T1**

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 0</th>
<th>12 11 10 9 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0</td>
<td>1 1 1 0</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector variant**

Applies when Q == 0.
VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

**128-bit SIMD vector variant**

Applies when Q == 1.
VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

**Decode for all variants of this encoding**

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE; add = (op == '0');
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**CONSTRAINED UNPREDICTABLE behavior**

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**T2**

![Instruction format diagram]

**Half-precision scalar variant**

Applies when `size == 01`.

\[ \text{VMLA}\{<c>\}\{<q>\}.F16 \langle Sd \rangle, \langle Sn \rangle, \langle Sm \rangle \]

**Single-precision scalar variant**

Applies when `size == 10`.

\[ \text{VMLA}\{<c>\}\{<q>\}.F32 \langle Sd \rangle, \langle Sn \rangle, \langle Sm \rangle \]

**Double-precision scalar variant**

Applies when `size == 11`.

\[ \text{VMLA}\{<c>\}\{<q>\}.F64 \langle Dd \rangle, \langle Dn \rangle, \langle Dm \rangle \]

**Decode for all variants of this encoding**

```plaintext
if size == '00' || (size == '01' & HaveFP16Ext()) then UNDEFINED;
if size == '01' & InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

**CONSTRAINED UNPREDICTABLE behavior**

If `size == '01' & InITBlock()`, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

- `<c>` For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.
  
- `<q>` For encoding A2, T1 and T2: see *Standard assembler syntax fields on page F1-7228*.

- `<s>` See *Standard assembler syntax fields on page F1-7228*.
<dt>
Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:
F32 when sz = 0
F16 when sz = 1
</dt>

<dd>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
</dd>

<nn>
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
</nn>

<mm>
Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
</mm>

<dd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
</dd>

<nn>
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
</nn>

<mm>
Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
</mm>

<sd>
Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
</sd>

<sn>
Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
</sn>

<sm>
Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
</sm>

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
if advsimd then  // Advanced SIMD instruction
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRValue());
            addend = if add then product else FPNeg(product);
            Elem[D[d+r],e,esize] = FPAdd(Elem[D[d+r],e,esize], addend, StandardFPSCRValue());
else            // VFP instruction
    case esize of
    when 16
        addend16 = if add then FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]) else FPNeg(FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]));
        S[d] = Zeros(16) : FPAdd(S[d]<15:0>, addend16, FPSCR[]);
        when 32
            addend32 = if add then FPMul(S[n], S[m], FPSCR[]) else FPNeg(FPMul(S[n], S[m], FPSCR[]));
            S[d] = FPAdd(S[d], addend32, FPSCR[]);
        when 64
            addend64 = if add then FPMul(D[n], D[m], FPSCR[]) else FPNeg(FPMul(D[n], D[m], FPSCR[]));
            D[d] = FPAdd(D[d], addend64, FPSCR[]);
F6.1.122 \textbf{VMLA (integer)}

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and adds the products to the corresponding elements of the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see \textit{Enabling Advanced SIMD and floating-point support} on page G1-8996.

A1

\begin{verbatim}
|31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 16| 15| 12| 11| 10| 9| 8| 7| 6| 5| 4| 3| 0 |
| 1| 1| 1| 0| 0| 0| D  | size | Vn | Vd | 1| 0| 0| 1| N  | Q | M | 0 | Vm |
\end{verbatim}

\textbf{64-bit SIMD vector variant}

Applies when Q == 0.

\texttt{VMLA\{<c>\}{<q>}.<dt> <Dd>, <Dn>, <Dm>}

\textbf{128-bit SIMD vector variant}

Applies when Q == 1.

\texttt{VMLA\{<c>\}{<q>}.<dt> <Qd>, <Qn>, <Qm>}

\textbf{Decode for all variants of this encoding}

if size == '11' then UNDEFINED;
if Q == '1' || (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

\begin{verbatim}
|15| 14| 13| 12| 11| 10| 9| 8| 7| 6| 5| 4| 3| 0| 15| 12| 11| 10| 9| 8| 7| 6| 5| 4| 3| 0 |
| 1| 1| 1| 0| 1| 1| 1| 0| D  | size | Vn | Vd | 1| 0| 0| 1| N  | Q | M | 0 | Vm |
\end{verbatim}

\textbf{64-bit SIMD vector variant}

Applies when Q == 0.

\texttt{VMLA\{<c>\}{<q>}.<dt> <Dd>, <Dn>, <Dm>}

\textbf{128-bit SIMD vector variant}

Applies when Q == 1.

\texttt{VMLA\{<c>\}{<q>}.<dt> <Qd>, <Qn>, <Qm>}

\textbf{Decode for all variants of this encoding}

if size == '11' then UNDEFINED;
if Q == '1' || (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Assembler symbols

<> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:
I8 when size = 00
I16 when size = 01
I32 when size = 10

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[<Dd>,<Dn>,<Dm>],e,esize,unsigned) * Int(Elem[<Dd>,<Dn>,<Dm>],e,esize,unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.123  VMLA (by scalar)

Vector Multiply Accumulate multiplies elements of a vector by a scalar, and adds the products to corresponding elements of the destination vector.

For more information about scalars see Advanced SIMD scalars on page F1-7254.

Depending on settings in the CPACR, NSACR, and HCPTTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8|7 6 5 4|3 0 |
|-----------|-----------|-----------|----|----|--------|----|----|----|
| 1 1 1 0 0 1 Q D | Vn Vd 0 0 F N 1 M 0 Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VMLA{c}{q}.<dt> <Dd>, <Dn>, <Dm[x]>

128-bit SIMD vector variant

Applies when Q == 1.

VMLA{c}{q}.<dt> <Qd>, <Qn>, <Dm[x]>

Decode for all variants of this encoding

if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 Q 1 1 1 1 D</td>
<td>Vn Vd 0 0 F N 1 M 0 Vm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VMLA{c}{q}.<dt> <Dd>, <Dn>, <Dm[x]>

128-bit SIMD vector variant

Applies when Q == 1.

VMLA{c}{q}.<dt> <Qd>, <Qn>, <Dm[x]>
Decode for all variants of this encoding

if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (V<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE; // 'Don't care' value: TRUE produces same functionality
add = (op == '0'); floating_point = (F == '1'); long_destination = FALSE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

Assembler symbols

<> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the scalar and the elements of the operand vector, encoded in the "F:size" field. It can have the following values:
I16 when F = 0, size = 01
I32 when F = 0, size = 10
F16 when F = 1, size = 01
F32 when F = 1, size = 10

<Qd> Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is I16 or F16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is I32 or F32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Dim[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Dim[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else
                    FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
    
Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.124 VMLAL (integer)

Vector Multiply Accumulate Long multiplies corresponding elements in two vectors, and add the products to the corresponding element of the destination vector. The destination vector element is twice as long as the elements that are multiplied.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see "Enabling Advanced SIMD and floating-point support on page G1-8996."

A1

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 1 & U & 1 & D & \text{size} & Vn & Vd & 1 & 0 & 0 & 0 & N & 0 & M & 0 & Vm \\
\end{array}
\]

A1 variant

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0'); long_destination = TRUE; unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;

T1

\[
\begin{array}{cccccccccccccccc}
\text{size} & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 \\
1 & 1 & 1 & U & 1 & 1 & 1 & 1 & D & \text{size} & Vn & Vd & 1 & 0 & 0 & 0 & N & 0 & M & 0 & Vm \\
\end{array}
\]

T1 variant

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0'); long_destination = TRUE; unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;

Notes for all encodings

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

Assembler symbols

<

For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.
Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

- **S8** when \( U = 0, \) size = \( 00 \)
- **S16** when \( U = 0, \) size = \( 01 \)
- **S32** when \( U = 0, \) size = \( 10 \)
- **U8** when \( U = 1, \) size = \( 00 \)
- **U16** when \( U = 1, \) size = \( 01 \)
- **U32** when \( U = 1, \) size = \( 10 \)

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

If `ConditionPassed()` then

```plaintext
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1
  for e = 0 to elements-1
    product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[n+r],e,esize],unsigned);
    addend = if add then product else -product;
    if long_destination then
      Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
    else
      Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.125 VMLAL (by scalar)

Vector Multiply Accumulate Long multiplies elements of a vector by a scalar, and adds the products to corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see Advanced SIMD scalars on page F1-7254.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if size == '00' || Vd== '1' then UNDEFINED;
unsigned = (U == '1');  add = (op == '0');  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
in integer esize;
in integer elements;
in integer m;
in integer index;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);

T1

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if size == '00' || Vd== '1' then UNDEFINED;
unsigned = (U == '1');  add = (op == '0');  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
in integer esize;
in integer elements;
in integer m;
in integer index;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
Notes for all encodings

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

Assembler symbols

<
  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
  For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q
  See Standard assembler syntax fields on page F1-7228.

<dt
  Is the data type for the scalar and the elements of the operand vector, encoded in the "U:size" field. It can have the following values:
  S16 when U = 0, size = 01
  S32 when U = 0, size = 10
  U16 when U = 1, size = 01
  U32 when U = 1, size = 10

<Qd>
  Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Dn>
  Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]>
  Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16 or U16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32 or U32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
      if floating_point then
        fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else
                      FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
        Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
      else
        addend = if add then op1val*op2val else -op1val*op2val;
        if long_destination then
          Elem[Q[d>>1],e,esize] = Elem[Qin[d>>1],e,esize] + addend;
        else
          Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.126  **VMLS (floating-point)**

Vector Multiply Subtract multiplies corresponding elements in two vectors, subtracts the products from corresponding elements of the destination vector, and places the results in the destination vector.

**Note**

Arm recommends that software does not use the VMLS instruction in the Round towards Plus Infinity and Round towards Minus Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

**A1**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 16 15 | 12 11 10 | 9 8 7 6 5 4 3 0 |
|-------------|-------------|-------------|-----------|-----------|-------------|-------------|
| 1 1 1 0 0 0 1 0 | D | 1 | sz | Vn | Vd | 1 1 0 1 | N | Q | M | 1 | Vm |

**64-bit SIMD vector variant**

Applies when Q == 0.

VMLS{c}{q}{dt} <Dd>, <Dn>, <Dm>

**128-bit SIMD vector variant**

Applies when Q == 1.

VMLS{c}{q}{dt} <Qd>, <Qn>, <Qm>

**Decode for all variants of this encoding**

if Q == '1' && (Vd<0> == 'I' || Vn<0> == 'I' || Vm<0> == 'I') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE; add = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
    d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**A2**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15</th>
<th>12 11 10</th>
<th>9 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 0</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Vn</td>
<td>Vd</td>
<td>1 0</td>
<td>size</td>
</tr>
</tbody>
</table>

**Half-precision scalar variant**

Applies when size == 01.

VMLS{c}{q}.F16 <Sd>, <Sn>, <Sm>

**Single-precision scalar variant**

Applies when size == 10.
VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

**Double-precision scalar variant**

Applies when size == 11.

VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

**Decode for all variants of this encoding**

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && !cond || '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

**CONstrained unPRedictable behavior**

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**t1**

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector variant**

Applies when Q == 0.

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

**128-bit SIMD vector variant**

Applies when Q == 1.

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

**decode for all variants of this encoding**

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && !InITBlock() then UNPREDICTABLE;
advsimd = TRUE; add = (op == '0');
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
  d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
**CONSTRAINED UNPREDICTABLE behavior**

If `sz == '1' & & InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

### Half-precision scalar variant

Applies when `size == 01`.

`VMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>`

### Single-precision scalar variant

Applies when `size == 10`.

`VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>`

### Double-precision scalar variant

Applies when `size == 11`.

`VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>`

### Decode for all variants of this encoding

```c
if size == '00' || (size == '01' & & !HaveFP16Ext()) then UNDEFINED;
if size == '01' & & InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

### CONSTRAINED UNPREDICTABLE behavior

If `size == '01' & & InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

### Assembler symbols

`<c>`

For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.
For encoding A2, T1 and T2: see *Standard assembler syntax fields* on page F1-7228.

See *Standard assembler syntax fields* on page F1-7228.

Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

- F32 when sz = 0
- F16 when sz = 1

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

**Operation for all encodings**

if `ConditionPassed()` then

EncodingSpecificOperations(); CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);

if advsimd then  // Advanced SIMD instruction

for r = 0 to regs-1

for e = 0 to elements-1

product = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRValue());

addend = if add then product else FPNeg(product);

Elem[D[d+r],e,esize] = FPAdd(Elem[D[d+r],e,esize], addend, StandardFPSCRValue());

else             // VFP instruction

case esize of

when 16

addend16 = if add then FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]) else FPNeg(FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]));

S[d] = Zeros(16) : FPAdd(S[d]<15:0>, addend16, FPSCR[]);

when 32

addend32 = if add then FPMul(S[n], S[m], FPSCR[]) else FPNeg(FPMul(S[n], S[m], FPSCR[]));

S[d] = FPAdd(S[d], addend32, FPSCR[]);

when 64

addend64 = if add then FPMul(D[n], D[m], FPSCR[]) else FPNeg(FPMul(D[n], D[m], FPSCR[]));

D[d] = FPAdd(D[d], addend64, FPSCR[]);
F6.1.127   VMLS (integer)

Vector Multiply Subtract multiplies corresponding elements in two vectors, and subtracts the products from the 
corresponding elements of the destination vector.

Depending on settings in the CPACR, NSACR, and HCPRTR registers, and the Security state and PE mode in which 
the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For 
more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

if size == 'll' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when Q == 0.

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

if size == 'll' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
        d = UInt(D;Vd);  n = UInt(N;Vn);  m = UInt(M;Vm);  regs = if Q == '0' then 1 else 2;

Assembler symbols

<> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

I8 when size = 00
I16 when size = 01
I32 when size = 10

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D;Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N;Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M;Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D;Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N;Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M;Vm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[D;Vd][n+r],e,esize],unsigned) * Int(Elem[D;Vd][m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q;Vd][d>>1],e,2*esize = Elem[Q;Vn][d>>1],e,2*esize] + addend;
            else
                Elem[D;d+r],e,esize] = Elem[D;Vd][d+r],e,esize] + addend;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.128  VMLS (by scalar)

Vector Multiply Subtract multiplies elements of a vector by a scalar, and either subtracts the products from corresponding elements of the destination vector.

For more information about scalars see Advanced SIMD scalars on page F1-7254.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

|   |   |   |   |  Q  | D | 11 |   | Vn | Vd | 0 | 1 | 0 | F | N | 1 | M | 0 | Vm |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|   | 1 | 1 | 1 | 0 | 0 | 1 |  |   |   |   |   |   |   |   |   |   |   |

64-bit SIMD vector variant

Applies when Q = 0.

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

128-bit SIMD vector variant

Applies when Q = 1.

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>

Decode for all variants of this encoding

if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' & size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
add = (op == '0'); floating_point = (F == '1'); long_destination = FALSE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm);
T1

|   |   |   |   |  Q  | D | 11 |   | Vn | Vd | 0 | 1 | 0 | F | N | 1 | M | 0 | Vm |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|   | 1 | 1 | 1 | 1 | 1 | 1 |   |   |   |   |   |   |   |   |   |   |   |

64-bit SIMD vector variant

Applies when Q = 0.

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

128-bit SIMD vector variant

Applies when Q = 1.

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
**Decode for all variants of this encoding**

if size == '11' then see "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
add = (op == '0'); floating_point = (F == '1'); long_destination = FALSE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

**CONSTRAINED UNPREDICTABLE behavior**

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Notes for all encodings**

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

**Assembler symbols**

<> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

<> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the scalar and the elements of the operand vector, encoded in the "F:size" field. It can have the following values:

- I16 when F = 0, size = 01
- I32 when F = 0, size = 10
- F16 when F = 1, size = 01
- F32 when F = 1, size = 10

<Qd> Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is I16 or F16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is I32 or F32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else
                    FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
    
Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.129  VMLSL (integer)

Vector Multiply Subtract Long multiplies corresponding elements in two vectors, and subtract the products from the corresponding elements of the destination vector. The destination vector element is twice as long as the elements that are multiplied.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

\[
\begin{array}{cccccccccccccc}
\text{size} & \text{op} & 1 & 1 & 1 & 0 & 0 & 1 & U & 1 & D & l=11 & \text{Vn} & \text{Vd} & 1 & 0 & 1 & 0 & \text{N} & 0 & \text{M} & 0 & \text{Vm}
\end{array}
\]

A1 variant

VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;

T1

\[
\begin{array}{cccccccccccccccccccc}
\text{size} & \text{op} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 3 & 0 & 15 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0
\end{array}
\]

T1 variant

VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;

Notes for all encodings

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

Assembler symbols

<c>  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.
See Standard assembler syntax fields on page F1-7228.

Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>S8</td>
<td>U = 0, size = 00</td>
</tr>
<tr>
<td>S16</td>
<td>U = 0, size = 01</td>
</tr>
<tr>
<td>S32</td>
<td>U = 0, size = 10</td>
</tr>
<tr>
<td>U8</td>
<td>U = 1, size = 00</td>
</tr>
<tr>
<td>U16</td>
<td>U = 1, size = 01</td>
</tr>
<tr>
<td>U32</td>
<td>U = 1, size = 10</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[n+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**F6.1.130 VMLSL (by scalar)**

Vector Multiply Subtract Long multiplies elements of a vector by a scalar, and subtracts the products from corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see *Advanced SIMD scalars* on page F1-7254.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

**A1**

<table>
<thead>
<tr>
<th>size</th>
<th>op</th>
<th>D</th>
<th>Vn</th>
<th>Vd</th>
<th>N</th>
<th>M</th>
<th>Vm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>U</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>U</td>
<td>D</td>
<td>I=11</td>
</tr>
</tbody>
</table>

**A1 variant**

VMLSL\{<c>\}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

**Decode for this encoding**

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1'); add = (op == '0'); floating_point = FALSE; long_destination = TRUE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = 1;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

**T1**

<table>
<thead>
<tr>
<th>size</th>
<th>op</th>
<th>D</th>
<th>Vn</th>
<th>Vd</th>
<th>N</th>
<th>M</th>
<th>Vm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>U</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>U</td>
<td>D</td>
</tr>
</tbody>
</table>

**T1 variant**

VMLSL\{<c>\}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

**Decode for this encoding**

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1'); add = (op == '0'); floating_point = FALSE; long_destination = TRUE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = 1;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);
Notes for all encodings

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

Assembler symbols

<> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the scalar and the elements of the operand vector, encoded in the "U:size" field. It can have the following values:

- S16 when U = 0, size = 01
- S32 when U = 0, size = 10
- U16 when U = 1, size = 01
- U32 when U = 1, size = 10

<Qd> Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16 or U16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32 or U32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[ DIN[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[ DIN[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else
                FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
                Elem[D[d+r],e,esize] = FPA dd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[0][d>1],e,2*esize] = Elem[ Qin[d>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.131 VMMLA

BFLOAT16 floating-point matrix multiply-accumulate. This instruction multiplies the 2x4 matrix of B16 values in the first 128-bit source vector by the 4x2 B16 matrix in the second 128-bit source vector. The resulting 2x2 single-precision matrix product is then added destructively to the 2x2 single-precision matrix in the 128-bit destination vector. This is equivalent to performing a 4-way dot product per destination element. The instruction does not update the FPSCR exception status.

Note

Arm expects that the VMMLA instruction will deliver a peak B16 multiply throughput that is at least as high as can be achieved using two VDOT instructions, with a goal that it should have significantly higher throughput.

A1

(FEAT-AA32BF16)

 VMMLA{<q>}.BF16 <Qd>, <Qn>, <Qm>

Decode for this encoding

if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = 2;

T1

(FEAT-AA32BF16)

 VMMLA{<q>}.BF16 <Qd>, <Qn>, <Qm>

Decode for this encoding

if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = 2;
Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

CheckAdvSIMDEnabled();

bits(128) op1 = Q[n>>1];
bits(128) op2 = Q[m>>1];
bits(128) acc = Q[d>>1];

Q[d>>1] = BFMatMulAdd(acc, op1, op2);
F6.1.132   VMOV (between two general-purpose registers and a doubleword floating-point register)

Copy two general-purpose registers to or from a SIMD&FP register copies two words from two general-purpose
registers into a doubleword register in the Advanced SIMD and floating-point register file, or from a doubleword
register in the Advanced SIMD and floating-point register file to two general-purpose registers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode
in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp
mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

![Instruction Format](image)

From general-purpose registers variant

Applies when op == 0.

\[
\text{VMOV}<\{c\}<\{q\}> <Dm>, <Rt>, <Rt2>
\]

To general-purpose registers variant

Applies when op == 1.

\[
\text{VMOV}<\{c\}<\{q\}> <Rt>, <Rt2>, <Dm>
\]

Decode for all variants of this encoding

\[
\text{to\_arm\_registers} = (op == '1'); t = \text{UInt}(Rt); t2 = \text{UInt}(Rt2); m = \text{UInt}(M;Vm);
\]

\[
\text{if } t == 15 || t2 == 15 \text{ then UNPREDICTABLE; } \text{// Armv8-A removes UNPREDICTABLE for R13}
\]

\[
\text{if to\_arm\_registers }&& t == t2 \text{ then UNPREDICTABLE;}
\]

CONSTRAINED UNPREDICTABLE behavior

If to\_arm\_registers \& t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1

![Instruction Format](image)

From general-purpose registers variant

Applies when op == 0.

\[
\text{VMOV}<\{c\}<\{q\}> <Dm>, <Rt>, <Rt2>
\]

To general-purpose registers variant

Applies when op == 1.

\[
\text{VMOV}<\{c\}<\{q\}> <Rt>, <Rt2>, <Dm>
\]
**Decode for all variants of this encoding**

to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(M:Vm);
if t == 15 || t2 == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if to_arm_registers && t == t2 then UNPREDICTABLE;

**CONSTRANGED UNPREDICTABLE behavior**

If to_arm_registers && t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

**Notes for all encodings**

For more information about the CONSTRANGED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VMOV (between two general-purpose registers and a doubleword floating-point register) on page K1-11579.

**Assembler symbols**

- `<Dm>` Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "M:Vm" field.
- `<Rt2>` Is the second general-purpose register that `<Dm>[63:32]` will be transferred to or from, encoded in the "Rt2" field.
- `<Rt>` Is the first general-purpose register that `<Dm>[31:0]` will be transferred to or from, encoded in the "Rt" field.
- `<c>` See **Standard assembler syntax fields** on page F1-7228.
- `<q>` See **Standard assembler syntax fields** on page F1-7228.

**Operation for all encodings**

if ConditionPassed() then
   EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
if to_arm_registers then
   R[t] = D[m]<31:0>;
   R[t2] = D[m]<63:32>;
else
   D[m]<31:0> = R[t];
   D[m]<63:32> = R[t2];

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.133  VMOV (between general-purpose register and half-precision)

Copy 16 bits of a general-purpose register to or from a 32-bit SIMD&FP register. This instruction transfers the value held in the bottom 16 bits of a 32-bit SIMD&FP register to the bottom 16 bits of a general-purpose register, or the value held in the bottom 16 bits of a general-purpose register to the bottom 16 bits of a 32-bit SIMD&FP register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

( FEAT_FP16 )

```
| 31 28|27 26 25 24|23 22 21 20|19 16|15 14|13|12|11 10 9 8|7 6 5 4|3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| !=1111| 1 1 1 0 0 0 0| op| Vn| Rt| 1 0 0 1| N[0][0][1]| 0[0][0][0][0]|
```

From general-purpose register variant

Applies when op == 0.

VMOV{<c>}{<q>}.F16 <Sn>, <Rt>

To general-purpose register variant

Applies when op == 1.

VMOV{<c>}{<q>}.F16 <Rt>, <Sn>

Decode for all variants of this encoding

```
if !HaveFP16Ext() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE;
to_arm_register = (op == '1'); t = UInt(Rt); n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

( FEAT_FP16 )

```
| 15 14 13 12|11 10 9 8|7 6 5 4|3 |0|15 12|11 10 9 8|7 6 5 4|3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 1 1 0 1 1 0 0 0 0| op| Vn| Rt| 1 0 0 1| N[0][0][1]| 0[0][0][0][0]|
```

From general-purpose register variant

Applies when op == 0.

VMOV{<c>}{<q>}.F16 <Sn>, <Rt>
To general-purpose register variant

Applies when op == 1.

VMOV{<c>}{<q>}.F16 <Rt>, <Sn>

Decode for all variants of this encoding

if !HaveFP16Ext() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
to_arm_register = (op == '1');  t = UInt(Rt);  n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<Rt> Is the general-purpose register that <Sn> will be transferred to or from, encoded in the "Rt" field.
<Sn> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Vn:N" field.
<c> See Standard assembler syntax fields on page F1-7228.
<q> See Standard assembler syntax fields on page F1-7228.

Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
if to_arm_register then
   R[t] = Zeros(16) : S[n]<15:0>;
else
   S[n] = Zeros(16) : R[t]<15:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.134  VMOV (immediate)

Copy immediate value to a SIMD&FP register places an immediate constant into every element of the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when \( Q == 0 \).

\texttt{VMOV\{<c>\}\{<q>\}.I32 <Dd>, #<imm>} \n
128-bit SIMD vector variant

Applies when \( Q == 1 \).

\texttt{VMOV\{<c>\}\{<q>\}.I32 <Qd>, #<imm>}

\texttt{Decode for all variants of this encoding}\n
\begin{itemize}
  \item \( \text{if op} == '0' \text{ && cmode}<0> == '1' \text{ && cmode}<3:2> != '11' \text{ then SEE "VORR (immediate)"; } \)
  \item \( \text{if op} == '1' \text{ && cmode} != '1110' \text{ then SEE "Related encodings"; } \)
  \item \( \text{if Q} == '1' \text{ && Vd}<b> == '1' \text{ then UNDEFINED; } \)
  \item single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
  \item d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
\end{itemize}

A2

Half-precision scalar variant

Applies when \( \text{size} == 01 \).

\texttt{VMOV\{<c>\}\{<q>\}.F16 <Sd>, #<imm>}

Single-precision scalar variant

Applies when \( \text{size} == 10 \).

\texttt{VMOV\{<c>\}\{<q>\}.F32 <Sd>, #<imm>}

Double-precision scalar variant

Applies when \( \text{size} == 11 \).

\texttt{VMOV\{<c>\}\{<q>\}.F64 <Dd>, #<imm>
Decode for all variants of this encoding

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
single_register = (size != '11'); advsimd = FALSE;
bits(16) imm16;
bits(32) imm32;
bits(64) imm64;
integer d;
integer regs;
case size of
  when '01' d = UInt(Vd:D);  imm16 = VFPExpandImm(imm4H:imm4L, 16); imm32 = Zeros(16) : imm16;
  when '10' d = UInt(Vd:D);  imm32 = VFPExpandImm(imm4H:imm4L, 32);
  when '11' d = UInt(D:Vd);  imm64 = VFPExpandImm(imm4H:imm4L, 64);  regs = 1;

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

A3

64-bit SIMD vector variant

Applies when Q == 0.
VMOV{<c>}{<q>}.I16 <Dd>, #<imm>

128-bit SIMD vector variant

Applies when Q == 1.
VMOV{<c>}{<q>}.I16 <Qd>, #<imm>

Decode for all variants of this encoding

if op == '0' && cmode<0> == '1' && cmode<3:2> != '1' then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;

A4
64-bit SIMD vector variant
Applies when $Q = 0$.

VMOV{<c>}{<q>}.<dt> <Dd>, #<imm>

128-bit SIMD vector variant
Applies when $Q = 1$.

VMOV{<c>}{<q>}.<dt> <Qd>, #<imm>

**Decode for all variants of this encoding**

if $op == '0'$ & cmode<0> == '1' & cmode<3:2> != '11' then SEE "VORR (immediate)"
if $op == '1'$ & cmode != '1110' then SEE "Related encodings"
if $Q == '1'$ & Vd<0> == '1' then UNDEFINED;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDEndExpImm(op, cmode, i:imm3:imm4);
$d = UInt(D:Vd)$; regs = if $Q == '0'$ then 1 else 2;

A5

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 12</th>
<th>11 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 i i</td>
<td>D 0 0 0</td>
</tr>
<tr>
<td>cmode op</td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant
Applies when $Q = 0$.

VMOV{<c>}{<q>}.I64 <Dd>, #<imm>

128-bit SIMD vector variant
Applies when $Q = 1$.

VMOV{<c>}{<q>}.I64 <Qd>, #<imm>

**Decode for all variants of this encoding**

if $op == '0'$ & cmode<0> == '1' & cmode<3:2> != '11' then SEE "VORR (immediate)"
if $op == '1'$ & cmode != '1110' then SEE "Related encodings"
if $Q == '1'$ & Vd<0> == '1' then UNDEFINED;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDEndExpImm(op, cmode, i:imm3:imm4);
$d = UInt(D:Vd)$; regs = if $Q == '0'$ then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 2 0</th>
<th>15 12 11 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
<td>i i i i D 0 0 0</td>
<td>imm3</td>
</tr>
<tr>
<td>cmode op</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant
Applies when $Q = 0$.

VMOV{<c>}{<q>}.I32 <Dd>, #<imm>
128-bit SIMD vector variant

Applies when Q == 1.

\[
\text{VMOV} \{<c>\} \{<q>\}.I32 \ <Qd>, \ #<imm>
\]

Decode for all variants of this encoding

if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;

T2

<table>
<thead>
<tr>
<th>15 14 13 12 11 10  9  8  7  6  5  4  3  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 1 0 1 1 0 1</td>
</tr>
</tbody>
</table>

Half-precision scalar variant

Applies when size == 01.

\[
\text{VMOV} \{<c>\} \{<q>\}.F16 \ <Sd>, \ #<imm>
\]

Single-precision scalar variant

Applies when size == 10.

\[
\text{VMOV} \{<c>\} \{<q>\}.F32 \ <Sd>, \ #<imm>
\]

Double-precision scalar variant

Applies when size == 11.

\[
\text{VMOV} \{<c>\} \{<q>\}.F64 \ <Dd>, \ #<imm>
\]

Decode for all variants of this encoding

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
single_register = (size != '11'); advsimd = FALSE;
b_bits(16) imm16;
b_bits(32) imm32;
b_bits(64) imm64;
integer d;
integer regs;
case size of
when '01' d = UInt(Vd:D);  imm16 = VFPExpandImm(imm4H:imm4L, 16); imm32 = Zeros(16) : imm16;
  when '10' d = UInt(Vd:D);  imm32 = VFPExpandImm(imm4H:imm4L, 32);
  when '11' d = UInt(D:Vd);  imm64 = VFPExpandImm(imm4H:imm4L, 64);  regs = 1;

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
T3

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 0</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>0 0 0</td>
<td>imm3</td>
<td>Vd</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VMOV{<c>}{<q>}.I16 <Dd>, #<imm>

128-bit SIMD vector variant

Applies when Q == 1.

VMOV{<c>}{<q>}.I16 <Qd>, #<imm>

Decode for all variants of this encoding

if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode == '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd); regs = if Q == '0' then 1 else 2;

t4

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 0</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>0 0 0</td>
<td>imm3</td>
<td>Vd</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VMOV{<c>}{<q>}.<dt> <Dd>, #<imm>

128-bit SIMD vector variant

Applies when Q == 1.

VMOV{<c>}{<q>}.<dt> <Qd>, #<imm>

Decode for all variants of this encoding

if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode == '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd); regs = if Q == '0' then 1 else 2;

t5

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 0</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>0 0 0</td>
<td>imm3</td>
<td>Vd</td>
</tr>
</tbody>
</table>
**64-bit SIMD vector variant**

Applies when Q == 0.

VMOV{<c>}{<q>}.I64 <Dd>, #<imm>

**128-bit SIMD vector variant**

Applies when Q == 1.

VMOV{<c>}{<q>}.I64 <Qd>, #<imm>

**Decode for all variants of this encoding**

if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDEncodeImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;

**Notes for all encodings**

Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.

**Assembler symbols**

- `<c>` For encoding A1, A3, A4 and A5: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
  
  For encoding A2, T1, T2, T3, T4 and T5: see Standard assembler syntax fields on page F1-7228.

- `<q>` See Standard assembler syntax fields on page F1-7228.

- `<dt>` The data type, encoded in the "cmode" field. It can have the following values:
  
  I32 when cmode = 110x
  
  I8 when cmode = 1110
  
  F32 when cmode = 1111

- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

- `<imm>` For encoding A1, A3, A4, A5, T1, T3, T4 and T5: is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<imm>`, see Modified immediate constants in T32 and A32 Advanced SIMD instructions on page F1-7245.
  
  For encoding A2 and T2: is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "imm4H:imm4L". For details of the range of constants available and the encoding of `<imm>`, see Modified immediate constants in T32 and A32 floating-point instructions on page F1-7246.

**Operation for all encodings**

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
  if single_register then
    S[d] = imm32;
  else
for r = 0 to regs-1
  D[d+r] = imm64;

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.135  VMOV (register)

Copy between FP registers copies the contents of one FP register to another.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A2

Single-precision scalar variant

Applies when size == 10.

VMOV{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar variant

Applies when size == 11.

VMOV{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;

single_register = (size == '10');  advsimd = FALSE;
integer d;
integer m;
integer regs;
if single_register then
  d = UInt(Vd:D);  m = UInt(Vm:M);
else
  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = 1;

T2

Single-precision scalar variant

Applies when size == 10.

VMOV{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar variant

Applies when size == 11.

VMOV{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;

single_register = (size == '10');  advsimd = FALSE;
integer d;
integer m;
integer regs;
if single_register then
    d = UInt(Vd:D);  m = UInt(Vm:M);
else
    d = UInt(D:Vd);  m = UInt(M:Vm);  regs = 1;

Assembler symbols

<\>  See Standard assembler syntax fields on page F1-7228.
<\>  See Standard assembler syntax fields on page F1-7228.
<\d>  Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<\m>  Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<\D>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<\M>  Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if single_register then
        S[d] = S[m];
    else
        for r = 0 to regs-1
            D[d+r] = D[m+r];

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
   — The values of the data supplied in any of its registers.
   — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
   — The values of the data supplied in any of its registers.
   — The values of the NZCV flags.
F6.1.136  VMOV (register, SIMD)

Copy between SIMD registers copies the contents of one SIMD register to another

This instruction is an alias of the VORR (register) instruction. This means that:

- The encodings in this description are named to match the encodings of VORR (register).
- The description of VORR (register) gives the operational pseudocode for this instruction.

### A1

| [31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8|7 6 5 4 3|0 1 0 0 0 D 1 0 0 0 1 N Q M 1 | Vm |

**64-bit SIMD vector variant**

Applies when Q == 0.

\[
\text{V MOV}\{<c>\}{<q>\}{.<dt>} <Dd>, <Dm>
\]

is equivalent to

\[
\text{V ORR}\{<c>\}{<q>\}{.<dt>} <Dd>, <Dm>, <Dm>
\]

and is the preferred disassembly when N:Vn == M:Vm.

**128-bit SIMD vector variant**

Applies when Q == 1.

\[
\text{V MOV}\{<c>\}{<q>\}{.<dt>} <Qd>, <Qm>
\]

is equivalent to

\[
\text{V ORR}\{<c>\}{<q>\}{.<dt>} <Qd>, <Qm>, <Qm>
\]

and is the preferred disassembly when N:Vn == M:Vm.

### T1

| [15 14 13 12|11 10 9 8|7 6 5 4 3|0 15 12|11 10 9 8|7 6 5 4 3|0 1 0 0 0 D 1 0 0 0 1 N Q M 1 | Vm |

**64-bit SIMD vector variant**

Applies when Q == 0.

\[
\text{V MOV}\{<c>\}{<q>\}{.<dt>} <Dd>, <Dm>
\]

is equivalent to

\[
\text{V ORR}\{<c>\}{<q>\}{.<dt>} <Dd>, <Dm>, <Dm>
\]

and is the preferred disassembly when N:Vn == M:Vm.

**128-bit SIMD vector variant**

Applies when Q == 1.

\[
\text{V MOV}\{<c>\}{<q>\}{.<dt>} <Qd>, <Qm>
\]
is equivalent to

\[ \text{VORR}\{\langle c\rangle\{\langle q\rangle\{.\langle dt\rangle\}\langle Qd\rangle, \langle Qm\rangle, \langle Qm\rangle} \]

and is the preferred disassembly when \(N:Vn = M:Vm\).

**Assembler symbols**

\(\langle c\rangle\)  
For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

\(\langle q\rangle\)  
See Standard assembler syntax fields on page F1-7228.

\(\langle dt\rangle\)  
An optional data type. \(\langle dt\rangle\) must not be F64, but it is otherwise ignored.

\(\langle Qd\rangle\)  
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(\langle Qd\rangle*2\).

\(\langle Qm\rangle\)  
Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as \(\langle Qm\rangle*2\).

\(\langle Dd\rangle\)  
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(\langle Dm\rangle\)  
Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

**Operation for all encodings**

The description of VORR (register) gives the operational pseudocode for this instruction.
F6.1.137  VMOV (general-purpose register to scalar)

Copy a general-purpose register to a vector element copies a byte, halfword, or word from a general-purpose register into an Advanced SIMD scalar.

On a Floating-point-only system, this instruction transfers one word to the upper or lower half of a double-precision floating-point register from a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see Advanced SIMD scalars on page F1-7254.

Depending on settings in the CPACR, NSACR, HCPTCR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

```
A1 variant

VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>
```

**Decode for this encoding**

```java
boolean advsimd;
integer esize;
integer index;
case opc1:opc2 of
  when '1xxx'  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
  when '0xx1'  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
  when '0x00'  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
  when '0x10'  UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T1

```
T1 variant

VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>
```

**Decode for this encoding**

```java
boolean advsimd;
integer esize;
integer index;
case opc1:opc2 of
  when '1xxx'  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
  when '0xx1'  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
  when '0x00'  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
  when '0x10'  UNDEFINED;
```
d = UInt(D:Vd);  t = UInt(Rt);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

Notes for all encodings
For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<cf> See Standard assembler syntax fields on page F1-7228.
<cf><cf> See Standard assembler syntax fields on page F1-7228.
<cf><cf> The data size. It must be one of:
\begin{enumerate}
\item 8 Encoded as \texttt{opc1<1> = 1}. [x] is encoded in \texttt{opc1<0>}, \texttt{opc2}.
\item 16 Encoded as \texttt{opc1<1> = 0}, \texttt{opc2<0> = 1}. [x] is encoded in \texttt{opc1<0>}, \texttt{opc2<1>}.
\item 32 Encoded as \texttt{opc1<1> = 0}, \texttt{opc2 = 0b00}. [x] is encoded in \texttt{opc1<0>}.
\item omitted Equivalent to 32.
\end{enumerate}
<cf><cf><cf> The scalar. The register \texttt{<Dd>} is encoded in D:Vd. For details of how [x] is encoded, see the description of \texttt{<size>}.
<cf><cf> The source general-purpose register.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    Elem[D[d],index,esize] = R[t]<esize-1:0>;

Operational information
If CPSR.DIT is 1 and this instruction passes its condition execution check:
\begin{itemize}
\item The execution time of this instruction is independent of:
    \begin{itemize}
    \item The values of the data supplied in any of its registers.
    \item The values of the NZCV flags.
    \end{itemize}
\item The response of this instruction to asynchronous exceptions does not vary based on:
    \begin{itemize}
    \item The values of the data supplied in any of its registers.
    \item The values of the NZCV flags.
    \end{itemize}
\end{itemize}
F6.1.138 VMOV (between general-purpose register and single-precision)

Copy a general-purpose register to or from a 32-bit SIMD&FP register. This instruction transfers the value held in a 32-bit SIMD&FP register to a general-purpose register, or the value held in a general-purpose register to a 32-bit SIMD&FP register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

From general-purpose register variant

Applies when \( op = 0 \).

\[ \text{VMOV}\{<c>\}{<q>} <Sn>, <Rt> \]

To general-purpose register variant

Applies when \( op = 1 \).

\[ \text{VMOV}\{<c>\}{<q>} <Rt>, <Sn> \]

Decode for all variants of this encoding

\[
\text{to} \_ \text{arm} \_ \text{register} = (op == '1'); \ t = \text{UInt}(Rt); \ n = \text{UInt}(Vn:N);
\]

if \( t = 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

T1

From general-purpose register variant

Applies when \( op = 0 \).

\[ \text{VMOV}\{<c>\}{<q>} <Sn>, <Rt> \]

To general-purpose register variant

Applies when \( op = 1 \).

\[ \text{VMOV}\{<c>\}{<q>} <Rt>, <Sn> \]

Decode for all variants of this encoding

\[
\text{to} \_ \text{arm} \_ \text{register} = (op == '1'); \ t = \text{UInt}(Rt); \ n = \text{UInt}(Vn:N);
\]

if \( t = 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<rt>  Is the general-purpose register that <sn> will be transferred to or from, encoded in the "Rt" field.

<sn>  Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Vn:N" field.

<->  See Standard assembler syntax fields on page F1-7228.

<q>  See Standard assembler syntax fields on page F1-7228.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_register then
        R[t] = S[n];
    else
        S[n] = R[t];

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

•  The execution time of this instruction is independent of:
    —  The values of the data supplied in any of its registers.
    —  The values of the NZCV flags.

•  The response of this instruction to asynchronous exceptions does not vary based on:
    —  The values of the data supplied in any of its registers.
    —  The values of the NZCV flags.
F6.1.139 VMOV (scalar to general-purpose register)

Copy a vector element to a general-purpose register with sign or zero extension copies a byte, halfword, or word from an Advanced SIMD scalar to a general-purpose register. Bytes and halfwords can be either zero-extended or sign-extended.

On a Floating-point-only system, this instruction transfers one word from the upper or lower half of a double-precision floating-point register to a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see Advanced SIMD scalars on page F1-7254.

Depending on settings in the CPACR, NSACR, HCPTLR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x]>

Decode for this encoding

boolean advsimd;
integer esize;
integer index;
case U:opc1:opc2 of
  when 'x1xxx'  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
  when 'x0xx1'  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
  when '00x00'  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
  when '10x00'  UNDEFINED;
  when 'x0x10'  UNDEFINED;
  t = UInt(Rt);  n = UInt(N:Vn);  unsigned = (U == '1');
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

T1

VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x]>

Decode for this encoding

boolean advsimd;
integer esize;
integer index;
case U:opc1:opc2 of
  when 'x1xxx'  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
  when 'x0xx1'  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
  when '00x00'  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
  when '10x00'  UNDEFINED;
when 'x0x10' UNDEFINED;

\[
t = \text{UInt}(R_t); \quad n = \text{UInt}(N;V_n); \quad \text{unsigned} = (U == '1');
\]

if \( t == 15 \) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

### Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 _Architectural Constraints on UNPREDICTABLE Behaviors_.

### Assembler symbols

- \(<c>\) See _Standard assembler syntax fields on page F1-7228_.
- \(<q>\) See _Standard assembler syntax fields on page F1-7228_.
- \(<dt>\) The data type. It must be one of:
  - \(S8\) Encoded as \(U = 0, \text{opc1}<1> = 1, [x] \) is encoded in \(\text{opc1}<0>, \text{opc2}\).
  - \(S16\) Encoded as \(U = 0, \text{opc1}<1> = 0, \text{opc2}<0> = 1, [x] \) is encoded in \(\text{opc1}<0>, \text{opc2}<1>\).
  - \(U8\) Encoded as \(U = 1, \text{opc1}<1> = 1, [x] \) is encoded in \(\text{opc1}<0>, \text{opc2}\).
  - \(U16\) Encoded as \(U = 1, \text{opc1}<1> = 0, \text{opc2}<0> = 1, [x] \) is encoded in \(\text{opc1}<0>, \text{opc2}<1>\).
  - \(32\) Encoded as \(U = 0, \text{opc1}<1> = 0, \text{opc2} = 0b00, [x] \) is encoded in \(\text{opc1}<0>\).
  - \(\text{omitted}\) Equivalent to \(32\).
- \(<Rt>\) The destination general-purpose register.
- \(<Dn[x]>\) The scalar. For details of how \([x]\) is encoded see the description of \(<dt>\).

### Operation for all encodings

if \( \text{ConditionPassed()} \) then

\[
\text{EncodingSpecificOperations();} \quad \text{CheckAdvSIMDorVFPEnabled(TRUE, advsimd)};
\]

if unsigned then

\[
R[t] = \text{ZeroExtend(Elem[D[n],index,esize], 32)};
\]

else

\[
R[t] = \text{SignExtend(Elem[D[n],index,esize], 32)};
\]

### Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
### F6.1.140 VMOV (between two general-purpose registers and two single-precision registers)

Copy two general-purpose registers to a pair of 32-bit SIMD&FP registers transfers the contents of two consecutively numbered single-precision Floating-point registers to two general-purpose registers, or the contents of two general-purpose registers to a pair of single-precision Floating-point registers. The general-purpose registers do not have to be contiguous.

Depending on settings in the CPACR, NSACR, HCPTTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support on page G1-8996*.

A1

```
<table>
<thead>
<tr>
<th>31 28 27 26 25 24 23 22 21 20</th>
<th>16 15 12 11 10 9 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>1 1 0 0 0 1 0 op</td>
</tr>
<tr>
<td>cond</td>
<td>Rt2 Rt 1 0 1 0 0 0 M 1 Vm</td>
</tr>
</tbody>
</table>
```

**From general-purpose registers variant**

Applies when \( \text{op} == 0 \).

\[
\text{VMOV}\{<c>\}{<q>} <Sm>, <Sm1>, <Rt>, <Rt2>
\]

**To general-purpose registers variant**

Applies when \( \text{op} == 1 \).

\[
\text{VMOV}\{<c>\}{<q>} <Rt>, <Rt2>, <Sm>, <Sm1>
\]

**Decode for all variants of this encoding**

\[
\text{to}_{\text{arm}}\text{_registers} = (\text{op} == '1'); \text{ t } = \text{UInt}(\text{Rt}); \text{ t2 } = \text{UInt}(\text{Rt2}); \text{ m } = \text{UInt}(\text{Vm}\text{M});
\]

\[
\text{if t } == 15 \text{ || t2 } == 15 \text{ || m } == 31 \text{ then UNPREDICTABLE;}
\]

\[
\text{if to}_{\text{arm}}\text{_registers} \&\& \text{ t } == \text{ t2 } \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( \text{to}_{\text{arm}}\text{_registers} \&\& \text{ t } == \text{ t2} \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

If \( \text{m} == 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

T1

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 15 12 11 10 9 8 7 6 5 4 3 0 |
|-----------------------|-----------------|-----------------|-----------------|
|                         | 1 1 1 0 1 1 0 0 0 | 1 0 op          |
|                         | Rt2 Rt 1 0 1 0 0 0 | M 1 Vm          |
```
From general-purpose registers variant

Applies when op == 0.

VMOV{<c>}{<q>} <Sm>, <Sm1>, <Rt>, <Rt2>

To general-purpose registers variant

Applies when op == 1.

VMOV{<c>}{<q>} <Rt>, <Rt2>, <Sm>, <Sm1>

Decode for all variants of this encoding

to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(Vm:M);
if t == 15 || t2 == 15 || m == 31 then UNPREDICTABLE;
if to_arm_registers && t == t2 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If to_arm_registers && t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

If m == 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VMOV (between two general-purpose registers and two single-precision registers) on page K1-11579.

Assembler symbols

<Rt2> Is the second general-purpose register that <Sm1> will be transferred to or from, encoded in the "Rt2" field.

<Rt> Is the first general-purpose register that <Sm> will be transferred to or from, encoded in the "Rt" field.

<Sm1> Is the 32-bit name of the second SIMD&FP register to be transferred. This is the next SIMD&FP register after <Sm>.

<Sm> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Vm:M" field.

<> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
if to_arm_registers then
    R[t] = S[m];
    R[t2] = S[m+1];
else
    S[m] = R[t];
    S[m+1] = R[t2];

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.141 VMOVL

Vector Move Long takes each element in a doubleword vector, sign or zero-extends them to twice their original length, and places the results in a quadword vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>

Decode for this encoding

if imm3H == '000' then SEE "Related encodings";
if imm3H == '001' & & imm3H != '010' & & imm3H != '100' then SEE "VSHLL";
if Vd<0> == '1' then UNDEFINED;
esize = 8 * UInt(imm3H);
unsigned = (U == '1');  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);

T1

VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>

Decode for this encoding

if imm3H == '000' then SEE "Related encodings";
if imm3H != '001' & & imm3H != '010' & & imm3H != '100' then SEE "VSHLL";
if Vd<0> == '1' then UNDEFINED;
esize = 8 * UInt(imm3H);
unsigned = (U == '1');  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);

Notes for all encodings

Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.

Assembler symbols

For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields* on page F1-7228.

<q>See *Standard assembler syntax fields* on page F1-7228.

<dt>Is the data type for the elements of the operand, encoded in the "U:imm3H" field. It can have the following values:

- **S8** when \( U = 0 \), \( \text{imm3H} = 001 \)
- **S16** when \( U = 0 \), \( \text{imm3H} = 010 \)
- **S32** when \( U = 0 \), \( \text{imm3H} = 100 \)
- **U8** when \( U = 1 \), \( \text{imm3H} = 001 \)
- **U16** when \( U = 1 \), \( \text{imm3H} = 010 \)
- **U32** when \( U = 1 \), \( \text{imm3H} = 100 \)

<Qd>Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<Qd>*2.\n
<Dm>Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```java
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Int(Elem[Din[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.142   VMOVN

Vector Move and Narrow copies the least significant half of each element of a quadword vector into the corresponding elements of a doubleword vector.

The operand vector elements can be any one of 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

This instruction is used by the pseudo-instructions VRSHRN (zero) and VSHRN (zero). The pseudo-instruction is never the preferred disassembly.

A1

```
[31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 12|11 10 9 8 7 6 5 4 |3 0 |
1 1 1 1 0 1 1 1 D 1 1 size 1 0 Vd 0 0 1 0 0 0 M 0 Vm]
```

A1 variant

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Decode for this encoding

```python
if size == '11' then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

```
[15 14 13 12|11 10 9 8 7 6 5 4|3 2 1 0|15 12|11 10 9 8 7 6 5 4 |3 0 |
1 1 1 1 1 1 1 1 D 1 1 size 1 0 Vd 0 0 1 0 0 0 M 0 Vm]
```

T1 variant

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Decode for this encoding

```python
if size == '11' then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm);
```

Assembler symbols

<> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:
   I16 when size = 00
   I32 when size = 01
   I64 when size = 10
   The encoding size = 11 is reserved.

<dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        Elem[D[d],e,esize] = Elem[Qin[m>>>1],e,2+esize<esize-1:0>];
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.143 VMOVX

Vector Move extraction. This instruction copies the upper 16 bits of the 32-bit source SIMD&FP register into the lower 16 bits of the 32-bit destination SIMD&FP register, while clearing the remaining bits to zero.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

(FEAT_FP16)

\[
\begin{array}{cccccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
D & Vd & M & Vm
\end{array}
\]

A1 variant

VMOVX{<q>}.F16 <Sd>, <Sm>

Decode for this encoding

if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); m = UInt(Vm:M);

T1

(FEAT_FP16)

\[
\begin{array}{cccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
D & Vd & M & Vm
\end{array}
\]

T1 variant

VMOVX{<q>}.F16 <Sd>, <Sm>

Decode for this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); m = UInt(Vm:M);

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    S[d] = Zeros(16) : S[m]<31:16>;
F6.1.144 VMRS

Move SIMD&FP Special register to general-purpose register moves the value of an Advanced SIMD and floating-point System register to a general-purpose register. When the specified System register is the FPSCR, a form of the instruction transfers the FPSCR. {N, Z, C, V} condition flags to the APSR. {N, Z, C, V} condition flags.

Depending on settings in the CPACR, NSACR, HCPR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

When these settings permit the execution of Advanced SIMD and floating-point instructions, if the specified floating-point System register is not the FPSCR, the instruction is UNDEFINED if executed in User mode.

In an implementation that includes EL2, when HCR.TID0 is set to 1, any VMRS access to FPSID from a Non-secure EL1 mode that would be permitted if HCR.TID0 was set to 0 generates a Hyp Trap exception. For more information, see ID group 0, Primary device identification registers on page G1-9019.

For simplicity, the VMRS pseudocode does not show the possible trap to Hyp mode.

A1

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>t=1111</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>reg</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 variant

VMRS<cc><cq> <Rt>, <spec_reg>

Decode for this encoding

\[
t = \text{UInt}(Rt);
\]

if !(reg IN {"000x", "0101", "011x", "1000"}) then UNPREDICTABLE;

if t == 15 && reg != '0001' then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

CONSTRANGED UNPREDICTABLE behavior

If !(reg IN {"000x", "0101", "011x", "1000"}), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers an UNKNOWN value to the specified target register. When the Rt field holds the value 0b1111, the specified target register is the APSR. {N, Z, C, V} bits, and these bits become UNKNOWN. Otherwise, the specified target register is the register specified by the Rt field, R0 - R14.

T1

| 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  0 |  15 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | reg | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

T1 variant

VMRS<cc><cq> <Rt>, <spec_reg>
### Decode for this encoding

\[
t = \text{UInt}(\text{Rt});
\]

\[
\text{if } \neg(\text{reg} \in \{'000x', '0101', '011x', '1000'\}) \text{ then UNPREDICTABLE;} \\
\text{if } t == 15 \&\& \text{reg} \neq '0001' \text{ then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13}
\]

### CONSTRAINED UNPREDICTABLE behavior

If \( \neg(\text{reg} \in \{'000x', '0101', '011x', '1000'\}) \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers an UNKNOWN value to the specified target register. When the Rt field holds the value 0b1111, the specified target register is the APSR\_{N, Z, C, V} bits, and these bits become UNKNOWN. Otherwise, the specified target register is the register specified by the Rt field, R0 - R14.

### Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

### Assembler symbols

- \(<c>\) See Standard assembler syntax fields on page F1-7228.
- \(<q>\) See Standard assembler syntax fields on page F1-7228.
- \(<\text{Rt}>\) Is the general-purpose destination register, encoded in the "Rt" field. Is one of:
  - R0-R14 General-purpose register.
  - APSR\_nzc\_v Permited only when \(<\text{spec_reg}>\) is FPSCR. Encoded as 0b1111. The instruction transfers the FPSCR\_\{N, Z, C, V\} condition flags to the APSR\_\{N, Z, C, V\} condition flags.
- \(<\text{spec_reg}>\) Is the source Advanced SIMD and floating-point System register, encoded in the "reg" field. It can have the following values:
  - FPSID when reg = 0000
  - FPSCR when reg = 0001
  - MVFR2 when reg = 0101
  - MVFR1 when reg = 0110
  - MVFR0 when reg = 0111
  - FPEXC when reg = 1000

The following encodings are UNPREDICTABLE:

- reg = 001x.
- reg = 0100.
- reg = 1001.
- reg = 101x.
- reg = 11xx.

### Operation for all encodings

\[
\text{if } \text{ConditionPassed()} \text{ then} \\
\text{EncodingSpecificOperations();} \\
\text{if } \text{reg} == '0001' \text{ then} \quad \text{// FPSCR} \\
\text{CheckVFPEnabled(TRUE);} \\
\text{if } t == 15 \text{ then}
\]
else
  R[t] = FPSCR;
elsif PSTATE.EI == EI0 then
  UNDEFINED;                        // Non-FPSCR registers accessible only at PL1 or above
else
  CheckVFPEnabled(FALSE);           // Non-FPSCR registers are not affected by FPEXC.EN
  AArch32.CheckAdvSIMDOrFPRegisterTraps(reg);
  case reg of
  when '0000'  R[t] = FPSID;
  when '0101'  R[t] = MVFR2;
  when '0110'  R[t] = MVFR1;
  when '0111'  R[t] = MVFR0;
  when '1000'  R[t] = FPEXC;
  otherwise  Unreachable();   // Dealt with above or in encoding-specific pseudocode
F6.1.145 VMSR

Move general-purpose register to SIMD&FP Special register moves the value of a general-purpose register to a floating-point System register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

When these settings permit the execution of Advanced SIMD and floating-point instructions:

- If the specified floating-point System register is FPSID or FPEXC, the instruction is UNDEFINED if executed in User mode.
- If the specified floating-point System register is the FPSID and the instruction is executed in a mode other than User mode, the instruction is ignored.

A1

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !=111 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | reg | | | | | | | | | | | | | | | | | | |

A1 variant

VMSR{<c>}{<q>} <spec_reg>, <Rt>

Decode for this encoding

\[
t = \text{UInt}(Rt);
\]
\[
\text{if } !(\text{reg} \text{ IN } \{‘000x’\}) \&\& \text{reg} \neq ‘1000’ \text{ then}
\]
\[
\text{Constraint } c = \text{ConstrainUnpredictable(Unpredictable_VMSR)};
\]
\[
\text{assert } c \text{ IN } \{\text{Constraint_UNDEF, Constraint_NOP}\};
\]
\[
\text{case } c \text{ of}
\]
\[
\text{when } \text{Constraint_UNDEF}
\]
\[
\text{UNDEFINED};
\]
\[
\text{when } \text{Constraint_NOP}
\]
\[
\text{EndOfInstruction>();}
\]
\[
\text{if } t = 15 \text{ then UNPREDICTABLE; } // \text{Armv8-A removes UNPREDICTABLE for R13}
\]

CONSTRAINED UNPREDICTABLE behavior

If \text{reg} \neq ‘000x’ \&\& \text{reg} \neq ‘1000’, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers the value in the general-purpose register to one of the allocated registers accessible using VMSR at the same Exception level.

T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | reg | | | | | | | | | | | | | | | | | | |

T1 variant

VMSR{<c>}{<q>} <spec_reg>, <Rt>
**Decode for this encoding**

```plaintext
t = UInt(Rt);
if !(reg IN \{"000x"\}) && reg != '1000' then
    Constraint c = ConstrainUnpredictable(Unpredictable_VMSR);
    assert c IN \{Constraint_UNDEF, Constraint_NOP\};
    case c of
        when Constraint_UNDEF
            UNDEFINED;
        when Constraint_NOP
            EndOfInstruction();
    if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

**CONSTRAINED UNPREDICTABLE behavior**

If reg != '000x' && reg != '1000', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers the value in the general-purpose register to one of the allocated registers accessible using VMSR at the same Exception level.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<spec_reg>` Is the destination Advanced SIMD and floating-point System register, encoded in the "reg" field. It can have the following values:
  - FPSID when reg = 0000
  - FPSCR when reg = 0001
  - FPEXC when reg = 1000
  The following encodings are UNPREDICTABLE:
    - reg = 001x.
    - reg = 01xx.
    - reg = 1001.
    - reg = 101x.
    - reg = 11xx.
- `<Rt>` Is the general-purpose source register, encoded in the "Rt" field.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then // FPSCR
        CheckVFPEnabled(TRUE);
        FPSCR = R[t];
    elsif PSTATE.EL == EL0 then
        UNDEFINED; // Non-FPSCR registers accessible only at PL1 or above
    else
```
CheckVFPEnable(FALSE); // Non-FPSCR registers are not affected by FPEXC.EN

case reg of
  when '0000' // VMSR access to FPSID is ignored
    when '1000' FPEXC = R[t];
  otherwise    Unreachable(); // Dealt with above or in encoding-specific pseudocode
F6.1.146 VMUL (floating-point)

Vector Multiply multiplies corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>0</td>
<td>sz</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>N</td>
<td>Q</td>
<td>M</td>
<td>1</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

A2

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | D | 0 | sz | Vn | Vd | 1 | 0 | size | N | 0 | M | 0 | Vm |

cond

Half-precision scalar variant

Applies when size == 01.

VMUL{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>

Single-precision scalar variant

Applies when size == 10.

VMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>

Double-precision scalar variant

Applies when size == 11.

VMUL{<c>}{<q>}.F64 {<Dd>,} <Dm>, <Dm>
Decode for all variants of this encoding

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>0</td>
<td>D 0 sz</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if sz == '1' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
  d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**T2**

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0</td>
<td>1 1 0</td>
<td>D</td>
<td>1 0</td>
<td>Vn</td>
<td>Vd</td>
<td>1 0</td>
<td>size</td>
<td>N</td>
<td>0</td>
</tr>
</tbody>
</table>

**Half-precision scalar variant**
Applies when size == 01.

\[\text{VMUL}\{<c>\}\{<q>\}.F16 \{<Sd>,\} <Sn>, <Sm>\]

**Single-precision scalar variant**
Applies when size == 10.

\[\text{VMUL}\{<c>\}\{<q>\}.F32 \{<Sd>,\} <Sn>, <Sm>\]

**Double-precision scalar variant**
Applies when size == 11.

\[\text{VMUL}\{<c>\}\{<q>\}.F64 \{<Dd>,\} <Dn>, <Dm>\]

**Decode for all variants of this encoding**

if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;

\[\text{advsimd} = \text{FALSE};\]
\[\text{integer } \text{esize};\]
\[\text{integer } d;\]
\[\text{integer } n;\]
\[\text{integer } m;\]
\[\text{case size of}\]
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**
If size == '01' && InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.

• The instruction executes as if it passes the Condition code check.

• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

\[<c>\] For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding A2, T1 and T2: see Standard assembler syntax fields on page F1-7228.

\[<q>\] See Standard assembler syntax fields on page F1-7228.

\[<dt>\] Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

\[F32 \quad \text{when } sz = 0\]
F16 when sz = 1

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                Elem[D[d+r],e,esize] = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRValue());
    else // VFP instruction
        case esize of
            when 16
                S[d] = Zeros(16) : FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            when 32
                S[d] = FPMul(S[n], S[m], FPSCR[]);
            when 64
                D[d] = FPMul(D[n], D[m], FPSCR[]);
F6.1.147 VMUL (integer and polynomial)

Vector Multiply multiplies corresponding elements in two vectors.

For information about multiplying polynomials, see Polynomial arithmetic over \{0, 1\} on page A1-54.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

\[
\begin{array}{cccccccccc|c}
1 & 1 & 1 & 0 & 0 & 1 & 0 & D & \text{size} & Vn & Vd & 1 & 0 & 0 & 1 & N & Q & M & 1 & Vm \\
\end{array}
\]

64-bit SIMD vector variant

Applies when Q == 0.

VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if size == '11' || (op == '1' && size != '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
polynomial = (op == '1');  long_destination = FALSE;
size = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

\[
\begin{array}{cccccccccc|c}
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & D & \text{size} & Vn & Vd & 1 & 0 & 0 & 1 & N & Q & M & 1 & Vm \\
\end{array}
\]

64-bit SIMD vector variant

Applies when Q == 0.

VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if size == '11' || (op == '1' && size != '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
polynomial = (op == '1');  long_destination = FALSE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Assembler symbols

<> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the elements of the operands, encoded in the "op:size" field. It can have the following values:
I8 when op = 0, size = 00
I16 when op = 0, size = 01
I32 when op = 0, size = 10
P8 when op = 1, size = 00

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            op2 = Elem[Din[m+r],e,esize];  op2val = Int(op2, unsigned);
            bits(2 * esize) product;
            if polynomial then
                product = PolynomialMult(op1,op2);
            else
                product = (op1val*op2val)<<2*esize-1:0;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = product;
            else
                Elem[D[d+r],e,esize] = product<esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
— The values of the NZCV flags.
F6.1.148   VMUL (by scalar)

Vector Multiply multiplies each element in a vector by a scalar, and places the results in a second vector.

For more information about scalars see Advanced SIMD scalars on page F1-7254.

Depending on settings in the CPACR, NSACR, and HCPtr registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VMUL{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>[<index>]

128-bit SIMD vector variant

Applies when Q == 1.

VMUL{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
floating_point = (F == '1'); long_destination = FALSE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
generate integer esize;
generate integer elements;
generate integer m;
generate integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

T1

64-bit SIMD vector variant

Applies when Q == 0.

VMUL{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>[<index>]

128-bit SIMD vector variant

Applies when Q == 1.

VMUL{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>[<index>]

This page is Non-Confidential
Decode for all variants of this encoding

if size == '11' then SEE "Related encodings";
if F == '1' & size == '01' & InITBlock() then UNPREDICTABLE;
if size == '00' || (F == '1' & size == '01' & HaveFP16Ext()) then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);

CONSTRANDED UNPREDICTABLE behavior

If F == '1' & size == '01' & InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

Assembler symbols

<>  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
    For encoding T1: see Standard assembler syntax fields on page F1-7228.

<>  See Standard assembler syntax fields on page F1-7228.
<dt>  Is the data type for the scalar and the elements of the operand vector, encoded in the "F:size" field. It can have the following values:
    I16   when F  = 0, size = 01
    I32   when F  = 0, size = 10
    F16   when F  = 1, size = 01
    F32   when F  = 1, size = 10

<qd>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <qd>*2.

<qn>  Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <qn>*2.

<dn>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<dn>  Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<dm>  Is the 64-bit name of the second SIMD&FP source register. When <dt> is I16 or F16, this is encoded in the "Vm<2:0>" field. Otherwise it is encoded in the "Vm" field.

<index>  Is the element index. When <dt> is I16 or F16, this is in the range 0 to 3 and is encoded in the "M:Vm<3>" field. Otherwise it is in the range 0 to 1 and is encoded in the "M" field.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                Elem[D[d+r],e,esize] = FPMul(op1, op2, StandardFPSCRValue());
            else
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = (op1val*op2val)<2*esize-1:0>;
                else
                    Elem[D[d+r],e,esize] = (op1val*op2val)<esize-1:0>;
```
F6.1.149 VMULL (integer and polynomial)

Vector Multiply Long multiplies corresponding elements in two vectors. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see Polynomial arithmetic over \{0, 1\} on page A1-54.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

**A1**

A1 variant

VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

if size == '11' then SEE "Related encodings";
unsigned = (U == '1'); polynomial = (op == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
if polynomial then
  if U == '1' || size == '01' then UNDEFINED;
  if size == '10' then // .p64
    if !HaveBit128PMULLExt() then UNDEFINED;
esize = 64; elements = 1;
if Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;

**T1**

T1 variant

VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

if size == '11' then SEE "Related encodings";
unsigned = (U == '1'); polynomial = (op == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
if polynomial then
  if U == '1' || size == '01' then UNDEFINED;
  if size == '10' then // .p64
    if !HaveBit128PMULLExt() then UNDEFINED;
esize = 64; elements = 1;
if Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;
**CONSTRAINED UNPREDICTABLE behavior**

If \( \text{op} == '1' \land \text{size} == '10' \land \text{InITBlock}() \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

**Assembler symbols**

\(<c>\)  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

\(<q>\)  See Standard assembler syntax fields on page F1-7228.

\(<dt>\)  Is the data type for the elements of the operands, encoded in the "op:U:size" field. It can have the following values:

- **S8** when \( \text{op} = 0, \text{U} = 0, \text{size} = 00 \)
- **S16** when \( \text{op} = 0, \text{U} = 0, \text{size} = 01 \)
- **S32** when \( \text{op} = 0, \text{U} = 0, \text{size} = 10 \)
- **U8** when \( \text{op} = 0, \text{U} = 1, \text{size} = 00 \)
- **U16** when \( \text{op} = 0, \text{U} = 1, \text{size} = 01 \)
- **U32** when \( \text{op} = 0, \text{U} = 1, \text{size} = 10 \)
- **P8** when \( \text{op} = 1, \text{U} = 0, \text{size} = 00 \)
- **P64** when \( \text{op} = 1, \text{U} = 0, \text{size} = 10 \)

\(<Qd>\)  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<Qd>\times 2\).

\(<Dn>\)  Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

\(<Dm>\)  Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

if \( \text{ConditionPassed}() \) then

   EncodingSpecificOperations();  CheckAdvSIMDEnabled();

   for \( r = 0 \) to \( \text{regs}-1 \)
      for \( e = 0 \) to \( \text{elements}-1 \)
         \( \text{op1} = \text{Elem}[\text{D}:(\text{n}+r),e,\text{esize}]; \text{op1val} = \text{Int}(\text{op1}, \text{unsigned}); \)
         \( \text{op2} = \text{Elem}[\text{D}:(\text{m}+r),e,\text{esize}]; \text{op2val} = \text{Int}(\text{op2}, \text{unsigned}); \)
         bits(2*\text{esize}) \text{ product};
         \text{if polynomial then}
            \text{product} = \text{PolynomialMult}(\text{op1}, \text{op2});
         \text{else}
            \text{product} = (\text{op1val} \times \text{op2val}) < 2*\text{esize}-1:0>;
         \text{if long_destination then}
            \text{Elem}[\text{Q}[d>>1],e,2*\text{esize}] = \text{product};
else
    Elem[D+d+r],e,esize] = product<esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.150 VMULL (by scalar)

Vector Multiply Long multiplies each element in a vector by a scalar, and places the results in a second vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see Advanced SIMD scalars on page F1-7254.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

```
A1 variant
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;  floating_point = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
integer esize;
integer elements;
integer index;
integer m;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

```
T1 variant
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;  floating_point = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
integer esize;
integer elements;
integer index;
integer m;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```
Notes for all encodings

Related encodings: See *Advanced SIMD data-processing* on page F3-7314 for the T32 instruction set, or *Advanced SIMD data-processing* on page F4-7422 for the A32 instruction set.

Assembler symbols

- `<c>`: For encoding A1: see *Standard assembler syntax fields* on page F1-7228. This encoding must be unconditional.
  
  For encoding T1: see *Standard assembler syntax fields* on page F1-7228.

- `<q>`: See *Standard assembler syntax fields* on page F1-7228.

- `<dt>`: Is the data type for the scalar and the elements of the operand vector, encoded in the "U:size" field. It can have the following values:
  
  - S16 when U = 0, size = 01
  - S32 when U = 0, size = 10
  - U16 when U = 1, size = 01
  - U32 when U = 1, size = 10

- `<Qd>`: Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.

- `<Dn>`: Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

- `<Dm>`: Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when `<dt>` is S16 or U16, otherwise the "Vm" field.

- `<index>`: Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when `<dt>` is S16 or U16, otherwise in range 0 to 1, encoded in the "M" field.

Operation for all encodings

```c
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                Elem[D[d+r],e,esize] = FPMul(op1, op2, StandardFPSCRValue());
            else
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = (op1val<op2val)<2*esize-1:0>;
                else
                    Elem[D[d+r],e,esize] = (op1val<op2val)<esize-1:0>;
            
        
    
```

F6-8585
F6.1.151 VMVN (immediate)

Vector Bitwise NOT (immediate) places the bitwise inverse of an immediate integer constant into every element of the destination register.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 12|11 8 |7 6 5 4 |3 0 |
|---|---|---|---|---|---|---|---|---|
| 1 1 1 1 0 0 1 | i 1 | D 0 0 0 | imm3 | Vd | 0 x | 0 | Q 1 | 1 | imm4 |

64-bit SIMD vector variant

Applies when Q == 0.

VMVN{<c>}{<q>}.I32 <Dd>, #<imm>

128-bit SIMD vector variant

Applies when Q == 1.

VMVN{<c>}{<q>}.I32 <Qd>, #<imm>

Decode for all variants of this encoding

if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;

A2

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 16|15 12|11 8 |7 6 5 4 |3 0 |
|---|---|---|---|---|---|---|---|---|
| 1 1 1 1 0 0 1 | i 1 | D 0 0 0 | imm3 | Vd | 1 0 | x | 0 | Q 1 | 1 | imm4 |

64-bit SIMD vector variant

Applies when Q == 0.

VMVN{<c>}{<q>}.I16 <Dd>, #<imm>

128-bit SIMD vector variant

Applies when Q == 1.

VMVN{<c>}{<q>}.I16 <Qd>, #<imm>

Decode for all variants of this encoding

if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
A3

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 16 15 12 11 8 7 6 5 4 3 2 0 |
|--------------------------|----------------|----------------|----------------|
| 1 1 1 1 0 0 0 1 1 1 0 0 0 | imm3 | D | 0 0 0 | Vd | 1 1 0 | x | 0 | Q | 1 | 1 | imm4 |

64-bit SIMD vector variant

Applies when Q == 0.

VMVN{<c>}{<q>}.I32 <Dd>, #<imm>

128-bit SIMD vector variant

Applies when Q == 1.

VMVN{<c>}{<q>}.I32 <Qd>, #<imm>

Decode for all variants of this encoding

if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm(1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 0 15 12 11 8 7 6 5 4 3 0 |
|--------------------------|----------------|----------------|----------------|
| 1 1 1 1 | 1 1 1 1 1 | 1 | D | 0 0 0 | imm3 | Vd | 0 | x | 0 | 0 | Q | 1 | 1 | imm4 |

64-bit SIMD vector variant

Applies when Q == 0.

VMVN{<c>}{<q>}.I32 <Dd>, #<imm>

128-bit SIMD vector variant

Applies when Q == 1.

VMVN{<c>}{<q>}.I32 <Qd>, #<imm>

Decode for all variants of this encoding

if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm(1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;

T2

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 0 15 12 11 8 7 6 5 4 3 0 |
|--------------------------|----------------|----------------|----------------|
| 1 1 1 1 | 1 1 1 1 1 | 1 | D | 0 0 0 | imm3 | Vd | 1 | 0 | x | 0 | 0 | Q | 1 | 1 | imm4 |
64-bit SIMD vector variant
Applies when Q == 0.

$$\text{VMVN}\{c\}\{q\}.I16 <Dd>, #<imm>$$

128-bit SIMD vector variant
Applies when Q == 1.

$$\text{VMVN}\{c\}\{q\}.I16 <Qd>, #<imm>$$

Decode for all variants of this encoding

```c
if (cmode<8> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDEncodedImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T3

<table>
<thead>
<tr>
<th>i5 i4 i3 i2</th>
<th>i1 i0</th>
<th>0</th>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 2</th>
<th>0</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm3</td>
<td>Vd</td>
<td>1 1 0 x</td>
</tr>
<tr>
<td>cmode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant
Applies when Q == 0.

$$\text{VMVN}\{c\}\{q\}.I32 <Dd>, #<imm>$$

128-bit SIMD vector variant
Applies when Q == 1.

$$\text{VMVN}\{c\}\{q\}.I32 <Qd>, #<imm>$$

Decode for all variants of this encoding

```c
if (cmode<8> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDEncodedImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

Notes for all encodings
Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.

Assembler symbols

- `<c>`
  For encoding A1, A2 and A3: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
  For encoding T1, T2 and T3: see Standard assembler syntax fields on page F1-7228.

- `<q>`
  See Standard assembler syntax fields on page F1-7228.

- `<qd>`
  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.`
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<imm> Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see Modified immediate constants in T32 and A32 Advanced SIMD instructions on page F1-7245.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = NOT(imm64);
```

### Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.152 VMVN (register)

Vector Bitwise NOT (register) takes a value from a register, inverts the value of each bit, and places the result in the destination register. The registers can be either doubleword or quadword.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15</th>
<th>12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1</td>
<td>D 1 1</td>
<td>size 0 0</td>
<td>Vd 0 1 0 1 1</td>
<td>Q M 0</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VMVN(<c><q>{<dt>} <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VMVN(<c><q>{<dt>} <Qd>, <Qm>

Decode for all variants of this encoding

if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15</th>
<th>12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>D 1 1</td>
<td>size 0 0</td>
<td>Vd 0 1 0 1 1</td>
<td>Q M 0</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VMVN(<c><q>{<dt>} <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VMVN(<c><q>{<dt>} <Qd>, <Qm>

Decode for all variants of this encoding

if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
Assembler symbols

\(<c>\)  For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields on page F1-7228*.

\(<q>\)  See *Standard assembler syntax fields on page F1-7228*.

\(<dt>\)  An optional data type. It is ignored by assemblers, and does not affect the encoding.

\(<qd>\)  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<qd>*2.\n
\(<qm>\)  Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as \(<qm>*2.\n
\(<dd>\)  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<dm>\)  Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = NOT(D[m+r]);

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.153   VNEG

Vector Negate negates each element in a vector, and places the results in a second vector. The floating-point version only inverts the sign bit.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 10 9 8 7 6 5 4 3 0 |
|---------------------------------|---------------------------------|---------------------------------|
| 1 1 1 1 0 1 1 1 1 D 1 1 size 0 1 | Vd 0 F 1 1 1 Q M 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VNEG{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VNEG{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE; floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

A2

| 31 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 10 9 8 7 6 5 4 3 0 |
|---------------------------------|---------------------------------|---------------------------------|
| 1=111 1 1 1 0 1 D 1 1 0 0 0 1 | Vd 1 0 size 0 1 M 0 | Vm |

Half-precision scalar variant

Applies when size == 01.

VNEG{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar variant

Applies when size == 10.

VNEG{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar variant

Applies when size == 11.

VNEG{<c>}{<q>}.F64 <Dd>, <Dm>
Decode for all variants of this encoding

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;
integer esize;
integer d;
integer m;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15</th>
<th>12 11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
<td>size 0 1</td>
<td>Vd 0</td>
<td>F</td>
<td>1</td>
<td>1</td>
<td>Q</td>
<td>0</td>
<td>Vm</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.
VNEG{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.
VNEG{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE; floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
**T2**

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>0</td>
<td>1</td>
<td>M</td>
</tr>
</tbody>
</table>
```

**Half-precision scalar variant**

Applies when size == 01.

```
VNEG{<c>}{<q>}.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

Applies when size == 10.

```
VNEG{<c>}{<q>}.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

Applies when size == 11.

```
VNEG{<c>}{<q>}.F64 <Dd>, <Dm>
```

**Decode for all variants of this encoding**

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

- `<c>`: For encoding A1: see *Standard assembler syntax fields* on page F1-7228. This encoding must be unconditional.

- `<q>`: For encoding A2, T1 and T2: see *Standard assembler syntax fields* on page F1-7228.

- `<dt>`: Is the data type for the elements of the vectors, encoded in the "F:size" field. It can have the following values:
  - S8 when F = 0, size = 00
  - S16 when F = 0, size = 01
  - S32 when F = 0, size = 10
F16 when F = 1, size = 01
F32 when F = 1, size = 10

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                if floating_point then
                    Elem[D[d+r],e,esize] = FPNeg(Elem[D[m+r],e,esize]);
                else
                    result = -SInt(Elem[D[m+r],e,esize]);
                    Elem[D[d+r],e,esize] = result<esize-1:0>;
                else
                    case esize of
                        when 16 S[d] = Zeros(16) : FPNeg(S[m]<15:0>);
                        when 32 S[d] = FPNeg(S[m]);
                        when 64 D[d] = FPNeg(D[m]);

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.154 VNMLA

Vector Negate Multiply Accumulate multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the negation of the product, and writes the result back to the destination register.

Note

Arm recommends that software does not use the VNMLA instruction in the Round towards Plus Infinity and Round towards Minus Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>D</td>
<td>0</td>
<td>1</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>N</td>
<td>1</td>
<td>M</td>
<td>0</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Half-precision scalar variant

Applies when size == 01.

VNMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar variant

Applies when size == 10.

VNMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar variant

Applies when size == 11.

VNMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```c
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 0 & 0 & 0 & D & 0 & 1 & Vn & Vd & 1 & 0 & size & N & 1 & M & 0 & Vm \\
\end{array}
\]

**Half-precision scalar variant**

Applies when size == 01.

\[
\text{VNNMLA}\{c\}\{q\}\cdot.F16 \ <Sd>, \ <Sn>, \ <Sm>
\]

**Single-precision scalar variant**

Applies when size == 10.

\[
\text{VNNMLA}\{c\}\{q\}\cdot.F32 \ <Sd>, \ <Sn>, \ <Sm>
\]

**Double-precision scalar variant**

Applies when size == 11.

\[
\text{VNNMLA}\{c\}\{q\}\cdot.F64 \ <Dd>, \ <Dn>, \ <Dm>
\]

**Decode for all variants of this encoding**

\[
\text{if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;}
\]

\[
\text{if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;}
\]

\[
\text{if size == '01' && InITBlock() then UNPREDICTABLE;}
\]

\[
vtype = \text{if op == '1' then } \text{VFPNegMul_VNMLA else } \text{VFPNegMul_VNNMLS;}
\]

\[
\text{integer esize;}
\]

\[
\text{integer d;}
\]

\[
\text{integer n;}
\]

\[
\text{integer m;}
\]

\[
\text{case size of}
\]

\[
\text{when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);}
\]

\[
\text{when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);}
\]

\[
\text{when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.

• The instruction executes as if it passes the Condition code check.

• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

\(<c>\)

See Standard assembler syntax fields on page F1-7228.

\(<q>\)

See Standard assembler syntax fields on page F1-7228.

\(<Sd>\)

Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

\(<Sn>\)

Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

\(<Sm>\)

Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    case esize of
        when 16
            product16 = FPMul($n<15:0>, S[m]<15:0>, FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(product16), FPSCR[]);
                when VFPNegMul_VNMLS S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16, FPSCR[]);
                when VFPNegMul_VNMUL S[d] = FPNeg(product16);
        when 32
            product32 = FPMul(S[n], S[m], FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA S[d] = FPAdd(FPNeg(S[d]<15:0>), FPNeg(product32), FPSCR[]);
                when VFPNegMul_VNMLS S[d] = FPAdd(FPNeg(S[d]<15:0>), product32, FPSCR[]);
                when VFPNegMul_VNMUL S[d] = FPNeg(product32);
        when 64
            product64 = FPMul(D[n], D[m], FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA D[d] = FPAdd(FPNeg(D[d]<15:0>), FPNeg(product64), FPSCR[]);
                when VFPNegMul_VNMLS D[d] = FPAdd(FPNeg(D[d]<15:0>), product64, FPSCR[]);
                when VFPNegMul_VNMUL D[d] = FPNeg(product64);
F6.1.155   VNMLS

Vector Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register.

Depending on settings in the CPACR, NSACR, HCPTTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

Half-precision scalar variant
Applies when size == 01.
VNMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar variant
Applies when size == 10.
VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar variant
Applies when size == 11.
VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
integer esize;
integer d;
integer n;
integer m;
integer size of
when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRANDED UNPREDICTABLE behavior
If size == '01' && cond != '1110', then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
T1

\begin{verbatim}
15 14 13 12|11 10 9 8|7 6 5 4|3 0 |1 1 1 0 1 1 0 0|D|0 1|Vn|Vd|1 0|size|N|0|M|0|Vm

Half-precision scalar variant
Applies when size == 01.
VNMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar variant
Applies when size == 10.
VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar variant
Applies when size == 11.
VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
integer esize;
integer d;
integer n;
integer m;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior
If size == '01' && InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols
<> See Standard assembler syntax fields on page F1-7228.
<q> See Standard assembler syntax fields on page F1-7228.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Do> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dr> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
\end{verbatim}
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:V'm" field.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    case esize of
      when 16
        product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
        case vtype of
          when VFPNegmul_VNMLA  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(product16), FPSCR[]);
          when VFPNegmul_VNMLS  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16, FPSCR[]);
          when VFPNegmul_VNMUL  S[d] = Zeros(16) : FPNeg(product16);
      when 32
        product32 = FPMul(S[n], S[m], FPSCR[]);
        case vtype of
          when VFPNegmul_VNMLA  S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR[]);
          when VFPNegmul_VNMLS  S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR[]);
          when VFPNegmul_VNMUL  S[d] = FPNeg(product32);
      when 64
        product64 = FPMul(D[n], D[m], FPSCR[]);
        case vtype of
          when VFPNegmul_VNMLA  D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR[]);
          when VFPNegmul_VNMLS  D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR[]);
          when VFPNegmul_VNMUL  D[d] = FPNeg(product64);
F6.1.156 VNMUL

Vector Negate Multiply multiplies together two floating-point register values, and writes the negation of the result to the destination register.

Depending on settings in the CPACR, NSACR, HCPtr, and FPExC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

Half-precision scalar variant

Applies when size == 01.

VNMUL{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>

Single-precision scalar variant

Applies when size == 10.

VNMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>

Double-precision scalar variant

Applies when size == 11.

VNMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>

Decode for all variants of this encoding

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '01' && !HaveFP16Ext() then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
vtype = VFPNegMul_VNMUL;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions
F6.1 Alphabetical list of Advanced SIMD and floating-point instructions

Half-precision scalar variant
Applies when size == 01.
VNMUL{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>

Single-precision scalar variant
Applies when size == 10.
VNMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>

Double-precision scalar variant
Applies when size == 11.
VNMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>

Decode for all variants of this encoding
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '01' && !HaveFP16Ext() then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
vtype = VFPNegMul_VNMUL;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior
If size == '01' && InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols
<c> See Standard assembler syntax fields on page F1-7228.
<q> See Standard assembler syntax fields on page F1-7228.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Db> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    case esize of
        when 16
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(product16), FPSCR[]);
                when VFPNegMul_VNMLS S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16, FPSCR[]);
                when VFPNegMul_VNMUL S[d] = Zeros(16) : FPNeg(product16);
        when 32
            product32 = FPMul(S[n], S[m], FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR[]);
                when VFPNegMul_VNMLS S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR[]);
                when VFPNegMul_VNMUL S[d] = FPNeg(product32);
        when 64
            product64 = FPMul(D[n], D[m], FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR[]);
                when VFPNegMul_VNMLS D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR[]);
                when VFPNegMul_VNMUL D[d] = FPNeg(product64);
F6.1.157 VORN (immediate)

Vector Bitwise OR NOT (immediate) performs a bitwise OR between a register value and the complement of an immediate value, and returns the result into the destination vector.

This instruction is a pseudo-instruction of the VORR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of VORR (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VORR (immediate) gives the operational pseudocode for this instruction.

A1

\[
VORN\{<c>\}{<q>}.I16 \{<Dd>,\} <Dd>, #<imm>
\]

\[
VORR\{<c>\}{<q>}.I16 <Dd>, #~<imm>
\]

is never the preferred disassembly.

128-bit SIMD vector variant

Applies when Q == 1.

\[
VORN\{<c>\}{<q>}.I32 \{<Qd>,\} <Qd>, #<imm>
\]

\[
VORR\{<c>\}{<q>}.I32 <Qd>, #~<imm>
\]

and is never the preferred disassembly.

A2

\[
VORN\{<c>\}{<q>}.I32 \{<Dd>,\} <Dd>, #<imm>
\]

\[
VORR\{<c>\}{<q>}.I32 <Dd>, #~<imm>
\]

and is never the preferred disassembly.
### 128-bit SIMD vector variant

Applies when \( Q = 1 \).

\[
\text{VORN}(<c>{<q>}.I32 \{<Qd>,} <Qd>, \#<imm>
\]

is equivalent to

\[
\text{VORR}(<c>{<q>}.I32 <Qd>, \#<imm>
\]

and is never the preferred disassembly.

#### T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2</th>
<th>0 15 12</th>
<th>11 8 7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
<td>1 1 1 1 1</td>
<td>D 0 0 0</td>
<td>imm3</td>
<td>Vd</td>
<td>0 x x 1</td>
<td>0 Q 0 1</td>
</tr>
</tbody>
</table>

#### 64-bit SIMD vector variant

Applies when \( Q = 0 \).

\[
\text{VORN}(<c>{<q>}.I16 \{<Dd>,} <Dd>, \#<imm>
\]

is equivalent to

\[
\text{VORR}(<c>{<q>}.I16 <Dd>, \#<imm>
\]

and is never the preferred disassembly.

### 128-bit SIMD vector variant

Applies when \( Q = 1 \).

\[
\text{VORN}(<c>{<q>}.I16 \{<Qd>,} <Qd>, \#<imm>
\]

is equivalent to

\[
\text{VORR}(<c>{<q>}.I16 <Qd>, \#<imm>
\]

and is never the preferred disassembly.

#### T2

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2</th>
<th>0 15 12</th>
<th>11 8 7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
<td>1 1 1 1 1</td>
<td>D 0 0 0</td>
<td>imm3</td>
<td>Vd</td>
<td>1 0 x 1</td>
<td>0 Q 0 1</td>
</tr>
</tbody>
</table>

#### 64-bit SIMD vector variant

Applies when \( Q = 0 \).

\[
\text{VORN}(<c>{<q>}.I32 \{<Dd>,} <Dd>, \#<imm>
\]

is equivalent to

\[
\text{VORR}(<c>{<q>}.I32 <Dd>, \#<imm>
\]

and is never the preferred disassembly.

### 128-bit SIMD vector variant

Applies when \( Q = 1 \).
VORN{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>

is equivalent to

VORR{<c>}{<q>}.I32 <Qd>, #~<imm>

and is never the preferred disassembly.

**Assembler symbols**

- `<c>`: For encoding A1 and A2: see *Standard assembler syntax fields* on page F1-7228. This encoding must be unconditional. For encoding T1 and T2: see *Standard assembler syntax fields* on page F1-7228.

- `<q>`: See *Standard assembler syntax fields* on page F1-7228.

- `<Qd>`: Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

- `<Dd>`: Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- `<imm>`: Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of `<imm>`, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions* on page F1-7245.

**Operation for all encodings**

The description of VORR (immediate) gives the operational pseudocode for this instruction.
F6.1.158 **VORN (register)**

Vector bitwise OR NOT (register) performs a bitwise OR NOT operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

**64-bit SIMD vector variant**

Applies when Q == 0.

\[
\text{VORN} \{<c>\} \{<q>\} \{.<dt>\} \{<Dd>\}, <Dn>, <Dm>
\]

**128-bit SIMD vector variant**

Applies when Q == 1.

\[
\text{VORN} \{<c>\} \{<q>\} \{.<dt>\} \{<Qd>,\} <Qn>, <Qm>
\]

**Decode for all variants of this encoding**

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

\[
d = \text{UInt}(D:Vd); \
 n = \text{UInt}(N:Vn); \
 m = \text{UInt}(M:Vm); \
 \text{regs} = \text{if Q == '0' then 1 else 2};
\]

T1

**64-bit SIMD vector variant**

Applies when Q == 0.

\[
\text{VORN} \{<c>\} \{<q>\} \{.<dt>\} \{<Dd>\}, <Dn>, <Dm>
\]

**128-bit SIMD vector variant**

Applies when Q == 1.

\[
\text{VORN} \{<c>\} \{<q>\} \{.<dt>\} \{<Qd>,\} <Qn>, <Qm>
\]

**Decode for all variants of this encoding**

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

\[
d = \text{UInt}(D:Vd); \
 n = \text{UInt}(N:Vn); \
 m = \text{UInt}(M:Vm); \
 \text{regs} = \text{if Q == '0' then 1 else 2};
\]
Assembler symbols

<&> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<dt> An optional data type. It is ignored by assemblers, and does not affect the encoding.

<qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as &Qd>*2.

<qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as &Qn>*2.

<qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as &Qm>*2.

<dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    D[d+r] = D[n+r] OR NOT(D[m+r]);

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.159 VORR (immediate)

Vector Bitwise OR (immediate) performs a bitwise OR between a register value and an immediate value, and returns the result into the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

This instruction is used by the pseudo-instruction VORN (immediate). The pseudo-instruction is never the preferred disassembly.

A1

![Instruction Format](image)

**64-bit SIMD vector variant**

Applies when \( Q == 0 \).

\[ \text{VORR}\{<c>\}\{<q>\}.I32 \{<Dd>,\} <Dd>, #<imm> \]

**128-bit SIMD vector variant**

Applies when \( Q == 1 \).

\[ \text{VORR}\{<c>\}\{<q>\}.I32 \{<Qd>,\} <Qd>, #<imm> \]

**Decode for all variants of this encoding**

if \( \text{cmode<0>} == '0' \) || \( \text{cmode<3:2>} == '11' \) then SEE "VMOV (immediate)";

if \( Q == '1' \) \&\& \( \text{Vd<0>} == '1' \) then UNDEFINED;

\( \text{imm64} = \text{AdvSIMDExpandImm('0', cmode, i:imm3:imm4)} \);

\( d = \text{UInt(D:Vd)}; \) \( \text{regs} = \) if \( Q == '0' \) then 1 else 2;

A2

![Instruction Format](image)

**64-bit SIMD vector variant**

Applies when \( Q == 0 \).

\[ \text{VORR}\{<c>\}\{<q>\}.I16 \{<Dd>,\} <Dd>, #<imm> \]

**128-bit SIMD vector variant**

Applies when \( Q == 1 \).

\[ \text{VORR}\{<c>\}\{<q>\}.I16 \{<Qd>,\} <Qd>, #<imm> \]
Decode for all variants of this encoding

if cmode<0> == '0' || cmode<3:2> == '11' then SEE "VMOV (immediate)";
if Q == '1' & Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 2 0 15</th>
<th>12 11 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
<td>1 1 1 1 1 D 0 0 0</td>
<td>Vd 0 x x 1 0 Q 0 1 imm4</td>
</tr>
<tr>
<td>cmode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant
Applies when Q == 0.
VORR{<c>{<q>}.I32 {<Dd>,} <Dd>, #<imm>

128-bit SIMD vector variant
Applies when Q == 1.
VORR{<c>{<q>}.I32 {<Qd>,} <Qd>, #<imm>

Decode for all variants of this encoding

if cmode<0> == '0' || cmode<3:2> == '11' then SEE "VMOV (immediate)";
if Q == '1' & Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;

T2

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 2 0 15</th>
<th>12 11 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
<td>1 1 1 1 1 D 0 0 0</td>
<td>Vd 1 0 x x 1 0 Q 0 1 imm4</td>
</tr>
<tr>
<td>cmode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant
Applies when Q == 0.
VORR{<c>{<q>}.I16 {<Dd>,} <Dd>, #<imm>

128-bit SIMD vector variant
Applies when Q == 1.
VORR{<c>{<q>}.I16 {<Qd>,} <Qd>, #<imm>

Decode for all variants of this encoding

if cmode<0> == '0' || cmode<3:2> == '11' then SEE "VMOV (immediate)";
if Q == '1' & Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
Assembler symbols

\(<c>\>

For encoding A1 and A2: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1 and T2: see Standard assembler syntax fields on page F1-7228.

\(<q>\>

See Standard assembler syntax fields on page F1-7228.

\(<qd>\>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<qd>\)*2.

\(<dd>\>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<imm>\>

Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of \(<imm>\), see Modified immediate constants in T32 and A32 Advanced SIMD instructions on page F1-7245.

The I8, I64, and F32 data types are permitted as pseudo-instructions, if the immediate can be represented by this instruction, and are encoded using a permitted encoding of the I16 or I32 data type.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[d+r] OR imm64;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.160 VORR (register)

Vector bitwise OR (register) performs a bitwise OR operation between two registers, and places the result in the
destination register. The operand and result registers can be quadword or doubleword. They must all be the same
size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which
the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For
more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

This instruction is used by the pseudo-instructions VMOV (register, SIMD), VRSHR (zero), and VSHR (zero). The
pseudo-instruction is never the preferred disassembly.

A1

|31 30 29 28|27 26 25 24|23 22 21 20|19|16|15|12|11 10 9 |8|7|6|5|4|3|0|
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|1 |1 |1 |1 |0 |0 |1 |0 |D |1 |0 |Vn |Vd |0 |0 |0 |1 |N |Q |M |1 |Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VORR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VORR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' | | Vn<0> == '1' | | Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

|15|14|13|12|11|10|9|8|7|6|5|4|3|0|15|12|11|10|9|8|7|6|5|4|3|0|
|1 |1 |1 |1 |0 |1 |1 |1 |1 |0 |D |1 |0 |Vn |Vd |0 |0 |0 |1 |N |Q |M |1 |Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VORR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VORR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' | | Vn<0> == '1' | | Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
Alias conditions

<table>
<thead>
<tr>
<th>Alias or pseudo-instruction</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMOV (register, SIMD)</td>
<td>N:Vn == M:Vm</td>
</tr>
<tr>
<td>VRSHR (zero)</td>
<td>Never</td>
</tr>
<tr>
<td>VSHR (zero)</td>
<td>Never</td>
</tr>
</tbody>
</table>

Assembler symbols

- <> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
- <> For encoding T1: see Standard assembler syntax fields on page F1-7228.
- <> See Standard assembler syntax fields on page F1-7228.
- <> An optional data type. It is ignored by assemblers, and does not affect the encoding.
- <> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
- <> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
- <> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
- <> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] OR D[m+r];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.161  VPADAL

Vector Pairwise Add and Accumulate Long adds adjacent pairs of elements of a vector, and accumulates the results into the elements of the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

The following figure shows the operation of VPADAL doubleword operation for data type S16.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 | 12 | 11 10 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|--------------|-------------|-------------|-------------|-----|-----|-------|---|---|---|---|---|---|---|---|---|---|
| 1 1 1 1 0 0 | 1 1 1 1 1 | D 1 1 | size 0 0 | Vd | 0 1 1 0 | op | Q | M | 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VPADAL{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VPADAL{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1</td>
<td>D 1 1</td>
<td>size 0 0</td>
<td>Vd</td>
<td>0 1 1 0</td>
<td>op</td>
<td>Q</td>
<td>M</td>
<td>0</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.
VPADAL{<c>}{<q>}.<dt> <Dd>, <Qm>

128-bit SIMD vector variant

Applies when Q == 1.

VPADAL{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Assembler symbols

{<c>}
For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

{<q>}
See Standard assembler syntax fields on page F1-7228.

{<dt>}
Is the data type for the elements of the vectors, encoded in the "op:size" field. It can have the following values:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S8</td>
<td>when op == 0, size == 00</td>
</tr>
<tr>
<td>S16</td>
<td>when op == 0, size == 01</td>
</tr>
<tr>
<td>S32</td>
<td>when op == 0, size == 10</td>
</tr>
<tr>
<td>U8</td>
<td>when op == 1, size == 00</td>
</tr>
<tr>
<td>U16</td>
<td>when op == 1, size == 01</td>
</tr>
<tr>
<td>U32</td>
<td>when op == 1, size == 10</td>
</tr>
</tbody>
</table>

The following encodings are reserved:

- op == 0, size == 11.
- op == 1, size == 11.

{<Qd>}
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

{<Qm>}
Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

{<Dd>}
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

{<Dm>}
Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  h = elements DIV 2;
  for r = 0 to regs-1
    for e = 0 to h-1
      op1 = Elem[D[m+r],2*e,esize]; op2 = Elem[D[m+r],2*e+1,esize];
      result = Int(op1, unsigned) + Int(op2, unsigned);
      Elem[D[d+r],e,2*esize] = Elem[D[d+r],e,2*esize] + result;
Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.162  VPADD (floating-point)

Vector Pairwise Add (floating-point) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The operands and result are doubleword vectors.

The operand and result elements are floating-point numbers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

|31 30 29 28|27 26 25 24|23 22 21 20|19|16|15|12|11|10|9|8|7|6|5|4|3|0|
|1 1 1 1 0 0 1 1 0 D 0 sz Vn Vd 1 1 0 1 N Q M 0 Vm|

A1 variant

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

if Q == '1' then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);

T1

|15 14 13 12|11 10 9|8|7|6|5|4|3|0|15|12|11|10|9|8|7|6|5|4|3|0|
|1 1 1 1 1 1 1 0 D 0 sz Vn Vd 1 1 0 1 N Q M 0 Vm|

T1 variant

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

if Q == '1' then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
**CONSTRAINED UNPREDICTABLE behavior**

If \( \text{sz} = '1' \) \&\& \( \text{InITBlock}() \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

\(<c>\) For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields on page F1-7228*.

\(<q>\) See *Standard assembler syntax fields on page F1-7228*.

\(<dt>\) Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

- F32 when \( \text{sz} = 0 \)
- F16 when \( \text{sz} = 1 \)

\(<Dd>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<Dn>\) Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

\(<Dm>\) Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;
    for e = 0 to h-1
        \( \text{Elem}[\text{dest}, e, \text{esize}] = \text{FPAdd}(\text{Elem}[\text{D}[\text{n}], 2*e, \text{esize}], \text{Elem}[\text{D}[\text{n}], 2*e+1, \text{esize}], \text{StandardFPSCRValue}()); \)
        \( \text{Elem}[\text{dest}, e+h, \text{esize}] = \text{FPAdd}(\text{Elem}[\text{D}[\text{m}], 2*e, \text{esize}], \text{Elem}[\text{D}[\text{m}], 2*e+1, \text{esize}], \text{StandardFPSCRValue}()); \)
    \( \text{D}[\text{d}] = \text{dest}; \)
```
### F6.1.163 VPADD (integer)

Vector Pairwise Add (integer) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The operands and result are doubleword vectors.

The operand and result elements must all be the same type, and can be 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

The following figure shows the operation of VPADD doubleword operation for data type I16.

![Diagram of VPADD operation](image)

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

**A1**

```plaintext
| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 | 16 15 | 12 | 11 10 9 8 7 6 5 4 | 3 0 |
| 1 1 1 1 0 0 1 0 | 0 | D | size | Vn | Vd | 1 0 1 1 N Q M 1 | Vm |
```

**A1 variant**

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

**Decode for this encoding**

if size == '11' || Q == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

**T1**

```plaintext
| 15 14 13 12 | 11 10 9 8 7 6 5 4 | 3 0 | 15 | 12 | 11 10 9 8 7 6 5 4 | 3 0 |
| 1 1 1 0 | 1 1 1 0 | D | size | Vn | Vd | 1 0 1 1 N Q M 1 | Vm |
```

**T1 variant**

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

**Decode for this encoding**

if size == '11' || Q == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
Assembler symbols

<c> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:
I8 when size = 00
I16 when size = 01
I32 when size = 10

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;
    for e = 0 to h-1
        Elem[dest,e,esize] = Elem[D[n],2*e,esize] + Elem[D[n],2*e+1,esize];
        Elem[dest,e+h,esize] = Elem[D[m],2*e,esize] + Elem[D[m],2*e+1,esize];
    D[d] = dest;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.164 VPADDL

Vector Pairwise Add Long adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

The following figure shows the operation of VPADDL doubleword operation for data type S16.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VPADDL{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VPADDL{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when Q == 0.
VPADDL{<c>}{<q>}.<dt> <Dd>, <Qm>

### 128-bit SIMD vector variant

Applies when Q == 1.

VPADDL{<c>}{<q>}.<dt> <Qd>, <Qm>

### Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

### Assembler symbols

- `<>` For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
- `<>` For encoding T1: see Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<dt>` Is the data type for the elements of the vectors, encoded in the "op:size" field. It can have the following values:
  - S8 when op = 0, size = 00
  - S16 when op = 0, size = 01
  - S32 when op = 0, size = 10
  - U8 when op = 1, size = 00
  - U16 when op = 1, size = 01
  - U32 when op = 1, size = 10
  - The following encodings are reserved:
    - op = 0, size = 11.
    - op = 1, size = 11.
- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.
- `<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.

### Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  h = elements DIV 2;
  for r = 0 to regs-1
    for e = 0 to h-1
      op1 = Elem[D[m+r],2*e,esize];  op2 = Elem[D[m+r],2*e+1,esize];
      result = Int(op1, unsigned) + Int(op2, unsigned);
      Elem[D[d+r],e,2*esize] = result<2*esize-1:0>;
Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.165  **VPMAX (floating-point)**

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

### A1

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>15 12 11 10 9 8 7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 1 0 D 0 sz</td>
<td>Vn</td>
<td>Vd</td>
<td>1 1 1 1 N 0 M 0</td>
<td>Vm</td>
<td></td>
</tr>
</tbody>
</table>

**op**

#### A1 variant

\[
\text{VPMAX}\{<c>\}<q>\{<dT>\} \{<Dd>, }<Dn>, <Dm>
\]

**Decode for this encoding**

- if sz == '1' && !HaveFP16Ext() then UNDEFINED;
- \(\text{maximum} = (\text{op} == '0');\)
- integer \(\text{esize} \);
- integer \(\text{elements} \);
- case \(\text{sz} \) of
  - when '0' \(\text{esize} = 32; \text{elements} = 2;\)
  - when '1' \(\text{esize} = 16; \text{elements} = 4;\)
- \(d = \text{UInt}(D:Vd); n = \text{UInt}(N:Vn); m = \text{UInt}(M:Vm);\)

### T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4</th>
<th>3 0</th>
<th>12 11 10 9 8 7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 0 D 0 sz</td>
<td>Vn</td>
<td>Vd</td>
<td>1 1 1 1 N 0 M 0</td>
<td>Vm</td>
</tr>
</tbody>
</table>

**op**

#### T1 variant

\[
\text{VPMAX}\{<c>\}<q>\{<dT>\} \{<Dd>, }<Dn>, <Dm>
\]

**Decode for this encoding**

- if sz == '1' && !HaveFP16Ext() then UNDEFINED;
- if sz == '1' && InITBlock() then UNPREDICTABLE;
- \(\text{maximum} = (\text{op} == '0');\)
- integer \(\text{esize} \);
- integer \(\text{elements} \);
- case \(\text{sz} \) of
  - when '0' \(\text{esize} = 32; \text{elements} = 2;\)
  - when '1' \(\text{esize} = 16; \text{elements} = 4;\)
- \(d = \text{UInt}(D:Vd); n = \text{UInt}(N:Vn); m = \text{UInt}(M:Vm);\)

**CONSTRAINED UNPREDICTABLE behavior**

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

<<> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

<<q> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

- F32 when sz = 0
- F16 when sz = 1

<Db> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;
    for e = 0 to h-1
        op1 = Elem[D[n],2*e,esize];  op2 = Elem[D[n],2*e+1,esize];
        Elem[dest,e,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else FPMin(op1,op2,StandardFPSCRValue());
    op1 = Elem[D[m],2*e,esize];  op2 = Elem[D[m],2*e+1,esize];
    Elem[dest,e+h,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else FPMin(op1,op2,StandardFPSCRValue());
    D[d] = dest;
```
F6.1.166  **VPMAX (integer)**

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

The following figure shows the operation of VPMax doubleword operation for data type S16 or U16.

Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

**A1**

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11 10 9</th>
<th>8</th>
<th>6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>U</td>
<td>0</td>
<td>D</td>
<td>size</td>
<td>Vn</td>
</tr>
</tbody>
</table>
```

**A1 variant**

**VPMax**<c>{<q>}{<d t>}{<Dd>{<Dn},<Dm>}

**Decode for this encoding**

- if size == '11' then UNDEFINED;
- maximum = (op == '0');
- unsigned = (U == '1');
- esize = 8 << UInt(size);
- elements = 64 DIV esize;
- d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);

**T1**

```
| 15 14 13 | 12 | 11 10 9 | 8 | 7 6 5 4 | 3 | 0 | 15 | 12 | 11 10 9 | 8 | 6 5 4 | 3 | 0 |
|----------|---|-------|---|------|---|----|---|-----|---|------|---|------|---|----|
| 1 | 1 | 1 | U | 1 | 1 | 1 | 0 | D | size | Vn | Vd | 1 | 0 | 1 | 0 | N | 0 | M | 0 | Vm |
```

**T1 variant**

**VPMax**<c>{<q>}{<d t>}{<Dd>{<Dn},<Dm>}

**Decode for this encoding**

- if size == '11' then UNDEFINED;
- maximum = (op == '0');
- unsigned = (U == '1');
- esize = 8 << UInt(size);
- elements = 64 DIV esize;
- d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
Assembler symbols

<-> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:
- S8 when U = 0, size = 00
- S16 when U = 0, size = 01
- S32 when U = 0, size = 10
- U8 when U = 1, size = 00
- U16 when U = 1, size = 01
- U32 when U = 1, size = 10

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;
    for e = 0 to h-1
        op1 = Int(Elem[D[n],2*e,esize], unsigned);
        op2 = Int(Elem[D[n],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e,esize] = result<esize-1:0>;
    op1 = Int(Elem[D[m],2*e,esize], unsigned);
    op2 = Int(Elem[D[m],2*e+1,esize], unsigned);
    result = if maximum then Max(op1,op2) else Min(op1,op2);
    Elem[dest,e+h,esize] = result<esize-1:0>;

    D[d] = dest;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.167 VPMIN (floating-point)

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

```
|31 30 29 28|27 26 25 24|23 22 21 20|19 16|15|12|11 10 9 |8 |7 |6 |5 |4 |3 |0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|1 |1 |1 |0 |0 |1 |1 |D |1 |sz |Vn |Vd |1 |1 |1 |N |0 |M |0 |Vm |
```

A1 variant

VPMIN{<c>}{<q>}.{<dt>}{<Dd>, }{<Dn>, }{<Dm>}

Decode for this encoding

```
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
    d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

```
|15 14 13 12|11 10 9 |8 |7 |6 |5 |4 |3 |0 |
|---|---|---|---|---|---|---|---|---|---|
|1 |1 |1 |1 |1 |1 |1 |0 |D |1 |sz |Vn |Vd |1 |1 |1 |N |0 |M |0 |Vm |
```

T1 variant

VPMIN{<c>}{<q>}.{<dt>}{<Dd>, }{<Dn>, }{<Dm>}

Decode for this encoding

```
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
maximum = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
    d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

<> For encoding A1: see *Standard assembler syntax fields* on page F1-7228. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields* on page F1-7228.

<q> See *Standard assembler syntax fields* on page F1-7228.

<dt> Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

- F32 when sz = 0
- F16 when sz = 1

<Db> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dr> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;
    for e = 0 to h-1
        op1 = Elem[D[n],2*e,esize]; op2 = Elem[D[n],2*e+1,esize];
        Elem[dest,e,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else FPMin(op1,op2,StandardFPSCRValue());
        op1 = Elem[D[m],2*e,esize]; op2 = Elem[D[m],2*e+1,esize];
        Elem[dest,e+h,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else FPMin(op1,op2,StandardFPSCRValue());
    D[d] = dest;
```
F6.1.168  **VPMIN (integer)**

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

**A1**

![A1 encoding](image)

**A1 variant**

VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

*Decode for this encoding*

if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);

**T1**

![T1 encoding](image)

**T1 variant**

VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

*Decode for this encoding*

if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);

**Assembler symbols**

- `<c>`  
  For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.
  For encoding T1: see *Standard assembler syntax fields on page F1-7228*.

- `<q>`  
  See *Standard assembler syntax fields on page F1-7228*.

- `<dt>`  
  Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:
  
  - S8  
    when U = 0, size = 00
  
  - S16  
    when U = 0, size = 01
  
  - S32  
    when U = 0, size = 10
U8 when U = 1, size = 00
U16 when U = 1, size = 01
U32 when U = 1, size = 10

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;
    for e = 0 to h-1
        op1 = Int(Elem[D[n],2*e,esize], unsigned);
        op2 = Int(Elem[D[n],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e,esize] = result<esize-1:0>;
    op1 = Int(Elem[D[m],2*e,esize], unsigned);
    op2 = Int(Elem[D[m],2*e+1,esize], unsigned);
    result = if maximum then Max(op1,op2) else Min(op1,op2);
    Elem[dest,e+h,esize] = result<esize-1:0>;
    D[d] = dest;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.169 VPOP

Pop SIMD&FP registers from stack loads multiple consecutive Advanced SIMD and floating-point register file registers from the stack.

This instruction is an alias of the VLDM, VLMDB, VLMIA instruction. This means that:

- The encodings in this description are named to match the encodings of VLDM, VLMDB, VLMIA.
- The description of VLDM, VLMDB, VLMIA gives the operational pseudocode for this instruction.

A1

\[
\begin{array}{ccccccccccccccccc}
\text{cond} & P & U & W & Rn & Vd & 1 & 0 & 1 & 0 & \text{imm8<7:1>} & 0 \\
\end{array}
\]

**Increment After variant**

VPOP{<c>}{<q>}{.<size>} <dreglist>

is equivalent to

VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>

and is always the preferred disassembly.

A2

\[
\begin{array}{ccccccccccccccccc}
\text{cond} & P & U & W & Rn & Vd & 1 & 0 & 1 & 0 & \text{imm8} & \\
\end{array}
\]

**Increment After variant**

VPOP{<c>}{<q>}{.<size>} <sreglist>

is equivalent to

VLDM{<c>}{<q>}{.<size>} SP!, <sreglist>

and is always the preferred disassembly.

T1

\[
\begin{array}{ccccccccccccccccc}
\text{imm8<0>} & \\
\end{array}
\]

**Increment After variant**

VPOP{<c>}{<q>}{.<size>} <dreglist>

is equivalent to

VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>

and is always the preferred disassembly.
T2

Increment After variant

VPOP{<c>}{<q>}{.<size>} <sreglist>

is equivalent to

VLDM{<c>}{<q>}{.<size>} SP!, <sreglist>

and is always the preferred disassembly.

Assembler symbols

<c> See Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<size> An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

<sreglist> is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.

<dreglist> is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

Operation for all encodings

The description of VLDM, VLDMDB, VLDMIA gives the operational pseudocode for this instruction.

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.170 VPUSH

Push SIMD&FP registers to stack stores multiple consecutive registers from the Advanced SIMD and floating-point register file to the stack.

This instruction is an alias of the VSTM, VSTMDB, VSTMIA instruction. This means that:

- The encodings in this description are named to match the encodings of VSTM, VSTMDB, VSTMIA.
- The description of VSTM, VSTMDB, VSTMIA gives the operational pseudocode for this instruction.

A1

```
[31  28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 |] 1 0 |
    !=1111 0 1 0 1 0 D 0 1 1 0 1 Vd 1 0 1 1 imm8<7:1> 0
  
  cond  P  U  W  Rn
  
imm8<0>
```

Decrement Before variant

VPUSH{<c>}{<q>}{.<size>} <dreglist>

is equivalent to

VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>

and is always the preferred disassembly.

A2

```
[31  28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 |] 0 |
    !=1111 0 1 0 1 0 D 0 1 1 0 1 Vd 1 0 1 0 imm8
  
  cond  P  U  W  Rn
  
```

Decrement Before variant

VPUSH{<c>}{<q>}{.<size>} <sreglist>

is equivalent to

VSTMDB{<c>}{<q>}{.<size>} SP!, <sreglist>

and is always the preferred disassembly.

T1

```
[15  14 13 12|11 10 9 8 7 6 5 4 3 |] 0 [15 12|11 10 9 8 7 |] 1 0 |
  1 1 1 0 1 0 1 0 D 0 1 1 0 1 Vd 1 0 1 1 imm8<7:1> 0
  
  P  U  W  Rn
  
imm8<0>
```

Decrement Before variant

VPUSH{<c>}{<q>}{.<size>} <dreglist>

is equivalent to

VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>

and is always the preferred disassembly.
### Decrement Before variant

VPUSH{<c>}{<q>}{.<size>} <sreglist>

is equivalent to

VSTMDB{<c>}{<q>}{.<size>} SP!, <sreglist>

and is always the preferred disassembly.

### Assembler symbols

- `<c>` See [Standard assembler syntax fields on page F1-7228.](#)
- `<q>` See [Standard assembler syntax fields on page F1-7228.](#)
- `<size>` An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.
- `<sreglist>` Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.
- `<dreglist>` Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

### Operation for all encodings

The description of VSTM, VSTMDB, VSTMIA gives the operational pseudocode for this instruction.

### Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.171  **VQABS**

Vector Saturating Absolute takes the absolute value of each element in a vector, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-7131.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

**A1**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 10 9 8 7 6 5 4 3 0 | 1 1 1 0 0 1 1 1 | D 1 1 | size 0 0 | Vd 0 1 1 1 0 | Q 0 0 | M 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VQABS{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VQABS{<c>}{<q>}.<dt> <Qd>, <Qm>

**Decode for all variants of this encoding**

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**T1**

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 12 11 10 9 8 7 6 5 4 3 0 | 1 1 1 1 1 1 1 1 | D 1 1 | size 0 0 | Vd 0 1 1 1 0 | Q 0 0 | M 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VQABS{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VQABS{<c>}{<q>}.<dt> <Qd>, <Qm>
Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Assembler symbols

<c>  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q>  See Standard assembler syntax fields on page F1-7228.

<dt>  Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:
S8   when size = 00
S16  when size = 01
S32  when size = 10
The encoding size = 11 is reserved.

<Qd>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>  Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>  Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
EncodingSpecificOperations();  CheckAdvSIMDEnabled();
for r = 0 to regs-1
  for e = 0 to elements-1
    result = Abs(SInt(Elem[D[m+r],e,esize]));
    boolean sat;
    (Elem[D[d+r],e,esize], sat) = SignedSatQ(result, esize);
    if sat then FPSCR.QC = '1';
F6.1.172 VQADD

Vector Saturating Add adds the values of corresponding elements of two vectors, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode description of saturation on page E1-7131.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16</th>
<th>12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1</td>
<td>D size</td>
<td>Vn</td>
<td>Vd</td>
<td>0 0 0 0</td>
<td>N Q M 1</td>
<td>Vm</td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VQADD{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VQADD{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
<th>12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 U 1 1 1 0</td>
<td>D size</td>
<td>Vn</td>
<td>Vd</td>
<td>0 0 0 0</td>
<td>N Q M 1</td>
<td>Vm</td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VQADD{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VQADD{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
Decode for all variants of this encoding

\[
\text{if } Q == '1' \land (Vd<0> == '1' \lor Vn<0> == '1' \lor Vm<0> == '1') \text{ then UNDEFINED;}
\]

\[
\text{unsigned} = (U == '1');
\]

\[
esize = 8 \ll \text{UInt(size)}; \quad \text{elements} = 64 \div \text{esize};
\]

\[
d = \text{UInt}(D:Vd); \quad n = \text{UInt}(N:Vn); \quad m = \text{UInt}(M:Vm); \quad \text{regs} = \text{if } Q == '0' \text{ then 1 else 2};
\]

Assembler symbols

\(<c>\quad \text{For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.}\)

\(<q>\quad \text{For encoding T1: see Standard assembler syntax fields on page F1-7228.}\)

\(<dt>\quad \text{Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:}\)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S8</td>
<td>when (U = 0, \text{size} = 00)</td>
</tr>
<tr>
<td>S16</td>
<td>when (U = 0, \text{size} = 01)</td>
</tr>
<tr>
<td>S32</td>
<td>when (U = 0, \text{size} = 10)</td>
</tr>
<tr>
<td>S64</td>
<td>when (U = 0, \text{size} = 11)</td>
</tr>
<tr>
<td>U8</td>
<td>when (U = 1, \text{size} = 00)</td>
</tr>
<tr>
<td>U16</td>
<td>when (U = 1, \text{size} = 01)</td>
</tr>
<tr>
<td>U32</td>
<td>when (U = 1, \text{size} = 10)</td>
</tr>
<tr>
<td>U64</td>
<td>when (U = 1, \text{size} = 11)</td>
</tr>
</tbody>
</table>

\(<qd>\quad \text{Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.}\)

\(<qn>\quad \text{Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.}\)

\(<qm>\quad \text{Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.}\)

\(<dd>\quad \text{Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vm" field.}\)

\(<dn>\quad \text{Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.}\)

\(<dm>\quad \text{Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.}\)

Operation for all encodings

\[
\text{if ConditionPassed() then}
\]

\[
\text{EncodingSpecificOperations(); CheckAdvSIMDEnabled();}
\]

\[
\text{for } r = 0 \text{ to } \text{regs-1}
\]

\[
\text{for } e = 0 \text{ to } \text{elements-1}
\]

\[
\text{sum} = \text{Int(Elem}[D[n+r],e,\text{esize}], \text{unsigned}) + \text{Int(Elem}[D[m+r],e,\text{esize}], \text{unsigned});
\]

\[
\text{boolean sat;}
\]

\[
(Elem[D[d+r],e,\text{esize}], \text{sat}) = \text{SatQ}(\text{sum, esize, unsigned});
\]

\[
\text{if sat then FPSCR.QC = '1';}
\]
F6.1.173 VQDMLAL

Vector Saturating Doubling Multiply Accumulate Long multiplies corresponding elements in two doubleword vectors, doubles the products, and accumulates the results into the elements of a quadword vector.

The second operand can be a scalar instead of a vector. For more information about scalars see Advanced SIMD scalars on page F1-7254.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode description of saturation on page E1-7131.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

\[
\begin{array}{cccccccccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & D & \text{!=11} & Vn & Vd & 1 & 0 & 0 & 1 & N & 0 & M & 0 & Vm
\end{array}
\]

A1 variant

VQDMLAL\{<c>\}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;

A2

\[
\begin{array}{cccccccccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & D & \text{!=11} & Vn & Vd & 0 & 0 & 1 & 1 & N & 1 & M & 0 & Vm
\end{array}
\]

A2 variant

VQDMLAL\{<c>\}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

Decode for this encoding

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);
index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);
index = UInt(M);
T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0 15</th>
<th>12 11 10 9</th>
<th>8 7 6 5</th>
<th>4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0</td>
<td>1 1 1 1 1</td>
<td>D</td>
<td>size</td>
<td>Vn</td>
<td>Vd</td>
<td>1 0 0 1</td>
</tr>
</tbody>
</table>

**T1 variant**

VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

**Decode for this encoding**

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;

T2

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0 15</th>
<th>12 11 10 9</th>
<th>8 7 6 5</th>
<th>4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
<td>1 1 1 1 1</td>
<td>D</td>
<td>size</td>
<td>Vn</td>
<td>Vd</td>
<td>0 0 1 1</td>
</tr>
</tbody>
</table>

**T2 variant**

VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

**Decode for this encoding**

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);

**Notes for all encodings**

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

**Assembler symbols**

- **<c>** For encoding A1 and A2: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
  
  For encoding T1 and T2: see Standard assembler syntax fields on page F1-7228.

- **<q>** See Standard assembler syntax fields on page F1-7228.

- **<dt>** Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

  - S16 when size = 01
S32 when size = 10

<Qd>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn>
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>
For encoding A1 and T1: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
For encoding A2 and T2: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when <dt> is S16, otherwise the "Vm" field.

<index>
Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when <dt> is S16, otherwise in range 0 to 1, encoded in the "M" field.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    integer op2;
    if scalar_form then op2 = SInt(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[Din[m],e,esize]);
        op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat1) = SignedSatQ(2*op1*op2, 2*esize);
        integer result;
        if add then
            result = SInt(Elem[Qin[d>>1],e,2*esize]) + SInt(product);
        else
            result = SInt(Elem[Qin[d>>1],e,2+esize]) - SInt(product);
        boolean sat2;
        (Elem[Q[d>>1],e,2+esize], sat2) = SignedSatQ(result, 2+esize);
        if sat1 || sat2 then FPSCR.QC = '1';
F6.1.174  VQDMLSL

Vector Saturating Doubling Multiply Subtract Long multiplies corresponding elements in two doubleword vectors, subtracts double the products from corresponding elements of a quadword vector, and places the results in the same quadword vector.

The second operand can be a scalar instead of a vector. For more information about scalars see Advanced SIMD scalars on page F1-7254.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode description of saturation on page E1-7131.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

11110010

size op

A1 variant

VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;

A2

11110010

size op

A2 variant

VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

Decode for this encoding

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
**T1 variant**

\[ VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm> \]

**Decode for this encoding**

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;

**T2 variant**

\[ VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>] \]

**Decode for this encoding**

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);

**Notes for all encodings**

Related encodings: See *Advanced SIMD data-processing on page F3-7314* for the T32 instruction set, or *Advanced SIMD data-processing on page F4-7422* for the A32 instruction set.

**Assembler symbols**

- **<>** For encoding A1 and A2: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.

  For encoding T1 and T2: see *Standard assembler syntax fields on page F1-7228*.

- **<q>** See *Standard assembler syntax fields on page F1-7228*.

- **<dt>** Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

  \[
  \begin{align*}
  S16 & \quad \text{when size = 01} \\
  & \quad \text{else when size = 00} \\
  \end{align*}
  \]
S32 when size = 10

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> For encoding A1 and T1: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

For encoding A2 and T2: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when <dt> is S16, otherwise the "Vm" field.

<index> Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when <dt> is S16, otherwise in range 0 to 1, encoded in the "M" field.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    integer op2;
    if scalar_form then op2 = SInt(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[Din[m],e,esize]);
        op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat1) = SignedSatQ(2*op1*op2, 2*esize);
        integer result;
        if add then
            result = SInt(Elem[Qin[d>>1],e,2*esize]) + SInt(product);
        else
            result = SInt(Elem[Qin[d>>1],e,2*esize]) - SInt(product);
        boolean sat2;
        (Elem[Q[d>>1],e,2*esize], sat2) = SignedSatQ(result, 2*esize);
        if sat1 || sat2 then FPSCR.QC = '1';
### F6.1.175 VQDMULH

Vector Saturating Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are truncated, for rounded results see VQRDMULH.

The second operand can be a scalar instead of a vector. For more information about scalars see Advanced SIMD scalars on page F1-7254.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode description of saturation on page E1-7131.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

#### A1

| [31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4 3 0 | 1 1 1 0 0 0 D size | Vn | Vd | 1 0 1 1 N | Q | M 0 | Vm |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

**64-bit SIMD vector variant**

Applies when Q == 0.

VQDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>

**128-bit SIMD vector variant**

Applies when Q == 1.

VQDMULH{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm[x]>

#### Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

#### A2

| [31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4 3 0 | 1 1 1 0 0 1 Q | D | l=11 | Vn | Vd | 1 1 0 0 N | 1 | M 0 | Vm |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

**64-bit SIMD vector variant**

Applies when Q == 0.

VQDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm[x]>

**128-bit SIMD vector variant**

Applies when Q == 1.

VQDMULH{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm[x]>
**Decode for all variants of this encoding**

```plaintext
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then
  esize = 16;
  elements = 4;
  m = UInt(Vm<2:0>);
  index = UInt(M:Vm<3>);
if size == '10' then
  esize = 32;
  elements = 2;
  m = UInt(Vm);
  index = UInt(M);
```

### T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0</td>
<td>1 1 1 1</td>
<td>0</td>
<td>D</td>
<td>size</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector variant**

Applies when Q == 0.

\(VQDMULH\{<c>\}{<q>\}.<dt> \{<D>, <Dn>, <Dm}\}

### 128-bit SIMD vector variant

Applies when Q == 1.

\(VQDMULH\{<c>\}{<q>\}.<dt> \{<Qd>, <Qn>, <Qm}\}

### Decode for all variants of this encoding

```plaintext
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);
integer elements;
integer m;
integer index;
if size == '01' then
  esize = 16;
  elements = 4;
  m = UInt(Vm<2:0>);
  index = UInt(M:Vm<3>);
if size == '10' then
  esize = 32;
  elements = 2;
  m = UInt(Vm);
  index = UInt(M);
```

### T2

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>1</td>
<td>D</td>
<td>size</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector variant**

Applies when Q == 0.

\(VQDMULH\{<c>\}{<q>\}.<dt> \{<D>, <Dm>\}

### 128-bit SIMD vector variant

Applies when Q == 1.

\(VQDMULH\{<c>\}{<q>\}.<dt> \{<Q>, <Qm>\}

```
Decode for all variants of this encoding

if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);

Notes for all encodings

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

Assembler symbols

<>  For encoding A1 and A2: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1 and T2: see Standard assembler syntax fields on page F1-7228.

<>  See Standard assembler syntax fields on page F1-7228.
<dt>  Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:
S16  when size = 01
S32  when size = 10
<qd>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <qd>*2.
<qn>  Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <qn>*2.
<qm>  Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <qm>*2.
<db>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<dn>  Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<dm[x]>  Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, <dm> is restricted to D0-D7. <dm> is encoded in "Vm<2:0>" and x is encoded in "M:Vm<3>". If <dt> is S32, <dm> is restricted to D0-D15. <dm> is encoded in "Vm", and x is encoded in "M".
<dm>  Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
integer op2;
if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
for r = 0 to regs-1
  for e = 0 to elements-1
    if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
    op1 = SInt(Elem[D[m+r],e,esize]);
    // The following only saturates if both op1 and op2 equal -(2^size)(esize-1)
    (result, sat) = SignedSatQ((2*op1*op2) >> esize, esize);
Elem[D[d+r],e,esize] = result;
if sat then FPSCR.QC = '1';
F6.1.176 VQDMULL

Vector Saturating Doubling Multiply Long multiplies corresponding elements in two doubleword vectors, doubles the products, and places the results in a quadword vector.

The second operand can be a scalar instead of a vector. For more information about scalars see Advanced SIMD scalars on page F1-7254.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode description of saturation on page E1-7131.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|-------------|-------------|-------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 1 1 1 0 0 | 1 0 1 1 D | !=11 Vn | Vd | 1 1 0 1 | N 0 | M 0 | Vm |

size

A1 variant

VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = FALSE; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 8 << UInt(size); elements = 64 DIV esize;

A2

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|-------------|-------------|-------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 1 1 1 0 0 | 1 0 1 1 D | !=11 Vn | Vd | 1 0 1 1 | N 1 | M 0 | Vm |

size

A2 variant

VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn);
type esize;
type elements;
type m;
type index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);
T1

VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;

T2

VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);

Notes for all encodings

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

Assembler symbols

For encoding A1 and A2: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1 and T2: see Standard assembler syntax fields on page F1-7228.

See Standard assembler syntax fields on page F1-7228.

Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

S16 when size = 01
S32 when size = 10
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    integer op2;
    if scalar_form then op2 = SInt(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[Din[m],e,esize]);
        op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat) = SignedSatQ(2*op1*op2, 2*esize);
        Elem[Q[d>>1],e,2*esize] = product;
        if sat then FPSCR.QC = '1';
```
**F6.1.177 VQMOVN, VQMOVUN**

Vector Saturating Move and Narrow copies each element of the operand vector to the corresponding element of the destination vector.

The operand is a quadword vector. The elements can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result is a doubleword vector. The elements are half the length of the operand vector elements. If the operand is unsigned, the results are unsigned. If the operand is signed, the results can be signed or unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode description of saturation on page E1-7131.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

This instruction is used by the pseudo-instructions VQRSHRN (zero), VQRSHRUN (zero), VQSHRN (zero), and VQSHRUN (zero). The pseudo-instruction is never the preferred disassembly.

### A1

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15  |12|11 10 9 8 7 6 5 4 |3 0 |
| 1 1 1 1 0 0 1 1 1 1 size 1 0 | Vd | 0 0 1 0 | op M 0 | Vm |
```

**Signed result variant**

Applies when $op == 1x$.

\[
\text{VQMOVN}\{<c>\}\{<q>\}.<dt> <Dd>, <Qm>
\]

**Unsigned result variant**

Applies when $op == 01$.

\[
\text{VQMOVUN}\{<c>\}\{<q>\}.<dt> <Dd>, <Qm>
\]

**Decode for all variants of this encoding**

- if $op == '00'$ then SEE "VMOVN";
- if $size == '11' || Vm\&0 == '1' then UNDEFINED;
- src\_unsigned = \( (op == '11') \);
- dest\_unsigned = \( (op\&0 == '1') \);
- esize = 8 << UInt(size); elements = 64 DIV esize;
- \( d = \text{UInt}(D:Vd); \) \( m = \text{UInt}(M:Vm); \)

### T1

```
| 15 14 13 12|11 10 9 8 7 6 5 4 |3 2 1 0 |15  |12|11 10 9 8 7 6 5 4 |3 0 |
| 1 1 1 1 1 1 1 1 size 1 0 | Vd | 0 0 1 0 | op M 0 | Vm |
```

**Signed result variant**

Applies when $op == 1x$.

\[
\text{VQMOVN}\{<c>\}\{<q>\}.<dt> <Dd>, <Qm>
\]
Unsigned result variant

Applies when op == 01.

VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>  

Decode for all variants of this encoding

if op == '00' then SEE "VMOVN";
if size == '11' || Vm<0> == '1' then UNDEFINED;
src_unsigned = (op == '11');  dest_unsigned = (op<0> == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);

Assembler symbols

<c>  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q>  See Standard assembler syntax fields on page F1-7228.

<dt>  For the signed result variant: is the data type for the elements of the operand, encoded in the "op<0>:size" field. It can have the following values:
S16  when op<0> = 0, size = 00
S32  when op<0> = 0, size = 01
S64  when op<0> = 0, size = 10
U16  when op<0> = 1, size = 00
U32  when op<0> = 1, size = 01
U64  when op<0> = 1, size = 10
The following encodings are reserved:
  •  op<0> = 0, size = 11.
  •  op<0> = 1, size = 11.

For the unsigned result variant: is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:
S16  when size = 00
S32  when size = 01
S64  when size = 10
The encoding size = 11 is reserved.

<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm>  Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for e = 0 to elements-1
    operand = Int(Elem[Qm][m+1],e,2*esize], src_unsigned);
    boolean sat;
    (Elem[D:d][e,esize], sat) = SatQ(operand, esize, dest_unsigned);
    if sat then FPSCR.QC = '1';
F6.1.178  **VQNEG**

Vector Saturating Negate negates each element in a vector, and places the results in the destination vector. If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-7131.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

**A1**

64-bit SIMD vector variant

Applies when Q == 0.

\[ \text{VQNEG\{<c>\}{<q>}.<dt> <Dd>, <Dm> \} \]

128-bit SIMD vector variant

Applies when Q == 1.

\[ \text{VQNEG\{<c>\}{<q>}.<dt> <Qd>, <Qm> \} \]

*Decode for all variants of this encoding*

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**T1**

64-bit SIMD vector variant

Applies when Q == 0.

\[ \text{VQNEG\{<c>\}{<q>}.<dt> <Dd>, <Dm> \} \]

128-bit SIMD vector variant

Applies when Q == 1.

\[ \text{VQNEG\{<c>\}{<q>}.<dt> <Qd>, <Qm> \} \]

*Decode for all variants of this encoding*

if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
Assembler symbols

<q> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

S8 when size = 00
S16 when size = 01
S32 when size = 10
The encoding size = 11 is reserved.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = -SInt(Elem[D+m+r],e,esize));
            boolean sat;
            (Elem[D+d+r],e,esize], sat) = SignedSatQ(result, esize);
            if sat then FPSCR.QC = '1';
F6.1.179   VQRDMLAH

Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half. This instruction multiplies the vector elements of the first source SIMD&FP register with either the corresponding vector elements of the second source SIMD&FP register or the value of a vector element of the second source SIMD&FP register, without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode description of saturation on page E1-7131.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

(FEAT_RDM)

|31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8|7 6 5 4|3 0 |
|---|---|---|---|---|---|---|---|---|
|1|1|1|0|0|1|0|D|size|Vn|Vd|1|0|1|N|Q|M|1|Vm|

64-bit SIMD vector variant

Applies when Q == 0.

VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

if !HaveVQRDMLAHExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = TRUE; scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

A2

(FEAT_RDM)

|31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8|7 6 5 4|3 0 |
|---|---|---|---|---|---|---|---|---|
|1|1|1|0|0|1|Q|1|D|size|Vn|Vd|1|1|0|N|1|M|0|Vm|

64-bit SIMD vector variant

Applies when Q == 0.

VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

128-bit SIMD vector variant

Applies when Q == 1.
VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

if !HaveQRDMLAHExt() then UNDEFINED;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
add = TRUE; scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

T1

(FEAT_RDM)

|1 5 14 13 12|1 1 1 0|D|size|Vn|Vd|N|Q|M|Vm|
|---|---|---|---|---|---|---|---|---|---|---|
|1 1 1 1 1 1 1 0|D|size|Vn|Vd|N|Q|M|Vm|

64-bit SIMD vector variant

Applies when Q == 0.
VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.
VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

if !HaveQRDMLAHExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = TRUE; scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

(FEAT_RDM)
64-bit SIMD vector variant

Applies when Q = 0.

\[
\text{VQRDMLAH\{<q>\}.<dt> <Dd>, <Dn>, <Dm[x]>}
\]

128-bit SIMD vector variant

Applies when Q = 1.

\[
\text{VQRDMLAH\{<q>\}.<dt> <Qd>, <Qn>, <Dm[x]>}
\]

Decode for all variants of this encoding

```
if !HaveQRDMLAHExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
add = TRUE; scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);
```

CONSTRANGED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

Assembler symbols

- \(<q>\) See Standard assembler syntax fields on page F1-7228.
- \(<dt>\) Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:
  - S16 when size = 01
  - S32 when size = 10
- \(<Qd>\) Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as \(<Qd>\)*2.
- \(<Qn>\) Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as \(<Qn>\)*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

EncodingSpecificOperations();  CheckAdvSIMDEnabled();
round_const = 1 << (esize-1);
integer op2;
if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
for r = 0 to regs-1
  for e = 0 to elements-1
    op1 = SInt(Elem[D[n+r],e,esize]);
    op3 = SInt(Elem[D[d+r],e,esize]) << esize;
    if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
    (result, sat) = SignedSatQ((op3 + 2*(op1*op2) + round_const) >> esize, esize);
    Elem[D[d+r],e,esize] = result;
    if sat then FPSCR.QC = '1';
F6.1.180 VQRDMLSH

Vector Saturating Rounding Doubling Multiply Subtract Returning High Half. This instruction multiplies the vector elements of the first source SIMD&FP register with either the corresponding vector elements of the second source SIMD&FP register or the value of a vector element of the second source SIMD&FP register, without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode description of saturation on page E1-7131.

Depending on settings in the CPACR, NSACR, and HCPTER registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1
(FEAT_RDM)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>size</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>N</td>
<td>Q</td>
<td>M</td>
<td>1</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

if !HaveQRDMLAHExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = FALSE; scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

A2
(FEAT_RDM)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Q</td>
<td>1</td>
<td>D</td>
<td>size</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>N</td>
<td>1</td>
<td>M</td>
<td>0</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VQRDMLSH{<q>}.<dt> <Dd>, <Dm> [x]

128-bit SIMD vector variant

Applies when Q == 1.
VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Qm>[

**Decode for all variants of this encoding**

if !HaveQRDMLAHExt() then UNDEFINED;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
ad = FALSE; scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = UInt(M);

t1

(FEAT_RDM)

[]

64-bit SIMD vector variant

Applies when Q == 0.

VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm>[

128-bit SIMD vector variant

Applies when Q == 1.

VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Qm>[

**Decode for all variants of this encoding**

if !HaveQRDMLAHExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
ad = FALSE; scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

**CONSTRAINED UNPREDICTABLE behavior**

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
T2

(�Feat_RDM)
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

EncodingSpecificOperations();  CheckAdvSIMDEnabled();
round_const = 1 << (esize-1);
integer op2;
if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
for r = 0 to regs-1
  for e = 0 to elements-1
    op1 = SInt(Elem[D[n+r],e,esize]);
    op3 = SInt(Elem[D[d+r],e,esize]) << esize;
    if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
    (result, sat) = SignedSatQ((op3 - 2*(op1*op2) + round_const) >> esize, esize);
    Elem[D[d+r],e,esize] = result;
    if sat then FPSCR.QC = '1';
F6.1.181 VQRDMULH

Vector Saturating Rounding Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are rounded. For truncated results see VQDMULH.

The second operand can be a scalar instead of a vector. For more information about scalars see Advanced SIMD scalars on page F1-7254.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode description of saturation on page E1-7131.

Depending on settings in the CPACR, NSACR, and HCPtr registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be undefined, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE; esize = 8 << Uint(size); elements = 64 DIV esize;
d = Uint(D:Vd); n = Uint(N:Vn); m = Uint(M:Vm); regs = if Q == '0' then 1 else 2;

A2

64-bit SIMD vector variant

Applies when Q == 0.

VQRDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm[x]>

128-bit SIMD vector variant

Applies when Q == 1.

VQRDMULH{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm[x]>
Decode for all variants of this encoding

if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);

T1

|15 14 13 12|11 10  9 8 | 7 6 5 4 3 | 0 | 15 12 11 10  9 8 | 7 6 5 4 3 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | D | size | Vn | Vd | 1 | 0 | 1 | 1 | N | Q | M | 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.
VQRDMULH{<cc>}{<qp>}.<dt> {<Db>, }<Dm>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.
VQRDMULH{<cc>}{<qp>}.<dt> {<Qb>, }<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T2

|15 14 13 12|11 10  9 8 | 7 6 5 4 3 | 0 | 15 12 11 10  9 8 | 7 6 5 4 3 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | D | size | Vn | Vd | 1 | 1 | 0 | 1 | N | 1 | M | 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.
VQRDMULH{<cc>}{<qp>}.<dt> {<Db>, }<Dm>, <Dm[x]>

128-bit SIMD vector variant

Applies when Q == 1.
VQRDMULH{<cc>}{<qp>}.<dt> {<Qb>, }<Qn>, <Qm[x]>
Decode for all variants of this encoding

if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' & (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);

Notes for all encodings

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

Assembler symbols

<>  For encoding A1 and A2: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1 and T2: see Standard assembler syntax fields on page F1-7228.
<q>  See Standard assembler syntax fields on page F1-7228.
<dt>  Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:
S16  when size = 01
S32  when size = 10
<Qd>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>  Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>  Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>  Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm[x]>  Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>"", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".
<Dm>  Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  round_const = 1 << (esize-1);
  integer op2;
  if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = SInt(Elem[D[m+r],e,esize]);
      if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
      (result, sat) = SignedSatQ((2*op1*op2 + round_const) >> esize, esize);
Elem[D+d+r],e,esize = result;
if sat then FPSCR.QC = '1';
F6.1.182 VQRSHL

Vector Saturating Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

For truncated results see VQSHL (register).

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode description of saturation on page E1-7131.

Depending on settings in the CPACR, NSACR, and HCPRTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

![64-bit SIMD vector variant](image)

Applies when Q == 0.

VQRSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>

128-bit SIMD vector variant

Applies when Q == 1.

VQRSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>

Decoding for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;

unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;

T1

![64-bit SIMD vector variant](image)

Applies when Q == 0.

VQRSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
128-bit SIMD vector variant

Applies when Q == 1.

VQRSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;

Assembler symbols

<c> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
    For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:
    S8 when U = 0, size = 00
    S16 when U = 0, size = 01
    S32 when U = 0, size = 10
    S64 when U = 0, size = 11
    U8 when U = 1, size = 00
    U16 when U = 1, size = 01
    U32 when U = 1, size = 10
    U64 when U = 1, size = 11

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<d> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<m> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<n> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(esize) result;
    boolean sat;
    for r = 0 to regs-1
        for e = 0 to elements-1
            integer element = Int(Elem[D[m+r], e, esize], unsigned);
            integer shift = SInt(Elem[D[n+r], e, esize]<7:0>);
            if shift >= 0 then // left shift
                element = element << shift;
            else               // rounding right shift
                shift = -shift;
                element = (element + (1 << (shift - 1))) >> shift;
(result, sat) = SatQ(element, esize, unsigned);
Elem[D[d+r], e, esize] = result;
if sat then FPSCR.QC = '1';
## F6.1.183 VQRSHRN (zero)

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the signed rounded results in a doubleword vector.

This instruction is a pseudo-instruction of the VQMOVN, VQMOVUN instruction. This means that:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

### A1

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>size</td>
<td>1 0</td>
<td>Vd</td>
<td>0 0 1 0 1</td>
</tr>
</tbody>
</table>

**Signed result variant**

VQRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

and is never the preferred disassembly.

### T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>size</td>
<td>1 0</td>
<td>Vd</td>
<td>0 0 1 0 1</td>
</tr>
</tbody>
</table>

**Signed result variant**

VQRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

and is never the preferred disassembly.

### Assembler symbols

- `<c>` For encoding A1: see [Standard assembler syntax fields on page F1-7228](#). This encoding must be unconditional.
  
  For encoding T1: see [Standard assembler syntax fields on page F1-7228](#).

- `<q>` See [Standard assembler syntax fields on page F1-7228](#).

- `<dt>` Is the data type for the elements of the operand, encoded in the "op<0>:size" field. It can have the following values:

  - S16 when op<0> = 0, size = 00
  - S32 when op<0> = 0, size = 01
  - S64 when op<0> = 0, size = 10
  - U16 when op<0> = 1, size = 00
U32 when op<0> = 1, size = 01
U64 when op<0> = 1, size = 10

The following encodings are reserved:
• op<0> = 0, size = 11.
• op<0> = 1, size = 11.

<d> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**Operation for all encodings**

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.
F6.1.184  \textbf{VQRSHRN, VQRSHRUN}

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the rounded results in a doubleword vector.

For truncated results, see \texttt{VQSHL} (register).

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, \texttt{FPSCR.QC}, is set if saturation occurs. For details see \texttt{Pseudocode description of saturation} on page \texttt{E1-7131}.

Depending on settings in the \texttt{CPACR}, \texttt{NSACR}, and \texttt{HCPTR} registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see \textit{Enabling Advanced SIMD and floating-point support} on page \texttt{G1-8996}.

\textbf{Signed result variant}

Applies when  \((\text{imm6} == 000xxx) \&\& \text{op} == 1\).

\texttt{VQRSHRN\{<c>\}{<q>}.<type><size> <Dd>, <Qm>, #<imm>}

\textbf{Unsigned result variant}

Applies when  \(U == 1 \&\& (\text{imm6} == 000xxx) \&\& \text{op} == 0\).

\texttt{VQRSHRUN\{<c>\}{<q>}.<type><size> <Dd>, <Qm>, #<imm>}

\textbf{Decode for all variants of this encoding}

\begin{verbatim}
if \text{imm6} IN {'000xxx'} then SEE "Related encodings";
if U == '0' \&\& \text{op} == '0' then SEE "VRSHRN";
if \text{Vm}<D> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case \text{imm6} of
when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - \text{UInt}(\text{imm6});
when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - \text{UInt}(\text{imm6});
when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - \text{UInt}(\text{imm6});
src_unsigned = (U == '1' \&\& \text{op} == '1');  dest_unsigned = (U == '1');
d = \text{UInt}(D:Vd);  m = \text{UInt}(M:Vm);
\end{verbatim}

T1

\begin{verbatim}
|15 14 13 12|11 10 9 8|7 6 5 | 0 |15|12|10 9 8|7 6 5 4|3 |0 |
|\hline
1 1 1 |1 |1 |1 |1 |D imm6 | Vd | 1 0 0 |op |0 |1 |M |1 | Vm |
\end{verbatim}
Signed result variant

Applies when !(imm6 == 000xxx) & op == 1.

VQRSHRN<c>{<q>}{<q>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Unsigned result variant

Applies when U == 1 & !(imm6 == 000xxx) & op == 0.

VQRSHRUN<c>{<q>}{<q>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Decode for all variants of this encoding

if imm IN {'000xxx'} then SEE "Related encodings";
if U == '0' & op == '0' then SEE "VRSHRN";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
  when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
  when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
  when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' & op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);

Notes for all encodings

Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<type> For the signed result variant: is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:
S when U = 0
U when U = 1

For the unsigned result variant: is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:
S when U = 1

<size> Is the data size for the elements of the vectors, encoded in the "imm6<5:3>" field. It can have the following values:
16 when imm<5:3> = 001
32 when imm<5:3> = 01x
64 when imm<5:3> = 1xx

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<imm> Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.
Operation for all encodings
if ConditionPassed() then
   EncodingSpecificOperations(); CheckAdvSIMDEnabled();
   round_const = 1 << (shift_amount - 1);
   for e = 0 to elements-1
      operand = Int(Elem[Qin][m>>1],e,2*esize], src_unsigned);
      (result, sat) = SatQ((operand + round_const) >> shift_amount, esize, dest_unsigned);
      Elem[D,d,e,esize] = result;
      if sat then FPSCR.QC = '1';
F6.1.185 VQRSHRUN (zero)

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the unsigned rounded results in a doubleword vector.

This instruction is a pseudo-instruction of the VQMOVN, VQMOVUN instruction. This means that:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

**A1**

Unsigned result variant

VQRSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>

and is never the preferred disassembly.

**T1**

Unsigned result variant

VQRSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>

and is never the preferred disassembly.

**Assembler symbols**

- `<c>` For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
  
  For encoding T1: see Standard assembler syntax fields on page F1-7228.

- `<q>` See Standard assembler syntax fields on page F1-7228.

- `<dt>` Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:
  
  S16 when size = 00
  S32 when size = 01
  S64 when size = 10

  The encoding size = 11 is reserved.
<D> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Q> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Q>*2.

**Operation for all encodings**

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.
F6.1.186   VQSHL, VQSHLU (immediate)

Vector Saturating Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in a second vector.

The operands must all be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are the same size as the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode description of saturation on page E1-7131.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21</th>
<th>16 15 12 11 10 9 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 U 1 D  imm6</td>
<td>Vd 0 1 1 op L Q M 1 Vm</td>
</tr>
</tbody>
</table>

64-bit SIMD vector, signed result variant

Applies when !(imm6 == 000xxx && L == 0) && op == 1 && Q == 0.

VQSHL{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>

64-bit SIMD vector, unsigned result variant

Applies when U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 0.

VQSHLU{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>

128-bit SIMD vector, signed result variant

Applies when !(imm6 == 000xxx && L == 0) && op == 1 && Q == 1.

VQSHL{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>

128-bit SIMD vector, unsigned result variant

Applies when U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 1.

VQSHLU{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>

Decode for all variants of this encoding

if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if U == '0' && op == '0' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = UInt(imm6) - 8;
  when '001xxxx' esize = 16; elements = 4; shift_amount = UInt(imm6) - 16;
  when '01xxxxx' esize = 32; elements = 2; shift_amount = UInt(imm6) - 32;
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = UInt(imm6);
src_unsigned = (U == '1' && op == '1'); dest_unsigned = (U == '1');

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

|15 14 13 12|11 10 9 8 |7 6 5 |0 |15 12|11 10 9 8 |7 6 5 |4 |3 |0 |
|1 1 1 U |1 1 1 1 |D |imm6 |Vd |0 1 1 op L Q M |1 |Vm |

64-bit SIMD vector, signed result variant

Applies when !(imm6 == 000xxx && L == 0) && op == 1 && Q == 0.

VQSHL{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>

64-bit SIMD vector, unsigned result variant

Applies when U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 0.

VQSHLU{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>

128-bit SIMD vector, signed result variant

Applies when !(imm6 == 000xxx && L == 0) && op == 1 && Q == 1.

VQSHL{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>

128-bit SIMD vector, unsigned result variant

Applies when U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 1.

VQSHLU{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>

Decode for all variants of this encoding

if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if U == '0' && op == '0' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = UInt(imm6) - 8;
  when '001xxxx' esize = 16; elements = 4; shift_amount = UInt(imm6) - 16;
  when '01xxxxx' esize = 32; elements = 2; shift_amount = UInt(imm6) - 32;
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = UInt(imm6);
src_unsigned = (U == '1' && op == '1'); dest_unsigned = (U == '1');
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

Notes for all encodings

Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.

Assembler symbols

For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.
<t>
See *Standard assembler syntax fields* on page F1-7228.
</t>

<type>
Is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>when U = 0</td>
</tr>
<tr>
<td>U</td>
<td>when U = 1</td>
</tr>
</tbody>
</table>
</type>

<size>
Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>when L = 0, imm6&lt;5:3&gt; = 001</td>
</tr>
<tr>
<td>16</td>
<td>when L = 0, imm6&lt;5:3&gt; = 01x</td>
</tr>
<tr>
<td>32</td>
<td>when L = 0, imm6&lt;5:3&gt; = 1xx</td>
</tr>
<tr>
<td>64</td>
<td>when L = 1, imm6&lt;5:3&gt; = xxx</td>
</tr>
</tbody>
</table>
</size>

<Qd>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>
Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm>
Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm>
Is an immediate value, in the range 0 to <size>-1, encoded in the "imm6" field.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            operand = Int(Elem[D[m+r],e,esize], src_unsigned);
            (result, sat) = SatQ(operand << shift_amount, esize, dest_unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

F6.1.187 **VQSHL (register)**

Vector Saturating Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

The results are truncated. For rounded results, see **VQRSHL**.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see *Pseudocode description of saturation* on page E1-7131.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

A1

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 | 12 | 11 10 9 8 | 7 6 5 4 | 3 0 |
|-----------|-----------|-----------|----|----|----|----|------|------|------|------|------|------|------|------|
| 1 1 1 1 0 0 | 1 | U | 0 | D | size | Vn | Vd | 0 1 0 0 | N | Q | M | 1 | Vm |

**64-bit SIMD vector variant**

Applies when Q == 0.

VQSHL{<c>}{<q>}.{<dt>}{<Dd>,} {<Dm>,} {<Dn>}

**128-bit SIMD vector variant**

Applies when Q == 1.

VQSHL{<c>}{<q>}.{<dt>}{<Qd>,} {<Qm>,} {<Qn>}

**Decode for all variants of this encoding**

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); reg = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
<td>U</td>
<td>1 1 1 0</td>
<td>D</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector variant**

Applies when Q == 0.

VQSHL{<c>}{<q>}.{<dt>}{<Dd>,} {<Dm>,} {<Dn>}

---

**Pseudocode description of saturation** on page E1-7131.

**Enabling Advanced SIMD and floating-point support** on page G1-8996.
128-bit SIMD vector variant

Applies when Q == 1.

\[ VQSHL{<c>}{<q>}{<dt>} {<Qd>}, {<Qm>}, {<Qn>} \]

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;

unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;

Assembler symbols

\(<c>\)

For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

\(<q>\)

See Standard assembler syntax fields on page F1-7228.

\(<dt>\)

Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:

- S8 when U = 0, size = 00
- S16 when U = 0, size = 01
- S32 when U = 0, size = 10
- S64 when U = 0, size = 11
- U8 when U = 1, size = 00
- U16 when U = 1, size = 01
- U32 when U = 1, size = 10
- U64 when U = 1, size = 11

\(<Qd>\)

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

\(<Qm>\)

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

\(<Qn>\)

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

\(<Dd>\)

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<Dm>\)

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

\(<Dn>\)

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        shift = SInt(Elem[D[r],e,esize]<7:0>);
        operand = Int(Elem[D[r],e,esize], unsigned);
        if shift >= 0 then
            (result,sat) = SatQ(operand << shift, esize, unsigned);
        else
            (result,sat) = SatQ(operand >> -shift, esize, unsigned);
        Elem[D[d+r],e,esize] = result;
        if sat then FPSCR.QC = '1';
F6.1.188  VQSHRN (zero)

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the signed truncated results in a doubleword vector.

This instruction is a pseudo-instruction of the VQMOVN, VQMOVUN instruction. This means that:

• The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
• The assembler syntax is used only for assembly, and is not used on disassembly.
• The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

A1

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1 1</td>
<td>D 1 1</td>
<td>size</td>
<td>1 0</td>
<td>Vd</td>
<td>0 0 1 0 1 x</td>
<td>M 0</td>
</tr>
</tbody>
</table>

Signed result variant

VQSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

and is never the preferred disassembly.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4 3 2 1 0</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1</td>
<td>D 1 1</td>
<td>size</td>
<td>1 0</td>
<td>Vd</td>
<td>0 0 1 0 1 x</td>
</tr>
</tbody>
</table>

Signed result variant

VQSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

and is never the preferred disassembly.

Assembler symbols

{<c>}

For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

{<q>}

See Standard assembler syntax fields on page F1-7228.

{<dt>}

Is the data type for the elements of the operand, encoded in the "op<0>:size" field. It can have the following values:

S16  when op<0> = 0, size = 00
S32  when op<0> = 0, size = 01
S64  when op<0> = 0, size = 10
U16  when op<0> = 1, size = 00
U32 when op<0> = 1, size = 01
U64 when op<0> = 1, size = 10

The following encodings are reserved:
• op<0> = 0, size = 11.
• op<0> = 1, size = 11.

<Ωd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Ωm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Ωm>*2.

**Operation for all encodings**

The description of *VQMOVN, VQMOVUN* gives the operational pseudocode for this instruction.
F6.1.189  VQSHRN, VQSHRUN

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the truncated results in a doubleword vector.

For rounded results, see VQRSHRN, VQRSHRUN.

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode description of saturation on page E1-7131.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

Signed result variant

Applies when !(imm6 == 000xxx) && op == 1.

VQSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Unsigned result variant

Applies when U == 1 && !(imm6 == 000xxx) && op == 0.

VQSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Decode for all variants of this encoding

if imm6 IN {'000xxx'} then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VSHRN";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
  when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
  when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
  when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);

T1
Signed result variant

Applies when !(imm6 == 000xxx) & op == 1.

VQSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Unsigned result variant

Applies when U == 1 & (imm6 == 000xxx) & op == 0.

VQSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Decode for all variants of this encoding

if imm6 IN {'000xxx'} then SEE "Related encodings";
if U == '0' & op == '0' then SEE "VSHRN";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
  when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
  when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
  when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);

Notes for all encodings

Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.

Assembler symbols

<c>  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
    For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q>  See Standard assembler syntax fields on page F1-7228.

<type>  For the signed result variant: is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:
S when U = 0
U when U = 1

For the unsigned result variant: is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:
S when U = 1

<size>  Is the data size for the elements of the vectors, encoded in the "imm6<5:3>" field. It can have the following values:
16 when imm6<5:3> = 001
32 when imm6<5:3> = 01x
64 when imm6<5:3> = 1xx

<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm>  Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<imm>  Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        operand = Int(Elem[Qin[m>=1],e,2*sizeof], src_unsigned);
        (result, sat) = SatQ(operand >> shift_amount, esize, dest_unsigned);
        Elem[D[d,e,esize] = result;
        if sat then FPSCR.QC = '1';
F6.1.190 VQSHRUN (zero)

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the unsigned truncated results in a doubleword vector.

This instruction is a pseudo-instruction of the VQMOVN, VQMOVUN instruction. This means that:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

A1

```
| 31 30 29 28| 27 26 25 24| 23 22 21 20| 19 18 17 16| 15 12| 11 10 9 8 | 7 6 5 4 | 3 0 |
| 1 1 1 1 0 0 1 1 1 D 1 1 | size | 1 0 | Vd | 0 0 1 0 0 1 | 0 | M | 0 | Vm |
```

Unsigned result variant

VQSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>

and is never the preferred disassembly.

T1

```
| 15 14 13 12| 11 10 9 8 | 7 6 5 4 | 3 2 1 0 | 15 12| 11 10 9 8 | 7 6 5 4 | 3 0 |
| 1 1 1 1 1 1 1 1 1 D 1 1 | size | 1 0 | Vd | 0 0 1 0 0 1 | 0 | M | 0 | Vm |
```

Unsigned result variant

VQSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>

and is never the preferred disassembly.

Assembler symbols

- `<c>`
  
  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

  For encoding T1: see Standard assembler syntax fields on page F1-7228.

- `<q>`
  
  See Standard assembler syntax fields on page F1-7228.

- `<dt>`
  
  Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

  - S16 when size = 00
  - S32 when size = 01
  - S64 when size = 10

  The encoding size = 11 is reserved.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**Operation for all encodings**

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.
F6.1.191 VQSUB

Vector Saturating Subtract subtracts the elements of the second operand vector from the corresponding elements of the first operand vector, and places the results in the destination vector. Signed and unsigned operations are distinct.

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSCR.QC, is set if saturation occurs. For details see Pseudocode description of saturation on page E1-7131.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

VQSUB{<c>}{<q>}.<dt> {<Dd>,}<Dn>, <Dm>

64-bit SIMD vector variant

Applies when Q == 0.

VQSUB{<c>}{<q>}.<dt> {<Dd>,}<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VQSUB{<c>}{<q>}.<dt> {<Qd>,}<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

[15 14 13 12|11 10 9 8 | 7 6 5 4 | 3 0 | 1 1 1 1 0 D size | Vn | Vd | 0 0 1 0 N Q M 1 | Vm]

64-bit SIMD vector variant

Applies when Q == 0.

VQSUB{<c>}{<q>}.<dt> {<Dd>,}<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VQSUB{<c>}{<q>}.<dt> {<Qd>,}<Qn>, <Qm>
**Decode for all variants of this encoding**

```plaintext
if Q == '1' || (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

**Assembler symbols**

- `<c>` For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.
  For encoding T1: see *Standard assembler syntax fields on page F1-7228*.

- `<q>` See *Standard assembler syntax fields on page F1-7228*.

- `<dt>` Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:

  - S8 when U = 0, size = 00
  - S16 when U = 0, size = 01
  - S32 when U = 0, size = 10
  - S64 when U = 0, size = 11
  - U8 when U = 1, size = 00
  - U16 when U = 1, size = 01
  - U32 when U = 1, size = 10
  - U64 when U = 1, size = 11

- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.

- `<Qn>` Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as `<Qn>*2.

- `<Qm>` Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        diff = Int(Elem[D[n+r],e,esize], unsigned) - Int(Elem[D[m+r],e,esize], unsigned);
        boolean sat;
        (Elem[D[n+r],e,esize], sat) = SatQ(diff, esize, unsigned);
        if sat then FPSCR.QC = '1';
```

---

*T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions*

*F6.1 Alphabetical list of Advanced SIMD and floating-point instructions*

**ARM DDI 0487I.a**

Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.

ID081822

Non-Confidential
F6.1.192 VRADDHN

Vector Rounding Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are rounded. For truncated results, see VADDHN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>1 1 1 0 0 1 1</th>
<th>D</th>
<th>D != 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vn</td>
<td>Vd</td>
<td>0 1 0 0</td>
</tr>
</tbody>
</table>

**A1 variant**

VRADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

**Decode for this encoding**

if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1

<table>
<thead>
<tr>
<th>1 1 1 1 1 1</th>
<th>D</th>
<th>D != 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vn</td>
<td>Vd</td>
<td>0 1 0 0</td>
</tr>
</tbody>
</table>

**T1 variant**

VRADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

**Decode for this encoding**

if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

**Notes for all encodings**

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

**Assembler symbols**

For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.
<q>
See Standard assembler syntax fields on page F1-7228.
</q>

<dt>
Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

- I16 when size = 00
- I32 when size = 01
- I64 when size = 10

<dd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<q>
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
</q>

<q>
Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
</q>

**Operation for all encodings**

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  round_const = 1 << (esize-1);
  for e = 0 to elements-1
    result = Elem[Qin[n>>1],e,2*esize] + Elem[Qin[m>>1],e,2*esize] + round_const;
    Elem[D[d],e,esize] = result<2*esize-1:esize>;

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.193 VRECPE

Vector Reciprocal Estimate finds an approximate reciprocal of each element in the operand vector, and places the results in the destination vector.

The operand and result elements are the same type, and can be floating-point numbers or unsigned integers.

For details of the operation performed by this instruction see Floating-point reciprocal square root estimate and step on page E1-7156.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1</td>
<td>D 1 1</td>
<td>size 1 1</td>
<td>Vd 0 1 0</td>
<td>F 0 Q M 0</td>
<td>Vm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VRECPE{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VRECPE{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1') then UNDEFINED;
if (size == '01' && (!HaveFP16Ext() || F == '0')) || size IN ('00', '11') then UNDEFINED;
floating_point = (F == '1');
integer size;
integer elements;
case size of
when '01' size = 16; elements = 4;
when '10' size = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1</td>
<td>D 1 1</td>
<td>size 1 1</td>
<td>Vd 0 1 0</td>
<td>F 0 Q M 0</td>
<td>Vm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VRECPE{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VRECPE{<c>}{<q>}.<dt> <Qd>, <Qm>
Decode for all variants of this encoding

if Q == '1' && (Q<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && (HaveFP16Ext() || F == '0')) || size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
floating_point = (F == '1');
integer esize;
integer elements;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<dt> Is the data type for the elements of the vectors, encoded in the "F:size" field. It can have the following values:
    U32 when F = 0, size = 10
    F16 when F = 1, size = 01
    F32 when F = 1, size = 10

<qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see Floating-point reciprocal estimate and step on page E1-7155.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      if floating_point then
        Elem[D[d+r],e,esize] = FPRecipEstimate(Elem[D[m+r],e,esize], StandardFPSCRValue());
      else
        Elem[D[d+r],e,esize] = UnsignedRecipEstimate(Elem[D[m+r],e,esize]);
F6.1.194   VRECPS

Vector Reciprocal Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 2.0, and places the results into the elements of the destination vector.

The operand and result elements are floating-point numbers.

For details of the operation performed by this instruction see Floating-point reciprocal estimate and step on page E1-7155.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when \(Q == 0\).

\[ VRECPS\{\langle c\rangle\}\{\langle q\rangle\}.\{\langle dt\rangle\} \{\langle Dd\rangle, \langle Dn\rangle, \langle Dm\rangle\} \]

128-bit SIMD vector variant

Applies when \(Q == 1\).

\[ VRECPS\{\langle c\rangle\}\{\langle q\rangle\}.\{\langle dt\rangle\} \{\langle Qd\rangle, \langle Qn\rangle, \langle Qm\rangle\} \]

Decode for all variants of this encoding

\[
\begin{align*}
\text{if } Q == '1' \&\& (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') \text{ then UNDEFINED;} \\
\text{if } sz == '1' \&\& !HaveFP16Ext() \text{ then UNDEFINED;} \\
\text{integer esize;} \\
\text{integer elements;} \\
\text{case sz of} \\
\text{when '0' esize = 32; elements = 2;} \\
\text{when '1' esize = 16; elements = 4;} \\
d = UInt(D:Vd); \\
n = UInt(N:Vn); \\
m = UInt(M:Vm); \\
\text{regs = if } Q == '0' \text{ then 1 else 2;} 
\end{align*}
\]

T1

64-bit SIMD vector variant

Applies when \(Q == 0\).

\[ VRECPS\{\langle c\rangle\}\{\langle q\rangle\}.\{\langle dt\rangle\} \{\langle Dd\rangle, \langle Dn\rangle, \langle Dm\rangle\} \]

128-bit SIMD vector variant

Applies when \(Q == 1\).

\[ VRECPS\{\langle c\rangle\}\{\langle q\rangle\}.\{\langle dt\rangle\} \{\langle Qd\rangle, \langle Qn\rangle, \langle Qm\rangle\} \]
Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRANDED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&gt;</td>
<td>For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional. For encoding T1: see Standard assembler syntax fields on page F1-7228.</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>See Standard assembler syntax fields on page F1-7228.</td>
</tr>
<tr>
<td>&lt;dt&gt;</td>
<td>Is the data type for the elements of the vectors, encoded in the &quot;sz&quot; field. It can have the following values: F32 when sz = 0 F16 when sz = 1</td>
</tr>
<tr>
<td>&lt;Qd&gt;</td>
<td>Is the 128-bit name of the SIMD&amp;FP destination register, encoded in the &quot;D:Vd&quot; field as &lt;Qd&gt;*2.</td>
</tr>
<tr>
<td>&lt;Qn&gt;</td>
<td>Is the 128-bit name of the first SIMD&amp;FP source register, encoded in the &quot;N:Vn&quot; field as &lt;Qn&gt;*2.</td>
</tr>
<tr>
<td>&lt;Qm&gt;</td>
<td>Is the 128-bit name of the second SIMD&amp;FP source register, encoded in the &quot;M:Vm&quot; field as &lt;Qm&gt;*2.</td>
</tr>
<tr>
<td>&lt;Dd&gt;</td>
<td>Is the 64-bit name of the SIMD&amp;FP destination register, encoded in the &quot;D:Vd&quot; field.</td>
</tr>
<tr>
<td>&lt;Dn&gt;</td>
<td>Is the 64-bit name of the first SIMD&amp;FP source register, encoded in the &quot;N:Vn&quot; field.</td>
</tr>
<tr>
<td>&lt;Dm&gt;</td>
<td>Is the 64-bit name of the second SIMD&amp;FP source register, encoded in the &quot;M:Vm&quot; field.</td>
</tr>
</tbody>
</table>

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see Floating-point reciprocal estimate and step on page E1-7155.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      Elem[D[d+r],e,esize] = FPRecipStep(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize]);
F6.1.195 VREV16

Vector Reverse in halfwords reverses the order of 8-bit elements in each halfword of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

The following figure shows the operation of VREV16 doubleword operation.

![VREV16 Diagram]

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</th>
<th>15 12 11 10 9 8 7 6 5 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1 1 D 1 1 size 0 0</td>
<td>Vd 0 0 0 1 0 Q M 0</td>
</tr>
<tr>
<td>op</td>
<td></td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VREV16{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VREV16{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if UInt(op)+UInt(size) >= 3 then UNDEFINED;

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
**T1**

64-bit SIMD vector variant

Applies when \( Q = 0 \).

\[
\text{VREV16}\{<c>\}\{<q>\}\.<dt> <Dd>, <Dm>
\]

128-bit SIMD vector variant

Applies when \( Q = 1 \).

\[
\text{VREV16}\{<c>\}\{<q>\}\.<dt> <Qd>, <Qm>
\]

**Decode for all variants of this encoding**

\[
\text{if Uint(op)+Uint(size)} \geq 3 \text{ then UNDEFINED;}
\]

\[
\text{if } Q = '1' \text{ AND (Vd<0> == '1' OR Vm<0> == '1')} \text{ then UNDEFINED;}
\]

\[
esize = 8 << \text{Uint(size)};
\]

\[
\text{integer container_size;}
\]

\[
\text{case op of}
\]

\[
\text{when '10' container_size = 16;}
\]

\[
\text{when '01' container_size = 32;}
\]

\[
\text{when '00' container_size = 64;}
\]

\[
\text{integer containers = 64 DIV container_size;}
\]

\[
\text{integer elements_per_container = container_size DIV esize;}
\]

\[
d = \text{Uint(D:Vd)}; m = \text{Uint(M:Vm)}; \text{regs = if } Q = '0' \text{ then 1 else 2;}
\]

**Assembler symbols**

\(<c>\) \hspace{1cm} \text{For encoding A1: see } \textit{Standard assembler syntax fields on page F1-7228}. \text{This encoding must be unconditional.}

\text{For encoding T1: see } \textit{Standard assembler syntax fields on page F1-7228}.\]

\(<q>\) \hspace{1cm} \text{See } \textit{Standard assembler syntax fields on page F1-7228}.\]

\(<dt>\) \hspace{1cm} \text{Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:}

\[
8 \text{ when size = 00}
\]

\text{The following encodings are reserved:}

\[
\text{• size = 01.}
\]

\[
\text{• size = 1x.}
\]

\(<Qd>\) \hspace{1cm} \text{Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as } <Qd>^2.\]

\(<Qm>\) \hspace{1cm} \text{Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as } <Qm>^2.\]

\(<Dd>\) \hspace{1cm} \text{Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.}\]

\(<Dm>\) \hspace{1cm} \text{Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.}
Operation for all encodings

if \texttt{ConditionPassed()} then
  \texttt{EncodingSpecificOperations(); \ CheckAdvSIMDEnabled();}

  \texttt{bits(64) result;}
  \texttt{integer element;}
  \texttt{integer rev\_element;}
  \texttt{for } r = 0 \texttt{ to regs-1} \\
  \texttt{element = 0;}
  \texttt{for } c = 0 \texttt{ to containers-1} \\
  \texttt{rev\_element = element + elements\_per\_container - 1;}
  \texttt{for } e = 0 \texttt{ to elements\_per\_container-1} \\
  \texttt{Elem[result, rev\_element, esize] = Elem[D[m+r], element, esize];}
  \texttt{element = element + 1;}
  \texttt{rev\_element = rev\_element - 1;}
  \texttt{D[d+r] = result;}

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.196 VREV32

Vector Reverse in words reverses the order of 8-bit or 16-bit elements in each word of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

The following figure shows the operation of VREV32 doubleword operations.

Depending on settings in the CPACR, NSACR, and HCPR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VREV32{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VREV32{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
T1

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | Vd | 0  | 0  | 0  | 0  | 1  | Q | M | 0  | Vm |

**64-bit SIMD vector variant**

Applies when Q == 0.

```
VREV32{<c>}{<q>}.<dt> <Dd>, <Dm>
```

**128-bit SIMD vector variant**

Applies when Q == 1.

```
VREV32{<c>}{<q>}.<dt> <Qd>, <Qm>
```

**Decode for all variants of this encoding**

```cpp
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);
integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

**Assembler symbols**

<

For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields on page F1-7228*.

<q>

See *Standard assembler syntax fields on page F1-7228*.

<dt>

Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

- 8 when size == 00
- 16 when size == 01

The encoding size = 1x is reserved.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<0d>

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<0m>

Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```cpp
if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
```
bits(64) result;
integer element;
integer rev_element;
for r = 0 to regs-1
  element = 0;
  for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
      Elem[result, rev_element, esize] = Elem[D[m+r], element, esize];
      element = element + 1;
      rev_element = rev_element - 1;
  D[d+r] = result;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.197  VREV64

Vector Reverse in doublewords reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

The following figure shows the operation of VREV64 doubleword operations.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 12 11 10 9 8 7 6 5 4 3 0 |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| 1 1 1 1 0 0 | 1 1 1 1 | D 1 1 | size 0 0 | Vd 0 0 0 | 0 0 | Q 0 | M 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VREV64{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VREV64{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
64-bit SIMD vector variant
Applies when $Q = 0$.

VREV64{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant
Applies when $Q = 1$.

VREV64{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```plaintext
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if $Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
    integer containers = 64 DIV container_size;
    integer elements_per_container = container_size DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if $Q == '0' then 1 else 2;
```

Assembler symbols

- `<c>`
  - For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.
  - For encoding T1: see *Standard assembler syntax fields on page F1-7228*.

- `<q>`
  - See *Standard assembler syntax fields on page F1-7228*.

- `<dt>`
  - Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:
    - 8 when $size = 00$
    - 16 when $size = 01$
    - 32 when $size = 10$
  - The encoding $size = 11$ is reserved.

- `<Qd>`
  - Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as $<Qd>*2$.

- `<Qm>`
  - Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as $<Qm>*2$.

- `<Dd>`
  - Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- `<Dm>`
  - Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();

  bits(64) result;
  integer element;
  integer rev_element;
  for r = 0 to regs-1
    element = 0;
    for c = 0 to containers-1
      rev_element = element + elements_per_container - 1;
      for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[D[m+r], element, esize];
        element = element + 1;
        rev_element = rev_element - 1;
      D[d+r] = result;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.198   VRHADD

Vector Rounding Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector.

The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The results of the halving operations are rounded. For truncated results, see VHADD.

Depending on settings in the CPACR, NSACR, and HCPR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

```
|31 30 29 28|27 26 25 24|23 22 21 20|19|16|15|12|11 10 9|8|7 6 5 4|3|0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|1|1|1|0|0|1|0|0|D|size|Vn|Vd|0|0|0|1|N|Q|M|0|Vm|
```

64-bit SIMD vector variant

Applies when Q == 0.

VRHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VRHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

T1

```
|15 14 13 12|11 10 9|8|7 6 5 4|3|0 |15|12|11 10 9|8|7 6 5 4|3|0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|1|1|1|1|1|1|0|D|size|Vn|Vd|0|0|0|1|N|Q|M|0|Vm|
```

64-bit SIMD vector variant

Applies when Q == 0.

VRHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VRHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Assembler symbols

<>()  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q>  See Standard assembler syntax fields on page F1-7228.

<dt>  Is the data type for the elements of the operands, encoded in the "U:size" field. It can have the following values:
S8 when U = 0, size = 00
S16 when U = 0, size = 01
S32 when U = 0, size = 10
U8 when U = 1, size = 00
U16 when U = 1, size = 01
U32 when U = 1, size = 10

<Qd>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>  Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>  Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>  Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>  Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = (op1 + op2 + 1) >> 1;
            Elem[D[d+r],e,esize] = result<esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.199   VRINTA (Advanced SIMD)

Vector Round floating-point to integer towards Nearest with Ties to Away rounds a vector of floating-point values to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

64-bit SIMD vector variant

Applies when $Q == 0$.

VRINTA{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when $Q == 1$.

VRINTA{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if $op<2> != op<0>$ then SEE "Related encodings";
if $Q == '1' && (Vd<0> == '1' || Vm<0> == '1')$ then UNDEFINED;
if $(size == '01' || !HaveFP16Ext())$ then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM($op<2>:NOT(op<1>)$);  exact = FALSE;
integer esize;
integer elements;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if $Q == '0'$ then 1 else 2;

T1

64-bit SIMD vector variant

Applies when $Q == 0$.

VRINTA{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when $Q == 1$.

VRINTA{<q>}.<dt> <Qd>, <Qm>
Decode for all variants of this encoding

if op<2> != op<0> then SEE "Related encodings";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
integer esize;
integer elements;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See Advanced SIMD two registers misc on page F3-7317 for the T32 instruction set, or Advanced SIMD two registers misc on page F4-7425 for the A32 instruction set.

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:
  F16 when size = 01
  F32 when size = 10
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
  for e = 0 to elements-1
    op1 = Elem[D[m+r],e,esize];
    result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
    Elem[D[d+r],e,esize] = result;
F6.1.200  VRINTA (floating-point)

Round floating-point to integer to Nearest with Ties to Away rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|-------------|-------------|-------------|-------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | D | 1 | 1 | 1 | 0 | 0 | 0 | Vd | 1 | 0 | !=00 | 0 | 1 | M | 0 | Vm |

**Half-precision scalar variant**

Applies when size == 01.

VRINTA{<q>}.F16 <Sd>, <Sm>

**Single-precision scalar variant**

Applies when size == 10.

VRINTA{<q>}.F32 <Sd>, <Sm>

**Double-precision scalar variant**

Applies when size == 11.

VRINTA{<q>}.F64 <Dd>, <Dm>

**Decode for all variants of this encoding**

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;

rounding = FPDecodeRM(RM);  exact = FALSE;

integer esize;
integer d;
integer m;

case size of
when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

T1

| 15 14 13 12 | 11 10 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|-------------|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | D | 1 | 1 | 1 | 0 | 0 | 0 | Vd | 1 | 0 | !=00 | 0 | 1 | M | 0 | Vm |

**Half-precision scalar variant**

Applies when size == 01.

VRINTA{<q>}.F16 <Sd>, <Sm>

**Single-precision scalar variant**

Applies when size == 10.

VRINTA{<q>}.F32 <Sd>, <Sm>
**Double-precision scalar variant**

Applies when size == 11.

```
VRINTA(<q>).F64 <Dd>, <Dm>
```

**Decode for all variants of this encoding**

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
CONSTRAINED UNPREDICTABLE behavior
If InITBlock(), then one of the following behaviors must occur:
  • The instruction is UNDEFINED.
  • The instruction executes as if it passes the Condition code check.
  • The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```
EncodingSpecificOperations(); CheckVFENable(TRUE);
case esize of
  when 16
    S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
```
F6.1.201 VRINTM (Advanced SIMD)

Vector Round floating-point to integer towards -Infinity rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VRINTM{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VRINTM{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
integer esize;
integer elements;
case size of
when '01' esize = 16; elements = 4;
when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when Q == 0.

VRINTM{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VRINTM{<q>}.<dt> <Qd>, <Qm>
Decode for all variants of this encoding

if op<2> != op<0> then SEE "Related encodings";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
integer esize;
integer elements;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See Advanced SIMD two registers misc on page F3-7317 for the T32 instruction set, or Advanced SIMD two registers misc on page F4-7425 for the A32 instruction set.

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:
  F16 when size = 01
  F32 when size = 10
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
  for e = 0 to elements-1
    op1 = Elem[D[m+r],e,esize];
    result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
    Elem[D[d+r],e,esize] = result;
F6.1.202 VRINTM (floating-point)

Round floating-point to integer towards -Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 0 1</td>
<td>D 1 1 1 0</td>
<td>1</td>
<td>Vd</td>
<td>1 0</td>
<td>!=00</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Half-precision scalar variant

Applies when size == 01.

VRINTM{<q>}.F16 <Sd>, <Sm>

Single-precision scalar variant

Applies when size == 10.

VRINTM{<q>}.F32 <Sd>, <Sm>

Double-precision scalar variant

Applies when size == 11.

VRINTM{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;

rounding = FPDecodeRM(RM); exact = FALSE;

integer esize;
integer d;
integer m;

```
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 0 1</td>
<td>D 1 1 1 0</td>
<td>1</td>
<td>Vd</td>
<td>1 0</td>
<td>!=00</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Half-precision scalar variant

Applies when size == 01.

VRINTM{<q>}.F16 <Sd>, <Sm>

Single-precision scalar variant

Applies when size == 10.

VRINTM{<q>}.F32 <Sd>, <Sm>
Double-precision scalar variant

Applies when size == 11.

VRINTM(<q>).F64 <Dd>, <Dm>

Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

See Standard assembler syntax fields on page F1-7228.

<q> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Db> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
F6.1.203 VRINTN (Advanced SIMD)

Vector Round floating-point to integer to Nearest rounds a vector of floating-point values to integral floating-point values of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

64-bit SIMD vector variant

Applies when \( Q = 0 \).

VRINTN\{<q>\}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when \( Q = 1 \).

VRINTN\{<q>\}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if op<2> \(!= \) op<0> then see “Related encodings”;
if \( Q = '1' \) \&\& (Vd<0> \( = '1' \) \|\| Vm<0> \( = '1' \)) then UNDEFINED;
if \( \{\text{size} = '01' \} \&\& \! \text{HaveFP16Ext()} \} \| \text{size IN \{‘00’, ‘11’\}} \) then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDEncodeRM(op<2>:NOT(op<1>)); exact = FALSE;
integer esize;
integer elements;
case size of
when ‘01’ esize = 16; elements = 4;
when ‘10’ esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q = ‘0’ then 1 else 2;

T1

64-bit SIMD vector variant

Applies when \( Q = 0 \).

VRINTN\{<q>\}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when \( Q = 1 \).

VRINTN\{<q>\}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if op<2> \(!= \) op<0> then see “Related encodings”;
if InITBlock() then UNPREDICTABLE;
if \( Q = '1' \) \&\& (Vd<0> \( = '1' \) \|\| Vm<0> \( = '1' \)) then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
    d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

**CONSTRAINED UNPREDICTABLE behavior**

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Notes for all encodings**

Related encodings: See *Advanced SIMD two registers misc on page F3-7317* for the T32 instruction set, or *Advanced SIMD two registers misc on page F4-7425* for the A32 instruction set.

**Assembler symbols**

<q> See *Standard assembler syntax fields on page F1-7228.*

<dt> Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:

- F16 when size = '01'
- F32 when size = '10'

<q>d> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <q>d>*2.

<q>m> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <q>m>*2.

<dt>d> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<dt>m> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
F6.1.204 VRINTN (floating-point)

Round floating-point to integer to Nearest rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 12 11 10 9 8 7 6 5 4 3 0 |
|-------------|-------------|-------------|-------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 1 1 1 1 1 | 1 1 1 1 1 1 | 0 1 | D | 1 1 1 | 0 1 | Vd | 1 0 | !=00 | 0 1 | M | 0 | Vm |

---

Half-precision scalar variant

Applies when size == 01.

VRINTN(<q>).F16 <Sd>, <Sm>

Single-precision scalar variant

Applies when size == 10.

VRINTN(<q>).F32 <Sd>, <Sm>

Double-precision scalar variant

Applies when size == 11.

VRINTN(<q>).F64 <Dd>, <Dm>

Decode for all variants of this encoding

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

T1

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 3 2 1 0 | 15 12 11 10 9 8 | 7 6 5 4 3 0 |
|-------------|-------------|-------------|-------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 1 1 1 1 1 | 1 1 1 1 1 1 | 0 1 | D | 1 1 1 | 0 1 | Vd | 1 0 | !=00 | 0 1 | M | 0 | Vm |

---

Half-precision scalar variant

Applies when size == 01.

VRINTN(<q>).F16 <Sd>, <Sm>

Single-precision scalar variant

Applies when size == 10.

VRINTN(<q>).F32 <Sd>, <Sm>
**Double-precision scalar variant**

Applies when \( \text{size} == 11 \).

VRINTN\(<q>\).F64 <Dd>, <Dm>

**Decode for all variants of this encoding**

- If `InITBlock()` then UNPREDICTABLE;
- If `\text{size} == \text{\texttt{00}}` || (\text{size} == \texttt{01} & & !\text{HaveFP16Ext()}) then UNDEFINED;
- \text{rounding} = \text{FPDecodeRM(RM)}; \text{exact} = \text{FALSE};
- \text{integer esize};
- \text{integer d};
- \text{integer m};
- case size of
  - when \( \text{\texttt{01}} \) \text{esize} = 16; \text{d} = \text{UInt(Vd:D)}; \text{m} = \text{UInt(Vm:M)};
  - when \( \text{\texttt{10}} \) \text{esize} = 32; \text{d} = \text{UInt(Vd:D)}; \text{m} = \text{UInt(Vm:M)};
  - when \( \text{\texttt{11}} \) \text{esize} = 64; \text{d} = \text{UInt(D:Vd)}; \text{m} = \text{UInt(M:Vm)};

**CONSTRAINED UNPREDICTABLE behavior**

If `InITBlock()`, then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

- \(<q>\) See Standard assembler syntax fields on page F1-7228.
- \(<Sd>\) Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- \(<Sm>\) Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
- \(<Dd>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- \(<Dm>\) Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
- case esize of
  - when 16
    - \( S[d] = \text{Zeros}(16) \): \text{FPRoundInt}(S[m]<15:0>, \text{FPSCR}[], \text{rounding}, \text{exact});
  - when 32
    - \( S[d] = \text{FPRoundInt}(S[m], \text{FPSCR}[], \text{rounding}, \text{exact});
  - when 64
    - \( D[d] = \text{FPRoundInt}(D[m], \text{FPSCR}[], \text{rounding}, \text{exact});

F6.1.205 VRINTP (Advanced SIMD)

Vector Round floating-point to integer towards +Infinity rounds a vector of floating-point values to integral floating-point values of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 12 11 10 9 | 7 6 5 4 | 3 | 0 |
|---|---|---|---|---|---|---|---|---|
| 1 1 1 1 0 0 1 1 | D | 1 | 1 | size | 1 | 0 | Vd | 0 | 1 1 1 1 | Q | M | 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VRINTP{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VRINTP{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if {size == '01' && !HaveFP16Ext()} || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12|11 10 9 8 | 7 6 5 4 | 3 | 2 | 1 | 0 |15 12|11 10 9 | 7 6 5 4 | 3 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 1 1 1 1 1 1 1 | D | 1 | 1 | size | 1 | 0 | Vd | 0 | 1 1 1 1 | Q | M | 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VRINTP{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VRINTP{<q>}.<dt> <Qd>, <Qm>
Decode for all variants of this encoding

if op<2> != op<0> then SEE "Related encodings";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRANDED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

Related encodings: See Advanced SIMD two registers misc on page F3-7317 for the T32 instruction set, or Advanced SIMD two registers misc on page F4-7425 for the A32 instruction set.

Assembler symbols

<q>  See Standard assembler syntax fields on page F1-7228.
<dt>  Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:
    F16  when size = 01
    F32  when size = 10
<Qd>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>  Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>  Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[0:m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[0:d+r],e,esize] = result;
F6.1.206  VRINTP (floating-point)

Round floating-point to integer towards +Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 0 1</td>
<td>D</td>
<td>1 1 1 0 1 0</td>
<td>Vd</td>
<td>1 0</td>
<td>!=00</td>
<td>0 1</td>
<td>M</td>
</tr>
</tbody>
</table>
```

**Half-precision scalar variant**

Applies when size == 01.

VRINTP{<q>}.F16 <Sd>, <Sm>

**Single-precision scalar variant**

Applies when size == 10.

VRINTP{<q>}.F32 <Sd>, <Sm>

**Double-precision scalar variant**

Applies when size == 11.

VRINTP{<q>}.F64 <Dd>, <Dm>

**Decode for all variants of this encoding**

```c
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 0 1</td>
<td>D</td>
<td>1 1 1 0 1 0</td>
<td>Vd</td>
<td>1 0</td>
<td>!=00</td>
<td>0 1</td>
<td>M</td>
</tr>
</tbody>
</table>
```

**Half-precision scalar variant**

Applies when size == 01.

VRINTP{<q>}.F16 <Sd>, <Sm>

**Single-precision scalar variant**

Applies when size == 10.

VRINTP{<q>}.F32 <Sd>, <Sm>
**Double-precision scalar variant**

Applies when size == 11.

VRINTP(<q>).F64 <Dd>, <Dm>

**Decode for all variants of this encoding**

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*.

**Assembler symbols**

- `<q>` See *Standard assembler syntax fields on page F1-7228*.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
  when 16  
    S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
  when 32  
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
  when 64  
    D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
F6.1.207   VRINTR

Round floating-point to integer rounds a floating-point value to an integral floating-point value of the same size using the rounding mode specified in the FPSCR. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

Half-precision scalar variant
Applies when size == 01.
VRINTR{c}{q}.F16 <Sd>, <Sm>

Single-precision scalar variant
Applies when size == 10.
VRINTR{c}{q}.F32 <Sd>, <Sm>

Double-precision scalar variant
Applies when size == 11.
VRINTR{c}{q}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

T1

Half-precision scalar variant
Applies when size == 01.
VRINTR{c}{q}.F16 <Sd>, <Sm>

Single-precision scalar variant
Applies when size == 10.
VRINTR{c}{q}.F32 <Sd>, <Sm>
Double-precision scalar variant

Applies when size == 11.

`VRINTR{<c>}{<q>}.F64 <Dd>, <Dm>`

**Decode for all variants of this encoding**

```c
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
exact = FALSE;
integer esize;
integer d;
integer m;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

**CONSTRAINED UNPREDICTABLE behavior**

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

`<c>` See Standard assembler syntax fields on page F1-7228.

`<q>` See Standard assembler syntax fields on page F1-7228.

`<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

`<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

`<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

`<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
    when 32
        S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
    when 64
        D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
```
F6.1.208 VRINTX (Advanced SIMD)

Vector round floating-point to integer inexact rounds a vector of floating-point values to integral floating-point values of the same size, using the Round to Nearest rounding mode, and raises the Inexact exception when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

| 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 12|11 10 9 8|7 6 5 4|3 0 |  
| 1 1 1 1 0 0 1 1 | D 1 1 | size 1 0 | Vd 0 1 0 0 | Q M 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VRINTX{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VRINTX{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_TIEEVEN;  exact = TRUE;
integer esize;
integer elements;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

| 15 14 13 12|11 10 9 8|7 6 5 4|3 2 1 0|15 12|11 10 9 8|7 6 5 4|3 0 |  
| 1 1 1 1 1 1 1 1 | D 1 1 | size 1 0 | Vd 0 1 0 0 | Q M 0 | Vm |
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:
    F16 when size = 01
    F32 when size = 10
<qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <qd>*2.
<qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <qm>*2.
<dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
F6.1.209  VRINTX (floating-point)

Round floating-point to integer inexact rounds a floating-point value to an integral floating-point value of the same size, using the rounding mode specified in the FPSCR, and raises an Inexact exception when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

| [31] | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| !="1111" | 1 | 1 | 1 | 0 | 1 | D | 1 | 0 | 1 | 1 | 1 | Vd | 1 | 0 | size | 0 | 1 | M | 0 | Vm |

cond

Half-precision scalar variant

Applies when size == 01.

VRINTX{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar variant

Applies when size == 10.

VRINTX{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar variant

Applies when size == 11.

VRINTX{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
exact = TRUE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
T1

<table>
<thead>
<tr>
<th>[15]</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>0</td>
<td>1</td>
<td>M</td>
<td>0</td>
<td>Vm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Half-precision scalar variant

Applies when size == 01.

VRINTX{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar variant

Applies when size == 10.

VRINTX{<c>}{<q>}.F32 <Sd>, <Sm>
**Double-precision scalar variant**

Applies when size == 11.

VRINTX{<c>}{<q>}.F64 <Dd>, <Dm>

**Decode for all variants of this encoding**

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && !InITBlock() then UNPREDICTABLE;

exact = TRUE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**

- `<c>` See Standard assembler syntax fields on page F1-7228.
- `<q>` See Standard assembler syntax fields on page F1-7228.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- `<Sm>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
  rounding = FPRoundingMode(FPSCR[]);
case esize of
  when 16
    S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
F6.1.210 VRINTZ (Advanced SIMD)

Vector round floating-point to integer towards Zero rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

VRINTZ{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VRINTZ{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_ZERO;  exact = FALSE;
integer esize;
integer elements;
case size of
when '01' esize = 16; elements = 4;
when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when Q == 0.

VRINTZ{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VRINTZ{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_ZERO;  exact = FALSE;
integer esize;
integer elements;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;

CONSTRUANCED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings

For more information about the CONSTRUANCED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:
  F16 when size = 01
  F32 when size = 10
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
  for e = 0 to elements-1
    op1 = Elem[D[m+r],e,esize];
    result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
    Elem[D[d+r],e,esize] = result;
F6.1.211 VRINTZ (floating-point)

Round floating-point to integer towards Zero rounds a floating-point value to an integral floating-point value of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A1

Half-precision scalar variant

Applies when size == 01.
VRINTZ{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar variant

Applies when size == 10.
VRINTZ{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar variant

Applies when size == 11.
VRINTZ{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
T1

Half-precision scalar variant

Applies when size == 01.
VRINTZ{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar variant

Applies when size == 10.
VRINTZ{<c>}{<q>}.F32 <Sd>, <Sm>
Double-precision scalar variant

Applies when size == 11.

VRINTZ{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<>{ See Standard assembler syntax fields on page F1-7228.
<q> See Standard assembler syntax fields on page F1-7228.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Do> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Mo> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
F6.1.212   VRSHL

Vector Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see VSHL.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VRSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>

128-bit SIMD vector variant

Applies when Q == 1.

VRSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when Q == 0.

VRSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>

128-bit SIMD vector variant

Applies when Q == 1.

VRSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;

Assembler symbols

<<> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.
<q> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:
S8 when U = 0, size = 00
S16 when U = 0, size = 01
S32 when U = 0, size = 10
S64 when U = 0, size = 11
U8 when U = 1, size = 00
U16 when U = 1, size = 01
U32 when U = 1, size = 10
U64 when U = 1, size = 11

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Operation for all encodings

if ConditionPassed() then
EncodingSpecificOperations();  CheckAdvSIMDEnabled();
integer result;
for r = 0 to regs-1
  for e = 0 to elements-1
    integer element = Int(Elem[D[m+r], e, esize], unsigned);
    integer shift = SInt(Elem[D[n+r], e, esize]<7:0>);
    if shift >= 0 then // left shift
      result = element << shift;
    else                // rounding right shift
      shift = -shift;
      result = (element + (1 << (shift - 1))) >> shift;
    Elem[D[d+r], e, esize] = result<esize-1:0>;


Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.213 VRSHR

Vector Rounding Shift Right takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see VSHR.

The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when !(imm6 == 000xxx && L == 0) && Q == 0.

VRSHR{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>

128-bit SIMD vector variant

Applies when !(imm6 == 000xxx && L == 0) && Q == 1.

VRSHR{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>

Decode for all variants of this encoding

if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
unsigned = (U == '1'); d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when !(imm6 == 000xxx && L == 0) && Q == 0.

VRSHR{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
128-bit SIMD vector variant

Applies when !(imm6 == 000xxx && L == 0) && Q == 1.

VRSH{<c>}{<q>}.<type><size>{<Qd>,} <Qm>, #<imm>

Decode for all variants of this encoding

if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
  when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
  when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
  when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
  unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Notes for all encodings

Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.

Assembler symbols

<c> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<type> Is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:
  S when U = 0
  U when U = 1

<size> Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:
  8 when L = 0, imm6<5:3> = 001
  16 when L = 0, imm6<5:3> = 01x
  32 when L = 0, imm6<5:3> = 1xx
  64 when L = 1, imm6<5:3> = xxx

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>. 
Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  round_const = 1 << (shift_amount - 1);
  for r = 0 to regs-1
    for e = 0 to elements-1
      result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) >> shift_amount;
      Elem[D[d+r],e,esize] = result<esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.214  VRSHR (zero)

Vector Rounding Shift Right copies the contents of one SIMD register to another. This instruction is a pseudo-instruction of the VORR (register) instruction. This means that:

- The encodings in this description are named to match the encodings of VORR (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VORR (register) gives the operational pseudocode for this instruction.

### 64-bit SIMD vector variant

Applies when \( Q = 0 \).

\[
\text{VRSHR}\{<c>\}{<q>}.<dt> <Dd>, <Dm>, #0
\]

is equivalent to

\[
\text{VORR}\{<c>\}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
\]

and is never the preferred disassembly.

### 128-bit SIMD vector variant

Applies when \( Q = 1 \).

\[
\text{VRSHR}\{<c>\}{<q>}.<dt> <Qd>, <Qm>, #0
\]

is equivalent to

\[
\text{VORR}\{<c>\}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>
\]

and is never the preferred disassembly.

### T1

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 | 0 | 16 | 15 12 | 11 10 9 8 | 7 6 5 4 | 3 | 0 |
|--------------|----------|----------|---|---|-----|----------|----------|----------|---|---|-----|
| 1 1 1 0 | 1 1 1 1 0 D 1 0 Vn | Vd | 0 0 0 1 N Q M 1 Vm |
VRSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0

is equivalent to
VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>

and is never the preferred disassembly.

**Assembler symbols**

- `<c>` For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.
  
  For encoding T1: see *Standard assembler syntax fields on page F1-7228*.

- `<q>` See *Standard assembler syntax fields on page F1-7228*.

- `<dt>` Is the data type for the elements of the vectors, and must be one of: S8, S16, S32, S64, U8, U16, U32 or U64.

- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>]*2.

- `<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as `<Qm>]*2.

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

**Operation for all encodings**

The description of VORR (register) gives the operational pseudocode for this instruction.
F6.1.215  VRSHRN

Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see VSHRN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

A1 variant

Applies when \text{imm6} \neq 000xxx.

\text{VRSHRN}\langle c\rangle\langle q\rangle.I\langle \text{size}\rangle \langle Dd\rangle, \langle Qm\rangle, \#\langle \text{imm}\rangle

Decode for this encoding

\begin{verbatim}
if \text{imm6} \in \{000xxx\} then SEE "Related encodings";
if \text{Vm}\langle 0\rangle == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case \text{imm6} of
when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - \text{UInt}(\text{imm6});
when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - \text{UInt}(\text{imm6});
when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - \text{UInt}(\text{imm6});
d = \text{UInt}(D:Vd);  m = \text{UInt}(M:Vm);
\end{verbatim}

T1

T1 variant

Applies when \text{imm6} \neq 000xxx.

\text{VRSHRN}\langle c\rangle\langle q\rangle.I\langle \text{size}\rangle \langle Dd\rangle, \langle Qm\rangle, \#\langle \text{imm}\rangle

Decode for this encoding

\begin{verbatim}
if \text{imm6} \in \{000xxx\} then SEE "Related encodings";
if \text{Vm}\langle 0\rangle == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case \text{imm6} of
when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - \text{UInt}(\text{imm6});
when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - \text{UInt}(\text{imm6});
\end{verbatim}
when '1xxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  d = UInt(D:Vd);  m = UInt(M:Vm);

Notes for all encodings

Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.

Assembler symbols

<> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<> See Standard assembler syntax fields on page F1-7228.

<size> Is the data size for the elements of the vectors, encoded in the "imm6<5:3>" field. It can have the following values:
  16  when imm6<5:3> = 001
  32  when imm6<5:3> = 01x
  64  when imm6<5:3> = 1xx

<DD> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<imm> Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  round_const = 1 << (shift_amount-1);
  for e = 0 to elements-1
    result = LSR(ElemQin[m>>1],e,esize) + round_const, shift_amount);
    ElemD[d],e,esize] = result<esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.216 VRSHRN (zero)

Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector.

This instruction is a pseudo-instruction of the VMOVN instruction. This means that:

- The encodings in this description are named to match the encodings of VMOVN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VMOVN gives the operational pseudocode for this instruction.

**A1**

VRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

and is never the preferred disassembly.

**T1**

VRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

and is never the preferred disassembly.

**Assembler symbols**

- `<c>`: For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional. For encoding T1: see Standard assembler syntax fields on page F1-7228.
- `<q>`: See Standard assembler syntax fields on page F1-7228.
- `<dt>`: Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:
  - I16 when size = 00
  - I32 when size = 01
  - I64 when size = 10
  - The encoding size = 11 is reserved.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**Operation for all encodings**

The description of **VMOVN** gives the operational pseudocode for this instruction.
VRSQRTE

Vector Reciprocal Square Root Estimate finds an approximate reciprocal square root of each element in a vector, and places the results in a second vector.

The operand and result elements are the same type, and can be floating-point numbers or unsigned integers.

For details of the operation performed by this instruction see Floating-point reciprocal estimate and step on page E1-7155.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

**64-bit SIMD vector variant**

Applies when Q == 0.

VRSQRTE{<c>}{<q>}.<dt> <Dd>, <Dm>

**128-bit SIMD vector variant**

Applies when Q == 1.

VRSQRTE{<c>}{<q>}.<dt> <Qd>, <Qm>

**Decode for all variants of this encoding**

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && (!HaveFP16Ext() || F == '0')) || size IN {'00', '11'} then UNDEFINED;
floating_point = (F == '1');
integer esize;
integer elements;
\[ \text{case size of } \]
\[ \text{when '01' esize = 16; elements = 4;} \]
\[ \text{when '10' esize = 32; elements = 2;} \]
d = UInt(D:Vd); m = UInt(M:Vm); \[ \text{regs = if Q == '0' then 1 else 2;} \]

**64-bit SIMD vector variant**

Applies when Q == 0.

VRSQRTE{<c>}{<q>}.<dt> <Dd>, <Dm>

**128-bit SIMD vector variant**

Applies when Q == 1.

VRSQRTE{<c>}{<q>}.<dt> <Qd>, <Qm>
Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' || (HaveFP16Ext() || F == '0')) || size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
floating_point = (F == '1');
integer esize;
integer elements;
case size of
  when '01' esize = 16; elements = 4;
  when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols

<*> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
   For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the vectors, encoded in the "F:size" field. It can have the following values:
   U32 when F = 0, size = 10
   F16 when F = 1, size = 01
   F32 when F = 1, size = 10
<qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see Floating-point reciprocal estimate and step on page E1-7155.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      if floating_point then
        Elem[D[d+r],e,esize] = FPRSqrtEstimate(Elem[D[m+r],e,esize], StandardFPSCRValue());
      else
        Elem[D[d+r],e,esize] = UnsignedRSqrtEstimate(Elem[D[m+r],e,esize]);

F6.1.218  VRSQRTS

Vector Reciprocal Square Root Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 3.0, divides these results by 2.0, and places the results into the elements of the destination vector.

The operand and result elements are floating-point numbers.

For details of the operation performed by this instruction see Floating-point reciprocal estimate and step on page E1-7155.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when Q == 0.

VRSQRTS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VRSQRTS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
integer esize;
in...
Decode for all variants of this encoding

if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' & !HaveFP16Ext() then UNDEFINED;
if sz == '1' & InITBlock() then UNPREDICTABLE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

CONSTRANGED UNPREDICTABLE behavior

If size == '01' & InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols
<>< For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
  For encoding T1: see Standard assembler syntax fields on page F1-7228.
<&> See Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:
  F32 when sz = 0
  F16 when sz = 1
<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see Floating-point reciprocal estimate and step on page E1-7155.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      Elem[D[d+r],e,esize] = FPRSqrtStep(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize]);
F6.1.219  VRSRA

Vector Rounding Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the rounded results into the destination vector. For truncated results, see VSRA.

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant
Applies when !(imm6 == 000xxx && L == 0) && Q == 0.

\[ VRSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm> \]

128-bit SIMD vector variant
Applies when !(imm6 == 000xxx && L == 0) && Q == 1.

\[ VRSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm> \]

Decode for all variants of this encoding

if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
  when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 -UInt(imm6);
  when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
  when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant
Applies when !(imm6 == 000xxx && L == 0) && Q == 0.

\[ VRSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm> \]
**128-bit SIMD vector variant**

Applies when !(imm6 == 000xxx && L == 0) && Q == 1.

\[
\text{VRSRA\{<c>\}{<q>\{<Qd>,\ <Qm>, \#<imm>}}
\]

**Decode for all variants of this encoding**

if (L:imm6) IN \{'0000xxx\}' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
  when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
  when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
  when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

**Notes for all encodings**

Related encodings: See *Advanced SIMD one register and modified immediate* on page F3-7322 for the T32 instruction set, or *Advanced SIMD one register and modified immediate* on page F4-7430 for the A32 instruction set.

**Assembler symbols**

- `<c>` For encoding A1: see *Standard assembler syntax fields* on page F1-7228. This encoding must be unconditional.
  
  For encoding T1: see *Standard assembler syntax fields* on page F1-7228.

- `<q>` See *Standard assembler syntax fields* on page F1-7228.

- `<type>` Is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:
  S when U = 0
  U when U = 1

- `<size>` Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:
  8 when L = 0, imm6<5:3> = 001
  16 when L = 0, imm6<5:3> = 01x
  32 when L = 0, imm6<5:3> = 1xx
  64 when L = 1, imm6<5:3> = xxx

- `<Qd>` Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>*2.

- `<Qm>` Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>*2.

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

- `<imm>` Is an immediate value, in the range 1 to `<size>`, encoded in the "imm6" field as `<size> - <imm>".
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = (Int(Elem[D+m+r],e,esize], unsigned) + round_const) >> shift_amount;
            Elem[D+d+r],e,esize] = Elem[D+d+r],e,esize] + result;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.220  VRSUBHN

Vector Rounding Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, takes the most significant half of each result, and places the final results in a doubleword vector. The results are rounded. For truncated results, see VSUBHN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 |7 6 5 4 |3 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | D !=11 | Vn |
|   |   |   |   |   |   |   |   | Vd |
|   |   |   |   |   | 0 | 1 | 1 | O |
|   |   |   |   |   | N | 0 | M | 0 |
|   |   |   |   |   |   |   |   | Vm |
```

**A1 variant**

VRSUBHN(<c>{<q>}.<dt> <Dd>, <Qn>, <Qm>)

**Decode for this encoding**

if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

T1

```
| 15 14 13 12|11 10 9 8 |7 6 5 4 |3 0 |15 12|11 10 9 8 |7 6 5 4 |3 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | D !=11 | Vn |
|   |   |   |   |   |   |   |   | Vd |
|   |   |   |   |   | 0 | 1 | 1 | N |
|   |   |   |   |   | 0 | M | 0 |
|   |   |   |   |   |   |   |   | Vm |
```

**T1 variant**

VRSUBHN(<c>{<q>}.<dt> <Dd>, <Qn>, <Qm>)

**Decode for this encoding**

if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

**Notes for all encodings**

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

**Assembler symbols**

<c>  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.
<dt> Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:
I16 when size = 00
I32 when size = 01
I64 when size = 10
<dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<qt> See Standard assembler syntax fields on page F1-7228.
<qt> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>.*2.
<qt> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>.*2.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  round_const = 1 << (esize-1);
  for e = 0 to elements-1
    result = (Elem[Qin[n>>1],e,2*esize] - Elem[Qin[m>>1],e,2*esize]) + round_const;
    Elem[D[d],e,esize] = result<2*esize-1:esize>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.221   VSDOT (by element)

Dot Product index form with signed integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

--- Note ---

ID_ISAR6.DP indicates whether this instruction is supported.

64-bit SIMD vector variant

Applies when Q == 0.

VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]

128-bit SIMD vector variant

Applies when Q == 1.

VSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;

64-bit SIMD vector variant

Applies when Q == 0.

VSDOT{<q>}.S8 <Dd>, <Dm>[<index>]
128-bit SIMD vector variant

Applies when Q == 1.

VSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;

Assembler symbols

&q; See Standard assembler syntax fields on page F1-7228.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.

<index> Is the element index in the range 0 to 1, encoded in the "M" field.

Operation for all encodings

bits(64) operand1;
bits(64) operand2 = D[m];
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
operand1 = D[n+r];
result = D[d+r];
integer element1, element2;
for e = 0 to 1
integer res = 0;
for i = 0 to 3
if signed then
    element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
    element2 = SInt(Elem[operand2, 4 * index + i, esize DIV 4]);
else
    element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
    element2 = UInt(Elem[operand2, 4 * index + i, esize DIV 4]);
res = res + element1 * element2;
Elem[result, e, esize] = Elem[result, e, esize] + res;
D[d+r] = result;
F6.1.222  VSDOT (vector)

Dot Product vector form with signed integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

--- Note ---
ID_ISAR6.DP indicates whether this instruction is supported.

A1

(FEAT_DotProd)

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>0 0</td>
<td>D</td>
<td>1 0</td>
<td>Vn</td>
<td>Vd</td>
<td>1 1 0 1</td>
<td>N</td>
</tr>
</tbody>
</table>
```

64-bit SIMD vector variant

Applies when \(Q == 0\).

\[
\text{VSDOT}(<q>).S8 \ <Dd>, <Dn>, <Dm>
\]

128-bit SIMD vector variant

Applies when \(Q == 1\).

\[
\text{VSDOT}(<q>).S8 \ <Qd>, <Qn>, <Qm>
\]

Decode for all variants of this encoding

```
if !HaveDOTPExt() then UNDEFINED;
if \(Q == '1' \land (Vd<0> == '1' \land Vn<0> == '1' \land Vm<0> == '1')\) then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
integer regs = if \(Q == '1'\) then 2 else 1;
```

T1

(FEAT_DotProd)

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>0 0</td>
<td>D</td>
<td>1 0</td>
</tr>
</tbody>
</table>
```

64-bit SIMD vector variant

Applies when \(Q == 0\).

\[
\text{VSDOT}(<q>).S8 \ <Dd>, <Dn>, <Dm>
\]
128-bit SIMD vector variant

Applies when \( Q = 1 \).

\[ \text{VSDOT}(\langle q \rangle).S8 \langle Qd \rangle, \langle Qn \rangle, \langle Qm \rangle \]

Decode for all variants of this encoding

\[
\begin{align*}
\text{if InITBlock()} \text{ then UNPREDICTABLE;} \\
\text{if !HaveDOTPExt()} \text{ then UNDEFINED;} \\
\text{if } Q = '1' \text{ && (\langle Vd<0> \rangle == '1' || \langle Vn<0> \rangle == '1' || \langle Vm<0> \rangle == '1') \text{ then UNDEFINED;} \\
\quad \text{boolean signed = U=='0';} \\
\quad \text{integer d = UInt(D:Vd);} \\
\quad \text{integer n = UInt(N:Vn);} \\
\quad \text{integer m = UInt(M:Vm);} \\
\quad \text{integer esize = 32;} \\
\quad \text{integer regs = if Q == '1' then 2 else 1;} \\
\end{align*}
\]

Assembler symbols

\( \langle q \rangle \)

See Standard assembler syntax fields on page F1-7228.

\( \langle Qd \rangle \)

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \( \langle Qd \rangle \times 2 \).

\( \langle Qn \rangle \)

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as \( \langle Qn \rangle \times 2 \).

\( \langle Qm \rangle \)

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as \( \langle Qm \rangle \times 2 \).

\( \langle Dd \rangle \)

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\( \langle Dn \rangle \)

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

\( \langle Dm \rangle \)

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

\[
\begin{align*}
\text{bits(64) operand1;} \\
\text{bits(64) operand2;} \\
\text{bits(64) result;} \\
\text{CheckAdvSIMDEnabled();} \\
\text{for } r = 0 \text{ to } \text{regs-1} \\
\quad \text{operand1} = D[n+r]; \\
\quad \text{operand2} = D[m+r]; \\
\quad \text{result} = D[d+r]; \\
\quad \text{integer element1, element2;} \\
\quad \text{for } e = 0 \text{ to } 1 \\
\quad \quad \text{integer res = 0;} \\
\quad \quad \text{for } i = 0 \text{ to } 3 \\
\quad \quad \quad \text{if signed then} \\
\quad \quad \quad \quad \text{element1} = \text{SInt(Elem[operand1, 4 + e + i, esize DIV 4]);} \\
\quad \quad \quad \quad \text{element2} = \text{SInt(Elem[operand2, 4 + e + i, esize DIV 4]);} \\
\quad \quad \quad \text{else} \\
\quad \quad \quad \quad \text{element1} = \text{UInt(Elem[operand1, 4 + e + i, esize DIV 4]);} \\
\quad \quad \quad \quad \text{element2} = \text{UInt(Elem[operand2, 4 + e + i, esize DIV 4]);} \\
\quad \quad \quad \quad \text{res = res + element1 \times element2;} \\
\quad \quad \quad \text{Elem[result, e, esize] = Elem[result, e, esize] + res;} \\
\quad \quad \text{D[d+r] = result;} \\
\end{align*}
\]
**F6.1.223 VSELEQ, VSELGE, VSELGT, VSELVS**

Floating-point conditional select allows the destination register to take the value in either one or the other source register according to the condition codes in the *The Application Program Status Register, APSR on page E1-7135.*

### A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 12 11 10  9  8  7  6  5  4  3  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 0 0 D cc Vn Vd 1 0 !=00 N 0 M 0 Vm</td>
<td>size</td>
</tr>
</tbody>
</table>

**Equal, half-precision scalar variant**

Applies when \( cc == 00 \) & \( size == 01 \).

VSELEQ.F16 <Sd>, <Sn>, <Sm> // Cannot be conditional

**Equal, single-precision scalar variant**

Applies when \( cc == 00 \) & \( size == 10 \).

VSELEQ.F32 <Sd>, <Sn>, <Sm> // Cannot be conditional

**Equal, double-precision scalar variant**

Applies when \( cc == 00 \) & \( size == 11 \).

VSELEQ.F64 <Dd>, <Dn>, <Dm> // Cannot be conditional

**Greater than or Equal, half-precision scalar variant**

Applies when \( cc == 10 \) & \( size == 01 \).

VSELGE.F16 <Sd>, <Sn>, <Sm> // Cannot be conditional

**Greater than or Equal, single-precision scalar variant**

Applies when \( cc == 10 \) & \( size == 10 \).

VSELGE.F32 <Sd>, <Sn>, <Sm> // Cannot be conditional

**Greater than or Equal, double-precision scalar variant**

Applies when \( cc == 10 \) & \( size == 11 \).

VSELGE.F64 <Dd>, <Dn>, <Dm> // Cannot be conditional

**Greater than, half-precision scalar variant**

Applies when \( cc == 11 \) & \( size == 01 \).

VSELGT.F16 <Sd>, <Sn>, <Sm> // Cannot be conditional

**Greater than, single-precision scalar variant**

Applies when \( cc == 11 \) & \( size == 10 \).

VSELGT.F32 <Sd>, <Sn>, <Sm> // Cannot be conditional

**Greater than, double-precision scalar variant**

Applies when \( cc == 11 \) & \( size == 11 \).

VSELGT.F64 <Dd>, <Dn>, <Dm> // Cannot be conditional
Unordered, half-precision scalar variant
Applies when \( cc = 01 \) \&\&\( size = 01\).
VSELV.S.F16 <Sd>, <Sn>, <Sm> // Cannot be conditional

Unordered, single-precision scalar variant
Applies when \( cc = 01 \) \&\&\( size = 10\).
VSELV.S.F32 <Sd>, <Sn>, <Sm> // Cannot be conditional

Unordered, double-precision scalar variant
Applies when \( cc = 01 \) \&\&\( size = 11\).
VSELV.S.F64 <Dd>, <Dn>, <Dm> // Cannot be conditional

Decode for all variants of this encoding

\[
\text{if } size = '00' \lor (size = '01' \&\& \text{HaveFP16Ext}) \text{ then UNDEFINED;}
\]
integer esize;
integer d;
integer n;
integer m;
case size of
when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
cond = cc:(cc<1> EOR cc<0>):'0';

T1

| 15 14 13 12|11 10 9 8|7 6 5 4|3 | 0 | 15|12|11 10 9 8|7 6 5 4|3 | 0 |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| 1 1 1 1 1 1 1 1 0 0 D cc Vn Vd 1 0 | 1 | 0 | 1 1 | 1 1 1 1 1 1 1 1 0 0 | D cc Vn Vd 1 0 |

Equal, half-precision scalar variant
Applies when \( cc = 00 \) \&\&\( size = 01\).
VSELEQ.S.F16 <Sd>, <Sn>, <Sm> // Not permitted in IT block

Equal, single-precision scalar variant
Applies when \( cc = 00 \) \&\&\( size = 10\).
VSELEQ.S.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block

Equal, double-precision scalar variant
Applies when \( cc = 00 \) \&\&\( size = 11\).
VSELEQ.S.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block

Greater than or Equal, half-precision scalar variant
Applies when \( cc = 10 \) \&\&\( size = 01\).
VSELGE.S.F16 <Sd>, <Sn>, <Sm> // Not permitted in IT block

Greater than or Equal, single-precision scalar variant
Applies when \( cc = 10 \) \&\&\( size = 10\).
VSELGE.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block

**Greater than or Equal, double-precision scalar variant**
Applies when cc == 10 && size == 11.

VSELGE.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block

**Greater than, half-precision scalar variant**
Applies when cc == 11 && size == 01.

VSELGT.F16 <Sd>, <Sn>, <Sm> // Not permitted in IT block

**Greater than, single-precision scalar variant**
Applies when cc == 11 && size == 10.

VSELGT.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block

**Greater than, double-precision scalar variant**
Applies when cc == 11 && size == 11.

VSELGT.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block

**Unordered, half-precision scalar variant**
Applies when cc == 01 && size == 01.

VSELVS.F16 <Sd>, <Sn>, <Sm> // Not permitted in IT block

**Unordered, single-precision scalar variant**
Applies when cc == 01 && size == 10.

VSELVS.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block

**Unordered, double-precision scalar variant**
Applies when cc == 01 && size == 11.

VSELVS.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block

**Decode for all variants of this encoding**

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
cond = cc:(cc<1> EOR cc<0>):'0';

**CONSTRAINED UNPREDICTABLE behavior**
If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
Assembler symbols

<\texttt{Dd}> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<\texttt{Dn}> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<\texttt{Dm}> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<\texttt{Sd}> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<\texttt{Sn}> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<\texttt{Sm}> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation for all encodings

EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = Zeros(16) : (if ConditionHolds(cond) then S[n] else S[m])<15:0>;
  when 32
    S[d] = if ConditionHolds(cond) then S[n] else S[m];
  when 64
    D[d] = if ConditionHolds(cond) then D[n] else D[m];
F6.1.224  VSHL (immediate)

Vector Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in the destination vector.

Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

128-bit SIMD vector variant

Applies when !(imm6 == 000xxx && L == 0) && Q == 1.

VSHL{<c>}{<q>}.I<size> {<Qd>,} <Qm>, #<imm>

Decode for all variants of this encoding

if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
  when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
  when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
  when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when !(imm6 == 000xxx && L == 0) && Q == 0.

VSHL{<c>}{<q>}.I<size> {<Dd>,} <Dm>, #<imm>

128-bit SIMD vector variant

Applies when !(imm6 == 000xxx && L == 0) && Q == 1.
VSHL{<c>}{<q>}.I<size> {<Qd>,} <Qm>, #<imm>

Decode for all variants of this encoding

if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
    d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Notes for all encodings

Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.

Assembler symbols

<> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
    For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<size> Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:
    8  when L = 0, imm6<5:3> = 001
    16 when L = 0, imm6<5:3> = 01x
    32 when L = 0, imm6<5:3> = 1xx
    64 when L = 1, imm6<5:3> = xxx

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<imm> Is an immediate value, in the range 0 to <size>-1, encoded in the "imm6" field.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = LSL(Elem[D[m+r],e,esize], shift_amount);
Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.225 VSHL (register)

Vector Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of
the corresponding element of a second vector, and places the results in the destination vector. If the shift value is
positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift.

For a rounding shift, see VRSHL.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the CPACR, NSACR, and HCPTER registers, and the Security state and PE mode in which
the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For
more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

### 64-bit SIMD vector variant

Applies when \( Q = 0 \).

\[
\text{VSHL\{<c>\}{<q>}.<dt> \{<Dd>,\} <Dm>, <Dn>}
\]

### 128-bit SIMD vector variant

Applies when \( Q = 1 \).

\[
\text{VSHL\{<c>\}{<q>}.<dt> \{<Qd>,\} <Qm>, <Qn>}
\]

**Decode for all variants of this encoding**

\[
\text{if } Q == '1' \&\& \quad (Vd<0> == '1' || \quad Vn<0> == '1' || \quad Vn<0> == '1') \text{ then UNDEFINED;}
\]

\[
\text{unsigned = (U == '1');}
\]

\[
\text{esize = 8 << UInt(size); elements = 64 DIV esize;}
\]

\[
\text{d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;}
\]

### 64-bit SIMD vector variant

Applies when \( Q = 0 \).

\[
\text{VSHL\{<c>\}{<q>}.<dt> \{<Dd>,\} <Dm>, <Dn>}
\]

### 128-bit SIMD vector variant

Applies when \( Q = 1 \).

\[
\text{VSHL\{<c>\}{<q>}.<dt> \{<Qd>,\} <Qm>, <Qn>}
\]
VSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>

**Decode for all variants of this encoding**

if Q == '1' & (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;

**Assembler symbols**

\(<c>\) For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields on page F1-7228*.
\(<q>\) See *Standard assembler syntax fields on page F1-7228*.
\(<dt>\) Is the data type for the elements of the vectors, encoded in the "U:size" field. It can have the following values:
- S8 when U = 0, size = 00
- S16 when U = 0, size = 01
- S32 when U = 0, size = 10
- S64 when U = 0, size = 11
- U8 when U = 1, size = 00
- U16 when U = 1, size = 01
- U32 when U = 1, size = 10
- U64 when U = 1, size = 11

\(<Qd>\) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<Qd>\)*2.
\(<Qm>\) Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as \(<Qm>\)*2.
\(<Qn>\) Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as \(<Qn>\)*2.
\(<Dd>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
\(<Dm>\) Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
\(<Dn>\) Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

**Operation for all encodings**

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      shift = SInt(Elem[D[n+r],e,esize]<7:0>);  
      if shift >= 0 then
        result = Int(Elem[D[m+r],e,esize], unsigned) << shift;
      else
        result = Int(Elem[D[m+r],e,esize], unsigned) >> -shift;
      Elem[D[d+r],e,esize] = result<esize-1:0>;
Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.226  VSHLL

Vector Shift Left Long takes each element in a doubleword vector, left shifts them by an immediate value, and places the results in a quadword vector.

The operand elements can be:
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.
- 8-bit, 16-bit, or 32-bit untyped integers, maximum shift only.

The result elements are twice the length of the operand elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

A1 variant

Applies when imm6 != 000xxx.

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

Decode for this encoding

if imm6 IN {'000xxx'} then SEE "Related encodings";
if Vd<> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
  when '001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
  when '01xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
  when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
if shift_amount == 0 then SEE "VMOVL";
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);

A2

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

A2 variant

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
**Decode for this encoding**

if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize; shift_amount = esize;
unsigned = FALSE; // Or TRUE without change of functionality
d = UInt(D:Vd); m = UInt(M:Vm);

### T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>U</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>imm6</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**T1 variant**

Applies when imm6 != 000xxx.

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

### Decode for this encoding

if imm6 IN {'000xxx'} then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
  when '001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
  when '01xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
  when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
if shift_amount == 0 then SEE "VMOVL";
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);

### T2

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>size</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**T2 variant**

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

### Decode for this encoding

if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize; shift_amount = esize;
unsigned = FALSE; // Or TRUE without change of functionality
d = UInt(D:Vd); m = UInt(M:Vm);

### Notes for all encodings

Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.
Assembler symbols

<<>> For encoding A1 and A2: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1 and T2: see Standard assembler syntax fields on page F1-7228.

<<q>> See Standard assembler syntax fields on page F1-7228.

<<type>> The data type for the elements of the operand. It must be one of:
- S Signed. In encoding T1/A1, encoded as U = 0.
- U Unsigned. In encoding T1/A1, encoded as U = 1.
- I Untyped integer, Available only in encoding T2/A2.

<<size>> The data size for the elements of the operand. The following table shows the permitted values and their encodings:

<table>
<thead>
<tr>
<th>&lt;&lt;size&gt;&gt;</th>
<th>Encoding T1/A1</th>
<th>Encoding T2/A2</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Encoded as imm6&lt;5:3&gt; = 0b001</td>
<td>Encoded as size = 0b00</td>
</tr>
<tr>
<td>16</td>
<td>Encoded as imm6&lt;5:4&gt; = 0b01</td>
<td>Encoded as size = 0b01</td>
</tr>
<tr>
<td>32</td>
<td>Encoded as imm6&lt;5&gt; = 1</td>
<td>Encoded as size = 0b10</td>
</tr>
</tbody>
</table>

<<Qd>> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <<Qd>*2.

<<Dm>> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<<imm>> The immediate value. <<imm>> must lie in the range 1 to <<size>>, and:
- If <<size>> == <<imm>>, the encoding is T2/A2.
- Otherwise, the encoding is T1/A1, and:
  - If <<size>> == 8, <<imm>> is encoded in imm6<2:0>.
  - If <<size>> == 16, <<imm>> is encoded in imm6<3:0>.
  - If <<size>> == 32, <<imm>> is encoded in imm6<4:0>.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  for e = 0 to elements-1
    result = Int(Elem[Dim[m],e,esize], unsigned) << shift_amount;
    Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.227   VSHR

Vector Shift Right takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see VRSHR.

The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when !(imm6 == 000xxx && L == 0) && Q == 0.

VSHR{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>

128-bit SIMD vector variant

Applies when !(imm6 == 000xxx && L == 0) && Q == 1.

VSHR{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>

Decode for all variants of this encoding

if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
  when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
  when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
  when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
  unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

64-bit SIMD vector variant

Applies when !(imm6 == 000xxx && L == 0) && Q == 0.

VSHR{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
128-bit SIMD vector variant

Applies when !(imm6 == 000xxx & L == 0) & Q == 1.

VSHR{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>

Decode for all variants of this encoding

if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' & (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Notes for all encodings

Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.

Assembler symbols

<> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<> See Standard assembler syntax fields on page F1-7228.

<type> Is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:
S when U = 0
U when U = 1

<size> Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:
8 when L = 0, imm6<5:3> = 001
16 when L = 0, imm6<5:3> = 01x
32 when L = 0, imm6<5:3> = 1xx
64 when L = 1, imm6<5:3> = xxx

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
<imm> Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.
Operation for all encodings

if ConditionPassed() then
   EncodingSpecificOperations();  CheckAdvSIMDEnabled();
   for r = 0 to regs-1
      for e = 0 to elements-1
         result = Int(Elem[D[m+r],e,esize], unsigned) >> shift_amount;
         Elem[D[d+r],e,esize] = result<esize-1:0>;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.228 VSHR (zero)

Vector Shift Right copies the contents of one SIMD register to another

This instruction is a pseudo-instruction of the VORR (register) instruction. This means that:

- The encodings in this description are named to match the encodings of VORR (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VORR (register) gives the operational pseudocode for this instruction.

A1

**64-bit SIMD vector variant**

Applies when Q == 0.

VSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>

and is never the preferred disassembly.

**128-bit SIMD vector variant**

Applies when Q == 1.

VSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>

and is never the preferred disassembly.

T1

**64-bit SIMD vector variant**

Applies when Q == 0.

VSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>

and is never the preferred disassembly.

**128-bit SIMD vector variant**

Applies when Q == 1.
VSHR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>

and is never the preferred disassembly.

**Assembler symbols**

<<c>> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

<<q>> See Standard assembler syntax fields on page F1-7228.

<<dt>> Is the data type for the elements of the vectors, and must be one of: S8, S16, S32, S64, U8, U16, U32 or U64.

<<Qd>> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<<Qm>> Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.

<<Dd>> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<<Dm>> Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

**Operation for all encodings**

The description of VORR (register) gives the operational pseudocode for this instruction.
F6.1.229  **VSHRN**

Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see VRSHRN.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

### A1

![A1 variant](image)

**A1 variant**

Applies when `imm6` != `000xxx`.

VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>

**Decode for this encoding**

if `imm6` IN `{000xxx}` then SEE "Related encodings";
if `Vm<0>` == '1' then UNDEFINED;
integer `esize`;
integer `elements`;
integer `shift_amount`;
case `imm6` of
  when `001xxx`  `esize` = 8;  `elements` = 8;  `shift_amount` = 16 - UInt(`imm6`);
  when `01xxxx`  `esize` = 16;  `elements` = 4;  `shift_amount` = 32 - UInt(`imm6`);
  when `1xxxxx`  `esize` = 32;  `elements` = 2;  `shift_amount` = 64 - UInt(`imm6`);

$d$ = UInt(`D:Vd`);  $m$ = UInt(`M:Vm`);

### T1

![T1 variant](image)

**T1 variant**

Applies when `imm6` != `000xxx`.

VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>

**Decode for this encoding**

if `imm6` IN `{000xxx}` then SEE "Related encodings";
if `Vm<0>` == '1' then UNDEFINED;
integer `esize`;
integer `elements`;
integer `shift_amount`;
case `imm6` of
  when `001xxx`  `esize` = 8;  `elements` = 8;  `shift_amount` = 16 - UInt(`imm6`);
  when `01xxxx`  `esize` = 16;  `elements` = 4;  `shift_amount` = 32 - UInt(`imm6`);
when 'lxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
  d = UInt(D:Vd);  m = UInt(M:Vm);

Notes for all encodings

Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.

Assembler symbols

<>  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
    For encoding T1: see Standard assembler syntax fields on page F1-7228.

<>  See Standard assembler syntax fields on page F1-7228.

<size>  Is the data size for the elements of the vectors, encoded in the "imm6<5:3>" field. It can have the following values:
        16  when imm6<5:3> = 001
        32  when imm6<5:3> = 01x
        64  when imm6<5:3> = 1xx

<DD>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm>  Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<imm>  Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for e = 0 to elements-1
    result = LSR(Elem[Qin[m>>1],e,2*esize], shift_amount);
    Elem[DD[e],e,esize] = result<esize-1:0>;
F6.1.230  VSHRN (zero)

Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector.

This instruction is a pseudo-instruction of the VMOVN instruction. This means that:

- The encodings in this description are named to match the encodings of VMOVN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VMOVN gives the operational pseudocode for this instruction.

**A1**

\[
\begin{array}{cccccccccccccccccccc}
|31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 12| 11| 10| 9| 8| 7| 6| 5| 4| 3| 0 | \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & D & 1 & 1 & size & 1 & 0 & Vd & 0 & 0 & 1 & 0 & 0 & 0 & M & 0 & Vm \\
\end{array}
\]

**A1 variant**

VSHRN\{<c>\}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

and is never the preferred disassembly.

**T1**

\[
\begin{array}{cccccccccccccccccccc}
|15| 14| 13| 12| 11| 10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0| 15| 12| 11| 10| 9| 8| 7| 6| 5| 4| 3| 0 | \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & D & 1 & 1 & size & 1 & 0 & Vd & 0 & 0 & 1 & 0 & 0 & 0 & M & 0 & Vm \\
\end{array}
\]

**T1 variant**

VSHRN\{<c>\}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

and is never the preferred disassembly.

**Assembler symbols**

\(<c>\)

For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.

\(<q>\)

See Standard assembler syntax fields on page F1-7228.

\(<dt>\)

Is the data type for the elements of the operand, encoded in the "size" field. It can have the following values:

- I16 when size = 00
- I32 when size = 01
- I64 when size = 10

The encoding size = 11 is reserved.
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**Operation for all encodings**

The description of VMOVN gives the operational pseudocode for this instruction.
F6.1.231 VSLI

Vector Shift Left and Insert takes each element in the operand vector, left shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTCR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

A1

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21</th>
<th>16 15 12 11 10 9 8</th>
<th>7 6 5 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1 1 D</td>
<td>imm6</td>
<td>Vd</td>
<td>0 1 0 1 L Q M 1</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector variant**

Applies when !(imm6 == 000xxx && L == 0) && Q == 0.

VSLI{<c>}{<q>}.<size> {<Dd>,} <Dm>, #<imm>

**128-bit SIMD vector variant**

Applies when !(imm6 == 000xxx && L == 0) && Q == 1.

VSLI{<c>}{<q>}.<size> {<Qd>,} <Qm>, #<imm>

**Decode for all variants of this encoding**

if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8</th>
<th>7 6 5</th>
<th>0</th>
<th>15 12 11 10 9 8</th>
<th>7 6 5 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
<td>1 1 1 1 1 1 1 D</td>
<td>imm6</td>
<td>Vd</td>
<td>0 1 0 1 L Q M 1</td>
<td>Vm</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector variant**

Applies when !(imm6 == 000xxx && L == 0) && Q == 0.

VSLI{<c>}{<q>}.<size> {<Dd>,} <Dm>, #<imm>

**128-bit SIMD vector variant**

Applies when !(imm6 == 000xxx && L == 0) && Q == 1.

VSLI{<c>}{<q>}.<size> {<Qd>,} <Qm>, #<imm>
Decode for all variants of this encoding

if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
  when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
  when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
  when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Notes for all encodings

Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.

Assembler symbols

<>  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
    For encoding T1: see Standard assembler syntax fields on page F1-7228.
<q>  See Standard assembler syntax fields on page F1-7228.
<size>  Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:
    8  when L = 0, imm6<5:3> = 001
    16 when L = 0, imm6<5:3> = 01x
    32 when L = 0, imm6<5:3> = 1xx
    64 when L = 1, imm6<5:3> = xxx
<Qd>  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>  Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>  Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
<imm>  Is an immediate value, in the range 0 to <size>-1, encoded in the "imm6" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  mask = LSL(Ones(esize), shift_amount);
  for r = 0 to regs-1
    for e = 0 to elements-1
      shifted_op = LSL(Elem[D[m+r],e,esize], shift_amount);
      Elem[D[d+r],e,esize] = (Elem[D[d+r],e,esize] AND NOT(mask)) OR shifted_op;
Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.232 VSMMLA

The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of signed 8-bit integer values held in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an optional instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

A1

(FEAT_AA32I8MM)

| 31 30 29 28|27 26 25 24|23 22 21 20|19 |16|15 |12|11 10 9 8 |7 6 5 4 |3 |0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|1 1 1 1|1 1|0|0|D|1 0|Vn|Vd|1|1|0|0|N|1|M|0|Vm|

A1 variant

VSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>

Decode for this encoding

if !HaveAArch32Int8MatMulExt() then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
case B:U of
  when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' op1_unsigned = TRUE; op2_unsigned = TRUE;
  when '10' op1_unsigned = TRUE; op2_unsigned = FALSE;
  when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);

T1

(FEAT_AA32I8MM)

| 15 14 13|12|11 10 9 8 |7 6 5 4 |3 |0 |15 |12|11 10 9 8 |7 6 5 4 |3 |0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|1 1 1 1|1 1|0|0|D|1 0|Vn|Vd|1|1|0|0|N|1|M|0|Vm|

T1 variant

VSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>

Decode for this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
case B:U of
  when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' op1_unsigned = TRUE; op2_unsigned = TRUE;
  when '10' op1_unsigned = TRUE; op2_unsigned = FALSE;
  when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);

**Assembler symbols**

<q> See *Standard assembler syntax fields* on page F1-7228.

<Qd> Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**Operation for all encodings**

CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
F6.1.233  VSQRT

Square Root calculates the square root of the value in a floating-point register and writes the result to another floating-point register.

Depending on settings in the CPACR, NSACR, HCPTF, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

\[
\begin{array}{ccccccccccccccc}
\hline
\text{l=1111} & 1 & 1 & 1 & 0 & 1 & D & 1 & 1 & 0 & 0 & 1 & Vd & 1 & 0 & \text{size} & 1 & 1 & M & 0 & Vm
\end{array}
\]

cond

**Half-precision scalar variant**

Applies when size == 01.

\[
\text{VSQRT}^{\{<c>\}{<q>}}.\text{F16} <Sd>, <Sm>
\]

**Single-precision scalar variant**

Applies when size == 10.

\[
\text{VSQRT}^{\{<c>\}{<q>}}.\text{F32} <Sd>, <Sm>
\]

**Double-precision scalar variant**

Applies when size == 11.

\[
\text{VSQRT}^{\{<c>\}{<q>}}.\text{F64} <Dd>, <Dm>
\]

**Decode for all variants of this encoding**

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

\[
\begin{array}{cccccccccccccccc}
| 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 |
\hline
1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & D & 1 & 1 & 0 & 0 & 0 & 1 & Vd & 1 & 0 & \text{size} & 1 & 1 & M & 0 & Vm
\end{array}
\]
Half-precision scalar variant
Applies when size == 01.
\texttt{VSQRT\langle c\rangle\langle q\rangle.F16 <Sd>, <Sm>}

Single-precision scalar variant
Applies when size == 10.
\texttt{VSQRT\langle c\rangle\langle q\rangle.F32 <Sd>, <Sm>}

Double-precision scalar variant
Applies when size == 11.
\texttt{VSQRT\langle c\rangle\langle q\rangle.F64 <Dd>, <Dm>}

Decode for all variants of this encoding
\begin{verbatim}
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
integer esize;
integer d;
integer m;
\textbf{case size of}
\textbf{when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);}.
\textbf{when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);}.
\textbf{when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);}.
\end{verbatim}

\textbf{CONSTRAINED UNPREDICTABLE behavior}
If size == '01' && InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler symbols
\begin{itemize}
\item \texttt{<>} See \textit{Standard assembler syntax fields on page F1-7228}.
\item \texttt{<q>} See \textit{Standard assembler syntax fields on page F1-7228}.
\item \texttt{<Sd>} Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
\item \texttt{<Sm>} Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
\item \texttt{<Dd>} Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
\item \texttt{<Dm>} Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
\end{itemize}

Operation for all encodings
\begin{verbatim}
if ConditionPassed() then
\texttt{EncodingSpecificOperations(); CheckVFPEnabled(TRUE);}
case esize of
\texttt{when 16 S[d] = Zeros(16) : FPSqrt(S[m]<15:0>, FPSCR[]);}
\texttt{when 32 S[d] = FPSqrt(S[m], FPSCR[]);}
\texttt{when 64 D[d] = FPSqrt(D[m], FPSCR[]);}
\end{verbatim}
F6.1.234 VSRA

Vector Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the truncated results into the destination vector. For rounded results, see VRSRA.

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

![VSRA Instruction Format](image)

### 64-bit SIMD vector variant

Applies when !(imm6 == 000xxx && L == 0) && Q == 0.

VSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>

### 128-bit SIMD vector variant

Applies when !(imm6 == 000xxx && L == 0) && Q == 1.

VSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>

#### Decode for all variants of this encoding

if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
unsigned = (U == '1'); d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

![T1 Instruction Format](image)

### 64-bit SIMD vector variant

Applies when !(imm6 == 000xxx && L == 0) && Q == 0.

VSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
128-bit SIMD vector variant

Applies when !(imm6 == 000xxx && L == 0) && Q == 1.

VSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>

Decode for all variants of this encoding

```plaintext
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
  case L:imm6 of
  when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
  when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
  when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
  when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
  endcase;
  unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Notes for all encodings

Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.

Assembler symbols

- `<c>`: For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
  
  For encoding T1: see Standard assembler syntax fields on page F1-7228.

- `<q>`: See Standard assembler syntax fields on page F1-7228.

- `<type>`: Is the data type for the elements of the vectors, encoded in the "U" field. It can have the following values:
  
  - S when U = 0
  - U when U = 1

- `<size>`: Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:
  
  - 8 when L = 0, imm6<5:3> = 001
  - 16 when L = 0, imm6<5:3> = 01x
  - 32 when L = 0, imm6<5:3> = 1xx
  - 64 when L = 1, imm6<5:3> = xxx

- `<Qd>`: Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as `<Qd>"*2.

- `<Qm>`: Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as `<Qm>"*2.

- `<Dd>`: Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- `<Dm>`: Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

- `<imm>`: Is an immediate value, in the range 1 to `<size>`, encoded in the "imm6" field as `<size> - <imm>".
Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      result = Int(Elem[D[m+r],e,esize], unsigned) >> shift_amount;
      Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + result;

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
Vector Shift Right and Insert takes each element in the operand vector, right shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the right of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

**64-bit SIMD vector variant**

Applies when !(imm6 == 000xxx && L == 0) && Q == 0.

\[ \text{VSRI}\{<c>\}{<q>}.<\text{size}> \{<Dd>,\} <Dm>, #<imm> \]

**128-bit SIMD vector variant**

Applies when !(imm6 == 000xxx && L == 0) && Q == 1.

\[ \text{VSRI}\{<c>\}{<q>}.<\text{size}> \{<Qd>,\} <Qm>, #<imm> \]

### Decode for all variants of this encoding

if (L:imm6) IN {‘0000xxx’} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '001xxxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
  when '01xxxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
  when '1xxxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

### 64-bit SIMD vector variant

Applies when !(imm6 == 000xxx && L == 0) && Q == 0.

\[ \text{VSRI}\{<c>\}{<q>}.<\text{size}> \{<Dd>,\} <Dm>, #<imm> \]

### 128-bit SIMD vector variant

Applies when !(imm6 == 000xxx && L == 0) && Q == 1.

\[ \text{VSRI}\{<c>\}{<q>}.<\text{size}> \{<Qd>,\} <Qm>, #<imm> \]
Decode for all variants of this encoding

if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
  when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
  when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
  when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

Notes for all encodings

Related encodings: See Advanced SIMD one register and modified immediate on page F3-7322 for the T32 instruction set, or Advanced SIMD one register and modified immediate on page F4-7430 for the A32 instruction set.

Assembler symbols

<q> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
   For encoding T1: see Standard assembler syntax fields on page F1-7228.
<q> See Standard assembler syntax fields on page F1-7228.
<size> Is the data size for the elements of the vectors, encoded in the "L:imm6<5:3>" field. It can have the following values:
  8 when L = 0, imm6<5:3> = 001
  16 when L = 0, imm6<5:3> = 01x
  32 when L = 0, imm6<5:3> = 1xx
  64 when L = 1, imm6<5:3> = xxx
<qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <qd>*2.
<qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <qm>*2.
<dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
<img> Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <img>.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  mask = LSR(Ones(esize), shift_amount);
  for r = 0 to regs-1
    for e = 0 to elements-1
      shifted_op = LSR(Elem[D[m+r],e,esize], shift_amount);
      Elem[D[d+r],e,esize] = (Elem[D[d+r],e,esize] AND NOT(mask)) OR shifted_op;
Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.236 VST1 (single element from one lane)

Store single element from one lane of one register stores one element to memory from one element of a register. For
details of the addressing mode, see *The Advanced SIMD addressing mode on page F1-7249.*

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which
the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For
more information, see *Enabling Advanced SIMD and floating-point support on page G1-8996.*

A1

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20 19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
</tr>
</tbody>
</table>

**Offset variant**

Applies when Rm == 1111.

VST1{<c>}{<q>}.<size> <list>, [Rn]{:<align>}

**Post-indexed variant**

Applies when Rm == 1101.

VST1{<c>}{<q>}.<size> <list>, [Rn]{:<align>}!

**Post-indexed variant**

Applies when Rm != 11x1.

VST1{<c>}{<q>}.<size> <list>, [Rn]{:<align>}, Rm

**Decode for all variants of this encoding**

if size == '11' then UNDEFINED;
if index_align<>0 then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); alignment = 1;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 & m != 13);
if n == 15 then UNPREDICTABLE;

A2

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20 19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
</tr>
</tbody>
</table>

**Offset variant**

Applies when Rm == 1111.

VST1{<c>}{<q>}.<size> <list>, [Rn]{:<align>}

**Post-indexed variant**

Applies when Rm == 1101.

VST1{<c>}{<q>}.<size> <list>, [Rn]{:<align>}!
Post-indexed variant

Applies when \( Rm \neq 11x1 \).

\[
\text{VST1}\{<c>\}{<q>}.<size> \ <\text{list}>, \ [<Rn>{:<align>}], \ <Rm>
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{if size} &= '11' \text{ then UNDEFINED;} \\
\text{if index\_align} <2> &= '0' \text{ then UNDEFINED;} \\
ebytes &= 2; \text{ index} = \text{UINT(index\_align<3:2>);} \\
alignment &= \text{if index\_align<0> == '0' then 1 else 2;} \\
d &= \text{UINT(D:Vd)}; \ n = \text{UINT(Rn)}; \ m = \text{UINT(Rm)}; \\
wback &= (m \neq 15); \ \text{register\_index} = (m \neq 15 \&\& m \neq 13); \\
\text{if n} &= 15 \text{ then UNPREDICTABLE;}
\end{align*}
\]

A3

\[
\begin{array}{cccccccccccccccccccccccccccc}
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 4 | 3 | 0 | \\
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>index_align</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\end{array}
\]

Offset variant

Applies when \( Rm \equiv 1111 \).

\[
\text{VST1}\{<c>\}{<q>}.<size> \ <\text{list}>, \ [<Rn>{:<align>}]!
\]

Post-indexed variant

Applies when \( Rm \neq 1101 \).

\[
\text{VST1}\{<c>\}{<q>}.<size> \ <\text{list}>, \ [<Rn>{:<align}>]!
\]

Post-indexed variant

Applies when \( Rm \neq 11x1 \).

\[
\text{VST1}\{<c>\}{<q>}.<size> \ <\text{list}>, \ [<Rn>{:<align}>], \ <Rm>
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{if size} &= '11' \text{ then UNDEFINED;} \\
\text{if index\_align}<2> &= '0' \text{ then UNDEFINED;} \\
ebytes &= 4; \text{ index} = \text{UINT(index\_align<3>);} \\
alignment &= \text{if index\_align<1:0> == '00' then 1 else 4;} \\
d &= \text{UINT(D:Vd)}; \ n = \text{UINT(Rn)}; \ m = \text{UINT(Rm)}; \\
wback &= (m \neq 15); \ \text{register\_index} = (m \neq 15 \&\& m \neq 13); \\
\text{if n} &= 15 \text{ then UNPREDICTABLE;}
\end{align*}
\]

T1

\[
\begin{array}{cccccccccccccccccccccccccccc}
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 0 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 4 | 3 | 0 | \\
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>index_align</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\end{array}
\]

Offset variant

Applies when \( Rm \equiv 1111 \).
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

**Post-indexed variant**
Applies when Rm == 1101.
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

**Post-indexed variant**
Applies when Rm != 11x1.
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

**Decode for all variants of this encoding**

if size == '11' then UNDEFINED;
if index_align[0] != '0' then UNDEFINED;
ebytes = 1; index = UInt(index_align[3:1]); alignment = 1;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;

T2

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 0 | 12 11 10 9 8 7 4 3 0 |
|-------------|--------|------|----|--------|--------|
| 1 1 1 1 0 0 1 1 D 0 0 Rn | Vd 0 1 0 0 index_align | Rm |

**Offset variant**
Applies when Rm == 1111.
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

**Post-indexed variant**
Applies when Rm == 1101.
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

**Post-indexed variant**
Applies when Rm != 11x1.
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

**Decode for all variants of this encoding**

if size == '11' then UNDEFINED;
if index_align[0] != '0' then UNDEFINED;
ebytes = 2; index = UInt(index_align[3:2]); alignment = if index_align[0] == '0' then 1 else 2;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
**T3**

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4</th>
<th>3 0</th>
<th>12 11 10 9 8 7 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1</td>
<td>D 0 0</td>
<td>Rn</td>
<td>Vd</td>
</tr>
</tbody>
</table>

**Offset variant**

Applies when Rm == 1111.

\[\text{VST1\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align>}]}\]

**Post-indexed variant**

Applies when Rm == 1101.

\[\text{VST1\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align}>]}!\]

**Post-indexed variant**

Applies when Rm != 11x1.

\[\text{VST1\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align}>]}, <Rm}\]

**Decode for all variants of this encoding**

if size == '11' then UNDEFINED;
if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;

ebytes = 4;
index = UInt(index_align<3>);
alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

**Assembler symbols**

\(<c>\)

For encoding A1, A2 and A3: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1, T2 and T3: see Standard assembler syntax fields on page F1-7228.

\(<q>\)

See Standard assembler syntax fields on page F1-7228.

\(<size>\)

Is the data size, encoded in the "size" field. It can have the following values:

- 8 when size = 00
- 16 when size = 01
- 32 when size = 10

\(<list>\)

Is a list containing the single 64-bit name of the SIMD&FP register holding the element.

The list must be \{<Dd>[<index>]\}.

The register <Dd> is encoded in the "D:Vd" field.

The permitted values and encoding of <index> depend on <size>:

- <size> == 8<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.
<size> == 16<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.
<size> == 32<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn>
Is the general-purpose base register, encoded in the "Rn" field.

<align>
When <size> == 8, <align> must be omitted, otherwise it is the optional alignment. Whenever <align> is omitted, the standard alignment is used, see Unaligned data access on page F2-7193, and the encoding depends on <size>:
<size> == 8Encoded in the "index_align<0>" field as 0.
<size> == 16Encoded in the "index_align<1:0>" field as 0b00.
<size> == 32Encoded in the "index_align<2:0>" field as 0b000.
Whenever <align> is present, the permitted values and encoding depend on <size>:
<size> == 16<align> is 16, meaning 16-bit alignment, encoded in the "index_align<1:0>" field as 0b01.
<size> == 32<align> is 32, meaning 32-bit alignment, encoded in the "index_align<2:0>" field as 0b11.
: is the preferred separator before the <align> value, but the alignment can be specified as 0<align>, see The Advanced SIMD addressing mode on page F1-7249.

<Rm>
Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see The Advanced SIMD addressing mode on page F1-7249.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  address = R[n]; iswrite = TRUE;
  MemU[address,ebytes] = Elem[D[d],index];
  if wback then
    if register_index then
      R[n] = R[n] + R[m];
    else
      R[n] = R[n] + ebytes;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.237  VST1 (multiple single elements)

Store multiple single elements from one, two, three, or four registers stores elements to memory from one, two, three, or four registers, without interleaving. Every element of each register is stored. For details of the addressing mode, see *The Advanced SIMD addressing mode* on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

A1

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4 | 3 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | D | 0 | 0 | Rn | Vd | 0 | 1 | 1 | size | align | Rm |
```

*Offset variant*

Applies when Rm == 1111.

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}

*Post-indexed variant*

Applies when Rm == 1101.

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}])!

*Post-indexed variant*

Applies when Rm != 11x1.

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]], <Rm>

*Decode for all variants of this encoding*

\[
\text{regs} = 1; \quad \text{if align} < 1 > '1' \text{ then UNDEFINED}; \\
\text{alignment} = \text{if align == '00' then 1 else } 4 \ll \text{UInt}(\text{align}); \\
\text{ebbytes} = \text{if align == '00' then 1 else } \text{UInt}(\text{size}); \quad \text{elements} = 8 \div \text{ebbytes}; \\
\text{d} = \text{UInt}(D:\text{Vd}); \quad \text{n} = \text{UInt}(\text{Rn}); \quad \text{m} = \text{UInt}(\text{Rm}); \\
\text{wback} = (\text{m} != 15); \quad \text{register index} = (\text{m} != 15 \&\& \text{m} != 13); \\
\text{if n == 15 then UNPREDICTABLE};
\]

A2

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4 | 3 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | D | 0 | 0 | Rn | Vd | 1 | 0 | 1 | size | align | Rm |
```

*Offset variant*

Applies when Rm == 1111.

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}

*Post-indexed variant*

Applies when Rm == 1101.

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}])!
**Post-indexed variant**

Applies when \( Rm \neq 11x1 \).

\[
\text{VST1}\{\langle c \rangle\}{\langle q \rangle}.<\text{size}> \ <\text{list}>\!, \ [\langle \text{Rn} \rangle\{:<\text{align}>\}], \ <\text{Rm}>
\]

**Decode for all variants of this encoding**

\[
\text{regs} = 2; \ \text{if align} = '11' \ \text{then UNDEFINED};
\]

\[
\text{alignment} = \text{if align} = '00' \ \text{then 1 else} \ 4 \ \ll\ \text{UInt(align)};
\]

\[
\text{ebytes} = 1 \ \ll\ \text{UInt(size)}; \ \text{elements} = 8 \ \div\ \text{ebytes};
\]

\[
d = \text{UInt(D:Vd)}; \ \text{n} = \text{UInt(Rn)}; \ \text{m} = \text{UInt(Rm)};
\]

\[
\text{wback} = (m \neq 15); \ \text{register_index} = (m \neq 15 \ \&\& \ m \neq 13);
\]

\[
\text{if n} = 15 \ \|\ | \text{d-regs} > 32 \ \text{then UNPREDICTABLE};
\]

**CONSTRANGED UNPREDICTABLE behavior**

If \( d+\text{regs} > 32 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

**A3**

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 16 15 12 11 10 9 8 7 6 5 4 3 0 |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| 1 1 1 1 0 1 0 0 0 | D 0 0 | Rn | Vd | 0 1 1 0 | size | align | Rm |

**Offset variant**

Applies when \( Rm = 1111 \).

\[
\text{VST1}\{\langle c \rangle\}{\langle q \rangle}.<\text{size}> \ <\text{list}>\!, \ [\langle \text{Rn} \rangle\{:<\text{align}>\}]!
\]

**Post-indexed variant**

Applies when \( Rm = 1101 \).

\[
\text{VST1}\{\langle c \rangle\}{\langle q \rangle}.<\text{size}> \ <\text{list}>\!, \ [\langle \text{Rn} \rangle\{:<\text{align}>\}]
\]

**Post-indexed variant**

Applies when \( Rm \neq 11x1 \).

\[
\text{VST1}\{\langle c \rangle\}{\langle q \rangle}.<\text{size}> \ <\text{list}>\!, \ [\langle \text{Rn} \rangle\{:<\text{align}>\}], \ <\text{Rm}>
\]

**Decode for all variants of this encoding**

\[
\text{regs} = 3; \ \text{if align} < 1 \ \text{then UNDEFINED};
\]

\[
\text{alignment} = \text{if align} = '00' \ \text{then 1 else} \ 4 \ \ll\ \text{UInt(align)};
\]

\[
\text{ebytes} = 1 \ \ll\ \text{UInt(size)}; \ \text{elements} = 8 \ \div\ \text{ebytes};
\]

\[
d = \text{UInt(D:Vd)}; \ \text{n} = \text{UInt(Rn)}; \ \text{m} = \text{UInt(Rm)};
\]

\[
\text{wback} = (m \neq 15); \ \text{register_index} = (m \neq 15 \ \&\& \ m \neq 13);
\]

\[
\text{if n} = 15 \ \|\ | \text{d-regs} > 32 \ \text{then UNPREDICTABLE};
\]
**CONSTRAINED UNPREDICTABLE behavior**

If \( d + \text{regs} > 32 \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The memory locations specified by the instruction and the number of registers specified by the instruction become **UNKNOWN**. If the instruction specifies writeback, then that register becomes **UNKNOWN**. This behavior does not affect any other memory locations.

### Offset variant

Applies when \( Rm == 1111 \).

\[ \text{VST1} \{<c>\}{<q>}.<size> <list>, [<Rn}\{:<align>\}] \]

### Post-indexed variant

Applies when \( Rm == 1101 \).

\[ \text{VST1} \{<c>\}{<q>}.<size> <list>, [<Rn}\{:<align>\}]! \]

### Post-indexed variant

Applies when \( Rm != 11x1 \).

\[ \text{VST1} \{<c>\}{<q>}.<size> <list>, [<Rn}\{:<align>\}], <Rm> \]

### Decode for all variants of this encoding

\[
\begin{align*}
\text{regs} = 4; \\
\text{alignment} &= \text{if align} == '00' \text{ then 1 else 4} \ll \text{UInt}(\text{align}); \\
\text{ebytes} &= 1 \ll \text{UInt}(\text{size}); \quad \text{elements} = 8 \ll \text{ebytes}; \\
\text{d} &= \text{UInt}(D:Vd); \quad n = \text{UInt}(Rn); \quad m = \text{UInt}(Rm); \\
\text{wback} &= (m != 15); \quad \text{register_index} = (m != 15 && m != 13); \\
\text{if n} &= 15 || \text{d} + \text{regs} > 32 \text{ then UNPREDICTABLE; }
\end{align*}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( d + \text{regs} > 32 \), then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The memory locations specified by the instruction and the number of registers specified by the instruction become **UNKNOWN**. If the instruction specifies writeback, then that register becomes **UNKNOWN**. This behavior does not affect any other memory locations.
T1

```

Offset variant
Applies when Rm == 1111.
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

```

Post-indexed variant
Applies when Rm == 1101.
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

```

Post-indexed variant
Applies when Rm != 11x1.
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```

Decode for all variants of this encoding
regs = 1;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T2

```

Offset variant
Applies when Rm == 1111.
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

```

Post-indexed variant
Applies when Rm == 1101.
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

```

Post-indexed variant
Applies when Rm != 11x1.
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```

Decode for all variants of this encoding
regs = 2;  if align<1> == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
```

d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;

**CONSTRAINED UNPREDICATABLE behavior**

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

**T3**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>0</td>
</tr>
</tbody>
</table>

**Decode for all variants of this encoding**

regs = 3;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(alignment);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;

**CONSTRANED UNPREDICATABLE behavior**

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.
Offset variant
Applies when \( Rm == 1111 \).
\[
\text{VST1}{<c>}{<q>}.<size> <\text{list}>, [\langle\text{Rn}\rangle:\langle\text{align}\rangle]}
\]

Post-indexed variant
Applies when \( Rm == 1101 \).
\[
\text{VST1}{<c>}{<q>}.<size> <\text{list}>, [\langle\text{Rn}\rangle:\langle\text{align}\rangle]!}
\]

Post-indexed variant
Applies when \( Rm != 11x1 \).
\[
\text{VST1}{<c>}{<q>}.<size> <\text{list}>, [\langle\text{Rn}\rangle:\langle\text{align}\rangle]}, \langle\text{Rm}\rangle
\]

Decode for all variants of this encoding
\[
\begin{align*}
\text{regs} &= 4; \\
\text{alignment} &= \text{if align} == '00' \text{ then } 1 \text{ else } 4 \ll \text{UInt}(\text{align}); \\
\text{ebytes} &= 1 \ll \text{UInt}(\text{size}); \text{ elements} = 8 \times \text{DIV ebytes}; \\
\text{d} &= \text{UInt}(D:Vd); \text{ n} = \text{UInt}(\text{Rn}); \text{ m} = \text{UInt}(\text{Rm}); \\
\text{wback} &= (m != 15); \text{ register_index} = (m != 15 \&\& m != 15); \\
\text{if n == 15 || d+regs > 32 then UNPREDICTABLE;}
\end{align*}
\]

CONSTRANGED UNPREDICTABLE behavior
If \( d+\text{regs} > 32 \), then one of the following behaviors must occur:
\begin{itemize}
  \item The instruction is UNDEFINED.
  \item The instruction executes as NOP.
  \item The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.
\end{itemize}

Notes for all encodings
For more information about the CONSTRANGED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VST1 (multiple single elements) on page K1-11579.

Related encodings: See Advanced SIMD element or structure load/store on page F3-7350 for the T32 instruction set, or Advanced SIMD element or structure load/store on page F4-7434 for the A32 instruction set.

Assembler symbols
\[
\begin{align*}
<\text{c}> & \quad \text{For encoding A1, A2, A3 and A4: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.} \\
& \quad \text{For encoding T1, T2, T3 and T4: see Standard assembler syntax fields on page F1-7228.} \\
<\text{q}> & \quad \text{See Standard assembler syntax fields on page F1-7228.}
\end{align*}
\]
<size> Is the data size, encoded in the "size" field. It can have the following values:
8 when size = 00
16 when size = 01
32 when size = 10
64 when size = 11

=list> Is a list containing the 64-bit names of the SIMD&FP registers.
The list must be one of:
{ <Dd> } Single register. Selects the A1 and T1 encodings of the instruction.
{ <Dd>, <Dd+1> } Two single-spaced registers. Selects the A2 and T2 encodings of the instruction.
{ <Dd>, <Dd+1>, <Dd+2> } Three single-spaced registers. Selects the A3 and T3 encodings of the instruction.
{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> } Four single-spaced registers. Selects the A4 and T4 encodings of the instruction.
The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see Unaligned data access on page E2-7193, and is encoded in the "align" field as 0000.
Whenever <align> is present, the permitted values are:
64 64-bit alignment, encoded in the "align" field as 0b01.
128 128-bit alignment, encoded in the "align" field as 0b10. Available only if <list> contains two or four registers.
256 256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.
: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see The Advanced SIMD addressing mode on page F1-7249.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.
For more information about <Rn>, !, and <Rm>, see The Advanced SIMD addressing mode on page F1-7249.

Operation for all encodings
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to regs-1
        for e = 0 to elements-1
            if ebytes != 8 then
                MemU[address,ebytes] = Elem[D[d+r],e];
            else
                - = AArch32.CheckAlignment(address, ebytes, AccType_NORMAL, iswrite);
                bits(64) data = Elem[0[d+r],e];
                MemU[address,4] = if BigEndian(AccType_NORMAL) then data<63:32> else data<31:0>;
                MemU[address+4,4] = if BigEndian(AccType_NORMAL) then data<31:0> else data<63:32>;
                address = address + ebytes;
            if wback then
                if register_index then
                    R[n] = R[n] + R[m];
                else
                    R[n] = R[n] + 8*regs;
            else
                MemU[address,e] = Elem[D[d+r],e];
        - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
        if wback then
            if register_index then
                R[n] = R[n] + R[m];
            else
                R[n] = R[n] + 8*regs;
        else
            MemU[address,e] = Elem[D[d+r],e];
    MemU[address,4] = if BigEndian(AccType_NORMAL) then data<63:32> else data<31:0>;
    MemU[address+4,4] = if BigEndian(AccType_NORMAL) then data<31:0> else data<63:32>;
    address = address + ebytes;
if wback then
    if register_index then
        R[n] = R[n] + R[m];
    else
        R[n] = R[n] + 8*regs;
Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.238  VST2 (single 2-element structure from one lane)

Store single 2-element structure from one lane of two registers stores one 2-element structure to memory from corresponding elements of two registers. For details of the addressing mode, see The Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}

Offset variant
Applies when \( \text{Rm} = 1111 \).

\[
\text{VST2}\{\langle c\rangle\}{\langle q\rangle}.\langle size\rangle\ \langle list\rangle, \ [\langle Rn\rangle\{\langle align\rangle\}] \]

Post-indexed variant
Applies when \( \text{Rm} = 1101 \).

\[
\text{VST2}\{\langle c\rangle\}{\langle q\rangle}.\langle size\rangle\ \langle list\rangle, \ [\langle Rn\rangle\{\langle align\rangle\}]! \]

Post-indexed variant
Applies when \( \text{Rm} != 11x1 \).

\[
\text{VST2}\{\langle c\rangle\}{\langle q\rangle}.\langle size\rangle\ \langle list\rangle, \ [\langle Rn\rangle\{\langle align\rangle\}], \ \langle Rm\rangle \]

Decode for all variants of this encoding

\[
\text{if size} == '11' \text{ then UNDEFINED};
\]
\[
ebytes = 1; \ \text{index} = \text{UInt}(\text{index}_\text{align}<3:1>); \ \text{inc} = 1;
\]
\[
\text{alignment} = \text{if} \ \text{index}_\text{align}<0> == '0' \text{ then 1 else 2};
\]
\[
d = \text{UInt}(D:Vd); \ \text{d2} = d + \text{inc}; \ \text{n} = \text{UInt}(\text{Rn}); \ \text{m} = \text{UInt}(\text{Rm});
\]
\[
\text{wback} = (\text{m} != 15); \ \text{register_index} = (\text{m} != 15 \ & \ & \text{m} != 13);
\]
\[
\text{if} \ \text{n} == 15 \ & \ & \text{d2} > 31 \text{ then UNPREDICTABLE};
\]

CONSTRANED UNPREDICTABLE behavior

If \( \text{d2} > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2
Offset variant
Applies when \( Rm = 1111 \).
\[
\text{VST2}\{<c>\}{<q>}.<\text{size}> \text{ <list>, } [<\text{Rn}>{:<\text{align}>}]
\]

Post-indexed variant
Applies when \( Rm = 1101 \).
\[
\text{VST2}\{<c>\}{<q>}.<\text{size}> \text{ <list>, } [<\text{Rn}>{:<\text{align}>}]!
\]

Post-indexed variant
Applies when \( Rm \neq 11x1 \).
\[
\text{VST2}\{<c>\}{<q>}.<\text{size}> \text{ <list>, } [<\text{Rn}>{:<\text{align}>}], <\text{Rm}>
\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{if size} & = \text{'11'} \text{ then UNDEFINED;} \\
ebytes & = 2; \text{ index} = \text{UInt}(\text{index\_align}<3:2>); \\
inc & = \text{if index\_align}<1> = '0' \text{ then 1 else 2}; \\
\text{alignment} & = \text{if index\_align}<0> = '0' \text{ then 1 else 4}; \\
d & = \text{UInt}(\text{D:Vd}); \ d2 & = d + inc; \ n & = \text{UInt}(\text{Rn}); \ m & = \text{UInt}(\text{Rm}); \\
\text{wback} & = (m \neq 15); \ \text{register\_index} = (m \neq 15 \&\& m \neq 13); \\
\text{if} \ n & = 15 || d2 > 31 \text{ then UNPREDICTABLE;}
\end{align*}
\]

CONSTRANDED UNPREDICTABLE behavior
If \( d2 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3

\[
\begin{array}{ccccccccccccccccccc}
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 & 16 & 15 & 12 & 11 & 10 & 9 & 8 & 7 & 4 & 3 & 0 |
\hline
1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & D & 0 & 0 & \text{Rn} & \text{Vd} & 1 & 0 & 0 & 1 & \text{index\_align} & \text{Rm}
\end{array}
\]

size

Offset variant
Applies when \( Rm = 1111 \).
\[
\text{VST2}\{<c>\}{<q>}.<\text{size}> \text{ <list>, } [<\text{Rn}>{:<\text{align}>}]
\]

Post-indexed variant
Applies when \( Rm = 1101 \).
\[
\text{VST2}\{<c>\}{<q>}.<\text{size}> \text{ <list>, } [<\text{Rn}>{:<\text{align}>}]!
\]

Post-indexed variant
Applies when \( Rm \neq 11x1 \).
\[
\text{VST2}\{<c>\}{<q>}.<\text{size}> \text{ <list>, } [<\text{Rn}>{:<\text{align}>}], <\text{Rm}>
\]
Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
</tr>
<tr>
<td>0</td>
<td>15</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>index_align</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Offset variant

Applies when Rm == 1111.
VST2{<c>}{<q>}.<size> <list>, [Rn]{:<align>}

Post-indexed variant

Applies when Rm == 1101.
VST2{<c>}{<q>}.<size> <list>, [Rn]{:<align>}!

Post-indexed variant

Applies when Rm != 11x1.
VST2{<c>}{<q>}.<size> <list>, [Rn]{:<align>}, <Rm>

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

**T2**

\[
\begin{array}{cccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 12 & 11 & 10 & 9 & 8 & 7 & 4 & 3 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & D & 0 & 0 & Rn & Vd & 0 & 1 & 0 & 1 & index \_align & Rm
\end{array}
\]

**Offset variant**
Applies when \( Rm = 1111 \).

\[ \text{VST2}\{<c>\}{<q>}.<\text{size}> \langle\text{list}\rangle, [\langle Rn\rangle{:<\text{align}}]\]  

**Post-indexed variant**
Applies when \( Rm = 1101 \).

\[ \text{VST2}\{<c>\}{<q>}.<\text{size}> \langle\text{list}\rangle, [\langle Rn\rangle{:<\text{align}}]!\]  

**Post-indexed variant**
Applies when \( Rm \neq 1111 \).

\[ \text{VST2}\{<c>\}{<q>}.<\text{size}> \langle\text{list}\rangle, [\langle Rn\rangle{:<\text{align}}], \langle Rm\rangle\]

**Decode for all variants of this encoding**

\[
\text{if size} = '11' \text{ then } \text{UNDEFINED;}
\]

\[
eb\text{bytes} = 2; \ \text{index} = \text{UInt(index\_align<3:2>)};
\]

\[
\text{inc} = \text{if index\_align<1> = '0' then } 1 \text{ else } 2;
\]

\[
\text{alignment} = \text{if index\_align<0> = '0' then } 1 \text{ else } 4;
\]

\[
\text{d} = \text{UInt(D:Vd)}; \ \text{d2} = \text{d} + \text{inc}; \ \text{n} = \text{UInt(Rn)}; \ \text{m} = \text{UInt(Rm)};
\]

\[
\text{wback} = (\text{m} \neq 15); \ \text{register\_index} = (\text{m} \neq 15 \ \&\& \ \text{m} \neq 13);
\]

\[
\text{if n} = 15 \ || \ d2 > 31 \text{ then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( d2 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

**T3**

\[
\begin{array}{cccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 12 & 11 & 10 & 9 & 8 & 7 & 4 & 3 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & D & 0 & 0 & Rn & Vd & 1 & 0 & 0 & 1 & index \_align & Rm
\end{array}
\]

**Offset variant**
Applies when \( Rm = 1111 \).
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]  

Post-indexed variant  
Applies when Rm == 1101.

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed variant  
Applies when Rm != 11x1.

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>); 
inc = if index_align<2> == '0' then 1 else 2; 
alignment = if index_align<0> == '0' then 1 else 8; 
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm); 
wback = (m != 15); register_index = (m != 15 && m != 13); 
if n == 15 || d2 > 31 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior
If d2 > 31, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Notes for all encodings
For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VST2 (single 2-element structure from one lane) on page K1-11580.

Assembler symbols

<<> For encoding A1, A2 and A3: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1, T2 and T3: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<size> Is the data size, encoded in the "size" field. It can have the following values:

8 when size = 00
16 when size = 01
32 when size = 10

<list> Is a list containing the 64-bit names of the two SIMD&FP registers holding the element.
The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>] } Single-spaced registers, encoded as "spacing" = 0.
{<Dd>[<index>], <Dd+2>[<index>] } Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.

The encoding of "spacing" depends on <size>:
<size> == 16"spacing" is encoded in the "index_align<1>" field.
<size> == 32"spacing" is encoded in the "index_align<2>" field.
The register <Dd> is encoded in the "D:Vd" field.
The permitted values and encoding of <index> depend on <size>:
<size> == 8<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.
<size> == 16<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.
<size> == 32<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see Unaligned data access on page E2-7193, and the encoding depends on <size>:
<size> == 8Encoded in the "index_align<0>" field as 0.
<size> == 16Encoded in the "index_align<0>" field as 0.
<size> == 32Encoded in the "index_align<1:0>" field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:
<size> == 8<align> is 16, meaning 16-bit alignment, encoded in the "index_align<0>" field as 1.
<size> == 16<align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.
<size> == 32<align> is 64, meaning 64-bit alignment, encoded in the "index_align<1:0>" field as 0b01.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see The Advanced SIMD addressing mode on page F1-7249.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see The Advanced SIMD addressing mode on page F1-7249.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    MemU[address, ebytes] = Elem[D[d], index];
    MemU[address+ebytes, ebytes] = Elem[D[d2], index];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.239 VST2 (multiple 2-element structures)

Store multiple 2-element structures from two or four registers stores multiple 2-element structures from two or four registers to memory, with interleaving. For more information, see Element and structure load/store instructions on page F2-7278. Every element of each register is saved. For details of the addressing mode, see The Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | D | 0 | 0 | Rn | Vd | 1 | 0 | 0 | x | size | align | Rm |

**Offset variant**

Applies when Rm == 1111.

VST2{<c>}{<q>}{<size> <list}, [<Rn>{:<align>}

**Post-indexed variant**

Applies when Rm == 1101.

VST2{<c>}{<q>}{<size> <list}, [<Rn>{:<align>}]!

**Post-indexed variant**

Applies when Rm != 11x1.

VST2{<c>}{<q>}{<size> <list}, [<Rn>{:<align}>], <Rm>

**Decode for all variants of this encoding**

pairs = 1; if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
inc = if itype == '1001' then 2 else 1;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wbback = (m != 15); register_index = (m != 15 & & m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If d2+pairs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.
A2

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16 15</th>
<th>12 11 10 9</th>
<th>8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 0 0 0</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>0 0 1 1</td>
<td>size</td>
</tr>
</tbody>
</table>

**Offset variant**

Applies when \( Rm == 1111 \).

\[ VST2\{<c>\}<q>\}.<size> <list>, [<Rn}\{:<align>\} \]

**Post-indexed variant**

Applies when \( Rm == 1101 \).

\[ VST2\{<c>\}<q>\}.<size> <list>, [<Rn}\{:<align>\}], <Rm> \]

**Post-indexed variant**

Applies when \( Rm != 11x1 \).

\[ VST2\{<c>\}<q>\}.<size> <list>, [<Rn}\{:<align>\}], <Rm> \]

**Decode for all variants of this encoding**

\[
pairs = 2; \quad inc = 2;
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); \quad elements = 8 DIV ebytes;
d = UInt(D:Vd); \quad d2 = d + inc; \quad n = UInt(Rn); \quad m = UInt(Rm);
wback = (m != 15); \quad register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( d2+pairs > 32 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

| 15 14 13 12 | 11 10 9 | 8 | 7 6 5 4 | 3 0 | 12 | 11 | 8 | 7 6 5 4 | 3 0 |
|--------------|---------|----|---|---|---|---|---|---|---|---|
| 1 1 1 1 0 1 0 0 0 | D | 0 | 0 | Rn | Vd | 1 0 0 x | size | align | Rm |

**Offset variant**

Applies when \( Rm == 1111 \).

\[ VST2\{<c>\}<q>\}.<size> <list>, [<Rn}\{:<align>\} \]

**Post-indexed variant**

Applies when \( Rm == 1101 \).
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]]!

**Post-indexed variant**

Applies when \( Rm \neq 11x1 \).

\[
\text{VST2}{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}, <Rm>
\]

**Decode for all variants of this encoding**

\[
\text{pairs} = 1; \text{ if align} = '11' \text{ then UNDEFINED;}
\text{if size} = '11' \text{ then UNDEFINED;}
\text{inc} = \text{ if itype} = '1001' \text{ then 2 else 1;}
\text{alignment} = \text{ if align} = '00' \text{ then 1 else 4} \ll \text{ UInt(align);} 
\text{ebytes} = 1 \ll \text{ UInt(size); elements} = 8 \text{ DIV ebytes;}
\text{d} = \text{ UInt(D:Vd); d2 = d + inc; n} = \text{ UInt(Rn); m} = \text{ UInt(Rm);}
\text{wback} = (m != 15); \text{ register_index} = (m != 15 & m != 13); 
\text{if n == 15 || d2+pairs > 32 then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( d2+\text{pairs} > 32 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

**T2**

| 15 14 13 12|11 10 9 8 7 6 5 4 3 0 |12|11 10 9 8 7 6 5 4 3 0 |1 1 1 1 0 1 0 D 0 0 Rn Vd 0 0 1 1 size align Rm |

**Offset variant**

Applies when \( Rm == 1111 \).

\[
\text{VST2}{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}
\]

**Post-indexed variant**

Applies when \( Rm == 1101 \).

\[
\text{VST2}{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
\]

**Post-indexed variant**

Applies when \( Rm != 11x1 \).

\[
\text{VST2}{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}, <Rm>
\]

**Decode for all variants of this encoding**

\[
\text{pairs} = 2; \text{ inc} = 2; \text{ if size} = '11' \text{ then UNDEFINED;}
\text{alignment} = \text{ if align} = '00' \text{ then 1 else 4} \ll \text{ UInt(align);} 
\text{ebytes} = 1 \ll \text{ UInt(size); elements} = 8 \text{ DIV ebytes;}
\text{d} = \text{ UInt(D:Vd); d2 = d + inc; n} = \text{ UInt(Rn); m} = \text{ UInt(Rm);}
\]
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If \( d2+pairs \) > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly VST2 (multiple 2-element structures) on page K1-11579.

Related encodings: See Advanced SIMD element or structure load/store on page F3-7350 for the T32 instruction set, or Advanced SIMD element or structure load/store on page F4-7434 for the A32 instruction set.

**Assembler symbols**

<\c> For encoding A1 and A2: see *Standard assembler syntax fields* on page F1-7228. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields* on page F1-7228.

<\q> See *Standard assembler syntax fields* on page F1-7228.

<size> Is the data size, encoded in the "size" field. It can have the following values:

- 8 when \( size = 00 \)
- 16 when \( size = 01 \)
- 32 when \( size = 10 \)

The encoding \( size = 11 \) is reserved.

<\list> Is a list containing the 64-bit names of the SIMD\&FP registers.

The list must be one of:

- \{ \langle Dd\rangle, \langle Dd+1\rangle \} Two single-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as 0b1000.
- \{ \langle Dd\rangle, \langle Dd+2\rangle \} Two double-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as 0b1001.
- \{ \langle Dd\rangle, \langle Dd+1\rangle, \langle Dd+2\rangle, \langle Dd+3\rangle \} Three single-spaced registers. Selects the A2 and T2 encodings of the instruction.

The register \langle Dd\rangle is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access on page E2-7193*, and is encoded in the "align" field as 0b00.

Whenever <align> is present, the permitted values are:

- 64 64-bit alignment, encoded in the "align" field as 0b01.
- 128 128-bit alignment, encoded in the "align" field as 0b10.
256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as 0<align>, see The Advanced SIMD addressing mode on page F1-7249.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see The Advanced SIMD addressing mode on page F1-7249.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to pairs-1
        for e = 0 to elements-1
            MemU[address, ebytes] = Elem[D[d+r], e];
            MemU[address+ebytes, ebytes] = Elem[D[d2+r], e];
            address = address + 2*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 16*pairs;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.240  VST3 (single 3-element structure from one lane)

Store single 3-element structure from one lane of three registers stores one 3-element structure to memory from corresponding elements of three registers. For details of the addressing mode, see The Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

Offset variant
Applies when Rm == 1111.
VST3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed variant
Applies when Rm == 1101.
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed variant
Applies when Rm != 1111.
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior
If d3 > 31, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2
Offset variant
Applies when Rm == 1111.
VST3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed variant
Applies when Rm == 1101.
VST3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed variant
Applies when Rm != 11x1.
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding

\[
\begin{align*}
\text{if size} & \text{ == '11' then UNDEFINED;} \\
\text{if index\_align<0> != '0' then UNDEFINED;} \\
e\text{bytes} & = 2; \quad \text{index} = \text{UInt(index\_align<3:2>);} \\
\text{inc} & = \text{if index\_align<1> == '0' then 1 else 2;} \\
d & = \text{UInt(D:Vd)}; \quad d2 = d + \text{inc}; \quad d3 = d2 + \text{inc}; \quad n = \text{UInt(Rn)}; \quad m = \text{UInt(Rm)}; \\
w\text{back} & = (m \neq 15); \quad \text{register\_index} = (m \neq 15 && m \neq 13); \\
\text{if n} & \text{ == 15 || d3 > 31 then UNPREDICTABLE;}
\end{align*}
\]

CONstrained UNPREDICTABLE behavior
If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 |8 |7 |4 |3 |0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 1 1 1 |0 1 0 1 |0 |1 0 |0 |0 |Rn |Vd |1 0 |1 0 |index\_align |Rm |
\|size
```
Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

| 15 14 13 12 | 11 10 9 8 7 6 5 4 3 | 0 | 15 | 12 | 11 10 9 8 7 | 4 | 3 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | D | 0 | 0 | Rn | Vd | 0 | 0 | 1 | 0 | index_align | Rm |

Offset variant

Applies when Rm == 1111.

VST3{<c>}{<q>}.<size> <list>, [Rn]

Post-indexed variant

Applies when Rm == 1101.

VST3{<c>}{<q>}.<size> <list>, [Rn]!

Post-indexed variant

Applies when Rm != 11x1.

VST3{<c>}{<q>}.<size> <list>, [Rn], [Rm]

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = 1;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

CONSTRANGED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

**Offset variant**

Applies when \( Rm = 1111 \).

\[
\text{VST3}\{<c>\}{<q>}.<\text{size}> <\text{list}>, [<Rn>]
\]

**Post-indexed variant**

Applies when \( Rm = 1101 \).

\[
\text{VST3}\{<c>\}{<q>}.<\text{size}> <\text{list}>, [<Rn>]!
\]

**Post-indexed variant**

Applies when \( Rm \neq 11x1 \).

\[
\text{VST3}\{<c>\}{<q>}.<\text{size}> <\text{list}>, [<Rn>], <Rm>
\]

**Decode for all variants of this encoding**

\[
\text{if size} = '11' \text{ then UNDEFINED}; \\
\text{if index\_align}\langle0\rangle = '0' \text{ then UNDEFINED}; \\
ebytes = 2; \text{ index} = \text{UInt(index\_align}\langle3:2\rangle); \\
inc = \text{if index\_align}\langle1\rangle = '0' \text{ then 1 else 2}; \\
d = \text{UInt(D:Vd)}; \text{ d2} = d + \text{inc}; \text{ d3} = d2 + \text{inc}; \text{ n} = \text{UInt(Rn)}; \text{ m} = \text{UInt(Rm)}; \\
wback = (m != 15); \text{ register\_index} = (m != 15 \&\& m != 13); \\
\text{if n} = 15 || d3 > 31 \text{ then UNPREDICTABLE};
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( d3 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as \text{NOP}.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

**Offset variant**

Applies when \( Rm = 1111 \).
VST3{<c>}{<q>}.<size> <list>, [<Rn>]

**Post-indexed variant**

Applies when Rm == 1101.

`VST3{<c>}{<q>}.<size> <list>, [<Rn>!]`  

**Post-indexed variant**

Applies when Rm != 1111.

`VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>`

**Decode for all variants of this encoding**

```c
if size == '11' then UNDEFINED;
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

**CONSTRAINED UNPREDICTABLE behavior**

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

**Notes for all encodings**

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly VST3 (single 3-element structure from one lane) on page K1-11580.

**Assembler symbols**

- `<<>>` For encoding A1, A2 and A3: see *Standard assembler syntax fields* on page F1-7228. This encoding must be unconditional.
  
  For encoding T1, T2 and T3: see *Standard assembler syntax fields* on page F1-7228.

- `<<>>` See *Standard assembler syntax fields* on page F1-7228.

- `<size>` Is the data size, encoded in the "size" field. It can have the following values:
  
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>when size = 00</td>
</tr>
<tr>
<td>16</td>
<td>when size = 01</td>
</tr>
<tr>
<td>32</td>
<td>when size = 10</td>
</tr>
</tbody>
</table>

- `<list>` Is a list containing the 64-bit names of the three SIMD&FP registers holding the element.
  The list must be one of:
  
  `{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>] } Single-spaced registers, encoded as "spacing" = 0.`
\{<Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>]\} Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.

The encoding of "spacing" depends on <size>:
- <size> == 8"spacing" is encoded in the "index_align<0>" field.
- <size> == 16"spacing" is encoded in the "index_align<1>" field, and "index_align<0>" is set to 0.
- <size> == 32"spacing" is encoded in the "index_align<2>" field, and "index_align<1:0>" is set to 0b00.

The register <Dd> is encoded in the "D:Vd" field.

The permitted values and encoding of <index> depend on <size>:
- <size> == 8<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.
- <size> == 16<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.
- <size> == 32<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see The Advanced SIMD addressing mode on page F1-7249.

Alignment

Standard alignment rules apply, see Alignment support on page B2-189.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n];
    MemU[address, ebytes] = Elem[D[d], index];
    MemU[address+ebytes, ebytes] = Elem[D[d2], index];
    MemU[address+2*ebytes, ebytes] = Elem[D[d3], index];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 3*ebytes;
    else
        R[n] = R[n] + 3*ebytes;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.241  VST3 (multiple 3-element structures)

Store multiple 3-element structures from three registers stores multiple 3-element structures to memory from three registers, with interleaving. For more information, see Element and structure load/store instructions on page F2-7278. Every element of each register is saved. For details of the addressing mode, see The Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

\[
\begin{array}{ccccccccccccccccccc}
\text{it} & 31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 & 23 & 22 & 21 & 20 & 19 & 16 & 15 & 12 & 11 & 8 & 7 & 6 & 5 & 4 & 3 & 0 \\
1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & D & 0 & 0 & Rn & Vd & 0 & 1 & 0 & x & size & align & Rm
\end{array}
\]

**Offset variant**

Applies when \( Rm = 1111 \).

\[
\text{VST3}\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align}>}
\]

**Post-indexed variant**

Applies when \( Rm = 1101 \).

\[
\text{VST3}\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align}>]}!
\]

**Post-indexed variant**

Applies when \( Rm \neq 11x1 \).

\[
\text{VST3}\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align}>}], <Rm>
\]

**Decode for all variants of this encoding**

\[
\text{if size} = \text{'11' || align}<1> = '1' \text{ then UNDEFINED; integer inc; case itype of when '0100' inc = 1; when '0101' inc = 2; otherwise SEE "Related encodings"; alignment = if align}<0> = '0' \text{ then 1 else 8; ebytes = 1 << UInt(size); elements = 8 DIV ebytes; d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm); wback = (m != 15); register_index = (m != 15 && m != 13); if n == 15 || d3 > 31 then UNPREDICTABLE;}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( d3 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.
### Offset variant

Applies when $Rm = 1111$.

$$VST3\{<c>\}{<q>}.<size>\ <list>,\ [<Rn}\{:<align}\]$$

### Post-indexed variant

Applies when $Rm = 1101$.

$$VST3\{<c>\}{<q>}.<size>\ <list>,\ [<Rn}\{:<align}\]!$$

### Post-indexed variant

Applies when $Rm != 11x1$.

$$VST3\{<c>\}{<q>}.<size>\ <list>,\ [<Rn}\{:<align}\],\ <Rm>$$

#### Decode for all variants of this encoding

```plaintext
if size == '11' || align<1> == '1' then UNDEFINED;
integer inc;
case itype of
  when '0100'
    inc = 1;
  when '0101'
    inc = 2;
  otherwise
    SEE "Related encodings";
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If $d3 > 31$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

#### Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly $VST3$ (multiple 3-element structures) on page K1-11580.

Related encodings: See Advanced SIMD element or structure load/store on page F3-7350 for the T32 instruction set, or Advanced SIMD element or structure load/store on page F4-7434 for the A32 instruction set.
Assembler symbols

<c>
For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

<size>
Is the data size, encoded in the "size" field. It can have the following values:

<table>
<thead>
<tr>
<th>Size Value</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>size = 00</td>
</tr>
<tr>
<td>16</td>
<td>size = 01</td>
</tr>
<tr>
<td>32</td>
<td>size = 10</td>
</tr>
</tbody>
</table>

The encoding size = 11 is reserved.

<list>
Is a list containing the 64-bit names of the SIMD&FP registers.
The list must be one of:

- \{ <Dd>, <Dd+1>, <Dd+2> \} Single-spaced registers, encoded in the "itype" field as 0b0100.
- \{ <Dd>, <Dd+2>, <Dd+4> \} Double-spaced registers, encoded in the "itype" field as 0b0101.

The register <Dd> is encoded in the "D:Vd" field.

<Rn>
Is the general-purpose base register, encoded in the "Rn" field.

<align>
Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see Unaligned data access on page E2-7193, and is encoded in the "align" field as 0b00.
Whenever <align> is present, the only permitted values is 64, meaning 64-bit alignment, encoded in the "align" field as 0b01.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see The Advanced SIMD addressing mode on page F1-7249.

<Rm>
Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see The Advanced SIMD addressing mode on page F1-7249.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for e = 0 to elements-1
        MemU[address, ebytes] = Elem[D[d], e];
        MemU[address+ebytes, ebytes] = Elem[D[d2], e];
        MemU[address+2*ebytes, ebytes] = Elem[D[d3], e];
        address = address + 3*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 24;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.242 VST4 (single 4-element structure from one lane)

Store single 4-element structure from one lane of four registers stores one 4-element structure to memory from corresponding elements of four registers. For details of the addressing mode, see The Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

 Offset variant

Applies when Rm == 1111.

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

 Post-indexed variant

Applies when Rm == 1101.

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

 Post-indexed variant

Applies when Rm != 11x1.

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if size != '00' then SEE "Related encodings";
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
alignment = if index_align<0> == '0' then 1 else 4;
    d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as NOP.
• The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.
A2

| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 |12|11 10 9 8 7 |4 |3 |0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|1 |1 |1 |1 |0 |1 |0 |0 |1 |D |0 |0 |Rn |Vd |0 |1 |1 |index_align |Rm |

**Offset variant**

Applies when Rm == 1111.

VST4{<c>}{<q>}{<size> <list>, [<Rn>{:<align>}]}

**Post-indexed variant**

Applies when Rm == 1101.

VST4{<c>}{<q>}{<size> <list>, [<Rn>{:<align>}]!}

**Post-indexed variant**

Applies when Rm != 11x1.

VST4{<c>}{<q>}{<size> <list>, [<Rn>{:<align>}]}, <Rm>

**Decode for all variants of this encoding**

- if size == '11' then UNDEFINED;
- if size != '01' then SEE "Related encodings";
- inc = if index_align<3:2> == '0' then 1 else 2;
- alignment = if index_align<2:0> == '0' then 1 else 8;
- d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
- wback = (m != 15); register_index = (m != 15 && m != 13);
- if n == 15 || d4 > 31 then UNPREDICTABLE;

**CONSTRAINED UNPREDICTABLE behavior**

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3

| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 |12|11 10 9 8 7 |4 |3 |0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|1 |1 |1 |1 |0 |1 |0 |0 |1 |D |0 |0 |Rn |Vd |1 |0 |1 |index_align |Rm |

**Offset variant**

Applies when Rm == 1111.

VST4{<c>}{<q>}{<size> <list>, [<Rn>{:<align>}]}


Post-indexed variant

Applies when \( Rm = 1101 \).

\[
\text{VST4}\{<c>}{<q>}.<\text{size} > \text{ <list>}, \ [<Rn>{:<\text{align}>}]!
\]

Post-indexed variant

Applies when \( Rm != 11x1 \).

\[
\text{VST4}\{<c>}{<q>}.<\text{size} > \text{ <list>}, \ [<Rn>{:<\text{align}>}], \ <Rm>
\]

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if size != '10' then SEE "Related encodings";
if index_align<1:0> == '11' then UNDEFINED;

\[
ebytes = 4; \text{ index} = \text{ UInt}(\text{index_align}<3>);
\]

inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 &lt; \text{ UInt}(\text{index_align}<1:0>);
\]

\[
d = \text{ UInt}(D:\text{Vd}); \ d2 = d + \text{ inc}; \ d3 = d2 + \text{ inc}; \ d4 = d3 + \text{ inc}; \ n = \text{ UInt}(Rn); \ m = \text{ UInt}(Rm);
\]

wback = (m != 15); \ register_index = (m != 15 & m != 13);
if size == '11' then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If \( d4 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as \( \text{NOP} \).
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

| 15 14 13 12 | 11 10 9 8 7 6 5 4 3 0 | 15 12 | 11 10 9 8 7 4 3 0 |
| 1 1 1 1 1 0 0 1 1 | D 0 0 | Rn | Vd | 0 0 1 1 | index_align | Rm |

Offset variant

Applies when \( Rm == 1111 \).

\[
\text{VST4}\{<c>}{<q>}.<\text{size} > \text{ <list>}, \ [<Rn>{:<\text{align}>}]!
\]

Post-indexed variant

Applies when \( Rm != 1101 \).

\[
\text{VST4}\{<c>}{<q>}.<\text{size} > \text{ <list>}, \ [<Rn>{:<\text{align}>}]!
\]

Post-indexed variant

Applies when \( Rm != 11x1 \).

\[
\text{VST4}\{<c>}{<q>}.<\text{size} > \text{ <list>}, \ [<Rn>{:<\text{align}>}], \ <Rm>
\]
Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if size != '00' then SEE “Related encodings”;
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 & & m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3</th>
<th>0 1 12 11 10 9 8 7 4 3 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 0 1 1 D 0 0 Rn</td>
<td>Vd</td>
<td>0 1 1 1 index_align Rm</td>
</tr>
</tbody>
</table>

Offset variant

Applies when Rm == 1111.

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed variant

Applies when Rm == 1101.

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed variant

Applies when Rm != 11x1.

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if size != '01' then SEE “Related encodings”;
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 & & m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONSTRANDED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
• The instruction executes as NOP.
• The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

**T3**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>1</td>
</tr>
</tbody>
</table>

### Offset variant

Applies when Rm == 1111.

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

### Post-indexed variant

Applies when Rm != 11x1.

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

### Post-indexed variant

Applies when Rm == 1101.

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

### Decode for all variants of this encoding

if size == '11' then UNDEFINED;
if size != '10' then SEE "Related encodings";
if index_align<1:0> == '11' then UNDEFINED;
  ebytes = 4;  index = UInt(index_align<3>);
  inc = if index_align<2> == '0' then 1 else 2;
  alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
  d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;
  n = UInt(Rn);  m = UInt(Rm);
  wback = (m != 15);  register_index = (m != 15 && m != 13);
  if n == 15 || d4 > 31 then UNPREDICTABLE;

### CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

### Notes for all encodings

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VST4 (single 4-element structure from one lane) on page K1-11580.
Assembler symbols

<<c>>

For encoding A1, A2 and A3: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1, T2 and T3: see Standard assembler syntax fields on page F1-7228.

<<p>>

See Standard assembler syntax fields on page F1-7228.

<<size>>

Is the data size, encoded in the "size" field. It can have the following values:

- 8 when size = 00
- 16 when size = 01
- 32 when size = 10

<<list>>

Is a list containing the 64-bit names of the four SIMD&FP registers holding the element. The list must be one of:

- \{<Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>], <Dd+3>[<index>]\} Single-spaced registers, encoded as "spacing" = 0.
- \{<Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>], <Dd+6>[<index>]\} Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.

The encoding of "spacing" depends on <size>:

- <size> == 16"spacing" is encoded in the "index_align<1>" field.
- <size> == 32"spacing" is encoded in the "index_align<2>" field.

The register <Dd> is encoded in the "D:Vd" field.

The permitted values and encoding of <index> depend on <size>:

- <size> == 8<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.
- <size> == 16<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.
- <size> == 32<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn>>

Is the general-purpose base register, encoded in the "Rn" field.

<align>

Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see Unaligned data access on page E2-7193, and the encoding depends on <size>:

- <size> == 8Encoded in the "index_align<0>" field as 0.
- <size> == 16Encoded in the "index_align<0>" field as 0.
- <size> == 32Encoded in the "index_align<1:0>" field as 00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

- <size> == 8<align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.
- <size> == 16<align> is 64, meaning 64-bit alignment, encoded in the "index_align<0>" field as 1.
- <size> == 32<align> can be 64 or 128. 64-bit alignment is encoded in the "index_align<1:0>" field as 0b01, and 128-bit alignment is encoded in the "index_align<1:0>" field as 0b10.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see The Advanced SIMD addressing mode on page F1-7249.

<Rm>>

Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see The Advanced SIMD addressing mode on page F1-7249.
Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    MemU[address,         ebytes] = Elem[D[d], index];
    MemU[address+ebytes, ebytes] = Elem[D[d2],index];
    MemU[address+2*ebytes,ebytes] = Elem[D[d3],index];
    MemU[address+3*ebytes,ebytes] = Elem[D[d4],index];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.243 VST4 (multiple 4-element structures)

Store multiple 4-element structures from four registers to memory from four registers, with interleaving. For more information, see Element and structure load/store instructions on page F2-7278. Every element of each register is saved. For details of the addressing mode, see The Advanced SIMD addressing mode on page F1-7249.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

Offset variant

Applies when \( Rm = 1111 \).

\[
VST4\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align}>]
\]

Post-indexed variant

Applies when \( Rm = 1101 \).

\[
VST4\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align}>]!
\]

Post-indexed variant

Applies when \( Rm \neq 11x1 \).

\[
VST4\{<c>\}{<q>}.<size> <list>, [<Rn>{:<align}>]}, <Rm>
\]

Decode for all variants of this encoding

if \( size = '11' \) then UNDEFINED;
integer inc;
case itype of
when '0000'
inc = 1;
when '0001'
inc = 2;
otherwise
SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << Uint(align);
ebytes = 1 << Uint(size); elements = 8 DIV ebytes;
d = Uint(D:Vd);
d2 = d + inc;
d3 = d2 + inc;
d4 = d3 + inc;
n = Uint(Rn); m = Uint(Rm);
wback = (m != 15);
register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If \( d4 > 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.
Offset variant
Applies when \( Rm = 1111 \).
\[
\text{VST4}\{<c>\}{<q>}.<size> \text{ <list>}, \[<Rn>\{:<align>\}]
\]

Post-indexed variant
Applies when \( Rm = 1101 \).
\[
\text{VST4}\{<c>\}{<q>}.<size> \text{ <list>}, \[<Rn>\{:<align>\}]!
\]

Post-indexed variant
Applies when \( Rm \neq 111x \).
\[
\text{VST4}\{<c>\}{<q>}.<size> \text{ <list>}, \[<Rn>\{:<align>\}], <Rm>
\]

Decode for all variants of this encoding
\[
\text{if size == '11' then UNDEFINED;}
\text{integer inc;}
\text{case itype of}
\quad \text{when '0000', inc = 1;}
\quad \text{when '0001', inc = 2;}
\quad \text{otherwise}
\quad \text{SEE "Related encodings";}
\text{alignment = if align == '00' then 1 else 4 << UInt(align);}
\text{ebytes = 1 << UInt(size); elements = 8 DIV ebytes;}
\text{d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);}
\text{wback = (m != 15); register_index = (m != 15 && m != 13);}
\text{if n == 15 || d4 > 31 then UNPREDICTABLE;}
\]

CONstrained UNPREDICTABLE behavior
If \( d4 > 31 \), then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Notes for all encodings
For more information about the CONstrained UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VST4 (multiple 4-element structures) on page K1-11580.

Related encodings: See Advanced SIMD element or structure load/store on page F3-7350 for the T32 instruction set, or Advanced SIMD element or structure load/store on page F4-7434 for the A32 instruction set.
Assembler symbols

<c>
For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

<size>
Is the data size, encoded in the "size" field. It can have the following values:
- 8 when size = 00
- 16 when size = 01
- 32 when size = 10
The encoding size = 11 is reserved.

<list>
Is a list containing the 64-bit names of the SIMD&FP registers.
The list must be one of:
- \{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> \} Single-spaced registers, encoded in the "itype" field as 0b0000.
- \{ <Dd>, <Dd+2>, <Dd+4>, <Dd+6> \} Double-spaced registers, encoded in the "itype" field as 0b0001.
The register <Dd> is encoded in the "D:Vd" field.

<Rn>
Is the general-purpose base register, encoded in the "Rn" field.

<align>
Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see Unaligned data access on page E2-7193, and is encoded in the "align" field as 0000.
Whenever <align> is present, the permitted values are:
- 64 64-bit alignment, encoded in the "align" field as 0b01.
- 128 128-bit alignment, encoded in the "align" field as 0b10.
- 256 256-bit alignment, encoded in the "align" field as 0b11.
: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see The Advanced SIMD addressing mode on page F1-7249.

<Rm>
Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.
For more information about the variants of this instruction, see The Advanced SIMD addressing mode on page F1-7249.

Operation for all encodings

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    address = R[n]; iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for e = 0 to elements-1
        Mem[address, ebytes] = Elem[D[d], e];
        Mem[address+ebytes, ebytes] = Elem[D[d2], e];
        Mem[address+2*ebytes, ebytes] = Elem[D[d3], e];
        Mem[address+3*ebytes, ebytes] = Elem[D[d4], e];
        address = address + 4*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 32;
Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.244  VSTM, VSTMDB, VSTMIA

Store multiple SIMD&FP registers stores multiple registers from the Advanced SIMD and floating-point register file to consecutive memory locations using an address from a general-purpose register.

Depending on settings in the CPACR, NSACR, HCPR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see Enabling Advanced SIMD and floating-point support on page G1-8996.

This instruction is used by the alias VPUSH. See Alias conditions on page F6-8845 for details of when each alias is preferred.

A1

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!1111</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>P</td>
<td>U</td>
<td>D</td>
<td>W</td>
<td>0</td>
<td>Rn</td>
<td>Vd</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>imm8&lt;7:1&gt;</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Decrement Before variant

Applies when P == 1 && U == 0 && W == 1.

VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>

Increment After variant

Applies when P == 0 && U == 1.

VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

Decode for all variants of this encoding

if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VSTR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE; add = (U == '1'); wback = (W == '1');
d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:<0>, 32);
regs = UInt(imm8) DIV 2; // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX",
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;

CONstrained UNpredictable behavior

If regs == 0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If regs > 16 || (d+regs) > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
• The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2

<table>
<thead>
<tr>
<th>31 28 27 26 25 24</th>
<th>23 22 21 20 19 16 15 12</th>
<th>11 10 9 8 7 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!l=1111</td>
<td>1 1 0</td>
<td>P</td>
</tr>
<tr>
<td>cond</td>
<td>Rn</td>
<td>Vd</td>
</tr>
</tbody>
</table>

**Decrement Before variant**

Applies when \( P = 1 \land U = 0 \land W = 1 \).

VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>

**Increment After variant**

Applies when \( P = 0 \land U = 1 \).

VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>

\( \text{VSTMIA}{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist> \)

**Decode for all variants of this encoding**

\[
\begin{align*}
&\text{if } P = '0' \land U = '0' \land W = '0' \text{ then SEE "Related encodings";} \\
&\text{if } P = '1' \land U = '0' \land W = '0' \text{ then SEE "VSTR";} \\
&\text{if } P = '1' \land U = '1' \text{ then UNDEFINED;} \\
&\text{if } P = '0' \land U = '0' \land W = '0' \text{ then UNDEFINED;} \\
&\text{if } P = '0' \land U = '1' \land W = '0' \text{ then \(VSTM\) with the same addressing mode but stores no registers.}
\end{align*}
\]

\[
\begin{align*}
&\text{if } n = 15 \land (wback \lor \text{CurrentInstrSet}() = \text{InstrSet_A32}) \text{ then UNPREDICTABLE;} \\
&\text{if } (d+regs) > 32 \text{ then UNPREDICTABLE;} \\
\end{align*}
\]

**CONSTRAINED UNPREDICTABLE behavior**

If \( \text{regs} = 0 \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.

• The instruction executes as \( \text{NOP} \).

• The instruction operates as a \( \text{VSTM} \) with the same addressing mode but stores no registers.

If \( (d+regs) > 32 \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.

• The instruction executes as \( \text{NOP} \).

• The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 0 15 12</th>
<th>11 10 9 8 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 1 0</td>
<td>P</td>
<td>U</td>
</tr>
<tr>
<td>Rn</td>
<td>Vd</td>
<td>1 0 1 1</td>
</tr>
</tbody>
</table>

imm8<0>
Decrement Before variant
Applies when \( P = 1 \\&\& U = 0 \\&\& W = 1 \).
\( \text{VSTMDB}\{<c>\}{<q>}{.<size>} <Rn>!, <\text{dreglist}> \)

Increment After variant
Applies when \( P = 0 \\&\& U = 1 \).
\( \text{VSTM}\{<c>\}{<q>}{.<size>} <Rn>{!}, <\text{dreglist}> \)
\( \text{VSTMIA}\{<c>\}{<q>}{.<size>} <Rn>{!}, <\text{dreglist}> \)

Decode for all variants of this encoding

\[
\text{if } P = '0' \&\& U = '0' \&\& W = '0' \text{ then SEE "Related encodings";}
\text{if } P = '1' \&\& W = '0' \text{ then SEE "VSTR";}
\text{if } P = U \&\& W = '1' \text{ then UNDEFINED;}
\]

// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
\( \text{single_regs} = \text{FALSE}; \text{ add} = (U == '1'); \text{ wback} = (W == '1'); \)
\( d = \text{UInt}(D:Vd); n = \text{UInt}(Rn); \text{ imm32} = \text{ZeroExtend}(\text{imm8}: '00', 32); \)
\( \text{regs} = \text{UInt}(\text{imm8}): \text{DIV} 2; // \text{IF UInt(imm8) is odd, see "FSTDMX, FSTMIAX".} \)
\( \text{if } n = 15 \&\& (\text{wback} \| \text{CurrentInstrSet() != InstrSet_A32}) \text{ then UNPREDICTABLE;}
\text{if } \text{regs} = 0 || \text{regs} > 16 || (d+\text{regs}) > 32 \text{ then UNPREDICTABLE;}
\text{if } \text{imm8} \&\& \text{d} = '1' \&\& (d+\text{regs}) > 16 \text{ then UNPREDICTABLE;} \)

CONSTRAINED UNPREDICTABLE behavior
If \( \text{regs} = 0 \), then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If \( \text{regs} > 16 \| (d+\text{regs}) > 32 \), then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
11 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 0 & 15 & 12 & 11 & 10 & 9 & 8 & 7 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 0 & P & U & D & W & 0 & Rn & Vd & 1 & 0 & 1 & 0 & \text{imm8} \\
\hline
\end{array}
\]

Decrement Before variant
Applies when \( P = 1 \\&\& U = 0 \\&\& W = 1 \).
\( \text{VSTMDB}\{<c>\}{<q>}{.<size>} <Rn>!, <\text{dreglist}> \)

Increment After variant
Applies when \( P = 0 \\&\& U = 1 \).
\( \text{VSTM}\{<c>\}{<q>}{.<size>} <Rn>{!}, <\text{dreglist}> \)
\( \text{VSTMIA}\{<c>\}{<q>}{.<size>} <Rn>{!}, <\text{dreglist}> \)
Decode for all variants of this encoding

if P == '0' & U == '0' & W == '0' thenSEE "Related encodings";
if P == '1' & W == '0' thenSEE "VSTR";
if P == U & W == '1' thenUNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE; add = (U == '1'); wback = (W == '1'); d = UInt(Vd:D); n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32); regs = UInt(imm8);
if n == 15 & wback || CurrentInstrSet() != InstrSet_A32 then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;

CONSTRUED UNPREDICTABLE behavior

If regs == 0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If (d+regs) > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

Notes for all encodings

For more information about the CONSTRUED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors, and particularly VSTM on page K1-11580.

Related encodings: See Advanced SIMD and floating-point 64-bit move on page F3-7324 for the T32 instruction set, or Advanced SIMD and floating-point 64-bit move on page F4-7410 for the A32 instruction set.

Alias conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPUSH</td>
<td>P == '1' &amp; U == '0' &amp; W == '1' &amp; Rn == '1101'</td>
</tr>
</tbody>
</table>

Assembler symbols

<-> See Standard assembler syntax fields on page F1-7228.
<#> See Standard assembler syntax fields on page F1-7228.
<size> An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.
<Rn> Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used. However, Arm deprecates use of the PC.
! Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.
<reglist> Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.
<dreglist> Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            MemA[address,4] = S[d+r]; address = address+4;
        else
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4] = if BigEndian(AccType_ATOMIC) then D[d+r]<63:32> else D[d+r]<31:0>;
            MemA[address+4,4] = if BigEndian(AccType_ATOMIC) then D[d+r]<31:0> else D[d+r]<63:32>;
            address = address+8;
        if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
### F6.1.245 VSTR

Store SIMD&FP register stores a single register from the Advanced SIMD and floating-point register file to memory, using an address from a general-purpose register, with an optional offset.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

<table>
<thead>
<tr>
<th>A1</th>
</tr>
</thead>
</table>

**Half-precision scalar variant**

Applies when `size == 01`.

\[ \text{VSTR} \{<c>\} \{<q>\}.16 \ <Sd>, \ [\langle Rn\rangle, \ #\{+/-\}<imm>] \]

**Single-precision scalar variant**

Applies when `size == 10`.

\[ \text{VSTR} \{<c>\} \{<q>\}.{32} \ <Sd>, \ [\langle Rn\rangle, \ #\{+/-\}<imm>] \]

**Double-precision scalar variant**

Applies when `size == 11`.

\[ \text{VSTR} \{<c>\} \{<q>\}.{64} \ <Dd>, \ [\langle Rn\rangle, \ #\{+/-\}<imm>] \]

**Decode for all variants of this encoding**

```c
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
esize = 8 << UInt(size); add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8: '0', 32) else ZeroExtend(imm8: '00', 32);
integer d;
case size of
  when '01' d = UInt(Vd:D);
  when '10' d = UInt(Vd:D);
  when '11' d = UInt(D:Vd);
n = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UNPREDICTABLE;
```

**CONSTRAINED UNPREDICTABLE behavior**

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.
Half-precision scalar variant
Applies when size == 01.
VSTR{<c>}{<q>}.16 <Sd>, [<Rn>{, #{+/-}<imm>}] 

Single-precision scalar variant
Applies when size == 10.
VSTR{<c>}{<q>}{.32} <Sd>, [<Rn>{, #{+/-}<imm>}] 

Double-precision scalar variant
Applies when size == 11.
VSTR{<c>}{<q>}{.64} <Dd>, [<Rn>{, #{+/-}<imm>}] 

Decode for all variants of this encoding
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
esize = 8 << UInt(size); add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
integer d;
case size of
when '01' d = UInt(Vd:D);
when '10' d = UInt(Vd:D);
when '11' d = UInt(D:Vd);
n = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior
If size == '01' && InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Notes for all encodings
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols
<
See Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

.64
Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

<d>
Is the 64-bit name of the SIMD&FP source register, encoded in the "D:Vd" field.
.32  Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

<Sd>  Is the 32-bit name of the SIMD&FP source register, encoded in the "Vd:D" field.

<Rn>  Is the general-purpose base register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

<+/->  Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
-  when $U = 0$
+  when $U = 1$

<imm>  For the single-precision scalar or double-precision scalar variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as <imm>/4.

For the half-precision scalar variant: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as <imm>/2.

**Operation for all encodings**

if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
  address = if add then (R[n] + imm32) else (R[n] - imm32);
  case esize of
    when 16
      MemA[address,2] = S[d]<15:0>;
    when 32
      MemA[address,4] = S[d];
    when 64
      // Store as two word-aligned words in the correct order for current endianness.
      MemA[address,4] = if BigEndian(AccType_ATOMIC) then D[d]<63:32> else D[d]<31:0>;
      MemA[address+4,4] = if BigEndian(AccType_ATOMIC) then D[d]<31:0> else D[d]<63:32>;

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
F6.1.246    VSUB (floating-point)

Vector Subtract (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

|31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8|7 6 5 4|3 0 |
|---|---|---|---|---|---|---|---|---|
|1 1 1 0 0 1 0 0 | D 1 | sz | Vn | Vd | 1 1 0 1 | N Q M 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

$\text{VSUB}\{<c>\}\{<q>\}.<dt>\{<Dd>,\}<Dn>,<Dm>$

128-bit SIMD vector variant

Applies when Q == 1.

$\text{VSUB}\{<c>\}\{<q>\}.<dt>\{<Qd>,\}<Qn>,<Qm>$

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' | Vn<0> == '1' | Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

A2

|31 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8|7 6 5 4|3 0 |
|---|---|---|---|---|---|---|---|---|
|1=1111 | 1 1 1 0 0 | D 1 | Vn | Vd | 1 0 | size | N 1 M 0 | Vm |

Half-precision scalar variant

Applies when size == 01.

$\text{VSUB}\{<c>\}\{<q>\}.F16\{<Sd>,\}<Sn>,<Sm>$

Single-precision scalar variant

Applies when size == 10.

$\text{VSUB}\{<c>\}\{<q>\}.F32\{<Sd>,\}<Sn>,<Sm>$

Double-precision scalar variant

Applies when size == 11.
VSUB{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>

**Decode for all variants of this encoding**

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1111' then UNDEFINED;
advsimd = FALSE;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
end case

**CONSTRAINED UNPREDICTABLE behavior**

If size == '01' && cond != '1111', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

```
| 15 14 13 12 | 11 10 9 8 7 6 5 4 3 0 |
|-------------|
| 1 1 1 1 1 0 D 1 sz Vn Vd 1 1 0 1 N Q M 0 Vm |
```

**64-bit SIMD vector variant**

Applies when Q == 0.

VSUB{<c>}{<q>}.<dt> {<Dd>,} {<Dn>,} <Dm>

**128-bit SIMD vector variant**

Applies when Q == 1.

VSUB{<c>}{<q>}.<dt> {<Qd>,} {<Qn>,} <Qm>

**Decode for all variants of this encoding**

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
end case

**CONSTRAINED UNPREDICTABLE behavior**

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 | 1 1 1 0 1 1 0 0 D 1 1 | Vn | Vd | 1 0 | size | N 1 | M 0 | Vm |

**Half-precision scalar variant**
Applies when size == 01.
VSUB{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>

**Single-precision scalar variant**
Applies when size == 10.
VSUB{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>

**Double-precision scalar variant**
Applies when size == 11.
VSUB{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>

**Decode for all variants of this encoding**
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
advsimd = FALSE;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

**CONSTRAINED UNPREDICTABLE behavior**
If size == '01' && InITBlock(), then one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction executes as if it passes the Condition code check.
• The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

**Assembler symbols**
<
For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.
For encoding A2, T1 and T2: see *Standard assembler syntax fields on page F1-7228*.

> See *Standard assembler syntax fields on page F1-7228*.
Is the data type for the elements of the vectors, encoded in the "sz" field. It can have the following values:

- F32 when \( sz = 0 \)
- F16 when \( sz = 1 \)

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<Qd>^2\).

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as \(<Qn>^2\).

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as \(<Qm>^2\).

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDorVFPEnabled(TRUE, advsimd);
  if advsimd then  // Advanced SIMD instruction
    for r = 0 to regs-1
      for e = 0 to elements-1
        Elem[D[d+r],e,esize] = FPSub(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRValue());
  else             // VFP instruction
    case esize of
      when 16
        S[d] = Zeros(16) : FPSub(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
      when 32
        S[d] = FPSub(S[n], S[m], FPSCR[]);
      when 64
        D[d] = FPSub(D[n], D[m], FPSCR[]);
```
F6.1.247 VSUB (integer)

Vector Subtract (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

<table>
<thead>
<tr>
<th>11110011</th>
<th>0</th>
<th>0</th>
<th>D</th>
<th>size</th>
<th>Vn</th>
<th>Vd</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>N</th>
<th>Q</th>
<th>M</th>
<th>0</th>
<th>Vm</th>
</tr>
</thead>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

<table>
<thead>
<tr>
<th>11111011</th>
<th>0</th>
<th>0</th>
<th>D</th>
<th>size</th>
<th>Vn</th>
<th>Vd</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>N</th>
<th>Q</th>
<th>M</th>
<th>0</th>
<th>Vm</th>
</tr>
</thead>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
Assembler symbols

<
  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be
  unconditional.
  For encoding T1: see Standard assembler syntax fields on page F1-7228.
>
<
  See Standard assembler syntax fields on page F1-7228.

<dt>
  Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following
  values:
  I8  when size = 00
  I16 when size = 01
  I32 when size = 10
  I64 when size = 11

<Qd>
  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>
  Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>
  Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as
  <Qm>*2.

<Dd>
  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>
  Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>
  Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      Elem[D[r]+d, e, esize] = Elem[D[r]+n, e, esize] - Elem[D[r]+m, e, esize];

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.248  VSUBHN

Vector Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, takes the most significant half of each result, and places the final results in a doubleword vector. The results are truncated. For rounded results, see VRSUBHN.

There is no distinction between signed and unsigned integers.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 |7 6 5 4 |3 0 |
|-------------|-------------|-------------|-----|-----|-------|-----|-----|-----|
| 1 1 1 0 0 1 0 1 | D !=11 | Vn | Vd | 0 1 1 0 | N | M | 0 | Vm |

Notes for all encodings

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

Assembler symbols

<>  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
   For encoding T1: see Standard assembler syntax fields on page F1-7228.
See *Standard assembler syntax fields* on page F1-7228.

Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:

- **I16** when size = 00
- **I32** when size = 01
- **I64** when size = 10

Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as \(<Qn>\)*2.

Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as \(<Qm>\)*2.

**Operation for all encodings**

```plaintext
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] - Elem[Qin[m>>1],e,2*esize];
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Vector Subtract Long subtracts the elements of one doubleword vector from the corresponding elements of another doubleword vector, and places the results in a quadword vector. Before subtracting, it sign-extends or zero-extends the elements of both operands.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

### T1 variant

**VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>**

**Decode for this encoding**

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize; is_vsubw = (op == '1');
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

### Notes for all encodings

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

**Assembler symbols**

For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.

For encoding T1: see Standard assembler syntax fields on page F1-7228.
See *Standard assembler syntax fields* on page F1-7228.

Is the data type for the elements of the second operand vector, encoded in the "U:size" field. It can have the following values:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>S8</td>
<td>U = 0, size = 00</td>
</tr>
<tr>
<td>S16</td>
<td>U = 0, size = 01</td>
</tr>
<tr>
<td>S32</td>
<td>U = 0, size = 10</td>
</tr>
<tr>
<td>U8</td>
<td>U = 1, size = 00</td>
</tr>
<tr>
<td>U16</td>
<td>U = 1, size = 01</td>
</tr>
<tr>
<td>U32</td>
<td>U = 1, size = 10</td>
</tr>
</tbody>
</table>

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

```c
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        integer op1;
        if is_vsubw then
            op1 = Int(Elem[Qin][n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din][n],e,esize], unsigned);
        result = op1 - Int(Elem[Din][m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.250  VSUBW

Vector Subtract Wide subtracts the elements of a doubleword vector from the corresponding elements of a quadword vector, and places the results in another quadword vector. Before subtracting, it sign-extends or zero-extends the elements of the doubleword operand.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

```
| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4 |3 0 |
|-----------|-----------|-----------|---|---|---|---|---|
| 1 1 1 1 0 0 1| 1| D | 1=11| Vn| Vd| 0 0 1| 1| N| 0| M| 0| Vm|

size | op

A1 variant

VSUBW{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if Vd<> == '1' || (op == '1' && Vn<> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize; is_vsubw = (op == '1');
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1

```
| 15 14 13 12|11 10 9 8 7 6 5 4 |3 |0 |
|-----------|---|---|---|---|
| 1 1 1| 1| D | 1=11| Vn| Vd| 0 0 1| 1| N| 0| M| 0| Vm|

size | op

T1 variant

VSUBW{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>

Decode for this encoding

if size == '11' then SEE "Related encodings";
if Vd<> == '1' || (op == '1' && Vn<> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize; is_vsubw = (op == '1');
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

Notes for all encodings

Related encodings: See Advanced SIMD data-processing on page F3-7314 for the T32 instruction set, or Advanced SIMD data-processing on page F4-7422 for the A32 instruction set.

Assembler symbols

<
For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.
See Standard assembler syntax fields on page F1-7228.

Is the data type for the elements of the second operand vector, encoded in the "U:size" field. It can have the following values:

- S8 when U = 0, size = 00
- S16 when U = 0, size = 01
- S32 when U = 0, size = 10
- U8 when U = 1, size = 00
- U16 when U = 1, size = 01
- U32 when U = 1, size = 10

Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        integer op1;
        if is_vsubw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        result = op1 - Int(Elem[Din[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.251 VSUDOT (by element)

Dot Product index form with signed and unsigned integers. This instruction performs the dot product of the four signed 8-bit integer values in each 32-bit element of the first source register with the four unsigned 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

A1

(Feat_AA32I8MM)

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 16 15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1</td>
<td>1 1 0 1</td>
<td>D 0 0</td>
<td>Vn</td>
<td>Vd</td>
<td>1</td>
<td>1 0</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VSUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]

128-bit SIMD vector variant

Applies when Q == 1.

VSUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;

T1

(Feat_AA32I8MM)

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1</td>
<td>1 1 0 1</td>
<td>D 0 0</td>
<td>Vn</td>
</tr>
</tbody>
</table>

64-bit SIMD vector variant

Applies when Q == 0.

VSUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]

128-bit SIMD vector variant

Applies when Q == 1.

VSUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]
Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.

<index> Is the element index in the range 0 to 1, encoded in the "M" field.

Operation for all encodings

CheckAdvSIMDEnabled();
bits(64) operand1;
bits(64) operand2;
bits(64) result;
operand2 = Din[m];
for r = 0 to regs-1
    operand1 = Din[n+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(32) res = Elem[result, e, 32];
        for b = 0 to 3
            element1 = Int(Elem[operand1, 4 * e + b, 8], op1_unsigned);
            element2 = Int(Elem[operand2, 4 * i + b, 8], op2_unsigned);
            res = res + element1 * element2;
        Elem[result, e, 32] = res;
    D[d+r] = result;
F6.1.252 VSWP

Vector Swap exchanges the contents of two vectors. The vectors can be either doubleword or quadword. There is no distinction between data types.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

\[
\begin{array}{ccccccccccccccccc}
\text{size} & \text{D} & \text{Vd} & \text{Q} & \text{M} & \text{Vm} \\
\hline
111100111 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
\end{array}
\]

64-bit SIMD vector variant

Applies when Q == 0.

\[\text{VSWP}\{<c><q><.dt}\} \text{<Dd>, <Dm>}\]

128-bit SIMD vector variant

Applies when Q == 1.

\[\text{VSWP}\{<c><q><.dt}\} \text{<Qd>, <Qm>}\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{if size} & != '00' \text{ then UNDEFINED;} \\
\text{if Q} & == '1' \&\& (\text{Vd}<> == '1' | \text{Vm}<> == '1') \text{ then UNDEFINED;} \\
\text{d} & = \text{UInt(D:Vd)}; \ \text{m} = \text{UInt(M:Vm)}; \ \text{regs} = \text{if Q} == '0' \text{ then 1 else 2;}
\end{align*}
\]

T1

\[
\begin{array}{ccccccccccccccccc}
\text{size} & \text{D} & \text{Vd} & \text{Q} & \text{M} & \text{Vm} \\
\hline
111111111 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
\end{array}
\]

64-bit SIMD vector variant

Applies when Q == 0.

\[\text{VSWP}\{<c><q><.dt}\} \text{<Dd>, <Dm>}\]

128-bit SIMD vector variant

Applies when Q == 1.

\[\text{VSWP}\{<c><q><.dt}\} \text{<Qd>, <Qm>}\]

Decode for all variants of this encoding

\[
\begin{align*}
\text{if size} & != '00' \text{ then UNDEFINED;} \\
\text{if Q} & == '1' \&\& (\text{Vd}<> == '1' | \text{Vm}<> == '1') \text{ then UNDEFINED;} \\
\text{d} & = \text{UInt(D:Vd)}; \ \text{m} = \text{UInt(M:Vm)}; \ \text{regs} = \text{if Q} == '0' \text{ then 1 else 2;}
\end{align*}
\]
Assembler symbols

\(<c>\) For encoding A1: see *Standard assembler syntax fields on page F1-7228*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields on page F1-7228*.

\(<p>\) See *Standard assembler syntax fields on page F1-7228*.

\(<dt>\) An optional data type. It is ignored by assemblers, and does not affect the encoding.

\(<qdt>\) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as \(<Qd>*2.

\(<qdt>\) Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as \(<Qm>*2.

\(<Qd>\) Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<Qm>\) Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

\(<Dd>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

\(<Dm>\) Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

\[\text{if}\ \text{ConditionPassed()}\ \text{then}\]
\[\text{EncodingSpecificOperations(); CheckAdvSIMDEnabled();}\]
\[\text{for}\ r = 0 \text{ to } \text{regs-1}\]
\[\text{if}\ d == m\ \text{then}\]
\[D[d+r] = \text{bits(64)}\ \text{UNKOWN};\]
\[\text{else}\]
\[D[d+r] = \text{Din}[m+r];\]
\[D[m+r] = \text{Din}[d+r];\]

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.253 VTBL, VTBX

Vector Table Lookup uses byte indexes in a control vector to look up byte values in a table and generate a new vector. Indexes out of range return 0.

Vector Table Extension works in the same way, except that indexes out of range leave the destination element unchanged.

Depending on settings in the CPACR, NSACR, and HCPtr registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

| 31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4 3 0 |
| 1 1 1 0 1 1 1 | D | 1 1 | Vn | Vd | 1 0 | len | N | op | M | 0 | Vm |

VTBL variant

Applies when op == 0.

VTBL{<c>}{<q>}.8 <Dd>, <list>, <Dm>

VTBX variant

Applies when op == 1.

VTBX{<c>}{<q>}.8 <Dd>, <list>, <Dm>

Decode for all variants of this encoding

is_vtbl = (op == '0'); length = UInt(len)+1;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
if n+length > 32 then UNPREDICTABLE;

CONSTRAINED UNPREDICTABLE behavior

If n + length > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. This behavior does not affect any general-purpose registers.

T1

| 15 14 13 12|11 10 9 8 7 6 5 4 3 0 | 15 12|11 10 9 8 7 6 5 4 3 0 |
| 1 1 1 1 1 1 1 | D | 1 1 | Vn | Vd | 1 0 | len | N | op | M | 0 | Vm |

VTBL variant

Applies when op == 0.

VTBL{<c>}{<q>}.8 <Dd>, <list>, <Dm>
VTBX variant
Applies when op == 1.

```
VTBX{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```

Decode for all variants of this encoding

```
is_vtbl = (op == '0');  length = UInt(len)+1;

d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);

if n+length > 32 then UNPREDICTABLE;
```

CONSTRANDED UNPREDICTABLE behavior

If n + length > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. This behavior does not affect any general-purpose registers.

Notes for all encodings

For more information about the CONSTRANDED UNPREDICTABLE behavior of this instruction, see Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

Assembler symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
</table>
| <c>   | For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.  
|       | For encoding T1: see Standard assembler syntax fields on page F1-7228. |
| <q>   | See Standard assembler syntax fields on page F1-7228. |
| <Dd>  | Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field. |
| <list> | The vectors containing the table. It must be one of:  
|       | {<Dn>}  Encoded as len = 0b00.  
|       | {<Dn>, <Dn+1>}Encoded as len = 0b01.  
|       | {<Dn>, <Dn+1>, <Dn+2>}Encoded as len = 0b10.  
|       | {<Dn>, <Dn+1>, <Dn+2>, <Dn+3>}Encoded as len = 0b11. |
| <Dm>  | Is the 64-bit name of the SIMD&FP source register holding the indices, encoded in the "M:Vm" field. |

Operation for all encodings

```c
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();

    // Create 256-bit = 32-byte table variable, with zeros in entries that will not be used.
    table3 = if length == 4 then D[n+3] else Zeros(64);
    table2 = if length >= 3 then D[n+2] else Zeros(64);
    table1 = if length >= 2 then D[n+1] else Zeros(64);
    table = table3 : table2 : table1 : D[n];

    for i = 0 to 7
        index = UInt(Elem[D[m],i,8]);
        if index < 8=length then
```
\[
\text{Elem}[D[d],i,8] = \text{Elem}[\text{table},\text{index},8];
\]
\[
\text{else}
\]
\[
\text{if is_vtbl then}
\]
\[
\text{Elem}[D[d],i,8] = \text{Zeros}(8);
\]
\[
// \text{else Elem}[D[d],i,8] \text{ unchanged}
\]

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Vector Transpose treats the elements of its operand vectors as elements of 2 x 2 matrices, and transposes the matrices.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

The following figure shows the operation of VTRN doubleword operations.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

This instruction is used by the pseudo-instructions VUZP (alias) and VZIP (alias). The pseudo-instruction is never the preferred disassembly.
64-bit SIMD vector variant

Applies when \( Q = 0 \).

\[
\text{VTRN}\{c\}\{q\}.<dt> <Dd>, <Dm>
\]

128-bit SIMD vector variant

Applies when \( Q = 1 \).

\[
\text{VTRN}\{c\}\{q\}.<dt> <Qd>, <Qm>
\]

Decode for all variants of this encoding

\[
\text{if size} = '11' \text{ then UNDEFINED;}
\]
\[
\text{if } Q = '1' \&\& (Vd<0> == '1' || Vm<0> == '1') \text{ then UNDEFINED;}
\]
\[
esize = 8 \ll \text{ UInt (size)}; \quad \text{elements} = 64 \text{ DIV esize;}
\]
\[
d = \text{ UInt (D:Vd)}; \quad m = \text{ UInt (M:Vm)};
\]
\[
\text{regs} = \text{ if } Q = '0' \text{ then } 1 \text{ else } 2;
\]

Assembler symbols

\(<c>\quad \text{For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.}
\]
\(<q>\quad \text{For encoding T1: see Standard assembler syntax fields on page F1-7228.}
\]
\(<dt>\quad \text{Is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:}
\]
\[
8 \quad \text{when size = 00}
\]
\[
16 \quad \text{when size = 01}
\]
\[
32 \quad \text{when size = 10}
\]
\[
\text{The encoding size = 11 is reserved.}
\]
\(<Qd>\quad \text{Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.}
\]
\(<Qm>\quad \text{Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.}
\]
\(<Dd>\quad \text{Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.}
\]
\(<Dm>\quad \text{Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.}
\]

Operation for all encodings

\[
\text{if ConditionPassed()} \quad \text{then}
\]
\[
\quad \text{EncodingSpecificOperations(); CheckAdvSIMDEnabled();}
\]
\[
h = \text{ elements DIV 2;}
\]
\[
\text{for } r = 0 \text{ to } \text{regs-1}
\]
\[
\text{if } d == m \text{ then}
\]
\[
\quad D[d+r] = \text{ bits (64) UNKNOWN;}
\]
\[
\text{else}
\]
\[
\text{for } e = 0 \text{ to } h-1
\]
\[
\quad \text{Elem}[D[d+r],2*e+1,esize] = \text{ Elem}[\text{Din}[m+r],2*e,esize];
\]
\[
\quad \text{Elem}[D[m+r],2*e,esize] = \text{ Elem}[\text{Din}[d+r],2*e+1,esize];
\]
Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.255   VTST

Vector Test Bits takes each element in a vector, and bitwise ANDs it with the corresponding element of a second vector. If the result is not zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit fields.

The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the CPACR, NSACR, and HCPR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

|31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 8 7 6 5 4 3 0 |
|---|---|---|---|---|---|---|---|---|---|
|1|1|1|0|0|D|size|Vn|Vd|1|0|0|0|N|Q|M|1|Vm|

64-bit SIMD vector variant

Applies when Q == 0.

VTST{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VTST{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>

Decode for all variants of this encoding:

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;

T1

|15 14 13|12|11 10 9|8 7 6 5 4 3|0|15 12|11 10 9 8|7 6 5 4 3 0 |
|---|---|---|---|---|---|---|---|---|---|
|1|1|1|0|1|1|1|1|0|D|size|Vn|Vd|1|0|0|0|N|Q|M|1|Vm|

64-bit SIMD vector variant

Applies when Q == 0.

VTST{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VTST{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
**Decide for all variants of this encoding**

if Q == '1' & (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;

**Assembler symbols**

<
For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q>
See Standard assembler syntax fields on page F1-7228.

<dt>
Is the data type for the elements of the operands, encoded in the "size" field. It can have the following values:
8 when size = 00
16 when size = 01
32 when size = 10

<Qd>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn>
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>
Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn>
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm>
Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if !IsZero(Elem[D[n+r],e,esize] AND Elem[D[m+r],e,esize]) then
                Elem[D[d+r],e,esize] = Ones(esize);
            else
                Elem[D[d+r],e,esize] = Zeros(esize);

**Operational information**

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
    — The values of the data supplied in any of its registers.
    — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
    — The values of the data supplied in any of its registers.
    — The values of the NZCV flags.
F6.1.256  VUDOT (by element)

Dot Product index form with unsigned integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

--- Note  ---
ID_ISAR6.DP indicates whether this instruction is supported.

A1

(FEAT_DotProd)

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 0 0 D 1 0</td>
<td>Vn</td>
<td>Vd</td>
</tr>
</tbody>
</table>
```

**64-bit SIMD vector variant**

Applies when $Q = 0$.

VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]

**128-bit SIMD vector variant**

Applies when $Q = 1$.

VUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]

**Decode for all variants of this encoding**

- if !HaveDOTPExt() then UNDEFINED;
- if $Q = '1' \&\& (Vd<0> == '1' | Vn<0> == '1') then UNDEFINED;
- boolean signed = (U=='0');
- integer d = UInt(D:Vd);
- integer n = UInt(N:Vn);
- integer m = UInt(Vm<3:0>);
- integer index = UInt(M);
- integer esize = 32;
- integer regs = if $Q == '1'$ then 2 else 1;

T1

(FEAT_DotProd)

```
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------|-------------------------|
| 1 1 1 1 1 1 1 1 0 0 D 1 0 | Vn | Vd | 1 1 0 1 N Q M 1 | Vm |
```

**64-bit SIMD vector variant**

Applies when $Q = 0$.

VUDOT{<q>}.U8 <Dd>, <Dm>, <Dm>[<index>]
128-bit SIMD vector variant

Applies when Q == 1.

VUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.

<index> Is the element index in the range 0 to 1, encoded in the "M" field.

Operation for all encodings

bits(64) operand1;
bits(64) operand2 = D[m];
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
  operand1 = D[n+r];
  result = D[d+r];
  integer element1, element2;
  for e = 0 to 1
    integer res = 0;
    for i = 0 to 3
      if signed then
        element1 = SInt(Elem[operand1, 4 + e + i, esize DIV 4]);
        element2 = SInt(Elem[operand2, 4 + index + i, esize DIV 4]);
      else
        element1 = UInt(Elem[operand1, 4 + e + i, esize DIV 4]);
        element2 = UInt(Elem[operand2, 4 + index + i, esize DIV 4]);
      res = res + element1 * element2;
      Elem[result, e, esize] = Elem[result, e, esize] + res;
      D[d+r] = result;
**F6.1.257  VUDOT (vector)**

Dot Product vector form with unsigned integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

--- Note ---

ID_ISAR6.DP indicates whether this instruction is supported.

---

**A1**

(FEAT_DotProd)

```plaintext
[31 30 29 28|27 26 25 24|23 22 21 20|19 16|15 12|11 10 9 |8 7 6 5 4 |3 0] 0 1 1 1 1 1 1 0 0 0 1 0 Vn Vd 1 1 1 0 1 N Q M 1 Vm U
```

**64-bit SIMD vector variant**

Applies when Q == 0.

VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>

**128-bit SIMD vector variant**

Applies when Q == 1.

VUDOT{<q>}.U8 <Qd>, <Qn>, <Qm>

**Decode for all variants of this encoding**

```plaintext
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

**T1**

(FEAT_DotProd)

```plaintext
[15 14 13 12|11 10 9 |8 7 6 5 4 |3 0] 0 1 1 1 1 1 1 0 0 0 1 0 Vn Vd 1 1 1 0 1 N Q M 1 Vm U
```

**64-bit SIMD vector variant**

Applies when Q == 0.

VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>
128-bit SIMD vector variant

Applies when Q == 1.

VUDOT{<q>}.U8 <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;

Assembler symbols

See Standard assembler syntax fields on page F1-7228.

<q>
Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qd>
Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm>
Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<N>
Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<M>
Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

bits(64) operand1;
bits(64) operand2;
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = D[m+r];
    result = D[d+r];
    integer element1, element2;
    for e = 0 to 1
        integer res = 0;
        for i = 0 to 3
            if signed then
                element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            else
                element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
        D[d+r] = result;
F6.1.258   VUMMLA

The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of unsigned 8-bit integer values held in the first source vector by the 8x2 matrix of unsigned 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an optional instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

A1

(FEAT-AA32I8MM)

```
|  0 | 1 2 3 4 5 6 7 8 | 9 10 11 12 13 14 15 |
|  0 | 1 2 3 4 5 6 7 8 |

B:  Vd  Vn  D  1 1 1 1 0
U:  M  N  1 1 1 0 0
```

A1 variant

VUMMLA{<q>}.U8 <Qd>, <Qn>, <Qm>

Decode for this encoding

```java
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
case B:U of
  when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
  when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
  when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
```

T1

(FEAT-AA32I8MM)

```
|  0 | 1 2 3 4 5 6 7 8 | 9 10 11 12 13 14 15 |
|  0 | 1 2 3 4 5 6 7 8 |

B:  Vd  Vn  D  1 1 1 1 0
U:  M  N  1 1 1 0 0
```

T1 variant

VUMMLA{<q>}.U8 <Qd>, <Qn>, <Qm>

Decode for this encoding

```java
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
case B:U of
  when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
  when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
  when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
  when '11' UNDEFINED;
```
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.
<Qd> Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation for all encodings

CheckAdvSIMDEnabled();
b=128 operan1 = Q[n>>1];
b=128 operan2 = Q[m>>1];
b=128 addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operan1, operan2, op1_unsigned, op2_unsigned);
F6.1.259  VUSDOT (by element)

Dot Product index form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

A1

(FEAT_AA32I8MM)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 16 15 12 11 10 9 8 7 6 5 4 3 0 |
| 1 1 1 1 1 1 1 0 1 | D | 0 | 0 | Vn | Vd | 1 | 1 | 0 | 1 | N | Q | M | 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VUSDOT(<q>).S8 <Dd>, <Dn>, <Dm>[<index>]

128-bit SIMD vector variant

Applies when Q == 1.

VUSDOT(<q>).S8 <Qd>, <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vs<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;

T1

(FEAT_AA32I8MM)

| 15 14 13 12 11 10 9 8 7 6 5 4 3 0 | 15 14 13 12 11 10 9 8 7 6 5 4 3 0 |
| 1 1 1 1 1 1 1 0 1 | D | 0 | 0 | Vn | Vd | 1 | 1 | 0 | 1 | N | Q | M | 0 | Vm |

64-bit SIMD vector variant

Applies when Q == 0.

VUSDOT(<q>).S8 <Dd>, <Dn>, <Dm>[<index>]

128-bit SIMD vector variant

Applies when Q == 1.

VUSDOT(<q>).S8 <Qd>, <Qn>, <Dm>[<index>]

111111101
0x0 D 00 Vn Vd 1101 N Q M 0 Vm

1 5 1 4 1 3 1 2 1 1 1 0 9 8 7 6 5 4 3 0
1 5 1 4 1 3 1 2 1 1 1 0 9 8 7 6 5 4 3 0
Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.

<index> Is the element index in the range 0 to 1, encoded in the "M" field.

Operation for all encodings

CheckAdvSIMDEnabled();
bits(64) operand1;
bits(64) operand2;
bits(64) result;
operand2 = Din[m];
for r = 0 to regs-1
    operand1 = Din[n+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(32) res = Elem[result, e, 32];
        for b = 0 to 3
            element1 = Int(Elem[operand1, 4*e+b, 8], op1_unsigned);
            element2 = Int(Elem[operand2, 4*i+b, 8], op2_unsigned);
            res = res + element1 * element2;
            Elem[result, e, 32] = res;
        D[d+r] = result;
F6.1.260 VUSDOT (vector)

Dot Product vector form with mixed-sign integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in the corresponding 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

A1

(FEAT_AA32I8MM)

64-bit SIMD vector variant

Applies when Q == 0.

VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VUSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = if Q == '1' then 2 else 1;

T1

(FEAT_AA32I8MM)

64-bit SIMD vector variant

Applies when Q == 0.

VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>

128-bit SIMD vector variant

Applies when Q == 1.

VUSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>
Decode for all variants of this encoding

if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' \&\& (Vd<0> == '1' \| Vn<0> == '1' \| Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = if Q == '1' then 2 else 1;

Assembler symbols

<q> See Standard assembler syntax fields on page F1-7228.

<Qd> Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qd> Is the 64-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field.

<Qn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Qm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

CheckAdvSIMDEnabled();
bits(64) operand1;
bits(64) operand2;
bits(64) result;
for r = 0 to regs-1
  operand1 = Din[n+r];
  operand2 = Din[m+r];
  result = Din[d+r];
  for e = 0 to 1
    bits(32) res = Elem[result, e, 32];
    for b = 0 to 3
      element1 = UInt(Elem[operand1, 4 * e + b, 8]);
      element2 = SInt(Elem[operand2, 4 * e + b, 8]);
      res = res + element1 * element2;
      Elem[result, e, 32] = res;
  D[d+r] = result;
F6.1.261 VUSMMLA

The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of unsigned 8-bit integer values held in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an optional instruction. ID_ISAR6.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

A1

(FEAT_AA32I8MM)

| 31 30 29 28 | 27 26 25 24 | 23 22 21 20 | 19 | 16 | 15 | 12 | 11 10 9 8 | 7 6 5 4 | 3 | 0 |
|-------------|-------------|-------------|----|----|----|----|----------|--------|----|----|----|
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | D | 1 | 0 | Vn | Vd | 1 | 1 | 0 | 0 | N | 1 | M | 0 | Vm |

A1 variant

VUSMMLA(<q>).S8 <Qd>, <Qn>, <Qm>

Decode for this encoding

if !HaveAArch32Int8MatMulExt() then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
case B:U of
when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
when '01' op1_unsigned = TRUE; op2_unsigned = TRUE;
when '10' op1_unsigned = TRUE; op2_unsigned = FALSE;
when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);

T1

(FEAT_AA32I8MM)

| 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 | 0 | 15 | 12 | 11 10 9 8 | 7 6 5 4 | 3 | 0 |
|-------------|----------|----------|----|----|----------|--------|----------|----------|----|----|----|
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | D | 1 | 0 | Vn | Vd | 1 | 1 | 0 | 0 | N | 1 | M | 0 | Vm |

T1 variant

VUSMMLA(<q>).S8 <Qd>, <Qn>, <Qm>

Decode for this encoding

if !InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
case B:U of
when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
when '01' op1_unsigned = TRUE; op2_unsigned = TRUE;
when '10' op1_unsigned = TRUE; op2_unsigned = FALSE;
when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);

**Assembler symbols**

<q> See Standard assembler syntax fields on page F1-7228.
<Qd> Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

**Operation for all encodings**

CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
F6.1.262  **VUZP**

Vector Unzip de-interleaves the elements of two vectors.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

The following figure shows the operation of VUZP doubleword operation for data type 8.

**VUZP.8, doubleword**

<table>
<thead>
<tr>
<th>Register state before operation</th>
<th>Register state after operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dd A7 A6 A5 A4 A3 A2 A1 A0</td>
<td>Dm B7 B6 B5 B4 B3 B2 B1 B0</td>
</tr>
</tbody>
</table>

The following figure shows the operation of VUZP quadword operation for data type 32.

**VUZP.32, quadword**

<table>
<thead>
<tr>
<th>Register state before operation</th>
<th>Register state after operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qd A3 A2 A1 A0</td>
<td>Qm B3 B2 B1 B0</td>
</tr>
</tbody>
</table>

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support* on page G1-8996.

**64-bit SIMD vector variant**

Applies when Q == 0.

VUZP{<c>}{<q>}.<dt> <Dd>, <Dm>

**128-bit SIMD vector variant**

Applies when Q == 1.

VUZP{<c>}{<q>}.<dt> <Qd>, <Qm>

**Decode for all variants of this encoding**

if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);

**T1**

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>D 1 1 size 1 0</td>
<td>D 1 1 size 1 0</td>
</tr>
<tr>
<td>Vd 0 0 0 1 0 Q M 0</td>
<td>Vm 0 0 0 1 0 Q M 0</td>
</tr>
</tbody>
</table>

**64-bit SIMD vector variant**

Applies when Q == 0.

VUZP{<c>}{<q>}.<dt> <Dd>, <Dm>
128-bit SIMD vector variant

Applies when Q == 1.

VUZP{<c>}{<q>}{<dt>}{<Qd}, {<Qm>}

Decide for all variants of this encoding

if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);

Assembler symbols

<c> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

<q> See Standard assembler syntax fields on page F1-7228.

<dt> For the 64-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:
8 when size = 00
16 when size = 01
The encoding size = 1x is reserved.
For the 128-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:
8 when size = 00
16 when size = 01
32 when size = 10
The encoding size = 11 is reserved.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Db> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Db> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
  if quadword_operation then
    if d == m then
      Q[d>>1] = bits(128) UNKNOWN;
    else
      zipped_q = Q[m>>1]:Q[d>>1];
      for e = 0 to (128 DIV esize) - 1
        Elem[Q[d>>1],e,esize] = Elem[zipped_q,2*e,esize];
        Elem[Q[m>>1],e,esize] = Elem[zipped_q,2*e+1,esize];
    else
      if d == m then
        D[d] = bits(64) UNKNOWN;
      else
        zipped_d = D[m]:D[d];
        for e = 0 to (64 DIV esize) - 1

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
F6.1.263 VUZP (alias)

Vector Unzip de-interleaves the elements of two vectors

This instruction is a pseudo-instruction of the VTRN instruction. This means that:

- The encodings in this description are named to match the encodings of VTRN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VTRN gives the operational pseudocode for this instruction.

A1

```
<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 1 1</td>
<td>D 1 1</td>
<td>size 1 0</td>
<td>Vd 0 0 0 1</td>
<td>0 M 0</td>
<td>Vm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

64-bit SIMD vector variant

VUZP{<c>}{<q>}.32 <Dd>, <Dm>

is equivalent to

VTRN{<c>}{<q>}.32 <Dd>, <Dm>

and is never the preferred disassembly.

T1

```
<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
<th>15 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>D 1 1</td>
<td>size 1 0</td>
<td>Vd 0 0 0 1</td>
<td>0 M 0</td>
<td>Vm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

64-bit SIMD vector variant

VUZP{<c>}{<q>}.32 <Dd>, <Dm>

is equivalent to

VTRN{<c>}{<q>}.32 <Dd>, <Dm>

and is never the preferred disassembly.

Assembler symbols

- `<c>` For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
  
  For encoding T1: see Standard assembler syntax fields on page F1-7228.

- `<q>` See Standard assembler syntax fields on page F1-7228.

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

- `<Dm>` Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

The description of VTRN gives the operational pseudocode for this instruction.
F6.1.264  VZIP

Vector Zip interleaves the elements of two vectors.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

The following figure shows the operation of VZIP doubleword operation for data type 8.

VZIP.8, doubleword

<table>
<thead>
<tr>
<th>Register state before operation</th>
<th>Register state after operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dd A7 A6 A5 A4 A3 A2 A1 A0</td>
<td>B3 A3 B2 A2 B1 A1 B0 A0</td>
</tr>
<tr>
<td>Dm B7 B6 B5 B4 B3 B2 B1 B0</td>
<td>A7 A6 B6 A6 B5 A5 B4 A4</td>
</tr>
</tbody>
</table>

The following figure shows the operation of VZIP quadword operation for data type 32.

VZIP.32, quadword

<table>
<thead>
<tr>
<th>Register state before operation</th>
<th>Register state after operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qd A3 A2 A1 A0</td>
<td>B1 A1 B0 A0</td>
</tr>
<tr>
<td>Qm B3 B2 B1 B0</td>
<td>A3 A2 A2 A2</td>
</tr>
</tbody>
</table>

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see Enabling Advanced SIMD and floating-point support on page G1-8996.

A1

64-bit SIMD vector variant

Applies when \( Q = 0 \).

VZIP{<c>}{<q>}{<dt> <Dd}, <Dm>

128-bit SIMD vector variant

Applies when \( Q = 1 \).

VZIP{<c>}{<q>}{<dt> <Qd}, <Qm>

Decode for all variants of this encoding

\[
\text{if size} == '11' \| (Q == '0' \&\& size == '10') \text{ then UNDEFINED;} \\
\text{if Q == '1' \&\& (Vd<0> == '1' \| Vm<0> == '1') then UNDEFINED;} \\
\text{quadword_operation} = (Q == '1'); \quad \text{esize} = 8 << \text{UInt(size)}; \\
d = \text{UInt}(D:Vd); \quad m = \text{UInt}(M:Vm);
\]

T1

64-bit SIMD vector variant

Applies when \( Q = 0 \).

VZIP{<c>}{<q>}{<dt> <Dd}, <Dm>
128-bit SIMD vector variant

Applies when $Q == 1$.

$VZIP\{<c>\}{<q>}.<dt> <Qd>, <Qm>$

Decode for all variants of this encoding

if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);

Assembler symbols

$<c>$  For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
For encoding T1: see Standard assembler syntax fields on page F1-7228.

$<q>$  See Standard assembler syntax fields on page F1-7228.

$<dt>$  For the 64-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:
8 when size = 00
16 when size = 01
The encoding size = 1x is reserved.
For the 128-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in the "size" field. It can have the following values:
8 when size = 00
16 when size = 01
32 when size = 10
The encoding size = 11 is reserved.

$<Qd>$  Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as $<Qd>*2.$

$<Qm>$  Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as $<Qm>*2.$

$<Dd>$  Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

$<Dm>$  Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation for all encodings

if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();
if quadword_operation then
  if d == m then
    $Q[d>1] = bits(128) UNKNOWN;$
  else
    bits(256) zipped_q;
    for e = 0 to (128 DIV esize) - 1
      Elem[zipped_q,2*e,esize] = Elem[Q[d>1],e,esize];
      Elem[zipped_q,2*e+1,esize] = Elem[Q[m>1],e,esize];
    $Q[d>1] = zipped_q<127:0>;  Q[m>1] = zipped_q<255:128>$;
  else
    if d == m then
      $D[d] = bits(64) UNKNOWN;$
    else
      bits(128) zipped_d;
      for e = 0 to (64 DIV esize) - 1
Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
F6.1.265   **VZIP (alias)**

Vector Zip interleaves the elements of two vectors

This instruction is a pseudo-instruction of the VTRN instruction. This means that:

- The encodings in this description are named to match the encodings of VTRN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VTRN gives the operational pseudocode for this instruction.

**A1**

\[
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 12 11 10 9 8 7 6 5 4 3 0 |
|-----------------|-----------------|-----------------|
| 1 1 1 1 0 0 1 1 0 1 1 \textcolor{gray}{D} 1 1 | \textcolor{gray}{size} 1 0 \textcolor{gray}{Vd} 0 0 0 0 1 0 \textcolor{gray}{M} 0 | \textcolor{gray}{Vm} 0 |

**64-bit SIMD vector variant**

VZIP\{<c>\}{<q>} .32 \textcolor{gray}{<Dd>}, \textcolor{gray}{<Dm>}

is equivalent to

VTRN\{<c>\}{<q>} .32 \textcolor{gray}{<Dd>}, \textcolor{gray}{<Dm>}

and is never the preferred disassembly.

**T1**

\[
| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 12 11 10 9 8 7 6 5 4 3 0 |
|-----------------|-----------------|-----------------|
| 1 1 1 1 1 1 1 1 1 1 1 \textcolor{gray}{D} 1 1 | \textcolor{gray}{size} 1 0 \textcolor{gray}{Vd} 0 0 0 0 1 0 \textcolor{gray}{M} 0 | \textcolor{gray}{Vm} 0 |

**64-bit SIMD vector variant**

VZIP\{<c>\}{<q>} .32 \textcolor{gray}{<Dd>}, \textcolor{gray}{<Dm>}

is equivalent to

VTRN\{<c>\}{<q>} .32 \textcolor{gray}{<Dd>}, \textcolor{gray}{<Dm>}

and is never the preferred disassembly.

**Assembler symbols**

- <c> For encoding A1: see Standard assembler syntax fields on page F1-7228. This encoding must be unconditional.
- <q> For encoding T1: see Standard assembler syntax fields on page F1-7228.
- \textcolor{gray}{<Dd>} See Standard assembler syntax fields on page F1-7228.
- \textcolor{gray}{<Dm>} Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- \textcolor{gray}{<M>} Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

**Operation for all encodings**

The description of VTRN gives the operational pseudocode for this instruction.
F6.1 Alphabetical list of Advanced SIMD and floating-point instructions
Part G
The AArch32 System Level Architecture
Chapter G1
The AArch32 System Level Programmers’ Model

This chapter gives a system level description of the programmers’ model for execution in AArch32 state. It contains the following sections:

- About the AArch32 System level programmers’ model on page G1-8898.
- Exception levels on page G1-8899.
- Exception terminology on page G1-8900.
- Execution state on page G1-8902.
- Instruction Set state on page G1-8904.
- Security state on page G1-8905.
- Security state, Exception levels, and AArch32 execution privilege on page G1-8908.
- Virtualization on page G1-8910.
- AArch32 state PE modes, and general-purpose and Special-purpose registers on page G1-8912.
- Process state, PSTATE on page G1-8921.
- Instruction set states on page G1-8927.
- Handling exceptions that are taken to an Exception level using AArch32 on page G1-8929.
- Routing of aborts taken to AArch32 state on page G1-8948.
- Exception return to an Exception level using AArch32 on page G1-8951.
- Asynchronous exception behavior for exceptions taken from AArch32 state on page G1-8956.
- AArch32 state exception descriptions on page G1-8964.
- Reset into AArch32 state on page G1-8986.
- Mechanisms for entering a low-power state on page G1-8988.
- The AArch32 System register interface on page G1-8993.
- Advanced SIMD and floating-point support on page G1-8996.
- Configurable instruction enables and disables, and trap controls on page G1-9001.
G1.1 About the AArch32 System level programmers’ model

An application programmer has only a restricted view of the system. The System level programmers’ model supports this application level view of the system, and includes features that are required for one or both of an operating system (OS) and a hypervisor to provide the programming environment seen by an application. This chapter describes the System level programmers’ model when executing at EL1 or higher in an Exception level that is using AArch32.

The system level programmers’ model includes all of the system features required to support operating systems and to handle hardware events.

The following sections give a system level introduction to the basic concepts of the Arm architecture AArch32 state, and the terminology that is used for describing the architecture when executing in this state:

- Exception levels on page G1-8899.
- Exception terminology on page G1-8900.
- Execution state on page G1-8902.
- Instruction Set state on page G1-8904.
- Security state on page G1-8905.
- Virtualization on page G1-8910.

The rest of this chapter describes the system level programmers’ model when executing in AArch32 state.

The other chapters in this part describe:

- The memory system architecture, as seen when executing in an Exception level that is using AArch32:
  - Chapter G4 The AArch32 System Level Memory Model describes the general features of the Armv8 memory model, when executing in AArch32 state, that are not visible at the application level.
  - Note Chapter E2 The AArch32 Application Level Memory Model describes the application level view of the memory model.
  - Chapter G5 The AArch32 Virtual Memory System Architecture describes the Virtual Memory System Architecture (VMSA) used in AArch32 state.

- The AArch32 System registers, see Chapter G8 AArch32 System Register Descriptions.
  - Note The T32 and A32 instruction sets include instructions that provide system level functionality, such as returning from an exception. See, for example, ERET on page F5-7570.
G1.2 Exception levels

The architecture defines a set of Exception levels, EL0 to EL3, where:

- If ELn is the Exception level, increased values of n indicate increased software execution privilege.
- Execution at EL0 is called unprivileged execution.
- EL2 provides support for virtualization.
- EL3 provides support for switching between two Security states, Secure state and Non-secure state.

An implementation might not include all of the Exception levels. All implementations must include EL0 and EL1. EL2 and EL3 are optional.

Note

A PE is not required to implement a contiguous set of Exception levels. For example, it is permissible for an implementation to include only EL0, EL1, and EL3.

Effect of not implementing an Exception level on page D1-4633 provides information on implementations.

When executing in AArch32 state, execution can move between Exception levels only on taking an exception or on returning from an exception:

- On taking an exception, the Exception level can only increase or remain the same.
- On returning from an exception, the Exception level can only decrease or remain the same.

The Exception level that execution changes to or remains in on taking an exception is called the target Exception level of the exception.

Each exception type has a target Exception level that is either:

- Implicit in the nature of the exception.
- Defined by configuration bits in the System registers.

An exception cannot target EL0.

Exception levels exist within Security states. The Armv8-A security model on page G1-8905 describes this. When executing at an Exception level, the PE can access both of the following:

- The resources that are available for the combination of the current Exception level and the current Security state.
- The resources that are available at all lower Exception levels, provided that those resources are available to the current Security state.

This means that if the implementation includes EL3, then because EL3 is only implemented in Secure state, execution at EL3 can access all resources available at all Exception levels, for both Security states.

Each Exception level other than EL0 has its own translation regime and associated control registers. For information on the translation regimes, see Chapter G5 The AArch32 Virtual Memory System Architecture.

G1.2.1 Typical Exception level usage model

The architecture does not specify what software uses which Exception level. Such choices are outside the scope of the architecture. However, the following is a common usage model for the Exception levels:

- **EL0** Applications.
- **EL1** OS kernel and associated functions that are typically described as privileged.
- **EL2** Hypervisor.
- **EL3** Secure monitor.
G1.3 Exception terminology

The following subsections define the terms that are used when describing exceptions:

• Terminology for taking an exception.
• Terminology for returning from an exception.
• Exception levels.
• Definition of a precise exception.
• Definitions of synchronous and asynchronous exceptions on page G1-8901.

G1.3.1 Terminology for taking an exception

An exception is generated when the PE first responds to an exceptional condition. The PE state at this time is the state that the exception is taken from. The PE state immediately after taking the exception is the state that the exception is taken to.

G1.3.2 Terminology for returning from an exception

To return from an exception, the PE must execute an exception return instruction. The PE state when an exception return instruction is committed for execution is the state the exception returns from. The PE state immediately after the execution of that instruction is the state that the exception returns to.

G1.3.3 Exception levels

An Exception level, ELn, with a larger value of n than another Exception level, is described as being a higher Exception level than the other Exception level. For example, EL3 is a higher Exception level than EL1.

An Exception level with a smaller value of n than another Exception level is described as being a lower Exception level than the other Exception level. For example, EL0 is a lower Exception level than EL1.

An Exception level is described as:

• Using AArch64 when execution in that Exception level is in the AArch64 Execution state.
• Using AArch32 when execution in that Exception level is in the AArch32 Execution state.

G1.3.4 Definition of a precise exception

An exception is described as precise when the exception handler receives the PE state and memory system state that is consistent with the PE having executed all of the instructions up to but not including the point in the instruction stream where the exception was taken, and none afterwards.

An exception is described as imprecise if it is not precise.

Other than the SError interrupt all exceptions that are taken to AArch32 state are required to be precise. For each occurrence of an SError interrupt, whether the interrupt is precise or imprecise is IMPLEMENTATION DEFINED.

The terms precise and imprecise can also apply to Debug entry state. See Imprecise entry to Debug state on page H2-10236.

Where a synchronous exception that is taken to AArch32 state is generated as part of an instruction that performs more than one single-copy atomic memory access, the definition of precise permits that the values in registers or memory affected by those instructions can be UNKNOWN, provided that:

• The accesses affecting those registers or memory locations do not, themselves, generate exceptions.
• The registers are not involved in the calculation of the memory address that is used by the instruction.

In AArch32 state, examples of instructions that perform more than one single-copy atomic memory access are the LDM and STM instructions.

Note

For the definition of a single-copy atomic access, see Properties of single-copy atomic accesses on page E2-7165.
G1.3.5 Definitions of synchronous and asynchronous exceptions

An exception is described as *synchronous* if all of the following apply:

- The exception is generated as a result of direct execution or attempted execution of an instruction.
- The return address presented to the exception handler is guaranteed to indicate the instruction that caused the exception.
- The exception is precise.

An exception is described as *asynchronous* if any of the following apply:

- The exception is not generated as a result of direct execution or attempted execution of the instruction stream.
- The return address presented to the exception handler is not guaranteed to indicate the instruction that caused the exception.
- The exception is imprecise.

For more information about exceptions, see *Handling exceptions that are taken to an Exception level using AArch32* on page G1-8929.
G1.4 Execution state

The Execution states are:

- **AArch64**: The 64-bit Execution state.
- **AArch32**: The 32-bit Execution state.

*Execution state on page A1-39* gives more information about them.

Exception levels *use* Execution states. For example, EL0, EL1 and EL2 might all be using AArch32, under EL3 using AArch64.

This means that:

- Different software layers, such as an application, an operating system kernel, and a hypervisor, executing at different Exception levels, can execute in different Execution states.

- The PE can change Execution states only either:
  - At reset.
  - On a change of Exception level.

--- Note ---

- *Typical Exception level usage model on page G1-8899* shows which Exception levels different software layers might typically use.

- *Effect of not implementing an Exception level on page D1-4633* gives information on supported configurations of Exception levels and Execution states.

The interaction between the AArch64 and AArch32 Execution states is called *interprocessing*. *Interprocessing on page D1-4684* describes this.

G1.4.1 About the AArch32 PE modes

AArch32 state provides a set of *PE modes* that support normal software execution and handle exceptions. The current mode determines the set of registers that are available, as described in *AArch32 general-purpose registers, the PC, and the Special-purpose registers on page G1-8917*.

The AArch32 modes are:

- Monitor mode. This mode always executes at Secure EL3.
- Hyp mode. This mode always executes at Non-secure EL2.
- System, Supervisor,Abort, Undefined, IRQ, and FIQ modes. The Exception level these modes execute at depends on the Security state, as described in *Security state on page G1-8905*.
- User mode. This mode always executes at EL0.

--- Note ---

AArch64 state does not support modes. Modes are a concept that is specific to AArch32 state. Modes that execute at a particular Exception level are only implemented if that Exception level supports using AArch32 state.

---

For more information on modes, see *AArch32 state PE mode descriptions on page G1-8912*.

The mode in use immediately before an exception is taken is described as the mode the exception is *taken from*. The mode that is used on taking the exception is described as the mode the exception is *taken to*.

All of the following define the mode that an exception is taken to:

- The type of exception.
- The mode the exception is taken from.
- Configuration settings defined at EL2 and EL3.
Monitor mode and Hyp mode can create system traps that cause exceptions to EL3 or EL2 respectively. There is an architected hierarchy where EL2 and EL3 configuration settings affect a common condition, for example interrupt routing. When no traps are enabled for a particular condition, the AArch32 mode an exception is taken to is called the default mode for that exception.

In AArch32 state, a number of different modes can exist at the same Exception level. All modes at a particular Exception level have the same execution privilege, meaning they have the same access rights for accesses to memory and to System registers. However, the mapping of PE modes to Exception levels depends on the Security state, as described in Security state on page G1-8905. Security state, Exception levels, and AArch32 execution privilege on page G1-8908 gives more information about the PE modes, their associated execution privilege, and how this maps onto the Exception levels.
G1.5 Instruction Set state

In AArch32 state, the Instruction Set state determines the instruction set that the PE is executing. In an implementation that follows the Arm recommendations, the available Instruction Set states are:

- **T32 state** The PE is executing T32 instructions.
- **A32 state** The PE is executing A32 instructions.

--- Note ---

In previous versions of the Arm architecture:
- The T32 instruction set was called the Thumb instruction set.
- The A32 instruction set was called the ARM instruction set.

---

For more information, see Process state, PSTATE on page E1-7133.
G1.6 Security state

The architecture provides two Security states, each with an associated physical memory address space, as follows:

**Secure state**
When in this state, the PE can access both the Secure physical address space and the Non-secure physical address space.

**Non-secure state**
When in this state, the PE:
- Can access only the Non-secure physical address space.
- Cannot access the Secure system control resources.

For information on how virtual addresses translate onto Secure physical and Non-secure addresses, see About VMSAv8-32 on page G5-9146.

G1.6.1 The Armv8-A security model

The principles of the Armv8-A security model are defined in Security state on page D1-4632.

The AArch32 security model, and execution privilege

The Exception level hierarchy of four Exception levels, EL0, EL1, EL2, and EL3, applies to execution in both Execution states. This section describes the mapping between Exception levels, AArch32 modes, and execution privilege.

The AArch32 modes Monitor, System, Supervisor, Abort, Undefined, IRQ, and FIQ all have the same execution privilege.

In Secure state:
- Monitor mode executes only at EL3, and is accessible only when EL3 is using AArch32.
- System mode, Supervisor mode, Abort mode, Undefined mode, IRQ mode, and FIQ mode all:
  - Execute at EL1 when EL3 is using AArch64.
  - Execute at EL3 when EL3 is using AArch32.

This means that there is a difference in the Secure state hierarchy that the PE is using, depending on which Execution state EL3 is using:

- If EL3 is using AArch64:
  - There is no support for Monitor mode.
  - If EL1 is using AArch32, System mode, Supervisor mode, Abort mode, Undefined mode, IRQ mode, and FIQ mode execute at Secure EL1.

- If EL3 is using AArch32:
  - Monitor mode is supported, and executes at Secure EL3.
  - System mode, Supervisor mode, Abort mode, Undefined mode, IRQ mode, and FIQ mode execute at Secure EL3.
  - There is no support for a Secure EL1 Exception level.

See Security behavior in Exception levels using AArch32 when EL2 or EL3 are using AArch64 on page G1-8941 for more information about operation in a Secure EL1 mode when EL3 is using AArch64.

In Non-secure state, the PL1 modes System, Supervisor, Abort, Undefined, IRQ, and FIQ always execute at EL1.

User mode always executes at EL0 and has the lowest possible execution privilege.

Hyp mode always executes in Non-secure state at EL2 and has higher execution privilege than all of:
- User mode.
- System mode, Supervisor mode, Abort mode, Undefined mode, IRQ mode, and FIQ mode.
Limited use of Privilege level in AArch32 state on page G1-8909 describes how, in some contexts, the concept of Privilege levels can be used to represent the execution privilege hierarchy.

For more information about the modes, see About the AArch32 PE modes on page G1-8902.

Figure G1-1 shows the security model when EL3 is using AArch32, and shows the expected use of the different Exception levels, and which modes execute at which Exception levels.

Figure G1-1 shows that when EL3 is using AArch32, the Exception levels and modes available in each Security state are as follows:

**Secure state**
- **EL0**: User mode.
- **EL3**: Any mode that is available in Secure state, other than User mode.

**Non-secure state**
- **EL0**: User mode.
- **EL1**: Any mode that is available in Non-secure state, other than Hyp mode and User mode.
- **EL2**: Hyp mode.

Execution at EL0 is described as unprivileged execution.

A mode associated with a particular Exception level, ELn, is described as an ELn mode.
The Exception level defines the ability to access resources in the current Security state, and does not imply anything about the ability to access resources in the other Security state.

When EL3 is using AArch32, many AArch32 System registers accessible at PL1 are *banked* between the Secure and Non-secure states.

When EL3 is using AArch64 and Secure EL1 is using AArch32, System registers accessible at PL1 are not banked between the Non-secure and Secure states. Software running at EL3 is expected to switch the content of the PL1-accessible System registers between the Secure and Non-secure context, in a similar manner to switching the contents of general purpose registers. For information on the relationship between AArch64 and AArch32 System registers in an interprocessing environment, see *Mapping of the System registers between the Execution states* on page D1-4686.

For more information on the System registers, see *The AArch32 System register interface* on page G1-8993.

The *Secure Monitor Call (SMC)* instruction provides software with a system call to EL3. When executing at a privileged Exception level, SMC instructions generates exceptions. For more information, see *Secure Monitor Call (SMC) exception* on page G1-8969 and *SMC* on page F5-7900.

Changing from Secure state to Non-secure state

Monitor mode is provided to support switching between Secure and Non-secure states. When executing in an Exception level that is using AArch32, except in Monitor mode and Hyp mode, the Security state is controlled:

- By the SCR.NS bit, when EL3 is using AArch32.
- By the SCR_EL3.NS bit, when EL3 is using AArch64.

The mapping of AArch32 privileged modes to the exception hierarchy means that it is possible when EL3 is using AArch32 to change from EL3 to Non-secure EL1 without an exception return. This can occur in one of the following ways:

- Using an MSR or CPS instruction to switch from Monitor mode to another privileged mode while SCR.NS is 1.
- Using an MCR instruction that writes SCR.NS to change from Secure to Non-secure state when in a privileged mode other than Monitor mode.

Arm strongly recommends that software executing at EL3 using AArch32 does not use either of these mechanisms to change from EL3 to Non-secure EL1 without an exception return. The use of both of these mechanisms is deprecated.
G1.7 Security state, Exception levels, and AArch32 execution privilege

The hierarchy of software execution privilege, within a particular Security state, is defined by the Exception levels, with higher Exception level numbers indicating higher privilege. Table G1-1 on page G1-8908 shows this hierarchy for each Security state.

Table G1-1 Execution privilege and Exception levels, by Security state

<table>
<thead>
<tr>
<th>Execution privilege</th>
<th>Secure state</th>
<th>Non-secure state</th>
<th>Typical use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>EL3</td>
<td>-a</td>
<td>Secure monitor</td>
</tr>
<tr>
<td>-</td>
<td>EL2b</td>
<td>EL2</td>
<td>Hypervisor</td>
</tr>
<tr>
<td>-</td>
<td>EL1</td>
<td>EL1</td>
<td>Secure or Non-secure OS</td>
</tr>
<tr>
<td>Lowest, Unprivileged</td>
<td>EL0</td>
<td>EL0</td>
<td>Secure or Non-secure application</td>
</tr>
</tbody>
</table>

a. EL3 is never implemented in Non-secure state.

b. If FEAT_SEL2 is implemented in AArch64 state, EL2 can be enabled in Secure state.

When executing in AArch32 state, within a given Security state, the current PE state, including the execution privilege, is primarily indicated by the current PE mode. In Secure state, how the PE modes map onto the Exception levels depends on whether EL3 is using AArch32 or is using AArch64, and:

- Figure G1-1 on page G1-8906 shows this mapping when EL3 is using AArch32.
- Figure G1-2 on page G1-8915 shows this mapping when EL3 is using AArch64.

Table G1-2 on page G1-8908 shows this mapping. In interpreting this table:

- Monitor mode is implemented only in Secure state, and only if EL3 is using AArch32.
- Hyp mode is implemented only in Non-secure state, and only if EL2 is using AArch32.
- System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented:
  - In Secure state if either:
    - EL3 is using AArch32.
    - EL3 is using AArch64 and EL1 is using AArch32.
  - In Non-secure state if EL1 is using AArch32.
- User mode is implemented if EL0 is using AArch32.

Table G1-2 Mapping of AArch32 PE modes to Exception levels

<table>
<thead>
<tr>
<th>Exception level</th>
<th>PE modes in the given Security state, and EL3 Execution state</th>
<th>Secure state, EL3 using AArch32</th>
<th>Secure state, EL3 using AArch64a</th>
<th>Non-secure state</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL3</td>
<td>Monitor, System, FIQ, IRQ, Supervisor, Abort, Undefined</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>EL2</td>
<td>-</td>
<td>-</td>
<td>Hyp</td>
<td></td>
</tr>
<tr>
<td>EL1</td>
<td>-</td>
<td>System, FIQ, IRQ, Supervisor, Abort, Undefined</td>
<td>System, FIQ, IRQ, Supervisor, Abort, Undefined</td>
<td></td>
</tr>
<tr>
<td>EL0</td>
<td>User</td>
<td>User</td>
<td>User</td>
<td></td>
</tr>
</tbody>
</table>

a. If FEAT_SEL2 is implemented and enabled in AArch64 State, this column can be applied to EL2.

Because AArch32 behavior is described in terms of the PE modes, and transitions between PE modes, the Exception levels are implicit in most of the description of operation in AArch32 state.
G1.7.1  Limited use of Privilege level in AArch32 state

As described in The VMSAv8-32 translation regimes on page G5-9148, a translation regime maps a virtual address (VA) to the corresponding physical address (PA). The VMSAv8-64 translation regimes are defined by the Exception levels that use them. However, because the mapping between PE modes and Exception levels in Secure state depends on whether EL3 is using AArch32 or is using AArch64, as shown in Table G1-2 on page G1-8908, the VMSAv8-32 translation regimes cannot be described simply in terms of either the Exception levels or the PE modes that use them.

To provide a consistent description of address translation as seen from AArch32 state, the VMSAv8-32 translation regimes are described in terms of the Privilege levels originally defined in the Armv7 descriptions of AArch32 state. Table G1-3 on page G1-8909 shows how the PE modes map to these Privilege levels:

Table G1-3 Mapping of PE modes to AArch32 Privilege levels

<table>
<thead>
<tr>
<th>Privilege level</th>
<th>Secure state</th>
<th>Non-secure state</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL2</td>
<td>-</td>
<td>Hyp(^a)</td>
</tr>
<tr>
<td>PL1</td>
<td>Monitor(^b), System, FIQ, IRQ, Supervisor, Abort, Undefined</td>
<td>System, FIQ, IRQ, Supervisor, Abort, Undefined</td>
</tr>
<tr>
<td>PL0</td>
<td>User</td>
<td>User</td>
</tr>
</tbody>
</table>

\(^a\) Implemented only in Non-secure state, and only if EL2 is using AArch32 state.

\(^b\) Implemented only in Secure state, and only if EL3 is using AArch32 state.

Comparing Table G1-3 on page G1-8909 with Table G1-2 on page G1-8908 shows that:

In Non-secure state

Each privilege level maps to the corresponding Exception level. For example, PL1 maps to EL1.

In Secure state

PL0 maps to EL0.

The mapping of PL1 depends on the Execution state being used by EL3, as follows:

- **EL3 using AArch64**  Secure PL1 maps to Secure EL1. Monitor mode is not implemented.
- **EL3 using AArch32**  Secure PL1 maps to Secure EL3. Monitor mode is implemented as one of the Secure PL1 modes.
G1.8 Virtualization

The support for virtualization described in this section applies only to an implementation that includes EL2. A PE is in Hyp mode when it is executing at EL2 in the AArch32 state. An exception return from Hyp mode to software running at EL1 or EL0 is performed using the ERET instruction.

EL2 provides a set of features that support virtualizing the Non-secure state of an A-profile implementation. The basic model of a virtualized system involves:

- A hypervisor, running in EL2, that is responsible for switching between virtual machines. A virtual machine is comprised of Non-secure EL1 and Non-secure EL0.
- A number of Guest operating systems, that each run in Non-secure EL1, on a virtual machine.
- For each Guest operating system, applications, that usually run in Non-secure EL0, on a virtual machine.

Note

In some systems, a Guest OS is unaware that it is running on a virtual machine, and is unaware of any other Guest OS. In other systems, a hypervisor makes the Guest OS aware of these facts. The architecture supports both of these models.

The hypervisor assigns a VMID to each virtual machine.

In AArch32 state, EL2 is implemented only in Non-secure state, to support Guest OS management. EL2 provides controls to:

- Provide virtual values for the contents of a small number of identification registers. A read of one of these registers by a Guest OS or the applications for a Guest OS returns the virtual value.
- Trap various operations, including memory management operations and accesses to many other registers. A trapped operation generates an exception that is taken to EL2.
- Route interrupts to the appropriate one of:
  - The current Guest OS.
  - A Guest OS that is not currently running.
  - The hypervisor.

In Non-secure state:

- The implementation provides an independent translation regime for memory accesses from EL2.
- For the PL1&0 translation regime, address translation occurs in two stages:
  - Stage 1 maps the virtual address (VA) to an intermediate physical address (IPA). This is managed at EL1, usually by a Guest OS. The Guest OS believes that the IPA is the physical address (PA).
  - Stage 2 maps the IPA to the PA. This is managed at EL2. The Guest OS might be completely unaware of this stage.

For more information on the translation regimes, see Chapter G5 The AArch32 Virtual Memory System Architecture.

G1.8.1 The effect of implementing EL2 on the Exception model

An implementation that includes EL2 implements the following exceptions:

- Hypervisor Call (HVC) exception.
- Traps to EL2. EL2 configurable controls on page G1-9010, describes these.
- All of the virtual interrupts:
  - Virtual SError.
  - Virtual IRQ.
  - Virtual FIQ.

HVC exceptions are always taken to EL2. All virtual interrupts are always taken to EL1, and can only be taken from Non-secure EL1 or EL0.
Each of the virtual interrupts can be independently enabled using controls at EL2.

Each of the virtual interrupts has a corresponding physical interrupt. See Virtual interrupts.

When a virtual interrupt is enabled, its corresponding physical exception is taken to EL2, unless EL3 has configured that physical exception to be taken to EL3. For more information, see Asynchronous exception behavior for exceptions taken from AArch32 state on page G1-8956.

An implementation that includes EL2 also:

- Provides controls that can be used to route some synchronous exceptions, taken from Non-secure state, to EL2. For more information, see:
  - Routing exceptions from Non-secure EL0 to EL2 on page G1-8944.
  - Routing debug exceptions to EL2 using AArch32 on page G1-8946.
  - Routing of aborts taken to AArch32 state on page G1-8948

- Provides mechanisms to trap PE operations to EL2. See EL2 configurable controls on page G1-9010.

When an operation is trapped to EL2, the hypervisor typically either:
  - Emulates the required operation. The application running in the Guest OS is unaware of the trap.
  - Returns an error to the Guest OS.

### Virtual interrupts

The virtual interrupts have names that correspond to the physical interrupts, as shown in Table G1-4.

<table>
<thead>
<tr>
<th>Physical interrupt</th>
<th>Corresponding virtual interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>External SError</td>
<td>Virtual SError</td>
</tr>
<tr>
<td>IRQ</td>
<td>Virtual IRQ</td>
</tr>
<tr>
<td>FIQ</td>
<td>Virtual FIQ</td>
</tr>
</tbody>
</table>

Software executing at EL2 can use virtual interrupts to signal physical interrupts to Non-secure EL1 and Non-secure EL0. Example G1-1 shows a usage model for virtual interrupts.

#### Example G1-1 Virtual interrupt usage model

A usage model is as follows:

1. Software executing at EL2 routes a physical interrupt to EL2.

2. When a physical interrupt of that type occurs, the exception handler executing in EL2 determines whether the interrupt can be handled in EL2 or requires routing to a Guest OS in EL1. If the interrupt requires routing to a Guest OS:
   - If the Guest OS is currently running, the hypervisor uses the appropriate virtual interrupt type to signal the physical interrupt to the Guest OS.
   - If the Guest OS is not currently running, the physical interrupt is marked as pending for the guest OS. When the hypervisor next switches to the virtual machine that is running that Guest OS, the hypervisor uses the appropriate virtual interrupt type to signal the physical interrupt to the Guest OS.

Non-secure EL1 and Non-secure EL0 modes cannot distinguish a virtual interrupt from the corresponding physical interrupt.

For more information, see Virtual exceptions when an implementation includes EL2 on page G1-8956.
G1.9 AArch32 state PE modes, and general-purpose and Special-purpose registers

The following sections describe the AArch32 PE modes and the general-purpose registers and the PC:

- AArch32 state PE mode descriptions.
- AArch32 general-purpose registers, the PC, and the Special-purpose registers on page G1-8917.
- Saved Program Status Registers (SPSRs) on page G1-8919.
- ELR_hyp on page G1-8920.

Note

The PC is included in the scope of this section because, in AArch32 state, it is defined as being part of the same register file as the general-purpose registers. That is, the AArch32 register file R0-R15 comprises:

- The general-purpose registers R0-R14.
- The PC, which can be described as R15.

G1.9.1 AArch32 state PE mode descriptions

Table G1-5 on page G1-8912 shows the PE modes defined by the Arm architecture, for execution in AArch32 state. In this table:

- The PE mode column gives the name of each mode and the abbreviation used, for example, in the general-purpose register name suffixes used in AArch32 general-purpose registers, the PC, and the Special-purpose registers on page G1-8917.
- The Encoding column gives the corresponding PSTATE.M field.
- The Exception level column gives the Exception level at which the mode is implemented, including dependencies on the current Security state and on whether EL3 is using AArch32, see Exception levels on page G1-8899.

<table>
<thead>
<tr>
<th>PE mode</th>
<th>Encoding</th>
<th>Security state</th>
<th>Exception level</th>
<th>Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>usr</td>
<td>10000</td>
<td>Both</td>
<td>EL0</td>
</tr>
<tr>
<td>FIQ</td>
<td>fiq</td>
<td>10001</td>
<td>Non-secure</td>
<td>EL1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Secure</td>
<td>EL1 or EL3^a</td>
</tr>
<tr>
<td>IRQ</td>
<td>irq</td>
<td>10010</td>
<td>Non-secure</td>
<td>EL1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Secure</td>
<td>EL1 or EL3^a</td>
</tr>
<tr>
<td>Supervisor</td>
<td>svc</td>
<td>10011</td>
<td>Non-secure</td>
<td>EL1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Secure</td>
<td>EL1 or EL3^a</td>
</tr>
<tr>
<td>Monitor</td>
<td>mon</td>
<td>10110</td>
<td>Secure</td>
<td>EL3</td>
</tr>
<tr>
<td>Abort</td>
<td>abt</td>
<td>10111</td>
<td>Non-secure</td>
<td>EL1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Secure</td>
<td>EL1 or EL3^a</td>
</tr>
<tr>
<td>Hyp</td>
<td>hyp</td>
<td>11010</td>
<td>Non-secure</td>
<td>EL2</td>
</tr>
<tr>
<td>Undefined</td>
<td>und</td>
<td>11011</td>
<td>Non-secure</td>
<td>EL1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Secure</td>
<td>EL1 or EL3^a</td>
</tr>
<tr>
<td>System</td>
<td>sys</td>
<td>11111</td>
<td>Non-secure</td>
<td>EL1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Secure</td>
<td>EL1 or EL3^a</td>
</tr>
</tbody>
</table>

^a. EL3 if EL3 is using AArch32. EL1 if EL3 is using AArch64 and EL1 is using AArch32.
FEAT_SEL2 is not supported if EL2 is using AArch32.

Mode changes can be made under software control, or can be caused by an external or internal exception.

Notes on the AArch32 PE modes

PE modes are defined only in AArch32 state. Because each mode is implemented as part of a particular Exception level that is using AArch32, the set of available modes depends on which Exception levels are implemented and using AArch32, as described in Effect of the EL3 Execution state on the PE modes and Exception levels on page G1-8914.

This section gives more information about each of the modes, when it is implemented.

User mode
Software executing in User mode executes at EL0. Execution in User mode is sometimes described as unprivileged execution. Application programs normally execute in User mode, and any program executed in User mode:
- Makes only unprivileged accesses to system resources, meaning it cannot access protected system resources.
- Makes only unprivileged access to memory.
- Cannot change mode except by causing an exception, see Handling exceptions that are taken to an Exception level using AArch32 on page G1-8929.

System mode
System mode is implemented at EL1 or EL3, see Effect of the EL3 Execution state on the PE modes and Exception levels on page G1-8914.
System mode has the same registers available as User mode, and is not entered by any exception.

Supervisor mode
Supervisor mode is implemented at EL1 or EL3, see Effect of the EL3 Execution state on the PE modes and Exception levels on page G1-8914.
Supervisor mode is the default mode to which a Supervisor Call exception is taken. Executing an SVC (Supervisor Call) instruction generates a Supervisor Call exception.
In an implementation where the highest implemented Exception level is using AArch32, if that Exception level is EL3 or EL1, a PE enters Supervisor mode on reset.

Abort mode
Abort mode is implemented at EL1 or EL3, see Effect of the EL3 Execution state on the PE modes and Exception levels on page G1-8914.
Abort mode is the default mode to which a Data Abort exception or Prefetch Abort exception is taken.

Undefined mode
Undefined mode is implemented at EL1 or EL3, see Effect of the EL3 Execution state on the PE modes and Exception levels on page G1-8914.
Undefined mode is the default mode to which an instruction-related exception, including any attempt to execute an UNDEFINED instruction, is taken.

FIQ mode
FIQ mode is implemented at EL1 or EL3, see Effect of the EL3 Execution state on the PE modes and Exception levels on page G1-8914.
FIQ mode is the default mode to which an FIQ interrupt is taken.

IRQ mode
IRQ mode is implemented at EL1 or EL3, see Effect of the EL3 Execution state on the PE modes and Exception levels on page G1-8914.
IRQ mode is the default mode to which an IRQ interrupt is taken.

Hyp mode
Hyp mode is the Non-secure EL2 mode.
Hyp mode is entered on taking an exception from Non-secure state that must be taken to EL2.
In an implementation where the highest implemented Exception level is EL2 and EL2 uses AArch32 on reset, a PE enters Hyp mode on reset.

The Hypervisor Call exception and Hyp Trap exception are implemented as part of EL2 and are always taken to Hyp mode when EL2 is using AArch32.

Executing an HVC (Hypervisor Call) instruction generates a Hypervisor Call exception. See Hypervisor Call (HVC) exception on page G1-8970.

For more information, see Hyp mode on page G1-8915.

Monitor mode

Monitor mode is the Secure EL3 mode. This means it is always in the Secure state, regardless of the value of the SCR.NS bit.

Monitor mode is the mode to which a Secure Monitor Call exception is taken. In a Non-secure EL1 mode, or a Secure EL3 mode, executing an SMC (Secure Monitor Call) instruction generates a Secure Monitor Call exception.

When EL3 is using AArch32, some exceptions that are taken to a different mode by default can be configured to be taken to EL3, see PE mode for taking exceptions on page G1-8939.

When EL3 is using AArch32, software executing in Monitor mode:
• Has access to both the Secure and Non-secure copies of System registers.
• Can perform an exception return to Secure state, or to Non-secure state.

This means that, when EL3 is using AArch32, Monitor mode provides the only recommended method of changing between the Secure and Non-secure Security states.

Secure and Non-secure modes

In an implementation that includes EL3, the names of most implemented modes can be qualified as Secure or Non-secure, to indicate whether the PE is also in Secure state or Non-secure state. For example:
• If a PE is in Supervisor mode and Secure state, it is in Secure Supervisor mode.
• If a PE is in User mode and Non-secure state, it is in Non-secure User mode.

Note

As indicated in the appropriate Mode descriptions:
• Monitor mode is a Secure mode, meaning it is always in the Secure state.
• Hyp mode is a Non-secure mode, meaning it is accessible only in Non-secure state.

Effect of the EL3 Execution state on the PE modes and Exception levels

Figure G1-1 on page G1-8906 shows the PE modes, Exception levels, and Security states, for an implementation that includes all of the Exception levels, when EL3 is using AArch32. Figure G1-2 on page G1-8915 shows how the implemented modes change when EL3 is using AArch64.
Comparing Figure G1-1 on page G1-8906 and Figure G1-2 shows how, in Secure state only, the implementation of System, FIQ, IRQ, Supervisor, Abort, and Undefined mode depends on the Execution state that EL3 is using. That is, these modes are implemented as follows:

**Non-secure state**

- If Non-secure EL1 is using AArch32, then System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented as part of EL1. Otherwise, these modes are not implemented in Non-secure state.

**Secure state**

- The implementation of these modes depends on the Execution state that EL3 is using, as follows:
  - **EL3 using AArch64** If Secure EL1 is using AArch32, then System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented as part of EL1. Otherwise, these modes are not implemented in Secure state.
  - **EL3 using AArch32** In Secure state, System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented as part of EL3, see Figure G1-1 on page G1-8906.

**Hyp mode**

Hyp mode is the Non-secure EL2 mode. When EL2 is using AArch32, it provides the usual method of controlling the virtualization of Non-secure execution at EL1 and EL0.

---

**Note**

The alternative method of controlling this functionality is by accessing the EL2 controls from EL3 with the SCR_EL3.NS or SCR.NS bit set to 1.
This section summarizes how Hyp mode differs from the other modes, and references where this part of the manual describes the features of Hyp mode in more detail:

- Software executing in Hyp mode executes at EL2, see Figure G1-1 on page G1-8906.
- Hyp mode is accessible only in Non-secure state. In Secure state, an attempt by a CPS or an MSR instruction to change PSTATE.M to Hyp mode is an illegal change to PSTATE.M, as described in Illegal changes to PSTATE.M on page G1-8925.
- In Non-debug state, the only mechanisms for changing to Hyp mode are:
  - An exception taken from a Non-secure EL1 or EL0 mode.
  - When EL3 is using AArch32, an exception return from Secure Monitor mode.
  - When EL3 is using AArch64, an exception return from EL3.
- In Hyp mode, the only exception return is execution of an ERET instruction, see ERET on page F5-7570.
- In Hyp mode, the CPACR has no effect on the execution of:
  - System register access instructions.
  - Advanced SIMD and floating-point instructions.
  - The HCPTR controls execution of these instructions in Hyp mode.
- If software running in Hyp mode executes an SVC instruction, the Supervisor Call exception generated by the instruction is taken to Hyp mode, see SVC on page F5-8055.
- An exception return with restored PSTATE specifying Hyp mode is an illegal return event, as described in Illegal return events from AArch32 state on page G1-8952, if any of the following applies:
  - EL3 is using AArch64 and the value of SCR_EL3.NS is 0.
  - EL3 is using AArch32 and the value of SCR.NS is 0.
  - The return is from a Non-secure EL1 mode.
- The instructions described in the following sections are UNDEFINED if executed in Hyp mode:
  - SRS. See SRS, SRSDA, SRSDB, SRSIA, SRSIB on page F5-7936.
  - RFE. See RFE, RFEDA, RFEDB, RFEIA, RFEIB on page F5-7830.
  - LDM (exception return) on page F5-7604.
  - LDM (User registers) on page F5-7606.
  - STM (User registers) on page F5-7976.
  - The SUBS PC, LR forms of the instructions described in SUB, SUBS (immediate) on page F5-8039.

  **Note**

  In T32 state, ERET is encoded as SUBS PC, LR, #0, and therefore this is a valid instruction.

  The exception return form of the instructions described in MOV, MOVS (register) on page F5-7719.

  In addition, deprecated forms of the A32 ADCS, ADDS, ANDS, BICS, EORS, MOVs, WNS, ORRS, RSBS, RCS, SBCS, and SUBS instructions with the PC as the destination register are UNDEFINED if executed in Hyp mode. The instruction descriptions identify these UNDEFINED cases.

- The Load unprivileged and Store unprivileged instructions LDRT, LDRSH, LDRHT, LDRBT, STRT, STRHT, and STRBT, are CONSTRAINED UNPREDICTABLE if executed in Hyp mode.

To permit entry to Hyp mode using the Hypervisor Call exception, Secure software must enable use of the HVC instruction:

- By setting the SCR_EL3.HCE bit to 1, if EL3 is using AArch64.
- By setting the SCR.HCE bit to 1, if EL3 is using AArch32.

If EL3 is implemented and using AArch32, and SCR.HCE is set to 0, the HVC instruction is UNPREDICTABLE in Hyp mode. The instruction is either UNDEFINED or executes as a NOP.
If EL3 is implemented and using AArch64, and SCR_EL3.HCE is set to 0, the HVC instruction is UNDEFINED in Hyp mode.

If EL3 is not implemented and HCR.HCD is set to 1, the HVC instruction is UNDEFINED in Hyp mode.

**Pseudocode description of mode operations**

The `BadMode()` function tests whether a 5-bit mode number corresponds to one of the permitted modes.

The `BadMode()` function is defined in Chapter J1 *Armv8 Pseudocode*.

**G1.9.2 AArch32 general-purpose registers, the PC, and the Special-purpose registers**

The general-purpose registers, and the PC, in AArch32 state on page E1-7131 describes the application level view of the general-purpose registers, and the PC. This view provides:

- The general-purpose registers R0-R14, of which:
  - The preferred name for R13 is SP (stack pointer).
  - The preferred name for R14 is LR (link register).
- The PC, which can be described as R15.

These registers are selected from a larger set of registers that includes banked copies of some registers, with the current register selected by the execution mode. The implementation and banking of the general-purpose registers depends on whether or not the implementation includes EL2 and EL3, and whether those Exception levels are using AArch32. Figure G1-3 on page G1-8918 shows the full set of banked general-purpose registers, and the Special-purpose registers:

- The Program Status Registers CPSR and SPSR.
- ELR_hyp.

*Note*

The architecture uses system level register names, such as R0_usr, R8_usr, and R8_fiq, when it must identify a specific register. The application level names refer to the registers for the current mode, and usually are sufficient to identify a register.
As described in PE mode for taking exceptions on page G1-8939, on taking an exception the PE changes mode, unless it is already in the mode to which it must take the exception. Each mode that the PE might enter in this way has:

- A banked copy of the stack pointer, for example SP_irq and SP_hyp.
- A register that holds a preferred return address for the exception. This is:
  - For the EL2 mode, Hyp mode, the Special-purpose register ELR_hyp.
  - For the other privileged modes to which exceptions can be taken, a banked copy of the link register, for example LR_und and LR_mon.
- A saved copy of PSTATE, made on exception entry, for example SPSR_irq and SPSR_hyp.

In addition, FIQ mode has banked copies of the general-purpose registers R8 to R12.

User mode and System mode share the same general-purpose registers.

User mode, System mode, and Hyp mode share the same LR.

For more information about the application level view of the SP, LR, and PC, and the alternative descriptions of them as R13, R14 and R15, see The general-purpose registers, and the PC, in AArch32 state on page E1-7131.

**AArch32 Special-purpose registers**

In AArch32 state, the Special-purpose registers are:

- The CPSR and its view as the APSR.
- The SPSR, including the banked copies SPSR_abt, SPSR_fiq, SPSR_hyp, SPSR_irq, SPSR_mon, SPSR_svc, and SPSR_und.
- The ELR_hyp.
Pseudocode description of general-purpose register and PC operations

The following pseudocode gives access to the general-purpose registers and the PC. These registers are an array, \( _R \), indexed by parameter \( n \). This array is common to AArch32 and AArch64 operation and therefore contains 31 64-bit registers. \( _PC \) is the Program Counter, and its definition is common to AArch32 and AArch64 operation and therefore its size is 64-bit.

\( \text{LookUpRIndex()} \) looks up the index value, \( n \), for the specified register number and PE mode, using \( \text{RBankSelect()} \) to evaluates the result.

\( _R \) accesses the specified general-purpose register in the current PE mode, using \( \text{Rmode[]} \) to access the register, accessing \( _R \) if necessary. \( \text{SP} \) accesses the stack pointer, \( \text{LR} \) accesses the link register, and \( \text{PC} \) accesses the Program Counter. Each function has a non-assignment form for register reads and an assignment form for register writes, other than \( \text{PC} \), which has only a non-assignment form.

\( \text{BranchTo()} \) performs a branch to the specified address.

The \( _R, _PC, \text{LR}, \text{SP}, \text{LookUpRIndex()}, \text{RBankSelect()}, \text{Rmode[]}, \text{and BranchTo()} \) functions are defined in Chapter J1 Armv8 Pseudocode.

G1.9.3 Saved Program Status Registers (SPSRs)

The Saved Program Status Registers (SPSRs) are used to save PE state on taking exceptions. In AArch32 state, there is an SPSR for every mode that an exception can be taken to, as shown in Figure G1-3 on page G1-8918. For example, the SPSR for Monitor mode is called \( \text{SPSR\_mon} \).

--- Note ---

Exceptions cannot be taken to EL0.

---

When the PE takes an exception, PE state is saved from \( \text{PSTATE} \) in the SPSR for the mode the exception is taken to. For example, if the PE takes an exception to Monitor mode, PE state is saved in \( \text{SPSR\_mon} \). For more information on \( \text{PSTATE} \), see Process state, \( \text{PSTATE} \) on page G1-8921.

--- Note ---

All \( \text{PSTATE} \) fields are saved, including those which have no direct read and write access.

---

Saving the \( \text{PSTATE} \) fields means the exception handler can:

- On return from the exception, restore the PE state to the values it had immediately before the exception was taken. When the PE returns from an exception, PE state is restored to the state stored in the SPSR of the mode the exception is returning from, if the exception return is made using one of:
  - \( \text{ERET} \).
  - \( \text{LDM} \).
  - The Exception return form of the instruction described in \( \text{MOV, MOVs (register)} \) on page F5-7719.
  - The Exception return form of the instruction described in \( \text{SUB, SUBS (immediate)} \) on page F5-8039.

For example, on returning from Monitor mode, PE state is restored to the state stored in \( \text{SPSR\_mon} \). If the exception return is made using the RFE instruction, the PE restores the PE state from an SPSR valued read from memory.

- Examine the value that \( \text{PSTATE} \) had when the exception was taken, for example to determine the instruction set state and privilege level in which the instruction that caused an Undefined Instruction exception was executed.

The SPSRs are \text{UNKNOWN} on a Warm reset. Any operation in a Non-secure EL1 or EL0 mode makes \( \text{SPSR\_hyp} \) unknown.

SPSR bits that are defined as \text{RES0} on an exception taken from AArch32 state are ignored on any exception return to AArch32 state.

For more information on SPSR, see SPSR, Saved Program Status Register on page G8-9706.
Pseudocode description of SPSR operations

The following pseudocode gives access to the SPSRs.

The `SPSR[]` function accesses the current SPSR and is common to AArch32 and AArch64 operation.

The `SPSRWriteByInstr()` function is used by the MSR (register) and MSR (immediate) instructions to update the current SPSR.

The `SPSR[]` and `SPSRWriteByInstr()` functions are defined in Chapter J1 Armv8 Pseudocode.

G1.9.4 ELR_hyp

Hyp mode does not provide its own banked copy of LR. Instead, on taking an exception to Hyp mode, the preferred return address is stored in ELR_hyp, a 32-bit Special-purpose register implemented for this purpose.

ELR_hyp can be accessed explicitly only by executing:
• An MRS or MSR instruction that targets ELR_hyp, see:
   — `MRS (Banked register)` on page F5-7736.
   — `MSR (Banked register)` on page F5-7740.

The `ERET` instruction uses the value in ELR_hyp as the return address for the exception. For more information, see `ERET` on page F5-7570.

Software execution in any Non-secure EL1 or EL0 mode makes ELR_hyp UNKNOWN.
G1.10 Process state, PSTATE

Process state or PSTATE is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

PSTATE includes all of the following:
- Fields that are meaningful only in AArch32 state.
- Fields that are meaningful only in AArch64 state.
- Fields that are meaningful in both Execution states.

PSTATE is defined in pseudocode as the `PSTATE` structure, of type `ProcState`. `ProcState` is defined in Chapter J1 *Armv8 Pseudocode*.

The PSTATE fields that are meaningful in AArch32 state are:

**The Condition flags**
- **N** Negative Condition flag.
- **Z** Zero Condition flag.
- **C** Carry Condition flag.
- **V** Overflow Condition flag.

*Process state, PSTATE on page E1-7133* gives more information about these.

**The overflow or saturation flag**
- **Q** See *Process state, PSTATE on page E1-7133*.

**The greater than or equal flags**
- **GE[3:0]** See *Process state, PSTATE on page E1-7133*.

**The PE state controls**
- **J, T** Instruction set state. See *Process state, PSTATE on page E1-7133*. J is RES0. On a Warm reset to AArch32 state, T is set to an IMPLEMENTATION DEFINED value. On taking an exception to:
  - A PL1 mode using AArch32, T is set to SCTLR.TE.
  - EL2 using AArch32, T is set to HSCTLR.TE.
- **IT[7:0]** IT block state bits. See *Process state, PSTATE on page E1-7133*. On a Warm reset or taking an exception to AArch32 state, these bits are set to 0.
- **E** Endianness of data accesses. See *Process state, PSTATE on page E1-7133*. If an implementation provides both Big-endian and Little-endian support, then:
  - On a Warm reset to AArch32 state this bit is set to the IMPLEMENTATION DEFINED reset value of:
    - SCTLR.EE if the highest implemented Exception level is not EL2.
    - HSCTLR.EE if the highest implemented Exception level is EL2.
  - On taking an exception to:
    - A PL1 mode using AArch32, this bit is set to SCTLR.EE.
    - EL2 using AArch32, this bit is set to HSCTLR.EE
- **IL** Illegal Execution state bit. See *The Illegal Execution state exception on page G1-8954*. On a Warm reset or taking an exception to AArch32 state, this bit is set to 0.

For information on how the J, T, IT[7:0], E, and IL fields can be accessed, see *Accessing the PE state controls and the Execution state bit on page G1-8924*.

**The asynchronous exception mask bits**
- **A** SError interrupt mask bit.
- **I** IRQ interrupt mask bit.
- **F** FIQ interrupt mask bit.
For each bit, the values are:

0   Exception not masked.
1   Exception masked.

On a Warm reset to AArch32 state, these bits are set to 1.
On taking an exception to AArch32 state, one or more of these bits are set to 1.
For more information, see both:

• Asynchronous exception masking controls on page G1-8959.
• PE state on exception entry on page G1-8942.

The mode bits

M[4:0]   Current mode of the PE. Table G1-5 on page G1-8912 lists the permitted values of this field. All other values are reserved. Illegal changes to PSTATE.M on page G1-8925 describes the effect of setting M[4:0] to a reserved value.
M[4] is:

M[4], Execution state

The current Execution state:

0   AArch64 state.
1   AArch32 state.

Note

This is consistent with the use of M[4:0] in previous versions of the architecture.

On a Warm reset to AArch32 state, M[4:0] is set to:

• 0b10011, meaning Supervisor mode, if the highest implemented Exception level is not EL2.
• 0b11010, meaning Hyp mode, if the highest implemented Exception level is EL2.

On taking an exception to AArch32 state, M[4:0] is set to the target mode for the exception type.
For more information about the PE modes, see:

• AArch32 state PE mode descriptions on page G1-8912.
• PE state on exception entry on page G1-8942.

Access control bits, from Armv8.1

PAN    Privileged Access Never (PAN) state bit, see About the PAN bit on page G5-9195.

Timing control bits

DIT    Data Independent Timing (DIT) bit. For more information, see About the DIT bit on page E1-7139.
This bit is implemented only when FEAT_DIT is implemented.
On a Warm reset to AArch32 state, this bit is set to 0.

Speculation control bits

SSBS   Speculative Store Bypass Safe (SSBS) bit. For more information, see Speculative Store Bypass Safe (SSBS) on page E2-7179.
This bit is implemented only when FEAT_SSBS is implemented.
On a Warm reset to AArch32 state, this bit is set to an IMPLEMENTATION DEFINED value.

G1.10.1 Accessing PSTATE fields

The PSTATE fields can be accessed as described in the following subsections:

• The Current Program Status Register, CPSR on page G1-8923.
• Accessing the PE state controls and the Execution state bit on page G1-8924.
• The CPS instruction on page G1-8924.
• The SETEND instruction on page G1-8925.
• The SETPAN instruction on page G1-8925.

### The Current Program Status Register, CPSR

Some PSTATE fields can be accessed using the Special-purpose Current Program Status Register (CPSR). The CPSR can be directly read using the MRS instruction, and directly written using the MSR (register) and MSR (immediate) instructions.

The CPSR bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>N</td>
</tr>
<tr>
<td>30</td>
<td>Z</td>
</tr>
<tr>
<td>29</td>
<td>C</td>
</tr>
<tr>
<td>28</td>
<td>V</td>
</tr>
<tr>
<td>27</td>
<td>Q</td>
</tr>
<tr>
<td>26</td>
<td>SSBS</td>
</tr>
<tr>
<td>25</td>
<td>GE[3:0]</td>
</tr>
<tr>
<td>24</td>
<td>E</td>
</tr>
<tr>
<td>23</td>
<td>A</td>
</tr>
<tr>
<td>22</td>
<td>I</td>
</tr>
<tr>
<td>21</td>
<td>F</td>
</tr>
<tr>
<td>20</td>
<td>M[4:0]</td>
</tr>
<tr>
<td>19</td>
<td>DIT</td>
</tr>
<tr>
<td>18</td>
<td>PAN</td>
</tr>
<tr>
<td>17</td>
<td>ENDIANNENESS</td>
</tr>
<tr>
<td>16</td>
<td>MODES</td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
</tr>
<tr>
<td>14</td>
<td>Reserved</td>
</tr>
<tr>
<td>13</td>
<td>Reserved</td>
</tr>
<tr>
<td>12</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>Reserved</td>
</tr>
<tr>
<td>10</td>
<td>Reserved</td>
</tr>
<tr>
<td>9</td>
<td>Reserved</td>
</tr>
<tr>
<td>8</td>
<td>Reserved</td>
</tr>
<tr>
<td>7</td>
<td>Reserved</td>
</tr>
<tr>
<td>6</td>
<td>Reserved</td>
</tr>
<tr>
<td>5</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
</tr>
<tr>
<td>2</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

- **N, Z, C, V, bits [31:28]**
  - The PSTATE Condition flags.
- **Q, bit [27]**
  - The PSTATE overflow or saturation flag.
- **Bits[26:23, 20, 15:10, 5]**
  - Reserved, RES0.
- **SSBS, bit [23]**
  - Speculative Store Bypass Safe (SSBS) bit, see Access permissions for instruction execution on page G5-9196.
- **Bit[22]**
  - In Armv8.0, Reserved, RES0.
  - In Armv8.1, Privileged Access Never (PAN) state bit, see About the PAN bit on page G5-9195.
- **DIT, bit [21]**
  - Shows the value of CPSR.DIT immediately before the exception was taken.
- **GE[3:0], bits [19:16]**
  - The PSTATE greater than or equal flags.
- **E, bit [9]**
  - The PSTATE endianness bit.
- **A, I, F, bits [8:6]**
  - The PSTATE asynchronous exception mask bits.
- **M[4:0], bits [4:0]**
  - The PSTATE mode bits.

The other PSTATE fields cannot be accessed by using the CPSR. For information on how to access them, see Accessing the PE state controls and the Execution state bit on page G1-8924.

The application level alias for the CPSR is the APSR. The APSR is a subset of the CPSR. See The Application Program Status Register, APSR on page E1-7135.

Writes to the CPSR have side-effects on various aspects of PE operation. All of these side-effects, except side-effects on memory accesses associated with fetching instructions, are synchronous to the CPSR write. This means that they are guaranteed:

- Not to be visible to earlier instructions in the execution stream.
- To be visible to later instructions in the execution stream.
The privilege level and address space of memory accesses associated with fetching instructions depend on the current Exception level and Security state. Writes to \texttt{PSTATE.M} can change one or both of the Exception level and Security state. The effect, on memory accesses associated with fetching instructions, of a change of Exception level or Security state is:

- Synchronous to the change of Exception level or Security state, if that change is caused by an exception entry or exception return.
- Guaranteed not to be visible to any memory access caused by fetching an earlier instruction in the execution stream.
- Guaranteed to be visible to any memory access caused by fetching any instruction after the next \textit{Context synchronization event} in the execution stream.
- Might or might not affect memory accesses caused by fetching instructions between the mode change instruction and the point where the mode change is guaranteed to be visible.

See \textit{Exception return to an Exception level using AArch32} on page G1-8951 for the definition of exception return instructions.

### Accessing the PE state controls and the Execution state bit

ThePE state controls are the \texttt{PSTATE.\{IL, IT[7:0], J, E, T\}} fields. Software can read or write these in an SPSR.

In the CPSR:

- The PE state controls, other than \texttt{PSTATE.E}, are RAZ when read by an \texttt{MRS} instruction.
- Writes to the PE state controls, other than \texttt{PSTATE.E}, by \texttt{MSR (register)} or \texttt{MSR (immediate)}, are ignored in all modes.

Instructions other than \texttt{MRS, MSR (register)}, or \texttt{MSR (immediate)} that access the PE state controls can read and write them in any mode.

Unlike the other \texttt{PSTATE} PE state controls, \texttt{PSTATE.E} can be read by an \texttt{MRS} instruction and might be written by \texttt{MSR (register)} or \texttt{MSR (immediate)}. However, Arm deprecates \texttt{PSTATE.E} having a different value from the equivalent System register EE bit, see \textit{Mixed-endian support} on page G4-9112.

#### Note

To determine the current endianness, software can use an \texttt{LDR} instruction to load a word from memory with a known value that differs if the endianness is reversed. For example, using an \texttt{LDR} instruction to load a word whose four bytes are \texttt{0x01, 0x00, 0x00, 0x00} in ascending order of memory address loads the destination register with:

- \texttt{0x00000001} if the current endianness is little-endian.
- \texttt{0x01000000} if the current endianness is big-endian.

The \texttt{PSTATE.M[4]} bit is the Execution state bit. When read by an \texttt{MRS} instruction in AArch32 state, this bit always reads as 1. When written by an \texttt{MSR (register)} instruction or \texttt{MSR (immediate)} instruction, writing a value other than 1 is an illegal change to the \texttt{PSTATE.M} field. See \textit{Illegal changes to PSTATE.M} on page G1-8925.

### The CPS instruction

The A32 and T32 instruction sets both include an instruction to manipulate \texttt{PSTATE.\{A, I, F\}} and \texttt{PSTATE.M}:

\begin{verbatim}
CPSIE <iflags> {, #<mode>}
\end{verbatim}

Sets the specified \texttt{PSTATE.\{A, I, F\}} exception masks to 0, enabling the exception, and optionally changes to the specified mode.

\begin{verbatim}
CPSID <iflags> {, #<mode>}
\end{verbatim}

Sets the specified \texttt{PSTATE.\{A, I, F\}} exception masks to 1, disabling the exception, and optionally changes to the specified mode.

\begin{verbatim}
CPS #<mode>
\end{verbatim}

Changes to the specified mode without affecting the \texttt{PSTATE.\{A, I, F\}} exception masks.
The CPS instruction is unconditional. For more information, see CPS, CPSID, CPSIE on page F5-7535.

The SETEND instruction

The A32 and T32 instruction sets both include an instruction to manipulate PSTATE.E:

SETEND BE  Sets PSTATE.E to 1, for big-endian operation.
SETEND LE  Sets PSTATE.E to 0, for little-endian operation.

The SETEND instruction is unconditional. For more information, see SETEND on page F5-7882. Arm deprecates use of the SETEND instruction.

The SETPAN instruction

FEAT_PAN adds the SETPAN instruction to the A32 and T32 instruction sets, to manipulate PSTATE.PAN:

SETPAN #0  Sets PSTATE.PAN to 0, disabling Privileged access-never operation.
SETPAN #1  Sets PSTATE.PAN to 1, enabling Privileged access-never operation.

The SETPAN instruction is unconditional.

• SETPAN on page F5-7883.
• About the PAN bit on page G5-9195.

G1.10.2 The Saved Program Status Registers (SPSRs)

On taking an exception, PSTATE is preserved in the SPSR of the mode to which the exception is taken. The SPSRs are described in Saved Program Status Registers (SPSRs) on page G1-8919.

G1.10.3 Illegal changes to PSTATE.M

In AArch32 PE modes other than User mode, MSR and CPS instructions can explicitly change PSTATE.M. The following changes to PSTATE.M by MSR or CPS instructions are illegal:

• A change to an encoding that Table G1-5 on page G1-8912 does not show.
• A change to a mode that is not implemented.
• A change to a mode that is not accessible from the context the MSR or CPS instruction is executed in, as follows:
  — A change to a mode that would cause entry to a higher Exception level.
  — When executing in Non-secure state, a change to Monitor mode.
  — When executing in Secure EL1, a change to Monitor mode when EL3 is using AArch64.
  — A change to Hyp mode from any other mode.
  — A change from Hyp mode to any other mode.
  — When the value of HCR.TGE is 1, attempting to change from Monitor mode to a Non-secure PL1 mode, see Trapping of general exceptions to Hyp mode on page K1-11582.

On executing an instruction that attempts an illegal change to PSTATE.M:

• PSTATE.M is unchanged, and the current mode remains unchanged.
• PSTATE.IL is set to 1.
• All other PSTATE fields are written to as normal.

Note

For the PSTATE fields that MSR and CPS instructions update, see the instruction descriptions:

• MSR (register) on page F5-7746.
• MSR (immediate) on page F5-7744.
• CPS, CPSID, CPSIE on page F5-7535.
When the value of `PSTATE.IL` is 1, any attempt to execute any instruction results in an Illegal Execution state exception. See *The Illegal Execution state exception* on page G1-8954.

--- Note ---

The PE ignores writes to `PSTATE.M` when executing at PL0.

### G1.10.4 Pseudocode description of PSTATE operations

The `CPSRWriteByInstr()` function is used by the MSR (register) and MSR (immediate) instructions to update PSTATE.

The `SetPSTATEFromPSR()` function updates PSTATE from a CPSR or SPSR.

*Chapter J1 Armv8 Pseudocode* defines these functions.
G1.11 Instruction set states

The instruction set states are described in Chapter E2 The AArch32 Application Level Memory Model and application level operations on them are described there. This section supplies more information about how they interact with system level functionality, in the sections:

- Exceptions and instruction set state.
- Unimplemented instruction sets.

G1.11.1 Exceptions and instruction set state

If an exception is taken to an EL1 mode, the SCTLR.TE bit for the Security state the exception is taken to determines the instruction set state that handles the exception, and if necessary, the PE changes to this instruction set state on exception entry.

If the exception is taken to Hyp mode, the HSCTLR.TE bit determines the instruction set state that handles the exception, and if necessary, the PE changes to this instruction set state on exception entry.

On coming out of reset, if the highest implemented Exception level is using AArch32:

- If the highest implemented Exception level is EL2, the PE starts execution in Hyp mode, in the instruction set state determined by the reset value of HSCTLR.TE.
- Otherwise, the PE starts execution in Supervisor mode, in the instruction set state determined by the reset value of SCTLR.TE. If the implementation includes EL3, this execution is in Secure Supervisor mode.

For more information about exception entry, see Overview of exception entry on page G1-8936.

G1.11.2 Unimplemented instruction sets

The PSTATE.T bit defines the current instruction set state, see Process state, PSTATE on page E1-7133.

From the introduction of the Armv8 architecture, there is no support for the hardware acceleration of Java bytecodes, and the Jazelle Instruction set state is obsolete. Every AArch32 implementation must support the Trivial Jazelle implementation described in Trivial implementation of the Jazelle extension.

Note

In previous versions of the Arm architecture, the PSTATE.{J, T} bits determined the Instruction set state. From the introduction of Armv8, PSTATE.J is RES0.

Trivial implementation of the Jazelle extension

The implementation of AArch32 state is required to include the trivial Jazelle implementation.

In a trivial implementation of the Jazelle extension:

- At EL1, EL2, or EL3, if the Exception level is using AArch32:
  - The JMCR and JOSCR are RAZ/WI.
  - The JIDR is a RAZ read-only register.
- At EL0 when EL0 is using AArch32:
  - It is IMPLEMENTATION DEFINED whether the JMCR and JOSCR are RAZ/WI or UNDEFINED.
  - It is IMPLEMENTATION DEFINED whether JIDR is RAZ or UNDEFINED.
- The BXJ instruction behaves identically to the BX instruction in all circumstances.

Note

This is consistent with the JMCR.JE bit being RAZ, and means that the A32 and T32 instruction sets do not provide any mechanism for attempting to enter Jazelle state.
Jazelle state, as defined in previous versions of the Arm architecture, is an unimplemented instruction set state.

These requirements ensure that operating systems that support an EJVM execute correctly.

A trivial implementation is not required to extend the PC to 32 bits, that is, it can implement PC[0] as RAZ/WI.

--- Note ---

This is because the only way that PC[0] is visible in A32 or T32 state is as a result of an exception occurring during Jazelle state execution, and Jazelle state execution cannot occur on a trivial implementation.
G1.12 Handling exceptions that are taken to an Exception level using AArch32

An exception causes the PE to suspend program execution to handle an event, such as an externally generated interrupt or an attempt to execute an undefined instruction. Exceptions can be generated by internal and external sources.

Normally, when an exception is taken the PE state is preserved immediately, before handling the exception. This means that, when the event has been handled, the original state can be restored and program execution resumed from the point where the exception was taken.

More than one exception might be generated at the same time, and a new exception can be generated while the PE is handling an exception.

The following sections describe exception handling:

- Exception vectors and the exception base address.
- Exception prioritization for exceptions taken to AArch32 state on page G1-8932.
- Overview of exception entry on page G1-8936.
- PE mode for taking exceptions on page G1-8939.
- PE state on exception entry on page G1-8942.
- Routing exceptions from Non-secure EL0 to EL2 on page G1-8944.
- Routing debug exceptions to EL2 using AArch32 on page G1-8946.

See also:

- Routing of aborts taken to AArch32 state on page G1-8948.
- Exception return to an Exception level using AArch32 on page G1-8951.
- Asynchronous exception behavior for exceptions taken from AArch32 state on page G1-8956.
- AArch32 state exception descriptions on page G1-8964.

G1.12.1 Exception vectors and the exception base address

When an exception is taken, PE execution is forced to an address that corresponds to the type of exception. This address is called the exception vector for that exception. The vectors for the different types of exception form a vector table.

Note

There are significant differences in the sets of exception vectors for exceptions taken to an Exception level that is using AArch32 and for exceptions taken to an Exception level that is using AArch64. This part of this manual describes only how exceptions are taken to an Exception level that is using AArch32.

When an exception is taken to an Exception level that is using AArch64, then the exception is taken as described in Chapter D1 The AArch64 System Level Programmers’ Model using the exception vectors described in Exception vectors on page D1-4640.

AArch32 state defines exception vector tables for exceptions taken to EL2 and EL3 when those Exception levels are using AArch32. Those vector tables are not used when the corresponding Exception levels are using AArch64.

A set of exception vectors for an Exception level that is using AArch32 comprises eight consecutive word-aligned memory addresses, starting at an exception base address. These eight vectors form an AArch32 vector table.

The number of possible exception base addresses, and therefore the number of vector tables, depends on the implemented Exception levels, as follows:

Implementation that does not include EL3

Any implementation that does not include EL3 must include the following AArch32 vector table if EL1 can use AArch32:

- An exception table for exceptions taken to EL1 modes other than System mode. This is the EL1 vector table, and is in the address space of the PL1&0 translation regime.
Note

Exceptions cannot be taken to System mode.

For this vector table:
- When SCTLR.V == 0, the VBAR holds the exception base address.
- When SCTLR.V == 1, the exception base address is 0xFFFF0000.

Implementation that includes EL2

Any implementation that includes EL2 must include the following additional AArch32 vector table if EL2 can use AArch32:

- An exception table for exceptions taken to Hyp mode. This is the Hyp vector table, and is in the address space of the Non-secure PL2 translation regime.
  For this vector table, HVBAR holds the exception base address.

Implementation that includes EL3

Any implementation that includes EL3 must include the following AArch32 vector tables:

- If EL3 can use AArch32, a vector table for exceptions taken to Secure Monitor mode. This is the Monitor vector table, and is in the address space of the Secure PL1&0 translation regime.
  For this vector table, MVBAR holds the exception base address.
- If Secure EL1 can use AArch32, a vector table for exceptions taken to Secure privileged modes other than Monitor mode and System mode. This is the Secure vector table, and is in the address space of the Secure PL1&0 translation regime.
  - When the Secure instance of SCTLR.V == 0, the Secure instance of VBAR holds the exception base address.
  - When the Secure instance of SCTLR.V == 1, the exception base address is 0xFFFF0000.
- If Non-secure EL1 can use AArch32, a vector table for exceptions taken to Non-secure PL1 modes. This is the Non-secure vector table, and is in the address space of the Non-secure PL1&0 translation regime.
  - When the Non-secure instance of SCTLR.V == 0, the Non-secure instance of VBAR holds the exception base address.
  - When the Non-secure instance of SCTLR.V == 1, the exception base address is 0xFFFF0000.

The following subsections give more information:

- The vector tables and exception offsets.
- Pseudocode determination of the exception base address on page G1-8932.

The vector tables and exception offsets

Table G1-6 on page G1-8931 defines the AArch32 vector table entries. In this table:

- The Hyp column defines the vector table entries for exceptions taken to Hyp mode.
- The Monitor column defines the vector table entries for exceptions taken to Monitor mode.
- The Secure and Non-secure columns define the Secure and Non-secure vector table entries, that are used for exceptions taken to modes other than Monitor mode, Hyp mode, System mode, and User mode. Table G1-7 on page G1-8931 shows the mode to which each of these exceptions is taken. Each of these modes is described as the default mode for taking the corresponding exception.

Note

Exceptions cannot be taken to System mode or User mode.
For more information about determining the mode to which an exception is taken, see *PE mode for taking exceptions on page G1-8939*.

When EL2 is using AArch32, it provides a number of additional exceptions, some of which are not shown explicitly in the vector tables. For more information, see *Offsets of AArch32 exceptions provided by EL2 on page G1-8932*.

Table G1-6 The AArch32 vector tables

<table>
<thead>
<tr>
<th>Offset</th>
<th>Hypa</th>
<th>Monitorb</th>
<th>SECUREc</th>
<th>Non-securec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Not used</td>
<td>Not used</td>
<td>Not usedd</td>
<td>Not used</td>
</tr>
<tr>
<td>0x04</td>
<td>Undefined Instruction, from Hyp mode</td>
<td>Monitor Trap</td>
<td>Undefined Instruction</td>
<td>Undefined Instruction</td>
</tr>
<tr>
<td>0x08</td>
<td>Hypervisor Call, from Hyp mode</td>
<td>Secure Monitor Call</td>
<td>Supervisor Call</td>
<td>Supervisor Call</td>
</tr>
<tr>
<td>0x0C</td>
<td>Prefetch Abort, from Hyp mode</td>
<td>Prefetch Abort</td>
<td>Prefetch Abort</td>
<td>Prefetch Abort</td>
</tr>
<tr>
<td>0x10</td>
<td>Data Abort, from Hyp mode</td>
<td>Data Abort</td>
<td>Data Abort</td>
<td>Data Abort</td>
</tr>
<tr>
<td>0x14</td>
<td>Hyp Trap, or Hyp mode entrye</td>
<td>Not used</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>0x18</td>
<td>IRQ interrupt</td>
<td>IRQ interrupt</td>
<td>IRQ interrupt</td>
<td>IRQ interrupt</td>
</tr>
<tr>
<td>0x1C</td>
<td>FIQ interrupt</td>
<td>FIQ interrupt</td>
<td>FIQ interrupt</td>
<td>FIQ interrupt</td>
</tr>
</tbody>
</table>

a. Non-secure state only. Implemented only if the implementation includes EL2 and EL2 can use AArch32.
b. Secure state only. Implemented only if the implementation includes EL3 and EL3 can use AArch32.
c. If the implementation does not include EL3 then there is a single vector table for exceptions taken to EL1 when EL1 is using AArch32. That table holds the vectors shown in the Secure column of this table.
d. In previous versions of the architecture, this entry has been used for the Reset vector, meaning the address at which execution starts on coming out of reset. From the introduction of Armv8, the AArch32 Reset vector is IMPLEMENTATION DEFINED. An implementation might use this vector table entry to hold the Reset vector.
e. See *Use of offset 0x14 in the Hyp vector table on page G1-8932*.

Table G1-7 Modes for taking the exceptions shown in the Secure or Non-secure vector table

<table>
<thead>
<tr>
<th>Exception</th>
<th>Mode taken to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined Instruction</td>
<td>Undefined</td>
</tr>
<tr>
<td>Supervisor Call</td>
<td>Supervisor</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>Abort</td>
</tr>
<tr>
<td>Data Abort</td>
<td>Abort</td>
</tr>
<tr>
<td>IRQ interrupt</td>
<td>IRQ</td>
</tr>
<tr>
<td>FIQ interrupt</td>
<td>FIQ</td>
</tr>
</tbody>
</table>

For more information about use of the vector tables, see *Overview of exception entry on page G1-8936*. 
Offsets of AArch32 exceptions provided by EL2

EL2 provides the following exceptions. When EL2 is using AArch32, these exceptions are taken to Hyp mode, and the PE enters the handlers for these exceptions using the following vector table entries shown in Table G1-6 on page G1-8931:

**Hypervisor Call**  
If taken from Hyp mode, shown explicitly in the Hyp mode vector table. Otherwise, see *Use of offset 0x14 in the Hyp vector table.*

**Hyp Trap**  
Shown explicitly in the Hyp mode vector table.

**Virtual Abort**  
Entered through the Data Abort vector in the Non-secure vector table.

**Virtual IRQ**  
Entered through the IRQ vector in the Non-secure vector table.

**Virtual FIQ**  
Entered through the FIQ vector in the Non-secure vector table.

---  
**Note**  
*Virtual exceptions when an implementation includes EL2 on page G1-8956* gives more information about the virtual exceptions.

---

**Use of offset 0x14 in the Hyp vector table**

The vector at offset 0x14 in the Hyp vector table is used for all exceptions that cause entry to Hyp mode from Non-secure EL0 and EL1, except for IRQ and FIQ exceptions.

---  
**Note**  
Virtual exceptions are never taken to Hyp mode.

---

**Pseudocode determination of the exception base address**

For an exception taken to a PL1 mode, the `ExcVectorBase()` function determines the exception base address. The `ExcVectorBase()` function is defined in Chapter J1 *Armv8 Pseudocode.*

---  
**Note**  
The PL1 modes to which exceptions can be taken are Supervisor mode, Undefined mode, Abort mode, IRQ mode, and FIQ mode. In Non-secure state, and in Secure state when EL3 is using AArch64, these are EL1 modes. However, in Secure state when EL3 is using AArch32, these are EL3 modes. For more information, see *Security state, Exception levels, and AArch32 execution privilege* on page G1-8908.

---

G1.12.2 Exception prioritization for exceptions taken to AArch32 state

The following sections describe the requirements for the prioritization of synchronous exceptions, and the limits on when asynchronous exceptions can be taken:

- *Synchronous exception prioritization for exceptions taken to AArch32 state on page G1-8933.*
- *Architectural requirements for taking asynchronous exceptions on page G1-8935.*

See also:

- *AArch32 state prioritization of synchronous aborts from a single stage of address translation on page G5-9248,* for information about:
  - The prioritization of aborts on a single memory access in a VMSA implementation.
  - The prioritization of exceptions generated during address translation.
- *Debug state entry and debug event prioritization on page H2-10235* for information about the relative prioritization of exceptions and the debug events that cause entry to Debug state.
Synchronous exception prioritization for exceptions taken to AArch32 state

In principle, any single instruction can generate a number of different synchronous exceptions, between the fetching of the instruction, its decode, and eventual execution. This section describes the prioritization of such exceptions when they are taken to an Exception level that is using AArch32.

Note

• An exception that is taken to an Exception level that is using AArch32 must have been taken from an Exception level that is using AArch32.
• The priority numbering in this list correlates with the equivalent AArch64 list in Prioritization of Synchronous exceptions taken to AArch64 state on page D1-4650.

For an exception that is taken to an Exception level that is using AArch32, exceptions are prioritized as follows, where 1 is the highest priority.

1-5 These priority numbers are used by AArch64 exceptions or debug events.

6 PC alignment fault exceptions. A PC alignment fault exception can only be taken to an Exception level that is using AArch32 as a result of:
• The CONstrained UNPREDICTABLE handling of a branch to an unaligned address, see Branching to an unaligned PC on page K1-11564.
• Exiting from Debug state to AArch32 specifying an unaligned PC value, see Exiting Debug state on page H2-10271.

A PC alignment fault exception that is taken to an Exception level that is using AArch32 is reported as a Prefetch Abort exception, see Prefetch Abort exception reporting a PC alignment fault exception on page G1-8972.

7 Prefetch Abort exceptions. See Prefetch Abort exception on page G1-8971 and AArch32 state prioritization of synchronous aborts from a single stage of address translation on page G5-9248.

8 Breakpoint exceptions or Address Matching Vector Catch exceptions. See:
• Breakpoint exceptions on page G2-9054.
• Vector Catch exceptions on page G2-9093.

Note

An Exception Trapping Vector Catch exception is generated on exception entry for an exception that has been prioritized as described in this section. This means that it does not have its own entry in this list.

9 Illegal Execution state exceptions. See The Illegal Execution state exception on page G1-8954.

10 Software Breakpoint Exceptions caused by the execution of a BKPT Exception generating instruction.

11 This priority number is used by AArch64 exceptions.

12 Exceptions taken from EL1 to EL2 because of one of the following configuration settings:
• HSTR.Tn.
• HCR.TIDCP.

13 Undefined Instruction exceptions that occur as a result of one or more of the following:
• An attempt to execute an unallocated instruction encoding, including an encoding for an instruction that is not implemented in the PE implementation.
• An attempt to execute an instruction that is defined never to be accessible at the current Exception level regardless of any enables or traps.
• Debug state execution of an instruction encoding that is not accessible in Debug state.
• Non-debug state execution of an instruction encoding that is not accessible in Non-debug state.
• Execution of an HVC instruction when HVC instructions are disabled by SCR.HCE or HCR.HCD.
• Execution of an HLT instruction when HLT instructions are disabled by EDSCR.HDE or when halting is prohibited.
• When FEAT_FGT and FEAT_PMUv3 are implemented, executing an MSR or MSR instruction in AArch64 state, or an MCR or MRS instruction in AArch32 state, that accesses a register associated with an unimplemented event counter.
• In Debug state:
  — Execution of a DCPS1 instruction in Non-secure EL0 when HCR.TGE is 1.
  — Execution of a DCPS2 instruction in EL1 or EL0 when SCR.NS is 0 or when EL2 is disabled or not implemented in the current Security state.
  — Execution of a DCPS3 instruction when EDSCR.SDD is 1 or when EL3 is not implemented.
  — When the value of EDSCR.SDD is 1, execution in EL2, EL1, or EL0 of an instruction that is trapped to EL3.
• Execution of an instruction that is UNDEFINED as a result of any of:
  — Being in an IT block when SCTLR.ITD is 1, or when HSCTLR.ITD is 1.
  — Executing a SETEND instruction when SCTLR.SED is 1, or when HSCTLR.SED is 1.
  — Executing a CP15DMB, CP15DSB, or CP15ISB barrier instruction when SCTLR.CP15BEN is 0, or when HSCTLR.CP15BEN is 0.

See Disabling or enabling PL0 and PL1 use of AArch32 optional functionality on page G1-9004 and Disabling or enabling EL2 use of AArch32 optional functionality on page G1-9013.
• Execution of an instruction that is UNDEFINED because at least one of FPSCR.\{Stride, Len\} is nonzero, when programming these bits to nonzero values is supported. See Floating-point exceptions and exception traps on page E1-7148.

14 Exceptions taken to EL1, or taken to EL2 because the value of HCR.TGE is 1, that are generated because of configurable access to instructions, and that are not covered by any of priorities 6-13.

15 Exceptions taken from EL0 to EL2 because of one of the following configuration settings:
• HSTR.Tn.
• HCR.TIDCP.

16 Exceptions taken to EL2 because of configuration settings in the HCPTR.

17 Exceptions taken to EL2 because of one of the following configuration settings:
• Any setting in HCR, other than the TIDCP bit.
• Any setting in CNTHCTL.
• Any setting in HDCR.
• If EL1 is using AArch64 state, any of the fine-grained traps in HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGITR_EL2, HFGTR_EL2, HFGWTR_EL2.

18 Exceptions taken to EL2 because of configurable access to instructions, and that are not covered by any of priorities 6-17.

19 Exceptions caused by the S\$C instruction being UNDEFINED because the value of SCR.SCD is 1.

20 Exceptions caused by the execution of an Exception generating instruction, SVC, HVC, or S\$C.

21-22 These priority numbers are used by AArch64 exceptions.

23 Exceptions taken to EL3 from EL0, EL1, or EL2 because of configuration settings in SDCR.

24 Exceptions taken to EL3 because of configurable access to instructions, and that are not covered by any of priorities 6-23.
This priority number is used by AArch64 exceptions.

Trapped floating-point exceptions, if supported. See Floating-point exceptions and exception traps on page E1-7148.

These priority numbers are used by AArch64 exceptions and debug events.

Data Abort exceptions other than a Data Abort exception generated by a synchronous External abort that was not generated by a translation table walk. That is, any Data Abort exception that is not covered by priority 33. See Data Abort exception on page G1-8975 and AArch32 state prioritization of synchronous aborts from a single stage of address translation on page G5-9248. It is IMPLEMENTATION DEFINED whether synchronous External aborts are prioritized here or as priority 33.

Watchpoint exceptions. See Watchpoint exceptions on page G2-9079.

Data Abort exception generated by a synchronous External abort that was not generated by a translation table walk, see External aborts on page G4-9139. It is IMPLEMENTATION DEFINED whether synchronous External aborts are prioritized here or as priority 31.

For priorities 31-33, if an instruction results in more than one single-copy atomic memory access, the prioritization between synchronous exceptions generated on each of those different memory accesses is not defined by the architecture.

Note

Exceptions generated by a translation table walk are reported and prioritized as either a Prefetch Abort exception, priority 7 in this list, or a Data Abort exception, priority 31 in this list. See also AArch32 state prioritization of synchronous aborts from a single stage of address translation on page G5-9248.

Architectural requirements for taking asynchronous exceptions

The Arm architecture does not define when asynchronous exceptions are taken. The prioritization of asynchronous exceptions, including virtual asynchronous exceptions, is IMPLEMENTATION DEFINED.

An asynchronous exception that is pending before a Context synchronization event in the following list, is taken before the first instruction after the context synchronizing event, provided that the pending asynchronous event is not masked:

- Execution of an ISB instruction that does not fail its Condition code check.
- Exception entry.
- Exception return.
- Exit from Debug state.

Note

- If the first instruction after the context synchronizing event generates a synchronous exception, then the architecture does not define the order in which that synchronous exception and the asynchronous exception are taken.
- The ISR identifies any pending asynchronous exceptions.
- Interrupts are masked when the PE is in Debug state, and therefore this list of context synchronizing events does not include the DCPS and DRPS instructions.

In the absence of a specific requirement to take an asynchronous exception, the only requirement of the architecture is that an unmasked asynchronous exception is taken in finite time.

Note

The taking of an unmasked asynchronous exception in finite time must occur with all code sequences, including with a sequence that consists of unconditional loops.
If an unmasked interrupt was pending but is changed to not pending before it is taken, then the architecture permits
the interrupt to be taken, but does not require this to happen. If the interrupt is taken, then it must be taken before
the first Context synchronization event after the interrupt was changed to not pending.

**PSTATE** includes a mask bit for each type of asynchronous exception. Setting one of these bits to 1 can prevent the
corresponding asynchronous exception from being taken, although when the PE is in Non-secure state other controls
can modify the effect of these bits. For more information, see Asynchronous exception behavior for exceptions taken
from AArch32 state on page G1-8956.

Taking an exception sets an exception-dependent subset of these mask bits.

**Note**

In some contexts, the **PSTATE.**{A, I, F} bits mask the taking of asynchronous exceptions. The way these are set on
exception entry, described in **PSTATE.**{A, I, F, M} values on exception entry on page G1-8943, can prevent an
exception handler being interrupted by an asynchronous exception.

**G1.12.3 Overview of exception entry**

There are some significant differences between the handling of exceptions taken to Hyp mode and exceptions taken
to other modes. Because Hyp mode is the EL2 mode, this means that the following descriptions sometimes
distinguish between the **EL2 mode** and the **non-EL2 modes**.

On taking an exception to an Exception level that is using AArch32:

1. The hardware determines the mode to which the exception must be taken, see PE mode for taking exceptions
   on page G1-8939.

2. A link value, indicating the [preferred return address](#) for the exception, is saved. This is a possible return
   address for the exception handler, and depends on:
   - The exception type.
   - Whether the exception is taken to the EL2 mode or to a non-EL2 mode.
   - For some exceptions taken to non-EL2 modes, the instruction set state when the exception was taken.
   Where the link value is saved depends on whether the exception is taken to the EL2 mode.
   For more information, see Link values saved on exception entry on page G1-8937.

3. The value of **PSTATE** is saved in the **SPSR** for the mode to which the exception must be taken. The value
   saved in **SPSR.IT[7:0]** is always correct for the preferred return address.

4. In an implementation that includes EL3, when EL3 is using AArch32:
   - If the exception is taken from Monitor mode, **SCR.NS** is cleared to 0.
   - Otherwise, taking the exception leaves **SCR.NS** unchanged.
   When EL3 is using AArch64, Monitor mode is not available.

5. **PSTATE** is updated with new context information for the exception handler. This includes:
   - Setting **PSTATE.M** to the PE mode to which the exception is taken.
   - Setting the appropriate **PSTATE** mask bits. This can disable the corresponding exceptions, preventing
     uncontrolled nesting of exception handlers.
   - Setting the instruction set state to the state required for exception entry.
   - Setting the endianness to the required value for exception entry.
   - Clearing the **PSTATE.IT[7:0]** bits to 0.
   For more information, see PE state on exception entry on page G1-8942.

6. The appropriate exception vector is loaded into the PC, see Exception vectors and the exception base address
   on page G1-8929.

7. Execution continues from the address held in the PC.
For an exception taken to a non-EL2 mode, on exception entry, the exception handler can use the SRS instruction to store the return state onto the stack of any mode at the same Exception level and in the same Security state, and can use the CPS instruction to change mode. For more information about the instructions, see SRS, SRSDA, SRSDB, SRSIA, SRSIB on page F5-7936 and CPS, CPSID, CPSIE on page F5-7535.

Later sections of this chapter describe each of the possible exceptions, and each of these descriptions includes a pseudocode description of the PE state changes on taking that exception. Table G1-8 on page G1-8937 gives an index to these descriptions:

### Table G1-8 Pseudocode descriptions of exception entry for exceptions taken to AArch32 state

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description of exception entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Pseudocode descriptions of reset on page G1-8987</td>
</tr>
<tr>
<td>Undefined Instruction</td>
<td>Pseudocode description of taking the Undefined Instruction exception on page G1-8966</td>
</tr>
<tr>
<td>Hyp Trap</td>
<td>Pseudocode description of taking the Hyp Trap exception on page G1-8968</td>
</tr>
<tr>
<td>Monitor Trap</td>
<td>Pseudocode description of taking the Monitor Trap exception on page G1-8967</td>
</tr>
<tr>
<td>Supervisor Call</td>
<td>Pseudocode description of taking the Supervisor Call exception on page G1-8969</td>
</tr>
<tr>
<td>Secure Monitor Call</td>
<td>Pseudocode description of taking the Secure Monitor Call exception on page G1-8970</td>
</tr>
<tr>
<td>Hypervisor Call</td>
<td>Pseudocode description of taking the Hypervisor Call exception on page G1-8971</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>Pseudocode description of taking the Prefetch Abort exception on page G1-8975</td>
</tr>
<tr>
<td>Data Abort</td>
<td>Pseudocode description of taking the Data Abort exception on page G1-8978</td>
</tr>
<tr>
<td>Virtual Abort</td>
<td>Pseudocode description of taking the Virtual SError interrupt exception on page G1-8980</td>
</tr>
<tr>
<td>IRQ</td>
<td>Pseudocode description of taking the physical IRQ exception on page G1-8981</td>
</tr>
<tr>
<td>Virtual IRQ</td>
<td>Pseudocode description of taking the Virtual IRQ exception on page G1-8982</td>
</tr>
<tr>
<td>FIQ</td>
<td>Pseudocode description of taking the FIQ exception on page G1-8984</td>
</tr>
<tr>
<td>Virtual FIQ</td>
<td>Pseudocode description of taking the Virtual FIQ exception on page G1-8984</td>
</tr>
</tbody>
</table>

The following sections give more information about the PE state changes, for different architecture implementations. However, you must refer to the pseudocode for a full description of the state changes:

- **PE mode for taking exceptions** on page G1-8939.
- **PE state on exception entry** on page G1-8942.

### Link values saved on exception entry

On exception entry, a link value for use on return from the exception, is saved. This link value is based on the preferred return address for the exception, as shown in Table G1-9 on page G1-8937:

### Table G1-9 Exception return addresses for exceptions taken to AArch32 state

<table>
<thead>
<tr>
<th>Exception</th>
<th>Preferred return address</th>
<th>Taken to a mode at</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined Instruction</td>
<td>Address of the UNDEFINED instruction</td>
<td>Non-EL2&lt;sup&gt;a&lt;/sup&gt;, or EL2&lt;sup&gt;c&lt;/sup&gt;</td>
</tr>
<tr>
<td>Hyp Trap</td>
<td>Address of the trapped instruction</td>
<td>EL2 only&lt;sup&gt;c&lt;/sup&gt;</td>
</tr>
<tr>
<td>Monitor Trap</td>
<td>Address of the trapped instruction</td>
<td>EL3 only</td>
</tr>
<tr>
<td>Supervisor Call</td>
<td>Address of the instruction after the SVC instruction</td>
<td>Non-EL2&lt;sup&gt;a&lt;/sup&gt; or EL2&lt;sup&gt;c&lt;/sup&gt;</td>
</tr>
</tbody>
</table>
Note

- Although Reset is described as an exception, it differs significantly from other exceptions. The architecture has no concept of a return from a Reset and therefore it is not listed in this section.

- For each exception, the preferred return address is not affected by the Exception level from which the exception was taken.

The link value saved, and where it is saved, depend on whether the exception is taken to a non-EL2 mode, or to an EL2 mode, as follows:

**Exception taken to a non-EL2 mode**

The link value is saved in the LR for the mode to which the exception is taken.

The saved link value is the preferred return address for the exception, plus an offset that depends on the instruction set state when the exception was taken, as Table G1-10 on page G1-8938 shows:

**Table G1-9 Exception return addresses for exceptions taken to AArch32 state (continued)**

<table>
<thead>
<tr>
<th>Exception</th>
<th>Preferred return address</th>
<th>Taken to a mode at</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure Monitor Call</td>
<td>Address of the instruction after the SMC instruction</td>
<td>EL3(^{b}), and only in Secure state</td>
</tr>
<tr>
<td>Hypervisor Call</td>
<td>Address of the instruction after the HVC instruction</td>
<td>EL2 only(^{c})</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>Address of aborted instruction fetch</td>
<td>Non-EL2(^{a}) or EL2(^{c})</td>
</tr>
<tr>
<td>Data Abort</td>
<td>Address of instruction that generated the abort</td>
<td>Non-EL2(^{a}) or EL2(^{c})</td>
</tr>
<tr>
<td>Virtual Abort</td>
<td>Address of next instruction to execute</td>
<td>EL1, and only in Non-secure state</td>
</tr>
<tr>
<td>IRQ or FIQ</td>
<td>Address of next instruction to execute</td>
<td>Non-EL2(^{a}) or EL2(^{c})</td>
</tr>
<tr>
<td>Virtual IRQ or Virtual FIQ</td>
<td>Address of next instruction to execute</td>
<td>EL1, and only in Non-secure state</td>
</tr>
</tbody>
</table>

\(^{a}\) EL1 if the exception is taken to a Non-secure mode, or is taken to a Secure mode when EL3 is using AArch64. EL3 if the exception is taken to a Secure mode when EL3 is using AArch64.

\(^{b}\) A Secure Monitor Call exception is taken to EL3, and therefore is taken to AArch32 state only if EL3 is using AArch32, in which case it is taken to Monitor mode.

\(^{c}\) EL2 is implemented only in Non-secure state when using AArch32 state. Therefore, an exception can be taken to EL2 mode only if it is taken from Non-secure state when using AArch32 state.

**Table G1-10 Offsets applied to Link value for exceptions taken to non-EL2 modes**

<table>
<thead>
<tr>
<th>Exception</th>
<th>Offset, for PE state of:</th>
<th>A32</th>
<th>T32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined Instruction</td>
<td></td>
<td>+4</td>
<td>+2</td>
</tr>
<tr>
<td>Monitor Trap</td>
<td></td>
<td>+4</td>
<td>+2</td>
</tr>
<tr>
<td>Supervisor Call</td>
<td></td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Secure Monitor Call</td>
<td></td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td></td>
<td>+4</td>
<td>+4</td>
</tr>
<tr>
<td>Data Abort</td>
<td></td>
<td>+8</td>
<td>+8</td>
</tr>
</tbody>
</table>
Table G1-10 Offsets applied to Link value for exceptions taken to non-EL2 modes (continued)

<table>
<thead>
<tr>
<th>Exception</th>
<th>Offset, for PE state of:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A32</td>
</tr>
<tr>
<td>Virtual Abort</td>
<td>+8</td>
</tr>
<tr>
<td>IRQ or FIQ</td>
<td>+4</td>
</tr>
<tr>
<td>Virtual IRQ or Virtual FIQ</td>
<td>+4</td>
</tr>
</tbody>
</table>

Exception taken to an EL2 mode

The link value is saved in the ELR_hyp Special-purpose register.
The saved link value is the preferred return address for the exception, as shown in Table G1-9 on page G1-8937, with no offset.

G1.12.4 PE mode for taking exceptions

The following principles determine the Exception level to which an exception is taken, and if that Exception level is using AArch32, the PE mode to which the exception is taken:

- An exception cannot be taken to the EL0 mode.
- An exception is taken either:
  - To the Exception level at which the PE was executing when it took the exception.
  - To a higher Exception level.

This means that, in Secure state:

- When EL3 is using AArch32, an exception is always taken to an EL3 mode.
- When EL3 is using AArch64, an exception that is taken to AArch32 state is taken to an EL1 mode.

- Configuration options and other features provided by EL2 and EL3 can determine the mode to which some exceptions are taken, as follows:

  **In an implementation that does not include EL2 or EL3**
  An exception is always taken to the default mode for that exception.

  **In an implementation that includes EL3**
  A Secure Monitor Call exception is always taken to EL3. This means:

  - If EL3 is using AArch32 the exception is taken to Secure Monitor mode.
  - If EL3 is using AArch64, then executing the instruction generates an exception that is taken to EL3, see *Execution of an SMC instruction from a privileged Exception level that is using AArch32 on page G1-8941*.

  IRQ, FIQ, and External abort exceptions can be configured to be taken to EL3. Therefore, if EL3 is using AArch32 the exceptions are taken to Secure Monitor mode.

  When EL3 is using AArch32, a Monitor Trap exception is taken to Secure Monitor mode.

  Any exception taken from Secure state that is not taken to Secure Monitor mode is taken to Secure state in the default mode for that exception. As described in *Security state, Exception levels, and AArch32 execution privilege on page G1-8908*, this means it is taken to:

  - An EL3 mode other than Monitor mode if EL3 is using AArch32.
  - An EL1 mode if EL3 is using AArch64.

  If the implementation does not include EL2, any exception taken from Non-secure state that is not taken to Secure Monitor mode is taken to Non-secure state to the default mode for that exception. The default mode will be an EL1 mode.
In an implementation that includes EL2

An exception taken from Non-secure state that is not taken to Secure Monitor mode is taken to Non-secure state and:

• If the exception is taken from Hyp mode, then it is taken to Hyp mode.
• Otherwise, the exception is either taken to Hyp mode, as described in Exceptions taken to Hyp mode, or taken to the default mode for the exception.

Note

• Hyp mode is the EL2 mode. The other modes to which an exception can be taken in Non-secure state are EL1 modes.
• Hyp mode has no effect on the handling of exceptions taken from Secure state.

Table G1-7 on page G1-8931 shows the default mode to which each exception is taken.

Asynchronous exception routing controls on page G1-8957 describes the exception routing controls provided by EL2 and EL3.

Routing of aborts taken to AArch32 state on page G1-8948 gives more information about the modes to which memory aborts are taken.

The possible modes for taking each exception on page G1-8941 shows all modes to which each exception might be taken, in any implementation. That is, it applies to implementations:

• That include neither EL2 nor EL3.
• That include EL2 but not EL3.
• That do not include EL2 but include EL3.
• That include both EL2 and EL3.

Exceptions taken to Hyp mode

In an implementation that includes EL2 and EL3, when EL2 is using AArch32:

• Any exception taken from Hyp mode that is not routed to EL3 by the controls described in Asynchronous exception routing controls on page G1-8957 is taken to Hyp mode.

• The following exceptions, if taken from Non-secure state, are taken to Hyp mode:
  — An abort that Routing of aborts taken to AArch32 state on page G1-8948 identifies as taken to Hyp mode.
  — A Hyp Trap exception, see EL2 configurable controls on page G1-9010.
  — A Hypervisor Call exception. This is generated by executing an HVC instruction in a Non-secure mode.
  — An SError interrupt exception, IRQ exception or FIQ exception that is not routed to EL3 but is explicitly routed to Hyp mode, as described in Asynchronous exception routing controls on page G1-8957.
  — A synchronous External abort, Alignment fault, Undefined Instruction exception, or Supervisor Call exception taken from the Non-secure EL0 mode and explicitly routed to Hyp mode, as described in Routing exceptions from Non-secure EL0 to EL2 on page G1-8944.

Note

A synchronous External abort can be routed to Hyp mode only if it is not routed to EL3.

Note

A debug exception that is explicitly routed to Hyp mode, as described in Routing debug exceptions to EL2 using AArch32 on page G1-8946.

Note

The virtual exceptions cannot be taken to Hyp mode. They are always taken to a Non-secure EL1 mode.
Security behavior in Exception levels using AArch32 when EL2 or EL3 are using AArch64

As described in The Armv8-A security model on page G1-8905, when EL3 is using AArch64, lower Exception levels, in either Security state, can be using AArch32. This means software executing in those Exception levels might try to access AArch32 security features that are not available. The following subsections describe the associated behaviors:

- Execution of an SMC instruction from a privileged Exception level that is using AArch32
- Non-secure reads of the NSACR
- Secure EL1 operations when Secure EL1 is using AArch32 state

Execution of an SMC instruction from a privileged Exception level that is using AArch32

When EL3 is using AArch64, an SMC instruction executed from Secure or Non-secure EL1 using AArch32, or from Non-secure EL2 using AArch32 when the value of HCR.TSC is 0, generates an exception that is taken to EL3. The exception syndrome is reported with an EC value of \(0x13\), SMC instruction executed in AArch32 state.

Non-secure reads of the NSACR

The NSACR is defined as being RO from Non-secure PE modes other than User mode. When EL3 is using AArch64, a read of the NSACR returns a fixed value of \(0x00000C00\) in the following cases:

- If the read is from a Non-secure EL1 mode when EL1 is using AArch32.
- If the read is from Hyp mode when EL2 is using AArch32.

Secure EL1 operations when Secure EL1 is using AArch32 state

When Secure EL1 is using AArch32 and if FEAT_SEL2 is implemented and enabled or EL3 is using AArch64:

- Any of the following operations performed in a Secure EL1 mode is trapped to Secure EL3:
  - A read or write of any of the SCR, NSACR, MVBAR, and SDCR.
  - Executing any of the \(\text{ATS12NSO}^{**}\) instructions.
  - Executing an \(\text{SRS}\) instruction that would use SP_mon, see SRS, SRSDA, SRSDB, SRSIA, SRSIB on page F5-7936.
  - Executing an \(\text{MRS}\) (banked register) or \(\text{MSR}\) (banked register) instruction that would access SPSR_mon, SP_mon, or LR_mon, see MRS (Banked register) on page F5-7736 and MSR (Banked register) on page F5-7740.

- Any attempt to move into Hypervisor mode, either by an exception return or by executing a CPS or MSR instruction, is treated as an illegal operation and is handled as described in Illegal return events from AArch32 state on page G1-8952.

- Any attempt to move into Monitor mode, either by an exception return or by executing a CPS or MSR instruction, is treated as an illegal operation and is handled as described in Illegal return events from AArch32 state on page G1-8952.

Note

This functionality supports a usage model where:

- EL3 uses AArch64.
- Secure software executed in Secure EL1 using AArch32 and Secure EL0 using AArch32.
- The Non-secure state uses AArch64.

The possible modes for taking each exception

Each of the exception descriptions in AArch32 state exception descriptions on page G1-8964 includes a subsection that describes the modes to which each exception can be taken. Those subsections are:

- The PE mode to which the Undefined Instruction exception is taken on page G1-8965.
• The PE mode to which the Hyp Trap exception is taken on page G1-8967.
• The PE mode to which the Monitor Trap exception is taken on page G1-8967.
• The PE mode to which the Supervisor Call exception is taken on page G1-8968.
• The PE mode to which the Secure Monitor Call exception is taken on page G1-8970.
• The PE mode to which the Hypervisor Call exception is taken on page G1-8971.
• The PE mode to which the Prefetch Abort exception is taken on page G1-8973.
• The PE mode to which the Data Abort exception is taken on page G1-8976.
• The PE mode to which the Virtual SError interrupt exception is taken on page G1-8980.
• The PE mode to which the physical IRQ exception is taken on page G1-8981.
• The PE mode to which the Virtual IRQ exception is taken on page G1-8982.
• The PE mode to which the physical FIQ exception is taken on page G1-8983.
• The PE mode to which the Virtual FIQ exception is taken on page G1-8984.

These descriptions also show the vector offset for the exception entry for each mode. These descriptions assume that all Exception levels are using AArch32, meaning:
• HCR, rather than HCR_EL2, controls the routing of exceptions to EL2.
• SCR, rather than SCR_EL3, controls the routing of exceptions to EL3.

For more information about:
• Vector offsets, see Exception vectors and the exception base address on page G1-8929.
• The routing of synchronous External aborts or SError, IRQ, and FIQ interrupt exceptions, and the virtual exceptions, see Asynchronous exception routing controls on page G1-8957.

UNPREDICTABLE cases when the value of HCR.TGE is 1

When the value of HCR.TGE is 1, exceptions that would otherwise be taken to EL1 are, instead, routed to EL2, see Routing exceptions from Non-secure EL0 to EL2 on page G1-8944. Related to this, when the value of HCR.TGE is 1, execution in a Non-secure EL1 mode is UNPREDICTABLE. The architecture does not constrain this UNPREDICTABLE behavior, but software that follows the Arm recommendations cannot get to this state. When following the Arm recommendations, any attempt to move to a Non-secure EL1 mode when the value of HCR.TGE is 1 is either:
• An illegal exception return, see Illegal return events from AArch32 state on page G1-8952.
• An illegal PE mode change, see Illegal changes to PSTATE.M on page G1-8925.

G1.12.5 PE state on exception entry

The description of each exception includes a pseudocode description of entry to that exception, as Table G1-8 on page G1-8937 shows. The following sections describe the PE state changes on entering an exception, for different implementations and operating states. However, you must always see the exception entry pseudocode for a full description of the state changes on exception entry:
• Instruction set state on exception entry on page G1-8943.
• PSTATE.E value on exception entry on page G1-8943.
• PSTATE.{A, I, F, M} values on exception entry on page G1-8943.

Note

The descriptions in these sections assume that EL2 and EL3, which control some aspects of the routing of exceptions taken from EL1 or EL0, are both using AArch32. If this is not the case:
• If EL2 is using AArch64:
  — Controls shown as provided by the HSCTLR are provided by the SCTLR_EL2.
  — Controls shown as provided by the HCR are provided by the HCR_EL2.
• If EL3 is using AArch64, controls shown as provided by the SCR are provided by the SCR_EL3.
Instruction set state on exception entry

Exception handlers can execute in either T32 state or A32 state. On exception entry, PSTATE.T is set to the required value, as determined by SCTLR.TE or HSCTLR.TE, depending on the mode the exception is taken to. Table G1-11 on page G1-8943 shows this:

Table G1-11 PSTATE.T bit value on exception entry

<table>
<thead>
<tr>
<th>Mode to which exception is taken</th>
<th>HSCTLR.TE</th>
<th>SCTLR.TE</th>
<th>PSTATE.T</th>
<th>Exception handler state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Hyp mode</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>A32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>T32</td>
</tr>
<tr>
<td>Hyp mode</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>A32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>x</td>
<td>T32</td>
</tr>
</tbody>
</table>

When an implementation includes EL3 and EL3 is using AArch32, SCTLR is banked for Secure and Non-secure states, and therefore the TE bit value might be different for Secure and Non-secure states. For an exception taken to a PE mode other than Hyp mode, the SCTLR.TE bit for the Security state to which the exception is taken determines the instruction set state for the exception handler. This means the instruction set state in which an exception handler might execute depends on the Security state to which the exception is taken.

PSTATE.E value on exception entry

PSTATE.E controls the load and store endianness for data handling. Table G1-12 on page G1-8943 show the value to which this bit is set on exception entry:

Table G1-12 PSTATE.E value on exception entry

<table>
<thead>
<tr>
<th>Exception mode</th>
<th>HSCTLR.EE</th>
<th>SCTLR.EE</th>
<th>Endianness for data loads and stores</th>
<th>PSTATE.E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure or Non-secure EL1</td>
<td>x</td>
<td>0</td>
<td>Little-endian</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Big-endian</td>
<td>1</td>
</tr>
<tr>
<td>Hyp</td>
<td>0</td>
<td>x</td>
<td>Little-endian</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>x</td>
<td>Big-endian</td>
<td>1</td>
</tr>
</tbody>
</table>

For more information, see the bit description in Saved Program Status Registers (SPSRs) on page G1-8919.

PSTATE.{A, I, F, M} values on exception entry

On exception entry, PSTATE.M is set to the value for the mode to which the exception is taken, as described in PE mode for taking exceptions on page G1-8939.

Table G1-13 on page G1-8944 shows the cases where PSTATE.{A, I, F} bits are set to 1 on an exception entry, and how this depends on the mode and Security state to which an exception is taken. If the table entry for a particular mode and Security state does not define a value for a PSTATE.{A, I, F} bit then that bit is unchanged by the exception entry. In this table:

- The PE mode exception is taken to column is the mode to which the exception is taken.
- The Non-secure column applies to exceptions taken to Non-secure state in an implementation that includes EL3 but does not include EL2.
- The Secure column applies to:
  - Exceptions taken to Secure state.
  - Implementations that do not include the EL3.
Exceptions taken to Non-secure state in an implementation that includes EL2.

Table G1-13 PSTATE.{A, I, F} values on exception entry

<table>
<thead>
<tr>
<th>PE mode exception is taken to</th>
<th>Security state</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non-secure</td>
</tr>
<tr>
<td>Hyp</td>
<td>If SCR.EA==0 then PSTATE.A is set to 1</td>
</tr>
<tr>
<td></td>
<td>If SCR.IRQ==0 then PSTATE.I is set to 1</td>
</tr>
<tr>
<td></td>
<td>If SCR.FIQ==0 then PSTATE.F is set to 1</td>
</tr>
<tr>
<td>Monitor</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>FIQ</td>
<td>PSTATE.A is set to 1</td>
</tr>
<tr>
<td></td>
<td>PSTATE.I is set to 1</td>
</tr>
<tr>
<td></td>
<td>PSTATE.F is set to 1</td>
</tr>
<tr>
<td>IRQ, Abort</td>
<td>PSTATE.A is set to 1</td>
</tr>
<tr>
<td></td>
<td>PSTATE.I is set to 1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Undefined, Supervisor</td>
<td>PSTATE.I is set to 1</td>
</tr>
</tbody>
</table>

Asynchronous exception behavior for exceptions taken from AArch32 state on page G1-8956 describes how, in some situations, the PSTATE.{A, I, F} bits mask the taking of SError interrupts, IRQ interrupts, and FIQ interrupts.

G1.12.6 Routing exceptions from Non-secure EL0 to EL2

--- Note ---

The routing control described in this section permits a Non-secure state usage model where applications execute in User mode under a hypervisor, which executes in Hyp mode, without a Guest OS running at Non-secure EL1. This control applies when the PE is executing in Non-secure EL0 using AArch32 and EL2 is using AArch32 and the value of HCR.TGE is 1.

If the PE is in Non-secure User mode, any exception that would otherwise be taken to Non-secure EL1 is taken to EL2 if either:

- EL2 is using AArch32 and the value of HCR.TGE is 1.
  - In this case the exception is taken to Hyp mode, instead of to the default Non-secure mode for handling the exception. For more information, see Exception reporting when HCR.TGE routes an exception to EL2 using AArch32 on page G1-8945.

- EL2 is using AArch64 and the value of HCR_EL2.TGE is 1.
  - In this case the exception is taken to EL2 using AArch64, see Exception entry on page D1-4641.

Any exception that is routed to Secure Monitor mode or to EL3 using AArch64 is unaffected by the value of HCR.TGE or HCR_EL2.TGE.

When the value of HCR.TGE is 1, meaning TGE routing from Non-secure EL0 using AArch32 to EL2 using AArch32 applies:

- The SCTLR.M bit is treated as 0 for all purposes other than a direct read of the SCTLR register.
- Each of the HCR.\{FMO, IMO, AMO\} bits is treated as 1 for all purposes other than a direct read of the HCR register.
• Each of the HDCR. {TDE, TDA, TDRA, TDOSA} bits is treated as 1 for all purposes other than a direct read of the HDCR register.

• An exception return to Non-secure EL1 is treated as an illegal exception return, see Illegal return events from AArch32 state on page G1-8952.

• All virtual interrupts, including any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts, are disabled.

Exception reporting when HCR.TGE routes an exception to EL2 using AArch32

The following sections give more information about the behavior of synchronous exceptions that are routed to Hyp mode because the value of HCR.TGE is 1:

• Undefined Instruction exception, when the value of HCR.TGE is 1.

• Supervisor Call exception, when the value of HCR.TGE is 1.

• Abort exceptions, when the value of HCR.TGE is 1.

• Reporting of exceptions routed to EL2 using AArch32 because the value of HCR.TGE is 1 on page G1-8946.

Undefined Instruction exception, when the value of HCR.TGE is 1

When HCR.TGE is set to 1, if the PE is executing in Non-secure User mode and attempts to execute an UNDEFINED instruction, it takes the Hyp Trap exception, instead of an Undefined Instruction exception. On taking the Hyp Trap exception, the HSR reports an unknown reason for the exception, using the EC value 0x00. For more information, see Use of the HSR on page G5-9265.

Supervisor Call exception, when the value of HCR.TGE is 1

When HCR.TGE is set to 1, if the PE executes an SVC instruction in Non-secure User mode, the Supervisor Call exception generated by the instruction is taken to Hyp mode.

The HSR reports that entry to Hyp mode was because of a Supervisor Call exception, and:

• If the SVC is unconditional, takes for the imm16 value in the HSR:
  — A zero-extended 8-bit immediate value for the T32 SVC instruction.

    Note
    The only T32 encoding for SVC is a 16-bit instruction encoding.

    — The bottom 16 bits of the immediate value for the A32 SVC instruction.

• If the SVC is conditional, the imm16 value in the HSR is UNKNOWN.

If the SVC is conditional, the PE takes the exception only if the instruction passes its Condition code check.

The HSR reports the exception as a Supervisor Call exception taken to Hyp mode, using the EC value 0x11. For more information, see Use of the HSR on page G5-9265.

    Note
    The effect of setting HCR.TGE to 1 is to route the Supervisor Call exception to Hyp mode, not to trap the execution of the SVC instruction. This means that the preferred return address for the exception, when routed to Hyp mode in this way, is the instruction after the SVC instruction.

Abort exceptions, when the value of HCR.TGE is 1

When the value of HCR.TGE is 1, if the PE is executing in Non-secure User mode then any abort exception that is not routed to Secure Monitor mode or to EL3 using AArch64 generates an exception that is taken as a Hyp Trap exception. Where an attempt to execute an instruction causes an abort, on taking the Hyp Trap exception, the HSR indicates whether a Data Abort exception or a Prefetch Abort exception caused the Hyp Trap exception entry, and presents a valid syndrome in the HSR.

When SCR.EA is set to 1, External aborts and SError interrupts are routed to EL3, and this routing takes priority over the HCR.TGE routing. For more information, see Routing of aborts taken to AArch32 state on page G1-8948.
An SError interrupt that is routed to Hyp mode because the value of HCR.TGE is 1 is reported as a Data Abort exception routed to Hyp mode.

The HSR reports the exception either:
- As a Prefetch Abort exception routed to Hyp mode, using the EC value 0x20.
- As a Data Abort exception routed to Hyp mode, using the EC value 0x24.

For more information about the exception reporting, see Use of the HSR on page G5-9265.

**Reporting of exceptions routed to EL2 using AArch32 because the value of HCR.TGE is 1**

PL1 configurable controls on page G1-9002 describes controls that, when the value of HCR.TGE is 0, can generate exceptions that are taken from Non-secure EL0 to EL1. When EL2 is using AArch32 and the value of HCR.TGE is 1, the exceptions generated by these controls are routed to Hyp mode. Table G1-14 shows how these exceptions are then reported in the HSR.

### Table G1-14 Syndrome reporting in HSR from HCR.TGE routing of traps, disables, and enables

<table>
<thead>
<tr>
<th>Control provided by PL1</th>
<th>Control typea</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCTLR.{nTWE, nTWI}</td>
<td>T</td>
<td>Uses EC value 0x00, Exception for an unknown reason</td>
</tr>
<tr>
<td>SCTLR.{SED, ITD}</td>
<td>D</td>
<td>Uses EC value 0x00, Exception for an unknown reason</td>
</tr>
<tr>
<td>SCTLR.CP15BEN</td>
<td>E</td>
<td>Uses EC value 0x00, Exception for an unknown reason</td>
</tr>
<tr>
<td>CPACR.TRCDIS</td>
<td>T</td>
<td>Uses EC value 0x00, Exception for an unknown reason</td>
</tr>
<tr>
<td>CPACR.{cp11, cp10}</td>
<td>E</td>
<td>Uses EC value 0x00, Exception for an unknown reason</td>
</tr>
<tr>
<td>FPEXC.EN</td>
<td>E</td>
<td>Uses EC value 0x00, Exception for an unknown reason</td>
</tr>
<tr>
<td>CPACR.ASEDIS</td>
<td>D</td>
<td>Uses EC value 0x00, Exception for an unknown reason</td>
</tr>
<tr>
<td>DBGDSCRxext.UDCCdis</td>
<td>T</td>
<td>Uses EC value 0x00, Exception for an unknown reason</td>
</tr>
<tr>
<td>CNTKCTL.{PL0PTEN, PL0VTEN, PL0PCTEN, PL0VCTEN}</td>
<td>T</td>
<td>Uses EC value 0x00, Exception for an unknown reason</td>
</tr>
<tr>
<td>PMUSERENR.{ER, CR, SW, EN}</td>
<td>T</td>
<td>Uses EC value 0x00, Exception for an unknown reason</td>
</tr>
</tbody>
</table>

---

**G1.12.7 Routing debug exceptions to EL2 using AArch32**

When the value of HDCR.TDE is 1, if the PE is executing in a Non-secure mode other than Hyp mode, any Debug exception is routed to Hyp mode. This means it generates a Hyp Trap exception. This applies to:

- Debug exceptions associated with an instruction fetch, that would otherwise generate a Prefetch Abort exception. These are the Breakpoint, Breakpoint Instruction, and Vector Catch exception, see Chapter G2 AArch32 Self-hosted Debug.
- Watchpoint exceptions associated with data accesses, that would otherwise generate a Data Abort exception. See Watchpoint exceptions on page G2-9079.

When the value of HDCR.TDE is 1, each of the HDCR.\{TDRA, TDOSA, TDA\} bits is treated as 1 for all purposes other than reading the HDCR register.

---

**Note**

- A Breakpoint or Watchpoint debug event that generates entry to Debug state cannot be trapped to Hyp mode. See Breakpoint and Watchpoint debug events on page H2-10234.
• When HDCR.TDE is set to 1, the Hyp Trap exception is generated instead of the Prefetch Abort exception or Data Abort exception that is otherwise generated by the Debug exception.

• Debug exceptions, other than Breakpoint Instruction exceptions, are never generated in Hyp mode.

When a Hyp Trap exception is generated because the value of HDCR.TDE is 1, The HSR reports the exception either:
• As a Prefetch Abort exception routed to Hyp mode, using the EC value 0x20.
• As a Data Abort exception routed to Hyp mode, using the EC value 0x24.

For more information, see Use of the HSR on page G5-9265.
G1.13 Routing of aborts taken to AArch32 state

A memory abort is either a Data Abort exception or a Prefetch Abort exception. When executing in AArch32 state, depending on the cause of the abort, and possibly on configuration settings, an abort is taken either:

• To the Exception level of the PE mode from which the abort is taken. In this case the abort is taken to AArch32 state.

• To a higher Exception level. In this case the Exception level to which the abort is taken is either:
  — Using AArch32. In this case, this chapter describes how the abort is handled.
  — Using AArch64. In this case, Chapter D8 The AArch64 Virtual Memory System Architecture describes how the abort is handled.

For an abort taken to an Exception level that is using AArch32, the mode to which a memory abort is taken depends on the reason for the exception, the mode the PE is in when it takes the exception, and configuration settings, as follows:

Memory aborts taken to Monitor mode

If an implementation includes EL3, when the value of SCR.EA is 1, all External aborts are taken to EL3, and if EL3 is using AArch32 they are taken to Monitor mode. This applies to aborts taken from Secure modes and from Non-secure modes.

Memory aborts taken to Secure Abort mode

If an implementation includes EL3, when the PE is executing in Secure state, all memory aborts that are not routed to EL3 are taken to Secure Abort mode.

——— Note —————

The only memory aborts that can be routed to Monitor mode are External aborts.

———

Memory aborts taken to Hyp mode

If an implementation includes EL2, when the PE is executing in Non-secure state, the following aborts are taken to EL2. If EL2 is using AArch32 this means they are taken to Hyp mode:

• Alignment faults taken:
  — When the PE is in Hyp mode.
  — When the PE is in a Non-secure PL1 or EL0 mode and the exception is generated because the Non-secure PL1&0 stage 2 translation identifies the target of an unaligned access as any type of Device memory.
  — When the PE is in Non-secure User mode and HCR.TGE is set to 1. For more information, see Abort exceptions, when the value of HCR.TGE is 1 on page G1-8945.

• When the PE is using the Non-secure PL1&0 translation regime:
  — MMU faults from stage 2 translations, for which the stage 1 translation did not cause an MMU fault.
  — Any abort taken during the stage 2 translation of an address accessed in a stage 1 translation table walk that is not routed to Secure Monitor mode, see Stage 2 fault on a stage 1 translation table walk on page G5-9246.

• When the PE is using the Non-secure EL2 translation regime, MMU faults from stage 1 translations.

——— Note —————

The Non-secure EL2 translation regime has only one stage of translation.

———

• External aborts, if SCR.EA is set to 0 and any of the following applies:
  — The PE was executing in Hyp mode when it took the exception.
The PE was executing in a Non-secure PL1 or EL0 mode when it took the exception, the abort is asynchronous, and HCR.AMO is set to 1. For more information, see *Asynchronous exception routing controls* on page G1-8957.

— The PE was executing in the Non-secure User mode when it took the exception, the abort is synchronous, and HCR.TGE is set to 1. For more information, see *Abort exceptions, when the value of HCR.TGE is 1* on page G1-8945.

— The Reliability, Availability, and Serviceability Extension is implemented, the PE was executing in a Non-secure PL1 or EL0 mode when it took the exception, the abort is synchronous, and the value of HCR2.TEA is 1.

— The abort occurred on a stage 2 translation table walk.

- Debug exceptions, if HDCR.TDE is set to 1. For more information, see *Routing debug exceptions to EL2 using AArch32* on page G1-8946.

### Memory aborts taken to Non-secure Abort mode

In an implementation that does not include EL3, all memory aborts that are taken to an Exception level that is using AArch32 are taken to Abort mode.

Otherwise, when the PE is executing in Non-secure state, the following aborts are taken to Non-secure Abort mode:

- When the PE is in a Non-secure PL1 or EL0 mode, Alignment faults taken for any of the following reasons:
  — SCTLR.A is set to 1.
  — An instruction that does not support unaligned accesses is committed for execution, and the instruction accesses an unaligned address.
  — The PL1&0 stage 1 translation identifies the target of an unaligned access as any type of Device memory.

**Note**

In an implementation that does not include EL2, this case results in a **CONSTRAINED UNPREDICTABLE** memory access, see *Cases where unaligned accesses are CONSTRAINED UNPREDICTABLE* on page E2-7194 and *Loads and Stores to unaligned locations* on page K1-11564.

If an implementation includes EL2 and the PE is in Non-secure User mode, these exceptions are taken to Abort mode only if the value of HCR.TGE is 0.

- When the PE is using the Non-secure PL1&0 translation regime, an MMU fault from a stage 1 translation.

- External aborts, if the PE was executing in a Non-secure PL1 or EL0 mode when it took the exception and both:
  — The value of SCR.EA is 0, meaning the abort is not taken to EL3.
  — The abort is not taken to EL2 for one of the reasons defined in *Memory aborts taken to Hyp mode*.

- Virtual Aborts, see *Virtual exceptions when an implementation includes EL2* on page G1-8956.

- When the value of HDCR.TDE is 0, Debug exceptions. For more information, see *Routing debug exceptions to EL2 using AArch32* on page G1-8946.

**Note**

If EL0 is using AArch32 and EL1 is using AArch64, then any of these memory aborts taken from User mode are taken to EL1, as described in *Chapter D8 The AArch64 Virtual Memory System Architecture*.
Memory aborts with IMPLEMENTATION DEFINED behavior

In addition, a PE can generate an abort for an IMPLEMENTATION DEFINED reason associated with lockdown. In an implementation that includes EL2, whether such an abort is taken to Non-secure Abort mode or is taken to EL2 is IMPLEMENTATION DEFINED, and an implementation might include a mechanism to select whether the abort is routed to Non-secure Abort mode or to EL2.

When the PE is in a Non-secure mode other than Hyp mode, if multiple factors cause an Alignment fault, the abort is taken to Non-secure Abort mode if any of the factors require the abort to be taken to Abort mode. For example, if the SCTLRA bit is set to 1, and the access is an unaligned access to an address that the stage 2 translation tables mark as Device-nGnRnE, then the abort is taken to Non-secure Abort mode.

For more information, see Handling exceptions that are taken to an Exception level using AArch32 on page G1-8929.
G1.14 Exception return to an Exception level using AArch32

In the Arm architecture, exception return to an Exception level that is using AArch32 requires the simultaneous restoration of the PC and PSTATE to values that are consistent with the desired state of execution on returning from the exception. Typically, exception return involves returning to one of:

- The instruction after the instruction boundary at which an asynchronous exception was taken.
- The instruction following an SVC, SMC, or HMC instruction, for an exception generated by one of those instructions.
- The instruction that caused the exception, after the reason for the exception has been removed.
- The subsequent instruction, if the instruction that caused the exception has been emulated in the exception handler.

The Arm architecture defines a preferred return address for each exception other than Reset, see Link values saved on exception entry on page G1-8937. The values of the SPSR.IT[7:0] bits generated on exception entry are always correct for this preferred return address, but might require adjustment by the exception handler if returning elsewhere.

In some cases, to calculate the appropriate preferred return address for a return to an Exception level that is using AArch32, a subtraction must be performed on the link value saved on taking the exception. The description of each exception includes any value that must be subtracted from the link value, and other information about the required exception return.

On an exception return, the PSTATE takes either:

- The value loaded by the RFE instruction.
- If the exception return is not performed by executing an RFE instruction, the value of the current SPSR at the time of the exception return.

If FEAT_MTE is implemented PSTATE.TCO is not updated on Exception return to AArch32 state.

Illegal return events from AArch32 state on page G1-8952 describes the behavior if the restored PE state would not be valid for the Exception level, PE mode, and Security state targeted by the exception return.

G1.14.1 Exception return instructions

The instructions that an exception handler can use to return from an exception depend on whether the exception was taken to an EL1 mode, or in an EL2 mode, see:

- Return from an exception taken to a PE mode other than Hyp mode.
- Return from an exception taken to Hyp mode on page G1-8952.

Return from an exception taken to a PE mode other than Hyp mode

For an exception taken to a PE mode other than Hyp mode, the Arm AArch32 architecture provides the following exception return instructions:

- From privileged modes other than System mode, the ERET instruction. After the exception return, execution resumes from the address held in the LR (R14) for the mode in which ERET is executed. See ERET on page F5-7570.
- Data-processing instructions with the S bit set and the PC as a destination, see MOV, MOVS (register) on page F5-7719 and SUB, SUBS (immediate) on page F5-8039.

Note

The A32 instruction set includes other instructions that can be used for an exception return, but Arm deprecates any use of those instructions.
Typically:
— A return where no subtraction is required uses `SUBS` with an operand of 0, or the equivalent `MOVS` instruction.
— A return requiring subtraction uses `SUBS` with a nonzero operand.

• The `RFE` instruction, see `RFE, RFEDA, RFEDB, RFEIA, RFEIB` on page F5-7830. If a subtraction is required, typically it is performed before saving the LR value to memory. After the exception return, execution resumes from the address held in the memory location indicated by the base register specified by the `RFE` instruction.

• In A32 state, a form of the `LDM` instruction in which the PC is one of the registers loaded, see `LDM (exception return)` on page F5-7604. If a subtraction is required, typically it is performed before saving the LR value to memory.

**Return from an exception taken to Hyp mode**

For an exception taken to Hyp mode, the Arm architecture provides the `ERET` instruction, see `ERET` on page F5-7570. An exception handler executing in Hyp mode must return using the `ERET` instruction.

Hyp mode is implemented only as part of EL2.

**G1.14.2 Alignment of exception returns**

The T bit of the value transferred to the `PSTATE` by an exception return controls the target instruction set of that return. The behavior of the hardware for exception returns for different values of the T bit is as follows:

\[
\begin{align*}
T &= 0 & \text{The target instruction set state is A32 state. Bits}[1:0]\text{ of the address transferred to the PC are ignored by the hardware.} \\
T &= 1 & \begin{align*}
\text{The target instruction set state is T32 state:} \\
\bullet & \text{Bit}[0] \text{ of the address transferred to the PC is ignored by the hardware.} \\
\bullet & \text{Bit}[1] \text{ of the address transferred to the PC is part of the instruction address.}
\end{align*}
\end{align*}
\]

**Note**

In previous versions of the Arm architecture, the `PSTATE.{J, T}` bits determined the Instruction set state. From the introduction of Armv8, `PSTATE.J` is `RES0`.

Arm deprecates any dependence on the requirements that the hardware ignores bits of the address. Arm recommends that the address transferred to the PC for an exception return is correctly aligned for the target instruction set.

After an exception entry other than Reset, the LR value has the correct alignment for the instruction set indicated by the `SPSR.T` bit. This means that if exception return instructions are used with the LR and `SPSR` values produced by such an exception entry, the only precaution software needs to take to ensure correct alignment is that any subtraction is of a multiple of four if returning to A32 state, or a multiple of two if returning to T32 state.

**G1.14.3 Illegal return events from AArch32 state**

Throughout this section:

**Return**

In AArch32 state, refers to any of:
• Execution of any exception return instruction.
• Execution of a `DRPS` instruction in Debug state.
• Exit from Debug state.

If an exception or debug return from an Exception level using AArch32 triggers an illegal exception return, then bit[1] of the PC is either:
• Zero.
• The value of bit[1] of the return address for the exception or debug return.
The choice between these two alternatives is made by the implementation, and might differ from instance to instance of an illegal exception return.

--- Note ---
This means software must support both alternatives.

**Saved process state value**
In AArch32 state, refers to any of:
- The value held in the SPSR for any exception return other than an exception return made by executing an RFE instruction.
- The value read from memory that is to be restored to PSTATE by the execution of an RFE instruction.
- The value held in the SPSR for the execution of a DRPS instruction in Debug state.
- The value held in the DSPSR for a Debug state exit.

**Link address**
In AArch32 state, refers to any of:
- The address held in the link register for any exception return other than an exception return made by executing an ERET, LDM, or RFE instruction.
- The address held in ELR_hyp for any exception return made by executing an ERET instruction.
- The address read from memory that is to be restored to the PC by the execution of an LDM or RFE instruction.
- The address held in the DLR for Debug state exit.

**Configured from reset**
Indicates the state determined on powerup or reset by a configuration input signal, or by another IMPLEMENTATION DEFINED mechanism.

The architecture has a generic mechanism for handling exception or debug returns to a mode or state that is illegal. In AArch32 state, this can occur as a result of any of the following situations:
- A return where the Exception level being returned to is higher than the current Exception level.
- A return where the mode being returned to is not implemented. For example:
  - A return to Hyp mode when EL2 is not implemented.
  - A return to Monitor mode, when EL3 is either not implemented or using AArch64 state.
- A return to EL2 when:
  - EL3 is implemented and using AArch64, and the values of SCR_EL3.{NS, EEL2} 0.
  - EL3 is implemented and using AArch32, and the value of the SCR.NS bit is 0.
- A return to Non-secure EL1 when:
  - EL2 is implemented and using AArch64, and the value of the HCR_EL2.TGE bit is 1.
  - EL2 is implemented and using AArch32, and the value of the HCR.TGE bit is 1.
- A return where the value of the saved process state M[4:0] field is not a valid AArch32 PE mode for the implementation. Table G1-5 on page G1-8912 shows the valid M[4:0] values for AArch32 PE modes.

In these cases:
- PSTATE.IL is set to 1, to indicate an illegal return.
- PSTATE.M is unchanged. This means the PE mode does not change.
- The SS bit is handled in the same way as any other exception or debug return, see *Software Step exceptions* on page D2-4742.
The following PSTATE bits are restored from the saved process state value:
- The Q Overflow or saturation flag.
- The GE Greater than or Equal flags.
- The E Endianness mapping bit.
- The A, I, F exception mask bits.
- The DIT Data Independent Timing bit.

The PSTATE.{IT, T} bits are each either:
- Set to 0.
- Copied from the saved process state in the SPSR for the PE mode in which the exception is handled.

The choice between these two options is determined by an implementation, and might vary dynamically within an implementation. Correspondingly software must regard the value as being an UNKNOWN choice between the two values.

The PC is restored from the link address, unless the illegal return is the execution of a DRPS instruction in Debug state.

When the value of the PSTATE.IL bit is 1, any attempt to execute any instruction results in an Illegal Execution state exception. See The Illegal Execution state exception.

All aspects of the illegal return, other than the effects described in this section, are the same as for a legal return.

G1.14.4 Legal returns that set PSTATE.IL to 1

In this section, return, saved process state value, and link address have the meaning that is defined in Illegal return events from AArch32 state on page G1-8952.

If the IL bit in the saved process state value is 1, then it is copied to PSTATE meaning that PSTATE.IL is set to 1. In this case, the PSTATE.{IT, T} bits are each either:
- Set to 0.
- Copied from the SPSR, or loaded from memory if the exception return was performed by executing an RFE instruction.

The choice between these two options is determined by an implementation, and might vary dynamically within the implementation. This means software must regard each value as being an UNKNOWN choice between the two permitted values.

Because the return sets the PSTATE.IL bit to 1, any attempt to execute any instruction results in an Illegal Execution state exception. See The Illegal Execution state exception.

G1.14.5 The Illegal Execution state exception

When the value of the PSTATE.IL bit is 1, any attempt to execute an instruction generates an Illegal Execution state exception. In AArch32 state, the PSTATE.IL bit can be set to 1 by one of the following:
- An illegal return, as described in Illegal return events from AArch32 state on page G1-8952.
- An illegal change to PSTATE.M, as described in Illegal changes to PSTATE.M on page G1-8925.
- A legal return that sets PSTATE.IL to 1, as described in Legal returns that set PSTATE.IL to 1.

An Illegal Execution state exception is taken in the same way as an Undefined Instruction exception in the current Exception level. If the current Exception level is EL2 using AArch32 state, the HSR provides additional syndrome information for the exception, see Use of the HSR on page G5-9265.

An Illegal Execution state exception has priority over any other Undefined Instruction exception that might arise from instruction execution.
This section only describes the handling of an Illegal Execution state exception that is taken to an Exception level that is using AArch32 state.

On taking any exception to an Exception level that is using AArch32 state:

1. The value of the PSTATE.IL bit is 1 and this is copied to the SPSR.IL bit for the PE mode to which the exception is taken.
2. The PSTATE.IL bit is cleared to 0.

Note
This means that it is not possible for software to observe the value of PSTATE.IL.

Pseudocode description of exception return

The AArch32.ExceptionReturn() function transfers the return address to the PC and restores PSTATE to its saved value.

This function uses the function SetPSTATEFromPSR().

The IllegalExceptionReturn() function checks for an Illegal Execution state exception.

Chapter J1 Armv8 Pseudocode includes the definitions of these functions.
G1.15  Asynchronous exception behavior for exceptions taken from AArch32 state

In an implementation that does not include EL2 or EL3, the asynchronous exceptions behave as follows when EL1 and EL0 are both using AArch32:

- An SError interrupt is taken to Abort mode.
- An IRQ exception is taken to IRQ mode.
- An FIQ exception is taken to FIQ mode.

These are the default PE modes for taking these exceptions.

However, the PSTATE.{A, I, F} bits mask the asynchronous exceptions, meaning that when the value of one of these PSTATE bits is 1, the corresponding exception is not taken.

If a masked asynchronous exception remains signaled, then the exception remains pending unless the value of the PSTATE bit is changed to 0.

EL2 and EL3 provide controls that affect:

- The routing of these exceptions, see Asynchronous exception routing controls on page G1-8957.
- Masking of these exceptions in Non-secure state, see Asynchronous exception masking controls on page G1-8959.

Similar register control bits are provided regardless of whether EL2 and EL3 are using AArch32 or AArch64:

- The EL2 controls are provided by the HCR when EL2 is using AArch32, and by the HCR_EL2 when EL2 is using AArch64.
- The EL3 controls are provided by the SCR when EL3 is using AArch32, and by the SCR_EL3 when EL3 is using AArch64.

Therefore, most references to the HCR or SCR in this section are to entries in Table K17-1 on page K17-11836, which disambiguates between AArch32 registers and AArch64 registers. However, the Execution states used by EL2 and EL3 do affect some aspects of the routing and masking of the asynchronous exceptions, see Asynchronous exception routing and masking with higher Exception levels using AArch64 on page G1-8961.

G1.15.1 Virtual exceptions when an implementation includes EL2

When implemented, EL2 provides the following virtual exceptions, which correspond to the physical asynchronous exceptions:

- Virtual SError, which corresponds to a physical external SError interrupt.
- Virtual IRQ, which corresponds to a physical IRQ.
- Virtual FIQ, which corresponds to a physical FIQ.

When the value of HCR.TGE is 0 and the value of an HCR.{AMO, IMO, FMO} routing control bit is 1, the corresponding virtual interrupt is enabled and a virtual exception is generated either:

- By setting the corresponding virtual interrupt pending bit, HCR.{VA, VI, VF}, to 1.
- For a Virtual IRQ or Virtual FIQ, by an IMPLEMENTATION DEFINED mechanism. This might be a signal from an interrupt controller. See, for example, the ARM Generic Interrupt Controller Architecture Specification.

When the value of HCR_EL2.TGE is 1 all virtual interrupts are disabled.

When a virtual interrupt is disabled:

- It cannot be taken.
- It cannot be seen in the ISR.

In AArch32 state, a virtual exception is taken only from a Non-secure EL1 or EL0 mode. In any other mode, if the exception is generated it is not taken.

A virtual exception is taken in Non-secure state to the default mode for the corresponding physical exception. This means:

- A Virtual SError is taken to Non-secure Abort mode.
• A Virtual IRQ is taken to Non-secure IRQ mode.
• A Virtual FIQ is taken to Non-secure FIQ mode.

Table G1-15 summarizes the HCR bits that route asynchronous exceptions to EL2, and the bits that generate the virtual exceptions.

### Table G1-15 HCR bits controlling asynchronous exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Routing the physical exception to EL2</th>
<th>Generating the virtual exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>SError</td>
<td>HCR.AMO</td>
<td>HCR.VA</td>
</tr>
<tr>
<td>IRQ</td>
<td>HCR.IMO</td>
<td>HCR.VI</td>
</tr>
<tr>
<td>FIQ</td>
<td>HCR.FMO</td>
<td>HCR.VF</td>
</tr>
</tbody>
</table>

The HCR.{VA, VI, VF} bits generate a virtual exception only if set to 1 when the value of the corresponding HCR.{AMO, IMO, FMO} is 1.

Similarly, if the implementation also includes EL3, the HCR.{AMO, IMO, FMO} bits route the corresponding physical exception to Hyp mode only if the physical exception is not routed to Monitor mode by the SCR.{EA, IRQ, FIQ} bit. For more information, see Asynchronous exception routing controls.

When the value of an HCR.{AMO, IMO, FMO} control bit is 1, the corresponding mask bit in PSTATE:
• Does not mask the physical exception.
• Masks the virtual exception when the PE is executing in a Non-secure EL1 or EL0 mode.

Taking a Virtual Abort exception clears HCR.VA to zero. Taking a Virtual IRQ exception or a Virtual FIQ exception does not affect the value of HCR.VI or HCR.VF.

--- Note ---
This means that the exception handler for a Virtual IRQ exception or a Virtual FIQ exception must cause software that is executing at EL2 or EL3 to update the HCR to clear the appropriate virtual exception bit to 0.

See WFE wake-up events on page G1-8990 and Wait For Interrupt on page G1-8991 for information about how virtual exceptions affect wake up from power-saving states.

--- Note ---
A hypervisor can use virtual exceptions to signal exceptions to the current Guest OS. The Guest OS takes a virtual exception exactly as it would take the corresponding physical exception, and is unaware of any distinction between virtual exception and the corresponding physical exception.

### Effects of the HCR.{AMO, IMO, FMO} bits

As described in this section, the HCR.{AMO, IMO, FMO} bits are part of the mechanism for enabling the virtual exceptions. In addition, for exceptions generated in Non-secure state:
• As mentioned in this section, affect the routing of the exceptions. See Asynchronous exception routing controls.
• Affect the masking of the exceptions. See Asynchronous exception masking controls on page G1-8959.

### G1.15.2 Asynchronous exception routing controls

--- Note ---
This section describes the behavior when all Exception levels are using AArch32. For the differences when this is not the case see Asynchronous exception routing and masking with higher Exception levels using AArch64 on page G1-8961.
In an implementation that includes EL3 the following bits in the SCR control the routing of asynchronous exceptions:

**SCR.EA**  
When the value of this bit is 1, any SError interrupt is taken to EL3.

--- **Note** ---
Although this section describes the asynchronous exception routing controls, SCR.EA also controls the routing of synchronous External aborts, see *Routing of aborts taken to AArch32 state* on page G1-8948.

**SCR.FIQ**  
When the value of this bit is 1, any FIQ exception is taken to EL3.

**SCR.IRQ**  
When the value of this bit is 1, any IRQ exception is taken to EL3.

When EL3 is using AArch32 and the value of one of the SCR.{EA, FIQ, IRQ} bits is 1, the exception is taken to Monitor mode. Only Secure software can change the values of these bits.

In an implementation that includes EL2, the following bits in the HCR route asynchronous exceptions to EL2, for exceptions that are both:

- Taken from a Non-secure EL1 or EL0 mode.
- If the implementation also includes EL3, not configured, by the SCR.{EA, FIQ, IRQ} controls, to be taken to EL3.

**HCR.AMO**  
When the value of this bit is 1, an SError interrupt exception taken from a Non-secure EL1 or EL0 mode is taken to EL2, instead of to Non-secure Abort mode. If the implementation also includes EL3, this control applies only if the value of SCR.EA is 0. When the value of SCR.EA is 1, the value of the AMO bit is ignored.

**HCR.FMO**  
When the value of this bit is 1, an FIQ exception taken from a Non-secure EL1 or EL0 mode is taken to EL2, instead of to Non-secure FIQ mode. If the implementation also includes EL3, this control applies only if the value of SCR.FIQ is 0. When the value of SCR.FIQ is 1, the value of the FMO bit is ignored.

**HCR.IMO**  
When the value of this bit is 1, an IRQ exception taken from a Non-secure EL1 or EL0 mode is taken to EL2, instead of to Non-secure IRQ mode. If the implementation also includes EL3, this control applies only if the value of SCR.IRQ is 0. When the value of SCR.IRQ is 1, the value of the IMO bit is ignored.

When EL2 is using AArch32 and the value of one of the HCR.{AMO, FMO, IMO} bits is 1, the exception is taken to Hyp mode. Only software executing in Hyp mode, or Secure software executing at EL3 with SCR.NS set to 1, can change the values of these bits. If EL3 is using AArch32, this requires the Secure software to be executing in Monitor mode.

The HCR.{AMO, FMO, IMO} bits also affect the masking of asynchronous exceptions in Non-secure state, as described in *Asynchronous exception masking controls* on page G1-8959.

The SCR.{EA, FIQ, IRQ} and HCR.{AMO, FMO, IMO} bits have no effect on the routing of Virtual Abort, Virtual FIQ, and Virtual IRQ exceptions.

--- **Note** ---
When the PE is in Hyp mode:
- Physical asynchronous exceptions that are not routed to Monitor mode are taken to Hyp mode.
- Virtual exceptions are not signaled to the PE.

See also *Asynchronous exception behavior for exceptions taken from AArch32 state* on page G1-8956.
### G1.15.3 Asynchronous exception masking controls

**Note**

This section describes the behavior when all Exception levels are using AArch32. For the differences when this is not the case see *Asynchronous exception routing and masking with higher Exception levels using AArch64 on page G1-8961.*

The PSTATE.{A, I, F} bits can mask the taking of the corresponding exceptions from AArch32 state, as follows:
- PSTATE.A can mask SError interrupt exceptions.
- PSTATE.I can mask IRQ exceptions.
- PSTATE.F can mask FIQ exceptions.

In an implementation that does not include either of EL2 and EL3, setting one of these bits to 1 masks the corresponding exception, meaning the exception cannot be taken.

In an implementation that includes EL2, the HCR.{AMO, IMO, FMO} bits modify the masking of exceptions taken from Non-secure state.

Similarly, in an implementation that includes EL3, the SCR.{AW, FW} bits modify the masking of exceptions taken from Non-secure state by the PSTATE.{A, F} bits.

An implementation that includes only EL1 and EL0 does not provide any masking of the PSTATE.{A, I, F} bits. The following subsections describe the masking of these bits in other implementations:
- Asynchronous exception masking in an implementation that includes EL2 but not EL3.
- Asynchronous exception masking in an implementation that includes EL3 but not EL2.
- Asynchronous exception masking in an implementation that includes both EL2 and EL3.
- Summary of the asynchronous exception masking controls on page G1-8960.

#### Asynchronous exception masking in an implementation that includes EL2 but not EL3

The HCR.{AMO, IMO, FMO} bits modify the effect of the PSTATE.{A, I, F} bits. When the value of an HCR.{AMO, IMO, FMO} mask override bit is 1, the value of the corresponding PSTATE.{A, I, F} bit is ignored when the exception is taken from a Non-secure mode other than Hyp mode.

#### Asynchronous exception masking in an implementation that includes EL3 but not EL2

The SCR.{AW, FW} bits modify the effect of the PSTATE.{A, F} bits. When the value of one of the SCR.{AW, FW} bits is 0, the corresponding PSTATE bit is ignored when both of the follow apply:
- The corresponding exception is taken from Non-secure state.
- The value of the corresponding SCR.{EA, IRQ, FIQ} bit is 1, routing the exception to EL3. This means the exception is routed to Monitor mode if EL3 is using AArch32.

**Note**

Whenever the value of PSTATE.I is 1, IRQ exceptions are masked and cannot be taken.

#### Asynchronous exception masking in an implementation that includes both EL2 and EL3

When the value of an HCR.{AMO, IMO, FMO} mask override bit is 1, the value of the corresponding PSTATE.{A, I, F} bit is ignored when both of the following apply:
- The exception is taken from Non-secure state.
- Either:
  - The corresponding SCR.{EA, IRQ, FIQ} bit routes the exception to Monitor mode.
  - The exception is taken from a Non-secure mode other than Hyp mode.
In addition, when the value of an SCR.{AW, FW} bit is 0, the value of the corresponding PSTATE.{A, F} bit is ignored when all of the following apply:

- The exception is taken from Non-secure state.
- The corresponding SCR.{EA, FIQ} bit routes the exception to Monitor mode.
- The corresponding HCR.{AMO, FMO} mask override bit is set to 0.

### Summary of the asynchronous exception masking controls

The tables in this section show the masking controls for each of the PSTATE.{A, I, F} bits. For an implementation that does not include all of the Exception levels:

**If the implementation includes only EL1 and EL0**

The PSTATE bits cannot be masked. The behavior is as shown in the Secure row of the tables.

**If the implementation includes EL2 but not EL3**

The behavior is as shown in the Non-secure table rows when the control bits in the SCR are both 0.

**If the implementation includes EL3 but not EL2**

The behavior is as shown in the table rows where the control bit in the HCR is 0.

Table G1-16 shows the controls of the masking of SError interrupt exceptions by PSTATE.A.

<table>
<thead>
<tr>
<th>Security state</th>
<th>HCR.AMO</th>
<th>SCR.EA</th>
<th>SCR.AW</th>
<th>Mode</th>
<th>PSTATE.A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>Masks SError interrupt, when set to 1</td>
</tr>
<tr>
<td>Non-secure</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>x</td>
<td></td>
<td>Masks SError interrupt, when set to 1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>x</td>
<td>x</td>
<td></td>
<td>Masks SError interrupt, when set to 1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>Hyp</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>x</td>
<td>Hyp</td>
<td></td>
<td>Masks SError interrupt, when set to 1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>x</td>
<td>x</td>
<td></td>
<td>Ignored</td>
</tr>
</tbody>
</table>

Table G1-17 shows the controls of the masking of IRQ exceptions by PSTATE.I.

<table>
<thead>
<tr>
<th>Security state</th>
<th>HCR.IMO</th>
<th>SCR.IRQ</th>
<th>Mode</th>
<th>PSTATE.I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>Masks IQs, when set to 1</td>
</tr>
<tr>
<td>Non-secure</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>Masks IQs, when set to 1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>x</td>
<td>Not Hyp</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>x</td>
<td>Hyp</td>
<td>Masks IQs, when set to 1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>x</td>
<td></td>
<td>Ignored</td>
</tr>
</tbody>
</table>
Table G1-18 shows the controls of the masking of FIQ exceptions by PSTATE.F:

<table>
<thead>
<tr>
<th>Security state</th>
<th>HCR.FMO</th>
<th>SCR.FIQ</th>
<th>SCR.FW</th>
<th>Mode</th>
<th>PSTATE.F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>Masks FIQs, when set to 1</td>
</tr>
<tr>
<td>Non-secure</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>Masks FIQs, when set to 1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>x</td>
<td></td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td>Masks FIQs, when set to 1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>x</td>
<td></td>
<td>Hyp</td>
<td>Masks FIQs, when set to 1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>x</td>
<td></td>
<td>Mon</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

G1.15.4 Asynchronous exception routing and masking with higher Exception levels using AArch64

Asynchronous exception routing controls on page G1-8957 and Asynchronous exception masking controls on page G1-8959 give full descriptions of the routing and masking of the asynchronous exceptions when all Exception levels are using AArch32. However, when EL0 and EL1 are using AArch32:

- As already described, the SCR and HCR controls might be from Exception levels that are using AArch64.
- If EL3 is using AArch64, or EL2 is using AArch64, there are some changes to the asynchronous exception behaviors.

Therefore, the following sections summarize the asynchronous exception behaviors, taking account of the Execution state being used at EL2 and EL3:

- Summary of physical interrupt routing.
- Summary of physical interrupt masking on page G1-8962.

Summary of physical interrupt routing

The Table G1-19 on page G1-8962 shows the routing of physical FIQ, IRQ and SError interrupts when the highest Exception level is using AArch32.

In this table:

- SCR: This is the *Effective value* of a field in SCR.
- HCR: This is the *Effective value* of a field in HCR.
- FIQ IRQ EA: The *Effective value* of the field that handles the asynchronous exception type in SCR.
- FMO IMO AMO: The *Effective value* of the mask override field for the asynchronous exception type in HCR, if EL2 is using AArch32 or HCR_EL2 if EL2 is using AArch64.
- FIQ IRQ Abt: The exception is taken to the FIQ mode, the IRQ mode or the Abort mode according to the type of asynchronous exception.
- Hyp: The exception is taken to AArch32 Hyp mode.
- Mon: The exception is taken to AArch32 Monitor mode.
n/a This field does not exist, or the Exception level is not accessible in this configuration.

### Table G1-19 Routing of physical asynchronous exceptions

<table>
<thead>
<tr>
<th>Control bits</th>
<th>SCR</th>
<th>NS</th>
<th>FIQ IRQ EA</th>
<th>TGE</th>
<th>FMO IMO AMO</th>
<th>HCR</th>
<th>Target when taken from EL0</th>
<th>Target when taken from EL1</th>
<th>Target when taken from EL2</th>
<th>Target when taken from EL3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCR</td>
<td>NS</td>
<td>FIQ IRQ EA</td>
<td>TGE</td>
<td>FMO IMO AMO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>FIQ IRQ Abt</td>
<td>n/a</td>
<td>n/a</td>
<td>FIQ IRQ Abt</td>
<td>n/a</td>
<td>FIQ IRQ Abt</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>FIQ IRQ Abt</td>
<td>FIQ IRQ Abt</td>
<td>Hyp</td>
<td>FIQ IRQ Abt</td>
<td>FIQ IRQ Abt</td>
<td>Hyp</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Hyp</td>
<td>Hyp</td>
<td>Hyp</td>
<td>FIQ IRQ Abt</td>
<td>Hyp</td>
<td>n/a</td>
<td>FIQ IRQ Abt</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Mon</td>
<td>Mon</td>
<td>Mon</td>
<td>Mon</td>
<td>Mon</td>
<td>Mon</td>
<td>Mon</td>
<td>Mon</td>
<td>Mon</td>
</tr>
</tbody>
</table>

### Summary of physical interrupt masking

Table G1-20 on page G1-8962 shows the masking of physical FIQ, IRQ and SError interrupts when the highest Exception level is using AArch32.

In this table:

- **SCR**: This is the *Effective value* of a field in SCR.
- **HCR**: This is the *Effective value* of a field in HCR.
- **FIQ IRQ EA**: The *Effective value* of the field that handles the asynchronous exception type in SCR.
- **FMO IMO AMO**: The *Effective value* of the mask override field for the asynchronous exception type in HCR.
- **FW AW**: For FIQ interrupts, the SCR.FW field, and for SError interrupts, the SCR.AW field. For IRQ interrupts, there is no equivalent field, so the *Effective value* is 0 and rows where this cell is 1 should be ignored.
- **A**: When the interrupt is asserted, it is taken regardless of the value of the PSTATE mask bit.
- **B**: When the interrupt is asserted, it is subject to the corresponding PSTATE mask bit. If the value of the mask is 1, the interrupt is not taken. If the value of the mask is 0, the interrupt is taken.
- **n/a**: This field does not exist, or the Exception level is not accessible in this configuration.

### Table G1-20 Masking of physical asynchronous exceptions

<table>
<thead>
<tr>
<th>Control bits</th>
<th>Effect of the interrupt mask when executing at:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCR</td>
<td>HCR</td>
</tr>
<tr>
<td>NS</td>
<td>FW AW</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
</tr>
</tbody>
</table>
G1.15.5 Taking an interrupt or other exception during a multiple-register load or store

In AArch32 state, an interrupt cannot be taken during a sequence of memory accesses caused by a single load or store instruction, except that when FEAT_LSMAOC is implemented and the value of the applicable LSMAOE field is 0, an interrupt can be taken between two memory accesses made by a single AArch32 Load Multiple (LDM) or Store Multiple (STM) instruction.

The applicable LSMAOE field is the field in the SCTLR_EL1, SCTLR_EL2, HSCTLR, or SCTLR register that applies to the Exception level and Security state at which the LDM or STM instruction is executed.

When the value of the LSMAOE bit is 0 and an interrupt is taken between two memory accesses made by a single AArch32 LDM or STM instruction, then:

- For a load, any register being loaded by the instruction other than a register used in the generation of the address by the instruction or the PC, can contain an UNKNOWN value. Any register used in the generation of the address is restored to its initial value and the LR is set on the interrupt to a value consistent with returning to the instruction.
- For a store, any data location being stored to by the instruction can contain an UNKNOWN value.
- For either a load or store, if the instruction specifies writeback of the base address, then that register is restored to its initial value.

Armv8.2 deprecates software relying on interrupts not being taken during the sequence of memory accesses caused by a single load or store instruction.

### Table G1-20 Masking of physical asynchronous exceptions (continued)

<table>
<thead>
<tr>
<th>Control bits</th>
<th>Effect of the interrupt mask when executing at:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCR</td>
<td>HCR</td>
</tr>
<tr>
<td>NS FW AW FIQ IRQ EA TGE FMO IMO AMO EL0 EL1 EL2 EL3</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>x</td>
<td>A</td>
<td>n/a</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>x</td>
<td>A</td>
<td>n/a</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>x</td>
<td>A</td>
<td>n/a</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

G1.16 AArch32 state exception descriptions

Handling exceptions that are taken to an Exception level using AArch32 on page G1-8929 gives general information about exception handling. This section describes each of the exceptions, in the following subsections:

- Undefined Instruction exception.
- Monitor Trap exception on page G1-8966.
- Hyp Trap exception on page G1-8967.
- Supervisor Call (SVC) exception on page G1-8968.
- Secure Monitor Call (SMC) exception on page G1-8969.
- Hypervisor Call (HVC) exception on page G1-8970.
- Prefetch Abort exception on page G1-8971.
- Data Abort exception on page G1-8975.
- Virtual SError interrupt exception on page G1-8979.
- IRQ exception on page G1-8980.
- Virtual IRQ exception on page G1-8982.
- FIQ exception on page G1-8982.
- Virtual FIQ exception on page G1-8984.

Additional pseudocode functions for exception handling on page G1-8984 gives additional pseudocode that is used in the pseudocode descriptions of a number of the exceptions.

G1.16.1 Undefined Instruction exception

An Undefined Instruction exception might be caused by:

- A System register access, floating-point, or Advanced SIMD instruction that is not accessible because of the settings in one or more of the CPACR, NSACR, HCPTR, and DBGDSCRext.
- A System register access, floating-point, or Advanced SIMD instruction that is not implemented.
- A System register access, floating-point, or Advanced SIMD instruction that causes an exception during execution. This includes:
  - Trapped floating-point exceptions that are taken to AArch32, if an implementation supports these traps. See Floating-point exceptions and exception traps on page E1-7148.
  - Execution of certain floating-point instructions when one or both of the FPSCR. {Stride, Len} fields in nonzero, in an implementation in which those fields are RW. The description of FPEXC specifies the instructions to which this applies.
- An instruction that is UNDEFINED.

Note

The Undefined Instruction exception is taken using offset 0x04 in the Hyp, Secure, or Non-secure vector table. In the Monitor vector table this offset is used for the Monitor Trap exception. See Monitor Trap exception on page G1-8966 and The vector tables and exception offsets on page G1-8930.

By default, an Undefined Instruction exception is taken to Undefined mode, but an Undefined Instruction exception can be taken to EL2, meaning it is taken to Hyp mode if EL2 is using AArch32, see The PE mode to which the Undefined Instruction exception is taken on page G1-8965.

The Undefined Instruction exception can provide:

- Signaling of an illegal instruction execution.
- Lazy context switching of System registers.
The preferred return address for an Undefined Instruction exception is the address of the instruction that generated the exception. For an exception taken to AArch32 state, this return is performed as follows:

- If returning from Secure or Non-secure Undefined mode, the exception return uses the SPSR and LR_und values generated by the exception entry, as follows:
  - If SPSR.T is 0, indicating that the exception occurred in A32 state, the return uses an exception return instruction with a subtraction of 4.
  - If SPSR.T is 1, indicating that the exception occurred in T32 state, the return uses an exception return instruction with a subtraction of 2.
- If returning from Hyp mode, the exception return is performed by an ERET instruction, using the SPSR and ELR_hyp values generated by the exception entry.

For more information, see Exception return to an Exception level using AArch32 on page G1-8951.

--- Note

If handling the Undefined Instruction exception requires instruction emulation, followed by return to the next instruction after the instruction that caused the exception, the instruction emulator must use the instruction length to calculate the correct return address, and to calculate the updated values of the IT bits if necessary.

The PE mode to which the Undefined Instruction exception is taken

Figure G1-4 shows how the implementation, state, and configuration options determine the PE mode to which an Undefined Instruction exception is taken, when the exception is taken to an Exception level that is using AArch32.

---

Figure G1-4 The PE mode an Undefined Instruction exception is taken to in AArch32 state

See also UNPREDICTABLE cases when the value of HCR.TGE is 1 on page G1-8942.
Pseudocode description of taking the Undefined Instruction exception

The AArch32.UndefinedFault() pseudocode procedure determines whether the Undefined Instruction exception is taken to AArch32 state. If it is taken to AArch32 state, the AArch32.TakeUndefInstrException() pseudocode procedure describes how the PE takes the exception.

An Undefined Instruction exception is taken to an Exception level using AArch64 if either:

- It is generated in User mode when EL1 is using AArch64.
- It is generated in User mode when EL2 is enabled in the current Security state and is using AArch64 and the value of HCR_EL2.TGE is 1.

Conditional execution of undefined instructions

The conditional execution rules described in Conditional execution on page F1-7229 apply to all instructions. This includes undefined instructions and other instructions that would cause entry to the Undefined Instruction exception.

If such an instruction fails its condition check, the behavior depends on the potential cause of entry to the Undefined Instruction exception, as follows:

- If the potential cause is the execution of the instruction itself and depends on data values used by the instruction, the instruction executes as a NOP and does not cause an Undefined Instruction exception.
- In the following cases, it is IMPLEMENTATION DEFINED whether the instruction executes as a NOP or causes an Undefined Instruction exception:
  - The potential cause is the execution of an earlier System register access instruction, floating-point instruction, or Advanced SIMD instruction.
  - The potential cause is the execution of the instruction itself without dependence on the data values used by the instruction.

An implementation must handle all such cases in the same way.

Interaction of UNDEFINED instruction behavior with UNPREDICTABLE or CONSTRAINED UNPREDICTABLE instruction behavior

If this manual describes an instruction as both:

- UNPREDICTABLE and UNDEFINED then the instruction is UNPREDICTABLE.
- CONSTRAINED UNPREDICTABLE and UNDEFINED then the instruction is CONSTRAINED UNPREDICTABLE.

Note

An example of this is where both:

- An instruction, or instruction class, is made UNDEFINED by some general principle, or by a configuration field.
- A particular encoding of that instruction or instruction class is specified as CONSTRAINED UNPREDICTABLE.

G1.16.2 Monitor Trap exception

The Monitor Trap exception is implemented only as part of EL3, and can be generated only if EL3 is using AArch32.

Note

The Monitor Trap exception is taken using offset 0x04 in the Monitor vector table. In the other vector tables, this offset is used for the Undefined Instruction exception. See Undefined Instruction exception on page G1-8964 and The vector tables and exception offsets on page G1-8930.
A Monitor Trap exception is generated if the PE is running in a mode other than Monitor mode, and commits for execution a `WFI` or `WFE` instruction that would otherwise cause suspension of execution when:

- In the case of the `WFI` instruction, the value of the `SCR.TWI` bit is 1.
- In the case of the `WFE` instruction, the value of the `SCR.TWE` bit is 1.

**Note**

Since a `WFE` or `WFI` can complete at any time, even without a Wakeup event, the traps on `WFE` or `WFI` are not guaranteed to be taken, even if the `WFE` or `WFI` is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The preferred return address for a Monitor Trap exception is the address of the instruction that generated the exception. The exception return uses the `SPSR` and `LR_mon` values generated by the exception entry, as follows:

- If `SPSR.T` is 0, indicating that the exception occurred in A32 state, the return uses an exception return instruction with a subtraction of 4.
- If `SPSR.T` is 1, indicating that the exception occurred in T32 state, the return uses an exception return instruction with a subtraction of 2.

For more information, see *Exception return to an Exception level using AArch32* on page G1-8951.

**The PE mode to which the Monitor Trap exception is taken**

When EL3 is using AArch32, a Monitor Trap exception is taken to Monitor mode, using a vector offset of `0x04` from the Monitor exception base address.

**Pseudocode description of taking the Monitor Trap exception**

The `AArch32.TakeMonitorTrapException()` pseudocode procedure describes how the PE takes the exception.

### G1.16.3 Hyp Trap exception

The Hyp Trap exception provides the standard mechanism for trapping Guest OS functions to the hypervisor.

The Hyp Trap exception is implemented only as part of EL2 and can be generated only if EL2 is using AArch32.

A Hyp Trap exception is generated if the PE is running in a Non-secure mode other than Hyp mode, and commits for execution an instruction that is trapped to Hyp mode. Instruction traps are enabled by setting bits to 1 in the `HCR`, `HCPTR`, `HDCR`, or `HSTR`. For more information, see *EL2 configurable controls* on page G1-9010.

Traps to Hyp mode never apply in Secure state, regardless of the value of the `SCR.NS` bit.

The preferred return address for a Hyp Trap exception is the address of the trapped instruction. The exception return is performed by an `ERET` instruction, using the `SPSR` and `ELR_hyp` values generated by the exception entry.

**Note**

The `SPSR` and `ELR_hyp` values generated on exception entry can be used, without modification, for an exception return to re-execute the trapped instruction. If the exception handler emulates the trapped instruction, and must return to the following instruction, the emulation of the instruction must include modifying `ELR_hyp`, and possibly updating `SPSR_hyp`.

When the PE enters the handler for a Hyp Trap exception, the `HSR` holds syndrome information for the exception. For more information, see *Use of the HSR* on page G5-9265.

**The PE mode to which the Hyp Trap exception is taken**

A Hyp Trap exception is taken to Hyp mode, using a vector offset of `0x14` from the Hyp exception base address.
Pseudocode description of taking the Hyp Trap exception

The `AArch32.TakeHypTrapException()` pseudocode procedure describes how the PE takes the exception.

### G1.16.4 Supervisor Call (SVC) exception

The Supervisor Call instruction, SVC, requests a supervisor function, typically to request an operating system function. When EL1 is using AArch32, executing an SVC instruction causes the PE to enter Supervisor mode. For more information, see SVC on page F5-8055.

--- Note ---

In an implementation that includes EL2, when EL2 is using AArch32:

- When an SVC instruction is executed in Hyp mode, the Supervisor Call exception is taken to Hyp mode. For more information, see SVC on page F5-8055.

- When the HCR.TGE bit is set to 1, the Supervisor Call exception generated by execution of an SVC instruction in Non-secure User mode is routed to Hyp mode. For more information, see Supervisor Call exception, when the value of HCR.TGE is 1 on page G1-8945.

By default, a Supervisor Call exception that is taken to AArch32 state is taken to Supervisor mode, but a Supervisor Call exception can be taken to EL2, meaning it is taken to Hyp mode if EL2 is using AArch32, see The PE mode to which the Supervisor Call exception is taken.

The preferred return address for a Supervisor Call exception is the address of the next instruction after the SVC instruction. For an exception taken to AArch32 state, this return is performed as follows:

- If returning from Secure or Non-secure Supervisor mode, the exception return uses the SPSR and LR_svc values generated by the exception entry, in an exception return instruction without subtraction.

- If returning from Hyp mode, the exception return is performed by an ERET instruction, using the SPSR and ELR_hyp values generated by the exception entry.

For more information, see Exception return to an Exception level using AArch32 on page G1-8951.

### The PE mode to which the Supervisor Call exception is taken

Figure G1-5 on page G1-8969 shows how the implementation, state, and configuration options determine the PE mode to which a Supervisor Call exception is taken, when the exception is taken to an Exception level that is using AArch32.
Figure G1-5 The PE mode the Supervisor Call exception is taken to in AArch32 state

See also UNPREDICTABLE cases when the value of HCR.TGE is 1 on page G1-8942.

Pseudocode description of taking the Supervisor Call exception

The AArch32.CallSupervisor() pseudocode procedure determines whether the Supervisor Call exception is taken to AArch32 state. If it is taken to AArch32 state, the AArch32.TakeSVCException() pseudocode procedure describes how the PE takes the exception.

An Supervisor Call exception is taken to an Exception level using AArch64 if either:

• It is generated by executing an SVC instruction in User mode when EL1 is using AArch64.
• It is generated by executing an SVC instruction in Non-secure User mode when EL2 is using AArch64 and the value of HCR_EL2.TGE is 1.

G1.16.5 Secure Monitor Call (SMC) exception

The Secure Monitor Call exception is implemented only as part of EL3. When EL3 is using AArch32, the exception is taken to Monitor mode.

The Secure Monitor Call instruction, SMC, requests a Secure Monitor function. When EL3 is using AArch32, executing an SMC instruction causes the PE to enter Monitor mode. For more information, see SMC on page F5-7900.

Note

• In an implementation that includes EL2, execution of an SMC instruction in a Non-secure EL1 mode can be trapped to EL2. When EL2 is using AArch32, this means that when HCR.TSC 1, execution of an SMC instruction in a Non-secure EL1 mode generates a Hyp Trap Exception that is taken to Hyp mode. For more information, see Traps to Hyp mode of Non-secure EL1 execution of SMC instructions on page G1-9017.
• The Operation pseudocode in the description of the AArch32 SMC instruction, in SMC on page F5-7900, identifies cases where execution of the instruction generates an exception that is taken to EL3 using AArch64.

The preferred return address for a Secure Monitor Call exception is the address of the next instruction after the SMC instruction. For an exception taken to AArch32 state, this return is performed using the SPSR and LR_mon values generated by the exception entry, using an exception return instruction without a subtraction.

For more information, see Exception return to an Exception level using AArch32 on page G1-8951.

Note
For an exception taken to AArch32 state, the exception handler can return to the SMC instruction itself by returning using a subtraction of 4, without any adjustment to the SPSR.IT[7:0] bits. If it does this, the return occurs, then asynchronous exceptions might occur and be handled, then the SMC instruction is re-executed and another Secure Monitor Call exception occurs.

This relies on:
• The SMC instruction being used correctly, either outside an IT block or as the last instruction in an IT block, so that the SPSR.IT[7:0] bits indicate unconditional execution.
• The Secure Monitor Call handler not changing the result of the original conditional execution test for the SMC instruction.

The PE mode to which the Secure Monitor Call exception is taken
The Secure Monitor Call exception is supported only as part of EL3. When EL3 is using AArch32, a Secure Monitor Call exception is taken to Monitor mode, using vector offset 0x08 from the Monitor exception base address.

Note
• An SMC instruction that is trapped to Hyp mode because HCR.TSC is set to 1 generates a Hyp Trap exception, see The PE mode to which the Hyp Trap exception is taken on page G1-8967.
• If EL3 is using AArch64 then Security behavior in Exception levels using AArch32 when EL2 or EL3 are using AArch64 on page G1-8941 describes the effect of executing an SMC instruction at an Exception level that is using EL1.

Pseudocode description of taking the Secure Monitor Call exception
The AArch32.TakeSMCException() pseudocode procedure describes how the PE takes the exception when the exception is taken to an Exception level that is using AArch32.

G1.16.6 Hypervisor Call (HVC) exception
The Hypervisor Call exception is implemented only as part of EL2.

The Hypervisor Call instruction, HVC, requests a hypervisor function. When EL2 is using AArch32, executing an HVC instruction generates a Hypervisor Call exception that is taken to Hyp mode. For more information, see HVC on page F5-7576.

Note
• Execution of HVC instructions is disabled when the value of SCR.HCE is 0. Descriptions of HVC instruction execution elsewhere in this section assume the Effective value of SCR.HCE is 1.
• When EL2 is using AArch64 an HVC instruction executed in a Non-secure EL1 mode generates an exception that is taken to EL2 using AArch64.
The preferred return address for a Hypervisor Call exception is the address of the next instruction after the HVC instruction. The exception return is performed by an ERET instruction, using the SPSR and ELR_hyp values generated by the exception entry.

For more information, see Exception return to an Exception level using AArch32 on page G1-8951.

When EL2 is using AArch32, executing an HVC instruction transfers the immediate argument of the instruction to the HSR. The exception handler retrieves the argument from the HSR, and therefore does not have to access the original HVC instruction. For more information, see Use of the HSR on page G5-9265.

**The PE mode to which the Hypervisor Call exception is taken**

The Hypervisor Call exception is supported only as part of EL2. When EL2 is using AArch32, a Hypervisor Call exception is taken to Hyp mode, using a vector offset that depends on the mode from which the exception is taken, as Figure G1-6 shows. This offset is from the Hyp exception base address.

![Figure G1-6 The PE mode the Hypervisor Call exception is taken to in AArch32 state](image)

**Pseudocode description of taking the Hypervisor Call exception**

The `AArch32.CallHypervisor()` pseudocode procedure determines whether the valid execution of an HVC instruction in AArch32 state generates an exception that is taken to EL2 using AArch64, or generates a Hypervisor Call exception taken to Hyp mode. The `AArch32.TakeHVCException()` pseudocode procedure describes how the PE takes a Hypervisor Call exception.

**G1.16.7 Prefetch Abort exception**

A Prefetch Abort exception can be generated by:

- A synchronous memory abort on an instruction fetch.

  **Note**

  Asynchronous External aborts on instruction fetches are reported as SError interrupts using the Data Abort exception, see Data Abort exception on page G1-8975.

  A Prefetch Abort exception entry is synchronous to the instruction whose fetch aborted.

  For more information about memory aborts see VMSAv8-32 memory aborts on page G5-9238.

- A Breakpoint, Vector Catch or Breakpoint Instruction exception, see Chapter G2 AArch32 Self-hosted Debug.

  **Note**

  If an implementation fetches instructions speculatively, it must handle a synchronous abort on such an instruction fetch by:

  - Generating a Prefetch Abort exception only if the instruction would be executed in a simple sequential execution of the program.

  - Ignoring the abort if the instruction would not be executed in a simple sequential execution of the program.
By default, when EL1 is using AArch32, a Prefetch Abort exception is taken to Abort mode, but a Prefetch Abort exception can be taken to:

- EL2, meaning it is taken to Hyp mode if EL2 is using AArch32.
- EL3, meaning it is taken to Monitor mode if EL3 is using AArch32.

For more information:

- About cases where the Prefetch Abort exception is taken to an Exception level that is using AArch32, see *The PE mode to which the Prefetch Abort exception is taken* on page G1-8973.
- About cases where the Prefetch Abort generates an exception that is taken to an Exception level that is using AArch64, see *Pseudocode description of taking the Prefetch Abort exception* on page G1-8975.

The preferred return address for a Prefetch Abort exception is the address of the aborted instruction. For an exception taken to AArch32 state this return is performed as follows:

- If returning from a mode other than Hyp mode, using the SPSR and LR values generated by the exception entry, using an exception return instruction with a subtraction of 4. This means using:
  - SPSR_abt and LR_abt if returning from Abort mode.
  - SPSR_mon and LR_mon if returning from Monitor mode.
- If returning from Hyp mode, using the SPSR_hyp and ELR_hyp values generated by the exception entry, using an ERET instruction.

For more information about the handling of Prefetch Abort exceptions in AArch32 state see *Exception return to an Exception level using AArch32* on page G1-8951.

### Prefetch Abort exception reporting a PC alignment fault exception

A PC alignment fault exception that is taken to an Exception level that is using AArch32 is reported as a Prefetch Abort exception, and:

**If the exception is taken to EL1 using AArch32 or EL3 using AArch32**

- The IFSR indicates the cause of the exception:
  - If the value of TTBCR.EAE is 0, IFSR.FS takes the value \(0b00001\).
  - If the value of TTBCR.EAE is 1, IFSR.STATUS takes the value \(0b100001\).
- IFAR holds the value of the address that faulted, including the misaligned low order bit or bits.
- R14_abt holds the address that faulted, including the misaligned low order bit or bits, with the standard offset for a Prefetch Abort exception.

**If the exception is taken to EL2 using AArch32**

- HSR.EC takes the value \(0b100010\).
- HSR.IL is **UNKNOWN**.
- HSR.ISS is **RES0**.
- HIFAR and ELR_hyp each hold the value of the address that faulted, including the misaligned low order bit or bits.

For a PC alignment fault exception taken to an Exception level that is using AArch32:

- If the exception occurred because of the **CONSTRAINED UNPREDICTABLE** behavior of a branch to an unaligned PC value, as described in *Branching to an unaligned PC* on page K1-11564, then bit[0] of the faulting address is forced to zero, and therefore the misalignment is because the value of bit[1] of this address is 1.
- If the exception occurred on an exit from Debug state, as described in *Exiting Debug state* on page H2-10271, then it is **CONSTRAINED UNPREDICTABLE** whether bit[0] of the faulting address is forced to zero.
The PE mode to which the Prefetch Abort exception is taken

Figure G1-7 on page G1-8974 shows how the implementation, state, and configuration options determine the PE mode to which a Prefetch Abort exception is taken, when the exception is taken to an Exception level that is using AArch32.

Note

In this figure, the Effective value of HCR2.TEA is 0 if The Reliability, Availability, and Serviceability Extension is not implemented.
Figure G1-7 The PE mode the Prefetch Abort exception is taken to in AArch32 state

See also UNPREDICTABLE cases when the value of HCR.TGE is 1 on page G1-8942.
Pseudocode description of taking the Prefetch Abort exception

The AArch32.Abort() pseudocode function determines whether the Prefetch Abort condition generates an exception that is taken to an Exception level that is using AArch64, or generates a Prefetch Abort exception that is taken in AArch32 state. When the exception is taken in AArch32 state, the AArch32.TakePrefetchAbortException() pseudocode procedure describes how the PE takes the exception.

The exception is taken to an Exception level using AArch64 if one of the following applies:

- The exception is generated in User mode when EL1 is using AArch64.
- The implementation includes EL2, EL2 is using AArch64, and one of the following applies:
  — The value of HCR_EL2.TGE is 1 and the exception is generated in Non-secure User mode.
  — The value of MDCR_EL2.TDE is 1 and the exception is generated by a Debug exception in a Non-secure EL1 or Non-secure EL0 mode.
  — The exception is generated by a stage 2 fault during a stage 1 translation table walk using the AArch32 Non-secure EL1&0 translation regime.
- The implementation includes EL3, EL3 is using AArch64, the value of SCR_EL3.EA is 1. and the exception is generated by an External abort in AArch32 state.

G1.16.8 Data Abort exception

In AArch32 state, a Data Abort exception can be generated by:

- A synchronous abort on a data read or write memory access. Exception entry is synchronous to the instruction that generated the memory access.
- An SError interrupt. The SError interrupt might be caused by an External abort on a memory access, which can be any of:
  — A data read or write access.
  — An instruction fetch.
  — In a VMSA memory system, a translation table access.

Exception entry occurs asynchronously.

As described in Asynchronous exception masking controls on page G1-8959, SError interrupts can be masked. When this happens, a generated SError interrupt is not taken until it is not masked.

- A watchpoint, see Watchpoint exceptions on page G2-9079.

By default, when EL1 is using AArch32 a Data Abort exception is taken to Abort mode, but a Data Abort exception can be taken to:

- EL2, meaning it is taken to Hyp mode if EL2 is using AArch32.
- EL3, meaning it is taken to Monitor mode if EL3 is using AArch32.

For more information:

- About cases where the Data Abort exception is taken to an Exception level that is using AArch32 see The PE mode to which the Data Abort exception is taken on page G1-8976.
- About memory aborts in AArch32 state see VMSAv8-32 memory aborts on page G5-9238.
- About cases where the Data Abort generates an exception that is taken to an Exception level that is using AArch64 see Pseudocode description of taking the Data Abort exception on page G1-8978.

The preferred return address for a Data Abort exception is the address of the instruction that generated the aborting memory access, or the address of the instruction following the instruction boundary at which an SError interrupt exception was taken. For an exception taken to AArch32 state, this return is performed as follows:

- If returning from a mode other than Hyp mode, using the SPSR and LR values generated by the exception entry, using an exception return instruction with a subtraction of 8. This means using:
  — SPSR_abt and LR_abt if returning from Abort mode.
— SPSR_mon and LR_mon if returning from Monitor mode.

- If returning from Hyp mode, using the SPSR_hyp and ELR_hyp values generated by the exception entry, using an ERET instruction.

For more information about the handling of Data Abort exceptions in AArch32 state see Exception return to an Exception level using AArch32 on page G1-8951.

The PE mode to which the Data Abort exception is taken

Figure G1-8 on page G1-8977 shows the determination of the mode to which a Data Abort exception is taken when the exception is taken to an Exception level that is using AArch32.

Note

In this figure, the Effective value of HCR2.TEA is 0 if The Reliability, Availability, and Serviceability Extension is not implemented.
Figure G1-8 The PE mode the Data Abort exception is taken to in AArch32 state

See also UNPREDICTABLE cases when the value of HCR.TGE is 1 on page G1-8942.
Pseudocode description of taking the Data Abort exception

The AArch32.Abort() pseudocode function determines whether the Data Abort condition generates an exception that is taken to an Exception level that is using AArch64, or generates a Data Abort exception that is taken in AArch32 state. When the exception is taken in AArch32 state, the AArch32.TakeDataAbortException() pseudocode procedure describes how the PE takes the exception.

The exception is taken to an Exception level using AArch64 if one of the following applies:

- The exception is generated in User mode when EL1 is using AArch64.
- The implementation includes EL2, EL2 is using AArch64, and one of the following applies:
  - The value of HCR_EL2.TGE is 1 and the exception is generated in Non-secure User mode.
  - The value of MDCR_EL2.TDE is 1 and the exception is generated by a Debug exception in a Non-secure EL1 or Non-secure EL0 mode.
  - The exception is generated by a stage 2 fault during a stage 1 translation table walk using the AArch32 Non-secure EL1&0 translation regime.
- The implementation includes EL3, EL3 is using AArch64, the value of SCR_EL3.EA is 1. and the exception is generated by an External abort in AArch32 state.

Effects of data-aborted instructions

An instruction that accesses data memory can modify memory by storing one or more values. If the execution of such an instruction generates a Data Abort exception, or causes Debug state entry because of a watchpoint set on the location, the value of each memory location that the instruction stores to is:

- Unchanged for any location for which one of the following applies:
  - An Alignment fault is generated.
  - An MMU fault is generated.
  - A Watchpoint is generated.
  - An External abort is generated, if that External abort is taken synchronously.
- UNKNOWN for any location for which no exception and no debug event is generated.

If the access to a memory location generates an External abort that is taken asynchronously, it is outside the scope of the architecture to define the effect of the store on that memory location, because this depends on the system-specific nature of the External abort. However, in general, Arm recommends that such locations are unchanged.

For External aborts and Watchpoints, where in principle faulting could be identified at byte or halfword granularity, the size of a location in this definition is the size for which a memory access is single-copy atomic.

In AArch32 state, instructions that access data memory can modify registers in the following ways:

- By loading values into one or more of the general-purpose registers. The registers loaded can include the PC.
- By loading values into one or more of the registers in the Advanced SIMD and floating-point register file.
- By specifying base register writeback, in which the base register used in the address calculation has a modified value written to it. All instructions that support base register writeback have CONSTRAINED UNPREDICTABLE results if base register writeback is specified with the PC as the base register. Only general-purpose registers can be modified reliably in this way.
- By a direct transfer to or from the Debug Communication Channel (DCC) register, using the LDC and STC instructions. For more information, see Chapter H4 The Debug Communication Channel and Instruction Transfer Register.
  
  If the instruction that accesses the DCC registers is an LDC or STC instruction, UNKNOWN values are left in the Data Transfer Register and DCC flow-control flags.
- By modifying PSTATE.
If the execution of such an instruction generates a synchronous Data Abort exception, the following rules determine the values left in these registers:

- **On entry to the Data Abort exception handler:**
  - The PC value is the Data Abort vector address, see Exception vectors and the exception base address on page G1-8929.
  - The LR_abt value is determined from the address of the aborted instruction. Neither value is affected by the results of any load specified by the instruction.
- The base register is restored to its original value if either:
  - The aborted instruction is a load and the list of registers to be loaded includes the base register.
  - The base register is being written back.
- If the instruction only loads one general-purpose register the value in that register is unchanged.
- If the instruction loads more than one general-purpose register, UNKNOWN values are left in destination registers other than the PC and the base register of the instruction.
- If the instruction affects any registers in the Advanced SIMD and floating-point register file, UNKNOWN values are left in the registers that are affected.
- **PSTATE** bits that are not defined as updated on exception entry retain their current value.
- If the instruction is a STREX, STREX®, STREXH, or STREXD, <Rd> is not updated.

After taking a Data Abort exception, the state of the Exclusives monitors is UNKNOWN. Therefore, Arm strongly recommends that the abort handler performs a CLREX instruction, or a dummy STREX instruction, to clear the Exclusives monitor state.

An External abort might signal a data corruption to the PE. For example, a memory location might have been corrupted. The error that caused the External abort might have been propagated. The RAS Extension provides mechanisms for software to determine the extent of the corruption and contain propagation of the error. For more information, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile.

**The Arm abort model**

The abort model used by an Arm PE is described as a Base Restored Abort Model. This means that if a synchronous Data Abort exception is generated by executing an instruction that specifies base register writeback, the value in the base register is unchanged.

The abort model applies uniformly across all instructions.

**G1.16.9 Virtual SError interrupt exception**

The Virtual SError interrupt exception is implemented only as part of EL2 is enabled in the current Security state.

A Virtual SError interrupt exception is generated in AArch32 state if all of the following apply:

- The PE is in a mode other than Hyp mode.
- The value of PSTATE.A is 0.
- Either:
  - EL2 is using AArch32 and the values of the HCR.{TGE, AMO, VA} bits are {0, 1, 1}.
  - EL2 is using AArch64 and the values of the HCR_EL2.{TGE, AMO, VA} bits are {0, 1, 1}.

The preferred return address for a Virtual SError interrupt exception is the address of the instruction immediately after the instruction boundary where the exception was taken. For an exception taken to AArch32 state, this return is performed using the SPSR and LR_abt values generated by the exception entry, using an exception return instruction without subtraction.
The PE mode to which the Virtual SError interrupt exception is taken

The Virtual SError interrupt exception is taken using a vector offset of 0x10 from the Non-secure exception base address.

The conditions for generating a Virtual SError interrupt exception in AArch32 state mean the exception is:

- Taken from a EL1 or EL0 mode.
- Taken to Abort mode if EL1 is using AArch32.
- Taken to EL1, when EL0 is using AArch32 and EL1 is using AArch64.

For more information, see Virtual exceptions when an implementation includes EL2 on page G1-8956.

Note

Because a Virtual SError interrupt exception taken to AArch32 state is always taken to Abort mode, on exception entry the preferred return address is always saved to LR_abt.

Pseudocode description of taking the Virtual SError interrupt exception

The AArch32.TakeVirtualSErrorException() pseudocode procedure describes how the PE takes the exception.

G1.16.10 IRQ exception

The IRQ exception is generated by IMPLEMENTATION DEFINED means. Typically this is by asserting an IRQ interrupt request input to the PE.

When an IRQ exception is taken, exception entry is precise to an instruction boundary.

As described in Asynchronous exception masking controls on page G1-8959, IRQ exceptions can be masked. When this happens, a generated IRQ exception is not taken until it is not masked.

By default, when EL1 is using AArch32, an IRQ exception is taken to IRQ mode, but an IRQ exception can be taken to:

- EL2, meaning it is taken to Hyp mode if EL2 is using AArch32.
- EL3, meaning it is taken to Monitor mode if EL3 is using AArch32.

For more information:

- About cases where the exception is taken to an Exception level using AArch32 see The PE mode to which the physical IRQ exception is taken on page G1-8981.
- About cases where the exception is taken to an Exception level using AArch64 see Pseudocode description of taking the physical IRQ exception on page G1-8981.

The preferred return address for an IRQ exception is the address of the instruction following the instruction boundary at which the exception was taken. For an exception taken to AArch32 state this return is performed as follows:

- If returning from a mode other than Hyp mode, using the SPSR and LR values generated by the exception entry, using an exception return instruction with a subtraction of 4. This means using:
  - SPSR_irq and LR_irq if returning from IRQ mode.
  - SPSR_mon and LR_mon if returning from Monitor mode.
- If returning from Hyp mode, using the SPSR_hyp and ELR_hyp values generated by the exception entry, using an ERET instruction.

For more information, see Exception return to an Exception level using AArch32 on page G1-8951.
The PE mode to which the physical IRQ exception is taken

Figure G1-9 shows how the implementation, state, and configuration options determine the mode to which an IRQ exception is taken when the exception is taken to an Exception level that is using AArch32.

![Figure G1-9 The PE mode the IRQ exception is taken to in AArch32 state](image)

Pseudocode description of taking the physical IRQ exception

The `AArch32.TakePhysicalIRQException()` pseudocode procedure describes how the PE takes the exception. This procedure includes the case where the exception is taken to an Exception level that is using AArch64. This happens if one of the following applies:

- The exception is taken from User mode and EL1 is using AArch64. The Exception is taken to EL1 using AArch64.
- The exception is taken from User mode, EL2 is implemented in the current Security state and using AArch64, and the value of `HCR_EL2.TGE` is 1. The Exception is taken to EL2 using AArch64.
- The exception is taken from EL0 or EL1 mode, EL2 is implemented in the current Security state and using AArch64, and the value of `HCR_EL2.IMO` is 1. The Exception is taken to EL2 using AArch64.
- The exception is taken from a PE mode other than Monitor mode, EL3 is implemented and using AArch64, and the value of `SCR_EL3.IRQ` is 1. The Exception is taken to EL3 using AArch64.
G1.16.11  Virtual IRQ exception

The Virtual IRQ exception is implemented only as part of EL2, if EL2 is enabled in the current Security state.

A Virtual IRQ exception is generated in AArch32 state if all of the following apply:

• The PE is in a mode other than Hyp mode.
• The value of PSTATE.I is 0.
• Either:
  — EL2 is using AArch32 and the value of HCR.{TGE, IMO} is {0, 1}.
  — EL2 is using AArch64 and the value of HCR_EL2.{TGE, IMO} is {0, 1}.
• One of the following applies:
  — EL2 is using AArch32 and the value of HCR.VI is 1.
  — EL2 is using AArch64 and the value of HCR_EL2.VI is 1.
  — A Virtual IRQ exception is generated by an IMPLEMENTATION DEFINED mechanism.

The preferred return address for a Virtual IRQ exception is the address of the instruction immediately after the instruction boundary where the exception was taken. For an exception taken to AArch32 state this return is performed using the SPSR and LR_irq values generated by the exception entry, using an exception return instruction with a subtraction of 4.

The PE mode to which the Virtual IRQ exception is taken

The Virtual IRQ exception uses a vector offset of 0x18.

The conditions for generating a Virtual IRQ exception in AArch32 state mean the exception is:

• Taken from an EL1 or EL0 mode.
• Taken to IRQ mode if EL1 is using AArch32.
• Taken to EL1 if EL0 is using AArch32 and EL1 is using AArch64.

For more information, see Virtual exceptions when an implementation includes EL2 on page G1-8956.

Pseudocode description of taking the Virtual IRQ exception

The AArch32.TakeVirtualIRQException() pseudocode procedure describes how the PE takes the exception.

G1.16.12  FIQ exception

The FIQ exception is generated by IMPLEMENTATION DEFINED means. Typically this is by asserting an FIQ interrupt request input to the PE.

When an FIQ exception is taken, exception entry is precise to an instruction boundary.

As described in Asynchronous exception masking controls on page G1-8959, FIQ exceptions can be masked. When this happens, a generated FIQ exception is not taken until it is not masked.

By default, an FIQ exception is taken to FIQ mode, but an FIQ exception can be taken to:

• EL2, meaning it is taken to Hyp mode if EL2 is using AArch32.
• EL3, meaning it is taken to Monitor mode if EL3 is using AArch32.

For more information:

• About cases where the exception is taken to an Exception level using AArch32 see The PE mode to which the physical FIQ exception is taken on page G1-8983.
• About cases where the exception is taken to an Exception level using AArch64 see Pseudocode description of taking the FIQ exception on page G1-8984.
The preferred return address for an FIQ exception is the address of the instruction following the instruction boundary at which the exception was taken. For an exception taken to AArch32 state this return is performed as follows:

- If returning from a mode other than Hyp mode, using the SPSR and LR values generated by the exception entry, using an exception return instruction with a subtraction of 4. This means using:
  - SPSR_fiq and LR_fiq if returning from FIQ mode.
  - SPSR_mon and LR_mon if returning from Monitor mode.
- If returning from Hyp mode, using the SPSR_hyp and ELR_hyp values generated by the exception entry, using an ERET instruction.

For more information, see Exception return to an Exception level using AArch32 on page G1-8951.

**The PE mode to which the physical FIQ exception is taken**

Figure G1-9 on page G1-8981 shows how the implementation, state, and configuration options determine the PE mode to which an FIQ exception is taken when the exception is taken to an Exception level that is using AArch32.

![FIQ exception diagram](image)

Figure G1-10 The PE mode the FIQ exception is taken to in AArch32 state
Pseudocode description of taking the FIQ exception

The `AArch32.TakePhysicalFIQException()` pseudocode procedure describes how the PE takes the exception. This procedure includes the case where the exception is taken to an Exception level that is using AArch64. This happens if one of the following applies:

- The exception is taken from User mode and EL1 is using AArch64. The Exception is taken to EL1 using AArch64.
- The exception is taken from User mode, EL2 is implemented in the current Security state and using AArch64, and the value of `HCR_EL2.TGE` is 1. The Exception is taken to EL2 using AArch64.
- The exception is taken from an EL0 or EL1 mode, EL2 is implemented in the current Security state and using AArch64, and the value of `HCR_EL2.FMO` is 1. The Exception is taken to EL2 using AArch64.
- The exception is taken from a PE mode other than Monitor mode, EL3 is implemented and using AArch64, and the value of `SCR_EL3.FIQ` is 1. The Exception is taken to EL3 using AArch64.

G1.16.13 Virtual FIQ exception

The Virtual FIQ exception is implemented only as part of EL2, if EL2 is enabled in the current Security state.

A Virtual FIQ exception is generated in AArch32 state if all of the following apply:

- The PE is in a mode other than Hyp mode.
- The value of `PSTATE.F` is 0.
- Either:
  - EL2 is using AArch32 and the value of `HCR.{TGE, FMO}` is {0, 1}.
  - EL2 is using AArch64 and the value of `HCR_EL2.{TGE, FMO}` is {0, 1}.
- One of the following applies:
  - EL2 is using AArch32 and the value of `HCR.VF` is 1.
  - EL2 is using AArch64 and the value of `HCR_EL2.VF` is 1.
  - A Virtual FIQ exception is generated by an IMPLEMENTATION DEFINED mechanism.

The preferred return address for a Virtual FIQ exception is the address of the instruction immediately after the instruction boundary where the exception was taken. For an exception taken to AArch32 state this return is performed using the `SPSR` and `LR_irq` values generated by the exception entry, using an exception return instruction with a subtraction of 4.

The PE mode to which the Virtual FIQ exception is taken

The Virtual FIQ exception is taken using a vector offset of 0x1C.

The conditions for generating a Virtual FIQ exception in AArch32 state mean the exception is:

- Taken from EL1 or EL0.
- Taken to FIQ mode if EL1 is using AArch32.
- Taken to EL1 if EL0 is using AArch32 and EL1 is using AArch64.

For more information, see Virtual exceptions when an implementation includes EL2 on page G1-8956.

Pseudocode description of taking the Virtual FIQ exception

The `AArch32.TakeVirtualFIQException()` pseudocode procedure describes how the PE takes the exception.

G1.16.14 Additional pseudocode functions for exception handling

The `AArch32.EnterMonitorMode()` pseudocode function changes the PE mode to Monitor mode, with the required state changes.
The `AArch32.EnterHypMode()` pseudocode function changes the PE mode to Hyp mode, with the required state changes.

The `AArch32.EnterMode()` pseudocode function changes the PE mode to a PL1 mode, with the required state changes. It is used for all exceptions that are not routed to Hyp mode or Monitor mode.

The `AArch32.EnterMonitorMode()`, `AArch32.EnterHypMode()`, and `AArch32.EnterMode()` functions are described in Chapter J1 *Armv8 Pseudocode*. 
G1.17 Reset into AArch32 state

Resets and power domains on page D1-4673 describes the Armv8 reset model, including the defined levels of reset. When reset is deasserted, the PE starts executing instructions in the highest implemented Exception level. If that Exception level is using AArch32, then it starts execution:

• In Secure state, if the implementation includes EL3.
• With interrupts disabled:
  — In Hyp mode, if the highest implemented Exception level is EL2.
  — In Supervisor mode, otherwise.

--- Note ---

• This section describes the architectural requirements for a reset into AArch32 state. It takes no account of whether Arm licenses any particular combination of Exception levels and Execution state. For more information about the licensed combinations, see Exception levels on page D1-4632.

• The Execution state in which the highest implemented Execution level starts executing instructions on coming out of reset might be determined by a configuration input signal.

Reset returns some PE state to architecturally-defined or IMPLEMENTATION DEFINED values, and makes other state UNKNOWN, as described in PE state on reset into AArch32 state. For more information about behavior when resetting into an Exception level using AArch32, see:

• Behavior of caches at reset on page G4-9119.
• Enabling stages of address translation on page G5-9156.
• TLB behavior at reset on page G5-9217.
• Reset and debug on page H6-10348.

When reset is deasserted, if the PE resets into an Exception level that is using AArch32, it is IMPLEMENTATION DEFINED whether execution starts:

• From an IMPLEMENTATION DEFINED address.
• If reset is into EL3 or EL1, from the low or high reset vector address, as determined by the reset value of the SCTLR.V bit. This reset value can be determined by an IMPLEMENTATION DEFINED configuration input signal.

--- Note ---

This option might be implemented for compatibility with earlier versions of the architecture.

Software might be able to identify the reset address:

• If reset is into EL3, by reading the reset value of MVBAR. That is, after coming out of reset, by reading MVBAR before the boot software has updated it. It is IMPLEMENTATION DEFINED whether this discovery mechanism is supported.
• If reset is into EL2 or EL1, by reading RVBAR. RVBAR can only be implemented at the highest implemented Exception level, and only if that Exception level is not EL3.

If RVBAR is not implemented, and at all Exception levels other than the highest implemented Exception level, the encoding for RVBAR is UNDEFINED.

The Arm architecture does not define any way of returning to a previous Execution state from a reset.

G1.17.1 PE state on reset into AArch32 state

Immediately after a reset, much of the PE state is architecturally UNKNOWN. However, some of the PE state is defined, see the individual register descriptions for more information. The state that is reset to known values is sufficient to permit predictable initial execution at the highest Exception level, such that this execution is then capable of initializing the remaining state of the system where necessary before use.
If the PE resets to AArch32 state using either a Cold or a Warm reset, the PE state that is defined is as follows:

- The global exclusive monitor and local exclusive monitor for the PE are **UNKNOWN**.
- If reset is into EL2 using AArch32, then reset is into Hyp mode and PSTATE.M resets to `0b1010`, otherwise reset is into Supervisor mode and PSTATE.M resets to `0b0011`.
- PSTATE.IL resets to 0.

### G1.17.2 Pseudocode descriptions of reset

The `AArch32.TakeReset()` pseudocode procedure describes how the PE behaves when reset is deasserted.

The `AArch32.ResetGeneralRegisters()` pseudocode function resets the general-purpose registers.

The `AArch32.ResetSIMDFPRegisters()` pseudocode function resets the SIMD and floating-point registers.

The `AArch32.ResetSpecialRegisters()` pseudocode function resets the Special-purpose registers, and the debug System registers DLR and DSPSR, which are used for handling Debug exceptions.

The `AArch32.ResetSystemRegisters()` pseudocode function resets all System registers in the `(coproc==0b111x)` encoding space to their reset state as defined in the register descriptions in Chapter G8 AArch32 System Register Descriptions.

--- Note ---

The `AArch32.ResetSystemRegisters()` function only resets the System registers. It has no effect on memory-mapped registers.

--- Note ---

The `ResetExternalDebugRegisters()` pseudocode function resets all external debug registers to their reset state as defined in the register descriptions in Chapter H9 External Debug Register Descriptions.
G1.18 Mechanisms for entering a low-power state

The following sections describe the architectural mechanisms that a PE can use to request entry to a low-power state:

- Wait For Event and Send Event.
- Wait For Interrupt on page G1-8991.

G1.18.1 Wait For Event and Send Event

The Wait For Event (WFE) mechanism permits a PE to request entry to a low-power state, and, if the request succeeds, to remain in that state until an event is generated by a Send Event operation, or another WFE wake-up event occurs. Example G1-2 describes how a spinlock implementation might use this mechanism to save energy.

Example G1-2 Spinlock as an example of using Wait For Event and Send Event

A multiprocessor operating system requires locking mechanisms to protect data structures from being accessed simultaneously by multiple PEs. These mechanisms prevent the data structures becoming inconsistent or corrupted if different PEs try to make conflicting changes. If a lock is busy, because a data structure is being used by one PE, it might not be practical for another PE to do anything except wait for the lock to be released. For example, if a PE is handling an interrupt from a device it might need to add data received from the device to a queue. If another PE is removing data from the queue, it will have locked the memory area that holds the queue. The first PE cannot add the new data until the queue is in a consistent state and the lock has been released. It cannot return from the interrupt handler until the data has been added to the queue, so it must wait.

Typically, a spin-lock mechanism is used in these circumstances:

- A PE requiring access to the protected data attempts to obtain the lock using single-copy atomic synchronization primitives such as the Load-Exclusive and Store-Exclusive operations described in Synchronization and semaphores on page E2-7211.
- If the PE obtains the lock, it performs its memory operation and releases the lock.
- If the PE cannot obtain the lock, it reads the lock value repeatedly in a tight loop until the lock becomes available. At this point, it again attempts to obtain the lock.

A spin-lock mechanism is not ideal for all situations:

- In a low-power system, the tight read loop is undesirable because it uses energy to no effect.
- In a multithreaded implementation, the execution of spin-locks by waiting threads can significantly degrade overall performance.

Using the Wait For Event and Send Event mechanism can improve the energy efficiency of a spinlock. In this situation, a PE that fails to obtain a lock can execute a Wait For Event instruction, WFE, to request entry to a low-power state. When a PE releases a lock, it must execute a Send Event instruction, SEV, causing any waiting PEs to wake up. Then, these PEs can attempt to gain the lock again.

The execution of a WFE instruction can cause suspension of execution only if all of the following are true:

- The instruction does not cause any other exception.
- When the instruction is executed:
  — The Event Register is not set.
  — There is not a pending WFE wake-up event.

For more information about the trap to EL2, see Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions on page G1-9020.

The architecture does not define the exact nature of the low power state entered as a result of executing a WFE instruction, but the execution of a WFE instruction must not cause a loss of memory coherency or architectural state.
— Note ——-

Although a complex operating system can contain thousands of distinct locks, the event sent by this mechanism does not indicate which lock has been released. If the event relates to a different lock, or if another PE acquires the lock more quickly, the PE fails to acquire the lock and can reenter the low-power state waiting for the next event.

The Wait For Event system relies on hardware and software working together to achieve energy saving:

- The hardware provides the mechanism to enter the Wait For Event low-power state.
- The operating system software is responsible for issuing:
  - A Wait For Event instruction, to request entry to the low-power state, used in the example when waiting for a spin-lock.
  - A Send Event instruction, required in the example when releasing a spin-lock.

The mechanism depends on the interaction of:

- WFE wake-up events, see WFE wake-up events on page G1-8990.
- The Event Register, see The Event Register.
- The Send Event instructions, see The Send Event instructions on page G1-8990.
- The Wait For Event instruction, see The Wait For Event instruction.

The Event Register

The Event Register is a single bit register for each PE. When set, an event register indicates that an event has occurred, since the register was last cleared, that might require some action by the PE. Therefore, the PE must not suspend operation on issuing a WFE instruction.

The reset value of the Event Register is UNKNOWN.

The Event Register for a PE is set by:

- The execution of an SEV instruction on any PE in the multiprocessor system.
- The execution of an SEVL instruction by the PE.
- An exception return.
- An event from a Generic Timer event stream, see Event streams on page G6-9295.
- An event sent by some IMPLEMENTATION DEFINED mechanism.

As shown in this list, the Event Register might be set by IMPLEMENTATION DEFINED mechanisms.

The Event Register is cleared only by a Wait For Event instruction.

Software cannot read or write the value of the Event Register directly.

The Wait For Event instruction

The action of the Wait For Event instruction depends on the state of the Event Register:

- If the Event Register is set, the instruction clears the register and completes immediately. Normally, if this happens the software makes another attempt to claim the lock.
- If the Event Register is clear the PE can suspend execution, and hardware might enter a low-power state. The PE can remain suspended until a WFE wake-up event or a reset occurs. When a WFE wake-up event occurs, or earlier if the implementation chooses, the WFE instruction completes.

The execution in AArch32 state of a WFE instruction that would otherwise cause suspension of execution might be trapped, see:

- Traps to Undefined mode of EL0 execution of WFE and WFI instructions on page G1-9004.
- Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions on page G1-9020.
- Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode on page G1-9032.
The Wait For Event instruction, \( \text{WFE} \), is available at all privilege levels, see \( \text{WFE} \) on page F5-8154.

Software using the Wait For Event mechanism must tolerate spurious wake-up events, including multiple wake ups.

**WFE wake-up events**

The following events are \( \text{WFE} \) wake-up events:

- The execution of an \( \text{SEV} \) instruction on any PE in the system.
- The execution of an \( \text{SEVL} \) instruction on the PE.
- A physical IRQ interrupt, unless masked by the \( \text{PSTATE}.I \) bit.
- A physical FIQ interrupt, unless masked by the \( \text{PSTATE}.F \) bit.
- A physical SError interrupt, unless masked by the \( \text{PSTATE}.A \) bit.
- In Non-secure state in any mode other than Hyp mode:
  - When \( \text{HCR}.I \) is set to 1, a virtual IRQ interrupt, unless masked by the \( \text{PSTATE}.I \) bit.
  - When \( \text{HCR}.F \) is set to 1, a virtual FIQ interrupt, unless masked by the \( \text{PSTATE}.F \) bit.
  - When \( \text{HCR}.A \) is set to 1, a virtual SError interrupt, unless masked by the \( \text{PSTATE}.A \) bit.
- An asynchronous External Debug Request debug event, if halting is allowed. For the definition of halting is allowed, see \( \text{Halting allowed and halting prohibited} \) on page H2-10233.
  See also \( \text{External Debug Request debug event} \) on page H3-10291.
- An event sent by the timer event stream, see \( \text{Event streams} \) on page D10-5239.
- An event sent by some IMPLEMENTATION DEFINED mechanism.
- An event caused by the clearing of the global monitor associated with the PE.

In addition to the possible masking of WFE wake-up events shown in this list, when invasive debug is enabled and \( \text{EDSCR}.\text{HDE} \) is set to 1, \( \text{EDSCR}.\text{INTdis} \) can mask interrupts, including masking them acting as WFE wake-up events. See the register description for more information.

As shown in the list of wake-up events, an implementation can include IMPLEMENTATION DEFINED hardware mechanisms to generate wake-up events.

——— \textbf{Note} ————

For more information about \( \text{PSTATE} \) masking, see \( \text{Asynchronous exception masking controls} \) on page G1-8959. If the configuration of the masking controls provided by EL2 and EL3 mean that a \( \text{PSTATE} \) mask bit cannot mask the corresponding exception, then the physical exception is a WFE wake-up event, regardless of the value of the \( \text{PSTATE} \) mask bit.

——— \textbf{Note} ————

The Send Event instructions

The Send Event instructions are:

**SEV, Send Event**

This causes an event to be signaled to all PEs in the multiprocessor system.

**SEVL, Send Event Local**

This must set the local Event Register. It might signal an event to other PEs, but is not required to do so.

The mechanism that signals an event to other PEs is IMPLEMENTATION DEFINED. The PE is not required to guarantee the ordering of this event with respect to the completion of memory accesses by instructions before the \( \text{SEV} \) instruction. Therefore, Arm recommends that software includes a \text{DSB} instruction before any \text{SEV} instruction.

——— \textbf{Note} ————

A \text{DSB} instruction ensures that no instruction, including any \text{SEV} instruction, that appears in program order after the \text{DSB} instruction, can execute until the \text{DSB} instruction has completed. For more information, see \( \text{Data Synchronization Barrier (DSB)} \) on page E2-7182.

——— \textbf{Note} ————

The \text{SEVL} instruction appears to execute in program order relative to any subsequent \text{WFE} instruction executed on the same PE, without the need for any explicit insertion of barrier instructions.
Execution of the Send Event instruction sets the Event Register.
The Send Event instructions are available at all privilege levels.

**Pseudocode description of the Wait For Event mechanism**

This section defines pseudocode functions that describe the operation of the Wait For Event mechanism.

The `ClearEventRegister()` pseudocode procedure clears the Event Register of the current PE.

The `IsEventRegisterSet()` pseudocode function returns TRUE if the Event Register of the current PE is set and FALSE if it is clear.

The `WaitForEvent()` pseudocode procedure optionally suspends execution until a WFE wake-up event or reset occurs, or until some earlier time if the implementation chooses. It is IMPLEMENTATION DEFINED whether restarting execution after the period of suspension causes a `ClearEventRegister()` to occur.

The `SendEvent()` pseudocode procedure sets the Event Register of every PE in the system.

### G1.18.2 Wait For Interrupt

AArch32 state supports Wait For Interrupt through an instruction, `WFI`, which is provided in the A32 and T32 instruction sets. For more information, see `WFI` on page F5-8156.

When a PE issues a `WFI` instruction, its execution can be suspended, and a low-power state can be entered.

The execution in AArch32 state of a `WFI` instruction that would otherwise cause suspension of execution might be trapped, see:

- Traps to Undefined mode of EL0 execution of WFE and WFI instructions on page G1-9004.
- Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions on page G1-9020.
- Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode on page G1-9032.

The execution of a `WFI` instruction can cause suspension of execution only if both:

- The instruction does not cause any other exception.
- When the instruction is executed, there is not a pending WFI wake-up event.

**WFI wake-up events**

The PE can remain suspended in its WFI state until it is reset, or one of the following WFI wake-up events occurs:

- A physical IRQ interrupt, regardless of the value of the `PSTATE.I` bit.
- A physical FIQ interrupt, regardless of the value of the `PSTATE.F` bit.
- A physical SError interrupt, regardless of the value of the `PSTATE.A` bit.
- In Non-secure state in any mode other than Hyp mode:
  - When `HCR.IMO` is set to 1, a virtual IRQ interrupt, regardless of the value of the `PSTATE.I` bit.
  - When `HCR.FMO` is set to 1, a virtual FIQ interrupt, regardless of the value of the `PSTATE.F` bit.
  - When `HCR.AMO` is set to 1, a virtual SError interrupt, regardless of the value of the `PSTATE.A` bit.
- An asynchronous External Debug Request debug event, if halting is allowed. For the definition of halting is allowed, see Halting allowed and halting prohibited on page H2-10233.

See also External Debug Request debug event on page H3-10291.

An implementation can include other IMPLEMENTATION DEFINED hardware mechanisms to generate WFI wake-up events.

When a WFI wake-up event is detected, or earlier if the implementation chooses, the `WFI` instruction completes.

WFI wake-up events cannot be masked by the mask bits in the `PSTATE`.

The architecture does not define the exact nature of the low power state, except that:

- The execution of a `WFI` instruction must not cause a loss of memory coherency.
• If the system is configured such that the WFI instruction can be completed, then the WFI instruction must not cause a loss of architectural state.

--- Note ---
In some implementations, based on the configuration of system specific registers, WFI can be used as part of a powerdown sequence where no interrupts will cause WFI wake-up events, and restoration of power involves resetting of the PE. In those cases, the WFI is permitted to cause a loss of architectural state, as it is assumed that this state will have been saved by software as part of the powerdown sequence before the WFI.

--- Note ---
• Because debug events are WFI wake-up events, Arm strongly recommends that Wait For Interrupt is used as part of an idle loop rather than waiting for a single specific interrupt event to occur and then moving forward. This ensures the intervention of debug while waiting does not significantly change the function of the program being debugged.

• In some previous implementations of Wait For Interrupt, the idle loop is followed by exit functions that must be executed before taking the interrupt. The operation of Wait For Interrupt remains consistent with this model, and therefore differs from the operation of Wait For Event.

• Some implementations of Wait For Interrupt drain down any pending memory activity before suspending execution. The Arm architecture does not require this operation, and software must not rely on Wait For Interrupt operating in this way.

Using WFI to indicate an idle state on bus interfaces
A common implementation practice is to complete any entry into powerdown routines with a WFI instruction. Typically, the WFI instruction:
1. Forces the completion of execution of any instructions that are in progress, and of all associated bus activity.
2. Suspends the execution of instructions by the PE.

The control logic required to do this tracks the activity of the bus interfaces used by the PE. This means it can signal to an external power controller when there is no ongoing bus activity.

However, memory-mapped and external debug interface accesses to debug registers must continue to be processed while the PE is in the WFI state. The indication of idle state to the system normally only applies to the non-debug functional interfaces used by the PE, not the debug interfaces.

If FEAT_DoubleLock is implemented and the value of DBGOSDLR.DLK, the OS Double Lock status bit, is set to 1, this idle state must not be signaled to the PE unless the system can guarantee, also, that the debug interface is idle.

--- Note ---
When separate Core and Debug power domains are implemented, the debug interface referred to in this section is the interface between the Core and Debug power domains, since the signal to the power controller indicates that the Core power domain is idle. For more information about the power domains, see Power domains and debug on page H6-10335.

---

The exact nature of this interface is IMPLEMENTATION DEFINED, but the use of Wait For Interrupt as the only architecturally-defined mechanism that completely suspends execution makes it very suitable as the preferred powerdown entry mechanism.

Pseudocode description of Wait For Interrupt
The WaitForInterrupt() pseudocode function optionally suspends execution until a WFI wake-up event or reset occurs, or until some earlier time if the implementation chooses.
G1.19 The AArch32 System register interface

Most System registers are accessed using the instructions described in System register access instructions on page F2-7277. The System register interface provides access to those instructions, and:

- These registers are encoded using the parameters \{coproc, opc1, CRn, CRm, opc2\}, with permitted coproc values of 0b1110 and 0b1111.
- Some of these encodings provide the AArch32 System instructions.
- To maintain compatibility with previous versions of the Arm architecture, the access controls for the AArch32 System registers include the access controls for AArch32 Advanced SIMD and floating-point functionality.

--- Note ---

See Background to the System register interface on page G1-8994 for more information.

The following sections give more information about the AArch32 System register interface:

- System registers in the coproc == 0b111x encoding space.
- Access to System registers.
- Access controls for Advanced SIMD and floating-point functionality.
- Background to the System register interface on page G1-8994.

G1.19.1 System registers in the coproc == 0b111x encoding space

In AArch32 state:

- The coproc == 0b1110 encoding space is reserved for the configuration and control of:
  - Debug features, see Debug registers on page G8-9831.
  - Trace features, see the Embedded Trace Macrocell Architecture Specification.
  - Identification registers for the Trivial Jazelle implementation, see Trivial implementation of the Jazelle extension on page G1-8927.
- The coproc == 0b1111 encoding space is reserved for the control and configuration of the PE, including architecture and feature identification. This means these encodings provide access to the System registers that control and return status information for PE operation.

For more information, see Chapter G8 AArch32 System Register Descriptions.

G1.19.2 Access to System registers

Most System registers are accessible only from EL1 or higher. For possible accesses from EL0 the register descriptions in Chapter G8 AArch32 System Register Descriptions indicate whether a register is accessible from EL0.

G1.19.3 Access controls for Advanced SIMD and floating-point functionality

The CPACR controls access to Advanced SIMD and floating-point functionality from software executing at PL1 or EL0 in AArch32 state:

- The \{cp10, cp11\} fields control access to all Advanced SIMD and floating-point functionality, and can:
  - Disable EL0 and PL1 access to this functionality.
  - Enable access to this functionality at PL1 only.
  - Enable access to this functionality at EL0 and PL1.
- The ASEDIS field controls access to Advanced SIMD instructions that are not also floating-point instructions.
Initially on powerup or reset into AArch32 state, all access to all Advanced SIMD and floating-point functionality from PL1 and EL0 is disabled.

Note
The CPACR has no effect on accesses from Hyp mode.

If an implementation includes EL3, the NSACR determines whether Advanced SIMD and floating-point functionality can be accessed from Non-secure state:
- The \( \{cp10, cp11\} \) fields control Non-secure access to all Advanced SIMD and floating-point functionality.
- The NSASEDIS field controls Non-secure access to Advanced SIMD instructions that are not also floating-point instructions.

If an implementation includes EL2, the HCPTR provides additional controls on Non-secure accesses to Advanced SIMD and floating-point functionality. For accesses that are otherwise permitted by the CPACR and NSACR settings, setting HCPTR bits to 1:
- Traps otherwise-permitted accesses from EL1 or EL0 to EL2. When EL2 is using AArch32, these accesses are trapped to Hyp mode.
- Makes accesses from EL2 mode UNDEFINED. When EL2 is using AArch32, this makes accesses from Hyp mode UNDEFINED.

In the HCPTR:
- The \( \{TCP10, TCP11\} \) fields control access to all Advanced SIMD and floating-point functionality.
- The TASE field controls access to Advanced SIMD instructions that are not also floating-point instructions.
- The TCPAC field traps Non-secure EL1 accesses to the CPACR to Hyp mode.

For more information, see General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers on page G1-9021.

Note
Whenever a pair of fields control the access to the Advanced SIMD and floating-point functionality, the values of each field of the pair must be identical. If these settings are not identical the behavior of the Advanced SIMD and floating-point functionality is CONSTRAINED UNPREDICTABLE, see Handling of System register control fields for Advanced SIMD and floating-point operation on page K1-11568.

For more information about Advanced SIMD and floating-point support, see Advanced SIMD and floating-point support on page G1-8996.

G1.19.4 Background to the System register interface

Note
This section is not part of the Architecture specification. It is included only to present the rationale of some aspects of the System register interface.

The interface to the System registers was originally defined as part of a generic coprocessor interface that gave access to 15 coprocessors, CP0 - CP15. Of these, CP8 - CP15 were reserved for use by Arm, while CP0 - CP7 were available for IMPLEMENTATION DEFINED coprocessors.

The coprocessors were accessed using coprocessor instructions. These instructions remain part of the T32 and A32 instruction sets, see System register access instructions on page F2-7277.

In the Arm coprocessor model, a coprocessor included both:
- Primary and secondary coprocessor registers, which form part of the coprocessor interface.
- A number of internal registers.
When accessing a 32-bit internal coprocessor register, using an MCR or MRC instruction, the instruction specified:

- The target coprocessor, specified by the coproc parameter and taking a value between \( p0 \) for CP0 and \( p15 \) for CP15.
- The primary coprocessor register, specified by the \( CRn \) parameter and taking a value between \( c0 \) and \( c15 \).
- The secondary coprocessor register, specified by the \( CRn \) parameter and taking a value between \( c0 \) and \( c15 \).
- Up to two additional parameters, \( opc1 \) and \( opc2 \), taking values between 0 and 7.

Other instructions in the group described in System register access instructions on page F2-7277 take a subset of these parameters:

- In the Armv7 definitions, LDC and STC instructions take parameters \{coproc, CRd\}, where \( CRd \) is the primary coprocessor register.
- MCR and MRRC instructions take parameters \{coproc, opc1, CRn\}, where \( CRn \) is the primary coprocessor register.

To maintain backwards compatibility, the arguments to an MCR or MRC instruction remain \{coproc, opc1, CRn, CRm, opc2\}. Correspondingly, the encoding of the AArch64 System registers is described using the parameters \{op0, op1, CRn, CRm, op2\}. However:

- The naming of these parameters no longer has any particular significance.
- While the coproc field is a 4-bit field, \( op0 \) is a 2-bit field.

Of the coprocessors reserved for use by Arm, in earlier versions of the architecture:

- CP15 provided access to the System registers relating to non-debug operation, and was originally called the System control coprocessor. From the introduction of Armv8, these registers are described as being in the \( \text{coproc} = \text{0b1111} \) encoding space.
- CP14 provided access to additional System registers, including those relating to debug and trace. From the introduction of Armv8, these registers are described as being in the \( \text{coproc} = \text{0b1110} \) encoding space.
- CP10 and CP11 were used for Advanced SIMD and floating-point control, and many coprocessor instruction encodings targeting CP10 and CP11 were used as floating-point instruction encodings:
  - Generally from the introduction of Armv8 the architecture does not relate these instructions to the coprocessor encoding space, but the naming of registers and register fields for Advanced SIMD and floating-point control reflects the historic coprocessor model.
  - Because the Advanced SIMD and floating-point functionality used both CP10 and CP11, some System register controls of this functionality have a pair of fields, for example NSACR.\{cp10, cp11\}. In these cases, both fields must be set to the same value. For more information, see Access controls for Advanced SIMD and floating-point functionality on page G1-8993.

From the introduction of Armv8:

- The AArch32 System registers include registers that were described as Special registers in Armv7 and earlier versions of the architecture. This means that the System registers include registers that are outside the earlier coprocessor model.
- The Armv7 AArch32 instruction encodings for LDC, STC, MCR, MRC, MCRR, and MRRC instructions with coproc field values other than \{1010, 1011, 1110, 1111\} are available for reuse. Armv8.2 re-uses some encodings in this way.
G1.20 Advanced SIMD and floating-point support

Advanced SIMD and floating-point instructions on page E1-7140 introduces:

- The scalar floating-point instructions in the A32 and T32 instruction sets.
- The Advanced SIMD integer and floating-point vector instructions in the A32 and T32 instruction sets.
- The SIMD and floating-point register file, which can be viewed as:
  - Singleword registers S0 - S31.
  - Doubleword registers D0 - D31.
  - Quadword registers Q0 - Q15.
- The Floating-Point Status and Control Register (FPSCR).
- Floating-point exceptions and exception traps on page E1-7148

For more information about the System registers for the Advanced SIMD and floating-point operation, see Advanced SIMD and floating-point System registers on page G1-8998. Software can interrogate these registers to discover the implemented Advanced SIMD and floating-point support.

AArch32 implications of not including support for Advanced SIMD and floating-point summarizes the effects of not supporting these instructions, and the following subsections give more information about the Advanced SIMD and Floating-point support:

- Enabling Advanced SIMD and floating-point support.
- Advanced SIMD and floating-point System registers on page G1-8998.
- Context switching when using Advanced SIMD and floating-point functionality on page G1-8999.

G1.20.1 AArch32 implications of not including support for Advanced SIMD and floating-point

The architecture generally requires the inclusion of the Advanced SIMD and floating-point instructions in all instruction sets. However, for implementations targeting specialized markets, Arm might produce or license implementations that do not provide any support for Advanced SIMD and floating-point instructions. In such an implementation, in AArch32 state:

- The CPACR.{ASEDIS, cp11, cp10} fields are RES0.
- The NSACR.{NSASEDIS, cp11, cp10} fields are RES0.
- The HCPTR.{TASE, TCP11, TCP10} fields are RES1.
- The FPEXC, FPSCR, FPSID, MVFR0, MVFR1, and MVFR2 registers are not implemented and their encodings are UNDEFINED.
- Attempted accesses to Advanced SIMD and floating-point functionality are UNDEFINED. This means:
  - All Advanced SIMD and floating-point instructions are UNDEFINED.
  - Attempts to access the Advanced SIMD and floating-point System registers are UNDEFINED.

G1.20.2 Enabling Advanced SIMD and floating-point support

Software must ensure that the required access to the Advanced SIMD and floating-point features is enabled. Most of those controls are described in Configurable instruction enables and disables, and trap controls on page G1-9001, and this section:

- Summarizes those controls.
- Provides additional information in the following subsections:
  - FPEXC control of access to Advanced SIMD and floating-point functionality on page G1-8998.
  - EL0 access to Advanced SIMD and floating-point functionality on page G1-8998.

Note

This section shows the controls when the controlling Exception levels are using AArch32. Similar controls are provided when the Exception levels are using AArch64, and then apply to lower Exception levels that are using AArch32.
The controls of access to Advanced SIMD and floating-point functionality are:

**General \{cp10, cp11\} or \{TCP10, TCP11\} controls**

This relates to the CPACR.\{cp10, cp11\}, NSACR.\{cp10, cp11\}, and HCPTR.\{TCP10, TCP11\} controls.

--- **Note** ---

*Background to the System register interface* on page G1-8994 explains the naming of these controls.

The \{cp10, cp11\} controls provide general control of the use of Advanced SIMD and floating-point functionality, as follows:

- **CPACR.\{cp10, cp11\}** control access from PE modes other than Hyp mode. These fields have no effect on accesses to Advanced SIMD and floating-point functionality from Hyp mode.
- **In an implementation that includes EL3, NSACR.\{cp10, cp11\}** control access from Non-secure state.
- **In an implementation that includes EL2, if NSACR.\{cp10, cp11\}** permit Non-secure accesses, or if EL3 is not implemented, HCPTR.\{TCP10, TCP11\} provide an additional control on those accesses.

In each case, the \{cp10, cp11\} controls must be programmed to the same value, otherwise operation is **CONSTRAINED UNPREDICTABLE**. The CONSTRAINED UNPREDICTABLE behavior is that, for all purposes other than reading the value of the register field, behavior is as if the cp11 field has the same value as the cp10 field. For more information, see *Handling of System register control fields for Advanced SIMD and floating-point operation* on page K1-11568.

For more information about these controls, see:

- Enabling PL0 and PL1 accesses to the SIMD and floating-point registers on page G1-9006.
- General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers on page G1-9021.
- Enabling Non-secure access to SIMD and floating-point functionality on page G1-9034.

**Control of accesses to the CPACR from Non-secure PL1 modes**

As stated in **General \{cp10, cp11\} or \{TCP10, TCP11\} controls**, the CPACR controls access to Advanced SIMD and floating-point functionality from PE modes other than Hyp mode. Accesses to the CPACR from Non-secure PL1 modes can be trapped to EL2, see *Traps to Hyp mode of Non-secure EL1 accesses to the CPACR* on page G1-9023.

**Additional controls of Advanced SIMD functionality**

- If implemented as an RW field, CPACR.ASEDIS can make all Advanced SIMD instructions UNDEFINED in all modes other than Hyp mode.
- **In an implementation that includes EL3, when CPACR.ASEDIS** permits use of the Advanced SIMD instructions or if the CPACR.ASEDIS control is not implemented, NSACR.NSASEDIS can make all Advanced SIMD instructions UNDEFINED in Non-secure state.
- **In an implementation that includes EL2, when the CPACR and NSACR** settings permit Non-secure use of the Advanced SIMD instructions, if HCPTR.TASE is implemented as an RW field it can make these instructions UNDEFINED in Hyp mode, and trap to Hyp mode any use of these instructions in a Non-secure PL0 or PL1 mode.

For more information about these controls, see:

- Disabling PL0 and PL1 execution of Advanced SIMD instructions on page G1-9007.
- Traps to Hyp mode of Non-secure accesses to Advanced SIMD functionality on page G1-9022.
- Disabling Non-secure access to Advanced SIMD functionality on page G1-9034.
Pseudocode description of enabling SIMD and floating-point functionality on page G1-9035 provides links to the pseudocode descriptions of all of these controls.

**FPEXC control of access to Advanced SIMD and floating-point functionality**

In addition, FPEXC.EN is an enable bit for most Advanced SIMD and floating-point operations. When FPEXC.EN is 0, all Advanced SIMD and floating-point instructions are treated as UNDEFINED except for:

- A VMSR to the FPEXC or FPSID register.
- A VMRS from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2 register.

These instructions can be executed only at EL1 or higher.

--- Note ---

- When the FPSID is accessible, any write access to the FPSID is ignored.
- When FPEXC.EN is 0, these operations are treated as UNDEFINED:
  - A VMSR to the FPSCR.
  - A VMRS from the FPSCR.

---

See Enabling access to the SIMD and floating-point registers on page G1-9007 for more information about the scope of the FPEXC.EN control.

When executing at EL0, the PE behaves as if the value of FPEXC.EN is 1 if either:

- EL1 is using AArch64.
- EL2 is enabled in the current Security state and is using AArch64 and the value of HCR_EL2.TGE is 1.

--- Note ---

In Non-secure state, if the value of HCR_EL2.RW is 0 then it is permitted for the value of FPEXC32_EL2.EN to control whether Advanced SIMD and floating-point functionality is enabled. However, Arm deprecates using the value of FPEXC32_EL2.EN to determine behavior.

**EL0 access to Advanced SIMD and floating-point functionality**

When the access controls summarized in this section permit EL0 access to the Advanced SIMD and floating-point functionality, this applies only to the subset of functionality that is available at EL0. In particular:

- Only Advanced SIMD and Floating-point System register that is accessible is the FPSCR.
- The Advanced SIMD and floating-point instructions are available.

Execution at EL0 corresponds to the application level view of the Advanced SIMD and floating-point functionality, as described in Advanced SIMD and floating-point System registers on page E1-7142.

**G1.20.3 Advanced SIMD and floating-point System registers**

AArch32 state provides a common set of System registers for the Advanced SIMD and floating-point functionality. This section gives general information about this set of registers, and indicates where each register is described in detail. It contains the following subsections:

- Register map of the Advanced SIMD and floating-point System registers on page G1-8999.
- Accessing the Advanced SIMD and floating-point System registers on page G1-8999.
Register map of the Advanced SIMD and floating-point System registers

Table G1-21 shows the register map of the Advanced SIMD and Floating-point registers. Each register is 32 bits wide.

<table>
<thead>
<tr>
<th>Name</th>
<th>Permitted access</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPEXC</td>
<td>RW</td>
</tr>
<tr>
<td>FPSCR</td>
<td>RW</td>
</tr>
<tr>
<td>FPSID</td>
<td>RW(^a)</td>
</tr>
<tr>
<td>MVFR0</td>
<td>RO</td>
</tr>
<tr>
<td>MVFR1</td>
<td>RO</td>
</tr>
<tr>
<td>MVFR2</td>
<td>RO</td>
</tr>
</tbody>
</table>

\(^a\) When FPSID is accessible, VMSR accesses to FPSID are ignored.

In an implementation that includes EL3, the Advanced SIMD and Floating-point registers are common registers, see Common System registers on page G5-9284.

Accessing the Advanced SIMD and floating-point System registers

Software accesses the Advanced SIMD and floating-point System registers using the VMRS and VMSR instructions, see:

- **VMRS on page F6-8565.**
- **VMSR on page F6-8568.**

For example:

- `VMRS <Rt>, FPSID` ; Read Floating-Point System ID Register
- `VMRS <Rt>, MVFR1` ; Read Media and VFP Feature Register 1
- `VMSR FPSCR, <Rt>` ; Write Floating-Point System Control Register

Software can access the Advanced SIMD and floating-point System registers only if the access controls permit the access, see Enabling Advanced SIMD and floating-point support on page G1-8996.

Note

All hardware ID information can be accessed only from EL1 or higher. This means:

- **The FPSID is accessible only from EL1 or higher.**
  
  This is a change introduced from VFPv3. Previously, the FPSID register can be accessed in all modes.

- **The MVFR registers are accessible only from EL1 or higher.**
  
  Unprivileged software must issue a system call to determine what features are supported.

G1.20.4 Context switching when using Advanced SIMD and floating-point functionality

When the Advanced SIMD and floating-point functionality is used by only a subset of processes, the operating system might implement lazy context switching of the Advanced SIMD and floating-point register file and System registers.
In the simplest lazy context switch implementation, the primary context switch software uses the CPACR. {cp10, cp11} controls to disable access to the Advanced SIMD and floating-point functionality, see Enabling Advanced SIMD and floating-point support on page G1-8996. Subsequently, when a process or thread attempts to use an Advanced SIMD or floating-point instruction, it triggers an Undefined Instruction exception. The operating system responds by saving and restoring the Advanced SIMD and floating-point register file and System registers. Typically, it then re-executes the Advanced SIMD or floating-point instruction that generated the Undefined Instruction exception.
G1.21 Configurable instruction enables and disables, and trap controls

This section describes the controls provided by AArch32 state for enabling, disabling, and trapping particular instructions. Each control is categorized as an instruction enable, an instruction disable, or a trap control:

Instruction enables and instruction disables

Enable or disable the use of one or more particular instructions at a particular Privilege level and Security state.

When an instruction is disabled as a result of an instruction enable or disable, it is UNDEFINED. The exception generated by attempting to execute an UNDEFINED instruction is:

- Taken to EL1 if the UNDEFINED instruction was executed at EL0, unless the instruction was executed at Non-secure EL0 and is routed to EL2 by the control described in Routing exceptions from Non-secure EL0 to EL2 on page G1-8944. When the exception is taken to EL1, it is taken to Undefined mode.
- Otherwise, taken to the Exception level at which the UNDEFINED instruction was executed:
  - If the instruction was executed in Hyp mode the exception is taken to Hyp mode.
  - Otherwise, the exception is taken to Undefined mode.

Trap controls

Control whether one or more instructions, when executed at a particular Privilege level, are trapped.

Note

AArch32 trap controls are described in terms of Privilege levels, rather than Exception levels, because the PL1 traps apply at and are controlled from:

EL1 In Non-secure state, and in Secure state when EL3 is using AArch64.
EL3 In Secure state when EL3 is using AArch32.

For more information, see Security state, Exception levels, and AArch32 execution privilege on page G1-8908.

Trap controls are grouped as:

PL1, excluding Monitor mode

Trapped instructions generate Undefined Instruction exceptions that are taken to Undefined mode, unless the instruction was executed at Non-secure EL0 and is routed to EL2 by the control described in Routing exceptions from Non-secure EL0 to EL2 on page G1-8944. For more information about these traps, see PL1 configurable controls on page G1-9002.

Hyp mode (PL2)

These traps apply only to execution in Non-secure state. This section only describes the traps that apply when EL2 is using AArch32.

Trapped instructions generate:

- Hyp Trap exceptions, taken to Hyp mode, if trapped from a mode other than Hyp mode.
- Undefined Instruction exceptions taken to Hyp mode, if trapped from Hyp mode.

For more information about these traps, see EL2 configurable controls on page G1-9010. See also Routing exceptions from Non-secure EL0 to EL2 on page G1-8944.

Monitor mode (Secure PL1)

This section describes only the traps that apply when EL3 is using AArch32.

Trapped instructions generate Monitor Trap exceptions, which are taken to Monitor mode.

For more information about these traps, see EL3 configurable controls on page G1-9030.
An exception generated as a result of an instruction enable or disable, or a trap control, is only taken if the instruction does not also generate a higher priority exception. Exception prioritization for exceptions taken to AArch32 state on page G1-8932 defines the prioritization of different exceptions on the same instruction.

Exceptions generated as a result of these controls are synchronous exceptions.

For exceptions taken to an Exception level that is using AArch32, only exceptions that are taken to Hyp mode are reported in a syndrome register, the HSR.

--- Note ---

- A particular control might have a mnemonic that suggests it is different type of control to the control type it is categorized as. For example, CPACR.TRCDIS is a trap control even though TRCDIS is a mnemonic for Trace Disable.

- An implementation might provide additional controls, in IMPLEMENTATION DEFINED registers, to provide control of trapping of IMPLEMENTATION DEFINED features.

- Configurable instruction controls on page D1-4665 describes controls provided by AArch64 state for enabling, disabling, and trapping instructions. Generally, where an AArch64 control applies to execution at lower Exception levels, it traps the equivalent functionality when that lower Exception level is using AArch32. See the AArch64 trap controls for more information.

This section is organized as follows:

- Register access instructions.
- PL1 configurable controls.
- EL2 configurable controls on page G1-9010.
- EL3 configurable controls on page G1-9030.
- Pseudocode description of configurable instruction enables, disables, and traps on page G1-9034.

G1.21.1 Register access instructions

When an instruction is disabled or trapped, the exception is taken before execution of the instruction. This means that if the instruction is a register access instruction:

- No access is made before the exception is taken.
- Side-effects that are normally associated with the access do not occur before the exception is taken.

G1.21.2 PL1 configurable controls

In AArch32 state, each control is associated with a particular System register field that is accessible:

- When EL3 is using AArch64, or when an implementation does not include EL3, from EL1.
- When EL3 is using AArch32:
  - In Non-secure state, from EL1.
  - In Secure state, from EL3.

This means that the controls are described as PL1 controls, because PL1 is defined as being the Privilege level of software that is executing:

- At EL3, if the PE is executing in EL3 and EL3 is using AArch32.
- At EL1 under all other conditions.

Where there is an AArch64 control that is equivalent to an AArch32 PL1 control, the AArch64 control is an EL1 control.

Any exception that is generated because of an AArch32 PL1 control is taken to a PL1 mode.

--- Note ---

Any exception generated because of an AArch32 PL1 control is taken to AArch32 state.
Table G1-22 shows the AArch32 System registers that contain these controls.

### Table G1-22 System registers that contain instruction enables and disables, and trap controls

<table>
<thead>
<tr>
<th>Register name</th>
<th>Register description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCTLR</td>
<td>System Control Register</td>
</tr>
<tr>
<td>FPEXC</td>
<td>Floating-point Exception Control Register</td>
</tr>
<tr>
<td>CPACR</td>
<td>Architectural Feature Access Control Register</td>
</tr>
<tr>
<td>DBGDSCRext</td>
<td>Monitor System Debug Control Register</td>
</tr>
<tr>
<td>PMUSERENR</td>
<td>Performance Monitors User Enable Register</td>
</tr>
<tr>
<td>AMUSERENR</td>
<td>Activity Monitors User Enable Register</td>
</tr>
</tbody>
</table>

Table G1-23 on page G1-9003 summarizes these controls.

### Table G1-23 Instruction enables and disables, and trap controls, for exceptions taken to Undefined mode

<table>
<thead>
<tr>
<th>Control</th>
<th>Control type&lt;sup&gt;a&lt;/sup&gt;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCTLR.{nTWE, nTWI}</td>
<td>T</td>
<td>Traps to Undefined mode of EL0 execution of WFE and WFI instructions on page G1-9004</td>
</tr>
<tr>
<td>SCTLR.{SED, ITD}</td>
<td>D</td>
<td>[Disabling or enabling PL0 and PL1 use of AArch32 optional functionality on page G1-9004]</td>
</tr>
<tr>
<td>SCTLR.CP15BEN</td>
<td>E</td>
<td>[Disabling or enabling PL0 and PL1 use of AArch32 optional functionality on page G1-9004]</td>
</tr>
<tr>
<td>CPACR.TRCDIS</td>
<td>T</td>
<td>Traps to Undefined mode of PL0 and PL1 System register accesses to trace registers on page G1-9005</td>
</tr>
<tr>
<td>CPACR.{cp11, cp10}</td>
<td>E</td>
<td>[Enabling use of Advanced SIMD and floating-point functionality on page G1-9006]</td>
</tr>
<tr>
<td>FPEXC.EN</td>
<td>E</td>
<td>[Enabling use of Advanced SIMD and floating-point functionality on page G1-9006]</td>
</tr>
<tr>
<td>CPACR.ASEDIS</td>
<td>D</td>
<td>[Disabling or enabling PL0 and PL1 use of AArch32 optional functionality on page G1-9004]</td>
</tr>
<tr>
<td>DBGDSCRext.UDCCdis</td>
<td>T</td>
<td>Traps to Undefined mode of EL0 accesses to the Debug Communications Channel (DCC) registers on page G1-9007</td>
</tr>
<tr>
<td>CNTKCTL.{PL0PTEN, PL0VTEN, PL0PCTEN, PL0VCTEN}</td>
<td>T</td>
<td>Traps to Undefined mode of EL0 accesses to the Generic Timer registers on page G1-9008</td>
</tr>
<tr>
<td>PMUSERENR.{ER, CR, SW, EN}</td>
<td>T</td>
<td>Traps to Undefined mode of EL0 accesses to Performance Monitors registers on page G1-9008</td>
</tr>
<tr>
<td>AMUSERENR.EN</td>
<td>T</td>
<td>Traps to Undefined mode of EL0 accesses to Activity Monitors registers on page G1-9009</td>
</tr>
</tbody>
</table>

<sup>a</sup> See Table G1-24 on page G1-9003.

### Table G1-24 Control types, for exceptions taken to Undefined mode

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Type</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Disable</td>
<td>Instruction enables and instruction disables on page G1-9001</td>
</tr>
<tr>
<td>E</td>
<td>Enable</td>
<td>Instruction enables and instruction disables on page G1-9001</td>
</tr>
<tr>
<td>T</td>
<td>Trap</td>
<td>Trap controls on page G1-9001</td>
</tr>
</tbody>
</table>
When generated in Non-secure User mode, exceptions generated by these controls can be routed to EL2, as described in Routing exceptions from Non-secure EL0 to EL2 on page G1-8944.

Instructions that fail their Condition code check

See Conditional execution of undefined instructions on page G1-8966.

Trapping to PL1 of instructions that are UNPREDICTABLE

For an instruction that is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE, when the instruction is disabled or trapped then it is CONSTRAINED UNPREDICTABLE whether execution of the instruction generates an Undefined Instruction exception.

Traps to Undefined mode of EL0 execution of WFE and WFI instructions

SCTL.R.{nTWE, nTWI} trap EL0 execution of WFE and WFI instructions to Undefined mode:

SCTL.R.nTWE

1  This control has no effect on the EL0 execution of WFE instructions.
0  Any attempt to execute a WFE instruction at EL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.

SCTL.R.nTWI

1  This control has no effect on the EL0 execution of WFI instructions.
0  Any attempt to execute a WFI instruction at EL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.

The attempted execution of a conditional WFE or WFI instruction is only trapped if the instruction passes its Condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE or WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When generated in Non-secure User mode, exceptions generated by these controls can be routed to EL2, as described in Routing exceptions from Non-secure EL0 to EL2 on page G1-8944.

For more information about these instructions, and when they can cause the PE to enter a low-power state, see:

•  Wait For Event and Send Event on page G1-8988.
•  Wait For Interrupt on page G1-8991.

Disabling or enabling PL0 and PL1 use of AArch32 optional functionality

Table G1-25 on page G1-9005 shows the optional AArch32 functionality that might have disable controls in the SCTL.R:

•  The SED control is implemented if the implementation supports mixed-endian operation at any Exception level.
•  Whether the ITD control is implemented is IMPLEMENTATION DEFINED.
•  Whether the CP15BEN control is implemented is IMPLEMENTATION DEFINED.
•  If a control is not implemented, then the associated functionality cannot be disabled.

When an instruction is disabled by one of these controls, it is UNDEFINED at PL0 and PL1. This means an attempt to execute the instruction at PL0 or PL1 generates an Undefined Instruction exception that is taken to Undefined mode, unless both of the following apply, in which case the attempted execution generates an exception that is taken to EL2, as described in Routing exceptions from Non-secure EL0 to EL2 on page G1-8944:

•  The instruction is executed at Non-secure EL0 using AArch32.
Either:
— EL2 is using AArch32 and the value of HCR.TGE is 1.
— EL2 is using AArch64 and the value of HCR_EL2.TGE is 1.

Table G1-25 PL1 controls for disabling and enabling PL0 and PL1 use of AArch32 optional functionality

<table>
<thead>
<tr>
<th>Optional AArch32 functionality</th>
<th>Instruction enable or disable in the SCTLR(^a)</th>
<th>Disabled instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETEND instructions</td>
<td>SED(^b)</td>
<td>SETEND instructions</td>
</tr>
<tr>
<td>Some uses of IT instructions</td>
<td>ITD(^c)</td>
<td>See the SCTLR.ITD description</td>
</tr>
<tr>
<td>Accesses to the CP15DMB, CP15DSB, and CP15ISB barrier instructions</td>
<td>CP15BEN(^d)</td>
<td>(\times) accesses to the CP15DMB, CP15DSB, and CP15ISB instructions</td>
</tr>
</tbody>
</table>

\(^a\) The controls that are implemented in SCTLR are also implemented in SCTLR_EL1, and apply when PL1 is using AArch64 and PL0 is using AArch32.
\(^b\) SETEND instruction disable. SETEND instructions are disabled when the value of this field is 1.
\(^c\) IT instruction disable. If this control is implemented, some uses of IT instructions are disabled when the value of this field is 1.
\(^d\) System register (\(\text{copyrc}==0b1111\)) memory barrier enable. If this control is implemented, the specified register accesses are disabled when the value of CP15BEN is 0.

Note

The uses of the IT instruction, and use of the CP15DMB, CP15DSB, and CP15ISB barrier instructions, are deprecated for performance reasons.

Traps to Undefined mode of PL0 and PL1 System register accesses to trace registers

If implemented, the CPACR.TRCDIS control traps PL0 and PL1 System register accesses to the trace registers to Undefined mode, as follows:

1. PL0 and PL1 accesses to the System register interface to the trace unit are trapped to Undefined mode
2. This control has no effect on PL0 and PL1 accesses to the System register interface to the trace unit.

If the CPACR.TRCDIS control is not implemented, then the CPACR.TRCDIS field is RAZ/WI. This means the CPACR does not provide a trap to Undefined mode of PL1 and PL0 System register accesses to trace registers. See the register description for more information.

Note

• System register accesses to the trace unit use the (\(\text{coproc}==0b1110\)) encoding space.
• The ETMv4 architecture does not permit EL0 to access the trace registers. If the architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace System registers are UNDEFINED.
• The architecture does not provide traps on trace register accesses through the optional memory-mapped external debug interface.

System register accesses to the trace System registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see Register access instructions on page G1-9002.

If EL3 is implemented and is using AArch32, and NSACR.NSTRCDIS is 1, CPACR.TRCDIS behaves as RAO/WI in Non-secure state. This behavior also applies if the CPACR.TRCDIS control is not implemented.

When generated in Non-secure User mode, an exception generated by this control can be routed to EL2, as described in Routing exceptions from Non-secure EL0 to EL2 on page G1-8944.
Enabling use of Advanced SIMD and floating-point functionality

Table G1-26 on page G1-9006 summarizes the controls of Advanced SIMD and floating-point functionality.

Table G1-26 Controls of use of Advanced SIMD and floating-point functionality

<table>
<thead>
<tr>
<th>Control</th>
<th>Type</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPACR.{cp11, cp10}</td>
<td>E</td>
<td>Enabling PL0 and PL1 accesses to the SIMD and floating-point registers on page G1-9006</td>
</tr>
<tr>
<td>FPEXC.EN</td>
<td>E</td>
<td>Enabling access to the SIMD and floating-point registers on page G1-9007</td>
</tr>
<tr>
<td>CPACR.ASEDIS</td>
<td>D</td>
<td>Disabling PL0 and PL1 execution of Advanced SIMD instructions on page G1-9007</td>
</tr>
</tbody>
</table>

If any of CPACR.{cp11, cp10}, FPEXC.EN, or for Advanced SIMD instructions, CPACR.ASEDIS, disable a floating-point or an Advanced SIMD instruction, the instruction is UNDEFINED. Support for the CPACR.ASEDIS control is optional, and if the control is not implemented behavior is as if the control permits the execution of Advanced SIMD instructions at PL1 and PL0.

When generated in Non-secure User mode, exceptions generated by these controls can be routed to EL2, as described in Routing exceptions from Non-secure EL0 to EL2 on page G1-8944.

Enabling PL0 and PL1 accesses to the SIMD and floating-point registers

CPACR.{cp11, cp10} enable PL0 and PL1 accesses to the SIMD and floating-point registers.

When CPACR.cp10 is:

- **00**: PL0 and PL1 accesses to Advanced SIMD and floating-point registers or instructions are UNDEFINED.
- **01**: PL0 accesses to Advanced SIMD and floating-point registers or instructions are UNDEFINED.
- **10**: Reserved. The effect of programming this field to this value is CONSTRAINED UNPREDICTABLE.
- **11**: This control permits full access to the Advanced SIMD and floating-point functionality from PL0 and PL1.

The value of CPACR.cp11 is ignored. If CPACR.cp11 is programmed with a different value to CPACR.cp10 then CPACR.cp11 is UNKNOWN on a direct read of the CPACR.

Note

- Software must set CPACR.cp11 and CPACR.cp10 to the same value.

Table G1-27 on page G1-9006 shows the registers for which accesses are enabled.

Table G1-27 Register accesses enabled at PL0 and PL1 by CPACR.{cp11, cp10}

<table>
<thead>
<tr>
<th>Enabled at</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL0 and PL1, or PL0 only(^a)</td>
<td>FPSR, FPEXC, FPSID, MVFR0, MVFR1, MVFR2, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers(^b)</td>
</tr>
</tbody>
</table>

\(^a\) Depending on the value of CPACR.{cp11, cp10}. See the register description for details.

\(^b\) Permitted VMSR accesses to the FPSID are ignored, but for the purposes of the \(\{cp10, cp11\}\) controls the architecture defines a VMSR accesses to the FPSID from EL1 or higher is an access to a SIMD and floating-point register.
If EL3 is implemented and is using AArch32, and NSACR.{cp11, cp10} are both set to 0, the functionality described in this section is disabled in Non-secure state, and CPACR.{cp11, cp10} are RAZ/WI in Non-secure state. See Enabling Non-secure access to SIMD and floating-point functionality on page G1-9034.

For more information about SIMD and floating-point support, see Advanced SIMD and floating-point support on page G1-8996.

**Enabling access to the SIMD and floating-point registers**

FPEXC.EN enables accesses to the SIMD and floating-point registers at all Exception levels, but does not control the following:

- VMSR accesses to the FPEXC or FPSID.
- VMRS accesses from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2.

When FPEXC.EN is:

- 1: Accesses to the registers shown in Table G1-28 on page G1-9007 are enabled at all Exception levels.
- 0: All accesses to the registers shown in Table G1-28 on page G1-9007 are UNDEFINED.

Table G1-28 on page G1-9007 shows the registers for which accesses are enabled, and for an exception taken to Hyp mode, how the exception is reported in HSR.

<table>
<thead>
<tr>
<th>Enabled at</th>
<th>Registers</th>
<th>Syndrome reporting in HSRa</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Exception levels</td>
<td>FPSCR, and any of the SIMD and floating-point registers Q0-Q15,</td>
<td>Exception for an unknown reason, using EC value 0x00</td>
</tr>
<tr>
<td></td>
<td>including their views as D0-D31 registers or S0-S31 registers.</td>
<td></td>
</tr>
</tbody>
</table>

a. Only for exceptions that are taken to Hyp mode.

For more information, see Advanced SIMD and floating-point support on page G1-8996.

**Disabling PL0 and PL1 execution of Advanced SIMD instructions**

If implemented as an RW field, CPACR.ASEDIS can disable PL0 and PL1 execution of Advanced SIMD instructions, as follows:

- 1: Advanced SIMD instructions are UNDEFINED at PL0 and PL1.
- 0: Advanced SIMD instruction execution is enabled at PL0 and PL1.

The instructions that CPACR.ASEDIS disables are those described in Controls of Advanced SIMD operation that do not apply to floating-point operation on page E1-7153.

When the control is not implemented, meaning the CPACR.ASEDIS field is RAZ/WI, behavior is as if the control permits execution of Advanced SIMD instructions at PL0 and PL1.

If EL3 is implemented and is using AArch32, and NSACR.NSASEDIS is 1, CPACR.ASEDIS is RAO/WI in Non-secure state. This also applies when the CPACR.ASEDIS control is not implemented.

**Traps to Undefined mode of EL0 accesses to the Debug Communications Channel (DCC) registers**

DBGDSCRext.UDCCdis traps EL0 accesses to the DCC registers to Undefined mode:

- 1: EL0 accesses to the DCC registers are trapped to Undefined mode
- 0: This control has no effect on EL0 accesses to the DCC registers.

Traps of EL0 accesses to the DBGDTRRXint and DBGDTRTXint are ignored in Debug state.
Table G1-29 shows the registers for which accesses are trapped.

**Table G1-29 Register accesses trapped to Undefined mode when DBGDSCRext.UDCCdis is 1**

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL0</td>
<td>DBGDSCRint, DBGDTRRXint, DBGDTRTXint, DBGDIDR, DBGDASAR, DBGDAR</td>
</tr>
</tbody>
</table>

**Note**

All accesses to these registers are trapped, including LDC and STC accesses to DBGDTRTXint and DBGDTRRXint, and MRRC accesses to DBGDASR and DBGDAR.

When generated in Non-secure User mode, an exception generated by this control can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-8944.

**Traps to Undefined mode of EL0 accesses to the Generic Timer registers**

CNTKCTL.{PL0PTEN, PL0VTEN, PL0PCTEN, PL0VCTEN} trap EL0 accesses to the Generic Timer registers to Undefined mode, as follows:

- CNTKCTL.PL0PTEN traps EL0 accesses to the physical timer registers.
- CNTKCTL.PL0VTEN traps EL0 accesses to the virtual timer registers.
- CNTKCTL.PL0PCTEN traps EL0 accesses to the frequency register and physical counter register.
- CNTKCTL.PL0VCTEN traps EL0 accesses to the frequency register and virtual counter register.

For all of these controls:

1. This control has no effect on EL0 accesses to the corresponding registers.
2. EL0 accesses to the corresponding registers are trapped to Undefined mode.

Accesses to the frequency register, CNTFRQ, are only trapped if CNTKCTL.PL0PCTEN and CNTKCTL.PL0VCTEN are both 0.

Table G1-30 shows the registers for which accesses are trapped.

**Table G1-30 Register accesses trapped to Undefined mode by CNTKCTL trap controls**

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Trap control</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL0</td>
<td>PL0PTEN</td>
<td>CNTP_CTL, CNTP_CVAL, CNTP_TVAL</td>
</tr>
<tr>
<td></td>
<td>PL0VTEN</td>
<td>CNTV_CTL, CNTV_CVAL, CNTV_TVAL</td>
</tr>
<tr>
<td></td>
<td>PL0PCTEN</td>
<td>CNTFRQ, CNTPCT</td>
</tr>
<tr>
<td></td>
<td>PL0VCTEN</td>
<td>CNTFRQ, CNTVCT</td>
</tr>
</tbody>
</table>

When generated in Non-secure User mode, an exception generated by this control can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-8944.

**Traps to Undefined mode of EL0 accesses to Performance Monitors registers**

PMUSERENR. {ER, CR, SW, EN} trap EL0 accesses to the Performance Monitors registers to Undefined mode. For each of these controls:

1. This control has no effect on EL0 accesses to the corresponding registers.
2. EL0 accesses to the corresponding registers are trapped to Undefined mode.

For those Performance Monitors registers that more than one PMUSERENR. {ER, CR, SW, EN} control applies to, accesses are only trapped if all controls that apply are set to 0.
The accesses that these trap controls trap might be reads, writes, or both.

--- Note ---

- The architecture does not provide traps on Performance Monitors register accesses through the memory-mapped external debug interface.
- If the Performance Monitors Extension is not implemented, the Performance Monitors registers, including PMUSERENR, are reserved.

Table G1-31 on page G1-9009 shows the registers for which EL0 accesses are trapped. For each register, the table shows the type of access trapped.

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Trap control</th>
<th>Registers</th>
<th>Access type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL0</td>
<td>ER</td>
<td>PMXEVCNTR, PMEVCNTR&lt;\text{n}&gt;</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PMSELR</td>
<td>RW</td>
</tr>
<tr>
<td>CR</td>
<td>PMCCCNTR, accessed using an MRC</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>CR</td>
<td>PMCCCNTR, accessed using an MRRC</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>PMSWINC</td>
<td></td>
<td>W</td>
</tr>
<tr>
<td>EN</td>
<td>PMCINTENSET, PMCINTENCLR, PMCR, PMOVSR, PMSWINC, PMSELR, PMCEID0, PMCEID1, PMCEID2, PMCEID3, PMCCNTR, PMXEVTYPE, PMXEVCNTR, PMOSSET, PMEVCNTR&lt;\text{n}&gt;, PMEVTYPER&lt;\text{n}&gt;, PMCCFILTER</td>
<td>RW&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
</tbody>
</table>

<sup>a</sup> The EL0 access is trapped only if the corresponding EL1 accesses is permitted. For example, the PMSWINC register is WO at EL1, and therefore, when the value of EN is 0:
- Write accesses to the register from EL0 are trapped.
- Read accesses to the register from EL0 are UNDEFINED, because read accesses to the register from EL1 are UNDEFINED.

When generated in Non-secure User mode, an exception generated by this control can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-8944.

**Traps to Undefined mode of EL0 accesses to Activity Monitors registers**

AMUSERENR.EN traps EL0 accesses to the Activity Monitors System registers other than AMUSERENR to Undefined mode:

1. This control has no effect on EL0 accesses to the corresponding registers.
2. EL0 accesses to the corresponding registers are trapped to Undefined mode.

--- Note ---

- The architecture does not provide traps on Activity Monitors register accesses through the memory-mapped external interface.
- If the Activity Monitors Extension is not implemented, the Activity Monitors registers, including AMUSERENR, are reserved.
Table G1-32 on page G1-9010 shows the registers for which EL0 accesses are trapped.

**Table G1-32 Register accesses trapped to Undefined mode when disabled from EL0**

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL0</td>
<td>AMCFGFR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPE0&lt;n&gt;, AMEVTYPE1&lt;n&gt;, AMEVCNTR0&lt;n&gt; or AMEVCNTR1&lt;n&gt;.</td>
</tr>
</tbody>
</table>

When generated in Non-secure User mode, an exception generated by this control can be routed to EL2, as described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-8944.

### G1.21.3 EL2 configurable controls

These controls are ignored in Secure state when using AArch32.

Table G1-33 on page G1-9010 shows the System registers that contain these controls.

**Table G1-33 System registers that contain instruction enables and disables, and trap controls**

<table>
<thead>
<tr>
<th>Register name</th>
<th>Register description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPEXC</td>
<td>Floating-point Exception Control Register</td>
</tr>
<tr>
<td>HCR</td>
<td>Hypervisor Configuration Register</td>
</tr>
<tr>
<td>HSTR</td>
<td>Hypervisor System Trap Register</td>
</tr>
<tr>
<td>HCPTR</td>
<td>Hyp Architectural Feature Trap Register</td>
</tr>
<tr>
<td>HDCR</td>
<td>Hyp Debug Control Register</td>
</tr>
</tbody>
</table>

---

**Note**

- FPEXC.EN is a control that is in a System register provided by PL1. However, some exceptions generated because the value of FPEXC.EN is 1 are taken to Hyp mode.
- For completeness, Table G1-34 on page G1-9010 includes the HCR.TGE routing control, which is described in *Routing exceptions from Non-secure EL0 to EL2* on page G1-8944.

Table G1-34 on page G1-9010 summarizes the controls.

**Table G1-34 Instruction enables and disables, and trap controls, for exceptions taken to Hyp mode**

<table>
<thead>
<tr>
<th>Control</th>
<th>Control typea</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSCTLR.{SED, ITD}</td>
<td>D</td>
<td>Disabling or enabling EL2 use of AArch32 optional functionality on page G1-9013</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>HCR.{TRVM, TVM}</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure EL1 accesses to virtual memory control registers on page G1-9014</td>
</tr>
<tr>
<td>HCR.HCD</td>
<td>D</td>
<td>Disabling Non-secure state execution of HVC instructions on page G1-9014</td>
</tr>
<tr>
<td>HCR.TGE</td>
<td>R</td>
<td>Routing exceptions from Non-secure EL0 to EL2 on page G1-8944</td>
</tr>
<tr>
<td>HCR.TTLB</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure EL1 execution of TLB maintenance instructions on page G1-9015</td>
</tr>
</tbody>
</table>

 performance and reliability.
The AArch32 System Level Programmers' Model
G1.21 Configurable instruction enables and disables, and trap controls

Table G1-34 Instruction enables and disables, and trap controls, for exceptions taken to Hyp mode (continued)

<table>
<thead>
<tr>
<th>Control</th>
<th>Control type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCR.{TSW, TPC, TPU}</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure EL1 execution of cache maintenance instructions on page G1-9015</td>
</tr>
<tr>
<td>HCR.TAC</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure EL1 accesses to the Auxiliary Control Register on page G1-9016</td>
</tr>
<tr>
<td>HCR.TIDCP</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations on page G1-9016</td>
</tr>
<tr>
<td>HCR.TSC</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure EL1 execution of SMC instructions on page G1-9017</td>
</tr>
<tr>
<td>HCR.{TID0, TID1, TID2, TID3}</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the ID registers on page G1-9018</td>
</tr>
<tr>
<td>HCR.{TWI, TWE}</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions on page G1-9020</td>
</tr>
<tr>
<td>HCPTR.TAM</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Activity Monitors registers on page G1-9021</td>
</tr>
<tr>
<td>HCPTR.{TCP11, TCP10}</td>
<td>T</td>
<td>General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers on page G1-9021</td>
</tr>
<tr>
<td>FPEXC.EN</td>
<td>T</td>
<td>Enabling access to the SIMD and floating-point registers on page G1-9022</td>
</tr>
<tr>
<td>HCPTR.TASE</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure accesses to Advanced SIMD functionality on page G1-9022</td>
</tr>
<tr>
<td>HCPTR.TCPAC</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure EL1 accesses to the CPACR on page G1-9023</td>
</tr>
<tr>
<td>HCPTR.TTA</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure System register accesses to trace registers on page G1-9023</td>
</tr>
<tr>
<td>HDCR.TTRF</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure System register accesses to trace filter control registers on page G1-9024</td>
</tr>
<tr>
<td>HSTR.{T0-T3, T5-T13, T15}</td>
<td>T</td>
<td>General trapping to Hyp mode of Non-secure EL0 and EL1 accesses to System registers in the (coproc==0b1111) encoding space on page G1-9024</td>
</tr>
<tr>
<td>HDCR.{TDRA, TDOSA, TDA}</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure System register accesses to debug registers on page G1-9026</td>
</tr>
<tr>
<td>CNTHCTL.{PL1PCEN, PL1PCTEN}</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the Generic Timer registers on page G1-9028</td>
</tr>
<tr>
<td>HDCR.{TPM, TPMCR}</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Performance Monitors registers on page G1-9029</td>
</tr>
<tr>
<td>HCR2.TERR</td>
<td>T</td>
<td>Traps to Hyp mode of Non-secure EL1 accesses to the RAS error record registers on page G1-9030</td>
</tr>
</tbody>
</table>

a. See Table G1-35 on page G1-9012.
Table G1-35 Control types, for exceptions taken to Hyp mode

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Type</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Disable</td>
<td>Instruction enables and instruction disables on page G1-9001</td>
</tr>
<tr>
<td>E</td>
<td>Enable</td>
<td>Instruction enables and instruction disables on page G1-9001</td>
</tr>
<tr>
<td>R</td>
<td>Routing control</td>
<td>Routing exceptions from Non-secure EL0 to EL2 on page G1-8944</td>
</tr>
<tr>
<td>T</td>
<td>Trap</td>
<td>Trap controls on page G1-9001</td>
</tr>
</tbody>
</table>

Also see the following:

- Register access instructions on page G1-9002.
- Instructions that fail their Condition code check.
- Trapping to EL2 of instructions that are UNPREDICTABLE on page G1-9013.

### Instructions that fail their Condition code check

For **UNDEFINED** instructions that fail their Condition code check, see Conditional execution of undefined instructions on page G1-8966.

For an instruction that has a Hyp trap set that fails its Condition code check:

- Unless the trap description states otherwise, it is IMPLEMENTATION DEFINED whether the instruction:
  - Generates a Hyp Trap exception.
  - Executes as a NOP.

Any implementation must be consistent in its handling of instructions that fail their Condition code check. This means that:

- Whenever a Hyp trap is set on an instruction it must either:
  - Always generate a Hyp Trap exception.
  - Always treat the instruction as a NOP.

- The IMPLEMENTATION DEFINED part of the requirements of Conditional execution of undefined instructions on page G1-8966 must be consistent with the handling of Hyp traps on instructions that fail their Condition code check. Table G1-36 on page G1-9012 shows this:

### Table G1-36 Consistent handling of instructions that fail their Condition code check

<table>
<thead>
<tr>
<th>Behavior of conditional <strong>UNDEFINED</strong> instruction(^a)</th>
<th>Hyp trap on instruction that fails its Condition code check(^b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executes as a NOP</td>
<td>Executes as a NOP</td>
</tr>
<tr>
<td>Generates an Undefined Instruction exception</td>
<td>Generates a Hyp Trap exception</td>
</tr>
</tbody>
</table>

\(^a\) As defined in Conditional execution of undefined instructions on page G1-8966. In Non-secure EL0 and EL1 modes, this applies only if no Hyp trap is set for the instruction, otherwise see the behavior in the other column of the table.

\(^b\) For a trapped instruction executed in a Non-secure EL1 or EL0 mode.

---

**Note**

Hyp traps on WFE and WFI instructions generate Hyp Trap exceptions only if the instruction passes its Condition code check. See Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions on page G1-9020.
Trapping to EL2 of instructions that are UNPREDICTABLE

For an instruction that is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE, when the instruction is disabled or trapped then it is CONSTRAINED UNPREDICTABLE whether execution of the instruction generates a Hyp Trap exception.

Note

UNPREDICTABLE and CONSTRAINED UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or lower Exception level using instructions that are not UNPREDICTABLE and are not CONSTRAINED UNPREDICTABLE. This means that disabling or trapping an instruction changes the set of instructions that might be executed in Non-secure state at EL1 or EL0. This indirectly affects the permitted behavior of UNPREDICTABLE and CONSTRAINED UNPREDICTABLE instructions.

If no instructions are trapped, the attempted execution of an UNPREDICTABLE instruction in a Non-secure EL1 or EL0 mode must not generate a Hyp Trap exception.

Disabling or enabling EL2 use of AArch32 optional functionality

Table G1-37 on page G1-9013 shows the optional AArch32 functionality that might have disable controls in the HSCTLR:

- The SED control is implemented if the implementation supports mixed-endian operation at EL2.
- Whether the ITD control is implemented is IMPLEMENTATION DEFINED.
- Whether the CP15BEN control is implemented is IMPLEMENTATION DEFINED.
- If a control is not implemented, then the associated functionality cannot be disabled.

These HSCTLR controls apply only to execution at EL2 using AArch32. When an instruction is disabled by one of these controls, it is UNDEFINED at EL2, meaning it is undefined in Hyp mode.

Table G1-37 EL2 controls for disabling and enabling EL2 use of AArch32 optional functionality

<table>
<thead>
<tr>
<th>Optional AArch32 functionality</th>
<th>Instruction enable or disable in the HSCTLR</th>
<th>Disabled instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETEND instructions</td>
<td>SED&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SETEND instructions</td>
</tr>
<tr>
<td>Some uses of IT instructions</td>
<td>ITD&lt;sup&gt;b&lt;/sup&gt;</td>
<td>See the HSCTLR.IT description</td>
</tr>
<tr>
<td>Accesses to the System register (coproc==0b1111) DMB, DSB, and ISB barrier operations</td>
<td>CP15BEN&lt;sup&gt;c&lt;/sup&gt;</td>
<td>MCR accesses to the CP15DMB, CP15DSB, and CP15ISB</td>
</tr>
</tbody>
</table>

Note

- SETEND instruction disable. SETEND instructions are disabled when the value of this field is 1.
- IT instruction disable. If this control is implemented, some uses of IT instructions are disabled when the value of this field is 1.
- System register (coproc==0b1111) memory barrier enable. If this control is implemented, the specified register accesses are disabled when the value of CP15BEN is 0.

Note

- These controls have no effect on instructions executed in any mode other than Hyp mode. The SCTLR provides similar controls that apply to execution in other modes.
- The uses of the IT instruction, and use of the CP15DMB, CP15DSB, and CP15ISB barrier instructions, are deprecated for performance reasons.
Traps to Hyp mode of Non-secure EL1 accesses to virtual memory control registers

HCR.{TRVM, TVM} trap Non-secure EL1 accesses to the virtual memory control registers to Hyp mode:

**HCR.TRVM, for read accesses:**

1  Non-secure EL1 reads of the virtual memory control registers are trapped to Hyp mode.
0  This control has no effect on Non-secure EL1 reads of the virtual memory control registers.

**HCR.TVM, for write access:**

1  Non-secure EL1 writes to the virtual memory control registers are trapped to Hyp mode.
0  This control has no effect on Non-secure EL1 writes to the virtual memory control registers.

Table G1-38 on page G1-9014 shows the registers for which:

- Reads are trapped to Hyp mode when HCR.TRVM is 1.
- Writes are trapped to Hyp mode when HCR.TVM is 1.

The table also shows how the exceptions are reported in HSR.

Table G1-38 Register read and write accesses trapped when HCR.{TRVM, TVM} are 1

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Registers</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure EL1</td>
<td>SCTLR, TTBR0, TTBR1, TTBCR, TTBCR2, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR</td>
<td>Trapped MCR or MRC access (coproc==0b1111), using EC value 0x03</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Trapped MCRR or MRRC access (coproc==0b1111), using EC value 0x04</td>
</tr>
</tbody>
</table>

--- Note ---

These registers are not accessible at EL0.

Disabling Non-secure state execution of HVC instructions

HCR.HCD disables Non-secure state execution of HVC instructions:

1  HVC instructions are UNDEFINED at EL2 and Non-secure EL1. The Undefined Instruction exception is taken from the current Exception level to the current Exception level.
0  HVC instruction execution is enabled at EL2 and Non-secure EL1.

--- Note ---

HVC instructions are always UNDEFINED at EL0.

HCR.HCD is only implemented if EL3 is not implemented. Otherwise, it is RES0. See the HCR register description. Table G1-39 on page G1-9014 shows how the exceptions are reported in HSR.

Table G1-39 Instruction that causes exceptions when HCR.HCD is 1

<table>
<thead>
<tr>
<th>Attempted execution in</th>
<th>Disabled instruction</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hyp mode</td>
<td>HVC</td>
<td>Exception for an unknown reason, using EC value 0x00</td>
</tr>
<tr>
<td>Mode other than Hyp mode</td>
<td>HVC</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>
Traps to Hyp mode of Non-secure EL1 execution of TLB maintenance instructions

The System instruction encoding space includes TLB maintenance instructions.

HCR.TTLB traps Non-secure EL1 execution of TLB maintenance instructions to Hyp mode:

1  Any attempt to execute a TLBI instruction at Non-secure EL1 is trapped to Hyp mode.
0  This control has no effect on the Non-secure EL1 execution of TLBI instructions.

Table G1-40 on page G1-9015 shows the instructions that are trapped, and how the exceptions are reported in HSR.

### Table G1-40 Instructions trapped to Hyp mode when HCR.TTLB is 1

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Trapped instructions</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure EL1</td>
<td>TLBIALLIS, TLBIMVAIS, TLBISIDIS, TLBIMVAAIS, TLBIVALIS, TLBIVALIS, ITLBIALL, ITLBMVA, ITLBMVAS, DTLBIALL, DTLBMVA, DTLBIASID, TLBLALL, TLBLMA, TLBLMVA, TLBLISID, TLIBMVA, TLIBMVAIS, TLIBMVAL, TLIBMVAAL.</td>
<td>Trapped MCR or MRC access (coproc==0b1111), using EC value 0x03</td>
</tr>
</tbody>
</table>

--- Note ---

These instructions are always UNDEFINED at EL0.

For more information about these instructions, see The scope of TLB maintenance instructions on page G5-9230.

Traps to Hyp mode of Non-secure EL1 execution of cache maintenance instructions

HCR.{TSW, TPC, TPU} trap cache maintenance instructions to Hyp mode:

0  The control has no effect on the execution of cache maintenance instructions.
1  Any attempt to execute one of the cache maintenance instructions shown in Table G1-42 on page G1-9015 at Non-secure EL1 is trapped to Hyp mode.

Table G1-41 Controls for trapping cache maintenance instructions to Hyp mode

<table>
<thead>
<tr>
<th>Trap control</th>
<th>Trapped instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCR.TSW</td>
<td>Data or unified cache maintenance by set/way</td>
</tr>
<tr>
<td>HCR.TPC</td>
<td>Data or unified cache maintenance to point of coherency</td>
</tr>
<tr>
<td>HCR.TPU</td>
<td>Cache maintenance to point of unification</td>
</tr>
</tbody>
</table>

Table G1-42 on page G1-9015 shows the instructions that are trapped to Hyp mode, and how the exceptions are reported in HSR.

### Table G1-42 Instructions trapped to Hyp mode when HCR.{TSW, TPC, TPU} are 1

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Trap control</th>
<th>Trapped instructions</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure EL1</td>
<td>TSW</td>
<td>DCISW, DCCSW, DCCSW</td>
<td>Trapped MCR or MRC access (coproc==0b1111), using EC value 0x03</td>
</tr>
<tr>
<td></td>
<td>TPC</td>
<td>DCIMVAC, DCCMVAC, DCCMVAC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TPU</td>
<td>ICIMVAU, ICIALLU, ICIALLUIS, DCCMVAC</td>
<td></td>
</tr>
</tbody>
</table>
Note
These instructions are always UNDEFINED at EL0.

For more information about these instructions, see Cache maintenance system instructions on page K17-11905.

Traps to Hyp mode of Non-secure EL1 accesses to the Auxiliary Control Register

HCR.TAC traps Non-secure EL1 accesses to the Auxiliary Control Registers to Hyp mode:

1   Non-secure EL1 accesses to the Auxiliary Control Registers are trapped to Hyp mode.
0   This control has no effect on Non-secure EL1 accesses to the Auxiliary Control Registers.

Table G1-43 on page G1-9016 shows the registers for which accesses are trapped, and how the exceptions are reported in HSR:

Table G1-43 Register accesses trapped to Hyp mode when HCR.TAC is 1

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Registers</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure EL1</td>
<td>ACTLR and, if implemented, ACTLR2.</td>
<td>Trapped MCR or MRC access (coproc==0b1111) access, using EC value 0x03</td>
</tr>
</tbody>
</table>

Note
The ACTLR and ACTLR2 are not accessible at EL0.

Traps to Hyp mode of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations

The lockdown, DMA, and TCM features are IMPLEMENTATION DEFINED. The architecture reserves the encodings of a number of System registers for control of these features.

HCR.TIDCP traps the execution of System register access instructions that access these registers, as follows:

1   At Non-secure EL1, any attempt to execute an MCR or MRC instruction with a reserved register encoding shown in Table G1-44 on page G1-9017 is trapped to Hyp mode.
    At Non-secure EL0, it is IMPLEMENTATION DEFINED whether attempts to execute MCR or MRC instructions with reserved register encodings are:
    • Trapped to Hyp mode.
    • UNDEFINED, and the PE takes the Undefined Instruction exception to Non-secure Undefined mode.
    Any lockdown fault in the memory system caused by the use of these operations in Non-secure state generates a Data Abort exception that is taken to Hyp mode.
0   This control has no effect on Non-secure EL0 and EL1 System register access instructions with reserved register encodings shown in Table G1-44 on page G1-9017.

Note
This means that a Hyp Trap exception taken from Non-secure EL1 to Hyp mode, generated because of a configuration setting in HCR.TIDCP is a higher priority exception than an Undefined Instruction exception generated because either the System register encoding is unallocated or because the register is never accessible at EL1. As Synchronous exception prioritization for exceptions taken to AArch32 state on page G1-8933 shows, this is an exception to the general exception prioritization rules that prioritize most Undefined Instruction exceptions taken to Undefined mode above traps to EL2.
Table G1-44 on page G1-9017 shows the register encodings for which accesses are trapped to Hyp mode, and how the exceptions are reported in HSR.

Table G1-44 Encodings trapped to Hyp mode when HCR.TIDCP is 1

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Register encodings</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure EL0 and EL1</td>
<td>An access to any of the following encodings:</td>
<td>Trapped MCR or MRC access (coproc==0b1111), using EC value 0x03</td>
</tr>
<tr>
<td></td>
<td>• CRn==c9, opc1=={0-7}, CRm=={c0-c2, c5-c8}, opc2=={0-7}.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• CRn==c10, opc1=={0-7}, CRm=={c0, c1, c4, c8}, opc2=={0-7}.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• CRn==c11, opc1=={0-7}, CRm=={c0-c8, c15}, opc2=={0-7}.</td>
<td></td>
</tr>
</tbody>
</table>

An implementation can also include IMPLEMENTATION DEFINED registers that provide additional controls, to give finer-grained control of the trapping of IMPLEMENTATION DEFINED features.

Note

Arm expects the trapping of Non-secure User mode accesses to these functions to Hyp mode to be unusual, and used only when the hypervisor is virtualizing User mode operation. Arm strongly recommends that unless the hypervisor must virtualize User mode operation, a Non-secure User mode access to any of these functions generates an Undefined Instruction exception, as it would if the implementation did not include EL2. The PE then takes this exception to Non-secure Undefined mode.

Traps to Hyp mode of Non-secure EL1 execution of SMC instructions

HCR.TSC traps Non-secure EL1 execution of SMC instructions to Hyp mode:

1  Any attempt to execute an SMC instruction at Non-secure EL1 is trapped to Hyp mode, regardless of the value of SCR.SCD.

0  This control has no effect on Non-secure EL1 execution of SMC instructions.

Table G1-45 on page G1-9017 shows how the exceptions are reported in HSR:

Table G1-45 SMC Instruction trapped to Hyp mode when HCR.TSC is 1

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Trapped instruction</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure EL1</td>
<td>SMC on page F5-7900</td>
<td>Trapped SMC instruction execution in AArch32 state, using EC value 0x13</td>
</tr>
</tbody>
</table>

The architecture permits, but does not require, this trap to apply to conditional SMC instructions that fail their Condition code check, in the same way as with traps on other conditional instructions.

Note

• This trap is implemented only if the implementation includes EL3.
• SMC instructions are always UNDEFINED at EL0.
• HCR.TSC traps execution of the SMC instruction. It is not a routing control for the SMC exception. Hyp Trap and SMC exceptions have different preferred return addresses.

For more information about SMC instructions, see SMC on page F5-7900.
Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the ID registers

Other than the MIDR, MPIDR, and PMCR.N, the ID registers are divided into groups, with a trap control in the HCR for each group.

<table>
<thead>
<tr>
<th>Trap control</th>
<th>Register group</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCR.TID0</td>
<td>ID group 0, Primary device identification registers on page G1-9019</td>
</tr>
<tr>
<td>HCR.TID1</td>
<td>ID group 1, Implementation identification registry on page G1-9019</td>
</tr>
<tr>
<td>HCR.TID2</td>
<td>ID group 2, Cache identification registers on page G1-9019</td>
</tr>
<tr>
<td>HCR.TID3</td>
<td>ID group 3, Detailed feature identification registers on page G1-9020</td>
</tr>
</tbody>
</table>

These controls trap register accesses from Non-secure EL0 or EL1 to Hyp mode, as follows:

**HCR.TID0**
- 0: This control has no effect on Non-secure EL1 reads of the ID group 0 registers.
- 1: Any attempt at Non-secure EL0 or EL1 to read any register in ID group 0 is trapped to Hyp mode.

**HCR.TID1**
- 0: This control has no effect on Non-secure EL1 reads of the ID group 1 registers.
- 1: Any attempt at Non-secure EL1 to read any register in ID group 1 is trapped to Hyp mode.

**HCR.TID2**
- 0: This control has no effect on Non-secure EL1 and EL0 accesses to the ID group 2 registers.
- 1: Any attempt at Non-secure EL0 or EL1 to read any register in ID group 2, and any attempt at Non-secure EL0 or EL1 to write to the CSSELR, is trapped to Hyp mode.

**HCR.TID3**
- 0: This control has no effect on Non-secure EL1 reads of the ID group 3 registers.
- 1: Any attempt at Non-secure EL1 to read any register in ID group 3 is trapped to Hyp mode.

For the MIDR and MPIDR, and for PMCR.N, the architecture provides read/write aliases. The original register becomes accessible only from Hyp mode and Secure state, and a Non-secure EL0 or EL1 read of the original register returns the value of the read/write alias. This substitution is invisible to the EL0 or EL1 software reading the register.

<table>
<thead>
<tr>
<th>Register</th>
<th>Original</th>
<th>Alias, EL2 using AArch32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main ID</td>
<td>MIDR</td>
<td>VPIDR</td>
</tr>
<tr>
<td>Multiprocessor Affinity</td>
<td>MPIDR</td>
<td>VMPIDR</td>
</tr>
<tr>
<td>Performance Monitors Control Register</td>
<td>PMCR.N</td>
<td>HDCR.HPMN</td>
</tr>
</tbody>
</table>

Reads of the MIDR, MPIDR, or PMCR.N from Hyp mode or Secure state are unchanged by the implementation of EL2, and access the physical registers.

**Note**

- If the optional Performance Monitors Extension is not implemented, HDCR.HPMN is RES0 and PMCR is reserved.
- HDCR.HPMN also affects whether a Performance Monitors counter can be accessed from Non-secure EL1 or EL0. See the register description of HDCR for more information.
• **PMCR** contains other fields that identify the implementation. For more information about trapping accesses to the PMCR, see *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Performance Monitors registers* on page G1-9029.

A reset into AArch32 state sets **VPIDR** to the **MIDR** value, **VMPIDR** to the **MPIDR** value, and **HDCR.HPMN** to the **PMCR.N** value.

**ID group 0, Primary device identification registers**

These registers identify some top-level implementation choices.

Table G1-48 on page G1-9019 shows the registers that are in ID group 0 for traps to Hyp mode, and how the exceptions are reported in HSR.

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Group 0 registers</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure EL1</td>
<td>FPSID</td>
<td>Trapped VMRS access, for ID group traps, using EC value 0x08</td>
</tr>
<tr>
<td>Non-secure EL0 and EL1</td>
<td>JIDR</td>
<td>Trapped MCR or MRC access (coproc==0b1110), using EC value 0x05</td>
</tr>
</tbody>
</table>

--- Note ---

The **FPSID** is not accessible at EL0.

If **HCPTR.\{TCP11, TCP10\}** traps accesses to SIMD and floating-point functionality, then for a read of **FPSID**, that trap has priority over this trap.

When the **FPSID** is accessible, a **VMSR** **FPSID**, <Rt> instruction is permitted but is ignored. The execution of this **VMSR** instruction is not trapped by the ID group 0 trap.

**ID group 1, Implementation identification registers**

These registers often provide coarse-grained identification mechanisms for implementation-specific features.

Table G1-49 on page G1-9019 shows the registers that are in ID group 1 for traps to Hyp mode, and how the exceptions are reported in HSR.

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Group 1 registers</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure EL1</td>
<td>TCMTR, TLBTR, REVIDR, AIDR</td>
<td>Trapped MCR or MRC access (coproc==0b1111), using EC value 0x03</td>
</tr>
</tbody>
</table>

**ID group 2, Cache identification registers**

These registers describe and control the cache implementation.

Table G1-50 on page G1-9019 shows the registers that are in ID group 2 for traps to Hyp mode, and how the exceptions are reported in HSR.

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Group 2 registers</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure EL0 and EL1</td>
<td>CTR, CCSIDR, CLIDR, CSSELR, and, if implemented, CCSIDR2.</td>
<td>Trapped MCR or MRC access (coproc==0b1111), using EC value 0x03</td>
</tr>
</tbody>
</table>
ID group 3, Detailed feature identification registers

These registers provide detailed information about the features of the implementation.

--- Note ---

These registers are called the CPUID registers. There is no requirement for this trap to apply to those registers that the CPUID Identification Scheme defines as reserved. See The CPUID identification scheme on page G4-4993.

Table G1-51 on page G1-9020 shows the registers that are in ID group 3 for traps to Hyp mode, and how the exceptions are reported in HSR.

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Group 3 registers</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure EL1</td>
<td>MVFR0, MVFR1, MVFR2.</td>
<td>Trapped VMRS access for ID group traps, using EC value 0x88</td>
</tr>
<tr>
<td></td>
<td>ID_PFR0, ID_PFR1, ID_DFR0, ID_AFR0.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ID_MMFRO, ID_MMFRO, ID_MMFRO, ID_MMFRO.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ID_ISRAR0, ID_ISRAR1, ID_ISRAR2, ID_ISRAR3, ID_ISRAR4, ID_ISRAR5.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ID_MMFR4, ID_ISRAR6, ID_DFR1 are trapped, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether HCR.TID3 traps accesses.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>It is IMPLEMENTATION DEFINED whether HCR.TID3 traps MRC accesses to registers with coproc==0b1111 to encodings in the following range that are not already mentioned in this table CRn == c0, opc1 == 0, CRn == {c2-c7}, opc2 == {0-7}.</td>
<td></td>
</tr>
</tbody>
</table>

If HCPTR traps accesses to SIMD and floating-point functionality, then for reads of MVFR0, MVFR1, and MVFR2, that trap has priority over this trap.

Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions

HCR. {TWE, TWI} trap Non-secure EL0 and EL1 execution of WFE and WFI instructions to Hyp mode:

HCR.TWE:

1  Any attempt to execute a WFE instruction at Non-secure EL0 or EL1 is trapped to Hyp mode, if the instruction would otherwise have caused the PE to enter a low-power state.

0  This control has no effect on Non-secure EL0 or EL1 execution of WFE instructions.

HCR.TWI:

1  Any attempt to execute a WFI instruction at Non-secure EL0 or EL1 is trapped to Hyp mode, if the instruction would otherwise have caused the PE to enter a low-power state.

0  This control has no effect on Non-secure EL0 or EL1 execution of WFI instructions.

Table G1-52 on page G1-9020 shows how the exceptions are reported in HSR.

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Trapped instructions</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure EL0 and EL1</td>
<td>WFE</td>
<td>Trapped WFE or WFI instruction, using EC value 0x01</td>
</tr>
<tr>
<td></td>
<td>WFI</td>
<td></td>
</tr>
</tbody>
</table>
The attempted execution of a conditional \texttt{WFE} or \texttt{WFI} instruction is only trapped if the instruction passes its Condition code check.

\begin{quote}
\underline{Note}
\end{quote}

Since a \texttt{WFE} or \texttt{WFI} can complete at any time, even without a \texttt{Wakeup} event, the traps on \texttt{WFE} of \texttt{WFI} are not guaranteed to be taken, even if the \texttt{WFE} or \texttt{WFI} is executed when there is no \texttt{Wakeup} event. The only guarantee is that if the instruction does not complete in finite time in the absence of a \texttt{Wakeup} event, the trap will be taken.

For more information about these instructions, and when they can cause the PE to enter a low-power state, see:
\begin{itemize}
  \item \textit{Wait For Event and Send Event} on page \pageref{Wait For Event and Send Event}.
  \item \textit{Wait For Interrupt} on page \pageref{Wait For Interrupt}.
\end{itemize}

\section*{Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Activity Monitors registers}

If the Activity Monitors Extension is implemented, \texttt{HCPTR.TAM} traps Non-secure EL0 and EL1 accesses to the Activity Monitors registers to Hyp mode:

\begin{itemize}
  \item \texttt{1} Non-secure EL0 and EL1 accesses to all Activity Monitors registers are trapped to Hyp mode.
  \item \texttt{0} This control has no effect on Non-secure EL0 and EL1 accesses to the Activity Monitors registers.
\end{itemize}

\begin{quote}
\underline{Note}
\end{quote}

\begin{itemize}
  \item EL2 does not provide traps on Activity Monitor register accesses through the optional memory-mapped external interface.
  \item If the Activity Monitors Extension is not implemented, \texttt{HCPTR.TAM} is \texttt{RES0}.
\end{itemize}

Table \ref{Table G1-53} on page \pageref{Table G1-53} shows the registers for which accesses are trapped, and how the exceptions are reported in HSR.

\begin{table}[h]
\centering
\caption{Register accesses trapped to Hyp mode when \texttt{HDCR.(TPM, TPMCR)} are \texttt{1}}
\begin{tabular}{|l|l|l|}
\hline
\textbf{Traps from} & \textbf{Trap control} & \textbf{Registers} & \textbf{Syndrome reporting in HSR} \\
\hline
Non-secure & TPM & AMCFGR, AMCGCR, AMCNTENCLR0, & Trapped \texttt{MCR} or \texttt{MRC} access (coproc==0b1111), using EC value 0x03. \\
EL0 and EL1 & & AMCNTENCLRI, AMCNTENSET0, AMCNTENSET1, & \\
& & AMCR, AMEVTYPE0\texttt{<>}, or AMEVTYPE1\texttt{<>}. & \\
& & AMEVCTR0\texttt{<>} or AMEVCTR1\texttt{<>}. & Trapped \texttt{MCR} or \texttt{MRC} access (coproc==0b1111), using EC value 0x04. \\
\hline
\end{tabular}
\label{Table G1-53}
\end{table}

\section*{General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers}

\texttt{HCPTR.(TCP11, TCP10)} trap Non-secure accesses to the SIMD and floating-point registers to Hyp mode:

\begin{itemize}
  \item \texttt{0b11} All Non-secure accesses to the SIMD and floating-point registers are trapped to Hyp mode. Trapped instructions generate:
    \begin{itemize}
      \item \texttt{Hyp Trap} exceptions, if the exception is taken from Non-secure EL0 or EL1.
      \item Undefined Instruction exceptions taken to Hyp mode, if the exception is taken from EL2.
    \end{itemize}
  \item \texttt{0b00} This control has no effect on Non-secure accesses to the SIMD and floating-point registers.
\end{itemize}
Note

Software must set HCPTR.TCP11 and HCPTR.TCP10 to the same value.

Table G1-54 on page G1-9022 shows the registers for which accesses are trapped, and how the exceptions are reported in HSR.

Table G1-54 Register accesses trapped to Hyp mode when HCPTR.{TCP11, TCP10} are both 0b11

| Traps from Non-secure state | Registers FPSID, MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers. See Advanced SIMD and floating-point System registers on page G1-8998. | Syndrome reporting in HSR Trapped access to SIMD and floating-point register, resulting from HCPTR, using EC value 0x07

---

a. VMSR accesses to the FPSID are ignored, but for the purposes of this trap the architecture defines a VMSR access to the FPSID from EL1 or higher as an access to a SIMD and floating-point register.

If EL3 is implemented and is using AArch32, and NSACR.{cp11, cp10} are both set to 0, then HCPTR.{TCP11, TCP10} behave as RAO/WI, regardless of their actual value.

For more information about SIMD and floating-point support, see Advanced SIMD and floating-point support on page G1-8996.

Enabling access to the SIMD and floating-point registers

FPEXC.EN is an instruction enable that enables access to the SIMD and floating-point registers from all Exception levels, but does not control the following:

• VMSR accesses to the FPEXC or FPSID.
• VMSR accesses from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2.

FPEXC.EN is a PL1 control that also applies at EL2. See Enabling access to the SIMD and floating-point registers on page G1-9007.

Traps to Hyp mode of Non-secure accesses to Advanced SIMD functionality

If implemented as an RW field, HCPTR.TASE can trap Non-secure execution of Advanced SIMD instructions to Hyp mode, as follows. This trap applies only when HCPTR.{TCP11, TCP10} are both 0:

1 Any attempt to execute an Advanced SIMD instruction in Non-secure state is trapped to Hyp mode. Trapped instructions generate:
   • Hyp Trap exceptions, if the exception is taken from Non-secure EL0 or EL1.
   • Undefined Instruction exceptions taken to Hyp mode, if the exception is taken from EL2.

0 This control has no effect on Non-secure execution of Advanced SIMD instructions.

When the control is not implemented, meaning the HCPTR.TASE field is RAZ/WI, the HCPTR does not provide a trap to Hyp mode of the Non-secure execution of Advanced SIMD instructions, other than the HCPTR.{TCP11, TCP10} trap that applies to Non-secure execution of both Advanced SIMD and floating-point instructions.
Table G1-55 Instructions trapped to Hyp mode when HCPTR.TASE is set to 1

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Instructions</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure state</td>
<td>All Advanced SIMD instructions that are not also floating-point instructions.</td>
<td>Trapped access to SIMD and floating-point register, resulting from HCPTR, using EC value 0x07</td>
</tr>
</tbody>
</table>

Table G1-56 Register accesses trapped to Hyp mode when HCPTR.TCPAC is 1

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Register</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure EL1</td>
<td>CPACR</td>
<td>Trapped MCR or MRC access to System register with coproc==0b1111, using EC value 0x03</td>
</tr>
</tbody>
</table>

---

**Note**

- The CPACR is not accessible at EL0.
- *In Armv7 and earlier versions of the Arm architecture, one use of the CPACR is to identify what coprocessor, or conceptual coprocessor, functionality is implemented. Legacy software might use this identification mechanism. A hypervisor can use this trap to emulate this mechanism. See *Background to the System register interface* on page G1-8994 for more information about this functionality.*

---

**Traps to Hyp mode of Non-secure System register accesses to trace registers**

If implemented, the HCPTR.TTA control traps System register accesses to the trace registers from Non-secure state to Hyp mode, as follows:

1. Non-secure System register accesses to the trace registers are trapped to Hyp mode. Trapped instructions generate:
   - Hyp Trap exceptions, if the exception is taken from Non-secure EL0 or EL1.
   - Undefined Instruction exceptions taken to Hyp mode, if the exception is taken from EL2.
2. This control has no effect on Non-secure System register accesses to the trace registers.

If the HCPTR.TTA control is not implemented, then HCPTR.TTA is RAO/WI. See the register description for more information.

---

**Note**

- System register accesses to the trace registers use the System register (coproc==0b1110) encoding space.
The ETMv4 architecture does not permit EL0 to access the trace registers. If the architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED. A resulting Undefined Instruction exception is higher priority than an HCPTR.TTA Hyp Trap exception.

EL2 does not provide traps on trace register accesses through the optional memory-mapped external debug interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see Register access instructions on page G1-9002.

Table G1-57 on page G1-9024 shows the registers for which accesses are trapped to Hyp mode when HCPTR.TTA is 1, and how the exceptions are reported in HSR.

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Registers</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure state</td>
<td>System register accesses to all implemented</td>
<td>For accesses using:</td>
</tr>
<tr>
<td></td>
<td>trace registers</td>
<td>• MCR or MRC instructions, trapped MCR or MRC access (coproc==0b1110), using EC value 0x05.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MCRR or MRRC instructions, trapped MCRR or MRRC access (coproc==0b1110), using EC value 0x0C.</td>
</tr>
</tbody>
</table>

If EL3 is implemented and is using AArch32, and NSACR.NSTRCDIS is 1, then HCPTR.TTA behaves as RAO/WI, regardless of its actual value. This behavior applies, also, when the HCPTR.TTA control is not implemented.

Traps to Hyp mode of Non-secure System register accesses to trace filter control registers

If implemented, the HDCR.TTRF control traps System register accesses to the trace filter control registers from Non-secure state to Hyp mode, as follows:

1 Non-secure System register accesses at EL1 to the trace filter control registers are trapped to Hyp mode. Trapped instructions generate Hyp Trap exceptions.

0 This control has no effect on Non-secure System register accesses to the trace registers.

Table G1-58 on page G1-9024 shows the registers for which accesses are trapped to Hyp mode when HDCR.TTRF is 1, and how the exceptions are reported in HSR.

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Registers</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure state</td>
<td>TRFCR</td>
<td>For accesses using MCR or MRC instructions, trapped MCR or MRC access (coproc==0b1111), using EC value 0x03.</td>
</tr>
</tbody>
</table>

General trapping to Hyp mode of Non-secure EL0 and EL1 accesses to System registers in the (coproc==0b1111) encoding space

HSTR.\{T0-T3, T5-T13, T15\} trap Non-secure EL0 and EL1 accesses, using MCR, MRC, MCRR, or MRRC instructions, to the System registers in the (coproc==0b1111) encoding space, by:

• The value of the CRn argument to the instruction, for MCR and MRC instructions.

• The value of the CRn argument to the instruction, for MCRR and MRRC instructions.

This applies for the set of CRn, or CRn, values \{c0-c3, c5-c13, c15\}. 

Table G1-57 Register accesses trapped to Hyp mode when HCPTR.TTA is 1

Table G1-58 Register accesses trapped to Hyp mode when HDCR.TTRF is 1
When an HSTR.Tn trap control is:

1  Non-secure EL1 accesses to the corresponding System registers in the (coproc == 0b1111) encoding space are trapped to Hyp mode.
   EL0 accesses to the corresponding System registers are trapped to Hyp mode if they would not be UNDEFINED if the bit was zero.

0  This control has no effect on Non-secure EL0 or EL1 accesses to System registers.

_____ Note _____

This means that a Hyp Trap exception taken from EL1 to EL2, generated because of a configuration setting in HSTR.Tn, is a higher priority exception than an Undefined Instruction exception generated because either the System register encoding is unallocated or because a register is never accessible at Non-secure EL1. As Synchronous exception prioritization for exceptions taken to AArch32 state on page G1-8933 shows, this is an exception to the general exception prioritization rules that prioritize most Undefined Instruction exceptions taken to Undefined mode above traps to EL2. This prioritization includes any access from Non-secure EL1 to a register that is only accessible in Secure state. So, for example, an access to the SCR from Non-secure EL1:

- When the value of HSTR.T1 is 0, generates an Undefined Instruction exception.
- When the value of HSTR.T1 is 1, generates a Hyp Trap exception.

Table G1-59 on page G1-9025 shows the accesses that are trapped, and how the exceptions are reported in HSR.

Table G1-59 Accesses trapped to Hyp mode when an HSTR.Tn trap is enabled

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Trap control</th>
<th>Trapped accesses</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure EL0 and EL1 a</td>
<td>Tn</td>
<td>MCR and MRC instructions, with coproc set to 0b1111 and CRn set to n</td>
<td>Trapped MCR or MRC access (coproc == 0b1111), using EC value 0x03</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MCRR and MRRC instructions, with coproc set to 0b1111 and CRm set to n</td>
<td>Trapped MCRR or MRRC access (coproc == 0b1111), using EC value 0x04</td>
</tr>
</tbody>
</table>

a. As described in this section, traps from EL1 apply whenever the value of HSTR.Tn is 1. Traps from EL0 apply only if the value of HSTR.Tn is 1 and the access would not be UNDEFINED if the value of HSTR.Tn was 0.

For example, when HSTR.T7 is 1, considering only accesses from Non-secure EL1:

- Any 32-bit access from a Non-secure PL1 mode using an MRC or MCR instruction with coproc set to 0b1111 and CRn set to c7, is trapped to Hyp mode.
- Any 64-bit access from a Non-secure PL1 mode using an MRRC or MCRR instructions with coproc set to 0b1111 and CRm set to c7, is trapped to Hyp mode.

_____ Note _____

- Bits[4,14] of the HSTR are reserved, RES0. Although the Generic Timer control registers are implemented in the coproc == 0b1111 encoding space with CRn==c14 for an MRC or MCR access, EL2 does not provide a trap on accesses to the Generic Timer System registers.
- An implementation might provide additional controls, in IMPLEMENTATION DEFINED registers, to provide finer-grained control of trapping of IMPLEMENTATION DEFINED features.
**System registers in the (coproc==0b1111) encoding space with IMPLEMENTATION DEFINED access permission from EL0**

For a System register in the (coproc==0b1111) encoding space, that is accessed using a CRn or CRm value that can be trapped by a HSTR.Tn control, if an access to the register from User mode is UNDEFINED when the value of the corresponding HSTR.Tn trap control is 0, then when that HSTR.Tn trap control is 1, it is IMPLEMENTATION DEFINED whether an access from Non-secure User mode generates:

- A Hyp Trap exception.
- An Undefined Instruction exception taken to Non-secure Undefined mode.

--- Note ---

Arm expects that trapping to Hyp mode of Non-secure User mode accesses to System register in the (coproc==0b1111) encoding space will be unusual, and used only when the hypervisor must virtualize User mode operation. Arm recommends that, whenever possible, Non-secure User mode accesses to System register in the (coproc==0b1111) encoding space behave as they would if the processor did not implement EL2, generating an Undefined Instruction exception taken to Non-secure Undefined mode if the architecture does not support the User mode access.

---

**Traps to Hyp mode of Non-secure System register accesses to debug registers**

HDCR.(TDRA, TDOSA, TDA) trap Non-secure System register accesses to debug registers to Hyp mode, as follows:

- HDCR.(TDRA, TDA) trap Non-secure EL0 and EL1 accesses.
- HDCR.TDOSA traps Non-secure EL1 accesses.

--- Note ---

EL2 does not provide traps of debug register accesses through the optional memory-mapped external debug interface.

---

System register accesses to the debug registers can have side-effects. When a System register access is trapped to Hyp mode, no side-effects occur before the exception is taken to Hyp mode. See Register access instructions on page G1-9002.

Table G1-60 on page G1-9026 shows the subsections that list the accesses trapped. The subsections describe how the traps are reported in HSR.

**Table G1-60 Traps of Non-secure EL0 and EL1 accesses to debug registers**

<table>
<thead>
<tr>
<th>Trap control</th>
<th>Subsection</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDCR.TDRA</td>
<td>Trapping Non-secure System register accesses to Debug ROM registers on page G1-9026</td>
</tr>
<tr>
<td>HDCR.TDOSA</td>
<td>Trapping Non-secure System register accesses to powerdown debug registers on page G1-9027</td>
</tr>
<tr>
<td>HDCR.TDA</td>
<td>Trapping general Non-secure System register accesses to debug registers on page G1-9027</td>
</tr>
</tbody>
</table>

--- Note ---

System register accesses to debug registers use the (coproc==0b1110) encoding space.

---

**Trapping Non-secure System register accesses to Debug ROM registers**

HDCR.TDRA traps Non-secure EL0 and EL1 System register accesses to the Debug ROM registers to Hyp mode:

1 Non-secure EL0 or EL1 System register accesses to the Debug ROM registers are trapped to Hyp mode.
This control has no effect on Non-secure EL0 and EL1 System register accesses to the Debug ROM registers.

Table G1-61 on page G1-9027 shows the register accesses that are trapped, and how the exceptions are reported in HSR:

**Table G1-61 Register accesses trapped to Hyp mode when HDCR.TDRA is 1**

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Registers</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
</table>
| Non-secure EL0 and EL1 | DBGDRAR, DBGDSAR | For accesses using:  

  - MCR or MRC instructions, trapped MCR or MRC access (coproc==0b1110), using EC value 0x05.  
  - MRRC instructions, trapped MRRC access (coproc==0b1110), using EC value 0x0C. |

If HDCR.TDE or HCR.TGE is 1, behavior is as if HDCR.TDRA is 1 other than for the purpose of a direct read.

**Trapping Non-secure System register accesses to powerdown debug registers**

HDCR.TDOSA traps Non-secure EL1 System register accesses to the powerdown debug registers to Hyp mode:

- 1 Non-secure EL1 System register accesses to the powerdown debug registers are trapped to Hyp mode.

- 0 This control has no effect on Non-secure EL1 System register accesses to the powerdown debug registers.

Table G1-62 on page G1-9027 shows the register accesses that are trapped, and how the exceptions are reported in HSR.

**Table G1-62 Register accesses trapped to Hyp mode when HDCR.TDOSA is 1**

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Registers</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure EL1</td>
<td>DBGOSLSR, DBGOSLAR, DBGOSDLR, DBGPRCR</td>
<td>Trapped MCR or MRC access (coproc==0b1110), using EC value 0x05</td>
</tr>
</tbody>
</table>

These registers are not accessible at EL0.

If HDCR.TDE or HCR.TGE is 1, behavior is as if HDCR.TDOSA is 1 other than for the purpose of a direct read.

**Trapping general Non-secure System register accesses to debug registers**

HDCR.TDA traps Non-secure EL0 and EL1 System register accesses to the debug registers that are not mentioned in either of the following:

- Traps to Hyp mode of Non-secure System register accesses to debug registers on page G1-9026.

- Trapping Non-secure System register accesses to powerdown debug registers.

This means that HDCR.TDA traps to Hyp mode Non-secure EL0 and EL1 System register accesses to all debug registers except the following:

- Non-secure System register accesses to DBGDRAR or DBGDSAR. The HDCR.TDRA trap traps these accesses.

- Non-secure System register access to DBGOSLSR, DBGOSLAR, DBGOSDLR, or DBGPRCR. The HDCR.TDOSA trap traps these accesses.
HDCR.TDA does not trap accesses to DBGDTRTXint or DBGDTRRXint when the PE is in Debug state.

When HDCR.TDA is:

1  Non-secure EL0 or EL1 System register accesses to any of the registers shown in Table G1-63 on page G1-9028 are trapped to Hyp mode.

0  This control has no effect on Non-secure EL0 or EL1 System register accesses.

Table G1-63 on page G1-9028 shows how the exceptions are reported in HSR.

Table G1-63 Accesses trapped to Hyp mode when HDCR.TDA is 1

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Trapped accesses</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure EL0 and EL1</td>
<td>Accesses to the DBGIDIDR, DBGDSCRint, DBGDCCINT, DBGDTRRXint, DBGDTRTXint, DBGWFAR, DBGVCR, DBGDSCRext, DBGDTRXtext, DBGDTRRXext, DBGVBR&lt;n&gt;, DBGBCR&lt;n&gt;, DBGXVR&lt;n&gt;, DBGWCR&lt;n&gt;, DBGVR&lt;n&gt;, DBGCLAIMSET, DBGCLAIMCLR, DBGAUTHSTATUS, DBGDEVID, DBGDEVID1, DBGDEVID2, and DBGOSECCR</td>
<td>For accesses using MCR or MRC instructions, trapped MCR or MRC access (coproc==0b1110), using EC value 0x05</td>
</tr>
</tbody>
</table>

If HDCR.TDE or HCR.TGE is 1, behavior is as if HDCR.TDA is 1 other than for the purpose of a direct read.

Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the Generic Timer registers

CNTHCTL.{PL1PCEN, PL1PCTEN} trap Non-secure EL0 and EL1 accesses to the Generic Timer registers to Hyp mode, as follows:

• CNTHCTL.PL1PCEN traps Non-secure EL0 and EL1 accesses to the physical timer registers.
• CNTHCTL.PL1PCTEN traps Non-secure EL0 and EL1 accesses to the physical counter register.

For each of these controls:

1  This control has no effect on Non-secure EL0 and EL1 accesses to the registers shown in Table G1-64 on page G1-9028.

0  Non-secure EL0 and EL1 accesses are trapped to Hyp mode.

Table G1-64 on page G1-9028 shows the registers for which accesses are trapped, and how the exceptions are reported in HSR.

Table G1-64 Register accesses trapped to Hyp mode by CNTHCTL trap controls

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Trap control</th>
<th>Registers</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
</table>
| Non-secure EL0 and EL1 | PL1PCEN | CNTP_CTL, CNTP_CVAL, CNTP_TVVAL | For accesses using:
• MCR or MRC instructions, trapped MCR or MRC access (coproc==0b1111), using EC value 0x03
• MCRR or MRRC instructions, trapped MCRR or MRRC access (coproc==0b1111), using EC value 0x04 |
| PL1PCTEN | CNTPCT | Trapped MCRR or MRRC access (coproc==0b1110), using EC value 0x04 |
Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Performance Monitors registers

If the Performance Monitors Extension is implemented, HDCR.{TPM, TPMCR} trap Non-secure EL0 and EL1 accesses to the Performance Monitors registers to Hyp mode:

**HDCR.TPM:**
- 1: Non-secure EL0 and EL1 accesses to all Performance Monitors registers are trapped to Hyp mode.
- 0: This control has no effect on Non-secure EL0 and EL1 accesses to the Performance Monitors registers.

**HDCR.TPMCR:**
- 1: Non-secure EL0 and EL1 accesses to the Performance Monitors Control Register are trapped to Hyp mode.

--- Note ---
The conditions for this trap are identical to those for the trap controlled by HDCR.TPM

- 0: This control has no effect on Non-secure EL0 and EL1 accesses to the Performance Monitors Control Registers.

--- Note ---
- EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.
- If the Performance Monitors Extension is not implemented, HDCR.{TPM, TPMCR} are RES0.

Table G1-65 on page G1-9029 shows the registers for which accesses are trapped, and how the exceptions are reported in HSR.

**Table G1-65 Register accesses trapped to Hyp mode when HDCR.{TPM, TPMCR} are 1**

<table>
<thead>
<tr>
<th>Traps from Non-secure EL0 and EL1</th>
<th>Trap control</th>
<th>Registers</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM</td>
<td>PMCR, PMCSENTSET, PMCSENTCLR, PMOVSR, PMSWINC, PMSLRR, PMCEID0, PMCEID1, PMCCNTR, PMXEVTPR, PMXEVNCNTR, PMUSERNR, PMINTENSEN, PMINTENCLR, PMOVSET, PMEVCNTR&lt;n&gt;, PMEVTYPER&lt;n&gt;, PMCCFILTER</td>
<td>For accesses using:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• MCR or MRC instructions, trapped MCR or MRC access (coproc==0b1111), using EC value 0x03.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• MCRR or MRRC instructions, trapped MCRR or MRRC access (coproc==0b1111), using EC value 0x04.</td>
</tr>
<tr>
<td>TPMCR</td>
<td>PMCR</td>
<td></td>
<td>Trapped MCR or MRC access (coproc==0b1111), using EC value 0x03</td>
</tr>
</tbody>
</table>

--- Note ---
HDCR.HPMN affects whether a counter can be accessed from Non-secure EL1 or EL0. See the register description of HDCR for more information.
Traps to Hyp mode of Non-secure EL1 accesses to the RAS error record registers

HCR2.TERR traps Non-secure EL1 accesses to the RAS ER* registers to Hyp mode. For more information on the RAS ER* registers, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile.

Table G1-66 Register accesses trapped to Hyp mode when HCR2.TERR is 1

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Trap control</th>
<th>Registers</th>
<th>Syndrome reporting in HSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure EL0 and EL1</td>
<td>TERR</td>
<td>ERRIDR, ERRSELR, ERXADDR, ERXADDR2, ERXCTRLR, ERXCTRLR2, ERXFR, ERXFR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, ERXMISC4, ERXMISC5, ERXMISC6, ERXMISC7, ERXSTATUS.</td>
<td>For accesses using:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• MCR or MRC instructions, trapped MCR or MRC access (coproc==0b1111), using EC value 0x03.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• MCRR or MRRC instructions, trapped MCRR or MRRC access (coproc==0b1111), using EC value 0x04.</td>
</tr>
</tbody>
</table>

G1.21.4 EL3 configurable controls

Table G1-67 on page G1-9030 shows the System registers that contain these controls.

Table G1-67 System registers that contain instruction enables and disables, and trap controls

<table>
<thead>
<tr>
<th>Register name</th>
<th>Register description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCR</td>
<td>Secure Configuration Register</td>
</tr>
<tr>
<td>NSACR</td>
<td>Non-secure Access Control Register</td>
</tr>
</tbody>
</table>

Table G1-68 on page G1-9030 summarizes the controls.

Table G1-68 EL3 Instruction enables and disables, and trap controls

<table>
<thead>
<tr>
<th>Control</th>
<th>Type of control</th>
<th>Trap</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCR.{TWE, TWI}</td>
<td>T</td>
<td>Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode on page G1-9032</td>
</tr>
<tr>
<td>SCR.HCE</td>
<td>E</td>
<td>Enabling EL2 and Non-secure EL1 execution of HVC instructions on page G1-9033</td>
</tr>
<tr>
<td>SCR.SCD</td>
<td>D</td>
<td>Disabling SMC instructions on page G1-9033</td>
</tr>
<tr>
<td>NSACR.NSTRCDIS</td>
<td>D</td>
<td>Disabling Non-secure System register access to the trace registers on page G1-9033</td>
</tr>
<tr>
<td>SDCR.TTRF</td>
<td>T</td>
<td>Traps to Monitor mode of System register accesses to the trace filter control registers on page G1-9034</td>
</tr>
<tr>
<td>NSACR.{cp11, cp10}</td>
<td>E</td>
<td>Enabling Non-secure access to SIMD and floating-point functionality on page G1-9034</td>
</tr>
<tr>
<td>NSACR.NSASEDIS</td>
<td>D</td>
<td>Disabling Non-secure access to Advanced SIMD functionality on page G1-9034</td>
</tr>
<tr>
<td>SCR.TERR</td>
<td>T</td>
<td>Traps to Monitor mode of accesses to RAS error record registers on page G1-9032</td>
</tr>
</tbody>
</table>

a. See Table G1-69 on page G1-9031.
The AArch32 System Level Programmers' Model
G1.21 Configurable instruction enables and disables, and trap controls

Table G1-69 Control types, for AArch32 EL3 controls

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Type</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Disable</td>
<td>Instruction enables and instruction disables on page G1-9001</td>
</tr>
<tr>
<td>E</td>
<td>Enable</td>
<td>Instruction enables and instruction disables on page G1-9001</td>
</tr>
<tr>
<td>T</td>
<td>Trap</td>
<td>Trap controls on page G1-9001</td>
</tr>
</tbody>
</table>

Also see the following:
- Register access instructions on page G1-9002.
- Instructions that fail their Condition code check.
- Trapping to EL3 of instructions that are UNPREDICTABLE on page G1-9032.

Instructions that fail their Condition code check

For UNDEFINED instructions that fail their Condition code check, see Conditional execution of undefined instructions on page G1-8966.

For an instruction that has a Monitor trap set that fails its Condition code check:

- Unless the trap description states otherwise, it is IMPLEMENTATION DEFINED whether the instruction:
  - Generates a Monitor Trap exception.
  - Executes as a NOP.

Any implementation must be consistent in its handling of instructions that fail their Condition code check. This means that:

- Whenever a Monitor trap is set on such an instruction it must either:
  - Always generate a Monitor trap exception.
  - Always treat the instruction as a NOP.

- The IMPLEMENTATION DEFINED part of the requirements of Conditional execution of undefined instructions on page G1-8966 must be consistent with the handling of Monitor traps on instructions that fail their Condition code check. Table G1-70 on page G1-9031 shows this:

Table G1-70 Consistent handling of instructions that fail their Condition code check

<table>
<thead>
<tr>
<th>Behavior of conditional UNDEFINED instruction&lt;sup&gt;a&lt;/sup&gt;</th>
<th>Monitor trap on instruction that fails its Condition code check&lt;sup&gt;b&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executes as a NOP</td>
<td>Generates as a NOP</td>
</tr>
<tr>
<td>Generates an Undefined Instruction exception</td>
<td>Generates a Monitor trap exception</td>
</tr>
</tbody>
</table>

a. As defined in Conditional execution of undefined instructions on page G1-8966. In Non-secure EL0 and EL1 modes, this applies only if no Monitor trap is set for the instruction, otherwise see the behavior in the other column of the table.
b. For a trapped instruction executed in a Non-secure EL1 or EL0 mode.

--- Note ---

When SCR {TWE, TWI} is set so that conditional WFE and WFI instructions are trapped to Monitor mode, the attempted execution of a conditional WFE or WFI instruction is only trapped if the instruction passes its Condition code check. See Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode on page G1-9032.
Trapping to EL3 of instructions that are UNPREDICTABLE

For an instruction that is UNPREDICTABLE, when the instruction is disabled or trapped then it is CONstrained
UNPREDICTABLE whether execution of the instruction generates a Monitor Trap exception.

Note

UNPREDICTABLE and CONstrained UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or lower Exception level using instructions that are not UNPREDICTABLE and are not CONstrained UNPREDICTABLE. This means that disabling or trapping an instruction changes the set of instructions that might be executed in modes other than Monitor mode. This affects, indirectly, the permitted behavior of UNPREDICTABLE and Constrained UNPREDICTABLE instructions.

If no instructions are trapped, the attempted execution of an UNPREDICTABLE instruction in a mode other than Monitor mode must not generate a Monitor Trap exception.

Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode

SCR{TWE, TWI} trap WFE and WFI instructions to Monitor mode:

\[
\begin{align*}
\text{SCR.TWE} & & 1 & & \text{Any attempt to execute a WFE instruction in any mode other than Monitor mode is trapped to Monitor mode, if the instruction would otherwise have caused the PE to enter a low-power state.} \\
& & 0 & & \text{This control has no effect on the execution of WFE instructions.}
\end{align*}
\]

\[
\begin{align*}
\text{SCR.TWI} & & 1 & & \text{Any attempt to execute a WFI instruction in any mode other than Monitor mode is trapped to Monitor mode, if the instruction would otherwise have caused the PE to enter a low-power state.} \\
& & 0 & & \text{This control has no effect on the execution of WFI instructions.}
\end{align*}
\]

For PL0 and PL1, these traps apply to WFE and WFI instruction execution in both Security states.

The attempted execution of a conditional WFE or WFI instruction is only trapped if the instruction passes its Condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE or WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about these instructions, and when they can cause the PE to enter a low-power state, see:

- [Wait For Event and Send Event](#)
- [Wait For Interrupt](#)

Traps to Monitor mode of accesses to RAS error record registers

SCR.TERR traps accesses to the RAS ER* registers from modes other than Monitor mode to Monitor mode.

Table G1-71 Register accesses trapped to EL3 when SCR.TERR is 1

<table>
<thead>
<tr>
<th>Traps from</th>
<th>Registers</th>
<th>Syndrome reporting in ESR_EL3</th>
</tr>
</thead>
</table>
| AArch32 state | ERRIDR, ERRSELR, ERXADDR, ERXADDR2, ERXCTLR, ERXCTLR2, ERXFRI, ERXFRI2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, ERXMISC4, ERXMISC5, ERXMISC6, ERXMISC7, ERXSTATUS. | For accesses using:
|             |           | • MCR or MRC instructions, trapped MCR or MRC access (coproc==0b1111), using EC value 0x03 |
|             |           | • MCR or MRC instructions, trapped MCR or MRC access, (coproc==0b1111) using EC value 0x04 |
This trap control applies to accesses from both Security states.

**Enabling EL2 and Non-secure EL1 execution of HVC instructions**

SCR.HCE enables EL2 and Non-secure EL1 execution of HVC instructions:

1  
HVC instruction execution is enabled at EL2 and Non-secure EL1.

0  
HVC instructions are:

- **Undefined** at Non-secure EL1. The Undefined Instruction exception is taken to Undefined mode.
- **Constrained Unpredictable** at EL2. The behavior must be one of the following:
  - The instruction is **undefined**.
  - The instruction executes as a NOP.

---

**Note**

- If EL2 is not implemented, SCR.HCE is RES0 and HVC is undefined.
- HVC instructions are always **Undefined** at EL0 and in Secure state.

**Disabling SMC instructions**

SCR.SCD disables SMC instructions:

1  
In Non-secure state

SMC instructions are **Undefined**. The Undefined Instruction exception is taken from the current Exception level to the current Exception level.

In Secure state

Behavior is one of the following:

- The instruction is **undefined**.
- The instruction executes as a NOP.

0  
SMC instructions are enabled.

---

**Note**

- SMC instructions are always **Undefined** at EL0.
- When the value of HCR.TSC is 1, any attempted execution of an SMC instruction at Non-secure EL1 is trapped to EL2, regardless of the value of SCR.SCD, see *Traps to Hyp mode of Non-secure EL1 execution of SMC instructions* on page G1-9017. As *Synchronous exception prioritization for exceptions taken to AArch32 state on page G1-8933* shows, this is an exception to the general exception prioritization rules that prioritize most Undefined Instruction exceptions taken to Undefined mode above traps to a higher Exception level.

**Disabling Non-secure System register access to the trace registers**

NSACR.NSTRCDIS disables Non-secure System register accesses to the trace registers, from all Privilege levels:

1  
Non-secure state accesses are disabled. Secure state accesses are enabled. If the PE is in Non-secure state:

- CPACR.TRCDIS behaves as RAO/WI, regardless of its actual value. See *Traps to Undefined mode of PL0 and PL1 System register accesses to trace registers* on page G1-9005. This behavior applies even if the CPACR.TRCDIS control is not implemented. See the referenced section for more information.

- HCPTR.TTA behaves as RAO/WI, regardless of its actual value. See *Traps to Hyp mode of Non-secure System register accesses to trace registers* on page G1-9023.

0  
There is no effect on accesses to CPACR.TRCDIS and HCPTR.TTA.
Note

- System register accesses to the trace registers use the \textit{(coproc==0b1111)} encoding space.
- \texttt{NSACR.NSTRCDIS} might be implemented as RAZ/WI. See the \texttt{NSACR} register description for more information.
- The ETMv4 architecture does not permit EL0 to access the trace registers. If the Armv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED.
- EL3 does not provide Non-secure access controls on trace register accesses through the optional memory-mapped external debug interface.

Traps to Monitor mode of System register accesses to the trace filter control registers

\texttt{SDCR.TTRF} traps any System register accesses to trace filter control registers to Monitor mode:

\begin{itemize}
  \item \texttt{1}: Any attempt to access a trace filter control register in any mode other than Monitor mode is trapped to Monitor mode.
  \item \texttt{0}: This control has no effect.
\end{itemize}

Enabling Non-secure access to SIMD and floating-point functionality

\texttt{NSACR.{cp11, cp10}} enable Non-secure access to the SIMD and floating-point registers, from all Privilege levels:

\begin{itemize}
  \item \texttt{0b11}: All accesses, from both Security states, are enabled.
  \item \texttt{0b00}: Non-secure state accesses are disabled. Secure state accesses are enabled. If the PE is in Non-secure state:
    \begin{itemize}
      \item \texttt{CPACR.{cp11, cp10}} behave as RAZ/WI. See \textit{Enabling PL0 and PL1 accesses to the SIMD and floating-point registers} on page G1-9006.
      \item \texttt{HCPTR.{TCP11, TCP10}} behave as RAO/WI. See \textit{General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers} on page G1-9021.
    \end{itemize}
\end{itemize}

Note

Software must set \texttt{NSACR.cp11} and \texttt{NSACR.cp10} to the same value.

For more information about SIMD and floating-point support, see \textit{Advanced SIMD and floating-point support} on page G1-8996.

Disabling Non-secure access to Advanced SIMD functionality

\texttt{NSACR.NSASEDIS} disables Non-secure accesses to the Advanced SIMD functionality, from all Privilege levels:

\begin{itemize}
  \item \texttt{1}: Non-secure state accesses are disabled. Secure accesses are enabled. If the PE is in Non-secure state:
    \begin{itemize}
      \item \texttt{CPACR.ASEDIS} behaves as RAO/WI. See \textit{Disabling PL0 and PL1 execution of Advanced SIMD instructions} on page G1-9007.
      \item \texttt{HCPTR.TASE} behaves as RAO/WI. See \textit{Traps to Hyp mode of Non-secure accesses to Advanced SIMD functionality} on page G1-9022.
    \end{itemize}
\end{itemize}

These behaviors apply even if one or both of the \texttt{CPACR.ASEDIS} and \texttt{HCPTR.TASE} controls is not implemented. See the referenced sections for more information.

\begin{itemize}
  \item \texttt{0}: There is no effect on \texttt{CPACR.ASEDIS} and \texttt{HCPTR.TASE}.
\end{itemize}

Pseudocode description of configurable instruction enables, disables, and traps

The pseudocode function \texttt{AArch32.CheckITEnabled()} checks whether the T32 IT instruction is enabled.
The pseudocode function `AArch32.CheckSETENDEnabled()` checks whether the SETEND instruction is disabled.

The pseudocode function for `AArch32.CheckForSMCF indefOrTrap()` checks for traps on an SMC instruction.

The `AArch32.CheckForWFxTrap()` pseudocode function checks for traps on WFE and WFI instructions:

**Pseudocode description of enabling SIMD and floating-point functionality**

The `AArch32.CheckAdvSIMDoorFPEnabled()` and `AArch32.CheckFPAdvSIMDTrap()` pseudocode functions take appropriate action if an SIMD or floating-point instruction is used when the SIMD and floating-point functionality is not enabled or is trapped.

The `CheckAdvSIMDoorVFPEnabled()`, `CheckAdvSIMDEnabled()`, and `CheckVFPEnabled()` wrapper functions support the `AArch32.CheckAdvSIMDoorFPEnabled()` and `AArch32.CheckFPAdvSIMDTrap()` functions.

The `AArch32.CheckAdvSIMDoorFPEnabled()`, `AArch32.CheckFPAdvSIMDTrap()`, `CheckAdvSIMDoorVFPEnabled()`, `CheckAdvSIMDEnabled()`, and `CheckVFPEnabled()` functions are described in Chapter J1 *Armv8 Pseudocode.*
G1.21 Configurable instruction enables and disables, and trap controls
Chapter G2
AArch32 Self-hosted Debug

When the PE is using self-hosted debug, it generates debug exceptions. This chapter describes the AArch32 self-hosted debug exception model. It is organized as follows:

Introductory information
- About self-hosted debug on page G2-9038.
- The debug exception enable controls on page G2-9042.

The debug Exception model
- Routing debug exceptions on page G2-9043.
- Enabling debug exceptions from the current Privilege level and Security state on page G2-9045.
- The effect of powerdown on debug exceptions on page G2-9047.
- Summary of permitted routing and enabling of debug exceptions on page G2-9048.
- Pseudocode description of debug exceptions on page G2-9050.

The debug exceptions
- Breakpoint Instruction exceptions on page G2-9051.
- Breakpoint exceptions on page G2-9054.
- Watchpoint exceptions on page G2-9079.
- Vector Catch exceptions on page G2-9093.

Synchronization requirements
The behavior of self-hosted debug after changes to System registers, or after changes to the authentication interface, but before a Context synchronization event guarantees the effects of the changes:
- Synchronization and debug exceptions on page G2-9100.
G2.1 About self-hosted debug

Self-hosted debug supports debugging through the generation and handling of debug exceptions, that are taken using the exception model described in:

- Chapter D1 The AArch64 System Level Programmers’ Model, if the exception is taken to AArch64 state.
- Chapter G1 The AArch32 System Level Programmers’ Model, if the exception is taken to AArch32 state.

This section introduces some terms used in describing self-hosted debug, and then introduces the debug exceptions. See:

- Definition of a debugger in the context of self-hosted debug.
- Context ID and Process ID.

G2.1.1 Definition of a debugger in the context of self-hosted debug

Within this chapter, debugger means that part of an operating system, or higher level of system software, that handles debug exceptions and programs the debug System registers. An operating system with rich application environments might provide debug services that support a debugger user interface executing at EL0. From the architectural perspective, the debug services are the debugger.

G2.1.2 Context ID and Process ID

In AArch32 state, the CONTEXTIDR identifies the current Context ID, that is used by:

- The debug logic, for breakpoint and watchpoint matching.
- Implemented trace logic, to identify the current process.

When using the Long-descriptor translation table format, the CONTEXTIDR has a single field, PROCID, that is defined as the Process Identifier (Process ID). Therefore, in AArch64 state, the Context ID and Process ID are identical when using this translation table format.

When using the Short-descriptor translation table format:

- CONTEXTIDR[31:0] defines the Context ID, that is used for breakpoint and watchpoint matching.
- CONTEXTIDR[31:8] defines the Process ID.
- CONTEXTIDR[7:0] defines the ASID. See Global and process-specific translation table entries on page G5-9216. This means that, when using the Short-descriptor translation table format, the ASID is always bits[7:0] of the Context ID.

G2.1.3 About debug exceptions

Debug exceptions occur during normal program flow if a debugger has programmed the PE to generate them. For example, a software developer might use a debugger contained in an operating system to debug an application. To do this, the debugger might enable one or more debug exceptions. The debug exceptions that can be generated in an AArch32 stage 1 translation regime are:

- Breakpoint Instruction exceptions on page G2-9039.
- Breakpoint exceptions on page G2-9039, generated by hardware breakpoints.
- Watchpoint exceptions on page G2-9040, generated by hardware watchpoints.
- Vector Catch exceptions on page G2-9040.

Note

In addition, Software Step exceptions can be generated in stage 1 of an AArch32 translation regime. However, these are always taken to AArch64 state. Software Step exceptions on page D2-4694 describes this.

The PE can only generate a particular debug exception when both:

1. Debug exceptions are enabled from the current Exception level and Security state.

   See Enabling debug exceptions from the current Privilege level and Security state on page G2-9045.

   Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.
2. A debugger has enabled that particular debug exception.

All of the debug exceptions except for Breakpoint Instruction exceptions have an enable control contained in the DBGDSCRext. See The debug exception enable controls on page G2-9042.

——— Note ————
If halting is allowed and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause entry to Debug state instead of causing debug exceptions. In Debug state, the PE is halted.

For the definition of halting is allowed, see Halting allowed and halting prohibited on page H2-10233.

——— Note ————
When a debug exception is taken to an Exception level that is using AArch32:
• If the debug exception is a Watchpoint exception, it is taken as a Data Abort exception.
• Otherwise, it is taken as a Prefetch Abort exception.

The following list summarizes each of the debug exceptions:

Breakpoint Instruction exceptions

Breakpoint instructions generate these. Breakpoint instructions are instructions that software developers can use to cause exceptions at particular points in the program flow.

The breakpoint instruction in the A32 and T32 instruction sets is BKPT #<immediate>. Whenever one of these is committed for execution, the PE takes a Breakpoint Instruction exception.

PE behavior
Breakpoint Instruction exceptions cannot be masked. The PE takes Breakpoint Instruction exceptions regardless of both of the following:
• The current Privilege level and AArch32 mode.
• The current Security state.

For more information, see Breakpoint Instruction exceptions on page G2-9051.

Breakpoint exceptions

The architecture provides 2-16 hardware breakpoints. These can be programmed to generate Breakpoint exceptions based on particular instruction addresses, or based on particular PE contexts, or both.

For example, a software developer might program a hardware breakpoint to generate a Breakpoint exception whenever the instruction with address 0x1000 is committed for execution.

The architecture supports the following types of hardware breakpoint for use in stage 1 of an AArch32 translation regime:

• Address:
  — Address Match.
  — Address Mismatch.

Comparisons are made with the virtual address of each instruction in the program flow.

• Context:
  — Context ID Match. Matches with the Context ID value held in the CONTEXTIDR.
  — VMID Match. Matches with the VMID value held in the VTTBR.
  — Context ID and VMID Match. Matches with both the Context ID and the VMID value.

An Address breakpoint can link to a Context breakpoint, so that the Address breakpoint only generates a Breakpoint exception if the PE is in a particular context when the address match or mismatch occurs.

A breakpoint generates a Breakpoint exception whenever an instruction that causes a match is committed for execution.

PE behavior
If halting is allowed and EDSCR.HDE is 1, hardware breakpoints cause entry to Debug state. That is, they halt the PE. See Chapter H2 Debug State.
Otherwise:

- If debug exceptions are enabled, hardware breakpoints cause Breakpoint exceptions.
- If debug exceptions are disabled, hardware breakpoints are ignored.

For more information, see *Breakpoint exceptions* on page G2-9054.

**Watchpoint exceptions**

The architecture provides 2-16 hardware watchpoints. These can be programmed to generate Watchpoint exceptions based on accesses to particular data addresses, or based on accesses to any address in a data address range.

For example, a software developer might program a hardware watchpoint to generate a Watchpoint exception on an access to any address in the data address range 0x1000 - 0x101F.

A hardware watchpoint can link to a hardware breakpoint if the hardware breakpoint is a *Linked Context* type. In this case, the watchpoint only generates a Watchpoint exception if the PE is in a particular context when the data address match occurs.

The smallest data address size that a watchpoint can be programmed to match on is a byte. A single watchpoint can be programmed to match on one or more bytes.

A watchpoint generates a Watchpoint exception whenever an instruction that initiates an access that causes a match is committed for execution.

**PE behavior**

If halting is allowed and EDSCR.HDE is 1, hardware watchpoints cause entry to Debug state. That is, they halt the PE. See Chapter H2 *Debug State*.

Otherwise:

- If debug exceptions are enabled, hardware watchpoints cause Watchpoint exceptions.
- If debug exceptions are disabled, hardware watchpoints are ignored.

For more information, see *Watchpoint exceptions* on page G2-9079.

**Vector Catch exceptions**

These are used to trap exceptions. The architecture provides two forms of vector catch, *address-matching* and *exception-trapping*. Only one form can be implemented.

Whichever form is implemented, a debugger must enable Vector Catch exceptions for one or more exception vectors by programming the DBGVCR. Generation of Vector Catch exceptions is then as follows:

- For the address-matching form, a Vector Catch exception is generated whenever the virtual address of an instruction matches a vector that Vector Catch exceptions are enabled for.
- For the Exception-trapping form, a Vector Catch exception is generated as part of exception entry for exception types that correspond to vectors that Vector Catch exceptions are enabled for.

**PE behavior**

If debug exceptions are:

- Enabled, Vector Catch exceptions can be generated.
- Disabled, vector catch is ignored.

For more information, see *Vector Catch exceptions* on page G2-9093.

Table G2-1 on page G2-9041 summarizes PE behavior and shows the location of the pseudocode for each of the debug exceptions.
### Table G2-1 PE behavior and pseudocode for each of the debug exceptions

<table>
<thead>
<tr>
<th>Debug exception</th>
<th>PE behavior if debug exceptions are:</th>
<th>Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Enabled</td>
<td>Disabled</td>
</tr>
<tr>
<td>Breakpoint Instruction exception</td>
<td>Takes Prefetch Abort exception</td>
<td>Takes Prefetch Abort exception</td>
</tr>
<tr>
<td>Breakpoint exception</td>
<td>Takes Prefetch Abort exception^a</td>
<td>Ignored</td>
</tr>
<tr>
<td>Watchpoint exception</td>
<td>Takes Data Abort exception^a</td>
<td>Ignored</td>
</tr>
<tr>
<td>Vector Catch exception</td>
<td>Takes Prefetch Abort exception</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

^a If halting is allowed and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause the PE to enter Debug state instead of causing debug exceptions. See Chapter H2 Debug State.
The debug exception enable controls

The enable controls for each debug exception are as follows:

**Breakpoint Instruction exceptions**
None. Breakpoint Instruction exceptions are always enabled.

**Breakpoint exceptions**
DBGDSCRext.MDBGen, plus an enable control for each breakpoint, DBGBCR<n>.E.

**Watchpoint exceptions**
DBGDSCRext.MDBGen, plus an enable control for each watchpoint, DBGWCR<n>.E.

**Vector Catch exceptions**
DBGDSCRext.MDBGen.

In addition, for all debug exceptions other than Breakpoint Instruction exceptions, software must configure the controls that enable debug exceptions from the current Exception level and Security state. See *Enabling debug exceptions from the current Privilege level and Security state* on page G2-9045.

The PE cannot take a debug exception if debug exceptions are disabled from either the current Exception level or the current Security state.

Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.
G2.3 Routing debug exceptions

Debug exceptions are usually routed to Abort mode. However, if EL2 is implemented, the routing of debug exceptions depends on the **Effective values** of HDCR.TDE and HCR.TGE:

**If the Effective value of {HDCR.TDE, HCR.TGE} is not {0, 0}**

Debug exceptions taken from Non-secure state are routed to Hyp mode.

If EL2 is using AArch64 and FEAT_SEL2 is implemented, debug exceptions taken from Secure EL0 and Secure EL1 may be routed to Secure EL2. For more information, see Routing debug exceptions on page D2-4697.

**Otherwise**

In Non-secure state debug exceptions behave as follows:

- Debug exceptions taken from Non-secure EL1 and Non-secure EL0 are routed to Non-secure Abort mode.
- Breakpoint Instruction exceptions taken from Hyp mode are routed to Hyp mode.
- All other debug exceptions are disabled from Hyp mode.

---

**Note**

If EL2 is not implemented, the **Effective value** of HCR.TGE is 0 and the **Effective value** of HDCR.TDE is 0.

Table G2-2 on page G2-9043, Table G2-3 on page G2-9043, and Table G2-4 on page G2-9044 show the routing of debug exceptions taken from an Exception level that is using AArch32 to an Exception level that is using AArch64.

In these tables:

- **TDE** Means the logical OR of HDCR.TDE and HCR.TGE.
- **(Hyp mode)** Means:
  - All debug exceptions other than Breakpoint Instruction exceptions are disabled from this Privilege level.
  - Breakpoint Instruction exceptions taken from this Privilege level are taken to Hyp mode.

### Table G2-2 Routing when both EL3 and EL2 are implemented

<table>
<thead>
<tr>
<th>TDE</th>
<th>Target AArch32 mode when executing in:</th>
<th>Secure state</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Non-secure:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PL0</td>
<td>Non-secure Abort mode</td>
</tr>
<tr>
<td></td>
<td>PL1</td>
<td>Non-secure Abort mode</td>
</tr>
<tr>
<td></td>
<td>PL2</td>
<td>(Hyp mode)</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table G2-3 Routing when EL3 is implemented and EL2 is not implemented

<table>
<thead>
<tr>
<th>Target AArch32 mode when executing in:</th>
<th>Secure state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure state</td>
<td>Secure Abort mode</td>
</tr>
<tr>
<td>Secure state</td>
<td></td>
</tr>
</tbody>
</table>
### Table G2-4 Routing when EL3 is not implemented and EL2 is implemented

<table>
<thead>
<tr>
<th>TDE</th>
<th>Target AArch32 mode when executing in Non-secure:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PL0</td>
</tr>
<tr>
<td>0</td>
<td>Non-secure Abort mode</td>
</tr>
<tr>
<td>1</td>
<td>Hyp mode</td>
</tr>
</tbody>
</table>

### G2.3.1 Pseudocode description of routing debug exceptions

`DebugTarget()` returns the current debug target Exception level. `DebugTargetFrom()` returns the debug target Exception level for the specified Security state.
G2.4 Enabling debug exceptions from the current Privilege level and Security state

A debug exception can only be taken if all of the following are true:

- The OS Lock is unlocked.
- DoubleLockStatus() == FALSE.
- The debug exception is enabled from the current Privilege level.
- The debug exception is enabled from the current Security state.

Table G2-5 on page G2-9045 shows when debug exceptions are enabled from the current Privilege level.

<table>
<thead>
<tr>
<th>Current Privilege level</th>
<th>Breakpoint Instruction exceptions</th>
<th>All other debug exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL2</td>
<td>Enabled</td>
<td>Disabled</td>
</tr>
<tr>
<td>PL1</td>
<td>Enabled</td>
<td>Enabled</td>
</tr>
<tr>
<td>PL0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table G2-6 on page G2-9045 shows when debug exceptions are enabled from the current Security state.

<table>
<thead>
<tr>
<th>Current Security state</th>
<th>Breakpoint Instruction exceptions</th>
<th>All other debug exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-secure</td>
<td>Enabled</td>
<td>Enabled from PL1 and PL0 only.</td>
</tr>
<tr>
<td>Secure</td>
<td>Enabled</td>
<td>Depends on SDCR.SPD and SDER.SUIDEN. See Disabling debug exceptions from Secure state on page G2-9045.</td>
</tr>
</tbody>
</table>

G2.4.1 Disabling debug exceptions from Secure state

If EL3 is implemented, software executing at EL3 can enable or disable all debug exceptions taken from Secure PL1 other than Breakpoint Instruction exceptions, by using one of:

- The Secure Privileged Debug field, SDCR.SPD, if EL3 is using AArch32.
- The AArch32 Secure Privileged Debug field, MDCR_EL3.SPD32, if EL3 is using AArch64.

If debug exceptions are disabled from Secure PL1, software executing at Secure PL1 can set the Secure User Invasive Debug Enable bit, SDER.SUIDEN, to 1 to enable all debug exceptions taken from Secure PL0 other than Breakpoint Instruction exceptions.

--- Note ---
Breakpoint Instruction exceptions are always enabled.

--- Note ---
The Armv8-A architecture does not support disabling debug in Non-secure state.

--- Note ---
If the boot software that is executed when reset is deasserted programs SUIDEN and SPD so that all debug exceptions are disabled from Secure state, software operating at EL3 never has to switch any of the debug registers between the Security states.
G2.4.2   Pseudocode description of enabling debug exceptions

AArch64.GenerateDebugExceptions() determines whether debug exceptions are enabled from the current Exception level and Security state. AArch64.GenerateDebugExceptionsFrom() determines whether debug exceptions are enabled from the specified Exception level and Security state.
G2.5 The effect of powerdown on debug exceptions

Debug OS Save and Restore sequences on page H6-10342 describes the powerdown save routine and the restore routine.

When executing either routine, software must use the OS Lock to disable generation of all of the following:

- Breakpoint exceptions.
- Watchpoint exceptions.
- Vector Catch exceptions.

This is because the generation of these exceptions depends on the state of the debug registers, and the state of the debug registers might be lost over these routines.

Debug exceptions other than Breakpoint Instruction exceptions are enabled only if both the OS Lock is unlocked and DoubleLockStatus() == FALSE.

Breakpoint Instruction exceptions are enabled regardless of the state of the OS Lock and the OS Double Lock.
### G2.6 Summary of permitted routing and enabling of debug exceptions

Behavior is as follows:

#### Breakpoint Instruction exceptions

These are always enabled, regardless of the current Privilege level and Security state. Table G2-7 on page G2-9048 shows the routing of these. In the table, n/a means not applicable.

#### Table G2-7 Routing of Breakpoint Instruction exceptions

<table>
<thead>
<tr>
<th>Current Security state</th>
<th>HDCR.TDEa:</th>
<th>Target when enabled from:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>PL0</td>
</tr>
<tr>
<td>Secure</td>
<td>X</td>
<td>Secure Abort modeb</td>
</tr>
<tr>
<td>Non-secure</td>
<td>0</td>
<td>Non-secure Abort mode</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Hyp mode</td>
</tr>
</tbody>
</table>

a. If EL2 is not implemented, behavior is as if the value of this bit is 0. Otherwise, if the value of HCR.TGE is 1, HDCR.TDE is treated as being 1 other than for a direct read of HDCR.

b. If EL3 is implemented and is using AArch32, Secure Abort mode is at EL3. Otherwise, Secure Abort mode is at EL1.

#### All other debug exceptions

The enabling and permitted routing is controlled by all of the following:

- SDCR.SPD.
- SDER.SUIDEN.
- HDCR.TDE.
- The IMPLEMENTATION DEFINED authentication interface.

Table G2-8 on page G2-9048 shows the valid combinations of the values of SDCR.SPD, SDER.SUIDEN, HDCR.TDE, and, in the Auth on page G2-9048 column, the input from the IMPLEMENTATION DEFINED authentication interface described by the pseudocode function AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled(). For each combination, the table shows where debug exceptions are enabled from and where they are taken to.

In the table, n/a means not applicable and a dash, -, means that debug exceptions are disabled from that Exception level.

#### Table G2-8 Breakpoint, Watchpoint, and Vector Catch exceptions

<table>
<thead>
<tr>
<th>Debug state</th>
<th>Locka</th>
<th>Current Security state</th>
<th>SPDb</th>
<th>Authc</th>
<th>SUIDENb</th>
<th>TDEd</th>
<th>Target AArch32 mode when enabled from:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PL0</td>
</tr>
<tr>
<td>Yes</td>
<td>X</td>
<td>X</td>
<td>0bXX</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>No</td>
<td>TRUE</td>
<td>X</td>
<td>0bXX</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>No</td>
<td>FALSE</td>
<td>Secure</td>
<td>0b00</td>
<td>FALSE</td>
<td>0</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>No</td>
<td>FALSE</td>
<td>Secure</td>
<td>0b00</td>
<td>FALSE</td>
<td>1</td>
<td>X</td>
<td>Secure Abort modec</td>
</tr>
<tr>
<td>No</td>
<td>FALSE</td>
<td>Secure</td>
<td>0b00</td>
<td>TRUE</td>
<td>X</td>
<td>X</td>
<td>Secure Abort modec</td>
</tr>
<tr>
<td>No</td>
<td>FALSE</td>
<td>Secure</td>
<td>0b10</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>-</td>
</tr>
</tbody>
</table>
Table G2-8 Breakpoint, Watchpoint, and Vector Catch exceptions (continued)

<table>
<thead>
<tr>
<th>Debug state</th>
<th>Lock⁵</th>
<th>Current Security state</th>
<th>SPD⁶</th>
<th>Auth⁷</th>
<th>SUIDEN⁷</th>
<th>TDE⁸</th>
<th>Target AArch32 mode when enabled from:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FALSE</td>
<td>Secure</td>
<td>0b10</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>Secure mode</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>Secure</td>
<td>0b11</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Secure mode</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>Non-secure</td>
<td>0bXX</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>Non-secure mode</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>Non-secure</td>
<td>0bXX</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>Hyp mode</td>
</tr>
</tbody>
</table>

a. The value of (OSLSR_EL1.OSLK == 1 || DoubleLockStatus()).
b. If EL3 is not implemented, behavior is as if SPD is 0b11 and SUIDEN is 0b0.
c. See the text that introduces this table for an explanation of the Auth column. An entry of TRUE indicates that the authentication mechanism permits the debug exceptions to be taken to their default target PE mode.
d. If HCR.TGE is 1, this bit is treated as being 1 other than for a direct read of HDCR. If EL2 is not implemented, behavior is as if TDE is 0.
e. If EL3 is implemented and is using AArch32, Secure Abort mode is at EL3. Otherwise, Secure Abort mode is at EL1.
G2.7  Pseudocode description of debug exceptions

AArch32.DebugFault() returns a FaultRecord() that indicates that a memory access has generated a debug exception.

The AArch32.Abort() function processes FaultRecord(), as described in Abort exceptions on page G4-9144, and generates:

• Data Abort exceptions for watchpoints.
• Prefetch Abort exceptions for all other debug exceptions.
G2.8 Breakpoint Instruction exceptions

This section describes Breakpoint Instruction exceptions in an AArch32 translation regime.

________ Note __________

When the PE is executing in EL0 using AArch32 and EL1 is using AArch64, it is using the AArch64 EL1&0 translation regime. A T32 or A32 BKPT instruction executed at EL0 can generate a Breakpoint Instruction exception that is taken to an Exception level that is using AArch64. For more information about the handling of these exceptions, see Breakpoint Instruction exceptions on page D2-4705.

It contains the following subsections:

• About Breakpoint Instruction exceptions.
• Breakpoint instruction in the A32 and T32 instruction sets.
• BKPT instructions as the first instruction in an IT block on page G2-9052.
• Exception syndrome information and preferred return address for a BKPT instruction on page G2-9052.
• Pseudocode description of Breakpoint Instruction exceptions on page G2-9053.

G2.8.1 About Breakpoint Instruction exceptions

A breakpoint is an event that results from the execution of an instruction, based on either:

• The instruction address, the PE context, or both. This type of breakpoint is called a hardware breakpoint.
• The instruction itself. That is, the instruction is a breakpoint instruction. These can be included in the program that the PE executes. This type of breakpoint is called a software breakpoint.

Breakpoint Instruction exceptions, that this section describes, are software breakpoints. Breakpoint exceptions on page G2-9054 describes hardware breakpoints.

There is no enable control for Breakpoint Instruction exceptions. They are always enabled, and cannot be masked.

A Breakpoint Instruction exception is generated whenever a breakpoint instruction is committed for execution, regardless of all of the following:

• The current Exception level.
• The current Security state.
• Whether the debug target Exception level, EL_D, is using AArch64 or AArch32.

________ Note __________

• EL_D is the Exception level that debug exceptions are targeting. See Enabling debug exceptions from the current Privilege level and Security state on page G2-9045.
• Debuggers using breakpoint instructions must be aware of the rules for concurrent modification and execution of instructions. See Concurrent modification and execution of instructions on page B2-156.

G2.8.2 Breakpoint instruction in the A32 and T32 instruction sets

The breakpoint instruction, in both instruction sets, is:

• BKPT #<immediate>

For details of the instruction encoding, see BKPT on page F5-7507.

About whether the BKPT instruction is conditional

In the T32 instruction set, BKPT instructions are always unconditional.
In the A32 instruction set:
• If the Condition code field is AL, the BKPT instruction is unconditional.
• If the Condition code field is anything other than AL, behavior is CONstrained UNPREDICTABLE, and is one of the following:
  — The instruction is UNDEFINED.
  — The instruction is treated as a NOP instruction.
  — The instruction is executed unconditionally.
  — The instruction is executed conditionally.

G2.8.3 BKPT instructions as the first instruction in an IT block

If the first instruction in an IT block is a T32 BKPT instruction, then in an implementation that supports the ITD control, if ITD field that applies to the current Exception level is:

0  The BKPT instruction generates a Breakpoint Instruction exception.

1  The combination of IT instruction and BKPT instruction is UNDEFINED. Either the IT instruction or the BKPT instruction generates an Undefined Instruction exception.

In such an implementation, to ensure consistent behavior when making the first instruction in one or more IT blocks a BKPT instruction, the debugger must replace the IT instruction.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0.

The ITD control fields are:
HSCTLR.ITD  Applies to execution at EL2 when EL2 is using AArch32.
SCTLR.ITD   Applies to execution at EL0 or EL1 when EL1 is using AArch32.
SCTLR_EL1.ITD  Applies to execution at EL0 using AArch32 when EL1 is using AArch64.

Note

T32 BKPT instructions are always unconditional, even when they are inside an IT block. See:
• Disabling or enabling PL0 and PL1 use of AArch32 optional functionality on page G1-9004.
• Disabling or enabling EL2 use of AArch32 optional functionality on page G1-9013.

G2.8.4 Exception syndrome information and preferred return address for a BKPT instruction

See the following:
• Exception syndrome information for a Breakpoint Instruction exception.
• Preferred return address for a Breakpoint Instruction exception on page G2-9053.

Note

Usually, the term exception syndrome is used only for exceptions taken to Hyp mode, or to AArch64 state. The referenced section uses the term more generally, to include exception information reported in the IFSR.

Exception syndrome information for a Breakpoint Instruction exception

The PE takes a Breakpoint Instruction exception as either:
• A Prefetch Abort exception if it is taken to PL1. In this case, it is taken to Abort mode.
• A Hyp Trap exception, if it is taken to PL2 because either HCR.TGE or HDCR.TDE is 1. In this case, it is taken to Hyp mode.
If the exception is taken to:

**PL1 Abort mode**

The PE sets all of the following:
- DBGDSCRext.MOE to 0b0011, to indicate a Breakpoint Instruction exception.
- IFSR.FS to the code for a debug, 0b00010.
- The IFAR with an UNKNOWN value.

**PL2 Hyp mode**

The PE does all of the following:
- Records information about the exception in the Hypervisor Syndrome Register, HSR. See Table G2-9 on page G2-9053.
- Sets DBGDSCRext.MOE to 0b0011, to indicate a Breakpoint Instruction exception.
- Sets the HIFAR to an unknown value.

---

Table G2-9 Information recorded in the HSR

<table>
<thead>
<tr>
<th>HSR field</th>
<th>Information recorded</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Exception Class, EC</em></td>
<td>The PE sets this to the code for a Prefetch Abort exception routed to Hyp mode, 0x20.</td>
</tr>
<tr>
<td><em>Instruction Length, IL</em></td>
<td>The PE sets this to:</td>
</tr>
<tr>
<td></td>
<td>• 0 for a T32 <em>BKPT</em> instruction.</td>
</tr>
<tr>
<td></td>
<td>• 1 for an A32 <em>BKPT</em> instruction.</td>
</tr>
<tr>
<td><em>Instruction Specific Syndrome, ISS</em></td>
<td></td>
</tr>
<tr>
<td>ISS[24:10]</td>
<td>RES0.</td>
</tr>
<tr>
<td>ISS[9]</td>
<td><em>External Abort type</em> (EA). The PE sets this to 0.</td>
</tr>
<tr>
<td>ISS[8:6]</td>
<td>RES0.</td>
</tr>
<tr>
<td>ISS[5:0]</td>
<td><em>Instruction Fault Status Code</em> (IFSC). The PE sets this to the code for a debug exception, 0b100010.</td>
</tr>
</tbody>
</table>

---

**Note**

For information about how debug exceptions can be routed to PL2, see Routing debug exceptions on page G2-9043.

---

**Preferred return address for a Breakpoint Instruction exception**

The preferred return address is the address of the breakpoint instruction, not the next instruction. This is different to the behavior of other exception-generating instructions, like SVC.

**G2.8.5 Pseudocode description of Breakpoint Instruction exceptions**

`AArch32 SoftwareBreakpoint()` generates a Prefetch Abort exception that is taken from AArch32 state.
G2.9 Breakpoint exceptions

This section describes Breakpoint exceptions in stage 1 of an AArch32 translation regime.

The PE is using an AArch32 translation regime when it is executing either:
• At EL1 or higher in an Exception level that is using AArch32.
• At EL0 using AArch32 when EL1 is using AArch32.

This section contains the following subsections:
• About Breakpoint exceptions.
• Breakpoint types and linking of breakpoints on page G2-9055.
• Execution conditions for which a breakpoint generates Breakpoint exceptions on page G2-9063.
• Breakpoint instruction address comparisons on page G2-9066.
• Breakpoint context comparisons on page G2-9071.
• Using breakpoints on page G2-9072.
• Exception syndrome information and preferred return address for a Breakpoint exception on page G2-9077.
• Pseudocode description of Breakpoint exceptions taken from AArch32 state on page G2-9078.

G2.9.1 About Breakpoint exceptions

A breakpoint is an event that results from the execution of an instruction, based on either:
• The instruction address, the PE context, or both. This type of breakpoint is called a hardware breakpoint.
• The instruction itself. That is, the instruction is a breakpoint instruction. These can be included in the program that the PE executes. This type of breakpoint is called a software breakpoint.

Breakpoint exceptions are generated by Breakpoint debug events. Breakpoint debug events are generated by hardware breakpoints. Software breakpoints are described in Breakpoint Instruction exceptions on page G2-9051.

An implementation can include between 2-16 hardware breakpoints. DBGIDR.BRPs shows how many are implemented.

To use an implemented hardware breakpoint, a debugger programs the following registers for the breakpoint:
• The Breakpoint Control Register, DBGBCR<\text{n}>. This contains controls for the breakpoint, for example an enable control.
• The Breakpoint Value Register, DBGBVR<\text{n}>. This holds a value used for breakpoint matching, that is one of:
  — An instruction virtual address.
  — A Context ID.
• If EL2 is implemented, the Breakpoint Extended Value Register, DBGBXVR<\text{n}>, that holds a VMID value used for breakpoint matching.

These registers are numbered, so that:
• DBGBCR1, DBGBVR1, and DBGBXVR1 are for breakpoint number one.
• DBGBCR2, DBGBVR2, and DBGBXVR2 are for breakpoint number two.
• ...
• ...
• DBGBCR<\text{n}>, DBGBVR<\text{n}>, and DBGBXVR<\text{n}> are for breakpoint number <\text{n}>.

A debugger can link a breakpoint that is programmed with an address and a breakpoint that is programmed with anything other than an address together, so that a Breakpoint debug event is only generated if both breakpoints match.

For each instruction in the program flow, all of the breakpoints are tested. When a breakpoint is tested, it generates a Breakpoint debug event if all of the following are true:
• The breakpoint is enabled. That is, the breakpoint enable control for it, DBGBCR<\text{n}>.E, is 1.
• The conditions specified in the DBGBCR<\text{n}> are met.
• The comparisons with the values held in one or both of the DBGBVR<\text{n}> and DBGBXVR<\text{n}>, as applicable, are successful.
• If the breakpoint is linked to another breakpoint, the comparisons made by that other breakpoint are also successful.
• The instruction is committed for execution.

If all of these conditions are met, the breakpoint generates the Breakpoint debug event regardless of the following:
• Whether the instruction passes its Condition code check.
• The instruction type.

If halting is allowed and EDSCR.HDE is 1, Breakpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are
• Enabled, Breakpoint debug events generate Breakpoint exceptions
• Disabled, Breakpoint debug events are ignored.

Note

The remainder of this Breakpoint exceptions section, including all subsections, describes breakpoints as generating Breakpoint exceptions. However, the behavior described also applies if breakpoints are causing entry to Debug state.

The debug exception enable controls on page G2-9042 describes the enable controls for Breakpoint debug events.

G2.9.2 Breakpoint types and linking of breakpoints

Each implemented breakpoint is one of the following:

• A context-aware breakpoint. This is a breakpoint that can be programmed to generate a Breakpoint exception on any one of the following:
  — An instruction address match.
  — An instruction address mismatch.
  — A Context ID match, with the value held in the CONTEXTIDR.
  — A VMID match, with the value held in the VTTBR.
  — Both a Context ID match and a VMID match.

• A breakpoint that is not context-aware. These can only be programmed to generate a Breakpoint exception on an instruction address match or an instruction address mismatch.

DBGDIDR.CTX_CMPs shows how many of the implemented breakpoints are context-aware breakpoints. At least one implemented breakpoint must be context-aware. The context-aware breakpoints are the highest numbered breakpoints.

Any breakpoint that is programmed to generate a Breakpoint exception on an instruction address match or mismatch is categorized as an Address breakpoint. Breakpoints that are programmed to match on anything else are categorized as Context breakpoints.

When a debugger programs a breakpoint to be an Address or a Context breakpoint, it must also program that breakpoint so that it is either:
• Used in isolation. In this case, the breakpoint is called an Unlinked breakpoint.
• Enabled for linking to another breakpoint. In this case, the breakpoint is called a Linked breakpoint.

By linking an Address breakpoint and a Context breakpoint together, the debugger can create a breakpoint pair that only generates a Breakpoint exception if the PE is in a particular context when an instruction address match or mismatch occurs. For example, a debugger might:

1. Program breakpoint number one to be a Linked Address Match breakpoint.
2. Program breakpoint number five to be a Linked Context ID Match breakpoint.
3. Link these two breakpoints together. A Breakpoint exception is only generated if both the instruction address matches and the Context ID matches.

The Breakpoint Type field for a breakpoint, DBGBCR<n>.BT, controls the breakpoint type and whether the breakpoint is enabled for linking. If BT[0] is 1, the breakpoint is enabled for linking.

Address breakpoints can be programmed to generate Breakpoint exceptions on addresses that are halfword-aligned but not word-aligned. This makes it possible to breakpoint on T32 instructions. See Specifying the halfword-aligned address that an Address breakpoint matches on on page G2-9066.

Rules for linking breakpoints

The rules for breakpoint linking are as follows:

- Only Linked breakpoint types can be linked.
- Any type of Linked Address breakpoint can link to any type of Linked Context breakpoint. The Linked Breakpoint Number field, DBGBCR<n>.LBN, for the Linked Address breakpoint specifies the particular Linked Context breakpoint that the Linked Address breakpoint links to, and:
  - DBGBCR<n>.{SSC, HMC, PMC} for the Linked Address breakpoint define the execution conditions that the breakpoint pair generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions on page G2-9063.
  - DBGBCR<n>.{SSC, HMC, PMC} for the Linked Context breakpoint are ignored.
- Linked Context breakpoint types can only be linked to. The LBN field for Context breakpoints is therefore ignored.
- Linked Address breakpoints cannot link to watchpoints. The LBN field can therefore only specify another breakpoint.
- If a Linked Address breakpoint links to a breakpoint that is not context-aware, the behavior of the Linked Address breakpoint is CONSTRAINED UNPREDICTABLE. See Other usage constraints for Address breakpoints on page G2-9076.
- If a Linked Address breakpoint links to an Unlinked Context breakpoint, the Linked Address breakpoint never generates any Breakpoint exceptions.
- Multiple Linked Address breakpoints can link to a single Linked Context breakpoint.
  
  __________ Note __________

  Multiple Linked watchpoints can also link to a single Linked Context breakpoint. Watchpoint exceptions on page G2-9079 describes watchpoints.

These rules mean that a single Linked Context breakpoint might be linked to by all, or any combination of, the following:

- Multiple Linked Address Match breakpoints.
- Multiple Linked Address Mismatch breakpoints.
- Multiple Linked watchpoints.

It is also possible that a Linked Context breakpoint might have no breakpoints or watchpoints linked to it.

Figure G2-1 on page G2-9057 shows an example of permitted breakpoint and watchpoint linking.
In Figure G2-1, each Linked Address breakpoint can only generate a Breakpoint exception if the comparisons made by both it, and the Linked Context breakpoint that it links to, are successful. Similarly, each Linked watchpoint can only generate a Watchpoint exception if the comparisons made by both it, and the Linked Context breakpoint that it links to, are successful.

**Breakpoint types defined by DBGBCRn.BT**

The following list provides more detail about each breakpoint type:

0b0000, **Unlinked Address Match breakpoint**

Generation of a Breakpoint exception depends on both:

- \text{DBGBCR}<n>\.\{SSC, HMC, PMC\}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See \textit{Execution conditions for which a breakpoint generates Breakpoint exceptions} on page G2-9063.

- A successful address match, as described in \textit{Breakpoint instruction address comparisons} on page G2-9066.

\text{DBGBCR}<n>\.LBN for this breakpoint is ignored.
0b0001, Linked Address Match breakpoint

Generation of a Breakpoint exception depends on all of the following:

- **DBGBCR<n>.{SSC, HMC, PMC}** for this breakpoint. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions on page G2-9063.
- A successful address match defined by this breakpoint, as described in Breakpoint instruction address comparisons on page G2-9066.
- A successful context match defined by the Linked Context breakpoint that this breakpoint links to.

**DBGBCR<n>.LBN** for this breakpoint selects the Linked Context breakpoint that this breakpoint links to.

0b0010, Unlinked Context ID Match breakpoint

**BT == 0b0010** is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- **DBGBCR<n>.{SSC, HMC, PMC}.** These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions on page G2-9063.
- A successful Context ID match, as described in Breakpoint context comparisons on page G2-9071.

The value of **DBGBVR<n>.ContextID** is compared with the current Context ID.

**CONTEXTIDR_EL2** holds the current Context ID when all of:

- The implementation includes FEAT_VHE.
- EL2 is implemented and enabled in the current Security state.
- EL2 using AArch64 and the value of HCR_EL2.E2H is 1.
- The PE is executing at EL0 and HCR_EL2.TGE is 1, or the PE is executing at EL2.

Otherwise, **CONTEXTIDR** holds the current Context ID.

**DBGBCR<n>.{LBN, BAS}** for this breakpoint are ignored

0b0011, Linked Context ID Match breakpoint

**BT == 0b0011** is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, either:

- This breakpoint does not generate any Breakpoint exceptions, if no Linked breakpoints or Linked watchpoints link to it.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons on page G2-9066.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons on page G2-9071.

- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons on page G2-9083.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons on page G2-9071.

The value of **DBGBVR<n>.ContextID** is compared with the current Context ID.

**CONTEXTIDR_EL2** holds the current Context ID when all of:

- The implementation includes FEAT_VHE.
- EL2 is implemented and enabled in the current Security state.
- EL2 using AArch64 and the value of HCR_EL2.E2H is 1.
• The PE is executing at EL0 and HCR_EL2.TGE is 1, or the PE is executing at EL2.
 Otherwise, CONTEXTIDR holds the current Context ID.

DBGBCR<n>.{LBN, SSC, HMC, BAS PMC} for this breakpoint are ignored.

0b0100, Unlinked Address Mismatch breakpoint

Generation of a Breakpoint exception depends on both:
• DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions on page G2-9063.
• A successful address mismatch, as described in Breakpoint instruction address comparisons on page G2-9066.

DBGBCR<n>.LBN for this breakpoint is ignored.

0b0101, Linked Address Mismatch breakpoint

Generation of a Breakpoint exception depends on all of the following:
• DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions on page G2-9063.
• A successful address mismatch defined by this breakpoint, as described in Breakpoint instruction address comparisons on page G2-9066.
• A successful context match defined by the Linked Context breakpoint that this breakpoint links to.

DBGBCR<n>.LBN for this breakpoint selects the Linked Context breakpoint that this breakpoint links to.

0b0110, Unlinked CONTEXTIDR_EL1 Match breakpoint

BT == 0b0110 is a reserved value if either:
• The breakpoint is not a context-aware breakpoint.
• The implementation does not include FEAT_VHE.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:
• DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
• A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons on page G2-9071.

The Context ID check is made against the value in CONTEXTIDR, or CONTEXTIDR_EL1. The value of DBGBVR<n>.ContextID is compared with the Context ID value held in CONTEXTIDR or CONTEXTIDR_EL1.

DBGBCR<n>.{LBN, BAS} for this breakpoint are ignored.

0b0111, Linked CONTEXTIDR_EL1 Match breakpoint

BT == 0b0111 is a reserved value if either:
• The breakpoint is not a context-aware breakpoint.
• The implementation does not include FEAT_VHE.

For context-aware breakpoints, one of the following applies:
• If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.
• Generation of a Breakpoint exception depends on both:
  — A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons on page G2-9066.
— A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons on page G2-9071.

• Generation of a Watchpoint exception depends on both:
  — A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons on page G2-9083.
  — A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons on page G2-9071.

The Context ID check is made against the value in CONTEXTIDR, or CONTEXTIDR_EL1. The value of DBGVBR<n>.ContextID is compared with the Context ID value held in CONTEXTIDR or CONTEXTIDR_EL1.

DBGBCR<n>.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

0b1000, Unlinked VMID Match breakpoint

BT == 0b1000 is a reserved value if either:
• The breakpoint is not a context-aware breakpoint.
• EL2 is not implemented.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:
• DBGBCR<n>.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions on page G2-9063.
• A successful VMID match, as described in Breakpoint context comparisons on page G2-9071.

DBGBCR<n>.{LBN, BAS} for this breakpoint are ignored.

0b1001, Linked VMID Match breakpoint

BT == 0b1000 is a reserved value if either:
• The breakpoint is not a context-matching breakpoint.
• EL2 is not implemented.

For context-aware breakpoints, either:
• This breakpoint does not generate any Breakpoint exceptions, if no Linked breakpoints or Linked watchpoints link to it.
• Generation of a Breakpoint exception depends on both:
  — A successful instruction address match, defined by a Linked Address Match breakpoint that links to this breakpoint. See Breakpoint instruction address comparisons on page G2-9066.
  — A successful VMID match defined by this breakpoint.
• Generation of a Watchpoint exception depends on both:
  — A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons on page G2-9083.
  — A successful VMID match defined by this breakpoint, as described in Breakpoint context comparisons on page G2-9071.

DBGBCR<n>.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

0b1010, Unlinked Context ID and VMID Match breakpoint

BT == 0b1010 is a reserved value if either:
• The breakpoint is not a context-matching breakpoint.
• EL2 is not implemented.
For context-matching breakpoints, generation of a Breakpoint exception depends on all of the following:

- \texttt{DBGBCR<n>.\{SSC, HMC, PMC\}}. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See \textit{Execution conditions for which a breakpoint generates Breakpoint exceptions} on page \textit{G2-9063}.

- A successful Context ID match, as described in \textit{Breakpoint context comparisons} on page \textit{G2-9071}.

- A successful VMID match.

The value of \texttt{DBGBVR<n>.ContextID} is compared with \texttt{CONTEXTIDR}.

\textit{Breakpoint context comparisons} on page \textit{G2-9071} describes the requirements for a successful Context ID match and a successful VMID match.

\texttt{DBGBCR<n>.\{LBN, BAS\}} for this breakpoint are ignored.

\textbf{0b1011, Linked CONTEXTID and VMID Match breakpoint}

\texttt{BT == 0b1011} is a reserved value if either:

- The breakpoint is not a context-matching breakpoint.
- EL2 is not implemented.

For context-matching breakpoints, either:

- This breakpoint does not generate any Breakpoint exceptions, if no Linked breakpoints or Linked watchpoints link to it.

- Generation of a Breakpoint exception depends on all of the following:
  - A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see \textit{Breakpoint instruction address comparisons} on page \textit{G2-9066}.
  - A successful Context ID match defined by this breakpoint, as described in \textit{Breakpoint context comparisons} on page \textit{G2-9071}.
  - A successful VMID match defined by this breakpoint.

- Generation of a Watchpoint exception depends on all of the following:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see \textit{Watchpoint data address comparisons} on page \textit{G2-9083}.
  - A successful Context ID match defined by this breakpoint, as described in \textit{Breakpoint context comparisons} on page \textit{G2-9071}.
  - A successful VMID match defined by this breakpoint.

The value of \texttt{DBGBVR<n>.ContextID} is compared with \texttt{CONTEXTIDR}.

\textit{Breakpoint context comparisons} on page \textit{G2-9071} describes the requirements for a successful Context ID match and a successful VMID match by this breakpoint.

\texttt{DBGBCR<n>.\{LBN, SSC, HMC, BAS, PMC\}} for this breakpoint are ignored.

\textbf{0b1100, Unlinked CONTEXTIDR_EL2 Match breakpoint}

\texttt{BT == 0b1100} is a reserved value if:

- The breakpoint is not a context-aware breakpoint.
- \texttt{FEAT_VHE} is not implemented and \texttt{FEAT_Debugv8p2} is not implemented, which means the implementation does not include \texttt{CONTEXTIDR_EL2}.
- EL2 is not implemented.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- \texttt{DBGBCR<n>.\{SSC, HMC, PMC\}}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.

- A successful \texttt{CONTEXTIDR_EL2} match. The value of \texttt{DBGBVR<n>.ContextID2} is compared with the Context ID value held in \texttt{CONTEXTIDR_EL2}, as described in \textit{Breakpoint context comparisons} on page \textit{G2-9071}.
The check against CONTEXTIDR_EL2 means this breakpoint can be generated only if EL2 is implemented and enabled in the current Security state and EL2 is using AArch64.

--- Note ---

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

---

DBGBCR<\(n\)\.\{LBN, BAS\} for this breakpoint are ignored.

**0b1101, Linked CONTEXTIDR_EL2 Match**

BT == 0b1101 is a reserved value if:

- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.
- EL2 is not implemented.

For context-aware breakpoints, either:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons on page G2-9066.
  - A successful CONTEXTIDR_EL2 match, as described in Breakpoint context comparisons on page G2-9071.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons on page G2-9083.
  - A successful CONTEXTIDR_EL2 match. The value of DBGBVR<\(n\)\.ContextID2 is compared with the Context ID value held in CONTEXTIDR_EL2, as described in Breakpoint context comparisons on page G2-9071.

The check against the CONTEXTIDR_EL2 means the breakpoint or watchpoint can be generated only if EL2 is implemented and enabled in the current Security state and EL2 is using AArch64.

--- Note ---

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

---

DBGBCR<\(n\)\.\{LBN, SSC, HMC, BAS, PMC\} for this breakpoint are ignored.

**0b1110, Unlinked Full Context ID Match breakpoint**

BT == 0b1110 is a reserved value if:

- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.
- EL2 is not implemented.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<\(n\)\.\{SSC, HMC, PMC\}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful Context ID match, as described in Breakpoint context comparisons on page G2-9071.

The Context ID check is made by checking both:

- The value of DBGBVR<\(n\)\.ContextID against the value in CONTEXTIDR, or CONTEXTIDR_EL1.
The value of DBGBXVR<n>.ContextID2 against the value in CONTEXTIDR_EL2. Both comparisons must match for the check to succeed.

The check against the CONTEXTIDR_EL2 means this breakpoint can be generated only if EL2 is implemented and enabled in the current Security state and EL2 is using AArch64.

--- Note ---

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

---

DBGBCR<n>.{LBN, BAS} for this breakpoint are ignored.

---

0b1111, Linked Full Context ID Match breakpoint

- BT == 0b1111 is a reserved value if:
  - The breakpoint is not a context-aware breakpoint.
  - FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.
  - EL2 is not implemented.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any Breakpoint exceptions.

- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons on page G2-9066.
  - A successful Context ID match, as described in Breakpoint context comparisons on page G2-9071.

- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons on page G2-9083.
  - A successful Context ID match, as described in Breakpoint context comparisons on page G2-9071.

The Context ID check is made by checking both:

- The value of DBGBVR<n>.ContextID against the value in CONTEXTIDR, or CONTEXTIDR_EL1.

- The value of DBGBXVR<n>.ContextID2 against the value in CONTEXTIDR_EL2.

Both comparisons must match for the check to succeed.

The check against the CONTEXTIDR_EL2 means the breakpoint or watchpoint can be generated only if EL2 is implemented and enabled in the current Security state and EL2 is using AArch64.

--- Note ---

The operation of this breakpoint does not depend on the value of HCR_EL2.E2H.

---

DBGBCR<n>.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

---

See Reserved DBGBCR<n>.BT values on page G2-9074 for the behavior of breakpoints programmed with reserved BT values.

---

G2.9.3 Execution conditions for which a breakpoint generates Breakpoint exceptions

Each breakpoint can be programmed so that it only generates Breakpoint exceptions for certain execution conditions. For example, a breakpoint might be programmed to generate Breakpoint exceptions only when the PE is executing at PL0 in Secure state.
DBGBCR<n>.{SSC, HMC, PMC} define the execution conditions the breakpoint generates Breakpoint exceptions for, as follows:

**Security State Control, SSC**

Controls whether the breakpoint generates Breakpoint exceptions only in Secure state, only in Non-secure state, or in both Security states.

--- **Note** ---

This is determined by the Security state of the PE, not from the NS attribute returned by the translation of the virtual address on which the breakpoint is set.

**Higher Mode Control, HMC, and Privileged Mode Control, PMC**

HMC and PMC together control which AArch32 modes the breakpoint generates Breakpoint exceptions in.

Table G2-10 shows the valid combinations of the values of HMC, SSC, and PMC, and for each combination shows which Privilege levels breakpoints generate Breakpoint exceptions in.

In the table:

- **Y**  
  Means that a breakpoint programmed with the values of HMC, SSC and PMC shown in that row can generate Breakpoint exceptions in AArch32 modes at that Privilege level.

- **-**  
  Means that a breakpoint programmed with the values of HMC, SSC and PMC shown in that row cannot generate Breakpoint exceptions in AArch32 modes at that Privilege level.

A combination in Table G2-10 might be reserved if an Exception level or Security state is not implemented. For information about which combinations of HMC, SSC and PMC are reserved if an Exception level or Security state are not implemented, see *Reserved DBGBCR<n>.{SSC, HMC, PMC} values* on page G2-9075.

### Table G2-10 Summary of breakpoint HMC, SSC, and PMC encodings

<table>
<thead>
<tr>
<th>HMC</th>
<th>SSC</th>
<th>PMC</th>
<th>Security state the breakpoint is programmed to match in</th>
<th>PL2&lt;sup&gt;a&lt;/sup&gt;</th>
<th>PL1</th>
<th>PL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>Both</td>
<td>-</td>
<td>Y&lt;sup&gt;b&lt;/sup&gt;</td>
<td>Y</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>10</td>
<td></td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>11</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>00</td>
<td>Non-Secure</td>
<td>-</td>
<td>Y&lt;sup&gt;b&lt;/sup&gt;</td>
<td>Y</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>01</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>10</td>
<td></td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>11</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>00</td>
<td>Secure</td>
<td>-</td>
<td>Y&lt;sup&gt;b&lt;/sup&gt;</td>
<td>Y</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>01</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>10</td>
<td></td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>11</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>00</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>01</td>
<td></td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>10</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>11</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
Table G2-10 Summary of breakpoint HMC, SSC, and PMC encodings (continued)

<table>
<thead>
<tr>
<th>HMC</th>
<th>SSC</th>
<th>PMC</th>
<th>Security state the breakpoint is programmed to match in</th>
<th>PL2</th>
<th>PL1</th>
<th>PL0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>01</td>
<td>Secure</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>11</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>Both</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>11</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>00</td>
<td>Non-secure</td>
<td>Y</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>01</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>01</td>
<td>Secure</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>00</td>
<td>Both</td>
<td>Y</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>01</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>11</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

a. Debug exceptions are not generated at PL2 using AArch32. This means that these combinations of HMC, SSC, and PMC are only relevant if breakpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSC, and PMC that generate Breakpoint exceptions at PL2 using AArch32.

b. Only in User, System and Supervisor modes.

All combinations of HMC, SSC, and PMC that this table does not show are reserved. See Reserved HMC, SSC, and PMC combinations on page G2-9075.
G2.9.4 Breakpoint instruction address comparisons

Address comparisons are made for each instruction in the program flow. The following subsections describe the criteria for a successful address comparison, for:

- Address Match breakpoints.
- Address Mismatch breakpoints.

Address Match breakpoints

An address match comparison is successful if both:

- Bits [31:2] of the current instruction virtual address are equal to DBGBVR<n>[31:2].
- The word or halfword selected by DBGBCR<n>.BAS matches. That is, either:
  - DBGBCR<n>.BAS is programmed with 0b0011 or 0b1111, and the instruction is at a word-aligned address.
  - DBGBCR<n>.BAS is programmed with 0b1100, and the instruction is not at a word-aligned address.

See Specifying the halfword-aligned address that an Address breakpoint matches on.

Note

DBGBVR<n>[1:0] are RES0 and are ignored.

Address Mismatch breakpoints

An address mismatch comparison is successful if either:

- Bits [31:2] of the current instruction virtual address are not equal to DBGBVR<n>[31:2].
- The word or halfword selected by DBGBCR<n>.BAS does not match. That is, either:
  - DBGBCR<n>.BAS is programmed with 0b0011 or 0b1111, and the instruction is not at a word-aligned address.
  - DBGBCR<n>.BAS is programmed with 0b1100, and the instruction is at a word-aligned address.

See Specifying the halfword-aligned address that an Address breakpoint matches on.

Note

DBGBVR<n>[1:0] are RES0 and are ignored.

Address Mismatch breakpoints can be used to single-step through code. See Using an Address Mismatch breakpoint to single-step an instruction on page G2-9072.

Specifying the halfword-aligned address that an Address breakpoint matches on

For an Address breakpoint, a debugger can use the Byte Address Selection field, DBGBCR<n>.BAS, so that the address comparison is successful on one of:

- The whole word starting at address DBGBVR<n>[31:2]:00.
- The halfword starting at address DBGBVR<n>[31:2]:00.
- The halfword starting at address ((DBGBVR<n>[31:2]:00) + 2).

Note

The address programmed into the DBGBVR<n> must be word-aligned.
DBGBCR<n>.BAS can be used in both Address Match breakpoints and Address Mismatch breakpoints, as follows:

- For an Address Match breakpoint, DBGBCR<n>.BAS selects which halfword-aligned address the breakpoint must generate a Breakpoint exception for. This means that an address comparison is successful only if both of the following match:
  - The instruction address held in bits [31:2] of the DBGBVR<n>.
  - The halfword defined by the BAS field.

  That is, a successful address comparison = DBGBVR<n>[31:2] match AND BAS match.

- For an Address Mismatch breakpoint, DBGBCR<n>.BAS selects which halfword-aligned address the breakpoint must not generate a Breakpoint exception for. This means that an address comparison is successful if either or both of the following do not match:
  - The instruction address held in bits [31:2] of the DBGBVR<n>.
  - The halfword defined by the BAS field.

  That is, a successful address comparison = NOT (DBGBVR<n>[31:2] match AND BAS match).

The following subsections show the supported BAS values:

- Using the BAS field in Address Match breakpoints.
- Using the BAS field in Address Mismatch breakpoints on page G2-9069.

For Context breakpoints, DBGBCR<n>.BAS is RES1 and is ignored.

**Using the BAS field in Address Match breakpoints**

The supported BAS values are:

- **0b0000**
  This value is reserved. Behavior is a CONSTRAINED UNPREDICTABLE choice of:
  - The breakpoint is disabled.
  - The breakpoint behaves as if BAS is 0b0011, 0b1100, or 0b1111.

- **0b0011**
  The breakpoint generates a Breakpoint exception if an instruction with an address described as follows is committed for execution:
  - Bits [31:2] of the address equals DBGBVR<n>[31:2].
  - Bits [1:0] of the address are 0b00.

  This means that breakpoints programmed with this BAS value generate Breakpoint exceptions for all of the following:
  - 32-bit T32 instructions at word-aligned addresses.
  - 16-bit T32 instructions at word-aligned addresses.
  - A32 instructions. These are always at word-aligned addresses.

  However, Arm recommends that a debugger uses this BAS value only for T32 instructions.

  It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address (DBGBVR<n>[31:2]:00) - 2).

- **0b1100**
  The breakpoint generates a Breakpoint exception if an instruction with an address described as follows is committed for execution:
  - Bits [31:2] of the address equals DBGBVR<n>[31:2].
  - Bits [1:0] of the address are 0b10.

  This means that breakpoints programmed with this BAS value generate Breakpoint exceptions for both of the following:
  - 32-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.
  - 16-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.

  It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on the second halfword of a 32-bit T32 or A32 instruction starting at a word-aligned address.
The breakpoint generates a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] of the address equals DBGBVR<\text{n}>[31:2].
- Bits [1:0] of the address are 0b00.

This means that breakpoints programmed with this BAS value generate Breakpoint exceptions for all of the following:

- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, Arm recommends that a debugger uses this BAS value only for A32 instructions.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address ((DBGBVR<\text{n}>[31:2]:00) - 2).

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on a 32-bit T32 instruction or a 16-bit T32 instruction at the halfword-aligned address ((DBGBVR<\text{n}>[31:2]:00) + 2).

All other BAS values are reserved. For these reserved other values, DBGBCR<\text{n}>.BAS[3,1] ignore writes and read the same values as DBGBCR<\text{n}>[2,0] respectively. This means that the smallest instruction size a debugger can program breakpoints to match on is a halfword.

Figure G2-2 on page G2-9069 shows a summary of when breakpoints programmed with particular BAS values generate Breakpoint exceptions.

The figure contains four parts:

- A column showing the row number, on the left.
- An instruction set and instruction size table.
- A location of instruction figure.
- A BAS field values table, on the right.

To use the figure, read across the rows. For example:

- Row 2 shows that a breakpoint with a BAS value of 0b1100 generates Breakpoint exceptions for 16-bit T32 instructions starting at the halfword-aligned address ((DBGBVR<\text{n}>[31:2]:00) + 2).
- Row 6 shows that a breakpoint with a BAS value of either 0b0011 or 0b1111 generates Breakpoint exceptions for A32 instructions. A32 instructions are always at word-aligned addresses.

In the figure:

- **Yes** means that the breakpoint generates a Breakpoint exception.
- **No** means that the breakpoint does not generate a Breakpoint exception.
- **UNP** means that it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception. See Other usage constraints for Address breakpoints on page G2-9076.
Using the BAS field in Address Mismatch breakpoints

An Address Mismatch breakpoint generates Breakpoint exceptions for all instructions committed for execution, except the instruction whose address the breakpoint is programmed to match.

The supported BAS values are:

- **0b0000**: The breakpoint ignores the address held in the DBGBVR<\text{n}> and generates Breakpoint exceptions for all instruction addresses.

- **0b0011**: The breakpoint does not generate a Breakpoint exception if an instruction with an address described as follows is committed for execution:
  - Bits [31:2] of the address equals DBGBVR<\text{n}>[31:2].
  - Bits [1:0] of the address are 0b00.

  This means that breakpoints programmed with this BAS value do not generate Breakpoint exceptions for any of the following:
  - 32-bit T32 instructions at word-aligned addresses.
  - 16-bit T32 instructions at word-aligned addresses.
  - A32 instructions. These are always at word-aligned addresses.

  However, Arm recommends that a debugger uses this BAS value only for T32 instructions.

  It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address (DBGBVR<\text{n}>[31:2]:00 - 2).

- **0b1100**: The breakpoint does not generate a Breakpoint exception if an instruction with an address described as follows is committed for execution:
  - Bits [31:2] equals DBGBVR<\text{n}>[31:2].
  - Bits [1:0] of the address are 0b10.

  This means that breakpoints programmed with this BAS value do not generate Breakpoint exceptions for either of the following:
  - 32-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.
  - 16-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.
It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on the second halfword of a 32-bit T32 or A32 instruction at a word-aligned address.

0b1111

The breakpoint does not generate a Breakpoint exception if an instruction with an address described as follows is committed for execution:

• Bits [31:2] of the address equals DBGBVR<n>[31:2].
• Bits [1:0] of the address are 0b00.

This means that breakpoints programmed with this BAS value do not generate Breakpoint exceptions for any of the following:

• 32-bit T32 instructions at word-aligned addresses.
• 16-bit T32 instructions at word-aligned addresses.
• A32 instructions. These are always at word-aligned addresses.

However, Arm recommends that a debugger uses this BAS value only for A32 instructions.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the word-aligned address ((DBGBVR<n>[31:2]:00) - 2).

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on a 32-bit T32 instruction or a 16-bit T32 instruction at the halfword-aligned address ((DBGBVR<n>[31:2]:00) + 2).

All other BAS values are reserved. For these reserved other values, DBGBCR<n>.BAS[3,1] ignore writes and read the same values as DBGBCR<n>[2,0] respectively. This means that the smallest instruction size that a breakpoint can never generate a Breakpoint exception for is a halfword.

Figure G2-3 on page G2-9071 shows a summary of when breakpoints programmed with particular BAS values generate Breakpoint exceptions.

The figure contains four parts:

• A column showing the row number, on the left.
• An instruction set and instruction size table.
• A location of instruction figure.
• A BAS field values table, on the right.

To use the figure, read across the rows. For example:

• Row 1 shows that a breakpoint with a BAS value of 0b1100 generates Breakpoint exceptions for 16-bit T32 instructions starting at the word-aligned address held in the DBGBVR<n>.
• Row 5 shows that a breakpoint with a BAS value of 0b0011 generates Breakpoint exceptions for 32-bit T32 instructions starting at the halfword-aligned address immediately after the word aligned address held in the DBGBVR<n>.

In the figure:

Yes Means that the breakpoint does generate a Breakpoint exception.
No Means that the breakpoint does not generate a Breakpoint exception.
UNP Means that it CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception. See Other usage constraints for Address breakpoints on page G2-9076.
### G2.9.5 Breakpoint context comparisons

The breakpoint type defined by DBGBCR<\texttt{n}>::BT determines what context comparison is required, if any. Table G2-11 on page G2-9071 shows the BT values that require a comparison, and the match required for the comparison to be successful.

<table>
<thead>
<tr>
<th>DBGBCR&lt;\texttt{n}&gt;::BT</th>
<th>Test required for successful context comparison</th>
</tr>
</thead>
</table>
| 0b001x                 | - When FEAT\_VHE is implemented, EL2 is using AArch64, the Effective value of HCR\_EL2.E2H is 1, and either the PE is executing at EL0 with HCR\_EL2.TGE set to 1, or the PE is executing at EL2, CONTEXTIDR\_EL2 must match the DBGVR<\texttt{n}>.ContextID value.  
- Otherwise, CONTEXTIDR must match the DBGVR<\texttt{n}>.ContextID value. |
| 0b011x                 | CONTEXTIDR, or CONTEXTIDR\_EL1, must match the DBGVR<\texttt{n}>.ContextID value. |
| 0b100x                 | VTTBR\_VMID must match the DBGXVR<\texttt{n}>.VMID value. |
| 0b101x                 | CONTEXTIDR, or CONTEXTIDR\_EL1, must match the DBGVR<\texttt{n}>.ContextID value, and VTTBR\_VMID must match the DBGXVR<\texttt{n}>.VMID value. |
| 0b110x                 | CONTEXTIDR\_EL2 must match the DBGXVR<\texttt{n}>.ContextID2 value. |
| 0b111x                 | Both:  
- CONTEXTIDR, or CONTEXTIDR\_EL1, must match the DBGVR<\texttt{n}>.ContextID value.  
- CONTEXTIDR\_EL2 must match the DBGXVR<\texttt{n}>.ContextID2 value. |

No context comparison is required for other valid DBGBCR<\texttt{n}>::BT values.

Context breakpoints do not generate Breakpoint exceptions when any of:

- The comparison uses the value of CONTEXTIDR, or CONTEXTIDR\_EL1, and any of:
  - The PE is executing at EL3 using AArch64.
  - The PE is executing at EL2.
  - FEAT\_VHE is implemented, EL2 is using AArch64, EL2 is implemented and enabled in the current Security state, and HCR\_EL2.E2H.TGE == \{1, 1\}.

- The comparison uses the value of CONTEXTIDR\_EL2 and any of:
  - Neither FEAT\_VHE is implemented, nor FEAT\_Debugv8p2 is implemented.

---

*aa* 0 means the word-aligned address held in the DBGVRn. The other locations are as follows:

- -2 means ((DBGVRn[31:2]:00) - 2).
- -1 means ((DBGVRn[31:2]:00) - 1).
- ...  
- ...  
- +5 means ((DBGVRn[31:2]:00) + 5).

The solid areas show the location of the instruction.
— EL2 is either not implemented or not enabled in the current Security state.
— EL2 is using AArch32.

• The comparison uses the current VMID value and any of:
  — EL2 is not implemented.
  — EL2 is either not implemented or not enabled in the current Security state.
  — The PE is executing at EL2.
  — FEAT_VHE is implemented, EL2 is using AArch64, EL2 is implemented and enabled in the current Security state, and HCR_EL2.{E2H, TGE} == {1, 1}.

Note
• For all Context breakpoints, DBGBCR<n>.BAS is RES1 and is ignored.
• For Linked Context breakpoints, DBGBCR<n>.{LBN, SSC, HMC, PMC} are RES0 and are ignored.

G2.9.6 Using breakpoints

This section contains the following:
• Using an Address Mismatch breakpoint to single-step an instruction.
• ITD control effects on address breakpoints on the first instruction in an IT block on page G2-9073.
• Breakpoint usage constraints on page G2-9074.

Using an Address Mismatch breakpoint to single-step an instruction

In execution conditions that an Address Mismatch breakpoint matches, defined by DBGBCR<n>.{LBN, SSC, PMC}, the breakpoint generates Breakpoint exceptions for all instructions committed for execution, except the instruction whose address the breakpoint is programmed with. Figure G2-4 shows an example of Address Mismatch breakpoint operation, for an Address Mismatch breakpoint programmed with address 0x1014.

![Figure G2-4 Operation of an Address Mismatch breakpoint](Image)

This means that an Address Mismatch breakpoint can be used to single-step an instruction.

In the example shown in Figure G2-4:

• If the target of a branch is an instruction other than the instruction at address 0x1014, the breakpoint generates a Breakpoint exception when the instruction is committed for execution.
• If the target of a branch is the instruction at address 0x1014, the PE executes the instruction at 0x1014 and the breakpoint does not generate a Breakpoint exception until the instruction at address 0x1018 is committed for execution. The instruction at address 0x1014 is therefore single-stepped.
However, if the instruction at 0x1014 generates a synchronous exception, or if the PE takes an asynchronous exception while the instruction is being stepped, the breakpoint is evaluated again after taking the exception. This means that behavior is as follows:

— If the exception handler executes in execution conditions that the breakpoint matches, the breakpoint generates a Breakpoint exception for the exception vector, because the exception vector is not address 0x1014. This means that software execution steps into the exception.

— If the exception handler executes in execution conditions that the breakpoint does not match, the breakpoint does not generate any Breakpoint exceptions after the PE has taken the exception, until the exception handler completes and executes an exception return instruction. The effect is to step over the exception. Whether the instruction is stepped again depends on whether the target of the exception return instruction is the instruction at 0x1014 or the instruction at 0x1018.

If the instruction at 0x1014 is single-stepped and branches to itself, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception after the PE has executed the branch.

This means that an instruction is only single-stepped if it is the target of a branch instruction and its address matches the address the breakpoint is programmed for. In the example shown in Figure G2-4 on page G2-9072, this is 0x1014.

Usually this branch instruction is an exception return instruction that changes PE mode, branching from a PE mode in which the breakpoint does not generate a Breakpoint exception. A branch instruction that does not change PE mode would itself generate a Breakpoint exception. However, it might be a branch-to-self instruction as described above.

Because Address Mismatch breakpoints can single-step instructions, the behavior of an address mismatch Breakpoint exception is similar to the behavior of an AArch64 Software Step exception.

Note

- The example shown in Figure G2-4 on page G2-9072 assumes an A32 instruction. The same behavior applies for both 32-bit and 16-bit T32 instructions.

- Software Step exceptions are the highest priority synchronous exception. Breakpoint exceptions are lower priority. See Prioritization of Synchronous exceptions taken to AArch64 state on page D1-4650.

ITD control effects on address breakpoints on the first instruction in an IT block

In an implementation that supports the ITD control, if the value of the ITD field that applies to the current Exception level is 1, all of the following are true:

- An IT instruction can only be used to apply to one 16-bit T32 instruction.

- Only certain combinations of an IT instruction and second single 16-bit T32 instruction are permitted.

- For a permitted combination, it is IMPLEMENTATION DEFINED whether the implementation treats the combination as:
  — A pair of 16-bit instructions.
  — One 32-bit instruction.

If the implementation treats the combination as one 32-bit instruction, then as described in Other usage constraints for Address breakpoints on page G2-9076, an Address breakpoint might not generate a Breakpoint exception for an address match only on the second halfword of the instruction.

For this reason, if the ITD bit associated with the current Exception level is 1, Arm recommends that a debugger that wants to program a breakpoint to match on the second T32 instruction programs it to match on the IT instruction instead.

However, if returning from an exception whose preferred return address is the address of the second T32 instruction, then because the debugger is aware that the implementation has treated the combination as a pair of 16-bit instructions, the debugger is permitted to program the breakpoint to match on the second T32 instruction.
The ITD control fields are:

**HSCTLR.ITD** Applies to execution at EL2 when EL2 is using AArch32.

**SCTLR.ITD** Applies to execution at EL0 or EL1 when EL1 is using AArch32.

**SCTLR_EL1.ITD** Applies to execution at EL0 using AArch32 when EL1 is using AArch64.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0, and therefore the information in this section does not apply to such an implementation.

--- **Note** ---

Programming the breakpoint to match on the second T32 instruction might be necessary when using an Address Mismatch breakpoint for single stepping.

### Breakpoint usage constraints

See the following sections:

- **Reserved DBGBCR<n>.BT values**.
- **Reserved DBGBCR<n>.{SSC, HMC, PMC} values** on page G2-9075.
- **Reserved DBGBCR<n>.BAS values** on page G2-9075.
- **Reserved DBGBCR<n>.LBN values** on page G2-9076.
- **Other usage constraints for Address breakpoints** on page G2-9076.
- **Other usage constraints for Context breakpoints** on page G2-9077.

**Reserved DBGBCR<n>.BT values**

Table G2-12 on page G2-9074 shows when particular DBGBCR<n>.BT values are reserved.

<table>
<thead>
<tr>
<th>BT value</th>
<th>Breakpoint type</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b001x</td>
<td>Context ID Match</td>
<td>If the breakpoint is not context-aware</td>
</tr>
<tr>
<td>0b010x</td>
<td>Address Mismatch</td>
<td>If EDSCR.HDE is 1 and halting is allowed</td>
</tr>
<tr>
<td>0b011x</td>
<td>CONTEXTIDR_EL1 Match</td>
<td>If FEAT_VHE is not implemented, or the breakpoint is not context-aware</td>
</tr>
<tr>
<td>0b100x</td>
<td>VMID Match</td>
<td>If EL2 is not implemented, or the breakpoint is not context-aware</td>
</tr>
<tr>
<td>0b101x</td>
<td>Context ID and VMID Match</td>
<td></td>
</tr>
<tr>
<td>0b110x</td>
<td>CONTEXTIDR_EL2 Match</td>
<td>If FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, or if the breakpoint is not context-aware</td>
</tr>
<tr>
<td>0b111x</td>
<td>Full Context ID Match</td>
<td></td>
</tr>
</tbody>
</table>

--- **Note** ---

For these BT values, breakpoints are not generated if EL2 is using AArch32.

If a breakpoint is programmed with one of these reserved BT values:

- The breakpoint must behave as if it is either:
  - Disabled.
  - Programmed with a BT value that is not reserved, other than for a direct or external read of DBGBCR<n>.
- For a direct or external read of DBGBCR<n>, if the reserved BT value:
  - Has no function for any execution conditions, the value read back is **UNKNOWN**.
Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the BT value so that the breakpoint functions for the other execution conditions.

The behavior of breakpoints with reserved BT values might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

Reserved DBGBCR<\textless;n\textgreater;.\{SSC, HMC, PMC\} values

Table G2-13 on page G2-9075 shows when particular combinations of DBGBCR<\textless;n\textgreater;.\{SSC, HMC, PMC\} are reserved.

<table>
<thead>
<tr>
<th>HMC, SSC, and PMC combination</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>All combinations with SSC set to 0b01 or 0b10.</td>
<td>When EL3 is not implemented and EL2 is implemented and EL2 is implemented.</td>
</tr>
<tr>
<td>Any combination where HMC or SSC is nonzero.</td>
<td>When both of EL2 and EL3 are not implemented.</td>
</tr>
<tr>
<td>The combinations with SSC set to 0b11 and the combination with HMC set to 1, SSC set to 0b01 and PMC set to 0b00.</td>
<td>When EL2 is not implemented and EL3 is implemented.</td>
</tr>
<tr>
<td>Any combination where HMC or SSC is nonzero.</td>
<td>When RME is not implemented.</td>
</tr>
<tr>
<td>The combination with HMC set to 1, SSC set to 0b01 and PMC set to 0b00, and the combinations with SSC set to 1, SSC set to 0b11 and PMC set to 0b00.</td>
<td>When both of EL2 and EL3 are implemented, but Secure EL2 is not implemented.</td>
</tr>
<tr>
<td>Combinations not included in Table G2-10 on page G2-9064.</td>
<td>Always.</td>
</tr>
</tbody>
</table>

For all breakpoints except Linked Context breakpoints, if a breakpoint is programmed with one of these reserved combinations:

- If the reserved combination has a function for other execution conditions:
  - The breakpoint must behave as if it is disabled.
  - A direct or external read of DBGBCR<\textless;n\textgreater;.\{SSC, HMC, PMC\} returns the values written. This means that software can save and restore the combination so that the breakpoint can function for the other execution conditions.

- If the reserved combination does not have a function for other execution conditions:
  - It must behave either as if it is programmed with a combination that is not reserved or as if it is disabled.
  - A direct or external read of DBGBCR<\textless;n\textgreater;.\{SSC, HMC, PMC\} returns UNKNOWN values.

If the breakpoint is a Linked Context breakpoint, then:

- The values of HMC, SSC, and PMC are ignored.
- A direct or external read of DBGBCR<\textless;n\textgreater;.\{SSC, HMC, PMC\} returns UNKNOWN values

The behavior of breakpoints with reserved combinations of HMC, SSC, and PMC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

Reserved DBGBCR<\textless;n\textgreater;.BAS values

For all Context breakpoints

DBGBCR<\textless;n\textgreater;.BAS is RES1 and is ignored.
For all Address breakpoints

The supported values of the BAS field for the Address Match and Address Mismatch breakpoints are shown in *Specifying the halfword-aligned address that an Address breakpoint matches on* on page G2-9066.

If a breakpoint is programmed with a reserved BAS value:

- The breakpoint must behave as if it is either:
  - Disabled.
  - Programmed with a BAS value that is not reserved, other than for a direct or external read of DBGBCR<\textit{n}>.
- A direct or external read of DBGBCR<\textit{n}>.BAS returns an UNKNOWN value.

Software must not rely on these properties as the behavior of reserved values might change in a future revision of the architecture.

**Reserved DBGBCR<\textit{n}>.LBN values**

For all Context breakpoints

DBGBCR<\textit{n}>.LBN reads UNKNOWN and its value is ignored.

For Linked Address breakpoints

A Linked Address breakpoint must link to a context-aware breakpoint. For a Linked Address breakpoint, any DBGBCR<\textit{n}>.LBN value that is not for a context-aware breakpoint is reserved.

If a Linked Address breakpoint links to a breakpoint that is not implemented, or that is not context-aware, then reads of DBGBCR<\textit{n}>.LBN return an unknown value and the behavior is CONSTRAINED UNPREDICTABLE. The Linked Address breakpoint behaves as if it is either:

- Disabled.
- Linked to an UNKNOWN context-aware breakpoint.

If a Linked Address breakpoint that links to a breakpoint that is implemented and that is context-aware, but that is either not enabled or not programmed as a Linked Context breakpoint, it behaves as if it is disabled.

For Unlinked Address breakpoints

DBGBCR<\textit{n}>.LBN reads UNKNOWN and its value is ignored.

**Other usage constraints for Address breakpoints**

For all Address breakpoints

- DBGBVR<\textit{n}>[1:0] are RES0 and are ignored.
- The DBGBXVR<\textit{n}> is ignored.

For Address Match breakpoints

- For 32-bit instructions, if a breakpoint matches on the address of the second halfword but not the address of the first halfword, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception.
- If DBGBCR<\textit{n}>.BAS is 0b1111, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception for a T32 instruction starting at address ((DBGBVR<\textit{n}>[31:2]:00) + 2). For T32 instructions, Arm recommends that the debugger programs the BAS field with either 0b0011 or 0b1100.

For Address Mismatch breakpoints

The constraints are the same as those described in *For Address Match breakpoints*, except that if two Address Mismatch breakpoints are programmed to match in the same Exception level and Security state, it is CONSTRAINED UNPREDICTABLE whether or not the instruction is stepped or a Breakpoint debug even is generated.
Other usage constraints for Context breakpoints

For all Context breakpoints

Any bits of DBGBVR<n> and DBGBXVR<n> that are not used to specify Context ID or VMID are RES0 and are ignored.

Note

This means that for Context ID Match breakpoints, the DBGBXVR<n> is RES0 and is ignored, and for VMID Match breakpoints, the DBGBVR<n> is RES0 and is ignored.

For Linked Context breakpoints

If no Linked Address breakpoints or Linked Watchpoints link to a Linked Context breakpoint, the Linked Context breakpoint does not generate any Breakpoint exceptions.

G2.9.7 Exception syndrome information and preferred return address for a Breakpoint exception

See the following:

• Exception syndrome information for a Breakpoint exception.
• Preferred return address for a Breakpoint exception on page G2-9078.

Note

Usually, the term exception syndrome is used only for exceptions taken to Hyp mode, or to AArch64 state. The referenced section uses the term more generally, to include exception information reported in the IFSR.

Exception syndrome information for a Breakpoint exception

The PE takes a Breakpoint exception as either:

• A Prefetch Abort exception if it is taken to PL1. In this case, it is taken to Abort mode.
• A Hyp trap exception, if it is taken to PL2 because HCR.TGE or HDCR.TDE is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

Abort mode

The PE sets all of the following:

• DBGDSCRext.MOE to 0b0001, to indicate a Breakpoint exception.
• IFSR_FS to the code for a debug exception, 0b00010.
• The IFAR with an UNKNOWN value.

Hyp mode

The PE does all of the following:

• Records information about the exception in the Hypervisor Syndrome Register, HSR. See Table G2-14 on page G2-9078.
• Sets DBGDSCRext.MOE to 0b0001, to indicate a Breakpoint exception.
Sets the HIFAR to an unknown value.

### Preferred return address for a Breakpoint exception

The preferred return address of a Breakpoint exception is the address of the instruction that was not executed because the PE took the Breakpoint exception instead. This means that the preferred return address is the address of the instruction that caused the exception.

#### Pseudocode description of Breakpoint exceptions taken from AArch32 state

AArch32.BreakpointValueMatch() returns a pair of results:
- A result for Address Match and Context breakpoints.
- A result for Address Mismatch breakpoints.

AArch32.StateMatch() tests the values in DBGBCR<n>.{SSC, HMC, PMC} and, if the breakpoint links to a Linked Context breakpoint, also tests the Linked Context breakpoint.

AArch32.BreakpointMatch() tests a committed instruction against all breakpoints.

AArch32.CheckBreakpoint() generates a FaultRecord. A Breakpoint exception is taken if all of the following are true:
- DBGDSCRext.MDBGen is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See Enabling debug exceptions from the current Privilege level and Security state on page G2-9045.
- All of the conditions required for Breakpoint exception generation are met. See About Breakpoint exceptions on page G2-9054.

Note: AArch32.CheckBreakpoint() might halt the PE and cause it to enter Debug state. External debug uses Debug state.

The AArch32.Abort() function processes the FaultRecord object returned by AArch32.CheckBreakpoint(), as described in Abort exceptions on page G4-9144. When a Breakpoint exception is taken to AArch32 state, the AArch32.Abort() function generates a Prefetch Abort exception.
G2.10 Watchpoint exceptions

This section describes Watchpoint exceptions in stage 1 of an AArch32 translation regime.

The PE is using an AArch32 translation regime when it is executing either:
• At EL1 or higher in an Exception level that is using AArch32.
• At EL0 using AArch32 when EL1 is using AArch32.

This section contains the following subsections:
• About Watchpoint exceptions.
• Watchpoint types and linking of watchpoints on page G2-9080.
• Execution conditions for which a watchpoint generates Watchpoint exceptions on page G2-9081.
• Watchpoint data address comparisons on page G2-9083.
• Determining the memory location that caused a Watchpoint exception on page G2-9086.
• Watchpoint behavior on other instructions on page G2-9087.
• Usage constraints on page G2-9088.
• Exception syndrome information and preferred return address on page G2-9090.
• Pseudocode description of Watchpoint exceptions taken from AArch32 state on page G2-9091.

G2.10.1 About Watchpoint exceptions

A watchpoint is an event that results from the execution of an instruction, based on a data address. Watchpoints are also known as data breakpoints.

A watchpoint operates as follows:
1. A debugger programs the watchpoint with a data address, or a data address range.
2. The watchpoint generates a Watchpoint debug event on an access to the address, or any address in the address range.

A watchpoint never generates a Watchpoint debug event on an instruction fetch.

An implementation can include between 2-16 watchpoints. In an implementation, DBGDIR.WRPs shows how many are implemented.

To use an implemented watchpoint, a debugger programs the following registers for the watchpoint:
• The Watchpoint Control Register, DBGWCR<n>. This holds control information for the watchpoint, for example an enable control.
• The Watchpoint Value Register, DBGWVR<n>. This holds the data virtual address used for watchpoint matching.

The registers are numbered, so that:
• DBGWCR1 and DBGWVR1 are for watchpoint number one.
• DBGWCR2 and DBGWVR2 are for watchpoint number two.
• …
• …
• DBGWCRn and DBGWVRn are for watchpoint number n.

A watchpoint can:
• Be programmed to generate Watchpoint debug events on read accesses only, on write accesses only, or on both types of access.
• Link to a Linked Context breakpoint, so that a Watchpoint debug event is only generated if the PE is in a particular context when the address match occurs.
A single watchpoint can be programmed to match on one or more address bytes. A watchpoint generates a Watchpoint debug event on an access to any byte that it is watching. The number of bytes a watchpoint is watching is either:

- One to eight bytes, provided that these bytes are contiguous and that they are all in the same naturally-aligned doubleword. A debugger uses the *Byte Address Select* field, \texttt{DBGWCR<n>.BAS}, to select the bytes. See *Programming a watchpoint with eight bytes or fewer on page G2-9084*.

- Eight bytes to 2GB, provided that both of the following are true:
  - The number of bytes is a power-of-two.
  - The range starts at an address that is aligned to the range size.

A debugger uses the *MASK* field, \texttt{DBGWCR<n>.MASK}, to program a watchpoint with eight bytes to 2GB. See *Programming a watchpoint with eight or more bytes on page G2-9085*.

A debugger must use either the BAS field or the MASK field. If it uses both, whether the watchpoint generates Watchpoint exceptions is CONSTRAINED UNPREDICTABLE. See *Programming dependencies of the BAS and MASK fields on page G2-9089*.

For each memory access, all of the watchpoints are tested. When a watchpoint is tested, it generates a Watchpoint debug event if all of the following are true:

- The watchpoint is enabled. That is, the watchpoint enable control for it, \texttt{DBGWCR<n>.E}, is 1.
- The conditions specified in the \texttt{DBGWCR<n>} are met.
- The comparison with the address held in the \texttt{DBGWVR<n>} is successful.
- If the watchpoint links to a Linked Context breakpoint, the comparison or comparisons made by the Linked Context breakpoint are successful. See * on page G2-9057 shows this. See also Breakpoint context comparisons on page G2-9071*.
- The instruction that initiates the memory access is committed for execution.
- The instruction that initiates the memory access passes its Condition code check.

If halting is allowed and \texttt{EDSCR.HDE} is 1, Watchpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are:
- Enabled, Watchpoint debug events generate Watchpoint exceptions.
- Disabled, Watchpoint debug events are ignored.

_____ Note  

The remainder of this Watchpoint Exceptions section, including all subsections, describes watchpoints as generating Watchpoint exceptions. However, the behavior described also applies if watchpoints are causing entry to Debug state.

The debug exception enable controls on page G2-9042 describes the enable controls for Watchpoint debug events.

### G2.10.2 Watchpoint types and linking of watchpoints

When a debugger programs a watchpoint, it must program that watchpoint so that it is either:

- Used in isolation. In this case, the watchpoint is called an *Unlinked watchpoint*.
- Enabled for linking to a Linked Context breakpoint. In this case, the watchpoint is called a *Linked watchpoint*.

When a Linked watchpoint links to a Linked Context breakpoint, the Linked watchpoint only generates a Watchpoint exception if the PE is in a particular context when the data address match occurs. For example, a debugger might:

1. Program watchpoint number one with a data address.
2. Program breakpoint number five to be a *Linked VMID Match breakpoin*:t.
3. Link the watchpoint and the breakpoint together. A Watchpoint exception is only generated if both the data address matches and the VMID matches.

The Watchpoint Type field for a watchpoint, DBGWCR<n>.WT, controls whether the watchpoint is enabled for linking. If DBGWCR<n>.WT is 1, the watchpoint is enabled for linking.

**Rules for linking watchpoints**

The rules for watchpoint linking are as follows:

- Only Linked watchpoints can be linked.
- A Linked watchpoint can link to any type of Linked Context breakpoint. The Linked Breakpoint Number field, DBGWCR<n>.LBN, for the Linked watchpoint specifies the particular Linked Context breakpoint that the Linked watchpoint links to, and:
  - DBGWCR<n>.WT.{SSC, HMC, PAC} for the Linked watchpoint define the execution conditions that the watchpoint generates Watchpoint exceptions for. See *Execution conditions for which a watchpoint generates Watchpoint exceptions*.
  - DBGBCR<n>.{SSC, HMC, PMC} for the Linked Context breakpoint are ignored.
- A Linked watchpoint cannot link to another watchpoint. The LBN field can therefore only specify a breakpoint.
- If a Linked watchpoint links to a breakpoint that is not context-aware, the behavior of the Linked watchpoint is CONSTRAINED UNPREDICTABLE. See *Usage constraints* on page G2-9088.
- If a Linked watchpoint links to an Unlinked Context breakpoint, the Linked watchpoint never generates any Watchpoint exceptions.
- Multiple Linked watchpoints can link to a single Linked Context breakpoint.

**Note**

Multiple Address breakpoints can also link to a single Linked Context breakpoint. *Breakpoint exceptions on page G2-9054* describes breakpoints.

Figure G2-1 on page G2-9057 shows an example of permitted watchpoint linking.

**G2.10.3 Execution conditions for which a watchpoint generates Watchpoint exceptions**

Each watchpoint can be programmed so that it only generates Watchpoint exceptions for certain execution conditions. For example, a watchpoint might be programmed to generate Watchpoint exceptions only when the PE is executing at EL2.

DBGWCR<n>.{SSC, HMC, PAC} define the execution conditions a watchpoint generates Watchpoint exceptions for, as follows:

**Security State Control, SSC**

Controls whether the watchpoint generates Watchpoint exceptions only in Secure state, only in Non-secure state, or in both Security states.

**Note**

This is determined by the Security state of the PE, not from the NS attribute returned by the translation of the virtual address on which the watchpoint is set.

**Higher Mode Control, HMC, and Privileged Access Control, PAC**

HMC and PAC together control which Privilege level the watchpoint generates Watchpoint exceptions in.

The PAC control relates to the privilege of the memory access, not to the Exception level or Privilege level at which the access was made.
This means that, if the PE executes a Load unprivileged or Store unprivileged instruction at PL1, the resulting data access triggers a watchpoint only if both:

- PAC is programmed to a value that generates watchpoints on PL0 accesses.
- All other conditions for generating the watchpoint are met.

Example A32/T32 Load unprivileged and Store unprivileged instructions are LDRT and STRT.

Table G2-15 shows the valid combinations of HMC, SSC, and PAC, and for each combination shows which Privilege levels watchpoints generate Watchpoint exceptions in.

In the table:

- **Y or -** Means that a watchpoint programmed with the values of HMC, SSC, and PAC shown in that row:
  - Y Can generate Watchpoint exceptions at that Privilege level.
  - Cannot generate Watchpoint exceptions at that Privilege level.
- **Res** Means that the combination of HMC, SSC, and PAC is reserved. See Reserved DBGWCR<n>.{SSC, HMC, PAC} values on page G2-9088.

### Table G2-15 Summary of watchpoint HMC, SSC, and PAC encodings

<table>
<thead>
<tr>
<th>HMC</th>
<th>SSC</th>
<th>PAC</th>
<th>Security state the watchpoint is programmed to match in</th>
<th>PL2&lt;sup&gt;a&lt;/sup&gt;</th>
<th>PL1</th>
<th>PL0</th>
<th>Implementation</th>
<th>No EL3</th>
<th>No EL2 and no EL3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>01</td>
<td>Both</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>10</td>
<td></td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>11</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01</td>
<td>Non-secure</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Res</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>10</td>
<td></td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>Res</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>Res</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01</td>
<td>Secure</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Res</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
<td></td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>Res</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>Res</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>01</td>
<td>Secure</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>11</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>Both</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>11</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>00</td>
<td>Non-secure</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>01</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>Res</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Res</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table G2-15 Summary of watchpoint HMC, SSC, and PAC encodings (continued)

<table>
<thead>
<tr>
<th>HMC</th>
<th>SSC</th>
<th>PAC</th>
<th>Security state the watchpoint is programmed to match in</th>
<th>PL2&lt;sup&gt;a&lt;/sup&gt;</th>
<th>PL1</th>
<th>PL0</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>01</td>
<td>Secure</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>Res</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td></td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>Res</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>00</td>
<td>Both</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>Res if no EL2&lt;sup&gt;b&lt;/sup&gt;</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>01</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>Res</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>11</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Res</td>
</tr>
</tbody>
</table>

<sup>a</sup> Debug exceptions are not generated at PL2 using AArch32. This means that these combinations of HMC, SSC, and PAC are only relevant if watchpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSC, and PAC that generate Watchpoint exceptions at PL2 using AArch32.

<sup>b</sup> This encoding is only reserved when EL2 is not implemented, regardless of whether EL3 is implemented.

All combinations of HMC, SSC, and PAC that this table does not show are reserved. See Reserved \( \text{DBGWCR}<n> \cdot \{\text{SSC, HMC, PAC}\} \) values on page G2-9088.

**G2.10.4 Watchpoint data address comparisons**

An address comparison is successful if bits [31:2] of the current data virtual address are equal to \( \text{DBGWVR}<n>[31:2] \), taking into account all of the following:

- The size of the access. See *Size of the data access*.
- The bytes selected by \( \text{DBGWVR}<n> \cdot \text{BAS} \). See *Programming a watchpoint with eight bytes or fewer* on page G2-9084.
- Any address ranges indicated by \( \text{DBGWVR}<n> \cdot \text{MASK} \). See *Programming a watchpoint with eight or more bytes* on page G2-9085.

---

**Note**

\( \text{DBGWVR}<n>[1:0] \) are RES0 and are ignored.

---

**Size of the data access**

Because watchpoints can be programmed to generate Watchpoint exceptions on individual bytes, the size of each access must be taken into account. See *Example G2-1*.

---

**Example G2-1**

1. A debugger programs a watchpoint to generate Watchpoint exceptions only when the byte at address 0x1009 is accessed.

2. The PE accesses the unaligned doubleword starting at address 0x1003.

In this scenario, the watchpoint must generate a Watchpoint exception.
The size of data accesses initiated by DCIMVAC instructions is an IMPLEMENTATION DEFINED size that is both:
  • From the inclusive range between:
    — The size that CTR.DminLine defines.
    — 2KB.
  • A power-of-two.

The lowest address accessed by a DCIMVAC instruction is the address supplied to the instruction, rounded down to the nearest multiple of the access size initiated by that instruction.

The highest address accessed is (size - 1) bytes above the lowest address accessed.

See also, Watchpoint behavior on accesses by DCIMVAC instructions on page G2-9088.

### Programming a watchpoint with eight bytes or fewer

The Byte Address Select field, DBGWCR<\text{n}>.BAS, selects which bytes in the doubleword starting at the address contained in the DBGWVR<\text{n}> the watchpoint generates Watchpoint exceptions for.

If the address programmed into the DBGWVR<\text{n}> is:

  • Doubleword-aligned:
    — All eight bits of DBGWCR<\text{n}>.BAS are used, and the descriptions given in Table G2-16 apply.
  • Word-aligned but not doubleword-aligned:
    — Only DBGWCR<\text{n}>.BAS[3:0] are used, and the descriptions given in Table G2-17 apply. In this case, DBGWCR<\text{n}>.BAS[7:4] are RES0.

<table>
<thead>
<tr>
<th>BAS value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00000000</td>
<td>Watchpoint never generates a Watchpoint exception</td>
</tr>
<tr>
<td>BAS[0] == 1</td>
<td>Generates a Watchpoint exception if byte at address DBGWVR&lt;\text{n}&gt;[31:3]:000 is accessed</td>
</tr>
<tr>
<td>BAS[1] == 1</td>
<td>Generates a Watchpoint exception if byte at address DBGWVR&lt;\text{n}&gt;[31:3]:001 is accessed</td>
</tr>
<tr>
<td>BAS[2] == 1</td>
<td>Generates a Watchpoint exception if byte at address DBGWVR&lt;\text{n}&gt;[31:3]:010 is accessed</td>
</tr>
<tr>
<td>BAS[3] == 1</td>
<td>Generates a Watchpoint exception if byte at address DBGWVR&lt;\text{n}&gt;[31:3]:011 is accessed</td>
</tr>
<tr>
<td>BAS[4] == 1</td>
<td>Generates a Watchpoint exception if byte at address DBGWVR&lt;\text{n}&gt;[31:3]:100 is accessed</td>
</tr>
<tr>
<td>BAS[5] == 1</td>
<td>Generates a Watchpoint exception if byte at address DBGWVR&lt;\text{n}&gt;[31:3]:101 is accessed</td>
</tr>
<tr>
<td>BAS[6] == 1</td>
<td>Generates a Watchpoint exception if byte at address DBGWVR&lt;\text{n}&gt;[31:3]:110 is accessed</td>
</tr>
<tr>
<td>BAS[7] == 1</td>
<td>Generates a Watchpoint exception if byte at address DBGWVR&lt;\text{n}&gt;[31:3]:111 is accessed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BAS value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00000000</td>
<td>Watchpoint never generates a Watchpoint exception</td>
</tr>
<tr>
<td>BAS[0] == 1</td>
<td>Generates a Watchpoint exception if byte at address DBGWVR&lt;\text{n}&gt;[31:2]:00 is accessed</td>
</tr>
</tbody>
</table>
If the BAS field is programmed with more than one byte, the bytes that it is programmed with must be contiguous. For watchpoint behavior when its BAS field is programmed with non-contiguous bytes, see Other usage constraints on page G2-9090.

When programming the BAS field with anything other than 0b11111111, a debugger must also program DBGWCR.<n>.MASK to be 0b00000. See Programming dependencies of the BAS and MASK fields on page G2-9089.

A watchpoint generates a Watchpoint exception whenever a watched byte is accessed, even if:

- The access size is smaller or larger than the address region being watched.
- The access is misaligned, and the base address of the access is not in the doubleword or word of memory addressed by the DBGWVR.<n>[31:3]. See Example G2-1 on page G2-9083.

The following are some example configurations of the BAS field:

- To program a watchpoint to generate a Watchpoint exception on the byte at address 0x1003, program:
  - DBGWVR.<n> with 0x1000.
  - DBGWCR.<n>_EL1.BAS to be 0b00001000.

- To program a watchpoint to generate a Watchpoint exception on the bytes at addresses 0x2003, 0x2004 and 0x2005, program:
  - DBGWVR.<n> with 0x2000.
  - DBGWCR.<n>_EL1.BAS to be 0b00111000.

- If the address programmed into the DBGWVR.<n> is doubleword-aligned:
  - To generate a Watchpoint exception when any byte in the word starting at the doubleword-aligned address is accessed, program DBGWCR.<n>.BAS to be 0b00001111.
  - To generate a Watchpoint exception when any byte in the word starting at address DBGWVR.<n>[31:3]:100 is accessed, program DBGWCR.<n>.BAS to be 0b11110000.

Note

Arm deprecates programming a DBGWVR.<n> with an address that is not doubleword-aligned.

Programming a watchpoint with eight or more bytes

A debugger can use the MASK field, DBGWCR.<n>.MASK, to program a single watchpoint with a data address range. The data address range must meet all of the following criteria:

- It is a size that is both:
  - A power-of-two.
  - A minimum of eight bytes.
  - A maximum of 2GB.

- It starts at an address that is aligned to the size.
The MASK field specifies the number of least significant data address bits that must be masked. Up to 31 least significant bits can be masked:

<table>
<thead>
<tr>
<th>MASK</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00000</td>
<td>No bits are masked.</td>
</tr>
<tr>
<td>0b00001</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0b00010</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0b00011</td>
<td>Three least significant bits are masked.</td>
</tr>
<tr>
<td>0b00100</td>
<td>Four least significant bits are masked.</td>
</tr>
<tr>
<td>0b00101</td>
<td>Five least significant bits are masked.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0b11111</td>
<td>31 least significant bits are masked.</td>
</tr>
</tbody>
</table>

If \( n \) least significant address bits are masked, the watchpoint generates a Watchpoint exception on all of the following:

- Address \( \text{DBGWVR}<n>[31:n]:000\ldots \)
- Address \( \text{DBGWVR}<n>[31:n]:111\ldots \)
- Any address between these two addresses.

For example, if the four least significant address bits are masked, Watchpoint exceptions are generated for all addresses between \( \text{DBGWVR}<n>[31:4]:0000 \) and \( \text{DBGWVR}<n>[31:4]:1111 \), including these addresses.

--- Note ---
- The most significant bit cannot be masked. This means that the full address cannot be masked.
- For watchpoint behavior when its MASK field is programmed with a reserved value, see *Reserved DBGWCR<\( n \).MASK values* on page G2-9090.

When masking address bits, a debugger must both:

- Program \( \text{DBGWCR}<n>.\text{BAS} \) to be 0b11111111. See *Programming dependencies of the BAS and MASK fields* on page G2-9089.
- In the \( \text{DBGWVR}<n> \), set the masked address bits to 0. For watchpoint behavior when any of the masked address bits are not 0, see *Other usage constraints* on page G2-9090.

## G2.10.5 Determining the memory location that caused a Watchpoint exception

On a Watchpoint exception, the PE records an address in a *Fault Address Register* that the debugger can use to determine the memory location that triggered the watchpoint.

The Fault Address Register (FAR) used is either:

- DFAR, if the exception is taken to PL1.
- HDFAR, if the exception is taken to PL2.

In cases where one instruction triggers multiple watchpoints, only one address is recorded.

On entering Debug state on a Watchpoint debug event, the PE records the address in the EDWAR.

--- Note ---
If Debug state was entered from AArch32 state, then EDWAR[63:32] is unknown and must be ignored by the debugger.

For more information, see the subsections that follow. These are:

- *Address recorded for Watchpoint exceptions generated by instructions other than data cache maintenance instructions* on page G2-9087.
- *Address recorded for Watchpoint exceptions generated by data cache maintenance instructions* on page G2-9087.
Address recorded for Watchpoint exceptions generated by instructions other than data cache maintenance instructions

The address recorded must be both:

- From the inclusive range between:
  - The lowest address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint.
  - The highest watchpointed address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint. A watchpointed address is an address that the watchpoint is watching.

- Within a naturally-aligned block of memory that is all of the following:
  - A power-of-two size.
  - No larger than the DC ZVA block size.
  - Contains a watchpointed address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint.

The size of the block is IMPLEMENTATION DEFINED. There is no architectural means of discovering the size.

Example G2-2 Address recorded for a watchpoint programmed on 0x8019

A debugger programs a watchpoint to generate a Watchpoint exception on any access to the byte 0x8019.

An A32 load multiple instruction then loads nine registers starting from address 0x8004 upwards. This triggers the watchpoint.

If the DC ZVA block size is:
  - 32 bytes, the address that the PE records must be between 0x8004 and 0x8019 inclusive.
  - 16 bytes, the address that the PE records must be between 0x8010 and 0x8019 inclusive.

Address recorded for Watchpoint exceptions generated by data cache maintenance instructions

The address recorded is the address passed to the instruction. This means that the address recorded might be higher than the address of the location that triggered the watchpoint.

G2.10.6 Watchpoint behavior on other instructions

Under normal operating conditions, the following do not generate Watchpoint exceptions:

- Instruction cache maintenance instructions.
- Address translation instructions.
- TLB maintenance instructions.
- Preload instructions.
- All data cache maintenance instructions except DCIMVAC.

However, the debug architecture allows for IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, to enable watchpoints on an implementation defined subset of these instructions. Whether a watchpoint treats the instruction as a load or a store, and the access size of instruction cache maintenance, address translation, and TLB maintenance instructions are implementation defined.

The access size of the IMPLEMENTATION DEFINED instruction cache maintenance, address translation, and TLB maintenance instructions that generate Watchpoint exceptions are IMPLEMENTATION DEFINED.

See also:

- Watchpoint behavior on accesses by Store-Exclusive instructions on page G2-9088.
- Watchpoint behavior on accesses by DCIMVAC instructions on page G2-9088.
Watchpoint behavior on accesses by Store-Exclusive instructions

If a watchpoint matches on a data access caused by a Store-Exclusive instruction, then:

- If the store fails because an Exclusives monitor does not permit it, it is IMPLEMENTATION DEFINED whether the watchpoint generates a Watchpoint exception.
- Otherwise, the watchpoint generates a Watchpoint exception.

Watchpoint behavior on accesses by DCIMVAC instructions

It is IMPLEMENTATION DEFINED whether DCIMVAC operations can generate Watchpoint exceptions. If they can, they are treated as data stores. This means that for a watchpoint to match on an access caused by a DCIMVAC instruction, the debugger must program DBGWCR<n>.LSC to be one of the following:

- Match on data stores only.
- Match on data stores and data loads.

**Note**

For the size of data accesses performed by DCIMVAC instructions, see Watchpoint data address comparisons on page G2-9083. The size of all data accesses must be considered because watchpoints can be programmed to match on individual bytes.

### G2.10.7 Usage constraints

See the following:

- Reserved DBGWCR<n>.{SSC, HMC, PAC} values.
- Reserved DBGWCR<n>.LBN values on page G2-9089.
- Programming dependencies of the BAS and MASK fields on page G2-9089.
- Reserved DBGWCR<n>.BAS values on page G2-9089.
- Reserved DBGWCR<n>.MASK values on page G2-9090.
- Other usage constraints on page G2-9090.

Reserved DBGWCR<n>.{SSC, HMC, PAC} values

Table G2-18 on page G2-9088 shows when particular combinations of DBGWCR<n>.{SSC, HMC, PAC} are reserved.

<table>
<thead>
<tr>
<th>HMC, SSC, and PAC combination</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>All combinations with SSC set to 0b01 or 0b10.</td>
<td>When EL3 is not implemented and EL2 is implemented.</td>
</tr>
<tr>
<td>Any combination where HMC or SSC is nonzero</td>
<td>When both of EL2 and EL3 are not implemented</td>
</tr>
<tr>
<td>The combination with HMC set to 1, SSC set to 0b11, and PAC set to 0b00</td>
<td>When EL2 is not implemented</td>
</tr>
<tr>
<td>The combinations with SSC set to 0b11 and PAC set to 0b01 or 0b11</td>
<td>When Secure EL2 is not implemented</td>
</tr>
<tr>
<td>The combination with HMC set to 1, SSC set to 0b01 and PAC set to 0b00</td>
<td>When Secure EL2 is not implemented</td>
</tr>
<tr>
<td>Combinations not included in Table G2-15 on page G2-9082.</td>
<td>Always</td>
</tr>
</tbody>
</table>

If a watchpoint is programmed with one of these reserved combinations:

- The watchpoint must behave as if it is either:
  - Disabled.
— Programmed with a combination that is not reserved, other than for a direct or external read of \texttt{DBGWCR<n>}.

- For a direct or external read of \texttt{DBGWCR<n>}, if the reserved combination:
  — Has no function for any execution conditions, the value read back for each of SSC, HMC, and PMC is \texttt{UNKNOWN}.
  — Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the combination so that the watchpoint functions for the other execution conditions.

The behavior of watchpoints with reserved combinations of SSC, HMC, and PAC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

**Reserved \texttt{DBGWCR<n>.LBN} values**

**For Linked watchpoints**

A Linked watchpoint must link to a context-aware breakpoint. For a Linked watchpoint, any \texttt{DBGWCR<n>.LBN} value that is not for a context-aware breakpoint is reserved.

If a Linked watchpoint links to a breakpoint that is not implemented, or that is not context-aware, then reads of \texttt{DBGWCR<n>.LBN} return an \texttt{UNKNOWN} value and the behavior is \texttt{CONSTRAINED UNPREDICTABLE}. The Linked watchpoint behaves as if it is either:

- Disabled.
- Linked to an \texttt{UNKNOWN} context-aware breakpoint.

If a Linked watchpoint links to a breakpoint that is implemented and is context-aware, but that is either not enabled or not programmed as a Linked Context breakpoint, it behaves as if it is disabled.

**For Unlinked watchpoints**

For Unlinked watchpoints, \texttt{DBGWCR<n>.LBN} reads \texttt{UNKNOWN} and its value is ignored.

**Programming dependencies of the BAS and MASK fields**

When programming a watchpoint, a debugger must use either:

- The MASK field, to program the watchpoint with an address range that can be eight bytes to 2GB.
- The BAS field, to select which bytes in the doubleword or word starting at the address contained in the \texttt{DBGWVR<n>} the watchpoint must generate Watchpoint exceptions for.

If the debugger uses the:

- MASK field, it must program BAS to be $0b11111111$, so that all bytes in the doubleword or word are selected.
- BAS field, it must program MASK to be $0b000000$, so that the MASK field does not indicate any address ranges.

If an enabled watchpoint has a MASK field that is nonzero and a BAS field that is not set to $0b11111111$, then for each byte in the address range, it is \texttt{CONSTRAINED UNPREDICTABLE} whether or not a Watchpoint exception is generated.

**Reserved \texttt{DBGWCR<n>.BAS} values**

The BAS field must be programmed with a value $\text{Zeros}(8-n-m) : \text{Ones}(n) : \text{Zeros}(m)$, where:

- $n$ is a nonzero positive integer less-than-or-equal-to 8.
- $m$ is a positive integer less-than 8.
- $n+m$ is less-than-or-equal-to 8.

All other values are reserved.


Note

If $x$ is zero, then $\text{Zeros}(x)$ is an empty bitstring.

If $\text{DBGWVR}<n>[2]$ is 1, $\text{DBGWCR}<n>$.BAS[7:4] are RES0 and are ignored.

If a watchpoint is programmed with a reserved BAS value:

• It is constrained unpredictable whether the watchpoint generates a Watchpoint exception for each byte in the doubleword or word of memory addressed by the $\text{DBGWVR}<n>$.

• A direct or external read of $\text{DBGWCR}<n>$.BAS returns an unknown value.

Software must not rely on these properties as the behavior of reserved values might change in a future revision of the architecture.

Reserved $\text{DBGWCR}<n>$.MASK values

If a watchpoint is programmed with a reserved MASK value:

• The watchpoint must behave as if it is either:
  — Disabled.
  — Programmed with an unknown value that is not reserved, that might be $\text{0b0000}$, other than for a direct or external read of $\text{DBGWCR}<n>$.

• A direct or external read of $\text{DBGWCR}<n>$.MASK returns an unknown value.

Other usage constraints

For all watchpoints:

• $\text{DBGWVR}<n>[1:0]$ are RES0 and are ignored.

• If $\text{DBGWCR}<n>$.MASK is nonzero, and any masked bits of $\text{DBGWVR}<n>$ are not 0, it is constrained unpredictable whether the watchpoint generates a Watchpoint exception when the unmasked bits match.

• A watchpoint never generates any Watchpoint exceptions if $\text{DBGWCR}<n>$.LSC is $\text{0b00}$.

G2.10.8 Exception syndrome information and preferred return address

See the following:

• Exception syndrome information.
• Preferred return address on page G2-9091.

Exception syndrome information

The PE takes a Watchpoint exception as either:

• A Data Abort exception, if it is taken to PL1. In this case, it is taken to Abort mode.

• A Hyp trap exception, if it is taken to PL2 because $\text{HCR.TGE}$ or $\text{HDCR.TDE}$ is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

Abort mode

The PE sets all of the following:

• $\text{DBGDSCRext}.\text{MOE}$ to $\text{0b1010}$, to indicate a Watchpoint exception.

• $\text{DFSR}.\text{CM}$ to indicate whether a cache maintenance instruction caused the exception.
- **DFSR.WnR** to indicate whether the exception was generated on a read instruction or a write instruction.
- **DFAR** to an address that the debugger can use to determine the memory location that triggered the watchpoint. See *Determining the memory location that caused a Watchpoint exception on page G2-9086.*

In addition, if using the:
- Short-descriptor format, the PE sets **DFSR.FS** to the code for a debug exception, 0b00010, and **DFSR.Domain** to an **UNKNOWN** value.
- Long-descriptor format, the PE sets **DFSR.STATUS** to the code for a debug exception, 0b100010.

**Hyp mode**

The PE does all of the following:
- Records information about the exception in the **Hypervisor Syndrome Register, HSR**. See Table G2-19 on page G2-9091.
- Sets **DBGDSCRext.MOE** to 0b1001, to indicate a Watchpoint exception.
- Sets the **HDFAR** to an address that the debugger can use to determine the memory location that triggered the watchpoint. See *Determining the memory location that caused a Watchpoint exception on page G2-9086.*

### Table G2-19 Information recorded in the **HSR**

<table>
<thead>
<tr>
<th>HSR field</th>
<th>Information recorded</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exception Class, EC</strong></td>
<td>The PE sets this to the code for a Data Abort exception routed to Hyp mode, 0x24.</td>
</tr>
<tr>
<td><strong>Instruction Length, IL</strong></td>
<td>The PE sets this to 1.</td>
</tr>
</tbody>
</table>
| **Instruction Specific Syndrome, ISS** | **ISV[24]** Instruction Syndrome Valid (ISV). The PE sets this to 0.  
**ISS[23:10]** [RES0.  
**ISS[9]** External Abort type (EA). The PE sets this to 0.  
**ISS[8]** Cache Maintenance (CM). The PE sets this to indicate whether a cache maintenance instruction caused the exception.  
**ISS[7]** [RES0.  
**ISS[6]** Write not Read (WnR). The PE sets this to indicate whether the exception was generated on a read instruction or a write instruction.  
**ISS[5:0]** Data Fault Status Code (DFSC). The PE sets this to the code for a debug exception, 0b100010. |

---

**Note**

For information about how debug exceptions can be routed to PL2, see *Routing debug exceptions on page G2-9043.*

**Preferred return address**

The preferred return address of a Watchpoint exception is the address of the instruction that was not executed because the PE took the Watchpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

---

**G2.10.9 Pseudocode description of Watchpoint exceptions taken from AArch32 state**

`AArch32.WatchpointByteMatch()` tests an individual byte accessed by an operation.
AArch32.StateMatch() tests the values in DBGWCR<n>.{HMC, SSC, PAC}, and if the watchpoint is Linked, also tests the Linked Context breakpoint that the watchpoint links to.

AArch32.WatchpointMatch() tests the value in DBGWVR<n>.

AArch32.CheckWatchpoint() generates a FaultRecord. A Watchpoint exception is taken if all of the following are true:

- DBGDSCRext.MDBGen is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See Enabling debug exceptions from the current Privilege level and Security state on page G2-9045.
- All of the conditions required for Watchpoint exception generation are met. See About Watchpoint exceptions on page G2-9079.

Note

AArch32.CheckWatchpoint might halt the PE and cause it to enter Debug state. External debug uses Debug state.

The AArch32.Abort() function processes the FaultRecord object returned by AArch32.CheckWatchpoint(), as described in Abort exceptions on page G4-9144. If a Watchpoint exception is taken to AArch32 state, the AArch32.Abort() function generates a Data Abort exception.
**G2.11 Vector Catch exceptions**

Arm deprecates the use of vector catch.

This section describes Vector Catch exceptions in stage 1 of an AArch32 translation regime.

The PE is using an AArch32 translation regime when it is executing either:
- At EL1 or higher in an Exception level that is using AArch32.
- At EL0 using AArch32 when EL1 is using AArch32.

--- **Note**

Vector Catch exceptions cannot be generated when the PE is using an AArch64 translation regime.

This section contains the following subsections:
- **About Vector Catch exceptions**
- **Exception vectors that Vector Catch exceptions can be enabled for** on page G2-9095.
- **Generation of Vector Catch exceptions** on page G2-9096.
- **Usage constraints** on page G2-9098.
- **Exception syndrome information and preferred return address for a Vector Catch exception** on page G2-9098.
- **Pseudocode description of Vector Catch exceptions** on page G2-9099.

**G2.11.1 About Vector Catch exceptions**

Whenever the PE takes an exception, execution is forced to an address that is the exception vector for that exception.

Vector catch permits a debugger to trap exceptions based on the exception vector, or based on the exception type associated with the exception vector, as follows:

- If the address-matching form of vector catch is implemented, the debugger can trap exceptions based on the exception vector.
- If the exception-trapping form of vector catch is implemented, the debugger can trap exceptions based on the exception type associated with the exception vector.

The architecture supports only these two forms of vector catch. Only one form can be implemented, and which is implemented is IMPLEMENTATION DEFINED. The DBGDEVID indicates which form is implemented.

Regardless of the form of vector catch implemented, a debugger enables Vector Catch exceptions for exception vectors or types by programming the DBGVCR. This register contains vector catch enable bits. Each of these bits corresponds to a different vector. When a debugger sets a vector catch enable bit to 1, Vector Catch exceptions are enabled for the corresponding exception vector or type.

--- **Note**

EL2 using AArch64 or EL3 using AArch64 can enable Vector Catch exceptions for vectors by programming the DBGVCR32_EL2. The DBGVCR32_EL2 is architecturally mapped to the DBGVCR.

When Vector Catch exceptions are enabled for an exception vector, this is called an enabled vector catch. The set of exception vectors that Vector Catch exceptions are enabled for is called the enabled vector catch set.

If the form of vector catch implemented is the:

**Address-matching form:**

The PE compares the virtual address of each instruction in the program flow with a subset of the enabled vector catch set.

If an address match occurs, a Vector Catch exception is generated when the instruction that caused the match is committed for execution.
Exception-trapping form

Whenever the PE takes an exception, if the vector the exception is taken to is included in a subset of the enabled vector catch set, a Vector Catch exception is generated.

The Vector Catch exception is generated as part of entry to the exception, and must be taken before the PE either executes any instructions or takes any further exceptions.

The addresses that comprise the subset depend on whether EL3 is implemented and, for the:

- Address-matching form, the current Security state.
- Exception-trapping form, the Security state that the exception is handled in.

See Generation of Vector Catch exceptions on page G2-9096.

Table G2-20 on page G2-9094 summarizes the differences between the address-matching and exception-trapping forms.

<table>
<thead>
<tr>
<th>Address-matching</th>
<th>Exception-trapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>An enabled vector catch generates a Vector Catch exception when an instruction that is fetched from the vector is committed for execution. This means that spurious Vector Catch exceptions might occur, where the Vector Catch exception does not result from an exception entry, but is instead caused by a branch to the vector. A branch to the vector might occur, for example, on a return from a nested exception or when simulating an exception entry.</td>
<td>An enabled vector catch generates a Vector Catch exception immediately after the PE takes the exception that is associated with the vector. This means that Vector Catch exceptions always result from exception entry, and not from branches to exception vectors.</td>
</tr>
<tr>
<td>A Vector Catch exception is generated as a result of an instruction fetch. This means that the Vector Catch exception has a priority relative to the other synchronous exceptions that result from an instruction fetch. Prioritization of Synchronous exceptions taken to AArch64 state on page D1-4650 describes this prioritization.</td>
<td>A Vector Catch exception is generated as a result of an exception entry. This means that the Vector Catch exception is part of the exception that caused the Vector Catch exception. Therefore, the Vector Catch exception has no priority associated with it. For this reason, Vector Catch exceptions are outside the scope of the prioritization that Prioritization of Synchronous exceptions taken to AArch64 state on page D1-4650 describes.</td>
</tr>
<tr>
<td>A Vector Catch exception can be preempted by another exception. If this happens, the Vector Catch exception is generated again when the exception handler branches back to the vector.</td>
<td>Vector Catch exceptions must be taken before other exceptions.</td>
</tr>
<tr>
<td>A Vector Catch exception can be generated as a result of an instruction fetch executed in any AArch32 mode except Hyp mode, including User mode.</td>
<td>Because a Vector Catch exception is generated as the result of an exception entry, the Vector Catch exception is only generated when the PE is in the AArch32 exception handling mode.</td>
</tr>
</tbody>
</table>
| If HCR.TGE is 1, Vector Catch exceptions can be generated for User mode instruction fetches from Non-secure PL1 vectors. | If HCR.TGE is 1, Vector Catch exceptions are never generated in Non-secure state, because:
  - Exceptions are routed away from Non-secure PL1 vectors, to PL2.
  - The architecture does not provide vector catch enable bits for the Hyp exception vectors. |

Depending on the implementation, some vector catch enable bits in the DBGVCR might be RES0. For example, if EL3 is not implemented or is implemented but is using AArch64, Monitor mode is not implemented, and so the enable bits for exception vectors for exceptions taken to Monitor mode are RES0. See Exception vectors that Vector Catch exceptions can be enabled for on page G2-9095 for the vector catch enable bits that exist for different implementations.

The debug exception enable controls on page G2-9042 describes the enable controls for Vector Catch exceptions.
G2.11.2 Exception vectors that Vector Catch exceptions can be enabled for

When the PE takes an exception, the exception vector is contained in a vector table at the Privilege level the exception is taken to.

Depending on the Security state and AArch32 mode the exception is taken to, when the exception is taken, the vector table used is the table that contains one of:

- Local exception vectors.
- Non-secure Local exception vectors.
- Secure Local exception vectors.
- Hyp exception vectors.
- Monitor exception vectors.

Table G2-21 on page G2-9095 shows which vector tables are implemented for different implementations. In the table:

- A dash, -, means that the Exception level is not implemented.
- 64 means that the Exception level is using AArch64.
- 32 means that the Exception level is using AArch32.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Vector table or tables implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL0 EL1 EL2 EL3</td>
<td></td>
</tr>
<tr>
<td>32 32 - -</td>
<td>Local exception vectors.</td>
</tr>
<tr>
<td>64 -</td>
<td>Non-secure Local exception vectors.</td>
</tr>
<tr>
<td>32 -</td>
<td>Non-secure Local exception vectors.</td>
</tr>
<tr>
<td></td>
<td>Hyp exception vectors.</td>
</tr>
<tr>
<td>- 64</td>
<td>Secure Local exception vectors.</td>
</tr>
<tr>
<td></td>
<td>Non-secure Local exception vectors.</td>
</tr>
<tr>
<td>- 32</td>
<td>Secure Local exception vectors.</td>
</tr>
<tr>
<td></td>
<td>Non-secure Local exception vectors.</td>
</tr>
<tr>
<td></td>
<td>Monitor exception vectors.</td>
</tr>
<tr>
<td>64 64</td>
<td>Secure Local exception vectors.</td>
</tr>
<tr>
<td></td>
<td>Non-secure Local exception vectors.</td>
</tr>
<tr>
<td>32 64</td>
<td>Secure Local exception vectors.</td>
</tr>
<tr>
<td></td>
<td>Non-secure Local exception vectors.</td>
</tr>
<tr>
<td></td>
<td>Hyp exception vectors.</td>
</tr>
<tr>
<td>32 32</td>
<td>Secure Local exception vectors.</td>
</tr>
<tr>
<td></td>
<td>Non-secure Local exception vectors.</td>
</tr>
<tr>
<td></td>
<td>Hyp exception vectors.</td>
</tr>
<tr>
<td></td>
<td>Monitor exception vectors.</td>
</tr>
</tbody>
</table>

For example, in an AArch32-only implementation that includes EL0, EL1, and EL3, when the PE takes an exception to Monitor mode, it uses the vector table containing Monitor exception vectors.

The tables that follow show the vectors that Vector Catch exceptions can be enabled for, and their corresponding vector catch enable bits in the DBGVCR:

- Table G2-22 on page G2-9096 shows the Local exception vectors, Secure Local exception vectors, and Non-secure Local exception vectors that Vector Catch exceptions can be enabled for.
The architecture does not provide vector catch enable bits for the Hyp exception vectors.

Table G2-22 Local exception vectors, Secure Local exception vectors, and Non-secure Local exception vectors that Vector Catch exceptions can be enabled for

<table>
<thead>
<tr>
<th>Vector catch enable bit</th>
<th>Exception type</th>
<th>Local exception vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local or Secure</td>
<td>Non-secure</td>
<td></td>
</tr>
<tr>
<td>Local exception vectors</td>
<td>Local exception vectors</td>
<td></td>
</tr>
<tr>
<td>SF</td>
<td>NSF</td>
<td>FIQ interrupt</td>
</tr>
<tr>
<td>SI</td>
<td>NSI</td>
<td>IRQ interrupt</td>
</tr>
<tr>
<td>SD</td>
<td>NSD</td>
<td>Data Abort</td>
</tr>
<tr>
<td>SP</td>
<td>NSP</td>
<td>Prefetch Abort</td>
</tr>
<tr>
<td>SS</td>
<td>NSS</td>
<td>Supervisor Call</td>
</tr>
<tr>
<td>SU</td>
<td>NSU</td>
<td>Undefined Instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a. If EL3 is implemented and is using AArch32, VBAR is banked. The Secure Local exception vectors use VBAR_{S} and the Non-secure Local Exception vectors use VBAR_{NS}.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table G2-23 Monitor exception vectors that Vector Catch exceptions can be enabled for

<table>
<thead>
<tr>
<th>Vector catch enable bit</th>
<th>Exception type</th>
<th>Monitor exception vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>MF</td>
<td>FIQ interrupt</td>
<td>MVBAR + 0x0000001C</td>
</tr>
<tr>
<td>MI</td>
<td>IRQ interrupt</td>
<td>MVBAR + 0x00000018</td>
</tr>
<tr>
<td>MD</td>
<td>Data Abort</td>
<td>MVBAR + 0x00000010</td>
</tr>
<tr>
<td>MP</td>
<td>PrefetchAbort</td>
<td>MVBAR + 0x0000000C</td>
</tr>
<tr>
<td>MS</td>
<td>Secure Monitor Call</td>
<td>MVBAR + 0x00000008</td>
</tr>
</tbody>
</table>

Note

There is no vector catch enable bit for Monitor trap exceptions.

G2.11.3 Generation of Vector Catch exceptions

How Vector Catch exceptions are generated depends on which form is implemented:

- **Address-matching form.**
- **Exception-trapping form on page G2-9097.**

**Address-matching form**

The PE compares the virtual address of each instruction in the program flow with some or all of the addresses in the enabled vector catch set, as follows:

- If EL3 is not implemented, the enabled vector catch set contains only Local exception vectors. The PE compares the virtual address of each instruction in the program flow, including those executed at EL0, with all addresses in the enabled vector catch set.
• If EL3 is implemented, the enabled vector catch set might contain one or more of the following:
  — Monitor exception vectors, if EL3 is using AArch32.
  — Secure Local exception vectors.
  — Non-secure Local exception vectors.

In this case, Table G2-24 on page G2-9097 shows which addresses, in the enabled vector catch set, the virtual address of each instruction in the program flow is compared with.

### Table G2-24 Comparisons made if the implementation includes EL3

<table>
<thead>
<tr>
<th>EL3 is using</th>
<th>For exceptions taken to:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Secure PL1 modes</td>
<td>Non-secure PL1 modes</td>
<td></td>
</tr>
<tr>
<td>AArch64</td>
<td>Secure Local exception vectors</td>
<td>Non-secure Local exception vectors</td>
<td></td>
</tr>
<tr>
<td>AArch32</td>
<td>Secure Local exception vectors and Monitor exception vectors</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For example, for exceptions taken to a Secure PL1 mode when EL3 is using AArch64, the virtual address of each instruction in the program flow is compared with each Secure Local exception vector in the enabled vector catch set.

For each instruction in the program flow, the PE tests for any possible Vector Catch exceptions before executing the instruction. If a match occurs, a Vector Catch exception is generated when the instruction is committed for execution, regardless of all of the following:

• Whether the instruction passes its Condition code check.
• Whether the instruction is executed as part of exception entry.
• If EL2 is implemented, what HCR. {IMO, FMO, AMO} are set to.
• If EL3 is implemented, what SCR. {IRQ, FIQ, EA} are set to.

### Exception-trapping form

When the PE takes an exception, it tests whether the exception is by branching to an exception vector in a subset of the enabled vector catch set, as follows:

• If EL3 is not implemented, the enabled vector catch set contains only Local exception vectors. The PE tests whether the exception is by branching to any address in the enabled vector catch set.

• If EL3 is implemented, the enabled vector catch set might contain one or more of the following:
  — Monitor exception vectors, if EL3 is using AArch32.
  — Secure Local exception vectors.
  — Non-secure Local exception vectors.

In this case, the PE tests whether the exception is by branching to a vector in one of the subsets that Table G2-25 on page G2-9097 shows. In the table, n/a means not applicable.

### Table G2-25 Subsets that the PE tests within if EL3 is implemented

<table>
<thead>
<tr>
<th>EL3 is using</th>
<th>For exceptions taken to:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Monitor mode</td>
</tr>
<tr>
<td>AArch64</td>
<td>n/a</td>
</tr>
<tr>
<td>AArch32</td>
<td>Monitor exception vectors</td>
</tr>
</tbody>
</table>

For example, for an exception taken to a Secure PL1 mode when EL3 is using AArch64, the PE tests whether the exception is by branching to any of the Secure Local exception vectors in the enabled vector address set.

If the exception is by branching to a vector in the subset, a Vector Catch exception is generated as part of exception entry. That is, a Vector Catch exception is generated instead of the exception handler executing its first instruction.
G2.11.4 Usage constraints

See the following subsections:

- Usage constraints that apply to both forms of vector catch.
- Usage constraints that apply only to the address-matching form.

Usage constraints that apply to both forms of vector catch

For Vector Catch exceptions enabled for either the Prefetch Abort exception vector or the Data Abort exception vector, if one of these exception types is taken to the Exception level that debug exceptions are targeting, behavior is CONSTRAINED UNPREDICTABLE. Either:

- Vector catch is ignored, therefore a Vector Catch exception is not generated.
- Vector catch generates a Prefetch Abort debug exception. For Vector Catch exceptions enabled for the Prefetch Abort exception vector, the PE might enter a recursive loop of Prefetch Abort exceptions causing Vector Catch exceptions and Vector Catch exceptions causing Prefetch Abort exceptions.

Note

The Exception level that debug exceptions are targeting is called the debug target Exception level, EL1. Routing debug exceptions on page G2-9043 describes how EL1 is derived.

Usage constraints that apply only to the address-matching form

Exception vectors are at word-aligned addresses, and:

- It is CONSTRAINED UNPREDICTABLE whether an enabled vector catch generates a Vector Catch exception for a 32-bit T32 instruction starting at the halfword-aligned address immediately prior to the vector address.
- T32 instructions that start at the halfword-aligned address immediately after the exception vector do not generate Vector Catch exceptions.

For the address-matching form, Vector Catch exceptions have the same priority as Breakpoint exceptions. If a single instruction causes both a Vector Catch exception and a Breakpoint exception, it is CONSTRAINED UNPREDICTABLE which of these debug exceptions the PE takes.

G2.11.5 Exception syndrome information and preferred return address for a Vector Catch exception

See the following:

- Exception syndrome information for a Vector Catch exception.
- Preferred return address for a Vector Catch exception on page G2-9099.

Note

Usually, the term exception syndrome is used only for exceptions taken to Hyp mode, or to AArch64 state. The referenced section uses the term more generally, to include exception information reported in the IFSR.

Exception syndrome information for a Vector Catch exception

The PE takes a Vector Catch exception as either:

- A Prefetch Abort exception if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp trap exception, if it is taken to PL2 because HCR.TGE or HDCR.TDE is 1. In this case, it is taken to Hyp mode.
If the exception is taken to:

**PL1 Abort mode**

The PE sets all of the following:
- **IFS.FS** to the code for a debug exception, 0b00010.
- **DBGDSCR.MOE** to 0b0101, to indicate a Vector Catch exception.
- The IFAR with an UNKNOWN value.

**PL2 Hyp mode**

The PE does all of the following:
- Records information about the exception in the Hypervisor Syndrome Register, HSR. See Table G2-26 on page G2-9099.
- Sets DBGDSCR.MOE to 0b0101, to indicate a Vector Catch exception.
- Sets the HIFAR to an unknown value.

---

**Preferred return address for a Vector Catch exception**

The preferred return address of a Vector Catch exceptions is the address of the instruction that was not executed because the PE took the Vector Catch exception instead.

This means that the preferred return address is the exception vector. This is true regardless of whether the address-matching form or the exception trapping form is implemented.

**G2.11.6 Pseudocode description of Vector Catch exceptions**

The `AArch32.VCRMatch()` pseudocode function checks whether the instruction at address generates a Vector Catch exception. It therefore shows the address-matching form of vector catch.

The `AArch32.CheckVectorCatch()` pseudocode function uses `AArch32.VCRMatch()` to test whether the instruction generates a Vector Catch exception, and if `AArch32.VCRMatch()` returns TRUE it generates that event.

The `AArch32.Abort()` function processes the FaultRecord object returned by `AArch32.CheckVectorCatch()`, as described in `Abort exceptions on page G4-9144`. If there is a Vector Catch exception, the `AArch32.Abort()` function generates a Prefetch Abort exception.
G2.12 Synchronization and debug exceptions

The behavior of debug depends on all of the following:

• The state of the external debug authentication interface.

• Indirect reads of:
  — External debug registers.
  — System registers, including system debug registers.
  — Special-purpose registers.

If a change is made to any of these, the effect of that change on debug exception generation cannot be relied on until after a Context synchronization event has occurred.

For any instructions executed between the time when the change is made and the time when the next Context synchronization event occurs, it is CONSTRAINED UNPREDICTABLE whether debug uses the state of the PE before the change, or the state of the PE after the change.

Example G2-3 Example of synchronization and Breakpoint exception generation

1. Software changes DBGDSCRext.MDBGen from 0 to 1.
2. An instruction is executed, that would cause a Breakpoint exception if self-hosted debug uses the state of the PE after the change.
3. A Context synchronization event occurs.

In this case, it is CONSTRAINED UNPREDICTABLE whether the instruction generates a Breakpoint exception.

Example G2-4 Example of synchronization and debug exceptions generation

1. Software unlocks the OS Lock.
2. The PE executes some instructions.
3. A Context synchronization event occurs.

During the time when the PE is executing some instructions, step 2, it is CONSTRAINED UNPREDICTABLE whether debug exceptions other than Breakpoint Instruction exceptions can be generated.

--- Note ---

Some register updates are self-synchronizing. Others require an explicit Context synchronization event. For more information, see:

• Synchronization of changes to AArch32 System registers on page G8-9326.
• Accessing PSTATE fields on page G1-8922.
• Synchronization of changes to the external debug registers on page H8-10358.

G2.12.1 State and mode changes without explicit context synchronization events

Most changes to the Exception level, and most changes to the Security state if EL3 is implemented, happen as a result of operations that are an explicit Context synchronization event. This is because taking an exception and returning from an exception are both explicit Context synchronization events, and the Privilege level and Security state can only change as a result of taking or returning from an exception.
However, some Security state and AArch32 mode changes can happen because of operations that are not an explicit *Context synchronization event*. These are:

- AArch32 mode changes caused by MSR and CPS instructions. A mode change might be to a mode at a lower Privilege level.

- If EL3 is using AArch32, a Security state change caused by a direct write to the SCR in a privileged mode other than Monitor mode, to set SCR.NS to 1.
Chapter G3
AArch32 Self-hosted Trace

This chapter describes the AArch32 self-hosted trace:

**Introductory information:**
- *About self-hosted trace* on page G3-9104.
- *Trace Sinks* on page G3-9104.
- *Register controls to enable self-hosted trace* on page G3-9104.

**Prohibited regions in trace:**
- *Controls to prohibit trace at Exception levels* on page G3-9105.
- *Self-hosted trace and address translation* on page G3-9105.

**Timestamps and Synchronization:**
- *Self-hosted trace timestamps* on page G3-9106.
- *Synchronization in self-hosted trace* on page G3-9107.
G3.1 About self-hosted trace

A trace unit generates trace data to describe the program flow of the PE.

The trace unit may be an implementation of any of the following:

- In Armv9, the Embedded Trace Extension (ETE). See Chapter D4 The Embedded Trace Extension.
- An IMPLEMENTATION DEFINED trace function.

If an Armv8.4-compliant PE implements an ETM Architecture trace unit that includes the ETM System register interface, FEAT_TRF must be implemented.

If an Armv8.4-compliant PE implements a Trace Unit that is either not an ETM Architecture trace unit or does not implement the ETM System register interface, Arm recommends that FEAT_TRF is implemented, but this is not mandatory. This is not applicable in Armv9.

If an Armv9-compliant PE implements FEAT_ETE, FEAT_TRF must be implemented.

Self-hosted trace happens when the agent controlling the trace collection is part of the same software stack as the software being traced. The agent controls prohibited regions. The information collected by the agent is sent to a trace sink.

The trace unit and the PE must have the same view of the debug authentication interface. If FEAT_TRF is implemented, ExternalNoninvasiveDebugEnabled() is always TRUE.

G3.1.1 Trace Sinks

The trace unit sends the trace data to a trace sink. A system might include multiple trace sinks, and allow software to configure which trace sink or sinks are used.

An example of an internal trace sink is an Embedded Trace Router (ETR), which allows software to define a buffer in memory. Trace data is written to this buffer.

If FEAT_TRBE is implemented, the PE includes a Trace Buffer Unit. Trace data is written directly to memory by the Trace Buffer Unit. See Chapter D6 The Trace Buffer Extension.

In Armv8, Arm recommends that a system that includes FEAT_TRF incorporates an ETR, and follows the system architecture described by the CoreSight Base System Architecture (CS-BSA).

G3.1.2 Register controls to enable self-hosted trace

For EL1 using AArch64, see Chapter D3 AArch64 Self-hosted Trace.

If FEAT_TRF is implemented, self-hosted trace is enabled if one of the following is true:

- EDSCR.TFO == 0.
- EDSCR.TFO == 1, EL3 is implemented, SDCR.STE == 1, and ExternalSecureNoninvasiveDebugEnabled() == FALSE.
- EDSCR.TFO == 1, EL3 is not implemented, the PE executes in Secure state, and ExternalSecureNoninvasiveDebugEnabled() = FALSE.

The pseudocode function SelfHostedTraceEnabled() shows these rules.

If FEAT_TRF is not implemented, SelfHostedTraceEnabled() returns FALSE.

While SelfHostedTraceEnabled() is FALSE, ExternalNoninvasiveDebugAllowed() controls whether tracing is prohibited or allowed in each Security state.

The self-hosted trace extensions do not provide any mechanism to control software access to the trace unit external debug interface.
G3.2 Prohibited regions in self-hosted trace

Trace is not generated in prohibited regions. The pseudocode function `TraceAllowed()` indicates whether tracing is allowed in the current Security state and Exception level.

The IMPLEMENTATION DEFINED debug authentication interface can allow an external agent to disable the self-hosted trace extension.

If `SelfHostedTraceEnabled()` == TRUE, tracing is prohibited in Secure state when `SDCR.STE == 0`.

G3.2.1 Controls to prohibit trace at Exception levels

If `SelfHostedTraceEnabled()` == TRUE, `TRFCR, TRFCR_EL1, TRFCR_EL2` and `HTRFCR` control whether trace is prohibited at an Exception level. While `SelfHostedTraceEnabled()` == FALSE, these registers are ignored.

If `SelfHostedTraceEnabled()` == TRUE, tracing is prohibited at EL0 if one of the following is true:
• The Effective value of `HCR_EL2.TGE == 0` and `TRFCR_EL1.E0TRE == 0`.
• The Effective value of `HCR.TGE == 0` and `TRFCR.E0TRE == 0`.
• The Effective value of `HCR_EL2.TGE == 1` and `TRFCR_EL2.E0HTRE == 0`.

If `SelfHostedTraceEnabled()` == TRUE, tracing is prohibited at EL1 if `TRFCR.E1TRE == 0`.

If `SelfHostedTraceEnabled()` == TRUE, tracing is prohibited at EL2 if `HTRFCR.E2TRE == 0`.

If `SelfHostedTraceEnabled()` == TRUE, tracing is prohibited at EL3 if one of the following is true:
• EL3 is in AArch64 state.
• EL3 is in AArch32 state and `TRFCR.E1TRE == 0`.

The pseudocode `TraceAllowed()` shows the preceding rules.

If `SelfHostedTraceEnabled()` == TRUE, no events are exported to the trace unit when tracing is prohibited.

If `SelfHostedTraceEnabled()` == FALSE, no events are exported to the trace unit when the PE is in Secure state and counting in Secure state is prohibited.

If `FEAT_ETE` is not implemented, when `PMCR_EL0.X==0` or `PMCR.X==0`, no PMU events are exported to the trace unit.

Otherwise, PMU events are exported to the trace unit.

G3.2.2 Self-hosted trace and address translation

A hypervisor can use `HTRFCR.CX` to control visibility of `VTTBR.VMID`.

If `SelfHostedTraceEnabled()` == TRUE, and `HTRFCR.CX == 0`, or if EL2 is not implemented:
• The value of `VTTBR.VMID` is not traced.
• Comparisons with `VTTBR.VMID` do not match and results of comparison are not exposed through the comparators.

The trace unit may either prohibit trace for these values, or may record a `VTTBR.VMID` value of zero in the trace.
G3.3 Self-hosted trace timestamps

For EL1 using AArch64, see Chapter D3 AArch64 Self-hosted Trace.

The trace timestamp is a value that represents the passage of time in real-time. It is calculated from a counter which increments all the time, when the PE is generating trace and when the PE is in a prohibited region.

While \texttt{SelfHostedTraceEnabled()} == FALSE, the external trace provides the trace timestamp. If the external trace is a standard CoreSight system, the relationship between CoreSight time and the Generic Timer counter is IMPLEMENTATION DEFINED.

When \texttt{SelfHostedTraceEnabled()} == TRUE, the trace time stamp is one of the following:

- Physical time, which is defined by the physical count value returned by \texttt{PhysicalCountInt()}.
- If \texttt{FEAT_ECV} is implemented and EL2 is executing at AArch64, offset physical time, which is defined as the value of (\texttt{PhysicalCountInt()} - \texttt{CNTPOFF_EL2}). However, the physical offset is treated as zero if \texttt{FEAT_ECV} is disabled.
- Virtual time, which is defined as the value of (\texttt{PhysicalCountInt()} - \texttt{CNTVOFF}). The virtual offset is always \texttt{CNTVOFF}, including when a read of \texttt{CNTVCT} at the current Exception level would treat the virtual offset as zero.

The fields \texttt{TRFCR_EL2.TS}, \texttt{TRFCR.TS} and \texttt{HTRFCR.TS} control which counter is used for self-hosted trace.

The timestamp used for trace is shown in Table G3-1 on page G3-9106.

<table>
<thead>
<tr>
<th>\texttt{SelfHostedTraceEnabled()}</th>
<th>\texttt{TRFCR_EL2.TS or HTRFCR.TS}</th>
<th>\texttt{TRFCR_EL1.TS}</th>
<th>Timestamp traced</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>xx</td>
<td>xx</td>
<td>CoreSight time</td>
</tr>
<tr>
<td>TRUE</td>
<td>0b00</td>
<td>0b01</td>
<td>\texttt{PhysicalCountInt()} - \texttt{CNTVOFF}</td>
</tr>
<tr>
<td></td>
<td>0b00</td>
<td>0b11</td>
<td>\texttt{PhysicalCountInt()}</td>
</tr>
<tr>
<td></td>
<td>0b01</td>
<td>xx</td>
<td>\texttt{PhysicalCountInt()} - \texttt{CNTVOFF} or \texttt{PhysicalCountInt()} - \texttt{CNTVOFF_EL2}</td>
</tr>
<tr>
<td></td>
<td>0b11</td>
<td>xx</td>
<td>\texttt{PhysicalCountInt()}</td>
</tr>
</tbody>
</table>

\textbf{Note}

The value of \texttt{HCR_EL2.E2H} does not affect the counter used for the trace timestamp.
G3.4 Synchronization in self-hosted trace

The trace unit is an indirect observer of the trace control registers.

While `SelfHostedTraceEnabled()` == TRUE, indirect reads of the trace filter control fields, TRFCR.{E1TRE, E0TRE} and HTRFCR.{E2TRE, E0HTRE} are treated as indirect reads made by the instruction being traced, and are subject to the standard requirements for synchronization of System register accesses.

The `TSB CSYNC` operation is used to ensure that a trace operation, due to a trace unit generating trace for an instruction has completed. The `TSB CSYNC` operation may be reordered with respect to other instructions, so must be combined with at least one Context synchronization event to ensure the operations are executed in the required order. This means that a direct write to TRFCR or HTRFCR is guaranteed to be observed by the trace unit only after a subsequent Context synchronization event. For more information, see Trace Synchronization Barrier (TSB CSYNC) on page E2-7184.

While `SelfHostedTraceEnabled()` == FALSE, the trace unit might impose stronger synchronization requirements.
Chapter G4
The AArch32 System Level Memory Model

This chapter provides a system level view of the general features of the memory system. It contains the following sections:

- About the memory system architecture on page G4-9110.
- Address space on page G4-9111.
- Mixed-endian support on page G4-9112.
- AArch32 cache and branch predictor support on page G4-9113.
- System register support for IMPLEMENTATION DEFINED memory features on page G4-9138.
- External aborts on page G4-9139.
- Memory barrier instructions on page G4-9141.
- Pseudocode description of general memory System instructions on page G4-9142.
G4.1 About the memory system architecture

The Arm architecture supports different implementation choices for the memory system microarchitecture and memory hierarchy, depending on the requirements of the system being implemented. In this respect, the memory system architecture describes a design space in which an implementation is made. The architecture does not prescribe a particular form for the memory systems. Key concepts are abstracted in a way that permits implementation choices to be made while enabling the development of common software routines that do not have to be specific to a particular microarchitectural form of the memory system. For more information about the concept of a hierarchical memory system, see Memory hierarchy on page E2-7188.

G4.1.1 Form of the memory system architecture

The A-profile architecture includes a Virtual Memory System Architecture (VMSA). Chapter G5 The AArch32 Virtual Memory System Architecture describes the AArch32 view of the VMSA.

G4.1.2 Memory attributes

Memory types and attributes on page E2-7198 describes the memory attributes, including how different memory types have different attributes. Each location in memory has a set of memory attributes, and the translation tables define the virtual memory locations, and the attributes for each location.

Table G4-1 on page G4-9110 shows the memory attributes that are visible at the system level.

<table>
<thead>
<tr>
<th>Memory type</th>
<th>Shareability</th>
<th>Cacheability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>Outer Shareable</td>
<td>Non-cacheable.</td>
</tr>
<tr>
<td>Normal</td>
<td>One of:</td>
<td>One of:</td>
</tr>
<tr>
<td></td>
<td>• Non-shareable.</td>
<td>• Non-cacheable.</td>
</tr>
<tr>
<td></td>
<td>• Inner Shareable.</td>
<td>• Write-Through Cacheable.</td>
</tr>
<tr>
<td></td>
<td>• Outer Shareable.</td>
<td>• Write-Back Cacheable.</td>
</tr>
</tbody>
</table>

a. Takes additional attributes, see Device memory on page E2-7202.
b. See also Cacheability, cache allocation hints, and cache transient hints on page G4-9116.

For more information on Cacheability and Shareability, see The Cacheability and Shareability memory attributes on page E2-7189, Non-shareable Normal memory on page E2-7200, and Caches and memory hierarchy on page E2-7188.
G4.2 Address space

The architecture is designed to support a wide range of applications with different memory requirements. It supports a range of physical address (PA) sizes, and provides associated control and identification mechanisms. For more information, see About VMSAv8-32 on page G5-9146.

G4.2.1 Address space overflow or underflow

This subsection describes address space overflow or underflow:

Instruction address space overflow

When a PE performs a normal, sequential execution of instructions, it calculates:

(address_of_current_instruction) + (size_of_executed_instruction)

This calculation is performed after each instruction to determine which instruction to execute next.

If the address calculation performed after executing an A32 or T32 instruction overflows 0xFFFF FFFF, the program counter becomes UNKNOWN.

If the PE executes an instruction for which the instruction address, size, and alignment mean that it contains the bytes 0xFFFFFFFF and 0x00000000, the bytes that apparently from 0x00000000 onwards come from an UNKNOWN address.

Data address space overflow and underflow

If the PE executes a load or store instruction for which the computed address, total access size, and alignment mean that it accesses bytes 0xFFFFFFFF and 0x00000000, then the bytes that apparently come from 0x00000000 onwards come from UNKNOWN addresses.
G4.3 Mixed-endian support

Table G4-2 on page G4-9112 shows the endianness of explicit data accesses and translation table walks.

<table>
<thead>
<tr>
<th>Exception level</th>
<th>Explicit data accesses</th>
<th>Stage 1 translation table walks</th>
<th>Stage 2 translation table walks</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL0</td>
<td>PSTATE.E</td>
<td>SCTLR(S/NS).EE</td>
<td>HSCTLR.EE</td>
</tr>
<tr>
<td>EL1</td>
<td>PSTATE.E</td>
<td>SCTLR(S/NS).EE</td>
<td>HSCTLR.EE</td>
</tr>
<tr>
<td>EL2</td>
<td>PSTATE.E</td>
<td>HSCTLR.EE</td>
<td>n/a</td>
</tr>
<tr>
<td>EL3</td>
<td>PSTATE.E</td>
<td>SCTLR(S).EE</td>
<td>n/a</td>
</tr>
</tbody>
</table>

AArch32 state provides the following options for endianness support:

- All Exception levels support mixed-endianness:
  - SCTLR(S/NS).EE, HSCTLR.EE, and PSTATE.E are RW.
- Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only little-endianness:
  - SCTLR(S/NS).EE and HSCTLR.EE are RES0. PSTATE.E is RW when in EL0 and RES0 when in EL1, EL2, or EL3. SPSR.E is also RES0 when not returning to EL0.
- Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only big-endianness:
  - SCTLR(S/NS).EE and HSCTLR.EE are RES1. PSTATE.E is RW when in EL0 and RES1 when in EL1, EL2, or EL3. SPSR.E is also RES1 when not returning to EL0.
- All Exception levels support only little-endianness:
  - Each of SCTLR(S/NS).EE, HSCTLR.EE, PSTATE.E, and SPSR.E is RES0.
- All Exception levels support only big-endianness:
  - Each of SCTLR(S/NS).EE, HSCTLR.EE, PSTATE.E, and SPSR.E is RES1.

If mixed endian support is implemented for an Exception level using AArch32, endianness is controlled by PSTATE.E. For exception returns to AArch32 state, PSTATE.E is copied from SPSR_ELx.E. If the target Exception level supports only little-endian accesses, SPSR_ELx.E is RES0. If the target Exception level supports only big-endian accesses, SPSR_ELx.E is RES1.

Note

- When using AArch32, Arm deprecates PSTATE.E having a different value from the equivalent System register EE bit when in EL1, EL2 or EL3. The use of the SETEND instruction is also deprecated.
- If the higher Exception levels are using AArch64, the corresponding registers are:
  - SCTLR_EL1 for SCTLR(NS).
  - SCTLR_EL2 for HSCTLR.
  - SCTLR_EL3 for SCTLR(S).

The BigEndian() function determines whether the current Exception level and Execution state is using big-endian data.

For more information about endianness in the Arm architecture, see Endian support on page E2-7195.
G4.4  AArch32 cache and branch predictor support

The following sections describe the support for caches and branch predictors in AArch32 state:

- **General behavior of the caches.**
- **Cache identification** on page G4-9114.
- **Cacheability, cache allocation hints, and cache transient hints** on page G4-9116.
- **Enabling and disabling the caching of memory accesses in AArch32 state** on page G4-9117.
- **Behavior of caches at reset** on page G4-9119.
- **About cache maintenance in AArch32 state** on page G4-9119.
- **AArch32 cache and branch predictor maintenance instructions** on page G4-9123.
- **Execution and data prediction restriction System instructions** on page G4-9134.
- **Cache lockdown** on page G4-9136.
- **System level caches** on page G4-9137.

See also Chapter G5 *The AArch32 Virtual Memory System Architecture*, and in particular *Caches in VMSAv8-32* on page G5-9235.

---

**Note**

- Branch predictors typically use a form of cache to hold branch target data. Therefore, they are included in this section.

- In the instruction mnemonics, MVA is a synonym for VA.

---

G4.4.1  General behavior of the caches

When a memory location is marked with a Normal Cacheable memory attribute, determining whether a copy of the memory location is held in a cache still depends on many aspects of the implementation. The following non-exhaustive list of factors might be involved:

- The size, line length, and associativity of the cache.
- The cache allocation algorithm.
- Activity by other elements of the system that can access the memory.
- Speculative instruction fetching algorithms.
- Speculative data fetching algorithms.
- Interrupt behaviors.

Given this range of factors, and the large variety of cache systems that might be implemented, the architecture cannot guarantee whether:

- A memory location present in the cache remains in the cache.
- A memory location not present in the cache is brought into the cache.

Instead, the following principles apply to the behavior of caches:

- The architecture has a concept of an entry locked down in the cache. How lockdown is achieved is IMPLEMENTATION DEFINED, and lockdown might not be supported by:
  - A particular implementation.
  - Some memory attributes.

- An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.

- A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.
The AArch32 System Level Memory Model
G4.4 AArch32 cache and branch predictor support

Note

For more information, see The interaction of cache lockdown with cache maintenance instructions on page G4-9136.

• Any memory location that has a Normal Cacheable attribute at either the current Exception level or at a higher Exception level can be allocated to a cache at any time.

• It is guaranteed that no memory location that does not have a Normal Cacheable attribute is allocated into the cache.

• It is guaranteed that no memory location is allocated to the cache if it has a Normal Non-cacheable attribute or any type of Device memory attribute in both:
  — The translation regime at the current Exception level.
  — The translation regime at any higher Exception level.

• For data accesses, any memory location with a Normal Inner Shareable or Normal Outer Shareable attribute is guaranteed to be coherent with all Requesters in its Shareability domain.

• Any memory location is not guaranteed to remain incoherent with the rest of memory.

• The eviction of a cache entry from a cache level can overwrite memory that has been written by another observer only if the entry contains a memory location that has been written to by an observer in the Shareability domain of that memory location. The maximum size of the memory that can be overwritten is called the Cache Write-back Granule. In some implementations the CTR identifies the Cache Write-back Granule.

• The allocation of a memory location into a cache cannot cause the most recent value of that memory location to become invisible to an observer, if it was previously visible to that observer.

Note

The Cacheability attribute of an address is determined by the applicable translation table entry for that address, as modified by any applicable System register Cacheability controls, such as the SCTLR. {I, C} controls.

For the purpose of these principles, a cache entry covers at least 16 bytes and no more than 2KB of contiguous address space, aligned to the size of the cache entry.

G4.4.2 Cache identification

The cache identification consists of a set of registers that describe the implemented caches that are affected by cache maintenance instructions executed on the PE. This includes cache maintenance instructions that:

• Affect the entire cache, for example ICIALLUIS.
• Operate by VA, for example ICIMVAU.
• Operate by set/way, for example DCISW.

The cache identification registers are:

• A single Cache Type Register, CTR, that defines:
  — The minimum line length of any of the instruction caches affected by the instruction cache maintenance instructions.
  — The minimum line length of any of the data or unified caches, affected by the data cache maintenance instructions.
  — The cache indexing and tagging policy of the Level 1 instruction cache.

Note

It is IMPLEMENTATION DEFINED whether caches beyond the PoC will be reported by this mechanism, and because of the possible existence of system caches some caches before the PoC might not be reported. For more information about system caches, see System level caches on page G4-9137.
• A single Cache Level ID Register, CLIDR, that defines:
  — The type of cache that is implemented and can be maintained using the architected cache maintenance
    instructions that operate by set/way or operate on the entire cache at each cache level, up to the
    maximum of seven levels.
  — The Level of Unification Inner Shareable (LoUIS), Level of Coherence (LoC) and the Level of
    Unification (LoU) for the caches. See Terms used in describing the cache maintenance instructions on
    page G4-9120 for a definition of these terms.
  — An optional ICB field to indicate the boundary between the caches used for caching Inner Cacheable
    memory regions and those used only for caching Outer Cacheable regions.

• A single Cache Size Selection Register, CSSELR, that selects the cache level and sort of cache (Instruction,
  Data/Unified/Tag) of the current Cache Size Identification Register.

• For each implemented cache that is identifiable by this mechanism, across all the levels of caching, a Cache
  Size Identification Register, that defines:
  — Whether the cache supports Write-Through, Write-Back, Read-Allocate, and Write-Allocate.
  — The number of sets, associativity, and line length of the cache. See Terms used in describing the cache
    maintenance instructions on page G4-9120 for a definition of these terms.

Note
From Armv8.3, it is possible to have multiple Cache Size Identification Registers. For more details, see
Possible formats of the Cache Size Identification Registers, CCSIDR and CCSIDR2.

To determine the cache topology associated with a PE:

1. Read the Cache Type Register to find the indexing and tagging policy used for the Level 1 instruction cache.
   This register also provides the size of the smallest cache lines used for the instruction caches, and for the data
   and unified caches. These values are used in cache maintenance instructions.

2. Read the Cache Level ID Register to find what caches are implemented. The register includes seven Cache
   type fields, for cache levels 1 to 7. Scanning these fields, starting from Level 1, identifies the instruction, data
   or unified caches implemented at each level. This scan ends when it reaches a level at which no caches are
   defined. The Cache Level ID Register also specifies the Level of Unification (LoU) and the Level of
   Coherence (LoC) for the cache implementation.

3. For each cache identified at stage 2:
   • Write to the Cache Size Selection Register to select the required cache. A cache is identified by its
     level, and whether it is:
     — An instruction cache.
     — A data or unified cache.
   • Read the Cache Size Identification Register to find details of the cache.

Possible formats of the Cache Size Identification Registers, CCSIDR and CCSIDR2

From Armv8.3, two different formats are available for defining the number of sets and associativity of the currently
selected cache. For a definition of these terms, see Terms used in describing the cache maintenance instructions on
page G4-9120.

When FEAT_CCIDX is implemented:

• There are two Cache Size Identification Registers, CCSIDR and CCSIDR2.
• The length of the CCSIDR.Assoc field is 21 bits. This limits the associativity of the currently selected cache
to 2^{21}.
• The length of the CCSIDR2.NumSets field is 24 bits. This limits the number of sets in the currently selected
  cache to 2^{24}.

This is the 64-bit format of the Cache Size Identification Register.
When FEAT_CCIDX is not implemented:

- There is a single Cache Size Identification Register, CCSIDR.
- The length of the CCSIDR.Assoc field is 10 bits. This limits the associativity of the currently selected cache to $2^{10}$.
- The length of the CCSIDR.NumSets field is 15 bits. This limits the number of sets in the currently selected cache to $2^{15}$.

This is the 32-bit format of the Cache Size Identification Register.

When one of these formats is implemented, it is implemented across all the levels of caching.

### G4.4.3 Cacheability, cache allocation hints, and cache transient hints

Cacheability applies only to Normal memory, and is defined independently for Inner and Outer cache locations. All types of Device memory are always treated as Non-cacheable.

As described in Memory types and attributes on page E2-7198, the memory attributes include a cacheability attribute that is one of:

- Non-cacheable.
- Write-Through cacheable.
- Write-Back cacheable.

Cacheability attributes other than Non-cacheable can be complemented by a cache allocation hint. This is an indication to the memory system of whether allocating a value to a cache is likely to improve performance. In addition, it is IMPLEMENTATION DEFINED whether a cache transient hint is supported, see Transient cacheability hint.

The cache allocation hints are assigned independently for read and write accesses, and therefore when the Transient hint is supported the following cache allocation hints can be used:

- **For read accesses:** Read-Allocate, Transient Read-Allocate, or No Read-Allocate.
- **For write accesses:** Write-Allocate, Transient Write-Allocate, or No Write-Allocate.

#### Note

- A Cacheable location with both No Read-Allocate and No Write-Allocate hints is not the same as a Non-cacheable location. A Non-cacheable location has coherency guarantees for all observers within the system that do not apply for a location that is Cacheable, No Read-Allocate, No Write-Allocate.
- Implementations can use the cache allocation hints to limit cache pollution to a part of a cache, such as to a subset of ways.
- For VMSAv8-32 translation table walks using the Long-descriptor translation table format, the appropriate TCR.\{IRGNn, ORGNn\} fields define the memory attributes of the translation tables, including the cacheability. However, this assignment supports only a subset of the cacheability attributes described in this section.

The architecture does not require an implementation to make any use of cache allocation hints. This means an implementation might not make any distinction between memory locations with attributes that differ only in their cache allocation hint.

#### Transient cacheability hint

It is IMPLEMENTATION DEFINED whether a Transient hint is supported for the VMSAv8-32 translation scheme when using the Long-descriptor translation table format. In an implementation that supports the Transient hint, the Transient hint is a qualifier of the cache allocation hints, and indicates that the benefit of caching is for a relatively short period. It indicates that it might be better to restrict allocation of transient entries, to avoid possibly casting-out other, less transient, entries.
Note
The architecture does not specify what is meant by a relatively short period.

When using the Short-descriptor translation table format, VMSAv8-32 cannot support the Transient hint.

The description of the MAIR0, MAIR1, HMAIR0, and HMAIR1 registers includes the assignment of the Transient attribute in an implementation that supports this option. In this assignment:

- The Transient hint is defined independently for Inner Cacheable and Outer Cacheable memory regions.
- A single Transient hint applies to both read and write accesses to a memory region.

G4.4.4 Enabling and disabling the caching of memory accesses in AArch32 state

Cacheability control fields can force all memory locations with the Normal memory type to be treated as Non-cacheable, regardless of their assigned Cacheability attribute. Independent controls are provided for each stage of address translation, with separate controls for:

- Data accesses. These controls also apply to accesses to the translation tables.
- Instruction accesses.

Note
These Cacheability controls replace the cache enable controls provided in previous versions of the Arm architecture.

In AArch32 state, the Cacheability control fields and their effects are as follows:

For the Non-secure PL1&0 translation regime

The Non-secure instance of SCTLR holds the EL1 controls that affect cacheability:

- When the value of SCTLR.C is 0:
  - All stage 1 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the PL1&0 stage 1 translation tables are Non-cacheable.
- When the value of SCTLR.I is 0:
  - All stage 1 translations for instruction accesses to Normal memory are Non-cacheable.
- When the value of HCR2.CD is 1:
  - All stage 2 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the PL1&0 stage 2 translation tables are Non-cacheable.
- When the value of HCR2.ID is 1:
  - All stage 2 translations for instruction accesses to Normal memory are Non-cacheable.
- When the value of HCR.DC is 1, all Non-secure stage 1 translations and all accesses to the Non-secure EL1&0 stage 1 translation tables, are treated as accesses to Normal Non-shareable Inner Write-Back Cacheable Read-Allocate Write-Allocate, Outer Write-Back Cacheable Read-Allocate Write-Allocate memory, regardless of the value of SCTLR.C. This applies to translations for both data and instruction accesses.

In addition, when the value of SCTLR.M is 0, indicating that the stage 1 translations are disabled for the translation regime, then if EL2 is using AArch32 and the value of HCR.DC is 0 or if EL2 is using AArch64 and the value of HCR_EL2.DC is 0, then:

- If the value of SCTLR.I is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
- If the value of SCTLR.I is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through cacheable, Outer Write-Through cacheable.
Note

- In Non-secure state, the stage 1 and stage 2 cacheability attributes are combined as described in Combining the Cacheability attribute on page G5-9214.
- The Non-secure SCTLR.{C, I} and HCR.DC fields have no effect on the Secure PL1&0 and EL2 translation regimes.
- The HCR2.{ID, CD} fields affect only stage 2 of the Non-secure PL1&0 translation regime.
- In Non-secure state, the PL1&0 translation regime can be described as the Non-secure EL1&0 translation regime. This is consistent with the equivalent AArch64 descriptions.
- When FEAT_XS is implemented SCTLR.{C, I} and HCR2.{ID, CD} fields have no effect on the value of the XS attribute.

For the Secure PL1&0 translation regime

The Secure instance of SCTLR holds the controls that determine cacheability:

- When the value of SCTLR.C is 0:
  - All data accesses to Normal memory using the Secure PL1&0 translation regime are Non-cacheable.
  - All accesses to the Secure PL1&0 translation tables are Non-cacheable.
- When the value of SCTLR.I is 0:
  - All instruction accesses to Normal memory using the Secure PL1&0 translation regime are Non-cacheable.

In addition, when the value of SCTLR.M is 0, indicating that stage 1 translations are disabled, then:

- If the value of SCTLR.I is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
- If the value of SCTLR.I is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through cacheable, Outer Write-Through cacheable.

Note

- The Secure SCTLR.{I, C, M} fields have no effect on the Non-secure PL1&0 and EL2 translation regimes.
- When FEAT_XS is implemented, the SCTLR.{I, C} fields have no effect on the value of the XS attribute.

For the EL2 translation regime

- When the value of HSCTLR.C is 0:
  - All data accesses to Normal memory using the EL2 translation regime are Non-cacheable.
  - All accesses to the EL2 translation tables are Non-cacheable.
- When the value of HSCTLR.I is 0:
  - All instruction accesses to Normal memory using the EL2 translation regime are Non-cacheable.

In addition, when the value of HSCTLR.M is 0, indicating that stage 1 translations are disabled, then:

- If the value of HSCTLR.I is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
- If the value of HSCTLR.I is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through cacheable, Outer Write-Through cacheable.
The effect of the SCTLR.C or HSCTLR.C and HCR2.CD bits is reflected in the result of the address translation instructions in the PAR.

--- Note ---

- The requirements in this section mean the architecturally required effects of SCTLR.I and HSCTLR.I are limited to their effects on caching instruction accesses in unified caches.

- This specification can give rise to different cacheability attributes between instruction and data accesses to the same location. Where this occurs, the measures for mismatch memory attributes described in MisMatched memory attributes on page E2-7208 must be followed to manage the corresponding loss of coherency.

---

**G4.4.5 Behavior of caches at reset**

The following rules apply to caches at reset:

- All caches reset to IMPLEMENTATION DEFINED states that might be UNKNOWN.

- The Cacheability control fields described in Enabling and disabling the caching of memory accesses in AArch32 state on page G4-9117 reset to values that force all memory locations to be treated as Non-cacheable.

--- Note ---

This applies only to the controls that apply to the Translation regime that is used by the Exception level, PE mode, and Security state entered on reset.

- An implementation can require the use of a specific cache initialization routine to invalidate its storage array before caching is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, and the routine must be documented clearly as part of the documentation of the device.

- If an implementation permits cache hits when the Cacheability control fields force all memory locations to be treated as Non-cacheable then the cache initialization routine must:
  - Provide a mechanism to ensure the correct initialization of the caches.
  - Be documented clearly as part of the documentation of the device.

In particular, if an implementation permits cache hits when the Cacheability controls force all memory locations to be treated as Non-cacheable, and the cache contents are not invalidated at reset, the initialization routine must avoid any possibility of running from an uninitialized cache. It is acceptable for an initialization routine to require a fixed instruction sequence to be placed in a restricted range of memory.

- Arm recommends that whenever an invalidation routine is required, it is based on the cache maintenance instructions.

Similar rules apply to:

- Branch predictor behavior, see Behavior of the branch predictors at reset on page G4-9127.

- TLB behavior, see TLB behavior at reset on page G5-9217.

---

**G4.4.6 About cache maintenance in AArch32 state**

The following sections give general information about cache maintenance:

- Terms used in describing the cache maintenance instructions on page G4-9120.

- Abstraction of the cache hierarchy on page G4-9122.
The following sections describe the AArch32 state cache maintenance instructions:

- AArch32 instruction cache maintenance instructions (IC*) on page G4-9124.
- AArch32 data cache maintenance instructions (DC*) on page G4-9125.

Note
Some descriptions of the cache maintenance instructions refer to the Cacheability of the address on which the instruction operates. The Cacheability of an address is determined by the applicable translation table entry for that address, as modified by any applicable System register Cacheability controls, such as the SCTLR.\{I, C\} controls.

Terms used in describing the cache maintenance instructions

Cache maintenance instructions are defined to act on particular memory locations. Instructions can be defined:

- By the virtual address of the memory location to be maintained, referred to as operating by VA.
- By a mechanism that describes the location in the hardware of the cache, referred to as operating by set/way.

In addition, for instruction caches and branch predictors, there are instructions that invalidate all entries.

The following subsections define the terms used in the descriptions of the cache maintenance instructions:

- Terminology for cache maintenance instructions operating by set/way.
- Terminology for Clean, Invalidate, and Clean and Invalidate instructions on page G4-9121.

Note
There is no terminology specific to cache maintenance instructions that operate by VA. When all applicable stages of translation are disabled, the VA used is identical to the PA. For more information about memory system behavior when address translation is disabled, see The effects of disabling address translation stages on VMSAv8-32 behavior on page G5-9154.

Terminology for cache maintenance instructions operating by set/way

Cache maintenance instruction that operate by set/way refer to the particular structures in a cache. Three parameters describe the location in a cache hierarchy that an instruction works on. These parameters are:

Level
The cache level of the hierarchy. The number of levels of cache is IMPLEMENTATION DEFINED. The cache levels that can be managed using the architected cache maintenance instructions that operate by set/way can be determined from the CLIDR.

In the Arm architecture, the lower numbered cache levels are those closest to the PE. See Memory hierarchy on page E2-7188.

Set
Each level of a cache is split up into a number of sets. Each set is a set of locations in a cache level to which an address can be assigned. Usually, the set number is an IMPLEMENTATION DEFINED function of an address.

In the Arm architecture, sets are numbered from 0.

Way
The associativity of a cache is the number of locations in a set to which a specific address can be assigned. The way number specifies one of these locations.

In the Arm architecture, ways are numbered from 0.

Note
Because the allocation of a memory address to a cache location is entirely IMPLEMENTATION DEFINED, Arm expects that most portable software will use only the cache maintenance instructions by set/way as single steps in a routine to perform maintenance on the entire cache.
Terminology for Clean, Invalidate, and Clean and Invalidate instructions

Caches introduce coherency problems in two possible directions:

1. An update to a memory location by a PE that accesses a cache might not be visible to other observers that can access memory. This can occur because new updates are still in the cache and are not visible yet to the other observers that do not access that cache.

2. Updates to memory locations by other observers that can access memory might not be visible to a PE that accesses a cache. This can occur when the cache contains an old, or stale, copy of the memory location that has been updated.

The Clean and Invalidate instructions address these two issues. The definitions of these instructions are:

Clean A cache clean instruction ensures that updates made by an observer that controls the cache are made visible to other observers that can access memory at the point to which the instruction is performed. Once the Clean has completed, the new memory values are guaranteed to be visible to the point to which the instruction is performed, for example to the Point of Unification.

The cleaning of a cache entry from a cache can overwrite memory that has been written by another observer only if the entry contains a location that has been written to by an observer in the Shareability domain of that memory location.

Invalidate A cache invalidate instruction ensures that updates made visible by observers that access memory at the point to which the invalidate is defined, are made visible to an observer that controls the cache. This might result in the loss of updates to the locations affected by the invalidate instruction that have been written by observers that access the cache, if those updates have not been cleaned from the cache since they were made.

If the address of an entry on which the invalidate instruction operates is Normal, Non-cacheable or any type of Device memory then an invalidate instruction also ensures that this address is not present in the cache.

Note Entries for addresses that are Normal Cacheable can be allocated to the cache at any time, and so the cache invalidate instruction cannot ensure that the address is not present in a cache.

Clean and Invalidate

A cache clean and invalidate instruction behaves as the execution of a clean instruction followed immediately by an invalidate instruction. Both instructions are performed to the same location.

The points to which a cache maintenance instruction can be defined differ depending on whether the instruction operates by VA or by set/way:

- For instructions operating by set/way, the point is defined to be to the next level of caching. For the All operations, the point is defined as the Point of Unification for each location held in the cache.

- For instruction operating by VA, two conceptual points are defined:

  Point of Coherency (PoC)

  The point at which all agents that can access memory are guaranteed to see the same copy of a memory location for accesses of any memory type or cacheability attribute. In many cases this is effectively the main system memory, although the architecture does not prohibit the implementation of caches beyond the PoC that have no effect on the coherency between memory system agents.

  Note The presence of system caches can affect the determination of the point of coherency as described in System level caches on page G4-9137.
Point of Unification (PoU)

The PoU for a PE is the point by which the instruction and data caches and the translation table walks of that PE are guaranteed to see the same copy of a memory location. In many cases, the Point of Unification is the point in a uniprocessor memory system by which the instruction and data caches and the translation table walks have merged.

The PoU for an Inner Shareable Shareability domain is the point by which the instruction and data caches and the translation table walks of all the PEs in that Inner Shareable Shareability domain are guaranteed to see the same copy of a memory location. Defining this point permits self-modifying software to ensure future instruction fetches are associated with the modified version of the software by using the standard correctness policy of:

1. Clean data cache entry by address.
2. Invalidate instruction cache entry by address.

The following fields in the CLIDR relate to these conceptual points:

LoC, Level of Coherence

This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Coherency. The LoC value is a cache level, so, for example, if LoC contains the value 3:

- A clean to the Point of Coherency operation requires the level 1, level 2 and level 3 caches to be cleaned.
- Level 4 cache is the first level that does not have to be maintained.

If the LoC field value is $0x0$, this means that no levels of cache need to be cleaned or invalidated when cleaning or invalidating to the Point of Coherency.

If the LoC field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Coherency.

LoUU, Level of Unification, uniprocessor

This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the PE. As with LoC, the LoUU value is a cache level.

If the LoUU field value is $0x0$, this means that no levels of cache need to be cleaned or invalidated when cleaning or invalidating to the Point of Unification.

If the LoUU field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

LoUIS, Level of Unification, Inner Shareable

In any implementation:

- This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable Shareability domain. As with LoC, the LoUIS value is a cache level.
- If the LoUIS field value is $0x0$, this means that no levels of cache need to be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable Shareability domain.
- If the LoUIS field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

For more information, see the CLIDR description.

Abstraction of the cache hierarchy

The following subsections describe the abstraction of the cache hierarchy:

- Cache maintenance instructions that operate by VA on page G4-9123.
- Cache maintenance instructions that operate by set/way on page G4-9123.
Cache maintenance instructions that operate by VA

The VA-based cache maintenance instructions are described as operating by VA. Each of these instructions is always qualified as being either:

- Performed to the Point of Coherency.
- Performed to the Point of Unification.

See Terms used in describing the cache maintenance instructions on page G4-9120 for definitions of Point of Coherency and Point of Unification, and more information about possible meanings of VA.

AArch32 cache and branch predictor maintenance instructions lists the VA-based maintenance instructions.

The CTR holds minimum line length values for:

- The instruction caches.
- The data and unified caches.

These values support efficient invalidation of a range of addresses, because this value is the most efficient address stride to use to apply a sequence of VA-based maintenance instructions to a range of VAs.

For the Invalidate data or unified cache line by VA instruction, the Cache Write-back Granule field of the CTR defines the maximum granule that a single invalidate instruction can invalidate. This meaning of the Cache Write-back Granule is in addition to its defining the maximum size that can be written back.

Cache maintenance instructions that operate by set/way

AArch32 cache and branch predictor maintenance instructions lists the set/way-based maintenance instructions. Some encodings of these instructions include a required field that specifies the cache level for the instruction:

- A clean instruction cleans from the level of cache specified through to at least the next level of cache, moving further from the PE.
- An invalidate instruction invalidates only at the level specified.

AArch32 cache and branch predictor maintenance instructions

The instruction and data cache maintenance instructions have the same functionality in AArch32 state and in AArch64 state. Table G4-3 on page G4-9123 shows the AArch32 System instructions. Instructions that take an argument include Rt in the instruction description.

AArch32 state also provides branch predictor maintenance instructions.

Note

- In Table G4-3 on page G4-9123 the Point of Unification is the Point of Unification of the PE executing the cache maintenance instruction.
- In AArch32 state, all of the maintenance instructions are available from EL1 or higher.
- In AArch64 state, branch predictors are always invisible to software, and therefore AArch64 state does not provide any branch predictor maintenance instructions.

Table G4-3 AArch32 System instructions for cache maintenance

<table>
<thead>
<tr>
<th>Register</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICIALLUIS</td>
<td>Invalidate all to Point of Unification, Inner Shareable</td>
</tr>
<tr>
<td>ICIALLU</td>
<td>Invalidate all to Point of Unification</td>
</tr>
<tr>
<td>ICIMVAU, Rt</td>
<td>Invalidate by virtual address to Point of Unification</td>
</tr>
</tbody>
</table>
A DS8 instruction intended to ensure the completion of cache or branch predictor maintenance instructions must have an access type of both loads and stores.

In an implementation where the branch predictors are architecturally invisible, the BPIMVA, BPIALLIS, and BPIALL instructions can execute as NOPs.

The following subsections give more information about these instructions:

- AArch32 instruction cache maintenance instructions (IC*).
- AArch32 data cache maintenance instructions (DC*) on page G4-9125.
- Branch predictors on page G4-9126.
- General requirements for the scope of cache and branch predictor maintenance instructions on page G4-9127.
- Effects of instructions that operate by VA to the Point of Coherency on page G4-9128.
- Effects of instructions that operate by VA but not to the Point of Coherency on page G4-9128.
- Effects of All and set/way maintenance instructions on page G4-9128.
- Effects of virtualization and security on the AArch32 cache maintenance instructions on page G4-9129.
- Boundary conditions for cache maintenance instructions on page G4-9131.
- Ordering of cache and branch predictor maintenance instructions on page G4-9132.
- Performing cache maintenance instructions on page G4-9133.

### AArch32 instruction cache maintenance instructions (IC*)

Where an address argument for these instructions is required, it takes the form of a 32-bit register that holds the virtual address argument. No alignment restrictions apply for this address.

Any cache maintenance instruction operating by VA includes as part of any required VA to PA translation:

- For an instruction executed at EL1, the current system ASID.
- The current Security state.
- Whether the instruction was performed from Hyp mode, or at EL1.
- For an instruction executed at EL1, the VMID.
That VA to PA translation might fault. However for an instruction cache maintenance instruction that operates by VA:

- It is IMPLEMENTATION DEFINED whether the operation can generate a Data Abort exception for a Translation fault or an Access flag fault.
- The operation cannot generate a Data Abort exception for a Domain fault or a Permission fault, except for the Permission fault case on a Stage 2 fault on a stage 1 translation table walk.

For more information about the possible faults on an instruction that operates by VA, see *Types of MMU faults on page G5-9239*.

An instruction cache maintenance instruction can complete at any time after it is executed, but is only guaranteed to be complete, and its effects visible to other observers, following a DSB instruction executed by the PE that executed the cache maintenance instruction. See also the completion requirements for cache and branch predictor maintenance instructions in *Completion and endpoint ordering on page E2-7176*.

See also *Ordering of cache and branch predictor maintenance instructions on page G4-9132*.

### AArch32 data cache maintenance instructions (DC*)

Data cache maintenance instructions that take a set/way/level argument take a 32-bit register. If a data cache maintenance by set/way instruction specifies a set, way, or level argument that is larger than the value supported by the implementation then the instruction is CONSTRAINED UNPREDICTABLE, see *Out of range values of the Set/Way/Index fields in cache maintenance instructions on page K1-11574* or the instruction description.

DCISW instructions executed at EL1 perform a clean and invalidate, meaning it performs the same maintenance as a DCCISW instruction, if all of the following apply:

- EL2 is implemented and enabled in the current Security state.
- Either:
  - EL2 is using AArch32 and the value of HCR.SWIO is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.SWIO is 1.

Where an address argument for these instructions is required, it takes the form of a 32-bit register that holds the virtual address argument. No alignment restrictions apply for this address.

Any cache maintenance instruction operating by VA includes as part of any required VA to PA translation:

- For an instruction executed at EL1, the current system ASID.
- The current Security state.
- Whether the instruction was performed from Hyp mode, or from EL1.
- For an instruction executed from EL1, the VMID.

That VA to PA translation might fault. However a data or unified cache maintenance instruction that operates by VA cannot generate a Data Abort exception for a Domain fault, and cannot generate a Data Abort exception for a Permission fault, except for the Permission fault case on a Stage 2 fault on a stage 1 translation table walk.

For more information about the possible faults on an instruction that operates by VA, see *Types of MMU faults on page G5-9239*.

DCIMVAC and DCISW instructions executed at EL1 perform a clean and invalidate, meaning they perform the same maintenance as a DCCIMVAC or DCCISW instruction respectively, if all of the following apply:

- EL2 is implemented and enabled in the current Security state.
- PL1&0 stage two address translation is enabled, meaning either:
  - EL2 is using AArch32 and the value of HCR.VM is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.VM is 1.

If a memory fault that sets FAR for the translation regime applicable for the cache maintenance instruction is generated from a data cache maintenance instruction, the FAR holds the address specified in the register argument of the instruction.

See also *Ordering of cache and branch predictor maintenance instructions on page G4-9132*.
Branch predictors

In AArch32 state it is IMPLEMENTATION DEFINED whether branch prediction is architecturally visible. This means that under some circumstances software must perform branch predictor maintenance to avoid incorrect execution caused by out-of-date entries in the branch predictor. For example, to ensure correct operation it might be necessary to invalidate branch predictor entries on a change to instruction memory, or a change of instruction address mapping. For more information, see Specific requirements for branch predictor maintenance instructions.

In an implementation where the branch predictors are architecturally invisible, the branch predictor maintenance instructions can execute as NOPs.

An invalidate all operation on the branch predictor ensures that any location held in the branch predictor has no functional effect on execution. An invalidate branch predictor by VA instruction operates on the address of the branch instruction, but can affect other branch predictor entries.

—  Note  ——

The architecture does not make visible the range of addresses in a branch predictor to which the invalidate operation applies. This means the address used in the invalidate by VA operation must be the address of the branch to be invalidated.

—  Note  ——

If branch prediction is architecturally visible, an instruction cache invalidate all operation also invalidates all branch predictors.

See also Ordering of cache and branch predictor maintenance instructions on page G4-9132.

Specific requirements for branch predictor maintenance instructions

If, for a given translation regime and a given ASID and VMID as appropriate, the instructions at any virtual address change, then branch predictor maintenance instructions must be performed to invalidate entries in the branch predictor, to ensure that the change is visible to subsequent execution. This maintenance is required when writing new values to instruction locations. It can also be required as a result of any of the following situations that change the translation of a virtual address to a physical address, if, as a result of the change to the translation, the instructions at the virtual addresses change:

• For any translation regime other than the Non-secure PL1&0 translation regime, enabling or disabling stage 1 translations.

• For the Non-secure PL1&0 translation regime:
  — When stage 2 translations are enabled, enabling or disabling stage 1 translations unless accompanied by a change of VMID.
  — When stage 2 translations are disabled, enabling or disabling stage 1 translations.
  — Enabling or disabling stage 2 translations.

• Writing new mappings to the translation tables.

• Any change to the TTBR0, TTBR1, or TTBCR registers, unless:
  — For a change to the Secure PL1&0 translation regime, the change is accompanied by a change to the ASID.
  — For a change to the stage 1 translations of the Non-secure PL1&0 translation regime, the change is accompanied by a change to the ASID or a change to the VMID.

• Any change to the VTTBR or VTCR registers, unless accompanied by a change to the VMID.

—  Note  ——

Invalidation is not required if the changes to the translations are such that the instructions associated with the non-faulting translations of a virtual address, for a given translation regime and a given ASID and VMID, as appropriate, remain unchanged throughout the sequence of changes to the translations. Examples of translation changes to which this applies are:
• Changing a valid translation to a translation that generates an MMU fault.
• Changing a translation that generates an MMU fault to a valid translation.

Failure to invalidate entries might give CONSTRAINED UNPREDICTABLE results, caused by the execution of old branches. For more information, see Ordering of cache and branch predictor maintenance instructions on page G4-9132.

Note
• There is no requirement to use the branch predictor maintenance operations to invalidate the branch predictor after:
  — Changing the ContextID or VMID.
  — A cache maintenance instruction that is identified as also flushing the branch predictors, see AArch32 cache and branch predictor maintenance instructions on page G4-9123.

Cache maintenance system instructions on page K17-11905 shows the branch predictor maintenance operations in a VMSA implementation.

**Behavior of the branch predictors at reset**

In AArch32 state:
• If branch predictors are not architecturally invisible:
  — The branch predictors reset to an IMPLEMENTATION DEFINED state that might be UNKNOWN.
  — The branch predictors are disabled at reset.
• An implementation can require the use of a specific branch predictor initialization routine to invalidate the branch predictor storage array before it is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, but the routine must be documented clearly as part of the documentation of the device.
• Arm recommends that whenever an invalidation routine is required, it is based on the AArch32 branch predictor maintenance operations.

Similar rules apply:
• To cache behavior, see Behavior of caches at reset on page G4-9119.
• To TLB behavior, see TLB behavior at reset on page G5-9217.

**General requirements for the scope of cache and branch predictor maintenance instructions**

The specification of the cache maintenance and branch predictor instructions describes what each instruction is guaranteed to do in a system. It does not limit other behaviors that might occur, provided they are consistent with the requirements described in General behavior of the caches on page G4-9113, Behavior of caches at reset on page G4-9119, and Preloading caches on page E2-7191.

This means that as a side-effect of a cache maintenance instruction:
• Any location in the cache might be cleaned.
• Any unlocked location in the cache might be cleaned and invalidated.

As a side-effect of a branch predictor maintenance instruction, any entry in the branch predictor might be invalidated.

Note
Arm recommends that, for best performance, such side-effects are kept to a minimum. Arm strongly recommends that the side-effects of operations performed in Non-secure state do not have a significant performance impact on execution in Secure state.
**Effects of instructions that operate by VA to the Point of Coherency**

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, these instructions must affect the caches of other PEs in the Shareability domain described by the Shareability attributes of the VA supplied with the instruction.

For Device memory and Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions must affect the caches of all PEs in the Outer Shareable Shareability domain of the PE on which the instruction is operating.

In all cases, for any affected PE, these instructions affect all data and unified caches to the Point of Coherency.

**Table G4-4 PEs affected by cache maintenance instructions to the Point of Coherency**

<table>
<thead>
<tr>
<th>Shareability</th>
<th>PEs affected</th>
<th>Effective to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-shareable</td>
<td>The PE performing the operation</td>
<td>The Point of Coherency of the entire system</td>
</tr>
<tr>
<td>Inner Shareable</td>
<td>All PEs in the same Inner Shareable Shareability domain as the PE performing the operation</td>
<td>The Point of Coherency of the entire system</td>
</tr>
<tr>
<td>Outer Shareable</td>
<td>All PEs in the same Outer Shareable Shareability domain as the PE performing the operation</td>
<td>The Point of Coherency of the entire system</td>
</tr>
</tbody>
</table>

**Effects of instructions that operate by VA but not to the Point of Coherency**

The following instruction operate by VA but not to the Point of Coherency:

- Clean data or unified cache line by MVA to the Point of Unification, DCCMVAU.
- Invalidate instruction cache line by MVA to Point of Unification, ICIMVAU.
- Invalidate by MVA from branch predictors, BPIMVA.

For these instructions, Table G4-5 on page G4-9128 shows how, for a VA in a Normal or Device memory location, the Shareability attribute of the VA determines the minimum set of PEs affected, and the point to which the instruction must be effective.

**Table G4-5 PEs affected by cache maintenance instructions to the Point of Unification**

<table>
<thead>
<tr>
<th>Shareability</th>
<th>PEs affected</th>
<th>Effective to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-shareable</td>
<td>The PE executing the instruction</td>
<td>The Point of Unification of instruction cache fills, data cache fills and write-backs, and translation table walks, on the PE executing the instruction</td>
</tr>
<tr>
<td>Inner Shareable</td>
<td>All PEs in the same Inner Shareable Shareability domain as the PE executing the instruction</td>
<td>The Point of Unification of instruction cache fills, data cache fills and write-backs, and translation table walks, of all PEs in the same Inner Shareable Shareability domain as the PE executing the instruction</td>
</tr>
</tbody>
</table>

**Note**

The set of PEs guaranteed to be affected is never greater than the PEs in the Inner Shareable Shareability domain containing the PE executing the instruction.

**Effects of All and set/way maintenance instructions**

The ICIALLU, BPIALL and Dc* set/way instructions apply only to the caches and branch predictors of the PE that performs the instruction. If the branch predictors are architecturally-visible, ICIALLU also performs a BPIALL operation.
The ICIALLUIS and BPIALLIS instructions can affect the caches and branch predictors of all PEs in the same Inner Shareable Shareability domain as the PE that performs the instruction. If the branch predictors are architecturally-visible, ICIALLUIS also performs a BPIALLIS operation. These instructions have an effect to the Point of Unification of instruction cache fills, data cache fills, and write-backs, and translation table walks, of all PEs in the same Inner Shareable Shareability domain.

--- Note ---

The possible presence of system caches, as described in System level caches on page G4-9137, means architecture does not guarantee that all levels of cache can be maintained using set/way instructions.

**Effects of virtualization and security on the AArch32 cache maintenance instructions**

Each Security state has its own physical address space, and therefore cache entries are associated with physical address space. In addition, cache maintenance and branch predictor instructions performed in Non-secure state have to take account of:

- Whether the instruction was performed at EL1 or at EL2.
- For instructions that operate by VA, the current VMID.

Table G4-6 on page G4-9129 shows the effects of virtualization and security on these maintenance instructions.

**Table G4-6 Effects of virtualization and security on the AArch32 cache maintenance instructions**

<table>
<thead>
<tr>
<th>Cache maintenance instructions</th>
<th>Specified entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data or unified cache maintenance instructions</td>
<td></td>
</tr>
</tbody>
</table>

- **Invalidate, Clean, or Clean and Invalidate by VA:** DCIMVAC, DCCMVAC, DCCMVACU, DCCIMVAC
  - All lines that hold the PA that, in the current translation regime, are mapped to by the combination of all of:
    - The specified VA.
    - For an instruction executed at EL1, the current ASID if the location is mapped to by a non-global page.
    - For a Non-secure instruction executed at EL1, the current VMID.
    - For a Non-secure instruction executed at EL2 when EL2 is using AArch64 and HCR_EL2.E2H, TGE is not {1,1}, the current VMID.
    - For a Secure instruction executed at EL1 when EL3 is using AArch64 and SCR_EL3.EEL2 is 1, the current VMID.
    - For a Secure instruction executed at EL0 when EL3 is using AArch64 and SCR_EL3.EEL2 is 1, and HCR_EL2.E2H, TGE is not {1,1}, the current VMID.

- **Invalidate, Clean, or Clean and Invalidate by set/way:** DCISW, DCCSW, DCCISW
  - For a Non-secure instruction, the line specified by set/way provided that the entry comes from the Non-secure PA space.
  - For a Secure instruction, the line specified by set/way regardless of the PA space that the entry has come from.
### Instruction cache maintenance instructions

Invalidates all lines corresponding to the specified VA\(^b\) in the current translation regime and:
- For an instruction executed at EL1 or EL0, the current ASID.
- For a Non-secure instruction executed at EL1 or EL0, the current VMID\(^a\).
- For a Secure instruction executed at EL1 when EL3 is using AArch64 and SCR\_EL3.EEL2 is 1, the current VMID\(^a\).

For a Secure instruction executed at EL0 when EL3 is using AArch64 and SCR\_EL3.EEL2 is 1, and HCR\_EL2.{E2H, TGE} is not \{1,1\}, the current VMID\(^a\).

### Invalidate All: ICIMVAU, ICIMV AUX

Can invalidate any unlocked entry in the instruction cache, and are required to invalidate:
- For a Non-secure instruction executed at EL1, all instruction cache lines containing Non-secure entries associated with the current VMID.
- For a Secure instruction executed at EL1 when EL3 is using AArch64 and SCR\_EL3.EEL2 is 1, all instruction cache lines containing entries associated with the current VMID.
- For a Secure instruction executed at EL0 when EL3 is using AArch64, SCR\_EL3.EEL2 is 1, and HCR\_EL2.{E2H, TGE} is not \{1,1\}, all instruction cache lines.
- For a Secure instruction executed at EL3 all instruction cache lines.

---

### Branch predictor instructions\(^c\)

Invalidates all lines that, in the current translation regime, are mapped to by the combination of all of:
- The specified VA.
- For an instruction executed at EL1 or EL0, the current ASID.
- For a Non-Secure instruction executed at EL1 or EL0, the current VMID\(^a\).
- For a Secure instruction executed at EL1 when EL3 is using AArch64 and SCR\_EL3.EEL2 is 1, the current VMID\(^a\).
- For a Secure instruction executed at EL0 when EL3 is using AArch64, SCR\_EL3.EEL2 is 1, and HCR\_EL2.{E2H, TGE} is not \{1,1\}, the current VMID\(^a\).

Invalidates all lines that, in the current translation regime, are mapped to by the combination of all:
- The specified VA.
- For an instruction executed at EL1 or EL0, the current ASID.
- For a Non-Secure instruction executed at EL1 or EL0, the current VMID\(^a\).
- For a Secure instruction executed at EL1 when EL3 is using AArch64 and SCR\_EL3.EEL2 is 1, the current VMID\(^a\).
- For a Secure instruction executed at EL0 when EL3 is using AArch64, SCR\_EL3.EEL2 is 1, and HCR\_EL2.{E2H, TGE} is not \{1,1\}, the current VMID\(^a\).

Invalidates all lines that, in the current translation regime, are mapped to by the combination of all:
- The specified VA.
- For an instruction executed at EL1 or EL0, the current ASID.
- For a Non-Secure instruction executed at EL1 or EL0, the current VMID\(^a\).
- For a Secure instruction executed at EL1 when EL3 is using AArch64 and SCR\_EL3.EEL2 is 1, the current VMID\(^a\).
- For a Secure instruction executed at EL0 when EL3 is using AArch64, SCR\_EL3.EEL2 is 1, and HCR\_EL2.{E2H, TGE} is not \{1,1\}, the current VMID\(^a\).

### Invalidate all: BPIMVA, BPI MV A

Can invalidate any unlocked entry in the branch predictor, and are required to invalidate:
- For a Non-secure instruction executed at EL1, all lines containing Non-secure entries associated with the current VMID.
- For a Secure instruction executed at EL1 when EL3 is using AArch64 and SCR\_EL3.EEL2 is 1, all lines containing entries associated with the current VMID.
- For a Secure instruction executed at EL0 when EL3 is using AArch64, SCR\_EL3.EEL2 is 1, and HCR\_EL2.{E2H, TGE} is not \{1,1\}, all lines.
- For a Secure instruction executed at EL3 all lines.

---

\(^a\) Dependencies on the VMID apply even when either EL2 is using AArch32 and the value of HCR.VM is 0 or EL2 is using AArch64 when enabled for the current Security state, and the value of HCR\_EL2.VM is 0. If the PE resets into an Exception level that is using AArch32, VTTBR.VMID resets to zero, meaning there is a valid VMID from reset. However, if the PE resets into an Exception level that is using AArch64, VTTBR\_EL2.VMID resets to a value that is architecturally UNKNOWN, and the VTTBR\_EL2.VMID field must be set to a known value, that might be zero, as part of the PE initialization sequence.
b. The type of instruction cache used affects the interpretation of the specified entries in this table such that:
   □ For a PIPT instruction cache, the cache maintenance applies to all entries whose physical address corresponds to the specified address.
   □ For a VIPT instruction cache, the cache maintenance applies to entries whose virtual index and physical tag corresponds to the specified address.

For information of types of instruction cache, see Instruction caches on page G5-9235.

For locked entries and entries that might be locked, the behavior of cache maintenance instructions described in The interaction of cache lockdown with cache maintenance instructions on page G4-9136 applies.

With an implementation that generates aborts if entries are locked or might be locked in the cache, when the use of lockdown aborts is enabled, these aborts can occur on any cache maintenance instructions.

In an implementation that includes EL2:

• The architecture does not require cache cleaning when switching between virtual machines. Cache invalidation by set/way must not present an opportunity for one virtual machine to corrupt state associated with a second virtual machine. To ensure this requirement is met, EL1 invalidate by set/way instructions executed in at EL1 when HCR_EL2.VM or HCR.VM is 1 and EL2 is enabled can, instead, perform a clean and invalidate by set/way.

• The AArch32 Data cache invalidate instructions DCIMVAC and DCISW perform a cache clean as well as a cache invalidate, meaning DCIMVAC performs the same invalidation as a DCCIMVAC instruction, and DCISW performs the same invalidation as a DCCISW instruction, if both of the following apply:
  — EL2 is using AArch32, the value of HCR.VM is 1, and the instruction is executed at Non-secure EL1.
  — EL2 is using AArch64, the value of HCR_EL2.VM is 1, EL2 is enabled, and the instruction is executed at EL1.

• The AArch32 Data cache invalidate by set/way instruction DCISW performs a cache clean as well as a cache invalidate, meaning it performs the same invalidation as a DCCISW instruction, if either of the following apply:
  — EL2 is using AArch32, the value of HCR.SWIO is 1, and the instruction is executed at Non-secure EL1.
  — EL2 is using AArch64, the value of HCR_EL2.SWIO is 1, EL2 is enabled, and the instruction is executed at EL1.

• TLB and instruction cache invalidate instructions are broadcast across the Inner Shareable domain when either:
  — EL2 is using AArch32, the value of HCR.FB is 1, and execution is at Non-secure EL1.
  — EL2 is using AArch64, the value of HCR_EL2.FWB is 1, EL2 is enabled, and the instruction is executed at EL1.

When EL1 is using AArch32, this applies to the TLBIMVA, TLBIA SID, TLBIMVAA, TLBIMVAL, TLBIMVAAL, and ICIALLU instructions. This means the instruction performs the invalidation that would be performed by the corresponding Inner Shareable instruction, for example ICIALLU performs the invalidation that would be performed by ICIALLUIS, and BPIALL performs the invalidation that would be performed by BPIALLIS.

For more information about the cache maintenance instructions, see About cache maintenance in AArch32 state on page G4-9119, AArch32 cache and branch predictor maintenance instructions on page G4-9123, and Chapter G5 The AArch32 Virtual Memory System Architecture.

Boundary conditions for cache maintenance instructions

Cache maintenance instructions operate on the caches regardless of whether the System register Cacheability controls force all memory accesses to be Non-cacheable.

For VA-based cache maintenance instructions, the instructions operate on the caches regardless of the memory type and cacheability attributes marked for the memory address in the VMSA translation table entries. This means that the effects of the cache maintenance instructions can apply regardless of:

• Whether the address accessed:
  — Is Normal memory or Device memory.
  — Has the Cacheable attribute or the Non-cacheable attribute.
• Any applicable domain control of the address accessed.
• The access permissions for the address accessed, other than the effect of the stage two write permission on data or unified cache invalidation instructions.

Ordering of cache and branch predictor maintenance instructions

The following rules describe the effect of the memory order model on the cache and branch predictor maintenance instructions:

• All cache and branch predictor maintenance instructions that do not specify an address execute, relative to each other, in program order.

All cache and branch predictor instructions that specify an address:

— Execute in program order relative to all cache and branch predictor operations that do not specify an address.
— Execute in program order relative to all cache and branch predictor operations that specify the same address.
— Can execute in any order relative to cache and branch predictor operations that specify a different address.

• Where a cache maintenance or branch predictor instruction appears in program order before a change to the translation tables, the architecture guarantees that the cache or branch predictor maintenance instruction uses the translations that were visible before the change to the translation tables.

• Where a change of the translation tables appears in program order before a cache maintenance or branch predictor instruction, software must execute the sequence outlined in Ordering and completion of TLB maintenance instructions on page G5-9223 before performing the cache or branch predictor maintenance instruction, to ensure that the maintenance operation uses the new translations.

• A DMB instruction causes the effect of all data or unified cache maintenance instructions appearing in program order before the DMB to be visible to all explicit memory read and write effects appearing in program order after the DMB.

Also, a DMB instruction ensures that the effects of any data or unified cache maintenance instruction appearing in program order before the DMB are observable by any observer in the same required Shareability domain before any data or unified cache maintenance or explicit memory operations appearing in program order after the DMB are observed by the same observer. Completion of the DMB does not guarantee the visibility of all data to other observers. For example, all data might not be visible to a translation table walk, or to instruction fetches.

• A DSB is required to guarantee the completion of all cache maintenance instruction that appear in program order before the DSB instruction.

• A Context synchronization event is required to guarantee the effects of any branch predictor maintenance operation. This means a Context synchronization event causes the effect of all completed branch predictor maintenance operations appearing in program order before the Context synchronization event to be visible to all instructions after the Context synchronization event. This means that, if a branch instruction appears after an invalidate branch predictor operation and before any Context synchronization event, it is CONSTRAINED UNPREDICTABLE whether the branch instruction is affected by the invalidate. Software must avoid this ordering of instructions, because it might cause CONSTRAINED UNPREDICTABLE behavior.

• Any data or unified cache maintenance instruction by VA must be executed in program order relative to any explicit memory read or write effect on the same PE to an address covered by the VA of the cache instruction if that load or store is to Normal Cacheable memory. The order of memory accesses that result from the cache maintenance instruction, relative to any other memory accesses to Normal Cacheable memory, are subject to the memory ordering rules. For more information, see Definition of the memory model on page E2-7168.

• There is no restriction on the ordering of data or unified cache maintenance instruction by VA relative to any explicit memory read or write effect on the same PE where the address of the explicit memory read or write effect is not covered by the VA of the cache instruction. Where the ordering must be restricted, a DMB instruction must be inserted to enforce ordering.
• There is no restriction on the ordering of a data or unified cache maintenance instruction by set/way relative
to any explicit memory read or write effect on the same PE. Where the ordering must be restricted, a DMB
instruction must be inserted to enforce ordering.
• Software must execute a Context synchronization event after the completion of an instruction cache
maintenance instruction, to guarantee that the effect of the maintenance instruction is visible to any
instruction fetch.

A DSB instruction intended to ensure the completion of cache maintenance instructions or branch predictor
instructions must have an access type of both loads and stores.

See also the completion requirements for cache and branch predictor maintenance instructions in Completion and
endpoint ordering on page E2-7176.

The scope of instruction cache maintenance depends on the type of the instruction cache. For more information, see
Instruction caches on page G5-9235.

**Example G4-1 Cache cleaning operations for self-modifying code**

The sequence of cache cleaning operations for a line of self-modifying code on a uniprocessor system is:

; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Rt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Rn. Use STRH in the first line
; instead of STR for a 16-bit instruction.

STR Rt, [Rn]
DCCMVAU Rn ; Clean data cache by MVA to point of unification (PoU)
DSB
ICIMVAU Rn ; Invalidate instruction cache by MVA to PoU
BPIMVA Rn ; Invalidate branch predictor by MVA to PoU
DSB ; Ensure completion of the invalidations
ISB ; Synchronize the fetched instruction stream

**Performing cache maintenance instructions**

To ensure all cache lines in a block of address space are maintained through all levels of cache Arm strongly
recommends that software:

• For data or unified cache maintenance, uses the CTR.DMinLine value to determine the loop increment size
for a loop of data cache maintenance by VA instructions.
• For instruction cache maintenance, uses the CTR.IMinLine value to determine the loop increment size for a
loop of instruction cache maintenance by VA instructions.

**Example code for cache maintenance instructions**

The cache maintenance instructions by set/way can be used to clean or invalidate, or both, the entirety of one or
more levels of cache attached to a PE. However, unless all PEs attached to the caches regard all memory locations
as Non-cacheable, it is not possible to prevent locations being allocated into the cache during such a sequence of
the cache maintenance instructions.

**Note**

Because the set/way instructions operate only locally, there is no guarantee of the atomicity of cache maintenance
between different PEs, even if those different PEs are each executing the same cache maintenance instructions at
the same time. Because any cacheable line can be allocated into the cache at any time, it is possible for a cache line
to migrate from an entry in the cache of one PE to the cache of a different PE in a way that means the cache line is
not affected by set/way based cache maintenance. Therefore, Arm strongly discourages the use of set/way
instructions to manage coherency in coherent systems. The expected use of the cache maintenance instructions that
operate by set/way is limited to the cache maintenance associated with the powerdown and powerup of caches, if
this is required by the implementation.
The limitations of cache maintenance by set/way mean maintenance by set/way does not happen on multiple PEs, and cannot be made to happen atomically for each address on each PE. Therefore in multiprocessor or multithreaded systems, the use of cache maintenance by set/way to clean, or clean and invalidate, the entire cache for coherency management with very large buffers or with buffers with unknown address can fail to provide the expected coherency results because of speculation by other PEs, or possibly by other threads. The only way that these instructions can be used in this way is to first ensure that all PEs that might cause speculative accesses to caches that need to be maintained are not capable of generating speculative accesses. This can be achieved by ensuring that those PEs have no memory locations with a Normal Cacheable attribute. Such an approach can have very large system performance effects, and Arm advises implementers to use hardware coherency mechanisms in systems where this will be an issue.

System level caches on page G4-9137 refers to other limitations of cache maintenance by set/way.

The following example code for cleaning a data or unified cache to the Point of Coherency illustrates a generic mechanism for cleaning the entire data or unified cache to the Point of Coherency. It assumes the current Cache Size Identification Register is in 32-bit format. For more information, see Possible formats of the Cache Size Identification Registers, CCSIDR and CCSIDR2 on page G4-9115.

```assembly
MRC p15, 1, R0, c0, c0, 1 ; Read CLIDR into R0
ANDS R3, R0, #0x07000000
MOV R3, R3, LSR #23 ; Cache level value (naturally aligned)
BEQ Finished
MOV R10, #0
Loop1
ADD R2, R10, R10, LSR #1 ; Work out 3 x cache level
MOV R1, R0, LSR R2 ; bottom 3 bits are the Cache type for this level
AND R1, R1, #7 ; get those 3 bits alone
CMP R1, #2 ; no cache or only instruction cache at this level
BLT Skip
MCR p15, 2, R10, c0, c0, 0 ; write CSSELR from R10
ISB ; ISB to sync the change to the CCSIDR
MRC p15, 1, R1, c0, c0, 0 ; read current CCSIDR to R1
AND R2, R1, #7 ; extract the line length field
ADD R2, R2, #4 ; add 4 for the line length offset (log2 16 bytes)
MOV R4, #0x3FF
ANDS R4, R4, R1, LSR #3 ; R4 is the max number on the way size (right aligned)
CLZ R5, R4 ; R5 is the bit position of the way size increment
MOV R9, R4 ; R9 working copy of the max way size (right aligned)
Loop2
MOV R7, #0x00007FFF
ANDS R7, R7, R1, LSR #13 ; R7 is the max number of the index size (right aligned)
Loop3
ORR R11, R10, R9, LSL R5 ; factor in the way number and cache number into R11
ORR R11, R11, R7, LSL R2 ; factor in the index number
MCR p15, 0, R11, c7, c10, 2 ; DCCSW, clean by set/way
SUBS R7, R7, #1 ; decrement the index
BGE Loop3
SUBS R9, R9, #1 ; decrement the way number
BGE Loop2
Skip
ADD R10, R10, #2 ; increment the cache number
CMP R3, R10 ; ensure completion of previous cache maintenance instruction
DSB
BGT Loop1
Finished
```

Similar approaches can be used for all cache maintenance instructions.

G4.4.8 Execution and data prediction restriction System instructions

When FEAT_SPECRES is implemented, the System instructions for prediction restriction listed in Table G4-7 on page G4-9135 prevent predictions based on information gathered from earlier execution within a particular execution context (CTX), from affecting the later speculative execution within that CTX, to the extent that the speculation execution is observable through side-channels.

G4-9134
The prediction restriction System instructions being used by a particular CTX apply to:

- All control flow prediction resources that predict execution addresses.
- Data value prediction.
- Cache allocation prediction.

For these System instructions, the CTX is defined by:

- The Security state.
- The Exception level.
- When executing at EL1, the VMID.
- When executing at EL0 when using the PL1&0 translation regime, the ASID and VMID.

Note

- The data value prediction applies to all prediction resources that use some form of training to speculate data values as part of an execution.
- The cache allocation applies to all instruction and data caches, and TLB prefetching hardware used by the executing PE that applies to the supplied context.

The context information is passed as a register argument, and is restricted so that:

- Execution of the System instruction at EL0 applies only to the current hardware defined context.
- Execution of the System instruction at EL1 applies only to the current VMID and Security state, and does not apply to EL2 or EL3.
- Execution of the System instruction at EL2 can apply only to the current Security state, and does not apply to EL3.

If the System instruction is specified to apply to Exception levels that are not implemented, or which are higher than the Exception level that the System instruction is executed at, then the System instruction is treated as a \texttt{NOP}.

When the System instruction is complete and synchronized, no predictions of the restricted type for the affected context are influenced by the execution of the program before the System instruction in a manner that can be observed by the use of any side channels.

Note

- Prediction restriction System instructions do not require the invalidation of prediction structures so long as the behavior described for completion is met by an implementation.
- Prediction restriction System instructions are permitted to invalidate more prediction information than is defined by the supplied CTX.

These System instructions are guaranteed to be complete following a \texttt{DSB} that covers both read and write behavior on the same PE that executed the original instruction. A subsequent Context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

In AArch32 state, EL0 access to the System instructions is controlled by \texttt{SCTL.RnRCTX}.

### Table G4-7 Prediction restriction System instructions

<table>
<thead>
<tr>
<th>Register</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFPRCTX</td>
<td>Control Flow Prediction Restriction by Context</td>
</tr>
<tr>
<td>CPPRCTX</td>
<td>Cache Prefetch Prediction Restriction by Context</td>
</tr>
<tr>
<td>DVPRCTX</td>
<td>Data Value Prediction Restriction by Context</td>
</tr>
</tbody>
</table>
G4.4.9 Cache lockdown

The concept of an entry locked in a cache is allowed, but not architecturally defined. How lockdown is achieved is IMPLEMENTATION DEFINED and might not be supported by:

- An implementation.
- Some memory attributes.

An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.

A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

The interaction of cache lockdown with cache maintenance instructions

The interaction of cache lockdown and cache maintenance instructions is IMPLEMENTATION DEFINED. However, an architecturally-defined cache maintenance instruction on a locked cache line must comply with the following general rules:

- The effect of the following instructions on locked cache entries is IMPLEMENTATION DEFINED:
  - Cache clean by set/way, DCCSW.
  - Cache invalidate by set/way, DCISW.
  - Cache clean and invalidate by set/way, DCISW.
  - Instruction cache invalidate all, ICIALLU and ICIALLUIS.

However, one of the following approaches must be adopted in all these cases:

1. If the instruction specified an invalidation, a locked entry is not invalidated from the cache.
2. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the Fault status code defined for this purpose. See Data Abort exception on page G1-8975.

This permits a usage model for cache invalidate routines to operate on a large range of addresses by performing the required operation on the entire cache, without having to consider whether any cache entries are locked.

The effect of the following instructions is IMPLEMENTATION DEFINED:

- Cache clean by virtual address, DCCMVAC and DCCMVAU.
- Cache invalidate by virtual address, DCIMVAC.
- Cache clean and invalidate by virtual address, DCCIMVAC.

However, one of the following approaches must be adopted in all these cases:

1. If the instruction specified an invalidation, a locked entry is invalidated from the cache. For the clean and invalidate instructions, the entry must be cleaned before it is invalidated.
2. If the instruction specified an invalidation, a locked entry is not invalidated from the cache. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the Fault status code defined for this purpose. See DFSR or HSR.

In an implementation that includes EL2, if HCR.TIDCP is set to 1, any exception relating to lockdown of an entry associated with Non-secure memory is routed to EL2.
Note
An implementation that uses an abort mechanism for entries that can be locked down but are not actually locked down must:

- Document the IMPLEMENTATION DEFINED instruction sequences that perform the required operations on entries that are not locked down.
- Implement one of the other permitted alternatives for the locked entries.

Arm recommends that, when possible, such IMPLEMENTATION DEFINED instruction sequences use architecturally-defined instructions. This minimizes the number of customized instructions required.

In addition, an implementation that uses an abort to handle cache maintenance instructions for entries that might be locked must provide a mechanism that ensures that no entries are locked in the cache.

The reset setting of the cache must be that no cache entries are locked.

Additional cache functions for the implementation of lockdown
An implementation can add additional cache maintenance functions for the handling of lockdown in the IMPLEMENTATION DEFINED space.

G4.4.10 System level caches
The Arm Architecture defines a system cache as a cache that is not described in the PE Cache Identification registers, CCSIDR, CCSIDR2, and CLIDR, and for which the set/way cache maintenance instructions do not apply.

Conceptually, three classes of system cache can be envisaged:

1. System caches which lie before the point of coherency and cannot be managed by cache maintenance instructions. Such systems fundamentally undermine the concept of cache maintenance instructions operating to the point of coherency, as they imply the use of non-architecture mechanisms to manage coherency. The use of such systems in the Arm architecture is explicitly prohibited.

2. System caches which lie before the point of coherency and can be managed by cache maintenance by address instructions that apply to the point of coherency, but cannot be managed by cache maintenance by set/way instructions. Where maintenance of the entire system cache must be performed, as is the case for power management, it must be performed using non-architectural mechanisms.

3. System caches which lie beyond the point of coherency and so are invisible to software. The management of such caches is outside the scope of architecture.
G4.5 System register support for IMPLEMENTATION DEFINED memory features

The VMSA v8-32 defines the following registers for describing IMPLEMENTATION DEFINED features of the memory system:

- The TCM Type Register, TCMTR must be implemented on any implementation where EL1 or above supports AArch32. The format of this register is IMPLEMENTATION DEFINED.
- The System register encoding space with \{\text{coproc}==0b1111, \text{CRn}==c9, \text{CRm}==\{c0-c2, c5-c7\}\} is IMPLEMENTATION DEFINED for all values of opc2 and opc1. This space is reserved for branch predictor, cache and TCM functionality, for example maintenance, override behaviors and lockdown.
- In a VMSA v8-32 implementation, part of the System register encoding space with \{\text{coproc}==0b1111, \text{CRn}==c10\} is IMPLEMENTATION DEFINED and reserved for TLB functionality, see TLB lockdown on page G5-9218.
- The System register encoding space with \{\text{coproc}==0b1111, \text{CRn}==c11, \text{CRm}==\{c0-c8, c15\}\} is IMPLEMENTATION DEFINED for all values of opc2 and opc1. This space is reserved for DMA operations to and from the TCMs.

In addition, the System register encoding space with \{\text{coproc}==0b1111, \text{CRn}==c15\} is reserved for IMPLEMENTATION DEFINED registers, and can provide additional registers for the memory system. For more information, see AArch32 VMSA organization of registers in the (coproc == 0b1111) encoding space on page G7-9304.
G4.6 External aborts

The Arm architecture defines External aborts as errors that occur in the memory system, other than those that are detected by the MMU or Debug hardware. An External abort might signal a data corruption to the PE. For example, a memory location might have been corrupted, and this corruption is detected by hardware using a parity or error correction code (ECC). The error might have been propagated. The RAS Extension provides mechanisms for software to determine the extent of the corruption and contain propagation of the error. For more information, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile.

An External abort is one of:
- Synchronous.
- Precise asynchronous.
- Imprecise asynchronous.

For more information, see Exception terminology on page G1-8900.

The RAS Extension provides an expanded taxonomy for describing aborts. When the RAS Extension is not implemented, the Arm architecture does not provide any method to distinguish between precise asynchronous and imprecise asynchronous External aborts.

VMSAv8-32 permits External aborts on data accesses, translation table walks, and instruction fetches to be either synchronous or asynchronous. The reported fault code identifies whether the External abort is synchronous or asynchronous.

It is IMPLEMENTATION DEFINED which External aborts, if any, are supported. Asynchronous External aborts generate SError interrupt exceptions.

In AArch32 state:
- SError interrupts are taken as asynchronous Data Abort exceptions.
- Synchronous External aborts:
  - On data accesses are taken as synchronous Data Abort exceptions.
  - On instruction fetches, or prefetches, are taken as synchronous Prefetch Abort exceptions.

See also:
- External abort on a translation table walk on page G5-9247.
- Handling exceptions that are taken to an Exception level using AArch32 on page G1-8929.

Normally, External aborts are rare. An imprecise asynchronous External abort is likely to be fatal to the process that is running. Arm recommends that implementations make External aborts precise wherever possible.

The following subsections give more information about possible External aborts:
- Provision for classification of External aborts.
- Parity or ECC error reporting, RAS Extension not implemented on page G4-9140.

The section Exception reporting in a VMSAv8-32 implementation on page G5-9251 describes the reporting of External aborts.

G4.6.1 Provision for classification of External aborts

For an External abort taken to a privileged mode other than Hyp mode, an implementation can use the DFSR.ExT or IFSR.ExT bits to provide more information about the External abort:
- DFSR.ExT provides an IMPLEMENTATION DEFINED classification of External aborts on data accesses.
- IFSR.ExT provides an IMPLEMENTATION DEFINED classification of External aborts on instruction accesses.

For an External abort taken to Hyp mode, the HSR.EA bit, provides an IMPLEMENTATION DEFINED classification of External aborts.

For all aborts other than External aborts these bits return a value of 0.
If the RAS Extension is implemented:

- The HSR.AET field provides information about the state of the PE following an SError interrupt exception taken to Hyp mode.
- The DFSR.AET field provides information about the state of the PE following an asynchronous Data Abort exception.
- The implementation might define error record registers.

For more information on the RAS Extension, see Arm® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile.

**G4.6.2 Parity or ECC error reporting, RAS Extension not implemented**

The Arm architecture supports the reporting of both synchronous and asynchronous parity or ECC errors from the cache systems. It is IMPLEMENTATION DEFINED what parity or ECC errors in the cache systems, if any, result in synchronous or asynchronous parity or ECC errors.

A fault code is defined for reporting parity or ECC errors, see Exception reporting in a VMSAv8-32 implementation on page G5-9251. However when parity or ECC error reporting is implemented it is IMPLEMENTATION DEFINED whether a parity or ECC error is reported using the assigned fault code, or using another appropriate encoding.

For all purposes other than the Fault status encoding, parity or ECC errors are treated as External aborts.
G4.7 Memory barrier instructions

Memory barriers on page E2-7181 describes the memory barrier instructions. This section describes the system level controls of those instructions.

G4.7.1 EL2 control of the Shareability of data barrier instructions executed at EL0 or EL1

In an implementation that includes EL2 and supports Shareability limitations on the data barrier instructions, the HCR.BSU field can modify the required Shareability of an instruction that is executed at EL0 or EL1 in Non-secure state. Table G4-8 on page G4-9141 shows the encoding of this field:

Table G4-8 EL2 control of Shareability of barrier instructions executed at EL0 or EL1

<table>
<thead>
<tr>
<th>HCR.BSU</th>
<th>Minimum Shareability of barrier instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No effect, Shareability is as specified by the instruction</td>
</tr>
<tr>
<td>01</td>
<td>Inner Shareable</td>
</tr>
<tr>
<td>10</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td>11</td>
<td>Full system</td>
</tr>
</tbody>
</table>

For an instruction executed at EL0 or EL1 in Non-secure state, Table G4-9 on page G4-9141 shows how the HCR.BSU is combined with the Shareability specified by the argument of the DMB or DSB instruction to give the scope of the instruction:

Table G4-9 Effect of the HCR_EL2.BSU on barrier instructions executed at Non-secure EL1 or EL1

<table>
<thead>
<tr>
<th>Shareability specified by the DMB or DSB argument</th>
<th>HCR.BSU</th>
<th>Resultant Shareability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full system</td>
<td>Any</td>
<td>Full system</td>
</tr>
<tr>
<td>Outer Shareable</td>
<td>00, 01, or 10</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td></td>
<td>11, Full system</td>
<td>Full system</td>
</tr>
<tr>
<td>Inner Shareable</td>
<td>00 or 01</td>
<td>Inner Shareable</td>
</tr>
<tr>
<td></td>
<td>10, Outer Shareable</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td></td>
<td>11, Full system</td>
<td>Full system</td>
</tr>
<tr>
<td>Non-shareable</td>
<td>00, No effect</td>
<td>Non-shareable</td>
</tr>
<tr>
<td></td>
<td>01, Inner Shareable</td>
<td>Inner Shareable</td>
</tr>
<tr>
<td></td>
<td>10, Outer Shareable</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td></td>
<td>11, Full system</td>
<td>Full system</td>
</tr>
</tbody>
</table>
G4.8 Pseudocode description of general memory System instructions

This section lists the pseudocode describing general memory operations:

- Memory data type definitions.
- Basic memory access.
- Aligned memory access.
- Unaligned memory access.
- Exclusives monitors operations.
- Access permission checking on page G4-9143.
- Abort exceptions on page G4-9144.
- Memory barriers on page G4-9144.

G4.8.1 Memory data type definitions

This section lists the memory data types.

The memory data types are:

- Address descriptor, defined by the AddressDescriptor type.
- Full address, defined by the FullAddress type.
- Memory attributes, defined by the MemoryAttributes type.
- Memory type, defined by the MemType enumeration.
- Device memory type, defined by the DeviceType enumeration.
- Normal memory attributes, defined by the MemAttrHints type.
- Cacheability attributes, defined by the MemAttr_NC, MemAttr_WT, and MemAttr_WB constants.
- Allocation hints, defined by the MemHint_No, MemHint_WA, MemHint_RA, and MemHint_RWA constants.
- Access permissions, defined by the Permissions type.

G4.8.2 Basic memory access

The PhysMemRead() and PhysMemWrite() functions perform single-copy atomic, aligned, little-endian memory accesses of size bytes to or from the underlying physical memory array of bytes.

The attributes in memaddrdesc.memattrs are used by the memory system to determine caching and ordering behaviors as described in Memory types and attributes on page E2-7198, Definition of the memory model on page E2-7168, and Atomicity in the Arm architecture on page E2-7164.

G4.8.3 Aligned memory access

The AArch32.MemSingle[] functions make atomic, little-endian accesses of size bytes.

G4.8.4 Unaligned memory access

See Unaligned data access on page E2-7193 for details of the SCTLR.A and HSCTLR.A controls on the generation of alignment faults. The HSCTLR control applies to Normal memory accesses from Hyp mode, and the SCTLR control applies to Normal memory accesses from all other modes.

The Mem_with_type[] functions make an access of the required type. If that access is naturally aligned, each form of the function performs an atomic access by making a single call to AArch32.MemSingle[]. If that access is not aligned but passes the AArch32.CheckAlignment() checks, each form of the function synthesizes the required access from multiple calls to AArch32.MemSingle[]. It also reverses the byte order if the access is big-endian.

G4.8.5 Exclusives monitors operations

The AArch32.SetExclusiveMonitors() function sets the Exclusives monitors for a Load-Exclusive instruction, for a block of bytes. The size of the blocks is determined by size, at the VA address. The ExclusiveMonitorsPass() function checks whether a Store-Exclusive instruction still has possession of the Exclusives monitors and therefore completes successfully.
The `AArch32.ExclusiveMonitorsPass()` function checks whether a Store-Exclusive instruction still has possession of the Exclusives monitors, by checking whether the Exclusives monitors are set to include the location of the memory block specified by size, at the virtual address defined by address. The atomic write that follows after the Exclusives monitors have been set must be to the same physical address. It is permitted, but not required, for this function to return FALSE if the virtual address is not the same as that used in the previous call to `AArch32.SetExclusiveMonitors()`.

The `ExclusiveMonitorsStatus()` function returns 0 if the previous atomic write was to the same physical memory locations selected by `ExclusiveMonitorsPass()` and therefore succeeded. Otherwise the function returns 1, indicating that the address translation delivered a different physical address.

The `MarkExclusiveGlobal()` procedure takes as arguments a FullAddress, paddress, the PE identifier processorid and the size of the transfer. The procedure records that the PE processorid has requested exclusive access covering at least size bytes from address paddress. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, up to a limit of 2KB and no smaller than two words, and aligned in the address space to the size of the location. It is CONSTRAINED UNPREDICTABLE whether this causes any previous request for exclusive access to any other address by the same PE to be cleared.

The `MarkExclusiveLocal()` procedure takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The procedure records in a local record that PE processorid has requested exclusive access to an address covering at least size bytes from address paddress. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, and can at its largest cover the whole of memory but is no smaller than two words, and is aligned in the address space to the size of the location. It is IMPLEMENTATION DEFINED whether this procedure also performs a `MarkExclusiveGlobal()` using the same parameters.

The `IsExclusiveGlobal()` function takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The function returns TRUE if the PE processorid has marked in a global record an address range as exclusive access requested that covers at least size bytes from address paddress. It is IMPLEMENTATION DEFINED whether it returns TRUE or FALSE if a global record has marked a different address as exclusive access requested. If no address is marked in a global record as exclusive access, `IsExclusiveGlobal()` returns FALSE. It is IMPLEMENTATION DEFINED whether this result is ANDed with the result of `IsExclusiveGlobal()` with the same parameters.

The `IsExclusiveLocal()` function takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The function returns TRUE if the PE processorid has marked an address range as exclusive access requested that covers at least the size bytes from address paddress. It is IMPLEMENTATION DEFINED whether this function returns TRUE or FALSE if the address marked as exclusive access requested does not cover all of size bytes from address paddress. If no address is marked as exclusive access requested, then this function returns FALSE. It is IMPLEMENTATION DEFINED whether this result is ANDed with the result of `IsExclusiveGlobal()` with the same parameters.

The `ClearExclusiveByAddress()` procedure takes as arguments a FullAddress paddress, the PE identifier processorid and the size of the transfer. The procedure clears the global records of all PEs, other than processorid, for which an address region including any of size bytes starting from paddress has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether the equivalent global record of the PE processorid is also cleared if any of size bytes starting from paddress has had a request for an exclusive access, or if any other address has had a request for an exclusive access.

The `ClearExclusiveLocal()` procedure takes as arguments the PE identifier processorid. The procedure clears the local record of PE processorid for which an address has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether this operation also clears the global record of PE processorid that an address has had a request for an exclusive access.

### G4.8.6 Access permission checking

The AArch32.S1LDHasPermissionsFault(), AArch32.S1SDHasPermissionsFault(), and AArch32.S2HasPermissionsFault() functions are used by the architecture to perform access permission checking based on attributes derived from the Translation Table descriptors.

The interpretation of access permission is shown in Memory access control on page G5-9192.
G4.8.7 Abort exceptions

The function `AArch32.Abort()` generates a Data Abort exception or a Prefetch Abort exception by calling the `AArch32.TakeDataAbortException()` or `AArch32.TakePrefetchAbortException()` function.

The `FaultRecord` type describes a fault. Functions that check for faults return a record of this type appropriate to the type of fault. *Pseudocode description of VMSAv8-32 memory system operations on page G5-9278* provides a number of wrappers to generate a `FaultRecord`.

The function `NoFault()` returns a null record that indicates no fault. The `IsFault()` function tests whether a `FaultRecord` contains a fault.

G4.8.8 Memory barriers

The definition for the memory barrier functions is given by the enumerations `MBReqDomain` and `MBReqTypes`.

These enumerations define the required Shareability domains and required access types used as arguments for `DMB` and `DSB` instructions.

The procedures `DataMemoryBarrier()`, `DataSynchronizationBarrier()`, and `InstructionSynchronizationBarrier()` perform the memory barriers.
Chapter G5
The AArch32 Virtual Memory System Architecture

This chapter describes the A-profile AArch32 Virtual Memory System Architecture (VMSA). It includes the following sections:

• About VMSA v8-32 on page G5-9146.
• The effects of disabling address translation stages on VMSA v8-32 behavior on page G5-9154.
• Translation tables on page G5-9158.
• The VMSA v8-32 Short-descriptor translation table format on page G5-9163.
• The VMSA v8-32 Long-descriptor translation table format on page G5-9172.
• Memory access control on page G5-9192.
• Memory region attributes on page G5-9203.
• Translation Lookaside Buffers (TLBs) on page G5-9216.
• TLB maintenance requirements on page G5-9220.
• Caches in VMSA v8-32 on page G5-9235.
• VMSA v8-32 memory aborts on page G5-9238.
• Exception reporting in a VMSA v8-32 implementation on page G5-9251.
• Address translation instructions on page G5-9271.
• Pseudocode description of VMSA v8-32 memory system operations on page G5-9278.
• About the System registers for VMSA v8-32 on page G5-9281.
• Functional grouping of VMSA v8-32 System registers on page G5-9286.

Note
This chapter must be read with Chapter G4 The AArch32 System Level Memory Model.
G5.1 About VMSAv8-32

This chapter describes the A-profile VMSA for AArch32 state, VMSAv8-32.

This chapter describes the control of the VMSA by Exception levels that are using AArch32. Security state, Exception levels, and AArch32 execution privilege on page G1-8908 summarizes how the AArch32 PE modes map onto the Exception levels.

FEAT_SEL2, if implemented, is not available in AArch32 state and EL2 only executes in Non-secure state.

FEAT_S2FWB, if implemented, is not available in AArch32 state. If EL2 is executing in AArch64 state 2 stage translations might be affected. For more informations see Chapter D8 The AArch64 Virtual Memory System Architecture.

Chapter D8 The AArch64 Virtual Memory System Architecture describes the control of the VMSA by Exception levels that are using AArch64.

The main function of the VMSA is to perform address translation, and access permissions and memory attribute determination and checking, for memory accesses made by the PE. Address translation, and permissions and attribute determination and checking, is performed by a stage of address translation.

In VMSAv8-32, the Memory Management Unit (MMU) provides a number of stages of address translation. This chapter describes only the stages that are visible from Exception levels that are using AArch32, which are as follows:

For operation in Secure state
A single stage of address translation, for use when executing at PL1 or EL0. This is the Secure PL1&0 stage 1 address translation stage.

For operation in Non-secure state
  • A single stage of address translation for use when executing at EL2. This is the Non-secure EL2 stage 1 address translation stage.
  • Two stages of address translation for use when executing at PL1 or EL0. These are:
    — The Non-secure PL1&0 stage 1 address translation stage.
    — The Non-secure PL1&0 stage 2 address translation stage.

The System registers provide independent control of each supported stage of address translation, including a control to disable that stage of translation.

However, if the PE is executing at EL0 using AArch32 when EL1 is using AArch64 then it is using the VMSAv8-64 EL1&0 translation regime, described in Chapter D8 The AArch64 Virtual Memory System Architecture.

These features mean the VMSAv8-32 can support a hierarchy of software supervision, for example an Operating System and a hypervisor.

Each stage of address translation uses address translations and associated memory properties held in memory mapped tables called translation tables.

For information about how the MMU features differ if an implementation does not include all of the Exception levels, see About address translation for VMSAv8-32 on page G5-9149.

The translation tables define the following properties:

Access to the Secure or Non-secure address map
The translation table entries determine whether an access from Secure state accesses the Secure or the Non-secure address map. Any access from Non-secure state accesses the Non-secure address map.

Memory access permission control
This controls whether a program is permitted to access a memory region. For instruction and data access, the possible settings are:
  • No access.
  • Read-only.
  • Write-only. This is possible only in a translation regime with two stages of translation.
• Read/write.
  For instruction accesses, additional controls determine whether instructions can be fetched and executed from the memory region.

If a PE attempts an access that is not permitted, a memory fault is signaled to the PE.

Memory region attributes
These describe the properties of a memory region. The top-level attribute, the Memory type, is one of Normal, or a type of Device memory, as follows:

• Both translation table formats support the following Device memory types:
  — Device-nGnRnE
  — Device-nGnRE

• The Long-descriptor translation table format supports, in addition, the following Device memory types:
  — Device-nGRE
  — Device-GRE

  Note
  Armv8 added the Device-nGRE and Device-GRE memory types. Also, in versions of the Arm architecture before Armv8:
  • Device-nGnRnE memory is described as Strongly-ordered memory.
  • Device-nGnRE memory is described as Device memory.

Normal memory regions can have additional attributes.
For more information, see Memory types and attributes on page E2-7198.

Address translation mappings
An address translation maps an input address to an output address.
A stage 1 translation takes the address of an explicit data access or instruction fetch, a virtual address (VA), as the input address, and translates it to a different output address:

• If only one stage of translation is provided, this output address is the physical address (PA).
• If two stages of address translation are provided, the output address of the stage 1 translation is an intermediate physical address (IPA).

  Note
  In the Armv8-32 architecture, a software agent, such as an Operating System, that uses or defines stage 1 memory translations, might be unaware of the distinction between IPA and PA.

A stage 2 translation translates the IPA to a PA.
The possible Security states and privilege levels of memory accesses define a set of translation regimes, where a translation regime maps an input VA to the corresponding PA, using one or two stages of translation. See The VMSAv8-32 translation regimes on page G5-9148.

System registers control VMSAv8-32, including defining the location of the translation tables, and enabling and configuring the MMU, including enabling and disabling the different address translation stages. Also, they report any faults that occur on a memory access. For more information, see Functional grouping of VMSAv8-32 System registers on page G5-9286.

The following sections give an overview of VMSAv8-32, and of the implementation options for VMSAv8-32:

• The VMSAv8-32 translation regimes on page G5-9148.
• Address types used in a VMSAv8-32 description on page G5-9148.
• Address spaces in VMSAv8-32 on page G5-9149.
• About address translation for VMSAv8-32 on page G5-9149.

The remainder of the chapter fully describes the VMSA, including the different implementation options, as summarized in Organization of the remainder of this chapter on page G5-9153.
G5.1.1 The VMSAv8-32 translation regimes

As introduced in Address translation mappings on page G5-9147, a translation regime maps an input VA to the corresponding PA, using one or two stages of translation. Figure G5-1 shows the VMSAv8-32 translation regimes, and their associated translation stages and the Exception levels from which they are controlled.

Translation regimes, for Exception levels that are using AArch32

Secure PL1&0 VA ——— Secure PL1&0 stage 1 ——— PA, Secure or Non-secure

Controlled from Secure PL1 modes†

Non-secure EL2 VA ——— Non-secure EL2 stage 1 ——— PA, Non-secure only

Controlled from Hyp mode†

Non-secure PL1&0 VA ——— Non-secure PL1&0 stage 1 ——— IPA

Controlled from Non-secure PL1 modes†

Non-secure PL1&0 stage 2 ——— PA, Non-secure only

Controlled from Hyp mode†

† Typical control when controlled from an Exception level using AArch32.

Figure G5-1 VMSAv8-32 translation regimes, and associated control

Note

Conceptually, a translation regime that has only a stage 1 address translation is equivalent to a regime with a fixed, flat stage 2 mapping from IPA to PA.

Limited use of Privilege level in AArch32 state on page G1-8909 describes the mapping between the PE modes and the Privilege levels (PLs).

Alternative descriptions of the PL1&0 translation regime

The PL1&0 is described in terms of Privilege level because of the way the AArch32 PE modes map onto the Exception levels, as described in Limited use of Privilege level in AArch32 state on page G1-8909. The description of this translation regime in terms of the Exception levels using depends on the current state of the PE, as follows:

• In Non-secure state, PL1 always maps to EL1, and therefore the Non-secure PL1&0 translation regime could be described as the Non-secure EL1&0 translation regime.

• In Secure state:
  — When EL3 is using AArch32, PL1 maps to EL3, and therefore under these conditions the Secure PL1&0 translation regime could be described as the Secure EL3&0 translation regime,
  — When EL3 is using AArch64, Secure PL1 maps to Secure EL1, and therefore under these conditions the Secure PL1&0 translation regime could be described as the Secure EL1&0 translation regime,

However, these descriptions all refer to the same translation regime, with the same System registers associated with its stage 1 translations. Therefore, the regime is generally referred to as the PL1&0 translation regime.

Note

As Figure G5-1 shows, stage 2 translation is supported only in Non-secure state.

G5.1.2 Address types used in a VMSAv8-32 description

A description of VMSAv8-32 refers to the following address types.
Note

These descriptions relate to a VMSAv8-32 description and therefore sometimes differ from the generic definitions given in the Glossary.

Virtual address (VA)

An address used in an instruction, as a data or instruction address, is a Virtual Address (VA).

An address held in the PC, LR, or SP, is a VA.

The VA map runs from zero to the size of the VA space. For AArch32 state, the maximum VA space is 4GB, giving a maximum VA range of `0x00000000-0xFFFFFFFF`.

Intermediate physical address (IPA)

In a translation regime that provides two stages of address translation, the IPA is the address after the stage 1 translation, and is the input address for the stage 2 translation.

In a translation regime that provides only one stage of address translation, the IPA is identical to the PA.

A VMSAv8-32 implementation provides only one stage of address translation:

- If the implementation does not include EL2.
- When executing in Secure state.
- When executing in Hyp mode.

Physical address (PA)

The address of a location in the Secure or Non-secure memory map. That is, an output address from the PE to the memory system.

G5.1.3 Address spaces in VMSAv8-32

For execution in AArch32 state, the architecture supports:

- A VA space of up to 32 bits. The actual width is IMPLEMENTATION DEFINED.
- An IPA space of up to 40 bits. The translation tables and associated System registers define the width of the implemented address space.

Note

AArch32 defines two translation table formats. The *Long-descriptor* format gives access to the full 40-bit IPA or PA space at a granularity of 4KB. The *Short-descriptor* format:

- Gives access to a 32-bit PA space at 4KB granularity.
- Gives access to a 40-bit PA space, but only at 16MB granularity, by the use of Supersections.

If an implementation includes EL3, the address maps are defined independently for Secure and Non-secure operation, providing two independent 40-bit address spaces, where:

- A VA accessed from Non-secure state can only be translated to the Non-secure address map.
- A VA accessed from Secure state can be translated to either the Secure or the Non-secure address map.

G5.1.4 About address translation for VMSAv8-32

Address translation is the process of mapping one address type to another, for example, mapping VAs to IPAs, or mapping VAs to PAs. A *translation table* defines the mapping from one address type to another, and a *Translation table base register (TTBR)* indicates the start of a translation table. Each implemented stage of address translation shown in Figure G5-1 on page G5-9148 requires its own translation tables.

For PL1&0 stage 1 translations, the mapping can be split between two tables, one controlling the lower part of the VA space, and the other controlling the upper part of the VA space. This can be used, for example, so that:

- One table defines the mapping for operating system and I/O addresses, that do not change on a context switch.
• A second table defines the mapping for application-specific addresses, and therefore might require updating on a context switch.

The VMAv8-32 implementation options determine the supported address translation stages. The following descriptions apply when all implemented Exception levels are using AArch32:

**VMAv8-32 without EL2 or EL3**
- Supports only a single PL1&0 stage 1 address translation. Translation of this stage of address translation can be split between two sets of translation tables, with base addresses defined by TTBR0 and TTBR1, and controlled by TTBCR.

**VMAv8-32 with EL3 but without EL2**
- Supports only the Secure PL1&0 stage 1 address translation and the Non-secure PL1&0 stage 1 address translation. In each Security state, this stage of translation can be split between two sets of translation tables, with base addresses defined by the Secure and Non-secure copies of TTBR0 and TTBR1, and controlled by the Secure and Non-secure copies of TTBCR.

**VMAv8-32 with EL2 but without EL3**
- The implementation supports the following stages of address translation:
  - **Non-secure EL2 stage 1 address translation**
    The HTTBR defines the base address of the translation table for this stage of address translation, controlled by HTCR.
  - **Non-secure PL1&0 stage 1 address translation**
    Translation of this stage of address translation can be split between two sets of translation tables, with base addresses defined by the Non-secure copies of TTBR0 and TTBR1, and controlled by the Non-secure instance of TTBCR.
  - **Non-secure PL1&0 stage 2 address translation**
    The VTTBR defines the base address of the translation table for this stage of address translation, controlled by VTCR.

**VMAv8-32 with EL2 and EL3**
- The implementation supports all of the stages of address translation, as follows:
  - **Secure PL1&0 stage 1 address translation**
    Translation of this stage of address translation can be split between two sets of translation tables, with base addresses defined by the Secure copies of TTBR0 and TTBR1, and controlled by the Secure instance of TTBCR.
  - **Non-secure EL2 stage 1 address translation**
    The HTTBR defines the base address of the translation table for this stage of address translation, controlled by HTCR.
  - **Non-secure PL1&0 stage 1 address translation**
    Translation of this stage of address translation can be split between two sets of translation tables, with base addresses defined by the Non-secure copies of TTBR0 and TTBR1, and controlled by the Non-secure instance of TTBCR.
  - **Non-secure PL1&0 stage 2 address translation**
    The VTTBR defines the base address of the translation table for this stage of address translation, controlled by VTCR.

Figure G5-2 on page G5-9151 shows the translation regimes and stages in a VMAv8-32 implementation that includes all of the Exception levels, and indicates the PE mode that, typically, defines each set of translation tables, if that stage of address translation is controlled by a Privilege level that is using AArch32:
The term *Typically configured* is used in Figure G5-2 to indicate the expected software usage. However, stages of address translation used in AArch32 state can also be configured:

- From an Exception level higher than the Exception level of the configuring PE mode shown in Figure G5-2, regardless of whether that Exception level is using AArch32 or is using AArch64, except that a Non-secure Exception level can never configure a stage of address translation that is used in Secure state.
- From an Exception level that is using AArch64 and is higher than the level at which the translation stage is being used. For example, if Non-secure EL0 is the only Non-secure Exception level that is using AArch32, then the Non-secure PL1&0 stage of address translation is configured from Non-secure EL1, that is using AArch64.

In general:

- The translation from VA to PA can require multiple *stages* of address translation, as Figure G5-2 shows.
- A single stage of address translation takes an *input address* and translates it to an *output address*.

A full translation table lookup is called a *translation table walk*. It is performed automatically by hardware, and can have a significant cost in execution time. To support fine granularity of the VA to PA mapping, a single input address to output address translation can require multiple accesses to the translation tables, with each access giving finer granularity. Each access is described as a *level* of address lookup. The final level of the lookup defines:
  - The required output address.
  - The *attributes* and access permissions of the addressed memory.

*Translation Lookaside Buffers* (TLBs) reduce the average cost of a memory access by caching the results of translation table walks. TLBs behave as caches of the translation table information, and VMSAv8-32 provides TLB maintenance instructions for the management of TLB contents.

The Arm architecture permits TLBs to hold any translation table entry that does not directly cause a Translation fault, an Address size fault, or an Access flag fault.

To reduce the software overhead of TLB maintenance, for the PL1&0 translation regimes VMSAv8-32 distinguishes between *Global pages* and *Process-specific pages*. The ASID identifies pages associated with a specific process and provides a mechanism for changing process-specific tables without having to maintain the TLB structures.

If an implementation includes EL2, the VMID identifies the current virtual machine, with its own independent ASID space. The TLB entries include this VMID information, meaning TLBs do not require explicit invalidation when changing from one virtual machine to another, if the virtual machines have different VMIDs. For stage 2 translations, all translations are associated with the current VMID. There is no mechanism to associate a particular stage 2 translation with multiple virtual machines.
Atomicity of register changes on changing virtual machine

From the viewpoint of software executing at Non-secure PL1 or EL0, when there is a switch from one virtual machine to another, the registers that control or affect address translation must be changed atomically. This applies to the registers for the Non-secure PL1&0 translation regime. This means that all of the following registers must change atomically:

- The registers associated with the stage 1 translations:
  - MAIR0, MAIR1, AMAIR0, and AMAIR1.
  - TTBR0, TTBR1, TTBCR, TTBCR2, and CONTEXTIDR.
  - SCTLR.

- The registers associated with the stage 2 translations:
  - VTTBR and VTCR.
  - HSCTLR.

**Note**

Only some fields of SCTLR affect the stage 1 translation, and only some fields of HSCTLR affect the stage 2 translation. However, in each case, changing these fields requires a write to the register, and that write must be atomic with the other register updates.

These registers apply to execution using the Non-secure PL1&0 translation regime. However, when updated as part of a switch of virtual machines they are updated by software executing at EL2. This means the registers are out of context when they are updated, and no synchronization precautions are required.

Use of out-of-context translation regimes

The architecture requires that:

- When executing at EL3 or EL2, the PE must not use the registers associated with the Non-secure PL1&0 translation regime for speculative memory accesses.
- When executing at EL3 the PE must not use the registers associated with the EL2 translation regime for speculative memory accesses.
- When executing at EL3, EL2, or Non-secure EL1, the PE must not use the registers associated with the Secure PL1&0 translation regime for speculative memory accesses.

If the Statistical Profiling Unit (SPU) is not in use for a lower Exception level when entering an Exception level on completion of a DSB instruction, then no new memory accesses using any translation table entries from a translation regime of an Exception level lower than the Exception level that has been entered will be observed by any observers to the extent that those accesses are required to be observed, as determined by the Shareability and Cacheability of those translation table entries.

If the SPU is in use for a lower Exception level when entering an Exception level on completion of a PSB CSYNC and a subsequent DSB instruction, then no new memory accesses using any translation table entries from a translation regime of an Exception level lower than the Exception level that has been entered will be observed by any observers to the extent that those accesses are required to be observed, as determined by the Shareability and Cacheability of those translation table entries.

**Note**

- This does not require that speculative memory accesses cannot be performed using those entries if it is impossible to tell that those memory accesses have been observed by the observers.
• This requirement does not imply that, on taking an exception to a higher Exception level, any translation table walks started before the exception was taken will be completed by the time the higher Exception level is entered, and therefore memory accesses required for such a translation table walk might, in effect, be performed speculatively. However, the execution of a DSB on entry to the higher Exception level ensures that these accesses are complete.

G5.1.5 Organization of the remainder of this chapter

The remainder of this chapter is organized as follows.

The next part of the chapter describes address translation and the associated memory properties held in the translation table entries, in the following sections:

• The effects of disabling address translation stages on VMSAv8-32 behavior on page G5-9154.
• Translation tables on page G5-9158.
• Secure and Non-secure address spaces on page G5-9161.
• The VMSAv8-32 Short-descriptor translation table format on page G5-9163.
• The VMSAv8-32 Long-descriptor translation table format on page G5-9172.
• Memory access control on page G5-9192.
• Memory region attributes on page G5-9203.
• Translation Lookaside Buffers (TLBs) on page G5-9216.
• TLB maintenance requirements on page G5-9220.

Caches in VMSAv8-32 on page G5-9235 describes VMSAv8-32-specific cache requirements.

The following sections then describe aborts on VMSAv8-32 memory accesses, and how these and other faults are reported:

• VMSAv8-32 memory aborts on page G5-9238.
• Exception reporting in a VMSAv8-32 implementation on page G5-9251.

Address translation instructions on page G5-9271 then describes these operations, and how they relate to address translation.

A number of sections then describe the System registers for VMSAv8-32. The following sections give general information about the System registers, and the organization of the registers in the primary encoding spaces, (coproc==0b1110) and (coproc==0b1111) for these registers:

• About the System registers for VMSAv8-32 on page G5-9281.
• Functional grouping of VMSAv8-32 System registers on page G5-9286.

Note

The System registers in the (coproc==0b1110) encoding space provide the following functionality:

• Self-hosted debug. These registers are described in Debug registers on page G8-9831.
• The System register interface to a trace unit These registers are not described in this manual.
• Jazelle registers. These registers are summarized in Legacy feature registers and system instructions on page K17-11908.

Therefore, there is no summary of these registers by functional groups.

Pseudocode description of VMSAv8-32 memory system operations on page G5-9278 then summarizes the pseudocode functions that describe many features of VMSAv8-32 operation.
G5.2 The effects of disabling address translation stages on VMSAv8-32 behavior

About VMSAv8-32 on page G5-9146 defines the translation regimes and the associated stages of address translation, each of which has its own System registers for control and configuration. VMSAv8-32 includes an enable bit for each stage of address translation, as follows:

- SCTLR.M, in the Secure instance of the register, controls Secure PL1&0 stage 1 address translation.
- SCTLR.M, in the Non-secure instance of the register, controls Non-secure PL1&0 stage 1 address translation.
- HCR.VM controls Non-secure PL1&0 stage 2 address translation.
- HSCTLR.M controls Non-secure EL2 stage 1 address translation.

Note

The descriptions throughout this chapter describe address translation as seen by Exception levels that are using AArch32. However, for the Non-secure PL1&0 translation regime, the stage 2 translation:
- Is controlled by the HCR if EL2 is using AArch32.
- Is controlled by the HCR_EL2 if EL2 is using AArch64.

For this reason, links to the HCR link to a table that disambiguates between the AArch32 HCR and the AArch64 HCR_EL2.

- If EL2 is using AArch64, then the equivalent of the Non-secure EL2 translation regime is described in Chapter D8 The AArch64 Virtual Memory System Architecture, not in this chapter.

The following sections describe the effect on VMSAv8-32 behavior of disabling each stage of translation:

- VMSAv8-32 behavior when stage 1 address translation is disabled.
- VMSAv8-32 behavior when stage 2 address translation is disabled on page G5-9156.
- Behavior of instruction fetches when all associated address translations are disabled on page G5-9156.

Enabling stages of address translation on page G5-9156 gives more information about each stage of address translation, in particular after a reset on an implementation that includes EL3.

G5.2.1 VMSAv8-32 behavior when stage 1 address translation is disabled

When stage 1 address translation is disabled, memory accesses that would otherwise be translated by that stage of address translation are treated as follows:

Non-secure PL1 and EL0 accesses when EL2 is implemented and HCR.DC is set to 1

In an implementation that includes EL2, for an access from a Non-secure PL1 or EL0 mode when HCR.DC is set to 1, the stage 1 translation assigns the Normal Non-shareable, Inner Write-Back Read-Allocate Write-Allocate, Outer Write-Back Read-Allocate Write-Allocate memory attributes.

When FEAT_XS is implemented and HCR.DC is 1, the XS attribute is set to 0 at stage 1 of the translation. Otherwise, the XS attribute is set to 1 at stage 1 of the translation.

See also Effect of the HCR.DC field on page G5-9155.

All other accesses

For all other accesses, when a stage 1 address translation is disabled, the assigned attributes depend on whether the access is a data access or an instruction access, as follows:

Data access

The stage 1 translation assigns the Device-nGnRnE memory type.

Instruction access

The stage 1 translation assigns Normal memory attribute, with the Cacheability and Shareability attributes determined by the value of:
- The Secure instance of SCTLR.M for the Secure PL1&0 translation regime.
• The Non-secure instance of SCTLR.I for the Non-secure PL1&0 translation regime.
• HSCTLR.I for the Non-secure EL2 translation regime.

In these cases, the meaning of the I field is as follows:

**When I is set to 0**

The stage 1 translation assigns the attributes Outer Shareable,
Non-cacheable.

**When I is set to 1**

The stage 1 translation assigns the attributes Inner Write-Through
Read-Allocate No Write-Allocate, Outer Write-Through Read-Allocate
No Write-Allocate Cacheable.

--- **Note** ---

On some implementations, if the SCTLR.TRE field is set to 0 then this behavior can be changed by the remap settings in the memory remap registers. The details of TEX remap when SCTLR.TRE is set to 0 are IMPLEMENTATION DEFINED, see SCTLR.TRE, SCTLR.M, and the effect of the TEX remap registers on page G5-9209.

--- **Note** ---

For this stage of translation, no memory access permission checks are performed, and therefore no MMU Permission faults relating to this stage of translation can be generated.

--- **Note** ---

Alignment checking is performed, and therefore Alignment faults can occur.

For every access, when stage 1 translation is disabled, the output address of the stage 1 translation is equal to the input address. This is called a flat address mapping. If the implementation supports output addresses of more than 32 bits then the output address bits above bit[31] are zero. For example, for a VA to PA translation on an implementation that supports 40-bit PAs, PA[39:32] is 0x00.

For a Non-secure PL1 or EL0 access, if the PL1&0 stage 2 address translation is enabled, the stage 1 memory attribute assignments and output address can be modified by the stage 2 translation.

See also *Behavior of instruction fetches when all associated address translations are disabled* on page G5-9156.

**Effect of the HCR.DC field**

The HCR.DC field determines the default memory attributes assigned for the first stage of the Non-secure PL1&0 translation regime when that stage of translation is disabled.

When executing in a Non-secure PL1 or EL0 mode with HCR.DC set to 1:

• For all purposes other than reading the value of the SCTLR, the PE behaves as if the value of the SCTLR.M field is 0. This means Non-secure PL1&0 stage 1 address translation is disabled.
• For all purposes other than reading the value of the HCR, the PE behaves as if the value of the HCR.VM field is 1. This means Non-secure PL1&0 stage 2 address translation is enabled.

The effect of HCR.DC might be held in TLB entries associated with a particular VMID. Therefore, if software executing at EL2 changes the HCR.DC value without also changing the current VMID, it must also invalidate all TLB entries associated with the current VMID. Otherwise, the behavior of Non-secure software executing at EL1 or EL0 is CONSTRAINED UNPREDICTABLE, see *CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values* on page K1-11567.

**Effect of disabling translation on maintenance and address translation instructions**

Cache maintenance instructions act on the target cache whether address translation is enabled or not, and regardless of the values of the memory attributes. However, if a stage of translation is disabled, they use the flat address mapping for that stage, and all mappings are considered global.

TLB invalidate operations act on the target TLB whether address translation is enabled or not.
When the Non-secure PL1&0 stage 1 address translation is disabled, any ATS1C** or ATS12NSO** address translation instruction that accesses the Non-secure state translation reflects the effect of the HCR.DC field.

**G5.2.2 VMSAv8-32 behavior when stage 2 address translation is disabled**

When stage 2 address translation is disabled:

- The IPA output from the stage 1 translation maps flat to the PA
- The memory attributes and permissions from the stage 1 translation apply to the PA.

If the stage 1 address translation and the stage 2 address translation are both disabled, see *Behavior of instruction fetches when all associated address translations are disabled*.

**G5.2.3 Behavior of instruction fetches when all associated address translations are disabled**

The information in this section applies to memory accesses:

- From Secure PL1 and EL0 modes, when the Secure PL1&0 stage 1 address translation is disabled
- From Hyp mode, when the Non-secure EL2 stage 1 address translation is disabled
- From Non-secure PL1 and EL0 modes, when all of the following apply:
  - The Non-secure PL1&0 stage 1 address translation is disabled.
  - The Non-secure PL1&0 stage 2 address translation is disabled.
  - HCR.DC is set to 0.

In these cases, when execution is in AArch32 state a memory location might be accessed as a result of an instruction fetch if either:

- The memory location is in the same 4KB block of memory, aligned to 4KB, as an instruction which a simple sequential execution of the program either requires to be fetched now or has required to be fetched since the last reset, or is in the 4KB block immediately following such a block.
- The memory location is the target of a direct branch that a simple sequential execution of the program would have taken since the most recent of:
  - The last reset.
  - If the branch predictor is architecturally invisible, the last synchronization of instruction cache maintenance targeting the address of the branch instruction.
  - If the branch predictor is not architecturally invisible, the last synchronization of branch predictor maintenance targeting the address of the branch instruction.

These accesses can be caused by speculative instruction fetches, regardless of whether the prefetched instruction is committed for execution.

--- **Note**

To ensure architectural compliance, software must ensure that both of the following apply:

- Instructions that will be executed when address translation is disabled are located in 4KB blocks of the address space that contain only memory that is tolerant to speculative accesses.
- Each 4KB block of the address space that immediately follows a 4KB block that holds instructions that will be executed when address translation is disabled also contains only memory that is tolerant to speculative accesses.

**G5.2.4 Enabling stages of address translation**

On powerup or Warm reset, only the SCTLRM field for the Exception level and Security state entered on reset is reset to 0, disabling address translation for the initial state of the PE. All other SCTLRM and HSCTLRM fields that are implemented are UNKNOWN after the reset.
This means, on powerup or reset:

- On an implementation that includes EL3, where EL3 is using AArch32:
  - The PL1&0 stage 1 address translation enable bit, SCTLR.M, is banked, meaning there are separate enables for operation in Secure and Non-secure state.
  - If EL3 is using AArch32, only the Secure instance of the SCTLR.M field resets to 0, disabling the Secure state PL1&0 stage 1 address translation. The reset value of the Non-secure instance of SCTLR.M is UNKNOWN.

- On an implementation that includes EL2, where EL2 is using AArch32, the HSCTLR.M field, that controls the Non-secure EL2 stage 1 address translation:
  - If the implementation does not include EL3, resets to 0.
  - Otherwise, is UNKNOWN.

- On an implementation that does not include either EL2 or EL3, there is a single stage of translation. This is controlled by SCTLR.M, that resets to 0.

Note

If, for the software that enables or disables a stage of address translation, the input address of a stage 1 translation differs from the output address of that stage 1 translation, and the software is running in translation regime that is affected by that stage of translation, then the requirement to synchronize changes to the System registers means it is uncertain where in the instruction stream the change of the translation takes place. For this reason, Arm strongly recommends that the input address and the output address are identical in this situation.
G5.3 Translation tables

VMSAv8-32 defines two alternative translation table formats:

Short-descriptor format
It uses 32-bit descriptor entries in the translation tables, and provides:
- Up to two levels of address lookup.
- 32-bit input addresses.
- Output addresses of up to 40 bits.
- Support for PAs of more than 32 bits by use of supersections, with 16MB granularity.
- Support for No access, Client, and Manager domains.

Long-descriptor format
It uses 64-bit descriptor entries in the translation tables, and provides:
- Up to three levels of address lookup.
- Input addresses of up to 40 bits, when used for stage 2 translations.
- Output addresses of up to 40 bits.
- 4KB assignment granularity across the entire PA range.
- No support for domains, all memory regions are treated as in a Client domain.
- Fixed 4KB table size, unless truncated by the size of the input address space.

Note
- Translation with a 40-bit input address range requires two concatenated 4KB top-level tables, aligned to 8KB.
- The VMSAv8-64 Long-descriptor translation table format is generally similar to this format, but supports input and output addresses of up to 48 bits, and has an assignment granularity and table size defined by its translation granule. This can be 4KB, 16KB, or 64KB. See Translation table descriptor formats on page D8-5123.

In all implementations, of the possible address translations shown in Figure G5-2 on page G5-9151, for stages of address translation that are using AArch32:
- In a particular Security state, the translation tables for the PL1&0 stage 1 translations can use either translation table format, and the TTBCR.EAE field indicates the current translation table format.
- The translation tables for the Non-secure EL2 stage 1 translations, and for the Non-secure PL1&0 stage 2 translations, must use the Long-descriptor translation table format.

Many aspects of performing a translation table walk depend on the current translation table format. Therefore, the following sections describe the two formats, including how the MMU performs a translation table walk for each format:
- The VMSAv8-32 Short-descriptor translation table format on page G5-9163.
- The VMSAv8-32 Long-descriptor translation table format on page G5-9172.

The following subsections describe aspects of the translation tables and translation table walks, for memory accesses from AArch32 state, that are independent of the translation table format:
- Translation table walks for memory accesses using VMSAv8-32 translation regimes on page G5-9159.
- Information returned by a translation table lookup on page G5-9159.
- Determining the translation table base address in the VMSAv8-32 translation regimes on page G5-9160.
- Control of translation table walks on a TLB miss on page G5-9161.
- Access to the Secure or Non-secure PA map on page G5-9161.

See also TLB maintenance requirements on page G5-9220.
G5.3.1 Translation table walks for memory accesses using VMSAv8-32 translation regimes

A translation table walk occurs as the result of a TLB miss, and starts with a read of the appropriate starting-level translation table. The result of that read determines whether additional translation table reads are required, for this stage of translation, as described in either:

- Translation table walks, when using the VMSAv8-32 Short-descriptor translation table format on page G5-9169.
- Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format on page G5-9187.

Note

When using the Short-descriptor translation table format, the starting level for a translation table walk is always a level 1 lookup. However, with the Long-descriptor translation table format, the starting-level can be either a level 1 or a level 2 lookup.

For the PL1&0 stage 1 translations, SCTLR.EE determines the endianness of the translation table lookups. SCTLR is banked, and therefore the endianness is determined independently for each Security state.

HSCTLR.EE defines the endianness for the Non-secure EL2 stage 1 and Non-secure PL1&0 stage 2 translations.

Note

Dynamically changing translation table endianness

Because any change to SCTLR.EE or HSCTLR.EE requires synchronization before it is visible to subsequent operations, Arm strongly recommends that:

- SCTLR.EE is changed only when either:
  - Executing in a mode that does not use the translation tables affected by SCTLR.EE.
  - Executing with SCTLR.M set to 0.
- HSCTLR.EE is changed only when either:
  - Executing in a mode that does not use the translation tables affected by HSCTLR.EE.
  - Executing with HSCTLR.M set to 0.

The PA of the base of the starting-level translation table is determined from the appropriate TTBR, see Determining the translation table base address in the VMSAv8-32 translation regimes on page G5-9160.

For more information, see Ordering and completion of TLB maintenance instructions on page G5-9223.

Translation table walks must access data or unified caches, or data and unified caches, of other agents participating in the coherency protocol, according to the Shareability attributes described in the TTBR. These Shareability attributes must be consistent with the Shareability attributes for the translation tables themselves.

G5.3.2 Information returned by a translation table lookup

When an associated stage of address translation is enabled, a memory access requires one or more translation table lookups. If the required Translation Table descriptor is not held in a TLB, a translation table walk is performed to obtain the descriptor. A lookup, whether from the TLB or as the result of a translation table walk, returns both:

- An output address that corresponds to the input address for the lookup.
- A set of properties that correspond to that output address.

The returned properties are classified as providing address map control, access controls, or region attributes. This classification determines how the descriptions of the properties are grouped. The classification is based on the following model:

Address map control

Memory accesses from Secure state can access either the Secure or the Non-secure address map, as summarized in Access to the Secure or Non-secure PA map on page G5-9161.

Memory accesses from Non-secure state can only access the Non-secure address map.
Access controls
Determine whether the PE, in its current state, can access the output address that corresponds to the given input address. If not, an MMU fault is generated and there is no memory access.

Memory access control on page G5-9192 describes the properties in this group.

Attributes
Are valid only for an output address that the PE, in its current state, can access. The attributes define aspects of the required behavior of accesses to the target memory region.

Memory region attributes on page G5-9203 describes the properties in this group.

G5.3.3 Determining the translation table base address in the VMSAv8-32 translation regimes

On a TLB miss, the VMSA must perform a translation table walk, and therefore must find the base address of the translation table to use for its lookup. A TTBR holds this address. As Figure G5-2 on page G5-9151 shows:

- For a Non-secure EL2 stage 1 translation, the HTTBR holds the required base address. The HTCR is the control register for these translations.
- For a Non-secure PL1&0 stage 2 translation, the VTTBR holds the required base address. The VTCR is the control register for these translations.
- For a PL1&0 stage 1 translation, either TTBR0 or TTBR1 holds the required base address. The TTBCR is the control register for these translations.

The Non-secure copies of TTBR0, TTBR1, and TTBCR, relate to the Non-secure PL1&0 stage 1 translation. The Secure copies of TTBR0, TTBR1, and TTBCR, relate to the Secure PL1&0 stage 1 translation.

For the PL1&0 translation table walks:

- TTBR0 can be configured to describe the translation of VAs in the entire address map, or to describe only the translation of VAs in the lower part of the address map.
- If TTBR0 is configured to describe the translation of VAs in the lower part of the address map, TTBR1 is configured to describe the translation of VAs in the upper part of the address map.

The contents of the appropriate instance of the TTBCR determine whether the address map is separated into two parts, and where the separation occurs. The details of the separation depend on the current translation table format, see:

- Selecting between TTBR0 and TTBR1, VMSAv8-32 Short-descriptor translation table format on page G5-9168.
- Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format on page G5-9181.

Example G5-1 shows a typical use of the two sets of translation tables:

Example G5-1 Example use of TTBR0 and TTBR1

An example of using the two TTBRs for PL1&0 stage 1 address translations is:

**TTBR0**
Used for process-specific addresses.
- Each process maintains a separate level 1 translation table. On a context switch:
  - TTBR0 is updated to point to the level 1 translation table for the new context.
  - TTBCR is updated if this change changes the size of the translation table.
  - The CONTEXTIDR is updated.
- TTBCR can be programmed so that all translations use TTBR0 in a manner compatible with architecture versions before Armv6.

**TTBR1**
Used for operating system and I/O addresses, that do not change on a context switch.
G5.3.4 Control of translation table walks on a TLB miss

Two fields in the TCR for the translation stage required by a memory access control whether a translation table walk is performed on a TLB miss. These two fields are the:

- PD0 and PD1 fields, on a PE using the Short-descriptor translation table format.
- EPD0 and EPD1 fields, on a PE using the Long-descriptor translation table format.

Note

For the VMSAv8-32 translation regimes, the different field names are because the fields are in different positions in TTBCR, depending on the translation table format.

The effect of these fields is:

{E}PDx == 0  If a TLB miss occurs based on TTBRx, a translation table walk is performed. The current Security state determines whether the memory access is Secure or Non-secure.

{E}PDx == 1  If a TLB miss occurs based on TTBRx, a level 1 Translation fault is returned, and no translation table walk is performed.

G5.3.5 Access to the Secure or Non-secure PA map

As stated in Address spaces in VMSAv8-32 on page G5-9149, a PE can access independent Secure and Non-secure address maps. When the PL1 Exception level is using AArch32, these are defined by the translation tables identified by the Secure TTBR0 and TTBR1. In both translation table formats in the Secure translation tables, the NS field in a descriptor indicates whether the descriptor refers to the Secure or the Non-secure address map:

NS == 0  Access the Secure PA space.

NS == 1  Access the Non-secure PA space.

Note

In the Non-secure translation tables, the corresponding field is SBZ. Non-secure accesses always access the Non-secure PA space, regardless of the value of this field.

The Long-descriptor translation table format extends this control, adding an NSTable field to the Secure translation tables, as described in Hierarchical control of Secure or Non-secure memory accesses, Long-descriptor format on page G5-9180. In the Non-secure translation tables, the corresponding field is SBZ, and Non-secure accesses ignore the value of this field.

The following sections describe the address map controls in the two implementations:

- Control of Secure or Non-secure memory access, VMSAv8-32 Short-descriptor format on page G5-9168.
- Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format on page G5-9180.

The following subsection gives more information.

Secure and Non-secure address spaces

EL3 provides two PA spaces, a Secure PA space and a Non-secure PA space.

As described in Access to the Secure or Non-secure PA map, for the PL1&0 stage 1 translations when controlled from an Exception level using AArch32, the registers that control the stage of translation, TTBR0, TTBR1, TTBCR, and TTBCR2 are banked to provide independent Secure and Non-secure instances of the registers, and the Security state of the PE when it performs a memory access whether the Secure or Non-secure instances are used. This means that for stage 1 of the PL1&0 translation regime there are independent Secure and Non-secure translation tables, and translation table walks are made to the PA space corresponding to the Security state of the translation tables used.

For a translation table walk caused by a memory access from Non-secure state, all memory accesses are to the Non-secure address space.
For a translation table walk caused by a memory access from Secure state:

- When address translation is using the Long-descriptor translation table format:
  - The initial lookup performed must access the Secure address space.
  - If a Table descriptor read from the Secure address space has the NSTable field set to 0, then the next level of lookup is from the Secure address space.
  - If a Table descriptor read from the Secure address space has the NSTable field set to 1, then the next level of lookup, and any subsequent level of lookup, is from the Non-secure address space.

  For more information, see Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format on page G5-9180.

- Otherwise, all memory accesses are to the Secure address space.

--- Note ---

- When executing in Non-secure state, additional translations are supported. For memory accesses from AArch32 state, these are:
  - Non-secure EL2 stage 1 translation.
  - Non-secure PL1&0 stage 2 translation.

  These translations can access only the Non-secure address space.

- A system implementation can alias parts of the Secure PA space to the Non-secure PA space in an implementation-specific way. As with any other aliasing of physical memory, the use of aliases in this way can require the use of cache maintenance instructions to ensure that changes to memory made using one alias of the physical memory are visible to accesses to the other alias of the physical memory.
G5.4 The VMSAv8-32 Short-descriptor translation table format

The Short-descriptor translation table format supports a memory map based on memory sections or pages:

**Supersections** Consist of 16MB blocks of memory. Support for Supersections is optional, except that an implementation that supports more than 32 bits of PA must also support Supersections to provide access to the entire PA space.

**Sections** Consist of 1MB blocks of memory.

**Large pages** Consist of 64KB blocks of memory.

**Small pages** Consist of 4KB blocks of memory.

Supersections, Sections, and Large pages map large regions of memory using only a single TLB entry.

--- Note ---

- Whether a VMSAv8-32 implementation of the Short-descriptor format translation tables supports supersections is IMPLEMENTATION DEFINED.
- The EL2 translation regime cannot use the Short-descriptor translation table format.

When using the Short-descriptor translation table format, two levels of translation tables are held in memory:

**Level 1 table**

Holds *level 1 descriptors* that contain the base address and
- Translation properties for a Section and Supersection.
- Translation properties and pointers to a level 2 table for a Large page or a Small page.

**Level 2 tables**

Hold *level 2 descriptors* that contain the base address and translation properties for a Small page or a Large page. With the Short-descriptor format, level 2 tables can be referred to as translation tables.

A level 2 table requires 1KB of memory.

In the translation tables, in general, a descriptor is one of:

- An invalid or fault entry.
- A translation table entry, that points to a next-level translation table.
- A page or section entry, that defines the memory properties for the access.
- A reserved format.

Bits\[1:0\] of the descriptor give the primary indication of the descriptor type.

Figure G5-3 on page G5-9164 gives a general view of address translation when using the Short-descriptor translation table format.
G5.4 The VMSAv8-32 Short-descriptor translation table format

Additional requirements for Short-descriptor format translation tables on page G5-9167 describes why, when using the Short-descriptor format, Supersection and Large page entries must be repeated 16 times, as shown in Figure G5-3.

VMSAv8-32 Short-descriptor Translation Table format descriptors, Memory attributes in the VMSAv8-32 Short-descriptor Translation Table format descriptors on page G5-9167, and Control of Secure or Non-secure memory access, VMSAv8-32 Short-descriptor format on page G5-9168 describe the format of the descriptors in the Short-descriptor format translation tables.

The following sections then describe the use of this translation table format:

- Selecting between TTBR0 and TTBR1, VMSAv8-32 Short-descriptor translation table format on page G5-9168.
- Translation table walks, when using the VMSAv8-32 Short-descriptor translation table format on page G5-9169.

G5.4.1 VMSAv8-32 Short-descriptor Translation Table format descriptors

The following sections describe the formats of the entries in the Short-descriptor Translation Tables:

- Short-descriptor Translation Table level 1 descriptor formats.
- Short-descriptor Translation Table level 2 descriptor formats on page G5-9166.

For more information about level 2 translation tables, see Additional requirements for Short-descriptor format translation tables on page G5-9167.

Note


Information returned by a translation table lookup on page G5-9159 describes the classification of the non-address fields in the descriptors as address map control, access control, or attribute fields.

Short-descriptor Translation Table level 1 descriptor formats

Each entry in the level 1 table describes the mapping of the associated 1MB VA range.

Figure G5-4 on page G5-9165 shows the possible level 1 descriptor formats.
Descriptor bits[1:0] identify the descriptor type. The encoding of these bits is:

0b00, **Invalid entry**

The associated VA is unmapped, and any attempt to access it generates a Translation fault. Bits[31:2] of the descriptor are ignored, see **IGNORED** on page Glossary-11937. This means software can use these bits for its own purposes.

0b01, **Translation table**

The descriptor gives the address of a level 2 translation table, that specifies the mapping of the associated 1MByte VA range.

0b10, **Section or Supersection**

The descriptor gives the base address of the Section or Supersection. Bit[18] determines whether the entry describes a Section or a Supersection.

This encoding also defines the PXN field as 0.

0b11, **Section or Supersection, if the implementation supports the PXN attribute**

This encoding is identical to 0b10, except that it defines the PXN field as 1.

**Note**

A VMSAv8-32 implementation can use the Short-descriptor translation table format for the PL1&0 stage 1 translations, by setting TTBCR.EAE to 0.

The address information in the level 1 descriptors is:

**Translation table** Bits[31:10] of the descriptor are bits[31:10] of the address of a translation table.

**Section** Bits[31:20] of the descriptor are bits[31:20] of the address of the Section.
Supersection  


For the Non-secure PL1&0 translation tables, the address in the descriptor is the IPA of the translation table, Section, or Supersection. Otherwise, the address is the PA of the translation table, Section, or Supersection.

For descriptions of the other fields in the descriptors, see Memory attributes in the VMSAv8-32 Short-descriptor Translation Table format descriptors on page G5-9167.

Short-descriptor Translation Table level 2 descriptor formats

Figure G5-5 shows the possible formats of a level 2 descriptor.

Descriptor bits[1:0] identify the descriptor type. The encoding of these bits is:

- **0b00, Invalid entry**
  
The associated VA is unmapped, and attempting to access it generates a Translation fault.

  Bits[31:2] of the descriptor are IGNORED, see IGNORED on page Glossary-11937. This means software can use these bits for its own purposes.

- **0b01, Large page**
  
The descriptor gives the base address and properties of the Large page.

- **0b1x, Small page**
  
The descriptor gives the base address and properties of the Small page.

  In this descriptor format, bit[0] of the descriptor is the XN field.

The address information in the level 2 descriptors is:

- **Large page**  
  Bits[31:16] of the descriptor are bits[31:16] of the address of the Large page.

- **Small page**  
  Bits[31:12] of the descriptor are bits[31:12] of the address of the Small page.

For the Non-secure PL1&0 translation tables, the address in the descriptor is the IPA of the translation table, Section, or Supersection. Otherwise, the address is the PA of the translation table, Section, or Supersection.

For descriptions of the other fields in the descriptors, see Memory attributes in the VMSAv8-32 Short-descriptor Translation Table format descriptors on page G5-9167.
Additional requirements for Short-descriptor format translation tables

When using Supersection or Large Page descriptors in the Short-descriptor translation table format, the input address field that defines the Supersection or Large Page descriptor address overlaps the table address field. In each case, the size of the overlap is 4 bits. The following diagrams show these overlaps:

- Figure K9-14 on page K9-11727 for the level 1 translation table entry for a Supersection.
- Figure K9-16 on page K9-11729 for the level 2 translation table entry for a Large page.

Considering the case of using Large Page descriptors in a level 2 translation table, this overlap means that for any specific Large page, the bottom four bits of the level 2 translation table entry might take any value from 0b0000 to 0b1111. Therefore, each of these 16 index values must point to a separate copy of the same descriptor.

This means that each Large page or Supersection descriptor must:

- Occur first on a sixteen-word boundary.
- Be repeated in 16 consecutive memory locations.

G5.4.2 Memory attributes in the VMSAv8-32 Short-descriptor Translation Table format descriptors

This section describes the descriptor fields other than the descriptor type field and the address field:

TEX[2:0], C, B
Memory region attribute fields, see Memory region attributes on page G5-9203. These fields are not present in a descriptor for a translation table.

XN bit
The Execute-never field, see Access permissions for instruction execution on page G5-9196. This bit is not present in a descriptor for a translation table.

PXN bit
The Privileged execute-never field, see Access permissions for instruction execution on page G5-9196.

When this field is set to 1 in the descriptor for a translation table, it indicates that all memory pages described in the corresponding translation table are Privileged execute-never.

NS bit
Non-secure bit. Specifies whether the translated PA is in the Secure or Non-secure address map, see Control of Secure or Non-secure memory access, VMSAv8-32 Short-descriptor format on page G5-9168.

This bit is not present in level 2 descriptors. The value of the NS bit in a level 1 descriptor for a translation table applies to all entries in the corresponding level 2 translation table.

Domain
Domain field, see Domains, Short-descriptor format only on page G5-9199.

This field is not present in a Supersection entry. Memory described by Supersections is in domain 0. This bit is not present in level 2 descriptors. The value of the Domain field in the level 1 descriptor for a translation table applies to all entries in the corresponding level 2 translation table.

An IMPLEMENTATION DEFINED bit
This bit is not present in level 2 descriptors.

AP[2], AP[1:0]
Access Permissions bits, see Memory access control on page G5-9192.

AP[0] can be configured as the Access flag, see The Access flag on page G5-9200. These bits are not present in a descriptor for a translation table.

S bit
Shareable bit. Used in determining the Shareability of the addressed region, see Memory region attributes on page G5-9203.

Note
The naming of this bit as the Shareable bit is carried forward from early versions of the Arm architecture. This name is no longer an adequate description of the interpretation of the bit.
This bit is not present in a descriptor for a translation table.

**nG bit**  
The not global bit. If a lookup using this descriptor is cached in a TLB, determines whether the TLB entry applies to all ASID values, or only to the current ASID value. See Global and process-specific translation table entries on page G5-9216.

This bit is not present in a descriptor for a translation table.

**Bit[18], when bits[1:0] indicate a Section or Supersection descriptor**

- 0  Descriptor is for a Section.
- 1  Descriptor is for a Supersection.

### G5.4.3 Control of Secure or Non-secure memory access, VMSAv8-32 Short-descriptor format

*Access to the Secure or Non-secure PA map on page G5-9161* describes how the NS bit in the translation table entries:

- For accesses from Secure state, determines whether the access is to Secure or Non-secure memory.
- Is ignored by accesses from Non-secure state.

In the Short-descriptor translation table format, the NS bit is defined only in the level 1 translation tables. This means that, in a level 1 descriptor for a translation table, the NS bit defines the PA map, Secure or Non-secure, for all of the Large pages and Small pages of memory described by that table.

The NS bit of a level 1 descriptor for a translation table has no effect on the PA map in which that translation table is held. As stated in Secure and Non-secure address spaces on page G5-9161, the PA of that translation table is in:

- The Secure address map if the translation table walk is in Secure state.
- The Non-secure address map if the translation table walk is in Non-secure state.

This means the granularity of the Secure and Non-secure memory maps is 1MB. However, in these memory maps, table entries can define physical memory regions with a granularity of 4KB.

### G5.4.4 Selecting between TTBR0 and TTBR1, VMSA v8-32 Short-descriptor translation table format

As described in Determining the translation table base address in the VMSA v8-32 translation regimes on page G5-9160, two sets of translation tables can be defined for each of the PL1&0 stage 1 translations, and TTBR0 and TTBR1 hold the base addresses for the two sets of tables. When using the Short-descriptor translation table format, the value of TTBCR.N indicates the number of most significant bits of the input VA that determine whether TTBR0 or TTBR1 holds the required translation table base address, as follows:

- If N == 0 then use TTBR0. Setting TTBCR.N to zero disables use of a second set of translation tables.
- If N > 0 then:
  - If bits[31:32-N] of the input VA are all zero, then use TTBR0.
  - Otherwise use TTBR1.

Table G5-1 on page G5-9168 shows how the value of N determines the lowest address translated using TTBR1, and the size of the level 1 translation table addressed by TTBR0.

### Table G5-1 Effect of TTBCR.N on address translation, Short-descriptor format

<table>
<thead>
<tr>
<th>TTBCR.N</th>
<th>First address translated with TTBR1</th>
<th>TTBR0 table Size</th>
<th>Index range</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000</td>
<td>TTBR1 not used</td>
<td>16KB</td>
<td>VA[31:20]</td>
</tr>
<tr>
<td>0b001</td>
<td>0x80000000</td>
<td>8KB</td>
<td>VA[30:20]</td>
</tr>
<tr>
<td>0b010</td>
<td>0x40000000</td>
<td>4KB</td>
<td>VA[29:20]</td>
</tr>
<tr>
<td>0b011</td>
<td>0x20000000</td>
<td>2KB</td>
<td>VA[28:20]</td>
</tr>
<tr>
<td>0b100</td>
<td>0x10000000</td>
<td>1KB</td>
<td>VA[27:20]</td>
</tr>
</tbody>
</table>
Whenever TTBCR.N is nonzero, the size of the translation table addressed by TTBR1 is 16KB.

Figure G5-6 shows how the value of TTBCR.N controls the boundary between VAs that are translated using TTBR0, and VAs that are translated using TTBR1.

In the selected TTBR, bits RGN, S, and IRGN[1:0] define the memory region attributes for the translation table walk.

Translation table walks, when using the VMSAv8-32 Short-descriptor translation table format describes the translation.

**G5.4.5 Translation table walks, when using the VMSAv8-32 Short-descriptor translation table format**

When using the Short-descriptor translation table format, and a memory access requires a translation table walk:

- A section-mapped access only requires a read of the level 1 translation table.
- A page-mapped access also requires a read of the level 2 translation table.

Reading a level 1 translation table on page G5-9170 describes how either TTBR1 or TTBR0 is used, with the accessed VA, to determine the address of the level 1 descriptor.

Reading a level 1 translation table on page G5-9170 shows the output address as A[39:0]:

- For a Non-secure PL1&0 stage 1 translation, this is the IPA of the required descriptor. A Non-secure PL1&0 stage 2 translation of this address is performed to obtain the PA of the descriptor.
- Otherwise, this address is the PA of the required descriptor.

The full translation flow for Sections, Supersections, Small pages and Large pages on page G5-9170 then shows the complete translation flow for each valid memory access.
**Reading a level 1 translation table**

When performing a fetch based on TTBR0:

- The address bits taken from TTBR0 vary between bits[31:14] and bits[31:7].
- The address bits taken from the VA, that is the input address for the translation, vary between bits[31:20] and bits[24:20].

The width of the TTBR0 and VA fields depend on the value of TTBCR.N, as Figure G5-7 shows.

When performing a fetch based on TTBR1, Bits TTBR1[31:14] are concatenated with bits[31:20] of the VA. This makes the fetch equivalent to that shown in Figure G5-7, with N==0.

--- **Note** ---

See *The address and Properties fields shown in the translation flows on page K9-11730* for more information about the *Properties* label used in this and other figures.

---

**Figure G5-7** Accessing level 1 translation table based on TTBR0, Short-descriptor format

Regardless of which register is used as the base for the fetch, the resulting output address selects a four-byte translation table entry that is one of:

- A level 1 descriptor for a Section or Supersection.
- A descriptor for a *translation table*, that points to a level 2 translation table. In this case:
  - A second fetch is performed to retrieve a level 2 descriptor.
  - The descriptor also contains some attributes for the access, see Figure G5-4 on page G5-9165.
- A faulting entry.

**The full translation flow for Sections, Supersections, Small pages and Large pages**

In a translation table walk, only the initial lookup uses the translation table base address from the appropriate TTBR. Subsequent lookups use a combination of address information from:

- The Table descriptor read in the previous lookup.
- The input address.

*Address translation examples using the VMSAv8-32 Short descriptor translation table format on page K9-11726* shows the full translation flow for each of the memory section and page options. As described in *VMSAv8-32 Short-descriptor Translation Table format descriptors on page G5-9164*, these options are:

- **Supersection**  A 16MB memory region, see *Translation flow for a Supersection on page K9-11726*.
- **Section**  A 1 MB memory region, see *Translation flow for a Section on page K9-11728*. 

---

‡ This field is absent if N is 0

N is the value of TTBCR.N

For details of the *Properties* field, see the register description
Large page  A 64KB memory region, described by the combination of:
  • A level 1 translation table entry that indicates the address of a level 2 translation table.
  • A level 2 descriptor that indicates a Large page.

See Translation flow for a Large page on page K9-11729.

Small page  A 4KB memory region, described by the combination of:
  • A level 1 translation table entry that indicates the address of a level 2 translation table.
  • A level 2 descriptor that indicates a Small page.

See Translation flow for a Small page on page K9-11730.
G5.5 The VMSAv8-32 Long-descriptor translation table format

The VMSAv8-32 Long-descriptor translation table format supports the assignment of memory attributes to memory Pages, at a granularity of 4KB, across the complete input address range. It also supports the assignment of memory attributes to blocks of memory, where a block can be 2MB or 1GB.

--- Note ---

- Although the VMSAv8-32 Long-descriptor format is limited to three levels of address lookup, its design and naming conventions support extension to additional levels, to support a larger input address range.
- Similarly, while the VMSAv8-32 implementation limits the output address range to 40 bits, its design supports extension to a larger output address range.

Figure G5-2 on page G5-9151 shows the different address translation stages. The Long-descriptor translation table format:

- Is used for:
  - The Non-secure EL2 stage 1 translation.
  - The Non-secure PL1&0 stage 2 translation.
- Can be used for the Secure and Non-secure PL1&0 translations.

When used for a stage 1 translation, the translation tables support an input address of up to 32 bits, corresponding to the VA address range of the PE.

When used for a stage 2 translation, the translation tables support an input address range of up to 40 bits, to support the translation from IPA to PA. If the input address for the stage 2 translation is a 32-bit address, then this address is zero-extended to 40 bits.

--- Note ---

When the Short-descriptor translation table format is used for the Non-secure stage 1 translations, this generates 32-bit IPAs. These are zero-extended to 40 bits to provide the input address for the stage 2 translation.

---

Overview of VMSAv8-32 address translation using Long-descriptor translation tables summarizes address translation from AArch32 state when using the Long-descriptor format translation tables.

The following sections then describe the format of the descriptors in the Long-descriptor format translation tables:

- VMSAv8-32 Long-descriptor Translation Table format descriptors on page G5-9173.
- Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors on page G5-9176.
- Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format on page G5-9180.

The following sections then describe this translation table format:

- Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format on page G5-9181.
- VMSAv8-32 Long-descriptor translation table format address lookup levels on page G5-9184.
- Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format on page G5-9187.
- The algorithm for finding the translation table entries, VMSAv8-32 Long-descriptor format on page G5-9190.

G5.5.1 Overview of VMSAv8-32 address translation using Long-descriptor translation tables

Figure G5-8 on page G5-9173 gives a general view of VMSAv8-32 stage 1 address translation when using the Long-descriptor translation table format.
G5.5 The VMSAv8-32 Long-descriptor translation table format

Figure G5-8 General view of VMSAv8-32 stage 1 address translation using Long-descriptor format

Figure G5-9 gives a general view of VMSAv8-32 stage 2 address translation. Stage 2 translation always uses the Long-descriptor translation table format.

G5.5.2 VMSAv8-32 Long-descriptor Translation Table format descriptors

As described in VMSAv8-32 Long-descriptor translation table format address lookup levels on page G5-9184, the Long-descriptor translation table format provides up to three levels of address lookup. A translation table walk starts either at level 1 or level 2 of the address lookup.

In general, a descriptor is one of:
- An invalid or fault entry.
- A table entry, that points to the next-level translation table.
- A block entry, that defines the memory properties for the access.
- A reserved format.

Bit[1] of the descriptor indicates the descriptor type, and bit[0] indicates whether the descriptor is valid.

The following sections describe the Long-descriptor Translation Table descriptor formats:
- VMSAv8-32 Long-descriptor level 1 and level 2 descriptor formats on page G5-9174.
- VMSAv8-32 Long-descriptor translation table level 3 descriptor formats on page G5-9175.
Information returned by a translation table lookup on page G5-9159 describes the classification of the non-address fields in the descriptors between address map control, access controls, and region attributes.

VMSAv8-32 Long-descriptor level 1 and level 2 descriptor formats

In the Long-descriptor translation tables, the formats of the level 1 and level 2 descriptors differ only in the size of the block of memory addressed by the Block descriptor. A block entry:

- In a level 1 table describes the mapping of the associated 1GB input address range.
- In a level 2 table describes the mapping of the associated 2MB input address range.

Figure G5-10 shows the Long-descriptor level 1 and level 2 descriptor formats:

![Figure G5-10 VMSAv8-32 Long-descriptor level 1 and level 2 descriptor formats](image)

For the level 1 descriptor, n is 30. For the level 2 descriptor, n is 21.

The level 1 descriptor returns the address of the level 2 table.
The level 2 descriptor returns the address of the level 3 table.

See the descriptions of the address fields for more information about bits[47:40] of the Block and Table descriptors.

Descriptor encodings, Long-descriptor level 1 and level 2 formats

Descriptor bit[0] identifies whether the descriptor is valid, and is 1 for a valid descriptor. If a lookup returns an invalid descriptor, the associated input address is unmapped, and any attempt to access it generates a Translation fault.

Descriptor bit[1] identifies the descriptor type, and is encoded as:

- **0, Block**: The descriptor gives the base address of a block of memory, and the attributes for that memory region.
- **1, Table**: The descriptor gives the address of the next level of translation table, and for a stage 1 translation, some attributes for that translation.

The other fields in the valid descriptors are:

**Block descriptor**

Gives the base address and attributes of a block of memory:

- For a level 1 Block descriptor, bits[39:30] are bits[39:30] of the output address that specifies a 1GB block of memory.

In both cases, if bits[47:40] of the descriptor are not zero then a translation that uses the descriptor will generate an Address size fault, see Address size fault on page G5-9240.
Bits[63:52, 11:2] provide attributes for the target memory block, see Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors on page G5-9176. The position and contents of these bits is identical in the level 2 Block descriptor and in the level 3 Page descriptor.

Table descriptor

Bits[39:m] are bits[39:m] of the address of the required next-level table. Bits[m-1:0] of the table address are zero:

• For a level 1 Table descriptor, this is the address of a level 2 table.
• For a level 2 Table descriptor, this is the address of a level 3 table.

In both cases, if bits[47:40] of the descriptor are not zero then a translation that uses the descriptor will generate an Address size fault, see Address size fault on page G5-9240.

For a stage 1 translation only, bits[63:59] provide attributes for the next-level lookup, see Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors on page G5-9176.

If the translation table defines the Non-secure PL1&0 stage 1 translations, then the output address in the descriptor is the IPA of the target block or table. Otherwise, it is the PA of the target block or table.

VMSAv8-32 Long-descriptor translation table level 3 descriptor formats

Each entry in a level 3 table describes the mapping of the associated 4KB input address range.

Figure G5-11 shows the Long-descriptor level 3 descriptor formats.

| 63 | 52 | 51 | 40 | 39 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|Invalid| | | | | | | | | | | | | | | | | | 1 | 0 |

Invalid

Reserved, invalid

<table>
<thead>
<tr>
<th>63</th>
<th>52</th>
<th>51</th>
<th>40</th>
<th>39</th>
<th>12</th>
<th>11</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, invalid</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reserved, invalid

Page

| 63 | 52 | 51 | 40 | 39 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|Upper page attributes| | | | | | | | | | | | | | | | | | 1 | 1 |

Upper page attributes

SBZ‡

Output address[39:12]

Lower page attributes

‡ See the description of the address field for more information about bits[47:40] of the Page descriptor.

Figure G5-11 VMSAv8-32 Long-descriptor level 3 descriptor formats

Descriptor bit[0] identifies whether the descriptor is valid, and is 1 for a valid descriptor. If a lookup returns an invalid descriptor, the associated input address is unmapped, and any attempt to access it generates a Translation fault.

Descriptor bit[1] identifies the descriptor type, and is encoded as:

0, Reserved, invalid

Behaves identically to encodings with bit[0] set to 0.

This encoding must not be used in level 3 translation tables.

1, Page

Gives the address and attributes of a 4KB page of memory.

At this level, the only valid format is the Page descriptor. The other fields in the Page descriptor are:

Page descriptor


If bits[47:40] of the descriptor are not zero, then a translation that uses the descriptor will generate an Address size fault, see Address size fault on page G5-9240.

Bits[63:52, 11:2] provide attributes for the target memory page, see Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors on page G5-9176. The position and contents of these bits are identical in the level 1 Block descriptor and in the level 2 Block descriptor.
If the translation table defines the Non-secure PL1&0 stage 1 translations, then the output address in the descriptor is the IPA of the target page. Otherwise, it is the PA of the target page.

**G5.5.3 Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors**

The memory attributes in the VMSAv8-32 Long-descriptor translation tables are based on those in the Short-descriptor translation table format, with some extensions. *Memory region attributes on page G5-9203* describes these attributes. In the Long-descriptor translation table format:

- Table entries for stage 1 translations define attributes for the next level of lookup, see *Next-level attributes in VMSAv8-32 Long-descriptor stage 1 Table descriptors*

  The hierarchical attributes in the translation tables, APTable, XNTable, and PXNTable, permit subtrees of the translation tables to be used by different agents. Not all operating systems use this functionality, and so FEAT_AA32HPD adds a facility to disable these bits.

  This ability to disable hierarchical attribute bits has no effect on the NSTable bit.

- Block and Page entries define memory attributes for the target block or page of memory. Stage 1 and stage 2 translations have some differences in these attributes, see:
  - *Attribute fields in VMSAv8-32 Long-descriptor stage 1 Block and Page descriptors on page G5-9177.*
  - *Attribute fields in VMSAv8-32 Long-descriptor stage 2 Block and Page descriptors on page G5-9179.*

**Next-level attributes in VMSAv8-32 Long-descriptor stage 1 Table descriptors**

In a Table descriptor for a stage 1 translation, bits[63:59] of the descriptor define the following attributes for the next-level translation table access:

- **NSTable, bit[63]**
  
  For memory accesses from Secure state, specifies the Security state for subsequent levels of lookup, see *Hierarchical control of Secure or Non-secure memory accesses, Long-descriptor format on page G5-9180.*

  For memory accesses from Non-secure state, this bit is ignored.

- **APTable, bits[62:61]**
  
  Access permissions limit for subsequent levels of lookup, see *Hierarchical control of access permissions, Long-descriptor format on page G5-9193.*

  APTable[0] is reserved, SBZ, in the Non-secure EL2 stage 1 translation tables.

  From Armv8.2, when FEAT_AA32HPD is implemented, this field can be disabled.

  When the value of TTBCR2.HPD0 or TTBCR2.HPD1 is 1, and the value of TTBCR.T2E is also 1:

  - The value of the corresponding APTable field is IGNORED by hardware, allowing the field to be used by software.
  - The behavior of the system is as if the value of the corresponding APTable field is 0, that is to say, the APTable field has an *Effective value* of 0.

- **XNTable, bit[60]**
  
  XN limit for subsequent levels of lookup, see *Hierarchical control of instruction fetching, Long-descriptor format on page G5-9198.*

  From Armv8.2, when FEAT_AA32HPD is implemented, this field can be disabled.

  When the value of TTBCR2.HPD0 or TTBCR2.HPD1 is 1, and the value of TTBCR.T2E is also 1:

  - The value of the corresponding XNTable field is IGNORED by hardware, allowing the field to be used by software.
  - The behavior of the system is as if the value of the corresponding XNTable field is 0, that is to say, the XNTable field has an *Effective value* of 0.

- **PXNTable, bit[59]**
  
  PXN limit for subsequent levels of lookup, see *Hierarchical control of instruction fetching, Long-descriptor format on page G5-9198.*

  This bit is RES0 in the Non-secure EL2 stage 1 translation tables.
From Armv8.2, when FEAT_AA32HPD is implemented, this field can be disabled.

When the value of TTBCR2.HPD0 or TTBCR2.HPD1 is 1 and the value of TTBCR.T2E is also 1:
- The value of the corresponding PXNTable field is ignored by hardware, allowing the field to be used by software.
- The behavior of the system is as if the value of the corresponding PXNTable field is 0, that is to say, the PXNTable field has an effective value of 0.

Attribute fields in VMSAv8-32 Long-descriptor stage 1 Block and Page descriptors

In Block and Page descriptors, the memory attributes are split into an upper block and a lower block as shown for a stage 1 translation:

- **PBHA**, bits[62:59]
  - Page-based hardware attributes bits.
  - These bits are ignored when FEAT_HPDS2 is not implemented.
  - When FEAT_HPDS2 is implemented, the HTCR and the TTBCR2 registers both contain a control bit for each PBHA bit in the translation tables that they control. When the value of that control bit is 1, and the value of the corresponding Hierarchical permission disables bit is 1, hardware can use that PBHA bit for implementation-defined purposes. When the PBHA bit is used for implementation-defined purposes, the value of 0 in the PBHA bit is a safe default setting that gives the same behavior as when the PBHA bit is not used for implementation-defined purposes.
  - The control bits for this feature are:
    - For a Non-secure EL2 translation regime:
      - HTCR.HWUnn
        - Controls whether Block or Page descriptor bit[nn] can be used by hardware.
        - These controls apply only when the value of HTCR.HPD is 1.
    - For a PL1&0 translation regime:
      - TTBCR2.HWU1nn
        - For the translation tables indicated by TTBR1, controls whether Block or Page descriptor bit[nn] can be used by hardware.
        - These controls apply only when the value of TTBCR2.HPD1 is 1 and the value of TTBCR.T2E is 1.
      - TTBCR2.HWU0nn
        - For the translation tables indicated by TTBR0, controls whether Block or Page descriptor bit[nn] can be used by hardware.
        - These controls apply only when the value of TTBCR2.HPD0 is 1 and the value of TTBCR.T2E is 1.
Implementation of FEAT_HPDS2 requires the implementation of FEAT_AA32HPD, which provides the Hierarchical permission disables bits. If FEAT_AA32HPD is implemented but FEAT_HPDS2 is not implemented, then the control bits are RAZ/WI but other aspects of FEAT_AA32HPD functionality are implemented. If neither feature is implemented, then:

- The control bits are RAZ/WI.
- The FEAT_AA32HPD identification registers indicate that the functionality is not supported, see FEAT_AA32HPD on page A2-89.
- The TTBCR2 register encoding is treated as unallocated.

**XN, bit[54]**  The Execute-never field, see Access permissions for instruction execution on page G5-9196.

**PXN, bit[53]**  The Privileged execute-never field, see Access permissions for instruction execution on page G5-9196.

This bit is RES0 in the Non-secure EL2 stage 1 translation tables.

**Contiguous, bit[52]**

 Indicates that 16 adjacent translation table entries point to contiguous memory regions, see Contiguous bit on page G5-9211.

**nG, bit[11]**  The not global bit. Determines how the translation is marked in the TLB, see Global and process-specific translation table entries on page G5-9216.

This bit is RES0 in the Non-secure EL2 stage 1 translation tables.

**AF, bit[10]**  The Access flag, see The Access flag on page G5-9200.

**SH, bits[9:8]**  Shareability field, see Memory region attributes on page G5-9203.

**AP[2:1], bits[7:6]**

 Access Permissions bits, see Memory access control on page G5-9192.

--- Note ---

For consistency with the Short-descriptor translation table formats, the Long-descriptor format defines AP[2:1] as the Access Permissions bits, and does not define an AP[0] bit.

---

AP[1] is RES1 in the Non-secure EL2 stage 1 translation tables.

**NS, bit[5]**  Non-secure bit. For memory accesses from Secure state, specifies whether the output address is in Secure or Non-secure memory, see Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format on page G5-9180.

For memory accesses from Non-secure state, this bit is RES0 and is ignored by the PE.

**AttrIndx[2:0], bits[4:2]**

 Stage 1 memory attributes index field, for the indicated Memory Attribute Indirection Register, see VMSAv8-32 Long-descriptor format memory region attributes on page G5-9210.

The definition of IGNORED means the architecture guarantees that the PE makes no use of the field, see IGNORED on page Glossary-11937. For more information about these fields, see Other fields in the Long-descriptor translation table format descriptors on page G5-9211.
Attribute fields in VMSAv8-32 Long-descriptor stage 2 Block and Page descriptors

In Block and Page descriptors, the memory attributes are split into an upper block and a lower block as shown for a stage 2 translation:

<table>
<thead>
<tr>
<th>Upper attributes</th>
<th>Lower attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>83 62 60 59 58</td>
<td>11 10 9 8 7 6 5 2</td>
</tr>
<tr>
<td>PBHA†</td>
<td>IGNORED</td>
</tr>
<tr>
<td>Reserved for use by System MMU</td>
<td></td>
</tr>
<tr>
<td>Reserved for software use</td>
<td></td>
</tr>
<tr>
<td>XN[1:0]‡</td>
<td></td>
</tr>
<tr>
<td>Contiguous</td>
<td></td>
</tr>
</tbody>
</table>

† Bit[53] is RES0 if FEAT_XNX is not implemented.
‡ Bits [62:60] are IGNORED and reserved for use by System MMU if FEAT_HPDS2 is not implemented.
Bits [59] is IGNORED if FEAT_HPDS2 is not implemented.

For a stage 2 descriptor, the attributes are:

**PBHA[3:1], bits[62:60]**

Page-based hardware attributes bits.
These bits are IGNORED and reserved for System MMU use when FEAT_HPDS2 is not implemented.

When FEAT_HPDS2 is implemented, VTCR_EL2 has a control bit for each PBHA bit in the EL1&0 stage 2 translation tables:
- When the value of that control bit is 1, hardware can use the corresponding PBHA bit for IMPLEMENTATION DEFINED purposes. When the PBHA bit is used for IMPLEMENTATION DEFINED purposes, the value of 0 in the PBHA bit is a safe default setting that gives the same behavior as when the PBHA bit is not used for IMPLEMENTATION DEFINED purposes.
- When the value of that control bit is 0, the corresponding PBHA bit is IGNORED and reserved for System MMU use.

**PBHA[0], bit[59]**

Page-based hardware attributes bit.
This bit is IGNORED when FEAT_HPDS2 is not implemented.

When FEAT_HPDS2 is implemented, VTCR_EL2 has a control bit for this bit in the EL1&0 stage 2 translation tables:
- When the value of that control bit is 1, hardware can use this bit for IMPLEMENTATION DEFINED purposes. When the PBHA bit is used for IMPLEMENTATION DEFINED purposes, the value of 0 in the PBHA bit is a safe default setting that gives the same behavior as when the PBHA bit is not used for IMPLEMENTATION DEFINED purposes.
- When the value of that control bit is 0, this bit is IGNORED.

**XN[1:0], bits[54:53]**

The stage 2 Execute-never field, see Access permissions for instruction execution on page G5-9196.
If FEAT_XNX is not implemented, bit[53] is RES0.

**Contiguous, bit[52]**

Indicates that 16 adjacent translation table entries point to contiguous memory regions, see Contiguous bit on page G5-9211.

**AF, bit[10]**
The Access flag, see The Access flag on page G5-9200.

**SH, bits[9:8]**
Shareability field, see EL2 control of Non-secure memory region attributes on page G5-9212.
S2AP, bits[7:6]
Stage 2 Access Permissions bits, see *Hyp mode control of Non-secure access permissions on page G5-9201.*

--- Note ---
In the original VMSAv7-32 Long-descriptor attribute definition, this field was called HAP[2:1], for consistency with the AP[2:1] field in the stage 1 descriptors and despite there being no HAP[0] bit. VMSAv8-32 renames the field for greater clarity.

MemAttr, bits[5:2]
Stage 2 memory attributes, see *EL2 control of Non-secure memory region attributes on page G5-9212.*

The definition of *IGNORED* means the architecture guarantees that the PE makes no use of the field, see *IGNORED on page Glossary-11937.* For more information about these fields, see *Other fields in the Long-descriptor translation table format descriptors on page G5-9211.*

### G5.5.4 Control of Secure or Non-secure memory access, VMSAv8-32 Long-descriptor format

*Access to the Secure or Non-secure PA map on page G5-9161* describes how the NS bit in the translation table entries:

- For accesses from Secure state, determines whether the access is to Secure or Non-secure memory.
- Is ignored by accesses from Non-secure state.

In the Long-descriptor format:

- The NS bit relates only to the memory block or page at the output address defined by the descriptor.
- The descriptors also include an NSTable bit, see *Hierarchical control of Secure or Non-secure memory accesses, Long-descriptor format.*

The NS and NSTable bits are valid only for memory accesses from Secure state. Memory accesses from Non-secure state ignore the values of these bits.

Hierarchical control of Secure or Non-secure memory accesses, Long-descriptor format

For Long-descriptor Format Table descriptors for stage 1 translations, the descriptor includes an NSTable bit, which indicates whether the table identified in the descriptor is in Secure or Non-secure memory. For accesses from Secure state, the meaning of the NSTable bit is:

\[
\text{NSTable} == 0 \quad \text{The defined table address is in the Secure PA map. In the descriptors in that translation table, NS bits and NSTable bits have their defined meanings.}
\]

\[
\text{NSTable} == 1 \quad \text{The defined table address is in the Non-secure PA map. Because this table is fetched from the Non-secure address map, the NS and NSTable bits in the descriptors in this table must be ignored. This means that, for this table:}
\]

\[
\begin{align*}
&\quad \text{• The value of the NS bit in any Block or Page descriptor is ignored. The block or page address refers to Non-secure memory.} \\
&\quad \text{• The value of the NSTable bit in any Table descriptor is ignored, and the table address refers to Non-secure memory. When this table is accessed, the NS bit in any Block or Page descriptor is ignored, and all descriptors in the table refer to Non-secure memory.}
\end{align*}
\]

In addition, an entry fetched in Secure state is treated as non-global if it is read from Non-secure memory. That is, these entries must be treated as if nG==1, regardless of the value of the nG bit. For more information about the nG bit, see *Global and process-specific translation table entries on page G5-9216.*
The effect of NSTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore, a change to NSTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.

**Note**

- When using the Long-descriptor Format, Table descriptors are defined only for the level 1 and level 2 of lookup.
- Stage 2 translations are performed only for operations in Non-secure state, that can access only the Non-secure address map. Therefore, the stage 2 descriptors do not include NS or NSTable bits.

**G5.5.5 Selecting between TTBR0 and TTBR1, VMSA v8-32 Long-descriptor translation table format**

As described in *Determining the translation table base address in the VMSA v8-32 translation regimes* on page G5-9160, two sets of translation tables can be defined for each of the PL1&0 stage 1 translations, and TTBR0 and TTBR1 hold the base addresses for the two sets of tables. The Long-descriptor translation table format provides more flexibility in defining the boundary between using TTBR0 and using TTBR1. When a PL1&0 stage 1 address translation is enabled, TTBR0 is always used. If TTBR1 is also used then:

- TTBR1 is used for the top part of the input address range.
- TTBR0 is used for the bottom part of the input address range.

The TTBCR.T0SZ and TTBCR.T1SZ size fields control the use of TTBR0 and TTBR1, as Table G5-2 on page G5-9181 shows.

<table>
<thead>
<tr>
<th>TTBCR</th>
<th>Input address range using:</th>
<th>TTBR0</th>
<th>TTBR1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T0SZ</td>
<td>T1SZ</td>
<td>All addresses</td>
</tr>
<tr>
<td>0b000</td>
<td>0b000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$M^a$</td>
<td>0b000</td>
<td>$N^a$</td>
<td>Zero to $(2^{32-M}-1)$</td>
</tr>
<tr>
<td>0b000</td>
<td>$N^a$</td>
<td></td>
<td>Zero to $(2^{32-2(M-N)}-1)$</td>
</tr>
<tr>
<td>$M^a$</td>
<td>$N^a$</td>
<td></td>
<td>Zero to $(2^{32-M}-1)$</td>
</tr>
</tbody>
</table>

a. $M, N$ must be greater than 0. The maximum possible value for each of T0SZ and T1SZ is 7.

For stage 1 translations, the input address is always a VA, and the maximum possible VA is $(2^{32}-1)$.

When address translation is using the Long-descriptor translation table format:

- Figure G5-12 on page G5-9182 shows how, when TTBCR.T1SZ is zero, the value of TTBCR.T0SZ controls the boundary between VAs that are translated using TTBR0, and VAs that are translated using TTBR1.
The AArch32 Virtual Memory System Architecture
G5.5 The VMSAv8-32 Long-descriptor translation table format

Figure G5-12 Control of TTBR boundary, when TTBCR.T1SZ is zero

- Figure G5-13 shows how, when TTBCR.T1SZ is nonzero, the values of TTBCR.T0SZ and TTBCR.T1SZ control the boundaries between VAs that are translated using TTBR0, and VAs that are translated using TTBR1.

Figure G5-13 Control of TTBR boundaries, when TTBCR.T1SZ is nonzero

- When T0SZ and T1SZ are both nonzero:
  - If both fields are set to 0b001, the boundary between the two regions is 0x80000000. This is identical to having T0SZ set to 0b000 and T1SZ set to 0b011.
  - Otherwise, the TTBR0 and TTBR1 regions are non-contiguous. In this case, any attempt to access an address that is in that gap between the TTBR0 and TTBR1 regions generates a Translation fault.

Note

The handling of the Contiguous bit can mean that the boundary between the translation regions defined by the TCR_EL1.TnSZ values and the region for which an access generates a Translation fault is wider than shown in Figure G5-13. That is, if the descriptor for an access to the region shown as generating a fault has the Contiguous bit set to 1, the access might not generate a fault. Possible errors in programming the translation table registers on page G5-9183 describes this possibility.

When using the Long-descriptor translation table format:

- The TTBCR contains fields that define memory region attributes for the translation table walk, for each TTBR. These are the SH0, ORGN0, IRGN0, SH1, ORGN1, and IRGN1 bits.
- TTBR0 and TTBR1 each contain an ASID field, and the TTBCR.A1 field selects which ASID to use.
For this translation table format, VMSAv8-32 Long-descriptor translation table format address lookup levels on page G5-9184 summarizes the lookup levels, and Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format on page G5-9187 describes the possible translations.

Possible errors in programming the translation table registers

In all the descriptions in this subsection, the size of the input address supported for a PL1&0 stage 1 translation refers to the size specified by a TTSCR.TxSZ field.

Note

For a PL1&0 stage 1 translation, the input address range can be split so that the lower addresses are translated by TTBR0 and the higher addresses are translated by TTBR1. In this case, each of input address sizes specified by TTSCR.{T0SZ, T1SZ} is smaller than the total address size supported by the stage of translation.

The following are possible errors in the programming of TTBR0, TTBR1, and TTSCR. For the translation of a particular address at a particular stage of translation, either:

- The block size being used to translate the address is larger than the size of the input address supported at a stage of translation used in performing the required translation. This can occur only for the PL1&0 stage 1 translations, and only when either TTSCR.T0SZ or TTSCR.T1SZ is zero, meaning there is no gap between the address range translated by TTBR0 and the range translated by TTBR1. In this case, this programming error occurs if a block translated from the region that has TxSZ set to zero straddles the boundary between the two address ranges. Example G5-2 shows an example of this mis-programming.

- The address range translated by a set of blocks marked as contiguous, by use of the contiguous bit, is larger than the size of the input address supported at a stage of translation used in performing the required translation.

Example G5-2 Error in programming the translation table registers

If TTSCR.T0SZ is programmed to 0 and TTSCR.T1SZ is programmed to 7, this means:

- TTBR0 translates addresses in the range 0x00000000-0xFDFFFFFF.
- TTBR1 translates addresses in the range 0xFE000000-0xFFFFFFFF.

The translation table indicated by TTBR0 might be programmed with a block entry for a 1GB region starting at 0xC0000000. This covers the address range 0xC0000000-0xFFFFFFFF, that overlaps the TTBR1 address range. This means this block size is larger than the input address size supported for translations using TTBR0, and therefore this is a programming error.

To understand why this must be a programming error, consider a memory access to address 0xFFFF0000. According to the TTSCR.{T0SZ, T1SZ} values, this must be translated using TTBR1. However, the access matches a TLB entry for the translation, using TTBR0, of the block at 0xC0000000. Hardware is not required to detect that the access to 0xFFFF0000 is being translated incorrectly.

In these cases, an implementation might use one of the following approaches:

- Treat such a block as causing a Translation fault, even though the block is valid, and the address accessed within that block is within the size of the input address supported at a stage of translation.
  The block might be a block within a contiguous set of blocks.

- Treat such a block as not causing a Translation fault, even though the address accessed within that block is outside the size of the input address supported at a stage of translation, provided that both of the following apply:
  - The block is valid.
  - At least one address within the block, or contiguous set of blocks, is within the size of the input address supported at a stage of translation.
  The block might be a block within a contiguous set of blocks.
Additional constraints apply to programming the VTCR, see Determining the required initial lookup level for stage 2 translations on page G5-9189.

G5.5.6 VMSAv8-32 Long-descriptor translation table format address lookup levels

As stated at the start of this section, because the Long-descriptor translation table format is used for the Non-secure PL1&0 stage 2 translations, the format must support input addresses of up to 40 bits.

Table G5-3 on page G5-9184 summarizes the properties of the different levels of address lookup when using this format.

Table G5-3 Properties of the three levels of address lookup with VMSAv8-32 Long-descriptor translation tables

<table>
<thead>
<tr>
<th>Level</th>
<th>Input address</th>
<th>Output addressa</th>
<th>Number of entries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td>Address rangeb</td>
<td>Size</td>
</tr>
<tr>
<td>First</td>
<td>Up to 512GB</td>
<td>Up to Address[38:0]</td>
<td>1GB</td>
</tr>
<tr>
<td>Second</td>
<td>Up to 1GB</td>
<td>Up to Address[29:0]</td>
<td>2MB</td>
</tr>
<tr>
<td>Third</td>
<td>2MB</td>
<td>Address[20:0]</td>
<td>4KB</td>
</tr>
</tbody>
</table>

a. Output address when an entry addresses a block of memory or a memory page. If an entry addresses the next level of address lookup it specifies Address[39:12] for the next-level translation table.
b. Input address range for the translation table. See Use of concatenated level 1 translation tables on page G5-9185 for details of support for additional bits of address at a given level, including possible support of a 40-bit input address range for stage 2 translations at level 1. For stage 1 translations at level 1 the input address range is limited to the VA size of [31:0].

For level 1 and level 2 tables, reducing the input address range reduces the number of addresses in the table and therefore reduces the table size. The appropriate Translation Table Control Register specifies the input address range.

Stage 1 translations require an input address range of up to 32 bits, corresponding to VA[31:0]. For these translations:

- For a memory access from a mode other than Hyp mode, the Secure or Non-secure TTBR0 or TTBR1 holds the translation table base address, and the Secure or Non-secure TTBCR is the control register.
- For a memory access from Hyp mode, HTTBR holds the translation table base address, and HTCR is the control register.

Note

For translations controlled by TTBR0 and TTBR1, if neither TTBR has an input address range larger than 1GB, then translation starts at level 2. Together, TTBR0 and TTBR1 can still cover the 32-bit VA input address range.

Stage 2 translations require an input address range of up to 40 bits, corresponding to IPA[39:0], and the supported input address size is configurable in the range 25-40 bits. Table G5-3 on page G5-9184 indicates a requirement for the translation mechanism to support a 39-bit input address range, Address[38:0]. Use of concatenated translation tables for the initial stage 2 lookup on page G5-9185 describes how a 40-bit IPA address range is supported. For stage 2 translations:

- VTTBR holds the translation table base address, and VTCR is the control register.
- If a supplied input address is larger than the configured input address size, a Translation fault is generated.
Use of concatenated translation tables for the initial stage 2 lookup

If a stage 2 translation would require 16 entries or fewer in its top-level translation table, that stage of translation can, instead, be configured so that:

- It requires the corresponding number of concatenated translation tables at the next translation level, aligned to the size of the block of concatenated translation tables.
- The stage 2 translation starts at that next translation level.

——— Note ————
Stage 2 translations always use the Long-descriptor translation table format.

This use of concatenated translation tables is:
- Required when the stage 2 translation supports a 40-bit input address range, see Use of concatenated level 1 translation tables.
- Supported for a stage 2 translation with an input address range of 31-34 bits, see Use of concatenated level 2 translation tables.

The use of concatenated translation tables requires the software that is defining the translation to:
- Define the concatenated translation tables with the required overall alignment.
- Program VTTBR to hold the address of the first of the concatenated translation tables.
- Program VTCR to indicate the required input address range and initial lookup level.

——— Note ————
The use of concatenated translation tables avoids the overhead of an additional level of translation.

Use of concatenated level 1 translation tables

The Long-descriptor format translation tables provide 9 bits of address resolution at each level of lookup. However, a 40-bit input address range with a translation granularity of 4KB requires a total of 28 bits of address resolution. Therefore, a stage 2 translation that supports a 40-bit input address range requires two concatenated level 1 translation tables, together aligned to 8KB, where:

- The table at the address with PA[12:0]==0b0_0000_0000_0000 defines the translations for input addresses with bit[39]==0.
- The table at the address with PA[12:0]==0b1_0000_0000_0000 defines the translations for input addresses with bit[39]==1.
- The 8KB alignment requirement means that both tables have the same value for PA[39:13].

Use of concatenated level 2 translation tables

A stage 2 translation with an input address range of 31-34 bits can start the translation either:
- With a level 1 lookup, accessing a level 1 translation table with 2-16 entries.
- With a level 2 lookup, accessing a set of concatenated level 2 translation tables.

Table G5-4 on page G5-9186 shows these options, for each of the input address ranges that can use this scheme.

——— Note ————
Because these are stage 2 translations, the input address range is an IPA range.
### Table G5-4 Possible uses of concatenated translation tables for level 2 lookup

<table>
<thead>
<tr>
<th>IPA range</th>
<th>Size</th>
<th>Lookup starts at level 1</th>
<th>Lookup starts at level 2</th>
<th>Required alignment^a</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPA[30:0]</td>
<td>2-byte</td>
<td>2</td>
<td>2</td>
<td>8KB</td>
</tr>
<tr>
<td>IPA[31:0]</td>
<td>2^2 bytes</td>
<td>4</td>
<td>4</td>
<td>16KB</td>
</tr>
<tr>
<td>IPA[32:0]</td>
<td>2^3 bytes</td>
<td>8</td>
<td>8</td>
<td>32KB</td>
</tr>
<tr>
<td>IPA[33:0]</td>
<td>2^4 bytes</td>
<td>16</td>
<td>16</td>
<td>64KB</td>
</tr>
</tbody>
</table>

^a. Required alignment of the set of concatenated level 2 tables.

See also *Determining the required initial lookup level for stage 2 translations* on page G5-9189.
G5.5.7   Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format

Figure G5-2 on page G5-9151 shows the possible address translations. If a stage of translation is controlled from an Exception level that is using AArch32, the input and output address constraints and the registers that control the translation are as follows:

**Stage 1 translations**

For all stage 1 translations:
- The input address range is up to 32 bits, as determined by either:
  - TTBCR.T0SZ or TTBCR.T1SZ, for a PL1&0 stage 1 translation.
  - HTCR.T0SZ, for an EL2 stage 1 translation.
- The output address range is 40 bits.

The stage 1 translations are:

**Non-secure PL1&0 stage 1 translation**

The stage 1 translation for memory accesses from Non-secure modes other than Hyp mode. This translates a VA to an IPA. For this translation, when Non-secure EL1 is using AArch32:
- Non-secure TTBR0 or TTBR1 holds the translation table base address.
- Non-secure TTBCR determines which TTBR is used.

**Non-secure EL2 stage 1 translation**

The stage 1 translation for memory accesses from Hyp mode, translates a VA to a PA. For this translation, when EL2 is using AArch32, HTTBR holds the translation table base address.

**Secure PL1&0 stage 1 translation**

The stage 1 translation for memory accesses from Secure modes, translates a VA to a PA. For this translation, when the Secure PL1 modes are using AArch32:
- Secure TTBR0 or TTBR1 holds the translation table base address.
- Secure TTBCR determines which TTBR is used.

**Stage 2 translation**

**Non-secure PL1&0 stage 2 translation**

The stage 2 translation for memory accesses from Non-secure modes other than Hyp mode, and translates an IPA to a PA. For this translation, when EL2 is using AArch32:
- The input address range is 40 bits, and VTCR.T0SZ determines the input address size.
- The output address range depends on the implemented memory system, and is up to 40 bits.
- VTTBR holds the translation table base address.
- VTCR specifies the required input address range, and whether the initial lookup is at level 1 or at level 2.

The descriptions of the VMSAv8-32 translation stages state that the maximum output address size is 40 bits. However, the register and Long-descriptor Format descriptor fields that hold these addresses are 48 bits wide. If bits[47:40] of an output address are not all zero, then the address generates an Address size fault.

The Long-descriptor translation table format provides up to three levels of address lookup, as described in VMSAv8-32 Long-descriptor translation table format address lookup levels on page G5-9184, and the initial lookup, in which the MMU reads the translation table base address, is at either level 1 or level 2. The following determines the level of the initial lookup:

- For a stage 1 translation, the required input address range. For more information, see Determining the required initial lookup level for stage 1 translations on page G5-9189.
- For a stage 2 translation, the level specified by the VTCR.SL0 field. For more information, see Determining the required initial lookup level for stage 2 translations on page G5-9189.
Note

For a stage 2 translation, the size of the required input address range constrains the VTCR.SL0 value.

Figure G5-14 shows how the descriptor address for the initial lookup for a translation using the Long-descriptor translation table format is determined from the input address and the TTBR value. This figure shows the lookup for a translation that starts with a level 1 lookup, that translates bits[39:30] of the input address, zero extended if necessary.

If bits[47:40] of the TTBR are not zero then the initial lookup will generate an Address size fault, see Address size fault on page G5-9240.

For a translation that starts with a level 1 lookup, as shown in Figure G5-14:

For a stage 1 translation

\( n \) is in the range 4-5 and:

- For a memory access from Hyp mode:
  - HTTBR is the TTBR.
  - \( n = 5 - (HTCR.T0SZ) \).
- For other accesses:
  - The Secure or Non-secure instance of TTBR0 or TTBR1 is the TTBR.
  - \( n = (5 - TTBCR.TxSZ) \), where \( x \) is 0 when using TTBR0, and 1 when using TTBR1.

For a stage 2 translation

\( n \) is in the range 4-13 and:

- VTTBR is the TTBR.
- \( n = 5 - (VTCR.T0SZ) \).
For a translation that starts with a level 2 lookup, the descriptor address is obtained in the same way, except that bits\(((n+17):21)\) of the input address provide bits\(((n-1):3)\) of the descriptor address, where:

**For a stage 1 translation**

\(n\) is in the range 7-12. As *Determining the required initial lookup level for stage 1 translations* shows, for a stage 1 translation to start with a level 2 lookup, the corresponding T0SZ or T1SZ field must be 2 or more. This means:

- For a memory access from Hyp mode, \(n=14-\text{HTCR.T0SZ}\).
- For other memory accesses, \(n=14-(\text{TTBCR.TxSZ})\), where \(x\) is 0 when using TTBR0, and 1 when using TTBR1.

**For a stage 2 translation**

\(n\) is in the range 7-16. For a stage 2 translation to start with a level 2 lookup, VTCR.SL0 is \(\text{0b00}\), and \(n=14-(\text{VTCR.T0SZ})\).

The following sections describe how the level of the initial lookup is determined:

- *Determining the required initial lookup level for stage 1 translations*.
- *Determining the required initial lookup level for stage 2 translations*.

*Address translation examples using the VMSAv8-32 Long descriptor translation table format* on page K9-11731 shows examples of full translation flows, to an entry for a 4KB memory page, for lookups starting at level 1 and lookups starting at level 2.

**Determining the required initial lookup level for stage 1 translations**

For a stage 1 translation, the required input address range, indicated by a T0SZ or T1SZ field in a translation table control register, determines the initial lookup level. The size of this input address region is \(2^{(32-\text{TxSZ})}\) bytes, and if this size is:

- Less than or equal to \(2^{30}\) bytes, the required start is at level 2, and translation requires two levels of table to map to 4KB pages. This corresponds to a TxSZ value of 2 or more.
- More than \(2^{30}\) bytes, the required start is at level 1, and translation requires three levels of table to map to 4KB pages. This corresponds to a TxSZ value that is less than 2.

For the PL1&0 stage 1 translations, the TTBCR:

- Splits the 32-bit VA input address range between TTBR0 and TTBR1, see *Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-desciption translation table format* on page G5-9181.
- Holds the input address range sizes for TTBR0 and TTBR1, in the TTBCR.T0SZ and TTBCR.T1SZ fields.

For the EL2 stage 1 translations, HTCR.T0SZ indicates the size of the required input address range. For example, if this field is \(\text{0b000}\), it indicates a 32-bit VA input address range, and translation lookup must start at level 1.

**Determining the required initial lookup level for stage 2 translations**

For a PL1&0 stage 2 translation, the output address range from the PL1&0 stage 1 translations determines the required input address range for the stage 2 translation.

VTCR.SL0 indicates the starting level for the lookup. The permitted SL0 values are:

- \(\text{0b0}\) Stage 2 translation lookup must start at level 2.
- \(\text{0b1}\) Stage 2 translation lookup must start at level 1.

In addition, VTCR.T0SZ must indicate the required input address range. The size of the input address region is \(2^{(32-\text{T0SZ})}\) bytes.
Note

VTCR.T0SZ holds a four-bit signed integer value, meaning it supports values from -8 to 7. This is different from the other translation control registers, where TnSZ holds a three-bit unsigned integer, supporting values from 0 to 7.

The programming of VTCR must follow the constraints shown in Table G5-5, otherwise any attempt to perform a translation table walk that uses the stage 2 address translation generates a stage 2 level 1 Translation Fault. The table also shows how the VTCR.SL0 and VTCR.T0SZ values determine the VTTBR.BADDR field width.

Note

If VTCR.SL0 is programmed to a reserved value then the constraints shown in Table G5-5 are not met, and a translation table walk that uses stage 2 translation generates a stage 2 level 1 Translation fault.

Table G5-5 Input address range constraints on programming VTCR

<table>
<thead>
<tr>
<th>VTCR.SL0</th>
<th>VTCR.T0SZ</th>
<th>Input address range, R</th>
<th>Initial lookup level</th>
<th>BADDR[39:x] width a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>2 to 7</td>
<td>R ≤ 2^30 bytes</td>
<td>Level 2</td>
<td>[39:12] to [39:7]</td>
</tr>
<tr>
<td>0b00</td>
<td>-2 to 1</td>
<td>2^30 &lt; R ≤ 2^34 bytes</td>
<td>Level 2</td>
<td>[39:16] to [39:13]</td>
</tr>
<tr>
<td>0b01</td>
<td>-2 to 1</td>
<td>Level 1</td>
<td></td>
<td>[39:7] to [39:4]</td>
</tr>
<tr>
<td>0b01</td>
<td>-8 to -3</td>
<td>2^34 &lt; R</td>
<td>Level 1</td>
<td>[39:13] to [39:8]</td>
</tr>
</tbody>
</table>

a. The first range corresponds to the first T0SZ value, the second range to the second T0SZ value.

In addition, VTCR.S must be programmed to the value of T0SZ[3], otherwise behavior is CONSTRAINED UNPREDICTABLE with the resulting behavior being that VTCR.T0SZ is treated as an UNKNOWN value.

Note

VTCR.T0SZ being treated as an UNKNOWN value results in a stage 2 level 1 Translation Fault if that UNKNOWN value is not consistent with the programmed value of VTCR.SL0.

CONSTRAINED UNPREDICTABLE behaviors associated with the VTCR on page K1-11581 describes these CONSTRAINED UNPREDICTABLE behaviors.

Where necessary, the initial lookup level provides multiple concatenated translation tables, as described in Use of concatenated level 2 translation tables on page G5-9185. This section also gives more information about the alternatives, shown in Table G5-5, when R is in the range 2^31-2^34.

G5.5.8 The algorithm for finding the translation table entries, VMSAv8-32 Long-descriptor format

This section gives the algorithm for finding the translation table entry that corresponds to a given IA, for each required level of lookup. The algorithm encodes the descriptions of address translation given earlier in this section. The VMSAv8-32 Long-descriptor format uses a 4KB translation granule.

The description uses the following terms:

BaseAddr

The base address for the level of lookup, as defined by:

- For the initial lookup level, the TTBR.BADDR base address field in the appropriate TTBR, see the description of TnSZ on page G5-9191.
- Otherwise, the translation table address returned by the previous level of lookup.

IA

The supplied IA for this stage of translation.
### The translation table size for this stage of translation:

**For PL1&0 stage 1**
- Either:
  - TTBCR.T0SZ if the translation is using TTBR0.
  - TTBCR.T1SZ if the translation is using TTBR1.

**For PL1&0 stage 2**
- VTCR.T0SZ. The translation uses VTTBR.

**For EL2 stage 1**
- HTCR.T0SZ. The translation uses HTTBR.

### SL0
- VTCR.SL0. Applies to the Non-secure PL1&0 stage 2 translation only.

Table G5-6 on page G5-9191 shows the Translation Table descriptor address, for each level of lookup. The table shows only architecturally-valid programming of the TCR. See also *Possible errors in programming the translation table registers* on page G5-9183.

#### Table G5-6 Translation table entry addresses, VMSAv8-32 using Long-descriptor format

<table>
<thead>
<tr>
<th>Lookup level</th>
<th>Entry address and conditions</th>
<th>Stage 2 translation</th>
<th>General conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>One BaseAddr[39:x]:IA[y:30]:0b000 if 0 ≤ TnSZ ≤ 1 then x = (5 - TnSZ)</td>
<td>BaseAddr[39:x]:IA[y:30]:0b000 if SL0b == 1 then if -8 ≤ T0SZ ≤ 1 then x = (5 - T0SZ)</td>
<td>y = (x + 26)</td>
<td></td>
</tr>
<tr>
<td>Two BaseAddr[39:x]:IA[y:21]:0b000 if 2 ≤ TnSZ ≤ 7 then x = (14 - TnSZ)</td>
<td>BaseAddr[39:x]:IA[y:21]:0b000 if SL0 == 0 then if -2 ≤ T0SZ ≤ 7 then x = (14 - T0SZ) elsif SL0b == 1 then x = 12</td>
<td>y = (x + 17)</td>
<td></td>
</tr>
<tr>
<td>Three BaseAddr[39:12]:IA[20:12]:0b000</td>
<td>BaseAddr[39:12]:IA[20:12]:0b000</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

- This line indicates the range of permitted values for TnSZ, for a lookup that starts at this level, see *Use of concatenated translation tables for the initial stage 2 lookup* on page G5-9185.
- SL0 == 0 if the initial lookup is level 2, SL0 == 1 if the initial lookup is level 1.
- This is the case where this level of lookup is not the initial level of lookup.
G5.6 Memory access control

In addition to an output address, a translation table entry that refers to page or region of memory includes fields that define properties of the target memory region. Information returned by a translation table lookup on page G5-9159 describes the classification of those fields as address map control, access control, and memory attribute fields. The access control fields, described in this section, determine whether the PE, in its current state, is permitted to perform the required access to the output address given in the Translation Table descriptor. If a translation stage does not permit the access, then an MMU fault is generated for that translation stage, and no memory access is performed.

The following sections describe the memory access controls:

• About access permissions.
• About the PAN bit on page G5-9195.
• Access permissions for instruction execution on page G5-9196.
• Domains, Short-descriptor format only on page G5-9199.
• The Access flag on page G5-9200.
• Hyp mode control of Non-secure access permissions on page G5-9201.

G5.6.1 About access permissions

The Translation Table descriptors include fields that define access permissions for data accesses and for instruction fetches. This section introduces those fields. In addition:

• System register controls can prevent execution from writable locations, see Preventing execution from writable locations on page G5-9198.

• In Armv8.1, the PSTATE.PAN can affect the access permissions for privileged data accesses, see About the PAN bit on page G5-9195.

Note

This section gives a general description of memory access permissions. Software executing at PL1 in Non-secure state can see only the access permissions defined by the Non-secure PL1&0 stage 1 translations. However, software executing at EL2 can modify these permissions, as described in Hyp mode control of Non-secure access permissions on page G5-9201. This modification is invisible to Non-secure software executing at EL1 or EL0.

Access permission bits in a Translation Table descriptor control access to the corresponding memory region. The details of this control depend on the translation table format, as follows:

Short-descriptor format

This format supports two options for defining the access permissions:

• Three bits, AP[2:0], define the access permissions.
• Two bits, AP[2:1], define the access permissions, and AP[0] can be used as an Access flag. SCTLR.AFE selects the access permissions option. Setting this bit to 1, to enable the Access flag, also selects use of AP[2:1] to define access permissions.

Arm deprecates any use of the AP[2:0] scheme for defining access permissions.

Long-descriptor format

AP[2:1] to control the access permissions, and the descriptors provide an AF bit for use as an Access flag. This means VMSAv8-32 behaves as if the value of SCTLR.AFE is 1, regardless of the value that software has written to this bit.

Note

When use of the Long-descriptor format is enabled, SCTLR.AFE is UNK/SBOP.

The Access flag on page G5-9200 describes the Access flag, for both translation table formats.

The XN and PXN bits provide additional access controls for instruction fetches, see Access permissions for instruction execution on page G5-9196.
An attempt to perform a memory access that the translation table access permission bits do not permit generates a Permission fault, for the corresponding stage of translation. However, when using the Short-descriptor translation table format, it generates the fault only if the access is to memory in the Client domain, see *Domains, Short-descriptor format only* on page G5-9199.

--- Note ---

For the Non-secure PL1&0 translation regime, memory accesses are subject to two stages of translation. Each stage of translation has its own, independent, fault checking. Fault handling is different for the two stages, see *Exception reporting in a VMSAv8-32 implementation* on page G5-9251.

The following sections describe the two access permissions models:

- **AP[2:1] access permissions model**
- **AP[2:0] access permissions control, Short-descriptor format only** on page G5-9194. This section includes some information on access permission control in earlier versions of the Arm VMSA.

### AP[2:1] access permissions model

--- Note ---

Arm recommends that this model is always used, even where the AP[2:0] model is permitted. Some documentation describes the AP[2:1] model as the simplified access permissions model.

This access permissions model is used if the translation is either:

- Using the Long-descriptor translation table format.
- Using Short-descriptor translation table format, and the SCTLR.AFE bit is set to 1.

In this model:

- One bit, AP[2], selects between read-only and read/write access.
- A second bit, AP[1], selects between Application level (EL0) and System level (PL1) control.
  
  For the Non-secure EL2 stage 1 translations, AP[1] is SBO.

This provides four access combinations:

- Read-only at all privilege levels.
- Read/write at all privilege levels.
- Read-only at PL1, no access by software executing at EL0.
- Read/write at PL1, no access by software executing at EL0.

Table G5-7 shows this access control model.

<table>
<thead>
<tr>
<th>AP[2], disable write access</th>
<th>AP[1], enable unprivileged access</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0a</td>
<td>Read/write, only at PL1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Read/write, at any privilege level</td>
</tr>
<tr>
<td>1</td>
<td>0a</td>
<td>Read-only, only at PL1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Read-only, at any privilege level</td>
</tr>
</tbody>
</table>

a. Not valid for Non-secure EL2 stage 1 translation tables. AP[1] is SBO in these tables.

**Hierarchical control of access permissions, Long-descriptor format**

The Long-descriptor translation table format introduces a mechanism that entries at one level of translation table lookup can use to set limits on the permitted entries at subsequent levels of lookup. This applies to the access permissions, and also to the restrictions on instruction fetching described in *Hierarchical control of instruction fetching, Long-descriptor format* on page G5-9198.
The AArch32 Virtual Memory System Architecture

G5.6 Memory access control

The restrictions apply only to subsequent levels of lookup at the same stage of translation. The APTable[1:0] field restricts the access permissions, as Table G5-8 on page G5-9194 shows.

However, in an implementation that includes FEAT_AA32HPD, when hierarchical control of data access permissions is disabled for a translation regime, the information in this subsection does not apply. See Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors on page G5-9176.

As stated in the table footnote, for the Non-secure EL2 stage 1 translation tables, APTable[0] is reserved, SBZ.

<table>
<thead>
<tr>
<th>APTable[1:0]</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No effect on permissions in subsequent levels of lookup.</td>
</tr>
<tr>
<td>01(^a)</td>
<td>Access at EL0 not permitted, regardless of permissions in subsequent levels of lookup.</td>
</tr>
<tr>
<td>10</td>
<td>Write access not permitted, at any Exception level, regardless of permissions in subsequent levels of lookup.</td>
</tr>
<tr>
<td>11(^a)</td>
<td>Regardless of permissions in subsequent levels of lookup:</td>
</tr>
<tr>
<td></td>
<td>• Write access not permitted, at any Exception level.</td>
</tr>
<tr>
<td></td>
<td>• Read access not permitted at EL0.</td>
</tr>
</tbody>
</table>

\(^a\) Not valid for the Non-secure EL2 stage 1 translation tables. In those tables, APTable[0] is SBZ.

---

**Note**

The APTable[1:0] settings are combined with the translation table access permissions in the Translation Tables descriptors accessed in subsequent levels of lookup. They do not restrict or change the values entered in those descriptors.

The Long-descriptor format provides APTable[1:0] control only for the stage 1 translations. The corresponding bits are SBZ in the stage 2 Translation Table descriptors.

The effect of APTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore, a change to APTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.

**AP[2:0] access permissions control, Short-descriptor format only**

This access permissions model applies when using the Short-descriptor translation tables format, and the SCTLR.AFE bit is set to 0. Arm deprecates any use of this access permissions model.

When SCTLR.AFE is set to 0, ensuring that the AP[0] bit is always set to 1 effectively changes the access model to the simpler model described in AP[2:1] access permissions model on page G5-9193.

Table G5-9 on page G5-9194 shows the full AP[2:0] access permissions model:

<table>
<thead>
<tr>
<th>AP[2]</th>
<th>AP[1:0]</th>
<th>PL1 access</th>
<th>Unprivileged access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>No access</td>
<td>No access</td>
<td>All accesses generate Permission faults</td>
</tr>
<tr>
<td>01</td>
<td>Read/write</td>
<td>No access</td>
<td>Access only at PL1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Read/write</td>
<td>Read-only</td>
<td>Writes at EL0 generate Permission faults</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Read/write</td>
<td>Read/write</td>
<td>Full access</td>
<td></td>
</tr>
</tbody>
</table>
The AArch32 Virtual Memory System Architecture
G5.6 Memory access control

Note

- VMSAv8-32 supports the full set of access permissions shown in Table G5-9 on page G5-9194 only when SCTLR.AFE is set to 0. When SCTLR.AFE is set to 1, the only supported access permissions are those described in AP[2:1] access permissions model on page G5-9193.


G5.6.2 About the PAN bit

When the value of PSTATE.PAN is 1, any privileged data access from PL1 or EL2 to a virtual memory address that is accessible at EL0 generates a Permission fault.

When the value of PSTATE.PAN is 0, the translation system is the same as in Armv8.0.

A corresponding PAN bit is added to CPSR and SPSR for exception returns, and DSPSR for entry to and exit from Debug state.

A new SPAN bit is added to SCTLR that controls whether the PAN state bit is set on taking an exception to EL1 from either Secure or Non-secure state, or to EL3 from Secure state when EL3 is using AArch32.

CPSR.PAN bit can be written using an MSR instruction at PL1 or higher. Data writes to CPSR.PAN using an MSR instruction at EL0 are ignored. The value that is returned for an MSR instruction of CPSR from EL0 is UNKNOWN. In keeping with all other writes to the CPSR, other than for instruction fetches, the effect of the PAN state bit does not need to be explicitly synchronized.

The PAN state bit has no effect on:
- Data Cache instructions.
- Address translation instructions, other than ATS1CPRP and ATS1CPWP when FEAT_PAN2 is implemented.
- Unprivileged instructions, LDRBT, LDRHT, LDRT, LDRSBT, LDRSHT, STRBT, STRHT, STRT, STRSBT, and STRSHT, unless HCR_EL2.{E2H, TGE} == {1, 0}.
- Instruction accesses.
- Manager domains.

The PAN bit has no effect when the first stage of translation is disabled for the current translation regime or when the first stage of translation for the current translation regime does not describe the permissions for access at EL0.

If access is disabled, then the access will give rise to a stage 1 Permission fault.

On an exception taken from AArch32:
- CPSR.PAN is copied to SPSR_ELx.PAN, when the target Exception level is AArch64.
- CPSR.PAN is copied to SPSR.PAN, when the target Exception level is AArch32.

On an exception return from AArch32 to AArch32, SPSR.PAN is copied to CPSR.PAN.

On entry to Debug state, CPSR.PAN is copied to DSPSR.PAN.
On exit from Debug state, DSPSR.PAN is copied to CPSR.PAN.

The CPSR.PAN bit is not an Execution state bit.

--- Note ---

- In Non-debug state, in AArch32 state, software can use the SETPAN #imm instruction to modify PSTATE.PAN.
- In Debug state, in AArch32 state, a debugger can use the ERET instruction to perform a DRPS operation to modify PSTATE.PAN.

### G5.6.3 Access permissions for instruction execution

Execute-never controls provide an additional level of control on memory accesses permitted by the access permissions settings. These controls are:

**XN, Execute-never**

Descriptor bit[54], defined as XN for:

- Stage 1 of any translation regime.
- Stage 2 translations when FEAT_XNX is not implemented.

--- Note ---

XN[1:0], Execute-never, stage 2 only describes the stage 2 control when FEAT_XNX is implemented.

This field applies to execution at any Exception level to which the stage of translation applies. A value of 0 indicates that this control permits execution.

**PXN, Privileged execute-never, stage 1 only**

Descriptor bit[53], used only for stage 1 of any translation regime for which the stage 1 translation can support two VA ranges:

- For stage 1 of a translation regime for which the stage 1 translation supports only a single VA range the stage 1 descriptors define a PXN field that is RES0, meaning it is ignored by hardware.

This field applies only to execution at an Exception level higher than EL0. A value of 0 indicates that this control permits execution.

**XN[1:0], Execute-never, stage 2 only**

Descriptor bits[54:53], defined as XN[1:0] for:

- Stage 2 translations when FEAT_XNX is implemented.

Table G5-10 on page G5-9196 shows the operation of this control.

<table>
<thead>
<tr>
<th>XN[1]</th>
<th>XN[0]</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>The stage 2 control permits execution at EL1 and EL0 if read access is permitted</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>The stage 2 control does not permit execution at EL1, but permits execution at EL0 if read access is permitted</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>The stage 2 control does not permit execution at EL1 or at EL0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>The stage 2 control permits execution at EL1 if read access is permitted, but does not permit execution at EL0</td>
</tr>
</tbody>
</table>
For stage 2 translations when FEAT_XNX is not implemented, descriptor bit[53] is RES0, meaning it is ignored by hardware, and bit[54] is the XN control, see XN, Execute-never on page G5-9196.

Executing an instruction at ELx in a particular Security state generates a Permission fault unless all of the following are true for the instruction address:

- Any stage 1 execute-never control that applies to execution at ELx in the current Security state permits execution.
- If the translation regime that applies to ELx in the current Security state has two stages of translations, the stage 2 execute-never control that applies to execution at ELx permits execution.
- Read access is permitted.

However, if a stage 1 translation is using the Short-descriptor translation table format and the address is in a Managers domain the stage 1 access permissions are not checked, and therefore the access cannot cause a stage 1 Permission fault, see Domains, Short-descriptor format only on page G5-9199.

See also Hyp mode control of Non-secure access permissions on page G5-9201.

In addition, System register controls can enforce execute-never restrictions, regardless of the settings in the translation table XN and PXN fields, see:
- Restriction on Secure instruction fetch on page G5-9199.
- Preventing execution from writable locations on page G5-9198.

The execute-never controls apply also to speculative instruction fetching. This means a speculative instruction fetch from a memory region that is execute-never at the current level of privilege is prohibited.

The execute-never controls means that, when the stage of address translation is enabled, the PE can fetch, or speculatively fetch, an instruction from a memory location only if all of the following apply:

- If using the Short-descriptor translation table format, the Translation Table descriptor for the location does not indicate that it is in a No access domain.
- If using the Long-descriptor translation table format, or using the Short descriptor format and the descriptor indicates that the location is in a Client domain, in the descriptor for the location the following apply:
  - The stage 1 execute-never control for the Exception level at which the instruction is executed permits execution.
  - For a translation regime with two stages of address translation, the stage 2 execute-never control that applies to the Exception level at which the instruction is executed permits execution.
  - The access permissions permit a read access from the current PE mode.
- No other Prefetch Abort condition exists.

The PXN control applies to the PE privilege when it attempts to execute the instruction. In an implementation that fetches instructions speculatively, this might not be the privilege when the instruction was prefetched. Therefore, the architecture does not require the PXN control to prevent instruction fetching.

Although the XN control applies to speculative fetching, on a speculative instruction fetch from an XN location, no Permission fault is generated unless the PE attempts to execute the instruction that would have been fetched from that location. This means that, if a speculative fetch from an XN location is attempted, but there is no attempt to execute the corresponding instruction, a Permission fault is not generated.

The software that defines a translation table must mark any region of memory that is read-sensitive as XN, to avoid the possibility of a speculative fetch accessing the memory region. This means it must mark any memory region that corresponds to a read-sensitive peripheral as XN. Hardware does not prevent speculative accesses to a region of any Device memory type unless that region is also marked as execute-never for all Exception levels from which it can be accessed.
• When using the Short-descriptor translation table format, the XN attribute is not checked for domains marked as Manager. Therefore, the system must not include read-sensitive memory in domains marked as Manager, because the XN field does not prevent speculative fetches from a Manager domain.

When no stage of address translation for the translation regime is enabled, memory regions cannot have XN or PXN attributes assigned. Behavior of instruction fetches when all associated address translations are disabled on page G5-9156 describes how disabling all MMUs affects instruction fetching.

Hierarchical control of instruction fetching, Long-descriptor format

The Long-descriptor translation table format introduces a mechanism that means entries at one level of translation tables lookup can set limits on the permitted entries at subsequent levels of lookup. This applies to the restrictions on instruction fetching, and also to the access permissions described in Hierarchical control of access permissions, Long-descriptor format on page G5-9193.

--- Note ---
Similar hierarchical controls apply to data accesses, see Hierarchical control of access permissions, Long-descriptor format on page G5-9193.

However, in an implementation that includes FEAT-AA32HPD, when hierarchical control of instruction fetching is disabled for a translation regime, the information in this subsection does not apply. See Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors on page G5-9176.

The restrictions apply only to subsequent levels of lookup at the same stage of translation, and:

• XNTable restricts the XN control:
  — When XNTable is set to 1, the XN field is treated as 1 in all subsequent levels of lookup, regardless of the actual value of the field.
  — When XNTable is set to 0 it has no effect.

• PXNTable restricts the PXN control:
  — When PXNTable is set to 1, the PXN field is treated as 1 in all subsequent levels of lookup, regardless of the actual value of the field.
  — When PXNTable is set to 0 it has no effect.

--- Note ---
The XNTable and PXNTable settings are combined with the XN and PXN fields in the Translation Table descriptors accessed at subsequent levels of lookup. They do not restrict or change the values entered in those descriptors.

The XNTable and PXNTable controls are provided only in the Long-descriptor translation table format, and only for stage 1 translations. The corresponding bits are SBZ in the stage 2 Translation Table descriptors.

The effect of XNTable or PXNTable applies to later entries in the translation table walk, and so its effects can be held in one or more TLB entries. Therefore, a change to XNTable or PXNTable requires coarse-grained invalidation of the TLB to ensure that the effect of the change is visible to subsequent memory transactions.

Preventing execution from writable locations

The architecture provides control bits that, when the corresponding stage 1 address translation is enabled, force writable memory to be treated as XN or PXN, regardless of the value of the XN or PXN field. When the translation stages are controlled by an Exception level that is using AArch32:

• For PL1&0 stage 1 translations, when SCTLR.WXN is set to 1, all regions that are writable at stage 1 of the address translation are treated as XN.

• For PL1&0 stage 1 translations, when SCTLR.UWXN is set to 1, an instruction fetch is treated as accessing a PXN region if it accesses a region that software executing at EL0 can write to.
• For Non-secure EL2 stage 1 translations, when HSCTRL.WXN is set to 1, all regions that are writable at stage 1 of the address translation are treated as XN.

________ Note _______
• The SCTLR.WXN controls are intended to be used in systems with very high security requirements.
• Setting a WXN or UWXN bit to 1 changes the interpretation of the translation table entry, overriding a zero value of an XN or PXN field. It does not cause any change to the translation table entry.

For any given virtual machine, Arm expects WXN and UWXN to remain static in normal operation. In particular, it is IMPLEMENTATION DEFINED whether TLB entries associated with a particular VMID reflect the effect of the values of these fields. A generic sequence to ensure synchronization of a change to these fields, when that change is made without a corresponding change of VMID, is:

Change the WXN or UWXN bit
ISB ; This ensures synchronization of the change
Invalidate entire TLB of associated entries
DSB ; This completes the TLB Invalidation
ISB ; This ensures instruction synchronization

As with all Permission fault checking, if the stage 1 translation is using the Short-descriptor translation table format, the permission checks are performed only for Client domains. For more information, see About access permissions on page G5-9192.

For more information about address translation, see About address translation for VMSAv8-32 on page G5-9149.

Restriction on Secure instruction fetch
EL3 provides a Secure instruction fetch bit, SCR.SIF. When this bit is 1, any attempt in Secure state to execute an instruction fetched from Non-secure physical memory causes a Permission fault. As with all Permission fault checking, when using the Short-descriptor format translation tables the check applies only to Client domains, see About access permissions on page G5-9192.

Arm expects SCR.SIF to be static during normal operation. In particular, whether the TLB holds the effect of the SIF bit is IMPLEMENTATION DEFINED. The generic sequence to ensure visibility of a change to the SIF bit is:

Change the SCR.SIF bit
ISB ; This ensures synchronization of the change
Invalidate entire TLB
DSB ; This completes the TLB Invalidation
ISB ; This ensures instruction synchronization

G5.6.4 Domains, Short-descriptor format only
A domain is a collection of memory regions. The Short-descriptor translation table format supports 16 domains, and requires the software that defines a translation table to assign each VMSAv8-32 memory region to a domain. When using the Short-descriptor format:

• Level 1 translation table entries for translation tables and Sections include a domain field.
• Translation table entries for Supersections do not include a domain field. The Short-descriptor format defines Supersections as being in domain 0.
• Level 2 translation table entries inherit a domain setting from the parent level 1 Translation Table descriptor.
• Each TLB entry includes a domain field.

The domain field specifies which of the 16 domains the entry is in, and a two-bit field in the DACR defines the permitted access for each domain. The possible settings for each domain are:

No access Any access using the Translation Table descriptor generates a Domain fault.
Clients On an access using the Translation Table descriptor, the access permission attributes are checked. Therefore, the access might generate a Permission fault.
Managers

On an access using the Translation Table descriptor, the access permission attributes are not checked. Therefore, the access cannot generate a Permission fault.

See The MMU fault-checking sequence on page G5-9242 for more information about how, when using the Short-descriptor translation table format, the Domain attribute affects the checking of the other attributes in the Translation Table descriptor.

Note

A single program might:

• Be a Client of some domains.
• Be a Manager of some other domains.
• Have no access to the remaining domains.

The Long-descriptor translation table format does not support domains. When a stage of translation is using this format, all memory is treated as being in a Client domain, and the settings in the DACR are ignored.

G5.6.5 The Access flag

The Access flag indicates when a page or section of memory is accessed for the first time since the Access flag in the corresponding Translation Table descriptor was set to 0:

• If address translation is using the Short-descriptor translation table format, it must set SCTLR.AFE to 1 to enable use of the Access flag. Setting this bit to 1 redefines the AP[0] bit in the Translation Table descriptors as an Access flag, and limits the access permissions information in the Translation Table descriptors to AP[2:1], as described in AP[2:1] access permissions model on page G5-9193.

• The Long-descriptor format always supports an Access flag bit in the Translation Table descriptors, and address translation using this format behaves as if SCTLR.AFE is set to 1, regardless of the value of that bit.

In Armv8.0, the Access flag is managed by software as described in Software management of the Access flag.

Note

Previous versions of the Arm architecture optionally supported hardware management of the Access flag. Armv8.0 obsolete this option. However, FEAT_HAFDBS provides a new mechanism for hardware management of the Access flag, that is supported only for the VMSAv8-64 translation regimes.

Software management of the Access flag

Armv8.0 requires that software manages the Access flag. This means an Access flag fault is generated whenever an attempt is made to read into the TLB a Translation Table descriptor entry for which the value of the Access flag is 0.

Note

When using the Short-descriptor translation table format, Access flag faults are generated only if SCTLR.AFE is set to 1, to enable use of a Translation Table descriptor bit as an Access flag.

The Access flag mechanism expects that, when an Access flag fault occurs, software resets the Access flag to 1 in the translation table entry that caused the fault. This prevents the fault occurring the next time that memory location is accessed. Entries with the Access flag set to 0 are never held in the TLB, meaning software does not have to flush the entry from the TLB after setting the flag.

Note

If a system incorporates components that can autonomously update translation table entries that are shared with the Arm PE, then the software must be aware of the possibility that such components can update the access flag autonomously.
In such a system, system software should perform any changes of translation table entries with an Access flag of 0, other than changes to the Access flag value, by using an Load-Exclusive/Store-Exclusive loop, to allow for the possibility of simultaneous updates.

G5.6.6 Hyp mode control of Non-secure access permissions

When EL2 is using AArch32, Non-secure software executing in Hyp mode controls two sets of translation tables, both of which use the Long-descriptor translation table format:

- The translation tables that control the Non-secure EL2 stage 1 translations. These map VAs to PAs, for memory accesses made when executing in Non-secure state in Hyp mode, and are indicated and controlled by the HTTBR and HTCR.
  
  These translations have similar access controls to other Non-secure stage 1 translations using the Long-descriptor translation table format, as described in:
  - Access permissions for instruction execution on page G5-9196.

  The differences from the Non-secure stage 1 translations are that:
  - The APTable[0], PXNTable, and PXN bits are reserved, SBZ.
  - AP[1] is reserved, SBO.

- The translation tables that control the Non-secure PL1&0 stage 2 translations. These map the IPAs from the stage 1 translation onto PAs, for memory accesses made when executing in Non-secure state at PL1 or EL0, and are indicated and controlled by the VTTBR and VTCR.

  The descriptors in the virtualization translation tables define stage 2 access permissions, that are combined with the permissions defined in the stage 1 translation. This section describes this combining of access permissions.

  **Note**

  The level 2 access permissions mean a hypervisor can define additional access restrictions to those defined by a Guest OS in the stage 1 translation tables. For a particular access, the actual access permission is the more restrictive of the permissions defined by:
  - The Guest OS, in the stage 1 translation tables.
  - The hypervisor, in the stage 2 translation tables.

  The stage 2 access controls defined from Hyp mode:
  - Affect only the Non-secure stage 1 access permissions settings.
  - Take no account of whether the accesses are from a Non-secure PL1 mode or a Non-secure EL0 mode.
  - Permit software executing in Hyp mode to assign a write-only attribute to a memory region.

  The S2AP field in the stage 2 descriptors define the stage 2 access permissions, as Table G5-11 on page G5-9201 shows:

<table>
<thead>
<tr>
<th>S2AP</th>
<th>Access permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No access permitted</td>
</tr>
<tr>
<td>01</td>
<td>Read-only. Writes to the region are not permitted, regardless of the stage 1 permissions.</td>
</tr>
<tr>
<td>10</td>
<td>Write-only. Reads from the region are not permitted, regardless of the stage 1 permissions.</td>
</tr>
<tr>
<td>11</td>
<td>Read/write. The stage 1 permissions determine the access permissions for the region.</td>
</tr>
</tbody>
</table>

For more information about the S2AP field, see Attribute fields in VMSAv8-32 Long-descriptor stage 2 Block and Page descriptors on page G5-9179.
If the stage 2 permissions cause a Permission fault, this is a stage 2 MMU fault. Stage 2 MMU faults are taken to Hyp mode, and reported in the HSR using an EC code of 0x20 or 0x24. For more information, see Use of the HSR on page G5-9265.

——— Note ————
In the HSR, the combination of the EC code and the DFSC or IFSC value in the ISS indicate that the fault is a stage 2 MMU fault.

——— Note ————
The stage 2 permissions include an XN attribute. If this identifies the region as execute-never, execution from the region is not permitted, regardless of the value of the XN or UXN attribute in the stage 1 translation. If a Permission fault is generated because the stage 2 XN field identifies the region as execute-never, this is reported as a stage 2 MMU fault.

——— Note ————
The stage 2 XN attribute:

- Is a single bit if FEAT_XNX is not implemented, see XN, Execute-never on page G5-9196.
- Is a 2-bit field if FEAT_XNX is implemented, see XN[1:0], Execute-never, stage 2 only on page G5-9196.

AArch32 state prioritization of synchronous aborts from a single stage of address translation on page G5-9248 describes the abort prioritization if both stages of a translation generate a fault.
Memory region attributes

In addition to an output address, a translation table entry that refers to a page or region of memory includes fields that define properties of that target memory region. Information returned by a translation table lookup describes the classification of those fields as address map control, access control, and memory attribute fields. The memory region attribute fields control the memory type, Cacheability, and Shareability of the region.

The following sections describe the assignment of memory region attributes for stage 1 translations:

- Overview of memory region attributes for stage 1 translations.
- Short-descriptor format memory region attributes, without TEX remap on page G5-9204.
- Short-descriptor format memory region attributes, with TEX remap on page G5-9207.
- VMSAv8-32 Long-descriptor format memory region attributes on page G5-9210.

For an implementation that is operating in Secure state, or in Hyp mode, these assignments define the memory attributes of the accessed region.

For an implementation that is operating in a Non-secure PL1 or EL0 mode, the Non-secure PL1&0 stage 2 translation can modify the memory attributes assigned by the stage 1 translation. EL2 control of Non-secure memory region attributes on page G5-9212 describes these stage 2 assignments.

Overview of memory region attributes for stage 1 translations

The description of the memory region attributes in a Translation descriptor divides into:

Memory type and attributes

These are described either:

- Directly, by bits in the Translation Table descriptor.
- Indirectly, by registers referenced by bits in the Table descriptor. This is described as remapping the memory type and attribute description.

The Short-descriptor translation table format can use either of these approaches, selected by the SCTLR.TRE bit:

- **TRE == 0** Remap disabled. The TEX[2:0], C, and B bits in the Translation Table descriptor define the memory region attributes. Short-descriptor format memory region attributes, without TEX remap on page G5-9204 describes this encoding.

--- Note ---

With the Short-descriptor format, remapping is called TEX remap, and the SCTLR.TRE bit is the TEX remap enabled bit.

The description of the TRE == 0 encoding includes information about the encoding in previous versions of the architecture.

- **TRE == 1** Remap enabled. The TEX[0], C, and B bits in the Translation Table descriptor are index bits to the remap registers, that define the memory region attributes:
  - The Primary Region Remap Register, PRRR.
  - The Normal Memory Remap Register, NMRR.

Short-descriptor format memory region attributes, with TEX remap on page G5-9207 describes this encoding scheme.

This scheme reassigns Translation Table descriptor bits TEX[2:1] for use as bits managed by the operating system.

The Long-descriptor translation table format always uses remapping. This means that when the value of TTBCR.EAE is 1, enabling use of the Long-descriptor translation table format, SCTLR.TRE is **RES1**.

VMSAv8-32 Long-descriptor format memory region attributes on page G5-9210 describes this encoding.
Shareability

In the Short-descriptor translation table format, the S bit in the Translation Table descriptor is used in determining the Shareability of the region. How the S bit is interpreted depends on whether TEX remap is enabled, see:

- **Shareability and the S bit, without TEX remap** on page G5-9206.
- **Determining the Shareability, with TEX remap** on page G5-9208.

In the Long-descriptor translation table format, the SH[1:0] field in the Translation Table descriptor encodes the Shareability of the region, see **Shareability, Long-descriptor format** on page G5-9210.

--- Note ---

Shareability is one of Non-shareable, Inner Shareable, and Outer Shareable. However, when using the Short-descriptor translation table format without TEX remap, VMSAv8-32 does not support any distinction between Inner Shareable and Outer Shareable memory, and a memory region is either Non-shareable or Outer Shareable.

### Stage 1 definition of the XS attribute

When **FEAT_XS** is implemented, all stage 1 memory types defined in the MAIR0, MAIR1, HMAIR0, HMAIR1, PRRR, and NMRR registers, or the TTBCR or HTCR registers, or in the page tables, have the XS attribute set to 1, unless they are Inner Write-Back Cacheable, Outer Write-back Cacheable, which have the XS attribute set to 0. This includes any memory types that are treated as Write-Back Cacheable as a result of **IMPLEMENTATION DEFINED** choices in the architecture.

### G5.7.2 Short-descriptor format memory region attributes, without TEX remap

When using the Short-descriptor translation table formats, TEX remap is disabled when the value of **SCTLR.TRE** is 0.

--- Note ---

- The Short-descriptor format scheme without TEX remap is the scheme used in VMSAv6.
- The B (Bufferable), C (Cacheable), and TEX (Type extension) bit names are inherited from earlier versions of the architecture. These names no longer adequately describe the function of the B, C, and TEX bits.

--- Table **G5-12** on page G5-9204 shows the C, B, and TEX[2:0] encodings when TEX remap is disabled. In the **Page Shareability** on page G5-9204 column, an entry of S bit indicates that the S bit in the Translation Table descriptor determines the Shareability, see **Shareability and the S bit, without TEX remap** on page G5-9206.

<table>
<thead>
<tr>
<th>TEX[2:0]</th>
<th>C</th>
<th>B Description</th>
<th>Memory type</th>
<th>Page Shareability</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>Device-nGnRnE</td>
<td>Device-nGnRnE</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Device-nGnREa</td>
<td>Device-nGnRE</td>
<td>Outer Shareablea</td>
</tr>
<tr>
<td>1 0</td>
<td>Outer and Inner Write-Through, Read-Allocate No Write-Allocate</td>
<td>Normal</td>
<td>S bit</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>Outer and Inner Write-Back, Read-Allocate No Write-Allocate</td>
<td>Normal</td>
<td>S bit</td>
<td></td>
</tr>
</tbody>
</table>
### Table G5-12 TEX, C, and B encodings when TRE == 0 (continued)

<table>
<thead>
<tr>
<th>TEX[2:0]</th>
<th>C</th>
<th>B</th>
<th>Description</th>
<th>Memory type</th>
<th>Page Shareability</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>0</td>
<td>0</td>
<td>Outer and Inner Non-cacheable</td>
<td>Normal</td>
<td>Outer Shareable&lt;sup&gt;b&lt;/sup&gt;</td>
</tr>
<tr>
<td>1</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>IMPLEMENTATION DEFINED</td>
<td>IMPLEMENTATION DEFINED</td>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Outer and Inner Write-Back, Read-Allocate Write-Allocate</td>
<td>Normal</td>
<td>S bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>0</td>
<td>Device-nGnRE&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Device-nGnRE</td>
<td>Outer Shareable&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>1</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Cacheability attributes, without TEX remap

When the value of TEX[2] is 0, the same Cacheability attribute applies to Inner Cacheable and Outer Cacheable memory regions, and the \{TEX[1:0], C, B\} values identify this attribute, as Table G5-12 on page G5-9204 shows.

When the value of TEX[2] is 1, the memory described by the translation table entry is cacheable, and the rest of the encoding defines the Inner Cacheability and Outer Cacheability attributes:

- TEX[1:0] Define the Outer Cacheability attribute.
- C, B Define the Inner Cacheability attribute.

The translation table entries use the same encoding for the Outer and Inner Cacheability attributes, as Table G5-13 on page G5-9206 shows.

Shareability and the S bit, without TEX remap

The Short-descriptor format translation table entries include an S bit. This bit:

- Is ignored if the entry refers to any type of Device memory, or to Normal memory that is Inner Non-cacheable, Outer Non-cacheable.
- For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, determines whether the memory region is Outer Shareable or Non-shareable:
  - $S == 0$: Normal memory region is Non-shareable.
  - $S == 1$: Normal memory region is Outer Shareable.

Without TEX remapping there is no distinction between Inner Shareable and Outer Shareable memory, meaning the S bit determines whether the region is Non-shareable or Outer Shareable.
G5.7.3 Short-descriptor format memory region attributes, with TEX remap

When using the Short-descriptor translation table formats, TEX remap is enabled when the value of SCTLR.TRE is 1. In this configuration:

- The software that defines the translation tables must program the PRRR and NMRR to define seven possible memory region attributes.
- The TEX[0], C, and B bits of the Translation Table descriptors define the memory region attributes, by indexing PRRR and NMRR.
- Hardware makes no use of TEX[2:1], see *The OS managed translation table bits* on page G5-9209.

When TEX remap is enabled:

- For seven of the eight possible combinations of the TEX[0], C and B bits, fields in the PRRR and NMRR define the region attributes, as described in this section.
- The meaning of the eighth combination for the TEX[0], C and B bits is IMPLEMENTATION DEFINED.
- If the TEX[0], C and B bits determine that the region is a Device memory type, or is Normal Inner Non-cacheable, Outer Non-cacheable, then the region is Outer Shareable. Otherwise, the Shareability is determined by the combination of:
  - The S bit from the Translation Table descriptor.
  - The value of the PRRR.NS0 or PRRR.NS1 bit.
  - The value of the appropriate PRRR.NOSn bit, as shown in Table G5-14 on page G5-9207.

For more information, see *Determining the Shareability, with TEX remap* on page G5-9208.

For each of the possible encodings of the TEX[0], C, and B bits in a translation table entry, Table G5-14 on page G5-9207 shows which fields of the PRRR and NMRR registers describe the memory region attributes.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Memory type</th>
<th>Cache attributes</th>
<th>Outer Shareable attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>TE[X][0]</td>
<td>C</td>
<td>B</td>
<td>Inner cacheability</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>PRRR.TR0</td>
</tr>
<tr>
<td>1</td>
<td>PRRR.TR1</td>
<td>NMRR.IR1</td>
<td>NMRR.OR1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>PRRR.TR2</td>
</tr>
<tr>
<td>1</td>
<td>PRRR.TR3</td>
<td>NMRR.IR3</td>
<td>NMRR.OR3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>PRRR.TR4</td>
</tr>
<tr>
<td>1</td>
<td>PRRR.TR5</td>
<td>NMRR.IR5</td>
<td>NMRR.OR5</td>
</tr>
<tr>
<td>1</td>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>PRRR.TR7</td>
<td>NMRR.IR7</td>
<td>NMRR.OR7</td>
</tr>
</tbody>
</table>

a. For details of the *Memory type* and *Outer Shareable* encodings see the description of the PRRR. For details of the *Cache attributes* encodings see the description of the NMRR.

b. Applies only if the memory type for the region is mapped as Normal memory.

c. Applies only if both of the following apply:

The memory type for the region is mapped as Normal memory that is not Inner Non-cacheable and Outer Non-cacheable.

The region is not Non-shareable.

See *Determining the Shareability, with TEX remap* on page G5-9208.
As Table G5-14 on page G5-9207 shows, when TEX remap is enabled, for a given set of \{TEX[0], C, B\} bits from a Translation Table descriptor:

1. The primary mapping, to memory type, is given by the PRRR.TRn field as shown in the Memory type column.
2. For any region that the PRRR.TRn maps as Normal memory, NMRR.IRn determines the Inner cacheability attribute, and NMRR.ORn determines the Outer cacheability attribute.
3. For a region that PRRR.TRn maps as Normal memory, if NMRR.{IRn, ORn} do not map the region as Inner Non-cacheable, Outer Non-cacheable, PRRR.{NS0, NS1} and PRRR.NOSn are used to determine the Shareability of the region, see Determining the Shareability, with TEX remap.

The TEX remap registers and the SCTLR.TRE bit are banked between the Secure and Non-secure Security states. For more information, see The effect of EL3 on TEX remap on page G5-9210.

The TEX remap registers must be static during normal operation. In particular, when the remap registers are changed:
- It is IMPLEMENTATION DEFINED when the changes take effect.
- It is CONSTRAINED UNPREDICTABLE whether the TLB caches the effect of the TEX remap on translation tables, see CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values on page K1-11567.

The software sequence to ensure the synchronization of changes to the TEX remap registers is:
1. Execute a DSB instruction. This ensures any memory accesses using the old mapping have completed.
2. Write the TEX remap registers or SCTLR.TRE bit.
3. Execute an ISB instruction. This ensures synchronization of the register updates.
4. Invalidate the entire TLB.
5. Execute a DSB instruction. This ensures completion of the entire TLB operation.
6. Clean and invalidate all caches. This removes any cached information associated with the old mapping.
7. Execute a DSB instruction. This ensures completion of the cache maintenance.
8. Execute an ISB instruction. This ensures instruction synchronization.

This extends the standard rules for the synchronization of changes to System registers described in Synchronization of changes to AArch32 System registers on page G8-9326, and provides implementation freedom as to whether or not the effect of the TEX remap is cached.

### Determining the Shareability, with TEX remap

The memory type of a region, as indicated in the Memory type column of Table G5-14 on page G5-9207, provides the first level of control of the Shareability of the region:
- If the memory is any type of Device memory, then the region is Outer Shareable, and any Shareability attributes in the Translation Table descriptor and PRRR for that region are ignored.
  - This applies also to a Normal memory region that the NMRR attributes identify as Inner Non-cacheable and Outer Non-cacheable,
- If using the Short descriptor translation table format then the Shareability of the region is determined using the value of the S bit in the Translation Table descriptor to index one of the PRRR.{NS1, NS0} bits, as described in this section.

Table G5-15 on page G5-9208 shows how the translation table S bit indexes into the PRRR:

<table>
<thead>
<tr>
<th>Memory type</th>
<th>Remapping when S == 0</th>
<th>Remapping when S == 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device or Normal Inner Non-cacheable, Outer Non-cacheable</td>
<td>Outer Shareable</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td>Normal, not Inner Non-cacheable, Outer Non-cacheable</td>
<td>PRRR.NS0</td>
<td>PRRR.NS1</td>
</tr>
</tbody>
</table>
For a Normal memory region that is not Inner Non-cacheable, Outer Non-cacheable, the appropriate bit of the
PRRR indicates whether the region is Non-shareable, as follows:

\[
\text{PRRR.NS} = 0 \quad \text{Non-shareable.}
\]
\[
\text{PRRR.NS} = 1
\]

The appropriate PRRR.NOS field, as shown in Table G5-14 on page G5-9207, indicates
whether the region is Inner Shareable or Outer Shareable:

\[
\text{PRRR.NOS} = 0 \quad \text{Region is Outer Shareable.}
\]
\[
\text{PRRR.NOS} = 1 \quad \text{Region is Inner Shareable.}
\]

Note
This means that TEX remapping can map a translation table entry with \(S = 0\) as shareable memory.

**SCTLR.TRE, SCTLR.M, and the effect of the TEX remap registers**

When TEX remap is disabled, because the value of the SCTLR.TRE bit is 0:

- The effect of the PRRR and NMRR registers can be IMPLEMENTATION DEFINED.
- The interpretation of the fields of the PRRR and NMRR registers can differ from the description given earlier
  in this section. One implication of this is that the implementation can provide an IMPLEMENTATION DEFINED
  mechanism to interpret the PRRR.NOS fields.

VMSAv8-32 requires that the effect of these registers is limited to remapping the attributes of memory locations.
These registers must not change whether any cache hardware or stages of address translation are enabled. The
mechanism by which the TEX remap registers have an effect when the value of the SCTLR.TRE bit is 0 is
IMPLEMENTATION DEFINED. The AArch32 architecture requires that from reset, if the IMPLEMENTATION DEFINED
mechanism has not been invoked:

- If the PL1&0 stage 1 address translation is enabled and is using the Short-descriptor format translation tables,
  the architecturally-defined behavior of the TEX[2:0], C, and B bits must apply, without reference to the TEX
  remap functionality. In other words, memory attribute assignment must comply with the scheme described
  in Short-descriptor format memory region attributes, without TEX remap on page G5-9204.
- If the PL1&0 stage 1 address translation is disabled, then the architecturally-defined behavior of VMSAv8-32
  with address translation disabled must apply, without reference to the TEX remap functionality. See The
effects of disabling address translation stages on VMSAv8-32 behavior on page G5-9154.

Possible mechanisms for enabling the IMPLEMENTATION DEFINED effect of the TEX remap registers when the value
of SCTLR.TRE is 0 include:

- A control bit in the ACTLR, or in an IMPLEMENTATION DEFINED System register.
- Changing the behavior when the PRRR and NMRR registers are changed from their IMPLEMENTATION
  DEFINED reset values.

In addition, if the stage of address translation is disabled and the value of the SCTLR.TRE bit is 1, the
architecturally-defined behavior of the VMSAv8-32 with the stage of address translation disabled must apply
without reference to the TEX remap functionality.

In an implementation that includes EL3, the IMPLEMENTATION DEFINED effect of these registers must only take
effect in the Security state of the registers. See also The effect of EL3 on TEX remap on page G5-9210.

**The OS managed translation table bits**

When TEX remap is enabled, the TEX[2:1] bits in the Translation Table descriptors are available as two bits that
can be managed by the operating system. In VMSAv8-32, as long as the SCTLR.TRE bit is set to 1, the values of
the TEX[2:1] bits are IGNORED by the PE. Software can write any value to these bits in the translation tables.
The effect of EL3 on TEX remap

In an implementation that includes EL3, when EL3 is using AArch32, the TEX remap registers are banked between
the Secure and Non-secure Security states. When EL3 is using AArch32, write accesses to the Secure register for
the current security state apply to all PL1&0 stage 1 translation table lookups in that state. The SCTLR.TRE bit is
banked in the Secure and Non-secure copies of the register, and the appropriate version of this bit determines
whether TEX remap is applied to translation table lookups in the current security state.

Write accesses to the Secure copies of the TEX remap registers are disabled when the CP15SDISABLE input is asserted HIGH, meaning the MCR operations to access these registers are UNDEFINED. For more information, see The CP15SDISABLE and CP15SDISABLE2 input signals on page G5-9285.

VMSAv8-32 Long-descriptor format memory region attributes

When a PE is using the VMSAv8-32 Long-descriptor translation table format, the AttrIndx[2:0] field in a block or
page Translation Table descriptor for a stage 1 translation indicates the 8-bit field, in the appropriate MAIR register,
that specifies the attributes for the corresponding memory region, as follows:

• AttrIndx[2] indicates the MAIR register to be used:
  AttrIndx[2] == 0 Use MAIR0.

• AttrIndx[2:0] indicates the required Attr field, Attrn, where n = AttrIndx[2:0].

Each AttrIndx field defines, for the corresponding memory region:

• The memory type, Normal or a type of Device memory.

• For Normal memory:
  — The Inner cacheability and Outer cacheability attributes, each of which is one of Non-cacheable,
    Write-Through Cacheable, or Write-Back Cacheable.
  — For Write-Through Cacheable and Write-Back Cacheable regions, the Read-Allocate and
    Write-Allocate policy hints, each of which is Allocate or No allocate.

For more information about the AttrIndx[2:0] descriptor field, see Attribute fields in VMSAv8-32 Long-descriptor
stage 1 Block and Page descriptors on page G5-9177.

Shareability, Long-descriptor format

When a PE is using the Long-descriptor translation table format, the SH[1:0] field in a Block or Page Translation
Table descriptor specifies the Shareability attributes of the corresponding memory region, if the MAIR entry for that
region identifies it as Normal memory that is not both Inner Non-cacheable and Outer Non-cacheable. Table G5-16
on page G5-9210 shows the encoding of this field.

<table>
<thead>
<tr>
<th>SH[1:0]</th>
<th>Normal memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Non-shareable</td>
</tr>
<tr>
<td>01</td>
<td>Reserved, CONstrained UNPREDICTABLE, see Reserved values in System and memory-mapped registers and translation table entries on page K1-11583 for the permitted behavior.</td>
</tr>
<tr>
<td>10</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td>11</td>
<td>Inner Shareable</td>
</tr>
</tbody>
</table>

See Combining the Shareability attribute on page G5-9215 for constraints on the Shareability attributes of a Normal
memory region that is Inner Non-cacheable, Outer Non-cacheable.
For any type of Device memory, and for Normal Inner Non-cacheable, Outer Non-cacheable memory, the value of the SH[1:0] field of the Translation Table descriptor is ignored.

**Other fields in the Long-descriptor translation table format descriptors**

The following subsections describe the other fields in the Translation Table Block and Page descriptors when a PE is using the Long-descriptor translation table format:

- **Contiguous bit**
- **IGNORED fields.**
- **Field reserved for software use**

**Contiguous bit**

The Long-descriptor Translation Table Format descriptors contain a Contiguous bit. Setting this bit to 1 indicates that 16 adjacent translation table entries point to a contiguous output address range. These 16 entries must be aligned in the translation table so that the top five bits of their input addresses, that index their position in the translation table, are the same. For example, to use this bit for a block of 16 entries in the level 3 translation table, bits[20:16] of the input addresses for the 16 entries must be the same.

The contiguous output address range must be aligned to size of 16 translation table entries at the same translation table level.

Use of this bit means that the TLB can cache a single entry to cover the 16 translation table entries.

This bit acts as a hint. The architecture does not require a PE to cache TLB entries in this way. To avoid TLB coherency issues, any TLB maintenance by address must not assume any optimization of the TLB tables that might result from use of this bit.

**Note**

The use of the contiguous bit is similar to the approach used, in the Short-descriptor translation table format, for optimized caching of Large Pages and Supersections in the TLB. However, an important difference in the contiguous bit capability is that TLB maintenance must be performed based on the size of the underlying translation table entries, to avoid TLB coherency issues. That is, any use of the contiguous bit has no effect on the minimum size of entry that must be invalidated from the TLB.

**IGNORED fields**

In the VMSA8-32 translation table long-descriptor format, the following fields are defined as IGNORED, meaning the architecture guarantees that a PE makes no use of these fields:

- In the stage 1 and stage 2 Table descriptors, bits[58:52] and bits[11:2].
- In the stage 1 and stage 2 Block and Page descriptors, bit[63] and bits[58:55].
- In the stage 1 and stage 2 Block and Page descriptors in an implementation that does not include FEAT_HPDS2, bits[62:59].

Of these fields:

- In the stage 1 and stage 2 Block and Page descriptors, bits[58:55] are reserved for software use, see Field reserved for software use.

- In the stage 2 Block and Page descriptors:
  - Bit[63] is reserved for use by a System MMU.
  - In an implementation that does not include FEAT_HPDS2, bits[62:59] are reserved for use by a System MMU.

**Field reserved for software use**

The architecture reserves a 4-bit IGNORED field in the Block and Translation Table descriptors, bits[58:55], for software use. In considering migration from using the Short-descriptor format to the Long-descriptor format, this field is an extension of the Short-descriptor field described in The OS managed translation table bits on page G5-9209.
Note

The definition of IGNORED means there is no need to invalidate the TLB if these bits are changed.

G5.7.5 EL2 control of Non-secure memory region attributes

Software executing at EL2 controls two sets of translation tables, both of which use the Long-descriptor translation table format. These are:

- The translation tables that control Non-secure EL2 stage 1 translations. These map VAs to PAs, and when EL2 is using AArch32 they are indicated and controlled by the HTTBR and HTCR. These translations have exactly the same memory region attribute controls as any other stage 1 translations, as described in VMSAv8-32 Long-descriptor format memory region attributes on page G5-9210.

- The translation tables that control Non-secure PL1&0 stage 2 translations. These map the IPAs from the stage 1 translation onto PAs, and are indicated and when EL2 is using AArch32 they are controlled by the VTTBR and VTCR. The descriptors in the virtualization translation tables define level 2 memory region attributes, that are combined with the attributes defined in the stage 1 translation. This section describes this combining of attributes.

VMSAv8-32 Long-descriptor Translation Table format descriptors on page G5-9173 describes the format of the entries in these tables.

Note

In a virtualization implementation, a hypervisor might usefully:

- Reduce the permitted Cacheability of a region.
- Increase the required Shareability of a region.

The combining of attributes from stage 1 and stage 2 translations supports both of these options.

In the stage 2 Translation Table descriptors for memory regions and pages, the MemAttr[3:0] and SH[1:0] fields describe the stage 2 memory region attributes:

- The definition of the stage 2 SH[1:0] field is identical to the same field for a stage 1 translation, see Shareability, Long-descriptor format on page G5-9210.

- MemAttr[3:2] give a top-level definition of the memory type, and of the cacheability of a Normal memory region, as Table G5-17 on page G5-9212 shows:

Table G5-17 Long-descriptor MemAttr[3:2] encoding, stage 2 translation

<table>
<thead>
<tr>
<th>MemAttr[3:2]</th>
<th>Memory type</th>
<th>Cacheability</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Device, of type determined by MemAttr[1:0]</td>
<td>Not applicable</td>
</tr>
<tr>
<td>01</td>
<td>Normal, Inner cacheability determined by MemAttr[1:0]</td>
<td>Outer Non-cacheable</td>
</tr>
<tr>
<td>10</td>
<td>Outer Write-Through Cacheable</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Outer Write-Back Cacheable</td>
<td></td>
</tr>
</tbody>
</table>
The encoding of MemAttr[1:0] depends on the Memory type indicated by MemAttr[3:2]:

— When MemAttr[3:2] == 0b00, indicating a type of Device memory, Table G5-18 shows the encoding of MemAttr[1:0]:

<table>
<thead>
<tr>
<th>MemAttr[1:0]</th>
<th>Meaning when MemAttr[3:2] == 0b00</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Region is Device-nGnRnE memory</td>
</tr>
<tr>
<td>01</td>
<td>Region is Device-nGnRE memory</td>
</tr>
<tr>
<td>10</td>
<td>Region is Device-nGRE memory</td>
</tr>
<tr>
<td>11</td>
<td>Region is Device-GRE memory</td>
</tr>
</tbody>
</table>

— When MemAttr[3:2] != 0b00, indicating Normal memory, Table G5-19 on page G5-9213 shows the encoding of MemAttr[1:0]:

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Reserved, CONSTRAINED UNPREDICTABLE, See Reserved values in System and memory-mapped registers and translation table entries on page K1-11583 for the permitted behavior.</td>
</tr>
<tr>
<td>01</td>
<td>Inner Non-cacheable</td>
</tr>
<tr>
<td>10</td>
<td>Inner Write-Through Cacheable</td>
</tr>
<tr>
<td>11</td>
<td>Inner Write-Back Cacheable</td>
</tr>
</tbody>
</table>

--- Note ---

The stage 2 translation does not assign any allocation hints.

The following sections describe how the memory type attributes assigned at stage 2 of the translation are combined with those assigned at stage 1:

• Combining the memory type attribute on page G5-9214.
• Combining the Cacheability attribute on page G5-9214.
• Combining the Shareability attribute on page G5-9215.

--- Note ---

• The following stage 2 translation table attribute settings leave the stage 1 settings unchanged:
  — MemAttr[1:0] == 0b11, Inner Write-Back Cacheable.

• In addition to the attribute combinations described in this section, Access permissions for instruction execution on page G5-9196 describes how the stage 1 and stage 2 execute-never permission fields are combined, so that a region is execute-never if it is defined as execute-never in at least one stage of translation.
Combining the memory type attribute

Table G5-20 shows how the stage 1 and stage 2 memory type assignments are combined:

<table>
<thead>
<tr>
<th>Assignment in stage 1</th>
<th>Assignment in stage 2</th>
<th>Resultant type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device-nGnRnE</td>
<td>Any</td>
<td>Device-nGnRnE</td>
</tr>
<tr>
<td>Device-nGnRE</td>
<td>Device-nGnRnE</td>
<td>Device-nGnRnE</td>
</tr>
<tr>
<td></td>
<td>Not Device-nGnRnE</td>
<td>Device-nGnRE</td>
</tr>
<tr>
<td>Device-nGRE</td>
<td>Device-nGnRnE</td>
<td>Device-nGnRnE</td>
</tr>
<tr>
<td></td>
<td>Device-nGnRE</td>
<td>Device-nGnRE</td>
</tr>
<tr>
<td></td>
<td>Not (Device-nGnRnE or Device-nGnRE)</td>
<td>Device-nGRE</td>
</tr>
<tr>
<td>Device-GRE</td>
<td>Device-nGnRnE</td>
<td>Device-nGnRnE</td>
</tr>
<tr>
<td></td>
<td>Device-nGnRE</td>
<td>Device-nGnRE</td>
</tr>
<tr>
<td></td>
<td>Device-nGRE</td>
<td>Device-nGRE</td>
</tr>
<tr>
<td></td>
<td>Device-GRE or Normal</td>
<td>Device-GRE</td>
</tr>
<tr>
<td>Normal</td>
<td>Any type of Device</td>
<td>Device type assigned at stage 2</td>
</tr>
<tr>
<td></td>
<td>Normal</td>
<td>Normal</td>
</tr>
</tbody>
</table>

See Combining the Shareability attribute on page G5-9215 for information about the Shareability of:

- A region for which the resultant type is any Device type.
- A region with a resultant type of Normal for which the resultant cacheability, described in Combining the Cacheability attribute, is Inner Non-cacheable, Outer Non-cacheable.

The combining of the memory type attribute means a translation table walk for a stage 1 translation can be made to a type of Device memory. If this occurs, then:

- If the value of HCR.PTW is 0, then the translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be done speculatively.
- If the value of HCR.PTW is 1, then the memory access generates a stage 2 Permission fault.

Combining the Cacheability attribute

For a Normal memory region, Table G5-21 on page G5-9214 shows how the stage 1 and stage 2 Cacheability assignments are combined. This combination applies, independently, for the Inner Cacheability and Outer Cacheability attributes:

<table>
<thead>
<tr>
<th>Assignment in stage 1</th>
<th>Assignment in stage 2</th>
<th>Resultant cacheability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-cacheable</td>
<td>Any</td>
<td>Non-cacheable</td>
</tr>
<tr>
<td>Any</td>
<td>Non-cacheable</td>
<td>Non-cacheable</td>
</tr>
</tbody>
</table>
Combining the Shareability attribute

In the following cases, a memory region is treated as Outer Shareable, regardless of any shareability assignments at either stage of translation:

- The resultant memory type attribute, described in Combining the memory type attribute on page G5-9214, is any type of Device memory.
- The resultant memory type attribute is Normal memory, and the resultant Cacheability, described in Combining the Cacheability attribute on page G5-9214, is Inner Non-cacheable Outer Non-cacheable.

For a memory region with a resultant memory type attribute of Normal that is not Inner Non-cacheable Outer Non-cacheable, Table G5-22 on page G5-9215 shows how the stage 1 and stage 2 shareability assignments are combined:

<table>
<thead>
<tr>
<th>Assignment in stage 1</th>
<th>Assignment in stage 2</th>
<th>Resultant Shareability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outer Shareable</td>
<td>Any</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td>Inner Shareable</td>
<td>Outer Shareable</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td>Inner Shareable</td>
<td>Inner Shareable</td>
<td>Inner Shareable</td>
</tr>
<tr>
<td>Inner Shareable</td>
<td>Non-shareable</td>
<td>Inner Shareable</td>
</tr>
<tr>
<td>Non-shareable</td>
<td>Outer Shareable</td>
<td>Outer Shareable</td>
</tr>
<tr>
<td>Non-shareable</td>
<td>Inner Shareable</td>
<td>Inner Shareable</td>
</tr>
<tr>
<td>Non-shareable</td>
<td>Non-shareable</td>
<td>Non-shareable</td>
</tr>
</tbody>
</table>

Note: Only Normal memory has a Cacheability attribute.
G5.8 Translation Lookaside Buffers (TLBs)

Translation Lookaside Buffers (TLBs) are an implementation technique that caches translations or translation table entries. TLBs avoid the requirement to perform a translation table walk in memory for every memory access. The Arm architecture does not specify the exact form of the TLB structures for any design. In a similar way to the requirements for caches, the architecture only defines certain principles for TLBs:

- The architecture has a concept of an entry locked down in the TLB. The method by which lockdown is achieved is IMPLEMENTATION DEFINED, and an implementation might not support lockdown.

- The architecture does not guarantee that an unlocked TLB entry remains in the TLB.

- The architecture guarantees that a locked TLB entry remains in the TLB. However, a locked TLB entry might be updated by subsequent updates to the translation tables. Therefore, when a change is made to the translation tables, the architecture does not guarantee that a locked TLB entry remains incoherent with an entry in the translation table.

- The architecture guarantees that a translation table entry that generates a Translation fault, an Address size fault, or an Access flag fault is not held in the TLB. However a translation table entry that generates a Domain fault or a Permission fault might be held in the TLB.

- When address translation is enabled, any translation table entry that does not generate a Translation fault, an Address size fault, or an Access flag fault and is not from a translation regime for an Exception level that is lower than the current Exception level can be allocated to a TLB at any time. The only translation table entries guaranteed not to be held in the TLB are those that generate a Translation fault, an Address size fault, or an Access flag fault.

Note: A TLB can hold translation table entries that do not generate a Translation fault but point to subsequent tables in the translation table walk. This can be referred to as intermediate caching of TLB entries.

- Software can rely on the fact that between disabling and re-enabling a stage of address translation, entries in the TLB relating to that stage of translation have not been corrupted to give incorrect translations.

The following sections give more information about TLB implementation:

- Global and process-specific translation table entries.
- TLB matching on page G5-9217.
- TLB behavior at reset on page G5-9217.
- TLB lockdown on page G5-9218.
- TLB conflict aborts on page G5-9218.

See also TLB maintenance requirements on page G5-9220.

Note: In addition to the functions described in this section, the TLB might cache information from control registers that are described as being "permitted to be cached in a TLB", even when any or all of the stages of translation are disabled. This caching of information gives rise to the maintenance requirements described in General TLB maintenance requirements on page G5-9220.

G5.8.1 Global and process-specific translation table entries

For VMSAv8-32, system software can divide a virtual memory map used by memory accesses at PL1 and EL0 into global and non-global regions, indicated by the nG bit in the Translation Table descriptors:

nG == 0 The translation is global, meaning the region is available for all processes.
nG == 1  The translation is non-global, or process-specific, meaning it relates to the current ASID, as defined by:
   • TTBR0.ASID or TTBR1.ASID, if using the Long-descriptor translation table format. In this case, TTBCR.A1 selects which ASID is current.
   • CONTEXTIDR.ASID, if using the Short-descriptor translation table format.

Each non-global region has an associated ASID. These identifiers mean different translation table mappings can co-exist in a caching structure such as a TLB. This means that software can create a new mapping of a non-global memory region without removing previous mappings.

For a symmetric multiprocessor cluster where a single operating system is running on the set of PEs, the architecture requires all ASID values to be assigned uniquely within any single Inner Shareable domain. In other words, each ASID value must have the same meaning to all PEs in the system.

In AArch32 state, the translation regime used for accesses made at EL2 never supports ASIDs, and all pages are treated as global.

When a PE is using the Long-descriptor translation table format, and is in Secure state, a translation must be treated as non-global, regardless of the value of the nG bit, if NSTable is set to 1 at any level of the translation table walk.

For more information, see Control of Secure or Non-secure memory access, VMSA v8-32 Long-descriptor format on page G5-9180.

G5.8.2 TLB matching

A TLB is a hardware caching structure for translation table information. Like other hardware caching structures, it is mostly invisible to software. However, there are some situations where it can become visible. These are associated with coherency problems caused by an update to the translation table that has not been reflected in the TLB. Use of the TLB maintenance instructions described in TLB maintenance requirements on page G5-9220 can prevent any TLB incoherency becoming a problem.

A particular case where the presence of the TLB can become visible is if the translation table entries that are in use under a particular ASID and VMID are changed without suitable invalidation of the TLB. This can occur only if the architecturally-required break-before-make sequence described in Using break-before-make when updating translation table entries on page G5-9221 is not used. If the break-before make sequence is not used, the TLB can hold two mappings for the same address, and this:
   • Might generate an exception that is reported using the TLB Conflict fault code, see TLB conflict aborts on page G5-9218.
   • Might lead to CONSTRAINED UNPREDICTABLE behavior. In this case, behavior will be consistent with one of the mappings held in the TLB, or with some amalgamation of the values held in the TLB, but cannot give access to regions of memory with permissions or attributes that could not be assigned by valid translation table entries in the translation regime being used for the access. See CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values on page K1-11567.

G5.8.3 TLB behavior at reset

The Arm architecture does not require a reset to invalidate the TLBs, and recognizes that an implementation might require caches, including TLBs, to maintain context over a system reset. Possible reasons for doing so include power management and debug requirements.

Therefore:
   • All TLBs reset to an IMPLEMENTATION DEFINED state that might be UNKNOWN.
   • All TLBs are disabled from reset. All stages of address translation that are used from the PE state entered on coming out of reset are disabled from reset, and the contents of the TLBs have no effect on address translation. For more information, see Enabling stages of address translation on page G5-9156.
• An implementation can require the use of a specific TLB invalidation routine, to invalidate the TLB arrays before they are enabled after a reset. The exact form of this routine is IMPLEMENTATION DEFINED, but if an invalidation routine is required it must be documented clearly as part of the documentation of the device.

Arm recommends that if an invalidation routine is required for this purpose, and the PE resets into AArch32 state, the routine is based on the AArch32 TLB maintenance instructions described in *The scope of TLB maintenance instructions on page G5-9230.*

Similar rules apply:
• To cache behavior, see *Behavior of caches at reset on page G4-9119.*
• To branch predictor behavior, see *Behavior of the branch predictors at reset on page G4-9127.*

**G5.8.4 TLB lockdown**

The Arm architecture recognizes that any TLB lockdown scheme is heavily dependent on the microarchitecture, making it inappropriate to define a common mechanism across all implementations. This means that:

• The architecture does not require TLB lockdown support.
• If TLB lockdown support is implemented, the lockdown mechanism is IMPLEMENTATION DEFINED. However, key properties of the interaction of lockdown with the architecture must be documented as part of the implementation documentation.

This means that:
• The TLB Type Register, TLBTR, does not define the lockdown scheme in use.
• In AArch32 state, a region of the `{coproc=0b1111, CRn=c10}` encodings is reserved for IMPLEMENTATION DEFINED TLB functions, such as TLB lockdown functions. The reserved encodings are those with:
  — `<Cr>` = `{c0, c1, c4, c8}`.
  — All values of `<opc2>` and `<opc1>`.

An implementation might use some of the `{coproc=0b1111, CRn=c10}` encodings that are reserved for IMPLEMENTATION DEFINED TLB functions to implement additional TLB control functions. These functions might include:
• Unlock all locked TLB entries.
• Preload into a specific level of TLB. This is beyond the scope of the PLI and PLD hint instructions.

The inclusion of EL2 in an implementation does not affect the TLB lockdown requirements. However, in an implementation that includes EL2, exceptions generated as a result of TLB lockdown when executing in a Non-secure PL1 mode or in Non-secure User mode can be routed to either:
• Non-secure Abort mode, using the Non-secure Data Abort exception vector.
• Hyp mode, using the Hyp Trap exception vector.

For more information, see *Traps to Hyp mode of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations on page G1-9016.*

**G5.8.5 TLB conflict aborts**

If an address matches multiple entries in the TLB, it is IMPLEMENTATION DEFINED whether a TLB conflict abort is generated.

An implementation can generate TLB conflict aborts on either or both instruction fetches and data accesses. A TLB conflict abort is classified as an MMU fault, see *Types of MMU faults on page G5-9239.* This means:
• A TLB conflict abort on an instruction fetch is reported as a Prefetch Abort exception,
• A TLB conflict abort on a data access is reported as a Data Abort exception,

Fault status codes for TLB conflict aborts are defined for both the Short-descriptor and Long-descriptor translation table formats, see:
• *PL1 fault reporting with the Short-descriptor translation table format on page G5-9256*
• *PL1 fault reporting with the Long-descriptor translation table format on page G5-9258.*
On a TLB conflict abort, the fault address register returns the address that generated the fault. That is, it returns the address that was being looked up in the TLB.

It is IMPLEMENTATION DEFINED whether a TLB conflict abort is a stage 1 abort or a stage 2 abort.

--- Note ---

- An address can hit multiple entries in the TLB if the TLB has been invalidated inappropriately, for example if TLB invalidation required by this manual has not been performed.
- A stage 2 abort cannot be generated if the Non-secure PL1&0 stage 2 address translation is disabled.

The priority of the TLB conflict abort is IMPLEMENTATION DEFINED, because it depends on the form of any TLB that can generate the abort. However, the TLB conflict abort must have higher priority than any abort that depends on a value held in the TLB.

If an address matches multiple entries in the TLB and no TLB conflict abort not generated, the resulting behavior is CONSTRAINED UNPREDICTABLE, see CONSTRANGED UNPREDICTABLE behaviors due to caching of System register control or data values on page K1-11567. The CONSTRAINED UNPREDICTABLE behavior must not permit access to regions of memory with permissions or attributes that mean they cannot be accessed in the current Security state at the current Privilege level.
G5.9 TLB maintenance requirements

Translation Lookaside Buffers (TLBs) on page G5-9216 describes the Arm architectural provision for TLBs. Although the Arm architecture does not specify the form of any TLB structures, it does define the mechanisms by which TLBs can be maintained. The following sections describe the VMSaV8-32 TLB maintenance instructions:

- General TLB maintenance requirements.
- Maintenance requirements on changing System register values on page G5-9226.
- Atomicity of register changes on changing virtual machine on page G5-9227.
- Synchronization of changes of ASID and TTBR on page G5-9227.
- The scope of TLB maintenance instructions on page G5-9230.

G5.9.1 General TLB maintenance requirements

TLB maintenance instructions provide a mechanism to invalidate entries from a TLB. As Translation Lookaside Buffers (TLBs) on page G5-9216 describes, when address translation is enabled translation table entries can be allocated to a TLB at any time. This means that software must perform TLB maintenance between updating translation table entries that apply in a particular context and accessing memory locations whose translation is determined by those entries in that context.

Note

This requirement applies to any translation table entry at any level of the translation tables, including an entry that points to further levels of the tables, provided that the entry in that level of the tables does not cause a Translation fault, an Address size fault, or an Access flag fault.

Note

The effects of certain System register control fields are permitted to be cached in a TLB. For more information, see TLB maintenance on page D8-5197.

In addition to any TLB maintenance requirement, when changing the cacheability attributes of an area of memory, software must ensure that any cached copies of affected locations are removed from the caches. For more information, see Cache maintenance requirement created by changing translation table attributes on page G5-9237.

Because a TLB never holds any translation table entry that generates a Translation fault, an Address size fault, or an Access flag fault, a change from a translation table entry that causes a Translation, Address size, or Access flag fault to one that does not fault, does not require any TLB or branch predictor invalidation. However, a Context synchronization event is required to ensure that instruction fetches are affected by a completed change to translation table entries that, before the change, generated a Translation, Address size, or Access flag fault.

Special considerations apply to translation table updates that change the memory type, cacheability, or output address of an entry, see Using break-before-make when updating translation table entries on page G5-9221.

In addition, software must perform TLB maintenance after updating the System registers if the update means that the TLB might hold information that applies to a current translation context, but is no longer valid for that context. Maintenance requirements on changing System register values on page G5-9226 gives more information about this maintenance requirement.

Each of the translation regimes defined in Figure G5-1 on page G5-9148 is a different context, and:

- For the Non-secure PL1&0 regime, a change in the VMID or ASID value changes the context.
- For the Secure PL1&0 regime, a change in the ASID value changes the context.

For operation in Non-secure PL1 or EL0 modes, a change of HCR.VM, unless made at the same time as a change of VMID, requires the invalidation of all TLB entries for the Non-secure PL1&0 translation regime that apply to the current VMID. Otherwise, there is no guarantee that the effect of the change of HCR.VM is visible to software executing in the Non-secure PL1 and EL0 modes.
Any TLB maintenance instruction can affect any other TLB entries that are not locked down.

AArch32 state defines \{coproc==0b1111, Crn==c8\} System instructions for TLB maintenance instructions, and supports the following operations:

- Invalidate all unlocked entries in the TLB.
- Invalidate a single TLB entry, by VA, or VA and ASID for a non-global entry.
- Invalidate all TLB entries that match a specified ASID.
- Invalidate all TLB entries that match a specified VA, regardless of the ASID.
- Operations that apply across multiprocessors in the same Inner Shareable domain.

--- Note ---

An address-based TLB maintenance instruction that applies to the Inner Shareable domain does so regardless of the Shareability attributes of the address supplied as an argument to the instruction.

A TLB maintenance instruction that specifies a VA that would generate any MMU fault, including a VA that is not in the range of VAs that can be translated, does not generate an abort.

EL2 provides additional TLB maintenance instructions for use in AArch32 state at EL2, and has some implications for the effect of the other TLB maintenance instructions, see The scope of TLB maintenance instructions on page G5-9230.

In an implementation that includes EL3, the TLB maintenance instructions take account of the current Security state, as part of the address translation required for the TLB maintenance instruction.

Some TLB maintenance instructions are defined as operating only on instruction TLBs, or only on data TLBs. AArch32 state includes these instructions for backwards compatibility. However, more recent TLB maintenance instructions do not support this distinction. From the introduction of Armv7, Arm deprecates any use of Instruction TLB maintenance instructions, or of Data TLB maintenance instructions, and developers must not rely on this distinction being maintained in future revisions of the Arm architecture.

The Arm architecture does not dictate the form in which the TLB stores translation table entries. However, for TLB invalidate instructions, the minimum size of the table entry that is invalidated from the TLB must be at least the size that appears in the translation table entry.

The scope of TLB maintenance instructions on page G5-9230 describes the TLB maintenance instructions. The following subsections give more information about the general requirements for TLB maintenance:

- Using break-before-make when updating translation table entries.
- The interaction of TLB lockdown with TLB maintenance instructions on page G5-9222.
- Ordering and completion of TLB maintenance instructions on page G5-9223.
- Use of ASIDs and VMIDs to reduce TLB maintenance requirements on page G5-9225.

Using break-before-make when updating translation table entries

To avoid possibly creating multiple TLB entries for the same address, and to avoid the effects of TLB caching possibly breaking coherency, single-copy atomicity properties, ordering guarantees or uniprocessor semantics, or possibly failing to clear the Exclusives monitors, the architecture requires the use of a break-before-make sequence when changing translation table entries whenever multiple threads of execution can use the same translation tables and the change to the translation table entries involves any of:

- A change of the memory type, including shareability.
- A change of the cacheability attributes.
- A change of the output address (OA), if the OA of at least one of the old translation table entries and the new translation table entry is writable.
- A change to the size of block used by the translation system. This applies both:
  — When changing from a smaller size to a larger size, for example by replacing a table mapping with a block mapping in a stage 2 translation table.
— When changing from a larger size to a smaller size, for example by replacing a block mapping with a table mapping in a stage 2 translation table.

• A change of the output address (OA), if the contents of memory at the new OA do not match the contents of memory at the previous OA.

• Creating a global entry when there might be non-global entries in a TLB that overlap with that global entry.

Note

Changes to the output address (OA) include changing between Secure and Non-secure output addresses.

A break-before-make sequence on changing from an old translation table entry to a new translation table entry requires the following steps:

1. Replace the old translation table entry with an invalid entry, and execute a DSB instruction.
2. Invalidate the translation table entry with a broadcast TLB invalidation instruction, and execute a DSB instruction to ensure the completion of that invalidation.
3. Write the new translation table entry, and execute a DSB instruction to ensure that the new entry is visible.

This sequence ensures that at no time are both the old and new entries simultaneously visible to different threads of execution, and therefore the problems described at the start of this subsection cannot arise.

The interaction of TLB lockdown with TLB maintenance instructions

The precise interaction of TLB lockdown with the TLB maintenance instructions is IMPLEMENTATION DEFINED. However, the architecturally-defined TLB maintenance instructions must comply with these rules:

• The effect on locked entry of a TLB invalidate all unlocked entries instruction or a TLB invalidate by VA all ASID instruction that would invalidate that entry if the entry was not locked must be one of the following, and it is IMPLEMENTATION DEFINED which behavior applies:
  — The instructions have no effect on entries that are locked down.
  — The instructions generate an IMPLEMENTATION DEFINED Data Abort exception if an entry is locked down, or might be locked down. For an invalidate instruction performed in AArch32 state, the \{coproc==0b1111, CRn==c5\} fault status register definitions include a Fault status code for cache and TLB lockdown faults, see Table G5-26 on page G5-9256 for the codes used with the Short-descriptor translation table formats, or Table G5-27 on page G5-9258 for the codes used with the Long-descriptor translation table formats.

In an implementation that includes EL2, if EL2 is using AArch32 and the value of HCR.TIDCP is 1, any such exceptions taken from a Non-secure PL1 mode are routed to Hyp mode, see Traps to Hyp mode of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations on page G1-9016.

This permits a usage model for TLB invalidate routines, where the routine invalidates a large range of addresses, without considering whether any entries are locked in the TLB.

• The effect on a locked TLB entry of a TLB invalidate by VA instruction or a TLB invalidate by ASID match instruction that would invalidate that entry if the entry was not locked must be one of the following, and it is IMPLEMENTATION DEFINED which behavior applies:
  — A locked entry is invalidated in the TLB.
  — The instruction has no effect on a locked entry in the TLB. In the case of the Invalidate single entry by VA, this means the PE treats the instruction as a NOP.
  — The instruction generates an IMPLEMENTATION DEFINED Data Abort exception if it operates on an entry that is locked down, or might be locked down. For an invalidate instruction performed in AArch32 state, the \{coproc==0b1111, CRn==c5\} fault status register definitions include a Fault status code for cache and TLB lockdown faults, see Table G5-26 on page G5-9256 and Table G5-27 on page G5-9258.
Any implementation that uses an abort mechanism for entries that can be locked down but are not actually locked down must:

- Document the IMPLEMENTATION DEFINED instruction sequences that perform the required invalidation on entries that are not locked down.

- Implement one of the other specified alternatives for the locked entries.

Arm recommends that, when possible, such IMPLEMENTATION DEFINED instruction sequences use the architecturally-defined maintenance instructions. This minimizes the number of customized maintenance operations required.

In addition, an implementation that uses an abort mechanism for handling TLB maintenance instructions on entries that can be locked down but are not actually locked down must also must provide a mechanism that ensures that no TLB entries are locked.

Similar rules apply to cache lockdown, see The interaction of cache lockdown with cache maintenance instructions on page G4-9136.

The architecture does not guarantee that any unlocked entry in the TLB remains in the TLB. This means that, as a side-effect of a TLB maintenance instruction, any unlocked entry in the TLB might be invalidated.

### Ordering and completion of TLB maintenance instructions

The following rules describe the relations between the memory order model and the TLB maintenance instructions:

- A TLB maintenance instruction executed by a PE, PEe, causes a TLB maintenance operation to be generated on each PE within the shareability domain of PEe that is specified by the instruction.
  - At EL2 or EL3, or at EL1 when the Effective value of HCRX_EL2.FnXS is 0, the associated TLB maintenance operations do not have the nXS qualifier.
  - At EL1, when the Effective value of HCRX_EL2.FnXS is 1, the behavior of the associated TLB maintenance operations is the same as described for the AArch64 TLB maintenance instructions with the nXS qualifier. See Ordering and completion of TLB maintenance instructions on page D8-5213.

- When FEAT_XS is not implemented, all TLB maintenance instructions do not have the nXS qualifier and the Effective value of HCRX_EL2 is 0.

- A TLB maintenance operation generated by a TLB maintenance instruction is finished for a PE when:
  - All memory accesses generated by that PE using in-scope old translation information are complete.
  - All memory accesses RWx generated by that PE are complete.

RWx is the set of all memory accesses generated by instructions for that PE that appear in program order before an instruction I1 executed by that PE where all of the following apply:

- I1 uses the in-scope old translation information.
- The use of the in-scope old translation information generates a synchronous Data Abort.
- If I1 did not generate an abort from use of the in-scope old translation information, I1 would generate a memory access that RWx would be locally-ordered-before.

In-scope old translation information is any translation information, for addresses that are in the scope of the TLB maintenance instruction, that is not consistent with either:

- The architectural translation information held in the translation tables at the time that the TLB maintenance instruction is executed by PEe.
- Any architecture translation information that is Coherence-after the information held in the translation tables at the time that the TLB maintenance instruction is executed by PEe.
Old translation information of this type might be held in TLBs or other non-coherent caching structures.

A TLB maintenance instruction is complete when the TLB maintenance operations specified by the TLB maintenance instruction are finished for all PEs.

After the TLB maintenance instruction is complete, no new memory accesses using the in-scope old translation information will be architecturally performed by any observer that is affected by the TLB maintenance instruction.

Speculative memory accesses can be performed using those entries if it is impossible for software running on any observer to tell that those memory accesses have been performed.

A TLB maintenance instruction is only guaranteed to be complete after the execution of a DSB instruction.

An ISB instruction, or a return from an exception, causes the effect of all completed TLB maintenance instructions that appear in program order before the ISB or return from exception to be visible to all subsequent instructions, including the instruction fetches for those instructions.

An exception causes all completed TLB maintenance instructions, that appear in the instruction stream before the point where the exception is taken, to be visible to all subsequent instructions, including the instruction fetches for those instructions.

All TLB maintenance instructions are executed in program order relative to each other.

The execution of a Data or Unified TLB maintenance instruction is only guaranteed to be visible to a subsequent explicit memory read or write effect instruction after both:
- The execution of a DSB instruction to ensure the completion of the TLB maintenance instruction.
- Execution of a subsequent Context synchronization event.

The execution of an Instruction or Unified TLB maintenance instruction is only guaranteed to be visible to a subsequent instruction fetch after both:
- The execution of a DSB instruction to ensure the completion of the TLB maintenance instruction.
- Execution of a subsequent Context synchronization event.

In all cases in this section where a DMB or DSB is referred to, it refers to a DMB or DSB whose required access type is both loads and stores. A DSB NSH is sufficient to ensure completion of TLB maintenance instructions that apply to a single PE. A DSB ISH is sufficient to ensure completion of TLB maintenance instructions that apply to PEs in the same Inner Shareable domain.

The following rules apply when writing translation table entries. They ensure that the updated entries are visible to subsequent accesses and cache maintenance instructions.

For TLB maintenance, the translation table walk is treated as a separate observer. This means:
- A write to the translation tables is only guaranteed to be seen by a translation table walk caused by an explicit memory read or write effect after the execution of both a DSB and an ISB.
  However, the architecture guarantees that any writes to the translation tables are not seen by any explicit memory effect that occurs in program order before the write to the translation tables.
- A write to the translation tables is only guaranteed to be seen by a translation table walk caused by the instruction fetch of an instruction that follows the write to the translation tables after both a DSB and an ISB.

Therefore, in a uniprocessor system, an example instruction sequence for writing a translation table entry, covering changes to the instruction or data mappings is:

```
STR rx, [Translation table entry] ; write new entry to the translation table
DSB ; ensures visibility of the new entry
Invalidate TLB entry by VA (and ASID if non-global) [page address]
Invalidate BTC
DSB ; ensure completion of the Invalidate TLB instruction
```
Use of ASIDs and VMIDs to reduce TLB maintenance requirements

To reduce the need for TLB maintenance on context switches, the lookups from some translation regimes can be associated with an ASID, or with an ASID and a VMID.

Note

The use of ASIDs and VMIDs in VMSAv8-32 is generally similar to their use in VMSAv8-64, see Use of ASIDs and VMIDs to reduce TLB maintenance requirements on page D8-5194.

For more information about the use of ASIDs in VMSAv8-32 see Global and process-specific translation table entries on page G5-9216.

Common not private translations in VMSAv8-32

In an implementation that includes FEAT_TTCNP, multiple PEs in the same Inner Shareable domain can use the same translation table entries for a given stage of address translation in a particular translation regime. This sharing is enabled by the TTBR.CnP field for the stage of address translation.

When the value of a TTBR.CnP field is 1, translation table entries pointed to by that TTBR are shared with all other PEs in the Inner Shareable domain for which the following conditions are met:

- The corresponding TTBR.CnP field has the value 1.
- That TTBR is using the Long-descriptor translation table format.
- If an ASID applies to the stage of translation corresponding to that TTBR then the current ASID value must be the same for all of the PEs that are sharing entries for any translation table entry that is not global or not leaf level.
- If a VMID applies to the stage of translation corresponding to that TTBR then the current VMID value must be the same for all of the PEs that are sharing entries.

Note

In an implementation that includes EL3, the Secure instances of TTBR0 and TTBR1 relate to the Secure PL1&0 translation regime, and the Non-secure instances of TTBR0 and TTBR1 relate to the Non-secure PL1&0 translation regime.

For a translation regime with both stage 1 and stage 2 translations, where a TLB combines information from stage 1 and stage 2 translation table entries into a single entry, this entry can be shared between different PEs only if the value of the TTBR.CnP bit is 1 for both stage 1 and stage 2 of the translation table walk.

The TTBR.CnP bit can be cached in a TLB.

For a given TTBR, if the value of TTBR.CnP is 1 on multiple PEs in the same Inner Shareable domain, and those PEs meet the other conditions for sharing translation table entries as defined in this section, but those TTBRs do not point to the same translation table entries, then the system is misconfigured, and performing an address translation using that TTBR:

- Might generate multiple hits in the TLB, and as a result generate an exception that is reported using the TLB conflict fault code, see TLB conflict aborts on page G5-9218.
- Otherwise, has a CONSTRAINED UNPREDICTABLE result, as described in CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values on page K1-11567.
### G5.9.2 Maintenance requirements on changing System register values

The TLB contents can be influenced by control bits in a number of System registers. This means the TLB entries associated with a translation regime affected by these control bits must be invalidated after any changes to these bits, unless the changes are accompanied by a change to the VMID or ASID, if appropriate depending on the translation regime, that defines the context to which the bits apply. The general form of the required invalidation sequence is as follows:

```plaintext
; Change control bits in System registers
ISB ; Synchronize changes to the control bits
; Perform TLB invalidation of all entries that might be affected by the changed control bits
```

The System register changes that this applies to are:

- Any change to the NMRR, PRRR, MAIR0, MAIR1, HMAIR0 or HMAIR1 registers.
- Any change to the SCTLR.AFE bit, see Changing the Access flag enable.
- Any change to any of the SCTLR.{TRE, WXN, UWXN} bits.
- Any change to the Translation table base 0 address in TTBR0.
- Any change to the Translation table base 1 address in TTBR1.
- Any change to HTTBR.BADDR.
- Any change to VTTBR.BADDR.
- Changing TTBCR.EAE, see Changing the current Translation table format on page G5-9227.
- In an implementation that includes EL3, any change to the SCR.SIF bit.
- In an implementation that includes EL2:
  - Any change to the HCR.VM bit.
  - Any change to HCR.PTW bit, see Changing HCR.PTW.
- When using the Short-descriptor translation table format:
  - Any change to the RGN, IRGN, S, or NOS fields in TTBR0 or TTBR1.
  - Any change to the N, EAE, PD0 or PD1 fields in TTBCR
- When using the Long-descriptor translation table format:
  - Any change to the EAE, TnSZ, ORGNn, IRGNn, SHn, or EPDn fields in the TTBCR, where \( n \) is 0 or 1.
  - Any change to the TTBCR2.
  - Any change to the T0SZ, ORGN0, IRGN0, or SH0 fields in the HTCR.
  - Any change to the T0SZ, ORGN0, IRGN0, or SH0 fields in the VTCR.

#### Changing the Access flag enable

In a PE that is using the Short-descriptor translation table format, it is CONstrained UNPredictable whether the TLB caches the effect of the SCTLR.AFE bit on translation tables. This means that, after changing the SCTLR.AFE bit software must invalidate the TLB before it relies on the effect of the new value of the SCTLR.AFE bit, otherwise behavior is CONstrained UNPredictable, see CONstrained UNPredictable behaviors due to caching of System register control or data values on page K1-11567.

---

**Note**

There is no enable bit for use of the Access flag when using the Long-descriptor translation table format.

---

#### Changing HCR.PTW

When EL2 is using AArch32 and the value of the Protected table walk bit, HCR.PTW, is 1, a stage 1 translation table access in the Non-secure PL1&0 translation regime, to an address that is mapped to any type of Device memory by its stage 2 translation, generates a stage 2 Permission fault. A TLB associated with a particular VMID might hold entries that depend on the effect of HCR.PTW. Therefore, if the value of HCR.PTW is changed without a change to the VMID value, all TLB entries associated with the current VMID must be invalidated before executing software in a Non-secure PL1 or EL0 mode. If this is not done, behavior is CONstrained UNPredictable, see CONstrained UNPredictable behaviors due to caching of System register control or data values on page K1-11567.
Changing the current Translation table format

The effect of changing TTBCR.EAE when executing in the translation regime affected by TTBCR.EAE with any stage of address translation for that translation regime enabled is CONSTRAINED UNPREDICTABLE. This means that, when TTBCR.EAE is changed for a given context, the TLB must be invalidated before resuming execution in that context, otherwise the effect is CONSTRAINED UNPREDICTABLE, see CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values on page K1-11567.

G5.9.3 Atomicty of register changes on changing virtual machine

From the viewpoint of software executing in a Non-secure PL1 or EL0 mode, when there is a switch from one virtual machine to another, the registers that control or affect address translation must be changed atomically. This applies to the registers for:

- Non-secure PL1&0 stage 1 address translations. This means that all of the following registers must change atomically:
  - PRRR and NMRR, if using the Short-descriptor translation table format.
  - MAIR0 and MAIR1, if using the Long-descriptor translation table format.
  - TTBR0, TTBR1, TTBCR, TTBCR2, DACR, and CONTEXTIDR.
  - The SCTLR.
- Non-secure PL1&0 stage 2 address translations. When EL2 is using AArch32, this means that all of the following registers and register fields must change atomically:
  - VTTBR and VTCR.
  - HMAIR0 and HMAIR1.
  - The HSCTLR.

Note
Only some bits of SCTLR affect the stage 1 translation, and only some bits of HSCTLR affect the stage 2 translation. However, in each case, changing these bits requires a write to the register, and that write must be atomic with the other register updates.

These registers apply to execution in Non-secure PL1&0 modes. However, when updated as part of a switch of virtual machines they are updated by software executing in Hyp mode. This means the registers are out of context when they are updated, and no synchronization precautions are required.

Note
By contrast, a translation table change associated with a change of ASID, made by software executing at PL1, can require changes to registers that are in context. Synchronization of changes of ASID and TTBR describes appropriate precautions for such a change.

G5.9.4 Synchronization of changes of ASID and TTBR

A common virtual memory management requirement is to change the ASID and TTBR together to associate the new ASID with different translation tables, without any change to the current translation regime. When using the Short-descriptor translation table format, different registers hold the ASID and the translation table base address, meaning these two values cannot be updated atomically. Since a PE can perform a speculative memory access at any time, this lack of atomicity is a problem that software must address. Such a change is complicated by:

- The depth of speculative fetch being IMPLEMENTATION DEFINED.
- The use of branch prediction.
When using the Short-descriptor translation table format, the virtual memory management operations must ensure the synchronization of changes of the ContextID and the translation table registers. For example, some or all of the TLBs, branch predictors, and other caching of ASID and translation information might become corrupt with invalid translations. Synchronization is required to avoid either:

- The old ASID being associated with translation table walks from the new translation tables.
- The new ASID being associated with translation table walks from the old translation tables.

There are a number of possible solutions to this problem, and the most appropriate approach depends on the system. Example G5-3, Example G5-4, and Example G5-5 on page G5-9229 describe three possible approaches.

--- Note ---

Another instance of the synchronization problem occurs if a branch is encountered between changing the ASID and performing the synchronization. In this case the value in the branch predictor might be associated with the incorrect ASID. Software can address this possibility using any of these approaches, but instead software might be written in a way that avoids such branches.

--- Example G5-3 Using a reserved ASID to synchronize ASID and TTBR changes ---

In this approach, a particular ASID value is reserved for use by the operating system, and is used only for the synchronization of the ASID and TTBR. This example uses the value of 0 for this purpose, but any value could be used.

This approach can be used only when the size of the mapping for any given VA is the same in the old and new translation tables.

The maintenance software uses the following sequence, which must be executed from memory marked as global:

```
Change ASID to 0
ISB
Change TTBR
ISB
Change ASID to new value
```

This approach ensures that any non-global pages fetched at a time when it is uncertain whether the old or new translation tables are being accessed are associated with the unused ASID value of 0. Since the ASID value of 0 is not used for any normal operations these entries cannot cause corruption of execution.

--- Example G5-4 Using translation tables containing only global mappings when changing the ASID ---

A second approach involves switching the translation tables to a set of translation tables that only contain global mappings while switching the ASID.

The maintenance software uses the following sequence, which must be executed from memory marked as global:

```
Change TTBR to the global-only mappings
ISB
Change ASID to new value
ISB
Change TTBR to new value
```

This approach ensures that no non-global pages can be fetched at a time when it is uncertain whether the old or new ASID value will be used.
This approach works without the need for TLB invalidations in systems that have caching of intermediate levels of translation tables, as described in General TLB maintenance requirements on page G5-9220, provided that the translation tables containing only global mappings have only level 1 translation table entries of the following kinds:

- Entries that are global.
- Pointers to level 2 tables that hold only global entries, and that are the same level 2 tables that are used for accessing global entries by both:
  - The set of translation tables that were used under the old ASID value.
  - The set of translation tables that will be used with the new ASID value.
- Invalid level 1 entries.

In addition, all sets of translation tables in this example should have the same Shareability and Cacheability attributes, as held in the TTBR0.{ORGN, IRGN} or TTBR1.{ORGN, IRGN} fields.

If these rules are not followed, then the implementation might cache level 1 translation table entries that require explicit invalidation.

---

**Example G5-5 Disabling non-global mappings when changing the ASID**

In systems where only the translation tables indexed by TTBR0 hold non-global mappings, maintenance software can use the TTBCR.PD0 field to disable use of TTBR0 during the change of ASID. This means the system does not require a set of global-only mappings.

The maintenance software uses the following sequence, which must be executed from a memory region with a translation that is accessed using the base address in the TTBR1 register, and is marked as global:

```
Set TTBCR.PD0 = 1
ISB
Change ASID to new value
Change TTBR to new value
ISB
Set TTBCR.PD0 = 0
```

This approach ensures that no non-global pages can be fetched at a time when it is uncertain whether the old or new ASID value will be used.

When using the Long-descriptor translation table format, TTBCR.A1 holds the number, 0 or 1, of the TTBR that holds the current ASID. This means the current TTBR can also hold the current ASID, and the current translation table base address and ASID can be updated atomically when:

- TTBR0 is the only TTBR being used. TTBCR.A1 must be set to 0.
- TTBR0 points to the only translation tables that hold non-global entries, and TTBCR.A1 is set to 0.
- TTBR1 points to the only translation tables that hold non-global entries, and TTBCR.A1 is set to 1.

In these cases, software can update the current translation table base address and ASID atomically, by updating the appropriate TTBR, and does not require a specific routine to ensure synchronization of the change of ASID and base address.

However, in all other cases using the Long-descriptor format, the synchronization requirements are identical to those when using the Short-descriptor formats, and the examples in this section indicate how synchronization might be achieved.

---

**Note**

When using the Long-descriptor translation table format, CONTEXTIDR.ASID has no significance for address translation, and is only an extension of the Context ID value.
The AArch32 Virtual Memory System Architecture

G5.9 TLB maintenance requirements

G5.9.5 The scope of TLB maintenance instructions

TLB maintenance instructions provide a mechanism for invalidating entries from TLB caching structures, to ensure that changes to the translation tables are reflected correctly in the TLB caching structures. To support TLB maintenance in multiprocessor systems, there are maintenance operations that apply to the TLBs of all PEs in the same Inner Shareable domain.

The architecture permits the caching of any translation table entry that has been returned from memory without a fault and that does not, itself, cause a Translation Fault, an Address size fault, or an Access Flag fault. This means the TLB:

- Cannot hold an entry that, when used for a translation table lookup, causes a Translation fault, an Address size fault, or an Access Flag fault.

- Can hold an entry for a translation table lookup for a translation that causes a Translation Fault, an Address size fault, or an Access Flag fault at a subsequent level of translation table lookup. For example, it can hold an entry for the level 1 lookup of a translation that causes a Translation fault, an Address size fault, or an Access Flag fault at level 2 or level 3 of lookup.

This means that entries cached in the TLB can include:

- Translation table entries that point to a subsequent table to be used in the current stage of translation.

- In an implementation that includes EL2:
  - Stage 2 translation table entries that are used as part of a stage 1 translation table walk.
  - Stage 2 translation table entries for translating the output address of a stage 1 translation.

Such entries might be held in intermediate TLB caching structures that are used during a translation table walk and that are distinct from the data caches in that they are not required to be invalidated as the result of writes of the data. The architecture makes no restriction on the form of these intermediate TLB caching structures when these caches are indexed by their input address. The architecture does not restrict having either:

- Translation table entry caching that is indexed by the physical address of the location holding the translation table entry.

- Translation table entry caching that is used for stage 1 translations and is indexed by the intermediate physical address of the location holding the translation table entry. However, FEAT_nTLBPA allows software discoverability of whether such caches exist, such that if FEAT_nTLBPA is implemented, such caching is not implemented.

If all of the following are true, a TLB maintenance instruction will ensure that any physical address or intermediate physical address indexed cached copies of translation table entries are invalidated for a PE:

- The TLB maintenance instruction applies to that PE with the context information that is relevant to translation table entry caching that is either:
  - Indexed by the physical address of the location holding the translation table entry.
  - Stage 1 translation information that is indexed by the intermediate physical address of the location holding the translation table entry.

- FEAT_nTLBPA is not implemented.

Note

Any TLB caching based on the physical address or intermediate physical address obeys the other rules regarding the caching to TLB entries described in this manner, including restrictions on types of entries that cannot be held in a TLB, and a requirement that entries held in a TLB are distinguished by context information such as translation regime, VMID, and ASID.

The architecture does not intend to restrict the form of TLB caching structures used for holding translation table entries. In particular for translation regimes that involve two stages of translation, it recognizes that such caching structures might contain:

- At any level of the translation table walk, entries containing information from stage 1 translation table entries.
• In an implementation that includes EL2:
  — At any level of the translation table walk, entries containing information from stage 2 translation table entries.
  — At any level of the translation table walk, entries combining information from both stage 1 and stage 2 translation table entries.

Note
For the purpose of TLB maintenance, the term TLB entry denotes any structure, including temporary working registers in translation table walk hardware, that holds a translation table entry.

For the TLB maintenance instructions:
• If a TLB maintenance instruction is required to apply to stage 1 entries then it must apply to any cached entry in the caching structures that includes any stage 1 information that would be used to translate the address being invalidated, including any entry that combines information from both stage 1 and stage 2 translation table entries.

  Note
  — Where stage 1 information has been cached in multiple TLB entries, as could occur from splintering a page when caching in the TLB, then the invalidation must apply to each cached entry containing stage 1 information from the page that is used to translate the address being invalidated, regardless of whether or not that cached entry would be used to translate the address being invalidated.
  — As stated in Global and process-specific translation table entries on page G5-9216, translation table entries from levels of translation other than the final level are treated as being non-global. Arm expects that, in at least some implementations, cached copies of levels of the translation table walk other than the last level are tagged with their ASID, regardless of whether the final level is global. This means that TLB invalidations that involve the ASID require the ASID to match such entries to perform the required invalidation.

• If a TLB maintenance instruction is required to apply to stage 2 entries only, then:
  — It is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.
  — It must apply to caching structures that contain information only from stage 2 translation table entries.

• If a TLB maintenance instruction is required to apply to both stage 1 and stage 2 entries, then it must apply to any entry in the caching structures that includes information from either a stage 1 translation table entry or a stage 2 translation table entry, including any entry that combines information from both stage 1 and stage 2 translation table entries.

Table G5-23 on page G5-9232 summarizes the required effect of the AArch32 TLB maintenance instructions, that operate only on TLBs on the PE that executes the instruction. Additional TLB maintenance instructions that:
• Apply across all PEs in the same Inner Shareable domain. Each instruction shown in the table has an Inner Shareable equivalent, identified by an IS suffix. For example, the Inner Shareable equivalent of TLBIALL is TLBIALLIS. See also EL2 forced broadcasting of TLB maintenance instructions on page G5-9234.
• Can apply to separate Instruction or Data TLBs. These instructions are indicated by a footnote to the table. Arm deprecates any use of these instructions.

  Note
  • The architecture permits a TLB invalidation instruction to affect any unlocked entry in the TLB. Table G5-23 on page G5-9232 defines only the entries that each instruction must invalidate.
  • All TLB instructions, including those that operate on a VA match, operate as described regardless of the value of SCTLR.M.
When interpreting the table:

**Related operations**

Each instruction description applies also to any equivalent instruction that either:

- Applies to all PEs in the same Inner Shareable domain.
- Applies only to a data TLB, or only to an instruction TLB.

So, for example, the TLBIALL instruction description applies also to TLBIALLIS, ITLBIALL, and DTLBIALL.

*TLB maintenance system instructions* on page K17-11907 lists all of the TLB maintenance instructions.

**Matches the VA**

Means the VA argument for the instruction must match the VA value in the TLB entry.

**Matches the ASID**

Means the ASID argument for the instruction must match the ASID in use when the TLB entry was assigned.

**Matches the current VMID**

Means the current VMID must match the VMID in use when the TLB entry was assigned.

The dependency on the VMID applies even when the value of HCR.VM is 0, including situations where there is no use of virtualization. However, VTTBR.VMID resets to zero, meaning there is a valid VMID from reset.

**Execution at EL2**

Descriptions of operations at EL2 apply only to implementations that include EL2.

For the definitions of the translation regimes referred to in the table see *About VMSAv8-32* on page G5-9146.

### Table G5-23 Effect of the TLB maintenance instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Executed from</th>
<th>Effect, must invalidate any entry that matches all stated conditions&lt;sup&gt;a&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLBIALL&lt;sup&gt;b&lt;/sup&gt;</td>
<td>Secure PL1</td>
<td>All entries for the Secure PL1&amp;0 translation regime. That is, all entries that were allocated in Secure state.</td>
</tr>
<tr>
<td>Non-secure PL1</td>
<td>Hyp</td>
<td>All entries for stage 1 of the Non-secure PL1&amp;0 translation regime that match the current VMID.</td>
</tr>
</tbody>
</table>
| TLBIMVA<sup>b</sup> | Secure PL1 | Any entry for the Secure PL1&0 translation regime that both:
- Matches the VA argument.
- Matches the ASID argument, or is global. |
| Non-secure PL1 or Hyp | Any entry for stage 1 of the Non-secure PL1&0 translation regime to which all of the following apply. The entry:
- Matches the VA argument.
- Matches the ASID argument, or is global.
- Matches the current VMID. |
## Table G5-23 Effect of the TLB maintenance instructions (continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Executed from</th>
<th>Effect, must invalidate any entry that matches all stated conditions&lt;sup&gt;a&lt;/sup&gt;</th>
</tr>
</thead>
</table>
| TLBIASID<sup>b</sup> Secure | PL1 | Any entry for the Secure PL1&0 translation regime that matches the specified ASID and either:  
- Is from a level of lookup above the final level.  
- Is a non-global entry from the final level of lookup. |
| Non-secure | PL1 or Hyp | Any entry for stage 1 of the Non-secure PL1&0 translation regime that both:  
- Matches the specified ASID and either:  
  - Is from a level of lookup above the final level.  
  - Is a non-global entry from the final level of lookup.  
- Matches the current VMID. |
| TLBIMVAA | Secure | PL1 | Any entry for the Secure PL1&0 translation regime that matches the VA argument. |
| Non-secure | PL1 or Hyp | Any entry for stage 1 of the Non-secure PL1&0 translation regime that both:  
- Matches the VA argument.  
- Matches the current VMID. |
| TLBIALLNSNH<sup>c</sup> Secure | Monitor | All entries for stage 1 or stage 2 of the Non-secure PL1&0 translation regime, regardless of the associated VMID. |
| Non-secure | Hyp | All entries for the Non-secure EL2 translation regime. That is, any entry that was allocated in Non-secure state from Hyp mode. |
| TLBIALLH<sup>c</sup> Secure | Monitor | All entries for the Non-secure EL2 translation regime that matches the VA argument. |
| Non-secure | Hyp | Any entry for stage 1 of the Secure PL1&0 translation regime that is from the last level of the translation table walk and both:  
- Matches the VA argument.  
- Matches the ASID argument, or is global. |
| Non-secure | PL1 or Hyp | Any entry for stage 1 of the Non-secure PL1&0 translation regime that is from the last level of the translation table walk and to which all of the following apply. The entry:  
- Matches the VA argument.  
- Matches the ASID argument, or is global.  
- Matches the current VMID. |
| TLBIMVAAL Secure | PL1 | Any entry for stage 1 of the Secure PL1&0 translation regime that is from the last level of the translation table walk and matches the VA argument. |
| Non-secure | PL1 or Hyp | Any entry for stage 1 of the Non-secure PL1&0 translation regime that is from the last level of the translation table walk and both:  
- Matches the VA argument.  
- Matches the current VMID. |
| TLBIMVAH<sup>c</sup> Secure | Monitor | Any entry for the Non-secure EL2 translation regime that matches the VA argument. |
| Non-secure | Hyp | |

<sup>a</sup> Note: Instructions with the prefix "TLBIM" invalidate entries that match all stated conditions, not just those that match ASID.
EL2 forced broadcasting of TLB maintenance instructions

In an implementation that includes EL2, when the value of HCR.FB is 1, the TLB maintenance instructions that are not broadcast across the Inner Shareable domain are forced to operate across the Inner Shareable domain when executed in a Non-secure PL1 mode. For example, when the value of HCR.FB is 1, a TLBIMVA instruction executed in a Non-secure PL1 mode performs the same invalidation as the invalidation performed by a TLBIMVAIS instruction.

TLB maintenance with different translation granule sizes

If a TLB maintenance instruction specifying a VA affecting the EL2 translation regime is broadcast from a PE using AArch32 to a PE using AArch64 using a translation granule size that is different from the AArch32 translation granule size for that same translation regime, the TLB maintenance instruction is not required to perform any invalidation on the recipient PE.

If a TLB maintenance instruction specifying a VA affecting the PL1 translation regime is broadcast from a PE using AArch32 using one translation granule size for that translation regime for a particular ASID, VMID (if applicable), and Security state, to a PE using AArch64 where EL1 for the same ASID, VMID (if applicable), and Security state, is using a translation granule size that is different from the AArch32 translation granule size, the TLB maintenance instruction is not required to perform any invalidation on the recipient PE.
G5.10 Caches in VMSAv8-32

The Arm architecture describes the required behavior of an implementation of the architecture. As far as possible it does not restrict the implemented microarchitecture, or the implementation techniques that might achieve the required behavior.

Maintaining this level of abstraction is difficult when describing the relationship between memory address translation and caches, especially regarding the indexing and tagging policy of caches. This section:

• Summarizes the architectural requirements for the interaction between caches and memory translation.
• Gives some information about the likely implementation impact of the required behavior.

The following sections give this information:

• Data and unified caches.
• Instruction caches.

In addition Cache maintenance requirement created by changing translation table attributes on page G5-9237 describes the cache maintenance required after updating the translation tables to change the attributes of an area of memory.

For more information about cache maintenance see:

• AArch32 cache and branch predictor support on page G4-9113. This section describes the Arm cache maintenance instructions.
• Cache maintenance system instructions on page K17-11905. This section summarizes the System register encodings used for these operations when executing in AArch32 state.

G5.10.1 Data and unified caches

For data and unified caches, the use of memory address translation is entirely transparent to any data access other than as described in Mismatched memory attributes on page E2-7208.

This means that the behavior of accesses from the same observer to different VAs, that are translated to the same PA with the same memory attributes, is fully coherent. This means these accesses behave as follows, regardless of which VA is accessed:

• Two writes to the same PA occur in program order.
• A read of a PA returns the value of the last successful write to that PA.
• A write to a PA that occurs, in program order, after a read of that PA, has no effect on the value returned by that read.

The memory system behaves in this way without any requirement to use barrier or cache maintenance instructions.

In addition, if cache maintenance is performed on a memory location, the effect of that cache maintenance is visible to all aliases of that physical memory location.

These properties are consistent with implementing all caches that can handle data accesses as Physically-indexed, physically-tagged (PIPT) caches.

G5.10.2 Instruction caches

In the Arm architecture, an instruction cache is a cache that is accessed only as a result of an instruction fetch. Therefore, an instruction cache is never written to by any load or store instruction executed by the PE.

The Arm architecture permits different behaviors for instruction caches. These are identified by descriptions of the associated expected implementation. The following subsections describe the behavior associated with these cache types, including any occasions where explicit cache maintenance is required to make the use of memory address translation transparent to the instruction cache:

• PIPT (Physically-indexed, physically-tagged) instruction caches on page G5-9236.
• VPIPT (VMID-aware PIPT ) instruction caches on page G5-9236.
• VIPT (Virtually-indexed, physically-tagged) instruction caches on page G5-9236.
In AArch32 state, the CTR.L1Ip field identifies the form of the instruction caches.

Note

For software to be portable between implementations that might use any of PIPT instruction caches, VPIPT instruction caches, or VIPT instruction caches, software must invalidate the instruction cache whenever any condition occurs that would require instruction cache maintenance for at least one of the instruction cache types.

PIPT (Physically-indexed, physically-tagged) instruction caches

For a PIPT instruction cache:

- The use of memory address translation is entirely transparent to all instruction fetches other than as described in Mismatched memory attributes on page E2-7208.
- If cache maintenance is performed on a memory location, the effect of that cache maintenance is visible to all aliases of that physical memory location.

An implementation that provides PIPT instruction caches implements the IVIPT Extension, see The IVIPT architecture Extension on page G5-9237.

VPIPT (VMID-aware PIPT) instruction caches

An Armv8.2 implementation can implement VPIPT instruction caches. If it does so then it is described as implementing FEAT_VPIPT.

The CTR.L1Ip field identifies the implemented cache type, meaning it identifies whether FEAT_VPIPT is implemented.

For a VPIPT instruction cache:

- If VMIDs are being used for the current Security state, instruction fetches from EL1 and EL0 are only permitted to hit in the cache if the instruction fetch is made using the VMID that was used when the entry in the instruction cache was fetched.
- If VMIDs are being used for the current Security state, an instruction cache maintenance instruction executed at EL0 or at EL1 is required to have an effect on entries in the instruction cache only if those entries were fetched using the VMID that is current when the cache maintenance instruction is executed.

All other requirements for the use of cache maintenance instructions are the same as for PIPT (Physically-indexed, physically-tagged) instruction caches.

An implementation that provides VPIPT instruction caches implements the IVIPT Extension, see The IVIPT architecture Extension on page G5-9237.

VIPT (Virtually-indexed, physically-tagged) instruction caches

For a VIPT instruction cache:

- The use of memory address translation is transparent to all instruction fetches other than for the effect of memory address translation on instruction cache invalidate by address operations or as described in Mismatched memory attributes on page E2-7208.

Note

Cache invalidation is the only cache maintenance instruction that can be performed on an instruction cache.

- If instruction cache invalidation by address is performed on a memory location, the effect of that invalidation is visible only to the VA supplied with the operation. The effect of the invalidation might not be visible to any other VA aliases of that physical memory location.
The only architecturally-guaranteed way to invalidate all aliases of a PA from a VIPT instruction cache is to invalidate the entire instruction cache.

An implementation that provides VIPT instruction caches implements the IVIPT Extension, see The IVIPT architecture Extension.

The IVIPT architecture Extension

Any permitted instruction cache implementation can be described as implementing the IVIPT Extension to the Arm architecture.

The formal definition of the Arm IVIPT Extension is that it reduces the instruction cache maintenance requirement to the following condition:

• Instruction cache maintenance is required only after writing new data to a PA that holds an instruction.

Note

Previous versions of the Arm architecture have permitted an instruction cache option that does not implement the Arm IVIPT Extension.

G5.10.3 Cache maintenance requirement created by changing translation table attributes

Any change to the translation tables to change the attributes of an area of memory can require maintenance of the translation tables, as described in General TLB maintenance requirements on page G5-9220. If the change affects the cacheability attributes of the area of memory, including any change between Write-Through and Write-Back attributes, software must ensure that any cached copies of affected locations are removed from the caches, typically by cleaning and invalidating the locations from the levels of cache that might hold copies of the locations affected by the attribute change. Any of the following changes to the inner cacheability or outer cacheability attribute creates this maintenance requirement:

• Write-Back to Write-Through.
• Write-Back to Non-cacheable.
• Write-Through to Non-cacheable.
• Write-Through to Write-Back.

The cache clean and invalidate avoids any possible coherency errors caused by mismatched memory attributes.

Similarly, to avoid possible coherency errors caused by mismatched memory attributes, the following sequence must be followed when changing the Shareability attributes of a cacheable memory location:

1. Make the memory location Non-cacheable, Outer Shareable.
2. Clean and invalidate the location from them cache.
3. Change the Shareability attributes to the required new values.
G5.11 VMSAv8-32 memory aborts

In a VMSAv8-32 implementation, the following mechanisms cause a PE to take an exception on a failed memory access:

- **Debug exception**: An exception caused by the debug configuration, see Chapter G2 AArch32 Self-hosted Debug.

- **Alignment fault**: An Alignment fault is generated if the address used for a memory access does not have the required alignment for the operation. For more information, see Unaligned data access on page E2-7193 and Alignment faults on page G5-9247.

- **MMU fault**: An MMU fault is a fault generated by the fault checking sequence for the current translation regime. See Types of MMU faults on page G5-9239.

- **External abort**: Any memory system fault other than a Debug exception, an Alignment fault, or an MMU fault.

Collectively, these mechanisms are called aborts. Chapter G2 AArch32 Self-hosted Debug and Chapter H3 Halting Debug Events describe Debug exceptions, and the remainder of this section describes Alignment faults, MMU faults, and External aborts.

An access that causes an abort is said to be aborted, and uses the Fault Address Registers (FARs) and Fault Status Registers (FSRs) or Exception Syndrome Registers (ESRs) to record context information.

The exception generated on a synchronous memory abort:
- **On an instruction fetch is called the Prefetch Abort exception.**
- **On a data access is called the Data Abort exception.**

Note: The Prefetch Abort exception applies to any synchronous memory abort on an instruction fetch. It is not restricted to speculative instruction fetches.

The Exception level and PE mode that a VMSAv8-32 memory abort is taken to depends on the translation regime and stage that generate the abort. The fault context is dependent on whether:
- **The abort is reported as a Prefetch Abort or as a Data Abort.**
- **The exception is taken from the same or a lower Exception level.**

Note: A memory access from AArch32 state may be subject to one or more VMSAv8-64 translation stages. For example, a Non-secure EL0 access when EL1 is using AArch64 is subject to both stages of the VMSAv8-64 Non-secure EL1&0 translation regime. A memory abort generated on a VMSAv8-64 translation stage is handled as described in Memory aborts on page D8-5180.

For more information, see Routing of aborts taken to AArch32 state on page G1-8948.

External aborts can be reported synchronously or asynchronously. Asynchronous External aborts are reported using the SError interrupt. For more information, see External aborts on page G4-9139.

In AArch32 state, asynchronous memory aborts are a type of External abort, and are treated as a type of Data Abort exception.

The following sections describe the abort mechanisms:
- Types of MMU faults on page G5-9239.
- VMSAv8-32 MMU fault terminology on page G5-9241.
- The MMU fault-checking sequence on page G5-9242.
- Alignment faults on page G5-9247.
- External abort on a translation table walk on page G5-9247.
- AArch32 state prioritization of synchronous aborts from a single stage of address translation on page G5-9248.
An access that causes an abort is said to be aborted. On an abort, System registers are used to record context information. For more information, see Exception reporting in a VMSAv8-32 implementation on page G5-9251.

G5.11 Types of MMU faults

This section describes the faults that might be detected during one of the fault-checking sequences described in The MMU fault-checking sequence on page G5-9242. Unless indicated otherwise, information in this section applies to the fault checking sequences for both the Short-descriptor translation table format and the Long-descriptor translation table format.

MMU faults are always synchronous.

When an MMU fault generates an abort for a region of memory, no memory access is made if that region is or could be marked as any type of Device memory.

The MMU faults that might be detected during a fault checking sequence are:

- Permission fault.
- Translation fault.
- Address size fault.
- Access flag fault.
- Domain fault, Short-descriptor translation tables only.
- TLB conflict abort.

See also External abort on a translation table walk on page G5-9247.

--- Note ---

- Although the TLB conflict abort is classified as an MMU fault, it is described in the section Translation Lookaside Buffers (TLBs) on page G5-9216.

- In VMSAv8-64 an External abort on a translation table walk is classified as an MMU fault. However, in VMSAv8-32, for consistency with earlier versions of the architecture these aborts are not classified as MMU faults.

---

Permission fault

A Permission fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. See About access permissions on page G5-9192 for information about conditions that cause a Permission fault.

--- Note ---

When using the Short-descriptor translation table format, the Translation Table descriptors are checked for Permission faults only for accesses to memory regions in Client domains.

A TLB might hold a translation table entry that cause a Permission fault. Therefore, if the handling of a Permission fault results in an update to the associated translation tables, the software that updates the translation tables must invalidate the appropriate TLB entry, to prevent the stale information in the TLB being used on a subsequent memory access. For more information, see the translation table entry update examples in Ordering and completion of TLB maintenance instructions on page G5-9223.

In an implementation that includes EL2, this maintenance requirement applies to Permission faults in both stage 1 and stage 2 translations.

Cache or branch predictor maintenance operations cannot cause a Permission fault, except that:

- A stage 1 translation table walk performed as part of a cache or branch predictor maintenance operation can generate a stage 2 Permission fault as described in Stage 2 fault on a stage 1 translation table walk.

- When FEAT_CMOW is implemented and the Effective value of HCRX_EL2.CMOW is 1, the DCCIMVAC and ICIMVAC cache maintenance instructions can generate a stage 2 Permission fault if they do not have read and write permission at EL1 or EL0. See Permission fault on page D8-5183.
Translation fault

A Translation fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. A Translation fault is generated if bits[1:0] of a Translation Table descriptor identify the descriptor as either a Fault encoding or a reserved encoding. For more information, see:

- VMSAv8-32 Short-descriptor Translation Table format descriptors on page G5-9164.
- VMSAv8-32 Long-descriptor Translation Table format descriptors on page G5-9173.

In addition, a Translation fault is generated if the input address for a translation either does not map onto an address range of a TTBR, or the TTBR range that it maps onto is disabled. In these cases the fault is reported as a level 1 Translation fault on the translation stage at which the mapping to a region described by a TTBR failed.

The architecture guarantees that any translation table entry that causes a Translation fault is not cached, meaning the TLB never holds such an entry. Therefore, when a Translation fault occurs, the fault handler does not have to perform any TLB maintenance instructions to remove the faulting entry.

A data or unified cache maintenance by VA instruction can generate a Translation fault. However:

- If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a data or unified cache maintenance by VA to the Point of Coherency instruction can generate a Translation fault.
- If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether a data or unified cache clean by VA to the Point of Unification instruction can generate a Translation fault.

It is IMPLEMENTATION DEFINED whether an instruction cache invalidate by VA operation can generate a Translation fault.

It is IMPLEMENTATION DEFINED whether a branch predictor maintenance operation can generate a Translation fault.

Address size fault

An Address size fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. An Address size fault is generated if the translation table entries or the TTBR for the stage of translation have nonzero address bits above the most significant bit of the maximum output address size. Because VMSAv8-32 supports a maximum PA and IPA size of 40 bits, this means any case where a translation table entry or the TTBR holds an address for which A[47:40] is nonzero generates an Address size fault.

A data or unified cache maintenance by VA instruction can generate an Address size fault. However:

- If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a data or unified cache maintenance by VA instruction can generate an Address size fault.
- If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether a data or unified cache clean by VA instruction can generate an Address size fault.

It is IMPLEMENTATION DEFINED whether an instruction cache invalidate by VA operation can generate an Address size fault.

It is IMPLEMENTATION DEFINED whether a branch predictor maintenance operation can generate an Address size fault.

The architecture guarantees that any translation table entry that causes an Address size fault is not cached, meaning the TLB never holds such an entry. Therefore, when an Address size fault occurs, the fault handler does not have to perform any TLB maintenance instructions to remove the faulting entry.

Access flag fault

An Access flag fault can be generated at any level of lookup, and the reported fault code identifies the lookup level. An Access flag fault is generated only if all of the following apply:

- The translation tables support an Access flag bit:
  - The Short-descriptor format supports an Access flag only when SCTLR.AFE is set to 1.
  - The Long-descriptor format always supports an Access flag.
• A Translation Table descriptor with the Access flag bit set to 0 is loaded.

For more information about the Access flag bit see:
• *VMSAv8-32 Short-descriptor Translation Table format descriptors* on page G5-9164
• *VMSAv8-32 Long-descriptor Translation Table format descriptors* on page G5-9173.

The architecture guarantees that any translation table entry that causes an Access flag fault is not cached, meaning the TLB never holds such an entry. Therefore, when an Access flag fault occurs, the fault handler does not have to perform any TLB maintenance instructions to remove the faulting entry.

Whether any cache maintenance instruction by VA can generate Access flag faults is IMPLEMENTATION DEFINED.

Whether branch predictor invalidate by VA operations can generate Access flag faults is IMPLEMENTATION DEFINED.

For more information, see *The Access flag* on page G5-9200.

### Domain fault, Short-descriptor format translation tables only

When using the Short-descriptor translation table format, a Domain fault can be generated at level 1 or level 2 of lookup. The reported fault code identifies the lookup level. The conditions for generating a Domain fault are:

**Level 1**

When a level 1 descriptor fetch returns a valid Section level 1 descriptor, the domain field of that descriptor is checked against the DACR. A level 1 Domain fault is generated if this check fails.

**Level 2**

When a level 2 descriptor fetch returns a valid level 2 descriptor, the domain field of the level 1 descriptor that required the level 2 fetch is checked against the DACR, and a level 2 Domain fault is generated if this check fails.

For more information, see *Domains, Short-descriptor format only* on page G5-9199.

Domain faults cannot occur on cache or branch predictor maintenance operations.

A TLB might hold a translation table entry that cause a Domain fault. Therefore, if the handling of a Domain fault results in an update to the associated translation tables, the software that updates the translation tables must invalidate the appropriate TLB entry, to prevent the stale information in the TLB being used on a subsequent memory access. For more information, see the translation table entry update examples in *Ordering and completion of TLB maintenance instructions* on page G5-9223.

Any change to the DACR must be synchronized by a Context synchronization event. For more information, see *Synchronization of changes to AArch32 System registers* on page G8-9326.

### VMSAv8-32 MMU fault terminology

The Armv7 Large Physical Address Extension introduced new terminology for faults on a stage of address translation, to provide consistent terminology across all implementations. Table G5-24 on page G5-9241 shows the terminology used in this manual for an MMU faults, compared with older Arm documentation. The current terms are the same for faults that occur with the Short descriptor translation table format and with the Long-descriptor format, and also apply to faults in a level 3 lookup when using the Long-descriptor translation table format.

<table>
<thead>
<tr>
<th>Current term</th>
<th>Old term</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1 Translation fault</td>
<td>Section Translation fault</td>
<td></td>
</tr>
<tr>
<td>Level 2 Translation fault</td>
<td>Page Translation fault</td>
<td></td>
</tr>
<tr>
<td>Level 3 Translation fault</td>
<td>Long descriptor translation table format only</td>
<td></td>
</tr>
</tbody>
</table>

For more information, see *Domains, Short-descriptor format only* on page G5-9199.
In an implementation that includes EL2, MMU faults are also classified by the translation stage at which the fault is generated. This means that a memory access from a Non-secure PL1 or EL0 mode can generate:

- A stage 1 MMU fault, for example, a stage 1 Translation fault.
- A stage 2 MMU fault, for example, a stage 2 Translation fault.

For more information about Prefetch Abort exceptions and Data Abort exceptions see Handling exceptions that are taken to an Exception level using AArch32 on page G1-8929.

In VMSAv8-32, all memory accesses require VA to PA translation. Therefore, when a corresponding stage of address translation is enabled, each access requires a lookup of the Translation Table descriptor for the accessed VA. For more information, see Translation tables on page G5-9158 and subsequent sections of this chapter. MMU fault checking is performed for each level of translation table lookup. If an implementation includes EL2 and is operating in Non-secure state, MMU fault checking is performed for each stage of address translation.

**Note**

In an implementation that includes EL2, if a PE is executing in Non-secure state, the operating system or similar Non-secure system software defines the stage 1 translation tables in the IPA address map, and typically is unaware of the stage 2 translation from IPA to PA. However, each Non-secure stage 1 translation table access is subject to stage 2 address translation, and might be faulted at that stage.

The MMU fault checking sequence is largely independent of the translation table format, as the figures in this section show. The differences are:

**When using the Short-descriptor format**

- There are one or two levels of lookup.
- Lookup always starts at level 1.
• The final level of lookup checks the Domain field of the descriptor and:
  — Faults if there is no access to the Domain.
  — Checks the access permissions only for Client domains.

**When using the Long-descriptor format**

• There are one, two, or three levels of lookup.
• Lookup starts at either level 1 or level 2.
• Domains are not supported. All accesses are treated as Client domain accesses.

The fault-checking sequence shows a translation from an Input address to an Output address. For more information about this terminology, see *About address translation for VMSAv8-32 on page G5-9149.*

--- **Note**

The descriptions in this section do not include the possibility that the attempted address translation generates a TLB conflict abort, as described in *TLB conflict aborts on page G5-9218.*

*Types of MMU faults on page G5-9239* describes the faults that an MMU fault-checking sequence can report.

*Figure G5-15 on page G5-9244* shows the process of fetching a descriptor from the translation table. For the top-level fetch for any translation, the descriptor is fetched only if the input address passes any required alignment check. As the figure shows, in an implementation that includes EL2, if the translation is stage 1 of the Non-secure PL1&0 translation regime, then the descriptor address is in the IPA address map, and is subject to a stage 2 translation to obtain the required PA. This stage 2 translation requires a recursive entry to the fault checking sequence.

--- **Note**

*Figure G5-15 on page G5-9244 and Figure G5-16 on page G5-9245* give an overview of the fault checking performed by the MMU. See *AArch32 state prioritization of synchronous aborts from a single stage of address translation on page G5-9248* for the complete set of possible faults and their prioritization.
Figure G5-15 Fetching the descriptor in a VMSAv8-32 translation table walk

Figure G5-16 on page G5-9245 shows the full VMSAv8-32 fault checking sequence, including the alignment check on the initial access.
Figure G5-16 VMSAv8-32 fault checking sequence
Stage 2 fault on a stage 1 translation table walk

When an implementation that includes EL2 is operating in a Non-secure PL1 or EL0 mode, any memory access goes through two stages of translation:

- Stage 1, from VA to IPA.
- Stage 2, from IPA to PA.

Note

In a virtualized system that is using AArch32, typically, a Guest OS operating in a Non-secure PL1 mode defines the translation tables and translation table register entries controlling the Non-secure PL1&0 stage 1 translations. A Guest OS has no awareness of the stage 2 address translation, and therefore believes it is specifying translation table addresses in the PA map. However, it actually specifies these addresses in its IPA map. Therefore, to support virtualization, translation table addresses for the Non-secure PL1&0 stage 1 translations are always defined in the IPA address map.

On performing a translation table walk for the stage 1 translations, the descriptor addresses must be translated from IPA to PA, using a stage 2 translation. This means that a memory access made as part of a stage 1 translation table lookup might generate, on a stage 2 translation:

- A Translation fault, Access flag fault, or Permission fault.
- A synchronous External abort on the memory access.

If SCR.EA is set to 1, a synchronous External abort is taken to EL3, and if EL3 is using AArch32 it is taken to Secure Monitor mode. Otherwise, these faults are reported as stage 2 memory aborts. When EL2 is using AArch32, HSR.ISS[7] is set to 1, to indicate a stage 2 fault during a stage 1 translation table walk, and the part of the ISS field that might contain details of the instruction is invalid. For more information, see Use of the HSR on page G5-9265.

Alternatively, a memory access made as part of a stage 1 translation table lookup might target an area of memory with the Device memory attribute assigned on the stage 2 translation of the address accessed. When the value of the HCR.PTW bit is 1, such an access generates a stage 2 Permission fault.

Note

- On most systems, such a mapping to a Device memory type on the stage 2 translation is likely to indicate a Guest OS error, where the stage 1 translation table is corrupted. Therefore, it is appropriate to trap this access to the hypervisor.

A TLB might hold entries that depend on the effect of HCR.PTW. Therefore, if HCR.PTW is changed without changing the current VMID, the TLBs must be invalidated before executing in a Non-secure PL1 or EL0 mode. For more information, see Changing HCR.PTW on page G5-9226.

A cache maintenance instruction executed at Non-secure PL1 can cause a stage 1 translation table walk that might generate a stage 2 Permission fault, as described in this section. However:

- If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a cache maintenance by VA instruction can generate a Permission fault in this way.
- If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether a data or unified cache clean by VA to the Point of Unification instruction can generate a Permission fault in this way.

Note

This is an exception to the general rule that a cache maintenance instruction cannot generate a Permission fault.
The level associated with MMU faults

When an MMU fault is from a stage of translation that is using Long-descriptor translation table format, Table G5-25 shows how the LL bits in the STATUS field of DFSR, IFSR, and HSR encode the lookup level associated with the fault.

Table G5-25 Use of LL bits to encode the lookup level at which the fault occurred

<table>
<thead>
<tr>
<th>LL bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Level 0 of translation or translation table base register.</td>
</tr>
<tr>
<td>01</td>
<td>Level 1.</td>
</tr>
<tr>
<td>10</td>
<td>Level 2.</td>
</tr>
<tr>
<td>11</td>
<td>Level 3. When xFSR.STATUS indicates a Domain fault, this value is reserved.</td>
</tr>
</tbody>
</table>

The level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because a stage of address translation is disabled, or because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a level 1 fault.
- For an Access flag fault, the lookup level of the translation table that gave the fault.
- For a Permission fault, including a Permission fault caused by hierarchical permissions, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Block or Page descriptor.

Also see Synchronous External abort errors from address translation caching structures on page G5-9250.

G5.11.4 Alignment faults

The Arm memory architecture requires support for strict alignment checking. This checking is controlled by:

- SCTLR.A, for accesses made from any PE mode other than Hyp mode.
- HSCTLR.A, for accesses made from Hyp mode.

In addition, some instructions do not support unaligned accesses, regardless of the value of SCTLR.A or HSCTLR.A.

Unaligned data access on page E2-7193:

- Defines when Alignment faults are generated, for both values of SCTLR.A or HSCTLR.A.
- Describes the possible generation of Alignment faults on accesses to Device memory by AArch32 Load Multiple or Store Multiple instructions when FEAT_LSMAOC is implemented.

An Alignment fault can occur on an access for which the stage of address translation is disabled.

Any unaligned access to memory region with any Device memory type attribute generates an Alignment fault.

Routing of aborts taken to AArch32 state on page G1-8948 defines the mode to which an Alignment fault is taken.

The prioritization of Alignment faults depends on whether the fault was generated because of an access to a Device memory type, or for another reason. For more information, see AArch32 state prioritization of synchronous aborts from a single stage of address translation on page G5-9248.

G5.11.5 External abort on a translation table walk

An External abort on a translation table walk can be either synchronous or asynchronous. For more information on External aborts, see External aborts on page G4-9139.
An External abort on a translation table walk is reported:

- If the External abort is synchronous, using:
  - A synchronous Prefetch Abort exception if the translation table walk is for an instruction fetch.
  - A synchronous Data Abort exception if the translation table walk is for a data access.
- If the External abort is asynchronous, using an SError interrupt, which is taken as an asynchronous Data Abort exception.

If an implementation reports the error in the translation table walk asynchronously from executing the instruction whose instruction fetch or memory access caused the translation table walk, these aborts behave essentially as interrupts. The aborts are masked when PSTATE.A is set to 1, otherwise they are reported using the Data Abort exception.

**Behavior of External aborts on a translation table walk caused by address translation instructions**

The address translation instructions summarized in [Address translation system instructions](#) on page K17-11906 require translation table walks. An External abort can occur in the translation table walk. The abort generates a Data Abort exception, and can be synchronous or asynchronous. For more information, see [Handling of faults and aborts during an address translation instruction](#) on page G5-9275.

### G5.11.6 AArch32 state prioritization of synchronous aborts from a single stage of address translation

*Exception prioritization for exceptions taken to AArch32 state* on page G1-8932 describes the prioritization of exceptions taken from an Exception level that is using AArch32. This section gives additional information about the prioritization of MMU faults from VMSAv8-32 translation regimes.

If a single instruction generates aborts on more than one memory access, the architecture does not define any prioritization between those aborts.

In general, the Arm architecture does not define when asynchronous events are taken, and therefore the prioritization of asynchronous events is IMPLEMENTATION DEFINED.

**Note**

The priority numbering in this list only shows the relative priorities of aborts from a single stage of address translation in a VMSAv8-32 translation regime. This numbering has no global significance and, for example, does not correlate with the equivalent AArch64 list in [MMU fault prioritization from a single address translation stage](#) on page D8-5190.

For a single stage of translation in a VMSAv8-32 translation regime, the following numbered list shows the priority of the possible memory management faults on a memory access. In this list:

- For memory accesses that undergo two stages of translation, the italic entries show where the faults from the stage 2 translation can occur. A stage 2 fault within a stage 1 translation table walk follows the same prioritization of faults.
- For synchronous External aborts from translation table walks see also [Synchronous External abort errors from address translation caching structures](#) on page G5-9250.

The priority order, from highest priority to lowest priority, is:

1. Alignment fault not caused by memory type. This is possible for a stage 1 translation only.
2. Translation fault due to the input address being out of the address range to be translated or requiring an AArch32 TTBR that is disabled. This includes VTCR.SL0 being inconsistent with VTCR.T0SZ or programmed to a reserved value.
3. Address size fault on an AArch32 TTBR caused by the PA being out of the range implemented.
4. *Second stage abort on a level 1 lookup of a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the PA being out of the range implemented.* This is second stage abort during a first stage translation table walk.
5. Synchronous parity or ECC error on a level 1 lookup of a translation table walk.
6. Synchronous External abort on a level 1 lookup level of a translation table walk.
7. Translation fault on a level 1 translation table entry.
8. Address size fault on a level 1 lookup translation table entry caused by the output address being out of the range implemented.
9. Second stage abort on a level 2 lookup of a a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the PA being out of the range implemented. This is second stage abort during a first stage translation table walk.
10. Synchronous parity or ECC error on a level 2 lookup of a translation table walk.
11. Synchronous External abort on a level 2 lookup level of a translation table walk.
12. Translation fault on a level 2 translation table entry.
13. Address size fault on a level 2 lookup translation table entry caused by the output address being out of the range implemented.
14. Second stage abort on a level 3 lookup of a a stage 1 table walk. When stage 2 address translation is enabled this includes an Address size fault caused by the PA being out of the range implemented. This is second stage abort during a first stage translation table walk.
15. Synchronous parity or ECC error on a level 3 lookup of a translation table walk.
16. Synchronous External abort on a level 3 lookup level of a translation table walk.
17. Translation fault on a level 3 translation table entry.
18. Address size fault on a level 3 lookup translation table entry caused by the output address being out of the range implemented.
19. Access Flag fault.
20. Alignment fault caused by the memory type.

--- Note ---
Domain faults are possible only when using the VMSAv8-32 Short-descriptor translation table format, see Domain fault, Short-descriptor format translation tables only on page G5-9241.

22. Permission fault.
23. A fault from the stage 2 translation of the memory access. When stage 2 address translation is enabled this includes an Address size fault caused by the PA being out of the range implemented.
24. Synchronous parity or ECC error on the memory access.
25. Synchronous External abort on the memory access.

--- Note ---
• The prioritization of TLB Conflict aborts is IMPLEMENTATION DEFINED, as the exact cause of these aborts depends on the form of TLBs implemented. However, the TLB conflict abort must have higher priority than any abort that depends on a value held in the TLB.
• The prioritization of IMPLEMENTATION DEFINED MMU faults for a Load-Exclusive or Store-Exclusive to an unsupported memory type is IMPLEMENTATION DEFINED.

See also The MMU fault-checking sequence on page G5-9242.
Synchronous External abort errors from address translation caching structures

A caching structure used for caching translation table walks might support:

- An arbitrary number of levels of translation table lookup.
- One or more stages of translation, which might not correspond to the stages of an address translation lookup.

This might mean that, on a synchronous External abort arising from the caching structure, such as from a parity or ECC error, the PE cannot precisely determine one or both of the translation stage and level of lookup at which the error occurred. In this case:

- If the PE cannot determine precisely the translation stage at which the error occurred, it is reported and prioritized as a stage 1 error.

- If the PE cannot determine precisely the lookup level at which the error occurred, the level is reported and prioritized as either:
  — The lowest-numbered level that could have given rise to the error.
  — Level 1 if the PE cannot determine any information about the level.
G5.12 Exception reporting in a VMSAv8-32 implementation

This section describes exception reporting, in AArch32 state, in a VMSAv8-32 implementation. That is, it describes only the reporting of exceptions that are taken to an Exception level that is using AArch32. EL2 provides an enhanced reporting mechanism for exceptions taken to the Non-secure EL2 mode, Hyp mode. This means that, for VMSAv8-32, the exception reporting depends on the mode to which the exception is taken.

Note

The enhanced reporting mechanism for exceptions that are taken to Hyp mode is generally similar to the reporting of exceptions that are taken to an Exception level that is using AArch64.

About exception reporting introduces the general approach to exception reporting, and the following sections then describe exception reporting at different privilege levels:

- Reporting exceptions taken to PL1 modes on page G5-9252.
- Fault reporting in PL1 modes on page G5-9255.
- Summary of register updates on faults taken to PL1 modes on page G5-9260.
- Reporting exceptions taken to Hyp mode on page G5-9261.
- Use of the HSR on page G5-9265.
- Summary of register updates on exceptions taken to Hyp mode on page G5-9268.

Note

The registers used for exception reporting also report information about debug exceptions. For more information, see:

- Data Abort exceptions, taken to a PL1 mode on page G5-9253.
- Prefetch Abort exceptions, taken to a PL1 mode on page G5-9255.
- Reporting exceptions taken to Hyp mode on page G5-9261.

G5.12.1 About exception reporting

In an implementation that includes EL2 and EL3, exceptions can be taken to:

- Monitor mode, if EL3 is using AArch32.
- Hyp mode, if EL2 is using AArch32.
- A Secure or Non-secure PL1 mode.

Monitor mode is a PL1 mode, but:

- It is accessible only when EL3 is using AArch32.
- It is present only in Secure state.
- When EL3 is using AArch32, System register controls route some exceptions from Non-secure state to Monitor mode. These are the only cases where taking an exception to an Exception level that is using AArch32 changes the Security state of the PE.

Exception reporting in Hyp mode differs significantly from that in the other modes, but in general, exception reporting returns:

- Information about the exception:
  - On taking an exception to Hyp mode, the Hyp Syndrome Register (HSR), returns syndrome information.
  - On taking an exception to any other mode, a Fault Status Register (FSR) returns status information.
- For synchronous exceptions, one or more addresses associated with the exceptions, returned in Fault Address Registers (FARs). For a permitted exception to this requirement see Fault address reporting on synchronous External aborts on page G5-9252.

In all modes, additional IMPLEMENTATION DEFINED registers can provide additional information about exceptions.
--- Note ---

- **PE mode for taking exceptions** on page G1-8939 describes how the mode to which an exception is taken is determined.

- **EL2** provides:
  - Specific exception types, that can only be taken from Non-secure PL1 and EL0 modes, and are always taken to Hyp mode.
  - Routing controls that can route some exceptions from Non-secure PL1 and EL0 modes to Hyp mode.

  These exceptions are reported using the same mechanism as the Hyp mode reporting of VMSAv8-32 memory aborts, as described in this section.

---

Memory system faults generate either a Data Abort exception or a Prefetch Abort exception, as summarized in:

- **Reporting exceptions taken to PL1 modes**.
- **Memory fault reporting in Hyp mode** on page G5-9263.

On an access that might have multiple aborts, the MMU fault checking sequence and the prioritization of aborts determine which abort occurs. For more information, see **The MMU fault-checking sequence** on page G5-9242 and **AArch32 state prioritization of synchronous aborts from a single stage of address translation** on page G5-9248.

### Fault address reporting on synchronous External aborts

The general architectural requirement is that, on a synchronous abort, the faulting address is recorded in a **Fault Address Register (FAR)**. This requirement is relaxed for the case of a synchronous External abort that is not a synchronous External abort on a translation table walk. In this case only:

- It is IMPLEMENTATION DEFINED whether the faulting address is recorded in a FAR.
- A bit in a fault reporting register, the FnV bit, indicates whether a valid address is recorded.

For exceptions taken to an Exception level that is using AArch32, the details of this reporting depend on whether the exception is taken to:

- A PL1 mode, as described in **Reporting exceptions taken to PL1 modes**.
- Hyp mode, as described in **Reporting exceptions taken to Hyp mode** on page G5-9261.

### G5.12.2 Reporting exceptions taken to PL1 modes

The following sections give general information about the reporting of exceptions when they are taken to a Secure or Non-secure PL1 mode:

- **Registers used for reporting exceptions taken to PL1 modes**.
- **Data Abort exceptions, taken to a PL1 mode** on page G5-9253.
- **Prefetch Abort exceptions, taken to a PL1 mode** on page G5-9255.

**Fault reporting in PL1 modes** on page G5-9255 then describes the fault reporting in these modes, including the encodings used for reporting the faults.

--- Note ---

*Security state, Exception levels, and AArch32 execution privilege* on page G1-8908 describes how the Secure and Non-secure PL1 modes map onto the Exception levels.

---

### Registers used for reporting exceptions taken to PL1 modes

AArch32 state defines the following registers, and register encodings, for exceptions taken to PL1 modes:

- The **DFSR** holds information about a Data Abort exception.
- The **DFAR** holds the faulting address for some synchronous Data Abort exceptions.
- The **IFSR** holds information about a Prefetch Abort exception.
- The **IFAR** holds the faulting address for some synchronous Prefetch Abort exceptions.
In addition, if implemented, the optional ADFSR and AIFSR can provide additional fault information, see Auxiliary Fault Status Registers.

**Auxiliary Fault Status Registers**

AArch32 state defines the following Auxiliary Fault Status Registers:

- The Auxiliary Data Fault Status Register, ADFSR.
- The Auxiliary Instruction Fault Status Register, AIFSR.

The position of these registers is architecturally-defined, but the content and use of the registers is IMPLEMENTATION DEFINED. An implementation can use these registers to return additional fault status information. An example use of these registers is to return more information for diagnosing parity or ECC errors.

An implementation that does not need to report additional fault information must implement these registers as RES0. This ensures that an attempt to access these registers from software executing at PL1 does not cause an Undefined Instruction exception.

**Data Abort exceptions, taken to a PL1 mode**

On taking a Data Abort exception to a PL1 mode:

- If the exception is on an instruction cache or branch predictor maintenance operation by VA, its reporting depends on the value of TTBCR.EAE. For more information about the registers used when reporting the exception, see Data Abort on an instruction cache or branch predictor maintenance instruction by VA on page G5-9254.
- Otherwise, the DFSR is updated with details of the fault, including the appropriate Fault status code. If the Data Abort exception is synchronous, DFSR.WnR is updated to indicate whether the faulted instruction was a read or a write. However, if the fault is on a cache maintenance instruction, or on an address translation instruction, WnR is set to 1, to indicate a fault on a write instruction, and the CM bit is set to 1.

  If the Data Abort is external, then DFSR provides fields for additional classification of the abort, see Provision for classification of External aborts on page G4-9139.

  If the RAS Extension is implemented, and the exception is a virtual SError interrupt exception, the classification reported in DFSR is taken from VDFSR or VSESR_EL2. For more information, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for the ARMv8-A architecture profile.

  See the register description for more information about the returned fault information. See also Data Abort on a Watchpoint exception on page G5-9254.

- Synchronous, the DFAR is updated with the VA that caused the exception, but see Fault address reporting on synchronous External aborts on page G5-952 for a permitted exception to this requirement.
- Asynchronous, the DFAR becomes UNKNOWN.

DFSR.WnR and DFSR.CM are UNKNOWN on an asynchronous Data Abort exception.

For all Data Abort exceptions, if the implementation includes EL3, the Security state of the PE in the mode to which the Data Abort exception is taken determines whether the Secure or Non-secure DFSR and DFAR are updated.
**Data Abort on an instruction cache or branch predictor maintenance instruction by VA**

If an instruction cache invalidation by VA or branch predictor invalidation by VA operation generates a Data Abort exception that is taken to a PL1 mode, the DFAR is updated to hold the faulting VA. However, the reporting of the fault depends on the value of TTBCR.EAE:

**TTBCR.EAE == 0**

When the value of TTBCR.EAE is 0, it is IMPLEMENTATION DEFINED which of the following is used when reporting the fault:

- The DFSR indicates an Instruction cache maintenance instruction fault, and the IFSR is valid and indicates the cause of the fault, a Translation fault or Access flag fault.
- The DFSR indicates the cause of the fault, a Translation fault or Access flag fault. The IFSR is UNKNOWN.

In either case:

- DFSR.WnR is set to 1.
- DFSR.CM is set to 1, to indicate a fault on a cache maintenance instruction.

**TTBCR.EAE == 1**

When the value of TTBCR.EAE is 1:

- DFSR.CM is set to 1, to indicate a fault on a cache maintenance instruction.
- DFSR.STATUS indicates the cause of the fault, a Translation or Access flag fault.
- DFSR.WnR is set to 1.
- The IFSR is UNKNOWN.

**Data Abort on a Watchpoint exception**

On taking a Data Abort exception caused by a watchpoint:

- DFSR.FS is updated to indicate a debug exception.
- DFSR.\{WnR, Domain\} are UNKNOWN.
- DFAR is set to the address that generated the watchpoint

--- Note ---

- \*lr\_ABT indicates the address of the instruction that triggered the watchpoint.

---

A watchpointed address can be any byte-aligned address. The address reported in DFAR might not be the watchpointed address, and:

- For a watchpoint due to an operation other than a Data Cache maintenance instruction, can be any address between and including:
  - The lowest address accessed by the instruction that triggered the watchpoint.
  - The highest watchpointed address accessed by that instruction.

If multiple watchpoints are set in this range, there is no guarantee of which watchpoint is generated.

The address must also be within a naturally-aligned block of memory of an IMPLEMENTATION DEFINED power-of-two size, containing a watchpoint address accessed by that location.

--- Note ---

- In particular, there is no guarantee of generating the watchpoint with the lowest address in the range.
- The IMPLEMENTATION DEFINED power-of-two size must be no larger than the block size of the AArch64 DC ZVA operation.

---

- For a watchpoint due to a Data Cache operation, the address is the address passed to the instruction. This might be an address that is above the watchpointed location.
Prefetch Abort exceptions, taken to a PL1 mode

For a Prefetch Abort exception generated by an instruction fetch, the Prefetch Abort exception is taken synchronously with the instruction that the abort is reported on. This means:

• If the PE attempts to execute the instruction a Prefetch Abort exception is generated.
• If an instruction fetch is issued but the PE does not attempt to execute the prefetched instruction, no Prefetch Abort exception is generated for that instruction. For example, if the execution flow branches round a prefetched instruction, no Prefetch Abort exception is generated.

In addition, Breakpoint Instruction, Breakpoint, and Vector Catch exceptions, generate a Prefetch Abort exception, see the following for more information:

• Exception syndrome information and preferred return address for a BKPT instruction on page G2-9052.
• Exception syndrome information and preferred return address for a Breakpoint exception on page G2-9077.
• Exception syndrome information and preferred return address for a Vector Catch exception on page G2-9098.

Note

Usually, the term exception syndrome is used only for exceptions taken to Hyp mode, or to AArch64 state. The referenced sections use the term more generally, to include exception information reported in the IFSR.

On taking a Prefetch Abort exception to a PL1 mode:

• The IFSR is updated with details of the fault, including the appropriate fault code. If appropriate, the fault code indicates that the exception was generated by a debug exception.
  See the register description for more information about the returned fault information.
• For a Prefetch Abort exception generated by an instruction fetch, the IFAR is updated with the VA that caused the exception, but see Fault address reporting on synchronous External aborts on page G5-9252 for a permitted exception to this requirement.
• For a Prefetch Abort exception generated by a debug exception, the IFAR is UNKNOWN.

If the implementation includes EL3, the security state of the PE in the mode to which it takes the Prefetch Abort exception determines whether the exception updates the Secure or Non-secure IFSR and IFAR.

G5.12.3 Fault reporting in PL1 modes

The FSRs provide fault information, including an indication of the fault that occurred. The following subsections describe fault reporting in PL1 modes for each of the translation table formats:

• PL1 fault reporting with the Short-descriptor translation table format on page G5-9256.
• PL1 fault reporting with the Long-descriptor translation table format on page G5-9258.

Reserved encoding in the IFSR and DFSR encodings tables on page G5-9259 gives some additional information about the encodings for both formats.

Summary of register updates on faults taken to PL1 modes on page G5-9260 shows which registers are updated on each of the reported faults.

Reporting of External aborts taken from Non-secure state to Monitor mode describes how the fault status register format is determined for those aborts. For all other aborts, the current translation table format determines the format of the fault status registers.

Reporting of External aborts taken from Non-secure state to Monitor mode

When an External abort is taken from Non-secure state to Monitor mode:

• For a Data Abort exception, the Secure DFSR and DFAR hold information about the abort.
• For a Prefetch Abort exception, the Secure IFSR and IFAR hold information about the abort.
• The abort does not affect the contents of the Non-secure copies of the fault reporting registers.

Normally, the current translation table format determines the format of the DFSR and IFSR. However, when SCR.EA is set to 1, to route External aborts to Monitor mode, and an External abort is taken from Non-secure state, this section defines the DFSR and IFSR format.

For an External abort taken from Non-secure state to Monitor mode, the DFSR or IFSR uses the format associated with the Long-descriptor translation table format, as described in PL1 fault reporting with the Long-descriptor translation table format on page G5-9258, if any of the following applies:

• The value of the Secure TTBCR.EAE field is 1.
• The External abort is synchronous and is taken from either:
  — Hyp mode.
  — A Non-secure PL1 or EL0 mode, and the value of the Non-secure TTBCR.EAE field is 1.

Otherwise:

• For a synchronous External abort from a stage 2 translation routed to Monitor mode when the value of the Secure TTBCR.EAE field is 0 it is IMPLEMENTATION DEFINED whether:
  — The format associated with the Long-descriptor translation table format is used, as described in PL1 fault reporting with the Long-descriptor translation table format on page G5-9258.
  — The format associated with the Short-descriptor translation table format is used, as described in PL1 fault reporting with the Short-descriptor translation table format. Arm deprecates using this format. When this format is used, the value of DFSR.FS[1] or IFSR.FS[1] is UNKNOWN when reporting a synchronous External abort, or a synchronous parity or ECC error, on the stage 2 translation.

• In all other cases the DFSR or IFSR uses the format associated with the Short-descriptor translation table format, as described in PL1 fault reporting with the Short-descriptor translation table format.

PL1 fault reporting with the Short-descriptor translation table format

This subsection describes the fault reporting for a fault taken to a PL1 when address translation is using the Short-descriptor translation table format.

On taking an exception, bit[9] of the FSR is RAZ, or set to 0, if the PE is using this FSR format.

An FSR encodes the fault in a 5-bit FS field, that comprises FSR[10, 3:0]. Table G5-26 on page G5-9256 shows the encoding of that field. Summary of register updates on faults taken to PL1 modes on page G5-9260 shows:

• Whether the corresponding FAR is updated on the fault. That is:
  — For a fault reported in the IFSR, whether the IFAR holds a valid address.
  — For a fault reported in the DFSR, whether the DFAR holds a valid address.
• For faults that update DFSR, whether DFSR.Domain is valid

When reading Table G5-26 on page G5-9256:

• FS values not shown in the table are reserved.
• FS values shown as DFSR only are reserved for the IFSR.

| FS   | Source                              | Notes                      |
|------|-------------------------------------|----------------------------|---|
| 00001| Alignment fault                     | DFSR only. Fault on initial lookup |
| 00100| Fault on instruction cache maintenance | DFSR only                  |
| 01100| Synchronous External abort on translation table walk\(^a,b\) | Level 1                      |
| 01110|                                      | Level 2                      |

\(^a\) Synchronous
\(^b\) External abort on translation table walk

Summary of register updates on faults taken to PL1 modes on page G5-9260 shows:
The level associated with MMU faults on a Short-descriptor translation table lookup

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because a stage of address translation is disabled, or because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a level 1 fault.
- For an Access flag fault, Permission fault, or Domain fault, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Supersection, Section, or Page descriptor.

Also see Synchronous External abort errors from address translation caching structures on page G5-9250.

The Domain field in the DFSR

The DFSR includes a Domain field. This is inherited from previous versions of the VMSA. The IFSR does not include a Domain field. Summary of register updates on faults taken to PL1 modes on page G5-9260 describes when DFSR Domain is valid.

Table G5-26 FSR encodings when using the Short-description translation table format (continued)

<table>
<thead>
<tr>
<th>FS</th>
<th>Source</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>11100</td>
<td>Synchronous parity or ECC error on translation table walkᵃ ᵇ</td>
<td>Level 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Level 2</td>
</tr>
<tr>
<td>11110</td>
<td>Translation faultᵃ</td>
<td>Level 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Level 2</td>
</tr>
<tr>
<td>00101</td>
<td>Translation faultᵃ</td>
<td>MMU fault</td>
</tr>
<tr>
<td>00111</td>
<td>Access flag faultᵃ</td>
<td>Level 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Level 2</td>
</tr>
<tr>
<td>00111</td>
<td>Access flag faultᵃ</td>
<td>MMU fault</td>
</tr>
<tr>
<td>00110</td>
<td>Domain faultᵃ</td>
<td>Level 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Level 2</td>
</tr>
<tr>
<td>01001</td>
<td>Domain faultᵃ</td>
<td>MMU fault</td>
</tr>
<tr>
<td>01011</td>
<td>Permission faultᵃ</td>
<td>Level 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Level 2</td>
</tr>
<tr>
<td>01101</td>
<td>Permission faultᵃ</td>
<td>MMU fault</td>
</tr>
<tr>
<td>01111</td>
<td>Permission faultᵃ</td>
<td>MMU fault</td>
</tr>
<tr>
<td>00010</td>
<td>Debug exception</td>
<td>See Chapter G2 AArch32 Self-hosted Debug</td>
</tr>
<tr>
<td>01000</td>
<td>Synchronous External abort</td>
<td>-</td>
</tr>
<tr>
<td>10000</td>
<td>TLB conflict abort</td>
<td>See TLB conflict aborts on page G5-9218</td>
</tr>
<tr>
<td>10100</td>
<td>IMPLEMENTATION DEFINED</td>
<td>Lockdown</td>
</tr>
<tr>
<td>10101</td>
<td>IMPLEMENTATION DEFINED</td>
<td>Unsupported Exclusive access</td>
</tr>
<tr>
<td>11001</td>
<td>Synchronous parity or ECC error on memory access</td>
<td>-</td>
</tr>
<tr>
<td>11010</td>
<td>SError interruptᵈ</td>
<td>DFSR only</td>
</tr>
<tr>
<td>11000</td>
<td>SError interruptᵈ from a parity or ECC error on memory access</td>
<td>DFSR only</td>
</tr>
</tbody>
</table>

a. See The level associated with MMU faults on a Short-descriptor translation table lookup.

b. FS[1] is UNKNOWN if the reported error is from a stage 2 translation.

c. Previously, this encoding was a deprecated encoding for Alignment fault. The extensive changes in the memory model in VMSAv8-32 mean there should be no possibility of confusing the new use of this encoding with its previous use

d. Including asynchronous External abort on a data access, a translation table walk, or an instruction fetch.
Arm deprecates any use of the Domain field in the DFSR. The Long-descriptor translation table format does not support a Domain field, and future versions of the Arm architecture might not support a Domain field in the Short-descriptor translation table format. Arm strongly recommends that new software does not use this field.

For both Data Abort exceptions and Prefetch Abort exceptions, software can find the domain information by performing a translation table read for the faulting address and extracting the Domain field from the translation table entry.

**PL1 fault reporting with the Long-descriptor translation table format**

This subsection describes the fault reporting for a fault taken to a PL1 mode when address translation is using the Long-descriptor translation table format.

When the PE takes an exception, bit[9] of the FSR is set to 1 if the PE is using this FSR format.

The FSRs encode the fault in a 6-bit STATUS field, that comprises FSR[5:0]. Table G5-27 on page G5-9258 shows the encoding of that field. In addition:

- For a fault taken to a PL1 mode, *Summary of register updates on faults taken to PL1 modes on page G5-9260* shows whether the corresponding FAR is updated on the fault. That is:
  - For a fault reported in the IFSR, whether the IFAR holds a valid address.
  - For a fault reported in the DFSR, whether the DFAR holds a valid address.

- For a fault taken to the Hyp mode, *Summary of register updates on exceptions taken to Hyp mode on page G5-9268* shows what registers are updated on the fault.

<table>
<thead>
<tr>
<th>STATUS</th>
<th>Source Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000LL</td>
<td>Address size fault. LL bits indicate level.</td>
<td>MMU fault</td>
</tr>
<tr>
<td>0001LL</td>
<td>Translation fault. LL bits indicate level.</td>
<td>MMU fault</td>
</tr>
<tr>
<td>0010LL</td>
<td>Access flag fault. LL bits indicate level.</td>
<td>MMU fault</td>
</tr>
<tr>
<td>0011LL</td>
<td>Permission fault. LL bits indicate level.</td>
<td>MMU fault</td>
</tr>
<tr>
<td>010000</td>
<td>Synchronous External abort.</td>
<td>-</td>
</tr>
<tr>
<td>011000</td>
<td>Synchronous parity or ECC error on memory access.</td>
<td>-</td>
</tr>
<tr>
<td>010001</td>
<td>SError interrupt.</td>
<td>DFSR only</td>
</tr>
<tr>
<td>011001</td>
<td>SError interrupt from a parity or ECC error on memory access.</td>
<td>DFSR only</td>
</tr>
<tr>
<td>0101LL</td>
<td>Synchronous External abort on translation table walk. LL bits indicate level.</td>
<td>-</td>
</tr>
<tr>
<td>0111LL</td>
<td>Synchronous parity or ECC error on memory access on translation table walk. LL bits indicate level.</td>
<td>-</td>
</tr>
<tr>
<td>100001</td>
<td>Alignment fault.</td>
<td>Fault on initial lookup</td>
</tr>
<tr>
<td>100010</td>
<td>Debug exception.</td>
<td>See Chapter G2 AArch32 Self-hosted Debug</td>
</tr>
<tr>
<td>110000</td>
<td>TLB conflict abort.</td>
<td>See <em>TLB conflict aborts on page G5-9218</em></td>
</tr>
</tbody>
</table>
The level associated with MMU faults on a Long-descriptor translation table lookup

For MMU faults, Table G5-28 on page G5-9259 shows how the LL bits in the xFSR.STATUS field encode the lookup level associated with the fault.

Table G5-27 FSR encodings when using the Long-descriptor translation table format (continued)

<table>
<thead>
<tr>
<th>STATUSa</th>
<th>Source</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>110100</td>
<td>IMPLEMENTATION DEFINED.</td>
<td>Lockdown, DFSR only</td>
</tr>
<tr>
<td>110101</td>
<td>IMPLEMENTATION DEFINED.</td>
<td>Unsupported Exclusive access</td>
</tr>
<tr>
<td>1111LL</td>
<td>Domain fault.</td>
<td>MMU fault. 64-bit PAR only, level 1 or level 2 only. Never used in DFSR, IFSR, or HSRd</td>
</tr>
<tr>
<td></td>
<td>LL bits indicate levelb.</td>
<td></td>
</tr>
</tbody>
</table>

a. STATUS values not shown in this table are reserved. STATUS values not supported in the IFSR or DFSR are reserved for the register or registers in which they are not supported.
b. See The level associated with MMU faults on a Long-descriptor translation table lookup.
c. Including asynchronous External abort on a data access, a translation table walk, or an instruction fetch.
d. A Domain fault can be reported using the Long-descriptor STATUS encodings only as a result of a fault on an address translation instruction. For more information, see MMU fault on an address translation instruction on page G5-9275.

Reserved encoding in the IFSR and DFSR encodings tables

With both the Short-descriptor and the Long-descriptor FSR format, the fault encodings reserve a single encoding for Cache and TLB lockdown faults. The details of these faults and any associated subsidiary registers are IMPLEMENTATION DEFINED.

<table>
<thead>
<tr>
<th>LL bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Address size fault</td>
</tr>
<tr>
<td></td>
<td>Address size fault in TTBR0 or TTBR1.</td>
</tr>
<tr>
<td></td>
<td>All other faults</td>
</tr>
<tr>
<td></td>
<td>Reserved.</td>
</tr>
<tr>
<td>01</td>
<td>Level 1.</td>
</tr>
<tr>
<td>10</td>
<td>Level 2.</td>
</tr>
<tr>
<td>11</td>
<td>Level 3. When xFSR.STATUS indicates a Domain fault, this value is reserved.</td>
</tr>
</tbody>
</table>

The lookup level associated with a fault is:

- For a fault generated on a translation table walk, the lookup level of the walk being performed.
- For a Translation fault, the lookup level of the translation table that gave the fault. If a fault occurs because a stage of address translation is disabled, or because the input address is outside the range specified by the appropriate base address register or registers, the fault is reported as a level 1 fault.
- For an Access flag fault, the lookup level of the translation table that gave the fault.
- For a Permission fault, including a Permission fault caused by hierarchical permissions, the lookup level of the final level of translation table accessed for the translation. That is, the lookup level of the translation table that returned a Block or Page descriptor.

Also see Synchronous External abort errors from address translation caching structures on page G5-9250.
### G5.12.4 Summary of register updates on faults taken to PL1 modes

For faults that generate exceptions that are taken to a PL1 mode, Table G5-29 on page G5-9260 shows the registers affected by each fault. In this table:
- **Yes** indicates that the register is updated.
- **UNK** indicates that the fault makes the register value UNKNOWN.
- A null entry, -, indicates that the fault does not affect the register.

For faults that update the DFSR using the Short-descriptor format FSR encodings, Table G5-30 on page G5-9261 shows whether DFSR.Domain is valid.

#### Table G5-29 Effect of a fault taken to a PL1 mode on the reporting registers

<table>
<thead>
<tr>
<th>Fault</th>
<th>IFSR</th>
<th>IFAR</th>
<th>DFSR</th>
<th>DFAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMU fault, always synchronous</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Synchronous External abort on translation table walk</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Synchronous parity or ECC error on translation table walk</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Synchronous External abort</td>
<td>Yes</td>
<td>IMP DEF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Synchronous parity or ECC error on memory access</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TLB conflict abort</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

#### Fault reported as Data Abort exception:

<table>
<thead>
<tr>
<th>Fault</th>
<th>IFSR</th>
<th>IFAR</th>
<th>DFSR</th>
<th>DFAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment fault, always synchronous</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MMU fault, always synchronous</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Fault on instruction cache maintenance, when using Long-descriptor translation table format&lt;sup&gt;b&lt;/sup&gt;</td>
<td>UNK</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Fault on instruction cache maintenance, when using Short descriptor translation table format&lt;sup&gt;c&lt;/sup&gt;</td>
<td>either</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>or</td>
<td>UNK</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Synchronous External abort on translation table walk</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Synchronous parity or ECC error on translation table walk</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Synchronous External abort</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>IMP DEF&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>Synchronous parity or ECC error on memory access</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SError interrupt</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>UNK</td>
</tr>
<tr>
<td>SError interrupt from a parity or ECC error on memory access</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>UNK</td>
</tr>
<tr>
<td>TLB conflict abort</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

#### Debug exceptions:

<table>
<thead>
<tr>
<th>Fault</th>
<th>IFSR</th>
<th>IFAR</th>
<th>DFSR</th>
<th>DFAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoint, Breakpoint Instruction, or Vector Catch&lt;sup&gt;d&lt;/sup&gt;</td>
<td>Yes</td>
<td>UNK</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Watchpoint&lt;sup&gt;e&lt;/sup&gt;</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

---

<sup>a</sup> IMPLEMENTATION DEFINED. The IFSR.FnV or DFSR.FnV bit indicates whether the register holds a valid address. See Fault address reporting on synchronous External aborts on page G5-9252.
b. When using the Long-descriptor translation table format, there is not a specific fault code for a fault on an instruction cache maintenance instruction. For more information, see Data Abort on an instruction cache or branch predictor maintenance instruction by VA on page G5-9254.

c. The two lines of this entry show the alternative ways of reporting the fault when using the Short-descriptor translation table format. It is IMPLEMENTATION DEFINED which methods is used, see Data Abort on an instruction cache or branch predictor maintenance instruction by VA on page G5-9254.

d. Generates a Prefetch Abort exception.

e. Generates a Data Abort exception.

For those faults for which Table G5-29 on page G5-9260 shows that the DFSR is updated, if the fault is reported using the Short-descriptor FSR encodings, Table G5-30 on page G5-9261 shows whether DFSR.Domain is valid. In this table, UNK indicates that the fault makes DFSR.Domain UNKNOWN.

Table G5-30 Validity of Domain field on faults that update the DFSR when using the Short-descriptor encodings

<table>
<thead>
<tr>
<th>DFSR.FS</th>
<th>Source</th>
<th>DFSR.Domain</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001</td>
<td>Alignment fault</td>
<td>UNK</td>
<td>-</td>
</tr>
<tr>
<td>00100</td>
<td>Fault on instruction cache maintenance instruction</td>
<td>UNK</td>
<td>-</td>
</tr>
<tr>
<td>01100</td>
<td>Synchronous External abort on translation table walk</td>
<td>Level 1</td>
<td>UNK</td>
</tr>
<tr>
<td>01110</td>
<td>Synchronous External abort on translation table walk</td>
<td>Level 2</td>
<td>Valid</td>
</tr>
<tr>
<td>11100</td>
<td>Synchronous parity or ECC error on translation table walk</td>
<td>Level 1</td>
<td>UNK</td>
</tr>
<tr>
<td>11110</td>
<td>Synchronous parity or ECC error on translation table walk</td>
<td>Level 2</td>
<td>Valid</td>
</tr>
<tr>
<td>00101</td>
<td>Translation fault</td>
<td>Level 1</td>
<td>UNK</td>
</tr>
<tr>
<td>00111</td>
<td>Translation fault</td>
<td>Level 2</td>
<td>Valid</td>
</tr>
<tr>
<td>00011a</td>
<td>Access flag fault</td>
<td>Level 1</td>
<td>UNK</td>
</tr>
<tr>
<td></td>
<td>Access flag fault</td>
<td>Level 2</td>
<td>Valid</td>
</tr>
<tr>
<td>01001</td>
<td>Domain fault</td>
<td>Level 1</td>
<td>Valid</td>
</tr>
<tr>
<td>01011</td>
<td>Domain fault</td>
<td>Level 2</td>
<td>Valid</td>
</tr>
<tr>
<td>01101</td>
<td>Permission fault</td>
<td>Level 1</td>
<td>UNK</td>
</tr>
<tr>
<td>01111</td>
<td>Permission fault</td>
<td>Level 2</td>
<td>UNK</td>
</tr>
<tr>
<td>01000</td>
<td>Synchronous External abort</td>
<td>UNK</td>
<td>-</td>
</tr>
<tr>
<td>10000</td>
<td>TLB conflict abort</td>
<td>UNK</td>
<td>-</td>
</tr>
<tr>
<td>11001</td>
<td>Synchronous parity or ECC error on memory access</td>
<td>UNK</td>
<td>-</td>
</tr>
<tr>
<td>10110</td>
<td>SError interrupt(^b)</td>
<td>UNK</td>
<td>-</td>
</tr>
<tr>
<td>11000</td>
<td>SError interrupt(^b) from a parity or ECC error on memory access</td>
<td>UNK</td>
<td>-</td>
</tr>
<tr>
<td>00010</td>
<td>Watchpoint</td>
<td>UNK</td>
<td></td>
</tr>
</tbody>
</table>

a. Previously, this encoding was a deprecated encoding for Alignment fault. The extensive changes in the memory model in VMSAv8-32 mean there should be no possibility of confusing the new use of this encoding with its previous use.

b. Including asynchronous External abort on a data access, a translation table walk, or an instruction fetch.

G5.12.5 Reporting exceptions taken to Hyp mode

Hyp mode is the Non-secure EL2 mode. It is entered by taking an exception to Hyp mode.
Note

Software executing in Monitor mode, or at EL3 when EL3 is using AArch64, can perform an exception return to Hyp mode. This means Hyp mode can be entered either by taking an exception, or by a permitted exception return.

When EL2 is using AArch32, the following exceptions are taken to Hyp mode:

- SError interrupt exceptions, IRQ exceptions, and FIQ exceptions, from Non-secure PL1 and EL0 modes, if not routed to Secure Monitor mode, and if configured by the AMO, FMO or IMO bits. For more information, see Asynchronous exception routing controls on page G1-8957.
- When HCR.TGE is set to 1, all exceptions that would be routed to Non-secure PL1 modes. For more information, see Routing exceptions from Non-secure EL0 to EL2 on page G1-8944.
- When HDCR.TDE is set to 1, any debug exception that would otherwise be taken to a Non-secure PL1 mode, see Routing debug exceptions to EL2 using AArch32 on page G1-8946.
- The privilege rules for taking exceptions mean that any exception taken from Hyp mode, if not routed to EL3, must be taken to Hyp mode.
- An abort that Routing of aborts taken to AArch32 state on page G1-8948 identifies as taken to Hyp mode.
- Hypervisor Call exceptions, and Hyp Trap exceptions, are always taken to Hyp mode. These exceptions are supported only as part of EL2.

Synchronous exceptions taken to Hyp mode provide syndrome information in the HSR.

On an abort exception taken to Hyp mode, the syndrome information in the HSR includes the Fault status code otherwise provided by the fault status register, and extends the fault reporting compared to that available for an exception taken to a PL1 mode.

In addition, for a Debug exception taken to Hyp mode, DBGDSRint.MOE or DBGDSRext.MOE shows what caused the Debug exception. This field is valid regardless of whether the Debug exception was taken from Hyp mode or from another Non-secure mode.

For more information, see the following subsections:

- Registers used for reporting exceptions taken to Hyp mode.
- Memory fault reporting in Hyp mode on page G5-9263.
- Use of the HSR on page G5-9265

**Registers used for reporting exceptions taken to Hyp mode**

The following registers are used for reporting exceptions taken to Hyp mode:

- The HSR holds syndrome information for the exception.
- The HDFAR holds the VA associated with a Data Abort exception.
- The HIFAR holds the VA associated with a Prefetch Abort exception.
- The HPFAR holds bits[39:12] of the IPA associated with some aborts on stage 2 address translations.

In addition, if implemented, the optional HADFSR and HAIIFSR can provide additional fault information, see Hyp Auxiliary Fault Syndrome Registers.

**Hyp Auxiliary Fault Syndrome Registers**

EL2 also defines encodings for the following Hyp Auxiliary Fault Syndrome Registers:

- The Hyp Auxiliary Data Fault Syndrome Register, HADFSR.
- The Hyp Auxiliary Instruction Fault Syndrome Register, HAIIFSR.
An implementation can use these registers to return additional fault status information for aborts taken to Hyp mode. They are the Hyp mode equivalents of the registers described in Auxiliary Fault Status Registers on page G5-9253. An example use of these registers is to return more information for diagnosing parity or ECC errors.

The architectural requirements for the HADFSR and HAIFSR are:

- The position of these registers is architecturally-defined, but the content and use of the registers is IMPLEMENTATION DEFINED.
- An implementation with no requirement for additional fault reporting can implement these registers as RES0, but the architecture does not require it to do so.

**Memory fault reporting in Hyp mode**

Prefetch Abort and Data Abort exceptions taken to Hyp mode report memory faults. For these aborts, the HSR contains the following fault status information:

- The HSR.EC field indicates the type of abort, as Table G5-31 on page G5-9263 shows.
- The HSR.ISS field holds more information about the abort. In particular:
  - Bits[5:0] of this field hold the STATUS field for the abort, using the encodings defined in **PL1 fault reporting with the Long-descriptor translation table format on page G5-9258**.
  - Other subfields of the ISS give more information about the exception, equivalent to the information returned in the FSR for a memory fault reported at PL1.

See the descriptions of the ISS fields for the memory faults, referenced from the Syndrome description column of Table G5-31 on page G5-9263, for information about the returned fault information.

<table>
<thead>
<tr>
<th>HSR.EC</th>
<th>Abort</th>
<th>Syndrome description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>Prefetch Abort taken from Non-secure PL1 or EL0 mode</td>
<td>ISS encoding for exception from a Prefetch Abort on page G8-9536</td>
</tr>
<tr>
<td>0x21</td>
<td>Prefetch Abort taken from Hyp mode</td>
<td></td>
</tr>
<tr>
<td>0x24</td>
<td>Data Abort taken from Non-secure PL1 or EL0 mode</td>
<td>ISS encoding for exception from a Data Abort on page G8-9538</td>
</tr>
<tr>
<td>0x25</td>
<td>Data Abort taken from Hyp mode</td>
<td></td>
</tr>
</tbody>
</table>

For more information, see Use of the HSR on page G5-9265.

A Prefetch Abort exception is taken synchronously with the instruction that the abort is reported on. This means:

- If the PE attempts to execute the instruction a Prefetch Abort exception is generated.
- If an instruction fetch is issued but the PE does not attempt to execute the prefetched instruction, no Prefetch Abort exception is generated for that instruction. For example, if the execution flow branches round a prefetched instruction that would abort if the PE attempted to execute it, no Prefetch Abort exception is generated.

**Register updates on exception reporting in Hyp mode**

The use of the HSR, and of the other registers listed in Registers used for reporting exceptions taken to Hyp mode on page G5-9262, depends on the cause of the Abort. In reporting these faults, in general:

- If the fault generates a synchronous Data Abort exception, the HDFAR holds the associated VA, but see Fault address reporting on synchronous External aborts on page G5-9252 for a permitted exception to this requirement.
- If the fault generates a Prefetch Abort exception, the HIFAR holds the associated VA, but see Fault address reporting on synchronous External aborts on page G5-9252 for a permitted exception to this requirement.
In the following cases, the HPFAR holds the faulting IPA:

— A Translation or Access flag fault on a stage 2 translation.
— A Translation, Access flag, or Permission fault on the stage 2 translation of an address accessed in a stage 1 translation table walk.
— A stage 2 Address size fault.

In all other cases, the HPFAR is UNKNOWN.

• On a Data Abort exception that is taken to Hyp mode, the HIFAR is UNKNOWN.

• On a Prefetch Abort exception that is taken to Hyp mode, the HDFAR is UNKNOWN.

In addition, the reporting of particular aborts is as follows:

Abort on the stage 1 translation for a memory access from Hyp mode

The HDFAR or HIFAR holds the VA that caused the fault. The STATUS subfield of HSR.ISS indicates the type of fault, Translation, Address size, Access flag, or Permission. The HPFAR is UNKNOWN.

Abort on the stage 2 translation for a memory access from a Non-secure PL1 or EL0 mode

This includes aborts on the stage 2 translation of a memory access made as part of a translation table walk for a stage 1 translation. The HDFAR or HIFAR holds the VA that caused the fault. The STATUS subfield of HSR.ISS indicates the type of fault, Translation, Address size, Access flag, or Permission.

For any Access flag fault or Translation fault, and also for any Permission fault on the stage 2 translation of a memory access made as part of a translation table walk for a stage 1 translation, the HPFAR holds the IPA that caused the fault. Otherwise, the HPFAR is UNKNOWN.

Abort caused by a synchronous External abort, or synchronous parity or ECC error, and taken to Hyp mode

The HDFAR or HIFAR holds the VA that caused the fault, but see Fault address reporting on synchronous External aborts on page G5-9252 for a permitted exception to this requirement. The HPFAR is UNKNOWN.

Data Abort caused by a Watchpoint exception and routed to Hyp mode because HDCR.TDE is set to 1

When HDCR.TDE is set to 1, a Watchpoint exception generated in a Non-secure PL1 or EL0 mode, that would otherwise generate a Data Abort exception, is routed to Hyp mode and generates a Hyp Trap exception.

HDFAR is set to the address that generated the watchpoint.

—— Note ———

ELR_hyp indicates the address of the instruction that triggered the watchpoint.

A watchpointed address can be any byte-aligned address. The address reported in HDFAR might not be the watchpointed address, and, for a watchpoint due to an operation other than a Data Cache maintenance instruction, can be any address between and including:

• The lowest address accessed by the instruction that triggered the watchpoint.
• The highest watchpointed address accessed by that instruction.

If multiple watchpoints are set in this range, there is no guarantee of which watchpoint is generated.

—— Note ———

In particular, there is no guarantee of generating the watchpoint with the lowest address in the range.

———

The address must also be within a naturally-aligned block of memory of an IMPLEMENTATION DEFINED power-of-two size, containing a watchpoint address accessed by that location.
The IMPLEMENTATION DEFINED power-of-two size must be no larger than the block size of the AArch64 DC ZVA operation.

See also Watchpoint exceptions on page G2-9079.

In all cases, HPFAR is UNKNOWN.

Prefetch Abort caused by a Breakpoint Instruction exception and taken to Hyp mode
This abort is generated if a $BKPT$ instruction is executed in Hyp mode. The abort leaves the HIFAR and HPFAR UNKNOWN.

See also Breakpoint Instruction exceptions on page G2-9051.

Prefetch Abort caused by a Breakpoint Instruction, Breakpoint, or Vector Catch exception, and routed to Hyp mode because HDCR.TDE is set to 1
When HDCR.TDE is set to 1, a debug exception, generated in a Non-secure PL1 or EL0 mode, that would otherwise generate a Prefetch Abort exception, is routed to Hyp mode and generates a Hyp Trap exception.

The abort leaves the HIFAR and HPFAR UNKNOWN. This is identical to the reporting of a Prefetch Abort exception caused by a Debug exception on a $BKPT$ instruction that is executed in Hyp mode.

Use of the HSR
The HSR holds syndrome information for any synchronous exception taken to Hyp mode. Compared with the reporting of exceptions taken to PL1 modes, the HSR:
• Always provides details of the fault. The DFSR and IFSR are not used.
• Provides more extensive information, for a wider range of exceptions.

IRQ and FIQ exceptions taken to Hyp mode do not report any syndrome information in the HSR.

This section summarizes the general form of the HSR register, to show how it encodes exception syndrome information, see the register description for more information. The register comprises:
• A 6-bit Exception class field, EC, that indicates the cause of the exception.
• An instruction length bit, IL. When an exception is caused by trapping an instruction to Hyp mode, this bit indicates the length of the trapped instruction, as follows:
  0 16-bit instruction trapped.
  1 32-bit instruction trapped.
  In other cases the IL field is not valid and is RES1.
• An instruction specific syndrome field, ISS. Architecturally, this field could be defined independently for each defined Exception class (EC), but in practice several ISS formats are common to more than one EC.
The format of the HSR depends on the value of the EC field, as follows:

\[ 0b000000 \leq EC \leq 0b001110 \]

The ISS part of the returned value includes the CV and COND fields described in *Encoding of ISS[24:20] when \( 0b000000 \leq EC \leq 0b001110 \).* Figure G5-17 shows the HSR format in this case.

![Figure G5-17 HSR format when the ISS includes CV and COND fields](image1)

EC == \( 0b000000 \) or EC \( 0b001110 \) There are no generic fields within the ISS. Figure G5-18 shows the HSR format in this case.

![Figure G5-18 HSR format when the ISS does not include a COND field](image2)

*Encoding of ISS[24:20] when \( 0b000000 \leq EC \leq 0b001110 \)*

For EC values that are nonzero and less than or equal to \( 0b001100 \), ISS[24:20] provides the Condition code field for the trapped instruction, together with a valid flag for this field. The encoding of this part of the ISS field is:

- **CV, ISS[24]** Condition code valid. Possible values of this bit are:
  - 0: The COND field is not valid.
  - 1: The COND field is valid.

- **COND, ISS[23:20]**
  - The Condition code for the trapped instruction. This field is valid only when CV is set to 1.
  - If CV is set to 0, this field is RES0.

The full descriptions of the HSR.ISS formats give more information about the CV field.

---

**Note**

In some circumstances, it is IMPLEMENTATION DEFINED whether a conditional instruction that fails its Condition code check generates an Undefined Instruction exception, see *Conditional execution of undefined instructions* on page G1-8966.
### HSR exception classes

Table G5-32 on page G5-9267 shows the encoding of the HSR exception class field, EC. Values of EC not shown in the table are reserved. For each EC value, the table references a subsection of the description of the HSR that describes the associated ISS format and gives information about the cause of the exception, for example the configuration required to enable the associated trap.

#### Table G5-32 HSR.EC field encoding

<table>
<thead>
<tr>
<th>EC</th>
<th>Exception class</th>
<th>ISS description, or notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>Unknown reason</td>
<td><strong>ISS encoding for exceptions with an unknown reason on page G8-9525.</strong></td>
</tr>
<tr>
<td>0b00 0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b00</td>
<td>Trapped WFI or WFE instruction</td>
<td><strong>ISS encoding for exception from a WFI or WFE instruction on page G8-9526.</strong></td>
</tr>
<tr>
<td>0b00 0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b00</td>
<td>Trapped MCR or MRC access with (coproc == 0b1111)</td>
<td><strong>ISS encoding for exception from an MCR or MRC access on page G8-9527.</strong></td>
</tr>
<tr>
<td>0b00 0011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b00</td>
<td>Trapped MCRR or MRRC access with (coproc == 0b1111)</td>
<td><strong>ISS encoding for exception from an MCRR or MRRC access on page G8-9529.</strong></td>
</tr>
<tr>
<td>0b00 0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b00</td>
<td>Trapped MCR or MRC access with (coproc == 0b1110)</td>
<td><strong>ISS encoding for exception from an MCR or MRC access on page G8-9527.</strong></td>
</tr>
<tr>
<td>0b00 0101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b00</td>
<td>Trapped LDC or STC access</td>
<td><strong>ISS encoding for exception from an LDC or STC instruction on page G8-9531.</strong></td>
</tr>
<tr>
<td>0b00 0110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b00</td>
<td>Advanced SIMD or floating-point functionality trapped by a HCPTR, {TASE, TCP10} control</td>
<td><strong>ISS encoding for exception from an access to SIMD or floating-point functionality, resulting from HCPTR on page G8-9533.</strong></td>
</tr>
<tr>
<td>0b00 0111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b00</td>
<td>Trapped VMRS access, from ID group traps, that is not reported using EC 0b000111</td>
<td><strong>ISS encoding for exception from an MCR or MRC access on page G8-9527.</strong> This trap is not taken if the HCPTR settings trap the access.</td>
</tr>
<tr>
<td>0b00 1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b00</td>
<td>Trapped MCRR access with (coproc == 0b1110)</td>
<td><strong>ISS encoding for exception from an MCRR or MRRC access on page G8-9529.</strong></td>
</tr>
<tr>
<td>0b00 1100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b00</td>
<td>Illegal exception return to AArch32 state</td>
<td><strong>ISS encoding for exception from an Illegal state or PC alignment fault on page G8-9538.</strong></td>
</tr>
<tr>
<td>0b00 1110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b01</td>
<td>Exception on SVC execution in AArch32 state routed to EL2</td>
<td><strong>ISS encoding for exception from HVC or SVC instruction execution on page G8-9534.</strong></td>
</tr>
<tr>
<td>0b01 0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b01</td>
<td>HVC instruction execution in AArch32 state, when HVC is not disabled</td>
<td><strong>ISS encoding for exception from SMC instruction execution on page G8-9535.</strong></td>
</tr>
<tr>
<td>0b01 0010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b01</td>
<td>Trapped execution of SMC instruction in AArch32 state</td>
<td><strong>ISS encoding for exception from a Prefetch Abort on page G8-9536.</strong></td>
</tr>
<tr>
<td>0b01 0011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>Prefetch Abort from a lower Exception level</td>
<td><strong>ISS encoding for exception from a Prefetch Abort on page G8-9536.</strong></td>
</tr>
<tr>
<td>0b10 0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>Prefetch Abort taken without a change in Exception level</td>
<td><strong>ISS encoding for exception from a Prefetch Abort on page G8-9536.</strong></td>
</tr>
<tr>
<td>0b10 0001</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table G5-32 HSR.EC field encoding (continued)

<table>
<thead>
<tr>
<th>EC</th>
<th>Exception class</th>
<th>ISS description, or notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>PC alignment exception.</td>
<td>*ISS encoding for exception from an Illegal state or PC alignment fault on page G8-9538.</td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>Data Abort from a lower Exception level</td>
<td>*ISS encoding for exception from a Data Abort on page G8-9538.</td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>Data Abort taken without a change in Exception level</td>
<td></td>
</tr>
</tbody>
</table>

All EC encodings not shown in Table G5-31 on page G5-9263 are reserved by Arm.

**G5.12.6 Summary of register updates on exceptions taken to Hyp mode**

For memory system faults that generate exceptions that are taken to Hyp mode, Table G5-33 on page G5-9268 shows the registers affected by each fault. In this table:

- Yes indicates that the register is updated.
- UNK indicates that the fault makes the register value UNKNOWN.
- A null entry, -, indicates that the fault does not affect the register.

Note

For a list of the MMU faults see *Types of MMU faults on page G5-9239.*

Table G5-33 Effect of an exception taken to Hyp mode on the reporting registers

<table>
<thead>
<tr>
<th>Fault</th>
<th>HSR</th>
<th>HIFAR</th>
<th>HDFAR</th>
<th>HPFAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faults reported as Prefetch Abort exceptions:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMU fault* at stage 1.</td>
<td>Yes</td>
<td>Yes</td>
<td>UNK</td>
<td>UNK</td>
</tr>
<tr>
<td>Translation or Access flag MMU fault* at stage 2.</td>
<td>Yes</td>
<td>Yes</td>
<td>UNK</td>
<td>Yes</td>
</tr>
<tr>
<td>Otherb MMU fault* at stage 2.</td>
<td>Yes</td>
<td>Yes</td>
<td>UNK</td>
<td>UNK</td>
</tr>
<tr>
<td>Stage 2 MMU fault* on a stage 1 translation.</td>
<td>Yes</td>
<td>Yes</td>
<td>UNK</td>
<td>Yes</td>
</tr>
<tr>
<td>Synchronous External abort on translation table walk.</td>
<td>Yes</td>
<td>Yes</td>
<td>UNK</td>
<td>UNK</td>
</tr>
<tr>
<td>Synchronous parity or ECC error on translation table walk.</td>
<td>Yes</td>
<td>Yes</td>
<td>UNK</td>
<td>UNK</td>
</tr>
<tr>
<td>Synchronous External abort.</td>
<td>Yes</td>
<td>IMP DEFIN</td>
<td>UNK</td>
<td>UNK</td>
</tr>
<tr>
<td>Synchronous parity or ECC error on memory access.</td>
<td>Yes</td>
<td>Yes</td>
<td>UNK</td>
<td>UNK</td>
</tr>
</tbody>
</table>

Fault reported as Data Abort exception:

| MMU fault* at stage 1.                             | Yes | UNK   | Yes   | UNK   |
| Translation or Access flag MMU fault* at stage 2.  | Yes | UNK   | Yes   | Yes   |
| Otherb MMU fault* at stage 2.                      | Yes | UNK   | Yes   | UNK   |
| Stage 2 MMU fault* on a stage 1 translation.       | Yes | UNK   | Yes   | Yes   |
| Synchronous External abort on translation table walk. | Yes | UNK   | Yes   | UNK   |
Note

Unlike Table G5-29 on page G5-9260, the Hyp mode fault reporting table does not include an entry for a fault on an instruction cache maintenance instruction. That is because, when the fault is taken to Hyp mode, the reporting indicates the cause of the fault, for example a Translation fault, and ISS.CM is set to 1 to indicate that the fault was on a cache maintenance instruction, see ISS encoding for exception from a Data Abort on page G8-9538.

Classification of MMU faults taken to Hyp mode

This subsection gives more information about the MMU faults shown in Table G5-33 on page G5-9268.

Note

All MMU faults are synchronous.

The table uses the following descriptions for MMU faults taken to Hyp mode:

**MMU fault at stage 1**

This is an MMU fault generated on a stage 1 translation performed in the Non-secure EL2 translation regime.

**MMU fault at stage 2**

This is an MMU fault generated on a stage 2 translation performed in the Non-secure PL1&0 translation regime.

As the table shows, for the faults in this group:

- Translation and Access flag faults update the HPFAR
- Permission faults leave the HPFAR UNKNOWN.

---

Table G5-33 Effect of an exception taken to Hyp mode on the reporting registers (continued)

<table>
<thead>
<tr>
<th>Fault</th>
<th>HSR</th>
<th>HIFAR</th>
<th>HDFAR</th>
<th>HPFAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous parity or ECC error on translation table walk.</td>
<td>Yes</td>
<td>UNK</td>
<td>Yes</td>
<td>UNK</td>
</tr>
<tr>
<td>Synchronous External abort.</td>
<td>Yes</td>
<td>UNK</td>
<td>IMP DEF</td>
<td>UNK</td>
</tr>
<tr>
<td>Synchronous parity or ECC error on memory access.</td>
<td>Yes</td>
<td>UNK</td>
<td>Yes</td>
<td>UNK</td>
</tr>
<tr>
<td>SError interrupt</td>
<td>Yes</td>
<td>UNK</td>
<td>UNK</td>
<td>UNK</td>
</tr>
<tr>
<td>SError interrupt from a parity or ECC error on memory access.</td>
<td>Yes</td>
<td>UNK</td>
<td>UNK</td>
<td>UNK</td>
</tr>
</tbody>
</table>

Debug exception:

Breakpoint Instruction\(^d\), generates a Prefetch Abort exception. | Yes | UNK | - | UNK |

Debug exception routed to Hyp mode because HDCR.TDE is set to 1. Generates a Hyp Trap exception.

Breakpoint Instruction or Vector Catch | Yes | UNK | - | UNK |

Watchpoint | Yes | - | - | UNK |

---

a. For more information, see Classification of MMU faults taken to Hyp mode
b. MMU fault other than a Translation fault or an Access flag fault.
c. IMPLEMENTATION DEFINED. The FnV bit in the HSR.ISS field indicates whether the register holds a valid address. See Fault address reporting on synchronous External aborts on page G5-9252.
d. All other debug exceptions are not permitted in Hyp mode.
MMU stage 2 fault on a stage 1 translation

This is an MMU fault generated on the stage 2 translation of an address accessed in a stage 1 translation table walk performed in the Non-secure PL1&0 translation regime. For more information about these faults see "Stage 2 fault on a stage 1 translation table walk" on page G5-9246.

Figure G5-1 on page G5-9148 shows the different translation regimes and associated stages of translation.
G5.13 Address translation instructions

The System register encoding space includes encodings for instructions that either:

- Translate a virtual address (VA) to a physical address (PA).
- Translate a virtual address (VA) to an intermediate physical address (IPA).

Address translation system instructions on page K17-11906 summarizes these instructions.

When using the Short-descriptor translation table format, all translations performed by these instructions take account of TEX remap when this is enabled, see Short-descriptor format memory region attributes, with TEX remap on page G5-9207.

An address translation instruction that executes successfully returns the output address, a PA or an IPA, in the PAR. This is a 64-bit register that can hold addresses of up to 40 bits.

It is IMPLEMENTATION DEFINED whether the address translation instructions return the values held in a TLB or the result of a translation table walk. Therefore, Arm recommends that these instructions are not used at a time when the TLB entries might be different from the underlying translation tables held in memory.

The following sections give more information about these instructions:

- Address translation instruction naming and operation summary.
- Encoding and availability of the address translation instructions on page G5-9273.
- Determining the PAR format on page G5-9275.
- Handling of faults and aborts during an address translation instruction on page G5-9275.

G5.13.1 Address translation instruction naming and operation summary

Some older documentation uses the original names for the address translation instructions that were included in the original Armv7 documentation. Table G5-34 on page G5-9271 summarizes the instructions that are available in AArch32 state, and relates the old instruction names to the current names.

### Table G5-34 Naming of address translation instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Old name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATS1CPR, ATS1CPW, ATS1CUR, ATS1CUW</td>
<td>V2PCWPR, V2PCWPW, V2PCWUR, V2PCWUW</td>
<td>See ATS1C**, Address translation stage 1, current security state on page G5-9272</td>
</tr>
<tr>
<td>ATS1CPRP, ATS1CPWP</td>
<td>Not applicablea</td>
<td></td>
</tr>
<tr>
<td>ATS12NSOPR, ATS12NSOPW, ATS12NSOUR, ATS12NSOUW</td>
<td>V2POWPR, V2POWPW, V2POWUR, V2POWUW</td>
<td>See ATS12NSO**, Address translation stages 1 and 2, Non-secure state only on page G5-9272</td>
</tr>
<tr>
<td>ATS1HR, ATS1HW</td>
<td>Not applicableb</td>
<td>See ATS1H*, Address translation stage 1, Hyp mode on page G5-9273</td>
</tr>
</tbody>
</table>

- a. Instructions are added by FEAT_PAN2 and do not have a previous name.
- b. Instructions are part of EL2 and have no equivalent in the older descriptions.

In an implementation that does not include EL2, there is no distinction between stage 1 translations and stage 1 and 2 combined translations.

For the stage 1 current state and stages 1 and 2 Non-secure state only instructions, the meanings of the final letters of the names are:

- **PR** PL1 mode, read operation.
- **PRP** PL1 mode, read operation, taking account of PSTATE.PAN.
- **PW** PL1 mode, write operation.
- **PWP** PL1 mode, write operation, taking account of PSTATE.PAN.
- **UR** User mode, read operation.
- **UW** User mode, write operation.
Note
User mode can be described as the unprivileged mode. It is the only EL0 mode.

For the stage 1 Hyp mode instructions, the last letter of the instruction name is R for the read operation and W for the write operation.

See also Encoding and availability of the address translation instructions on page G5-9273.

ATS1C**, Address translation stage 1, current security state

Any VMSAv8-32 implementation supports the ATS1C** instructions. They can be executed by any software executing at PL1 or higher, in either Security state.

The ATS1C** instructions are ATS1CPR, ATS1CPW, ATS1CUR, and ATS1CUW and, when FEAT_PAN2 is implemented, ATS1CPRP and ATS1CPWP. These instructions perform the address translations of the PL1&0 translation regime.

In an implementation that includes EL2, when executed in Non-secure state, these instructions return the IPA that is the output address of the stage 1 translation. Figure G5-1 on page G5-9148 shows the different translation regimes.

Note
The Non-secure PL1 and EL0 modes have no visibility of the stage 2 address translations, that can be defined only at EL2, and translate IPAs to be PAs.

See Determining the PAR format on page G5-9275 for the format used when returning the result of these instructions.

ATS12NSO**, Address translation stages 1 and 2, Non-secure state only

A VMSAv8-32 implementation supports the ATS12NSO** instructions only if it includes EL2. In an implementation that includes EL2, in AArch32 state, they can be executed:

• By software executing in Non-secure state at EL2. This means by software executing in Hyp mode.
• If the implementation includes EL3, when EL3 is using AArch32, by software executing in Secure state at PL1.

The ATS12NSO** instructions are ATS12NSOPR, ATS12NSOPW, ATS12NSOUR, and ATS12NSOUW.

In an implementation that includes EL3, when EL3 is using AArch64 and EL1 is using AArch32, any execution of an ATS12NSO** instruction at Secure EL1 is trapped as an exception that is taken to EL3.

In an implementation that does not include EL2, but includes EL3 then these instructions are not UNDEFINED but each instruction behaves in the same way as the equivalent ATS1C** instruction.

If an implementation does not include EL2 and does not include EL3 then these instructions are CONSTRAINED UNPREDICTABLE, with the permitted behavior that the instructions are UNDEFINED, see Unallocated System register access instructions on page K1-11565.

Arm deprecates use of these instructions from any Secure PL1 mode other than Monitor mode.

In Secure state and in Non-secure Hyp mode these instructions perform the translations made by the Non-secure PL1&0 translation regime.

These instructions always return the PA and final attributes generated by the translation. That is, for an implementation that includes EL2, they return:

• The result of the two stages of address translation for the specified Non-secure input address.
• The memory attributes obtained by the combination of the stage 1 and stage 2 attributes.
From Hyp mode, the ATS1C** and ATS12NSO** instructions both return the results of address translations that would be performed in the Non-secure modes other than Hyp mode. The difference is:

- The ATS1C** instructions return the Non-secure PL1 view of the associated address translation. That is, they return the IPA output address corresponding to the VA input address.
- The ATS12NSO** instructions return the EL2, or Hyp mode, view of the associated address translation. That is, they return the PA output address corresponding to the VA input address, generated by two stages of translation.

See Determining the PAR format on page G5-9275 for the format used when returning the result of these instructions.

ATS1H*, Address translation stage 1, Hyp mode

A VMSAv8-32 implementation supports the ATS1H* instructions only if it includes EL2. They can be executed by:

- Software executing in Non-secure state at EL2. This means by software executing in Hyp mode.
- Software executing in Secure state in Monitor mode.

The ATS1H* instructions are ATS1HR and ATS1HW. In an implementation that includes EL3, these instructions are CONSTRAINED UNPREDICTABLE if executed in a Secure PL1 mode other than Monitor mode.

If an implementation does not include EL2 then these instructions are CONSTRAINED UNPREDICTABLE, with the permitted behavior that the instructions are UNDEFINED, see Unallocated System register access instructions on page K1-11565.

These instructions perform the translations made by the Non-secure EL2 translation regime. The instruction takes a VA input address and returns a PA output address.

These instructions always return a result in a 64-bit format PAR.

G5.13.2 Encoding and availability of the address translation instructions

Software executing at EL0 never has any visibility of the address translation instructions, but software executing at PL1 or higher can use the unprivileged address translation instructions to find the address translations used for memory accesses by software executing at PL1 and EL0.

For information about translations when the stage of address translation is disabled see The effects of disabling address translation stages on VMSAv8-32 behavior on page G5-9154.

Table G5-35 on page G5-9273 shows the encodings for the address translation instructions, and their availability in different implementations in different PE modes and states.

Table G5-35 Address translation instructions in AArch32 state

<table>
<thead>
<tr>
<th>opc1</th>
<th>CRm</th>
<th>opc2</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All VMSAv8-32 implementations, in all modes, at PL1 or higher, see ATS1C**, Address translation stage 1, current security state on page G5-9272</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The result of an instruction is always returned in the PAR. The PAR is a RW register and:

- In all implementations, the 32-bit format PAR is accessed using an \texttt{MCR} or \texttt{MRC} instruction with CRn set to c7, CRm set to c4, and opc1 and opc2 both set to 0.
- The 64-bit format PAR is accessed using an \texttt{MCRR} or \texttt{MRRC} instruction with CRm set to c7, and opc1 set to 0.

Address translation instructions that are not available in a particular implementation are reserved and CONSTRAINED UNPREDICTABLE. For example:

- In an implementation that does not include EL2, the encodings with an opc1 value of 4 are reserved and CONSTRAINED UNPREDICTABLE. These are the \texttt{ATS1H*} instructions.
- In an implementation that does not include either EL2 or EL3, the encodings with opc2 values of 4-7 are reserved and CONSTRAINED UNPREDICTABLE. These are the \texttt{ATS12NSO**} instructions.

The CONSTRAINED UNPREDICTABLE behavior of these encodings is that they are UNDEFINED, see *Unallocated System register access instructions* on page K1-11565.
G5.13.3 Determining the PAR format

The PAR is a 64-bit register that supports both 32-bit and 64-bit PAR formats. This section describes how the PAR format is determined, for returning a result from each of the groups of address translation instructions. The returned result might be the translated address, or might indicate a fault on the translation, see Handling of faults and aborts during an address translation instruction.

ATS1C** instructions

Address translations for the current state. From modes other than Hyp mode:

- TTBCR.EAE determines whether the result is returned using the 32-bit or the 64-bit PAR format.
- If the implementation includes EL3, the translation performed is for the current security state and, depending on that state:
  - The Secure or Non-secure TTBCR.EAE determines the PAR format.
  - The result is returned to the Secure or Non-secure instance of the PAR

Instructions executed in Hyp mode always return a result to the Non-secure PAR, using the 64-bit format.

ATS12NSO** instructions

Address translations for the Non-secure PL1 and EL0 modes. These instructions return a result using the 64-bit PAR format if at least one of the following is true:

- The Non-secure TTBCR.EAE bit is set to 1.
- The implementation includes EL2, and the value of HCR.VM is 1.

Otherwise, the instruction returns a result using the 32-bit PAR format.

Instructions executed in a Secure PL1 mode return a result to the Secure PAR. Instructions executed in Hyp mode return a result to the Non-secure PAR.

ATS1H* instructions

Address translations from Hyp mode. These instructions always return a result using the 64-bit PAR format.

Instructions executed in Secure Monitor mode return a result to the Secure PAR. Instructions executed in Non-secure Hyp mode return a result to the Non-secure PAR.

G5.13.4 Handling of faults and aborts during an address translation instruction

When a stage of address translation is enabled, any corresponding address translation instruction requires a translation table lookup, and this might require a translation table walk. However, the input address for the translation might be a faulting address, either because:

- The translation table entries used for the translation indicate a fault.
- A stage 2 fault or an External abort occurs on the required translation table walk.

VMSAv8-32 memory aborts on page G5-9238 describes the faults that might occur on a translation table walk in AArch32 state.

How the fault is handled, and whether it generates an exception, depends on the cause of the fault, as described in:

- MMU fault on an address translation instruction.
- External abort during an address translation instruction on page G5-9276.
- Stage 2 fault on a current state address translation instruction on page G5-9276.

MMU fault on an address translation instruction

In the following cases, an MMU fault on an address translation is reported in the PAR, and no abort is taken. This applies:

- For a faulting address translation instruction executed in Hyp mode, or in a Secure PL1 mode.
• For a faulting address translation instruction executed in a Non-secure PL1 mode, for cases where the fault would generate a stage 1 abort if it occurred on the equivalent load or store operation.

Using the PAR to report a fault on an address translation instruction gives more information about how these faults are reported.

—— Note ————

• The Domain fault encodings shown in Table G5-27 on page G5-9258 are used only for reporting a fault on an address translation instruction that uses the 64-bit PAR format. That is, they are used only in an implementation that includes EL2, and are used for reporting a Domain fault on either:
  — An ATS1C** instruction executed in Hyp mode.
  — An ATS12NSO** instruction executed when the value of HCR.VM is 1.

These encodings are never used for fault reporting in the DFSR, IFSR, or HSR.

• For an address translation instruction executed in a Non-secure PL1 mode, for a fault that would generate a stage 2 abort if it occurred on the equivalent load or store operation, the stage 2 abort is generated as described in Stage 2 fault on a current state address translation instruction.

Using the PAR to report a fault on an address translation instruction

For a fault on an address translation instruction for which no abort is taken, the PAR is updated with the following information, to indicate the fault:

• The fault code, that would normally be written to the Fault status register. The code used depends on the current translation table format, as described in either:
  — PL1 fault reporting with the Short-descriptor translation table format on page G5-9256.
  — PL1 fault reporting with the Long-descriptor translation table format on page G5-9258.

See also the Note at the start of Determining the PAR format on page G5-9275 about the Domain fault encodings shown in Table G5-27 on page G5-9258.

• A status bit, that indicates that the translation operation failed.

The fault does not update any Fault Address Register.

External abort during an address translation instruction

As stated in External abort on a translation table walk on page G5-9247, an External abort on a translation table walk generates a Data Abort exception. The abort can be synchronous or asynchronous, and behaves as follows:

Synchronous External abort on a translation table walk

The fault status and fault address registers of the Security state to which the abort is taken are updated. The fault status register indicates the appropriate External abort on a Translation fault, and the fault address register indicates the input address for the translation.

The PAR is UNKNOWN.

Asynchronous External abort on a translation table walk

The fault status register of the Security state to which the abort is taken is updated, to indicate the asynchronous External abort. No fault address registers are updated.

The PAR is UNKNOWN.

Stage 2 fault on a current state address translation instruction

If the PE is in a Non-secure PL1 mode and executes one of the ATS1C** instructions, then a fault in the stage 2 translation of an address accessed in a stage 1 translation table lookup generates an exception. This is equivalent to the case described in Stage 2 fault on a stage 1 translation table walk on page G5-9246. When this fault occurs on an ATS1C** address translation instruction:

• A Hyp Trap exception is taken to Hyp mode.
• The **PAR** is **UNKNOWN**.
• The **HSR** indicates that:
  — The fault occurred on a translation table walk.
  — The operation that faulted was a cache maintenance instruction.
• The **HPFAR** holds the IPA that faulted.
• The **HDFAR** holds the VA that the executing software supplied to the address translation instruction.
G5.14 Pseudocode description of VMSAv8-32 memory system operations

This section contains a list of pseudocode functions describing VMSAv8-32 memory operations. The following subsections describe the pseudocode functions:

• Full Physical Address.
• Translation regime.
• Address translation.
• Long-descriptor Translation table walk on page G5-9279.
• Short-descriptor Translation table walk on page G5-9279.
• Memory attribute decoding on page G5-9279.
• Fault detection on page G5-9279.

See also the descriptions of pseudocode for general memory system operations in Pseudocode description of general memory System instructions on page G4-9142.

G5.14.1 Full Physical Address

A complete physical address necessary to identify a location in physical memory is captured by the type FullAddress. This is composed of:

• A bitstring address, which identifies the physical address.
• An enumeration paspace, which identifies the physical address space.

G5.14.2 Translation regime

The architecture specifies translation regimes in terms of Privilege Level (PL). An alternative approach is used in pseudocode where regimes are expressed in terms of ELs instead, mirroring regimes in AArch64. The pseudocode and ARM use a differently named but equivalent set of regimes:

<table>
<thead>
<tr>
<th>Pseudocode Regime</th>
<th>Equivalent ARM regime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regime_EL10</td>
<td>Secure PL1&amp;0 when EL3 is AArch64 or Non-Secure PL1&amp;0</td>
</tr>
<tr>
<td>Regime_EL30</td>
<td>Secure PL1&amp;0 when EL3 is AArch32</td>
</tr>
<tr>
<td>Regime_EL2</td>
<td>Non-Secure PL2</td>
</tr>
</tbody>
</table>

G5.14.3 Address translation

AArch32.TranslateAddress() acts as the entry point to VMSAv8-32 and performs the required address translation based on the provided parameters and system register configurations. The function returns an AddressDescriptor structure holding valid data for either of the following:

• Target memory address and attributes for a non-faulting translation.
• Fault details holding data to be populated in syndrome registers.

AArch32.FullTranslate() selects the translation regime and performs first and potentially second stage of translation returning the physical address (PA) and attributes of target memory. AArch32.S1TranslateLD() carries out the first stage of translation when stage 1 is not disabled and Long-descriptor format is used, mapping the virtual address (VA) to the intermediate physical address (IPA) and carrying out permission checks. Alternatively AArch32.S1TranslateSD() carries out the first stage of translation using the Short-descriptor format along with Domain checks and TEX memory attribute mapping. Otherwise, AArch32.S1DisabledOutput() assigns the appropriate memory attributes and flat maps the input address to the output address. AArch32.S2Translate() carries out stage 2 translation for Regime_EL10 when enabled, mapping the IPA to the PA. Otherwise, the IPA is the PA.
G5.14.4 Long-descriptor Translation table walk

A separate walk function is dedicated for Stage 1 Long-descriptor format, AArch32.S1WalkLD(), and Stage 2, AArch32.S2Walk(), which supports only Long-descriptor format. Each use walk parameters extracted from related system registers and held in S1TTWParams for stage 1 and S2TTWParams for stage 2. Parameters are collected based on the active translation regime. For instance, stage 1 EL2 translation regime parameters are obtained and returned by the function AArch32.S1TTWParamsEL2(). Given these parameters, a walk initializes a walk state of the type TTWState, holding the base address of the first translation table.

The walk progressively fetches and decodes Translation Table descriptors, updating the walk state to the next base address as it descends through the levels of tables until a Block or Page descriptor is discovered or an invalid descriptor is fetched. Decoding the descriptor for both stage 1 and stage 2 walks is carried out by the function AArch32.DecodeDescriptorTypeLD().

For a non-faulting walk, a valid final walk state is returned, otherwise a faulting walk could report one of the following at a specified level:
- Translation Fault.
- Address Size Fault.
- Access Flag Fault.

G5.14.5 Short-descriptor Translation table walk

Short-Descriptor format is only supported for Regime_EL10 and Regime_EL30 (PL1&0) Stage 1 and a separate walk function is dedicated for that, AArch32.S1WalkSD(). The limited number of parameters are collected in the walk function and would otherwise follow a similar flow to Long-descriptor formats of iteratively updating the walk state. The walk notably collects the domain and Short-descriptor format type which are unique to Short-descriptor formats. The descriptor type is decoded using AArch32.DecodeDescriptorTypeSD().

For a non-faulting walk, a valid final walk state is returned, otherwise a faulting walk could report one of the following at a specified level:
- Translation Fault.
- Address Size Fault.
- Access Flag Fault (when SCTLR.AFE is configured to support Access flags).

G5.14.6 Memory attribute decoding

If a stage of translation is enabled, Fetched Leaf descriptors encode memory attributes assigned to the output of translation. Stage 1 Long-descriptor format memory attributes are decoded by the function S1DecodeMemAttrs(). Likewise, stage 2 memory attributes are decoded by the function S2DecodeMemAttrs() followed by combining stage 1 and stage 2 attributes by the function S2CombineS1MemAttrs(). A separate set of functions are used to assign memory attributes to the output of Short-descriptor format. AArch32.DefaultTEXDecode() is used when TEX remapping is disabled, otherwise AArch32.RemappedTEXDecode() defines output memory attributes.

G5.14.7 Fault detection

As soon as translation is invoked, a reserve FaultRecord accompanies the process, capturing the stage and level of translation as it proceeds. When a fault is detected, it is reflected in the FaultRecord and reported back as the result of translation with the most recent state to be reported already captured within. The following functions detect a certain type of fault, their outputs are all boolean with a TRUE value on detection:

- AArch32.S1LDHasPermissionsFault() and AArch32.S2HasPermissionsFault() detect a permissions fault for stage 1 and stage 2 respectively for Long-descriptor format. AArch32.S1SDHasPermissionsFault() detects a permissions fault for a translation in Short-descriptor format. Note that for atomic instructions introduced by FEAT_LSE, these functions are called twice, once to check for read permissions and another for write allowing the correct failure to be reported.

- AArch32.S1 HasAlignmentFault() and AArch32.S2HasAlignmentFault() detect an alignment fault for stage 1 and stage 2 respectively.
• **AArch32.S2InconsistentSL()** detects a stage 2 translation fault caused by erroneous configuration of the VTCR.SL0 field.

• **AArch32.VAIsOutOfRange()** detects a stage 1 translation fault caused by virtual addresses larger than the address input size configured. Similarly, **AArch64.IPAIsOutOfRange()** detects a stage 2 translation fault caused by the output of stage 1 being larger than the configured input size for stage 2. Both are solely part of Long-descriptor format translation.

--- Note ---

Domain faults are detected inline as part of **AArch32.S1TranslateSD()** since they are a simple equality check.
G5.15 About the System registers for VMSAv8-32

The System registers and System instructions that are accessible in AArch32 state are almost all in the encoding space described in *The AArch32 System register interface* on page G1-8993. This section gives general information about these registers, which comprise:

- Registers in the \( \text{coproc} = 0b1111 \) encoding space, that provide control and status information for the PE in Non-debug state.
- Registers in the \( \text{coproc} = 0b1110 \) encoding space, including:
  - Debug registers.
  - Trace registers.
  - Legacy execution environment registers.

*AArch32 VMSA organization of registers in the \( \text{coproc} = 0b1110 \) encoding space* on page G7-9301 summarizes the registers in the \( \text{coproc} = 0b1110 \) encoding space, and indicates where these registers are described, either in this manual or in other architecture specifications.

*AArch32 VMSA organization of registers in the \( \text{coproc} = 0b1111 \) encoding space* on page G7-9304 summarizes the registers in the \( \text{coproc} = 0b1111 \) encoding space, and indicates where in this manual these registers are described.

**Note**

Many implementations include other interfaces to some System registers, for example a memory-mapped interface to some debug System registers. These are described in the appropriate sections of this manual.

G5.15.1 Classification of System registers

Features provided by EL3 and EL2 integrate with many features of the architecture. Therefore, the descriptions of the individual System registers include information about how these Exception levels affect the register. This section:

- Summarizes how EL3 and EL2 affect the implementation of the System registers, and the classification of those registers.
- Summarizes how EL3 controls access to the System registers.
- Describes an EL3 signal that can control access to some registers in the \( \text{coproc} = 0b1111 \) encoding space.

It contains the following subsections:

- Banked System registers.
- Restricted access System registers on page G5-9282.
- Configurable access System registers on page G5-9282.
- EL2-mode System registers on page G5-9283.
- Common System registers on page G5-9284.
- Access to registers from Monitor mode on page G5-9284.
- *The CP15SDISABLE and CP15SDISABLE2 input signals* on page G5-9285.

**Note**

EL3 defines the register classifications of Banked, Restricted access, Configurable, and Common. EL2 defines the EL2-mode classification.

It is IMPLEMENTATION DEFINED whether each IMPLEMENTATION DEFINED register is Banked, Restricted access, Configurable, EL2-mode, or Common.

**Banked System registers**

In an implementation that includes EL3 using AArch32, some System registers are banked. Banked System registers have two copies, one Secure and one Non-secure. The SCR.NS bit selects the Secure or Non-secure instance of the register.
A Banked System register can contain a mixture of:
- Fields that are banked.
- Fields that are read-only in Non-secure PL1 or EL2 modes but read/write in the Secure state.

The System Control Register SCTLR is an example of a register that contains this mixture of fields.

The Secure copies of the Banked System registers are sometimes referred to as the Secure Banked System registers. The Non-secure copies of the Banked System registers are sometimes referred to as the Non-secure Banked System registers.

**Restricted access System registers**

In an implementation that includes EL3, some System registers are present only in Secure state. These are called Restricted access registers, and their read/write access permissions are:

- In Non-secure state, software cannot modify Restricted access registers.
- For the NSACR, in Non-secure state:
  - Software running at PL1 or higher can read the register.
  - Unprivileged software, meaning software running at EL0, cannot read the register.

This means that Non-secure software running at PL1 or higher can read the access permissions for System registers that have Configurable access.

If EL3 is using AArch64, then any read of the NSACR from Non-secure EL2 using AArch32, or Non-secure EL1 using AArch32, returns the value 0x00000C00.

- For all other Restricted access registers, Non-secure software cannot read the register.

In an implementation that does not include EL3:
- SDER is implemented only in Secure state.
- Any read of the NSACR returns the value 0x00000C00.
- All other accesses to Restricted access System registers are UNDEFINED.

**Configurable access System registers**

Secure software can configure the access to some System registers. These registers are called Configurable access registers, and the control can be:

- A bit in the control register determines whether the register is:
  - Accessible from Secure state only.
  - Accessible from both Secure and Non-secure states.

- A bit in the control register changes the accessibility of a register bit or field. For example, setting a bit in the control register might mean that an RW field behaves as RAZ/WI when accessed from Non-secure state.

Bits in the NSACR control access.

In an AArch32 implementation that includes EL3:
- There are no Configurable access System registers in the (coproc==0b1110) encoding space.
- The only required Configurable access register in the (coproc==0b1111) encoding space is the CPACR.
  - Floating-point Status and Control Register, FPSCR
  - Floating-point Exception register, FPEXC.
  - Floating-point System ID register, FPSID.
  - Media and VFP Feature Register 0, MVFR0.
  - Media and VFP Feature Register 1, MVFR1.
  - Media and VFP Feature Register 2, MVFR2.
EL2-mode System registers

In an implementation that includes EL2, if EL2 can use AArch32, the implementation provides a number of registers for use in the EL2 mode, Hyp mode. As with other System register encodings, some of these register encodings provide write-only operations. When the implementation includes EL3 and EL3 is using AArch32, these registers are also accessible from Monitor mode when the value of SCR.NS is 1.

The following subsections describe the EL2-mode registers:

- Hyp mode read/write registers in the (coproc==0b1111) encoding space.
- Hyp mode encodings for shared (coproc==0b1111) System registers.
- Hyp mode (coproc==0b1111) write-only System instructions on page G5-9284.

There are no EL2-mode registers in the (coproc==0b1110) encoding space.

Hyp mode read/write registers in the (coproc==0b1111) encoding space

These registers are implemented only in Non-secure state, and in Non-secure state they are accessible only from Hyp mode.

Except for accesses to CNTVOFF in an implementation that includes EL3 but not EL2, the behavior of accesses to these registers is as follows:

- In Secure state, the registers can be accessed from EL3 when SCR.NS is set to 1, see Access to registers from Monitor mode on page G5-9284.
- The following accesses are UNDEFINED:
  - Accesses from Non-secure PL1 modes.
  - Accesses in Secure state when SCR.NS is set to 0.

In an implementation that includes EL3 but not EL2, the behavior of accesses to CNTVOFF is as follows:

- Any access from Secure Monitor mode is CONSTRAINED UNPREDICTABLE, regardless of the value of SCR.NS. The CONSTRAINED UNPREDICTABLE behavior is that the access is UNDEFINED, see Unallocated System register access instructions on page K1-11565.
- All other accesses are UNDEFINED.

Note

Except for CNTVOFF, the Hyp mode registers are part of EL2, meaning they are implemented only if the implementation includes EL2. However, conceptually, CNTVOFF is part of any implementation of the Generic Timer, see The virtual offset register on page G6-9295. This means the behavior of CNTVOFF in an implementation that does not include EL2 is not covered by the general definition of the behavior of the Hyp mode (coproc==0b1111) read/write registers.

Hyp mode encodings for shared (coproc==0b1111) System registers

Some Hyp mode registers share the Secure instance of an existing banked register. In this case, the implementation includes an encoding for the register that is accessible only in Hyp mode, or in Monitor mode when SCR.NS is set to 1.

For these registers, the following accesses are UNDEFINED:

- Accesses from Non-secure PL1 modes.
- Accesses in Secure state when SCR.NS is set to 0.

In Monitor mode, the Secure copies of these registers can be accessed either:

- Using the DFAR or IFAR encoding with SCR.NS set to 0.
- Using the HDFAR or HIFAR encoding with SCR.NS set to 1.

However, between accessing a register using one alias and accessing the register using the other alias, a Context synchronization event is required to ensure the ordering of the accesses.
Hyp mode (coproc==0b1111) write-only System instructions

Architecturally, these encodings are an extension of the banked register encodings described in Banked System registers on page G5-9281, where:

- The implementation does not implement the operation in Secure state.
- In Non-secure state, the operation is accessible only at EL2, that is, only from Hyp mode.

In Secure state:

- These instructions can be accessed from Monitor mode regardless of the value of SCR.NS, see Access to registers from Monitor mode.
- Accesses to these instructions are CONSTRAINED UNPREDICTABLE if executed in a Secure mode other than Monitor mode.

Accesses to these instructions are UNDEFINED if accessed from a Non-secure PL1 mode.

Common System registers

Some System registers and operations are common to the Secure and Non-secure Security states. These are described as the Common access registers, or simply as the Common registers. These registers include:

- Read-only registers that hold configuration information.
- Register encodings used for various memory system operations, rather than to access registers.
- The ISR.
- All System registers in the (coproc==0b1110) encoding space.

Secure System registers for the (coproc==0b1111) encoding space

The Secure System registers in the (coproc==0b1111) encoding space comprise:

- The Secure copies of the Banked System registers in the (coproc==0b1111) encoding space.
- The Restricted access System registers in the (coproc==0b1111) encoding space.
- The Configurable access System registers in the (coproc==0b1111) encoding space that are configured to be accessible only from Secure state.

In an implementation that includes EL3, the Non-secure System registers are the System registers other than the Secure System registers.

Access to registers from Monitor mode

When the PE is in Monitor mode, the PE is in Secure state regardless of the value of the SCR.NS bit. In Monitor mode, the SCR.NS bit determines whether, for System registers in the (coproc==0b1111) encoding space, valid uses of the MRC, MCR, MRRC, and MCRR instructions access the Secure Banked System registers or the Non-secure Banked System registers. That is, when:

\[ NS == 0 \]

Common, Restricted access, and Secure Banked System registers are accessed by MRC, MCR, MRRC, and MCRR instructions that target the (coproc==0b1111) encoding space.

If the implementation includes EL2, the registers listed in Hyp mode read/write registers in the (coproc==0b1111) encoding space on page G5-9283 and Hyp mode encodings for shared (coproc==0b1111) System registers on page G5-9283 are not accessible, and any attempt to access them generates an Undefined Instruction exception.

Note

The operations listed in Hyp mode (coproc==0b1111) write-only System instructions are accessible in Monitor mode regardless of the value of SCR.NS.
System instructions in the (coproc==0b1111) encoding space use the Security state to determine all resources used, that is, all operations performed by these instructions are performed in Secure state.

NS == 1

Common, Restricted access and Non-secure Banked System registers are accessed by MRC, MCR, MRRC, and MCRR instructions that target the (coproc==0b1111) encoding space.

If the implementation includes EL2, all the registers and operations listed in the subsections of EL2-mode System registers on page G5-9283 are accessible, using the MRC, MCR, MRRC, or MCRR instructions required to access them from Hyp mode.

System instructions in the (coproc==0b1111) encoding space use the Security state to determine all resources used, that is, all operations by these instructions are performed in Secure state.

The Security state determines whether the Secure or Non-secure banked registers determine the control state.

Note

Where the contents of a register select the value accessed by an MRC or MCR access to a different register, then the register that is used for selection is being used as control state. For example, CSSEL R selects the current Cache Size Identification Register, and therefore CSSEL R is used as control state. Therefore, in Monitor mode:

- SCR.NS determines whether the Secure or Non-secure CSSEL R is accessible.
- Because the PE is in Secure state, the Secure CSSEL R selects the current Cache Size Identification Register.

From Armv8.3, it is possible to have multiple Cache Size Identification Registers. For more details, see Possible formats of the Cache Size Identification Registers, CCSIDR and CCSIDR2 on page G4-9115.

The CP15SDisable and CP15SDisable2 input signals

When EL3 is using AArch32, it provides an input signal, CP15SDisable, that disables write access to some of the Secure registers when asserted HIGH. The CP15SDisable signal has no effect on:

- Register accesses from AArch64 state.
- Register accesses from Secure EL1 when EL3 is using AArch64 and EL1 is using AArch32.

Note

When EL3 is using AArch32, the interaction between CP15SDisable and any IMPLEMENTATION DEFINED register is IMPLEMENTATION DEFINED.

On a Warm reset by the external system that resets the PE into EL3 using AArch32, the CP15SDisable input signal must be taken LOW. This permits the Reset code to set up the configuration of EL3 features. When the input is asserted HIGH, any attempt to write to the Secure registers that are affected by CP15SDisable results in an Undefined Instruction exception.

The CP15SDisable input does not affect reading Secure registers, or reading or writing Non-secure registers. It is IMPLEMENTATION DEFINED how the input is changed and when changes to this input are reflected in the PE, and an implementation might not provide any mechanism for driving the CP15SDisable input HIGH. However, in an implementation in which the CP15SDisable input can be driven HIGH, changes in the state of CP15SDisable must be reflected as quickly as possible. Any change must occur before completion of an Instruction Synchronization Barrier operation, issued after the change, is visible to the PE with respect to instruction execution boundaries. Software must perform an Instruction Synchronization Barrier operation meeting the above conditions to ensure all subsequent instructions are affected by the change to CP15SDisable.

When EL3 is using AArch32, use of CP15SDisable means key Secure features that are accessible only at PL1 can be locked in a known state. This provides an additional level of overall system security. Arm expects control of CP15SDisable to reside in the system, in a block dedicated to security.

When FEAT CP15SDisable2 is implemented and EL3 is using AArch32, EL3 provides a second input signal, CP15SDisable2. CP15SDisable2 has all of the properties of CP15SDisable described above. The difference between CP15SDisable and CP15SDisable2 is only in the set of registers each signal affects.

Information on whether a given register is affected by CP15SDisable, or CP15SDisable2 when it is implemented, can be found in the access pseudocode for that register, as described in Chapter G8 AArch32 System Register Descriptions.
G5.16  Functional grouping of VMSAv8-32 System registers

This section describes how the System registers in an VMSAv8-32 implementation divide into functional groups. The functional groups of AArch32 registers are:

- Special-purpose registers.
- VMSA-specific registers.
- ID registers.
- Performance monitors registers.
- Activity monitors registers.
- Debug registers.
- The Reliability, Availability, and Serviceability Extension registers.
- Generic timer registers.
- Cache maintenance System instructions.
- Address translation System instructions.
- TLB maintenance System instructions.
- Base system registers.
- Legacy feature registers and System instructions.

For a list of these functional groups and the registers in each group, see Functional index of AArch32 registers and System instructions on page K17-11899.

Chapter G8 AArch32 System Register Descriptions describes each of these registers.

Note

- The functional groups defined in this section mainly consist of the VMSAv8-32 System registers, but include some additional System registers.
- Some registers belong to more than one functional group.

For other related information, see:

- The AArch32 System register interface on page G1-8993 for general information about the access to the AArch32 System registers, including the main register access instructions MRC and MCR.
- About the System registers for VMSAv8-32 on page G5-9281.
- AArch32 VMSA organization of registers in the (coproc==0b1111) encoding space on page G7-9301.
- AArch32 VMSA organization of registers in the (coproc==0b1111) encoding space on page G7-9304.
- About the AArch32 System registers on page G8-9322.

The register descriptions in Chapter G8 AArch32 System Register Descriptions, assume you are familiar with these functional groups, and use conventions and other information from them without any explanation.
Chapter G6
The Generic Timer in AArch32 state

This chapter describes the implementation of the Arm Generic Timer as an extension to an Armv8 AArch32 implementation. It includes an overview of the AArch32 System register interface to an Arm Generic Timer.

It contains the following sections:
• About the Generic Timer in AArch32 state on page G6-9288.
• The AArch32 view of the Generic Timer on page G6-9292.

Chapter D10 The Generic Timer in AArch64 state describes the AArch64 view of the Generic Timer, including additional timers that can be implemented in AArch64 state, and Chapter I2 System Level Implementation of the Generic Timer describes the system level implementation of the Generic Timer.
G6.1 About the Generic Timer in AArch32 state

Figure G6-1 shows an example system-on-chip that uses the Generic Timer as a system timer. In this figure:

- This manual defines the architecture of the individual PEs in the multiprocessor blocks.
- The ARM Generic Interrupt Controller Architecture Specification defines a possible architecture for the interrupt controllers.
- Generic Timer functionality is distributed across multiple components.

The Generic Timer:

- Provides a system counter, that measures the passing of time in real-time.

  — **Note**
  
  The Generic Timer can also provide other components at a system level, but Figure G6-1 does not show any such components.

- Supports *virtual counters* that measure the passing of virtual-time. That is, a virtual counter can measure the passing of time on a particular virtual machine.

- Timers, that can trigger events after a period of time has passed. The timers:
  - Can be used as count-up or as count-down timers.
  - Can operate in real-time or in virtual-time.

This chapter describes an instance of the Generic Timer component that Figure G6-1 shows as Timer_0 or Timer_1 within the Multiprocessor A or Multiprocessor B block. This component can be accessed from AArch64 state or AArch32 state, and this chapter describes access from AArch32 state. Chapter D10 The Generic Timer in AArch64 state describes access to this component from AArch64 state.

  — **Note**

  The reset requirements of Generic Timer registers are more strict when they are accessed from AArch32 state than when they are accessed from AArch64 state.
A Generic Timer implementation must also include a memory-mapped system component. This component:

- Must provide the System counter shown in Figure G6-1 on page G6-9288.
- Optionally, can provide timer components for use at a system level.

Chapter 12 System Level Implementation of the Generic Timer describes this memory-mapped component.

### G6.1 The full set of Generic Timer components

Within a system that might include multiple PEs, a full set of Generic Timer components is as follows:

**The system counter**

This provides a uniform view of system time, see The system counter on page G6-9290. Because this must be implemented at the system level, it is accessed through The system level memory-mapped implementation of the Generic Timer. However, during initialization, a status register in each implemented timer in the system must be programmed with the frequency of the system counter, so that software can read this frequency.

**PE implementations of the Generic Timer**

Each PE implementation of the Generic Timer provides the following components:

- A physical counter, that gives access to the count value of the system counter. When FEAT_ECV is implemented, EL2 is using AArch64, and EL2 is implemented and enabled in the current Security state, the CNTPOFF_EL2 register allows offsetting of AArch32 physical timers and counters.
- A virtual counter, that gives access to virtual time. In AArch32 state, the CNTVOFF register defines the offset between physical time, as defined by the value of the system counter, and virtual time.
- A number of timers. In an implementation where all Exception levels are implemented and can use AArch32 state, the timers that are accessible from AArch32 state are:
  - A Secure PL1 physical timer.
  - A Non-secure EL1 physical timer.
  - A Non-secure EL2 physical timer.
  - An EL1 virtual timer.
  - A Non-secure EL2 virtual timer.
  - A Secure EL2 virtual timer.
  - A Secure EL2 physical timer.

The Non-secure EL2 virtual timer is available when FEAT_VHE is implemented. The Secure EL2 timers are available when FEAT_SEL2 is implemented, but are only accessible in AArch32 state if using EL0, when EL0 is using AArch32, Secure EL2 is using AArch64, and HCR_EL2.{E2H,TGE} == {1, 1}.

**Note**

The Secure PL1 physical timer uses the Secure banked instances of the CNTP_CTL, CNTP_CVAL, and CNTP_TVVAL registers, and the Non-secure EL1 physical timer uses the Non-secure instances of the same registers.

*The AArch32 view of the Generic Timer on page G6-9292 describes these components.*

**The system level memory-mapped implementation of the Generic Timer**

The memory-mapped registers that control the components of the system level implementation of the Generic Timer are grouped into frames. The Generic Timer architecture defines the offset of each register within its frame, but the base address of each frame is IMPLEMENTATION DEFINED, and defined by the system.
Each system level component has one or two register frames. The possible system level components are:

The memory-mapped counter module, required
This module controls the system counter. It has two frames:
• A control frame, CNTControlBase.
• A status frame, CNTReadBase.

The memory-mapped timer control module, required
The system level implementation of the Generic Timer can provide up to eight timers, and the memory-mapped timer control module identifies:
• Which timers are implemented.
• The features of each implemented timer.
This module has a single frame, CNTCTLBase.

Memory-mapped timers, optional
An implemented memory-mapped timer:
• Must provide a privileged view of the timer, in the CNTBaseN frame.
• Optionally, provides an unprivileged view of the timer in the CNTEL0BaseN frame.

N is the timer number, and the corresponding frame number, in the range 0-7.

Chapter 12 System Level Implementation of the Generic Timer describes these components.

G6.1.2 The system counter

The Generic Timer provides a system counter with the following specification:

Width
From Armv8.0 to Armv8.5 inclusive, at least 56 bits wide. The value returned by any 64-bit read of the counter is zero-extended to 64 bits.
From Armv8.6, must be 64 bits wide.

Frequency
From Armv8.0 to Armv8.5 inclusive, increments at a fixed frequency, typically in the range 1-50MHz. It can support one or more alternative operating modes in which it increments by larger amounts at a lower frequency, typically for power-saving.
From Armv8.6, increments at a fixed frequency of 1GHz.

Roll-over
Roll-over time of not less than 40 years.

Accuracy
Arm does not specify a required accuracy, but recommends that the counter does not gain or lose more than ten seconds in a 24-hour period.
Use of lower-frequency modes must not affect the implemented accuracy.

Start-up
Starts operating from zero.

The system counter, once configured and running, must provide a uniform view of system time. More precisely, it must be impossible for the following sequence of events to show system time going backwards:
1. Device A reads the time from the system counter.
2. Device A communicates with another agent in the system, Device B.
3. After recognizing the communication from Device A, Device B reads the time from the system counter.

The system counter must be implemented in an always-on power domain.

To support lower-power operating modes in architectures from Armv8.0 to Armv8.5, the counter can increment by larger amounts at a lower frequency. For example, a 10MHz system counter might either increment:
• By 1 at 10MHz.
• By 500 at 20kHz, when the system lowers the clock frequency, to reduce power consumption.

In this case, the counter must support transitions between high-frequency, high-precision operation, and lower-frequency, lower-precision operation, without any impact on the required accuracy of the counter.

From Armv8.6 the counter operates at a higher fixed frequency of 1GHz.
Note

Though each unit of the counter is set to 1ns, this does not require that the counter is incremented every 1ns. A step in the counter might be more than a single bit increment. It is recommended that the count is not incremented at a rate that is less than 50MHz in normal running operation.

The CNTFRQ register is intended to hold a copy of the current clock frequency to allow fast reference to this frequency by software running on the PE. For more information see Initializing and reading the system counter frequency.

The mechanism by which the count from the system counter is distributed to system components is IMPLEMENTATION DEFINED, but each PE with a System register interface to the system counter must have a counter input that can capture each increment of the counter.

Note

So that the system counter can be clocked independently from the PE hardware, the count value might be distributed using a Gray code sequence. Gray-count scheme for timer distribution scheme on page K7-11704 gives more information about this possibility.

Initializing and reading the system counter frequency

The CNTFRQ register must be programmed to the clock frequency of the system counter. Typically, this is done only during the system boot process, by using the System register interface to write the system counter frequency to the CNTFRQ register. Only software executing at the highest implemented Exception level can write to CNTFRQ.

Note

The CNTFRQ register is UNKNOWN at reset, and therefore the counter frequency must be set as part of the system boot process.

Software can read the CNTFRQ register, to determine the current system counter frequency, in the following states and modes:

- Hyp mode.
- Secure PL1 modes and Non-secure EL1 modes.
- When CNTKCTL.{PL0PCTEN, PL0VCTEN} is not {0,0}, Secure and Non-secure EL0 modes.

Memory-mapped controls of the system counter

Some system counter controls are accessible only through the memory-mapped interface to the system counter. These controls are:

- Enabling and disabling the counter.
- Setting the counter value.
- Changing the operating mode, to change the update frequency and increment value.
- Enabling Halt-on-debug, that a debugger can then use to suspend counting.

For descriptions of these controls, see Chapter I2 System Level Implementation of the Generic Timer.
G6.2 The AArch32 view of the Generic Timer

The following sections describe the components and features of a PE implementation of the Generic Timer, as seen from AArch32 state:

- The physical counter.
- The virtual counter on page G6-9293.
- Event streams on page G6-9295.
- Timers on page G6-9296.

G6.2.1 The physical counter

The PE includes a physical counter that contains the count value of the system counter. The $\text{CNTPCT}$ register holds the current physical counter value. When $\text{FEAT_ECV}$ is implemented and EL2 is executing in AArch64 state, the $\text{CNTPOFF_EL2}$ register holds the optional physical offset that can be applied to EL0 and EL1 whether EL0 and EL1 are using AArch64 state or AArch32 state. For more information, see The physical offset register on page D10-5237.

Reads of $\text{CNTPCT}$ can occur speculatively and out of order relative to other instructions executed on the same PE.

The self-synchronized view of the physical counter

When $\text{FEAT_ECV}$ is implemented, an alternative way to read the physical counter is supported. The $\text{CNTPCTSS}$ register is a non-speculative view of the physical counter, as seen from the Exception level that $\text{CNTPCTSS}$ is read from.

Access to the $\text{CNTPCTSS}$ are subject to the same traps as accesses to the $\text{CNTPCT}$.

Reads of $\text{CNTPCT}$ occur in program order relative to reads of $\text{CNTPCT}$ or $\text{CNTPCTSS}$.

Reads of $\text{CNTPCTSS}$ occur in program order relative to reads of $\text{CNTPCT}$ or $\text{CNTPCTSS}$.

Example G6-1 Ensuring reads of the physical counter occur after signal read from memory

If a read from memory is used to obtain a signal from another agent that indicates that $\text{CNTPCT}$ must be read, an $\text{ISB}$ is used to ensure that the read of $\text{CNTPCT}$ occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop ; polling for some communication to indicate a requirement to read the timer
  LDR R1, [R2]          ; has had the value 1 written to it
  CMP R1, #1
  BNE loop
  ISB ; without this the CNTPCT could be read before the memory location in [R2]
  MRC R1, CNTPCT
```

When $\text{FEAT_ECV}$ is implemented, an access to $\text{CNTPCTSS}$ can be used in place of the $\text{CNTPCT}$ which, because it cannot be accessed speculatively, allows the $\text{ISB}$ to be removed. This means that the following code sequence can be used:

```
loop ; polling for some communication to indicate a requirement to read the timer
  LDR R1, [R2]
  CMP R1, #1
  BNE loop
  MRC R1, CNTPCTSS
```

Similarly where a read of the physical counter is required to take place after the completion of all loads and stores appearing in program order before the read of the counter, then the following code sequences can be used:

```
... ; earlier loads and stores
DSB ; completes earlier loads and stores
ISB ; without this the CNTPCT could be read before the completion of the earlier loads
      ; and stores
  MRC R1, CNTPCT
```
Or, if FEAT_ECV is implemented:

```asm
... ; earlier loads and stores
DSB ; completes earlier loads and stores
MRC R1, CNTPCTSS
```

Neither view of the physical counter ensures that:

- Context changes occurring in program order before the read of the counter have been synchronized.
- Accesses to memory appearing in program order after the read of the counter are executed before the counter has been read.

Where there is a Dependency through registers dependency from the read of the physical counter to a Register effect generated by a read or write, the read or write will be executed after the read of the counter.

**Example G6-2 Ensuring reads of the physical counter occur after previous memory accesses**

To ensure that all previous memory accesses have completed and all previous context changes have been synchronized before the read of the counter, one of the following sequences should be used:

```asm
DSB
ISB
MRC Rn, CNTPCT{SS} ; either view of the physical counter has the same effect in this example
```

or

```asm
DSB
ISB
CBZ Ra, next
```

```asm
ISB ; this ISB is not needed if the MRC is accessing CNTPCT{SS}
MRC Rn, CNTPCT{SS}
```

To ensure that a memory access occurs only after a read of the counter, then either of the following sequences should be used:

```asm
MRC Rn, CNTPCT{SS} ; either view of the physical counter has the same effect in this example
ISB
LDR Ra, [Rb] ; this load will be executed after the timer has been read
```

or

```asm
MRC Rn, CNTPCT{SS} ; either view of the physical counter has the same effect in this example
EOR Rm, Rn, Rn
LDR Ra, [Rb] ; this load will be executed after the counter has been read
```

**G6.2.2 The virtual counter**

An implementation of the Generic Timer always includes a virtual counter, that indicates virtual time.

The virtual counter contains the value of the physical counter minus a 64-bit virtual offset. When executing in a Non-secure EL1 or EL0 mode, the virtual offset value relates to the current virtual machine.

The CNTVOFF register contains the virtual offset, see The virtual offset register on page G6-9295.

The CNTVCT register holds the current virtual counter value.

Reads of CNTVCT can occur speculatively and out of order relative to other instructions executed on the same PE.
The self-synchronized view of the virtual counter

When FEAT_ECV is implemented, an alternative way to read the virtual counter is supported. The CNTVCTSS register is a non-speculative view of the virtual counter, as seen from the Exception level that CNTVCTSS is read from.

Accesses to the CNTVCTSS are subject to the same traps as accesses to the CNTVCT.

Reads of CNTVCT occur in program order relative to reads of CNTVCT or CNTVCTSS.

Reads of CNTVCTSS occur in program order relative to reads of CNTVCT or CNTVCTSS.

Example G6-3 Ensuring reads of virtual counter occur after signal read from memory

If a read from memory is used to obtain a signal from another agent that indicates that CNTVCT must be read, an ISB is used to ensure that the read of CNTVCT occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop ; polling for some communication to indicate a requirement to read the timer
  LDR R1, [R2] ; earlier loads and stores
  CMP R1, #1 ; has had the value 1 written to it
  BNE loop
  ISB ; without this the CNTVCT could be read before the memory location in [R2]
  MRC R1, CNTVCT ; and stores
```

When FEAT_ECV is implemented, an access to CNTVCTSS can be used in place of the CNTVCT, which, because it cannot be accessed speculatively, allows the ISB to be removed. This means that the following code sequence can be used:

```
loop ; polling for some communication to indicate a requirement to read the timer
  LDR R1, [R2] ; earlier loads and stores
  CMP R1, #1 ; has had the value 1 written to it
  BNE loop
  MRC R1, CNTVCTSS
```

Similarly where a read of the virtual counter is required to take place after the completion of all loads and stores appearing in program order before the read of the counter, then the following two sequences can be used:

```
... ; earlier loads and stores
DSB ; completes earlier loads and stores
ISB ; without this the CNTVCT could be read before the completion of the earlier loads
MRC R1, CNTVCT ; and stores
```

Or, if FEAT_ECV is implemented:

```
... ; earlier loads and stores
DSB ; completes earlier loads and stores
MRC R1, CNTVCTSS
```

 Neither view of the virtual counter ensures that:

• Context changes occurring in program order before the read of the counter have been synchronized.

• Accesses to memory appearing in program order after the read of the counter are executed before the counter has been read.

Where there is a Dependency through registers dependency from the read of the virtual counter to a Register effect generated by a read or write, the read or write will be executed after the read of the counter.
Example G6-4 Ensuring reads of virtual counter occur after previous memory accesses

To ensure that all previous memory accesses have completed and all previous context changes have been synchronized before the read of the counter, one of the following sequences should be used:

```
DSB
ISB
MRC Rn, CNTVCT{SS} ; either view of the virtual counter has the same effect in this example
```

or

```
DSB
ISB
CBZ Ra, next
next
ISB ; this ISB is not needed if the MRS is accessing CNTVCT{SS}
MRC Rn, CNTVCT{SS}
```

To ensure that a memory access occurs only after a read of the counter, then either of the following sequences should be used:

```
MRC Rn, CNTVCT{SS} ; either view of the virtual counter has the same effect in this example
ISB
LDR Ra, [Rb] ; this load will be executed after the timer has been read
```

or

```
MRC Rn, CNTVCT{SS} ; either view of the virtual counter has the same effect in this example
EOR Rm, Rn, Rn
LDR Ra, [Rb] ; this load will be executed after the counter has been read
```

The virtual offset register

The virtual counter is a counter that has a virtual offset relative to the physical counter as viewed from EL2 and EL3. This virtual offset is held in the register CNTVOFF. The virtual counter value is the count compared by the EL1 virtual timer.

If EL2 is not implemented and enabled, then the virtual counter uses a fixed virtual offset of zero.

G6.2.3 Event streams

Any implementation of the Generic Timer can use the system counter to generate one or more event streams, to generate periodic wake-up events as part of the mechanism described in Wait for Event on page D1-4676.

Note

An event stream might be used:
- To impose a time-out on a Wait For Event polling loop.
- To safeguard against any programming error that means an expected event is not generated.

The CNTHCTL.{EVNTEN, EVNTDIR, EVNTI, EVNTIS} fields define an event stream that is generated from the virtual counter.

In all implementations the CNTHCTL.{EVNTEN, EVNTDIR, EVNTI, EVNTIS} fields define an event stream that is generated from the physical counter.

The event stream is configured as follows:
- EVNTI selects the counter bit that triggers the event.
- If FEAT_ECV is not implemented, EVNTI selects between bits[0:15].
- If FEAT_ECV is implemented, EVNTIS selects whether ENVTI selects between bits[0:15] or bits[8:23].
• EVNTDIR selects whether the event is generated on each 0 to 1 transition, or each 1 to 0 transition, of the selected counter bit.

--- Note ---
If the event stream is configured to produce events from the low order bits of the counter when the counter frequency is very high (for example 1GHz), then the practical update rate of the counter might mean that the event stream is not generated as the low order bit might not change. Software can rely on an event stream rate of at least 1MHz in normal operation.

The pseudocode descriptions of the operation of an event stream are SetEventRegister, TestEventCNTV, and TestEventCNTP.

G6.2.4 Timers

In an implementation of the Generic Timer that includes EL3 the following timers are accessible from AArch32 state, provided the appropriate Exception level can use AArch32:
• A Non-secure EL1 physical timer. A Non-secure EL1 control determines whether this register is accessible from Non-secure EL0.
• A Secure PL1 physical timer. This timer is accessible from EL3 when EL3 is using AArch32.

--- Note ---
When EL3 is using AArch64, the AArch32 EL1 timers are not banked between Secure and Non-secure state.

A Secure PL1 control determines whether this register is accessible from Secure EL0.
• A Non-secure EL2 physical timer, accessible from Non-secure EL2, or EL3 when SCR.NS is set to 1.
• An EL1 virtual timer.
• When FEAT_VHE is implemented, a Non-secure EL2 virtual timer.
• When FEAT_SEL2 is implemented, a Secure EL2 physical timer.
• When FEAT_SEL2 is implemented, a Secure EL2 virtual timer.

--- Note ---
The Secure EL2 timers are accessible in AArch32 state if using EL0, when EL0 is using AArch32 state, Secure EL2 is using AArch64, and HCR_EL2.{E2H,TGE} == {1, 1}.

The output of each implemented timer:
• Provides an output signal to the system.
• If the PE interfaces to a Generic Interrupt Controller (GIC), signals a Private Peripheral Interrupt (PPI) to that GIC. In a multiprocessor implementation, each PE must use the same interrupt number for each timer.

Each timer:
• Is based around a 64-bit CompareValue that provides a 64-bit unsigned upcounter.
• Provides an alternative view of the CompareValue, called the TimerValue, that appears to operate as a 32-bit downcounter.
• Has, in addition, a 32-bit Control register.
In all implementations, the AArch32 System registers for the EL1 (or PL1) physical timer are banked, to provide the Secure and Non-secure implementations of the timer. Table G6-1 on page G6-9297 shows the physical timer registers and Table G6-2 on page G6-9297 show the virtual timer registers.

### Table G6-1 Physical timer registers summary for the Generic Timer

<table>
<thead>
<tr>
<th>Timer register</th>
<th>Secure PL1 or Non-secure EL1 physical timer</th>
<th>Non-secure EL2 physical timer</th>
<th>Secure EL2 physical timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV</td>
<td>CNTP_CV AL</td>
<td>CNTHP_CV AL</td>
<td>CNTHPS_CV AL</td>
</tr>
<tr>
<td>TV</td>
<td>CNTP_TV AL</td>
<td>CNTHP_TV AL</td>
<td>CNTHPS_TV AL</td>
</tr>
<tr>
<td>Control</td>
<td>CNTP_CTL</td>
<td>CNTHP_CTL</td>
<td>CNTHPS_CTL</td>
</tr>
</tbody>
</table>

a. In this column, CV indicates the CompareValue register, and TV indicates the TimerValue register.

b. Only present when the implementation implements FEAT_SEL2.

c. In AArch32 state, these registers are banked to provide the Non-secure EL1 physical timer and the Secure PL1 physical timer.

### Table G6-2 Virtual timer register summary for the Generic Timer

<table>
<thead>
<tr>
<th>Timer register</th>
<th>EL1 virtual timer</th>
<th>Non-secure EL2 virtual timer</th>
<th>Secure EL2 virtual timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV</td>
<td>CNTV_CV AL</td>
<td>CNTHV_CV AL</td>
<td>CNTHVS_CV AL</td>
</tr>
<tr>
<td>TV</td>
<td>CNTV_TV AL</td>
<td>CNTHV_TV AL</td>
<td>CNTHVS_TV AL</td>
</tr>
<tr>
<td>Control</td>
<td>CNTV_CTL</td>
<td>CNTHV_CTL</td>
<td>CNTHVS_CTL</td>
</tr>
</tbody>
</table>

a. In this column, CV indicates the CompareValue register, and TV indicates the TimerValue register.

b. Only when the implementation includes FEAT_VHE.

c. Only present when the implementation includes FEAT_SEL2.

### Operation of the CompareValue views of the timers

The CompareValue view of a timer operates as a 64-bit upcounter. The timer condition is met when the appropriate counter reaches the value programmed into its CompareValue register. When the timer condition is met an interrupt is generated if the interrupt is not masked in the corresponding timer control register, CNTP_CTL, CNTHP_CTL, CNTHPS_CTL, CNTV_CTL, CNTHV_CTL, or CNTHVS_CTL. For CNTP_CTL, the interrupt is the same as the interrupt asserted by the Non-secure instance of the AArch64 register CNTP_CTL_EL0.

The operation of this view of a timer is:

\[
\text{TimerConditionMet} = (((\text{Counter}[63:0] - \text{Offset}[63:0])[63:0] - \text{CompareValue}[63:0]) >= 0)
\]

Where:

- **TimerConditionMet** Is TRUE if the timer condition for this counter is met, and FALSE otherwise.
- **Counter** The physical counter value, that can be read from the CNTPCT register.
- **Offset** For the EL1 physical timer, if ID_AA64MMFR0_EL1.ECV is 0b10, EL2 is using AArch64 and is implemented and enabled in the current Security state, and CNTHCTL_EL2.ECV is 0b1, then the offset value is held in the CNTPOFF_EL2. Otherwise the offset value for the EL1 physical timer is zero.
- **CompareValue** The value of the appropriate CompareValue register, CNTP_CV AL, CNTHP_CV AL, CNTHPS_CV AL, CNTV_CV AL, CNTHV_CV AL, or CNTHVS_CV AL.
In this view of a timer, Counter, Offset, and CompareValue are all 64-bit unsigned values.

--- Note ---

This means that a timer with a CompareValue of, or close to, \(0x\text{FFFFFF_FFFF_FFFF}\) might never meet its timer condition. However, there is no practical requirement to use values close to the counter wrap value.

Software can observe the counter value by the offset in some situations by reading CNTVCT.

--- Operation of the TimerValue views of the timers ---

The TimerValue view of a timer appears to operate as a signed 32-bit downcounter. A TimerValue register is programmed with a count value. This value decrements on each increment of the appropriate counter, and the timer condition is met when the value reaches zero. When the timer condition is met, an interrupt is generated if the interrupt is not masked in the corresponding timer control register, \(\text{CNTP}_\text{CTL}, \text{CNTHP}_\text{CTL}, \text{CNTHPS}_\text{CTL}, \text{CNTV}_\text{CTL}, \text{CNTHV}_\text{CTL}, \text{or CNTHVS}_\text{CTL}\).

This view of a timer depends on the following behavior of accesses to TimerValue registers:

\begin{align*}
\text{Reads} & \quad \text{TimerValue} = (\text{CompareValue} - (\text{Counter} - \text{Offset}))[31:0] \\
\text{Writes} & \quad \text{CompareValue} = ((\text{Counter} - \text{Offset})[63:0] + \text{SignExtend}(\text{TimerValue}))[63:0]
\end{align*}

Where the arguments other than \(\text{TimerValue}\) have the definitions used in \textit{Operation of the CompareValue views of the timers} on page G6-9297, and in addition:

\(\text{TimerValue}\) The value of a TimerValue register, \(\text{CNTP}_\text{TV} \_\text{AL}, \text{CNTHP}_\text{TV} \_\text{AL}, \text{CNTHPS}_\text{TV} \_\text{AL}, \text{CNTV}_\text{TV} \_\text{AL}, \text{CNTHV}_\text{TV} \_\text{AL}, \text{or CNTHVS}_\text{TV} \_\text{AL}\).

In this view of a timer, values are signed, in standard two's complement form.

A read of a TimerValue register after the timer condition has been met indicates the time since the timer condition was met.

--- Note ---

\begin{itemize}
\item \textit{Operation of the CompareValue views of the timers} on page G6-9297 gives a strict definition of \(\text{TimerConditionMet}\). However, provided that the TimerValue is not expected to wrap as a 32-bit signed value when decremented from \(0x80000000\), the TimerValue view can be used as giving an effect equivalent to:
\[\text{TimerConditionMet} = (\text{TimerValue} \leq 0)\]

\item Programming TimerValue to a negative number with magnitude greater than \((\text{Counter} - \text{Offset})\) can lead to an arithmetic overflow that causes the CompareValue to be an extremely large positive value. This potentially delays meeting the timer condition for an extremely long period of time.
\end{itemize}
Chapter G7
AArch32 System Register Encoding

This chapter describes the AArch32 System register encoding space. It contains the following sections:

• The AArch32 System register encoding space on page G7-9300.
• AArch32 VMSA organization of registers in the (coproc==0b1110) encoding space on page G7-9301.
• AArch32 VMSA organization of registers in the (coproc==0b1111) encoding space on page G7-9304.
G7.1 The AArch32 System register encoding space

The T32 and A32 instruction sets include instructions that access the System register encoding space. These instructions provide:

- Access to System registers, including the debug registers, that provide system control, and system status information.
- The cache, branch predictor, and TLB maintenance instructions, and address translation instructions.

The AArch32 System register interface on page G1-8993 describes the instructions that provide access to these registers and instructions. Chapter G8 AArch32 System Register Descriptions describes these registers and encodings.

When accessing 32-bit registers, or executing these instructions, entries in the encoding space are characterized by the parameter set \{coproc, CRn, opc1, CRm, opc2\}. This encoding space is defined only for the coproc values 0b1110 and 0b1111.

Note

- When accessing 64-bit registers entries in the encoding space are characterized by the parameter set \{coproc, CRn, opc1\}, for the coproc values 0b1110 and 0b1111. A CRn value in this parameter set is equivalent to a CRn value in the parameter set for accessing 32-bit registers.
- Background to the System register interface on page G1-8994 gives more information about this encoding model.

The following describe this encoding space:

- AArch32 VMSA organization of registers in the (coproc==0b1110) encoding space on page G7-9301.
- AArch32 VMSA organization of registers in the (coproc==0b1111) encoding space on page G7-9304.
G7.2 AArch32 VMSA organization of registers in the (coproc==0b1110) encoding space

The System registers in the (coproc==0b1110) encoding space provide a number of distinct control functions, covering:

- Debug.
- Trace.
- Execution environment control, for identification of the trivial Jazelle implementation.

Because these functions are distinct, the descriptions of these registers are distributed, as follows:

- In this manual, Debug registers on page G8-9831 describes the Debug registers.
- The Embedded Trace Macrocell Architecture Specification describes the Trace registers.

This section summarizes the allocation of the System registers in the (coproc==0b1110) encoding space between these different functions, and the register encodings in this space that are reserved.

The 32-bit System register encodings are classified by the \{opc1, CRn, opc2, CRm\} values required to access them using an MCR or an MRC instruction. The 64-bit System register encodings are classified by the \{opc1, CRm\} values required to access them using an MCRR or an MRRC instruction. For the registers in the (coproc==0b1110) encoding space, the opc1 value determines the primary allocation of these registers, as follows:

- **opc1==0** Debug registers.
- **opc1==1** Trace registers.
- **opc1==7** Jazelle registers. Jazelle registers are implemented as required for a trivial Jazelle implementation.

**Other opc1 values**

Reserved.

--- Note ---

Primary allocation of (coproc==0b1110) register function by opc1 value differs from the allocation of (coproc==0b1111) registers, where primary allocation is by CRn value for 32-bit register accesses, or CRn value for 64-bit register accesses.

---

For the Debug and Jazelle registers, Table G7-1 on page G7-9302 defines:

- The \{opc1, CRn, opc2, CRm\} values used for accessing the 32-bit registers using the MRC and MCR instructions.
- The \{opc1, CRm\} values used for accessing the 64-bit register using the MRRC instruction.

Some Debug registers can also be accessed using the LDC and STC instructions. Table G7-1 on page G7-9302 defines the CRn values used for accessing the registers using these instructions.

--- Note ---

The only permitted uses of the LDC and STC instructions are:

- An LDC access to load data from memory to DBGDTRTXint.
- An STC access to store data to memory from DBGDTRRXint.

In the LDC and STC syntax descriptions in this Manual, the required coproc value of p14 and CRn value of c5 are given explicitly.
G7.2.1 Register access instruction arguments, \((\text{coproc}==\text{0b1110})\) registers

Table G7-1 on page G7-9302 shows the MCR, MRC, and MRRC instruction arguments required for accesses to each register that can be visible in the System register interface in the \((\text{coproc}==\text{0b1110})\) encoding.

**Table G7-1 Mapping of \((\text{coproc}==\text{0b1110})\) MCR, MRC, and MRRC instruction arguments to System registers**

<table>
<thead>
<tr>
<th>Name</th>
<th>opc1</th>
<th>CRn</th>
<th>opc2</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBGDIDRa</td>
<td>0</td>
<td>c0</td>
<td>0</td>
<td>c0</td>
</tr>
<tr>
<td>DBGDSCRint</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGDCCINT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGDTRRXint</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGDTRTXint</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGVCRA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGDTRRXext</td>
<td>2</td>
<td>c0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGDSCRext</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGDTRTXext</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGOSECCR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGGBVR&lt;n&gt;</td>
<td>4</td>
<td>c0-15b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGBCR&lt;n&gt;</td>
<td>5</td>
<td>c0-15b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGGWVR&lt;n&gt;</td>
<td>6</td>
<td>c0-15b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGWCR&lt;n&gt;</td>
<td>7</td>
<td>c0-15b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGDAR 32 bits wide</td>
<td>c1</td>
<td>0</td>
<td>c0</td>
<td></td>
</tr>
<tr>
<td>DBGDAR 64 bits wide</td>
<td>-</td>
<td>-</td>
<td>c1</td>
<td></td>
</tr>
<tr>
<td>DBGBXVR&lt;n&gt;</td>
<td>e1</td>
<td>1</td>
<td>c0-15b</td>
<td></td>
</tr>
<tr>
<td>DBGOSLAR</td>
<td>4</td>
<td>c0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGOSLSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGOSDLR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGPRCR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGDSAR 32 bits wide</td>
<td>c2</td>
<td>0</td>
<td>c0</td>
<td></td>
</tr>
<tr>
<td>DBGDSAR 64 bits wide</td>
<td>-</td>
<td>-</td>
<td>c2</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>e4</td>
<td>0-3</td>
<td>c0-15</td>
<td></td>
</tr>
</tbody>
</table>
Table G7-1 Mapping of \((\text{coproc}==0b1110)\) MCR, MRC, and MRRC instruction arguments to System registers (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>opc1</th>
<th>CRn</th>
<th>opc2</th>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBGCLAIMSET</td>
<td>0</td>
<td>c7</td>
<td>6</td>
<td>c8</td>
</tr>
<tr>
<td>DBGCLAIMCLR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGAUTHSTATUS</td>
<td></td>
<td>c14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBGDEVID2</td>
<td></td>
<td>7</td>
<td></td>
<td>c0</td>
</tr>
<tr>
<td>DBGDEVID1</td>
<td></td>
<td></td>
<td></td>
<td>c1</td>
</tr>
<tr>
<td>DBGDEVID</td>
<td></td>
<td></td>
<td></td>
<td>c2</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>c0-c7</td>
<td>0-7</td>
<td>c0-c15</td>
</tr>
<tr>
<td>JIDR(^c)</td>
<td></td>
<td>7</td>
<td></td>
<td>c0</td>
</tr>
<tr>
<td>JOSCR(^c)</td>
<td></td>
<td></td>
<td></td>
<td>c1</td>
</tr>
<tr>
<td>JMCR(^c)</td>
<td></td>
<td></td>
<td></td>
<td>c2</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td>All other encodings</td>
</tr>
</tbody>
</table>

- a. If EL1 cannot use AArch32, this register is OPTIONAL and deprecated. See the register description for details.
- b. Accesses to not implemented breakpoint and watchpoint register access instructions are UNDEFINED. If EL2 is not implemented or breakpoint \(n\) is not context-aware, DBGBXVR\(<n>\) is unallocated. CRm encodes \(<n>\), the breakpoint or watchpoint number.
- c. Legacy register.

Table G7-2 shows the LDC and STC instruction arguments required for accesses to the Debug registers that can be accessed using these instructions.

Table G7-2 Mapping of LDC and STC instruction arguments to System registers

<table>
<thead>
<tr>
<th>Name</th>
<th>CRn</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBGDTRTXint</td>
<td>c5</td>
<td>LDC</td>
<td>Debug Data Transfer Register, Transmit, Internal View</td>
</tr>
<tr>
<td>DBGDTRRXint</td>
<td>c5</td>
<td>STC</td>
<td>Debug Data Transfer Register, Receive, Internal View</td>
</tr>
</tbody>
</table>

**Note**

In the instruction syntax descriptions for the LDC and STC instructions, the required coproc and CRn values are given explicitly as coproc==p14, CRn==c5.
G7.3 AArch32 VMSA organization of registers in the (coproc==0b1111) encoding space

For 32-bit accesses to the System registers in the (coproc==0b1111) encoding space, the ordered set of parameters \( \{CRn, opc1, CRm, opc2\} \) determine the register order. Within this ordering, the \( CRn \) value originally provided a functional grouping of these registers. As the number of System registers has increased this ordering has become less appropriate.

This document now:

- Groups the System registers in the (coproc==0b1111) encoding space by functional group, see Functional index of AArch32 registers and System instructions on page K17-11899.
- Describes all of the System registers for the AArch32 VMSA, in Chapter G8 AArch32 System Register Descriptions.
- Gives additional information about the organization of the AArch32 VMSA System registers in the (coproc==0b1111) encoding space, in the remainder of this section.

Note

Not all System registers introduced by architectural extensions are described in Chapter G8 AArch32 System Register Descriptions. For information about the System registers introduced by architectural extensions, see Chapter A2 Armv8-A Architecture Extensions.

This section presents information about the register ordering by \( \{CRn, opc1, CRm, opc2\} \). It contains the following subsections:

- System register summary for (coproc==0b1111) encodings by CRn value on page G7-9305.
- Full list of AArch32 VMSA System registers in the (coproc==0b1111) encoding space on page G7-9307.

Note

The ordered listing of (coproc==0b1111) registers by the \( \{CRn, opc1, CRm, opc2\} \) encoding of the 32-bit registers is most likely to be useful to those implementing AArch32 state, and to those validating such implementations. However, otherwise, the grouping of registers by function is more logical.

In addition, the indexes in Appendix K17 Registers Index include all of the System registers.
### G7.3.1 System register summary for (coproc==0b1111) encodings by CRn value

Figure G7-1 summarizes the grouping of the System registers in the (coproc==0b1111) encoding space, for an AArch32 VMSA implementation, by the value of CRn used for a 32-bit access to the register.

<table>
<thead>
<tr>
<th>CRn</th>
<th>opc1</th>
<th>CRm</th>
<th>opc2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0</td>
<td>0-2, 4</td>
<td>(c0-c2)</td>
<td>0-7</td>
<td>ID registers</td>
</tr>
<tr>
<td>c1</td>
<td>0, 4</td>
<td>(c0, c1)</td>
<td>0-7</td>
<td>System control registers</td>
</tr>
<tr>
<td>c2</td>
<td>0, 4</td>
<td>(c0, c1)</td>
<td>0-2</td>
<td>Memory system control registers</td>
</tr>
<tr>
<td>c3</td>
<td>0</td>
<td>c0</td>
<td>0</td>
<td>Memory system fault registers</td>
</tr>
<tr>
<td>c4</td>
<td>0</td>
<td>c6</td>
<td>0</td>
<td>GIC System register *, Debug exception registers</td>
</tr>
<tr>
<td>c5</td>
<td>0, 4</td>
<td>(c0, c1)</td>
<td>0-1</td>
<td>Cache maintenance, address translations, legacy operations</td>
</tr>
<tr>
<td>c6</td>
<td>0, 4</td>
<td>c0</td>
<td>0, 2, 4</td>
<td>Memory mapping registers and TLB operations</td>
</tr>
<tr>
<td>c7</td>
<td>0, 4</td>
<td>Various</td>
<td>Various</td>
<td>Performance monitors</td>
</tr>
<tr>
<td>c8</td>
<td>0-7</td>
<td>Various</td>
<td>Various</td>
<td>System control registers, GIC System registers *</td>
</tr>
<tr>
<td>c9</td>
<td>0-7</td>
<td>Various</td>
<td>0-7</td>
<td>Reserved for DMA operations for TCM access</td>
</tr>
<tr>
<td>c10</td>
<td>0-7</td>
<td>Various</td>
<td>0-7</td>
<td>Process, Context, Thread ID registers, Activity Monitors registers *</td>
</tr>
<tr>
<td>c11</td>
<td>0-7</td>
<td>(c0-c8, c15)</td>
<td>0-7</td>
<td>Generic Timer registers *, Performance Monitors registers *</td>
</tr>
<tr>
<td>c12</td>
<td>0-2, 4, 6</td>
<td>Various</td>
<td>0-1</td>
<td>IMPLEMENTATION DEFINED registers</td>
</tr>
<tr>
<td>c13</td>
<td>0, 4</td>
<td>Various</td>
<td>Various</td>
<td></td>
</tr>
<tr>
<td>c14</td>
<td>0-7</td>
<td>(c0-c15)</td>
<td>0-7</td>
<td></td>
</tr>
<tr>
<td>c15</td>
<td>0-7</td>
<td>(c0-c15)</td>
<td>0-7</td>
<td></td>
</tr>
</tbody>
</table>

#### Note

For the System registers in the (coproc==0b1111) encoding space, Figure G7-1 gives only an overview of the assigned encodings for 32-bit registers for each of the CRn values c0-c15. For more information, see:

- The full list of registers in the (coproc==0b1111) encoding space, in *Full list of AArch32 VMSA System registers in the (coproc==0b1111) encoding space* on page G7-9307, for the definition of the assigned and unassigned encodings for that register.
- The register definitions in *Chapter G8 AArch32 System Register Descriptions* for any dependencies on the implemented Exception levels.

In general, System register accesses using an unallocated set of {CRn, opc1, CRm, opc2} values are UNDEFINED. *Behavior of AArch32 VMSA System registers with (coproc==0b1111, CRn==c0)* on page G7-9306 described the only exceptions to this rule.

The 32-bit System registers with (coproc==0b1111, CRn==c15), and the corresponding 64-bit System registers, are reserved for IMPLEMENTATION DEFINED registers. For more information, see *Reserved encodings in the AArch32 VMSA System register (coproc==0b1111) space* on page G7-9306.

### The HSTR.Tn trap on (coproc==0b1111) System registers

As *General trapping to Hyp mode of Non-secure EL0 and EL1 accesses to System registers in the (coproc==0b1111) encoding space* on page G1-9024 describes, when the value of HSTR.Tn is 1, Non-secure PL1 accesses to System registers in the (coproc==0b1111) encoding space using a CRn or CRm value that corresponds to the value of Tn are trapped to EL2, even if the encoding is UNDEFINED when the value of HSTR.Tn is 0. This applies:

- For 32 bit register accesses when the value of Rn in the MCR or MRC instruction corresponds to Tn.
- For 64 bit register accesses when the value of Rn in the MCRR or MRRC instruction corresponds to Tn.

If there are matching System register encodings that are accessible from Non-secure EL0 then those accesses are also trapped to EL2 when the value of HSTR.Tn is 1.
Behavior of AArch32 VMSA System registers with \((\text{coproc}==0b1111, \text{CRn}==c0)\)

In the \((\text{coproc}==0b1111)\) encoding space, the 32-bit System registers with \((\text{CRn}==c0)\) provide device and feature identification.

Table G7-3 on page G7-9308 shows all of the architecturally required System registers with \{(\text{coproc}==0b1111, \text{CRn}==c0)\}. The behavior of 32-bit System register encodings in this group that are not shown in the table, and encodings that are part of an unimplemented Exception level, depends on the value of \(\text{opc1}\), and possibly on the value of \(\text{CRm}\) and \(\text{opc2}\), as follows:

\[
\begin{align*}
\text{opc1} &= 0 & \text{All write accesses to the encodings are UNDEFINED.} \\
& & \text{For read accesses:} \\
& & \quad \text{The following encodings return an UNKNOWN value:} \\
& & \quad \quad \text{CRn}==3, \text{opc2}==\{0, 1, 2\}. \\
& & \quad \quad \text{CRn}==\{4, 6, 7\}, \text{opc2}==\{0, 1\}. \\
& & \quad \quad \text{CRn}==5, \text{opc2}==\{0, 1, 4, 5\}. \\
& & \quad \quad \text{All other encodings are RES0.} \\
\text{opc1} &> 0 & \text{All accesses to the encodings are UNDEFINED.}
\end{align*}
\]

See also \text{Accesses to unallocated encodings in the (\text{coproc}==0b111x) encoding space on page G8-9324.} 

\[\text{Note}\]

Some of these registers were previously described as being part of the CPUID identification scheme, see \text{The CPUID identification scheme on page G8-9323.}

Reserved encodings in the AArch32 VMSA System register \((\text{coproc}==0b1111)\) space

AArch32 state reserves a number of regions in the \((\text{coproc}==0b1111)\) encoding space for IMPLEMENTATION DEFINED System registers. These reservations are defined in terms of the encoding of 32-bit accesses to the System register encoding space. That is, they are defined by the reserved 32-bit \{\text{CRn}, \text{opc1}, \text{CRm}, \text{opc2}\} encodings.

Reserved encodings that do not have an IMPLEMENTATION DEFINED function are UNDEFINED.

The following subsections give more information about these reserved encodings:

\[
\begin{align*}
\text{Reserved 32-bit encodings with } & \{\text{coproc}==0b1111, \text{CRn}==c9\}. \\
& \text{Reserved 32-bit encodings with } \{\text{coproc}==0b1111, \text{CRn}==c10\} \text{ on page G7-9307.} \\
& \text{Reserved 32-bit encodings with } \{\text{coproc}==0b1111, \text{CRn}==c11\} \text{ on page G7-9307.} \\
& \text{Reserved 32-bit encodings with } \{\text{coproc}==0b1111, \text{CRn}==c15\} \text{ on page G7-9307.}
\end{align*}
\]

\text{Reserved 32-bit encodings with } \{\text{coproc}==0b1111, \text{CRn}==c9\}

In the AArch32 encoding space, for 32-bit encodings with \{\text{coproc}==0b1111, \text{CRn}==c9\}, the following encodings are reserved for IMPLEMENTATION DEFINED purposes:

\[
\begin{align*}
\text{Encodings with } & \{\text{coproc}==0b1111, \text{CRn}==c9, \text{opc1}==\{0-7\}, \text{opc2}==\{0-7\}, \text{CRm}==\{c0-c2, c5-c8\}\} \text{ are reserved for IMPLEMENTATION DEFINED branch predictor, cache, and TCM operations.} \\
\text{Encodings with } & \{\text{coproc}==0b1111, \text{CRn}==c9, \text{opc1}==\{0-7\}, \text{opc2}==\{0-7\}, \text{CRm}==c15\} \text{ are reserved for IMPLEMENTATION DEFINED performance monitors.}
\end{align*}
\]

\[\text{Note}\]

These are distinct from the OPTIONAL Arm Performance Monitors Extension, the registers for which use the encoding space \{\text{coproc}==0b1111, \text{CRn}==c9, \text{opc1}==\{0-7\}, \text{opc2}==\{0-7\}, \text{CRm}==\{c12-c14\}\}.
Reserved 32-bit encodings with \( \{ \text{coproc} = 0b1111, \text{CRn} = c10 \} \)

In the AArch32 encoding space, for 32-bit encodings with \( \{ \text{coproc} = 0b1111, \text{CRn} = c10 \} \), the following encodings are reserved for IMPLEMENTATION DEFINED purposes:

- Encodings with \( \{ \text{coproc} = 0b1111, \text{CRn} = c10, \text{opc} = \{0-7\}, \text{CRm} = \{c0, c1, c4, c8\} \} \) are reserved for IMPLEMENTATION DEFINED TLB lockdown operations.

Reserved 32-bit encodings with \( \{ \text{coproc} = 0b1111, \text{CRn} = c11 \} \)

In the AArch32 encoding space, for 32-bit encodings with \( \{ \text{coproc} = 0b1111, \text{CRn} = c11 \} \), the following encodings are reserved for IMPLEMENTATION DEFINED purposes:

- Encodings with \( \{ \text{coproc} = 0b1111, \text{CRn} = c11, \text{opc} = \{0-7\}, \text{CRm} = \{c0-c8, c15\} \} \) are reserved for IMPLEMENTATION DEFINED DMA operations for TCM access.

The remainder of the AArch32 \( \{ \text{coproc} = 0b1111, \text{CRn} = c11 \} \) encoding space is UNDEFINED.

Reserved 32-bit encodings with \( \{ \text{coproc} = 0b1111, \text{CRn} = c15 \} \)

The AArch32 System register encodings are reserved with \( \{ \text{coproc} = 0b1111, \text{CRn} = c15 \} \) for IMPLEMENTATION DEFINED purposes, and there are no restrictions on the use of these encodings. The documentation of the Arm implementation must describe fully any registers implemented in the \( \{ \text{coproc} = 0b1111, \text{CRn} = c15 \} \) encoding space. Normally, for processor implementations by Arm, this information is included in the Technical Reference Manual for the processor.

Typically, an implementation uses the \( \{ \text{coproc} = 0b1111, \text{CRn} = c15 \} \) encodings to provide test features, and any required configuration options that are not covered by this Manual.

This reservation means that the AArch32 64-bit encodings with \( \{ \text{coproc} = 0b1111, \text{CRn} = c15 \} \) are also reserved for IMPLEMENTATION DEFINED purposes, without any restrictions on the use of these encodings.

### G7.3.2 Full list of AArch32 VMSA System registers in the \( \{ \text{coproc} = 0b1111 \} \) encoding space

Table G7-3 on page G7-9308 shows the System registers in the \( \{ \text{coproc} = 0b1111 \} \) encoding space in the AArch32 VMSA, in the order of the \( \{ \text{CRn}, \text{opc1}, \text{CRm}, \text{opc2} \} \) parameter values used in \text{MCR} or \text{MRC} accesses to the 32-bit registers:

- For \text{MCR} or \text{MRC} accesses to the 32-bit registers, \text{CRn} is the primary identifier of the target System register for the access. This applies, also, to \text{MCR} or \text{MRC} instructions that provide 32-bit accesses to a single word of a 64-bit System register.

- For \text{MCRR} or \text{MRRC} accesses to the 64-bit registers, \text{CRm} is the primary identifier of the target System register for the access. Table G7-3 on page G7-9308 orders the 64-bit registers with the 32-bit registers accessed using the same primary register identifier. For example, the 64-bit encoding of \text{TTBR0}, that is accessed with (\text{CRn} = c2), is listed with the 32-bit registers that are accessed with (\text{CRn} = c2).
Table G7-3 AArch32 VMSA (coproc==0b1111) register summary, in MCR/MRC parameter order

<table>
<thead>
<tr>
<th>Name</th>
<th>CRn</th>
<th>opc1</th>
<th>CRm</th>
<th>opc2</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIDR</td>
<td>c0</td>
<td>0</td>
<td>c0</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>CTR</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>TCMTR</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>v8.0</td>
</tr>
<tr>
<td>TLBTR</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>v8.0</td>
</tr>
<tr>
<td>MIDR</td>
<td></td>
<td></td>
<td></td>
<td>4, 6, 7</td>
<td>v8.0</td>
</tr>
<tr>
<td>MPIDR</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>v8.0</td>
</tr>
<tr>
<td>REVIDR</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_PFR0</td>
<td></td>
<td></td>
<td></td>
<td>c1</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_PFR1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_DFR0</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_AFR0</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_MMFR0</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_MMFR1</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_MMFR2</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_MMFR3</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_ISAR0</td>
<td></td>
<td></td>
<td></td>
<td>c2</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_ISAR1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_ISAR2</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_ISAR3</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_ISAR4</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_ISAR5</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_MMFR4</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_ISAR6</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_PFR2</td>
<td></td>
<td></td>
<td></td>
<td>c3</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_DFR1</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>v8.0</td>
</tr>
<tr>
<td>ID_MMFR5</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>v8.0</td>
</tr>
<tr>
<td>CCSIDR</td>
<td>1</td>
<td>c0</td>
<td>0</td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>CLIDR</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>v8.0</td>
</tr>
<tr>
<td>CCSIDR2</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>v8.0</td>
</tr>
<tr>
<td>AIDR</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>v8.0</td>
</tr>
</tbody>
</table>
### Table G7-3 AArch32 VMSA (coproc==0b1111) Register Summary, in MCR/MRC Parameter Order

<table>
<thead>
<tr>
<th>Name</th>
<th>CRn</th>
<th>opc1</th>
<th>CRm</th>
<th>opc2</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSSELR</td>
<td>c0</td>
<td>2</td>
<td>c0</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>VPIDR&lt;sup&gt;c&lt;/sup&gt;</td>
<td></td>
<td>4</td>
<td>c0</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>VMPIDR&lt;sup&gt;c&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>v8.0</td>
</tr>
<tr>
<td>SCTLR</td>
<td>c1</td>
<td>0</td>
<td>c0</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>ACTLR</td>
<td></td>
<td>1</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>CPACR</td>
<td></td>
<td>2</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>ACTLR&lt;sub&gt;2&lt;/sub&gt;</td>
<td></td>
<td>3</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>SCR&lt;sup&gt;d&lt;/sup&gt;</td>
<td>c1</td>
<td>0</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>SDER&lt;sup&gt;d&lt;/sup&gt;</td>
<td></td>
<td>1</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>NSACR&lt;sup&gt;d&lt;/sup&gt;</td>
<td></td>
<td>2</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TRFCR</td>
<td>c2</td>
<td>1</td>
<td></td>
<td>v8.4</td>
<td></td>
</tr>
<tr>
<td>SDCR</td>
<td>c3</td>
<td>1</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>HSCCTL&lt;sup&gt;e&lt;/sup&gt;</td>
<td>4</td>
<td>c0</td>
<td>0</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>HACTLR&lt;sup&gt;e&lt;/sup&gt;</td>
<td></td>
<td>1</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>HACTLR&lt;sub&gt;2&lt;/sub&gt;&lt;sup&gt;e&lt;/sup&gt;</td>
<td></td>
<td>3</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>HCR&lt;sup&gt;e&lt;/sup&gt;</td>
<td>c1</td>
<td>0</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>HDCR&lt;sup&gt;e&lt;/sup&gt;</td>
<td></td>
<td>1</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>HCPTR&lt;sup&gt;e&lt;/sup&gt;</td>
<td></td>
<td>2</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>HSTR&lt;sup&gt;e&lt;/sup&gt;</td>
<td></td>
<td>3</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>HCR&lt;sub&gt;2&lt;/sub&gt;&lt;sup&gt;e&lt;/sup&gt;</td>
<td></td>
<td>4</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>HACR&lt;sup&gt;e&lt;/sup&gt;</td>
<td></td>
<td>7</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>HTRFCR</td>
<td>c2</td>
<td>1</td>
<td></td>
<td>v8.4</td>
<td></td>
</tr>
<tr>
<td>TTBR0, 32 bits wide</td>
<td>c2</td>
<td>0</td>
<td>c0</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>TTBR0, 64 bits wide</td>
<td></td>
<td>0</td>
<td>c2</td>
<td>-</td>
<td>v8.0</td>
</tr>
<tr>
<td>TTBR1, 32 bits wide</td>
<td>c2</td>
<td>0</td>
<td>c0</td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>TTBR1, 64 bits wide</td>
<td>-</td>
<td>1</td>
<td>c2</td>
<td>-</td>
<td>v8.0</td>
</tr>
<tr>
<td>TTBCR</td>
<td>c2</td>
<td>0</td>
<td>c0</td>
<td>2</td>
<td>v8.0</td>
</tr>
<tr>
<td>TTBCR&lt;sub&gt;2&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>v8.2</td>
</tr>
<tr>
<td>HTCR&lt;sup&gt;e&lt;/sup&gt;</td>
<td>4</td>
<td>c0</td>
<td>2</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>VTCR&lt;sup&gt;e&lt;/sup&gt;</td>
<td>c1</td>
<td>2</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>HTTBR&lt;sup&gt;e&lt;/sup&gt;, 64 bits wide</td>
<td>-</td>
<td>4</td>
<td>c2</td>
<td>-</td>
<td>v8.0</td>
</tr>
</tbody>
</table>
### Table G7-3 AArch32 VMSA (coproc==0b1111) register summary, in MCR/MRC parameter order

<table>
<thead>
<tr>
<th>Name</th>
<th>CRn</th>
<th>opc1</th>
<th>CRm</th>
<th>opc2</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>VTIBR(^c)</td>
<td>-</td>
<td>6</td>
<td>c2</td>
<td>-</td>
<td>v8.0</td>
</tr>
<tr>
<td>DACR</td>
<td>c3</td>
<td>0</td>
<td>c0</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>ICC_PMR</td>
<td>c4</td>
<td>0</td>
<td>c6</td>
<td>0</td>
<td>GIC(^e)</td>
</tr>
<tr>
<td>ICV_PMR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSPSR(^r)</td>
<td>c4</td>
<td>3</td>
<td>c5</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>DLR</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>DFRS</td>
<td>c5</td>
<td>0</td>
<td>c0</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>IFSR</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>ADFSR</td>
<td></td>
<td></td>
<td></td>
<td>c1</td>
<td>v8.0</td>
</tr>
<tr>
<td>AIFS</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>ERRIDR</td>
<td></td>
<td></td>
<td></td>
<td>c3</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>ERRSELR</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>ERXFR</td>
<td></td>
<td></td>
<td></td>
<td>c4</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>ERXCTRLR</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>ERXSTATUS</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>ERXADDR</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>ERXFR2</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>ERXCTRLR2</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>ERXADDR2</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>ERXMISC0</td>
<td></td>
<td></td>
<td>c5</td>
<td>0</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>ERXMISC1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>ERXMISC4</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>ERXMISC5</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>ERXMISC2</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>ERXMISC3</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>ERXMISC6</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>ERXMISC7</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>HADFSR(^c)</td>
<td>4</td>
<td>c1</td>
<td>0</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>HAIFSR</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>HSR(^c)</td>
<td></td>
<td></td>
<td>c2</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>VDFS</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>RAS(^g)</td>
</tr>
<tr>
<td>Name</td>
<td>CRn</td>
<td>opc1</td>
<td>CRm</td>
<td>opc2</td>
<td>Source</td>
</tr>
<tr>
<td>--------------</td>
<td>-----</td>
<td>------</td>
<td>-----</td>
<td>------</td>
<td>--------</td>
</tr>
<tr>
<td>DFAR</td>
<td>c6</td>
<td>0</td>
<td>c0</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>IFAR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HDFAR&lt;sup&gt;c&lt;/sup&gt;</td>
<td>4</td>
<td>c0</td>
<td>0</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>HIFAR&lt;sup&gt;c&lt;/sup&gt;</td>
<td>2</td>
<td>c0</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>HPFAR&lt;sup&gt;c&lt;/sup&gt;</td>
<td>c6</td>
<td>4</td>
<td>c0</td>
<td>4</td>
<td>v8.0</td>
</tr>
<tr>
<td>ICIALLUI</td>
<td>c7</td>
<td>0</td>
<td>c1</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>BPIALLI</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>v8.0</td>
</tr>
<tr>
<td>CFPRCTX</td>
<td></td>
<td></td>
<td>c3</td>
<td>4</td>
<td>v8.0&lt;sup&gt;h&lt;/sup&gt;</td>
</tr>
<tr>
<td>DVPRTX</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>v8.0&lt;sup&gt;h&lt;/sup&gt;</td>
</tr>
<tr>
<td>CPPRTX</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>v8.0&lt;sup&gt;h&lt;/sup&gt;</td>
</tr>
<tr>
<td>PAR, 32 bits wide</td>
<td>c4</td>
<td>0</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>PAR, 64 bits wide</td>
<td>-</td>
<td>0</td>
<td>c7</td>
<td>-</td>
<td>v8.0</td>
</tr>
<tr>
<td>ICIALLU</td>
<td>c7</td>
<td>0</td>
<td>c5</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>ICIMVAAU</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>CP15ISB&lt;sup&gt;i&lt;/sup&gt;</td>
<td>4</td>
<td></td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>BPIALL</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>v8.0</td>
</tr>
<tr>
<td>BPIMV</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>v8.0</td>
</tr>
<tr>
<td>DCIMVAC</td>
<td></td>
<td></td>
<td>c6</td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>DCISW</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>v8.0</td>
</tr>
<tr>
<td>ATS1CPR</td>
<td>c8</td>
<td>0</td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>ATS1CPW</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>ATS1CUR</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>v8.0</td>
</tr>
<tr>
<td>ATS1CUW</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>v8.0</td>
</tr>
<tr>
<td>ATS12NSOPR&lt;sup&gt;d&lt;/sup&gt;</td>
<td>4</td>
<td></td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>ATS12NSOPW&lt;sup&gt;d&lt;/sup&gt;</td>
<td>5</td>
<td></td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>ATS12NSOUR&lt;sup&gt;d&lt;/sup&gt;</td>
<td>6</td>
<td></td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>ATS12NSOUW&lt;sup&gt;d&lt;/sup&gt;</td>
<td>7</td>
<td></td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>DCCMVAC</td>
<td></td>
<td></td>
<td>c10</td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>DCCSW</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>v8.0</td>
</tr>
<tr>
<td>CP15DSB&lt;sup&gt;i&lt;/sup&gt;</td>
<td>4</td>
<td></td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>CP15DMB&lt;sup&gt;i&lt;/sup&gt;</td>
<td>5</td>
<td></td>
<td></td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>CRn</td>
<td>opc1</td>
<td>CRm</td>
<td>opc2</td>
<td>Source</td>
</tr>
<tr>
<td>--------------</td>
<td>-----</td>
<td>------</td>
<td>-----</td>
<td>------</td>
<td>--------</td>
</tr>
<tr>
<td>DCCMVAU</td>
<td>c7</td>
<td>0</td>
<td>c11</td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>DCCIMVAC</td>
<td></td>
<td></td>
<td>c14</td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>DCCISW</td>
<td></td>
<td></td>
<td>2</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>ATS1HR&lt;sup&gt;c&lt;/sup&gt;</td>
<td>4</td>
<td>c8</td>
<td>0</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>ATS1HW&lt;sup&gt;c&lt;/sup&gt;</td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLBIALL</td>
<td></td>
<td>c8</td>
<td>0</td>
<td>c3</td>
<td>0</td>
</tr>
<tr>
<td>TLBIMVAIS</td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLBIASIDIS</td>
<td></td>
<td></td>
<td>2</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLBIMVAAIS</td>
<td></td>
<td></td>
<td>3</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLBIMVALIS</td>
<td></td>
<td></td>
<td>5</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLBIMVAALIS</td>
<td></td>
<td></td>
<td>7</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>ITLBIAI&lt;sup&gt;L&lt;/sup&gt;</td>
<td></td>
<td>c5</td>
<td>0</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>ITLBIMVA</td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>ITLBISID</td>
<td></td>
<td></td>
<td>2</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>DTLBIALL</td>
<td></td>
<td>c6</td>
<td>0</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>DTLBIMVA</td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>DTLBIASID</td>
<td></td>
<td></td>
<td>2</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLI&lt;sup&gt;BL&lt;/sup&gt;A&lt;sup&gt;L&lt;/sup&gt;L</td>
<td></td>
<td>c7</td>
<td>0</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLI&lt;sup&gt;BM&lt;/sup&gt;V&lt;sup&gt;MA&lt;/sup&gt;&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLI&lt;sup&gt;BI&lt;/sup&gt;A&lt;sup&gt;SID&lt;/sup&gt;</td>
<td></td>
<td></td>
<td>2</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLI&lt;sup&gt;BM&lt;/sup&gt;V&lt;sup&gt;AA&lt;/sup&gt;A</td>
<td></td>
<td></td>
<td>3</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLI&lt;sup&gt;BM&lt;/sup&gt;V&lt;sup&gt;AL&lt;/sup&gt;L</td>
<td></td>
<td></td>
<td>5</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLI&lt;sup&gt;BM&lt;/sup&gt;V&lt;sup&gt;VA&lt;/sup&gt;AL</td>
<td></td>
<td></td>
<td>7</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLI&lt;sup&gt;BI&lt;/sup&gt;P&lt;sup&gt;AS&lt;/sup&gt;2&lt;sup&gt;IS&lt;/sup&gt;</td>
<td></td>
<td>4</td>
<td>c0</td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>TLI&lt;sup&gt;BI&lt;/sup&gt;P&lt;sup&gt;AS&lt;/sup&gt;2&lt;sup&gt;L&lt;/sup&gt;IS</td>
<td></td>
<td></td>
<td>5</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLI&lt;sup&gt;BL&lt;/sup&gt;A&lt;sup&gt;LH&lt;/sup&gt;IS&lt;sup&gt;c&lt;/sup&gt;</td>
<td></td>
<td>c3</td>
<td>0</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLI&lt;sup&gt;BM&lt;/sup&gt;V&lt;sup&gt;AH&lt;/sup&gt;IS&lt;sup&gt;c&lt;/sup&gt;</td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLI&lt;sup&gt;BL&lt;/sup&gt;A&lt;sup&gt;LNS&lt;/sup&gt;NS&lt;sup&gt;NH&lt;/sup&gt;IS&lt;sup&gt;c&lt;/sup&gt;</td>
<td></td>
<td>4</td>
<td>v8.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLI&lt;sup&gt;BM&lt;/sup&gt;V&lt;sup&gt;AL&lt;/sup&gt;HIS</td>
<td></td>
<td></td>
<td>5</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLI&lt;sup&gt;BI&lt;/sup&gt;P&lt;sup&gt;AS&lt;/sup&gt;2</td>
<td></td>
<td>c4</td>
<td>1</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>TLI&lt;sup&gt;BI&lt;/sup&gt;P&lt;sup&gt;AS&lt;/sup&gt;2L</td>
<td></td>
<td></td>
<td>5</td>
<td>v8.0</td>
<td></td>
</tr>
</tbody>
</table>

Table G7-3 AArch32 VMSA (coproc==0b1111) register summary, in MCR/MRC parameter order

AArch32 System Register Encoding
G7.3 AArch32 VMSA organization of registers in the (coproc==0b1111) encoding space
## Table G7-3 AArch32 VMSA (coproc==0b1111) register summary, in MCR/MRC parameter order

<table>
<thead>
<tr>
<th>Name</th>
<th>CRn</th>
<th>opc1</th>
<th>CRm</th>
<th>opc2</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLBIALLHc</td>
<td>c8</td>
<td>4</td>
<td>c7</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>TLBIMVAHc</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>TLBIALLNSNHc</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>v8.0</td>
</tr>
<tr>
<td>TLBIMVALH</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>v8.0</td>
</tr>
<tr>
<td>Reservedd</td>
<td>c9</td>
<td>0-7</td>
<td>c0-</td>
<td>0-7</td>
<td>-</td>
</tr>
<tr>
<td>Reservedd</td>
<td></td>
<td></td>
<td>c5-</td>
<td>0-7</td>
<td>-</td>
</tr>
<tr>
<td>PMCRk</td>
<td></td>
<td>0</td>
<td>c12</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMCNTENSETk</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMCNTENCLRk</td>
<td></td>
<td>2</td>
<td></td>
<td>2</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMOVSRk</td>
<td></td>
<td>3</td>
<td></td>
<td>3</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMSWINCk</td>
<td></td>
<td>4</td>
<td></td>
<td>4</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMSELk</td>
<td></td>
<td>5</td>
<td></td>
<td>5</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMCID0k</td>
<td></td>
<td>6</td>
<td></td>
<td>6</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMCID1k</td>
<td></td>
<td>7</td>
<td></td>
<td>7</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMCCNTRk</td>
<td></td>
<td>c13</td>
<td>0</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMCCNTR_EL0k</td>
<td></td>
<td>0</td>
<td>c9</td>
<td>-</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMXEVTYPEPk</td>
<td>c9</td>
<td>0</td>
<td>c13</td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMXEVCNTRk</td>
<td></td>
<td>2</td>
<td></td>
<td>2</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMUSERENRk</td>
<td></td>
<td>c14</td>
<td>0</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMINTENSETk</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMINTENCLRk</td>
<td></td>
<td>2</td>
<td></td>
<td>2</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMOVSSSETc,k</td>
<td></td>
<td>3</td>
<td></td>
<td>3</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMCID2k</td>
<td></td>
<td>4</td>
<td></td>
<td>4</td>
<td>v8.1</td>
</tr>
<tr>
<td>PMCID3k</td>
<td></td>
<td>5</td>
<td></td>
<td>5</td>
<td>v8.1</td>
</tr>
<tr>
<td>PMMIR</td>
<td></td>
<td>6</td>
<td></td>
<td>6</td>
<td>v8.4</td>
</tr>
<tr>
<td>Reservedd</td>
<td></td>
<td>0-7</td>
<td>c15</td>
<td>0-7</td>
<td>-</td>
</tr>
</tbody>
</table>
### Table G7-3 AArch32 VMSA (coproc==0b1111) register summary, in MCR/MRC parameter order

<table>
<thead>
<tr>
<th>Name</th>
<th>CRn</th>
<th>opc1</th>
<th>CRm</th>
<th>opc2</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved&lt;sup&gt;m&lt;/sup&gt;</td>
<td>c10</td>
<td>0</td>
<td>c0- c1</td>
<td>0-7</td>
<td>-</td>
</tr>
<tr>
<td>PRRR&lt;sup&gt;n&lt;/sup&gt;</td>
<td>c2</td>
<td>0</td>
<td>v8.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAIR0&lt;sup&gt;n&lt;/sup&gt;</td>
<td>v8.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NMRR&lt;sup&gt;n&lt;/sup&gt;</td>
<td>v8.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAIR1&lt;sup&gt;n&lt;/sup&gt;</td>
<td>v8.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMAIR0</td>
<td>c3</td>
<td>0</td>
<td>v8.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMAIR1</td>
<td>1</td>
<td>v8.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved&lt;sup&gt;m&lt;/sup&gt;</td>
<td>c4, c8</td>
<td>0-7</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved&lt;sup&gt;m&lt;/sup&gt;</td>
<td>c0, c1, c4, c8</td>
<td>0-7</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved&lt;sup&gt;m&lt;/sup&gt;</td>
<td>4 c0, c1</td>
<td>0-7</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HMAIR0&lt;sup&gt;e&lt;/sup&gt;</td>
<td>c2</td>
<td>0</td>
<td>v8.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HMAIR1&lt;sup&gt;e&lt;/sup&gt;</td>
<td>1</td>
<td>v8.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HAMAIR0&lt;sup&gt;e&lt;/sup&gt;</td>
<td>c3</td>
<td>0</td>
<td>v8.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HAMAIR1&lt;sup&gt;e&lt;/sup&gt;</td>
<td>1</td>
<td>v8.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved&lt;sup&gt;m&lt;/sup&gt;</td>
<td>c4, c8</td>
<td>0-7</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved&lt;sup&gt;m&lt;/sup&gt;</td>
<td>5-7 c0, c1, c4, c8</td>
<td>0-7</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved&lt;sup&gt;e&lt;/sup&gt;</td>
<td>c11</td>
<td>0-7 c0-c8</td>
<td>0-7</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Reserved&lt;sup&gt;e&lt;/sup&gt;</td>
<td>c15</td>
<td>0-7</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_SG1IR, 64 bits wide</td>
<td>-</td>
<td>c12</td>
<td>0</td>
<td>GIC&lt;sup&gt;e&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>VBAR</td>
<td>c12</td>
<td>0</td>
<td>c0</td>
<td>v8.0</td>
<td></td>
</tr>
<tr>
<td>MVBAR&lt;sup&gt;d&lt;/sup&gt;</td>
<td>1</td>
<td>v8.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RVBAR</td>
<td>v8.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RMR&lt;sup&gt;p&lt;/sup&gt;</td>
<td>2</td>
<td>v8.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISR&lt;sup&gt;d&lt;/sup&gt;</td>
<td>c1</td>
<td>0</td>
<td>v8.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DISR</td>
<td>1</td>
<td>RAS&lt;sup&gt;g&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VDISR</td>
<td>4</td>
<td>c1</td>
<td>1 RAS&lt;sup&gt;g&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_IAR0</td>
<td>0</td>
<td>c8</td>
<td>0</td>
<td>GIC&lt;sup&gt;e&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>ICC_EOIR0</td>
<td>1</td>
<td>GIC&lt;sup&gt;e&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_HPPIR0</td>
<td>2</td>
<td>GIC&lt;sup&gt;e&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table G7-3 AArch32 VMSA (coproc==0b1111) register summary, in MCR/MRC parameter order

<table>
<thead>
<tr>
<th>Name</th>
<th>CRn</th>
<th>opc1</th>
<th>CRm</th>
<th>opc2</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICC_BPR0</td>
<td>c12</td>
<td>0</td>
<td>c8</td>
<td>3</td>
<td>GICe</td>
</tr>
<tr>
<td>ICV_BPR0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_AP0R0</td>
<td>c12</td>
<td>0</td>
<td>c8</td>
<td>4</td>
<td>GICe</td>
</tr>
<tr>
<td>ICV_AP0R0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_AP0R1</td>
<td>c12</td>
<td>0</td>
<td>c8</td>
<td>5</td>
<td>GICe</td>
</tr>
<tr>
<td>ICV_AP0R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_AP0R2</td>
<td>c12</td>
<td>0</td>
<td>c8</td>
<td>6</td>
<td>GICe</td>
</tr>
<tr>
<td>ICV_AP0R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_AP0R3</td>
<td>c12</td>
<td>0</td>
<td>c8</td>
<td>7</td>
<td>GICe</td>
</tr>
<tr>
<td>ICV_AP0R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_AP1R0</td>
<td>c9</td>
<td>0</td>
<td>c8</td>
<td>1</td>
<td>GICe</td>
</tr>
<tr>
<td>ICV_AP1R0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_AP1R1</td>
<td>c9</td>
<td>0</td>
<td>c8</td>
<td>2</td>
<td>GICe</td>
</tr>
<tr>
<td>ICV_AP1R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_AP1R2</td>
<td>c9</td>
<td>0</td>
<td>c8</td>
<td>3</td>
<td>GICe</td>
</tr>
<tr>
<td>ICV_AP1R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_AP1R3</td>
<td>c9</td>
<td>0</td>
<td>c8</td>
<td>4</td>
<td>GICe</td>
</tr>
<tr>
<td>ICV_AP1R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_DIR</td>
<td>c11</td>
<td>1</td>
<td>c8</td>
<td>1</td>
<td>GICe</td>
</tr>
<tr>
<td>ICV_DIR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_RPR</td>
<td>c12</td>
<td>0</td>
<td>c8</td>
<td>2</td>
<td>GICe</td>
</tr>
<tr>
<td>ICV_RPR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_IAR1</td>
<td>c12</td>
<td>0</td>
<td>c8</td>
<td>3</td>
<td>GICe</td>
</tr>
<tr>
<td>ICV_IAR1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_EOIR1</td>
<td>c12</td>
<td>0</td>
<td>c8</td>
<td>4</td>
<td>GICe</td>
</tr>
<tr>
<td>ICV_EOIR1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_HPPIR1</td>
<td>c12</td>
<td>0</td>
<td>c8</td>
<td>5</td>
<td>GICe</td>
</tr>
<tr>
<td>ICV_HPPIR1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_BPR1</td>
<td>c12</td>
<td>0</td>
<td>c8</td>
<td>6</td>
<td>GICe</td>
</tr>
<tr>
<td>ICV_BPR1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_CTLR</td>
<td>c12</td>
<td>0</td>
<td>c8</td>
<td>7</td>
<td>GICe</td>
</tr>
<tr>
<td>ICV_CTLR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_SRE</td>
<td>c12</td>
<td>0</td>
<td>c8</td>
<td>8</td>
<td>GICe</td>
</tr>
<tr>
<td>ICC_IGRPEN0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICV_IGRPEN0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_IGRPEN1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICV_IGRPEN1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICC_ASGI1R, 64 bits wide</td>
<td>-</td>
<td>1</td>
<td>c12</td>
<td>9</td>
<td>GICe</td>
</tr>
</tbody>
</table>
### Table G7-3 AArch32 VMSA (coproc==0b1111) register summary, in MCR/MRC parameter order

<table>
<thead>
<tr>
<th>Name</th>
<th>CRn</th>
<th>opc1</th>
<th>CRm</th>
<th>opc2</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICC_SGI0R, 64 bits wide</td>
<td>-</td>
<td>2</td>
<td>c12</td>
<td>-</td>
<td>GICe</td>
</tr>
<tr>
<td>HVBARc</td>
<td>c12</td>
<td>4</td>
<td>c0</td>
<td>0</td>
<td>v8.0e</td>
</tr>
<tr>
<td>HRMRp</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>v8.0e</td>
</tr>
<tr>
<td>ICH_AP0R0</td>
<td>c8</td>
<td></td>
<td>0</td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_AP0R1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_AP0R2</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_AP0R3</td>
<td>c12</td>
<td>4</td>
<td>c8</td>
<td>3</td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_AP1R0</td>
<td>e9</td>
<td></td>
<td>0</td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_AP1R1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_AP1R2</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_AP1R3</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICC_HSRE</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_HCR</td>
<td>c11</td>
<td>0</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_VTR</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_MISR</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_EISR</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_ELRSR</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_VMCR</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_LR&lt;n&gt;, for n==0 to 7</td>
<td>c12</td>
<td>0-7</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_LR&lt;n&gt;, for n==8 to 15</td>
<td>c13</td>
<td>0-7</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_LRC&lt;n&gt;, for n==0 to 7</td>
<td>c14</td>
<td>0-7</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICH_LRC&lt;n&gt;, for n==8 to 15</td>
<td>c15</td>
<td>0-7</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICC_MCTRLR</td>
<td>6</td>
<td>c12</td>
<td>4</td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICC_MSRE</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
<tr>
<td>ICC_MGRPEN1</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td>GICe</td>
</tr>
</tbody>
</table>
### Table G7-3 AArch32 VMSA (coproc==0b1111) register summary, in MCR/MRC parameter order

<table>
<thead>
<tr>
<th>Name</th>
<th>CRn</th>
<th>opc1</th>
<th>CRm</th>
<th>opc2</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCSEIDR</td>
<td>c13</td>
<td>0</td>
<td>c0</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>CONTEXTIDR</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>v8.0</td>
</tr>
<tr>
<td>TPIDRURW</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>v8.0</td>
</tr>
<tr>
<td>TPIDRUARIO</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>v8.0</td>
</tr>
<tr>
<td>TPIDRPURW</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>v8.0</td>
</tr>
<tr>
<td>AMCR</td>
<td>c2</td>
<td>0</td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>AMCFGFR</td>
<td>c2</td>
<td>1</td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>AMCGCR</td>
<td>c2</td>
<td>2</td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>AMUSERENR</td>
<td>c2</td>
<td>3</td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>AMCNTENCLR0</td>
<td>c2</td>
<td>4</td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>AMCNTENSET0</td>
<td>c2</td>
<td>5</td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>AMCNTENCLR1</td>
<td>c3</td>
<td>0</td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>AMCNTENSET1</td>
<td>c3</td>
<td>1</td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>AMEVTYPE0&lt;n&gt;,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>for n==0 to 7</td>
<td>c6</td>
<td>0-7</td>
<td></td>
<td>AMUq</td>
<td></td>
</tr>
<tr>
<td>AMEVTYPE0&lt;n&gt;,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>for n==8 to 15</td>
<td>c7</td>
<td></td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>AMEVTYPE1&lt;n&gt;,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>for n==0 to 7</td>
<td>c14</td>
<td></td>
<td></td>
<td>AMUq</td>
<td></td>
</tr>
<tr>
<td>AMEVTYPE1&lt;n&gt;,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>for n==8 to 15</td>
<td>c15</td>
<td></td>
<td></td>
<td>AMUq</td>
<td></td>
</tr>
<tr>
<td>AMEVCNTR0&lt;n&gt;,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>for n==0 to 7, 64 bits wide</td>
<td>-</td>
<td>0-7</td>
<td>c0</td>
<td>AMUq</td>
<td></td>
</tr>
<tr>
<td>AMEVCNTR0&lt;n&gt;,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>for n==8 to 15, 64 bits wide</td>
<td>-</td>
<td>c1</td>
<td>AMUq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMEVCNTR1&lt;n&gt;,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>for n==0 to 7, 64 bits wide</td>
<td>-</td>
<td>c4</td>
<td>AMUq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMEVCNTR1&lt;n&gt;,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AMUq</td>
</tr>
<tr>
<td>for n==8 to 15, 64 bits wide</td>
<td>-</td>
<td>c5</td>
<td>AMUq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTPIDRC</td>
<td>c13</td>
<td>4</td>
<td>c0</td>
<td>2</td>
<td>v8.0</td>
</tr>
<tr>
<td>CNTPCTR, 64 bits wide</td>
<td>-</td>
<td>0</td>
<td>c14</td>
<td>-</td>
<td>v8.0</td>
</tr>
</tbody>
</table>
### Table G7-3 AArch32 VMSA (coproc==0b1111) register summary, in MCR/MRC parameter order

<table>
<thead>
<tr>
<th>Name</th>
<th>CRn</th>
<th>opc1</th>
<th>CRm</th>
<th>opc2</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNTFRQ&lt;sup&gt;r&lt;/sup&gt;</td>
<td>c14</td>
<td>0</td>
<td>c0</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>CNTKCTL&lt;sup&gt;r&lt;/sup&gt;</td>
<td>c1</td>
<td>0</td>
<td>c1</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>CNTP_TVAL&lt;sup&gt;r&lt;/sup&gt;</td>
<td>c2</td>
<td>0</td>
<td>c2</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>CNTP_CTL&lt;sup&gt;r&lt;/sup&gt;</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>v8.0</td>
</tr>
<tr>
<td>CNTV_TVAL&lt;sup&gt;r&lt;/sup&gt;</td>
<td>c3</td>
<td>0</td>
<td>c3</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>CNTV_CTL&lt;sup&gt;r&lt;/sup&gt;</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>v8.0</td>
</tr>
<tr>
<td>PMEVCNTR&lt;sub&gt;n&lt;/sub&gt;, for n==0 to 7&lt;sup&gt;k&lt;/sup&gt;</td>
<td>c8</td>
<td>0-7</td>
<td>c8</td>
<td>0-7</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMEVCNTR&lt;sub&gt;n&lt;/sub&gt;, for n==8 to 15&lt;sup&gt;k&lt;/sup&gt;</td>
<td>c9</td>
<td>0-7</td>
<td>c9</td>
<td>0-7</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMEVCNTR&lt;sub&gt;n&lt;/sub&gt;, for n==16 to 23&lt;sup&gt;k&lt;/sup&gt;</td>
<td>c10</td>
<td>0-7</td>
<td>c10</td>
<td>0-7</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMEVCNTR&lt;sub&gt;n&lt;/sub&gt;, for n==24 to 30&lt;sup&gt;k&lt;/sup&gt;</td>
<td>c11</td>
<td>0-6</td>
<td>c11</td>
<td>0-6</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMEVTYPEP&lt;sub&gt;n&lt;/sub&gt;, for n==0 to 7&lt;sup&gt;k&lt;/sup&gt;</td>
<td>c12</td>
<td>0-7</td>
<td>c12</td>
<td>0-7</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMEVTYPEP&lt;sub&gt;n&lt;/sub&gt;, for n==8 to 15&lt;sup&gt;k&lt;/sup&gt;</td>
<td>c13</td>
<td>0-7</td>
<td>c13</td>
<td>0-7</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMEVTYPEP&lt;sub&gt;n&lt;/sub&gt;, for n==16 to 23&lt;sup&gt;k&lt;/sup&gt;</td>
<td>c14</td>
<td>0-7</td>
<td>c14</td>
<td>0-7</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMEVTYPEP&lt;sub&gt;n&lt;/sub&gt;, for n==17 to 30&lt;sup&gt;k&lt;/sup&gt;</td>
<td>c15</td>
<td>0-6</td>
<td>c15</td>
<td>0-6</td>
<td>v8.0</td>
</tr>
<tr>
<td>PMCCFILTR&lt;sup&gt;k&lt;/sup&gt;</td>
<td>c15</td>
<td>7</td>
<td>c15</td>
<td>7</td>
<td>v8.0</td>
</tr>
<tr>
<td>CNTVCTR&lt;sup&gt;r&lt;/sup&gt;, 64 bits wide</td>
<td>-</td>
<td>1</td>
<td>c14</td>
<td>-</td>
<td>v8.0</td>
</tr>
<tr>
<td>CNTP_CVAL&lt;sup&gt;r&lt;/sup&gt;, 64 bits wide</td>
<td>-</td>
<td>2</td>
<td>c14</td>
<td>-</td>
<td>v8.0</td>
</tr>
<tr>
<td>CNTV_CVAL&lt;sup&gt;r&lt;/sup&gt;, 64 bits wide</td>
<td>-</td>
<td>3</td>
<td>c14</td>
<td>-</td>
<td>v8.0</td>
</tr>
<tr>
<td>CNTVOFF&lt;sup&gt;s&lt;/sup&gt;, 64 bits wide</td>
<td>-</td>
<td>4</td>
<td>c14</td>
<td>-</td>
<td>v8.0</td>
</tr>
<tr>
<td>CNTHCTL&lt;sup&gt;r&lt;/sup&gt;</td>
<td>c14</td>
<td>4</td>
<td>c1</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>CNTHP_TVAL&lt;sup&gt;r&lt;/sup&gt;</td>
<td>c14</td>
<td>4</td>
<td>c2</td>
<td>0</td>
<td>v8.0</td>
</tr>
<tr>
<td>CNTHP_CTL&lt;sup&gt;r&lt;/sup&gt;</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>v8.0</td>
</tr>
<tr>
<td>CNTHP_CVAL&lt;sup&gt;r&lt;/sup&gt;, 64 bits wide</td>
<td>-</td>
<td>6</td>
<td>c14</td>
<td>-</td>
<td>v8.0</td>
</tr>
<tr>
<td>CNTPCTSS&lt;sup&gt;r&lt;/sup&gt;, 64 bits wide</td>
<td>-</td>
<td>8</td>
<td>c14</td>
<td>-</td>
<td>v8.6</td>
</tr>
<tr>
<td>CNTVCTSS&lt;sup&gt;r&lt;/sup&gt;, 64 bits wide</td>
<td>-</td>
<td>9</td>
<td>c14</td>
<td>-</td>
<td>v8.6</td>
</tr>
<tr>
<td>Reserved&lt;sup&gt;t&lt;/sup&gt;</td>
<td>c15</td>
<td>0-7</td>
<td>c0-c15</td>
<td>0-7</td>
<td>-</td>
</tr>
</tbody>
</table>
a. **REVIDR** is an optional register. If it is not implemented, the encoding with opc2 set to 2 is an alias of **MIDR**.

b. When **FEAT_CCIDX** is implemented, **CCSIDR2** is implemented.

c. Implemented only as part of EL2 when EL2 is using AArch32. Otherwise, encoding is unallocated and **UNDEFINED**.

d. Implemented only as part of EL3 when EL3 is using AArch32. Otherwise, encoding is unallocated and **UNDEFINED**.

e. GIC System register, see **About the GIC System registers** on page G7-9319. As that subsection describes, each **ICV_** register uses the same encoding as the corresponding **ICC_** register.

f. This register is accessible only in Debug state.

g. RAS Extension System registers, see **The Reliability, Availability, and Serviceability Extension** on page A2-122.

h. When **FEAT_SPECRES** is implemented, the execution and data prediction restriction instructions are implemented, see **Execution and data prediction restriction System instructions** on page G4-9134.

i. For performance reasons, Arm deprecates any use of these memory barrier operations.

j. Reserved for **IMPLEMENTATION DEFINED** branch predictor, cache, and TCM operations, see **Reserved 32-bit encodings with {coproc==0b1111, CRn==c9}** on page G7-9306.

k. Performance Monitors Extension System register, see **Performance Monitors registers** on page G8-9963.

l. Reserved for **IMPLEMENTATION DEFINED** performance monitors, see **Reserved 32-bit encodings with {coproc==0b1111, CRn==c9}** on page G7-9306.

m. Reserved for **IMPLEMENTATION DEFINED** TLB lockdown operations, see **Reserved 32-bit encodings with {coproc==0b1111, CRn==c10}** on page G7-9307.

n. When an implementation is using the Long descriptor translation table format, these encodings access the **MAIR0** and **MAIR1** registers. Otherwise, they use **PRRR** and **NMRR**.

o. Reserved for **IMPLEMENTATION DEFINED** DMA operations for TCM access, see **Reserved 32-bit encodings with {coproc==0b1111, CRn==c11}** on page G7-9307.

p. Only one of **RMR** and **HRMR** is implemented, corresponding to the highest implemented Exception level, and the register is implemented only if that Exception level is using AArch32.

q. Activity Monitors System register, see **Activity Monitors registers** on page G8-10047.

r. Generic Timer System register, see **Generic Timer registers** on page D17-7014.

s. Implemented as RW as part of the Generic Timer on an implementation that includes EL2 and when EL2 is using AArch32. For more information, see **The virtual offset register** on page G6-9295.

t. Reserved for **IMPLEMENTATION DEFINED** purposes, see **Reserved 32-bit encodings with {coproc==0b1111, CRn==c15}** on page G7-9307.

### About the GIC System registers

From version 3.0 of the GIC architecture specification, the specification defines three groups of System registers, identified by the prefix of the register name:

- **ICC_** GIC physical CPU interface System registers.
- **ICH_** GIC virtual interface control System registers.
- **ICV_** GIC Virtual CPU interface System registers.

**Note**

These registers are in addition to the GIC memory-mapped register groups **GICC_**, **GICD_**, **GICH_**, **GICR_**, **GICV_**, and **GITS_**.

In AArch32, the GIC System registers are all in the (coproc==0b1111) encoding space with (CRn==c12). The **ICV_** registers have the same {CRn, opc1, CRn, op2} encodings as the corresponding **ICC_** registers. For these encodings, GIC register configuration fields determine which register is accessed.
When implemented, the GIC System registers form part of an Arm processor implementation, and therefore these registers are included in the register summaries. However, the registers are defined only in the GIC Architecture Specification.

For more information, see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0* (ARM IHI 0069).
Chapter G8
AArch32 System Register Descriptions

This chapter describes each of the AArch32 System registers.

It contains the following sections:

- About the AArch32 System registers on page G8-9322.
- General system control registers on page G8-9337.
- Debug registers on page G8-9831.
- Performance Monitors registers on page G8-9963.
- Activity Monitors registers on page G8-10047.
- RAS registers on page G8-10083.
- Generic Timer registers on page G8-10144.
G8.1 About the AArch32 System registers

For general information about the AArch32 System registers, see:

In Chapter G5 The AArch32 Virtual Memory System Architecture:
- About the System registers for VMSAv8-32 on page G5-9281.
- Functional grouping of VMSAv8-32 System registers on page G5-9286.

In Chapter G7 AArch32 System Register Encoding:
- AArch32 VMSA organization of registers in the (coproc == 0b1110) encoding space on page G7-9301.
- AArch32 VMSA organization of registers in the (coproc == 0b1111) encoding space on page G7-9304.

In this chapter:
- Fixed values in the System register descriptions.
- General behavior of System registers.
- Principles of the ID scheme for fields in ID registers on page G8-9332.
- About AArch32 System register accesses on page G8-9334.

The remainder of this chapter describes the AArch32 System registers, in the following sections:
- General system control registers on page G8-9337.
- Debug registers on page G8-9831.
- Performance Monitors registers on page G8-9963.
- Generic Timer registers on page G8-10144.

G8.1.1 Fixed values in the System register descriptions

See Fixed values in AArch32 instruction and System register descriptions on page F1-7235. This section defines how the glossary terms RAZ, RES0, RAO, and RES1 can be represented in the System register descriptions.

G8.1.2 General behavior of System registers

Except where indicated, System registers are 32-bits wide. As stated in About the System registers for VMSAv8-32 on page G5-9281, there are some 64-bit registers, and these include cases where software can access either a 32-bit view or a 64-bit view of a register. The register summaries, and the individual register descriptions, identify the 64-bit registers and how they can be accessed.

The following sections give information about the general behavior of these registers:
- Register names.
- Read-only bits in read/write registers on page G8-9323.
- The CPUID identification scheme on page G8-9323.
- IMPLEMENTATION DEFINED performance monitors on page G8-9323.
- UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for AArch32 System register accesses on page G8-9323.
- Read-only and write-only register encodings on page G8-9325.
- Reset behavior of AArch32 System registers on page G8-9326.
- Synchronization of changes to AArch32 System registers on page G8-9326.

Unless otherwise indicated, information in the listed sections applies to all AArch32 System registers

See also About AArch32 System register accesses on page G8-9334.

Register names

The Arm architecture guarantees not to define any register name prefixed with IMP_ as part of the standard Arm architecture.
Note

Arm strongly recommends that any register names created in the IMPLEMENTATION DEFINED register spaces be prefixed with IMP_, where appropriate.

Read-only bits in read/write registers

Some read/write registers include bits that are read-only. These bits ignore writes.

The CPUID identification scheme

The ID_* registers were originally called the CPUID identification scheme registers. However, functionally, there is no value in separating these registers from the slightly larger Identification registers functional group. See Table K17-29 on page K17-11900 for a list of the ID_* registers.

IMPLEMENTATION DEFINED performance monitors

VMSAv8-32 reserves some additional System register encodings in the (coproc==0b1111) encoding space for optional additional IMPLEMENTATION DEFINED performance monitors. Table G8-1 on page G8-9323 shows the allocation of these encodings:

<table>
<thead>
<tr>
<th>CRn</th>
<th>opc1</th>
<th>CRm</th>
<th>opc2</th>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>c9</td>
<td>0-7</td>
<td>c12-c14</td>
<td>0-7</td>
<td>Performance Monitors Extension registers, see Performance monitors registers on page K17-11901</td>
<td>RW or ROa</td>
</tr>
<tr>
<td>c15</td>
<td>0-7</td>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
<td></td>
<td>b</td>
</tr>
</tbody>
</table>

a. The table referenced in the Name on page G8-9323 entry shows the type of each of the OPTIONAL Performance Monitors Extension registers.
b. Access depends on the register or operation, and is IMPLEMENTATION DEFINED.

UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for AArch32 System register accesses

This section defines UNPREDICTABLE and UNDEFINED behaviors for accesses to System registers, including those cases where the Armv8 behavior is CONSTRAINED UNPREDICTABLE.

In AArch32 state the following operations are UNDEFINED:

- All LDC and STC accesses, except for the LDC access to DBGDTRXint and the STC access to DBGDTRXint specified in Table G7-2 on page G7-9303.
- All MCR and MRC operations to the (coproc==0b111x) encoding space, except for those explicitly defined as accessing 64-bit System registers specified in Table G7-1 on page G7-9302 and Table G7-3 on page G7-9308.

Unless otherwise indicated in the individual register descriptions:

- Reserved fields in registers are RES0.
- Assigning a reserved value to a field has a CONSTRAINED UNPREDICTABLE effect, see Reserved values in System and memory-mapped registers and translation table entries on page K1-11583.

The following subsections give more information about UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for accesses to the (coproc==0b111x) encoding space:

- Accesses to unallocated encodings in the (coproc==0b111x) encoding space on page G8-9324.
- Additional rules for MCR and MRC accesses to System registers on page G8-9324.
- Effects of EL3 and EL2 on System register accesses on page G8-9324.
**Accesses to unallocated encodings in the \( \text{coproc} = 0b111x \) encoding space**

In Armv8-A, accesses to unallocated register encodings in the \( \text{coproc} = 0b111x \) encoding space are UNDEFINED.

--- **Note** ---

In Armv7, except for 32-bit registers encoded with a \( \text{CRn} \) value of c12, accesses to unallocated 32-bit registers were UNPREDICTABLE. The Armv8 CONSTRAINED UNPREDICTABLE behavior of these accesses is that they are UNDEFINED, see *Unallocated System register access instructions* on page K1-11565.

---

**Additional rules for MCR and MRC accesses to System registers**

The following operations are CONSTRAINED UNPREDICTABLE for all encodings in the \( \text{coproc} = 0b111x \) encoding space:

- All MCR operations from the PC.
- All MRC operations to \( \text{APSR}_{\text{nzcv}} \), except for the \( \text{coproc} = 0b1110 \) MRC operation to \( \text{APSR}_{\text{nzcv}} \) from \( \text{DBGDSCR}_{\text{int}} \).

The CONSTRAINED UNPREDICTABLE behavior of these operations is described in *Using R15 by instruction* on page K1-11563.

For registers and operations that are accessible from a particular Privilege level, any attempt to access those registers from a lower Privilege level is UNDEFINED.

Some individual registers can be made inaccessible by setting configuration bits, possibly including IMPLEMENTATION DEFINED configuration bits, to disable access to the register. The effects of the architecturally-defined configuration bits are defined individually in this manual. Unless explicitly stated otherwise in this manual, setting a configuration bit to disable access to a register results in the register becoming UNDEFINED for MRC and MCR accesses.

See also *Read-only and write-only register encodings* on page G8-9325.

---

**Effects of EL3 and EL2 on System register accesses**

EL2 and EL3 introduce classes of System registers, described in *Classification of System registers* on page G5-9281. Some of these classes of register are either:

- Accessible only from certain modes or states.
- Accessible from certain modes or states only when configuration settings permit the access.

Accesses to these registers that are not permitted are UNDEFINED, meaning execution of the register access instruction generates an Undefined Instruction exception.

--- **Note** ---

This section applies only to registers that are accessible from some modes and states. That is, it applies only to register access instructions using an encoding that, under some circumstances, would perform a valid register access.

---

The following register classes restrict access in this way:

**Restricted access System registers**

This register class is defined in any implementation that includes EL3.

Restricted access registers other than the NSACR are accessible only from Secure EL3 modes. All other accesses to these registers are UNDEFINED.

The NSACR is a special case of a Restricted access register and:

- The NSACR is:
  - Read/write accessible from Secure PL1 modes.
  - Is Read-only accessible from Non-secure PL2 and PL1 modes.
- All other accesses to the NSACR are UNDEFINED.
For more information, including behavior when EL3 is using AArch64 or is not implemented, see Restricted access System registers on page G5-9282.

Configurable access System registers

This register class is defined in any implementation that includes EL3.

Most Configurable access registers are accessible from Non-secure state only if control bits in the NSACR permit Non-secure access to the register. Otherwise, a Non-secure access to the register is UNDEFINED.

For other Configurable access registers, control bits in the NSACR control the behavior of bits or fields in the register when it is accessed from Non-secure state. That is, Non-secure accesses to the register are permitted, but the NSACR controls how they behave. The only architecturally-defined register of this type is the CPACR.

For more information, see Configurable access System registers on page G5-9282.

EL2-mode System registers

This register class is defined only in an implementation that includes EL2.

EL2-mode registers are accessible only from:

• The Non-secure EL2 mode, Hyp mode.
• Secure Monitor mode when SCR.NS is set to 1.

All other accesses to these registers are UNDEFINED.

For more information, see Hyp mode read/write registers in the (coproc == 0b1111) encoding space on page G5-9283 and Hyp mode encodings for shared (coproc == 0b1111) System registers on page G5-9283.

EL2-mode write-only operations

This register class is defined only in an implementation that includes EL2.

EL2-mode write-only operations are accessible only from:

• The Non-secure EL2 mode, Hyp mode.
• Secure Monitor mode, regardless of the value of SCR.NS.

Write accesses to these operations are:

• CONSTRAINED UNPREDICTABLE in Secure EL3 modes other than Monitor mode.
• UNDEFINED in Non-secure modes other than Hyp mode.

For more information, see Hyp mode (coproc == 0b1111) write-only System instructions on page G5-9284.

In addition, in any implementation that includes EL3, when EL3 is using AArch32, if write access to a register is disabled by the CP15SDISABLE signal then any MCR access to that register is UNDEFINED.

Read-only and write-only register encodings

Some System registers are read-only (RO) or write-only (WO). For example:

• Most identification registers are read-only.
• Most encodings that perform an operation, such as a cache maintenance instruction, are write-only.

If a particular Privilege level defines a register to be:

• RO, then any attempt to write to that register, at that Privilege level, is UNDEFINED. This means that any access to that register with L == 0 is UNDEFINED.
• WO, then any attempt to read from that register, at that Privilege level, is UNDEFINED. This means that any access to that register with L== 1 is UNDEFINED.

For IMPLEMENTATION DEFINED encoding spaces, the treatment of the encodings is IMPLEMENTATION DEFINED.
This section applies only to registers that this manual defines as RO or WO. It does not apply to registers for which other access permissions are explicitly defined.

**Reset behavior of AArch32 System registers**

Reset values apply only to RW registers and fields, however:

- Some RO registers or fields, including feature ID registers and some status registers or register fields, always return a known value.
- Some RW and RO registers or register fields return status information about the PE. Unless the register description indicates that the value is UNKNOWN on reset, a read of the register immediately after a reset returns valid information.
- Some RW and RO registers and fields are aliases of other registers or fields. In these cases, the reset behavior of the aliased register or field determines the value returned by a read of the register immediately after a reset.
- WO registers that only have an effect on writes do not have meaningful reset values. However, an access to a WO register might affect underlying state, and that state might have a defined reset value.
- IMPLEMENTATION DEFINED registers have IMPLEMENTATION DEFINED reset behavior.

After a reset, only a limited subset of the PE state is guaranteed to be set to defined values. Also, for debug and trace System registers, reset requirements must take account of different levels of reset. For more information about the reset behavior of System registers when the PE resets into an Exception level that is using AArch32, see:

- **PE state on reset into AArch32 state on page G1-8986.**
- The appropriate Trace architecture specification, for the Trace System registers.

When the PE resets into an Exception level that is using AArch64, PE state that relates to execution in AArch32 state, including the System register values, is UNKNOWN. The only exception to this is state that applies to execution in both AArch64 state and AArch32 state and that has a defined reset value on the reset into AArch64 state. An example of such PE state is the EDPRSR.SR bit.

For a PE reset into an Exception level that is using AArch32, the architecture defines which AArch32 System registers have a defined reset value, and when that defined reset value applies. The register descriptions include this information, and **PE state on reset into AArch32 state on page G1-8986** summarizes these architectural requirements. Otherwise, RW registers reset to an architecturally unknown value.

**Note**

In an implementation that includes EL3, unless this manual explicitly states otherwise, only the Secure instance of a banked register is reset to the defined value. This means that software must program the Non-secure instance of the register with the required values. Typically, this programming is part of the PE boot sequence.

**Pseudocode description of resetting System registers**

The `AArch32.ResetControlRegisters()` pseudocode function resets all System registers, and register fields, that have defined reset values, as described in this section and **PE state on reset into AArch32 state on page G1-8986.**

**Note**

For debug and trace System registers, this function resets registers as defined for the appropriate level of reset.

**Synchronization of changes to AArch32 System registers**

In this section, this PE means the PE on which accesses are being synchronized.
### Note

See Definitions of direct and indirect reads and writes and their side-effects on page G8-9331 for definitions of the terms direct write, direct read, indirect write, and indirect read.

A direct write to a System register might become visible at any point after the change to the register, but without a Context synchronization event there is no guarantee that the change becomes visible.

Any direct write to a System register is guaranteed not to affect any instruction that appears, in program order, before the instruction that performed the direct write, and any direct write to a System register must be synchronized before any instruction that appears after the direct write, in program order, can rely on the effect of that write. The only exceptions to this are:

- All direct writes to the same register, using the same encoding, are guaranteed to occur in program order.
- All direct writes to a register are guaranteed to occur in program order relative to all direct reads of the same register using the same encoding.
- Any System register access that an Arm Architecture Specification or equivalent specification defines as not requiring synchronization.
- If an instruction that appears in program order before the direct write performs a memory access, such as a memory-mapped register access, that causes an indirect read or write to a register, that memory access is subject to the memory order model. In this case, if permitted by the memory order model, the instruction that appears in program order before the direct write can be affected by the direct write. For information about the memory order model, see Definition of the memory model on page E2-7168.

These rules mean that an instruction that writes to one of the address translation instructions described in Address translation instructions on page G5-9271 must be explicitly synchronized to guarantee that the result of the address translation instruction is visible in the PAR.

### Note

In this case, the direct write to the encoding of the address translation instruction causes an indirect write to the PAR. Without a Context synchronization event after the direct write, there is no guarantee that the indirect write to the PAR is visible.

Conceptually, the explicit synchronization occurs as the first step of any Context synchronization event. This means that if the operation uses the state that had been changed but not synchronized before the operation occurred, the operation is guaranteed to use the state as if it had been synchronized.

### Note

- This explicit synchronization is applied as the first step of the execution of any instruction that causes the synchronization operation. This means it does not synchronize any effect of changes to the System registers that might affect the fetch and decode of the instructions that cause the operation, such as breakpoints or changes to translation tables.
- For a synchronous exception, the control state in use at the time the exception is generated determines the exception syndrome information, and this syndrome information is not changed by this synchronization at the start of taking the exception.

Except for the register reads listed in Registers with some architectural guarantee of ordering or observability on page G8-9329, if no Context synchronization event is performed, direct reads of System registers can occur in any order.

Table G8-2 on page G8-9328 shows the synchronization requirement between two reads or writes that access the same System register. In the column headings, First and Second refer to:

- Program order, for any read or write caused by the execution of an instruction by this PE, other than a read or write caused by a memory access made by that instruction.
The order of arrival of asynchronous reads or writes made by this PE relative to the execution of instructions by this PE.

In addition:

- For indirect reads or writes caused by an external agent, such as a debugger, the mechanism that determines the order of the reads or writes is defined by that external agent. The external agent can provide mechanisms that ensure that any read or write it makes arrives at the PE. These indirect reads and writes are asynchronous to software execution on the PE.

- For indirect reads or writes caused by memory-mapped reads or writes made by this PE, the ordering of the memory accesses is subject to the memory order model, including the effect of the memory type of the accessed memory address. This applies, for example, if this PE reads or writes one of its registers in a memory-mapped register interface.

The mechanism for ensuring completion of these memory accesses, including ensuring the arrival of the asynchronous read or write at the PE, is defined by the system.

---

**Note**

Such accesses are likely to be given a Device memory attribute, but requiring this is outside the scope of the architecture.

---

- For indirect reads or writes caused by autonomous asynchronous events that are counted, for example events caused by the passage of time, the events are ordered so that:
  - Counts progress monotonically.
  - The events arrive at the PE in finite time and without undue delay.

### Table G8-2 Synchronization requirements for updates to System registers

<table>
<thead>
<tr>
<th>First read or write</th>
<th>Second read or write</th>
<th>Context synchronization event required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct read</td>
<td>Direct read</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Direct write</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Indirect read</td>
<td>No&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>Indirect write</td>
<td>No&lt;sup&gt;a&lt;/sup&gt;, but see text in this section for exceptions</td>
</tr>
<tr>
<td>Direct write</td>
<td>Direct read</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Direct write</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Indirect read</td>
<td>Yes&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>Indirect write</td>
<td>No, but see text in this section for exceptions</td>
</tr>
<tr>
<td>Indirect read</td>
<td>Direct read</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Direct write</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Indirect read</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Indirect write</td>
<td>No</td>
</tr>
<tr>
<td>Indirect write</td>
<td>Direct read</td>
<td>Yes, but see text in this section for exceptions</td>
</tr>
<tr>
<td></td>
<td>Direct write</td>
<td>No, but see text in this section for exceptions</td>
</tr>
<tr>
<td></td>
<td>Indirect read</td>
<td>Yes, but see text in this section for exceptions</td>
</tr>
<tr>
<td></td>
<td>Indirect write</td>
<td>No, but see text in this section for exceptions</td>
</tr>
</tbody>
</table>
a. Although no synchronization is required between a Direct write and a Direct read, or between a Direct read and an Indirect write, this does not imply that a Direct read causes synchronization of a previous Direct write. This means that the sequence Direct write followed by Direct read followed by Indirect read, with no intervening context synchronization, does not guarantee that the Indirect read observes the result of the Direct write.

If the indirect write is to a register that Registers with some architectural guarantee of ordering or observability shows as having some guarantee of the visibility of an indirect write, synchronization might not be required.

If a direct read or a direct write to a register is followed by an indirect write to that register that is caused by an external agent, or by an autonomous asynchronous event, or as a result of a memory-mapped write, then synchronization is required to guarantee the ordering of the indirect write relative to the direct read or direct write.

If an indirect write caused by a direct write is followed by an indirect write caused by an external agent, or by an autonomous asynchronous event, or as a result of a memory-mapped write, then synchronization is required to guarantee the ordering of the two indirect writes.

Where an indirect write occurs as a side-effect of an access, this happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect. For other information about indirect writes after a direct read or a direct write, see Definitions of direct and indirect reads and writes and their side-effects on page G8-9331

Note

Where a register has more that one encoding, a direct write to the register using a particular encoding is not an indirect write to the same register with a different encoding.

Where an indirect write is caused by the action of an external agent, such as a debugger, or by a memory-mapped read or write by the PE, then an indirect write by that agent to a register using a particular access mechanism, followed by an indirect read by that agent to the same register using the same access mechanism and address does not need synchronization.

Without explicit synchronization to guarantee the order of the accesses, where the same register is accessed by two or more of a System register access instruction, and external agent, and autonomous asynchronous event, or as a result of a memory-mapped access, the behavior must be as if the accesses occurred atomically and in any order. This applies even if the accesses occur simultaneously.

For information about the additional synchronization requirements for memory-mapped registers, see Synchronization requirements for AArch64 System registers on page D17-5547.

To guarantee the visibility of changes to some registers, additional operations might be required before the Context synchronization event. For such a register, the definition of the register identifies these additional requirements.

In this manual, unless the context indicates otherwise:

• Accessing a System register refers to a direct read or write of the register.
• Using a System register refers to an indirect read or write of the register.

Registers with some architectural guarantee of ordering or observability

For the registers for which Table G8-3 on page G8-9330 shows that the ordering of direct reads is guaranteed, multiple direct reads of a single register, using the same encoding, occur in program order without any explicit ordering.

For the registers for which Table G8-3 on page G8-9330 shows that some observability of indirect writes is guaranteed, an indirect write to the register caused by an external agent, an autonomous asynchronous event, or as a result of a memory-mapped write, is both:

• Observable to direct reads of the register, in finite time, without explicit synchronization.
• Observable to subsequent indirect reads of the register without explicit synchronization.
These two sets of registers are similar, as Table G8-3 on page G8-9330 shows:

<table>
<thead>
<tr>
<th>Register</th>
<th>Ordering of direct reads</th>
<th>Observability of indirect writes</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISR</td>
<td>Guaranteed</td>
<td>Guaranteed</td>
<td>Interrupt Status Register</td>
</tr>
<tr>
<td>DBGCLAIMCLR</td>
<td>Guaranteed</td>
<td>Guaranteed</td>
<td>Debug CLAIM registers</td>
</tr>
<tr>
<td>DBGCLAIMSET</td>
<td>-</td>
<td>Guaranteed</td>
<td></td>
</tr>
<tr>
<td>DBGDTRRXint</td>
<td>Guaranteed</td>
<td>Guaranteed</td>
<td>Debug Communication Channel registers</td>
</tr>
<tr>
<td>DBGDTTRXint</td>
<td>-</td>
<td>Guaranteed</td>
<td></td>
</tr>
<tr>
<td>The DCC flags in DBGDSRint</td>
<td>Guaranteed</td>
<td>Guaranteed</td>
<td></td>
</tr>
<tr>
<td>CNTPCT</td>
<td>Guaranteed</td>
<td>Guaranteed</td>
<td>Generic Timer registers</td>
</tr>
<tr>
<td>CNTP_TVAL</td>
<td>Guaranteed</td>
<td>Guaranteed</td>
<td></td>
</tr>
<tr>
<td>CNTVCT</td>
<td>Guaranteed</td>
<td>Guaranteed</td>
<td></td>
</tr>
<tr>
<td>CNTV_TVAL</td>
<td>Guaranteed</td>
<td>Guaranteed</td>
<td></td>
</tr>
<tr>
<td>CNTHP_TVAL</td>
<td>Guaranteed</td>
<td>Guaranteed</td>
<td></td>
</tr>
<tr>
<td>PMCCNTR</td>
<td>Guaranteed</td>
<td>Guaranteed</td>
<td>Performance Monitors Extension registers, if the implementation includes the extension</td>
</tr>
<tr>
<td>PMEVCNTR&lt;n&gt;</td>
<td>Guaranteed</td>
<td>Guaranteed</td>
<td></td>
</tr>
<tr>
<td>PMXEVCNTR</td>
<td>Guaranteed</td>
<td>Guaranteed</td>
<td></td>
</tr>
<tr>
<td>PMOVSET</td>
<td>Guaranteed</td>
<td>Guaranteed</td>
<td></td>
</tr>
<tr>
<td>PMOVSR</td>
<td>Guaranteed</td>
<td>Guaranteed</td>
<td></td>
</tr>
<tr>
<td>EDSR_PipeAdv and the DCC flags in EDSR</td>
<td>-</td>
<td>Guaranteed</td>
<td>Fields of the External Debug Status and Control Register</td>
</tr>
</tbody>
</table>

In addition to the requirements shown in Table G8-3 on page G8-9330:

- Indirect writes to the following registers as a result of memory-mapped writes, including accesses by external agents, are required to be observable to the indirect read made in determining the response to a subsequent memory-mapped access without explicit synchronization:
  
  - **OSLAR_EL1.** *OSLAR_EL1* is indirectly read to determine whether the subsequent access is permitted.

    **Note**
    
    *OSLAR_EL1* maps to the AArch32 System register DBGOSLR.

  
  - **EDLR,** if implemented. *EDLR* is indirectly read to determine whether a subsequent write or side-effect of an access is ignored.

    **Note**
    
    This requirement is stricter than the general requirement for the observability of indirect writes.

- The requirement that an indirect write to the registers in Table G8-3 on page G8-9330 is observable to direct reads in finite time does not imply that all observers will observe the indirect write at the same time.
For example, an increment of the system counter is an autonomous asynchronous event that performs an indirect write to the counter. This asynchronous event might generate a timer interrupt request, resulting in a Context synchronization event. When a GIC is used, the timer interrupt might arrive at the GIC after the PE has taken an interrupt request from another source, but before software reads the current interrupt ID from the GIC. This means that the GIC might identify the timer interrupt as the current interrupt. Software must not assume that a subsequent direct read of the counter register is guaranteed to observe the updated value of that register.

Although this example uses the counter-timer registers, it applies equally to other registers that might be linked to interrupt requests, including the PMU and Statistical Profiling status registers.

- When the PE is in Debug state, there are synchronization requirements for the Debug Communication Channel and Instruction Transfer registers. See DCC and ITR access in Debug state on page H4-10313.

The possibility that direct reads can occur early, in the absence of context synchronization, described in Ordering of reads of System registers on page G8-9334, still applies to the registers listed in Table G8-3 on page G8-9330.

Definitions of direct and indirect reads and writes and their side-effects

Direct and indirect reads and writes are defined as follows:

**Direct read**
Is a read of a register, using an MRC, MRRC, or STC instruction, that the architecture permits for the current PE state.

If a direct read of a register has a side-effect of changing the value of a register, the effect of a direct read on that register is defined to be an indirect write, and has the synchronization requirements of an indirect write. This means the indirect write is guaranteed to have occurred, and to be visible to subsequent direct or indirect reads and writes only if synchronization is performed after the direct read.

--- Note ---
The indirect write described here can affect either the register written to by the direct write, or some other register. The synchronization requirement is the same in both cases.

**Direct write**
Is a write to a register, using an MCR, MCRR, or LDC instruction, that the architecture permits for the current PE state.

In the following cases, the side-effect of the direct write is defined to be an indirect write of the affected register, and has the synchronization requirements of an indirect write:

- If the direct write has a side-effect of changing the value of a register other than the register accessed by the direct write.
- If the direct write has a side-effect of changing the value of the register accessed by the direct write, so that the value in that register might not be the value that the direct write wrote to the register.

In both cases, this means that the indirect write is not guaranteed to be visible to subsequent direct or indirect reads and writes unless synchronization is performed after the direct write.

--- Note ---

As an example of a direct write to a register having an effect that is an indirect write of that register, writing 1 to a PMCNTENCLR.Pr bit is also an indirect write, because if the Pr bit had the value 1 before the direct write, the side-effect of the write changes the value of that bit to 0.

- The indirect write described here can affect either the register written to by the direct write, or some other register. The synchronization requirement is the same in both cases. For example, writing 1 to a PMCNTENCLR.Pr bit that is set to 1 also changes the corresponding PMCNTENSET.Pr bit from 1 to 0. This means that the direct write to the PMCNTENCLR defines indirect writes to both itself and to the PMCNTENSET.
**Indirect read** is a use of the register by an instruction to establish the operating conditions for the instruction. Examples of operating conditions that might be determined by an indirect read are the translation table base address, or whether memory accesses are forced to be Non-cacheable.

Indirect reads include situations where the value of one register determines what value is returned by a second register. This means that any read of the second register is an indirect read of the register that determines what value is returned.

Indirect reads also include:

- Reads of the System registers by external agents, such as debuggers, as described in *Debug registers on page G8-9831*.
- Memory-mapped reads of the System registers made by the PE on which the System registers are implemented.

Where an indirect read of a register has a side-effect of changing the value of a register, that change is defined to be an indirect write, and has the synchronization requirements of an indirect write.

**Indirect write** is an update to the value of a register as a consequence of either:

- An exception, operation, or execution of an instruction that is not a direct write to that register.
- The asynchronous operation of an external agent.

This can include:

- The passage of time, as seen in counters or timers, including performance counters.
- The assertion of an interrupt.
- A write from an external agent, such as a debugger.

However, for some registers, the architecture gives some guarantee of visibility without any explicit synchronization, see *Registers with some architectural guarantee of ordering or observability on page G8-9329*.

---

**Note**

Taking an exception is a *Context synchronization event*. Any indirect write performed as part of an exception entry does not require additional synchronization. This includes the indirect writes to the registers that report the exception, as described in *Exception reporting in a VMSA Arm®-32 implementation on page G5-9251*.

---

**G8.1.3 Principles of the ID scheme for fields in ID registers**

The Arm architecture specifies a number of *ID registers* that are characterized as comprising a set of 4-bit *ID fields*. Each ID field identifies the presence, and possibly the level of support for, a particular feature in an implementation of the architecture. These fields follow an architectural model that aids their use by software and provides future compatibility. This section describes that model. *AArch32 ID registers to which this scheme applies on page G8-9333* identifies the set of ID registers that are accessible from AArch32 state.

A small number of ID fields do not follow the scheme described in this section. In these cases, the field description states that it does not follow this scheme.

---

**Note**

- The ID fields described here are distinct from register fields that enumerate the number of resources, such as the number of breakpoints, watchpoints, or performance monitors, or the amount of memory.
- ID fields that do not follow this scheme include the *ID_AA64DFR0_EL1.PMUVer*, *ID_DFR0_EL1.PerfMon*, *ID_DFR0.PerfMon* and *EDDFR.PMUVer* fields, see *Alternative ID scheme used for the Performance Monitors Extension version on page G8-9334*.
- The presence of an ID field for a feature does not imply that the feature is optional.

---

To provide forward compatibility, software can rely on the features of these fields that are described in this section.
The ID fields, which are either signed or unsigned, use increasing numerical values to indicate increases in functionality. Therefore, if a value of \(0x1\) indicates the presence of some instructions, then the value \(0x2\) will indicate the presence of those instructions plus some additional instructions or functionality. This means software can be written in the form:

```c
if (value >= number) {
    // do something that relies on the value of the feature
}
```

For ID fields where the value \(0x0\) defines that a feature is not present, the field holds an unsigned value. This covers the vast majority of such fields.

In a few cases, the architecture has been changed to permit implementations to exclude a feature that has previously been required and for which no ID field has been defined. In these cases, a new ID field is defined and:

- The field holds a signed value.
- The field value \(0xF\) indicates that the feature is not implemented.
- The field value \(0x0\) indicates that the feature is implemented.
- Software that depends on the feature can use the test:
  ```c
  if value >= 0 {
      // Software features that depend on the presence of the hardware feature
  }
  ```

In some cases, it has been decided retrospectively that the increase in functionality between two consecutive numerical values is too great, and it is desirable to permit an intermediate degree of functionality, and the means to discover this. This is done by the introduction of a fractional field that both:

- Is referred to in the definition of the original field.
- Applies only when the original field is at the lower value of the step.

In principle, a fractional field can be used for two different fractional steps, with different meanings associated with each of these steps. For this reason, a fractional field must be interpreted in the context of the field to which it relates and the value of that field. Example G8-1 shows the use of such a field.

### Example G8-1 Example of the use of a fractional field

For a field describing some class of functionality:

- The value \(0x1\) was defined as indicating that item A is present.
- The value \(0x2\) was defined as indicating that items B and C are present, in addition to item A.

Subsequently, it might be necessary to introduce a second ID field to indicate that A and B only are present. This new field is a fractional field, and might be defined as having the value \(0x1\) when A and B only are present. This fractional field is valid only when the original ID field has the value \(0x1\).

This approach means that:

- Software that depends on the test if \((value >= 0x2)\) can rely on features A, B, and C being present,
- Software that depends on the test if \((value >= 0x1)\) can rely on feature A being present.
- If new software needs to check only that features A and B are present, then it can test:
  ```c
  if (value >= 0x2 || (value == 0x1 && fractional_value >= 0x1)) {
      // Software features that depend on A and B only
  }
  ```

A fractional field uses the same approach of increasing numerical values indicating increasing functionality, and the fractional approach can also be applied recursively to fractional fields.

Unused ID fields, and fractional fields that are not applicable, are RES0 to allow their future use when features, or fractional implementation options, are added.

### AArch32 ID registers to which this scheme applies

- The Auxiliary Feature register ID_AFR0.
• The Processor Feature registers ID_PFR0 and ID_PFR1.
• The Debug Feature register ID_DFR0.
• The Memory Model Feature registers ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, and ID_MMFR4.
• The Instruction Set Attribute registers ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.
• The Media and VFP Feature registers MVFR0, MVFR1, and MVFR2.

Note

Principles of the ID scheme for fields in ID registers on page D17-5551 includes information about the AArch64 System registers and the memory-mapped registers to which this scheme applies.

Alternative ID scheme used for the Performance Monitors Extension version

The ID_AA64DFR0_EL1.PMUVer, ID_DFR0_EL1.PerfMon, ID_DFR0.PerfMon, and EDDFR.PMUVer fields, which identify the version of the Performance Monitors Extension, do not follow the standard ID scheme. Software must treat these fields as follows:

• The value 0xF indicates that the Arm-architected Performance Monitors Extension is not implemented.
• If the field value is not 0xF the field is treated as an unsigned value, as described for the standard ID scheme.

This means that software that depends on the implementation of a particular version of the Arm Performance Monitors Extension must be written in the form:

```c
if (value != 0xF and value >= number) {
    // do something that relies on version 'number' of the feature
}
```

For these fields, Arm deprecates use of the value 0xF in new implementations.

G8.1.4 About AArch32 System register accesses

The following subsections give more information about accesses to the AArch32 System registers:

• Ordering of reads of System registers.
• Accessing 32-bit System registers on page G8-9335.
• Accessing 64-bit System registers on page G8-9336.

Ordering of reads of System registers

Reads of the System registers can occur out of order with respect to earlier instructions executed on the same PE, provided that both:

• Any data dependencies between the instructions, as specified in Synchronization of changes to AArch32 System registers on page G8-9326, including read-after-read dependencies, are respected.
• The reads to the register do not occur earlier than the most recent Context synchronization event to its architectural position in the instruction stream.

Note

In particular, the values read from System registers that hold self-incrementing counts, such as the Performance Monitors counters or the Generic Timer counter or timers, could be accessed from any time after the previous Context synchronization event. For example, where a memory access is used to communicate a read of such a counter, an ISB must be inserted between the read of the memory location that is known to have returned its data, either as a result of a condition on that data or of the read having completed, and the read of the counter, if it is necessary that the counter returns a count value after the memory communication.
Accessing 32-bit System registers

Software accesses most 32-bit System registers using the generic MCR and MRC System register access instructions, specifying some or all of the parameters \{coproc, CRn, opc1, CRm, opc2\}, where:

- **coproc** Identifies the primary region of the System register encoding space. Takes one of the values:
  - p14 Encoded as 0b1110.
  - p15 Encoded as 0b1111.

- **CRn** Takes a value in the range c0-c15, encoded the corresponding 4-bit binary value, 0b0000-0b1111.
  
  In the (coproc==0b1110) encoding space, the opc1 value identifies the System register functional group, and CRn is the most significant identifier for the required register within that group.
  
  In the (coproc==0b1111) encoding space, CRn is the most significant identifier for the required register.

- **opc1** Takes a value in the range 0-7, encoded as its 3-bit binary value.
  
  In the (coproc==0b1110) encoding space, the opc1 value identifies the System register functional group, and can take the following values:
  - 0 Debug System registers.
  - 1 Trace System registers.
  - 7 Legacy Jazelle System registers.
  
  In the (coproc==0b1111) encoding space, opc1 can take any value in the range 0-7.

- **CRm** Takes a value in the range c0-c15, encoded the corresponding 4-bit binary value, 0b0000-0b1111.

- **opc2** Takes a value in the range 0-7, encoded as its 3-bit binary value.

  opc2 is optional in the MCR and MRC instruction syntax, and if no value is specified the encoding defaults to 0b000.

- **Rt** A general-purpose register to hold a 32-bit value to transfer to or from the System register. Takes a value in the range R0-R14, encoded as the corresponding 4-bit binary value, 0b0000-0b1110.

This means an MCR or MRC access to a specific 32-bit System register uses:

- A unique combination of coproc, CRn, opc1, CRm, and opc2, to specify the required System register.
- A general-purpose register, Rt, for the transferred 32-bit value.

See also:

- **MCR** on page F5-7707.
- **MRC** on page F5-7730.

A small number of AArch32 debug System registers are accessed using LDC or STC instructions. In these cases, the register to be accessed is identified in the instruction syntax by the use of p14, c5 where:

- **p14** Identifies that the access is to the (coproc==0b1110) encoding space.
- **c5** Identifies the target debug System register.

See the instruction descriptions:

- **LDC (immediate)** on page F5-7596.
- **LDC (literal)** on page F5-7598.
- **STC** on page F5-7952.

The only uses of LDC and STC permitted in Armv8-A are:

- An LDC access to load data from memory to DBGDTRTXint, see **LDC (immediate)** on page F5-7596 and **LDC (literal)** on page F5-7598.
- An STC access to store data to memory from DBGDTRRXint, see **STC** on page F5-7952.

A small number of AArch32 System registers are accessed using MRS, MSR, VMRS, or VMSR instructions, see the appropriate register and instruction description for more information, see:

- **MRS** on page F5-7734.
- **MSR (immediate)** on page F5-7744.
- **MSR (register)** on page F5-7746.
- **VMRS** on page F6-8565.
Accessing 64-bit System registers

Software accesses a 64-bit System register using the generic MCRR and MRRC System register access instructions, specifying the parameters \( \{ \text{coproc}, \text{Cr}_m, \text{opc1} \} \), where:

- **coproc**
  - Identifies the primary region of the System register encoding space. Takes one of the values:
    - p14 Encoded as \( 0b1110 \).
    - p15 Encoded as \( 0b1111 \).

- **Cr\_m**
  - Takes a value in the range \( c0-c15 \), encoded the corresponding 4-bit binary value, \( 0b0000-0b1111 \).
  - In the \( (\text{coproc}==0b1110) \) encoding space, the opc1 value identifies the System register functional group, and \( \text{Cr}_m \) is the most significant identifier for the required register within that group.
  - In the \( (\text{coproc}==0b1111) \) encoding space, \( \text{Cr}_m \) is the most significant identifier for the required register.

- **opc1**
  - Takes a value in the range 0-15, encoded as its 3-bit binary value.
  - In the \( (\text{coproc}==0b1110) \) encoding space, the opc1 value identifies the System register functional group, and can take the following values:
    - 0 Debug System registers.
    - 1 Trace System registers.
  - In the \( (\text{coproc}==0b1111) \) encoding space, opc1 can take any value in the range 0-15.

- **Rt**
  - A general-purpose register to hold bits[31:0] of the value to transfer to or from the System register.
  - Takes a value in the range \( R0-R14 \), encoded as the corresponding 4-bit binary value, \( 0b0000-0b1111 \).

- **Rt2**
  - A general-purpose register to hold bits[63:32] of the value to transfer to or from the System register.
  - Takes a value in the range \( R0-R14 \), encoded as the corresponding 4-bit binary value, \( 0b0000-0b1111 \).

This means an MCRR or MRRC access to a specific 64-bit System register uses:

- A unique combination of coproc, \( \text{Cr}_m \) and opc1, to specify the required 64-bit System register.
- Two general-purpose registers, each holding 32 bits of the value to transfer.

This means a PE can access a 64-bit System register using:

- An MCRR instruction to write to a System register, see **MCRR on page F5-7709**.
- An MRRC instruction to read a System register, see **MRRC on page F5-7732**.

When using an MCRR or MRRC instruction the System register access is 64-bit atomic.

Some 64-bit registers also have an MCR and MRC encoding. The MCR and MRC encodings for these registers access the least significant 32 bits of the register. For example, to access the PAR, software can:

- Use the following instructions to access all 64 bits of the register:
  
  ```
  MCRR p15, 0, <Rt>, <Rt2>, c7 ; Read 64-bit PAR into Rt (low word) and Rt2 (high word)
  MRRC p15, 0, <Rt>, <Rt2>, c7 ; Write Rt (low word) and Rt2 (high word) to 64-bit PAR
  ```

- Use the following instructions to access the least-significant 32 bits of the register:
  
  ```
  MRC p15, 0, <Rt>, c7, c4, 0 ; Read PAR[31:0] into Rt
  MCR p15, 0, <Rt>, c7, c4, 0 ; Write Rt to PAR[31:0]
  ```
G8.2 General system control registers

This section lists the System registers in AArch32 state that are not part of one of the other listed groups.
### G8.2.1 ACTLR, Auxiliary Control Register

The ACTLR characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED configuration and control options for execution at EL1 and EL0.

**Configurations**

AArch32 System register ACTLR bits [31:0] are architecturally mapped to AArch64 System register ACTLR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ACTLR are UNDEFINED.

Some bits might define global configuration settings, and be common to the Secure and Non-secure instances of the register.

**Attributes**

ACTLR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing ACTLR**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC\{c\}\{q\} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TACR == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TAC == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    R[t] = ACTLR_NS;
else
  R[t] = ACTLR;
elsif PSTATE.EL == EL2 then
if HaveEL(EL3) & ELUsingAArch32(EL3) then
    R[t] = ACTLR_NS;
else
    R[t] = ACTLR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        R[t] = ACTLR_S;
    else
        R[t] = ACTLR_NS;
    endif
else
    ACTLR_NS = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) & ELUsingAArch32(EL3) then
        ACTLR_NS = R[t];
    else
        ACTLR = R[t];
    endif
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        ACTLR_S = R[t];
    else
        ACTLR_NS = R[t];
    endif
endif

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if HaveEL(EL3) & ELUsingAArch32(EL3) then
    ACTLR_NS = R[t];
else
    ACTLR = R[t];
endif

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & ELUsingAArch32(EL2) & HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.TACR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() & ELUsingAArch32(EL2) & HCR.TAC == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif HaveEL(EL3) & ELUsingAArch32(EL3) then
        ACTLR_NS = R[t];
    else
        ACTLR = R[t];
    endif
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) & ELUsingAArch32(EL3) then
        ACTLR_NS = R[t];
    else
        ACTLR = R[t];
    endif
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        ACTLR_S = R[t];
    else
        ACTLR_NS = R[t];
    endif
G8.2.2 ACTLR2, Auxiliary Control Register 2

The ACTLR2 characteristics are:

**Purpose**

Provides additional space to the ACTLR register to hold IMPLEMENTATION DEFINED trap functionality for execution at EL1 and EL0.

**Configurations**

AArch32 System register ACTLR2 bits [31:0] are architecturally mapped to AArch64 System register `ACTLR_EL1[63:32]`. This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ACTLR2 are UNDEFINED.

In Armv8.0 and Armv8.1, it is IMPLEMENTATION DEFINED whether this register is implemented, or whether it causes UNDEFINED exceptions when accessed. The implementation of this register can be detected by examining `ID_MMFR4.AC2`.

From Armv8.2 this register must be implemented.

**Attributes**

ACTLR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing ACTLR2**

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
undefined;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TACR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsing AAArch32(EL2) && HCR_EL2.TAC == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        R[t] = ACTLR2_NS;
else
    R[t] = ACTLR2;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) & ELUsingAArch32(EL3) then
        R[t] = ACTLR2_NS;
    else
        R[t] = ACTLR2;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        R[t] = ACTLR2_S;
    else
        R[t] = ACTLR2_NS;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & !ELUsingAArch32(EL2) & HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TACR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR.TAC == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif HaveEL(EL3) & ELUsingAArch32(EL3) then
        ACTLR2_NS = R[t];
    else
        ACTLR2 = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) & ELUsingAArch32(EL3) then
        ACTLR2_NS = R[t];
    else
        ACTLR2 = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        ACTLR2_S = R[t];
    else
        ACTLR2_NS = R[t];
G8.2.3 ADFSR, Auxiliary Data Fault Status Register

The ADFSR characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED fault status information for Data Abort exceptions taken to EL1 modes, and EL3 modes when EL3 is implemented and is using AArch32.

**Configurations**

AArch32 System register ADFSR bits [31:0] are architecturally mapped to AArch64 System register AFSR0_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ADFSR are UNDEFINED.

**Attributes**

ADFSR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [31:0]**

- IMPLEMENTATION DEFINED.
- The reset behavior of this field is:
  - On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing ADFSR**

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && !ELUsingAAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif EL2Enabled() && !ELUsingAAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif HaveEL(EL3) && ELUsingAAArch32(EL3) then
        R[t] = ADFSR_NS;
    else
        R[t] = ADFSR;
elsif PSTATE_EL == EL2 then
    if HaveEL(EL3) && ELUsingAAArch32(EL3) then
        R[t] = ADFSR_NS;
```

else
  R[t] = ADFSR;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    R[t] = ADFSR_S;
  else
    R[t] = ADFSR_NS;
else
  ADFSR = R[t];
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    ADFSR_NS = R[t];
  else
    ADFSR = R[t];
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    ADFSR_S = R[t];
  else
    ADFSR_NS = R[t];

**MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}**

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>
G8.2.4 AIDR, Auxiliary ID Register

The AIDR characteristics are:

**Purpose**
Provides IMPLEMENTATION DEFINED identification information.
The value of this register must be used in conjunction with the value of MIDR.

**Configurations**
AArch32 System register AIDR bits [31:0] are architecturally mapped to AArch64 System register AIDR_EL1[31:0].
This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to AIDR are UNDEFINED.

**Attributes**
AIDR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [31:0]**
IMPLEMENTATION DEFINED.

**Accessing AIDR**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>}, \{#\}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & ELUsingAArch32(EL2) & HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.TID1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR.TID1 == '1' then
    AArch32.TakeHypTrapException(0x03);
else
  R[t] = AIDR;
elsif PSTATE.EL == EL2 then
  R[t] = AIDR;
elsif PSTATE.EL == EL3 then
  R[t] = AIDR;
### G8.2.5 AIFSR, Auxiliary Instruction Fault Status Register

The AIFSR characteristics are:

**Purpose**

Provides additional IMPLEMENTATION DEFINED fault status information for Prefetch Abort exceptions taken to EL1 modes, and EL3 modes when EL3 is implemented and is using AArch32.

**Configurations**

AArch32 System register AIFSR bits [31:0] are architecturally mapped to AArch64 System register AFSR1_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to AIFSR are UNDEFINED.

**Attributes**

AIFSR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [31:0]**

**IMPLEMENTATION DEFINED.**

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing AIFSR**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>}\ <\text{coproc}, \ (#<opc1>), <Rt>, <CRn>, <CRm>, (#<opc2>)}\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0000</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        R[t] = AIFSR_NS;
    else
        R[t] = AIFSR;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && ELUsingAArch32(EL3) then
            R[t] = AIFSR_NS;
```
else
    $R[t] = AIFS R$
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        $R[t] = AIFS R S$
    else
        $R[t] = AIFS R NS$
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && ELUsingAArch32(EL3) then
            $AIFS R NS = R[t]$
        else
            $AIFS R = R[t]$
        endif
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        $AIFS R S = R[t]$
    else
        $AIFS R NS = R[t]$
    endif
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        $AIFS R NS = R[t]$
    else
        $AIFS R = R[t]$
    endif
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        $AIFS R NS = R[t]$
    else
        $AIFS R = R[t]$
    endif
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        $AIFS R S = R[t]$
    else
        $AIFS R NS = R[t]$
    endif
G8.2.6 AMAIR0, Auxiliary Memory Attribute Indirection Register 0

The AMAIR0 characteristics are:

**Purpose**

When using the Long-descriptor format translation tables for stage 1 translations, provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR0.

**Configurations**

AArch32 System register AMAIR0 bits [31:0] are architecturally mapped to AArch64 System register AMAIR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to AMAIR0 are UNDEFINED.

**Attributes**

AMAIR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

This register is RES0 in the following cases:

- When an implementation does not provide any IMPLEMENTATION DEFINED memory attributes.
- When the Long-descriptor translation table format is not used.

If EL3 is implemented and is using AArch32:

- AMAIR0(S) gives the value for memory accesses from Secure state.
- AMAIR0(NS) gives the value for memory accesses from Non-secure states other than Hyp mode.

Any IMPLEMENTATION DEFINED memory attributes are additional qualifiers for the memory locations and must not change the architectured behavior specified by MAIR0 and MAIR1.

In a typical implementation, AMAIR0 and AMAIR1 split into eight one-byte fields, corresponding to the MAIRn.Attr<n> fields, but the architecture does not require them to do so.

**IMPLEMENTATION DEFINED, bits [31:0]**

- IMPLEMENTATION DEFINED.

  The reset behavior of this field is:

  - On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing AMAIR0**

Accesses to this register use the following encodings in the System register encoding space:
**AArch32 System Register Descriptions**

**G8.2 General system control registers**

### MRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0000</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    R[t] = AMAIR0_NS;
  else
    R[t] = AMAIR0;
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    R[t] = AMAIR0_NS;
  else
    R[t] = AMAIR0;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    R[t] = AMAIR0_S;
  else
    R[t] = AMAIR0_NS;
else
  R[t] = AMAIR0_NS;

### MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0000</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    AMAIR0_NS = R[t];
  else
    AMAIR0 = R[t];
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    AMAIR0_NS = R[t];
  else
    AMAIR0 = R[t];
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' && CP15SDISABLE == HIGH then
    UNDEFINED;
elsif SCR.NS == '0' & CP15DISABLE2 == HIGH then
    UNDEFINED;
else
    if SCR.NS == '0' then
        AMAIR0_S = R[t];
    else
        AMAIR0_NS = R[t];
    end if;
G8.2.7   AMAIR1, Auxiliary Memory Attribute Indirection Register 1

The AMAIR1 characteristics are:

**Purpose**

When using the Long-descriptor format translation tables for stage 1 translations, provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR1.

**Configurations**

AArch32 System register AMAIR1 bits [31:0] are architecturally mapped to AArch64 System register AMAIR_EL1[63:32].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to AMAIR1 are UNDEFINED.

When EL3 is using AArch32, write access to AMAIR1(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

**Attributes**

AMAIR1 is a 32-bit register.

**Field descriptions**

This register is RES0 in the following cases:

- When an implementation does not provide any IMPLEMENTATION DEFINED memory attributes.
- When the Long-descriptor translation table format is not used.

If EL3 is implemented and is using AArch32:

- AMAIR1(S) gives the value for memory accesses from Secure state.
- AMAIR1(NS) gives the value for memory accesses from Non-secure states other than Hyp mode.

Any IMPLEMENTATION DEFINED memory attributes are additional qualifiers for the memory locations and must not change the architected behavior specified by MAIR0 and MAIR1.

In a typical implementation, AMAIR0 and AMAIR1 split into eight one-byte fields, corresponding to the MAIRn.Attr<n> fields, but the architecture does not require them to do so.

**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing AMAIR1**

Accesses to this register use the following encodings in the System register encoding space:
### MRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0000</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elshf PSTATE.EL == EL1 then
  if EL2Enabled() & ELUsingAArch32(EL2) & HSTR_EL2.T10 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elshf EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T10 == '1' then
    AArch32.TakeHypTrapException(0x03);
elshf EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.TVM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elshf EL2Enabled() & ELUsingAArch32(EL2) & HCR.TVM == '1' then
    AArch32.TakeHypTrapException(0x03);
elshf HaveEL(EL3) & ELUsingAArch32(EL3) then
    R[t] = AMAIR1_NS;
elshf PSTATE.EL == EL2 then
  if HaveEL(EL3) & ELUsingAArch32(EL3) then
    R[t] = AMAIR1_NS;
elshf PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    R[t] = AMAIR1_S;
elshf PSTATE.EL == EL0 then
  UNDEFINED;
elshf PSTATE.EL == EL1 then
  if EL2Enabled() & ELUsingAArch32(EL2) & HSTR_EL2.T10 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elshf EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T10 == '1' then
    AArch32.TakeHypTrapException(0x03);
elshf EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.TVM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elshf EL2Enabled() & ELUsingAArch32(EL2) & HCR.TVM == '1' then
    AArch32.TakeHypTrapException(0x03);
elshf HaveEL(EL3) & ELUsingAArch32(EL3) then
    AMAIR1_NS = R[t];
elshf PSTATE.EL == EL2 then
  if HaveEL(EL3) & ELUsingAArch32(EL3) then
    AMAIR1_NS = R[t];
elshf PSTATE.EL == EL3 then
  if SCR.NS == '0' & CP15SDISABLE == HIGH then
    UNDEFINED;

### MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0000</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b0001</td>
</tr>
</tbody>
</table>
elsif SCR.NS == '0' && CP15DISABLE2 == HIGH then
    UNDEFINED;
else
    if SCR.NS == '0' then
        AMAIR1_S = R[t];
    else
        AMAIR1_NS = R[t];
    end
G8.2.8 APSR, Application Program Status Register

The APSR characteristics are:

**Purpose**

Hold program status and control information.

--- **Note** ---

Some of the fields in this register are permitted to return the value of the PSTATE field on a read. This is an exception to the general rule that an UNKNOWN field must not return information that cannot be obtained, at the current Privilege level, by an architectured mechanism.

For more information see *The Application Program Status Register, APSR* on page E1-7135.

**Configurations**

This register is present only when AArch32 is supported. Otherwise, direct accesses to APSR are UNDEFINED.

**Attributes**

APSR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>10</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Z</td>
<td>C</td>
<td>V</td>
<td>Q</td>
<td>RES0</td>
<td>GE</td>
<td>RES0</td>
<td>E</td>
<td>A</td>
<td>I</td>
<td>F</td>
<td>M[4:0]</td>
<td>RES0</td>
<td>PAN</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**N, bit [31]**

Negative condition flag. Set to bit[31] of the result of the last flag-setting instruction. If the result is regarded as a two's complement signed integer, then N is set to 1 if the result was negative, and N is set to 0 if the result was positive or zero.

**Z, bit [30]**

Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

**C, bit [29]**

Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.

**V, bit [28]**

Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.

**Q, bit [27]**

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

**Bits [26:23]**

Reserved, RES0.

**PAN, bit [22]**

*When FEAT_PAN is implemented:*

Privileged Access Never. This field is UNKNOWN, but is permitted to return the value of PSTATE.PAN field. On writes, this field is treated as Do-Not-Modify.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bits [21:20]**

Reserved, RES0.

**GE, bits [19:16]**

Greater than or Equal flags, for parallel addition and subtraction.

**Bits [15:10]**

Reserved, RES0.

**E, bit [9]**

Endianness. This field is UNKNOWN, but is permitted to return the value of PSTATE.E field. On writes, this field is treated as Do-Not-Modify.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**A, bit [8]**

SError interrupt mask. This field is UNKNOWN, but is permitted to return the value of PSTATE.A field. On writes, this field is treated as Do-Not-Modify.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**I, bit [7]**

IRQ interrupt mask. This field is UNKNOWN, but is permitted to return the value of PSTATE.I field. On writes, this field is treated as Do-Not-Modify.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**F, bit [6]**

FIQ interrupt mask. This field is UNKNOWN, but is permitted to return the value of PSTATE.F field. On writes, this field is treated as Do-Not-Modify.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [5]**

Reserved, RES0.

**M[4:0], bits [4:0]**

Mode. This field is UNKNOWN, but is permitted to return the value of PSTATE.M[4:0] field. On writes, this field is treated as Do-Not-Modify.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
G8.2.9 ATS12NSOPR, Address Translate Stages 1 and 2 Non-secure Only PL1 Read

The ATS12NSOPR characteristics are:

**Purpose**

Performs stage 1 and 2 address translations as defined for PL1 and the Non-secure state, with permissions as if reading from the given virtual address.

**Configurations**

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to ATS12NSOPR are UNDEFINED.

**Attributes**

ATS12NSOPR is a 32-bit System instruction.

**Field descriptions**

Bits [31:0]

Input address for translation. The resulting address can be read from the PAR. This System instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.

**Executing ATS12NSOPR instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
MCR\{<c>\}{<q>} <coproc>, \{#<opc1>, <Rt>, <CRn>, <CRm>\}, \{#<opc2>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '0' then
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  AArch32.AT(R[t], TranslationStage_12, EL1, ATAccess_Read);
elsif PSTATE.EL == EL3 then
  AArch32.AT(R[t], TranslationStage_12, EL1, ATAccess_Read);
ATS12NSOPW, Address Translate Stages 1 and 2 Non-secure Only PL1 Write

The ATS12NSOPW characteristics are:

**Purpose**

Performs stage 1 and 2 address translations as defined for PL1 and the Non-secure state, with permissions as if writing to the given virtual address.

**Configurations**

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to ATS12NSOPW are UNDEFINED.

**Attributes**

ATS12NSOPW is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th>Input address for translation</th>
</tr>
</thead>
</table>

Input address for translation. The resulting address can be read from the PAR. This System instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.

**Executing ATS12NSOPW instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[ MCR<\{c\}\{q\}> <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2}> \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  else
    UNDEFINED;
  elsif PSTATE.EL == EL2 then
    AArch32.AT(R[t], TranslationStage_12, EL1, ATAccess_Write);
elsif PSTATE.EL == EL3 then
  AArch32.AT(R[t], TranslationStage_12, EL1, ATAccess_Write);
G8.2.11  ATS12NSOUR, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Read

The ATS12NSOUR characteristics are:

**Purpose**

Performs stage 1 and 2 address translations as defined for PL0 and the Non-secure state, with permissions as if reading from the given virtual address.

**Configurations**

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to ATS12NSOUR are UNDEFINED.

**Attributes**

ATS12NSOUR is a 32-bit System instruction.

**Field descriptions**

Bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.

**Executing ATS12NSOUR instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1111</td>
<td>0b1000</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        AArch32.AT(R[t], TranslationStage_12, EL0, ATAccess_Read);
    elseif PSTATE.EL == EL3 then
        AArch32.AT(R[t], TranslationStage_12, EL0, ATAccess_Read);
```
G8.2.12  ATS12NSOUW, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Write

The ATS12NSOUW characteristics are:

**Purpose**

Performs stage 1 and 2 address translations as defined for PL0 and the Non-secure state, with permissions as if writing to the given virtual address.

**Configurations**

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to ATS12NSOUW are UNDEFINED.

**Attributes**

ATS12NSOUW is a 32-bit System instruction.

**Field descriptions**

![Input address for translation](image)

**Bits [31:0]**

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.

**Executing ATS12NSOUW instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[ \text{MCR}\{c\}\{q\} \ <\text{coproc}>\, \{\#<op1>\}, \ <\text{Rt}>\, \ <\text{CRn}>\, \ <\text{CRm}>\{, \{\#<op2>\}\} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1011</td>
<td>0b1000</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & & !ELUsingAArch32(EL2) & & HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() & & ELUsingAArch32(EL2) & & HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) & & SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL3) & & SCR_EL3.<NS> == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        AArch32.AT(R[t], TranslationStage_12, EL0, ATAccess_Write);
    elsif PSTATE.EL == EL3 then
        AArch32.AT(R[t], TranslationStage_12, EL0, ATAccess_Write);

else
    UNDEFINED;
G8.2.13   ATS1CPR, Address Translate Stage 1 Current state PL1 Read

The ATS1CPR characteristics are:

Purpose

Performs stage 1 address translation as defined for PL1 and the current Security state, with permissions as if reading from the given virtual address.

Configurations

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ATS1CPR are UNDEFINED.

Attributes

ATS1CPR is a 32-bit System instruction.

Field descriptions

Bits [31:0]

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing ATS1CPR instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[ MCR\{c\}\{q\} \text{ coproc, \{#\}opc1, \text{ \textless }Rt, \text{ \textless CRn}, \text{ \textless CRm}\{, \text{ \{#\}opc2}\} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then

UNDEFINED;

elsif PSTATE.EL == EL1 then

if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then

AArch64.AArch32SystemAccessTrap(EL2, 0x03);

elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then

AArch32.TakeHypTrapException(0x03);

else

AArch32.AT(R[t], TranslationStage_1, EL1, ATAccess_Read);

elsif PSTATE.EL == EL2 then

AArch32.AT(R[t], TranslationStage_1, EL1, ATAccess_Read);

elsif PSTATE.EL == EL3 then

if SCR.NS == '0' then

AArch32.AT(R[t], TranslationStage_1, EL3, ATAccess_Read);

else

AArch32.AT(R[t], TranslationStage_1, EL1, ATAccess_Read);

else

AArch32.AT(R[t], TranslationStage_1, EL1, ATAccess_Read);
G8.2.14   ATS1CPRP, Address Translate Stage 1 Current state PL1 Read PAN

The ATS1CPRP characteristics are:

**Purpose**

Performs a stage 1 address translation at PL1 and in the current Security state, where the value of PSTATE.PAN determines if a read from a location will generate a Permission fault for a privileged access.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32 and FEAT_PAN2 is implemented. Otherwise, direct accesses to ATS1CPRP are UNDEFINED.

**Attributes**

ATS1CPRP is a 32-bit System instruction.

**Field descriptions**

![Input address for translation](image)

**Bits [31:0]**

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

**Executing ATS1CPRP instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} {<coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    AArch32.AT(R[t], TranslationStage_1, EL1, ATAccess_ReadPAN);
  endif
elsif PSTATE.EL == EL2 then
  AArch32.AT(R[t], TranslationStage_1, EL1, ATAccess_ReadPAN);
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    AArch32.AT(R[t], TranslationStage_1, EL3, ATAccess_ReadPAN);
  else
    AArch32.AT(R[t], TranslationStage_1, EL1, ATAccess_ReadPAN);
  endif
```
G8.2.15 ATS1CPW, Address Translate Stage 1 Current state PL1 Write

The ATS1CPW characteristics are:

**Purpose**

Performs stage 1 address translation as defined for PL1 and the current Security state, with permissions as if writing to the given virtual address.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ATS1CPW are undefined.

**Attributes**

ATS1CPW is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Input address for translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td></td>
</tr>
</tbody>
</table>

**Input address for translation**

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

**Executing ATS1CPW instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{MCR}\{<c>\}{<q>\} <coproc>, \{#<opc1>\}, <Rt>, <CRn>, <CRm>{, \{#<opc2>\}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if \(\text{PSTATE.EL} = \text{EL0}\) then

UNDEFINED;

elsif \(\text{PSTATE.EL} = \text{EL1}\) then

if \(\text{EL2Enabled()} \&\& \text{ELUsingAArch32(EL2)} \&\& \text{HSTR.EL2.T7} = '1'\) then

\(\text{AArch64.AArch32SystemAccessTrap(EL2, 0x03);}\)

elsif \(\text{EL2Enabled()} \&\& \text{ELUsingAArch32(EL2)} \&\& \text{HSTR.T7} = '1'\) then

\(\text{AArch32.TakeHypTrapException(0x03);}\)

else

\(\text{AArch32.AT(R[t], TranslationStage_1, EL1, ATAccess_Write);}\)

elsif \(\text{PSTATE.EL} = \text{EL2}\) then

\(\text{AArch32.AT(R[t], TranslationStage_1, EL1, ATAccess_Write);}\)

elsif \(\text{PSTATE.EL} = \text{EL3}\) then

if \(\text{SCR.NS} = '0'\) then

\(\text{AArch32.AT(R[t], TranslationStage_1, EL3, ATAccess_Write);}\)

else

\(\text{AArch32.AT(R[t], TranslationStage_1, EL1, ATAccess_Write);}\)
G8.2.16 ATS1CPWP, Address Translate Stage 1 Current state PL1 Write PAN

The ATS1CPWP characteristics are:

**Purpose**

Performs a stage 1 address translation at PL1 and in the current Security state, where the value of PSTATE.PAN determines if a write to the location will generate a Permission fault for a privileged access.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32 and FEAT_PAN2 is implemented. Otherwise, direct accesses to ATS1CPWP are UNDEFINED.

**Attributes**

ATS1CPWP is a 32-bit System instruction.

**Field descriptions**

![Input address for translation](Input address for translation)

**Bits [31:0]**

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

**Executing ATS1CPWP instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    AArch32.AT(R[t], TranslationStage_1, EL1, ATAccess_WritePAN);
elsif PSTATE.EL == EL2 then
  AArch32.AT(R[t], TranslationStage_1, EL1, ATAccess_WritePAN);
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    AArch32.AT(R[t], TranslationStage_1, EL3, ATAccess_WritePAN);
  else
    AArch32.AT(R[t], TranslationStage_1, EL1, ATAccess_WritePAN);
G8.2.17  ATS1CUR, Address Translate Stage 1 Current state Unprivileged Read

The ATS1CUR characteristics are:

**Purpose**

Performs stage 1 address translation as defined for PL0 and the current Security state, with permissions as if reading from the given virtual address.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ATS1CUR are UNDEFINED.

**Attributes**

ATS1CUR is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Input address for translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits [31:0]</td>
</tr>
</tbody>
</table>

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

**Executing ATS1CUR instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
MCR\{<c>\}{<q>}<coproc>, \{#\}<opc1>, <Rt>, <CRn>, <CRm>\{, \{#\}<opc2>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then

UNDEFINED;

elsif PSTATE.EL == EL1 then

if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1'

AArch64.AArch32SystemAccessTrap(EL2, 0x03);

elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1'

AArch32.TakeHypTrapException(0x03);

else

AArch32.AT(R[t], TranslationStage_1, EL0, ATAccess_Read);

elsif PSTATE.EL == EL2 then

AArch32.AT(R[t], TranslationStage_1, EL0, ATAccess_Read);

elsif PSTATE.EL == EL3 then

AArch32.AT(R[t], TranslationStage_1, EL0, ATAccess_Read);
G8.2.18   ATS1CUW, Address Translate Stage 1 Current state Unprivileged Write

The ATS1CUW characteristics are:

**Purpose**

Performs stage 1 address translation as defined for PL0 and the current Security state, with permissions as if writing to the given virtual address.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ATS1CUW are UNDEFINED.

**Attributes**

ATS1CUW is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th>Input address for translation</th>
</tr>
</thead>
</table>

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

**Executing ATS1CUW instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
MCR\{<c>\}{<q>} \{<opc1>,<Rt>,<CRn>,<CRm>,\{<opc2}\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b011</td>
<td>0b1000</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        AArch32.AT(R[t], TranslationStage_1, EL0, ATAccess_Write);
    endif
elsif PSTATE.EL == EL2 then
    AArch32.AT(R[t], TranslationStage_1, EL0, ATAccess_Write);
elsif PSTATE.EL == EL3 then
    AArch32.AT(R[t], TranslationStage_1, EL0, ATAccess_Write);
G8.2.19   ATS1HR, Address Translate Stage 1 Hyp mode Read

The ATS1HR characteristics are:

**Purpose**

Performs stage 1 address translation as defined for PL2 and the Non-secure state, with permissions as if reading from the given virtual address.

**Configurations**

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to ATS1HR are UNDEFINED.

**Attributes**

ATS1HR is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th>Input address for translation</th>
</tr>
</thead>
</table>

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the translation.

**Executing ATS1HR instruction**

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{MCR}\{<c>\}{<q>}\{,#\}<\text{coproc}>, {,#}<\text{opc1}>, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>{,#<\text{opc2}>}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.EL2.T7 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
AArch32.TakeHypTrapException(0x03);
else
UNDEFINED;
elsif PSTATE.EL == EL2 then
AArch32.AT(R[t], TranslationStage_1, EL2, ATAccess_Read);
elsif PSTATE.EL == EL3 then
    AArch32.AT(R[t], TranslationStage_1, EL2, ATAccess_Read);
G8.2.20   ATS1HW, Address Translate Stage 1 Hyp mode Write

The ATS1HW characteristics are:

**Purpose**

Performs stage 1 address translation as defined for PL2 and the Non-secure state, with permissions as if writing to the given virtual address.

**Configurations**

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to ATS1HW are UNDEFINED.

**Attributes**

ATS1HW is a 32-bit System instruction.

**Field descriptions**

**Bits [31:0]**

Input address for translation. The resulting address can be read from the PAR.

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the translation.

**Executing ATS1HW instruction**

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{MCR\{<c>\}<q>\{<opc1>, \{<Rt>, \{<CRn>, \{CRm\{<opc2>\}\}\}\}\]\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
    endif
elsif PSTATE.EL == EL2 then
    AArch32.AT(R[t], TranslationStage_1, EL2, ATAccess_Write);
elsif PSTATE_EL == EL3 then
    AArch32.AT(R[t], TranslationStage_1, EL2, ATAccess_Write);
G8.2.21  BPIALL, Branch Predictor Invalidate All

The BPIALL characteristics are:

**Purpose**

Invalidate all entries from branch predictors.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to BPIALL are UNDEFINED.

In an implementation where the branch predictors are architecturally invisible, this instruction can execute as a NOP.

**Attributes**

BPIALL is a 32-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

**Executing BPIALL instruction**

The PE ignores the value of <Rt>. Software does not have to write a value to this register before issuing this instruction.

When HCR.FB is 1, at Non-secure EL1 this instruction executes as a BPIALLIS.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0101</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.FB == '1' then
    BPIALLIS();
  else
    BPIALL();
elsif PSTATE.EL == EL2 then
  BPIALL();
else if PSTATE.EL == EL3 then
  BPIALL();
```
G8.2.22 BPIALLIS, Branch Predictor Invalidate All, Inner Shareable

The BPIALLIS characteristics are:

**Purpose**

Invalidate all entries from branch predictors Inner Shareable.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to BPIALLIS are UNDEFINED.

In an implementation where the branch predictors are architecturally invisible, this instruction can execute as a NOP.

**Attributes**

BPIALLIS is a 32-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

**Executing BPIALLIS instruction**

The PE ignores the value of <Rt>. Software does not have to write a value to this register before issuing this instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{MCR}\{<c>\},\{<q>\} \text{ <coproc>, \{#<opc1>, <Rt>, <CRn>, <CRm>, \{#<opc2>\}}
\]

```
coproc opc1 CRn CRm opc2
0b1111 0b000 0b0111 0b0001 0b110
```

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        BPIALLIS();
    elsif PSTATE.EL == EL2 then
        BPIALLIS();
    elsif PSTATE.EL == EL3 then
        BPIALLIS();
```
G8.2.23  **BPIMVA, Branch Predictor Invalidate by VA**

The BPIMVA characteristics are:

**Purpose**

Invalidate virtual address from branch predictors.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to BPIMVA are UNDEFINED.

In an implementation where the branch predictors are architecturally invisible, this instruction can execute as a NOP.

**Attributes**

BPIMVA is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th>Virtual address to use</th>
</tr>
</thead>
</table>

**Executing BPIMVA instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
MCR\{<c}>\{<q}> \langle coproc \rangle, \langle \# \langle opc1 \rangle \rangle, \langle Rt \rangle, \langle CRn \rangle, \langle CRm \rangle, \langle \# \langle opc2 \rangle \rangle
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0101</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
   UNDEFINED;
elsif PSTATE.EL == EL1 then
   if EL2Enabled() \&\& ELUsingAArch32(EL2) \&\& HSTR.EL2.T7 == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
   elsif EL2Enabled() \&\& ELUsingAArch32(EL2) \&\& HSTR.T7 == '1' then
      AArch32.TakeHypTrapException(0x03);
   else
      BPIMVA(R[t]);
   endif
elsif PSTATE.EL == EL2 then
   BPIMVA(R[t]);
elsif PSTATE.EL == EL3 then
   BPIMVA(R[t]);
G8.2.24 CCSIDR, Current Cache Size ID Register

The CCSIDR characteristics are:

**Purpose**

Provides information about the architecture of the currently selected cache. When FEAT_CCIDX is implemented, this register is used in conjunction with CCSIDR2.

**Configurations**

AArch32 System register CCSIDR bits [31:0] are architecturally mapped to AArch64 System register CCSIDR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to CCSIDR are UNDEFINED.

The implementation includes one CCSIDR for each cache that it can access. CSSEL and the Security state select which Cache Size ID Register is accessible.

**Attributes**

CCSIDR is a 32-bit register.

**Field descriptions**

*When FEAT_CCIDX is implemented:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td><strong>RES0</strong></td>
</tr>
<tr>
<td>24-23</td>
<td><strong>Associativity</strong></td>
</tr>
<tr>
<td>2-0</td>
<td><strong>LineSize</strong></td>
</tr>
</tbody>
</table>

--- Note ---

The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

---

**Bits [31:24]**

Reserved, RES0.

**Associativity, bits [23:3]**

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

**LineSize, bits [2:0]**

(Log2(Number of bytes in cache line)) - 4. For example:

- For a line length of 16 bytes: Log2(16) = 4, LineSize entry = 0. This is the minimum line length.
- For a line length of 32 bytes: Log2(32) = 5, LineSize entry = 1.

--- Note ---

The C++ 17 specification has two defined parameters relating to the granularity of memory that does not interfere. For generic software and tools, Arm will set the hardware_destructive_interference_size parameter to 256 bytes and the hardware_constructive_interference_size parameter to 64 bytes.
**Otherwise:**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>13</th>
<th>10</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNKNOWN</td>
<td>NumSets</td>
<td>Associativity</td>
<td>LineSize</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

--- **Note** ---

The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

---

**Bits [31:28]**

Reserved, UNKNOWN.

**NumSets, bits [27:13]**

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

**Associativity, bits [12:3]**

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

**LineSize, bits [2:0]**

(Log₂(Number of bytes in cache line)) - 4. For example:

For a line length of 16 bytes: Log₂(16) = 4, LineSize entry = 0. This is the minimum line length.

For a line length of 32 bytes: Log₂(32) = 5, LineSize entry = 1.

--- **Note** ---

The C++ 17 specification has two defined parameters relating to the granularity of memory that does not interfere. For generic software and tools, Arm will set the hardware-destructive interference size parameter to 256 bytes and the hardware-constructive interference size parameter to 64 bytes.

---

**Accessing CCSIDR**

If CSSEL.R.[Level, InD] is programmed to a cache level that is not implemented, then on a read of the CCSIDR the behavior is CONSTRAINED UNPREDICTABLE, and can be one of the following:

- The CCSIDR read is treated as NOP.
- The CCSIDR read is UNDEFINED.
- The CCSIDR read returns an UNKNOWN value.

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TID4 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        AArch32.TakeHypTrapException(0x03);
    end if;
elsif PSTATE.EL == EL2 then
    R[t] = CCSIDR;
elsif PSTATE.EL == EL3 then
    R[t] = CCSIDR;
else
    R[t] = CCSIDR;
end if;

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>
G8.2.25  CCSIDR2, Current Cache Size ID Register 2

The CCSIDR2 characteristics are:

**Purpose**

Provides information about the architecture of the currently selected cache.

**Configurations**

AArch32 System register CCSIDR2 bits [31:0] are architecturally mapped to AArch64 System register CCSIDR2_EL1[31:0].

This register is present only when EL1 is capable of using AArch32 and FEAT_CCIDX is implemented. Otherwise, direct accesses to CCSIDR2 are UNDEFINED.

The implementation includes one CCSIDR2 for each cache that it can access. CSSELR and the Security state select which Cache Size ID Register is accessible.

**Attributes**

CCSIDR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0</td>
</tr>
<tr>
<td>24</td>
<td>NumSets</td>
</tr>
</tbody>
</table>

**Bits [31:24]**

Reserved, RES0.

**NumSets, bits [23:0]**

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

**Accessing CCSIDR2**

If CSSELR.{Level, InD} is programmed to a cache level that is not implemented, then on a read of the CCSIDR2 the behavior is CONSTRAINED UNPREDICTABLE, and can be one of the following:

- The CCSIDR2 read is treated as NOP.
- The CCSIDR2 read is UNDEFINED.
- The CCSIDR2 read returns an UNKNOWN value.

Accesses to this register use the following encodings in the System register encoding space:

**MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rp>, <CRn>, <CRm>{, {#}<opc2>**

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() AndAlso ELUsingAArch32(EL2) AndAlso HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() AndAlso ELUsingAArch32(EL2) AndAlso HSTR.T0 == '1' then

AArch32.TakeHypTrapException(0x03);
elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then 
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID4 == '1' then 
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID2 == '1' then 
  AArch32.TakeHypTrapException(0x03);
elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TID4 == '1' then 
  AArch32.TakeHypTrapException(0x03);
else
  R[t] = CCSIDR2;
elsif PSTATE.EL == EL2 then
  R[t] = CCSIDR2;
elsif PSTATE.EL == EL3 then
  R[t] = CCSIDR2;
G8.2.26 CFPRCTX, Control Flow Prediction Restriction by Context

The CFPRCTX characteristics are:

Purpose
Control Flow Prediction Restriction by Context applies to all Control Flow Prediction Resources that predict execution based on information gathered within the target execution context or contexts. Control flow predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control speculative execution occurring after the instruction is complete and synchronized.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note
This instruction does not require the invalidation of prediction structures so long as the behavior described for completion of this instruction is met by the implementation.

On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

Configurations
This instruction is present only when AArch32 is supported and FEAT_SPECRES is implemented. Otherwise, direct accesses to CFPRCTX are UNDEFINED.

Attributes
CFPRCTX is a 32-bit System instruction.

Field descriptions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>28</td>
<td>GVMID, bit [27]</td>
</tr>
<tr>
<td>27</td>
<td>Execution of this instruction applies to all VMIDs or a specified VMID.</td>
</tr>
<tr>
<td>26</td>
<td>Applies to specified VMID for an EL0 or EL1 target execution context.</td>
</tr>
<tr>
<td>25</td>
<td>Applies to all VMIDs for an EL0 or EL1 target execution context.</td>
</tr>
<tr>
<td>24</td>
<td>For target execution contexts other than EL0 or EL1, this field is RES0.</td>
</tr>
<tr>
<td>23</td>
<td>If the instruction is executed at EL0 or EL1, this field has an Effective value of 0.</td>
</tr>
<tr>
<td>22</td>
<td>If EL2 is not implemented or not enabled for the target Security state, this field is RES0.</td>
</tr>
<tr>
<td>16</td>
<td>NS, bit [26]</td>
</tr>
<tr>
<td>15</td>
<td>Security State.</td>
</tr>
<tr>
<td>14</td>
<td>Secure state.</td>
</tr>
<tr>
<td>13</td>
<td>Non-secure state.</td>
</tr>
<tr>
<td>9</td>
<td>ASID</td>
</tr>
<tr>
<td>0</td>
<td>GASID</td>
</tr>
</tbody>
</table>
EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

- **0b00**: EL0.
- **0b01**: EL1.
- **0b10**: EL2.
- **0b11**: EL3.

If the instruction is executed at an Exception level lower than the specified level, this instruction is treated as a NOP.

VMID, bits [23:16]

- Only applies when bit[27] is 0 and the target execution context is either:
  - EL1.
  - EL0 when (HCR_EL2.E2H==0 or HCR_EL2.TGE==0) or EL2 is using AArch32 state.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0 and (HCR_EL2.E2H==0 or HCR_EL2.TGE==0 or ELUsingAArch32(EL2)), this field is treated as the current VMID.

When the instruction is executed at EL0 and (HCR_EL2.E2H==1 and HCR_EL2.TGE==1 and !ELUsingAArch32(EL2)), this field is ignored.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

Bits [15:9]

Reserved, RES0.

GASID, bit [8]

Execution of this instruction applies to all ASIDs or a specified ASID.

- **0b0**: Applies to specified ASID for an EL0 target execution context.
- **0b1**: Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field is treated as 0.

ASID, bits [7:0]

- Only applies for an EL0 target execution context and when bit[8] is 0.
- Otherwise, this field is RES0.

When the instruction is executed at EL0, this field is treated as the current ASID.

**Executing CFPRCTX instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0011</td>
<td>0b100</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
  if !ELUsingAArch32(EL1) && !ELEnabled() && HCR_EL2.E2H,TGE == '11' && SCTLR_EL1.EnRCTX == '0' then
    if ELEnabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
```

G8-9378  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  Non-Confidential
else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif ELUsingAArch32(EL1) && SCTLR.EnRCTX == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
        AArch32.TakeHypTrapException(0x00);
    else
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif PSTATE.EL == EL1 then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
            AArch32.TakeHypTrapException(0x03);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.NV == '1' then
            AArch64.SystemAccessTrap(EL2, 0x03);
        else
            AArch32.RestrictPrediction(R[t], RestrictType_ControlFlow);
        else
            AArch32.RestrictPrediction(R[t], RestrictType_ControlFlow);
    elsif PSTATE.EL == EL2 then
        AArch32.RestrictPrediction(R[t], RestrictType_ControlFlow);
    elsif PSTATE.EL == EL3 then
        AArch32.RestrictPrediction(R[t], RestrictType_ControlFlow);
    else
        AArch32.RestrictPrediction(R[t], RestrictType_ControlFlow);
    elsif PSTATE.EL == EL3 then
        AArch32.RestrictPrediction(R[t], RestrictType_ControlFlow);
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
            AArch32.TakeHypTrapException(0x03);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && !HaveEL(EL3) ||
            SCR_EL3.FGEn == '1') && HFGITR_EL2.CFPRCTX == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.EnRCTX == '0' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch32.RestrictPrediction(R[t], RestrictType_ControlFlow);
        else
            AArch32.RestrictPrediction(R[t], RestrictType_ControlFlow);
    else
        AArch32.RestrictPrediction(R[t], RestrictType_ControlFlow);
    elsif PSTATE.EL == EL2 then
        AArch32.RestrictPrediction(R[t], RestrictType_ControlFlow);
    elsif PSTATE.EL == EL3 then
        AArch32.RestrictPrediction(R[t], RestrictType_ControlFlow);
G8.2.27   CLIDR, Cache Level ID Register

The CLIDR characteristics are:

**Purpose**

Identifies the type of cache, or caches, that are implemented at each level and can be managed using the architected cache maintenance instructions that operate by set/way, up to a maximum of seven levels. Also identifies the Level of Coherence (LoC) and Level of Unification (LoU) for the cache hierarchy.

**Configurations**

AArch32 System register CLIDR bits [31:0] are architecturally mapped to AArch64 System register CLIDR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to CLIDR are UNDEFINED.

**Attributes**

CLIDR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-30</td>
<td>ICB, Inner cache boundary. This field indicates the boundary for caching Inner Cacheable memory regions.</td>
</tr>
<tr>
<td>29-27</td>
<td>LoUU, Level of Unification Uniprocessor for the cache hierarchy. For a description of the values of this field, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions on page G4-9121.</td>
</tr>
<tr>
<td>26-24</td>
<td>LoC, Level of Coherence for the cache hierarchy. For a description of the values of this field, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions on page G4-9121.</td>
</tr>
<tr>
<td>23-21</td>
<td>LoUIS, Level of Unification Inner Shareable for the cache hierarchy. For a description of the values of this field, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions on page G4-9121.</td>
</tr>
<tr>
<td>20-18</td>
<td>Ctype7, Ctype6, Ctype5, Ctype4</td>
</tr>
<tr>
<td>17-15</td>
<td>Ctype3, Ctype2, Ctype1</td>
</tr>
<tr>
<td>14-12</td>
<td>Ctype4</td>
</tr>
<tr>
<td>11-9</td>
<td>Ctype5</td>
</tr>
<tr>
<td>8-6</td>
<td>Ctype6</td>
</tr>
<tr>
<td>5-3</td>
<td>Ctype7</td>
</tr>
<tr>
<td>0</td>
<td>Ctype8</td>
</tr>
</tbody>
</table>

**Note**

When FEAT_S2FWB is implemented, the architecture requires that this field is zero so that no levels of data cache need to be cleaned in order to manage coherency with instruction fetches.
Note

When FEAT_S2FWB is implemented, the architecture requires that this field is zero so that no levels of data cache need to be cleaned in order to manage coherency with instruction fetches.

Ctype<\text{n}>, bits [3(n-1)+2:3(n-1)], for \text{n} = 7 to 1

Cache Type fields. Indicate the type of cache that is implemented and can be managed using the architectured cache maintenance instructions that operate by set/way at each level, from Level 1 up to a maximum of seven levels of cache hierarchy.

- 0b000: No cache.
- 0b001: Instruction cache only.
- 0b010: Data cache only.
- 0b011: Separate instruction and data caches.
- 0b100: Unified cache.

All other values are reserved.

If software reads the Cache Type fields from Ctype1 upwards, once it has seen a value of 000, no caches that can be managed using the architectured cache maintenance instructions that operate by set/way exist at further-out levels of the hierarchy. So, for example, if Ctype3 is the first Cache Type field with a value of 000, the values of Ctype4 to Ctype7 must be ignored.

Accessing CLIDR

Accesses to this register use the following encodings in the System register encoding space:

\[ \text{MRC\{<c>\}{<q>} \text{<coproc>, \{#<opc1>, <Rt>, <CRn>, <CRm>{, \{#<opc2}\}}}} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b001</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TID4 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID2 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TID4 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    R[t] = CLIDR;
  endif
elsif PSTATE.EL == EL2 then
  R[t] = CLIDR;
elsif PSTATE.EL == EL3 then
  R[t] = CLIDR;
G8.2.28 CONTEXTIDR, Context ID Register

The CONTEXTIDR characteristics are:

Purpose

Identifies the current Process Identifier and, when using the Short-descriptor translation table format, the Address Space Identifier.

The value of the whole of this register is called the Context ID and is used by:

- The debug logic, for Linked and Unlinked Context ID matching.
- The trace logic, to identify the current process.

The significance of this register is for debug and trace use only.

Configurations

AArch32 System register CONTEXTIDR bits [31:0] are architecturally mapped to AArch64 System register CONTEXTIDR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to CONTEXTIDR are UNDEFINED.

The register format depends on whether address translation is using the Long-descriptor or the Short-descriptor translation table format.

Attributes

CONTEXTIDR is a 32-bit register.

Field descriptions

When TTBCR.EAE == 0:

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCID</td>
<td>ASID</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PROCID, bits [31:8]

Process Identifier. This field must be programmed with a unique value that identifies the current process.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ASID, bits [7:0]

Address Space Identifier. This field is programmed with the value of the current ASID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When TTBCR.EAE == 1:

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCID</td>
<td></td>
</tr>
</tbody>
</table>

PROCID, bits [31:0]

Process Identifier. This field must be programmed with a unique value that identifies the current process.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CONTEXTIDR**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC\{<c>\}{<q>} \coproc, \{#<opc1>, <Rt>, <CRn>, <CRm}\{, \{#<opc2>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

- if PSTATE_EL == EL0 then
  UNDEFINED;
- elsif PSTATE_EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    \( R[t] = \text{CONTEXTIDR\_NS} \);
  else
    \( R[t] = \text{CONTEXTIDR} \);
  endif
- elsif PSTATE_EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    \( R[t] = \text{CONTEXTIDR\_NS} \);
  else
    \( R[t] = \text{CONTEXTIDR} \);
  endif
- elsif PSTATE_EL == EL3 then
  if SCR.NS == '0' then
    \( R[t] = \text{CONTEXTIDR\_S} \);
  else
    \( R[t] = \text{CONTEXTIDR\_NS} \);
  endif

\[
MCR\{<c>\}{<q>} \coproc, \{#<opc1>, <Rt>, <CRn>, <CRm}\{, \{#<opc2>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

- if PSTATE_EL == EL0 then
  UNDEFINED;
- elsif PSTATE_EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    \( \text{CONTEXTIDR\_NS} = R[t] \);
  else

CONTEXTIDR = R[t];
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    CONTEXTIDR_NS = R[t];
  else
    CONTEXTIDR = R[t];
  endif
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    CONTEXTIDR_S = R[t];
  else
    CONTEXTIDR_NS = R[t];
  endif
G8.2.29   **CP15DMB, Data Memory Barrier System instruction**

The CP15DMB characteristics are:

**Purpose**

Performs a Data Memory Barrier.

Arm deprecates any use of this System instruction, and strongly recommends that software use the DMB instruction instead.

**Configurations**

This instruction is present only when AArch32 is supported. Otherwise, direct accesses to CP15DMB are UNDEFINED.

**Attributes**

CP15DMB is a 32-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

**Executing CP15DMB instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
 MCR{<c}>{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b101</td>
<td></td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.CP15BEN == '0' then
  UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.CP15BEN == '0' then
  UNDEFINED;
elsif ELUsingAArch32(EL1) && SCTLR.CP15BEN == '0' then
  UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && HSTR_EL2.T7 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && HSTR.T7 == '1' then
  AArch32.TakeHypTrapException(0x03);
else
  CP15DMB();
elsif PSTATE.EL == EL1 then
if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif SCTLR.CP15BEN == '0' then
  UNDEFINED;
else
  CP15DMB();
elsif PSTATE.EL == EL2 then
if HSCTLR.CP15BEN == '0' then
  UNDEFINED;
else
  CP15DMB();
elsif PSTATE_EL == EL3 then
  if SCTLR.CP15BEN == '0' then
    UNDEFINED;
  else
    CP15DMB();
  endif
endif
G8.2.30 CP15DSB, Data Synchronization Barrier System instruction

The CP15DSB characteristics are:

**Purpose**

Performs a Data Synchronization Barrier.

Arm deprecates any use of this System instruction, and strongly recommends that software use the DSB instruction instead.

**Configurations**

This instruction is present only when AArch32 is supported. Otherwise, direct accesses to CP15DSB are UNDEFINED.

**Attributes**

CP15DSB is a 32-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

**Executing CP15DSB instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1010</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !ELUsingAArch32(EL1) && !EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL1.CP15BEN == '0'
    then UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.CP15BEN == '0'
    then UNDEFINED;
  elsif EL2Enabled() && AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && AArch32.TakeHypTrapException(0x03);
  else CP15DSB();
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && AArch32.TakeHypTrapException(0x03);
  elsif HSTR_EL2.T7 == '1' then
    UNDEFINED;
  else CP15DSB();
elsif PSTATE.EL == EL2 then
  if HSCTLR.CP15BEN == '0' then
    UNDEFINED;
  else CP15DSB();
elsif PSTATE_EL == EL3 then
  if SCTLR.CP15BEN == '0' then
    UNDEFINED;
  else
    CP15DSB();

G8.2.31 CP15ISB, Instruction Synchronization Barrier System instruction

The CP15ISB characteristics are:

**Purpose**

Performs an Instruction Synchronization Barrier.

Arm deprecates any use of this System instruction, and strongly recommends that software use the ISB instruction instead.

**Configurations**

This instruction is present only when AArch32 is supported. Otherwise, direct accesses to CP15ISB are UNDEFINED.

**Attributes**

CP15ISB is a 32-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

**Executing CP15ISB instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
MCR{<c>}{<q>} \{<coproc>, {#}<opc1>\}, \{<CRn>, <CRm>{, {#}<opc2>}\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0101</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.CP15BEN == '0' then
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.CP15BEN == '0' then
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL1) && SCTLR.CP15BEN == '0' then
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    CP15ISB();
  elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL1) && HSTR_EL2.T7 == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
      AArch32.TakeHypTrapException(0x03);
    elsif SCTLR.CP15BEN == '0' then
      UNDEFINED;
    else
      CP15ISB();
  elsif PSTATE.EL == EL2 then
    if HSCTLR.CP15BEN == '0' then
      UNDEFINED;
    else
      CP15ISB();
  end

elsif PSTATE_EL == EL3 then
    if SCTLR.CP15BEN == '0' then
        UNDEFINED;
    else
        CP15ISB();
G8.2.32 CPACR, Architectural Feature Access Control Register

The CPACR characteristics are:

**Purpose**

Controls access to trace, and to Advanced SIMD and floating-point functionality from EL0, EL1, and EL3.

In an implementation that includes EL2, the CPACR has no effect on instructions executed at EL2.

**Configurations**

AArch32 System register CPACR bits [31:0] are architecturally mapped to AArch64 System register CPACR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to CPACR are UNDEFINED.

Bits in the NSACR control Non-secure access to the CPACR fields. See the field descriptions for more information.

--- Note ---

In the register field descriptions, controls are described as applying at specified Privilege levels. This is because, in Secure state, a PL1 control:

- Applies to execution in a Secure EL3 mode when EL3 is using AArch32.
- Applies to execution in a Secure EL1 mode when EL3 is using AArch64.

See Security state, Exception levels, and AArch32 execution privilege on page G1-8908.

**Attributes**

CPACR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASEDIS</td>
<td>Enables PL0 and PL1 execution of Advanced SIMD instructions.</td>
</tr>
<tr>
<td>TRCDIS</td>
<td></td>
</tr>
</tbody>
</table>

**ASEDIS, bit [31]**

Disables PL0 and PL1 execution of Advanced SIMD instructions.

- **0b0**: This control permits execution of Advanced SIMD instructions at PL0 and PL1.
- **0b1**: All instruction encodings that are Advanced SIMD instruction encodings, but are not also floating-point instruction encodings, are UNDEFINED at PL0 and PL1.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0. Otherwise, it is IMPLEMENTATION DEFINED whether this field is implemented as a RW field.

If it is not implemented as a RW field, it is RAZ/WI.

If EL3 is implemented and is using AArch32, and the value of NSACR.NSASEDIS is 1, this field behaves as RAO/WI in Non-secure state, regardless of its actual value. This applies even if the field is implemented as RAZ/WI.

For the list of instructions affected by this field, see Controls of Advanced SIMD operation that do not apply to floating-point operation on page E1-7153.

See the description of CPACR.cp10 for a list of other controls that can disable or trap execution of Advanced SIMD instructions in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.
G8.2 General system control registers

Bits [30:29]

Reserved, RES0.

TRCDIS, bit [28]
Traps PL0 and PL1 System register accesses to all implemented trace registers to Undefined mode.
0b0 This control has no effect on PL0 and PL1 System register accesses to trace registers.
0b1 PL0 and PL1 System register accesses to all implemented trace registers are trapped to Undefined mode.

If the implementation does not include a trace unit, or does not include a System register interface to the trace unit registers, this field is RES0. Otherwise, it is IMPLEMENTATION DEFINED whether this field is implemented as a RW field. If it is not implemented as a RW field, it is RAZ/WI.

If EL3 is implemented and is using AArch32, and the value of NSACR.NSTRCDIS is 1, this field behaves as RAO/WI in Non-secure state, regardless of its actual value. This applies even if the field is implemented as RAZ/WI.

Note
- The ETMv4 architecture and ETE do not permit EL0 to access the trace registers. If the trace unit implements FEAT_ETMv4 or FEAT_ETE, EL0 accesses to the trace registers are UNDEFINED.
- The Arm architecture does not provide traps on trace register accesses through the optional memory-mapped external debug interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:24]

Reserved, RES0.

cp11, bits [23:22]
The value of this field is ignored. If this field is programmed with a different value to the cp10 field then this field is UNKNOWN on a direct read of the CPACR.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

In Non-secure state, if EL3 is implemented and is using AArch32, when the value of NSACR.cp10 is 0, this field behaves as RAZ/WI, regardless of its actual value.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

\[
\text{EL3 is implemented} \quad \text{EL3 is using AArch32} \quad \text{!IsCurrentSecurityState(SS_Secure)} \\
\text{NSACR.cp10} = 0 \\
\]

, access to this field is RAZ/WI.

cp10, bits [21:20]
 Defines the access rights for the Advanced SIMD and floating-point functionality. Possible values of the field are:
0b00 PL0 and PL1 accesses to Advanced SIMD and floating-point registers or instructions are UNDEFINED.
0b01 PL0 accesses to Advanced SIMD and floating-point registers or instructions are UNDEFINED.
0b10 Reserved. The effect of programming this field to this value is CONSTRAINED UNPREDICTABLE. See Handling of System register control fields for Advanced SIMD and floating-point operation on page K1-11568.
This control permits full access to the Advanced SIMD and floating-point functionality from PL0 and PL1.

The Advanced SIMD and floating-point features controlled by these fields are:

- Execution of any floating-point or Advanced SIMD instruction.
- Any access to the Advanced SIMD and floating-point registers D0-D31 and their views as S0-S31 and Q0-Q15.
- Any access to the FPSCR, FPSID, MVFR0, MVFR1, MVFR2, or FPEXC System registers.

**Note**

The CPACR has no effect on Advanced SIMD and floating-point accesses from PL2. These can be disabled by the HCPTR_TCP10 field.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

In Non-secure state, if EL3 is implemented and is using AArch32, when the value of NSACR.cp10 is 0, this field behaves as RAZ/WI, regardless of its actual value.

Execution of Advanced SIMD and floating-point instructions in AArch32 state can be disabled or trapped by the following controls:

- CPACR.cp10, or, if executing at EL0, CPACR_EL1.FPEN.
- FPEXC.EN.
- If executing in Non-secure state:
  - HCPTR_TCP10, or if EL2 is using AArch64, CPTR_EL2.TFP.
  - NSACR.cp10, or if EL3 is using AArch64, CPTR_EL3.TFP.
- For Advanced SIMD instructions only:
  - CPACR.ASEDIS.
  - If executing in Non-secure state, HCPTR.TASE and NSACR.NSTRCDIS.

See the descriptions of the controls for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

Bits [19:0]

- Reserved, RES0.

**Accessing CPACR**

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}(<q>) <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & CPTR_EL3.TCPAC == '1' then
    UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TCPAC == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPR.TCPAC == '1' then
AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
R[t] = CPACR;
elsif PSTATE_EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" & !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
R[t] = CPACR;
elsif PSTATE_EL == EL3 then
R[t] = CPACR;

MCR{<c}<{q} {coproc}, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elsif PSTATE_EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" & !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TCPAC == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPR.TCPAC == '1' then
AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
CPACR = R[t];
elsif PSTATE_EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" & !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
    CPACR = R[t];
elsif PSTATE.EL == EL3 then
    CPACR = R[t];
G8.2.33  CPSR, Current Program Status Register

The CPSR characteristics are:

**Purpose**

Holds PE status and control information.

**Configurations**

This register is present only when AArch32 is supported. Otherwise, direct accesses to CPSR are UNDEFINED.

**Attributes**

CPSR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N, bit [31]</td>
<td>Negative condition flag. Set to bit[31] of the result of the last flag-setting instruction. If the result is regarded as a two's complement signed integer, then N is set to 1 if the result was negative, and N is set to 0 if the result was positive or zero.</td>
</tr>
<tr>
<td>Z, bit [30]</td>
<td>Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.</td>
</tr>
<tr>
<td>C, bit [29]</td>
<td>Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.</td>
</tr>
<tr>
<td>V, bit [28]</td>
<td>Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.</td>
</tr>
<tr>
<td>Q, bit [27]</td>
<td>Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.</td>
</tr>
<tr>
<td>Bits [26:24]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>SSBS, bit [23]</td>
<td>When FEAT_SSBS is implemented: Speculative Store Bypass Safe. Prohibits speculative loads or stores that might practically allow a cache timing side channel. A cache timing side channel might be exploited where a load or store uses an address that is derived from a register that is being loaded from memory using a load instruction speculatively read from a memory location. If PSTATE.SSBS is enabled, the address derived from the load instruction might be from earlier in the coherence order than the latest store to that memory location with the same virtual address. 0b0 Hardware is not permitted to load or store speculatively in the manner described.</td>
</tr>
</tbody>
</table>
0b1 Hardware is permitted to load or store speculatively in the manner described.
The value of this bit is usually set to the value described by the SCTLR.DSSBS bit on exceptions
to any mode except Hyp mode, and the value described by HSCTRL.DSSBS on exceptions to Hyp
mode.
The reset behavior of this field is:
• On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:
Reserved, RES0.

PAN, bit [22]
When FEAT_PAN is implemented:
Privileged Access Never.
0b0 The translation system is the same as Armv8.0.
0b1 Disables privileged read and write accesses to addresses accessible at EL0.
The value of this bit is usually preserved on taking an exception, except in the following situations:
• When the target of the exception is EL1, and the value of the SCTLR.SPAN bit for the current
  Security state is 0, this bit is set to 1.
• When the target of the exception is EL3, from Secure state, and the value of the Secure
  SCTLR.SPAN is 0, this bit is set to 1.
• When the target of the exception is EL3, from Non-secure state, this bit is set to 0 regardless
  of the value of the Secure SCTLR.SPAN bit.

Otherwise:
Reserved, RES0.

DIT, bit [21]
When FEAT_DIT is implemented:
Data Independent Timing.
0b0 The architecture makes no statement about the timing properties of any instructions.
0b1 The architecture requires that:
• The timing of every load and store instruction is insensitive to the value of the
  data being loaded or stored.
• For certain data processing instructions, the instruction takes a time that is independent of:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
• For certain data processing instructions, the response of the instruction to
  asynchronous exceptions does not vary based on:
  — The values of the data supplied in any of its registers.
  — The values of the NZCV flags.
The data processing instructions affected by this bit are:
• All cryptographic instructions. These instructions are:
  — AESD, AESE, AESIMC, AESMC, SHA1C, SHA1H, SHA1M, SHA1P, SHA1SU0, SHA1SU1, SHA256H,
    SHA256H2, SHA256SU0, and SHA256SU1.
• A subset of the instructions that use the general-purpose register file. For these instructions,
  the effects of CPSR.DIT apply only if they do not use R15 as either their source or destination
  and pass their condition execution check. These instructions are:
  — BFI, BFC, CLZ, CNV, CMP, MLA, MLAS, MLAS, MOV, MUL, NOP, PKHBT, PKHTB, RBIT, REV, REV16,
    REVSH, RRX, SADD16, SADD8, SASX, SBFX, SHADD16, SHADD8, SHASX, SHAESX, SHSUB16, SHSUB8,
    SMLAL**, SMLAW*, SMLSD*, SMLAS*, SMLAS*, SMUL*, SMUL*, SMUAD*, SMUAD*, SSAX, SSUB16, SSUB8,
Bit [20]

Reserved, RES0.

Otherwise:

Reserved, RES0.

A subset of the instructions that use the general-purpose register file. For these instructions, the effects of CPSR.DIT apply only if they do not use R15 as either their source or destination. The effects of CPSR.DIT do not depend on these instructions passing their condition execution check. These instructions are:

- ADC (immediate), ADC (register), ADCS (immediate), ADCS (register), ADD (immediate), ADD (register), ADDS (immediate), ADDS (register), AND (immediate), AND (register), ANDS (immediate), ANDS (register), ASR (immediate), ASR (register), ASRS (immediate), ASRS (register), BIC (immediate), BIC (register), BICS (immediate), BICS (register), EOR (immediate), EOR (register), EORS (immediate), EORS (register), LSL (immediate), LSL (register), LSR (immediate), LSR (register), LSRS (immediate), LSRS (register), MOV (immediate), MOV (register), MOVs (immediate), MOVs (register), MVN (immediate), MVN (register), WNS (immediate), WNS (register), ORR (immediate), ORR (register), ORRS (immediate), ORRS (register), ROR (immediate), ROR (register), RORS (immediate), RORS (register), RSB (immediate), RSB (register), RSBS (immediate), RSBS (register), RSC (immediate), RSC (register), RSCS (immediate), RSCS (register), SBC (immediate), SBC (register), SBCS (immediate), SBCS (register), SUB (immediate), SUB (register), SUBS (immediate), SUBS (register).

If FEAT_CRC32 is implemented, CRC32B, CRC32H, CRC32W, CRC32CB, CRC32CH, and CRC32CW.

A subset of the instructions that use the SIMD&FP register file. For these instructions, the effects of CPSR.DIT apply only if they pass their condition execution check. These instructions are:

- VABA+, VABD+ (integer), VADD (integer), VADDH, VADDL, VADDW, VADD, VAND, VBIC, VBIF, VBIT, VBSL, VCLS, VCLS, VCLZ, VCN, VDUP, VEDO, VEXT, VADDH, VSUB, VMAX (integer), VMIN (integer), VMLA (integer), VMLAL, VMLS (integer), VMLSL, VMOD, VMODL, VMUL (integer and polynomial), VMULL (integer and polynomial), VMVN, VORH, VORR, VPAADDL, VPAADD (integer), VPADDL, VPADDL, VPADDL, VPMAX (integer), VPMIN (integer), VRAADH, VREV+, VRADH, VRHLSL, VRHLSL, VRSRA, VRSUBH, VSLR, VSLL, VSHR, VSLI, VSR, VSB, VSUB (integer), VSUBH, VSBUL, VSUBW, VSWP, VTLB, VTXB, VTRN, VST, VUZP, and VZIP.

Another subset of the instructions that use the SIMD&FP register file. For these instructions, the effects of CPSR.DIT apply only if they pass their condition execution check and apply only when the instructions are operating on integer vector elements. These instructions are:

- VABS, VCEG, VCGT, VCLE, VCLT, VMLA (by scalar), VMUL (by scalar), and VNEG.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.
G8.2 General system control registers

GE, bits [19:16]
Greater than or Equal flags, for parallel addition and subtraction.

Bits [15:10]
Reserved, RES0.

E, bit [9]
Endianness state bit. Controls the load and store endianness for data accesses:

0b0 Little-endian operation
0b1 Big-endian operation.

Instruction fetches ignore this bit.
If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.
If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.
Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

The reset behavior of this field is:
• When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

A, bit [8]
SError interrupt mask bit.
0b0 Exception not masked.
0b1 Exception masked.

I, bit [7]
IRQ mask bit.
0b0 Exception not masked.
0b1 Exception masked.

F, bit [6]
FIQ mask bit.
0b0 Exception not masked.
0b1 Exception masked.

Bit [5]
Reserved, RES0.

Bit [4]
Reserved, RES1.

M, bits [3:0]
Current PE mode.
0b0000 User.
0b0001 FIQ.
0b0010 IRQ.
0b0011 Supervisor.
0b0110 Monitor.
0b0111 Abort.
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1010</td>
<td>Hyp.</td>
</tr>
<tr>
<td>0b1011</td>
<td>Undefined.</td>
</tr>
<tr>
<td>0b1111</td>
<td>System.</td>
</tr>
</tbody>
</table>
G8.2.34 CPPRCTX, Cache Prefetch Prediction Restriction by Context

The CPPRCTX characteristics are:

**Purpose**

Cache Prefetch Prediction Restriction by Context applies to all Cache Allocation Resources that predict cache allocations based on information gathered within the target execution context or contexts.

The actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control cache prefetch predictions occurring after the instruction is complete and synchronized.

This instruction applies to all:
- Instruction caches.
- Data caches.
- TLB prefetching hardware used by the executing PE that applies to the supplied context or contexts.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

**Note**

This instruction does not require the invalidation of Cache Allocation Resources so long as the behavior described for completion of this instruction is met by the implementation.

On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

**Configurations**

This instruction is present only when AArch32 is supported and FEAT_SPECRES is implemented. Otherwise, direct accesses to CPPRCTX are UNDEFINED.

**Attributes**

CPPRCTX is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>16</th>
<th>15</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>NS</td>
<td>EL</td>
<td>VMID</td>
<td>RES0</td>
<td>ASID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:28]**

Reserved, RES0.

**GVMID, bit [27]**

Execution of this instruction applies to all VMIDs or a specified VMID.

- 0b0 Applies to specified VMID for an EL0 or EL1 target execution context.
- 0b1 Applies to all VMIDs for an EL0 or EL1 target execution context.

For target execution contexts other than EL0 or EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, then this field has an Effective value of 0.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.
NS, bit [26]

Security State.
- \(0b0\) Secure state.
- \(0b1\) Non-secure state.

If the instruction is executed in Non-secure state, this field is treated as 1.

EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.
- \(0b00\) EL0.
- \(0b01\) EL1.
- \(0b10\) EL2.
- \(0b11\) EL3.

If the instruction is executed at an Exception level lower than the specified level, this instruction is treated as a NOP.

VMID, bits [23:16]

Only applies when bit[27] is 0 and the target execution context is either:
- EL1.
- EL0 when \(HCR\_EL2.E2H==0\) or \(HCR\_EL2.TGE==0\) or EL2 is using AArch32 state.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.
When the instruction is executed at EL0 and \((HCR\_EL2.E2H==0\) or \(HCR\_EL2.TGE==0\) or ELUsingAArch32(EL2)), this field is treated as the current VMID.
When the instruction is executed at EL0 and \((HCR\_EL2.E2H==1\) and \(HCR\_EL2.TGE==1\) and !ELUsingAArch32(EL2)), this field is ignored.
If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

Bits [15:9]

Reserved, RES0.

GASID, bit [8]

Execution of this instruction applies to all ASIDs or a specified ASID.
- \(0b0\) Applies to specified ASID for an EL0 target execution context.
- \(0b1\) Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.
If the instruction is executed at EL0, this field has an Effective value of 0.

ASID, bits [7:0]

Only applies for an EL0 target execution context and when bit[8] is 0.
Otherwise, this field is RES0.
When the instruction is executed at EL0, this field is treated as the current ASID.

Executing CPPRCTX instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.EnRCTX == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    endif
else
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
    endif
else
    if EL2Enabled() && !ELUsingAArch32(EL2) && !HaveEL(EL3) || SCR_EL3.FGTEn == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x03);
    else
        AArch32.RestrictPrediction(R[t], RestrictType_CachePrefetch);
    endif
else
    if PSTATE.EL == EL1 then
        if !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
            AArch32.TakeHypTrapException(0x03);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.NV == '1' then
            AArch64.SystemAccessTrap(EL2, 0x03);
        else
            AArch32.RestrictPrediction(R[t], RestrictType_CachePrefetch);
        endif
    elsif PSTATE.EL == EL2 then
        AArch32.RestrictPrediction(R[t], RestrictType_CachePrefetch);
    elsif PSTATE.EL == EL3 then
        AArch32.RestrictPrediction(R[t], RestrictType_CachePrefetch);
    endif

### Table: MCR{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>)}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b111</td>
<td>0b0011</td>
<td>0b111</td>
</tr>
</tbody>
</table>

---

AArch32 System Register Descriptions

G8.2 General system control registers

---

ARM DDI 0487I.a

Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.

Non-Confidential

G8-9403

ID081822
G8.2.35 CSSELR, Cache Size Selection Register

The CSSELR characteristics are:

Purpose

Selects the current Cache Size ID Register, CCSIDR, by specifying the required cache level and the cache type, which is either instruction cache or data cache.

If FEAT_CCIDX is implemented, CSSELR also selects the current CCSIDR2.

Configurations

AArch32 System register CSSELR bits [31:0] are architecturally mapped to AArch64 System register CSSELR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to CSSELR are UNDEFINED.

Attributes

CSSELR is a 32-bit register.

Field descriptions

Bits [31:4]

Reserved, RES0.

Level, bits [3:1]

Cache level of required cache. Permitted values are:

- 0b000: Level 1 cache.
- 0b001: Level 2 cache.
- 0b010: Level 3 cache.
- 0b011: Level 4 cache.
- 0b100: Level 5 cache.
- 0b101: Level 6 cache.
- 0b110: Level 7 cache.

All other values are reserved.

If CSSELR.{Level, InD} is programmed to a cache level that is not implemented, then the value for this field on a read of CSSELR is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

InD, bit [0]

Instruction not Data bit. Permitted values are:

- 0b0: Data or unified cache.
- 0b1: Instruction cache.

If CSSELR.{Level, InD} is programmed to a cache level that is not implemented, then the value for this field on a read of CSSELR is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Accessing CSSELR

Accesses to this register use the following encodings in the System register encoding space:

**MRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>**

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b010</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TID4 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TID4 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        R[t] = CSSELR_NS;
    else
        R[t] = CSSELR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        R[t] = CSSELR_NS;
    else
        R[t] = CSSELR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        R[t] = CSSELR_S;
    else
        R[t] = CSSELR_NS;

**MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>**

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b010</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TID2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TID4 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TID4 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TID4 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TID4 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
AArch32.TakeHypTrapException(0x03);
eelsif HaveEL(EL3) & ELUsingAArch32(EL3) then
  CSSELR_NS = R[t];
else
  CSSELR = R[t];
eelsif PSTATE_EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    CSSELR_NS = R[t];
  else
    CSSELR = R[t];
eelsif PSTATE_EL == EL3 then
  if SCR.NS == '0' then
    CSSELR_S = R[t];
  else
    CSSELR_NS = R[t];
G8.2.36 CTR, Cache Type Register

The CTR characteristics are:

**Purpose**

Provides information about the architecture of the caches.

**Configurations**

AArch32 System register CTR bits [31:0] are architecturally mapped to AArch64 System register CTR_EL0[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to CTR are UNDEFINED.

**Attributes**

CTR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES1.</td>
</tr>
<tr>
<td>30</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>29</td>
<td>DIC, bit Instruction cache invalidation requirements for data to instruction coherence. 0b0 Instruction cache invalidation to the Point of Unification is required for data to instruction coherence. 0b1 Instruction cache invalidation to the Point of Unification is not required for data to instruction coherence.</td>
</tr>
<tr>
<td>28</td>
<td>IDC, bit Data cache clean requirements for instruction to data coherence. The meaning of this bit is: 0b0 Data cache clean to the Point of Unification is required for instruction to data coherence, unless CLIDR.LoC == 0b000 or (CLIDR.LoUIS == 0b000 &amp;&amp; CLIDR.LoUU == 0b000). 0b1 Data cache clean to the Point of Unification is not required for instruction to data coherence.</td>
</tr>
<tr>
<td>27:24</td>
<td>CWG, bits Cache writeback granule. Log2 of the number of words of the maximum size of memory that can be overwritten as a result of the eviction of a cache entry that has had a memory location in it modified. A value of 0b0000 indicates that this register does not provide Cache writeback granule information and either: The architectural maximum of 512 words (2KB) must be assumed. The Cache writeback granule can be determined from maximum cache line size encoded in the Cache Size ID Registers. Values greater than 0b1001 are reserved.</td>
</tr>
</tbody>
</table>
Arm recommends that an implementation that does not support cache write-back implements this field as 0b0001. This applies, for example, to an implementation that supports only write-through caches.

**ERG, bits [23:20]**

Exclusives reservation granule. $\log_2$ of the number of words of the maximum size of the reservation granule that has been implemented for the Load-Exclusive and Store-Exclusive instructions.

The use of the value 0b0000 is deprecated.

The value 0b0001 and values greater than 0b1001 are reserved.

**DminLine, bits [19:16]**

$\log_2$ of the number of words in the smallest cache line of all the data caches and unified caches that are controlled by the PE.

**L1Ip, bits [15:14]**

Level 1 instruction cache policy. Indicates the indexing and tagging policy for the L1 instruction cache. Possible values of this field are:

- 0b00: When FEAT_VPIPT is implemented: VMID aware Physical Index, Physical tag (VPIPT).
- 0b01: ASID-tagged Virtual Index, Virtual Tag (AIVIVT).
- 0b10: Virtual Index, Physical Tag (VIPT).
- 0b11: Physical Index, Physical Tag (PIPT).

From Armv8, the value 0b01 is not permitted.

The value 0b00 is permitted only in an implementation that includes FEAT_VPIPT.

**Bits [13:4]**

Reserved, RES0.

**IminLine, bits [3:0]**

$\log_2$ of the number of words in the smallest cache line of all the instruction caches that are controlled by the PE.

### Accessing CTR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elif PSTATE_EL == EL1 then
if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && ELUsingArch32(EL2) && HCR_EL2.TID2 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingArch32(EL2) && HCR.TID2 == '1' then
AArch32.TakeHypTrapException(0x03);
else
R[t] = CTR;
```
elsif PSTATE_EL == EL2 then
  R[t] = CTR;
elsif PSTATE_EL == EL3 then
  R[t] = CTR;
G8.2.37 DACR, Domain Access Control Register

The DACR characteristics are:

**Purpose**

Defines the access permission for each of the sixteen memory domains.

**Configurations**

AArch32 System register DACR bits [31:0] are architecturally mapped to AArch64 System register DACR32_EL2[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DACR are UNDEFINED.

This register has no function when TTBCR.EAE is set to 1, to select the Long-descriptor translation table format.

**Attributes**

DACR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>D15</th>
<th>D14</th>
<th>D13</th>
<th>D12</th>
<th>D11</th>
<th>D10</th>
<th>D9</th>
<th>D8</th>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
</tbody>
</table>

**D<n>, bits [2n+1:2n], for n = 15 to 0**

Domain n access permission, where n = 0 to 15. Permitted values are:

- **0b00**: No access. Any access to the domain generates a Domain fault.
- **0b01**: Client. Accesses are checked against the permission bits in the translation tables.
- **0b11**: Manager. Accesses are not checked against the permission bits in the translation tables.

The value **0b10** is reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing DACR**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>}, \{#\}<opc1>, <Rt>, <CRn>, <CRm>, \{#\}<opc2>\
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0000</td>
<td>0b0011</td>
<td>0b00000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T3 == '1' then
        AArch32.TakeHypTrapException(0x83);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x83);
elsif HaveEL(EL3) \&\& ELUsingAArch32(EL3) then
    \text{R}[t] = DACR_NS;
else
    \text{R}[t] = DACR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) \&\& ELUsingAArch32(EL3) then
        \text{R}[t] = DACR_NS;
    else
        \text{R}[t] = DACR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        DACR_NS = \text{R}[t];
    else
        DACR = \text{R}[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) \&\& ELUsingAArch32(EL3) then
        DACR_NS = \text{R}[t];
else
    DACR = \text{R}[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' \&\& CP15SDISABLE == HIGH then
        UNDEFINED;
elsif SCR.NS == '0' \&\& CP15SDISABLE2 == HIGH then
    UNDEFINED;
else
    if SCR.NS == '0' then
        DACR_S = \text{R}[t];
    else
        DACR_NS = \text{R}[t];

\textbf{MCR\{c\}\{q\} <coproc>, \{#\}<opc1>, <Rt>, <CRn>, <CRm>, \{#\}<opc2>}
\begin{center}
\begin{tabular}{cccccc}
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
0b1111 & 0b0000 & 0b0011 & 0b0000 & 0b0000
\end{tabular}
\end{center}
G8.2.38  DCCIMVAC, Data Cache line Clean and Invalidate by VA to PoC

The DCCIMVAC characteristics are:

**Purpose**
Clean and Invalidate data or unified cache line by virtual address to PoC.

**Configurations**
AArch32 System register DCCIMVAC performs the same function as AArch64 System register DC CIVAC.

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DCCIMVAC are UNDEFINED.

**Attributes**
DCCIMVAC is a 32-bit System instruction.

**Field descriptions**

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Virtual address to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td></td>
</tr>
</tbody>
</table>
```

**Bits [31:0]**
Virtual address to use. No alignment restrictions apply to this VA.

**Executing DCCIMVAC instruction**
Execution of this instruction might require an address translation from VA to PA, and that translation might fault.

For more information about faults, see *Permission fault* on page G5-9239.

For more information about data cache maintenance instructions, see *AArch32 data cache maintenance instructions (DC*)* on page G4-9125.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>}, <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1110</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TPC == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    AArch32.DC(R[t], CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
  AArch32.DC(R[t], CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE_EL == EL3 then
    AArch32.DC(R[t], CacheOp_CleanInvalidate, CacheOpScope_PoC);
**G8.2.39  DCCISW, Data Cache line Clean and Invalidate by Set/Way**

The DCCISW characteristics are:

**Purpose**
Clean and Invalidate data or unified cache line by set/way.

**Configurations**
AArch32 System register DCCISW performs the same function as AArch64 System register DC CISW.
This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DCCISW are UNDEFINED.

**Attributes**
DCCISW is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-4</td>
<td>SetWay</td>
<td>Contains two fields:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Way, bits[31:32-A], the number of the way to operate on.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Set, bits[B-1:L], the number of the set to operate on.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits[L-1:4] are RES0.</td>
</tr>
<tr>
<td>3-1</td>
<td>Level</td>
<td>Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.</td>
</tr>
<tr>
<td>0</td>
<td>Bit</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**SetWay, bits [31:4]**
Contains two fields:
- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.
Bits[L-1:4] are RES0.

\[ A = \log_2(\text{ASSOCIATIVITY}), L = \log_2(\text{LINELEN}), B = (L + S), S = \log_2(\text{NSETS}) \]

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

**Level, bits [3:1]**
Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

**Bit [0]**
Reserved, RES0.

**Executing DCCISW instruction**
If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Accesses to this instruction use the following encodings in the System instruction encoding space:
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TSW == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TSW == '1' then
    AArch32.TakeHypTrapException(0x03);
else
  AArch32.DC(R[t], CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elsif PSTATE_EL == EL2 then
  AArch32.DC(R[t], CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elsif PSTATE_EL == EL3 then
  AArch32.DC(R[t], CacheOp_CleanInvalidate, CacheOpScope_SetWay);
**G8.2.40 DCCMVAC, Data Cache line Clean by VA to PoC**

The DCCMVAC characteristics are:

**Purpose**
Clean data or unified cache line by virtual address to PoC.

**Configurations**
AArch32 System register DCCMVAC performs the same function as AArch64 System register DC
CVAC.

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses
to DCCMVAC are UNDEFINED.

**Attributes**
DCCMVAC is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th>Virtual address to use</th>
</tr>
</thead>
</table>

**Bits [31:0]**
Virtual address to use. No alignment restrictions apply to this VA.

**Executing DCCMVAC instruction**

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *AArch32 data cache maintenance instructions (DC*) on page G4-9125.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{MCR\{<c>\}{<q>} <coproc>, \{#<opc1>, <Rt>, <CRn>, <CRm>{, \{#<opc2>\}}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
AArch32.TakeHypTrapException(0x03);
elif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TPC == '1' then
AArch32.TakeHypTrapException(0x03);
else
AArch32.DC(R[t], CacheOp_Clean, CacheOpScope_PoC);
elif PSTATE.EL == EL2 then
AArch32.DC(R[t], CacheOp_Clean, CacheOpScope_PoC);
elif PSTATE.EL == EL3 then
AArch32.DC(R[t], CacheOp_Clean, CacheOpScope_PoC);
G8.2.41 DCCMVAU, Data Cache line Clean by VA to PoU

The DCCMVAU characteristics are:

**Purpose**
Clean data or unified cache line by virtual address to PoU.

**Configurations**
AArch32 System register DCCMVAU performs the same function as AArch64 System register DC
CVAU.

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses
to DCCMVAU are UNDEFINED.

**Attributes**
DCCMVAU is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th>Virtual address to use</th>
</tr>
</thead>
</table>

**Executing DCCMVAU instruction**

Execution of this instruction might require an address translation from VA to PA, and that translation might fault.
For more information, see *AArch32 data cache maintenance instructions (DC*) on page G4-9125.

Accesses to this instruction use the following encodings in the System instruction encoding space:

$\text{MCR}<c>\{<q>\} <\text{coproc}>, \{#\}<\text{opc}1>, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>, \{#\}<\text{opc}2>$

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPU == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TOCU == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TPU == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TOCU == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    AArch32.DC(R[t], CacheOp_Clean, CacheOpScope_PoU);
elsif PSTATE.EL == EL2 then
  AArch32.DC(R[t], CacheOp_Clean, CacheOpScope_PoU);
elsif PSTATE_EL == EL3 then
  AArch32.DC(R[t], CacheOp_Clean, CacheOpScope_PoU);
G8.2.42 DCCSW, Data Cache line Clean by Set/Way

The DCCSW characteristics are:

Purpose

Clean data or unified cache line by set/way.

Configurations

AArch32 System register DCCSW performs the same function as AArch64 System register DC CSW.

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DCCSW are UNDEFINED.

Attributes

DCCSW is a 32-bit System instruction.

Field descriptions

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DCCSW instruction

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Accesses to this instruction use the following encodings in the System instruction encoding space:
MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b000</td>
<td>0b011</td>
<td>0b101</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TSW == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TSW == '1' then
    AArch32.TakeHypTrapException(0x03);
else
  AArch32.DC(R[t], CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL2 then
  AArch32.DC(R[t], CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
  AArch32.DC(R[t], CacheOp_Clean, CacheOpScope_SetWay);
G8.2.43 DCIMVAC, Data Cache line Invalidate by VA to PoC

The DCIMVAC characteristics are:

**Purpose**
Invalid data or unified cache line by virtual address to PoC.

**Configurations**
AArch32 System register DCIMVAC performs the same function as AArch64 System register DCIVAC.

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DCIMVAC are UNDEFINED.

**Attributes**
DCIMVAC is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th>Virtual address to use</th>
</tr>
</thead>
</table>

**Virtual address to use**

No alignment restrictions apply to this VA.

**Executing DCIMVAC instruction**

It is IMPLEMENTATION DEFINED whether, when this instruction is executed, it can generate a watchpoint. If this instruction can generate a watchpoint this is prioritized in the same way as other watchpoints.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see *AArch32 data cache maintenance instructions (DC*) on page G4-9125.

Accesses to this instruction use the following encodings in the System instruction encoding space:

<table>
<thead>
<tr>
<th>MCR{&lt;c&gt;}{&lt;q&gt;} &lt;coproc&gt;, (#)&lt;opc1&gt;, &lt;Rt&gt;, &lt;CRn&gt;, &lt;CRm&gt;{, (#)&lt;opc2&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>coproc   opc1   CRn   CRm   opc2</td>
</tr>
<tr>
<td>0b1111 0b000 0b0111 0b0110 0b001</td>
</tr>
</tbody>
</table>

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TPCP == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TPC == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        AArch32.DC(R[t], CacheOp_Invalidate, CacheOpScope_PoC);
    elsif PSTATE.EL == EL2 then
        AArch32.DC(R[t], CacheOp_Invalidate, CacheOpScope_PoC);
```
elsif PSTATE_EL == EL3 then
  AArch32.DC(R[t], CacheOp_Invalidate, CacheOpScope_PoC);
G8.2.44 DCISW, Data Cache line Invalidate by Set/Way

The DCISW characteristics are:

**Purpose**

Invalidate data or unified cache line by set/way.

**Configurations**

AArch32 System register DCISW performs the same function as AArch64 System register DCISW.

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DCISW are UNDEFINED.

**Attributes**

DCISW is a 32-bit System instruction.

**Field descriptions**

```
+--------+--------+--------+---+--------+
| 31     | 30     | ...   | 1  | 0      |
| SetWay | Level  | RES0   |    | RES0   |
```

**SetWay, bits [31:4]**

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

\[
A = \log_2(\text{ASSOCIATIVITY}), \quad L = \log_2(\text{LINELEN}), \quad B = (L + S), \quad S = \log_2(\text{NSETS}).
\]

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

**Level, bits [3:1]**

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

**Bit [0]**

Reserved, RES0.

**Executing DCISW instruction**

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Accesses to this instruction use the following encodings in the System instruction encoding space:
MCR{c|q} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0110</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TSW == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TSW == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    AArch32.DC(R[t], CacheOp_Invalidate, CacheOpScope_SetWay);
  end-if
elsif PSTATE.EL == EL2 then
  AArch32.DC(R[t], CacheOp_Invalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL3 then
  AArch32.DC(R[t], CacheOp_Invalidate, CacheOpScope_SetWay);
### G8.2.45 DFAR, Data Fault Address Register

The DFAR characteristics are:

**Purpose**

Holds the virtual address of the faulting address that caused a synchronous Data Abort exception.

**Configurations**

AArch32 System register DFAR bits [31:0] are architecturally mapped to AArch64 System register FAR_EL1[31:0].

AArch32 System register DFAR bits [31:0] (DFAR_S) are architecturally mapped to AArch32 System register HDFAR[31:0] when EL2 is implemented, EL3 is implemented and the implementation only supports execution in AArch32 state.

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DFAR are UNDEFINED.

**Attributes**

DFAR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th>VA of faulting address of synchronous Data Abort exception</th>
</tr>
</thead>
</table>

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing DFAR**

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
    UNDEFINED;
elses if PSTATE_EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
e else if EL2Enabled() && ELUsingAAArch32(EL2) && HSTR.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
e else if EL2Enabled() && !ELUsingAAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
e else if EL2Enabled() && ELUsingAAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
e else if HaveEL(EL3) && ELUsingAAArch32(EL3) then
        R[t] = DFAR_NS;
e else
        R[t] = DFAR;
elses if PSTATE_EL == EL2 then
```
if HaveEL(EL3) && ELUsingAArch32(EL3) then
  R[t] = DFAR_NS;
else
  R[t] = DFAR;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    R[t] = DFAR_S;
  else
    R[t] = DFAR_NS;
else if PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    DFAR_NS = R[t];
  else
    DFAR = R[t];
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    DFAR_S = R[t];
  else
    DFAR_NS = R[t];

MCR{<c>}{<q>}{<coproc>}, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else if EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T6 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else if EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
    AArch32.TakeHypTrapException(0x03);
  else if HaveEL(EL3) && ELUsingAArch32(EL3) then
    DFAR_NS = R[t];
  else
    DFAR = R[t];
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    DFAR_NS = R[t];
  else
    DFAR = R[t];
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    DFAR_S = R[t];
  else
    DFAR_NS = R[t];
G8.2.46   DFSR, Data Fault Status Register

The DFSR characteristics are:

**Purpose**

Holds status information about the last data fault.

**Configurations**

AArch32 System register DFSR bits [31:0] are architecturally mapped to AArch64 System register ESR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DFSR are UNDEFINED.

The current translation table format determines which format of the register is used.

**Attributes**

DFSR is a 32-bit register.

**Field descriptions**

*When TTBCR.EAE == 0:*

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31 | RES0 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 4 | 3 | 2 | 1 | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Bits [31:17]**

Reserved, RES0.

**FnV, bit [16]**

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

0b0  DFAR is valid.

0b1  DFAR is not valid, and holds an UNKNOWN value.

This field is valid only for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Data Abort exceptions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**AET, bits [15:14]**

*When FEAT_RAS is implemented:*

Asynchronous Error Type. When DFSC is 0b010001, describes the PE error state after taking the SError interrupt exception. Possible values are:

0b00  Uncontainable (UC).

0b01  Unrecoverable state (UEU).

0b10  Restartable state (UEO).

0b11  Recoverable state (UER).

This field is valid only if the DFSC code is 0b010001. It is RES0 for all other aborts.

In the event of multiple errors taken as a single SError interrupt exception, the overall PE error state is reported.
Note

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CM, bit [13]

Cache maintenance fault. For synchronous faults, this bit indicates whether a cache maintenance instruction generated the fault.

- 0b0: Abort not caused by execution of a cache maintenance instruction.
- 0b1: Abort caused by execution of a cache maintenance instruction, or on an address translation.

On a synchronous Data Abort exception on a translation table walk, this bit is UNKNOWN.

On an asynchronous fault, this bit is UNKNOWN.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ExT, bit [12]

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.


Write not Read bit. Indicates whether the abort was caused by a write or a read instruction.

- 0b0: Abort caused by a read instruction.
- 0b1: Abort caused by a write instruction.

For faults on the cache maintenance and address translation System instructions in the (coproc==0b1111) encoding space this bit always returns a value of 1.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FS, bits [10, 3:0]

Fault status bits. Possible values of FS[4:0] are:

- 0b00001: Alignment fault.
- 0b00010: Debug exception.
- 0b00011: Access flag fault, level 1.
- 0b00100: Fault on instruction cache maintenance.
- 0b00101: Translation fault, level 1.
- 0b00110: Access flag fault, level 2.
- 0b00111: Translation fault, level 2.
- 0b01000: Synchronous External abort, not on translation table walk.
0b01001  Domain fault, level 1.
0b01011  Domain fault, level 2.
0b01100  Synchronous External abort, on translation table walk, level 1.
0b01101  Permission fault, level 1.
0b01110  Synchronous External abort, on translation table walk, level 2.
0b01111  Permission fault, level 2.
0b10000  TLB conflict abort.
0b10100  IMPLEMENTATION DEFINED fault (Lockdown fault).
0b10101  IMPLEMENTATION DEFINED fault (Unsupported Exclusive access fault).
0b10110  SError interrupt.
0b11000  When FEAT_RAS is not implemented:
          SError interrupt, from a parity or ECC error on memory access.
0b11001  When FEAT_RAS is not implemented:
          Synchronous parity or ECC error on memory access, not on translation table walk.
0b11100  When FEAT_RAS is not implemented:
          Synchronous parity or ECC error on translation table walk, level 1.
0b11110  When FEAT_RAS is not implemented:
          Synchronous parity or ECC error on translation table walk, level 2.

All other values are reserved.

For more information about the lookup level associated with a fault, see The level associated with MMU faults on a Short-descriptor translation table lookup on page G5-9257.

The FS field is split as follows:
• FS[4] is DFSR[10].
• FS[3:0] is DFSR[3:0].
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

LPAE, bit [9]

On taking a Data Abort exception, this bit is set as follows:
0b0  Using the Short-descriptor translation table formats.
0b1  Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

Domain, bits [7:4]

The domain of the fault address.

Arm deprecates any use of this field, see The Domain field in the DFSR on page G5-9257.

This field is UNKNOWN for certain faults where the DFSR is updated and reported using the Short-descriptor FSR encodings, see Table G5-30 on page G5-9261.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
**When TTBCR.EAE == 1:**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:17</td>
<td>Reserved, RES0</td>
</tr>
</tbody>
</table>

**FnV, bit [16]**

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

- 0b0   DFAR is valid.
- 0b1   DFAR is not valid, and holds an UNKNOWN value.

This field is valid only for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Data Abort exceptions.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**AET, bits [15:14]**

*When FEAT_RAS is implemented:*

Asynchronous Error Type. When DFSC is 0b010001, describes the PE error state after taking the SError interrupt exception. Possible values are:

- 0b0   Uncontainable (UC).
- 0b1   Unrecoverable state (UEU).
- 0b10  Restartable state (UEO).
- 0b11  Recoverable state (UER).

This field is valid only if the DFSC code is 0b010001. It is RES0 for all other aborts.

In the event of multiple errors taken as a single SError interrupt exception, the overall PE error state is reported.

---- **Note** ----

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

----

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**CM, bit [13]**

Cache maintenance fault. For synchronous faults, this bit indicates whether a cache maintenance instruction generated the fault.

- 0b0   Abort not caused by execution of a cache maintenance instruction.
- 0b1   Abort caused by execution of a cache maintenance instruction.

On a synchronous Data Abort exception on a translation table walk, this bit is UNKNOWN.

On an asynchronous fault, this bit is UNKNOWN.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ExT, bit [12]**

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**WnR, bit [11]**

Write not Read bit. Indicates whether the abort was caused by a write or a read instruction.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Abort caused by a read instruction.</td>
</tr>
<tr>
<td>0b1</td>
<td>Abort caused by a write instruction.</td>
</tr>
</tbody>
</table>

For faults on the cache maintenance and address translation System instructions in the (coproc==@b1111) encoding space this bit always returns a value of 1.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [10]**

Reserved, RES0.

**LPAE, bit [9]**

On taking a Data Abort exception, this bit is set as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Using the Short-descriptor translation table formats.</td>
</tr>
<tr>
<td>0b1</td>
<td>Using the Long-descriptor translation table formats.</td>
</tr>
</tbody>
</table>

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [8:6]**

Reserved, RES0.

**STATUS, bits [5:0]**

Fault status bits. Possible values of this field are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000000</td>
<td>Address size fault in translation table base register.</td>
</tr>
<tr>
<td>0b000001</td>
<td>Address size fault, level 1.</td>
</tr>
<tr>
<td>0b000010</td>
<td>Address size fault, level 2.</td>
</tr>
<tr>
<td>0b000011</td>
<td>Address size fault, level 3.</td>
</tr>
<tr>
<td>0b000101</td>
<td>Translation fault, level 1.</td>
</tr>
<tr>
<td>0b000110</td>
<td>Translation fault, level 2.</td>
</tr>
<tr>
<td>0b001110</td>
<td>Translation fault, level 3.</td>
</tr>
<tr>
<td>0b010001</td>
<td>Access flag fault, level 1.</td>
</tr>
<tr>
<td>0b010010</td>
<td>Access flag fault, level 2.</td>
</tr>
<tr>
<td>0b010111</td>
<td>Access flag fault, level 3.</td>
</tr>
<tr>
<td>0b011001</td>
<td>Permission fault, level 1.</td>
</tr>
<tr>
<td>0b011100</td>
<td>Permission fault, level 2.</td>
</tr>
</tbody>
</table>


0b001111  Permission fault, level 3.
0b010000  Synchronous External abort, not on translation table walk.
0b010001  Asynchronous SError interrupt.
0b010101  Synchronous External abort on translation table walk, level 1.
0b010110  Synchronous External abort on translation table walk, level 2.
0b010111  Synchronous External abort on translation table walk, level 3.
0b011000  When FEAT_RAS is not implemented:
               Synchronous parity or ECC error on memory access, not on translation table walk.
0b011001  When FEAT_RAS is not implemented:
               Asynchronous SError interrupt, from a parity or ECC error on memory access.
0b011110  When FEAT_RAS is not implemented:
               Synchronous parity or ECC error on memory access on translation table walk, level 1.
0b011111  When FEAT_RAS is not implemented:
               Synchronous parity or ECC error on memory access on translation table walk, level 2.
0b011111  When FEAT_RAS is not implemented:
               Synchronous parity or ECC error on memory access on translation table walk, level 3.
0b100001  Alignment fault.
0b100100  Debug exception.
0b110000  TLB conflict abort.
0b110100  IMPLEMENTATION DEFINED fault (Lockdown).
0b110101  IMPLEMENTATION DEFINED fault (Unsupported Exclusive access).
All other values are reserved.

For more information about the lookup level associated with a fault, see The level associated with MMU faults on a Long-descriptor translation table lookup on page G5-9259.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DFSR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
   UNDEFINED;
elsif PSTATE.EL == EL1 then
   if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
   elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
      AArch32.TakeHypTrapException(0x03);
   elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
   elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
      AArch32.TakeHypTrapException(0x03);
   elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
      R[t] = DFSR_NS;
   else
      \[coproc\] opc1 CRn CRm opc2
      0b1111 0b000 0b0101 0b0000 0b000
   end
end
R[t] = DFSR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        R[t] = DFSR_NS;
    else
        R[t] = DFSR;
    end elsif
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        R[t] = DFSR_S;
    else
        R[t] = DFSR_NS;
    end elsif
else
    R[t] = DFSR_NS;
end elsif

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
G8.2.47  DTLBIALL, Data TLB Invalidate All

The DTLBIALL characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from data TLBs that are from any level of the translation table walk. The entries that are invalidated are as follows:

- If executed at EL1, all entries that:
  - Would be required for the EL1&0 translation regime.
  - Match the current VMID, if EL2 is implemented and enabled in the current Security state.
- If executed in Secure state when EL3 is using AArch32, all entries that would be required for the Secure PL1&0 translation regime.
- If executed at EL2, and if EL2 is enabled in the current Security state, the stage 1 or stage 2 translation table entries that would be required for the Non-secure PL1&0 translation regime and matches the current VMID.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backwards compatibility with earlier versions of the Arm architecture.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DTLBIALL are UNDEFINED.

**Attributes**

DTLBIALL is a 32-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

**Executing DTLBIALL instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{MCR}\{<c>\}{<q>} \text{ coproc}, \{#\}\text{opc1}, <Rt>, <CRn>, <CRm>{, \{#\}opc2}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & ELUsingAArch32(EL2) & HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.TTLB == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR.TTLB == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    if IsFeatureImplemented(FEAT_XS) & ELUsingAArch32(EL2) & IsFeatureImplemented(FEAT_HCX) &
      IsHCRX_EL2Enabled() & HCRX_EL2.FnXS == '1' then
      AArch32.TakeHypTrapException(0x03);
```
AArch32.DTLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_NSH, TLBI_ExcludeXS);
else
AArch32.DTLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_NSH, TLBI_AllAttr);
elsif PSTATE.EL == EL2 then
AArch32.DTLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_NSH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
AArch32.DTLBI_ALL(SecurityStateAtEL(EL3), Regime_EL30, Shareability_NSH, TLBI_AllAttr);
G8.2.48 DTLBIASID, Data TLB Invalidate by ASID match

The DTLBIASID characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from data TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backwards compatibility with earlier versions of the Arm architecture.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DTLBIASID are UNDEFINED.

**Attributes**

DTLBIASID is a 32-bit System instruction.

**Field descriptions**

```
       31  8  7  0
          RES0  ASID
```

**Bits [31:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this System instruction.

**Executing DTLBIASID instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0110</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    if IsFeatureImplemented(FEAT_XS) && !ELUsingAArch32(EL2) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then
      AArch32.DTLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_ExcudeXS, R[t]);
    else
      AArch32.DTLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AllAttr, R[t]);
    endif;
  endif;
elsif PSTATE.EL == EL2 then
  AArch32.DTLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AllAttr, R[t]);
elsif PSTATE.EL == EL3 then
  AArch32.DTLBI_ASID(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_NSH, TLBI_AllAttr, R[t]);
DTLBIMVA, Data TLB Invalidate by VA

The DTLBIMVA characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from data TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backwards compatibility with earlier versions of the Arm architecture.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DTLBIMVA are UNDEFINED.

**Attributes**

DTLBIMVA is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>VA, bits [31:12]</td>
</tr>
<tr>
<td>12</td>
<td>Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.</td>
</tr>
<tr>
<td>11</td>
<td>Bits [11:8]</td>
</tr>
<tr>
<td>8</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>0</td>
<td>ASID, bits [7:0]</td>
</tr>
<tr>
<td></td>
<td>ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.</td>
</tr>
<tr>
<td></td>
<td>Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.</td>
</tr>
</tbody>
</table>
Executing DTLBIMVA instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{MCR}\{<c>\}{<q>}, \{#<opc1>, <Rt>, <CRn>, <CRm>{, \{#<opc2}\}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0110</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
elif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    if IsFeatureImplemented(FEAT_XS) && !ELUsingAArch32(EL2) && IsFeatureImplemented(FEAT_HCX) &&
      IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then
      AArch32.DTLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_ExcludeXS, R[t]);
    else
      AArch32.DTLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, R[t]);
    endif
  endif
elseif PSTATE.EL == EL2 then
  AArch32.DTLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, R[t]);
elseif PSTATE.EL == EL3 then
  AArch32.DTLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, R[t]);
G8.2.50   DVPRCTX, Data Value Prediction Restriction by Context

The DVPRCTX characteristics are:

**Purpose**

Data Value Prediction Restriction by Context applies to all Data Value Prediction Resources that predict execution based on information gathered within the target execution context or contexts.

Data value predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control speculative execution occurring after the instruction is complete and synchronized.

This instruction is guaranteed to be complete following a DSBI that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

___ **Note** ___

This instruction does not require the invalidation of prediction structures so long as the behavior described for completion of this instruction is met by the implementation.

On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

**Configurations**

This instruction is present only when AArch32 is supported and FEAT SPECRES is implemented. Otherwise, direct accesses to DVPRCTX are UNDEFINED.

**Attributes**

DVPRCTX is a 32-bit System instruction.

**Field descriptions**

```
  | 31  | 28  | 27  | 26  | 25  | 24  | 16  | 15  | 9   | 8   | 7   | 0   |
  +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----|
  RES0 | NS  | EL  | VMID| RES0| ASID| GASID|
  +-----+-----+-----+-----+-----+-----+-----|
```

**Bits [31:28]**

Reserved, RES0.

**GVMID, bit [27]**

Execution of this instruction applies to all VMIDs or a specified VMID.

0b0  Applies to specified VMID for an EL0 or EL1 target execution context.

0b1  Applies to all VMIDs for an EL0 or EL1 target execution context.

For target execution contexts other than EL0 or EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, this field has an Effective value of 0.

If EL2 is not implemented or not enabled for the target Security state, this field is RES0.

**NS, bit [26]**

Security State.

0b0  Secure state.

0b1  Non-secure state.

If the instruction is executed in Non-secure state, this field has an Effective value of 1.
EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.
0b00 EL0.
0b01 EL1.
0b10 EL2.
0b11 EL3.
If the instruction is executed at an Exception level lower than the specified level, this instruction is treated as a NOP.

VMID, bits [23:16]

Only applies when bit[27] is 0 and the target execution context is either:
• EL1.
• EL0 when (HCR_EL2.E2H==0 or HCR_EL2.TGE==0) or EL2 is using AArch32 state.
Otherwise this field is res0.
When the instruction is executed at EL1, this field is treated as the current VMID.
When the instruction is executed at EL0 and (HCR_EL2.E2H==0 or HCR_EL2.TGE==0 or ELUsingAArch32(EL2)), this field is treated as the current VMID.
When the instruction is executed at EL0 and (HCR_EL2.E2H==1 and HCR_EL2.TGE==1 and !ELUsingAArch32(EL2)), this field is ignored.
If EL2 is not implemented or not enabled for the target Security state, this field is res0.

Bits [15:9]

Reserved, res0.

GASID, bit [8]

Execution of this instruction applies to all ASIDs or a specified ASID.
0b0 Applies to specified ASID for an EL0 target execution context.
0b1 Applies to all ASIDs for an EL0 target execution context.
For target execution contexts other than EL0, this field is res0.
If the instruction is executed at EL0, this field has an Effective value of 0.

ASID, bits [7:0]

Only applies for an EL0 target execution context and when bit[8] is 0.
Otherwise, this field is res0.
When the instruction is executed at EL0, this field is treated as the current ASID.

Executing DVPRCTX instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[\text{MCR}\{<c>\}<q>\ <\text{coproc}, \#<\text{opc1}, <Rt>, <CRn>, <CRm>, \#<\text{opc2})\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0011</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.EnRCTX == '0'
  then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
  AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif ELUsingAArch32(EL1) && SCTLR.EnRCTX == '0' then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
    AArch32.TakeHypTrapException(0x00);
  else
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
else
  AArch32.RestrictPrediction(R[t], RestrictType_DataValue);
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.NV == '1' then
    AArch64.SystemAccessTrap(EL2, 0x03);
  else
    AArch32.RestrictPrediction(R[t], RestrictType_DataValue);
elsif PSTATE.EL == EL2 then
  AArch32.RestrictPrediction(R[t], RestrictType_DataValue);
elsif PSTATE.EL == EL3 then
  AArch32.RestrictPrediction(R[t], RestrictType_DataValue);
G8.2.51 **ELR_hyp, Exception Link Register (Hyp mode)**

The ELR_hyp characteristics are:

**Purpose**
When taking an exception to Hyp mode, holds the address to return to.

**Configurations**
AArch32 System register ELR_hyp bits [31:0] are architecturally mapped to AArch64 System register ELR_EL2[31:0].
This register is present only when AArch32 is supported. Otherwise, direct accesses to ELR_hyp are UNDEFINED.

**Attributes**
ELR_hyp is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th>Return address</th>
</tr>
</thead>
</table>

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing ELR_hyp**
ELR_hyp is accessible only at Hyp mode and Monitor mode.
Accesses to this register use the following encodings in the System register encoding space:

\[ MRS\{<c>\}{<q>} <Rd>, ELR_hyp \]

<table>
<thead>
<tr>
<th>R</th>
<th>M</th>
<th>M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>0b1110</td>
</tr>
</tbody>
</table>

\[ MSR\{<c>\}{<q>} ELR_hyp, <Rn> \]

<table>
<thead>
<tr>
<th>R</th>
<th>M</th>
<th>M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>0b1110</td>
</tr>
</tbody>
</table>
G8.2.52  FCSEIDR, FCSE Process ID register

The FCSEIDR characteristics are:

**Purpose**

Identifies whether the Fast Context Switch Extension (FCSE) is implemented.

From Armv8, the FCSE is not implemented, so this register is RAZ/WI. Software can access this register to determine that the implementation does not include the FCSE.

**Configurations**

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to FCSEIDR are UNDEFINED.

**Attributes**

FCSEIDR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RAZ/WI.</td>
</tr>
</tbody>
</table>

Accessing FCSEIDR

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{\text{c}\}\{\text{q}\}\ , \{\text{#}\}\text{coproc}, \text{<Rt>}, \text{<CRn>}, \text{<CRm>}, \{\text{#}\}\text{opc2}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        R[t] = FCSEIDR;
    elsif PSTATE.EL == EL2 then
        R[t] = FCSEIDR;
    elsif PSTATE.EL == EL3 then
        R[t] = FCSEIDR;
### MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        FCSEIDR = R[t];
    elseif PSTATE.EL == EL2 then
        FCSEIDR = R[t];
    elseif PSTATE.EL == EL3 then
        FCSEIDR = R[t];
G8.2.53 FPEXC, Floating-Point Exception Control register

The FPEXC characteristics are:

**Purpose**

Provides a global enable for the implemented Advanced SIMD and floating-point functionality, and reports floating-point status information.

**Configurations**

AArch32 System register FPEXC bits [31:0] are architecturally mapped to AArch64 System register FPEXC32_EL2[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to FPEXC are **UNDEFINED**.

Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

**Attributes**

FPEXC is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>EX</th>
<th>EN</th>
<th>VV</th>
<th>RES0</th>
<th>VECITR</th>
<th>RES0</th>
<th>TFV</th>
<th>DEX</th>
<th>FP2V</th>
<th>IDF</th>
<th>IXF</th>
<th>UFF</th>
<th>DZF</th>
<th>IOF</th>
<th>OFF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**EX, bit [31]**

Exception bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

Access to this field is **RAZ/WI**.

**EN, bit [30]**

Enables access to the Advanced SIMD and floating-point functionality from all Exception levels, except that setting this field to 0 does not disable the following:

- VMSR accesses to the FPEXC or FPSID.
- VMRS accesses from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2.

0b0 Accesses to the FPSCR, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers, are **UNDEFINED** at all Exception levels.

0b1 This control permits access to the Advanced SIMD and floating-point functionality at all Exception levels.

Execution of Advanced SIMD and floating-point instructions in AArch32 state can be disabled or trapped by the following controls:

- CPACR.cp10, or, if executing at EL0, CPACR_EL1.FPEN.
- FPEXC.EN.
- If executing in Non-secure state:
  - HCPTR.TCP10, or if EL2 is using AArch64, CPTR_EL2.TFP.
  - NSACR.cp10, or if EL3 is using AArch64, CPTR_EL3.TFP.
For Advanced SIMD instructions only:

- CPACR.ASEDIS.
- If executing in Non-secure state, HCPTR.TASE and NSACR.NSTRCDIS.

See the descriptions of the controls for more information.

### Note

When executing at EL0 using AArch32:

- If EL1 is using AArch64, then the Effective value of FPEXC.EN is 1. This includes when EL2 is using AArch64 and is enabled in the current Security state, HCR_EL2.TGE is 1, and the Effective value of HCR_EL2.RW is 1.
- If EL2 is using AArch64 and is enabled in the current Security state, HCR_EL2.TGE is 1, and the Effective value of HCR_EL2.RW is 0, then it is IMPLEMENTATION DEFINED whether the Effective value of FPEXC.EN is 1 or the value written to FPEXC.EN. However, Arm deprecates using the value of FPEXC.EN to determine behavior.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

### DEX, bit [29]

Defined synchronous exception on floating-point execution.

This field identifies whether a synchronous exception generated by the attempted execution of an instruction was generated by an unallocated encoding. The instruction must be in the encoding space that is identified by the pseudocode function ExecutingCP10or11Instr() returning TRUE. This field also indicates whether the FPEXC.TFV field is valid.

The meaning of this bit is:

0b0 The exception was generated by the attempted execution of an unallocated instruction in the encoding space that is identified by the pseudocode function ExecutingCP10or11Instr(). If FPEXC.TFV is RW then it is invalid and UNKNOWN. If FPEXC.{IDF,IXF,UFF,OFF,DZF,IOF} are RW then they are invalid and UNKNOWN.

0b1 The exception was generated during the execution of an allocated encoding. FPEXC.TFV is valid and indicates the cause of the exception.

On an exception that sets this bit to 1 the exception-handling routine must clear this bit to 0.

On an implementation that both does not support trapping of floating-point exceptions and implements the FPSCR.\{Stride, Len\} fields as RAZ, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### FP2V, bit [28]

FPINST2 instruction valid bit. From Armv8, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RES0.

### VV, bit [27]

VECITR valid bit. From Armv8, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RES0.
TFV, bit [26]

Trapped Fault Valid bit. Valid only when the value of FPEXC.DEX is 1. When valid, it indicates the cause of the exception and therefore whether FPEXC. {IDF, IXF, UFF, OFF, DZF, IOF} are valid.

0b0 The exception was caused by the execution of a floating-point VABS, VADD, VDIV, VFMA, VFMS, VFNMA, VFNMS, VMLA, VMLS, VMOV, VMUL, VNEG, VNMLA, VNMLS, VNMUL, VSQRT, or VSUB instruction when one or both of FPSCR. {Stride, Len} was non-zero. If FPEXC. {IDF, IXF, UFF, OFF, DZF, IOF} are RW then they are invalid and UNKNOWN.

0b1 FPEXC. {IDF, IXF, UFF, OFF, DZF, IOF} indicate the presence of trapped floating-point exceptions that had occurred at the time of the exception. Bits are set for all trapped exceptions that had occurred at the time of the exception.

This bit returns a status value and ignores writes.

When the value of FPEXC.DEX is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

• When !ImpDefBool("Support trapping of floating-point exceptions"), access to this field is RAZ/WI.

• When ImpDefBool("Implemented FPSCR LEN, STRIDE as RAZ"), access to this field is RAO/WI.

Bits [25:11]

Reserved, RES0.

VECITR, bits [10:8]

Vector iteration count. From Armv8, this field is RES1.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RES1.

IDF, bit [7]

Input Denormal trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Input Denormal exception occurred while FPSCR.IDE was 1:

0b0 Input Denormal exception has not occurred.

0b1 Input Denormal exception has occurred.

Input Denormal exceptions can occur only when FPSCR.FZ is 1.

Note

A half-precision floating-point value that is flushed to zero because the value of FPSCR.FZ16 is 1 does not generate an Input Denormal exception.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.
IXF, bit [4]
Inexact trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Inexact exception occurred while FPSCR.IXE was 1:

0b0  Inexact exception has not occurred.
0b1  Inexact exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.
On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

UFF, bit [3]
Underflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Underflow exception occurred while FPSCR.UFE was 1:

0b0  Underflow exception has not occurred.
0b1  Underflow exception has occurred.

Underflow trapped exceptions can occur:
• On half-precision data-processing instructions only when FPSCR.FZ16 is 0.
• Otherwise only when FPSCR.FZ is 0.

This bit must be cleared to 0 by the exception-handling routine.
When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.
On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

OFF, bit [2]
Overflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Overflow exception occurred while FPSCR.OFE was 1:

0b0  Overflow exception has not occurred.
0b1  Overflow exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.
When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.
On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

DZF, bit [1]
Divide by Zero trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether a Divide by Zero exception occurred while FPSCR.DZE was 1:

0b0  Divide by Zero exception has not occurred.
0b1  Divide by Zero exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.
When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.
On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IOE, bit [0]**

Invalid Operation trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Invalid Operation exception occurred while FPSCR.IOE was 1:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Invalid Operation exception has not occurred.</td>
</tr>
<tr>
<td>0b1</td>
<td>Invalid Operation exception has occurred.</td>
</tr>
</tbody>
</table>

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

On an implementation that does not support the trapping of floating-point exceptions this bit is RAZ/WI.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing FPEXC**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{VMRS}\{<c>}\{<q>\} <Rt>, <\text{spec}\_\text{reg}> \\
\text{reg} \\
0b1000
\]

```c
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif (ELUsingAArch32(EL3) & SCR.NS == '1' & NSACR.cp10 == '0') || CPACR.cp10 == '00' then
        UNDEFINED;
    elsif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.E2H != '1' & CPTR_EL2.TFP == '1' then
        AArch32.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.E2H == '1' & CPTR_EL2.FPEN == 'x0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() & !ELUsingAArch32(EL2) & ((ELUsingAArch32(EL3) & SCR.NS == '1' & NSACR.cp10 == '0') || HCPR.TCP10 == '1') then
        AArch32.TakeHypTrapException(0x08);
    elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & CPTR_EL3.TFP == '1' then
        if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x07);
        else
            R[t] = FPEXC;
        elsif PSTATE_EL == EL2 then
            if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & CPTR_EL3.TFP == '1' then
                UNDEFINED;
            elsif EL2Enabled() & ((ELUsingAArch32(EL3) & SCR.NS == '1' & NSACR.cp10 == '0') || HCPR.TCP10 == '1') then
                AArch32.TakeHypTrapException(0x08);
            elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & CPTR_EL3.TFP == '1' then
                if Halted() & EDSCR.SDD == '1' then
                    UNDEFINED;
                else
                    AArch64.AArch32SystemAccessTrap(EL3, 0x07);
        end
```
else
  R[t] = FPEXC;
elsif PSTATE_EL == EL3 then
  if CPACR.cp10 == '00' then
    UNDEFINED;
  else
    R[t] = FPEXC;

VMSR(<c>[<q>]<spec_reg>, <Rt>)

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x07);
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x07);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && (ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPTR_EL3.TFP == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x07);
  endif
else
  FPEXC = R[t];
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elseif EL2Enabled() && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || HCR_EL3.TCP10 == '1') then
    AArch32.TakeHypTrapException(0x00);
  elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x07);
    endif
else
  FPEXC = R[t];
elsif PSTATE_EL == EL3 then
  if CPACR.cp10 == '00' then
    UNDEFINED;
  else
    FPEXC = R[t];

G8.2.54 FPSCR, Floating-Point Status and Control Register

The FPSCR characteristics are:

**Purpose**

Provides floating-point system status information and control.

**Configurations**

AArch32 System register FPSCR bits [31:27] are architecturally mapped to AArch64 System register FPSR[31:27].

AArch32 System register FPSCR bit [7] is architecturally mapped to AArch64 System register FPSR[7].

AArch32 System register FPSCR bits [4:0] are architecturally mapped to AArch64 System register FPSR[4:0].

AArch32 System register FPSCR bits [26:15] are architecturally mapped to AArch64 System register FPCR[26:15].

AArch32 System register FPSCR bits [12:8] are architecturally mapped to AArch64 System register FPCR[12:8].

This register is present only when AArch32 is supported. Otherwise, direct accesses to FPSCR are UNDEFINED.

It is IMPLEMENTATION DEFINED whether the Len and Stride fields can be programmed to non-zero values, which will cause some AArch32 floating-point instruction encodings to be UNDEFINED, or whether these fields are RAZ.

Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

**Attributes**

FPSCR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>N</td>
<td>Negative condition flag. This is updated by floating-point comparison operations.</td>
</tr>
<tr>
<td>30</td>
<td>Z</td>
<td>Zero condition flag. This is updated by floating-point comparison operations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N, bit [31]

Negative condition flag. This is updated by floating-point comparison operations.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero condition flag. This is updated by floating-point comparison operations.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.
C, bit [29]

Carry condition flag. This is updated by floating-point comparison operations.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow condition flag. This is updated by floating-point comparison operations.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

QC, bit [27]

Cumulative saturation bit, Advanced SIMD only. This bit is set to 1 to indicate that an Advanced SIMD integer operation has saturated since 0 was last written to this bit.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

AHP, bit [26]

Alternative half-precision control bit:
0b0 IEEE half-precision format selected.
0b1 Alternative half-precision format selected.
This bit is used only for conversions between half-precision floating-point and other floating-point formats.
The data-processing instructions added as part of the FEAT_FP16 extension always use the IEEE half-precision format, and ignore the value of this bit.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

DN, bit [25]

Default NaN mode control bit:
0b0 NaN operands propagate through to the output of a floating-point operation.
0b1 Any operation involving one or more NaNs returns the Default NaN.
The value of this bit controls only scalar floating-point arithmetic. Advanced SIMD arithmetic always uses the Default NaN setting, regardless of the value of the DN bit.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

FZ, bit [24]

Flush-to-zero mode control bit:
0b0 Flush-to-zero mode disabled. Behavior of the floating-point system is fully compliant with the IEEE 754 standard.
0b1 Flush-to-zero mode enabled.
The value of this bit controls only scalar floating-point arithmetic. Advanced SIMD arithmetic always uses the Flush-to-zero setting, regardless of the value of the FZ bit.
This bit has no effect on half-precision calculations.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

RMode, bits [23:22]

Rounding Mode control field. The encoding of this field is:
0b00 Round to Nearest (RN) mode.
0b01 Round towards Plus Infinity (RP) mode.
0b10 Round towards Minus Infinity (RM) mode.
0b11 Round towards Zero (RZ) mode.

The specified rounding mode is used by almost all scalar floating-point instructions. Advanced SIMD arithmetic always uses the Round to Nearest setting, regardless of the value of the RMode bits.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Stride, bits [21:20]

It is IMPLEMENTATION DEFINED whether this field is RW or RAZ.

If this field is RW and is set to a value other than zero, some floating-point instruction encodings are UNDEFINED. The instruction pseudocode identifies these instructions.

Arm strongly recommends that software never sets this field to a value other than zero.

The value of this field is ignored when processing Advanced SIMD instructions.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

FZ16, bit [19]

When FEAT_FP16 is implemented:

Flush-to-zero mode control bit on half-precision data-processing instructions:

0b0 Flush-to-zero mode disabled. Behavior of the floating-point system is fully compliant with the IEEE 754 standard.
0b1 Flush-to-zero mode enabled.

The value of this bit applies to both scalar and Advanced SIMD floating-point half-precision calculations.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Len, bits [18:16]

It is IMPLEMENTATION DEFINED whether this field is RW or RAZ.

If this field is RW and is set to a value other than zero, some floating-point instruction encodings are UNDEFINED. The instruction pseudocode identifies these instructions.

Arm strongly recommends that software never sets this field to a value other than zero.

The value of this field is ignored when processing Advanced SIMD instructions.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

IDE, bit [15]

Input Denormal floating-point exception trap enable.

0b0 Untrapped exception handling selected. If the floating-point exception occurs, the IDC bit is set to 1.
0b1 Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the IDC bit.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RAZ/WI.
When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [14:13]**

Reserved, RES0.

**IXE, bit [12]**

Inexact floating-point exception trap enable.
- 0b0: Untrapped exception handling selected. If the floating-point exception occurs, the IXC bit is set to 1.
- 0b1: Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the IXC bit.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RAZ/WI.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**UFE, bit [11]**

Underflow floating-point exception trap enable.
- 0b0: Untrapped exception handling selected. If the floating-point exception occurs, the UFC bit is set to 1.
- 0b1: Trapped exception handling selected. If the floating-point exception occurs and Flush-to-zero is not enabled, the PE does not update the UFC bit.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RAZ/WI.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**OFE, bit [10]**

Overflow floating-point exception trap enable.
- 0b0: Untrapped exception handling selected. If the floating-point exception occurs, the OFC bit is set to 1.
- 0b1: Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the OFC bit.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RAZ/WI.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**DZE, bit [9]**

Divide by Zero floating-point exception trap enable.
- 0b0: Untrapped exception handling selected. If the floating-point exception occurs, the DZC bit is set to 1.
This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RAZ/WI. When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**IOE, bit [8]**

Invalid Operation floating-point exception trap enable.

0b0 Untrapped exception handling selected. If the floating-point exception occurs, the IOC bit is set to 1.

0b1 Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the IOC bit.

This bit is RW only if the implementation supports the trapping of floating-point exceptions. In an implementation that does not support floating-point exception trapping, this bit is RAZ/WI. When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**IDC, bit [7]**

Input Denormal cumulative floating-point exception bit. This bit is set to 1 to indicate that the Input Denormal floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the IDE bit.

Advanced SIMD instructions set this bit if the Input Denormal floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the IDE bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Bits [6:5]**

Reserved, **RES0**.

**IXC, bit [4]**

Inexact cumulative floating-point exception bit. This bit is set to 1 to indicate that the Inexact floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the IXE bit.

Advanced SIMD instructions set this bit if the Inexact floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the IXE bit.

The criteria for the Inexact floating-point exception to occur are different in Flush-to-zero mode. For more information, see *Flushing denormalized numbers to zero* on page A1-58.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**UFC, bit [3]**

Underflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Underflow floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the UFE bit.
Advanced SIMD instructions set this bit if the Underflow floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, if FPSCR.UFE is 0 or if Flush-to-zero is enabled.

The criteria for the Underflow floating-point exception to occur are different in Flush-to-zero mode. For more information, see Flushing denormalized numbers to zero on page A1-58.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OFC, bit [2]

Overflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Overflow floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the OFE bit.

Advanced SIMD instructions set this bit if the Overflow floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the OFE bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DZC, bit [1]

Divide by Zero cumulative floating-point exception bit. This bit is set to 1 to indicate that the Divide by Zero floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the DZE bit.

Advanced SIMD instructions set this bit if the Divide by Zero floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the DZE bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IOC, bit [0]

Invalid Operation cumulative floating-point exception bit. This bit is set to 1 to indicate that the Invalid Operation floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the IOE bit.

Advanced SIMD instructions set this bit if the Invalid Operation floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the IOE bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FPSCR

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{VMRS\{<c>\}{<q>} <Rt>, <spec\_reg>}
\]

\[
\begin{array}{l}
\text{if } \text{PSTATE.EL} = \text{EL0} \text{ then} \\
\quad \text{if Halted()} \& \& \text{HaveEL(EL3)} \& \& \text{EDSCR.SDD} = '1' \& \& \text{boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \& \& !ELUsingArch32(EL3) \& \& CPTR_EL3.TFP == '1' then} \\
\quad \text{UNDEFINED;}
\end{array}
\]
elsif !ELUsingAArch32(EL1) && ((EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.FPEN != '11') then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x00);
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x07);
  end if;
elsif ELUsingAArch32(EL1) && ((ELUsingAArch32(EL3) && SCR.NS == '1') && NSACR.cp10 == '0') ||
  CPACR.cp10 == '0x') then
  UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.FPEN != '11' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x07);
elsif EL2Enabled() && ELUsingAArch32(EL2) && (ELUsingAArch32(EL3) && SCR.NS == '1') && NSACR.cp10 == '0') ||
  HCPTR.TCP10 == '1') then
  AArch32.TakeHypTrapException(0x08);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x07);
  end if;
else
  R[t] = FPSCR;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    UNDEFINED;
elsif CPACR_EL1.FPEN == 'x0' then
  AArch64.AArch32SystemAccessTrap(EL1, 0x07);
elsif (ELUsingAArch32(EL3) && SCR.NS == '1') && NSACR.cp10 == '0') ||
  CPACR.cp10 == '00' then
  UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x07);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.TFP == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x07);
else
  R[t] = FPSCR;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    UNDEFINED;
elsif EL2Enabled() && ((ELUsingAArch32(EL3) && SCR.NS == '1') && NSACR.cp10 == '0') ||
  HCPTR.TCP10 == '1') then
  AArch32.TakeHypTrapException(0x08);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x07);
  end if;
else
  R[t] = FPSCR;
elsif PSTATE.EL == EL3 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    UNDEFINED;
elsif EL2Enabled() && ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') ||
  CPACR.cp10 == '0' then
  UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0x07);
else
  R[t] = FPSCR;
VMSR{<c>}{<q>}, <spec_reg>, <Rt>

```c
if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.FPEN != '11' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x07);
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x07);
  end
elsif EL2Enabled() && (ELUsingAArch32(EL3) & SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10 == '0x' then
  UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.FPEN != '11' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x07);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x07);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x07);
elsif EL2Enabled() && ELUsingAArch32(EL1) && (ELUsingAArch32(EL3) & SCR.NS == '1' && NSACR.cp10 == '0') || HCPTR.TCP10 == '1' then
  AArch32.TakeHypTrapException(0x08);
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & SCR.NS == '1' & NSACR.cp10 == '0' then
  UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0x07);
else
  FPSCR = R[t];
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    UNDEFINED;
elsif CPACR_EL1.FPEN == '0x' then
  AArch64.AArch32SystemAccessTrap(EL1, 0x07);
elsif (ELUsingAArch32(EL3) & SCR.NS == '1' & NSACR.cp10 == '0') || CPACR.cp10 == '00' then
  UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x07);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x07);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN != '11' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x07);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x07);
elsif EL2Enabled() && ELUsingAArch32(EL1) && (ELUsingAArch32(EL3) & SCR.NS == '1' && NSACR.cp10 == '0') || HCPTR.TCP10 == '0' then
  AArch32.TakeHypTrapException(0x08);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0x07);
else
  FPSCR = R[t];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    UNDEFINED;
elsif EL2Enabled() && ((ELUsingAArch32(EL3) & SCR.NS == '1' & NSACR.cp10 == '0') || HCPTR.TCP10 == '1') then
  AArch32.TakeHypTrapException(0x08);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
```

```
if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x07);
else
    FPSCR = R[t];
elsif PSTATE.EL == EL3 then
    if CPACR.cp10 == '00' then
        UNDEFINED;
    else
        FPSCR = R[t];
    end
else
    FPSCR = R[t];
G8.2.55   FPSID, Floating-Point System ID register

The FPSID characteristics are:

**Purpose**

Provides top-level information about the floating-point implementation.

This register largely duplicates information held in the MIDR. Arm deprecates use of it.

**Configurations**

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to FPSID are UNDEFINED.

Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

**Attributes**

FPSID is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-24</td>
<td>Implementer</td>
</tr>
<tr>
<td>23</td>
<td>SW</td>
</tr>
<tr>
<td>22-16</td>
<td>Subarchitecture</td>
</tr>
<tr>
<td>15-8</td>
<td>PartNum</td>
</tr>
<tr>
<td>7-4</td>
<td>Variant</td>
</tr>
<tr>
<td>3-0</td>
<td>Revision</td>
</tr>
</tbody>
</table>

**Implementer, bits [31:24]**

Implementer codes are the same as those used for the MIDR.

For an implementation by Arm this field is 0x41, the ASCII code for A.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**SW, bit [23]**

Software bit. Defined values are:

0b0  The implementation provides a hardware implementation of the floating-point instructions.

0b1  The implementation supports only software emulation of the floating-point instructions.

In Armv8-A, the only permitted value is 0b0.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Subarchitecture, bits [22:16]**

Subarchitecture version number. For an implementation by Arm, defined values are:

0b0000000  VFPv1 architecture with an IMPLEMENTATION DEFINED subarchitecture.

0b0000001  VFPv2 architecture with Common VFP subarchitecture v1.

0b0000010  VFPv3 architecture, or later, with Common VFP subarchitecture v2. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers.

0b0000011  VFPv3 architecture, or later, with Null subarchitecture. The entire floating-point implementation is in hardware, and no software support code is required. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers. This value can be used only by an implementation that does not support the trap enable bits in the FPSCR.

0b0000100  VFPv3 architecture, or later, with Common VFP subarchitecture v3, and support for trap enable bits in FPSCR. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers.
For a subarchitecture designed by Arm the most significant bit of this field, register bit[22], is 0. Values with a most significant bit of 0 that are not listed here are reserved.

When the subarchitecture designer is not Arm, the most significant bit of this field, register bit[22], must be 1. Each implementer must maintain its own list of subarchitectures it has designed, starting at subarchitecture version number 0x40.

In Armv8-A, the permitted values are 0b0000011 and 0b0000100.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**PartNum, bits [15:8]**

Part Number for the floating-point implementation, assigned by the implementer.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Variant, bits [7:4]**

Variant number. Typically, this field distinguishes between different production variants of a single product.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Revision, bits [3:0]**

Revision number for the floating-point implementation.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Accessing FPSID**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{VMRS}\{<c>\}{<q>} \langle Rt \rangle, \langle \text{spec\_reg} \rangle
\]

\[
\begin{array}{c|c}
\text{reg} & 0b0000 \\
\end{array}
\]

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elsif (ELUsingAArch32(EL3) && SCR.NS == '1') || CPACR.cp10 == '00' then
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPACR.cp10 == '10' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x07);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x07);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
    AArch32.TakeHypTrapException(0x08);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID0 == '1' then
    AArch32.TakeHypTrapException(0x08);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TID0 == '1' then
    AArch32.TakeHypTrapException(0x08);
else
  if Halted() && EDSCR.SDD == '1' then
    reg 0b0000
  else
    UNDEFINED;
  end if
end if
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elsif EL2Enabled() && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10 == '00') then
    AArch32.TakeHypTrapException(0x00);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && HCR_EL3.TID0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL3, 0x07);
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
      UNDEFINED;
    elsif EL2Enabled() && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10 == '00') then
      AArch32.TakeHypTrapException(0x00);
    elsif Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x07);
  end if;
else
  return;
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
      UNDEFINED;
    elsif EL2Enabled() && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10 == '00') then
      AArch32.TakeHypTrapException(0x00);
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x07);
  end if;
else
  R[t] = FPSID;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elsif EL2Enabled() && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10 == '00') then
    AArch32.TakeHypTrapException(0x00);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x07);
  end if;
else
  R[t] = FPSID;
else
  R[t] = FPSID;
endif

VMSR<qc><qc> <spec_reg>, <Rt>

VMSR<qc><qc> <spec_reg>, <Rt>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elsif EL2Enabled() && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10 == '00') then
    AArch32.TakeHypTrapException(0x00);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x07);
  end if;
else
  return;
  elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
      UNDEFINED;
    elsif EL2Enabled() && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10 == '00') then
      AArch32.TakeHypTrapException(0x00);
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x07);
  end if;
else
  R[t] = FPSID;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elsif EL2Enabled() && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10 == '00') then
    AArch32.TakeHypTrapException(0x00);
  elsif Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x07);
  end if;
else
  R[t] = FPSID;
elsif PSTATE.EL == EL3 then
  if CPACR.cp10 == '00' then
    UNDEFINED;
  else
    R[t] = FPSID;
else
  R[t] = FPSID;
endif
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x07);
  else
    return;
elsif PSTATE.EL == EL3 then
  if CPACR.cp10 == '00' then
    UNDEFINED;
  else
    return;
G8.2.56  HACR, Hyp Auxiliary Configuration Register

The HACR characteristics are:

**Purpose**

Controls trapping to Hyp mode of IMPLEMENTATION DEFINED aspects of Non-secure EL1 or EL0 operation.

**Configurations**

AArch32 System register HACR bits [31:0] are architecturally mapped to AArch64 System register HACR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HACR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HACR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing HACR**

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

```
coproc  opc1  CRn  CRm  opc2  
0b1111  0b100  0b0001  0b0001  0b111
```

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    R[t] = HACR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        R[t] = HACR;
```
### MCR<coproc><cq> <opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}</cq>

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAAArch32(EL2) && HSTR_EL2.T1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAAArch32(EL2) && HSTR.T1 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  HACR = R[t];
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    HACR = R[t];
G8.2.57 HACTLR, Hyp Auxiliary Control Register

The HACTLR characteristics are:

**Purpose**

Controls IMPLEMENTATION DEFINED features of Hyp mode operation.

**Configurations**

AArch32 System register HACTLR bits [31:0] are architecturally mapped to AArch64 System register ACTLR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HACTLR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HACTLR is a 32-bit register.

**Field descriptions**

```
31 0
```

IMPLEMENTATION DEFINED

**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing HACTLR**

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  R[t] = HACTLR;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    R[t] = HACTLR;
### MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  HACTLR = R[t];
else if PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    HACTLR = R[t];
G8.2.58 HACTLR2, Hyp Auxiliary Control Register 2

The HACTLR2 characteristics are:

**Purpose**

Provides additional space to the HACTLR register to hold IMPLEMENTATION DEFINED trap functionality.

**Configurations**

AArch32 System register HACTLR2 bits [31:0] are architecturally mapped to AArch64 System register ACTLR_EL2[63:32].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HACTLR2 are UNDEFINED.

In Armv8.0 and Armv8.1, it is IMPLEMENTATION DEFINED whether this register is implemented, or whether it causes UNDEFINED exceptions when accessed. The implementation of this register can be detected by examining ID_MMFR4.AC2.

From Armv8.2 this register must be implemented.

**Attributes**

HACTLR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing HACTLR2**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC\{<c>\}{<q>} \langle coproc \rangle, \{#\}\langle opc1 \rangle, \langle Rp \rangle, \langle CRn \rangle, \langle CRm\rangle, \{#\}\langle opc2 \rangle
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
    endif
elsif PSTATE.EL == EL2 then
    R[t] = HACTLR2;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        UNDEFINED;
    endif
endif
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elseif PSTATE.EL == EL2 then
    HACTLR2 = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HACTLR2 = R[t];
    end

MCR{<c>}{<q>}<coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b011</td>
</tr>
</tbody>
</table>
G8.2.59   HADFSR, Hyp Auxiliary Data Fault Status Register

The HADFSR characteristics are:

**Purpose**
Provides additional **IMPLEMENTATION DEFINED** syndrome information for Data Abort exceptions taken to Hyp mode.

**Configurations**
AArch32 System register HADFSR bits [31:0] are architecturally mapped to AArch64 System register AFSR0_EL2[31:0].
This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HADFSR are **UNDEFINED**.
This is an optional register. An implementation that does not require this register can implement it as RES0.
If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**
HADFSR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td><strong>IMPLEMENTATION DEFINED</strong></td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [31:0]**
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Accessing HADFSR**
Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC\{c\}\{q\}\{coproc\},\{#\}\{opc1\},\{R\},\{CRn\},\{CRm\}\{,\{#\}\{opc2\}\}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  if EL2Enabled() & !ELUsingAArch32(EL2) & HSTR_EL2.T5 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseself EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T5 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
elseself PSTATE.EL == EL2 then
    R[t] = HADFSR;
elseself PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
else
    R[t] = HADFSR;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}  

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
    else
        PSTATE.EL == EL2 then
            HADFSR = R[t];
    elseif PSTATE.EL == EL3 then
        if SCR.NS == '0' then
            UNDEFINED;
        else
            HADFSR = R[t];
HAIFSR, Hyp Auxiliary Instruction Fault Status Register

The HAIFSR characteristics are:

Purpose
Provides additional IMPLEMENTATION DEFINED syndrome information for Prefetch Abort exceptions taken to Hyp mode.

Configurations
AArch32 System register HAIFSR bits [31:0] are architecturally mapped to AArch64 System register AFSR1_EL2[31:0].
This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HAIFSR are UNDEFINED.
This is an optional register. An implementation that does not require this register can implement it as RES0.
If EL2 is not implemented, this register is RES0 from EL3.

Attributes
HAIFSR is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td>Bits [31:0]</td>
</tr>
<tr>
<td>The reset behavior of this field is:</td>
<td></td>
</tr>
<tr>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
<td></td>
</tr>
</tbody>
</table>

Accessing HAIFSR
Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rp>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  R[t] = HAIFSR;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
else
    R[t] = HAIFSR;

\[
MCR<coproc>,<Rt>,<CRn>,<CRm>,<opc1>,<opc2>\
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
else if PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        HAIFSR = R[t];
    elsif PSTATE.EL == EL3 then
        if SCR.NS == '0' then
            UNDEFINED;
        else
            HAIFSR = R[t];
    else
        HAIFSR = R[t];
G8.2.61 HAMAIR0, Hyp Auxiliary Memory Attribute Indirection Register 0

The HAMAIR0 characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED memory attributes for the memory attribute encodings defined by HMAIR0. These IMPLEMENTATION DEFINED attributes can only provide additional qualifiers for the memory attribute encodings, and cannot change the memory attributes defined in HMAIR0.

**Configurations**

AArch32 System register HAMAIR0 bits [31:0] are architecturally mapped to AArch64 System register AMAIR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HAMAIR0 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HAMAIR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

If an implementation does not provide any IMPLEMENTATION DEFINED memory attributes, this register is RES0.

**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing HAMAIR0**

Accesses to this register use the following encodings in the System register encoding space:

**MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}**

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  R[t] = HAMAIR0;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
else
    R[t] = HAMAIR0;

\text{MCR}\{<c>\}<\{<q>\} \text{<coproc>, \{<opc1>, <Rt>, <CRn>, <CRm>\} \{, \{<opc2>\}\}}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HAMAIR0 = R[t];
else if PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HAMAIR0 = R[t];
HAMAIR1, Hyp Auxiliary Memory Attribute Indirection Register 1

The HAMAIR1 characteristics are:

**Purpose**

Provides IMPLEMENTATION DEFINED memory attributes for the memory attribute encodings defined by HMAIR1. These IMPLEMENTATION DEFINED attributes can only provide additional qualifiers for the memory attribute encodings, and cannot change the memory attributes defined in HMAIR1.

**Configurations**

AArch32 System register HAMAIR1 bits [31:0] are architecturally mapped to AArch64 System register AMAIR_EL2[63:32].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HAMAIR1 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HAMAIR1 is a 32-bit register.

**Field descriptions**

If an implementation does not provide any IMPLEMENTATION DEFINED memory attributes, this register is RES0.

**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing HAMAIR1**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC\{<c>\}{<q>} < coproc > , (#) < opc1 > , < Rt > , < CRn > , < CRm > , ( # ) < opc2 > }
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  R[t] = HAMAIR1;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
else
  R[t] = HAMAIR1;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1010</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>
G8.2.63 HCPTR, Hyp Architectural Feature Trap Register

The HCPTR characteristics are:

Purpose

Controls:

• Trapping to Hyp mode of Non-secure access, at EL1 or EL0, to trace, and to Advanced SIMD and floating-point functionality.

• Hyp mode access to trace, and to Advanced SIMD and floating-point functionality.

——— Note ————

Accesses to this functionality:

• From Non-secure modes other than Hyp mode are also affected by settings in the CPACR and NSACR.

• From Hyp mode are also affected by settings in the NSACR.

Exceptions generated by the CPACR and NSACR controls are higher priority than those generated by the HCPTR controls.

Configurations

AArch32 System register HCPTR bits [31:0] are architecturally mapped to AArch64 System register CPTR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HCPTR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes

HCPTR is a 32-bit register.

Field descriptions

TCPAC, bit [31]

Traps Non-secure EL1 accesses to the CPACR to Hyp mode.

0b0 This control does not cause any instructions to be trapped.

0b1 Non-secure EL1 accesses to the CPACR are trapped to Hyp mode.

——— Note ————

The CPACR is not accessible at EL0.

——— The reset behavior of this field is:

• On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.
TAM, bit [30]

*When FEAT_AMUv1 is implemented:*

Trap Activity Monitor access. Traps Non-secure EL1 and EL0 accesses to all Activity Monitor registers to EL2.

- **0b0** Accesses from Non-secure EL1 and EL0 to Activity Monitor registers are not trapped.
- **0b1** Accesses from Non-secure EL1 and EL0 to Activity Monitor registers are trapped to Hyp mode.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

Bits [29:21]

Reserved, RES0.

TTA, bit [20]

Traps Non-secure System register accesses to all implemented trace registers to Hyp mode.

- **0b0** This control does not cause any instructions to be trapped.
- **0b1** Any Non-secure System register access to an implemented trace register is trapped to Hyp mode, unless the access is trapped to EL1 by a CPACR or NSACR control, or the access is from Non-secure EL0 and the definition of the register in the appropriate trace architecture specification indicates that the register is not accessible from EL0. A trapped instruction generates:
  - A Hyp Trap exception, if the exception is taken from Non-secure EL0 or EL1.
  - An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

If the implementation does not include a trace unit, or does not include a System register interface to the trace unit registers, it is IMPLEMENTATION DEFINED whether this bit:

- Is RES0.
- Is RES1.
- Can be written from Hyp mode, and from Secure Monitor mode when SCR.NS is 1.

If EL3 is implemented and is using AArch32, and the value of NSACR.NSTRCDIS is 1, in Non-secure state this field behaves as RAO/WI, regardless of its actual value.

**Note**

- The ETMv4 architecture and ETE do not permit EL0 to access the trace registers. If the trace unit implements FEAT_ETMv4 or FEAT_ETE, EL0 accesses to the trace registers are UNDEFINED, and a resulting Undefined Instruction exception is higher priority than a HCPTR.TTA Hyp Trap exception.
- The Arm architecture does not provide traps on trace register accesses through the optional memory-mapped debug interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
— Otherwise, this field resets to an architecturally UNKNOWN value.

**Bits [19:16]**

Reserved, RES0.

**TASE, bit [15]**

Traps Non-secure execution of Advanced SIMD instructions to Hyp mode when the value of HCPTR.TCP10 is 0.

- **0b0** This control does not cause any instructions to be trapped.
- **0b1** When the value of HCPTR.TCP10 is 0, any attempt to execute an Advanced SIMD instruction in Non-secure state is trapped to Hyp mode, unless it is trapped to EL1 by a CPACR or NSACR control. A trapped instruction generates:
  - A Hyp Trap exception, if the exception is taken from Non-secure EL0 or EL1.
  - An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

When the value of HCPTR.TCP10 is 1, the value of this field is ignored.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1. Otherwise, it is IMPLEMENTATION DEFINED whether this field is implemented as a RW field. If it is not implemented as a RW field, then it is RAZ/WI.

If EL3 is implemented and is using AArch32, and the value of NSACR.NSASEDIS is 1, in Non-secure state this field behaves as RAO/WI, regardless of its actual value. This applies even if the field is implemented as RAZ/WI.

For the list of instructions affected by this field, see Controls of Advanced SIMD operation that do not apply to floating-point operation on page E1-7153.

The reset behavior of this field is:
- On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.

**Bit [14]**

Reserved, RES0.

**Bits [13:12]**

Reserved, RES1.

**TCP11, bit [11]**

When **FEAT_FP** is implemented and **FEAT_AdvSIMD** is implemented:

The value of this field is ignored. If this field is programmed with a different value to the TCP10 bit then this field is UNKNOWN on a direct read of the HCPTR.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1.

If EL3 is implemented and is using AArch32, and the value of NSACR.cp10 is 0, in Non-secure state this field behaves as RAO/WI, regardless of its actual value.

The reset behavior of this field is:
- On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:
- Access is RAO/WI if all of the following are true:
  — EL3 is implemented.
  — EL3 is using AArch32.
—   NSACR.cp10 == 0.

**Otherwise:**

Reserved, RES1.

TCP10, bit [10]

**When FEAT_FP is implemented and FEAT_AdvSIMD is implemented:**

Trap Non-secure accesses to Advanced SIMD and floating-point functionality to Hyp mode:

0b0   This control does not cause any instructions to be trapped.

0b1   Any attempted access to Advanced SIMD and floating-point functionality from Non-secure state is trapped to Hyp mode, unless it is trapped to EL1 by a CPACR or NSACR control. A trapped instruction generates:

•   A Hyp Trap exception, if the exception is taken from Non-secure EL0 or EL1.

•   An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

The Advanced SIMD and floating-point features controlled by these fields are:

•   Execution of any floating-point or Advanced SIMD instruction.

•   Any access to the Advanced SIMD and floating-point registers D0-D31 and their views as S0-S31 and Q0-Q15.

•   Any access to the FPSCR, FPSID, MVFR0, MVFR1, MVFR2, or FPEXC System registers.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1.

If EL3 is implemented and is using AArch32, and the value of NSACR.cp10 is 0, in Non-secure state this field behaves as RAO/WI, regardless of its actual value.

The reset behavior of this field is:

•   On a Warm reset:

  —   When the PE resets into EL2 or EL3, this field resets to 0.

  —   Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

•   Access is RAO/WI if all of the following are true:

  —   EL3 is implemented.

  —   EL3 is using AArch32.


  —   NSACR.cp10 == 0.

**Otherwise:**

Reserved, RES1.

**Bits [9:0]**

Reserved, RES1.

**Accessing HCPTR**

Accesses to this register use the following encodings in the System register encoding space:
### MRC{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>)

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'' && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
            UNDEFINED;
        elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.AArch32SystemAccessTrap(EL3, 0x03);
            end
        else
            R[t] = HCPTR;
        end
        elseif PSTATE.EL == EL3 then
            if SCR.NS == '0' then
                UNDEFINED;
            else
                R[t] = HCPTR;
            end
        else
            HCPTR = R[t];
        end
    end
else
    R[t] = HCPTR;
end

### MCR{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>)

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
    elseif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'' && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
            UNDEFINED;
        elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.AArch32SystemAccessTrap(EL3, 0x03);
            end
        else
            HCPTR = R[t];
        end
    else
        UNDEFINED;
end
else
    HCPTR = R[t];
else
    if SCR.NS == '0' then
        UNDEFINED;
    end
end
else
    HCPR = R[t];
G8.2.64  HCR, Hyp Configuration Register

The HCR characteristics are:

**Purpose**

Provides configuration controls for virtualization, including defining whether various Non-secure operations are trapped to Hyp mode.

**Configurations**

AArch32 System register HCR bits [31:0] are architecturally mapped to AArch64 System register HCR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HCR are **UNDEFINED**.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HCR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit [31]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TRVM, bit [30]</strong></td>
<td>Trap Reads of Virtual Memory controls. Traps Non-secure EL1 reads of the virtual memory control registers to EL2, when EL2 is enabled in the current Security state. The registers for which read accesses are trapped are as follows: SCTLR, TTBR0, TTBR1, TTBCR, TTBCR2, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIFS, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR.</td>
</tr>
<tr>
<td>0b0</td>
<td>This control does not cause any instructions to be trapped.</td>
</tr>
<tr>
<td>0b1</td>
<td>Non-secure EL1 read accesses to the specified Virtual Memory controls are trapped to EL2. The reset behavior of this field is:</td>
</tr>
<tr>
<td>• On a Warm reset:</td>
<td></td>
</tr>
<tr>
<td>— When the PE resets into EL2 or EL3, this field resets to 0.</td>
<td></td>
</tr>
<tr>
<td>— Otherwise, this field resets to an architecturally <strong>UNKNOWN</strong> value.</td>
<td></td>
</tr>
</tbody>
</table>
HCD, bit [29]

When EL3 is not implemented:

HVC instruction disable. Disables Non-secure EL1 and EL2 execution of HVC instructions, when EL2 is enabled in the current Security state.

- **0b0**: HVC instruction execution is enabled at EL2 and EL1.
- **0b1**: HVC instructions are UNDEFINED at EL2 and Non-secure EL1.

The Undefined Instruction exception is taken to the Exception level at which the HVC instruction is executed.

——— Note ————

HVC instructions are always UNDEFINED at EL0.

———

The reset behavior of this field is:

- **On a Warm reset:**
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [28]

Reserved, RES0.

TGE, bit [27]

Trap General Exceptions, from Non-secure EL0.

- **0b0**: This control has no effect on execution at EL0.
- **0b1**: When EL2 is not enabled in the current Security state, this control has no effect on execution at EL0.

When EL2 is enabled in the current Security state, then:

- All exceptions that would be routed to EL1 are routed to EL2.
- The SCTLR.M bit is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR.
- The HCR.{FMO, IMO, AMO} bits are treated as being 1 for all purposes other than returning the result of a direct read of HCR.
- All virtual interrupts are disabled.
- Any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts are disabled.
- An exception return to EL1 is treated as an illegal exception return.
- Monitor mode execution of an MSR or CPS instruction that changes PSTATE.M to a Non-secure EL1 mode is an illegal change to PSTATE.M. For more information see Illegal changes to PSTATE.M on page G1-8925.

Also, when HCR.TGE is 1:

- If EL3 is using AArch32, an attempt to change from a Secure PL1 mode to a Non-secure EL1 mode by changing SCR.NS from 0 to 1 results in SCR.NS remaining as 0.
- The HDCR.{TDRA, TDOSA, TDA, TDE} bits are ignored and treated as being 1 other than for the purpose of a direct read of HDCR.

The reset behavior of this field is:

- **On a Warm reset:**
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.
TVM, bit [26]

Trap Virtual Memory controls. Traps Non-secure EL1 writes to the virtual memory control registers to EL2, when EL2 is enabled in the current Security state.

The registers for which write accesses are trapped are as follows:
SCTLR, TTBR0, TTBR1, TTBCR, TTBCR2, DACR, DFSR, IFSR, DFAR, IFAR, ADFSR, AIIFSR, PRRR, NMRR, MAIR0, MAIR1, AMAIR0, AMAIR1, CONTEXTIDR.

0'b0 This control does not cause any instructions to be trapped.
0'b1 Non-secure EL1 write accesses to the specified virtual memory control registers are trapped to EL2.

The reset behavior of this field is:
• On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.

TTLB, bit [25]

Trap TLB maintenance instructions. Traps Non-secure EL1 execution of a TLBI instruction to EL2, when EL2 is enabled in the current Security state.

This applies to the following instructions:
TLBIAALLIS, TLBIMVAIS, TLBIASIDIS, TLBIMVALIS, TLBIMVAALIS, ITLBIALL, ITLBIJAVA, ITLBIASID, DTLBIALL, DTLBIMVA, DTLBIASID, TBLIALL,
TBLIMVA, TBLIASID, TBLIMVAAL, TBLIMVAAL

0'b0 This control does not cause any instructions to be trapped.
0'b1 Non-secure EL1 accesses to the specified TLB maintenance instructions are trapped to EL2.

The reset behavior of this field is:
• On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.

TPU, bit [24]

Trap cache maintenance instructions that operate to the Point of Unification. Traps Non-secure EL1 execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state.

This applies to the following instructions:
• ICIMVAU, ICIAULLU, ICIAULLUIS, DCCMVAAU.

Note
An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

0'b0 This control does not cause any instructions to be trapped.
0'b1 Non-secure EL1 execution of the specified cache maintenance instructions is trapped to EL2.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.
The reset behavior of this field is:
• On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.

TPC, bit [23]
Trap data or unified cache maintenance instructions that operate to the Point of Coherency. Traps Non-secure EL1 execution of those cache maintenance instructions to EL2, when EL2 is enabled in the current Security state.
This applies to the following instructions:
• DCIMVAC, DCCIMVAC, DCCMVAC.

Note
An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

0b0 This control does not cause any instructions to be trapped.
0b1 Non-secure EL1 execution of the specified cache maintenance instructions is trapped to EL2.

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, invalidate, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 1.

The reset behavior of this field is:
• On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.

TSW, bit [22]
Trap data or unified cache maintenance instructions that operate by Set/Way. Traps Non-secure EL1 execution of those cache maintenance instructions by set/way to EL2, when EL2 is enabled in the current Security state.
This applies to the following instructions:
• DCISW, DCCSW, DCCISW.

Note
An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

0b0 This control does not cause any instructions to be trapped.
0b1 Non-secure EL1 execution of the specified cache maintenance instructions is trapped to EL2.

The reset behavior of this field is:
• On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.

TAC, bit [21]
Trap Auxiliary Control Registers. Traps Non-secure EL1 accesses to the Auxiliary Control Registers to EL2, when EL2 is enabled in the current Security state, from both Execution states.
This applies to the following register accesses:
ACTLR and, if implemented, ACTLR2.

0b0  This control does not cause any instructions to be trapped.

0b1  Non-secure EL1 accesses to the specified registers are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**TIDCP, bit [20]**

Trap IMPLEMENTATION DEFINED functionality. Traps Non-secure EL1 accesses to the encodings for IMPLEMENTATION DEFINED System Registers to EL2, when EL2 is enabled in the current Security state.

MCR and MRC instructions accessing the following encodings:

- All coproc==p15, CRn==c9, Opcode1 == {0-7}, CRm == {c0-c2, c5-c8}, opcode2 == {0-7}.
- All coproc==p15, CRn==c10, Opcode1 == {0-7}, CRm == {c0, c1, c4, c8}, opcode2 == {0-7}.
- All coproc==p15, CRn==c11, Opcode1 == {0-7}, CRm == {c0-c8, c15}, opcode2 == {0-7}.

When HCR.TIDCP is set to 1, it is IMPLEMENTATION DEFINED whether any of this functionality accessed from Non-secure EL0 is trapped to EL2. Otherwise, it is UNDEFINED and the PE takes an Undefined Instruction exception to Non-secure Undefined mode.

0b0  This control does not cause any instructions to be trapped.

0b1  Non-secure EL1 accesses to the specified System register encodings for IMPLEMENTATION DEFINED functionality are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**TSC, bit [19]**

Trap SMC instructions. Traps Non-secure EL1 execution of SMC instructions to Hyp mode.

0b0  This control does not cause any instructions to be trapped.

0b1  Any attempt to execute an SMC instruction at Non-secure EL1 is trapped to Hyp mode, regardless of the value of SCR.SCD.

The Armv8-A architecture permits, but does not require, this trap to apply to conditional SMC instructions that fail their condition code check, in the same way as with traps on other conditional instructions.

--- Note ---

- This trap is only implemented if the implementation includes EL3.
- SMC instructions are always UNDEFINED at PL0.
- This bit traps execution of the SMC instruction. It is not a routing control for the SMC exception. Hyp Trap exceptions and SMC exceptions have different preferred return addresses.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.
TID3, bit [18]
Trap ID group 3. Traps Non-secure EL1 reads of the following registers to EL2, when EL2 is enabled in the current Security state as follows:

- VMRS access to MVFR0, MVFR1, and MVFR2, reported using EC syndrome value 0x88, unless access is also trapped by HCPTR which takes priority.
- MRC access to the following registers are reported using EC syndrome value 0x03:
  - ID_PFR0, ID_PFR1, ID_PFR2, ID_DFR0, ID_AFR0, ID_MMFR0, ID_MMFR1, ID_MMFR2, ID_MMFR3, ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.
  - If FEAT_FGT is implemented:
    - ID_MMFR4 and ID_MMFR5 are trapped to EL2.
    - ID_ISAR6 is trapped to EL2.
    - ID_DFR1 is trapped to EL2.
    - This field traps all MRC accesses to registers in the following range that are not already mentioned in this field description: coproc == p15, opc1 == 0, CRn == c0, CRm == {c2-c7}, opc2 == {0-7}.
  - If FEAT_FGT is not implemented:
    - ID_MMFR4 and ID_MMFR5 are trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_MMFR4 or ID_MMFR5 are trapped.
    - ID_ISAR6 is trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_ISAR6 are trapped to EL2.
    - ID_DFR1 is trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to ID_DFR1 are trapped to EL2.
    - Otherwise, it is IMPLEMENTATION DEFINED whether this bit traps MRC accesses to registers not already mentioned, with coproc == p15, opc1 == 0, CRn == c0, CRm == {c2-c7}, opc2 == {0-7}.
- 0b0  This control does not cause any instructions to be trapped.
- 0b1  The specified Non-secure EL1 read accesses to ID group 3 registers are trapped to EL2.

The reset behavior of this field is:
- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TID2, bit [17]
Trap ID group 2. Traps the following register accesses to EL2, when EL2 is enabled in the current Security state:

- Non-secure EL1 and EL0 reads of the CTR, CCSIDR, CCSIDR2, CLIDR, and CSSELR.
- Non-secure EL1 and EL0 writes to the CSSELR.
- 0b0  This control does not cause any instructions to be trapped.
- 0b1  The specified Non-secure EL1 and EL0 accesses to ID group 2 registers are trapped to EL2.

The reset behavior of this field is:
- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.
TID1, bit [16]

Trap ID group 1. Traps Non-secure EL1 reads of the following registers to EL2, when EL2 is enabled in the current Security state:

TCMTR, TLBTR, REVIDR, AIDR.

0b0 This control does not cause any instructions to be trapped.
0b1 The specified Non-secure EL1 read accesses to ID group 1 registers are trapped to EL2.

The reset behavior of this field is:
• On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.

TID0, bit [15]

Trap ID group 0. Traps the following register accesses to EL2, when EL2 is enabled in the current Security state:

• Non-secure EL1 reads of the JIDR and FPSID.
• If the JIDR is RAZ from Non-secure EL0, Non-secure EL0 reads of the JIDR.

Note
• It is IMPLEMENTATION DEFINED whether the JIDR is RAZ or UNDEFINED at EL0. If it is UNDEFINED at EL0 then the Undefined Instruction exception takes precedence over this trap.
• The FPSID is not accessible at EL0.
• Writes to the FPSID are ignored, and not trapped by this control.

0b0 This control does not cause any instructions to be trapped.
0b1 The specified Non-secure EL1 read accesses to ID group 0 registers are trapped to EL2.

The reset behavior of this field is:
• On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.

TWE, bit [14]

Traps Non-secure EL0 and EL1 execution of WFE instructions to EL2, when EL2 is enabled in the current Security state.

0b0 This control does not cause any instructions to be trapped.
0b1 Any attempt to execute a WFE instruction at Non-secure EL0 or EL1 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWE.

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Note
Since a WFE can complete at any time, even without a Wakeup event, the traps on WFE are not guaranteed to be taken, even if the WFE is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:
• On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.
TWI, bit [13]

Traps Non-secure EL0 and EL1 execution of WFI instructions to EL2, when EL2 is enabled in the current Security state.

0b0  This control does not cause any instructions to be trapped.

0b1  Any attempt to execute a WFI instruction at Non-secure EL0 or EL1 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWI.

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFI can complete at any time, even without a Wakeup event, the traps on WFI are not guaranteed to be taken, even if the WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:
• On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.

DC, bit [12]

Default Cacheability.

0b0  This control has no effect on the Non-secure EL1&0 translation regime.

0b1  In Non-secure state:
  • The SCTLR.M field behaves as 0 for all purposes other than a direct read of the value of the field.
  • The HCR.VM field behaves as 1 for all purposes other than a direct read of the value of the field.
  • The memory type produced by the first stage of the EL1&0 translation regime is Normal Non-Shareable, Inner Write-Back Read-Allocate Write-Allocate, Outer Write-Back Read-Allocate Write-Allocate.

This field has no effect on the EL2 and EL3 translation regimes.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:
• On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.

BSU, bits [11:10]

Barrier Shareability upgrade. This field determines the minimum shareability domain that is applied to any barrier instruction executed from Non-secure EL1 or Non-secure EL0:

0b00  No effect.

0b01  Inner Shareable.

0b10  Outer Shareable.

0b11  Full system.

This value is combined with the specified level of the barrier held in its instruction, using the same principles as combining the shareability attributes from two stages of address translation.
The reset behavior of this field is:
- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**FB, bit [9]**

Force broadcast. Causes the following instructions to be broadcast within the Inner Shareable domain when executed from Non-secure EL1:

- BPIALL, TLBIALLL, TLBIMVA, TLBIASID, DTLBIALLL, DTLBIMVA, DTLBIASID,
- ITLBIALL, ITLBIIMVA, ITLBIASID, TLBIMVA, ICIALLU, TLBIVAL, TLBIMVAAL.

0b0 This field has no effect on the operation of the specified instructions.

0b1 When one of the specified instruction is executed at Non-secure EL1, the instruction is broadcast within the Inner Shareable shareability domain.

The reset behavior of this field is:
- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**VA, bit [8]**

Virtual SError interrupt exception.

0b0 This mechanism is not making a virtual SError interrupt pending.

0b1 A virtual SError interrupt is pending because of this mechanism.

The virtual SError interrupt is enabled only when the value of HCR.\{TGE, AMO\} is \{0, 1\}.

The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.

The reset behavior of this field is:
- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**VI, bit [7]**

Virtual IRQ exception.

0b0 This mechanism is not making a virtual IRQ pending.

0b1 A virtual IRQ is pending because of this mechanism.

The virtual IRQ is enabled only when the value of HCR.\{TGE, IMO\} is \{0, 1\}.

The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.

The reset behavior of this field is:
- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**VF, bit [6]**

Virtual FIQ exception.

0b0 This mechanism is not making a virtual FIQ pending.

0b1 A virtual FIQ is pending because of this mechanism.

The virtual FIQ is enabled only when the value of HCR.\{TGE, FMO\} is \{0, 1\}.

The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.
The reset behavior of this field is:
- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**AMO, bit [5]**

SError interrupt Mask Override. When this bit is set to 1, it overrides the effect of PSTATE.A, and enables virtual exception signaling by the VA bit.
If the value of HCR.TGE is 0, then virtual SError interrupts are enabled in Non-secure state.
If the value of HCR.TGE is 1, then in Non-secure state the HCR.AMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.
The reset behavior of this field is:
- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**IMO, bit [4]**

IRQ Mask Override. When this bit is set to 1, it overrides the effect of PSTATE.I, and enables virtual exception signaling by the VI bit.
If the value of HCR.TGE is 0, then Virtual IRQ interrupts are enabled in the Non-secure state.
If the value of HCR.TGE is 1, then in Non-secure state the HCR.IMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.
The reset behavior of this field is:
- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**FMO, bit [3]**

FIQ Mask Override. When this bit is set to 1, it overrides the effect of PSTATE.F, and enables virtual exception signaling by the VF bit.
If the value of HCR.TGE is 0, then Virtual FIQ interrupts are enabled in the Non-secure state.
If the value of HCR.TGE is 1, then in Non-secure state the HCR.FMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.
The reset behavior of this field is:
- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**PTW, bit [2]**

Protected Table Walk. In the Non-secure PL1&0 translation regime, a translation table access made as part of a stage 1 translation table walk is subject to a stage 2 translation. The combining of the memory type attributes from the two stages of translation means the access might be made to a type of Device memory. If this occurs then the value of this bit determines the behavior:

0b0  The translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be made speculatively.

0b1  The memory access generates a stage 2 Permission fault.
This bit is permitted to be cached in a TLB.
The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**SWIO, bit [1]**

Set/Way Invalidation Override. Causes Non-secure EL1 execution of the data cache invalidate by set/way instructions to perform a data cache clean and invalidate by set/way.

- **0b0** This control has no effect on the operation of data cache invalidate by set/way instructions.
- **0b1** Data cache invalidate by set/way instructions perform a data cache clean and invalidate by set/way.

When this bit is set to 1, DCISW performs the same invalidation as a DCCISW instruction.

As a result of changes to the behavior of DCISW, this bit is redundant in Armv8. This bit can be implemented as RES1.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**VM, bit [0]**

Virtualization enable. Enables stage 2 address translation for the Non-secure EL1&0 translation regime.

- **0b0** Non-secure EL1&0 stage 2 address translation disabled.
- **0b1** Non-secure EL1&0 stage 2 address translation enabled.

If the HCR.DC bit is set to 1, then the behavior of the PE when executing in a Non-secure mode other than Hyp mode is consistent with HCR.VM being 1, regardless of the actual value of HCR.VM, other than the value returned by an explicit read of HCR.VM.

When the value of this bit is 1, data cache invalidate instructions executed at Non-secure EL1 perform a data cache clean and invalidate. For the invalidate by set/way instruction this behavior applies regardless of the value of the HCR.SWIO bit.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

### Accessing HCR

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>}\{<coproc>, \{#{opc1}\}, <Rt>, <CRn>, <CRm>\}, \{#{opc2}\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
else if PSTATE.EL == EL1 then
if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
    AArch32.TakeHypTrapException(0x03);
else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    R[t] = HCR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        R[t] = HCR;
else
    MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
    else
    PSTATE.EL == EL2 then
        HCR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HCR = R[t];

## MCR{<c>}{<q>} {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>
**HCR2, Hyp Configuration Register 2**

The HCR2 characteristics are:

**Purpose**

Provides additional configuration controls for virtualization.

**Configurations**

AArch32 System register HCR2 bits [31:0] are architecturally mapped to AArch64 System register HCR_EL2[63:32].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HCR2 are **UNDEFINED**.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HCR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0</td>
</tr>
<tr>
<td>29</td>
<td>TTLBIS</td>
</tr>
<tr>
<td>28</td>
<td>TID4</td>
</tr>
<tr>
<td>27</td>
<td>TOCU</td>
</tr>
<tr>
<td>26</td>
<td>TICAB</td>
</tr>
<tr>
<td>25</td>
<td>MIOCNCE</td>
</tr>
<tr>
<td>24</td>
<td>TERR</td>
</tr>
<tr>
<td>23</td>
<td>TEA</td>
</tr>
</tbody>
</table>

**Bits [31:23]**

Reserved, RES0.

**TTLBIS, bit [22]**

*When FEAT_EVT is implemented:*

Trap TLB maintenance instructions that operate on the Inner Shareable domain. Traps execution of the following TLB maintenance instructions at EL1 to EL2:

- **TLBIALLIS, TLBIMVAIS, TLBIASIDIS, TLBIMVAAIS, TLBIMVALIS, TLBIMVAALIS**

  | 0b0 | This control does not cause any instructions to be trapped. |
  | 0b1 | Non-secure EL1 execution of the specified TLB maintenance instructions is trapped to EL2. |

When **FEAT_VHE** and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**

Reserved, RES0.

**Bit [21]**

Reserved, RES0.
TOCU, bit [20]

When FEAT_EVT is implemented:

Trap cache maintenance instructions that operate to the Point of Unification. Traps execution of those cache maintenance instructions at EL1 or EL0 using AArch64, and at EL1 using AArch32, to EL2.

This applies to the following instructions:

- When Non-secure EL0 is using AArch64, IC IVAU, DC CVAU. However, if the value of SCTLR_EL1.UCI is 0 these instructions are UNDEFINED at EL0 and any resulting exception is higher priority than this trap to EL2.
- When EL1 is using AArch64, IC IVAU, IC IALLU, DC CVAU.
- When Non-secure EL1 is using AArch32, ICIMVAU, ICIALLU, DCCMVAU.

--- Note

An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:

- IC IALLUIS and IC IALLU are always UNDEFINED at EL0 using AArch64.
- ICIMVAU, ICIALLU, ICIALLUIS, and DCCMVAU are always UNDEFINED at EL0 using AArch32.

0b0  This control does not cause any instructions to be trapped.
0b1  Non-secure execution of the specified cache maintenance instructions is trapped to EL2.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [19]

Reserved, RES0.

TICAB, bit [18]

When FEAT_EVT is implemented:

Trap ICIALLUIS cache maintenance instructions. Traps execution of those cache maintenance instructions at EL1 to EL2.

This applies to the following instructions:

ICIALLUIS.

0b0  This control does not cause any instructions to be trapped.
0b1  Non-secure EL1 execution of the specified cache maintenance instructions is trapped to EL2.
If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.

When FEAT_VHE and the value of HCR_EL2.{E2H, TGE} is \{1, 1\}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

TID4, bit [17]

*When FEAT_EVT is implemented:*

Trap ID group 4. Traps the following register accesses to EL2:

- EL1 reads of CCSIDR, CCSIDR2, CLIDR, and CSSELR.
- EL1 writes to CSSELR.

0b0 This control does not cause any instructions to be trapped.

0b1 The specified Non-secure EL1 and EL0 accesses to ID group 4 registers are trapped to EL2.

When FEAT_VHE is implemented and the value of HCR_EL2.{E2H, TGE} is \{1, 1\}, this field behaves as 0 for all purposes other than a direct read of the value of this bit.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

Bits [16:7]

Reserved, RES0.

MIOCNCE, bit [6]

Mismatched Inner/Outer Cacheable Non-Coherency Enable, for the Non-secure PL1&0 translation regime.

0b0 For the Non-secure PL1&0 translation regime, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there must be no loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

0b1 For the Non-secure PL1&0 translation regime, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there might be a loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

For more information, see *Mismatched memory attributes on page E2-7208.*

This field can be implemented as RAZ/WI.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to an architecturally UNKNOWN value.
TEA, bit [5]

When FEAT_RAS is implemented:

Route synchronous External abort exceptions from EL0 and EL1 to EL2.

0b0  Does not route synchronous External abort exceptions from Non-secure EL0 and EL1 to EL2.

0b1  Route synchronous External abort exceptions from Non-secure EL0 and EL1 to EL2, if not routed to EL3.

The reset behavior of this field is:

• On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TERR, bit [4]

When FEAT_RAS is implemented:

Trap Error record accesses from EL1 to EL2. Trap accesses to the following registers from EL1 to EL2:

ERRIDR, ERRSELR, ERXADDR, ERXADDR2, ERXCTLR, ERXCTLR2, ERXFR, ERXFR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, and ERXSTATUS.

When FEAT_RASv1p1 is implemented, ERXMISC4, ERXMISC5, ERXMISC6, and ERXMISC7.

0b0  This control does not cause any instructions to be trapped.

0b1  Accesses to the specified registers from EL1 generate a Trap exception to EL2.

The reset behavior of this field is:

• On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [3:2]

Reserved, RES0.

ID, bit [1]

Stage 2 Instruction access cacheability disable. For the Non-secure PL1&0 translation regime, when HCR.VM==1, this control forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

0b0  This control has no effect on stage 2 of the Non-secure PL1&0 translation regime.

0b1  For the Non-secure PL1&0 translation regime, forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

This bit has no effect on the EL2 translation regime.

The reset behavior of this field is:

• On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.
CD, bit [0]

Stage 2 Data access cacheability disable. When HCR.VM==1, this forces all stage 2 translations for
data accesses and translation table walks to Normal memory to be Non-cacheable for the
Non-secure PL1&0 translation regime.

0b0  This control has no effect on stage 2 of the Non-secure PL1&0 translation regime for
data accesses and translation table walks.

0b1  For the Non-secure PL1&0 translation regime, forces all stage 2 translations for data
accesses and translation table walks to Normal memory to be Non-cacheable.

This bit has no effect on the EL2 translation regime.

The reset behavior of this field is:

•  On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing HCR2

Accesses to this register use the following encodings in the System register encoding space:

**MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}**

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  R[t] = HCR2;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    R[t] = HCR2;

**MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}**

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
  ...
UNDEFINED;
elsif PSTATE.EL == EL2 then
    HCR2 = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HCR2 = R[t];
end if;
**G8.2.66 HDFAR, Hyp Data Fault Address Register**

The HDFAR characteristics are:

**Purpose**

Holds the virtual address of the faulting address that caused a synchronous Data Abort exception that is taken to Hyp mode.

**Configurations**

AArch32 System register HDFAR bits [31:0] are architecturally mapped to AArch64 System register FAR_EL2[31:0].

AArch32 System register HDFAR bits [31:0] are architecturally mapped to AArch32 System register DFAR[31:0] (DFAR_S) when EL2 is implemented, EL3 is implemented and the implementation only supports execution in AArch32 state.

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HDFAR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HDFAR is a 32-bit register.

**Field descriptions**

![Field description diagram](image)

**Bits [31:0]**

VA of faulting address of synchronous Data Abort exception taken to Hyp mode.

On a Prefetch Abort exception, this register is UNKNOWN.

Any execution in a Non-secure EL1 or Non-secure EL0 mode makes this register UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing HDFAR**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} \text{<coproc>, } \{<opc1>, \text{<Rt>, <CRn>, <CRm>, } \{<opc2>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
    end if
else
    UNDEFINED;
end if
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & ELUsingAArch32(EL2) & HSTR_EL2.T6 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T6 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  HDFAR = R[t];
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    HDFAR = R[t];

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

UNDEFINED;
elif PSTATE.EL == EL2 then
  R[t] = HDFAR;
elif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    R[t] = HDFAR;
G8.2.67 HIFAR, Hyp Instruction Fault Address Register

The HIFAR characteristics are:

**Purpose**

Holds the virtual address of the faulting address that caused a synchronous Prefetch Abort exception that is taken to Hyp mode.

**Configurations**

AArch32 System register HIFAR bits [31:0] are architecturally mapped to AArch64 System register FAR_EL2[63:32].

AArch32 System register HIFAR bits [31:0] are architecturally mapped to AArch32 System register IFAR[31:0] (IFAR_S) when EL2 is implemented, EL3 is implemented and the implementation only supports execution in AArch32 state.

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HIFAR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HIFAR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>VA of faulting address of synchronous Prefetch Abort exception taken to Hyp mode</td>
</tr>
</tbody>
</table>

**Accessing HIFAR**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC\{<c>\}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2}>\
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else

UNDEFINED;
elseif PSTATE_EL == EL2 then
    R[t] = HIFAR;
elseif PSTATE_EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        R[t] = HIFAR;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elseif PSTATE_EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elseif PSTATE_EL == EL2 then
    HIFAR = R[t];
elseif PSTATE_EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HIFAR = R[t];
G8.2.68  HMAIR0, Hyp Memory Attribute Indirection Register 0

The HMAIR0 characteristics are:

**Purpose**

Along with HMAIR1, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations for memory accesses from Hyp mode.

AttrIndx[2] indicates the HMAIR register to be used:

- When AttrIndx[2] is 0, HMAIR0 is used.
- When AttrIndx[2] is 1, HMAIR1 is used.

**Configurations**

AArch32 System register HMAIR0 bits [31:0] are architecturally mapped to AArch64 System register MAIR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HMAIR0 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HMAIR0 is a 32-bit register.

**Field descriptions**

*When TTBCR.EAE == 1:*

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attr3</td>
<td>Attr2</td>
<td>Attr1</td>
<td>Attr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Attr<n>, bits [8n+7:8n], for n = 3 to 0**

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIndx[2:0] gives the value of <n> in Attr<n>.
- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

<table>
<thead>
<tr>
<th>Attr&lt;n&gt;[7:4]</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>Device memory. See encoding of Attr&lt;n&gt;[3:0] for the type of Device memory.</td>
</tr>
<tr>
<td>0b00RW, RW not 0b00</td>
<td>Normal memory, Outer Write-Through Transient.</td>
</tr>
<tr>
<td>0b0100</td>
<td>Normal memory, Outer Non-cacheable.</td>
</tr>
<tr>
<td>0b01RW, RW not 0b00</td>
<td>Normal memory, Outer Write-Back Transient.</td>
</tr>
<tr>
<td>0b10RW</td>
<td>Normal memory, Outer Write-Through Non-transient.</td>
</tr>
<tr>
<td>0b11RW</td>
<td>Normal memory, Outer Write-Back Non-transient.</td>
</tr>
</tbody>
</table>

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.
The meaning of bits \([3:0]\) depends on the value of bits \([7:4]\):

<table>
<thead>
<tr>
<th>Attr(&lt;n&gt;[3:0]</th>
<th>Meaning when Attr(&lt;n&gt;[7:4] is 0b0000</th>
<th>Meaning when Attr(&lt;n&gt;[7:4] is not 0b0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>Device-nGnRnE memory</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>0b00RW, RW not 0b00</td>
<td>UNPREDICTABLE</td>
<td>Normal memory, Inner Write-Through Transient</td>
</tr>
<tr>
<td>0b0100</td>
<td>Device-nGnRE memory</td>
<td>Normal memory, Inner Non-cacheable</td>
</tr>
<tr>
<td>0b01RW, RW not 0b00</td>
<td>UNPREDICTABLE</td>
<td>Normal memory, Inner Write-Back Transient</td>
</tr>
<tr>
<td>0b1000</td>
<td>Device-nGRE memory</td>
<td>Normal memory, Inner Write-Through Non-transient (RW=0b00)</td>
</tr>
<tr>
<td>0b10RW, RW not 0b00</td>
<td>UNPREDICTABLE</td>
<td>Normal memory, Inner Write-Through Non-transient</td>
</tr>
<tr>
<td>0b1100</td>
<td>Device-GRE memory</td>
<td>Normal memory, Inner Write-Back Non-transient (RW=0b00)</td>
</tr>
<tr>
<td>0b11RW, RW not 0b00</td>
<td>UNPREDICTABLE</td>
<td>Normal memory, Inner Write-Back Non-transient</td>
</tr>
</tbody>
</table>

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in some Attr\(<n\>) fields have the following meanings:

<table>
<thead>
<tr>
<th>R or W</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>No Allocate</td>
</tr>
<tr>
<td>0b1</td>
<td>Allocate</td>
</tr>
</tbody>
</table>

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNDEFINED value.

**Accessing HMAIR0**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC\{<c>\}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>, {#}<opc2>\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else

UNDEFINED;
elsif PSTATE_EL == EL2 then
    R[t] = HMAIR0;
elsif PSTATE_EL == EL3 then
    if SCR_NS == '0' then
        UNDEFINED;
    else
        R[t] = HMAIR0;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE_EL == EL2 then
    HMAIR0 = R[t];
elsif PSTATE_EL == EL3 then
    if SCR_NS == '0' then
        UNDEFINED;
    else
        HMAIR0 = R[t];
HMAIR1, Hyp Memory Attribute Indirection Register 1

The HMAIR1 characteristics are:

**Purpose**

Along with HMAIR0, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations for memory accesses from Hyp mode.

AttrIndx[2] indicates the HMAIR register to be used:
- When AttrIndx[2] is 0, HMAIR0 is used.
- When AttrIndx[2] is 1, HMAIR1 is used.

**Configurations**

AArch32 System register HMAIR1 bits [31:0] are architecturally mapped to AArch64 System register MAIR_EL2[63:32].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HMAIR1 are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HMAIR1 is a 32-bit register.

**Field descriptions**

*When TTBCR.EAE == 1:*

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>24</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attr7</td>
<td>Attr6</td>
<td>Attr5</td>
<td>Attr4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Attr<n>, bits [8(n-4)+7:8(n-4)], for n = 7 to 4

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:
- AttrIndx[2:0] gives the value of <n> in Attr<n>.
- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

<table>
<thead>
<tr>
<th>Attr&lt;n&gt;[7:4]</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>Device memory. See encoding of Attr&lt;n&gt;[3:0] for the type of Device memory.</td>
</tr>
<tr>
<td>0b00RW, RW not 0b00</td>
<td>Normal memory, Outer Write-Through Transient.</td>
</tr>
<tr>
<td>0b0100</td>
<td>Normal memory, Outer Non-cacheable.</td>
</tr>
<tr>
<td>0b01RW, RW not 0b00</td>
<td>Normal memory, Outer Write-Back Transient.</td>
</tr>
<tr>
<td>0b10RW</td>
<td>Normal memory, Outer Write-Through Non-transient.</td>
</tr>
<tr>
<td>0b11RW</td>
<td>Normal memory, Outer Write-Back Non-transient.</td>
</tr>
</tbody>
</table>

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.
The meaning of bits [3:0] depends on the value of bits [7:4]:

<table>
<thead>
<tr>
<th>Attr&lt;n&gt;[3:0]</th>
<th>Meaning when Attr&lt;n&gt;[7:4] is 0b0000</th>
<th>Meaning when Attr&lt;n&gt;[7:4] is not 0b0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>Device-nGnRE memory</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>0b00RW, RW not 0b00</td>
<td>UNPREDICTABLE</td>
<td>Normal memory, Inner Write-Through Transient</td>
</tr>
<tr>
<td>0b0100</td>
<td>Device-nGnRE memory</td>
<td>Normal memory, Inner Non-cacheable</td>
</tr>
<tr>
<td>0b01RW, RW not 0b00</td>
<td>UNPREDICTABLE</td>
<td>Normal memory, Inner Write-Back Transient</td>
</tr>
<tr>
<td>0b1000</td>
<td>Device-nGRE memory</td>
<td>Normal memory, Inner Write-Through Non-transient (RW=0b00)</td>
</tr>
<tr>
<td>0b10RW, RW not 0b00</td>
<td>UNPREDICTABLE</td>
<td>Normal memory, Inner Write-Through Non-transient</td>
</tr>
<tr>
<td>0b1100</td>
<td>Device-GRE memory</td>
<td>Normal memory, Inner Write-Back Non-transient (RW=0b00)</td>
</tr>
<tr>
<td>0b11RW, RW not 0b00</td>
<td>UNPREDICTABLE</td>
<td>Normal memory, Inner Write-Back Non-transient</td>
</tr>
</tbody>
</table>

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in some Attr<n> fields have the following meanings:

<table>
<thead>
<tr>
<th>R or W</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>No Allocate</td>
</tr>
<tr>
<td>0b1</td>
<td>Allocate</td>
</tr>
</tbody>
</table>

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing HMAIR1**

Accesses to this register use the following encodings in the System register encoding space:

```
MRC<coproc>, <CRn>, <CRm), <opc1>, <opc2>, <c>, <q>
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HMAIR1 = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HMAIR1 = R[t];

**MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>]**

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL2 then
    R[t] = HMAIR1;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        R[t] = HMAIR1;
G8.2.70 HPFAR, Hyp IPA Fault Address Register

The HPFAR characteristics are:

**Purpose**

Holds the faulting IPA for some aborts on a stage 2 translation taken to Hyp mode.

**Configurations**

AArch32 System register HPFAR bits [31:0] are architecturally mapped to AArch64 System register HPFAR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HPFAR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HPFAR is a 32-bit register.

**Field descriptions**

![Field descriptions](image)

Execution in any Non-secure mode other than Hyp mode makes this register UNKNOWN.

**FIPA[39:12], bits [31:4]**

Bits [39:12] of the faulting intermediate physical address.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [3:0]**

Reserved, RES0.

**Accessing HPFAR**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC\{<c><q}\} \{<coproc>, \{#<opc1>, \{<Rt>, \{<CRn>, \{<CRm>, \{#<opc2>\}\}\}\}\}\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
    endif;
elsif PSTATE.EL == EL2 then
    R[t] = HPFAR;
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then

UNDEFINED;
else
    R[t] = HPFAR;

\textit{MCR\{<c>\}{<q>} <coproc>, \{#\}<opc1>, <Rt>, <CRn>, <CRm>\{, \{#\}<opc2>\}}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HPFAR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HPFAR = R[t];
G8.2.71 HRMR, Hyp Reset Management Register

The HRMR characteristics are:

Purpose

If EL2 is the highest implemented Exception level and this register is implemented:
- A write to the register at EL2 can request a Warm reset.
- If EL2 can use AArch32 and AArch64, this register specifies the Execution state that the PE boots into on a Warm reset.

Configurations

AArch32 System register HRMR bits [31:0] are architecturally mapped to AArch64 System register RMR_EL2[31:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to HRMR are UNDEFINED.

Only implemented if EL2 is the highest implemented Exception level. In this case:
- If EL2 can use AArch32 and AArch64 then this register must be implemented.
- If EL2 cannot use AArch64 then it is IMPLEMENTATION DEFINED whether the register is implemented.

Attributes

HRMR is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>RR</td>
<td>AA64</td>
<td></td>
</tr>
</tbody>
</table>

Bits [31:2]

Reserved, RES0.

RR, bit [1]

Reset Request. Setting this bit to 1 requests a Warm reset.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

AA64, bit [0]

When EL2 can use AArch64, determines which Execution state the PE boots into after a Warm reset:
- 0b0 AArch32.
- 0b1 AArch64.

On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.

If EL2 cannot use AArch64 this bit is RAZ/WI.

The reset behavior of this field is:
- When implemented as a RW field, this field resets to 0 on a Cold reset.

Accessing HRMR

Accesses to this register use the following encodings in the System register encoding space:
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif PSTATE.EL == EL2 && IsHighestEL(EL2) then
  R[t] = HRMR;
else
  UNDEFINED;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && IsHighestEL(EL2) && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif PSTATE.EL == EL2 && IsHighestEL(EL2) then
  HRMR = R[t];
else
  UNDEFINED;
G8.2.72 HSCTLR, Hyp System Control Register

The HSCTLR characteristics are:

**Purpose**

Provides top level control of the system operation in Hyp mode.

**Configurations**

AArch32 System register HSCTLR bits [31:0] are architecturally mapped to AArch64 System register SCTLR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HSCTLR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HSCTLR is a 32-bit register.

**Field descriptions**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| TE | RES1 | RES0 | EE | RES1 | RES0 | RES0 | I  | RES0 | RES0 | RES0 | RES1 | RES1 | RES1 | RES1 | RES1 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 | RES0 |

**DSSBS, bit [31]**

When FEAT_SSSBS is implemented:

Default PSTATE.SSBS value on Exception Entry. The defined values are:

- 0b0 PSTATE.SSBS is set to 0 on an exception to Hyp mode.
- 0b1 PSTATE.SSBS is set to 1 on an exception to Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

**Otherwise:**

Reserved, RES0.

**TE, bit [30]**

T32 Exception Enable. This bit controls whether exceptions to EL2 are taken to A32 or T32 state:

- 0b0 Exceptions, including reset, taken to A32 state.
- 0b1 Exceptions, including reset, taken to T32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

**Bits [29:28]**

Reserved, RES1.

**Bits [27:26]**

Reserved, RES0.
The value of the PSTATE.E bit on entry to Hyp mode, the endianness of stage 1 translation table walks in the EL2 translation regime, and the endianness of stage 2 translation table walks in the PL1&0 translation regime.

- **0b0**: Little-endian. PSTATE.E is cleared to 0 on entry to Hyp mode. Stage 1 translation table walks in the EL2 translation regime, and stage 2 translation table walks in the PL1&0 translation regime are little-endian.
- **0b1**: Big-endian. PSTATE.E is set to 1 on entry to Hyp mode. Stage 1 translation table walks in the EL2 translation regime, and stage 2 translation table walks in the PL1&0 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception levels higher than EL0, this bit is RES1.

The reset behavior of this field is:
- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

**Bit [24]**
Reserved, RES0.

**Bits [23:22]**
Reserved, RES1.

**Bits [21:20]**
Reserved, RES0.

**WXN, bit [19]**
Write permission implies XN (Execute-never). For the EL2 translation regime, this bit can force all memory regions that are writable to be treated as XN.

- **0b0**: This control has no effect on memory access permissions.
- **0b1**: Any region that is writable in the EL2 translation regime is forced to XN for accesses from software executing at EL2.

This bit applies only when HSCTLR.M bit is set.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

**Bit [18]**
Reserved, RES1.

**Bit [17]**
Reserved, RES0.

**Bit [16]**
Reserved, RES1.

**Bits [15:13]**
Reserved, RES0.

**I, bit [12]**
Instruction access Cacheability control, for accesses at EL2:

- **0b0**: All instruction access to Normal memory from EL2 are Non-cacheable for all levels of instruction and unified cache.
If the value of HSCTLR.M is 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

0b1
All instruction access to Normal memory from EL2 can be cached at all levels of instruction and unified cache.

If the value of HSCTLR.M is 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the PL1&0 translation regime.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

Bit [11]
Reserved, RES1.

Bits [10:9]
Reserved, RES0.

SED, bit [8]

SETEND instruction disable. Disables SETEND instructions at EL2.

0b0
SETEND instruction execution is enabled at EL2.

0b1
SETEND instructions are UNDEFINED at EL2.

If the implementation does not support mixed-endian operation at EL2, this bit is RES1.

The reset behavior of this field is:

• On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

ITD, bit [7]

IT Disable. Disables some uses of IT instructions at EL2.

0b0
All IT instruction functionality is enabled at EL2.

0b1
Any attempt at EL2 to execute any of the following is UNDEFINED:

• All encodings of the IT instruction with hw1[3:0]=1000.
• All encodings of the subsequent instruction with the following values for hw1:
  — 11xxxxxxxxxxxx: All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.
  — 1011xxxxxxxxxxxx: All instructions in Miscellaneous 16-bit instructions on page F3-7303.
  — 10100xxxxxxxxxxx: ADD Rd, PC, #imm
  — 01001xxxxxxxxxxx: LDR Rd, [PC, #imm]
  — 0100x1xx1111xxx: ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.
  — 01001xx1xxx1111: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers unpredictable cases with BLX Rn.

These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block. It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:

• A 16-bit instruction, that can only be followed by another 16-bit instruction.
• The first half of a 32-bit instruction.

This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.
An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If an instruction in an active IT block that would be disabled by this field sets this field to 1 then behavior is CONSTRAINED UNPREDICTABLE. For more information, see Changes to an ITD control by an instruction in an IT block on page E1-7138.

ITD is optional, but if it is implemented in the HSCTLR then it must also be implemented in the SCTLR_EL1, SCTLR_EL2, and SCTLR.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement ITD, access to this field is RAZ/WI.

**Bit [6]**

Reserved, RES0.

**CP15BEN, bit [5]**

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==0b1111) encoding space from EL2:

- 0b0  EL2 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED.
- 0b1  EL2 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the HSCTLR then it must also be implemented in the SCTLR_EL1, SCTLR_EL2, and SCTLR.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement CP15BEN, access to this field is RAO/WI.

**LSMAOE, bit [4]**

*When FEAT_LSMAOC is implemented:*

Load Multiple and Store Multiple Atomicity and Ordering Enable.

- 0b0  For all memory accesses at EL2, A32 and T32 Load Multiple and Store Multiple can have an interrupt taken during the sequence memory accesses, and the memory accesses are not required to be ordered.
- 0b1  The ordering and interrupt behavior of A32 and T32 Load Multiple and Store Multiple at EL2 is as defined for Armv8.0.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2, this field resets to 1.

*Otherwise:*

Reserved, RES1.

**nTLSMD, bit [3]**

*When FEAT_LSMAOC is implemented:*

No Trap Load Multiple and Store Multiple to Device-nGRE/Device-nGnRE/Device-nGnRnE memory.

- 0b0  All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL2 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are trapped and generate a stage 1 Alignment fault.
- 0b1  All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL2 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are not trapped.
This bit is permitted to be cached in a TLB.
The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 1.

**Otherwise:**
Reserved, RES1.

**C, bit [2]**

Cacheability control, for data accesses at EL2:
- **0b0** All data access to Normal memory from EL2, and all accesses to the EL2 translation tables, are Non-cacheable for all levels of data and unified cache.
- **0b1** All data access to Normal memory from EL2, and all accesses to the EL2 translation tables, can be cached at all levels of data and unified cache.

This bit has no effect on the PL1&0 translation regime.
The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**A, bit [1]**

Alignment check enable. This is the enable bit for Alignment fault checking at EL2:
- **0b0** Alignment fault checking disabled when executing at EL2.
  Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element or data elements being accessed.
- **0b1** Alignment fault checking enabled when executing at EL2.
  All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element or data elements being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.
The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to an architecturally UNKNOWN value.

**M, bit [0]**

MMU enable for EL2 stage 1 address translation. Possible values of this bit are:
- **0b0** EL2 stage 1 address translation disabled.
  See the HSCTL.R1 field for the behavior of instruction accesses to Normal memory.
- **0b1** EL2 stage 1 address translation enabled.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2, this field resets to 0.

**Accessing HSCTL.R**

Accesses to this register use the following encodings in the System register encoding space:
```plaintext
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  R[t] = HSCTRL;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    HSCTRL = R[t];
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  HSCTRL = R[t];
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    HSCTRL = R[t];
```

G8.2.73  HSR, Hyp Syndrome Register

The HSR characteristics are:

**Purpose**

Holds syndrome information for an exception taken to Hyp mode.

**Configurations**

AArch32 System register HSR bits [31:0] are architecturally mapped to AArch64 System register ESR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HSR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HSR is a 32-bit register.

**Field descriptions**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EC</td>
<td>IL</td>
<td>ISS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution in any Non-secure PE mode other than Hyp mode makes this register UNKNOWN.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL2, the value of HSR is UNKNOWN. The value written to HSR must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

**EC, bits [31:26]**

Exception Class. Indicates the reason for the exception that this register holds information about. Possible values of this field are:

**EC == 0b000000**

Unknown reason.

See ISS encoding for exceptions with an unknown reason.

**EC == 0b000001**

Trapped WFI or WFE instruction execution.

Conditional WFE and WFI instructions that fail their condition code check do not cause an exception.

See ISS encoding for exception from a WFI or WFE instruction.

**EC == 0b000011**

Trapped MCR or MRC access with (coproc==0b1111) that is not reported using EC 0b000000.

See ISS encoding for exception from an MCR or MRC access.

**EC == 0b000100**

Trapped MCRR or MRRC access with (coproc==0b1110).

See ISS encoding for exception from an MCRR or MRRC access.

**EC == 0b000101**

Trapped MCR or MRC access with (coproc==0b1110).

See ISS encoding for exception from an MCR or MRC access.
EC == 0b000110
Trapped LDC or STC access.
The only architected uses of these instructions are:
• An STC to write data to memory from DBGDTRRXint.
• An LDC to read data from memory to DBGDTRTXint.
See ISS encoding for exception from an LDC or STC instruction.

EC == 0b000111
Access to Advanced SIMD or floating-point functionality trapped by a HCPTR. {TASE, TCP10} control.
Excludes exceptions generated because Advanced SIMD and floating-point are not implemented. These are reported with EC value 0b000000.
See ISS encoding for exception from an access to SIMD or floating-point functionality, resulting from HCPTR.

EC == 0b001000
Trapped VMRS access, from ID group trap, that is not reported using EC 0b000111.
See ISS encoding for exception from an MCR or MRC access.

EC == 0b001100
Trapped MRRC access with (coproc==0b1110).
See ISS encoding for exception from an MCRR or MRRC access.

EC == 0b001110
Illegal exception return to AArch32 state.
See ISS encoding for exception from an Illegal state or PC alignment fault.

EC == 0b010001
Exception on SVC instruction execution in AArch32 state routed to EL2.
See ISS encoding for exception from HVC or SVC instruction execution.

EC == 0b010010
HVC instruction execution in AArch32 state, when HVC is not disabled.
See ISS encoding for exception from HVC or SVC instruction execution.

EC == 0b010011
Trapped execution of SMC instruction in AArch32 state.
See ISS encoding for exception from SMC instruction execution.

EC == 0b100000
Prefetch Abort from a lower Exception level.
See ISS encoding for exception from a Prefetch Abort.

EC == 0b100001
Prefetch Abort taken without a change in Exception level.
See ISS encoding for exception from a Prefetch Abort.

EC == 0b100010
PC alignment fault exception.
See ISS encoding for exception from an Illegal state or PC alignment fault.

EC == 0b100100
Data Abort exception from a lower Exception level.
See ISS encoding for exception from a Data Abort.

EC == 0b100101
Data Abort exception taken without a change in Exception level.
See ISS encoding for exception from a Data Abort.

All other EC values are reserved by Arm, and:
• Unused values in the range 0b000000 - 0b101100 (0x00 - 0x2C) are reserved for future use for synchronous exceptions.
• Unused values in the range 0b101101 - 0b111111 (0x2D - 0x3F) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [25]

Instruction length bit. Indicates the size of the instruction that has been trapped to Hyp mode. When this bit is valid, possible values of this bit are:

0b0 16-bit instruction trapped.
0b1 32-bit instruction trapped.

This field is RES1 and not valid for the following cases:
• When the EC field is 0b000000, indicating an exception with an unknown reason.
• Prefetch Aborts.
• Data Abort exceptions for which the HSR.ISS.ISV field is 0.
• When the EC value is 0b01110, indicating an Illegal state exception.

The IL field is not valid and is UNKNOWN on an exception from a PC alignment fault.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS, bits [24:0]

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

ISS encoding for exceptions with an unknown reason

<table>
<thead>
<tr>
<th>24</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td></td>
</tr>
</tbody>
</table>

Bits [24:0]

Reserved, RES0.

This EC code is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

• The attempted execution of an instruction bit pattern that has no allocated instruction or is not accessible in the current PE mode in the current Security state, including:
  — A read access using a System register encoding pattern that is not allocated for reads or that does not permit reads in the current PE mode and Security state.
  — A write access using a System register encoding pattern that is not allocated for writes or that does not permit writes in the current PE mode and Security state.
  — Instruction encodings that are unallocated.
  — Instruction encodings for instructions not implemented in the implementation.

• In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.

• In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.

• The attempted execution of a short vector floating-point instruction.
• In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.

• An exception generated because of the value of one of the SCTLR.{ITD, SED, CP15BEN} control bits.

• Attempted execution of:
  — An HVC instruction when disabled by HCR.HCD, SCR.HCE, or SCR_EL3.HCE.
  — An SMC instruction when disabled by SCR.SCD or SCR_EL3.SMD.
  — An HLT instruction when disabled by EDSCR.HDE.

• An HVC instruction when disabled by HCR.HCD, SCR.HCE, or SCR_EL3.HCE. An SMC instruction when disabled by SCR.SCD or SCR_EL3.SMD. An HLT instruction when disabled by EDSCR.HDE.

• An exception generated because of the attempted execution of an MSR (Banked register) or MRS (Banked register) instruction that would access a Banked register that is not accessible from the Security state and PE mode at which the instruction was executed.

--- Note ---

An exception is generated only if the CONSTRAINED UNPREDICTABLE behavior of the instruction is that it is UNDEFINED, see MSR (banked register) and MRS (banked register) on page K1-11582.

---

• Attempted execution, in Debug state, of:
  — A DCPS1 instruction in Non-secure state from EL0 when EL2 is using AArch32 and the value of HCR.TGE is 1.
  — A DCPS2 instruction at EL1 or EL0 when EL2 is not implemented, or when EL3 is using AArch32 and the value of SCR.NS is 0, or when EL3 is using AArch64 and the value of SCR_EL3.NS is 0.
  — A DCPS3 instruction when EL3 is not implemented, or when the value of EDSCR.SDD is 1.

• In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, or EL0 of an instruction that is configured to trap to EL3.

**Undefined Instruction exception, when the value of HCR.TGE is 1** on page G1-8945 describes the configuration settings for a trap that returns an HSR.EC value of 0b000000.

**ISS encoding for exception from a WFI or WFE instruction**

<table>
<thead>
<tr>
<th>CV</th>
<th>COND</th>
<th>RES0</th>
<th>TI</th>
</tr>
</thead>
</table>

**CV, bit [24]**

Condition code valid. Possible values of this bit are:

- 0b0: The COND field is not valid.
- 0b1: The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**COND, bits [23:20]**

The condition code for the trapped instruction.
When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to 0b1110.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to 0b1110, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:1]

Reserved, RES0.

TI, bit [0]

Trapped instruction. Possible values of this bit are:

- 0b0 WFI trapped.
- 0b1 WFE trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions on page G1-9020 describes the configuration settings for this trap.

**ISS encoding for exception from an MCR or MRC access**

<table>
<thead>
<tr>
<th>28</th>
<th>23</th>
<th>20</th>
<th>19</th>
<th>17</th>
<th>16</th>
<th>14</th>
<th>13</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>5</th>
<th>6</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV</td>
<td>COND</td>
<td>Opc2</td>
<td>Opc1</td>
<td>CRn</td>
<td>Rn</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td></td>
</tr>
<tr>
<td>RES0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CV, bit [24]

Condition code valid. Possible values of this bit are:

- 0b0 The COND field is not valid.
- 0b1 The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
COND, bits [23:20]
The condition code for the trapped instruction.
When an A32 instruction is trapped, CV is set to 1 and:
- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to 0b1110.
A conditional A32 instruction that is known to pass its condition code check can be presented either:
- With COND set to 0b1110, the value for unconditional.
- With the COND value held in the instruction.
When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc2, bits [19:17]
The Opc2 value from the issued instruction.
For a trapped VMRS access, holds the value 0b000.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc1, bits [16:14]
The Opc1 value from the issued instruction.
For a trapped VMRS access, holds the value 0b111.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]
The CRn value from the issued instruction.
For a trapped VMRS access, holds the reg field from the VMRS instruction encoding.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [9]
Reserved, RES0.

Rt, bits [8:5]
The Rt value from the issued instruction, the general-purpose register used for the transfer.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]
The CRm value from the issued instruction.
For a trapped VMRS access, holds the value 0b0000.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Direction, bit [0]**

Indicates the direction of the trapped instruction.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Write to System register space. MCR instruction.</td>
</tr>
<tr>
<td>0b1</td>
<td>Read from System register space. MRC or VMRS instruction.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

The following sections describe configuration settings for traps that are reported using EC value 0b000011:

- Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the ID registers on page G1-9018.
- Traps to Hyp mode of Non-secure EL0 and EL1 accesses to lockdown, DMA, and TCM operations on page G1-9016.
- Traps to Hyp mode of Non-secure EL1 execution of cache maintenance instructions on page G1-9015.
- Traps to Hyp mode of Non-secure EL1 execution of TLB maintenance instructions on page G1-9015.
- Traps to Hyp mode of Non-secure EL1 accesses to the Auxiliary Control Register on page G1-9016.
- Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Performance Monitors registers on page G1-9029.
- Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Activity Monitors registers on page G1-9021.
- Traps to Hyp mode of Non-secure EL1 accesses to the CPACR on page G1-9023.
- Traps to Hyp mode of Non-secure EL1 accesses to virtual memory control registers on page G1-9014.
- General trapping to Hyp mode of Non-secure EL0 and EL1 accesses to System registers in the (coproc==0b1111) encoding space on page G1-9024.

The following sections describe configuration settings for traps that are reported using EC value 0b000101:

- ID group 0, Primary device identification registers on page G1-9019.
- Traps to Hyp mode of Non-secure System register accesses to trace registers on page G1-9023.
- Trapping Non-secure System register accesses to Debug ROM registers on page G1-9026.
- Trapping Non-secure System register accesses to powerdown debug registers on page G1-9027.
- Trapping general Non-secure System register accesses to debug registers on page G1-9027.

The following sections describes configuration settings for traps that are reported using EC value 0b001000:

- ID group 0, Primary device identification registers on page G1-9019.
- ID group 3, Detailed feature identification registers on page G1-9020.

**ISS encoding for exception from an MCRR or MRRC access**

```
<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CV</td>
<td>COND</td>
<td>Opc1</td>
<td>RES0</td>
<td>Rt2</td>
<td>Rt</td>
<td>RES0</td>
<td>CRm</td>
<td>Direction</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
CV, bit [24]
Condition code valid. Possible values of this bit are:

- 0b0 The COND field is not valid.
- 0b1 The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.
When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]
The condition code for the trapped instruction.
When an A32 instruction is trapped, CV is set to 1 and:
- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to 0b1110.

A conditional A32 instruction that is known to pass its condition code check can be presented either:
- With COND set to 0b1110, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc1, bits [19:16]
The Opc1 value from the issued instruction.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:14]
Reserved, RES0.

Rt2, bits [13:10]
The Rt2 value from the issued instruction, the second general-purpose register used for the transfer.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [9]
Reserved, RES0.

Rt, bits [8:5]
The Rt value from the issued instruction, the first general-purpose register used for the transfer.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**CRm, bits [4:1]**

The CRm value from the issued instruction.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Direction, bit [0]**

Indicates the direction of the trapped instruction.

- **0b0**: Write to System register space. MCRR instruction.
- **0b1**: Read from System register space. MRRC instruction.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

The following sections describe configuration settings for traps that are reported using EC value **0b001000**:
- Traps to Hyp mode of Non-secure EL1 accesses to virtual memory control registers on page G1-9014.
- Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Performance Monitors registers on page G1-9029.
- Traps to Hyp mode of Non-secure EL0 and EL1 accesses to Activity Monitors registers on page G1-9021.
- Traps to Hyp mode of Non-secure EL0 and EL1 accesses to the Generic Timer registers on page G1-9028.
- General trapping to Hyp mode of Non-secure EL0 and EL1 accesses to System registers in the (coproc==0b1111) encoding space on page G1-9024.

The following sections describe configuration settings for traps that are reported using EC value **0b001100**:
- Traps to Hyp mode of Non-secure System register accesses to trace registers on page G1-9023.
- Trapping Non-secure System register accesses to Debug ROM registers on page G1-9026.

**ISS encoding for exception from an LDC or STC instruction**

```
   23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 CV  COND  imm8  RES0  Rn  AM  Offset  Direction
```

**CV, bit [24]**

Condition code valid. Possible values of this bit are:
- **0b0**: The COND field is not valid.
- **0b1**: The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.
When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**COND, bits [23:20]**

The condition code for the trapped instruction.
When an A32 instruction is trapped, CV is set to 1 and:
  • If the instruction is conditional, COND is set to the condition code field value from the
    instruction.
  • If the instruction is unconditional, COND is set to 0b1110.
A conditional A32 instruction that is known to pass its condition code check can be presented either:
  • With COND set to 0b1110, the value for unconditional.
  • With the COND value held in the instruction.
When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  • CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT
    field to determine the condition, if any, of the T32 instruction.
  • CV is set to 1 and COND is set to the condition code for the condition that applied to the
    instruction.
For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped
conditional instruction only if the instruction passes its condition code check, these definitions mean
that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110,
or to the value of any condition that applied to the instruction.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

imm8, bits [19:12]
The immediate value from the issued instruction.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:9]
Reserved, RES0.

Rn, bits [8:5]
The Rn value from the issued instruction. Valid only when AM[2] is 0, indicating an immediate
form of the LDC or STC instruction.
When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Offset, bit [4]
Indicates whether the offset is added or subtracted:
  0b0  Subtract offset.
  0b1  Add offset.
This bit corresponds to the U bit in the instruction encoding.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

AM, bits [3:1]
Addressing mode. The permitted values of this field are:
  0b000  Immediate unindexed.
  0b001  Immediate post-indexed.
  0b010  Immediate offset.
  0b011  Immediate pre-indexed.
  0b100  Literal unindexed.
LDC instruction in A32 instruction set only.
For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.

0b110  Literal offset.
LDC instruction only.
For a trapped STC instruction, this encoding is reserved.

The values 0b1101 and 0b1111 are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRANGED UNPREDICTABLE.

Bit [2] in this subfield indicates the instruction form, immediate or literal.
Bits [1:0] in this subfield correspond to the bits {P, W} in the instruction encoding.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]
Indicates the direction of the trapped instruction.
0b0  Write to memory. STC instruction.
0b1  Read from memory. LDC instruction.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Trapping general Non-secure System register accesses to debug registers on page G1-9027 describes the configuration settings for the trap that is reported using EC value 0b000110.

ISS encoding for exception from an access to SIMD or floating-point functionality, resulting from HCPTR

Excludes exceptions that occur because Advanced SIMD and floating-point functionality is not implemented, or because the value of HCR.TGE or HCR_EL2.TGE is 1. These are reported with EC value 0b000000.

CV, bit [24]
Condition code valid. Possible values of this bit are:
0b0  The COND field is not valid.
0b1  The COND field is valid.
When an A32 instruction is trapped, CV is set to 1.
When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]
The condition code for the trapped instruction.
When an A32 instruction is trapped, CV is set to 1 and:
• If the instruction is conditional, COND is set to the condition code field value from the instruction.
• If the instruction is unconditional, COND is set to 0b1110.
A conditional A32 instruction that is known to pass its condition code check can be presented either:
- With COND set to 0b1110, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [19:6]**

Reserved, RES0.

**TA, bit [5]**

Indicates trapped use of Advanced SIMD functionality.
- 0b0 Exception was not caused by trapped use of Advanced SIMD functionality.
- 0b1 Exception was caused by trapped use of Advanced SIMD functionality.

Any use of an Advanced SIMD instruction that is not also a floating-point instruction that is trapped to Hyp mode because of a trap configured in the HCPTR sets this bit to 1.

For a list of these instructions, see Controls of Advanced SIMD operation that do not apply to floating-point operation on page E1-7153.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [4]**

Reserved, RES0.

**coproc, bits [3:0]**

When the HSR.TA field returns the value 1, this field returns the value 0b1010. Otherwise, this field is RES0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

The following sections describe the configuration settings for the traps that are reported using EC value 0b000111:
- General trapping to Hyp mode of Non-secure accesses to the SIMD and floating-point registers on page G1-9021.
- Traps to Hyp mode of Non-secure accesses to Advanced SIMD functionality on page G1-9022.

**ISS encoding for exception from HVC or SVC instruction execution**

<table>
<thead>
<tr>
<th>24</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>imm16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [24:16]**

Reserved, RES0.
imm16, bits [15:0]
The value of the immediate field from the HVC or SVC instruction.
For an HVC instruction, this is the value of the imm16 field of the issued instruction.
For an SVC instruction:
• If the instruction is unconditional, then:
  — For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.
  — For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
• For the T32 instruction, this field is zero-extended from the imm8 field of the instruction. For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
• If the instruction is conditional, this field is UNKNOWN.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

The HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

Supervisor Call exception, when the value of HCR.TGE is 1 on page G1-8945 describes the configuration settings for the trap reported with EC value 0b010001.

ISS encoding for exception from SMC instruction execution

CV, bit [24]
Condition code valid. Possible values of this bit are:
0b0 The COND field is not valid.
0b1 The COND field is valid.
When an A32 instruction is trapped, CV is set to 1.
When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.
This field is valid only if CCKNOWNPASS is 1, otherwise it is RES0.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]
The condition code for the trapped instruction.
When an A32 instruction is trapped, CV is set to 1 and:
• If the instruction is conditional, COND is set to the condition code field value from the instruction.
• If the instruction is unconditional, COND is set to 0b1110.
A conditional A32 instruction that is known to pass its condition code check can be presented either:
• With COND set to 0b1110, the value for unconditional.
• With the COND value held in the instruction.
When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field is valid only if CCKNOWNPASS is 1, otherwise it is RES0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CCKNOWNPASS, bit [19]
Indicates whether the instruction might have failed its condition code check.

- 0b0 The instruction was unconditional, or was conditional and passed its condition code check.
- 0b1 The instruction was conditional, and might have failed its condition code check.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [18:0]
Reserved, RES0.

Traps to Hyp mode of Non-secure EL1 execution of SMC instructions on page G1-9017 describes the configuration settings for this trap, for instructions executed in Non-secure EL1.

ISS encoding for exception from a Prefetch Abort

Bits [24:11]
Reserved, RES0.

FnV, bit [10]
FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

- 0b0 HIFAR is valid.
- 0b1 HIFAR is not valid, and holds an UNKNOWN value.

This field is valid only if the IFSC code is 0b010000. It is RES0 for all other aborts.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]
External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [8]**

Reserved, RES0.

**S1PTW, bit [7]**

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

- 0b0 Fault not on a stage 2 translation for a stage 1 translation table walk.
- 0b1 Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [6]**

Reserved, RES0.

**IFSC, bits [5:0]**

Instruction Fault Status Code. Possible values of this field are:

- 0b000000 Address size fault in translation table base register.
- 0b000001 Address size fault, level 1.
- 0b000010 Address size fault, level 2.
- 0b000011 Address size fault, level 3.
- 0b000100 Translation fault, level 1.
- 0b000101 Translation fault, level 2.
- 0b000110 Translation fault, level 3.
- 0b001001 Access flag fault, level 1.
- 0b001010 Access flag fault, level 2.
- 0b001011 Access flag fault, level 3.
- 0b001101 Permission fault, level 1.
- 0b001110 Permission fault, level 2.
- 0b001111 Permission fault, level 3.
- 0b010000 Synchronous External abort, not on translation table walk.
- 0b010101 Synchronous External abort on translation table walk, level 1.
- 0b010110 Synchronous External abort on translation table walk, level 2.
- 0b010111 Synchronous External abort on translation table walk, level 3.
- 0b011000 When FEAT_RAS is not implemented:
  Synchronous parity or ECC error on memory access, not on translation table walk.
- 0b011101 When FEAT_RAS is not implemented:
  Synchronous parity or ECC error on memory access on translation table walk, level 1.
- 0b011110 When FEAT_RAS is not implemented:
  Synchronous parity or ECC error on memory access on translation table walk, level 2.
- 0b011111 When FEAT_RAS is not implemented:
  Synchronous parity or ECC error on memory access on translation table walk, level 3.
- 0b100010 Debug exception.
- 0b110000 TLB conflict abort.
All other values are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults on a Long-descriptor translation table lookup* on page G5-9259.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

The following sections describe cases where Prefetch Abort exceptions can be routed to Hyp mode, generating exceptions that are reported in the HSR with EC value 0b100000:
- *Abort exceptions, when the value of HCR.TGE is 1* on page G1-8945.
- *Routing debug exceptions to EL2 using AArch32* on page G1-8946.

### ISS encoding for exception from an Illegal state or PC alignment fault

<table>
<thead>
<tr>
<th>Bit 24</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>RES0</td>
</tr>
</tbody>
</table>

**Bits [24:0]**

Reserved, RES0.

For more information about the Illegal state exception, see:
- *Illegal changes to PSTATE.M* on page G1-8925.
- *Illegal return events from AArch32 state* on page G1-8952.
- *Legal returns that set PSTATE.IL to 1* on page G1-8954.
- *The Illegal Execution state exception* on page G1-8954.

For more information about the PC alignment fault exception, see *Branching to an unaligned PC* on page K1-11564.

### ISS encoding for exception from a Data Abort

<table>
<thead>
<tr>
<th>Bit 28</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAV</td>
<td>RES0</td>
</tr>
<tr>
<td>SRT</td>
<td>RES0</td>
</tr>
<tr>
<td>AP</td>
<td>RES0</td>
</tr>
<tr>
<td>EA</td>
<td>CM</td>
</tr>
<tr>
<td>WnR</td>
<td>S1PTW</td>
</tr>
<tr>
<td>DFSC</td>
<td>RES0</td>
</tr>
</tbody>
</table>

**ISV, bit [24]**

Instruction Syndrome Valid. Indicates whether the syndrome information in ISS[23:14] is valid.

- 0b0 No valid instruction syndrome. ISS[23:14] are RES0.
- 0b1 ISS[23:14] hold a valid instruction syndrome.

This bit is 0 for all faults except Data Abort exceptions generated by stage 2 address translations for which all the following apply to the instruction that generated the Data Abort exception:
- The instruction is an LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT instruction.
- The instruction is not performing register writeback.
- The instruction is not using the PC as a source or destination register.

For these cases, ISV is UNKNOWN if the exception was generated in Debug state in memory access mode, as described in Data Aborts in Memory access mode on page H4-10304, and otherwise indicates whether ISS[23:14] hold a valid syndrome.

--- Note ---

In the A32 instruction set, LDR*T and STR*T instructions always perform register writeback and therefore never return a valid instruction syndrome.

When FEAT_RAS is implemented, ISV is 0 for any synchronous External abort.
ISV is set to 0 on a stage 2 abort on a stage 1 translation table walk.

When FEAT_RAS is not implemented, it is IMPLEMENTATION DEFINED whether ISV is set to 1 or 0 on a synchronous External abort on a stage 2 translation table walk.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SAS, bits [23:22]

Syndrome Access Size. When ISV is 1, indicates the size of the access attempted by the faulting operation.
0b00  Byte
0b01  Halfword
0b10  Word
0b11  Doubleword

This field is UNKNOWN when the value of ISV is UNKNOWN.
This field is RES0 when the value of ISV is 0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SSE, bit [21]

Syndrome Sign Extend. When ISV is 1, for a byte, halfword, or word load operation, indicates whether the data item must be sign extended. For these cases, the possible values of this bit are:
0b0  Sign-extension not required.
0b1  Data item must be sign-extended.

For all other operations this bit is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.
This field is RES0 when the value of ISV is 0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [20]

Reserved, RES0.

SRT, bits [19:16]

Syndrome Register Transfer. When ISV is 1, the register number of the Rt operand of the faulting instruction.

This field is UNKNOWN when the value of ISV is UNKNOWN.
This field is RES0 when the value of ISV is 0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Bit [15]
Reserved, RES0.

AR, bit [14]
Acquire/Release. When ISV is 1, the possible values of this bit are:
0b0 Instruction did not have acquire/release semantics.
0b1 Instruction did have acquire/release semantics.
This field is UNKNOWN when the value of ISV is UNKNOWN.
This field is RES0 when the value of ISV is 0.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [13:12]
Reserved, RES0.

AET, bits [11:10]
When FEAT_RAS is implemented:
AET
Asynchronous Error Type. When DFSC is 0b010001, describes the PE error state after taking the SError interrupt exception.
0b00 Uncontainable (UC).
0b01 Unrecoverable state (UEU).
0b10 Restartable state (UEO).
0b11 Recoverable state (UER).
On a synchronous Data Abort exception, this field is RES0.
In the event of multiple errors taken as a single SError interrupt exception, the overall PE error state is reported.

——— Note ————
Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

——— ————
When FEAT_RAS is not implemented, or when DFSC is not 0b010001:
• Bit[11] is RES0.
• Bit[10] forms the FnV field.

——— Note ————
Armv8.2 requires the implementation of FEAT_RAS.

——— ————
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FnV, bit[0] of bits [11:10]
FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.
0b0 HDFAR is valid.
0b1 HDFAR is not valid, and holds a \textit{UNKNOWN} value.

When FEAT\_RAS is not implemented, this field is valid only if DFSC is 0b010000. It is RES0 for all other aborts.

When FEAT\_RAS is implemented:

\begin{itemize}
  \item If DFSC is 0b010000, this field is valid.
  \item If DFSC is 0b010001, this bit forms part of the AET field, becoming AET[0].
  \item This field is RES0 for all other aborts.
\end{itemize}

\textbf{Note}

Armv8.2 requires the implementation of FEAT\_RAS.

The reset behavior of this field is:

\begin{itemize}
  \item On a Warm reset, this field resets to an architecturally \textit{UNKNOWN} value.
\end{itemize}

\textbf{EA, bit [9]}

External Abort type. This bit can provide an \textit{IMPLEMENTATION DEFINED} classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

\begin{itemize}
  \item On a Warm reset, this field resets to an architecturally \textit{UNKNOWN} value.
\end{itemize}

\textbf{CM, bit [8]}

Cache Maintenance. For a synchronous fault, identifies fault that comes from a cache maintenance or address translation instruction. For synchronous faults, the possible values of this bit are:

\begin{itemize}
  \item 0b0 Fault not generated by a cache maintenance or address translation instruction.
  \item 0b1 Fault generated by a cache maintenance or address translation instruction.
\end{itemize}

For an asynchronous Data Abort exception, this bit is 0.

The reset behavior of this field is:

\begin{itemize}
  \item On a Warm reset, this field resets to an architecturally \textit{UNKNOWN} value.
\end{itemize}

\textbf{S1PTW, bit [7]}

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

\begin{itemize}
  \item 0b0 Fault not on a stage 2 translation for a stage 1 translation table walk.
  \item 0b1 Fault on the stage 2 translation of an access for a stage 1 translation table walk.
\end{itemize}

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

\begin{itemize}
  \item On a Warm reset, this field resets to an architecturally \textit{UNKNOWN} value.
\end{itemize}

\textbf{WnR, bit [6]}

Write not Read. Indicates whether a synchronous abort was caused by a write instruction or a read instruction.

\begin{itemize}
  \item 0b0 Abort caused by a read instruction.
  \item 0b1 Abort caused by a write instruction.
\end{itemize}

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

On an asynchronous Data Abort exception:

\begin{itemize}
  \item When FEAT\_RAS is not implemented, this bit is \textit{UNKNOWN}.
  \item When FEAT\_RAS is implemented, this bit is RES0.
\end{itemize}
Note

Armv8.2 requires the implementation of FEAT_RAS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code. Possible values of this field are:

- 0b000000: Address size fault in translation table base register.
- 0b000001: Address size fault, level 1.
- 0b000010: Address size fault, level 2.
- 0b000011: Address size fault, level 3.
- 0b000101: Translation fault, level 1.
- 0b000110: Translation fault, level 2.
- 0b000111: Translation fault, level 3.
- 0b001001: Access flag fault, level 1.
- 0b001010: Access flag fault, level 2.
- 0b001011: Access flag fault, level 3.
- 0b001101: Permission fault, level 1.
- 0b001110: Permission fault, level 2.
- 0b001111: Permission fault, level 3.
- 0b010000: Synchronous External abort, not on translation table walk.
- 0b010001: Asynchronous SError interrupt.
- 0b010101: Synchronous External abort on translation table walk, level 1.
- 0b010110: Synchronous External abort on translation table walk, level 2.
- 0b010111: Synchronous External abort on translation table walk, level 3.
- 0b011000: When FEAT_RAS is not implemented:
  Synchronous parity or ECC error on memory access, not on translation table walk.
- 0b011001: When FEAT_RAS is not implemented:
  Asynchronous SError interrupt, from a parity or ECC error on memory access.
- 0b011101: When FEAT_RAS is not implemented:
  Synchronous parity or ECC error on memory access on translation table walk, level 1.
- 0b011110: When FEAT_RAS is not implemented:
  Synchronous parity or ECC error on memory access on translation table walk, level 2.
- 0b011111: When FEAT_RAS is not implemented:
  Synchronous parity or ECC error on memory access on translation table walk, level 3.
- 0b100001: Alignment fault.
- 0b100010: Debug exception.
- 0b110000: TLB conflict abort.
- 0b110100: IMPLEMENTATION DEFINED fault (Lockdown).
- 0b110101: IMPLEMENTATION DEFINED fault (Unsupported Exclusive access).

All other values are reserved.

For more information about the lookup level associated with a fault, see The level associated with MMU faults on a Long-descriptor translation table lookup on page G5-9259.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

The following describe cases where Data Abort exceptions can be routed to Hyp mode, generating exceptions that are reported in the HSR with EC value **0b100100**:

- *Abort exceptions, when the value of HCR.TGE is 1 on page G1-8945.*
- *Routing debug exceptions to EL2 using AArch32 on page G1-8946.*

The following describe cases that can cause a Data Abort exception that is taken to Hyp mode, and reported in the HSR with EC value of **0b100000** or **0b100100**:

- *Hyp mode control of Non-secure access permissions on page G5-9201.*
- *Memory fault reporting in Hyp mode on page G5-9263.*

### Accessing HSR

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC} \{<c>\} \{<q>\} \text{<coproc>, \{#<opc1>, <Rt>, <CRn>, <CRm}\{, \{#<opc2}\}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

\[
\text{MCR} \{<c>\} \{<q>\} \text{<coproc>, \{#<opc1>, <Rt>, <CRn>, <CRm}\{, \{#<opc2}\}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

\[
\text{if PSTATE.EL == EL0 then}
\begin{align*}
\text{UNDEFINED;}
\end{align*}
\]

\[
\text{elsif PSTATE.EL == EL1 then}
\begin{align*}
\text{if EL2Enabled()} \&\& \text{!ELUsingAAArch32(EL2) \&\& HSTR_EL2.T5 == '1' then}
\text{AArch64.AArch32SystemAccessTrap(EL2, 0x03);}
\text{elsif EL2Enabled()} \&\& \text{ELUsingAAArch32(EL2) \&\& HSTR.T5 == '1' then}
\text{AArch32.TakeHypTrapException(0x03);}
\text{else}
\text{UNDEFINED;}
\end{align*}
\]

\[
\text{elsif PSTATE.EL == EL2 then}
\begin{align*}
\text{R[t] = HSR;}
\end{align*}
\]

\[
\text{elsif PSTATE.EL == EL3 then}
\begin{align*}
\text{if SCR.NS == '0' then}
\text{UNDEFINED;}
\text{else}
\text{R[t] = HSR;}
\end{align*}
\]
elsif PSTATE.EL == EL2 then
    HSR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HSR = R[t];
    end if;
else
    HSR = R[t];
G8.2.74 HSTR, Hyp System Trap Register

The HSTR characteristics are:

**Purpose**

Controls trapping to Hyp mode of Non-secure accesses, at EL1 or lower, to System registers in the coproc == 0b1111 encoding space:

- By the CRn value used to access the register using MCR or MRC instruction.
- By the CRm value used to access the register using MCRR or MRRC instruction.

**Configurations**

AArch32 System register HSTR bits [31:0] are architecturally mapped to AArch64 System register HSTR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HSTR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HSTR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0</td>
</tr>
<tr>
<td>30-16, 14-13</td>
<td>T15, T14, T13</td>
</tr>
<tr>
<td>12-8</td>
<td>T9, T8, T7, T6, T5</td>
</tr>
<tr>
<td>7-0</td>
<td>T3, T2, T1, T0</td>
</tr>
</tbody>
</table>

**Bits [31:16, 14, 4]**

Reserved, RES0.

**T<n>, bit [n], for n = 15, 13 to 5, 3 to 0**

The remaining fields control whether Non-secure EL0 and EL1 accesses, using MCR, MRC, MCRR, and MRRC instructions, to the System registers in the coproc == 0b1111 encoding space are trapped to Hyp mode:

- **0b0** This control has no effect on Non-secure EL0 or EL1 accesses to System registers.
- **0b1** Any Non-secure EL1 MCR or MRC access with coproc == 0b1111 and CRn == <n> is trapped to Hyp mode. A Non-secure EL0 MCR or MRC access with these values is trapped to Hyp mode only if the access is not UNDEFINED when the value of this field is 0.

For example, when HSTR.T7 is 1, for instructions executed at Non-secure EL1:

- An MCR or MRC instruction with coproc set to 0b1111 and <CRn> set to c7 is trapped to Hyp mode.
- An MCRR or MRRC instruction with coproc set to 0b1111 and <CRm> set to c7 is trapped to Hyp mode.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing HSTR

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}\{<q>\} <\text{coproc}, \{#\}<\text{opc1}>, <Rt>, <\text{CRn}>, <\text{CRm}>, \{#\}<\text{opc2}>\]
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b011</td>
</tr>
</tbody>
</table>

\[
\text{MCR}\{<c>\}\{<q>\} <\text{coproc}, \{#\}<\text{opc1}>, <Rt>, <\text{CRn}>, <\text{CRm}>, \{#\}<\text{opc2}>\]
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    HSTR = R[t];
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    HSTR = R[t];
G8.2.75 HTCR, Hyp Translation Control Register

The HTCR characteristics are:

**Purpose**

The control register for stage 1 of the EL2 translation regime.

--- Note ---

This stage of translation always uses the Long-descriptor translation table format.

**Configurations**

AArch32 System register HTCR bits [31:0] are architecturally mapped to AArch64 System register TCR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HTCR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HTCR is a 32-bit register.

**Field descriptions**

Bit [31]

Reserved, RES1.

**IMPLEMENTATION DEFINED, bit [30]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [29]

Reserved, RES0.

**HWU62, bit [28]**

*When FEAT_HPDS2 is implemented:*

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry.

0b0 Bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1 Bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of HTCR.HPD is 1.

The Effective value of this field is 0 if the value of HTCR.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
**Otherwise:**

Reserved, RES0.

**HWU61, bit [27]**

*When FEAT_HPDS2 is implemented:*

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry.

- **0b0**: Bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
- **0b1**: Bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of HTCR.HPD is 1.

The Effective value of this field is 0 if the value of HTCR.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**HWU60, bit [26]**

*When FEAT_HPDS2 is implemented:*

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry.

- **0b0**: Bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
- **0b1**: Bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of HTCR.HPD is 1.

The Effective value of this field is 0 if the value of HTCR.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**HWU59, bit [25]**

*When FEAT_HPDS2 is implemented:*

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry.

- **0b0**: Bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
- **0b1**: Bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of HTCR.HPD is 1.

The Effective value of this field is 0 if the value of HTCR.HPD is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.
### HPD, bit [24]

*When FEAT_AA32HPD is implemented:*

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, XNTable, and PXNTable, in the PL2 translation regime.

<table>
<thead>
<tr>
<th>Bit Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Hierarchical permissions are enabled.</td>
</tr>
<tr>
<td>01</td>
<td>Hierarchical permissions are disabled.</td>
</tr>
</tbody>
</table>

When disabled, the permissions are treated as if the bits are zero.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN value**.

### Bit [23]

Reserved, RES0.

### Bits [22:14]

Reserved, RES0.

### SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using HTTBR.

<table>
<thead>
<tr>
<th>Bit Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Non-shareable.</td>
</tr>
<tr>
<td>010</td>
<td>Outer Shareable.</td>
</tr>
<tr>
<td>011</td>
<td>Inner Shareable.</td>
</tr>
</tbody>
</table>

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is **CONSTRAINED UNPREDICTABLE**.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN value**.

### ORGN0, bits [11:10]

Outer cacheability attribute for memory associated with translation table walks using HTTBR.

<table>
<thead>
<tr>
<th>Bit Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Normal memory, Outer Non-cacheable.</td>
</tr>
<tr>
<td>001</td>
<td>Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.</td>
</tr>
<tr>
<td>010</td>
<td>Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.</td>
</tr>
<tr>
<td>011</td>
<td>Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN value**.

### IRGN0, bits [9:8]

Inner cacheability attribute for memory associated with translation table walks using HTTBR.

<table>
<thead>
<tr>
<th>Bit Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Normal memory, Inner Non-cacheable.</td>
</tr>
<tr>
<td>001</td>
<td>Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.</td>
</tr>
<tr>
<td>010</td>
<td>Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.</td>
</tr>
<tr>
<td>011</td>
<td>Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN value**.

### Bits [7:3]

Reserved, RES0.
T0SZ, bits [2:0]

The size offset of the memory region addressed by HTTBR. The region size is 2^(32-T0SZ) bytes.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HTCR

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>}\ <\text{coproc}, \{#\}<\text{opc1}>, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>, \{#\}<\text{opc2}>\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

\[
\text{MCR}\{<c>\}{<q>}\ <\text{coproc}, \{#\}<\text{opc1}>, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>, \{#\}<\text{opc2}>\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>
G8.2.76   HTPIDR, Hyp Software Thread ID Register

The HTPIDR characteristics are:

**Purpose**

Provides a location where software running in Hyp mode can store thread identifying information that is not visible to Non-secure software executing at EL0 or EL1, for hypervisor management purposes.

The PE makes no use of this register.

**Configurations**

AArch32 System register HTPIDR bits [31:0] are architecturally mapped to AArch64 System register TPIDR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HTPIDR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

--- Note ---

The PE never updates this register.

**Attributes**

HTPIDR is a 32-bit register.

**Field descriptions**

![Thread ID](image)

**Bits [31:0]**

- Thread ID. Thread identifying information stored by software running at this Exception level.
- The reset behavior of this field is:
  - On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing HTPIDR**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC\{c\}\{q\} <\text{coproc}\rangle, \langle#\rangle<\text{opc1}\rangle, <\text{Rt}\rangle, <\text{CRn}\rangle, <\text{CRm}\rangle, \langle#\rangle<\text{opc2}\rangle
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

\[
\text{if } \text{PSTATE.EL} = \text{EL0 then UNDEFINED;}
\]

\[
\text{elsif } \text{PSTATE.EL} = \text{EL1 then }
\]

\[
\text{if } \text{EL2Enabled()} \&\& !\text{ELUsingAArch32(EL2)} \&\& \text{HSTR_EL2.T13} = '1' \text{ then AArch64.AArch32SystemAccessTrap(EL2, 0x03);}
\]

\[
\text{elsif } \text{EL2Enabled()} \&\& \text{ELUsingAArch32(EL2)} \&\& \text{HSTR.T13} = '1' \text{ then AArch32.TakeHypTrapException(0x03);}
\]

\[
\text{else UNDEFINED;}
\]

\[
\text{elsif } \text{PSTATE.EL} = \text{EL2 then}
\]

---
R[t] = HTPIDR;
elsif PSTATE_EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    R[t] = HTPIDR;
endif

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
  endif
elsif PSTATE_EL == EL2 then
  HTPIDR = R[t];
elseif PSTATE_EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    HTPIDR = R[t];
### G8.2.77 HTTBR, Hyp Translation Table Base Register

The HTTBR characteristics are:

**Purpose**

Holds the base address of the translation table for the initial lookup for stage 1 of an address translation in the EL2 translation regime, and other information for this translation regime.

**Configurations**

AArch32 System register HTTBR bits [47:1] are architecturally mapped to AArch64 System register TTBR0_EL2[47:1].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HTTBR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HTTBR is a 64-bit register.

**Field descriptions**

![Field diagram](image)

**Bits [63:48]**

Reserved, RES0.

**BADDR, bits [47:1]**

Translation table base address, bits[47:x]. Bits [x-1:1] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:3] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

x is determined from the value of HTCR.T0SZ as follows:

- If HTCR.T0SZ is 0 or 1, x = 5 - HTCR.T0SZ.
- If HTCR.T0SZ is greater than 1, x = 14 - HTCR.T0SZ.

If bits[47:40] of the translation table base address are not zero, an Address size fault is generated. The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
CnP, bit [0]

**When FEAT_TTCNP is implemented:**

Common not Private. This bit indicates whether each entry that is pointed to by HTTBR is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of HTTBR.CnP is 1.

- **0b0** The translation table entries pointed to by HTTBR are permitted to differ from corresponding entries for HTTBR for other PEs in the Inner Shareable domain. This is not affected by the value of HTTBR.CnP on those other PEs.

- **0b1** The translation table entries pointed to by HTTBR are the same as the translation table entries pointed to by HTTBR on every other PE in the Inner Shareable domain for which the value of HTTBR.CnP is 1.

**Note**

If the value of the HTTBR.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those HTTBRs do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are **CONSTRAINED UNPREDICTABLE**, see **CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values** on page K1-11567.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Accessing HTTBR**

Accesses to this register use the following encodings in the System register encoding space:

**MRRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <Rt2>, <CRm>**

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0010</td>
<td>0b0100</td>
</tr>
</tbody>
</table>

```
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
    AArch32.TakeHypTrapException(0x04);
  else
    UNDEFINED;
elsif PSTATE_EL == EL2 then
  (R[t2], R[t]) = (HTTBR<63:32>, HTTBR<31:0>);
elsif PSTATE_EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    (R[t2], R[t]) = (HTTBR<63:32>, HTTBR<31:0>);
```
MCRR(<c>{<q>}{coproc}, (#{opc1}, <Rt>, <Rt2>, <CRm>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0010</td>
<td>0b0100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
    AArch32.TakeHypTrapException(0x04);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  HTTBR = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    HTTBR = R[t2]:R[t];
### G8.2.78 HVAR, Hyp Vector Base Address Register

The HVAR characteristics are:

**Purpose**

Holds the vector base address for any exception that is taken to Hyp mode.

**Configurations**

AArch32 System register HVAR bits [31:0] are architecturally mapped to AArch64 System register VBAR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HVAR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

HVAR is a 32-bit register.

**Field descriptions**

- **Bits [31:5]**
  
  Vector Base Address. Bits[31:5] of the base address of the exception vectors for exceptions taken to this Exception level. Bits[4:0] of an exception vector are the exception offset.

  The reset behavior of this field is:
  
  - On a Warm reset, this field resets to an architecturally UNKNOWN value.

- **Bits [4:0]**

  Reserved, RES0.

**Accessing HVAR**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC\{<c>\}{<q>} <coproc>, \{#<opc1>, <Rt>, <CRn>, <CRm}\{, \{#<opc2}>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
t[0] = HVAR;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
### MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    HVBAR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HVBAR = R[t];

G8.2.79 ICIALLU, Instruction Cache Invalidate All to PoU

The ICIALLU characteristics are:

**Purpose**

Invalidate all instruction caches of the PE executing the instruction to the Point of Unification. If branch predictors are architecturally visible, also flush branch predictors.

**Configurations**

AArch32 System register ICIALLU performs the same function as AArch64 System register IC IALLU.

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ICIALLU are UNDEFINED.

**Attributes**

ICIALLU is a 32-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

**Executing ICIALLU instruction**

The PE ignores the value of <Rt>. Software does not have to write a value to this register before issuing this instruction.

When HCR.FB is 1, at Non-secure EL1 this instruction executes as a ICIALLUIS.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[ MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0101</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPU == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TOCU == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TPU == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TOCU == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.FB == '1' then
        AArch32.IC(CacheOpScope_ALLUIS);
    else
        AArch32.IC(CacheOpScope_ALLU);
    elsif PSTATE.EL == EL2 then
        AArch32.IC(CacheOpScope_ALLU);
elsif PSTATE_EL == EL3 then
    AArch32.IC(CacheOpScope_ALLU);
G8.2.80  ICIALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable

The ICIALLUIS characteristics are:

**Purpose**

Invalidate all instruction caches in the Inner Shareable domain of the PE executing the instruction to the Point of Unification. If branch predictors are architecturally visible, also flush branch predictors.

**Configurations**

AArch32 System register ICIALLUIS performs the same function as AArch64 System register IC IALLUIS.

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ICIALLUIS are UNDEFINED.

**Attributes**

ICIALLUIS is a 32-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

**Executing ICIALLUIS instruction**

The PE ignores the value of <Rt>. Software does not have to write a value to this register before issuing this instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
MCR\{<c>\}{<q>} \text{<coproc>, } \text{(##<opc1>, } \text{<Rt>, } \text{<CRn>, } \text{<CRm>}(, \text{(##<opc2>)}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPU == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TICAB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TPU == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TICAB == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        AArch32.IC(CacheOpScope_ALLUIS);
    elsif PSTATE.EL == EL2 then
        AArch32.IC(CacheOpScope_ALLUIS);
    elsif PSTATE.EL == EL3 then
        AArch32.IC(CacheOpScope_ALLUIS);
G8.2.81 ICIMVAU, Instruction Cache line Invalidate by VA to PoU

The ICIMVAU characteristics are:

Purpose

Invalidate instruction cache line by virtual address to PoU.

Configurations

AArch32 System register ICIMVAU performs the same function as AArch64 System register ICIVAU.

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ICIMVAU are UNDEFINED.

Attributes

ICIMVAU is a 32-bit System instruction.

Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
</table>
|     Virtual address to use

Bits [31:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing ICIMVAU instruction

Execution of this instruction might require an address translation from VA to PA, and that translation might fault.

For more information about faults, see Permission fault on page G5-9239.

For more information about data cache maintenance instructions, see AArch32 instruction cache maintenance instructions (IC*) on page G4-9124.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\textit{MCR\{<c>\}{<q>}} \text{<coproc>, \{#<opc1>, <Rt>, <CRn>, <CRm}\} {, \{#<opc2>}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0101</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TPU == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TOCU == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TPU == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TOCU == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        
else

ARM DDI 04871.a  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. Non-Confidential
AArch32.IC(R[t], CacheOpScope_PoU);
elsif PSTATE_EL == EL2 then
  AArch32.IC(R[t], CacheOpScope_PoU);
elsif PSTATE_EL == EL3 then
  AArch32.IC(R[t], CacheOpScope_PoU);
G8.2.82  ID_AFR0, Auxiliary Feature Register 0

The ID_AFR0 characteristics are:

**Purpose**

Provides information about the IMPLEMENTATION DEFINED features of the PE in AArch32 state.

Must be interpreted with the Main ID Register, MIDR.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register ID_AFR0 bits [31:0] are architecturally mapped to AArch64 System register ID_AFR0_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_AFR0 are UNDEFINED.

**Attributes**

ID_AFR0 is a 32-bit register.

**Field descriptions**

![Field descriptions diagram]

**Bits [31:16]**

Reserved, RES0.

**IMPLEMENTATION DEFINED, bits [15:12]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [11:8]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [7:4]**

IMPLEMENTATION DEFINED.

**IMPLEMENTATION DEFINED, bits [3:0]**

IMPLEMENTATION DEFINED.

**Accessing ID_AFR0**

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        R[t] = ID_AFR0;
    endif
elsif PSTATE.EL == EL2 then
    R[t] = ID_AFR0;
elsif PSTATE.EL == EL3 then
    R[t] = ID_AFR0;
G8.2.83 ID_DFR0, Debug Feature Register 0

The ID_DFR0 characteristics are:

**Purpose**

Provides top level information about the debug system in AArch32 state.

Must be interpreted with the Main ID Register, MIDR.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register ID_DFR0 bits [31:0] are architecturally mapped to AArch64 System register ID_DFR0_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_DFR0 are UNDEFINED.

**Attributes**

ID_DFR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>24</th>
<th>19</th>
<th>15</th>
<th>12</th>
<th>8</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PerfMon</td>
<td>MProfDbg</td>
<td>MMapTrc</td>
<td>CopTrc</td>
<td>MMapDbg</td>
<td>CopSDbg</td>
<td>CopDbg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TraceFilt, bits [31:28]**

Armv8.4 Self-hosted Trace Extension version. Defined values are:

0b0000  Armv8.4 Self-hosted Trace Extension not implemented.
0b0001  Armv8.4 Self-hosted Trace Extension implemented.

All other values are reserved.

**FEAT_TRF** implements the functionality added by the value 0b0001.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

**PerfMon, bits [27:24]**

Performance Monitors Extension version.

This field does not follow the standard ID scheme, but uses the alternative ID scheme described in *Alternative ID scheme used for the Performance Monitors Extension version* on page G8-9334.

Defined values are:

0b0000  Performance Monitors Extension not implemented.
0b0001  Performance Monitors Extension, PMUv1 implemented.
0b0010  Performance Monitors Extension, PMUv2 implemented.
0b0011  Performance Monitors Extension, PMUv3 implemented.
0b0100  PMUv3 for Armv8.1. As 0b0011, and adds support for:

• Extended 16-bit PMEVTPYR<\(n\)>.evtCount field.
• If EL2 is implemented, the HDCR.HPMD control.
0b0101  PMUv3 for Armv8.4. As 0b0100, and adds support for the PMMIR register.
0b0110  PMUv3 for Armv8.5. As 0b0101, and adds support for:

• 64-bit event counters.
• If EL2 is implemented, the HDCR.HCCD control.
• If EL3 is implemented, the SDCR.SCCD control.

0b0111  PMUv3 for Armv8.7. As 0b0110, and adds support for:
  • The PMCR.FZO and, if EL2 is implemented, HPCR.HPMFZO controls.
  • If EL3 is implemented and using AArch64, the MDCR_EL3.{MPMX,MCCD} controls.

0b1000  PMUv3 for Armv8.8. As 0b0111, and:
  • Extends the Common event number space to include 0x0040 to 0x00BF and 0x4040 to 0x40BF.
  • Removes the CONSTRANGED UNPREDICTABLE behaviors if a reserved or unimplemented PMU event number is selected.

0b1111  IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported. Arm does not recommend this value for new implementations.

All other values are reserved.

FEAT_PMUv3 implements the functionality identified by the value 0b0011.
FEAT_PMUv3p1 implements the functionality identified by the value 0b0100.
FEAT_PMUv3p4 implements the functionality identified by the value 0b0101.
FEAT_PMUv3p5 implements the functionality identified by the value 0b0110.
FEAT_PMUv3p7 implements the functionality identified by the value 0b0111.
FEAT_PMUv3p8 implements the functionality identified by the value 0b1000.

In any Armv8 implementation, the values 0b0001 and 0b0010 are not permitted.
From Armv8.1, if FEAT_PMUv3 is implemented, the value 0b0011 is not permitted.
From Armv8.4, if FEAT_PMUv3 is implemented, the value 0b0100 is not permitted.
From Armv8.5, if FEAT_PMUv3 is implemented, the value 0b0101 is not permitted.
From Armv8.7, if FEAT_PMUv3 is implemented, the value 0b0110 is not permitted.
From Armv8.8, if FEAT_PMUv3 is implemented, the value 0b0111 is not permitted.

Note

PMUv1 and PMUv2 are not permitted in an Armv8 implementation.

MPProfDbg, bits [23:20]
M-profile Debug. Support for memory-mapped debug model for M-profile processors. Defined values are:
  0b0000  Not supported.
  0b0001  Support for M-profile Debug architecture, with memory-mapped access.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

MMapTrc, bits [19:16]
Memory-mapped Trace. Support for memory-mapped trace model. Defined values are:
  0b0000  Not supported.
  0b0001  Support for Arm trace architecture, with memory-mapped access.
All other values are reserved.
In Armv8-A, the permitted values are 0b0000 and 0b0001.
For more information, see the ARM® Embedded Trace Macrocell Architecture Specification, ETMv4 (ARM IHI 0064).
CopTrc, bits [15:12]
Support for System registers-based trace model, using registers in the coproc == 0b1110 encoding space. Defined values are:

- 0b0000  Not supported.
- 0b0001  Support for Arm trace architecture, with System registers access.

All other values are reserved.
In Armv8-A, the permitted values are 0b0000 and 0b0001.
For more information, see the ARM® Embedded Trace Macrocell Architecture Specification, ETMv4 (ARM IHI 0064).

MMapDbg, bits [11:8]
Memory-mapped Debug. Support for Armv7 memory-mapped debug model for A and R-profile processors. Defined values are:

- 0b0000  Not supported.
- 0b0100  Support for Armv7, v7 Debug architecture, with memory-mapped access.
- 0b0101  Support for Armv7, v7.1 Debug architecture, with memory-mapped access.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.
The optional memory map defined by Armv8 is not compatible with Armv7.

CopSDbg, bits [7:4]
Support for a System registers-based Secure debug model, using registers in the coproc == 0b1110 encoding space, for an A-profile processor that includes EL3.
If EL3 is not implemented and the implemented Security state is Non-secure state, this field is RES0. Otherwise, this field reads the same as bits [3:0].

CopDbg, bits [3:0]
Debug architecture version. Indicates presence of Armv8 debug architecture. Defined values are:

- 0b0000  Not supported.
- 0b0010  Armv6, v6 Debug architecture, with System registers access.
- 0b0011  Armv6, v6.1 Debug architecture, with System registers access.
- 0b0100  Armv7, v7 Debug architecture, with System registers access.
- 0b0101  Armv7, v7.1 Debug architecture, with System registers access.
- 0b0110  Armv8 debug architecture.
- 0b0111  Armv8 debug architecture with Virtualization Host Extensions.
- 0b1000  Armv8.2 debug architecture, FEAT_Debugv8p2.
- 0b1001  Armv8.4 debug architecture, FEAT_Debugv8p4.
- 0b1010  Armv8.8 debug architecture, FEAT_Debugv8p8.

All other values are reserved.
The values 0b0000, 0b0010, 0b0011, 0b0100, and 0b0101 are not permitted in Armv8.
FEAT_VHE adds the functionality identified by the value 0b0011.
FEAT_Debugv8p2 adds the functionality identified by the value 0b1000.
FEAT_Debugv8p4 adds the functionality identified by the value 0b1001.
FEAT_Debugv8p8 adds the functionality identified by the value 0b1010.
From Armv8.1, when FEAT_VHE is implemented the value 0b0110 is not permitted.
From Armv8.2, the values 0b0110 and 0b0111 are not permitted.
From Armv8.4, the value 0b1000 is not permitted.
From Armv8.8, the value 0b1001 is not permitted.

**Accessing ID_DFR0**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC} \langle c \rangle \{q\} \langle \text{coproc} \rangle, \langle \# \text{opc1} \rangle, \langle R_t \rangle, \langle \text{CRn} \rangle, \langle \text{CRm} \rangle, \langle \# \text{opc2} \rangle
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && HCR_EL2.TID3 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    R[t] = ID_DFR0;
elsif PSTATE.EL == EL2 then
  R[t] = ID_DFR0;
elsif PSTATE.EL == EL3 then
  R[t] = ID_DFR0;
ID_DFR1, Debug Feature Register 1

The ID_DFR1 characteristics are:

**Purpose**

Provides top level information about the debug system in AArch32.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register ID_DFR1 bits [31:0] are architecturally mapped to AArch64 System register ID_DFR1_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_DFR1 are UNDEFINED.

**Note**

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

**Attributes**

ID_DFR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0</td>
</tr>
<tr>
<td>8</td>
<td>HPMN0</td>
</tr>
<tr>
<td>4</td>
<td>MTPMU</td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**HPMN0, bits [7:4]**

Zero PMU event counters for a Guest operating system. Defined values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>Setting HDCR.HPMN to zero has CONSTRAINED UNPREDICTABLE behavior.</td>
</tr>
<tr>
<td>0b0001</td>
<td>Setting HDCR.HPMN to zero has defined behavior.</td>
</tr>
</tbody>
</table>

All other values are reserved.

If FEAT_PMUv3 is not implemented, FEAT_FGT is not implemented, or EL2 is not implemented, the only permitted value is 0b0000.

FEAT_HPMN0 implements the functionality identified by the value 0b0001.

From Armv8.8, in an implementation that includes FEAT_PMUv3, FEAT_FGT, and EL2, the value 0b0000 is not permitted.

**MTPMU, bits [3:0]**

Multi-threaded PMU extension. Defined values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>FEAT_MTPMU not implemented. If FEAT_PMUv3 is implemented, it is IMPLEMENTATION DEFINED whether PMEVTYPER&lt;\n&gt;.MT are read/write or RES0.</td>
</tr>
<tr>
<td>0b0001</td>
<td>FEAT_MTPMU and FEAT_PMUv3 implemented. PMEVTYPER&lt;\n&gt;.MT are read/write. When FEAT_MTPMU is disabled, the Effective values of PMEVTYPER&lt;\n&gt;.MT are 0.</td>
</tr>
<tr>
<td>0b1111</td>
<td>FEAT_MTPMU not implemented. If FEAT_PMUv3 is implemented, PMEVTYPER&lt;\n&gt;.MT are RES0.</td>
</tr>
</tbody>
</table>

All other values are reserved.
FEAT_MTPMU implements the functionality identified by the value 0b0001.
From Armv8.6, in an implementation that includes FEAT_PMUv3, the value 0b0000 is not permitted.
In an implementation that does not include FEAT_PMUv3, the value 0b0001 is not permitted.

Accessing ID_DFR1
Accesses to this register use the following encodings in the System register encoding space:

\[ MRC\{<c>\}{<q>\} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>\{, {#}<opc2>\} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0011</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_DFR1) ||
    boolean IMPLEMENTATION_DEFINED "ID_DFR1 trapped by HCR_EL2.TID3"
    && HCR_EL2.TID3 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && (IsFeatureImplemented(FEAT_FGT) ||
    IsZero(ID_DFR1) ||
    boolean IMPLEMENTATION_DEFINED "ID_DFR1 trapped by HCR.TID3"
    && HCR.TID3 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    \[ R[t] = ID_DFR1; \]
  elsif PSTATE.EL == EL2 then
    \[ R[t] = ID_DFR1; \]
  elsif PSTATE.EL == EL3 then
    \[ R[t] = ID_DFR1; \]
ID_ISAR0, Instruction Set Attribute Register 0

The ID_ISAR0 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register ID_ISAR0 bits [31:0] are architecturally mapped to AArch64 System register ID_ISAR0_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_ISAR0 are UNDEFINED.

**Attributes**

ID_ISAR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>23</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESO</td>
<td>Divide</td>
<td>Debug</td>
<td>Coproc</td>
<td>BitField</td>
<td>BitCount</td>
<td>Swap</td>
<td>CmpBranch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:28]**

Reserved, RES0.

**Divide, bits [27:24]**

Indicates the implemented Divide instructions. Defined values are:

0b0000  None implemented.

0b0001  Adds SDIV and UDIV in the T32 instruction set.

0b0010  As for 0b0001, and adds SDIV and UDIV in the A32 instruction set.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0010.

**Debug, bits [23:20]**

Indicates the implemented Debug instructions. Defined values are:

0b0000  None implemented.

0b0001  Adds BKPT.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

**Coproc, bits [19:16]**

Indicates the implemented System register access instructions. Defined values are:

0b0000  None implemented, except for instructions separately attributed by the architecture to provide access to AArch32 System registers and System instructions.

0b0001  Adds generic CDP, LDC, MCR, MRC, and STC.

0b0010  As for 0b0001, and adds generic CDP2, LDC2, MCR2, MRC2, and STC2.

0b0011  As for 0b0010, and adds generic MCRR and MRRC.
0b0100  As for 0b0011, and adds generic MCRR2 and MRRC2.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

**CmpBranch, bits [15:12]**
Indicates the implemented combined Compare and Branch instructions in the T32 instruction set.
Defined values are:
0b0000  None implemented.
0b0001  Adds CBNZ and CBZ.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

**BitField, bits [11:8]**
Indicates the implemented BitField instructions. Defined values are:
0b0000  None implemented.
0b0001  Adds BFC, BFI, SBFX, and UBFX.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

**BitCount, bits [7:4]**
Indicates the implemented Bit Counting instructions. Defined values are:
0b0000  None implemented.
0b0001  Adds CLZ.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

**Swap, bits [3:0]**
Indicates the implemented Swap instructions in the A32 instruction set. Defined values are:
0b0000  None implemented.
0b0001  Adds SWP and SWPB.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

**Accessing ID_ISAR0**
Accesses to this register use the following encodings in the System register encoding space:

```plaintext
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
    AArch32.TakeHypTrapException(0x03);
else
    R[t] = ID_ISAR0;
elsif PSTATE.EL == EL2 then
    R[t] = ID_ISAR0;
elsif PSTATE.EL == EL3 then
    R[t] = ID_ISAR0;
G8.2.86  ID_ISAR1, Instruction Set Attribute Register 1

The ID_ISAR1 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register ID_ISAR1 bits [31:0] are architecturally mapped to AArch64 System register ID_ISAR1_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_ISAR1 are UNDEFINED.

**Attributes**

ID_ISAR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Jazelle</td>
<td>Indicates the implemented Jazelle extension instructions. Defined values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0000 No support for Jazelle.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0001 Adds the BXJ instruction, and the J bit in the PSR. This setting might indicate a trivial implementation of the Jazelle extension.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other values are reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In Armv8-A, the only permitted value is 0b0001.</td>
</tr>
<tr>
<td>27</td>
<td>Interwork</td>
<td>Indicates the implemented Interworking instructions. Defined values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0000 None implemented.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0001 Adds the BX instruction, and the T bit in the PSR.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0010 As for 0b0001, and adds the BLX instruction. PC loads have BX-like behavior.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0011 As for 0b0010, and guarantees that data-processing instructions in the A32 instruction set with the PC as the destination and the S bit clear have BX-like behavior.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other values are reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In Armv8-A, the only permitted value is 0b0011.</td>
</tr>
<tr>
<td>20</td>
<td>Immediate</td>
<td>Indicates the implemented data-processing instructions with long immediates. Defined values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0000 None implemented.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0001 Adds:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The MOVT instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The MOV instruction encodings with zero-extended 16-bit immediates.</td>
</tr>
</tbody>
</table>
• The T32 ADD and SUB instruction encodings with zero-extended 12-bit immediates, and the other ADD, ADR, and SUB encodings cross-referenced by the pseudocode for those encodings.

All other values are reserved.
In Armv8-A, the only permitted value is \texttt{0b0001}.

\textbf{IfThen, bits [19:16]}

Indicates the implemented If-Then instructions in the T32 instruction set. Defined values are:

\begin{itemize}
  \item \texttt{0b0000}  None implemented.
  \item \texttt{0b0001}  Adds the IT instructions, and the IT bits in the PSRs.
\end{itemize}

All other values are reserved.
In Armv8-A, the only permitted value is \texttt{0b0001}.

\textbf{Extend, bits [15:12]}

Indicates the implemented Extend instructions. Defined values are:

\begin{itemize}
  \item \texttt{0b0000}  No scalar sign-extend or zero-extend instructions are implemented, where scalar instructions means non-Advanced SIMD instructions.
  \item \texttt{0b0001}  Adds the SXTB, SXTH, UXTB, and UXTH instructions.
  \item \texttt{0b0010}  As for \texttt{0b0001}, and adds the SXTB16, SXTAB, SXTAB16, SXTAH, UXTB16, UXTAB, UXTAB16, and UXTAH instructions.
\end{itemize}

All other values are reserved.
In Armv8-A, the only permitted value is \texttt{0b0010}.

\textbf{Except_AR, bits [11:8]}

Indicates the implemented A and R-profile exception-handling instructions. Defined values are:

\begin{itemize}
  \item \texttt{0b0000}  None implemented.
  \item \texttt{0b0001}  Adds the SRS and RFE instructions, and the A and R-profile forms of the CPS instruction.
\end{itemize}

All other values are reserved.
In Armv8-A, the only permitted value is \texttt{0b0001}.

\textbf{Except, bits [7:4]}

Indicates the implemented exception-handling instructions in the A32 instruction set. Defined values are:

\begin{itemize}
  \item \texttt{0b0000}  Not implemented. This indicates that the User bank and Exception return forms of the LDM and STM instructions are not implemented.
  \item \texttt{0b0001}  Adds the LDM (exception return), LDM (user registers), and STM (user registers) instruction versions.
\end{itemize}

All other values are reserved.
In Armv8-A, the only permitted value is \texttt{0b0001}.

\textbf{Endian, bits [3:0]}

Indicates the implemented Endian instructions. Defined values are:

\begin{itemize}
  \item \texttt{0b0000}  None implemented.
  \item \texttt{0b0001}  Adds the SETEND instruction, and the E bit in the PSRs.
\end{itemize}

All other values are reserved.
In Armv8-A, the permitted values are \texttt{0b0000} and \texttt{0b0001}. 
Accessing ID_ISAR1

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC\{<c>\}{<q>} \text{ <coproc>}, \text{ (#<opc1>), <Rt>, <CRn>, <CRm>}, \text{ (#<opc2>)} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    R[t] = ID_ISAR1;
  endif
elsif PSTATE.EL == EL2 then
  R[t] = ID_ISAR1;
elsif PSTATE.EL == EL3 then
  R[t] = ID_ISAR1;
G8.2.87  ID_ISAR2, Instruction Set Attribute Register 2

The ID_ISAR2 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32 state.
Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR3, ID_ISAR4, and ID_ISAR5.
For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register ID_ISAR2 bits [31:0] are architecturally mapped to AArch64 System register ID_ISAR2_EL1[31:0].
This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_ISAR2 are UNDEFINED.

**Attributes**

ID_ISAR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Positions</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-28</td>
<td>Reversal</td>
<td>Indicates the implemented Reversal instructions. Defined values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0000  None implemented.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0001  Adds the REV, REV16, and REVSH instructions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0010  As for 0b0001, and adds the RBIT instruction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other values are reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In Armv8-A, the only permitted value is 0b0010.</td>
</tr>
<tr>
<td>27-24</td>
<td>PSR_AR</td>
<td>Indicates the implemented A and R-profile instructions to manipulate the PSR. Defined values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0000  None implemented.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0001  Adds the MRS and MSR instructions, and the exception return forms of data-processing instructions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other values are reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In Armv8-A, the only permitted value is 0b0001.</td>
</tr>
<tr>
<td>23-20</td>
<td>MultU</td>
<td>Indicates the implemented advanced unsigned Multiply instructions. Defined values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0000  None implemented.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0001  Adds the UMULL and UMLAL instructions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0010  As for 0b0001, and adds the UMAAL instruction.</td>
</tr>
<tr>
<td>19-16</td>
<td>MultS</td>
<td></td>
</tr>
<tr>
<td>15-12</td>
<td>Mult</td>
<td></td>
</tr>
<tr>
<td>11-0</td>
<td>MemHint</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>MultiAccessInt</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>LoadStore</td>
<td></td>
</tr>
</tbody>
</table>

**Reversal, bits [31:28]**

Indicates the implemented Reversal instructions. Defined values are:

- 0b0000: None implemented.
- 0b0001: Adds the REV, REV16, and REVSH instructions.
- 0b0010: As for 0b0001, and adds the RBIT instruction.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0010.

**PSR_AR, bits [27:24]**

Indicates the implemented A and R-profile instructions to manipulate the PSR. Defined values are:

- 0b0000: None implemented.
- 0b0001: Adds the MRS and MSR instructions, and the exception return forms of data-processing instructions.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.
The exception return forms of the data-processing instructions are:
- In the A32 instruction set, data-processing instructions with the PC as the destination and the S bit set. These instructions might be affected by the WithShifts attribute.
- In the T32 instruction set, the SUBS PC,LR,#N instruction.

**MultU, bits [23:20]**

Indicates the implemented advanced unsigned Multiply instructions. Defined values are:

- 0b0000: None implemented.
- 0b0001: Adds the UMULL and UMLAL instructions.
- 0b0010: As for 0b0001, and adds the UMAAL instruction.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0010.

**MultS, bits [19:16]**
Indicates the implemented advanced signed Multiply instructions. Defined values are:

- **0b0000**: None implemented.
- **0b0001**: Adds the SMULL and SMLAL instructions.
- **0b0010**: As for 0b0001, and adds the SMLABB, SMLABT, SMLALBB, SMLALBT, SMLALT, SMLATB, SMLATT, SMLAWB, SMLAWT, SMULBB, SMULBT, SMULTB, SMULWT, SMLULB, SMLULT, SMLUTB, SMLWBT, and SMULWT instructions. Also adds the Q bit in the PSRs.
- **0b0011**: As for 0b0010, and adds the SMLAD, SMLADX, SMLALD, SMLALDX, SMLSD, SMLSDX, SMLSLD, SMLSDLX, SMLLA, SMLRAR, SMMLS, SMMLSR, SMMLUL, SMMLULR, SMMUL, SMMULR, SMUAD, SMUADX, SMUSD, and SMUSDX instructions.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0011.

**Mult, bits [15:12]**
Indicates the implemented additional Multiply instructions. Defined values are:

- **0b0000**: No additional instructions implemented. This means only MUL is implemented.
- **0b0001**: Adds the MLA instruction.
- **0b0010**: As for 0b0001, and adds the MLS instruction.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0010.

**MultiAccessInt, bits [11:8]**
Indicates the support for interruptible multi-access instructions. Defined values are:

- **0b0000**: No support. This means the LDM and STM instructions are not interruptible.
- **0b0001**: LDM and STM instructions are restartable.
- **0b0010**: LDM and STM instructions are continuable.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

**MemHint, bits [7:4]**
Indicates the implemented Memory Hint instructions. Defined values are:

- **0b0000**: None implemented.
- **0b0001**: Adds the PLD instruction.
- **0b0010**: Adds the PLD instruction. (0b0001 and 0b0010 have identical effects.)
- **0b0011**: As for 0b0001 (or 0b0010), and adds the PLI instruction.
- **0b0100**: As for 0b0011, and adds the PLDW instruction.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0100.

**LoadStore, bits [3:0]**
Indicates the implemented additional load/store instructions. Defined values are:

- **0b0000**: No additional load/store instructions implemented.
- **0b0001**: Adds the LDRD and STRD instructions.
As for 0b0001, and adds the Load Acquire (LDAB, LDAH, LDA, LDAEXB, LDAEXH, LDAEX, LDAEXD) and Store Release (STLB, STLH, STL, STLEXB, STLEXH, STLEX, STLEXD) instructions.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0010.

### Accessing ID_ISAR2

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
  R[t] = ID_ISAR2;
elif PSTATE.EL == EL2 then
  R[t] = ID_ISAR2;
elif PSTATE.EL == EL3 then
  R[t] = ID_ISAR2;
G8.2.88  ID_ISAR3, Instruction Set Attribute Register 3

The ID_ISAR3 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32 state.
Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR4, and ID_ISAR5.
For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register ID_ISAR3 bits [31:0] are architecturally mapped to AArch64 System register ID_ISAR3_EL1[31:0].
This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_ISAR3 are UNDEFINED.

**Attributes**

ID_ISAR3 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>23</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>T32EE</td>
<td>TrueNOP</td>
<td>T32Copy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SVC</td>
<td>SIMD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TabBranch</td>
<td></td>
<td></td>
<td></td>
<td>SynchPrim</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**T32EE, bits [31:28]**

Indicates the implemented T32EE instructions. Defined values are:

- 0b0000  None implemented.
- 0b0001  Adds the ENTERX and LEA VEX instructions, and modifies the load behavior to include null checking.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

**TrueNOP, bits [27:24]**

Indicates the implemented true NOP instructions. Defined values are:

- 0b0000  None implemented. This means there are no NOP instructions that do not have any register dependencies.
- 0b0001  Adds true NOP instructions in both the T32 and A32 instruction sets. This also permits additional NOP-compatible hints.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

**T32Copy, bits [23:20]**

Indicates the support for T32 non flag-setting MOV instructions. Defined values are:

- 0b0000  Not supported. This means that in the T32 instruction set, encoding T1 of the MOV (register) instruction does not support a copy from a low register to a low register.
- 0b0001  Adds support for T32 instruction set encoding T1 of the MOV (register) instruction, copying from a low register to a low register.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.
TabBranch, bits [19:16]
Indicates the implemented Table Branch instructions in the T32 instruction set. Defined values are:

0b0000  None implemented.
0b0001  Adds the TBB and TBH instructions.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

SynchPrim, bits [15:12]
Used in conjunction with ID_ISAR4.SynchPrim_frac to indicate the implemented Synchronization Primitive instructions. Defined values are:

0b0000  If SynchPrim_frac == 0b000, no Synchronization Primitives implemented.
0b0001  If SynchPrim_frac == 0b000, adds the LDREX and STREX instructions.
         If SynchPrim_frac == 0b011, also adds the CLREX, LDREXB, STREXB, and STREXH instructions.
0b0010  If SynchPrim_frac == 0b000, as for [0b001, 0b011] and also adds the LDREXD and STREXD instructions.
All other combinations of SynchPrim and SynchPrim_frac are reserved.
In Armv8-A, the only permitted value is 0b0010.

SVC, bits [11:8]
Indicates the implemented SVC instructions. Defined values are:

0b0000  Not implemented.
0b0001  Adds the SVC instruction.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

SIMD, bits [7:4]
Indicates the implemented SIMD instructions. Defined values are:

0b0000  None implemented.
0b0001  Adds the SSAT and USAT instructions, and the Q bit in the PSRs.
0b0011  As for 0b0001, and adds the PKHBT, PKHTB, QADD16, QADD8, QASX, QSUB16, QSUB8, QSAX, SADD16, SADD8, SASX, SEL, SHADD16, SHADD8, SHASX, SHSUB16, SHSUB8, SHSAX, SSAT16, SSUB16, SSUB8, SSAX, SXTAB16, SXTB16, UADD16, UADD8, UASX, UHADD16, UHADD8, UHASX, UHSUB16, UHSUB8, UHSAX, UQADD16, UQADD8, UQASX, UQSUB16, UQSUB8, UQSAX, USAD8, USADA8, USAT16, USUB16, USUB8, USAX, UXTAB16, and UXTB16 instructions. Also adds support for the GE[3:0] bits in the PSRs.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0011.
The SIMD field relates only to implemented instructions that perform SIMD operations on the general-purpose registers. In an implementation that supports Advanced SIMD and floating-point instructions, MVFR0, MVFR1, and MVFR2 give information about the implemented Advanced SIMD instructions.

Saturate, bits [3:0]
Indicates the implemented Saturate instructions. Defined values are:

0b0000  None implemented. This means no non-Advanced SIMD saturate instructions are implemented.
0b0001  Adds the QADD, QDADD, QDSUB, and QSUB instructions, and the Q bit in the PSRs.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

**Accessing ID_ISAR3**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC\{c\}{q\} <coproc>, \{\#}<opc1>, <Rt>, <CRn>, <CRm>{, \{\#}<opc2>}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
else
    R[t] = ID_ISAR3;
elsif PSTATE.EL == EL2 then
    R[t] = ID_ISAR3;
elsif PSTATE.EL == EL3 then
    R[t] = ID_ISAR3;
G8.2.89 **ID_ISAR4, Instruction Set Attribute Register 4**

The ID_ISAR4 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, and ID_ISAR5.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register ID_ISAR4 bits [31:0] are architecturally mapped to AArch64 System register ID_ISAR4_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_ISAR4 are UNDEFINED.

**Attributes**

ID_ISAR4 is a 32-bit register.

**Field descriptions**

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| SWP_frac | PSR_M | Barrier | SMC | | | | | | | | | | | | | | | | | | | | | |
| SynchPrim_frac | | WithShifts | | Writeback |

**SWP_frac, bits [31:28]**

Indicates support for the memory system locking the bus for SWP or SWPB instructions. Defined values are:

- 0b0000: SWP or SWPB instructions not implemented.
- 0b0001: SWP or SWPB implemented but only in a uniprocessor context. SWP and SWPB do not guarantee whether memory accesses from other Requesters can come between the load memory access and the store memory access of the SWP or SWPB.

All other values are reserved. This field is valid only if ID_ISAR0.Swap is 0b0000.

In Armv8-A, the only permitted value is 0b0000.

**PSR_M, bits [27:24]**

Indicates the implemented M-profile instructions to modify the PSRs. Defined values are:

- 0b0000: None implemented.
- 0b0001: Adds the M-profile forms of the CPS, MRS, and MSR instructions.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

**SynchPrim_frac, bits [23:20]**

Used in conjunction with ID_ISAR3.SynchPrim to indicate the implemented Synchronization Primitive instructions. Possible values are:

- 0b0000: If SynchPrim == 0b0000, no Synchronization Primitives implemented. If SynchPrim == 0b0001, adds the LDREX and STREX instructions. If SynchPrim == 0b0010, also adds the CLREX, LDREXB, LDREXH, STREX, STREXB, STREXH, LDREXD, and STREXD instructions.
- 0b0011: If SynchPrim == 0b0011, adds the LDREX, STREX, CLREX, LDREXB, LDREXH, STREXB, and STREXH instructions.
All other combinations of SynchPrim and SynchPrim_frc are reserved.  
In Armv8-A, the only permitted value is 0b0000.

**Barrier, bits [19:16]**

Indicates the implemented Barrier instructions in the A32 and T32 instruction sets. Defined values are:
- 0b0000: None implemented. Barrier operations are provided only as System instructions in the (coproc==0b1111) encoding space.
- 0b0001: Adds the DMB, DSB, and ISB barrier instructions.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

**SMC, bits [15:12]**

Indicates the implemented SMC instructions. Defined values are:
- 0b0000: None implemented.
- 0b0001: Adds the SMC instruction.

All other values are reserved.
In Armv8-A, the permitted values are:
- If EL3 is implemented, the only permitted value is 0b0001.
- If neither EL3 nor EL2 is implemented, the only permitted value is 0b0000.

**Writeback, bits [11:8]**

Indicates the support for Writeback addressing modes. Defined values are:
- 0b0000: Basic support. Only the LDM, STM, PUSH, POP, SRS, and RFE instructions support writeback addressing modes. These instructions support all of their writeback addressing modes.
- 0b0001: Adds support for all of the writeback addressing modes.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

**WithShifts, bits [7:4]**

Indicates the support for instructions with shifts. Defined values are:
- 0b0000: Nonzero shifts supported only in MOV and shift instructions.
- 0b0001: Adds support for shifts of loads and stores over the range LSL 0-3.
- 0b0011: As for 0b0001, and adds support for other constant shift options, both on load/store and other instructions.
- 0b0100: As for 0b0011, and adds support for register-controlled shift options.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0100.

**Unpriv, bits [3:0]**

Indicates the implemented unprivileged instructions. Defined values are:
- 0b0000: None implemented. No T variant instructions are implemented.
- 0b0001: Adds the LDRBT, LDRT, STRBT, and STRT instructions.
- 0b0010: As for 0b0001, and adds the LDRHT, LDSRBT, LDRSHT, and STRHT instructions.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0010.
Accessing ID_ISAR4

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC{<c>}{<q>} \coproc, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} \]

\[
\begin{array}{c|c|c|c|c}
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
\hline
0b1111 & 0b000 & 0b0000 & 0b0010 & 0b100 \\
\end{array}
\]

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    R[t] = ID_ISAR4;
  endif
elsif PSTATE.EL == EL2 then
  R[t] = ID_ISAR4;
elsif PSTATE.EL == EL3 then
  R[t] = ID_ISAR4;
G8.2.90  ID_ISAR5, Instruction Set Attribute Register 5

The ID_ISAR5 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.
Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, and ID_ISAR4.
For general information about the interpretation of the ID registers see Principles of the ID scheme for fields in ID registers on page G8-9332.

Configurations

AArch32 System register ID_ISAR5 bits [31:0] are architecturally mapped to AArch64 System register ID_ISAR5_EL1[31:0].
This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_ISAR5 are UNDEFINED.

Attributes

ID_ISAR5 is a 32-bit register.

Field descriptions

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| VCMA | RDM | RES0 | CRC32 | SHA2 | SHA1 | AES | SEVL |

VCMA, bits [31:28]

Indicates AArch32 support for complex number addition and multiplication where numbers are stored in vectors. Defined values are:

0b0000  The VCMLA and VCADD instructions are not implemented in AArch32.
0b0001  The VCMLA and VCADD instructions are implemented in AArch32.
All other values are reserved.
FEAT_FCMA implements the functionality identified by 0b0001.
From Armv8.3, the only permitted value is 0b0001.

RDM, bits [27:24]

Indicates support for the VQRDMLAH and VQRDMLSH instructions in AArch32 state. Defined values are:

0b0000  No VQRDMLAH and VQRDMLSH instructions implemented.
0b0001  VQRDMLAH and VQRDMLSH instructions implemented.
All other values are reserved.
FEAT_RDM implements the functionality identified by the value 0b0001.
From Armv8.1, the only permitted value is 0b0001.

Bits [23:20]

Reserved, RES0.

CRC32, bits [19:16]

Indicates support for the CRC32 instructions in AArch32 state. Defined values are:

0b0000  No CRC32 instructions implemented.
0b0001  CRC32B, CRC32H, CRC32W, CRC32CB, CRC32CH, and CRC32CW instructions implemented.
All other values are reserved.
In Armv8.0, the permitted values are 0b0000 and 0b0001.
From Armv8.1, the only permitted value is 0b0001.

**SHA2, bits [15:12]**
Indicates support for the SHA2 instructions in AArch32 state.
- 0b0000  No SHA2 instructions implemented.
- 0b0001  SHA256H, SHA256H2, SHA256SU0, and SHA256SU1 implemented.
All other values are reserved.
In Armv8-A, the permitted values are 0b0000 and 0b0001.

**SHA1, bits [11:8]**
Indicates support for the SHA1 instructions are implemented in AArch32 state. Defined values are:
- 0b0000  No SHA1 instructions implemented.
- 0b0001  SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 implemented.
All other values are reserved.
In Armv8-A, the permitted values are 0b0000 and 0b0010.

**AES, bits [7:4]**
Indicates support for the AES instructions in AArch32 state. Defined values are:
- 0b0000  No AES instructions implemented.
- 0b0001  AESE, AESD, AESMC, and AESIMC implemented.
- 0b0010  As for 0b0001, plus VMULL (polynomial) instructions operating on 64-bit data quantities.
All other values are reserved.
In Armv8-A, the permitted values are 0b0000 and 0b0010.

**SEVL, bits [3:0]**
Indicates support for the SEVL instruction in AArch32 state. Defined values are:
- 0b0000  SEVL is implemented as a NOP.
- 0b0001  SEVL is implemented as Send Event Local.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

**Accessing ID_ISAR5**
Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{c\}\{q\} \coproc, \{#\}opc1, \langle Rt \rangle, \langle CRn \rangle, \langle CRm \rangle, \{#\}opc2
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
  AArch32.TakeHypTrapException(0x03);
else
  R[t] = ID_ISAR5;
elsif PSTATE_EL == EL2 then
  R[t] = ID_ISAR5;
elsif PSTATE_EL == EL3 then
  R[t] = ID_ISAR5;
G8.2.91  ID_ISAR6, Instruction Set Attribute Register 6

The ID_ISAR6 characteristics are:

**Purpose**

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register ID_ISAR6 bits [31:0] are architecturally mapped to AArch64 System register ID_ISAR6_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_ISAR6 are UNDEFINED.

--- **Note** ---

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

**Attributes**

ID_ISAR6 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Field Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-28</td>
<td>RES0</td>
</tr>
<tr>
<td>27-24</td>
<td>I8MM</td>
</tr>
<tr>
<td>23-20</td>
<td>BF16</td>
</tr>
<tr>
<td>19-16</td>
<td>SPECRES</td>
</tr>
<tr>
<td>15-12</td>
<td>SB</td>
</tr>
<tr>
<td>11-8</td>
<td>FHM</td>
</tr>
<tr>
<td>7-4</td>
<td>DP</td>
</tr>
<tr>
<td>3-0</td>
<td>JSCVT</td>
</tr>
</tbody>
</table>

**Bits [31:28]**

Reserved, RES0.

**I8MM, bits [27:24]**

Indicates support for Advanced SIMD and floating-point Int8 matrix multiplication instructions in AArch32 state. Defined values are:

- 0b0000: Int8 matrix multiplication instructions are not implemented.
- 0b0001: VSMMLA, VSUDOT, VUMMLA, VUSMMLA, and VUSDOT instructions are implemented.

All other values are reserved.

FEAT_AA32I8MM implements the functionality identified by 0b0001.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

**BF16, bits [23:20]**

Indicates support for Advanced SIMD and floating-point BFloat16 instructions in AArch32 state. Defined values are:

- 0b0000: BFloat16 instructions are not implemented.
- 0b0001: VCVT, VCVTB, VCVTT, VDOT, VFMA, VFMA, and VMLA instructions with BFloat16 operand or result types are implemented.

All other values are reserved.

FEAT_AA32BF16 implements the functionality identified by 0b0001.
From Armv8.2, the permitted values are 0b0000 and 0b0001.

**SPECRES, bits [19:16]**
Indicates support for Speculation invalidation instructions in AArch32 state. Defined values are:
- 0b0000  CFPRCTX, DVPRCTX, and CPPRCTX instructions are not implemented.
- 0b0001  CFPRCTX, DVPRCTX, and CPPRCTX instructions are implemented.
All other values are reserved.
From Armv8.5, the only permitted value is 0b0001.

**SB, bits [15:12]**
Indicates support for SB instruction in AArch32 state. Defined values are:
- 0b0000  SB instruction is not implemented.
- 0b0001  SB instruction is implemented.
All other values are reserved.
From Armv8.5, the only permitted value is 0b0001.

**FHM, bits [11:8]**
Indicates support for Advanced SIMD and floating-point VFMAL and VFMSL instructions in AArch32 state. Defined values are:
- 0b0000  VFMAL and VMFSL instructions not implemented.
- 0b0001  VFMAL and VMFSL instructions implemented.
FEAT_FHM implements the functionality identified by the value 0b0001.

**DP, bits [7:4]**
Indicates support for dot product instructions in AArch32 state. Defined values are:
- 0b0000  No dot product instructions implemented.
- 0b0001  VUDOT and VSDOT instructions implemented.
All other values are reserved.
FEAT_DotProd implements the functionality identified by the value 0b0001.

**JSCVT, bits [3:0]**
Indicates support for the Javascript conversion instruction in AArch32 state. Defined values are:
- 0b0000  The VJCVT instruction is not implemented.
- 0b0001  The VJCVT instruction is implemented.
All other values are reserved.
In Armv8.0, the only permitted value is 0b0000.
FEAT_JSCVT implements the functionality identified by 0b0001.
From Armv8.3, if Advanced SIMD or Floating-point is implemented, the only permitted value is 0b0001.
From Armv8.3, if Advanced SIMD or Floating-point is not implemented, the only permitted value is 0b0000.

**Accessing ID_ISAR6**
Accesses to this register use the following encodings in the System register encoding space:
# MRC{<c>}{<q>} <coproc>, {#<opc1>, <Rt>, <CRn>, <CRm>{, {#<opc2>}

\[
\begin{array}{cccccc}
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
0b1111 & 0b000 & 0b0000 & 0b0010 & 0b111
\end{array}
\]

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_ISAR6) ||
        boolean IMPLEMENTATION_DEFINED "ID_ISAR6 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_ISAR6) ||
        boolean IMPLEMENTATION_DEFINED "ID_ISAR6 trapped by HCR.TID3") && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        R[t] = ID_ISAR6;
    elsif PSTATE.EL == EL2 then
        R[t] = ID_ISAR6;
    elsif PSTATE.EL == EL3 then
        R[t] = ID_ISAR6;
    \end{if}

G8.2.92  ID_MMFR0, Memory Model Feature Register 0

The ID_MMFR0 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register ID_MMFR0 bits [31:0] are architecturally mapped to AArch64 System register ID_MMFR0_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_MMFR0 are UNDEFINED.

**Attributes**

ID_MMFR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field Description</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>InnerShr</td>
<td>[31:28]</td>
</tr>
<tr>
<td>FCSE</td>
<td>[27:24]</td>
</tr>
<tr>
<td>AuxReg</td>
<td>[23:20]</td>
</tr>
<tr>
<td>TCM</td>
<td>[19:16]</td>
</tr>
<tr>
<td>ShareLvl</td>
<td>[15:12]</td>
</tr>
<tr>
<td>OuterShr</td>
<td>[11:8]</td>
</tr>
<tr>
<td>PMSA</td>
<td>[7:4]</td>
</tr>
<tr>
<td>VMSA</td>
<td>[3:0]</td>
</tr>
</tbody>
</table>

**InnerShr, bits [31:28]**

Innermost Shareability. Indicates the innermost shareability domain implemented. Defined values are:

- **0b0000**: Implemented as Non-cacheable.
- **0b0001**: Implemented with hardware coherency support.
- **0b1111**: Shareability ignored.
- All other values are reserved.

In Armv8-A, the permitted values are **0b0000**, **0b0001**, and **0b1111**.

This field is valid only if the implementation supports two levels of shareability, as indicated by ID_MMFR0.ShareLvl having the value **0b0001**.

When ID_MMFR0.ShareLvl is zero, this field is **UNKNOWN**.

**FCSE, bits [27:24]**

Indicates whether the implementation includes the FCSE. Defined values are:

- **0b0000**: Not supported.
- **0b0001**: Support for FCSE.
- All other values are reserved.

In Armv8-A, the only permitted value is **0b0000**.

**AuxReg, bits [23:20]**

Auxiliary Registers. Indicates support for Auxiliary registers. Defined values are:

- **0b0000**: None supported.
- **0b0001**: Support for Auxiliary Control Register only.
- **0b0010**: Support for Auxiliary Fault Status Registers (AIFSR and ADFSR) and Auxiliary Control Register.
- All other values are reserved.
In Armv8-A, the only permitted value is 0b0010.

--- Note ---
Accesses to unimplemented Auxiliary registers are UNDEFINED.

TCM, bits [19:16]
Indicates support for TCMs and associated DMAs. Defined values are:
0b0000  Not supported.
0b0001  Support is IMPLEMENTATION DEFINED.
0b0010  Support for TCM only, Armv6 implementation.
0b0011  Support for TCM and DMA, Armv6 implementation.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

ShareLvl, bits [15:12]
Shareability Levels. Indicates the number of shareability levels implemented. Defined values are:
0b0000  One level of shareability implemented.
0b0001  Two levels of shareability implemented.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

OuterShr, bits [11:8]
Outermost Shareability. Indicates the outermost shareability domain implemented. Defined values are:
0b0000  Implemented as Non-cacheable.
0b0001  Implemented with hardware coherency support.
0b1111  Shareability ignored.
All other values are reserved.
In Armv8-A, the permitted values are 0b0000, 0b0001, and 0b1111.

PMSA, bits [7:4]
Indicates support for a PMSA. Defined values are:
0b0000  Not supported.
0b0001  Support for IMPLEMENTATION DEFINED PMSA.
0b0010  Support for PMSAv6, with a Cache Type Register implemented.
0b0011  Support for PMSAv7, with support for memory subsections. Armv7-R profile.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

VMSA, bits [3:0]
Indicates support for a VMSA. Defined values are:
0b0000  Not supported.
0b0001  Support for IMPLEMENTATION DEFINED VMSA.
0b0010  Support for VMSAv6, with Cache and TLB Type Registers implemented.
0b0011  Support for VMSAv7, with support for remapping and the Access flag. Armv7-A profile.
0b0100  As for 0b0011, and adds support for the PXN bit in the Short-descriptor translation table format descriptors.
As for 0b0100, and adds support for the Long-descriptor translation table format. All other values are reserved.

In Armv8-A, the only permitted value is 0b0101.

**Accessing ID_MMFR0**

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    R[t] = ID_MMFR0;
  elsif PSTATE.EL == EL2 then
    R[t] = ID_MMFR0;
  elsif PSTATE.EL == EL3 then
    R[t] = ID_MMFR0;
```
G8.2.93  ID_MMFR1, Memory Model Feature Register 1

The ID_MMFR1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see Principles of the ID scheme for fields in ID registers on page G8-9332.

Configurations

AArch32 System register ID_MMFR1 bits [31:0] are architecturally mapped to AArch64 System register ID_MMFR1_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_MMFR1 are UNDEFINED.

Attributes

ID_MMFR1 is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPred</td>
<td>L1TstCln</td>
<td>L1Uni</td>
<td>L1Hvd</td>
<td>L1UniSW</td>
<td>L1HvdSW</td>
<td>L1UniVA</td>
<td>L1HvdVA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BPred, bits [31:28]

Branch Predictor. Indicates branch predictor management requirements. Defined values are:

0b0000  No branch predictor, or no MMU present. Implies a fixed MPU configuration.

0b0001  Branch predictor requires flushing on:
                          • Enabling or disabling a stage of address translation.
                          • Writing new data to instruction locations.
                          • Writing new mappings to the translation tables.
                          • Changes to the TTBR0, TTBR1, or TTBCR registers.
                          • Changes to the ContextID or ASID, or to the FCSE ProcessID if this is supported.

0b0100  Branch predictor requires flushing on:
                          • Enabling or disabling a stage of address translation.
                          • Writing new data to instruction locations.
                          • Writing new mappings to the translation tables.
                          • Any change to the TTBR0, TTBR1, or TTBCR registers without a change to the corresponding ContextID or ASID, or FCSE ProcessID if this is supported.

0b0111  Branch predictor requires flushing only on writing new data to instruction locations.

0b0100  For execution correctness, branch predictor requires no flushing at any time.

All other values are reserved.

In Armv8-A, the permitted values are 0b0010, 0b0011, or 0b0100. For values other than 0b0000 and 0b0100, the Arm Architecture Reference Manual, or the product documentation, might give more information about the required maintenance.
L1TstCln, bits [27:24]
Level 1 cache Test and Clean. Indicates the supported Level 1 data cache test and clean operations, for Harvard or unified cache implementations. Defined values are:

0b0000  None supported.
0b0001  Supported Level 1 data cache test and clean operations are:
       •  Test and clean data cache.
0b0010  As for 0b0001, and adds:
       •  Test, clean, and invalidate data cache.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

L1Uni, bits [23:20]
Level 1 Unified cache. Indicates the supported entire Level 1 cache maintenance operations for a unified cache implementation. Defined values are:

0b0000  None supported.
0b0001  Supported entire Level 1 cache operations are:
       •  Invalidate cache, including branch predictor if appropriate.
       •  Invalidate branch predictor, if appropriate.
0b0010  As for 0b0001, and adds:
       •  Clean cache, using a recursive model that uses the cache dirty status bit.
       •  Clean and invalidate cache, using a recursive model that uses the cache dirty status bit.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

L1Hvd, bits [19:16]
Level 1 Harvard cache. Indicates the supported entire Level 1 cache maintenance operations for a Harvard cache implementation. Defined values are:

0b0000  None supported.
0b0001  Supported entire Level 1 cache operations are:
       •  Invalidate instruction cache, including branch predictor if appropriate.
       •  Invalidate branch predictor, if appropriate.
0b0010  As for 0b0001, and adds:
       •  Invalidate data cache.
       •  Invalidate data cache and instruction cache, including branch predictor if appropriate.
0b0011  As for 0b0010, and adds:
       •  Clean data cache, using a recursive model that uses the cache dirty status bit.
       •  Clean and invalidate data cache, using a recursive model that uses the cache dirty status bit.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

L1UniSW, bits [15:12]
Level 1 Unified cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a unified cache implementation. Defined values are:

0b0000  None supported.
0b0001  Supported Level 1 unified cache line maintenance operations by set/way are:
  • Clean cache line by set/way.

0b0010  As for 0b0001, and adds:
  • Clean and invalidate cache line by set/way.

0b0011  As for 0b0010, and adds:
  • Invalidate cache line by set/way.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

L1HvdSW, bits [11:8]
Level 1 Harvard cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a Harvard cache implementation. Defined values are:
0b0000  None supported.
0b0001  Supported Level 1 Harvard cache line maintenance operations by set/way are:
  • Clean data cache line by set/way.
  • Clean and invalidate data cache line by set/way.

0b0010  As for 0b0001, and adds:
  • Invalidate data cache line by set/way.

0b0011  As for 0b0010, and adds:
  • Invalidate instruction cache line by set/way.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

L1UniVA, bits [7:4]
Level 1 Unified cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a unified cache implementation. Defined values are:
0b0000  None supported.
0b0001  Supported Level 1 unified cache line maintenance operations by VA are:
  • Clean cache line by VA.
  • Invalidate cache line by VA.
  • Clean and invalidate cache line by VA.

0b0010  As for 0b0001, and adds:
  • Invalidate branch predictor by VA, if branch predictor is implemented.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

L1HvdVA, bits [3:0]
Level 1 Harvard cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a Harvard cache implementation. Defined values are:
0b0000  None supported.
0b0001  Supported Level 1 Harvard cache line maintenance operations by VA are:
  • Clean data cache line by VA.
  • Invalidate data cache line by VA.
  • Clean and invalidate data cache line by VA.
  • Clean instruction cache line by VA.

0b0010  As for 0b0001, and adds:
  • Invalidate branch predictor by VA, if branch predictor is implemented.
All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

### Accessing ID_MMFR1

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} <\text{coproc}>, \{#<\text{opc1}>\}, \text{<Rt>}, \text{<CRn>}, \text{<CRm>}, \{#<\text{opc2}>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b101</td>
</tr>
</tbody>
</table>

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        R[t] = ID_MMFR1;
    end
elsif PSTATE.EL == EL2 then
    R[t] = ID_MMFR1;
elsif PSTATE.EL == EL3 then
    R[t] = ID_MMFR1;
```
G8.2.94 ID_MMFR2, Memory Model Feature Register 2

The ID_MMFR2 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register ID_MMFR2 bits [31:0] are architecturally mapped to AArch64 System register ID_MMFR2_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_MMFR2 are UNDEFINED.

**Attributes**

ID_MMFR2 is a 32-bit register.

**Field descriptions**

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| HWAccFlg | WFIStall | MemBarr | UniTLB | HvdTLB | L1HvdRng | L1HvdBG | L1HvdFG |

**HWAccFlg, bits [31:28]**

Hardware Access Flag. In earlier versions of the Arm Architecture, this field indicates support for a Hardware Access flag, as part of the VMSAv7 implementation. Defined values are:

- **0b0000** Not supported.
- **0b0001** Support for VMSAv7 Access flag, updated in hardware.

All other values are reserved.

In Armv8-A, the only permitted value is **0b0000**.

**WFIStall, bits [27:24]**

Wait For Interrupt Stall. Indicates the support for Wait For Interrupt (WFI) stalling. Defined values are:

- **0b0000** Not supported.
- **0b0001** Support for WFI stalling.

All other values are reserved.

In Armv8-A, the permitted values are **0b0000** and **0b0001**.

**MemBarr, bits [23:20]**

Memory Barrier. Indicates the supported memory barrier System instructions in the (coproc == 1111) encoding space. Defined values are:

- **0b0000** None supported.
- **0b0001** Supported memory barrier System instructions are:
  - Data Synchronization Barrier (DSB).
- **0b0010** As for 0b0001, and adds:
  - Instruction Synchronization Barrier (ISB).
  - Data Memory Barrier (DMB).

All other values are reserved.
In Armv8-A, the only permitted value is 0b0010.
Arm deprecates the use of these operations. ID_ISAR4.Barrier_instrs indicates the level of support for the preferred barrier instructions.

**UniTLB, bits [19:16]**

Unified TLB. Indicates the supported TLB maintenance operations, for a unified TLB implementation. Defined values are:

- **0b0000** Not supported.
- **0b0001** Supported unified TLB maintenance operations are:
  - Invalidate all entries in the TLB.
  - Invalidate TLB entry by VA.
- **0b0010** As for 0b0001, and adds:
  - Invalidate TLB entries by ASID match.
- **0b0011** As for 0b0010, and adds:
  - Invalidate instruction TLB and data TLB entries by VA All ASID. This is a shared unified TLB operation
- **0b0100** As for 0b0011, and adds:
  - Invalidate Hyp mode unified TLB entry by VA.
  - Invalidate entire Non-secure PL1&0 unified TLB.
  - Invalidate entire Hyp mode unified TLB.
- **0b0101** As for 0b0100, and adds the following operations: TLBIMVALIS, TLBIMVAALIS, TLBIMVALHIS, TLBIMVAL, TLBIMVAAL,TLBIMVALH.
- **0b0110** As for 0b0101, and adds the following operations: TLBIIIPAS2IS, TLBIIIPAS2LIS, TLBIIIPAS2, TLBIIIPAS2L.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0110.

**HvdTLB, bits [15:12]**

If the value of ID_MMFR2.UniTLB is not 0b0000, then the meaning of this field is IMPLEMENTATION DEFINED. Arm deprecates the use of this field by software.

**L1HvdRng, bits [11:8]**

Level 1 Harvard cache Range. Indicates the supported Level 1 cache maintenance range operations, for a Harvard cache implementation. Defined values are:

- **0b0000** Not supported.
- **0b0001** Supported Level 1 Harvard cache maintenance range operations are:
  - Invalidate data cache range by VA.
  - Invalidate instruction cache range by VA.
  - Clean data cache range by VA.
  - Clean and invalidate data cache range by VA.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

**L1HvdBG, bits [7:4]**

Level 1 Harvard cache Background fetch. Indicates the supported Level 1 cache background fetch operations, for a Harvard cache implementation. When supported, background fetch operations are non-blocking operations. Defined values are:

- **0b0000** Not supported.
Supported Level 1 Harvard cache background fetch operations are:
- Fetch instruction cache range by VA.
- Fetch data cache range by VA.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

**L1HvdFG, bits [3:0]**
Level 1 Harvard cache Foreground fetch. Indicates the supported Level 1 cache foreground fetch operations, for a Harvard cache implementation. When supported, foreground fetch operations are blocking operations. Defined values are:
- 0b0000 Not supported.
- 0b0001 Supported Level 1 Harvard cache foreground fetch operations are:
  - Fetch instruction cache range by VA.
  - Fetch data cache range by VA.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

**Accessing ID_MMFR2**
Accesses to this register use the following encodings in the System register encoding space:

$$MRC{<c>}{<q>}{<coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}}$$

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & ELUsingAArch32(EL2) & HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.TID3 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR.TID3 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    R[t] = ID_MMFR2;
elsif PSTATE.EL == EL2 then
  R[t] = ID_MMFR2;
elsif PSTATE.EL == EL3 then
  R[t] = ID_MMFR2;
G8.2.95  ID_MMFR3, Memory Model Feature Register 3

The ID_MMFR3 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see Principles of the ID scheme for fields in ID registers on page G8-9332.

**Configurations**

AArch32 System register ID_MMFR3 bits [31:0] are architecturally mapped to AArch64 System register ID_MMFR3_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_MMFR3 are UNDEFINED.

**Attributes**

ID_MMFR3 is a 32-bit register.

**Field descriptions**

### Supersec, bits [31:28]

Supersections. On a VMSA implementation, indicates whether Supersections are supported.

Defined values are:

- **0b0000**  Supersections supported.
- **0b1111**  Supersections not supported.

All other values are reserved.

In Armv8-A, the permitted values are **0b0000** and **0b1111**.

### CMemSz, bits [27:24]

Cached Memory Size. Indicates the physical memory size supported by the caches. Defined values are:

- **0b0000**  4GB, corresponding to a 32-bit physical address range.
- **0b0001**  64GB, corresponding to a 36-bit physical address range.
- **0b0010**  1TB or more, corresponding to a 40-bit or larger physical address range.

All other values are reserved.

In Armv8-A, the permitted values are **0b0000**, **0b0001**, and **0b0010**.

### CohWalk, bits [23:20]

Coherent Walk. Indicates whether Translation table updates require a clean to the Point of Unification. Defined values are:

- **0b0000**  Updates to the translation tables require a clean to the Point of Unification to ensure visibility by subsequent translation table walks.
- **0b0001**  Updates to the translation tables do not require a clean to the Point of Unification to ensure visibility by subsequent translation table walks.

All other values are reserved.

In Armv8-A, the only permitted value is **0b0001**.
PAN, bits [19:16]

Privileged Access Never. Indicates support for the PAN bit in CPSR, SPSR, and DSPSR in AArch32 state. Defined values are:

- 0b0000  PAN not supported.
- 0b0001  PAN supported.
- 0b0010  PAN supported and ATS1CPRP and ATS1CPWP instructions supported.

All other values are reserved.

FEAT_PAN implements the functionality identified by the value 0b0001.

FEAT_PAN2 implements the functionality added by the value 0b0010.

In Armv8.1, the value 0b0000 is not permitted.

From Armv8.2, the only permitted value is 0b0010.

MaintBest, bits [15:12]

Maintenance Broadcast. Indicates whether Cache, TLB, and branch predictor operations are broadcast. Defined values are:

- 0b0000  Cache, TLB, and branch predictor operations only affect local structures.
- 0b0001  Cache and branch predictor operations affect structures according to shareability and defined behavior of instructions. TLB operations only affect local structures.
- 0b0010  Cache, TLB, and branch predictor operations affect structures according to shareability and defined behavior of instructions.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0010.

BPMaint, bits [11:8]

Branch Predictor Maintenance. Indicates the supported branch predictor maintenance operations in an implementation with hierarchical cache maintenance operations. Defined values are:

- 0b0000  None supported.
- 0b0001  Supported branch predictor maintenance operations are:
  - Invalidate all branch predictors.
- 0b0010  As for 0b0001, and adds:
  - Invalidate branch predictors by VA.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0010.

CMaintSW, bits [7:4]

Cache Maintenance by Set/Way. Indicates the supported cache maintenance operations by set/way, in an implementation with hierarchical caches. Defined values are:

- 0b0000  None supported.
- 0b0001  Supported hierarchical cache maintenance instructions by set/way are:
  - Invalidate data cache by set/way.
  - Clean data cache by set/way.
  - Clean and invalidate data cache by set/way.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

In a unified cache implementation, the data cache maintenance operations apply to the unified caches.
CMaintVA, bits [3:0]

Cache Maintenance by Virtual Address. Indicates the supported cache maintenance operations by VA, in an implementation with hierarchical caches. Defined values are:

0b0000  None supported.
0b0001  Supported hierarchical cache maintenance operations by VA are:
    • Invalidate data cache by VA.
    • Clean data cache by VA.
    • Clean and invalidate data cache by VA.
    • Invalidate instruction cache by VA.
    • Invalidate all instruction cache entries.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.
In a unified cache implementation, data cache maintenance operations apply to the unified caches, and the instruction cache maintenance instructions are not implemented.

Accessing ID_MMFR3

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} \ <coproc>, \{#\}<opc1>, \ <Rt>, \ <CRn>, \ <CRm>\{, \{#\}<opc2>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        R[t] = ID_MMFR3;
    endif
elsif PSTATE_EL == EL2 then
    R[t] = ID_MMFR3;
elsif PSTATE_EL == EL3 then
    R[t] = ID_MMFR3;
G8.2.96  ID_MMFR4, Memory Model Feature Register 4

The ID_MMFR4 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register ID_MMFR4 bits [31:0] are architecturally mapped to AArch64 System register ID_MMFR4_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_MMFR4 are UNDEFINED.

**Attributes**

ID_MMFR4 is a 32-bit register.

**Field descriptions**

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| EVT | CCIDX | LSM | HPDS | CnP | XNX | AC2 | SpecSEI |

**EVT, bits [31:28]**

Enhanced Virtualization Traps. If EL2 is implemented, indicates support for the HCR2.{TTLBIS, TOCU, TICAB, TID4} traps. Defined values are:

- **0b0000**: HCR2.{TTLBIS, TOCU, TICAB, TID4} traps are not supported.
- **0b0001**: HCR2.{TOCU, TICAB, TID4} traps are supported. HCR2.TTLBIS trap is not supported.
- **0b0010**: HCR2.{TTLBIS, TOCU, TICAB, TID4} traps are supported.

All other values are reserved.

**FEAT_EVT** implements the functionality identified by the values **0b0001** and **0b0010**.

If EL2 is not implemented supporting AArch32, the only permitted value is **0b0000**.

In Armv8.2, the permitted values are **0b0000**, **0b0001**, and **0b0010**.

From Armv8.5, the permitted values are:

- **0b0000** when EL2 is not implemented or does not support AArch32.
- **0b0010** when EL2 is implemented and supports AArch32.

**CCIDX, bits [27:24]**

Support for use of the revised CCSIDR format and the presence of the CCSIDR2 is indicated. Defined values are:

- **0b0000**: 32-bit format implemented for all levels of the CCSIDR, and the CCSIDR2 register is not implemented.
- **0b0001**: 64-bit format implemented for all levels of the CCSIDR, and the CCSIDR2 register is implemented.

All other values are reserved.

**FEAT_CCIDX** implements the functionality identified by **0b0001**.

From Armv8.3, the permitted values are **0b0000** and **0b0001**.
LSM, bits [23:20]
Indicates support for LSMAOE and nTLSMD bits in HSCTLR and SCTLR. Defined values are:

- **0b0000**: LSMAOE and nTLSMD bits not supported.
- **0b0001**: LSMAOE and nTLSMD bits supported.

All other values are reserved.

**FEAT_LSMAOC** implements the functionality identified by the value **0b0001**.

From Armv8.2, the permitted values are **0b0000** and **0b0001**.

HPDS, bits [19:16]
Hierarchical permission disables bits in translation tables. Defined values are:

- **0b0000**: Disabling of hierarchical controls not supported.
- **0b0001**: Supports disabling of hierarchical controls using the TTBCR2.HPD0, TTBCR2.HPD1, and HTCR.HPD bits.
- **0b0100**: As for value **0b0001**, and adds possible hardware allocation of bits[62:59] of the Translation table descriptors from the final lookup level for IMPLEMENTATION DEFINED use.

All other values are reserved.

**FEAT_AA32HPD** implements the functionality identified by the value **0b0001**.

**FEAT_HPDS2** implements the functionality added by the value **0b0100**.

--- **Note**

The value **0b0000** implies that the encoding for TTBCR2 is UNDEFINED.

---

CnP, bits [15:12]
Common not Private translations. Defined values are:

- **0b0000**: Common not Private translations not supported.
- **0b0001**: Common not Private translations supported.

All other values are reserved.

**FEAT_TTCNP** implements the functionality identified by the value **0b0001**.

From Armv8.2, the only permitted value is **0b0001**.

XNX, bits [11:8]
Support for execute-never control distinction by Exception level at stage 2. Defined values are:

- **0b0000**: Distinction between EL0 and EL1 execute-never control at stage 2 not supported.
- **0b0001**: Distinction between EL0 and EL1 execute-never control at stage 2 supported.

All other values are reserved.

**FEAT_XNX** implements the functionality identified by the value **0b0001**.

When **FEAT_XNX** is implemented:

- If all of the following conditions are true, it is IMPLEMENTATION DEFINED whether the value of ID_MMFR4.XNX is **0b0000** or **0b0001**:
  - **ID_AA64MMFR1_EL1.XNX == 1**.
  - EL2 cannot use AArch32.
  - EL1 can use AArch32.
- If EL2 can use AArch32 then the only permitted value is **0b0001**.
AC2, bits [7:4]

Indicates the extension of the ACTLR and HACTLR registers using ACTLR2 and HACTLR2.
Defined values are:

0b0000  ACTLR2 and HACTLR2 are not implemented.
0b0001  ACTLR2 and HACTLR2 are implemented.

All other values are reserved.
In Armv8.0, the permitted values are 0b0000 and 0b0001.
From Armv8.2, the only permitted value is 0b0001.

SpecSEI, bits [3:0]

When FEAT_RAS is implemented:

- Describes whether the PE can generate SError interrupt exceptions from speculative reads of memory, including speculative instruction fetches.

  0b0000  The PE never generates an SError interrupt due to an External abort on a speculative read.

  0b0001  The PE might generate an SError interrupt due to an External abort on a speculative read.

All other values are reserved.

Otherwise:

- Reserved, RES0.

Accessing ID_MMFR4

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC} \text{<c>}{<q}> \text{<coproc>, (#}<opc1>, <Rt>, <CRn>, <CRm>{, (#}<opc2>}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAAArch32(EL2) && HSTR_T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_MMFR4) || boolean IMPLEMENTATION_DEFINED "ID_MMFR4 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAAArch32(EL2) && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_MMFR4) || boolean IMPLEMENTATION_DEFINED "ID_MMFR4 trapped by HCR.TID3") && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        R[t] = ID_MMFR4;
    end
elsif PSTATE.EL == EL2 then
    R[t] = ID_MMFR4;
elsif PSTATE.EL == EL3 then
    R[t] = ID_MMFR4;
### G8.2.97 ID_MMFR5, Memory Model Feature Register 5

The ID_MMFR5 characteristics are:

**Purpose**

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register ID_MMFR5 bits [31:0] are architecturally mapped to AArch64 System register ID_MMFR5_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_MMFR5 are UNDEFINED.

**Attributes**

ID_MMFR5 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0</td>
</tr>
<tr>
<td>8−31</td>
<td>nTLBPA</td>
</tr>
<tr>
<td>0−7</td>
<td>ETS</td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**nTLBPA, bits [7:4]**

Indicates support for intermediate caching of translation table walks. Defined values are:

- 0b0000: The intermediate caching of translation table walks might include non-coherent physical translation caches.
- 0b0001: The intermediate caching of translation table walks does not include non-coherent physical translation caches.

Non-coherent physical translation caches are non-coherent caches of previous valid translation table entries since the last completed relevant TLBI applicable to the PE, where either:

- The caching is indexed by the physical address of the location holding the translation table entry.
- The caching is used for stage 1 translations and is indexed by the intermediate physical address of the location holding the translation table entry.

All other values are reserved.

**ETS, bits [3:0]**

Indicates support for Enhanced Translation Synchronization. Defined values are:

- 0b0000: Enhanced Translation Synchronization is not supported.
- 0b0001: Enhanced Translation Synchronization is supported.

All other values are reserved.

From Armv8.0, the permitted values are 0b0000 and 0b0001.
From Armv8.7, the only permitted value is 0b0001.

**Accessing ID_MMFR5**

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC\{<c>\}{<q>} \{coproc\}, \{#\}<opc1>, <Rt>, <CRn>, <CRm>{, \{#\}<opc2>} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0011</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_MMFR5) ||
    boolean IMPLEMENTATION_DEFINED "ID_MMFR5 trapped by HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_MMFR5) ||
    boolean IMPLEMENTATION_DEFINED "ID_MMFR5 trapped by HCR.TID3") && HCR.TID3 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    R[t] = ID_MMFR5;
  elsif PSTATE.EL == EL2 then
    R[t] = ID_MMFR5;
  elsif PSTATE.EL == EL3 then
    R[t] = ID_MMFR5;
else
  R[t] = ID_MMFR5;
G8.2.98  ID_PFR0, Processor Feature Register 0

The ID_PFR0 characteristics are:

**Purpose**

Gives top-level information about the instruction sets and other features supported by the PE in AArch32 state.

Must be interpreted with ID_PFR1.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register ID_PFR0 bits [31:0] are architecturally mapped to AArch64 System register ID_PFR0_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_PFR0 are UNDEFINED.

**Attributes**

ID_PFR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>24</th>
<th>23</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>DIT</td>
<td>AMU</td>
<td>CSV2</td>
<td>State3</td>
<td>State2</td>
<td>State1</td>
<td>State0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RAS, bits [31:28]**

RAS Extension version. Defined values are:

- 0b0000  No RAS Extension.
- 0b0001  RAS Extension implemented.
- 0b0010  FEAT_RASv1p1 implemented. As 0b0001, and adds support for additional ERXMISC<m> System registers.

Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS and support for the optional RAS Timestamp Extension.

All other values are reserved.

FEAT_RAS implements the functionality identified by the value 0b0001.

FEAT_RASv1p1 implements the functionality identified by the value 0b0010.

In Armv8.0 and Armv8.1, the permitted values are 0b0000 and 0b0001.

In Armv8.2, the only permitted value is 0b0001.

From Armv8.4, if FEAT_DoubleFault is implemented, the only permitted value is 0b0010.

From Armv8.4, when FEAT_DoubleFault is not implemented, and ERRIDR.NUM is 0, the permitted values are IMPLEMENTATION DEFINED 0b0001 or 0b010.

**Note**

When the value of this field is 0b0001, ID_PFR2.RAS_frac indicates whether FEAT_RASv1p1 is implemented.

**DIT, bits [27:24]**

Data Independent Timing. Defined values are:

- 0b0000  AArch32 does not guarantee constant execution time of any instructions.
AArch32 provides the PSTATE.DIT mechanism to guarantee constant execution time of certain instructions.

All other values are reserved.

FEAT_DIT implements the functionality identified by the value 0b0001.

From Armv8.4, the only permitted value is 0b0001.

**AMU, bits [23:20]**

Indicates support for Activity Monitors Extension. Defined values are:

- 0b0000: Activity Monitors Extension is not implemented.
- 0b0001: FEAT_AMUv1 is implemented.
- 0b0010: FEAT_AMUv1p1 is implemented. As 0b0001 and adds support for virtualization of the activity monitor event counters.

All other values are reserved.

FEAT_AMUv1 implements the functionality identified by the value 0b0001.

FEAT_AMUv1p1 implements the functionality identified by the value 0b0010.

In Armv8.0, the only permitted value is 0b0000.

In Armv8.4, the permitted values are 0b0000 and 0b0001.

From Armv8.6, the permitted values are 0b0000, 0b0001, and 0b0010.

**CSV2, bits [19:16]**

Speculative use of out of context branch targets. Defined values are:

- 0b0000: The implementation does not disclose whether FEAT_CSV2 is implemented.
- 0b0001: FEAT_CSV2 is implemented, but FEAT_CSV2_1p1 is not implemented.
- 0b0010: FEAT_CSV2_1p1 is implemented.

All other values are reserved.

FEAT_CSV2 implements the functionality identified by the value 0b0001.

FEAT_CSV2_1p1 implements the functionality identified by the value 0b0010.

From Armv8.5, the permitted values are 0b0001 and 0b0010.

**State3, bits [15:12]**

T32EE instruction set support. Defined values are:

- 0b0000: Not implemented.
- 0b0001: T32EE instruction set implemented.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

**State2, bits [11:8]**

Jazelle extension support. Defined values are:

- 0b0000: Not implemented.
- 0b0001: Jazelle extension implemented, without clearing of JOSCR.CV on exception entry.
- 0b0010: Jazelle extension implemented, with clearing of JOSCR.CV on exception entry.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

**State1, bits [7:4]**

T32 instruction set support. Defined values are:

- 0b0000: T32 instruction set not implemented.
0b0001  T32 encodings before the introduction of Thumb-2 technology implemented:
  • All instructions are 16-bit.
  • A BL or BLX is a pair of 16-bit instructions.
  • 32-bit instructions other than BL and BLX cannot be encoded.

0b0011  T32 encodings after the introduction of Thumb-2 technology implemented, for all
  16-bit and 32-bit T32 basic instructions.

All other values are reserved.
In Armv8-A, the only permitted value is 0b0011.

State0, bits [3:0]
A32 instruction set support. Defined values are:
0b0000  A32 instruction set not implemented.
0b0001  A32 instruction set implemented.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0001.

Accessing ID_PFR0
Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} <\text{coproc}>, \{\#<\text{opc1}>\}, <\text{Rt}>, <\text{CRn}>, <\text{CRm}\{, \{\#<\text{opc2}>\}\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    R[t] = ID_PFR0;
elsif PSTATE.EL == EL2 then
  R[t] = ID_PFR0;
elsif PSTATE.EL == EL3 then
  R[t] = ID_PFR0;
ID_PFR1, Processor Feature Register 1

The ID_PFR1 characteristics are:

**Purpose**

Gives information about the AArch32 programmers' model.

Must be interpreted with ID_PFR0.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register ID_PFR1 bits [31:0] are architecturally mapped to AArch64 System register ID_PFR1_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_PFR1 are UNDEFINED.

**Attributes**

ID_PFR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>GIC</td>
<td>Sec_frac</td>
<td>GenTimer</td>
<td>MPrgMod</td>
<td>Security</td>
<td>ProgMod</td>
<td>GIC</td>
<td>Virt_frac</td>
<td>Virtualization</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**GIC, bits [31:28]**

System register GIC CPU interface. Defined values are:

- 0b0000  GIC CPU interface system registers not implemented.
- 0b0001  System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.
- 0b0011  System register interface to version 4.1 of the GIC CPU interface is supported.

All other values are reserved.

**Virt_frac, bits [27:24]**

Virtualization fractional field. When the Virtualization field is 0b0000, determines the support for Virtualization Extensions. Defined values are:

- 0b0000  No Virtualization Extensions are implemented.
- 0b0001  The following Virtualization Extensions are implemented:
  - The SCR.SIF bit, if EL3 is implemented.
  - The modifications to the SCR.AW and SCR.FW bits described in the Virtualization Extensions, if EL3 is implemented.
  - The MSR (banked register) and MRS (banked register) instructions.
  - The ERET instruction.

All other values are reserved.

In Armv8-A, the permitted values are:

- 0b0000 when EL2 is implemented.
- 0b0001 when EL2 is not implemented.

This field is valid only when the value of ID_PFR1.Virtualization is 0, otherwise it holds the value 0b0000.
Note
The ID_ISAR registers do not identify whether the instructions added by the Virtualization Extensions are implemented.

Sec_frac, bits [23:20]
Security fractional field. When the Security field is 0b0000, determines the support for Security Extensions. Defined values are:

- 0b0000: No Security Extensions are implemented.
- 0b0001: The following Security Extensions are implemented:
  - The VBAR register.
  - The TTBCR.PD0 and TTBCR.PD1 bits.
- 0b0010: As for 0b0001, plus the ability to access Secure or Non-secure physical memory is supported.

All other values are reserved.

In Armv8-A, the permitted values are:
- 0b0000 when EL3 is implemented.
- 0b0001 or 0b0010 when EL3 is not implemented.

This field is valid only when the value of ID_PFR1.Security is 0, otherwise it holds the value 0b0000.

GenTimer, bits [19:16]
Generic Timer support. Defined values are:

- 0b0000: Generic Timer is not implemented.
- 0b0001: Generic Timer is implemented.
- 0b0010: Generic Timer is implemented, and also includes support for CNTHCTL.EVTNIS and CNTKCTL.EVTNIS fields, and CNTPCTSS and CNTVCTSS counter views.

All other values are reserved.

FEAT_ECV implements the functionality identified by the value 0b0010.

In Armv8.0, the only permitted value is 0b0001.

From Armv8.6, the only permitted value is 0b0010.

Virtualization, bits [15:12]
Virtualization support. Defined values are:

- 0b0000: EL2, Hyp mode, and the HVC instruction not implemented.
- 0b0001: EL2, Hyp mode, the HVC instruction, and all the features described by Virt_fracli == 0b0001 implemented.

All other values are reserved.

In Armv8-A, the permitted values are:
- 0b0000 when EL2 is not implemented.
- 0b0001 when EL2 is implemented.

In an implementation that includes EL2, if EL2 cannot use AArch32 but EL1 can use AArch32 then this field has the value 0b0001.

Note
The ID_ISARs do not identify whether the HVC instruction is implemented.
MProgMod, bits [11:8]

M-profile programmers' model support. Defined values are:

- 0b0000: Not supported.
- 0b0010: Support for two-stack programmers' model.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0000.

Security, bits [7:4]

Security support. Defined values are:

- 0b0000: EL3, Monitor mode, and the SMC instruction not implemented.
- 0b0001: EL3, Monitor mode, the SMC instruction, and all the features described by Sec_frac == 0b0001 implemented.
- 0b0010: As for 0b0001, and adds the ability to set the NSACR.RFR bit. Not permitted in Armv8 as the NSACR.RFR bit is RES0.

All other values are reserved.

In Armv8-A, the permitted values are:

- 0b0000 when EL3 is not implemented.
- 0b0001 when EL3 is implemented.

In an implementation that includes EL3, if EL3 cannot use AArch32 but EL1 can use AArch32 then this field has the value 0b0001.

ProgMod, bits [3:0]

Support for the standard programmers' model for ARMv4 and later. Model must support User, FIQ, IRQ, Supervisor, Abort, Undefined, and System modes. Defined values are:

- 0b0000: Not supported.
- 0b0001: Supported.

All other values are reserved.

In Armv8-A, the only permitted value is 0b0001.

Accessing ID_PFR1

Accesses to this register use the following encodings in the System register encoding space:

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR.TID3 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        R[t] = ID_PFR1;
    elsif PSTATE.EL == EL2 then
```
R[t] = ID_PFR1;
else if PSTATE_EL == EL3 then
    R[t] = ID_PFR1;
G8.2.100  ID_PFR2, Processor Feature Register 2

The ID_PFR2 characteristics are:

Purpose

Gives information about the AArch32 programmers' model.
Must be interpreted with ID_PFR0 and ID_PFR1.
For general information about the interpretation of the ID registers see Principles of the ID scheme for fields in ID registers on page G8-9332.

Configurations

AArch32 System register ID_PFR2 bits [31:0] are architecturally mapped to AArch64 System register ID_PFR2_EL1[31:0].
This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ID_PFR2 are UNDEFINED.

Attributes

ID_PFR2 is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>RAS_frac</td>
<td>SSBS</td>
<td>CSV3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits [31:12]

Reserved, RES0.

RAS_frac, bits [11:8]

RAS Extension fractional field.
0b0000  If ID_PFR0.RAS == 0b0001, RAS Extension implemented.
0b0001  If ID_PFR0.RAS == 0b0001, as 0b0000 and adds support for additional ERXMISC<m> System registers.

Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS and support for the optional RAS Timestamp Extension.
All other values are reserved.

This field is valid only if ID_PFR0.RAS == 0b0001.

SSBS, bits [7:4]

Speculative Store Bypassing controls in AArch64 state. Defined values are:
0b0000  AArch32 provides no mechanism to control the use of Speculative Store Bypassing.
0b0001  AArch32 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypass Safe.

In Armv8.0, the permitted values are 0b0000 and 0b0001.
From Armv8.5, the only permitted value is 0b0001.

All other values are reserved.
CSV3, bits [3:0]
Speculative use of faulting data. Defined values are:

0b0000  This PE does not disclose whether data loaded under speculation with a permission or domain fault can be used to form an address or generate condition codes or SVE predicate values to be used by other instructions in the speculative sequence.

0b0001  Data loaded under speculation with a permission or domain fault cannot be used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence. The execution timing of any other instructions in the speculative sequence is not a function of the data loaded under speculation.

All other values are reserved.
FEAT_CSV3 implements the functionality identified by the value 0b0001.
In Armv8.0, the permitted values are 0b0000 and 0b0001.
From Armv8.5, the only permitted value is 0b0001.
If FEAT_E0PD is implemented, FEAT_CSV3 must be implemented.

Accessing ID_PFR2
Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0011</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    R[t] = ID_PFR2;
  elsif PSTATE.EL == EL2 then
    R[t] = ID_PFR2;
  elsif PSTATE.EL == EL3 then
    R[t] = ID_PFR2;
G8.2.101 IFAR, Instruction Fault Address Register

The IFAR characteristics are:

**Purpose**
Holds the virtual address of the faulting address that caused a synchronous Prefetch Abort exception.

**Configurations**
- **AArch32 System register IFAR bits [31:0]** are architecturally mapped to AArch64 System register FAR_EL1[63:32].
- **AArch32 System register IFAR bits [31:0]** (IFAR_S) are architecturally mapped to AArch32 System register HIFAR[31:0] when EL2 is implemented, EL3 is implemented and the implementation only supports execution in AArch32 state.
- **AArch32 System register IFAR bits [31:0]** (IFAR_S) are architecturally mapped to AArch64 System register FAR_EL2[63:32] when EL2 is implemented.

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to IFAR are **UNDEFINED**.

**Attributes**
IFAR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>VA of faulting address of synchronous Prefetch Abort exception</td>
</tr>
</tbody>
</table>

Bits [31:0]
VA of faulting address of synchronous Prefetch Abort exception.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Accessing IFAR**

Accesses to this register use the following encodings in the System register encoding space:

`MRC<c>{q}<coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}`

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.EL2.T6 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.EL2.T6 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR.EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then

R[t] = IFAR_NS;
else
R[t] = IFAR;
elsif PSTATE.EL == EL2 then
if HaveEL(EL3) & ELUsingAArch32(EL3) then
R[t] = IFAR_NS;
else
R[t] = IFAR;
elsif PSTATE.EL == EL3 then
if SCR.NS == '0' then
R[t] = IFAR_S;
else
R[t] = IFAR_NS;
else
if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if EL2Enabled() & !ELUsingAArch32(EL2) & HSTR_EL2.T6 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR_EL2.T6 == '1' then
AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TVM == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR.TVM == '1' then
AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) & ELUsingAArch32(EL3) then
IFAR_NS = R[t];
else
IFAR = R[t];
elsif PSTATE.EL == EL2 then
if HaveEL(EL3) & ELUsingAArch32(EL3) then
IFAR_NS = R[t];
else
IFAR = R[t];
elsif PSTATE.EL == EL3 then
if SCR.NS == '0' then
IFAR_S = R[t];
else
IFAR_NS = R[t];
endif

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0110</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>
G8.2.102 IFSR, Instruction Fault Status Register

The IFSR characteristics are:

Purpose

Holds status information about the last instruction fault.

Configurations

AArch32 System register IFSR bits [31:0] are architecturally mapped to AArch64 System register IFSR32_EL2[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to IFSR are UNDEFINED.

The current translation table format determines which format of the register is used.

Attributes

IFSR is a 32-bit register.

Field descriptions

When TTBCR.EAE == 0:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0</td>
<td></td>
</tr>
<tr>
<td>30-24</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>LPAE</td>
<td></td>
</tr>
<tr>
<td>22-16</td>
<td>FS[4]</td>
<td></td>
</tr>
<tr>
<td>15-13</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>ExT, External abort type</td>
<td></td>
</tr>
<tr>
<td>11-8</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>7-0</td>
<td>FS[3:0]</td>
<td></td>
</tr>
</tbody>
</table>

Bits [31:17]

Reserved, RES0.

FnV, bit [16]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

0b0  IFAR is valid.
0b1  IFAR is not valid, and holds an UNKNOWN value.

This field is valid only for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:13]

Reserved, RES0.

ExT, bit [12]

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.
Bit [11]

Reserved, RES0.

FS, bits [10, 3:0]


- **0b00001**: PC alignment fault.
- **0b00010**: Debug exception.
- **0b00011**: Access flag fault, level 1.
- **0b00101**: Translation fault, level 1.
- **0b00110**: Access flag fault, level 2.
- **0b00111**: Translation fault, level 2.
- **0b01000**: Synchronous External abort, not on translation table walk.
- **0b01001**: Domain fault, level 1.
- **0b01011**: Domain fault, level 2.
- **0b01100**: Synchronous External abort, on translation table walk, level 1.
- **0b01101**: Permission fault, level 1.
- **0b01110**: Synchronous External abort, on translation table walk, level 2.
- **0b01111**: Permission fault, level 2.
- **0b10000**: TLB conflict abort.
- **0b10010**: IMPLEMENTATION DEFINED fault (Lockdown fault).
- **0b11001**: When FEAT_RAS is not implemented:
  - Synchronous parity or ECC error on memory access, not on translation table walk.
- **0b11100**: When FEAT_RAS is not implemented:
  - Synchronous parity or ECC error on translation table walk, level 1.
- **0b11110**: When FEAT_RAS is not implemented:
  - Synchronous parity or ECC error on translation table walk, level 2.

All other values are reserved.

For more information about the lookup level associated with a fault, see *The level associated with MMU faults on a Short-descriptor translation table lookup on page G5-9257*.

The FS field is split as follows:

- **FS[4]** is IFSR[10].
- **FS[3:0]** is IFSR[3:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

LPAE, bit [9]

On taking a Data Abort exception, this bit is set as follows:

- **0b0**: Using the Short-descriptor translation table formats.
- **0b1**: Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:4]

Reserved, RES0.
When TTBCR.EAE == 1:

Bits [31:17]
Reserved, RES0.

FnV, bit [16]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

0b0 IFAR is valid.

0b1 IFAR is not valid, and holds an UNKNOWN value.

This field is valid only for a synchronous External abort other than a synchronous External abort on a translation table walk. It is RES0 for all other Prefetch Abort exceptions.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:13]
Reserved, RES0.

ExT, bit [12]

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:10]
Reserved, RES0.

LPAE, bit [9]

On taking a Data Abort exception, this bit is set as follows:

0b0 Using the Short-descriptor translation table formats.

0b1 Using the Long-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:6]
Reserved, RES0.

STATUS, bits [5:0]

Fault status bits. Possible values of this field are:

0b000000 Address size fault in translation table base register.

0b000001 Address size fault, level 1.
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000010</td>
<td>Address size fault, level 2.</td>
</tr>
<tr>
<td>0b000011</td>
<td>Address size fault, level 3.</td>
</tr>
<tr>
<td>0b000101</td>
<td>Translation fault, level 1.</td>
</tr>
<tr>
<td>0b001101</td>
<td>Translation fault, level 2.</td>
</tr>
<tr>
<td>0b001111</td>
<td>Translation fault, level 3.</td>
</tr>
<tr>
<td>0b010010</td>
<td>Access flag fault, level 2.</td>
</tr>
<tr>
<td>0b010110</td>
<td>Access flag fault, level 3.</td>
</tr>
<tr>
<td>0b011101</td>
<td>Permission fault, level 1.</td>
</tr>
<tr>
<td>0b011110</td>
<td>Permission fault, level 2.</td>
</tr>
<tr>
<td>0b011111</td>
<td>Permission fault, level 3.</td>
</tr>
<tr>
<td>0b010000</td>
<td>Synchronous External abort, not on translation table walk.</td>
</tr>
<tr>
<td>0b010101</td>
<td>Synchronous External abort on translation table walk, level 1.</td>
</tr>
<tr>
<td>0b010110</td>
<td>Synchronous External abort on translation table walk, level 2.</td>
</tr>
<tr>
<td>0b010111</td>
<td>Synchronous External abort on translation table walk, level 3.</td>
</tr>
<tr>
<td>0b011000</td>
<td>When FEAT_RAS is not implemented: Synchronous parity or ECC error on memory access, not on translation table walk.</td>
</tr>
<tr>
<td>0b011101</td>
<td>When FEAT_RAS is not implemented: Synchronous parity or ECC error on memory access on translation table walk, level 1.</td>
</tr>
<tr>
<td>0b011110</td>
<td>When FEAT_RAS is not implemented: Synchronous parity or ECC error on memory access on translation table walk, level 2.</td>
</tr>
<tr>
<td>0b011111</td>
<td>When FEAT_RAS is not implemented: Synchronous parity or ECC error on memory access on translation table walk, level 3.</td>
</tr>
<tr>
<td>0b010001</td>
<td>PC alignment fault.</td>
</tr>
<tr>
<td>0b010010</td>
<td>Debug exception.</td>
</tr>
<tr>
<td>0b010000</td>
<td>TLB conflict abort.</td>
</tr>
</tbody>
</table>

All other values are reserved.

When FEAT_RAS is implemented, 0b011100, 0b011101, 0b011110, and 0b011111 are reserved.

For more information about the lookup level associated with a fault, see The level associated with MMU faults on a Long-descriptor translation table lookup on page G5-9259.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing IFSR**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} \text{ <coproc}, \{#<opc1>, <Rt>, <CRn}, <CRm>{, {#}<opc2}>}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
   UNDEFINED;
elsif PSTATE.EL == EL1 then
   if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
  R[t] = IFSR_NS;
else
  R[t] = IFSR;
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    R[t] = IFSR_NS;
  else
    R[t] = IFSR;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    R[t] = IFSR_S;
  else
    R[t] = IFSR_NS;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    IFSR_NS = R[t];
  else
    IFSR = R[t];
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    IFSR_NS = R[t];
  else
    IFSR = R[t];
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    IFSR_S = R[t];
  else
    IFSR_NS = R[t];

| MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} |
|------------------|---|---|---|---|---|
| coproc | opc1 | CRn | CRm | opc2 |
| 0b1111 | 0b000 | 0b0101 | 0b0000 | 0b001 |
G8.2.103  ISR, Interrupt Status Register

The ISR characteristics are:

**Purpose**

Shows the pending status of the IRQ, FIQ, or SError.

When executing at EL2, EL3, or Secure EL1, when \( \text{SCR\_EL3.EEL2} = \overline{0b0} \), this shows the pending status of the physical interrupts.

When executing at Non-secure EL1, or at Secure EL1, when \( \text{SCR\_EL3.EEL2} = \overline{0b01} \):
- If the \( \text{HCR.\{IMO,FMO,AMO\}} \) bit has a value of 1, the corresponding ISR.\{I,F,A\} bit shows the pending status of the virtual IRQ, FIQ, or SError.
- If the \( \text{HCR.\{IMO,FMO,AMO\}} \) bit has a value of 0, the corresponding ISR.\{I,F,A\} bit shows the pending status of the physical IRQ, FIQ, or SError.

**Configurations**

AArch32 System register ISR bits [31:0] are architecturally mapped to AArch64 System register ISR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ISR are UNDEFINED.

**Attributes**

ISR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>SError interrupt pending bit:</td>
</tr>
<tr>
<td>I</td>
<td>IRQ pending bit. Indicates whether an IRQ interrupt is pending:</td>
</tr>
<tr>
<td>F</td>
<td>FIQ pending bit. Indicates whether an FIQ interrupt is pending.</td>
</tr>
</tbody>
</table>

- **Bits [31:9]**
  - Reserved, RES0.

- **A, bit [8]**
  - SError interrupt pending bit:
    - \( \overline{0b0} \) No pending SError interrupt.
    - \( \overline{0b1} \) An SError interrupt is pending.

- **I, bit [7]**
  - IRQ pending bit. Indicates whether an IRQ interrupt is pending:
    - \( \overline{0b0} \) No pending IRQ.
    - \( \overline{0b1} \) An IRQ interrupt is pending.

- **F, bit [6]**
  - FIQ pending bit. Indicates whether an FIQ interrupt is pending:
    - \( \overline{0b0} \) No pending FIQ.
    - \( \overline{0b1} \) An FIQ interrupt is pending.

- **Bits [5:0]**
  - Reserved, RES0.
Accessing ISR

Accesses to this register use the following encodings in the System register encoding space:

\[ \text{MRC}\{<c>\}{<q>}\text{ <coproc>}, \{#\text{<opc1>}, \text{<Rt>}, \text{<CRn>}, \text{<CRm>}, \{#\text{<opc2>}} \}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if \text{PSTATE.EL} == \text{EL0} then
  \text{UNDEFINED};
elsif \text{PSTATE.EL} == \text{EL1} then
  if \text{EL2Enabled()} \&\& \text{ELUsingAArch32(EL2)} \&\& \text{HSTR.EL2.T12 == '1'} then
    \text{AArch64.AArch32SystemAccessTrap(EL2, 0x03)};
  elseif \text{EL2Enabled()} \&\& \text{ELUsingAArch32(EL2)} \&\& \text{HSTR.T12 == '1'} then
    \text{AArch32.TakeHypTrapException(0x03)};
  else
    \text{R[t] = ISR};
  endif
elsif \text{PSTATE.EL} == \text{EL2} then
  \text{R[t] = ISR};
elsif \text{PSTATE.EL} == \text{EL3} then
  \text{R[t] = ISR};
G8.2.104  ITLBIALL, Instruction TLB Invalidate All

The ITLBIALL characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from instruction TLBs that are from any level of the translation table walk. The entries that are invalidated are as follows:

- If executed at EL1, all entries that:
  - Would be required for the EL1&0 translation regime.
  - Match the current VMID, if EL2 is implemented and enabled in the current Security state.
- If executed in Secure state when EL3 is using AArch32, all entries that would be required for the Secure PL1&0 translation regime.
- If executed at EL2, and if EL2 is enabled in the current Security state, the stage 1 or stage 2 translation table entries that would be required for the Non-secure PL1&0 translation regime and matches the current VMID.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backwards compatibility with earlier versions of the Arm architecture.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ITLBIALL are UNDEFINED.

**Attributes**

ITLBIALL is a 32-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

**Executing ITLBIALL instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[ \text{MCR}\{<c>\}{<q>} <\text{coproc}>, {#}<\text{opc1}>, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>{, {#}<\text{opc2}>} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0101</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    if IsFeatureImplemented(FEAT_XS) && !ELUsingAArch32(EL2) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then

G8-9628  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. Non-Confidential
AArch32.ITLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_NSH, TLBI_ExcludeXS);
else
  AArch32.ITLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_NSH, TLBI_AllAttr);
elsif PSTATE.EL == EL2 then
  AArch32.ITLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_NSH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
  AArch32.ITLBI_ALL(SecurityStateAtEL(EL3), Regime_EL30, Shareability_NSH, TLBI_AllAttr);
**G8.2.105 ITLBIASID, Instruction TLB Invalidate by ASID match**

The ITLBIASID characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from instruction TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backwards compatibility with earlier versions of the Arm architecture.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ITLBIASID are undefined.

**Attributes**

ITLBIASID is a 32-bit System instruction.

**Field descriptions**

```
<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>ASID</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Bits [31:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this System instruction.

**Executing ITLBIASID instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:
MCR{<c>}{<q>} <coproc>, (#)opc1, <Rt>, <CRn>, <CRm>{, (#)opc2}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b000</td>
<td>0b100</td>
<td>0b010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.EL2.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR.EL2.TTLB == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
    AArch32.TakeHypTrapException(0x03);
else
  if IsFeatureImplemented(FEAT_XS) && !ELUsingAArch32(EL2) && IsFeatureImplemented(FEAT_HCX) &&
    IsHCRXEL2Enabled() && HCRX.EL2.FnXS == '1' then
    AArch32.ITLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH,
    TLBI.ExcludeXS, R[t]);
  else
    AArch32.ITLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH,
    TLBI.AllAttr, R[t]);
  elsif PSTATE.EL == EL2 then
    AArch32.ITLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI.AllAttr,
    R[t]);
  elsif PSTATE.EL == EL3 then
    AArch32.ITLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Shareability_NSH, TLBI.AllAttr,
    R[t]);

ITLBIMVA, Instruction TLB Invalidate by VA

The ITLBIMVA characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from instruction TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backwards compatibility with earlier versions of the Arm architecture.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to ITLBIMVA are UNDEFINED.

**Attributes**

ITLBIMVA is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA, bits</td>
<td>Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.</td>
<td>[31:12]</td>
</tr>
<tr>
<td>ASID, bits</td>
<td>ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction. Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.</td>
<td>[7:0]</td>
</tr>
</tbody>
</table>
Executing ITLBIMVA instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[ \text{MCR}\{<c>\}\{<q>\} <\text{coproc}>, \{#\}<\text{opc1}>, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>, \{#\}<\text{opc2}> \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0101</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif PSTATE.EL == EL2 then
    AArch32.ITLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBLevel.Any, TLBI_ExcludeXS, R[t]);
  elseif PSTATE.EL == EL3 then
    AArch32.ITLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_NSH, TLBLevel.Any, TLBI_AllAttr, R[t]);
else
  if IsFeatureImplemented(FEAT_XS) && !ELUsingAArch32(EL2) && IsFeatureImplemented(FEAT_HCX) &&
    IsHCRX_EL2Enabled() && HCRX_EL2.FnXS == '1' then
    AArch32.ITLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBLevel.Any, TLBI_AllAttr, R[t]);
  else
    AArch32.ITLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBLevel.Any, TLBI_AllAttr, R[t]);
  endif
else
  AArch32.ITLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_NSH, TLBLevel.Any, TLBI_AllAttr, R[t]);
endif
G8.2.107  JIDR, Jazelle ID Register

The JIDR characteristics are:

**Purpose**

A Jazelle register, which identified the Jazelle architecture version.

**Configurations**

This register is present only when AArch32 is supported. Otherwise, direct accesses to JIDR are UNDEFINED.

**Attributes**

JIDR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits [31:0]</td>
<td>Reserved, RAZ.</td>
</tr>
</tbody>
</table>

**Accessing JIDR**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC} \{<c>\} \{<q>\} <\text{coproc}>, \{#\text{opc1}\}, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>\{, \{#\text{opc2}\}\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b11110</td>
<td>0b111</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if boolean IMPLEMENTATION_DEFINED "JIDR UNDEFINED at EL0" then
        UNDEFINED;
    elsif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.<E2H,TGE> != '11' & HCR_EL2.TID0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR.TID0 == '1' then
        AArch32.TakeHypTrapException(0x05);
    else
        R[t] = JIDR;
    end if;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TID0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR.TID0 == '1' then
        AArch32.TakeHypTrapException(0x05);
    else
        R[t] = JIDR;
    end if;
elsif PSTATE.EL == EL2 then
    R[t] = JIDR;
elsif PSTATE.EL == EL3 then
    R[t] = JIDR;
G8.2.108 JMC
c, Jazelle Main Configuration Register

The JMC characteristics are:

**Purpose**

A Jazelle register, which provides control of the Jazelle extension.

**Configurations**

This register is present only when AArch32 is supported. Otherwise, direct accesses to JMC are UNDEFINED.

**Attributes**

JMC is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, RAZ/WI.</td>
</tr>
</tbody>
</table>

**Accessing JMC
c**

For accesses from EL0 it is IMPLEMENTATION DEFINED whether the register is RW or UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC\{<c>\}{<q>} <coproc>, \{#\}<opc1>, <Rt>, <CRn>, <CRm>{, \{#\}<opc2>} \]

<table>
<thead>
<tr>
<th>[coproc]</th>
<th>[opc1]</th>
<th>[CRn]</th>
<th>[CRm]</th>
<th>[opc2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b111</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if boolean IMPLEMENTATION_DEFINED "JMC UNDEFINED at EL0" then
        UNDEFINED;
    else
        R[t] = JMC;
    elsif PSTATE.EL == EL1 then
        R[t] = JMC;
    elsif PSTATE.EL == EL2 then
        R[t] = JMC;
    elsif PSTATE.EL == EL3 then
        R[t] = JMC;
}
### MCR\{c\}\{q\} \{#\}coproc, \{#\}opc1, <Rt>, <CRn>, <CRm>{, \{#\}opc2}\}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b110</td>
<td>0b11</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if boolean IMPLEMENTATION_DEFINED "JMCR UNDEFINED at EL0" then
    UNDEFINED;
  else
    return;
elsif PSTATE.EL == EL1 then
  return;
elsif PSTATE.EL == EL2 then
  return;
elsif PSTATE.EL == EL3 then
  return;
G8.2.109  JOSCR, Jazelle OS Control Register

The JOSCR characteristics are:

Purpose
A Jazelle register, which provides operating system control of the Jazelle Extension.

Configurations
This register is present only when AArch32 is supported. Otherwise, direct accesses to JOSCR are UNDEFINED.

Attributes
JOSCR is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, RAZ/WI</td>
<td></td>
</tr>
</tbody>
</table>

Accessing JOSCR
Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} \langle\text{coproc}\rangle, \{\#\langle\text{opc1}\rangle, \langle\text{Rt}\rangle, \langle\text{CRn}\rangle, \langle\text{CRm}\rangle\}, \{\#\langle\text{opc2}\rangle\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b111</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if \text{PSTATE.EL} == \text{EL0} then
if boolean IMPLEMENTATION_DEFINED “JOSCR UNDEFINED at EL0” then
UNDEFINED;
else
R[t] = JOSCR;
elsif \text{PSTATE.EL} == \text{EL1} then
R[t] = JOSCR;
elsif \text{PSTATE.EL} == \text{EL2} then
R[t] = JOSCR;
elsif \text{PSTATE.EL} == \text{EL3} then
R[t] = JOSCR;

\[
\text{MCR}\{<c>\}{<q>} \langle\text{coproc}\rangle, \{\#\langle\text{opc1}\rangle, \langle\text{Rt}\rangle, \langle\text{CRn}\rangle, \langle\text{CRm}\rangle\}, \{\#\langle\text{opc2}\rangle\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b111</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if \text{PSTATE.EL} == \text{EL0} then
if boolean IMPLEMENTATION_DEFINED “JOSCR UNDEFINED at EL0” then
UNDEFINED;
else
    return;
elsif PSTATE.EL == EL1 then
    return;
elsif PSTATE.EL == EL2 then
    return;
elsif PSTATE.EL == EL3 then
    return;
G8.2.110  MAIR0, Memory Attribute Indirection Register 0

The MAIR0 characteristics are:

**Purpose**

Along with MAIR1, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations.

AttrIndx[2] indicates the MAIR register to be used:
- When AttrIndx[2] is 0, MAIR0 is used.
- When AttrIndx[2] is 1, MAIR1 is used.

**Configurations**

AArch32 System register MAIR0 bits [31:0] are architecturally mapped to AArch64 System register MAIR_EL1[31:0] when EL3 is not implemented or EL3 is using AArch64.

AArch32 System register MAIR0 bits [31:0] are architecturally mapped to AArch32 System register PRRR[31:0] when EL3 is not implemented or EL3 is using AArch64.

AArch32 System register MAIR0 bits [31:0] (MAIR0_NS) are architecturally mapped to AArch32 System register PRRR[31:0] (PRRR_NS) when EL3 is using AArch32.

AArch32 System register MAIR0 bits [31:0] (MAIR0_S) are architecturally mapped to AArch32 System register PRRR[31:0] (PRRR_S) when EL3 is using AArch32.

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to MAIR0 are UNDEFINED.

MAIR0 and PRRR are the same register, with a different view depending on the value of TTBCR.EAE:
- When it is set to 0, the register is as described in PRRR.
- When it is set to 1, the register is as described in MAIR0.

When EL3 is using AArch32, write access to MAIR0(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

**Attributes**

MAIR0 is a 32-bit register.

**Field descriptions**

*When TTBCR.EAE == 1:*

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attr3</td>
<td>Attr2</td>
<td>Attr1</td>
<td>Attr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Attr<n>, bits [8n+7:8n], for n = 3 to 0**

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:
- AttrIndx[2:0] gives the value of <n> in Attr<n>.
- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.
Bits [7:4] are encoded as follows:

<table>
<thead>
<tr>
<th>Attr&lt;n&gt;[7:4]</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>Device memory. See encoding of Attr&lt;n&gt;[3:0] for the type of Device memory.</td>
</tr>
<tr>
<td>0b00RW, RW not 0b0</td>
<td>Normal memory, Outer Write-Through Transient.</td>
</tr>
<tr>
<td>0b0100</td>
<td>Normal memory, Outer Non-cacheable.</td>
</tr>
<tr>
<td>0b01RW, RW not 0b0</td>
<td>Normal memory, Outer Write-Back Transient.</td>
</tr>
<tr>
<td>0b10RW</td>
<td>Normal memory, Outer Write-Through Non-transient.</td>
</tr>
<tr>
<td>0b11RW</td>
<td>Normal memory, Outer Write-Back Non-transient.</td>
</tr>
</tbody>
</table>

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.
The meaning of bits [3:0] depends on the value of bits [7:4]:

<table>
<thead>
<tr>
<th>Attr&lt;n&gt;[3:0]</th>
<th>Meaning when Attr&lt;n&gt;[7:4] is 0b0000</th>
<th>Meaning when Attr&lt;n&gt;[7:4] is not 0b0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>Device-nGnRnE memory</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>0b00RW, RW not 0b0</td>
<td>UNPREDICTABLE</td>
<td>Normal memory, Inner Write-Through Transient</td>
</tr>
<tr>
<td>0b0100</td>
<td>Device-nGnRE memory</td>
<td>Normal memory, Inner Non-cacheable</td>
</tr>
<tr>
<td>0b01RW, RW not 0b0</td>
<td>UNPREDICTABLE</td>
<td>Normal memory, Inner Write-Back Transient</td>
</tr>
<tr>
<td>0b1000</td>
<td>Device-nGRE memory</td>
<td>Normal memory, Inner Write-Through Non-transient (RW=0b00)</td>
</tr>
<tr>
<td>0b10RW, RW not 0b0</td>
<td>UNPREDICTABLE</td>
<td>Normal memory, Inner Write-Through Non-transient</td>
</tr>
<tr>
<td>0b1100</td>
<td>Device-GRE memory</td>
<td>Normal memory, Inner Write-Back Non-transient (RW=0b00)</td>
</tr>
<tr>
<td>0b11RW, RW not 0b0</td>
<td>UNPREDICTABLE</td>
<td>Normal memory, Inner Write-Back Non-transient</td>
</tr>
</tbody>
</table>

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.
The R and W bits in some Attr<n> fields have the following meanings:

<table>
<thead>
<tr>
<th>R or W</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>No Allocate</td>
</tr>
<tr>
<td>0b1</td>
<td>Allocate</td>
</tr>
</tbody>
</table>

When `FEAT_XS` is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally `UNKNOWN` value.
Accessing MAIR0

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC\{<c>\}{<q>} \{coproc\}, \{\#\}<opc1>, <Rt>, <CRn>, <CRm>{, \{\#\}<opc2>}\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>
MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.TI0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.TI0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    if TTBCR.EAE == '1' then
      MAIR0_NS = R[t];
    else
      PRRR_NS = R[t];
    end if;
  elsif TTBCR.EAE == '1' then
    MAIR0 = R[t];
  else
    PRRR = R[t];
  end if;
elsif PSTATE_EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    if TTBCR.EAE == '1' then
      MAIR0_NS = R[t];
    else
      PRRR_NS = R[t];
    end if;
  elsif TTBCR.EAE == '1' then
    MAIR0 = R[t];
  else
    PRRR = R[t];
  end if;
elsif PSTATE_EL == EL3 then
  if SCR_NS == '0' && CP15SDISABLE == HIGH then
    UNDEFINED;
  elsif SCR_NS == '0' && CP15SDISABLE2 == HIGH then
    UNDEFINED;
  else
    if TTBCR.EAE == '1' then
      if SCR_NS == '0' then
        MAIR0_S = R[t];
      else
        MAIR0_NS = R[t];
      end if;
    else
      PRRR_S = R[t];
    end if;
  end if;
G8.2.111 MAIR1, Memory Attribute Indirection Register 1

The MAIR1 characteristics are:

**Purpose**

Along with MAIR0, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations.

AttrIndx[2] indicates the MAIR register to be used:

- When AttrIndx[2] is 0, MAIR0 is used.
- When AttrIndx[2] is 1, MAIR1 is used.

**Configurations**

AArch32 System register MAIR1 bits [31:0] are architecturally mapped to AArch64 System register MAIR_EL1[63:32] when EL3 is not implemented or EL3 is using AArch64.

AArch32 System register MAIR1 bits [31:0] are architecturally mapped to AArch32 System register NMRR[31:0] when EL3 is not implemented or EL3 is using AArch64.

AArch32 System register MAIR1 bits [31:0] (MAIR1_NS) are architecturally mapped to AArch32 System register NMRR[31:0] (NMRR_NS) when EL3 is using AArch32.

AArch32 System register MAIR1 bits [31:0] (MAIR1_S) are architecturally mapped to AArch32 System register NMRR[31:0] (NMRR_S) when EL3 is using AArch32.

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to MAIR1 are UNDEFINED.

MAIR1 and NMRR are the same register, with a different view depending on the value of TTBCR.EAE:

- When it is set to 0, the register is as described in NMRR.
- When it is set to 1, the register is as described in MAIR1.

When EL3 is using AArch32, write access to MAIR1(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

**Attributes**

MAIR1 is a 32-bit register.

**Field descriptions**

*When TTBCR.EAE == 1:*

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>28</td>
<td>23</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>Attr7</td>
<td>Attr6</td>
<td>Attr5</td>
<td>Attr4</td>
<td></td>
</tr>
</tbody>
</table>

**Attr<n>, bits [8(n-4)+7:8(n-4)], for n = 7 to 4**

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIndx[2:0] gives the value of <n> in Attr<n>.
- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.
Bits [7:4] are encoded as follows:

<table>
<thead>
<tr>
<th>Attr&lt;n&gt;[7:4]</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>Device memory. See encoding of Attr&lt;n&gt;[3:0] for the type of Device memory.</td>
</tr>
<tr>
<td>0b00RW, RW not 0b00</td>
<td>Normal memory, Outer Write-Through Transient.</td>
</tr>
<tr>
<td>0b0100</td>
<td>Normal memory, Outer Non-cacheable.</td>
</tr>
<tr>
<td>0b01RW, RW not 0b00</td>
<td>Normal memory, Outer Write-Back Transient.</td>
</tr>
<tr>
<td>0b10RW</td>
<td>Normal memory, Outer Write-Through Non-transient.</td>
</tr>
<tr>
<td>0b11RW</td>
<td>Normal memory, Outer Write-Back Non-transient.</td>
</tr>
</tbody>
</table>

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

The meaning of bits [3:0] depends on the value of bits [7:4]:

<table>
<thead>
<tr>
<th>Attr&lt;n&gt;[3:0]</th>
<th>Meaning when Attr&lt;n&gt;[7:4] is 0b0000</th>
<th>Meaning when Attr&lt;n&gt;[7:4] is not 0b0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>Device-nGnRnE memory</td>
<td>UNPREDICTABLE</td>
</tr>
<tr>
<td>0b00RW, RW not 0b00</td>
<td>UNPREDICTABLE</td>
<td>Normal memory, Inner Write-Through Transient</td>
</tr>
<tr>
<td>0b0100</td>
<td>Device-nGnRE memory</td>
<td>Normal memory, Inner Non-cacheable</td>
</tr>
<tr>
<td>0b01RW, RW not 0b00</td>
<td>UNPREDICTABLE</td>
<td>Normal memory, Inner Write-Back Transient</td>
</tr>
<tr>
<td>0b1000</td>
<td>Device-nGRE memory</td>
<td>Normal memory, Inner Write-Through Non-transient (RW=0b00)</td>
</tr>
<tr>
<td>0b10RW, RW not 0b00</td>
<td>UNPREDICTABLE</td>
<td>Normal memory, Inner Write-Through Non-transient</td>
</tr>
<tr>
<td>0b1100</td>
<td>Device-GRE memory</td>
<td>Normal memory, Inner Write-Back Non-transient (RW=0b00)</td>
</tr>
<tr>
<td>0b11RW, RW not 0b00</td>
<td>UNPREDICTABLE</td>
<td>Normal memory, Inner Write-Back Non-transient</td>
</tr>
</tbody>
</table>

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in some Attr<n> fields have the following meanings:

<table>
<thead>
<tr>
<th>R or W</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>No Allocate</td>
</tr>
<tr>
<td>0b1</td>
<td>Allocate</td>
</tr>
</tbody>
</table>

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Accessing MAIR1

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC\{<c>\}{<q>} \coproc, \{#\}<opc1>, <Rt>, <CRn>, <CRm>\{, \{#\}<opc2>\} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T10 == '1' then
    AArch32.TakeHypTrapException(x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
    AArch32.TakeHypTrapException(x03);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    if TTBCR.EAE == '1' then
      R[t] = MAIR1_NS;
    else
      R[t] = NMRR_NS;
    end
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    if TTBCR.EAE == '1' then
      R[t] = MAIR1;
    else
      R[t] = NMRR;
    end
  elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
      if TTBCR.EAE == '1' then
        R[t] = MAIR1_NS;
      else
        R[t] = NMRR_NS;
      end
    else
      if TTBCR.EAE == '1' then
        R[t] = MAIR1;
      else
        R[t] = NMRR;
      end
    end
  elif PSTATE.EL == EL3 then
    if TTBCR.EAE == '1' then
      if SCR.NS == '0' then
        R[t] = MAIR1_S;
      else
        R[t] = MAIR1_NS;
      end
    else
      if SCR.NS == '0' then
        R[t] = NMRR_S;
      else
        R[t] = NMRR_NS;
      end
    end
  else
    if SCR.NS == '0' then
      R[t] = MAIR1_S;
    else
      R[t] = NMRR_S;
    end
  end
MCR{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>)}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T10 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        if TTBCR.EAE == '1' then
            MAIR1_NS = R[t];
        else
            NMRR_NS = R[t];
        else
            if TTBCR.EAE == '1' then
                MAIR1 = R[t];
            else
                NMRR = R[t];
        elsif PSTATE.EL == EL2 then
            if HaveEL(EL3) && ELUsingAArch32(EL3) then
                if TTBCR.EAE == '1' then
                    MAIR1_NS = R[t];
                else
                    NMRR_NS = R[t];
                else
                    if TTBCR.EAE == '1' then
                        MAIR1_S = R[t];
                    else
                        NMRR_S = R[t];
                else
                    if SCR_NS == '0' && CP15SDISABLE == HIGH then
                        UNDEFINED;
                    elsif SCR_NS == '0' && CP15SDISABLE2 == HIGH then
                        UNDEFINED;
                    else
                        if TTBCR.EAE == '1' then
                            if SCR_NS == '0' then
                                MAIR1_S = R[t];
                            else
                                MAIR1_NS = R[t];
                            else
                                if SCR_NS == '0' then
                                    NMRR_S = R[t];
                                else
                                    NMRR_NS = R[t];
                            end
                        end
                    end
                end
            end
        end
    end
end
G8.2.112  MIDR, Main ID Register

The MIDR characteristics are:

**Purpose**

Provides identification information for the PE, including an implementer code for the device and a device ID number.

**Configurations**

AArch32 System register MIDR bits [31:0] are architecturally mapped to AArch64 System register MIDR_EL1[31:0].

AArch32 System register MIDR bits [31:0] are architecturally mapped to External register MIDR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to MIDR are UNDEFINED.

Some fields of the MIDR are IMPLEMENTATION DEFINED. For more information about the values of these fields for a particular Armv8 implementation, and any implementation-specific significance of these values, see the product documentation.

**Attributes**

MIDR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>6</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementer</td>
<td>Variant</td>
<td>PartNum</td>
<td>Revision</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Implementer, bits [31:24]**

The Implementer code. This field must hold an implementer code that has been assigned by Arm. Assigned codes include the following:

- **0x00**  Reserved for software use.
- **0x41**  Arm Limited.
- **0x42**  Broadcom Corporation.
- **0x43**  Cavium Inc.
- **0x44**  Digital Equipment Corporation.
- **0x46**  Fujitsu Ltd.
- **0x49**  Infineon Technologies AG.
- **0x40**  Motorola or Freescale Semiconductor Inc.
- **0x4E**  NVIDIA Corporation.
- **0x50**  Applied Micro Circuits Corporation.
- **0x51**  Qualcomm Inc.
- **0x56**  Marvell International Ltd.
- **0x69**  Intel Corporation.
- **0xC0**  Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.
Variant, bits [23:20]

Variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Architecture, bits [19:16]

Architecture version. Defined values are:
- 0b0001 Armv4.
- 0b0010 Armv4T.
- 0b0011 Armv5 (obsolete).
- 0b0100 Armv5T.
- 0b0101 Armv5TE.
- 0b0110 Armv5TEJ.
- 0b0111 Armv6.
- 0b1111 Architectural features are individually identified in the ID_* registers.

All other values are reserved.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

PartNum, bits [15:4]

Primary Part Number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Revision, bits [3:0]

Revision number for the device.

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Accessing MIDR

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} <coproc>, \{#<opc1>, <Rt>, <CRn>, <CRm>{, \{#<opc2>\}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>
elsif EL2Enabled() && ELUsingAArch32(EL2) then
    R[t] = VPIDR;
elsif PSTATE.EL == EL2 then
    R[t] = MIDR;
elsif PSTATE.EL == EL3 then
    R[t] = MIDR;
else
    R[t] = MIDR;
G8.2.113 MPIDR, Multiprocessor Affinity Register

The MPIDR characteristics are:

**Purpose**

In a multiprocessor system, provides an additional PE identification mechanism for scheduling purposes.

**Configurations**

AArch32 System register MPIDR bits [31:0] are architecturally mapped to AArch64 System register MPIDR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to MPIDR are UNDEFINED.

In a uniprocessor system, Arm recommends that each Aff<n> field of this register returns a value of 0.

**Attributes**

MPIDR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>bit [31]</td>
</tr>
<tr>
<td>U</td>
<td>bit [30]</td>
</tr>
<tr>
<td>RES0</td>
<td>bit [29:25]</td>
</tr>
<tr>
<td>MT</td>
<td>bit [24]</td>
</tr>
<tr>
<td>Aff2</td>
<td>bits [23:16]</td>
</tr>
<tr>
<td>Aff1</td>
<td></td>
</tr>
<tr>
<td>Aff0</td>
<td></td>
</tr>
</tbody>
</table>

**M, bit [31]**

Indicates whether this implementation includes the functionality introduced by the Armv7 Multiprocessing Extensions.

- 0b0: This implementation does not include the Armv7 Multiprocessing Extensions functionality.
- 0b1: This implementation includes the Armv7 Multiprocessing Extensions functionality.

Access to this field is RAO/WI.

**U, bit [30]**

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

- 0b0: Processor is part of a multiprocessor system.
- 0b1: Processor is part of a uniprocessor system.

**Bits [29:25]**

Reserved, RES0.

**MT, bit [24]**

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multithreading type approach. See the description of Aff0 for more information about affinity levels.

- 0b0: Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
- 0b1: Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

**Aff2, bits [23:16]**

Affinity level 2. See the description of Aff0 for more information.
Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

Aff0, bits [7:0]

Affinity level 0. This is the affinity level that is most significant for determining PE behavior. Higher affinity levels are increasingly less significant in determining PE behavior. The assigned value of the MPIDR.\{Aff2, Aff1, Aff0\} or MPIDR.EL1.\{Aff3, Aff2, Aff1, Aff0\} set of fields of each PE must be unique within the system as a whole.

Accessing MPIDR

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC\{<c>\{<q>\} \langle coproc\rangle, \langle#opc1\rangle, \langle Rt\rangle, \langle CRn\rangle, \langle CRm\rangle\{, \langle#opc2\rangle\} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) then
    R[t] = VMPIDR_EL2<31:0>;
  elsif EL2Enabled() && ELUsingAArch32(EL2) then
    R[t] = VMPIDR;
else
  R[t] = MPIDR;
elsif PSTATE.EL == EL2 then
  R[t] = MPIDR;
elsif PSTATE.EL == EL3 then
  R[t] = MPIDR;
G8.2.114   MVBAR, Monitor Vector Base Address Register

The MVBAR characteristics are:

**Purpose**

When EL3 is implemented and can use AArch32, holds the vector base address for any exception that is taken to Monitor mode.

Secure software must program the MVBAR with the required initial value as part of the PE boot sequence.

**Configurations**

This register is present only when EL3 is capable of using AArch32. Otherwise, direct accesses to MVBAR are UNDEFINED.

It is IMPLEMENTATION DEFINED whether MVBAR[0] has a fixed value and ignored writes, or takes the last value written to it.

On a Warm reset into EL3 using AArch32, the reset value of MVBAR is an IMPLEMENTATION DEFINED choice between the following:

- MVBAR[31:5] = an IMPLEMENTATION DEFINED value, which might be UNKNOWN, MVBAR[4:1] = RES0, and MVBAR[0] = 0.
- MVBAR[31:1] = an IMPLEMENTATION DEFINED value that is bits[31:1] of the AArch32 reset address, and MVBAR[0] = 1.

**Attributes**

MVBAR is a 32-bit register.

**Field descriptions**

*When programmed with a vector base address:*

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-5</td>
<td>Vector Base Address</td>
</tr>
<tr>
<td>4-0</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

**Bits [31:5]**

Vector Base Address. Bits[31:5] of the base address of the exception vectors for exceptions taken to this Exception level. Bits[4:0] of an exception vector are the exception offset.

**Reserved, bits [4:0]**

Reserved, see Configurations.

**Accessing MVBAR**

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC\{<c>\}{<q>\} \langle\{#\}\langle opc1\rangle, <Rt>, <CRn>, <CRm\rangle, \{#\}\langle opc2\rangle\rangle \]

<table>
<thead>
<tr>
<th></th>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0b1111</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if IsHighestEL(EL1) then
R[t] = RVBAR;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
  AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  if IsHighestEL(EL2) then
    R[t] = RVBAR;
  else
    UNDEFINED;
elsif PSTATE.EL == EL3 then
  R[t] = MVBAR;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.EL == EL3 then
  if CP15SDISABLE == HIGH then
    UNDEFINED;
  elsif CP15SDISABLE2 == HIGH then
    UNDEFINED;
  else
    MVBAR = R[t];
G8.2.115   MVFR0, Media and VFP Feature Register 0

The MVFR0 characteristics are:

**Purpose**

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

Must be interpreted with MVFR1 and MVFR2.

For general information about the interpretation of the ID registers see Principles of the ID scheme for fields in ID registers on page G8-9332.

**Configurations**

AArch32 System register MVFR0 bits [31:0] are architecturally mapped to AArch64 System register MVFR0_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to MVFR0 are UNDEFINED.

Implemented only if the implementation includes Advanced SIMD and floating-point instructions.

**Attributes**

MVFR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>19</th>
<th>18</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPRound</td>
<td>FPShVec</td>
<td>FPSqrt</td>
<td>FPDivide</td>
<td>FPTrap</td>
<td>FPDP</td>
<td>FPSP</td>
<td>SIMDReg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FPRound, bits [31:28]**

Floating-Point Rounding modes. Indicates whether the floating-point implementation provides support for rounding modes. Defined values are:

- **0b0000**: Not implemented, or only Round to Nearest mode supported, except that Round towards Zero mode is supported for VCVT instructions that always use that rounding mode regardless of the FPSCR setting.
- **0b0001**: All rounding modes supported.

All other values are reserved.

In Armv8-A, the permitted values are **0b0000** and **0b0001**.

**FPShVec, bits [27:24]**

Short Vectors. Indicates whether the floating-point implementation provides support for the use of short vectors. Defined values are:

- **0b0000**: Short vectors not supported.
- **0b0001**: Short vector operation supported.

All other values are reserved.

In Armv8-A, the only permitted value is **0b0000**.

**FPSqrt, bits [23:20]**

Square Root. Indicates whether the floating-point implementation provides support for the ARMv6 VFP square root operations. Defined values are:

- **0b0000**: Not supported in hardware.
- **0b0001**: Supported.

All other values are reserved.
In Armv8-A, the permitted values are \texttt{0b0000} and \texttt{0b0001}.
The VSQRT.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VSQRT.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

**FPDivide**, bits [19:16]

Indicates whether the floating-point implementation provides support for VFP divide operations. Defined values are:
- \texttt{0b0000} Not supported in hardware.
- \texttt{0b0001} Supported.
All other values are reserved.

In Armv8-A, the permitted values are \texttt{0b0000} and \texttt{0b0001}.
The VDIV.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VDIV.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

**FPTrap**, bits [15:12]

Floating Point Exception Trapping. Indicates whether the floating-point implementation provides support for exception trapping. Defined values are:
- \texttt{0b0000} Not supported.
- \texttt{0b0001} Supported.
All other values are reserved.

A value of \texttt{0b0001} indicates that, when the corresponding trap is enabled, a floating-point exception generates an exception.

**FPDP**, bits [11:8]

Double Precision. Indicates whether the floating-point implementation provides support for double-precision operations. Defined values are:
- \texttt{0b0000} Not supported in hardware.
- \texttt{0b0001} Supported, VFPv2.
- \texttt{0b0010} Supported, VFPv3, VFPv4, or Armv8. VFPv3 and Armv8 add an instruction to load a double-precision floating-point constant, and conversions between double-precision and fixed-point values.
All other values are reserved.

In Armv8-A, the permitted values are \texttt{0b0000} and \texttt{0b0010}.

A value of \texttt{0b0001} or \texttt{0b0010} indicates support for all VFP double-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:
- VSQRT.F64 is only available if the Square root field is \texttt{0b0001}.
- VDIV.F64 is only available if the Divide field is \texttt{0b0001}.
- Conversion between double-precision and single-precision is only available if the single-precision field is nonzero.

**FPSP**, bits [7:4]

Single Precision. Indicates whether the floating-point implementation provides support for single-precision operations. Defined values are:
- \texttt{0b0000} Not supported in hardware.
- \texttt{0b0001} Supported, VFPv2.
- \texttt{0b0010} Supported, VFPv3 or VFPv4. VFPv3 adds an instruction to load a single-precision floating-point constant, and conversions between single-precision and fixed-point values.
All other values are reserved.
In Armv8-A, the permitted values are \texttt{0b0000} and \texttt{0b0010}.

A value of \texttt{0b0001} or \texttt{0b0010} indicates support for all VFP single-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- VSQRT.F32 is only available if the Square root field is \texttt{0b0001}.
- VDIV.F32 is only available if the Divide field is \texttt{0b0001}.
- Conversion between double-precision and single-precision is only available if the double-precision field is nonzero.

\textbf{SIMDReg, bits [3:0]}

Advanced SIMD registers. Indicates whether the Advanced SIMD and floating-point implementation provides support for the Advanced SIMD and floating-point register bank. Defined values are:

- \texttt{0b0000} The implementation has no Advanced SIMD and floating-point support.
- \texttt{0b0001} The implementation includes floating-point support with 16 x 64-bit registers.
- \texttt{0b0010} The implementation includes Advanced SIMD and floating-point support with 32 x 64-bit registers.

All other values are reserved.

In Armv8-A, the permitted values are \texttt{0b0000} and \texttt{0b0010}.

\textbf{Accessing MVFR0}

Accesses to this register use the following encodings in the System register encoding space:

\texttt{VMRS<\texttt{c}>>\texttt{q}>>\texttt{Rt},<\texttt{spec_reg}>}

\begin{align*}
\text{reg} & \quad \text{0b}0111 \\
\end{align*}

if PSTATE.EL == EL0 then
UNDfINED;
elsif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
UNDfINED;
elself (ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10 == '00' then
UNDfINED;
elself EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x07);
elself EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x07);
elself EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || HCPR.TCP10 == '1') then
AArch32.TakeHypTrapException(0x08);
elself EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x08);
elself EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
AArch32.TakeHypTrapException(0x08);
elself HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDfINED;
elself
AArch64.AArch32SystemAccessTrap(EL3, 0x07);
elself
R[t] = MVFR0;
elself PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
UNDEFINED;
elif EL2Enabled() && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || HCPRTR.TCP10 == '1') then
    AArch32.TakeHypTrapException(0x00);
elif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x07);
    else
        R[t] = MVFR0;
else if PSTATE.EL == EL3 then
    if CPACR.cp10 == '00' then
        UNDEFINED;
    else
        R[t] = MVFR0;
G8.2.116  MVFR1, Media and VFP Feature Register 1

The MVFR1 characteristics are:

**Purpose**

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

Must be interpreted with MVFR0 and MVFR2.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register MVFR1 bits [31:0] are architecturally mapped to AArch64 System register MVFR1_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to MVFR1 are UNDEFINED.

Implemented only if the implementation includes Advanced SIMD and floating-point instructions.

**Attributes**

MVFR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th></th>
<th>SIMDFMAC</th>
<th>FPHP</th>
<th>SIMDHP</th>
<th>SIMDSP</th>
<th>SIMDInt</th>
<th>SIMDLSS</th>
<th>FPDNaN</th>
<th>FFNZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SIMDFMAC, bits [31:28]**

Advanced SIMD Fused Multiply-Accumulate. Indicates whether the Advanced SIMD implementation provides fused multiply accumulate instructions. Defined values are:

- **0b0000**  Not implemented.
- **0b0001**  Implemented.

All other values are reserved.

In Armv8-A, the permitted values are **0b0000** and **0b0001**.

The Advanced SIMD and floating-point implementations must provide the same level of support for these instructions.

**FPHP, bits [27:24]**

Floating Point Half Precision. Indicates the level of half-precision floating-point support. Defined values are:

- **0b0000**  Not supported.
- **0b0001**  Floating-point half-precision conversion instructions are supported for conversion between single-precision and half-precision.
- **0b0010**  As for **0b0001**, and adds instructions for conversion between double-precision and half-precision.
- **0b0011**  As for **0b0010**, and adds support for half-precision floating-point arithmetic.

All other values are reserved.

In Armv8-A, the permitted values are:

- 0b0000 in an implementation without floating-point support.
- 0b0010 in an implementation with floating-point support that does not include the FEAT_FP16 extension.
• 0b0011 in an implementation with floating-point support that includes the FEAT_FP16 extension.

The level of support indicated by this field must be equivalent to the level of support indicated by the SIMDHP field, meaning the permitted values are:

<table>
<thead>
<tr>
<th>Half Precision instructions supported</th>
<th>FPHP</th>
<th>SIMDHP</th>
</tr>
</thead>
<tbody>
<tr>
<td>No support</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
<tr>
<td>Conversions only</td>
<td>0b0010</td>
<td>0b0001</td>
</tr>
<tr>
<td>Conversions and arithmetic</td>
<td>0b0011</td>
<td>0b0010</td>
</tr>
</tbody>
</table>

SIMDHP, bits [23:20]

Advanced SIMD Half Precision. Indicates the level of half-precision floating-point support. Defined values are:

0b0000  Not supported.
0b0001  SIMD half-precision conversion instructions are supported for conversion between single-precision and half-precision.
0b0010  As for 0b0001, and adds support for half-precision floating-point arithmetic.

All other values are reserved.

In Armv8-A, the permitted values are:

• 0b0000 in an implementation without SIMD floating-point support.
• 0b0001 in an implementation with SIMD floating-point support that does not include the FEAT_FP16 extension.
• 0b0010 in an implementation with SIMD floating-point support that includes the FEAT_FP16 extension.

The level of support indicated by this field must be equivalent to the level of support indicated by the FPHP field, meaning the permitted values are:

<table>
<thead>
<tr>
<th>Half Precision instructions supported</th>
<th>FPHP</th>
<th>SIMDHP</th>
</tr>
</thead>
<tbody>
<tr>
<td>No support</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
<tr>
<td>Conversions only</td>
<td>0b0010</td>
<td>0b0001</td>
</tr>
<tr>
<td>Conversions and arithmetic</td>
<td>0b0011</td>
<td>0b0010</td>
</tr>
</tbody>
</table>

SIMDSP, bits [19:16]

Advanced SIMD Single Precision. Indicates whether the Advanced SIMD and floating-point implementation provides single-precision floating-point instructions. Defined values are:

0b0000  Not implemented.
0b0001  Implemented. This value is permitted only if the SIMDInt field is 0b0001.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0001.

SIMDInt, bits [15:12]

Advanced SIMD Integer. Indicates whether the Advanced SIMD and floating-point implementation provides integer instructions. Defined values are:

0b0000  Not implemented.
0b0001  Implemented.

All other values are reserved.
In Armv8-A, the permitted values are 0b0000 and 0b0001.

SIMDLS, bits [11:8]
Advanced SIMD Load/Store. Indicates whether the Advanced SIMD and floating-point implementation provides load/store instructions. Defined values are:
0b0000 Not implemented.
0b0001 Implemented.
All other values are reserved.
In Armv8-A, the permitted values are 0b0000 and 0b0001.

FPDNaN, bits [7:4]
Default NaN mode. Indicates whether the floating-point implementation provides support only for the Default NaN mode. Defined values are:
0b0000 Not implemented, or hardware supports only the Default NaN mode.
0b0001 Hardware supports propagation of NaN values.
All other values are reserved.
In Armv8-A, the permitted values are 0b0000 and 0b0001.

FPFtZ, bits [3:0]
Flush to Zero mode. Indicates whether the floating-point implementation provides support only for the Flush-to-Zero mode of operation. Defined values are:
0b0000 Not implemented, or hardware supports only the Flush-to-Zero mode of operation.
0b0001 Hardware supports full denormalized number arithmetic.
All other values are reserved.
In Armv8-A, the permitted values are 0b0000 and 0b0001.

Accessing MVFR1
Accesses to this register use the following encodings in the System register encoding space:

\[
\text{VMRS}\{<c>\}{<q>}\ <Rt>, <\text{spec}\_\text{reg}>
\]

```
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    UNDEFINED;
  elsif EL2Enabled() && ELUsingAArch32(EL2) && SCR.NS == '1' && NSACR.cp10 == '0' then
    UNDEFINED;
  elsif EL2Enabled() && ELUsingAArch32(EL2) && (ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.cp10 == '00' then
    UNDEFINED;
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x07);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x07);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || CPACR.TCPI0 == '1') then
    AArch32.TakeHypTrapException(0x08);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x08);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID3 == '1' then
    AArch32.TakeHypTrapException(0x08);
```
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x07);
    else
        R[t] = MVFR1;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED “EL3 trap priority when SDD == ’1’” && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
        UNDEFINED;
elsif EL2Enabled() && ((ELUsingAArch32(EL3) && SCR.NS == '1' && NSACR.cp10 == '0') || HCPTR.TCP10 == '1') then
    AArch32.TakeHypTrapException(0x00);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TFP == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x07);
else
    R[t] = MVFR1;
elsif PSTATE.EL == EL3 then
    if CPACR.cp10 == '00' then
        UNDEFINED;
    else
        R[t] = MVFR1;
**G8.2.117 MVFR2, Media and VFP Feature Register 2**

The MVFR2 characteristics are:

**Purpose**

Describes the features provided by the AArch32 Advanced SIMD and Floating-point implementation.

Must be interpreted with MVFR0 and MVFR1.

For general information about the interpretation of the ID registers see *Principles of the ID scheme for fields in ID registers* on page G8-9332.

**Configurations**

AArch32 System register MVFR2 bits [31:0] are architecturally mapped to AArch64 System register MVFR2_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to MVFR2 are UNDEFINED.

Implemented only if the implementation includes Advanced SIMD and floating-point instructions.

**Attributes**

MVFR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>FPMisc</td>
<td>SIMDMisc</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**FPMisc, bits [7:4]**

Indicates whether the floating-point implementation provides support for miscellaneous VFP features.

- **0b0000**: Not implemented, or no support for miscellaneous features.
- **0b0001**: Support for Floating-point selection.
- **0b0010**: As 0b0001, and Floating-point Conversion to Integer with Directed Rounding modes.
- **0b0011**: As 0b0010, and Floating-point Round to Integer Floating-point.
- **0b0100**: As 0b0011, and Floating-point MaxNum and MinNum.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0100.

**SIMDMisc, bits [3:0]**

Indicates whether the Advanced SIMD implementation provides support for miscellaneous Advanced SIMD features.

- **0b0000**: Not implemented, or no support for miscellaneous features.
- **0b0001**: Floating-point Conversion to Integer with Directed Rounding modes.
- **0b0010**: As 0b0001, and Floating-point Round to Integer Floating-point.
- **0b0011**: As 0b0010, and Floating-point MaxNum and MinNum.

All other values are reserved.

In Armv8-A, the permitted values are 0b0000 and 0b0011.
Accessing MVFR2

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{VMRS}(<c>){q} <Rt>, <\text{spec\_reg}>
\]

\[
\begin{align*}
\text{reg} &= 0b0101
\end{align*}
\]
G8.2.118   NMRR, Normal Memory Remap Register

The NMRR characteristics are:

**Purpose**

Provides additional mapping controls for memory regions that are mapped as Normal memory by their entry in the PRRR.

Used in conjunction with the PRRR.

**Configurations**

AArch32 System register NMRR bits [31:0] are architecturally mapped to AArch64 System register MAIR_EL1[63:32] when EL3 is not implemented or EL3 is using AArch64.

AArch32 System register NMRR bits [31:0] are architecturally mapped to AArch32 System register MAIR1[31:0] when EL3 is not implemented or EL3 is using AArch64.

AArch32 System register NMRR bits [31:0] (NMRR_S) are architecturally mapped to AArch32 System register MAIR1[31:0] (MAIR1_S) when EL3 is using AArch32.

AArch32 System register NMRR bits [31:0] (NMRR_NS) are architecturally mapped to AArch32 System register MAIR1[31:0] (MAIR1_NS) when EL3 is using AArch32.

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to NMRR are UNDEFINED.

MAIR1 and NMRR are the same register, with a different view depending on the value of TTBCR.EAE:

- When it is set to 0, the register is as described in NMRR.
- When it is set to 1, the register is as described in MAIR1.

**Attributes**

NMRR is a 32-bit register.

**Field descriptions**

*When TTBCR.EAE == 0:*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| OR7 | OR6 | OR5 | OR4 | OR3 | OR2 | OR1 | OR0 | IR7 | IR6 | IR5 | IR4 | IR3 | IR2 | IR1 | IR0 |

**OR<n>**, bits [2n+17:2n+16], for n = 7 to 0

Outer Cacheable property mapping for memory attributes n, if the region is mapped as Normal memory by the PRRR.TR<n> entry. n is the value of the TEX[0], C, and B bits concatenated.

- 0b00  Region is Non-cacheable.
- 0b01  Region is Write-Back, Write-Allocate.
- 0b10  Region is Write-Through, no Write-Allocate.
- 0b11  Region is Write-Back, no Write-Allocate.

The meaning of the field with n = 6 is IMPLEMENTATION DEFINED and might differ from the meaning given here. This is because the meaning of the attribute combination \{TEX[0] = 1, C = 1, B = 0\} is IMPLEMENTATION DEFINED.

When FEAT_XS is implemented, stage 1 Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
IR\textless{}n\textgreater{}, bits \texttt{[2n+1:2n]}, for \(n = 7\) to 0

Inner Cacheable property mapping for memory attributes \(n\), if the region is mapped as Normal memory by the PRRR.TR\textless{}n\textgreater{} entry. \(n\) is the value of the TEX[0], C, and B bits concatenated.

\begin{itemize}
  \item \texttt{0b00} Region is Non-cacheable.
  \item \texttt{0b01} Region is Write-Back, Write-Allocate.
  \item \texttt{0b10} Region is Write-Through, no Write-Allocate.
  \item \texttt{0b11} Region is Write-Back, no Write-Allocate.
\end{itemize}

The meaning of the field with \(n = 6\) is IMPLEMENTATION DEFINED and might differ from the meaning given here. This is because the meaning of the attribute combination \{TEX[0] = 1, C = 1, B = 0\} is IMPLEMENTATION DEFINED.

When \texttt{FEAT\_XS} is implemented, stage 1 Inner Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing NMRR

Accesses to this register use the following encodings in the System register encoding space:

\[ \text{MRC}\{<c>\}\{<q>\} <coproc>, \{#\}<opc1>, <Rt>, <CRn>, <CRm>{{\{, \{#\}<opc2}\}} } \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{0b111}</td>
<td>\texttt{0b000}</td>
<td>\texttt{0b1010}</td>
<td>\texttt{0b0010}</td>
<td>\texttt{0b001}</td>
</tr>
</tbody>
</table>

\[
\text{if PSTATE.EL} = \text{EL0 then UNDEFINED;} \\
\text{elsif PSTATE.EL} = \text{EL1 then} \\
\text{if EL2Enabled()} \&\& \text{ELUsingAArch32(EL2)} \&\& \text{HSTR.EL2.TJ0} = '1' then \\
\text{AArch64.AArch32SystemAccessTrap(EL2, 0x03);} \\
\text{elsif EL2Enabled()} \&\& \text{ELUsingAArch32(EL2)} \&\& \text{HSTR.T10} = '1' then \\
\text{AArch32.TakeHypTrapException(0x03);} \\
\text{elsif EL2Enabled()} \&\& \text{ELUsingAArch32(EL2)} \&\& \text{HCR_EL2.TRVM} = '1' then \\
\text{AArch64.AArch32SystemAccessTrap(EL2, 0x03);} \\
\text{elsif EL2Enabled()} \&\& \text{ELUsingAArch32(EL2)} \&\& \text{HCR.TRVM} = '1' then \\
\text{AAArch32.TakeHypTrapException(0x03);} \\
\text{elsif HaveEL(EL3)} \&\& \text{ELUsingAArch32(EL3)} then \\
\text{if TTBCR.EAE} = '1' then \\
\text{R}[t] = \text{MAIR1\_NS; } \\
\text{else} \\
\text{R}[t] = \text{NMRR\_NS; } \\
\text{else} \\
\text{if TTBCR.EAE} = '1' then \\
\text{R}[t] = \text{MAIR1; } \\
\text{else} \\
\text{R}[t] = \text{NMRR; } \\
\text{elsif PSTATE.EL} = \text{EL2 then} \\
\text{if HaveEL(EL3)} \&\& \text{ELUsingAArch32(EL3)} then \\
\text{if TTBCR.EAE} = '1' then \\
\text{R}[t] = \text{MAIR1\_NS; } \\
\text{else} \\
\text{R}[t] = \text{NMRR\_NS; } \\
\text{else} \\
\text{if TTBCR.EAE} = '1' then \\
\text{R}[t] = \text{MAIR1; } \\
\text{else} \\
\text{R}[t] = \text{NMRR; } \\
\text{elsif PSTATE.EL} = \text{EL3 then}
\]
if \( \text{TBCR.EAE} = '1' \) then
  if \( \text{SCR.NS} = '0' \) then
    \( \text{R}[t] = \text{MAIR1}_S \);
  else
    \( \text{R}[t] = \text{MAIR1}_NS \);
  else
    if \( \text{SCR.NS} = '0' \) then
      \( \text{R}[t] = \text{NMRR}_S \);
    else
      \( \text{R}[t] = \text{NMRR}_NS \);

\[
\text{MCR}\{<c>\}<q> \text{<coproc>, \(#\text{<opc1>, <Rt>, <CRn>}, <CRm}\{, \(#\text{<opc2>}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if \( \text{PSTATE.EL} = \text{EL0} \) then
  \text{UNDEFINED};
elsif \( \text{PSTATE.EL} = \text{EL1} \) then
  if \( \text{EL2Enabled()} \) && \( \text{ELUsingAArch32(EL2)} \) && \( \text{HSTR.EL2.T10} = '1' \) then
    \text{AArch64.AArch32SystemAccessTrap(EL2, 0x03)};
  elsif \( \text{EL2Enabled()} \) && \( \text{ELUsingAArch32(EL2)} \) && \( \text{HSTR.EL2.T10} = '1' \) then
    \text{AArch32.TakeHypTrapException(0x03)};
  elsif \( \text{EL2Enabled()} \) && \( \text{ELUsingAArch32(EL2)} \) && \( \text{HCR.EL2.TVM} = '1' \) then
    \text{AArch64.AArch32SystemAccessTrap(EL2, 0x03)};
  elsif \( \text{EL2Enabled()} \) && \( \text{ELUsingAArch32(EL2)} \) && \( \text{HCR.TVM} = '1' \) then
    \text{AArch32.TakeHypTrapException(0x03)};
elsif \( \text{PSTATE.EL} = \text{EL2} \) then
  if \( \text{AArch32.SystemAccessTrap(EL2, 0x03)} \) then
    \text{MAIR1}_S = \( \text{R}[t] \);
  else
    \( \text{NMRR}_S = \text{R}[t] \);
else
  if \( \text{TTBCR.EAE} = '1' \) then
    \( \text{R}[t] = \text{MAIR1}_S \);
  else
    \( \text{R}[t] = \text{NMRR}_S \);
else
  if \( \text{PSTATE.EL} = \text{EL3} \) then
    if \( \text{EL2Enabled()} \) && \( \text{ELUsingAArch32(EL3)} \) then
      if \( \text{TTBCR.EAE} = '1' \) then
        \( \text{MAIR1}_NS = \text{R}[t] \);
      else
        \( \text{NMRR}_S = \text{R}[t] \);
    else
      if \( \text{TTBCR.EAE} = '1' \) then
        \( \text{MAIR1}_NS = \text{R}[t] \);
      else
        \( \text{NMRR}_S = \text{R}[t] \);
    if \( \text{SCR.NS} = '0' \) \&\& \( \text{CP15SDISABLE} \) = \text{HIGH} then
      \text{UNDEFINED};
    elsif \( \text{SCR.NS} = '0' \) \&\& \( \text{CP15SDISABLE2} \) = \text{HIGH} then
      \text{UNDEFINED};
    else
      if \( \text{TTBCR.EAE} = '1' \) then
        if \( \text{SCR.NS} = '0' \) then
          \( \text{MAIR1}_S = \text{R}[t] \);
        else
          \( \text{MAIR1}_NS = \text{R}[t] \);
        else
          if \( \text{SCR.NS} = '0' \) then
            \( \text{NMRR}_S = \text{R}[t] \);
else
    NMRR_NS = R[t];
G8.2.119 NSACR, Non-Secure Access Control Register

The NSACR characteristics are:

**Purpose**

When EL3 is implemented and can use AArch32, defines the Non-secure access permissions to Trace, Advanced SIMD and floating-point functionality. Also includes IMPLEMENTATION DEFINED bits that can define Non-secure access permissions for IMPLEMENTATION DEFINED functionality.

**Configurations**

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to NSACR are UNDEFINED.

--- Note ---

In AArch64 state, the NSACR controls are replaced by controls in CPTR_EL3.

**Attributes**

NSACR is a 32-bit register.

**Field descriptions**

If EL3 is implemented and is using AArch64 then:

- Any read of the NSACR from Non-secure EL2 or Non-secure EL1 returns a value of 0x00000C00.
- Any read or write to NSACR from Secure EL1 is trapped as an exception to EL3.

If EL3 is not implemented, then any read of the NSACR from EL2 or EL1 returns a value of 0x00000C00.

**Bits [31:21]**

Reserved, RES0.

**NSTRCDIS, bit [20]**

Disables Non-secure System register accesses to all implemented trace registers.

- **0**: This control has no effect on:
  - System register access to implemented trace registers.
  - The behavior of CPACR.TRCDIS and HCPR.TTA.

- **1**: Non-secure System register accesses to all implemented trace registers are disabled, meaning:
  - CPACR.TRCDIS behaves as RAO/WI in Non-secure state, regardless of its actual value.
  - HCPR.TTA behaves as RAO/WI, regardless of its actual value.

The implementation of this field must correspond to the implementation of the CPACR.TRCDIS field:

- If CPACR.TRCDIS is RAZ/WI, this field is RAZ/WI.
- If CPACR.TRCDIS is RW, this field is RW.
Note

- The ETMv4 architecture and ETE do not permit EL0 to access the trace registers. If the trace unit implements FEAT_ETMv4 or FEAT_ETE, EL0 accesses to the trace registers are UNDEFINED.
- The Arm architecture does not provide Non-secure access controls on trace register accesses through the optional memory-mapped external debug interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

Bit [19]
Reserved, RES0.

IMPLEMENTATION DEFINED, bits [18:16]
IMPLEMENTATION DEFINED.

NSASEDIS, bit [15]

Disables Non-secure access to the Advanced SIMD functionality.

0b0 This control has no effect on:
- Non-secure access to Advanced SIMD functionality.
- The behavior of CPACR.ASEDIS and HCPTR.TASE.

0b1 Non-secure access to the Advanced SIMD functionality is disabled, meaning:
- CPACR.ASEDIS behaves as RAO/WI in Non-secure state, regardless of its actual value.
- HCPTR.TASE behaves as RAO/WI, regardless of its actual value.

The implementation of this field must correspond to the implementation of the CPACR.ASEDIS field:
- If CPACR.ASEDIS is RES0, this field is RES0. If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.
- If CPACR.ASEDIS is RAZ/WI, this field is RAZ/WI.
- If CPACR.ASEDIS is RW, this field is RW.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

Bits [14:12]
Reserved, RES0.

cp11, bit [11]
The value of this field is ignored. If this field is programmed with a different value to the cp10 field then this field is UNKNOWN on a direct read of the NSACR.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally UNKNOWN value.
**cp10, bit [10]**

Enable Non-secure access to the Advanced SIMD and floating-point features. Possible values of the fields are:

- **0b0**: Advanced SIMD and floating-point features can be accessed only from Secure state. Any attempt to access this functionality from Non-secure state is **UNDEFINED**. When the PE is in Non-secure state:
  - The CPACR.{cp11, cp10} fields ignore writes and read as 0b00, access denied.
  - The HCPTR.{TCP11, TCP10} fields behave as RAO/WI, regardless of their actual values.

- **0b1**: Advanced SIMD and floating-point features can be accessed from both Security states. If Non-secure access to the Advanced SIMD and floating-point functionality is enabled, the CPACR must be checked to determine the level of access that is permitted.

The Advanced SIMD and floating-point features controlled by these fields are:

- Execution of any floating-point or Advanced SIMD instruction.
- Any access to the Advanced SIMD and floating-point registers D0-D31 and their views as S0-S31 and Q0-Q15.
- Any access to the FPSCR, FPSID, MVFR0, MVFR1, MVFR2, or FPEXC System registers.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is **RES0**.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally **UNKNOWN** value.

**Bits [9:0]**

Reserved, **RES0**.

**Accessing NSACR**

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC\{<c>\}{<q>} <coproc>, \{#\}<opc1>, <Rt>, <CRn>, <CRm>\{, \{#\}<opc2>\} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL3) && SCR_EL3.<NS> == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif !HaveEL(EL3) || (!ELUsingAArch32(EL3) && SCR_EL3.<NS> == '1') then
        R[t] = Zeros(20):'1100':Zeros(8);
    else
        R[t] = NSACR;
    elsif PSTATE.EL == EL2 then
        if !HaveEL(EL3) || (!ELUsingAArch32(EL3) && SCR_EL3.<NS> == '1') then
            R[t] = Zeros(20):'1100':Zeros(8);
        else
```
R[t] = NSACR;
elsif PSTATE.EL == EL3 then
    R[t] = NSACR;

\[
MCR{<c>}{<q>}, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if CP15SDISABLE2 == HIGH then
        UNDEFINED;
else
    NSACR = R[t];
G8.2.120 PAR, Physical Address Register

The PAR characteristics are:

**Purpose**

Returns the output address (OA) from an Address translation instruction that executed successfully, or fault information if the instruction did not execute successfully.

**Configurations**

AArch32 System register PAR bits [63:0] are architecturally mapped to AArch64 System register PAR_EL1[63:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to PAR are UNDEFINED.

PAR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits[31:0] and do not modify bits[63:32].

The Configurations section specifies the cases where each PAR format is used.

PAR is accessed as a 32-bit value:

- When the PE is not in Hyp mode and is using the Short-descriptor translation table format.
- When the PE is in Hyp mode and executes an ATS12NSOPR, ATS12NSOPW, ATS12NSOUR, or ATS12NSOUW instruction and the value of HCR.VM is 0 and the value of TTBCR.EAE is 0.

In these cases, PAR[63:32] is RES0.

Otherwise, the PAR is accessed as a 64-bit value, if any of the following is true:

- When using the Long-descriptor translation table format.
- If the stage 1 address translation is disabled and TTBCR.EAE is set to 1.
- In an implementation that includes EL2, for the result of an ATS1Cxx instruction performed from Hyp mode.

For PL1&0 stage 1 translations, TTBCR.EAE selects the translation table format.

**Attributes**

PAR is a 64-bit register.

**Field descriptions**

*When the instruction returned a 32-bit value to the PAR, PAR.F==0:*

```
  63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8
  7  6  5  4  3  2  1  0
    RES0
      12   11  10  9  8  7  6  5  4  3  2  1  0
      PA   NS  SH  SS  F
    LPAE  NOS  Outer[1:0] Inner[2:0] IMPLEMENTATION DEFINED
```

This section describes the register value returned by the successful execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.
On a successful conversion, the PAR can return a value that indicates the resulting attributes, rather than the values that appear in the Translation table descriptors. More precisely:

- Memory attribute fields are permitted to report the resulting attributes, as determined by any permitted implementation choices and any applicable configuration bits, instead of reporting the values that appear in the Translation table descriptors. This applies to the NOS, SH, Inner, and Outer fields.
- See the NS bit description for constraints on the value it returns.

**Bits [63:32]**

Reserved, RES0.

**PA, bits [31:12]**

Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits [31:12].

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**LPAE, bit [11]**

When updating the PAR with the result of the translation operation, this bit is set as follows:

- 0b0 Short-descriptor translation table format used. This means the PAR returned a 32-bit value.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**NOS, bit [10]**

Not Outer Shareable. When the returned value of PAR.SH is 1, indicates the Shareability attribute for the physical memory region:

- 0b0 Memory region is Outer Shareable.
- 0b1 Memory region is Inner Shareable.

When the returned value of PAR.SH is 0 the value returned to this field is UNKNOWN.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**NS, bit [9]**

Non-secure. The NS attribute for a translation table entry from a Secure translation regime.

For a result from a Secure translation regime, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IMPLEMENTATION DEFINED, bit [8]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
SH, bit [7]
Shareability. Indicates whether the physical memory region is Non-shareable:

0b0  Memory is Non-shareable.
0b1  Memory is shareable, and PAR.NOS indicates whether the region is Outer Shareable or Inner Shareable.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally UNKNOWN value.

Inner[2:0], bits [6:4]
Inner cacheability attribute for the region. Permitted values are:

0b000  Non-cacheable.
0b001  Device-nGnRnE.
0b011  Device-nGnRE.
0b101  Write-Back, Write-Allocate.
0b110  Write-Through.
0b111  Write-Back, no Write-Allocate.

The values 0b10 and 0b100 are reserved.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally UNKNOWN value.

Outer[1:0], bits [3:2]
Outer cacheability attribute for the region. Permitted values are:

0b00  Non-cacheable.
0b01  Write-Back, Write-Allocate.
0b10  Write-Through, no Write-Allocate.
0b11  Write-Back, no Write-Allocate.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:
* On a Warm reset, this field resets to an architecturally UNKNOWN value.

SS, bit [1]
Supersection. Used to indicate if the result is a Supersection:

0b0  Result is not a Supersection. PAR[31:12] contains OA[31:12].
0b1  Result is a Supersection, and:
    * PAR[15:12] contains 0b0000.

If an implementation supports less than 40 bits of physical address, the bits in the PAR field that correspond to physical address bits that are not implemented are UNKNOWN.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**F, bit [0]**
Indicates whether the instruction performed a successful address translation.
- 0b0 Address translation completed successfully.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*When the instruction returned a 32-bit value to the PAR, PAR.F==1:*

```
  63       32
   RES0    RES0
  24       15
 IMPLEMENTATION DEFINED  RES0
  12       5
   RES0     FS[4:0]
  10       0
   LPAE FS[5]
```

This section describes the register value returned by a fault on the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

**Bits [63:32]**
Reserved, RES0.

**IMPLEMENTATION DEFINED, bits [31:16]**
IMPLEMENTATION DEFINED.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [15:12]**
Reserved, RES0.

**LPAE, bit [11]**
When updating the PAR with the result of the translation operation, this bit is set as follows:
- 0b0 Short-descriptor translation table format used. This means the PAR returned a 32-bit value.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [10:7]**
Reserved, RES0.

**FS[5], bit [6]**
Fault status bits, External abort type. Provides an IMPLEMENTATION DEFINED classification of an External abort. Values are as in the DFSR.ExT field when using the Short-descriptor translation table format.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
FS[4:0], bits [5:1]

Fault status bits. Values are as in the DFSR.FS field when using the Short-descriptor translation table format.

- 0b00001  Alignment fault.
- 0b00011  Access flag fault, level 1.
- 0b00100  Fault on instruction cache maintenance.
- 0b00101  Translation fault, level 1.
- 0b00110  Access flag fault, level 2.
- 0b00111  Translation fault, level 2.
- 0b01001  Domain fault, level 1.
- 0b01011  Domain fault, level 2.
- 0b01100  Synchronous External abort, on translation table walk, level 1.
- 0b01101  Permission fault, level 1.
- 0b01110  Synchronous External abort, on translation table walk, level 2.
- 0b01111  Permission fault, level 2.
- 0b10000  TLB conflict abort.
- 0b11001  When FEAT_RAS is not implemented: Synchronous parity or ECC error on memory access, not on translation table walk.
- 0b11100  When FEAT_RAS is not implemented: Synchronous parity or ECC error on translation table walk, level 1.
- 0b11110  When FEAT_RAS is not implemented: Synchronous parity or ECC error on translation table walk, level 2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an UNKNOWN value.

F, bit [0]

Indicates whether the instruction performed a successful address translation.

- 0b1  Address translation aborted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When the instruction returned a 64-bit value to the PAR, PAR.F==0:

This section describes the register value returned by the successful execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

On a successful conversion, the PAR can return a value that indicates the resulting attributes, rather than the values that appear in the Translation table descriptors. More precisely:

- Memory attribute fields are permitted to report the resulting attributes, as determined by any permitted implementation choices and any applicable configuration bits, instead of reporting the values that appear in the Translation table descriptors. This applies to the ATTR and SH fields.
• See the NS bit description for constraints on the value it returns.

**ATTR, bits [63:56]**

Memory attributes for the returned output address. This field uses the same encoding as the Attr<n> fields in MAIR0 and MAIR1.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [55:40]**

Reserved, RES0.

**PA, bits [39:12]**

Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits[39:12].

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**LPAE, bit [11]**

When updating the PAR with the result of the translation operation, this bit is set as follows:
0b1 Long-descriptor translation table format used. This means the PAR returned a 64-bit value.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IMPLEMENTATION DEFINED, bit [10]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**NS, bit [9]**

Non-secure. The NS attribute for a translation table entry from a Secure translation regime.

For a result from a Secure translation regime, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**SH, bits [8:7]**

Shareability attribute, for the returned output address. Permitted values are:

0b00 Non-shareable.
0b10 Outer Shareable.
0b11 Inner Shareable.

The value 0b01 is reserved.

**Note**

This field returns the value 0b10 for:
• Any type of Device memory.
• Normal memory with both Inner Non-cacheable and Outer Non-cacheable attributes.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [6:1]**
Reserved, RES0.

**F, bit [0]**
Indicates whether the instruction performed a successful address translation.

0b0 Address translation completed successfully.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

*When the instruction returned a 64-bit value to the PAR, PAR.F==1:*

This section describes the register value returned by a fault on the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

**IMPLEMENTATION DEFINED, bits [63:56]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IMPLEMENTATION DEFINED, bits [55:52]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IMPLEMENTATION DEFINED, bits [51:48]**

IMPLEMENTATION DEFINED.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [47:12]**
Reserved, RES0.
LPAE, bit [11]

When updating the PAR with the result of the translation operation, this bit is set as follows:

0b1  Long-descriptor translation table format used. This means the PAR returned a 64-bit value.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [10]

Reserved, RES0.

FSTAGE, bit [9]

Indicates the translation stage at which the translation aborted:

0b0  Translation aborted because of a fault in the stage 1 translation.
0b1  Translation aborted because of a fault in the stage 2 translation.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S2WLK, bit [8]

If this bit is set to 1, it indicates the translation aborted because of a stage 2 fault during a stage 1 translation table walk.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

FST, bits [6:1]

Fault status field. Values are as in the DFSR.STATUS and IFSR.STATUS fields when using the Long-descriptor translation table format.

0b000000  Address size fault in translation table base register.
0b000001  Address size fault, level 1.
0b000010  Address size fault, level 2.
0b000011  Address size fault, level 3.
0b000101  Translation fault, level 1.
0b000110  Translation fault, level 2.
0b000111  Translation fault, level 3.
0b001001  Access flag fault, level 1.
0b001010  Access flag fault, level 2.
0b001011  Access flag fault, level 3.
0b001101  Permission fault, level 1.
0b001110  Permission fault, level 2.
0b001111  Permission fault, level 3.
0b010101  Synchronous External abort on translation table walk, level 1.
0b010110  Synchronous External abort on translation table walk, level 2.
0b010111  Synchronous External abort on translation table walk, level 3.
0b011101  When FEAT_RAS is not implemented:
    Synchronous parity or ECC error on memory access on translation table walk, level 1.
0b011110  When FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access on translation table walk, level 2.

0b01111

When FEAT_RAS is not implemented:
Synchronous parity or ECC error on memory access on translation table walk, level 3.

0b11000

TLB conflict abort.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [0]

Indicates whether the instruction performed a successful address translation.

0b1

Address translation aborted.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PAR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.EL2.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.EL2.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    R[t] = PAR_NS<31:0>;
  else
    R[t] = PAR<31:0>;
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    R[t] = PAR_NS<31:0>;
  else
    R[t] = PAR<31:0>;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    R[t] = PAR_S<31:0>;
  else
    R[t] = PAR_NS<31:0>;
else
  if SCR.NS == '0' then
    R[t] = PAR_S<31:0>;
  else
    R[t] = PAR_NS<31:0>;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.EL2.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.EL2.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    R[t] = PAR_NS<31:0>;
  else
    R[t] = PAR<31:0>;
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    R[t] = PAR_NS<31:0>;
  else
    R[t] = PAR<31:0>;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    R[t] = PAR_S<31:0>;
  else
    R[t] = PAR_NS<31:0>;
else
  if SCR.NS == '0' then
    R[t] = PAR_S<31:0>;
  else
    R[t] = PAR_NS<31:0>;}
AArch32 System Register Descriptions

8.2 General system control registers

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & ELUsingAArch32(EL2) & HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T7 == '1' then
    AArch32.TakeHypTrapException(0x03);
elif HaveEL(EL3) & ELUsingAArch32(EL3) then
    PAR_NS = ZeroExtend(R[t]);
  else
    PAR = ZeroExtend(R[t]);
elif PSTATE.EL == EL2 then
  if HaveEL(EL3) & ELUsingAArch32(EL3) then
    PAR_NS = ZeroExtend(R[t]);
  else
    PAR = ZeroExtend(R[t]);
elif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    PAR_S = ZeroExtend(R[t]);
  else
    PAR_NS = ZeroExtend(R[t]);

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0111</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & ELUsingAArch32(EL2) & HSTR_EL2.T7 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T7 == '1' then
    AArch32.TakeHypTrapException(0x04);
elif HaveEL(EL3) & ELUsingAArch32(EL3) then
    (R[t2], R[t]) = (PAR_NS<63:32>, PAR_NS<31:0>);
  else
    (R[t2], R[t]) = (PAR<63:32>, PAR<31:0>);
elif PSTATE.EL == EL2 then
  if HaveEL(EL3) & ELUsingAArch32(EL3) then
    (R[t2], R[t]) = (PAR_NS<63:32>, PAR_NS<31:0>);
  else
    (R[t2], R[t]) = (PAR<63:32>, PAR<31:0>);
elif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    (R[t2], R[t]) = (PAR_S<63:32>, PAR_S<31:0>);
  else
    (R[t2], R[t]) = (PAR_NS<63:32>, PAR_NS<31:0>);

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0111</td>
<td>0b0000</td>
</tr>
</tbody>
</table>
PAR_NS = R[t2]:R[t];
else
  PAR = R[t2]:R[t];
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    PAR_NS = R[t2]:R[t];
  else
    PAR = R[t2]:R[t];
  endif
else
  PAR = R[t2]:R[t];
endif
if SCR.NS == '0' then
  PAR_NS = R[t2]:R[t];
else
  PAR_NS = R[t2]:R[t];
endif
G8.2.121 PRRR, Primary Region Remap Register

The PRRR characteristics are:

**Purpose**

Controls the top level mapping of the TEX[0], C, and B memory region attributes.

**Configurations**

AArch32 System register PRRR bits [31:0] are architecturally mapped to AArch64 System register MAIR_EL1[31:0] when EL3 is not implemented or EL3 is using AArch64.

AArch32 System register PRRR bits [31:0] are architecturally mapped to AArch32 System register MAIR0[31:0] when EL3 is not implemented or EL3 is using AArch64.

AArch32 System register PRRR bits [31:0] (PRRR_S) are architecturally mapped to AArch32 System register MAIR0[31:0] (MAIR0_S) when EL3 is using AArch64.

AArch32 System register PRRR bits [31:0] (PRRR_NS) are architecturally mapped to AArch32 System register MAIR0[31:0] (MAIR0_NS) when EL3 is using AArch32.

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to PRRR are UNDEFINED.

MAIR0 and PRRR are the same register, with a different view depending on the value of TTBCR.EAE:

- When it is set to 0, the register is as described in PRRR.
- When it is set to 1, the register is as described in MAIR0.

**Attributes**

PRRR is a 32-bit register.

**Field descriptions**

*When TTBCR.EAE == 0:*

NOS<n>, bit [n+24], for n = 7 to 0

Not Outer Shareable. NOS<n> is the Outer Shareable property for memory attributes n, if the region is mapped as Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, and the appropriate PRRR. {NS0, NS1} field identifies the region as shareable. n is the value of the concatenation of the {TEX[0], C, B} bits from the Translation table descriptor. The possible values of each NOS<n> field other than NOS6 are:

- 0b0 Memory region is Outer Shareable.
- 0b1 Memory region is Inner Shareable.

The value of this bit is ignored if the region is:

- Device memory
- Normal memory that is at least one of:
  - Inner Non-cacheable, Outer Non-cacheable.
  - Identified by the appropriate PRRR. {NS0, NS1} field as Non-shareable.

The meaning of the NOS6 field is IMPLEMENTATION DEFINED.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [23:20]
Reserved, RES0.

NS1, bit [19]
Mapping of S = 1 attribute for Normal memory regions. This field is used in determining the Shareability of a memory region that is mapped to Normal memory and both:
• Is not Inner Non-cacheable, Outer Non-cacheable.
• Has the S bit in the Translation table descriptor set to 1.
0b0 Region is Non-shareable.
0b1 Region is shareable. The value of the appropriate PRRR.NOS<n> field determines whether the region is Inner Shareable or Outer Shareable.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

NS0, bit [18]
Mapping of S = 0 attribute for Normal memory regions. This field is used in determining the Shareability of a memory region that is mapped to Normal memory and both:
• Is not Inner Non-cacheable, Outer Non-cacheable.
• Has the S bit in the Translation table descriptor set to 0.
0b0 Region is Non-shareable.
0b1 Region is shareable. The value of the appropriate PRRR.NOS<n> field determines whether the region is Inner Shareable or Outer Shareable.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

DS1, bit [17]
Mapping of S = 1 attribute for Device memory. From Armv8, all types of Device memory are Outer Shareable, and therefore this bit is RES1.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

DS0, bit [16]
Mapping of S = 0 attribute for Device memory. From Armv8, all types of Device memory are Outer Shareable, and therefore this bit is RES1.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

TR<n>, bits [2n+1:2n], for n = 7 to 0
TR<n> is the primary TEX mapping for memory attributes n, and defines the mapped memory type for a region with attributes n. n is the value of the concatenation of the {TEX[0], C, B} bits from the Translation table descriptor. The possible values for each field other than TR6 are:
0b00 Device-nGnRnE memory
0b01 Device-nGnRE memory
0b10 Normal memory
The value 0b11 is reserved. The effect of programming a field to 0b11 is CONSTRAINED UNPREDICTABLE.
The meaning of the TR6 field is IMPLEMENTATION DEFINED.
When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Accessing PRRR**

Accesses to this register use the following encodings in the System register encoding space:

$$MRC\{c\}\{q\} <\text{coproc}, \{\#}<\text{opc1}>, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>\{, \{\#}<\text{opc2}>\}$$

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & ELUsingAArch32(EL2) & HSTR.EL2.T10 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.EL2.T10 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR.EL2.TRVM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR.TRVM == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) & ELUsingAArch32(EL3) then
  if TTBCR.EAE == '1' then
    R[t] = MAIR0_NS;
  else
    R[t] = PRRR_NS;
  else
    R[t] = MAIR0;
  elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) & ELUsingAArch32(EL3) then
      if TTBCR.EAE == '1' then
        R[t] = MAIR0_NS;
      else
        R[t] = PRRR_NS;
      else
        if TTBCR.EAE == '1' then
          R[t] = MAIR0_NS;
        else
          if TTBCR.EAE == '1' then
            R[t] = MAIR0_NS;
          else
            if TTBCR.EAE == '1' then
              R[t] = MAIR0_NS;
            else
              if TTBCR.EAE == '1' then
                R[t] = MAIR0_NS;
              else
                R[t] = PRRR_NS;
              else
                R[t] = PRRR_NS;
              else
                R[t] = PRRR_NS;
              else
                R[t] = PRRR_NS;
              else
                R[t] = PRRR_NS;
              else
                R[t] = PRRR_NS;
            else
              R[t] = PRRR_NS;
          else
            R[t] = PRRR_NS;
        else
          R[t] = PRRR_NS;
      else
        R[t] = MAIR0_NS;
    else
      R[t] = MAIR0_NS;
else
  R[t] = PRRR_NS;
G8.2 General system control registers

**MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}**

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1010</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsf PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.TI0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsf EL2Enabled() && ELUsingAArch32(EL2) && HSTR.TI0 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsf EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsf EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
    AArch32.TakeHypTrapException(0x03);
elsf HaveEL(EL3) && ELUsingAArch32(EL3) then
  if TTBCR.EAE == '1' then
    MAIR0_NS = R[t];
  else
    PRRR_NS = R[t];
  else
    if TTBCR.EAE == '1' then
      MAIR0 = R[t];
    else
      PRRR = R[t];
elsf PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    if TTBCR.EAE == '1' then
      MAIR0_NS = R[t];
    else
      PRRR_NS = R[t];
  else
    if TTBCR.EAE == '1' then
      MAIR0 = R[t];
    else
      PRRR = R[t];
elsf PSTATE.EL == EL3 then
  if SCR.NS == '0' && CP15SDISABLE == HIGH then
    UNDEFINED;
elsf SCR.NS == '0' && CP15SDISABLE2 == HIGH then
    UNDEFINED;
  else
    if TTBCR.EAE == '1' then
      if SCR.NS == '0' then
        MAIR0_S = R[t];
      else
        MAIR0_NS = R[t];
    else
      if SCR.NS == '0' then
        PRRR_S = R[t];
      else
        PRRR_NS = R[t];

AArch32 System Register Descriptions

Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ARM DDI 0487I.a
ID081822
G8.2.122   REVIDR, Revision ID Register

The REVIDR characteristics are:

Purpose

Provides implementation-specific minor revision information.

Configurations

AArch32 System register REVIDR bits [31:0] are architecturally mapped to AArch64 System register REVIDR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to REVIDR are UNDEFINED.

If REVIDR has the same value as MIDR, then its contents have no significance.

Attributes

REVIDR is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>IMPLEMENTATION DEFINED</th>
</tr>
</thead>
</table>

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing REVIDR

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC} \{<c>\}\{<q>\} \langle\text{coproc}\rangle, \{\#\}<\text{opc1}>, \langle\text{Rt}\rangle, \langle\text{CRn}\rangle, \langle\text{CRm}\rangle, \{\#\}<\text{opc2}> \\
\]

\[
\begin{array}{c c c c c}
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
0b1111 & 0b0000 & 0b0000 & 0b0000 & 0b1110 \\
\end{array}
\]

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    R[t] = REVIDR;
  endif
elsif PSTATE_EL == EL2 then
  R[t] = REVIDR;
elsif PSTATE_EL == EL3 then
  R[t] = REVIDR;
**G8.2.123  RMR, Reset Management Register**

The RMR characteristics are:

**Purpose**

If EL1 or EL3 is the highest implemented Exception level and this register is implemented:
- A write to the register at the highest implemented Exception level can request a Warm reset.
- If the highest implemented Exception level can use AArch32 and AArch64, this register specifies the Execution state that the PE boots into on a Warm reset.

**Configurations**

AArch32 System register RMR bits [31:0] are architecturally mapped to AArch64 System register RMR_EL1[31:0] when the highest implemented Exception level is EL1.

AArch32 System register RMR bits [31:0] are architecturally mapped to AArch64 System register RMR_EL3[31:0] when EL3 is implemented.

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to RMR are UNDEFINED.

Only implemented if EL1 or EL3 is the highest implemented Exception level. In this case:
- If the highest implemented Exception level can use AArch32 and AArch64 then this register must be implemented.
- If the highest implemented Exception level cannot use AArch64 then it is IMPLEMENTATION DEFINED whether the register is implemented.

**Attributes**

RMR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>2</td>
<td>RR, bit [1] Reset Request. Setting this bit to 1 requests a Warm reset. The reset behavior of this field is:</td>
</tr>
<tr>
<td>1</td>
<td>On a Warm reset, this field resets to 0.</td>
</tr>
<tr>
<td>0</td>
<td>AA64, bit [0] When the highest implemented Exception level can use AArch64, determines which Execution state the PE boots into after a Warm reset:</td>
</tr>
<tr>
<td></td>
<td>0b0 AArch32.</td>
</tr>
<tr>
<td></td>
<td>0b1 AArch64.</td>
</tr>
<tr>
<td></td>
<td>On coming out of the Warm reset, execution starts at the IMPLEMENTATION DEFINED reset vector address of the specified Execution state.</td>
</tr>
<tr>
<td></td>
<td>If the highest implemented Exception level cannot use AArch64 this bit is RAZ/WI.</td>
</tr>
<tr>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>0b0 When implemented as a RW field, this field resets to 0 on a Cold reset.</td>
</tr>
</tbody>
</table>
Accessing RMR

When EL3 is implemented, Arm deprecates accessing this register from any PE mode other than Monitor mode.

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{\langle c\rangle}\{\langle q\rangle\} <\text{coproc}>, \{\#\langle opc1\rangle\}, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>, \{\#\langle opc2\rangle\}
\]

\[
\begin{array}{cccccc}
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
0b1111 & 0b000 & 0b1100 & 0b0000 & 0b010 \\
\end{array}
\]

\[
\text{if } \text{PSTATE.EL IN } \{\text{EL1, EL3}\} \&\& \text{IsHighestEL(PSTATE.EL)} \text{ then} \\
R[t] = \text{RMR}; \\
\text{else} \\
\text{UNDEFINED;}
\]

\[
\text{MCR}\{\langle c\rangle}\{\langle q\rangle\} <\text{coproc}>, \{\#\langle opc1\rangle\}, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>, \{\#\langle opc2\rangle\}
\]

\[
\begin{array}{cccccc}
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
0b1111 & 0b000 & 0b1100 & 0b0000 & 0b010 \\
\end{array}
\]

\[
\text{if } \text{PSTATE.EL == EL0 then} \\
\text{UNDEFINED;}
\]
\[
\text{elsif } \text{PSTATE.EL == EL1 then} \\
\text{if IsHighestEL(EL1) then} \\
\text{RMR = R[t];} \\
\text{else} \\
\text{UNDEFINED;}
\]
\[
\text{elsif } \text{PSTATE.EL == EL2 then} \\
\text{UNDEFINED;}
\]
\[
\text{elsif } \text{PSTATE.EL == EL3 then} \\
\text{if CP15SDISABLE == HIGH then} \\
\text{UNDEFINED;}
\text{elsif CP15SDISABLE2 == HIGH then} \\
\text{UNDEFINED;}
\text{else} \\
\text{RMR = R[t];}
\]
**G8.2.124   RVBAR, Reset Vector Base Address Register**

The RVBAR characteristics are:

**Purpose**
If EL3 is not implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch32 state.

**Configurations**
This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to RVBAR are UNDEFINED.

This register is only implemented if the highest Exception level implemented is capable of using AArch32, and is not EL3.

**Attributes**
RVBAR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResetAddress</td>
<td>Bits [31:1] of the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 32-bit state. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.</td>
</tr>
<tr>
<td>Bit [0]</td>
<td>Reserved, RES1.</td>
</tr>
</tbody>
</table>

**Accessing RVBAR**
Accesses to this register use the following encodings in the System register encoding space:

\[
MRC\{<c>\}<q>\{<opc>\}, \{<Rt>\}, \{<CRn>\}, \{<CRm>\}, \{<opc2>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
  if IsHighestEL(EL1) then
    R[t] = RVBAR;
nseIf EL2Enabled() \&\& \&\& ELUsingAArch32(EL2) \&\& HSTR_EL2.T12 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() \&\& ELUsingAArch32(EL2) \&\& HSTR.T12 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif !ELUsingAArch32(EL3) \&\& SCR_EL3.NS,EEL2> == '01' then
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else if !ELUsingAArch32(EL3) \&\& SCR_EL3.NS == '0' then
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  if IsHighestEL(EL2) then
    R[t] = RVBAR;
  else
    UNDEFINED;
elsif PSTATE.EL == EL3 then
  R[t] = MVBAR;
G8.2.125 SCR, Secure Configuration Register

The SCR characteristics are:

**Purpose**

When EL3 is implemented and can use AArch32, defines the configuration of the current Security state. It specifies:

- The Security state, either Secure or Non-secure.
- What mode the PE branches to if anIRQ, FIQ, or External abort occurs.
- Whether the PSTATE.F or PSTATE.A bits can be modified when SCR.NS==1.

**Configurations**

This register is present only when EL3 is capable of using AArch32. Otherwise, direct accesses to SCR are UNDEFINED.

**Attributes**

SCR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>15</td>
<td>TERR, bit [15]</td>
</tr>
<tr>
<td>13</td>
<td>TWE, bit [13]</td>
</tr>
</tbody>
</table>

**TERR, bit [15]**

*When FEAT_RAS is implemented:*

Trap Error record accesses. Generate a Monitor Trap exception on accesses to the following registers from modes other than Monitor mode:

ERRIDR, ERXADDR, ERXADDR2, ERXCTRLR, ERXCTRLR2, ERXFR, ERXFR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, and ERXSTATUS. When FEAT_RASv1p1 is implemented, ERXMISC4, ERXMISC5, ERXMISC6, ERXMISC7.

- 0b0 This control does not cause any instructions to be trapped.
- 0b1 Accesses to the specified registers from modes other than Monitor mode generate a Monitor Trap exception.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

**Bit [14]**

Reserved, RES0.

**TWE, bit [13]**

Traps WFE instructions to Monitor mode.

- 0b0 This control does not cause any instructions to be trapped.
Any attempt to execute a WFE instruction in any mode other than Monitor mode is trapped to Monitor mode, if the instruction would otherwise have caused the PE to enter a low-power state and the attempted execution does not generate an exception that is taken to EL1 or EL2 by SCTLR.nTWE or HCR.TWE.

Any exception that is taken to EL1 or to EL2 has priority over this trap.

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

--- Note ---
Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

**TWI, bit [12]**
Traps WFI instructions to Monitor mode.
- **0b0** This control does not cause any instructions to be trapped.
- **0b1** Any attempt to execute a WFI instruction in any mode other than Monitor mode is trapped to Monitor mode, if the instruction would otherwise have caused the PE to enter a low-power state and the attempted execution does not generate an exception that is taken to EL1 or EL2 by SCTLR.nTWI or HCR.TWI.

Any exception that is taken to EL1 or to EL2 has priority over this trap.

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

--- Note ---
Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

**Bits [11:10]**
Reserved, RES0.

**SIF, bit [9]**
Secure instruction fetch. When the PE is in Secure state, this bit disables instruction execution from Non-secure memory.
- **0b0** Secure state instruction execution from Non-secure memory is permitted.
- **0b1** Secure state instruction execution from Non-secure memory is not permitted.

This bit is permitted to be cached in a TLB.
The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.
HCE, bit [8]
Hypervisor Call instruction enable. If EL2 is implemented, enables execution of HVC instructions at Non-secure EL1 and EL2.

0b0 HVC instructions are:
• UNDEFINED at Non-secure EL1. The Undefined Instruction exception is taken from PL1 to PL1.
• UNPREDICTABLE at EL2. Behavior is one of the following:
  — The instruction is UNDEFINED.
  — The instruction executes as a NOP.

0b1 HVC instructions are enabled at Non-secure EL1 and EL2.

Note
HVC instructions are always UNDEFINED at EL0 and in Secure state.

If EL2 is not implemented, this bit is RES0 and HVC is UNDEFINED.
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

SCD, bit [7]
Secure Monitor Call disable. Disables SMC instructions.

0b0 SMC instructions are enabled.

0b1 In Non-secure state, SMC instructions are UNDEFINED. The Undefined Instruction exception is taken from the current Exception level to the current Exception level.
In Secure state, behavior is one of the following:
• The instruction is UNDEFINED.
• The instruction executes as a NOP.

Note
SMC instructions are always UNDEFINED at PL0.
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

nET, bit [6]
Not Early Termination. This bit disables early termination.

0b0 Early termination permitted. Execution time of data operations can depend on the data values.

0b1 Disable early termination. The number of cycles required for data operations is forced to be independent of the data values.

This IMPLEMENTATION DEFINED mechanism can disable data dependent timing optimizations from multiplies and data operations. It can provide system support against information leakage that might be exploited by timing correlation types of attack.

On implementations that do not support early termination or do not support disabling early termination, this bit is RES0.
The reset behavior of this field is:
• On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.
AW, bit [5]

When the value of SCR.EA is 1 and the value of HCR.AMO is 0, this bit controls whether PSTATE.A masks an External abort taken from Non-secure state.

0b0  External aborts taken from Non-secure state are not masked by PSTATE.A, and are taken to EL3.

0b1  External aborts taken from Secure state are masked by PSTATE.A.

When SCR.EA is 0 or HCR.AMO is 1, this bit has no effect.

The reset behavior of this field is:
•  On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

FW, bit [4]

When the value of SCR.FIQ is 1 and the value of HCR.FMO is 0, this bit controls whether PSTATE.F masks an FIQ interrupt taken from Non-secure state.

0b0  An FIQ taken from Non-secure state is not masked by PSTATE.F, and is taken to EL3.

0b1  An FIQ taken from Secure state is masked by PSTATE.F.

When SCR.FIQ is 0 or HCR.FMO is 1, this bit has no effect.

The reset behavior of this field is:
•  On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

EA, bit [3]

External Abort handler. This bit controls which mode takes External aborts and SError interrupt exceptions.

0b0  External aborts taken to Abort mode.

0b1  External aborts taken to Monitor mode.

The reset behavior of this field is:
•  On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

FIQ, bit [2]

FIQ handler. This bit controls which mode takes FIQ exceptions.

0b0  FIQs taken to FIQ mode.

0b1  FIQs taken to Monitor mode.

The reset behavior of this field is:
•  On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

IRQ, bit [1]

IRQ handler. This bit controls which mode takes IRQ exceptions.

0b0  IRQs taken to IRQ mode.

0b1  IRQs taken to Monitor mode.

The reset behavior of this field is:
•  On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

NS, bit [0]

Non-secure bit. Except when the PE is in Monitor mode, this bit determines the Security state of the PE.

0b0  PE is in Secure state.

0b1  PE is in Non-secure state.
If the HCR.TGE bit is set, an attempt to change from a Secure PL1 mode to a Non-secure EL1 mode by changing the SCR.NS bit from 0 to 1 results in the SCR.NS bit remaining as 0.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

**Accessing SCR**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC\{<c}\{<q}\{ coproc, \{#opc1\}, \{Rt\}, \{CRn\}, \{CRm\}\{, \{#opc2\}\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

\[
MCR\{<c}\{<q}\{ coproc, \{#opc1\}, \{Rt\}, \{CRn\}, \{CRm\}\{, \{#opc2\}\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>
G8.2.126  SCTLVR, System Control Register

The SCTLVR characteristics are:

Purpose

Provides the top level control of the system, including its memory system.

Configurations

AArch32 System register SCTLVR bits [31:0] are architecturally mapped to AArch64 System register SCTLVR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to SCTLVR are UNDEFINED.

Some bits in the register are read-only. These bits relate to non-configurable features of an implementation, and are provided for compatibility with previous versions of the architecture.

Attributes

SCTLVR is a 32-bit register.

Field descriptions

DSSBS, bit [31]

When FEAT_SSBS is implemented:

Default PSTATE.SSBS value on Exception Entry. The defined values are:

0b0  PSTATE.SSBS is set to 0 on an exception to any mode in this security state except Hyp mode

0b1  PSTATE.SSBS is set to 1 on an exception to any mode in this security state except Hyp mode

----- Note -----

When EL3 is implemented and is using AArch32, this bit is banked between the two Security states.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

TE, bit [30]

T32 Exception Enable. This bit controls whether exceptions to an Exception level that is executing at PL1 are taken to A32 or T32 state:

0b0  Exceptions, including reset, taken to A32 state.
0b1  Exceptions, including reset, taken to T32 state.
The reset behavior of this field is:
  •  On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

AFE, bit [29]
Access Flag Enable. When using the Short-descriptor translation table format for the PL1&0
translation regime, this bit enables use of the AP[0] bit in the translation descriptors as the Access
flag, and restricts access permissions in the translation descriptors to the simplified model.

0b0  In the Translation table descriptors, AP[0] is an access permissions bit. The full range
of access permissions is supported. No Access flag is implemented.
0b1  In the Translation table descriptors, AP[0] is the Access flag. Only the simplified model
for access permissions is supported.

When using the Long-descriptor translation table format, the VMSA behaves as if this bit is set to
1, regardless of the value of this bit.
The AFE bit is permitted to be cached in a TLB.
The reset behavior of this field is:
  •  On a Warm reset, this field resets to 0.

TRE, bit [28]
TEX remap enable. This bit enables remapping of the TEX[2:1] bits in the PL1&0 translation
regime for use as two translation table bits that can be managed by the operating system. Enabling
this remapping also changes the scheme used to describe the memory region attributes in the
VMSA.

0b0  TEX remap disabled. TEX[2:0] are used, with the C and B bits, to describe the memory
region attributes.
0b1  TEX remap enabled. TEX[2:1] are reassigned for use as bits managed by the operating
system. The TEX[0], C, and B bits are used to describe the memory region attributes,
with the MMU remap registers.

When the value of TTBCR.EAE is 1, this bit is RES1.
The TRE bit is permitted to be cached in a TLB.
The reset behavior of this field is:
  •  On a Warm reset, this field resets to 0.

Bits [27:26]
Reserved, RES0.

EE, bit [25]
The value of the PSTATE.E bit on branch to an exception vector or coming out of reset, and the
endianness of stage 1 translation table walks in the PL1&0 translation regime.

0b0  Little-endian. PSTATE.E is cleared to 0 on taking an exception or coming out of reset.
Stage 1 translation table walks in the PL1&0 translation regime are little-endian.
0b1  Big-endian. PSTATE.E is set to 1 on taking an exception or coming out of reset. Stage
1 translation table walks in the PL1&0 translation regime are big-endian.

If an implementation does not provide Big-endian support for data accesses at Exception levels
higher than EL0, this bit is RES0.
If an implementation does not provide Little-endian support for data accesses at Exception levels
higher than EL0, this bit is RES1.
The reset behavior of this field is:
  •  On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Bit [24]
Reserved, RES0.
SPAN, bit [23]

*When FEAT_PAN is implemented:*

Set Privileged Access Never, on taking an exception to EL1 from either Secure or Non-secure state, or to EL3 from Secure state when EL3 is using AArch32.

- **0b0** PSTATE.PAN is set to 1 in the following situations:
  - In Non-secure state, on taking an exception to EL1.
  - In Secure state, when EL3 is using AArch64, on taking an exception to EL1.
  - In Secure state, when EL3 is using AArch32, on taking an exception to EL3.
- **0b1** The value of PSTATE.PAN is left unchanged on taking an exception to EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

*Otherwise:*

Reserved, RES1.

Bit [22]

Reserved, RES1.

Bit [21]

Reserved, RES0.

UWXN, bit [20]

Unprivileged write permission implies PL1 XN (Execute-never). This bit can force all memory regions that are writable at PL0 to be treated as XN for accesses from software executing at PL1.

- **0b0** This control has no effect on memory access permissions.
- **0b1** Any region that is writable at PL0 forced to XN for accesses from software executing at PL1.

The UWXN bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

WXN, bit [19]

Write permission implies XN (Execute-never). For the PL1&0 translation regime, this bit can force all memory regions that are writable to be treated as XN.

- **0b0** This control has no effect on memory access permissions.
- **0b1** Any region that is writable in the PL1&0 translation regime is forced to XN for accesses from software executing at PL1 or PL0.

This bit applies only when SCTLR.M bit is set.

The WXN bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

nTWE, bit [18]

Traps EL0 execution of WFE instructions to Undefined mode.

- **0b0** Any attempt to execute a WFE instruction at EL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.
- **0b1** This control does not cause any instructions to be trapped.

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.
Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:
- On a Warm reset, this field resets to 1.

Bit [17]

Reserved, RES0.

nTWI, bit [16]

Traps EL0 execution of WFI instructions to Undefined mode.

0b0 Any attempt to execute a WFI instruction at EL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.

0b1 This control does not cause any instructions to be trapped.

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:
- On a Warm reset, this field resets to 1.

Bits [15:14]

Reserved, RES0.

V, bit [13]

Vectors bit. This bit selects the base address of the exception vectors for exceptions taken to a PE mode other than Monitor mode or Hyp mode:

0b0 Normal exception vectors. Base address is held in VBAR.

0b1 High exception vectors (Hivecs), base address 0xFFFF0000. This base address cannot be remapped.

The reset behavior of this field is:
- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

I, bit [12]

Instruction access Cacheability control, for accesses at EL1 and EL0:

0b0 All instruction access to Normal memory from PL1 and PL0 are Non-cacheable for all levels of instruction and unified cache.

If the value of SCTLR.M is 0, instruction accesses from stage 1 of the PL1&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

0b1 All instruction access to Normal memory from PL1 and PL0 can be cached at all levels of instruction and unified cache.

If the value of SCTLR.M is 0, instruction accesses from stage 1 of the PL1&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.
Instruction accesses to Normal memory from EL1 and EL0 are Cacheable regardless of the value of the 
SCTLR.I bit if either:

- EL2 is using AArch32 and the value of HCR.DC is 1.
- EL2 is using AArch64 and the value of HCR_EL2.DC is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

Bit [11]
Reserved, RES1.

EnRCTX, bit [10]

When FEAT_SPECRES is implemented:
Enable EL0 access to the AArch32 CFPRCTX, DVPRCTX, and CPPRCTX instructions.

- 0b0 EL0 access to these instructions is disabled, and these instructions are trapped to EL1.
- 0b1 EL0 access to these instructions is enabled.

Note
When EL3 is implemented and is using AArch32, this bit is banked between the two Security states.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bit [9]
Reserved, RES0.

SED, bit [8]
SETEND instruction disable. Disables SETEND instructions at PL0 and PL1.

- 0b0 SETEND instruction execution is enabled at PL0 and PL1.
- 0b1 SETEND instructions are UNDEFINED at PL0 and PL1.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

ITD, bit [7]
IT Disable. Disables some uses of IT instructions at PL1 and PL0.

- 0b0 All IT instruction functionality is enabled at PL1 and PL0.
- 0b1 Any attempt at PL1 or PL0 to execute any of the following is UNDEFINED:
  - All encodings of the IT instruction with hw1[3:0]=1000.
  - All encodings of the subsequent instruction with the following values for hw1:
    - 11xxxxxxxxxxxx: All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM.
    - 1011xxxxxxxxxx: All instructions in Miscellaneous 16-bit instructions on page F3-7303.
    - 10100xxxxxxxxxxx: ADD Rd, PC, #imm
    - 01001xxxxxxxxxxx: LDR Rd, [PC, #imm]
    - 01001xxxxxxxxxxx: ADD Rd, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC.
— 010001xx1xxxx111: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers unpredictable cases with BLX Rn.

These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block.

It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:

• A 16-bit instruction, that can only be followed by another 16-bit instruction.
• The first half of a 32-bit instruction.

This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED.

An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If an instruction in an active IT block that would be disabled by this field sets this field to 1 then behavior is CONstrained UNpredictable. For more information see Changes to an ITD control by an instruction in an IT block on page E1-7138.

ITD is optional, but if it is implemented in the SCTLR then it must also be implemented in the SCTLR_EL1, SCTLR_EL2, and HSCTLR.

The reset behavior of this field is:

• On a Warm reset, this field resets to 0.

When an implementation does not implement ITD, access to this field is RAZ/WI.

UNK, bit [6]

 Writes to this bit are IGNORED. Reads of this bit return an UNKNOWN value.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

CP15BEN, bit [5]

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==0b1111) encoding space from PL1 and PL0:

0b0 PL0 and PL1 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED.

0b1 PL0 and PL1 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the SCTLR then it must also be implemented in the SCTLR_EL1, SCTLR_EL2, and HSCTLR.

The reset behavior of this field is:

• On a Warm reset, this field resets to 1.

When an implementation does not implement CP15BEN, access to this field is RAO/WI.

LSMAOE, bit [4]

When FEAT_LSMAOC is implemented:

Load Multiple and Store Multiple Atomicity and Ordering Enable.

0b0 For all memory accesses at EL1 or EL0, A32 and T32 Load Multiple and Store Multiple can have an interrupt taken during the sequence memory accesses, and the memory accesses are not required to be ordered.

0b1 The ordering and interrupt behavior of A32 and T32 Load Multiple and Store Multiple at EL1 or EL0 is as defined for Armv8.0.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

• On a Warm reset, this field resets to 1.
nTLSMD, bit [3]

When FEAT_LSMAOC is implemented:

No Trap Load Multiple and Store Multiple to Device-nGRE/Device-nGnRE/Device-nGnRnE memory.

0b0  All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL1 or EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are trapped and generate a stage 1 Alignment fault.

0b1  All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL1 or EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are not trapped.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

• On a Warm reset, this field resets to 1.

Otherwise:

Reserved, RES1.

C, bit [2]

Cacheability control, for data accesses at EL1 and EL0:

0b0  All data access to Normal memory from PL1 and PL0, and all accesses to the PL1&0 stage 1 translation tables, are Non-cacheable for all levels of data and unified cache.

0b1  All data access to Normal memory from PL1 and PL0, and all accesses to the PL1&0 stage 1 translation tables, can be cached at all levels of data and unified cache.

The PE ignores SCTLR.C for Non-secure state and data accesses to Normal memory from EL1 and EL0 are Cacheable if either:

• EL2 is using AArch32 and the value of HCR.DC is 1.
• EL2 is using AArch64 and the value of HCR_EL2.DC is 1.

The reset behavior of this field is:

• On a Warm reset, this field resets to 0.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at PL1 and PL0:

0b0  Alignment fault checking disabled when executing at PL1 or PL0. Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.

0b1  Alignment fault checking enabled when executing at PL1 or PL0. All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

The reset behavior of this field is:

• On a Warm reset, this field resets to 0.

M, bit [0]

MMU enable for EL1 and EL0 stage 1 address translation. Possible values of this bit are:

0b0  EL1 and EL0 stage 1 address translation disabled. See the SCTLR.I field for the behavior of instruction accesses to Normal memory.
0b1 EL1 and EL0 stage 1 address translation enabled.

In the Non-secure state the PE behaves as if the value of the SCTLR.M field is 0 for all purposes other than returning the value of a direct read of the field if either:

- EL2 is using AArch32 and the value of HCR.{DC, TGE} is not {0, 0}.
- EL2 is using AArch64 and the value of HCR_EL2.{DC, TGE} is not {0, 0}.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

### Accessing SCTLR

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} <\text{coproc}>, {#}<\text{opc1}>, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>\{, {#}<\text{opc2}>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
   UNDEFINED;
elsif PSTATE.EL == EL1 then
   if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T1 == 1 then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
   elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == 1 then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
   elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TRVM == 1 then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
   elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == 1 then
      AArch32.TakeHypTrapException(0x03);
   elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
      R[t] = SCTLR_NS;
   else
      R[t] = SCTLR;
   elsif PSTATE.EL == EL2 then
      if HaveEL(EL3) && ELUsingAArch32(EL3) then
         R[t] = SCTLR_NS;
      else
         R[t] = SCTLR;
      elsif PSTATE.EL == EL3 then
         if SCR.NS == 0 then
            R[t] = SCTLR_S;
         else
            R[t] = SCTLR_NS;
      else
         R[t] = SCTLR_NS;

### MCR\{<c>\}{<q>} <\text{coproc}>, {#}<\text{opc1}>, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>\{, {#}<\text{opc2}>\}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
   UNDEFINED;
elsif PSTATE.EL == EL1 then
   if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T1 == 1 then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
   elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == 1 then
      AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    SCTLR_NS = R[t];
else
    SCTLR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        SCTLR_NS = R[t];
    else
        SCTLR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR_NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;
    elsif SCR_NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        if SCR_NS == '0' then
            SCTLR_S = R[t];
        else
            SCTLR_NS = R[t];
    end if
end if

G8.2.127  SPSR, Saved Program Status Register

The SPSR characteristics are:

**Purpose**

Holds the saved process state for the current mode.

**Configurations**

This register is present only when AArch32 is supported. Otherwise, direct accesses to SPSR are UNDEFINED.

**Attributes**

SPSR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31  | N      | Negative Condition flag. Set to the value of PSTATE.N on taking an exception to the current mode, and copied to PSTATE.N on executing an exception return operation in the current mode. The reset behavior of this field is:  
  - On a Warm reset, this field resets to an architecturally UNKNOWN value. |
| 30  | Z      | Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to the current mode, and copied to PSTATE.Z on executing an exception return operation in the current mode. The reset behavior of this field is:  
  - On a Warm reset, this field resets to an architecturally UNKNOWN value. |
| 29  | C      | Carry Condition flag. Set to the value of PSTATE.C on taking an exception to the current mode, and copied to PSTATE.C on executing an exception return operation in the current mode. The reset behavior of this field is:  
  - On a Warm reset, this field resets to an architecturally UNKNOWN value. |
| 28  | V      | Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to the current mode, and copied to PSTATE.V on executing an exception return operation in the current mode. The reset behavior of this field is:  
  - On a Warm reset, this field resets to an architecturally UNKNOWN value. |
| 27  | Q      | Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to the current mode, and copied to PSTATE.Q on executing an exception return operation in the current mode. The reset behavior of this field is:  
  - On a Warm reset, this field resets to an architecturally UNKNOWN value. |

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>IL</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>GE</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>J</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>IT[1:0]</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>SSBS</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>DIT</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>PAN</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>IT[7:2]</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>M[4:0]</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>M[4:0]</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>M[4:0]</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>M[4:0]</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>M[4:0]</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>M[4:0]</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>M[4:0]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>M[4:0]</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>M[4:0]</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>M[4:0]</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>M[4:0]</td>
<td></td>
</tr>
</tbody>
</table>
IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to the current mode, and copied to PSTATE.IT on executing an exception return operation in the current mode.

SPSR.IT must contain a value that is valid for the instruction being returned to.

The IT field is split as follows:
- IT[1:0] is SPSR[26:25].
- IT[7:2] is SPSR[15:10].

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:
Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to the current mode, and copied to PSTATE.SSBS on executing an exception return operation in the current mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:
Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to the current mode, and copied to PSTATE.PAN on executing an exception return operation in the current mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:
Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to the current mode, and copied to PSTATE.DIT on executing an exception return operation in the current mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to the current mode, and copied to PSTATE.IL on executing an exception return operation in the current mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
GE, bits [19:16]
Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to the current mode, and copied to PSTATE.GE on executing an exception return operation in the current mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]
Endianness. Set to the value of PSTATE.E on taking an exception to the current mode, and copied to PSTATE.E on executing an exception return operation in the current mode.

If the implementation does not support big-endian operation, SPSR.E is RES0. If the implementation does not support little-endian operation, SPSR.E is RES1. On executing an exception return operation in the current mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR.E is RES1.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]
SError interrupt mask. Set to the value of PSTATE.A on taking an exception to the current mode, and copied to PSTATE.A on executing an exception return operation in the current mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]
IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to the current mode, and copied to PSTATE.I on executing an exception return operation in the current mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]
FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to the current mode, and copied to PSTATE.F on executing an exception return operation in the current mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]
T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to the current mode, and copied to PSTATE.T on executing an exception return operation in the current mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]
Mode. Set to the value of PSTATE.M[4:0] on taking an exception to the current mode, and copied to PSTATE.M[4:0] on executing an exception return operation in the current mode.

- 0b10000: User.
- 0b10001: FIQ.
- 0b10010: IRQ.
- 0b10011: Supervisor.
- 0b10110: Monitor.
- 0b10111: Abort.
- 0b11010: Hyp.
0b11011  Undefined.
0b11111  System.

Other values are reserved. If SPSR.M[4:0] has a Reserved value, or a value for an unimplemented
Exception level, executing an exception return operation in the current mode is an illegal return
event, as described in Illegal return events from AArch32 state on page G1-8952.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.
G8.2.128 SPSR_abt, Saved Program Status Register (Abort mode)

The SPSR_abt characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to Abort mode.

**Configurations**

AArch32 System register SPSR_abt bits [31:0] are architecturally mapped to AArch64 System register SPSR_abt[31:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to SPSR_abt are UNDEFINED.

**Attributes**

SPSR_abt is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>N</td>
<td>Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Abort mode, and copied to PSTATE.N on executing an exception return operation in Abort mode. On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>30</td>
<td>Z</td>
<td>Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Abort mode, and copied to PSTATE.Z on executing an exception return operation in Abort mode. On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>29</td>
<td>C</td>
<td>Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Abort mode, and copied to PSTATE.C on executing an exception return operation in Abort mode. On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>28</td>
<td>V</td>
<td>Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Abort mode, and copied to PSTATE.V on executing an exception return operation in Abort mode. On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>27</td>
<td>Q</td>
<td>Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Abort mode, and copied to PSTATE.Q on executing an exception return operation in Abort mode. On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
</tbody>
</table>
IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Abort mode, and copied to PSTATE.IT on executing an exception return operation in Abort mode.

SPSR_abt.IT must contain a value that is valid for the instruction being returned to.

The IT field is split as follows:

- IT[1:0] is SPSR_abt[26:25].
- IT[7:2] is SPSR_abt[15:10].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

*When FEAT_SSBS is implemented:*
Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Abort mode, and copied to PSTATE.SSBS on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

PAN, bit [22]

*When FEAT_PAN is implemented:*
Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Abort mode, and copied to PSTATE.PAN on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

DIT, bit [21]

*When FEAT_DIT is implemented:*
Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Abort mode, and copied to PSTATE.DIT on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Abort mode, and copied to PSTATE.IL on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
GE, bits [19:16]
Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Abort mode, and copied to PSTATE.GE on executing an exception return operation in Abort mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]
Endianness. Set to the value of PSTATE.E on taking an exception to Abort mode, and copied to PSTATE.E on executing an exception return operation in Abort mode.

If the implementation does not support big-endian operation, SPSR_abt.E is RES0. If the implementation does not support little-endian operation, SPSR_abt.E is RES1. On executing an exception return operation in Abort mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_abt.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_abt.E is RES1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]
SError interrupt mask. Set to the value of PSTATE.A on taking an exception to Abort mode, and copied to PSTATE.A on executing an exception return operation in Abort mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]
IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Abort mode, and copied to PSTATE.I on executing an exception return operation in Abort mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]
FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Abort mode, and copied to PSTATE.F on executing an exception return operation in Abort mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]
T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Abort mode, and copied to PSTATE.T on executing an exception return operation in Abort mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]
Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Abort mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Abort mode.

<table>
<thead>
<tr>
<th>Value</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10000</td>
<td>User</td>
</tr>
<tr>
<td>0b10001</td>
<td>FIQ</td>
</tr>
<tr>
<td>0b10010</td>
<td>IRQ</td>
</tr>
<tr>
<td>0b10011</td>
<td>Supervisor</td>
</tr>
<tr>
<td>0b1011</td>
<td>Abort</td>
</tr>
<tr>
<td>0b11011</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
0b11111 System.

Other values are reserved. If SPSR_abt.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Abort mode is an illegal return event, as described in Illegal return events from AArch32 state on page G1-8952.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_abt

SPSR_abt is accessible in all modes other than User mode and Abort mode.

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS}\{c\}\{q\} \; <Rd>, \; \text{SPSR\_abt}
\]

\[
\begin{array}{ccc}
\text{R} & \text{M} & \text{M1} \\
0b1 & 0b1 & 0b100 \\
\end{array}
\]

\[
\text{MSR}\{c\}\{q\} \; \text{SPSR\_abt}, \; <Rn>
\]

\[
\begin{array}{ccc}
\text{R} & \text{M} & \text{M1} \\
0b1 & 0b1 & 0b100 \\
\end{array}
\]
G8.2.129  SPSR_fiq, Saved Program Status Register (FIQ mode)

The SPSR_fiq characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to FIQ mode.

**Configurations**

AArch32 System register SPSR_fiq bits [31:0] are architecturally mapped to AArch64 System register SPSR_fiq[31:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to SPSR_fiq are UNDEFINED.

**Attributes**

SPSR_fiq is a 32-bit register.

**Field descriptions**

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to FIQ mode, and copied to PSTATE.N on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to FIQ mode, and copied to PSTATE.Z on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to FIQ mode, and copied to PSTATE.C on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to FIQ mode, and copied to PSTATE.V on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to FIQ mode, and copied to PSTATE.Q on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to FIQ mode, and copied to PSTATE.IT on executing an exception return operation in FIQ mode.

SPSR_fiq.IT must contain a value that is valid for the instruction being returned to.

The IT field is split as follows:
- IT[1:0] is SPSR_fiq[26:25].
- IT[7:2] is SPSR_fiq[15:10].

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to FIQ mode, and copied to PSTATE.SSBS on executing an exception return operation in FIQ mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to FIQ mode, and copied to PSTATE.PAN on executing an exception return operation in FIQ mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to FIQ mode, and copied to PSTATE.DIT on executing an exception return operation in FIQ mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to FIQ mode, and copied to PSTATE.IL on executing an exception return operation in FIQ mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
GE, bits [19:16]
Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to FIQ mode, and copied to PSTATE.GE on executing an exception return operation in FIQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]
Endianness. Set to the value of PSTATE.E on taking an exception to FIQ mode, and copied to PSTATE.E on executing an exception return operation in FIQ mode.

If the implementation does not support big-endian operation, SPSR_fiq.E is RES0. If the implementation does not support little-endian operation, SPSR_fiq.E is RES1. On executing an exception return operation in FIQ mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_fiq.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_fiq.E is RES1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]
SError interrupt mask. Set to the value of PSTATE.A on taking an exception to FIQ mode, and copied to PSTATE.A on executing an exception return operation in FIQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]
IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to FIQ mode, and copied to PSTATE.I on executing an exception return operation in FIQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]
FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to FIQ mode, and copied to PSTATE.F on executing an exception return operation in FIQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]
T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to FIQ mode, and copied to PSTATE.T on executing an exception return operation in FIQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]
Mode. Set to the value of PSTATE.M[4:0] on taking an exception to FIQ mode, and copied to PSTATE.M[4:0] on executing an exception return operation in FIQ mode.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10000</td>
<td>User</td>
</tr>
<tr>
<td>0b10001</td>
<td>FIQ</td>
</tr>
<tr>
<td>0b10010</td>
<td>IRQ</td>
</tr>
<tr>
<td>0b10011</td>
<td>Supervisor</td>
</tr>
<tr>
<td>0b10111</td>
<td>Abort</td>
</tr>
<tr>
<td>0b11011</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
0b11111 System.

Other values are reserved. If SPSR_fiq.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in FIQ mode is an illegal return event, as described in *Illegal return events from AArch32 state on page G1-8952*.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing SPSR_fiq**

SPSR_fiq is accessible in all modes other than User mode and FIQ mode.

Accesses to this register use the following encodings in the System register encoding space:

\[
MRS\{<c>\}\{<q>\} \ <Rd>, SPSR_fiq
\]

<table>
<thead>
<tr>
<th>R</th>
<th>M</th>
<th>M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>0b1110</td>
</tr>
</tbody>
</table>

\[
MSR\{<c>\}\{<q>\} SPSR_fiq, <Rn>
\]

<table>
<thead>
<tr>
<th>R</th>
<th>M</th>
<th>M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>0b1110</td>
</tr>
</tbody>
</table>
SPSR_hyp, Saved Program Status Register (Hyp mode)

The SPSR_hyp characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to Hyp mode.

**Configurations**

AArch32 System register SPSR_hyp bits [31:0] are architecturally mapped to AArch64 System register SPSR_EL2[31:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to SPSR_hyp are UNDEFINED.

**Attributes**

SPSR_hyp is a 32-bit register.

**Field descriptions**

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Hyp mode, and copied to PSTATE.N on executing an exception return operation in Hyp mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Hyp mode, and copied to PSTATE.Z on executing an exception return operation in Hyp mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Hyp mode, and copied to PSTATE.C on executing an exception return operation in Hyp mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Hyp mode, and copied to PSTATE.V on executing an exception return operation in Hyp mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Hyp mode, and copied to PSTATE.Q on executing an exception return operation in Hyp mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Hyp mode, and copied to PSTATE.IT on executing an exception return operation in Hyp mode.

SPSR_hyp.IT must contain a value that is valid for the instruction being returned to.

The IT field is split as follows:

- IT[1:0] is SPSR_hyp[26:25].
- IT[7:2] is SPSR_hyp[15:10].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Hyp mode, and copied to PSTATE.SSBS on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Hyp mode, and copied to PSTATE.PAN on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Hyp mode, and copied to PSTATE.DIT on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Hyp mode, and copied to PSTATE.IL on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
GE, bits [19:16]
Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Hyp mode, and copied to PSTATE.GE on executing an exception return operation in Hyp mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]
Endianness. Set to the value of PSTATE.E on taking an exception to Hyp mode, and copied to PSTATE.E on executing an exception return operation in Hyp mode.
If the implementation does not support big-endian operation, SPSR_hyp.E is RES0. If the implementation does not support little-endian operation, SPSR_hyp.E is RES1. On executing an exception return operation in Hyp mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_hyp.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_hyp.E is RES1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]
SError interrupt mask. Set to the value of PSTATE.A on taking an exception to Hyp mode, and copied to PSTATE.A on executing an exception return operation in Hyp mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]
IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Hyp mode, and copied to PSTATE.I on executing an exception return operation in Hyp mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]
FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Hyp mode, and copied to PSTATE.F on executing an exception return operation in Hyp mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]
T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Hyp mode, and copied to PSTATE.T on executing an exception return operation in Hyp mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]
Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Hyp mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Hyp mode.

0b10000 User.
0b10001 FIQ.
0b10010 IRQ.
0b10011 Supervisor.
0b10111 Abort.
0b11010 Hyp.
0b11011  Undefined.
0b11111  System.

Other values are reserved. If SPSR_hyp.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Hyp mode is an illegal return event, as described in *Illegal return events from AArch32 state on page G1-8952*.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing SPSR_hyp

SPSR_hyp is accessible only in Monitor mode.

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS}\{c\}\{q\} \text{ } \text{SPSR}_\text{hyp}
\]

<table>
<thead>
<tr>
<th>R</th>
<th>M</th>
<th>M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>0b1110</td>
</tr>
</tbody>
</table>

\[
\text{MSR}\{c\}\{q\} \text{ } \text{SPSR}_\text{hyp}, \text{ } \text{<Rn>}
\]

<table>
<thead>
<tr>
<th>R</th>
<th>M</th>
<th>M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>0b1110</td>
</tr>
</tbody>
</table>
G8.2.131  SPSR_irq, Saved Program Status Register (IRQ mode)

The SPSR_irq characteristics are:

Purpose

Holds the saved process state when an exception is taken to IRQ mode.

Configurations

AArch32 System register SPSR_irq bits [31:0] are architecturally mapped to AArch64 System register SPSR_irq[31:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to SPSR_irq are UNDEFINED.

Attributes

SPSR_irq is a 32-bit register.

Field descriptions

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to IRQ mode, and copied to PSTATE.N on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to IRQ mode, and copied to PSTATE.Z on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to IRQ mode, and copied to PSTATE.C on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to IRQ mode, and copied to PSTATE.V on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to IRQ mode, and copied to PSTATE.Q on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.
IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to IRQ mode, and copied to PSTATE.IT on executing an exception return operation in IRQ mode.

SPSR_irq.IT must contain a value that is valid for the instruction being returned to.

The IT field is split as follows:
- IT[1:0] is SPSR_irq[26:25].
- IT[7:2] is SPSR_irq[15:10].

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:
Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to IRQ mode, and copied to PSTATE.SSBS on executing an exception return operation in IRQ mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:
Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to IRQ mode, and copied to PSTATE.PAN on executing an exception return operation in IRQ mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:
Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to IRQ mode, and copied to PSTATE.DIT on executing an exception return operation in IRQ mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to IRQ mode, and copied to PSTATE.IL on executing an exception return operation in IRQ mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
GE, bits [19:16]
Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to IRQ mode, and copied to PSTATE.GE on executing an exception return operation in IRQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]
Endianness. Set to the value of PSTATE.E on taking an exception to IRQ mode, and copied to PSTATE.E on executing an exception return operation in IRQ mode.

If the implementation does not support big-endian operation, SPSR_irq.E is RES0. If the implementation does not support little-endian operation, SPSR_irq.E is RES1. On executing an exception return operation in IRQ mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_irq.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_irq.E is RES1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]
SError interrupt mask. Set to the value of PSTATE.A on taking an exception to IRQ mode, and copied to PSTATE.A on executing an exception return operation in IRQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]
IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to IRQ mode, and copied to PSTATE.I on executing an exception return operation in IRQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]
FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to IRQ mode, and copied to PSTATE.F on executing an exception return operation in IRQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]
T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to IRQ mode, and copied to PSTATE.T on executing an exception return operation in IRQ mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]
Mode. Set to the value of PSTATE.M[4:0] on taking an exception to IRQ mode, and copied to PSTATE.M[4:0] on executing an exception return operation in IRQ mode.
0b10000 User.
0b10001 FIQ.
0b10010 IRQ.
0b10011 Supervisor.
0b10111 Abort.
0b11011 Undefined.
0b1111 System.

Other values are reserved. If SPSR_irq.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in IRQ mode is an illegal return event, as described in *Illegal return events from AArch32 state on page G1-8952.*

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing SPSR_irq**

SPSR_irq is accessible in all modes other than User mode and IRQ mode.

Accesses to this register use the following encodings in the System register encoding space:

\[ MRS{<c>}{<q>} <Rd>, SPSR_irq \]

<table>
<thead>
<tr>
<th>R</th>
<th>M</th>
<th>M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

\[ MSR{<c>}{<q>} SPSR_irq, <Rn> \]

<table>
<thead>
<tr>
<th>R</th>
<th>M</th>
<th>M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>0b0000</td>
</tr>
</tbody>
</table>
G8.2.132  SPSR_mon, Saved Program Status Register (Monitor mode)

The SPSR_mon characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to Monitor mode.

**Configurations**

This register is present only when AArch32 is supported. Otherwise, direct accesses to SPSR_mon are UNDEFINED.

**Attributes**

SPSR_mon is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31  | N     | Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Monitor mode, and copied to PSTATE.N on executing an exception return operation in Monitor mode. The reset behavior of this field is:  
| 30  | Z     | Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Monitor mode, and copied to PSTATE.Z on executing an exception return operation in Monitor mode. The reset behavior of this field is:  
| 29  | C     | Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Monitor mode, and copied to PSTATE.C on executing an exception return operation in Monitor mode. The reset behavior of this field is:  
| 28  | V     | Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Monitor mode, and copied to PSTATE.V on executing an exception return operation in Monitor mode. The reset behavior of this field is:  
| 27  | Q     | Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Monitor mode, and copied to PSTATE.Q on executing an exception return operation in Monitor mode. The reset behavior of this field is:  

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
IT, bits \([15:10, 26:25]\)

If-Then. Set to the value of PSTATE.IT on taking an exception to Monitor mode, and copied to PSTATE.IT on executing an exception return operation in Monitor mode.

SPSR_mon.IT must contain a value that is valid for the instruction being returned to.

The IT field is split as follows:
- IT[1:0] is SPSR_mon[26:25].
- IT[7:2] is SPSR_mon[15:10].

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the \(\{J, T\}\) bits determined the AArch32 Instruction set state. Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When \textit{FEAT SSBS is implemented}:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Monitor mode, and copied to PSTATE.SSBS on executing an exception return operation in Monitor mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When \textit{FEAT PAN is implemented}:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Monitor mode, and copied to PSTATE.PAN on executing an exception return operation in Monitor mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When \textit{FEAT DIT is implemented}:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Monitor mode, and copied to PSTATE.DIT on executing an exception return operation in Monitor mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Monitor mode, and copied to PSTATE.IL on executing an exception return operation in Monitor mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
GE, bits [19:16]
Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Monitor mode, and copied to PSTATE.GE on executing an exception return operation in Monitor mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]
Endianness. Set to the value of PSTATE.E on taking an exception to Monitor mode, and copied to PSTATE.E on executing an exception return operation in Monitor mode.

If the implementation does not support big-endian operation, SPSR_mon.E is RES0. If the implementation does not support little-endian operation, SPSR_mon.E is RES1. On executing an exception return operation in Monitor mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_mon.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_mon.E is RES1.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]
SError interrupt mask. Set to the value of PSTATE.A on taking an exception to Monitor mode, and copied to PSTATE.A on executing an exception return operation in Monitor mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]
IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Monitor mode, and copied to PSTATE.I on executing an exception return operation in Monitor mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]
FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Monitor mode, and copied to PSTATE.F on executing an exception return operation in Monitor mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]
T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Monitor mode, and copied to PSTATE.T on executing an exception return operation in Monitor mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]
Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Monitor mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Monitor mode.

0b10000 User.
0b10001 FIQ.
0b10010 IRQ.
0b10011 Supervisor.
0b10110 Monitor.
0b10111 Abort.
0b11010  Hyp.
0b11011  Undefined.
0b11111  System.

Other values are reserved. If SPSR_mon.M[4:0] has a Reserved value, or a value for an
unimplemented Exception level, executing an exception return operation in Monitor mode is an
illegal return event, as described in Illegal return events from AArch32 state on page G1-8952.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_mon

SPSR_mon is only accessible in EL3 modes other than Monitor mode.

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS}\{<c>\}{<q>} <Rd>, \text{SPSR}\_\text{mon}
\]

<table>
<thead>
<tr>
<th>R</th>
<th>M</th>
<th>M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>0b1100</td>
</tr>
</tbody>
</table>

\[
\text{MSR}\{<c>\}{<q>} \text{SPSR}\_\text{mon}, <Rn>
\]

<table>
<thead>
<tr>
<th>R</th>
<th>M</th>
<th>M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>0b1100</td>
</tr>
</tbody>
</table>
### G8.2.133 SPSR_svc, Saved Program Status Register (Supervisor mode)

The SPSR_svc characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to Supervisor mode.

**Configurations**

AArch32 System register SPSR_svc bits [31:0] are architecturally mapped to AArch64 System register `SPSR_EL1[31:0]`.

This register is present only when AArch32 is supported. Otherwise, direct accesses to SPSR_svc are UNDEFINED.

**Attributes**

SPSR_svc is a 32-bit register.

**Field descriptions**

- **N**, bit [31]
  - Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Supervisor mode, and copied to PSTATE.N on executing an exception return operation in Supervisor mode.
  - The reset behavior of this field is:
    - On a Warm reset, this field resets to an architecturally UNKNOWN value.

- **Z**, bit [30]
  - Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Supervisor mode, and copied to PSTATE.Z on executing an exception return operation in Supervisor mode.
  - The reset behavior of this field is:
    - On a Warm reset, this field resets to an architecturally UNKNOWN value.

- **C**, bit [29]
  - Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Supervisor mode, and copied to PSTATE.C on executing an exception return operation in Supervisor mode.
  - The reset behavior of this field is:
    - On a Warm reset, this field resets to an architecturally UNKNOWN value.

- **V**, bit [28]
  - Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Supervisor mode, and copied to PSTATE.V on executing an exception return operation in Supervisor mode.
  - The reset behavior of this field is:
    - On a Warm reset, this field resets to an architecturally UNKNOWN value.

- **Q**, bit [27]
  - Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Supervisor mode, and copied to PSTATE.Q on executing an exception return operation in Supervisor mode.
  - The reset behavior of this field is:
    - On a Warm reset, this field resets to an architecturally UNKNOWN value.
IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Supervisor mode, and copied to PSTATE.IT on executing an exception return operation in Supervisor mode.

SPSR_svc.IT must contain a value that is valid for the instruction being returned to.

The IT field is split as follows:
- IT[1:0] is SPSR_svc[26:25].
- IT[7:2] is SPSR_svc[15:10].

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the \{J, T\} bits determined the AArch32 Instruction set state. Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When \textit{FEAT_SSBS is implemented}:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Supervisor mode, and copied to PSTATE.SSBS on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When \textit{FEAT_PAN is implemented}:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Supervisor mode, and copied to PSTATE.PAN on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When \textit{FEAT_DIT is implemented}:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Supervisor mode, and copied to PSTATE.DIT on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Supervisor mode, and copied to PSTATE.IL on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
GE, bits [19:16]
Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Supervisor mode, and copied to PSTATE.GE on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]
Endianness. Set to the value of PSTATE.E on taking an exception to Supervisor mode, and copied to PSTATE.E on executing an exception return operation in Supervisor mode.

If the implementation does not support big-endian operation, SPSR_svc.E is RES0. If the implementation does not support little-endian operation, SPSR_svc.E is RES1. On executing an exception return operation in Supervisor mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_svc.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_svc.E is RES1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]
SError interrupt mask. Set to the value of PSTATE.A on taking an exception to Supervisor mode, and copied to PSTATE.A on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]
IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Supervisor mode, and copied to PSTATE.I on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]
FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Supervisor mode, and copied to PSTATE.F on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]
T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Supervisor mode, and copied to PSTATE.T on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]
Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Supervisor mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Supervisor mode.

0b10000 User.
0b10001 FIQ.
0b10010 IRQ.
0b10011 Supervisor.
0b10111 Abort.
0b11011 Undefined.
0b1111 System.

Other values are reserved. If SPSR_svc.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Supervisor mode is an illegal return event, as described in *Illegal return events from AArch32 state* on page G1-8952.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing SPSR_svc**

SPSR_svc is accessible in all modes other than User mode and Supervisor mode.

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRS\{<c>\}{<q>}} <Rd>, \text{SPSR\_svc} \\
\text{MSR\{<c>\}{<q>}} \text{SPSR\_svc}, <Rn>
\]

<table>
<thead>
<tr>
<th>R</th>
<th>M</th>
<th>M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>0b0010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th>M</th>
<th>M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>0b0010</td>
</tr>
</tbody>
</table>
G8.2.134  SPSR_und, Saved Program Status Register (Undefined mode)

The SPSR_und characteristics are:

**Purpose**

Holds the saved process state when an exception is taken to Undefined mode.

**Configurations**

AArch32 System register SPSR_und bits [31:0] are architecturally mapped to AArch64 System register SPSR_und[31:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to SPSR_und are UNDEFINED.

**Attributes**

SPSR_und is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>N</td>
<td>Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Undefined mode, and copied to PSTATE.N on executing an exception return operation in Undefined mode. The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>30</td>
<td>Z</td>
<td>Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Undefined mode, and copied to PSTATE.Z on executing an exception return operation in Undefined mode. The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>29</td>
<td>C</td>
<td>Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Undefined mode, and copied to PSTATE.C on executing an exception return operation in Undefined mode. The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>28</td>
<td>V</td>
<td>Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Undefined mode, and copied to PSTATE.V on executing an exception return operation in Undefined mode. The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>27</td>
<td>Q</td>
<td>Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Undefined mode, and copied to PSTATE.Q on executing an exception return operation in Undefined mode. The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
</tbody>
</table>
IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Undefined mode, and copied to PSTATE.IT on executing an exception return operation in Undefined mode.

SPSR_und.IT must contain a value that is valid for the instruction being returned to.

The IT field is split as follows:

- IT[1:0] is SPSR_und[26:25].
- IT[7:2] is SPSR_und[15:10].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

SSBS, bit [23]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to Undefined mode, and copied to PSTATE.SSBS on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to Undefined mode, and copied to PSTATE.PAN on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DIT, bit [21]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to Undefined mode, and copied to PSTATE.DIT on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Undefined mode, and copied to PSTATE.IL on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
GE, bits [19:16]
Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Undefined mode, and copied to PSTATE.GE on executing an exception return operation in Undefined mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]
Endianness. Set to the value of PSTATE.E on taking an exception to Undefined mode, and copied to PSTATE.E on executing an exception return operation in Undefined mode.

If the implementation does not support big-endian operation, SPSR_und.E is RES0. If the implementation does not support little-endian operation, SPSR_und.E is RES1. On executing an exception return operation in Undefined mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_und.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_und.E is RES1.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]
SError interrupt mask. Set to the value of PSTATE.A on taking an exception to Undefined mode, and copied to PSTATE.A on executing an exception return operation in Undefined mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]
IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Undefined mode, and copied to PSTATE.I on executing an exception return operation in Undefined mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]
FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Undefined mode, and copied to PSTATE.F on executing an exception return operation in Undefined mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]
T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Undefined mode, and copied to PSTATE.T on executing an exception return operation in Undefined mode.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]
Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Undefined mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Undefined mode.

0b10000 User.
0b10001 FIQ.
0b10010 IRQ.
0b10011 Supervisor.
0b10111 Abort.
0b11011 Undefined.
0b11111 System.
Other values are reserved. If SPSR_und.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Undefined mode is an illegal return event, as described in Illegal return events from AArch32 state on page G1-8952.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing SPSR Und**

SPSR_und is accessible in all modes other than User mode and Undefined mode.

Accesses to this register use the following encodings in the System register encoding space:

\[ \text{MRS}<c>{<q}> <Rd>, \text{SPSR}_{\text{und}} \]

\[
\begin{array}{c|c|c}
\text{R} & \text{M} & \text{M1} \\
0b1 & 0b1 & 0b0110 \\
\end{array}
\]

\[ \text{MSR}<c>{<q}> \text{SPSR}_{\text{und}}, <Rn> \]

\[
\begin{array}{c|c|c}
\text{R} & \text{M} & \text{M1} \\
0b1 & 0b1 & 0b0110 \\
\end{array}
\]
TCMTR, TCM Type Register

The TCMTR characteristics are:

**Purpose**

Provides information about the implementation of the TCM.

**Configurations**

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TCMTR are UNDEFINED.

If EL1 or above can use AArch32 then this register must be implemented.

**Attributes**

TCMTR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

**Accessing TCMTR**

Accesses to this register use the following encodings in the System register encoding space:

`MRC{<c>}{<q>}, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}`

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TID1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID1 == '1' then
    AArch32.TakeHypTrapException(0x03);
else
  R[t] = TCMTR;
elseif PSTATE.EL == EL2 then
  R[t] = TCMTR;
elseif PSTATE.EL == EL3 then
  R[t] = TCMTR;
G8.2.136 TLBIALL, TLB Invalidate All

The TLBIALL characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk. The entries that are invalidated are as follows:

- If executed at EL1, all entries that:
  - Would be required for the EL1&0 translation regime.
  - Match the current VMID, if EL2 is implemented and enabled in the current Security state.
- If executed in Secure state when EL3 is using AArch32, all entries that would be required for the Secure PL1&0 translation regime.
- If executed at EL2, and if EL2 is enabled in the current Security state, the stage 1 or stage 2 translation table entries that would be required for the PL1&0 translation regime and matches the current VMID.

The invalidation only applies to the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TLBIALL are UNDEFINED.

**Attributes**

TLBIALL is a 32-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

**Executing TLBIALL instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[ MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} \]

<table>
<thead>
<tr>
<th></th>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0b1111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && HCRX_EL2.FnXS == '1' then
      AArch32.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBIExcludeXS);
    else
      coproc opc1 CRn CRm opc2
      0b1111 0b000 0b1000 0b0111 0b000
    end
  end
else
  coproc opc1 CRn CRm opc2
  0b1111 0b000 0b1000 0b0111 0b000
end
AArch32.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_AllAttr);
elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR.FB == '1' then
  AArch32.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_AllAttr);
else
  if IsFeatureImplemented(FEAT_XS) && !ELUsingAArch32(EL2) && IsFeatureImplemented(FEAT_HCX) &&
     IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then
    AArch32.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_ExcludeXS);
  else
    AArch32.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AllAttr);
  elsif PSTATE.EL == EL2 then
    AArch32.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AllAttr);
  elseif PSTATE.EL == EL3 then
    AArch32.TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL30, Shareability_NSH, TLBI_ExcludeXS);
  else
    AArch32.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AllAttr);
  endif
G8.2.137 TLBIALLH, TLB Invalidate All, Hyp mode

The TLBIALLH characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are
from any level of the translation table walk that would be required for the Non-secure EL2
translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configurations

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses
to TLBIALLH are UNDEFINED.

Attributes

TLBIALLH is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing TLBIALLH instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is
CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction is treated as a NOP.
• The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{MCR}\{c\}\{q\} \ <\text{coproc}, \ {#}<\text{opc1}, <\text{Rt}, <\text{CRn}, <\text{CRm}, {#}<\text{opc2}\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b000</td>
</tr>
</tbody>
</table>

\[
\text{if PSTATE.EL == EL0 then UNDEFINED;}
\text{elsif PSTATE.EL == EL1 then}
\text{ if EL2Enabled() \&\& !ELUsingAArch32(EL2) \&\& HSTR_EL2.T8 == '1' then}
\text{ AArch64.AArch32SystemAccessTrap(EL2, 0x03);}
\text{ elsif EL2Enabled() \&\& ELUsingAArch32(EL2) \&\& HSTR.T8 == '1' then}
\text{ AArch32.TakeHypTrapException(0x03);}
\text{ else UNDEFINED;}
\text{elsif PSTATE.EL == EL2 then}
\text{ AArch32.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Shareability_NSH, TLBI_AllAttr);}
\text{elsif PSTATE.EL == EL3 then}
\text{ if !HaveEL(EL2) then UNDEFINED;}
\text{ else AArch32.TLBI_ALL(SS_NonSecure, Regime_EL2, Shareability_NSH, TLBI_AllAttr);}
\]
G8.2.138  TLBIALLHIS, TLB Invalidate All, Hyp mode, Inner Shareable

The TLBIALLHIS characteristics are:

**Purpose**

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for the Non-secure EL2 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to TLBIALLHIS are **UNDEFINED**.

**Attributes**

TLBIALLHIS is a 32-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by `<Rt>` is ignored.

**Executing TLBIALLHIS instruction**

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is **CONSTRAINED UNPREDICTABLE**, and one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```plaintext
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

```plaintext
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
    endif;
elsif PSTATE.EL == EL2 then
    AArch32.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Shareability_ISH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    else
        AArch32.TLBI_ALL(SS_NonSecure, Regime_EL2, Shareability_ISH, TLBI_AllAttr);
    endif;
else
    AArch32.TLBI_ALL(Regime_EL2, Shareability_ISH, TLBI_AllAttr);
```
TLBALLIS, TLB Invalidate All, Inner Shareable

The TLBALLIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk. The entries that are invalidated are as follows:

- If executed at EL1, all entries that:
  - Would be required for the EL1&0 translation regime.
  - Match the current VMID, if EL2 is implemented and enabled in the current Security state.
- If executed in Secure state when EL3 is using AArch32, all entries that would be required for the Secure PL1&0 translation regime.
- If executed at EL2, and if EL2 is enabled in the current Security state, the stage 1 or stage 2 translation table entries that would be required for the PL1&0 translation regime and matches the current VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configurations

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TLBALLIS are UNDEFINED.

Attributes

TLBALLIS is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing TLBALLIS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{MCR\{c\}\{q\} <coproc>, \{#\}<opc1>, <Rt>, <CRn>, <CRm\}, \{#\}<opc2>}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TTLBIS == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR.TTLB == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR2.TTLBIS == '1' then
  AArch32.TakeHypTrapException(0x03);
else
    if IsFeatureImplemented(FEAT_XS) && !ELUsingAArch32(EL2) && IsFeatureImplemented(FEAT_HCX) &&
    IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then
        AArch32.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
        TLBI_ExcludeXS);
    else
        AArch32.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
        TLBI_AllAttr);
    elsif PSTATE.EL == EL2 then
        AArch32.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_AllAttr);
    elsif PSTATE.EL == EL3 then
        AArch32.TLBI_ALL(SecurityStateAtEL(EL3), Regime_EL30, Shareability_ISH, TLBI_ExcludeXS);
G8.2.140 TLBIALLNSNH, TLB Invalidate All, Non-Secure Non-Hyp

The TLBIALLNSNH characteristics are:

**Purpose**

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for stage 1 or stage 2 of the Non-secure PL1&0 translation regime, regardless of the associated VMID.

The invalidation only applies to the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to TLBIALLNSNH are UNDEFINED.

**Attributes**

TLBIALLNSNH is a 32-bit System instruction.

**Field descriptions**

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

**Executing TLBIALLNSNH instruction**

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{MCR}\{<c>\}{<q>}\{\#}\text{<opc1>}, \langle Rt\rangle, \langle CRn\rangle, \langle CRm\rangle, \{\#\text{<opc2>}}
\]

```plaintext
<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b100</td>
</tr>
</tbody>
</table>
```

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_NSH, TLBI_AllAttr);
  else
    UNDEFINED;
  endif
elsif PSTATE.EL == EL2 then
  AArch32.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_NSH, TLBI_AllAttr);
elsif PSTATE.EL == EL3 then
  if !HaveEL(EL2) then
    UNDEFINED;
  else
    AArch32.TLBI_ALL(SS_NonSecure, Regime_EL10, Shareability_NSH, TLBI_AllAttr);
  endif
```
G8.2.141 TLBIALLNSNHIS, TLB Invalidate All, Non-Secure Non-Hyp, Inner Shareable

The TLBIALLNSNHIS characteristics are:

**Purpose**
If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for stage 1 or stage 2 of the Non-secure PL1&0 translation regime, regardless of the associated VMID.
The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

**Configurations**
This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to TLBIALLNSNHIS are UNDEFINED.

**Attributes**
TLBIALLNSNHIS is a 32-bit System instruction.

**Field descriptions**
This instruction has no applicable fields.
The value in the register specified by <Rt> is ignored.

**Executing TLBIALLNSNHIS instruction**
If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[ MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
    endif;
elif PSTATE.EL == EL2 then
    AArch32.TLB_All(SecurityStateAtEL(EL1), Regime_EL10, Shareability_NSH, TLB_AllAttr);
elif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    else
        AArch32.TLB_ALL(SS_NonSecure, Regime_EL10, Shareability_NSH, TLB_AllAttr);
    endif;
else
    AArch32.TLB_ALL(SecurityStateAtEL(EL1), Regime_EL10, Shareability_NSH, TLB_AllAttr);
endif;
**G8.2.142 TLBIASID, TLB Invalidate by ASID match**

The TLBIASID characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TLBIASID are **UNDEFINED**.

**Attributes**

TLBIASID is a 32-bit System instruction.

**Field descriptions**

![Field Descriptions Diagram]

**Bits [31:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this System instruction.

**Executing TLBIASID instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && HCRX_EL2.FnXS == '1' then
      AArch32.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_ExcludeXS, R[t]);
    else
      AArch32.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_AllAttr, R[t]);
    end if
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.FB == '1' then
    AArch32.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_AllAttr, R[t]);
  else
    if IsFeatureImplemented(FEAT_XS) && !ELUsingAArch32(EL2) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then
      AArch32.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_ExcludeXS, R[t]);
    else
      AArch32.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AllAttr, R[t]);
    end if
  elsif PSTATE.EL == EL2 then
    AArch32.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBI_AllAttr, R[t]);
  elseif PSTATE.EL == EL3 then
    AArch32.TLBI_ASID(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_NSH, TLBI_AllAttr, R[t]);
  end if

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b010</td>
</tr>
</tbody>
</table>
G8.2.143 TLBIASIDIS, TLB Invalidate by ASID match, Inner Shareable

The TLBIASIDIS characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TLBIASIDIS are UNDEFINED.

**Attributes**

TLBIASIDIS is a 32-bit System instruction.

**Field descriptions**

```
31 8
RES0
7 0
ASID
```

**Bits [31:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this System instruction.

**Executing TLBIASIDIS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        if IsFeatureImplemented(FEAT_XS) && !ELUsingAArch32(EL2) && IsFeatureImplemented(FEAT_HCX) &&
            IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then
            AArch32.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_ExcludeXS, R[t]);
        else
            AArch32.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_AllAttr, R[t]);
        end
    end
elsif PSTATE.EL == EL2 then
    AArch32.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBI_AllAttr, R[t]);
else
    if PSTATE.EL == EL3 then
        AArch32.TLBI_ASID(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_ISH, TLBI_AllAttr, R[t]);
else
        AArch32.TLBI_ASID(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_ISH, TLBI_AllAttr, R[t]);
eend

G8.2.144 TLBIIPAS2, TLB Invalidate by Intermediate Physical Address, Stage 2

The TLBIIPAS2 characteristics are:

**Purpose**

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that meet the following requirements:
- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- SCR.NS is 1.
- The entry would be used for the specified IPA.
- The entry would be used with the current VMID.
- The entry would be required for the PL1&0 translation regime.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation only applies to the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to TLBIIPAS2 are UNDEFINED.

---

**Note**

This System instruction is not implemented in architecture versions before Armv8.

---

**Attributes**

TLBIIPAS2 is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0</td>
</tr>
<tr>
<td>28-27</td>
<td>IPA[39:12]</td>
</tr>
</tbody>
</table>

**Bits [31:28]**

Reserved, RES0.

**IPA[39:12], bits [27:0]**

Bits[39:12] of the intermediate physical address to match.

**Executing TLBIIPAS2 instruction**

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:
**MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>**}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  AArch32.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, R[t]);
else
  AArch32.TLBI_IPAS2(SS_NonSecure, Regime_EL10, VMID_NONE, Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, R[t]);

if !HaveEL(EL2) then
  UNDEFINED;
else
  if SCR.NS == '0' then
    return;
else
    AArch32.TLBI_IPAS2(SS_NonSecure, Regime_EL10, VMID_NONE, Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, R[t]);

G8.2.145 TLBIIPAS2IS, TLB Invalidate by Intermediate Physical Address, Stage 2, Inner Shareable

The TLBIIPAS2IS characteristics are:

**Purpose**

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- SCR.NS is 1.
- The entry would be used for the specified IPA.
- The entry would be used with the current VMID.
- The entry would be required for the PL1&0 translation regime.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to TLBIIPAS2IS are UNDEFINED.

**Note**

This System instruction is not implemented in architecture versions before Armv8.

**Attributes**

TLBIIPAS2IS is a 32-bit System instruction.

**Field descriptions**

```
<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>IPA[39:12]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Bits [31:28]**

Reserved, RES0.

**IPA[39:12], bits [27:0]**

Bits[39:12] of the intermediate physical address to match.

**Executing TLBIIPAS2IS instruction**

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:
MCR\{c\}\{q\} \{coproc\}, \{#\}\{opc1\}, \{Rt\}, \{CRn\}\{, \{#\}\{opc2\}\}
G8.2.146   TLBIIPAS2L, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level

The TLBIIPAS2L characteristics are:

Panel: Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that meet
the following requirements:
• The entry is a stage 2 only translation table entry, from the final level of the translation table
  walk.
• SCR.NS is 1.
• The entry would be used for the specified IPA.
• The entry would be used with the current VMID.
• The entry would be required for the PL1&0 translation regime.
The invalidation is not required to apply to caching structures that combine stage 1 and stage 2
translation table entries.
The invalidation only applies to the PE that executes this System instruction.

Panel: Configurations

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses
to TLBIIPAS2L are UNDEFINED.

Panel: Note

This System instruction is not implemented in architecture versions before Armv8.

Panel: Attributes

TLBIIPAS2L is a 32-bit System instruction.

Panel: Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>...</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>IPA[39:12]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits [31:28]
Reserved, RES0.

IPA[39:12], bits [27:0]

Bits[39:12] of the intermediate physical address to match.

Panel: Executing TLBIIPAS2L instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is
CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:
• The instruction is UNDEFINED.
• The instruction is treated as a NOP.
• The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:
AArch32 System Register Descriptions
G8.2 General system control registers

\[
\texttt{MCR} \{\texttt{c}<q>\} \texttt{<coproc>, (#)<opc1>, <Rt>, <CRn>, CRm}\}; \{ (#)<opc2>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0100</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  AArch32.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBIlevel_Last, TLBI_AllAttr, R[t]);
elsif PSTATE.EL == EL3 then
  if !HaveEL(EL2) then
    UNDEFINED;
  elsif SCR.NS == '0' then
    return;
  else
    AArch32.TLBI_IPAS2(SS_NonSecure, Regime_EL10, VMID_NONE, Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
G8.2.147 TLBIPAS2LIS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, Inner Shareable

The TLBIPAS2LIS characteristics are:

**Purpose**

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- SCR.NS is 1.
- The entry would be used for the specified IPA.
- The entry would be used with the current VMID.
- The entry would be required for the PL1&0 translation regime.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to TLBIPAS2LIS are UNDEFINED.

--- Note ---

This System instruction is not implemented in architecture versions before Armv8.

**Attributes**

TLBIPAS2LIS is a 32-bit System instruction.

**Field descriptions**

```
<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>IPA[39:12]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- **Bits [31:28]**
  
  Reserved, RES0.

- **IPA[39:12], bits [27:0]**
  
  Bits[39:12] of the intermediate physical address to match.

**Executing TLBIPAS2LIS instruction**

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
  AArch32.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, R[t]);
elsif PSTATE.EL == EL3 then
  if !HaveEL(EL2) then
    UNDEFINED;
  elsif SCR.NS == '0' then
    return;
  else
    AArch32.TLBI_IPAS2(SS_NonSecure, Regime_EL10, VMID_NONE, Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, R[t]);
G8.2.148 TLBIMVA, TLB Invalidate by VA

The TLBIMVA characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TLBIMVA are UNDEFINED.

**Attributes**

TLBIMVA is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA</td>
<td>RES0</td>
<td>ASID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**VA, bits [31:12]**

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

**Bits [11:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.
Executing TLBIMVA instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[ MCR\{<c>\}{<q>}\ <coproc>, \{#\}<opc1>, <Rt>, <CRn>, <CRm>{, \{#\}<opc2>} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_ExcludeXS, R[t]);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
        if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && HCRX_EL2.FnXS == '1' then
            AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_ExcludeXS, R[t]);
        else
            AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, R[t]);
    elsif PSTATE.EL == EL2 then
        AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, R[t]);
    elsif PSTATE.EL == EL3 then
        AArch32.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, R[t]);
else
    if PSTATE.EL == EL3 then
        AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, R[t]);
    elsif PSTATE.EL == EL2 then
        AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, R[t]);
    elise
    PSTATE.EL == EL3 then
        AArch32.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, R[t]);
G8.2.149 TLBIMVAA, TLB Invalidate by VA, All ASID

The TLBIMVAA characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified address.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TLBIMVAA are **UNDEFINED**.

**Attributes**

TLBIMVAA is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA, bits [31:12]</td>
<td>Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.</td>
</tr>
<tr>
<td>Bits [11:0]</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**Executing TLBIMVAA instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{MCR}\{<c>\}{<q>\} <\text{coproc}, \{#}<\text{opc}1>, <\text{Rt}, <\text{CRn}, <\text{CRm}, \{#}<\text{opc}2>\}\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
elif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
    AArch32.TakeHypTrapException(0x03);
elif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && HCRX_EL2.FnXS == '1' then
        AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
        TLBILevel_Any, TLBI_ExcludeXS, R[t]);
elif !ELUsingAArch32(EL2) && IsFeatureImplemented(FEAT_HCX) && HCRX_EL2.FnXS == '1' then
        AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
        TLBILevel_Any, TLBI_AllAttr, R[t]);
elif EL2Enabled() && ELUsingAArch32(EL2) && HCR.FB == '1' then
        AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any,
        TLBI_AllAttr, R[t]);
elif PSTATE.EL == EL2 then
    AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
    TLBILevel_Any, TLBI_AllAttr, R[t]);
else
    if IsFeatureImplemented(FEAT_XS) && !ELUsingAArch32(EL2) && IsFeatureImplemented(FEAT_HCX) &&
    IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then
        AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH,
        TLBILevel_Any, TLBI_ExcludeXS, R[t]);
else
    AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH,
    TLBILevel_Any, TLBI_AllAttr, R[t]);
eelif PSTATE.EL == EL3 then
    AArch32.TLBI_VAA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_NSH, TLBILevel_Any,
    TLBI_AllAttr, R[t]);
eelse
    AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Any,
    TLBI_AllAttr, R[t]);

G8.2.150 TLBIMVAAIS, TLB Invalidate by VA, All ASID, Inner Shareable

The TLBIMVAAIS characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified address.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TLBIMVAAIS are **UNDEFINED**.

**Attributes**

TLBIMVAAIS is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>VA, bits [31:12]</th>
<th>31 12</th>
<th>11 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Executing TLBIMVAAIS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:
if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TTLBIS == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        if IsFeatureImplemented(FEAT_XS) && !ELUsingAArch32(EL2) && IsFeatureImplemented(FEAT_HCX) &&
            IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then
            AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_ExcludeXS, R[t]);
        else
            AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, R[t]);
        end if;
    end if;
elsif PSTATE_EL == EL2 then
    AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, R[t]);
elsif PSTATE_EL == EL3 then
    AArch32.TLBI_VAA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, R[t]);

---

<p>|MCR{&lt;c&gt;}{&lt;q&gt;}&lt;coproc&gt;, {#}&lt;opc1&gt;, &lt;Rt&gt;, &lt;CRn&gt;, &lt;CRm&gt;{, {#}&lt;opc2&gt;}|</p>
<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b011</td>
</tr>
</tbody>
</table>

---
G8.2.151 TLBIMVAAL, TLB Invalidate by VA, All ASID, Last level

The TLBIMVAAL characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified address.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TLBIMVAAL are UNDEFINED.

---

**Note**

This System instruction is not implemented in architecture versions before Armv8.

---

**Attributes**

TLBIMVAAL is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>VA, bits [31:12] Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.</td>
</tr>
<tr>
<td>11-0</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**Executing TLBIMVAAL instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && HCRX_EL2.FnXS == '1' then
      AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_ExcludeXS, R[t]);
    else
      AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, R[t]);
    else
      AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, R[t]);
    else
      if IsFeatureImplemented(FEAT_XS) && !ELUsingAArch32(EL2) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then
        AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_ExcludeXS, R[t]);
      else
        AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
      elsif PSTATE.EL == EL2 then
        AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
      elsif PSTATE.EL == EL3 then
        AArch32.TLBI_VAA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
      else
        AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
      else
        AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
      else
        if PSTATE.EL == EL2 then
          AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
        elsif PSTATE.EL == EL3 then
          AArch32.TLBI_VAA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
        else
          AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
        end
      end
    end
  end
elsif PSTATE.EL == EL2 then
  AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
elsif PSTATE.EL == EL3 then
  AArch32.TLBI_VAA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
else
  AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
end
G8.2.152 TLBIMVAALIS, TLB Invalidate by VA, All ASID, Last level, Inner Shareable

The TLBIMVAALIS characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified address.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TLBIMVAALIS are UNDEFINED.

---

**Note**

This System instruction is not implemented in architecture versions before Armv8.

**Attributes**

TLBIMVAALIS is a 32-bit System instruction.

**Field descriptions**

```
  31 12  11  0

VA    RES0
```

VA, bits [31:12]

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Bits [11:0]

Reserved, RES0.

**Executing TLBIMVAALIS instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR2.TTLBIS == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TTLBIS == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif IsFeatureImplemented(FEAT_XS) && !ELUsingAArch32(EL2) && IsFeatureImplemented(FEAT_HCX) &&
        IsHCRXEL2Enabled() && HCRX_EL2.FnxXS == '1' then
        AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH,
        TLBILevel_Last, TLBIEcludeXS, R[t]);
    else
        AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID_NONE, Shareability_ISH,
        TLBILevel_Last, TLBI_AlAttr, R[t]);
    elsif PSTATE.EL == EL2 then
        AArch32.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last,
        TLBI_AlAttr, R[t]);
    elseif PSTATE.EL == EL3 then
        AArch32.TLBI_VAA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_ISH, TLBILevel_Last,
        TLBI_AlAttr, R[t]);
else
    end if
end if

---

### MCR<coproc>,<CRn>,<CRm>{, <opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b111</td>
</tr>
</tbody>
</table>
G8.2.153 TLBIMVAH, TLB Invalidate by VA, Hyp mode

The TLBIMVAH characteristics are:

**Purpose**

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for the Non-secure EL2 translation regime and used to translate the specified address.

The invalidation only applies to the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to TLBIMVAH are UNDEFINED.

**Attributes**

TLBIMVAH is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA, bits [31:12]</td>
<td>Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.</td>
</tr>
<tr>
<td>Bits [11:0]</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**Executing TLBIMVAH instruction**

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

```
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
```
UNDEFINED;
elsim PSTATE_EL == EL2 then
   AArch32.TLBI VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Shareability_NSH, TLBILevel_Any,
   TLBI_AllAttr, R[t]);
elsim PSTATE_EL == EL3 then
   if !HaveEL(EL2) then
      UNDEFINED;
   else
      AArch32.TLBI VA(SS_NonSecure, Regime_EL2, VMID[], Shareability_NSH, TLBILevel_Any, TLBI_AllAttr,
      R[t]);
G8.2.154  TLBIMVAHIS, TLB Invalidate by VA, Hyp mode, Inner Shareable

The TLBIMVAHIS characteristics are:

**Purpose**

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for the Non-secure EL2 translation regime and used to translate the specified address.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to TLBIMVAHIS are UNDEFINED.

**Attributes**

TLBIMVAHIS is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA</td>
<td>Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.</td>
</tr>
<tr>
<td>RES0</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**Executing TLBIMVAHIS instruction**

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
else if PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else if EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
else
    UNDEFINED;
elif PSTATE.EL == EL2 then
    AArch32.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Shareability_ISH, TLBLevel_Any,
    TLBI_AllAttr, R[t]);
elif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    else
        AArch32.TLBI_VA(SS_NonSecure, Regime_EL2, VMID[], Shareability_ISH, TLBLevel_Any, TLBI_AllAttr,
        R[t]);
G8.2.155 TLBIMVAIS, TLB Invalidate by VA, Inner Shareable

The TLBIMVAIS characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TLBIMVAIS are UNDEFINED.

**Attributes**

TLBIMVAIS is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA</td>
<td>RES0</td>
<td>ASID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**VA, bits [31:12]**

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

**Bits [11:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.
Executing TLBIMVAIS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[ MCR{<c>}{<q>} <\text{coproc}>, \{#<\text{opc1}>\}, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>, \{#<\text{opc2}>\} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(@0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TTLBIS == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
    AArch32.TakeHypTrapException(@0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TTLBIS == '1' then
    AArch32.TakeHypTrapException(@0x03);
  else
    if IsFeatureImplemented(FEAT_XS) && ELUsingAArch32(EL2) && IsFeatureImplemented(FEAT_HCX) &&
       IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then
      AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, R[t]);
    else
      AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, R[t]);
    endif
  endif
else
  if PSTATE.EL == EL2 then
    AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, R[t]);
  elsif PSTATE.EL == EL3 then
    AArch32.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_ISH, TLBILevel_Any, TLBI_AllAttr, R[t]);
  endif
end
**TLBIMVAL, TLB Invalidate by VA, Last level**

The TLBIMVAL characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TLBIMVAL are **UNDEFINED**.

This System instruction is not implemented in architecture versions before Armv8.

**Attributes**

TLBIMVAL is a 32-bit System instruction.

**Field descriptions**

```
+---+---+---+---+---+---+
|   |   |   | VA|   | ASID|
+---+---+---+---+---+---+
```

**VA, bits [31:12]**

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

**Bits [11:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

**Executing TLBIMVAL instruction**

Accesses to this instruction use the following encodings in the System instruction encoding space:
MCR{<c>}{<q>} {<coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TTLB == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TTLB == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.FB == '1' then
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && HCRX_EL2.FnXS == '1' then
      AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[ ], Shareability_ISH, TLBILevel_Last, TLBI_EcludeXS, R[t]);
    else
      AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[ ], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, R[t]);
    end
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCRX_EL2.Enable() && !IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) && HCRX_EL2.FnXS == '1' then
    AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[ ], Shareability_NSH, TLBILevel_Last, TLBI_EcludeXS, R[t]);
  else
    AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[ ], Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
  end
elsif PSTATE.EL == EL2 then
  AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[ ], Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
elsif PSTATE.EL == EL3 then
  AArch32.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
else
  if IsFeatureImplemented(FEAT_XS) && !ELUsingAArch32(EL2) && IsFeatureImplemented(FEAT_HCX) &&
  IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then
    AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[ ], Shareability_NSH, TLBILevel_Last, TLBI_EcludeXS, R[t]);
  else
    AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[ ], Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
  end
elsif PSTATE.EL == EL2 then
  AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[ ], Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
elsif PSTATE.EL == EL3 then
  AArch32.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_NSH, TLBILevel_Last, TLBI_AllAttr, R[t]);
G8.2.157 TLBIMVALH, TLB Invalidate by VA, Last level, Hyp mode

The TLBIMVALH characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from the final level of the translation table walk that would be required for the Non-secure EL2 translation regime and used to translate the specified address.

The invalidation only applies to the PE that executes this System instruction.

Configurations

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to TLBIMVALH are UNDEFINED.

This System instruction is not implemented in architecture versions before Armv8.

Attributes

TLBIMVALH is a 32-bit System instruction.

Field descriptions

<table>
<thead>
<tr>
<th></th>
<th>VA</th>
<th></th>
<th>RES0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>12</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVALH instruction

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction is treated as a NOP.
• The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[ \text{MCR}\{<c>\}\{<q>\} <coproc>, \{#<opc1>\}, <Rt>, <CRn>, <CRm>\{, \{#<opc2>\}\} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1000</td>
<td>0b0111</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    end
end
AArch32.TakeHypTrapException(0x03);
else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch32.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Shareability_NSH, TLBILevel_Last,
    TLBI_AllAttr, R[t]);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    else
        AArch32.TLBI_VA(SS_NonSecure, Regime_EL2, VMID[], Shareability_NSH, TLBILevel_Last, TLBI_AllAttr,
        R[t]);
G8.2.158 TLBIMVALHIS, TLB Invalidate by VA, Last level, Hyp mode, Inner Shareable

The TLBIMVALHIS characteristics are:

**Purpose**

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from the final level of the translation table walk that would be required for the Non-secure EL2 translation regime and used to translate the specified address.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to TLBIMVALHIS are UNDEFINED.

This System instruction is not implemented in architecture versions before Armv8.

**Attributes**

TLBIMVALHIS is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>VA, bits [31:12]</th>
<th>RES0, bits [11:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.</td>
<td></td>
</tr>
<tr>
<td>Reserved, RES0.</td>
<td></td>
</tr>
</tbody>
</table>

**Executing TLBIMVALHIS instruction**

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
MCR\{<c>\}{<q>\} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
\]

\[
\begin{array}{cccccc}
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
0b1111 & 0b100 & 0b1000 & 0b0011 & 0b101
\end{array}
\]

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  
```
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
else
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch32.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Shareability_ISH, TLBLevel_Last,
    TLBI_AllAttr, R[t]);
elsif PSTATE.EL == EL3 then
    if !HaveEL(EL2) then
        UNDEFINED;
    else
        AArch32.TLBI_VA(SS_NonSecure, Regime_EL2, VMID[], Shareability_ISH, TLBLevel_Last, TLBI_AllAttr,
        R[t]);
    endif
    if !HaveEL(EL2) then
        UNDEFINED;
    else
        AArch32.TLBI_VA(SS_NonSecure, Regime_EL2, VMID[], Shareability_ISH, TLBLevel_Last, TLBI_AllAttr,
        R[t]);
    endif
endif
G8.2.159 TLBIMVALIS, TLB Invalidate by VA, Last level, Inner Shareable

The TLBIMVALIS characteristics are:

**Purpose**

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

**Configurations**

This instruction is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TLBIMVALIS are UNDEFINED.

This System instruction is not implemented in architecture versions before Armv8.

**Attributes**

TLBIMVALIS is a 32-bit System instruction.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>VA</td>
</tr>
<tr>
<td>12</td>
<td>RES0</td>
</tr>
<tr>
<td>8</td>
<td>ASID</td>
</tr>
</tbody>
</table>

**VA, bits [31:12]**

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

**Bits [11:8]**

Reserved, RES0.

**ASID, bits [7:0]**

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.
Executing TLBIMVALIS instruction

Accesses to this instruction use the following encodings in the System instruction encoding space:

\[
\text{MCR\{}\{c\}\{q\}\text{ <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>, (#<opc2>)}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b000</td>
<td>0b1000</td>
<td>0b0011</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingArch32(EL2) && HSTR.EL2.T8 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingArch32(EL2) && HSTR.T8 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingArch32(EL2) && HCR.EL2.TTLB == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingArch32(EL2) && HCR.EL2.TTLBIS == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingArch32(EL2) && HCR.TTLB == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingArch32(EL2) && HCR2.TTLBIS == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif PSTATE.EL == EL2 then
    AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, R[t]);
  elsif PSTATE.EL == EL3 then
    AArch32.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, R[t]);
else
  if IsFeatureImplemented(FEAT_XS) && ELUsingArch32(EL2) && IsFeatureImplemented(FEAT_HCX) && IsHCRXEL2Enabled() && HCX.EL2.FnXS == '1' then
    AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, R[t]);
  else
    AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, R[t]);
  elsif PSTATE.EL == EL2 then
    AArch32.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, R[t]);
  elsif PSTATE.EL == EL3 then
    AArch32.TLBI_VA(SecurityStateAtEL(EL3), Regime_EL30, VMID_NONE, Shareability_ISH, TLBILevel_Last, TLBI_AllAttr, R[t]);
G8.2.160   TLBTR, TLB Type Register

The TLBTR characteristics are:

**Purpose**

Provides information about the TLB implementation. The register must define whether the implementation provides separate instruction and data TLBs, or a unified TLB. Normally, the IMPLEMENTATION DEFINED information in this register includes the number of lockable entries in the TLB.

**Configurations**

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TLBTR are UNDEFINED.

**Attributes**

TLBTR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED, bits [31:1]</td>
<td>IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>nU, bit [0]</td>
<td>Not Unified TLB. Indicates whether the implementation has a unified TLB:</td>
</tr>
<tr>
<td>0b0</td>
<td>Unified TLB.</td>
</tr>
<tr>
<td>0b1</td>
<td>Separate Instruction and Data TLBs.</td>
</tr>
</tbody>
</table>

**Accessing TLBTR**

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC{<c>}{<q>}, <coproc>, {#<opc1>}, <Rt>, <CRn>, <CRm>{, {#<opc2>}} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b011</td>
</tr>
</tbody>
</table>

```java
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.EL2.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TID1 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    R[t] = TLBTR;
  end if;
elsif PSTATE.EL == EL2 then
  R[t] = TLBTR;
```
elsif PSTATE.EL == EL3 then
    R[t] = TLBTR;
G8.2.161 TPIDRPRW, PL1 Software Thread ID Register

The TPIDRPRW characteristics are:

Purpose

Provides a location where software executing at EL1 or higher can store thread identifying information that is not visible to software executing at EL0, for OS management purposes.

The PE makes no use of this register.

Configurations

AArch32 System register TPIDRPRW bits [31:0] are architecturally mapped to AArch64 System register TPIDR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TPIDRPRW are UNDEFINED.

Note

The PE never updates this register.

Attributes

TPIDRPRW is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread ID</td>
<td></td>
</tr>
</tbody>
</table>

Bits [31:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TPIDRPRW

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>}\{<q>\} <\text{coproc}>, \{#<opc1>\}, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>, \{#<opc2>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b100</td>
</tr>
</tbody>
</table>

\[
\text{if PSTATE.EL == EL0 then UNDEFINED;}
\text{elsif PSTATE.EL == EL1 then}
\text{if EL2Enabled() \&\& !ELUsingAArch32(EL2) \&\& HSTR.EL2.T13 == '1' then}
\text{AArch64.AArch32SystemAccessTrap(EL2, 0x03);}
\text{elsif EL2Enabled() \&\& ELUsingAArch32(EL2) \&\& HSTR.T13 == '1' then}
\text{AArch32.TakeHypTrapException(0x03);}
\text{elsif HaveEL(EL3) \&\& ELUsingAArch32(EL3) then}
\text{R[t] = TPIDRPRW_NS;}
\text{else}
\text{R[t] = TPIDRPRW;}
\text{elsif PSTATE.EL == EL2 then}
\]
if HaveEL(EL3) && ELUsingAArch32(EL3) then
  R[t] = TPIDRPRW_NS;
else
  R[t] = TPIDRPRW;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    R[t] = TPIDRPRW_S;
  else
    R[t] = TPIDRPRW_NS;
endif

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif HaveEL(EL3) && ELUsingAArch32(EL3) then
    TPIDRPRW_NS = R[t];
  else
    TPIDRPRW = R[t];
  endif
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    TPIDRPRW_NS = R[t];
  else
    TPIDRPRW = R[t];
  endif
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    TPIDRPRW_S = R[t];
  else
    TPIDRPRW_NS = R[t];
  endif
G8.2.162 TPIDRURO, PL0 Read-Only Software Thread ID Register

The TPIDRURO characteristics are:

**Purpose**

Provides a location where software executing at EL1 or higher can store thread identifying information that is visible to software executing at EL0, for OS management purposes.

The PE makes no use of this register.

**Configurations**

AArch32 System register TPIDRURO bits [31:0] are architecturally mapped to AArch64 System register TPIDRRO_EL0[31:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to TPIDRURO are UNDEFINED.

--- Note ---

The PE never updates this register.

**Attributes**

TPIDRURO is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>Bit 0</th>
</tr>
</thead>
</table>
| Thread ID

**Bits [31:0]**

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing TPIDRURO**

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2}> \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b011</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif EL2Enabled() && !ELUsingAArch32(EL1) && !HaveEL(EL3) ||
        SCR_EL3.FGTEn == '1') && HGFRTR_EL2.TPIDRRO_EL0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        R[t] = TPIDRURO;
    elseif PSTATE_EL == EL1 then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
```
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elszf EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
elszf HaveEL(EL3) && ELUsingAArch32(EL3) then
    R[t] = TPIDRURO_NS;
else
    R[t] = TPIDRURO;
elszf PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        R[t] = TPIDRURO_NS;
    else
        R[t] = TPIDRURO;
elszf PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        R[t] = TPIDRURO_S;
    else
        R[t] = TPIDRURO_NS;
if PSTATE.EL == EL0 then
    UNDEFINED;
elszf PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elszf EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
elszf HaveEL(EL3) && ELUsingAArch32(EL3) then
    TPIDRURO_NS = R[t];
else
    TPIDRURO = R[t];
elszf PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        TPIDRURO_NS = R[t];
else
    TPIDRURO = R[t];
elszf PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        TPIDRURO_S = R[t];
else
    TPIDRURO_NS = R[t];

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elszf PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elszf EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
elszf HaveEL(EL3) && ELUsingAArch32(EL3) then
    TPIDRURO_NS = R[t];
else
    TPIDRURO = R[t];
elszf PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        TPIDRURO_NS = R[t];
else
    TPIDRURO = R[t];
elszf PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        TPIDRURO_S = R[t];
else
    TPIDRURO_NS = R[t];
G8.2.163 TPIDRURW, PL0 Read/Write Software Thread ID Register

The TPIDRURW characteristics are:

**Purpose**

Provides a location where software executing at EL0 can store thread identifying information, for
OS management purposes.

The PE makes no use of this register.

**Configurations**

AArch32 System register TPIDRURW bits [31:0] are architecturally mapped to AArch64 System
register TPIDR_EL0[31:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to TPIDRURW
are UNDEFINED.

________ Note _________

The PE never updates this register.

**Attributes**

TPIDRURW is a 32-bit register.

**Field descriptions**

```
<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Bits [31:0]**

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

\* On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing TPIDRURW**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC\{<c>\}{<q>\}} \coproc, \{#<opc1>, <Rt>, <CRn>, <CRm>, \{#<opc2>\}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

\[
\text{if PSTATE.EL == EL0 then}
\text{if L2Enabled()} && \text{ELUsingAArch32}(\text{EL2}) && \text{HCR_EL2.<E2H,TGE>} != '11' \&\& \text{HSTR_EL2.T13} == '1' then
\text{AArch64.AArch32SystemAccessTrap(EL2, 0x03)};
\text{elsif L2Enabled()} && \text{ELUsingAArch32}(\text{EL2}) && \text{HSTR.T13} == '1' then
\text{AArch32.TakeHypTrapException(0x03)};
\text{elsif L2Enabled()} && \text{ELUsingAArch32}(\text{EL1}) && \text{HCR_EL2.<E2H,TGE>} != '11' \&\& \text{(!HaveEL(EL3) || SCR_EL3.FGTEn == '1')} \&\& \text{HFRTR_EL2.TPIDR_EL0} == '1' then
\text{AArch64.AArch32SystemAccessTrap(EL2, 0x03)};
\text{else}
\text{R[t] = TPIDRURW;}
\text{elsif PSTATE.EL == EL1 then}
\text{if L2Enabled()} && \text{ELUsingAArch32}(\text{EL2}) && \text{HSTR_EL2.T13} == '1' then
\text{AArch64.AArch32SystemAccessTrap(EL2, 0x03)};
\text{else}
\text{R[t] = TPIDRURW;}
\]
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.T13 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
  R[t] = TPIDURW_NS;
else
  R[t] = TPIDURW;
elsif PSTATE_EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    R[t] = TPIDURW_NS;
  else
    R[t] = TPIDURW;
elsif PSTATE_EL == EL3 then
  if SCR_NS == '0' then
    R[t] = TPIDURW_S;
  else
    R[t] = TPIDURW_NS;
else
  R[t] = TPIDURW_NS;

MCR<coproc>, <CRn>, <CRm>, <Rt>,<opc1>,<opc2>

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>
G8.2.164 TTBCR, Translation Table Base Control Register

The TTBCR characteristics are:

**Purpose**

The control register for stage 1 of the PL1&0 translation regime. Its controls include:

- Where the VA range is split between addresses translated using TTBR0 and addresses translated using TTBR1.
- The translation table format used by this stage of translation.

From Armv8.2, when the value of TTBCR.{EAE, T2E} is \{1, 1\}, TTBCR is used with TTBCR2.

**Configurations**

AArch32 System register TTBCR bits [31:0] are architecturally mapped to AArch64 System register TCR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TTBCR are UNDEFINED.

The current translation table format determines which format of the register is used.

Some RW fields of this register have defined reset values. These apply only if the PE resets into an Exception level that is using AArch32. If the PE resets into EL3 using AArch32 then:

- The EAE bit resets to 0 in both the Secure and the Non-secure instances of the register.
- Other reset values apply only to the Secure instance of the register.

**Attributes**

TTBCR is a 32-bit register.

**Field descriptions**

*When TTBCR.EAE == 0:*

![Field diagram]

**EAE, bit [31]**

Extended Address Enable.

0b0 Use the VMSAv8-32 translation system with the Short-descriptor translation table format.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

**Bits [30:6]**

Reserved, RES0.

**PD1, bit [5]**

Translation table walk disable for translations using TTBR1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1.

0b0 Perform translation table walks using TTBR1.

0b1 A TLB miss on an address that is translated using TTBR1 generates a Translation fault. No translation table walk is performed.
The reset behavior of this field is:
• On a Warm reset, this field resets to 0.

**PD0, bit [4]**

Translation table walk disable for translations using TTBR0. This bit controls whether a translation table walk is performed on a TLB miss for an address that is translated using TTBR0.

0b0  Perform translation table walks using TTBR0.
0b1  A TLB miss on an address that is translated using TTBR0 generates a Translation fault. No translation table walk is performed.

The reset behavior of this field is:
• On a Warm reset, this field resets to 0.

**Bit [3]**

Reserved, RES0.

**N, bits [2:0]**

Indicate the width of the base address held in TTBR0. In TTBR0, the base address field is bits[31:14-N]. The value of N also determines:
• Whether TTBR0 or TTBR1 is used as the base address for translation table walks.
• The size of the translation table pointed to by TTBR0.

N can take any value from 0 to 7, that is, from 0b000 to 0b111.
When N has its reset value of 0, the translation table base is compatible with Armv5 and Armv6.
The reset behavior of this field is:
• On a Warm reset, this field resets to 0.

*When TTBCR.EAE == 1:*

**EAE, bit [31]**

Extended Address Enable.

0b1  Use the VMSAv8-32 translation system with the Long-descriptor translation table format.

The reset behavior of this field is:
• On a Warm reset, this field resets to 0.

**IMPLEMENTATION DEFINED, bit [30]**

IMPLEMENTATION DEFINED.
The reset behavior of this field is:
• On a Warm reset, this field resets to 0.

**SH1, bits [29:28]**

Shareability attribute for memory associated with translation table walks using TTBR1.

0b00  Non-shareable.
0b10  Outer Shareable.
0b11  Inner Shareable.
Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**ORGN1, bits [27:26]**

Outer cacheability attribute for memory associated with translation table walks using TTBR1.
- 0b00 Normal memory, Outer Non-cacheable.
- 0b01 Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
- 0b10 Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
- 0b11 Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**IRGN1, bits [25:24]**

Inner cacheability attribute for memory associated with translation table walks using TTBR1.
- 0b00 Normal memory, Inner Non-cacheable.
- 0b01 Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
- 0b10 Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
- 0b11 Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**EPD1, bit [23]**

Translation table walk disable for translations using TTBR1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1.
- 0b0 Perform translation table walks using TTBR1.
- 0b1 A TLB miss on an address that is translated using TTBR1 generates a Translation fault. No translation table walk is performed.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**A1, bit [22]**

Selects whether TTBR0 or TTBR1 defines the ASID.
- 0b0 TTBR0.ASID defines the ASID.
- 0b1 TTBR1.ASID defines the ASID.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**Bits [21:19]**

Reserved, RES0.

**T1SZ, bits [18:16]**

See *Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format* on page G5-9181 for how TTBCR.T1SZ, T0SZ determine the input address ranges and memory region sizes translated using TTBR0 and TTBR1.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.
Bits [15:14]
Reserved, RES0.

SH0, bits [13:12]
Shareability attribute for memory associated with translation table walks using TTBR0.
- 0b00  Non-shareable
- 0b10  Outer Shareable
- 0b11  Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

ORGN0, bits [11:10]
Outer cacheability attribute for memory associated with translation table walks using TTBR0.
- 0b00  Normal memory, Outer Non-cacheable.
- 0b01  Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
- 0b10  Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
- 0b11  Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

IRGN0, bits [9:8]
Inner cacheability attribute for memory associated with translation table walks using TTBR0.
- 0b00  Normal memory, Inner Non-cacheable.
- 0b01  Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
- 0b10  Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
- 0b11  Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

EPD0, bit [7]
Translation table walk disable for translations using TTBR0. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR0.
- 0b0  Perform translation table walks using TTBR0.
- 0b1  A TLB miss on an address that is translated using TTBR0 generates a Translation fault.

No translation table walk is performed.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

T2E, bit [6]

When FEAT_AA32HPD is implemented:
TTBCR2 Enable.
- 0b0  TTBCR2 is disabled. The contents of TTBCR2 are treated as 0 for all purposes other than reading or writing the register.
- 0b1  TTBCR2 is enabled.

If TTBCR.EAE==0, then the behavior is as if the bit is 0.
Otherwise:

Reserved, RES0.

Bits [5:3]

Reserved, RES0.

**T0SZ, bits [2:0]**

See Selecting between TTBR0 and TTBR1, VMSAv8-32 Long-descriptor translation table format on page G5-9181 for how TTBCR, {T1SZ, T0SZ} determine the input address ranges and memory region sizes translated using TTBR0 and TTBR1.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

**Accessing TTBCR**

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC\{c\}\{q\} <coproc>, \{\#<opc1>, <Rt>, <CRn>, <CRm>\}, \{\#<opc2}\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    R[t] = TTBCR_NS;
else
  R[t] = TTBCR;
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    R[t] = TTBCR_NS;
else
  R[t] = TTBCR;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    R[t] = TTBCR_S;
else
  R[t] = TTBCR_NS;

**MCR\{c\}\{q\} <coproc>, \{\#<opc1>, <Rt>, <CRn>, <CRm>\}, \{\#<opc2}\)

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBCR_NS = R[t];
    else
        TTBCR = R[t];
    elsif PSTATE_EL == EL2 then
        if HaveEL(EL3) && ELUsingAArch32(EL3) then
            TTBCR_NS = R[t];
        else
            TTBCR = R[t];
        elsif PSTATE_EL == EL3 then
            if SCR_NS == '0' && CP15SDISABLE == HIGH then
                UNDEFINED;
            elsif SCR_NS == '0' && CP15SDISABLE2 == HIGH then
                UNDEFINED;
            else
                if SCR_NS == '0' then
                    TTBCR_S = R[t];
                else
                    TTBCR_NS = R[t];
                end
        end
    end
G8.2.165 TTBCR2, Translation Table Base Control Register 2

The TTBCR2 characteristics are:

**Purpose**

The second control register for stage 1 of the PL1&0 translation regime.

If FEAT_AA32HPD is not implemented then this register is not implemented and its encoding is UNDEFINED. Otherwise:

- When the value of TTBCR.{EAE, T2E} is not \{1, 1\} the contents of TTBCR2 are treated as zero for all purposes other than reading or writing the register.
- When the value of TTBCR.{EAE, T2E} is \{1, 1\} TTBCR2 is used with TTBCR.

**Configurations**

AArch32 System register TTBCR2 bits [31:0] are architecturally mapped to AArch64 System register TCR_EL1[63:32].

This register is present only when EL1 is capable of using AArch32 and FEAT_AA32HPD is implemented. Otherwise, direct accesses to TTBCR2 are UNDEFINED.

**Attributes**

TTBCR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:19]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>HWU162, bit [18]</td>
</tr>
</tbody>
</table>

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR1.

- For translations using TTBR1, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
- For translations using TTBR1, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD1 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD1 is 0 or the value of TTBCR.T2E is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.
HWU161, bit [17]

When **FEAT_HPDS2** is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR1.

0b0  For translations using TTBR1, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1  For translations using TTBR1, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD1 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD1 is 0 or the value of TTBCR.T2E is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU160, bit [16]

When **FEAT_HPDS2** is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR1.

0b0  For translations using TTBR1, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1  For translations using TTBR1, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD1 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD1 is 0 or the value of TTBCR.T2E is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU159, bit [15]

When **FEAT_HPDS2** is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR1.

0b0  For translations using TTBR1, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1  For translations using TTBR1, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD1 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD1 is 0 or the value of TTBCR.T2E is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
HWU062, bit [14]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR0.

0b0 For translations using TTBR0, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1 For translations using TTBR0, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD0 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD0 is 0 or the value of TTBCR.T2E is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU061, bit [13]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR0.

0b0 For translations using TTBR0, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1 For translations using TTBR0, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD0 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD0 is 0 or the value of TTBCR.T2E is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU060, bit [12]

When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR0.

0b0 For translations using TTBR0, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1 For translations using TTBR0, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD0 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD0 is 0 or the value of TTBCR.T2E is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
HWU059, bit [11]  
When FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR0.

0b0  For translations using TTBR0, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1  For translations using TTBR0, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TTBCR2.HPD0 is 1.

The Effective value of this field is 0 if the value of TTBCR2.HPD0 is 0 or the value of TTBCR.T2E is 0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD1, bit [10]  
Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, XNTable, and PXNTable, in the translation tables pointed to by TTBR1.

0b0  Hierarchical permissions are enabled.
0b1  Hierarchical permissions are disabled if TTBCR.T2E == 1.

When disabled, the permissions are treated as if the bits are 0.

The Effective value of this field is 0 if the value of TTBCR.T2E is 0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

HPD0, bit [9]  
Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, XNTable, and PXNTable, in the translation tables pointed to by TTBR0.

0b0  Hierarchical permissions are enabled.
0b1  Hierarchical permissions are disabled if TTBCR.T2E == 1.

When disabled, the permissions are treated as if the bits are 0.

The Effective value of this field is 0 if the value of TTBCR.T2E is 0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:0]  
Reserved, RES0.

Accessing TTBCR2

Accesses to this register use the following encodings in the System register encoding space:
MRC<\{c\}<q\}> <coproc>, (#<opc2>, <Rt>, <CRn>, <CRm>), (#<opc2>)

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
eltsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
eltsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x00);
eltsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
eltsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x00);
eltsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        R[t] = TTBCR2_NS;
eltselse
        R[t] = TTBCR2;
eltsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        R[t] = TTBCR2_NS;
eltselse
        R[t] = TTBCR2;
eltsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        R[t] = TTBCR2_NS;
eltselse
        R[t] = TTBCR2;
MCR<\{c\}<q\}> <coproc>, (#<opc2>, <Rt>, <CRn>, <CRm>), (#<opc2>)

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
eltsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
eltsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x00);
eltsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
eltsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x00);
eltsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBCR2_NS = R[t];
eltselse
        TTBCR2 = R[t];
eltsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBCR2_NS = R[t];
eltselse
        TTBCR2 = R[t];
eltsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE == HIGH then
        UNDEFINED;

ARM DDI 04871.a  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  G8-9801
ID081822  Non-Confidential
elsif SCR.NS == '0' && CP15DISABLE2 == HIGH then
    UNDEFINED;
else
    if SCR.NS == '0' then
        TTBCR2_S = R[t];
    else
        TTBCR2_NS = R[t];
G8.2.166   TTBR0, Translation Table Base Register 0

The TTBR0 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the lower VA range in the PL1&0 translation regime, and other information for this translation regime.

Configurations

AArch32 System register TTBR0 bits [63:0] are architecturally mapped to AArch64 System register TTBR0_EL1[63:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TTBR0 are UNDEFINED.

TTBR0 is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits [31:0] and do not modify bits [63:32]. TTBCR.EAE determines which TTBR0 format is used:

• TTBCR.EAE == 0b0: 32-bit format is used. TTBR0[63:32] are ignored.
• TTBCR.EAE == 0b1: 64-bit format is used.

When EL3 is using AArch32, write access to TTBR0(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

Used in conjunction with the TTBCR. When the 64-bit TTBR0 format is used, cacheability and shareability information is held in the TTBCR, not in TTBR0.

Attributes

TTBR0 is a 64-bit register.

Field descriptions

When TTBCR.EAE == 0:

Bits [63:32]

Reserved, RES0.

TTB0, bits [31:7]

Translation table base address, bits[31:x], where x is 14-(TTBCR.N). Register bits [x-1:7] are RES0, with the additional requirement that if these bits are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

• Register bits [x-1:7] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
• The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.
IRGN, bits [0, 6]

Inner region bits. Bits [0,6] of this register together indicate the Inner Cacheability attributes for the memory associated with the translation table walks. The possible values of IRGN[1:0] are:

- **0b00**: Normal memory, Inner Non-cacheable.
- **0b01**: Normal memory, Inner Write-Back Write-Allocate Cacheable.
- **0b10**: Normal memory, Inner Write-Through Cacheable.
- **0b11**: Normal memory, Inner Write-Back no Write-Allocate Cacheable.

--- Note ---

The encoding of the IRGN bits is counter-intuitive, with register bit[6] being IRGN[0] and register bit[0] being IRGN[1]. This encoding is chosen to give a consistent encoding of memory region types and to ensure that software written for ARMv7 without the Multiprocessing Extensions can run unmodified on an implementation that includes the functionality introduced by the ARMv7 Multiprocessing Extensions.

The IRGN field is split as follows:

- **IRGN[0]** is TTBR0[6].
- **IRGN[1]** is TTBR0[0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NOS, bit [5]

Not Outer Shareable. When the value of TTBR0.S is 1, indicates whether the memory associated with a translation table walk is Inner Shareable or Outer Shareable:

- **0b0**: Memory is Outer Shareable.
- **0b1**: Memory is Inner Shareable.

This bit is ignored when the value of TTBR0.S is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

RGN, bits [4:3]

Region bits. Indicates the Outer cacheability attributes for the memory associated with the translation table walks:

- **0b00**: Normal memory, Outer Non-cacheable.
- **0b01**: Normal memory, Outer Write-Back Write-Allocate Cacheable.
- **0b10**: Normal memory, Outer Write-Through Cacheable.
- **0b11**: Normal memory, Outer Write-Back no Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMP, bit [2]

The effect of this bit is IMPLEMENTATION DEFINED. If the translation table implementation does not include any IMPLEMENTATION DEFINED features this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S, bit [1]

Shareable. Indicates whether the memory associated with the translation table walks is Shareable:

- **0b0**: Memory is Non-shareable.
Memory is Shareable. The TTBR0.NOS field indicates whether the memory is Inner Shareable or Outer Shareable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

*When TTBCR.EAE == 1:*

Bits [63:56]

Reserved, RES0.

ASID, bits [55:48]

An ASID for the translation table base address. The TTBCR.A1 field selects either TTBR0.ASID or TTBR1.ASID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

BADDR, bits [47:1]

Translation table base address, bits[47:x]. Bits [x-1:1] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:3] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

x is determined from the value of TTBCR.T0SZ as follows:

- If TTBCR.T0SZ is 0 or 1, x = 5 - TTBCR.T0SZ.
- If TTBCR.T0SZ is greater than 1, x = 14 - TTBCR.T0SZ.

If bits[47:40] of the translation table base address are not zero, an Address size fault is generated.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CnP, bit [0]

*When FEAT_TTCNP is implemented:*

Common not Private. When TTBCR.EAE == 1, this bit indicates whether each entry that is pointed to by TTBR0 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0.CnP is 1.

0b0

The translation table entries pointed to by this instance of TTBR0, for the current ASID, are permitted to differ from corresponding entries for this instance of TTBR0 for other PEs in the Inner Shareable domain. This is not affected by:

- The value of TTBR0.CnP on those other PEs.
- The value of TTBCR.EAE on those other PEs.
- The value of the current ASID or, for the Non-secure instance of TTBR0, the value of the current VMID.
The translation table entries pointed to by this instance of TTBR0 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR0.CnP is 1 for this instance of TTBR0 and all of the following apply:

- The translation table entries are pointed to by this instance of TTBR0.
- The value of the applicable TTBCR.EAE field is 1.
- The ASID is the same as the current ASID.
- For the Non-secure instance of TTBR0, the VMID is the same as the current VMID.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

**Note**

If the value of the TTBR0.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR0s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see [CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values](#) on page K1-11567.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

### Accessing TTBR0

Accesses to this register use the following encodings in the System register encoding space:

$$MRC{<c>}{<q>}{<coproc>}, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}$$

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b010</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    R[t] = TTBR0_NS<31:0>;
  else
    R[t] = TTBR0<31:0>;
  elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
      R[t] = TTBR0_NS<31:0>;
    else
      R[t] = TTBR0<31:0>;
    elsif PSTATE.EL == EL3 then
      if SCR.NS == '0' then
        R[t] = TTBR0<31:0>;
      else
        R[t] = TTBR0<31:0>;
      end if
    end if
  else
    R[t] = TTBR0<31:0>;
  end if

```
else
    R[t] = TTBR0_NS<31:0>;

MCR{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

MRRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <Rt2>, <CRm>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL3) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && !ELUsingAArch32(EL3) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL3) && HCR.TVM == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL3) && HCR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL3) && HCR.T2 == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL3) then
        (R[t2], R[t]) = (TTBR0_NS<63:32>, TTBR0_NS<31:0>);
else
    (R[t2], R[t]) = (TTBR0<63:32>, TTBR0<31:0>);
elsif PSTATE.EL == EL2 then

MRRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <Rt2>, <CRm>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0010</td>
<td>0b0000</td>
</tr>
</tbody>
</table>
if HaveEL(EL3) & ELUsingAArch32(EL3) then
  (R[t2], R[t]) = (TTBR0_NS<63:32>, TTBR0_NS<31:0>);
else
  (R[t2], R[t]) = (TTBR0<63:32>, TTBR0<31:0>);
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    (R[t2], R[t]) = (TTBR0_S<63:32>, TTBR0_S<31:0>);
  else
    (R[t2], R[t]) = (TTBR0_NS<63:32>, TTBR0_NS<31:0>);
else
  (R[t2], R[t]) = (TTBR0_NS<63:32>, TTBR0_NS<31:0>);

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0010</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() & ELUsingAArch32(EL2) & HSTR_EL2.T2 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T2 == '1' then
    AArch32.TakeHypTrapException(0x04);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.TVM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR.TVM == '1' then
    AArch32.TakeHypTrapException(0x04);
  elsif HaveEL(EL3) & ELUsingAArch32(EL3) then
    TTBR0_NS = R[t2]:R[t];
  else
    TTBR0 = R[t2]:R[t];
else
  TTBR0 = R[t2]:R[t];
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) & ELUsingAArch32(EL3) then
    TTBR0_NS = R[t2]:R[t];
  else
    TTBR0 = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' & CP15SDISABLE == HIGH then
    UNDEFINED;
else
  if SCR.NS == '0' then
    TTBR0_S = R[t2]:R[t];
  else
    TTBR0_NS = R[t2]:R[t];
G8.2.167   TTBR1, Translation Table Base Register 1

The TTBR1 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the higher VA range in the PL1&0 translation regime, and other information for this translation regime.

Configurations

AArch32 System register TTBR1 bits [63:0] are architecturally mapped to AArch64 System register TTBR1_EL1[63:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to TTBR1 are UNDEFINED.

TTBR1 is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits [31:0] and do not modify bits [63:32].

TTBCR.EAE determines which TTBR1 format is used:

• TTBCR.EAE == 0b0: 32-bit format is used. TTBR1[63:32] are ignored.
• TTBCR.EAE == 0b1: 64-bit format is used.

Used in conjunction with the TTBCR. When the 64-bit TTBR1 format is used, cacheability and shareability information is held in the TTBCR, not in TTBR1.

Attributes

TTBR1 is a 64-bit register.

Field descriptions

When TTBCR.EAE == 0:

Bits [63:32]

Reserved, RES0.

TTB1, bits [31:7]

Translation table base address, bits[31:14]. Register bits [13:7] are RES0, with the additional requirement that if these bits are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

• Register bits [13:7] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
• The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.
IRGN, bits [6, 0]

Inner region bits. IRGN[1:0] indicate the Inner Cacheability attributes for the memory associated with the translation table walks. The possible values of IRGN[1:0] are:

- 0b00: Normal memory, Inner Non-cacheable.
- 0b01: Normal memory, Inner Write-Back Write-Allocate Cacheable.
- 0b10: Normal memory, Inner Write-Through Cacheable.
- 0b11: Normal memory, Inner Write-Back no Write-Allocate Cacheable.

Note

The encoding of the IRGN bits is counter-intuitive, with register bit[6] being IRGN[0] and register bit[0] being IRGN[1]. This encoding is chosen to give a consistent encoding of memory region types and to ensure that software written for Armv7 without the Multiprocessing Extensions can run unmodified on an implementation that includes the functionality introduced by the ARMv7 Multiprocessing Extensions.

The IRGN field is split as follows:
- IRGN[1] is TTBR1[6].
- IRGN[0] is TTBR1[0].

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NOS, bit [5]

Not Outer Shareable. When the value of TTBR1.S is 1, indicates whether the memory associated with a translation table walk is Inner Shareable or Outer Shareable:

- 0b0: Memory is Outer Shareable.
- 0b1: Memory is Inner Shareable.

This bit is ignored when the value of TTBR1.S is 0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

RGN, bits [4:3]

Region bits. Indicates the Outer cacheability attributes for the memory associated with the translation table walks:

- 0b00: Normal memory, Outer Non-cacheable.
- 0b01: Normal memory, Outer Write-Back Write-Allocate Cacheable.
- 0b10: Normal memory, Outer Write-Through Cacheable.
- 0b11: Normal memory, Outer Write-Back no Write-Allocate Cacheable.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMP, bit [2]

The effect of this bit is IMPLEMENTATION DEFINED. If the translation table implementation does not include any IMPLEMENTATION DEFINED features this bit is RES0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S, bit [1]

Shareable. Indicates whether the memory associated with the translation table walks is Shareable:

- 0b0: Memory is Non-shareable.
Memory is Shareable. The TTBR1.NOS field indicates whether the memory is Inner Shareable or Outer Shareable.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

*When TTBCR.EAE == 1:*

### Bits [63:56]

Reserved, RES0.

### ASID, bits [55:48]

An ASID for the translation table base address. The TTBCR.A1 field selects either TTBR0.ASID or TTBR1.ASID.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

### BADDR, bits [47:1]

Translation table base address, bits[47:x], Bits [x-1:1] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned translation table base address, with effects that are **CONSTRAINED UNPREDICTABLE**, and must be one of the following:
- Register bits [x-1:3] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

x is determined from the value of TTBCR.T1SZ as follows:
- If TTBCR.T1SZ is 0 or 1, x = 5 - TTBCR.T1SZ.
- If TTBCR.T1SZ is greater than 1, x = 14 - TTBCR.T1SZ.

If bits[47:40] of the translation table base address are not zero, an Address size fault is generated.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

### CnP, bit [0]

*When FEAT_TTCNP is implemented:*

Common not Private. When TTBCR.EAE ==1, this bit indicates whether each entry that is pointed to by TTBR1 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR1.CnP is 1.

0b0 The translation table entries pointed to by this instance of TTBR1, for the current ASID, are permitted to differ from corresponding entries for this instance of TTBR1 for other PEs in the Inner Shareable domain. This is not affected by:
- The value of TTBR1.CnP on those other PEs.
- The value of TTBCR.EAE on those other PEs.
- The value of the current ASID or, for the Non-secure instance of TTBR1, the value of the current VMID.
The translation table entries pointed to by this instance of TTBR1 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR1.CnP is 1 for this instance of TTBR1 and all of the following apply:

- The translation table entries are pointed to by this instance of TTBR1.
- The value of the applicable TTBCR.EAE field is 1.
- The ASID is the same as the current ASID.
- For the Non-secure instance of TTBR1, the VMID is the same as the current VMID.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

**Note**

If the value of the TTBR1.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR1s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONstrained UNPREDICTABLE, see [CONstrained UNPREDICTABLE behaviors due to caching of System register control or data values](#).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

### Accessing TTBR1

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC\{<c>\}{<q>}<coproc>, \{\#<opc1>, <Rt>, <CRn>, <CRm}\}, \{\#<opc2}\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() & !ELUsingAArch32(EL2) & HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) & ELUsingAArch32(EL3) then
        R[t] = TTBR1_NS<31:0>;
    else
        R[t] = TTBR1<31:0>;
    elsif PSTATE.EL == EL2 then
    elsif PSTATE.EL == EL3 then
        if SCR.NS == '0' then
            R[t] = TTBR1_S<31:0>;
        else
            R[t] = TTBR1<31:0>;
        elsif PSTATE.EL == EL3 then
            if SCR.NS == '0' then
                R[t] = TTBR1_S<31:0>;
            else
                R[t] = TTBR1<31:0>;
            end
        end
end

else  
R[t] = TTBR1_NS<31:0>;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBR1_NS = ZeroExtend(R[t]);
    else
        TTBR1 = ZeroExtend(R[t]);
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        TTBR2_NS = ZeroExtend(R[t]);
    else
        TTBR1 = ZeroExtend(R[t]);
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' && CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        if SCR.NS == '0' then
            TTBR1_S = ZeroExtend(R[t]);
        else
            TTBR1_NS = ZeroExtend(R[t]);

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0001</td>
<td></td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TRVM == '1' then
        AArch32.TakeHypTrapException(0x04);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        (R[t2], R[t]) = (TTBR1_NS<63:32>, TTBR1_NS<31:0>);
    else
        (R[t2], R[t]) = (TTBR1<63:32>, TTBR1<31:0>);
elsif PSTATE.EL == EL2 then
...
if HaveEL(EL3) && ELUsingAArch32(EL3) then
  (R[t2], R[t]) = (TTBR1_NS<63:32>, TTBR1_NS<31:0>);
else
  (R[t2], R[t]) = (TTBR1<63:32>, TTBR1<31:0>);
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    (R[t2], R[t]) = (TTBR1_S<63:32>, TTBR1_S<31:0>);
  else
    (R[t2], R[t]) = (TTBR1_NS<63:32>, TTBR1_NS<31:0>);
endif

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0010</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
    AArch32.TakeHypTrapException(0x04);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TVM == '1' then
    AArch32.TakeHypTrapException(0x04);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    TTBR1_NS = R[t2]:R[t];
  else
    TTBR1 = R[t2]:R[t];
endif
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    TTBR1_NS = R[t2]:R[t];
  else
    TTBR1 = R[t2]:R[t];
endif
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' && CP15DISABLE2 == HIGH then
    UNDEFINED;
  else
    if SCR.NS == '0' then
      TTBR1_S = R[t2]:R[t];
    else
      TTBR1_NS = R[t2]:R[t];
  endif
G8.2.168 VBAR, Vector Base Address Register

The VBAR characteristics are:

**Purpose**

When high exception vectors are not selected, holds the vector base address for exceptions that are not taken to Monitor mode or to Hyp mode.

Software must program VBAR(NS) with the required initial value as part of the PE boot sequence.

**Configurations**

AArch32 System register VBAR bits [31:0] are architecturally mapped to AArch64 System register VBAR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to VBAR are UNDEFINED.

**Attributes**

VBAR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:5]</th>
<th>Vector Base Address</th>
<th>RES0</th>
</tr>
</thead>
</table>

**Bits [31:5]**

Vector Base Address. Bits[31:5] of the base address of the exception vectors for exceptions taken to this Exception level. Bits[4:0] of an exception vector are the exception offset.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

**Bits [4:0]**

Reserved, RES0.

**Accessing VBAR**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>}, \text{(#}<opc1>, <Rt>, <CRn>, <CRm>, \text{(#}<opc2>)}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        R[t] = VBAR_NS;
    else
        R[t] = VBAR NS;
    endif
elsif PSTATE.EL == EL2 then
if HaveEL(EL3) && ELUsingAArch32(EL3) then
R[t] = VBAR_NS;
else
R[t] = VBAR;
elsif PSTATE.EL == EL3 then
if SCR.NS == '0' then
R[t] = VBAR_S;
else
R[t] = VBAR_NS;
MCR{<c>}{<q>}{<coproc>}, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
VBAR_NS = R[t];
else
VBAR = R[t];
elsif PSTATE.EL == EL2 then
if HaveEL(EL3) && ELUsingAArch32(EL3) then
VBAR_NS = R[t];
else
VBAR = R[t];
elsif PSTATE.EL == EL3 then
if SCR.NS == '0' && CP15SDISABLE == HIGH then
UNDEFINED;
elsif SCR.NS == '0' && CP15SDISABLE2 == HIGH then
UNDEFINED;
else
if SCR.NS == '0' then
VBAR_S = R[t];
else
VBAR_NS = R[t];
G8.2.169 VMPIDR, Virtualization Multiprocessor ID Register

The VMPIDR characteristics are:

**Purpose**

Holds the value of the Virtualization Multiprocessor ID. This is the value returned by Non-secure EL1 reads of MPIDR.

**Configurations**

AArch32 System register VMPIDR bits [31:0] are architecturally mapped to AArch64 System register VMPIDR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to VMPIDR are UNDEFINED.

If EL2 is not implemented but EL3 is implemented, this register takes the value of the MPIDR.

**Attributes**

VMPIDR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>M</td>
</tr>
<tr>
<td>30</td>
<td>U</td>
</tr>
<tr>
<td>29</td>
<td>RES0</td>
</tr>
<tr>
<td>28</td>
<td>MT</td>
</tr>
<tr>
<td>27</td>
<td>Aff2</td>
</tr>
<tr>
<td>26</td>
<td>Aff1</td>
</tr>
<tr>
<td>25</td>
<td>Aff0</td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**M, bit [31]**

Indicates whether this implementation includes the functionality introduced by the Armv7 Multiprocessing Extensions.

0b0 This implementation does not include the Armv7 Multiprocessing Extensions functionality.

0b1 This implementation includes the Armv7 Multiprocessing Extensions functionality.

Access to this field is RES1.

**U, bit [30]**

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

0b0 Processor is part of a multiprocessor system.

0b1 Processor is part of a uniprocessor system.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to the value in MPIDR.U.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**Bits [29:25]**

Reserved, RES0.

**MT, bit [24]**

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multithreading type approach. See the description of Aff0 for more information about affinity levels.

0b0 Performance of PEs at the lowest affinity level is largely independent.

0b1 Performance of PEs at the lowest affinity level is very interdependent.
The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to the value in MPIDR.MT.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**Aff2, bits [23:16]**

Affinity level 2. See the description of Aff0 for more information.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to the value in MPIDR.Aff2.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**Aff1, bits [15:8]**

Affinity level 1. See the description of Aff0 for more information.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to the value in MPIDR.Aff1.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**Aff0, bits [7:0]**

Affinity level 0. This is the affinity level that is most significant for determining PE behavior. Higher affinity levels are increasingly less significant in determining PE behavior. The assigned value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to the value in MPIDR.Aff0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**Accessing VMPIDR**

Accesses to this register use the following encodings in the System register encoding space:

```assembly
MRC{<c}<{q}> <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>), (#<opc2>)
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
    elsif PSTATE_EL == EL2 then
        R[t] = VMPIDR;
    elsif PSTATE_EL == EL3 then
        if HaveEL(EL2) then
            R[t] = MPIDR;
```

G8-9818  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  Non-Confidential
```plaintext
elif SCR.NS == '0' then
  UNDEFINED;
else
  R[t] = VMPIDR;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.EL2.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
  endif
elsif PSTATE.EL == EL2 then
  VMPIDR = R[t];
elsif PSTATE.EL == EL3 then
  if !HaveEL(EL2) then
  return;
  elsif SCR.NS == '0' then
    UNDEFINED;
  else
    VMPIDR = R[t];
  endif
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.EL2.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) then
    R[t] = VMPIDR_EL2<31:0>;
  elsif EL2Enabled() && ELUsingAArch32(EL2) then
    R[t] = VMPIDR;
  else
    R[t] = MPIDR;
  endif
elsif PSTATE.EL == EL2 then
  R[t] = MPIDR;
elsif PSTATE.EL == EL3 then
  R[t] = MPIDR;
```

G8.2.170 VPIDR, Virtualization Processor ID Register

The VPIDR characteristics are:

**Purpose**

Holds the value of the Virtualization Processor ID. This is the value returned by Non-secure EL1 reads of MIDR.

**Configurations**

AArch32 System register VPIDR bits [31:0] are architecturally mapped to AArch64 System register VPIDR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to VPIDR are UNDEFINED.

If EL2 is not implemented but EL3 is implemented, this register takes the value of the MIDR.

**Attributes**

VPIDR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>23</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementer</td>
<td>Variant</td>
<td></td>
<td></td>
<td></td>
<td>PartNum</td>
<td></td>
<td>Revision</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Implementer, bits [31:24]**

The Implementer code. This field must hold an implementer code that has been assigned by Arm.

Assigned codes include the following:

- **0x00** Reserved for software use.
- **0x41** Arm Limited.
- **0x42** Broadcom Corporation.
- **0x43** Cavium Inc.
- **0x44** Digital Equipment Corporation.
- **0x46** Fujitsu Ltd.
- **0x49** Infineon Technologies AG.
- **0x40** Motorola or Freescale Semiconductor Inc.
- **0x4E** NVIDIA Corporation.
- **0x50** Applied Micro Circuits Corporation.
- **0x51** Qualcomm Inc.
- **0x56** Marvell International Ltd.
- **0x69** Intel Corporation.
- **0xC0** Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to the value in MIDR.Implementer.
  - Otherwise, this field resets to an architecturally UNKNOWN value.
Variant, bits [23:20]

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to the value in MIDR.Variant.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Architecture, bits [19:16]

Architecture version. Defined values are:

- 0b0001  Armv4.
- 0b0010  Armv4T.
- 0b0011  Armv5 (obsolete).
- 0b0100  Armv5T.
- 0b0101  Armv5TE.
- 0b0110  Armv5TEJ.
- 0b0111  Armv6.
- 0b1111  Architectural features are individually identified in the ID_* registers.

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to the value in MIDR.Architecture.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

PartNum, bits [15:4]

An IMPLEMENTATION DEFINED primary part number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to the value in MIDR.PartNum.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Revision, bits [3:0]

An IMPLEMENTATION DEFINED revision number for the device.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to the value in MIDR.Revision.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing VPIDR

Accesses to this register use the following encodings in the System register encoding space:
### MRC{<c>}{<q>}, {#}opc1, <Rt>, <CRn>, <CRm>{, {#}opc2}:

If PSTATE.EL = EL0 then
UNDEFINED;
else if PSTATE.EL = EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
else
  UNDEFINED;
else if PSTATE.EL = EL2 then
  R[t] = VPIDR;
else if PSTATE.EL = EL3 then
  if !HaveEL(EL2) then
    return;
  elsif SCR.NS == '0' then
    UNDEFINED;
  else
    VPIDR = R[t];
else
  VPIDR = R[t];

### MCR{<c>}{<q>}, {#}opc1, <Rt>, <CRn>, <CRm>{, {#}opc2}:

If PSTATE.EL = EL0 then
UNDEFINED;
else if PSTATE.EL = EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
else
  UNDEFINED;
else if PSTATE.EL = EL2 then
  VPIDR = R[t];
else if PSTATE.EL = EL3 then
  if !HaveEL(EL2) then
    return;
  elsif SCR.NS == '0' then
    UNDEFINED;
  else
    VPIDR = R[t];
MRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>opcodes</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch325SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T0 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif EL2Enabled() && !ELUsingAArch32(EL2) then
        R[t] = VPIDR_EL2<31:0>;
    elsif EL2Enabled() && ELUsingAArch32(EL2) then
        R[t] = VPIDR;
    else
        R[t] = MIDR;
    elsif PSTATE.EL == EL2 then
        R[t] = MIDR;
else
    if PSTATE.EL == EL3 then
        R[t] = MIDR;
G8.2.171 VTCR, Virtualization Translation Control Register

The VTCR characteristics are:

**Purpose**

The control register for stage 2 of the Non-secure PL1&0 translation regime.

--- Note ---

This stage of translation always uses the Long-descriptor translation table format.

**Configurations**

AArch32 System register VTCR bits [31:0] are architecturally mapped to AArch64 System register VTCR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to VTCR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

VTCR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES1.</td>
</tr>
<tr>
<td>30-29</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**HWU62, bit [28]**

*When FEAT_HPDS2 is implemented:*

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 2 translation table Block or Page entry.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit[62] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.</td>
</tr>
<tr>
<td>1</td>
<td>Bit[62] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:

* On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.
HWU61, bit [27]

When `FEAT_HPDS2` is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 2 translation table Block or Page entry.

0b0  Bit[61] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1  Bit[61] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU60, bit [26]

When `FEAT_HPDS2` is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 2 translation table Block or Page entry.

0b0  Bit[60] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1  Bit[60] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU59, bit [25]

When `FEAT_HPDS2` is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 2 translation table Block or Page entry.

0b0  Bit[59] of each stage 2 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

0b1  Bit[59] of each stage 2 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [24:14]

Reserved, RES0.

SH0, bits [13:12]

Shareability attribute for memory associated with translation table walks using VTTBR.

0b00  Non-shareable.

0b10  Outer Shareable.

0b11  Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ORGN0, bits [11:10]**
Outer cacheability attribute for memory associated with translation table walks using VTTBR.
- 0b00: Normal memory, Outer Non-cacheable.
- 0b01: Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
- 0b10: Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
- 0b11: Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IRGN0, bits [9:8]**
Inner cacheability attribute for memory associated with translation table walks using VTTBR.
- 0b00: Normal memory, Inner Non-cacheable.
- 0b01: Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
- 0b10: Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
- 0b11: Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**SL0, bits [7:6]**
Starting level for translation table walks using VTTBR.
- 0b00: Start at level 2
- 0b01: Start at level 1

All other values are reserved. If this field is programmed to a reserved value, or to a value that is not consistent with the programming of T0SZ, then a stage 2 level 1 Translation fault is generated.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [5]**
Reserved, RES0.

**S, bit [4]**
Sign extension bit. This bit must be programmed to the value of T0SZ[3]. If it is not, then the stage 2 T0SZ value is treated as an UNKNOWN value.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**T0SZ, bits [3:0]**
The size offset of the memory region addressed by VTTBR. The region size is $2^{(32-\text{T0SZ})}$ bytes.
This field holds a four-bit signed integer value, meaning it supports values from -8 to 7.

--- Note ---

This is different from the other translation control registers, where TnSZ holds a three-bit unsigned integer, supporting values from 0 to 7.

---

If this field is programmed to a value that is not consistent with the programming of SL0 then a stage 2 level 1 Translation fault is generated.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally \textsc{unknown} value.

### Accessing VTCR

Accesses to this register use the following encodings in the System register encoding space:

\begin{verbatim}
MRC\{<c>\}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
\end{verbatim}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0010</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

\begin{verbatim}
MCR\{<c>\}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
\end{verbatim}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0010</td>
<td>0b0001</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if \text{PSTATE.EL} == \text{EL0} then
\text{UNDEFINED};
else if \text{PSTATE.EL} == \text{EL1} then
  if \text{EL2Enabled()} \&\& \text{!ELUsingAArch32(EL2)} \&\& \text{HSTR_EL2.T2 == '1'} then
    \text{AArch64.AArch32SystemAccessTrap(EL2, 0x03)};
else if \text{EL2Enabled()} \&\& \text{ELUsingAArch32(EL2)} \&\& \text{HSTR.T2 == '1'} then
  \text{AArch32.TakeHypTrapException(0x03)};
else
  \text{UNDEFINED};
else if \text{PSTATE.EL} == \text{EL2} then
  \text{R[t] = VTCR};
else if \text{PSTATE.EL} == \text{EL3} then
  if \text{SCR.NS == '0'} then
    \text{UNDEFINED};
else
  \text{VTCR = R[t]};

if \text{PSTATE.EL} == \text{EL0} then
\text{UNDEFINED};
else if \text{PSTATE.EL} == \text{EL1} then
  if \text{EL2Enabled()} \&\& \text{ELUsingAArch32(EL2)} \&\& \text{HSTR_EL2.T2 == '1'} then
    \text{AArch64.AArch32SystemAccessTrap(EL2, 0x03)};
else if \text{EL2Enabled()} \&\& \text{ELUsingAArch32(EL2)} \&\& \text{HSTR.T2 == '1'} then
  \text{AArch32.TakeHypTrapException(0x03)};
else
  \text{UNDEFINED};
else if \text{PSTATE.EL} == \text{EL2} then
  \text{VTCR = R[t]};
else if \text{PSTATE.EL} == \text{EL3} then
  if \text{SCR.NS == '0'} then
    \text{UNDEFINED};
else
  \text{VTCR = R[t]};
G8.2.172 VTTBR, Virtualization Translation Table Base Register

The VTTBR characteristics are:

**Purpose**

Holds the base address of the translation table for the initial lookup for stage 2 of an address translation in the Non-secure PL1&0 translation regime, and other information for this translation regime.

**Configurations**

AArch32 System register VTTBR bits [63:0] are architecturally mapped to AArch64 System register VTTBR_EL2[63:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to VTTBR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

VTTBR is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-56</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>55-48</td>
<td>VMID</td>
</tr>
<tr>
<td>47-32</td>
<td>BADDR</td>
</tr>
<tr>
<td>31-1</td>
<td>BADDR</td>
</tr>
<tr>
<td>0</td>
<td>CnP</td>
</tr>
</tbody>
</table>

**Bits [63:56]**

Reserved, RES0.

**VMID, bits [55:48]**

The VMID for the translation table.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**BADDR, bits [47:1]**

Translation table base address, bits[47:x]. Bits [x-1:1] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Register bits [x-1:3] are treated as if all the bits are zero. The value read back from these bits is either the value written or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

x is determined from the value of VTCR.SL0 and VTCR.T0SZ as follows:

- If VTCR.SL0 is @b00, meaning that lookup starts at level 2, then x is 14 - VTCR.T0SZ.
- If VTCR.SL0 is @b01, meaning that lookup starts at level 1, then x is 5 - VTCR.T0SZ.
- If VTCR.SL0 is either @b10 or @b11 then a stage 2 level 1 Translation fault is generated.

If bits[47:40] of the translation table base address are not zero, an Address size fault is generated.
The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to an architecturally UNKNOWN value.

**CnP, bit [0]**

**When FEAT_TTCNP is implemented:**

Common not Private. This bit indicates whether each entry that is pointed to by VTTBR is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of VTTBR.CnP is 1.

- 0b0: The translation table entries pointed to by VTTBR are permitted to differ from the entries for VTTBR for other PEs in the Inner Shareable domain. This is not affected by the value of the current VMID.
- 0b1: The translation table entries pointed to by VTTBR are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of VTTBR.CnP is 1 and the VMID is the same as the current VMID.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

---

**Note**

If the value of the VTTBR.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those VTTBRs do not point to the same translation table entries when the VMID value is the same as the current VMID, then the results of translations are CONSTRAINED UNPREDICTABLE, see **CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values** on page K1-11567.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Accessing VTTBR**

Accesses to this register use the following encodings in the System register encoding space:

```
MRRC(<c>){<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0010</td>
<td>0b0110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x04);
    else
        UNDEFINED;
    end if;
elsif PSTATE.EL == EL2 then
    (R[t2], R[t]) = (VTTBR<63:32>, VTTBR<31:0>);
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    end if;
```
else
    (R[t2], R[t]) = (VTTBR[63:32], VTTBR[31:0]);

\textbf{MCRR\{<c>\}{<q>}} \{<coproc>, \{#<opc1>, <Rt>, <Rt2>, <CRm>\}

\begin{tabular}{|c|c|c|}
\hline
\textbf{coproc} & \textbf{CRm} & \textbf{opc1} \\
\hline
0b1111 & 0b0010 & 0b0110 \\
\hline
\end{tabular}

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() \&\& ELUsingAArch32(EL2) \&\& HSTR.EL2.T2 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() \&\& ELUsingAArch32(EL2) \&\& HSTR.T2 == '1' then
        AArch32.TakeHypTrapException(0x04);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VTTBR = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        VTTBR = R[t2]:R[t];
G8.3 Debug registers

This section lists the Debug System registers in AArch32 state, in alphabetic order.
### G8.3.1 DBGAUTHSTATUS, Debug Authentication Status register

The DBGAUTHSTATUS characteristics are:

**Purpose**

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

**Configurations**

AArch32 System register DBGAUTHSTATUS bits [31:0] are architecturally mapped to AArch64 System register DBGAUTHSTATUS_EL1[31:0].

AArch32 System register DBGAUTHSTATUS bits [31:0] are architecturally mapped to External register DBGAUTHSTATUS_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGAUTHSTATUS are UNDEFINED.

This register is required in all implementations.

**Attributes**

DBGAUTHSTATUS is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0</td>
</tr>
<tr>
<td>30-8</td>
<td>SNID</td>
</tr>
<tr>
<td>7-6</td>
<td>SID</td>
</tr>
<tr>
<td>5-4</td>
<td>NSID</td>
</tr>
<tr>
<td>3</td>
<td>NSNID</td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**SNID, bits [7:6]**

*When FEAT_Debugv8p4 is implemented:*

Secure Non-Invasive Debug.

This field has the same value as DBGAUTHSTATUS.SID.

*Otherwise:*

Secure Non-Invasive Debug.

- **0b00** Not implemented. EL3 is not implemented and the Effective value of SCR.NS is 1.
- **0b10** Implemented and disabled. ExternalSecureNoninvasiveDebugEnabled() == FALSE.
- **0b11** Implemented and enabled. ExternalSecureNoninvasiveDebugEnabled() == TRUE.

All other values are reserved.

**SID, bits [5:4]**

Secure Invasive Debug.

- **0b00** Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 1.
- **0b10** Implemented and disabled. ExternalSecureInvasiveDebugEnabled() == FALSE.
- **0b11** Implemented and enabled. ExternalSecureInvasiveDebugEnabled() == TRUE.

All other values are reserved.
NSID, bits [3:2]

*When FEAT_Debugv8p4 is implemented:*

- **0b00**: Not implemented. EL3 is not implemented and the Effective value of SCR.NS is 0.
- **0b11**: Implemented and enabled. EL3 is implemented or the Effective value of SCR.NS is 1.

All other values are reserved.

*Otherwise:*

- **0b00**: Not implemented. EL3 is not implemented and the Effective value of SCR.NS is 0.
- **0b10**: Implemented and disabled. ExternalNoninvasiveDebugEnabled() == FALSE.
- **0b11**: Implemented and enabled. ExternalNoninvasiveDebugEnabled() == TRUE.

All other values are reserved.

NSID, bits [1:0]

- **0b00**: Not implemented. EL3 is not implemented or the Effective value of SCR_EL3.NS is 0.
- **0b10**: Implemented and disabled. ExternalInvasiveDebugEnabled() == FALSE.
- **0b11**: Implemented and enabled. ExternalInvasiveDebugEnabled() == TRUE.

All other values are reserved.

**Accessing DBGAUTHSTATUS**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC\{<c>\}{<q> \ coproc, \ (#<opc1>, \ <Rt>, \ <CRn>, \ <CRm>{, \ (#<opc2>))
}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1110</td>
<td>0b110</td>
</tr>
</tbody>
</table>

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDA != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        R[t] = DBGAUTHSTATUS;
    end
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elsif EL2Enabled() && ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        end
    else
        R[t] = DBGAUTHSTATUS;
    end
```

"AArch32 System Register Descriptions
G8.3 Debug registers"
AArch64.AArch32SystemAccessTrap(EL3, 0x05);
else
  R[t] = DBGAUTHSTATUS;
elsif PSTATE.EL == EL3 then
  R[t] = DBGAUTHSTATUS;
G8.3.2 **DBGBCR<n>, Debug Breakpoint Control Registers, n = 0 - 15**

The DBGBCR<n> characteristics are:

**Purpose**

Holds control information for a breakpoint. Forms breakpoint n together with value register DBGBVR<n>. If EL2 is implemented and this breakpoint supports Context matching, DBGBVR<n> can be associated with a Breakpoint Extended Value Register DBGBXVR<n> for VMID matching.

**Configurations**

AArch32 System register DBGBCR<n> bits [31:0] are architecturally mapped to AArch64 System register DBGBCR<n>_EL1[31:0].

AArch32 System register DBGBCR<n> bits [31:0] are architecturally mapped to External register DBGBCR<n>_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGBCR<n> are UNDEFINED.

If breakpoint n is not implemented then accesses to this register are UNDEFINED.

**Attributes**

DBGBCR<n> is a 32-bit register.

**Field descriptions**

When the E field is zero, all the other fields in the register are ignored.

**Bits [31:24]**

Reserved, RES0.

**BT, bits [23:20]**

Breakpoint Type. Possible values are:

- **0b0000**: Unlinked instruction address match. DBGBVR<n> is the address of an instruction.
- **0b0001**: As 0b0000, but linked to a Context matching breakpoint.
- **0b0010**: Unlinked Context ID match. When FEAT_VHE is implemented, EL2 is using AArch64, and the Effective value of HCR_EL2.E2H is 1, if either the PE is executing at EL0 with HCR_EL2.TGE set to 1 or the PE is executing at EL2, then DBGBCR<n>.ContextID must match the CONTEXTIDR_EL2 value. Otherwise, DBGBCR<n>.ContextID must match the CONTEXTIDR value.
- **0b0011**: As 0b0010 with linking enabled.
- **0b0100**: Unlinked instruction address mismatch. DBGBVR<n> is the address of an instruction to be stepped.
- **0b0101**: As 0b0100, but linked to a Context matching breakpoint.
- **0b0110**: Unlinked CONTEXTIDR_EL1 match. DBGBVR<n>.ContextID is a Context ID compared against CONTEXTIDR.
- **0b0111**: As 0b0110 with linking enabled.
- **0b1000**: Unlinked VMID match. DBGBXVR<n>.VMID is a VMID compared against VTTBR.VMID.
- **0b1001**: As 0b1000 with linking enabled.
### LBN, bits [19:16]

Linked breakpoint number. For Linked address matching breakpoints, this specifies the index of the Context-matching breakpoint linked to.

For all other breakpoint types this field is ignored and reads of the register return an UNKNOWN value.

This field is ignored when the value of DBGBCR<\text-prev>.E is 0.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

### SSC, bits [15:14]

Security state control. Determines the Security states under which a Breakpoint debug event for breakpoint \text-prev is generated. This field must be interpreted along with the HMC and PMC fields, and there are constraints on the permitted values of the \{HMC, SSC, PMC\} fields.

For more information, see *Execution conditions for which a breakpoint generates Breakpoint exceptions on page G2-9063* and *Reserved DBGBCR<\text-prev>.\{SSC, HMC, PMC\} values on page G2-9075*.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

### HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Breakpoint debug event for breakpoint \text-prev is generated. This field must be interpreted along with the SSC and PMC fields, and there are constraints on the permitted values of the \{HMC, SSC, PMC\} fields. For more information see the SSC, bits [15:14] description.

For more information on the operation of the SSC, HMC, and PMC fields, see *Execution conditions for which a breakpoint generates Breakpoint exceptions on page G2-9063*.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

### Bits [12:9]

Reserved, RES0.

### BAS, bits [8:5]

Byte address select. Defines which half-words an address-matching breakpoint matches, regardless of the instruction set and Execution state.

The permitted values depend on the breakpoint type.
For Address match breakpoints, the permitted values are:

<table>
<thead>
<tr>
<th>BAS</th>
<th>Match instruction at</th>
<th>Constraint for debuggers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0011</td>
<td>DBGBVR&lt;n&gt;</td>
<td>Use for T32 instructions</td>
</tr>
<tr>
<td>0b1100</td>
<td>DBGBVR&lt;n&gt;+2</td>
<td>Use for T32 instructions</td>
</tr>
<tr>
<td>0b1111</td>
<td>DBGBVR&lt;n&gt;</td>
<td>Use for A32 instructions</td>
</tr>
</tbody>
</table>

All other values are reserved. For more information, see Reserved DBGCR<n>.BAS values on page G2-9075.

For more information on using the BAS field in Address Match breakpoints, see Using the BAS field in Address Match breakpoints on page G2-9067.

For Address mismatch breakpoints in an AArch32 stage 1 translation regime, the permitted values are:

<table>
<thead>
<tr>
<th>BAS</th>
<th>Step instruction at</th>
<th>Constraint for debuggers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>-</td>
<td>Use for a match anywhere breakpoint</td>
</tr>
<tr>
<td>0b0011</td>
<td>DBGBVR&lt;n&gt;</td>
<td>Use for T32 instructions</td>
</tr>
<tr>
<td>0b1100</td>
<td>DBGBVR&lt;n&gt;+2</td>
<td>Use for T32 instructions</td>
</tr>
<tr>
<td>0b1111</td>
<td>DBGBVR&lt;n&gt;</td>
<td>Use for A32 instructions</td>
</tr>
</tbody>
</table>

All other values are reserved. For more information, see Reserved DBGCR<n>.BAS values on page G2-9075.

For more information on using the BAS field in address mismatch breakpoints, see Using the BAS field in Address Match breakpoints on page G2-9067.

For Context matching breakpoints, this field is RES1 and ignored.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Bits [4:3]**

Reserved, RES0.

**PMC, bits [2:1]**

Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the DBGCR<n>.SSC description.

For more information on the operation of the SSC, HMC, and PMC fields, see Execution conditions for which a breakpoint generates Breakpoint exceptions on page G2-9063.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**E, bit [0]**

Enable breakpoint DBGBVR<n>. Possible values are:

- 0b0 Breakpoint disabled.
- 0b1 Breakpoint enabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
Accessing DBGBCR<n>

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC\{}<c>\{<q}\} <\text{coproc}, \{\#<\text{opc1}\}, <\text{Rt}, <\text{CRn}, <\text{CRm}\}\{, \{\#<\text{opc2}\}\}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b0000</td>
<td>0b0000</td>
<td>m[3:0]</td>
<td>0b101</td>
</tr>
</tbody>
</table>

\[
\text{integer } m = \text{UInt(CRm<3:0>);}
\]

\[
\begin{align*}
\text{if } m &= \text{NUM\_BREAKPOINTS} \text{ then } \\
\text{UNDEFINED;}
\text{elsif } \text{PSTATE.EL} &= \text{EL0} \text{ then } \\
\text{UNDEFINED;}
\text{elsif } \text{PSTATE.EL} &= \text{EL1} \text{ then } \\
\text{if } \text{Halted()} \text{ && HaveEL(EL3)} \text{ && EDSCR.SDD} &= \text{’1’} \text{ && boolean IMPLEMENTATION\_DEFINED “EL3 trap priority when } \\
\text{SDD} &= \text{’1’} \text{” \&& MDCR\_EL3.TDA} &= \text{’1’} \text{ then } \\
\text{UNDEFINED;}
\text{elsif } \text{EL2\_Enabled()} \text{ && !ELUsingAArch32(EL2)} \text{ && MDCR\_EL2.\text{TDE, TDA}} &= \text{’00’} \text{ then } \\
\text{AArch64.AArch32SystemAccessTrap(EL2, 0x05);}
\text{elsif } \text{EL2\_Enabled()} \text{ && ELUsingAArch32(EL2)} \text{ && HDCR.\text{TDE, TDA}} &= \text{’00’} \text{ then } \\
\text{AArch32.\text{TakeHypTrapException}(0x05);}
\text{elsif } \text{HaveEL(EL3)} \text{ && !ELUsingAArch32(EL3)} \text{ && MDCR\_EL3.TDA} &= \text{’1’} \text{ then } \\
\text{if } \text{Halted()} \text{ && EDSCR.SDD} &= \text{’1’} \text{ then } \\
\text{UNDEFINED;}
\text{else} \\
\text{AArch64.AArch32SystemAccessTrap(EL3, 0x05);}
\text{ elsif } \text{DBGOSLSR.\text{OSLK}} &= \text{’0’} \text{ && HaltingAllowed()} \text{ && EDSCR.TDA} &= \text{’1’} \text{ then } \\
\text{Halt(Debug\_Halt\_SoftwareAccess);}
\text{else} \\
R[t] &= \text{DBGBCR}[m];
\text{elsif } \text{PSTATE.EL} &= \text{EL2} \text{ then } \\
\text{if } \text{Halted()} \text{ && HaveEL(EL3)} \text{ && EDSCR.SDD} &= \text{’1’} \text{ && boolean IMPLEMENTATION\_DEFINED “EL3 trap priority when } \\
\text{SDD} &= \text{’1’} \text{” \&& MDCR\_EL3.TDA} &= \text{’1’} \text{ then } \\
\text{UNDEFINED;}
\text{elsif } \text{HaveEL(EL3)} \text{ && !ELUsingAArch32(EL3)} \text{ && MDCR\_EL3.TDA} &= \text{’1’} \text{ then } \\
\text{if } \text{Halted()} \text{ && EDSCR.SDD} &= \text{’1’} \text{ then } \\
\text{UNDEFINED;}
\text{else} \\
\text{AArch64.AArch32SystemAccessTrap(EL3, 0x05);}
\text{ elsif } \text{DBGOSLSR.\text{OSLK}} &= \text{’0’} \text{ && HaltingAllowed()} \text{ && EDSCR.TDA} &= \text{’1’} \text{ then } \\
\text{Halt(Debug\_Halt\_SoftwareAccess);}
\text{else} \\
R[t] &= \text{DBGBCR}[m];
\text{elsif } \text{PSTATE.EL} &= \text{EL3} \text{ then } \\
\text{if } \text{DBGOSLSR.\text{OSLK}} &= \text{’0’} \text{ && HaltingAllowed()} \text{ && EDSCR.TDA} &= \text{’1’} \text{ then } \\
\text{Halt(Debug\_Halt\_SoftwareAccess);}
\text{else} \\
R[t] &= \text{DBGBCR}[m];
\end{align*}
\]

\[
\text{MCR\{}<c>\{<q}\} <\text{coproc}, \{\#<\text{opc1}\}, <\text{Rt}, <\text{CRn}, <\text{CRm}\}\{, \{\#<\text{opc2}\}\}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b0000</td>
<td>0b0000</td>
<td>m[3:0]</td>
<td>0b101</td>
</tr>
</tbody>
</table>

\[
\text{integer } m = \text{UInt(CRm<3:0>);}
\]
if m >= NUM_BREAKPOINTS then
  UNDEFINED;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
    AArch32.TakeHypTrapException(0x05);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    elsif DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
      Halt(DebugHalt_SoftwareAccess);
    else
      DBGBCR[m] = R[t];
  else
    DBGBCR[m] = R[t];
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
  elsif DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
  else
    DBGBCR[m] = R[t];
elsif PSTATE_EL == EL3 then
  if DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
  else
    DBGBCR[m] = R[t];
G8.3.3 DBGBVR<n>, Debug Breakpoint Value Registers, n = 0 - 15

The DBGBVR<n> characteristics are:

**Purpose**

Holds a value for use in breakpoint matching, either the virtual address of an instruction or a context ID. Forms breakpoint n together with control register DBGCR<n>. If EL2 is implemented and this breakpoint supports Context matching, DBGBVR<n> can be associated with a Breakpoint Extended Value Register DBGBXVR<n> for VMID matching.

**Configurations**

AArch32 System register DBGBVR<n> bits [31:0] are architecturally mapped to AArch64 System register DBGBVR<n>_EL1[31:0].

AArch32 System register DBGBVR<n> bits [31:0] are architecturally mapped to External register DBGBVR<n>_EL1[31:0].

**Note**

Writes to DBGBVR<n> do not modify DBGBVR<n>_EL1[63:32].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGBVR<n> are UNDEFINED.

How this register is interpreted depends on the value of DBGCR<n>.BT.

- When DBGCR<n>.BT is 0b00x, this register holds a virtual address.
- When DBGCR<n>.BT is 0bxx1x, this register holds a Context ID.

For other values of DBGCR<n>.BT, this register is RES0.

Some breakpoints might not support Context ID comparison. For more information, see the description of the DBGIDR.CTX_CMPs field.

If breakpoint n is not implemented then accesses to this register are UNDEFINED.

**Attributes**

DBGBVR<n> is a 32-bit register.

**Field descriptions**

*When DBGCR<n>.BT == 0b000x:

<table>
<thead>
<tr>
<th>31</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA[31:2]</td>
<td></td>
<td></td>
<td>RES0</td>
</tr>
</tbody>
</table>

VA[31:2], bits [31:2]

Bits[31:2] of the address value for comparison.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.
When DBGBCR<\textit{n}>,\textit{BT} == 0b001\textit{x}:

ContextID, bits [31:0]
Context ID value for comparison.
The value is compared against CONTEXTIDR_EL2 when all of the following are true:
- FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented.
- HCR_EL2.{E2H, TGE} is \{1,1\}.
- The PE is executing at EL0.
- EL2 is using AArch64 and is enabled in the current Security state.
Otherwise, the value is compared against CONTEXTIDR.
The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<\textit{n}>,\textit{BT} == 0b101\textit{x} and EL2 is implemented:

ContextID, bits [31:0]
Context ID value for comparison against CONTEXTIDR.
The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<\textit{n}>,\textit{BT} == 0bx11x, EL2 is implemented and (FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented):

ContextID, bits [31:0]
Context ID value for comparison against CONTEXTIDR.
The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGBVR<\textit{n}>
Accesses to this register use the following encodings in the System register encoding space:
MRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b000</td>
<td>m[3:0]</td>
<td>0b100</td>
</tr>
</tbody>
</table>

integer m = UInt(CRm<3:0>);

if m >= NUM_BREAKPOINTS then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() & !ELUsingAArch32(EL2) & MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
  elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch32.TakeHypTrapException(0x05);
    end if
  end if
else
  R[t] = DBGVR[m];
end if

elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    end if
  else
    if DBGOSLSR.OSLK == '0' & HaltingAllowed() & EDSCR.TDA == '1' then
      Halt(DebugHalt_SoftwareAccess);
    else
      R[t] = DBGVR[m];
    end if
  end if
else
  R[t] = DBGVR[m];
end if

MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b000</td>
<td>m[3:0]</td>
<td>0b100</td>
</tr>
</tbody>
</table>

integer m = UInt(CRn<3:0>);

if m >= NUM_BREAKPOINTS then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if DBGOSLSR.OSLK == '0' & HaltingAllowed() & EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
  else
    R[t] = DBGVR[m];
  end if
else
  R[t] = DBGVR[m];
end if
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" & & !ELUsingAArch32(EL3) & & MDCR_EL3.TDA == '1' then
    UNDEFINED;
elsif EL2Enabled() & & !ELUsingAArch32(EL2) & & MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif EL2Enabled() & & ELUsingAArch32(EL2) & & HDCR.<TDE,TDA> != '00' then
    AArch32.TakeHypTrapException(0x05);
elsif HaveEL(EL3) & & !ELUsingAArch32(EL3) & & MDCR_EL3.TDA == '1' then
    if Halted() & & EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    end
elsif DBGOSLSR.OSLK == '0' & & HaltingAllowed() & & EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
else
    DBGBVR[m] = R[t];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    endif
elsif DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
else
    DBGBVR[m] = R[t];
elsif PSTATE.EL == EL3 then
    if DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
else
    DBGBVR[m] = R[t];
end
G8.3.4 DBGBXVR\(<n>\), Debug Breakpoint Extended Value Registers, \(n = 0 - 15\)

The DBGBXVR\(<n>\) characteristics are:

**Purpose**

Holds a value for use in breakpoint matching, to support VMID matching. Used in conjunction with a control register DBGBCR\(<n>\) and a value register DBGBVR\(<n>\), where EL2 is implemented and breakpoint \(n\) supports Context matching.

**Configurations**

- AArch32 System register DBGBXVR\(<n>\) bits [31:0] are architecturally mapped to AArch64 System register DBGBVR\(<n>\)_EL1[63:32].
- AArch32 System register DBGBXVR\(<n>\) bits [31:0] are architecturally mapped to External register DBGBVR\(<n>\)_EL1[63:32].

--- Note ---

Writes to DBGBXVR\(<n>\) do not modify DBGBVR\(<n>\)_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGBXVR\(<n>\) are UNDEFINED.

How this register is interpreted depends on the value of DBGBCR\(<n>\).BT.

- When DBGBCR\(<n>\).BT is 0b10xx, this register holds a VMID.
- When DBGBCR\(<n>\).BT is 0b11xx, this register holds a Context ID.

For other values of DBGBCR\(<n>\).BT, this register is RES0.

Accesses to this register are UNDEFINED in any of the following cases:

- Breakpoint \(n\) is not implemented.
- Breakpoint \(n\) does not support Context matching.
- EL2 is not implemented.

For more information, see the description of the DBGDIDR.CTX_CMPs field.

**Attributes**

DBGBXVR\(<n>\) is a 32-bit register.

**Field descriptions**

*When DBGBCR\(<n>\).BT == 0b10xx and EL2 is implemented:*

![](image)

- **Bits [31:16]**
  - Reserved, RES0.
- **VMID[15:8]**, bits [15:8]
  - *When FEAT_VMID16 is implemented and VTCR_EL2.VS == 1:*
    - Extension to VMID[7:0]. For more information, see VMID[7:0].
  - The reset behavior of this field is:
    - On a Cold reset, this field resets to an architecturally UNKNOWN value.
  - *Otherwise:*
    - Reserved, RES0.
VMID[7:0], bits [7:0]

VMID value for comparison. The VMID is 8 bits when any of the following are true:

- EL2 is using AArch32.
- VTCR_EL2.VS is 0.
- FEAT_VMID16 is not implemented.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>.BT == 0b11xx and EL2 is implemented:

ContextID2, bits [31:0]

When FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented:

Context ID value for comparison against CONTEXTIDR_EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing DBGBXVR<n>

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC\{<c>\}{<q> \{<opc1>, <Rt>, <CRn>, <CRm}\{, {<opc2>}\}}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0001</td>
<td>m[3:0]</td>
<td>0b001</td>
</tr>
</tbody>
</table>

\[
\text{integer } m = \text{UInt}(CRm<3:0>);
\]

\[
\text{if } m >= \text{NUM_BREAKPOINTS} \text{ then UNDEFINED;}
\]

\[
\text{elsif PSTATE.EL == EL0 then UNDEFINED;}
\]

\[
\text{elsif PSTATE.EL == EL1 then UNDEFINED;}
\]

- if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then UNDEFINED;

- elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TDA && MDCR_EL2.TDE != '00' then AArch32.AArch32SystemAccessTrap(EL2, 0x05);

- elsif HaveEL(EL3) && EDSCR.SDD == '1' then UNDEFINED;

else

- AArch64.AArch64SystemAccessTrap(EL3, 0x05);

- elsif DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then Halt(DebugHalt_SoftwareAccess);

else

- R[t] = DBGBXVR[m];


elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
  end
  if DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
  else
    R[t] = DBGBXVR[m];
  end
elsif PSTATE_EL == EL3 then
  if DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
  else
    R[t] = DBGBXVR[m];
  end
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
  end
  if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
    AArch32.TakeHypTrapException(0x05);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
    AArch32.TakeHypTrapException(0x05);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    end
    if DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
      Halt(DebugHalt_SoftwareAccess);
    else
      DBGBXVR[m] = R[t];
    end
else
  DBGBXVR[m] = R[t];
end
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
    AArch32.TakeHypTrapException(0x05);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    end
    if DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
      Halt(DebugHalt_SoftwareAccess);
    else
      DBGBXVR[m] = R[t];
    end
else
  DBGBXVR[m] = R[t];
end
elsif PSTATE_EL == EL3 then
  if DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
else
  DBGBXVR[m] = R[t];
end

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc opc1 CRn CRm opc2

0b1110 0b000 0b0001 m[3:0] 0b001

integer m = UInt(CRm<3:0>);
else
    DBGXVR[m] = R[t];
G8.3.5 DBGCLAIMCLR, Debug CLAIM Tag Clear register

The DBGCLAIMCLR characteristics are:

**Purpose**

Used by software to read the values of the CLAIM tag bits, and to clear CLAIM tag bits to 0.

The architecture does not define any functionality for the CLAIM tag bits.

--- Note ---

CLAIM tags are typically used for communication between the debugger and target software.

---

CLAIM tags are used in conjunction with the DBGCLAIMSET register.

**Configurations**

AArch32 System register DBGCLAIMCLR bits [31:0] are architecturally mapped to AArch64 System register DBGCLAIMCLR_EL1[31:0].

AArch32 System register DBGCLAIMCLR bits [31:0] are architecturally mapped to External register DBGCLAIMCLR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGCLAIMCLR are UNDEFINED.

An implementation must include eight CLAIM tag bits.

**Attributes**

DBGCLAIMCLR is a 32-bit register.

**Field descriptions**

![Field Descriptions Diagram]

**Bits [31:8]**

Reserved, RAZ/WI.

**CLAIM, bits [7:0]**

Read or clear CLAIM tag bits. Reading this field returns the current value of the CLAIM tag bits.

Writing a 1 to one of these bits clears the corresponding CLAIM tag bit to 0. This is an indirect write to the CLAIM tag bits. A single write operation can clear multiple CLAIM tag bits to 0.

Writing 0 to one of these bits has no effect.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.

**Accessing DBGCLAIMCLR**

Accesses to this register use the following encodings in the System register encoding space:
MRC{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1001</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    UNDEFINED;
elsif EL2Enabled() & !ELUsingAArch32(EL2) & MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HDCR.<TDE,TDA> != '00' then
    AArch32.TakeHypTrapException(0x05);
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
elsesel
      R[t] = DBGCLAIMCLR;
else
  if PSTATE.EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
      UNDEFINED;
eI elif EL2Enabled() & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
        if Halted() & EDSCR.SDD == '1' then
          UNDEFINED;
else
  R[t] = DBGCLAIMCLR;
eI elif PSTATE.EL == EL3 then
  R[t] = DBGCLAIMCLR;
eI else
  DBGCLAIMCLR = R[t];
eI elif PSTATE.EL == EL2 then

MC{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1001</td>
<td>0b110</td>
</tr>
</tbody>
</table>
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAAP32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAAP32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        DBGCLAIMCLR = R[t];
    end
elsif PSTATE.EL == EL3 then
    DBGCLAIMCLR = R[t];
G8.3.6 DBGCLAIMSET, Debug CLAIM Tag Set register

The DBGCLAIMSET characteristics are:

**Purpose**

- Used by software to set the CLAIM tag bits to 1.
- The architecture does not define any functionality for the CLAIM tag bits.

--- **Note** ---

 CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMCLR register.

**Configurations**

- AArch32 System register DBGCLAIMSET bits [31:0] are architecturally mapped to AArch64 System register DBGCLAIMSET_EL1[31:0].
- AArch32 System register DBGCLAIMSET bits [31:0] are architecturally mapped to External register DBGCLAIMSET_EL1[31:0].
- This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGCLAIMSET are UNDEFINED.
- An implementation must include eight CLAIM tag bits.

**Attributes**

- DBGCLAIMSET is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>8-7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAZ/WI</td>
<td>CLAIM</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:8]**

- Reserved, RAZ/WI.

**CLAIM, bits [7:0]**

- Set CLAIM tag bits.
- This field is RAO.
- Writing a 1 to one of these bits sets the corresponding CLAIM tag bit to 1. This is an indirect write to the CLAIM tag bits. A single write operation can set multiple CLAIM tag bits to 1.
- Writing 0 to one of these bits has no effect.
- The reset behavior of this field is:
  - On a Cold reset, this field resets to 0.

**Accessing DBGCLAIMSET**

Accesses to this register use the following encodings in the System register encoding space:
MRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
      AArch32.TakeHypTrapException(0x05);
elif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
      R[t] = DBGCLAIMSET;
elseif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
  else
    R[t] = DBGCLAIMSET;
elseif PSTATE.EL == EL3 then
  R[t] = DBGCLAIMSET;
else
  R[t] = DBGCLAIMSET;

MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b1000</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
      AArch32.TakeHypTrapException(0x05);
elif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
  else
    DBGCLAIMSET = R[t];
elseif PSTATE.EL == EL2 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
  UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
  else
    DBGCLAIMSET = R[t];
elsif PSTATE.EL == EL3 then
  DBGCLAIMSET = R[t];
G8.3.7   DBGDCCINT, DCC Interrupt Enable Register

The DBGDCCINT characteristics are:

**Purpose**

Enables interrupt requests to be signaled based on the DCC status flags.

**Configurations**

AArch32 System register DBGDCCINT bits [31:0] are architecturally mapped to AArch64 System register \( \text{MDCCINT\_EL1}[31:0] \).

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGDCCINT are UNDEFINED.

**Attributes**

DBGDCCINT is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>30</td>
<td>RX, DCC interrupt request enable control for DTRRX.</td>
</tr>
<tr>
<td></td>
<td>Enables a common ( \text{COMMIRQ} ) interrupt request to be signaled</td>
</tr>
<tr>
<td></td>
<td>based on the DCC status flags.</td>
</tr>
<tr>
<td>0</td>
<td>No interrupt request generated by DTRRX.</td>
</tr>
<tr>
<td>1</td>
<td>Interrupt request will be generated on RXfull == 1.</td>
</tr>
<tr>
<td></td>
<td>If legacy ( \text{COMMRX} ) and ( \text{COMMTX} ) signals are</td>
</tr>
<tr>
<td></td>
<td>implemented, then these are not affected by the value of this bit.</td>
</tr>
<tr>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On a Warm reset, this field resets to 0.</td>
</tr>
<tr>
<td>29</td>
<td>TX, DCC interrupt request enable control for DTRTX.</td>
</tr>
<tr>
<td></td>
<td>Enables a common ( \text{COMMIRQ} ) interrupt request to be signaled</td>
</tr>
<tr>
<td></td>
<td>based on the DCC status flags.</td>
</tr>
<tr>
<td>0</td>
<td>No interrupt request generated by DTRTX.</td>
</tr>
<tr>
<td>1</td>
<td>Interrupt request will be generated on TXfull == 0.</td>
</tr>
<tr>
<td></td>
<td>If legacy ( \text{COMMRX} ) and ( \text{COMMTX} ) signals are</td>
</tr>
<tr>
<td></td>
<td>implemented, then these are not affected by the value of this bit.</td>
</tr>
<tr>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On a Warm reset, this field resets to 0.</td>
</tr>
<tr>
<td>28-0</td>
<td>Reserved, RES0</td>
</tr>
</tbody>
</table>

**Accessing DBGDCCINT**

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
    AArch32.TakeHypTrapException(0x05);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
    AArch32.TakeHypTrapException(0x05);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    endif
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch32.TakeMonitorTrapException();
    endif
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    endif
  else
    R[t] = DBGDCCINT;
  endif
elsif PSTATE.EL == EL1 then
  R[t] = DBGDCCINT;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
    AArch32.TakeHypTrapException(0x05);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
    AArch32.TakeHypTrapException(0x05);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    endif
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch32.TakeMonitorTrapException();
    endif
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    endif
  else
    AArch32.TakeMonitorTrapException();
  endif
else
  R[t] = DBGDCCINT;
endif

### Table: MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}###

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

---

AArch32 System Register Descriptions
G8.3 Debug registers

ID081822
Non-Confidential

Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
  else
    R[t] = DBGDCINT;
elsif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
    AArch32.TakeMonitorTrapException();
  else
    R[t] = DBGDCINT;

MCR<coproc>, <CRn>, <CRm>, <op1>, <op2>, <op3>, <op4>, <op5>

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b0000</td>
</tr>
</tbody>
</table>
UNDEFINED;
elif Halted() && haveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap
priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
  UNDEFINED;
elif haveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
elif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
    AArch32.TakeMonitorTrapException();
  else
    DBGDCCINT = R[t];
G8.3.8   DBGDEVID, Debug Device ID register 0

The DBGDEVID characteristics are:

**Purpose**

Adds to the information given by the DBGDIDR by describing other features of the debug implementation.

**Configurations**

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGDEVID are UNDEFINED.

This register is required in all implementations.

**Attributes**

DBGDEVID is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>24</th>
<th>23</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIDMask</td>
<td>AuxRegs</td>
<td>DoubleLock</td>
<td>VirtExtns</td>
<td>WPAddrMask</td>
<td>BPAddrMask</td>
<td>VectorCatch</td>
<td>PCSample</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CIDMask, bits [31:28]**

Indicates the level of support for the Context ID matching breakpoint masking capability. Defined values are:

- 0b0000  Context ID masking is not implemented.
- 0b0001  Context ID masking is implemented.

All other values are reserved. The value of this for Armv8 is 0b0000.

**AuxRegs, bits [27:24]**

Indicates support for Auxiliary registers. Permitted values for this field are:

- 0b0000  None supported.
- 0b0001  Support for External Debug Auxiliary Control Register, EDACR.

All other values are reserved.

**DoubleLock, bits [23:20]**

OS Double Lock implemented. Defined values are:

- 0b0000  OS Double Lock is not implemented. DBGOSDLR is RAZ/WI.
- 0b0001  OS Double Lock is implemented. DBGOSDLR is RW.

FEAT_DoubleLock implements the functionality identified by the value 0b0001.

All other values are reserved.

**VirtExtns, bits [19:16]**

Indicates whether EL2 is implemented. Defined values are:

- 0b0000  EL2 is not implemented.
- 0b0001  EL2 is implemented.

All other values are reserved.
VectorCatch, bits [15:12]

Defines the form of Vector Catch exception implemented. Defined values are:

- 0b0000  Address matching Vector Catch exception implemented.
- 0b0001  Exception matching Vector Catch exception implemented.

All other values are reserved.

BPAddrMask, bits [11:8]

Indicates the level of support for the instruction address matching breakpoint masking capability. Defined values are:

- 0b0000  Breakpoint address masking might be implemented. If not implemented, DBGBCR<n>[28:24] is RAZ/WI.
- 0b0001  Breakpoint address masking is implemented.
- 0b1111  Breakpoint address masking is not implemented. DBGBCR<n>[28:24] is RES0.

All other values are reserved. The value of this for Armv8 is 0b1111.

WPAddrMask, bits [7:4]

Indicates the level of support for the data address matching watchpoint masking capability. Defined values are:

- 0b0000  Watchpoint address masking might be implemented. If not implemented, DBGWCR<n>.MASK (Address mask) is RAZ/WI.
- 0b0001  Watchpoint address masking is implemented.
- 0b1111  Watchpoint address masking is not implemented. DBGWCR<n>.MASK (Address mask) is RES0.

All other values are reserved. The value of this for Armv8 is 0b0001.

PCSample, bits [3:0]

Indicates the level of PC Sample-based Profiling support using external debug registers. Defined values are:

- 0b0000  PC Sample-based Profiling Extension is not implemented in the external debug registers space.
- 0b0010  Only EDPCSР and EDCIDSР are implemented. This option is only permitted if EL3 and EL2 are not implemented.
- 0b0011  EDPCSР, EDCIDSР, and EDVIDSR are implemented.

All other values are reserved.

When FEAT_PCSRv8p2 is implemented, the only permitted value is 0b0000.

--- Note ---

FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors register space, as indicated by the value of PMDEVID.PCSample.

Accessing DBGDEVID

Accesses to this register use the following encodings in the System register encoding space:
MRC{<c>}{<q>}{<coproc>}, {#}{<opc1>}, <Rt>, <CRn>, <CRm>{, {#}{<opc2>}}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b00010</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
    AArch32.TakeHypTrapException(0x05);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    end
  else
    R[t] = DBGDEVID;
  else
    if PSTATE_EL == EL2 then
      if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        UNDEFINED;
      elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        end
      else
        R[t] = DBGDEVID;
      else
        if PSTATE_EL == EL3 then
          R[t] = DBGDEVID;
        

else
  R[t] = DBGDEVID;
elsif PSTATE_EL == EL3 then
  R[t] = DBGDEVID;
G8.3.9 DBGDEVID1, Debug Device ID register 1

The DBGDEVID1 characteristics are:

Purpose

Adds to the information given by the DBGDIDR by describing other features of the debug implementation.

Configurations

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGDEVID1 are UNDEFINED.

This register is required in all implementations.

Attributes

DBGDEVID1 is a 32-bit register.

Field descriptions

Bits [31:4]

Reserved, RES0.

PCSROffset, bits [3:0]

This field indicates the offset applied to PC samples returned by reads of EDPCSR. Permitted values of this field in Armv8 are:

- 0b0000: EDPCSR is not implemented.
- 0b0010: EDPCSR implemented. Samples have no offset applied and do not sample the instruction set state in AArch32 state.

When FEAT_PCSRv8p2 is implemented, the only permitted value is 0b0000.

Note

FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors register space, as indicated by the value of PMDEVID.PCSample.

Accessing DBGDEVID1

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC\{<c}\{<q}\} <coproc>, \{#<opc1>, <Rt>, <CRn>, <CRm>\}, \{#<opc2}\} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0111</td>
<td>0b0001</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & IELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
        UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
  AArch32.TakeHypTrapException(0x05);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
  else
    R[t] = DBGDEVID1;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
else
  R[t] = DBGDEVID1;
elsif PSTATE.EL == EL3 then
  R[t] = DBGDEVID1;
G8.3.10 DBGDEVID2, Debug Device ID register 2

The DBGDEVID2 characteristics are:

**Purpose**

Reserved for future descriptions of features of the debug implementation.

**Configurations**

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGDEVID2 are UNDEFINED.

**Attributes**

DBGDEVID2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

Accessing DBGDEVID2

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{c\}\{q\} <\text{coproc}>, \{#\}<\text{opc1}>, <Rt>, <CRn>, <CRm>,\{ #\}<\text{opc2}> \]

\[
\begin{array}{cccccc}
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
0b1110 & 0b0000 & 0b0111 & 0b0000 & 0b111 \\
\end{array}
\]

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
    AArch32.AArch32SystemAccessTrap(EL2, 0x05);
  elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    end else
    R[t] = DBGDEVID2;
  else
    R[t] = DBGDEVID2;
  endif
else
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      
      \]
AArch64.AArch32SystemAccessTrap(EL3, 0x05);

else
  R[t] = DBGDEVID2;
elsif PSTATE.EL == EL3 then
  R[t] = DBGDEVID2;
G8.3.11  **DBGDIDR, Debug ID Register**

The DBGDIDR characteristics are:

**Purpose**
Specifies which version of the Debug architecture is implemented, and some features of the debug implementation.

**Configurations**
This register is present only when AArch32 is supported. Otherwise, direct accesses to DBGDIDR are UNDEFINED.

If EL1 cannot use AArch32 then the implementation of this register is OPTIONAL and deprecated.

**Attributes**
DBGDIDR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Permitted values</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRPs, bits</td>
<td>The number of watchpoints implemented, minus 1.</td>
<td>from 0b0001 to 0b1111</td>
</tr>
<tr>
<td>BRPs, bits</td>
<td>The number of breakpoints implemented, minus 1.</td>
<td>from 0b0001 to 0b1111</td>
</tr>
<tr>
<td>CTX_CMPs, bits</td>
<td>The number of breakpoints that can be used for Context matching, minus 1.</td>
<td>from 0b0000 to 0b1111</td>
</tr>
<tr>
<td>Version, bits</td>
<td>Debug architecture version. Indicates presence of Armv8 debug architecture. Defined values are:</td>
<td>0b0000 Not supported, 0b0001 Armv6, v6 Debug architecture, with System registers access, 0b0010 Armv6, v6.1 Debug architecture, with System registers access</td>
</tr>
</tbody>
</table>

**WRPs, bits [31:28]**
The number of watchpoints implemented, minus 1.

**BRPs, bits [27:24]**
The number of breakpoints implemented, minus 1.

**CTX_CMPs, bits [23:20]**
The number of breakpoints that can be used for Context matching, minus 1.

**Version, bits [19:16]**
Debug architecture version. Indicates presence of Armv8 debug architecture. Defined values are:
0b0011  Armv7, v7 Debug architecture, with only baseline System registers.
0b0100  Armv7, v7 Debug architecture, with all System registers implemented.
0b0101  Armv7, v7.1 Debug architecture, with System registers access.
0b0110  Armv8 debug architecture.
0b0111  Armv8 debug architecture with Virtualization Host Extensions.
0b1000  Armv8.2 debug architecture, FEAT_Debugv8p2.
0b1001  Armv8.4 debug architecture, FEAT_Debugv8p4.
0b1010  Armv8.8 debug architecture, FEAT_Debugv8p8.

All other values are reserved.
The values 0b0000, 0b0001, 0b0010, 0b0011, 0b0100, and 0b0101 are not permitted in Armv8.

FEAT_VHE adds the functionality identified by the value 0b0111.
FEAT_Debugv8p2 adds the functionality identified by the value 0b1000.
FEAT_Debugv8p4 adds the functionality identified by the value 0b1001.
FEAT_Debugv8p8 adds the functionality identified by the value 0b1010.

From Armv8.1, when FEAT_VHE is implemented the value 0b0110 is not permitted.
From Armv8.2, the values 0b0110 and 0b0111 are not permitted.
From Armv8.4, the value 0b1000 is not permitted.
From Armv8.8, the value 0b1001 is not permitted.

Bit [15]
Reserved, RES1.

nSUHD_imp, bit [14]
Previously indicated that Secure User Halting Debug is not implemented.
The value of this bit must match the value of the SE_imp bit.

Bit [13]
Reserved, RES0.

SE_imp, bit [12]
EL3 implemented. The meanings of the values of this bit are:
0b0   EL3 not implemented.
0b1   EL3 implemented.
The value of this bit must match the value of the nSUHD_imp bit.

Bits [11:0]
Reserved, RES0.

Accessing DBGDIDR
Arm deprecates any access to this register from EL0.
Accesses to this register use the following encodings in the System register encoding space:
if Halted() && ConstrantUnpredictableBool(UNPREDICTABLE_IGNORETRAPINDEBUG) then
    \( R[t] = \text{DBGIDR} \);
elsif PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == ‘1’ && boolean IMPLEMENTATION_DEFINED “EL3 trap priority when SDD == ‘1’” && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == ‘1’ then
        UNDEFINED;
    elsif \( \text{EL2Enabled()} \) && ELUsingAArch32(EL2) && EDSCR.TGE == ‘1’ then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
else
        AArch64.AArch32SystemAccessTrap(EL1, 0x05);
    elsif EL2Enabled() && EDSCR.TGE == ‘1’ then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        R[t] = DBGIDR;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == ‘1’ && boolean IMPLEMENTATION_DEFINED “EL3 trap priority when SDD == ‘1’” && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == ‘1’ then
        UNDEFINED;
    elsif EL2Enabled() && ELUsingAArch32(EL2) && EDSCR.TGE == ‘1’ then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && EDSCR.TGE == ‘1’ then
        AArch32.TakeHypTrapException(0x00);
    else
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && EDSCR.TGE == ‘1’ then
        AArch32.TakeHypTrapException(0x00);
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        R[t] = DBGIDR;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == ‘1’ && boolean IMPLEMENTATION_DEFINED “EL3 trap priority when SDD == ‘1’” && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == ‘1’ then
        UNDEFINED;
    elsif EL2Enabled() && ELUsingAArch32(EL2) && EDSCR.TGE == ‘1’ then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && EDSCR.TGE == ‘1’ then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        R[t] = DBGIDR;
elsif PSTATE.EL == EL3 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == ‘1’ && boolean IMPLEMENTATION_DEFINED “EL3 trap priority when SDD == ‘1’” && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == ‘1’ then
        UNDEFINED;
    elsif EL2Enabled() && ELUsingAArch32(EL2) && EDSCR.TGE == ‘1’ then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    else
        R[t] = DBGIDR;
else
    R[t] = DBGIDR;
G8.3.12 DBGDRAR, Debug ROM Address Register

The DBGDRAR characteristics are:

**Purpose**

Defines the base physical address of a 4KB-aligned memory-mapped debug component, usually a ROM table that locates and describes the memory-mapped debug components in the system. Armv8 deprecates any use of this register.

**Configurations**

AArch32 System register DBGDRAR bits [63:0] are architecturally mapped to AArch64 System register MDRAR_EL1[63:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to DBGDRAR are UNDEFINED.

DBGDRAR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, bits [31:0] are read.

If EL3 cannot use AArch32 then the implementation of this register is OPTIONAL and deprecated.

**Attributes**

DBGDRAR is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-48</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>47-12</td>
<td>ROMADD[47:12]</td>
</tr>
<tr>
<td>11-2</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>1-0</td>
<td>Valid</td>
</tr>
</tbody>
</table>

**Bits [63:48]**

Reserved, RES0.

**ROMADD[47:12], bits [47:12]**

Bits[47:12] of the ROM table physical address.

If the physical address size in bits (PAsize) is less than 48 then the register bits corresponding to ROMADD[47:PAsize] are RES0.

Bits [11:0] of the ROM table physical address are zero.

Arm strongly recommends that bits ROMADD[(PAsize-1):32] are zero in any system where the implementation only supports execution in AArch32 state.

In an implementation that includes EL3, ROMADD is an address in Non-secure memory. It is IMPLEMENTATION DEFINED whether the ROM table is also accessible in Secure memory.

If DBGDRAR.Valid == 0b00, then this field is UNKNOWN.

**Bits [11:2]**

Reserved, RES0.

**Valid, bits [1:0]**

This field indicates whether the ROM Table address is valid.

0b00  ROM Table address is not valid. Software must ignore ROMADD.

0b11  ROM Table address is valid.

Other values are reserved.
Arm recommends implementations set this field to zero.

**Accessing DBGDRAR**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{\langle c\rangle,\langle q\rangle\} \langle \text{coproc}, \{\langle #\rangle\langle opc1\rangle, \langle Rt\rangle, \langle CRn\rangle, \langle CRm\rangle, \{\langle #\rangle\langle opc2\rangle\} \}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b00</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    \[R[t] = \text{DBGDRAR}<31:0>\];
elsif PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        UNDEFINED;
elsif !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
else
    AArch64.AArch32SystemAccessTrap(EL1, 0x05);
elsif ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif !EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif !EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDSCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
else
    R[t] = DBGDRAR<31:0>;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        UNDEFINED;
elsif !ELUsingAArch32(EL2) && MDSCR_EL2.TDCC == '1' then
    if Halted() && EDSCR.SDD == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDSCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
else
    R[t] = DBGDRAR<31:0>;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDSCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0×05);
else
  R[t] = DBGDRAR<31:0>;
elsif PSTATE.EL == EL3 then
  R[t] = DBGDRAR<31:0>;

MRRC<coproc>, <CRm>, <opc1>, <Rt>, <Rt2>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b0001</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if Halted() & ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
  (R[t], R[t]) = (DBGDRAR<63:32>, DBGDRAR<31:0>);
elsif PSTATE.EL == EL0 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD != '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD = '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDA != '1' then
    UNDEFINED;
elsif !ELUsingAArch32(EL1) & MDSCR_EL1.TDCC == '1' then
  if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0×0C);
else
  AArch64.AArch32SystemAccessTrap(EL1, 0×0C);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || HDCR.<TDE,TDRA> != '00') then
  AArch32.TakeHypTrapException(0×0C);
else
  UNDEFINED;
elsif EL2Enabled() & !ELUsingAArch32(EL2) & HaveEL(EL3) && !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0×0C);
else
  (R[t], R[t]) = (DBGDRAR<63:32>, DBGDRAR<31:0>);
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD = '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || HDCR.<TDE,TDRA> != '00') then
  AArch32.TakeHypTrapException(0×0C);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0×0C);
else
  (R[t], R[t]) = (DBGDRAR<63:32>, DBGDRAR<31:0>);
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD = '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
  if Halted() & EDSCR.SDD == '1' then

UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL3, 0x0C);
else
(R[t2], R[t]) = (DBGDRAR<63:32>, DBGDRAR<31:0>);
elsis PSTATE_EL == EL3 then
(R[t2], R[t]) = (DBGDRAR<63:32>, DBGDRAR<31:0>);
G8.3.13  DBGDSAR, Debug Self Address Register

The DBGDSAR characteristics are:

**Purpose**

In earlier versions of the Arm Architecture, this register defines the offset from the base address defined in DBGDRAR of the physical base address of the debug registers for the PE. Armv8 deprecates any use of this register.

**Configurations**

This register is present only when AArch32 is supported. Otherwise, direct accesses to DBGDSAR are UNDEFINED.

DBGDSAR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, bits [31:0] are read.

If EL1 cannot use AArch32 then the implementation of this register is OPTIONAL and deprecated.

**Attributes**

DBGDSAR is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:2]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[1:0]</td>
<td>Reserved, RAZ.</td>
</tr>
</tbody>
</table>

**Accessing DBGDSAR**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}(<c>)(<q>) \langle\text{coproc}, \{\#}\langle\text{opc1}, \langle\text{Rt}, \langle\text{CRn}, \langle\text{CRm}\rangle, \{\#}\langle\text{opc2}\rangle\rangle\rangle\rangle\rangle
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0010</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if Halted() && ConstrainingUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
\[ R[t] = \text{DBGDSAR}[31:0]; \]
elsf if PSTATE_EL == EL0 then
\[ R[t] = \text{DBGDSAR}[31:0]; \]
elesf if Halted() && HaveEL(EL3) && EDSCR.SDO = '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDO == 1" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
UNDEFINED;
elesf if !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
\[ \text{AArch64.AArch32SystemAccessTrap(EL2, 0x05)}; \]

else
    AArch64.AArch32SystemAccessTrap(EL1, 0x05);
elsif EL2Enabled() && DBGDSRExt.UDDCDis == '1' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
        AArch32.TakeHypTrapException(0x05);
    else
        UNDEFINED;
    end if;
else
    UNDEFINED;
end if;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
else
    R[t] = DBGDSAR<31:0>;
end if;
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" & EDSCR.SDD == '1' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDE,TDRA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        end if;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    end if;
else
    R[t] = DBGDSAR<31:0>;
end if;
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" & EDSCR.SDD == '1' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDE,TDRA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        end if;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    end if;
else
    R[t] = DBGDSAR<31:0>;
end if;
elsif PSTATE_EL == EL3 then
    R[t] = DBGDSAR<31:0>;
end if;

MRRC<q> coproc, <opc1>, <Rt>, <Rt2>, <CRm>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b0010</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if Halted() && ConstrunPredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    R[t2], R[t] = (DBGDSAR<63:32>, DBGDSAR<31:0>);
elsif PSTATE_EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" & EDSCR.SDD == '1' then
        UNDEFINED;
    elsif !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
        end if;
    else
        AArch32.TakeHypTrapException(0x00);
AArch64.AArch32SystemAccessTrap(EL2, 0x0C);
else
  AArch64.AArch32SystemAccessTrap(EL1, 0x0C);
elsif EL2Enabled() && !ELUsingAArch32(EL1) && DBGDSAR<63:32> == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x0C);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
  AArch32.TakeHypTrapException(EL2, 0x0C);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x0C);
  end if;
else
  (R[t2], R[t]) = (DBGDSAR<63:32>, DBGDSAR<31:0>);
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'
  & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDRA> != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x0C);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDRA> != '00' then
    AArch32.TakeHypTrapException(EL2, 0x0C);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x0C);
    end if;
else
  (R[t2], R[t]) = (DBGDSAR<63:32>, DBGDSAR<31:0>);
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'
  & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x0C);
    end if;
else
  (R[t2], R[t]) = (DBGDSAR<63:32>, DBGDSAR<31:0>);
elsif PSTATE.EL == EL3 then
  (R[t2], R[t]) = (DBGDSAR<63:32>, DBGDSAR<31:0>);
G8.3.14  DBGDSCRext, Debug Status and Control Register, External View

The DBGDSCRext characteristics are:

**Purpose**

Main control register for the debug implementation.

**Configurations**

AArch32 System register DBGDSCRext bits [31:0] are architecturally mapped to AArch64 System register MDSCR_EL1[31:0].

AArch32 System register DBGDSCRext bit [15] is architecturally mapped to AArch32 System register DBGDSCRint[15].

AArch32 System register DBGDSCRext bit [12] is architecturally mapped to AArch32 System register DBGDSCRint[12].

AArch32 System register DBGDSCRext bits [5:2] are architecturally mapped to AArch32 System register DBGDSCRint[5:2].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGDSCRext are UNDEFINED.

This register is required in all implementations.

**Attributes**

DBGDSCRext is a 32-bit register.

**Field descriptions**

```
   31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  7  6  5  2  1  0
  | TFO |
  | RES0 |
  | RXO  |
  | RXU  |
  | TXO  |
  | TXU  |
  | INTdis |
  | TDA  |
  | SC2  |
  | RES0 |
  | RES0 |
  | UDCCdis |
  | HDE  |
  | MDBGen |
  | SPIDdis |
  | SPNIDdis |
  | RES0 |
  | RES0 |
  | ERR  |
```

**TFO, bit [31]**

*When FEAT_TRF is implemented:*

Trace Filter override. Used for save/restore of EDSR_TFO.

When the OS Lock is unlocked, DBGOSLR.OSLK == 0, software must treat this bit as UNK/SBZP.

When the OS Lock is locked, DBGOSLR.OSLK == 1, this bit holds the value of EDSR_TFO.

Reads and writes of this bit are indirect accesses to EDSR_TFO.

Accessing this field has the following behavior:

- When DBGOSLR.OSLK == 1, access to this field is RW.
- When DBGOSLR.OSLK == 0, access to this field is RO.

*Otherwise:*

Reserved, RES0.

**RXfull, bit [30]**

DTRRX full. Used for save/restore of EDSR_RXfull.

When DBGOSLR.OSLK == 0, software must treat this bit as UNK/SBZP.
When `DBGOSLSR.OSLK == 1`, this bit holds the value of `EDSCR.RXfull`. Reads and writes of this bit are indirect accesses to `EDSCR.RXfull`.

Arm deprecates use of this bit other than for save/restore. Use `DBGDSCRint` to access the DTRRX full status.

The reset behavior of this field is:
- The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:
- When `DBGOSLSR.OSLK == 1`, access to this field is RW.
- When `DBGOSLSR.OSLK == 0`, access to this field is RO.

**TXfull, bit [29]**

DTRTX full. Used for save/restore of `EDSCR.TXfull`.

When `DBGOSLSR.OSLK == 0`, software must treat this bit as UNK/SBZP.

When `DBGOSLSR.OSLK == 1`, this bit holds the value of `EDSCR.TXfull`. Reads and writes of this bit are indirect accesses to `EDSCR.TXfull`.

Arm deprecates use of this bit other than for save/restore. Use `DBGDSCRint` to access the DTRTX full status.

The reset behavior of this field is:
- The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:
- When `DBGOSLSR.OSLK == 1`, access to this field is RW.
- When `DBGOSLSR.OSLK == 0`, access to this field is RO.

**Bit [28]**

Reserved, RES0.

**RXO, bit [27]**

Used for save/restore of `EDSCR.RXO`.

When `DBGOSLSR.OSLK == 0`, software must treat this bit as UNK/SBZP.

When `DBGOSLSR.OSLK == 1`, this bit holds the value of `EDSCR.RXO`. Reads and writes of this bit are indirect accesses to `EDSCR.RXO`.

When `DBGOSLSR.OSLK == 1`, if bits [27,6] of the value written to `DBGDSCRext` are \{1,0\}, that is, the RXO bit is 1 and the ERR bit is 0, the PE sets `EDSCR.{RXO,ERR}` to UNK/SBZP values.

The reset behavior of this field is:
- The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:
- When `DBGOSLSR.OSLK == 1`, access to this field is RW.
- When `DBGOSLSR.OSLK == 0`, access to this field is RO.

**TXU, bit [26]**

Used for save/restore of `EDSCR.TXU`.

When `DBGOSLSR.OSLK == 0`, software must treat this bit as UNK/SBZP.

When `DBGOSLSR.OSLK == 1`, this bit holds the value of `EDSCR.TXU`. Reads and writes of this bit are indirect accesses to `EDSCR.TXU`.

When `DBGOSLSR.OSLK == 1`, if bits [26,6] of the value written to `DBGDSCRext` are \{1,0\}, that is, the TXU bit is 1 and the ERR bit is 0, the PE sets `EDSCR.{TXU,ERR}` to UNK/SBZP values.

The reset behavior of this field is:
- The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:
- When `DBGOSLSR.OSLK == 1`, access to this field is RW.
• When DBGOSLSR.OSLK == 0, access to this field is RO.

**Bits [25:24]**

Reserved, RES0.

**INTdis, bits [23:22]**

Used for save/restore of EDSCR.INTdis.

When DBGOSLSR.OSLK == 0, this field is RO, and software must treat it as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this field holds the value of EDSCR.INTdis. Reads and writes of this field are indirect accesses to EDSCR.INTdis.

The reset behavior of this field is:

• The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

• When DBGOSLSR.OSLK == 1, access to this field is RW.

• When DBGOSLSR.OSLK == 0, access to this field is RO.

**TDA, bit [21]**

Used for save/restore of EDSCR.TDA.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.TDA. Reads and writes of this bit are indirect accesses to EDSCR.TDA.

The reset behavior of this field is:

• The architected behavior of this field determines the value it returns after a reset.

Accessing this field has the following behavior:

• When DBGOSLSR.OSLK == 1, access to this field is RW.

• When DBGOSLSR.OSLK == 0, access to this field is RO.

**Bit [20]**

Reserved, RES0.

**SC2, bit [19]**

*When FEAT_PCSRv8 is implemented, FEAT_VHE is implemented and FEAT_PCSRv8p2 is not implemented:*

Used for save/restore of EDSCR.SC2.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.SC2. Reads and writes of this bit are indirect accesses to EDSCR.SC2.

Accessing this field has the following behavior:

• When DBGOSLSR.OSLK == 1, access to this field is RW.

• When DBGOSLSR.OSLK == 0, access to this field is RO.

**Otherwise:**

Reserved, RES0.

**NS, bit [18]**

Non-secure status.

Arm deprecates use of this field.

0b0 Secure state.

0b1 Non-secure state.

Access to this field is RO.
SPNIDis, bit [17]

When EL3 is implemented:

Secure privileged profiling disabled status bit.

0b0  Profiling allowed in Secure privileged modes.
0b1  Profiling prohibited in Secure privileged modes.

This field reads as 0 if any of the following applies, and reads as 1 otherwise:

- FEAT_Debugv8p2 is not implemented and ExternalSecureNoninvasiveDebugEnabled() returns TRUE.
- EL3 is using AArch32 and the value of SDCR.SPME is 1.
- EL3 is using AArch64 and the value of MDCR_EL3.SPME is 1.

Arm deprecates use of this field.

Access to this field is RO.

Otherwise:

Reserved, RES0.

SPIDis, bit [16]

When EL3 is implemented:

Secure privileged AArch32 invasive self-hosted debug disabled status bit. The value of this bit depends on the value of SDCR.SPD and the pseudocode function AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled().

0b0  Self-hosted debug enabled in Secure privileged AArch32 modes.
0b1  Self-hosted debug disabled in Secure privileged AArch32 modes.

This bit reads as 1 if any of the following is true and reads as 0 otherwise:

- EL3 is using AArch32 and SDCR.SPD has the value 0b10.
- EL3 is using AArch64 and MDCR_EL3.SPD32 has the value 0b10.
- EL3 is using AArch32, SDCR.SPD has the value 0b00, and AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled() returns FALSE.
- EL3 is using AArch64, MDCR_EL3.SPD32 has the value 0b00, and AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled() returns FALSE.

Arm deprecates use of this field.

Access to this field is RO.

Otherwise:

Reserved, RES0.

MDBGen, bit [15]

Monitor debug events enable. Enable Breakpoint, Watchpoint, and Vector Catch exceptions.

0b0  Breakpoint, Watchpoint, and Vector Catch exceptions disabled.
0b1  Breakpoint, Watchpoint, and Vector Catch exceptions enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

HDE, bit [14]

Used for save/restore of EDSCR.HDE.

When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.

When DBGOSLSR.OSLK == 1, this bit holds the value of EDSCR.HDE. Reads and writes of this bit are indirect accesses to EDSCR.HDE.

The reset behavior of this field is:

- The architected behavior of this field determines the value it returns after a reset.
Accessing this field has the following behavior:
- When DBGOSLSR.OSLK == 1, access to this field is RW.
- When DBGOSLSR.OSLK == 0, access to this field is RO.

**Bit [13]**
Reserved, RES0.

**UDCCdis, bit [12]**
Traps EL0 accesses to the DCC registers to Undefined mode.
- 0b0: This control does not cause any instructions to be trapped.
- 0b1: EL0 accesses to the DBGDSCRint, DBGDTRRXint, DBGDTTRXint, DBGDIDR, DBGDSAR, and DBGDRAR are trapped to Undefined mode.

**Note**
All accesses to these registers are trapped, including LDC and STC accesses to DBGDTRRXint and DBGDTTRXint, and MRRR accesses to DBGDSAR and DBGDRAR.

Traps of EL0 accesses to the DBGDTRRXint and DBGDTTRXint are ignored in Debug state.
The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**Bits [11:7]**
Reserved, RES0.

**ERR, bit [6]**
Used for save/restore of EDSR.ERR.
When DBGOSLSR.OSLK == 0, software must treat this bit as UNK/SBZP.
When DBGOSLSR.OSLK == 1, this bit holds the value of EDSR.ERR. Reads and writes of this bit are indirect accesses to EDSR.ERR.
The reset behavior of this field is:
- The architected behavior of this field determines the value it returns after a reset.
Accessing this field has the following behavior:
- When DBGOSLSR.OSLK == 1, access to this field is RW.
- When DBGOSLSR.OSLK == 0, access to this field is RO.

**MOE, bits [5:2]**
Method of Entry for debug exception. When a debug exception is taken to an Exception level using AArch32, this field is set to indicate the event that caused the exception:
- 0b0001: Breakpoint.
- 0b0011: Software breakpoint (BKPT) instruction.
- 0b0101: Vector catch.
- 0b0100: Watchpoint.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [1:0]**
Reserved, RES0.

**Accessing DBGDSCRext**
Individual fields within this register might have restricted accessibility when the OS Lock is unlocked, DBGOSLSR.OSLK == 0. See the field descriptions for more detail.
Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} \{#<opc1>\}, <Rt>, <CRn>, <CRm>\{, \{#<opc2>\}\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elseif EL2Enabled() & !ELUsingAArch32(EL2) & MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
  elseif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    end
  endif
else
  R[t] = DBGDSCRext;
endif

if PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elseif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    end
  endif
else
  R[t] = DBGDSCRext;
endif

if PSTATE.EL == EL3 then
  R[t] = DBGDSCRext;
endif
else
    DBGDSRext = R[t];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        end
    else
        DBGDSRext = R[t];
    end
elsif PSTATE.EL == EL3 then
    DBGDSRext = R[t];
### G8.3.15  DBGDSCRint, Debug Status and Control Register, Internal View

The DBGDSCRint characteristics are:

**Purpose**

Main control register for the debug implementation. This is an internal, read-only view.

**Configurations**

AArch32 System register DBGDSCRint bits [30:29] are architecturally mapped to AArch64 System register MDCCSR_EL0[30:29].

AArch32 System register DBGDSCRint bit [15] is architecturally mapped to AArch64 System register MDSCR_EL1[15].

AArch32 System register DBGDSCRint bit [12] is architecturally mapped to AArch64 System register MDSCR_EL1[12].

AArch32 System register DBGDSCRint bits [5:2] are architecturally mapped to AArch64 System register MDSCR_EL1[5:2].

AArch32 System register DBGDSCRint bit [15] is architecturally mapped to AArch32 System register DBGDSCRext[15].

AArch32 System register DBGDSCRint bit [12] is architecturally mapped to AArch32 System register DBGDSCRext[12].


This register is present only when AArch32 is supported. Otherwise, direct accesses to DBGDSCRint are UNDEFINED.

This register is required in all implementations.

DBGDSCRint. {NS, SPNIdis, SPIdis, MDBGen, UDCCdis, MOE} are UNKNOWN when the register is accessed at EL0. However, although these values are not accessible at EL0 by instructions that are neither UNPREDICTABLE nor return UNKNOWN values, it is permissible for an implementation to return the values of DBGDSCRext. {NS, SPNIdis, SPIdis, MDBGen, UDCCdis, MOE} for these fields at EL0.

It is also permissible for an implementation to return the same values as defined for a read of DBGDSCRint at EL1 or above. (This is the case even if the implementation does not support AArch32 at EL1 or above.)

**Attributes**

DBGDSCRint is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>30</td>
<td>RXfull</td>
</tr>
<tr>
<td>29</td>
<td>DTRRX full. Read-only view of the equivalent bit in the EDSCR.</td>
</tr>
<tr>
<td>28</td>
<td>RES0</td>
</tr>
<tr>
<td>27</td>
<td>SPNIdis</td>
</tr>
<tr>
<td>26</td>
<td>SPIdis</td>
</tr>
<tr>
<td>25</td>
<td>MDBGen</td>
</tr>
<tr>
<td>24</td>
<td>UDCCdis</td>
</tr>
<tr>
<td>23</td>
<td>RES0</td>
</tr>
<tr>
<td>22</td>
<td>RES0</td>
</tr>
<tr>
<td>21</td>
<td>RES0</td>
</tr>
<tr>
<td>20</td>
<td>MOE</td>
</tr>
<tr>
<td>19</td>
<td>RES0</td>
</tr>
<tr>
<td>18</td>
<td>RES0</td>
</tr>
<tr>
<td>17</td>
<td>RES0</td>
</tr>
<tr>
<td>16</td>
<td>RES0</td>
</tr>
<tr>
<td>15</td>
<td>RES0</td>
</tr>
<tr>
<td>14</td>
<td>RES0</td>
</tr>
<tr>
<td>13</td>
<td>RES0</td>
</tr>
<tr>
<td>12</td>
<td>RES0</td>
</tr>
<tr>
<td>11</td>
<td>RES0</td>
</tr>
<tr>
<td>10</td>
<td>RES0</td>
</tr>
<tr>
<td>9</td>
<td>RES0</td>
</tr>
<tr>
<td>8</td>
<td>RES0</td>
</tr>
<tr>
<td>7</td>
<td>RES0</td>
</tr>
<tr>
<td>6</td>
<td>RES0</td>
</tr>
<tr>
<td>5</td>
<td>RES0</td>
</tr>
<tr>
<td>4</td>
<td>RES0</td>
</tr>
<tr>
<td>3</td>
<td>RES0</td>
</tr>
<tr>
<td>2</td>
<td>RES0</td>
</tr>
<tr>
<td>1</td>
<td>RES0</td>
</tr>
<tr>
<td>0</td>
<td>RES0</td>
</tr>
</tbody>
</table>

**Bit [31]**

Reserved, RES0.

**RXfull, bit [30]**

DTRRX full. Read-only view of the equivalent bit in the EDSCR.

**TXfull, bit [29]**

DTRTX full. Read-only view of the equivalent bit in the EDSCR.
Bits [28:19]
Reserved, RES0.

NS, bit [18]
Non-secure status.
Read-only view of the equivalent bit in the DBGDSCRext. Arm deprecates use of this field.

SPNIDdis, bit [17]
Secure privileged non-invasive debug disable.
Read-only view of the equivalent bit in the DBGDSCRext. Arm deprecates use of this field.

SPIDdis, bit [16]
Secure privileged invasive debug disable.
Read-only view of the equivalent bit in the DBGDSCRext. Arm deprecates use of this field.

MDBGen, bit [15]
Monitor debug events enable.
Read-only view of the equivalent bit in the DBGDSCRext.

Bits [14:13]
Reserved, RES0.

UDCCdis, bit [12]
User mode access to Debug Communications Channel disable.
Read-only view of the equivalent bit in the DBGDSCRext. Arm deprecates use of this field.

Bits [11:6]
Reserved, RES0.

MOE, bits [5:2]
Method of Entry for debug exception. When a debug exception is taken to an Exception level using AArch32, this field is set to indicate the event that caused the exception:

- 0b0001 Breakpoint
- 0b0011 Software breakpoint (BKPT) instruction
- 0b0101 Vector catch
- 0b1010 Watchpoint

Read-only view of the equivalent bit in the DBGDSCRext.

Bits [1:0]
Reserved, RES0.

**Accessing DBGDSCRint**

When <Rt> is APSR_nzcv, encoded as R15, then instead of reading the entire register, the access copies DBGDSCRint[31:28] into the PSTATE NZCV flags.

Accesses to this register use the following encodings in the System register encoding space:
MRC<coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>, (#<opc2>)

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
  if t == 15 then
    PSTATE.<N,Z,C,V> = DBGDSCRint<31:28>;
  else
    R[t] = DBGDSCRint;
  elsif PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
      UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
      UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
      UNDEFINED;
    elsif !ELUsingAArch32(EL1) && MDCR_EL1.TDCC == '1' then
      if EL2Enabled() && !ELUsingAArch32(EL2) && HDCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
      else
        AArch64.AArch32SystemAccessTrap(EL1, 0x05);
      end if;
    else
      if EL2Enabled() && !ELUsingAArch32(EL2) && HDCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
      else
        UNDEFINED;
      end if;
    end if;
  elsif ELUsingAArch32(EL1) && DBGDSCRext.UDCCdis == '1' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HDCR_EL2.TGE == '1' then
      AArch32.TakeHypTrapException(0x00);
    else
      UNDEFINED;
    end if;
  else
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TGE == '1' then
      AArch32.TakeHypTrapException(0x05);
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00') then
      AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && (HCR.TGE == '1' || HDCR.<TDE,TDA> != '00') then
      AArch32.TakeHypTrapException(0x05);
    elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
      end if;
    elseif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
      end if;
    else
      if t == 15 then
        PSTATE.<N,Z,C,V> = DBGDSCRint<31:28>;
      else
        R[t] = DBGDSCRint;
      end if;
    end if;
  elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
      UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
      UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
      UNDEFINED;
    elsif !ELUsingAArch32(EL1) && MDCR_EL1.TDCC == '1' then
      if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
      else
        AArch64.AArch32SystemAccessTrap(EL1, 0x05);
      end if;
    else
      if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch32.TakeHypTrapException(0x00);
      else
        UNDEFINED;
      end if;
    end if;
  elsif ELUsingAArch32(EL1) && DBGDSCRext.UDCCdis == '1' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HDCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TGE == '1' then
      AArch32.TakeHypTrapException(0x05);
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00') then
      AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && (HCR.TGE == '1' || HDCR.<TDE,TDA> != '00') then
      AArch32.TakeHypTrapException(0x05);
    elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
      end if;
    elseif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
      end if;
    else
      if t == 15 then
        PSTATE.<N,Z,C,V> = DBGDSCRint<31:28>;
      else
        R[t] = DBGDSCRint;
      end if;
    end if;
  end if;
end if;
elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & ELUSINGAArch32(EL3) & SDCR.TDCC == '1' then UNDEFINED;
elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUSINGAArch32(EL3) & MDCR_EL3.TDA == '1' then UNDEFINED;
elsif EL2Enabled() & !ELUSINGAArch32(EL2) & MDCR_EL2.TDC == '1' then AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif EL2Enabled() & ELUSINGAArch32(EL2) & MDCR_EL2.TDC == '1' then
    AArch32.TakeHypTrapException(0x05);
elsif EL2Enabled() & !ELUSINGAArch32(EL2) & MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif HaveEL(EL3) & !ELUSINGAArch32(EL3) & MDCR_EL3.TDC == '1' then
    if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elsif HaveEL(EL3) & ELUSINGAArch32(EL3) & SDCR.TDCC == '1' then
    if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
    AArch32.TakeMonitorTrapException();
elsif HaveEL(EL3) & !ELUSINGAArch32(EL3) & MDCR_EL3.TDA == '1' then
    if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
else
    if t == 15 then
        PSTATE.<N,Z,C,V> = DBGDSCRint<31:28>;
    else
        R[t] = DBGDSCRint;
    end else
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & ELUSINGAArch32(EL3) & MDCR_EL3.TDC == '1' then UNDEFINED;
elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUSINGAArch32(EL3) & SDCR.TDCC == '1' then UNDEFINED;
elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & ELUSINGAArch32(EL3) & MDCR_EL3.TDA == '1' then UNDEFINED;
elsif HaveEL(EL3) & !ELUSINGAArch32(EL3) & MDCR_EL3.TDA == '1' then
    if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elsif HaveEL(EL3) & ELUSINGAArch32(EL3) & SDCR.TDCC == '1' then
    if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
    AArch32.TakeMonitorTrapException();
elsif HaveEL(EL3) & !ELUSINGAArch32(EL3) & MDCR_EL3.TDA == '1' then
    if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
else
    if t == 15 then
        PSTATE.<N,Z,C,V> = DBGDSCRint<31:28>;
    else
        R[t] = DBGDSCRint;
    end else
    if PSTATE.EL == EL2 then
        if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
            AArch32.TakeMonitorTrapException();
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        end
    else
        if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & ELUSINGAArch32(EL3) & MDCR_EL3.TDC == '1' then UNDEFINED;
elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & ELUSINGAArch32(EL3) & SDCR.TDCC == '1' then UNDEFINED;
elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUSINGAArch32(EL3) & MDCR_EL3.TDA == '1' then UNDEFINED;
elsif HaveEL(EL3) & !ELUSINGAArch32(EL3) & MDCR_EL3.TDA == '1' then
    if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elsif HaveEL(EL3) & ELUSINGAArch32(EL3) & SDCR.TDCC == '1' then
    if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
    AArch32.TakeMonitorTrapException();
elsif HaveEL(EL3) & !ELUSINGAArch32(EL3) & MDCR_EL3.TDA == '1' then
    if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
    AArch32.TakeMonitorTrapException();
else
    if t == 15 then
        PSTATE.<N,Z,C,V> = DBGDSCRint<31:28>;
    else
        R[t] = DBGDSCRint;
    end else
    if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
if \( t = 15 \) then
\[
PSTATE.\{N,Z,C,V\} = DBGDSCRInt[31:28];
\]
else
\[
R[t] = DBGDSCRInt;
\]
G8.3.16 DBGDTRRXext, Debug OS Lock Data Transfer Register, Receive, External View

The DBGDTRRXext characteristics are:

**Purpose**

Used for save/restore of DBGDTRRXint. It is a component of the Debug Communications Channel.

**Configurations**

AArch32 System register DBGDTRRXext bits [31:0] are architecturally mapped to AArch64 System register OSDTRRX_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGDTRRXext are UNDEFINED.

**Attributes**

DBGDTRRXext is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Update DTRRX without side-effect.</td>
</tr>
<tr>
<td></td>
<td>Writes to this register update the value in DTRRX and do not change RXfull.</td>
</tr>
<tr>
<td></td>
<td>Reads of this register return the last value written to DTRRX and do not change RXfull.</td>
</tr>
<tr>
<td></td>
<td>For the full behavior of the Debug Communications Channel, see Chapter H4 The Debug Communication Channel and Instruction Transfer Register.</td>
</tr>
<tr>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
</tbody>
</table>

**Accessing DBGDTRRXext**

Arm deprecates reads and writes of DBGDTRRXext through the System register interface when the OS Lock is unlocked, DBGOSLSR.OSLK == 0.

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC{<c>}{<q>}{<coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>)}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif Halted() & & ConstrainsUnpredictableBool(Unpredictable.IGNORETRAPINDEBUG) then
  R[t] = DBGDTRRXext;
elsif PSTATE.EL == EL1 then
  if Halted() & & HaveEL(EL3) & & EDSCR.SDD == '1' & & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & & !ELUsingAArch32(EL3) & & MDCR_EL3.TDCC == '1' then
    UNDEFINED;
  elsif Halted() & & HaveEL(EL3) & & EDSCR.SDD == '1' & & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & & !ELUsingAArch32(EL3) & & SDCR.TDCC == '1' then
    UNDEFINED;

elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then UNDEFINED;
elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then AArch32.TakeHypTrapException(0x05);
elsif EL2Enabled() && ELUsingAArch32(EL2) && SDCR.TDCC == '1' && HDCR.TDE != '00' then AArch32.TakeHypTrapException(0x05);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC != '00' then AArch32.TakeMonitorTrapException();
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDA != '00' then AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
  AArch32.TakeMonitorTrapException();
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA != '00' then AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
  AArch32.TakeMonitorTrapException();
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SDCR.TDCC == '1' then if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
  AArch32.TakeMonitorTrapException();
else R[t] = DBGDTRRXext;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then if Halted() & EDSCR.SDD == '1' then UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then if Halted() & EDSCR.SDD == '1' then UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then if Halted() & EDSCR.SDD == '1' then UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
  else
    R[t] = DBGDTRRXext;
elsif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
    AArch32.TakeMonitorTrapException();
  else
    R[t] = DBGDTRRXext;
MCR<coproc>,<CRn>,<CRm>,<opc1>,<opc2>,<Rt>

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0000</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRANSPARENT) then
    DBGDTRRXext = R[t];
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && EDUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
        AArch32.TakeHypTrapException(0x05);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        endif
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch32.TakeMonitorTrapException();
        endif
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        endif
    else
        DBGDTRRXext = R[t];
    endif
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && EDUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && EDUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !EDSCR.SDD == '1' then
        UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && EDSCR.SDD == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !EDSCR.SDD == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        endif
    elsif HaveEL(EL3) && EDSCR.SDD == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch32.TakeMonitorTrapException();
        endif
    else
        DBGDTRRXext = R[t];
    endif
endif
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
  else
    DBGDTRRXext = R[t];
elsif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor & SD3R.TDCC == '1' then
    AArch32.TakeMonitorTrapException();
  else
    DBGDTRRXext = R[t];
G8.3.17  DBGDTRRXint, Debug Data Transfer Register, Receive

The DBGDTRRXint characteristics are:

**Purpose**

Transfers data from an external debugger to the PE. For example, it is used by a debugger transferring commands and data to a debug target. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communications Channel.

**Configurations**

AArch32 System register DBGDTRRXint bits [31:0] are architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0].

AArch32 System register DBGDTRRXint bits [31:0] are architecturally mapped to External register DBGDTRRX_EL0[31:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to DBGDTRRXint are UNDEFINED.

**Attributes**

DBGDTRRXint is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Update DTRRX</td>
</tr>
</tbody>
</table>

**Bits [31:0]**

Update DTRRX.

Reads of this register:

- If RXfull is set to 1, return the last value written to DTRRX.
- If RXfull is set to 0, return an UNKNOWN value.

After the read, RXfull is cleared to 0.

For the full behavior of the Debug Communications Channel, see Chapter H4 The Debug Communication Channel and Instruction Transfer Register.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Accessing DBGDTRRXint**

Data can be stored to memory from this register using STC.

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC\{c\}\{q\} <coproc>, \{#<opc1>, <Rt>, <CRn>, <CRm>, \{#<opc2>\}
\]

if Halted() then
    R[t] = DBGDTRRXint;
elsif PSTATE_EL == EL0 then
    if !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then
        Update DTRRX
    end

\[
\begin{array}{cccc}
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
0b1110 & 0b000 & 0b0000 & 0b0101 & 0b000
\end{array}
\]
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
else
    AArch64.AArch32SystemAccessTrap(EL1, 0x05);
elsif ELUsingAArch32(EL1) && DBGDCRext.UDCCdis == '1' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
        AArch32.TakeHypTrapException(0x00);
    else
        UNDEFINED;
    endif
else EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
    AArch32.TakeHypTrapException(0x05);
elsif EL2Enabled() && ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00') then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif EL2Enabled() && ELUsingAArch32(EL2) && (HCR.TGE == '1' || HDCR.<TDE,TDA> != '00') then
    AArch32.TakeHypTrapException(0x05);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
    AArch32.TakeMonitorTrapException();
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
else
    R[t] = DBGDTRRXint;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
        AArch32.TakeHypTrapException(0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        R[t] = DBGDTRRXint;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        R[t] = DBGDTRRXint;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
    else
        R[t] = DBGDTRRXint;
G8.3.18   DBGDTRTXext, Debug OS Lock Data Transfer Register, Transmit

The DBGDTRTXext characteristics are:

Purpose

Used for save/restore of DBGDTRTXint. It is a component of the Debug Communication Channel.

Configurations

AArch32 System register DBGDTRTXext bits [31:0] are architecturally mapped to AArch64 System register OSDTRTX_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGDTRTXext are UNDEFINED.

Attributes

DBGDTRTXext is a 32-bit register.

Field descriptions

Bits [31:0]

Return DTRTX without side-effect.

Reads of this register return the value in DTRTX and do not change TXfull.

Writes of this register update the value in DTRTX and do not change TXfull.

For the full behavior of the Debug Communications Channel, see Chapter H4 The Debug Communication Channel and Instruction Transfer Register.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRTXext

Arm deprecates reads and writes of DBGDTRTXext through the System register interface when the OS Lock is unlocked, DBGOSLSR.OSLK == 0.

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{c\}\{q\} <\text{coproc}, \ (#)\langle opc1\rangle, <Rt>, <\text{CRn}>, <\text{CRm}\rangle\{, \ (#)\langle opc2\rangle\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif Halted() && ConstrainUnpredictableBool(\text{Unpredictable\_IGNORETRAPINDEBUG}) then
    R[t] = DBGDTRTXext;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && \text{EDSCR.\_SDD} == '1' && boolean IMPLEMENTATION\_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        UNDEFINED;
    elsif Halted() && HaveEL(EL3) && \text{EDSCR.\_SDD} == '1' && boolean IMPLEMENTATION\_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
        UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
    UNDEFINED;
elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCU == '1' then
    AArch32.TakeHypTrapException(0x05);
elsif EL2Enabled() && ELUsingAArch32(EL2) && EDSCR.TDCC != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    end if;
else
    R[t] = DBGDTRXext;
end else;
elsif PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
    AArch32.TakeMonitorTrapException();
else
    R[t] = DBGDTRXext;
end if;

**MCR{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>)}**

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0011</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif Halted() & HaltedUnpredictable(UNpredictable_IGNORERETINDEBUG) then
DBGDTRText = R[t];
elsif PSTATE.EL == EL1 then
if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDCC == '1' then
UNDEFINED;
elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDCC == '1' then
UNDEFINED;
elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDCC == '1' then
UNDEFINED;
elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDCC == '1' then
UNDEFINED;
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TDCC == '1' then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & SDCCR.TDCC == '1' then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch32.TakeMonitorTrapException();
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TDCC == '1' then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL3, 0x05);
else
DBGDTRText = R[t];
elsif PSTATE.EL == EL2 then
if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDCC == '1' then
UNDEFINED;
elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDCC == '1' then
UNDEFINED;
elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDCC == '1' then
UNDEFINED;
elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDCC == '1' then
UNDEFINED;
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & EDSCR.SDD == '1' then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & SDCCR.TDCC == '1' then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch32.TakeMonitorTrapException();

elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
  else
    DBGDTRXext = R[t];
elsif PSTATE_EL == EL3 then
  if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
    AArch32.TakeMonitorTrapException();
  else
    DBGDTRXext = R[t];
G8.3.19  DBGDTRTXint, Debug Data Transfer Register, Transmit

The DBGDTRTXint characteristics are:

Purpose

Transfers data from the PE to an external debugger. For example, it is used by a debug target to transfer data to the debugger. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communication Channel.

Configurations

AArch32 System register DBGDTRTXint bits [31:0] are architecturally mapped to AArch64 System register DBGDTRTX_EL0[31:0].

AArch32 System register DBGDTRTXint bits [31:0] are architecturally mapped to External register DBGDTRTX_EL0[31:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to DBGDTRTXint are UNDEFINED.

Attributes

DBGDTRTXint is a 32-bit register.

Field descriptions

Bits [31:0]

Return DTRTX.

Writes to this register:

- If TXfull is set to 1, set DTRTX to UNKNOWN.
- If TXfull is set to 0, update the value in DTRTX.

After the write, TXfull is set to 1.

For the full behavior of the Debug Communications Channel, see Chapter H4 The Debug Communication Channel and Instruction Transfer Register.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRTXint

Data can be loaded from memory into this register using LDC (immediate) and LDC (literal).

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MCR\{<c>\}{<q}> <coproc>, \{#<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0101</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if Halted() then
    DBGDTRTXint = R[t];
else if PSTATE_EL == EL0 then
    if !ELUsingAArch32(EL1) && MDSCR_EL1.TDCC == '1' then

if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
else
    AArch64.AArch32SystemAccessTrap(EL1, 0x05);
elif ELUsingAArch32(EL1) && DBGDSCRext.UDDCdis == '1' then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
    AArch32.TakeHypTrapException(0x00);
else
    UNDEFINED;
elif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
    AArch32.TakeHypTrapException(0x05);
elif EL2Enabled() && ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00') then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elif EL2Enabled() && ELUsingAArch32(EL2) && (HCR.TGE == '1' || HDCR.<TDE,TDA> != '00') then
    AArch32.TakeHypTrapException(0x05);
elif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TDCC == '1' then
    AArch32.TakeMonitorTrapException();
elif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
else
    DBGDTRTXint = R[t];
elifs PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elifs PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elifs PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR.TDCC == '1' then
        AArch32.TakeMonitorTrapException();
else
    DBGDTRTXint = R[t];
G8.3.20  DBGOSDLR, Debug OS Double Lock Register

The DBGOSDLR characteristics are:

**Purpose**

Locks out the external debug interface.

**Configurations**

AArch32 System register DBGOSDLR bits [31:0] are architecturally mapped to AArch64 System register OSDLR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGOSDLR are UNDEFINED.

**Attributes**

DBGOSDLR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:1]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, RES0.</td>
<td></td>
</tr>
</tbody>
</table>

**DLK, bit [0]**

*When FEAT_DoubleLock is implemented:*

OS Double Lock control bit.

- 0b0  OS Double Lock unlocked.
- 0b1  OS Double Lock locked, if DBGPRCR.CORENPDRQ (Core no powerdown request) bit is set to 0 and the PE is in Non-debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

*Otherwise:*

Reserved, RAZ/WI.

**Accessing DBGOSDLR**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}<q> <\text{coproc}, \{#<opc1>, <Rt>, <CRn>, <CRm>\}, \{#<opc2>\}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0011</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' && (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by MDCR_EL3.TDOSA")
then UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' &&
    (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by MDCR_EL2.TDOSA")
    then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDOSA> != '00' &&
    (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by HDCR.TDOSA") then
        AArch32.TakeHypTrapException(0x05);
    elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' &&
    (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by MDCR_EL3.TDOSA")
    then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        else
            R[t] = DBGOSDLR;
        endif
    elseif PSTATE.EL == EL2 then
    endif
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1" && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' &&
    (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by MDCR_EL3.TDOSA")
    then
        UNDEFINED;
    elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' &&
    (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by MDCR_EL3.TDOSA")
    then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        else
            R[t] = DBGOSDLR;
        endif
    elseif PSTATE.EL == EL3 then
    endif
    if PSTATE.EL == EL0 then
    endif
    elseif PSTATE.EL == EL1 then
    endif
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1" && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' &&
    (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by MDCR_EL3.TDOSA")
    then
        UNDEFINED;
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' &&
    (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by MDCR_EL2.TDOSA")
    then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDOSA> != '00' &&
    (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by HDCR.TDOSA") then
        AArch32.TakeHypTrapException(0x05);
    elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' &&
    (IsFeatureImplemented(FEAT[DoubleLock] || boolean IMPLEMENTATION_DEFINED "Trapped by MDCR_EL3.TDOSA")
    then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        end
    
MCR<coproc><crn> <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>, (#)<opc2>

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0011</td>
<td>0b100</td>
</tr>
</tbody>
</table>
else
    DBGOSDLR = R[t];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' &&
        boolean IMPLEMENTATION_DEFINED "EL3 trap priority"
        when SDD == '1'" && !ELUsingAArch32(EL3) &&
        MDCR_EL3.TDOSA == '1' &&
        (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by MDCR_EL3.TDOSA")
        then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) &&
          MDCR_EL3.TDOSA == '1' &&
          (IsFeatureImplemented(FEAT_DoubleLock) || boolean IMPLEMENTATION_DEFINED "Trapped by MDCR_EL3.TDOSA")
          then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        end
    else
        DBGOSDLR = R[t];
elsif PSTATE.EL == EL3 then
    DBGOSDLR = R[t];
G8.3.21  DBGOSECCR, Debug OS Lock Exception Catch Control Register

The DBGOSECCR characteristics are:

**Purpose**

Provides a mechanism for an operating system to access the contents of EDECCR that are otherwise invisible to software, so it can save/restore the contents of EDECCR over powerdown on behalf of the external debugger.

**Configurations**

AArch32 System register DBGOSECCR bits [31:0] are architecturally mapped to AArch64 System register OSECCR_EL1[31:0].

AArch32 System register DBGOSECCR bits [31:0] are architecturally mapped to External register EDECCR[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGOSECCR are UNDEFINED.

If DBGOSLSR.OSLK == 0 then DBGOSECCR returns an UNKNOWN value on reads and ignores writes.

**Attributes**

DBGOSECCR is a 32-bit register.

**Field descriptions**

When DBGOSLSR.OSLK == 1:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDECCR</td>
<td>Bits [31:0] Used for save/restore to EDECCR over powerdown. Reads or writes to this field are indirect accesses to EDECCR.</td>
</tr>
</tbody>
</table>

**Accessing DBGOSECCR**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC\{<c>\}{<q>} \{<opc1>, \{#\}<opc2>, <Rt>, <CRn>, <CRm}\{, \{#\}<opc2>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0110</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && ELUsingAArch32(EL2) && MDSE_EL2,<TDE,TDA> != '00' then
    AArch32.TakeHypTrapException(0x05);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
    AArch32.TakeHypTrapException(0x05);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDSE_EL3,TDA == '1' then
if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elsif DBGOSLSR.OSLK == '0' then
    R[t] = bits(32) UNKNOWN;
else
    R[t] = DBGOSECCR;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        elsif DBGOSLSR.OSLK == '0' then
            R[t] = bits(32) UNKNOWN;
        else
            R[t] = DBGOSECCR;
    elsif PSTATE.EL == EL3 then
        if DBGOSLSR.OSLK == '0' then
            R[t] = bits(32) UNKNOWN;
        else
            R[t] = DBGOSECCR;
    
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0110</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        elsif DBGOSLSR.OSLK == '0' then
            return;
        else
            DBGOSECCR = R[t];
    elsif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            UNDEFINED;
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            elsif DBGOSLSR.OSLK == '0' then
                return;
            else
                AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        elsif DBGOSLSR.OSLK == '0' then
            return;
        else

ARM DDI 0487I.a  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.  G8-9903
ID081822  Non-Confidential
DBOSECCR = R[t];
elsif PSTATE_EL == EL3 then
    if DBGOSLSR.OSLK == '0' then
        return;
    else
        DBOSECCR = R[t];
    end if;
else
    DBOSECCR = R[t];
end if;
**G8.3.22** DBGOSLAR, Debug OS Lock Access Register

The DBGOSLAR characteristics are:

**Purpose**
Provided a lock for the debug registers. The OS Lock also disables some debug exceptions and debug events.

**Configurations**
This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGOSLAR are UNDEFINED.

The OS Lock can also be locked or unlocked using the AArch64 System register OSLAR_EL1 and External register OSLAR_EL1.

**Attributes**
DBGOSLAR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>OSLA, bits [31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS Lock Access. Writing the value 0x5ACE55 to the DBGOSLAR sets the OS Lock to 1. Writing any other value sets the OS Lock to 0. Use DBGOSLSR.OSLK to check the current status of the lock.</td>
</tr>
</tbody>
</table>

**Accessing DBGOSLAR**

Accesses to this register use the following encodings in the System register encoding space:

**MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>**

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b0000</td>
<td>0b0001</td>
<td>0b0000</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1’" && \!ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
    UNDEFINED;
elif EL2Enabled() && \!ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elif EL2Enabled() && EDSCR.SDD == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elif EL2Enabled() && MDCR_EL3.TDOSA == '1' then
    UNDEFINED;
elif Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
elif HaveEL(EL3) && EDSCR.SDD == '1' then
    UNDEFINED;
elif MDCR_EL3.TDOSA == '1' then
    UNDEFINED;
else
    DBGOSLAR = R[t];
elif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1’" && \!ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
    UNDEFINED;
elif Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
elif HaveEL(EL3) && EDSCR.SDD == '1' then
    UNDEFINED;
elif MDCR_EL3.TDOSA == '1' then
    UNDEFINED;
else
    DBGOSLAR = R[t];
when SDD == '1' && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
  UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
  else
    DBGOSLR = R[t];
elsif PSTATE.EL == EL3 then
  DBGOSLR = R[t];
### G8.3.23 DBGOSLSR, Debug OS Lock Status Register

The DBGOSLSR characteristics are:

**Purpose**

Provides status information for the OS Lock.

**Configurations**

AArch32 System register DBGOSLSR bits [31:0] are architecturally mapped to AArch64 System register OSLSR_EL1[31:0].  

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGOSLSR are UNDEFINED.  

The OS Lock status is also visible in the external debug interface through EDPRSR.

**Attributes**

DBGOSLSR is a 32-bit register.

**Field descriptions**

- **Bits [31:4]**
  - Reserved, RES0.
- **OSLM, bits [3, 0]**
  - OS Lock model implemented. Identifies the form of OS save and restore mechanism implemented.
    - 0b00: OS Lock not implemented.
    - 0b10: OS Lock implemented.
  - All other values are reserved. In an Armv8 implementation the value 0b00 is not permitted.  
  - The OSLM field is split as follows:
    - OSLM[1] is DBGOSLSR[3].
    - OSLM[0] is DBGOSLSR[0].
- **nTT, bit [2]**
  - Not 32-bit access. This bit is always RAZ. It indicates that a 32-bit access is needed to write the key to the OS Lock Access Register.
- **OSLK, bit [1]**
  - OS Lock Status.
    - 0b0: OS Lock unlocked.
    - 0b1: OS Lock locked.
  - The OS Lock is locked and unlocked by writing to the OS Lock Access Register.
  - The reset behavior of this field is:
    - On a Cold reset, this field resets to 1.

### Accessing DBGOSLSR

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDOSA> != '00' then
    AArch32.TakeHypTrapException(0x05);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    end
  else
    R[t] = DBGOSLSR;
  end
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    end
  else
    R[t] = DBGOSLSR;
  end
elsif PSTATE_EL == EL3 then
  R[t] = DBGOSLSR;
G8.3.24  DBGPRCR, Debug Power Control Register

The DBGPRCR characteristics are:

**Purpose**

Controls behavior of the PE on powerdown request.

**Configurations**

AArch32 System register DBGPRCR bits [31:0] are architecturally mapped to AArch64 System register DBGPRCR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGPRCR are UNDEFINED.

Bit [0] of this register is mapped to EDPRCR.CORENPDRQ, bit [0] of the external view of this register.

The other bits in these registers are not mapped to each other.

**Attributes**

DBGPRCR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>30</td>
<td>CORENPDRQ, bit [0]</td>
</tr>
</tbody>
</table>

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

0b0  If the system responds to a powerdown request, it powers down Core power domain.

0b1  If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

In an implementation that includes the recommended external debug interface, this bit drives the DBGNOPWRDWN signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the Cold reset value on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see Core power domain power states on page H6-10336.

--- Note ---

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:

- On a Cold reset, if the powerup request is implemented and the powerup request has been asserted, this field is set to an IMPLEMENTATION DEFINED choice of 0 or 1. If the powerup request is not asserted, this field is set to 0.
Otherwise:

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

0b0 If the system responds to a powerdown request, it powers down Core power domain.

0b1 If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

In an implementation that includes the recommended external debug interface, this bit drives the DBGNOPWRDWN signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the value of EDPSCR.COREPURQ on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see Core power domain power states on page H6-10336.

——— Note ———

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:

- On a Cold reset, this field resets to the value in EDPSCR.COREPURQ.

Accessing DBGPRCR

Accesses to this register use the following encodings in the System register encoding space:

$$\text{MRC}\{<c>\}{<q>}\text{<coproc>}{, \{#\}<opc1>}{, \langle Rt \rangle}{, \langle CRn \rangle}{, \langle CRm \rangle}{, \{#\}<opc2>}$$

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0100</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsf PSTATE.EL == EL1 then
if Halted() && HaveEl(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
UNDEFINED;
elsf EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' then
AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsf EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDOSA> != '00' then
AArch32.TakeHypTrapException(0x05);
elsf HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
elsf
AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elsf
else
R[t] = DBGPRCR;
elsf PSTATE.EL == EL2 then
if Halted() && HaveEl(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
UNDEFINED;
elsf HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDOSA == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
elsf
AArch64.AArch32SystemAccessTrap(EL3, 0x05);
else
    R[t] = DBGPRCR;
elsif PSTATE.EL == EL3 then
    R[t] = DBGPRCR;

MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0100</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDOSA == '1' then
        UNDEFINED;
    elsif EL2Enabled() & !ELUsingAArch32(EL2) & MDCR_EL2.<TDE,TDOSA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HDCR.<TDE,TDOSA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) & ELUsingAArch32(EL3) & MDCR_EL3.TDOSA == '1' then
        if Halted() & EZRCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        end
    elsif PSTATE.EL == EL2 then
        if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDOSA == '1' then
            UNDEFINED;
        elsif HaveEL(EL3) & ELUsingAArch32(EL3) & MDCR_EL3.TDOSA == '1' then
            if Halted() & EZRCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.AArch32SystemAccessTrap(EL3, 0x05);
            end
        else
            DBGPRCR = R[t];
        end
    elsif PSTATE.EL == EL3 then
        DBGPRCR = R[t];
### G8.3.25 DBGVCR, Debug Vector Catch Register

The DBGVCR characteristics are:

**Purpose**

Controls Vector Catch debug events.

**Configurations**

AArch32 System register DBGVCR bits [31:0] are architecturally mapped to AArch64 System register DBGVCR32_EL2[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGVCR are **UNDEFINED**.

This register is required in all implementations.

**Attributes**

DBGVCR is a 32-bit register.

**Field descriptions**

*When EL3 is implemented and EL3 is using AArch32:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value Offset</th>
<th>Reset Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>NSF, FIQ vector catch enable in Non-secure state</td>
<td>0x1C</td>
<td>On a Warm reset, this field resets to an architecturally <strong>UNKNOWN</strong> value.</td>
</tr>
<tr>
<td>30</td>
<td>NSI, IRQ vector catch enable in Non-secure state</td>
<td>0x18</td>
<td>On a Warm reset, this field resets to an architecturally <strong>UNKNOWN</strong> value.</td>
</tr>
<tr>
<td>29</td>
<td>Reserved, RES0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>NSD, Data Abort exception vector catch enable in Non-secure state</td>
<td>0x10</td>
<td>On a Warm reset, this field resets to an architecturally <strong>UNKNOWN</strong> value.</td>
</tr>
<tr>
<td>27</td>
<td>NSP, Prefetch Abort vector catch enable in Non-secure state</td>
<td>0x0C</td>
<td></td>
</tr>
</tbody>
</table>
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**NSS, bit [26]**
Supervisor Call (SVC) vector catch enable in Non-secure state.
The exception vector offset is **0x08**.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**NSU, bit [25]**
Undefined Instruction vector catch enable in Non-secure state.
The exception vector offset is **0x04**.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Bits [24:16]**
Reserved, **RES0**.

**MF, bit [15]**
FIQ vector catch enable in Monitor mode.
The exception vector offset is **0x1C**.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**MI, bit [14]**
IRQ vector catch enable in Monitor mode.
The exception vector offset is **0x18**.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Bit [13]**
Reserved, **RES0**.

**MD, bit [12]**
Data Abort exception vector catch enable in Monitor mode.
The exception vector offset is **0x10**.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**MP, bit [11]**
Prefetch Abort vector catch enable in Monitor mode.
The exception vector offset is **0x0C**.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**MS, bit [10]**
Secure Monitor Call (SMC) vector catch enable in Monitor mode.
The exception vector offset is **0x08**.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
Bits [9:8]

Reserved, RES0.

SF, bit [7]

FIQ vector catch enable in Secure state.
The exception vector offset is 0x1C.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

SI, bit [6]

IRQ vector catch enable in Secure state.
The exception vector offset is 0x18.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

SD, bit [4]

Data Abort exception vector catch enable in Secure state.
The exception vector offset is 0x10.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

SP, bit [3]

Prefetch Abort vector catch enable in Secure state.
The exception vector offset is 0x0C.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

SS, bit [2]

Supervisor Call (SVC) vector catch enable in Secure state.
The exception vector offset is 0x08.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

SU, bit [1]

Undefined Instruction vector catch enable in Secure state.
The exception vector offset is 0x04.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.
When EL3 is implemented and EL3 is using AArch64:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Exception Vector Offset</th>
<th>Reset Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>NSF, bit [31] – FIQ vector catch enable in Non-secure state</td>
<td>0x1C</td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>30</td>
<td>NSI, bit [30] – IRQ vector catch enable in Non-secure state</td>
<td>0x18</td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>29</td>
<td>Reserved, RES0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>NSD, bit [28] – Data Abort exception vector catch enable in Non-secure state</td>
<td>0x10</td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>27</td>
<td>NSP, bit [27] – Prefetch Abort vector catch enable in Non-secure state</td>
<td>0x0C</td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>26</td>
<td>NSS, bit [26] – Supervisor Call (SVC) vector catch enable in Non-secure state</td>
<td>0x08</td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>25</td>
<td>NSU, bit [25] – Undefined Instruction vector catch enable in Non-secure state</td>
<td>0x04</td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>24-8</td>
<td>Reserved, RES0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SF, bit [7]
FIQ vector catch enable in Secure state.
The exception vector offset is 0x1C.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

SI, bit [6]
IRQ vector catch enable in Secure state.
The exception vector offset is 0x18.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]
Reserved, RES0.

SD, bit [4]
Data Abort exception vector catch enable in Secure state.
The exception vector offset is 0x10.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

SP, bit [3]
Prefetch Abort vector catch enable in Secure state.
The exception vector offset is 0x0C.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

SS, bit [2]
Supervisor Call (SVC) vector catch enable in Secure state.
The exception vector offset is 0x08.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

SU, bit [1]
Undefined Instruction vector catch enable in Secure state.
The exception vector offset is 0x04.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]
Reserved, RES0.

When EL3 is not implemented:
Bits [31:8]

Reserved, RES0.

F, bit [7]

FIQ vector catch enable.
The exception vector offset is 0x1C.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [6]

IRQ vector catch enable.
The exception vector offset is 0x18.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

D, bit [4]

Data Abort exception vector catch enable.
The exception vector offset is 0x10.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

P, bit [3]

Prefetch Abort vector catch enable.
The exception vector offset 0x0C.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

S, bit [2]

Supervisor Call (SVC) vector catch enable.
The exception vector offset 0x08.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [1]

Undefined Instruction vector catch enable.
The exception vector offset 0x04.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

Accessing DBGVCR

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    UNDEFINED;
elseif EL2Enabled() & !ELUsingAArch32(EL2) & MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elseif EL2Enabled() & ELUsingAArch32(EL2) & HDCR.<TDE,TDA> != '00' then
    AArch32.TakeHypTrapException(0x05);
elseif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
      R[t] = DBGVCR;
  else
    if PSTATE.EL == EL2 then
      if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
        UNDEFINED;
elseif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
        if Halted() & EDSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        else
          R[t] = DBGVCR;
      else
        if PSTATE.EL == EL3 then
          R[t] = DBGVCR;
        else
          DBGVCR = R[t];
      else
        if PSTATE.EL == EL0 then
          UNDEFINED;
elseif PSTATE.EL == EL1 then
          if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
            UNDEFINED;
elseif EL2Enabled() & !ELUsingAArch32(EL2) & MDCR_EL2.<TDE,TDA> != '00' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elseif EL2Enabled() & ELUsingAArch32(EL2) & HDCR.<TDE,TDA> != '00' then
            AArch32.TakeHypTrapException(0x05);
elseif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
            if Halted() & EDSCR.SDD == '1' then
              UNDEFINED;
            else
              AArch64.AArch32SystemAccessTrap(EL3, 0x05);
            else
              DBGVCR = R[t];
          else
            if PSTATE.EL == EL2 then
              DBGVCR = R[t];
            else
              if PSTATE.EL == EL3 then
                DBGVCR = R[t];
              else
                DBGVCR = R[t];
            else
              DBGVCR = R[t];
            else
              DBGVCR = R[t];
          else
            DBGVCR = R[t];
        else
          DBGVCR = R[t];
      else
        DBGVCR = R[t];
    else
      DBGVCR = R[t];
  else
    DBGVCR = R[t];
else
  DBGVCR = R[t];
else
  DBGVCR = R[t];

if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
   UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
   if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
   else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
   else
      DBGVCR = R[t];
else if PSTATE.EL == EL3 then
   DBGVCR = R[t];
G8.3.26    DBGWCR<n>, Debug Watchpoint Control Registers, n = 0 - 15

The DBGWCR<n> characteristics are:

**Purpose**

Holds control information for a watchpoint. Forms watchpoint n together with value register DBGWVR<n>.

**Configurations**

AArch32 System register DBGWCR<n> bits [31:0] are architecturally mapped to AArch64 System register DBGWCR<n>_EL1[31:0].

AArch32 System register DBGWCR<n> bits [31:0] are architecturally mapped to External register DBGWCR<n>_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGWCR<n> are UNDEFINED.

If watchpoint n is not implemented then accesses to this register are UNDEFINED.

**Attributes**

DBGWCR<n> is a 32-bit register.

**Field descriptions**

When the E field is zero, all the other fields in the register are ignored.

**Bits [31:29]**

Reserved, RES0.

**MASK, bits [28:24]**

Address mask. Only objects up to 2GB can be watched using a single mask.

- 0b00000  No mask.
- 0b00001  Reserved.
- 0b00010  Reserved.

If programmed with a reserved value, a watchpoint must behave as if either:

- MASK has been programmed with a defined value, which might be 0 (no mask), other than for a direct read of DBGWCRn_EL1.
- The watchpoint is disabled.

Software must not rely on this property because the behavior of reserved values might change in a future revision of the architecture.

Other values mask the corresponding number of address bits, from 0b000011 masking 3 address bits (0x00000007 mask for address) to 0b11111111 masking 31 address bits (0x7FFFFFFF mask for address).

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Bits [23:21]**

Reserved, RES0.
WT, bit [20]

Watchpoint type. Possible values are:

0b0  Unlinked data address match.
0b1  Linked data address match.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked breakpoint number. For Linked data address watchpoints, this specifies the index of the Context-matching breakpoint linked to.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the HMC and PAC fields.

For more information, see Execution conditions for which a watchpoint generates Watchpoint exceptions on page G2-9081, and Reserved DBGWCR<n>.{SSC, HMC, PAC} values on page G2-9088.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and PAC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see Execution conditions for which a watchpoint generates Watchpoint exceptions on page G2-9081.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

BAS, bits [12:5]

Byte address select. Each bit of this field selects whether a byte from within the word or double-word addressed by DBGWVR<n> is being watched.

<table>
<thead>
<tr>
<th>BAS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0bxxxxxxx1</td>
<td>Match byte at DBGWVR&lt;n&gt;</td>
</tr>
<tr>
<td>0bxxxxxx1x</td>
<td>Match byte at DBGWVR&lt;n&gt;+1</td>
</tr>
<tr>
<td>0bxxxxx1xx</td>
<td>Match byte at DBGWVR&lt;n&gt;+2</td>
</tr>
<tr>
<td>0bxxxx1xxx</td>
<td>Match byte at DBGWVR&lt;n&gt;+3</td>
</tr>
</tbody>
</table>

In cases where DBGWVR<n> addresses a double-word:

<table>
<thead>
<tr>
<th>BAS</th>
<th>Description, if DBGWVR&lt;n&gt;[2] == 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0bxxx1xxxx</td>
<td>Match byte at DBGWVR&lt;n&gt;+4</td>
</tr>
</tbody>
</table>

The valid values for BAS are non-zero binary numbers all of whose set bits are contiguous. All other values are reserved and must not be used by software. See Reserved DBGWCR<\text{n}>.BAS values on page G2-9089.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

LSC, bits [4:3]

Load/store control. This field enables watchpoint matching on the type of access being made. Possible values of this field are:
0b01  Match instructions that load from a watchpointed address.
0b10  Match instructions that store to a watchpointed address.
0b11  Match instructions that load from or store to a watchpointed address.

All other values are reserved, but must behave as if the watchpoint is disabled. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

PAC, bits [2:1]

Privilege of access control. Determines the Exception level or levels at which a Watchpoint debug event for watchpoint \( \text{n} \) is generated. This field must be interpreted along with the SSC and HMC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see Execution conditions for which a watchpoint generates Watchpoint exceptions on page G2-9081.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Enable watchpoint \( \text{n} \). Possible values are:
0b0  Watchpoint disabled.
0b1  Watchpoint enabled.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

### Accessing DBGWCR<\text{n}>

Accesses to this register use the following encodings in the System register encoding space:

<table>
<thead>
<tr>
<th>BAS</th>
<th>Description, if DBGWVR&lt;\text{n}&gt;[2] == 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0bxx1xxxx</td>
<td>Match byte at DBGWVR&lt;\text{n}&gt;+5</td>
</tr>
<tr>
<td>0bx1xxxxxx</td>
<td>Match byte at DBGWVR&lt;\text{n}&gt;+6</td>
</tr>
<tr>
<td>0bx1xxxxxx</td>
<td>Match byte at DBGWVR&lt;\text{n}&gt;+7</td>
</tr>
</tbody>
</table>
MRC{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>)}}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0000</td>
<td>m[3:0]</td>
<td>0b111</td>
</tr>
</tbody>
</table>

integer m = UInt(CRm<3:0>);

if m >= NUM_WATCHPOINTS then
  UNDEFINED;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'"
    & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() & !ELUsingAArch32(EL2) & MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & MDCR_EL2.<TDE,TDA> != '00' then
    AArch32.TakeHypTrapException(0x05);
  elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    endif
  elsif DBGOSLSR.OSLK == '0' & HaltingAllowed() & EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
else
  R[t] = DBGWCR[m];
endif

MCR{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>)}}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0000</td>
<td>m[3:0]</td>
<td>0b111</td>
</tr>
</tbody>
</table>

integer m = UInt(CRm<3:0>);

if m >= NUM_WATCHPOINTS then
  UNDEFINED;
elsif PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'"
    & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif EL2Enabled() & !ELUsingAArch32(EL2) & MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & MDCR_EL2.<TDE,TDA> != '00' then
    AArch32.TakeHypTrapException(0x05);
  elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TDA == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    endif
  elsif DBGOSLSR.OSLK == '0' & HaltingAllowed() & EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
else
  R[t] = DBGWCR[m];
endif

MCR{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>)}}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0000</td>
<td>m[3:0]</td>
<td>0b111</td>
</tr>
</tbody>
</table>
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
    AArch32.TakeHypTrapException(0x05);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    end if
elsif DBGOSLRS.0SLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
else
    DBGWCR[m] = R[t];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        end if
    else
        DBGWCR[m] = R[t];
    end if
elsif PSTATE.EL == EL3 then
    if DBGOSLRS.0SLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWCR[m] = R[t];
    end if
else
    DBGWCR[m] = R[t];
else
    DBGWCR[m] = R[t];
G8.3.27 DBGWFAR, Debug Watchpoint Fault Address Register

The DBGWFAR characteristics are:

**Purpose**

Previously returned information about the address of the instruction that accessed a watchpointed address. Is now deprecated and RES0.

**Configurations**

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGWFAR are UNDEFINED.

**Attributes**

DBGWFAR is a 32-bit register.

**Field descriptions**

Bits [31:0]  
Reserved, RES0.

**Accessing DBGWFAR**

Accesses to this register use the following encodings in the System register encoding space:

\[\text{MRC}\{c\}\{q\} <\text{coproc}, \{#}\text{opc1}, <\text{Rt}, <\text{CRn}, <\text{CRm}{, \{#}\text{opc2}}\}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0000</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
ellif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
ellif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TDA != '00' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
ellif EL2Enabled() && EDSCR.EL2.TDA != '00' then
    HDCR.<TDE,TDA> != '00' then
    AArch32.TakeTrapException(0x80);
ellif HaveEL(EL3) && EDSCR.EL3.TDA == '1' then
    if Halted() && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
      UNDEFINED;
ellif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.EL3.TDA == '1' then
      UNDEFINED;
ellif PSTATE.EL == EL2 then
  if Halted() && !ELUsingAArch32(EL3) && EDSCR.EL3.TDA == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
ellif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.EL3.TDA == '1' then
    UNDEFINED;
```plaintext
MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b000</td>
<td>0b0110</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        else
            DBGWFAR = R[t];
    elsif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            UNDEFINED;
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.AArch32SystemAccessTrap(EL3, 0x05);
        else
            DBGWFAR = R[t];
    elsif PSTATE.EL == EL3 then
        DBGWFAR = R[t];
```
G8.3.28  **DBGWVR<n>, Debug Watchpoint Value Registers, n = 0 - 15**

The DBGWVR<n> characteristics are:

**Purpose**

Holds a data address value for use in watchpoint matching. Forms watchpoint n together with control register DBGWCR<n>.

**Configurations**

AArch32 System register DBGWVR<n> bits [31:0] are architecturally mapped to AArch64 System register DBGWVR<n>_EL1[31:0].

AArch32 System register DBGWVR<n> bits [31:0] are architecturally mapped to External register DBGWVR<n>_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGWVR<n> are UNDEFINED.

If watchpoint n is not implemented then accesses to this register are UNDEFINED.

**Attributes**

DBGWVR<n> is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA</td>
<td>RES0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**VA, bits [31:2]**

Bits[31:2] of the address value for comparison.


The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Bits [1:0]**

Reserved, RES0.

**Accessing DBGWVR<n>**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC\{<c>\}{<q>} <coproc>, \{#<opc1>, <Rt>, <CRn>, <CRm>, \{#<opc2}\}
\]

```
coproc   opc1  CRn   CRm   opc2
0b1110   0b000 0b0000 m[3:0]  0b110
```

integer m = UInt(CRm<3:0>);

if m >= NUM_WATCHPOINTS then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & & MDCR_EL3.TDA == '1' then
# AArch32 System Register Descriptions

## G8.3 Debug registers

### MCR\{<c>\}{<q>}, \{#<opc1>, <Rt>, <CRn>, <CRm>{, {#<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0000</td>
<td>m[3:0]</td>
<td>0b110</td>
</tr>
</tbody>
</table>

integer m = UInt(CRm<3:0>);
if m >= NUM_WATCHPOINTS then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
  AArch32.TakeHypTrapException(0x05);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elsif DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
  Halt(DebugHaltSoftwareAccess);
else
  R[t] = DBGWVR[m];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elsif DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
  Halt(DebugHaltSoftwareAccess);
else
  R[t] = DBGWVR[m];
elsif PSTATE.EL == EL3 then
  if DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHaltSoftwareAccess);
else
  R[t] = DBGWVR[m];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
  AArch32.TakeHypTrapException(0x05);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
elsif DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
  Halt(DebugHaltSoftwareAccess);
else
  DBGWVR[m] = R[t];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1' && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
  UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
  elsif DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
  else
    DBGWVR[m] = R[t];
  end if
elsif PSTATE.EL == EL3 then
  if DBGOSLSR.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
  else
    DBGWVR[m] = R[t];
  end if
### G8.3.29 DLR, Debug Link Register

The DLR characteristics are:

**Purpose**

In Debug state, holds the address to restart from.

**Configurations**

AArch32 System register DLR bits [31:0] are architecturally mapped to AArch64 System register `DLR_EL0[31:0]`.

This register is present only when AArch32 is supported. Otherwise, direct accesses to DLR are UNDEFINED.

**Attributes**

DLR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Restart address</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:0]**

Restart address.

**Accessing DLR**

Accesses to this register use the following encodings in the System register encoding space:

**MRC\{<c>\}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2}>**

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0101</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !Halted() then
  UNDEFINED;
else
  R[t] = DLR;

**MCR\{<c>\}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2}>**

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b011</td>
<td>0b0100</td>
<td>0b0101</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if !Halted() then
  UNDEFINED;
else
  DLR = R[t];
G8.3.30  **DSPSR, Debug Saved Program Status Register**

The DSPSR characteristics are:

**Purpose**

Holds the saved process state for Debug state. On entering Debug state, PSTATE information is written to this register. On exiting Debug state, values are copied from this register to PSTATE.

**Configurations**

AArch32 System register DSPSR bits [31:0] are architecturally mapped to AArch64 System register DSPSR_EL0[31:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to DSPSR are UNDEFINED.

**Attributes**

DSPSR is a 32-bit register.

**Field descriptions**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
```

**N, bit [31]**

Negative Condition flag. Set to the value of PSTATE.N on entering Debug state, and copied to PSTATE.N on exiting Debug state.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Z, bit [30]**

Zero Condition flag. Set to the value of PSTATE.Z on entering Debug state, and copied to PSTATE.Z on exiting Debug state.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**C, bit [29]**

Carry Condition flag. Set to the value of PSTATE.C on entering Debug state, and copied to PSTATE.C on exiting Debug state.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**V, bit [28]**

Overflow Condition flag. Set to the value of PSTATE.V on entering Debug state, and copied to PSTATE.V on exiting Debug state.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Q, bit [27]**

Overflow or saturation flag. Set to the value of PSTATE.Q on entering Debug state, and copied to PSTATE.Q on exiting Debug state.
The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IT, bits [15:10, 26:25]**

If-Then. Set to the value of PSTATE.IT on entering Debug state, and copied to PSTATE.IT on exiting Debug state.

DSPSR.IT must contain a value that is valid for the instruction being returned to.

The IT field is split as follows:

• IT[1:0] is DSPSR[26:25].
• IT[7:2] is DSPSR[15:10].

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**DIT, bit [24]**

*When FEAT_DIT is implemented:*

Data Independent Timing. Set to the value of PSTATE.DIT on entering Debug state, and copied to PSTATE.DIT on exiting Debug state.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

**SSBS, bit [23]**

*When FEAT_SSBS is implemented:*

Speculative Store Bypass. Set to the value of PSTATE.SSBS on entering Debug state, and copied to PSTATE.SSBS on exiting Debug state.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

**PAN, bit [22]**

*When FEAT_PAN is implemented:*

Privileged Access Never. Set to the value of PSTATE.PAN on entering Debug state, and copied to PSTATE.PAN on exiting Debug state.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

**SS, bit [21]**

Software Step. Set to the value of PSTATE.SS on entering Debug state, and conditionally copied to PSTATE.SS on exiting Debug state.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

**IL, bit [20]**

Illegal Execution state. Set to the value of PSTATE.IL on entering Debug state, and copied to PSTATE.IL on exiting Debug state.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**GE, bits [19:16]**

Greater than or Equal flags. Set to the value of PSTATE.GE on entering Debug state, and copied to PSTATE.GE on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**E, bit [9]**

Endianness. Set to the value of PSTATE.E on entering Debug state, and copied to PSTATE.E on exiting Debug state.

If the implementation does not support big-endian operation, DSPSR.E is RES0. If the implementation does not support little-endian operation, DSPSR.E is RES1. On exiting Debug state, if the implementation does not support big-endian operation at the Exception level being returned to, DSPSR.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, DSPSR.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**A, bit [8]**

SError interrupt mask. Set to the value of PSTATE.A on entering Debug state, and copied to PSTATE.A on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**I, bit [7]**

IRQ interrupt mask. Set to the value of PSTATE.I on entering Debug state, and copied to PSTATE.I on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**F, bit [6]**

FIQ interrupt mask. Set to the value of PSTATE.F on entering Debug state, and copied to PSTATE.F on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**T, bit [5]**

T32 Instruction set state. Set to the value of PSTATE.T on entering Debug state, and copied to PSTATE.T on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**M[4:0], bits [4:0]**

Mode. Set to the value of PSTATE.M[4:0] on entering Debug state, and copied to PSTATE.M[4:0] on exiting Debug state.

- 0b10000 User
- 0b10001 FIQ
- 0b10010 IRQ
- 0b10011 Supervisor
- 0b10110 Monitor
Other values are reserved. If DSPSR.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, exiting Debug state is an illegal return event, as described in *Illegal return events from AArch32 state* on page G1-8952.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing DSPSR

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} \text{ <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>}}
\]

```
coproc  opc1  CRn  CRm  opc2
0b1111  0b011  0b0100  0b0101  0b000
```

if !Halted() then
    UNDEFINED;
else
    \( R[t] = \text{DSPSR}; \)

\[
\text{MCR}\{<c>\}{<q>} \text{ <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>}}
\]

```
coproc  opc1  CRn  CRm  opc2
0b1111  0b011  0b0100  0b0101  0b000
```

if !Halted() then
    UNDEFINED;
else
    \( \text{DSPSR} = R[t]; \)
**G8.3.31 HDCR, Hyp Debug Control Register**

The HDCR characteristics are:

**Purpose**

Controls the trapping to Hyp mode of Non-secure accesses, at EL1 or lower, to functions provided by the debug and trace architectures and the Performance Monitors Extension.

**Configurations**

AArch32 System register HDCR bits [31:0] are architecturally mapped to AArch64 System register MDCR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to HDCR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3, and other than for a direct read of the register, the PE behaves as if HDCR.HPMN \(=\) PMCR.N.

**Attributes**

HDCR is a 32-bit register.

**Field descriptions**

Bits [31:30]

Reserved, RES0.

HPMFZO, bit [29]

*When FEAT_PMUv3p7 is implemented:*

Hyp Performance Monitors Freeze-on-overflow. Stop event counters on overflow.

0b0  Do not freeze on overflow.

0b1  Event counters do not count when PMOVSR[(PMCR.N-1):HDCR.HPMN] is nonzero.

If HDCR.HPMN is less than PMCR.N, this field affects the operation of event counters in the range [HDCR.HPMN .. (PMCR.N-1)].

This field does not affect the operation of other event counters and PMCCNTR.

The operation of this field applies even when EL2 is disabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

MTPME, bit [28]

*When FEAT_MTPMU is implemented and EL3 is not implemented:*

Multi-threaded PMU Enable. Enables use of the PMEVTYPE\(<n>\).MT bits.

0b0  FEAT_MTPMU is disabled. The Effective value of PMEVTYPE\(<n>\).MT is zero.

0b1  PMEVTYPE\(<n>\).MT bits not affected by this bit.
If FEAT_MTPMU is disabled for any other PE in the system that has the same level 1 Affinity as the PE, it is IMPLEMENTATION DEFINED whether the PE behaves as if this bit is 0b0.

The reset behavior of this field is:

- **On a Cold reset:**
  - When the PE resets into EL2 or EL3, this field resets to 1.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**TDCC, bit [27]**

When FEAT_FGT is implemented:

Trap DCC. Traps use of the Debug Comms Channel at EL1 and EL0 to EL2.

0b0  This control does not cause any register accesses to be trapped.

0b1  If EL2 is implemented and enabled in the current Security state, accesses to the DCC registers at EL1 and EL0 generate a Hyp Trap exception, unless the access also generates a higher priority exception.

Traps on the DCC data transfer registers are ignored when the PE is in Debug state.

The DCC registers trapped by this control are:

- DBGDTRRXext, DBGDTRRXtext, DBGDSRCRint, DBGDCCINT, and, when the PE is in Non-debug state, DBGDTRRXint and DBGDTRTXint.

The traps are reported with EC syndrome value:

- 0x05 for trapped MRC and MCR accesses with coproc == 0b1110.
- 0x06 for trapped LDC to DBGDTRRXint and STC from DBGDTRRXint.

When the PE is in Debug state, HDCR.TDCC does not trap any accesses to:

- DBGDTRRXint and DBGDTRTXint.

The reset behavior of this field is:

- **On a Warm reset:**
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**HLP, bit [26]**

When FEAT_PMUv3p5 is implemented:

Hypervisor Long event counter enable. Determines when unsigned overflow is recorded by an event counter overflow bit.

0b0  Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<ν>[31:0].

0b1  Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<ν>[63:0].

If the highest implemented Exception level is using AArch32, it is IMPLEMENTATION DEFINED whether this bit is read/write or RAZ/WI.

If HDCR.HPMN is less than PMCR.N, this bit affects the operation of event counters in the range [HDCR.HPMN..(PMCR.N-1)].

This field does not affect the operation of other event counters.

The operation of this field applies even when EL2 is disabled in the current Security state.
--- Note ---


The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**Bits [25:24]**
Reserved, RES0.

**HCCD, bit [23]**

*When FEAT_PMUv3p5 is implemented:*

Hypervisor Cycle Counter Disable. Prohibits PMCCNTR from counting at EL2.

- 0b0 Cycle counting by PMCCNTR is not affected by this mechanism.
- 0b1 Cycle counting by PMCCNTR is prohibited at EL2.

This field does not affect the CPU_CYCLES event or any other event that counts cycles.

The reset behavior of this field is:
- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**Bits [22:20]**
Reserved, RES0.

**TTRF, bit [19]**

*When FEAT_TRF is implemented:*

Traps use of the Trace Filter Control registers at EL1 to EL2.

- 0b0 Accesses to TRFCR at EL1 are not affected by this control bit.
- 0b1 Accesses to TRFCR at EL1 generate a Hyp Trap exception.

The reset behavior of this field is:
- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**Bit [18]**
Reserved, RES0.

**HPMD, bit [17]**

*When FEAT_PMUv3p1 is implemented and FEAT_Debugv8p2 is implemented:*

Guest Performance Monitors Disable. Controls event counting by some event counters at EL2.

- 0b0 Event counting and PMCCNTR are not affected by this mechanism.
Event counting by some event counters is prohibited in Hyp mode. If PMCR.DP is 1, PMCCNTR is disabled in Hyp mode. Otherwise, PMCCNTR is not affected by this mechanism.

If HDCR.HPMN is not 0, this field affects the operation of event counters in the range \([0..(HDCR.HPMN-1)]\).

This field does not affect the operation of other event counters.

If PMCR.DP is 1, this field affects PMCCNTR.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**When FEAT_PMUv3p1 is implemented:**

Guest Performance Monitors Disable. Controls event counting by some event counters at EL2.

- Event counting and PMCCNTR are not affected by this mechanism.
- If ExternalSecureNoninvasiveDebugEnabled() is FALSE, event counting by some event counters is prohibited in Hyp mode, and if PMCR.DP is 1, PMCCNTR is disabled in Hyp mode.
- If ExternalSecureNoninvasiveDebugEnabled() is TRUE, this field does not affect the event counters and does not affect PMCCNTR.

Otherwise:

- If HDCR.HPMN is not 0, this field affects the operation of event counters in the range \([0..(HDCR.HPMN-1)]\).
- This field does not affect the operation of other event counters.
- If PMCR.DP is 1, this field affects PMCCNTR.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bits [16:12]**

Reserved, RES0.

**TDRA, bit [11]**

Trap Debug ROM Address register access. Traps Non-secure EL0 and EL1 System register accesses to the Debug ROM registers to Hyp mode.

- This control does not cause any instructions to be trapped.
- Non-secure EL0 and EL1 System register accesses to the DBGDRAR or DBGDSAR are trapped to Hyp mode, unless it is trapped by DBGDSCR.ext.UDCCdis.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.
TDOSA, bit [10]

When FEAT_DoubleLock is implemented:

Trap debug OS-related register access. Traps Non-secure EL1 System register accesses to the powerdown debug registers to Hyp mode.

0b0  This control does not cause any instructions to be trapped.
0b1  Non-secure EL1 System register accesses to the powerdown debug registers are trapped to Hyp mode.

The registers for which accesses are trapped are as follows:

- DBGOSLSR, DBGOSLAR, DBGOSDLR, and DBGPRCR.
- Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

Note

These registers are not accessible at EL0.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Trap debug OS-related register access. Traps Non-secure EL1 System register accesses to the powerdown debug registers to Hyp mode.

0b0  This control does not cause any instructions to be trapped.
0b1  Non-secure EL1 System register accesses to the powerdown debug registers are trapped to Hyp mode.

The registers for which accesses are trapped are as follows:

- DBGOSLSR, DBGOSLAR, and DBGPRCR.
- Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

It is IMPLEMENTATION DEFINED whether accesses to DBGOSDLR are trapped.

Note

These registers are not accessible at EL0.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

TDA, bit [9]

Trap debug access. Traps Non-secure EL0 and EL1 System register accesses to those debug System registers in the (copro==0b1110) encoding space that are not trapped by either of the following:

- HDCR.TDRA.
- HDCR.TDOSA.

0b0  This control does not cause any instructions to be trapped.
Non-secure EL0 or EL1 System register accesses to the debug registers, other than the registers trapped by HDCR.TDRA and HDCR.TDOSA, are trapped to Hyp mode, unless it is trapped by DBGDSCRext.UDCCdis.

Traps of AArch32 accesses to DBGDTRRXint and DBGDTRTXint are ignored in Debug state.

If HCR.TGE or HDCR.TDE is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**TDE, bit [8]**

Trap Debug exceptions. Controls routing of Debug exceptions, and defines the debug target Exception level, EL_D:

- The debug target Exception level is EL1.
- If EL2 is enabled for the current Effective value of SCR.NS, the debug target Exception level is EL2, otherwise the debug target Exception level is EL1.

The HDCR.{TDRA, TDOSA, TDA} fields are treated as being 1 for all purposes other than returning the result of a direct read of the register.

For more information, see Routing debug exceptions on page G2-9043.

When HCR.TGE == 1, the PE behaves as if the value of this field is 1 for all purposes other than returning the value of a direct read of the register.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

**HPME, bit [7]**

*When FEAT_PMUv3 is implemented:*

[HDCR.HPMN.\((N-1)\)] event counters enable.

- Event counters in the range [HDCR.HPMN.\((PMCR.N-1)\)] are disabled.
- Event counters in the range [HDCR.HPMN.\((PMCR.N-1)\)] are enabled by PMCNTENSET.

If HDCR.HPMN is less than PMCR.N, this field affects the operation of event counters in the range [HDCR.HPMN.\((PMCR.N-1)\)].

This field does not affect the operation of other event counters.

The operation of this field applies even when EL2 is disabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

**TPM, bit [6]**

*When FEAT_PMUv3 is implemented:*

Trap Performance Monitors accesses. Traps Non-secure EL0 and EL1 accesses to all Performance Monitors registers to Hyp mode.

- This control does not cause any instructions to be trapped.

---

G8-9940 Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. ARM DDI 0487I.a Non-Confidential
Non-secure EL0 and EL1 accesses to all Performance Monitors registers are trapped to Hyp mode.

--- Note ---
EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

TPMCR, bit [5]
When FEAT_PMUv3 is implemented:
Trap PMCR accesses. Traps Non-secure EL0 and EL1 accesses to the PMCR to Hyp mode.

0b0	This control does not cause any instructions to be trapped.

0b1	Non-secure EL0 and EL1 accesses to the PMCR are trapped to Hyp mode, unless it is trapped by PMUSERENR.EN.

--- Note ---
EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

HPMN, bits [4:0]
When FEAT_PMUv3 is implemented:
Defines the number of event counters that are accessible from Non-secure EL1 modes, and from Non-secure EL0 modes if unprivileged access is enabled.

If HPMN is not 0 and is less than PMCR.N, HPMN divides the event counters into a first range [0..(HPMN-1)], and a second range [HPMN..(PMCR.N-1)].

If FEAT_HPMN0 is implemented and this field is 0, all event counters are in the second range and none are in the first range.

If HPMN is equal to PMCR.N, all event counters are in the first range, and none are in the second range.

For an event counter <n> in the first range:

- The counter is accessible from EL1, EL2, EL3.
- The counter is accessible from EL0 if permitted by PMUSERENR.
- If FEAT_PMUv3p5 is implemented, PMCR.LP determines whether the counter overflows at PMEVCNTR<n>[31:0] or PMEVCNTR<n>[63:0].
- PMCR.E and PMCNTENSET[n] enable the operation of event counter n.

For an event counter <n> in the second range:

- The counter is accessible from EL2 and EL3.
• If EL2 is disabled in the current Security state, the event counter is also accessible from EL1, and from EL0 if permitted by PMUSERENR.
• If FEAT_PMUV3p5 is implemented, HDCR.HLP determines whether the counter overflows at PMEVCTR<n>[31:0] or PMEVCTR<n>[63:0].
• HDCR.HPME and PMCNTENSET[n] enable the operation of event counter n.

If HPMN is larger than PMCR.N, or if FEAT_HPMN0 is not implemented and HPMN is 0, the following CONSTRAINED UNPREDICTABLE behaviors apply:
• The value returned by a direct read of HDCR.HPMN is UNKNOWN.
• Either:
  — An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if HDCR.HPMN is set to an UNKNOWN non-zero value less than or equal to PMCR.N.
  — All counters are reserved for EL2 use, meaning no counters are accessible from Non-secure EL1 and Non-secure EL0.

The reset behavior of this field is:
• On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to the value in PMCR.N.
  — Otherwise, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

### Accessing HDCR

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} \text{ <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2}>}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
else
  UNDEFINED;
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end
  else
    R[t] = HDCR;
elsif PSTATE_EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
```
else
    R[t] = HDCR;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            UNDEFINED;
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.AArch32SystemAccessTrap(EL3, 0x03);
            end
    else
        HDCR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        HDCR = R[t];

### G8.3.32 HTRFCR, Hyp Trace Filter Control Register

The HTRFCR characteristics are:

#### Purpose

Provides EL2 controls for Trace.

#### Configurations

AArch32 System register HTRFCR bits [31:0] are architecturally mapped to AArch64 System register TRFCR_EL2[31:0].

This register is present only when EL2 is capable of using AArch32 and FEAT_TRF is implemented. Otherwise, direct accesses to HTRFCR are UNDEFINED.

If EL2 is not implemented, this register is RES0 from Monitor mode when SCR.NS == 1.

#### Attributes

HTRFCR is a 32-bit register.

#### Field descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits [31:7]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>TS, bits [6:5]</td>
<td>Timestamp Control. Controls which timebase is used for trace timestamps.</td>
</tr>
<tr>
<td>0b00</td>
<td>The timestamp is controlled by TRFCR.TS.</td>
</tr>
<tr>
<td>0b01</td>
<td>Virtual timestamp. The traced timestamp is the physical counter value minus the value of CNTVOFF.</td>
</tr>
<tr>
<td>0b11</td>
<td>Physical timestamp. The traced timestamp is the physical counter value.</td>
</tr>
<tr>
<td>When SelfHostedTraceEnabled() == FALSE, this field is ignored.</td>
<td></td>
</tr>
<tr>
<td>The reset behavior of this field is:</td>
<td></td>
</tr>
<tr>
<td>• On a Warm reset:</td>
<td></td>
</tr>
<tr>
<td>— When the PE resets into EL2 or EL3, this field resets to 0.</td>
<td></td>
</tr>
<tr>
<td>— Otherwise, this field resets to an architecturally UNKNOWN value.</td>
<td></td>
</tr>
<tr>
<td>Bit [4]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>0b0</td>
<td>VMID tracing is not allowed.</td>
</tr>
<tr>
<td>0b1</td>
<td>VMID tracing is allowed.</td>
</tr>
<tr>
<td>When SelfHostedTraceEnabled() == FALSE, this field is ignored.</td>
<td></td>
</tr>
<tr>
<td>The reset behavior of this field is:</td>
<td></td>
</tr>
<tr>
<td>• On a Warm reset:</td>
<td></td>
</tr>
<tr>
<td>— When the PE resets into EL2 or EL3, this field resets to 0.</td>
<td></td>
</tr>
</tbody>
</table>
Bit [2]

Reserved, RES0.

E2TRE, bit [1]

EL2 Trace Enable.

0b0  Tracing is prohibited at EL2.

0b1  Tracing is allowed at EL2.

When SelfHostedTraceEnabled() == FALSE, this field is ignored.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

E0HTRE, bit [0]

EL0 Trace Enable.

0b0  Tracing is prohibited at EL0 when HCR.TGE == 1.

0b1  Tracing is allowed at EL0 when HCR.TGE == 1.

This field is ignored if any of the following are true:

- The PE is in Secure state.
- SelfHostedTraceEnabled() == FALSE.
- HCR.TGE == 0.

The reset behavior of this field is:

- On a Warm reset:
  - When the PE resets into EL2 or EL3, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing HTRFCR

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} \langle\text{coproc}\rangle, \{\#\}<\text{opc1}>, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>, \{\#\}<\text{opc2}> \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then

UNDEFINED;

elsif PSTATE.EL == EL1 then

if EL2Enabled() & EDSCR.AttachReserved & HSTR_EL2.T1 == '1' then

AArch64.AArch32SystemAccessTrap(EL2, 0x03);

elsif EL2Enabled() & EDSCR.AttachReserved & HSTR_EL2.T1 == '1' then

AArch32.TakeHypTrapException(0x03);
else

UNDEFINED;

elsif PSTATE.EL == EL2 then

if Halted() & HaveEL(EL3) & EDSCR.AttachReserved == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & EDSCR.AttachReserved & MDCR_EL3.TTRF == '1' then

UNDEFINED;

elsif Halted() & HaveEL(EL3) & EDSCR.AttachReserved == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & EDSCR.AttachReserved & SDCR.TTRF == '1' then

...
UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TTRF == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
  endif;
else
  R[t] = HTRFCR;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    R[t] = HTRFCR;
  endif;
MCR{<c>}{<q>} <coproc>, {%#<opc1>}, <Rt>, <CRn>, <CRm>{, {%#<opc2>}}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
  endif;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SDCR.TTRF == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    endif;
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TTRF == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch32.TakeMonitorTrapException();
    endif;
  else
    HTRFCR = R[t];
  endif;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    HTRFCR = R[t];
  endif;
G8.3.33 PMMIR, Performance Monitors Machine Identification Register

The PMMIR characteristics are:

**Purpose**

Describes Performance Monitors parameters specific to the implementation to software.

**Configurations**

This register is present only when EL1 is capable of using AArch32 and FEAT_PMUv3p4 is implemented. Otherwise, direct accesses to PMMIR are UNDEFINED.

**Attributes**

PMMIR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>[31:24]  RES0</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>THWIDTH, Bits [23:20]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PMEVTYPER&lt;(n)&gt;_EL0.TH width. Indicates implementation of the FEAT_PMUv3_TH feature, and, if implemented, the size of the PMEVTYPER&lt;(n)&gt;_EL0.TH field.</td>
<td></td>
</tr>
<tr>
<td>0b0000</td>
<td>FEAT_PMUv3_TH is not implemented.</td>
<td></td>
</tr>
<tr>
<td>0b0001</td>
<td>1 bit. PMEVTYPER&lt;(n)&gt;_EL0.TH[11:1] are RES0.</td>
<td></td>
</tr>
<tr>
<td>0b0010</td>
<td>2 bits. PMEVTYPER&lt;(n)&gt;_EL0.TH[11:2] are RES0.</td>
<td></td>
</tr>
<tr>
<td>0b0011</td>
<td>3 bits. PMEVTYPER&lt;(n)&gt;_EL0.TH[11:3] are RES0.</td>
<td></td>
</tr>
<tr>
<td>0b0100</td>
<td>4 bits. PMEVTYPER&lt;(n)&gt;_EL0.TH[11:4] are RES0.</td>
<td></td>
</tr>
<tr>
<td>0b0101</td>
<td>5 bits. PMEVTYPER&lt;(n)&gt;_EL0.TH[11:5] are RES0.</td>
<td></td>
</tr>
<tr>
<td>0b0110</td>
<td>6 bits. PMEVTYPER&lt;(n)&gt;_EL0.TH[11:6] are RES0.</td>
<td></td>
</tr>
<tr>
<td>0b0111</td>
<td>7 bits. PMEVTYPER&lt;(n)&gt;_EL0.TH[11:7] are RES0.</td>
<td></td>
</tr>
<tr>
<td>0b1000</td>
<td>8 bits. PMEVTYPER&lt;(n)&gt;_EL0.TH[11:8] are RES0.</td>
<td></td>
</tr>
<tr>
<td>0b1001</td>
<td>9 bits. PMEVTYPER&lt;(n)&gt;_EL0.TH[11:9] are RES0.</td>
<td></td>
</tr>
<tr>
<td>0b1010</td>
<td>10 bits. PMEVTYPER&lt;(n)&gt;_EL0.TH[11:10] are RES0.</td>
<td></td>
</tr>
<tr>
<td>0b1101</td>
<td>11 bits. PMEVTYPER&lt;(n)&gt;_EL0.TH[11] is RES0.</td>
<td></td>
</tr>
<tr>
<td>0b1110</td>
<td>12 bits.</td>
<td></td>
</tr>
</tbody>
</table>

All other values are reserved.

If FEAT_PMUv3_TH is not implemented, this field is zero.

Otherwise, the largest value that can be written to PMEVTYPER<\(n\)>_EL0.TH is \(2^{(\text{PMMIR.THWIDTH}) - 1}\).

**Note**

PMEVTYPER<\(n\)>_EL0.TH cannot be accessed through PMEVTYPER<\(n\)>.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.
**BUS_WIDTH, bits [19:16]**

Bus width. Indicates the number of bytes each BUS_ACCESS event relates to. Encoded as \( \log_2(\text{number of bytes}) \), plus one.

- 0b0000  The information is not available.
- 0b0011  Four bytes.
- 0b0100  8 bytes.
- 0b0101  16 bytes.
- 0b0110  32 bytes.
- 0b0111  64 bytes.
- 0b1000  128 bytes.
- 0b1001  256 bytes.
- 0b1010  512 bytes.
- 0b1011  1024 bytes.
- 0b1100  2048 bytes.

All other values are reserved.

Each transfer is up to this number of bytes. An access might be smaller than the bus width.

When this field is nonzero, each access counted by BUS_ACCESS is at most BUS_WIDTH bytes. An implementation might treat a wide bus as multiple narrower buses, such that a wide access on the bus increments the BUS_ACCESS counter by more than one.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**BUS_SLOTS, bits [15:8]**

Bus count. The largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle.

When this field is nonzero, the largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle is BUS_SLOTS.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**SLOTS, bits [7:0]**

Operation width. The largest value by which the STALL_SLOT event might increment by in a single cycle. If the STALL SLOT event is not implemented, this field might read as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Accessing PMMIR**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{\langle c\rangle\}\{\langle q\rangle\} \langle \text{coproc}\rangle, \langle\#\langle \text{opc1}\rangle\rangle, \langle \text{Rt}\rangle, \langle \text{CRn}\rangle, \langle \text{CRm}\rangle, \langle\#\langle \text{opc2}\rangle\rangle
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1' && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
  UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  else
    R[t] = PMMIR;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED “EL3 trap priority
  when SDD == '1'” && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  R[t] = PMMIR;
elsif PSTATE.EL == EL3 then
  R[t] = PMMIR;
G8.3.34   SDCR, Secure Debug Control Register

The SDCR characteristics are:

**Purpose**

Provides EL3 configuration options for self-hosted debug, trace, and the Performance Monitors Extension.

**Configurations**

This register is present only when EL3 is capable of using AArch32. Otherwise, direct accesses to SDCR are UNDEFINED.

**Attributes**

SDCR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th></th>
<th>Bits [31:29]</th>
<th>Bits [28]</th>
<th>Bits [27]</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>31 29</td>
<td>RES0</td>
<td>MTPME</td>
</tr>
<tr>
<td>RES0</td>
<td>28</td>
<td>TRDC</td>
<td>SCCD</td>
</tr>
<tr>
<td>SPD</td>
<td>27</td>
<td>RES0</td>
<td>SPME</td>
</tr>
<tr>
<td>SPD</td>
<td>26</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>25</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>24</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>23</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>22</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>21</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>20</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>19</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>18</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>17</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>16</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>15</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>14</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>13</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>12</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>11</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>10</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>9</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>8</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>7</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>6</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>5</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>4</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>3</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>2</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>1</td>
<td>RES0</td>
<td>SPD</td>
</tr>
<tr>
<td>SPD</td>
<td>0</td>
<td>RES0</td>
<td>SPD</td>
</tr>
</tbody>
</table>

**MTPME, bit [28]**

*When FEAT_MTPMU is implemented:*

Multi-threaded PMU Enable. Enables use of the PMEVTYPER<\(n\)>.MT bits.

- 0b0  FEAT_MTPMU is disabled. The Effective value of PMEVTYPER<\(n\)>.MT is 0.
- 0b1  PMEVTYPER<\(n\)>.MT bits not affected by this bit.

If FEAT_MTPMU is disabled for any other PE in the system that has the same level 1 Affinity as the PE, it is IMPLEMENTATION DEFINED whether the PE behaves as if this bit is 0.

The reset behavior of this field is:

- On a Cold reset, in a system where the PE resets into EL3, this field resets to 1.

**Otherwise:**

Reserved, RES0.

**TDCC, bit [27]**

*When FEAT_FGT is implemented:*

Trap DCC. Traps use of the Debug Comms Channel in modes other than Monitor mode to Monitor mode.

- 0b0  This control does not cause any register accesses to be trapped.
- 0b1  Accesses to the DCC registers in modes other than Monitor mode generate a Monitor Trap exception, unless the access also generates a higher priority exception.

The DCC registers trapped by this control are:

- **DBGDTRRX**ext, **DBGDTRTX**ext, **DBGDSCR**int, **DBGDCCINT**, and, when the PE is in Non-debug state, **DBGDTRRX**int and **DBGDTRTX**int.
When the PE is in Debug state, SDCR.TDCC does not trap any accesses to:

- **DBGDTRRXInt and DBGDTRTXInt.**

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL3, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**
Reserved, RES0.

**Bits [26:24]**
Reserved, RES0.

**SCCD, bit [23]**

*When FEAT_PMUv3p5 is implemented:*
Secure Cycle Counter Disable. Prohibits PMCCNTR from counting in Secure state.

- **0b0** Cycle counting by PMCCNTR is not affected by this mechanism.
- **0b1** Cycle counting by PMCCNTR is prohibited in Secure state.

This field does not affect the CPU_CYCLES event or any other event that counts cycles.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**Bit [22]**
Reserved, RES0.

**EPMAD, bit [21]**

*When FEAT_Debugv8p4 is implemented and FEAT_PMUv3 is implemented:*
External Performance Monitors Non-secure access disable. Controls Non-secure access to Performance Monitors registers by an external debugger.

- **0b0** Non-secure access to the Performance Monitors registers from an external debugger is permitted.
- **0b1** Non-secure access to the Performance Monitors registers from an external debugger is not permitted.

If the Performance Monitors Extension does not support external debug interface accesses, this bit is RES0.

Otherwise, if EL3 is not implemented and the Effective value of SCR.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

*When FEAT_PMUv3 is implemented:*
External Performance Monitors access disable. Controls access to Performance Monitors registers by an external debugger.

- **0b0** Access to Performance Monitors registers from an external debugger is permitted.
- **0b1** Access to Performance Monitors registers from an external debugger is not permitted, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

If the Performance Monitors Extension does not support external debug interface accesses, this bit is RES0.

Otherwise, if EL3 is not implemented and the Effective value of SCR.NS is 0, then the Effective value of this field is 1.
The reset behavior of this field is:
- On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**EDAD, bit [20]**

- **When FEAT_Debugv8p4 is implemented:**
  External debug Non-secure access disable. Controls Non-secure access to breakpoint, watchpoint, and OSLAR_EL1 registers by an external debugger.
  - 0b0 Non-secure access to debug registers from an external debugger is permitted.
  - 0b1 Non-secure access to breakpoint registers, watchpoint registers, and OSLAR_EL1 from an external debugger is not permitted.

  If EL3 is not implemented and the Effective value of SCR.NS is 0, then the Effective value of this field is 1.

  The reset behavior of this field is:
  - On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

- **When FEAT_Debugv8p2 is implemented:**
  External debug access disable. Controls access to breakpoint, watchpoint, and OSLAR_EL1 registers by an external debugger.
  - 0b0 Access to debug registers from an external debugger is permitted.
  - 0b1 Access to breakpoint registers, watchpoint registers, and OSLAR_EL1 from an external debugger is not permitted, unless overridden by the IMPLEMENTATION DEFINED authentication interface.

  If EL3 is not implemented and the Effective value of SCR.NS is 0, then the Effective value of this field is 1.

  The reset behavior of this field is:
  - On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

  **Otherwise:**
  External debug access disable. Controls access to breakpoint, watchpoint, and optionally OSLAR_EL1 registers by an external debugger.
  - 0b0 Access to debug registers from an external debugger is permitted.
  - 0b1 Access to breakpoint registers and watchpoint registers from an external debugger is not permitted, unless overridden by the IMPLEMENTATION DEFINED authentication interface. It is IMPLEMENTATION DEFINED whether access to the OSLAR_EL1 register from an external debugger is permitted or not permitted.

  If EL3 is not implemented and the Effective value of SCR.NS is 0, then the Effective value of this field is 1.

  The reset behavior of this field is:
  - On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

**TTRF, bit [19]**

- **When FEAT_TRF is implemented:**
  Trap Trace Filter controls. Controls whether accesses in modes other than Monitor mode to the trace filter control registers generate a Monitor Trap exception.
  - 0b0 Accesses to HTRFCR and TRFCR are not affected by this control bit.
  - 0b1 When not in Monitor mode, accesses to HTRFCR and TRFCR generate a Monitor Trap exception, unless the access generates a higher priority exception.

  The reset behavior of this field is:
  - On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.
**Otherwise:**

Reserved, RES0.

**STE, bit [18]**

*When FEAT_TRF is implemented:*

Secure Trace Enable. This bit enables tracing in Secure state and controls the level of authentication required by an external debugger to enable external tracing.

- **0b0**: Trace is prohibited in Secure state unless overridden by the IMPLEMENTATION DEFINED authentication interface.
- **0b1**: Trace in Secure state is not affected by this bit.

This bit also controls the level of authentication required by an external debugger to enable external tracing. See *Register controls to enable self-hosted trace on page G3-9104*.

If EL3 is not implemented and the Effective value of SCR.NS is 0, the PE behaves as if this bit is set to 1.

The reset behavior of this field is:

- **On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.**

**Otherwise:**

Reserved, RES0.

**SPME, bit [17]**

*When FEAT_PMUv3 is implemented and FEAT_Debugv8p2 is implemented:*

Secure Performance Monitors Enable. Controls event counting in Secure state.

- **0b0**: Event counting is prohibited in Secure state. If PMCR.DP is 1, PMCCNTR is disabled in Secure state. Otherwise, PMCCNTR is not affected by this mechanism.
- **0b1**: Event counting and PMCCNTR are not affected by this mechanism.

This field affects the operation of all event counters in Secure state, and if PMCR.DP is 1, the operation of PMCCNTR in Secure state. When PMCR.DP is 0, PMCCNTR is not affected by this field.

If EL3 is not implemented and the Effective value of SCR.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- **On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.**

*When FEAT_PMUv3 is implemented:*

Secure Performance Monitors Enable. Controls event counting in Secure state.

- **0b0**: If ExternalSecureNoninvasiveDebugEnabled() is FALSE, event counting is prohibited in Secure state, and if PMCR.DP is 1, PMCCNTR is disabled in Secure state.
- **0b1**: Event counting and PMCCNTR are not affected by this mechanism.

If ExternalSecureNoninvasiveDebugEnabled() is TRUE, the event counters and PMCCNTR are not affected by this field. Otherwise, this field affects the operation of all event counters in Secure state, and if PMCR.DP is 1, the operation of PMCCNTR in Secure state. When PMCR.DP is 0, PMCCNTR is not affected by this field.

If EL3 is not implemented and the Effective value of SCR.NS is 0, then the Effective value of this field is 1.

The reset behavior of this field is:

- **On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.**

**Otherwise:**

Reserved, RES0.
Bit [16]

Reserved, RES0.

SPD, bits [15:14]

AArch32 Secure self-hosted Privileged Debug. Enables or disables debug exceptions from EL3, other than Breakpoint Instruction exceptions.

0b00 Legacy mode. Debug exceptions from EL3 are enabled by the authentication interface.
0b10 Secure privileged debug disabled. Debug exceptions from EL3 are disabled.
0b11 Secure privileged debug enabled. Debug exceptions from EL3 are enabled.

Other values are reserved, and have the CONSTRAINED UNPREDICTABLE behavior that they must have the same behavior as 0b00. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

This field has no effect on Breakpoint Instruction exceptions. These are always enabled.

This field is ignored in Non-secure state.

If debug exceptions from EL3 are enabled, then debug exceptions from Secure EL0 are also enabled.

Otherwise, debug exceptions from Secure EL0 are enabled only if the value of SDER.SUIDEN is 1.

If EL3 is not implemented and the Effective value of SCR.NS is 0, then the Effective value of this field is 0b11.

The reset behavior of this field is:

- On a Warm reset, in a system where the PE resets into EL3, this field resets to 0.

Bits [13:0]

Reserved, RES0.

Accessing SDCR

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} <\text{coproc}>, \{#<\text{opc1}>\} , <\text{Rt}>, <\text{CRn}>, <\text{CRm}>, \{#<\text{opc2}>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        UNDEFINED;
    elsif PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.EL == EL3 then
        R[t] = SDCR;
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 = '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 = '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    UNDEFINED;
elseif PSTATE.EL == EL2 then
  UNDEFINED;
elseif PSTATE.EL == EL3 then
  if CP15SDISABLE2 == HIGH then
    UNDEFINED;
  else
    SDCR = R[t];
G8.3.35  **SDER, Secure Debug Enable Register**

The SDER characteristics are:

**Purpose**

Controls invasive and non-invasive debug in the Secure EL0 mode.

**Configurations**

AArch32 System register SDER bits [31:0] are architecturally mapped to AArch64 System register SDER32_EL2[31:0] when EL2 is implemented and FEAT_SEL2 is implemented.

AArch32 System register SDER bits [31:0] are architecturally mapped to AArch64 System register SDER32_EL3[31:0] when EL3 is implemented.

This register is present only when (EL3 is implemented and EL3 is capable of using AArch32) or (EL1 is capable of using AArch32 and Secure EL1 is implemented). Otherwise, direct accesses to SDER are UNDEFINED.

This register is ignored by the PE when one or more of the following are true:

- The PE is in Non-secure state.
- EL1 is using AArch64.

**Attributes**

SDER is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Secure User Non-Invasive Debug Enable.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Secure User Invasive Debug Enable.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SUNIDEN, bit [1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Reserved, RES0</td>
<td>0x0</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:2]**

Reserved, RES0.

**SUNIDEN, bit [1]**

Secure User Non-Invasive Debug Enable.

- **0b0** This bit has no effect on non-invasive debug.
- **0b1** Non-invasive debug is allowed in Secure EL0 using AArch32.

When EL3 or Secure EL1 is using AArch32, the forms of non-invasive debug affected by this control are:

- The PC Sample-based Profiling Extension. See About the PC Sample-based Profiling Extension.
- When SelfHostedTraceEnabled() == FALSE, processor trace.
- When EL3 is implemented, Performance Monitors.

When Secure EL1 is using AArch64, this bit has no effect.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

**SUIDEN, bit [0]**

When EL3 is implemented:

Secure User Invasive Debug Enable.

- **0b0** This bit does not affect the generation of debug exceptions at Secure EL0.
- **0b1** If EL3 or EL1 is using AArch32, debug exceptions from Secure EL0 are enabled.
The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**Accessing SDER**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} \text{ <coproc>, \{<opc1>, <Rt>, <CRn>, <CRm>, \{<opc2>\}}} \\
\text{coproc opc1 CRn CRm opc2} \\
\begin{array}{ccccc}
0b111 & 0b000 & 0b0001 & 0b0001 & 0b001 \\
\end{array}
\]

\[
\text{MCR}\{<c>\}{<q>} \text{ <coproc>, \{<opc1>, <Rt>, <CRn>, <CRm>, \{<opc2>\}}} \\
\text{coproc opc1 CRn CRm opc2} \\
\begin{array}{ccccc}
0b111 & 0b000 & 0b0001 & 0b0001 & 0b001 \\
\end{array}
\]
elsif PSTATE.EL == EL3 then
    if CP15SDISABLE2 == HIGH then
        UNDEFINED;
    else
        SDER = R[t];

G8.3.36 TRFCR, Trace Filter Control Register

The TRFCR characteristics are:

**Purpose**

Provides EL1 controls for Trace.

**Configurations**

AArch32 System register TRFCR bits [31:0] are architecturally mapped to AArch64 System register TRFCR_EL1[31:0]. This register is present only when EL1 is capable of using AArch32 and FEAT_TRF is implemented. Otherwise, direct accesses to TRFCR are UNDEFINED.

**Attributes**

TRFCR is a 32-bit register.

**Field descriptions**

![Registers Diagram]

**Bits [31:7]**

Reserved, RES0.

**TS, bits [6:5]**

Timestamp Control. Controls which timebase is used for trace timestamps.

- **0b01** Virtual timestamp. The traced timestamp is the physical counter value minus the value of CNTVOFF.
- **0b10** *When FEAT_ECV is implemented:*
  
  Guest physical timestamp. The traced timestamp is the physical counter value minus a physical offset. If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2:
  
  - EL3 is implemented and is using AArch32.
  - EL3 is implemented, using AArch64, and SCR_EL3.ECVEn == 0b0.
  - EL2 is using AArch32.
  - EL2 is using AArch64 and CNTHCTL_EL2.ECV == 0b0.

- **0b11** Physical timestamp. The traced timestamp is the physical counter value.

All other values are reserved.

This field is ignored by the PE when any of the following are true:

- EL2 is implemented and HTRFCR.TS != 0b0.
- SelfHostedTraceEnabled () == FALSE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [4:2]**

Reserved, RES0.

**E1TRE, bit [1]**

EL1 Trace Enable.

- **0b0** Tracing is prohibited in PL1 modes.
0b1  Tracing is allowed in PL1 modes.
This field is ignored if SelfHostedTraceEnabled() == FALSE.
The reset behavior of this field is:
  • On a Warm reset, this field resets to 0.

**E0TRE, bit [0]**
EL0 Trace Enable.
0b0  Tracing is prohibited at EL0.
0b1  Tracing is allowed at EL0.
This field is ignored if any of the following are true:
  • SelfHostedTraceEnabled() == FALSE.
  • EL2 is implemented and enabled in the current security state and HCR.TGE == 1.
The reset behavior of this field is:
  • On a Warm reset, this field resets to 0.

### Accessing TRFCR

Accesses to this register use the following encodings in the System register encoding space:


<table>
<thead>
<tr>
<th>MRC{&lt;c&gt;}{&lt;q&gt;} &lt;coproc&gt;, {#}&lt;opc1&gt;, &lt;Rt&gt;, &lt;CRn&gt;, &lt;CRm&gt;{, {#}&lt;opc2&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>coproc opc1 CRn CRm opc2</td>
</tr>
<tr>
<td>0b1111 0b000 0b0001 0b0010 0b001</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
    UNDEFINED;
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TTRF == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TTRF == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
    if Halting() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SDGR.TTRF == '1' then
    if Halting() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch32.TakeMonitorTrapException();
    end
  else
    R[t] = TRFCR;
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
    UNDEFINED;
  end
```

else if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SDCR.TTRF == '1' then
  UNDEFINED;
else if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
else if HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TTRF == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
  endif
else
  UNDEFINED;
else
  R[t] = TRFCR;
endif

if PSTATE.EL == EL0 then
  UNDEFINED;
else if PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
    UNDEFINED;
  else if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SDCR.TTRF == '1' then
    AArch32.TakeMonitorTrapException();
  else
    R[t] = TRFCR;
  endif
else
  if PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TTRF == '1' then
      UNDEFINED;
    else if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SDCR.TTRF == '1' then
      AArch32.TakeMonitorTrapException();
    else
      if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        TRFCR = R[t];
      endif
      elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
      endif
    endif
  endif
endif

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0001</td>
<td>0b0010</td>
<td>0b0001</td>
</tr>
</tbody>
</table>
if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SDCR.TTRF == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch32.TakeMonitorTrapException();
    else
        TRFCR = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SDCR.TTRF == '1' then
        AArch32.TakeMonitorTrapException();
    else
        TRFCR = R[t];
G8.4 Performance Monitors registers

This section lists the Performance Monitors registers in AArch32.
## G8.4.1  PMCCFILTR, Performance Monitors Cycle Count Filter Register

The PMCCFILTR characteristics are:

### Purpose

Determines the modes in which the Cycle Counter, PMCCNTR, increments.

### Configurations

AArch32 System register PMCCFILTR bits [31:0] are architecturally mapped to AArch64 System register PMCCFILTR_EL0[31:0].

AArch32 System register PMCCFILTR bits [31:0] are architecturally mapped to External register PMCCFILTR_EL0[31:0].

This register is present only when AArch32 is supported and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCCFILTR are undefined.

### Attributes

PMCCFILTR is a 32-bit register.

### Field descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>User filtering bit. Controls counting in EL0.</td>
</tr>
<tr>
<td>NSK</td>
<td>When EL3 is implemented: Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1.</td>
</tr>
</tbody>
</table>

#### P, bit [31]

Privileged filtering bit. Controls counting in EL1.

If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the PMCCFILTR.NSK bit.

0b0  Count cycles in EL1.
0b1  Do not count cycles in EL1.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

#### U, bit [30]

User filtering bit. Controls counting in EL0.

If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the PMCCFILTR.NSU bit.

If FEAT_RME is implemented, then counting in Realm EL0 is further controlled by the PMCCFILTR.RLU bit.

0b0  Count cycles in EL0.
0b1  Do not count cycles in EL0.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

#### NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1.

If the value of this bit is equal to the value of PMCCFILTR.P, cycles in Non-secure EL1 are counted.

Otherwise, cycles in Non-secure EL1 are not counted.
The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**NSU, bit [28]**

*When EL3 is implemented:*
Non-secure EL0 (Unprivileged) filtering. Controls counting in Non-secure EL0.
- If the value of this bit is equal to the value of PMCCFILTR.U, cycles in Non-secure EL0 are counted.
- Otherwise, cycles in Non-secure EL0 are not counted.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**NSH, bit [27]**

*When EL2 is implemented:*
EL2 (Hyp mode) filtering bit. Controls counting in EL2.
- 0b0: Do not count cycles in EL2.
- 0b1: Count cycles in EL2.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**Bits [26:22]**
Reserved, RES0.

**RLU, bit [21]**

*When FEAT_RME is implemented:*
Realm EL0 (unprivileged) filtering bit. Controls counting in Realm EL0.
- If the value of this bit is equal to the value of the PMCCFILTR.U bit, cycles in Realm EL0 are counted.
- Otherwise, cycles in Realm EL0 are not counted.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**Bits [20:0]**
Reserved, RES0.

**Accessing PMCCFILTR**

PMCCFILTR can also be accessed by using PMXEVTYPER with PMSELR.SEL set to 0b111111.

Accesses to this register use the following encodings in the System register encoding space:
MRC{<c>}{<q>} <coproc>, (#)opc1, <Rt>, <CRn>, <CRm>{, (#)opc2}
```c
MCR<coproc>,<CRn>,<CRm>{,#<opc1>,,#<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b111</td>
<td>0b111</td>
</tr>
</tbody>
</table>

```c
if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elseif !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  elseif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      UNDEFINED;
  elseif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGWTR_EL2.PMCCFILTR_EL0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  else
    PMCCFILTR = R[t];
  elseif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
      AArch32.TakeHypTrapException(0x03);
    elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
      PMCCFILTR = R[t];
  elseif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
      PMCCFILTR = R[t];
  elseif PSTATE.EL == EL3 then
    PMCCFILTR = R[t];
```

G8.4.2 PMCCNTR, Performance Monitors Cycle Count Register

The PMCCNTR characteristics are:

**Purpose**

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. See *Time as measured by the Performance Monitors cycle counter* on page D11-5246 for more information. PMCCFILTR determines the modes and states in which the PMCCNTR can increment.

**Configurations**

AArch32 System register PMCCNTR bits [63:0] are architecturally mapped to AArch64 System register PMCCNTR_EL0[63:0].

AArch32 System register PMCCNTR bits [63:0] are architecturally mapped to External register PMCCNTR_EL0[63:0].

This register is present only when AArch32 is supported and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCCNTR are UNDEFINED.

PMCCNTR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits [31:0] and do not modify bits [63:32].

All counters are subject to any changes in clock frequency, including clock stopping caused by the WFI and WFE instructions. This means that it is **CONSTRAINED UNPREDICTABLE** whether or not PMCCNTR continues to increment when clocks are stopped by WFI and WFE instructions.

**Attributes**

PMCCNTR is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CCNT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CCNT, bits [63:0]

Cycle count. Depending on the values of PMCR.{LC,D}, this field increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to PMCR.C sets this field to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Accessing PMCCNTR**

Accesses to this register use the following encodings in the System register encoding space:
MRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1101</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elseif !ELUsingAArch32(EL1) & PMUSERENR_EL0.<CR,EN> == '00' then
    if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  elseif EL2Enabled() & !ELUsingAArch32(EL1) & SCR_EL3.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() & !ELUsingAArch32(EL1) & PMUSERENR.<CR,EN> == '00' then
    if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  elseif !ELUsingAArch32(EL1) & PMUSERENR.<CR,EN> == '00' then
    if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  elseif EL2Enabled() & !ELUsingAArch32(EL1) & HCR_EL2.TGE == '1' then
    AArch32.TakeHypTrapException(0x00);
  else
    UNDEFINED;
elseif EL2Enabled() & !ELUsingAArch32(EL1) & SCR_EL3.TPM == '1' then
  if Halted() & EDSCR.SDD == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elseif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elseif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() & !ELUsingAArch32(EL2) & PMUSERENR.<CR,EN> == '00' then
    if EL2Enabled() & !ELUsingAArch32(EL3) & SCR_EL3.TPM == '1' then
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
      R[t] = PMCCNTR<31:0>;
  else
    R[t] = PMCCNTR<31:0>;
MCR<coproc>, <t><opc1>, <Rt>, <CRn>, <CRm>{, <opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL1) && PMUSEREN.EL0.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    endif
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  endif
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL2) && PMUSEREN.EL1.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    endif
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  endif
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL2) && PMUSEREN.EL2.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    endif
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  endif
elsif PSTATE.EL == EL3 then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
else
  PMCONTR = ZeroExtend(R[t]);
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL2) && PMUSEREN.EL0.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    endif
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  endif
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL2) && PMUSEREN.EL1.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    endif
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  endif
elsif PSTATE.EL == EL3 then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  PMCCNTR = ZeroExtend(R[t]);
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end
else
  PMCCNTR = ZeroExtend(R[t]);
eelsif PSTATE.EL == EL3 then
  PMCCNTR = ZeroExtend(R[t]);
end

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

| coproc CRm opc1 |
|-----------------|-----------------|-----------------|
| 0b1111 0b1001 0b0000 |

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL1) && PMUSERENR.<CR,EN> == '00' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    end
else
  AArch64.AArch32SystemAccessTrap(EL1, 0x04);
endif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x04);
else
  AArch64.AArch32SystemAccessTrap(EL2, 0x04);
deif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x04);
eif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' && HDFGRTR_EL2.PMCCNTR_EL0 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x04);
else
  AArch64.AArch32SystemAccessTrap(EL2, 0x04);
deif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x04);
deif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
  AArch32.TakeHypTrapException(0x04);
deif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x04);
  end
else
  (R[t2], R[t]) = (PMCCNTR<33:32>, PMCCNTR<31:0>);
AArch32.TakeHypTrapException(0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
 AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
 AArch32.TakeHypTrapException(0x04);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
 if Halted() && EDSCR.SDD == '1' then
 UNDEFINED;
 else
 AArch64.AArch32SystemAccessTrap(EL3, 0x04);
 else
 (R[t2], R[t]) = (PMCCNTR<63:32>, PMCCNTR<31:0>);
elsif PSTATE.EL == EL2 then
 if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
 when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
 UNDEFINED;
 elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
 if Halted() && EDSCR.SDD == '1' then
 UNDEFINED;
 else
 AArch64.AArch32SystemAccessTrap(EL3, 0x04);
 else
 (R[t2], R[t]) = (PMCCNTR<63:32>, PMCCNTR<31:0>);
 elsif PSTATE.EL == EL3 then
 (R[t2], R[t]) = (PMCCNTR<63:32>, PMCCNTR<31:0>);
if PSTATE.EL == EL0 then
 if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
 when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
 UNDEFINED;
 elsif !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
 if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
 AArch64.AArch32SystemAccessTrap(EL2, 0x04);
 else
 AArch64.AArch32SystemAccessTrap(EL1, 0x04);
 elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
 if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
 AArch64.AArch32SystemAccessTrap(EL2, 0x04);
 elsif EL2Enabled() && ELUsingAArch32(EL1) && HCR.TGE == '1' then
 AArch32.TakeHypTrapException(0x00);
 else
 UNDEFINED;
 elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
 AArch64.AArch32SystemAccessTrap(EL2, 0x04);
 elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
 AArch32.TakeHypTrapException(0x04);
 elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) ||
 SCR_EL3.FGTE == '1') && HDFGWTR_EL2.PMCCNTR_EL0 == '1' then
 AArch64.AArch32SystemAccessTrap(EL2, 0x04);
 elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
 AArch64.AArch32SystemAccessTrap(EL2, 0x04);
 elsif EL2Enabled() && ELUsingAArch32(EL1) && MDCR_EL3.TPM == '1' then
 AArch32.TakeHypTrapException(0x04);
 elsif EL2Enabled() && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
 if Halted() && EDSCR.SDD == '1' then
 UNDEFINED;
 else
 AArch64.AArch32SystemAccessTrap(EL3, 0x04);

---

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1001</td>
<td>0b0000</td>
</tr>
</tbody>
</table>
else
  PMCNTR = R[t2]:R[t];
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && EDSCR.SDD == '1' then
    AArch32.TakeHypTrapException(0x04);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.T9 == '1' then
    AArch32.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && EDSCR.T9 == '1' then
    AArch32.TakeHypTrapException(0x04);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x04);
    end
  else
    PMCNTR = R[t2]:R[t];
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x04);
    end
  else
    PMCNTR = R[t2]:R[t];
  end
elsif PSTATE.EL == EL3 then
  PMCNTR = R[t2]:R[t];
G8.4.3 PMCEID0, Performance Monitors Common Event Identification register 0

The PMCEID0 characteristics are:

**Purpose**

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range 0x0000 to 0x001F.

For more information about the Common events and the use of the PMCEIDn registers, see *The PMU event number space and common events* on page D11-5275.

**Configurations**

AArch32 System register PMCEID0 bits [31:0] are architecturally mapped to AArch64 System register PMCEID0_EL0[31:0].

AArch32 System register PMCEID0 bits [31:0] are architecturally mapped to External register PMCEID0[31:0].

This register is present only when AArch32 is supported and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCEID0 are UNDEFINED.

**Attributes**

PMCEID0 is a 32-bit register.

**Field descriptions**

| ID31 | ID30 | ID29 | ID28 | ID27 | ID26 | ID25 | ID24 | ID23 | ID22 | ID21 | ID20 | ID19 | ID18 | ID17 | ID16 | ID15 | ID14 | ID13 | ID12 | ID11 | ID10 | ID9  | ID8  | ID7  | ID6  | ID5  | ID4  | ID3  | ID2  | ID1  | ID0  |
|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|

ID<\(n\)>, bit [\(n\)], for \(n = 31\) to 0

ID[\(n\)] corresponds to Common event n.

For each bit:

\(0b0\) The Common event is not implemented, or not counted.

\(0b1\) The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

**Note**

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.
Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID0

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{c\}\{q\} \text{ <coproc>, \{#\}opc1, <Rt>, <CRn>, <CRm>\{, \{#\}opc2\}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    endif;
  elsif EL2Enabled() && PMUSERENR.EN == '0' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      UNDEFINED;
    endif;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
  endif;
else
  if EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    UNDEFINED;
  endif;
endif;

if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
    R[t] = PMCEID0;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        else
            R[t] = PMCEID0;
    elsif PSTATE.EL == EL3 then
        R[t] = PMCEID0;
    endif
else
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    endif
else
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        R[t] = PMCEID0;
    endif
endif
PMCEID1, Performance Monitors Common Event Identification register 1

The PMCEID1 characteristics are:

**Purpose**

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range \(0x0020\) to \(0x003F\).

For more information about the Common events and the use of the PMCEIDn registers see *The PMU event number space and common events* on page D11-5275.

**Configurations**

AArch32 System register PMCEID1 bits [31:0] are architecturally mapped to AArch64 System register PMCEID1_EL0[31:0].

AArch32 System register PMCEID1 bits [31:0] are architecturally mapped to External register PMCEID1[31:0].

This register is present only when AArch32 is supported and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCEID1 are UNDEFINED.

**Attributes**

PMCEID1 is a 32-bit register.

**Field descriptions**

ID\(<n>\), bit \([n]\), for \(n = 31\) to \(0\)

ID[\(n\)] corresponds to Common event \((0x0020 + n)\).

For each bit:

- \(0b0\) The Common event is not implemented, or not counted.
- \(0b1\) The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

**Note**

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.
Such an event might be added retrospectively to an earlier version of the PMU architecture,
provided the event does not require any additional PMU features and has an event number that can
be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

### Accessing PMCEID1

Accesses to this register use the following encodings in the System register encoding space:

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    end
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch32.TakeHypTrapException(0x03);
    else
      UNDEFINED;
    end
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
      AArch32.TakeHypTrapException(0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  R[t] = PMCEID1;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end
  else
    R[t] = PMCEID1;
else
  if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
      AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
      AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
      end
    else
      R[t] = PMCEID1;
else
  if PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
      AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
      AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
      end
    else
      R[t] = PMCEID1;
else
  if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
      AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
      AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
      end
    else
      R[t] = PMCEID1;
if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
    R[t] = PMCEID1;
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
    when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        else
            R[t] = PMCEID1;
        endif
    else
        if PSTATE_EL == EL3 then
            R[t] = PMCEID1;
        endif
    endif
end if
G8.4.5 PMCEID2, Performance Monitors Common Event Identification register 2

The PMCEID2 characteristics are:

**Purpose**

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range 0x4000 to 0x401F.

For more information about the Common events and the use of the PMCEIDn registers see *The PMU event number space and common events* on page D11-5275.

**Configurations**

AArch32 System register PMCEID2 bits [31:0] are architecturally mapped to AArch64 System register PMCEID0_EL0[63:32].

AArch32 System register PMCEID2 bits [31:0] are architecturally mapped to External register PMCEID2[63:32].

This register is present only when AArch32 is supported and FEAT_PMUv3p1 is implemented. Otherwise, direct accesses to PMCEID2 are UNDEFINED.

**Attributes**

PMCEID2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>IDhi&lt;n&gt;</th>
<th>bit [n], for n = 31 to 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDhi31</td>
<td></td>
</tr>
<tr>
<td>IDhi30</td>
<td></td>
</tr>
<tr>
<td>IDhi29</td>
<td></td>
</tr>
<tr>
<td>IDhi28</td>
<td></td>
</tr>
<tr>
<td>IDhi27</td>
<td></td>
</tr>
<tr>
<td>IDhi26</td>
<td></td>
</tr>
<tr>
<td>IDhi25</td>
<td></td>
</tr>
<tr>
<td>IDhi24</td>
<td></td>
</tr>
<tr>
<td>IDhi23</td>
<td></td>
</tr>
<tr>
<td>IDhi22</td>
<td></td>
</tr>
<tr>
<td>IDhi21</td>
<td></td>
</tr>
<tr>
<td>IDhi20</td>
<td></td>
</tr>
<tr>
<td>IDhi19</td>
<td></td>
</tr>
<tr>
<td>IDhi18</td>
<td></td>
</tr>
<tr>
<td>IDhi17</td>
<td></td>
</tr>
<tr>
<td>IDhi16</td>
<td></td>
</tr>
<tr>
<td>IDhi15</td>
<td></td>
</tr>
<tr>
<td>IDhi14</td>
<td></td>
</tr>
<tr>
<td>IDhi13</td>
<td></td>
</tr>
<tr>
<td>IDhi12</td>
<td></td>
</tr>
<tr>
<td>IDhi11</td>
<td></td>
</tr>
<tr>
<td>IDhi10</td>
<td></td>
</tr>
<tr>
<td>IDhi9</td>
<td></td>
</tr>
<tr>
<td>IDhi8</td>
<td></td>
</tr>
<tr>
<td>IDhi7</td>
<td></td>
</tr>
<tr>
<td>IDhi6</td>
<td></td>
</tr>
<tr>
<td>IDhi5</td>
<td></td>
</tr>
<tr>
<td>IDhi4</td>
<td></td>
</tr>
<tr>
<td>IDhi3</td>
<td></td>
</tr>
<tr>
<td>IDhi2</td>
<td></td>
</tr>
<tr>
<td>IDhi1</td>
<td></td>
</tr>
<tr>
<td>IDhi0</td>
<td></td>
</tr>
</tbody>
</table>

IDhi<n> corresponds to Common event (0x4000 + n).

For each bit:

| 0b0 | The Common event is not implemented, or not counted. |
| 0b1 | The Common event is implemented. |

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

**Note**

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.
Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID2

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn}, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    end
    elsif EL2Enabled() && ELUsingAArch32(EL1) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    end
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
    if !EL2Enabled() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end
  else
    R[t] = PMCEID2;
  end
```

### coproc opc1 CRn CRm opc2

```
0b1111 0b000 0b1001 0b1110 0b100
```
if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
    R[t] = PMCEID2;
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        end
    else
        R[t] = PMCEID2;
    end
elsif PSTATE_EL == EL3 then
    R[t] = PMCEID2;
else
    R[t] = PMCEID2;
G8.4.6 PMCEID3, Performance Monitors Common Event Identification register 3

The PMCEID3 characteristics are:

**Purpose**

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range 0x4020 to 0x403F.

For more information about the Common events and the use of the PMCEIDn registers see *The PMU event number space and common events on page D11-5275.*

**Configurations**

AArch32 System register PMCEID3 bits [31:0] are architecturally mapped to AArch64 System register PMCEID1_EL0[63:32].

AArch32 System register PMCEID3 bits [31:0] are architecturally mapped to External register PMCEID3[63:32].

This register is present only when AArch32 is supported and FEAT_PMUv3p1 is implemented. Otherwise, direct accesses to PMCEID3 are UNDEFINED.

**Attributes**

PMCEID3 is a 32-bit register.

**Field descriptions**

**IDhi<n>, bit [n], for n = 31 to 0**

IDhi[n] corresponds to Common event (0x4020 + n).

For each bit:

| 0b0 | The Common event is not implemented, or not counted. |
| 0b1 | The Common event is implemented. |

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

--- **Note** ---

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.
Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID3

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC\{c\}\{q\} \langle coproc \rangle, \langle \#\rangle\langle opc1 \rangle, \langle Rt \rangle, \langle CRn \rangle, \langle CRm \rangle\{\langle \#\rangle\langle opc2 \rangle\} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR.EL3.TPM == '1' then UNDEFINED;
elsif !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR.EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif ELUsingAArch32(EL1) && PMUSERENR.EL0.EN == '0' then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR.EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.EL2.TGE == '1' then
AArch32.TakeHypTrapException(0x00);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.EL2.T9 != '11' && HSTR.EL2.T9 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.EL2.T9 == '1' then
AArch32.TakeHypTrapException(0x00);
elsif EL2Enabled() && !ELUsingAArch32(EL1) && HSTR.EL2.T9 != '11' && (!HaveEL(EL3) || SCR.EL3.FGTEn == '1') && HDFGRTR.EL2.\text{PMCEIDn.EL0} == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDFGRTR.EL2.\text{PMCEIDn.EL0} == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.EL2.TPM == '1' then
AArch32.TakeHypTrapException(0x00);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR.EL3.TPM == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR.EL3.TPM == '1' then UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.EL2.T9 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.EL2.T9 == '1' then
AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL1) && MDCR.EL2.TPM == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR.EL3.TPM == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
R[t] = PMCEID3;
else
PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR.EL3.TPM == '1' then UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.EL2.T9 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch32.TakeHypTrapException(0x03);
else
AArch32.TakeHypTrapException(0x03);
else
AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
R[t] = PMCEID3;
else
PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR.EL3.TPM == '1' then UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.EL2.T9 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch32.TakeHypTrapException(0x03);
else
AArch32.TakeHypTrapException(0x03);
else
AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
R[t] = PMCEID3;
else
PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR.EL3.TPM == '1' then UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.EL2.T9 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch32.TakeHypTrapException(0x03);
else
AArch32.TakeHypTrapException(0x03);
else
AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
R[t] = PMCEID3;
if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
    R[t] = PMCEID3;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        else
            R[t] = PMCEID3;
    elsif PSTATE.EL == EL3 then
        R[t] = PMCEID3;
    else
        R[t] = PMCEID3;
## G8.4.7 PMCNTENCLR, Performance Monitors Count Enable Clear register

The PMCNTENCLR characteristics are:

**Purpose**

Disables the Cycle Count Register, PMCCNTR, and any implemented event counters PMEVCNTR\(<n>\). Reading this register shows which counters are enabled.

PMCNTENCLR is used in conjunction with the PMCNTENSET register.

**Configurations**

AArch32 System register PMCNTENCLR bits [31:0] are architecturally mapped to AArch64 System register PMCNTENCLR_EL0[31:0].

AArch32 System register PMCNTENCLR bits [31:0] are architecturally mapped to External register PMCNTENCLR_EL0[31:0].

This register is present only when AArch32 is supported and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCNTENCLR are UNDEFINED.

**Attributes**

PMCNTENCLR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>When read, means the cycle counter is disabled. When written, has no effect.</td>
</tr>
<tr>
<td>0b1</td>
<td>When read, means the cycle counter is enabled. When written, disables the cycle counter.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

<table>
<thead>
<tr>
<th>P&lt;(n&gt;), bit [n], for n = 30 to 0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Event counter disable bit for PMEVCNTR(&lt;n&gt;).</td>
<td></td>
</tr>
</tbody>
</table>

If \(N\) is less than 31, then bits [30:0] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, \(N\) is the value in MDCR_EL2.HPMN if EL2 is using AArch64, or in HDCR.HPMN if EL2 is using AArch32. Otherwise, \(N\) is the value in PMCR.N.

| 0b0 | When read, means that PMEVCNTR\(<n>\) is disabled. When written, has no effect. |
| 0b1 | When read, means that PMEVCNTR\(<n>\) is enabled. When written, disables PMEVCNTR\(<n>\). |

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Accessing PMCNTENCLR

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>}\ <\text{coproc}>, \{<opc1>\}, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>, \{<opc2>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL1) & PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    endif
  else
    UNDEFINED;
  endif
endif

if PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    UNDEFINED;
  endif
endif

if PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    endif
  else
    if EL2Enabled() & !ELUsingAArch32(EL2) & EDMEMgewater == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      UNDEFINED;
    endif
  endif
endif

if PSTATE.EL == EL3 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL2) & EDMEMgewater == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    UNDEFINED;
  endif
endif

R[t] = PMCNTENCLR;

else
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL2) & EDMEMgewater == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    UNDEFINED;
  endif
endif

R[t] = PMCNTENCLR;

else
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL2) & EDMEMgewater == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    UNDEFINED;
  endif
endif

R[t] = PMCNTENCLR;

else
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL2) & EDMEMgewater == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    UNDEFINED;
  endif
endif

R[t] = PMCNTENCLR;
when SDD == '1' && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end if;
else
    R[t] = PMCNTENCLR;
elsif PSTATE_EL == EL3 then
    R[t] = PMCNTENCLR;
end if;

\[ \text{MCR<q> coproc, (#opc1), <Rt>, <CRn>, <CRm>, (#opc2)} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
        end if;
    else
        UNDEFINED;
    end if;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
    AArch32.TakeHypTrapException(0x00);
else
    UNDEFINED;
end if;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end if;
else
    PMCNTENCLR = R[t];
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif !ELUsingAArch32(EL1) && PMUSERENR_EL1.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
        end if;
    else
        UNDEFINED;
    end if;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL1) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL1) && HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end if;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  end
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  end
else
  PMCNTENCLR = R[t];
elsif PSTATE_EL == EL3 then
  PMCNTENCLR = R[t];
elsif PSTATE_EL == EL3 then
  PMCNTENCLR = R[t];
G8.4.8 PMCNTENSET, Performance Monitors Count Enable Set register

The PMCNTENSET characteristics are:

**Purpose**

Enables the Cycle Count Register, PMCCNTR, and any implemented event counters PMEVCNTR<\textless n\textgreater>. Reading this register shows which counters are enabled.

 PMCNTENSET is used in conjunction with the PMCNTENCLR register.

**Configurations**

AArch32 System register PMCNTENSET bits \([31:0]\) are architecturally mapped to AArch64 System register PMCNTENSET_EL0\([31:0]\).

AArch32 System register PMCNTENSET bits \([31:0]\) are architecturally mapped to External register PMCNTENSET_EL0\([31:0]\).

This register is present only when AArch32 is supported and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCNTENSET are UNDEFINED.

**Attributes**

PMCNTENSET is a 32-bit register.

**Field descriptions**

![Field Diagram]

\(C\), bit \([31]\)

PMCCNTR enable bit. Enables the cycle counter register.

- \(0\): When read, means the cycle counter is disabled. When written, has no effect.
- \(1\): When read, means the cycle counter is enabled. When written, enables the cycle counter.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

\(P\textless n\textgreater\), bit \([n]\), for \(n = 30\) to \(0\)

Event counter enable bit for PMEVCNTR<\textless n\textgreater>.

If \(N\) is less than 31, then bits \([30:N]\) are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, \(N\) is the value in MDCR_EL2.HPMN if EL2 is using AArch64, or in HDCR.HPMN if EL2 is using AArch32. Otherwise, \(N\) is the value in PMCR.N.

- \(0\): When read, means that PMEVCNTR<\textless n\textgreater> is disabled. When written, has no effect.
- \(1\): When read, means that PMEVCNTR<\textless n\textgreater> event counter is enabled. When written, enables PMEVCNTR<\textless n\textgreater>.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Accessing PMCNTENSET

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}<\{q\}\} <\text{coproc}, \{<\text{opc}\}> <\text{Rt}>, <\text{CRn}>, <\text{CRm}>\{, \{<\text{opc2}\}>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elseif !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    endif
  elsif EL2Enabled() && PMUSERENR.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
      AArch32.TakeHypTrapException(0x00);
    else
      UNDEFINED;
    endif
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCNTEN == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    endif
  else
    R[t] = PMCNTENSET;
  endif
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elseif !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    endif
  elseif EL2Enabled() && PMUSERENR.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
      AArch32.TakeHypTrapException(0x00);
    else
      UNDEFINED;
    endif
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HDFGRTR_EL2.PMCNTEN == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    endif
  else
    R[t] = PMCNTENSET;
  endif
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1' && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        R[t] = PMCNTENSET;
elsif PSTATE_EL == EL3 then
    R[t] = PMCNTENSET;
end if;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            UNDEFINED;
        end if;
    elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
        end if;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        PMCNTENSET = R[t];
    end if;
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && HCR_EL2.TGE != '11' && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && HSC_EL2.TGE == '1' && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && HCR_EL2.TGE == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && HCR_EL2.TGE != '11' && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && HCR_EL2.TGE != '11' && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && HCR_EL2.TGE == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        PMCNTENSET = R[t];
    end if;
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        end
    else
        PMCNTENSET = R[t];
    end
elsif PSTATE.EL == EL3 then
    PMCNTENSET = R[t];
elsif PSTATE.EL == EL3 then
    PMCNTENSET = R[t];
**G8.4.9 PMCR, Performance Monitors Control Register**

The PMCR characteristics are:

** Purpose **
Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

** Configurations **
AArch32 System register PMCR bits [31:0] are architecturally mapped to AArch64 System register PMCR_EL0[31:0].
AArch32 System register PMCR bits [7:0] are architecturally mapped to External register PMCR_EL0[7:0].
This register is present only when AArch32 is supported and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCR are UNDEFINED.

** Attributes **
PMCR is a 32-bit register.

** Field descriptions **

<table>
<thead>
<tr>
<th>IMb</th>
<th>IDCODE</th>
<th>N</th>
<th>LP</th>
<th>LC</th>
<th>DP</th>
<th>X</th>
<th>D</th>
<th>C</th>
<th>P</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>28</td>
<td>23</td>
<td>16</td>
<td>15</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>RESO</td>
<td>RES0</td>
<td>FZO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

** IMP, bits [31:24] **

*When FEAT_PMUv3p7 is not implemented:*
Implementer code.
If this field is zero, then PMCR.IDCODE is RES0 and software must use MIDR to identify the PE. Otherwise, this field and PMCR.IDCODE identify the PMU implementation to software. The implementer codes are allocated by Arm. A non-zero value has the same interpretation as MIDR. Implementer.
Use of this field is deprecated.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

*Otherwise:*
Reserved, RAZ.

** IDCODE, bits [23:16] **

*When PMCR.IMP != 0b00000000000:*
Identification code. Use of this field is deprecated.
Each implementer must maintain a list of identification codes that are specific to the implementer. A specific implementation is identified by the combination of the implementer code and the identification code.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

*Otherwise:*
Reserved, RES0.
N, bits [15:11]

Indicates the number of event counters implemented. This value is in the range of 0b0000 - 0b1111. If the value is 0b0000, then only PMCCNTR is implemented. If the value is 0b1111, then PMCCNTR and 31 event counters are implemented.

In an implementation that includes EL2:
- If EL2 is using AArch32, reads of this field from Non-secure EL1 and Non-secure EL0 return the value of HDCR.HPMN.
- If EL2 is using AArch64 and is enabled in the current Security state, reads of this field from EL1 and EL0 return the value of MDCR_EL2.HPMN.

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Bit [10]

Reserved, RES0.

FZO, bit [9]

When FEAT_PMUv3p7 is implemented:

Freeze-on-overflow. Stop event counters on overflow.

In the description of this field:
- If EL2 is implemented and is using AArch32, PMN is HDCR.HPMN.
- If EL2 is implemented and is using AArch64, PMN is MDCR_EL2.HPMN.
- If EL2 is not implemented, PMN is PMCR.N.

0b0 Do not freeze on overflow.
0b1 Event counter PMEVCNTR<n> does not count when PMOVSR[(PMN-1):0] is nonzero and n is in the range of affected event counters.

If PMN is not 0, this field affects the operation of event counters in the range [0 .. (PMN-1)].
This field does not affect the operation of other event counters and PMCCNTR.
The operation of this field applies even when EL2 is disabled in the current Security state.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [8]

Reserved, RES0.

LP, bit [7]

When FEAT_PMUv3p5 is implemented:

Long event counter enable. Determines when unsigned overflow is recorded by an event counter overflow bit.

In the description of this field:
- If EL2 is implemented and is using AArch32, PMN is HDCR.HPMN.
- If EL2 is implemented and is using AArch64, PMN is MDCR_EL2.HPMN.
- If EL2 is not implemented, PMN is PMCR.N.

0b0 Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>[31:0].
0b1 Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>[63:0].
If the highest implemented Exception level is using AArch32, it is IMPLEMENTATION DEFINED whether this bit is RW or RAZ/WI.

If PMN is not 0, this bit affects the operation of event counters in the range [0 .. (PMN-1)].

This field does not affect the operation of other event counters and PMCCNTR. PMEVCNTR[n][63:32] cannot be accessed directly in AArch32 state.

The operation of this field applies even when EL2 is disabled in the current Security state.

The reset behavior of this field is:
  • On a Warm reset, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**LC, bit [6]**

Long cycle counter enable. Determines when unsigned overflow is recorded by the cycle counter overflow bit.

0b0  Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR[31:0].

0b1  Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR[63:0].

Arm deprecates use of PMCR.LC = 0.

The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**DP, bit [5]**

*When EL3 is implemented or (FEAT_PMUv3p1 is implemented and EL2 is implemented):*

Disable cycle counter when event counting is prohibited.

0b0  Cycle counting by PMCCNTR is not affected by this mechanism.

0b1  Cycle counting by PMCCNTR is disabled in prohibited regions and when event counting is frozen:

  • If FEAT_PMUv3p1 is implemented, EL2 is implemented, and HDCR.HPMD is 1, then cycle counting by PMCCNTR is disabled at EL2.

  • If FEAT_PMUv3p7 is implemented, EL3 is implemented and using AArch64, and MDCR_EL3.MPMX is 1, then cycle counting by PMCCNTR is disabled at EL3.

  • If FEAT_PMUv3p7 is implemented and event counting is frozen by PMCR.FZO, then cycle counting by PMCCNTR is disabled.

  • If EL3 is implemented, MDCR_EL3.SPME or SDCR.SPME is 0, and either FEAT_PMUv3p7 is not implemented, EL3 is using AArch32, or MDCR_EL3.MPMX is 0, then cycle counting by PMCCNTR is disabled at EL3 and in Secure state.

If HDCR.HPMN is not 0, this is when event counting by event counters in the range [0 .. (HDCR.HPMN-1)] is prohibited or frozen.

For more information, see *Prohibiting event and cycle counting* on page D11-5256.

The reset behavior of this field is:
  • On a Warm reset, this field resets to 0.

**Otherwise:**

Reserved, RES0.
X, bit [4]

When the implementation includes a PMU event export bus:

Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

0b0  Do not export events.
0b1  Export events where not prohibited.

This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device, for example to an OPTIONAL trace unit.

No events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

The reset behavior of this field is:

• On a Warm reset, this field resets to 0.

Otherwise:

Reserved, RAZ/WI.

D, bit [3]

Clock divider.

0b0  When enabled, PMCCNTR counts every clock cycle.
0b1  When enabled, PMCCNTR counts once every 64 clock cycles.

If PMCR.LC == 1, this bit is ignored and the cycle counter counts every clock cycle.

Arm deprecates use of PMCR.D = 1.

The reset behavior of this field is:

• On a Warm reset, this field resets to 0.

C, bit [2]

Cycle counter reset. The effects of writing to this bit are:

0b0  No action.
0b1  Reset PMCCNTR to zero.

Note

Resetting PMCCNTR does not change the cycle counter overflow bit. If FEAT_PMUv3p5 is implemented, the value of PMCR.LC is ignored, and bits [63:0] of the cycle counter are reset.

Access to this field is WO/RAZ.

P, bit [1]

Event counter reset.

In the description of this field:

• If EL2 is implemented and is using AArch32, PMN is HDCR.HPMN.
• If EL2 is implemented and is using AArch64, PMN is MDCR_EL2.HPMN.
• If EL2 is not implemented, PMN is PMCR.N.
0b0  No action.
0b1  If n is in the range of affected event counters, resets each event counter PMEVCNTR<n> to zero.

The effects of writing to this bit are:

• If EL2 is implemented and enabled in the current Security state, in EL0 and EL1, if PMN is not 0, a write of 1 to this bit resets event counters in the range [0 .. (PMN-1)].
• If EL2 is disabled in the current Security state, a write of 1 to this bit resets all the event counters.
In EL2 and EL3, a write of 1 to this bit resets all the event counters.

This field does not affect the operation of other event counters and PMCCNTR.

--- Note ---
Resetting the event counters does not change the event counter overflow bits.

If FEAT_PMUv3p5 is implemented, the values of HDCR.HLP and PMCR.LP are ignored and bits [63:0] of all affected event counters are reset.

Access to this field is WO/RAZ.

**E, bit [0]**

Enable.

In the description of this field:

- If EL2 is implemented and is using AArch32, PMN is HDCR.HPMN.
- If EL2 is implemented and is using AArch64, PMN is MDCR_EL2.HPMN.
- If EL2 is not implemented, PMN is PMCR.N.

0b0 PMCCNTR is disabled and event counters PMEVCNTR<n>, where n is in the range of affected event counters, are disabled.

0b1 PMCCNTR and event counters PMEVCNTR<n> where n is in the range of affected event counters, are enabled by PMCNTENSET.

If PMN is not 0, this field affects the operation of event counters in the range [0 .. (PMN-1)].

This field does not affect the operation of other event counters.

The operation of this field applies even when EL2 is disabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

**Accessing PMCR**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC{c}<q> <coproc>, {#<opc1>, <Rt>, <CRn>, <CRm>{, {#<opc2>}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCHR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAAarch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAAarch32(EL1) && PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && !ELUsingAAarch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  else
    UNDEFINED;
  endif
elsif EL2Enabled() && !ELUsingAAarch32(EL1) && PMUSERENR.EN == '0' then
  if EL2Enabled() && !ELUsingAAarch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && !ELUsingAAarch32(EL2) && HCR.TGE == '1' then
    AArch32.TakeHypTrapException(0x00);
  elseif ELUsingAAarch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAAarch32(EL2) && HSTR.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
  UNDEFINED;
endif

if EL2Enabled() && !ELUsingAAarch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
  UNDEFINED;
endif
AArch32.TakeHypTrapException(0x03);
elseif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMCR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HDCR.TPM == '1' then
  AArch32.TakeHypTrapException(0x03);
elseif EL2Enabled() && !ELUsingAArch32(EL2) && HDCR.TPMCR == '1' then
  AArch32.TakeHypTrapException(0x03);
elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  end
  endif
else
  R[t] = PMCR;
endif

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  else
    if Halted() && !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
      UNDEFINED;
    else
      AArch32.TakeHypTrapException(0x03);
    endif
  endif
else
  if PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    else
      if HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        endif
      else
        R[t] = PMCR;
      endif
    endif
  else
    R[t] = PMCR;
  endif
endif

if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPMCR == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPMCR == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMCR = R[t];
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TPMCR == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPMCR == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
    PMCR = R[t];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
    PMCR = R[t];
elsif PSTATE.EL == EL3 then
    PMCR = R[t];
G8.4.10 PMEVCNTR<n>, Performance Monitors Event Count Registers, n = 0 - 30

The PMEVCNTR<n> characteristics are:

**Purpose**

Holds event counter n, which counts events, where n is 0 to 30.

**Configurations**

AArch32 System register PMEVCNTR<n> bits [31:0] are architecturally mapped to AArch64 System register PMEVCNTR<n>_EL0[31:0].

AArch32 System register PMEVCNTR<n> bits [31:0] are architecturally mapped to External register PMEVCNTR<n>_EL0[31:0].

This register is present only when AArch32 is supported and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMEVCNTR<n> are UNDEFINED.

**Attributes**

PMEVCNTR<n> is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event counter n</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:0]**

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

If FEAT_PMUv3p5 is implemented, the event counter is 64 bits and only the least-significant part of the event counter is accessible in AArch32 state:

- Reads from PMEVCNTR<n> return bits [31:0] of the counter.
- Writes to PMEVCNTR<n> update bits [31:0] and leave bits [63:32] unchanged.
- There is no means to access bits [63:32] directly from AArch32 state.
- If the implementation does not support AArch64, bits [63:32] are not required to be implemented.

If FEAT_PMUv3p5 is not implemented, the event counter is 32 bits.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing PMEVCNTR<n>**

PMEVCNTR<n> can also be accessed by using PMXEVcntr with PMSELr.SEL set to n.

If FEAT_FGT is implemented and <n> is greater than or equal to the number of accessible event counters, then the behavior of permitted reads and writes of PMEVCNTR<n> is as follows:

- If <n> is an unimplemented event counter, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If FEAT_FGT is not implemented and <n> is greater than or equal to the number of accessible event counters, then reads and writes of PMEVCNTR<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if\(<n>\) is an\(\text{UNKNOWN}\) value less-than-or-equal-to the index of the highest accessible event counter.
- If EL2 is implemented and enabled in the current Security state, and \(<n>\) is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

---

**Note**

In EL0, an access is permitted if it is enabled by PMUSERENR.\{ER,EN\} or PMUSERENR_EL0.\{ER,EN\}.

If EL2 is implemented and enabled in the current Security state, at EL0 and EL1:

- If EL2 is using AArch32, HDCR.HPMN identifies the number of accessible event counters.
- If EL2 is using AArch64, MDCR_EL2.HPMN identifies the number of accessible event counters.

Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see HDCR.HPMN and MDCR_EL2.HPMN.

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}\{<q>\} <\text{coproc}>\{<\#>\text{opc1}>\}, <Rt>, <CRn>, <CRm>\{<\#>\text{opc2}>\}
\]

<table>
<thead>
<tr>
<th>(\text{coproc})</th>
<th>(\text{opc1})</th>
<th>(\text{CRn})</th>
<th>(\text{CRm})</th>
<th>(\text{opc2})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b10:m[4:3]</td>
<td>m[2:0]</td>
</tr>
</tbody>
</table>

integer \(m = \text{UInt(CRm}<1:0>:\text{opc2}<2:0>)\);

if \(m \geq \text{NUM\_PMU\_COUNTERS}\) then
  if IsFeatureImplemented(FEAT_FGT) then
    UNDEFINED;
  else
    ConstrainUnpredictableProcedure(Unpredictable\_PMUEVENTCOUNTER);
  endif
else
  if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    endif
  else
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch32.TakeHypTrapException(0x03);
    endif
  endif
else
  if EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    UNDEFINED;
  endif
else
  if EL2Enabled() && m >= AArch32.GetNumEventCountersAccessible() then
    if ELUsingAArch32(EL1) then
      AArch32.TakeHypTrapException(0x03);
    endif
  endif
endif
else
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    if Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
    R[t] = PMEVCNTR[m];
elsif PSTATE.EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
        UNDEFINED;
elsif EL2Enabled() & !ELUsingAArch32(EL2) & MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() & m >= AArch32.GetNumEventCountersAccessible() then
    if ELUsingAArch32(EL2) then
        AArch32.TakeHypTrapException(0x03);
    else
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif PSTATE.EL == EL2 then
        if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
            UNDEFINED;
        elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
            if Halted() & EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.AArch32SystemAccessTrap(EL3, 0x03);
            endif
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        endif
    else
        R[t] = PMEVCNTR[m];
    endif
elsif PSTATE.EL == EL0 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
        UNDEFINED;
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    if Halted() & EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        R[t] = PMEVCNTR[m];
    endif
else
    if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    endif
endif

MCR<coproc><CRn>, <CRm>, <opc1>, <Rt>, <opc2>

coproc opc1 CRn CRm opc2
0b1111 0b0000 0b1110 0b10:m[4:3] m[2:0]

integer m = UInt(CRm<1:0>:opc2<2:0>);
if m >= NUM_PMU_COUNTERS then
    if IsFeatureImplemented(FEAT_FGT) then
        UNDEFINED;
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    endif
elsif PSTATE.EL == EL0 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
        UNDEFINED;
elsif !ELUsingAArch32(EL1) & PMUSERENV_EL0.EN == '0' then
    if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    endif
endif
AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
        AArch32.TakeHypTrapException(0x00);
    else
        UNDEFINED;
    elsif EL2Enabled() && ELUsingAArch32(EL1) && SCR_EL1.FGTE == '1' && HCR_EL2.TGE != '11' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        UNDEFINED;
    end if;
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    if ELUsingAArch32(EL2) then
        AArch32.TakeHypTrapException(0x03);
    else
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    end if;
else
    UNDEFINED;
end if;
elsif EL3Enabled() && ELUsingAArch32(EL3) && SCR_EL3.FGTE == '1' then
    if ELUsingAArch32(EL3) then
        AArch32.TakeHypTrapException(0x03);
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end if;
else
    UNDEFINED;
end if;
PSTATE.EL == EL1 then
    if Halted() && ELUsingAArch32(EL1) then
        UNDEFINED;
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        UNDEFINED;
    end if;
PSTATE.EL == EL2 then
    if Halted() && ELUsingAArch32(EL2) then
        UNDEFINED;
    elsif EL3Enabled() && ELUsingAArch32(EL3) && HCR_EL3.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        UNDEFINED;
    end if;
PSTATE.EL == EL3 then
    if Halted() && ELUsingAArch32(EL3) then
        UNDEFINED;
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        UNDEFINED;
    end if;
PSTATE.EL = R[t];
else
    UNDEFINED;
end if;
G8.4.11 PMEVTYPER<n>, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n> characteristics are:

Purpose

Configures event counter n, where n is 0 to 30.

Configurations

AArch32 System register PMEVTYPER<n> bits [31:0] are architecturally mapped to AArch64 System register PMEVTYPER<n>_EL0[31:0].

AArch32 System register PMEVTYPER<n> bits [31:0] are architecturally mapped to External register PMEVTYPER<n>_EL0[31:0].

This register is present only when AArch32 is supported and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMEVTYPER<n> are UNDEFINED.

Attributes

PMEVTYPER<n> is a 32-bit register.

Field descriptions

The following table describes the bits in PMEVTYPER<n>:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Bit Description</th>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>Privileged filtering bit</td>
<td>P</td>
<td>Controls counting in EL1.</td>
</tr>
<tr>
<td>29</td>
<td>User filtering bit</td>
<td>U</td>
<td>Controls counting in EL0.</td>
</tr>
<tr>
<td>28</td>
<td>Non-secure EL1 (kernel) modes filtering bit</td>
<td>NSK</td>
<td>Controls counting in Non-secure EL1.</td>
</tr>
<tr>
<td>27</td>
<td>normals</td>
<td>RESO</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>remaining</td>
<td>RESO</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>event counter</td>
<td>evtCount[15:10]</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>event counter</td>
<td>evtCount[9:0]</td>
<td></td>
</tr>
</tbody>
</table>

P, bit [31]

Privileged filtering bit. Controls counting in EL1.

If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the PMEVTYPER<n>.NSK bit.

0b0 Count events in EL1.
0b1 Do not count events in EL1.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [30]

User filtering bit. Controls counting in EL0.

If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the PMEVTYPER<n>.NSU bit.

If FEAT_RME is implemented, then counting in Realm EL0 is further controlled by the PMEVTYPER<n>.RLU bit.

0b0 Count events in EL0.
0b1 Do not count events in EL0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1.

If the value of this bit is equal to the value of PMEVTYPER<n>.P, events in Non-secure EL1 are counted.

Otherwise, events in Non-secure EL1 are not counted.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**NSU, bit [28]**

*When EL3 is implemented:*

Non-secure EL0 (Unprivileged) filtering. Controls counting in Non-secure EL0.

If the value of this bit is equal to the value of PMEVTYPER<\(n\).U, events in Non-secure EL0 are counted.

Otherwise, events in Non-secure EL0 are not counted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**NSH, bit [27]**

*When EL2 is implemented:*

EL2 (Hyp mode) filtering bit. Controls counting in EL2.

- 0b0: Do not count events in EL2.
- 0b1: Count events in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bit [26]**

Reserved, RES0.

**MT, bit [25]**

*When FEAT_MTPMU is implemented or an IMPLEMENTATION DEFINED multi-threaded PMU extension is implemented:*

Multithreading.

- 0b0: Count events only on controlling PE.
- 0b1: Count events from any PE with the same affinity at level 1 and above as this PE.

From Armv8.6, the IMPLEMENTATION DEFINED multi-threaded PMU extension is not permitted, meaning if FEAT_MTPMU is not implemented, this bit is RES0. See ID_DFR1.MTPMU.

This bit is ignored by the PE and treated as zero when FEAT_MTPMU is implemented and Disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bits [24:22]**

Reserved, RES0.
RLU, bit [21]

When **FEAT_RME** is implemented:

Realm EL0 (unprivileged) filtering bit. Controls counting in Realm EL0.

If the value of this bit is equal to the value of the PMEVTYPER<n>.U bit, events in Realm EL0 are counted.

Otherwise, events in Realm EL0 are not counted.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

Otherwise:

Reserved, RES0.

Bits [20:16]

Reserved, RES0.

evtCount[15:10], bits [15:10]

When **FEAT_PMUv3p1** is implemented:

Extension to evtCount[9:0]. For more information, see evtCount[9:0].

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

Otherwise:

Reserved, RES0.

evtCount[9:0], bits [9:0]

Event to count.

The event number of the event that is counted by event counter PMEVCNTR<n>.

The ranges of event numbers allocated to each type of event are shown in Table D11-7 on page D11-5275.

If **FEAT_PMUv3p8** is implemented and PMEVTYPER<n>.evtCount is programmed to an event that is reserved or not supported by the PE, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>.evtCount field is the value written to the field.

—— Note

Arm recommends this behavior for all implementations of **FEAT_PMUv3**.

—— Note

Otherwise, if PMEVTYPER<n>.evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the value written:

• For the range 0x0000 to 0x003F, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>.evtCount field is the value written to the field.

• If **FEAT_PMUv3p1** is implemented, for the range 0x4000 to 0x403F, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>.evtCount field is the value written to the field.

• For other values, it is **UNPREDICTABLE** what event, if any, is counted and the value returned by a direct or external read of the PMEVTYPER<n>.evtCount field is **UNKNOWN**.

—— Note

**UNPREDICTABLE** means the event must not expose privileged information.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
Accessing PMEVTYPER<\text{n}>

PMEVTYPER<\text{n}> can also be accessed by using PMXEVTYPER with PMSELR.SEL set to \text{n}.

If \text{FEAT_FGT} is implemented and \text{<n>} is greater than or equal to the number of accessible event counters, then the behavior of permitted reads and writes of PMEVTYPER<\text{n}> is as follows:

- If \text{<n>} is an unimplemented event counter, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If \text{FEAT_FGT} is not implemented and \text{<n>} is greater than or equal to the number of accessible event counters, then reads and writes of PMEVTYPER<\text{n}> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if \text{<n>} is an UNKNOWN value less-than-or-equal-to the index of the highest accessible event counter.
- If EL2 is implemented and enabled in the current Security state, and \text{<n>} is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

\text{Note}

In EL0, an access is permitted if it is enabled by PMUSERENR.EN or PMUSERENR_EL0.EN.

If EL2 is implemented and enabled in the current Security state, at EL0 and EL1:

- If EL2 is using AArch32, HDCR.HPMN identifies the number of accessible event counters.
- If EL2 is using AArch64, MDCR_EL2.HPMN identifies the number of accessible event counters.

Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see HDCR.HPMN and MDCR_EL2.HPMN.

Accesses to this register use the following encodings in the System register encoding space:

\text{MRC\{<c>\}{<q>} <coproc>, \{#\}<opc1>, <Rt>, <CRn>, <CRm>{, \{#\}<opc2>}}

\begin{tabular}{|c|c|c|c|c|}
\hline
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
\hline
0b1111 & 0b000 & 0b1110 & 0b11:m[4:3] & m[2:0] \\
\hline
\end{tabular}

\begin{verbatim}
integer m = UInt(CRm<1:0>:opc2<2:0>);
if m >= NUM_PMU_COUNTERS then
  if IsFeatureImplemented(\text{FEAT_FGT}) then
    UNDEFINED;
  else
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
else
  PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSICR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1" then
      UNDEFINED;
    elsif PMUSERENR_EL0.EN == '0' then
      if EL2Enabled() && AArch64.AArch32SystemAccessTrap(EL2, 0x03); else
        AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
\end{verbatim}
elsif ELUsingAArch32(EL1) & PMUSERENR.EN == '0' then
  if EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
    AArch32.TakeHypTrapException(0x00);
  else
    UNDEFINED;
  endif
elsif EL2Enabled() & !ELUsingAArch32(EL1) & SCR_EL2.FGTE == '1' & HCR_EL2.TGE == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & MDCR_EL2.TPM == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HDCR.TPM == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() & m >= AArch32.GetNumEventCountersAccessible() then
  if ELUsingAArch32(EL1) then
    AArch32.TakeHypTrapException(0x03);
  else
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  endif
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
else
  R[t] = PMEVTYPER[m];
endif
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() & !ELUsingAArch32(EL2) & MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() & m >= AArch32.GetNumEventCountersAccessible() then
    if ELUsingAArch32(EL2) then
      AArch32.TakeHypTrapException(0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    endif
  else
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  endif
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
else
  R[t] = PMEVTYPER[m];
endif
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    endif
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
else
  R[t] = PMEVTYPER[m];
endif
elsif PSTATE.EL == EL3 then
  R[t] = PMEVTYPER[m];
integer m = UInt(CRm<1:0>:opc2<2:0>);

if m >= NUM_PMU_COUNTERS then
    if IsFeatureImplemented(FEAT_FGT) then
        UNDEFINED;
    else
        ConstraintUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    endif;
else
    PSTATE_EL == EL0 then
        if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MD_C_EL3.TPM == '1' then
            UNDEFINED;
        elseif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
        endif;
    else
        EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elseif EL2Enabled() & ELUsingAArch32(EL2) & HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x03);
        elseif EL2Enabled() & m >= AArch32.GetNumEventCountersAccessible() then
            if ELUsingAArch32(EL1) then
                AArch32.TakeHypTrapException(0x03);
            else
                AArch64.AArch32SystemAccessTrap(EL2, 0x03);
            endif;
        elseif HaveEL(EL3) & !ELUsingAArch32(EL3) & MD_C_EL3.TPM == '1' then
            if Halted() & EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.AArch32SystemAccessTrap(EL3, 0x03);
            endif;
        else
            PM_EVTYPER[m] = R[t];
        endif;
    else
        PSTATE_EL == EL1 then
            if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MD_C_EL3.TPM == '1' then
                UNDEFINED;
            elseif EL2Enabled() & ELUsingAArch32(EL2) & MD_C_EL2.TPM == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x03);
            elseif EL2Enabled() & ELUsingAArch32(EL2) & HCR.TPM == '1' then
                AArch32.TakeHypTrapException(0x03);
            elseif EL2Enabled() & m >= AArch32.GetNumEventCountersAccessible() then
                if ELUsingAArch32(EL2) then
                    AArch32.TakeHypTrapException(0x03);
                else
                    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
                endif;
            elseif HaveEL(EL3) & !ELUsingAArch32(EL3) & MD_C_EL3.TPM == '1' then
                if Halted() & EDSCR.SDD == '1' then
                    UNDEFINED;
                else
                    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
                endif;
            else
                UNDEFINED;
            endif;
        else
            UNDEFINED;
        endif;
    endif;
else
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
endif;
AArch64.AArch32SystemAccessTrap(EL3, 0x03);

else
    PMEVTYPER[m] = R[t];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMEVTYPER[m] = R[t];
elsif PSTATE.EL == EL3 then
    PMEVTYPER[m] = R[t];
G8.4.12 PMINTENCLR, Performance Monitors Interrupt Enable Clear register

The PMINTENCLR characteristics are:

**Purpose**

Disables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR, and the event counters PMEVCNTR<n>. Reading the register shows which overflow interrupt requests are enabled.

PMINTENCLR is used in conjunction with the PMINTENSET register.

**Configurations**

AArch32 System register PMINTENCLR bits [31:0] are architecturally mapped to AArch64 System register PMINTENCLR_EL1[31:0].

AArch32 System register PMINTENCLR bits [31:0] are architecturally mapped to External register PMINTENCLR_EL1[31:0].

This register is present only when EL1 is capable of using AArch32 and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMINTENCLR are UNDEFINED.

**Attributes**

PMINTENCLR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td><strong>PMCCNTR</strong> overflow interrupt request disable bit.</td>
</tr>
<tr>
<td></td>
<td>0:0 When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.</td>
</tr>
<tr>
<td></td>
<td>0:1 When read, means the cycle counter overflow interrupt request is enabled. When written, disables the cycle count overflow interrupt request.</td>
</tr>
<tr>
<td>P&lt;n&gt;</td>
<td><strong>PMEVCNTR&lt;n&gt;</strong> event counter overflow interrupt request disable bit for PMEVCNTR&lt;n&gt;.</td>
</tr>
</tbody>
</table>

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1, N is the value in MDCR_EL2.HPMN if EL2 is using AArch64, or in HDCR.HPMN if EL2 is using AArch32. Otherwise, N is the value in PMCR.N.

0:0 When read, means that the PMEVCNTR<n> event counter interrupt request is disabled. When written, has no effect.
When read, means that the PMEVCNTR<\text{n}> event counter interrupt request is enabled.
When written, disables the PMEVCNTR<\text{n}> interrupt request.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally \textbf{UNKNOWN} value.

### Accessing PMINTENCLR

Accesses to this register use the following encodings in the System register encoding space:

$$ MRC\{<c>\}{<q>\} \ <coproc>, \{#<opc1>, <Rt>, <CRn>, <CRm>, \{#<opc2>\}$$

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
elif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
elif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
elif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
elif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
elif PSTATE.EL == EL3 then
    R[t] = PMINTENCLR;
elif PSTATE.EL == EL0 then
    UNDEFINED;
else
    R[t] = PMINTENCLR;

### Accessing PMINTENCLR

Accesses to this register use the following encodings in the System register encoding space:

$$ MCR\{<c>\}{<q>\} \ <coproc>, \{#<opc1>, <Rt>, <CRn>, <CRm>, \{#<opc2>\}$$

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b010</td>
</tr>
</tbody>
</table>
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
      PMINTENCLR = R[t];
  elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
      else
        PMINTENCLR = R[t];
    elsif PSTATE_EL == EL3 then
      PMINTENCLR = R[t];
  else
    PMINTENCLR = R[t];

G8.4.13  PMINTENSET, Performance Monitors Interrupt Enable Set register

The PMINTENSET characteristics are:

Purpose

Enables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR, and the event counters PMEVCNTR<n>. Reading the register shows which overflow interrupt requests are enabled.

PMINTENSET is used in conjunction with the PMINTENCLR register.

Configurations

AArch32 System register PMINTENSET bits [31:0] are architecturally mapped to AArch64 System register PMINTENSET_EL1[31:0].

AArch32 System register PMINTENSET bits [31:0] are architecturally mapped to External register PMINTENSET_EL1[31:0].

This register is present only when EL1 is capable of using AArch32 and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMINTENSET are UNDEFINED.

Attributes

PMINTENSET is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>C, bit [31]</th>
<th>PMCCNTR overflow interrupt request enable bit.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.</td>
</tr>
<tr>
<td>0b1</td>
<td>When read, means the cycle counter overflow interrupt request is enabled. When written, enables the cycle count overflow interrupt request.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

<table>
<thead>
<tr>
<th>P&lt;n&gt;, bit [n], for n = 30 to 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event counter overflow interrupt request enable bit for PMEVCNTR&lt;n&gt;.</td>
</tr>
</tbody>
</table>

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1, N is the value in MDCR_EL2.HPMN if EL2 is using AArch64, or in HDRR.HPMN if EL2 is using AArch32. Otherwise, N is the value in PMCR.N.

<table>
<thead>
<tr>
<th>0b0</th>
<th>When read, means that the PMEVCNTR&lt;n&gt; event counter interrupt request is disabled. When written, has no effect.</th>
</tr>
</thead>
</table>
0b1 When read, means that the PMEVCNTR<n> event counter interrupt request is enabled. When written, enables the PMEVCNTR<n> interrupt request.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMINTENSET

Accesses to this register use the following encodings in the System register encoding space:

\[
\begin{array}{cccccc}
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
0b1111 & 0b000 & 0b1001 & 0b1110 & 0b001 \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
0b1111 & 0b000 & 0b1001 & 0b1110 & 0b001 \\
\end{array}
\]

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        else
            R[t] = PMINTENSET;
    elseif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
            UNDEFINED;
        elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.AArch32SystemAccessTrap(EL3, 0x03);
            else
                R[t] = PMINTENSET;
        elseif PSTATE.EL == EL3 then
            R[t] = PMINTENSET;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

\[
\begin{array}{cccccc}
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
0b1111 & 0b000 & 0b1001 & 0b1110 & 0b001 \\
\end{array}
\]

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif HaveEL(EL3) && ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
      PMINTENSET = R[t];
  elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elseif HaveEL(EL3) && ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
      else
        PMINTENSET = R[t];
  elsif PSTATE_EL == EL3 then
    PMINTENSET = R[t];
G8.4.14  PMOVSR, Performance Monitors Overflow Flag Status Register

The PMOVSR characteristics are:

**Purpose**

Contains the state of the overflow bit for the Cycle Count Register, PMCCNTR, and each of the implemented event counters PMEVCNTR<n>. Writing to this register clears these bits.

**Configurations**

AArch32 System register PMOVSR bits [31:0] are architecturally mapped to AArch64 System register PMOVSCLR_EL0[31:0].

AArch32 System register PMOVSR bits [31:0] are architecturally mapped to External register PMOVSCLR_EL0[31:0].

This register is present only when AArch32 is supported and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMOVSR are UNDEFINED.

**Attributes**

PMOVSR is a 32-bit register.

**Field descriptions**

C, bit [31]

Cycle counter overflow clear bit. Possible values are:

0b0  When read, means the cycle counter has not overflowed since this bit was last cleared. When written, has no effect.

0b1  When read, means the cycle counter has overflowed since this bit was last cleared. When written, clears the cycle counter overflow bit to 0.

PMCR.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR[31:0] or unsigned overflow of PMCCNTR[63:0].

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<n>, bit [n], for n = 30 to 0

Event counter overflow clear bit for PMEVCNTR<n>.

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, N is the value in MDCR_EL2.HPMN if EL2 is using AArch64, or in HDCR.HPMN if EL2 is using AArch32. Otherwise, N is the value in PMCR.N.

0b0  When read, means that PMEVCNTR<n> has not overflowed since this bit was last cleared. When written, has no effect.
0b1 When read, means that PMEVCNTR<n> has overflowed since this bit was last cleared. When written, clears the PMEVCNTR<n> overflow bit to 0.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP, HDCR.HLP, and PMCR.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<n>[31:0] or unsigned overflow of PMEVCNTR<n>[63:0]. PMEVCNTR<n>[63:32] cannot be accessed directly in AArch32 state.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

## Accessing PMOVSR

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b00</td>
<td>0b101</td>
<td>0b100</td>
<td>0b11</td>
</tr>
</tbody>
</table>

If PSTATE.EL == EL0 then

if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
  UNDEFINED;
elsif !ELUsingAArch32(EL1) & PMUSERENR_EL0.EN == '0' then
  if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif EL2Enabled() & PMUSERENR.EN == '0' then
  if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR.TGE == '1' then
  AArch32.TakeTrapException(EL1, 0x00);
else
  UNDEFINED;
elsif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE != '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.TGE != '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL1) & SCR_EL3.FGTEn == '1' & HDFGRTR_EL2.PMOVSR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & HSTR_EL2.T9 == '1' then
  AArch32.TakeTrapException(0x00);
elsif HSTR_EL2.T9 == '1' then
  AArch32.TakeTrapException(0x00);
elsif EL2Enabled() & !ELUsingAArch32(EL2) & MDCR_EL2.TPM == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & MDCR_EL2.TPM == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & HCR.TPM == '1' then
  AArch32.TakeTrapException(0x00);
elsif HSTR.T9 == '1' then
  AArch32.TakeTrapException(0x03);
elsif EL2Enabled() & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  R[t] = PMOVS;
elsif PSTATE.EL == EL1 then

if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
  UNDEFINED;
elsif EL2Enabled() & !ELUsingAArch32(EL2) & HSTR_EL2.T9 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T9 == '1' then
  AArch32.TakeTrapException(0x03);
elsif EL2Enabled() & !ELUsingAArch32(EL2) & MDCR_EL2.TPM == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
if Halted() && EDSCR.SDD == '1' then
  UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  R[t] = PMOVSR;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
      R[t] = PMOVSR;
  elsif PSTATE.EL == EL3 then
    if PSTATE.EL == EL0 then
      if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
      elsif !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
          AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
          AArch64.AArch32SystemAccessTrap(EL1, 0x03);
        end if
      elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
        if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
          AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
          AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
          AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
          UNDEFINED;
        end if
      elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
          AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.PMOVS == '1' then
          AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.PMOVS == '1' then
          AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.PMOVS == '1' then
          AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.PMOVS == '1' then
          AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elseif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.PMOVS == '1' then
          AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
          UNDEFINED;
        end if
      elsif EL2Enabled() && HAVEEL(EL3) && (SCF_EL3.FCTEN == '1') && HDFGWTR_EL2.PMOVSR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
      elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
      elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
      elsif HAVEEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        end if
      else
        PMOVSR = R[t];
    end if
  end if
end if
end if

MCR<{coproc}> <opc1>, <Rt>, <CRn>, <CRm>, <opc2>

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1100</td>
<td>0b011</td>
</tr>
</tbody>
</table>
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end
  else
    PMOVSR = R[t];
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end
  else
    PMOVSR = R[t];
  end
elsif PSTATE.EL == EL3 then
  PMOVSR = R[t];
G8.4.15 PMOVSSET, Performance Monitors Overflow Flag Status Set register

The PMOVSSET characteristics are:

**Purpose**

Sets the state of the overflow bit for the Cycle Count Register, PMCCNTR, and each of the implemented event counters PMEVCNTR\(<n>\).

**Configurations**

AArch32 System register PMOVSSET bits [31:0] are architecturally mapped to AArch64 System register PMOVSSET_EL0[31:0].

AArch32 System register PMOVSSET bits [31:0] are architecturally mapped to External register PMOVSSET_EL0[31:0].

This register is present only when AArch32 is supported and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMOVSSET are UNDEFINED.

**Attributes**

PMOVSSET is a 32-bit register.

**Field descriptions**

**C, bit [31]**

Cycle counter overflow set bit.

- **0b0** When read, means the cycle counter has not overflowed since this bit was last cleared. When written, has no effect.
- **0b1** When read, means the cycle counter has overflowed since this bit was last cleared. When written, sets the cycle counter overflow bit to 1.

**PMCR.LC** controls whether an overflow is detected from unsigned overflow of PMCCNTR[31:0] or unsigned overflow of PMCCNTR[63:0].

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**P\(<n>\), bit [n], for n = 30 to 0**

Event counter overflow set bit for PMEVCNTR\(<n>\).

If N is less than 31, then bits [30:N] are RAZ/WI. When EL2 is implemented and enabled in the current Security state, in EL1 and EL0, N is the value in MDCR_EL2.HPMN if EL2 is using AArch64, or in HDCR.HPMN if EL2 is using AArch32. Otherwise, N is the value in PMCR.N.

- **0b0** When read, means that PMEVCNTR\(<n>\) has not overflowed since this bit was last cleared. When written, has no effect.
When read, means that PMEVCNTR<n> has overflowed since this bit was last. When written, sets the PMEVCNTR<n> overflow bit to 1.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP, HDCR.HLP, and PMCR.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<n>[31:0] or unsigned overflow of PMEVCNTR<n>[63:0]. PMEVCNTR<n>[63:32] cannot be accessed directly in AArch32 state.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing PMOVSSET

Accesses to this register use the following encodings in the System register encoding space:

$$MRC\{<c>\}{<q>} <coproc>, \{<opc1>, <Rt>, <CRn>, <CRm}\{, \{<opc2}\}$

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    end
  elsif EL2Enabled() && PMUSERENR.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
      AArch32.TakeHypTrapException(0x00);
    else
      UNDEFINED;
    end
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end
  else
    R[t] = PMOVSSET;
  end
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end
  else
    R[t] = PMOVSSET;
  end
endif
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HDCR.TPM == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
  if Halted() & EDSCR.SDD == '1' then
  UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  R[t] = PMOVSSET;
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
  UNDEFINED;
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
  if Halted() & EDSCR.SDD == '1' then
  UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  R[t] = PMOVSSET;
elsif PSTATE.EL == EL3 then
  R[t] = PMOVSSET;

MCR<coproc>,<CRn>,<CRm>,<opc1>,<opc2>

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
  UNDEFINED;
elsif !ELUsingAArch32(EL1) & PMUSERENR_EL0.EN == '0' then
  if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif ELUsingAArch32(EL1) & PMUSERENR.EN == '0' then
  if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
    AArch32.TakeHypTrapException(0x03);
else
  UNDEFINED;
elsif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.<E2H,TGE> != '11' & HSTR_EL2.T9 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.<E2H,TGE> == '11' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() & ELUsingAArch32(EL1) & HCR_EL2.<E2H,TGE> != '11' & (HaveEL(EL3) || SCR_EL3.FCTEN == '1') & HFGWTR_EL2.PMOV == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & !ELUsingAArch32(EL2) & MDCR_EL2.TPM == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & MDCR_EL2.TPM == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HDCR.TPM == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
  if Halted() & EDSCR.SDD == '1' then
  UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  PMOVSSET = R[t];
elsif PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        else
            PMOVSET = R[t];
    elsif PSTATE_EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
            UNDEFINED;
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.AArch32SystemAccessTrap(EL3, 0x03);
            else
                PMOVSET = R[t];
        elsif PSTATE_EL == EL3 then
            PMOVSET = R[t];
    else
        PMOVSET = R[t];
    endif
else
    PMOVSET = R[t];
endif
G8.4.16 PMSELR, Performance Monitors Event Counter Selection Register

The PMSELR characteristics are:

Purpose

Selects the current event counter PMEVCNTR<\text{n}> or the cycle counter, CCNT. PMSELR is used in conjunction with PMXEVTYPER to determine the event that increments a selected event counter, and the modes and states in which the selected counter increments. It is also used in conjunction with PMXEVCNTR, to determine the value of a selected event counter.

Configurations

AArch32 System register PMSELR bits [31:0] are architecturally mapped to AArch64 System register PMSELR_EL0[31:0].

This register is present only when AArch32 is supported and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMSELR are UNDEFINED.

Attributes

PMSELR is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>4:0</td>
<td>SEL, selects event counter PMEVCNTR&lt;\text{n}&gt; where \text{n} is the value held in this field. This value identifies which event counter is accessed when a subsequent access to PMXEVTYPER or PMXEV CNTR occurs. This field can take any value from 0 (0b00000) to (PMCR.N)-1, or 31 (0b11111). When PMSELR.SEL is 0b11111, it selects the cycle counter and:</td>
</tr>
<tr>
<td></td>
<td>• A read of the PMXEVTYPER returns the value of PMCCFILTR.</td>
</tr>
<tr>
<td></td>
<td>• A write of the PMXEVTYPER writes to PMCCFILTR.</td>
</tr>
<tr>
<td></td>
<td>• A read or write of PMXEV CNTR has CONSTRAINED UNPREDICTABLE effects. For more information, see PMXEV CNTR.</td>
</tr>
<tr>
<td></td>
<td>For more information about the results of accesses to event counters, see PMXEVTYPER and PMXEV CNTR.</td>
</tr>
<tr>
<td></td>
<td>For more information about the number of counters accessible at each Exception level, see HDR.C.HPMN and MDCR_EL2.C.HPMN.</td>
</tr>
<tr>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
</tbody>
</table>

Accessing PMSELR

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif EL2Enabled() && PMUSERENR_EL0.E<ER,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        endif
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        R[t] = PMSELR;
    endif
elif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif EL2Enabled() && PMUSERENR_EL1.E<ER,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        endif
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        endif
    else
        R[t] = PMSELR;
    endif
else
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        endif
    else
        R[t] = PMSELR;
    endif
endif

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b000</td>
<td>0b101</td>
<td>0b100</td>
<td>0b101</td>
</tr>
</tbody>
</table>
MCR<coproc>, {#<opc1>, <Rt>, <CRn>, <CRm>}, {#<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b100</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        R[t] = PMSELR;
elsif PSTATE_EL == EL3 then
    R[t] = PMSELR;

if PSTATE_EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif !ELUsingAArch32(EL1) && PMUSERENR_EL0.<ER,EN> == '00' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
        elsif EL2Enabled() && PMUSERENR.<ER,EN> == '00' then
            if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
                AArch32.TakeHypTrapException(0x03);
            else
                UNDEFINED;
            elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x03);
            elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
                AArch32.TakeHypTrapException(0x03);
            elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
                if Halted() && EDSCR.SDD == '1' then
                    UNDEFINED;
                else
                    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
                else
                    PMSELR = R[t];
    elsif PSTATE_EL == EL1 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
            UNDEFINED;
        else
            if EL2Enabled() && !ELUsingAArch32(EL3) && PMUSERENR_EL0.<ER,EN> == '00' then
                if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
                else
                    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
                elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
                elsif EL2Enabled() && PMUSERENR.<ER,EN> == '00' then
                    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
                    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
                        AArch32.TakeHypTrapException(0x03);
                    elsif EL2Enabled() && !ELUsingAArch32(EL1) && HSTR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
                        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
                    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
                        AArch32.TakeHypTrapException(0x03);
                    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
                        if Halted() && EDSCR.SDD == '1' then
                            UNDEFINED;
                        else
                            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
                        else
                            PMSELR = R[t];
AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
    PMSELR = R[t];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        end if
    else
        PMSELR = R[t];
    end if
elsif PSTATE.EL == EL3 then
    PMSELR = R[t];
G8.4.17 PMSWINC, Performance Monitors Software Increment register

The PMSWINC characteristics are:

**Purpose**

Increments a counter that is configured to count the Software increment event, event 0x0. For more information, see SW_INCR.

**Configurations**

AArch32 System register PMSWINC bits [31:0] are architecturally mapped to AArch64 System register PMSWINC_EL0[31:0].

AArch32 System register PMSWINC bits [31:0] are architecturally mapped to External register PMSWINC_EL0[31:0].

This register is present only when AArch32 is supported and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMSWINC are UNDEFINED.

**Attributes**

PMSWINC is a 32-bit register.

**Field descriptions**

![Register Diagram]

**Bit [31]**

Reserved, RES0.

**P<\text{n}>, bit [\text{n}], for \text{n} = 30 to 0**

Event counter software increment bit for PMEVCNTR<\text{n}>.

If N is less than 31, then bits [30:N] are WE. When EL2 is implemented and enabled in the current Security state, EL1 and EL0, N is the value in MDCR_EL2.HPMN if EL2 is using AArch64, or in HDCR.HPMN if EL2 is using AArch32. Otherwise, N is the value in PMCR.N.

0b0 No action. The write to this bit is ignored.

0b1 If PMEVCNTR<\text{n}> is enabled and configured to count the software increment event, increments PMEVCNTR<\text{n}> by 1. If PMEVCNTR<\text{n}> is disabled, or not configured to count the software increment event, the write to this bit is ignored.

**Accessing PMSWINC**

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elif !EL2Enabled() && PMUSERENR_EL0.<SW,EN> == '00' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    endif
  else
    UNDEFINED;
  endif
else
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
    AArch32.TakeHypTrapException(0x03);
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
endif
UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap[EL3, 0x03];
else
  PMSWINC = R[t];
elsif PSTATE.EL == EL3 then
  PMSWINC = R[t];
G8.4.18 PMUSERENR, Performance Monitors User Enable Register

The PMUSERENR characteristics are:

**Purpose**

Enables or disables User mode access to the Performance Monitors.

**Configurations**

AArch32 System register PMUSERENR bits [31:0] are architecturally mapped to AArch64 System register PMUSERENR_EL0[31:0].

This register is present only when AArch32 is supported and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMUSERENR are UNDEFINED.

**Attributes**

PMUSERENR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>3</td>
<td>ER, bit [3] Event counter read trap control:</td>
</tr>
<tr>
<td></td>
<td>0b0 EL0 reads of the PMXEVCNTR and PMEVCNTR&lt;n&gt;, and EL0 RW access to the PMSELR, are trapped to Undefined mode if PMUSERENR.EN is also 0.</td>
</tr>
<tr>
<td></td>
<td>0b1 Overrides PMUSERENR.EN and enables RO access to PMXEVCNTR and PMEVCNTR&lt;n&gt;, and RW access to PMSELR.</td>
</tr>
<tr>
<td>2</td>
<td>CR, bit [2] Cycle counter read trap control:</td>
</tr>
<tr>
<td></td>
<td>0b0 EL0 reads of the PMCCNTR are trapped to Undefined mode if PMUSERENR.EN is also 0.</td>
</tr>
<tr>
<td></td>
<td>0b1 Overrides PMUSERENR.EN and enables access to PMCCNTR.</td>
</tr>
<tr>
<td>1</td>
<td>SW, bit [1] Software increment write trap control:</td>
</tr>
<tr>
<td></td>
<td>0b0 EL0 writes to the PMSWINC are trapped to Undefined mode if PMUSERENR.EN is also 0.</td>
</tr>
<tr>
<td></td>
<td>0b1 Overrides PMUSERENR.EN and enables access to PMSWINC.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.
EN, bit [0]

Traps EL0 accesses to the Performance Monitors registers to Undefined mode, as follows:

- PMCR, PMOVSR, PMSELR, PMCEID0, PMCEID1, PMCCNTR, PMXEVTYPEPR, PMXEVCTR, PMCNTENSET, PMCNTENCLR, PMOVSSET, PMEVCNTR<\n>, PMXEVTYPEPR<\n>, PMCCFILTR, PMSWINC.
- If FEAT_PMUv3p1 is implemented, PMCEID2, and PMCEID3.
- If FEAT_PMUv3p4 is implemented, PMMIR.

0b0 While at EL0, accesses to the specified registers at EL0 are trapped to Undefined mode, unless overridden by one of PMUSERENR.{ER, CR, SW}.
0b1 While at EL0, software can access all of the specified registers.

The reset behavior of this field is:
- On a Warm reset, this field resets to 0.

### Accessing PMUSERENR

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}^{<c>}{<q>\text{ <coproc}, \{#<opc1>, <Rt>, <CRn>, <CRm>{, \{#<opc2}\}}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.<E2H,TGE> != '11' & HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T9 == '1' then
    AArch32.TakeHypTrapException(0x83);
elsif EL2Enabled() & !ELUsingAArch32(EL3) & HCR_EL2.<E2H,TGE> != '11' & (!HaveEL(EL3) || SCR_EL3.FGTR == '1') & HDFGRTR_EL2.PMUSERENR_EL0 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & MDCR_EL2.TPM == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HDCR.TPM == '1' then
  AArch32.TakeHypTrapException(0x83);
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
  if Halted() & EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  except
    if Halted() & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elsif EL2Enabled() & !ELUsingAArch32(EL2) & HSTR_EL2.T9 == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T9 == '1' then
      AArch32.TakeHypTrapException(0x83);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & MDCR_EL2.TPM == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HDCR.TPM == '1' then
      AArch32.TakeHypTrapException(0x83);
    elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
      if Halted() & EDSCR.SDD == '1' then

if PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() & !ELUsingAArch32(EL2) & HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T9 == '1' then
    AArch32.TakeHypTrapException(0x83);
elsif EL2Enabled() & !ELUsingAArch32(EL2) & MDCR_EL2.TPM == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HDCR.TPM == '1' then
  AArch32.TakeHypTrapException(0x83);
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
  if Halted() & EDSCR.SDD == '1' then

UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  R[t] = PMUSERENR;
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
      R[t] = PMUSERENR;
  elsif PSTATE_EL == EL3 then
    R[t] = PMUSERENR;

MCR{<c>{<q>}}<coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1110</td>
<td>0b000</td>
</tr>
</tbody>
</table>
G8.4.19  PMXEVCNTR, Performance Monitors Selected Event Count Register

The PMXEVCNTR characteristics are:

Purpose
Reads or writes the value of the selected event counter, PMEVCNTR<n>. PMSELR.SEL determines which event counter is selected.

Configurations
AArch32 System register PMXEVCNTR bits [31:0] are architecturally mapped to AArch64 System register PMXEVCNTR_EL0[31:0].
This register is present only when AArch32 is supported and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMXEVCNTR are UNDEFINED.

Attributes
PMXEVCNTR is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMEVCNTR&lt;n&gt;</td>
<td></td>
</tr>
</tbody>
</table>

PMEVCNTR<n>, bits [31:0]
Value of the selected event counter, PMEVCNTR<n>, where n is the value stored in PMSELR.SEL.
If FEAT_PMUv3p5 is implemented, the event counter is 64 bits and only the least-significant part of the event counter is accessible in AArch32 state:
• Reads from PMXEVCNTR return bits [31:0] of the counter.
• Writes to PMXEVCNTR update bits [31:0] and leave bits [63:32] unchanged.
• There is no means to access bits [63:32] directly from AArch32 state.
• If the implementation does not support AArch64, bits [63:32] are not required to be implemented.
If FEAT_PMUv3p5 is not implemented, the event counter is 32 bits.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMXEVCNTR
If FEAT_FGT is implemented and PMSELR.SEL is greater than or equal to the number of accessible event counters, then the behavior of permitted reads and writes of PMXEVCNTR is as follows:
• If PMSELR.SEL selects an unimplemented event counter, the access is UNDEFINED.
• Otherwise, the access is trapped to EL2.
If FEAT_FGT is not implemented and PMSELR.SEL is greater than or equal to the number of accessible event counters, then reads and writes of PMXEVCNTR are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:
• Accesses to the register are UNDEFINED.
• Accesses to the register behave as RAZ/WI.
• Accesses to the register execute as a NOP
• Accesses to the register behave as if PMSEL.R.SEL has an UNKNOWN value less than the number of event counters accessible at the current Exception level and Security state.

• If EL2 is implemented and enabled in the current Security state, and PMSEL.R.SEL is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR.ER or PMUSERENR_EL0.ER.

If EL2 is implemented and enabled in the current Security state, at EL0 and EL1:

• If EL2 is using AArch32, HDCR.HPMN identifies the number of accessible event counters.

• If EL2 is using AArch64, MDCR_EL2.HPMN identifies the number of accessible event counters.

Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see HDCR.HPMN and MDCR_EL2.HPMN.

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC<coproc><CRn><CRm> <coproc>, CRn>, <CRm>}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1101</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if UInt(PMSEL.R.SEL) >= NUM_PMU_COUNTERS then
    if IsFeatureImplemented(FEAT_FGT) then
        UNDEFINED;
    else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    elsif PSTATE.EL == EL0 then
        if Halted() && HaveEL(EL3) && EDSCR.SOD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SOD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
            UNDEFINED;
        elsif :ELUsingAArch32(EL1) && PMUSERENR_EL0.ER == '00' then
            if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x03);
            else
                AArch64.AArch32SystemAccessTrap(EL1, 0x03);
            endif
        else
            UNDEFINED;
        endif
    else
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            UNDEFINED;
        endif
    endif
else
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR.TGE == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
    endif
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  R[t] = PMEVCNTR[UInt(PMSELR.SEL)];
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T9 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
  if EL2Enabled() && UInt(PMSELR.SEL) >= AArch32.GetNumEventCountersAccessible() then
    if ELUsingAArch32(EL2) then
      AArch32.TakeHypTrapException(0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && UInt(PMSELR.SEL) >= NUM_PMU_COUNTERS then
      if IsFeatureImplemented(FEAT_FGT) then
        UNDEFINED;
      else
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
      elseif PSTATE_EL == EL0 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
          UNDEFINED;
        else
          AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        elseif PSTATE_EL == EL3 then
          R[t] = PMEVCNTR[UInt(PMSELR.SEL)];
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  R[t] = PMEVCNTR[UInt(PMSELR.SEL)];
elsif PSTATE_EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
  R[t] = PMEVCNTR[UInt(PMSELR.SEL)];
else
  R[t] = PMEVCNTR[UInt(PMSELR.SEL)];

MCR<coproc><CRn><CRm> opc1 opc2

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0000</td>
<td>0b1001</td>
<td>0b1101</td>
<td>0b010</td>
</tr>
</tbody>
</table>
AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR.TGE == '1' then
  AArch32.TakeHypTrapException(0x00);
else
  UNDEFINED;
elsif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.<E2H,TGE> != '11' & HSTR_EL2.T9 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T9 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() & UInt(PMSELR.SEL) >= AArch32.GetNumEventCountersAccessible() then
  if ELUsingAArch32(EL1) then
    AArch32.TakeHypTrapException(0x03);
  else
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
    if Halted() & EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
      PMEVCTR(UInt(PMSELR.SEL)) = R[t];
  elseif PSTATE.EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
      UNDEFINED;
elsif EL2Enabled() & !ELUsingAArch32(EL2) & HSTR_EL2.<E2H,TGE> != '11' & HSTR_EL2.T9 == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T9 == '1' then
      AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() & UInt(PMSELR.SEL) >= AArch32.GetNumEventCountersAccessible() then
      if ELUsingAArch32(EL2) then
        AArch32.TakeHypTrapException(0x03);
      else
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
      elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
        if Halted() & EDSCR.SDD == '1' then
          UNDEFINED;
        else
          AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        else
          PMEVCTR(UInt(PMSELR.SEL)) = R[t];
      elsif PSTATE.EL == EL3 then
        if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
          UNDEFINED;
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & MDCR_EL3.TPM == '1' then
          if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
          else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
          else
            PMEVCTR(UInt(PMSELR.SEL)) = R[t];
          }
elsif PSTATE.EL == EL3 then
    PMEVCNTR[UInt(PMSELR.SEL)] = R[t];
G8.4.20 PMXEVTYPER, Performance Monitors Selected Event Type Register

The PMXEVTYPER characteristics are:

**Purpose**

When PMSELR.SEL selects an event counter, this accesses a PMEVTYPER<n> register. When PMSELR.SEL selects the cycle counter, this accesses PMCCFILTR.

**Configurations**

AArch32 System register PMXEVTYPER bits [31:0] are architecturally mapped to AArch64 System register PMXEVTYPER_EL0[31:0].

This register is present only when AArch32 is supported and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMXEVTYPER are UNDEFINED.

**Attributes**

PMXEVTYPER is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event type register or PMCCFILTR</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:0]**

Event type register or PMCCFILTR.

When PMSELR.SEL == 31, this register accesses PMCCFILTR.

Otherwise, this register accesses PMEVTYPER<n> where n is the value in PMSELR.SEL.

**Accessing PMXEVTYPER**

If FEAT_FGT is implemented, and PMSELR.SEL is not 31 and is greater than or equal to the number of accessible event counters, then the behavior of permitted reads and writes of PMXEVTYPER is as follows:

- If PMSELR.SEL selects an unimplemented event counter, the access is UNDEFINED.
- Otherwise, the access is trapped to EL2.

If FEAT_FGT is not implemented, and PMSELR.SEL is not 31 and is greater than or equal to the number of accessible event counters, then reads and writes of PMXEVTYPER are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a **NOP**
- Accesses to the register behave as if PMSELR.SEL has an **UNKNOWN** value less than the number of event counters accessible at the current Exception level and Security state.
- Accesses to the register behave as if PMSELR.SEL is 31.
- If EL2 is implemented and enabled in the current Security state, and PMSELR.SEL is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

**Note**

In EL0, an access is permitted if it is enabled by PMUSERENR.EN or PMUSERENR_EL0.EN.
If EL2 is implemented and enabled in the current Security state, at EL0 and EL1:

- If EL2 is using AArch32, HDCR.HPMN identifies the number of accessible event counters.
- If EL2 is using AArch64, MDCR_EL2.HPMN identifies the number of accessible event counters.

Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see HDCR.HPMN and MDCR_EL2.HPMN.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} {<coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1001</td>
<td>0b1010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if UInt(PMSELR.SEL) != 31 && UInt(PMSELR.SEL) >= NUM_PMU_COUNTERS then
  if IsFeatureImplemented(FEAT_FGT) then
    UNDEFINED;
  else
    ConstrainUnpredictableProcedure(Unpredictable_PMU_EVENTCOUNTER);
else
  if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'' && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      UNDEFINED;
    elsif PSTATE.EL == EL1 then
      if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
      elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
      else
        UNDEFINED;
      end
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
      AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL1) && MDCR_EL2.TPM == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
      AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && UInt(PMSELR.SEL) != 31 && UInt(PMSELR.SEL) >=
      AArch32.GetNumEventCountersAccessible() then
      if ELUsingAArch32(EL1) then
        AArch32.TakeHypTrapException(0x03);
      else
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
      end
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
      end
    else
      if UInt(PMSELR.SEL) == 31 then
        R[t] = PMCCFILTR;
      else
        UNDEFINED;
      end
    end
  end
end
R[t] = PMEVTYPER(UInt(AArch32-PMSELR.SEL));
elsf PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    R[t] = PMCCFILTR;
  else
    R[t] = PMEVTYPER(UInt(AArch32-PMSELR.SEL));
elsf PSTATE_EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    R[t] = PMCCFILTR;
  else
    R[t] = PMEVTYPER(UInt(AArch32-PMSELR.SEL));

coproccop1CRnCRmopc2
0b1111 0b000 0b1001 0b1101 0b001

if UInt(PMSELR.SEL) != 31 && UInt(PMSELR.SEL) >= NUM_PMU_COUNTERS then
  if IsFeatureImplemented(FEAT_FGT) then
    UNDEFINED;
  else
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsf PSTATE_EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
    UNDEFINED;
elsif !ELUsingAArch32(EL1) && PMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    endif
elsif ELUsingAArch32(EL1) && PMUSERENR.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && HSTR_EL2.T9 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T9 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TPM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif EL2Enabled() && UInt(PMSELR.SEL) != 31 && UInt(PMSELR.SEL) >= AArch32.GetNumEventCountersAccessible() then
        if ELUsingAArch32(EL1) then
            AArch32.TakeHypTrapException(0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        endif
    elseif PSTATE.EL == EL1 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        else
            PMEVTYPER[UInt(AArch32-PMSELR.SEL)] = R[t];
        endif
    elsif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        else
            PMEVTYPER[UInt(AArch32-PMSELR.SEL)] = R[t];
        endif
    else
        if UInt(PMSELR.SEL) == 31 then
            PMCCFILTR = R[t];
        else
            PMEVTYPER[UInt(AArch32-PMSELR.SEL)] = R[t];
        endif
    endif
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMEVTYPER[UInt(AArch32-PMSELR.SEL)] = R[t];
    endif
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMEVTYPER[UInt(AArch32-PMSELR.SEL)] = R[t];
    endif
elsif PSTATE.EL == EL3 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && MDCR_EL3.TPM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        PMEVTYPER[UInt(AArch32-PMSELR.SEL)] = R[t];
    endif
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
  UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TPM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  end
else
  if UInt(PMSELR.SEL) == 31 then
    PMCCFILTR = R[t];
  else
    PMEVTYPE[UInt(AArch32-PMSELR.SEL)] = R[t];
  end
elsif PSTATE.EL == EL3 then
  if UInt(PMSELR.SEL) == 31 then
    PMCCFILTR = R[t];
  else
    PMEVTYPE[UInt(AArch32-PMSELR.SEL)] = R[t];
end
G8.5  Activity Monitors registers

This section lists the Activity Monitoring registers in AArch32.
G8.5.1 AMCFGR, Activity Monitors Configuration Register

The AMCFGR characteristics are:

Purpose

Global configuration register for the activity monitors.

Provides information on supported features, the number of counter groups implemented, the total number of activity monitor event counters implemented, and the size of the counters. AMCFGR is applicable to both the architected and the auxiliary counter groups.

Configurations

AArch32 System register AMCFGR bits [31:0] are architecturally mapped to AArch64 System register AMCFGR_EL0[31:0].

AArch32 System register AMCFGR bits [31:0] are architecturally mapped to External register AMCFGR[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCFGR are UNDEFINED.

Attributes

AMCFGR is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCG</td>
<td>Defines the number of counter groups. The number of implemented counter groups is [AMCFGR.NCG + 1]. If the number of implemented auxiliary activity monitor event counters is zero, this field has a value of 0b0000. Otherwise, this field has a value of 0b0001. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.</td>
</tr>
<tr>
<td>RES0</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>HDBG</td>
<td>Halt-on-debug supported. This feature must be supported, and so this bit is 0b1. 0b0 AMCR.HDBG is RES0. 0b1 AMCR.HDBG is read/write. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.</td>
</tr>
<tr>
<td>RAZ</td>
<td>Reserved, RAZ.</td>
</tr>
<tr>
<td>SIZE</td>
<td>Defines the size of activity monitor event counters.</td>
</tr>
</tbody>
</table>

NCG, bits [31:28]

HDBG, bit [24]
The size of the activity monitor event counters implemented by the Activity Monitors Extension is \([\text{AMCFGR.SIZE} + 1]\). The counters are 64-bit.

--- Note ---
Software also uses this field to determine the spacing of counters in the memory-map. The counters are at doubleword-aligned addresses.

---
Reads as 0b1111111.
Access to this field is RO.

**N, bits [7:0]**
Defines the number of activity monitor event counters.
The total number of counters implemented in all groups by the Activity Monitors Extension is \([\text{AMCFGR.N} + 1]\).
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

### Accessing AMCFGR
Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{c\}\{q\} <\text{coproc}, \{\#\}<\text{opc1}, <\text{Rt}, <\text{CRn}, <\text{CRm}\}, \{\#\}<\text{opc2}>
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0010</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == 0 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TAM == '1' then
UNDEFINED;
elsif !ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif !ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && HCR_EL2.TGE == '1' then
AArch32.TakeHypTrapException(0x00);
else
UNDEFINED;
elsif !ELUsingAArch32(EL2) && !EL2Enabled() && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch32.TakeHypTrapException(0x00);
else
UNDEFINED;
elsif Halted() && EDSCR.SDD == '1' then
if EL2Enabled() && CPTR_EL3.TAM == '1' then
if Halted() && CPTR_EL3.TAM == '1' then
UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
R[t] = AMCFGR;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR_SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR_SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end if
  else
    R[t] = AMCFGR;
  end if
end if
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR_SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR_SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end if
  else
    R[t] = AMCFGR;
  end if
elsif PSTATE_EL == EL3 then
  R[t] = AMCFGR;
endif
G8.5.2 AMCGCR, Activity Monitors Counter Group Configuration Register

The AMCGCR characteristics are:

**Purpose**

Provides information on the number of activity monitor event counters implemented within each counter group.

**Configurations**

AArch32 System register AMCGCR bits [31:0] are architecturally mapped to AArch64 System register AMCGCR_EL0[31:0].

AArch32 System register AMCGCR bits [31:0] are architecturally mapped to External register AMCGCR[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCGCR are UNDEFINED.

**Attributes**

AMCGCR is a 32-bit register.

**Field descriptions**

![AMCGCR Register Diagram]

**Bits [31:16]**

Reserved, RES0.

**CG1NC, bits [15:8]**

Counter Group 1 Number of Counters. The number of counters in the auxiliary counter group. In an implementation that includes FEAT_AMUv1, the permitted range of values is 0 to 16. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.

**CG0NC, bits [7:0]**

Counter Group 0 Number of Counters. The number of counters in the architected counter group. Reads as 0x04. Access to this field is RO.

**Accessing AMCGCR**

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC{<c>}{<q>}{<coproc>}{(#)<opc1>}{<Rt>}{<CRn>}{<CRm>{(, (#<opc2>)}} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0010</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION DEFINED "EL3 trap priority when SDD == '1" && !ELUsingArch32(EL3) && CPTR_EL3.TAM == '1' then
UNDEFINED;
elsif !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  endif
elsif ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    UNDEFINED;
  endif
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE != '1' && HSTR_EL2.T13 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCPTR_EL2.TAM == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
else
  R[t] = AMCGCR;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" & !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
else
  R[t] = AMCGCR;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" & !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
else
  R[t] = AMCGCR;
elsif PSTATE.EL == EL3 then
  R[t] = AMCGCR;
### G8.5.3 AMCNTENCLR0, Activity Monitors Count Enable Clear Register 0

The AMCNTENCLR0 characteristics are:

**Purpose**

Disable control bits for the architected activity monitors event counters, `AMEVCTR0<n>`.

**Configurations**

AArch32 System register AMCNTENCLR0 bits [31:0] are architecturally mapped to AArch64 System register `AMCNTENCLR0_EL0[31:0]`.

AArch32 System register AMCNTENCLR0 bits [31:0] are architecturally mapped to External register `AMCNTENCLR0[31:0]`.

This register is present only when `FEAT_AMUv1` is implemented. Otherwise, direct accesses to AMCNTENCLR0 are **UNDEFINED**.

**Attributes**

AMCNTENCLR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>RAZ/WI</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td></td>
<td></td>
<td>RAZ/WI</td>
<td>P3</td>
<td>P2</td>
<td>P1</td>
<td>P0</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:16]**

Reserved, `RES0`.

**Bits [15:4]**

Reserved, `RAZ/WI`.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

**P<n>, bit [n], for n = 3 to 0**

Activity monitor event counter disable bit for `AMEVCTR0<n>`.

**Note**

`AMCGCR.CG0NC` identifies the number of architected activity monitor event counters. In an implementation that includes `FEAT_AMUv1`, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

- 0b0 When read, means that `AMEVCTR0<n>` is disabled. When written, has no effect.
- 0b1 When read, means that `AMEVCTR0<n>` is enabled. When written, disables `AMEVCTR0<n>`.

The reset behavior of this field is:

- On an AMU reset, this field resets to 0.

**Accessing AMCNTENCLR0**

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then UNDEFINED;
    elsif !ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch32.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
        end if;
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        UNDEFINED;
    end if;
elseif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then '11' && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
elseif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
    AArch32.TakeHypTrapException(0x03);
elseif EL2Enabled() && !ELUsingAArch32(EL2) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
end if;
else
    R[t] = AMCNTENCLR0;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then UNDEFINED;
elseif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
elseif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
    AArch32.TakeHypTrapException(0x03);
elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
end if;
else
    R[t] = AMCNTENCLR0;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then UNDEFINED;
elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0010</td>
<td>0b100</td>
</tr>
</tbody>
</table>
UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  R[t] = AMCNTENCLR0;
elsif PSTATE.EL == EL3 then
  R[t] = AMCNTENCLR0;
elsif PSTATE.EL == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif IsHighestEL(PSTATE.EL) then
  AMCNTENCLR0 = R[t];
else
  UNDEFINED;

\begin{tabular}{|c|c|c|c|c|}
\hline
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
\hline
0b1111 & 0b000 & 0b1101 & 0b0010 & 0b100 \\
\hline
\end{tabular}

\begin{align*}
\text{MCR}\{c\}\{q\} <\text{coproc}>\,, \{#\}<\text{opc1}>\,, <\text{Rt}>\,, <\text{CRn}>\,, <\text{CRm}>\{, \{#\}<\text{opc2}>\}
\end{align*}

if PSTATE.EL == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif IsHighestEL(PSTATE.EL) then
  AMCNTENCLR0 = R[t];
else
  UNDEFINED;

G8.5.4  AMCNTENCLR1, Activity Monitors Count Enable Clear Register 1

The AMCNTENCLR1 characteristics are:

**Purpose**

Disable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>.

**Configurations**

AArch32 System register AMCNTENCLR1 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENCLR1_EL0[31:0].

AArch32 System register AMCNTENCLR1 bits [31:0] are architecturally mapped to External register AMCNTENCLR1[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCNTENCLR1 are UNDEFINED.

**Attributes**

AMCNTENCLR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

P<n>, bit [n], for n = 15 to 0

Activity monitor event counter disable bit for AMEVCNTR1<n>. When N is less than 16, bits [15:N] are RAZ/WI, where N is the value in AMCGCR.CG1NC.

Possible values of each bit are:

0b0  When read, means that AMEVCNTR1<n> is disabled. When written, has no effect.

0b1  When read, means that AMEVCNTR1<n> is enabled. When written, disables AMEVCNTR1<n>.

The reset behavior of this field is:

- On an AMU reset, this field resets to 0.

**Accessing AMCNTENCLR1**

If the number of auxiliary activity monitor event counters implemented is zero, reads and writes of AMCNTENCLR1 are UNDEFINED.

--- Note ---

The number of auxiliary activity monitor event counters implemented is zero exactly when AMCFGR.NCG == 0b0000.

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" then
        UNDEFINED;
    elsif !ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
        end if
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch32.TakeHypTrapException(0x00);
    else
        UNDEFINED;
    end if
else
    if EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
        AArch32.TakeHypTrapException(0x00);
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTER.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif EL2Enabled() && !ELUsingAArch32(EL1) && HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTER.TAM == '1' then
        AArch32.TakeHypTrapException(0x03);
    elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.FCEN == '1' && !HAFCTR_EL2.AMCNEN1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        end if
    end if
else
    R[t] = AMCNTENCLR1;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        end if
    end if
else
    R[t] = AMCNTENCLR1;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        end if
    end if
else
    R[t] = AMCNTENCLR1;
end if

| MRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>} |
|---------------------|----------------|----------------|----------------|
| coproc | opc1 | CRn | CRm | opc2 |
| 0b1111 | 0b0000 | 0b1101 | 0b0011 | 0b0000 |
UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  R[t] = AMCNTENCLR1;
elsif PSTATE_EL == EL3 then
  R[t] = AMCNTENCLR1;

if PSTATE_EL == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE_EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif IsHighestEL(PSTATE_EL) then
  AMCNTENCLR1 = R[t];
else
  UNDEFINED;

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>
G8.5.5 AMCNTENSET0, Activity Monitors Count Enable Set Register 0

The AMCNTENSET0 characteristics are:

**Purpose**

Enable control bits for the architected activity monitors event counters, AMEVCNTR0<n>.

**Configurations**

AArch32 System register AMCNTENSET0 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENSET0_EL0[31:0].

AArch32 System register AMCNTENSET0 bits [31:0] are architecturally mapped to External register AMCNTENSET0[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCNTENSET0 are UNDEFINED.

**Attributes**

AMCNTENSET0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>RAZ/WI</td>
</tr>
<tr>
<td>P3</td>
</tr>
<tr>
<td>P2</td>
</tr>
<tr>
<td>P1</td>
</tr>
<tr>
<td>P0</td>
</tr>
</tbody>
</table>

**Bits [31:16]**

Reserved, RES0.

**Bits [15:4]**

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

**P<n>, bit [n], for n = 3 to 0**

Activity monitor event counter enable bit for AMEVCNTR0<n>.

--- **Note** ---

AMCGCR.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

- 0b0: When read, means that AMEVCNTR0<n> is disabled. When written, has no effect.
- 0b1: When read, means that AMEVCNTR0<n> is enabled. When written, enables AMEVCNTR0<n>.

The reset behavior of this field is:

* On an AMU reset, this field resets to 0.

**Accessing AMCNTENSET0**

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE_EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && AMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    end if;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
    AArch32.TakeHypTrapException(0x00);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL3) && (HaveEL(EL3) || SCR_EL3.FCTEN == '1') && HAFCTR_EL2_AMCNTEN0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end if;
  else
    R[t] = AMCNTENSET0;
  end if;
elsif PSTATE_EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end if;
  else
    R[t] = AMCNTENSET0;
  end if;
elsif PSTATE_EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end if;
  else
    R[t] = AMCNTENSET0;
  end if;
endif;
UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
    R[t] = AMCNSENSET0;
elseif PSTATE.EL == EL3 then
    R[t] = AMCNSENSET0;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0010</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseif PSTATE.EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
elseif IsHighestEL(PSTATE.EL) then
    AMCNSENSET0 = R[t];
else
    UNDEFINED;
G8.5.6 AMCNTENSET1, Activity Monitors Count Enable Set Register 1

The AMCNTENSET1 characteristics are:

**Purpose**

Enable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>.

**Configurations**

AArch32 System register AMCNTENSET1 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENSET1_EL0[31:0].

AArch32 System register AMCNTENSET1 bits [31:0] are architecturally mapped to External register AMCNTENSET1[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCNTENSET1 are UNDEFINED.

**Attributes**

AMCNTENSET1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>16-1</td>
<td>Reserved.</td>
</tr>
<tr>
<td>n</td>
<td>Activity monitor event counter enable bit for AMEVCNTR1&lt;n&gt;.</td>
</tr>
</tbody>
</table>

When N is less than 16, bits [15:N] are RAZ/WI, where N is the value in AMCGCR.CG1NC.

Possible values of each bit are:

- 0b0: When read, means that AMEVCNTR1<n> is disabled. When written, has no effect.
- 0b1: When read, means that AMEVCNTR1<n> is enabled. When written, enables AMEVCNTR1<n>.

The reset behavior of this field is:

- On an AMU reset, this field resets to 0.

**Accessing AMCNTENSET1**

If the number of auxiliary activity monitor event counters implemented is zero, reads and writes of AMCNTENSET1 are UNDEFINED.

--- Note ---

The number of auxiliary activity monitor counters implemented is zero when AMCFGR.NCG == 0b0000.

Accesses to this register use the following encodings in the System register encoding space:
MRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0011</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif ELUsingAArch32(EL1) && AMUSERENR_EL0.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    end if
  elsif ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      UNDEFINED;
    end if
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  end if
else
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.<E2H,TGE> != '11' && HAFGRTR_EL2.AMCNTEN1 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end if
else
  R[t] = AMCNTENSET1;
else if PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
    AArch32.TakeHypTrapException(0x03);
  elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end if
else
  R[t] = AMCNTENSET1;
else if PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
    end if
  end if
UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
    R[t] = AMCNTENSET1;
elsif PSTATE_EL == EL3 then
    R[t] = AMCNTENSET1;

MCR\{\texttt{c}><\texttt{q}}\}<\texttt{coproc}, \{\#}<\texttt{opc1}, \texttt{R}<\texttt{t}, \texttt{CR}<\texttt{n}, \texttt{CR}<\texttt{m}\{, \{\#}<\texttt{opc2}>\}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL1 && ELEnabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE_EL == EL1 && ELEnabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif IsHighestEL(PSTATE_EL) then
    AMCNTENSET1 = R[t];
else
    UNDEFINED;
G8.5.7  **AMCR, Activity Monitors Control Register**

The AMCR characteristics are:

**Purpose**

Global control register for the activity monitors implementation. AMCR is applicable to both the architected and the auxiliary counter groups.

**Configurations**

AArch32 System register AMCR bits [31:0] are architecturally mapped to AArch64 System register AMCR_EL0[31:0].

AArch32 System register AMCR bits [31:0] are architecturally mapped to External register AMCR[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCR are UNDEFINED.

**Attributes**

AMCR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-18</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>17</td>
<td>CG1RZ</td>
</tr>
<tr>
<td>16-11</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>10</td>
<td>HDBG</td>
</tr>
</tbody>
</table>

**Bits [31:18]**

Reserved, RES0.

**CG1RZ, bit [17]**

*When FEAT_AMUv1p1 is implemented:*

Counter Group 1 Read Zero.

- **0b0**  
  System register reads of AMEVCNTR1<n> return the event count at all implemented and enabled Exception levels.

- **0b1**  
  If the current Exception level is the highest implemented Exception level, system register reads of AMEVCNTR1<n> return the event count. Otherwise, reads of AMEVCNTR1<n> return a zero value.

**Note**

Reads from the memory-mapped view are unaffected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [16:11]**

Reserved, RES0.

**HDBG, bit [10]**

This bit controls whether activity monitor counting is halted when the PE is halted in Debug state.

- **0b0**  
  Activity monitors do not halt counting when the PE is halted in Debug state.

- **0b1**  
  Activity monitors halt counting when the PE is halted in Debug state.
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [9:0]**

Reserved, RES0.

**Accessing AMCR**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC\{<c>\}/\{q\} <coproc>, \{#\}<opc1>, <Rt>, <CRn>, <CRm>\{, \{#\}<opc2>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && IELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then UNDEFINED;
elsif !ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
if EL2Enabled() && IELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL1, 0x03);
else UNDEFINED;
else EL2Enabled() && !IELUsingAArch32(EL2) && HCR_EL2.TGE != '1' then
if Halted() && EDSCR.SDD == '1' then
AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else R[t] = AMCR;
else PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && IELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then UNDEFINED;
else EL2Enabled() && IELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else EL2Enabled() && IELUsingAArch32(EL2) && HSTR.T13 == '1' then
AArch32.TakeHypTrapException(0x03);
else EL2Enabled() && IELUsingAArch32(EL2) && HSTR.T13 != '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else HaveEL(EL3) && IELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
R[t] = AMCR;
else PSTATE.EL == EL0 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && IELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then UNDEFINED;
else EL2Enabled() && IELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else EL2Enabled() && IELUsingAArch32(EL2) && HSTR.T13 == '1' then
AArch32.TakeHypTrapException(0x03);
else EL2Enabled() && IELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else HaveEL(EL3) && IELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
if Halted() && EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
R[t] = AMCR;
elsif PSTATE.EL == EL2 then
if Halted() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
AArch32.TakeHypTrapException(0x03);
elsif IsHighestEL(PSTATE.EL) then
AMCR = R[t];
else
UNDEFINED;
else
R[t] = AMCR;
elsif PSTATE.EL == EL3 then
R[t] = AMCR;
endif

coproc opc1 CRn CRm opc2
0b1111 0b0000 0b1101 0b0010 0b0000
G8.5.8  **AMEVCNTR0<n>, Activity Monitors Event Counter Registers 0, n = 0 - 3**

The AMEVCNTR0<n> characteristics are:

**Purpose**

Provides access to the architected activity monitor event counters.

**Configurations**

AArch32 System register AMEVCNTR0<n> bits [63:0] are architecturally mapped to AArch64 System register AMEVCNTR0<n>_EL0[63:0].

AArch32 System register AMEVCNTR0<n> bits [63:0] are architecturally mapped to External register AMEVCNTR0<n>[63:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMEVCNTR0<n> are UNDEFINED.

**Attributes**

AMEVCNTR0<n> is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th></th>
<th>ACNT, bits [63:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Architected activity monitor event counter n.</td>
</tr>
<tr>
<td></td>
<td>Value of architected activity monitor event counter n, where n is the number of this register and is a number from 0 to 3.</td>
</tr>
<tr>
<td></td>
<td>If FEAT_AMUv1p1 is implemented, HCR_EL2.AMVOFFEN is 1, SCR_EL3.AMVOFFEN is 1, HCR_EL2.{E2H, TGE} is not {1,1}, and EL2 is using AArch64 and is implemented in the current Security state, access to these registers at EL0 or EL1 return (PCount&lt;63:0&gt; - AMEVVCNTVOFF0&lt;n&gt;_EL2&lt;63:0&gt;).</td>
</tr>
<tr>
<td></td>
<td>PCount is the physical count returned when AMEVCNTR0&lt;n&gt; is read from EL2 or EL3.</td>
</tr>
<tr>
<td></td>
<td>If the counter is enabled, writes to this register have UNPREDICTABLE results.</td>
</tr>
<tr>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On an AMU reset, this field resets to 0.</td>
</tr>
</tbody>
</table>

**Accessing AMEVCNTR0<n>**

If <n> is greater than or equal to the number of architected activity monitor event counters, reads and writes of AMEVCNTR0<n> are UNDEFINED.

--- **Note** ---

AMCGCR.CG0NC identifies the number of architected activity monitor event counters.

Accesses to this register use the following encodings in the System register encoding space:
G8.5 Activity Monitors registers

integer m = UInt(CRm<0>:opc1<2:0>);

if m >= 4 then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD = '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        UNDEFINED;
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    end if;
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && m < 8 && HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
    AArch32.TakeHypTrapException(0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL1) && HCR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && m < 8 && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
    AArch32.TakeHypTrapException(0x04);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    AArch64.AArch32SystemAccessTrap(EL3, 0x04);
elsif EL2Enabled() && HAFCTR_EL2.AMEVCNTR0.m<63:32> == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x04);
    end if;
else
    (R[t], R[t]) = (AMEVCNTR0.m<63:32>, AMEVCNTR0.m<31:0>);
end if;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD = '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        UNDEFINED;
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T0 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    end if;
elsif EL2Enabled() && ELUsingAArch32(EL2) && m < 8 && HSTR.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
    AArch32.TakeHypTrapException(0x04);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x04);
    end if;
else
    (R[t], R[t]) = (AMEVCNTR0.m<63:32>, AMEVCNTR0.m<31:0>);
end if;
elsif PSTATE.EL == EL2 then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    end if;
else
    (R[t], R[t]) = (AMEVCNTR0.m<63:32>, AMEVCNTR0.m<31:0>);
end if;
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
  UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x04);
  end
elsif PSTATE.EL == EL3 then
  (R[t2], R[t]) = (AMEVCNTR0[m]<63:32>, AMEVCNTR0[m]<31:0>);
else
  integer m = UInt(CRm<0>:opc1<2:0>);
  if m >= 4 then
    UNDEFINED;
  elsif PSTATE.EL == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && m < 8 && HSTR_EL2.T0 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif PSTATE.EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && m < 8 && HSTR.T0 == '1' then
    AArch32.TakeHypTrapException(0x04);
  elsif IsHighestEL(PSTATE.EL) then
    AMEVCNTR0[m] = R[t2]:R[t];
  else
    UNDEFINED;
end
G8.5.9 AMEVCNTR1<n>, Activity Monitors Event Counter Registers 1, n = 0 - 15

The AMEVCNTR1<n> characteristics are:

**Purpose**

Provides access to the auxiliary activity monitor event counters.

**Configurations**

AArch32 System register AMEVCNTR1<n> bits [63:0] are architecturally mapped to AArch64 System register AMEVCNTR1<n>_EL0[63:0].

AArch32 System register AMEVCNTR1<n> bits [63:0] are architecturally mapped to External register AMEVCNTR1<n>[63:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMEVCNTR1<n> are UNDEFINED.

**Attributes**

AMEVCNTR1<n> is a 64-bit register.

**Field descriptions**

```
<table>
<thead>
<tr>
<th>63</th>
<th>ACNT</th>
</tr>
</thead>
</table>
```

ACNT, bits [63:0]

Auxiliary activity monitor event counter n.

Value of auxiliary activity monitor event counter n, where n is the number of this register and is a number from 0 to 15.

If FEAT_AMUv1p1 is implemented, HCR_EL2.AMVOFFEN is 1, SCR_EL3.AMVOFFEN is 1, HCR_EL2.{E2H, TGE} is not {1,1}, EL2 is using AArch64 and is implemented in the current Security state, and AMCR_EL0.CG1RZ is 0, reads to these registers at EL0 or EL1 return (PCount<63:0> - AMEVCNTVOFF1<n>_EL2<63:0>).

PCount is the physical count returned when AMEVCNTR1<n> is read from EL2 or EL3.

If the counter is enabled, writes to this register have UNPREDICTABLE results.

The reset behavior of this field is:

- On an AMU reset, this field resets to 0.

**Accessing AMEVCNTR1<n>**

If <n> is greater than or equal to the number of auxiliary activity monitor event counters, reads and writes of AMEVCNTR1<n> are UNDEFINED.

--- Note ---

AMCGCR.CG1NC identifies the number of auxiliary activity monitor event counters.

Accesses to this register use the following encodings in the System register encoding space:
integer m = UInt(CRm<0>:opc1<2:0>);
if m >= NUM_AMU_CG1_MONITORS then
  UNDEFINED;
elsif !IsG1ActivityMonitorImplemented(m) then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elseif !ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    end
  elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x04);
    end
  else
    (R[t2], R[t]) = (Zeros(32), Zeros(32));
    if PSTATE.EL == EL1 then
      if Halted() && HaveEL(EL1) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL1 trap priority when SDD == '1'" && !ELUsingAArch32(EL1) && CPTR_EL3.TAM == '1' then
        UNDEFINED;
      elseif !ELUsingAArch32(EL2) && m >= 8 && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
      elseif EL2Enabled() && !ELUsingAArch32(EL2) && m >= 8 && HSTR_T5 == '1' then
        AArch32.TakeHypTrapException(0x00);
      elseif !ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
      elseif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
        AArch32.TakeHypTrapException(0x04);
      else
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
      end
    else
      (R[t2], R[t]) = (AMEVCNTR1[m]<63:32>, AMEVCNTR1[m]<31:0>);
    end
  end
end
UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x04);
elsif !IsHighestEL(PSTATE.EL) && HaveAArch64() && AMCR_EL0.CG1RZ == '1' then
    (R[t2], R[t]) = (Zeros(32), Zeros(32));
elsif !IsHighestEL(PSTATE.EL) && !HaveAArch64() && AMCR.CG1RZ == '1' then
    (R[t2], R[t]) = (Zeros(32), Zeros(32));
else
    (R[t2], R[t]) = (AMEVCNTR1[m]<63:32>, AMEVCNTR1[m]<31:0>);
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x04);
        end if
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x04);
    end if
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x04);
elsif !IsHighestEL(PSTATE.EL) && HaveAArch64() && AMCR_EL0.CG1RZ == '1' then
    (R[t2], R[t]) = (Zeros(32), Zeros(32));
elsif !IsHighestEL(PSTATE.EL) && !HaveAArch64() && AMCR.CG1RZ == '1' then
    (R[t2], R[t]) = (Zeros(32), Zeros(32));
else
    (R[t2], R[t]) = (AMEVCNTR1[m]<63:32>, AMEVCNTR1[m]<31:0>);
elsif PSTATE.EL == EL3 then
    (R[t2], R[t]) = (AMEVCNTR1[m]<63:32>, AMEVCNTR1[m]<31:0>);
else
    (R[t2], R[t]) = (AMEVCNTR1[m]<63:32>, AMEVCNTR1[m]<31:0>);
end if

integer m = UInt(CRm<0>:opc1<2:0>);
if m >= NUM_AMU_CG1_MONITORS then
    UNDEFINED;
elsif !IsGIActivityMonitorImplemented(m) then
    UNDEFINED;
elsif PSTATE.EL == EL1 && EL2Enabled() && (ELUSingAArch32(EL2) && m >= 8 && HSTR_EL2.T5 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif PSTATE.EL == EL1 && EL2Enabled() && ELUSingAArch32(EL2) && m >= 8 && HSTR.T5 == '1' then
    AArch32.TakeHypTrapException(0x04);
elsif IsHighestEL(PSTATE.EL) then
    AMEVCNTR1[m] = R[t2]:R[t];
else
    UNDEFINED;
G8.5.10  AMEVTYPER0<n>, Activity Monitors Event Type Registers 0, n = 0 - 3

The AMEVTYPER0<n> characteristics are:

**Purpose**

Provides information on the events that an architected activity monitor event counter AMEVCNTR0<n> counts.

**Configurations**

AArch32 System register AMEVTYPER0<n> bits [31:0] are architecturally mapped to AArch64 System register AMEVTYPER0<n>_EL0[31:0].

AArch32 System register AMEVTYPER0<n> bits [31:0] are architecturally mapped to External register AMEVTYPER0<n>[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMEVTYPER0<n> are UNDEFINED.

**Attributes**

AMEVTYPER0<n> is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0</td>
</tr>
<tr>
<td>15</td>
<td>evtCount</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:16]**

Reserved, RES0.

**evtCount, bits [15:0]**

Event to count. The event number of the event that is counted by the architected activity monitor event counter AMEVCNTR0<n>. The value of this field is architecturally mandated for each architected counter.

The following table shows the mapping between required event numbers and the corresponding counters:

<table>
<thead>
<tr>
<th>Event Number</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0011</td>
<td>Processor frequency cycles</td>
</tr>
<tr>
<td>0x4004</td>
<td>Constant frequency cycles</td>
</tr>
<tr>
<td>0x0008</td>
<td>Instructions retired</td>
</tr>
<tr>
<td>0x4005</td>
<td>Memory stall cycles</td>
</tr>
</tbody>
</table>

**Accessing AMEVTYPER0<n>**

If <n> is greater than or equal to the number of architected activity monitor event counters, reads and writes of AMEVTYPER0<n> are UNDEFINED.

--- **Note** ---

AMCGCR.CG0NC identifies the number of architected activity monitor event counters.

Accesses to this register use the following encodings in the System register encoding space:
integer m = UInt(CRm<0>:opc2<2:0>);

if m >= 4 then
  UNDEFINED;
elsif PSTATE.EL == EL0 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  else if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  R[t] = AMEVTYPER0[m];
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  else if EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
if EL2Enabled() && ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  R[t] = AMEVTYPER0[m];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  R[t] = AMEVTYPER0[m];
if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
    R[t] = AMEVTYPER0[m];
elsif PSTATE.EL == EL3 then
    R[t] = AMEVTYPER0[m];
G8.5.11  **AMEVTYPER1<n>, Activity Monitors Event Type Registers 1, n = 0 - 15**

The AMEVTYPER1<n> characteristics are:

**Purpose**

Provides information on the events that an auxiliary activity monitor event counter AMEVCNTR1<n> counts.

**Configurations**

AArch32 System register AMEVTYPER1<n> bits [31:0] are architecturally mapped to AArch64 System register AMEVTYPER1<n>_EL0[31:0].

AArch32 System register AMEVTYPER1<n> bits [31:0] are architecturally mapped to External register AMEVTYPER1<n>[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMEVTYPER1<n> are UNDEFINED.

**Attributes**

AMEVTYPER1<n> is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits [31:16]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td><strong>evtCount</strong>, bits [15:0]</td>
<td>Event to count. The event number of the event that is counted by the auxiliary activity monitor event counter AMEVCNTR1&lt;n&gt;. It is IMPLEMENTATION DEFINED what values are supported by each counter. If software writes a value to this field which is not supported by the corresponding counter AMEVCNTR1&lt;n&gt;, then:</td>
</tr>
<tr>
<td></td>
<td>• It is UNPREDICTABLE which event will be counted.</td>
</tr>
<tr>
<td></td>
<td>• The value read back is UNKNOWN.</td>
</tr>
<tr>
<td></td>
<td>The event counted by AMEVCNTR1&lt;n&gt; might be fixed at implementation. In this case, the field is read-only and writes are UNDEFINED. If the corresponding counter AMEVCNTR1&lt;n&gt; is enabled, writes to this register have UNPREDICTABLE results. The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
</tbody>
</table>

**Accessing AMEVTYPER1<n>**

If <n> is greater than or equal to the number of auxiliary activity monitor event counters, reads and writes of AMEVTYPER1<n> are UNDEFINED.

--- **Note** ---

AMCGCR.CG1NC identifies the number of auxiliary activity monitor event counters.

Accesses to this register use the following encodings in the System register encoding space:
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b111:m[3]</td>
<td>m[2:0]</td>
</tr>
</tbody>
</table>

```plaintext
cp = UInt(CRm<0>:opc2<2:0>);
if (cp >= NUM_AMU.CG1_MONITORS)
    UNDEFINED;
elsif !IsG1ActivityMonitorImplemented(cp)
    UNDEFINED;
elsif PSTATE_EL == EL0
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1'
        UNDEFINED;
    elsif !ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR.EL2.TGE == '1'
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    end elsif ELUsingAArch32(EL1) && AMUSERENR.EN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR.EL2.TGE == '1'
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            EL2Enabled() && !ELUsingAArch32(EL2) && HCR.TGE == '1'
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
        end elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1'
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1'
            AArch32.TakeHypTrapException(0x03);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.<E2H,TGE> != '11'
            if !ELUsingAArch32(EL3) || (HaveEL(EL3) || SCR.EL3.FGTEn == '1') && HAFGRTR.EL2.AMEVTYPER1<en>EL0 == '1'
                AArch64.AArch32SystemAccessTrap(EL2, 0x03);
            elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1'
                if Halted() && EDSCR.SDD == '1'
                    UNDEFINED;
                else
                    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
                end else
                    R[t] = AMEVTYPER1[m];
            end elsif PSTATE_EL == EL1
                if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1'
                    UNDEFINED;
                elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.EL2.T13 == '1'
                    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
                elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1'
                    AArch32.TakeHypTrapException(0x03);
                elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TAM == '1'
                    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
                elsif EL2Enabled() && ELUsingAArch32(EL2) && HCPTR.TAM == '1'
                    AArch32.TakeHypTrapException(0x03);
                elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.<E2H,TGE> != '11'
                    if !ELUsingAArch32(EL3) || (HaveEL(EL3) || SCR.EL3.FGTEn == '1') && HAFGRTR.EL2.AMEVTYPER1<en>EL0 == '1'
                        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
                    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1'
                        if Halted() && EDSCR.SDD == '1'
                            UNDEFINED;
                        else
                            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
                        end else
                            R[t] = AMEVTYPER1[m];
```
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
  when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end
  else
    R[t] = AMEVTYPE1[m];
  end
elsif PSTATE.EL == EL3 then
  R[t] = AMEVTYPE1[m];
elsif PSTATE.EL == EL2 then
  if m >= NUM_AMU.CG1.MONITORS then
    UNDEFINED;
  elsif !IsG1ActivityMonitorImplemented(m) then
    UNDEFINED;
  elsif PSTATE.EL == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif PSTATE.EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif IsHighestEL(PSTATE.EL) && !boolean IMPLEMENTATION_DEFINED "AMEVCNTR1[m] is fixed" then
    AMEVTYPE1[m] = R[t];
  else
    UNDEFINED;
  end

MCR{<c>}{<q>} <coproc>, {#<opc1>}, <Rt>, <CRn>, <CRm>{, {#<opc2>}}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b111:</td>
<td>m[2:0]</td>
</tr>
</tbody>
</table>

integer m = UInt(CRm<0>:opc2<2:0>);

if m >= NUM_AMU.CG1.MONITORS then
  UNDEFINED;
elsif !IsG1ActivityMonitorImplemented(m) then
  UNDEFINED;
elsif PSTATE.EL == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T13 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif PSTATE.EL == EL1 && EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T13 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif IsHighestEL(PSTATE.EL) && !boolean IMPLEMENTATION_DEFINED "AMEVCNTR1[m] is fixed" then
  AMEVTYPE1[m] = R[t];
else
  UNDEFINED;
AMUSERENR, Activity Monitors User Enable Register

The AMUSERENR characteristics are:

**Purpose**

Global user enable register for the activity monitors. Enables or disables EL0 access to the activity monitors. AMUSERENR is applicable to both the architected and the auxiliary counter groups.

**Configurations**

AArch32 System register AMUSERENR bits [31:0] are architecturally mapped to AArch64 System register AMUSERENR_EL0[31:0].

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMUSERENR are UNDEFINED.

**Attributes**

AMUSERENR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:1</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>0</td>
<td>EN, control bit.</td>
</tr>
</tbody>
</table>

**EN, bit [0]**

Traps EL0 accesses to the activity monitors registers to EL1.

0b0: EL0 accesses to the activity monitors registers are trapped to EL1.

0b1: This control does not cause any instructions to be trapped. Software can access all activity monitor registers at EL0.

**Note**

- AMUSERENR can always be read at EL0 and is not governed by this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing AMUSERENR**

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC\langle c\rangle\langle q\rangle <\text{coproc}, \langle\#\rangle<\text{opc}1>, \langle RT\rangle, <\text{CRn}>, <\text{CRm}>, \langle\#\rangle<\text{opc}2> \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1101</td>
<td>0b0010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & CPTR_EL3.TAM == '1' then
        UNDEFINED;
    elsif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.<E2H,TGE> != '11' & HSTR_EL2.T13 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T13 == '1' then
AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() & !ELUsingAArch32(EL2) & CPTR_EL2.TAM == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HCPTR.TAM == '1' then
AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & CPTR_EL3.TAM == '1' then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
R[t] = AMUSERENR;
elsif PSTATE_EL == EL1 then
if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" & !ELUsingAArch32(EL3) & CPTR_EL3.TAM == '1' then
UNDEFINED;
elsif EL2Enabled() & !ELUsingAArch32(EL2) & HSTR_EL2.T13 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T13 == '1' then
AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() & !ELUsingAArch32(EL2) & HCPTR.TAM == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & CPTR_EL2.TAM == '1' then
AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() & !ELUsingAArch32(EL2) & CPTR_EL2.TAM == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & CPTR_EL3.TAM == '1' then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
R[t] = AMUSERENR;
elsif PSTATE_EL == EL2 then
if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" & ELUsingAArch32(EL3) & CPTR_EL3.TAM == '1' then
UNDEFINED;
elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & CPTR_EL3.TAM == '1' then
if Halted() & EDSCR.SDD == '1' then
UNDEFINED;
else
AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
R[t] = AMUSERENR;
elsif PSTATE_EL == EL3 then
R[t] = AMUSERENR;

MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>({(#)<opc2>})

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0000</td>
<td>0b1101</td>
<td>0b0010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
UNDEFINED;
elsif PSTATE_EL == EL1 then
if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" & !ELUsingAArch32(EL3) & CPTR_EL3.TAM == '1' then
UNDEFINED;
elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR_EL2.T13 == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T13 == '1' then
AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HCPTR.TAM == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HPTR.TAM == '1' then 
AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then 
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
else
  AMUSERENR = R[t];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TAM == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  else
    AMUSERENR = R[t];
elsif PSTATE.EL == EL3 then
  AMUSERENR = R[t];
G8.6 RAS registers

This section lists The Reliability, Availability, and Serviceability Extension registers in AArch32.
### G8.6.1 DISR, Deferred Interrupt Status Register

The DISR characteristics are:

**Purpose**

Records that an SError interrupt has been consumed by an E58 instruction.

**Configurations**

AArch32 System register DISR bits [31:0] are architecturally mapped to AArch64 System register DISR_EL1[31:0] when the highest implemented Exception level is using AArch64.

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to DISR are UNDEFINED.

**Attributes**

DISR is a 32-bit register.

**Field descriptions**

*When the ESB instruction is executed at EL2:*

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| A     | Set to 1 when an E58 instruction defers an asynchronous SError interrupt. If the implementation does not include any sources of SError interrupt that can be synchronized by an Error Synchronization Barrier, then this bit is RES0. The reset behavior of this field is:  
  - On a Warm reset, this field resets to an architecturally UNKNOWN value. |
| Bits [30:12] | Reserved, RES0. |
| AET     | Asynchronous Error Type. See the description of HSR.AET for an SError interrupt. The reset behavior of this field is:  
  - On a Warm reset, this field resets to an architecturally UNKNOWN value. |
| EA      | External abort Type. See the description of HSR.EA for an SError interrupt. The reset behavior of this field is:  
  - On a Warm reset, this field resets to an architecturally UNKNOWN value. |
| Bits [8:6] | Reserved, RES0. |
| DFSC    | Fault Status Code. See the description of HSR.DFSC for an SError interrupt. The reset behavior of this field is:  
  - On a Warm reset, this field resets to an architecturally UNKNOWN value. |
When the ESB instruction is executed at EL0 or EL1 and where TTBCR.EAE == 0:

<table>
<thead>
<tr>
<th>A</th>
<th>RES0</th>
<th>AET</th>
<th>RES0</th>
<th>FS[3:0]</th>
<th>RES0</th>
<th>LPAE</th>
<th>FS[4]</th>
<th>RES0</th>
</tr>
</thead>
</table>

A, bit [31]
Set to 1 when an ESB instruction defers an asynchronous SError interrupt. If the implementation does not include any sources of SError interrupt that can be synchronized by an Error Synchronization Barrier, then this bit is RES0.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:16]
Reserved, RES0.

AET, bits [15:14]
Asynchronous Error Type. See the description of DFSR.AET for an SError interrupt.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [13]
Reserved, RES0.

ExT, bit [12]
External abort Type. See the description of DFSR.ExT for an SError interrupt.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [11]
Reserved, RES0.

FS, bits [10, 3:0]
Fault Status Code. See the description of DFSR.FS for an SError interrupt.

The FS field is split as follows:
• FS[4] is DISR[10].
• FS[3:0] is DISR[3:0].

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

LPAE, bit [9]
Format.

0b0 Using the Short-descriptor translation table format.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:4]
Reserved, RES0.
When the ESB instruction is executed at EL0 or EL1 and where TTBCR.EAE == 1:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>RES0</td>
<td>AET</td>
<td>RES0</td>
<td>RES0</td>
<td>STATUS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A, bit [31]
Set to 1 when an ESB instruction defers an asynchronous SError interrupt. If the implementation does not include any sources of SError interrupt that can be synchronized by an Error Synchronization Barrier, then this bit is RES0.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:16]
Reserved, RES0.

AET, bits [15:14]
Asynchronous Error Type. See the description of DFSR.AET for an SError interrupt.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [13]
Reserved, RES0.

ExT, bit [12]
External abort Type. See the description of DFSR.ExT for an SError interrupt.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:10]
Reserved, RES0.

LPAE, bit [9]

Format.
0b1 Using the Long-descriptor translation table format.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:6]
Reserved, RES0.

STATUS, bits [5:0]
Fault Status Code. See the description of DFSR.FS for an SError interrupt.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DISR
An indirect write to DISR made by an ESB instruction does not require an explicit synchronization operation for the value that is written to be observed by a direct read of DISR occurring in program order after the ESB instruction.
DISR is RAZ/WI if EL3 is implemented, the PE is in Non-debug state, and any of the following apply:

- EL3 is using AArch64, SCR_EL3.EA == 1, and any of the following apply:
  - The PE is executing at EL2.
  - The PE is executing at EL1 and ((SCR_EL3.NS == 0 && SCR_EL3.EEL2 == 0) || HCR_EL2.AMO == 0).

- EL3 is using AArch32, SCR.EA == 1, and any of the following apply:
  - The PE is executing at EL2.
  - The PE is executing at EL1 and (SCR.NS == 0 || HCR.AMO == 0).

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AMO == '1' then
    R[t] = VDISR_EL2<31:0>;
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.AMO == '1' then
    R[t] = VDISR;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && !Halted() && SCR_EL3.EA == '1' then
    R[t] = Zeros(32);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && !Halted() && SCR.EA == '1' then
    R[t] = Zeros(32);
  else
    R[t] = DISR;
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && !ELUsingAArch32(EL3) && !Halted() && SCR_EL3.EA == '1' then
    R[t] = Zeros(32);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && !Halted() && SCR.EA == '1' then
    R[t] = Zeros(32);
  else
    R[t] = DISR;
elsif PSTATE.EL == EL3 then
  R[t] = DISR;

\[
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AMO == '1' then
    R[t] = VDISR_EL2<31:0>;
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.AMO == '1' then
    R[t] = VDISR;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && !Halted() && SCR_EL3.EA == '1' then
    R[t] = Zeros(32);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && !Halted() && SCR.EA == '1' then
    R[t] = Zeros(32);
  else
    R[t] = DISR;
AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AMO == '1' then
  VDISR_EL2 = R[t];
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.AMO == '1' then
  VDISR = R[t];
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && !Halted() && HCR_EL3.EA == '1' then
  return;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && !Halted() && Halted() && SCR_EL3.EA == '1' then
  return;
else
  DISR = R[t];
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && !ELUsingAArch32(EL3) && !Halted() && SCR_EL3.EA == '1' then
    return;
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && !Halted() && SCR_EL3.EA == '1' then
    return;
  else
    DISR = R[t];
  elsif PSTATE.EL == EL3 then
    DISR = R[t];
G8.6.2 ERRIDR, Error Record ID Register

The ERRIDR characteristics are:

**Purpose**
Defines the highest numbered index of the error records that can be accessed through the Error Record System registers.

**Configurations**
AArch32 System register ERRIDR bits [31:0] are architecturally mapped to AArch64 System register ERRIDR_EL1[31:0].
This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERRIDR are UNDEFINED.

**Attributes**
ERRIDR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>NUM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:16]**
Reserved, RES0.

**NUM, bits [15:0]**
Highest numbered index of the records that can be accessed through the Error Record System registers plus one. Zero indicates that no records can be accessed through the Error Record System registers.
Each implemented record is owned by a node. A node might own multiple records.

**Accessing ERRIDR**
Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} <\text{coproc}>, \{#\}<\text{opc1}>, <\text{Rp}>, <\text{CRn}>, <\text{CRm}>, \{#\}<\text{opc2}>
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03); elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then AArch32.TakeHypTrapException(0x03); elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then if Halted() && EDSCR.SDD == '1' then UNDEFINED; else AArch64.AArch32SystemAccessTrap(EL3, 0x03); elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then if Halted() && EDSCR.SDD == '1' then UNDEFINED; else AArch32.TakeMonitorTrapException(); else R[t] = ERRIDR; elsif PSTATE.EL == EL2 then if Halted() && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then UNDEFINED; elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then UNDEFINED; elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then if Halted() && SCR.TERR == '1' then UNDEFINED; else AArch64.AArch32SystemAccessTrap(EL3, 0x03); elsif HaveEL(EL3) && SCR.TERR == '1' then if Halted() && EDSCR.SDD == '1' then UNDEFINED; else AArch32.TakeMonitorTrapException(); else R[t] = ERRIDR; elsif PSTATE.EL == EL3 then if PSTATE.M != M32.Monitor && SCR.TERR == '1' then AArch32.TakeMonitorTrapException(); else R[t] = ERRIDR;
G8.6.3   ERRSELR, Error Record Select Register

The ERRSELR characteristics are:

**Purpose**

Selects an error record to be accessed through the Error Record System registers.

**Configurations**

AArch32 System register ERRSELR bits [31:0] are architecturally mapped to AArch64 System register ERRSEL_EL1[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERRSELR are UNDEFINED.

If ERRIDR indicates that zero error records are implemented, then it is IMPLEMENTATION DEFINED whether ERRSELR is UNDEFINED or RES0.

**Attributes**

ERRSELR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>15–16</td>
<td>SEL, bits [15:0]</td>
</tr>
</tbody>
</table>

Reserved, RES0.

SEL, bits [15:0]

Selects the error record accessed through the ERX registers.

For example, if ERRSELR.SEL is 0x0004, then direct reads and writes of ERXSTATUS access ERR4STATUS.

If ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then all of the following apply:

- The value read back from ERRSELR.SEL is UNKNOWN.
- One of the following occurs:
  - An **UNKNOWN** error record is selected.
  - The ERX* registers are RAZ/WI.
  - ERX* register reads and writes are NOPs.
  - ERX* register reads and writes are UNDEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Accessing ERRSELR**

Accesses to this register use the following encodings in the System register encoding space:
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch32.TakeMonitorTrapException();
    end
  else
    R[t] = ERRSELR;
  end
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch32.TakeHypTrapException();
    end
  else
    R[t] = ERRSELR;
  end
elsif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
  else
    R[t] = ERRSELR;
end

MRC{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>)}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & SCR_EL3.TERR == '1' then
        UNDEFINED;
    elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & ELUsingAArch32(EL3) & SCR.TERR == '1' then
        UNDEFINED;
    elsif EL2Enabled() & !ELUsingAArch32(EL2) & HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & SCR_EL3.TERR == '1' then
        if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        end
    else
        AArch32.TakeMonitorTrapException();
    end
    ERRSELR = R[t];
elsif PSTATE.EL == EL2 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & SCR_EL3.TERR == '1' then
        UNDEFINED;
    elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & ELUsingAArch32(EL3) & SCR.TERR == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & SCR_EL3.TERR == '1' then
        if Halted() & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        end
    else
        AArch32.TakeMonitorTrapException();
    end
    ERRSELR = R[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor & SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        ERRSELR = R[t];
    end
G8.6.4 ERXADDR, Selected Error Record Address Register

The ERXADDR characteristics are:

**Purpose**

Accesses bits [31:0] of ERR<\text{n}>ADDR for the error record <\text{n}> selected by ERRSELR.SEL.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

**Configurations**

AArch32 System register ERXADDR bits [31:0] are architecturally mapped to AArch64 System register ERXADDR_EL1[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXADDR are UNDEFINED.

**Attributes**

ERXADDR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits [31:0] of ERR&lt;\text{n}&gt;ADDR</td>
<td></td>
</tr>
</tbody>
</table>

**Accessing ERXADDR**

If ERRIDR.NUM is 0x0000 or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXADDR is RAZ/WI.
- Direct reads and writes of ERXADDR are NOPs.
- Direct reads and writes of ERXADDR are UNDEFINED.

ERR<\text{n}>ADDR describes additional constraints that also apply when ERR<\text{n}>ADDR is accessed through ERXADDR.

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} <\text{coproc}, \{#/}\text{opc1}, \langle Rt\rangle, \langle CRn\rangle, \langle CRm\rangle, \{#/\text{opc2}\}}
\]

\[
\begin{array}{ccccc}
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
\hline
0b1111 & 0b000 & 0b0101 & 0b0100 & 0b011 \\
\end{array}
\]

if PSTATE.EL == EL0 then

UNDEFINED;

elsif PSTATE.EL == EL1 then

if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority" when SDD == '1' && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then

UNDEFINED;
elseif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  UNDEFINED;
elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.TS == '1' then
  AArch32.TakeHypTrapException(0x03);
elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
  AArch32.TakeHypTrapException(0x03);
elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
elseif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
  endif
else
  R[t] = ERXADDR;
elseif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
elseif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
elseif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
  endif
else
  R[t] = ERXADDR;
elseif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
  else
    R[t] = ERXADDR;
  endif
endif

**MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rp>, <CRn>, <CRm>{, (#)<opc2>}
**

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elseif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
elseif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.TS == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.TS == '1' then
  AArch32.TakeHypTrapException(0x03);
elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
  AArch32.TakeHypTrapException(0x03);
elseif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
elseif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
  endif
else
  R[t] = ERXADDR;
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  end if;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
  end if;
elex PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end if;
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch32.TakeMonitorTrapException();
    end if;
elex PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
  else
    ERXADDR = R[t];
  end if;
elex PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
  else
    ERXADDR = R[t];
  end if;
G8.6.5 ERXADDR2, Selected Error Record Address Register 2

The ERXADDR2 characteristics are:

**Purpose**

Accesses bits [63:32] of ERR\(<n>\)ADDR for the error record \(<n>\) selected by ERRSELR.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

AArch32 System register ERXADDR2 bits [31:0] are architecturally mapped to AArch64 System register ERXADDR_EL1[63:32].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXADDR2 are UNDEFINED.

**Attributes**

ERXADDR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>ERXADDR2 accesses bits [63:32] of ERR(&lt;n&gt;)ADDR, where (&lt;n&gt;) is the value in ERRSELR.SEL.</td>
</tr>
</tbody>
</table>

**Accessing ERXADDR2**

If ERRIDR.NUM is 0x0000 or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXADDR2 is RAZ/WI.
- Direct reads and writes of ERXADDR2 are NOPs.
- Direct reads and writes of ERXADDR2 are UNDEFINED.

ERR\(<n>\)ADDR describes additional constraints that also apply when ERR\(<n>\)ADDR is accessed through ERXADDR2.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b111</td>
</tr>
</tbody>
</table>
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  UNDEFINED;
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
  if HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch32.TakeMonitorTrapException();
    endif
  endif
else
  R[t] = ERXADDR2;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
  if HaveEL(EL3) && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch32.TakeMonitorTrapException();
    endif
  endif
else
  R[t] = ERXADDR2;
elsif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
  else
    R[t] = ERXADDR2;
endif

MCR{<c>}{<q>} {<coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b000</td>
<td>0b010</td>
<td>0b0100</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
  if HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch32.TakeMonitorTrapException();
    endif
  endif
else
  R[t] = ERXADDR2;
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.TS == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            elsif SCR_EL3.TERR == '1' then
                AArch32.TakeMonitorTrapException();
            else
                ERXADDR2 = R[t];
            endif
        endif
    endif
else
    if PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
            UNDEFINED;
    elif PSTATE.EL == EL3 then
        if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
            AArch32.TakeMonitorTrapException();
    else
        ERXADDR2 = R[t];
    endif
endif
G8.6.6 ERXCTLR, Selected Error Record Control Register

The ERXCTLR characteristics are:

**Purpose**

Accesses bits [31:0] of ERR<n>CTRLR for the error record <n> selected by ERRSEL.R.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Arv8-A architecture profile.

**Configurations**

AArch32 System register ERXCTLR bits [31:0] are architecturally mapped to AArch64 System register ERXCTLR_EL1[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXCTLR are UNDEFINED.

**Attributes**

ERXCTLR is a 32-bit register.

**Field descriptions**

```
+----+----+
\    \  
| 31 | 0  |
\    \  
+----+----+
```

Bits [31:0] of ERR<n>CTRLR

ERXCTLR accesses bits [31:0] of ERR<n>CTRLR, where <n> is the value in ERRSEL.R.SEL.

**Accessing ERXCTLR**

If ERRIDR.NUM is 0x0000 or ERRSEL.R.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXCTLR is RAZ/WI.
- Direct reads and writes of ERXCTLR are NOPs.
- Direct reads and writes of ERXCTLR are UNDEFINED.

If ERRSEL.R.SEL is not the index of the first error record owned by a node, then ERR<n>CTRLR[31:0] is not present, meaning reads and writes of ERXCTLR are RES0.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
else
  R[t] = ERXCTLR;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
else
  R[t] = ERXCTLR;
elsif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
else
  R[t] = ERXCTLR;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  UNDEFINED;
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
  UNDEFINED;
else
  R[t] = ERXCTLR;
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  end if;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
  end if;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end if;
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch32.TakeMonitorTrapException();
    end if;
else
  ERXCTRLR = R[t];
elsif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
  else
    ERXCTRLR = R[t];
end if;
G8.6.7   ERXCTLR2, Selected Error Record Control Register 2

The ERXCTLR2 characteristics are:

**Purpose**

Accesses bits [63:32] of ERR<n>CTLR for the error record <n> selected by ERRSEL.R.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

AArch32 System register ERXCTLR2 bits [31:0] are architecturally mapped to AArch64 System register ERXCTLR_EL1[63:32].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXCTLR2 are UNDEFINED.

**Attributes**

ERXCTLR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>ERXCTLR2 accesses bits [63:32] of ERR&lt;n&gt;CTLR, where &lt;n&gt; is the value in ERRSEL.R.SEL.</td>
</tr>
</tbody>
</table>

**Accessing ERXCTLR2**

If ERRIDR.NUM is 0x0000 or ERRSEL.R.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXCTLR2 is RAZ/WI.
- Direct reads and writes of ERXCTLR2 are NOPs.
- Direct reads and writes of ERXCTLR2 are UNDEFINED.

If ERRSEL.R.SEL is not the index of the first error record owned by a node, then ERR<n>CTLR[63:32] is not present, meaning reads and writes of ERXCTLR2 are RES0.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>CRn</th>
<th>CRm</th>
<th>opc1</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b101</td>
<td>0b0100</td>
<td>0b000</td>
<td>0b111</td>
</tr>
</tbody>
</table>

```
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
```
 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 socially

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 statically

 static
AArch64.AArch32SystemAccessTrap(EL2, 0x03);  
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.TS == '1' then  
  AArch32.TakeHypTrapException(0x03);  
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then  
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);  
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then  
  AArch32.TakeHypTrapException(0x03);  
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then  
  if Halted() && EDSCR.SDD == '1' then  
    UNDEFINED;  
  else  
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);  
  endelsif  
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then  
  if Halted() && EDSCR.SDD == '1' then  
    UNDEFINED;  
  else  
    AArch32.TakeMonitorTrapException();  
else  
  ERXCTRL2 = R[t];  
elsif PSTATE_EL == EL2 then  
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then  
    UNDEFINED;  
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then  
    UNDEFINED;  
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then  
    if Halted() && EDSCR.SDD == '1' then  
      UNDEFINED;  
    else  
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);  
    endelsif  
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then  
    if Halted() && EDSCR.SDD == '1' then  
      UNDEFINED;  
    else  
      AArch32.TakeMonitorTrapException();  
else  
  ERXCTRL2 = R[t];  
elsif PSTATE_EL == EL3 then  
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then  
    AArch32.TakeMonitorTrapException();  
else  
  ERXCTRL2 = R[t];
G8.6.8 ERXFR, Selected Error Record Feature Register

The ERXFR characteristics are:

**Purpose**

Accesses bits [31:0] of ERR<n>FR for the error record <n> selected by ERRSELR.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

AArch32 System register ERXFR bits [31:0] are architecturally mapped to AArch64 System register ERXFR_EL1[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXFR are UNDEFINED.

**Attributes**

ERXFR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 0</td>
<td>ERXFR accesses bits [31:0] of ERR&lt;n&gt;FR, where &lt;n&gt; is the value in ERRSELR.SEL.</td>
</tr>
</tbody>
</table>

**Accessing ERXFR**

If ERRIDR.NUM is 0x0000 or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXFR is RAZ.
- Direct reads of ERXFR are NOPs.
- Direct reads of ERXFR are UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}\{<q>\} <\text{coproc}>, \{#<\text{opc1}>\}, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>\{, \{#<\text{opc2}>\}\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingArch32(EL3) & SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & ELUsingArch32(EL3) & SCR.TERR == '1' then
    UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    if boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'":
      if EL3Enabled() && SCR_EL3.TERR == '1' then
        R[t] = ERXFR;
      end if
    end if
  end if
elsif EL3Enabled() && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
  endif
else
  R[t] = ERXFR;
endif
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    if boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'":
      if EL3Enabled() && SCR_EL3.TERR == '1' then
        R[t] = ERXFR;
      end if
    end if
  end if
elsif EL3Enabled() && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
  endif
else
  R[t] = ERXFR;
endif
elsif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
  else
    R[t] = ERXFR;
  endif
end if
ERXFR2, Selected Error Record Feature Register 2

The ERXFR2 characteristics are:

Purpose

Accesses bits [63:32] of ERR<n>FR for the error record <n> selected by ERRSEL.R.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

Configurations

AArch32 System register ERXFR2 bits [31:0] are architecturally mapped to AArch64 System register ERXFR_EL1[63:32].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXFR2 are UNDEFINED.

Attributes

ERXFR2 is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bits [63:32] of ERR&lt;n&gt;FR</th>
</tr>
</thead>
</table>

Bits [31:0]

ERXFR2 accesses bits [63:32] of ERR<n>FR, where <n> is the value in ERRSEL.R.SEL.

Accessing ERXFR2

If ERRIDR.NUM is 0x0000 or ERRSEL.R.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

• An UNKNOWN error record is selected.
• ERXFR2 is RAZ.
• Direct reads of ERXFR2 are NOPs.
• Direct reads of ERXFR2 are UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC\{<c>\}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} \]

<table>
<thead>
<tr>
<th></th>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
\[ \text{UNDEFINED}; \]
else if PSTATE.EL == EL1 then
\[ \text{UNDEFINED}; \]
else if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
\[ \text{UNDEFINED}; \]
else if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
\[ \text{UNDEFINED}; \]
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    endif
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch32.TakeMonitorTrapException();
    endif
else
    R[t] = ERXFR2;
    elsif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
            UNDEFINED;
        elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
            UNDEFINED;
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.AArch32SystemAccessTrap(EL3, 0x03);
            endif
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch32.TakeMonitorTrapException();
            endif
        else
            R[t] = ERXFR2;
        endif
        elsif PSTATE_EL == EL3 then
            if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
                AArch32.TakeMonitorTrapException();
            else
                R[t] = ERXFR2;
            endif
        else
            R[t] = ERXFR2;
        endif
G8.6.10  ERXMISC0, Selected Error Record Miscellaneous Register 0

The ERXMISC0 characteristics are:

**Purpose**

Accesses bits [31:0] of ERR<n>MISC0 for the error record <n> selected by ERRSELR.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

AArch32 System register ERXMISC0 bits [31:0] are architecturally mapped to AArch64 System register ERXMISC0_EL1[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXMISC0 are UNDEFINED.

**Attributes**

ERXMISC0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERXMISC0 accesses bits [31:0] of ERR&lt;n&gt;MISC0, where &lt;n&gt; is the value in ERRSELR.SEL.</td>
</tr>
</tbody>
</table>

**Accessing ERXMISC0**

If ERRIDR.NUM is 0x0000 or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC0 is RAZ/WI.
- Direct reads and writes of ERXMISC0 are NOPs.
- Direct reads and writes of ERXMISC0 are UNDEFINED.

ERR<n>MISC0 describes additional constraints that also apply when ERR<n>MISC0 is accessed through ERXMISC0.

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC{<c>}{<q>} <coproc>, {#<opc1}, <Rt>, <CRn>, <CRm>{, {#<opc2}}> 
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
   UNDEFINED;
elsif PSTATE.EL == EL1 then
   if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
      UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && SCR_EL3.TERR == '1' then
  AArch32.TakeMonitorTrapException();
elsif Halted() && EDSCR.SDD == '1' then
  UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0x03);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
  else
    R[t] = ERXMISC0;
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  UNDEFINED;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
else
  R[t] = ERXMISC0;
elsif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
  else
    R[t] = ERXMISC0;
else
  if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  UNDEFINED;
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then

### MCR<coproc>({<q>},{<Rt>},{<CRn>},{<CRm>},{opc2})

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  UNDEFINED;
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then

MCR{<c>}{<q>}, <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>, {#}<opc2>
AArch64.AArch32SystemAccessTrap(EL2, 0x03);  
eelsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.TS == '1' then  
AArch32.TakeHypTrapException(0x03);  
eelsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then  
AArch64.AArch32SystemAccessTrap(EL2, 0x03);  
eelsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then  
AArch32.TakeHypTrapException(0x03);  
eelsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then  
if Halted() && EDSCR.SDD == '1' then  
UNDEFINED;  
eelse  
AArch64.AArch32SystemAccessTrap(EL3, 0x03);  
eelsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then  
if Halted() && EDSCR.SDD == '1' then  
UNDEFINED;  
eelse  
AArch32.TakeMonitorTrapException();  
eelse  
ERXMISC0 = R[t];  
eelsif PSTATE.EL == EL2 then  
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then  
UNDEFINED;  
eelsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then  
UNDEFINED;  
eelsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then  
if Halted() && EDSCR.SDD == '1' then  
UNDEFINED;  
eelse  
AArch64.AArch32SystemAccessTrap(EL3, 0x03);  
eelsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then  
if Halted() && EDSCR.SDD == '1' then  
UNDEFINED;  
eelse  
AArch32.TakeMonitorTrapException();  
eelse  
ERXMISC0 = R[t];  
eelsif PSTATE.EL == EL3 then  
if PSTATE.M != M32_Monitor && SCR.TERR == '1' then  
AArch32.TakeMonitorTrapException();  
eelse  
ERXMISC0 = R[t];
G8.6.11 ERXMISC1, Selected Error Record Miscellaneous Register 1

The ERXMISC1 characteristics are:

**Purpose**

Accesses bits [63:32] of ERR\(<n>\)MISC0 for the error record \(<n>\) selected by ERRSELR.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

AArch32 System register ERXMISC1 bits [31:0] are architecturally mapped to AArch64 System register ERXMISC0_EL1[63:32].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXMISC1 are UNDEFINED.

**Attributes**

ERXMISC1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:32] of ERR(&lt;n&gt;)MISC0</th>
</tr>
</thead>
</table>

**Bits [31:0]**

ERXMISC1 accesses bits [63:32] of ERR\(<n>\)MISC0, where \(<n>\) is the value in ERRSELR.SEL.

**Accessing ERXMISC1**

If ERRIDR.NUM is 0x0000 or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC1 is RAZ/WI.
- Direct reads and writes of ERXMISC1 are NOPs.
- Direct reads and writes of ERXMISC1 are UNDEFINED.

ERR\(<n>\)MISC0 describes additional constraints that also apply when ERR\(<n>\)MISC0 is accessed through ERXMISC1.

Accesses to this register use the following encodings in the System register encoding space:

\[\text{MRC}\{<c>\}{<q>}\ <\text{coproc}>, \{#\}<\text{opc1}>, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>, \{#, \{#\}<\text{opc2}>\}\]

\[
\begin{array}{cccc}
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} \\
0b111 & 0b000 & 0b010 & 0b010 & 0b001 \\
\end{array}
\]

if PSTATE.EL == EL0 then
 UNDEFINED;
elsif PSTATE.EL == EL1 then
 if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& !ELUsingAArch32(EL3) \&\& SCR_EL3.TERR == '1' then
 UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  UNDEFINED;
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.TS == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch32.TakeMonitorTrapException();
      endif
    else
      R[t] = ERXMISC1;
    endif
  else
    R[t] = ERXMISC1;
  endif
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      else
        AArch32.TakeMonitorTrapException();
      endif
    else
      R[t] = ERXMISC1;
    endif
  else
    R[t] = ERXMISC1;
  endif
elsif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
  else
    R[t] = ERXMISC1;
  endif
endif

MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>({, (#)<opc2>})

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.TS == '1' then
    UNDEFINED;
endif
AArch64.AArch32SystemAccessTrap(EL2, 0x03); elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then AArch32.TakeHypTrapException(0x03); elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then AArch64.AArch32SystemAccessTrap(EL2, 0x03); elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR2.TERR == '1' then AArch32.TakeHypTrapException(0x03); elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then if Halted() && EDSCR.SDD == '1' then UNDEFINED; else AArch64.AArch32SystemAccessTrap(EL3, 0x03); elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR == '1' then if Halted() && EDSCR.SDD == '1' then UNDEFINED; else AArch32.TakeMonitorTrapException(); else ERXMISC1 = R[t]; elsif PSTATE.EL == EL2 then if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SC_F1.TERR == '1' then UNDEFINED; elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then UNDEFINED; elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then if Halted() && EDSCR.SDD == '1' then UNDEFINED; else AArch64.AArch32SystemAccessTrap(EL3, 0x03); elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then if Halted() && EDSCR.SDD == '1' then UNDEFINED; else AArch32.TakeMonitorTrapException(); else ERXMISC1 = R[t]; elsif PSTATE.EL == EL3 then if PSTATE.M != M32_Monitor && SCR.TERR == '1' then AArch32.TakeMonitorTrapException(); else ERXMISC1 = R[t];
G8.6.12 ERXMISC2, Selected Error Record Miscellaneous Register 2

The ERXMISC2 characteristics are:

**Purpose**

Accesses bits [31:0] of ERR<n>MISC1 for the error record <n> selected by ERRSELR.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

AArch32 System register ERXMISC2 bits [31:0] are architecturally mapped to AArch64 System register ERXMISC1_EL1[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXMISC2 are UNDEFINED.

**Attributes**

ERXMISC2 is a 32-bit register.

**Field descriptions**

| Bits [31:0] | \_
|---|---

ERXMISC2 accesses bits [31:0] of ERR<n>MISC1, where <n> is the value in ERRSELR.SEL.

**Accessing ERXMISC2**

If ERRIDR.NUM is 0x0000 or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC2 is RAZ/WI.
- Direct reads and writes of ERXMISC2 are NOPs.
- Direct reads and writes of ERXMISC2 are UNDEFINED.

ERR<n>MISC1 describes additional constraints that also apply when ERR<n>MISC1 is accessed through ERXMISC2.

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>}\ <\text{coproc}, \ (#)\ <\text{opc1}, \ <\text{Rt}, \ <\text{CRn}, \ <\text{CRm}\{, \ (#)\ <\text{opc2}\}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b000</td>
<td>0b010</td>
<td>0b010</td>
<td>0b100</td>
</tr>
</tbody>
</table>

```bash
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        UNDEFINED;
```
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end if;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch32.TakeMonitorTrapException();
    end if;
else
    R[t] = ERXMISC2;
end if;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        end if;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch32.TakeMonitorTrapException();
        end if;
    else
        R[t] = ERXMISC2;
    end if;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        R[t] = ERXMISC2;
    end if;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        coproc opc1 CRn CRm opc2
        0b1111 0b000 0b0101 0b0101 0b100
    else
        R[t] = ERXMISC2;
    end if;
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        R[t] = ERXMISC2;
    end if;
else
    R[t] = ERXMISC2;
end if;

MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>, (#)<opc2>
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
eelsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.TS == '1' then
  AArch32.TakeHypTrapException(0x03);
eelsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
eelsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
  AArch32.TakeHypTrapException(0x03);
eelsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
eelsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
eelse
  ERXMSC2 = R[t];
eelsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
eelsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
eelsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
eelsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
eelse
  ERXMSC2 = R[t];
eelsif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
eelse
  ERXMSC2 = R[t];


G8.6.13 ERXMISC3, Selected Error Record Miscellaneous Register 3

The ERXMISC3 characteristics are:

**Purpose**

Accesses bits [63:32] of ERR<n>MISC1 for the error record <n> selected by ERRSELR.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

AArch32 System register ERXMISC3 bits [31:0] are architecturally mapped to AArch64 System register ERXMISC1_EL1[63:32].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXMISC3 are UNDEFINED.

**Attributes**

ERXMISC3 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:32] of ERR&lt;n&gt;MISC1</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 0</td>
</tr>
</tbody>
</table>

ERXMISC3 accesses bits [63:32] of ERR<n>MISC1, where <n> is the value in ERRSELR.SEL.

**Accessing ERXMISC3**

If ERRIDR.NUM is 0x0000 or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC3 is RAZ/WI.
- Direct reads and writes of ERXMISC3 are NOPs.
- Direct reads and writes of ERXMISC3 are UNDEFINED.

ERR<n>MISC1 describes additional constraints that also apply when ERR<n>MISC1 is accessed through ERXMISC3.

Accesses to this register use the following encodings in the System register encoding space:

\[ \text{MRC\{<c>\}{<q>} <coproc>, \{#\}<opc1>, <Rt>, <CRn>, <CRm>\{, \{#\}<opc2>\}} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  UNDEFINED;
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && EDSCR.SDD == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03); 
  endif
elsif EL2Enabled() && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
  endif
else
  R[t] = ERXMISC3;
elsif PSTATE.EL == EL2 then
  if Halted() && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  UNDEFINED;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch32.TakeMonitorTrapException();
  endif
else
  R[t] = ERXMISC3;
elsif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
else
  R[t] = ERXMISC3;
endif

MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>(, (#)<opc2>)

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b101</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
elsif Halted() && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
  UNDEFINED;
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0x03);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch32.TakeMonitorTrapException();
else
  ERXMISC3 = R[t];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0x03);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch32.TakeMonitorTrapException();
else
  ERXMISC3 = R[t];
elsif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
else
  ERXMISC3 = R[t];
G8.6.14 ERXMISC4, Selected Error Record Miscellaneous Register 4

The ERXMISC4 characteristics are:

**Purpose**

Accesses bits [31:0] of ERR<n>MISC2 for the error record <n> selected by ERRSELR.SEL.
For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification,
Armv8, for the Armv8-A architecture profile.

**Configurations**

AArch32 System register ERXMISC4 bits [31:0] are architecturally mapped to AArch64 System register ERXMISC2_EL1[31:0].
This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXMISC4 are UNDEFINED.

**Attributes**

ERXMISC4 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0] of ERR&lt;n&gt;MISC2</th>
</tr>
</thead>
</table>

ERXMISC4 accesses bits [31:0] of ERR<n>MISC2, where <n> is the value in ERRSELR.SEL.

**Accessing ERXMISC4**

If ERRIDR.NUM is 0x0000 or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC4 is RAZ/WI.
- Direct reads and writes of ERXMISC4 are NOPs.
- Direct reads and writes of ERXMISC4 are UNDEFINED.

ERR<n>MISC2 describes additional constraints that also apply when ERR<n>MISC2 is accessed through ERXMISC4.

Accesses to this register use the following encodings in the System register encoding space:

```plaintext
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end if;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch32.TakeMonitorTrapException();
    end if;
else
    R[t] = ERXMISC4;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        end if;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch32.TakeMonitorTrapException();
        end if;
else
    R[t] = ERXMISC4;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        R[t] = ERXMISC4;
    end if;
else
    MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        end if;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch32.TakeMonitorTrapException();
        end if;
else
    R[t] = ERXMISC4;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        UNDEFINED;
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        end if;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch32.TakeMonitorTrapException();
        end if;
else
    R[t] = ERXMISC4;
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        R[t] = ERXMISC4;
    end if;
else
    MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b010</td>
</tr>
</tbody>
</table>
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.T5 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR2.TERR == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  end
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
else
  ERXMISC4 = R[t];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch64.AArch32SystemAccessTrap(EL3, 0x03);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
else
  AArch32.TakeMonitorTrapException();
else
  ERXMISC4 = R[t];
elsif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
else
  ERXMISC4 = R[t];
G8.6.15 ERXMISC5, Selected Error Record Miscellaneous Register 5

The ERXMISC5 characteristics are:

**Purpose**

Accesses bits [63:32] of ERR<n>MISC2 for the error record <n> selected by ERRSEL.R.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

AArch32 System register ERXMISC5 bits [31:0] are architecturally mapped to AArch64 System register ERXMISC2_EL1[63:32].

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXMISC5 are UNDEFINED.

**Attributes**

ERXMISC5 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:32] of ERR&lt;n&gt;MISC2</th>
</tr>
</thead>
</table>

**Accessing ERXMISC5**

If ERRIDR.NUM is 0x0000 or ERRSEL.R.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC5 is RAZ/WI.
- Direct reads and writes of ERXMISC5 are NOPs.
- Direct reads and writes of ERXMISC5 are UNDEFINED.

ERR<n>MISC2 describes additional constraints that also apply when ERR<n>MISC2 is accessed through ERXMISC5.

Accesses to this register use the following encodings in the System register encoding space:

$$\text{MRC} \{<c>\} \{<q>\} <\text{coproc}>, \{<\text{opc1}>\}, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>, \{<\text{opc2}>\}$$

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1110</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & SCR_EL3.TERR == '1' then
        UNDEFINED;
elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.T5 == '1' then
    AArch32.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
    AArch32.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        if Halted() && EDSCR.SDD == '1' then
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        else
            AArch32.TakeMonitorTrapException();
        endif
    endif
else
    R[t] = ERXMISC5;
endif
endif

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
        UNDEFINED;
    elseif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
        UNDEFINED;
    endif
else
    R[t] = ERXMISC5;
endif

if PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        AArch32.TakeMonitorTrapException();
    endif
else
    R[t] = ERXMISC5;
endif
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && mLUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR2.TERR == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && mLUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  elsif
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    elsif HaveEL(EL3) && mLUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
      if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
      elsif
        AArch32.TakeMonitorTrapException();
        elsif HaveEL(EL3) && mLUsingAArch32(EL3) && SCR.TERR == '1' then
          if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
          elsif
            AArch32.TakeMonitorTrapException();
            elsif PSTATE.EL == EL2 then
              if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
                UNDEFINED;
              elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
                  UNDEFINED;
                elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
                  if Halted() && EDSCR.SDD == '1' then
                    UNDEFINED;
                  elsif
                    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
                    elsif HaveEL(EL3) && mLUsingAArch32(EL3) && SCR.TERR == '1' then
                      if Halted() && EDSCR.SDD == '1' then
                        UNDEFINED;
                      elsif
                        AArch32.TakeMonitorTrapException();
                        elsif PSTATE.EL == EL3 then
                          if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
                            AArch32.TakeMonitorTrapException();
                          else
                            ERXMISC5 = R[t];
                          else
                            PSTATE.EL == EL3 then
                              if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
                                AArch32.TakeMonitorTrapException();
                              else
                                ERXMISC5 = R[t];
                              else
                                ERXMISC5 = R[t];
                              else
                                ERXMISC5 = R[t];
                              else
                                ERXMISC5 = R[t];
                            else
                                ERXMISC5 = R[t];
                            else
                                ERXMISC5 = R[t];
                            else
                                ERXMISC5 = R[t];
                            else
                                ERXMISC5 = R[t];
                            else
                                ERXMISC5 = R[t];
                          else
                            ERXMISC5 = R[t];
                          else
                            ERXMISC5 = R[t];
                        else
                            ERXMISC5 = R[t];
                      else
                        ERXMISC5 = R[t];
                    else
                        ERXMISC5 = R[t];
                  else
                    ERXMISC5 = R[t];
                  else
                    ERXMISC5 = R[t];
                    elsif PSTATE.EL == EL2 then
                      if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
                        UNDEFINED;
                      elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
                          UNDEFINED;
                        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
                          if Halted() && EDSCR.SDD == '1' then
                            UNDEFINED;
                          elsif
                            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
                            elsif HaveEL(EL3) && mLUsingAArch32(EL3) && SCR.TERR == '1' then
                              if Halted() && EDSCR.SDD == '1' then
                                UNDEFINED;
                              elsif
                                AArch32.TakeMonitorTrapException();
                              elsif PSTATE.EL == EL3 then
                                if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
                                  AArch32.TakeMonitorTrapException();
                                else
                                  ERXMISC5 = R[t];
                                else
                                  ERXMISC5 = R[t];
                              else
                                ERXMISC5 = R[t];
                            else
                                ERXMISC5 = R[t];
                          else
                            ERXMISC5 = R[t];
                        else
                            ERXMISC5 = R[t];
                      else
                        ERXMISC5 = R[t];
                    else
                        ERXMISC5 = R[t];
                    else
                        ERXMISC5 = R[t];
                    else
                        ERXMISC5 = R[t];
                else
                    ERXMISC5 = R[t];
              else
                ERXMISC5 = R[t];
            else
              ERXMISC5 = R[t];
G8.6.16 ERXMISC6, Selected Error Record Miscellaneous Register 6

The ERXMISC6 characteristics are:

**Purpose**

Accesses bits [31:0] of ERR<\textless n\textgreater>MISC3 for the error record \textless n\textgreater{} selected by ERRSELR.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

AArch32 System register ERXMISC6 bits [31:0] are architecturally mapped to AArch64 System register ERXMISC3_EL1[31:0].

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXMISC6 are UNDEFINED.

**Attributes**

ERXMISC6 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>ERXMISC6 accesses bits [31:0] of ERR&lt;\textless n\textgreater&gt;MISC3, where &lt;n&gt; is the value in ERRSELR.SEL.</td>
</tr>
</tbody>
</table>

**Accessing ERXMISC6**

If ERRIDR.NUM is 0x0000 or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC6 is RAZ/WI.
- Direct reads and writes of ERXMISC6 are NOPs.
- Direct reads and writes of ERXMISC6 are UNDEFINED.

ERR<\textless n\textgreater>MISC3 describes additional constraints that also apply when ERR<\textless n\textgreater>MISC3 is accessed through ERXMISC6.

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC{\langle c\rangle}{\langle q\rangle} <\text{coproc}>, \{\#\text{opc1}\}, <\text{Rt}>, <\text{CRn}>, <\text{CRm}>, \{\#\text{opc2}\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>0b000</td>
<td>0b010</td>
<td>0b010</td>
<td>0b110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() \&\& HaveEL(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& !ELUsingAArch32(EL3) \&\& SCR_EL3.TERR == '1' then
    UNDEFINED;
elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'", & ELUsingAArch32(EL3) & SCR.TERR == '1' then UNDEFINED;
  elsif EL2Enabled() & !ELUsingAArch32(EL2) & HSTR_EL2.T5 == '1' then AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T5 == '1' then AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.TERR == '1' then AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & SCR_EL3.TERR == '1' then if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
  if PSTATE.M != M32_Monitor & SCR.TERR == '1' then AArch32.TakeMonitorTrapException();
else
  R[t] = ERXMISC6;
elsif PSTATE.EL == EL2 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'", & !ELUsingAArch32(EL3) & SCR_EL3.TERR == '1' then UNDEFINED;
  elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'", & ELUsingAArch32(EL3) & SCR.TERR == '1' then UNDEFINED;
  elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & SCR_EL3.TERR == '1' then if Halted() & EDSCR.SDD == '1' then UNDEFINED;
else
  if PSTATE.M != M32_Monitor & SCR.TERR == '1' then AArch32.TakeMonitorTrapException();
else
  R[t] = ERXMISC6;
elsif PSTATE.EL == EL3 then
  if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'", & !ELUsingAArch32(EL3) & SCR_EL3.TERR == '1' then UNDEFINED;
  elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'", & ELUsingAArch32(EL3) & SCR.TERR == '1' then UNDEFINED;
  elsif EL2Enabled() & !ELUsingAArch32(EL2) & HSTR_EL2.T5 == '1' then
    coproc opc1 CRn CRm opc2
    0b1111 0b000 0b0101 0b0101 0b110

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() & HaveEL(EL1) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'", & !ELUsingAArch32(EL1) & SCR_EL3.TERR == '1' then UNDEFINED;
  elsif Halted() & HaveEL(EL1) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'", & ELUsingAArch32(EL1) & SCR.TERR == '1' then UNDEFINED;
  elsif EL2Enabled() & !ELUsingAArch32(EL2) & HSTR_EL2.T5 == '1' then

MCR{<c>}{<q>} {coproc}, {#}<opc1>, <Rt>, <CRn>, <CRM>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b110</td>
</tr>
</tbody>
</table>
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.TS == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  endif
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
  endif
else
  ERXMISC6 = R[t];
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    endif
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch32.TakeMonitorTrapException();
    endif
else
  ERXMISC6 = R[t];
elsif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
  else
    ERXMISC6 = R[t];
endif
G8.6.17    ERXMISC7, Selected Error Record Miscellaneous Register 7

The ERXMISC7 characteristics are:

Purpose

Accesses bits [63:32] of ERR<n>MISC3 for the error record <n> selected by ERRSELR.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

Configurations

AArch32 System register ERXMISC7 bits [31:0] are architecturally mapped to AArch64 System register ERXMISC3_EL1[63:32].

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXMISC7 are UNDEFINED.

Attributes

ERXMISC7 is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Bits [63:32] of ERR&lt;n&gt;MISC3</td>
</tr>
</tbody>
</table>

Bits [31:0]

ERXMISC7 accesses bits [63:32] of ERR<n>MISC3, where <n> is the value in ERRSELR.SEL.

Accessing ERXMISC7

If ERRIDR.NUM is 0x0000 or ERRSELR.SEL is greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC7 is RAZ/WI.
- Direct reads and writes of ERXMISC7 are NOPs.
- Direct reads and writes of ERXMISC7 are UNDEFINED.

ERR<n>MISC3 describes additional constraints that also apply when ERR<n>MISC3 is accessed through ERXMISC7.

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC\{<c>\}\{<q>\} <coproc>, \{#\}<opc1>, <Rt>, <CRn>, <CRm>\{, \{#\}<opc2>\}}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if Halted() \&\& HaveEl(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& !ELUsingAArch32(EL3) \&\& SCR_EL3.TERR == '1' then
    UNDEFINED;
**MCR<coproc>,<CRn>, CRm, opc2**

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0101</td>
<td>0b111</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & !ELUsingAArch32(EL3) & SCR_EL3.TERR == '1' then
        UNDEFINED;
    elsif Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" & ELUsingAArch32(EL3) & SCR.TERR == '1' then
        UNDEFINED;
    elsif EL2Enabled() & !ELUsingAArch32(EL2) & HSTR_EL2.T5 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HSTR.T5 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR2.TERR == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif HaveEL(EL3) & !ELUsingAArch32(EL3) & SCR_EL3.TERR == '1' then
        if Halted() & & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            R[t] = ERXMISC7;
        end if
    elsif HaveEL(EL3) & ELUsingAArch32(EL3) & SCR_EL3.TERR == '1' then
        if Halted() & & EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch32.TakeMonitorTrapException();
        end if
    else
        R[t] = ERXMISC7;
    end if
elsif PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor & SCR.TERR == '1' then
        AArch32.TakeMonitorTrapException();
    else
        R[t] = ERXMISC7;
    end if
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
eelsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR.T5 == '1' then
  AArch32.TakeHypTrapException(0x03);
eelsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
eelsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR2.TERR == '1' then
  AArch32.TakeHypTrapException(0x03);
eelsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  end elsif
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
  end elsif
  AArch32.TakemonitorTrapException();
else
  ERXMISC7 = R[t];
eelsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
eelsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
eelsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  end elsif
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
  else
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
  else
    ERXMISC7 = R[t];
else
  if PSTATE.EL == EL3 then
    if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
      AArch32.TakeMonitorTrapException();
    else
      ERXMISC7 = R[t];
    end if
  else
    ERXMISC7 = R[t];
  end if
end if
end if
end if
end if
ERXSTATUS, Selected Error Record Primary Status Register

The ERXSTATUS characteristics are:

**Purpose**

Accesses bits [31:0] of ERR<n>STATUS for the error record selected by ERRSELR.SEL.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

AArch32 System register ERXSTATUS bits [31:0] are architecturally mapped to AArch64 System register ERXSTATUS_EL1[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXSTATUS are UNDEFINED.

**Attributes**

ERXSTATUS is a 32-bit register.

**Field descriptions**

Bits [31:0]

ERXSTATUS accesses bits [31:0] of ERR<n>STATUS, where n is the value in ERRSELR.SEL.

**Accessing ERXSTATUS**

If ERRIDR.NUM == 0 or ERRSELR.SEL is set to a value greater than or equal to ERRIDR.NUM, then one of the following occurs:

- An UNKNOWN record is selected.
- ERXSTATUS is RAZ/WI.
- Direct reads and writes of ERXSTATUS are NOPs.
- Direct reads and writes of ERXSTATUS are UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b0101</td>
<td>0b0100</td>
<td>0b010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elseif PSTATE.EL == EL1 then
  if Halted() \&\& HaveEl(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& EDUsingAArch32(EL3) \&\& SCR_EL3.TERR == '1' then
    UNDEFINED;
  elseif Halted() \&\& HaveEl(EL3) \&\& EDSCR.SDD == '1' \&\& boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" \&\& EDUsingAArch32(EL3) \&\& SCR.TERR == '1' then
    UNDEFINED;
  else
    UNDEFINED;
else
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
    AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        AArch32.TakeMonitorTrapException();
    end;
else
    R[t] = ERXSTATUS;
end;

R[t] = ERXSTATUS;
else
    PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x03);
        end
    elsif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' then
        UNDEFINED;
    end
end;

MCR<coproc>,<CRn>,<CRm>,<opc1>,<Rt>,<opc2>
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TERR == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR2.TERR == '1' then
  AArch32.TakeHypTrapException(0x03);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL3, 0x03);
  end if;  
elsif HaveEL(EL3) && ELUsingAArch32(EL3) && PSTATE.M != M32_Monitor && SCR.TERR == '1' then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    AArch32.TakeMonitorTrapException();
  end if;  
elsif PSTATE.EL == EL2 then
  if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    UNDEFINED;
  elseif Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    UNDEFINED;
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    end if;
  elseif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.TERR == '1' then
    if Halted() && EDSCR.SDD == '1' then
      UNDEFINED;
    else
      AArch32.TakeMonitorTrapException();
    end if;
  else
    ERXSTATUS = R[t];
  end if;
elsif PSTATE.EL == EL3 then
  if PSTATE.M != M32_Monitor && SCR.TERR == '1' then
    AArch32.TakeMonitorTrapException();
  else
    ERXSTATUS = R[t];
  end if;
else
  ERXSTATUS = R[t];
end if;
G8.6.19   VDFSR, Virtual SError Exception Syndrome Register

The VDFSR characteristics are:

**Purpose**

Provides the syndrome value reported to software on taking a virtual SError interrupt exception to EL1, or on executing an E58 instruction at EL1.

When the virtual SError interrupt injected using HCR.VA is taken to EL1 using AArch32, then the syndrome value is reported in DFSR.{AET, ExT} and the remainder of DFSR is set as defined by VMSAv8-32. For more information, see Chapter G5 The AArch32 Virtual Memory System Architecture.

If the virtual SError interrupt injected using HCR.VA is deferred by an E58 instruction, then the syndrome value is written to VDISR.

**Configurations**

AArch32 System register VDFSR bits [31:0] are architecturally mapped to AArch64 System register VSESR_EL2[31:0] when the highest implemented Exception level is using AArch64.

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to VDFSR are UNDEFINED.

If EL2 is not implemented, then VDFSR is RES0 from Monitor mode when SCR.NS == 1.

**Attributes**

VDFSR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>[31:16]</td>
</tr>
<tr>
<td>AET</td>
<td>[15:14]</td>
</tr>
<tr>
<td>RES0</td>
<td>[13:0]</td>
</tr>
</tbody>
</table>

**Bits [31:16]**

Reserved, RES0.

**AET, bits [15:14]**

When a virtual SError interrupt is taken to EL1 using AArch32, DFSR[15:14] is set to VDFSR.AET.

When a virtual SError interrupt is deferred by an E58 instruction, VDISR[15:14] is set to VDFSR.AET.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bit [13]**

Reserved, RES0.

**ExT, bit [12]**

When a virtual SError interrupt is taken to EL1 using AArch32, DFSR[12] is set to VDFSR.ExT.

When a virtual SError interrupt is deferred by an E58 instruction, VDISR[12] is set to VDFSR.ExT.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [11:0]**

Reserved, RES0.
Accessing VDFSR

Direct reads and writes of VDFSR are UNDEFINED if EL3 is implemented and using AArch32 in all Secure privileged modes other than Monitor mode.

If EL2 is not implemented, then VDFSR is RES0 from Monitor mode when SCR.NS == 1.

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC\{<c>\}{<q>} \{<opc1>, <Rt>, <CRn>, <CRm>\}{, {#}<opc2>} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNCHANGED;
elself PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == ’1’ then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elself EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == ’1’ then
        AArch32.TakeHypTrapException(0x03);
else
UNCHANGED;
elself PSTATE.EL == EL2 then
    R[t] = VDFSR;
elself PSTATE.EL == EL3 then
    if SCR.NS == ’0’ then
UNCHANGED;
elself R[t] = VDFSR;

\[ MCR\{<c>\}{<q>} \{<opc1>, <Rt>, <CRn>, <CRm>\}{, {#}<opc2>} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b0101</td>
<td>0b0010</td>
<td>0b011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNCHANGED;
elself PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T5 == ’1’ then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elself EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T5 == ’1’ then
        AArch32.TakeHypTrapException(0x03);
else
UNCHANGED;
elself PSTATE.EL == EL2 then
    VDFSR = R[t];
elself PSTATE.EL == EL3 then
    if SCR.NS == ’0’ then
UNCHANGED;
elself VDFSR = R[t];
G8.6.20  VDISR, Virtual Deferred Interrupt Status Register

The VDISR characteristics are:

Purpose

Records that an SError interrupt has been consumed by an E58 instruction.

Configurations

AArch32 System register VDISR bits [31:0] are architecturally mapped to AArch64 System register VDISR_EL2[31:0].

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to VDISR are UNDEFINED.

If EL2 is not implemented, then VDISR is RES0 from Monitor mode when SCR.NS == 1.

Attributes

VDISR is a 32-bit register.

Field descriptions

When TTBCR.EAE == 0:

A, bit [31]

Set to 1 when an E58 instruction defers a virtual SError interrupt.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:16]

Reserved, RES0.

AET, bits [15:14]

The value copied from VDFSR.AET.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

ExT, bit [12]

The value copied from VDFSR.ExT.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [11]

Reserved, RES0.
FS, bits [10, 3:0]
Fault status code. Set to 0b10110 when an E58 instruction defers a virtual SError interrupt.

0b10110  Asynchronous SError interrupt.
All other values are reserved.
The FS field is split as follows:
• FS[4] is VDISR[10].
• FS[3:0] is VDISR[3:0].
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

LPAE, bit [9]
Format.
Set to TTBCR.EAE when an E58 instruction defers a virtual SError interrupt.
0b0  Using the Short-descriptor translation table format.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:4]
Reserved, RES0.

When TTBCR.EAE == 1:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>RES0</td>
<td>AET</td>
<td>RES0</td>
<td>RES0</td>
<td>STATUS</td>
<td>RES0</td>
<td>EX</td>
<td>LPAE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A, bit [31]
Set to 1 when an E58 instruction defers a virtual SError interrupt.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:16]
Reserved, RES0.

AET, bits [15:14]
The value copied from VDFS.R.AET.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [13]
Reserved, RES0.

ExT, bit [12]
The value copied from VDFS.R.ExT.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:10]
Reserved, RES0.
LPAE, bit [9]

   Format.
   Set to TTBCR.EAE when an ESB instruction defers a virtual SError interrupt.
   0b1   Using the Long-descriptor translation table format.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

STATUS, bits [5:0]

   Fault status code. Set to 0b010001 when an ESB instruction defers a virtual SError interrupt.
   0b010001   Asynchronous SError interrupt.
   All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VDISR

Direct reads and writes of VDFSR are UNDEFINED if EL3 is implemented and using AArch32 in all Secure privileged modes other than Monitor mode.

An indirect write to VDISR made by an ESB instruction does not require an explicit synchronization operation for the value that is written to be observed by a direct read of DISR occurring in program order after the ESB instruction.

If EL2 is not implemented, then VDISR is RES0 from Monitor mode when SCR.NS == 1.

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>}\ <\text{coproc}>\ ,\ (<#>\text{opc1})\ ,\ (<Rt>)\ ,\ (<CRn>)\ ,\ (<CRm>)\ ,\ (<#>\text{opc2})
\]

\[
\begin{array}{cccc}
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
0b1111 & 0b100 & 0b100 & 0b0001 & 0b001 \\
\end{array}
\]

if PSTATE.EL == EL0 then
   UNDEFINED;
elsif PSTATE.EL == EL1 then
   if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
      elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
      AArch32.TakeHypTrapException(0x03);
      else
         UNDEFINED;
      elsif PSTATE.EL == EL2 then
         R[t] = VDISR;
      elsif PSTATE.EL == EL3 then
         if SCR.NS == '0' then
            UNDEFINED;
         else
            R[t] = VDISR;
         end if;
      else
      end if;
end if;
MCR<coproc>, <#opc1>, <Rt>, <CRn>, <CRm>{, <#opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1100</td>
<td>0b0001</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    VDISR = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        VDISR = R[t];

MRC<coproc>, <#opc1>, <Rt>, <CRn>, <CRm>{, <#opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0001</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAAArch32(EL2) && HCR_EL2.AMO == '1' then
        R[t] = VDISR_EL2<31:0>;
    elsif EL2Enabled() && ELUsingAAArch32(EL2) && HCR.AMO == '1' then
        R[t] = VDISR;
    elsif HaveEL(EL3) && !ELUsingAAArch32(EL3) && !Halted() && SCR_EL3.EA == '1' then
        R[t] = Zeros(32);
    elsif HaveEL(EL3) && ELUsingAAArch32(EL3) && !Halted() && SCR.EA == '1' then
        R[t] = Zeros(32);
    else
        R[t] = DISR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !ELUsingAAArch32(EL3) && !Halted() && SCR_EL3.EA == '1' then
        R[t] = Zeros(32);
    elsif HaveEL(EL3) && ELUsingAAArch32(EL3) && !Halted() && SCR.EA == '1' then
        R[t] = Zeros(32);
    else
        R[t] = DISR;
elsif PSTATE.EL == EL3 then
    R[t] = DISR;
MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1100</td>
<td>0b0001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  UNDEFINED;
elsif PSTATE_EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AMO == '1' then
    VDISR_EL2 = R[t];
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.AMO == '1' then
    VDISR = R[t];
  elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && !Halted() && SCR_EL3.EA == '1' then
    return;
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && !Halted() && SCR.EA == '1' then
    return;
  else
    DISR = R[t];
else PSTATE_EL == EL2 then
  if HaveEL(EL3) && !ELUsingAArch32(EL3) && !Halted() && SCR_EL3.EA == '1' then
    return;
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) && !Halted() && SCR.EA == '1' then
    return;
  else
    DISR = R[t];
elsif PSTATE_EL == EL3 then
  DISR = R[t];
G8.7  Generic Timer registers

This section lists the Generic Timer registers in AArch32.
### G8.7.1 CNTFRQ, Counter-timer Frequency register

The CNTFRQ characteristics are:

#### Purpose
This register is provided so that software can discover the frequency of the system counter. It must be programmed with this value as part of system initialization. The value of the register is not interpreted by hardware.

#### Configurations
AArch32 System register CNTFRQ bits [31:0] are architecturally mapped to AArch64 System register CNTFRQ_EL0[31:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to CNTFRQ are UNDEFINED.

#### Attributes
CNTFRQ is a 32-bit register.

#### Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock frequency</td>
<td></td>
</tr>
</tbody>
</table>

#### Bits [31:0]
Clock frequency. Indicates the system counter clock frequency, in Hz.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTFRQ**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC\langle c\rangle\langle q\rangle \langle coproc\rangle, \langle \#\rangle\langle opc1\rangle, \langle Rt\rangle, \langle CRn\rangle, \langle CRm\rangle, \langle \#\rangle\langle opc2\rangle
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.<EL0PCTEN,EL0VCTEN> == '00' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PCTEN == '0' && CNTKCTL.PL0VCTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch32.TakeHypTrapException(0x00);
    else
      UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.<EL0PCTEN,EL0VCTEN> == '00' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
  R[t] = CNTFRQ;
elsif PSTATE.EL == EL1 then
  R[t] = CNTFRQ;
elsif PSTATE.EL == EL2 then
  R[t] = CNTFRQ;
elsif PSTATE.EL == EL3 then
  R[t] = CNTFRQ;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if IsHighestEL(PSTATE.EL) then
  CNTFRQ = R[t];
else
  UNDEFINED;
G8.7.2 CNGTHCTL, Counter-timer Hyp Control register

The CNGTHCTL characteristics are:

**Purpose**
Controls the generation of an event stream from the physical counter, and access from Non-secure EL1 modes to the physical counter and the Non-secure EL1 physical timer.

**Configurations**
AArch32 System register CNGTHCTL bits [31:0] are architecturally mapped to AArch64 System register CNGTHCTL_EL2[31:0].
This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to CNGTHCTL are UNDEFINED.
If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**
CNGTHCTL is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>17</td>
<td>EVNTIS, bit [17]</td>
</tr>
<tr>
<td>8-16</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>7-4</td>
<td>EVNTI, bits [7:4]</td>
</tr>
<tr>
<td>0</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**EVNTIS, bit [17]**
*When FEAT_ECV is implemented:*
- 0: The CNGTHCTL.EVNTI field applies to CNTPCT[15:0].
- 1: The CNGTHCTL.EVNTI field applies to CNTPCT[23:8].
The reset behavior of this field is:
  - On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
- Reserved, RES0.

**Bits [16:8]**
- Reserved, RES0.

**EVNTI, bits [7:4]**
Selects which bit of CNTPCT, as seen from EL2, is the trigger for the event stream generated from that counter when that stream is enabled.
If FEAT_ECV is implemented, and CNGTHCTL.EVNTIS is 1, this field selects a trigger bit in the range 8 to 23 of CNTPCT.
Otherwise, this field selects a trigger bit in the range 0 to 15 of CNTPCT.
The reset behavior of this field is:
  - On a Warm reset, this field resets to an architecturally UNKNOWN value.
**EVNTDIR, bit [3]**

Controls which transition of the CNTPCT trigger bit, as seen from EL2 and defined by EVNTI, generates an event when the event stream is enabled.

- **0b0** A 0 to 1 transition of the trigger bit triggers an event.
- **0b1** A 1 to 0 transition of the trigger bit triggers an event.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**EVNTEN, bit [2]**

Enables the generation of an event stream from CNTPCT as seen from EL2.

- **0b0** Disables the event stream.
- **0b1** Enables the event stream.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**PL1PCEN, bit [1]**

Traps Non-secure EL0 and EL1 accesses to the physical timer registers to Hyp mode.

- **0b0** Non-secure EL0 and EL1 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVVAL are trapped to Hyp mode, unless the it is trapped by CNTKCTL.PL0PTEN.
- **0b1** This control does not cause any instructions to be trapped.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**PL1PCTEN, bit [0]**

Traps Non-secure EL0 and EL1 accesses to the physical counter register to Hyp mode.

- **0b0** Non-secure EL0 and EL1 accesses to the CNTPCT are trapped to Hyp mode, unless it is trapped by CNTKCTL.PL0PCTEN.
- **0b1** This control does not cause any instructions to be trapped.

If EL3 is implemented and EL2 is not implemented, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

### Accessing CNTHCTL

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0001</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
```
R[t] = CNTHCTL;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    R[t] = CNTHCTL;

MCR<coproc>,<Rt>,<CRn>,<CRm>{,#<opc1>},{,#<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  CNTHCTL = R[t];
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    CNTHCTL = R[t];
G8.7.3 CNTHP_CTL, Counter-timer Hyp Physical Timer Control register

The CNTHP_CTL characteristics are:

Purpose
Control register for the Hyp mode physical timer.

Configurations
AArch32 System register CNTHP_CTL bits [31:0] are architecturally mapped to AArch64 System register CNTHP_CTL_EL2[31:0].
This register is present only when AArch32 is supported. Otherwise, direct accesses to CNTHP_CTL are UNDEFINED.
If EL2 is not implemented, this register is RES0 from EL3.

Attributes
CNTHP_CTL is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:3</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>
| 2 | ISTATUS, bit [2]
The status of the timer. This bit indicates whether the timer condition is met:
0b0 Timer condition is not met.
0b1 Timer condition is met.
When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.
When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value. Access to this field is RO.

| 1 | IMASK, bit [1]
Timer interrupt mask bit. Permitted values are:
0b0 Timer interrupt is not masked by the IMASK bit.
0b1 Timer interrupt is masked by the IMASK bit.
For more information, see the description of the ISTATUS bit.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

| 0 | ENABLE, bit [0]
Enables the timer. Permitted values are:
0b0 Timer disabled.
0b1 Timer enabled.
Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHP_TV continues to count down.

______ Note ________
Disabling the output signal might be a power-saving option.

The reset behavior of this field is:
• On a Warm reset:
  — When the PE resets into EL2 or EL3, this field resets to 0.
  — Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing CNTHP_CTL

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  R[t] = CNTHP_CTL;
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    R[t] = CNTHP_CTL;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  CNTHP_CTL = R[t];
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    CNTHP_CTL = R[t];
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !ELEnabled() && HCR_EL2.E2H == '1' && CNTKCTL_EL1.EL0PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    endif
    elsif ELEnabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        UNDEFINED;
    endif
    elsif ELEnabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        ELEnabled() && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    endif
    elsif ELEnabled() && HCR_EL2.TGE == '1' then
        AArch32.TakeHypTrapException(0x00);
    elsif ELEnabled() && HCR_EL2.E2H == '0' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
        R[t] = CNTHPS_CTL_EL2<31:0>;
    elsif ELEnabled() && HCR_EL2.E2H == '1' && SCR_EL3.NS == '1' then
        R[t] = CNTHP_CTL_EL2<31:0>;
    else
        R[t] = CNTP_CTL;
    endif
    elsif PSTATE.EL == EL1 then
        if ELEnabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            ELEnabled() && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x03);
            elsif ELEnabled() && HCR_EL2.TGE == '1' then
                AArch32.TakeHypTrapException(0x03);
            else
                R[t] = CNTP_CTL_NS;
            endif
        elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
            R[t] = CNTP_CTL_NS;
        else
            R[t] = CNTP_CTL;
        endif
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && ELUsingAArch32(EL3) then
            R[t] = CNTP_CTL_NS;
        else
            R[t] = CNTP_CTL;
        endif
    elsif PSTATE.EL == EL3 then
        if SCR.NS == '0' then
            R[t] = CNTP_CTL_S;
        else
            R[t] = CNTP_CTL_NS;
        endif
    else
        R[t] = CNTP_CTL_NS;
MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !ELUsingAArch32(EL1) && !EL2Enabled() && HCR_EL2.<E2H,TGE> == '1' && CNTKCTL_EL1.EL0PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  elsif ELUsingAArch32(EL1) && !EL2Enabled() && HCR_EL2.<E2H,TGE> == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    UNDEFINED;
  endif
else
  if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTMCTL_EL2.EL0PCENT == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTMCTL_EL2.EL0PCENT == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    UNDEFINED;
  endif
else
  if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTMCTL_EL2.EL1PCENT == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTMCTL_EL2.EL1PCENT == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    UNDEFINED;
  endif
else
  if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTMCTL_EL2.EL1PCENT == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
    IsFeatureImplemented(FEAT_SEL2) then
    CNTMPS_CTL_EL2 = R[t];
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
    CNTMPS_CTL_EL2 = R[t];
  else
    CNTM_CTL = R[t];
  endif
else
  if PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTMCTL_EL2.EL1PCENT == '0' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '1' && CNTMCTL_EL2.EL1PCENT == '0' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTMPS_EL3.PCEN == '0' then
      AArch32.TakeHypTrapException(0x03);
    else
      CNTM_CTL = R[t];
    endif
  elsif PSTATE.EL == EL2 then
    if EL2Enabled() && ELUsingAArch32(EL3) then
      CNTM_CTL_NS = R[t];
    else
      CNTM_CTL = R[t];
    endif
  elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
      CNTM_CTL_S = R[t];
    else
      CNTM_CTL_NS = R[t];
    endif
  else
    CNTM_CTL = R[t];
  endif
endif
G8.7.4 CNTHP_CVAL, Counter-timer Hyp Physical CompareValue register

The CNTHP_CVAL characteristics are:

**Purpose**

Holds the compare value for the Hyp mode physical timer.

**Configurations**

AArch32 System register CNTHP_CVAL bits [63:0] are architecturally mapped to AArch64 System register CNTHP_CVAL_EL2[63:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to CNTHP_CVAL are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

**Attributes**

CNTHP_CVAL is a 64-bit register.

**Field descriptions**

**CompareValue, bits [63:0]**

Holds the EL2 physical timer CompareValue.

When CNTHP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer.

When the timer condition is met:

- CNTHP_CTL.ISTATUS is set to 1.
- If CNTHP_CTL.IMASK is 0, an interrupt is generated.

When CNTHP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTHP_CVAL**

Accesses to this register use the following encodings in the System register encoding space:

\[ \text{MRRC}\{\text{c}\}\{\text{q}\} \langle\text{coproc}\rangle, \langle\#\text{opc1}\rangle, \langle\text{Rt}\rangle, \langle\text{Rt2}\rangle, \langle\text{CRm}\rangle \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1110</td>
<td>0b0110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elsif PSTATE.EL == EL1 then
UNDEFINED;
elseif PSTATE.EL == EL2 then
    (R[t2], R[t]) = (CNTHP_CVAL<63:32>, CNTHP_CVAL<31:0>);
elseif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        (R[t2], R[t]) = (CNTHP_CVAL<63:32>, CNTHP_CVAL<31:0>);
    end

MCRR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <Rt2>, <CRm>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1110</td>
<td>0b0110</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
UNDEFINED;
elseif PSTATE.EL == EL1 then
UNDEFINED;
elseif PSTATE.EL == EL2 then
    CNTHP_CVAL = R[t2]:R[t];
elseif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        UNDEFINED;
    else
        CNTHP_CVAL = R[t2]:R[t];
    end

MRRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <Rt2>, <CRm>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1110</td>
<td>0b0010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL2.EL0PTEN == '0' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
        end
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        UNDEFINED;
    end
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTKCTL_EL2.EL1PCEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR.TGE == '1' then
    AArch32.TakeHypTrapException(0x00);
else
    UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTKCTL_EL2.EL1PCEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '10' && CNTKCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '11' && CNTKCTL_EL2.EL0PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '11' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && CNTKCTL.PL1PCEN == '0' then
    AArch32.TakeHypTrapException(0x00);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '11' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
(R[t2], R[t]) = (CNTHPS_CVAL_EL2<63:32>, CNTHPS_CVAL_EL2<31:0>);
elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.<E2H,TGE> == '11' & SCR_EL3.NS == '1' then
  (R[t2], R[t]) = (CNTHP_CVAL_EL2<63:32>, CNTHP_CVAL_EL2<31:0>);
else
  (R[t2], R[t]) = (CNTP_CVAL<63:32>, CNTP_CVAL<31:0>);
elsif PSTATE_EL == EL1 then
  if EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.E2H == '0' & HCR_EL2.E2H == '0' & CNTHCTL_EL2.E1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.E2H == '1' & HCR_EL2.E2H == '1' & CNTHCTL_EL2.E1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() & ELUsingAArch32(EL2) & CNTHCTL.PL1PCEN == '0' then
    AArch32.TakeHypTrapException(0x04);
  elsif HaveEL(EL3) & ELUsingAArch32(EL3) then
    (R[t2], R[t]) = (CNTP_CVAL_NS<63:32>, CNTP_CVAL_NS<31:0>);
  else
    (R[t2], R[t]) = (CNTP_CVAL<63:32>, CNTP_CVAL<31:0>);
elsif PSTATE_EL == EL2 then
  if HaveEL(EL3) & ELUsingAArch32(EL3) then
    (R[t2], R[t]) = (CNTP_CVAL_NS<63:32>, CNTP_CVAL_NS<31:0>);
  else
    (R[t2], R[t]) = (CNTP_CVAL<63:32>, CNTP_CVAL<31:0>);
elsif PSTATE_EL == EL3 then
  if SCR.NS == '0' then
    (R[t2], R[t]) = (CNTP_CVAL_S<63:32>, CNTP_CVAL_S<31:0>);
  else
    (R[t2], R[t]) = (CNTP_CVAL_NS<63:32>, CNTP_CVAL_NS<31:0>);
elsif PSTATE_EL == EL0 then
  if ELUsingAArch32(EL1) & !EL2Enabled() & HCR_EL2.<E2H,TGE> == '11' & CNTHCTL_EL1.EL0PTEN == '0' then
    if EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    end;
  elsif ELUsingAArch32(EL1) & CNTHCTL.PL0PTEN == '0' then
    if EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
      UNDEFINED;
  end;
  elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.<E2H,TGE> == '11' & CNTHCTL_EL2.E1PTEN == '0' then
    if EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.E2H == '0' & CNTHCTL_EL2.E1PTEN == '0' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.<E2H,TGE> == '10' & CNTHCTL_EL2.E1PTEN == '0' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.<E2H,TGE> == '11' & CNTHCTL_EL2.E1PTEN == '0' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.<E2H,TGE> == '11' & CNTHCTL_EL2.E1PTEN == '0' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & CNTHCTL.PL1PCEN == '0' then
      AArch32.TakeHypTrapException(0x04);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.<E2H,TGE> == '11' & SCR_EL3.NS == '0' & IsFeatureImplemented(FEAT_SEL2) then
      CNTHPS_CVAL_EL2 = R[t2]:R[t];
    elsif EL2Enabled() & ELUsingAArch32(EL2) & HCR_EL2.<E2H,TGE> == '11' & SCR_EL3.NS == '1' then
      CNTHP_CVAL_EL2 = R[t2]:R[t];
    else
      CNTP_CVAL = R[t2]:R[t];

MCRR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <Rt2>, <CRm>
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
    AArch32.TakeHypTrapException(0x04);
  elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    CNTP_CVAL_NS = R[t2]:R[t];
  else
    CNTP_CVAL = R[t2]:R[t];
  endif
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    CNTP_CVAL_NS = R[t2]:R[t];
  else
    CNTP_CVAL = R[t2]:R[t];
  endif
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    CNTP_CVAL_S = R[t2]:R[t];
  else
    CNTP_CVAL_NS = R[t2]:R[t];
  endif
G8.7.5   CNTHP_TVAL, Counter-timer Hyp Physical Timer TimerValue register

The CNTHP_TVAL characteristics are:

Purpose

Holds the timer value for the Hyp mode physical timer.

Configurations

AArch32 System register CNTHP_TVAL bits [31:0] are architecturally mapped to AArch64 System register CNTHP_TVAL_EL2[31:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to CNTHP_TVAL are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3.

Attributes

CNTHP_TVAL is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimerValue</td>
<td></td>
</tr>
</tbody>
</table>

TimerValue, bits [31:0]

The TimerValue view of the EL2 physical timer.

On a read of this register:

- If CNTHP_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHP_CTL.ENABLE is 1, the value returned is (CNTHP_CVAL - CNTPCT).

On a write of this register, CNTHP_CVAL is set to (CNTPCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CNTHP_CVAL) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer.

When the timer condition is met:

- CNTHP_CTL.ISTATUS is set to 1.
- If CNTHP_CTL.IMASK is 0, an interrupt is generated.

When CNTHP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHP_TVAL

Accesses to this register use the following encodings in the System register encoding space:
**G8.7 Generic Timer registers**

### MRC{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  if CNTHP_CTL.ENABLE == '0' then
    R[t] = bits(32) UNKNOWN;
  else
    R[t] = (CNTHP_CVAL - PhysicalCountInt())<31:0>
  elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
      UNDEFINED;
    else
      if CNTHP_CTL.ENABLE == '0' then
        R[t] = bits(32) UNKNOWN;
      else
        R[t] = (CNTHP_CVAL - PhysicalCountInt())<31:0>
      end
    end
end

### MCR{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b100</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  CNTHP_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    CNTHP_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
end

### MRC{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    end
  elsif ELUsingAArch32(EL1) && CNTKCTL.PLOPTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    end
  end
end
AArch32 System Register Descriptions
G8.7 Generic Timer registers

AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
AArch32.TakeHypTrapException(0x00);
else
UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0'
then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0'
then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FEAT_SEL2) then
if CNTHPS_CTL_EL2.ENABLE == '0' then
R[t] = bits(32) UNKNOWN;
else
R[t] = (CNTHPS_CVAL_EL2 - PhysicalCountInt())<31:0>;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
if CNTHP_CTL_EL2.ENABLE == '0' then
R[t] = bits(32) UNKNOWN;
else
R[t] = (CNTHP_CVAL_EL2 - PhysicalCountInt())<31:0>;
elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
&& CNTHCTL_EL2.ECV == '1' && HCR_EL2.<E2H,TGE> != '11' then
if CNTP_CTL.ENABLE == '0' then
R[t] = bits(32) UNKNOWN;
else
R[t] = (CNTP_CVAL - (PhysicalCountInt() - CNTPOFF_EL2))<31:0>;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
if SCR.NS == '1' then
if CNTP_CTL_NS.ENABLE == '0' then
R[t] = bits(32) UNKNOWN;
else
R[t] = (CNTP_CVAL_NS - PhysicalCountInt())<31:0>;
else
if CNTP_CTL_S.ENABLE == '0' then
R[t] = bits(32) UNKNOWN;
else
R[t] = (CNTP_CVAL_S - PhysicalCountInt())<31:0>;
else
if CNTP_CTL.ENABLE == '0' then
R[t] = bits(32) UNKNOWN;
else
R[t] = (CNTP_CVAL - PhysicalCountInt())<31:0>;
elsif PSTATE.EL == EL1 then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
AArch32.TakeHypTrapException(0x03);
elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
&& CNTHCTL_EL2.ECV == '1' then
if CNTP_CTL.ENABLE == '0' then
R[t] = bits(32) UNKNOWN;
else
R[t] = (CNTP_CVAL - (PhysicalCountInt() - CNTPOFF_EL2))<31:0>;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
if CNTP_CTL_NS.ENABLE == '0' then
R[t] = bits(32) UNKNOWN;
else
R[t] = (CNTP_CVAL_NS - PhysicalCountInt())<31:0>;
else
if CNTP_CTL.ENABLE == '0' then

G8-10160

Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ARM DDI 0487I.a
ID081822


R[t] = bits(32) UNKNOWN;
else
R[t] = (CNTP_CVAL - PhysicalCountInt())<31:0>;
elsif PSTATE.EL == EL2 then
if HaveEL(EL3) && ELUsingAArch32(EL3) then
if CNTP_CTL_NS.ENABLE == '0' then
R[t] = bits(32) UNKNOWN;
else
R[t] = (CNTP_CVAL_NS - PhysicalCountInt())<31:0>;
else
if CNTP_CTL.ENABLE == '0' then
R[t] = bits(32) UNKNOWN;
else
R[t] = (CNTP_CVAL - PhysicalCountInt())<31:0>;
elsif PSTATE.EL == EL3 then
if SCR.NS == '0' then
if CNTP_CTL_S.ENABLE == '0' then
R[t] = bits(32) UNKNOWN;
else
R[t] = (CNTP_CVAL_S - PhysicalCountInt())<31:0>;
else
if CNTP_CTL_NS.ENABLE == '0' then
R[t] = bits(32) UNKNOWN;
else
R[t] = (CNTP_CVAL_NS - PhysicalCountInt())<31:0>;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0' then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
AArch32.TakeHypTrapException(0x00);
else
UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FEAT_SEL2) then
CNTHPS_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
elsif EL2Enabled() && ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
& CNTHCTL_EL2.ECV == '1' && HCR_EL2.<E2H,TGE> != '11' then
CNTHPS_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
nbsp; elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
& CNTHCTL_EL2.ECV == '1' && HCR_EL2.<E2H,TGE> != '11' then
CNTHPS_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
nbsp; elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
& CNTHCTL_EL2.ECV == '1' && HCR_EL2.<E2H,TGE> != '11' then
CNTHPS_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b000</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0' then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
AArch32.TakeHypTrapException(0x00);
else
UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FEAT_SEL2) then
CNTHPS_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
elsif EL2Enabled() && ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
& CNTHCTL_EL2.ECV == '1' && HCR_EL2.<E2H,TGE> != '11' then
CNTHPS_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
nbsp; elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
& CNTHCTL_EL2.ECV == '1' && HCR_EL2.<E2H,TGE> != '11' then
CNTHPS_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
nbsp; elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
& CNTHCTL_EL2.ECV == '1' && HCR_EL2.<E2H,TGE> != '11' then
CNTHPS_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
elsif HaveEL(EL3) & ELUsingAArch32(EL3) then
  if SCR.NS == '1' then
    CNTP_CVAL_NS = SignExtend(R[t], 64) + PhysicalCountInt();
  else
    CNTP_CVAL_S = SignExtend(R[t], 64) + PhysicalCountInt();
  else
    CNTP_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
  elsif PSTATE.EL == EL1 then
    if EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.E2H == '0' & CNTHCTL_EL2.E2PCEN == '0' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() & !ELUsingAArch32(EL2) & HCR_EL2.E2H == '1' & CNTHCTL_EL2.E2PTEN == '0' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() & ELUsingAArch32(EL2) & CNTHCTL.PL1PCEN == '0' then
      AArch32.TakeHypTrapException(0x03);
    elsif IsFeatureImplemented(FEAT_ECV) & EL2Enabled() & !ELUsingAArch32(EL2) & SCR_EL3.ECVEn == '1'
      & CNTHCTL_EL2.ECV == '1' then
      CNTP_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTPOFF_EL2;
    elsif HaveEL(EL3) & ELUsingAArch32(EL3) then
      CNTP_CVAL_NS = SignExtend(R[t], 64) + PhysicalCountInt();
    else
      CNTP_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
  elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) & ELUsingAArch32(EL3) then
      CNTP_CVAL_NS = SignExtend(R[t], 64) + PhysicalCountInt();
    else
      CNTP_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
  elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
      CNTP_CVAL_S = SignExtend(R[t], 64) + PhysicalCountInt();
    else
      CNTP_CVAL_NS = SignExtend(R[t], 64) + PhysicalCountInt();
  }
G8.7.6 CNTHPS_CTL, Counter-timer Secure Physical Timer Control Register (EL2)

The CNTHPS_CTL characteristics are:

**Purpose**

Provides AArch32 access from EL0 to the Secure EL2 physical timer.

**Configurations**

AArch32 System register CNTHPS_CTL bits [31:0] are architecturally mapped to AArch64 System register CNTHPS_CTL_EL2[31:0].

This register is present only when AArch32 is supported and FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHPS_CTL are UNDEFINED.

**Attributes**

CNTHPS_CTL is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Permitted Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:3]</td>
<td>Reserved, RES0</td>
<td>0x0</td>
</tr>
<tr>
<td>[2]</td>
<td>ISTATUS, bit [2]</td>
<td>0b0, 0b1</td>
</tr>
<tr>
<td>[1]</td>
<td>IMASK, bit [1]</td>
<td>0b0, 0b1</td>
</tr>
<tr>
<td>[0]</td>
<td>ENABLE, bit [0]</td>
<td>0b0, 0b1</td>
</tr>
</tbody>
</table>

**Bits [31:3]**

Reserved, RES0.

**ISTATUS, bit [2]**

The status of the timer. This bit indicates whether the timer condition is met:

- **0b0**: Timer condition is not met.
- **0b1**: Timer condition is met.

When the value of the CNTHPS_CTL.ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the CNTHPS_CTL.ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

**IMASK, bit [1]**

Timer interrupt mask bit. Permitted values are:

- **0b0**: Timer interrupt is not masked by the IMASK bit.
- **0b1**: Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ENABLE, bit [0]**

Enables the timer. Permitted values are:

- **0b0**: Timer disabled.
- **0b1**: Timer enabled.
Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHPS_TV AL_EL2 continues to count down.

--- Note ---
Disabling the output signal might be a power-saving option.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTHPS_CTL**

This register is accessed using the encoding for CNTP_CTL.

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if !ELUsingAArch32(EL1) && !ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    AArch32.TakeHypTrapException(0x00);
    else
      UNDEFINED;
  elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    R[t] = CNTHP_CTL_EL2<31:0>;
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      UNDEFINED;
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    R[t] = CNTHP_CTL_EL2<31:0>;
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      UNDEFINED;
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    R[t] = CNTHP_CTL_EL2<31:0>;
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      UNDEFINED;
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    R[t] = CNTHP_CTL_EL2<31:0>;
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      UNDEFINED;
  elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    R[t] = CNTHP_CTL_EL2<31:0>;
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      UNDEFINED;
R[t] = CNTP_CTL;
else if PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    R[t] = CNTP_CTL_NS;
  else
    R[t] = CNTP_CTL;
else if PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    R[t] = CNTP_CTL_S;
  else
    R[t] = CNTP_CTL_NS;
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b0000</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.E2H,TGE == '11') && CNTKCTL_EL1.EL0PTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    endif
  else
    UNDEFINED;
  endif
else if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && HCR_EL2.TGE == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
  UNDEFINED;
endif
else if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '10' && CNTKCTL_EL2.EL1PTEN == '0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '11' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
  CNTHPS_CTL_EL2 = R[t];
else if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '11' && SCR_EL3.NS == '1' then
  CNTHP_CTL_EL2 = R[t];
else
  CNTP_CTL = R[t];
else if PSTATE_EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTKCTL_EL2.EL1PCEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    UNDEFINED;
  endif
else if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTKCTL_EL2.EL1PTEN == '0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' && CNTKCTL_EL2.EL1PTEN == '0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else if EL2Enabled() && ELUsingAArch32(EL2) && CNTKCTL.PL1PCEN == '0' then
  AArch32.TakeHypTrapException(0x03);
else if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTKCTL_EL2.EL0PTEN == '0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
  UNDEFINED;
endif
else if PSTATE_EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    CNTP_CTL_NS = R[t];
  else
    CNTP_CTL = R[t];
else if PSTATE_EL == EL3 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    CNTP_CTL_NS = R[t];
  else
    CNTP_CTL = R[t];
CNP_TCTL = R[t];
elsif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        CNTP_CTL_S = R[t];
    else
        CNTP_CTL_NS = R[t];
    end
end
G8.7.7  CNTHPS_CVAL, Counter-timer Secure Physical Timer CompareValue Register (EL2)

The CNTHPS_CVAL characteristics are:

**Purpose**

Provides AArch32 access from EL0 to the compare value for the Secure EL2 physical timer.

**Configurations**

AArch32 System register CNTHPS_CVAL bits [63:0] are architecturally mapped to AArch64 System register CNTHPS_CVAL_EL2[63:0].

This register is present only when AArch32 is supported and FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHPS_CVAL are UNDEFINED.

**Attributes**

CNTHPS_CVAL is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompareValue</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CompareValue, bits [63:0]**

Holds the EL2 physical timer CompareValue.

When CNTHPS_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTHPS_CTL_EL2.ISTATUS is set to 1.
- If CNTHPS_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHPS_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTHPS_CVAL**

This register is accessed using the encoding for CNTP_CVAL.

Accesses to this register use the following encodings in the System register encoding space:
MRRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <Rt2>, <CRm>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1110</td>
<td>0b0010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !ELUsingAArch32(EL1) && !EL2Enabled() && HCR_EL2.<E2H,TGE> == '1' && CNTKCTL_EL1.EL0PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x04);
  elsif EL2Enabled() && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  else
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  else
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' && IsFeatureImplemented(�ETEL2) then
    if (R[t2], R[t]) = (CNTP_CVAL<63:32>, CNTP_CVAL<31:0>) then
      if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        (R[t2], R[t]) = (CNTP_CVAL_NS<63:32>, CNTP_CVAL_NS<31:0>)
      else
        (R[t2], R[t]) = (CNTP_CVAL<63:32>, CNTP_CVAL<31:0>)
    elsif PSTATE.EL == EL1 then
      if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
      else
        AArch32.TakeHypTrapException(�0x0�);
      elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
      else
        UNDEFINED;
      elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' && IsFeature Implemented(�ETEL2) then
        if (R[t2], R[t]) = (CNTP_CVAL<63:32>, CNTP_CVAL<31:0>) then
          if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
            (R[t2], R[t]) = (CNTP_CVAL_NS<63:32>, CNTP_CVAL_NS<31:0>)
          else
            (R[t2], R[t]) = (CNTP_CVAL<63:32>, CNTP_CVAL<31:0>)
        elsif PSTATE.EL == EL2 then
          if EL2Enabled() && !ELUsingAArch32(EL2) then
            (R[t2], R[t]) = (CNTP_CVAL<63:32>, CNTP_CVAL<31:0>)
          else
            (R[t2], R[t]) = (CNTP_CVAL<63:32>, CNTP_CVAL<31:0>)
          elsif EL2Enabled() && !ELUsingAArch32(EL3) then
            (R[t2], R[t]) = (CNTP_CVAL<63:32>, CNTP_CVAL<31:0>)
          elsif PSTATE.EL == EL3 then
            if SCR.NS == '0' then
              (R[t2], R[t]) = (CNTP_CVAL_S<63:32>, CNTP_CVAL_S<31:0>)
            else
              (R[t2], R[t]) = (CNTP_CVAL_NS<63:32>, CNTP_CVAL_NS<31:0>)
          else
            (R[t2], R[t]) = (CNTP_CVAL_C<63:32>, CNTP_CVAL_C<31:0>)
        else
          (R[t2], R[t]) = (CNTP_CVAL<63:32>, CNTP_CVAL<31:0>)
      else
        (R[t2], R[t]) = (CNTP_CVAL<63:32>, CNTP_CVAL<31:0>)
MCRR{<c>}{<q>}<coproc>,(#)<opc1>,<Rt>,<Rt2>,<CRm>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1110</td>
<td>0b0010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if !ELUsingAArch32(EL1) && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTKCTL_EL1.EL0PTEN == '0' then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
else
AArch64.AArch32SystemAccessTrap(EL1, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL1) && HCR_EL2.EL0PTEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTTHCTL_EL2.EL1PCEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FeaT_SEL2) then
CNPSP_CVAL_EL2 = R[t2]:R[t];
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
CNPSP_CVAL_EL2 = R[t2]:R[t];
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTTHCTL_EL2.EL1PTEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
AArch32.TakeHypTrapException(0x00);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
CNPSP_CVAL_EL2 = R[t2]:R[t];
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
CNPSP_CVAL_EL2 = R[t2]:R[t];
else
CNPSP_CVAL_EL2 = R[t2]:R[t];
elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
CNPSP_CVAL_NS = R[t2]:R[t];
else
CNPSP_CVAL_NS = R[t2]:R[t];
elsif PSTATE.EL == EL2 then
if EL2Enabled() && ELUsingAArch32(EL3) then
CNPSP_CVAL_NS = R[t2]:R[t];
else
CNPSP_CVAL_NS = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
if SCR.NS == '0' then
CNPSP_CVAL_S = R[t2]:R[t];
else
CNPSP_CVAL_S = R[t2]:R[t];
G8.7.8  **CNTHPS_TVAL, Counter-timer Secure Physical Timer TimerValue Register (EL2)**

The CNTHPS_TVAL characteristics are:

**Purpose**

Provides AArch32 access from EL0 to the timer value for the Secure EL2 physical timer.

**Configurations**

AArch32 System register CNTHPS_TVAL bits [31:0] are architecturally mapped to AArch64 System register CNTHPS_TVAL_EL2[31:0].

This register is present only when AArch32 is supported and FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHPS_TVAL are UNDEFINED.

**Attributes**

CNTHPS_TVAL is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimerValue</td>
<td></td>
</tr>
</tbody>
</table>

**TimerValue, bits [31:0]**

The TimerValue view of the EL2 physical timer.

On a read of this register:

- If CNTHPS_CTL_EL2.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHPS_CTL_EL2.ENABLE is 1, the value returned is (CNTHPS_CVVAL_EL2 - CNTPCT_EL0).

On a write of this register, CNTHPS_CVVAL_EL2 is set to (CNTPCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHPS_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CNTHPS_CVVAL_EL2) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- **CNTHPS_CTL_EL2.ISTATUS** is set to 1.
- **If CNTHPS_CTL_EL2.IMASK** is 0, an interrupt is generated.

When CNTHPS_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTHPS_TVAL**

This register is accessed using the encoding for CNTP_TVAL.

Accesses to this register use the following encodings in the System register encoding space:
MRC\{c\}\{q\} \{coproc\}, \{#\}\{opc1\}, <Rt>, <CRn>, <CRm>, \{#\}\{opc2\} 

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if EL2Enabled() && !EL2UsingAArch32(EL2) && HCR_EL2.E2H.TGE == '1' && CNTKCTL_EL1.EL0PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif EL2Enabled() && EL2UsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && EL2UsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTKCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        UNDEFINED;
    elsif EL2Enabled() && EL2UsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTKCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && EL2UsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTKCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        UNDEFINED;
    end;

if EL2Enabled() && EL2UsingAArch32(EL2) && HCR_EL2.TGE == '1' && CNTKCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
    UNDEFINED;
end;

if EL2Enabled() && EL2UsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTKCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
    UNDEFINED;
end;

if EL2Enabled() && EL2UsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTKCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
    UNDEFINED;
end;

if EL2Enabled() && EL2UsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTKCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
    UNDEFINED;
end;

if EL2Enabled() && EL2UsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTKCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
    UNDEFINED;
end;

if EL2Enabled() && EL2UsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTKCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
    UNDEFINED;
end;

if EL2Enabled() && EL2UsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTKCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
    UNDEFINED;
end;

if EL2Enabled() && EL2UsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTKCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
    UNDEFINED;
end;

if EL2Enabled() && EL2UsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTKCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
    UNDEFINED;
end;

if EL2Enabled() && EL2UsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTKCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
    UNDEFINED;
end;
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elseif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
  AArch32.TakeHypTrapException(0x03);
elseif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
  && CNTHCTL_EL2.ECV == '1' then
  if CNTP_CTL.ENABLE == '0' then
    R[t] = bits(32) UNKNOWN;
  else
    R[t] = (CNTP_CVAL - (PhysicalCountInt() - CNTPOFF_EL2))<31:0>;
  endif
endif

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0'
  then
    if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    endif
  elseif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
    if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch32.TakeHypTrapException(0x03);
    endif
  endif
else
  UNDEFINED;
endif
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
  AArch32.TakeHypTrapException(0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FEAT_SEL2) then
  CNTHPS_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
  CNTHP_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
&& CNTHCTL_EL2.ECV == '1' && HCR_EL2.<E2H,TGE> != '11' then
  CNTP_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTPOFF_EL2;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
  if SCR.NS == '1' then
    CNTP_CVAL_NS = SignExtend(R[t], 64) + PhysicalCountInt();
  else
    CNTP_CVAL_S = SignExtend(R[t], 64) + PhysicalCountInt();
  end
  CNTP_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FEAT_SEL2) then
  CNTHPS_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
  CNTHP_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
&& CNTHCTL_EL2.ECV == '1' then
  CNTP_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTPOFF_EL2;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
  if SCR.NS == '1' then
    CNTP_CVAL_NS = SignExtend(R[t], 64) + PhysicalCountInt();
  else
    CNTP_CVAL_S = SignExtend(R[t], 64) + PhysicalCountInt();
  end
  CNTP_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
elsif PSTATE.EL == EL2 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FEAT_SEL2) then
  CNTHPS_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
  CNTHP_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
&& CNTHCTL_EL2.ECV == '1' then
  CNTP_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTPOFF_EL2;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
  if SCR.NS == '1' then
    CNTP_CVAL_NS = SignExtend(R[t], 64) + PhysicalCountInt();
  else
    CNTP_CVAL_S = SignExtend(R[t], 64) + PhysicalCountInt();
  end
  CNTP_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
elsif PSTATE.EL == EL3 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
    AArch32.TakeHypTrapException(0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FEAT_SEL2) then
  CNTHPS_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
  CNTHP_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
&& CNTHCTL_EL2.ECV == '1' then
  CNTP_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTPOFF_EL2;
G8.7.9   **CNTHV_CTL, Counter-timer Virtual Timer Control register (EL2)**

The CNTHV_CTL characteristics are:

**Purpose**

Provides AArch32 access to the control register for the EL2 virtual timer.

**Configurations**

AArch32 System register CNTHV_CTL bits [31:0] are architecturally mapped to AArch64 System register CNTHV_CTL_EL2[31:0].

This register is present only when AArch32 is supported and FEAT_VHE is implemented. Otherwise, direct accesses to CNTHV_CTL are UNDEFINED.

**Attributes**

CNTHV_CTL is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td><strong>RES0</strong></td>
</tr>
<tr>
<td>3</td>
<td>ISTATUS, bit [2]</td>
</tr>
<tr>
<td>2</td>
<td>ENABLE, bit [0]</td>
</tr>
<tr>
<td>1</td>
<td>IMASK, bit [1]</td>
</tr>
</tbody>
</table>

**ISTATUS, bit [2]**

The status of the timer. This bit indicates whether the timer condition is met:
- 0b0: Timer condition is not met.
- 0b1: Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted. When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

**IMASK, bit [1]**

Timer interrupt mask bit. Permitted values are:
- 0b0: Timer interrupt is not masked by the IMASK bit.
- 0b1: Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ENABLE, bit [0]**

Enables the timer. Permitted values are:
- 0b0: Timer disabled.
- 0b1: Timer enabled.
Setting this bit to 0 disables the timer output signal, but the timer value accessible from
**CNTHV_TVAL** continues to count down.

--- Note ---
Disabling the output signal might be a power-saving option.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Accessing CNTHV_CTL**

This register is accessed using the encoding for **CNTV_CTL**.

Accesses to this register use the following encodings in the System register encoding space:

**MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}**

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0'
    then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'
        then
          AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
          AArch32.TakeHypTrapException(0x00);
        endif
      else
        UNDEFINED;
      endif
    else
      UNDEFINED;
    endif
  else
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0'
      then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
      else
        AArch64.AArch32SystemAccessTrap(EL1, 0x03);
      endif
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    endif
  endif
else
  if PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1VTEN == '1'
      then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
      else
        AArch64.AArch32SystemAccessTrap(EL1, 0x03);
      endif
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    endif
  else
    if PSTATE.EL == EL2 then
      R[t] = CNTV_CTL;
    else
      R[t] = CNTV_CTL;
    endif
  else
    if PSTATE.EL == EL3 then
      R[t] = CNTV_CTL;
    else
      R[t] = CNTV_CTL;
    endif
  endif
if PSTATE.EL == EL0 then
  if EL2Enabled() && (EL1Enabled() && HCR_EL2.<E2H,TGE> == '1') && CNTKCTL_EL1.EL0VTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  elsif EL1UsingAArch32(EL1) && CNTKCTL_EL1.EL0VTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
      AArch32.TakeHypTrapException(0x00);
    else
      UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
    CNTHV_CTL_EL2 = R[t];
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
    CNTHV_CTL_EL2 = R[t];
  else
    CNTV_CTL = R[t];
  elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1TVT == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      CNTV_CTL = R[t];
  elseif PSTATE.EL == EL2 then
    CNTV_CTL = R[t];
  elseif PSTATE.EL == EL3 then
    CNTV_CTL = R[t];
  else
    UNDEFINED;
G8.7.10 CNTHV_CVAL, Counter-timer Virtual Timer CompareValue register (EL2)

The CNTHV_CVAL characteristics are:

**Purpose**

Provides AArch32 access to the compare value for the EL2 virtual timer.

**Configurations**

AArch32 System register CNTHV_CVAL bits [63:0] are architecturally mapped to AArch64 System register CNTHV_CVAL_EL2[63:0]. This register is present only when AArch32 is supported and FEAT_VHE is implemented. Otherwise, direct accesses to CNTHV_CVAL are UNDEFINED.

**Attributes**

CNTHV_CVAL is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CompareValue, bits [63:0]**

Holds the EL2 virtual timer CompareValue.

When CNTHV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer.

When the timer condition is met:
- CNTHV_CTL.ISTATUS is set to 1.
- If CNTHV_CTL.IMASK is 0, an interrupt is generated.

When CNTHV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

**Accessing CNTHV_CVAL**

Accesses to this register use the following encodings in the System register encoding space:

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

```
<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b110</td>
<td>0b0011</td>
</tr>
</tbody>
</table>
```

if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '1') &&CNTKCTL_EL1.EL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    else
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
```

ARM DDI 0487I.a

Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.

ID081822

Non-Confidential
```c
逻辑代码

```
elsif PSTATE_EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1TVT == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        CNTV_CVAL = R[t2]:R[t];
    elsif PSTATE_EL == EL2 then
        CNTV_CVAL = R[t2]:R[t];
    elsif PSTATE_EL == EL3 then
        CNTV_CVAL = R[t2]:R[t];
G8.7.11  CNTHV_TVAL, Counter-timer Virtual Timer TimerValue register (EL2)

The CNTHV_TVAL characteristics are:

**Purpose**

Provides AArch32 access to the timer value for the EL2 virtual timer.

**Configurations**

AArch32 System register CNTHV_TVAL bits [31:0] are architecturally mapped to AArch64 System register CNTHV_TVAL_EL2[31:0].

This register is present only when AArch32 is supported and FEAT_VHE is implemented. Otherwise, direct accesses to CNTHV_TVAL are UNDEFINED.

**Attributes**

CNTHV_TVAL is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>TimerValue, bits [31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>The TimerValue view of the EL2 virtual timer.</td>
</tr>
<tr>
<td>On a read of this register:</td>
</tr>
<tr>
<td>• If CNTHV_CTL.ENABLE is 0, the value returned is UNKNOWN.</td>
</tr>
<tr>
<td>• If CNTHV_CTL.ENABLE is 1, the value returned is (CNTHV_CVAL - CNTVCT).</td>
</tr>
<tr>
<td>On a write of this register, CNTHV_CVAL is set to (CNTVCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.</td>
</tr>
<tr>
<td>When CNTHV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CNTHV_CVAL) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer.</td>
</tr>
<tr>
<td>When the timer condition is met:</td>
</tr>
<tr>
<td>• CNTHV_CTL.ISTATUS is set to 1.</td>
</tr>
<tr>
<td>• If CNTHV_CTL.IMASK is 0, an interrupt is generated.</td>
</tr>
<tr>
<td>When CNTHV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count, so the TimerValue view appears to continue to count down.</td>
</tr>
</tbody>
</table>

**Accessing CNTHV_TVAL**

This register is accessed using the encoding for CNTV_TVAL.

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC\{<c>\}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if !ELUsingAArch32(EL1) && ((EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0') then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif EL2Enabled() \&\& CNTKCTL.PLOVTEN == '0' then
if EL2Enabled() \&\& !ELUsingAArch32(EL2) \&\& HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() \&\& ELUsingAArch32(EL2) \&\& HCR.TGE == '1' then
AArch32.TakeHypTrapException(0x00);
else
UNDEFINED;
elsif EL2Enabled() \&\& !ELUsingAArch32(EL2) \&\& HCR_EL2.<E2H,TGE> == '11' \&\& CNTHCTL_EL2.EL0VTEN == '0'
 then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() \&\& !ELUsingAArch32(EL2) \&\& HCR_EL2.<E2H,TGE> == '11' \&\& CNTHCTL_EL2.EL1TVT == '1'
 then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() \&\& !ELUsingAArch32(EL2) \&\& HCR_EL2.<E2H,TGE> == '11' \&\& SCR_EL3.NS == '0' \&\&
IsFeatureImplemented(FEAT_SEL2)
then
if CNTHVS_CTL_EL2.ENABLE == '0'
then
R[t] = bits(32) UNKNOWN;
else
R[t] = (CNTHVS_CVAL_EL2 - PhysicalCountInt())<31:0>
;endif
elsif EL2Enabled() \&\& !ELUsingAArch32(EL2) \&\& HCR_EL2.<E2H,TGE> == '11' \&\& SCR_EL3.NS == '1'
then
if CNTHV_CTL_EL2.ENABLE == '0'
then
R[t] = bits(32) UNKNOWN;
else
R[t] = (CNTHV_CVAL_EL2 - PhysicalCountInt())<31:0>
;endif
else
if CNTV_CTL.ENABLE == '0'
then
R[t] = bits(32) UNKNOWN;
elsif HaveEL(EL2) \&\& !ELUsingAArch32(EL2) then
R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF_EL2))<31:0>
;endif
elsif HaveEL(EL2) \&\& ELUsingAArch32(EL2) then
R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF))<31:0>
;endif
else
R[t] = (CNTV_CVAL - PhysicalCountInt())<31:0>
;endif
elsif PSTATE.EL == EL1 then
if EL2Enabled() \&\& !ELUsingAArch32(EL2) \&\& CNTHCTL_EL2.EL1TVT == '1'
 then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
if CNTV_CTL.ENABLE == '0'
then
R[t] = bits(32) UNKNOWN;
elsif HaveEL(EL2) \&\& !ELUsingAArch32(EL2) then
R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF_EL2))<31:0>
;endif
elsif HaveEL(EL2) \&\& ELUsingAArch32(EL2) then
R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF))<31:0>
;endif
else
R[t] = (CNTV_CVAL - PhysicalCountInt())<31:0>
;endif
elsif PSTATE.EL == EL2 then
if CNTV_CTL.ENABLE == '0'
then
R[t] = bits(32) UNKNOWN;
else
R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF))<31:0>
;endif
elsif PSTATE.EL == EL3 then
if CNTV_CTL.ENABLE == '0'
then
R[t] = bits(32) UNKNOWN;
elsif HaveEL(EL2) then
R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF))<31:0>
;endif
else
R[t] = (CNTV_CVAL - PhysicalCountInt())<31:0>
;endif
MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EI == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '1') && CNTKCTL_EL1.EL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
        endif
    else
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    endif
    elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
            AArch32.TakeHypTrapException(0x00);
        else
            UNDEFINED;
        endif
    else
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL2.EL0VTEN == '0' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && CNTKTL_EL2.EL1VT == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
            CNTHVS_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
            CNTHV_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
        else
            if HaveEL(EL2) && !ELUsingAArch32(EL2) then
                CNTV_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTVOFF_EL2;
            elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
                CNTV_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTVOFF;
            else
                CNTV_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
            endif
        endif
    else
        if PSTATE.EI == EL1 then
            if EL2Enabled() && !ELUsingAArch32(EL2) && CNTKCTL_EL2.EL1VT == '1' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x03);
            else
                if HaveEL(EL2) && !ELUsingAArch32(EL2) then
                    CNTV_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTVOFF_EL2;
                elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
                    CNTV_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTVOFF;
                else
                    CNTV_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
                endif
            endif
        else
            if PSTATE.EI == EL3 then
                if HaveEL(EL2) then
                    CNTV_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTVOFF;
                else
                    CNTV_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
                endif
            else
                CNTV_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
            endif
        endif
    endif
endif
G8.7.12 CNTHVS_CTL, Counter-timer Secure Virtual Timer Control Register (EL2)

The CNTHVS_CTL characteristics are:

**Purpose**

Provides AArch32 access from EL0 to the Secure EL2 virtual timer.

**Configurations**

AArch32 System register CNTHVS_CTL[31:0] are architecturally mapped to AArch64 System register CNTHVS_CTL_EL2[31:0].

This register is present only when AArch32 is supported and FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHVS_CTL are UNDEFINED.

**Attributes**

CNTHVS_CTL is a 32-bit register.

**Field descriptions**

![Field Descriptions Diagram]

**Bits [31:3]**

Reserved, RES0.

**ISTATUS, bit [2]**

The status of the timer. This bit indicates whether the timer condition is met:

- 0b0 Timer condition is not met.
- 0b1 Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

**IMASK, bit [1]**

Timer interrupt mask bit. Permitted values are:

- 0b0 Timer interrupt is not masked by the IMASK bit.
- 0b1 Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**ENABLE, bit [0]**

Enables the timer. Permitted values are:

- 0b0 Timer disabled.
- 0b1 Timer enabled.
Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHVS_TV continues to count down.

---

**Note**

Disabling the output signal might be a power-saving option.

---

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

### Accessing CNTHVS_CTL

This register is accessed using the encoding for CNTV_CTL.

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

```c
if PSTATE_EL == EL0 then
  if !EL2Enabled() && HCR_EL2.<E2H,TGE> == '1' && CNTHCTL_EL0.EL0VTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.EL2.TGE == '1' then
    AArch64.TakeHypTrapException(0x00);
  else
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
    R[t] = CNTHVS_CTL_EL2<31:0>;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
    R[t] = CNTHV_CTL_EL2<31:0>;
  else
    R[t] = CNTV_CTL;
  elsif PSTATE_EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1TVT == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      R[t] = CNTV_CTL;
  elsif PSTATE_EL == EL2 then
    R[t] = CNTV_CTL;
  elsif PSTATE_EL == EL3 then
    R[t] = CNTV_CTL;
```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  elsif EL2Enabled() && CNTKCTL.PL0VTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
      AArch32.TakeHypTrapException(0x00);
    else
      UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
    IsFeatureImplemented(FEAT_SEL2) then
    CNTHV_CTL_EL2 = R[t];
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
    CNTHV_CTL_EL2 = R[t];
  else
    CNTV_CTL = R[t];
  elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1VTEN == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      CNTV_CTL = R[t];
  elsif PSTATE.EL == EL2 then
    CNTV_CTL = R[t];
  elsif PSTATE.EL == EL3 then
    CNTV_CTL = R[t];
G8.7.13  CNTHVS_CVAL, Counter-timer Secure Virtual Timer CompareValue Register (EL2)

The CNTHVS_CVAL characteristics are:

**Purpose**

Provides AArch32 access to the compare value for the Secure EL2 virtual timer.

**Configurations**

AArch32 System register CNTHVS_CVAL bits [63:0] are architecturally mapped to AArch64 System register CNTHVS_CVAL_EL2[63:0].

This register is present only when AArch32 is supported and FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHVS_CVAL are UNDEFINED.

**Attributes**

CNTHVS_CVAL is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field Description</th>
<th>Bit Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompareValue</td>
<td>63:0</td>
</tr>
</tbody>
</table>

**CompareValue, bits [63:0]**

Holds the EL2 virtual timer CompareValue.

When CNTHVS_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer.

When the timer condition is met:

- CNTHVS_CTL.ISTATUS is set to 1.
- If CNTHVS_CTL.IMASK is 0, an interrupt is generated.

When CNTHVS_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

**Accessing CNTHVS_CVAL**

This register is accessed using the encoding for CNTV_CVAL.

Accesses to this register use the following encodings in the System register encoding space:

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1110</td>
<td>0b0011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.TGE == '11') && CNTKCTL_EL1.EL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then

G8-10186  Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential
## AArch32 System Register Descriptions

### G8.7 Generic Timer registers

```cpp
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
```

```cpp
else
    AArch64.AArch32SystemAccessTrap(EL1, 0x04);
```

```cpp
elsif EL2Enabled() && CNTKCTL.PL0VTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
        AArch32.TakeHypTrapException(0x00);
    else
        UNDEFINED;
    end
```

```cpp
elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FEAT_SEL2) then
    (R[t2], R[t]) = (CNTHVS_CVAL_EL2<63:32>, CNTHVS_CVAL_EL2<31:0>);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
    (R[t2], R[t]) = (CNTHVS_CVAL_EL2<63:32>, CNTHVS_CVAL_EL2<31:0>);
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL1.EL0VTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL1.EL0VTEN == '0' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
        end
    end
```

```cpp
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
    AArch32.TakeHypTrapException(0x00);
elsif PSTATE.EL == EL2 then
    (R[t2], R[t]) = (CNTV_CVAL<63:32>, CNTV_CVAL<31:0>);
elsif PSTATE.EL == EL3 then
    (R[t2], R[t]) = (CNTV_CVAL<63:32>, CNTV_CVAL<31:0>);
```

### MCRR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <Rt2>, <CRm>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1110</td>
<td>0b0011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTKCTL_EL1.EL0VTEN == '0' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
        end
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTKCTL.PL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FEAT_SEL2) then
            CNTHVS_CVAL_EL2 = R[t2]:R[t];
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
            CNTHVS_CVAL_EL2 = R[t2]:R[t];
```
else
    CNTV_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1TVT == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        CNTV_CVAL = R[t2]:R[t];
    end if
elsif PSTATE.EL == EL2 then
    CNTV_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    CNTV_CVAL = R[t2]:R[t];
G8.7.14   CNTHVS_TVAL, Counter-timer Secure Virtual Timer TimerValue Register (EL2)

The CNTHVS_TVAL characteristics are:

**Purpose**

Provides AArch32 access to the timer value for the Secure EL2 virtual timer.

**Configurations**

AArch32 System register CNTHVS_TVAL bits [31:0] are architecturally mapped to AArch64 System register CNTHVS_TVAL_EL2[31:0].

This register is present only when AArch32 is supported and FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHVS_TVAL are UNDEFINED.

**Attributes**

CNTHVS_TVAL is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>TimerValue, bits [31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>The TimerValue view of the EL2 virtual timer.</td>
</tr>
<tr>
<td>On a read of this register:</td>
</tr>
<tr>
<td>• If CNTHVS_CTL.ENABLE is 0, the value returned is UNKNOWN.</td>
</tr>
<tr>
<td>• If CNTHVS_CTL.ENABLE is 1, the value returned is (CNTHVS_CV AL - CNTVCT).</td>
</tr>
<tr>
<td>On a write of this register, CNTHVS_CV AL is set to (CNTVCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.</td>
</tr>
<tr>
<td>When CNTHVS_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CNTHVS_CV AL) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:</td>
</tr>
<tr>
<td>• CNTHVS_CTL.ISTATUS is set to 1.</td>
</tr>
<tr>
<td>• If CNTHVS_CTL.IMASK is 0, an interrupt is generated.</td>
</tr>
<tr>
<td>When CNTHVS_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count, so the TimerValue view appears to continue to count down.</td>
</tr>
</tbody>
</table>

**Accessing CNTHVS_TVAL**

This register is accessed using the encoding for CNTV_TVAL.

Accesses to this register use the following encodings in the System register encoding space:

\[ MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} \]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && (EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
  AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
  AArch32.TakeHypTrapException(0x00);
else
  UNDEFINED;
endif;
elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '1' && CNTHCTL_EL2.EL1TVT == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
  IsFeatureImplemented(FEATURE_EL2) then
  if CNTHSV_CTL_EL2.ENABLE == '0' then
    R[t] = bits(32) UNKNOWN;
  else
    R[t] = (CNTHSV_CVAL_EL2 - PhysicalCountInt())<31:0>;
  endif;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
  if CNTHV_CTL_EL2.ENABLE == '0' then
    R[t] = bits(32) UNKNOWN;
  else
    R[t] = (CNTHV_CVAL_EL2 - PhysicalCountInt())<31:0>;
  endif;
elseif HaveEL(EL2) && !ELUsingAArch32(EL2) then
  if CNTV_CTL.ENABLE == '0' then
    R[t] = bits(32) UNKNOWN;
  elsif HaveEL(EL3) && ELUsingAArch32(EL2) then
    R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF_EL2))<31:0>;
  else
    R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF))<31:0>;
  endif;
else
  if PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1TVT == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      if CNTV_CTL.ENABLE == '0' then
        R[t] = bits(32) UNKNOWN;
      elsif HaveEL(EL2) && !ELUsingAArch32(EL2) then
        R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF_EL2))<31:0>;
      else
        R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF))<31:0>;
      endif;
    endif;
  else
    if PSTATE.EL == EL2 then
      if CNTV_CTL.ENABLE == '0' then
        R[t] = bits(32) UNKNOWN;
      else
        R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF))<31:0>;
      endif;
    elseif PSTATE.EL == EL3 then
      if CNTV_CTL.ENABLE == '0' then
        R[t] = bits(32) UNKNOWN;
      elseif HaveEL(EL2) then
        R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF))<31:0>;
      else
        R[t] = (CNTV_CVAL - (PhysicalCountInt()))<31:0>;
      endif;
    fi;
  fi;
fi;
MCR{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>)}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '1') && CNTKCTL_EL1.EL0VTEN == '0' then
    if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  else
    ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
      if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
      else
        UNDEFINED;
    endif
  endif
else
  ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '1' && CNTHCTL_EL2.EL0VTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '1' && CNTHCTL_EL2.EL1VT == '1' then
      if HaveEL(EL2) && ELUsingAArch32(EL2) then
        CNTV_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTVOFF_EL2;
      else
        CNTV_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
      endif
    else
      PSTATE.EL == EL1 then
        if EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1VT == '1' then
          AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
          if HaveEL(EL2) && ELUsingAArch32(EL2) then
            CNTV_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTVOFF_EL2;
          else
            CNTV_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
          endif
        endif
      else
        PSTATE.EL == EL2 then
          CNTV_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTVOFF;
        else
          PSTATE.EL == EL3 then
            if HaveEL(EL2) then
              CNTV_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTVOFF;
            else
              CNTV_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
            endif
        endif
      endif
    else
      EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch32.TakeHypTrapException(0x00);
      else
        EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
          AArch32.SystemAccessTrap(EL2, 0x03);
        else
          UNDEFINED;
        endif
      else
        UNDEFINED;
      endif
    endif
  endif
endif
### G8.7.15 CNTKCTL, Counter-timer Kernel Control register

The CNTKCTL characteristics are:

#### Purpose

Controls the generation of an event stream from the virtual counter, and access from EL0 modes to the physical counter, virtual counter, EL1 physical timers, and the virtual timer.

#### Configurations

AArch32 System register CNTKCTL bits [31:0] are architecturally mapped to AArch64 System register CNTKCTL_EL1[31:0].

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to CNTKCTL are UNDEFINED.

#### Attributes

CNTKCTL is a 32-bit register.

#### Field descriptions

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field Description</th>
<th>Value 1 (0b0)</th>
<th>Value 2 (0b1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>EVNTIS, bit [17]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16:10</td>
<td>Reserved, RES0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>PL0PTEN, bit [9]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>EVNTDIR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>EVNTEN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Bits [31:18]

Reserved, RES0.

#### EVNTIS, bit [17]

*When FEAT_ECV is implemented:*

Controls the scale of the generation of the event stream.

0b0 The CNTKCTL.EVNTI field applies to CNTVCT[15:0].

0b1 The CNTKCTL.EVNTI field applies to CNTVCT[23:8].

This control applies regardless of the value of the CNTHCTL_EL2.ECV bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

#### Bits [16:10]

Reserved, RES0.

#### PL0PTEN, bit [9]

Traps PL0 accesses to the physical timer registers to Undefined mode.

0b0 PL0 accesses to the CNTP_CTL, CNTP_CVAL, and CNTP_TVAL registers are trapped to Undefined mode.

0b1 This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.
**PL0VTEN, bit [8]**

Traps PL0 accesses to the virtual timer registers to Undefined mode.

- **0b0**: PL0 accesses to the CNTV_CTL, CNTV_CVAL, and CNTV_TVAL registers are trapped to Undefined mode.
- **0b1**: This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**EVNTI, bits [7:4]**

Selects which bit of CNTVCT, as seen from EL1, is the trigger for the event stream generated from that counter when that stream is enabled.

If FEAT_ECV is implemented, and CNTKCTL.EVNTIS is 1, this field selects a trigger bit in the range 8 to 23 of CNTVCT.

Otherwise, this field selects a trigger bit in the range 0 to 15 of CNTVCT.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**EVNTDIR, bit [3]**

Controls which transition of the CNTVCT trigger bit, as seen from EL1 and defined by EVNTI, generates an event when the event stream is enabled.

- **0b0**: A 0 to 1 transition of the trigger bit triggers an event.
- **0b1**: A 1 to 0 transition of the trigger bit triggers an event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**EVNTEN, bit [2]**

Enables the generation of an event stream from CNTVCT as seen from EL1.

- **0b0**: Disables the event stream.
- **0b1**: Enables the event stream.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**PL0VCTEN, bit [1]**

Traps PL0 accesses to the frequency register and virtual counter register to Undefined mode.

- **0b0**: PL0 accesses to the CNTVCT are trapped to Undefined mode.
- **0b1**: PL0 accesses to the CNTFRQ register are trapped to Undefined mode, if CNTKCTL.PL0PCTEN is also 0.
- **0b1**: This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**PL0PCTEN, bit [0]**

Traps PL0 accesses to the frequency register and physical counter register to Undefined mode.

- **0b0**: PL0 accesses to the CNTPCT are trapped to Undefined mode.
- **0b1**: PL0 accesses to the CNTFRQ register are trapped to Undefined mode, if CNTKCTL.PL0VCTEN is also 0.
- **0b1**: This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
Accessing CNTKCTL

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} <\text{coproc}>, \{#<\text{opc}1>, <\text{R}t>, <\text{CRn}>, <\text{CRm}>\}, \{#<\text{opc}2>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>

\[
\text{MCR}\{<c>\}{<q>} <\text{coproc}>, \{#<\text{opc}1>, <\text{R}t>, <\text{CRn}>, <\text{CRm}>\}, \{#<\text{opc}2>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0001</td>
<td>0b000</td>
</tr>
</tbody>
</table>
G8.7.16 CNTP_CTL, Counter-timer Physical Timer Control register

The CNTP_CTL characteristics are:

**Purpose**

Control register for the EL1 physical timer.

**Configurations**

AArch32 System register CNTP_CTL bits [31:0] are architecturally mapped to AArch64 System register CNTP_CTL_EL0[31:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to CNTP_CTL are UNDEFINED.

**Attributes**

CNTP_CTL is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:3]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>ISTATUS, bit [2]</td>
<td>The status of the timer. This bit indicates whether the timer condition is met:</td>
</tr>
<tr>
<td>0b0</td>
<td>Timer condition is not met.</td>
</tr>
<tr>
<td>0b1</td>
<td>Timer condition is met.</td>
</tr>
<tr>
<td>IMASK, bit [1]</td>
<td>Timer interrupt mask bit. Permitted values are:</td>
</tr>
<tr>
<td>0b0</td>
<td>Timer interrupt is not masked by the IMASK bit.</td>
</tr>
<tr>
<td>0b1</td>
<td>Timer interrupt is masked by the IMASK bit.</td>
</tr>
<tr>
<td>ENABLE, bit [0]</td>
<td>Enables the timer. Permitted values are:</td>
</tr>
<tr>
<td>0b0</td>
<td>Timer disabled.</td>
</tr>
<tr>
<td>0b1</td>
<td>Timer enabled.</td>
</tr>
</tbody>
</table>
Setting this bit to 0 disables the timer output signal, but the timer value accessible from \textbf{CNTP-TVAL} continues to count down.

\textbf{Note}
Disabling the output signal might be a power-saving option.

The reset behavior of this field is:
- On a Warm reset, this field resets to $0$.

\textbf{Accessing CNTP_CTL}

Accesses to this register use the following encodings in the System register encoding space:

\begin{align*}
\text{MRC} \{<c>\} \{<q>\} & \quad <\text{coproc}>, \quad \{#\}<\text{opc1}>, \quad <\text{Rt}>, \quad <\text{CRn}>, \quad <\text{CRm}>, \quad \{#\}<\text{opc2}> \\
\begin{array}{cccccc}
\text{coproc} & \text{opc1} & \text{CRn} & \text{CRm} & \text{opc2} \\
0b1111 & 0b000 & 0b1110 & 0b0010 & 0b001
\end{array}
\end{align*}
if HaveEL(EL3) && ELUsingAArch32(EL3) then
    R[t] = CNTP_CTL_NS;
else
    R[t] = CNTP_CTL;
elif PSTATE.EL == EL3 then
    if SCR.NS == '0' then
        R[t] = CNTP_CTL_S;
    else
        R[t] = CNTP_CTL_NS;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>b1110</td>
<td>b0010</td>
<td>b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if EL1UsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0' then
        if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.E2H == '0' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
        endif
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
        AArch32.TakeHypTrapException(0x00);
    else
        UNDEFINED;
    endif
elsif EL2Enabled() && ELUsingAArch32(EL2) && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
    CNTHPS_CTL_EL2 = R[t];
elifs EL2Enabled() && ELUsingAArch32(EL2) && SCR_EL3.NS == '1' then
    CNTHP_CTL_EL2 = R[t];
else
    CNTP_CTL = R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.E2H == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.E2H == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elseif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
    IsFeatureImplemented(FEAT_SEL2) then
        CNTHPS_CTL_EL2 = R[t];
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        CNTHP_CTL_EL2 = R[t];
    else
        CNTP_CTL = R[t];
elsif PSTATE.EL == EL2 then
    if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elseif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
        AArch32.TakeHypTrapException(0x03);
    elseif HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_CTL_NS = R[t];
    else
        CNTP_CTL = R[t];
elsif PSTATE.EL == EL3 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        CNTP_CTL_NS = R[t];
    else
        CNTP_CTL = R[t];
if SCR_NS == '0' then
    CNTP_CTL_S = R[t];
else
    CNTP_CTL_NS = R[t];
G8.7.17  CNTP_CVAL, Counter-timer Physical Timer CompareValue register

The CNTP_CVAL characteristics are:

Purpose

Holds the compare value for the EL1 physical timer.

Configurations

AArch32 System register CNTP_CVAL bits [63:0] are architecturally mapped to AArch64 System register CNTP_CVAL_EL0[63:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to CNTP_CVAL are UNDEFINED.

Attributes

CNTP_CVAL is a 64-bit register.

Field descriptions

CompareValue, bits [63:0]

Holds the EL1 physical timer CompareValue.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer.

When the timer condition is met:

• CNTP_CTL.ISTATUS is set to 1.
• If CNTP_CTL.IMASK is 0, an interrupt is generated.

When CNTP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_CVAL

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRRC\{<c>\}{<q>\} <coproc>, \{#\}<opc1>, <Rt>, <Rt2>, <CRm>}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0b1110</td>
<td>0b0010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
else
AArch64.AArch32SystemAccessTrap(EL1, 0x04);
elsif EL2Enabled() && CNTKCTL.PL0PTEN == '0' then
if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch32.TakeHypTrapException(0x00);
else
UNDEFINED;
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTKCTL_EL2.EL1PCEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '0' && CNTKCTL_EL2.EL1PTEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FeaT_SEL2) then
(R[t2], R[t]) = (CNTHPS_CVAL_EL2<63:32>, CNTHPS_CVAL_EL2<31:0>);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' && SCR_EL3.NS == '1' then
(R[t2], R[t]) = (CNTHP_CVAL_EL2<63:32>, CNTHP_CVAL_EL2<31:0>);
else
(R[t2], R[t]) = (CNTP_CVAL<63:32>, CNTP_CVAL<31:0>);
elsif PSTATE.EL == EL1 then
if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTKCTL_EL2.EL1PCEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTKCTL_EL2.EL1PTEN == '0' then
AArch32.TakeHypTrapException(0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTKCTL.PL1PCEN == '0' then
AArch32.TakeHypTrapException(0x04);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
(R[t2], R[t]) = (CNTP_CVAL_NS<63:32>, CNTP_CVAL_NS<31:0>);
else
(R[t2], R[t]) = (CNTP_CVAL<63:32>, CNTP_CVAL<31:0>);
elsif PSTATE.EL == EL2 then
if HaveEL(EL3) && ELUsingAArch32(EL3) then
(R[t2], R[t]) = (CNTP_CVAL_NS<63:32>, CNTP_CVAL_NS<31:0>);
else
(R[t2], R[t]) = (CNTP_CVAL<63:32>, CNTP_CVAL<31:0>);
elsif PSTATE.EL == EL3 then
if SCR.NS == '0' then
(R[t2], R[t]) = (CNTP_CVAL_S<63:32>, CNTP_CVAL_S<31:0>);
else
(R[t2], R[t]) = (CNTP_CVAL_NS<63:32>, CNTP_CVAL_NS<31:0>);

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1110</td>
<td>0b0010</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if ELUsingAArch32(EL1) && !(EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1') && CNTKCTL_EL1.EL0PTEN == '0' then
if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
else
AArch64.AArch32SystemAccessTrap(EL1, 0x04);
elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PTEN == '0' then
if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then

AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
  AArch32.TakeHypTrapException(0x00);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PTEN == '0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
  AArch32.TakeHypTrapException(0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
  IsFeatureImplemented(FEAT_SEL2) then
  CNTHPS_CVAL_EL2 = R[t2]:R[t];
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
  CNTHP_CVAL_EL2 = R[t2]:R[t];
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.EL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
  CNTP_CVAL_NS = R[t2]:R[t];
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    CNTP_CVAL_NS = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    CNTP_CVAL_S = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    CNTP_CVAL_NS = R[t2]:R[t];
G8.7.18 **CNTP_TVAL, Counter-timer Physical Timer TimerValue register**

The CNTP_TVAL characteristics are:

**Purpose**
Holds the timer value for the EL1 physical timer.

**Configurations**
AArch32 System register CNTP_TVAL bits [31:0] are architecturally mapped to AArch64 System register CNTP_TVAL_EL0[31:0].
This register is present only when AArch32 is supported. Otherwise, direct accesses to CNTP_TVAL are UNDEFINED.

**Attributes**
CNTP_TVAL is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimerValue</td>
<td></td>
</tr>
</tbody>
</table>

**TimerValue, bits [31:0]**

The TimerValue view of the EL1 physical timer.
On a read of this register:
- If CNTP_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTP_CTL.ENABLE is 1, the value returned is (CNTP_CVAL - CNTPCT).
On a write of this register, CNTP_CVAL is set to (CNTPCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.
When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CNTP_CVAL) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer.
When the timer condition is met:
- CNTP_CTL.ISTATUS is set to 1.
- If CNTP_CTL.IMASK is 0, an interrupt is generated.
When CNTP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count, so the TimerValue view appears to continue to count down.
The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTP_TVAL**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRC}\{<c>\}{<q>} \coproc \{#<opc1>, <Rt>, <CRn>, <CRm>\{, (#)<opc2>\}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTPCTL_EL1.EL0PTEN == '0'
  then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif ELUsingAArch32(EL1) && CNTCTL.PL0PTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTCTL.PL1PTEN == '0' then
        AArch32.TakeHypTrapException(0x00);
    else
        UNDEFINED;
    endif
else
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTTHCTL_EL2.EL1PCEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1' && CNTHTCTL_EL2.ECV == '1' && CNTPOST_CTL.ENABLE == '0' then
        R[t] = (CNTP_CVAL - (PhysicalCountInt() - CNTPOFF_EL2))<31:0>;
    else
        if SCR.NS == '1' then
            if CNTP_CTL_NS.ENABLE == '0' then
                R[t] = bits(32) UNKNOWN;
            else
                R[t] = (CNTP_CVAL_NS - PhysicalCountInt())<31:0>;
            endif
        else
            if CNTP_CTL_S.ENABLE == '0' then
                R[t] = bits(32) UNKNOWN;
            else
                R[t] = (CNTP_CVAL_S - PhysicalCountInt())<31:0>;
            endif
        endif
    else
        if SCR_NS == '0' then
            if CNTP_CTL.ENABLE == '0' then
                R[t] = bits(32) UNKNOWN;
            else
                R[t] = (CNTP_CVAL - PhysicalCountInt())<31:0>;
            endif
        else
            if PSTATE.EL == EL1 then
                if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTTHCTL_EL2.EL1PCEN == '0' then
                    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
                elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTTHCTL_EL2.EL1PTEN == '0' then
                    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
                elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHTCTL.PL1PCEN == '0' then
                    AArch32.TakeHypTrapException(0x03);
                elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1' && CNTHTCTL_EL2.ECV == '1' && CNTHPS_CTL_EL2.ENABLE == '0' then
                    R[t] = (CNTHPS_CVAL_EL2 - PhysicalCountInt())<31:0>;
                else
                    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '11' && CNTHTCTL_EL2.EL0PTEN == '0' then
                        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
                    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHTCTL_EL2.EL1PTEN == '0' then
                        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
                    endif
                endif
            else
                if SCR_NS == '0' then
                    if CNTP_CTL_NS.ENABLE == '0' then
                        R[t] = bits(32) UNKNOWN;
                    else
                        R[t] = (CNTP_CVAL_NS - PhysicalCountInt())<31:0>;
                    endif
                else
                    if CNTP_CTL_S.ENABLE == '0' then
                        R[t] = bits(32) UNKNOWN;
                    else
                        R[t] = (CNTP_CVAL_S - PhysicalCountInt())<31:0>;
                    endif
                endif
            endif
        endif
    endif
endif
if CNTP_CTL_NS.ENABLE == '0' then
    R[t] = bits(32) UNKNOWN;
else
    if CNTP_CTL_ENABLE == '0' then
        R[t] = bits(32) UNKNOWN;
    else
        R[t] = (CNTP_CVAL_NS - PhysicalCountInt())<31:0>;
    endif
else
    if CNTP_CTL.ENABLE == '0' then
        R[t] = bits(32) UNKNOWN;
    else
        R[t] = (CNTP_CVAL - PhysicalCountInt())<31:0>;
    endif
endif

elseif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        if CNTP_CTL_NS.ENABLE == '0' then
            R[t] = bits(32) UNKNOWN;
        else
            R[t] = (CNTP_CVAL_NS - PhysicalCountInt())<31:0>;
        endif
    else
        if CNTP_CTL.ENABLE == '0' then
            R[t] = bits(32) UNKNOWN;
        else
            R[t] = (CNTP_CVAL - PhysicalCountInt())<31:0>;
        endif
    endif
endif

elseif PSTATE.EL == EL3 then
    if SCR_NS == '0' then
        if CNTP_CTL_S.ENABLE == '0' then
            R[t] = bits(32) UNKNOWN;
        else
            R[t] = (CNTP_CVAL_S - PhysicalCountInt())<31:0>;
        endif
    else
        if CNTP_CTL_NS.ENABLE == '0' then
            R[t] = bits(32) UNKNOWN;
        else
            R[t] = (CNTP_CVAL_NS - PhysicalCountInt())<31:0>;
        endif
    endif
endif

MCR<{c}<q} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0010</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    if ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0PTEN == '0' then
        if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
        endif
    else
        if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x03);
        endif
    endif
else
    if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL1PTEN == '0' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x03);
            else
                AArch64.AArch32SystemAccessTrap(EL1, 0x03);
            endif
        endif
    endif
else
    if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' && IsFeatureImplemented(FEAT_SEL2) then
                AArch32.TakeHypTrapException(0x00);
            else
                AArch64.AArch32SystemAccessTrap(EL2, 0x03);
            endif
        endif
    endif
else
    if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL1PTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
        if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PTEN == '0' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x03);
        else
            if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL1PTEN == '0' then
                AArch64.AArch32SystemAccessTrap(EL2, 0x03);
            else
                AArch64.AArch32SystemAccessTrap(EL1, 0x03);
            endif
        endif
    endif
endif
G8.7 Generic Timer registers

CNTHPS_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();

elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
  CNTHPS_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
  & CNTHCTL_EL2.ECV == '1' && HCR_EL2.<E2H,TGE> != '11' then
  CNTP_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTPOFF_EL2;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
  if SCR.NS == '1' then
    CNTP_CVAL_NS = SignExtend(R[t], 64) + PhysicalCountInt();
  else
    CNTP_CVAL_S = SignExtend(R[t], 64) + PhysicalCountInt();
  else
    CNTP_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.PL1PCEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CNTHCTL_EL2.PL1PTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCEN == '0' then
    AArch32.TakeHypTrapException(0x03);
elsif IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
  & CNTHCTL_EL2.ECV == '1' then
  CNTP_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTPOFF_EL2;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
  CNTP_CVAL_NS = SignExtend(R[t], 64) + PhysicalCountInt();
else
  CNTP_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
elsif PSTATE.EL == EL2 then
  if HaveEL(EL3) && ELUsingAArch32(EL3) then
    CNTP_CVAL_NS = SignExtend(R[t], 64) + PhysicalCountInt();
else
    CNTP_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    CNTP_CVAL_S = SignExtend(R[t], 64) + PhysicalCountInt();
else
    CNTP_CVAL_NS = SignExtend(R[t], 64) + PhysicalCountInt();
**G8.7.19 CNTPCT, Counter-timer Physical Count register**

The CNTPCT characteristics are:

**Purpose**

Holds the 64-bit physical count value.

**Configurations**

AArch32 System register CNTPCT bits [63:0] are architecturally mapped to AArch64 System register CNTPCT_EL0[63:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to CNTPCT are UNDEFINED.

All reads to the CNTPCT occur in program order relative to reads to CNTPCTSS or CNTPCT.

**Attributes**

CNTPCT is a 64-bit register.

**Field descriptions**

Bits [63:0]

Physical count value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTPCT**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRRC}(<c>){(q)} \ <\text{coproc}, \{#\}<opc1>, <Rt>, <Rt2>, <CRm>
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1110</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if ElUsingAArch32(EL1) && ((El2Enabled() && HCR_EL2.<E2H,TGE> == '1') && CNTKCTL_EL1.EL0PCTEN == '0' then
    if El2Enabled() && ElUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x04);
  elsif ElUsingAArch32(EL1) && CNTKCTL.PL0PCTEN == '0' then
    if El2Enabled() && !ElUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif El2Enabled() && ElUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif El2Enabled() && !ElUsingAArch32(EL2) && HCR_EL2.E2H == '0' then
      UNDEFINED;
    else
      UNDEFINED;
  elsif El2Enabled() && !ElUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCTEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL1PCTEN == '0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0PCTEN == '0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1PCTEN == '0' then
  AArch32.TakeHypTrapException(0x04);
else
  if IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1' && CNTHCTL_EL2.ECV == '1' && HCR_EL2.<E2H,TGE> != '11' then
    (R[t2], R[t]) = Split64to32(PhysicalCountInt() - CNTPOFF_EL2);
  else
    (R[t2], R[t]) = Split64to32(PhysicalCountInt());
  endif
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1PCTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL_EL2.PL1PCTEN == '0' then
    AArch32.TakeHypTrapException(0x04);
  else
    if IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1' && CNTHCTL_EL2.ECV == '1' then
      (R[t2], R[t]) = Split64to32(PhysicalCountInt() - CNTPOFF_EL2);
    else
      (R[t2], R[t]) = Split64to32(PhysicalCountInt());
    endif
  endif
elsif PSTATE.EL == EL2 then
  (R[t2], R[t]) = Split64to32(PhysicalCountInt());
elsif PSTATE.EL == EL3 then
  (R[t2], R[t]) = Split64to32(PhysicalCountInt());
G8.7.20 CNTPCTSS, Counter-timer Self-Synchronized Physical Count register

The CNTPCTSS characteristics are:

**Purpose**

Holds the 64-bit physical count value.

**Configurations**

AArch32 System register CNTPCTSS bits [63:0] are architecturally mapped to AArch64 System register CNTPCTSS_EL0[63:0].

This register is present only when AArch32 is supported and FEAT_ECV is implemented. Otherwise, direct accesses to CNTPCTSS are **UNDEFINED**.

All reads to the CNTPCTSS occur in program order relative to reads to CNTPCT or CNTPCTSS. This register is a self-synchronised view of the CNTPCT counter, and cannot be read speculatively.

**Attributes**

CNTPCTSS is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Self-Synchronized Physical count value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Self-Synchronized Physical count value</td>
</tr>
</tbody>
</table>

Bits [63:0]

Self-Synchronized Physical count value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**Accessing CNTPCTSS**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRRC\{<c>\}{<q>} \langle coproc \rangle, \langle\#\langle opc1 \rangle, \langle Rt \rangle, \langle Rt2 \rangle, \langle CRm \rangle
\]

<table>
<thead>
<tr>
<th>proc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1110</td>
<td>0b1000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '1') && CNTKCTL_EL1.EL0PCTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x04);
  elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PCTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
      AArch32.TakeHypTrapException(0x00);
    else
      AArch32.TrapException(0x00);
  else
    AArch32.TrapException(0x00);

if PSTATE.EL == EL1 then
  if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '1') && CNTKCTL_EL1.EL0PCTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x04);
  elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PCTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
      AArch32.TakeHypTrapException(0x00);
    else
      AArch32.TrapException(0x00);
  else
    AArch32.TrapException(0x00);

if PSTATE.EL == EL2 then
  if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '1') && CNTKCTL_EL1.EL0PCTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x04);
  elsif ELUsingAArch32(EL1) && CNTKCTL.PL0PCTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
      AArch32.TakeHypTrapException(0x00);
    else
      AArch32.TrapException(0x00);
  else
    AArch32.TrapException(0x00);
UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '0' && CNTHCTL_EL2.EL1PCTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '10' && CNTHCTL_EL2.EL0PCTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCTEN == '0' then
    AArch32.TakeHypTrapException(0x04);
else
    if IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
        && CNTHCTL_EL2.ECV == '1' && HCR_EL2.<E2H,TGE> != '11' then
        (R[t2], R[t]) = Split64to32(PhysicalCountInt() - CNTPOFF_EL2);
    else
        (R[t2], R[t]) = Split64to32(PhysicalCountInt());
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1PCTEN == '0' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        elsif EL2Enabled() && ELUsingAArch32(EL2) && CNTHCTL.PL1PCTEN == '0' then
            AArch32.TakeHypTrapException(0x04);
        else
            if IsFeatureImplemented(FEAT_ECV) && EL2Enabled() && !ELUsingAArch32(EL2) && SCR_EL3.ECVEn == '1'
                && CNTHCTL_EL2.ECV == '1' then
                (R[t2], R[t]) = Split64to32(PhysicalCountInt() - CNTPOFF_EL2);
            else
                (R[t2], R[t]) = Split64to32(PhysicalCountInt());
            end;
        end;
    elsif PSTATE.EL == EL2 then
        (R[t2], R[t]) = Split64to32(PhysicalCountInt());
    elsif PSTATE.EL == EL3 then
        (R[t2], R[t]) = Split64to32(PhysicalCountInt());
    end;
end;

G8.7.21 CNTV_CTL, Counter-timer Virtual Timer Control register

The CNTV_CTL characteristics are:

Purpose
Control register for the virtual timer.

Configurations
AArch32 System register CNTV_CTL bits [31:0] are architecturally mapped to AArch64 System register CNTV_CTL_EL0[31:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to CNTV_CTL are UNDEFINED.

Attributes
CNTV_CTL is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved</td>
<td>RES0</td>
</tr>
<tr>
<td>3</td>
<td>ISTATUS</td>
<td>The status of the timer. This bit indicates whether the timer condition is met:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: Timer condition is not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Timer condition is met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted. When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN. The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>2</td>
<td>IMASK</td>
<td>Timer interrupt mask bit. Permitted values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: Timer interrupt is not masked by the IMASK bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Timer interrupt is masked by the IMASK bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see the description of the ISTATUS bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
<tr>
<td>0</td>
<td>ENABLE</td>
<td>Enables the timer. Permitted values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: Timer disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Timer enabled.</td>
</tr>
</tbody>
</table>

Access to this field is RO.
Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTV_TV continues to count down.

--- Note ---
Disabling the output signal might be a power-saving option.

---

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

**Accessing CNTV_CTL**

Accesses to this register use the following encodings in the System register encoding space:

\[
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b001</td>
<td>0b001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '1') && CNTKCTL_EL1.EL0VTEN == '0' then
    if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    UNDEFINED;
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '1' && CNTKCTL_EL1.EL0VTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '1' && CNTKCTL_EL2.EL1VTEN == '0' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      UNDEFINED;
  else
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  end if;
else
  if PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CNTKCTL_EL2.EL1VTEN == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      R[t] = CNTV_CTL;
    end if;
  elseif PSTATE.EL == EL2 then
    R[t] = CNTV_CTL;
  elseif PSTATE.EL == EL3 then
    R[t] = CNTV_CTL;
  end if;
MCR{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <CRn>, <CRm>{, (#<opc2>)}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '1') && CNTHCTL_EL1.EL0VTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  endif
elsif ELUsingAArch32(EL1) && CNTHCTL.PL0VTEN == '0' then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  endif
else
  UNDEFINED;
endif
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '1' && CNTHCTL_EL2.EL0VTEN == '0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x03);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '1' && CNTHCTL_EL2.EL1VT == '1' then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '1' && SCR_EL3.NS == '0' &&
    IsFeatureImplemented(FEAT_SEL2) then
    CNTHV_CTL_EL2 = R[t];
  else
    CNTV_CTL = R[t];
  endif
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1VT == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    CNTV_CTL = R[t];
  endif
elsif PSTATE.EL == EL2 then
  CNTV_CTL = R[t];
elsif PSTATE.EL == EL3 then
  CNTV_CTL = R[t];
else
  UNDEFINED;
endif
CNTV_CVAL, Counter-timer Virtual Timer CompareValue register

The CNTV_CVAL characteristics are:

**Purpose**

Holds the compare value for the virtual timer.

**Configurations**

AArch32 System register CNTV_CVAL bits [63:0] are architecturally mapped to AArch64 System register CNTV_CVAL_EL0[63:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to CNTV_CVAL are UNDEFINED.

**Attributes**

CNTV_CVAL is a 64-bit register.

**Field descriptions**

![Field descriptions diagram]

**CompareValue, bits [63:0]**

Holds the EL1 virtual timer CompareValue.

When CNTV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer.

When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- If CNTV_CTL.IMASK is 0, an interrupt is generated.

When CNTV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTV_CVAL**

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRRC}\{<c>\}{<q>} <\text{coproc}>, \{#\}<\text{opc1}>, <\text{Rt}>, <\text{Rt2}>, <\text{CRm}>
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1110</td>
<td>0b0011</td>
</tr>
</tbody>
</table>

if (PSTATE_EL == EL0) then
    if (!ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.TGE == '1') && CNTKCTL_EL1.EL0VTEN == '0') then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
else
AArch64.AArch32SystemAccessTrap(EL1, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
AArch32.TakeHypTrapException(0x00);
else
UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FEAT_SEL2) then
(R[tz], R[t]) = (CNTHVS_CVAL_EL2<63:32>, CNTHVS_CVAL_EL2<31:0>);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
(R[tz], R[t]) = (CNTHV_CVAL_EL2<63:32>, CNTHV_CVAL_EL2<31:0>);
elsif PSTATE.EL == EL1 then
if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1TVT == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
else
(R[tz], R[t]) = (CNTV_CVAL<63:32>, CNTV_CVAL<31:0>);
elsif PSTATE.EL == EL2 then
(R[tz], R[t]) = (CNTV_CVAL<63:32>, CNTV_CVAL<31:0>);
elsif PSTATE.EL == EL3 then
(R[tz], R[t]) = (CNTV_CVAL<63:32>, CNTV_CVAL<31:0>);
else
MCRR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <Rt2>, <CRm>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1110</td>
<td>0b0011</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
if !ELUsingAArch32(EL1) && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL1.EL0VTEN == '0' then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
else
AArch64.AArch32SystemAccessTrap(EL1, 0x04);
elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' then
AArch32.TakeHypTrapException(0x00);
else
UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0' then
AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && SCR_EL3.NS == '0' &&
IsFeatureImplemented(FEAT_SEL2) then
CNTHVS_CVAL_EL2 = R[tz]:R[t];
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
CNTHV_CVAL_EL2 = R[tz]:R[t];
else
    CNTV_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1TVT == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        CNTV_CVAL = R[t2]:R[t];
    end if
elsif PSTATE.EL == EL2 then
    CNTV_CVAL = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
    CNTV_CVAL = R[t2]:R[t];
G8.7.23 CNTV_TVAL, Counter-timer Virtual Timer TimerValue register

The CNTV_TVAL characteristics are:

**Purpose**

Holds the timer value for the virtual timer.

**Configurations**

AArch32 System register CNTV_TVAL bits [31:0] are architecturally mapped to AArch64 System register CNTV_TVAL_EL0[31:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to CNTV_TVAL are UNDEFINED.

**Attributes**

CNTV_TVAL is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>TimerValue</th>
</tr>
</thead>
</table>

**TimerValue, bits [31:0]**

The TimerValue view of the virtual timer.

On a read of this register:

- If CNTV_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTV_CTL.ENABLE is 1, the value returned is (CNTV_CVAL - CNTVCT).

On a write of this register, CNTV_CVAL is set to (CNTVCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CNTP_CVAL) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- If CNTV_CTL.IMASK is 0, an interrupt is generated.

When CNTV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTV_TVAL**

Accesses to this register use the following encodings in the System register encoding space:

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VTEN == '0' then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
        AArch32.TakeHypTrapException(0x00);
    else
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VTEN == '0' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVT == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' &&
        IsFeatureImplemented(FEAT_SEL2) then
        if CNTHVS_CTL_EL2.ENABLE == '0' then
            R[t] = bits(32) UNKNOWN;
        else
            R[t] = (CNTHVS_CVAL_EL2 - PhysicalCountInt())<31:0>;
        endif
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '1' then
        if CNTHV_CTL_EL2.ENABLE == '0' then
            R[t] = bits(32) UNKNOWN;
        else
            R[t] = (CNTHV_CVAL_EL2 - PhysicalCountInt())<31:0>;
        endif
    else
        if CNTV_CTL.ENABLE == '0' then
            R[t] = bits(32)UNKNOWN;
        elsif HaveEL(EL2) && !ELUsingAArch32(EL2) then
            R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF_EL2))<31:0>;
        elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
            R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF))<31:0>;
        else
            R[t] = (CNTV_CVAL - PhysicalCountInt())<31:0>;
        endif
    endif
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1TVT == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif CNTV_CTL.ENABLE == '0' then
        if HaveEL(EL2) && !ELUsingAArch32(EL2) then
            R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF_EL2))<31:0>;
        elif HaveEL(EL2) && ELUsingAArch32(EL2) then
            R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF))<31:0>;
        else
            R[t] = (CNTV_CVAL - PhysicalCountInt())<31:0>;
        endif
    elsif PSTATE.EL == EL2 then
        if CNTV_CTL.ENABLE == '0' then
            R[t] = (CNTV_CVAL - (PhysicalCountInt() - CNTVOFF))<31:0>;
        else
            R[t] = (CNTV_CVAL - PhysicalCountInt())<31:0>;
        endif
    elsif PSTATE.EL == EL3 then
        if CNTV_CTL.ENABLE == '0' then
            R[t] = (CNTV_CVAL - PhysicalCountInt())<31:0>;
        else
            R[t] = (CNTV_CVAL - PhysicalCountInt())<31:0>;
        endif
    else
        if CNTV_CTL.ENABLE == '0' then
            R[t] = (CNTV_CVAL - PhysicalCountInt())<31:0>;
        else
            R[t] = (CNTV_CVAL - PhysicalCountInt())<31:0>;
        endif
    endif

MCR{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <CRn>, <CRm>{, (#)<opc2>}

<table>
<thead>
<tr>
<th>coproc</th>
<th>opc1</th>
<th>CRn</th>
<th>CRm</th>
<th>opc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b000</td>
<td>0b1110</td>
<td>0b0011</td>
<td>0b000</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  if !ELUsingAArch32(EL1) && !EL2Enabled() && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    AArch64.AArch32SystemAccessTrap(EL1, 0x03);
  elsif EL2Enabled() && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    UNDEFINED;
else
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
    AArch32.TakeHypTrapException(0x00);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && SCR_EL3.NS == '0' then
    CNTHS_CVAL_EL2 = SignExtend(R[t], 64) + PhysicalCountInt();
  else
    if HaveEL(EL2) && !ELUsingAArch32(EL2) then
      CNTV_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTVOFF_EL2;
    elseif HaveEL(EL2) && ELUsingAArch32(EL2) then
      CNTV_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTVOFF;
    else
      CNTV_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
 endif
endif
if PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x03);
  else
    if HaveEL(EL2) && !ELUsingAArch32(EL2) then
      CNTV_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTVOFF_EL2;
    elseif HaveEL(EL2) && ELUsingAArch32(EL2) then
      CNTV_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTVOFF;
    else
      CNTV_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
  endif
endif
if PSTATE.EL == EL2 then
  CNTV_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTVOFF;
else
  if HaveEL(EL2) then
    CNTV_CVAL = (SignExtend(R[t], 64) + PhysicalCountInt()) - CNTVOFF;
  else
    CNTV_CVAL = SignExtend(R[t], 64) + PhysicalCountInt();
G8.7.24 CNTVCT, Counter-timer Virtual Count register

The CNTVCT characteristics are:

**Purpose**

Holds the 64-bit virtual count value. The virtual count value is equal to the physical count value minus the virtual offset visible in CNTVOFF.

**Configurations**

AArch32 System register CNTVCT bits [63:0] are architecturally mapped to AArch64 System register CNTVCT_EL0[63:0].

This register is present only when AArch32 is supported. Otherwise, direct accesses to CNTVCT are UNDEFINED.

The value of this register is the same as the value of CNTPCT in the following conditions:

- When EL2 is not implemented.
- When EL2 is implemented and is using AArch64, HCR_EL2.{E2H, TGE} is \{1, 1\}, and this register is read from Non-secure EL0.

All reads to the CNTVCT occur in program order relative to reads to CNTVCTSS or CNTVCT.

**Attributes**

CNTVCT is a 64-bit register.

**Field descriptions**

![Virtual count value](image)

**Bits [63:0]**

Virtual count value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTVCT**

Accesses to this register use the following encodings in the System register encoding space:

```
MRRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <Rt2>, <CRm>
```

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1110</td>
<td>0b0001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VCTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x04);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x04);
    else
        AArch64.AArch32SystemAccessTrap(EL1, 0x04);
else if ELUsingAArch32(EL1) && CNTKCTL.PL0VCTEN == '0' then
if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
    AArch32.TakeHypTrapException(0x00);
else
    UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VCTEN == '0' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVCT == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
else
    if HaveEL(EL2) && !ELUsingAArch32(EL2) && (!EL2Enabled() || HCR_EL2.<E2H,TGE> != '11') then
        (R[t2], R[t]) = Split64to32(PhysicalCountInt() - CNTVOFF_EL2);
    elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
        (R[t2], R[t]) = Split64to32(PhysicalCountInt() - CNTVOFF);
    else
        (R[t2], R[t]) = Split64to32(PhysicalCountInt());
    endif
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1TVCT == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
        if HaveEL(EL2) && !ELUsingAArch32(EL2) then
            (R[t2], R[t]) = Split64to32(PhysicalCountInt() - CNTVOFF_EL2);
        elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
            (R[t2], R[t]) = Split64to32(PhysicalCountInt() - CNTVOFF);
        else
            (R[t2], R[t]) = Split64to32(PhysicalCountInt());
        endif
    endif
elsif PSTATE.EL == EL2 then
    if HaveEL(EL2) then
        (R[t2], R[t]) = Split64to32(PhysicalCountInt() - CNTVOFF);
    else
        (R[t2], R[t]) = Split64to32(PhysicalCountInt());
    endif
elsif PSTATE.EL == EL3 then
    if HaveEL(EL2) then
        (R[t2], R[t]) = Split64to32(PhysicalCountInt() - CNTVOFF);
    else
        (R[t2], R[t]) = Split64to32(PhysicalCountInt());
    endif
G8.7.25 CNTVCTSS, Counter-timer Self-Synchronized Virtual Count register

The CNTVCTSS characteristics are:

**Purpose**

Holds the 64-bit virtual count value. The virtual count value is equal to the physical count value visible in CNTPCT minus the virtual offset visible in CNTVOFF.

**Configurations**

AArch32 System register CNTVCTSS bits [63:0] are architecturally mapped to AArch64 System register CNTVCTSS_EL0[63:0].

This register is present only when AArch32 is supported and FEAT_ECV is implemented. Otherwise, direct accesses to CNTVCTSS are UNDEFINED.

All reads to the CNTVCTSS occur in program order relative to reads to CNTVCT or CNTVCTSS. This register is a self-synchronised view of the CNTVCT counter, and cannot be read speculatively.

**Attributes**

CNTVCTSS is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self-Synchronized Virtual count value</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self-Synchronized Virtual count value</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:0]**

Self-Synchronized Virtual count value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing CNTVCTSS**

Accesses to this register use the following encodings in the System register encoding space:

### MRRC{<c>}{<q>} <coproc>, (#)<opc1>, <Rt>, <Rt2>, <CRm>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1110</td>
<td>0b1001</td>
</tr>
</tbody>
</table>

if PSTATE_EL == EL0 then
  if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '1') && CNTKCTL_EL1.EL0VCTEN == '0' then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x04);
  elsif ELUsingAArch32(EL1) && CNTKCTL.PL0VCTEN == '0' then
    if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x04);
    elseif ELUsingAArch32(EL1) && CNTKCTL.PL0VCTEN == '0' then
      if EL2Enabled() && ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x04);
      elseif EL2Enabled() && ELUsingAArch32(EL2) && HCR.TGE == '1' then
        AArch32.TakeHypTrapException(0x00);
else
  UNDEFINED;
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VCTEN == '0' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x04);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> != '11' && CNTHCTL_EL2.EL1TVCT == '1' then
  AArch64.AArch32SystemAccessTrap(EL2, 0x04);
else
  if HaveEL(EL2) && !ELUsingAArch32(EL2) && (EL2Enabled() || HCR_EL2.<E2H,TGE> != '11') then
    (R[t2], R[t]) = Split64to32(PhysicalCountInt() - CNTVOFF_EL2);
  elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
    (R[t2], R[t]) = Split64to32(PhysicalCountInt() - CNTVOFF);
  else
    (R[t2], R[t]) = Split64to32(PhysicalCountInt());
  endif
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && !ELUsingAArch32(EL2) && CNTHCTL_EL2.EL1TVCT == '1' then
    AArch64.AArch32SystemAccessTrap(EL2, 0x04);
  elsif EL2Enabled() && !ELUsingAArch32(EL2) && (!EL2Enabled() || HCR_EL2.<E2H,TGE> != '11') then
    (R[t2], R[t]) = Split64to32(PhysicalCountInt() - CNTVOFF_EL2);
  elsif EL2Enabled() && ELUsingAArch32(EL2) then
    (R[t2], R[t]) = Split64to32(PhysicalCountInt() - CNTVOFF);
  else
    (R[t2], R[t]) = Split64to32(PhysicalCountInt());
  endif
elsif PSTATE.EL == EL2 then
  if HaveEL(EL2) then
    (R[t2], R[t]) = Split64to32(PhysicalCountInt() - CNTVOFF);
  elsif PSTATE.EL == EL3 then
    if HaveEL(EL2) then
      (R[t2], R[t]) = Split64to32(PhysicalCountInt() - CNTVOFF);
    else
      (R[t2], R[t]) = Split64to32(PhysicalCountInt());
    endif
  endif

G8.7.26 CNTVOFF, Counter-timer Virtual Offset register

The CNTVOFF characteristics are:

Purpose

Holds the 64-bit virtual offset. This is the offset between the physical count value visible in CNTPCT and the virtual count value visible in CNTVCT.

Configurations

AArch32 System register CNTVOFF bits [63:0] are architecturally mapped to AArch64 System register CNTVOFF_EL2[63:0].

This register is present only when EL2 is capable of using AArch32. Otherwise, direct accesses to CNTVOFF are UNDEFINED.

If EL2 is not implemented, this register is RES0 from EL3 and the virtual counter uses a fixed virtual offset of zero.

Note

When EL2 is implemented and is using AArch64, if HCR_EL2.{E2H, TGE} is {1, 1}, the virtual counter uses a fixed virtual offset of zero when CNTVCT is read from Non-secure EL0.

Attributes

CNTVOFF is a 64-bit register.

Field descriptions

Bits [63:0]

Virtual offset.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVOFF

Accesses to this register use the following encodings in the System register encoding space:

\[
\text{MRRC}(<c>\{<q>\}, \{#\}<opc1>, <Rt>, <Rt2>, <CRm>
\]

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1110</td>
<td>0b0100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
UNDEFINED;
elsif PSTATE.EL == EL2 then
  (R[t2], R[t]) = (CNTVOFF<63:32>, CNTVOFF<31:0>);
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    (R[t2], R[t]) = (CNTVOFF<63:32>, CNTVOFF<31:0>);
end if;


MCRR{<c>}{<q>} <coproc>, (#<opc1>, <Rt>, <Rt2>, <CRm>

<table>
<thead>
<tr>
<th>coproc</th>
<th>CRm</th>
<th>opc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>0b1110</td>
<td>0b0100</td>
</tr>
</tbody>
</table>

if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  UNDEFINED;
elsif PSTATE.EL == EL2 then
  CNTVOFF = R[t2]:R[t];
elsif PSTATE.EL == EL3 then
  if SCR.NS == '0' then
    UNDEFINED;
  else
    CNTVOFF = R[t2]:R[t];
end if;
Part H
External Debug
Chapter H1
About External Debug

This chapter gives an overview of A-profile external debug and specifies the required debug authentication. It contains the following sections:

• *Introduction to external debug on page H1-10228.*
• *External debug on page H1-10229.*
• *Required debug authentication on page H1-10230.*

--- Note ---
For information about self-hosted debug, see Chapter D2 *AArch64 Self-hosted Debug* and Chapter G2 *AArch32 Self-hosted Debug.*
H1.1 Introduction to external debug

The A-profile supports both:

**Self-hosted debug**

The PE itself hosts a debugger. That is, developers developing software to run on the PE use debugger software running on the same PE.

**External debug**

The debugger is external to the PE. The debugging might be either on-chip, for example in a second PE, or off-chip, for example a JTAG debugger that accesses the chip through a Debug Access Port.

External debug is particularly useful for:

- **Hardware bring-up.** That is, debugging during development when a system is first powered up and not all of the software functionality is available.
- **PEs that are deeply embedded inside systems.**

To support external debug, the Arm architecture defines required features that are called *external debug features*.

**Note**

An external debugger has a potentially high level of control over and visibility into the PE. The system sets this level using debug authentication. See *Required debug authentication* on page H1-10230.

If the debug authentication level is set too low, agents may be able to bypass elements of the security and privilege models. This includes both off-chip agents and on-chip agents such as unprivileged or Non-secure software.

H1.1.1 Definition and constraints of a debugger in the context of external debug

When the description of external debug in this Part of the manual describes a *debugger* as controlling external debug this debugger might be a second on-chip PE or an off-chip device such as a JTAG debugger using a Debug Access Port (DAP).

If a Debug Access Port is implemented:

- When debug is prohibited at the Debug Access Port, the port must not generate accesses to the external debug interface of the PE.
- When Secure debug is prohibited at the Debug Access Port, the port must not generate Secure accesses to the external debug interface of the PE.
- When Secure accesses are allowed at the Debug Access Port, the port must be able to generate Secure accesses.

If `FEAT_Debugv8p4` is not implemented, accesses to the PE are controlled by the external authentication interface functions, `ExternalInvasiveDebugEnabled()`, `ExternalNoninvasiveDebugEnabled()`, `ExternalSecureNoninvasiveDebugEnabled()` and `ExternalSecureInvasiveDebugEnabled()`. The external authentication interface functions override MDCR_EL3.{EPMAD, EDAD}.

If `FEAT_TRF` is implemented, the bus Requester, which may be the Debug Access Port, controls the accesses it makes to the PE and MDCR_EL3.{EPMAD, EDAD} control Non-secure access to registers.

The Debug Access Port is not required to use the same authentication interface as the PE.

Arm recommends the following authentication interface:

- When `ExternalSecureInvasiveDebugEnabled()` == FALSE at the PE, Secure debug is disabled at the DAP.
- When `ExternalInvasiveDebugEnabled()` == FALSE at the PE, all debug is prohibited at the DAP.
H1.2 External debug

Debug events allow an external debugger to halt the PE. The A-profile provides the following debug events:

- **Halting Step debug events** on page H3-10276:
  - The debugger can use this resource to make the PE step through code one line at a time.

- **Halt Instruction debug event** on page H3-10286:
  - This might occur when software executes the Halting breakpoint instruction, HLT.

- **Exception Catch debug event** on page H3-10287:
  - This can be programmed to occur on all entries to a given Exception level.

- **External Debug Request debug event** on page H3-10291:
  - An embedded cross-trigger can signal this debug event.

- **OS Unlock Catch debug event** on page H3-10292:
  - This might occur when the state of the OS Lock changes from locked to unlocked.

- **Reset Catch debug events** on page H3-10293:
  - This might occur when the PE exits reset state.

- **Software Access debug event** on page H3-10294:
  - This can be programmed to occur when software tries to access the Breakpoint Value registers, the Breakpoint Control registers, the Watchpoint value registers, or the Watchpoint Control registers. It caused a trap to Debug state.

Breakpoints and watchpoints can also halt the PE.

When the PE is in Debug state:

- It stops executing instructions from the location indicated by the program counter, and is instead controlled through the external debug interface.
- The Instruction Transfer Register, ITR, passes instructions to the PE to execute in Debug state:
  - The ITR contains a single register, EDITR, and associated flow-control flags.
- The Debug Communications Channel, DCC, passes data between the PE and the debugger:
  - The DCC includes the data transfer registers, DTRRX and DTRTX, and associated flow-control flags.
  - Although the DCC is an essential part of Debug state operation, it can also be used in Non-debug state.
- The PE cannot service any interrupts in Debug state.

Chapter H2 Debug State describes Debug state in more detail.
H1.3 Required debug authentication

Any implementation must provide the debug authentication defined in this section, which controls:

- Whether the PE can halt.
- Whether some aspects of non-invasive debug are permitted.
- Some legacy aspects of the AArch32 self-hosted debug model.

The pseudocode functions shown in Table H1-1 on page H1-10230, and the conditions that follow that table, define the architectural requirements for debug authentication.

The following conditions always apply:

- If `ExternalInvasiveDebugEnabled()` is FALSE then `ExternalSecureInvasiveDebugEnabled()` is FALSE.
- If `ExternalNoninvasiveDebugEnabled()` is FALSE then `ExternalSecureNoninvasiveDebugEnabled()` is FALSE.
- If `ExternalInvasiveDebugEnabled()` is TRUE then `ExternalNoninvasiveDebugEnabled()` is TRUE.
- If `ExternalSecureInvasiveDebugEnabled()` is TRUE then `ExternalSecureNoninvasiveDebugEnabled()` is TRUE.

If FEAT_Debugv8p4 is implemented:

- `ExternalNoninvasiveDebugEnabled()` always returns TRUE.
- `ExternalSecureNoninvasiveDebugEnabled()` returns the same as `ExternalSecureInvasiveDebugEnabled()`.

Arm recommends the use of the interface described in Recommended authentication interface on page K6-11687 to provide this debug authentication. The pseudocode functions in Chapter J1 Armv8 Pseudocode, which are linked to by the entries in the Pseudocode function on page H1-10230 column of Table H1-1 on page H1-10230, assume that this interface is implemented.

### Table H1-1 Debug authentication functions

<table>
<thead>
<tr>
<th>Pseudocode function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ExternalSecureNoninvasiveDebugEnabled()</code></td>
<td>Returns TRUE if Secure non-invasive debug is enabled.</td>
</tr>
<tr>
<td><code>AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()</code></td>
<td>Returns TRUE if Secure invasive self-hosted debug is enabled in AArch32 state.</td>
</tr>
<tr>
<td><code>ExternalSecureInvasiveDebugEnabled()</code></td>
<td>Returns TRUE if Secure invasive debug is enabled.</td>
</tr>
<tr>
<td><code>ExternalNoninvasiveDebugEnabled()</code></td>
<td>Returns TRUE if Non-secure non-invasive debug is enabled.</td>
</tr>
<tr>
<td><code>ExternalInvasiveDebugEnabled()</code></td>
<td>Returns TRUE if Non-secure invasive debug is enabled.</td>
</tr>
</tbody>
</table>

The following conditions always apply:

- If `ExternalInvasiveDebugEnabled()` is FALSE then `ExternalSecureInvasiveDebugEnabled()` is FALSE.
- If `ExternalNoninvasiveDebugEnabled()` is FALSE then `ExternalSecureNoninvasiveDebugEnabled()` is FALSE.
- If `ExternalInvasiveDebugEnabled()` is TRUE then `ExternalNoninvasiveDebugEnabled()` is TRUE.
- If `ExternalSecureInvasiveDebugEnabled()` is TRUE then `ExternalSecureNoninvasiveDebugEnabled()` is TRUE.

If FEAT_Debugv8p4 is implemented:

- `ExternalNoninvasiveDebugEnabled()` always returns TRUE.
- `ExternalSecureNoninvasiveDebugEnabled()` returns the same as `ExternalSecureInvasiveDebugEnabled()`.
This chapter describes Debug state. It contains the following sections:

- About Debug state on page H2-10232.
- Halting the PE on debug events on page H2-10233.
- Entering Debug state on page H2-10239.
- Behavior in Debug state on page H2-10242.
- Exiting Debug state on page H2-10271.

**Note**

Table K17-1 on page K17-11836 disambiguates the general register references used in this chapter.
H2.1 About Debug state

In external debug, debug events allow an external debugger to halt the PE. The PE then enters Debug state. When the PE is in Debug state:

- It stops executing instructions from the location indicated by the program counter, and is instead controlled through the external debug interface.
- The Instruction Transfer Register, ITR, passes instructions to the PE to execute in Debug state.
- The Debug Communications Channel, DCC, passes data between the PE and the debugger.

The PE cannot service any interrupts in Debug state.
H2.2 Halting the PE on debug events

For details of debug events, see Introduction to Halting debug events on page H3-10274 and Breakpoint and Watchpoint debug events on page H2-10234.

On a debug event, the PE must do one of the following:
• Enter Debug state.
• Pend the debug event.
• Generate a debug exception.
• Ignore the debug event.

This behavior depends on both:
• Whether halting is allowed by the current state of the debug authentication interface. See Halting allowed and halting prohibited.
• The type of debug event and the programming of the debug control registers.
  — See Halting debug events for all Halting debug events.
  — See Breakpoint and Watchpoint debug events on page H2-10234 for Breakpoint and Watchpoint debug events.

See also Other debug exceptions on page H2-10234.

This means that behavior can be CONSTRAINED UNPREDICTABLE if the conditions change. See Synchronization and Halting debug events on page H3-10295.

Summary of debug events and possible outcomes on page H3-10274 summarizes the possible outcomes of each type of debug event.

H2.2.1 Halting allowed and halting prohibited

Halting can be either allowed or prohibited:
• Halting is always prohibited in Debug state.
• Halting is always prohibited when DoubleLockStatus() == TRUE.
  — This means that FEAT_DoubleLock is implemented and OS Double lock is locked.
• Halting is also controlled by the IMPLEMENTATION DEFINED authentication interface, and is prohibited when either:
  — The PE is in Non-secure state and ExternalInvasiveDebugEnabled() == FALSE.
  — The PE is in Secure state and ExternalSecureInvasiveDebugEnabled() == FALSE.

——— Note ————
See Appendix K6 Recommended External Debug Interface for more information on these functions.

• Otherwise, halting is allowed.

For more information, see:
• Pseudocode description of Halting on debug events on page H2-10238
• Required debug authentication on page H1-10230.

H2.2.2 Halting debug events

The Halting debug events are described in Chapter H3 Halting Debug Events.

When a Halting debug event is generated, it causes entry to Debug state if all of the following are true:
• Halting is allowed. See Halting allowed and halting prohibited.
• The Halting debug event is one of:
  — A Halt Instruction debug event and EDSRCR.HDE == 1.
  — A Software Access debug event and OSLSR_EL1.OSLK == 0, meaning that the OS Lock is unlocked.
  — Neither a Halt Instruction debug event nor a Software Access debug event.
Note

— A Halt Instruction debug event is the only Halting debug event that relies on \texttt{EDSCR.HDE == 1}.
— Halting on Breakpoint and Watchpoint debug events is also controlled by \texttt{EDSCR.HDE}. See \textit{Breakpoint and Watchpoint debug events}.
— \texttt{EDSCR.HDE} can be written by software when the OS Lock is locked. Privileged code can use \texttt{MDCR_EL3.TDOSA} and \texttt{HDCR.TDOSA} to trap writes to these registers.

If a Halting debug event does not generate entry to Debug state because the conditions listed in this section do not hold, then:

• If the Halting debug event is a Halt Instruction debug event, the instruction that generated the Halting debug event is treated as \texttt{UNDEFINED}.
• If the Halting debug event is a Software Access debug event, it is ignored.
• If the Halting debug event is an Exception Catch debug event, it is ignored except when \texttt{FEAT_Debugv8p8} is implemented, in which case the event might be pended.

In all other cases the Halting debug event is pended, see \textit{Pending Halting debug events on page H3-10295}.

\textit{Summary of actions from debug events on page H2-10237} summarizes the possible outcome for each type of Debug event.

Note

Halting debug events never generate debug exceptions.

H2.2.3 Breakpoint and Watchpoint debug events

A breakpoint or watchpoint generates an entry to Debug state if all of the following conditions hold:

• Halting debug is enabled, that is \texttt{EDSCR.HDE == 1}.
• Halting is allowed. See \textit{Halting allowed and halting prohibited on page H2-10233}.
• The OS Lock is unlocked, that is \texttt{OSLSR.OSLK == 0}.

The Address Mismatch breakpoint type is reserved when all of these conditions are met.

\texttt{MDSCR_EL1.MDE} or \texttt{DBGDSCRExt.MDBGen} is ignored when determining whether to enter Debug state. A breakpoint or watchpoint that generates entry to Debug state is a Breakpoint or Watchpoint debug event and does not generate a debug exception.

A breakpoint or watchpoint that does not generate an entry to Debug state either:

• Generates a Breakpoint or Watchpoint exception.
• Is ignored.

Note

\texttt{EDSCR.HDE} is ignored when determining whether to generate a debug exception. The debug exception is suppressed only if the PE enters Debug state. This means that the use of Halting debug mode in Non-secure state does not affect the Exception model in Secure state.

See Chapter D2 \textit{AArch64 Self-hosted Debug}, Chapter G2 \textit{AArch32 Self-hosted Debug}, and the Note in \textit{Other debug exceptions}.

H2.2.4 Other debug exceptions

The following events never generate entry to Debug state:

• Breakpoint Instruction exceptions.
• Software Step exceptions.
• Vector Catch exceptions.
The behavior of these events is unchanged when Halting debug mode is enabled, that is when EDSCR.HDE == 1. This means that these events can do one of the following:

• They can generate a debug exception.
• They can be ignored.

For additional information, see Chapter D2 AArch64 Self-hosted Debug and Chapter G2 AArch32 Self-hosted Debug.

## H2.2.5 Debug state entry and debug event prioritization

The following are synchronous debug events:

• Breakpoint debug event.
• Watchpoint debug event.
• Halting Step debug event.
• Halt Instruction debug event.
• Exception Catch debug event.
• Software Access debug event.
• Reset Catch debug event.

Each of these synchronous debug events are treated as a synchronous exception generated by an instruction, or by the taking of an exception or reset. That is, if halting is allowed, the synchronous debug event must be taken before any subsequent instructions are executed. Reset Catch debug events must be taken before the PE executes the instruction at the reset vector.

Note

• Reset Catch and Exception Catch debug events might be generated asynchronously, because they can result from an asynchronous exception. However, if halting is allowed after the reset or asynchronous exception has been processed, the Reset Catch or Exception Catch debug event is taken synchronously.

• The Halting Step debug event is generated by the instruction after the stepped instruction. Therefore, if the stepped instruction generates any other synchronous exceptions or debug events these are taken first.

If halting is prohibited then Halting Step debug events and Reset Catch debug events might be pended and taken asynchronously. OS Unlock Catch debug events are always pended and taken asynchronously. See Pending Halting debug events on page H3-10295.

The architecture does not define when asynchronous debug events are taken, and therefore the prioritization of asynchronous debug events is IMPLEMENTATION DEFINED. See Synchronization and Halting debug events on page H3-10295.

The following list shows how the synchronous debug events are prioritized, with 1 being the highest priority.

Note

The priority numbering is the same as the numbering for AArch64 synchronous exception priorities listed in Synchronous exception types on page D1-4646, and in particular Prioritization of interrupts on page D1-4664. This numbering correlates with the equivalent AArch32 list in Exception prioritization for exceptions taken to AArch32 state on page G1-8932.

The priority for synchronous debug events is as follows:

1

Reset Catch debug event. See Reset Catch debug events on page H3-10293.

This debug event has a higher priority than the synchronous exceptions listed in Synchronous exception types on page D1-4646.

2

Exception Catch debug event. See Exception Catch debug event on page H3-10287.

This debug event can be assigned one of two priorities. When it has a priority of 2, it has a higher priority than the synchronous exceptions listed in the Exception model.

3

Halting Step debug event. See Halting Step debug events on page H3-10276.
This debug event has a higher priority than the synchronous exceptions listed in the Exception model.

4 This event is not a debug event.
5 Exception Catch debug event. See Exception Catch debug event on page H3-10287.
   This debug event can be assigned one of two priorities, 0 or 5.
6-7 These events are not debug events.
8 Breakpoint exception or debug event or Address Matching Vector Catch exception. See Breakpoint exceptions on page D2-4707, and Vector Catch exceptions on page G2-9093.
   These two debug events have the same priority.
9 This event is not a debug event.
10 Halt Instruction debug event. See Halt Instruction debug event on page H3-10286.
11-26 These events are not debug events.
27 Software Access debug event. See Software Access debug event on page H3-10294.
28-31 These events are not debug events.
32 Watchpoint exception or debug event. See Watchpoint exceptions on page D2-4726 for exceptions taken from AArch64 state, or Watchpoint exceptions on page G2-9079 for exceptions taken from AArch32 state.
33 This event is not a debug event.

For Reset Catch debug events and Halting Step debug events, the priorities listed in this section apply only when halting is allowed at the time the event is generated. This means that the event is taken synchronously and not pended.

For more information on the prioritization of exceptions, see:
- Synchronous exception types on page D1-4646.
- Prioritization of interrupts on page D1-4664.
- Exception prioritization for exceptions taken to AArch32 state on page G1-8932. This section covers synchronous and asynchronous exceptions.

**Breakpoint debug events and Vector Catch exception**

An Address Matching Vector Catch exception has the same priority as a Breakpoint debug event. See Prioritization of Synchronous exceptions taken to AArch64 state on page D1-4650.

The prioritization of these events is unchanged even if the breakpoint generates entry to Debug state instead of a Breakpoint exception. This means that if a single instruction generates both an Address Matching Vector Catch exception and a Breakpoint debug event, there is a CONstrained UNpredictable choice of:
- The PE entering Debug state due to the Breakpoint debug event.
- A Vector Catch exception.

This applies only if all of the following are true:
- Halting debug is enabled.
- Halting is allowed.
- The OS Lock is unlocked.

An Exception Trapping Vector Catch exception must be generated immediately following the exception that generated it. This means that it does not appear in the priority table.

**H2.2.6 Imprecise entry to Debug state**

Debug state entry is normally precise. This means that the PE cannot enter Debug state if it can neither complete nor abandon all currently executing instructions and leave the PE in a precise state. See Definition of a precise exception and imprecise exception on page D1-4638.

A debugger can write a value of 1 to EDRCR.CBRRQ to allow imprecise entry to Debug state. An External Debug Request debug event must be pending before writing 1 to this bit. Support for this feature is OPTIONAL and it is IMPLEMENTATION DEFINED when it is effective at forcing entry to Debug state.
The PE ignores writes to this bit if either:
- External debugging is not enabled, meaning `ExternalInvasiveDebugEnabled()` == FALSE.
- Secure external debugging is not enabled, meaning `ExternalSecureInvasiveDebugEnabled()` == FALSE, and either:
  - EL3 is not implemented and the implemented Security state is Secure state.
  - EL3 is implemented.

**Example H2-1** shows how entry to Debug state can be forced.

---

**Example H2-1 Forcing entry to Debug state**

The debugger pends an External Debug Request debug event through the CTI to halt a program that has stopped responding. However, the memory system is not responding and a memory access instruction cannot complete. This means that Debug state cannot be entered precisely. The debugger writes a value of 1 to `EDCR.CBRRQ`. The PE cancels all outstanding memory accesses and enters Debug state. As some instructions might not have completed correctly, entry to Debug state is imprecise.

When Debug state is entered imprecisely, all memory access instructions executed through the ITR have CONSTRAINED UNPREDICTABLE behavior. The value of all registers is UNKNOWN, but might be useful for diagnostic purposes.

---

**H2.2.7 Summary of actions from debug events**

**Table H2-1 on page H2-10238** shows the Software and Halting debug events. In **Table H2-1 on page H2-10238**, the columns have the following meaning:

**Debug event type**

This means the type of debug event where:

- **Other software**
  - Means one of:
    - *Software Step exceptions* on page D2-4742.
    - *Breakpoint Instruction exceptions* on page D2-4705.
    - *Vector Catch exceptions* on page D2-4741 for AArch64 state or *Vector Catch exceptions* on page G2-9093 for AArch32 state.

- **Other Halting**
  - Means one of the following:
    - *Halting Step debug events* on page H3-10276.
    - *External Debug Request debug event* on page H3-10291.
    - *Reset Catch debug events* on page H3-10293.
    - *OS Unlock Catch debug event* on page H3-10292.

Other debug events are referred to explicitly.

**Authentication**

This means halting is allowed by the IMPLEMENTATION DEFINED external authentication interface. It is the result of one of the following pseudocode functions:

- **In Secure state** `ExternalInvasiveDebugEnabled()`.
- **In Non-secure state** `ExternalSecureInvasiveDebugEnabled()`.

**DLK**

This indicates whether `FEAT_DoubleLock` is implemented and locked, `DoubleLockStatus()` == TRUE.

**OSLK**

This is the value of `OSLSR.OSLK`. It indicates whether the OS Lock is locked.

**HDE**

This is the value of `EDSCR.HDE`. It indicates whether Halting debug is enabled.
The letter X in Table H2-1 on page H2-10238 indicates that the value can be either 0 or 1.

### Table H2-1 Debug authentication for external debug

<table>
<thead>
<tr>
<th>Debug event type</th>
<th>Authentication</th>
<th>DLK</th>
<th>OSLK</th>
<th>HDE</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other software</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Handled by the Exception model</td>
</tr>
<tr>
<td>Breakpoint or Watchpoint debug event</td>
<td>X</td>
<td>TRUE</td>
<td>X</td>
<td>X</td>
<td>Handled by the Exception model (ignored)</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>FALSE</td>
<td>1</td>
<td>X</td>
<td>Handled by the Exception model (ignored)</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>FALSE</td>
<td>0</td>
<td>X</td>
<td>Handled by the Exception model</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>FALSE</td>
<td>0</td>
<td>0</td>
<td>Handled by the Exception model</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>FALSE</td>
<td>0</td>
<td>1</td>
<td>Entry to Debug state</td>
</tr>
<tr>
<td>Halt Instruction debug event</td>
<td>FALSE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>UNDEFINED</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>TRUE</td>
<td>X</td>
<td>X</td>
<td>UNDEFINED</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>FALSE</td>
<td>X</td>
<td>0</td>
<td>UNDEFINED</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>FALSE</td>
<td>X</td>
<td>1</td>
<td>Entry to Debug state</td>
</tr>
<tr>
<td>Exception Catch debug event</td>
<td>FALSE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Might be pended (a)</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>TRUE</td>
<td>X</td>
<td>X</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>FALSE</td>
<td>X</td>
<td>X</td>
<td>Entry to Debug state</td>
</tr>
<tr>
<td>Software Access debug event</td>
<td>FALSE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>TRUE</td>
<td>X</td>
<td>X</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>FALSE</td>
<td>1</td>
<td>X</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>FALSE</td>
<td>0</td>
<td>X</td>
<td>Entry to Debug state</td>
</tr>
<tr>
<td>Other Halting</td>
<td>FALSE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Debug event is pended</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>TRUE</td>
<td>X</td>
<td>X</td>
<td>Debug event is pended</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>FALSE</td>
<td>X</td>
<td>X</td>
<td>Entry to Debug state</td>
</tr>
</tbody>
</table>

\(a\). See Exception Catch debug event on page H3-10287.

### H2.2.8 Pseudocode description of Halting on debug events

The `Halted()`, `Restarting()`, `HaltingAllowed()`, and `HaltOnBreakpointOrWatchpoint()` functions are described in the pseudocode.
H2.3 Entering Debug state

On entry to Debug state, the preferred restart address and PSTATE are saved in DLR and DSPSR. The PE remains in the mode and security state from which it entered Debug state.

If EDRCR.CBRRQ has a value of 0, entry to Debug state is precise. If EDRCR.CBRRQ has a value of 1, then imprecise entry to Debug state is permitted.

If a Watchpoint debug event causes an entry to Debug state, the address of the access that generated the Watchpoint debug event is recorded in EDWAR.

For more information, see:

- Determining the memory location that caused a Watchpoint exception on page D2-4734 for a debug event taken from AArch64 state.
- Determining the memory location that caused a Watchpoint exception on page G2-9086 for a debug event taken from AArch32 state.

Other than the effect on PSTATE and EDSCR, entry to Debug state is not a Context synchronization event. The effects of entry to Debug state on PSTATE and EDSCR are synchronized.

H2.3.1 Entering Debug state from AArch32 state

When entering Debug state from AArch32 state, the PE remains in AArch32 state. In AArch32 Debug state the PE executes T32 instructions, regardless of the value of PSTATE.T before entering Debug state.

To allow the debugger to determine the state of the PE, the current Execution state for all four Exception levels can be read from EDSCR.RW, and the current Exception level can be read from EDSCR.EL.

The current endianness state, PSTATE.E, is unchanged on entry to Debug state.

Note

- If EL1 is using AArch32 state, the current endianness state can differ from that indicated by SCTLR.EE.
- If EL2 is using AArch32 state, the current endianness state can differ from that indicated by HSCTLR.EE.
- On entry to Debug state from AArch32 state, PSTATE.SS is copied to DSPSR.SS, even though the PE remains in AArch32 state.

See also Effect of entering Debug state on PSTATE on page H2-10240.

H2.3.2 Effect of Debug state entry on DLR and DSPSR

DLR is set to the preferred restart address for the debug event, which depends on the event type. The value of PSTATE is saved in DSPSR.

For entry to Debug state from AArch32 state, the values saved in DSPSR.IT are always correct for the preferred restart address.

For synchronous Halting debug events, the preferred restart address is the address of the instruction that generated the debug event. It is CONstrained UNPREDICTable whether DSPSR_EL0.BTYPE is set to the value of PSTATE.BTYPE or 0 for synchronous debug events other than the following debug events:

- A Halting Step debug event.
- A Breakpoint debug event.
- A Halt Instruction debug event.

For asynchronous Halting debug events, including pending Halting debug events taken asynchronously, the preferred restart address is the address of the first instruction that must be executed on exit from Debug state.

This means that:

- For Breakpoint and Watchpoint debug events, the preferred restart address is the same as the preferred return address for a debug exception, as described in Chapter D2 AArch64 Self-hosted Debug and Chapter G2 AArch32 Self-hosted Debug.
• For Halt Instruction debug events, DLR is set to the address of the HLT instruction and DSPSR.IT is correct for the HLT instruction.
• For Software Access debug events, DLR is set to the address of the accessing instruction and DSPSR.IT is correct for this instruction.
• For Halting Step debug events taken synchronously, DLR and DSPSR are set as the ELR and SPSR would be set for a Software Step exception. This is usually the address of, and PSTATE for, the instruction after the one that was stepped.
• For Exception Catch debug events, DLR is set to the address of the first instruction that must be executed on exit from Debug state, and DSPSR is correct for this instruction. This is the exception vector or reset vector address.
  — If the debug event is generated on taking an exception to a trapped Exception level, the DLR is set to the address of the exception vector the PE would have started fetching from. This is UNKNOWN if the VBAR for the Exception level has never been initialized. The DSPSR records the value of PSTATE after taking the exception. The Exception Catch occurs after the SPSR and the Link register are set, and the debugger can use these registers to determine where in the application program the exception occurred.
  — If the debug event is generated on an exception return to a trapped Exception level, the DLR is set to the target address of the exception return and the DSPSR records the value of PSTATE after the exception return.
• Reset Catch debug events taken synchronously behave like Exception Catch debug events.
• For Reset Catch debug events and Exception Catch debug events generated on reset to a trapped Exception level, the DLR is set to the address of the last instruction that must be executed on exit from Debug state and the DSPSR records the reset value of PSTATE.
• For pending Halting debug events and External Debug Request debug events, DLR is set to the address of the first instruction that must be executed on exit from Debug state and DSPSR.IT is correct for this instruction. See Pending Halting debug events on page H3-10295.

Normally DLR is aligned according to the instruction set state indicated in DSPSR. However, a debug event might be taken at a point where the PC is not aligned.

**H2.3.3 Effect of Debug state entry on System registers, the Event register, and Exclusives monitors**

Entering Debug state has no effect on System registers other than DLR and DSPSR. In particular, ESRs, FARs, and FSRs are not updated on entering Debug state. SCR is unchanged, even when entering Debug state from EL3.

Entering Debug state has no architecturally-defined effect on the Event Register and Exclusives monitors.

— Entry to Debug state might set the Event Register or clear the Exclusives monitors, or both. However, this is not a requirement, and debuggers must not rely on any implementation specific behavior.

Unless otherwise described in this reference manual, instructions executed in Debug state have their architecturally-defined effects on the System registers, the Event register, and Exclusives monitors.

**H2.3.4 Effect of entering Debug state on PSTATE**

The effect of an entry to Debug state on PSTATE is described in Entering Debug state on page H2-10239 and Entering Debug state from AArch32 state on page H2-10239.

On entry to Debug state after PSTATE is saved in DSPSR:
• PSTATE.IL is cleared to 0.
• PSTATE.TCO is set to 1.
• PSTATE.BTYPE is set to 0.
• PSTATE.{IT, T, SS, D, A, I, F, SSBS, ALLINT} are set to UNKNOWN values
PSTATE.\{N, Z, C, V, GE, E, M, nRW, EL, SP, PAN, UAO, DIT\} are unchanged.

For more information, see PSTATE in Debug state on page H2-10242.

H2.3.5 Entering Debug state during loads and stores

The PE can enter Debug state during instructions that perform a sequence of memory accesses, as opposed to a single single-copy atomic access, because of a Watchpoint debug event. The effect of entering Debug state on such an instruction is the same as taking a Data Abort exception during such an instruction.

In addition, when executing in AArch64 state, the PE can enter Debug state during instructions that perform a sequence of memory accesses because of an External Debug Request debug event. The effect of entering Debug state on such an instruction is the same as taking an interrupt exception during such an instruction.

This applies to all memory types.

H2.3.6 Entering Debug state and Software Step

When Software Step is active, a debug event that causes entry to Debug state behaves like an exception taken to an Exception level above the debug target Exception level. That is:

- If the instruction that is stepped generates a synchronous debug event that causes entry to Debug state, or an asynchronous debug event is taken before the step completes, the PE enters Debug state with DSPSR.SS set to 1.
- A pending Halting debug event or an asynchronous debug event can be taken after the step has completed. In this case the PE enters Debug state with DSPSR.SS set to 0.

In addition:

- If the instruction that is stepped generates an exception trapped by an Exception Catch debug event, the PE enters Debug state at the exception vector with DSPSR.SS set to 0. This is because PSTATE.SS is set to 0 by taking the exception.
- If the PE is reset, PSTATE.SS is reset to 0. If the following debug events are enabled, and halting is allowed, the PE enters Debug state with DSPSR.SS set to 0:
  - Reset Catch debug event at the reset Exception level.
  - Exception Catch debug event at the reset Exception level.
  - Halting Step debug event.
- If Halting Step is also active, then Halting Step and Software Step operate in parallel and can both become active-pending. In this case Halting step has a higher priority than Software step. This means that the PE enters Debug state and DSPSR.SS is set to 0.

H2.3.7 Pseudocode description of entering Debug state

The DebugHalt constants are described in shared/debug/halting/DebugHalt on page J1-11369 in the pseudocode. The UpdateEDSCRFields() and Halt() functions are described in Chapter J1 Armv8 Pseudocode.
**H2.4 Behavior in Debug state**

Instructions are executed in Debug state from the Instruction Transfer Register, ITR. The debugger controls which instructions are executed in Debug state by writing the instructions to the External Debug Instruction Transfer register, EDITR. The Execution state of the PE determines which instruction set is executed:

- If the PE is in AArch64 state it executes A64 instructions.
- If the PE is in AArch32 state it executes T32 instructions:
  - For a 32-bit T32 instruction, EDITR[15:0] specifies the first halfword and EDITR[31:16] specifies the second halfword.
  - For a 16-bit T32 instruction, EDITR[15:0] contains the instruction and EDITR[31:16] is ignored. All 16-bit T32 instructions are UNPREDICTABLE in Debug state.

The PE does not execute A32 instructions in Debug state.

Some instructions are available only in Debug state. See *Debug state operations, DCPS, DRPS, MRS, MSR on page H2-10262*. In Non-debug state these instructions are UNDEFINED.

The following sections describe behavior in Debug state:

- **PSTATE in Debug state**
- **Executing instructions in Debug state** on page H2-10243.
- **Decode tables** on page H2-10256.
- **Security in Debug state** on page H2-10261.
- **Privilege in Debug state** on page H2-10262.
- **Debug state operations, DCPS, DRPS, MRS, MSR** on page H2-10262.
- **Exceptions in Debug state** on page H2-10265.
- **Accessing registers in Debug state** on page H2-10267.
- **Accessing memory in Debug state** on page H2-10270.

This section specifies the CONSTRAINED UNPREDICTABLE behaviors that apply in Debug state, but see *Changing the value of EDECR.SS when not in Debug state on page H3-10283* for a change in Non-debug state that causes CONSTRAINED UNPREDICTABLE behavior.

**H2.4.1 PSTATE in Debug state**

**PSTATE**. {N, Z, C, V, IT, GE, SS, ALLINT, SSBS, BTYPE, D, A, I, F, T} are all ignored in Debug state:

- There are no conditional instructions in Debug state.
- In AArch32 state, the PE executes only T32 instructions and **PSTATE.IT** is ignored.
- Software step is inactive.
- Asynchronous exceptions and debug events are ignored.
- If **FEAT_SSBS** is implemented, then hardware is permitted to load or store speculatively, regardless of the value of **PSTATE.SSBS**.
- In AArch64 state, **PSTATE.BTYPE** is treated as 0b00.

Instructions executed in Debug state indirectly read **PSTATE**. {TCO, UAO, PAN, IL, E, M, nRW, EL, SP} as they would in Non-debug state.

[Note] **PSTATE.DIT** is not guaranteed to have any effect in Debug state.

In Debug state:

- **PSTATE.PAN** is set to 1 by:
  - A **DCPS** instruction to EL1 using AArch64 if **SCTLR_EL1.SPAN == 0**.
  - A **DCPS** instruction to EL2 using AArch64 if **SCTLR_EL2.SPAN == 0**.
- **PSTATE.UAO** is set to 0 by a **DCPS** instruction to AArch64 state.
- **PSTATE.TCO** is set to 1 by a **DCPS** instruction to AArch64 state.
**PSTATE** can also be changed by taking exceptions in Debug state, and by the execution of **DCPS** and **DRPS** instructions.

When **FEAT_MTE** is implemented, if Memory-access mode is enabled and **PSTATE.TCO** is 0, reads and writes to the external debug interface DTR registers are **CONSTRAINED UNPREDICTABLE**, with the following permitted behaviors:

- The PE behaves as if **PSTATE.TCO** is 0. That is, the load or store operation performs the tag check if required.
- The PE behaves as if **PSTATE.TCO** is 1. That is, the load or store operation does not perform the tag check.

For more information, see Chapter D9 The Memory Tagging Extension.

**H2.4.2 Executing instructions in Debug state**

The instructions executed in Debug state must be either A64 instructions or T32 instructions, depending on the current Execution state.

Each instruction falls into one of the following groups:

- **Debug state instructions.** These are instructions that are changed in Debug state. See *A64 instructions that are changed in Debug state* and *T32 instructions that are changed in Debug state* on page H2-10251.
- **Instructions that are unchanged in Debug state.** See *A64 instructions that are unchanged in Debug state* on page H2-10244 and *T32 instructions that are unchanged in Debug state* on page H2-10252.
- **Instructions that are UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Debug state.** See *A64 instructions that are CONSTRAINED UNPREDICTABLE in Debug state* on page H2-10247 and *T32 instructions that are CONSTRAINED UNPREDICTABLE in Debug state* on page H2-10254.

All T32 instructions are treated as unconditional, regardless of **PSTATE.IT**. See *PSTATE in Debug state* on page H2-10242.

If **EDSCR.SDD** == 1 then an instruction executed in Non-secure state cannot cause entry into Secure state. See *Security in Debug state* on page H2-10261.

**Executing A64 instructions in Debug state**

The following sections describe the behavior of the A64 instructions in Debug state:

- *A64 instructions that are changed in Debug state.*
- *A64 instructions that are unchanged in Debug state* on page H2-10244.
- *A64 instructions that are CONSTRAINED UNPREDICTABLE in Debug state* on page H2-10247.

**A64 instructions that are changed in Debug state**

The following A64 instructions are defined in Debug state, but are **UNDEFINED** in Non-debug state:

- **DCPS.**
  
  Note
  
  DCPS can be **UNDEFINED** in certain conditions in Debug state. See *DCPS<n>* on page H2-10262.

- **DRPS.**
- **MRS (DLR_EL0), MRS (DSPSR_EL0), MSR (DLR_EL0), MSR (DSPSR_EL0)**
- **When FEAT_SVE is implemented, CMPNE (immediate) with byte element size.**

  Note

  In Debug state, CMPNE (immediate) with byte element size sets **DLR_EL0** and **DSPSR_EL0** to **UNKNOWN** values. However, the instruction is unchanged with respect to the SVE vector and SVE predicate source and destination registers.

  For more information, see *Debug state operations, DCPS, DRPS, MRS, MSR* on page H2-10262.
A64 instructions that are unchanged in Debug state

The following list shows the instructions that are unchanged in Debug state:

Any instruction that is undefined in Non-debug state

This list of instructions excludes:

- Any instruction listed in A64 instructions that are changed in Debug state on page H2-10243.
- Any instruction listed in A64 instructions that are constrained unpredictable in Debug state on page H2-10247 that is undefined because an enable or disable bit is not RES0 or RES1.

Instructions that move System or Special-purpose registers to or from a general-purpose register

This list of instructions:

- Includes the instructions to transfer a general-purpose register to or from the DTR, which can be executed at any Exception level.
- Excludes PSTATE access instructions.

These instructions are:

- MRS <special_reg>, MSR <special_reg>.

Note

This does not include NZCV, DAIF, DAIFSet, DAIFClr, SPSel, CurrentEL, PAN, UAO, DIT, and TCO.

- MRS <system_reg>, MSR <system_reg>.

Floating-point moves between a SIMD&FP register and a general-purpose register

These instructions are:

- FMOV (between a general-purpose register and a half-precision register).
- FMOV (between a general-purpose register and a single-precision register).
- FMOV (between a general-purpose register and a double-precision register).
- FMOV (between a general-purpose register and a SIMD element).

SIMD moves between a SIMD&FP register and a general-purpose register

These instructions are:

- INS (from a general-purpose register to a SIMD element).
- UMOV (from a SIMD element to a general-purpose register).

SVE instructions

When FEAT_SVE is implemented, these instructions are:

- CPY.
- DUP (scalar).
- EXT.
- INSR (scalar).
- PTRUE with ALL constraint and byte element size.
- RDFFR (unpredicated).
- RDVL.
- WRFFR.

Barriers

These instructions are:

- DMB.
- DSB.
- ISB.
- CSDB.
- SSBB.
- PSSBB.
• When FEAT_SB is implemented, SB.
• When FEAT_SPE is implemented, PSB CSYNC.
• When FEAT_TRBE is implemented, TSB CSYNC.
• When FEAT_RAS is implemented, ESB.

Memory access instructions at various access sizes

The following constraints apply:
• General purpose-registers only.
• One of the following addressing modes:
  — Unscaled (9-bit signed) immediate offset.
  — Immediate (9-bit signed) post-indexed.
  — Immediate (9-bit signed) pre-indexed.
  — Unprivileged (9-bit signed).
• Not literal.
• One of the following types:
  — (Single) register.
  — Exclusive.
  — Exclusive pair.
  — Acquire/Release.
  — Acquire/Release Exclusive.
  — Acquire/Release Exclusive pair.
• 32-bit and 64-bit target register variants.

These instructions are:
• LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW (immediate, not literal).
• LDUR, LDURB, LDURH, LDURSB, LDURSH, LDURSW (immediate).
• LDTR, LDTRB, LDTRH, LDTRSB, LDTRSH, LDTRSW (immediate).
• LDAR, LDARB, LDARH, LDXRB, LDXRH, LDAXR, LDAXRB, LDAXRH.
• LDXP, LDAXP.
• STR, STRB, STRH (immediate).
• STUR, STURB, STURH (immediate).
• STTR, STTRB, STTRH (immediate).
• STL, STLAR, STLARH, STXR, STXRB, STXRH, STLXR, STLXRB, STLXRH.
• STLX, STLXRB.
• When FEAT_LOR is implemented:
  — LDLAR, LDLARB, LDLARH.
  — STLLR, STLLRB, STLLRH.
• When FEAT_LSE is implemented:
  — CAS, CASB, CASH, CASP.
  — SWP, SWPB, SWPH.
  — LDADD, LDADDB, LDADDB.
  — LDCLR, LDCLB, LDCLRH.
  — LDEOR, LDEORB, LDEORH.
  — LDSET, LDSETB, LDSETBH.
  — LDSMAX, LDSMAXB, LDSMAXH.
  — LDSMIN, LDSMINB, LDSMINH.
  — LDUMAX, LDUMAXB, LDUMAXH.
  — LDUMIN, LDUMINB, LDUMINH.
  — STADD, STADDB, STADDH.
  — STCLR, STCLB, STCLRH.
  — STEOR, STEORB, STEORH.
  — STSET, STSETB, STSETH.
Move immediate to general-purpose register

These instructions are:
- MOVZ, MOVN, MOVK (immediate).
- MOV (between a general-purpose register and the stack pointer).

System instructions, Send Event, NOP, Clear Exclusive, and Prediction

In this context, the System instructions are the Cache maintenance instructions, TLB maintenance instructions, Address translation instructions, and the prediction restriction instructions.

These instructions are:
- IC.
- DC.
- TLBI.
- AT.
- SEV, SEVL.
- NOP.
- CLRdX.
- CFP.
- CPP.
- DVP.

Basic pointer authentication instructions

When FEAT_PAuth is implemented, these instructions are:
- AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIAZ.
- AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIBZ.
- AUTDA, AUTDAZ.
- AUTDB, AUTDBZ.
- PACIA, PACIA1716, PACIASP, PACIAZ, PACIZ.
- PACIB, PACIB1716, PACIBSP, PACIBZ, PACIBZ.
- PACDA, PACDAZ.
- PACDB, PACDBZ.
- PACCA.
- XPACD, XPA I, XPA CRI.

Memory Tagging Extension instructions

These instructions are:
- When FEAT_MTE is implemented:
  - ADDG.
  - SUBG.
H2.4 Behavior in Debug state

A64 instructions that are CONSTRAINED UNPREDICTABLE in Debug state

This subsection describes all instructions that are not listed in either:

- A64 instructions that are changed in Debug state on page H2-10243.
- A64 instructions that are unchanged in Debug state on page H2-10244.

These instructions are CONSTRAINED UNPREDICTABLE in Debug state. In general, the permissible behaviors are:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- If the instruction reads the PC or PSTATE, it uses an UNKNOWN value.
- If the instruction modifies the PC or PSTATE, other than by advancing the PC to the sequentially next instruction, it sets DLR_EL0 and DSPSR_EL0 to UNKNOWN values.
- If the instruction is similar to a Debug state instruction, it executes as that Debug state instruction.
- The instruction has the same behavior as in Non-debug state.

The following list shows the permissible behaviors for A64 instruction in Debug state. An instruction might appear multiple times in the list, in which case the choice of permissible behaviors is any of those listed. An example of this is COMP.

Exception-generating instructions

These instructions are:

- SVC.
- HVC.
- SMC.
- BRK.
- HLT.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- SVC behaves as DCP51.
- HVC behaves as DCP52.
- SMC behaves as DCP53.
- They generate the exception that the instruction would generate in Non-debug state. The exception is taken as described in Exceptions in Debug state on page H2-10265.
Note

SMC must not generate a Secure Monitor Call exception from Non-secure state if EDSR.SDD is set to 1.

Instructions that explicitly write to the PC (branches)

These instructions are:
• B, B.cond, BL, BLR, BR, CBZ, CBNZ, RET, TBZ, TBNZ.
• When FEAT_HBC is implemented, BC.cond.

These instructions behave in one of the following ways:
• They are UNDEFINED.
• They execute as a NOP.
• They execute as in Non-debug state without branching and set DSPSR_EL0 and DLR_EL0 to UNKNOWN values.

Exception return and related instructions

This instruction is:
• ERET.

This instruction behaves in one of the following ways:
• It is UNDEFINED.
• It executes as a NOP.
• It executes as in Non-debug state without branching. It sets DSPSR_EL0 and DLR_EL0 to UNKNOWN values, and either:
  — Executes the DRPS operation instead of performing an exception return, using UNKNOWN SPSR values.
  — Does not change the Exception level.

Instructions that request entry to a low-power state

These instructions are:
• WFE, WFI.
• When FEAT_WFxT is implemented, WFET, WFIT.

These instructions behave in one of the following ways:
• They are UNDEFINED.
• They execute as a NOP.
• They generate a synchronous exception if the corresponding instruction would be trapped in Non-debug state. See Configurable instruction controls on page D1-4665.
• A WFE instruction clears the Event register if it is set.

Note

This means that these instructions must not suspend execution.

Instructions that read the PC

These instructions are:
• LDR (literal), LDRSW (literal).
• ADR, ADRP.
• PRFM (literal).

These instructions behave in one of the following ways:
• They are UNDEFINED.
• They execute as a NOP.
• They execute as in Non-debug state, using an UNKNOWN value for the PC operand.
Instructions that explicitly modify `PSTATE`, other than `DCPS` and `DRPS`:

These instructions are:

- ADDS, SUBS, ADCS, SBCS, ANDS, BICS, COMN, COMP.
- FCMPL, FCMPU, FCMPO, FCMPOE.
- MSR DAIFSet (immediate), MSR DAIFClr (immediate), MSR SPSel (immediate).
- MSR N2CV (register), MSR DAIF (register), MSR SPSel (register).
- When `FEAT_PAN` is implemented:
  - MSR PAN (immediate).
  - MSR PAN (register).
- When `FEAT_UAO` is implemented:
  - MSR UAO (immediate).
  - MSR UAO (register).
- When `FEAT_FlagM` is implemented:
  - CFINV.
  - RMIF.
  - SETF8.
  - SETF16.
- When `FEAT_DIT` is implemented, MSR DIT.
- When `FEAT_FlagM2` is implemented:
  - AXFLAG.
  - XAFLAG.
- When `FEAT_MTE` is implemented, MSR TC0.
- When `FEAT_RNG` is implemented:
  - MRS RNDR.
  - MRS RNDRS.
- When `FEAT_NMI` is implemented:
  - MSR ALLINT (immediate).
  - MSR ALLINT (register).
- When `FEAT_MOPS` is implemented:
  - CPYFP, CPYFPN, CPYFPR, CPYFPRTN, CPYFPRTRN, CPYFPT, CPYFPTN, CPYFPTR, CPYFPTRW, CPYFPTWN, CPYFPWPWTN, CPYFPWTRN, CPYFPWTRWN.
  - CPYP, CPYPN, CPYPRT, CPYPRTRN, CPYPRTRWN, CPYPRPT, CPYPRPTN, CPYPRTR, CPYPRTRN, CPYPRTRWN, CPYPRWT, CPYPRWTN.
  - SETP, SETPN, SETPT, SETPTN.
  - SETGP, SETGPN, SETGPT, SETGPTN.
- When `FEAT_SVE` is implemented:
  - ANDS
  - BICS
  - BRKAS
  - BRK8S
  - BRK8N
  - BRKPAS
  - BRKPBS
  - CMPEQ
  - CMPEGE
  - CMPEG
  - CMPI
  - CMPIE
  - CMPL0
  - CMPLS
H2.4 Behavior in Debug state

- CMPLT
- CMPNE
- CTERMEQ
- CTERMNE
- EORS
- MATCH
- NANDS
- NMATCH
- NORS
- ORNS
- ORRS
- PFIRST
- PNEXT
- PTEST
- PROES
- RDFFRS
- WHILEGE
- WHILEGT
- WHILEHI
- WHILEHS
- WHILELE
- WHILELO
- WHILELS
- WHILELT
- WHILERW
- WHILERW

These instructions behave in one of the following ways:
- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state, setting DSPSR_EL0 and DLR_EL0 to UNKNOWN values.

Instructions that read PSTATE.{N, Z, C, V} or other PSTATE fields

These instructions are:
- CSEL, CSINC, CSINV, CSNEG, CONN, COMP, FCSER, FCOMP, FCOMPE.
- ADC, ADCS, SBC, SBCS.
- MRS NZCV, MRS DAIF, MRS SPSel, MRS CurrentEl.
- When FEAT_PAN is implemented, MRS PAN.
- When FEAT_UAO is implemented, MRS UAO.
- When FEAT_FlagM is implemented, CFINV.
- When FEAT_FlagM2 is implemented, AXFLAG, XAFLAG.
- When FEAT_DIT is implemented, MRS DIT.
- When FEAT_MTE is implemented, MRS TCO.
- When FEAT_NMI is implemented, MRS ALLINT.
- When FEAT_MOPS is implemented:
  - CPYFM, CPYFMN, CPYFMRN, CPYFMRT, CPYFMRTN, CPYFMMR, CPYFMMRN, CPYFMMRT, CPYFMMRTN, CPYFMMRN, CPYFMMRT.
  - CPYFEN, CPYFRN, CPYFERT, CPYFERTRN, CPYFERTWN, CPYFET, CPYFETN, CPYFETR, CPYFETRN, CPYFETWN, CPYFETFR, CPYFETRW, CPYFETW, CPYFETRN, CPYFETWN.
  - CPYM, CPYMN, CPYMRT, CPYMRTN, CPYMRTWN, CPYMRT, CPYMRTN, CPYMRTWN, CPYMRT, CPYMRTN, CPYMRTWN, CPYMRT, CPYMRTN, CPYMRTWN, CPYMRT, CPYMRTN, CPYMRTWN, CPYMRT, CPYMRTN, CPYMRTWN, CPYMRT, CPYMRTN, CPYMRTWN.
When FEAT_SVE is implemented, all SVE instructions that use a predicate.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as aNOP.
- They execute as in Non-debug state:
  - For the conditional operations and those that use Condition flags as an input, these instructions use UNKNOWN values for the Condition flag.
  - For the MRS instruction, they return an UNKNOWN value.

Hint instructions

When FEAT_DGH is implemented, this instruction is:

- DGH.

This instruction behaves in one of the following ways:

- It executes as aNOP.
- It executes as in Non-debug state.

All other instructions

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as aNOP.
- They execute as in Non-debug state.

--- Note ---

This includes instructions defined as UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Non-debug state. These instructions are UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Debug state.

Executing T32 instructions in Debug state

The following sections describe the behavior of the T32 instructions in Debug state:

- T32 instructions that are changed in Debug state.
- T32 instructions that are unchanged in Debug state on page H2-10252.
- T32 instructions that are CONSTRAINED UNPREDICTABLE in Debug state on page H2-10254.

T32 instructions that are changed in Debug state

The following T32 instructions are defined in Debug state, but are UNDEFINED in Non-debug state:

- DCPS.

--- Note ---

DCPS can be UNDEFINED in certain conditions in Debug state. See DCPS<n> on page H2-10262.

- MRC p15,3,<Rt>,c4,c5,0 (DSPSR).
- MCR p15,3,<Rt>,c4,c5,0 (DSPSR).
- MRC p15,3,<Rt>,c4,c5,1 (DLR).
- MCR p15,3,<Rt>,c4,c5,1 (DLR).

In addition, ERET executes the DRPS operation in Debug state.
For more information, see Debug state operations, DCPS, DRPS, MRS, MSR on page H2-10262.

**T32 instructions that are unchanged in Debug state**

The following list shows the instructions that are unchanged in Debug state. Any T32 instruction that uses the PC or APSR.\{N, Z, C, V\} as the source or destination register is not included in the list. Moreover, the list includes only the 32-bit T32 encodings.

**Any instruction that is undefined in Non-debug state**

The list of instructions:

- Excludes any instruction listed in T32 instructions that are changed in Debug state on page H2-10251.
- Excludes any instruction listed in T32 instructions that are CONSTRAINED UNPREDICTABLE in Debug state on page H2-10254 that is undefined because an enable or disable bit is not RES0 or RES1.

**Instructions that move System or Special-purpose registers to or from a general-purpose register**

The list of instructions:

- Includes the instructions to transfer a general-purpose register to or from the DTR, which can be executed at any Exception level.
- Excludes APSR and CPSR access instructions.
- Excludes instructions for accessing banked registers for the current mode.

These instructions are:

- \textsf{MRS <banked_reg>, MSR <banked_reg>}.  

\textbf{Note}

This does not apply to cases which are UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Non-debug state in the current mode.

- \textsf{MRC, MCR}.

\textbf{Note}

This includes all allocated System registers in the (\textsf{coproc}==0b111x) encoding space other than an \textsf{MRC} move to APSR\_nzcv.

- \textsf{MRS SPSR, MSR SPSR\_{fsxc} (register)}.  

- \textsf{VMRS <vfp\_system\_reg>, VMSR <vfp\_system\_reg>}.  

\textbf{Note}

This includes all allocated Advanced SIMD and floating-point System registers, other than an \textsf{VMRS} move to APSR\_nzcv.

**Floating-point moves between a SIMD&FP register and a general-purpose register**

These instructions are:

- \textsf{VMOV (between a general-purpose register and a single-precision register)}.  

- \textsf{VMOV (between a general-purpose register and a doubleword floating-point register)}.  

**SIMD moves between a SIMD&FP register and a general-purpose register**

These instructions are:

- \textsf{VMOV (between a general-purpose register and a scalar)}.  

**Barriers**

These instructions are:

- \textsf{CSDB}.  

- \textsf{DMB}.  

- \textsf{DSB}.  

- \textsf{ISB}.  


Memory access instructions at various access sizes

The following constraints apply:

- General purpose registers only.
- One of the following addressing modes:
  - Immediate (8-bit or 12-bit) offset.
  - Immediate (8-bit) post-indexed.
  - Immediate (8-bit) pre-indexed.
  - Unprivileged (8-bit).
- Not literal.
- One of the following types:
  - (Single) register.
  - Dual.
  - Exclusive.
  - Exclusive doubleword.
  - Acquire/Release.
  - Acquire/Release Exclusive.
  - Acquire/Release Exclusive doubleword.

These instructions are:

- LDRT, LDRBT, LDRHT, LDRSBT, LDRSHT (immediate).
- LDREX, LDREXB, LDRH, LDAB, LDAH, LDEX, LDAEXB, LDAEXH.
- LDREXD, LDAEXD.
- STR.W, STRB.W, STRH.W, STRD (immediate).
- STRT, STRBT, STRHT (immediate).
- STREX, STREXB, STREXH, STL, STLH, STLEX, STLEXB, STLEXH.
- STREXD, STLEXD.

Move to general-purpose register

These instructions are:

- MOVW, MOVT (immediate).

System instructions, Send Event, NOP, and Clear Exclusive

The System instructions are Cache maintenance instructions, TLB maintenance instructions, and Address translation instructions. These are encoded in the (coproc == 0b1111) System register encoding space.

These instructions are:

- ICIALLU, ICIALLUIS, ICIMVAU.
- DCCIMVAC, DCCISW, DCCMVAC, DCCMVAU, DCCSW, DCMVAC, DCISW.
- TLBIAL, TLBIALLH, TLBIALLHIS, TLBIALLIS, TLBIALLNNSNI, TLBIALLNNSNH, TLBIALNSHIS, TLBIALSID, TLBIALSIDS, TLBIALPAS2, TLBIALPAS2IS, TLBIALPAS2L, TLBIALPAS2LIS, TLBIMV, TLBIMVAAI, TLBIMVAAIS, TLBIMVAAL, TLBIMVAALIS, TLBIMVAH, TLBIMVAHIS, TLBIMVAIS, TLBIMVAL, TLBIMVALH, TLBIMVALHIS, TLBIMVALIS.
- ATS12NSOPR, ATS12NSOPW, ATS12NSOUR, ATS12NSOUW, ATS1CPR, ATS1CPUW, ATS1HR, ATS1HW.
- BPIALL, BPIALLIS, BPIMVA.
- SEV.W, SEVL.W.
**T32 instructions that are CONSTRAINED UNPREDICTABLE in Debug state**

This subsection describes all instruction not listed in either:
- **T32 instructions that are changed in Debug state** on page H2-10251.
- **T32 instructions that are unchanged in Debug state** on page H2-10252.

These instructions are CONSTRAINED UNPREDICTABLE in Debug state. In general, the permissible behaviors are:
- The instruction generates an Undefined Instruction exception.
- The instruction executes as a NOP.
- If the instruction reads the PC or PSTATE, it uses an UNKNOWN value.
- If the instruction modifies the PC or PSTATE, other than by advancing the PC to the sequentially next instruction, it sets DLR and DSPSR to UNKNOWN values.
- If the instruction is similar to a Debug state instruction, it executes as that Debug state instruction.
- The instruction has the same behavior as in Non-debug state.

The following list shows the permissible behaviors for T32 instruction in Debug state. An instruction might appear multiple times in the list, in which case the choice of permissible behaviors is any of those listed.

**Exception-generating instructions**

These instructions are:
- SVC.
- HVC.
- SMC.
- UDF.
- BKPT.
- HLT.

These instructions behave in one of the following ways:
- They are UNDEFINED.
- They execute as a NOP.
- SVC behaves as DCP51.
- HVC behaves as DCP52.
- SMC behaves as DCP53.
- They generate the exception the instruction would generate in Non-debug state. The exception is taken as described in **Exceptions in Debug state** on page H2-10265.

**Note**

SMC must not generate a Secure Monitor Call exception from Non-secure state if EDSR.SDD is set to 1.

**Instructions that explicitly write to the PC (branches)**

These instructions are:
- B, B (conditional), CBZ, CBNZ BL.
- BX, BLX (register or immediate).
- BXJ, TB8, TBH.
- MOV pc and related instructions.
- LDR pc, LDM (with a register list includes the PC), POP (with a register list that includes the PC).

These instructions behave in one of the following ways:
- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state without branching and set DSPSR and DLR to UNKNOWN values.
Exception return and related instructions, other than ERET

These instructions are:
- SRS, RFE, SUBS pc, lr, and related instructions.

These instructions behave in one of the following ways:
- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state without branching, setting DLR and DSPSR to UNKNOWN values, and either:
  - Execute the DRPS operation instead of performing an exception return, using UNKNOWN SPSR values.
  - Not changing Exception level or PE mode.

Instructions that request entry to a low-power state

These instructions are:
- WFE, WFI, WFET, WFIT

These instructions behave in one of the following ways:
- They are UNDEFINED.
- They execute as a NOP.
- They generate a synchronous exception if the corresponding instruction would be trapped in Non-debug state. See Configurable instruction enables and disables, and trap controls on page G1-9001.
- A WFE instruction is permitted to clear the Event register if it is set.

--- Note ---
This means that these instructions must not suspend execution.

Instructions that read the PC

These instructions are:
- LDR (literal), LDRB (literal), LDRH (literal), LDRSB (literal), LDRSH (literal).
- ADR, ADRL, ADRH.
- PLD (literal), PLI (literal).

These instructions behave in one of the following ways:
- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state using an UNKNOWN value for the PC operand.

Instructions that explicitly modify PSTATE, other than DCPS and ERET

These instructions are:
- CMP, TST, TEQ, CMN.
- <op>=5.
- MRC p14,0,APSR_nzcv,c0,c1,0 (accessing DBGDSCRint).
- CPS, SETEND, IT.
- MSR CPSR (immediate), MSR CPSR (register), MSR APSR (immediate), MSR APSR (register).
- VMRS APSR_nzcv, FPSCR.
- QADD, QADD, QSUB, QSUB.
- SMLABB, SMLABT, SMLATB, SMLATT, SMLAD, SMLAWB, SMLAWT, SMLSD, SMLAD.
- SSAT, SSAT16, USAT, USAT16.
- SADD, SADD, SADD, SASX, SSAX, SSUB, SSUB, SSUB8, SSUB16.
- UADD, UADD, UADD, USAX, USAUB, USUN8, USUB16.
- When FEAT_PAN is implemented, SETPAN.
These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state, setting DSPSR_EL0 and DLR_EL0 to UNKNOWN values.

**Instructions that read PSTATE.{N, Z, C, V} or other PSTATE fields**

These instructions are:

- SEL, VSEL.
- ADC, SBC, all instructions with an RRX shift.
- MRS CPSR.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state:
  - For the conditional operations and those using the PSTATE.C flag as an input, these instructions use an UNKNOWN value for the Condition flag.
  - For the MRS instruction, they return an UNKNOWN value.

**All other instructions**

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They have the same behavior as in Non-debug state.

--- **Note** ---

This includes instructions defined as UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Non-debug state. These instructions are CONSTRAINED UNPREDICTABLE in Debug state. This includes some T32 instructions that specify R15 as a destination or source register.

**Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors** describes the CONSTRAINED UNPREDICTABLE behavior for these instructions. In Debug state these CONSTRAINED UNPREDICTABLE choices are further restricted:

- Instructions that specify R15 as a destination register:
  - Are not permitted to branch, because the architecture does not define a branch operation in Debug state.
  - Might set DLR and DSPSR to UNKNOWN values.
  - Might have any of the other permitted behaviors.

- Instructions that specify R15 as a source operand:
  - Cannot use PC + offset, because there is no architecturally-defined PC in Debug state.
  - Might have any of the other permitted behaviors, including using an UNKNOWN value.

---

**H2.4.3 Decode tables**

The syntax in the tables is defined as follows:

- **1** The bit has a fixed value of 1.
- **0** The bit has a fixed value of 0.
- **!**\(=\) The field has any value other than the value or values specified. The field might be an encoding field in the instruction whose value is supplied by the debugger.
Note

The instruction encodings in Chapter C6 A64 Base Instruction Descriptions and Chapter F5 T32 and A32 Base Instruction Set Instruction Descriptions might show these bits as (0) or (1). A debugger must set these bits to 0 or 1, as appropriate.

Any other value indicates an encoding field in the instruction whose value is supplied by the debugger. Some values might be reserved or UNDEFINED, in which case the instruction is UNDEFINED or CONSTRAINED UNPREDICTABLE in Debug state, as it is in Non-debug state.

For more information about the instruction encodings, see:
• Chapter C6 A64 Base Instruction Descriptions.
• Chapter F5 T32 and A32 Base Instruction Set Instruction Descriptions.

For information about the syntax used in Table H2-2, Table H2-3, Table H2-4 on page H2-10257, and Table H2-5 on page H2-10259, see:
• Common syntax terms on page C1-228.
• Assembler symbols on page F1-7225.

Table H2-2 shows the A64 instructions that are modified in Debug state. For details of how these are packed in the EDITR, see the register description.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCPS&lt;opt&gt;</td>
<td></td>
</tr>
<tr>
<td>MRS</td>
<td>MSR accessing DSPSR_EL0</td>
</tr>
<tr>
<td>MRS</td>
<td>MSR accessing DLR_EL0</td>
</tr>
<tr>
<td>DRPS</td>
<td></td>
</tr>
<tr>
<td>CMPNE (immediate)</td>
<td></td>
</tr>
</tbody>
</table>

Table H2-3 shows the T32 instructions that are modified in Debug state, with the first halfword on the left side and the second halfword on the right side. For details of how these are packed in the EDITR, see the register description.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRC</td>
<td>MCR accessing DSPSR</td>
</tr>
<tr>
<td>MRC</td>
<td>MCR accessing DLR</td>
</tr>
<tr>
<td>ERET</td>
<td></td>
</tr>
</tbody>
</table>

Table H2-4 on page H2-10257 lists the A64 instructions that are unchanged in Debug state, other than some unallocated and UNDEFINED instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV &lt;Rn&gt;,SP</td>
<td></td>
</tr>
</tbody>
</table>
Debug State
H2.4 Behavior in Debug state

Table H2-4 A64 instructions that are unchanged in Debug state (continued)
31302928272625242322212019181716 15 1413121110987 6 5 43210Description
0 0 1 0 0 1 0 1 0 0 0 1 1 0 0 1 1 1 1 1 0 0 000 0 0 0Pd
0 0 1 0 0 1 0 1 0 0 1 0 1 0 0 0 1 0 0 1 0 0 0Pn

RDFFR (unpredicated)

00000 WRFFR

0 0 1 0 0 1 0 1 0 0 0 1 1 0 0 0 1 1 1 0 0 0 000 0 0 0Pd

PTRUE

1 0 0 1 1 0 1 0 1 1 0 Rm

Rd

PACGA

Rd

MOVN, MOVK, MOVZ

sf !=01 1 0 0 1 0 1 hw

0 0 1 1 0 0 Rn

imm16

1 1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0 000 0 0 11111 NOP
1 1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0 001 0 L 11111 SEV, SEVL
1 1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0 001 1 1 11111 XPACLRI
1 1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0 01op2

11111 PAC(IA|IB)1716, AUT(IA|IB)1716

1 1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0 10!=010 11111 CSDB, ESB, PSB
1 1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0 100 1 0 11111If FEAT_TRBE is implemented, TSB CSYNC
1 1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0 11op2
1 1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 1 1 CRm

11111 PAC(IA|IB)(Z|SP), AUT(IA|IB)(Z|SP)

0 1 0 11111 CLREX

1 1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 1 1 option 1 op2 11111 DSB, DMB, ISB, SB, SSBB, PSSBB
1 1 0 1 0 1 0 1 0 0 0 0 1 op1

CRn

CRm

op2

Rt

IC, DC, TLBI, AT

1 1 0 1 0 1 0 1 0 0 L 1 0 op1

CRn

CRm

op2

Rt

MRS|MSR accessing System register

1 1 0 1 0 1 0 1 0 0 L 1 1 op1

!=0100

CRm

op2

Rt

MRS|MSR accessing System register

1 1 0 1 0 1 0 1 0 0 L 1 1 op1

0 1 0 0 !=0010 op2

Rt

MRS|MSR accessing Special-purpose register

Rd

PAC(IA|IB|DA|DB), AUT(IA|IB|DA|DB)

1 1 0 1 1 0 1 0 1 1 0 0 0 0 0 1 0 0 0 opc

Rn

1 1 0 1 1 0 1 0 1 1 0 0 0 0 0 1 0 0 1 opc

111 1 1 Rd

PAC(IZA|IZB|DZA|DZB), AUT(IZA|IZB|DZA|DZB)

1 1 0 1 1 0 1 0 1 1 0 0 0 0 0 1 0 1 0 0 0 op 111 1 1 Rd

XPAC(I|D)

size 0 0 1 0 0 0 o2 L 0 Rs

o0 Rt2

size 0 0 1 0 0 0 o2 L 1 Rs

o0 Rt2

Rn

Rt

LD(A|LA|X|AX)R{B|H},
ST(L|LL|X|LX)R{B|H},
CAS{A|L|AL}{B|H}

Rn

Rt

LD{A}XP, ST{L}XP, CASP{A|L|AL}

0 x 0 1 1 0 0 1 opc 0 imm9

0 0 Rn

Rt

LDAPUR(B|H|SB|SH), STLUR(B|H)

1 0 0 1 1 0 0 1 !=11 0 imm9

0 0 Rn

Rt

LDAPUR{SW}, STLUR

1 1 0 1 1 0 0 1 !=1x 0 imm9

0 0 Rn

Rt

LDAPUR, STLUR

0 x 1 1 1 0 0 0 opc 0 imm9

0 0 Rn

Rt

LDUR(B|H|SB|SH),STUR(B|H)

1 0 1 1 1 0 0 0 !=11 0 imm9

0 0 Rn

Rt

LDUR{SW},STUR

1 1 1 1 1 0 0 0 !=1x 0 imm9

0 0 Rn

Rt

LDUR,STUR

size 1 1 1 0 0 0 opc 0 imm9

1 0 Rn

Rt

LDTR{B|H|SB|SH|SW}, STTR{B|H}

P 1 Rn

size 1 1 1 0 0 0 opc 0 imm9

Rt

LDR{B|H|SB|SH|SW}, STR{B|H}

size 1 1 1 0 0 0 A R 1 Rs

0 opc

0 0 Rn

Rt

LD<op>{A|L|AL}{B|H},
ST<op>{A|L|AL}{B|H}

size 1 1 1 0 0 0 A R 1 Rs

1 0 0 0 0 0 Rn

Rt

SWP{A|L|AL}{B|H}

size 1 1 1 0 0 0 1 0 1 Rs

1 1 0 0 0 0 Rn

Rt

LDAPR{B|H}

0 1 0 0 1 1 1 0 0 0 0 imm5

0 0 0 1 1 1 Rn

Rd

INS <Vd>.<Ts>[<index>],<R><n>

0 Q 0 0 1 1 1 0 0 0 0 imm5

0 0 1 1 1 1 Rn

Rd

UMOV <R><d>,<Vn>.<Ts>[<index>]

0 0 0 1 1 1 1 0 0 0 1 0 0 1 1 op 0 0 0 0 0 0 Rn

Rd

FMOV <Sd>,<Wn>, FMOV <Wd>,<Sn>

0 0 0 1 1 1 1 0 1 1 1 0 0 1 1 op 0 0 0 0 0 0 Rn

Rd

FMOV <Hd>, <Wn>, FMOV <Wd>, <Hn>

1 0 0 1 1 1 1 0 ft 1 1 0 0 1 1 op 0 0 0 0 0 0 Rn

Rd

FMOV <Dd|Hd>,<Xn>, FMOV <Xd>,<Dn|Hn>

H2-10258

Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ARM DDI 0487I.a
ID081822


Debug State
H2.4 Behavior in Debug state

Table H2-4 A64 instructions that are unchanged in Debug state (continued)
31302928272625242322212019181716 15 1413121110987 6 5 43210Description
1 0 0 1 1 1 1 0 1 0 1 0 1 1 1 op 0 0 0 0 0 0 Rn

Rd

FMOV <Vd>.D[1],<Xn>
FMOV <Xd>,<Vn>.D[1]

1 0 0 1 0 0 0 1 1 0 uimm6

(0)(0)uimm4

Xn

Xd

ADDG <Xd|SP>, <XN|SP>, #<uimm6>, #<uimm4>

1 0 Xn

Xd

STG <Xt|SP>, [<Xn|SP>{, #<simm}], Signed offset

1 1 0 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 Xn

Xd

STGM <Xt>, [<Xt|SP>]

1 1 0 1 1 0 0 1 1 0 1 imm9

1 0 Xn

Xd

ST2G <Xt|>, [Xt|SP>{, #<simm>}] Signed offset

1 1 0 1 1 0 0 1 0 1 1 imm9

1 0 Xn

Xd

STZG <XT|SP>, [<Xn|SP{. #<simm>}] Signed offset

1 1 0 1 1 0 0 1 1 1 1 imm9

1 0 Xn

Xd

STZ2G <XT|SP>, [<Xn|SP{. #<simm>}] Signed offset

1 1 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 Xn

Xd

STZGM, <Xt>, [Xn|SP>]

0 1 1 0 1 0 0 1 0 0 simm7

Xn

Xt

STGP <xt1>, <Xt2>, [<Xn|SP>{, #<imm>}] Signed offset

1 1 0 1 1 0 0 1 0 0 1 imm9

1 1 0 1 0 0 0 1 1 0 uimm6

Xt2
op3 uimm4

Xn

Xd

SUBG <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>

0 0 Xn

Xd

LDG <Xt>, [<Xn|SP>{, #<simm>}]

1 1 0 1 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 Xn

Xd

LDGM <Xt>,[<Xn|SP>]

1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 0 0 Rn

Rt

ST64B <Xt>, [<Xn|SP> {,#0}]

1 1 1 1 1 0 0 0 0 0 1 Rs

1 0 1 0 0 0 Rn

Rt

ST64BV0 <Xs>, <Xt>, [<Xn|SP>]

1 1 1 1 1 0 0 0 0 0 1 Rs

1 0 1 1 0 0 Rn

Rt

ST64BV <Xs>, <Xt>, [<Xn|SP>]

1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 0 1 0 0 Rn

Rt

LD64B <Xt>, [<Xn|SP> {,#0}]

1 1 0 1 1 0 0 1 0 1 1 imm9

Table H2-5 on page H2-10259 lists the T32 instructions that are unchanged in Debug state, other than some
unallocated and UNDEFINED instructions. It shows the T32 instructions with the first halfword on the left side and
the second halfword on the right side.
Table H2-5 T32 instructions that are unchanged in Debug state
151413121110987 6 5 4 3 2 1 0 15141312111098 7 6 5 4 32 10 Description
1 1 1 0 1 1 000 1 0 op!=1111 !=1111

1 0 11 0 0 M 1 Vm

1 1 1 0 1 1 100 0 0 opVn

!=1111

1 0 10 N0 0 1 00 00 VMOV <Sn>,<Rt>, VMOV <Rt>,<Sn>

1 1 1 0 1 1 100 opc 0 Vd

!=1111

1 0 11 Dopc21 00 00 VMOV.<size> <Dd>[<x>],<Rt>

1 1 1 0 1 1 10Uopc 1 Vn

!=1111

1 0 11 Dopc21 00 00 VMOV.<dt> <Rt>,<Dd>[<x>]

1 1 1 0 1 1 101 1 1 opreg

!=1111

1 0 10 0 0 0 1 00 00 VMRS, VMSR

VMOV <Dm>,<Rt>,<Rt2>
VMOV <Rt>,<Rt2>,<Dm>

1 1 1 0 1 1 000 1 0 op!=1111 !=1111

1 1 1cpopc1

1 1 1 0 1 1 10opc1 opCRn

1 1 1cpopc2 1 CRm MCR|MRC accessing System registers

!=1111

CRm MCRR|MRRC accessing System registers

1 1 1 0 1 0 000 1 0 L !=1111 !=1111

Rd

imm8

LDREX, STREX

1 1 1 0 1 0 001 1 0 L !=1111 !=1111

Rt2

0 1 !=10 Rd

LDREX(B|H|D), STREX(B|H|D)

1 1 1 0 1 0 001 1 0 L !=1111 !=1111

Rt2

1 op3 Rd

LDA{EX}{B|H|D}, STL{EX}{B|H|D}

1 1 1 0 1 0 0 !=0x10 L !=1111 !=1111
!=xx0x

!=1111 imm8

LDRD, STRD

1 1 1 1 0 i 10T 1 0 0 imm4 0 imm3

!=1111 imm8

MOVW, MOVT

1 1 1 1 0 0 111 0 0 R !=1111 1 0 0 0 M1

0 0 1 M00 00 MSR <spec_reg><mode>,<Rn>

1 1 1 1 0 0 111 0 0 1 !=1111 1 0 0 0 1 1 11 0 0 0 0 00 00 MSR SPSR, <Rn>
1 1 1 1 0 0 111 0 1 0 1 1 1 1 1 0 0 0 0 0 00 0 0 0 0 00 00 NOP.W
1 1 1 1 0 0 111 0 1 0 1 1 1 1 1 0 0 0 0 0 00 0 0 0 0 01 0L SEV.W, SEVL.W
1 1 1 1 0 0 111 0 1 0 1 1 1 1 1 0 0 0 0 0 00 0 0 0 1 0op00 ESB, CSDB

ARM DDI 0487I.a
ID081822

Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2-10259


### Table H2-5 T32 instructions that are unchanged in Debug state (continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLREX</td>
<td></td>
</tr>
<tr>
<td>DSB, DMB, ISB, SSBB, PSSBB, SB</td>
<td></td>
</tr>
<tr>
<td>MRS &lt;Rd&gt;, &lt;spec_reg&gt;&lt;mode&gt;</td>
<td></td>
</tr>
<tr>
<td>MRS &lt;Rd&gt;, SPSR</td>
<td></td>
</tr>
<tr>
<td>STR{B</td>
<td>H}.W (12-bit immediate)</td>
</tr>
<tr>
<td>STR{B</td>
<td>H}{T} (8-bit immediate)</td>
</tr>
<tr>
<td>LDR{SB</td>
<td>SH</td>
</tr>
<tr>
<td>LDR{SB</td>
<td>SH</td>
</tr>
</tbody>
</table>
H2.4.4 Security in Debug state

If EL3 is implemented or the implemented Security state is Secure state, security in Debug state is governed by the Secure debug disabled flag, EDSCR.SDD.

On entry to Debug state

If entering in Secure state, EDSCR.SDD is set to 0. Otherwise EDSCR.SDD is set to the inverse of ExternalSecureInvasiveDebugEnabled(). That is:

- If ExternalSecureInvasiveDebugEnabled() == TRUE, EDSCR.SDD is set to 0.
- If ExternalSecureInvasiveDebugEnabled() == FALSE, EDSCR.SDD is set to 1.

Note

Normally, if ExternalSecureInvasiveDebugEnabled() == FALSE then halting is prohibited and it is not possible to enter Debug state from Secure state. However, because changes to the authentication signals require a Context synchronization event to guarantee their effect, there is a period during which the PE might halt even though the authentication signals prohibit halting.

In Debug state

The value of EDSCR.SDD does not change, even if ExternalSecureInvasiveDebugEnabled() changes.

Note

- DBGAUTHSTATUS_EL1.{SNID, SID, NSNID, NSID} are not frozen in Debug state.
- If EDSCR.SDD set to 1 in Debug state, then there is no means to enter Secure state from Non-secure state. In this case it is impossible for the PE to be in Secure state. This is a general principle of behavior in Debug state.

In Non-debug state

EDSCR.SDD returns the inverse of ExternalSecureInvasiveDebugEnabled(). If the authentication signals that control ExternalSecureInvasiveDebugEnabled() change, a Context synchronization event is required to guarantee their effect.

Note

- In Non-debug state, EDSCR.SDD is unaffected by the Security state of the PE.
- A Context synchronization event is also required to guarantee that changes in the authentication signals are visible in DBGAUTHSTATUS_EL1.{SNID, SID, NSNID, NSID}.

If EL3 is not implemented and the implemented Security state is Non-Secure state, EDSCR.SDD is RES1.
H2.4.5 Privilege in Debug state

The only additional privileges offered to Debug state are:

• The privilege to execute *Debug state operations, DCPS, DRPS, MRS, MSR.*
• The privilege to execute DTR access instructions regardless of the Exception level and traps.

The DTR access instructions can be executed at any Exception level, including EL0, regardless of any control register settings that might force these instructions to be undefined or trapped in Non-debug state. These instruction are:

• The *MRS* and *MSR* instructions that access DBGDTR_EL0, DBGDTRTX_EL0, and DBGDTRRX_EL0 in AArch64 state.
• The *MRC* and *MCR* instructions that access DBGDTRTXint and DBGDTRRXint in AArch32 state.

All other instructions operate with the privilege determined by the current Exception level and security state. This applies to all Special-purpose and System registers accesses, memory accesses, and undefined instructions, and includes generating exceptions when the System registers trap or disable an instruction.

H2.4.6 Debug state operations, DCPS, DRPS, MRS, MSR

The architecture defines operations to change between Exception levels in Debug state. These operations can also change the mode at the current Exception level.

**DCPS<\text{n}>**

Executing a *DCPS<\text{n}>* instruction in Debug state moves the PE to a higher Exception level or to a specific mode at the current Exception level.

If the *DCPS<\text{n}>* instruction is executed in AArch32 state and the target Exception level is using AArch64:

• The current instruction set switches from T32 to A64.
• The effect on registers that are not visible or only partially visible in AArch32 state is the same as for system calls in Non-debug state. See *Execution state* on page D1-4632.

Otherwise, the instruction set state does not change.

If the target Exception level is the same as the current Exception level, then the PE does not change Exception level. However, the PE might change mode.

The effect on endianness is the same as for exceptions and exception returns in Non-debug state:

• In AArch64 state the current endianness is determined by the value of SCTLR_ELx.EE for the target Exception level.
• In AArch32 state the current endianness is determined by the value of SCTLR.EE or HSCTLR.EE for the target Exception level.

The *DCPS<\text{n}>* instructions are:

**In AArch64 state**

• DCPS1
• DCPS2
• DCPS3

**In AArch32 state, in the T32 instruction set only**

• DCPS1
• DCPS2
• DCPS3

The DCPS instructions are undefined in Non-debug state.

Table H2-6 on page H2-10263 shows the target of the instruction. In Table H2-6 on page H2-10263, the entries have the following meaning:

**EL1h/Svc**  
This means that the target is:

• EL1h if EL1 is using AArch64.
H2.4 Behavior in Debug state

- EL1 and Supervisor mode if EL1 is using AArch32.

EL2h/Hyp

This means that the target is:
- EL2h if EL2 is using AArch64.
- EL2 and Hyp mode if EL2 is using AArch32.

EL3h/Monitor

This means that the target is:
- EL3h if EL3 is using AArch64.
- EL3 and Monitor mode if EL3 is using AArch32.

Svc

Secure Supervisor mode, in EL3 using AArch32.

Monitor

Secure Monitor mode, in EL3 using AArch32.

Table H2-6 Target for DCPS instructions in Debug state

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Target when DCPS instruction executed at stated Exception level:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EL0</td>
</tr>
<tr>
<td>DCPS1</td>
<td>EL1h/Svc</td>
</tr>
<tr>
<td>DCPS2</td>
<td>EL2h/Hyp</td>
</tr>
<tr>
<td>DCPS3</td>
<td>EL3h/Monitor</td>
</tr>
</tbody>
</table>

In AArch32 Monitor mode, DCPS1 and DCPS3 clear SCR.NS to 0.

Note

In AArch64 state, at EL3, DCPS<n> does not change SCR_EL3.NS.

However:
- **DCPS1** is **UNDEFINED** at EL0 if either:
  - EL2 is implemented and enabled in the current Security state, and is using AArch64 and HCR_EL2.TGE == 1.
  - In Non-secure state, EL2 is implemented and using AArch32 and HCR.TGE == 1.
- **DCPS2** is **UNDEFINED** at all Exception levels if EL2 is not implemented.
- **DCPS2** is **UNDEFINED** at the following Exception levels if EL2 is implemented:
  - At EL0 and EL1 in Secure state if EL2 is disabled in the current Security state.
  - At EL3 if EL3 is using AArch32.
- **DCPS3** is **UNDEFINED** at all Exception levels if either:
  - EDSR.SDD == 1.
  - EL3 is not implemented.

Note

The references to DCPS1, DCPS2, and DCPS3 in this section link to the descriptions of the instructions in the A64 instruction set. The DCPS<n> instructions are also defined in the T32 instruction set, see DCPS1, DCPS2, DCPS3. These instructions are not defined in the A32 instruction set, because A32 instructions cannot be executed in Debug state.

On executing a DCPS instruction:
- If the target Exception level is using AArch64:
  - ELR_ELx of the target Exception level becomes UNKNOWN.
  - SPSR_ELx of the target Exception level becomes UNKNOWN.
  - ESR_ELx of the target Exception level becomes UNKNOWN.
Debug State
H2.4 Behavior in Debug state

— DLR_EL0 and DSPSR_EL0 become UNKNOWN.

• If the target Exception level is using AArch32 DLR and DSPSR become UNKNOWN and:
  — If the target Exception level is EL1 or EL3, the LR and SPSR of the target mode become UNKNOWN.
  — If the target Exception level is EL2, then ELR_hyp, SPSR_hyp, and HSR become UNKNOWN.

If the target Exception level is using AArch32, and the target Exception level is EL1 or EL3, the LR and SPSR of
the target mode become UNKNOWN.

If FEAT_SSBS is implemented, the DCPS<-> instruction leaves the PSTATE.SSBS bit UNKNOWN.

The DCPSInstruction() function is described in Chapter J1 Armv8 Pseudocode.

DRPS

Executing the DRPS operation in Debug state moves the PE to a lower Exception level, or to another PE mode at
the current Exception level, by copying the current SPSR to PSTATE.

If DRPS is executed in AArch64 state and the target Exception level is using AArch32:
• The current instruction set switches from A64 to T32.
• The effect on registers that are not visible or only partially visible in AArch32 state is the same as for
  exception returns in Non-debug state. See Execution state on page D1-4632.

Otherwise, the instruction set state does not change.

If the target Exception level is the same as the current Exception level, then the PE does not change Exception level.
However, the PE might change mode.

The effect on endianness is the same as for exceptions and exception returns in Non-debug state:
• If targeting an Exception level using AArch64, current endianness is set according to SCTLR_ELx.EE, or
  SCTLR_EL1.E0E for the target Exception level.
• If targeting an Exception level using AArch32, current endianness is set by SPSR.E as appropriate.

The DRPS instructions are:
In AArch64 state

• DRPS

In AArch32 state, in the T32 instruction set only

• ERET

If the SPSR specifies an illegal exception return, then PSTATE.{M, nRW, EL, SP} are unchanged and PSTATE.IL
is set to 1. For further information on illegal exception returns, see Illegal exception returns from AArch64 state
on page D1-4645.

PSTATE.{N, Z, C, V, Q, GE, IT, T, SS, D, A, I, F} are ignored in Debug state. This means that the effect of the DRPS
operation on these fields is to set them to an UNKNOWN value that might be the value from the SPSR. For more
information, see PSTATE in Debug state on page H2-10242.

All other PSTATE fields are copied from SPSR.

DRPS is UNDEFINED at EL0 and in Non-debug state.

Note

Unlike an exception return, the DRPS operation has no architecturally-defined effect on the Event Register and
Exclusives monitors. DRPS might set the Event Register or clear the Exclusives monitors, or both, but this is not a
requirement and debuggers must not rely on any implementation specific behavior.

On executing a DRPS instruction:
• If the target Exception level is using AArch64:
  — DLR_EL0 and DSPSR_EL0 become UNKNOWN.
• If the target Exception level is using AArch32:
  — DLR and DSPSR become UNKNOWN.
If FEAT_SSBS is implemented, the DRPS instruction leaves the PSTATE.SSBS bit UNKNOWN.

The DRPSInstruction() function is described in Chapter J1 Armv8 Pseudocode.

**MRS and MSR**

The other Debug state instructions are used to read or write DLR_EL0 and DSPSR_EL0.

These instructions are:

**In AArch64 state**
- MRS
- MSR (register)

**In AArch32 state**
- MRC
- MCR

MRS <Xt>, DLR_EL0 ; Copy DLR_EL0 to <Xt>
MRS <Xt>, DSPSR_EL0 ; Copy DSPSR_EL0 to <Xt>
MSR DLR_EL0, <Xt> ; Copy <Xt> to DLR_EL0
MSR DSPSR_EL0, <Xt> ; Copy <Xt> to DSPSR_EL0

These instructions can be executed at any Exception level when in Debug state, including EL0. They are UNDEFINED in Non-debug state.

**H2.4.7 Exceptions in Debug state**

The following sections describe how exceptions are handled in Debug state:

- *Generating exceptions when in Debug state.*
- *Taking exceptions when in Debug state* on page H2-10266.
- *Reset in Debug state on page H2-10267.*

### Generating exceptions when in Debug state

In Debug state:

- Instruction Abort exceptions cannot happen because instructions are not fetched from memory.
- Interrupts, including SError and virtual interrupts are ignored and remain pending:
  - The pending interrupt remains visible in ISR.
- Debug exceptions and debug events are ignored.
- SCR.EA is treated as if it were set to 0, regardless of its actual state, other than for the purpose of reading the bit.
- Any attempt to execute an instruction bit pattern that is an allocated instruction at the current Exception level, but is listed in *Executing instructions in Debug state on page H2-10243* as UNDEFINED in Debug state, generates an exception.

________ Note __________

If the exception is taken to an Exception level that is using AArch32 then it is taken as an Undefined Instruction exception.

The priority and syndrome for these exceptions is the same as for executing an encoding that does not have an allocated instruction.

- Instructions executed at EL2, EL1 and EL0 that are configured by EL3 control registers to trap to EL3:
  - When the value of EDSCR.SDD is 0, generate the appropriate trap exception that is taken to EL3.
  - When the value of EDSCR.SDD is 1, are treated as UNDEFINED and generate an exception.

If the exception is taken to an Exception level using AArch64 or to AArch32 Hyp mode, then it is reported with an EC value of 0x00.
Otherwise, synchronous exceptions are generated as they would be in Non-debug state and taken to the appropriate Exception level in Debug state.

Note

If EDSCR.SDD == 1 then an exception from Non-secure state is never taken to Secure state. See Security in Debug state on page H2-10261.

Taking exceptions when in Debug state

When the PE is in Debug state, all exceptions are synchronous. When an exception is generated, it is taken to Debug state. This means that:

• The target Exception level is as defined for the exception in Non-debug state.
• If the target Exception level is using AArch32 then the target PE mode is as defined for the exception in Non-debug state.
• The exception syndrome is reported as defined for the exception in Non-debug state, except for the case described in Data Aborts in Memory access mode on page H4-10304 for which the reporting requirements are relaxed.

The exception syndrome is reported using the syndrome register or registers for the target Exception level. In AArch64 state, these are ESR_ELx, and FAR_ELx. In AArch32 state, these are DFSR, DFAR, HSR, HDFAR, and HPFAR. For example:

— If a Data Abort exception is taken to Abort mode at EL1 or EL3 and the exception is taken from AArch32 state and using the Short-descriptor translation table format, the DFSR reports the exception using the Short-descriptor format fault encoding. For exceptions other than Data Abort exceptions taken to Abort mode, DFSR is not updated.
— If an instruction is trapped to an Exception level using AArch64 due to a configurable trap, disable, or enable, the exception code reported is the same as it would be in Non-debug state.

The effect on auxiliary syndrome registers, such as AFSR, is IMPLEMENTATION DEFINED.

Note

Generally, the AArch32 Fault Address Registers (FARs) and Fault Status Registers (FSRs) are not described as syndrome registers, although the term is appropriate to their function.

• The PE remains in Debug state and changes to the target mode.
• If EL3 is using AArch32 and the exception is taken from Monitor mode, SCR.NS is cleared to 0.
• If the exception is taken to an Exception level using AArch32, the PE continues to execute T32 instructions, regardless of the TE bit in the System register for the target Exception level.
• The endianness switches to that indicated by the EE bit of the System register for the target Exception level.
• The SPSR for the target Exception level or mode is corrupted and becomes UNKNOWN.
• If the target Exception level is EL2 using AArch32, ELR_hyp becomes unknown.
• If the target Exception level is EL1 or EL3 using AArch32, LR_<mode> for the target mode becomes unknown.
• DLR and DSPSR become UNKNOWN.
• The cumulative error flag, EDSCR.ERR, is set to 1. See Cumulative error flag on page H4-10308.
• PSTATE.IL is cleared to 0.
• PSTATE.{IT, T, SS, D, A, I, F, ALLINT} are set to UNKNOWN values, and PSTATE.{N, Z, C, V, Q, GE} are unchanged. However, these fields are ignored and are not observable in Debug state. For more information, see PSTATE in Debug state on page H2-10242.

The debugger must save any state that can be corrupted by an exception before executing an instruction that might generate another exception.
Pseudocode description of taking exceptions in Debug state

The pseudocode function `AArch64.TakeException()` shows the behavior when the PE takes an exception to an Exception level using AArch64 in Non-debug state. In Debug state, this is replaced with the function `AArch64.TakeExceptionInDebugState()`.

The pseudocode functions `AArch32.EnterMode()`, `AArch32.EnterHypMode()`, and `AArch32.EnterMonitorMode()` show the behavior when the PE takes an exception to an Exception level using AArch32 in Non-debug state. In Debug state:

- `AArch32.EnterMode()` is replaced with the function `AArch32.EnterModeInDebugState()`.
- `AArch32.EnterHypMode()` is replaced with the function `AArch32.EnterHypModeInDebugState()`.
- `AArch32.EnterMonitorMode()` is replaced with `AArch32.EnterMonitorModeInDebugState()`.

Reset in Debug state

If the PE is reset when in Debug state, it exits Debug state and enters Non-debug reset state. When the PE is in reset state, `EDSCR.STATUS == 0b000010` and writes to `EDITR` are ignored.

Note

If `EDECR.RCE == 1` or `CTIDEVCTL.RCE == 1`, meaning that a Reset Catch debug event is programmed, and if halting is allowed on exiting reset state, then on exiting reset state the PE halts and re-enters Debug state. See Reset Catch debug events on page H3-10293. All PE registers have taken their reset values, which might be UNKNOWN.

H2.4.8 Accessing registers in Debug state

Register accesses are unchanged in Debug state. The view of each register is determined by either the current Exception level or the mode, or both, and accesses might be disabled or trapped by controls at a higher Exception level.

General-purpose register access, other than AArch64 state SP access

A single general-purpose register can be read by issuing an MSR instruction through the ITR to write `DBGDTR_EL0` in AArch64 state, or an MCR instruction through the ITR to write `DBGDTRXTint` in AArch32 state. The debugger can then read the DTR register or registers through the external debug interface. The reverse sequence writes to a general-purpose register.

Figure H2-1 on page H2-10268 shows the reading and writing of general-purpose registers, other than SP, in Debug state in AArch64 state.
Figure H2-1 Reading and writing general-purpose registers, other than SP, in Debug state in AArch64 state

Figure H2-2 on page H2-10269 shows the reading and writing of general-purpose registers in Debug state in AArch32 state.
### SIMD and floating-point register, System register, and AArch64 state SP accesses

To read a SIMD and floating-point register or a System register, the debugger must first copy the value into a general-purpose register using:

- An FMOV instruction in AArch64 state or a VMOV instruction in AArch32 state for floating-point transfers to SIMD and FP registers.
- A UMOV instruction in AArch64 state or a VMOV instruction in AArch32 state for SIMD transfers to SIMD and FP registers.
- An MSR instruction in AArch64 state or an MRC instruction in AArch32 state for System registers.
- A MOV Xd, SP instruction for the SP register in AArch64 state.

The debugger can then read out the particular general-purpose register. The reverse sequence writes a register.

### PC and PSTATE access

The debugger reads the program counter and PSTATE of the process being debugged through the DLR_EL0 and DSPSR_EL0 System registers. The actual values of PC and PSTATE cannot be directly observed in Debug state:

- Instructions that are used for direct reads and writes of PC and PSTATE in Non-debug state are undefined in Debug state.
- On taking an exception, ELR_ELx and SPSR_ELx at the target Exception level are unknown. They do not record the PC and PSTATE.

PSTATE {TCO, UAO, PAN, IL, E, M, nRW, EL, SP} are indirectly read by instructions executed in Debug state, but all other PSTATE fields are ignored and cannot be observed. See also:

- **PSTATE in Debug state on page H2-10242.**
- **Executing instructions in Debug state on page H2-10243.**
- **Exceptions in Debug state on page H2-10265.**
H2.4.9   Accessing memory in Debug state

How the PE accesses memory is unchanged in Debug state. This includes:

- The operation of the MMU, including address translation, tagged address handling, access permissions, memory attribute determination, and the operation of any TLBs.
- The operation of any caches and coherency mechanisms.
- Alignment support.
- Endianness support.
- The Memory order model.

Simple memory transfers

Simple memory accesses can be performed in Debug state by issuing memory access instructions through the ITR and passing data through the DTR registers. Executing instructions in Debug state on page H2-10243 lists the memory access instructions that are supported in Debug state.

Bulk memory transfers

Memory access mode can accelerate bulk memory transfers in Debug state. See DCC and ITR access modes on page H4-10302.
H2.5 Exiting Debug state

The PE exits Debug state when it receives a Restart request trigger event. If EDSCR.ITE == 0 the behavior of any instruction issued through the ITR in Normal access mode or an operation issued by a DTR access in memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state after the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

--- Note ---

- Implementations can set EDSCR.ITE to 1 to indicate that further instructions can be accepted by ITR before the previous instructions have completed. If any previous instruction has not completed and EDSCR.ITE == 1, then the PE must complete these instructions in Debug state before executing the restart sequence. EDSCR.ITE == 0 indicates that the PE is not ready to restart.

- A debugger must observe that any instructions issued through EDITR that might generate a synchronous exception, as complete, before issuing a restart request. It can do this by observing the completion of a later instruction, as synchronous exceptions must occur in program order. For example, a debugger can observe that an instruction that reads or writes a DTR register is complete because of its effect on the EDSCR.{TXfull, RXfull} flags.

On exiting Debug state, the PE sets the program counter to the address in DLR, where:

- If exiting to AArch32 state:
  - Bits[31:1] of the PC are set to the value of bits[31:1] of DLR.
  - Bit[0] of the PC is set to a CONSTRAINED UNPREDICTABLE choice of 0 or the value of bit[0] in DLR.
- If exiting to AArch64 state:
  - Bits[63:56] of DLR_EL0 might be ignored as part of tagged address handling. See Address tagging on page D8-5162.
  - Otherwise the PC is set from DLR_EL0.

--- Note ---

Bits[63:32] of DLR_EL0 are ignored when exiting to AArch32 state.

Exit from Debug state can give rise to a PC alignment fault exception when the program counter is used. Unlike an exception return, this might also happen when returning to AArch32 state. For more information, see PC alignment checking on page D1-4668.

On exiting Debug state, PSTATE is set from DSPSR in the same way that an exception return sets PSTATE from SPSR_ELx:

- The same illegal exception return checks that apply to an exception return also apply to exiting Debug state. If the return from Debug state is an illegal exception return then the effect on PSTATE and the PC is the same as for any other illegal exception return. See Exception return on page D1-4644 and Exception return to an Exception level using AArch32 on page G1-8951.

- The checks on the PSTATE.IT bits that apply to exiting Debug state into AArch32 state are the same as those that apply to an exception return. See Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

- PSTATE.SS is copied from DSPSR.SS if all of the following hold:
  - MDSCR_EL1.SS == 1.
  - The debug target Exception level is using AArch64.
  - Software step exceptions from the restart Exception level are enabled. Otherwise PSTATE.SS is set to 0.

--- Note ---

Unlike a return using ERET, PSTATE.SS must be restored from DSPSR.SS because otherwise it is UNKNOWN.
However, if OSDLR.DLK == 1 and DBGPRCR.CORENPDRQ == 0, meaning FEAT_DoubleLock is implemented and locked in Non-debug state and therefore Software Step exceptions are disabled, but otherwise Software Step exceptions would be enabled from the restart Exception level, it is CONSTRAINED UNPREDICTABLE whether PSTATE.SS is copied from DSPSR.SS.

- If FEAT_SBS is implemented, on exit from Debug state to AArch64 state, DSPSR_EL0.SSBS is copied to PSTATE.SSBS.
- If FEAT_SBS is implemented, on exit from Debug state to AArch32 state, DSPSR.SSBS is copied to CPSR.SSBS.
- If FEAT_PAN is implemented, DSPSR_EL0.PAN is copied to PSTATE.PAN.
- If FEAT_UAO is implemented, DSPSR_EL0.UAO is copied to PSTATE.UAO.
- If FEAT_DIT is implemented, on exit from Debug state to AArch64 state, DSPSR_EL0.DIT is copied to PSTATE.DIT.
- If FEAT_DIT is implemented, on exit from Debug state to AArch32 state, DSPSR.DIT is copied to CPSR.DIT.
- If FEAT_MTE is implemented, on exit from Debug state to AArch64 state, DSPSR_EL0.TCO is copied to PSTATE.TCO. On exit from Debug state to AArch32 state, PSTATE.TCO is not updated.
- If FEAT_BTI is implemented, DSPSR_EL0.BTYPE is copied to PSTATE.BTYPE.
- If FEAT_NMI is implemented, DSPSR_EL0.ALLINT is copied to PSTATE.ALLINT.

--- Note ---

- One important difference between Debug state exit and an exception return is that the PE can exit Debug state at EL0. Despite this, the behavior of an exit from Debug state is similar to an exception return. For example, PSTATE.{D, A, I, F} is updated regardless of the value of SCTLR_EL1.UMA.

- Exit from Debug state has no architecturally-defined effect on the Event Register and Exclusives monitors. An exit from Debug state might set the Event Register or clear the Exclusives monitors, or both, but this is not a requirement and debuggers must not rely on any implementation specific behavior.

---

The ExitDebugState() function is described in Chapter J1 Armv8 Pseudocode.
Chapter H3
Halting Debug Events

This chapter describes a particular class of debug events. It contains the following sections:

• Introduction to Halting debug events on page H3-10274.
• Halting Step debug events on page H3-10276.
• Halt Instruction debug event on page H3-10286.
• Exception Catch debug event on page H3-10287.
• External Debug Request debug event on page H3-10291.
• OS Unlock Catch debug event on page H3-10292.
• Reset Catch debug events on page H3-10293.
• Software Access debug event on page H3-10294.
• Synchronization and Halting debug events on page H3-10295.

——— Note ————
Table K17-1 on page K17-11836 disambiguates the general register references used in this chapter.
H3.1 Introduction to Halting debug events

External debug defines Halting debug events. The following Halting debug events are available from the introduction of Armv8:

- Halting Step debug events on page H3-10276.
- Halt Instruction debug event on page H3-10286.
- Exception Catch debug event on page H3-10287.
- External Debug Request debug event on page H3-10291.
- OS Unlock Catch debug event on page H3-10292.
- Reset Catch debug events on page H3-10293.
- Software Access debug event on page H3-10294.

If halting is allowed, a Halting debug event halts the PE. The PE enters Debug state.

In addition, breakpoints and watchpoints might halt the PE if halting is allowed. See Breakpoint and Watchpoint debug events on page H2-10234. Because breakpoints and watchpoints can generate an exception or halt the PE, Breakpoint and Watchpoint debug events are not classified as Halting debug events.

For a definition of Debug state, see Chapter H2 Debug State. For a definition of halting allowed, see Halting allowed and halting prohibited on page H2-10233.

Debug state entry and debug event prioritization on page H2-10235 describes the behavior when multiple debug events are generated by an instruction.

See also Synchronization and Halting debug events on page H3-10295.

Table H3-1 on page H3-10274 shows the behavior of Breakpoint, Watchpoint, and Halting debug events.

<table>
<thead>
<tr>
<th>Debug event type</th>
<th>PE behavior when halting is:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Allowed</td>
</tr>
<tr>
<td>Breakpoint and Watchpoint debug events on page H2-10234</td>
<td>Halt</td>
</tr>
<tr>
<td>Halt Instruction debug event on page H3-10286</td>
<td>Halt</td>
</tr>
<tr>
<td>Software Access debug event on page H3-10294</td>
<td>Halt</td>
</tr>
<tr>
<td>Exception Catch debug event on page H3-10287</td>
<td>Halt</td>
</tr>
<tr>
<td>Halting Step debug events on page H3-10276</td>
<td>Halt</td>
</tr>
<tr>
<td>External Debug Request debug event on page H3-10291</td>
<td>Halt</td>
</tr>
<tr>
<td>Reset Catch debug events on page H3-10293</td>
<td>Halt</td>
</tr>
<tr>
<td>OS Unlock Catch debug event on page H3-10292</td>
<td>Pended</td>
</tr>
</tbody>
</table>

Table H3-2 on page H3-10274 shows where the pseudocode for each Halting debug event type is located.

<table>
<thead>
<tr>
<th>Halting debug event type</th>
<th>Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Halt Instruction debug event on page H3-10286</td>
<td>HLT on page C6-1506 for AArch64 and HLT on page F5-7574 for AArch32</td>
</tr>
<tr>
<td>Software Access debug event on page H3-10294</td>
<td>Pseudocode description of Software Access debug event on page H3-10294</td>
</tr>
<tr>
<td>Exception Catch debug event on page H3-10287</td>
<td>Pseudocode description of Exception Catch debug events on page H3-10290</td>
</tr>
<tr>
<td>Halting debug event type</td>
<td>Pseudocode</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Halting Step debug events on page H3-10276</td>
<td>Pseudocode description of Halting Step debug events on page H3-10285</td>
</tr>
<tr>
<td>External Debug Request debug event on page H3-10291</td>
<td>Pseudocode description of External Debug Request debug events on page H3-10291</td>
</tr>
<tr>
<td>Reset Catch debug events on page H3-10293</td>
<td>Pseudocode description of Reset Catch debug event on page H3-10293</td>
</tr>
<tr>
<td>OS Unlock Catch debug event on page H3-10292</td>
<td>Pseudocode description of OS Unlock Catch debug event on page H3-10292</td>
</tr>
</tbody>
</table>
H3.2 Halting Step debug events

Halting Step is a debug resource that a debugger can use to make the PE step through code one instruction at a time. This section describes the Halting Step debug events. It is divided into the following sections:

- Overview of a Halting Step debug event.
- The Halting Step state machine.
- Using Halting Step on page H3-10279.
- Detailed Halting Step state machine behavior on page H3-10279.
- Synchronization and the Halting Step state machine on page H3-10282.
- Stepping T32 IT instructions on page H3-10283.
- Disabling interrupts while stepping on page H3-10284.
- Syndrome information on Halting Step on page H3-10284.
- Pseudocode description of Halting Step debug events on page H3-10285.

The architecture describes the behavior as a simple Halting Step state machine. See The Halting Step state machine.

H3.2.1 Overview of a Halting Step debug event

The behavior of Halting Step is defined by a state machine, shown in Figure H3-1 on page H3-10278. A Halting Step debug event executes a single instruction and then returns control to the debugger. When the debugger software wants to execute a Halting Step:

1. With the PE in Debug state, the debugger activates Halting Step.
2. The debugger signals the PE to exit Debug state and return to the instruction that is to be stepped.
3. The PE executes that single instruction.
4. The PE enters Debug state before executing the next instruction.

However, an exception might be generated while the instruction is being stepped. That is either:

- A synchronous exception generated by the instruction being stepped.
- An asynchronous exception taken before or after the instruction being stepped.

Halting Step has its own enable control bit, EDECR.SS and EDESR.SS.

--- Note ---

Because the Halting Step state machine states occur as a result of normal PE operation, the states can be described as both:

- PE states.
- Halting Step states.

H3.2.2 The Halting Step state machine

The state machine states are:

**Inactive**

Halting Step is inactive. No Halting Step debug events can be generated, therefore execution is not affected by Halting Step. The PE is in this state whenever either of the following is true:

- Halting Step is disabled. That is, EDECR.SS is set to 0 and EDESR.SS is set to 0.
- Halting is prohibited. See Halting the PE on debug events on page H2-10233. In this state, if EDECR.SS is set to 1, then a Halting Step debug event is pending.

In Figure H3-1 on page H3-10278, this state is shown in red.

**Active-not-pending**

Halting Step is enabled and active. This is the state in which the PE steps an instruction. EDECR.SS == 1 and EDESR.SS == 0. Software must not set EDECR.SS to 1 unless the PE is in Debug state, otherwise behavior is CONSTRAINED UNPREDICTABLE, as described in Changing the value of EDECR.SS when not in Debug state on page H3-10283.

In Figure H3-1 on page H3-10278, this state is shown in green.
Active-pending

Halting Step is enabled and active. The step has completed and the PE enters Debug state.

EDESR.SS == 1.

In Figure H3-1 on page H3-10278, this state is shown in green.

Whenever Halting Step is enabled and active, whether the state machine is in the active-not-pending state or in the active-pending state depends on EDESR.SS. Halting Step state machine states on page H3-10279 shows this.

In the simple sequential execution of the program, the PE executes the Halting Step state machine as follows:
1. Initially, Halting Step is inactive.
2. After exiting Debug state, Halting Step is active-not-pending.
3. The PE executes an instruction and Halting Step is active-pending.
4. The pending Debug state entry is taken on the next instruction and the step is complete.

Exceptions and other changes to the PE context can interrupt this sequence.

Figure H3-1 on page H3-10278 shows a Halting Step state machine.
Halting Debug Events

H3.2 Halting Step debug events

Figure H3-1 Halting Step state machine

a. Step completed occurs when:
   • A debug event, other than a Halting Step debug event, causes entry into Debug state.

b. Step completed occurs when:
   • An instruction is executed without taking an exception.
   • An exception is taken to a state where halting is allowed.
   • A reset.

c. Step completed occurs when:
   • An SMC exception is taken to Secure state where halting is prohibited.

d. An asynchronous exception taken to a state where halting is allowed.

e. An asynchronous exception taken to Secure state where halting is prohibited.
Note

Figure H3-1 on page H3-10278 describes only state transitions to and from the inactive state by exit from Debug state, executing an exception return, or taking an exception. Other changes to the PE context, including writes to registers such as EDECR and OSDLR and changes to the authentication interface can also cause changes to the Halting Step state machine. These can lead to UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior. See Synchronization and the Halting Step state machine on page H3-10282.

The following bits control the state machine, as shown in Table H3-3:

- EDECR.SS. This is the Halting Step enable bit.
- EDESR.SS. This is the Halting Step debug event pending bit.

Table H3-3 shows the Halting Step state machine states. The letter X in a register column means that the relevant bit can be set to either zero or one.

<table>
<thead>
<tr>
<th>Halting</th>
<th>EDECR.SS</th>
<th>EDESR.SS</th>
<th>Halting Step state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prohibited</td>
<td>X</td>
<td>X</td>
<td>Inactive (Halting Step debug even not pending)</td>
</tr>
<tr>
<td>Prohibited</td>
<td>X</td>
<td>1</td>
<td>Inactive (Halting Step debug event pending)</td>
</tr>
<tr>
<td>Allowed</td>
<td>0</td>
<td>0</td>
<td>Inactive</td>
</tr>
<tr>
<td>Allowed</td>
<td>1</td>
<td>0</td>
<td>Active-not-pending</td>
</tr>
<tr>
<td>Allowed</td>
<td>X</td>
<td>1</td>
<td>Active-pending</td>
</tr>
</tbody>
</table>

H3.2.3 Using Halting Step

To step a single instruction the PE must be in Debug state:
1. The debugger sets EDECR.SS to 1 to enable Halting step.
2. The debugger signals the PE to exit Debug state with DLR set to the address of the instruction being stepped. The PE clears EDESR.SS to 0 and the Halting Step state machine enter the active-not-pending state.
3. The PE executes the instruction being stepped. If an exception is taken to a state where halting is prohibited, then EDESR.SS is always correct for the preferred return address of the exception.
4. The PE enters Debug state before executing the next instruction and the step is complete.

Note

- If FEAT_DoPD is not implemented, EDECR.SS value is in the Debug power domain, meaning that the state machine is maintained over a powerdown of the Core power domain.
- If FEAT_DoPD is implemented, the values of EDECR.SS and EDESR.SS are set to 0 on a Cold reset, and, if the PE was stepping an instruction, EDESR.SS is effectively UNKNOWN after a Warm reset. A debugger must use a Reset Catch debug event to step over a powerdown state.
- A debugger must only change the value of EDECR.SS when the PE is in Debug state, otherwise behavior is CONSTRAINED UNPREDICTABLE as described in Changing the value of EDECR.SS when not in Debug state on page H3-10283.

H3.2.4 Detailed Halting Step state machine behavior

The behavior of the Halting Step state machine is described in the following sections:

- Entering the active-not-pending state on page H3-10280.
- PE behavior in the active-not-pending state on page H3-10280.
- Entering the active-pending state on page H3-10281.
Halting Debug Events

H3.2 Halting Step debug events

- **PE behavior in the active-pending state on page H3-10281.**
- **PE behavior in the inactive state when in Non-debug state on page H3-10282.**
- **PE behavior in Debug state on page H3-10282.**

### Entering the active-not-pending state

The PE enters the active-not-pending state:
- By exiting Debug state to a state where halting is allowed with $EDECR.SS == 1$.
- By an exception return from a state where halting is prohibited to a state where halting is allowed with $EDECR.SS == 1$ and $EDESR.SS == 0$.
- As described in *Synchronization and the Halting Step state machine on page H3-10282.*

### PE behavior in the active-not-pending state

When the PE is in the active-not-pending state it does one of the following:
- It executes one instruction and does one of the following:
  - Completes it without taking a synchronous exception.
  - Takes a synchronous exception generated by the instruction.
  - Generates a debug event that causes entry to Debug state.
- It takes an asynchronous exception without executing any instruction.
- It takes an asynchronous debug event into Debug state.

**If no exception or debug event is generated**

If no exception or debug event is generated the PE sets $EDESR.SS$ to 1. This means that the Halting Step state machine advances to the active-pending state.

**If an exception or debug event is generated**

The PE sets $EDESR.SS$ according to all of the following:
- The type of exception.
- The target Exception level of the exception.
- If the exception is taken to Secure state, whether halting is prohibited in Secure state.
  - This is determined by the result of $ExternalSecureInvasiveDebugEnabled()$.

If an exception or debug event is generated, the PE sets $EDESR.SS$ to 1 if one of the following applies:
- A synchronous exception is generated by the instruction and one of the following applies:
  - The exception is taken to EL1 or EL2.
  - The exception is taken to EL3, it is not an SMC exception, and $ExternalSecureInvasiveDebugEnabled() \equiv TRUE$.
  - The exception is an SMC exception.
- An asynchronous exception is generated before executing an instruction and this is either:
  - Taken to EL1 or EL2.
  - Taken to EL3 and $ExternalSecureInvasiveDebugEnabled() \equiv TRUE$.
- A PE reset occurs.

Otherwise, $EDESR.SS$ is unchanged. This happens when:
- No instruction is executed because either:
  - An asynchronous exception is taken to EL3 and $ExternalSecureInvasiveDebugEnabled() \equiv FALSE$.
  - An asynchronous debug event causes entry to Debug state.
- An instruction is executed and either:
  - Generates a synchronous exception other than an SMC exception which is taken to EL3, and $ExternalSecureInvasiveDebugEnabled() \equiv FALSE$.
  - Generates a synchronous debug event and causes entry to Debug state.
It is **UNPREDICTABLE** whether `EDESR.SS` is set to 1 or unchanged when an SError interrupt is taken to EL3 without executing the instruction, and `ExternalSecureInvasiveDebugEnabled()` == FALSE.

If halting is prohibited after taking the exception or debug event, then the Halting Step state machine advances to the inactive state. Otherwise, the Halting Step state machine advances to the active-pending state.

**Note**

The underlying criteria for the value of `EDESR.SS` on an exception are:

- Whether halting is allowed at the target of the exception. If halting is allowed, the PE must step into the exception. If halting is prohibited, the PE must step over the exception.
- Whether the preferred return address of the exception is the instruction itself or the next instruction, if the PE steps over the exception.

Table H3-4 on page H3-10281 shows the behavior of the active-not-pending state. The letter X indicates that `ExternalSecureInvasiveDebugEnabled()` can be either TRUE or FALSE.

<table>
<thead>
<tr>
<th>Event</th>
<th>Target Exception level</th>
<th><code>ExternalSecureInvasiveDebugEnabled()</code></th>
<th>Value written to <code>EDESR.SS</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>No exception or debug event</td>
<td>Not applicable</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>SMC exception</td>
<td>EL3</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>Reset</td>
<td>Highest</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>Exception, other than SMC exception</td>
<td>EL1</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>EL2</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>EL3</td>
<td>TRUE</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FALSE</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Debug event</td>
<td>Debug state</td>
<td>X</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>

**Entering the active-pending state**

The PE enters the active-pending state by one of the following:

- From the active-not-pending state by:
  - Executing an instruction without taking an exception.
  - Taking an exception so that the PE remains in a state where halting is allowed.
- An exception return from a state where halting is prohibited when `EDESR.SS` == 1.

**Note**

That is, an exception return from Secure state with `ExternalSecureInvasiveDebugEnabled()` == FALSE to Non-secure state with `ExternalInvasiveDebugEnabled()` == TRUE.

- A reset when the value of `EDECR.SS` == 1, regardless of the state the PE was in before the reset occurred.
- From the active-pending state by taking an asynchronous exception to a state where halting is allowed.
- Following the description in *Synchronization and the Halting Step state machine on page H3-10282.*

**PE behavior in the active-pending state**

When the PE is in the active-pending state, it enters Debug state before executing an instruction.
The entry into Debug state has higher priority than all other types of synchronous debug event and synchronous exception. However, the architecture does not define the prioritization of this Debug state entry with respect to any unmasked pending asynchronous exception. If an asynchronous exception is prioritized over the entry to Debug state, then EDESR.SS is unchanged.

For more information on the prioritization of debug events, see Debug state entry and debug event prioritization on page H2-10235.

**PE behavior in the inactive state when in Non-debug state**

EDESR.SS is not updated by the execution of an instruction or the taking of an exception when Halting Step is inactive. This means that EDESR.SS is not changed by an exception handled in a state where halting is prohibited.

On return to a state where halting is allowed, the Halting Step state machine is restored either to the active-pending state or the active-not-pending state, depending on the value of EDESR.SS. The return to a state where halting is allowed is normally by an exception return, which in some situations is a Context synchronization event.

See also Synchronization and the Halting Step state machine.

**PE behavior in Debug state**

Halting Step is inactive in Debug state because halting is prohibited, see Halting allowed and halting prohibited on page H2-10233.

Entry to Debug state does not change EDESR.SS.

EDESR.SS is cleared to 0 on exiting Debug state as the result of a restart request. If EDECR.SS == 1, Halting Step enters the active-not-pending state.

--- Note ---

This means that EDESR.SS is never cleared to 0 by the execution of an instruction in Debug state, or by taking an exception when in Debug state as described in PE behavior in the active-not-pending state on page H3-10280, because the Halting Step state machine is not in the active-not-pending state. EDESR.SS can be cleared by a write to EDESR, see the register description.

---

However, if the PE exits Debug state as the result of a PE reset and EDECR.SS == 1, then Halting Step immediately enters the active-pending state, as EDESR.SS is set to the value of EDECR.SS.

**H3.2.5 Synchronization and the Halting Step state machine**

The Halting Step state machine also changes state if:

- Halting becomes allowed or prohibited other than by exit from Debug state, an exception return, or taking an exception. This means that halting becomes allowed or prohibited because:
  - The Security state changes without an exception return. See State and mode changes without explicit context synchronization events on page G2-9100.
  - The external authentication interface changes.
  - FEAT_DoubleLock is implemented and the status, DoubleLockStatus(), changes.
- A write to EDECR when the PE is in Non-debug state changes the value of EDECR.SS.

--- Note ---

Behavior is CONSTRAINED UNPREDICTABLE if the value of EDECR.SS is changed when the PE is in Non-debug state, see Changing the value of EDECR.SS when not in Debug state on page H3-10283.

---

- A write to EDESR when the PE is in Non-debug state clears EDESR.SS to 0.

These operations are guaranteed to take effect only after a Context synchronization event. If the instruction being stepped generates a Context synchronization event, then the PE might use the old or new state.
The PE must perform the required behavior of the new state before or immediately following the next \textit{Context synchronization event}, but it is not required to do so immediately. This means that the PE can perform the required behavior of the old state before the next \textit{Context synchronization event}. This is illustrated in Example H3-1 and Example H3-2.

\textbf{Example H3-1 Synchronization requirements 1}

\texttt{EDECR.SS} is set to 1 in Debug state, requesting the active-not-pending state on exit from Debug state. On exit from Debug state the PE immediately takes an exception to Secure state. \texttt{ExternalSecureInvasiveDebugEnabled()} == FALSE, meaning that halting is prohibited in Secure state. The PE does not step any instructions but executes the software in Secure state as normal. \texttt{EDESR.SS} remains set to 0. If \texttt{ExternalSecureInvasiveDebugEnabled()} subsequently becomes TRUE, meaning that halting is now allowed, the PE must perform the required behavior of the active-not-pending state before or immediately following the next \textit{Context synchronization event}, but it is not required to do so immediately.

\textbf{Example H3-2 Synchronization requirements 2}

\texttt{EDECR.SS} is set to 1 in Debug state. On exit from Debug the PE executes an \texttt{MSR} instruction that sets \texttt{OSDLR_EL1.DLK} to 1 and \texttt{DoubleLockStatus()} becomes TRUE. This change requires a \textit{Context synchronization event} to guarantee its effect, meaning it is \textbf{CONSTRAINED UNPREDICTABLE} whether:

\begin{itemize}
  \item Halting is allowed:
    \begin{itemize}
      \item The PE enters Debug state on the next instruction.
    \end{itemize}
  \item Halting is prohibited:
    \begin{itemize}
      \item The PE does not enter Debug state.
    \end{itemize}
\end{itemize}

The value in \texttt{EDESR.SS} depends on whether halting was allowed or prohibited when the write to \texttt{OSDLR_EL1.DLK} completed, and so it might be 0 or 1. If a second \texttt{MSR} instruction clears \texttt{OSDLR_EL1.DLK} to 0, the PE must perform the required behavior of the state indicated by \texttt{EDESR.SS} before or immediately following the next \textit{Context synchronization event}, but it is not following the behavior to do so immediately.

See also \textit{Synchronization and Halting debug events} on page H3-10295.

\textbf{Changing the value of EDECR.SS when not in Debug state}

If software changes the value of \texttt{EDECR.SS} when the PE is not in Debug state then behavior is \textbf{CONSTRAINED UNPREDICTABLE}, and one or more of the following behaviors occurs:

\begin{itemize}
  \item The value of \texttt{EDECR.SS} becomes \texttt{UNKNOWN}.
  \item The state of the Halting Step state machine becomes \texttt{UNKNOWN}.
  \item On a reset of the PE, the value of \texttt{EDECR.SS} and the state of the Halting Step state machine are \texttt{UNKNOWN}.
\end{itemize}

\textbf{H3.2.6 Stepping T32 IT instructions}

In an implementation that supports the ITD control, the architecture permits a combination of one T32 IT instruction and another 16-bit T32 instruction to be treated as a single 32-bit instruction when the value of the ITD field that applies to the current Exception level is 1.

For the purpose of stepping an item, it is \textbf{IMPLEMENTATION DEFINED} whether:

\begin{itemize}
  \item The PE considers such a pair of instructions to be one instruction.
  \item The PE considers such a pair of instructions be two instructions.
\end{itemize}

It is \textbf{IMPLEMENTATION DEFINED} whether this behavior depends on the value of the applicable ITD bit. For example:

\begin{itemize}
  \item The debug logic might consider such a pair of instructions as one instruction, regardless of the state of the applicable ITD field.
\end{itemize}
• The debug logic might consider such a pair of instructions as two instructions, regardless of the state of the applicable ITD field.
• The debug logic might consider such a pair of instructions as one instruction when the value of the applicable ITD field is 1, and as two instructions when the value of the ITD field is 0.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0.

The ITD control fields are:

**HSCTLR.ITD**  
Applies to execution at EL2 when EL2 is using AArch32.

**SCTLR.ITD**  
Applies to execution at EL0 or EL1 when EL1 is using AArch32.

**SCTLR_EL1.ITD**  
Applies to execution at EL0 using AArch32 when EL1 is using AArch64.

**H3.2.7 Disabling interrupts while stepping**

When using Halting Step, the sequence of entering Debug state, interacting with the debugger, and then exiting Debug state for each instruction reduces the rate at which the PE executes instructions. However, the rate at which certain interrupts, such as timer interrupts, are generated might be fixed by the system. This means it might be necessary to disable interrupts while using Halting Step by setting EDSCR.INTdis, to allow the code being debugged to make forward progress.

**H3.2.8 Syndrome information on Halting Step**

Three EDSCR.STATUS encodings record different scenarios for entering Debug state on a Halting Step debug event:

**Halting Step, normal**  
An instruction other than a Load-Exclusive instruction was stepped.

**Halting Step, exclusive**  
A Load-Exclusive instruction was stepped.

**Halting Step, no syndrome**  
The syndrome data is not available.

If the PE enters Debug state due to a Halting Step debug event immediately after stepping an instruction in the active-not-pending state, EDSCR.STATUS is set to either:

• Halting Step, normal, if the stepped instruction was not a Load-Exclusive instruction.
• Halting Step, exclusive, if the stepped instruction was a Load-Exclusive instruction.

If the stepped instruction was a conditional Load-Exclusive instruction that failed its Condition code check, EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of Halting Step, normal, or Halting Step, exclusive.

Otherwise, the PE enters Debug state without stepping an instruction. This means that the Halting Step state machine enters the active-pending state directly from the inactive state, without going through active-not-pending state. In this case, EDSCR.STATUS is set to Halting Step, no syndrome. This happens when:

• The PE enters directly into the active-pending state on an exception return to Non-secure state from EL3 when Halting is prohibited in Secure state.
• The active-pending state is entered for other reasons. See Synchronization and the Halting Step state machine on page H3-10282.

In addition, EDSCR.STATUS is CONSTRAINED UNPREDICTABLE when:

• A different exception is taken before the Halting Step debug event.
In this case `EDSCR.STATUS` is set to a CONSTRAINED UNPREDICTABLE choice of:

- Halting Step, no syndrome, or Halting Step, normal, if the stepped instruction was not a Load-Exclusive instruction.
- Halting Step, no syndrome, or Halting Step, exclusive, if the stepped instruction was a Load-Exclusive instruction.

- The instruction that was stepped was an Exception Return instruction or an ISB. As these instructions are not in the Load-Exclusive instructions, `EDSCR.STATUS` is set to a CONSTRAINED UNPREDICTABLE choice of Halting Step, no syndrome or Halting Step, normal.
- The PE enters directly into the active-pending state on a Warm reset because `EDECR.SS` is set to 1. `EDSCR.STATUS` is set to a CONSTRAINED UNPREDICTABLE choice of Halting Step, no syndrome or Halting Step, normal.

In all cases, if `EDSCR.STATUS` is not set to Halting Step, no syndrome, then it must indicate whether the stepped instruction was a Load-Exclusive instruction by setting `EDSCR.STATUS` to Halting Step, normal or Halting Step, exclusive.

--- Note ---

If the PE cannot determine whether the stepped instruction was a Load-Exclusive instruction or not, then it sets `EDSCR.STATUS` to Halting Step, no syndrome. For example, the exception is taken before the PE decodes the stepped instruction, or the exception means the PE has no valid stepped instruction to decode.

--- Note ---

In an implementation that always sets `EDSCR.STATUS` to Halting Step, no syndrome is not compliant.

### H3.2.9 Pseudocode description of Halting Step debug events

There are two pseudocode functions for Halting Step debug events:

- `RunHaltingStep()`. This is called after an instruction has executed and any exception generated by the instruction is taken. It is also called after taking a reset before executing any instructions. That is, reset is treated like an asynchronous exception, even if `EDECR.RCE == 1` or `CTIDEVCTL.RCE == 1`. `RunHaltingStep()` affects the next instruction.
- `CheckHaltingStep()`. This is called before the next instruction is executed. If a step is pending, it generates the debug event.
H3.3 Halt Instruction debug event

A Halt Instruction debug event is generated when EDSCR.HDE == 1, halting is allowed, and the PE executes the Halt instruction, HLT.

The pseudocode for Halt Instruction debug events is described in HLT on page C6-1506 for A64 and HLT on page F5-7574 for A32 and T32.

HLT never generates a debug exception. It is treated as UNDEFINED if EDSCR.HDE == 0, or if halting is prohibited.

A debugger can replace a program instruction with a Halt instruction to generate a Halt Instruction debug event. Debuggers that use the HLT instruction must be aware of the rules for concurrent modification of executable code, CMODX. The rules for concurrent modification and execution of instructions do not allow one thread of execution or an external debugger to replace an instruction with an HLT instruction when these same instructions are being executed by a different thread of execution. See Concurrent modification and execution of instructions on page B2-156.

The T32 HLT instruction is unconditionally executed inside an IT block, even when it is treated as undefined. The A32 HLT instruction is CONSTRAINED UNPREDICTABLE if the Condition code field is not 0b1110, with the set of behaviors the same as for BKPT. See Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors.

The HLT instruction is part of the external debug solution from the introduction of Armv8-A. As such, the presence of the HLT instruction is not indicated in the ID registers. In particular, the AArch32 System register ID_ISAR0. Debug does not indicate the presence of the HLT instruction.

H3.3.1 HLT instructions as the first instruction in a T32 IT block

In an implementation that supports the ITD control, the architecture permits a combination of one T32 IT instruction and certain other 16-bit T32 instruction to be treated as a single 32-bit instruction when the value of the ITD field that applies to the current Exception level is 1.

The T32 HLT instruction cannot be combined with an IT instruction in this way. In an implementation that supports the ITD control, if the first instruction in an IT block is an HLT instruction, then the behavior of the instruction depends on the value of the applicable ITD field:

• If the value of the ITD field is 1, then the combination is treated as undefined and an Undefined Instruction exception is generated either by the IT instruction or by the HLT instruction.
• If the value of the ITD field is 0, then the HLT instruction unconditionally executed.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0.

To set an Halt Instruction debug event on the first instruction of an IT block, debuggers must replace the IT instruction with an HLT instruction to ensure consistent behavior.

The ITD control fields are:

**HSCTLR.ITD**
Applies to execution at EL2 when EL2 is using AArch32.

**SCTLR.ITD**
Applies to execution at EL0 or EL1 when EL1 is using AArch32.

**SCTLR_EL1.ITD**
Applies to execution at EL0 using AArch32 when EL1 is using AArch64.

A HLT instruction is always unconditional, even within an IT block.
H3.4 Exception Catch debug event

Exception Catch debug events:

- Are generated when the corresponding bit in the Exception Catch Control Register, EDECCR, is set to 1 on all entries to a given Exception level. This means:
  - Exceptions taken to the Exception level.
  - Exception returns to the Exception level.
  - It is IMPLEMENTATION DEFINED whether a reset into an Exception level generates an Exception Catch debug event.
- Might be taken synchronously, after the exception or reset entry or the exception return has been processed by the PE.
- Ignore the Execution state of the target Exception level.
- Might be ignored if halting is prohibited.
- If FEAT_Debugv8p8 is implemented, might be pended.

The EDECCR register contains fields that control when Exception Catch debug events are generated.

See Controlling Exception Catch debug events when FEAT_Debugv8p2 is implemented on page H3-10289 and Controlling Exception Catch debug events when FEAT_Debugv8p2 is not implemented on page H3-10289.

For exception returns, the final Exception level of the exception return determines whether an Exception Catch debug event is generated. On an illegal exception return, an Exception Catch debug event is generated only if EDECCR is programmed to generate an Exception Catch debug event for an exception return to the current Exception level.

H3.4.1 Prioritization of Exception Catch debug events

The following rules define the prioritization of Exception Catch debug events:

- It is IMPLEMENTATION DEFINED whether Exception Catch debug events are higher or lower priority than each of Software Step exceptions and Halting Step debug events.
- Exception Catch debug events are higher priority than all synchronous exceptions other than Software Step exceptions and debug events other than Halting Step debug events.
- Exception Catch debug events are lower priority than Reset Catch debug events.
- When FEAT_Debugv8p2 is implemented and FEAT_Debugv8p8 is not implemented, Exception Catch debug events are higher priority than pending asynchronous exceptions. Otherwise, the prioritization between asynchronous exceptions, asynchronous debug events, and an Exception Catch debug event, is IMPLEMENTATION DEFINED.

--- Note ---

As described in Prioritization of Synchronous exceptions taken to AArch64 state on page D1-4650, an exception trapping form of a Vector Catch debug event might generate a second debug exception as part of the exception entry, before the Exception Catch debug event is taken. See Vector Catch exceptions on page D2-4741 or Vector Catch exceptions on page G2-9093.

H3.4.2 Generating Exception Catch debug events when FEAT_Debugv8p2 is not implemented

When an Exception Catch debug event is generated after exception entry and halting is allowed at the target of the exception, the PE halts and enters Debug state:

- The PE halts and enters Debug state before the first instruction at the handler is executed, after the exception entry has updated the program counter, PSTATE, and syndrome registers for the exception.
- The PE does not fetch instructions from the vector address before entering Debug state, if address translation is disabled in the translation regime at the target Exception level.
- A second unmasked asynchronous exception can be taken before the PE enters Debug state. If this second exception does not generate an Exception Catch debug event, the exception handler executed at the higher Exception level later returns to the trapped Exception level, causing the Exception Catch debug event to be generated again.
When an Exception Catch debug event is generated on exception return and halting is allowed at the target of the exception return:

- There is no prioritization between asynchronous exceptions, asynchronous debug events, and an Exception Catch debug event generated on an exception return.
- The PE halts and enters Debug state after the exception return has updated the program counter and PSTATE, and before the execution of the first instruction at the return address is completed.

Otherwise, when the PE is executing code at a given Exception level, and the corresponding EDECCR bit is 1, it is CONstrained UNpredictable whether an Exception Catch debug event is generated.

Examples of this are:

- If the debugger writes to EDECCR so that the current Exception level is trapped.
- If the OS restore code writes to OSECCR so that the current Exception level is trapped.
- If the code executing in AArch32 state changes the Exception level or Security state other than by an exception return, and the target Exception level is trapped. See State and mode changes without explicit context synchronization events on page G2-9100.

See also Debug state entry and debug event prioritization on page H2-10235.

Note

It is possible to generate Exception Catch debug events:

- As a trap on all instruction fetches from the trapped Exception level as part of an instruction fetch.
- On entry to the Exception level, as described in Detailed Halting Step state machine behavior on page H3-10279.

This is similar to the implementation options allowed for Vector Catch debug events. The architecture does not require that the event is generated following an ISB operation executed at the Exception level.

H3.4.3 Generating Exception Catch debug events when FEAT_Debugv8p2 is implemented and FEAT_Debugv8p8 is not implemented

Exception Catch debug events are generated as described in Generating Exception Catch debug events when FEAT_Debugv8p2 is not implemented on page H3-10287, except:

- An Exception Catch debug event is generated only on exception entry or return.
- When an Exception Catch debug event is generated after exception entry and halting is allowed at the target of the exception, the PE halts and enters Debug state before any other asynchronous exception or debug event.

H3.4.4 Generating Exception Catch debug events when FEAT_Debugv8p8 is implemented

Exception Catch debug events are generated as described in Generating Exception Catch debug events when FEAT_Debugv8p2 is not implemented on page H3-10287, except:

- An Exception Catch debug event is generated only on exception entry or return.
- When an Exception Catch debug event is generated after exception entry and halting is allowed at the target of the exception, the PE sets EDESR.EC to 1, and one of the following occurs:
  - The PE halts immediately. On entry to Debug state, EDESR.EC might be 0 or 1.
  - The PE might take an unmasked asynchronous exception, changing Exception level. This means that the PE might enter a state where halting is prohibited before the PE halts and enters Debug state.
- When an Exception Catch debug event is generated on exception return and halting is allowed at the target of the exception, EDESR.EC is unchanged.
- When EDESR.EC is 1 and halting is allowed, the PE halts and enters Debug state before completing any instruction.
- When EDESR.EC is 1 and halting is prohibited, an Exception Catch debug event is pending.
- EDESR.EC is cleared to 0 upon:
  - A write of 1 to EDESR.EC.
  - Exit from Debug state.
Note

If EDESR.EC is 1 and the PE executes a Context synchronizing exception return from a state where halting is prohibited to a state where halting is allowed, then the Exception Catch debug event is prioritized over any synchronous exception or synchronous debug event generated by the first instruction after the Context synchronization event, other than possibly a Halting Step debug event or a Software Step exception.

H3.4.5 Controlling Exception Catch debug events when FEAT_Debugv8p2 is implemented

When FEAT_Debugv8p2 is implemented, the fields EDECCR.{NSR, SR, NSE, SE} control generation of Exception Catch debug events:
- On exception entry but not on exception return.
- On exception return but not on exception entry.
- On exception entry and exception return.

Exception entry, reset and exception return Exception Catch debug events are enabled as shown in Table H3-5 on page H3-10289.

<table>
<thead>
<tr>
<th>(NSR)&lt;n&gt;</th>
<th>(NSE)&lt;n&gt;</th>
<th>Behavior on exception return to ELn</th>
<th>Behavior on exception taken to ELn, and if resets are Exception Catch debug events, reset into ELn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No action.</td>
<td>No action.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Halt if allowed.</td>
<td>Halt if allowed.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Halt if allowed.</td>
<td>No action.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>No action.</td>
<td>Halt if allowed.</td>
</tr>
</tbody>
</table>

H3.4.6 Controlling Exception Catch debug events when FEAT_Debugv8p2 is not implemented

When FEAT_Debugv8p2 is not implemented, all Exception Catch debug events are enabled by a combination of the fields NSE and SE in EDECCR, as shown in Table H3-6 on page H3-10289.

<table>
<thead>
<tr>
<th>(NSE)&lt;n&gt;</th>
<th>Behavior on exception taken to ELn, return to ELn, and if resets are Exception Catch debug events, reset into ELn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No action.</td>
</tr>
<tr>
<td>1</td>
<td>Halt if allowed.</td>
</tr>
</tbody>
</table>

H3.4.7 Examples of Exception Catch debug events

If EDECCR == 0x0020, meaning that the Exception Catch debug event is enabled for Non-secure EL1, then the following exceptions generate Exception Catch debug events:
- An exception taken from Non-secure EL0 to Non-secure EL1.
- An exception return from EL2 to Non-secure EL1.
- An exception return from EL3 to Non-secure EL1.

For example, on taking a Data Abort exception from Non-secure EL0 to Non-secure EL1, using AArch64:
- ELR_EL1 and SPSR_EL1 are written with the preferred return address and PE state for a return to EL0.
- ESR_EL1 and FAR_EL1 are written with the syndrome information for the exception.
Halting Debug Events

H3.4 Exception Catch debug event

- DLR_EL0 is set to VBAR_EL1 + 0x400, the synchronous exception vector.
- DSPSR_EL0 is written with the PE state for an exit to EL1.

The following do not generate Exception Catch debug events:
- An exception taken from EL0 to EL2, in either Security state, or EL3.
- An exception return from EL2, in either Security state, to EL0.
- An exception taken from Secure EL0 to Secure EL1.
- An exception return from EL3 to Secure EL1.

H3.4.8 Pseudocode description of Exception Catch debug events

The pseudocode functions CheckExceptionCatch() and CheckPendingExceptionCatch() are described in Chapter J1 Armv8 Pseudocode.
H3.5  **External Debug Request debug event**

External Debug Request debug events are asynchronous debug events.

An External Debug Request debug event is generated when signaled by the embedded cross-trigger. See Chapter H5 *The Embedded Cross-Trigger Interface*.

--- Note ---

From the introduction of Armv8-A, the implementation of an embedded cross-trigger is required.

---

An implementation might also support IMPLEMENTATION DEFINED ways of generating an External Debug Request debug event.

### H3.5.1 Synchronization and External Debug Request debug events

An External Debug Request debug event that is asserted before a Context synchronization event is taken and the PE enters Debug state before the first instruction following the Context synchronization event completes its execution, provided that halting is allowed after completion of the Context synchronization event.

An External Debug Request debug event that is being asserted when the PE comes out of reset is taken, and the PE enters Debug state before the first instruction after the reset completes its execution, provided that halting is allowed when the PE exits reset state.

If the first instruction after the Context synchronization event or after coming out of reset generates a synchronous exception then the architecture does not define the order in which the debug event and the exception or exceptions are taken.

Otherwise, when all of the following are true, External Debug Request debug events must be taken in finite time, without requiring the synchronization of any necessary change to the external authentication interface:

- Halting is allowed.
- The PE is not in a low-power state that is not required to exit on an External Debug Request. See Core power domain power states on page H6-10336.

External Debug Request is a wake-up event for WFI, WFIT, WFE, or WFET instructions. See Mechanisms for entering a low-power state on page D1-4676.

--- Note ---

These rules are based on the rules that apply when taking asynchronous exceptions. See Asynchronous exception types on page D1-4656.

---

If an unmasked External Debug Request debug event was pending but is changed to not pending before it is taken, then the architecture permits the External Debug Request debug event to be taken, but does not require this to happen. If the External Debug Request debug event is taken then it must be taken before the first Context synchronization event after the External Debug Request debug event was changed to not pending.

Example H3-3 shows an example of the synchronization requirements.

#### Example H3-3 Synchronization requirements

Secure software locks up in a tight loop, so it executes indefinitely without any synchronization operations. An External debug request must be able to break the PE out of that loop. This is a requirement even if DBGEN or SPIDEN or both are LOW on entry to the loop, meaning that halting is prohibited, and are only asserted HIGH later.

---

### H3.5.2 Pseudocode description of External Debug Request debug events

The `ExternalDebugRequest()` function is described in Chapter J1 *Armv8 Pseudocode*.
H3.6 OS Unlock Catch debug event

An OS Unlock Catch debug event is generated when enabled and the state of the OS Lock changes from locked to unlocked. When FEAT_DoPD is implemented, CTIDEVCTL.OSUCE enables an OS Unlock Catch debug event, otherwise EDECR.OSUCE enables an OS Unlock Catch debug event.

When the OS Lock is unlocked, the PE sets EDESR.OSUC to 1 if the OS Unlock Catch debug event is enabled, and the PE is in Non-debug state, meaning the OS Unlock Catch debug event becomes pending. However, this is an indirect write to EDESR.OSUC, meaning the OS Unlock Catch debug event is not guaranteed to be taken before a subsequent Context synchronization event. If the PE enters Debug state or the OS Unlock Catch debug event is disabled before EDESR.OSUC becomes set to 1, then EDESR.OSUC might not be set.

OS Unlock Catch debug events are not generated if the OS Lock is unlocked when the PE is in Debug state. See also Synchronization and Halting debug events on page H3-10295.

EDESR.OSUC is cleared to 0 on a Warm reset and on exiting Debug state.

H3.6.1 Using the OS Unlock Catch debug event

When the Core power domain is completely off or in a low-power state, a debugger is permitted to access a debug register that is implemented in the External debug power domain. However, if a debugger attempts to access a debug register that is implemented in the Core power domain when the Core power domain registers cannot be accessed, and that access returns an error, the debugger must retry the access.

Regularly powering down the Core power domain can result in unreliable debugger behavior.

The debugger can program a Reset Catch debug event to halt the PE when it has powered up, and can program the debug registers from Debug state. However, if the PE boot software restores the debug registers, as described in Debug OS Save and Restore sequences on page H6-10342, then newly written values are overwritten by the restore sequence.

The debugger can program an OS Unlock Catch debug event to halt the PE after the restore sequence has completed, and program the debug registers from Debug state.

H3.6.2 Pseudocode description of OS Unlock Catch debug event

The CheckOSUnlockCatch() function is called when the OS Lock is unlocked.

The CheckPendingOSUnlockCatch() function is called before an instruction is executed. If an OS Unlock Catch is pending, it generates the debug event.
H3.7 Reset Catch debug events

A Reset Catch debug event is generated when enabled, and the PE exits reset state. When the Reset Catch debug event is generated, it is recorded by setting EDESR.RC to 1. When FEAT_DoPD is implemented, CTIDEVCTL.RCE enables a Reset Catch debug event, otherwise EDECR.RCE enables a Reset Catch debug event.

If halting is allowed when the event is generated, the Reset Catch debug event is taken immediately and synchronously. On entering Debug state, DLR has the address of the reset vector. The PE must not fetch any instructions from memory.

Otherwise, the Reset Catch debug event is pended and taken when halting is allowed. See Synchronization and Halting debug events on page H3-10295 for more information.

This means that EDESR.RC is set to the value of EDECR.RCE or CTIDEVCTL.RCE on a Warm reset. EDESR.RC is cleared to 0 on exiting Debug state.

H3.7.1 Pseudocode description of Reset Catch debug event

The CheckResetCatch() function is called after reset before executing any instruction.

The CheckPendingResetCatch() function is called before an instruction is executed. If a Reset Catch is pending, it generates the Reset Catch debug event.
H3.8 Software Access debug event

When the value of EDSCR.TDA is 1, software access to the following AArch64 and AArch32 debug System registers generate a Software Access debug event:

- The Breakpoint Value Registers, DBGBVR.
- The Breakpoint Control Registers, DBGBCR.
- The Watchpoint Value Registers, DBGWVR.
- The Watchpoint Control Registers, DBGWCR.

However, EDSCR.TDA is ignored if any of the following applies:

- The value of OSLSR.OSLK == 1, meaning that the OS Lock is locked.
- Halting is prohibited. See Halting allowed and halting prohibited on page H2-10233.
- The register access generates an exception.

Note

- The only accesses to the specified registers that generate a Software Access debug event are:
  - Accesses to System registers in AArch64 state.
  - Accesses to System registers in the (coproc==0b1110) encoding space in AArch32 state.

- Accesses by a PE using the external debug interface never generate a Software Access debug event.

H3.8.1 Pseudocode description of Software Access debug event

The CheckSoftwareAccessToDebugRegisters() function is described in Chapter J1 Armv8 Pseudocode.
H3.9  Synchronization and Halting debug events

The behavior of external debug depends on:

- Indirect reads of:
  - External debug registers.
  - System registers, including system debug registers.
  - Special-purpose registers.
- The state of the external authentication interface.

For some registers, all read and write accesses that update the register occur in program order, without any additional synchronization, but others require an explicit Context synchronization event. For more information on the synchronization of register updates, see:

- Synchronization requirements for AArch64 System registers on page D17-5547.
- Synchronization of changes to the external debug registers on page H8-10358.
- State and mode changes without explicit context synchronization events on page G2-9100.

Changes to the external authentication interface do not require explicit synchronization to affect External Debug Request debug events. See Synchronization and External Debug Request debug events on page H3-10291.

For changes that require explicit synchronization, it is CONSTRAINED UNPREDICTABLE whether instructions between the change and the Context synchronization event observe the old state or the new state.

This means that any change to these registers or the external authentication interface requires explicit synchronization by a Context synchronization event before the change takes effect. This ensures that for instructions appearing in program order after the change, the change affects the following:

- The generation and behavior of Breakpoint and Watchpoint debug events. See Synchronization and debug exceptions on page D2-4755 for exceptions taken from AArch64 state, or Synchronization and debug exceptions on page G2-9100 for exceptions taken from AArch32 state.
- The generation of all Halting debug events by instructions.
- Taking a pending Halting debug event or other asynchronous debug event. See:
  - Pending Halting debug events.
  - Synchronization and External Debug Request debug events on page H3-10291.
- The behavior of the Halting Step state machine. See Synchronization and the Halting Step state machine on page H3-10282.

H3.9.1  Pending Halting debug events

A Halting debug event might be pending:

1. If Halting Step of an instruction sets EDESR.SS is set to 1, and halting is prohibited following the step, then the Halting Step state machine is inactive but a Halting Step debug event is pending.
2. If a Reset Catch debug event sets EDESR.RC to 1, and halting is prohibited following reset, then a Reset Catch debug event is pending.
3. If an OS Unlock Catch debug event sets EDESR.OSUC to 1, then an OS Unlock Catch debug event is pending.
4. If FEAT_Debugv8p8 is implemented and an Exception Catch debug event sets EDESR.EC to 1, and before the PE takes the Exception Catch debug event, the PE takes an asynchronous exception, then an Exception Catch debug event is pending.

Pending Halting debug events are taken asynchronously when halting is allowed.

Pending Halting debug events are discarded by a Cold reset. The debugger can also force a pending event to be dropped by writing to EDESR.

Any Halting debug event that is observed as pending in the EDESR before a Context synchronization event is taken and the PE enters Debug state before the first instruction following the Context synchronization event completes its execution. This is possible only if halting is allowed after completion of the Context synchronization event.
If the first instruction after the **Context synchronization event** generates a synchronous exception then the architecture does not define the order in which the debug event and the exception or exceptions are taken, unless both:

- A Halting Step debug event is pending. \( \text{EDESRS} \cdot \text{SS} = 1 \).
- The **Context synchronization event** is an exception return from a state where halting is prohibited to a state where halting is allowed.

--- **Note** ---

This applies to an exception return from Secure state with \( \text{ExternalSecureInvasiveDebugEnabled()} \) == FALSE to Non-secure state with \( \text{ExternalInvasiveDebugEnabled()} \) == TRUE.

In this case the order in which the debug events are handled is specified to avoid a double-step. See [Entering the active-pending state](#) on page H3-10281.

If \( \text{FEAT Debugv8p8} \) is implemented and \( \text{EDESRE} \cdot \text{EC} \) is 1, when a pending Exception Catch debug event is taken following a **Context synchronization event**, the PE enters Debug state before fetching any instruction:

- At ELn in Non-secure state, if \( \text{EDECCR} \cdot \text{NSE}<n> \) is 1 and address translation is disabled at ELn.
- At ELn in Secure state, if \( \text{EDECCR} \cdot \text{SE}<n> \) is 1 and address translation is disabled at ELn.

If an asynchronous exception is also pending after the **Context synchronization event** then the architecture does not define the order in which the debug event and the exception or exceptions are taken.

--- **Note** ---

These rules are based on the rules that apply to taking asynchronous exceptions. See [Asynchronous exception types](#) on page D1-4656.
Chapter H4
The Debug Communication Channel and Instruction Transfer Register

This chapter describes communication between a debugger and the implemented debug logic, using the Debug Communications Channel (DCC) and the Instruction Transfer Register (ITR), and associated control flags. It contains the following sections:
• Introduction on page H4-10298.
• DCC and ITR registers on page H4-10299.
• DCC and ITR access modes on page H4-10302.
• Flow control of the DCC and ITR registers on page H4-10306.
• Synchronization of DCC and ITR accesses on page H4-10309.
• Interrupt-driven use of the DCC on page H4-10314.
• Pseudocode description of the operation of the DCC and ITR registers on page H4-10315.

Note
Where necessary, Table K17-1 on page K17-11836 disambiguates the general register references used in this chapter.
H4.1 Introduction

The **Debug Communications Channel**, DCC, is a channel for passing data between the PE and an external agent, such as a debugger. The DCC provides a communications channel between:

- An external debugger, described as the *debug host*.
- The debug implementation on the PE, described as the *debug target*.

The DCC can be used:

- As a 32-bit full-duplex channel.
- As a 64-bit half-duplex channel.

The DCC is an essential part of Debug state operation and can also be used in Non-debug state.

The **Instruction Transfer Register**, ITR, passes instructions to the PE to execute in Debug state.

The PE includes flow-control mechanisms for both the DCC and ITR.
H4.2 DCC and ITR registers

The DCC comprises data transfer registers, the DTRs, and associated flow-control flags. The data transfer registers are DTRRX and DTRTX.

The ITR comprises a single register, EDITR, and associated flow-control flags.

In AArch64 state, software can access the data transfer registers as:
- A receive and transmit pair for 32-bit full-duplex operation:
  - The write-only DBGDTRTX_EL0 register to transmit data.
  - The read-only DBGDTRRX_EL0 register to receive data.
- A single 64-bit read/write register, DBGDTR_EL0, for 64-bit half-duplex operation.
- The read/write OSDTRTX_EL1 and OSDTRRX_EL1 registers for save and restore.

In AArch32 state, software can access the data transfer registers only as:
- A receive and transmit pair, for 32-bit full-duplex operation:
  - The write-only DBGDTRTXint register to transmit data.
  - The read-only DBGDTRRXint register to receive data.
- The read/write DBGDTRTXext and DBGDTRRXext registers for save and restore.

The data transfer registers are also accessible by the external debug interface as a pair of 32-bit registers, DBGDTRRX_EL0 and DBGDTRTX_EL0. Both registers are read/write, allowing both 32-bit full-duplex and 64-bit half-duplex operation.

The DCC flow-control flags are EDSCR.{RXfull, TXfull, RXO, TXU}:
- The RXfull and TXfull ready flags are used for flow-control and are visible to software in the Debug system registers in DCCSR.
- The RX overrun flag, RXO, and the TX underrun flag, TXU, report flow-control errors.
- The flow-control flags are also accessible by software as simple read/write bits for saving and restoring over a powerdown when the OS Lock is locked in DSCR.
- The flow-control flags are accessible from the external debug interface in EDSCR.

Figure H4-1 on page H4-10300 shows the System register and external debug interface views of the EDSCR and DTR registers in both AArch64 state and AArch32 state. These figures do not include the save and restore views.
The Debug Communication Channel and Instruction Transfer Register

H4.2 DCC and ITR registers

EDITR and the ITR flow-control flags, EDSCR. [ITE, ITO] are accessible only by the external debug interface:
  - The EDITR specifies an instruction to execute in Debug state.
  - The ITR empty flag, ITE, is used for flow-control.
  - The ITR overrun flag, ITO, reports flow-control errors.

The sticky overflow flag, EDSCR.ERR, is used by both the DCC and ITR to report flow-control errors.

To save and restore the DCC registers for an external debugger over powerdown, software uses:
  - The MDSCR_EL1, OSDTRTX_EL1, and OSDTRRX_EL1 registers in AArch64 state.

Figure H4-1 System register and external debug interface views of EDSCR and DTR registers, Normal access mode

Figure H4-2 External debug interface views of EDSCR and EDITR registers, Normal access mode
The DBGDSCRext, DBGDTRTXext, and DBGDTRRXext registers in AArch32 state.

Note

There is no save and restore mechanism for the ITR registers as the ITR is used only in Debug state.

Figure H4-3 System register views of EDSR and DTR registers for save and restore
The Debug Communication Channel and Instruction Transfer Register

H4.3 DCC and ITR access modes

The DCC and ITR support two access modes:

<table>
<thead>
<tr>
<th>DCC and ITR access mode, links to description</th>
<th>Applies when:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal access mode on page H4-10302</td>
<td>EDSCR.MA == 0 or the PE is in Non-debug state</td>
</tr>
<tr>
<td>Memory access mode on page H4-10303</td>
<td>EDSCR.MA == 1 and the PE is in Debug state</td>
</tr>
</tbody>
</table>

H4.3.1 Normal access mode

The Normal access mode allows use of the DCC as a communications channel between target and host. It also allows the use of the ITR for issuing instructions to the PE in Debug state.

In Normal access mode, if there is no overrun or underrun, the following occurs:

For accesses by software:
- Direct writes to DBGDTRTX update the value in DTRTX and indirectly write 1 to TXfull.
- Direct reads from DBGDTRRX return the value in DTRRX and indirectly write 0 to RXfull.
- In AArch64 state, direct writes to DBGDTR_EL0 update both DTRTX and DTRRX, indirectly write 1 to TXfull, and do not change RXfull:
  - DTRTX is set from bits[31:0] of the transfer register.
  - DTRRX is set from bits[63:32] of the transfer register.
- In AArch64 state, direct reads from DBGDTR_EL0 return the concatenation of DTRRX and DTRTX, indirectly write 0 to RXfull, and do not change TXfull:
  - Bits[31:0] of the transfer register are set from DTRRX.
  - Bits[63:32] of the transfer register are set from DTRTX.

--- Note ---
For DBGDTR_EL0, the word order is reversed for reads with respect to writes.

Software reads TXfull and RXfull using DCCSR.

For accesses by the external debug interface:
- Writes to EDITR trigger the instruction to be executed if the PE is in Debug state:
  - If the PE is in AArch64 state, this is an A64 instruction.
  - If the PE is in AArch32 state, this is a T32 instruction. The T32 instruction is a pair of halfwords where the first halfword is taken from the lower 16-bits, and the second halfword is taken from the upper 16-bits.
- Reads of DBGDTRTX_EL0 return the value in DTRTX and indirectly write 0 to TXfull.
- Writes to DBGDTRTX_EL0 update the value in DTRTX and do not change TXfull.
- Reads of DBGDTRRX_EL0 return the value in DTRRX and do not change RXfull.
- Writes to DBGDTRRX_EL0 update the value in DTRRX and indirectly write 1 to RXfull.

TXfull and RXfull are visible to the external debug interface in EDSCR.

The PE detects overrun and underrun by the external debug interface, and records errors in EDSCR. {TXU, RXO, ITO, ERR}. See Flow control of the DCC and ITR registers on page H4-10306.

See also Synchronization of DCC and ITR accesses on page H4-10309.
H4.3.2  Memory access mode

When the PE is in Debug state, Memory access mode can be selected to accelerate word-aligned block reads or writes of memory by an external debugger. Memory access mode can be enabled only in Debug state, and no instructions can be issued directly by the debugger when in Memory access mode.

If there is no overrun or underrun when in Memory access mode, an access by the external debug interface results in the following:

- **External reads from DBGDTRTX_EL0 cause:**
  1. The existing value in DTRTX to be returned. This clears EDSCR.TXfull to 0.
  2. The equivalent of LDR W1,[X0],#4, if in AArch64 state, or LDR R1,[R0],#4, if in AArch32 state, to be executed.
  3. The equivalent of the MSR DBGDTRX_EL0,X1 instruction, if in AArch64 state, or the MCR p14,0,R1,c0,c5,0 instruction, if in AArch32 state, to be executed.
  4. EDSCR.{TXfull, ITE} to be set to {1,1}, and X1 or R1 to be set to an UNKNOWN value.

- **External writes to DBGDTRRX_EL0 cause:**
  1. The value in DTRRX to be updated. This sets EDSCR.RXfull to 1.
  2. The equivalent of the instruction MRS X1,DBGDTRRX_EL0, if in AArch64 state, or MRC p14,0,R1,c0,c5,0 if in AArch32 state, to be executed.
  3. The equivalent of the instruction STR W1,[X0],#4, if in AArch64 state, or STR R1,[R0],#4, if in AArch32 state, to be executed.
  4. EDSCR.{RXfull, ITE} to be set to {0,1}, and X1 or R1 to be set to an UNKNOWN value.

- **External reads from DBGDTRRX_EL0 return the last value written to DTRRX.**
- **External writes to EDITR generate an overrun error.**

During these accesses, EDSCR.{TXfull, RXfull, ITE} are used for flow control.

**Note**

An overrun or underrun might result in EDSCR.ERR being set to 1 asynchronously to the sequence of operations that are outlined in this section. As this is timing-dependent, it is UNPREDICTABLE when the EDSCR.ERR flag affects the instructions and therefore whether neither instruction, only the first instruction, or both instructions are executed. If the second instruction is executed, then the first instruction must have been executed. However, in each case X1 or R1 is set to an UNKNOWN value. This means that:

- In both cases, if the memory access instruction is not executed, then the base register X0 or R0 is not updated, meaning the debugger can determine the last accessed location.
- In the list describing External reads from DBGDTRTX_EL0, DTRTX and EDSCR.TXfull get set to UNKNOWN values. If the load was executed, then the value that was read by the PE is lost. This means the operation might need to be repeated by the debugger, and it is not advisable to use Memory access mode to read from read-sensitive locations using the underrun and overrun detection for flow control.
- In the list describing External writes to DBGDTRRX_EL0, EDSCR.RXfull is set to an UNKNOWN value.

A Data Abort from the memory access can also set EDSCR.ERR to 1. See Data Aborts in Memory access mode on page H4-10304.

The architecture does not require precisely when these flags are set or cleared by the sequence of operations outlined in this section. For example, in the case of an external write to DBGDTRRX_EL0, in AArch64 state, RXfull might be cleared after step 2, or it might not be cleared until after step 3, as an implementation is free to fuse these steps into a single operation. The architecture does require that the flags are set as at step 4 when the PE is ready to accept a further read or write without causing an overrun error or an underrun error.

The process outlined in this section represents a simple sequential execution model of Memory access mode. An implementation is free to pipeline, buffer, and re-order instructions and transactions, as long as the following remain true:

- Data items are transferred into and out of the DTR in order and without loss of data, other than as a result of an overrun or an underrun.
- Data Aborts occur in order.
• The constraints of the memory type are met.
• In the list describing External reads from DBGDTRTX_EL0 on page H4-10303:
  — The MSR equivalent operation at step 3 of the sequence reads the value loaded by step 2.
  — If the list is performed in a loop, for all but the first iteration of this list, the value read by step 1 returns the values written by the MSR equivalent operation at the previous iteration of step 3.
• In the list describing External writes to DBGDTRRX_EL0 on page H4-10303:
  — The MSR equivalent operation at step 2 of the sequence returns the value written at step 1.
  — The STR equivalent at step 3 of the sequence writes the value read at step 2.
• If the PE cannot accept a read or write, as applicable, during the sequence, then the flags are updated to indicate an overrun or underrun.

See Flow control of the DCC and ITR registers on page H4-10306 for more information on overrun and underrun.

Ordering, access sizes and effect on Exclusives monitors

For the purposes of memory ordering, access sizes, and effect on the Exclusives monitor, accesses in Memory access mode are consistent with load/store word instructions executed by the PE.

The simple sequential access model of Memory-access mode, as stated in Memory access mode on page H4-10303, must also be ordered with respect to instructions executed as a result of External writes to EDITR in Normal mode both before and after accesses to the DTR registers in Memory-access mode.

Data Aborts in Memory access mode

If a memory access generates a Data Abort, then:
• The Data Abort exception is taken. See Exceptions in Debug state on page H2-10265:
  — This means EDSR.ERR is set to 1, see Cumulative error flag on page H4-10308.
  — If the Data Abort occurs on stage 2 of an address translation, then the values returned in the ISV field and in bits[23:14] of the ISS are UNKNOWN.
    If this Data Abort is taken to EL2 using AArch64, the ISS is returned by ESR_EL2. ISS encoding for an exception from a Data Abort on page D17-5685 describes the usual encoding of this ISS.
    If EL2 is using AArch32 and this Data Abort is taken to Hyp mode, the ISS is returned by HSR. ISS encoding for exception from a Data Abort on page G8-9538 describes the usual encoding of this ISS.
  • Register R0 retains the address that generated the abort.
  • Register R1 is set to an UNKNOWN value.
  • EDSR.TXfull, for a load, or EDSR.RXfull, for a store, is set to an UNKNOWN value.
  • DTRTX, for a load, or DTRRX, for a store, is set to an UNKNOWN value.
  • EDSR.ITE is set to 1.

Illegal Execution state exception

If PSTATE.IL is set to 1 when EDSR.MA == 1, then on an external write access to DBGDTRRX_EL0 or an external read from DBGDTRTX_EL0, it is CONSTRAINED UNPREDICTABLE whether the PE:
• Takes an Illegal Execution state exception without performing any operations. In this case:
  — EDSR.ERR is set to 1, see Cumulative error flag on page H4-10308.
  — Register R0 is unchanged.
  — Register R1 is set to an UNKNOWN value.
  — EDSR.TXfull or EDSR.RXfull, as applicable, is set to an UNKNOWN value.
  — DTRTX or DTRRX, as applicable, is set an UNKNOWN value.
  — EDSR.ITE is set to 1.

See also Exceptions in Debug state on page H2-10265.
• Ignores PSTATE.IL.
Note

The typical usage model for Memory access mode involves executing instructions in Normal access mode to set up X0 before setting EDSCR.MA to 1. These instructions generate an Illegal state exception if PSTATE.IL is set to 1.

Alignment constraints

If the address in R0 is not aligned to a multiple of four, the behavior is as follows:

1. The PE makes an unaligned memory access to R0. If alignment checking is enabled for the memory access, this generates an Alignment fault.
2. The PE makes a memory access to Align(X[0],4) in AArch64 state, or Align(R[0],4) in AArch32 state.
3. The PE generates an Alignment fault, regardless of whether alignment checking is enabled.
4. The PE does nothing.

Following each memory access, if there is no Data Abort, R0 is updated with an UNKNOWN value.

For external writes to DBGDTRRX_EL0, if the PE writes to memory, an UNKNOWN value is written.

For external reads of DBGDTRTX_EL0 an UNKNOWN value is returned.

The RXfull and TXfull flags are left in an UNKNOWN state, meaning that a DBGDTRTX_EL0 read can trigger a TX underrun, and a DBGDTRTX_EL0 write can trigger an RX overrun.

H4.3.3 Memory-mapped accesses to the DCC and ITR

Writes to the flags in EDSCR by external debug interface accesses to the DCC and the ITR registers are indirect writes, because they are a side-effect of the access. The indirect write might not occur for a memory-mapped access to the external debug interface. For more information, see Register access permissions for memory-mapped accesses on page H8-10362.
H4.4 Flow control of the DCC and ITR registers

- **Ready flags.**
- **Buffering writes to EDITR.**
- **Overrun and underrun flags.**
- **Cumulative error flag on page H4-10308.**

### H4.4.1 Ready flags

In Normal access mode:

- For the DTR registers there are two ready flags:
  - \( \text{EDSCR.RXfull} = 1 \) indicates that \( \text{DBGDTTRRX_EL0} \) contains a valid value that has been written by the external debugger and not yet read by software running on the target.
  - \( \text{EDSCR.TXfull} = 1 \) indicates that \( \text{DBGDTTRTX_EL0} \) contains a valid value that has been written by software running on the target and not yet read by an external debugger.

- For the ITR register there is a single ready flag:
  - \( \text{EDSCR.ITE} = 1 \) indicates that the PE is ready to accept an instruction to the ITR.

**Note**

The architecture permits a PE to continue to accept and buffer instructions when previous instructions have not completed their architecturally defined behavior, as long as those instructions are discarded if \( \text{EDSCR.ERR} \) is set, either by an underrun or overrun or by any of the other error conditions described in this architecture, such as an instruction generating an abort.

In Memory access mode:

- \( \text{EDSCR.\{RXfull, ITE\}} = \{0, 1\} \) indicates that \( \text{DBGDTTRRX_EL0} \) is empty and the PE is ready to accept a word external write to \( \text{DBGDTTRRX_EL0} \).
- \( \text{EDSCR.\{TXfull, ITE\}} = \{1, 1\} \) indicates that \( \text{DBGDTTRTX_EL0} \) is full and the PE is ready to accept a word external read from \( \text{DBGDTTRTX_EL0} \).

All other values indicate that the PE is not ready, and result in a DTR overrun or underrun error, an ITR overrun error, or both, as defined in Overrun and underrun flags.

\( \text{EDSCR.\{ITE, RXfull, TXfull\}} \) shows the status of the ITR and DCC registers. It ignores the question of whether a read or write cannot be accepted because, for example, \( \text{EDSCR.ERR} \) is set or the OPTIONAL Software Lock is locked for memory-mapped accesses (\( \text{EDLSR.SLK} = 1 \)).

### H4.4.2 Buffering writes to EDITR

The architecture permits a processor to continue to accept and buffer instructions when previous instructions have not completed their architecturally defined behavior, provided that:

- Those instructions are discarded if \( \text{EDSCR.ERR} \) is set to 1, either by an underrun or an overrun, or by any other error conditions described in this architecture, such as an instruction generating an abort.

- The PE maintains the simple sequential execution model with the order of instructions determined by the order in which the PE accepts the EDITR writes. In particular, the buffered instructions must be executed in the Execution state consistent with a simple sequential execution of the instructions, even if one of the previous instructions is a state changing operation, such as \( \text{DCPS} \) or \( \text{DRPS} \).

### H4.4.3 Overrun and underrun flags

Each of the ready flags has a corresponding overrun or a corresponding underrun flag. These are sticky status flags that are set if the register is accessed using the external debug interface when the corresponding ready flag is not in the ready state.

If the PE is in Debug state and Memory access mode, the corresponding error flag is also set if the PE is not ready to accept an operation because a previous load or store is still in progress. The sticky status flag remains set until cleared by writing 1 to \( \text{EDCR.CSE} \).
The architecture permits a PE to continue to accept and buffer data to write to memory in Memory access mode.

Table H4-1 on page H4-10307 shows DCC and ITR ready flags and the overrun and underrun flags associated with them.

<table>
<thead>
<tr>
<th>External debug interface access</th>
<th>Overrun/Underrun condition</th>
<th>EDSCR flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write DBGDTRRX_EL0</td>
<td>EDSCR.RXfull == ‘1’</td>
<td></td>
</tr>
<tr>
<td>Read DBGDTRTX_EL0</td>
<td>EDSCR.TXfull == ‘0’</td>
<td></td>
</tr>
<tr>
<td>Write EDITR</td>
<td>Halted() &amp;&amp; (EDSCR.ITE == ‘0’</td>
<td></td>
</tr>
</tbody>
</table>

When an overrun or underrun flag is set to 1, the cumulative error flag, EDSCR.ERR, described in *Cumulative error flag* on page H4-10308, is also set to 1.

In the event of an external write to DBGDTRRX_EL0 or EDITR generating an overrun, or an external read from DBGDTRTX_EL0 generating an underrun:
- For a write, the written value is ignored.
- For a read, an UNKNOWN value is returned.
- EDSCR.TXfull, EDSCR.RXfull or EDSCR.ITE, as applicable, are not updated.

There is no overrun or underrun detection on external reads of DBGDTRRX_EL0 or external writes of DBGDTRTX_EL0.

There is no overrun or underrun detection of direct reads and direct writes of the DTR System registers by software:
- If RXfull == 0, a direct read of DBGDTRRX or DBGDTR_EL0 returns UNKNOWN.
- If TXfull == 1, a direct write of:
  - DBGDTRTX sets DTRTX to UNKNOWN.
  - DBGDTR_EL0 sets DTRRX and DTRTX to UNKNOWN.

See *DCC accesses in Non-debug state* on page H4-10310 for more information.

### Accessing 64-bit data

In AArch64 state, a software access to the DBGDTR_EL0 register and an external debugger access to both DBGDTRRX_EL0 and DBGDTRTX_EL0 can perform a 64-bit half-duplex operation.

However, there is only overrun and underrun detection on one of the external debug registers. That is:
- If software directly writes a 64-bit value to DBGDTR_EL0, only TXfull is set to 1, meaning:
  - A subsequent external write to DBGDTRRX_EL0 would not be detected as an overrun.
  - If the external debugger reads DBGDTRTX_EL0 first, software might observe MDCCSR_EL0.TXfull == 0 and send a second value before the external debugger reads DBGDTRRX_EL0, leading to an undetected overrun.
- On external writes to both DBGDTRRX_EL0 and DBGDTRTX_EL0, only RXfull is set to 1, meaning:
  - A subsequent direct write of DBGDTRTX_EL0 would not be detected as an overrun.
  - If the external debugger writes to DBGDTRRX_EL0 first, software might observe MDCCSR_EL0.RXfull == 1 and read a full 64-bit value, before the external debugger writes to DBGDTRTX_EL0, leading to an undetected underrun.

To avoid this, debuggers need to be aware of the data size used by software for transfers and ensure that 64-bit data is read or written in the correct order. If the PE is in Non-debug state, this order is as follows:
- The external debugger must check EDSCR.{RXfull, TXfull} before each transfer.
To receive a 64-bit value from the target, the external debugger must read **DBGDTRRX_EL0** before reading **DBGDTRTX_EL0**.

To send a 64-bit value to the target, the external debugger must write to **DBGDTRTX_EL0** before writing **DBGDTRRX_EL0**.

Because three accesses are required to transfer 64 bits of data, 64-bit transfers are not recommended for regular communication between host and target. The use of underrun and overrun detection means that only one access is required for 32 bits of data when using 32-bit transfers.

In Debug state, the debugger controls the instructions executed by the PE, so these limitations do not apply. 64-bit transfers provide a means to transfer a 64-bit general register between the host and the target in Debug state.

### H4.4.4 Cumulative error flag

The cumulative error flag, **EDSCR.ERR**, is set to 1:
- On taking an exception from Debug state.
- On any signaled overrun or underrun in the DCC or ITR.

When **EDSCR.ERR == 1**:
- External reads of **DBGDTRTX_EL0** do not have any side-effects.
- External writes to **DBGDTRRX_EL0** are ignored.
- External writes to **EDITR** are ignored.
- No further instructions can be issued in Debug state. This includes any instructions previously accepted as external writes to **EDITR** that occur in program order after the instruction or access that caused the error.

This allows a debugger to stream data, or, in Debug state, instructions, to the target without having to:
- Check **EDSCR.\{RXfull, TXfull, ITE\}** before each access.
- Check **EDSCR.\{ITO, RXO, TXU\}** following each access, for overrun or underrun.
- Check **PSTATE** or other syndrome registers, or both, for an exception following each instruction executed in Debug state that might generate a synchronous exception.

The cumulative error flag remains set until cleared to 0 by writing 1 to **EDRCR.CSE**. However, the effect of writing 1 to **EDRCR.CSE** to clear **EDSCR.ERR** is **CONSTRAINED UNPREDICTABLE** when both of the following apply:
- The PE is in Debug state.
- The value of **EDSCR.ITE** is 0.

When these conditions apply and a value of 1 is written to **EDRCR.CSE**, either or both of the following might occur:
- **EDSCR.ERR** is not cleared to 0.
- Any instructions in **EDITR** that have not been executed might be executed subsequently, rather than being ignored.

--- Note ---

This means that a debugger must poll **EDSCR.ITE** until it has the value 1, indicating that **EDITR** is empty, before writing to **EDRCR.CSE** to clear the **EDSCR.ERR** flag to 0.

---

For overruns and underruns, **EDSCR.\{ITO, RXO, TXU\}** record the error type.

### Pseudocode description of clearing the error flag

The **ClearStickyErrors()** pseudocode function is described in Chapter J1 *Armv8 Pseudocode*. 

---
H4.5 Synchronization of DCC and ITR accesses

In addition to the standard synchronization requirements for register accesses, the following subsections describe additional requirements that apply for the DCC and ITR registers:

- Summary of System register accesses to the DCC.
- DCC accesses in Non-debug state on page H4-10310.
- Synchronization of DCC interrupt request signals on page H4-10312.
- DCC and ITR access in Debug state on page H4-10313.

In these sections, accesses by the external debug interface are referred to as external reads and external writes. Accesses to System registers are referred to as direct reads, direct writes, indirect reads, and indirect writes.

Note

In Synchronization requirements for AArch64 System registers on page D17-5547 external reads and external writes are described as forms of indirect access. This whole section uses more explicit terminology.

The DTR registers and the DCC flags, TXfull and RXfull, form a communication channel, with one end operating asynchronously to the other. Implementations must respect the ordering of accesses to these registers in order to maintain the correct behavior of the channel.

External reads of, and external writes to DBGDTRRX_EL0 and DBGDTRTX_EL0 are asynchronous to direct reads of, and direct writes to, DBGDTRRX, DBGDTRTX, and in AArch64 state DBGDTR_EL0, made by software using System register access instructions. The direct reads and direct writes indirectly write to the DCC flags. The external reads and external writes indirectly read the DCC flags to check for underrun and overrun.

Throughout this section:

**DCC flags**

- The EDSCR.{RXfull.TXfull} ready flags.
- The EDSCR.RXO overrun flag.
- The EDSCR.TXU underrun flag.
- The EDSCR.ERR cumulative error flag.

**ITR flags**

- The EDSCR.ITE ready flag.
- The EDSCR.ITO overrun flag.
- The EDSCR.ERR cumulative error flag.

H4.5.1 Summary of System register accesses to the DCC

System register accesses to the DTR registers are direct reads and writes of those registers, as shown in Table H4-2 on page H4-10310. Several of these instructions access the same registers using different encodings.

DBGDTRRX_EL0 and DBGDTRTX_EL0 are encoded as MRS and MSR accesses respectively to the same System register, even though they access different underlying register values. DBGDTRRX and DBGDTRTX are similarly encoded as MRC and MCR accesses respectively to the same System register. The encoding means that direct reads and writes using these encodings must be ordered with respect to each other. For more information, see Synchronization requirements for AArch64 System registers on page D17-5547 and Synchronization of changes to AArch32 System registers on page G8-9326.
Table H4-2 on page H4-10310 shows a summary of System register accesses to the DCC.

<table>
<thead>
<tr>
<th>Operation</th>
<th>OS Lock</th>
<th>AArch64 (MRS/MSR)</th>
<th>AArch32 (MRC/MCR)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>-</td>
<td>DBGDTRRX_EL0</td>
<td>DBGDTRRXint</td>
<td>Direct read of DTRRX. Indirect write to the DCC flags. An STC instruction that reads DBGDTRRXint makes an indirect write to DBGDSCRint.RXfull.</td>
</tr>
<tr>
<td>Write</td>
<td>-</td>
<td>DBGDTRTX_EL0</td>
<td>DBGDTRTXint</td>
<td>Direct read of DTRTX. Indirect write to the DCC flags. An LDC instruction that writes to DBGDTRTXint using a value read from memory is a direct write to DBGDTRTXint.</td>
</tr>
<tr>
<td>Read/write</td>
<td>-</td>
<td>DBGDTR_EL0</td>
<td>-</td>
<td>Direct read/write of both DTRRX and DTRTX. Indirect write to the DCC flags.</td>
</tr>
<tr>
<td>Read</td>
<td>-</td>
<td>MDCCSR_EL0</td>
<td>DBGDSCRint</td>
<td>Direct read of the DCC flags.</td>
</tr>
<tr>
<td>Read/write</td>
<td>-</td>
<td>OSDTRRX_EL1</td>
<td>DBGDTRRXext</td>
<td>Direct read/write of DTRRX.</td>
</tr>
<tr>
<td>Read/write</td>
<td>-</td>
<td>OSDTRTX_EL1</td>
<td>DBGDTRTXext</td>
<td>Direct read/write of DTRTX.</td>
</tr>
<tr>
<td>Read</td>
<td>Unlocked</td>
<td>MDSCR_EL1</td>
<td>DBGDSCRext</td>
<td>Direct read of DCC flags.</td>
</tr>
<tr>
<td>Read/write</td>
<td>Locked</td>
<td>MDSCR_EL1</td>
<td>DBGDSCRext</td>
<td>Direct read/write of DCC flags.</td>
</tr>
</tbody>
</table>

**Table H4-2 Summary of System register accesses to the DCC**

**H4.5.2 DCC accesses in Non-debug state**

In Non-debug state DCC accesses are as described in *Normal access mode* on page H4-10302:

- If a direct read of DCCSR returns RXfull == 1, then a following direct read of DBGDTRRX, or in AArch64 state of DBGDTR_EL0, returns valid data and indirectly writes 0 to DCCSR.RXfull as a side-effect.
- If a direct read of DCCSR returns TXfull == 0, then a following direct write to DBGDTRTX, or in AArch64 state to DBGDTR_EL0, writes the intended value, and indirectly writes 1 to DCCSR.TXfull as a side-effect.

No *Context synchronization event* is required between these two instructions. Overrun and underrun detection prevents intervening external reads and external writes affecting the outcome of the second instruction.

The indirect write to the **DCC flags** as part of the DTR access instruction is made atomically with the DTR access.

Because a direct read of DBGDTRRX is an indirect write to DCCSR.RXfull, it must occur in program order with respect to the direct read of DCCSR, meaning it must not return a speculative value for DTRRX that predates the RXfull flag returned by the read of DCCSR. The direct write to DBGDTRTX must not be executed speculatively.

Direct reads of DBGDTRRX, or in AArch64 state DBGDTR_EL0, and DCCSR, must occur in program order with respect to other direct reads of the same register using the same encoding.

The following accesses have an implied order within the atomic access:

- In the simple sequential execution of the program the indirect write of the **DCC flags** occurs immediately after the direct DTR access.

**Note**

For an access to DBGDTR_EL0, this means the indirect write happens after both DBGDTRRX_EL0 and DBGDTRTX_EL0 have been accessed.

- In the simple sequential execution model, for an external read of DBGDTRTX_EL0 or an external write of DBGDTRRX_EL0:
  - The check of the **DCC flags** for overrun or underrun occurs immediately before the access.
If there is no underrun or overrun, the update of the DCC flags occurs immediately after the access.

If there is underrun or overrun, the update of the DCC underrun or overrun flags occurs immediately after the access.

All observers must observe the same order for accesses.

--- Note ---

These requirements do not create order where order does not otherwise exist. It applies only for ordered accesses.

Without explicit synchronization following external writes and external reads:

- The value written by the external write to DBGDTRRX_EL0 that does not overrun, must be observable to direct reads of DBGDTRRX and DBGDTR_EL0 in finite time.
- The DCC flags that are updated as a side-effect of the external write or external read must be observable:
  - To subsequent external reads of EDSCR.
  - To subsequent external reads of DBGDTRRX_EL0 when checking for underrun.
  - To subsequent external writes to DBGDTRTX_EL0 when checking for overrun.
  - To direct reads of DCCSR in finite time.

However, explicit synchronization is required to guarantee that a direct read of DCCSR returns up-to-date DCC flags. This means that if a signal is received from another agent that indicates that DCCSR must be read, an ISB is required to ensure that the direct read of DCCSR occurs after the signal has been received. This also synchronizes the value in DBGDTRRX, if applicable. However, if that signal is an interrupt exception triggered by COMMIRQ, COMMTX, or COMMRX, the exception entry is sufficient synchronization. See Synchronization of DCC interrupt request signals on page H4-10312.

Explicit synchronization is required following a direct read or direct write:

- To ensure that a value directly written to DBGDTRTX is observable to external reads of DBGDTRTX_EL0.
- To ensure that a value directly written to DBGDTR_EL0 is observable to external reads of DBGDTRTX_EL0 and DBGDTRRX_EL0.
- To guarantee that the indirect writes to the DCC flags that were a side-effect of the direct read or direct write have occurred, and therefore that the updated values are:
  - Observable to external reads of EDSCR.
  - Observable to external reads of DBGDTRRX_EL0 when checking for underrun.
  - Observable to external writes of DBGDTRTX_EL0 when checking for overrun.
  - Returned by a following direct read of DCCSR.

See also Memory-mapped accesses to the DCC and ITR on page H4-10305 and Synchronization of changes to the external debug registers on page H8-10358.

--- Note ---

These ordering rules mean that software:

- Must not read DBGDTRRX without first checking DCCSR.RXfull or if the previously-read value of DCCSR.RXfull is 0.
  It is not sufficient to read both registers and then later decide whether to discard the read value, as there might be an intervening write from the external debug interface.
- Must not write DBGDTRTX without first checking DCCSR.TXfull or if the previously-read value of DCCSR.TXfull is 1.
  The write to DBGDTRTX overwrites the value in DTRTX, and the external debugger might or might not have read this value.
- Must ensure there is an explicit Context synchronization event following a DTR access, even if not immediately returning to read DCCSR again. This synchronization operation can be an exception return.
Derived requirements

The rules for DCC accesses in Non-debug state are as follows:

- Following a direct read of DBGDTRRX when RXfull is 1:
  - If an external write to DBGDTRRX checks the RXfull flag for overrun and observes that the value of RXfull is 0, the value returned by the previous direct read must not be affected by the external write.
  - If an external read of EDSCR returns a RXfull value of 0, then the value returned by the previous direct read must not be affected by a following external write to DBGDTRRX, and the following external write does not overrun.

- Following a direct read of DBGDTR_EL0, when RXfull is 1:
  - If an external write to DBGDTRRX checks the RXfull flag for overrun and observes that the value of RXfull is 0, the value returned by the previous direct read must not be affected by the external write nor by a following direct write to DBGDTRTX.
  - If an external read of EDSCR returns a RXfull value of 0, then the value returned by the previous direct read must not be affected by subsequent external writes to DBGDTRRX and DBGDTRTX in any order, and the following external write of DBGDTRRX will not overrun.

- Following a direct write to DBGDTRTX, when TXfull is 0:
  - If an external read of DBGDTRTX checks the TXfull flag for underrun and observes that the value of TXfull is 1, the value returned by the external read must be the value written by the previous direct write.
  - If an external read of EDSCR returns a TXfull value of 1, then the value returned by a following external read of DBGDTRRX must be the value written by the previous direct read, and the subsequent external read will not underrun.

- Following a direct write to DBGDTR_EL0, when TXfull is 0:
  - If an external read of DBGDTRTX checks the TXfull flag for underrun and observes that the value of TXfull is 1, the values returned by the external read and by a subsequent external read of DBGDTRRX must be the value written by the previous direct write.
  - If an external read of EDSCR returns a TXfull value of 1, then the value returned by subsequent external reads of DBGDTRRX and DBGDTRTX, in any order, must be the value written by the previous direct read, and the subsequent external read of DBGDTRTX does not underrun.

- Following an external read of DBGDTRTX that does not underrun, if a direct read of DCCSR returns a TXfull value of 0, then the value returned by the external read must not be affected by a following direct write to DBGDTRTX.

- Following a first external read DBGDTRRX and a following second external read of DBGDTRTX that does not underrun, if a direct read of DCCSR returns a TXfull value of 0, then the values returned by the external reads must not be affected by a following direct write to DBGDTR_EL0.

- Following an external write to DBGDTRRX that does not overrun, if a direct read of DCCSR returns an RXfull value of 1, then the value returned by a following direct read of DBGDTRRX or DBGDTR_EL0 must be the value written by the previous external write.

- Following a first external write to DBGDTRTX and a following second external write to DBGDTRRX that does not overrun, if a direct read of DCCSR returns an RXfull value of 1, then the value returned by a subsequent direct read of DBGDTR_EL0 must return the values written by the previous external writes.

H4.5.3 Synchronization of DCC interrupt request signals

Following an external read or external write access to the DTR registers, the interrupt request signals, COMMIRQ, COMMTX, and COMMRX, must be updated in finite time without explicit synchronization.

The updated values must be observable to a direct read of DCCSR or DBGDTRRX, or a direct write of DBGDTRTX executed after taking an interrupt exception generated by the interrupt request. The updated values must also be observable to a direct write of DBGDTRTX executed after taking an interrupt exception generated by the interrupt request.
H4.5 Synchronization of DCC and ITR accesses

Note

The requirement that indirect writes to registers are observable to direct reads in finite time does not imply that all observers will observe the indirect write at the same time. For more information, see Synchronization requirements for AArch64 System registers on page D17-5547 and Synchronization of changes to AArch32 System registers on page G8-9326.

Following a direct read of DBGDTRRX or a direct write to DBGDTRRX, software must execute a Context synchronization event to guarantee the interrupt request signals have been updated in finite time. This synchronization operation can be an exception return.

H4.5.4 DCC and ITR access in Debug state

In Debug state, stricter observability rules apply for instructions issued through the ITR, to maintain communication between a debugger and the PE, without requiring excessive explicit synchronization.

In Normal access mode, without explicit synchronization:

- A direct read or direct write of the DTR registers by an instruction written to EDITR must be observable to an external write or an external read in finite time:
  - A direct read of DBGDTRRX must be observable to an external write of DBGDTRRX_EL0.
  - A direct read of DBGDTR_EL0 must be observable to an external write of DBGDTRRX_EL0 and DBGDTRTX_EL0.
  - A direct write of DBGDTRTX must be observable to an external read of DBGDTRTX_EL0.
  - A direct write of DBGDTR_EL0 must be observable to an external read of DBGDTRRX_EL0 and DBGDTRTX_EL0.

This includes the indirect write to the DCC flags that occurs atomically with the access as described in DCC accesses in Non-debug state on page H4-10310.

The subsequent external write or external read must observe either the old or the new values of both the DTR contents and DCC flags. If the old values are observed, this typically results in overrun or underrun, assuming the old values of the DCC flags indicate an overrun or underrun condition, as would normally be the case. This means the debugger can observe the direct read or direct write without explicit synchronization and without explicitly testing the DCC flags in EDSCR, because it can rely on overrun and underrun tests.

- External reads of DBGDTRTX_EL0 that do not underrun and external writes to DBGDTRRX_EL0 that do not overrun must be observable to an instruction subsequently written to EDITR on completion of the first external access. This includes the indirect write to the DCC flags.

This means that without explicit synchronization and without the need to first check the DCC flags in DCCSR:

- If the instruction is a direct read of DBGDTRRX, it observes the external write.
- If the instruction is a direct write of DBGDTRTX, it observes the external read.

- Writes to EDITR that do not overrun commit an instruction for execution immediately. The instruction must complete execution in finite time without requiring any further operation by the debugger.

- After an external write to the EDITR, the ITR flags that are updated as a side effect of that write must be observable by:
  - An external read of the EDSCR that follows the external write to the EDITR.
  - When checking for overrun, another external write to the EDITR that follows the original external write to the EDITR.

In Memory access mode, these requirements shift to the instructions implicitly executed by external reads and external writes of the DTR registers, as described in Memory access mode on page H4-10303.
H4.6 Interrupt-driven use of the DCC

Arm recommends implementations provide a level-sensitive DCC interrupt request through the IMPLEMENTATION DEFINED interrupt controller as a private peripheral interrupt for the originating PE.

--- Note ---

- In addition to connection to the interrupt controller Arm also recommends COMMIRQ, COMMTX, and COMMRX signals that might be implemented for use by any legacy system peripherals.
- GICv3 reserves a private peripheral interrupt number for the COMMIRQ interrupt.

The DCCINT register provides a first level of interrupt masking within the PE, meaning only a single interrupt source, COMMIRQ, is needed at the interrupt controller.

See also Synchronization of DCC interrupt request signals on page H4-10312.
H4.7 Pseudocode description of the operation of the DCC and ITR registers

The basic operation of the DCC and ITR registers is shown by the following pseudocode functions. These functions do not cover the behavior when OSLSR.OSLK == 1, meaning that the OS Lock is locked:

- DBGDTR_EL0[].
- DBGDTRRX_EL0[].
- DBGDTRTX_EL0[].
- EDITR[].
- CheckForDCCInterrupts().

For the definition of the DTR Registers, see shared/debug/dccanditr/DTR on page J1-11366.
The Debug Communication Channel and Instruction Transfer Register
H4.7 Pseudocode description of the operation of the DCC and ITR registers
Chapter H5
The Embedded Cross-Trigger Interface

This chapter describes the embedded cross-trigger interface. It contains the following sections:

- About the Embedded Cross-Trigger on page H5-10318.
- Basic operation on the ECT on page H5-10320.
- Cross-triggers on a PE in an Arm A-profile implementation on page H5-10324.
- Description and allocation of CTI triggers on page H5-10325.
- CTI registers programmers’ model on page H5-10329.
- Examples on page H5-10330.
H5.1 About the Embedded Cross-Trigger

The Embedded Cross-Trigger (ECT) allows a debugger to:

- Send trigger events to a PE. For example, this might be done to halt the PE.
- Send a trigger event to one or more PEs, or other system components, when a trigger event occurs on another PE or system component. For example, this might be done to halt all PEs when one individual PE halts.

Figure H5-1 shows the logical structure of an ECT.

The ECT can deliver many types of trigger events, which are described in the following sections:

- Debug request trigger event on page H5-10326.
- Restart request trigger event on page H5-10326.
- Cross-halt trigger event on page H5-10326.
- Performance Monitors overflow trigger event on page H5-10326.
- Generic trace external input trigger events on page H5-10327.
- Generic trace external output trigger events on page H5-10327.
- Generic CTI interrupt trigger event on page H5-10327.

From the introduction of Armv8, an A-profile implementation must:

- Include a cross-trigger interface, CTI.
- Implement at least the input and output triggers defined in this architecture.

In addition, see Cross-triggers on a PE in an Arm A-profile implementation on page H5-10324.

Arm recommends that this cross-trigger interface includes:

- The ability to route trigger events between Trace Units, which typically have advanced event triggering logic.
- An output trigger to the interrupt controller.

Also, Arm recommends that the Embedded Cross-Trigger includes the capability to send and receive IMPLEMENTATION DEFINED system trigger events to and from other system components, including a system counter, using a system CTI. See Halt-on-debug on page I2-10726.

Note

The ECT and CTI must only signal trigger events for external debugging. They must not route software events, such as interrupts. For example, the Performance Monitors overflow input trigger is provided to allow entry to Debug state on a counter overflow, and the output trigger to the interrupt controller is provided to generally allow events from the external debug sub-system to be routed to a software agent. However, the combination of the two must not be used as a mechanism to route Performance Monitors overflows to an interrupt controller.
H5.1 About the Embedded Cross-Trigger

Note

CTI version 1 (CTIv1) is defined by the CoreSight™ SoC Technical Reference Manual. CTIv2 extends CTIv1 with the addition of the input channel gate. See Implementation with CTIv2.

H5.1.1 Implementation with a CoreSight CTI

For details of the recommended connections for an A-profile implementation, see Appendix K6 Recommended External Debug Interface. See also CoreSight™ SoC Technical Reference Manual.

H5.1.2 Implementation with CTIv2

If the CTI implemented is CTIv2 then:

- The CTIDEVARCH, CTIDEVAFF0, and CTIDEVAFF1 registers must be implemented.
- If the channel gate function is implemented, it applies to both input and output channels.
- The input channel gate function must be implemented if either of the following is true:
  - The CTM is implemented and the architecture variant is Armv8.5 or higher.
  - The CTIDEVARCH.REVISION field reads as 0b0001 or higher.

Implementation of CTIv2 features in architecture variants below Armv8.5 is OPTIONAL, but Arm recommends that CTIv2 is implemented, CTIv2 must be implemented from Armv8.5.
H5.2 Basic operation on the ECT

The ECT comprises a Cross-Trigger Matrix, CTM, and one Cross-Trigger Interface, CTI, for each PE. The ECT might also include other CTIs for other system components. The CTM passes events between the CTI blocks over channels. The CTM can have a maximum of 32 channels.

The main interfaces of the cross-trigger interface, CTI, are:

- The input triggers:
  - These are trigger event inputs from the PE to the CTI.
- The output triggers:
  - These are trigger event outputs from the CTI to the PE.
- The input channels:
  - These are channel event inputs from the cross-trigger matrix, CTM, to the CTI.
- The output channels:
  - These are channel event outputs from the CTI to the CTM.

Each CTI block has:

- Up to 32 input triggers that come from the PE:
  - The input triggers are numbered 0-31.
- Up to 32 output triggers that go to the PE:
  - The output triggers are numbered 0-31.

If the CTI is not powered up when the Core power domain is powered up, the CTI ignores all input triggers and input channel events, and does not generate any output triggers or output channel events.

Figure H5.2 on page H5-10321 shows the logical internal structure of a CTI.
Note

- The number of triggers in IMPLEMENTATION DEFINED. Figure H5-2 shows eight input and eight output triggers.
- The number of channels is IMPLEMENTATION DEFINED. Figure H5-2 shows four channels.
- In Figure H5-2 the input channel gate function is a CTIv2 feature.

When the CTI receives an input trigger event, this generates channel events on one or more internal channels, according to the mapping function defined by the Input trigger→output channel mapping registers, CTIINEN[n].

The CTI also contains an application trigger and channel pulse to allow a debugger to create channel events directly on internal channels by writing to the CTI control registers.
Channel events on each internal channel are passed to a corresponding output channel that is controlled by a channel gate. The channel gate can block propagation of channel events from an internal channel to an output channel.

--- Note ---
If the CTM is implemented:
• The gate function must be implemented.
• If the CTI is CTIv1, the gate function applies to output triggers only.

The output channels from a CTI are combined, using a logical OR function, with the output channels from all other CTIs to form the input channels on other CTIs. The input channels of this CTI are the logical OR of the output channels on all other CTIs. This is the cross-trigger matrix, CTM. Therefore, the number of input channels must equal the number of output channels.

--- Note ---
The number of input triggers and output triggers is not required to be the same.

The internal channels form an internal cross-trigger matrix within the CTI. This delivers events directly from the input triggers to the output triggers. Therefore the number of internal channels is the same as the number of input and output channels on the external CTM, and there is a direct mapping between the two.

Channel events received on each input channel are passed to the corresponding internal channel. It is IMPLEMENTATION DEFINED whether the cross-trigger gate also blocks propagation of channel events from input channels to internal channels.

--- Note ---
If CTIv2 is implemented, the cross-trigger gate also blocks propagation of channel events from input channels to internal channels.

When the CTI receives a channel event on an internal channel this generates trigger events on one or more output triggers, according to the mapping function defined by the Input channel \(\rightarrow\) output trigger mapping registers, CTIOUTEN\(<n>\).

The CTI contains the input and output trigger interfaces to the PE and the interface of the cross-trigger matrix. The architecture does not define the signal protocol used on the trigger interfaces, and:
• It is IMPLEMENTATION DEFINED whether the CTI supports multicycle input trigger events.
• It is IMPLEMENTATION DEFINED whether the CTM supports multicycle channel events.

See Multicycle events.

However, an output trigger is asserted until acknowledged. The output trigger can be:
• Self-acknowledging. This means that no further action is required from the debugger.
• Acknowledged by the debugger writing 1 to the corresponding bit of CTIINTACK.

The time taken to propagate a trigger event from the first PE, through its CTI, across the CTM to another CTI, and thereby to a second PE is IMPLEMENTATION DEFINED.

--- Note ---
Arm recommends that this path is not longer than the shortest software communication path between those PEs. This is because if the first PE halts, the Cross-halt trigger event can propagate through the ECT and halt the second PE without causing software on the second PE to malfunction because the first PE is in Debug state and is not responding.

### H5.2.1 Multicycle events

A multicycle event is one with a continuous state that might persist over many cycles, as opposed to a discrete event. A typical implementation of a multicycle event is a level-based signal interface, whereas a discrete event might be implemented as a pulse signal or message.
CTI support for multicycle trigger events is IMPLEMENTATION DEFINED. Use of multicycle trigger events is deprecated. Of the architecturally defined input trigger events, the Performance Monitors overflow trigger event and Generic trace external output trigger events can be multicycle input triggers.

CTM support for multicycle channel events is IMPLEMENTATION DEFINED. A CTM that does not support multicycle channel events cannot propagate a multicycle trigger event between CTIs.

**Note**

A full ECT might comprise a mix of CTIs, some of which can support multicycle trigger events. In bridging these components, multicycle channel events become single channel events at the boundary between the CTIs.

---

**An ECT that supports multicycle trigger events**

When an ECT supports multicycle trigger events, an input trigger event to the CTI continuously asserts channel events on all output channels mapped to it until either:

- The input trigger event is removed.
- The channel mapping function is disabled.

This means that an input trigger that is asserted for multiple cycles causes any channels that are mapped to it to become active for multiple cycles. Consequently, any output triggers mapped from that channel are asserted for multiple cycles.

**Note**

The output trigger remains asserted for at least as long as the channel remains active. This means that even if the output trigger is acknowledged, it remains asserted until the channel deactivates.

---

The CTI does not guarantee that these events have precisely the same duration, as the triggers and channels can cross between clock domains.

**CTIAPPSET** and **CTIAPPCLEAR** can set a channel active for multiple cycles. **CTIAPPULSE** generates a single channel event. **CTICHINSTATUS** and **CTICHOUTSTATUS** can report whether a channel is active.

---

**An ECT that does not support multicycle trigger events**

When an ECT does not support multicycle trigger events, an input trigger event to the CTI generates a single channel event on all output channels mapped to it, regardless of how long the input trigger event is asserted.

This means that an input trigger event that is asserted for multiple cycles generates a single channel event on any channels mapped to it. Consequently any self-acknowledging output triggers mapped from those channels are single trigger events.

**Note**

A single event is typically a single cycle, but there is no guarantee that this is always the case.

---

**CTIAPPSET** and **CTIAPPCLEAR** can only generate a single channel event. **CTIAPPULSE** generates a single channel event. If the ECT does not support multicycle channel events, use of **CTIAPPSET** and **CTIAPPCLEAR** is deprecated, and the debugger must only use **CTIAPPULSE**. **CTICHINSTATUS** and **CTICHOUTSTATUS** must be treated as **UNKNOWN**.
H5.3 Cross-triggers on a PE in an Arm A-profile implementation

An A-Profile PE must include a cross-trigger interface, and the implementation must include at least the input and output triggers defined in this architecture. The number of channels in the cross-trigger matrix is IMPLEMENTATION DEFINED, but there must be a minimum of three. Software can read CTIDEVID.NUMCHAN to discover the number of implemented channels.

The CTM must connect to all PEs in the same Inner Shareability domain as the PE that includes the cross-trigger interface, but can also connect to additional PEs. Arm strongly recommends that the CTM connects all PEs implementing a CTI in the system. This includes other PEs that can be connected using a CoreSight CTI module.

Note

In a uniprocessor system the CTM is OPTIONAL. In a multiprocessor system the CTM is required. The CTM might be connected other CTI modules for non-PEs, such as triggers for system visibility components. Arm recommends that the CTM is implemented.

Any CTI connected to a PE that is not an Arm PE must implement at least:

- The Debug request trigger event.
- The Restart trigger event.
- The Cross-halt trigger event.

For more information about the CTI, see the CoreSight™ SoC Technical Reference Manual. Arm architecture refines the generic CTI by defining roles for each of the implemented input and output triggers.
H5.4 Description and allocation of CTI triggers

Table H5-1 on page H5-10325 shows the output trigger events defined by the architecture and the related trigger numbers.

Table H5-1 Allocation of CTI output trigger events

<table>
<thead>
<tr>
<th>Number</th>
<th>Source</th>
<th>Destination</th>
<th>Event description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CTI</td>
<td>PE</td>
<td>Debug request trigger event on page H5-10326</td>
</tr>
<tr>
<td>1</td>
<td>CTI</td>
<td>PE</td>
<td>Restart request trigger event on page H5-10326</td>
</tr>
<tr>
<td>2</td>
<td>CTI</td>
<td>IRQ controller</td>
<td>Generic CTI interrupt trigger event on page H5-10327</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>4 - 7</td>
<td>CTI</td>
<td>Trace unit</td>
<td>OPTIONAL Generic trace external input trigger events on page H5-10327</td>
</tr>
</tbody>
</table>

Note

Output triggers from the CTI are inputs to other blocks.

Table H5-2 on page H5-10325 shows the input trigger events defined by the architecture and the related trigger numbers.

Table H5-2 Allocation of CTI input trigger events

<table>
<thead>
<tr>
<th>Number</th>
<th>Source</th>
<th>Destination</th>
<th>Event description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PE</td>
<td>CTI</td>
<td>Cross-halt trigger event on page H5-10326</td>
</tr>
<tr>
<td>1</td>
<td>PE</td>
<td>CTI</td>
<td>Performance Monitors overflow trigger event on page H5-10326</td>
</tr>
<tr>
<td>2</td>
<td>PE</td>
<td>CTI</td>
<td>Statistical Profiling Extension sample trigger event on page H5-10327</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>4 - 7</td>
<td>Trace unit</td>
<td>CTI</td>
<td>OPTIONAL Generic trace external output trigger events on page H5-10327</td>
</tr>
</tbody>
</table>

Note

Input triggers to the CTI are outputs from other blocks.

Table H5-1 on page H5-10325 and Table H5-2 on page H5-10325 show the minimum set of trigger events defined by the architecture. However:

- The Generic trace external input and output trigger events are required only if the OPTIONAL trace unit is implemented. If the OPTIONAL trace unit is not implemented, these trigger events are reserved.
- Support for the generic CTI interrupt trigger event is IMPLEMENTATION DEFINED because details of interrupt handling in the system, including any interrupt controllers, are IMPLEMENTATION DEFINED. Details regarding how the CTI interrupt is connected to an interrupt controller and its allocated interrupt number lie outside the scope of the architecture. Arm strongly recommends that implementations provide a means to generate interrupts based on external debug events.
- The other trigger events are required by the architecture.

From the introduction of Armv8-A, an implementation can extend the CTI with additional triggers. These start with the number eight.
H5.4.1 Debug request trigger event

This is an output trigger event from the CTI, and an input trigger event to the PE, asserted by the CTI to force the PE into Debug state. The trigger event is asserted until acknowledged by the debugger. The debugger acknowledges the trigger event by writing 1 to CTIINTACK[0].

Note

A debugger must poll CTITRIGOUTSTATUS[0] until it reads as 0, to confirm that the output trigger has been deasserted before generating any event that must be ordered after the write to CTIINTACK, such as a write to CTIAPPPULSE to activate another trigger.

If the PE is already in Debug state, the PE ignores the trigger event, but the CTI continues to assert it until it is removed by the debugger. See also External Debug Request debug event on page H3-10291.

H5.4.2 Restart request trigger event

This is an output trigger event from the CTI, and an input trigger event to the PE, asserted by the CTI to request the PE to exit Debug state. If the PE is in Non-debug state, the request is ignored by the PE.

If a Restart request trigger event is received at or about the same time as the PE enters Debug state, it is CONstrained UNpredictable whether:

• The request is ignored by the PE. In this case the PE enters Debug state and remains in Debug state.
• The PE enters Debug state and then immediately restarts.

Debuggers must program the CTI to send Restart request trigger events only to PEs that are halted. To enable the PE to disambiguate discrete Restart request trigger events, after sending a Restart request trigger event, the debugger must confirm that the PE has restarted and halted before sending another Restart request trigger event. Debuggers can use EDPRSR.{SDR, HALTED} to determine the Execution state of the PE.

Note

Before generating a Restart request trigger event for a PE, a debugger must ensure any Debug request trigger event targeting that PE is cleared. Debug request trigger event describes how to do this.

The trigger event is self-acknowledging, meaning that the debugger requires no further action to remove the trigger event. The trigger event is acknowledged even if the request is ignored by the PE. See also Exiting Debug state on page H2-10271.

H5.4.3 Cross-halt trigger event

This is an input trigger event to the CTI, and an output trigger event from the PE, asserted by a PE when it is entering Debug state.

Note

To reduce the latency of halting, Arm recommends that an implementation issues the Cross-halt trigger event early in the committed process of entering Debug state. This means that there is no requirement to wait until all aspects of entry to Debug state have completed before issuing the trigger event. Speculative emission of Cross-halt trigger events is not allowed. The Cross-halt trigger event must not be issued early enough for a subsequent Debug request trigger event, which might be derived from the Cross-halt trigger event, to be recorded in the EDSR.STATUS field. This applies to Debug request trigger events that are acting as inputs to the PE.

H5.4.4 Performance Monitors overflow trigger event

This is an input trigger event to the CTI, and an output trigger event from the PE, asserted each time the PE asserts a new Performance Monitors counter overflow interrupt request. See Chapter D11 The Performance Monitors Extension.
If the CTI supports multicycle trigger events, then the trigger event remains asserted until the overflow is cleared by a write to PMOVSCLR_EL0. Otherwise, the trigger event is asserted when the value of PMOVSCLR_EL0 changes from zero to a nonzero value.

**Note**
- This does not replace the recommended connection of Performance Monitors overflow trigger event to an interrupt controller. Software must be able to program an interrupt on Performance Monitors overflow without programming the CTI.
- Events can be counted when ExternalNoninvasiveDebugEnabled() == FALSE, and, in Secure state, when ExternalSecureNoninvasiveDebugEnabled() == FALSE. Secure software must be aware that overflow trigger events are nevertheless visible to the CTI.

### H5.4.5 Statistical Profiling Extension sample trigger event

If the Statistical Profiling Extension is implemented, and a sample record is written to memory, CTI input trigger 2 is asserted. This trigger might also be directly connected to other IMPLEMENTATION DEFINED debug features.

For more information, see Chapter D13 *The Statistical Profiling Extension*.

### H5.4.6 Generic trace external input trigger events

These are output trigger events from the CTI, and input trigger events to the OPTIONAL trace unit, that are used in conjunction with the Generic trace external output trigger events to pass trigger events between:
- The PE and the OPTIONAL trace unit.
- The OPTIONAL trace unit and any other component attached to the CTM, including other Trace Units.

There are four Generic trace external input trigger events.

The trigger events are self-acknowledging. This means that the debugger does not have to take any further action to remove the events.

### H5.4.7 Generic trace external output trigger events

These are input trigger events to the CTI, and output trigger events from the OPTIONAL trace unit, used in conjunction with the Generic trace external input trigger events to pass trigger events between:
- The PE and the OPTIONAL trace unit.
- The OPTIONAL trace unit and any other component attached to the CTM, including other Trace Units.

There are four Generic trace external output trigger events.

### H5.4.8 Generic CTI interrupt trigger event

This is an output trigger event from the CTI, and an input to an IMPLEMENTATION DEFINED interrupt controller, and can transfer trigger events from the PE, trace units, or any other component attached to the CTI and CTM to software as an interrupt. The Generic CTI interrupt trigger event must be connected to the interrupt controller as an interrupt that can target the originating PE.

**Note**
- Arm recommends that the Generic CTI interrupt trigger event is a private peripheral interrupt, but implementations might instead make this trigger event available as a shared peripheral interrupt or a local peripheral interrupt.
- GICv3 reserves a private peripheral interrupt number for this interrupt.

It is IMPLEMENTATION DEFINED whether this trigger event is:
- Self-acknowledging. This means that the debugger is not required to take any further action, and that the interrupt controller must treat the trigger event as a pulse or edge-sensitive interrupt.
• Acknowledged by the debugger. The debugger acknowledges the trigger event by writing 1 to CTIINTACK[2]. This means that the interrupt controller must treat the trigger event as a level-sensitive interrupt.

Arm recommends that the Generic CTI interrupt trigger event is a self-acknowledging trigger event.
H5.5 CTI registers programmers’ model

The CTI registers programmers’ model is described in Chapter H8 About the External Debug Registers. The following sections contain information specific to the CTI:

- External debug register resets on page H8-10378.
- External debug interface register access permissions on page H8-10364.
- Cross-trigger interface registers on page H8-10375.
- The individual register descriptions in Cross-Trigger Interface registers on page H9-10660.

See also Memory-mapped accesses to the external debug interface on page H8-10362.

H5.5.1 CTI reset

An External Debug reset resets the CTI. See External debug register resets on page H8-10378 for details of CTI register resets. All CTI output triggers and output channels are deasserted on an External Debug reset.

--- Note ---

An indirect read of an output trigger might not observe the deasserted state until the processor is Cold reset. For more information, see Synchronization of changes to the external debug registers on page H8-10358.

H5.5.2 CTI authentication

The CTI ignores the state of the IMPLEMENTATION DEFINED authentication interface. This means that:

- CTRIGINSTATUS shows the status of the input triggers and CTICHINSTATUS shows the status of the input channels, regardless of the value of ExternalNoninvasiveDebugEnabled().

--- Note ---

The PE does not generate the Cross-halt trigger event and the trace unit does not generate Generic trace external output trigger events when ExternalNoninvasiveDebugEnabled() == FALSE. However, the PE can generate Performance Monitors overflow trigger events.

--- Note ---

The CTI can generate external triggers regardless of the value of ExternalInvasiveDebugEnabled().

--- Note ---

The PE ignores Debug request and Restart request trigger events when ExternalInvasiveDebugEnabled() == FALSE. The trace unit ignores Generic trace external input trigger events when ExternalNoninvasiveDebugEnabled() == FALSE. The behavior of Generic CTI interrupt requests is part of the IMPLEMENTATION DEFINED handling of these interrupts, but it is permissible for an interrupt controller to receive these requests even when ExternalInvasiveDebugEnabled() == FALSE.
The Embedded Cross-Trigger Interface

H5.6 Examples

The Cross-trigger Interface (CTI) is fully programmable and allows for flexible cross-triggering of events within a Processing Element (PE) and between PEs in a multiprocessor system. For example:

- The Cross-halt trigger event and the Debug request trigger event can be used for cross-triggering in a multiprocessor system.
- The Cross-halt trigger event and the Generic interrupt trigger event can be used for event-driven debugging in a multiprocessor system.
- The Performance Monitors overflow trigger event and the Debug request trigger event can force entry to Debug state on overflow of a Performance Monitors event counter, for event-driven profiling.

**Note**

This does not replace the recommended connection of Performance Monitors overflow trigger events to an interrupt controller. Software must be able to program an interrupt on Performance Monitors overflow without programming the CTI. Arm recommends that the Performance Monitors overflow signal is directly available as a local interrupt source.

- The Generic trace external input and Generic trace external output trigger events can pass trace events into and out of the event logic of the trace unit. They can do this:
  - To pass trace events between Trace Units.
  - In conjunction with the Performance Monitors overflow trigger event, to couple the Performance Monitors to the PE trace unit.
  - In conjunction with the Debug request trigger event, to trigger entry to Debug state on a trace event.
  - In conjunction with other CTIs, to signal a trace trigger event onto a CoreSight trace interconnect.

The following sections describe some examples in more detail:

- **Halting a single PE.**
- **Halting all PEs in a group when any one PE halts.**
- **Synchronously restarting a group of PEs on page H5-10331.**
- **Halting a single PE on Performance Monitors overflow on page H5-10331.**

Example H5-1 Halting a single PE

To halt a single PE, set:

1. CTI$\text{GATE}[0]$ to 0, so that the CTI does not pass channel events on internal channel 0 to the CTM.
2. CTI$\text{OUTEN}[0]$ to 1, so that the CTI generates a Debug request trigger event in response to a channel event on channel 0.

**Note**

The Cross-halt trigger event is input trigger 0, meaning it is controlled by the instance of CTI$\text{OUTEN}<n>$ for which $<n>$ is 0.

3. CTI$\text{APPULSE}[0]$ to 1, to generate a channel event on channel 0.

When the PE has entered Debug state, clear the Debug request trigger event by writing 1 to CTI$\text{INTACK}[0]$, before restarting the PE.

Example H5-2 Halting all PEs in a group when any one PE halts

To program a group of PEs so that when one PE in the group halts, all of the PEs in that group halt, set the following registers for each PE in the group:

1. CTI$\text{GATE}[2]$ to 1, so that each CTI passes channel events on internal channel 2 to the CTM.
2. CTI$\text{INEN}[2]$ to 1, so that each CTI generates a channel event on channel 2 in response to a Cross-halt trigger event.
3. **CTIOUTEN0[2]** to 1, so that each CTI generates a Debug request trigger event in response to a channel event on channel 2.

   __Note__
   The Cross-halt trigger event is input trigger 0, meaning it is controlled by the instances of **CTIINEN<n>** and **CTIOUTEN<n>** for which <n> is 0.

When a PE has halted, clear the Debug request trigger event by writing a value of 1 to **CTIINTACK[0]**, before restarting the PE.

---

**Example H5-3 Synchronously restarting a group of PEs**

To restart a group of PEs, for each PE in the group:

1. If the PE was halted because of a Debug request trigger event, the debugger must ensure the trigger event is deasserted. It can do this by:
   a. Writing 1 to **CTIINTACK[0]** to clear the Debug request trigger event.
   b. Polling **CTITRIGOUTSTATUS[0]**, until it reads as 0, to confirm that the trigger event has been deasserted.

2. Set **CTIGATE[1]** to 1, so that each CTI passes channel events on internal channel 1 to the CTM.

3. Set **CTIOUTEN1[1]** to 1, so that each CTI generates a Restart request trigger event in response to a channel event on channel 1.

   __Note__
   This example must use the instance of **CTIOUTEN<n>** for which <n> is 1.

4. Set **CTIAPPPULSE[1]** to 1 on any one PE in the group, to generate a channel event on channel 1.

---

**Example H5-4 Halting a single PE on Performance Monitors overflow**

To halt a single PE on a Performance Monitors overflow set:

1. **CTIGATE[3]** to 0, so that the CTI does not pass channel events on internal channel 3 to the CTM.

2. **CTIINEN1[3]** to 1, so that the CTI generates a channel event on channel 3 in response to a Performance Monitors overflow trigger event.

   __Note__
   This step of this example must use the instance of **CTIINEN<n>** for which <n> is 1.

3. **CTIOUTEN0[3]** to 1, so that the CTI generates a Debug request trigger event in response to a channel event on channel 3.

   __Note__
   This step of this example must use the instance of **CTIOUTEN<n>** for which <n> is 0.

When the PE has entered Debug state, clear the Debug request trigger event by writing 1 to **CTIINTACK[0]**, before restarting the PE. Clear the overflow status by writing to **PMOVSCLR_EL0**.
The Embedded Cross-Trigger Interface
H5.6 Examples
Chapter H6
Debug Reset and Powerdown Support

This chapter describes the reset and powerdown support in the Debug architecture. It contains the following sections:

- About Debug over powerdown on page H6-10334.
- Power domains and debug on page H6-10335.
- Core power domain power states on page H6-10336.
- Emulating low-power states on page H6-10340.
- Powerup request mechanism on page H6-10338.
- Debug OS Save and Restore sequences on page H6-10342.
- Reset and debug on page H6-10348.

Note
Where necessary, Table K17-1 on page K17-11836 disambiguates the general register references used in this chapter.
H6.1 About Debug over powerdown

Arm A-profile external debug defines a logical model for the hardware on which a PE executes. This hardware is logically split into the **Core power domain** and the **Debug power domain**, and the model contains descriptions of the states of those domains. See:

- *Power domains and debug* on page H6-10335.
- *Core power domain power states* on page H6-10336.

An implementation may allow power domains to be powered up and down independently. Debug over powerdown provides:

- A facility for software executing on the PE to save and restore the PE state on behalf of a self-hosted or external debugger or both. See *Debug OS Save and Restore sequences* on page H6-10342.
- A facility for an external debugger to request power up of the Core power domain. See *Powerup request mechanism* on page H6-10338.
- A facility for an external debugger, or software executing on the PE, to request emulation of powerdown of the Core power domain. See *Emulating low-power states* on page H6-10340.
H6.2 Power domains and debug

Arm A-profile external debug has two logical power domains, each with its own reset:

- The Debug power domain contains the interface between the PE and the external debugger, and is powered up whenever an external debugger is connected to the SoC. It remains powered up while the external debugger is connected. When the Core power domain is completely off or in a low-power state, a debugger is permitted to access a register that is implemented in the Debug power domain. Registers in this domain are reset by an External Debug reset.
- The Core power domain contains the rest of the PE, and might be allowed to power up and power down independently of the Debug power domain.

--- Note ---
- The model of two logical power domains has an impact on the reset and access permission requirements of the debug programmers’ model.
- The power domains are described as logical because the architecture defines the requirements but does not require two physical power domains. Any power domain split that meets the requirements of the programmers’ model is a valid implementation.

The Core power domain contains several types of registers:

- Non-debug logic refers to all registers and logic that are not associated with debug.
- Self-hosted debug logic refers to registers and logic associated solely with the self-hosted debug aspects of the architecture.
- Shared debug logic refers to registers and logic associated with both the self-hosted and external debug aspects of the architecture.
- External debug logic refers to registers and logic associated solely with the external debug aspects of the architecture.

For information about which groups of registers and components are in each power domain, and which registers change power domain if FEAT_DoPD is implemented, see:

- Access permissions for the External debug interface registers on page H8-10370.
- Cross-trigger interface registers on page H8-10375.
- Management register access permissions on page K6-11689.
- Access permissions for external views of the Performance Monitors on page I3-10736.
H6.3 Core power domain power states

The Arm architecture does not define the power states of the PE as these are not normally visible to software. However, they are visible to the external debugger. Arm A-profile external debug uses a four logical power states model for the Core power domain. *Mechanisms for entering a low-power state on page D1-4676* describes the architectural mechanisms for entering low-power states.

When the PE enters a low-power state other than Powerdown by executing a WFI, WFET, WFE, or WFET instruction, it remains in that low-power state until it receives a wake-up event. See *Mechanisms for entering a low-power state on page D1-4676* for the definition of wake-up events. When halting is allowed, an External Debug Request is a wake-up event. In addition, if the PE enters the Standby state for any reason, it will leave that state to service an External Debug Request debug event.

The four logical power states are as follows:

**Normal**

The Core power domain is fully powered up and the debug registers are accessible.

**Standby**

The Core power domain is on, but there are measures to reduce energy consumption. There can be other IMPLEMENTATION DEFINED measures the OS can take to enter standby.

The PE preserves the PE state, including the debug logic state. Changing from standby to normal operation does not involve a reset of the PE.

Standby is the least invasive OS energy saving state. Standby implies only that the PE is unavailable and does not clear any debug settings. For standby, the Debug architecture requires only the following:

- If the external debug interface is accessed, the PE must respond to that access. Arm recommends that, if the PE executed a WFI or WFE instruction to enter standby, then it does not retire that instruction.

--- Note ---

When FEAT_WFxT is implemented, this also applies to the WFET and WFIT instructions.

---

Standby is transparent, meaning that to software and to an external debugger it is indistinguishable from normal operation.

**Retention**

The PE state, including debug settings, is preserved in low-power structures, allowing the Core power domain to be at least partially turned off.

Changing from low-power retention to normal operation does not involve a reset of the PE. The saved PE state is restored on changing from low-power retention state to normal operation. If software has to use an IMPLEMENTATION DEFINED code sequence before entering, or after leaving, a retention state, this is referred to as a software-visible retention state. It is IMPLEMENTATION DEFINED whether the value of DBGPRCR.CORENPDRQ is set to its Cold reset value on leaving the software-visible retention state. See the description of DBGPRCR.CORENPDRQ for more information.

--- Note ---

- This model of retention does not include implementations where the PE exits the state in response to a debug register access. From the Debug architecture perspective, implementations like this are forms of standby.

---

**Powerdown**

These measures must include the OS saving any PE state, including the debug settings, that must be preserved over powerdown.

If FEAT_DoubleLock is implemented, it is used during powerdown.

Changing from powerdown to normal operation must include:

- A Cold reset of the PE after the power level has been restored.
- The OS restoring the saved PE state.

External Debug Request debug events stay pending and debug registers in the Core power domain cannot be accessed.
An implementation might support enabling and disabling threads, either dynamically or once at reset time. Threads that are disabled in this way must appear to the external debugger as either:

- Powered off, meaning they are either:
  - In a powerdown state.
  - In a retention state.
- Held in reset state.

Arm A-profile external debug uses a simpler two states model for the Debug power domain. The two states are:

**Off**
- The Debug power domain is turned off.

**On**
- The Debug power domain is turned on.

The available power states, including the cross-product of Core power domain and Debug power domain power states is IMPLEMENTATION DEFINED. Implementations are not required to implement all of these states and might include additional states. These additional states must appear to the debugger as one of the logical power states defined by this model. The control of power states is IMPLEMENTATION DEFINED.

--- Note

As a result, it is IMPLEMENTATION DEFINED whether it is possible for the Debug power domain to be on when the Core power domain is off.

---

If the Debug power domain is implemented but is not powered up when the Core power domain is powered up, the Reset Catch debug event and the OS Unlock Catch debug event are disabled.


**H6.4 Powerup request mechanism**

If a powerup request mechanism is implemented, asserting the powerup request requests the power controller to power up the Core power domain, and to emulate any subsequent powerdown requests, until the powerup request mechanism is deasserted.

**H6.4.1 Powerup request mechanism if FEAT_DoPD is implemented**

If FEAT_DoPD is implemented, the external debug component implements an OPTIONAL powerup request mechanism.

If the powerup request mechanism is implemented, Arm strongly recommends that the powerup request is a CoreSight Class 0x9 ROM table block that contains both:

- A parent entry for the debug registers of the PE.
- A parent entry for the PMU registers of the PE, if the OPTIONAL PMU with an external debug interface is implemented.

A parent entry of a component is an entry in a ROM table that either locates the component, or locates another ROM table that contains the parent entry for the component.

--- **Note** ---

The ROM table and any descendants might describe other debug components, including debug components for other PEs.

The ROM table might have a parent entry in a second ROM table and that parent entry might also have a powerup request mechanism in the second ROM table. This applies recursively.

---

The parent entries for the debug components have the following properties:

**For the debug registers and Performance Monitors registers:**

These components are in the Core power domain.

The POWERIDVALID bit is 1.

All parent entries must have the same IMPLEMENTATION DEFINED POWERID value.

--- **Note** ---

The IMPLEMENTATION DEFINED POWERID value does not need to be unique for each PE.

---

**For the CTI registers:**

This component is in the Debug power domain.

The POWERIDVALID bit is IMPLEMENTATION DEFINED.

If the POWERIDVALID bit is 1, the entries must have a valid POWERID value.

--- **Note** ---

If the Core power domain can be powered down independently of the Debug power domain, Arm recommends the system implements an external debug component with a powerup request mechanism which can request the Core power domain to be powered up.

---

For more information about CoreSight Class 0x9 ROM Tables, see *ARM® CoreSight™ Architecture Specification*.

On a Cold reset, if FEAT_DoPD is implemented, DBGPRCR.CORENPDRQ is set to an IMPLEMENTATION DEFINED choice between 0 and 1 if the powerup request is implemented and asserted, and 0 otherwise.

**H6.4.2 Powerup request mechanism if FEAT_DoPD is not implemented**

If FEAT_DoPD is not implemented, the bit EDPRCR.COREPURQ is the powerup request mechanism.
The control registers DBGPRCR.CORENPDRQ and EDPRCR.COREPURQ provide an interface between the power controller and the PE. They typically map directly to signals in the recommended external debug interface.

On Cold reset, if FEAT_DoPD is not implemented, DBGPRCR.CORENPDRQ is set to the value of EDPRCR.COREPURQ.
H6.5 Emulating low-power states

DBGPRCR.CORENPDRQ and the powerup request mechanism can request the power controller to emulate states where the Core power domain is completely off or in a low-power state where the Core power domain registers cannot be accessed. This simplifies the requirements on software by sacrificing entirely realistic behavior.

If FEAT_DoPD is not implemented, EDPRSR.{SPD, PU} indicates the Core power domain power state. For more information, see:

• The DBGPRCR_EL1 and DBGPRCR System register descriptions.
• The EDPRCR and EDPRSR external debug register descriptions.
• Appendix K6 Recommended External Debug Interface.

The measures to emulate powerdown are IMPLEMENTATION DEFINED. The ability of the debugger to access the state of the PE and the system might be limited as a result of the measures adopted.

In an emulated powerdown state, the debugger must be able to access all debug, PMU, CTI, and trace unit registers that are accessible on the external debug interface and are in one of:

• The Debug power domain.
• The Core power domain.
• When a trace unit with a separate trace unit Core power domain is implemented, and the trace unit Core power domain is powered on, the trace unit Core power domain.

That is, the debugger must be able to read and write to such registers without receiving errors. This allows an external debugger to debug the powerup sequence.

Arm recommends that any IMPLEMENTATION DEFINED registers that are on the external debug interface and in either the Core power domain or the Debug power domain are also accessible in an emulated powerdown state.

If FEAT_DoubleLock is implemented, DoubleLockStatus() == FALSE when DBGPRCR.CORENPDRQ == 1.

Otherwise, the behavior of the PE in emulated powerdown must be similar to that in a real powerdown state. In particular, the PE must not respond to other system stimuli, such as interrupts.

Example H6-1 and Example H6-2 are examples of two approaches to emulating powerdown.

Example H6-1: An example of emulating powerdown

The PE is held in Standby state, isolated from any system stimuli. It is IMPLEMENTATION DEFINED whether the PE can respond to debug stimuli such as an External Debug Request debug event.

If the PE can enter Debug state, then the external debugger is able to use the ITR to execute instructions, such as loads and stores. This causes the external debugger to interact with the system. If the external debugger restarts the PE, the PE leaves Standby state and restarts fetching instructions from memory.

Example H6-2: Another example of emulating powerdown

The PE is held in Warm reset. This limits the ability of an external debugger to access the resources of the PE. For example, the PE cannot be put into Debug state.

On exit from emulated powerdown the PE is reset. However, the debug registers that are only reset by a Cold reset must not be reset. Typically this means that a Warm reset is substituted for the Cold reset. As such, the effect of accessing any register that is reset by a Warm reset while the PE is in the emulated powerdown state will have an IMPLEMENTATION DEFINED effect on that register.
Note

- Warm reset and Cold reset have different effects apart from resetting the debug registers. In particular, RMR_ELx is reset by a Cold reset and controls the reset state on a Warm reset. This means that if a Cold reset is substituted by a Warm reset, the behavior of the reset code might be different.

- The timing effects of powering down are typically not factored in the powerdown emulation. Examples of these timing effects are clock and voltage stabilization.

- Emulation does not model the state lost during powerdown, meaning that it might mask errors in the state storage and recovery routines.
H6.6 Debug OS Save and Restore sequences

From the introduction of Armv8-A, the following registers provide the OS Save and Restore mechanism:

- The **OS Lock Access Register**, OSLAR, locks the OS Lock to restrict access to debug registers before starting an OS Save sequence, and unlocks the OS Lock after an OS Restore sequence.
- The **OS Lock Status Register**, OSLSR, shows the status of the OS Lock.
- The PE can be configured to generate an OS Unlock Catch debug event on page H3-10292 when the OS Lock is unlocked.
- If **FEAT_DoubleLock** is implemented, the OS Double Lock locks out an external debug interface entirely. This is only used immediately before a powerdown sequence.

See also:
- **FEAT_DoubleLock** on page A2-78
- **Reset and debug** on page H6-10348
- **Appendix K10 Example OS Save and Restore Sequences**

H6.6.1 EDPRSR.{DLK, SPD, PU} and the Core power domain

If **FEAT_DoPD** is not implemented, a debugger uses EDPRSR.{DLK, SPD, PU} to determine whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read.

If **FEAT_DoPD** is implemented, accesses to EDPRSR return an error when the Core power domain is off or in a retention state, meaning successful reads of EDPRSR always return 1 for EDPRSR.PU.

When **FEAT_Debugv8p4** is implemented, and whenever **FEAT_DoubleLock** is not implemented, EDPRSR.DLK is always 0.

If **FEAT_DoubleLock** is not implemented, **DoubleLockStatus()** always returns FALSE.

If the Core power domain is powered up and **DoubleLockStatus()** == TRUE, then:

- When **FEAT_Debugv8p2** is not implemented, EDPRSR.{DLK, SPD, PU} can read either {1, UNKNOWN, 1} or {UNKNOWN, 0, 0}.
- When **FEAT_Debugv8p2** is implemented, and **FEAT_Debugv8p4** is not implemented, EDPRSR.{DLK, SPD, PU} can only read {UNKNOWN, 0, 0}.

<table>
<thead>
<tr>
<th>EDPRSR</th>
<th>Core power domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLK</td>
<td>SPD</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table H6-1 Interpretation of the EDPRSR.{DLK, SPD, PU} bits**
H6.6.2 EDPRSR.SPD when the Core domain is in either retention or powerdown state

If FEAT_DoPD is not implemented, when the Core power domain is in either the retention or powerdown state, EDPRSR.SPD is not cleared following a read of EDPRSR and it is IMPLEMENTATION DEFINED whether:

- EDPRSR.SPD shows whether the state of the debug registers in the Core power domain has been lost since the last time that EDPRSR was read. This means that:
  - When the Core power domain is in the powerdown state, EDPRSR.SPD is RAO, this indicates that the state of the debug registers has been lost.
  - When the Core power domain is in the retention state, EDPRSR.SPD indicates whether the state of the debug registers was lost before the Core power domain entered retention state.

- EDPRSR.SPD is RAZ, and:
  - On leaving the powerdown state, EDPRSR.SPD is set to 1 which indicates that the state of the debug registers has been lost.
  - On leaving the retention state, EDPRSR.SPD reverts the value it had on entering the retention state.

Note

If FEAT_DoPD is implemented, accesses to EDPRSR return an error when the Core power domain is off or in a retention state.

H6.6.3 EDPRSR.{DLK, R} and reset state

If FEAT_DoPD is implemented, accesses to EDPRSR return an error when the Core power domain is off or in a retention state, meaning successful reads of EDPRSR always return 1 for EDPRSR.PU.

When FEAT_Debugv8p4 is implemented, and whenever FEAT_DoubleLock is not implemented, EDPRSR.DLK is always 0.

If FEAT_DoubleLock is not implemented, DoubleLockStatus() always returns FALSE.

If FEAT_DoubleLock is implemented and enabled, the behavior of all registers and fields except EDPRSR.DLK is the same as their behavior if FEAT_Debugv8p4 is not implemented.

If FEAT_Debugv8p4 is implemented EDPRSR.DLK is always 0 and does not give any information about the OS Double Lock.

EDPRSR.R is UNKNOWN when DoubleLockStatus() == TRUE. OSDLR_EL1.DLK is cleared to 0 by a reset. If the Core power domain is powered up and entered reset state with the OS Double Lock locked, it is CONSTRAINED UNPREDICTABLE whether a read of EDPRSR while the PE is in reset state returns:

- EDPRSR.{DLK, R, PU} == {1, UNKNOWN, 1} indicating that the OS Double Lock is locked. This is not permitted from Armv8.2.
- EDPRSR.{DLK, R, PU} == {0, 1, 1} indicating that the PE is in reset state.
- EDPRSR.{DLK, R, PU} == {UNKNOWN, UNKNOWN, 0} indicating that the registers in the Core power domain cannot be accessed because the OS Double Lock is locked.

If the PE was powered up and the OS Double Lock was unlocked when the PE was reset, then EDPRSR.{DLK, R, PU} reads as {0, 1, 1} while the PE is in reset state.

On leaving reset state, EDPRSR.{DLK, R} reads as {0, 0}.

H6.6.4 Debug registers to save over powerdown

Table H6-2 on page H6-10344 shows the different requirements for self-hosted debug over powerdown and external debug over powerdown:

- The column labeled Self-hosted lists registers that software must preserve over powerdown so that it can support self-hosted debug over powerdown. This does not require use of the OS Save and Restore mechanism.
The column labeled External lists registers that software must preserve over powerdown so that it can support external debug over powerdown. This requires use of the OS Save and Restore mechanism:

- Some external debug registers are not normally accessible to software executing on the PE. Additional debug registers are provided that give software the required access to save and restore these external debug registers when OSLSR.OSLK is locked. These registers include OSECCR, OSDTRRX, and OSDTRTX.

- Some registers might only present in some implementations, or might not be accessible at all Exception levels or in Non-secure state. DBGVCR32_EL2 and SDER32_EL3 are only required to support AArch32.

Table H6-2 does not include registers for the optional Trace and Performance Monitor extensions.

<table>
<thead>
<tr>
<th>Register in AArch64 state</th>
<th>Register in AArch32 state</th>
<th>Self-hosted</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDSCR_EL1</td>
<td>DBGDSCRx</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DBGBVR&lt;n&gt;_EL1</td>
<td>DBGBVR&lt;n&gt;</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DBGBCR&lt;n&gt;_EL1</td>
<td>DBGBCR&lt;n&gt;</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DBGWVR&lt;n&gt;_EL1</td>
<td>DBGWVR&lt;n&gt;</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DBGWCR&lt;n&gt;_EL1</td>
<td>DBGWCR&lt;n&gt;</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DBGVCR32_EL2</td>
<td>DBGVCR</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>MDCR_EL2</td>
<td>HDCR</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>SDER32_EL3</td>
<td>SDER</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>MDCR_EL3</td>
<td>SDCR</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>MDCCINT_EL1</td>
<td>DBGDCCINT</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>DBGCLAIMSET_EL1</td>
<td>DBGCLAIMSET,</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>DBGCLAIMCLR_EL1</td>
<td>DBGCLAIMCLR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OSECCR_EL1</td>
<td>DBGOSECCR</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>OSDTRRX_EL1</td>
<td>DBGDTRRX</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>OSDTRTX_EL1</td>
<td>DBGDTRTX</td>
<td>-</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**H6.6.5 OS Save sequence**

To preserve the debug logic state over a powerdown, the state must be saved to nonvolatile storage. This means the OS Save sequence must:

1. Lock the OS Lock by:
   - Writing the key value 0xC5ACCE55 to the DBGOSLAR in AArch32 state.
   - Writing 1 to OSLAR_EL1.OSLK in AArch64 state.

2. Execute an ISB instruction.

3. Walk through the debug registers listed in *Debug registers to save over powerdown* on page H6-10343 and save the values to the nonvolatile storage.

If the FEAT_DoubleLock is implemented, before removing power from the Core power domain, software must:

1. Lock the OS Double Lock by:
   - Writing 1 to DBGOSDLR.DLK in AArch32 state.
   - Writing 1 to OSDLR_EL1.DLK in AArch64 state.

If FEAT_DoubleLock is not implemented, OSDLR_EL1 and DBGOSDLR ignore writes.
2. Execute a Context synchronization event.

H6.6.6 OS Restore sequence

After a powerdown, the OS Restore sequence must perform the following steps to restore the debug logic state from the non-volatile storage:

1. Lock the OS Lock, as described in OS Save sequence on page H6-10344. The OS Lock is generally locked by the Cold reset, but this step ensures that it is locked.

2. Execute an ISB instruction.

3. To ensure that, if an external debugger clears the OS Lock before the end of this sequence, no debug exceptions are generated:
   - Write 0 to MDSCR_EL1 if executing in AArch64 state.
   - Write 0 to DBGDSCRExt if executing in AArch32 state.

4. Walk through the debug registers listed in Debug registers to save over powerdown on page H6-10343, and restore the values from the nonvolatile storage. The last register to be restored must be:
   - MDSCR_EL1 if executing in AArch64 state.
   - DBGDSCRExt if executing in AArch32 state.

5. Execute an ISB instruction.

6. Unlock the OS Lock by:
   - Writing any non-key value to DBGOSLAR if executing in AArch32 state.
   - Writing 0 to OSLAR_EL1.OSLK if executing in AArch64 state.

7. Execute a Context synchronization event.

--- Note

The OS Restore sequence overwrites the debug registers with the values that were saved. If there are valid values in these registers immediately before the restore sequence, then those values are lost.

H6.6.7 Debug behavior when the OS Lock is locked

The main purpose of the OS Lock is to prevent updates to debug registers during an OS Save or OS Restore operation. The OS Lock is locked on a Cold reset.

When the OS Lock is locked:

- Access to debug registers through the System register interface is mainly unchanged except that:
  - Certain registers are read and written without side-effects.
  - Fields in DSCR and OSECCR that are normally read-only become read/write.

This allows the state to be saved or restored. For more information, see the relevant register description in Chapter H9 External Debug Register Descriptions.

- Access to debug registers by the external debug interface is restricted to prevent an external debugger modifying the registers that are being saved or restored. For more information, see External debug interface register access permissions summary on page H8-10365.

- Debug exceptions, other than Breakpoint Instruction exceptions are not generated.

- Breakpoint and Watchpoint debug events are not generated. The OS Lock has no effect on Breakpoint Instruction exceptions and other debug events.

H6.6.8 Debug behavior when the OS Lock is unlocked

When the OS Lock is unlocked, the PE sets EDESR.OSUC to 1 if the OS Unlock Catch debug event is enabled and the PE is in Non-debug state, meaning the OS Unlock Catch debug event becomes pending. See OS Unlock Catch debug event on page H3-10292.
H6.6.9 Debug behavior when the OS Double Lock is locked

If the FEAT_DoubleLock is implemented, software locks the OS Double Lock immediately before a powerdown sequence.

The OS Double Lock ensures that it is safe to remove core power by forcing the debug interfaces to be quiescent.

When DoubleLockStatus() == TRUE:
- The external debug interface has only restricted access to the debug registers, so that it is quiescent before removing power. See External debug interface register access permissions summary on page H8-10365.
- Debug exceptions, other than Breakpoint Instruction exceptions, are not generated.
- Halting is prohibited. See Halting allowed and halting prohibited on page H2-10233.

Note
Pending Halting debug events might be lost when core power is removed.

- No asynchronous debug events are WFI wake-up events.

If the FEAT_DoubleLock is not implemented, the PE ensures these conditions are met before allowing power to be removed.

Software must synchronize the update to OSDLR before it indicates to the system that core power can be removed. The interface between the PE and its power controller is IMPLEMENTATION DEFINED.

Typically software indicates that core power can be removed by entering the Wait For Interrupt state. This means that software must explicitly synchronize the OSDLR update before issuing the WFI instruction.

OSDLR.DLK is ignored and DoubleLockStatus() == FALSE if either:
- The PE is in Debug state.
- DBGPRCR.CORENPDRQ is set to 1.

Note
It is possible to enter Debug state with OSDLR.DLK set to 1. This is because a Context synchronization event is required to ensure the OS Double Lock is locked, meaning that Debug state might be entered before the OSDLR update is synchronized.

Because OSDLR.DLK is ignored when DBGPRCR.CORENPDRQ is set to 1, an external debugger can write to DBGPRCR.CORENPDRQ, and the FEAT_DoubleLock is not always implemented, software must not rely on using the OS Double Lock to disable debug exceptions or to prohibit halting, or both. Arm deprecates use of the OS Double Lock for these purposes, and instead recommends that software:
- Uses the OS Lock to disable debug exceptions during save or restore sequences.
- Uses the debug authentication interface to prohibit halting and external debug access to debug registers at times other than immediately prior to removing power.

As the purpose of the OS Double Lock is to ensure that it is safe to remove core power, if the FEAT_DoubleLock is implemented, it is important to avoid race conditions that defeat this purpose. Arm recommends that:
- Once the write to OSDLR.DLK has been synchronized by a Context synchronization event and DoubleLockStatus() == TRUE, a PE must:
  - Not allow a debug event generated before the Context synchronization event to cause an entry to Debug state or act as a wake-up event for a WFI or WFE instruction after the Context synchronization event has completed.

Note
When FEAT_WFxT is implemented, this also applies to the WFET and WFIT instructions.

- Complete any external debug access started before the Context synchronization event by the time the Context synchronization event completes.
--- Note ---

A debug register access might be in progress when software sets OSDLR.DLK to 1. An implementation must not permit the synchronization of locking the OS Double Lock to stall indefinitely while waiting for that access to complete. This means that any debug register access that is in progress when software sets OSDLR.DLK to 1 must complete or return an error in finite time.

- If a write to DBGPRCR or EDPRCR made when OSDLR.DLK == 1 changes DBGPRCR.CORENPDRQ or EDPRCR.CORENPDRQ from 1 to 0, meaning DoubleLockStatus() changes from FALSE to TRUE, then before signaling to the system that the CORENPDRQ field has been cleared and emulation of powerdown is no longer requested, meaning the system can remove core power, the PE must ensure that all the requirements for DoubleLockStatus() == TRUE listed in this section are met.

In a standard OS Save sequence, the OS Lock is locked before the OS Double Lock is locked. This means that writes to CORENPDRQ are ignored by the time the OS Double Lock is locked. However, if DoubleLockStatus() == FALSE, an external debugger can clear the OS Lock at any time, and then write to EDPRCR.
H6.7 Reset and debug

All registers in the Core power domain are either:
• Reset by both a Cold and a Warm reset.
• Reset only by a Cold reset and are not changed by a Warm reset.

For more information, see Resets and power domains on page D1-4673.

All registers in the Debug power domain are reset by an External Debug reset.

Figure H6-1 shows this reset scheme. The following three reset signals are an example implementation of the reset scheme:
• CORERESET, which must be asserted for a Warm reset.
• CPUPORESET, which must be asserted for a Cold reset.
• PRESETDBG, which must be asserted for an External Debug reset.

As shown in the figure, the external debug logic is split between the Debug power domain and the Core power domain.

![Figure H6-1 Power and reset domains](image)

For more information about power domains and power states, see Power domains and debug on page H6-10335.

When power is first applied to the Debug power domain, PRESETDBG must be asserted.

When power is first applied to the Core power domain, CPUPORESET must be asserted.

Note

In this scheme, logic in the Warm reset domain is reset by asserting either CORERESET or CPUPORESET. This implies a particular implementation style that permits these approaches.

CPUPORESET is not normally asserted on moving from a low-power state, where power has not been removed, to a full-power state. This can occur, for example, on exiting a low-power retention state. See also Emulating low-power states on page H6-10340 and the EDPFR register description.

H6.7.1 External debug interface accesses to registers in reset

If a reset signal is asserted and the external debug interface:
• Writes a register, or indirectly writes a register or register field as a side-effect of an access:
  • Then, if the register or register field is reset by that reset signal, it is CONSTRAINED UNPREDICTABLE whether the register or register field takes the reset value or the value written. The reset value might be UNKNOWN.
— Otherwise, the register or register field takes the value that is written.

• Reads a register, or indirectly reads a register or register field, as part of an access:
  — Then, if the register or register field is reset by that reset signal, the value returned in UNKNOWN.
  — Otherwise, the value of the register or register field is returned.

It is IMPLEMENTATION DEFINED whether any register can be accessed when External Debug reset is being asserted. The result of these accesses is IMPLEMENTATION DEFINED.
Debug Reset and Powerdown Support
H6.7 Reset and debug
Chapter H7
The PC Sample-based Profiling Extension

This chapter describes the OPTIONAL PC Sample-based Profiling Extension that provides a non-invasive external debug component.

It contains the following section:
- About the PC Sample-based Profiling Extension on page H7-10352.
H7.1 About the PC Sample-based Profiling Extension

The PC Sample-based Profiling Extension is an OPTIONAL extension that provides coarse-grained, non-invasive profiling by an external debugger. See also Non-invasive behavior on page D11-5248.

PC Sample-based Profiling creates samples so that tools can populate a statistical model of the performance of software executing on the PE.

Note

Data returned by periodic sampling of PC Sample-based Profiling registers is sufficient to allow tools to estimate the distribution of time spent executing software on the PE.

The delay between an instruction being executed by the PE and its address appearing in the PC Sample Register is not defined, and the architecture does not require that the sampled instruction was recently executed. For example, if a piece of software executes a load instruction that reads the PC Sample Register of the PE it is running on, there is no guaranteed relationship between the address of the load instruction and the value read. The PC Sample Register is intended only for use by an external agent to provide statistical information for software profiling.

It must be possible to sample references to branch targets. It is IMPLEMENTATION DEFINED whether references to other instructions can be sampled. The branch target for a conditional branch instruction that fails its condition check is the instruction that follows the conditional branch instruction. The branch target for an exception is the exception vector address.

To keep the implementation and validation cost low, a reasonable degree of inaccuracy in the sampled data is acceptable. Arm does not define a reasonable degree of inaccuracy but recommends the following guidelines:

- In exceptional circumstances, such as a change in Security state or other boundary condition, it is acceptable for the sample to represent an instruction that was not committed for execution.
- Under unusual non-repeating pathological cases, the sample can represent an instruction that was not committed for execution. These cases are likely to occur as a result of asynchronous exceptions, such as interrupts, where the chance of a systematic error in sampling is very unlikely.
- Under normal operating conditions, the sample must reference an instruction that was committed for execution, including its context, and must not reference instructions that are fetched but not committed for execution.

Note

In the Armv7 PC Sample-based Profiling Extension, an offset was applied to the sampled program counter value and this offset and the instruction set state indicated in bits [1:0] of the sampled value. From the introduction of the Armv8 PC Sample-based Profiling Extension, the sampled value is the address of an instruction that has executed, with no offset and no indication of the instruction set state.

- Controlling the PC Sample-based Profiling Extension.
- Registers implemented by the PC Sample-based Profiling Extension on page H7-10353.
- Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN on page H7-10354.
- Pseudocode description of PC Sample-based Profiling on page H7-10354.

H7.1.1 Controlling the PC Sample-based Profiling Extension

PC Sample-based Profiling is controlled by the IMPLEMENTATION DEFINED authentication interface ExternalNoninvasiveDebugEnabled().

PC Sample-based Profiling is prohibited unless both:

- It is allowed by the IMPLEMENTATION DEFINED authentication interface ExternalNoninvasiveDebugEnabled().
- At least one of the following applies:
  - The PE is executing in Non-secure state.
  - The PE is executing in Secure state, and non-invasive debug is allowed by the IMPLEMENTATION DEFINED authentication interface ExternalSecureNoninvasiveDebugEnabled().
H7.1 About the PC Sample-based Profiling Extension

The state of the IMPLEMENTATION DEFINED authentication interface is visible through DBGAUTHSTATUS_EL1. See Recommended authentication interface on page K6-11687.

H7.1.2 Registers implemented by the PC Sample-based Profiling Extension

The options for implementing the PC Sample-based Profiling extension are:

- The extension is implemented in the external debug register space. EDDEVID.PCSample and identifies the implemented level of profiling, and EDDEVID1.PCSROffset also indicates that this option is implemented. From Armv8.2 this option is not permitted.
- FEAT_PCSRv8p2 is implemented, meaning the PC Sample-based Profiling extension is implemented in the Performance Monitors memory-mapped register space. PMDEVID.PCSample identifies the implemented level of profiling.

If PC Sample-based Profiling is implemented in the external debug register space:

- The following external debug registers can be implemented:
  - EDCIDSR.
  - EDPCSR.
  - EDVIDSR.
  - DBGAUTHSTATUS_EL1.
  - PMCCIDSR.
  - PMPCSR.
  - PMVIDSR.
  - PMVCIDSR.

See External debug interface registers on page H8-10368.

- If FEAT_VHE is implemented, EDSCR.SC2 controls what PC Sample-based Profiling samples.

If FEAT_PCSRv8p2 is implemented and the 64-bit external PMU programmers’ model extension is not implemented, the following registers can be implemented in the Performance Monitors memory-mapped register space:

- PMCID1SR and PMCID2SR.
- PMPCSR.
- PMVIDSR.

When the 64-bit external PMU programmers’ model extension is implemented, these registers are:

- PMCCIDSR.
- PMPCSR.
- PMVIDSR.
- PMVCIDSR.

See Performance Monitors external register views on page I5-10748.

If the PC Sample-based Profiling Extension is implemented with FEAT_PCSRv8p2 but the Performance Monitors Extension is not implemented, then the PC Sample-based Profiling Extension is implemented in its own memory-mapped register space, within the area that is reserved for the Performance Monitors, see Table H7-1 on page H7-10353. If CoreSight compliance is required:

- The management registers are defined as in Table K6-3 on page K6-11688.
- The support for PC Sample-based profiling is defined in the following registers:
  - PMDEVTYPEMAJOR has the value 0x0.
  - PMDEVACHR.ARCHID has the value 0x0A10.

Table H7-1  PC Sample-based Profiling register map without the Performance Monitors Extension

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x200</td>
<td>PMPCSR[31:0]</td>
</tr>
<tr>
<td>0x204</td>
<td>PMPCSR[63:32]</td>
</tr>
<tr>
<td>0x208</td>
<td>PMCID1SR</td>
</tr>
<tr>
<td>0x20C</td>
<td>PMVIDSR</td>
</tr>
</tbody>
</table>
H7.1 About the PC Sample-based Profiling Extension

H7.1.3 Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN

The architecture permits IMPLEMENTATION DEFINED extensions to external debug to define mechanisms that make the values of the PC Sample-based profiling registers UNKNOWN. However, it requires that any such mechanism is disabled by default. This means that powerup or a hard reset of the PE must leave the PE in a state where the PC Sample-based Profiling Extension, if implemented, exhibits its architecturally-defined behavior.

--- Note ---
A mechanism that, when enabled, makes the PC Sample-based profiling registers UNKNOWN might use other sample-based profiling events that are appropriate for a use that is independent of PC Sample-based Profiling.

If no branch instruction has been retired since the PE left Debug state, reset state, or a state where PC Sample-based profiling is prohibited, the sampled value is UNKNOWN. If a branch instruction has been retired but this is the first time the PMPCSR or EDPCSR is read since the PE left reset state, the sampled value is permitted but not required to return the value 0xFFFFFFF.

When FEAT_MOPS and FEAT_PCSRv8p2 are not implemented, if no branch instruction has been retired since the last read of PMPCSR[31:0], then the value of PMPCSR[31:0] is UNKNOWN.

H7.1.4 Pseudocode description of PC Sample-based Profiling

When PC Sample-based Profiling is implemented but not with FEAT_PCSRv8p2, the functionality is described by the pseudocode functions:

- CreatePCSample(), which populates a variable of type PCSample.
- EDPCSRlo[()], which writes a PC sample to the EDPCSR and associated registers.

When FEAT_PCSRv8p2 is implemented, the functionality is described by the pseudocode functions:

- CreatePCSample(), which populates a variable of type PCSample.
- PMPCSR[()], which writes a PC Sample to the PMPCSR and associated registers.

---

Table H7-1 PC Sample-based Profiling register map without the Performance Monitors Extension

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x220</td>
<td>PMPCSR[31:0] (alias)</td>
</tr>
<tr>
<td>0x224</td>
<td>PMPCSR[63:32] (alias)</td>
</tr>
<tr>
<td>0x228</td>
<td>PMCID1SR (alias)</td>
</tr>
<tr>
<td>0x22C</td>
<td>PMCID2SR</td>
</tr>
<tr>
<td>0x600-0x6FC</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>0xE80-0xEFC</td>
<td>IMPLEMENTATION DEFINED for CoreSight compliance</td>
</tr>
<tr>
<td>0xFF0-0xFFC</td>
<td>Management and CoreSight compliance registers</td>
</tr>
</tbody>
</table>
Chapter H8
About the External Debug Registers

This chapter provides some additional information about the external debug registers. It contains the following sections:

- Relationship between external debug and System registers on page H8-10356.
- Endianness and supported access sizes on page H8-10357.
- Synchronization of changes to the external debug registers on page H8-10358.
- Memory-mapped accesses to the external debug interface on page H8-10362.
- External debug interface register access permissions on page H8-10364.
- External debug interface registers on page H8-10368.
- Cross-trigger interface registers on page H8-10375.
- External debug register resets on page H8-10378.

Note
Where necessary, Table K17-1 on page K17-11836 disambiguates the general register references used in this chapter.
H8.1 Relationship between external debug and System registers

Table H8-1 on page H8-10356 shows the relationship between external debug registers and System registers. Where no relationship exists, the registers are not listed.

Table H8-1  Equivalence between external debug and System registers

<table>
<thead>
<tr>
<th>System register</th>
<th>AArch64</th>
<th>AArch32</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBGDTRRX_EL0</td>
<td>DBGDTRRX_EL0</td>
<td>DBGDTRRXint</td>
<td>See also Summary of System register accesses to the DCC on page H4-10309</td>
</tr>
<tr>
<td>DBGDTRTX_EL0</td>
<td>DBGDTRTX_EL0</td>
<td>DBGDTRTXint</td>
<td></td>
</tr>
<tr>
<td>OSLAR_EL1</td>
<td>OSLAR_EL1</td>
<td>DBGOSLAR</td>
<td></td>
</tr>
<tr>
<td>DBGVR&lt;n&gt;_EL1[31:0]</td>
<td>DBGVR&lt;n&gt;_EL1[31:0]</td>
<td>DBGVR&lt;n&gt;</td>
<td></td>
</tr>
<tr>
<td>DBGBCR&lt;n&gt;_EL1</td>
<td>DBGBCR&lt;n&gt;_EL1</td>
<td>DBGBCR&lt;n&gt;</td>
<td></td>
</tr>
<tr>
<td>DBGWVR&lt;n&gt;_EL1[31:0]</td>
<td>DBGWVR&lt;n&gt;_EL1[31:0]</td>
<td>DBGWVR&lt;n&gt;</td>
<td></td>
</tr>
<tr>
<td>DBGCLAIMSET_EL1</td>
<td>DBGCLAIMSET_EL1</td>
<td>DBGCLAIMSET</td>
<td></td>
</tr>
<tr>
<td>DBGCLAIMCLR_EL1</td>
<td>DBGCLAIMCLR_EL1</td>
<td>DBGCLAIMCLR</td>
<td></td>
</tr>
<tr>
<td>DBGAUTHSTATUS_EL1</td>
<td>DBGAUTHSTATUS_EL1</td>
<td>DBGAUTHSTATUS</td>
<td>Read-only</td>
</tr>
<tr>
<td>EDSCR</td>
<td>MDSCR_EL1</td>
<td>DBGDSCR_ext</td>
<td>Only some fields map</td>
</tr>
<tr>
<td>EDECCR</td>
<td>OSECCR_EL1</td>
<td>DBGOSECCR</td>
<td>Applies when the OS Lock is locked</td>
</tr>
<tr>
<td>MIDR_EL1</td>
<td>MIDR_EL1</td>
<td>MIDR</td>
<td>Read-only copies of Processor ID Registers</td>
</tr>
<tr>
<td>MPIDR_EL1[31:0]</td>
<td>MPIDR_EL1[31:0]</td>
<td>MPIDR</td>
<td>Read-only copies of system ID registers</td>
</tr>
</tbody>
</table>

a. This is a word of a 64-bit register.

In addition:
- EDSCR.TXfull, RXfull are read-only aliases for DCCSR.TXfull, RXfull.
- EDPRCR.CORENPDRQ is a read/write alias for DBGPRCR.CORENPDRQ.
- EDPRSR.OSLK is a read-only alias for OSLR.OSLK.
- If the FEAT_DoubleLock is implemented, EDPRSR.DLK is a read-only function of OSDLR.DLK.
H8.2 Endianness and supported access sizes

The debug registers, Performance Monitors registers, and CTI registers are implemented as memory-mapped peripherals. The Arm architecture requires memory-mapped peripherals to be little-endian.

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the debug registers, Performance Monitors registers, and CTI registers, implementations must:

• Comply with the requirements of Supported access sizes on page I1-10716.
• Support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register mapped to a doubleword-aligned pair of adjacent 32-bit locations, even if all components with direct memory access to the memory-mapped peripheral support making 64-bit accesses.

Note

These requirements mean that a system implementing the debug registers using a 32-bit bus, such as a AMBA APB3, with a wider system interconnect must implement a bridge between the system and the debug bus that can split 64-bit accesses.

For accesses from the external debug interface, the size of an access is determined by the interface. For an access from an ADIv5-compliant Memory Access Port, MEM-AP, this is specified by the MEM-AP CSW register.
H8.3 Synchronization of changes to the external debug registers

This section describes the synchronization requirements for the external debug interface.

For more information on how these requirements affect debug, see:

- Synchronization and debug exceptions on page D2-4755 for exceptions taken from AArch64 state.
- Synchronization and debug exceptions on page G2-9100 for exceptions taken from AArch32 state.
- Synchronization and Halting debug events on page H3-10295.
- Synchronization of DCC and ITR accesses on page H4-10309.

This section refers to accesses from the external debug interface as external reads and external writes. It refers to accesses to System registers as direct reads, direct writes, indirect reads, and indirect writes.

Note

Synchronization requirements for AArch64 System registers on page D17-5547 and Synchronization of changes to AArch32 System registers on page G8-9326 define direct read, direct write, indirect read, and indirect write, and classifies external reads as indirect reads, and external writes as indirect writes.

For general information about synchronization, access completion, ordering, and observability, see Synchronization of memory-mapped registers on page I1-10718.

Writes to the same register are serialized, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes. With the exception of DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGWCR<n>_EL1, and DBGWVR<n>_EL1, external writes to different registers are not necessarily observed in the same order by all observers as the order in which they complete.

Synchronization of DCC and ITR accesses on page H4-10309 describes the synchronization requirements for the DCC and ITR.

Changes to the IMPLEMENTATION DEFINED authentication interface are external writes to the authentication status registers by the Requester of the authentication interface. See Synchronization and the authentication interface on page H8-10359.

The external agent must be able to guarantee completion of a write. For example by:

- Marking the memory as Device-nGnRnE and executing a DSBB barrier, if the system supports this property.
- Reading back the value written.
- Some guaranteed property of the connection between the PE and the external agent.

Note

For an external Debug Access Port, access completion is an IMPLEMENTATION DEFINED property. For a CoreSight system using APB-AP to access a debug APB, accesses complete in order.

However, the external agent cannot force synchronization of completed writes without halting the PE. Executing an ISB instruction, either in Debug state or in Non-debug state, and exiting from Debug state forces synchronization. If the PE is in Debug state, executing an ISB instruction is guaranteed to explicitly synchronize any external reads, external writes, and changes to the authentication interface that are ordered before the external write to EDITR.

For any given observer, external writes to the following register groups are guaranteed to be observable in the same order in which they complete:

- The breakpoint registers, DBGBCR<n>_EL1 and DBGBVR<n>_EL1.
- The watchpoint registers, DBGWCR<n>_EL1 and DBGWVR<n>_EL1.
This guarantee applies only to external writes to registers within one of these groups. There is no guarantee regarding the ordering of the observability of external writes within these groups with respect to external writes to registers, for example EDSCR, or between breakpoints and watchpoints, including watchpoints linked to context matching breakpoints.

Note
This means that a debugger can rely on the external writes to be observed in the same order in which they complete. It does not mean that a debugger can rely on the external writes being observed in finite time.

In a simple sequential execution an indirect write that occurs as a side-effect of an access happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect.

If two or more interfaces simultaneously access a register, the behavior must be as if the accesses occurred atomically and in any order. This is described in Examples of the synchronization of changes to the external debug registers.

Some registers have the property that for certain bits a write of 0 is ignored and a write of 1 has an effect. This means that simultaneous writes must be merged. Registers that have this property and support both external debug and System register access include DBGCLAIMSET_EL1, DBGCLAIMCLR_EL1, PMCR_EL0;{C,P}, PMOVSET_EL0, PMOVSCLR_EL0, PMCNTENSEL0, PMCNTENCCLR_EL0, PMINTENSEL1, PMINTENCLR_EL1, and PMSWINC_EL0. This last register is OPTIONAL and deprecated in the external debug interface.

H8.3.1 Synchronization and the authentication interface
Changes to the authentication interface are indirect writes to the state of the PE by the Requester of the authentication interface.

For an external debug interface read of any Authentication Status register, or an indirect read of the authentication interface made in determining the response to a subsequent external debug interface access, a change on the authentication interface must be observable following a subsequent explicit Context synchronization event, and:

• It is IMPLEMENTATION DEFINED whether a change is guaranteed to be observable in finite time.
• It is IMPLEMENTATION DEFINED whether a change is guaranteed to be observable following an entry to Debug state.

For a System register read of DBGAUTHSTATUS_EL1, a change on the authentication interface is guaranteed to be observable only after a Context synchronization event.

Note
• In some systems, the authentication interface is fixed by configuration or is changed under the control of software. These systems can require explicit synchronization for any change to the authentication interface.
• In other systems, the authentication interface is controlled dynamically by an external agent. In these systems, it is desirable that changes to the authentication interface do not require explicit synchronization by software executing on the PE to be observable by subsequent external debug interface accesses, and are either observable in finite time or are synchronized by entry to Debug state. Otherwise, there are scenarios where a debugger is not able to halt and debug the system.

H8.3.2 Examples of the synchronization of changes to the external debug registers
Example H8-1, Example H8-2 on page H8-10360, and Example H8-3 on page H8-10360 show the synchronization of changes to the external debug registers.

Example H8-1 Order of synchronization of Breakpoint and Watchpoint register writes
Initially DBGVR<n>_EL1 is 0x8000 and DBGBCR<n>_EL1 is 0x0181. This means that a breakpoint is enabled on the halfword T32 instruction at address 0x8000.
A sequence of external writes occurs in the following order:
1. $0x0000$ is written to $\text{DBGBCR}<n>_\text{EL1}$, disabling the breakpoint.
2. $0x9000$ is written to $\text{DBGBVR}<n>_\text{EL1}[31:0]$.
3. $0x061$ is written to $\text{DBGBCR}<n>_\text{EL1}$, enabling a breakpoint on the halfword at address $0x9002$.

The external writes must be observable to indirect reads in the same order as the external writes complete. This means that at no point is there a breakpoint enabled on either of the halfwords at address $0x8002$ and $0x9000$.

Similarly a breakpoint or watchpoint must be disabled:
• If both halves of a 64-bit address have to be updated.
• If any of the $\text{DBGBCR}<n>_\text{EL1}$ or $\text{DBGWCR}<n>_\text{EL1}$ fields are modified at the same time as updating the address.

---

### Example H8-2 Simultaneous accesses to DTR registers

Initially $\text{EDSCR.}\{\text{TXfull, TXU, ERR}\}$ are 0. Then:
• $0x0CDA7A$ is directly written to $\text{DBGDTRTX}_\text{EL0}$ by an MSR instruction.
• $\text{DBGDTRTX}_\text{EL0}$ is indirectly read by the external debug interface.

These accesses might happen at the same time and in any order.

If the direct write of $0x0CDA7A$ to $\text{DBGDTRTX}_\text{EL0}$ is handled first, then:
• The external debug interface read of $\text{DBGDTRTX}_\text{EL0}$ clears $\text{EDSCR.TXfull}$ to 0.
• $\text{EDSCR.}\{\text{TXU, ERR}\}$ are unchanged.
• The external debug interface read returns $0x0CDA7A$.

If the indirect read of $\text{DBGDTRTX}_\text{EL0}$ by the external debug interface is handled first, then:
• The external debug interface read of $\text{DBGDTRTX}_\text{EL0}$ causes an underrun and as a result $\text{EDSCR.}\{\text{TXU, ERR}\}$ are both set to 1.
• The external debug interface returns an \text{UNKNOWN} value.
• Writing $0x0CDA7A$ to $\text{DBGDTRTX}_\text{EL0}$ sets $\text{DTRTX}$ to $0x0CDA7A$ and $\text{EDSCR.TXfull}$ to 1.

---

### Example H8-3 Simultaneous writes to CLAIM registers

Initially all CLAIM tag bits are 0. Then:
• $0x01$ is written to $\text{DBGCLAIMSET}_\text{EL1}$ by a direct write, followed by an explicit \text{Context synchronization event}.
• $0x02$ is written to $\text{DBGCLAIMSET}_\text{EL1}$ by an external write.

These events might happen at the same time and in either order.

After this:
• $\text{DBGCLAIMCLR}_\text{EL1}$ is read by a direct read.
• $\text{DBGCLAIMCLR}_\text{EL1}$ is read by an external read.

In this case, a direct read can return either $0x01$ or $0x03$, and the external read can return either $0x02$ or $0x03$.

The only permitted final result for the CLAIM tags is the value $0x03$, because this would be the result regardless of whether $0x01$ or $0x02$ is written first. This is because the external write is guaranteed to be observable to a direct read in finite time. See \text{Synchronization requirements for AArch64 System registers} on page D17-5547.

It is not possible for a direct read to return $0x01$ and the external read to return $0x02$, because the writes to $\text{DBGCLAIMCLR}_\text{EL1}$ are serialized.
In the following scenario, there is only one permitted result. Both observers observe the value 0x03, and then, at the same time, two writes occur:

- 0x04 is written to DBGCLAIMSET_EL1 by a direct write, followed by an explicit *Context synchronization event*.
- 0x01 is written to DBGCLAIMCLR_EL1 by an external write.

In this case, the only permitted final result for the CLAIM tags is the value 0x06.
H8.4 Memory-mapped accesses to the external debug interface

Support for memory-mapped access to the external debug interface is OPTIONAL. When memory-mapped access to the external debug interface is supported, the external debug interface is accessed as a little-endian memory-mapped peripheral.

If the external debug interface is CoreSight compliant, then an OPTIONAL Software Lock can be implemented for memory-mapped accesses to each component.

The Software Lock is OPTIONAL and deprecated. If FEAT_Debugv8p4 is implemented, the Software Lock is not implemented. If it is not implemented, the behavior is as if it is unlocked. The Software Locks are controlled by EDLSR and EDLAR, PMLSR and PMLAR, and CTILSR and CTILAR. See Management registers and CoreSight compliance on page K6-11688.

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components in the Core power domain.

With the exception of these registers and the effect of the Software Lock, the behavior of the memory-mapped accesses is the same as for other accesses to the external debug interface.

Note: The recommended memory-mapped accesses to the external debug interface are not compatible with the memory-mapped interface defined in Armv7. In particular:

- The memory map is different.
- Memory-mapped accesses do not behave differently to Debug Access Port accesses when OSLSR.OSLK == 1, meaning that the OS Lock is locked.

The following sections give more information about these memory-mapped accesses:

- Register access permissions for memory-mapped accesses.
- Synchronization of memory-mapped accesses to external debug registers on page H8-10363.

See also Supported access sizes on page I1-10716.

H8.4.1 Register access permissions for memory-mapped accesses

It is IMPLEMENTATION DEFINED whether unprivileged memory-mapped accesses are allowed. Privileged software is responsible for controlling memory-mapped accesses using the MMU.

If FEAT_Debugv8p4 is implemented, the Secure view of a debug component is mapped into Secure physical memory and the Non-secure view is mapped into Non-secure physical memory.

If FEAT_Debugv8p4 is implemented, the access permissions are different in each Security state, but Secure and Non-secure views of the debug components are identical. Arm recommends the views are located at the same address in the Secure and Non-secure physical address maps.

If memory-mapped accesses are made through an ADIv5 interface, the Debug Access Port can block the access using DBGSWENABLE. This is outside the scope of the Arm architecture. See Arm® Debug Interface Architecture Specification ADIv5.0 to ADIv5.2.

Effect of the OPTIONAL Software Lock on memory-mapped access

For memory-mapped accesses, if other controls permit access to a register, the OPTIONAL Software Lock is implemented, and EDLSR.SLK, PMLSR.SLK, or CTILSR.SLK is set to 1, meaning the Software Lock is locked, then with the exception of the LAR itself:

- If other controls permit access to a register, then writes are ignored. That is:
  - Read/write (RW) registers become read-only, writes ignored (RO/WI).
  - Write-only (WO) registers become writes ignored (WI).
- Reads and writes have no side-effects. A side-effect is where a direct read or a direct write of a register creates an indirect write of the same or another register. When the Software Lock is locked, the indirect write does not occur.
• Writes to EDLAR, PMLAR, and CTILAR are unaffected.

This behavior must also apply to all IMPLEMENTATION DEFINED registers.

For example, if EDLSR.SLK is set to 1:
• EDSCR.\{TXfull, TXU, ERR\} are unchanged by a memory-mapped read from DBGDTRTX_EL0.
• EDSCR.\{RXfull, RXO, ERR\} are unchanged by a memory-mapped write to DBGDTRRX_EL0 that is ignored.
• EDSCR.\{ITE, ITO, ERR\} are unchanged by a memory-mapped write to EDITR that is ignored.
• OSLSR.OSLK is unchanged by a memory-mapped write to OSLAR_EL1 that is ignored.
• EDPCSR[63:32], EDCIDSR, and EDVIDSR are unchanged by a memory-mapped read from EDPCSR[31:0].

--- Note ---
Updating EDVIDSR, EDCIDSR, and EDPCSRhi are side-effects of reading EDPCSRlo, such that these registers contain the matching context for EDPCSRlo. The process that updates EDPCSRlo with PC samples is not a side-effect of the access. Reads of EDPCSRlo made when the Software Lock is locked can be used to profile software.

--- Note ---
Updating PMVDSR, PMCID1SR/PMCID2SR, and PMVIDSR are unchanged by a memory-mapped read from PMPCSR[31:0].

--- Note ---
Updating PMVDSR, PMCID1SR/PMCID2SR, and PMPCSR[31:0] are side-effects of reading PMPCSR[63:32], such that these registers contain the matching context for PMPCSR[63:32]. The process that updates PMPCSR[63:32] with PC samples is not a side-effect of the access. Reads of PMPCSR[63:32] made when the Software Lock is locked can be used to profile software.

• EDPRSR.\{SDR, SPMAD, SDAD, SR, SPD\} are unchanged by a memory-mapped read from EDPRSR.
• EDPRSR.SDAD is not set if an error response is returned due to a memory-mapped read or write of any debug register as the result of the value of the EDAD field.
• The CLAIM tags are unchanged by memory-mapped writes to DBGCLAIMSET_EL1 and DBGCLAIMCLR_EL1 which are ignored.

Similarly, if PMLSR.SLK is set to 1, then EDPRSR.SPMAD is not set if an error response is returned to a memory-mapped read or write of any Performance Monitors register due to the value of the EPMAD field.

**Behavior of a not permitted memory-mapped access**

Where the architecture requires that an external debug interface access generates an error response, a memory-mapped access must also generate an error response. However, it is IMPLEMENTATION DEFINED how the error response is handled, as this depends on the system.

Arm recommends that the error is returned as either:
• A synchronous external Data Abort.
• An SError interrupt.

**H8.4.2 Synchronization of memory-mapped accesses to external debug registers**

The synchronization requirements for memory-mapped accesses to the external debug interface is described in *Synchronization of changes to the external debug registers* on page H8-10358 and *Synchronization of memory-mapped registers* on page I1-10718.

The synchronization requirements between different routes to the external debug interface, that is, between Debug Access Port accesses and memory-mapped accesses are IMPLEMENTATION DEFINED.
H8.5 External debug interface register access permissions

Some external accesses to debug registers and Performance Monitor registers are not permitted and return an error response if:

- The Core power domain is powered down or is in low-power state where the registers cannot be accessed.
- OSLSR.OSLK == 1. The OS Lock is locked.
- FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE. The OS Double Lock is locked.
- The access is disabled by either the authentication interface or secure monitor.

Not all registers are affected in all of these cases. For more information, see External debug interface register access permissions summary on page H8-10365.

H8.5.1 External debug over powerdown and locks

Accessing registers using the external debug interface is not possible when the Debug power domain is off. In this case, all accesses return an error.

External accesses to debug and Performance Monitors registers in the Core power domain are not permitted and return an error response if:

- The Core power domain is off or in low-power state where the registers cannot be accessed.
- OSLSR.OSLK == 1, meaning that the OS Lock is locked. This allows software to prevent external debugger modification of the registers while it saves and restores them over powerdown.
- FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE. This means that the OS Double Lock is locked. The OS Double Lock ensures that it is safe to remove Core power by forcing the debug interface to be quiescent.

If FEAT_DoubleLock is not implemented, the hardware must provide another method to safely remove Core power.

The OS Lock condition does not apply to the following debug registers:

- OSLAR_EL1. This means that an external debugger can override this lock.
- EDESR. This means that an external debugger can program a debug event for when software unlocks the OS Lock. See OS Unlock Catch debug event on page H3-10292.
- The ID registers that describe the PE to the debugger.

See also Debug registers to save over powerdown on page H6-10343.

H8.5.2 External access disabled

Accesses are further controlled by the external authentication interface. An untrusted external debugger cannot program the breakpoint and watchpoint registers to generate spurious debug exceptions. If external invasive debugging is not enabled, these external accesses to the registers are disabled. If EL3 is implemented, then SDCR provides additional external access controls for those registers.

The disable applies to:

- The DBGVR<el1>, DBGBCR<el1>, DBGWVR<el1>, and DBGWCR<el1> registers.
- From Armv8.2, the OSLAR_EL1 register.

If FEAT_Debugv8p2 is not implemented, it is IMPLEMENTATION DEFINED whether the disable applies to OSLAR_EL1.

If FEAT_Debugv8p4 is not implemented, the external debug interface cannot access these registers if any of the following are true:

- ExternalInvasiveDebugEnabled() == FALSE.
- ExternalSecureInvasiveDebugEnabled() == FALSE, EL3 is not implemented, and the PE behaves as if the Security state is Secure.
- ExternalSecureInvasiveDebugEnabled() == FALSE, EL3 is implemented and SDCR.EDAD == 1.

If FEAT_Debugv8p4 is implemented, Non-secure accesses from the external debug interface to these registers are not permitted if any of the following are true:

- EL3 is not implemented and the PE behaves as if the Security state is Secure.
• EL3 is implemented and SDCR.EDAD == 1.

The AllowExternalDebugAccess() pseudocode function describes these accessibility rules.

PEs might also provide an OPTIONAL external debug interface to the Performance Monitor registers. The authentication interface and SDCR provide similar external access disable controls for those registers.

If FEAT_Debugv8p4 is not implemented, the external debug interface cannot access the Performance Monitor registers if any of the following are true:
• ExternalNoninvasiveDebugEnabled() == FALSE.
• ExternalSecureNoninvasiveDebugEnabled() == FALSE, EL3 is not implemented and the PE behaves as if the Security state is Secure.
• ExternalSecureNoninvasiveDebugEnabled() == FALSE, EL3 is implemented and SDCR.EPMAD == 1.

Note
• Arm recommends that Secure software that is not making use of debug hardware does not lock out the external debug interface.
• The Arm architecture does not provide the equivalent control over access to Trace extension registers, which means if FEAT_Debugv8p4 is implemented, the Non-secure and Secure views are identical.

If FEAT_Debugv8p4 is implemented, Non-secure accesses from the external debug interface to these registers are not permitted if any of the following are true:
• EL3 is not implemented and the PE behaves as if the Security state is Secure.
• EL3 is implemented and SDCR.EPMAD == 1.

The AllowExternalPMUAccess() pseudocode function describes these accessibility rules.

H8.5.3 Behavior of a not permitted access

For an external debug interface access by a Debug Access Port, the Debug Access Port receives the error response and must signal this to the external debugger. For an ADIv5 implementation of a Debug Access Port, the error sets a sticky error flag in the Debug Access Port that the debugger can poll, and that suppresses further accesses until it is explicitly cleared.

When an error is returned because external access is disabled, and this is the highest priority error condition, a sticky error flag in EDPRSR is indirectly written to 1 as a side-effect of the access:
• For a debug register access when AllowExternalDebugAccess() == FALSE, EDPRSR.SDAD is indirectly written to 1.
• For Performance Monitor register access when AllowExternalPMUAccess() == FALSE, EDPRSR.SPMAD is indirectly written to 1.

The indirect write might not occur for a memory-mapped access to the external debug interface. For more information, see Register access permissions for memory-mapped accesses on page H8-10362.

If no error is returned, or the error is returned because of a higher priority error condition, the flag in EDPRSR is unchanged.

See also Behavior of a not permitted memory-mapped access on page H8-10363.

For more information, see Arm® Debug Interface Architecture Specification.

H8.5.4 External debug interface register access permissions summary

For accesses to:
• IMPLEMENTATION DEFINED registers, see IMPLEMENTATION DEFINED registers on page H8-10366.
• OPTIONAL registers for CoreSight compliance, see Management registers and CoreSight compliance on page K6-11688.
• Reserved, unallocated, or unimplemented registers, writes to read-only registers, and reads of write-only registers, see Reserved and unallocated registers on page H8-10366.
For all other external debug interface, CTI, and Performance Monitor registers, Table H8-3 on page H8-10371, Table H8-4 on page H8-10373, Table H8-6 on page H8-10376 and Table I3-2 on page I3-10738, show the response of the PE to accesses by the external debug interface.

H8.5.5 IMPLEMENTATION DEFINED registers

For debug registers, Performance Monitors registers, CTI registers, register access permissions for IMPLEMENTATION DEFINED registers are IMPLEMENTATION DEFINED.

If OPTIONAL memory-mapped access to the external debug interface is supported, there are additional constraints on memory-mapped accesses to registers. These constraints must also apply to IMPLEMENTATION DEFINED registers. For more information, see Register access permissions for memory-mapped accesses on page H8-10362.

If FEAT_DoPD is not implemented, the power domain of these registers in which these registers are implemented is also IMPLEMENTATION DEFINED. The registers must apply the constraint that if the OPTIONAL Software Lock is locked, writes are ignored and accesses have no side-effects.

If FEAT_DoPD is implemented, then:

- For debug registers and Performance Monitors registers, IMPLEMENTATION DEFINED registers are implemented in the Core power domain. Accesses return an error when the Core power domain is off or in a low-power state.
- For CTI registers, IMPLEMENTATION DEFINED registers are implemented in the Debug power domain.

H8.5.6 Reserved and unallocated registers

The default access requirements for reserved and unallocated registers are described in Access requirements for reserved and unallocated registers on page I1-10720.

Note

Reads of WO and writes to RO refers to the default access permissions for a register. For example, when the SLK field is set, meaning that the relevant registers become RO, a memory-mapped write to a RW register is ignored, and not treated as a reserved access.

The following reserved registers are RES0 in all conditions, other than when debug power is off:

- All reserved CTI registers.
- For the debug registers, and Performance Monitors registers, if the implementation is CoreSight architecture compliant, and either FEAT_DoPD is not implemented or the Core power domain is on, all reserved registers in the range 0xF00 - 0xFFC. See Management register access permissions on page K6-11689.

Otherwise, the architecture defines that:

1. If debug power is off, all register accesses, including reserved accesses, return an error.

2. For reserved debug registers and Performance Monitors registers, if FEAT_DoPD is implemented, and the Core power domain is off or in a low-power state, the response is an error. Otherwise, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0, when any of the following hold:

   - Off: The Core power domain is either completely off or in a low-power state in which the Core power domain registers cannot be accessed.
   - DLK: FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE. The OS Double Lock is locked.
   - OSLK: OSLSR.OSLK == 1. The OS Lock is locked.

3. In addition, for reserved debug registers in the address ranges 0x400 - 0x4FC and 0x800 - 0x8FC, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when conditions 1 or 2 do not apply and:

   - EDAD: AllowExternalDebugAccess() == FALSE. External debug is disabled.

Note

See also Behavior of a not permitted access on page H8-10365.
4. In addition, for reserved Performance Monitors registers in the address ranges 0x000 - 0xEFC, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when conditions 1 or 2 do not apply and:

   EPMAD AllowExternalPMUAccess() == FALSE. External Performance Monitor access is disabled.

   **Note**

   See also *Behavior of a not permitted access* on page H8-10365.

5. For reads of WO locations, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when the architecture permits or requires a write to the location to return an error.

6. For writes of RO locations, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when the architecture permits or requires a read to the location to return an error.

7. For reads and writes of locations for features that are not implemented, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when the architecture permits or requires an access to the location to return an error if the feature is implemented.
## H8.6 External debug interface registers

The external debug interface register map is described by:
- *External debug interface register map on page H8-10368.*
- *Cross-trigger interface registers on page H8-10375.*
- *Performance Monitors external register views on page I5-10748.*

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Description</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x020</td>
<td>EDESR</td>
<td>External Debug Event Status Register</td>
<td>RW</td>
</tr>
<tr>
<td>0x024</td>
<td>EDECR</td>
<td>External Debug Execution Control Register</td>
<td>RW</td>
</tr>
<tr>
<td>0x030</td>
<td>EDWAR[31:0]</td>
<td>External Debug Watchpoint Address Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x034</td>
<td>EDWAR[63:32]</td>
<td>External Debug Watchpoint Address Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x038</td>
<td>EDHSR</td>
<td>External Debug Halt Status Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x03C</td>
<td>EDHSR</td>
<td>External Debug Halt Status Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x080</td>
<td>DBGDTRRX_EL0</td>
<td>Debug Data Transfer Register, Receive</td>
<td>RW</td>
</tr>
<tr>
<td>0x084</td>
<td>EDITR</td>
<td>External Debug Instruction Transfer Register</td>
<td>WO</td>
</tr>
<tr>
<td>0x088</td>
<td>EDSCR</td>
<td>External Debug Status and Control Register</td>
<td>RW</td>
</tr>
<tr>
<td>0x08C</td>
<td>DBGDTRTX_EL0</td>
<td>Debug Data Transfer Register, Transmit</td>
<td>RW</td>
</tr>
<tr>
<td>0x090</td>
<td>EDRCR</td>
<td>External Debug Reserve Control Register</td>
<td>WO</td>
</tr>
<tr>
<td>0x094</td>
<td>EDACR</td>
<td>External Debug Auxiliary Control Register</td>
<td>RW</td>
</tr>
<tr>
<td>0x098</td>
<td>EDECCR</td>
<td>External Debug Exception Catch Control Register</td>
<td>RW</td>
</tr>
<tr>
<td>0x0A0</td>
<td>EDPCSRlo</td>
<td>External Debug Program Counter Sample Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x0A4</td>
<td>EDCIDSRO</td>
<td>External Debug Context ID Sample Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x0A8</td>
<td>EDVIDSR</td>
<td>External Debug Virtual Context Sample Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x0AC</td>
<td>EDPCSRhi</td>
<td>External Debug Program Counter Sample Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x300</td>
<td>OSLAR_EL1</td>
<td>OS Lock Access Register</td>
<td>WO</td>
</tr>
<tr>
<td>0x310</td>
<td>EDPRCSR</td>
<td>External Debug Power/Reset Control Register</td>
<td>RW</td>
</tr>
<tr>
<td>0x314</td>
<td>EDPRSR</td>
<td>External Debug Processor Status Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x400 + (16 * n)</td>
<td>DBGBVR&lt;(n)&gt;_EL1[31:0]</td>
<td>Debug Breakpoint Value Registers</td>
<td>RW</td>
</tr>
<tr>
<td>0x404 + (16 * n)</td>
<td>DBGBVR&lt;(n)&gt;_EL1[63:32]</td>
<td>Debug Breakpoint Value Registers</td>
<td>RW</td>
</tr>
<tr>
<td>0x408 + (16 * n)</td>
<td>DBGBCR&lt;(n)&gt;_EL1</td>
<td>Debug Breakpoint Control Registers</td>
<td>RW</td>
</tr>
<tr>
<td>0x800 + (16 * n)</td>
<td>DBGWVVR&lt;(n)&gt;_EL1[31:0]</td>
<td>Debug Watchpoint Value Registers</td>
<td>RW</td>
</tr>
<tr>
<td>0x804 + (16 * n)</td>
<td>DBGWVVR&lt;(n)&gt;_EL1[63:22]</td>
<td>Debug Watchpoint Value Registers</td>
<td>RW</td>
</tr>
<tr>
<td>0x808 + (16 * n)</td>
<td>DBGWCR&lt;(n)&gt;_EL1</td>
<td>Debug Watchpoint Control Registers</td>
<td>RW</td>
</tr>
<tr>
<td>0xC00-0xCF0</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED</td>
<td>-</td>
</tr>
<tr>
<td>0x000</td>
<td>MIDR_EL1</td>
<td>Main ID Register</td>
<td>RO</td>
</tr>
<tr>
<td>Offset</td>
<td>Name</td>
<td>Description</td>
<td>Access</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------</td>
<td>--------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>0x004-0x01C</td>
<td>-</td>
<td>Reserved, RES0</td>
<td>-</td>
</tr>
<tr>
<td>0x020</td>
<td>EDPFR[31:0]</td>
<td>External Debug Processor Feature Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x024</td>
<td>EDPFR[63:32]</td>
<td>External Debug Processor Feature Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x028</td>
<td>EDDFR[31:0]</td>
<td>External Debug Feature Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x02C</td>
<td>EDDFR[63:32]</td>
<td>External Debug Feature Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x030</td>
<td>-</td>
<td>Reserved, RES0</td>
<td>-</td>
</tr>
<tr>
<td>0x034</td>
<td>-</td>
<td>RES0</td>
<td>-</td>
</tr>
<tr>
<td>0x038</td>
<td>-</td>
<td>UNKNOWN</td>
<td>-</td>
</tr>
<tr>
<td>0x03C</td>
<td>-</td>
<td>RES0</td>
<td>-</td>
</tr>
<tr>
<td>0x040-0x0FC</td>
<td>-</td>
<td>Reserved, RES0</td>
<td>-</td>
</tr>
<tr>
<td>0x060</td>
<td>EDAA32PFR[31:0]</td>
<td>External Debug Auxiliary Processor Feature Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x060</td>
<td>EDAA32PFR[63:32]</td>
<td>External Debug Auxiliary Processor Feature Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x080-0x0EF</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED</td>
<td>-</td>
</tr>
<tr>
<td>0x080</td>
<td>EDITCTRL</td>
<td>External Debug Integration mode Control register</td>
<td>RW</td>
</tr>
<tr>
<td>0x0A0</td>
<td>DBGCLAIMSET_EL1</td>
<td>Debug CLAIM Tag Set register</td>
<td>RW</td>
</tr>
<tr>
<td>0x0A4</td>
<td>DBGCLAIMCLR_EL1</td>
<td>Debug CLAIM Tag Clear register</td>
<td>RW</td>
</tr>
<tr>
<td>0x0A8</td>
<td>EDDEVAFF0</td>
<td>External Debug Device Affinity register 0</td>
<td>RO</td>
</tr>
<tr>
<td>0x0AC</td>
<td>EDDEVAFF1</td>
<td>External Debug Device Affinity register 1</td>
<td>RO</td>
</tr>
<tr>
<td>0x0B0</td>
<td>EDLAR</td>
<td>External Debug Lock Access Register</td>
<td>WO</td>
</tr>
<tr>
<td>0x0B4</td>
<td>EDLSR</td>
<td>External Debug Lock Status Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x0B8</td>
<td>DBGAUTHSTATUS_EL1</td>
<td>Debug Authentication Status register</td>
<td>RO</td>
</tr>
<tr>
<td>0x0BC</td>
<td>EDDEVAARCH</td>
<td>External Debug Device Architecture register</td>
<td>RO</td>
</tr>
<tr>
<td>0x0C0</td>
<td>EDDEVID2</td>
<td>External Debug Device ID register 2</td>
<td>RO</td>
</tr>
<tr>
<td>0x0C4</td>
<td>EDDEVID1</td>
<td>External Debug Device ID register 1</td>
<td>RO</td>
</tr>
<tr>
<td>0x0C8</td>
<td>EDDEVID</td>
<td>External Debug Device ID register 0</td>
<td>RO</td>
</tr>
<tr>
<td>0x0CC</td>
<td>EDDEVTYPE</td>
<td>External Debug Device Type register</td>
<td>RO</td>
</tr>
<tr>
<td>0x0D0</td>
<td>EDPIDR4</td>
<td>External Debug Peripheral Identification Register 4</td>
<td>RO</td>
</tr>
<tr>
<td>0x0E0</td>
<td>EDPIDR0</td>
<td>External Debug Peripheral Identification Register 0</td>
<td>RO</td>
</tr>
<tr>
<td>0x0E4</td>
<td>EDPIDR1</td>
<td>External Debug Peripheral Identification Register 1</td>
<td>RO</td>
</tr>
<tr>
<td>0x0E8</td>
<td>EDPIDR2</td>
<td>External Debug Peripheral Identification Register 2</td>
<td>RO</td>
</tr>
<tr>
<td>0x0EC</td>
<td>EDPIDR3</td>
<td>External Debug Peripheral Identification Register 3</td>
<td>RO</td>
</tr>
<tr>
<td>0xFF0</td>
<td>EDCIDR0</td>
<td>External Debug Component Identification Register 0</td>
<td>RO</td>
</tr>
</tbody>
</table>
About the External Debug Registers

H8.6 External debug interface registers

DBGBVR<n>_EL1 and DBGWVR<n>_EL1 are 64-bit registers mapped to pairs of 32-bit locations. Doubleword accesses to these registers are not guaranteed to be 64-bit single copy atomic. Software must ensure a breakpoint or watchpoint is disabled before altering the value registers. See Endianness and supported access sizes on page H8-10357 for more information.

H8.6.1 Access permissions for the External debug interface registers

Table H8-3 on page H8-10371 and Table H8-4 on page H8-10373 show the access permissions for the external debug interface registers in an Arm A-profile Debug implementation. The terms are defined as follows:

<table>
<thead>
<tr>
<th>Domain</th>
<th>Conditions</th>
<th>Description</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>The Core power domain is completely off, or in low-power state. In these cases the Core power domain registers cannot be accessed, and if FEAT_DoPD is not implemented, EDPRSR.PU will read as 0.</td>
<td>The Core power domain is logically implemented. Registers described as implemented in the Core power domain might be implemented in the Debug power domain, as long as they exhibit the required behavior.</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>Note</td>
<td>If FEAT_DoPD is implemented, most External debug interface registers are in the Core power domain, as shown in Table H8-3 on page H8-10371. If FEAT_DoPD is not implemented, most of the registers are in the Debug Power Domain, as shown in Table H8-4 on page H8-10373.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>To determine the access permissions for a register, read these columns from left to right, and stop at first column that lists the condition as being true.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DLK</td>
<td>FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE. The OS Double Lock is locked. If FEAT_DoPD is implemented, FEAT_DoubleLock is not implemented and so Table H8-3 on page H8-10371 does not include this column.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OSLK</td>
<td>OSLSR.OSLK == 1. The OS Lock is locked.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDAD</td>
<td>AllowExternalDebugAccess() == FALSE. External debug access is disabled for the access. If FEAT_Debugv8p4 is implemented, this applies only for Non-secure accesses to the register. See also Behavior of a not permitted access on page H8-10365.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EPMAD</td>
<td>AllowExternalPMUAccess() == FALSE. Access to the external Performance Monitors is disabled for the access. If FEAT_Debugv8p4 is implemented, this applies only for Non-secure accesses to the register. See also Behavior of a not permitted access on page H8-10365.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SLK</td>
<td>The Software Lock is implemented and SoftwareLockStatus() == TRUE. This provides the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the OPTIONAL Software Lock is locked. See Register access</td>
<td></td>
</tr>
</tbody>
</table>

Table H8-2 External debug interface register map (continued)

Offset | Name | Description | Access |
--------|------|-------------|--------|
0xFF4   | EDCIDR1 | External Debug Component Identification Register 1 | RO |
0xFF8   | EDCIDR2 | External Debug Component Identification Register 2 | RO |
0xFFC   | EDCIDR3 | External Debug Component Identification Register 3 | RO |
permissions for memory-mapped accesses on page H8-10362. If FEAT_DoPD is implemented, the Software Lock is not locked or not implemented, this column is ignored.

Default  This provides the default access permissions, if there are no conditions that prevent access to the register.

The access permissions are:

- This means that the default access permission applies. See the Default column, or the SLK column, if applicable.

RO  This means that the register or field is read-only, and:

  - Unless the register description states otherwise, a RO field in an RW register ignores writes.
  - Where the SLK control makes a RW register RO, the register ignores writes.

RW  This means that the register or field is read/write. Individual fields within the register might be RO or WO. See the relevant register description for details.

RC  This means that a read of the register bit clears the field to 0.

WO  This means that the register or field is write-only. Unless the register description states otherwise, a WO field in a RW register returns an UNKNOWN value on a read of the register.

WI  This means that the register or field ignores writes.

IMP DEF  This means that the access permissions are IMPLEMENTATION DEFINED.

If OPTIONAL memory-mapped access to the external debug interface is supported, there might be additional constraints on memory-mapped accesses. See Register access permissions for memory-mapped accesses on page H8-10362.

For the reset values for the external debug interface registers, see Table H8-7 on page H8-10378.

Table H8-3 Access permissions for the external debug interface registers if FEAT_DoPD is implemented

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Conditions (priority from left to right)</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x020</td>
<td>EDESR</td>
<td>Core</td>
<td>Error - - -</td>
<td>RW</td>
</tr>
<tr>
<td>0x024</td>
<td>EDECR</td>
<td>Core</td>
<td>Error - - -</td>
<td>RW</td>
</tr>
<tr>
<td>0x030</td>
<td>EDWAR[31:0]</td>
<td>Core</td>
<td>Error - Error -</td>
<td>RO</td>
</tr>
<tr>
<td>0x034</td>
<td>EDWAR[63:32]</td>
<td>Core</td>
<td>Error - Error -</td>
<td>RO</td>
</tr>
<tr>
<td>0x080</td>
<td>DBGDTRRX_EL0</td>
<td>Core</td>
<td>Error - Error -</td>
<td>RW</td>
</tr>
<tr>
<td>0x084</td>
<td>EDITR</td>
<td>Core</td>
<td>Error - Error -</td>
<td>WO</td>
</tr>
<tr>
<td>0x088</td>
<td>EDSCR</td>
<td>Core</td>
<td>Error - Error -</td>
<td>RW</td>
</tr>
<tr>
<td>0x08C</td>
<td>DBGDTRTX_EL0</td>
<td>Core</td>
<td>Error - Error -</td>
<td>RW</td>
</tr>
<tr>
<td>0x090</td>
<td>EDRCR</td>
<td>Core</td>
<td>Error - Error -</td>
<td>WO</td>
</tr>
<tr>
<td>0x094</td>
<td>EDACR</td>
<td>Core</td>
<td>Error - Error -</td>
<td>RW</td>
</tr>
<tr>
<td>0x098</td>
<td>EDECCR</td>
<td>Core</td>
<td>Error - Error -</td>
<td>RW</td>
</tr>
<tr>
<td>0x0A0</td>
<td>EDPCSR[31:0]</td>
<td>Core</td>
<td>Error - Error -</td>
<td>RO</td>
</tr>
<tr>
<td>0x0A4</td>
<td>EDCIDSRA</td>
<td>Core</td>
<td>Error - Error -</td>
<td>RO</td>
</tr>
</tbody>
</table>
## About the External Debug Registers

**H8.6 External debug interface registers**

### Table H8-3 Access permissions for the external debug interface registers if FEAT_DoPD is implemented (continued)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Conditions (priority from left to right)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0A8</td>
<td>EDVIDSRa</td>
<td>Core</td>
<td>Error - Error - RO</td>
</tr>
<tr>
<td>0x0AC</td>
<td>EDPCSR[63:32]a</td>
<td>Core</td>
<td>Error - Error - RO</td>
</tr>
<tr>
<td>0x300</td>
<td>OSLAR_EL1</td>
<td>Core</td>
<td>Error - - Error WO</td>
</tr>
<tr>
<td>0x310</td>
<td>EDPRCR</td>
<td>Core</td>
<td>Error - - - RW</td>
</tr>
<tr>
<td>0x314</td>
<td>EDPRSR</td>
<td>Core</td>
<td>Error - - - RO</td>
</tr>
<tr>
<td>0x400+16×n</td>
<td>DBGVR&lt;n&gt;_EL1[31:0]b</td>
<td>Core</td>
<td>Error - Error - RW</td>
</tr>
<tr>
<td>0x404+16×n</td>
<td>DBGVR&lt;n&gt;_EL1[63:32]b</td>
<td>Core</td>
<td>Error - Error - RW</td>
</tr>
<tr>
<td>0x408+16×n</td>
<td>DBGVR&lt;n&gt;_EL1[31:0]b</td>
<td>Core</td>
<td>Error - Error - RW</td>
</tr>
<tr>
<td>0x800+16×n</td>
<td>DBGVR&lt;n&gt;_EL1[63:32]b</td>
<td>Core</td>
<td>Error - Error - RW</td>
</tr>
<tr>
<td>0x808+16×n</td>
<td>DBGVR&lt;n&gt;_EL1[31:0]b</td>
<td>Core</td>
<td>Error - Error - RW</td>
</tr>
<tr>
<td>0x80C+16×n</td>
<td>DBGVR&lt;n&gt;_EL1[63:32]b</td>
<td>Core</td>
<td>Error - Error - RW</td>
</tr>
<tr>
<td>0x000</td>
<td>MIDE_R 1</td>
<td>Core</td>
<td>Error - - - RO</td>
</tr>
<tr>
<td>0x020</td>
<td>EDPF_R[31:0]</td>
<td>Core</td>
<td>Error - - - RO</td>
</tr>
<tr>
<td>0x024</td>
<td>EDPF[63:32]</td>
<td>Core</td>
<td>Error - - - RO</td>
</tr>
<tr>
<td>0x028</td>
<td>EDDF_R[31:0]</td>
<td>Core</td>
<td>Error - - - RO</td>
</tr>
<tr>
<td>0x02C</td>
<td>EDDF[63:32]</td>
<td>Core</td>
<td>Error - - - RO</td>
</tr>
<tr>
<td>0x060</td>
<td>EDAA32PFR[31:0]</td>
<td>Core</td>
<td>Error - - - RO</td>
</tr>
<tr>
<td>0x064</td>
<td>EDAA32PFR[63:32]</td>
<td>Core</td>
<td>Error - - - RO</td>
</tr>
<tr>
<td>0xFA0</td>
<td>DBGCLAIMSET_EL1</td>
<td>Core</td>
<td>Error - - - RO</td>
</tr>
<tr>
<td>0xFA4</td>
<td>DBGCLAIMCLR_EL1</td>
<td>Core</td>
<td>Error - - - RO</td>
</tr>
<tr>
<td>0xFA8</td>
<td>EDDEVAFF0</td>
<td>Core</td>
<td>Error - - - RO</td>
</tr>
<tr>
<td>0xFAC</td>
<td>EDDEVAFF1</td>
<td>Core</td>
<td>Error - - - RO</td>
</tr>
<tr>
<td>0xF88</td>
<td>DBGAUTHSTATUS_EL1</td>
<td>Core</td>
<td>Error - - - RO</td>
</tr>
<tr>
<td>0xFC0</td>
<td>EDDEVID2</td>
<td>Core</td>
<td>Error - - - RO</td>
</tr>
<tr>
<td>0xFC4</td>
<td>EDDEVID1</td>
<td>Core</td>
<td>Error - - - RO</td>
</tr>
<tr>
<td>0xFC8</td>
<td>EDDEVID</td>
<td>Core</td>
<td>Error - - - RO</td>
</tr>
</tbody>
</table>

a. Implemented only if the PC Sample-based Profiling Extension is implemented and FEAT_PCSRv8p2 is not implemented.

b. Implemented breakpoints and watchpoints only. $n$ is the breakpoint or watchpoint number.
### Table H8-4 Access permissions for the external debug interface registers if FEAT_DoPD is not implemented

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Conditions (priority from left to right)</th>
<th>Off</th>
<th>DLK</th>
<th>OSLK</th>
<th>EDAD</th>
<th>Default</th>
<th>SLK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x020</td>
<td>EDESR</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>-</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0x024</td>
<td>EDECR</td>
<td>Debug</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0x030</td>
<td>EDWAR[31:0]</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0x034</td>
<td>EDWAR[63:32]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x080</td>
<td>DBGDTRRX_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0x084</td>
<td>EDITR</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>WO</td>
<td>WI</td>
</tr>
<tr>
<td>0x088</td>
<td>EDSR</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0x08C</td>
<td>DBGDTRTX_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0x090</td>
<td>EDRCR</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>WO</td>
<td>WI</td>
</tr>
<tr>
<td>0x094</td>
<td>EDACR</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>-</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0x098</td>
<td>EDECCR</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0x0A0</td>
<td>EDPCSR[31:0](^a)</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0x0A4</td>
<td>EDCIDSR(^b)</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0x0A8</td>
<td>EDVIDSR(^b)</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0x0AC</td>
<td>EDPCSR[63:32]</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0x300</td>
<td>OSLAR_EL1</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>IM Def(^b)</td>
<td>WO</td>
</tr>
<tr>
<td>0x310</td>
<td>EDPRCR</td>
<td>Core and Debug</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0x314</td>
<td>EDPRSR</td>
<td>Core and Debug</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0x400+16xn</td>
<td>DBGBVR&lt;n&gt;_EL1[31:0]d</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0x404+16xn</td>
<td>DBGBVR&lt;n&gt;_EL1[63:32]d</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0x408+16xn</td>
<td>DBGBCR&lt;n&gt;_EL1d</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0x800+16xn</td>
<td>DBGWVR&lt;n&gt;_EL1[31:0]d</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0x804+16xn</td>
<td>DBGWVR&lt;n&gt;_EL1[63:32]d</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0x808+16xn</td>
<td>DBGWCR&lt;n&gt;_EL1d</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0xD00</td>
<td>MIDR_EL1</td>
<td>IMP DEF</td>
<td>IMP DEF(^e)</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xD20</td>
<td>EDPRF[31:0]</td>
<td>IMP DEF</td>
<td>IMP DEF(^e)</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xD24</td>
<td>EDPRF[63:32]</td>
<td>IMP DEF</td>
<td>IMP DEF(^e)</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xD28</td>
<td>EDDFR[31:0]</td>
<td>IMP DEF</td>
<td>IMP DEF(^e)</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xD2C</td>
<td>EDDFR[63:32]</td>
<td>IMP DEF</td>
<td>IMP DEF(^e)</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xD60</td>
<td>EDAA32PFR[31:0]</td>
<td>IMP DEF</td>
<td>IMP DEF(^e)</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xD64</td>
<td>EDAA32PFR[63:32]</td>
<td>IMP DEF</td>
<td>IMP DEF(^e)</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>IMP DEF</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
</tbody>
</table>

\(^a\) When embroidery is used, any access to this register causes a debug exception.

\(^b\) When embroidery is used, any access to this register changes the value of SLK.

\(^c\) A register with the Core and Debug domain is readable by the core and debug interface.

\(^d\) An access to the register is allowed when the corresponding access control bit is enabled.

\(^e\) Access control to the register is disabled.
About the External Debug Registers

H8.6 External debug interface registers

Table H8-4 Access permissions for the external debug interface registers if FEAT_DoPD is not implemented

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Off</th>
<th>DLK</th>
<th>OSLK</th>
<th>EDAD</th>
<th>Default</th>
<th>SLK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFA0</td>
<td>DBGCLAIMSET_EL1</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0xFA4</td>
<td>DBGCLAIMCLR_EL1</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0xFA8</td>
<td>EDDEVAFF0</td>
<td>Debug</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xFAC</td>
<td>EDDEVAFF1</td>
<td>Debug</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xFB8</td>
<td>DBGAUTHSTATUS_EL1</td>
<td>Debug</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xFC0</td>
<td>EDDEVID2</td>
<td>Debug</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xFC4</td>
<td>EDDEVID1</td>
<td>Debug</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xFC8</td>
<td>EDDEVID</td>
<td>Debug</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
</tbody>
</table>

- a. Implemented only if the PC Sample-based Profiling Extension is implemented.
- b. If FEAT_Debugv8p2 is not implemented, it is IMPLEMENTATION DEFINED whether an error is returned. See External access disabled on page H8-10364. If no error is returned, the access is permitted.
- c. Some bits are in the Debug power domain and some bits are in the Core power domain. See register field descriptions for information.
- d. Implemented breakpoints and watchpoints only. \( n \) is the breakpoint or watchpoint number.
- e. It is IMPLEMENTATION DEFINED whether an error is returned. See External debug over powerdown and locks on page H8-10364. If no error is returned, the access is permitted.
H8.7 Cross-trigger interface registers

The Embedded Cross-Trigger Interface, CTI, is located within its own block of the external debug memory map. There must be one such block for each PE.

If the CTI of a PE does not implement the CTIDEV AFF0 or CTIDEV AFF1 registers it must be located 64KB above the debug registers in the external debug interface.

When FEAT_Debugv8p4 is implemented, each debug component has a Secure and Non-secure view. The Secure view of a debug component is mapped into Secure physical memory and the Non-secure view of a debug component is mapped into Non-secure memory. Apart from access conditions, the Non-secure and Secure views of the debug components are identical.

Table H8-5 on page H8-10375 shows the CTI register map.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Description</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>CTICONTROL</td>
<td>CTI Control register</td>
<td>RW</td>
</tr>
<tr>
<td>0x010</td>
<td>CTIINTACK</td>
<td>CTI Output Trigger Acknowledge register</td>
<td>WO</td>
</tr>
<tr>
<td>0x014</td>
<td>CTIAPPSET</td>
<td>CTI Application Trigger Set register</td>
<td>RW</td>
</tr>
<tr>
<td>0x018</td>
<td>CTIAPPCLEAR</td>
<td>CTI Application Trigger Clear register</td>
<td>WO</td>
</tr>
<tr>
<td>0x01C</td>
<td>CTIAPPPULSE</td>
<td>CTI Application Pulse register</td>
<td>WO</td>
</tr>
<tr>
<td>0x020 + (4 * n)</td>
<td>CTIINEN&lt;n&gt;</td>
<td>CTI Input Trigger to Output Channel Enable registers</td>
<td>RW</td>
</tr>
<tr>
<td>0x040 + (4 * n)</td>
<td>CTIOUTEN&lt;n&gt;</td>
<td>CTI Input Channel to Output Trigger Enable registers</td>
<td>RW</td>
</tr>
<tr>
<td>0x130</td>
<td>CTITRIGINSTATUS</td>
<td>CTI Trigger In Status register</td>
<td>RO</td>
</tr>
<tr>
<td>0x134</td>
<td>CTITRIGOUTSTATUS</td>
<td>CTI Trigger Out Status register</td>
<td>RO</td>
</tr>
<tr>
<td>0x138</td>
<td>CTICHINSTATUS</td>
<td>CTI Channel In Status register</td>
<td>RO</td>
</tr>
<tr>
<td>0x13C</td>
<td>CTICHOUTSTATUS</td>
<td>CTI Channel Out Status register</td>
<td>RO</td>
</tr>
<tr>
<td>0x140</td>
<td>CTIGATE</td>
<td>CTI Channel Gate Enable register</td>
<td>RW</td>
</tr>
<tr>
<td>0x144</td>
<td>ASICCTL</td>
<td>CTI External Multiplexer Control register</td>
<td>RO</td>
</tr>
<tr>
<td>0x150</td>
<td>CTIDEVCTRL</td>
<td>CTI Device Control register</td>
<td>RW</td>
</tr>
<tr>
<td>0x1F0</td>
<td>CTITCTRL</td>
<td>CTI Integration mode Control register</td>
<td>RW</td>
</tr>
<tr>
<td>0x1F4</td>
<td>CTICLAIMSET</td>
<td>CTI CLAIM Tag Set register</td>
<td>RW</td>
</tr>
<tr>
<td>0x1F8</td>
<td>CTICLAIMCLR</td>
<td>CTI CLAIM Tag Clear register</td>
<td>RW</td>
</tr>
<tr>
<td>0x208</td>
<td>CTIDEV AFF0</td>
<td>CTI Device Affinity register 0</td>
<td>RO</td>
</tr>
<tr>
<td>0x20C</td>
<td>CTIDEV AFF1</td>
<td>CTI Device Affinity register 1</td>
<td>RO</td>
</tr>
<tr>
<td>0x210</td>
<td>CTILAR</td>
<td>CTI Lock Access Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x214</td>
<td>CTILSR</td>
<td>CTI Lock Status Register</td>
<td>RO</td>
</tr>
<tr>
<td>0x218</td>
<td>CTIAUTHSTATUS</td>
<td>CTI Authentication Status register</td>
<td>RO</td>
</tr>
<tr>
<td>0x21C</td>
<td>CTIDEVARCH</td>
<td>CTI Device Architecture register</td>
<td>RO</td>
</tr>
<tr>
<td>0xF00</td>
<td>CTIDEVID2</td>
<td>CTI Device ID register 2</td>
<td>RO</td>
</tr>
</tbody>
</table>
Table H8-5 Summary of external debug register resets, CTI registers (continued)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Description</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0C4</td>
<td>CTIDEVID1</td>
<td>CTI Device ID register 1</td>
<td>RO</td>
</tr>
<tr>
<td>0x0C8</td>
<td>CTIDEVID</td>
<td>CTI Device ID register 0</td>
<td>RO</td>
</tr>
<tr>
<td>0x0CC</td>
<td>CTIDEVTYPE</td>
<td>CTI Device Type register</td>
<td>RO</td>
</tr>
<tr>
<td>0x0D0</td>
<td>CTIPIDR4</td>
<td>CTI Peripheral Identification Register 4</td>
<td>RO</td>
</tr>
<tr>
<td>0x0E0</td>
<td>CTIPIDR0</td>
<td>CTI Peripheral Identification Register 0</td>
<td>RO</td>
</tr>
<tr>
<td>0x0E4</td>
<td>CTIPIDR1</td>
<td>CTI Peripheral Identification Register 1</td>
<td>RO</td>
</tr>
<tr>
<td>0x0E8</td>
<td>CTIPIDR2</td>
<td>CTI Peripheral Identification Register 2</td>
<td>RO</td>
</tr>
<tr>
<td>0x0EC</td>
<td>CTIPIDR3</td>
<td>CTI Peripheral Identification Register 3</td>
<td>RO</td>
</tr>
<tr>
<td>0x0F0</td>
<td>CTICIDR0</td>
<td>CTI Component Identification Register 0</td>
<td>RO</td>
</tr>
<tr>
<td>0x0F4</td>
<td>CTICIDR1</td>
<td>CTI Component Identification Register 1</td>
<td>RO</td>
</tr>
<tr>
<td>0x0F8</td>
<td>CTICIDR2</td>
<td>CTI Component Identification Register 2</td>
<td>RO</td>
</tr>
<tr>
<td>0x0FC</td>
<td>CTICIDR3</td>
<td>CTI Component Identification Register 3</td>
<td>RO</td>
</tr>
</tbody>
</table>

Table H8-6 on page H8-10376 shows the access permissions for the CTI registers in an Arm A-profile Debug implementation. For a definition of the terms used, see External debug interface registers on page H8-10368.

Table H8-6 Access permissions for the CTI registers

<table>
<thead>
<tr>
<th>Conditions (priority from left to right)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Offset</strong></td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>0x000</td>
</tr>
<tr>
<td>0x010</td>
</tr>
<tr>
<td>0x014</td>
</tr>
<tr>
<td>0x018</td>
</tr>
<tr>
<td>0x01C</td>
</tr>
<tr>
<td>0x020+4xn</td>
</tr>
<tr>
<td>0x0A0+4xn</td>
</tr>
<tr>
<td>0x130</td>
</tr>
<tr>
<td>0x134</td>
</tr>
<tr>
<td>0x138</td>
</tr>
<tr>
<td>0x13C</td>
</tr>
<tr>
<td>0x140</td>
</tr>
<tr>
<td>0xFC0</td>
</tr>
<tr>
<td>0xFC4</td>
</tr>
<tr>
<td>0xFC8</td>
</tr>
</tbody>
</table>
For the reset values of the CTI registers, see Table H8-8 on page H8-10380.
H8.8 External debug register resets

Each register or field has a defined reset domain:
- Registers and fields in the Warm reset domain are also reset by a Cold reset and unchanged by an External Debug reset that is not coincident with a Cold reset or a Warm reset.
- Registers and fields in the Cold reset domain are unchanged by a Warm reset or an External Debug reset that is not coincident with a Cold reset.
- Registers and fields in the External Debug reset domain are unchanged by a Cold reset or a Warm reset that is not coincident with an External Debug reset.

A reset might change the value of a register. Specific rules apply to the observability of registers in the External Debug reset domain by indirect reads from the Core power domain when an External Debug reset is asserted without a coincident Cold reset. For more information, see Synchronization of changes to the external debug registers on page H8-10358.

Table H8-2 on page H8-10368 and Table H8-5 on page H8-10375 show the external debug register and CTI register resets. For other debug registers, see Management register resets on page K6-11695.

Note

By reference to Figure H6-1 on page H6-10348 the power domain can be deduced from the reset domain. Table K6-9 on page K6-11695 also shows reset power domains.

Table H8-2 on page H8-10368 and Table H8-5 on page H8-10375 do not include:
- Read-only identification registers, such as Processor ID Registers and PMCFGR, that have a fixed value from reset.
- Read-only status registers, such as EDSCR.RW, that are evaluated each time the register is read and that have no meaningful reset value.
- Write-only registers, such as EDRCR, that only have an effect on writes, and have no meaningful reset value.
- Read/write registers, such as breakpoint and watchpoint registers, and EDPRCR.CORENPDRQ, that alias other registers. The reset values are described by the descriptions of those other registers.
- IMPLEMENTATION DEFINED registers. The reset values and reset domains of these registers are also IMPLEMENTATION DEFINED and might be UNKNOWN.

All other fields in the registers are set to an IMPLEMENTATION DEFINED value that can be UNKNOWN. The register is in the specified reset domain.

Note

An IMPLEMENTATION DEFINED reset value, which can be UNKNOWN, means that hardware is not required to reset the register on the specified reset, but software must not rely on the register being preserved over reset.

<table>
<thead>
<tr>
<th>Register</th>
<th>Reset domain</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBGPRCR_EL1</td>
<td>Cold into AArch64 state</td>
<td>CORENPDRQ</td>
<td>The value of the powerup request a</td>
<td>Debug Power Control Register.</td>
</tr>
<tr>
<td>DBGPRCR</td>
<td>Cold into AArch32 state</td>
<td>CORENPDRQ</td>
<td>The value of the powerup request a</td>
<td>Debug Power Control Register.</td>
</tr>
</tbody>
</table>
### Table H8-7 Summary of external debug register resets, debug registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Reset domain</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDESR</td>
<td>Warm</td>
<td>SS</td>
<td>EDECR.SS</td>
<td>Halting Step debug event pending</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RC</td>
<td>EDECR.RCE</td>
<td>Reset Catch debug event pending</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OSUC</td>
<td>0</td>
<td>OS Unlock Catch debug event pending</td>
</tr>
<tr>
<td>EDESR</td>
<td>Cold</td>
<td>SS</td>
<td>0</td>
<td>Halting Step debug event pending</td>
</tr>
<tr>
<td>if FEAT_DoPD is implemented</td>
<td>Warm</td>
<td>RC</td>
<td>CTDEVCTL.RCE</td>
<td>Reset Catch debug event pending</td>
</tr>
<tr>
<td>EDECR</td>
<td>Cold</td>
<td>SS</td>
<td>0</td>
<td>Halting Step debug event enable</td>
</tr>
<tr>
<td>if FEAT_DoPD is implemented</td>
<td>External debug</td>
<td>SS</td>
<td>0</td>
<td>Halting Step debug event enable</td>
</tr>
<tr>
<td>EDECR</td>
<td>Cold</td>
<td>RCE</td>
<td>0</td>
<td>Reset Catch debug event enable</td>
</tr>
<tr>
<td>if FEAT_DoPD is not implemented</td>
<td>OSUCE</td>
<td>0</td>
<td>OS Unlock Catch debug event enable</td>
<td></td>
</tr>
<tr>
<td>EDWAR</td>
<td>Cold</td>
<td>-</td>
<td>-</td>
<td>All fields</td>
</tr>
<tr>
<td>EDSR</td>
<td>Cold</td>
<td>RXfull</td>
<td>0</td>
<td>DTRRX register full</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TXfull</td>
<td>0</td>
<td>DTRTX register full</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RXO</td>
<td>0</td>
<td>DTRRX overrun</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TXU</td>
<td>0</td>
<td>DTRTX underrun</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTdis</td>
<td>0</td>
<td>Interrupt disable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TDA</td>
<td>0</td>
<td>Trap debug register accesses to Debug state</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MA</td>
<td>0</td>
<td>Memory access mode in Debug state</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HDE</td>
<td>0</td>
<td>Halting debug mode enable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ERR</td>
<td>0</td>
<td>Cumulative error flag</td>
</tr>
<tr>
<td>EDECCR</td>
<td>Cold</td>
<td>NSE[2:1]</td>
<td>0b00</td>
<td>Coarse-grained Non-secure Exception Catch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SE[3:1]</td>
<td>0b00</td>
<td>Coarse-grained Secure Exception Catch</td>
</tr>
<tr>
<td>EDPCSR</td>
<td>Cold</td>
<td>-</td>
<td>-</td>
<td>All fields</td>
</tr>
<tr>
<td>EDCIDSR</td>
<td>Cold</td>
<td>-</td>
<td>-</td>
<td>All fields</td>
</tr>
<tr>
<td>EDVIDSR</td>
<td>Cold</td>
<td>-</td>
<td>-</td>
<td>All fields</td>
</tr>
</tbody>
</table>
## Table H8-7  Summary of external debug register resets, debug registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Reset domain</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDPRCR</td>
<td>Cold</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>External debug</td>
<td>COREPURQ&lt;sup&gt;b&lt;/sup&gt;</td>
<td>-</td>
<td>Core powerup request</td>
</tr>
<tr>
<td>EDPRSR</td>
<td>Warm</td>
<td>SDR</td>
<td>-</td>
<td>Sticky debug restart</td>
</tr>
<tr>
<td></td>
<td>Cold</td>
<td>SPMAD</td>
<td>0</td>
<td>Sticky EPMAD error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SDAD</td>
<td>0</td>
<td>Sticky EDAD error</td>
</tr>
<tr>
<td></td>
<td>Warm</td>
<td>SR</td>
<td>1</td>
<td>Sticky reset status</td>
</tr>
<tr>
<td></td>
<td>Cold</td>
<td>SPD</td>
<td>1</td>
<td>Sticky powerdown status</td>
</tr>
</tbody>
</table>

a. If FEAT_DoPD is not implemented, the powerup request is the EDPRCR.COREPURQ control bit.

b. If FEAT_DoPD is not implemented, on a Cold reset into AArch64 state, DBGPRCR_EL1.CORENPDRQ resets to the value of EDPRCR.COREPURQ. On a Cold reset into AArch32 state, DBGPRCR.CORENPDRQ resets to the value of EDPRCR.COREPURQ. If an External Debug reset and a Cold reset coincide, both EDPRCR.COREPURQ and the CORENPDRQ field of the appropriate System register are reset to 0.

Table H8-5 on page H8-10375 shows the reset values for the CTI registers

## Table H8-8  Summary of external debug register resets, CTI registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Reset domain</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTICONTROL</td>
<td>External debug</td>
<td>GLBEN</td>
<td>0</td>
<td>CTI global enable</td>
</tr>
<tr>
<td>CTIDEVCTL</td>
<td>External debug</td>
<td>RCE</td>
<td>0</td>
<td>If FEAT_DoPD is implemented, Reset Catch debug event enable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OSUCE</td>
<td>0</td>
<td>If FEAT_DoPD is implemented, OS Unlock Catch debug event enable</td>
</tr>
<tr>
<td>CTIAPPSET</td>
<td>External debug</td>
<td>-</td>
<td>-</td>
<td>All fields</td>
</tr>
<tr>
<td>CTIINEN&lt;sub&gt;n&lt;/sub&gt;</td>
<td>External debug</td>
<td>-</td>
<td>-</td>
<td>All fields</td>
</tr>
<tr>
<td>CTIOUTEN&lt;sub&gt;n&lt;/sub&gt;</td>
<td>External debug</td>
<td>-</td>
<td>-</td>
<td>All fields</td>
</tr>
<tr>
<td>CTIGATE</td>
<td>External debug</td>
<td>-</td>
<td>-</td>
<td>All fields</td>
</tr>
<tr>
<td>ASICCTL</td>
<td>IMPLEMENTATION DEFINED</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED</td>
<td>All of register</td>
</tr>
</tbody>
</table>
Chapter H9
External Debug Register Descriptions

This chapter provides a description of the external debug registers. It contains the following sections:

• About the debug registers on page H9-10382.
• External debug registers on page H9-10383.
• Cross-Trigger Interface registers on page H9-10660.
H9.1 About the debug registers

The following sections describe the registers that are accessible through the external debug interface:

- External debug registers on page H9-10383.
- Cross-Trigger Interface registers on page H9-10660.
H9.2  External debug registers

This section describes the debug registers that are accessible through the external debug interface and are used for external debug.
H9.2.1 **DBGAUTHSTATUS_EL1, Debug Authentication Status register**

The DBGAUTHSTATUS_EL1 characteristics are:

**Purpose**

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

**Configurations**

External register DBGAUTHSTATUS_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGAUTHSTATUS_EL1[31:0].

External register DBGAUTHSTATUS_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGAUTHSTATUS[31:0].

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

**Attributes**

DBGAUTHSTATUS_EL1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:28</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>27:26</td>
<td>RTNID, Root non-invasive debug.</td>
</tr>
<tr>
<td>25:24</td>
<td>RTID, Root invasive debug.</td>
</tr>
<tr>
<td>15:14</td>
<td>RLNID, Realm non-invasive debug.</td>
</tr>
<tr>
<td>13:12</td>
<td>RLID, Realm invasive debug.</td>
</tr>
<tr>
<td>8:0</td>
<td>NSID</td>
</tr>
</tbody>
</table>

**Bits [31:28]**

Reserved, RES0.

**RTNID, bits [27:26]**

Root non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RTID.

**RTID, bits [25:24]**

Root invasive debug.

- **0b00**: Not implemented.
- **0b10**: Implemented and disabled.
  
  ```
  static bool ExternalRootInvasiveDebugEnabled() { return false; }
  ```

- **0b11**: Implemented and enabled.
  
  ```
  static bool ExternalRootInvasiveDebugEnabled() { return true; }
  ```

All other values are reserved.

If FEAT_RME is not implemented, the only permitted value is 00.

**Bits [23:16]**

Reserved, RES0.

**RLNID, bits [15:14]**

Realm non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RLID.

**RLID, bits [13:12]**

Realm invasive debug.

- **0b00**: Not implemented.
0b10  Implemented and disabled.
     ExternalRealmInvasiveDebugEnabled() == FALSE.
0b11  Implemented and enabled.
     ExternalRealmInvasiveDebugEnabled() == TRUE.

All other values are reserved.

If FEAT_RME is not implemented, the only permitted value is 00.

**Bits [11:8]**

Reserved, RES0.

**SNID, bits [7:6]**

- **When FEAT_Debugv8p4 is implemented:**
  - Secure non-invasive debug.
  - This field has the same value as DBGAUTHSTATUS_EL1.SID.
- **Otherwise:**
  - Secure non-invasive debug.
  
    - 0b00  Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 1.
    - 0b10  Implemented and disabled. ExternalSecureNoninvasiveDebugEnabled() == FALSE.
    - 0b11  Implemented and enabled. ExternalSecureNoninvasiveDebugEnabled() == TRUE.

All other values are reserved.

**SID, bits [5:4]**

- Secure invasive debug.
  
    - 0b00  Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 1.
    - 0b10  Implemented and disabled. ExternalSecureInvasiveDebugEnabled() == FALSE.
    - 0b11  Implemented and enabled. ExternalSecureInvasiveDebugEnabled() == TRUE.

All other values are reserved.

**NSNID, bits [3:2]**

- **When FEAT_Debugv8p4 is implemented:**
  - Non-secure non-invasive debug.
    
      - 0b00  Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 0.
      - 0b11  Implemented and enabled. ExternalNoninvasiveDebugEnabled() == TRUE.
  
     If the Effective value of SCR_EL3.NS is 1, or if EL3 is implemented and EL2 is not implemented, this field reads as 0b11.
  
     All other values are reserved.
- **Otherwise:**
  - Non-secure non-invasive debug.
    
      - 0b00  Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 0.
      - 0b10  Implemented and disabled. ExternalNoninvasiveDebugEnabled() == FALSE.
      - 0b11  Implemented and enabled. ExternalNoninvasiveDebugEnabled() == TRUE.

All other values are reserved.

**NSID, bits [1:0]**

- Non-secure invasive debug.
  
    - 0b00  Not implemented. EL3 is not implemented and the Effective value of SCR_EL3.NS is 0.
    - 0b10  Implemented and disabled. ExternalInvasiveDebugEnabled() == FALSE.
    - 0b11  Implemented and enabled. ExternalInvasiveDebugEnabled() == TRUE.
All other values are reserved.

**Accessing the DBGAUTHSTATUS_EL1:**

DBGAUTHSTATUS_EL1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFB8</td>
<td>DBGAUTHSTATUS_EL1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
H9.2.2 DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 15

The DBGBCR<n>_EL1 characteristics are:

**Purpose**

Holds control information for a breakpoint. Forms breakpoint n together with value register DBGBVR<n>_EL1.

**Configurations**

External register DBGBCR<n>_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGBCR<n>_EL1[31:0].

External register DBGBCR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGBCR<n>[31:0].

DBGBCR<n>_EL1 is in the Core power domain.

If breakpoint n is not implemented then accesses to this register are:

- RES0 when IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess().
- A CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR otherwise.

**Attributes**

DBGBCR<n>_EL1 is a 32-bit register.

**Field descriptions**

| 31 | 28 | 23 | 20 | 19 | 16 | 15 | 14 | 13 | 12 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| RES0 | BT | LBN | SSC | RES0 | BAS | RES0 | PMC | E |

When the E field is zero, all the other fields in the register are ignored.

**Bits [31:24]**

Reserved, RES0.

**BT, bits [23:20]**

Breakpoint Type. Possible values are:

- **0b0000**: Unlinked instruction address match. DBGBVR<n>_EL1 is the address of an instruction.
- **0b0001**: As 0b0000 but linked to a Context matching breakpoint.
- **0b0010**: Unlinked Context ID match. When FEAT_VHE is implemented, EL2 is using AArch64, and the Effective value of HCR_EL2.E2H is 1, if either the PE is executing at EL0 with HCR_EL2.TGE set to 1 or the PE is executing at EL2, then DBGBVR<n>_EL1.ContextID must match the CONTEXTIDR_EL2 value. Otherwise, DBGBVR<n>_EL1.ContextID must match the CONTEXTIDR_EL1 value.
- **0b0011**: As 0b0010, with linking enabled.
- **0b0100**: Unlinked instruction address mismatch. DBGBVR<n>_EL1 is the address of an instruction to be stepped.
- **0b0101**: As 0b0100, but linked to a Context matching breakpoint.
- **0b0110**: Unlinked CONTEXTIDR_EL1 match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1.
- **0b0111**: As 0b0110, with linking enabled.
- **0b1000**: Unlinked VMID match. DBGBVR<n>_EL1.VMID is a VMID compared against VTTBR_EL2.VMID.
**LBN, bits [19:16]**

Linked breakpoint number. For Linked address matching breakpoints, this specifies the index of the Context-matching breakpoint linked to.

For all other breakpoint types this field is ignored and reads of the register return an **UNKNOWN** value.

This field is ignored when the value of DBGBCR<n>_EL1.E is 0.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally **UNKNOWN** value.

**SSC, bits [15:14]**

Security state control. Determines the Security states under which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the HMC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information, including the effect of programming the fields to a reserved set of values, see **Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values** on page D2-4722.

For more information on the operation of the SSC, HMC, and PMC fields, see **Execution conditions for which a breakpoint generates Breakpoint exceptions** on page D2-4717.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally **UNKNOWN** value.

**HMC, bit [13]**

Higher mode control. Determines the debug perspective for deciding when a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see **DBGBCR<n>_EL1.SSC description**.

For more information on the operation of the SSC, HMC, and PMC fields, see **Execution conditions for which a breakpoint generates Breakpoint exceptions** on page D2-4717.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally **UNKNOWN** value.

**Bits [12:9]**

Reserved, **RES0**.
BAS, bits [8:5]

When AArch32 is supported:
Byte address select. Defines which half-words an address-matching breakpoint matches, regardless of the instruction set and Execution state.
The permitted values depend on the breakpoint type.
For Address match breakpoints in either AArch32 or AArch64 state, the permitted values are:

<table>
<thead>
<tr>
<th>BAS</th>
<th>Match instruction at</th>
<th>Constraint for debuggers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0011</td>
<td>DBGBVR&lt;(n)&gt;_EL1</td>
<td>Use for T32 instructions</td>
</tr>
<tr>
<td>0b1100</td>
<td>DBGBVR&lt;(n)&gt;_EL1 + 2</td>
<td>Use for T32 instructions</td>
</tr>
<tr>
<td>0b1111</td>
<td>DBGBVR&lt;(n)&gt;_EL1</td>
<td>Use for A64 and A32 instructions</td>
</tr>
</tbody>
</table>

All other values are reserved.
For more information, see Using the BAS field in Address Match breakpoints on page G2-9067.
For Address mismatch breakpoints in an AArch32 stage 1 translation regime, the permitted values are:

<table>
<thead>
<tr>
<th>BAS</th>
<th>Match instruction at</th>
<th>Constraint for debuggers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>-</td>
<td>Use for a match anywhere breakpoint</td>
</tr>
<tr>
<td>0b0011</td>
<td>DBGBVR&lt;(n)&gt;_EL1</td>
<td>Use for stepping T32 instructions</td>
</tr>
<tr>
<td>0b1100</td>
<td>DBGBVR&lt;(n)&gt;_EL1 + 2</td>
<td>Use for stepping T32 instructions</td>
</tr>
<tr>
<td>0b1111</td>
<td>DBGBVR&lt;(n)&gt;_EL1</td>
<td>Use for stepping A64 and A32 instructions</td>
</tr>
</tbody>
</table>

For more information, see Using the BAS field in Address Match breakpoints on page G2-9067.
For Context matching breakpoints, this field is RES1 and ignored.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES1.

Bits [4:3]
Reserved, RES0.

PMC, bits [2:1]
Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the \{HMC, SSC, PMC\} fields. For more information see the DBGBCR<\(n\)>_EL1.SSC description.

For more information on the operation of the SSC, HMC, and PMC fields, see Execution conditions for which a breakpoint generates Breakpoint exceptions on page D2-4717.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]
Enable breakpoint DBGBVR<\(n\)>_EL1. Possible values are:
0b0 Breakpoint disabled.
0b1 Breakpoint enabled.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally **UNKNOWN** value.

**Accessing the DBGBCR<n>_EL1:**

_____ Note _______
SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new
definition from Armv8.4. Refer to the Pseudocode definitions for more information.

DBGBCR<n>_EL1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x408 + (16 * n)</td>
<td>DBGBCR&lt;n&gt;_EL1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

• When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and SoftwareLockStatus() accesses to this register are RO.
• When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and !SoftwareLockStatus() accesses to this register are RW.
• Otherwise accesses to this register generate an error response.
H9.2.3 DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15

The DBGBVR<n>_EL1 characteristics are:

**Purpose**

Holds a virtual address, or a VMID and/or a context ID, for use in breakpoint matching. Forms breakpoint n together with control register DBGBCR<n>_EL1.

**Configurations**

External register DBGBVR<n>_EL1 bits [63:0] are architecturally mapped to AArch64 System register DBGBVR<n>_EL1[63:0].

External register DBGBVR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGBVR<n>[31:0].

If the breakpoint is context-aware and EL2 is implemented then External register DBGBVR<n>_EL1[63:32] is architecturally mapped to AArch32 System register DBGBXVR<n>. Otherwise, there is no External register access to DBGBVR<n>_EL1[63:32] from AArch32 state.

External register DBGBVR<n>_EL1 bits [63:32] are architecturally mapped to AArch32 System register DBGBXVR<n>[31:0].

DBGBVR<n>_EL1 is in the Core power domain.

How this register is interpreted depends on the value of DBGBCR<n>_EL1.BT.

- When DBGBCR<n>_EL1.BT is 0b000x, this register holds a virtual address.
- When DBGBCR<n>_EL1.BT is 0b001x, 0b011x, or 0b110x, this register holds a Context ID.
- When DBGBCR<n>_EL1.BT is 0b100x, this register holds a VMID.
- When DBGBCR<n>_EL1.BT is 0b101x, this register holds a VMID and a Context ID.
- When DBGBCR<n>_EL1.BT is 0b111x, this register holds two Context ID values.

For other values of DBGBCR<n>_EL1.BT, this register is RES0.

If breakpoint n is not implemented then accesses to this register are:

- RES0 when IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess().
- A CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR otherwise.

**Attributes**

DBGBVR<n>_EL1 is a 64-bit register.

**Field descriptions**

*When DBGBCR<n>_EL1.BT == 0b000x:*

<table>
<thead>
<tr>
<th>63</th>
<th>53</th>
<th>49</th>
<th>48</th>
<th>32</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits [52:49]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA[48:2]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
</tr>
</tbody>
</table>

**RESS[14:4], bits [63:53]**

Reserved, Sign extended. Software must treat this field as RES0 if the most significant bit of VA is 0 or RES0, and as RES1 if the most significant bit of VA is 1.

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of the most significant bit of VA, meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.
The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

VA[52:49], bits [52:49]

When FEAT_LVA is implemented:

VA[52:49]

Extension to VA[48:2]. For more information, see VA[48:2].

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RESS[3:0]

Extension to RESS[14:4]. For more information, see RESS[14:4].

VA[48:2], bits [48:2]

If the address is being matched in an AArch64 stage 1 translation regime:

• This field contains bits[48:2] of the address for comparison.

• When FEAT_LVA is implemented, VA[52:49] forms the upper part of the address value. Otherwise, VA[52:49] are RESS.

If the address is being matched in an AArch32 stage 1 translation regime, the first 20 bits of this field are RES0, and the rest of the field contains bits[31:2] of the address for comparison.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT == 0b001x:

Bits [63:32]

Reserved, RES0.

ContextID, bits [31:0]

Context ID value for comparison.

The value is compared against CONTEXTIDR_EL2 when (FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented), EL2 is using AArch64, HCR_EL2.E2H is 1, and either:

• The PE is executing at EL2.

• HCR_EL2.TGE is 1, the PE is executing at EL0, and EL2 is enabled in the current Security state.

Otherwise, the value is compared against the following:

• CONTEXTIDR when the PE is executing at AArch32.

• CONTEXTIDR_EL1 when the PE is executing at AArch64.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.
When DBGBCR<\(n\)_EL1.BT == 0b011x, EL2 is implemented and (FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented):

Bits [63:32]  
Reserved, RES0.

ContextID, bits [31:0]  
Context ID value for comparison against CONTEXTIDR_EL1.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<\(n\)_EL1.BT == 0b100x and EL2 is implemented:

Bits [63:48]  
Reserved, RES0.

VMID[15:8], bits [47:40]  
When FEAT_VHE is implemented and VTCR_EL2.VS == 1:  
Extension to VMID[7:0]. For more information, see DBGBVR<\(n\)_EL1.VMID[7:0].
If EL2 is using AArch32, this field is RES0.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.
Otherwise:
Reserved, RES0.

VMID[7:0], bits [39:32]  
VMID value for comparison.
The VMID is 8 bits when any of the following are true:
• EL2 is using AArch32.
• VTCR_EL2.VS is 0.
• FEAT_VMID16 is not implemented.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]  
Reserved, RES0.
When DBGBCR<{}>_EL1.BT == 0b101x and EL2 is implemented:

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-48</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>47-40</td>
<td>VMID[15:8]</td>
</tr>
<tr>
<td>39-32</td>
<td>VMID[7:0]</td>
</tr>
<tr>
<td>31-0</td>
<td>ContextID</td>
</tr>
</tbody>
</table>

Bits [63:48]

Reserved, RES0.

VMID[15:8], bits [47:40]

When FEAT_VMID16 is implemented and VTCR_EL2.VS == 1:

Extension to VMID[7:0]. For more information, see DBGBCR<{}>_EL1.VMID[7:0].
If EL2 is using AArch32, or if the implementation has an 8-bit VMID, this field is RES0.
The reset behavior of this field is:
  • On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID[7:0], bits [39:32]

VMID value for comparison.
The VMID is 8 bits when any of the following are true:
• EL2 is using AArch32.
• VTCR_EL2.VS is 0.
• FEAT_VMID16 is not implemented.
The reset behavior of this field is:
  • On a Cold reset, this field resets to an architecturally UNKNOWN value.

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.
The reset behavior of this field is:
  • On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<{}>_EL1.BT == 0b110x, EL2 is implemented and (FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented):

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>ContextID2</td>
</tr>
<tr>
<td>31-0</td>
<td>RES0</td>
</tr>
</tbody>
</table>

ContextID2, bits [63:32]

Context ID value for comparison against CONTEXTIDR_EL2.
The reset behavior of this field is:
  • On a Cold reset, this field resets to an architecturally UNKNOWN value.
Bits [31:0]

Reserved, RES0.

*When DBGBCRₙ_EL1.BT == 0b111x, EL2 is implemented and (FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented)*:

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContextID2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContextID</td>
<td></td>
</tr>
</tbody>
</table>

**ContextID2, bits [63:32]**

Context ID value for comparison against CONTEXTIDR_EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**ContextID, bits [31:0]**

Context ID value for comparison against CONTEXTIDR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

### Accessing the DBGBVRₙ_EL1:

**Note**

SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

DBGBVRₙ_EL1[63:0] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x400 + (16 * n)</td>
<td>DBGBVRₙ_EL1</td>
<td>63:0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.
H9.2.4  DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register

The DBGCLAIMCLR_EL1 characteristics are:

Purpose

Used by software to read the values of the CLAIM tag bits, and to clear CLAIM tag bits to 0. The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software. Used in conjunction with the DBGCLAIMSET_EL1 register.

Configurations

External register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGCLAIMCLR_EL1[31:0].

External register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGCLAIMCLR[31:0].

DBGCLAIMCLR_EL1 is in the Core power domain.

An implementation must include eight CLAIM tag bits.

Attributes

DBGCLAIMCLR_EL1 is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAZ/WI</td>
<td>CLAIM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits [31:8]

Reserved, RAZ/WI.

CLAIM, bits [7:0]

Read or clear CLAIM tag bits. Reading this field returns the current value of the CLAIM tag bits.

Writing a 1 to one of these bits clears the corresponding CLAIM tag bit to 0. This is an indirect write to the CLAIM tag bits. A single write operation can clear multiple CLAIM tag bits to 0.

Writing 0 to one of these bits has no effect.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.

Accessing the DBGCLAIMCLR_EL1:

DBGCLAIMCLR_EL1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFA4</td>
<td>DBGCLAIMCLR_EL1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
• When `IsCorePowered()`, `!DoubleLockStatus()`, `!OSLockStatus()` and `!SoftwareLockStatus()` accesses to this register are RW.
• Otherwise accesses to this register generate an error response.
H9.2.5  DBGCLAIMSET_EL1, Debug CLAIM Tag Set register

The DBGCLAIMSET_EL1 characteristics are:

**Purpose**

Used by software to set the CLAIM tag bits to 1.

The architecture does not define any functionality for the CLAIM tag bits.

--- **Note** ---

CLAIM tags are typically used for communication between the debugger and target software.

---

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMCLR_EL1 register.

**Configurations**

External register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGCLAIMSET_EL1[31:0].

External register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGCLAIMSET[31:0].

DBGCLAIMSET_EL1 is in the Core power domain.

An implementation must include eight CLAIM tag bits.

**Attributes**

DBGCLAIMSET_EL1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RAZ/WI</td>
</tr>
<tr>
<td>8</td>
<td>Claim, bits [7:0]</td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RAZ/WI.

**CLAIM, bits [7:0]**

Set CLAIM tag bits.

This field is RAO.

Writing a 1 to one of these bits sets the corresponding CLAIM tag bit to 1. This is an indirect write to the CLAIM tag bits. A single write operation can set multiple CLAIM tag bits to 1.

Writing 0 to one of these bits has no effect.

**Accessing the DBGCLAIMSET_EL1:**

DBGCLAIMSET_EL1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFA0</td>
<td>DBGCLAIMSET_EL1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
• When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.

• Otherwise accesses to this register generate an error response.
H9.2.6  DBGDTRRX_EL0, Debug Data Transfer Register, Receive

The DBGDTRRX_EL0 characteristics are:

**Purpose**

Transfers data from an external debugger to the PE. For example, it is used by a debugger transferring commands and data to a debug target. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communications Channel.

**Configurations**

External register DBGDTRRX_EL0 bits [31:0] are architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0].

External register DBGDTRRX_EL0 bits [31:0] are architecturally mapped to AArch32 System register DBGDTRRXint[31:0].

DBGDTRRX_EL0 is in the Core power domain.

**Attributes**

DBGDTRRX_EL0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th>Update DTRRX</th>
</tr>
</thead>
</table>

**Bits [31:0]**

Update DTRRX.

Writes to this register:
- If RXfull is set to 1, set DTRRX to UNKNOWN.
- If RXfull is set to 0, update the value in DTRRX.

After the write, RXfull is set to 1.

Reads of this register:
- If RXfull is set to 1, return the last value written to DTRRX.
- If RXfull is set to 0, return an UNKNOWN value.

After the read, RXfull remains unchanged.

For the full behavior of the Debug Communications Channel, see Chapter H4 The Debug Communication Channel and Instruction Transfer Register.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Accessing the DBGDTRRX_EL0:**

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any operation issued by a DTR access in memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.
DBGDTRRX_EL0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x80</td>
<td>DBGDTRRX_EL0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.
H9.2.7 DBGDTTX_EL0, Debug Data Transfer Register, Transmit

The DBGDTTX_EL0 characteristics are:

**Purpose**

Transfers data from the PE to an external debugger. For example, it is used by a debug target to transfer data to the debugger. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communication Channel.

**Configurations**

External register DBGDTTX_EL0 bits [31:0] are architecturally mapped to AArch64 System register DBGDTRTX_EL0[31:0].

External register DBGDTTX_EL0 bits [31:0] are architecturally mapped to AArch32 System register DBGDTRTXint[31:0].

DBGDTTX_EL0 is in the Core power domain.

**Attributes**

DBGDTTX_EL0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit [31:0]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>Return DTRTX</td>
</tr>
</tbody>
</table>

- **Bits [31:0]**
  - Return DTRTX.
  - Reads of this register:
    - If TXfull is set to 1, return the last value written to DTRTX.
    - If TXfull is set to 0, return an UNKNOWN value.
  - After the read, TXfull is cleared to 0.
  - Writes to this register:
    - If TXfull is set to 1, set DTRTX to UNKNOWN.
    - If TXfull is set to 0, update the value in DTRTX.
  - After the write, TXfull remains unchanged.

- For the full behavior of the Debug Communications Channel, see Chapter H4 The Debug Communication Channel and Instruction Transfer Register.

- The reset behavior of this field is:
  - On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Accessing the DBGDTTX_EL0:**

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any operation issued by a DTR access in memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.
DBGDTRTX_EL0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x08C</td>
<td>DBGDTRTX_EL0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.
H9.2.8  **DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 15**

The DBGWCR<n>_EL1 characteristics are:

**Purpose**

Holds control information for a watchpoint. Forms watchpoint n together with value register DBGWVR<n>_EL1.

**Configurations**

External register DBGWCR<n>_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGWCR<n>_EL1[31:0].

External register DBGWCR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGWCR<n>[31:0].

DBGWCR<n>_EL1 is in the Core power domain.

If watchpoint n is not implemented then accesses to this register are:

- When IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess(), RES0.
- Otherwise, a CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR.

**Attributes**

DBGWCR<n>_EL1 is a 32-bit register.

**Field descriptions**

When the E field is zero, all the other fields in the register are ignored.

**Bits [31:29]**

Reserved, RES0.

**MASK, bits [28:24]**

Address mask. Only objects up to 2GB can be watched using a single mask.

- 0b00000  No mask.
- 0b00001  Reserved.
- 0b00010  Reserved.

If programmed with a reserved value, a watchpoint must behave as if either:

- MASK has been programmed with a defined value, which might be 0 (no mask), other than for a direct read of DBGWCRn_EL1.
- The watchpoint is disabled.

Software must not rely on this property because the behavior of reserved values might change in a future revision of the architecture.

Other values mask the corresponding number of address bits, from 0b0000011 masking 3 address bits (0x00000007 mask for address) to 0b1111111 masking 31 address bits (0xFFFFFFFF mask for address).

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Bits [23:21]**

Reserved, RES0.
WT, bit [20]
Watchpoint type. Possible values are:
0b0 Unlinked data address match.
0b1 Linked data address match.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]
Linked breakpoint number. For Linked data address watchpoints, this specifies the index of the Context-matching breakpoint linked to.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]
Security state control. Determines the Security states under which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the HMC and PAC fields.
For more information on the operation of the SSC, HMC, and PAC fields, see Execution conditions for which a watchpoint generates Watchpoint exceptions on page D2-4728.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]
Higher mode control. Determines the debug perspective for deciding when a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and PAC fields.
For more information on the operation of the SSC, HMC, and PAC fields, see Execution conditions for which a watchpoint generates Watchpoint exceptions on page D2-4728.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

BAS, bits [12:5]
Byte address select. Each bit of this field selects whether a byte from within the word or double-word addressed by DBGWVR<n>_EL1 is being watched.

<table>
<thead>
<tr>
<th>BAS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxxxxx1</td>
<td>Match byte at DBGWVR&lt;n&gt;_EL1</td>
</tr>
<tr>
<td>xxxxxx1x</td>
<td>Match byte at DBGWVR&lt;n&gt;_EL1 + 1</td>
</tr>
<tr>
<td>xxxxx1xx</td>
<td>Match byte at DBGWVR&lt;n&gt;_EL1 + 2</td>
</tr>
<tr>
<td>xxxx1xxx</td>
<td>Match byte at DBGWVR&lt;n&gt;_EL1 + 3</td>
</tr>
</tbody>
</table>

In cases where DBGWVR<n>_EL1 addresses a double-word:

<table>
<thead>
<tr>
<th>BAS</th>
<th>Description, if DBGWVR&lt;n&gt;_EL1[2] == 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx1xxxx</td>
<td>Match byte at DBGWVR&lt;n&gt;_EL1 + 4</td>
</tr>
</tbody>
</table>

The valid values for BAS are non-zero binary number all of whose set bits are contiguous. All other values are reserved and must not be used by software. See Reserved DBGWCR<\textit{n}>, BAS values on page G2-9089.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

**LSC, bits [4:3]**

Load/store control. This field enables watchpoint matching on the type of access being made. Possible values of this field are:
0b01 Match instructions that load from a watchpointed address.
0b10 Match instructions that store to a watchpointed address.
0b11 Match instructions that load from or store to a watchpointed address.

All other values are reserved, but must behave as if the watchpoint is disabled. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

**PAC, bits [2:1]**

Privilege of access control. Determines the Exception level or levels at which a Watchpoint debug event for watchpoint \textit{n} is generated. This field must be interpreted along with the SSC and HMC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see Execution conditions for which a watchpoint generates Watchpoint exceptions on page D2-4728.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

**E, bit [0]**

Enable watchpoint \textit{n}. Possible values are:
0b0 Watchpoint disabled.
0b1 Watchpoint enabled.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Accessing the DBGWCR<\textit{n}> EL1:**

--- Note ---
SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.
DBGWCRₙ_EL1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x888 + (16 * n)</td>
<td>DBGWCRₙ_EL1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.
H9.2.9 **DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 15**

The DBGWVR<n>_EL1 characteristics are:

**Purpose**

Holds a data address value for use in watchpoint matching. Forms watchpoint n together with control register DBGWCR<n>_EL1.

**Configurations**

External register DBGWVR<n>_EL1 bits [63:0] are architecturally mapped to AArch64 System register DBGWVR<n>_EL1[63:0].

External register DBGWVR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGWVR<n>[31:0].

DBGWVR<n>_EL1 is in the Core power domain.

If watchpoint n is not implemented then accesses to this register are:

- When IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess(), RES0.
- Otherwise, a CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR.

**Attributes**

DBGWVR<n>_EL1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RESS[14:4], bits [63:53]**

Reserved, Sign extended. Hardware and software must treat this field as RES0 if the most significant bit of VA is 0 or RES0, and as RES1 if the most significant bit of VA is 1.

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of the most significant bit of VA, meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.
- The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

**VA[52:49], bits [52:49]**

*When FEAT_LVA is implemented:*

VA[52:49]

Extension to VA[48:2]. For more information, see VA[48:2].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

RESS[3:0]

Extension to RESS[14:4]. For more information, see RESS[14:4].
VA[48:2], bits [48:2]

Bits[48:2] of the address value for comparison.

When FEAT_LVA is implemented, VA[52:49] forms the upper part of the address value. Otherwise, VA[52:49] are RESV.


The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

Accessing the DBGWVR<n>_EL1:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

DBGWVR<n>_EL1[63:0] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x800 + (16 * n)</td>
<td>DBGWVR&lt;n&gt;_EL1</td>
<td>63:0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

• When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and SoftwareLockStatus() accesses to this register are RO.

• When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and !SoftwareLockStatus() accesses to this register are RW.

• Otherwise accesses to this register generate an error response.
H9.2.10 EDAA32PFR, External Debug Auxiliary Processor Feature Register

The EDAA32PFR characteristics are:

**Purpose**

Provides information about implemented PE features.

--- **Note** ---

The register mnemonic, EDAA32PFR, is derived from previous definitions of this register that defined this register only when AArch64 was not supported.

For general information about the interpretation of the ID registers, see *Principles of the ID scheme for fields in ID registers* on page D17-5551.

**Configurations**

It is IMPLEMENTATION DEFINED whether EDAA32PFR is implemented in the Core power domain or in the Debug power domain.

**Attributes**

EDAA32PFR is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:20]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[19:16]</td>
<td>MSA_frac</td>
</tr>
<tr>
<td>[15:12]</td>
<td>EL3</td>
</tr>
<tr>
<td>[11:8]</td>
<td>EL2</td>
</tr>
<tr>
<td>[7:4]</td>
<td>PMSA</td>
</tr>
<tr>
<td>[3:0]</td>
<td>VMSA</td>
</tr>
</tbody>
</table>

**Bits [63:20]**

Reserved, RES0.

**MSA_frac, bits [19:16]**

When **EDAA32PFR.PMSA == 0b0000 and EDAA32PFR.VMSA == 0b1111:**

Memory System Architecture fractional field. This holds the information on additional Memory System Architectures supported. Defined values are:

- 0b0001: PMSAv8-64 supported in all translation regimes. VMSAv8-64 not supported.
- 0b0010: PMSAv8-64 supported in all translation regimes. In addition to PMSAv8-64, stage 1 EL1&0 translation regime also supports VMSAv8-64.

All other values are reserved.

**Otherwise:**

Reserved, RES0.

**EL3, bits [15:12]**

When **EDPFR.EL3 == 0b0000:**

AArch32 EL3 Exception level handling. Defined values are:

- 0b0000: EL3 is not implemented or can be executed in AArch64 state.
- 0b0001: EL3 can be executed in AArch32 state only.

All other values are reserved.
EDPF.R.{EL1, EL0} indicate whether EL1 and EL0 can only be executed in AArch32 state.

Otherwise:
Reserved, RAZ.

EL2, bits [11:8]

When EDPFR.EL2 == 0b0000:
AArch32 EL2 Exception level handling. Defined values are:
0b0000   EL2 is not implemented or can be executed in AArch64 state.
0b0001   EL2 can be executed in AArch32 state only.
All other values are reserved.

Note: EDPFR.EL1, EL0 indicate whether EL1 and EL0 can only be executed in AArch32 state.

Otherwise:
Reserved, RAZ.

PMSA, bits [7:4]
Indicates support for a 32-bit PMSA. Defined values are:
0b0000   PMSA-32 not supported.
0b0100   PMSAv8-32 supported.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

VMSA, bits [3:0]

When EDAA32PFR.PMSA != 0b0000:
Indicates support for a VMSA in addition to a 32-bit PMSA. Defined values are:
0b0000   VMSA not supported.
All other values are reserved.

When EDAA32PFR.PMSA == 0b0000:
Defined values are:
0b0000   VMSAv8-64 supported.
0b1111   Memory system architecture described by EDAA32PFR.MSA_frac.
All other values are reserved.
In Armv8-A, the only permitted value is 0b0000.

Otherwise:
Reserved, RAZ.

Accessing the EDAA32PFR:
EDAA32PFR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x060</td>
<td>EDAA32PFR</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When `IsCorePowered()` and `!DoubleLockStatus()` accesses to this register are RO.
- Otherwise accesses to this register are IMPDEF.
H9.2.11 EDACR, External Debug Auxiliary Control Register

The EDACR characteristics are:

**Purpose**

Allows implementations to support IMPLEMENTATION DEFINED controls.

**Configurations**

It is IMPLEMENTATION DEFINED whether EDACR is implemented in the Core power domain or in the Debug power domain.

Implementation of this register is OPTIONAL.

This register is only implemented if DBGDEVID.AuxRegs == 0b0001.

If FEAT_DoPD is implemented, this register is implemented in the Core power domain.

If FEAT_DoPD is not implemented, the power domain that this register is implemented in is IMPLEMENTATION DEFINED.

Changing this register from its reset value causes IMPLEMENTATION DEFINED behavior, including possible deviation from the architecturally-defined behavior.

If the EDACR contains any control bits that must be preserved over power down, then these bits must be accessible by the external debug interface when the OS Lock is locked, OSLSR_EL1.OSLK == 1, and when the Core is powered off.

**Attributes**

EDACR is a 32-bit register.

**Field descriptions**

| 31 | IMPLEMENTATION DEFINED | 0 |

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- The following resets apply:
  - If the register is implemented in the Core power domain:
    - On a Cold reset, this field resets to an architecturally UNKNOWN value.
    - On an External debug reset, the value of this field is unchanged.
    - On a Warm reset, the value of this field is unchanged.
  - If the register is implemented in the External debug power domain:
    - On a Cold reset, the value of this field is unchanged.
    - On an External debug reset, this field resets to an architecturally UNKNOWN value.
    - On a Warm reset, the value of this field is unchanged.
Accessing the EDACR:

EDACR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x094</td>
<td>EDACR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register are IMPDEF.
H9.2.12 EDCIDR0, External Debug Component Identification Register 0

The EDCIDR0 characteristics are:

**Purpose**

Provides information to identify an external debug component.

For more information, see *About the Component Identification scheme on page K6-11699*.

**Configurations**

Implementation of this register is OPTIONAL.

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

**Attributes**

EDCIDR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-8</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>7-0</td>
<td>PRMBL_0, bits [7:0]</td>
</tr>
</tbody>
</table>

**PRMBL_0, bits [7:0]**

Preamble.

Reads as 0x0D.

Access to this field is RO.

**Accessing the EDCIDR0:**

EDCIDR0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFFF0</td>
<td>EDCIDR0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
H9.2.13 EDCIDR1, External Debug Component Identification Register 1

The EDCIDR1 characteristics are:

**Purpose**

Provides information to identify an external debug component.

For more information, see *About the Component Identification scheme on page K6-11699.*

**Configurations**

Implementation of this register is OPTIONAL.

If `FEAT_DoPD` is implemented, this register is in the Core power domain. If `FEAT_DoPD` is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

**Attributes**

EDCIDR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-8</td>
<td>RES0</td>
</tr>
<tr>
<td>7-4</td>
<td>CLASS</td>
</tr>
<tr>
<td>3-0</td>
<td>PRMBL_1</td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**CLASS, bits [7:4]**

Component class.

0b1001 CoreSight component.

Other values are defined by the CoreSight Architecture.

This field reads as 0x9.

**PRMBL_1, bits [3:0]**

Preamble.

Reads as 0b0000.

Access to this field is RO.

**Accessing the EDCIDR1:**

EDCIDR1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFF4</td>
<td>EDCIDR1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When `FEAT_DoPD` is not implemented or `IsCorePowered()` accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
H9.2.14   EDCIDR2, External Debug Component Identification Register 2

The EDCIDR2 characteristics are:

**Purpose**

Provides information to identify an external debug component.

For more information, see *About the Component Identification scheme* on page K6-11699.

**Configurations**

Implementation of this register is OPTIONAL.

If `FEAT_DoPD` is implemented, this register is in the Core power domain. If `FEAT_DoPD` is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

**Attributes**

EDCIDR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>7</th>
<th>0</th>
<th>RES0</th>
<th>PRMBL_2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 0 0 0 0 1 0 1</td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**PRMBL_2, bits [7:0]**

Preamble.
Reads as 0x05.
Access to this field is RO.

**Accessing the EDCIDR2:**

EDCIDR2 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFF8</td>
<td>EDCIDR2</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When `FEAT_DoPD` is not implemented or `IsCorePowered()` accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
H9.2.15 EDCIDR3, External Debug Component Identification Register 3

The EDCIDR3 characteristics are:

**Purpose**

Provides information to identify an external debug component.

For more information, see *About the Component Identification scheme* on page K6-11699.

**Configurations**

Implementation of this register is OPTIONAL.

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

**Attributes**

EDCIDR3 is a 32-bit register.

**Field descriptions**

```
+-----+-----+-----+-----+-----+-----+-----+
| 31  | 30  | 29  | 28  | 27  | 26  | 25  |
|-----+-----+-----+-----+-----+-----+-----|
| RES0|     |     |     |     |     |     |
| 0 1 0 1 1 0 0 1|
|   | PRMBL_3|
```

**Bits [31:8]**

Reserved, RES0.

**PRMBL_3, bits [7:0]**

Preamble.

Reads as 0xB1.

Access to this field is RO.

**Accessing the EDCIDR3:**

EDCIDR3 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFFC</td>
<td>EDCIDR3</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
H9.2.16 EDCIDSR, External Debug Context ID Sample Register

The EDCIDSR characteristics are:

**Purpose**
Contains the sampled value of the Context ID, captured on reading EDPCSR[31:0].

**Configurations**
EDCIDSR is in the Core power domain.
This register is present only when FEAT_PCSRv8 is implemented and FEAT_PCSRv8p2 is not implemented. Otherwise, direct accesses to EDCIDSR are RES0.
Implemented only if the OPTIONAL PC Sample-based Profiling Extension is implemented in the external debug registers space.

--- Note ---
FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors registers space.

**Attributes**
EDCIDSR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>CONTEXTIDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**CONTEXTIDR, bits [31:0]**
Context ID. The value of CONTEXTIDR that is associated with the most recent EDPCSR sample. When the most recent EDPCSR sample is generated:

- If EL1 is using AArch64, then the Context ID is sampled from CONTEXTIDR_EL1.
- If EL1 is using AArch32, then the Context ID is sampled from CONTEXTIDR.
- If EL3 is implemented and is using AArch32, then CONTEXTIDR is a banked register, and EDCIDSR samples the current banked copy of CONTEXTIDR for the Security state that is associated with the most recent EDPCSR sample.

Because the value written to EDCIDSR is an indirect read of CONTEXTIDR, it is constrained unpredictable whether EDCIDSR is set to the original or new value if EDPCSR samples:

- An instruction that writes to CONTEXTIDR.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Accessing the EDCIDSR:**
IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see *Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN* on page H7-10354.
EDCIDSR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x0A4</td>
<td>EDCIDS</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
H9.2.17 EDDEVAFF0, External Debug Device Affinity register 0

The EDDEVAFF0 characteristics are:

**Purpose**

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the external debug component relates to.

**Configurations**

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

**Attributes**

EDDEVAFF0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIDR_EL1lo</td>
<td>MPIDR_EL1 low half. Read-only copy of the low half of MPIDR_EL1, as seen from the highest implemented Exception level.</td>
</tr>
</tbody>
</table>

**Accessing the EDDEVAFF0:**

EDDEVAFF0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xF8</td>
<td>EDDEVAFF0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
H9.2.18 EDDEVAFF1, External Debug Device Affinity register 1

The EDDEVAFF1 characteristics are:

**Purpose**

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the external debug component relates to.

**Configurations**

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

**Attributes**

EDDEVAFF1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIDR_EL1hi</td>
<td>31 0  MPIDR_EL1 high half. Read-only copy of the high half of MPIDR_EL1, as seen from the highest implemented Exception level.</td>
</tr>
</tbody>
</table>

**Accessing the EDDEVAFF1:**

EDDEVAFF1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFAC</td>
<td>EDDEVAFF1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
H9.2.19   EDDEVARCH, External Debug Device Architecture register

The EDDEVARCH characteristics are:

Purpose

Identifies the programmers’ model architecture of the external debug component.

Configurations

Implementation of this register is OPTIONAL.

If FEAT_DoPD is implemented, this register is in the Core power domain.

If FEAT_DoPD is not implemented, this register is in the Debug power domain.

Attributes

EDDEVARCH is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARCHITECT</td>
<td>[31:21]</td>
<td>Defines the architecture of the component. For debug, this is Arm Limited.</td>
</tr>
<tr>
<td>REVISION</td>
<td>[19:16]</td>
<td>Defines the architecture revision. For architectures defined by Arm this is the minor revision. For debug, the revision defined by Armv8 is 0x0. All other values are reserved.</td>
</tr>
<tr>
<td>ARCHVER</td>
<td>[15:12]</td>
<td>Architecture Version. Defines the architecture version of the component. Defined values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0110 Armv8 debug architecture.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0111 Armv8 debug architecture with Virtualization Host Extensions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b1000 Armv8.2 debug architecture, FEAT_Debugv8p2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b1001 Armv8.4 debug architecture, FEAT_Debugv8p4.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b1010 Armv8.8 debug architecture, FEAT_Debugv8p8.</td>
</tr>
<tr>
<td>ARCHPART</td>
<td>[11:0]</td>
<td>EDDEVARCH.ARCHVER and EDDEVARCH.ARCHID are also defined as a single field, EDDEVARCH.ARCHID, so that EDDEVARCH.ARCHVER is EDDEVARCH.ARCHID[15:12].</td>
</tr>
</tbody>
</table>

ARCHITECT, bits [31:21]

Defines the architecture of the component. For debug, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, 0x4.

Bits [27:21] are the JEP106 ID code, 0x3B.

Reads as 0b01000111011.

Access to this field is RO.

PRESENT, bit [20]

Indicates that the DEVARCH is present.

Reads as 0b1.

Access to this field is RO.

REVISION, bits [19:16]

Defines the architecture revision. For architectures defined by Arm this is the minor revision. For debug, the revision defined by Armv8 is 0x0.

All other values are reserved.

Reads as 0b0000.

Access to this field is RO.
FEAT_VHE adds the functionality identified by the value 0b0111.
FEAT_Debugv8p2 adds the functionality identified by the value 0b1000.
FEAT_Debugv8p4 adds the functionality identified by the value 0b1001.
FEAT_Debugv8p8 adds the functionality identified by the value 0b1010.
From Armv8.1, when FEAT_VHE is implemented the value 0b0110 is not permitted.
From Armv8.2, the values 0b0110 and 0b0111 are not permitted.
From Armv8.4, the value 0b1000 is not permitted.
From Armv8.8, the value 0b1001 is not permitted.

ARCHPART, bits [11:0]
Architecture Part. Defines the architecture of the component.
0xA15 Armv8-A debug architecture.
EDDEVARCH.ARCHVER and EDDEVARCH.ARCHPART are also defined as a single field, EDDEVARCH.ARCHID, so that EDDEVARCH.ARCHPART is EDDEVARCH.ARCHID[11:0].
Armv8-A debug architecture.
Access to this field is RO.

Accessing the EDDEVARCH:
EDDEVARCH can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xF8C</td>
<td>EDDEVARCH</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
H9.2.20 EDDEVID, External Debug Device ID register 0

The EDDEVID characteristics are:

Purpose

Provides extra information for external debuggers about features of the debug implementation.

Configurations

If FEAT_DoPD is implemented, this register is in the Core power domain.
If FEAT_DoPD is not implemented, this register is in the Debug power domain.

Attributes

EDDEVID is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>Bit 28</th>
<th>Bit 27</th>
<th>Bit 24</th>
<th>Bit 23</th>
<th>Bit 8</th>
<th>Bit 7</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>AuxRegs</td>
<td>RES0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PCSample</td>
<td>DebugPower</td>
<td></td>
</tr>
</tbody>
</table>

Bits [31:28]

Reserved, RES0.

AuxRegs, bits [27:24]

Indicates support for Auxiliary registers. Defined values are:
0b0000 None supported.
0b0001 Support for External Debug Auxiliary Control Register, EDACR.
All other values are reserved.

Bits [23:8]

Reserved, RES0.

DebugPower, bits [7:4]

Indicates support for the FEAT_DoPD feature. Defined values are:
0b0000 FEAT_DoPD not implemented. Registers in the external debug interface register map are implemented in a mix of the Debug and Core power domains.
0b0001 FEAT_DoPD implemented. All registers in the external debug interface register map are implemented in the Core power domain.
FEAT_DoPD implements the functionality added by the value 0b0001.
All other values are reserved.

PCSample, bits [3:0]

Indicates the level of PC Sample-based Profiling support using external debug registers. Defined values are:
0b0000 PC Sample-based Profiling Extension is not implemented in the external debug registers space.
0b0010 Only EDPCSR and EDCIDSR are implemented. This option is only permitted if EL3 and EL2 are not implemented.
0b0011 EDPCSR, EDCIDSR, and EDVIDSR are implemented.
All other values are reserved.

When FEAT_PCSRv8p2 is implemented, the only permitted value is 0b0000.
External Debug Register Descriptions
H9.2 External debug registers

--- Note ---
FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors register space, as indicated by the value of PMDEVID.PCSample.

Accessing the EDDEVID:

EDDEVID can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFC8</td>
<td>EDDEVID</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
H9.2.21  EDDEVID1, External Debug Device ID register 1

The EDDEVID1 characteristics are:

**Purpose**

Provides extra information for external debuggers about features of the debug implementation.

**Configurations**

If FEAT_DoPD is implemented, this register is in the Core power domain.

If FEAT_DoPD is not implemented, this register is in the Debug power domain.

**Attributes**

EDDEVID1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:8]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[7:4]</td>
<td>HSR</td>
</tr>
<tr>
<td>[3:0]</td>
<td>PCSROffset</td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**HSR, bits [7:4]**

Indicates support for the External Debug Halt Status Register (EDHSR). Defined values are:

- **0b0000**  EDHSR is not implemented. The PE follows behaviors consistent with the EDHSR fields having a zero value.
- **0b0001**  EDHSR is implemented.

All other values are reserved.

If FEAT_SME is implemented, the permitted values are **0b0000** and **0b0001**.

If FEAT_SME is not implemented, the only permitted value is **0b0000**.

**PCSROffset, bits [3:0]**

Indicates the offset applied to PC samples returned by reads of EDPCSR. Permitted values of this field in Armv8 are:

- **0b0000**  EDPCSR not implemented.
- **0b0010**  EDPCSR implemented, and samples have no offset applied and do not sample the instruction set state in AArch32 state.

When FEAT_PCSRv8p2 is implemented, the only permitted value is **0b0000**.

---

**Note**

FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors register space, as indicated by the value of PMDEVID.PCSample.

---

**Accessing the EDDEVID1:**

EDDEVID1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFC4</td>
<td>EDDEVID1</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
### H9.2.22 EDDEVID2, External Debug Device ID register 2

The EDDEVID2 characteristics are:

**Purpose**

Reserved for future descriptions of features of the debug implementation.

**Configurations**

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

**Attributes**

EDDEVID2 is a 32-bit register.

**Field descriptions**

| Bits [31:0] | RES0 |

Reserved, RES0.

**Accessing the EDDEVID2:**

EDDEVID2 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFC0</td>
<td>EDDEVID2</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
H9.2.23 EDDEVTYPE, External Debug Device Type register

The EDDEVTYPE characteristics are:

**Purpose**

Indicates to a debugger that this component is part of a PE's debug logic.

**Configurations**

Implementation of this register is **OPTIONAL**.

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

**Attributes**

EDDEVTYPE is a 32-bit register.

**Field descriptions**

| Bits [31:8] | Reserved, RES0. |
| SUB, bits [7:4] | Subtype. Indicates this is a component within a PE. Reads as 0b0001. Access to this field is RO. |
| MAJOR, bits [3:0] | Major type. Indicates this is a debug logic component. Reads as 0b0101. Access to this field is RO. |

**Accessing the EDDEVTYPE:**

EDDEVTYPE can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFCC</td>
<td>EDDEVTYPE</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
E2DFR, External Debug Feature Register

The E2DFR characteristics are:

**Purpose**

Provides top level information about the debug system.

--- Note ---

Debuggers must use EDDEVARCH to determine the Debug architecture version.

---

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers on page D17-5551.

**Configurations**

It is **IMPLEMENTATION DEFINED** whether E2DFR is implemented in the Core power domain or in the Debug power domain.

**Attributes**

E2DFR is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
<th>Value</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:60]</td>
<td>Reserved, UNKNOWN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[59:56]</td>
<td>Reserved, RES0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[55:44]</td>
<td>Reserved, UNKNOWN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[43:40]</td>
<td>TraceFilt, bits [43:40]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Armv8.4 Self-hosted Trace Extension version. Defined values are:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0b0000 Armv8.4 Self-hosted Trace Extension is not implemented.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0b0001 Armv8.4 Self-hosted Trace Extension is implemented.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>All other values are reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FEAT_TRF implements the functionality added by 0b0001.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>From Armv8.4, the permitted values are 0b0000 and 0b0001.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[39:32]</td>
<td>Reserved, UNKNOWN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[31:28]</td>
<td>CTX_CMPs, bits [31:28]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of breakpoints that are context-aware, minus 1. These are the highest numbered breakpoints.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>In an Armv8-A implementation that supports AArch64, this field returns the value of ID_AA64DFR0_EL1.CTX_CMPs.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

**Bits [27:24]**
Reserved, RES0.

**WRPs, bits [23:20]**
Number of watchpoints, minus 1. The value of 0b0000 is reserved.
In an Armv8-A implementation that supports AArch64, this field returns the value of ID_AA64DFR0_EL1.WRPs.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

**Bits [19:16]**
Reserved, RES0.

**BRPs, bits [15:12]**
Number of breakpoints, minus 1. The value of 0b0000 is reserved.
In an Armv8-A implementation that supports AArch64, this field returns the value of ID_AA64DFR0_EL1.BRPs.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

**PMUVer, bits [11:8]**
Performance Monitors Extension version.
This field does not follow the standard ID scheme, but uses the alternative ID scheme described in Alternative ID scheme used for the Performance Monitors Extension version on page D17-5553.
Defined values are:

- **0b0000**: Performance Monitors Extension not implemented.
- **0b0001**: Performance Monitors Extension, PMUv3 implemented.
- **0b0100**: PMUv3 for Armv8.1. As 0b0001, and adds support for:
  - Extended 16-bit PMEVTYPE<El0>.evtCount field.
  - If EL2 is implemented, the MDCR_EL2.HPMD control.
- **0b0101**: PMUv3 for Armv8.4. As 0b0100, and adds support for the PMMIR_EL1 register.
- **0b0110**: PMUv3 for Armv8.5. As 0b0101, and adds support for:
  - 64-bit event counters.
  - If EL2 is implemented, the MDCR_EL2.HCCD control.
  - If EL3 is implemented, the MDCR_EL3.SCCD control.
- **0b0111**: PMUv3 for Armv8.7. As 0b0110, and adds support for:
  - The PMCR_EL0.FZO and, if EL2 is implemented, MDCR_EL2.HPMFZO controls.
  - If EL3 is implemented, the MDCR_EL3.\{MPMX,MCCD\} controls.
- **0b1000**: PMUv3 for Armv8.8. As 0b0111, and:
  - Extends the Common event number space to include 0x0040 to 0x00BF and 0x4040 to 0x40BF.
  - Removes the CONSTRAINED UNPREDICTABLE behaviors if a reserved or unimplemented PMU event number is selected.
- **0b1111**: IMPLEMENTATION DEFINED form of performance monitors supported, PMUv3 not supported. Arm does not recommend this value for new implementations.
All other values are reserved.

**FEAT_PMUv3** implements the functionality identified by the value \(0b0001\).

**FEAT_PMUv3p1** implements the functionality identified by the value \(0b0100\).

**FEAT_PMUv3p4** implements the functionality identified by the value \(0b0101\).

**FEAT_PMUv3p5** implements the functionality identified by the value \(0b0110\).

**FEAT_PMUv3p7** implements the functionality identified by the value \(0b0111\).

**FEAT_PMUv3p8** implements the functionality identified by the value \(0b1000\).

From Armv8.1, if **FEAT_PMUv3** is implemented, the value \(0b0001\) is not permitted.

From Armv8.4, if **FEAT_PMUv3** is implemented, the value \(0b0100\) is not permitted.

From Armv8.5, if **FEAT_PMUv3** is implemented, the value \(0b0101\) is not permitted.

From Armv8.7, if **FEAT_PMUv3** is implemented, the value \(0b0110\) is not permitted.

From Armv8.8, if **FEAT_PMUv3** is implemented, the value \(0b0111\) is not permitted.

**TraceVer, bits [7:4]**

Trace support. Indicates whether System register interface to a trace unit is implemented. Defined values are:

- \(0b0000\) Trace unit System registers not implemented.
- \(0b0001\) Trace unit System registers implemented.

All other values are reserved.

A value of \(0b0000\) only indicates that no System register interface to a trace unit is implemented. A trace unit might nevertheless be implemented without a System register interface.

In an Armv8-A implementation that supports AArch64, this field returns the value of ID_AA64DFR0_EL1.TraceVer.

**Bits [3:0]**

Reserved, UNKNOWN.

**Accessing the EDDFR:**

EDDFR[31:0] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x028</td>
<td>EDDFR</td>
<td>31:0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered() and !DoubleLockStatus() accesses to EDDFR[31:0] are RO.
- Otherwise accesses to EDDFR[31:0] are IMPDEF.

EDDFR[63:32] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x02C</td>
<td>EDDFR</td>
<td>63:32</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered() and !DoubleLockStatus() accesses to EDDFR[63:32] are RO.
- Otherwise accesses to EDDFR[63:32] are IMPDEF.
H9.2.25  EDECCR, External Debug Exception Catch Control Register

The EDECCR characteristics are:

**Purpose**

Controls Exception Catch debug events. For more information, see Exception Catch debug event on page H3-10287.

**Configurations**

External register EDECCR bits [31:0] are architecturally mapped to AArch64 System register OSECCR_EL1[31:0].

External register EDECCR bits [31:0] are architecturally mapped to AArch32 System register DBGOSECCR[31:0].

EDECCR is in the Core power domain.

**Attributes**

EDECCR is a 32-bit register.

**Field descriptions**

Bits [31:23]

Reserved, RES0.

RLR2, bit [22]

When FEAT_RME is implemented:

Controls exception catch on exception return to Realm EL2 in conjunction with EDECCR.RLE2.

0b0  If EDECCR.RLE2 is 0, then Exception Catch debug events are disabled for Realm EL2.

0b1  If EDECCR.RLE2 is 1, then Exception Catch debug events are enabled for exception entry to Realm EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.

**Otherwise:**

Reserved, RES0.
RLR1, bit [21]

When FEAT_RME is implemented:

Controls exception catch on exception return to Realm EL1 in conjunction with EDECCR.RLE1.

0b0 If EDECCR.RLE1 is 0, then Exception Catch debug events are disabled for Realm EL1.
If EDECCR.RLE1 is 1, then Exception Catch debug events are enabled for exception entry and exception return to Realm EL1.

0b1 If EDECCR.RLE1 is 0, then Exception Catch debug events are enabled for exception entry to Realm EL1.
If EDECCR.RLE1 is 1, then Exception Catch debug events are enabled for exception entry to Realm EL1.

The reset behavior of this field is:
• On a Cold reset, this field resets to 0.

Otherwise:
Reserved, RES0.

RLR0, bit [20]

When FEAT_RME is implemented:

Controls exception catch on exception return to Realm EL0.

0b0 Exception Catch debug events are disabled for Realm EL0.

0b1 Exception Catch debug events are enabled for exception returns to Realm EL0.

The reset behavior of this field is:
• On a Cold reset, this field resets to 0.

Otherwise:
Reserved, RES0.

Bit [19]

Reserved, RES0.

RLE2, bit [18]

When FEAT_RME is implemented:

Controls exception catch on exception entry to Realm EL2. Also controls exception catch on exception return to Realm EL2 in conjunction with EDECCR.RLR2.

0b0 If EDECCR.RLR2 is 0, then Exception Catch debug events are disabled for Realm EL2.
If EDECCR.RLR2 is 1, then Exception Catch debug events are enabled for exception returns to Realm EL2.

0b1 If EDECCR.RLR2 is 0, then Exception Catch debug events are enabled for exception entry and exception return to Realm EL2.
If EDECCR.RLR2 is 1, then Exception Catch debug events are enabled for exception entry to Realm EL2.

The reset behavior of this field is:
• On a Cold reset, this field resets to 0.

Otherwise:
Reserved, RES0.

RLE1, bit [17]

When FEAT_RME is implemented:

Controls exception catch on exception entry to Realm EL1. Also controls exception catch on exception return to Realm EL1 in conjunction with EDECCR.RLR1.

0b0 If EDECCR.RLR1 is 0, then Exception Catch debug events are disabled for Realm EL1.
If EDECCR.RLR1 is 1, then Exception Catch debug events are enabled for exception returns to Realm EL1.

- If EDECCR.RLR1 is 0, then Exception Catch debug events are enabled for exception entry and exception return to Realm EL1.
- If EDECCR.RLR1 is 1, then Exception Catch debug events are enabled for exception entry to Realm EL1.

The reset behavior of this field is:
- On a Cold reset, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**RLE0, bit [16]**
Access to this field is RES0.

**NSR3, bit [15]**
Access to this field is RES0.

**NSR2, bit [14]**

*When FEAT_Debugv8p2 is implemented and Non-secure EL2 is implemented:*
Controls exception catch on exception return to Non-secure EL2 in conjunction with EDECCR.NSE2.

- If EDECCR.NSE2 is 0, then Exception Catch debug events are disabled for Non-secure EL2.
- If EDECCR.NSE2 is 1, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Non-secure EL2.

The reset behavior of this field is:
- On a Cold reset, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**NSR1, bit [13]**

*When FEAT_Debugv8p2 is implemented and Non-secure EL1 is implemented:*
Controls exception catch on exception return to Non-secure EL1 in conjunction with EDECCR.NSE1.

- If EDECCR.NSE1 is 0, then Exception Catch debug events are disabled for Non-secure EL1.
- If EDECCR.NSE1 is 1, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Non-secure EL1.

The reset behavior of this field is:
- On a Cold reset, this field resets to 0.

**Otherwise:**
Reserved, RES0.
NSR0, bit [12]

When `FEAT_Debugv8p2` is implemented and Non-secure EL0 is implemented:

Controls exception catch on exception return to Non-secure EL0.

- **0b0**: Exception Catch debug events are disabled for Non-secure EL0.
- **0b1**: Exception Catch debug events are enabled for exception returns to Non-secure EL0.

The reset behavior of this field is:
- On a Cold reset, this field resets to 0.

Otherwise:
Reserved, RES0.


When `FEAT_Debugv8p2` is implemented and EL3 is implemented:

Controls exception catch on exception return to EL3 in conjunction with EDECCR.SE3.

- **0b0**: If EDECCR.SE3 is 0, then Exception Catch debug events are disabled for EL3.
  
  If EDECCR.SE3 is 1, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to EL3.

- **0b1**: If EDECCR.SE3 is 0, then Exception Catch debug events are enabled for exception returns to EL3.
  
  If EDECCR.SE3 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to EL3.

The reset behavior of this field is:
- On a Cold reset, this field resets to 0.

Otherwise:
Reserved, RES0.

SR2, bit [10]

When `FEAT_Debugv8p2` is implemented and `FEAT_SEL2` is implemented:

Controls exception catch on exception return to Secure EL2 in conjunction with EDECCR.SE2.

- **0b0**: If EDECCR.SE2 is 0, then Exception Catch debug events are disabled for Secure EL2.
  
  If EDECCR.SE2 is 1, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Secure EL2.

- **0b1**: If EDECCR.SE2 is 0, then Exception Catch debug events are enabled for exception returns to Secure EL2.
  
  If EDECCR.SE2 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Secure EL2.

The reset behavior of this field is:
- On a Cold reset, this field resets to 0.

Otherwise:
Reserved, RES0.

SR1, bit [9]

When `FEAT_Debugv8p2` is implemented and Secure EL1 is implemented:

Controls exception catch on exception return to Secure EL1 in conjunction with EDECCR.SE1.

- **0b0**: If EDECCR.SE1 is 0, then Exception Catch debug events are disabled for Secure EL1.
  
  If EDECCR.SE1 is 1, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Secure EL1.

- **0b1**: If EDECCR.SE1 is 0, then Exception Catch debug events are enabled for exception returns to Secure EL1.
If EDECCR.SE1 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Secure EL1.

The reset behavior of this field is:
- On a Cold reset, this field resets to 0.

Otherwise:
Reserved, RES0.

SR0, bit [8]

When FEAT_Debugv8p2 is implemented and Secure EL0 is implemented:
Controls exception catch on exception return to Secure EL0.
0b0 Exception Catch debug events are disabled for Secure EL0.
0b1 Exception Catch debug events are enabled for exception returns to Secure EL0.

The reset behavior of this field is:
- On a Cold reset, this field resets to 0.

Otherwise:
Reserved, RES0.

NSE3, bit [7]
Access to this field is RES0.

NSE2, bit [6]

When FEAT_Debugv8p2 is implemented and Non-secure EL2 is implemented:
Controls exception catch on exception entry to Non-secure EL2. Also controls exception catch on exception return to Non-secure EL2 in conjunction with EDECCR.NSR2.
0b0 If EDECCR.NSR2 is 0, then Exception Catch debug events are disabled for Non-secure EL2.
0b1 If EDECCR.NSR2 is 1, then Exception Catch debug events are enabled for exception returns to Non-secure EL2.

The reset behavior of this field is:
- On a Cold reset, this field resets to 0.

Otherwise:
Reserved, RES0.

Note
It is IMPLEMENTATION DEFINED whether a reset entry to an Exception level will generate an Exception Catch debug event.

The reset behavior of this field is:
- On a Cold reset, this field resets to 0.

When Non-secure EL2 is implemented:
Coarse-grained exception catch for Non-secure EL2. Controls Exception Catch debug events for Non-secure EL2.
0b0 Exception Catch debug events are disabled for Non-secure EL2.
0b1 Exception Catch debug events are enabled for Non-secure EL2.

The reset behavior of this field is:
- On a Cold reset, this field resets to 0.

Otherwise:
Reserved, RES0.
NSE1, bit \[5\]

*When FEAT_Debugv8p2 is implemented and Non-secure EL1 is implemented:*

Controls exception catch on exception entry to Non-secure EL1. Also controls exception catch on exception return to Non-secure EL1 in conjunction with EDECCR.NSR1.

0b0  
If EDECCR.NSR1 is 0, then Exception Catch debug events are disabled for Non-secure EL1.
If EDECCR.NSR1 is 1, then Exception Catch debug events are enabled for exception returns to Non-secure EL1.

0b1  
If EDECCR.NSR1 is 0, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Non-secure EL1.
If EDECCR.NSR1 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Non-secure EL1.

--- Note ---

It is IMPLEMENTATION DEFINED whether a reset entry to an Exception level will generate an Exception Catch debug event.

---

The reset behavior of this field is:

* On a Cold reset, this field resets to 0.

NSE0, bit \[4\]

Access to this field is RES0.

SE3, bit \[3\]

*When FEAT_Debugv8p2 is implemented and EL3 is implemented:*

Controls exception catch on exception entry to EL3. Also controls exception catch on exception return to EL3 in conjunction with EDECCR.SR3.

0b0  
Exception Catch debug events are disabled for Non-secure EL1.

0b1  
Exception Catch debug events are enabled for Non-secure EL1.

The reset behavior of this field is:

* On a Cold reset, this field resets to 0.

--- Note ---

It is IMPLEMENTATION DEFINED whether a reset entry to an Exception level will generate an Exception Catch debug event.

---

The reset behavior of this field is:

* On a Cold reset, this field resets to 0.
When FEAT_Debugv8p2 is not implemented and EL3 is implemented:

Coarse-grained exception catch for EL3. Controls Exception Catch debug events for EL3.

- 0b0: Exception Catch debug events are disabled for EL3.
- 0b1: Exception Catch debug events are enabled for EL3.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.

Otherwise:

Reserved, RES0.

SE2, bit [2]

When FEAT_Debugv8p2 is implemented and FEAT_SEL2 is implemented:

Controls exception catch on exception entry to Secure EL2. Also controls exception catch on exception return to Secure EL2 in conjunction with EDECCR.SR2.

- 0b0: If EDECCR.SR2 is 0, then Exception Catch debug events are disabled for Secure EL2. If EDECCR.SR2 is 1, then Exception Catch debug events are enabled for exception returns to Secure EL2.
- 0b1: If EDECCR.SR2 is 0, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Secure EL2. If EDECCR.SR2 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Secure EL2.

--- Note ---

It is IMPLEMENTATION DEFINED whether a reset entry to an Exception level will generate an Exception Catch debug event.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.

Otherwise:

Reserved, RES0.

SE1, bit [1]

When FEAT_Debugv8p2 is implemented and Secure EL1 is implemented:

Controls exception catch on exception entry to Secure EL1. Also controls exception catch on exception return to Secure EL1 in conjunction with EDECCR.SR1.

- 0b0: If EDECCR.SR1 is 0, then Exception Catch debug events are disabled for Secure EL1. If EDECCR.SR1 is 1, then Exception Catch debug events are enabled for exception returns to Secure EL1.
- 0b1: If EDECCR.SR1 is 0, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Secure EL1. If EDECCR.SR1 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Secure EL1.

--- Note ---

It is IMPLEMENTATION DEFINED whether a reset entry to an Exception level will generate an Exception Catch debug event.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.
**When Secure EL1 is implemented:**

Coarse-grained exception catch for Secure EL1. Controls Exception Catch debug events for Secure EL1.

- **0b0** Exception Catch debug events are disabled for Secure EL1.
- **0b1** Exception Catch debug events are enabled for Secure EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.

**Otherwise:**

Reserved, RES0.

**SE0, bit [0]**

Access to this field is RES0.

**Accessing the EDECCR:**

EDECCR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x098</td>
<td>EDECCR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.
H9.2.26   EDECR, External Debug Execution Control Register

The EDECR characteristics are:

Purpose
Controls Halting debug events.

Configurations
- If FEAT_DoPD is implemented, this register is in the Core power domain.
- If FEAT_DoPD is not implemented, this register is in the Debug power domain.

Attributes
- EDECR is a 32-bit register.

Field descriptions

Res0  SS  RCE  OSUCE

Bits [31:3]
Reserved, RES0.

SS, bit [2]
- Halting step enable. Possible values of this field are:
  - 0b0  Halting step debug event disabled.
  - 0b1  Halting step debug event enabled.

If the value of EDECR.SS is changed when the PE is in Non-debug state, behavior is CONSTRAINED UNPREDICTABLE as described in Changing the value of EDECR.SS when not in Debug state on page H3-10283.

The reset behavior of this field is:
- On a Cold reset, when FEAT_DoPD is implemented, this field resets to 0.
- On an External debug reset, when FEAT_DoPD is not implemented, this field resets to 0.

RCE, bit [1]
When FEAT_DoPD is not implemented:
- Reset Catch Enable.
  - 0b0  Reset Catch debug event disabled.
  - 0b1  Reset Catch debug event enabled.

The reset behavior of this field is:
- On an External debug reset, this field resets to 0.

Otherwise:
- Reserved, RES0.

OSUCE, bit [0]
When FEAT_DoPD is not implemented:
- OS Unlock Catch Enable.
  - 0b0  OS Unlock Catch debug event disabled.
  - 0b1  OS Unlock Catch debug event enabled.
The reset behavior of this field is:

- On an External debug reset, this field resets to 0.

Otherwise:

Reserved, RES0.

**Accessing the EDECR:**

EDECR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x024</td>
<td>EDECR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When (FEAT_DoPD is not implemented or IsCorePowered()) and SoftwareLockStatus() accesses to this register are RO.

- When (FEAT_DoPD is not implemented or IsCorePowered()) and !SoftwareLockStatus() accesses to this register are RW.

- Otherwise accesses to this register generate an error response.
H9.2.27  EDESR, External Debug Event Status Register

The EDESR characteristics are:

Purpose

Indicates the status of internally pending Halting debug events.

Configurations

EDESR is in the Core power domain.

Attributes

EDESR is a 32-bit register.

Field descriptions

Bits [31:4]

Reserved, RES0.

EC, bit [3]

When FEAT_Debugv8p8 is implemented:

Exception Catch debug event pending.

0b0  Exception Catch debug event is not pending.

0b1  Exception Catch debug event is pending.

The reset behavior of this field is:

•  On a Warm reset, this field resets to 0.

Access to this field is W1C.

Otherwise:

Reserved, RES0.

SS, bit [2]

When FEAT_DoPD is implemented:

Halting step debug event pending. Possible values of this field are:

0b0  Reading this means that a Halting step debug event is not pending. Writing this means no action.

0b1  Reading this means that a Halting step debug event is pending. Writing this clears the pending Halting step debug event.

The reset behavior of this field is:

•  On a Cold reset, this field resets to 0.

Otherwise:

Halting step debug event pending. Possible values of this field are:

0b0  Reading this means that a Halting step debug event is not pending. Writing this means no action.

0b1  Reading this means that a Halting step debug event is pending. Writing this clears the pending Halting step debug event.
The reset behavior of this field is:

- On a Warm reset, this field resets to the value in EDECR.SS.

**RC, bit [1]**

Reset Catch debug event pending. Possible values of this field are:

0b0 Reading this means that a Reset Catch debug event is not pending. Writing this means no action.

0b1 Reading this means that a Reset Catch debug event is pending. Writing this clears the pending Reset Catch debug event.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_DoPD is implemented, this field resets to the value in CTIDEVCTL.RCE.
  - When FEAT_DoPD is not implemented, this field resets to the value in EDECR.RCE.

**OSUC, bit [0]**

OS Unlock Catch debug event pending. Possible values of this field are:

0b0 Reading this means that an OS Unlock Catch debug event is not pending. Writing this means no action.

0b1 Reading this means that an OS Unlock Catch debug event is pending. Writing this clears the pending OS Unlock Catch debug event.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.

**Accessing the EDESR:**

If a request to clear a pending Halting debug event is received at or about the time when halting becomes allowed, it is CONstrained UNPREDICTABLE whether the event is taken.

If Core power is removed while a Halting debug event is pending, it is lost. However, it might become pending again when the Core is powered back on and Cold reset.

EDESR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x020</td>
<td>EDESR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.
H9.2.28  EDHSR, External Debug Halt Status Register

The EDHSR characteristics are:

**Purpose**

Provides Debug Halt Status information.

**Configurations**

EDHSR is in the Core power domain.

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to EDHSR are RES0.

This register is only valid when the PE is in Debug state and \texttt{EDSCR.STATUS} is 0b101011, indicating a Watchpoint debug event. Otherwise, it has an UNKNOWN value.

The field \texttt{EDDEVID1.HSR} indicates support for this register.

**Attributes**

EDHSR is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-24</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>31-23</td>
<td>WPT</td>
<td>0 to 15</td>
</tr>
<tr>
<td>18-16</td>
<td>All other values</td>
<td>Reserved</td>
</tr>
<tr>
<td>15-10</td>
<td>The reset behavior of this field is:</td>
<td></td>
</tr>
<tr>
<td>9-0</td>
<td>WPTV</td>
<td>0b0</td>
</tr>
<tr>
<td>10-9</td>
<td>The WPT field is invalid, and holds an UNKNOWN value.</td>
<td></td>
</tr>
<tr>
<td>9-0</td>
<td>WPTV</td>
<td>0b1</td>
</tr>
<tr>
<td>10-9</td>
<td>The WPT field is valid, and holds the number of a watchpoint that triggered an entry to Debug state.</td>
<td></td>
</tr>
</tbody>
</table>

When an entry to Debug state is triggered by a watchpoint match:

- If the PE sets any of \texttt{FnV}, \texttt{FnP}, or \texttt{WPF} to 1, then the PE sets \texttt{WPTV} to 1.
- If the PE sets all of \texttt{FnV}, \texttt{FnP}, and \texttt{WPF} to 0, then the PE sets \texttt{WPTV} to an IMPLEMENTATION DEFINED value, 0 or 1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
WPF, bit [16]

Watchpoint might be false-positive.

0b0 The watchpoint matched the original access or set of contiguous accesses.
0b1 The watchpoint matched an access or set of contiguous accesses where the lowest accessed address was rounded down to the nearest multiple of 16 bytes and the highest accessed address was rounded up to the nearest multiple of 16 bytes minus 1, but the watchpoint might not have matched the original access or set of contiguous accesses.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

FnP, bit [15]

FAR not Precise.

This field only has meaning if the EDWAR is valid; that is, when the FnV field is 0. If the FnV field is 1, the FnP field is 0.

0b0 If the FnV field is 0, the EDWAR holds the virtual address of an access or set of contiguous accesses that triggered an entry to Debug state.
0b1 The EDWAR holds any address within the smallest implemented translation granule that contains the virtual address of an access or set of contiguous accesses that triggered an entry to Debug state.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [14:11]

Reserved, RES0.

FnV, bit [10]

FAR not Valid.

0b0 The EDWAR is valid, and its value is as described by the FnP field.
0b1 The EDWAR is invalid, and holds an UNKNOWN value.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [9:0]

Reserved, RES0.

Accessing the EDHSR:

EDHSR[31:0] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x038</td>
<td>EDHSR</td>
<td>31:0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:
• When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to EDHSR[31:0] are RO.
• Otherwise accesses to EDHSR[31:0] generate an error response.
EDHSR[63:32] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x03C</td>
<td>EDHSR</td>
<td>63:32</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to EDHSR[63:32] are RO.
- Otherwise accesses to EDHSR[63:32] generate an error response.
H9.2.29 EDITCTRL, External Debug Integration mode Control register

The EDITCTRL characteristics are:

**Purpose**

Enables the external debug to switch from its default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

**Configurations**

It is IMPLEMENTATION DEFINED whether EDITCTRL is implemented in the Core power domain or in the Debug power domain.

Implementation of this register is OPTIONAL.

**Attributes**

EDITCTRL is a 32-bit register.

**Field descriptions**

![Field description diagram](image)

**Bits [31:1]**

Reserved, RES0.

**IME, bit [0]**

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection. The integration mode behavior is IMPLEMENTATION DEFINED.

0b0 Normal operation.

0b1 Integration mode enabled.

The reset behavior of this field is:

- The following resets apply:
  - Whichever power domain the register is implemented in, this field resets to 0.
  - Otherwise, the value of this field is unchanged.

**Accessing the EDITCTRL:**

EDITCTRL can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xF00</td>
<td>EDITCTRL</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
• Otherwise accesses to this register are IMPDEF.
H9.2.30   EDITR, External Debug Instruction Transfer Register

The EDITR characteristics are:

**Purpose**

Used in Debug state for passing instructions to the PE for execution.

**Configurations**

EDITR is in the Core power domain.

**Attributes**

EDITR is a 32-bit register.

**Field descriptions**

*When AArch32 is supported and in AArch32 state:*

| 31 | hw2 | 16 | 15 | hw1 | 0 |

**hw2, bits [31:16]**

Second halfword of the T32 instruction to be executed on the PE. When EDITR contains a 16-bit T32 instruction, this field is ignored. For more information, see Behavior in Debug state on page H2-10242.

**Note**

The hw2 field is displayed on the left. This is not the usual convention for display of T32 instruction halfwords.

**hw1, bits [15:0]**

First halfword of the T32 instruction to be executed on the PE.

**Note**

The hw1 field is displayed on the right. This is not the usual convention for display of T32 instruction halfwords.

*When AArch64 is supported and in AArch64 state:*

| 31 | A64 instruction to be executed on the PE | 0 |

**Bits [31:0]**

A64 instruction to be executed on the PE.

**Accessing the EDITR:**

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any instruction issued through the ITR in Normal access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

EDITR ignores writes if the PE is in Non-debug state.

EDITR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x004</td>
<td>EDITR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are WI.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are WO.
- Otherwise accesses to this register generate an error response.
H9.2.31   EDLAR, External Debug Lock Access Register

The EDLAR characteristics are:

Purpose

Allows or disallows access to the external debug registers through a memory-mapped interface. The optional Software Lock provides a lock to prevent memory-mapped writes to the debug registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the debug registers. It does not, and cannot, prevent all accidental or malicious damage.

Configurations

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE in the Core power domain.

If FEAT_DoPD is not implemented, this register is in the Debug power domain.

Software uses EDLAR to set or clear the lock, and EDLSR to check the current status of the lock.

Attributes

EDLAR is a 32-bit register.

Field descriptions

When Software Lock is implemented:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY</td>
<td>Lock Access control. Writing the key value 0xC5ACCE55 to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface. Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.</td>
</tr>
</tbody>
</table>

Otherwise:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

Accessing the EDLAR:

EDLAR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFB0</td>
<td>EDLAR</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are WO.
- Otherwise accesses to this register generate an error response.
H9.2.32 EDLSR, External Debug Lock Status Register

The EDLSR characteristics are:

**Purpose**

Indicates the current status of the software lock for external debug registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the debug registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the debug registers. It does not, and cannot, prevent all accidental or malicious damage.

**Configurations**

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE in the Core power domain.

If FEAT_DoPD is not implemented, this register is in the Debug power domain.

Software uses EDLAR to set or clear the lock, and EDLSR to check the current status of the lock.

**Attributes**

EDLSR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>3</td>
<td>nTT, bit [2] Not thirty-two bit access required. RAZ.</td>
</tr>
<tr>
<td>2</td>
<td>SLK, bit [1] When Software Lock is implemented:</td>
</tr>
<tr>
<td></td>
<td>Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when Software Lock is not implemented, this field is RES0.</td>
</tr>
<tr>
<td></td>
<td>For memory-mapped accesses when Software Lock is implemented, possible values of this field are:</td>
</tr>
<tr>
<td></td>
<td>0b0 Lock clear. Writes are permitted to this component's registers.</td>
</tr>
<tr>
<td></td>
<td>0b1 Lock set. Writes to this component's registers are ignored, and reads have no side effects.</td>
</tr>
<tr>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On a Cold reset, when FEAT_DoPD is implemented, this field resets to 1.</td>
</tr>
<tr>
<td></td>
<td>• On an External debug reset, when FEAT_DoPD is not implemented, this field resets to 1.</td>
</tr>
<tr>
<td>1</td>
<td>SLI, bit [0] Otherwise:</td>
</tr>
<tr>
<td></td>
<td>Reserved, RAZ.</td>
</tr>
<tr>
<td>0</td>
<td>SLI, bit [0] Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ. For memory-mapped accesses, the value of this field is IMPLEMENTATION DEFINED. Permitted values are:</td>
</tr>
<tr>
<td></td>
<td>0b0 Software Lock not implemented or not memory-mapped access.</td>
</tr>
</tbody>
</table>
Software Lock implemented and memory-mapped access.

**Accessing the EDLSR:**

EDLSR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xF84</td>
<td>EDLSR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When `FEAT_DoPD` is not implemented or `IsCorePowered()` accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
H9.2.33  **EDPCSR, External Debug Program Counter Sample Register**

The EDPCSR characteristics are:

**Purpose**

Holds a sampled instruction address value.

**Configurations**

EDPCSR is in the Core power domain.

This register is present only when FEAT_PCSRv8 is implemented and FEAT_PCSRv8p2 is not implemented. Otherwise, direct accesses to EDPCSR are RES0.

EDPCSR[63:32] and EDPCSR[31:0] are accessed at 32-bit memory mapped addresses that are not contiguous.

If FEAT_VHE is implemented, the format of this register differs depending on the value of EDSCR.SC2.

Implemented only if the OPTIONAL PC Sample-based Profiling Extension is implemented in the external debug registers space.

--- **Note** ---

FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors registers space.

**Attributes**

EDPCSR is a 64-bit register.

**Field descriptions**

*When FEAT_VHE is not implemented or EDSCR.SC2 == 0:*

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PC Sample high word, EDPCSRhi</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC Sample low word</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:32]**

PC Sample high word, EDPCSRhi. If EDVIDSR.HV == 0 then this field is RAZ, otherwise bits [63:32] of the sampled instruction address value. The translation regime that EDPCSR samples can be determined from EDVIDSR.{NS,E2,E3}.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Bits [31:0]**

PC Sample low word. EDPCSRlo, bits[31:0] of the sampled instruction address value.

EDPCSRlo reads as 0xffffffff when any of the following are true:

- The PE is in Debug state.
- PC Sample-based profiling is prohibited.

If a branch instruction has retired since the PE left reset state, then the first read of EDPCSR[31:0] is permitted but not required to return 0xffffffff.

EDPCSRlo reads as an UNKNOWN value when any of the following are true:

- The PE is in reset state.
• No branch instruction has retired since the PE left reset state, Debug state, or a state where PC Sample-based Profiling is prohibited.
• No branch instruction has retired since the last read of EDPCSR[31:0].

For the cases where a read of EDPCSR[31:0] returns 0xFFFFFFFF or an UNKNOWN value, the read has the side-effect of setting EDPCSRhi, EDCIDSR, and EDVIDSR to UNKNOWN values. Otherwise, a read of EDPCSR[31:0] returns bits [31:0] of the sampled instruction address value and has the side-effect of indirectly writing to EDPCSRhi, EDCIDSR, and EDVIDSR. The translation regime that EDPCSR samples can be determined from EDVIDSR.\{NS,E2,E3\}. For a read of EDPCSR[31:0] from the memory-mapped interface, if EDLSR.SLK == 1, meaning the OPTIONAL Software Lock is locked, then the side-effect of the access does not occur and EDPCSRhi, EDCIDSR, and EDVIDSR are unchanged.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

**When FEAT_VHE is implemented and EDSCR.SC2 == 1:**

```
<table>
<thead>
<tr>
<th>NS</th>
<th>EL</th>
<th>RES0</th>
<th>PC Sample high word, EDPCSRhi</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
</tr>
<tr>
<td>63</td>
<td>62</td>
<td>61</td>
<td>60</td>
</tr>
</tbody>
</table>
```

**NS, bit [63]**
Non-secure state sample. Indicates the Security state that is associated with the most recent EDPCSR sample or, when it is read as a single atomic 64-bit read, the current EDPCSR sample. The translation regime that EDPCSR samples can be determined from EDPCSR.\{NS,EL\}.

If EL3 is not implemented, this bit indicates the Effective value of SCR.NS.

0b0  Sample is from Secure state.
0b1  Sample is from Non-secure state.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

**EL, bits [62:61]**
Exception level status sample. Indicates the Exception level that is associated with the most recent EDPCSR sample or, when it is read as a single atomic 64-bit read, the current EDPCSR sample. The translation regime that EDPCSR samples can be determined from EDPCSR.\{NS,EL\}.

0b00  Sample is from EL0.
0b01  Sample is from EL1.
0b10  Sample is from EL2.
0b11  Sample is from EL3.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Bits [60:56]**
Reserved, RES0.

**Bits [55:32]**
PC Sample high word, EDPCSRhi. Bits [55:32] of the sampled instruction address value. The translation regime that EDPCSR samples can be determined from EDPCSR.\{NS,EL\}.
The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Bits [31:0]**

PC Sample low word. EDPCSRlo, bits[31:0] of the sampled instruction address value.

EDPCSRlo reads as 0xFFFFFFFF when any of the following are true:
- The PE is in Debug state.
- PC Sample-based profiling is prohibited.

If a branch instruction has retired since the PE left reset state, then the first read of EDPCSR[31:0] is permitted but not required to return 0xFFFFFFFF.

EDPCSRlo reads as an UNKNOWN value when any of the following are true:
- The PE is in reset state.
- No branch instruction has retired since the PE left reset state, Debug state, or a state where PC Sample-based Profiling is prohibited.
- No branch instruction has retired since the last read of EDPCSR[31:0].

For the cases where a read of EDPCSR[31:0] returns 0xFFFFFFFF or an UNKNOWN value, the read has the side-effect of setting EDPCSRhi, EDCIDSR, and EDVIDSR to UNKNOWN values.

Otherwise, a read of EDPCSR[31:0] returns bits [31:0] of the sampled instruction address value and has the side-effect of indirectly writing to EDPCSRhi, EDCIDSR, and EDVIDSR. The translation regime that EDPCSR samples can be determined from EDPCSR.[NS,EL].

For a read of EDPCSR[31:0] from the memory-mapped interface, if EDLSR.SLK == 1, meaning the OPTIONAL Software Lock is locked, then the side-effect of the access does not occur and EDPCSRhi, EDCIDSR, and EDVIDSR are unchanged.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Accessing the EDPCSR:**

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see *Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN* on page H7-10354

EDPCSR[31:0] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x0A0</td>
<td>EDPCSR</td>
<td>31:0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:
- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to EDPCSR[31:0] are RO.
- Otherwise accesses to EDPCSR[31:0] generate an error response.

EDPCSR[63:32] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x0AC</td>
<td>EDPCSR</td>
<td>63:32</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:
- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to EDPCSR[63:32] are RO.
- Otherwise accesses to EDPCSR[63:32] generate an error response.
H9.2.34  EDPFR, External Debug Processor Feature Register

The EDPFR characteristics are:

**Purpose**

Provides information about implemented PE features.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers on page D17-5551.

**Configurations**

It is IMPLEMENTATION DEFINED whether EDPFR is implemented in the Core power domain or in the Debug power domain.

**Attributes**

EDPFR is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>60</th>
<th>59</th>
<th>56</th>
<th>55</th>
<th>52</th>
<th>51</th>
<th>48</th>
<th>47</th>
<th>44</th>
<th>43</th>
<th>40</th>
<th>39</th>
<th>36</th>
<th>35</th>
<th>32</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>24</th>
<th>23</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Unknown]</td>
<td>![GIC]</td>
<td>![AdvSIMD]</td>
<td>![FP]</td>
<td>![EL3]</td>
<td>![EL2]</td>
<td>![EL1]</td>
<td>![EL0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:60]**

*From Armv8.5:*

Reserved, UNKNOWN.

*Otherwise:*

Reserved, RES0.

**Bits [59:56]**

*From Armv8.5:*

Reserved, UNKNOWN.

*Otherwise:*

Reserved, RES0.

**Bits [55:52]**

Reserved, RES0.

**Bits [51:48]**

*From Armv8.4:*

Reserved, UNKNOWN.

*Otherwise:*

Reserved, RES0.

**AMU, bits [47:44]**

Indicates support for Activity Monitors Extension. Defined values are:

0b0000  Activity Monitors Extension is not implemented.

0b0001  FEAT_AMUv1 is implemented.

0b0010  FEAT_AMUv1p1 is implemented. As 0b0001 and adds support for virtualization of the activity monitor event counters.
All other values are reserved.

FEAT_AMUv1 implements the functionality identified by the value 0b0001.

FEAT_AMUv1p1 implements the functionality identified by the value 0b0010.

In Armv8.0, the only permitted value is 0b0000.

In Armv8.4, the permitted values are 0b0000 and 0b0001.

From Armv8.6, the permitted values are 0b0000, 0b0001, and 0b0010.

**Bits [43:40]**

*From Armv8.2:*

Reserved, UNKNOWN.

*Otherwise:*

Reserved, RES0.

**SEL2, bits [39:36]**

Secure EL2. Defined values are:

- 0b0000 Secure EL2 is not implemented.
- 0b0001 Secure EL2 is implemented.

All other values are reserved.

**SVE, bits [35:32]**

Scalable Vector Extension. Defined values are:

- 0b0000 SVE is not implemented.
- 0b0001 SVE is implemented.

All other values are reserved.

**Bits [31:28]**

*From Armv8.2:*

Reserved, UNKNOWN.

*Otherwise:*

Reserved, RES0.

**GIC, bits [27:24]**

System register GIC interface support. Defined values are:

- 0b0000 GIC CPU interface system registers not implemented.
- 0b0001 System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.
- 0b0011 System register interface to version 4.1 of the GIC CPU interface is supported.

All other values are reserved.

In an Armv8-A implementation that supports AArch64, this field returns the value of ID_AA64PFR0_EL1.GIC.

**AdvSIMD, bits [23:20]**

Advanced SIMD. Defined values are:

- 0b0000 Advanced SIMD is implemented, including support for the following SISD and SIMD operations:
  - Integer byte, halfword, word and doubleword element operations.
  - Single-precision and double-precision floating-point arithmetic.
  - Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.
- 0b0001 As for 0b0000, and also includes support for half-precision floating-point arithmetic.
0b1111        Advanced SIMD is not implemented.
All other values are reserved.
This field must have the same value as the FP field.
The permitted values are:
  • 0b0000 in an implementation with Advanced SIMD support, that does not include the
    FEAT_FP16 extension.
  • 0b0001 in an implementation with Advanced SIMD support, that includes the FEAT_FP16
    extension.
  • 0b1111 in an implementation without Advanced SIMD support.
In an Armv8-A implementation that supports AArch64, this field returns the value of
ID_AA64PFR0_EL1.AdvSIMD.

FP, bits [19:16]
Floating-point. Defined values are:
0b0000  Floating-point is implemented, and includes support for:
    • Single-precision and double-precision floating-point types.
    • Conversions between single-precision and half-precision data types, and
      double-precision and half-precision data types.
0b0001  As for 0b0000, and also includes support for half-precision floating-point arithmetic.
0b1111  Floating-point is not implemented.
All other values are reserved.
This field must have the same value as the AdvSIMD field.
The permitted values are:
  • 0b0000 in an implementation with floating-point support, that does not include the
    FEAT_FP16 extension.
  • 0b0001 in an implementation with floating-point support, that includes the FEAT_FP16
    extension.
  • 0b1111 in an implementation without floating-point support.
In an Armv8-A implementation that supports AArch64, this field returns the value of
ID_AA64PFR0_EL1.FP.

EL3, bits [15:12]
AArch64 EL3 Exception level handling. Defined values are:
0b0000  EL3 is not implemented or cannot be executed in AArch64 state.
0b0001  EL3 can be executed in AArch64 state only.
0b0010  EL3 can be executed in both Execution states.
When the value of EDAA32PFR.EL3 is non-zero, this field must be 0b0000.
All other values are reserved.
In an Armv8-A implementation that supports AArch64, this field returns the value of
ID_AA64PFR0_EL1.EP.

EL2, bits [11:8]
AArch64 EL2 Exception level handling. Defined values are:
0b0000  EL2 is not implemented or cannot be executed in AArch64 state.
0b0001  EL2 can be executed in AArch64 state only.
0b0010  EL2 can be executed in both Execution states.
When the value of EDAA32PFR.EL2 is non-zero, this field must be 0b0000.
All other values are reserved.
In an Armv8-A implementation that supports AArch64, this field returns the value of ID_AA64PFR0_EL1.EL2.

**EL1, bits [7:4]**

AArch64 EL1 Exception level handling. Defined values are:

- **0b0000**  EL1 cannot be executed in AArch64 state.
  EL1 can be executed in AArch32 state only.
- **0b0001**  EL1 can be executed in AArch64 state only.
- **0b0010**  EL1 can be executed in both Execution states.

All other values are reserved.

In an Armv8-A implementation that supports AArch64, this field returns the value of ID_AA64PFR0_EL1.EL1.

**EL0, bits [3:0]**

AArch64 EL0 Exception level handling. Defined values are:

- **0b0000**  EL0 cannot be executed in AArch64 state.
  EL0 can be executed in AArch32 state only.
- **0b0001**  EL0 can be executed in AArch64 state only.
- **0b0010**  EL0 can be executed in both Execution states.

All other values are reserved.

In an Armv8-A implementation that supports AArch64, this field returns the value of ID_AA64PFR0_EL1.EL0.

**Accessing the EDPFR:**

EDPFR[31:0] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x20</td>
<td>EDPFR</td>
<td>31:0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered() and !DoubleLockStatus() accesses to EDPFR[31:0] are RO.
- Otherwise accesses to EDPFR[31:0] are IMPDEF.

EDPFR[63:32] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x24</td>
<td>EDPFR</td>
<td>63:32</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered() and !DoubleLockStatus() accesses to EDPFR[63:32] are RO.
- Otherwise accesses to EDPFR[63:32] are IMPDEF.
H9.2.35  **EDPIDR0, External Debug Peripheral Identification Register 0**

The EDPIDR0 characteristics are:

**Purpose**

Provides information to identify an external debug component.

For more information, see *About the Peripheral identification scheme* on page K6-11696.

**Configurations**

Implementation of this register is OPTIONAL.

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

**Attributes**

EDPIDR0 is a 32-bit register.

**Field descriptions**

![Diagram of EDPIDR0 register]

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:8</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>7:0</td>
<td>Part number, least significant byte. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.</td>
</tr>
</tbody>
</table>

**Accessing the EDPIDR0:**

EDPIDR0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFE0</td>
<td>EDPIDR0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
H9.2.36  EDPIDR1, External Debug Peripheral Identification Register 1

The EDPIDR1 characteristics are:

**Purpose**

Provides information to identify an external debug component.

For more information, see About the Peripheral identification scheme on page K6-11696.

**Configurations**

Implementation of this register is OPTIONAL.

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

**Attributes**

EDPIDR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>DES_0</td>
<td>PART_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**DES_0, bits [7:4]**

Designer, least significant nibble of JEP106 ID code. For Arm Limited, this field is 0b1011.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**PART_1, bits [3:0]**

Part number, most significant nibble.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Accessing the EDPIDR1:**

EDPIDR1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFE4</td>
<td>EDPIDR1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
H9.2.37   EDPIDR2, External Debug Peripheral Identification Register 2

The EDPIDR2 characteristics are:

**Purpose**

Provides information to identify an external debug component.

For more information, see *About the Peripheral identification scheme* on page K6-11696.

**Configurations**

Implementation of this register is optional.

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

**Attributes**

EDPIDR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>REVISION</td>
<td>1</td>
<td>DES_1</td>
<td>JEDEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**REVISION, bits [7:4]**

Part major revision. Parts can also use this field to extend Part number to 16-bits.

This field has an implementation defined value.

Access to this field is RO.

**JEDEC, bit [3]**

Indicates a JEP106 identity code is used.

Reads as 0b1.

Access to this field is RO.

**DES_1, bits [2:0]**

Designer, most significant bits of JEP106 ID code. For Arm Limited, this field is 0b11.

This field has an implementation defined value.

Access to this field is RO.

**Accessing the EDPIDR2:**

EDPIDR2 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFE8</td>
<td>EDPIDR2</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
• Otherwise accesses to this register generate an error response.
H9.2.38   EDPIDR3, External Debug Peripheral Identification Register 3

The EDPIDR3 characteristics are:

**Purpose**

Provides information to identify an external debug component.

For more information, see *About the Peripheral identification scheme* on page K6-11696.

**Configurations**

Implementation of this register is OPTIONAL.

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

**Attributes**

EDPIDR3 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:8]</th>
<th>Reserved, RES0</th>
</tr>
</thead>
<tbody>
<tr>
<td>REVAND, bits [7:4]</td>
<td>Part minor revision. Parts using EDPIDR2.REVISION as an extension to the Part number must use this field as a major revision number.</td>
</tr>
<tr>
<td>CMOD, bits [3:0]</td>
<td>Customer modified. Indicates someone other than the Designer has modified the component.</td>
</tr>
</tbody>
</table>

**Accessing the EDPIDR3:**

EDPIDR3 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component Offset Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug 0xFEC EDPIDR3</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
H9.2.39 EDPIDR4, External Debug Peripheral Identification Register 4

The EDPIDR4 characteristics are:

**Purpose**

Provides information to identify an external debug component.

For more information, see *About the Peripheral identification scheme on page K6-11696*.

**Configurations**

Implementation of this register is OPTIONAL.

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

**Attributes**

EDPIDR4 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>8</td>
<td>SIZE</td>
<td>Size of the component. Log₂ of the number of 4KB pages from the start of the component to the end of the component ID registers. Reads as $0b0000. Access to this field is RO.</td>
</tr>
<tr>
<td>3</td>
<td>DES_2</td>
<td>Designer, JEP106 continuation code, least significant nibble. For Arm Limited, this field is $0b0100. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.</td>
</tr>
</tbody>
</table>

**Accessing the EDPIDR4:**

EDPIDR4 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0xFD0</td>
<td>EDPIDR4</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
H9.2.40   EDPRCR, External Debug Power/Reset Control Register

The EDPRCR characteristics are:

**Purpose**

Controls the PE functionality related to powerup, reset, and powerdown.

**Configurations**

EDPRCR contains fields that are in the Core power domain and fields that are in the Debug power domain.

If FEAT_DoPD is implemented then all fields in this register are in the Core power domain.

CORENPDRQ is the only field that is mapped between the EDPRCR and DBGPRCR and DBGPRCR_EL1.

**Attributes**

EDPRCR is a 32-bit register.

**Field descriptions**

*When FEAT_DoPD is implemented:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0, reserved</td>
</tr>
<tr>
<td>30-2</td>
<td>CWRR, bit [1]</td>
</tr>
<tr>
<td>1</td>
<td>CORENPDRQ, reserved</td>
</tr>
</tbody>
</table>

**Bits [31:2]**

Reserved, RES0.

**CWRR, bit [1]**

*When FEAT_RME is implemented:*

The PE ignores all writes to this bit.

*Otherwise:*

Warm reset request.

The extent of the reset is IMPLEMENTATION DEFINED, but must be one of:

- The request is ignored.
- Only this PE is Warm reset.
- This PE and other components of the system, possibly including other PEs, are Warm reset.

Arm deprecates use of this bit, and recommends that implementations ignore the request.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>No action</td>
</tr>
<tr>
<td>0b1</td>
<td>Request Warm reset</td>
</tr>
</tbody>
</table>

This field is in the Core power domain.

The PE ignores writes to this bit if any of the following are true:

- ExternalInvasiveDebugEnabled() == FALSE, EL3 is not implemented, and the implemented Security state is Non-secure state.
- ExternalSecureInvasiveDebugEnabled() == FALSE, EL3 is not implemented, and the implemented Security state is Secure state.
- ExternalSecureInvasiveDebugEnabled() == FALSE and EL3 is implemented.

In an implementation that includes the recommended external debug interface, this bit drives the DBGRSTREQ signal.
The reset behavior of this field is:
  • On a Warm reset, this field resets to 0.

Accessing this field has the following behavior:
  • RAZ/WI if any of the following are true:
    — `OSLockStatus()`
    — `SoftwareLockStatus()`
  • Otherwise, access to this field is WO/RAZ

**CORENPDRQ**, bit [0]

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

- 0b0 If the system responds to a powerdown request, it powers down Core power domain.
- 0b1 If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

When this bit reads as UNKNOWN, the PE ignores writes to this bit.

This field is in the Core power domain, and permitted accesses to this field map to the `DBGPRCR.CORENPDRQ` and `DBGPRCR_EL1.CORENPDRQ` fields.

In an implementation that includes the recommended external debug interface, this bit drives the `DBGNOPWRDWN` signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the Cold reset value on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states, see *Core power domain power states on page H6-10336*.

---

**Note**

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

---

The reset behavior of this field is:
  • On a Cold reset, if the powerup request is implemented and the powerup request has been asserted, this field is an IMPLEMENTATION DEFINED choice of 0 or 1. If the powerup request is not asserted, this field is set to 0.

Accessing this field has the following behavior:
  • When `OSLockStatus()`, access to this field is UNKNOWN/WI.
  • When `SoftwareLockStatus()`, access to this field is RO.
  • Otherwise, access to this field is RW

**Otherwise:**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| RES0 | CORENPDRQ | COREPURQ | RES0 | CWRR |

**Bits [31:4]**

Reserved, RES0.
COREPURQ, bit [3]

Core powerup request. Allows a debugger to request that the power controller power up the core, enabling access to the debug register in the Core power domain, and that the power controller emulates powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

0b0 Do not request power up of the Core power domain.
0b1 Request power up of the Core power domain, and emulation of powerdown.

In an implementation that includes the recommended external debug interface, this bit drives the DBGPWRUPREQ signal.

This field is in the Debug power domain and can be read and written when the Core power domain is powered off.

--- Note ---

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:
• On an External debug reset, this field resets to 0.

Accessing this field has the following behavior:
• When SoftwareLockStatus(), access to this field is RO.
• Otherwise, access to this field is RW

Bit [2]

Reserved, RES0.

CWRR, bit [1]

When FEAT_RME is implemented:

The PE ignores all writes to this bit.

Otherwise:

Warm reset request.

The extent of the reset is IMPLEMENTATION DEFINED, but must be one of:
• The request is ignored.
• Only this PE is Warm reset.
• This PE and other components of the system, possibly including other PEs, are Warm reset.

Arm deprecates use of this bit, and recommends that implementations ignore the request.

0b0 No action.
0b1 Request Warm reset.

This field is in the Core power domain.

The PE ignores writes to this bit if any of the following are true:
• ExternalInvasiveDebugEnabled() == FALSE, EL3 is not implemented, and the implemented Security state is Non-secure state.
• ExternalSecureInvasiveDebugEnabled() == FALSE, EL3 is not implemented, and the implemented Security state is Secure state.
• ExternalSecureInvasiveDebugEnabled() == FALSE and EL3 is implemented.

In an implementation that includes the recommended external debug interface, this bit drives the DBGRSTREQ signal.
The reset behavior of this field is:
• On a Warm reset, this field resets to 0.

Accessing this field has the following behavior:
• RAZ/WI if any of the following are true:
  — !IsCorePowered()
  — DoubleLockStatus()
  — OSLockStatus()
  — SoftwareLockStatus()
• Otherwise, access to this field is WO/RAZ

CORENPDRQ, bit [0]

Core no powerdown request. Requests emulation of powerdown.
This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

0b0 If the system responds to a powerdown request, it powers down Core power domain.
0b1 If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

When this bit reads as UNKNOWN, the PE ignores writes to this bit.
This field is in the Core power domain, and permitted accesses to this field map to the DBGPRCR.CORENPDRQ and DBGPRCR_EL1.CORENPDRQ fields.

In an implementation that includes the recommended external debug interface, this bit drives the DBGNOPWRDWN signal.
It is IMPLEMENTATION DEFINED whether this bit is reset to the value of EDPCCR.COREPURQ on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states, see Core power domain power states on page H6-10336.

——- Note ———
Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:
• On a Cold reset, this field resets to the value in EDPCCR.COREPURQ.

Accessing this field has the following behavior:
• UNKNOWN/WI if any of the following are true:
  — !IsCorePowered()
  — DoubleLockStatus()
  — OSLockStatus()
• When SoftwareLockStatus(), access to this field is RO.
• Otherwise, access to this field is RW

Accessing the EDPCCR:

On permitted accesses to the register, other access controls affect the behavior of some fields. See the field descriptions for more information.

EDPCCR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x310</td>
<td>EDPCCR</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When (FEAT_DoPD is not implemented or IsCorePowered()) and SoftwareLockStatus() accesses to this register are RO.
- When (FEAT_DoPD is not implemented or IsCorePowered()) and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.
H9.2.41 EDPRSR, External Debug Processor Status Register

The EDPRSR characteristics are:

**Purpose**

Holds information about the reset and powerdown state of the PE.

**Configurations**

EDPRSR contains fields that are in the Core power domain and fields that are in the Debug power domain.

If FEAT_DoPD is implemented then all fields in this register are in the Core power domain.

**Attributes**

EDPRSR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>30</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>29</td>
<td>EPMADE, bit [16]</td>
</tr>
<tr>
<td>28</td>
<td>ETADE, bit [15]</td>
</tr>
</tbody>
</table>

**EPMADE, bit [16]**

*When FEAT_RME is implemented, FEAT_PMUv3 is implemented and external debugger access to the Performance Monitors Extension registers is implemented:*

External Performance Monitors Access Disable Extended Status. Together with EDPRSRE.PMAD, reports whether access to Performance Monitor registers by an external debugger is permitted.

For a description of the values derived by evaluating PMAD and EPMADE together, see EDPRSRE.PMAD.

*Otherwise:*

Reserved, RES0.

**ETADE, bit [15]**

*When FEAT_RME is implemented, external debugger access to the trace unit registers is implemented and FEAT_TRBE is implemented:*

External Trace Access Disable Extended Status. Together with EDPRSRE.TAD, reports whether access to trace unit registers by an external debugger is permitted.

For a description of the values derived by evaluating TAD and ETADE together, see EDPRSRE.TAD.

*Otherwise:

Reserved, RES0.*
EDADE, bit [14]

When FEAT_RME is implemented:

External Debug Access Disable Extended Status. Together with EDPRSR.EDAD, reports whether access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger is permitted.

For a description of the values derived by evaluating EDAD and EDADE together, see EDPRSR.EDAD.

Otherwise:

Reserved, RES0.

STAD, bit [13]

When external debugger access to the trace unit registers is implemented and FEAT_TRBE is implemented:

Sticky ETAD error. Set to 1 when a Non-secure external debug interface access to an external trace register returns an error because AllowExternalTraceAccess () == FALSE for the access.

0b0 Since EDPRSR was last read, no external accesses to the trace unit registers have failed because AllowExternalTraceAccess () was FALSE for the access.

0b1 Since EDPRSR was last read, at least one external access to the trace unit registers has failed because AllowExternalTraceAccess () was FALSE for the access.

If IsCorePowered () == TRUE, the Core power domain is powered up, then, following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or DoubleLockStatus () == FALSE then this bit clears to 0.
- If FEAT_DoubleLock is implemented and DoubleLockStatus () == TRUE then it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This bit is in the Core power domain.

Note

If FEAT_DoPD is implemented, FEAT_DoubleLock is not implemented.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.

Accessing this field has the following behavior:

- UNKNOWN/WI if any of the following are true:
  - DoubleLockStatus()
  - FEAT_DoPD is not implemented and !IsCorePowered()
  - EDPRSR.R == 1

- Otherwise, access to this field is RC/WI

Otherwise:

Reserved, RES0.
ETAD, bit [12]

When FEAT_RME is implemented, external debugger access to the trace unit registers is implemented and FEAT_TRBE is implemented:

External Trace Access Disable Status. Together with EDPRSR.ETADE, reports whether access to trace unit registers by an external debugger is permitted.

<table>
<thead>
<tr>
<th>ETAEDE</th>
<th>ETAD</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b0</td>
<td>Access to trace unit registers by an external debugger is permitted.</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>Root and Secure access to trace unit registers by an external debugger is permitted. Realm and Non-secure access to trace unit registers by an external debugger is not permitted.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>Root and Realm access to trace unit registers by an external debugger is permitted. Secure and Non-secure access to trace unit registers by an external debugger is not permitted.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>Root access to trace unit registers by an external debugger is permitted. Secure, Non-secure, and Realm access to trace unit registers by an external debugger is not permitted.</td>
</tr>
</tbody>
</table>

When external debugger access to the trace unit registers is implemented and FEAT_TRBE is implemented:

External Trace Access Disable status.

| 0b0 | External Non-secure trace unit accesses enabled. AllowExternalTraceAccess() == TRUE for a Non-secure access. |
| 0b1 | External Non-secure trace unit accesses disabled. AllowExternalTraceAccess() == FALSE for a Non-secure access. |

This bit is in the Core power domain.

Note
If FEAT_DoPD is implemented, FEAT_DoubleLock is not implemented.

Accessing this field has the following behavior:
- UNKNOWN/WI if any of the following are true:
  - DoubleLockStatus()
  - FEAT_DoPD is not implemented and !IsCorePowered()
  - EDPRSR.R == 1
- Otherwise, access to this field is RO

Otherwise:
Reserved, RES0.

SDR, bit [11]

Sticky Debug Restart. Set to 1 when the PE exits Debug state.
Permitted values are:

| 0b0 | The PE has not restarted since EDPRSR was last read. |
| 0b1 | The PE has restarted since EDPRSR was last read. |

Note
If a reset occurs when the PE is in Debug state, the PE exits Debug state. SDR is UNKNOWN on Warm reset, meaning a debugger must also use the SR bit to determine whether the PE has left Debug state.

If the Core power domain is powered up, then following a read of EDPRSR:
- If FEAT_DoubleLock is not implemented or DoubleLockStatus() == FALSE this bit clears to 0.
• If `FEAT_DoubleLock` is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:
• UNKNOWN/WI if any of the following are true:
  — `FEAT_DoPD` is not implemented and !IsCorePowered()
  — `DoubleLockStatus()`
  — `EDPRSR.R == 1`
• When `SoftwareLockStatus()`, access to this field is RO.
• Otherwise, access to this field is RC/WI

**SPMAD, bit [10]**

*When `FEAT_Debugv8p4` is implemented, `FEAT_PMUv3` is implemented and external debugger access to the Performance Monitors Extension registers is implemented:*

Sticky EPMAD error. Set to 1 if an external debug interface access to a Performance Monitors register returns an error because `AllowExternalPMUAccess()` == FALSE.

Permitted values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>No Non-secure external debug interface accesses to the external Performance Monitors registers have failed because <code>AllowExternalPMUAccess()</code> == FALSE for the access since <code>EDPRSR</code> was last read.</td>
</tr>
<tr>
<td>0b1</td>
<td>At least one Non-secure external debug interface access to the external Performance Monitors register has failed and returned an error because <code>AllowExternalPMUAccess()</code> == FALSE for the access since <code>EDPRSR</code> was last read.</td>
</tr>
</tbody>
</table>

If the Core power domain is powered up, then following a read of `EDPRSR`:
• If `FEAT_DoubleLock` is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
• If `FEAT_DoubleLock` is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:
• On a Cold reset, this field resets to 0.

Accessing this field has the following behavior:
• UNKNOWN/WI if any of the following are true:
  — `FEAT_DoPD` is not implemented and !IsCorePowered()
  — `DoubleLockStatus()`
  — `EDPRSR.R == 1`
• When `SoftwareLockStatus()`, access to this field is RO.
• Otherwise, access to this field is RC/WI

*When `FEAT_PMUv3` is implemented and external debugger access to the Performance Monitors Extension registers is implemented:*

Sticky EPMAD error.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>No external debug interface accesses to the Performance Monitors registers have failed because <code>AllowExternalPMUAccess()</code> == FALSE since <code>EDPRSR</code> was last read.</td>
</tr>
<tr>
<td>0b1</td>
<td>At least one external debug interface access to the Performance Monitors registers has failed and returned an error because <code>AllowExternalPMUAccess()</code> == FALSE since <code>EDPRSR</code> was last read.</td>
</tr>
</tbody>
</table>
If the Core power domain is powered up, then, following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or DoubleLockStatus() == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented, and DoubleLockStatus() == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.

Accessing this field has the following behavior:

- UNKNOWN/WI if any of the following are true:
  - FEAT_DoPD is not implemented and !IsCorePowered()
  - OSLockStatus()
  - DoubleLockStatus()
  - EDPRSR.R == 1
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RC/WI

**Otherwise:**

Reserved, RES0.

**EPMAD, bit [9]**

*When FEAT_RME is implemented, FEAT_PMUv3 is implemented and external debugger access to the Performance Monitors Extension registers is implemented:*

External Performance Monitors Access Disable Status. Together with EDPRSR.EPMADE, reports whether access to Performance Monitor registers by an external debugger is permitted.

<table>
<thead>
<tr>
<th>EPMAD</th>
<th>EPMAD</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b0</td>
<td>Access to Performance Monitor registers by an external debugger is permitted.</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>Root and Secure access to Performance Monitor registers by an external debugger is permitted. Realm and Non-secure access to Performance Monitor registers by an external debugger is not permitted.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>Root and Realm access to Performance Monitor registers by an external debugger is permitted. Secure and Non-secure access to Performance Monitor registers by an external debugger is not permitted.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>Root access to Performance Monitor registers by an external debugger is permitted. Secure, Non-secure, and Realm access to Performance Monitor registers by an external debugger is not permitted.</td>
</tr>
</tbody>
</table>

*When FEAT_Debugv8p4 is implemented, FEAT_PMUv3 is implemented and external debugger access to the Performance Monitors Extension registers is implemented:*

External Performance Monitors Non-secure Access Disable status.

- 0b0 External Non-secure Performance Monitors access enabled. AllowExternalPMUAccess() == TRUE for a Non-secure access.
- 0b1 External Non-secure Performance Monitors access disabled. AllowExternalPMUAccess() == FALSE for a Non-secure access.

This field is in the Core power domain.

Accessing this field has the following behavior:

- UNKNOWN/WI if any of the following are true:
  - FEAT_DoPD is not implemented and !IsCorePowered()
— DoubleLockStatus()
— EDPRS.R == 1

• Otherwise, access to this field is RO

When FEAT_PMUv3 is implemented, external debugger access to the Performance Monitors Extension registers is implemented and FEAT_Debugv8p4 is not implemented:

External Performance Monitors access disable status.

0b0 External Performance Monitors access enabled. AllowExternalPMUAccess() == TRUE.
0b1 External Performance Monitors access disabled. AllowExternalPMUAccess() == FALSE.

This field is in the Core power domain.

Accessing this field has the following behavior:

• UNKNOW/NWI if any of the following are true:
  — FEAT_DoPD is not implemented and !IsCorePowered()
  — OSI LockStatus()
  — DoubleLockStatus()
  — EDPRS.R == 1

• Otherwise, access to this field is RO

Otherwise:

Reserved, RES0.

SDAD, bit [8]

When FEAT_Debugv8p4 is implemented:

Sticky EDAD error. Set to 1 if an external debug interface access to a debug register returns an error because AllowExternalDebugAccess() == FALSE.

0b0 No Non-secure external debug interface accesses to the debug registers have failed because AllowExternalDebugAccess() == FALSE for the access since EDPRS.R was last read.
0b1 At least one Non-secure external debug interface access to the debug registers has failed and returned an error because AllowExternalDebugAccess() == FALSE for the access since EDPRS.R was last read.

If the Core power domain is powered up, then, following a read of EDPRS.R:

• If FEAT_DoubleLock is not implemented or DoubleLockStatus() == FALSE this bit clears to 0.
• If FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

• On a Cold reset, this field resets to 0.

Accessing this field has the following behavior:

• UNKNOWN/WI if any of the following are true:
  — FEAT_DoPD is not implemented and !IsCorePowered()
  — DoubleLockStatus()
  — EDPRS.R == 1

• Otherwise, access to this field is RO

Otherwise:

Sticky EDAD error. Set to 1 if an external debug interface access to a debug register returns an error because AllowExternalDebugAccess() == FALSE.

0b0 No external debug interface accesses to the debug registers have failed because AllowExternalDebugAccess() == FALSE since EDPRS.R was last read.
At least one external debug interface access to the debug registers has failed and returned an error because AllowExternalDebugAccess() == FALSE since EDPRSR was last read.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or DoubleLockStatus() == FALSE this bit clears to 0.
- If FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This bit is UNKNOWN on reads if OSLockStatus() == TRUE and external debug writes to OSLAR_EL1 do not return an error when AllowExternalDebugAccess() == FALSE.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.

Accessing this field has the following behavior:

- UNKNOWN/WI if any of the following are true:
  - FEAT_DoPD is not implemented and !IsCorePowered()
  - DoubleLockStatus()
  - EDPRSR.R == 1
- Otherwise, access to this field is RO

**EDAD, bit [7]**

*When FEAT_RME is implemented:*

External Debug Access Disable Status. Together with EDPRSR.EDADE, reports whether access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger is permitted.

<table>
<thead>
<tr>
<th>EDADE</th>
<th>EDAD</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b0</td>
<td>Access to Debug registers by an external debugger is permitted.</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>Root and Secure access to Debug registers by an external debugger is permitted. Realm and Non-secure access to Debug registers by an external debugger is not permitted.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>Root and Realm access to Debug registers by an external debugger is permitted. Secure and Non-secure access to Debug registers by an external debugger is not permitted.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>Root access to Debug registers by an external debugger is permitted. Secure, Non-secure, and Realm access to Debug registers by an external debugger is not permitted.</td>
</tr>
</tbody>
</table>

*When FEAT_Debugv8p4 is implemented:*

External Debug Access Disable status.

- 0b0: External Non-secure access to breakpoint registers, watchpoint registers, and OSLAR_EL1 enabled. AllowExternalDebugAccess() == TRUE for a Non-secure access.
- 0b1: External Non-secure access to breakpoint registers, watchpoint registers, and OSLAR_EL1 disabled. AllowExternalDebugAccess() == FALSE for a Non-secure access.

This field is in the Core power domain.

Accessing this field has the following behavior:

- UNKNOWN/WI if any of the following are true:
  - FEAT_DoPD is not implemented and !IsCorePowered()
  - DoubleLockStatus()
  - EDPRSR.R == 1
- Otherwise, access to this field is RO
When FEAT_Debugv8p2 is implemented:

External Debug Access Disable status.

0b0  External access to breakpoint registers, watchpoint registers, and OSLAR_EL1 enabled. AllowExternalDebugAccess() == TRUE.

0b1  External access to breakpoint registers, watchpoint registers, and OSLAR_EL1 disabled. AllowExternalDebugAccess() == FALSE.

This bit is not valid and reads UNKNOWN if OSLockStatus() == TRUE and external debug writes to OSLAR_EL1 do not return an error when AllowExternalDebugAccess() == FALSE.

This field is in the Core power domain.

Accessing this field has the following behavior:

- UNKNOWN/WI if any of the following are true:
  - FEAT_DoPD is not implemented and !IsCorePowered()
  - DoubleLockStatus()
  - EDPRSR.R == 1
- Otherwise, access to this field is RO

Otherwise:

External Debug Access Disable status.

0b0  External access to breakpoint registers, watchpoint registers, and OSLAR_EL1 enabled. AllowExternalDebugAccess() == TRUE.

0b1  External access to breakpoint registers, watchpoint registers disabled. It is IMPLEMENTATION DEFINED whether accesses to OSLAR_EL1 are enabled or disabled. AllowExternalDebugAccess() == FALSE.

This field is in the Core power domain.

Accessing this field has the following behavior:

- UNKNOWN/WI if any of the following are true:
  - FEAT_DoPD is not implemented and !IsCorePowered()
  - DoubleLockStatus()
  - EDPRSR.R == 1
- Otherwise, access to this field is RO

DLK, bit [6]

When FEAT_Debugv8p4 is implemented:

This field is RES0.

When FEAT_Debugv8p2 is implemented and FEAT_DoubleLock is implemented:

Double Lock.

From Armv8.2, this field is deprecated.

This field is in the Core power domain.

Accessing this field has the following behavior:

- RAZ/WI if all of the following are true:
  - IsCorePowered()
  - !DoubleLockStatus()
- Otherwise, access to this field is UNKNOWN/WI

When FEAT_DoubleLock is implemented:

Double Lock.

This field returns the result of the pseudocode function DoubleLockStatus().

If the Core power domain is powered up and DoubleLockStatus() == TRUE, it is IMPLEMENTATION DEFINED whether:

- EDPRSR.PU reads as 1, EDPRSR.DLK reads as 1, and EDPRSR.SPD is UNKNOWN.
• EDPRSR.PU reads as 0, EDPRSR.DLK is UNKNOWN, and EDPRSR.SPD reads as 0.
  This field is in the Core power domain.
  0b0   DoubleLockStatus() returns FALSE.
  0b1   DoubleLockStatus() returns TRUE and the Core power domain is powered up.
  Accessing this field has the following behavior:
  • UNKNOWN/WI if all of the following are true:
    — FEAT_DoPD is not implemented
    — !IsCorePowered()
    — EDPRSR.R == 1
  • Otherwise, access to this field is RO

Otherwise:
  Reserved, RES0.

OSLK, bit [5]
  OS Lock status bit.
  A read of this bit returns the value of OSLSR_EL1.OSLK.
  This field is in the Core power domain.
  Accessing this field has the following behavior:
  • UNKNOWN/WI if all of the following are true:
    — FEAT_DoPD is not implemented
    — !IsCorePowered()
    — DoubleLockStatus()
  • Otherwise, access to this field is RO

HALTED, bit [4]
  Halted status bit.
  0b0   PE is in Non-debug state.
  0b1   PE is in Debug state.
  This field is in the Core power domain.
  Accessing this field has the following behavior:
  • UNKNOWN/WI if all of the following are true:
    — FEAT_DoPD is not implemented
    — !IsCorePowered()
  • Otherwise, access to this field is RO

SR, bit [3]
  Sticky core Reset status bit.
  Permitted values are:
  0b0   The non-debug logic of the PE is not in reset state and has not been reset since the last
time EDPRSR was read.
  0b1   The non-debug logic of the PE is in reset state or has been reset since the last time
EDPRSR was read.
  If EDPRSR.PU reads as 1 and EDPRSR.R reads as 0, which means that the Core power domain is
in a powerup state and that the non-debug logic of the PE is not in reset state, then following a read
of EDPRSR:
  • If FEAT_DoubleLock is not implemented or DoubleLockStatus() == FALSE this bit clears to
  0.
  • If FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE, it is CONSTRAINED
  UNPREDICTABLE whether this bit clears to 0 or is unchanged.
This field is in the Core power domain.
The reset behavior of this field is:
• On a Warm reset, this field resets to 1.
Accessing this field has the following behavior:
• UNKNOWN/WI if any of the following are true:
  — FEAT_DoPD is not implemented and !IsCorePowered()
  — DoubleLockStatus()
• When SoftwareLockStatus(), access to this field is RO.
• Otherwise, access to this field is RC/WI

R, bit [2]
PE Reset status bit.
Permitted values are:
0b0 The non-debug logic of the PE is not in reset state.
0b1 The non-debug logic of the PE is in reset state.
If FEAT_DoubleLock is implemented, the PE is in reset state, and the PE entered reset state with
the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more
information, see EDPRSR.{DLK, R} and reset state on page H6-10343.
This field is in the Core power domain.
Accessing this field has the following behavior:
• UNKNOWN/WI if any of the following are true:
  — FEAT_DoPD is not implemented and !IsCorePowered()
  — DoubleLockStatus()
• Otherwise, access to this field is RO

SPD, bit [1]
Sticky core Powerdown status bit.
If FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE, then:
• If FEAT_Debugv8p2 is implemented, this bit reads as 0.
• If FEAT_Debugv8p2 is not implemented, this bit might read as 0 or 1.
For more information, see EDPRSR.{DLK, SPD, PU} and the Core power domain on page H6-10342.
0b0 If EDPRSR.PU is 0, it is not known whether the state of the debug registers in the Core
power domain is lost.
  If EDPRSR.PU is 1, the state of the debug registers in the Core power domain has not
been lost.
0b1 The state of the debug registers in the Core power domain has been lost.
If the Core power domain is powered up, then, following a read of EDPRSR:
• If FEAT_DoubleLock is not implemented or DoubleLockStatus() == FALSE this bit clears to
  0.
• If FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE, it is CONSTRAINED
  UNPREDICTABLE whether this bit clears to 0 or is unchanged.
When FEAT_DoPD is not implemented and the Core power domain is in either retention or
powerdown state, the value of EDPRSR.SPD is IMPLEMENTATION DEFINED. For more information,
see EDPRSR.SPD when the Core domain is in either retention or powerdown state on page H6-10343.
EDPRSR.{DLK, SPD, PU} describe whether registers in the Core power domain can be accessed,
and whether their state has been lost since the last time the register was read. For more information,
see EDPRSR.{DLK, SPD, PU} and the Core power domain on page H6-10342.
This field is in the Core power domain.
The reset behavior of this field is:

- On a Cold reset, this field resets to 1.

Accessing this field has the following behavior:

- RAZ/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- UNKNOWN/WI if all of the following are true:
  - !IsCorePowered()
  - DoubleLockStatus()
- Otherwise, access to this field is RO

**PU, bit [0]**

*When FEAT_DoPD is implemented:*

Core powerup status bit.
Access to this field is RAO/WI.

*When FEAT_Debugv8p2 is implemented:*

Core Powerup status bit. Indicates whether the debug registers in the Core power domain can be accessed.

- 0b0  Either the Core power domain is in a low-power or powerdown state, or
  FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE, meaning the debug registers in the Core power domain cannot be accessed.

- 0b1  The Core power domain is in a powerup state, and either FEAT_DoubleLock is not implemented or DoubleLockStatus() == FALSE, meaning the debug registers in the Core power domain can be accessed.

If FEAT_DoubleLock is implemented, the PE is in reset state, and the PE entered reset state with the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more information, see EDPRSR.{DLK, R} and reset state on page H6-10343

EDPRSR.{DLK, SPD, PU} describe whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read. For more information, see EDPRSR.{DLK, SPD, PU} and the Core power domain on page H6-10342

Access to this field is RO.

*Otherwise:*

Core Powerup status bit. Indicates whether the debug registers in the Core power domain can be accessed.

When the Core power domain is powered-up and DoubleLockStatus() == TRUE, then the value of EDPRSR.PU is IMPLEMENTATION DEFINED. See the description of the DLK bit for more information.

Otherwise, permitted values are:

- 0b0  Core power domain is in a low-power or powerdown state where the debug registers in the Core power domain cannot be accessed.

- 0b1  Core power domain is in a powerup state where the debug registers in the Core power domain can be accessed.

If FEAT_DoubleLock is implemented, the Core power domain is powered up, and DoubleLockStatus() == TRUE, it is IMPLEMENTATION DEFINED whether this bit reads as 0 or 1.

If FEAT_DoubleLock is implemented, the PE is in reset state, and the PE entered reset state with the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more information see EDPRSR.{DLK, R} and reset state on page H6-10343

EDPRSR.{DLK, SPD, PU} describe whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read. For more information, see EDPRSR.{DLK, SPD, PU} and the Core power domain on page H6-10342.
Access to this field is RO.

**Accessing the EDPRSR:**

On permitted accesses to the register, other access controls affect the behavior of some fields. See the field descriptions for more information.

If the Core power domain is powered up (EDPRSR.PU == 1), then following a read of EDPRSR:

- If **FEAT_DoubleLock** is not implemented or **DoubleLockStatus() == FALSE**, then:
  - EDPRSR.{SDR, SPMAD, SDAD, SPD} are cleared to 0.
  - EDPRSR.SR is cleared to 0 if the non-debug logic of the PE is not in reset state (EDPRSR.R == 0).

- If **FEAT_DoubleLock** is implemented and **DoubleLockStatus() == TRUE**, it is **CONSTRAINED UNPREDICTABLE** whether or not this clearing occurs.

If FEAT_DoPD is not implemented and the Core power domain is powered down (EDPRSR.PU == 0), then:

- EDPRSR.{SDR, SPMAD, SDAD, SR} are all **UNKNOWN**, and are either reset or restored on being powered up.

- EDPRSR.SPD is not cleared following a read of EDPRSR. See the SPD bit description for more information.

The clearing of bits is an indirect write to EDPRSR.

EDPRSR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x314</td>
<td>EDPRSR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.

- Otherwise accesses to this register generate an error response.
H9.2.42  EDRCR, External Debug Reserve Control Register

The EDRCR characteristics are:

**Purpose**

This register is used to allow imprecise entry to Debug state and clear sticky bits in EDSCR.

**Configurations**

EDRCR is in the Core power domain.

**Attributes**

EDRCR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:5]</td>
<td>Reserved, RES0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4]</td>
<td>CBRRQ, bit [4]</td>
<td></td>
<td>Allow imprecise entry to Debug state. The actions on writing to this bit are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0</td>
<td>No action.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b1</td>
<td>Allow imprecise entry to Debug state, for example by canceling pending bus accesses.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Setting this bit to 1 allows a debugger to request imprecise entry to Debug state. An External Debug Request debug event must be pending before the debugger sets this bit to 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This feature is optional. If this feature is not implemented, writes to this bit are ignored.</td>
</tr>
<tr>
<td>[3]</td>
<td>CSPA, bit [3]</td>
<td></td>
<td>Clear Sticky Pipeline Advance. This bit is used to clear the EDSCR.PipeAdv bit to 0. The actions on writing to this bit are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0</td>
<td>No action.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b1</td>
<td>Clear the EDSCR.PipeAdv bit to 0.</td>
</tr>
<tr>
<td>[2]</td>
<td>CSE, bit [2]</td>
<td></td>
<td>Clear Sticky Error. Used to clear the EDSCR cumulative error bits to 0. The actions on writing to this bit are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0</td>
<td>No action.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b1</td>
<td>Clear the EDSCR.{TXU, RXO, ERR} bits, and, if the PE is in Debug state, the EDSCR.ITO bit, to 0.</td>
</tr>
<tr>
<td>[1:0]</td>
<td>Reserved, RES0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Accessing the EDRCR:

EDCR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x090</td>
<td>EDR CR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are WI.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are WO.
- Otherwise accesses to this register generate an error response.
H9.2.43  EDSCR, External Debug Status and Control Register

The EDSCR characteristics are:

**Purpose**

Main control register for the debug implementation.

**Configurations**

External register EDSCR bits [30:29] are architecturally mapped to AArch64 System register MDCCSR_EL0[30:29].

EDSCR is in the Core power domain.

**Attributes**

EDSCR is a 32-bit register.

**Field descriptions**

TFO, bit [31]

*When FEAT_TRF is implemented:*

Trace Filter Override. Overrides the Trace Filter controls allowing the external debugger to trace any visible Exception level.

- **0b0** Trace Filter controls are not affected.
- **0b1** Trace Filter controls in TRFCR_EL1 and TRFCR_EL2 are ignored.
  Trace Filter controls TRFCR and HTRFCR are ignored.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through the following System registers:

- MDSCR_EL1.
- DBGDSCRExt.

This bit is ignored by the PE when any of the following is true:

- ExternalSecureNoninvasiveDebugEnabled() is FALSE and the Effective value of MDCR_EL3.STE is 1.
- FEAT_RME is implemented, ExternalRealmNoninvasiveDebugEnabled() is FALSE, and the Effective value of MDCR_EL3.RLTE is 1.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.

**Otherwise:**

Reserved, RES0.

RXfull, bit [30]

DTRRX full.
The reset behavior of this field is:
- On a Cold reset, this field resets to 0.
Access to this field is RO.

**TXfull, bit [29]**

DTRTX full.
The reset behavior of this field is:
- On a Cold reset, this field resets to 0.
Access to this field is RO.

**ITO, bit [28]**

ITR overrun. Set to 0 on entry to Debug state.
Accessing this field has the following behavior:
- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO

**RXO, bit [27]**

DTRRX overrun.
The reset behavior of this field is:
- On a Cold reset, this field resets to 0.
Access to this field is RO.

**TXU, bit [26]**

DTRTX underrun.
The reset behavior of this field is:
- On a Cold reset, this field resets to 0.
Access to this field is RO.

**PipeAdv, bit [25]**

Pipeline Advance. Indicates that software execution is progressing.

0b0  No progress has been made by the PE since the last time this field was cleared to zero by writing 1 to EDRCR.CSPA.

0b1  Progress has been made by the PE since the last time this field was cleared to zero by writing 1 to EDRCR.CSPA.

The architecture does not define precisely when this field is set to 1. It requires only that this happen periodically in Non-debug state to indicate that software execution is progressing. For example, a PE might set this field to 1 each time the PE retires one or more instructions, or at periodic intervals during the progression of an instruction.

When FEAT_MOPS is implemented, CPY, CPYF, SET, and SETG are examples of instructions that periodically make forward progress.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Access to this field is RO.

**ITE, bit [24]**

ITR empty.
Accessing this field has the following behavior:
- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO
INTdis, bits [23:22]

When FEAT_RME is implemented:

Interrupt disable. Disables taking interrupts in Non-debug state.

0b00  This bit has no effect on the masking of interrupts.
0b01  If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts taken to Non-secure state are masked.
      If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts taken to Secure state are masked.
      If ExternalRootInvasiveDebugEnabled() is TRUE, then all interrupts taken to Root state are masked.
      If ExternalRealmInvasiveDebugEnabled() is TRUE, then all interrupts taken to Realm state are masked.

——— Note ————
All interrupts includes virtual and SError interrupts.

When OSLSR_EL1.OSLK is 1, this field can be indirectly read and written through the following System registers:

• MDSCR_EL1.
• DBGDSCRext.

The Effective value of this field is 0b00 when ExternalInvasiveDebugEnabled() is FALSE.

When FEAT_RME is implemented, bit[23] of this register is RES0.

The reset behavior of this field is:

• On a Cold reset, this field resets to 0.

When FEAT_Debugv8p4 is implemented:

Interrupt disable. Disables taking interrupts in Non-debug state.

0b00  Masking of interrupts is controlled by PSTATE and interrupt routing controls.
0b01  If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts taken to Non-secure state are masked.
      If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts taken to Secure state are masked.

——— Note ————
All interrupts includes virtual and SError interrupts.

When OSLSR_EL1.OSLK is 1, this field can be indirectly read and written through the following System registers:

• MDSCR_EL1.
• DBGDSCRext.

The Effective value of this field is 0b00 when ExternalInvasiveDebugEnabled() is FALSE.

When FEAT_Debugv8p4 is implemented, bit[23] of this register is RES0.

The reset behavior of this field is:

• On a Cold reset, this field resets to 0.

Otherwise:

Interrupt disable. Disables taking interrupts in Non-debug state.

0b00  Masking of interrupts is controlled by PSTATE and interrupt routing controls.
0b01  If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts taken to Non-secure EL1 are masked.
0b10  If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts taken to Non-secure state are masked.
If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts taken to Secure EL1 are masked.

0b11  If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts taken to Non-secure state are masked.
If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts taken to Secure state are masked.

**Note**
All interrupts includes virtual and SError interrupts.

When OSLSR_EL1.OSLK is 1, this field can be indirectly read and written through the following System registers:
- MDSCR_EL1.
- DBGDSCRext.

The Effective value of this field is 0b00 when ExternalInvasiveDebugEnabled() is FALSE.

Support for the values 0b01 and 0b10 is IMPLEMENTATION DEFINED. If these values are not supported, they are reserved. If programmed with a reserved value, the PE behaves as if INTdis has been programmed with a defined value, other than for a direct read of EDSCR, and the value returned by a read of EDSCR.INTdis is UNKNOWN.

The reset behavior of this field is:
- On a Cold reset, this field resets to 0.

**TDA, bit [21]**
Traps accesses to the following debug System registers:
- AArch64: DBGBCR<\n>_EL1, DBGBVR<\n>_EL1, DBGWCR<\n>_EL1, DBGWVR<\n>_EL1.
- AArch32: DBGBCR<\n>, DBGBVR<\n>, DBGBXVR<\n>, DBGWCR<\n>, DBGWVR<\n>.

0b0  Accesses to debug System registers do not generate a Software Access Debug event.
0b1  Accesses to debug System registers generate a Software Access Debug event, if OSLSR_EL1.OSLK is 0 and if halting is allowed.

The reset behavior of this field is:
- On a Cold reset, this field resets to 0.

**MA, bit [20]**
Memory access mode. Controls the use of memory-access mode for accessing ITR and the DCC. This bit is ignored if in Non-debug state and set to zero on entry to Debug state.

Possible values of this field are:
- 0b0  Normal access mode.
- 0b1  Memory access mode.

The reset behavior of this field is:
- On a Cold reset, this field resets to 0.

**SC2, bit [19]**
*When FEAT_PCSRv8 is implemented, (FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented) and FEAT_PCSRv8p2 is not implemented:*
Sample CONTEXTIDR_EL2. Controls whether the PC Sample-based Profiling Extension samples CONTEXTIDR_EL2 or VTTBR_EL2.VMID.

0b0  Sample VTTBR_EL2.VMID.
0b1  Sample CONTEXTIDR_EL2.
The reset behavior of this field is:
  • On a Cold reset, this field resets to 0.

**Otherwise:**
Reserved, RES0.

**NS, bit [18]**

*When FEAT_RME is implemented:*
Non-secure status. Together with the NSE field, gives the current Security state:

<table>
<thead>
<tr>
<th>NSE</th>
<th>NS</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b0</td>
<td>Secure.</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>Non-secure.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>Root.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>Realm.</td>
</tr>
</tbody>
</table>

Accessing this field has the following behavior:
  • When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
  • Otherwise, access to this field is RO

**Otherwise:**
Non-secure status. In Debug state, gives the current Security state:
0b0 Secure state.
0b1 Non-secure state.

Accessing this field has the following behavior:
  • When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
  • Otherwise, access to this field is RO

**Bit [17]**
Reserved, RES0.

**SDD, bit [16]**

*When FEAT_RME is implemented:*
EL3 debug disabled.
On entry to Debug state:
  • If entering from EL3, SDD is set to 0.
  • Otherwise, SDD is set to the inverse of ExternalRootInvasiveDebugEnabled ()

In Debug state, the value of SDD does not change, even if ExternalRootInvasiveDebugEnabled () changes.
In Non-debug state, SDD returns the inverse of ExternalRootInvasiveDebugEnabled ().

Access to this field is RO.

**Otherwise:**
Secure debug disabled.
On entry to Debug state:
  • If entering in Secure state, SDD is set to 0.
  • If entering in Non-secure state, SDD is set to the inverse of ExternalSecureInvasiveDebugEnabled ().
In Debug state, the value of the SDD bit does not change, even if ExternalSecureInvasiveDebugEnabled() changes.

In Non-debug state:
- SDD returns the inverse of ExternalSecureInvasiveDebugEnabled(). If the authentication signals that control ExternalSecureInvasiveDebugEnabled() change, a context synchronization event is required to guarantee their effect.
- This bit is unaffected by the Security state of the PE.

If EL3 is not implemented and the implementation is Non-secure, this bit is RES1.

Access to this field is RO.

NSE, bit [15]

*When FEAT_RME is implemented:*
Together with the NS field, this field gives the current Security state.
For a description of the values derived by evaluating NS and NSE together, see EDSCR.NS.

In Non-debug state, this bit is UNKNOWN.
Access to this field is RO.

*Otherwise:*
Reserved, RES0.

HDE, bit [14]
Halting debug enable.
0b0 Halting disabled for Breakpoint, Watchpoint and Halt Instruction debug events.
0b1 Halting enabled for Breakpoint, Watchpoint and Halt Instruction debug events.
The reset behavior of this field is:
- On a Cold reset, this field resets to 0.

RW, bits [13:10]
Exception level Execution state status. In Debug state, each bit gives the current Execution state of each Exception level.

0b1111 Any of the following:
- The PE is in Non-debug state.
- The PE is at EL0 using AArch64.
- The PE is not at EL0, and EL1, EL2, and EL3 are using AArch64.

0b1110 When AArch32 is supported:
The PE is in Debug state at EL0. EL0 is using AArch32. EL1, EL2, and EL3 are using AArch64.

0b110x When AArch32 is supported and EL2 is implemented:
The PE is in Debug state. EL0 and EL1 are using AArch32. EL2 is enabled in the current Security state and is using AArch64. If implemented, EL3 is using AArch64.

0b10xx When AArch32 is supported and EL3 is implemented:
The PE is in Debug state. EL0 and EL1 are using AArch32. EL2 is not implemented, disabled in the current Security state, or using AArch32. EL3 is using AArch64.

0b0xxx When AArch32 is supported:
The PE is in Debug state. All Exception levels are using AArch32.

Accessing this field has the following behavior:
- When the PE is in Non-debug state, access to this field is RAO/WI.
- Otherwise, access to this field is RO.
EL, bits [9:8]

Exception level. In Debug state, gives the current Exception level of the PE.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is RAZ/WI.
- Otherwise, access to this field is RO

A, bit [7]

SError interrupt pending. In Debug state, indicates whether an SError interrupt is pending:

- If $HCR_{EL2}.\{AMO, TGE\} = \{1, 0\}$, EL2 is enabled in the current Security state, and the PE is executing at EL0 or EL1, a virtual SError interrupt.
- Otherwise, a physical SError interrupt.

0b0 No SError interrupt pending.
0b1 SError interrupt pending.

A debugger can read EDSCR to check whether an SError interrupt is pending without having to execute further instructions. A pending SError might indicate data from target memory is corrupted.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO

ERR, bit [6]

Cumulative error flag. This bit is set to 1 following exceptions in Debug state and on any signaled overrun or underrun on the DTR or EDITR.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.

Access to this field is RO.

STATUS, bits [5:0]

Debug status flags.

0b000001 PE is restarting, exiting Debug state.
0b000010 PE is in Non-debug state.
0b000111 Breakpoint.
0b010011 External debug request.
0b011011 Halting step, normal.
0b011111 Halting step, exclusive.
0b100011 OS Unlock Catch.
0b101111 HLT instruction.
0b110011 Reset Catch.
0b101011 Watchpoint.
0b111111 Software access to debug register.
0b110111 Exception Catch.
0b111011 Halting step, no syndrome.

All other values of STATUS are reserved.

Access to this field is RO.
**Accessing the EDSCR:**

EDSCR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x88</td>
<td>EDSCR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.
EDVIDSR, External Debug Virtual Context Sample Register

The EDVIDSR characteristics are:

**Purpose**
Contains sampled values captured on reading EDPCSR[31:0].

**Configurations**
EDVIDSR is in the Core power domain.
This register is present only when FEAT_PCSRv8 is implemented and FEAT_PCSRv8p2 is not implemented. Otherwise, direct accesses to EDVIDSR are RES0.
If FEAT_VHE is implemented, the format of this register differs depending on the value of EDSCR.SC2.
Implemented only if the OPTIONAL PC Sample-based Profiling Extension is implemented in the external debug registers space.
When the PC Sample-based Profiling Extension is implemented in the external debug registers space, if EL2 is not implemented and EL3 is not implemented, it is IMPLEMENTATION DEFINED whether EDVIDSR is implemented.

**Note**
FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors registers space.

**Attributes**
EDVIDSR is a 32-bit register.

**Field descriptions**

*When FEAT_VHE is not implemented or EDSCR.SC2 == 0:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>NS</td>
<td>Non-secure state sample. Indicates the Security state associated with the most recent EDPCSR sample.</td>
</tr>
<tr>
<td>30</td>
<td>EZ</td>
<td>Non-secure state. Indicates whether the most recent EDPCSR sample was associated with EL2.</td>
</tr>
<tr>
<td>29</td>
<td>E3</td>
<td>Non-secure state. Indicates whether the most recent EDPCSR sample was associated with EL2.</td>
</tr>
<tr>
<td>28</td>
<td>HV</td>
<td>High Voltage</td>
</tr>
<tr>
<td>27</td>
<td>RES0</td>
<td>Reserved</td>
</tr>
<tr>
<td>16</td>
<td>VMID[15:8]</td>
<td>VMID</td>
</tr>
<tr>
<td>15</td>
<td>VMID</td>
<td>VMID</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This format applies in all Armv8.0 implementations.

**NS, bit [31]**
Non-secure state sample. Indicates the Security state associated with the most recent EDPCSR sample.
If EL3 is not implemented, this bit indicates the Effective value of SCR.NS.

*0b0* Sample is from Secure state.

*0b1* Sample is from Non-secure state.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**E2, bit [30]**

*When EL2 is implemented:*
Exception level 2 status sample. Indicates whether the most recent EDPCSR sample was associated with EL2.

*0b0* Sample is not from EL2.

*0b1* Sample is from EL2.
The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**E3, bit [29]**

*When EL3 is implemented and AArch64 is supported:*

Exception level 3 status sample. Indicates whether the most recent EDPCSR sample was associated with EL3 using AArch64.

- 0b0 Sample is not from EL3 using AArch64.
- 0b1 Sample is from EL3 using AArch64.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**HV, bit [28]**

EDPCSRhi (EDPCSR[63:32]) valid. Indicates whether bits [63:32] of the most recent EDPCSR sample might be nonzero:

- 0b0 Bits[63:32] of the most recent EDPCSR sample are zero.
- 0b1 Bits[63:32] of the most recent EDPCSR sample might be nonzero.

An EDVIDSR.HV value of 1 does not mean that the value of EDPCSRhi is nonzero. An EDVIDSR.HV value of 0 is a hint that EDPCSRhi (EDPCSR[63:32]) does not need to be read.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Bits [27:16]**

Reserved, RES0.

**VMID[15:8], bits [15:8]**

*When FEAT_VMID16 is implemented and EL2 is implemented:*

Extension to VMID[7:0]. For more information, see VMID[7:0].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.

**VMID, bits [7:0]**

*When EL2 is implemented:*

VMID sample. The VMID associated with the most recent EDPCSRlo (EDPCSR[31:0]) sample. When the most recent EDPCSR sample was generated:

- This field is RES0 if any of the following apply:
  - The PE is executing in Secure state.
  - The PE is executing at EL2.
- Otherwise:
  - If EL2 is using AArch64 and either FEAT_VMID16 is not implemented or VTCR_EL2.VS is 1, this field is set to VTTBR_EL2.VMID.
— If EL2 is using AArch64, `FEAT_VMID16` is implemented, and `VTCR_EL2.VS` is 0, `PMVIDSR.VMID[7:0]` is set to `VTTBR_EL2.VMID[7:0]` and `PMVIDSR.VMID[15:8]` is `RES0`.

— If EL2 is using AArch32, this field is set to `VTTBR.VMID`.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally `UNKNOWN` value.

Otherwise:

Reserved, `RES0`.

When (FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented) and EDSCR.SC2 == 1:

### CONTEXTIDR_EL2

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTEXTIDR_EL2</td>
<td></td>
</tr>
</tbody>
</table>

Context ID. The value of `CONTEXTIDR_EL2` that is associated with the most recent `EDPCSR` sample. When the most recent `EDPCSR` sample is generated:

• If the PE is not executing at EL3, EL2 is using AArch64, and EL2 is enabled in the current Security state, then this field is set to the Context ID sampled from `CONTEXTIDR_EL2`.

• Otherwise, this field is set to an `UNKNOWN` value.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally `UNKNOWN` value.

Accessing the EDVIDSR:

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register `UNKNOWN`, see Permitted behavior that might make the PC Sample-based profiling registers `UNKNOWN` on page H7-10354.

EDVIDSR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x0A8</td>
<td>EDVIDSR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

• When `IsCorePowered()`, `!DoubleLockStatus()` and `!OSLockStatus()` accesses to this register are RO.

• Otherwise accesses to this register generate an error response.
H9.2.45  EDWAR, External Debug Watchpoint Address Register

The EDWAR characteristics are:

Purpose

Returns the virtual data address being accessed when a Watchpoint Debug Event was triggered.

Configurations

EDWAR is in the Core power domain.

Attributes

EDWAR is a 64-bit register.

Field descriptions

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
</tr>
<tr>
<td>[63:0]</td>
</tr>
</tbody>
</table>

Watchpoint address. The data virtual address being accessed when a Watchpoint Debug Event was triggered and caused entry to Debug state. This address must be within a naturally-aligned block of memory of power-of-two size no larger than the DC ZVA block size.

The value of this register is UNKNOWN if the PE is in Non-debug state, or if Debug state was entered other than for a Watchpoint debug event.

The value of EDWAR[63:32] is UNKNOWN if Debug state was entered for a Watchpoint debug event taken from AArch32 state.

The EDWAR is subject to the same alignment rules as the reporting of a watchpointed address in the FAR. See Determining the memory location that caused a Watchpoint exception on page D2-4734.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the EDWAR:

EDWAR[31:0] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x030</td>
<td>EDWAR</td>
<td>31:0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to EDWAR[31:0] are RO.
- Otherwise accesses to EDWAR[31:0] generate an error response.

EDWAR[63:32] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x034</td>
<td>EDWAR</td>
<td>63:32</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to EDWAR[63:32] are RO.
- Otherwise accesses to EDWAR[63:32] generate an error response.
H9.2.46 MIDR_EL1, Main ID Register

The MIDR_EL1 characteristics are:

**Purpose**

Provides identification information for the PE, including an implementer code for the device and a device ID number.

**Configurations**

External register MIDR_EL1 bits [31:0] are architecturally mapped to AArch64 System register MIDR_EL1[31:0].

External register MIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register MIDR[31:0].

It is IMPLEMENTATION DEFINED whether MIDR_EL1 is implemented in the Core power domain or in the Debug power domain.

**Attributes**

MIDR_EL1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>28 [23]</th>
<th>20 [19]</th>
<th>16 [15]</th>
<th>4</th>
<th>3</th>
<th>0</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementer</td>
<td>Variant</td>
<td>PartNum</td>
<td>Revision</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Implementer, bits [31:24]**

The Implementer code. This field must hold an implementer code that has been assigned by Arm. Assigned codes include the following:

- 0x00: Reserved for software use.
- 0x41: Arm Limited.
- 0x42: Broadcom Corporation.
- 0x43: Cavium Inc.
- 0x44: Digital Equipment Corporation.
- 0x46: Fujitsu Ltd.
- 0x49: Infineon Technologies AG.
- 0x4D: Motorola or Freescale Semiconductor Inc.
- 0x4E: NVIDIA Corporation.
- 0x50: Applied Micro Circuits Corporation.
- 0x51: Qualcomm Inc.
- 0x56: Marvell International Ltd.
- 0x69: Intel Corporation.
- 0xC0: Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.
Variant, bits [23:20]

Variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Architecture, bits [19:16]

Architecture version. Defined values are:

- `0b0001` Armv4.
- `0b0010` Armv4T.
- `0b0011` Armv5 (obsolete).
- `0b0100` Armv5T.
- `0b0101` Armv5TE.
- `0b0110` Armv5TEJ.
- `0b0111` Armv6.
- `0b1111` Architectural features are individually identified in the ID_* registers.

All other values are reserved.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

PartNum, bits [15:4]

Primary Part Number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are `0x0` or `0x7`, the variant and architecture are encoded differently.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [3:0]

Revision number for the device.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing the MIDR_EL1:

MIDR_EL1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x00</td>
<td>MIDR_EL1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered() and !DoubleLockStatus() accesses to this register are RO.
- Otherwise accesses to this register are IMPDEF.
**H9.2.47 OSLAR_EL1, OS Lock Access Register**

The OSLAR_EL1 characteristics are:

**Purpose**

Used to lock or unlock the OS Lock.

**Configurations**

External register OSLAR_EL1 bits [31:0] are architecturally mapped to AArch64 System register OSLAR_EL1[31:0].

OSLAR_EL1 is in the Core power domain.

The OS Lock can also be locked or unlocked using DBGOSLR.

If FEAT_Debugv8p2 is not implemented, it is IMPLEMENTATION DEFINED whether external debug accesses to OSLAR_EL1 are ignored and return an error when AllowExternalDebugAccess() returns FALSE for the access.

If FEAT_Debugv8p2 is implemented, external debug accesses to OSLAR_EL1 are ignored and return an error when AllowExternalDebugAccess() returns FALSE for the access.

**Attributes**

OSLAR_EL1 is a 32-bit register.

**Field descriptions**

![Field Description Diagram]

**Bits [31:1]**

Reserved, RES0.

**OSLK, bit [0]**

On writes to OSLAR_EL1, bit[0] is copied to the OS Lock.

Use EDPRSR.OSLK to check the current status of the lock.

**Accessing the OSLAR_EL1:**

______ Note _______

SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

OSLAR_EL1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>0x300</td>
<td>OSLAR_EL1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), AllowExternalDebugAccess() and SoftwareLockStatus() accesses to this register are WI.
- When IsCorePowered(), !DoubleLockStatus(), AllowExternalDebugAccess() and !SoftwareLockStatus() accesses to this register are WO.
• When IsCorePowered(), !DoubleLockStatus(), !AllowExternalDebugAccess() and FEAT_Debugv8p2 is not implemented accesses to this register are IMPDEF.

• Otherwise accesses to this register generate an error response.
H9.3 **External trace registers**

This section lists the external Trace unit registers.
H9.3.1 TRCACATR<n>, Address Comparator Access Type Register <n>, n = 0 - 15

The TRCACATR<n> characteristics are:

Purpose

Defines the type of access for the corresponding TRCACVR<n> Register. This register configures
the context type, Exception levels, alignment, masking that is applied by the Address Comparator,
and how the Address Comparator behaves when it is one half of an Address Range Comparator.

Configurations

External register TRCACATR<n> bits [63:0] are architecturally mapped to AArch64 System
register TRCACATR<n>[63:0].

This register is present only when FEAT_ETE is implemented and
UInt(TRCIDR4.NUMACPAIRS) * 2 > n. Otherwise, direct accesses to TRCACATR<n> are RES0.

Attributes

TRCACATR<n> is a 64-bit register.

Field descriptions

Bits [63:19]

Reserved, RES0.

EXLEVEL_RL_EL2, bit [18]

When TRCIDR6.EXLEVEL_RL_EL2 == 1:

Realm EL2 address comparison control. Controls whether a comparison can occur at EL2 in Realm
state.

0b0 When TRCACATR<n>.EXLEVEL_NS_EL2 is 0 the Address Comparator performs
comparisons in Realm EL2.
When TRCACATR<n>.EXLEVEL_NS_EL2 is 1 the Address Comparator does not
perform comparisons in Realm EL2.

0b1 When TRCACATR<n>.EXLEVEL_NS_EL2 is 0 the Address Comparator does not
perform comparisons in Realm EL2.
When TRCACATR<n>.EXLEVEL_NS_EL2 is 1 the Address Comparator performs
comparisons in Realm EL2.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
EXLEVEL_RL_EL1, bit [17]

When TRCIDR6.EXLEVEL_RL_EL1 == 1:

Realm EL1 address comparison control. Controls whether a comparison can occur at EL1 in Realm state.

\[0b0\] When TRCACATR<n>.EXLEVEL_NS_EL1 is 0 the Address Comparator performs comparisons in Realm EL1.
When TRCACATR<n>.EXLEVEL_NS_EL1 is 1 the Address Comparator does not perform comparisons in Realm EL1.

\[0b1\] When TRCACATR<n>.EXLEVEL_NS_EL1 is 0 the Address Comparator does not perform comparisons in Realm EL1.
When TRCACATR<n>.EXLEVEL_NS_EL1 is 1 the Address Comparator performs comparisons in Realm EL1.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_RL_EL0, bit [16]

When TRCIDR6.EXLEVEL_RL_EL0 == 1:

Realm EL0 address comparison control. Controls whether a comparison can occur at EL0 in Realm state.

\[0b0\] When TRCACATR<n>.EXLEVEL_NS_EL0 is 0 the Address Comparator performs comparisons in Realm EL0.
When TRCACATR<n>.EXLEVEL_NS_EL0 is 1 the Address Comparator does not perform comparisons in Realm EL0.

\[0b1\] When TRCACATR<n>.EXLEVEL_NS_EL0 is 0 the Address Comparator does not perform comparisons in Realm EL0.
When TRCACATR<n>.EXLEVEL_NS_EL0 is 1 the Address Comparator performs comparisons in Realm EL0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [15]

Reserved, RES0.

EXLEVEL_NS_EL2, bit [14]

When Non-secure EL2 is implemented:

Non-secure EL2 address comparison control. Controls whether a comparison can occur at EL2 in Non-secure state.

\[0b0\] The Address Comparator performs comparisons in Non-secure EL2.
\[0b1\] The Address Comparator does not perform comparisons in Non-secure EL2.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.
EXLEVEL_NS_EL1, bit [13]

When Non-secure EL1 is implemented:

Non-secure EL1 address comparison control. Controls whether a comparison can occur at EL1 in Non-secure state.

0b0 The Address Comparator performs comparisons in Non-secure EL1.
0b1 The Address Comparator does not perform comparisons in Non-secure EL1.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EXLEVEL_NS_EL0, bit [12]

When Non-secure EL0 is implemented:

Non-secure EL0 address comparison control. Controls whether a comparison can occur at EL0 in Non-secure state.

0b0 The Address Comparator performs comparisons in Non-secure EL0.
0b1 The Address Comparator does not perform comparisons in Non-secure EL0.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EXLEVEL_S_EL3, bit [11]

When EL3 is implemented:

EL3 address comparison control. Controls whether a comparison can occur at EL3.

0b0 The Address Comparator performs comparisons in EL3.
0b1 The Address Comparator does not perform comparisons in EL3.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EXLEVEL_S_EL2, bit [10]

When EL2 is implemented and FEAT_SEL2 is implemented:

Secure EL2 address comparison control. Controls whether a comparison can occur at EL2 in Secure state.

0b0 The Address Comparator performs comparisons in Secure EL2.
0b1 The Address Comparator does not perform comparisons in Secure EL2.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EXLEVEL_S_EL1, bit [9]

When Secure EL1 is implemented:

Secure EL1 address comparison control. Controls whether a comparison can occur at EL1 in Secure state.

0b0 The Address Comparator performs comparisons in Secure EL1.
The Address Comparator does not perform comparisons in Secure EL1.
The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKOWN value.

**Otherwise:**
Reserved, RES0.

**EXLEVEL_S_EL0, bit [8]**

*When Secure EL0 is implemented:*
Secure EL0 address comparison control. Controls whether a comparison can occur at EL0 in Secure state.

- **0b0** The Address Comparator performs comparisons in Secure EL0.
- **0b1** The Address Comparator does not perform comparisons in Secure EL0.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKOWN value.

**Otherwise:**
Reserved, RES0.

**Bit [7]**
Reserved, RES0.

**CONTEXT, bits [6:4]**

*When TRCIDR4.NUMCIDC != 0b0000 or TRCIDR4.NUMVMIDC != 0b0000:*
Selects a Context Identifier Comparator or Virtual Context Identifier Comparator:

- **0b000** Comparator 0.
- **0b001** *When UInt(TRCIDR4.NUMCIDC) > 1 or UInt(TRCIDR4.NUMVMIDC) > 1:*
  Comparator 1.
- **0b010** *When UInt(TRCIDR4.NUMCIDC) > 2 or UInt(TRCIDR4.NUMVMIDC) > 2:*
  Comparator 2.
- **0b011** *When UInt(TRCIDR4.NUMCIDC) > 3 or UInt(TRCIDR4.NUMVMIDC) > 3:*
  Comparator 3.
- **0b100** *When UInt(TRCIDR4.NUMCIDC) > 4 or UInt(TRCIDR4.NUMVMIDC) > 4:*
  Comparator 4.
- **0b101** *When UInt(TRCIDR4.NUMCIDC) > 5 or UInt(TRCIDR4.NUMVMIDC) > 5:*
  Comparator 5.
- **0b110** *When UInt(TRCIDR4.NUMCIDC) > 6 or UInt(TRCIDR4.NUMVMIDC) > 6:*
  Comparator 6.
- **0b111** *When UInt(TRCIDR4.NUMCIDC) > 7 or UInt(TRCIDR4.NUMVMIDC) > 7:*
  Comparator 7.

The width of this field is dependent on the maximum number of Context Identifier Comparators or Virtual Context Identifier Comparators implemented. Unimplemented bits are RES0.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKOWN value.

**Otherwise:**
Reserved, RES0.
CONTEXTTYPE, bits [3:2]

When TRCIDR4.NUMCIDC != 0b0000 or TRCIDR4.NUMVMIDC != 0b0000:

Controls whether the Address Comparator is dependent on a Context Identifier Comparator, a Virtual Context Identifier Comparator, or both comparisons.

0b00 When TRCIDR4.NUMCIDC != 0b0000:
The Address Comparator is not dependent on the Context Identifier Comparators or Virtual Context Identifier Comparators.

0b01 When TRCIDR4.NUMCIDC != 0b0000:
The Address Comparator is dependent on the Context Identifier Comparator that TRCACATR<n>.CONTEXT specifies. The Address Comparator signals a match only if both the Context Identifier Comparator and the address comparison match.

0b10 When TRCIDR4.NUMVMIDC != 0b0000:
The Address Comparator is dependent on the Virtual Context Identifier Comparator that TRCACATR<n>.CONTEXT specifies. The Address Comparator signals a match only if both the Virtual Context Identifier Comparator and the address comparison match.

0b11 When TRCIDR4.NUMCIDC != 0b0000 and TRCIDR4.NUMVMIDC != 0b0000:
The Address Comparator is dependent on the Context Identifier Comparator and Virtual Context Identifier Comparator that TRCACATR<n>.CONTEXT specifies. The Address Comparator signals a match only if the Context Identifier Comparator, the Virtual Context Identifier Comparator, and address comparison all match.

If TRCIDR4.NUMCIDC == 0b0000, then bit [2] is RES0.
If TRCIDR4.NUMVMIDC == 0b0000, then bit [3] is RES0.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bits [1:0]
Reserved, RES0.

Accessing the TRCACATR<n>: 
Must be programmed if any of the following are true:

• TRCBBCTRL.RANGE[n/2] == 1.
• TRCRSCTRL<n>.GROUP == 0b0100 and TRCRSCTRL<n>.SAC[n] == 1.
• TRCRSCTRL<n>.GROUP == 0b0101 and TRCRSCTRL<n>.ARC[n/2] == 1.
• TRCVIIECTRL.EXCLUDE[n/2] == 1.
• TRCVIIECTRL.INCLUDE[n/2] == 1.
• TRCVISSCTRL.START[n] == 1.
• TRCVISSCTRL.STOP[n] == 1.
• TRCSSCCR<>.ARC[n/2] == 1.
• TRCSSCCR<>.SAC[n] == 1.
• TRCQCTRL.RANGE[n/2] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.
TRCACATR<\textit{n}> can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>$0x480 + (8 \times n)$</td>
<td>TRCACATR&lt;\textit{n}&gt;</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.2 TRCACVR<n>, Address Comparator Value Register <n>, n = 0 - 15

The TRCACVR<n> characteristics are:

**Purpose**
Contains the address value.

**Configurations**
External register TRCACVR<n> bits [63:0] are architecturally mapped to AArch64 System register TRCACVR<n>[63:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR4.NUMACPAIRS) * 2 > n. Otherwise, direct accesses to TRCACVR<n> are RES0.

**Attributes**
TRCACVR<n> is a 64-bit register.

**Field descriptions**

ADDRESS, bits [63:0]
Address Value.

The Address Comparators can support implementations that use multiple address widths. When the trace unit compares the ADDRESS field with an address that has a width less than this field, then
the address must be zero-extended to the ADDRESS field width. The trace unit then compares all implemented bits. For example, in a system that supports both 32-bit and 64-bit addresses, when the
PE is in AArch32 state the comparator must zero-extend the 32-bit address and compare against the full 64 bits that are stored in the TRCACVR<n>. This requires that the trace analyzer always
programs all implemented bits of the TRCACVR<n>.

The result of writing a value other than all zeros or all ones to ADDRESS at bits[63:P] is an
UNKNOWN value, where P is defined as the maximum virtual address size supported by the PE.

The result of writing a value of all zeros or all ones to ADDRESS at bits[63:P] is the written value,
and a read of the register returns the written value.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing the TRCACVR<n>:**
Must be programmed if any of the following are true:

- TRCBBCTLR.RANGE[n/2] == 1.
- TRCRSCTLR<n>.GROUP == 0b0100 and TRCRSCTLR<n>.SAC[n] == 1.
- TRCRSCTLR<n>.GROUP == 0b0101 and TRCRSCTLR<n>.ARC[n/2] == 1.
- TRCVIIECTLR.EXCLUDE[n/2] == 1.
- TRCVIIECTLR.INCLUDE[n/2] == 1.
- TRCVISSCTLR.START[n] == 1.
• TRCVISSCTRL.STOP[n] == 1.
• TRCSSCCR<>.ARC[n/2] == 1.
• TRCSSCCR<>.SAC[n] == 1.
• TRCQCTRL.RANGE[n/2] == 1.

Writes are constrained unpredictable if the trace unit is not in the Idle state.

TRCACVR<n> can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x400 + (8 * n)</td>
<td>TRCACVR&lt;n&gt;</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

• When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
• Otherwise accesses to this register are RW.
H9.3.3 TRCAUTHSTATUS, Authentication Status Register

The TRCAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

For additional information, see the CoreSight Architecture Specification.

Configurations

External register TRCAUTHSTATUS bits [31:0] are architecturally mapped to AArch64 System register TRCAUTHSTATUS[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCAUTHSTATUS are RES0.

Attributes

TRCAUTHSTATUS is a 32-bit register.

Field descriptions

| 31  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| RES0| RTID| RES0| RLID| HNID| HID | SNID| SID | RES0| RTNID| RLNID| NSNID|

Bits [31:28]

Reserved, RES0.

RTNID, bits [27:26]

Root non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RTNID.

RTID, bits [25:24]

Root invasive debug.

0b00 Not implemented.

Bits [23:16]

Reserved, RES0.

RLNID, bits [15:14]

Realm non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RLNID.

RLID, bits [13:12]

Realm invasive debug.

0b00 Not implemented.

HNID, bits [11:10]

Hyp Non-invasive Debug. Indicates whether a separate enable control for EL2 non-invasive debug features is implemented and enabled.

0b00 Separate Hyp non-invasive debug enable not implemented, or EL2 non-invasive debug features not implemented.

0b10 Implemented and disabled.

0b11 Implemented and enabled.
All other values are reserved.
This field reads as 0b00.

**HID, bits [9:8]**

Hyp Invasive Debug. Indicates whether a separate enable control for EL2 invasive debug features is implemented and enabled.

- 0b00: Separate Hyp invasive debug enable not implemented, or EL2 invasive debug features not implemented.
- 0b10: Implemented and disabled.
- 0b11: Implemented and enabled.

All other values are reserved.
This field reads as 0b00.

**SNID, bits [7:6]**

Secure Non-invasive Debug. Indicates whether Secure non-invasive debug features are implemented and enabled.

- 0b00: Secure non-invasive debug features not implemented.
- 0b10: Implemented and disabled.
- 0b11: Implemented and enabled.

All other values are reserved.
When EL3 is implemented, this field takes the value 0b10 or 0b11 depending whether Secure non-invasive debug is enabled.
When EL3 is not implemented and the PE is Non-secure, this field reads as 0b00.
When EL3 is not implemented and the PE is Secure, this field takes the value 0b10 or 0b11 depending whether Secure non-invasive debug is enabled.

**SID, bits [5:4]**

Secure Invasive Debug. Indicates whether Secure invasive debug features are implemented and enabled.

- 0b00: Secure invasive debug features not implemented.
- 0b10: Implemented and disabled.
- 0b11: Implemented and enabled.

All other values are reserved.
This field reads as 0b00.

**NSNID, bits [3:2]**

Non-secure Non-invasive Debug. Indicates whether Non-secure non-invasive debug features are implemented and enabled.

- 0b00: Non-secure non-invasive debug features not implemented.
- 0b10: Implemented and disabled.
- 0b11: Implemented and enabled.

All other values are reserved.
When EL3 is implemented, this field reads as 0b11.
When EL3 is not implemented and the PE is Non-secure, this field reads as 0b11.
When EL3 is not implemented and the PE is Secure, this field reads as 0b00.

**NSID, bits [1:0]**

Non-secure Invasive Debug. Indicates whether Non-secure invasive debug features are implemented and enabled.

- 0b00: Non-secure invasive debug features not implemented.
0b10 Implemented and disabled.
0b11 Implemented and enabled.
All other values are reserved.
This field reads as 0b00.

**Accessing the TRCAUTHSTATUS:**

For implementations that support multiple access mechanisms, different access mechanisms can return different values for reads of TRCAUTHSTATUS if the authentication signals have changed and that change has not yet been synchronized by a Context synchronization event. This scenario can happen if, for example, the external debugger view is implemented separately from the system instruction view to allow for separate power domains, and so observes changes on the signals differently.

External debugger accesses to this register are unaffected by the OS Lock.

TRCAUTHSTATUS can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xF8</td>
<td>TRCAUTHSTATUS</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.4 TRCAUXCTLR, Auxiliary Control Register

The TRCAUXCTLR characteristics are:

Purpose
The function of this register is IMPLEMENTATION DEFINED.

Configurations
External register TRCAUXCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCAUXCTLR[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCAUXCTLR are RES0.

Attributes
TRCAUXCTLR is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
</tbody>
</table>

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.
This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to 0.

Accessing the TRCAUXCTLR:
If this register is nonzero then it might cause the behavior of a trace unit to contradict this architecture specification. See the documentation of the specific implementation for information about the IMPLEMENTATION DEFINED support for this register.

TRCAUXCTLR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x018</td>
<td>TRCAUXCTLR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:
• When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
• Otherwise accesses to this register are RW.
H9.3.5 TRCBBCTRLR, Branch Broadcast Control Register

The TRCBBCTRLR characteristics are:

**Purpose**

Controls the regions in the memory map where branch broadcasting is active.

**Configurations**

External register TRCBBCTRLR bits [31:0] are architecturally mapped to AArch64 System register TRCBBCTRLR[31:0].

This register is present only when FEAT_ETE is implemented, TRCIDR0.TRCBB == 1 and UInt(TRCIDR4.NUMACPAIRS) > 0. Otherwise, direct accesses to TRCBBCTRLR are RES0.

**Attributes**

TRCBBCTRLR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-9</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>MODE, bit [8]</td>
<td></td>
</tr>
<tr>
<td>7-0</td>
<td>RANGE[m], for m = 7 to 0</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:9]**

Reserved, RES0.

**MODE, bit [8]**

Mode.

0b0  Exclude Mode.

Branch broadcasting is not active for instructions in the address ranges defined by TRCBBCTRLR.RANGE.

If TRCBBCTRLR.RANGE == 0x00 then branch broadcasting is active for all instructions.

0b1  Include Mode.

Branch broadcasting is active for instructions in the address ranges defined by TRCBBCTRLR.RANGE.

If TRCBBCTRLR.RANGE == 0x00 then the behavior of the trace unit is CONSTRAINED UNPREDICTABLE. That is, the trace unit might or might not consider any instructions to be in a branch broadcasting region.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**RANGE[m], bit [m], for m = 7 to 0**

Address range field.

Selects whether Address Range Comparator <m> is used with branch broadcasting.

0b0  The address range that Address Range Comparator <m> defines, is not selected.

0b1  The address range that Address Range Comparator <m> defines, is selected.

This bit is RES0 if m >= TRCIDR4.NUMACPAIRS.
The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally **UNKNOWN** value.

**Accessing the TRCBBCTLRL:**

Must be programmed if TRCCONFIGR.BB == 1.

Writes are **CONSTRAINED UNPREDICTABLE** if the trace unit is not in the Idle state.

TRCBBCTLRL can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x03C</td>
<td>TRCBBCTLRL</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.

- Otherwise accesses to this register are RW.
H9.3.6  TRCCCCTLR, Cycle Count Control Register

The TRCCCCTLR characteristics are:

Purpose
Set the threshold value for cycle counting.

Configurations
External register TRCCCCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCCCCTLR[31:0].
This register is present only when FEAT_ETE is implemented and TRCIDR0.TRCCCI == 1. Otherwise, direct accesses to TRCCCCTLR are RES0.

Attributes
TRCCCCTLR is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>31-12</td>
</tr>
<tr>
<td>THRESHOLD</td>
<td>11-0</td>
</tr>
</tbody>
</table>

Bits [31:12]
Reserved, RES0.

THRESHOLD, bits [11:0]
Sets the threshold value for instruction trace cycle counting.
The minimum threshold value that can be programmed into THRESHOLD is given in TRCIDR3.CCITMIN. If the THRESHOLD value is smaller than the value in TRCIDR3.CCITMIN then the behavior is CONSTRAINED UNPREDICTABLE. That is, cycle counts might or might not be included in the trace and the cycle count threshold is not known.
Writing a value of zero when TRCCONFIGR.CCI enables instruction trace cycle counting results in CONSTRAINED UNPREDICTABLE behavior. That is, cycle counts might or might not be included in the trace and the cycle count threshold is not known.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing the TRCCCCTLR:
Must be programmed if TRCCONFIGR.CCI == 1.
Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCCCTLR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x038</td>
<td>TRCCCCTLR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:
• When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
• Otherwise accesses to this register are RW.
H9.3.7 TRCCIDCCTRL0, Context Identifier Comparator Control Register 0

The TRCCIDCCTRL0 characteristics are:

**Purpose**

Contains Context identifier mask values for the TRCCIDCVR<n> registers, for n = 0 to 3.

**Configurations**

External register TRCCIDCCTRL0 bits [31:0] are architecturally mapped to AArch64 System register TRCCIDCCTRL0[31:0].

This register is present only when FEAT_ETE is implemented, UInt(TRCIDR4.NUMCIDC) > 0x0 and UInt(TRCIDR2.CIDSIZE) > 0. Otherwise, direct accesses to TRCCIDCCTRL0 are RES0.

**Attributes**

TRCCIDCCTRL0 is a 32-bit register.

**Field descriptions**

```
<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>24</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP3[&lt;m&gt;]</td>
<td>COMP2[&lt;m&gt;]</td>
<td>COMP1[&lt;m&gt;]</td>
<td>COMP0[&lt;m&gt;]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**COMP3[<m>], bit [m+24], for m = 7 to 0**

*When UInt(TRCIDR4.NUMCIDC) > 3:*

TRCCIDCVR3 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR3. Each bit in this field corresponds to a byte in TRCCIDCVR3.

- **0b0** The trace unit includes TRCCIDCVR3[(m×8+7):(m×8)] when it performs the Context identifier comparison.
- **0b1** The trace unit ignores TRCCIDCVR3[(m×8+7):(m×8)] when it performs the Context identifier comparison.

This bit is RES0 if m >= TRCIDR2.CIDSIZE.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**COMP2[<m>], bit [m+16], for m = 7 to 0**

*When UInt(TRCIDR4.NUMCIDC) > 2:*

TRCCIDCVR2 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR2. Each bit in this field corresponds to a byte in TRCCIDCVR2.

- **0b0** The trace unit includes TRCCIDCVR2[(m×8+7):(m×8)] when it performs the Context identifier comparison.
- **0b1** The trace unit ignores TRCCIDCVR2[(m×8+7):(m×8)] when it performs the Context identifier comparison.

This bit is RES0 if m >= TRCIDR2.CIDSIZE.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**COMP1[<m>], bit [m+8], for m = 7 to 0**

*When UInt(TRCIDR4.NUMCIDC) > 1:*

TRCCIDCVR1 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR1. Each bit in this field corresponds to a byte in TRCCIDCVR1.

- **0b0** The trace unit includes TRCCIDCVR1[(m×8+7):(m×8)] when it performs the Context identifier comparison.
The trace unit ignores TRCCIDCVR1[(m×8+7):(m×8)] when it performs the Context identifier comparison.

This bit is RES0 if m >= TRCIDR2.CIDSIZE.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

COMP0[<m>], bit [m], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 0:
TRCCIDCVR0 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR0. Each bit in this field corresponds to a byte in TRCCIDCVR0.

This bit is RES0 if m >= TRCIDR2.CIDSIZE.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

### Accessing the TRCCIDCCTRL0:

If software uses the TRCCIDCVR<n> registers, for n = 0 to 3, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCCIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCCIDCVR<n> is not 0x00, the behavior of the Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCIDCCTRL0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x680</td>
<td>TRCCIDCCTRL0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:
- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
### H9.3.8 TRCCIDCCTRL1, Context Identifier Comparator Control Register 1

The TRCCIDCCTRL1 characteristics are:

**Purpose**

Contains Context identifier mask values for the TRCCIDCVR\(<n>\) registers, for \(n = 4\) to \(7\).

**Configurations**

External register TRCCIDCCTRL1 bits [31:0] are architecturally mapped to AArch64 System register TRCCIDCCTRL1\[31:0\].

This register is present only when FEAT_ETE is implemented, UInt(TRCIDR4.NUMCIDC) > 0x4 and UInt(TRCIDR2.CIDSIZE) > 0. Otherwise, direct accesses to TRCCIDCCTRL1 are RES0.

**Attributes**

TRCCIDCCTRL1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP7(&lt;m&gt;), bit [m+24], for m = 7 to 0</td>
</tr>
</tbody>
</table>

*When UInt(TRCIDR4.NUMCIDC) > 7:*

- TRCCIDCVR7 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR7. Each bit in this field corresponds to a byte in TRCCIDCVR7.

  \- 0b0 The trace unit includes TRCCIDCVR7[(m×8+7):(m×8)] when it performs the Context identifier comparison.

  \- 0b1 The trace unit ignores TRCCIDCVR7[(m×8+7):(m×8)] when it performs the Context identifier comparison.

- This bit is RES0 if m >= TRCIDR2.CIDSIZE.

  The reset behavior of this field is:

  - On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

<table>
<thead>
<tr>
<th>Field description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP6(&lt;m&gt;), bit [m+16], for m = 7 to 0</td>
</tr>
</tbody>
</table>

*When UInt(TRCIDR4.NUMCIDC) > 6:*

- TRCCIDCVR6 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR6. Each bit in this field corresponds to a byte in TRCCIDCVR6.

  \- 0b0 The trace unit includes TRCCIDCVR6[(m×8+7):(m×8)] when it performs the Context identifier comparison.

  \- 0b1 The trace unit ignores TRCCIDCVR6[(m×8+7):(m×8)] when it performs the Context identifier comparison.

- This bit is RES0 if m >= TRCIDR2.CIDSIZE.

  The reset behavior of this field is:

  - On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

<table>
<thead>
<tr>
<th>Field description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP5(&lt;m&gt;), bit [m+8], for m = 7 to 0</td>
</tr>
</tbody>
</table>

*When UInt(TRCIDR4.NUMCIDC) > 5:*

- TRCCIDCVR5 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR5. Each bit in this field corresponds to a byte in TRCCIDCVR5.

  \- 0b0 The trace unit includes TRCCIDCVR5[(m×8+7):(m×8)] when it performs the Context identifier comparison.
0b1 The trace unit ignores TRCCIDCVR5[(m×8+7):(m×8)] when it performs the Context identifier comparison.

This bit is RES0 if m >= TRCIDR2.CIDSIZE.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

COMP4[<m>], bit [m], for m = 7 to 0

When UInt(TRCIDR4,NUMCIDC) > 4:

TRCCIDCVR4 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR4. Each bit in this field corresponds to a byte in TRCCIDCVR4.

0b0 The trace unit includes TRCCIDCVR4[(m×8+7):(m×8)] when it performs the Context identifier comparison.

0b1 The trace unit ignores TRCCIDCVR4[(m×8+7):(m×8)] when it performs the Context identifier comparison.

This bit is RES0 if m >= TRCIDR2.CIDSIZE.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing the TRCCIDCCTLR1:

If software uses the TRCCIDCVR<n> registers, for n = 4 to 7, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCCIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCCIDCVR<n> is not 0x00, the behavior of the Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

 Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCIDCCTLR1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x684</td>
<td>TRCCIDCCTLR1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

• When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.

• Otherwise accesses to this register are RW.
H9.3.9 TRCCIDCVR<\text{n}>, Context Identifier Comparator Value Registers <\text{n}>, n = 0 - 7

The TRCCIDCVR<\text{n}> characteristics are:

**Purpose**

Contains a Context identifier value.

**Configurations**

External register TRCCIDCVR<\text{n}> bits [63:0] are architecturally mapped to AArch64 System register TRCCIDCVR<\text{n}>[63:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR4.NUMCIDC) > \text{n}. Otherwise, direct accesses to TRCCIDCVR<\text{n}> are RES0.

**Attributes**

TRCCIDCVR<\text{n}> is a 64-bit register.

**Field descriptions**

![Field descriptions diagram]

**VALUE**, bits [63:0]

Context identifier value. The width of this field is indicated by TRCIDR2.CIDSIZE. Unimplemented bits are RES0. After a PE Reset, the trace unit assumes that the Context identifier is zero until the PE updates the Context identifier.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing the TRCCIDCVR<\text{n}>:**

Must be programmed if any of the following are true:

- TRCRSCTLR<\text{n}>,GROUP == 0b0110 and TRCRSCTLR<\text{n}>,CID[\text{n}] == 1.
- TRCACATR<\text{n}>,CONTEXTTYPE == 0b01 or 0b11 and TRCACATR<\text{n}>,CONTEXT == \text{n}.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCIDCVR<\text{n}> can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x600 + (8 * \text{n})</td>
<td>TRCCIDCVR&lt;\text{n}&gt;</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.10 TRCCIDR0, Component Identification Register 0

The TRCCIDR0 characteristics are:

**Purpose**

Provides discovery information about the component.
For additional information, see the CoreSight Architecture Specification.

**Configurations**

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCCIDR0 are RES0.

**Attributes**

TRCCIDR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:8]</th>
<th>00001101</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRMBL_0, bits [7:0]</td>
<td>00001101</td>
</tr>
</tbody>
</table>

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Component identification preamble, segment 0.
Reads as 0x0D.
Access to this field is RO.

**Accessing the TRCCIDR0:**

External debugger accesses to this register are unaffected by the OS Lock.
TRCCIDR0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFF0</td>
<td>TRCCIDR0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.11 TRCCIDR1, Component Identification Register 1

The TRCCIDR1 characteristics are:

**Purpose**

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

**Configurations**

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCCIDR1 are RES0.

**Attributes**

TRCCIDR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:8</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>7:4</td>
<td>CLASS, component class.</td>
</tr>
<tr>
<td>0b1001</td>
<td>CoreSight peripheral. Other values are defined by the CoreSight Architecture. This field reads as 0x9.</td>
</tr>
<tr>
<td>3:0</td>
<td>PRMBL_1, component identification preamble, segment 1. Reads as 0b0000. Access to this field is RO.</td>
</tr>
</tbody>
</table>

**Accessing the TRCCIDR1:**

External debugger accesses to this register are unaffected by the OS Lock.

TRCCIDR1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFF4</td>
<td>TRCCIDR1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.12 TRCCIDR2, Component Identification Register 2

The TRCCIDR2 characteristics are:

**Purpose**

Provides discovery information about the component. 
For additional information, see the CoreSight Architecture Specification.

**Configurations**

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCCIDR2 are RES0.

**Attributes**

TRCCIDR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:8]</td>
<td>Reserved</td>
<td>RES0</td>
</tr>
<tr>
<td>[7:0]</td>
<td>PRMBL_2</td>
<td>0x05</td>
</tr>
</tbody>
</table>

Access to this field is RO.

**Accessing the TRCCIDR2:**

External debugger accesses to this register are unaffected by the OS Lock.

TRCCIDR2 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFF8</td>
<td>TRCCIDR2</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.13   TRCCIDR3, Component Identification Register 3

The TRCCIDR3 characteristics are:

**Purpose**

Provides discovery information about the component.
For additional information, see the CoreSight Architecture Specification.

**Configurations**

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCCIDR3 are RES0.

**Attributes**

TRCCIDR3 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:8]</td>
<td>Reserved, RES0</td>
<td>RES0</td>
</tr>
<tr>
<td>[7:0]</td>
<td>PRMBL_3, Component identification preamble, segment 3.</td>
<td>0xB1</td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**PRMBL_3, bits [7:0]**

Component identification preamble, segment 3.
Reads as 0xB1.
Access to this field is RO.

**Accessing the TRCCIDR3:**

External debugger accesses to this register are unaffected by the OS Lock.

TRCCIDR3 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFFF</td>
<td>TRCCIDR3</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.14  TRCCLAIMCLR, Claim Tag Clear Register

The TRCCLAIMCLR characteristics are:

**Purpose**

In conjunction with TRCCLAIMSET, provides Claim Tag bits that can be separately set and cleared to indicate whether functionality is in use by a debug agent.

For additional information, see the CoreSight Architecture Specification.

**Configurations**

External register TRCCLAIMCLR bits [31:0] are architecturally mapped to AArch64 System register TRCCLAIMCLR[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCCLAIMCLR are RES0.

**Attributes**

TRCCLAIMCLR is a 32-bit register.

**Field descriptions**

CLR[<m>], bit [m], for m = 31 to 0

Claim Tag Clear. Indicates the current status of Claim Tag bit <m>, and is used to clear Claim Tag bit <m> to 0.

- **0b0**  On a read: Claim Tag bit <m> is not set.
  On a write: Ignored.

- **0b1**  On a read: Claim Tag bit <m> is set.
  On a write: Clear Claim tag bit <m> to 0.

The number of Claim Tag bits implemented is indicated in TRCCLAIMSET.

This bit reads-as-zero and ignores writes if m > the number of Claim Tag bits.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to 0.

Access to this field is W1C.
Accessing the TRCCLAIMCLR:

TRCCLAIMCLR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFA4</td>
<td>TRCCLAIMCLR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.15 TRCCLAIMSET, Claim Tag Set Register

The TRCCLAIMSET characteristics are:

**Purpose**

In conjunction with TRCCLAIMCLR, provides Claim Tag bits that can be separately set and cleared to indicate whether functionality is in use by a debug agent.

For additional information, see the CoreSight Architecture Specification.

**Configurations**

External register TRCCLAIMSET bits [31:0] are architecturally mapped to AArch64 System register TRCCLAIMSET[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCCLAIMSET are RES0.

The number of claim tag bits implemented is IMPLEMENTATION DEFINED. Arm recommends that implementations support a minimum of four claim tag bits, that is, SET[3:0] reads as 0b1111.

**Attributes**

TRCCLAIMSET is a 32-bit register.

**Field descriptions**

```
SET[<m>], bit [m], for m = 31 to 0

Claim Tag Set. Indicates whether Claim Tag bit <m> is implemented, and is used to set Claim Tag bit <m> to 1.

0b0   On a read: Claim Tag bit <m> is not implemented.
     On a write: Ignored.

0b1   On a read: Claim Tag bit <m> is implemented.
     On a write: Set Claim Tag bit <m> to 1.

This bit reads-as-zero and ignores writes if m > the number of Claim Tag bits.
Access to this field is RAO/W1S.
```
Accessing the TRCCLAIMSET:

TRCCLAIMSET can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFA0</td>
<td>TRCCLAIMSET</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.16 TRCCNTCTLR<n>, Counter Control Register <n>, n = 0 - 3

The TRCCNTCTLR<n> characteristics are:

**Purpose**
Controls the operation of Counter <n>.

**Configurations**
External register TRCCNTCTLR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCCNTCTLR<n>[31:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR5.NUMCNTR) > n. Otherwise, direct accesses to TRCCNTCTLR<n> are RES0.

**Attributes**
TRCCNTCTLR<n> is a 32-bit register.

**Field descriptions**

![Register Diagram]

**Bits [31:18]**
- **Reserved, RES0.**

**CNTCHAIN, bit [17]**
For TRCCNTCTLR3 and TRCCNTCTLR1, this field controls whether the Counter decrements when a reload event occurs for Counter <n-1>.

- **0b0** The Counter does not decrement when a reload event for Counter <n-1> occurs.
- **0b1** Counter <n> decrements when a reload event for Counter <n-1> occurs. This concatenates Counter <n> and Counter <n-1>, to provide a larger count value.

CNTCHAIN is not implemented for TRCCNTCTLR0 and TRCCNTCTLR2.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**RLDSELF, bit [16]**
Controls whether a reload event occurs for the Counter, when the Counter reaches zero.

- **0b0** Normal mode.
  - The Counter is in Normal mode.

- **0b1** Self-reload mode.
  - The Counter is in Self-reload mode.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**RLDEVENT_TYPE, bit [15]**
Chooses the type of Resource Selector.

Selects an event, that when it occurs causes a reload event for Counter <n>.

- **0b0** A single Resource Selector.
  - TRCCNTCTLR<n>.RLDEVENT_SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
A Boolean-combined pair of Resource Selectors.

TRCCNTCTLR<\text{n}>.RLDEVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCCNTCTLR<\text{n}>.RLDEVENT.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Bits [14:13]**

Reserved, RES0.

**RLDEVENT_SEL, bits [12:8]**

Defines the selected Resource Selector or pair of Resource Selectors.

TRCCNTCTLR<\text{n}>.RLDEVENT.TYPE controls whether TRCCNTCTLR<\text{n}>.RLDEVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

Selects an event, that when it occurs causes a reload event for Counter \(<\text{n}>\).

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**CNTEVENT_TYPE, bit [7]**

Chooses the type of Resource Selector.

Selects an event, that when it occurs causes Counter \(<\text{n}>\) to decrement.

- **0b0** A single Resource Selector.
  
  TRCCNTCTLR<\text{n}>.CNTEVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

- **0b1** A Boolean-combined pair of Resource Selectors.
  
  TRCCNTCTLR<\text{n}>.CNTEVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCCNTCTLR<\text{n}>.CNTEVENT.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Bits [6:5]**

Reserved, RES0.

**CNTEVENT_SEL, bits [4:0]**

Defines the selected Resource Selector or pair of Resource Selectors.

TRCCNTCTLR<\text{n}>.CNTEVENT.TYPE controls whether TRCCNTCTLR<\text{n}>.CNTEVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

Selects an event, that when it occurs causes Counter \(<\text{n}>\) to decrement.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
Accessing the TRCCNTCTRL<n>:

Must be programmed if \( \text{TRCRSCTRL}<n> \).GROUP == 0b0010 and \( \text{TRCRSCTRL}<n> \).COUNTERS\[n\] == 1.

Writes are constrained unpredictable if the trace unit is not in the Idle state.

TRCCNTCTRL<n> can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x150 + (4 * n)</td>
<td>TRCCNTCTRL&lt;n&gt;</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.17 TRCCNTRLDVR<n>, Counter Reload Value Register <n>, n = 0 - 3

The TRCCNTRLDVR<n> characteristics are:

**Purpose**

This sets or returns the reload count value for Counter <n>.

**Configurations**

External register TRCCNTRLDVR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCCNTRLDVR<n>[31:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR5.NUMCNTR) > n. Otherwise, direct accesses to TRCCNTRLDVR<n> are RES0.

**Attributes**

TRCCNTRLDVR<n> is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:16]</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>[15:0]</td>
<td>VALUE, bits [15:0]</td>
</tr>
</tbody>
</table>

VALUE, bits [15:0]

Contains the reload value for Counter <n>. When a reload event occurs for Counter <n> then the trace unit copies the VALUE<n> field into Counter <n>.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing the TRCCNTRLDVR<n>:**

Must be programmed if TRCRSCTLR<n>.GROUP == 0b0010 and TRCRSCTLR<n>.COUNTERS[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCNTRLDVR<n> can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x140 + (4 * n)</td>
<td>TRCCNTRLDVR&lt;n&gt;</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.18 TRCCNTVR<n>, Counter Value Register <n>, n = 0 - 3

The TRCCNTVR<n> characteristics are:

Purpose

This sets or returns the value of Counter <n>.

Configurations

External register TRCCNTVR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCCNTVR<n>[31:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR5.NUMCNTR) > n. Otherwise, direct accesses to TRCCNTVR<n> are RES0.

Attributes

TRCCNTVR<n> is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>VALUE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits [31:16]

Reserved, RES0.

VALUE, bits [15:0]

Contains the count value of Counter.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing the TRCCNTVR<n>:

Must be programmed if TRCRSCTLR<n>.GROUP == 0b0010 and TRCRSCTLR<n>.COUNTERS[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCCNTVR<n> can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x160 + (4 * n)</td>
<td>TRCCNTVR&lt;n&gt;</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.19 TRCCONFIGR, Trace Configuration Register

The TRCCONFIGR characteristics are:

**Purpose**

Controls the tracing options.

**Configurations**

External register TRCCONFIGR bits [31:0] are architecturally mapped to AArch64 System register TRCCONFIG[R][31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCCONFIGR are RES0.

**Attributes**

TRCCONFIGR is a 32-bit register.

**Field descriptions**

- **Bits [31:16]**
  - Reserved, RES0.
- **VMIDOPT, bit [15]**
  - When TRCIDR2.VMIDOPT == 0b01:
    - Virtual context identifier selection control.
    - 0b0 VTTBR_EL2.VMID is used as the Virtual context identifier.
    - 0b1 CONTEXTIDR_EL2.PROCID is used as the Virtual context identifier.
  - When TRCIDR2.VMIDOPT == 0b00:
    - Reserved, RES0.
  - Virtual context identifier selection control.
    - VTTBR_EL2.VMID is used as the Virtual context identifier.
  - When TRCIDR2.VMIDOPT == 0b10:
    - Reserved, RES1.
  - Virtual context identifier selection control.
    - CONTEXTIDR_EL2.PROCID is used as the Virtual context identifier.
- **Otherwise:**
  - Reserved, RES0.
- **QE, bits [14:13]**
  - When TRCIDR0.QSUPP == 0b01:
    - Q element generation control.
    - 0b0 Q elements are disabled.
    - 0b1 Q elements with instruction counts are enabled.
    - Q elements without instruction counts are disabled.
  - All other values are reserved.
When TRCIDR0.QSUPP == 0b10:
  Q element generation control.
  0b00 Q elements are disabled.
  0b11 Q elements with instruction counts are enabled.
  Q elements without instruction counts are enabled.
  All other values are reserved.

When TRCIDR0.QSUPP == 0b11:
  Q element generation control.
  0b00 Q elements are disabled.
  0b01 Q elements with instruction counts are enabled.
  Q elements without instruction counts are disabled.
  0b11 Q elements with instruction counts are enabled.
  Q elements without instruction counts are enabled.
  All other values are reserved.

Otherwise:
  Reserved, RES0.

RS, bit [12]
When TRCIDR0.RETSTACK == 1:
  Return stack control.
  0b0 Return stack is disabled.
  0b1 Return stack is enabled.
  The reset behavior of this field is:
  • On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
  Reserved, RES0.

TS, bit [11]
When TRCIDR0.TSSIZE != 0b00000:
  Global timestamp tracing control.
  0b0 Global timestamp tracing is disabled.
  0b1 Global timestamp tracing is enabled.
  The reset behavior of this field is:
  • On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
  Reserved, RES0.

Bits [10:8]
Reserved, RES0.

VMID, bit [7]
When TRCIDR2.VMIDSIZE != 0b00000:
  Virtual context identifier tracing control.
  0b0 Virtual context identifier tracing is disabled.
  0b1 Virtual context identifier tracing is enabled.
  The reset behavior of this field is:
  • On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
Otherwise:

Reserved, RES0.

CID, bit [6]

When TRCIDR2.CIDSIZE != 0b00000:

Context identifier tracing control.

0b0   Context identifier tracing is disabled.
0b1   Context identifier tracing is enabled.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [5]

Reserved, RES0.

CCI, bit [4]

When TRCIDR0.TRCCCI == 1:

Cycle counting instruction tracing control.

0b0   Cycle counting instruction tracing is disabled.
0b1   Cycle counting instruction tracing is enabled.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BB, bit [3]

When TRCIDR0.TRCBB == 1:

Branch broadcasting control.

0b0   Branch broadcasting is disabled.
0b1   Branch broadcasting is enabled.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [2:1]

Reserved, RES0.

Bit [0]

Reserved, RES1.

Accessing the TRCCONFIGR:

Must always be programmed.

TRCCONFIGR.QE must be set to 0b00 if TRCCONFIGR.BB is not 0.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.
TRCCONFIGR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x10</td>
<td>TRCCONFIGR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !IsTraceCorePowered() or !AllowExternalTraceAccess() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
**H9.3.20 TRCDEVAFF, Device Affinity Register**

The TRCDEVAFF characteristics are:

**Purpose**

For additional information, see the CoreSight Architecture Specification.

Reads the same value as the MPIDR_EL1 register for the PE that this trace unit has affinity with.

**Configurations**

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCDEVAFF are RES0.

**Attributes**

TRCDEVAFF is a 64-bit register.

**Field descriptions**

- **MPIDR_EL1, bits [63:0]**
  
  Read-only copy of MPIDR_EL1, as seen from the highest implemented Exception level.

**Accessing the TRCDEVAFF:**

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVAFF can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFA8</td>
<td>TRCDEVAFF</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.21 TRCDEVARCH, Device Architecture Register

The TRCDEVARCH characteristics are:

**Purpose**

Provides discovery information for the component.
For additional information, see the CoreSight Architecture Specification.

**Configurations**

External register TRCDEVARCH bits [31:0] are architecturally mapped to AArch64 System register TRCDEVARCH[31:0].
This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCDEVARCH are RES0.

**Attributes**

TRCDEVARCH is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>21</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARCHITECT</td>
<td>REVISION</td>
<td>ARCHVER</td>
<td>ARCHPART</td>
<td>PRESENT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ARCHITECT, bits [31:21]**

Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code.

0b01000111011 JEP106 continuation code 0x4, ID code 0x3B. Arm Limited.
Other values are defined by the JEDEC JEP106 standard.
This field reads as 0x23B.

**PRESENT, bit [20]**

DEVARCH Present. Defines that the DEVARCH register is present.

0b0 Device Architecture information not present.
0b1 Device Architecture information present.
This field reads as 1.

**REVISION, bits [19:16]**

Revision. Defines the architecture revision of the component.

0b0000 ETEv1.0, FEAT_ETE.
0b0001 ETEv1.1, FEAT_ETEv1p1.
0b0010 ETEv1.2, FEAT_ETEv1p2.
All other values are reserved.

**ARCHVER, bits [15:12]**

Architecture Version. Defines the architecture version of the component.

0b010 ETEv1.
ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].
This field reads as 0x5.
ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.
0xA13 Arm PE trace architecture.
ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].
This field reads as 0xA13.

**Accessing the TRCDEVARCH:**

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVARCH can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFBC</td>
<td>TRCDEVARCH</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.22 TRCDEVID, Device Configuration Register

The TRCDEVID characteristics are:

**Purpose**

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

**Configurations**

External register TRCDEVID bits [31:0] are architecturally mapped to AArch64 System register TRCDEVID[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCDEVID are RES0.

**Attributes**

TRCDEVID is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0</td>
</tr>
</tbody>
</table>

**Bits [31:0]**

Reserved, RES0.

**Accessing the TRCDEVID:**

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVID can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFC8</td>
<td>TRCDEVID</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.23   TRCDEVID1, Device Configuration Register 1

The TRCDEVID1 characteristics are:

**Purpose**

Provides discovery information for the component.
For additional information, see the CoreSight Architecture Specification.

**Configurations**

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCDEVID1 are RES0.

**Attributes**

TRCDEVID1 is a 32-bit register.

**Field descriptions**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | RES0 |

Bits [31:0]

Reserved, RES0.

**Accessing the TRCDEVID1:**

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVID1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFc4</td>
<td>TRCDEVID1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.24 TRCDEVID2, Device Configuration Register 2

The TRCDEVID2 characteristics are:

**Purpose**

Provides discovery information for the component. For additional information, see the CoreSight Architecture Specification.

**Configurations**

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCDEVID2 are RES0.

**Attributes**

TRCDEVID2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>RES0</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**Accessing the TRCDEVID2:**

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVID2 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xF0</td>
<td>TRCDEVID2</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.25 TRCDEVTYPE, Device Type Register

The TRCDEVTYPE characteristics are:

**Purpose**

Provides discovery information for the component. If the part number field is not recognized, a debugger can report the information that is provided by TRCDEVTYPE about the component instead.

For additional information, see the CoreSight Architecture Specification.

**Configurations**

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCDEVTYPE are RES0.

**Attributes**

TRCDEVTYPE is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>0</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>SUB</td>
<td>MAJOR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**SUB, bits [7:4]**

Component sub-type.

- **0b0001**: When MAJOR == 0x3 (Trace source): Associated with a PE. This field reads as 0x1.

**MAJOR, bits [3:0]**

Component major type.

- **0b0011**: Trace source. Other values are defined by the CoreSight Architecture. This field reads as 0x3.

**Accessing the TRCDEVTYPE:**

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVTYPE can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFCC</td>
<td>TRCDEVTYPE</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.26 TRCEVENTCTL0R, Event Control 0 Register

The TRCEVENTCTL0R characteristics are:

Purpose

Controls the generation of ETEEvents.

Configurations

External register TRCEVENTCTL0R bits [31:0] are architecturally mapped to AArch64 System register TRCEVENTCTL0R[31:0].

This register is present only when FEAT_ETE is implemented and TRCIDR4.NUMRSPAIR != 0b0000. Otherwise, direct accesses to TRCEVENTCTL0R are RES0.

Attributes

TRCEVENTCTL0R is a 32-bit register.

Field descriptions

EVENT3_TYPE, bit [31]

When TRCIDR4.NUMRSPAIR != 0b0000 and UInt(TRCIDR0.NUMEVENT) >= 3:

Chooses the type of Resource Selector.

0b0 A single Resource Selector.

TRCEVENTCTL0R.EVENT3.TYPE[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

0b1 A Boolean-combined pair of Resource Selectors.

TRCEVENTCTL0R.EVENT3.TYPE[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT3.TYPE[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [30:29]

Reserved, RES0.

EVENT3_SEL, bits [28:24]

When TRCIDR4.NUMRSPAIR != 0b0000 and UInt(TRCIDR0.NUMEVENT) >= 3:

 Defines the selected Resource Selector or pair of Resource Selectors. TRCEVENTCTL0R.EVENT3.TYPE controls whether TRCEVENTCTL0R.EVENT3_SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.
When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[3] == 1, then Event element 3 is generated in the instruction trace element stream.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

### EVENT2_TYPE, bit [23]

When TRCIDR4.NUMRSPAIR != 0b0000 and UInt(TRCIDR0.NUMEVENT) >= 2:

Chooses the type of Resource Selector.

- 0b0 A single Resource Selector. TRCEVENTCTL0R.EVENT2.TYPE selects the single Resource Selector, from 0-31, used to activate the resource event.

- 0b1 A Boolean-combined pair of Resource Selectors. TRCEVENTCTL0R.EVENT2.TYPE selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT2_SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

Bits [22:21]

Reserved, RES0.

### EVENT2_SEL, bits [20:16]

When TRCIDR4.NUMRSPAIR != 0b0000 and UInt(TRCIDR0.NUMEVENT) >= 2:

Defines the selected Resource Selector or pair of Resource Selectors. TRCEVENTCTL0R.EVENT2_SEL controls whether TRCEVENTCTL0R.EVENT2_SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[2] == 1, then Event element 2 is generated in the instruction trace element stream.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

### EVENT1_TYPE, bit [15]

When TRCIDR4.NUMRSPAIR != 0b0000 and UInt(TRCIDR0.NUMEVENT) >= 1:

Chooses the type of Resource Selector.

- 0b0 A single Resource Selector. TRCEVENTCTL0R.EVENT1_SEL[4] selects the single Resource Selector, from 0-31, used to activate the resource event.

- 0b1 A Boolean-combined pair of Resource Selectors.
TRCEVENTCTL0R.EVENT1.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT1.SEL[4] is RES0.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [14:13]
Reserved, RES0.

EVENT1_SEL, bits [12:8]

When TRCIDR4.NUMRSPAIR != 0b0000 and UInt(TRCIDR0.NUMEVENT) >= 1:

Defines the selected Resource Selector or pair of Resource Selectors.

TRCEVENTCTL0R.EVENT1.TYPE controls whether TRCEVENTCTL0R.EVENT1.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[1] == 1, then Event element 1 is generated in the instruction trace element stream.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EVENT0_TYPE, bit [7]

When TRCIDR4.NUMRSPAIR != 0b0000:

Chooses the type of Resource Selector.

0b0 A single Resource Selector.

TRCEVENTCTL0R.EVENT0.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

0b1 A Boolean-combined pair of Resource Selectors.

TRCEVENTCTL0R.EVENT0.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT0.SEL[4] is RES0.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [6:5]
Reserved, RES0.

EVENT0_SEL, bits [4:0]

When TRCIDR4.NUMRSPAIR != 0b0000:

Defines the selected Resource Selector or pair of Resource Selectors.

TRCEVENTCTL0R.EVENT0.TYPE controls whether TRCEVENTCTL0R.EVENT0.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.
If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[0] == 1, then Event element 0 is generated in the instruction trace element stream.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Accessing the TRCEVENTCTL0R:**

Must be programmed if implemented.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCEVENTCTL0R can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x020</td>
<td>TRCEVENTCTL0R</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.27 TRCEVENTCTL1R, Event Control 1 Register

The TRCEVENTCTL1R characteristics are:

**Purpose**

Controls the behavior of the ETEEvents that TRCEVENTCTL0R selects.

**Configurations**

External register TRCEVENTCTL1R bits [31:0] are architecturally mapped to AArch64 System register TRCEVENTCTL1R[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCEVENTCTL1R are RES0.

**Attributes**

TRCEVENTCTL1R is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>30</td>
<td>LPOVERRIDE</td>
<td>Low-power Override Mode select.</td>
</tr>
<tr>
<td>12</td>
<td>When TRCIDR5.LPOVERRIDE == 1:</td>
<td>Trace unit Low-power Override Mode is not enabled. That is, the trace unit is permitted to enter low-power state.</td>
</tr>
<tr>
<td>11</td>
<td>ATB</td>
<td>AMBA Trace Bus (ATB) trigger enable.</td>
</tr>
<tr>
<td></td>
<td>When TRCIDR5.ATBTRIG == 1:</td>
<td>AMBA Trace Bus (ATB) trigger enable.</td>
</tr>
</tbody>
</table>

When TRCIDR5.LPOVERRIDE == 1:

- **LPOVERRIDE**
  - **0b0**: Trace unit Low-power Override Mode is not enabled. That is, the trace unit is permitted to enter low-power state.
  - **0b1**: Trace unit Low-power Override Mode is enabled. That is, entry to a low-power state does not affect the trace unit resources or trace generation.


When TRCIDR5.ATBTRIG == 1:

- **ATB**
  - AMBA Trace Bus (ATB) trigger enable.

If a CoreSight ATB interface is implemented then when ETEEvent 0 occurs the trace unit sets:

- **ATID** = 0x7D.
- **ATDATA** to the value of TRCTRACEIDR.

If the width of ATDATA is greater than the width of TRCTRACEIDR.TRACEID then the trace unit zeros the upper ATDATA bits.

If ETEEvent 0 is programmed to occur based on program execution, such as an Address Comparator, the ATB trigger might not be inserted into the ATB stream at the same time as any trace generated by that program execution is output by the trace unit. Typically, the generated trace might be buffered in a trace unit which means that the ATB trigger would be output before the associated trace is output.
If ETEEvent 0 is asserted multiple times in close succession, the trace unit is required to generate an ATB trigger for the first assertion, but might ignore one or more of the subsequent assertions. Arm recommends that the window in which ETEEvent 0 is ignored is limited only by the time taken to output an ATB trigger.

- 0b0: ATB trigger is disabled.
- 0b1: ATB trigger is enabled.

**Otherwise:**
- Reserved, RES0.

**Bits [10:4]**
- Reserved, RES0.

**INSTEN[_m_], bit [m], for m = 3 to 0**
- Event element control.
  - 0b0: The trace unit does not generate an Event element <m>.
  - 0b1: The trace unit generates an Event element <m>.

This bit is RES0 if m >= the number indicated by TRCIDR0.NUMEVENT.

**Accessing the TRCEVENTCTL1R:**

Must be programmed.

Writes are **CONSTRAINED UNPREDICTABLE** if the trace unit is not in the Idle state.

TRCEVENTCTL1R can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x824</td>
<td>TRCEVENTCTL1R</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSlockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.28 TRCEXTINSELR<n>, External Input Select Register <n>, n = 0 - 3

The TRCEXTINSELR<n> characteristics are:

**Purpose**

Use this to set, or read, which External Inputs are resources to the trace unit.

The name TRCEXTINSELR is an alias of TRCEXTINSELR0.

**Configurations**

External register TRCEXTINSELR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCEXTINSELR<n>[31:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR5.NUMEXTINSEL) > n. Otherwise, direct accesses to TRCEXTINSELR<n> are RES0.

**Attributes**

TRCEXTINSELR<n> is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>evtCount</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:16]**

Reserved, RES0.

**evtCount, bits [15:0]**

PMU event to select.

The event number as defined by the Arm ARM.

Software must program this field with a PMU event that is supported by the PE being programmed.

There are three ranges of PMU event numbers:

- PMU event numbers in the range 0x0000 to 0x003F are common architectural and microarchitectural events.
- PMU event numbers in the range 0x0040 to 0x00FF are Arm recommended common architectural and microarchitectural PMU events.
- PMU event numbers in the range 0x00C0 to 0x03FF are IMPLEMENTATION DEFINED PMU events.

If evtCount is programmed to a PMU event that is reserved or not supported by the PE, the behavior depends on the PMU event type:

- For the range 0x0000 to 0x003F, then the PMU event is not active, and the value returned by a direct or external read of the evtCount field is the value written to the field.
- For IMPLEMENTATION DEFINED PMU events, it is UNPREDICTABLE what PMU event, if any, is counted, and the value returned by a direct or external read of the evtCount field is UNKNOWN.

UNPREDICTABLE means the PMU event must not expose privileged information.

Arm recommends that the behavior across a family of implementations is defined such that if a given implementation does not include a PMU event from a set of common IMPLEMENTATION DEFINED PMU events, then no PMU event is counted and the value read back on evtCount is the value written.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
Accessing the TRCEXTINSELR\langle n\rangle:

Must be programmed if any of the following is true: TRCRSCTRLR\langle n\rangle.GROUP == 0b0000 and TRCRSCTRLR\langle n\rangle.EXTIN[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCEXTINSELR\langle n\rangle can be accessed through the external debug interface:

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x120 + (4 \times n)</td>
<td>TRCEXTINSELR\langle n\rangle</td>
</tr>
</tbody>
</table>
H9.3.29 TRCIDR0, ID Register 0

The TRCIDR0 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

External register TRCIDR0 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR0[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR0 are RES0.

**Attributes**

TRCIDR0 is a 32-bit register.

**Field descriptions**

![Diagram of TRCIDR0 register]

**Bit [31]**

Reserved, RES0.

**COMMTRANS, bit [30]**

Transaction Start element behavior.

- 0b0: Transaction Start elements are P0 elements.
- 0b1: Transaction Start elements are not P0 elements.

**COMMOPT, bit [29]**

Indicates the contents and encodings of Cycle count packets.

- 0b0: Commit mode 0.
- 0b1: Commit mode 1.

The Commit mode defines the contents and encodings of Cycle Count packets, in particular how Commit elements are indicated by these packets. See the descriptions of these packets for more details.

Accessing this field has the following behavior:

- RAO/WI if all of the following are true:
  - TRCIDR0.TRCCCI == 1
  - UInt(TRCIDR8.MAXSPEC) == 0x0
- When TRCIDR0.TRCCCI == 0, access to this field is RAZ/WI.
- Otherwise, access to this field is RO
TSSIZE, bits [28:24]

Indicates that the trace unit implements Global timestamping and the size of the timestamp value.

- **0b00000**: Global timestamping not implemented.
- **0b01000**: Global timestamping implemented with a 64-bit timestamp value.

All other values are reserved.

This field reads as **0b01000**.

TSMARK, bit [23]

*When FEAT_ETEv1p1 is implemented:*

Indicates whether Timestamp Marker elements are generated.

- **0b0**: Timestamp Marker elements are not generated.
- **0b1**: Timestamp Marker elements are generated.

*Otherwise:*

Reserved, RES0.

Bits [22:18]

Reserved, RES0.

TRCEXDATA, bit [17]

*When TRCIDR0.TRCDATA != 0b00:*

Indicates if the trace unit implements tracing of data transfers for exceptions and exception returns. Data tracing is not implemented in ETE and this field is reserved for other trace architectures.

- **0b0**: Tracing of data transfers for exceptions and exception returns not implemented.
- **0b1**: Tracing of data transfers for exceptions and exception returns implemented.

*Otherwise:*

Reserved, RES0.

QSUPP, bits [16:15]

Indicates that the trace unit implements Q element support.

- **0b00**: Q element support is not implemented.
- **0b01**: Q element support is implemented, and only supports Q elements with instruction counts.
- **0b10**: Q element support is implemented, and only supports Q elements without instruction counts.
- **0b11**: Q element support is implemented, and supports:
  - Q elements with instruction counts.
  - Q elements without instruction counts.

QFILT, bit [14]

Indicates if the trace unit implements Q element filtering.

- **0b0**: Q element filtering is not implemented.
- **0b1**: Q element filtering is implemented.

If TRCIDR0.QSUPP == 0b00 then this field is 0.
CONDTYPE, bits [13:12]

When TRCIDR0.TRCCOND == 1:
Indicates how conditional instructions are traced. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

0b00  Conditional instructions are traced with an indication of whether they pass or fail their condition code check.
0b01  Conditional instructions are traced with an indication of the APSR condition flags.
All other values are reserved.

Otherwise:
Reserved, RES0.

NUMEVENT, bits [11:10]

When TRCIDR4.NUMRSPAIR == 0b0000:
Indicates the number of ETEEvents implemented.
0b00  The trace unit supports 0 ETEEvents.
All other values are reserved.

When TRCIDR4.NUMRSPAIR != 0b0000:
Indicates the number of ETEEvents implemented.
0b00  The trace unit supports 1 ETEEvent.
0b01  The trace unit supports 2 ETEEvents.
0b10  The trace unit supports 3 ETEEvents.
0b11  The trace unit supports 4 ETEEvents.

Otherwise:
Reserved, RES0.

RETSTACK, bit [9]
Indicates if the trace unit supports the return stack.
0b0  Return stack not implemented.
0b1  Return stack implemented.

Bit [8]
Reserved, RES0.

TRCCCI, bit [7]
Indicates if the trace unit implements cycle counting.
0b0  Cycle counting not implemented.
0b1  Cycle counting implemented.
This field reads as 1.

TRCCOND, bit [6]
Indicates if the trace unit implements conditional instruction tracing. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures.
0b0  Conditional instruction tracing not implemented.
0b1  Conditional instruction tracing implemented.
This field reads as 0.

TRCBB, bit [5]
Indicates if the trace unit implements branch broadcasting.
0b0  Branch broadcasting not implemented.
0b1       Branch broadcasting implemented.
This field reads as 1.

**TRCDATA, bits [4:3]**
Indicates if the trace unit implements data tracing. Data tracing is not implemented in ETE and this
field is reserved for other trace architectures.
0b00       Data tracing not implemented.
0b11       Data tracing implemented.
All other values are reserved.
This field reads as 0b00.

**INSTP0, bits [2:1]**
Indicates if load and store instructions are P0 instructions. Load and store instructions as P0
instructions is not implemented in ETE and this field is reserved for other trace architectures.
0b00       Load and store instructions are not P0 instructions.
0b11       Load and store instructions are P0 instructions.
All other values are reserved.
This field reads as 0b00.

**Bit [0]**
Reserved, RES1.

**Accessing the TRCIDR0:**
TRCIDR0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x1E0</td>
<td>TRCIDR0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.30 TRCIDR1, ID Register 1

The TRCIDR1 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

External register TRCIDR1 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR1[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR1 are RES0.

**Attributes**

TRCIDR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>24</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESIGNER</td>
<td>RES0</td>
<td>RES1</td>
<td>REVISION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**DESIGNER, bits [31:24]**

Indicates which company designed the trace unit. The permitted values of this field are the same as MIDR_EL1.Implementer.

**Bits [23:16]**

Reserved, RES0.

**Bits [15:12]**

Reserved, RES1.

**TRCARCHMAJ, bits [11:8]**

Major architecture version.

0b1111 If both TRCARCHMAJ and TRCARCHMIN == 0xF then refer to TRCDEVARCH.

All other values are reserved.

This field reads as 0b1111.

**TRCARCHMIN, bits [7:4]**

Minor architecture version.

0b1111 If both TRCARCHMAJ and TRCARCHMIN == 0xF then refer to TRCDEVARCH.

All other values are reserved.

This field reads as 0b1111.

**REVISION, bits [3:0]**

Implementation revision.

Returns an IMPLEMENTATION DEFINED value that identifies the revision of the trace unit.

Arm deprecates any use of this field and recommends that implementations set this field to zero.
**Accessing the TRCIDR1:**

TRCIDR1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x1E4</td>
<td>TRCIDR1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.31 TRCIDR10, ID Register 10

The TRCIDR10 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configurations

External register TRCIDR10 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR10[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR10 are RES0.

Attributes

TRCIDR10 is a 32-bit register.

Field descriptions

NUMP1KEY, bits [31:0]

When TRCIDR0.TRCDATA != 0b00:

Indicates the number of P1 right-hand keys. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

Otherwise:

Reserved, RES0.

Accessing the TRCIDR10:

TRCIDR10 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x188</td>
<td>TRCIDR10</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.32 TRCIDR11, ID Register 11

The TRCIDR11 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

External register TRCIDR11 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR11[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR11 are RES0.

**Attributes**

TRCIDR11 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMP1SPC</td>
<td>bits [31:0]</td>
</tr>
</tbody>
</table>

When TRCIDR0.TRCDATA ! = 0b00:

Indicates the number of special P1 right-hand keys. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

Otherwise:

Reserved, RES0.

**Accessing the TRCIDR11:**

TRCIDR11 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x18C</td>
<td>TRCIDR11</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.33 TRCIDR12, ID Register 12

The TRCIDR12 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

External register TRCIDR12 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR12[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR12 are RES0.

**Attributes**

TRCIDR12 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th></th>
<th>NUMCONDKEY</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

NUMCONDKEY, bits [31:0]

*When TRCIDR0.TRCCOND == 1:*

Indicates the number of conditional instruction right-hand keys. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

*Otherwise:*

Reserved, RES0.

**Accessing the TRCIDR12:**

TRCIDR12 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x190</td>
<td>TRCIDR12</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.34 TRCIDR13, ID Register 13

The TRCIDR13 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

External register TRCIDR13 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR13[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR13 are RES0.

**Attributes**

TRCIDR13 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMCONDSPC</td>
<td>Indicates the number of special conditional instruction right-hand keys.</td>
</tr>
<tr>
<td></td>
<td>Conditional instruction tracing is not implemented in ETE and this field is</td>
</tr>
<tr>
<td></td>
<td>reserved for other trace architectures. Allocated in other trace architectures.</td>
</tr>
</tbody>
</table>

When TRCIDR0.TRCCOND == 1:

Indicates the number of special conditional instruction right-hand keys. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

**Otherwise:**

Reserved, RES0.

**Accessing the TRCIDR13:**

TRCIDR13 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x194</td>
<td>TRCIDR13</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.35   TRCIDR2, ID Register 2

The TRCIDR2 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configurations

External register TRCIDR2 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR2[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR2 are RES0.

Attributes

TRCIDR2 is a 32-bit register.

Field descriptions

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| WFXMODE | VMIDOPT |

WFXMODE, bit [31]

Indicates whether WFI, WFIT, WFE, and WFET instructions are classified as P0 instructions:

- 0b0: WFI, WFIT, WFE, and WFET instructions are not classified as P0 instructions.
- 0b1: WFI, WFIT, WFE, and WFET instructions are classified as P0 instructions.

VMIDOPT, bits [30:29]

Indicates the options for Virtual context identifier selection.

- 0b00: Virtual context identifier selection not supported. TRCCONFIGR.VMIDOPT is RES0.
- 0b01: Virtual context identifier selection supported. TRCCONFIGR.VMIDOPT is implemented.
- 0b10: Virtual context identifier selection not supported. TRCCONFIGR.VMIDOPT is RES1.

All other values are reserved.

If TRCIDR2.VMIDSIZE == 0b00000 then this field is 0b00.

If TRCIDR2.VMIDSIZE != 0b00000 then this field is 0b10.

CCSIZE, bits [28:25]

Indicates the size of the cycle counter.

When TRCIDR0.TRCCCI == 1:

- 0b0000: The cycle counter is 12 bits in length.
- 0b0001: The cycle counter is 13 bits in length.
- 0b0010: The cycle counter is 14 bits in length.
- 0b0011: The cycle counter is 15 bits in length.
- 0b0100: The cycle counter is 16 bits in length.
- 0b0101: The cycle counter is 17 bits in length.
- 0b0110: The cycle counter is 18 bits in length.
- 0b0111: The cycle counter is 19 bits in length.
- 0b1000: The cycle counter is 20 bits in length.
All other values are reserved.

**Otherwise:**
Reserved, RES0.

**DVSIZE, bits [24:20]**

*When TRCIDR0.TRCDATA != 0b00:*
Indicates the data value size in bytes. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.
- 0b00000: Data value tracing not implemented.
- 0b00100: Data value tracing has a maximum of 32-bit data values.
- 0b01000: Data value tracing has a maximum of 64-bit data values.
All other values are reserved.

**Otherwise:**
Reserved, RES0.

**DASIZE, bits [19:15]**

*When TRCIDR0.TRCDATA != 0b00:*
Indicates the data address size in bytes. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.
- 0b00000: Data address tracing not implemented.
- 0b00100: Data address tracing has a maximum of 32-bit data addresses.
- 0b01000: Data address tracing has a maximum of 64-bit data addresses.
All other values are reserved.

**Otherwise:**
Reserved, RES0.

**VMIDSIZE, bits [14:10]**
Indicates the trace unit Virtual context identifier size.
- 0b00000: Virtual context identifier tracing is not supported.
- 0b00001: 8-bit Virtual context identifier size.
- 0b00010: 16-bit Virtual context identifier size.
- 0b00100: 32-bit Virtual context identifier size.
All other values are reserved.
If the PE does not implement EL2 then this field is 0b00000.
If the PE implements EL2 then this field is 0b00100.

**CIDSIZE, bits [9:5]**
Indicates the Context identifier size.
- 0b00000: Context identifier tracing is not supported.
- 0b00100: 32-bit Context identifier size.
All other values are reserved.
This field reads as 0b00100.

**IASIZE, bits [4:0]**
Virtual instruction address size.
- 0b00100: Maximum of 32-bit instruction address size.
- 0b01000: Maximum of 64-bit instruction address size.
All other values are reserved.
This field reads as 0b01000.

**Accessing the TRCIDR2:**

TRCIDR2 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x1E8</td>
<td>TRCIDR2</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.36 TRCIDR3, ID Register 3

The TRCIDR3 characteristics are:

**Purpose**

Returns the base architecture of the trace unit.

**Configurations**

External register TRCIDR3 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR3[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR3 are RES0.

**Attributes**

TRCIDR3 is a 32-bit register.

**Field descriptions**

NOOVERFLOW, bit [31]

Indicates if overflow prevention is implemented.

- 0b0: Overflow prevention is not implemented.
- 0b1: Overflow prevention is implemented.

If TRCIDR3.STALLCTL == 0 then this field is 0.

NUMPROC, bits [13:12, 30:28]

Indicates the number of PEs available for tracing.

- 0b00000: The trace unit can trace one PE.

This field reads as 0b00000.

The NUMPROC field is split as follows:

- NUMPROC[2:0] is TRCIDR3[30:28].
- NUMPROC[4:3] is TRCIDR3[13:12].

SYSSTALL, bit [27]

Indicates if stalling of the PE is permitted.

- 0b0: Stalling of the PE is not permitted.
- 0b1: Stalling of the PE is permitted.

The value of this field might be dynamic and change based on system conditions.

If TRCIDR3.STALLCTL == 0 then this field is 0.
STALLCTL, bit [26]
Indicates if trace unit implements stalling of the PE.
0b0  Stalling of the PE is not implemented.
0b1  Stalling of the PE is implemented.

SYNCR, bit [25]
Indicates if an implementation has a fixed synchronization period.
0b0  TRCSYNCR is read/write so software can change the synchronization period.
0b1  TRCSYNCR is read-only so the synchronization period is fixed.
This field reads as 0.

TRCERR, bit [24]
Indicates forced tracing of System Error exceptions is implemented.
0b0  Forced tracing of System Error exceptions is not implemented.
0b1  Forced tracing of System Error exceptions is implemented.
This field reads as 1.

Bit [23]
Reserved, RES0.

EXLEVEL_NS_EL2, bit [22]
Indicates if Non-secure EL2 is implemented.
0b0  Non-secure EL2 is not implemented.
0b1  Non-secure EL2 is implemented.

EXLEVEL_NS_EL1, bit [21]
Indicates if Non-secure EL1 is implemented.
0b0  Non-secure EL1 is not implemented.
0b1  Non-secure EL1 is implemented.

EXLEVEL_NS_EL0, bit [20]
Indicates if Non-secure EL0 is implemented.
0b0  Non-secure EL0 is not implemented.
0b1  Non-secure EL0 is implemented.

EXLEVEL_S_EL3, bit [19]
Indicates if EL3 is implemented.
0b0  EL3 is not implemented.
0b1  EL3 is implemented.

EXLEVEL_S_EL2, bit [18]
Indicates if Secure EL2 is implemented.
0b0  Secure EL2 is not implemented.
0b1  Secure EL2 is implemented.

EXLEVEL_S_EL1, bit [17]
Indicates if Secure EL1 is implemented.
0b0  Secure EL1 is not implemented.
0b1  Secure EL1 is implemented.
**EXLEVEL_S_EL0, bit 16**

Indicates if Secure EL0 is implemented.
- `0b0` Secure EL0 is not implemented.
- `0b1` Secure EL0 is implemented.

**Bits [15:14]**

Reserved, RES0.

**CCITMIN, bits [11:0]**

Indicates the minimum value that can be programmed in TRCCCTRLR.THRESHOLD.
- If TRCIDR0.TRCCCI == 1 then the minimum value of this field is 0x001.
- If TRCIDR0.TRCCCI == 0 then this field is zero.

**Accessing the TRCIDR3:**

TRCIDR3 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x1EC</td>
<td>TRCIDR3</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.37 TRCIDR4, ID Register 4

The TRCIDR4 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

External register TRCIDR4 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR4[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR4 are RES0.

**Attributes**

TRCIDR4 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>NUMVMIDC</th>
<th>NUMCIDC</th>
<th>NUMSSCC</th>
<th>NUMPC</th>
<th>RES0</th>
<th>NUMDVC</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 28 23 20 16 15 12 11 9 8 7 6 3 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NUMRPAIR</td>
<td>SUPPDA</td>
<td>NUMACPAIRS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NUMVMIDC, bits [31:28]**

Indicates the number of Virtual Context Identifier Comparators that are available for tracing.

- 00000: No Virtual Context Identifier Comparators are available.
- 00001: The implementation has one Virtual Context Identifier Comparator.
- 00010: The implementation has two Virtual Context Identifier Comparators.
- 00011: The implementation has three Virtual Context Identifier Comparators.
- 00100: The implementation has four Virtual Context Identifier Comparators.
- 00101: The implementation has five Virtual Context Identifier Comparators.
- 00110: The implementation has six Virtual Context Identifier Comparators.
- 00111: The implementation has seven Virtual Context Identifier Comparators.
- 01000: The implementation has eight Virtual Context Identifier Comparators.

All other values are reserved.

**NUMCIDC, bits [27:24]**

Indicates the number of Context Identifier Comparators that are available for tracing.

- 00000: No Context Identifier Comparators are available.
- 00001: The implementation has one Context Identifier Comparator.
- 00010: The implementation has two Context Identifier Comparators.
- 00011: The implementation has three Context Identifier Comparators.
- 00100: The implementation has four Context Identifier Comparators.
- 00101: The implementation has five Context Identifier Comparators.
- 00110: The implementation has six Context Identifier Comparators.
- 00111: The implementation has seven Context Identifier Comparators.
- 01000: The implementation has eight Context Identifier Comparators.

All other values are reserved.
NUMSSCC, bits [23:20]
Indicates the number of Single-shot Comparator Controls that are available for tracing.
0b0000 No Single-shot Comparator Controls are available.
0b0001 The implementation has one Single-shot Comparator Control.
0b0010 The implementation has two Single-shot Comparator Controls.
0b0011 The implementation has three Single-shot Comparator Controls.
0b0100 The implementation has four Single-shot Comparator Controls.
0b0101 The implementation has five Single-shot Comparator Controls.
0b0110 The implementation has six Single-shot Comparator Controls.
0b0111 The implementation has seven Single-shot Comparator Controls.
0b1000 The implementation has eight Single-shot Comparator Controls.
All other values are reserved.

NUMRSPAIR, bits [19:16]
Indicates the number of resource selector pairs that are available for tracing.
0b0000 The implementation has zero resource selectors.
0b0001 The implementation has two resource selector pairs.
0b0010 The implementation has three resource selector pairs.
0b0011 The implementation has four resource selector pairs.
0b0100 The implementation has five resource selector pairs.
0b0101 The implementation has six resource selector pairs.
0b0110 The implementation has seven resource selector pairs.
0b0111 The implementation has eight resource selector pairs.
0b1000 The implementation has nine resource selector pairs.
0b1001 The implementation has ten resource selector pairs.
0b1010 The implementation has eleven resource selector pairs.
0b1011 The implementation has twelve resource selector pairs.
0b1100 The implementation has thirteen resource selector pairs.
0b1101 The implementation has fourteen resource selector pairs.
0b1110 The implementation has fifteen resource selector pairs.
0b1111 The implementation has sixteen resource selector pairs.
All other values are reserved.

NUMPC, bits [15:12]
Indicates the number of PE Comparator Inputs that are available for tracing.
0b0000 No PE Comparator Inputs are available.
0b0001 The implementation has one PE Comparator Input.
0b0010 The implementation has two PE Comparator Inputs.
0b0011 The implementation has three PE Comparator Inputs.
0b0100 The implementation has four PE Comparator Inputs.
0b0101 The implementation has five PE Comparator Inputs.
0b0110 The implementation has six PE Comparator Inputs.
0b0111 The implementation has seven PE Comparator Inputs.
0b1000 The implementation has eight PE Comparator Inputs.
All other values are reserved.
Bits [11:9]

Reserved, RES0.

SUPPDAC, bit [8]

When TRCIDR4.NUMACPAIRS \neq 0b0000:

Indicates whether data address comparisons are implemented. Data address comparisons are not
implemented in ETE and are reserved for other trace architectures. Allocated in other trace
architectures.

0b0 Data address comparisons not implemented.
0b1 Data address comparisons implemented.

This field reads as 0.

Otherwise:

Reserved, RES0.

NUMDVC, bits [7:4]

Indicates the number of data value comparators. Data value comparators are not implemented in
ETE and are reserved for other trace architectures. Allocated in other trace architectures.

0b0000 No data value comparators implemented.
0b0001 One data value comparator implemented.
0b0010 Two data value comparators implemented.
0b0011 Three data value comparators implemented.
0b0100 Four data value comparators implemented.
0b0101 Five data value comparators implemented.
0b0110 Six data value comparators implemented.
0b0111 Seven data value comparators implemented.
0b1000 Eight data value comparators implemented.

All other values are reserved.

This field reads as 0b0000.

NUMACPAIRS, bits [3:0]

Indicates the number of Address Comparator pairs that are available for tracing.

0b0000 No Address Comparator pairs are available.
0b0001 The implementation has one Address Comparator pair.
0b0010 The implementation has two Address Comparator pairs.
0b0011 The implementation has three Address Comparator pairs.
0b0100 The implementation has four Address Comparator pairs.
0b0101 The implementation has five Address Comparator pairs.
0b0110 The implementation has six Address Comparator pairs.
0b0111 The implementation has seven Address Comparator pairs.
0b1000 The implementation has eight Address Comparator pairs.

All other values are reserved.
Accessing the TRCIDR4:

TRCIDR4 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x1F0</td>
<td>TRCIDR4</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
### H9.3.38 TRCIDR5, ID Register 5

The TRCIDR5 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

External register TRCIDR5 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR5[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR5 are RES0.

**Attributes**

TRCIDR5 is a 32-bit register.

**Field descriptions**

![Field descriptions diagram]

- **Bit [31]**
  - Reserved, RES0.

- **NUMCNTR, bits [30:28]**
  - Indicates the number of Counters that are available for tracing.
  - 0b000 No Counters are available.
  - 0b001 One Counter implemented.
  - 0b010 Two Counters implemented.
  - 0b011 Three Counters implemented.
  - 0b100 Four Counters implemented.
  - All other values are reserved.
  - If TRCIDR4.NUMRSPAIR == 0b0000 then this field is 0b000.

- **NUMSEQSTATE, bits [27:25]**
  - Indicates if the Sequencer is implemented and the number of Sequencer states that are implemented.
  - 0b000 The Sequencer is not implemented.
  - 0b100 Four Sequencer states are implemented.
  - All other values are reserved.
  - If TRCIDR4.NUMRSPAIR == 0b0000 then this field is 0b000.

- **Bit [24]**
  - Reserved, RES0.

- **LPOVERRIDE, bit [23]**
  - Indicates support for Low-power Override Mode.
  - 0b0 The trace unit does not support Low-power Override Mode.
  - 0b1 The trace unit supports Low-power Override Mode.
ATBTRIG, bit [22]
Indicates if the implementation can support ATB triggers.
0b0    The implementation does not support ATB triggers.
0b1    The implementation supports ATB triggers.
If TRCIDR4.NUMRSPAIR == 0b0000 then this field is 0.

TRACEIDSIZE, bits [21:16]
Indicates the trace ID width.
0b000000    The external trace interface is not implemented.
0b000111    The implementation supports a 7-bit trace ID.
All other values are reserved.
Note that AMBA ATB requires a 7-bit trace ID width.

Bits [15:12]
Reserved, RES0.

NUMEXTINSEL, bits [11:9]
Indicates how many External Input Selector resources are implemented.
0b000    No External Input Selector resources are available.
0b001    1 External Input Selector resource is available.
0b010    2 External Input Selector resources are available.
0b011    3 External Input Selector resources are available.
0b100    4 External Input Selector resources are available.
All other values are reserved.

NUMEXTIN, bits [8:0]
Indicates how many External Inputs are implemented.
0b111111111    Unified PMU event selection.
All other values are reserved.

Accessing the TRCIDR5:
TRCIDR5 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xF4</td>
<td>TRCIDR5</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.39 TRCIDR6, ID Register 6

The TRCIDR6 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

External register TRCIDR6 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR6[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR6 are RES0.

**Attributes**

TRCIDR6 is a 32-bit register.

**Field descriptions**

![Field diagram]

*Bits [31:3]*

Reserved, RES0.

*EXLEVEL_RL_EL2, bit [2]*

Indicates if Realm EL2 is implemented.

- **0b0**: Realm EL2 is not implemented.
- **0b1**: Realm EL2 is implemented.

*EXLEVEL_RL_EL1, bit [1]*

Indicates if Realm EL1 is implemented.

- **0b0**: Realm EL1 is not implemented.
- **0b1**: Realm EL1 is implemented.

*EXLEVEL_RL_EL0, bit [0]*

Indicates if Realm EL0 is implemented.

- **0b0**: Realm EL0 is not implemented.
- **0b1**: Realm EL0 is implemented.

**Accessing the TRCIDR6:**

TRCIDR6 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xF8</td>
<td>TRCIDR6</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.40 TRCIDR7, ID Register 7

The TRCIDR7 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configurations

External register TRCIDR7 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR7[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR7 are RES0.

Attributes

TRCIDR7 is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0</td>
</tr>
</tbody>
</table>

Accessing the TRCIDR7:

TRCIDR7 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x1FC</td>
<td>TRCIDR7</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.41 TRCIDR8, ID Register 8

The TRCIDR8 characteristics are:

**Purpose**

Returns the maximum speculation depth of the instruction trace element stream.

**Configurations**

External register TRCIDR8 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR8[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR8 are RES0.

**Attributes**

TRCIDR8 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXSPEC</td>
<td></td>
</tr>
</tbody>
</table>

**MAXSPEC, bits [31:0]**

Indicates the maximum speculation depth of the instruction trace element stream. This is the maximum number of P0 elements in the trace element stream that can be speculative at any time.

**Accessing the TRCIDR8:**

TRCIDR8 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x180</td>
<td>TRCIDR8</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.42 TRCIDR9, ID Register 9

The TRCIDR9 characteristics are:

**Purpose**

Returns the tracing capabilities of the trace unit.

**Configurations**

External register TRCIDR9 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR9[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIDR9 are RES0.

**Attributes**

TRCIDR9 is a 32-bit register.

**Field descriptions**

NUMP0KEY, bits [31:0]

*When TRCIDR0.TRCDATA != 0b00:*

Indicates the number of P0 right-hand keys. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

*Otherwise:*

Reserved, RES0.

**Accessing the TRCIDR9:**

TRCIDR9 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x184</td>
<td>TRCIDR9</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.43 TRCIMSPEC0, IMP DEF Register 0

The TRCIMSPEC0 characteristics are:

**Purpose**

TRCIMSPEC0 shows the presence of any IMPLEMENTATION DEFINED features, and provides an interface to enable the features that are provided.

**Configurations**

External register TRCIMSPEC0 bits [31:0] are architecturally mapped to AArch64 System register TRCIMSPEC0[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCIMSPEC0 are RES0.

**Attributes**

TRCIMSPEC0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>EN</td>
<td>Enable. Controls whether the IMPLEMENTATION DEFINED features are enabled.</td>
</tr>
<tr>
<td>SUPPORT</td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On a Trace unit reset, this field resets to 0.</td>
</tr>
</tbody>
</table>

When TRCIMSPEC0.SUPPORT != 0b0000:

Enable. Controls whether the IMPLEMENTATION DEFINED features are enabled.

- 0b0000 The IMPLEMENTATION DEFINED features are not enabled. The trace unit must behave as if the IMPLEMENTATION DEFINED features are not supported.
- 0b0001 The trace unit behavior is IMPLEMENTATION DEFINED.
- 0b0010 The trace unit behavior is IMPLEMENTATION DEFINED.
- 0b0011 The trace unit behavior is IMPLEMENTATION DEFINED.
- 0b0100 The trace unit behavior is IMPLEMENTATION DEFINED.
- 0b0101 The trace unit behavior is IMPLEMENTATION DEFINED.
- 0b0110 The trace unit behavior is IMPLEMENTATION DEFINED.
- 0b0111 The trace unit behavior is IMPLEMENTATION DEFINED.
- 0b1000 The trace unit behavior is IMPLEMENTATION DEFINED.
- 0b1001 The trace unit behavior is IMPLEMENTATION DEFINED.
- 0b1010 The trace unit behavior is IMPLEMENTATION DEFINED.
- 0b1011 The trace unit behavior is IMPLEMENTATION DEFINED.
- 0b1100 The trace unit behavior is IMPLEMENTATION DEFINED.
- 0b1101 The trace unit behavior is IMPLEMENTATION DEFINED.
- 0b1110 The trace unit behavior is IMPLEMENTATION DEFINED.
- 0b1111 The trace unit behavior is IMPLEMENTATION DEFINED.
Otherwise:

Reserved, RES0.

**SUPPORT, bits [3:0]**

Indicates whether the implementation supports IMPLEMENTATION DEFINED features.

- **0b0000** No IMPLEMENTATION DEFINED features are supported.
- **0b0001** IMPLEMENTATION DEFINED features are supported.
- **0b0010** IMPLEMENTATION DEFINED features are supported.
- **0b0011** IMPLEMENTATION DEFINED features are supported.
- **0b0100** IMPLEMENTATION DEFINED features are supported.
- **0b0101** IMPLEMENTATION DEFINED features are supported.
- **0b0110** IMPLEMENTATION DEFINED features are supported.
- **0b0111** IMPLEMENTATION DEFINED features are supported.
- **0b1000** IMPLEMENTATION DEFINED features are supported.
- **0b1001** IMPLEMENTATION DEFINED features are supported.
- **0b1010** IMPLEMENTATION DEFINED features are supported.
- **0b1011** IMPLEMENTATION DEFINED features are supported.
- **0b1100** IMPLEMENTATION DEFINED features are supported.
- **0b1101** IMPLEMENTATION DEFINED features are supported.
- **0b1110** IMPLEMENTATION DEFINED features are supported.
- **0b1111** IMPLEMENTATION DEFINED features are supported.

Use of nonzero values requires written permission from Arm.

Access to this field is RO.

**Accessing the TRCIMSPEC0:**

TRCIMSPEC0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x1C0</td>
<td>TRCIMSPEC0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.44 TRCIMSPEC<n>, IMP DEF Register <n>, n = 1 - 7

The TRCIMSPEC<n> characteristics are:

**Purpose**
These registers might return information that is specific to an implementation, or enable features specific to an implementation to be programmed. The product Technical Reference Manual describes these registers.

**Configurations**
External register TRCIMSPEC<n> bits [31:0] are architecturally mapped to AArch64 System register TRCIMSPEC<n>[31:0].
This register is present only when the trace unit implements this OPTIONAL register and FEAT_ETE is implemented. Otherwise, direct accesses to TRCIMSPEC<n> are RES0.

**Attributes**
TRCIMSPEC<n> is a 32-bit register.

**Field descriptions**

**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.
This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

**Accessing the TRCIMSPEC<n>:**
TRCIMSPEC<n> can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x1C0 + (4 * n)</td>
<td>TRCIMSPEC&lt;n&gt;</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.45 TRCITCTRL, Integration Mode Control Register

The TRCITCTRL characteristics are:

**Purpose**

A component can use TRCITCTRL to dynamically switch between functional mode and integration mode. In integration mode, topology detection is enabled. After switching to integration mode and performing integration tests or topology detection, reset the system to ensure correct behavior of CoreSight and other connected system components.

For additional information, see the CoreSight Architecture Specification.

**Configurations**

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCITCTRL are RES0.

**Attributes**

TRCITCTRL is a 32-bit register.

**Field descriptions**

![Field diagram](image)

Bits [31:1]

Reserved, RES0.

IME, bit [0]

When topology detection or integration functionality is implemented:

Integration Mode Enable.

- 0b0 Component functional mode.
- 0b1 Component integration mode. Support for topology detection and integration testing is enabled.

Otherwise:

Reserved, RES0.

**Accessing the TRCITCTRL:**

External debugger accesses to this register are IMPLEMENTATION DEFINED when the trace unit is not in the Idle state.

TRCITCTRL can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xF00</td>
<td>TRCITCTRL</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.46 TRCLAR, Lock Access Register

The TRCLAR characteristics are:

Purpose

Used to lock and unlock the Software Lock.
Note that ETE does not implement the Software Lock.
For additional information, see the CoreSight Architecture Specification.

Configurations

This register is present only when FEAT_ETE is implemented and the Software Lock is implemented. Otherwise, direct accesses to TRCLAR are RES0.

Attributes

TRCLAR is a 32-bit register.

Field descriptions

KEY, bits [31:0]

When Software Lock is implemented:
Software Lock Key.
A value of 0xC5ACCE55 unlocks the Software Lock.
Any other value locks the Software Lock.

Otherwise:
Reserved, RES0.

Accessing the TRCLAR:

External debugger accesses to this register are unaffected by the OS Lock.

TRCLAR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFB0</td>
<td>TRCLAR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are WO.
H9.3.47 TRCLSR, Lock Status Register

The TRCLSR characteristics are:

**Purpose**

Indicates whether the Software Lock is implemented, and the current status of the Software Lock.

For additional information, see the CoreSight Architecture Specification.

**Configurations**

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCLSR are RES0.

**Attributes**

TRCLSR is a 32-bit register.

**Field descriptions**

![Field descriptions diagram]

**Bits [31:3]**

Reserved, RES0.

**nTT, bit [2]**

Software lock size.

Reads as 0b0.

Access to this field is RO.

**SLK, bit [1]**

The current Software Lock status.

0b0  Software Lock is unlocked.

0b1  Software Lock is locked. Writes to the other registers in this component, except for the TRCLAR, are ignored.

This field reads as 0.

**SLI, bit [0]**

Indicates whether the Software Lock is implemented.

0b0  Software Lock is not implemented. Writes to the TRCLAR are ignored.

0b1  Software Lock is implemented.

This field reads as 0.

**Accessing the TRCLSR:**

External debugger accesses to this register are unaffected by the OS Lock.

TRCLSR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFB4</td>
<td>TRCLSR</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.48 TRCOSLSR, Trace OS Lock Status Register

The TRCOSLSR characteristics are:

**Purpose**

Returns the status of the Trace OS Lock.

**Configurations**

External register TRCOSLSR bits [31:0] are architecturally mapped to AArch64 System register TRCOSLSR[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCOSLSR are RES0.

**Attributes**

TRCOSLSR is a 32-bit register.

**Field descriptions**

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
   RES0
```

**Bits [31:5]**

Reserved, RES0.

**OSLM, bits [4:3, 0]**

OS Lock model.

- **0b00**  Trace OS Lock is not implemented.
- **0b01**  Trace OS Lock is implemented.
- **0b10**  Trace OS Lock is not implemented, and the trace unit is controlled by the PE OS Lock.

All other values are reserved.

This field reads as 0b100.

The OSLM field is split as follows:

- OSLM[2:1] is TRCOSLSR[4:3].
- OSLM[0] is TRCOSLSR[0].

**Bit [2]**

Reserved, RES0.

**OSLK, bit [1]**

OS Lock status.

- **0b**  The OS Lock is unlocked.
- **0b1**  The OS Lock is locked.

Note that this field indicates the state of the PE OS Lock.

**Accessing the TRCOSLSR:**

External debugger accesses to this register are unaffected by the OS Lock.
TRCOSLSR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x304</td>
<td>TRCOSLSR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.49 TRCPDCR, PowerDown Control Register

The TRCPDCR characteristics are:

**Purpose**
Requests the system to provide power to the trace unit.

**Configurations**
This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCPDCR are RES0.

**Attributes**
TRCPDCR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits [31:4]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>0b0</td>
<td>The system can remove power from the trace unit core power domain, or requests for power to the trace unit core power domain are implemented outside of the trace unit.</td>
</tr>
<tr>
<td>0b1</td>
<td>The system must provide power to the trace unit core power domain.</td>
</tr>
<tr>
<td>Bits [2:0]</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**Accessing the TRCPDCR:**

External debugger accesses to this register are unaffected by the OS Lock.

TRCPDCR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x310</td>
<td>TRCPDCR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When `!IsTraceCorePowered()` accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
**H9.3.50  TRCPDSR, PowerDown Status Register**

The TRCPDSR characteristics are:

**Purpose**

Indicates the power status of the trace unit.

**Configurations**

This register is present only when FEAT_ESE is implemented. Otherwise, direct accesses to TRCPDSR are RES0.

**Attributes**

TRCPDSR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:6</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td></td>
<td>0b0 The OS Lock is unlocked.</td>
</tr>
<tr>
<td></td>
<td>0b1 The OS Lock is locked.</td>
</tr>
<tr>
<td></td>
<td>Note that this field indicates the state of the PE OS Lock.</td>
</tr>
<tr>
<td>4:2</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>1</td>
<td>STICKYPD, bit [1] Sticky powerdown status. Indicates whether the trace register state is valid.</td>
</tr>
<tr>
<td></td>
<td>0b0 The state of TRCOSLSR and the trace registers are valid.</td>
</tr>
<tr>
<td></td>
<td>0b1 The state of TRCOSLSR and the trace registers might not be valid.</td>
</tr>
<tr>
<td></td>
<td>This field is set to 1 if the power to the trace unit core power domain is removed and the trace unit register state is not valid.</td>
</tr>
<tr>
<td></td>
<td>The STICKYPD field is read-sensitive. On a read of the TRCPDSR, this field is cleared to 0 after the register has been read.</td>
</tr>
<tr>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On a Trace unit reset, this field resets to 1.</td>
</tr>
<tr>
<td>0</td>
<td>POWER, bit [0] Power Status.</td>
</tr>
<tr>
<td></td>
<td>0b0 The trace unit core power domain is not powered. All trace unit registers are not accessible and they all return an error response.</td>
</tr>
<tr>
<td></td>
<td>0b1 The trace unit core power domain is powered. Trace unit registers are accessible.</td>
</tr>
<tr>
<td></td>
<td>Access to this field is RAO/WI.</td>
</tr>
</tbody>
</table>
Accessing the TRCPDSR:

External debugger accesses to this register are unaffected by the OS Lock.

TRCPDSR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x314</td>
<td>TRCPDSR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.51 TRCPIDR0, Peripheral Identification Register 0

The TRCPIDR0 characteristics are:

**Purpose**

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

**Configurations**

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCPIDR0 are RES0.

**Attributes**

TRCPIDR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:8]</td>
<td>RES0</td>
</tr>
<tr>
<td>[7:0]</td>
<td>PART_0</td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**PART_0, bits [7:0]**

Part number, bits [7:0].

The part number is selected by the designer of the component, and is stored in TRCPIDR1.PART_1 and TRCPIDR0.PART_0.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Accessing the TRCPIDR0:**

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFE0</td>
<td>TRCPIDR0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
### H9.3.52 TRCPIDR1, Peripheral Identification Register 1

The TRCPIDR1 characteristics are:

**Purpose**

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

**Configurations**

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCPIDR1 are RES0.

**Attributes**

TRCPIDR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:8</td>
<td>RES0</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>
| 7:4  | DES_0  | Designer, JEP106 identification code, bits [3:0]. TRCPIDR1.DES_0 and TRCPIDR2.DES_1 together form the JEDEC-assigned JEP106 identification code for the designer of the component. The parity bit in the JEP106 identification code is not included. The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.  
**Note**  
For a component designed by Arm Limited, the JEP106 identification code is 0x3B.  
**Note**  
This field has an IMPLEMENTATION DEFINED value.  
Access to this field is RO. |
| 3:0  | PART_1 | Part number, bits [11:8]. The part number is selected by the designer of the component, and is stored in TRCPIDR1.PART_1 and TRCPIDR0.PART_0.  
**Note**  
This field has an IMPLEMENTATION DEFINED value.  
Access to this field is RO. |

**Accessing the TRCPIDR1:**

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFE4</td>
<td>TRCPIDR1</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.53   TRCPIDR2, Peripheral Identification Register 2

The TRCPIDR2 characteristics are:

Purpose

Provides discovery information about the component.
For additional information, see the CoreSight Architecture Specification.

Configurations

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCPIDR2 are RES0.

Attributes

TRCPIDR2 is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>REVISION</td>
<td>DES_1</td>
<td>JEDEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Component major revision. TRCPIDR2.REVISION and TRCPIDR3.REVAND together form the revision number of the component, with TRCPIDR2.REVISION being the most significant part and TRCPIDR3.REVAND the least significant part. When a component is changed, TRCPIDR2.REVISION or TRCPIDR3.REVAND are increased to ensure that software can differentiate the different revisions of the component. TRCPIDR3.REVAND should be set to 0b0000 when TRCPIDR2.REVISION is increased.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used.
Reads as 0b1.
Access to this field is RO.

DES_1, bits [2:0]

Designer, JEP106 identification code, bits [6:4]. TRCPIDR1.DES_0 and TRCPIDR2.DES_1 together form the JEDEC-assigned JEP106 identification code for the designer of the component. The parity bit in the JEP106 identification code is not included. The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

Note

For a component designed by Arm Limited, the JEP106 identification code is 0x3B.

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.
Accessing the TRCPIDR2:

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR2 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFE8</td>
<td>TRCPIDR2</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.54   TRCPIDR3, Peripheral Identification Register 3

The TRCPIDR3 characteristics are:

**Purpose**
Provides discovery information about the component.
For additional information, see the CoreSight Architecture Specification.

**Configurations**
This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCPIDR3 are RES0.

**Attributes**
TRCPIDR3 is a 32-bit register.

**Field descriptions**

```
+----------------+----------------+----------------+----------------+
|     31         |       8        |       3        |       0        |
| RES0           | REV AND       | CMOD           |
+----------------+----------------+----------------+
```

**Bits [31:8]**
Reserved, RES0.

**REVAND, bits [7:4]**
Component minor revision. TRCPIDR2.REVISION and TRCPIDR3.REV AND together form the revision number of the component, with TRCPIDR2.REVISION being the most significant part and TRCPIDR3.REV AND the least significant part. When a component is changed, TRCPIDR2.REVISION or TRCPIDR3.REV AND are increased to ensure that software can differentiate the different revisions of the component. TRCPIDR3.REV AND should be set to 0b0000 when TRCPIDR2.REVISION is increased.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

**CMOD, bits [3:0]**
Customer Modified.
Indicates the component has been modified.
A value of 0b0000 means the component is not modified from the original design.
Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.
For any two components with the same Unique Component Identifier:
- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same non-zero value, it does not necessarily mean that they have the same modifications.
- If the value of the CMOD field of either of the two components is non-zero, they might not be identical, even though they have the same Unique Component Identifier.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

**Accessing the TRCPIDR3:**
External debugger accesses to this register are unaffected by the OS Lock.
TRCPIDR3 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFEC</td>
<td>TRCPIDR3</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.55 TRCPIDR4, Peripheral Identification Register 4

The TRCPIDR4 characteristics are:

**Purpose**

Provides discovery information about the component. For additional information, see the CoreSight Architecture Specification.

**Configurations**

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCPIDR4 are RES0.

**Attributes**

TRCPIDR4 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:8]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIZE, bits [7:4]</td>
<td></td>
</tr>
<tr>
<td>Size of the component. The distance from the start of the address space used by this component to the end of the component identification registers. A value of 0b0000 means one of the following is true:</td>
<td></td>
</tr>
<tr>
<td>• The component uses a single 4KB block. • The component uses an IMPLEMENTATION DEFINED number of 4KB blocks.</td>
<td></td>
</tr>
<tr>
<td>Any other value means the component occupies 2^{TRCPIDR4.SIZE} 4KB blocks. Using this field to indicate the size of the component is deprecated. This field might not correctly indicate the size of the component. Arm recommends that software determine the size of the component from the Unique Component Identifier fields, and other IMPLEMENTATION DEFINED registers in the component. Reads as 0b0000. Access to this field is RO.</td>
<td></td>
</tr>
<tr>
<td>DES_2, bits [3:0]</td>
<td></td>
</tr>
<tr>
<td>Designer, JEP106 continuation code. This is the JEDEC-assigned JEP106 bank identifier for the designer of the component, minus 1. The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC <a href="http://www.jedec.org">http://www.jedec.org</a>.</td>
<td></td>
</tr>
</tbody>
</table>

**Note**

For a component designed by Arm Limited, the JEP106 bank is 5, meaning this field has the value 0x4.

This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.
Accessing the TRCPIDR4:

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR4 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFD0</td>
<td>TRCPIDR4</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.56 TRCPIDR5, Peripheral Identification Register 5

The TRCPIDR5 characteristics are:

**Purpose**

Provides discovery information about the component.
For additional information, see the CoreSight Architecture Specification.

**Configurations**

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCPIDR5 are RES0.

**Attributes**

TRCPIDR5 is a 32-bit register.

**Field descriptions**

```
+-------+-------+
| 31    | 0     |
| RES0  | RES0  |
```

**Bits [31:0]**

Reserved, RES0.

**Accessing the TRCPIDR5:**

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR5 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFD4</td>
<td>TRCPIDR5</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RO.
H9.3.57   TRCPIDR6, Peripheral Identification Register 6

The TRCPIDR6 characteristics are:

**Purpose**

Provides discovery information about the component.
For additional information, see the CoreSight Architecture Specification.

**Configurations**

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCPIDR6 are RES0.

**Attributes**

TRCPIDR6 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>

**Accessing the TRCPIDR6:**

External debugger accesses to this register are unaffected by the OS Lock.
TRCPIDR6 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFD8</td>
<td>TRCPIDR6</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

* When !IsTraceCorePowered() accesses to this register generate an error response.
* Otherwise accesses to this register are RO.
H9.3.58 TRCPIDR7, Peripheral Identification Register 7

The TRCPIDR7 characteristics are:

**Purpose**

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

**Configurations**

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCPIDR7 are RES0.

**Attributes**

TRCPIDR7 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>RES0</td>
</tr>
</tbody>
</table>

**Bits [31:0]**

Reserved, RES0.

**Accessing the TRCPIDR7:**

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR7 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0xFD</td>
<td>TRCPIDR7</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When !IsTraceCorePowered() accesses to this register generate an error response.

- Otherwise accesses to this register are RO.
H9.3.59 TRCPRGCTLR, Programming Control Register

The TRCPRGCTLR characteristics are:

**Purpose**

Enables the trace unit.

**Configurations**

External register TRCPRGCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCPRGCTLR[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCPRGCTLR are RES0.

**Attributes**

TRCPRGCTLR is a 32-bit register.

**Field descriptions**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>EN</td>
</tr>
</tbody>
</table>

**Bits [31:1]**

Reserved, RES0.

**EN, bit [0]**

Trace unit enable.

0b0 The trace unit is disabled.

0b1 The trace unit is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to 0.

**Accessing the TRCPRGCTLR:**

Must be programmed.

TRCPRGCTLR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x004</td>
<td>TRCPRGCTLR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.

- Otherwise accesses to this register are RW.
H9.3.60 TRCQCTLR, Q Element Control Register

The TRCQCTLR characteristics are:

**Purpose**

Controls when Q elements are enabled.

**Configurations**

External register TRCQCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCQCTLR[31:0].

This register is present only when FEAT_ETE is implemented and TRCIDR0.QFILT == 1. Otherwise, direct accesses to TRCQCTLR are RES0.

**Attributes**

TRCQCTLR is a 32-bit register.

**Field descriptions**

![Field Descriptions Diagram]

**Bits [31:9]**

Reserved, RES0.

**MODE, bit [8]**

Selects whether the Address Range Comparators selected by TRCQCTLR.RANGE indicate address ranges where the trace unit is permitted to generate Q elements or address ranges where the trace unit is not permitted to generate Q elements:

- **0b0** Exclude mode.
  
  The Address Range Comparators selected by TRCQCTLR.RANGE indicate address ranges where the trace unit must not generate Q elements. If no ranges are selected, Q elements are permitted across the entire memory map.

- **0b1** Include Mode.
  
  The Address Range Comparators selected by TRCQCTLR.RANGE indicate address ranges where the trace unit can generate Q elements. If all the implemented bits in RANGE are set to 0 then Q elements are disabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**RANGE[<m>], bit [m], for m = 7 to 0**

Specifies whether Address Range Comparator <m> controls Q elements.

- **0b0** The address range that Address Range Comparator <m> defines, is not selected.
- **0b1** The address range that Address Range Comparator <m> defines, is selected.

This bit is RES0 if m >= TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
**Accessing the TRCQCTRL:**

Must be programmed if `TRCCONFIGR.QE != 0b00`.

Writes are **CONSTRAINED UNPREDICTABLE** if the trace unit is not in the Idle state.

TRCQCTRL can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x044</td>
<td>TRCQCTRL</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When `OSLockStatus()`, or `!AllowExternalTraceAccess()` or `!IsTraceCorePowered()` accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.61 TRCRSCTLR<n>, Resource Selection Control Register <n>, n = 2 - 31

The TRCRSCTLR<n> characteristics are:

**Purpose**

Controls the selection of the resources in the trace unit.

**Configurations**

External register TRCRSCTLR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCRSCTLR<n>[31:0].

This register is present only when FEAT_ETE is implemented and (UInt(TRCIDR4.NUMRSPAIR) + 1) * 2 > n. Otherwise, direct accesses to TRCRSCTLR<n> are RES0.

Resource selector 0 always returns FALSE.

Resource selector 1 always returns TRUE.

Resource selectors are implemented in pairs. Each odd numbered resource selector is part of a pair with the even numbered resource selector that is numbered as one less than it. For example, resource selectors 2 and 3 form a pair.

**Attributes**

TRCRSCTLR<n> is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>21</th>
<th>19</th>
<th>18</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>GROUP</td>
<td>SELECT</td>
<td>PAIRINV</td>
<td>INV</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:22]**

Reserved, RES0.

**PAIRINV, bit [21]**

_When n MOD 2 == 0:_

Controls whether the combined result from a resource selector pair is inverted.

- **0b0** Do not invert the combined output of the 2 resource selectors.
- **0b1** Invert the combined output of the 2 resource selectors.

If:

- A is the register TRCRSCTLR<n>.
- B is the register TRCRSCTLR<n+1>.

Then the combined output of the 2 resource selectors A and B depends on the value of (A.PAIRINV, A.INV, B.INV) as follows:

- **0b000** -> A and B.
- **0b001** -> Reserved.
- **0b010** -> not(A) and B.
- **0b011** -> not(A) and not(B).
- **0b100** -> not(A) or not(B).
- **0b101** -> not(A) or B.
- **0b110** -> Reserved.
- **0b111** -> A or B.
The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**

Reserved, RES0.

**INV, bit [20]**

Controls whether the resource, that TRCRSCTLR<n>.GROUP and TRCRSCTLR<n>.SELECT selects, is inverted.

- **0b0**  Do not invert the output of this selector.
- **0b1**  Invert the output of this selector.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally **UNKNOWN** value.

**GROUP, bits [19:16]**

Selects a group of resources.

- **0b0000**  External Input Selectors.
- **0b0001**  PE Comparator Inputs.
- **0b0010**  Counters and Sequencer.
- **0b0011**  Single-shot Comparator Controls.
- **0b0100**  Single Address Comparators.
- **0b0101**  Address Range Comparators.
- **0b0110**  Context Identifier Comparators.
- **0b0111**  Virtual Context Identifier Comparators.

All other values are reserved.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally **UNKNOWN** value.

**SELECT, bits [15:0]**

Resource Specific Controls. Contains the controls specific to the resource group selected by GROUP, described in the following sections.

**SELECT encoding for External Input Selectors**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>4-0</td>
<td>EXTIN[3:0]</td>
</tr>
</tbody>
</table>

**Bits [15:4]**

Reserved, RES0.

**EXTIN[<m>], bit [m], for m = 3 to 0**

Selects one or more External Inputs.

- **0b0**  Ignore EXTIN <m>.
- **0b1**  Select EXTIN <m>.

This bit is RES0 if m >= TRCIDR5.NUMEXTINSEL.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally **UNKNOWN** value.
**SELECT encoding for PE Comparator Inputs**

Bits [15:8]

Reserved, RES0.

PECOMP[<m>], bit [m], for m = 7 to 0
Selects one or more PE Comparator Inputs.

- **0b0**: Ignore PE Comparator Input <m>.
- **0b1**: Select PE Comparator Input <m>.

This bit is RES0 if m >= TRCIDR4.NUMPC.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**SELECT encoding for Counters and Sequencer**

Bits [15:8]

Reserved, RES0.

SEQUENCER[<m>], bit [m+4], for m = 3 to 0
Sequencer states.

- **0b0**: Ignore Sequencer state <m>.
- **0b1**: Select Sequencer state <m>.

This bit is RES0 if m >= TRCIDR5.NUMSEQSTATE.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

COUNTERS[<m>], bit [m], for m = 3 to 0
Counters resources at zero.

- **0b0**: Ignore Counter <m>.
- **0b1**: Select Counter <m> is zero.

This bit is RES0 if m >= TRCIDR5.NUMCNTR.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
**SELECT encoding for Single-shot Comparator Controls**

Bits [15:8]

Reserved, RES0.

SINGLE_SHOT[m], bit [m], for m = 7 to 0

Selects one or more Single-shot Comparator Controls.

0b0 Ignore Single-shot Comparator Control <m>.
0b1 Select Single-shot Comparator Control <m>.

This bit is RES0 if m >= TRCIDR4.NUMSSCC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**SELECT encoding for Single Address Comparators**

SAC[m], bit [m], for m = 15 to 0

Selects one or more Single Address Comparators.

0b0 Ignore Single Address Comparator <m>.
0b1 Select Single Address Comparator <m>.

This bit is RES0 if m >= 2 × TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
**SELECT encoding for Address Range Comparators**

Bits [15:8]
Reserved, RES0.

ARC[<m>], bit [m], for m = 7 to 0
Selects one or more Address Range Comparators.
- 0b0: Ignore Address Range Comparator <m>.
- 0b1: Select Address Range Comparator <m>.

This bit is RES0 if m >= TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**SELECT encoding for Context Identifier Comparators**

Bits [15:8]
Reserved, RES0.

CID[<m>], bit [m], for m = 7 to 0
Selects one or more Context Identifier Comparators.
- 0b0: Ignore Context Identifier Comparator <m>.
- 0b1: Select Context Identifier Comparator <m>.

This bit is RES0 if m >= TRCIDR4.NUMCIDC.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**SELECT encoding for Virtual Context Identifier Comparators**

No diagram provided for this section.
Bits [15:8]

Reserved, RES0.

VMID[m], bit [m], for m = 7 to 0

Selects one or more Virtual Context Identifier Comparators.

0b0  Ignore Virtual Context Identifier Comparator <m>.

0b1  Select Virtual Context Identifier Comparator <m>.

This bit is RES0 if m >= TRCIDR4.NUMVMIDC.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing the TRCRSCtlR<n>:

Must be programmed if any of the following are true:

• TRCCNCTL<n>.RLDEVENT.TYPE == 0 and TRCCNCTL<n>.RLDEVENT.SEL == n.
• TRCCNCTL<n>.RLDEVENT.TYPE == 1 and TRCCNCTL<n>.RLDEVENT.SEL == n/2.
• TRCCNCTL<n>.CNTEVENT.TYPE == 0 and TRCCNCTL<n>.CNTEVENT.SEL == n.
• TRCCNCTL<n>.CNTEVENT.TYPE == 1 and TRCCNCTL<n>.CNTEVENT.SEL == n/2.
• TRCEVENTCTL0R.EVENT0.TYPE == 0 and TRCEVENTCTL0R.EVENT0.SEL == n.
• TRCEVENTCTL0R.EVENT0.TYPE == 1 and TRCEVENTCTL0R.EVENT0.SEL == n/2.
• TRCEVENTCTL0R.EVENT1.TYPE == 0 and TRCEVENTCTL0R.EVENT1.SEL == n.
• TRCEVENTCTL0R.EVENT1.TYPE == 1 and TRCEVENTCTL0R.EVENT1.SEL == n/2.
• TRCEVENTCTL0R.EVENT2.TYPE == 0 and TRCEVENTCTL0R.EVENT2.SEL == n.
• TRCEVENTCTL0R.EVENT2.TYPE == 1 and TRCEVENTCTL0R.EVENT2.SEL == n/2.
• TRCEVENTCTL0R.EVENT3.TYPE == 0 and TRCEVENTCTL0R.EVENT3.SEL == n.
• TRCEVENTCTL0R.EVENT3.TYPE == 1 and TRCEVENTCTL0R.EVENT3.SEL == n/2.
• TRCSEQEVR<n>.B.TYPE == 0 and TRCSEQEVR<n>.B.SEL = n.
• TRCSEQEVR<n>.B.TYPE == 1 and TRCSEQEVR<n>.B.SEL = n/2.
• TRCSEQEVR<n>.F.TYPE == 0 and TRCSEQEVR<n>.F.SEL = n.
• TRCSEQEVR<n>.F.TYPE == 1 and TRCSEQEVR<n>.F.SEL = n/2.
• TRCSQRTSR<n>.RST.TYPE == 0 and TRCSQRTSR<n>.RST.SEL == n.
• TRCSQRTSR<n>.RST.TYPE == 1 and TRCSQRTSR<n>.RST.SEL == n/2.
• TRCTSTCTRL.EVENT.TYPE == 0 and TRCTSTCTRL.EVENT.SEL == n.
• TRCTSTCTRL.EVENT.TYPE == 1 and TRCTSTCTRL.EVENT.SEL == n/2.
• TRCVICTLR.EVENT.TYPE == 0 and TRCVICTLR.EVENT.SEL == n.
• TRCVICTLR.EVENT.TYPE == 1 and TRCVICTLR.EVENT.SEL == n/2.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.
TRCRSCTLR<\text{n}> can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x200 + (4 * n)</td>
<td>TRCRSCTLR&lt;n&gt;</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.62 TRCRSR, Resources Status Register

The TRCRSR characteristics are:

Purpose

Use this to set, or read, the status of the resources.

Configurations

External register TRCRSR bits [31:0] are architecturally mapped to AArch64 System register TRCRSR[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCRSR are RES0.

Attributes

TRCRSR is a 32-bit register.

Field descriptions

Bits [31:13]

Reserved, RES0.

TA, bit [12]

Tracing active.

0b0 Tracing is not active.

0b1 Tracing is active.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

EVENT[<m>], bit [m+8], for m = 3 to 0

Untraced status of ETEEvents.

0b0 An ETEEvent <m> has not occurred.

0b1 An ETEEvent <m> has occurred while the resources were in the Paused state.

This bit is RES0 if TRCIDR4.NUMRSPAIR == 0 || m > TRCIDR0.NUMEVENT.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [7:4]

Reserved, RES0.

EXTIN[<m>], bit [m], for m = 3 to 0

The sticky status of the External Input Selectors.

0b0 An event selected by External Input Selector <m> has not occurred.

0b1 At least one event selected by External Input Selector <m> has occurred while the resources were in the Paused state.

This bit is RES0 if m >= TRCIDR5.NUMEXTINSEL.
The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing the TRCRSR:**

Must always be programmed.

Writes are **CONSTRAINED UNPREDICTABLE** if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCRSR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x028</td>
<td>TRCRSR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.63 TRCSEQEVR<n>, Sequencer State Transition Control Register <n>, n = 0 - 2

The TRCSEQEVR<n> characteristics are:

**Purpose**

Moves the Sequencer state:
- Backwards, from state n+1 to state n when a programmed resource event occurs.
- Forwards, from state n to state n+1 when a programmed resource event occurs.

**Configurations**

External register TRCSEQEVR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCSEQEVR<n>[31:0].

This register is present only when FEAT_ETE is implemented and TRCIDR5.NUMSEQSTATE != 0b000. Otherwise, direct accesses to TRCSEQEVR<n> are RES0.

**Attributes**

TRCSEQEVR<n> is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-16</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>15</td>
<td>B_TYPE, bit [15]</td>
</tr>
<tr>
<td>14-13</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>12-8</td>
<td>B_SEL, bits [12:8]</td>
</tr>
<tr>
<td>7-0</td>
<td>F_SEL, bits [7:0]</td>
</tr>
</tbody>
</table>

**Bits [31:16]**

Reserved, RES0.

**B_TYPE, bit [15]**

Chooses the type of Resource Selector.

Backward field. Defines whether the backward resource event is a single Resource Selector or a Resource Selector pair. When the resource event occurs then the Sequencer state moves from state n+1 to state n. For example, if TRCSEQEVR2.B.SEL == 0x14 then when event 0x14 occurs, the Sequencer moves from state 3 to state 2.

- 0b0 A single Resource Selector.
  TRCSEQEVR<n>.B.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

- 0b1 A Boolean-combined pair of Resource Selectors.
  TRCSEQEVR<n>.B.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCSEQEVR<n>.B.SEL[4] is RES0.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Bits [14:13]**

Reserved, RES0.

**B_SEL, bits [12:8]**

Defines the selected Resource Selector or pair of Resource Selectors. TRCSEQEVR<n>.B.TYPE controls whether TRCSEQEVR<n>.B.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

Backward field. Selects the single Resource Selector or Resource Selector pair.
If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**F_TYPE, bit [7]**

Chooses the type of Resource Selector.

Backward field. Defines whether the forward resource event is a single Resource Selector or a Resource Selector pair. When the resource event occurs then the Sequencer state moves from state n to state n+1. For example, if TRCSEQEVR1.F_SEL == 0x12 then when event 0x12 occurs, the Sequencer moves from state 1 to state 2.

- 0b0 A single Resource Selector.
  - TRCSEQEVR<n>.F_SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

- 0b1 A Boolean-combined pair of Resource Selectors.
  - TRCSEQEVR<n>.F_SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event.
  - TRCSEQEVR<n>.F_SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Bits [6:5]**

Reserved, RES0.

**F_SEL, bits [4:0]**

Defines the selected Resource Selector or pair of Resource Selectors. TRCSEQEVR<n>.F_TYPE controls whether TRCSEQEVR<n>.F_SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

Forward field. Selects the single Resource Selector or Resource Selector pair.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing the TRCSEQEVR<n>**:  

Must be programmed if TRCRSCTRL<n>.GROUP == 0b0010 and TRCRSCTRL<n>.SEQUENCER != 0b0000.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCSEQEVR<n> can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x100 + (4 * n)</td>
<td>TRCSEQEVR&lt;n&gt;</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.64 TRCSEQRSTEV, Sequencer Reset Control Register

The TRCSEQRSTEV characteristics are:

Purpose

Moves the Sequencer to state 0 when a programmed resource event occurs.

Configurations

External register TRCSEQRSTEV bits [31:0] are architecturally mapped to AArch64 System register TRCSEQRSTEV[31:0].

This register is present only when FEAT_ETE is implemented and TRCIDR5.NUMSEQSTATE != 0b000. Otherwise, direct accesses to TRCSEQRSTEV are RES0.

Attributes

TRCSEQRSTEV is a 32-bit register.

Field descriptions

Bits [31:8]
Reserved, RES0.

RST_TYPE, bit [7]
Chooses the type of Resource Selector.

0b0 A single Resource Selector.

TRCSEQRSTEV.RST.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

0b1 A Boolean-combined pair of Resource Selectors.

TRCSEQRSTEV.RST.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCSEQRSTEV.RST.SEL[4] is RES0.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]
Reserved, RES0.

RST_SEL, bits [4:0]
Defines the selected Resource Selector or pair of Resource Selectors.

TRCSEQRSTEV.RST.TYPE controls whether TRCSEQRSTEV.RST_SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
**Accessing the TRCSEQRSTEVR:**

Must be programmed if \( \text{TRCRSCTRL}<n>.\text{GROUP} = 0b0010 \) and \( \text{TRCRSCTRL}<n>.\text{SEQUENCER} \neq 0b0000 \).

Writes are **CONSTRAINED UNPREDICTABLE** if the trace unit is not in the Idle state.

TRCSEQRSTEVR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x118</td>
<td>TRCSEQRSTEVR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.65 TRCSEQSTR, Sequencer State Register

The TRCSEQSTR characteristics are:

**Purpose**

Use this to set, or read, the Sequencer state.

**Configurations**

External register TRCSEQSTR bits [31:0] are architecturally mapped to AArch64 System register TRCSEQSTR[31:0].

This register is present only when FEAT_ETE is implemented and TRCIDR5.NUMSEQSTATE != 0b000. Otherwise, direct accesses to TRCSEQSTR are res0.

**Attributes**

TRCSEQSTR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:2]</th>
<th>Reserved, res0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE, bits [1:0]</td>
<td>Set or returns the state of the Sequencer.</td>
</tr>
<tr>
<td>0b00</td>
<td>State 0.</td>
</tr>
<tr>
<td>0b01</td>
<td>State 1.</td>
</tr>
<tr>
<td>0b10</td>
<td>State 2.</td>
</tr>
<tr>
<td>0b11</td>
<td>State 3.</td>
</tr>
</tbody>
</table>

The reset behavior of this field is:

* On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing the TRCSEQSTR:**

Must be programmed if TRCRSCTRL<n>.GROUP == 0b0010 and TRCRSCTRL<n>.SEQUENCER != 0b0000.

Writes are CONstrained UNPREDICatable if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCSEQSTR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x11C</td>
<td>TRCSEQSTR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

* When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
* Otherwise accesses to this register are RW.
H9.3.66 TRCSSCCR<n>, Single-shot Comparator Control Register <n>, n = 0 - 7

The TRCSSCCR<n> characteristics are:

**Purpose**

Controls the corresponding Single-shot Comparator Control resource.

**Configurations**

External register TRCSSCCR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCSSCCR<n>[31:0].

This register is present only when FEAT ETE is implemented and UInt(TRCIDR4.NUMSSCC) > n. Otherwise, direct accesses to TRCSSCCR<n> are RES0.

**Attributes**

TRCSSCCR<n> is a 32-bit register.

**Field descriptions**

Bits [31:25]

Reserved, RES0.

RST, bit [24]

Selects the Single-shot Comparator Control mode.

0b0 The Single-shot Comparator Control is in single-shot mode.

0b1 The Single-shot Comparator Control is in multi-shot mode.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

ARC[<m>], bit [m+16], for m = 7 to 0

Selects one or more Address Range Comparators for Single-shot control.

0b0 The Address Range Comparator <m>, is not selected for Single-shot control.

0b1 The Address Range Comparator <m>, is selected for Single-shot control.

This bit is RES0 if m >= TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
SAC[m], bit [m], for m = 15 to 0

Selects one or more Single Address Comparators for Single-shot control.

- 0b0 The Single Address Comparator <m>, is not selected for Single-shot control.
- 0b1 The Single Address Comparator <m>, is selected for Single-shot control.

This bit is RES0 if m >= 2 × TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing the TRCSSCCR<n>:**

Must be programmed if any TRCRSCTL<n>.GROUP == 0b011 and TRCRSCTL<n>.SINGLE_SHOT[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCSSCCR<n> can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x280 + (4 * n)</td>
<td>TRCSSCCR&lt;n&gt;</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.67 TRCSSCSR<\(n\)>, Single-shot Comparator Control Status Register \(<n\), \(n = 0 \text{ - 7}\),

The TRCSSCSR<\(n\)> characteristics are:

**Purpose**

Returns the status of the corresponding Single-shot Comparator Control.

**Configurations**

External register TRCSSCSR<\(n\)> bits [31:0] are architecturally mapped to AArch64 System register TRCSSCSR<\(n\)>[31:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR4.NUMSSCC) > \(n\). Otherwise, direct accesses to TRCSSCSR<\(n\)> are RES0.

**Attributes**

TRCSSCSR<\(n\)> is a 32-bit register.

**Field descriptions**

![Field Description Diagram]

**STATUS, bit [31]**

Single-shot Comparator Control status. Indicates if any of the comparators selected by this Single-shot Comparator control have matched. The selected comparators are defined by TRCSSCCR<\(n\)>-ARC, TRCSSCCR<\(n\)>-SAC, and TRCSSPCICR<\(n\)>-PC.

- \(0b0\) No match has occurred. When the first match occurs, this field takes a value of 1. It remains at 1 until explicitly modified by a write to this register.
- \(0b1\) One or more matches has occurred. If TRCSSCCR<\(n\)>-RST == 0 then:
  - There is only one match and no more matches are possible.
  - Software must reset this field to 0 to re-enable the Single-shot Comparator Control.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**PENDING, bit [30]**

Single-shot pending status. The Single-shot Comparator Control fired while the resources were in the Paused state.

- \(0b0\) No match has occurred.
- \(0b1\) One or more matches has occurred.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Bits [29:4]**

Reserved, RES0.
PC, bit [3]

PE Comparator Input support. Indicates if the Single-shot Comparator Control supports PE Comparator Inputs.

0b0  This Single-shot Comparator Control does not support PE Comparator Inputs. Selecting any PE Comparator Inputs using the associated TRCSSPCICR<n> results in CONSTRAINED UNPREDICTABLE behavior of the Single-shot Comparator Control resource. The Single-shot Comparator Control might match unexpectedly or might not match.

0b1  This Single-shot Comparator Control supports PE Comparator Inputs.

Access to this field is RO.

DV, bit [2]

Data value comparator support. Data value comparisons are not implemented in ETE and are reserved for other trace architectures. Allocated in other trace architectures.

0b0  This Single-shot Comparator Control does not support data value comparisons.

0b1  This Single-shot Comparator Control supports data value comparisons.

This field reads as 0.

Access to this field is RO.

DA, bit [1]

Data Address Comparator support. Data address comparisons are not implemented in ETE and are reserved for other trace architectures. Allocated in other trace architectures.

0b0  This Single-shot Comparator Control does not support data address comparisons.

0b1  This Single-shot Comparator Control supports data address comparisons.

This field reads as 0.

Access to this field is RO.

INST, bit [0]

Instruction Address Comparator support. Indicates if the Single-shot Comparator Control supports instruction address comparisons.

0b0  This Single-shot Comparator Control does not support instruction address comparisons.

0b1  This Single-shot Comparator Control supports instruction address comparisons.

This field reads as 1.

Access to this field is RO.

**Accessing the TRCSSCSR<n>:**

Must be programmed if TRCRSCTRL<n>, GROUP == 0b0011 and TRCRSCTRL<n>, SINGLE_SHOT[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCSSCSR<n> can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x2A0 + (4 * n)</td>
<td>TRCSSCSR&lt;n&gt;</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
• Otherwise accesses to this register are RW.
H9.3.68 TRCSSPCICR\(\langle n\rangle\), Single-shot Processing Element Comparator Input Control Register \(\langle n\rangle\),
\(n = 0 - 7\)

The TRCSSPCICR\(\langle n\rangle\) characteristics are:

**Purpose**

Returns the status of the corresponding Single-shot Comparator Control.

**Configurations**

External register TRCSSPCICR\(\langle n\rangle\) bits [31:0] are architecturally mapped to AArch64 System register TRCSSPCICR\(\langle n\rangle\)[31:0].

This register is present only when FEAT_ETE is implemented, UInt(TRCIDR4.NUMSSCC) > \(n\), UInt(TRCIDR4.NUMPC) > 0 and TRCSSCSR\(\langle n\rangle\).PC == 1. Otherwise, direct accesses to TRCSSPCICR\(\langle n\rangle\) are RES0.

**Attributes**

TRCSSPCICR\(\langle n\rangle\) is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>8</td>
<td>PC[7]</td>
</tr>
<tr>
<td>7</td>
<td>PC[6]</td>
</tr>
<tr>
<td>6</td>
<td>PC[5]</td>
</tr>
<tr>
<td>5</td>
<td>PC[4]</td>
</tr>
<tr>
<td>4</td>
<td>PC[3]</td>
</tr>
<tr>
<td>3</td>
<td>PC[2]</td>
</tr>
<tr>
<td>2</td>
<td>PC[1]</td>
</tr>
<tr>
<td>1</td>
<td>PC[0]</td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**PC[<m>], bit [m], for m = 7 to 0**

Selects one or more PE Comparator Inputs for Single-shot control.

- \(0\)\(\emptyset\) The single PE Comparator Input \(<m>\), is not selected as for Single-shot control.
- \(0\)\(\emptyset\) The single PE Comparator Input \(<m>\), is selected as for Single-shot control.

This bit is RES0 if \(m >=\) TRCIDR4.NUMPC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing the TRCSSPCICR\(\langle n\rangle\):**

Must be programmed if implemented and any TRCRSCTL\(\langle n\rangle\).GROUP == \(0\emptyset\emptyset\) and TRCRSCTL\(\langle n\rangle\).SINGLE_SSHOT\[n\] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCSSPCICR\(\langle n\rangle\) can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x2C0 + (4 (\times) (n))</td>
<td>TRCSSPCICR(\langle n\rangle)</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

• When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.

• Otherwise accesses to this register are RW.
H9.3.69  TRCSTALLCTRLR, Stall Control Register

The TRCSTALLCTRLR characteristics are:

Purpose

Enables trace unit functionality that prevents trace unit buffer overflows.

Configurations

External register TRCSTALLCTRLR bits [31:0] are architecturally mapped to AArch64 System register TRCSTALLCTRLR[31:0].

This register is present only when FEAT_ETE is implemented and TRCIDR3.STALLCTL == 1. Otherwise, direct accesses to TRCSTALLCTRLR are RES0.

Attributes

TRCSTALLCTRLR is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>RES0</td>
<td>RES0</td>
<td>LEVEL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits [31:14]

Reserved, RES0.

NOOVERFLOW, bit [13]

*When TRCIDR3.NOOVERFLOW == 1:

Trace overflow prevention.

0b0  Trace unit buffer overflow prevention is disabled.

0b1  Trace unit buffer overflow prevention is enabled.

Note that enabling this feature might cause a significant performance impact.

The reset behavior of this field is:

•  On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [12:9]

Reserved, RES0.

STALL, bit [8]

Instruction stall control. Controls if a trace unit can stall the PE when the trace buffer space is less than LEVEL.

0b0  The trace unit must not stall the PE.

0b1  The trace unit can stall the PE.

The reset behavior of this field is:

•  On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [7:4]

Reserved, RES0.
LEVEL, bits [3:0]
Threshold level field. The field can support 16 monotonic levels from 0b0000 to 0b1111.
The value 0b0000 defines the Minimal invasion level. This setting has a greater risk of a trace unit buffer overflow.
The value 0b1111 defines the Maximum invasion level. This setting has a reduced risk of a trace unit buffer overflow.
Note that for some implementations, invasion might occur at the minimal invasion level.
One or more of the least significant bits of LEVEL are permitted to be RES0. Arm recommends that LEVEL[3:2] are fully implemented. Arm strongly recommends that LEVEL[3] is always implemented. If one or more bits are RES0 and are written with a non-zero value, the effective value of LEVEL is rounded down to the nearest power of 2 value which has the RES0 bits as zero. For example, if LEVEL[1:0] are RES0 and a value of 0b1110 is written to LEVEL, the effective value of LEVEL is 0b1100.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing the TRCSTALLCTLR:
Must be programmed if implemented.
Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.
TRCSTALLCTLR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x02C</td>
<td>TRCSTALLCTLR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:
• When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
• Otherwise accesses to this register are RW.
H9.3.70  TRCSTATR, Trace Status Register

The TRCSTATR characteristics are:

Purpose

Returns the trace unit status.

Configurations

External register TRCSTATR bits [31:0] are architecturally mapped to AArch64 System register TRCSTATR[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCSTATR are RES0.

Attributes

TRCSTATR is a 32-bit register.

Field descriptions

Bits [31:2]

Reserved, RES0.

PMSTABLE, bit [1]

Programmers' model stable.

0b0 The programmers' model is not stable.

0b1 The programmers' model is stable.

Accessing this field has the following behavior:

• When the trace unit is enabled, access to this field is UNKNOWN/WI.

• Otherwise, access to this field is RO

IDLE, bit [0]

Idle status.

0b0 The trace unit is not idle.

0b1 The trace unit is idle.

Accessing the TRCSTATR:

TRCSTATR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x00C</td>
<td>TRCSTATR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

• When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.

• Otherwise accesses to this register are RO.
H9.3.71 TRCSYNCPR, Synchronization Period Register

The TRCSYNCPR characteristics are:

**Purpose**
Controls how often trace protocol synchronization requests occur.

**Configurations**
External register TRCSYNCPR bits [31:0] are architecturally mapped to AArch64 System register TRCSYNCPR[31:0].

This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCSYNCPR are RES0.

**Attributes**
TRCSYNCPR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-5</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>4-0</td>
<td>PERIOD</td>
</tr>
</tbody>
</table>

- **Bits [31:5]**
  - Reserved, RES0.

- **PERIOD, bits [4:0]**
  - Defines the number of bytes of trace between each periodic trace protocol synchronization request.
  - 0b00000 Trace protocol synchronization is disabled.
  - 0b01000 Trace protocol synchronization request occurs after 2^8 bytes of trace.
  - 0b01001 Trace protocol synchronization request occurs after 2^9 bytes of trace.
  - 0b01010 Trace protocol synchronization request occurs after 2^10 bytes of trace.
  - 0b01011 Trace protocol synchronization request occurs after 2^11 bytes of trace.
  - 0b01100 Trace protocol synchronization request occurs after 2^12 bytes of trace.
  - 0b01101 Trace protocol synchronization request occurs after 2^13 bytes of trace.
  - 0b01110 Trace protocol synchronization request occurs after 2^14 bytes of trace.
  - 0b01111 Trace protocol synchronization request occurs after 2^15 bytes of trace.
  - 0b10000 Trace protocol synchronization request occurs after 2^16 bytes of trace.
  - 0b10001 Trace protocol synchronization request occurs after 2^17 bytes of trace.
  - 0b10010 Trace protocol synchronization request occurs after 2^18 bytes of trace.
  - 0b10011 Trace protocol synchronization request occurs after 2^19 bytes of trace.
  - 0b10100 Trace protocol synchronization request occurs after 2^20 bytes of trace.
  - Other values are reserved. If a reserved value is programmed into PERIOD, then the behavior of the synchronization period counter is CONSTRAINED UNPREDICTABLE and one of the following behaviors occurs:
    - No trace protocol synchronization requests are generated by this counter.
    - Trace protocol synchronization requests occur at the specified period.
    - Trace protocol synchronization requests occur at some other UNKNOWN period which can vary.
  - The reset behavior of this field is:
    - On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
Accessing the TRCSYNCPR:

Must be programmed if TRCIDR3.SYNCPR == 0.

Writes are CONSTRANDED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCSYNCPR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x034</td>
<td>TRCSYNCPR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.72 TRCTRACEIDR, Trace ID Register

The TRCTRACEIDR characteristics are:

**Purpose**
Sets the trace ID for instruction trace.

**Configurations**
External register TRCTRACEIDR bits [31:0] are architecturally mapped to AArch64 System register TRCTRACEIDR[31:0].
This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCTRACEIDR are RES0.

**Attributes**
TRCTRACEIDR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:7</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>6:0</td>
<td>TRACEID</td>
</tr>
</tbody>
</table>

**TRCEID, bits [6:0]**
Trace ID field. Sets the trace ID value for instruction trace. The width of the field is indicated by the value of TRCIDR5.TRACEIDSIZE. Unimplemented bits are RES0.
If an implementation supports AMBA ATB, then:
- The width of the field is 7 bits.
- Writing a reserved trace ID value does not affect behavior of the trace unit but it might cause UNPREDICTABLE behavior of the trace capture infrastructure.

See the AMBA ATB Protocol Specification for information about which ATID values are reserved.
The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing the TRCTRACEIDR:**
Must be programmed if implemented.
Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.
TRCTRACEIDR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x040</td>
<td>TRCTRACEIDR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:
- When OSLockStatus(), or !IsTraceCorePowered() or !AllowExternalTraceAccess() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.73 TRCTSCTLR, Timestamp Control Register

The TRCTSCTLR characteristics are:

**Purpose**

Controls the insertion of global timestamps in the trace stream.

**Configurations**

External register TRCTSCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCTSCTLR[31:0].

This register is present only when FEAT_ETE is implemented and TRCIDR0.TSSIZE != 0b0000. Otherwise, direct accesses to TRCTSCTLR are RES0.

**Attributes**

TRCTSCTLR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>RES0</td>
<td>EVENT_SEL</td>
<td>EVENT_TYPE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**EVENT_TYPE, bit [7]**

*When TRCIDR4.NUMRSPAIR != 0b0000:

Chooses the type of Resource Selector.

0b0 A single Resource Selector. TRCTSCTLR.EVENT_SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

0b1 A Boolean-combined pair of Resource Selectors. TRCTSCTLR.EVENT_SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCTSCTLR.EVENT_SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bits [6:5]**

Reserved, RES0.

**EVENT_SEL, bits [4:0]**

*When TRCIDR4.NUMRSPAIR != 0b0000:

Defines the selected Resource Selector or pair of Resource Selectors. TRCTSCTLR.EVENT.TYPE controls whether TRCTSCTLR.EVENT_SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.
Selecting Resource Selector pair 0 using this field is **UNPREDICTABLE**, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**

Reserved, RES0.

**Accessing the TRCTSCTRLR:**

Must be programmed if `TRCCONFIGR.TS == 1`.

Writes are **CONSTRAINED UNPREDICTABLE** if the trace unit is not in the Idle state.

`TRCTSCTRLR` can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x030</td>
<td>TRCTSCTRLR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When `OSLockStatus()`, or `! AllowExternalTraceAccess()` or `! IsTraceCorePowered()` accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
**H9.3.74 TRCVICTLR, ViewInst Main Control Register**

The TRCVICTLR characteristics are:

**Purpose**

Controls instruction trace filtering.

**Configurations**

External register TRCVICTLR bits [31:0] are architecturally mapped to AArch64 System register TRCVICTLR[31:0]. This register is present only when FEAT_ETE is implemented. Otherwise, direct accesses to TRCVICTLR are RES0.

**Attributes**

TRCVICTLR is a 32-bit register.

**Field descriptions**

![Register Diagram]

**Bits [31:27]**

Reserved, RES0.

**EXLEVEL_RL_EL2, bit [26]**

When TRCIDR6.EXLEVEL_RL_EL2 == 1:

Filter instruction trace for EL2 in Realm state.

- **0b0** When TRCVICTLR.EXLEVEL_NS_EL2 is 0 the trace unit generates instruction trace for EL2 in Realm state.
- **0b1** When TRCVICTLR.EXLEVEL_NS_EL2 is 1 the trace unit does not generate instruction trace for EL2 in Realm state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**EXLEVEL_RL_EL1, bit [25]**

When TRCIDR6.EXLEVEL_RL_EL1 == 1:

Filter instruction trace for EL1 in Realm state.

- **0b0** When TRCVICTLR.EXLEVEL_NS_EL1 is 0 the trace unit generates instruction trace for EL1 in Realm state.
When TRCVICCTRL.EXLEVEL_NS_EL1 is 1 the trace unit does not generate instruction trace for EL1 in Realm state.

0b1 When TRCVICCTRL.EXLEVEL_NS_EL1 is 0 the trace unit does not generate instruction trace for EL1 in Realm state.
When TRCVICCTRL.EXLEVEL_NS_EL1 is 1 the trace unit generates instruction trace for EL1 in Realm state.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EXLEVEL_RL_EL0, bit [24]

When TRCIDR6.EXLEVEL_RL_EL0 == 1:
Filter instruction trace for EL0 in Realm state.

0b0 When TRCVICCTRL.EXLEVEL_NS_EL0 is 0 the trace unit generates instruction trace for EL0 in Realm state.
When TRCVICCTRL.EXLEVEL_NS_EL0 is 1 the trace unit does not generate instruction trace for EL0 in Realm state.

0b1 When TRCVICCTRL.EXLEVEL_NS_EL0 is 0 the trace unit does not generate instruction trace for EL0 in Realm state.
When TRCVICCTRL.EXLEVEL_NS_EL0 is 1 the trace unit generates instruction trace for EL0 in Realm state.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bit [23]
Reserved, RES0.

EXLEVEL_NS_EL2, bit [22]

When Non-secure EL2 is implemented:
Filter instruction trace for EL2 in Non-secure state.

0b0 The trace unit generates instruction trace for EL2 in Non-secure state.
0b1 The trace unit does not generate instruction trace for EL2 in Non-secure state.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

EXLEVEL_NS_EL1, bit [21]

When Non-secure EL1 is implemented:
Filter instruction trace for EL1 in Non-secure state.

0b0 The trace unit generates instruction trace for EL1 in Non-secure state.
0b1 The trace unit does not generate instruction trace for EL1 in Non-secure state.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.
EXLEVEL_NS_EL0, bit [20]

*When Non-secure EL0 is implemented:*

Filter instruction trace for EL0 in Non-secure state.

- **0b0** The trace unit generates instruction trace for EL0 in Non-secure state.
- **0b1** The trace unit does not generate instruction trace for EL0 in Non-secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

EXLEVEL_S_EL3, bit [19]

*When EL3 is implemented:*

Filter instruction trace for EL3.

- **0b0** The trace unit generates instruction trace for EL3.
- **0b1** The trace unit does not generate instruction trace for EL3.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

EXLEVEL_S_EL2, bit [18]

*When EL2 is implemented and FEAT_SEL2 is implemented:*

Filter instruction trace for EL2 in Secure state.

- **0b0** The trace unit generates instruction trace for EL2 in Secure state.
- **0b1** The trace unit does not generate instruction trace for EL2 in Secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

EXLEVEL_S_EL1, bit [17]

*When Secure EL1 is implemented:*

Filter instruction trace for EL1 in Secure state.

- **0b0** The trace unit generates instruction trace for EL1 in Secure state.
- **0b1** The trace unit does not generate instruction trace for EL1 in Secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

EXLEVEL_S_EL0, bit [16]

*When Secure EL0 is implemented:*

Filter instruction trace for EL0 in Secure state.

- **0b0** The trace unit generates instruction trace for EL0 in Secure state.
- **0b1** The trace unit does not generate instruction trace for EL0 in Secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
Otherwise:
Reserved, RES0.

Bits [15:12]
Reserved, RES0.

TRCERR, bit [11]

When TRCIDR3.TRCTR == 1:
Controls the forced tracing of System Error exceptions.

0b0 Forced tracing of System Error exceptions is disabled.
0b1 Forced tracing of System Error exceptions is enabled.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

TRCRESET, bit [10]

Controls the forced tracing of PE Resets.

0b0 Forced tracing of PE Resets is disabled.
0b1 Forced tracing of PE Resets is enabled.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SSSTATUS, bit [9]

ViewInst start/stop function status.

0b0 Stopped State.
The ViewInst start/stop function is in the stopped state.
0b1 Started State.
The ViewInst start/stop function is in the started state.

Before software enables the trace unit, it must write to this field to set the initial state of the ViewInst start/stop function. If the ViewInst start/stop function is not used then set this field to 1. Arm recommends that the value of this field is set before each trace session begins.

If the trace unit becomes disabled while a start point or stop point is still speculative, then the value of TRCVICTLR.SSSTATUS is UNKNOWN and might represent the result of a speculative start point or stop point.

If software which is running on the PE being traced disables the trace unit, either by clearing TRCPRGCTLR.EN or locking the OS Lock, Arm recommends that a DSB and an ISB instruction are executed before disabling the trace unit to prevent any start points or stop points being speculative at the point of disabling the trace unit. This procedure assumes that all start points or stop points occur before the barrier instructions are executed. The procedure does not guarantee that there are no speculative start points or stop points when disabling, although it helps minimize the probability.

The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:
• RES1 if all of the following are true:
  — TRCIDR4.NUMACPAIRS == 0b0000
  — TRCIDR4.NUMPC == 0b0000
• Otherwise, access to this field is RW
Bit [8]
Reserved, RES0.

**EVENT_TYPE, bit [7]**

*When TRCIDR4.NUMRSPAIR != 0b0000:*

Chooses the type of Resource Selector.

- **0b0** A single Resource Selector.
  
  TRCVICTLR.EVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.

- **0b1** A Boolean-combined pair of Resource Selectors.

  TRCVICTLR.EVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCVICTLR.EVENT.SEL[4] is RES0.

The reset behavior of this field is:

- **On a Trace unit reset, this field resets to an architecturally UNKNOWN value.**

*Otherwise:*

Reserved, RES0.

Bits [6:5]
Reserved, RES0.

**EVENT_SEL, bits [4:0]**

*When TRCIDR4.NUMRSPAIR != 0b0000:*

**EVENT_SEL**

Defines the selected Resource Selector or pair of Resource Selectors. TRCVICTLR.EVENT.TYPE controls whether TRCVICTLR.EVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- **On a Trace unit reset, this field resets to an architecturally UNKNOWN value.**

*When TRCIDR4.NUMRSPAIR == 0b0000:*

**Reserved**

This field is reserved:

- **Bits [4:1] are RES0.**
- **Bit [0] is RES1.**

*Otherwise:*

Reserved, RES0.

**Accessing the TRCVICTLR:**

Must be programmed.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.
TRCVICTLR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x80</td>
<td>TRCVICTLR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.75 TRCVIIECTLR, ViewInst Include/Exclude Control Register

The TRCVIIECTLR characteristics are:

Purpose

Use this to select, or read, the Address RangeComparators for the ViewInst include/exclude function.

Configurations

External register TRCVIIECTLR bits [31:0] are architecturally mapped to AArch64 System register TRCVIIECTLR[31:0].

This register is present only when FEAT ETE is implemented and UInt(TRCIDR4.NUMACPAIRS) > 0. Otherwise, direct accesses to TRCVIIECTLR are RES0.

Attributes

TRCVIIECTLR is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th></th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Bits [31:24]

Reserved, RES0.

EXCLUDE[<m>], bit [m+16], for m = 7 to 0

Exclude Address Range Comparator <m>. Selects whether Address Range Comparator <m> is in use with the ViewInst exclude function.

0b0 The address range that Address Range Comparator <m> defines, is not selected for the ViewInst exclude function.

0b1 The address range that Address Range Comparator <m> defines, is selected for the ViewInst exclude function.

This bit is RES0 if m >= TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [15:8]

Reserved, RES0.

INCLUDE[<m>], bit [m], for m = 7 to 0

Include Address Range Comparator <m>.

Selects whether Address Range Comparator <m> is in use with the ViewInst include function.

Selecting no comparators for the ViewInst include function indicates that all instructions are included by default.
The ViewInst exclude function then indicates which ranges are excluded.

0b0  The address range that Address Range Comparator <m> defines, is not selected for the ViewInst include function.

0b1  The address range that Address Range Comparator <m> defines, is selected for the ViewInst include function.

This bit is RES0 if m >= TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing the TRCVIIECTLR:**

Must be programmed if TRCIDR4.NUMACPAIRS > 0b0000.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCVIIECTLR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x084</td>
<td>TRCVIIECTLR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.

- Otherwise accesses to this register are RW.
H9.3.76 TRCVIPCSSCTL, ViewInst Start/Stop PE Comparator Control Register

The TRCVIPCSSCTL characteristics are:

**Purpose**
Use this to select, or read, which PE Comparator Inputs can control the ViewInst start/stop function.

**Configurations**
External register TRCVIPCSSCTL bits [31:0] are architecturally mapped to AArch64 System register TRCVIPCSSCTL[31:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR4.NUMPC) > 0. Otherwise, direct accesses to TRCVIPCSSCTL are RES0.

**Attributes**
TRCVIPCSSCTL is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Address</th>
<th>Field Description</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-24</td>
<td>Reserved, RES0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>STOP[7]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>STOP[6]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>STOP[5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>STOP[4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>STOP[3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>STOP[2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>STOP[1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>STOP[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-15</td>
<td>Reserved, RES0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>START[7]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>START[6]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>START[5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>START[4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>START[3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>START[2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>START[1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>START[0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:24]**
Reserved, RES0.

**STOP[<m>], bit [m+16], for m = 7 to 0**
Selects whether PE Comparator Input <m> is in use with ViewInst start/stop function, for the purpose of stopping trace.

- **0b0** The PE Comparator Input <m>, is not selected as a stop resource.
- **0b1** The PE Comparator Input <m>, is selected as a stop resource.

This bit is RES0 if m >= TRCIDR4.NUMPC.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Bits [15:8]**
Reserved, RES0.

**START[<m>], bit [m], for m = 7 to 0**
Selects whether PE Comparator Input <m> is in use with ViewInst start/stop function, for the purpose of starting trace.

- **0b0** The PE Comparator Input <m>, is not selected as a start resource.
- **0b1** The PE Comparator Input <m>, is selected as a start resource.

This bit is RES0 if m >= TRCIDR4.NUMPC.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**Accessing the TRCVIPCSSCTL:**
Must be programmed if TRCIDR4.NUMPC != 0b0000.
Writes are **CONSTRAINED UNPREDICTABLE** if the trace unit is not in the Idle state.

TRCVIPCSSCTLR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x08C</td>
<td>TRCVIPCSSCTLR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.77  TRCVISSCTRL, ViewInst Start/Stop Control Register

The TRCVISSCTRL characteristics are:

**Purpose**

Use this to select, or read, the Single Address Comparators for the ViewInst start/stop function.

**Configurations**

External register TRCVISSCTRL bits [31:0] are architecturally mapped to AArch64 System register TRCVISSCTRL[31:0].

This register is present only when FEAT_ETE is implemented and UInt(TRCIDR4.NUMACPAIRS) > 0. Otherwise, direct accesses to TRCVISSCTRL are RES0.

**Attributes**

TRCVISSCTRL is a 32-bit register.

**Field descriptions**

STOP[<m>], bit [m+16], for m = 15 to 0

Selects whether Single Address Comparator <m> is used with the ViewInst start/stop function, for the purpose of stopping trace.

0b0  The Single Address Comparator <m>, is not selected as a stop resource.
0b1  The Single Address Comparator <m>, is selected as a stop resource.

This bit is RES0 if m >= 2 × TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:

* On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

START[<m>], bit [m], for m = 15 to 0

Selects whether Single Address Comparator <m> is used with the ViewInst start/stop function, for the purpose of starting trace.

0b0  The Single Address Comparator <m>, is not selected as a start resource.
0b1  The Single Address Comparator <m>, is selected as a start resource.

This bit is RES0 if m >= 2 × TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:

* On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
Accessing the TRCVISSCTRL:

Must be programmed if TRCIDR4.NUMACPAIRS > 0b0000.

For any 2 comparators selected for the ViewInst start/stop function, the comparator containing the lower address must be a lower numbered comparator.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCVISSCTRL can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x088</td>
<td>TRCVISSCTRL</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When OSlockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
- Otherwise accesses to this register are RW.
H9.3.78 TRCVMIDCCTLR0, Virtual Context Identifier Comparator Control Register 0

The TRCVMIDCCTLR0 characteristics are:

**Purpose**

Virtual Context Identifier Comparator mask values for the TRCVMIDCVR<n> registers, where n=0-3.

**Configurations**

External register TRCVMIDCCTLR0 bits [31:0] are architecturally mapped to AArch64 System register TRCVMIDCCTLR0[31:0].

This register is present only when FEAT_ETE is implemented, UInt(TRCIDR4.NUMVMIDC) > 0x0 and UInt(TRCIDR2.VMIDSIZE) > 0. Otherwise, direct accesses to TRCVMIDCCTLR0 are RES0.

**Attributes**

TRCVMIDCCTLR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP3[m]</td>
<td></td>
<td>COMP2[m]</td>
<td></td>
<td>COMP1[m]</td>
<td></td>
<td>COMP0[m]</td>
<td></td>
</tr>
</tbody>
</table>

**COMP3[m], bit [m+24], for m = 7 to 0**

*When UInt(TRCIDR4.NUMVMIDC) > 3:*

TRCVMIDCVR3 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR3. Each bit in this field corresponds to a byte in TRCVMIDCVR3.

0b0 The trace unit includes TRCVMIDCVR3[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

0b1 The trace unit ignores TRCVMIDCVR3[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

This bit is RES0 if m >= TRCIDR2.VMIDSIZE.

The reset behavior of this field is:

* On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**COMP2[m], bit [m+16], for m = 7 to 0**

*When UInt(TRCIDR4.NUMVMIDC) > 2:*

TRCVMIDCVR2 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR2. Each bit in this field corresponds to a byte in TRCVMIDCVR2.

0b0 The trace unit includes TRCVMIDCVR2[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

0b1 The trace unit ignores TRCVMIDCVR2[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

This bit is RES0 if m >= TRCIDR2.VMIDSIZE.

The reset behavior of this field is:

* On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
COMP1[<m>], bit [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 1:

TRCVMIDCVR1 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR1. Each bit in this field corresponds to a byte in TRCVMIDCVR1.

0b0   The trace unit includes TRCVMIDCVR1[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

0b1   The trace unit ignores TRCVMIDCVR1[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

This bit is RES0 if m >= TRCIDR2.VMIDSIZE.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

COMP0[<m>], bit [m], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 0:

TRCVMIDCVR0 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR0. Each bit in this field corresponds to a byte in TRCVMIDCVR0.

0b0   The trace unit includes TRCVMIDCVR0[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

0b1   The trace unit ignores TRCVMIDCVR0[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

This bit is RES0 if m >= TRCIDR2.VMIDSIZE.

The reset behavior of this field is:

• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing the TRCVMIDCCTLR0:

If software uses the TRCVMIDCVR<n> registers, where n=0-3, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCVMIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCVMIDCVR<n> is not 0x00, the behavior of the Virtual Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCVMIDCCTLR0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x688</td>
<td>TRCVMIDCCTLR0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

• When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.

• Otherwise accesses to this register are RW.
H9.3.79 TRCVMIDCCTRL1, Virtual Context Identifier Comparator Control Register 1

The TRCVMIDCCTRL1 characteristics are:

**Purpose**

Virtual Context Identifier Comparator mask values for the TRCVMIDCVR<\text{n}> registers, where n=4-7.

**Configurations**

External register TRCVMIDCCTRL1 bits [31:0] are architecturally mapped to AArch64 System register TRCVMIDCCTRL1[31:0].

This register is present only when FEAT ETE is implemented, UInt(TRCIDR4.NUMVMIDC) > 0x4 and UInt(TRCIDR2.VMIDSIZE) > 0. Otherwise, direct accesses to TRCVMIDCCTRL1 are RES0.

**Attributes**

TRCVMIDCCTRL1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>COMP7[m]</td>
<td>Virtual Context Identifier Comparator mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR7. Each bit in this field corresponds to a byte in TRCVMIDCVR7.</td>
</tr>
<tr>
<td>28-24</td>
<td>COMP6[m]</td>
<td>Virtual Context Identifier Comparator mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR6. Each bit in this field corresponds to a byte in TRCVMIDCVR6.</td>
</tr>
<tr>
<td>16-15</td>
<td>COMP5[m]</td>
<td>Virtual Context Identifier Comparator mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR5. Each bit in this field corresponds to a byte in TRCVMIDCVR5.</td>
</tr>
<tr>
<td>8-7</td>
<td>COMP4[m]</td>
<td>Virtual Context Identifier Comparator mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR4. Each bit in this field corresponds to a byte in TRCVMIDCVR4.</td>
</tr>
</tbody>
</table>

**COMP7[m], bit [m+24], for m = 7 to 0**

*When UInt(TRCIDR4.NUMVMIDC) > 7:*

- The trace unit includes TRCVMIDCVR7[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
- The trace unit ignores TRCVMIDCVR7[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

This bit is RES0 if m >= TRCIDR2.VMIDSIZE.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

**COMP6[m], bit [m+16], for m = 7 to 0**

*When UInt(TRCIDR4.NUMVMIDC) > 6:*

- The trace unit includes TRCVMIDCVR6[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
- The trace unit ignores TRCVMIDCVR6[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

This bit is RES0 if m >= TRCIDR2.VMIDSIZE.

The reset behavior of this field is:
- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.
COMP5[<m>], bit [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 5:

TRCVMIDCVR5 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR5. Each bit in this field corresponds to a byte in TRCVMIDCVR5.

0b0  The trace unit includes TRCVMIDCVR5[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

0b1  The trace unit ignores TRCVMIDCVR5[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

This bit is RES0 if m >= TRCIDR2.VMIDSIZE.

The reset behavior of this field is:
•  On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

COMP4[<m>], bit [m], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 4:

TRCVMIDCVR4 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR4. Each bit in this field corresponds to a byte in TRCVMIDCVR4.

0b0  The trace unit includes TRCVMIDCVR4[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

0b1  The trace unit ignores TRCVMIDCVR4[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

This bit is RES0 if m >= TRCIDR2.VMIDSIZE.

The reset behavior of this field is:
•  On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing the TRCVMIDCCTRL1:

If software uses the TRCVMIDCVR<n> registers, where n=4-7, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCVMIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCVMIDCVR<n> is not 0x00, the behavior of the Virtual Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCVMIDCCTRL1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x68C</td>
<td>TRCVMIDCCTRL1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

•  When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.

•  Otherwise accesses to this register are RW.
H9.3.80  TRCVMIDCVR<n>, Virtual Context Identifier Comparator Value Register <n>, n = 0 - 7

The TRCVMIDCVR<n> characteristics are:

**Purpose**
Contains the Virtual Context Identifier Comparator value.

**Configurations**
External register TRCVMIDCVR<n> bits [63:0] are architecturally mapped to AArch64 System register TRCVMIDCVR<n>[63:0].
This register is present only when FEAT_ETE is implemented and UInt(TRCIDR4.NUMVMIDC) > n. Otherwise, direct accesses to TRCVMIDCVR<n> are RES0.

**Attributes**
TRCVMIDCVR<n> is a 64-bit register.

**Field descriptions**

VALUE, bits [63:0]
Virtual context identifier value. The width of this field is indicated by TRCIDR2.VMIDSIZE. Unimplemented bits are RES0. After a PE Reset, the trace unit assumes that the Virtual context identifier is zero until the PE updates the Virtual context identifier.
The reset behavior of this field is:
• On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing the TRCVMIDCVR<n>:
Must be programmed if any of the following are true:
• TRCRSCTLR<n>.GROUP == 0b0111 and TRCRSCTLR<n>.VMID[n] == 1.
• TRCACATR<n>.CONTEXTTYPE == 0b10 or 0b11 and TRCACATR<n>.CONTEXT == n.

TRCVMIDCVR<n> can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETE</td>
<td>0x640 + (8 * n)</td>
<td>TRCVMIDCVR&lt;n&gt;</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:
• When OSLockStatus(), or !AllowExternalTraceAccess() or !IsTraceCorePowered() accesses to this register generate an error response.
• Otherwise accesses to this register are RW.
H9.4 Cross-Trigger Interface registers

This section lists the Cross-Trigger Interface registers.
H9.4.1 ASICCTL, CTI External Multiplexer Control register

The ASICCTL characteristics are:

**Purpose**

Can be used to provide IMPLEMENTATION DEFINED controls for the CTI. For example, the register might be used to control multiplexors for additional IMPLEMENTATION DEFINED triggers. The IMPLEMENTATION DEFINED controls provided by this register might modify the architecturally defined behavior of the CTI.

--- **Note** ---

The architecturally-defined triggers must not be multiplexed.

**Configurations**

It is IMPLEMENTATION DEFINED whether ASICCTL is implemented in the Core power domain or in the Debug power domain.

If it is implemented in the Core power domain then it is IMPLEMENTATION DEFINED whether it is in the Cold reset domain or the Warm reset domain.

This register must reset to a value that supports the architecturally-defined behavior of the CTI. Changing the value of the register from its reset value causes IMPLEMENTATION DEFINED behavior that might differ from the architecturally-defined behavior of the CTI.

Other than the requirements listed in this register description, all aspects of the reset behavior of the ASICCTL are IMPLEMENTATION DEFINED.

**Attributes**

ASICCTL is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

**Accessing the ASICCTL:**

ASICCTL can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0x144</td>
<td>ASICCTL</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess() and SoftwareLockStatus() accesses to this register are RO.

- Otherwise accesses to this register are IMPDEF.
H9.4.2 CTIAPPCLEAR, CTI Application Trigger Clear register

The CTIAPPCLEAR characteristics are:

**Purpose**
Clears the application triggers.

**Configurations**
CTIAPPCLEAR is in the Debug power domain.

**Attributes**
CTIAPPCLEAR is a 32-bit register.

**Field descriptions**

APPCLEAR<x>, bit [x], for x = 31 to 0
Application trigger <x> disable.

Bits [31:N] are RAZ/WI. N is the number of ECT channels implemented as defined by the CTIDVID.NUMCHAN field.

Writing to this bit has the following effect:

- **0b0**: No effect.
- **0b1**: Clear corresponding application trigger to 0 and clear the corresponding channel event.

If the ECT does not support multicycle channel events, use of CTIAPPCLEAR is deprecated and the debugger must only use CTIAPPULSE.

**Accessing the CTIAPPCLEAR:**

CTIAPPCLEAR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0x018</td>
<td>CTIAPPCLEAR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are WI.
• When !SoftwareLockStatus() accesses to this register are WO.
### H9.4.3 CTIAPPPULSE, CTI Application Pulse register

The CTIAPPPULSE characteristics are:

**Purpose**
Causes event pulses to be generated on ECT channels.

**Configurations**
CTIAPPPULSE is in the Debug power domain.

**Attributes**
CTIAPPPULSE is a 32-bit register.

**Field descriptions**

**APPPULSE<\textit{x}>, bit \!<\textit{x}>!, for \textit{x} = 31 to 0**
Generate event pulse on ECT channel \!<\textit{x}>.

Bits [31:N] are RAZ/WI. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Writing to this bit has the following effect:

- \(0b0\): No effect.
- \(0b1\): Channel \!<\textit{x}> event pulse generated.

**Note**
- The CTIAPPPULSE operation does not affect the state of the application trigger. If the channel is active, either because of an earlier event or from the application trigger, then the value written to CTIAPPPULSE might have no effect.
- Multiple pulse events that occur close together might be merged into a single pulse event.

**Accessing the CTIAPPPULSE:**

It is **CONSTRAINED UNPREDICTABLE** whether a write to CTIAPPPULSE generates an event on a channel if CTICONTROL.GLBEN is 0.
CTIAPPPULSE can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0x01c</td>
<td>CTIAPPPULSE</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are WI.
- When !SoftwareLockStatus() accesses to this register are WO.
H9.4.4 CTIAPPSET, CTI Application Trigger Set register

The CTIAPPSET characteristics are:

Purpose
Sets the application triggers.

Configurations
CTIAPPSET is in the Debug power domain.

Attributes
CTIAPPSET is a 32-bit register.

Field descriptions

APPSET<x>, bit [x], for x = 31 to 0
Application trigger <x> enable.

0b0 Reading this means the application trigger is inactive. Writing this has no effect.
0b1 Reading this means the application trigger is active. Writing this sets the corresponding
application trigger to 1 and generates a channel event.

If the ECT does not support multicycle channel events, use of CTIAPPSET is deprecated and the
debugger must only use CTIAPPPULSE.

The reset behavior of this field is:

• On an External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing the CTIAPPSET:
CTIAPPSET can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0x14</td>
<td>CTIAPPSET</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are RO.
- When !SoftwareLockStatus() accesses to this register are RW.
H9.4.5 CTIAUTHSTATUS, CTI Authentication Status register

The CTIAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for CTI.

Configurations

CTIAUTHSTATUS is in the Debug power domain.

This register is OPTIONAL, and is required for CoreSight compliance.

Attributes

CTIAUTHSTATUS is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>28 - 27</th>
<th>24 - 23</th>
<th>16 - 15</th>
<th>12 - 11</th>
<th>8 - 7</th>
<th>4 - 3</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>RAZ</td>
<td>RES0</td>
<td>RAZ</td>
<td>RES0</td>
<td>RAZ</td>
<td>NSNID</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits [31:28]

Reserved, RES0.

Bits [27:24]

Reserved, RAZ.

Bits [23:16]

Reserved, RES0.

Bits [15:12]

Reserved, RAZ.

Bits [11:8]

Reserved, RES0.

Bits [7:4]

Reserved, RAZ.

NSNID, bits [3:2]

If EL3 is implemented, this field holds the same value as DBGAUTHSTATUS_EL1.NSNID.

If EL3 is not implemented and the implemented Security state is Secure state, this field holds the same value as DBGAUTHSTATUS_EL1.SNID.

NSID, bits [1:0]

If EL3 is implemented, this field holds the same value as DBGAUTHSTATUS_EL1.NSID.

If EL3 is not implemented and the implemented Security state is Secure state, this field holds the same value as DBGAUTHSTATUS_EL1.SID.
Accessing the CTIAUTHSTATUS:

CTIAUTHSTATUS can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFB8</td>
<td>CTIAUTHSTATUS</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.6 CTICHINSTATUS, CTI Channel In Status register

The CTICHINSTATUS characteristics are:

**Purpose**
Provides the raw status of the ECT channel inputs to the CTI.

**Configurations**
CTICHINSTATUS is in the Debug power domain.

**Attributes**
CTICHINSTATUS is a 32-bit register.

**Field descriptions**

CHIN<n>, bit [n], for n = 31 to 0
Input channel <n> status.

Bits [31:N] are RAZ. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

0b0 Input channel <n> is inactive.
0b1 Input channel <n> is active.

If the ECT channels do not support multicycle events then it is IMPLEMENTATION DEFINED whether an input channel can be observed as active.

**Accessing the CTICHINSTATUS:**
CTICHINSTATUS can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0x138</td>
<td>CTICHINSTATUS</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.7 CTICHOUTSTATUS, CTI Channel Out Status register

The CTICHOUTSTATUS characteristics are:

**Purpose**
Provides the status of the ECT channel outputs from the CTI.

**Configurations**
CTICHOUTSTATUS is in the Debug power domain.

**Attributes**
CTICHOUTSTATUS is a 32-bit register.

**Field descriptions**

CHOUT<\(n\)>, bit \([n]\), for \(n = 31\) to \(0\)
Output channel \(<n>\) status.

Bits \([31:N]\) are RAZ, \(N\) is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Possible values of this bit are:

- **0b0**: Output channel \(<n>\) is inactive.
- **0b1**: Output channel \(<n>\) is active.

If the ECT channels do not support multicycle events then it is IMPLEMENTATION DEFINED whether an output channel can be observed as active.

--- **Note** ---

The value in CTICHOUTSTATUS is after gating by the channel gate. For more information, see CTIGATE.

---

**Accessing the CTICHOUTSTATUS:**

CTICHOUTSTATUS can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0x13C</td>
<td>CTICHOUTSTATUS</td>
</tr>
</tbody>
</table>
Accesses to this interface are RO.
H9.4.8   CTICIDR0, CTI Component Identification Register 0

The CTICIDR0 characteristics are:

Purpose

Provides information to identify a CTI component.
For more information, see About the Component Identification scheme on page K6-11699.

Configurations

CTICIDR0 is in the Debug power domain.
Implementation of this register is OPTIONAL.
This register is required for CoreSight compliance.

Attributes

CTICIDR0 is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bits [31:8]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRMBL_0, bits [7:0]</td>
<td>Preamble. Reads as 0x0D. Access to this field is RO.</td>
</tr>
</tbody>
</table>

Accessing the CTICIDR0:

CTICIDR0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFF0</td>
<td>CTICIDR0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.9 CTICIDR1, CTI Component Identification Register 1

The CTICIDR1 characteristics are:

**Purpose**

Provides information to identify a CTI component.

For more information, see *About the Component Identification scheme on page K6-11699*.

**Configurations**

CTICIDR1 is in the Debug power domain.

Implementation of this register is **OPTIONAL**.

This register is required for CoreSight compliance.

**Attributes**

CTICIDR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:8</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>7:4</td>
<td>CLASS</td>
<td>0b1001</td>
</tr>
<tr>
<td></td>
<td>Component class</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CoreSight component</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Other values are defined by the CoreSight Architecture.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This field reads as 0x9.</td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td>PRMBL_1</td>
<td>0b0000</td>
</tr>
<tr>
<td></td>
<td>Preamble. RAZ.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reads as 0b0000.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Access to this field is RO.</td>
<td></td>
</tr>
</tbody>
</table>

**Accessing the CTICIDR1:**

CTICIDR1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFF4</td>
<td>CTICIDR1</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.10 CTICIDR2, CTI Component Identification Register 2

The CTICIDR2 characteristics are:

**Purpose**

- Provides information to identify a CTI component.
- For more information, see *About the Component Identification scheme on page K6-11699*.

**Configurations**

- CTICIDR2 is in the Debug power domain.
- Implementation of this register is **OPTIONAL**.
- This register is required for CoreSight compliance.

**Attributes**

- CTICIDR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:8]**

- Reserved, RES0.

**PRMBL_2, bits [7:0]**

- Preamble.
- Reads as 0x05.
- Access to this field is **RO**.

**Accessing the CTICIDR2:**

CTICIDR2 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFF8</td>
<td>CTICIDR2</td>
</tr>
</tbody>
</table>

Accesses to this interface are **RO**.
H9.4.11  CTICIDR3, CTI Component Identification Register 3

The CTICIDR3 characteristics are:

**Purpose**

Provides information to identify a CTI component.

For more information, see *About the Component Identification scheme on page K6-11699*.

**Configurations**

CTICIDR3 is in the Debug power domain.

Implementation of this register is OPTIONAL.

This register is required for CoreSight compliance.

**Attributes**

CTICIDR3 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:8</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>7:0</td>
<td>PRMBL_3, Preamble. Reads as 0xB1. Access to this field is RO.</td>
</tr>
</tbody>
</table>

**Accessing the CTICIDR3:**

CTICIDR3 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFFC</td>
<td>CTICIDR3</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.12 CTICLAIMCLR, CTI CLAIM Tag Clear register

The CTICLAIMCLR characteristics are:

**Purpose**

Used by software to read the values of the CLAIM bits, and to clear CLAIM tag bits to 0.

**Configurations**

CTICLAIMCLR is in the Debug power domain.

Implementation of this register is OPTIONAL.

**Attributes**

CTICLAIMCLR is a 32-bit register.

**Field descriptions**

CLAIM<x>, bit [x], for x = 31 to 0

CLAIM tag clear bit.

Reads return the value of CLAIM<x>, writes have the following behavior:

- **0b0** No action.
- **0b1** Indirectly clear CLAIM<x> to 0.

A single write to CTICLAIMCLR can clear multiple tags to 0.

If x is greater than or equal to the IMPLEMENTATION DEFINED number of CLAIM tags, this bit is RAZ/WI.

The reset behavior of this field is:

- An External Debug reset clears the CLAIM tag bits to 0.

**Accessing the CTICLAIMCLR:**

CTICLAIMCLR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFA4</td>
<td>CTICLAIMCLR</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are RO.
- When !SoftwareLockStatus() accesses to this register are RW.
H9.4.13  **CTICLAIMSET, CTI CLAIM Tag Set register**

The CTICLAIMSET characteristics are:

**Purpose**

Used by software to set CLAIM bits to 1.

**Configurations**

CTICLAIMSET is in the Debug power domain.

Implementation of this register is OPTIONAL.

**Attributes**

CTICLAIMSET is a 32-bit register.

**Field descriptions**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| CLAIM31 | CLAIM30 | CLAIM29 | CLAIM28 | CLAIM27 | CLAIM26 | CLAIM25 | CLAIM24 | CLAIM23 | CLAIM22 | CLAIM21 | CLAIM20 | CLAIM19 | CLAIM18 | CLAIM17 | CLAIM16 | CLAIM15 | CLAIM14 | CLAIM13 | CLAIM12 | CLAIM11 | CLAIM10 | CLAIM9 | CLAIM8 | CLAIM7 | CLAIM6 | CLAIM5 | CLAIM4 | CLAIM3 | CLAIM2 | CLAIM1 | CLAIM0 |

CLAIM<x>, bit [x], for x = 31 to 0

CLAIM tag set bit.

If x is less than the IMPLEMENTATION DEFINED number of CLAIM tags, this field is RAO and the behavior on writes is:

0b0  No action.

0b1  Indirectly set CLAIM<x> tag to 1.

A single write to CTICLAIMSET can set multiple tags to 1.

If x is greater than or equal to the IMPLEMENTATION DEFINED number of CLAIM tags, this bit is RAZ/WI.

The reset behavior of this field is:

- An External Debug reset clears the CLAIM tag bits to 0.

**Accessing the CTICLAIMSET:**

CTICLAIMSET can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFA0</td>
<td>CTICLAIMSET</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are RO.
- When !SoftwareLockStatus() accesses to this register are RW.
H9.4.14 CTICONTROL, CTI Control register

The CTICONTROL characteristics are:

**Purpose**

Controls whether the CTI is enabled.

**Configurations**

CTICONTROL is in the Debug power domain.

**Attributes**

CTICONTROL is a 32-bit register.

**Field descriptions**

![Register Diagram]

**Bits [31:1]**

Reserved, RES0.

**GLBEN, bit [0]**

Enables or disables the CTI mapping functions. Possible values of this field are:

0b0   CTI mapping functions and application trigger disabled.
0b1   CTI mapping functions and application trigger enabled.

When GLBEN is 0, the input channel to output trigger, input trigger to output channel, and application trigger functions are disabled and do not signal new events on either output triggers or output channels. If a previously asserted output trigger has not been acknowledged, it is **CONSTRAINED UNPREDICTABLE** which of the following occurs:

- The output trigger remains asserted after the mapping functions are disabled.
- The output trigger is deasserted after the mapping functions are disabled.

All output triggers are disabled by CTI reset.

If the ECT supports multicycle channel events any existing output channel events will be terminated.

The reset behavior of this field is:

- On an External debug reset, this field resets to 0.

**Accessing the CTICONTROL:**

CTICONTROL can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0x0000</td>
<td>CTICONTROL</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are RO.
- When !SoftwareLockStatus() accesses to this register are RW.
H9.4.15 CTIDEVAFF0, CTI Device Affinity register 0

The CTIDEVAFF0 characteristics are:

**Purpose**

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the CTI component relates to.

**Configurations**

CTIDEVAFF0 is in the Debug power domain.

If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.

Arm recommends that the CTI is CTIv2.

In an Armv8.5 compliant implementation, the CTI must be CTIv2.

If this register is implemented, then CTIDEVAFF1 must also be implemented. If the CTI of a PE does not implement the CTI Device Affinity registers, the CTI block of the external debug memory map must be located 64KB above the debug registers in the external debug interface.

**Attributes**

CTIDEVAFF0 is a 32-bit register.

**Field descriptions**

![MPIDR_EL1lo](31 0)

**MPIDR_EL1lo, bits [31:0]**

MPIDR_EL1 low half. Read-only copy of the low half of MPIDR_EL1, as seen from the highest implemented Exception level.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Accessing the CTIDEVAFF0:**

CTIDEVAFF0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFA8</td>
<td>CTIDEVAFF0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.16 CTIDEVAFF1, CTI Device Affinity register 1

The CTIDEVAFF1 characteristics are:

**Purpose**
Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the CTI component relates to.

**Configurations**
CTIDEVAFF1 is in the Debug power domain.
If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.
Arm recommends that the CTI is CTIv2.
In an Armv8.5 compliant implementation, the CTI must be CTIv2.
If this register is implemented, then CTIDEVAFF0 must also be implemented. If the CTI of a PE does not implement the CTI Device Affinity registers, the CTI block of the external debug memory map must be located 64KB above the debug registers in the external debug interface.

**Attributes**
CTIDEVAFF1 is a 32-bit register.

**Field descriptions**

```
+---------+---------+
| 31      | 0       |
+---------+---------+
MPIDR_EL1hi
```

**MPIDR_EL1hi, bits [31:0]**
MPIDR_EL1 high half. Read-only copy of the high half of MPIDR_EL1, as seen from the highest implemented Exception level.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

**Accessing the CTIDEVAFF1:**
CTIDEVAFF1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFAC</td>
<td>CTIDEVAFF1</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.17 CTIDEVARCH, CTI Device Architecture register

The CTIDEVARCH characteristics are:

**Purpose**

Identifies the programmers' model architecture of the CTI component.

**Configurations**

CTIDEVARCH is in the Debug power domain.

- If the CTI is CTIv1, this register is **optional**. If the CTI is CTIv2, this register is mandatory.
- Arm recommends that the CTI is CTIv2.
- In an Armv8.5 compliant implementation, the CTI must be CTIv2.
- If this register is not implemented, CTIDEVAFF0 and CTIDEVAFF1 are also not implemented.

**Attributes**

CTIDEVARCH is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:21 ARCHITECT</td>
<td>Defines the architecture of the component. For CTI, this is Arm Limited.</td>
</tr>
<tr>
<td>20 PRESENT</td>
<td>Indicates that the DEVARCH is present.</td>
</tr>
</tbody>
</table>
| 19:16 REVISION | Defines the architecture revision of the component. When FEAT_DoPD is implemented: 

- 0b0000 First revision.
- 0b0001 As 0b0000, and also adds support for CTIDEVCTL.
- All other values are reserved.

This field has an **IMPLEMENTATION DEFINED** value.

Access to this field is RO.

**Otherwise:**

- Revision.
- Defines the architecture revision of the component.
- All other values are reserved.
Reads as 0b0000.
Access to this field is RO.

**ARCHID, bits [15:0]**

Defines this part to be an Armv8 debug component. For architectures defined by Arm this is further subdivided.

For CTI:
- Bits [15:12] are the architecture version, 0x1.
- Bits [11:0] are the architecture part number, 0xA14.

This corresponds to CTI architecture version CTIv2.
Reads as 0xA14.
Access to this field is RO.

**Accessing the CTIDEVARCH:**

CTIDEVARCH can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xF8C</td>
<td>CTIDEVARCH</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
CTIDEVCTL, CTI Device Control register

The CTIDEVCTL characteristics are:

**Purpose**

Provides target-specific device controls

**Configurations**

CTIDEVCTL is in the Debug power domain.

This register is present only when FEAT_DoPD is implemented. Otherwise, direct accesses to CTIDEVCTL are RES0.

**Attributes**

CTIDEVCTL is a 32-bit register.

**Field descriptions**

![Field descriptions diagram]

**Bits [31:2]**

Reserved, RES0.

**RCE, bit [1]**

Reset Catch Enable.

0b0 Reset Catch debug event disabled.
0b1 Reset Catch debug event enabled.

The reset behavior of this field is:

- On an External debug reset, this field resets to 0.

**OSUCE, bit [0]**

OS Unlock Catch Enable

0b0 OS Unlock Catch debug event disabled.
0b1 OS Unlock Catch debug event enabled.

The reset behavior of this field is:

- On an External debug reset, this field resets to 0.

**Accessing the CTIDEVCTL:**

CTIDEVCTL can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0x150</td>
<td>CTIDEVCTL</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are RO.
- When !SoftwareLockStatus() accesses to this register are RW.
H9.4.19 CTIDEVID, CTI Device ID register 0

The CTIDEVID characteristics are:

**Purpose**

Describes the CTI component to the debugger.

**Configurations**

CTIDEVID is in the Debug power domain.

**Attributes**

CTIDEVID is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>18</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>RES0</td>
<td>NUMCHAN</td>
<td>RES0</td>
<td>NUMTRIG</td>
<td>RES0</td>
<td>EXTNUMNUM</td>
<td>INOUT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:26]**

Reserved, RES0.

**INOUT, bits [25:24]**

Input/output options. Indicates presence of the input gate. If the CTM is not implemented or CTIv2 is not implemented, this field is RAZ.

- 0b00 **CTIGATE** does not mask propagation of input events from external channels.
- 0b01 **CTIGATE** masks propagation of input events from external channels.

All other values are reserved.

**Bits [23:22]**

Reserved, RES0.

**NUMCHAN, bits [21:16]**

Number of ECT channels implemented. For Armv8, valid values are:

- 0b000011 3 channels (0..2) implemented.
- 0b000100 4 channels (0..3) implemented.
- 0b000101 5 channels (0..4) implemented.
- 0b000110 6 channels (0..5) implemented.

and so on up to 0b100000, 32 channels (0..31) implemented.

All other values are reserved.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Bits [15:14]**

Reserved, RES0.

**NUMTRIG, bits [13:8]**

Upper bound for number of triggers. The indices of all implemented input and output triggers are less than this value.
All other values are reserved. If the PE contains a Trace extension, this field must be at least 0b001000. There is no guarantee that all of the input and output triggers, including the highest numbered, are connected to any components, or that the implementation of input and output triggers is symmetrical.

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

**Bits [7:5]**

Reserved, RES0.

**EXTMUXNUM, bits [4:0]**

Number of multiplexors available on triggers. This value is used in conjunction with External Control register, ASICCTL.

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

**Accessing the CTIDEVID:**

CTIDEVID can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFC8</td>
<td>CTIDEVID</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.20  CTIDEVID1, CTI Device ID register 1

The CTIDEVID1 characteristics are:

**Purpose**
Reserved for future information about the CTI component to the debugger.

**Configurations**
CTIDEVID1 is in the Debug power domain.

**Attributes**
CTIDEVID1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th>Reserved, RES0</th>
</tr>
</thead>
</table>

**Accessing the CTIDEVID1:**
CTIDEVID1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFC4</td>
<td>CTIDEVID1</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.21 CTIDEVID2, CTI Device ID register 2

The CTIDEVID2 characteristics are:

**Purpose**
Reserved for future information about the CTI component to the debugger.

**Configurations**
CTIDEVID2 is in the Debug power domain.

**Attributes**
CTIDEVID2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:0]</th>
<th>Reserved, RES0</th>
</tr>
</thead>
</table>

Accessing the CTIDEVID2:

CTIDEVID2 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFC0</td>
<td>CTIDEVID2</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.22  CTIDEVTYPE, CTI Device Type register

The CTIDEVTYPE characteristics are:

Purpose
Indicates to a debugger that this component is part of a PEs cross-trigger interface.

Configurations
CTIDEVTYPE is in the Debug power domain.
Implementation of this register is OPTIONAL.

Attributes
CTIDEVTYPE is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0</td>
<td>0000</td>
</tr>
<tr>
<td>7-4</td>
<td>Subtype, SUB</td>
<td>0001</td>
</tr>
<tr>
<td>3-0</td>
<td>Major Type, MAJOR</td>
<td>0100</td>
</tr>
</tbody>
</table>

Bits [31:8]
Reserved, RES0.

SUB, bits [7:4]
Subtype. Indicates this is a component within a PE.
Reads as 0b0001.
Access to this field is RO.

MAJOR, bits [3:0]
Major type. Indicates this is a cross-trigger component.
Reads as 0b0100.
Access to this field is RO.

Accessing the CTIDEVTYPE:
CTIDEVTYPE can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFCC</td>
<td>CTIDEVTYPE</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.23  **CTIGATE, CTI Channel Gate Enable register**

The CTIGATE characteristics are:

**Purpose**

Determines whether events on channels propagate through the CTM to other ECT components, or from the CTM into the CTI.

**Configurations**

CTIGATE is in the Debug power domain.

**Attributes**

CTIGATE is a 32-bit register.

**Field descriptions**

GATE\(<x>\), bit \([x]\), for \(x = 31\) to \(0\)

Channel \(<x>\) gate enable.

Bits \([31: N]\) are RAZ/WI. \(N\) is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

- **0b0** Disable output and, if CTIDEVID.INOUT == 0b01, input channel \(<x>\) propagation.
- **0b1** Enable output and, if CTIDEVID.INOUT == 0b01, input channel \(<x>\) propagation.

If GATE\(<x>\) is set to 0, no new events will be propagated to the ECT, and if the ECT supports multicycle channel events any existing output channel events will be terminated.

The reset behavior of this field is:

- On an External debug reset, this field resets to an architecturally UNKNOWN value.

**Accessing the CTIGATE:**

CTIGATE can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0x140</td>
<td>CTIGATE</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When `SoftwareLockStatus()` accesses to this register are RO.
- When `!SoftwareLockStatus()` accesses to this register are RW.
H9.4.24  CTIINEN<n>, CTI Input Trigger to Output Channel Enable registers, n = 0 - 31

The CTIINEN<n> characteristics are:

**Purpose**

Enables the signaling of an event on output channels when input trigger event n is received by the CTI.

**Configurations**

CTIINEN<n> is in the Debug power domain.

If input trigger n is not implemented or not connected, CTIINEN<n> is RES0.

**Attributes**

CTIINEN<n> is a 32-bit register.

**Field descriptions**

INEN<x>, bit [x], for x = 31 to 0

Input trigger <n> to output channel <x> enable.

Bits [31:N] are RAZ/WI. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

- 0b0 Input trigger <n> will not generate an event on output channel <x>.
- 0b1 Input trigger <n> will generate an event on output channel <x>.

The reset behavior of this field is:

- On an External debug reset, this field resets to an architecturally UNKNOWN value.

**Accessing the CTIINEN<n>:**

CTIINEN<n> can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0x820 + (4 * n)</td>
<td>CTIINEN&lt;n&gt;</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are RO.
• When !SoftwareLockStatus() accesses to this register are RW.
**H9.4.25  CTIINTACK, CTI Output Trigger Acknowledge register**

The CTIINTACK characteristics are:

**Purpose**
Can be used to deactivate the output triggers.

**Configurations**
CTIINTACK is in the Debug power domain.

**Attributes**
CTIINTACK is a 32-bit register.

**Field descriptions**

---

**ACK<n>, bit [n], for n = 31 to 0**

Acknowledge for output trigger <n>.  
Bits [31:N] are RAZ/WI. N is the number of CTI triggers implemented as defined by the CTIDEVID.NUMTRIG field.  

If any of the following is true, writes to ACK<n> are ignored:

- n \(\geq\) CTIDEVID.NUMTRIG, the number of implemented triggers.  
- Output trigger n is not active.  
- The channel mapping function output, as controlled by CTIOUTEN<n>, is still active.

Otherwise, if any of the following are true, ACK<n> is RES0:

- Output trigger n is not implemented.  
- Output trigger n is not connected.  
- Output trigger n is self-acknowledging and does not require software acknowledge.

Otherwise, the behavior on writes to ACK<n> is as follows:

- 0b0  No effect  
- 0b1  Deactivate the trigger.
**Accessing the CTIINTACK:**

A debugger must read CTITRIGOUTSTATUS to confirm that the output trigger has been acknowledged before generating any event that must be ordered after the write to CTIINTACK, such as a write to CTIAPPPULSE to activate another trigger.

CTIINTACK can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0x10</td>
<td>CTIINTACK</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When SoftwareLockStatus() accesses to this register are WI.
- When !SoftwareLockStatus() accesses to this register are WO.
H9.4.26  CTIITCTRL, CTI Integration mode Control register

The CTIITCTRL characteristics are:

**Purpose**

Enables the CTI to switch from its default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

**Configurations**

It is IMPLEMENTATION DEFINED whether CTIITCTRL is implemented in the Core power domain or in the Debug power domain.

Implementation of this register is OPTIONAL.

**Attributes**

CTIITCTRL is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:1</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>0</td>
<td>IME, Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection. The integration mode behavior is IMPLEMENTATION DEFINED.</td>
</tr>
</tbody>
</table>

0b0 Normal operation.

0b1 Integration mode enabled.

The reset behavior of this field is:

- The following resets apply:
  - If the register is implemented in the Core power domain:
    - On a Cold reset, this field resets to 0.
    - On an External debug reset, the value of this field is unchanged.
    - On a Warm reset, the value of this field is unchanged.
  - If the register is implemented in the External debug power domain:
    - On a Cold reset, the value of this field is unchanged.
    - On an External debug reset, this field resets to 0.
    - On a Warm reset, the value of this field is unchanged.

**Accessing the CTIITCTRL:**

CTIITCTRL can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xF00</td>
<td>CTIITCTRL</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register are IMPDEF.
H9.4.27 **CTILAR, CTI Lock Access Register**

The CTILAR characteristics are:

**Purpose**

Allows or disallows access to the CTI registers through a memory-mapped interface.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Cross-Trigger Interface registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Cross-Trigger Interface registers. It does not, and cannot, prevent all accidental or malicious damage.

**Configurations**

CTILAR is in the Debug power domain.

If FEAT_Debugv8p4 is implemented, the Software Lock is not implemented.

Software uses CTILAR to set or clear the lock, and CTILSR to check the current status of the lock.

**Attributes**

CTILAR is a 32-bit register.

**Field descriptions**

*When Software Lock is implemented:*

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>KEY</td>
</tr>
</tbody>
</table>

**KEY, bits [31:0]**

Lock Access control. Writing the key value \(0xC5ACCE55\) to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

**Otherwise:**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>RES0</td>
</tr>
</tbody>
</table>

Otherwise

**Bits [31:0]**

Reserved, RES0.

**Accessing the CTILAR:**

CTILAR can be accessed through a memory-mapped interface access to the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xF80</td>
<td>CTILAR</td>
</tr>
</tbody>
</table>

Accesses to this interface are WO.
H9.4.28   CTILSR, CTI Lock Status Register

The CTILSR characteristics are:

**Purpose**

Indicates the current status of the Software Lock for CTI registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Cross-Trigger Interface registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Cross-Trigger Interface registers. It does not, and cannot, prevent all accidental or malicious damage.

**Configurations**

CTILSR is in the Debug power domain.

If FEAT_Debugv8p4 is implemented, the Software Lock is not implemented.

Software uses CTILAR to set or clear the lock, and CTILSR to check the current status of the lock.

**Attributes**

CTILSR is a 32-bit register.

**Field descriptions**

![Field descriptions diagram]

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value 0x01</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>nTT, bit [2]</td>
<td>Not thirty-two bit access required. RAZ.</td>
</tr>
<tr>
<td>3</td>
<td>SLK, bit [1]</td>
<td>When Software Lock is implemented:</td>
</tr>
<tr>
<td>2</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SLI, bit [0]</td>
<td>Otherwise:</td>
</tr>
<tr>
<td>0</td>
<td>Reserved, RAZ</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:3]**

Reserved, RES0.

**nTT, bit [2]**

Not thirty-two bit access required. RAZ.

**SLK, bit [1]**

*When Software Lock is implemented:*

Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when the Software Lock is not implemented, this field is RES0.

For memory-mapped accesses when the Software Lock is implemented, possible values of this field are:

- 0b0 Lock clear. Writes are permitted to this component's registers.
- 0b1 Lock set. Writes to this component's registers are ignored, and reads have no side effects.

The reset behavior of this field is:

- On an External debug reset, this field resets to 1.

*Otherwise:*

Reserved, RAZ.

**SLI, bit [0]**

Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ. For memory-mapped accesses, the value of this field is IMPLEMENTATION DEFINED.

Permitted values are:

- 0b0 Software Lock not implemented or not memory-mapped access.
- 0b1 Software Lock implemented and memory-mapped access.
Accessing the CTILSR:

CTILSR can be accessed through a memory-mapped interface access to the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xF84</td>
<td>CTILSR</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.29 CTIOUTEN<n>, CTI Input Channel to Output Trigger Enable registers, n = 0 - 31

The CTIOUTEN<n> characteristics are:

**Purpose**

Defines which input channels generate output trigger n.

**Configurations**

CTIOUTEN<n> is in the Debug power domain.

If output trigger n is not implemented or not connected, CTIOUTEN<n> is RES0.

**Attributes**

CTIOUTEN<n> is a 32-bit register.

**Field descriptions**

OUTEN<x>, bit [x], for x = 31 to 0

Input channel <x> to output trigger <n> enable.

Bits [31:N] are RAZ/WI. N is the number of ECT channels implemented as defined by the CTIDEVID.NUMCHAN field.

Possible values of this bit are:

- 0b0 An event on input channel <x> will not cause output trigger <n> to be asserted.
- 0b1 An event on input channel <x> will cause output trigger <n> to be asserted.

The reset behavior of this field is:

- On an External debug reset, this field resets to an architecturally UNKNOWN value.

**Accessing the CTIOUTEN<n>:**

CTIOUTEN<n> can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xA0 + (4 * n)</td>
<td>CTIOUTEN&lt;n&gt;</td>
</tr>
</tbody>
</table>


This interface is accessible as follows:

- When `SoftwareLockStatus()` accesses to this register are RO.
- When `!SoftwareLockStatus()` accesses to this register are RW.
H9.4.30  CTIPIDR0, CTI Peripheral Identification Register 0

The CTIPIDR0 characteristics are:

Purpose
Provides information to identify a CTI component.
For more information, see About the Peripheral identification scheme on page K6-11696.

Configurations
CTIPIDR0 is in the Debug power domain.
Implementation of this register is OPTIONAL.
This register is required for CoreSight compliance.

Attributes
CTIPIDR0 is a 32-bit register.

Field descriptions

| Bits [31:8] | RESERVED, RES0 |
| PART_0, bits [7:0] | PART_0 |

Accessing the CTIPIDR0:
CTIPIDR0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFE0</td>
<td>CTIPIDR0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.31 CTIPIDR1, CTI Peripheral Identification Register 1

The CTIPIDR1 characteristics are:

**Purpose**

Provides information to identify a CTI component.
For more information, see *About the Peripheral identification scheme* on page K6-11696.

**Configurations**

CTIPIDR1 is in the Debug power domain.
Implementation of this register is OPTIONAL.
This register is required for CoreSight compliance.

**Attributes**

CTIPIDR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits [31:8]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>DES_0</td>
<td>Designer, least significant nibble of JEP106 ID code. For Arm Limited, this field is 0b1011.</td>
</tr>
<tr>
<td>PART_1</td>
<td>Part number, most significant nibble.</td>
</tr>
</tbody>
</table>

Access to this field is RO.

Access to this field is RO.

**Accessing the CTIPIDR1:**

CTIPIDR1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFE4</td>
<td>CTIPIDR1</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.32 CTIPIDR2, CTI Peripheral Identification Register 2

The CTIPIDR2 characteristics are:

**Purpose**

Provides information to identify a CTI component.

For more information, see "About the Peripheral identification scheme on page K6-11696."

**Configurations**

CTIPIDR2 is in the Debug power domain.

Implementation of this register is *OPTIONAL*.

This register is required for CoreSight compliance.

**Attributes**

CTIPIDR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:8</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>7:4</td>
<td>REVISION, Part major revision. Parts can also use this field to extend Part number to 16-bits.</td>
</tr>
<tr>
<td></td>
<td>This field has an <em>IMPLEMENTATION DEFINED</em> value.</td>
</tr>
<tr>
<td></td>
<td>Access to this field is RO.</td>
</tr>
<tr>
<td>3</td>
<td>JEDEC, Indicates a JEP106 identity code is used.</td>
</tr>
<tr>
<td></td>
<td>Reads as 0b1.</td>
</tr>
<tr>
<td></td>
<td>Access to this field is RO.</td>
</tr>
<tr>
<td>2:0</td>
<td>DES_1, Designer, most significant bits of JEP106 ID code. For Arm Limited, this field is 0b01.</td>
</tr>
<tr>
<td></td>
<td>This field has an <em>IMPLEMENTATION DEFINED</em> value.</td>
</tr>
<tr>
<td></td>
<td>Access to this field is RO.</td>
</tr>
</tbody>
</table>

**Accessing the CTIPIDR2:**

CTIPIDR2 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFE8</td>
<td>CTIPIDR2</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.33  CTIPIDR3, CTI Peripheral Identification Register 3

The CTIPIDR3 characteristics are:

**Purpose**

Provides information to identify a CTI component.

For more information, see *About the Peripheral identification scheme on page K6-11696*.

**Configurations**

CTIPIDR3 is in the Debug power domain.

Implementation of this register is **OPTIONAL**.

This register is required for CoreSight compliance.

**Attributes**

CTIPIDR3 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REVAND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMOD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**REVAND, bits [7:4]**

Part minor revision. Parts using CTIPIDR2.REVISION as an extension to the Part number must use this field as a major revision number.

This field has an **IMPLEMENTATION DEFINED** value.

Access to this field is RO.

**CMOD, bits [3:0]**

Customer modified. Indicates someone other than the Designer has modified the component.

This field has an **IMPLEMENTATION DEFINED** value.

Access to this field is RO.

**Accessing the CTIPIDR3:**

CTIPIDR3 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFE</td>
<td>CTIPIDR3</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.34   CTIPIDR4, CTI Peripheral Identification Register 4

The CTIPIDR4 characteristics are:

**Purpose**

Provides information to identify a CTI component.

For more information, see *About the Peripheral identification scheme* on page K6-11696.

**Configurations**

CTIPIDR4 is in the Debug power domain.

Implementation of this register is OPTIONAL.

This register is required for CoreSight compliance.

**Attributes**

CTIPIDR4 is a 32-bit register.

**Field descriptions**

![Field Descriptions Diagram](image)

**Bits [31:8]**

Reserved, RES0.

**SIZE, bits [7:4]**

Size of the component. $\log_2$ of the number of 4KB pages from the start of the component to the end of the component ID registers.

Reads as 0b0000.

Access to this field is RO.

**DES_2, bits [3:0]**

Designer, JEP106 continuation code, least significant nibble. For Arm Limited, this field is 0b0100.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Accessing the CTIPIDR4:**

CTIPIDR4 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0xFD0</td>
<td>CTIPIDR4</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.35   CTITRIGINSTATUS, CTI Trigger In Status register

The CTITRIGINSTATUS characteristics are:

**Purpose**

Provides the status of the trigger inputs.

**Configurations**

CTITRIGINSTATUS is in the Debug power domain.

**Attributes**

CTITRIGINSTATUS is a 32-bit register.

**Field descriptions**

TRIN<\(n\)>, bit [\(n\)], for \(n = 31\) to 0

Trigger input <\(n\)> status.

Bits [31:N] are RAZ. \(N\) is the number of CTI triggers implemented as defined by the CTIDEVID.NUMTRIG field.

0b0 Input trigger \(n\) is inactive.

0b1 Input trigger \(n\) is active.

Not implemented and not-connected input triggers are always inactive.

It is IMPLEMENTATION DEFINED whether an input trigger that does not support multicycle events can be observed as active.

**Accessing the CTITRIGINSTATUS:**

CTITRIGINSTATUS can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0x130</td>
<td>CTITRIGINSTATUS</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
H9.4.36  CTITRIGOUTSTATUS, CTI Trigger Out Status register

The CTITRIGOUTSTATUS characteristics are:

**Purpose**

Provides the raw status of the trigger outputs, after processing by any IMPLEMENTATION DEFINED trigger interface logic. For output triggers that are self-acknowledging, this is only meaningful if the CTI implements multicyle channel events.

**Configurations**

CTITRIGOUTSTATUS is in the Debug power domain.

**Attributes**

CTITRIGOUTSTATUS is a 32-bit register.

**Field descriptions**

TROUT<n>, bit [n], for n = 31 to 0

Trigger output <n> status.

Bits [31:N] are RAZ. N is the value in CTIDEVID.NUMTRIG.

If n < N, and output trigger <n> is implemented and connected, and either the trigger is not self-acknowledging or the CTI implements multicyle channel events, then permitted values for TROUT<n> are:

0b0  Output trigger n is inactive.
0b1  Output trigger n is active.

Otherwise when n < N it is IMPLEMENTATION DEFINED whether TROUT<n> behaves as described here or is RAZ.

**Accessing the CTITRIGOUTSTATUS:**

CTITRIGOUTSTATUS can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTI</td>
<td>0x134</td>
<td>CTITRIGOUTSTATUS</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
Part I

Memory-mapped Components of the Arm Architecture
Chapter I1
Requirements for Memory-mapped Components

This chapter provides some additional information about memory-mapped components. It contains the following sections:

- Supported access sizes on page I1-10716.
- Synchronization of memory-mapped registers on page I1-10718.
- Access requirements for reserved and unallocated registers on page I1-10720.
I1.1 Supported access sizes

The information in this section applies to all accesses to memory-mapped components of the Armv8 and later architectures, unless a register or component description explicitly states otherwise.

The memory access sizes that are supported by any peripheral are IMPLEMENTATION DEFINED by the peripheral.

When `HaveSecureExtDebugView() == TRUE`, each debug component has a Secure and Non-secure view. The Secure view of a debug component is mapped into Secure physical memory and the Non-secure view of a debug component is mapped into Non-secure memory. Apart from access conditions, the Non-secure and Secure views of the debug components are identical.

An implementation of a memory-mapped component that is compatible with the Armv8 and later architectures must support the following:

- Word-aligned 32-bit accesses to access 32-bit registers.
- If the system includes any component with direct memory access to the memory-mapped component which needs to access the component but does not support making 64-bit accesses, word-aligned 32-bit accesses to either half of a 64-bit register that is mapped to a doubleword-aligned pair of adjacent 32-bit locations. This includes, but is not limited to:
  - A PE that supports AArch32 at any Exception level.
  - A PE that implements a 32-bit ISA that does not include 64-bit memory operations. For example, Armv8-M T32.
  - A Debug Access Port that cannot make 64-bit accesses.

Arm deprecates support for 32-bit accesses to either half of 64-bit registers.

**Note**

Although AArch32 implementations might make 64-bit accesses for `LDP` and `STP` instructions, this is not architecturally required. To guarantee ordering of accesses, portable AArch32 software should use `LDR` and `STR`. For compatibility with such software, the memory-mapped component should treat the PE that supports AArch32 as not making 64-bit accesses using portable code.

**Note**

Some memory-mapped components of the Arm architecture require support for word-aligned 32-bit accesses to either half of a 64-bit memory mapped register even if all components with direct memory access to the memory-mapped component support making 64-bit accesses. These include:

- The memory-mapped interface to the external debug and CTI registers that are described in Chapter H9 External Debug Register Descriptions.
- The memory-mapped interfaces to the Generic Timer registers that are described in Chapter I2 System Level Implementation of the Generic Timer.
- When the 64-bit external PMU programmers’ model extension is not implemented, the memory-mapped interfaces to the Performance Monitors registers that are described in Chapter I3 Recommended External Interface to the Performance Monitors.
- The memory-mapped interfaces to the Activity Monitors registers that are described in Chapter I4 Recommended External Interface to the Activity Monitors.

- Doubleword-aligned 64-bit accesses to access 64-bit registers that are mapped to a doubleword-aligned pair of adjacent 32-bit locations.

All registers are only single-copy atomic at word granularity. This means that for 64-bit accesses to a 64-bit register, the system might generate a pair of 32-bit accesses. The order in which the two halves are accessed is not specified.

The following accesses are not supported:

- Byte accesses.
- Halfword accesses.
- Unaligned word accesses. These accesses are not word single-copy atomic.
- Unaligned doubleword accesses. These accesses are not doubleword single-copy atomic.
- Doubleword accesses to a pair of 32-bit locations that are not a doubleword-aligned pair that forms a 64-bit register.
• Quadword accesses or higher accesses.
• Exclusive accesses.

For unsupported accesses, it is **CONSTRAINED UNPREDICTABLE** whether:
• The access generates an External abort or not.
• The defined side-effects of a read occur or not. A read returns **UNKNOWN** values.
• A write is ignored or sets the accessed register or registers to **UNKNOWN**.
• The access generates a fault handling interrupt or not. A read returns **UNKNOWN** data.

For memory-mapped accesses from a PE that complies with an Arm architecture, the single-copy atomicity rules for the instruction, the type of instruction, and the type of memory that is accessed, determine the size of the access that is made by an instruction. **Example 11-1** shows this.

---

**Example 11-1 Access sizes for memory-mapped accesses**

Two Load Doubleword instructions that are made to consecutive doubleword-aligned locations generate a pair of single-copy atomic doubleword reads. However, if the accesses are made to Normal memory or Device-GRE memory they might appear as a single quadword access that is not supported by the peripheral.

---

The Arm architecture does not require the size of each element that is accessed by a multi-register load or store instruction to be identifiable by the memory system beyond the PE. Unless otherwise specified by the component, any access to a memory-mapped component of the Arm architecture is defined to be beyond the PE.

Software must use a Device-nGRE or stronger memory type, and only single register load and store instructions, to create memory accesses that are supported by the peripheral. For more information, see **Memory types and attributes on page B2-196.**
11.2 Synchronization of memory-mapped registers

This section describes the synchronization requirements for the memory-mapped accesses to System registers.

This section refers to accesses to external system control registers as external reads and external writes. It refers to accesses to System registers as direct reads, direct writes, indirect reads, and indirect writes.

Note

Synchronization requirements for AArch64 System registers on page D17-5547 and Synchronization of changes to AArch32 System registers on page G8-9326 define direct read, direct write, indirect read, and indirect write, and classifies external reads as indirect reads and external writes as indirect writes.

Writes to the same register are serialized, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes. Unless otherwise stated, external writes to different registers are not necessarily observed in the same order by all observers as the order in which they complete.

Explicit synchronization is not required for an external read or an external write by an external agent to be observable to a following external read or external write by that agent to the same register using the same address, and so is never required for registers that are accessible as external system control registers.

Unless required to be observable to all observers in finite time, without explicit synchronization, explicit synchronization is normally required following an external write to any register for that write to be observable by:

- A direct access.
- An indirect read by an instruction.
- An external read of the register using a different address.

This means that an external write by an external agent is guaranteed to have an effect on subsequent instructions executed by the PE only if all of the following are true:

- The write has completed.
- The PE has executed a Context synchronization event.
- The Context synchronization event was executed after the write completed.

The order and synchronization of direct reads and direct writes of System registers is defined by:

- Synchronization requirements for AArch64 System registers on page D17-5547.
- Synchronization of changes to AArch32 System registers on page G8-9326.

The external agent must be able to guarantee completion of a write. For example, the agent can:

- Mark the memory as Device-nGnRnE and executing a DSB barrier, if the system supports this property.
- If the register is read/write and reads are not destructive, read back the value written.
- Use some guaranteed property of the connection between the PE and the external agent.

The external agent and PE can guarantee ordering by, for example, passing messages in an ordered way with respect to the external write and the Context synchronization event, and relying on the memory ordering rules provided by the memory model.

External reads and external write complete in the order in which they arrive at the PE. For accesses to different register locations, the external agent must create this order. The agent can:

- Mark the memory as Device-nGnRnE or Device-nGnRE.
- Use the appropriate memory barriers.
- Rely on some guaranteed property of the connection between the PE and the external agent.

However, the external agent cannot force the synchronization of completed writes.

In a simple sequential execution, an indirect write that occurs as a side-effect of an access happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect.

Without explicit synchronization to guarantee the order of the accesses, where the same register is accessed by two or more of a System register access instruction, and external agent, and autonomous asynchronous event, or as a result of a memory-mapped access, the behavior must be as if the accesses occurred atomically and in any order. This applies even if the accesses occur simultaneously.
For example, some registers have the property that for certain bits a write of 0 is ignored and a write of 1 has an effect. This means the simultaneous writes must be merged.
I1.3 Access requirements for reserved and unallocated registers

This section describes the access requirements for reserved and unallocated memory-mapped components.

The following information relates to certain types of reserved accesses:

- Reads and writes of unallocated locations. These accesses are reserved for the architecture.
- Reads and writes of locations for features that are not implemented, including:
  - OPTIONAL features that are not implemented.
  - Breakpoints and watchpoints that are not implemented.
  - Performance Monitors counters that are not implemented.
  - CTI triggers that are not implemented.
  - Error records that are not implemented.
  These accesses are reserved.
- Reads of WO locations. These accesses are reserved for the architecture.
- Writes to RO locations. These accesses are reserved for the architecture.

Reserved accesses are normally RAZ/WI. However, software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture. Software must treat reserved accesses as RES0.
Chapter I2
System Level Implementation of the Generic Timer

This chapter defines the system level implementation of the Generic Timer. It contains the following sections:

- About the Generic Timer specification on page I2-10722.
- Memory-mapped counter module on page I2-10724.
- Memory-mapped timer components on page I2-10728.

Note
- Generic Timer memory-mapped register descriptions on page I5-10875 describes the System level Generic Timer registers. These registers are memory-mapped.

- Appendix K Additional Information for Implementations of the Generic Timer gives additional information, that does not form part of the architectural definition of a system level implementation of the Generic Timer.

- Chapter D10 The Generic Timer in AArch64 state gives a general description of the AArch64 state view of the Generic Timer, and describes the AArch64 System register interface to the Generic Timer.

- Chapter G6 The Generic Timer in AArch32 state gives a general description of the AArch32 state view of the Generic Timer, and describes the AArch32 System register interface to the Generic Timer.
I2.1 About the Generic Timer specification

Chapter D10 The Generic Timer in AArch64 state describes the Arm Generic Timer and its implementation as seen from AArch64 state. Chapter G6 The Generic Timer in AArch32 state describes the Arm Generic Timer and its implementation as seen from AArch32 state. These chapters include the definition of the low-latency System register interface to the Generic Timer. However, the Arm Generic Timer architecture also defines a memory-mapped component, that comprises:

- A memory-mapped counter module, that controls the generation of the Count value used by the Generic Timer.
  
  This memory-mapped counter module is required in any Arm Generic Timer implementation that requires software control of the Count value of the Generic Timer.

- Optional memory-mapped timer modules. These give a standardized way of providing timers for programmable system components other than PEs that implement the Arm architecture.

The full set of Generic Timer components on page D10-5233 summarizes these components as seen from AArch64 state, and The full set of Generic Timer components on page G6-9289 summarizes them as seen from AArch32 state. The system level components of the Generic Timer on page I2-10723 summarizes the system level components.

I2.1.1 Registers in the system level implementation of the Generic Timer

Registers that control components of the system level implementation of the Generic Timer are grouped into frames. This specification defines the registers in each frame, and their offsets within the frame. The system defines the position of each frame in the memory map. This means the base addresses for each frame are IMPLEMENTATION DEFINED.

--- Note ---

The final 12 words of the first or only 4KB block of a register memory frame is an ID block.

--- Note ---

Each frame must be in its own memory page, or memory protection region, and must be aligned to the size of the translation granule or protection granule.

--- Note ---

When a system level implementation of the Generic Timer is accessed by a PE:

- Using a VMSA, each frame is in its own memory page, aligned to the size of the translation granule.

- Using a PMSA, each frame is in its own memory protection region, aligned to the size of the memory protection granule.

---

The following sections give more information about the requirements for the system level Generic Timer component:

- Endianness and supported access sizes.

- Power and reset domains for the system level implementation of the Generic Timer on page I2-10723.

Endianness and supported access sizes

All memory-mapped peripherals defined in the Arm architecture must be little-endian. This means the system-level Generic Timer registers, and the register frames, are little-endian.

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the memory-mapped Generic Timer registers implementations must:

- Comply with the requirements of Supported access sizes on page 11-10716.

- Support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register mapped to a doubleword-aligned pair of adjacent 32-bit locations, even if all components with direct memory access to the Generic Timer support making 64-bit accesses.
Power and reset domains for the system level implementation of the Generic Timer

The power and reset domains of the system level implementation of the Generic Timer are IMPLEMENTATION DEFINED as part of the system implementation. In register descriptions, they are called Timer resets to indicate they can be outside the PE power and reset domains defined by the remainder of this manual.

The Arm architecture requires that the CNTCR.{FCREQ, EN} and CNTSR.FCACK fields reset to 0. These Timer reset values apply only on powerup of the power domain in which the registers are implemented or a reset of the reset domain in which they are implemented.

Every other register, or register field, of a system level implementation of the Generic Timer resets to a value that is architecturally UNKNOWN if it has a meaningful reset value. These Timer resets apply on powerup of the power domain in which the register is implemented, and on a reset of the reset domain in which it is implemented.

I2.1.2 The system level components of the Generic Timer

Each system level component has one or two register frames. The possible system level components are:

The memory-mapped counter module, required

This module controls the system counter. It has two frames:

- A control frame, CNTControlBase.
- A status frame, CNTReadBase.

Memory-mapped counter module on page I2-10724 describes this component.

The memory-mapped timer control module, required

The system level implementation of the Generic Timer can provide up to eight timers, and the memory-mapped timer control module identifies:

- Which timers are implemented.
- The features of each implemented timer.

This module has a single frame, CNTCTLBase.

The CNTCTLBase frame on page I2-10728 describes this frame.

Memory-mapped timers, optional

An implemented memory-mapped timer:

- Must provide a privileged view of the timer, in the CNTBaseN frame.
- Optionally provides an unprivileged view of the timer in the CNTEL0BaseN frame.

N is the timer number, and the corresponding frame number, in the range 0-7.

The CNTBaseN and CNTEL0BaseN frames on page I2-10729 describes these frames.
I2.2 Memory-mapped counter module

The memory-mapped counter module provides top-level control of the system counter. The CNTControlBase frame holds the registers for the memory-mapped counter, and provides:

- An RW control register, CNTCR, that provides:
  - An enable bit for the system counter.
  - An enable bit for Halt-on-debug. For more information, see Halt-on-debug on page I2-10726.
  - A field that can be written to request a change to the update frequency of the system counter, with a corresponding change to the increment made at each update. This mechanism means that, for example, if the update frequency is halved, the increment at each update is doubled.
    For more information, see Control of counter operating frequency and increment on page I2-10725.

Writes to this register are rare. In a system that supports two Security states, this register is writable only by Secure writes.

- A RO status register, CNTSR, that provides:
  - A bit that indicates whether the system counter is halted because of an asserted Halt-on-debug signal.
  - A field that indicates the current update frequency of the system counter. This field can be polled to determine when a requested change to the update frequency has been made.

- Two contiguous 32-bit RW registers that hold the current system counter value, CNTCV. If the system supports 64-bit atomic accesses, these two registers must be accessible by such accesses.

The system counter must be disabled before writing to these registers, otherwise the effect of the write is UNPREDICTABLE.

Writes to these registers are rare. In a system that supports two Security states, these registers are writable only by Secure writes.

- A Frequency modes table of one or more 32-bit entries, where:
  - The first entry in the table defines the base frequency of the system counter. This is the maximum frequency at which the counter updates.
  - Each subsequent entry in the table defines an alternative frequency of the system counter, that must be an exact divisor of the base frequency.

A 32-bit zero entry immediately follows the last table entry.

This table can be RO or RW. For more information, see The Frequency modes table on page I2-10725.

In addition, the CNTReadBase frame includes a read-only copy of the system counter value, CNTCV, as two contiguous 32-bit RO registers. If the system supports 64-bit atomic accesses, these two registers must be accessible by such accesses.

Counter module control and status register summary on page I2-10726 describes CNTReadBase and CNTControlBase memory maps, and the registers in each frame.
I2.2.1 Control of counter operating frequency and increment

The system counter has a fixed base frequency, and must maintain the required counter accuracy, meaning Arm recommends that it does not gain or lose more than ten seconds in a 24-hour period, see The system counter on page D10-5234. However, the counter can increment at a lower frequency than the base frequency, using a correspondingly larger increment. For example, it can increment by four at a quarter of the base frequency. Any lower-frequency operation, and any switching between operating frequencies, must not reduce the accuracy of the counter.

Control of the system counter frequency and increment is provided only through the memory-mapped counter module. The following sections describe this control:

- The Frequency modes table.
- Changing the system counter and increment.

The Frequency modes table

The Frequency modes table starts at offset 0x20 in the CNTControlBase frame.

Table entries are 32-bits, and each entry specifies a system counter update frequency, in Hz.

The first entry in the table specifies the base frequency of the system counter.

When the system counter is operating at a lower frequency than the base frequency, the increment applied at each counter update is given by:

\[ \text{increment} = \frac{\text{base_frequency}}{\text{selected_frequency}} \]

A 32-bit word of zero value marks the end of the table. That is, the word of memory immediately after the last entry in the table must be zero.

The only required entry in the table is the entry for the base frequency.

Typically, the Frequency modes table is in RO memory. However, a system implementation might use RW memory for the table, and initialize the table entries as part of its startup sequence. Therefore, the CNTControlBase memory map shows the table region as RO or RW.

Arm strongly recommends that the Frequency modes table is not updated once the system is running.

The architecture can support up to 1004 entries in the Frequency modes table, including the zero-word end marker, and the number of entries is IMPLEMENTATION DEFINED, up to this limit.

Note

- Arm considers it likely that implementations will require significantly fewer entries than the architectural limit.
- In the CNTControlBase frame, the offset range 0x0C0-0x0FC can be used for IMPLEMENTATION DEFINED registers. If any registers are defined in this space, then the Frequency modes table cannot extend beyond offset 0x088, with a zero word at offset 0x08C. This means that if any IMPLEMENTATION DEFINED registers are defined the maximum number of entries in the table is 40, including the zero-word end marker.

Changing the system counter and increment

The value of the CNTCR.FCREQ field specifies which entry in the Frequency modes table specifies the system counter update frequency.

Changing the value of CNTCR.FCREQ requests a change to the system counter update frequency. To ensure the frequency change does not affect the overall accuracy of the counter, a change is made as follows:

- When changing from a higher frequency to a lower frequency, the counter:
  1. Continues running at the higher frequency until the count reaches an integer multiple of the required lower frequency.
  2. Switches to operating at the lower frequency.
• When changing from a lower frequency to a higher frequency, the counter:
  1. Waits until the end of the current lower-frequency cycle.
  2. Makes the counter increment required for operation at that lower frequency.
  3. Switches to operating at the higher frequency.

When the frequency has changed, CNTSR is updated to indicate the new frequency. Therefore, a system component
that is waiting for a frequency change can poll CNTSR to detect the change.

### I2.2.2 Halt-on-debug

The CNTCR register provides an enable bit for an OPTIONAL Halt-on-debug signal.

When the CNTCR.HDBG bit is set to 1, and the Halt-on-debug signal is implemented and asserted, the system
counter is halted. Otherwise, the system counter ignores the state of this signal.

Arm recommends that a system counter implements a Halt-on-debug signal that can be controlled by a debugger
using the Embedded Cross-Trigger (ECT) using a system-level cross-trigger interface that includes:
• A debug request output trigger event that asserts the Halt-on-debug signal.
• A restart request output trigger event that deasserts the Halt-on-debug signal.

For more information, see About the Embedded Cross-Trigger on page H5-10318.

--- Note ---

Software must use the Halt-on-debug enable bit to ensure that the timers cannot be halted maliciously in an attempt
to prohibit progress.

---

For more information about Halt-on-debug, contact Arm.

### I2.2.3 Counter module control and status register summary

The Counter module control and status registers are memory-mapped registers in the following register memory
frames:
• A control frame, with base address CNTControlBase.
• A status frame, with base address CNTReadBase.

Each of these register memory frames is in its own memory page or memory protection region, and the frame base
address points to the start of this region. Each base address must be aligned to the size of the translation granule or
 protection granule.

--- Note ---

Each frame of a memory-mapped Generic Timer takes the name of its base address.

---

In each register memory frame, the memory at offset 0xFD0-0xFFF is reserved for twelve 32-bit IMPLEMENTATION
DEFINED ID registers, see the CounterID<n> register descriptions for more information.

--- Note ---

The Arm architecture requires memory-mapped peripherals to be little-endian, and therefore the counter is
little-endian.

---

In an implementation that supports Secure and Non-secure memory maps, CNTControlBase is accessible only by
Secure accesses.

Table I2-1 on page I2-10727 shows the CNTControlBase control registers, in order of their offsets from the
CNTControlBase base address, for an implementation that includes registers in the IMPLEMENTATION DEFINED
register space 0x0C0-0x0FC, and also has fewer than 39 CNTFID<n> registers. The Frequency modes table on
page I2-10725 describes how this memory map differs if more CNTFID<n> registers are implemented.
Generic Timer memory-mapped register descriptions on page I5-10875 describes each of these registers.

### Table I2-1 CNTControlBase memory map

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>CNTCR</td>
<td>RW</td>
<td>Counter Control Register.</td>
</tr>
<tr>
<td>0x004</td>
<td>CNTSR</td>
<td>RO</td>
<td>Counter Status Register.</td>
</tr>
<tr>
<td>0x008</td>
<td>CNTCV[31:0]</td>
<td>RW</td>
<td>Counter Count Value register.</td>
</tr>
<tr>
<td>0x00C</td>
<td>CNTCV[63:32]</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>0x010</td>
<td>CNTSCRa</td>
<td>RW</td>
<td>Counter Scale Register.</td>
</tr>
<tr>
<td>0x014-0x018</td>
<td>RES0</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>0x01C</td>
<td>CNTIDa</td>
<td>RO</td>
<td>Counter Identification Register.</td>
</tr>
<tr>
<td>0x020</td>
<td>CNTFID0</td>
<td>RO or RW</td>
<td>Frequency modes table, and end marker. For more information, see The Frequency modes table on page I2-10725.</td>
</tr>
<tr>
<td>0x020+4n</td>
<td>CNTFID&lt;n&gt;</td>
<td>RO or RW</td>
<td></td>
</tr>
<tr>
<td>0x024+4n</td>
<td>-</td>
<td>RO or RW, RAZ</td>
<td></td>
</tr>
<tr>
<td>(0x028+4n)-0x08C</td>
<td>-</td>
<td>RO, RES0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0x080-0x0FC</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED</td>
<td>Reserved for IMPLEMENTATION DEFINED registers.</td>
</tr>
<tr>
<td>0x100-0xFFC</td>
<td>-</td>
<td>RO, RES0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0xF00-0xFFF</td>
<td>CounterID&lt;n&gt;</td>
<td>RO</td>
<td>Counter ID registers 0-11.</td>
</tr>
</tbody>
</table>

a. Implemented only if FEAT_CNTSC is implemented.

Table I2-2 on page I2-10727 shows the CNTReadBase control registers, in order of their offsets from the CNTReadBase base address. Generic Timer memory-mapped register descriptions on page I5-10875 describes each of these registers.

### Table I2-2 CNTReadBase memory map

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>CNTCV[31:0]</td>
<td>RO</td>
<td>Counter Count Value register.</td>
</tr>
<tr>
<td>0x004</td>
<td>CNTCV[63:32]</td>
<td>RO</td>
<td></td>
</tr>
<tr>
<td>0x008</td>
<td>-</td>
<td>RES0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0xF00-0xFFF</td>
<td>CounterID&lt;n&gt;</td>
<td>RO</td>
<td>Counter ID registers 0-11.</td>
</tr>
</tbody>
</table>
I2.3 Memory-mapped timer components

This part of the Arm Generic Timer specification defines an optional memory-mapped timer component. This can be implemented as part of any programmable system component that does not incorporate a System register mapped Arm Generic Timer, to provide that system component with the timer functionality of an Arm Generic Timer.

The memory map consists of up to eight timer frames. The base address of a frame is CNTBaseN, where \( N \) numbers from 0 up to a maximum permitted value of 7.

Each CNTBaseN timer frame:
- Provides its own set of timers and associated interrupts.
- Is implemented in its own memory page or memory protection region.
- Is implemented at a base address, identified as CNTBaseN, that is aligned to the size of the translation granule or memory protection region.

For each implemented CNTBaseN frame the system can optionally provide an unprivileged view of the frame, described as the EL0 view of the frame. The base address of this second view of the CNTBaseN frame is CNTEL0BaseN.

--- Note ---

In the naming of the registers associated with a CNTBaseN or CNTEL0BaseN frame, the value of \( N \) is represented as \(<n>\), for example CNTACR\(<n>\).

---

If a CNTEL0BaseN frame is implemented:
- Is implemented in its own memory page or memory protection region and is aligned to the size of the translation granule or memory protection region.
- All registers visible in CNTBaseN, except for CNTVOFF and CNTEL0ACR, can be visible in CNTEL0BaseN.
  - Control fields in CNTEL0ACR determine whether each register is visible.
- The offsets of all visible registers are the same as their offsets in the CNTBaseN frame.

In addition to the implemented CNTBaseN and CNTEL0BaseN frames, the system must provide a single control frame at base address CNTCTLBase. CNTCTLBase must be implemented in its own memory page or memory protection region and is aligned to the size of the translation granule or memory protection region.

The system defines the position of each frame in the memory map. This means the values of each of the CNTBaseN, CNTEL0BaseN, and CNTCTLBase base addresses is IMPLEMENTATION DEFINED.

--- Note ---

The Arm architecture requires memory-mapped peripherals to be little-endian, and therefore the memory-mapped timers are little-endian.

The following sections describe the implementation of a memory-mapped view of the counter and timer:
- The CNTCTLBase frame.
- The CNTBaseN and CNTEL0BaseN frames on page I2-10729.

--- Note ---

Providing a complete set of features in a system level implementation on page K7-11702 gives an implementation example for a system level Generic Timer implementation that provides equivalent features to a System registers Generic Timer implementation in a PE that includes all of the Exception levels.

I2.3.1 The CNTCTLBase frame

The CNTCTLBase frame contains:
- An identification register for the features of the memory-mapped counter and timer implementation.
- Access controls for each CNTBaseN frame.
- A virtual offset register for frames that implement a virtual timer.
Table I2-3 on page I2-10729 shows the CNTCTLBase registers, in order of their offsets from the CNTCTLBase base address.

--- Note ---
CNTFRQ and CNTVOFF registers are also implemented in a System register interface to the Generic Timer.

---

*Generic Timer memory-mapped register descriptions on page I2-10875* describes each of these registers.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Type</th>
<th>Security</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>CNTFRQb</td>
<td>RW</td>
<td>Secure</td>
<td>Counter Frequency register.</td>
</tr>
<tr>
<td>0x004</td>
<td>CNTNSAR</td>
<td>RW</td>
<td>Secure</td>
<td>Counter Non-Secure Access register.</td>
</tr>
<tr>
<td>0x008</td>
<td>CNTTIDR</td>
<td>RO</td>
<td>Both</td>
<td>Counter Timer ID register.</td>
</tr>
<tr>
<td>0x00c-0x03f</td>
<td>-</td>
<td>RES0</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0x040+4Nc</td>
<td>CNTACR&lt;n&gt;</td>
<td>RW</td>
<td>Configurable</td>
<td>Counter Access Control register N.</td>
</tr>
<tr>
<td>0x060-0x07f</td>
<td>-</td>
<td>RES0</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0x080+8Nc</td>
<td>CNTVOFF&lt;n&gt;[31:0]</td>
<td>RW</td>
<td>Configurable</td>
<td>Virtual Offset register N. If the CNTBaseN frame has virtual timer capability then CNTVOFF is implemented as an RW register, otherwise its location is RAZ/WI.</td>
</tr>
<tr>
<td>0x084+8Nc</td>
<td>CNTVOFF&lt;n&gt;[63:32]</td>
<td>RW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0c0-0x0fc</td>
<td>-</td>
<td>RES0</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0x100-0x7fc</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0x800-0xffc</td>
<td>-</td>
<td>RES0</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0xfc0-0xffc</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0xfd0-0xffc</td>
<td>CounterID&lt;n&gt;</td>
<td>RO</td>
<td>Both</td>
<td>Counter ID registers 0-11.</td>
</tr>
</tbody>
</table>

a. Access security requirement in an implementation that supports two Security states. In an implementation that does not support multiple Security states all registers are accessible as shown in the *Type* column.
b. These registers are also defined in the System register interface to the Generic Timer, and therefore are also described in *Generic Timer registers on page D17-7014* and *Generic Timer registers on page G8-10144*. The bit assignments of the registers are identical in the System register interface and in the memory-mapped system level interface.
c. Implemented for each value of N from 0 to 7 for which a CNTBaseN frame is implemented.
d. The CNTNSAR determines the Non-secure accessibility of the CNTACR<n>s and the CNTVOFF<n> in the CNTCTLBase frame. For more information, see the register descriptions.
e. Address is reserved, RAZ/WI if register not implemented.

All implementations of the Generic Timer include the virtual counter. Therefore, conceptually, all implementations include the CNTVOFF register that defines the virtual offset between the physical count and the virtual count. If a memory-mapped Generic Timer component does not distinguish between real time and virtual time, then it can implement CNTVOFF as RAZ/WI. Otherwise CNTVOFF is an RW register, and Arm strongly recommends that the system only permits access to CNTVOFF from EL2 or higher.

### I2.3.2 The CNTBaseN and CNTEL0BaseN frames

Each CNTBaseN frame, or {CNTBaseN, CNTEL0BaseN} pair of frames, provides a memory-mapped counter and timer, see:
- *The CNTBaseN frame on page I2-10730.*
- *The CNTEL0BaseN frame on page I2-10730.*
- *CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-10731.*
The CNTBaseN frame

Table I2-4 on page I2-10730 shows the CNTBaseN registers, in order of their offsets from the CNTBaseN base address. Whether a frame includes a virtual timer is IMPLEMENTATION DEFINED. If it does not, then memory at offsets 0x030-0x03C is RAZ/WI. Except for CNTEL0ACR and the CounterID<n> registers, equivalent registers are also implemented in a System register interface to the timer component of a Generic Timer.

Generic Timer memory-mapped register descriptions on page I5-10875 describes each of these registers.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>CNTPCT[31:0]</td>
<td>RO</td>
<td>Physical Count register.</td>
</tr>
<tr>
<td>0x04</td>
<td>CNTPCT[63:32]</td>
<td>RO</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td>CNTVCT[31:0]</td>
<td>RO</td>
<td>Virtual Count register.</td>
</tr>
<tr>
<td>0x0C</td>
<td>CNTVCT[63:32]</td>
<td>RO</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>CNTFRQ</td>
<td>RO</td>
<td>Counter Frequency register.</td>
</tr>
<tr>
<td>0x14</td>
<td>CNTEL0ACR</td>
<td>RW</td>
<td>Counter EL0 Access Control Register, optional in the CNTBaseN memory map.</td>
</tr>
<tr>
<td>0x18</td>
<td>CNTVOFF[31:0]</td>
<td>RO</td>
<td>Virtual Offset register. If CNTVOFF in the CNTCTLBase frame is an RW register, a read of this register returns the value of that register. Otherwise is RAZ.</td>
</tr>
<tr>
<td>0x1C</td>
<td>CNTVOFF[63:32]</td>
<td>RO</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td>CNTP_CVAL[31:0]</td>
<td>RW</td>
<td>Physical Timer CompareValue register.</td>
</tr>
<tr>
<td>0x24</td>
<td>CNTP_CVAL[63:32]</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>0x28</td>
<td>CNTP_TVAl</td>
<td>RW</td>
<td>Physical TimerValue register.</td>
</tr>
<tr>
<td>0x2C</td>
<td>CNTP_CTL</td>
<td>RW</td>
<td>Physical Timer Control register.</td>
</tr>
<tr>
<td>0x30</td>
<td>CNTV_CVAL[31:0]</td>
<td>RW</td>
<td>Virtual Timer CompareValue register, optional in the CNTBaseN memory map.</td>
</tr>
<tr>
<td>0x34</td>
<td>CNTV_CVAL[63:32]</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>0x38</td>
<td>CNTV_TVAl</td>
<td>RW</td>
<td>Virtual TimerValue register, optional in the CNTBaseN memory map.</td>
</tr>
<tr>
<td>0x3C</td>
<td>CNTV_CTL</td>
<td>RW</td>
<td>Virtual Timer Control register, optional in the CNTBaseN memory map.</td>
</tr>
<tr>
<td>0x40-0xCF</td>
<td>-</td>
<td>RES0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0xFD0-0xFFC</td>
<td>CounterID&lt;n&gt;</td>
<td>RO</td>
<td>Counter ID registers 0-11.</td>
</tr>
</tbody>
</table>

a. These registers are also defined in the System register interface to the Generic Timer, and therefore are also described in Generic Timer registers on page D17-7014 and Generic Timer registers on page G8-10144. The bit assignments of the registers are identical in the System register interface and in the memory-mapped system level interface.
b. Address is reserved, RAZ/WI if register not implemented.
c. The CNTCTLBase frame includes an RW view of this register.

The CNTEL0BaseN frame

For any value of N, the layout of the registers in the CNTEL0BaseN frame is identical to the CNTBaseN frame, except that, in the CNTEL0BaseN frame:

- CNTVOFF is never visible, and the memory at 0x018-0x01C is RAZ/WI.
- CNTEL0ACR is never visible, and the memory at 0x14 is RAZ/WI.
• If implemented in the CNTBaseN frame, CNTEL0ACR controls whether CNTPCT, CNTVCT, CNTFRQ, the Physical Timer, and the Virtual Timer registers are visible in the CNTEL0BaseN frame. If CNTEL0ACR is not implemented then these registers are not visible in the CNTEL0BaseN frame, and their addresses in that frame are RAZ/WI.

If an implementation supports 64-bit atomic accesses, then CNTPCT, CNTVCT, CNTVOFF, CNTP_CVAL, and CNTV_CVAL must be accessible as atomic 64-bit values.

CNTCTRLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames

In the CNTCTRLBase frame:

CNTTIDR controls:
- Whether each CNTBaseN frame is implemented.
- If a CNTBaseN frame is implemented, whether:
  - That CNTBaseN frame has virtual timer capability.
  - A corresponding CNTEL0BaseN frame is implemented.

CNTNSAR controls:
In an implementation that recognizes two Security states, determines whether each implemented CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

This control also determines whether, in the CNTCTRLBase frame, the CNTACR<n> and CNTVOFF<n> registers are accessible by Non-secure accesses.

The CNTACR<n> registers control:
For each implemented CNTBaseN frame, the accessibility of the following registers in that frame:
- CNTP_CTL, CNTP_CVAL, and CNTP_TVAL.
- CNTV_CTL, CNTV_CVAL, and CNTV_TVAL.
- CNTVOFF.
- CNTFRQ.
- CNTPCT.
- CNTVCT.

For CNTACR<n>, the value of <n> corresponds to the value of N for the controlled CNTBaseN frame.

The CNTVOFF<n> registers provide:
For each implemented CNTBaseN frame that has virtual capability, the RW copy of the CNTVOFF register for that frame.

——— Note ————
In a CNTBaseN frame that has virtual timer capability the CNTVOFF register is RO.

———
For CNTVOFF<n>, the value of <n> corresponds to the value of N for the controlled CNTBaseN frame.
Chapter I3
Recommended External Interface to the Performance Monitors

This chapter describes the recommended external interface to the Performance Monitors. It contains the following section:
•  About the external interface to the Performance Monitors registers on page I3-10734.

Note

Performance Monitors external register descriptions on page I5-10753 describes the external view of the Performance Monitors registers.
I3.1 About the external interface to the Performance Monitors registers

Arm recommends that:

- An implementation provides the OPTIONAL external debug interface to the Performance Monitors registers.

  Note
  A debugger can use this interface to access counters in the Performance Monitors.

- The implementation includes the OPTIONAL support for memory-mapped access to the External debug interface.

  Note
  — Software running on any PE in a system can use this interface to access counters in the Performance Monitors.
  — Privileged software should use the MMU to control access to this interface.

- The external debug interface is implemented as defined in Appendix K6 Recommended External Debug Interface.

When the 64-bit external PMU programmers’ model extension is implemented, all Performance Monitors registers are 64-bit except the 32-bit CoreSight management registers.

Note
This means that register views in the external debug interface to the Performance Monitors registers are the same size as the register views in the System interface when the PE is using AArch64. The 32-bit external view is not accessible for Performance Monitors registers which are extended to 64 bits.

The following sections describe the memory-mapped views of the Performance Monitors registers:

- Differences in the external views of the Performance Monitors registers on page I3-10735.
- Synchronization of changes to the memory-mapped views on page I3-10736.
- Access permissions for external views of the Performance Monitors on page I3-10736.

In this section, unless the context explicitly indicates otherwise, any reference to a memory-mapped view applies equally to a register view using:

- An access through an external debug interface.
- A memory-mapped access.

I3.1.1 Endianness and supported access sizes

When an implementation supports memory-mapped access to the external debug interface the interface is accessed as a little-endian memory-mapped peripheral. External Performance Monitors registers summary on page I5-10748 gives the memory map of these registers.

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the external interface to the Performance Monitors registers implementations must:

- Comply with the requirements of Supported access sizes on page I1-10716.
- When the 64-bit external PMU programmers’ model extension is not implemented, support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register that is mapped to a doubleword-aligned pair of adjacent 32-bit locations, even if all components with direct memory access to the PMU support making 64-bit accesses.

Permitted word-aligned 32-bit accesses are single-copy atomic at word granularity. When the 64-bit external PMU programmers’ model extension is implemented, permitted doubleword-aligned 64-bit accesses are single-copy atomic at doubleword granularity.
I3.1.2 Differences in the external views of the Performance Monitors registers

An external view of the Performance Monitors registers accesses the same registers as the System registers interface described in Performance Monitors Extension registers on page D11-5428, except that:

- The PMSELR is accessible only in the System registers interface.
- The following registers are accessible only in external views:
  - PMPCSR.
  - PMCID1SR.
  - PMVIDSR.
  - PMCID2SR.
  - PMCFGR.
  - PMIIDR.
  - PMDEVAFF0.
  - PMDEVAFF1.
  - PMLAR.
  - PMLSR.
  - PMAUTHSTATUS.
  - PMDEVAFFARCH.
  - PMDEVTYPE.
  - PMPIDR0.
  - PMPIDR1.
  - PMPIDR2.
  - PMPIDR3.
  - PMPIDR4.
  - PMCIDR0.
  - PMCIDR1.
  - PMCIDR2.
  - PMCIDR3.

Performance Monitors external register descriptions on page I5-10753 describes these registers.

- The following controls do not affect the external views:
  - PMSELR.
  - PMUSERENR.
  - HDCR.{TPM, TPMCR, HPMN}.

Instead, see the register descriptions in Chapter I5 External System Control Register Descriptions.

- The PMSWINC_EL0 register is OPTIONAL in the external views.

External views of the Performance Monitors registers when the 64-bit external PMU programmers' model extension is implemented

When the 64-bit external PMU programmers' model extension is implemented, the following registers are accessible only in external views:

- PMDEVAFF, provided as a concatenation of PMDEVAFF0 and PMDEVAFF1.
- If FEAT_PCSRv8p2 is implemented:
  - PMCCIDSR, provided as a concatenation of PMCID1SR and PMCID2SR.
  - PMVCCIDSR, provided as a concatenation of PMCID1SR and PMVIDSR.
- The following registers, which provide direct read/write access to the event counter control and status bits, and access the same registers as the corresponding SET and CLR pairs in the System register interface:
  - PMCNTEN.
Recommended External Interface to the Performance Monitors

I3.1 About the external interface to the Performance Monitors registers

---

— PMINTEN.
— PMOVS.

--- Note ---
— Writing 1 to a bit in one of these registers causes the corresponding SET and CLR register values to be 1.
— Writing 0 to a bit in one of these registers causes the corresponding SET and CLR register values to be 0.
— Bits [63:32] of these registers are RES0.

PMSWINC_EL0, otherwise OPTIONAL in the external views, is not accessible through the memory-mapped interface.

Performance Monitors external register descriptions on page I5-10753 describes these registers.

I3.1.3 Synchronization of changes to the memory-mapped views

Synchronization must comply with Synchronization of memory-mapped registers on page I1-10718.

In particular, if a Performance Monitor is visible in both System register and an external view, and is accessed simultaneously through these two mechanisms, the behavior must be as if the accesses occurred atomically and in any order. For more information, see Synchronization of changes to the external debug registers on page H8-10358.

I3.1.4 Access permissions for external views of the Performance Monitors

For more information, see External debug interface register access permissions on page H8-10364.

The following tables show the access permissions for the Performance Monitors registers in a Debug implementation for Armv8 or later architectures:

- Table I3-1 on page I3-10737 when the 64-bit external PMU programmers' model extension is implemented.
- Table I3-2 on page I3-10738 when the 64-bit external PMU programmers' model extension is not implemented.

These tables use the following terms:

**DLK**
When FEAT_DoubleLock is implemented and locked, DoubleLockStatus() == TRUE, accesses to some registers produce an error. Applies to both interfaces.

**EPMAD**
When AllowExternalPMUAccess() == FALSE, external debug access is disabled for the access. If FEAT_Debugv8p4 is implemented, this applies only for Non-secure access to the register. See also Behavior of a not permitted memory-mapped access on page H8-10363.

**Error**
Indicates that the access gives an error response.

**Default**
This shows the default access permissions, if none of the conditions in this list prevent access to the register.

**Off**
The Core power domain is completely off, or in a low-power state where the Core power domain registers cannot be accessed, and EDPRSR.PU will read as zero.

--- Note ---
If debug power is off, then all external debug interface accesses return an error.

**OSLK**
When the OS Lock is locked, OSLAR_EL1.OSLK == 1, accesses to some registers produces an error. This column shows the effect of this control on accesses using the external debug interface.

**SLK**
This indicates the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the optional Software Lock is locked. See Register access permissions for memory-mapped accesses on page H8-10362.
For all other accesses, this column is ignored.

--- Note ---

When the 64-bit external PMU programmers’ model extension is implemented, the Software Lock is not implemented.

 Indicates that the control has no effect on the behavior of the access:
• If no other control affects the behavior, the Default access behavior applies.
• However, another control might determine the behavior.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Off</th>
<th>DLK</th>
<th>OSLK</th>
<th>EPMAD</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000+8xn</td>
<td>PMEVNCNTR&lt;n&gt;_EL0a</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
</tr>
<tr>
<td>0x0F8</td>
<td>PMCCNTR_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
</tr>
<tr>
<td>0x200</td>
<td>PMPCSRb</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
</tr>
<tr>
<td>0x208</td>
<td>PMVCIDSRb</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
</tr>
<tr>
<td>0x220</td>
<td>PMPCSRb</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
</tr>
<tr>
<td>0x228</td>
<td>PMCCIDSRb</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
</tr>
<tr>
<td>0x400+8xn</td>
<td>PMEVTYPEYER&lt;n&gt;_EL0a</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
</tr>
<tr>
<td>0x4F8</td>
<td>PMCCFILTR_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
</tr>
<tr>
<td>0x600-0x7FC</td>
<td>-</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED registersc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xC00</td>
<td>PMCNTENSET_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
</tr>
<tr>
<td>0xC10</td>
<td>PMCNTEN</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
</tr>
<tr>
<td>0xC20</td>
<td>PMCNTENCLR_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
</tr>
<tr>
<td>0xC40</td>
<td>PMINTENSET_EL1</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
</tr>
<tr>
<td>0xC50</td>
<td>PMINTEN</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
</tr>
<tr>
<td>0xC60</td>
<td>PMINTENCLR_EL1</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
</tr>
<tr>
<td>0xC80</td>
<td>PMOVSCCLR_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
</tr>
<tr>
<td>0xC90</td>
<td>PMOVS</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
</tr>
<tr>
<td>0xCC0</td>
<td>PMOVSET_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
</tr>
<tr>
<td>0xD80-0x0FC</td>
<td>-</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED registersc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xE00</td>
<td>PMCFGR</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xE08</td>
<td>PMIIDR</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xE10</td>
<td>PMCR_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
</tr>
<tr>
<td>0xE30-0xE3C</td>
<td>-</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED registersc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xE40</td>
<td>PMMIR</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RO</td>
</tr>
</tbody>
</table>
Table I3-1  Access permissions for the Performance Monitors registers when the 64-bit external PMU programmers’ model extension is implemented (continued)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Off</th>
<th>DLK</th>
<th>OSLK</th>
<th>EPMAD</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE80-0xEF6</td>
<td>Integration registers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xF00-0xFFFF6</td>
<td>Management registers and CoreSight compliance on page K6-11688</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. Implemented event counters only. $n$ is the counter number.

b. Implemented only if FEAT_PCSRv8p2 is implemented. See Chapter H7 The PC Sample-based Profiling Extension.

c. See IMPLEMENTATION DEFINED registers on page H8-10366.

Table I3-2  Access permissions for the Performance Monitors registers when the 64-bit external PMU programmers’ model is not implemented

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Off</th>
<th>DLK</th>
<th>OSLK</th>
<th>EPMAD</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000+8xn</td>
<td>PMEVCNTR&lt;$n$&gt;_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW RO</td>
</tr>
<tr>
<td>0x008</td>
<td>PMCCNTR_EL0[31:0]</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW RO</td>
</tr>
<tr>
<td>0x0FC</td>
<td>PMCCNTR_EL0[63:32]</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW RO</td>
</tr>
<tr>
<td>0x200</td>
<td>PMPCSR[31:0]b</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0x204</td>
<td>PMPCSR[63:32]b</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0x208</td>
<td>PMCID1ISRb</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0x20C</td>
<td>PMVIDSRb</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0x220</td>
<td>PMPCSR[31:0]b</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0x224</td>
<td>PMPCSR[63:32]b</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0x228</td>
<td>PMCID1ISRb</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0x22C</td>
<td>PMCID2ISRb</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>-</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0x400+4xn</td>
<td>PMEVTYPER&lt;$n$&gt;_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW RO</td>
</tr>
<tr>
<td>0x47C</td>
<td>PMCCFILTER_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW RO</td>
</tr>
<tr>
<td>0x600-0x7FC</td>
<td>-</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED registers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xA00+4xn</td>
<td>PMEVTYPER&lt;$n$&gt;_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW RO</td>
</tr>
<tr>
<td>0xA80-0xBFC</td>
<td>-</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED registers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xC00</td>
<td>PMCNTENSET_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW RO</td>
</tr>
<tr>
<td>0xC20</td>
<td>PMCNTENCLR_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW RO</td>
</tr>
<tr>
<td>0xC40</td>
<td>PMINTENSET_EL1</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW RO</td>
</tr>
<tr>
<td>0xC60</td>
<td>PMINTENCLR_EL1</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW RO</td>
</tr>
<tr>
<td>0xC80</td>
<td>PMOVSCLR_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW RO</td>
</tr>
<tr>
<td>0xCA0</td>
<td>PMSWINC_EL0c</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>WO W1</td>
</tr>
</tbody>
</table>
### Table I3-2 Access permissions for the Performance Monitors registers when the 64-bit external PMU programmers' model is not implemented (continued)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Off</th>
<th>DLK</th>
<th>OSLK</th>
<th>EPMAD</th>
<th>Default</th>
<th>SLK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xC0</td>
<td>PMOVSET_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0xD80-0xDFC</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IMPLEMENTATION DEFINED registers</td>
<td></td>
</tr>
<tr>
<td>0xE00</td>
<td>PMCFGR</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xE04</td>
<td>PMCR_EL0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RW</td>
<td>RO</td>
</tr>
<tr>
<td>0xE08</td>
<td>PMIIDR</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xE20</td>
<td>PMCEID0</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xE24</td>
<td>PMCEID1</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xE28</td>
<td>PMCEID2</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xE2C</td>
<td>PMCEID3</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xE80-0xEFC</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IMPLEMENTATION DEFINED registers</td>
<td></td>
</tr>
<tr>
<td>0xE40</td>
<td>PMMIR</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
<td>RO</td>
<td>RO</td>
</tr>
<tr>
<td>0xE80-0xEFC</td>
<td>Integration registers</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IMPLEMENTATION DEFINED registers</td>
<td></td>
</tr>
<tr>
<td>0xF00-0xFFC</td>
<td>Management registers and CoreSight compliance on page K6-11688</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IMPLEMENTATION DEFINED registers</td>
<td></td>
</tr>
</tbody>
</table>

- Implemented event counters only. \( n \) is the counter number.
- Implemented only if FEAT_PCSRv8p2 is implemented. See Chapter H7 The PC Sample-based Profiling Extension.
- Only if the OPTIONAL PMSWINC_EL0 register is implemented in the external debug interface.
- See IMPLEMENTATION DEFINED registers on page H8-10366.
I3.1 About the external interface to the Performance Monitors registers
Chapter I4
Recommended External Interface to the Activity Monitors

This chapter describes the optional external interface to the Activity Monitors Extension registers. It contains the following section:

- About the external interface to the Activity Monitors Extension registers on page I4-10742

Note

Activity Monitors external register descriptions on page I5-10838 describes the external view of the Activity Monitors Extension registers.
I4.1  About the external interface to the Activity Monitors Extension registers

If an implementation supports the Activity Monitors Extension, it may optionally support an external memory-mapped interface to the Activity Monitors Extension, and, if so, may further optionally support CoreSight device registers and ID registers.

The memory access sizes supported by the external interface to the Activity Monitors registers:

- Comply with the requirements of Supported access sizes on page I1-10716.
- Include word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register mapped to a doubleword-aligned pair of adjacent 32-bit locations, even if all components with direct memory access to the AMU support making 64-bit accesses.

The base address of the memory-mapped view is aligned to a 4KB boundary, but is otherwise IMPLEMENTATION DEFINED. The address offsets for the memory-mapped view are given in Table I5-3 on page I5-10836.

I4.1.1  Differences in the external views of the Activity Monitors Extension registers

The external memory-mapped interface view of the Activity Monitors Extension registers accesses the same registers as the System registers interface to the registers, except that:

- The following are accessible only in the System registers interface:
  - AMUSERENR_EL0
  - AMEVCTVOFF0<el>: EL2
  - AMEVCTVOFF1<el>: EL2
  - AMCG1IDR_EL0

- If implemented, the following registers are accessible only in the memory-mapped view:
  - AMIIDR
  - AMDEVAFF0
  - AMDEVAFF1
  - AMDEVARCH
  - AMDEVTYPER
  - AMPIDR0
  - AMPIDR1
  - AMPIDR2
  - AMPIDR3
  - AMPIDR4
  - AMCIDR0
  - AMCIDR1
  - AMCIDR2
  - AMCIDR3

Activity Monitors external register descriptions on page I5-10838 describes these registers.

- If FEAT_AMUv1p1 virtualization of the activity monitors is enabled, the memory-mapped view of the registers presents the physical view of the counter without any offset. Virtualization of the Activity Monitors does not affect the memory-mapped view of the registers.

Note: The memory mapped view of the activity monitors is unaffected by AMCR_EL0.CG1RZ and AMCR.CG1RZ.
I4.1.2 Access during reset and power transitions

As described in Power and reset domains on page D12-5431, the power and reset domains of the activity monitoring unit are named the AMU domain and AMU reset, and when reset of the AMU power domain occurs, the activity monitoring unit is reset and the counters are reset to zero.

If the AMU domain is an always-on power domain, while the PE is reset or powered down counter values may be preserved and might be accessible by memory-mapped access.

If the AMU domain is the Core power domain, while the PE is reset or powered down and when a memory-mapped access occurs, the access reads as zero and the bus access completes without an error.
Chapter I5
External System Control Register Descriptions

This chapter describes the external system control registers. It excludes the External debug registers that are described in Chapter H9 External Debug Register Descriptions. It contains the following sections:

- About the external system control register descriptions on page I5-10746.
- External Performance Monitors registers summary on page I5-10748.
- Performance Monitors external register descriptions on page I5-10753.
- External Activity Monitors Extension registers summary on page I5-10836.
- Activity Monitors external register descriptions on page I5-10838.
- Generic Timer memory-mapped registers overview on page I5-10874.
- Generic Timer memory-mapped register descriptions on page I5-10875.
- RAS register descriptions on page I5-10919.
I5.1 About the external system control register descriptions

This chapter describes the external system control registers other than the external debug registers. That is, it describes:

An external view of the Performance Monitors registers

Arm recommends that implementations provide access to the Performance Monitors registers through the OPTIONAL External debug interface, and provide the OPTIONAL memory-mapped interface to this interface:

* External Performance Monitors registers summary on page I5-10748 lists the registers that are accessible in this view of the Performance Monitors, and describes their memory map.
* Performance Monitors external register descriptions on page I5-10753 describes each of the memory-mapped registers.

Chapter I3 Recommended External Interface to the Performance Monitors describes the recommended interface to these registers.

Note

Chapter D11 The Performance Monitors Extension describes the Performance Monitors. The following sections describe the System register interfaces to the Performance Monitors:

* Performance Monitors registers on page D17-6740, for accesses from an Exception level that is using AArch64.
* Performance Monitors registers on page G8-9963, for accesses from an Exception level that is using AArch32.

An external view of the Activity Monitors Extension registers

An implementation which supports the Activity Monitors Extension may support an optional external memory-mapped interface to the Activity Monitors Extension registers.

* External Activity Monitors Extension registers summary on page I5-10836 lists the registers that are accessible in this view of the Performance Monitors, and describes their memory map.
* Activity Monitors external register descriptions on page I5-10838 describes each of the memory-mapped registers.

Chapter I3 Recommended External Interface to the Performance Monitors describes the recommended interface to these registers.

Note

Chapter D12 The Activity Monitors Extension describes the Activity Monitors. The following sections describe the System register interfaces to the Activity Monitors:

* Activity Monitors registers on page D17-6816, for accesses from an Exception level that is using AArch64.
* Activity Monitors registers on page G8-10047, for accesses from an Exception level that is using AArch32.

The registers for the system level Generic Timer component

Any implementation that includes the Generic Timer must include the memory-mapped system level component described in Chapter I2 System Level Implementation of the Generic Timer. In this chapter:

* Generic Timer memory-mapped registers overview on page I5-10874 gives an overview of the registers, referring to Chapter I2 for more information.
* Generic Timer memory-mapped register descriptions on page I5-10875 describes each of the memory-mapped registers.
Chapter D10 The Generic Timer in AArch64 state describes the Generic Timer component that is accessible using the System registers. The following sections describe the System register interfaces to that component:

- *Generic Timer registers on page D17-7014*, for accesses from an Exception level that is using AArch64.
- *Generic Timer registers on page G8-10144*, for accesses from an Exception level that is using AArch32.

Chapter H9 External Debug Register Descriptions describes the external debug registers.
## 5.2 External Performance Monitors registers summary

When an implementation provides access to the Performance Monitors registers through the External debug interface, that interface provides access to:

- Performance Monitors System registers.
- A read-only configuration register, PMCFGR.
- The optional CoreSight registers for the Performance Monitors, if they are implemented.

The locations of the registers are defined as offsets from a system-defined base address. *Performance Monitors external register views* defines this memory map.

### 5.2.1 Performance Monitors external register views

The following tables show the external view of the Performance Monitors registers:

- Table I5-1 on page I5-10748 when the 64-bit external PMU programmers' model extension is implemented.
- Table I5-2 on page I5-10750 when the 64-bit external PMU programmers' model extension is not implemented.

All other entries are reserved.

**Note**

- Counters that are reserved because HDCR.HPMN has been changed from its reset value remain visible in any external view.
- The registers that relate to an implemented event counter, PMNx, are PMEVCNTR<n> and PMEVTYPER<n>.
- Tables in this section only list the Armv8 registers. For encoding information of the registers introduced by Armv9, see the individual register descriptions in this chapter.

Each entry in the Name column links to the register description in *Performance Monitors external register descriptions* on page I5-10753.

### Table I5-1 Performance Monitors external register views when the 64-bit external PMU programmers' model extension is implemented

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMEVCNTR&lt;n&gt;_EL0</td>
<td>RW</td>
<td>Performance Monitors Event Counter Register</td>
<td>0x000+8xn</td>
</tr>
<tr>
<td>PMCCNTR_EL0</td>
<td>RW</td>
<td>Performance Monitors Cycle Counter Register</td>
<td>0x0F8</td>
</tr>
<tr>
<td>PMPCSR&lt;sup&gt;a&lt;/sup&gt;</td>
<td>RW</td>
<td>Program Counter Sample Register</td>
<td>0x200</td>
</tr>
<tr>
<td>PMVCIDSR&lt;sup&gt;a&lt;/sup&gt;</td>
<td>RW</td>
<td>CONTEXTIDR_EL1 and VMID Sample Register</td>
<td>0x208</td>
</tr>
<tr>
<td>PMPCSR&lt;sup&gt;a&lt;/sup&gt;</td>
<td>RW</td>
<td>Program Counter Sample Register, alias</td>
<td>0x220</td>
</tr>
<tr>
<td>PMCCIDSR&lt;sup&gt;a&lt;/sup&gt;</td>
<td>RW</td>
<td>CONTEXTIDR_ELx Sample Register</td>
<td>0x228</td>
</tr>
<tr>
<td>PMEVTYPER&lt;n&gt;_EL0</td>
<td>RW</td>
<td>Performance Monitors Event Type and Filter Register</td>
<td>0x400+8xn</td>
</tr>
<tr>
<td>PMCCFILTR_EL0</td>
<td>RW</td>
<td>Performance Monitors Cycle Counter Filter Register</td>
<td>0x4F8</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>IMPLEMENTATION DEFINED</td>
<td>0x600-0x7FC</td>
</tr>
<tr>
<td>PMCNTENSET_EL0</td>
<td>RW</td>
<td>Performance Monitors Count Enable Set Register</td>
<td>0xC00</td>
</tr>
<tr>
<td>PMCNTEN</td>
<td>RW</td>
<td>Performance Monitors Count Enable Register</td>
<td>0xC10</td>
</tr>
<tr>
<td>PMCNTENCLR_EL0</td>
<td>RW</td>
<td>Performance Monitors Count Enable Clear Register</td>
<td>0xC20</td>
</tr>
</tbody>
</table>
### Table I5-1 Performance Monitors external register views when the 64-bit external PMU programmers’ model extension is implemented (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMINTENSET_EL1</td>
<td>RW</td>
<td>Performance Monitors Interrupt Enable Set Register</td>
<td>0xC40</td>
</tr>
<tr>
<td>PMINTEN</td>
<td>RW</td>
<td>Performance Monitors Interrupt Enable Register</td>
<td>0xC50</td>
</tr>
<tr>
<td>PMINTENCLR_EL1</td>
<td>RW</td>
<td>Performance Monitors Interrupt Enable Clear Register</td>
<td>0xC60</td>
</tr>
<tr>
<td>PMOVSCLR_EL0</td>
<td>RW</td>
<td>Performance Monitors Overflow Flag Status Clear Register</td>
<td>0xC80</td>
</tr>
<tr>
<td>PMOVS</td>
<td>RW</td>
<td>Performance Monitors Overflow Flag Status Register</td>
<td>0xC90</td>
</tr>
<tr>
<td>PMOVSET_EL0</td>
<td>RW</td>
<td>Performance Monitors Overflow Flag Status Set Register</td>
<td>0xCC0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED</td>
<td>0xD80-0xDFC</td>
</tr>
<tr>
<td>PMCFGR</td>
<td>RO</td>
<td>Performance Monitors Configuration Register</td>
<td>0xE00</td>
</tr>
<tr>
<td>PMIIDR</td>
<td>RO</td>
<td>Performance Monitors Implementation Identification Register</td>
<td>0xE08</td>
</tr>
<tr>
<td>PMCR_EL0</td>
<td>RW</td>
<td>Performance Monitors Control Register</td>
<td>0xE10</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED</td>
<td>0xE30-0xE3C</td>
</tr>
<tr>
<td>PMMIR</td>
<td>RO</td>
<td>Performance Monitors Machine Identification Register</td>
<td>0xE40</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED</td>
<td>0xE80-0xEF0</td>
</tr>
<tr>
<td>PMITCTRL(b)</td>
<td>RW</td>
<td>Integration Model Control registers</td>
<td>0xF00</td>
</tr>
<tr>
<td>PMDEVAFF(b)</td>
<td>RO</td>
<td>Device Affinity Register</td>
<td>0xFA8</td>
</tr>
<tr>
<td>PMLAR(b, c)</td>
<td>WO</td>
<td>Lock Access Register</td>
<td>0xFB0</td>
</tr>
<tr>
<td>PMLSR(b, c)</td>
<td>RO</td>
<td>Lock Status Register</td>
<td>0xFB4</td>
</tr>
<tr>
<td>PMAUTHSTATUS(b)</td>
<td>RO</td>
<td>Authentication Status Register</td>
<td>0xFB8</td>
</tr>
<tr>
<td>PMDEVAR(b)</td>
<td>RO</td>
<td>Device Architecture Register</td>
<td>0xFBC</td>
</tr>
<tr>
<td>PMDEVID(a)</td>
<td>RO</td>
<td>Performance Monitors Device ID Register</td>
<td>0xFC8</td>
</tr>
<tr>
<td>PMDEVTYPE(b)</td>
<td>RO</td>
<td>Device Type Register</td>
<td>0xFCC</td>
</tr>
<tr>
<td>PMPIDR4(b)</td>
<td>RO</td>
<td>Peripheral ID registers</td>
<td>0xFD0</td>
</tr>
<tr>
<td>PMPIDR0(b)</td>
<td>RO</td>
<td></td>
<td>0xFE0</td>
</tr>
<tr>
<td>PMPIDR1(b)</td>
<td>RO</td>
<td></td>
<td>0xFE4</td>
</tr>
<tr>
<td>PMPIDR2(b)</td>
<td>RO</td>
<td></td>
<td>0xFE8</td>
</tr>
<tr>
<td>PMPIDR3(b)</td>
<td>RO</td>
<td></td>
<td>0xFEC</td>
</tr>
<tr>
<td>PMCIDR0(b)</td>
<td>RO</td>
<td>Component ID registers</td>
<td>0xFF0</td>
</tr>
<tr>
<td>PMCIDR1(b)</td>
<td>RO</td>
<td></td>
<td>0xFF4</td>
</tr>
<tr>
<td>PMCIDR2(b)</td>
<td>RO</td>
<td></td>
<td>0xFF8</td>
</tr>
<tr>
<td>PMCIDR3(b)</td>
<td>RO</td>
<td></td>
<td>0xFFC</td>
</tr>
</tbody>
</table>
a. PC Sample-based Profiling Extension registers. Implemented only when FEAT_PCSRv8p2 is implemented, except that from Armv8.2 PMDEVIDt is required regardless of whether FEAT_PCSRv8p2 is implemented. Before Armv8.2, the PC Sample-based Profiling Extension can, instead, be implemented in the memory-mapped debug registers space, see Chapter H7 The PC Sample-based Profiling Extension.

b. CoreSight interface registers, see Management registers and CoreSight compliance on page K6-11688.

c. The Software lock registers are defined as part of CoreSight compliance, but their contents depend on the type of access that is made and whether the OPTIONAL Software lock is implemented. See the register description for details.

### Table I5-2 Performance Monitors external register views when the 64-bit external PMU programmers' model extension is not implemented

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMEVCNTR&lt;n&gt;_EL0</td>
<td>RW</td>
<td>Performance Monitors Event Counter Register</td>
<td>0x000+8xn</td>
</tr>
<tr>
<td>PMCCNTR_EL0[31:0]</td>
<td>RW</td>
<td>Performance Monitors Cycle Counter Register</td>
<td>0x0F8</td>
</tr>
<tr>
<td>PMCCNTR_EL0[63:32]</td>
<td>RW</td>
<td>Performance Monitors Cycle Counter Register</td>
<td>0x0FC</td>
</tr>
<tr>
<td>PMPCSR[31:0]&lt;b&gt;</td>
<td>RW</td>
<td>Program Counter Sample Register, bits[31:0]</td>
<td>0x200</td>
</tr>
<tr>
<td>PMPCSR[63:32]&lt;b&gt;</td>
<td>RW</td>
<td>Program Counter Sample Register, bits[63:32]</td>
<td>0x204</td>
</tr>
<tr>
<td>PMCID1SR&lt;b&gt;</td>
<td>RW</td>
<td>CONTEXTIDR_EL1 Sample Register</td>
<td>0x208</td>
</tr>
<tr>
<td>PMVIDSR&lt;b&gt;</td>
<td>RW</td>
<td>VMID Sample Register</td>
<td>0x20C</td>
</tr>
<tr>
<td>PMPCSR[31:0]&lt;b&gt;</td>
<td>RW</td>
<td>Program Counter Sample Register, bits[31:0], alias</td>
<td>0x220</td>
</tr>
<tr>
<td>PMPCSR[63:32]&lt;b&gt;</td>
<td>RW</td>
<td>Program Counter Sample Register, bits[63:32], alias</td>
<td>0x224</td>
</tr>
<tr>
<td>PMCID1SR&lt;b&gt;</td>
<td>RW</td>
<td>CONTEXTIDR_EL1 Sample Register (alias)</td>
<td>0x228</td>
</tr>
<tr>
<td>PMCID2SR&lt;b&gt;</td>
<td>RW</td>
<td>CONTEXTIDR_EL2 Sample Register</td>
<td>0x22C</td>
</tr>
<tr>
<td>PMEVTYPER&lt;n&gt;_EL0[31:0]</td>
<td>RW</td>
<td>Performance Monitors Event Type and Filter Register</td>
<td>0x400+4xn</td>
</tr>
<tr>
<td>PMCCFILTR_EL0</td>
<td>RW</td>
<td>Performance Monitors Cycle Counter Filter Register</td>
<td>0x47C</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED</td>
<td>0x600-0x7FC</td>
</tr>
<tr>
<td>PMEVTYPER&lt;n&gt;_EL0[63:32]</td>
<td>RW</td>
<td>Performance Monitors Event Type and Filter Register</td>
<td>0xA00+4xn</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED</td>
<td>0xA80-0xBFC</td>
</tr>
<tr>
<td>PMCNTENSET_EL0</td>
<td>RW</td>
<td>Performance Monitors Count Enable Set Register</td>
<td>0xC00</td>
</tr>
<tr>
<td>PMCNTENCLR_EL0</td>
<td>RW</td>
<td>Performance Monitors Count Enable Clear Register</td>
<td>0xC20</td>
</tr>
<tr>
<td>PMINTENSET_EL1</td>
<td>RW</td>
<td>Performance Monitors Interrupt Enable Set Register</td>
<td>0xC40</td>
</tr>
<tr>
<td>PMINTENCLR_EL1</td>
<td>RW</td>
<td>Performance Monitors Interrupt Enable Clear Register</td>
<td>0xC60</td>
</tr>
<tr>
<td>PMOVSCLR_EL0</td>
<td>RW</td>
<td>Performance Monitors Overflow Flag Status Clear Register</td>
<td>0xC80</td>
</tr>
<tr>
<td>PMSWINC_EL0</td>
<td>WO</td>
<td>Performance Monitors Software Increment Register</td>
<td>0xCA0</td>
</tr>
<tr>
<td>PMOVSSET_EL0</td>
<td>RW</td>
<td>Performance Monitors Overflow Flag Status Set Register</td>
<td>0xCC0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>IMPLEMENTATION DEFINED</td>
<td>0xD80-0xDBFC</td>
</tr>
<tr>
<td>PMCFGR</td>
<td>RO</td>
<td>Performance Monitors Configuration Register</td>
<td>0xE00</td>
</tr>
<tr>
<td>PMCR_EL0</td>
<td>RW</td>
<td>Performance Monitors Control Register</td>
<td>0xE04</td>
</tr>
</tbody>
</table>
## Table I5-2 Performance Monitors external register views when the 64-bit external PMU programmers' model extension is not implemented (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMIIDR</td>
<td>RO</td>
<td>Performance Monitors Implementation Identification Register</td>
<td>0xE08</td>
</tr>
<tr>
<td>PMCEID0</td>
<td>RO</td>
<td>Performance Monitors Common Event Identification Register 0</td>
<td>0xE20</td>
</tr>
<tr>
<td>PMCEID1</td>
<td>RO</td>
<td>Performance Monitors Common Event Identification Register 1</td>
<td>0xE24</td>
</tr>
<tr>
<td>PMCEID2</td>
<td>RO</td>
<td>Performance Monitors Common Event Identification Register 2</td>
<td>0xE28</td>
</tr>
<tr>
<td>PMCEID3</td>
<td>RO</td>
<td>Performance Monitors Common Event Identification Register 3</td>
<td>0xE2C</td>
</tr>
<tr>
<td>PMMITCTRL&lt;sup&gt;c&lt;/sup&gt;</td>
<td>RW</td>
<td>Integration Model Control registers</td>
<td>0xF00</td>
</tr>
<tr>
<td>PMDEVAFFF0&lt;sup&gt;c&lt;/sup&gt;</td>
<td>RO</td>
<td>Device Affinity registers</td>
<td>0xFA8</td>
</tr>
<tr>
<td>PMDEVAFF1&lt;sup&gt;c&lt;/sup&gt;</td>
<td>RO</td>
<td></td>
<td>0xFAC</td>
</tr>
<tr>
<td>PMLAR&lt;sup&gt;c, d&lt;/sup&gt;</td>
<td>WO</td>
<td>Lock Access Register</td>
<td>0xFB0</td>
</tr>
<tr>
<td>PMLSR&lt;sup&gt;c, d&lt;/sup&gt;</td>
<td>RO</td>
<td>Lock Status Register</td>
<td>0xFB4</td>
</tr>
<tr>
<td>PMAUTHSTATUS&lt;sup&gt;c&lt;/sup&gt;</td>
<td>RO</td>
<td>Authentication Status Register</td>
<td>0xFB8</td>
</tr>
<tr>
<td>PMDEVARCH&lt;sup&gt;c&lt;/sup&gt;</td>
<td>RO</td>
<td>Device Architecture Register</td>
<td>0xFBc</td>
</tr>
<tr>
<td>PMDEVID&lt;sup&gt;b&lt;/sup&gt;</td>
<td>RO</td>
<td>Performance Monitors Device ID Register</td>
<td>0xFC8</td>
</tr>
<tr>
<td>PMDEVTYPE&lt;sup&gt;c&lt;/sup&gt;</td>
<td>RO</td>
<td>Device Type Register</td>
<td>0xFCc</td>
</tr>
<tr>
<td>PMPIDR4&lt;sup&gt;c&lt;/sup&gt;</td>
<td>RO</td>
<td>Peripheral ID registers</td>
<td>0xFD0</td>
</tr>
<tr>
<td>PMPIDR0&lt;sup&gt;c&lt;/sup&gt;</td>
<td>RO</td>
<td></td>
<td>0xFE0</td>
</tr>
<tr>
<td>PMPIDR1&lt;sup&gt;c&lt;/sup&gt;</td>
<td>RO</td>
<td></td>
<td>0xFE4</td>
</tr>
<tr>
<td>PMPIDR2&lt;sup&gt;c&lt;/sup&gt;</td>
<td>RO</td>
<td></td>
<td>0xFE8</td>
</tr>
<tr>
<td>PMPIDR3&lt;sup&gt;c&lt;/sup&gt;</td>
<td>RO</td>
<td></td>
<td>0xFEc</td>
</tr>
<tr>
<td>PMCIDR0&lt;sup&gt;c&lt;/sup&gt;</td>
<td>RO</td>
<td>Component ID registers</td>
<td>0xFF0</td>
</tr>
<tr>
<td>PMCIDR1&lt;sup&gt;c&lt;/sup&gt;</td>
<td>RO</td>
<td></td>
<td>0xFF4</td>
</tr>
<tr>
<td>PMCIDR2&lt;sup&gt;c&lt;/sup&gt;</td>
<td>RO</td>
<td></td>
<td>0xFF8</td>
</tr>
<tr>
<td>PMCIDR3&lt;sup&gt;c&lt;/sup&gt;</td>
<td>RO</td>
<td></td>
<td>0xFFc</td>
</tr>
</tbody>
</table>

a. The interface must support at least single-copy atomic 32-bit accesses. If single-copy atomic 64-bit access to the registers is not possible, software must use a high-low-high read access to read the counter value if the counter is enabled.

b. PC Sample-based Profiling Extension registers. Implemented only when FEAT_PCSRv8p2 is implemented, except that from Armv8.2 PMDEVID is required regardless of whether FEAT_PCSRv8p2 is implemented.

Before Armv8.2, the PC Sample-based Profiling Extension can, instead, be implemented in the memory-mapped debug registers space, see Chapter H7 The PC Sample-based Profiling Extension.

c. CoreSight interface registers, see Management registers and CoreSight compliance on page K6-11688.
d. The Software lock registers are defined as part of CoreSight compliance, but their contents depend on the type of access that is made and whether the OPTIONAL Software lock is implemented. See the register description for details.
I5.3 Performance Monitors external register descriptions

This section describes the external view of the Performance Monitors registers. *External Performance Monitors registers summary* on page I5-10748 lists these registers in offset order.
I5.3.1 PMAUTHSTATUS, Performance Monitors Authentication Status register

The PMAUTHSTATUS characteristics are:

**Purpose**

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for Performance Monitors.

**Configurations**

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is OPTIONAL, and is required for CoreSight compliance. Arm recommends that this register is implemented.

**Attributes**

PMAUTHSTATUS is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-28</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>27-26</td>
<td>RTNID, bits [27:26]</td>
<td></td>
</tr>
<tr>
<td>25-24</td>
<td>Root non-invasive debug</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This field has the same value as</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DBGAUTHSTATUS_EL1.RTNID.</td>
<td></td>
</tr>
<tr>
<td>23-16</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>15-14</td>
<td>RLNID, bits [15:14]</td>
<td></td>
</tr>
<tr>
<td>13-12</td>
<td>Realm non-invasive debug</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This field has the same value as</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DBGAUTHSTATUS_EL1.RLNID.</td>
<td></td>
</tr>
<tr>
<td>11-8</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>7-6</td>
<td>SNID, bits [7:6]</td>
<td></td>
</tr>
<tr>
<td>5-4</td>
<td>Holds the same value as</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DBGAUTHSTATUS_EL1.SNID.</td>
<td></td>
</tr>
<tr>
<td>3-0</td>
<td>SID, bits [3:0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Secure invasive debug</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Possible values of this field are:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0b00 Not implemented.</td>
<td></td>
</tr>
</tbody>
</table>

RES0 31 28 27 26 25 24 23 16 15 14 13 12 11 8 7 6 5 4 3 2 1 0
RTNID RES0 RLID RES0 SNID SID NSNID
RLNID RES0
RLID RES0
SNID RES0
SID RES0
NSNID
All other values are reserved.

**NSID, bits [3:2]**

Holds the same value as `DBGAUTHSTATUS_EL1.NSID`.

**NSID, bits [1:0]**

Non-secure invasive debug. Possible values of this field are:

- 0b00 Not implemented.
- All other values are reserved.

**Accessing the PMAUTHSTATUS:**

PMAUTHSTATUS can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFB8</td>
<td>PMAUTHSTATUS</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When `FEAT_DoPD` is not implemented or `IsCorePowered()` accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
### 5.3.2 PMCCFILTR_EL0, Performance Monitors Cycle Counter Filter Register

The PMCCFILTR_EL0 characteristics are:

**Purpose**

Determines the modes in which the Cycle Counter, PMCCNTR_EL0, increments.

**Configurations**

External register PMCCFILTR_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMCCFILTR_EL0[31:0].

External register PMCCFILTR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCCFILTR[31:0].

PMCCFILTR_EL0 is in the Core power domain.

On a Warm or Cold reset, RW fields in this register reset to:

- Architecturally **UNKNOWN** values if the reset is to an Exception level that is using AArch64.
- 0 if the reset is to an Exception level that is using AArch32.

The register is not affected by an External debug reset.

**Attributes**

PMCCFILTR_EL0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the PMCCFILTR_EL0.NSK bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If FEAT_RME is implemented, then counting in Realm EL1 is further controlled by the PMCCFILTR_EL0.RLK bit.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Count cycles in EL1.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Do not count cycles in EL1.</td>
</tr>
<tr>
<td>U</td>
<td>30</td>
<td>User filtering bit. Controls counting in EL0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the PMCCFILTR_EL0.NSU bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If FEAT_RME is implemented, then counting in Realm EL0 is further controlled by the PMCCFILTR_EL0.RLU bit.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Count cycles in EL0.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Do not count cycles in EL0.</td>
</tr>
<tr>
<td>NSK</td>
<td>29</td>
<td>Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1.</td>
</tr>
</tbody>
</table>

*When EL3 is implemented:*
If the value of this bit is equal to the value of the PMCCFILTR_EL0.P bit, cycles in Non-secure EL1 are counted.
Otherwise, cycles in Non-secure EL1 are not counted.

**Otherwise:**
Reserved, RES0.

**NSU, bit [28]**

*When EL3 is implemented:*
Non-secure EL0 (Unprivileged) filtering bit. Controls counting in Non-secure EL0.
If the value of this bit is equal to the value of the PMCCFILTR_EL0.U bit, cycles in Non-secure EL0 are counted.
Otherwise, cycles in Non-secure EL0 are not counted.

**Otherwise:**
Reserved, RES0.

**NSH, bit [27]**

*When EL2 is implemented:*
EL2 (Hypervisor) filtering bit. Controls counting in EL2.
If FEAT_SEL2 and EL3 are implemented, counting in Secure EL2 is further controlled by the PMCCFILTR_EL0.SH bit.
If FEAT_RME is implemented, then counting in Realm EL2 is further controlled by the PMCCFILTR_EL0.RLH bit.

0b0  Do not count cycles in EL2.
0b1  Count cycles in EL2.

**Otherwise:**
Reserved, RES0.

**M, bit [26]**

*When EL3 is implemented:*
Secure EL3 filtering bit.
If the value of this bit is equal to the value of the PMCCFILTR_EL0.P bit, cycles in Secure EL3 are counted.
Otherwise, cycles in Secure EL3 are not counted.
Most applications can ignore this field and set its value to 0.

--- **Note** ---
This field is not visible in the AArch32 PMCCFILTR System register.

**Otherwise:**
Reserved, RES0.

**Bit [25]**
Reserved, RES0.

**SH, bit [24]**

*When FEAT_SEL2 is implemented and EL3 is implemented:*
Secure EL2 filtering.
If the value of this bit is not equal to the value of the PMCCFILTR_EL0.NSH bit, cycles in Secure EL2 are counted.
Otherwise, cycles in Secure EL2 are not counted.
Note
This field is not visible in the AArch32 PMCCFILTR System register.

Otherwise:
Reserved, RES0.

Bit [23]
Reserved, RES0.

RLK, bit [22]
When FEAT_RME is implemented:
Realm EL1 (kernel) filtering bit. Controls counting in Realm EL1.
If the value of this bit is equal to the value of the PMCCFILTR_EL0.P bit, cycles in Realm EL1 are counted.
Otherwise, cycles in Realm EL1 are not counted.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

RLU, bit [21]
When FEAT_RME is implemented:
Realm EL0 (unprivileged) filtering bit. Controls counting in Realm EL0.
If the value of this bit is equal to the value of the PMCCFILTR_EL0.U bit, cycles in Realm EL0 are counted.
Otherwise, cycles in Realm EL0 are not counted.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

RLH, bit [20]
When FEAT_RME is implemented:
Realm EL2 filtering bit. Controls counting in Realm EL2.
If the value of this bit is not equal to the value of the PMCCFILTR_EL0.NSH bit, cycles in Realm EL2 are counted.
Otherwise, cycles in Realm EL2 are not counted.
The reset behavior of this field is:
• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bits [19:0]
Reserved, RES0.
Accessing the PMCCFILTR_EL0:

--- Note ---

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCCFILTR_EL0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0x47C</td>
<td>PMCCFILTR_EL0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.
I5.3.3 PMCCNTR_EL0, Performance Monitors Cycle Counter

The PMCCNTR_EL0 characteristics are:

**Purpose**

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. For more information, see *Time as measured by the Performance Monitors cycle counter on page D11-5246*. PMCCFILTR_EL0 determines the modes and states in which the PMCCNTR_EL0 can increment.

**Configurations**

External register PMCCNTR_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMCCNTR_EL0[63:0].

External register PMCCNTR_EL0 bits [63:0] are architecturally mapped to AArch32 System register PMCCNTR[63:0].

PMCCNTR_EL0 is in the Core power domain.

**Attributes**

PMCCNTR_EL0 is a 64-bit register.

**Field descriptions**

CCNT, bits [63:0]

Cycle count. Depending on the values of PMCR_EL0.{LC,D}, the cycle count increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to PMCR_EL0.C sets this field to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing the PMCCNTR_EL0:**

--- Note ---

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCCNTR_EL0[31:0] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0x0F8</td>
<td>PMCCNTR_EL0</td>
<td>31:0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to PMCCNTR_EL0[31:0] are RO.
• When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to PMCCNTR_EL0[31:0] are RW.

• Otherwise accesses to PMCCNTR_EL0[31:0] generate an error response.

PMCCNTR_EL0[63:32] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0x0FC</td>
<td>PMCCNTR_EL0</td>
<td>63:32</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

• When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to PMCCNTR_EL0[63:32] are RO.

• When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to PMCCNTR_EL0[63:32] are RW.

• Otherwise accesses to PMCCNTR_EL0[63:32] generate an error response.
I5.3.4 PMCEID0, Performance Monitors Common Event Identification register 0

The PMCEID0 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range 0x0000 to 0x001F.

For more information about the Common events and the use of the PMCEIDn registers, see *The PMU event number space and common events* on page D11-5275.

Note

This view of the register was previously called PMCEID0_EL0.

Configurations

External register PMCEID0 bits [31:0] are architecturally mapped to AArch64 System register PMCEID0_EL0[31:0].

External register PMCEID0 bits [31:0] are architecturally mapped to AArch32 System register PMCEID0[31:0].

PMCEID0 is in the Core power domain.

Attributes

PMCEID0 is a 32-bit register.

Field descriptions

ID<n>, bit [n], for n = 31 to 0

ID[n] corresponds to Common event n.

For each bit:

0b0 The Common event is not implemented, or not counted.

0b1 The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.
A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing the PMCEID0:

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCEID0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xE20</td>
<td>PMCEID0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and AllowExternalPMUAccess() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.5 PMCEID1, Performance Monitors Common Event Identification register 1

The PMCEID1 characteristics are:

**Purpose**

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range 0x020 to 0x03F.

For more information about the Common events and the use of the PMCEIDn registers, see *The PMU event number space and common events* on page D11-5275.

--- Note ---

This view of the register was previously called PMCEID1_EL0.

--- Configurations ---

External register PMCEID1 bits [31:0] are architecturally mapped to AArch64 System register PMCEID1_EL0[31:0].

External register PMCEID1 bits [31:0] are architecturally mapped to AArch32 System register PMCEID1[31:0].

PMCEID1 is in the Core power domain.

--- Attributes ---

PMCEID1 is a 32-bit register.

--- Field descriptions ---

`ID<n>`, bit [n], for n = 31 to 0

ID[n] corresponds to Common event (0x020 + n).

For each bit:

- 0b0 The Common event is not implemented, or not counted.
- 0b1 The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

--- Note ---

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.
A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

--- Note ---
Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<\text{n}> registers of that earlier version of the PMU architecture.

### Accessing the PMCEID1:

--- Note ---
AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCEID1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xE24</td>
<td>PMCEID1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and AllowExternalPMUAccess() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.6 PMCEID2, Performance Monitors Common Event Identification register 2

The PMCEID2 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range 0x4000 to 0x401F.

For more information about the Common events and the use of the PMCEIDn registers, see The PMU event number space and common events on page D11-5275.

Configurations

External register PMCEID2 bits [31:0] are architecturally mapped to AArch64 System register PMCEID0_EL0[63:32].

External register PMCEID2 bits [31:0] are architecturally mapped to AArch32 System register PMCEID2[31:0].

PMCEID2 is in the Core power domain.

This register is present only when FEAT_PMUv3p1 is implemented. Otherwise, direct accesses to PMCEID2 are RES0.

Attributes

PMCEID2 is a 32-bit register.

Field descriptions

IDhi<n>, bit [n], for n = 31 to 0

IDhi[n] corresponds to Common event (0x4000 + n).

For each bit:

0b0 The Common event is not implemented, or not counted.

0b1 The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.
Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<e<n> registers of that earlier version of the PMU architecture.

Accessing the PMCEID2:

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCEID2 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xE28</td>
<td>PMCEID2</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and AllowExternalPMUAccess() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.7 PMCEID3, Performance Monitors Common Event Identification register 3

The PMCEID3 characteristics are:

**Purpose**

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range 0x4020 to 0x403F.

For more information about the Common events and the use of the PMCEIDn registers, see *The PMU event number space and common events* on page D11-5275.

**Configurations**

External register PMCEID3 bits [31:0] are architecturally mapped to AArch64 System register PMCEID1_EL0[63:32].

External register PMCEID3 bits [31:0] are architecturally mapped to AArch32 System register PMCEID3[31:0].

PMCEID3 is in the Core power domain.

This register is present only when FEAT_PMUv3p1 is implemented. Otherwise, direct accesses to PMCEID3 are RES0.

**Attributes**

PMCEID3 is a 32-bit register.

**Field descriptions**

IDhi<n>, bit [n], for n = 31 to 0

IDhi[n] corresponds to Common event (0x4020 + n).

For each bit:

- **0b0** The Common event is not implemented, or not counted.
- **0b1** The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

--- Note ---

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.
Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing the PMCEID3:

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCEID3 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xE2C</td>
<td>PMCEID3</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and AllowExternalPMUAccess() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
### PMCFGR, Performance Monitors Configuration Register

The PMCFGR characteristics are:

**Purpose**

Contains PMU-specific configuration data.

**Configurations**

PMCFGR is in the Core power domain.

**Attributes**

PMCFGR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCG, bits [31:28]</td>
<td>This feature is not supported, so this field is RAZ. Reads as 0000. Access to this field is RO.</td>
</tr>
<tr>
<td>FZO, bit [21]</td>
<td>Freeze-on-overflow supported. Defined values are: 0b0 Freeze-on-overflow mechanism is not supported. PMCR_EL0.FZO is RES0. 0b1 Freeze-on-overflow mechanism is supported. PMCR_EL0.FZO is RW. FEAT_PMUv3p7 implements the functionality added by the value 0b1. From Armv8.7, if FEAT_PMUv3 is implemented, the only permitted value is 0b1.</td>
</tr>
<tr>
<td>UEN, bit [19]</td>
<td>User-mode Enable Register supported. PMUSERENR_EL0 is not visible in the external debug interface, so this bit is RAZ. Reads as 000. Access to this field is RO.</td>
</tr>
<tr>
<td>WT, bit [18]</td>
<td>This feature is not supported, so this bit is RAZ. Reads as 000. Access to this field is RO.</td>
</tr>
<tr>
<td>NA, bit [17]</td>
<td>This feature is not supported, so this bit is RAZ.</td>
</tr>
</tbody>
</table>
Reads as 0b0.
Access to this field is RO.

**EX, bit [16]**

Export supported. Value is IMPLEMENTATION DEFINED.

0b0  PMCR_EL0.X is RES0.
0b1  PMCR_EL0.X is read/write.

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

**CCD, bit [15]**

Cycle counter has prescale.
This is RES1 if AArch32 is supported, and RAZ otherwise.

0b0  PMCR_EL0.D is RES0.
0b1  PMCR_EL0.D is read/write.

**CC, bit [14]**

Dedicated cycle counter (counter 31) supported.
Reads as 0b1.
Access to this field is RO.

**SIZE, bits [13:8]**

Size of counters, minus one. This field defines the size of the largest counter implemented by the Performance Monitors Unit.
From Armv8, the largest counter is 64-bits, so the value of this field is 0b111111.
This field is used by software to determine the spacing of the counters in the memory-map. From Armv8, the counters are a doubleword-aligned addresses.
Reads as 0b111111.
Access to this field is RO.

**N, bits [7:0]**

Number of counters implemented in addition to the cycle counter, PMCCNTR_EL0. The maximum number of event counters is 31.
0x00  Only PMCCNTR_EL0 implemented.
0x01..0x1F  PMCCNTR_EL0 plus this number of event counters are implemented.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

### Accessing the PMCFGR:

#### Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCFGR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xE00</td>
<td>PMCFGR</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and AllowExternalPMUAccess() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
### PMCIDR0, Performance Monitors Component Identification Register 0

The PMCIDR0 characteristics are:

**Purpose**

Provides information to identify a Performance Monitor component.

For more information, see *About the Component Identification scheme on page K6-11699.*

**Configurations**

Implementation of this register is OPTIONAL.

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

**Attributes**

PMCIDR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRMBL_0</td>
<td>[7:0]</td>
<td>Preamble</td>
<td>0x0D</td>
</tr>
</tbody>
</table>

Accessing the PMCIDR0:

PMCIDR0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFF0</td>
<td>PMCIDR0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.10  PMCIDR1, Performance Monitors Component Identification Register 1

The PMCIDR1 characteristics are:

Purpose

Provides information to identify a Performance Monitor component. For more information, see About the Component Identification scheme on page K6-11699.

Configurations

Implementation of this register is OPTIONAL. If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain. This register is required for CoreSight compliance.

Attributes

PMCIDR1 is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-8</td>
<td>RES0</td>
<td></td>
</tr>
<tr>
<td>7-4</td>
<td>CLASS</td>
<td>0b1001</td>
</tr>
<tr>
<td>3-0</td>
<td>PRMBL_1</td>
<td>0b0000</td>
</tr>
</tbody>
</table>

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class.

0b1001  CoreSight component.

Other values are defined by the CoreSight Architecture. This field reads as 0x9.

PRMBL_1, bits [3:0]

Preamble. RAZ.

Reads as 0b0000. Access to this field is RO.

Accessing the PMCIDR1:

PMCIDR1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFF4</td>
<td>PMCIDR1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.11 PMCIDR2, Performance Monitors Component Identification Register 2

The PMCIDR2 characteristics are:

**Purpose**

Provides information to identify a Performance Monitor component.

For more information, see *About the Component Identification scheme on page K6-11699*.

**Configurations**

Implementation of this register is OPTIONAL.

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

**Attributes**

PMCIDR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:8]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRMBL_2, bits [7:0]</td>
<td>Preamble. Reads as 0x05. Access to this field is RO.</td>
</tr>
</tbody>
</table>

**Accessing the PMCIDR2:**

PMCIDR2 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFFF8</td>
<td>PMCIDR2</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.12 PMCIDR3, Performance Monitors Component Identification Register 3

The PMCIDR3 characteristics are:

**Purpose**

Provides information to identify a Performance Monitor component.

For more information, see *About the Component Identification scheme on page K6-11699*.

**Configurations**

Implementation of this register is OPTIONAL.

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

**Attributes**

PMCIDR3 is a 32-bit register.

**Field descriptions**

```
  31  8  7  6  5  4  3  2  1  0
   |   |   |   |   |   |   |   |
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td><strong>PRMBL_3</strong></td>
<td>Preamble</td>
</tr>
<tr>
<td></td>
<td>Reads as 0xB1</td>
</tr>
<tr>
<td></td>
<td>Access to this field is RO</td>
</tr>
</tbody>
</table>

**Accessing the PMCIDR3:**

PMCIDR3 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFFF</td>
<td>PMCIDR3</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.13 PMCID1SR, CONTEXTIDR_EL1 Sample Register

The PMCID1SR characteristics are:

**Purpose**
Contains the sampled value of CONTEXTIDR_EL1, captured on reading PMPCSR[31:0].

**Configurations**
PMCID1SR is in the Core power domain.
This register is present only when FEAT_PCSRv8p2 is implemented. Otherwise, direct accesses to PMCID1SR are RES0.

_____ Note _____
Before Armv8.2, the PC Sample-based Profiling Extension can be implemented in the external debug register space, as indicated by the value of EDDEVID.PCSample.

**Attributes**
PMCID1SR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>ContextIDR_EL1, bits [31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context ID. The value of CONTEXTIDR that is associated with the most recent PMPCSR sample. When the most recent PMPCSR sample is generated:</td>
</tr>
<tr>
<td>- If EL1 is using AArch64, then the Context ID is sampled from CONTEXTIDR_EL1.</td>
</tr>
<tr>
<td>- If EL1 is using AArch32, then the Context ID is sampled from CONTEXTIDR.</td>
</tr>
<tr>
<td>- If EL3 is implemented and is using AArch32, then CONTEXTIDR is a banked register and PMCID1SR samples the current banked copy of CONTEXTIDR for the Security state that is associated with the most recent PMPCSR sample.</td>
</tr>
</tbody>
</table>

Because the value written to PMCID1SR is an indirect read of CONTEXTIDR, it is constrained unpredictable whether PMCID1SR is set to the original or new value if PMPCSR samples:

- An instruction that writes to CONTEXTIDR.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Accessing the PMCID1SR:**

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see *Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN* on page H7-10354.

PMCID1SR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0x288</td>
<td>PMCID1SR</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.

PMCID1SR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0x228</td>
<td>PMCID1SR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.14 PMCID2SR, CONTEXTIDR_EL2 Sample Register

The PMCID2SR characteristics are:

Purpose

Contains the sampled value of CONTEXTIDR_EL2, captured on reading PMPCSR[31:0].

Configurations

PMCID2SR is in the Core power domain.

This register is present only when FEAT_PCSRv8p2 is implemented and EL2 is implemented. Otherwise, direct accesses to PMCID2SR are RES0.

——— Note ————

If FEAT_PCSRv8p2 is not implemented, the PC Sample-based Profiling Extension can be implemented in the external debug register space, as indicated by the value of EDDEVID.PCSample.

Attributes

PMCID2SR is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTEXTIDR_EL2</td>
<td></td>
</tr>
</tbody>
</table>

CONTEXTIDR_EL2, bits [31:0]

Context ID. The value of CONTEXTIDR_EL2 that is associated with the most recent PMPCSR sample. When the most recent PMPCSR sample is generated:

- If the PE is not executing at EL3, EL2 is using AArch64, and EL2 is enabled in the current Security state, then this field is set to the Context ID sampled from CONTEXTIDR_EL2.
- Otherwise, this field is set to an UNKNOWN value.

Because the value written to PMCID2SR is an indirect read of CONTEXTIDR_EL2, it is CONSTRAINED UNPREDICTABLE whether PMCID2SR is set to the original or new value if PMPCSR samples:

- An instruction that writes to CONTEXTIDR_EL2.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the PMCID2SR:

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN on page H7-10354.

PMCID2SR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0x22C</td>
<td>PMCID2SR</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.15 PMCNTENCLR_EL0, Performance Monitors Count Enable Clear register

The PMCNTENCLR_EL0 characteristics are:

**Purpose**

Disables the Cycle Count Register, PMCCNTR_EL0, and any implemented event counters PMEVCNTR<\(n\)>_EL0. Reading this register shows which counters are enabled.

**Configurations**

External register PMCNTENCLR_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMCNTENCLR_EL0[31:0].

External register PMCNTENCLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCNTENCLR[31:0].

PMCNTENCLR_EL0 is in the Core power domain.

**Attributes**

PMCNTENCLR_EL0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>C, bit [31]</th>
<th>PMCCNTR_EL0 disable bit. Disables the cycle counter register. Possible values are:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>When read, means the cycle counter is disabled. When written, has no effect.</td>
</tr>
<tr>
<td>0b1</td>
<td>When read, means the cycle counter is enabled. When written, disables the cycle</td>
</tr>
<tr>
<td></td>
<td>counter.</td>
</tr>
<tr>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On a Warm reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P&lt;(n)&gt;_bit [(n)], for (n = 30) to 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event counter disable bit for PMEVCNTR&lt;(n)&gt;_EL0.</td>
</tr>
<tr>
<td>IF PMCFG.]N is less than 31, bits [30:PMCFG.]N] are RAZ/WI.</td>
</tr>
<tr>
<td>0b0</td>
</tr>
<tr>
<td>0b1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Accessing the PMCNTENCLR_EL0:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCNTENCLR_EL0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xC20</td>
<td>PMCNTENCLR_EL0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.
### 5.3.16 PMCNTENSET_EL0, Performance Monitors Count Enable Set register

The PMCNTENSET_EL0 characteristics are:

**Purpose**

Enables the Cycle Count Register, PMCCNTR_EL0, and any implemented event counters PMEVCNTR<n>_EL0. Reading this register shows which counters are enabled.

**Configurations**

External register PMCNTENSET_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMCNTENSET_EL0[31:0].

External register PMCNTENSET_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCNTENSET[31:0].

PMCNTENSET_EL0 is in the Core power domain.

**Attributes**

PMCNTENSET_EL0 is a 32-bit register.

**Field descriptions**

C, bit [31]

PMCCNTR_EL0 enable bit. Enables the cycle counter register. Possible values are:

- \(0b0\) When read, means the cycle counter is disabled. When written, has no effect.
- \(0b1\) When read, means the cycle counter is enabled. When written, enables the cycle counter.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

P<n>, bit [n], for n = 30 to 0

Event counter enable bit for PMEVCNTR<n>_EL0.

If PMCFGR.N is less than 31, bits [30:PMCFGR.N] are RAZ/WI.

- \(0b0\) When read, means that PMEVCNTR<n>_EL0 is disabled. When written, has no effect.
- \(0b1\) When read, means that PMEVCNTR<n>_EL0 event counter is enabled. When written, enables PMEVCNTR<n>_EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
**Accessing the PMCNTENSET_EL0:**

--- **Note**

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCNTENSET_EL0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xC00</td>
<td>PMCNTENSET_EL0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.
I5.3.17 PMCR_EL0, Performance Monitors Control Register

The PMCR_EL0 characteristics are:

**Purpose**

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

**Configurations**

External register PMCR_EL0 bits [7:0] are architecturally mapped to AArch32 System register PMCR[7:0].

External register PMCR_EL0 bits [7:0] are architecturally mapped to AArch64 System register PMCR_EL0[7:0].

PMCR_EL0 is in the Core power domain.

This register is only partially mapped to the internal PMCR System register. An external agent must use other means to discover the information held in PMCR[31:11], such as accessing PMCFGR and the ID registers.

**Attributes**

PMCR_EL0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 31:11</td>
<td>Reserved, RAZ/WI. Hardware must implement this field as RAZ/WI. Software must not rely on the register reading as zero, and must use a read-modify-write sequence to write to the register.</td>
</tr>
<tr>
<td>Bit 10</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>
| FZO, bit 9 | Freeze-on-overflow. Stop event counters on overflow. When FEAT_PMUv3p7 is implemented: Freeze-on-overflow. Stop event counters on overflow. In the description of this field:  
  - If EL2 is implemented and is using AArch32, PMN is HDCR.HPMN.  
  - If EL2 is implemented and is using AArch64, PMN is MDCR_EL2.HPMN.  
  - If EL2 is not implemented, PMN is PMCR_EL0.N.  
  - 0b0 Do not freeze on overflow.  
  - 0b1 Event counter PMEVCNTR<n>_EL0 does not count when PMOVSCCLR_EL0[(PMN-1):0] is nonzero and n is in the range of affected event counters. If PMN is not 0, this field affects the operation of event counters in the range [0 .. (PMN-1)]. This field does not affect the operation of other event counters and PMCCNTR_EL0. The operation of this field applies even when EL2 is disabled in the current Security state. |
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bit [8]**

Reserved, RES0.

**LP, bit [7]**

When FEAT_PMUv3p5 is implemented:

Long event counter enable. Determines when unsigned overflow is recorded by an event counter overflow bit.

In the description of this field:

- If EL2 is implemented and is using AArch32, PMN is HDCR.HPMN.
- If EL2 is implemented and is using AArch64, PMN is MDCR_EL2.HPMN.
- If EL2 is not implemented, PMN is PMCR_EL0.N.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Event counter overflow on increment that causes unsigned overflow of PMEVCNTR&lt;n&gt;_EL0[31:0].</td>
</tr>
<tr>
<td>0b1</td>
<td>Event counter overflow on increment that causes unsigned overflow of PMEVCNTR&lt;n&gt;_EL0[63:0].</td>
</tr>
</tbody>
</table>

If PMN is not 0, this bit affects the operation of event counters in the range [0 .. (PMN-1)]. The field does not affect the operation of other event counters and PMCCNTR_EL0. The operation of this field applies even when EL2 is disabled in the current Security state. The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**LC, bit [6]**

When AArch32 is supported:

Long cycle counter enable. Determines when unsigned overflow is recorded by the cycle counter overflow bit.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[31:0].</td>
</tr>
<tr>
<td>0b1</td>
<td>Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[63:0].</td>
</tr>
</tbody>
</table>

Arm deprecates use of PMCR_EL0.LC = 0. The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES1.

**DP, bit [5]**

When EL3 is implemented or (FEAT_PMUv3p1 is implemented and EL2 is implemented):

Disable cycle counter when event counting is prohibited. The possible values of this bit are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Cycle counting by PMCCNTR_EL0 is not affected by this mechanism.</td>
</tr>
</tbody>
</table>
0b1  Cycle counting by PMCCNTR_EL0 is disabled in prohibited regions and when event counting is frozen:
    • If FEAT_PMUv3p1 is implemented, EL2 is implemented, and MDCR_EL2.HPMD is 1, then cycle counting by PMCCNTR_EL0 is disabled at EL2.
    • If FEAT_PMUv3p7 is implemented, EL3 is implemented and using AArch64, and MDCR_EL3.MPMX is 1, then cycle counting by PMCCNTR_EL0 is disabled at EL3.
    • If FEAT_PMUv3p7 is implemented and event counting is frozen by PMCR_EL0.FZO, then cycle counting by PMCCNTR_EL0 is disabled.
    • If EL3 is implemented, MDCR_EL3.SPME is 0, and either FEAT_PMUv3p7 is not implemented or MDCR_EL3.MPMX is 0, then cycle counting by PMCCNTR_EL0 is disabled at EL3 and in Secure state.
If MDCR_EL2.HPMN is not 0, this is when event counting by event counters in the range [0..(MDCR_EL2.HPMN-1)] is prohibited or frozen.
For more information, see Prohibiting event and cycle counting on page D11-5256.
The reset behavior of this field is:
    • On a Warm reset:
      — When the implementation only supports execution in AArch32 state, this field resets to 0.
      — Otherwise, this field resets to an architecturally UNKNOWN value.
    Otherwise:
      Reserved, RES0.

X, bit [4]
When the implementation includes a PMU event export bus:
Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.
0b0  Do not export events.
0b1  Export events where not prohibited.
This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device, for example to an OPTIONAL trace unit.
No events are exported when counting is prohibited.
This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.
The reset behavior of this field is:
    • When this register has an architecturally-defined reset value, if this field is implemented as an RW field it resets to:
      — A value that is architecturally UNKNOWN if the reset is into an Exception level that is using AArch64.
      — 0 if the reset is into an Exception level that is using AArch32.
    Otherwise:
      Reserved, RAZ/WI.

D, bit [3]
When AArch32 is supported:
Clock divider.
0b0  When enabled, PMCCNTR_EL0 counts every clock cycle.
0b1  When enabled, PMCCNTR_EL0 counts once every 64 clock cycles.
If PMCR_EL0.LC == 1, this bit is ignored and the cycle counter counts every clock cycle.
Arm deprecates use of PMCR_EL0.D = 1.

The reset behavior of this field is:

- When this register has an architecturally-defined reset value, if this field is implemented as an RW field it resets to:
  - A value that is architecturally UNKNOWN if the reset is into an Exception level that is using AArch64.
  - 0 if the reset is into an Exception level that is using AArch32.

**Otherwise:**

Reserved, RES0.

**C, bit [2]**

Cycle counter reset. The effects of writing to this bit are:

0b0  No action.
0b1  Reset PMCCNTR_EL0 to zero.

--- Note ---

Resetting PMCCNTR_EL0 does not change the cycle counter overflow bit. If FEAT_PMUv3p5 is implemented, the value of PMCR_EL0.LC is ignored, and bits [63:0] of the cycle counter are reset.

---

Access to this field is WO/RAZ.

**P, bit [1]**

Event counter reset. The effects of writing to this bit are:

0b0  No action.
0b1  Reset all event counters, not including PMCCNTR_EL0, to zero.

--- Note ---

Resetting the event counters does not change the event counter overflow bits. If FEAT_PMUv3p5 is implemented, the value of MDCR_EL2.HLP, or PMCR_EL0.LP is ignored and bits [63:0] of all affected event counters are reset.

---

Access to this field is WO/RAZ.

**E, bit [0]**

Enable.

In the description of this field:

- If EL2 is implemented and is using AArch32, PMN is HDCR.HPMN.
- If EL2 is implemented and is using AArch64, PMN is MDCR_EL2.HPMN.
- If EL2 is not implemented, PMN is PMCR_EL0.N.

0b0  PMCCNTR_EL0 is disabled and event counters PMEVCNTR<n>_EL0, where n is in the range of affected event counters, are disabled.
0b1  PMCCNTR_EL0 and event counters PMEVCNTR<n>_EL0, where n is in the range of affected event counters, are enabled by PMCNTENSET_EL0.

If PMN is not 0, this field affects the operation of event counters in the range [0 .. (PMN-1)].

This field does not affect the operation of other event counters.

The operation of this field applies even when EL2 is disabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0.
Accessing the PMCR_EL0:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMCR_EL0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xE04</td>
<td>PMCR_EL0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.
I5.3.18 PMDEVAF0, Performance Monitors Device Affinity register 0

The PMDEVAF0 characteristics are:

**Purpose**

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the Performance Monitor component relates to.

**Configurations**

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain. This register is required if the external interface to the PMU is implemented.

**Attributes**

PMDEVAF0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIDR_EL1lo</td>
<td>MPIDR_EL1 low half. Read-only copy of the low half of MPIDR_EL1, as seen from the highest implemented Exception level.</td>
</tr>
</tbody>
</table>

**Accessing the PMDEVAF0:**

PMDEVAF0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFA8</td>
<td>PMDEVAF0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.19 PMDEVAFF1, Performance Monitors Device Affinity register 1

The PMDEVAFF1 characteristics are:

**Purpose**

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the Performance Monitor component relates to.

**Configurations**

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required if the external interface to the PMU is implemented.

**Attributes**

PMDEVAFF1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field Description</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIDR_EL1hi</td>
<td>0xFAC</td>
</tr>
</tbody>
</table>

**MPIDR_EL1hi, bits [31:0]**

MPIDR_EL1 high half. Read-only copy of the high half of MPIDR_EL1, as seen from the highest implemented Exception level.

**Accessing the PMDEVAFF1:**

PMDEVAFF1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFAC</td>
<td>PMDEVAFF1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.20 PMDEVARCH, Performance Monitors Device Architecture register

The PMDEVARCH characteristics are:

**Purpose**
Identifies the programmers’ model architecture of the Performance Monitor component.

**Configurations**
If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

**Attributes**
PMDEVARCH is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARCHITECT</td>
<td>31:21</td>
<td>Defines the architecture of the component. For Performance Monitors, this is Arm Limited. Bits [31:28] are the JEP106 continuation code, 0x4. Bits [27:21] are the JEP106 ID code, 0x3B. Reads as 0b01000111011. Access to this field is RO.</td>
</tr>
<tr>
<td>PRESENT</td>
<td>20</td>
<td>Indicates that the DEVARCH is present. Reads as 0b1. Access to this field is RO.</td>
</tr>
<tr>
<td>REVISION</td>
<td>19:16</td>
<td>Defines the architecture revision. For architectures defined by Arm this is the minor revision. For Performance Monitors, the revision defined by Armv8 is 0x0. All other values are reserved. Reads as 0b0000. Access to this field is RO.</td>
</tr>
<tr>
<td>ARCHVER</td>
<td>15:12</td>
<td>Architecture Version. Defines the architecture version of the component. 0b0010 Performance Monitors Extension version 3, PMUv3. All other values are reserved. PMDEVARCH.ARCHVER and PMDEVARCH.ARCHPART are also defined as a single field, PMDEVARCH.ARCHID, so that PMDEVARCH.ARCHVER is PMDEVARCH.ARCHID[15:12]. Access to this field is RO.</td>
</tr>
<tr>
<td>ARCHPART</td>
<td>11:0</td>
<td>Architecture Part. Defines the architecture of the component. 0xA16 Armv8-A PE performance monitors.</td>
</tr>
</tbody>
</table>

01000111011 31 21
1 1 0 1 1 1 0 0 0 0 0 0 1 0 11 0 0
ARCHITECT  PRESENT  REVISION  ARCHVER

ARCHITECT, bits [31:21]

Defines the architecture of the component. For Performance Monitors, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, 0x4.

Bits [27:21] are the JEP106 ID code, 0x3B.

Reads as 0b01000111011.

Access to this field is RO.

PRESENT, bit [20]

Indicates that the DEVARCH is present.

Reads as 0b1.

Access to this field is RO.

REVISION, bits [19:16]

Defines the architecture revision. For architectures defined by Arm this is the minor revision.

For Performance Monitors, the revision defined by Armv8 is 0x0.

All other values are reserved.

Reads as 0b0000.

Access to this field is RO.

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component.

0b0010 Performance Monitors Extension version 3, PMUv3.

All other values are reserved.

PMDEVARCH.ARCHVER and PMDEVARCH.ARCHPART are also defined as a single field, PMDEVARCH.ARCHID, so that PMDEVARCH.ARCHVER is PMDEVARCH.ARCHID[15:12]. Access to this field is RO.

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

0xA16 Armv8-A PE performance monitors.
0xA26  From Armv8.8:
Armv8-A PE performance monitors, including the 64-bit programmers' model extension.

PMDEVARCH.ARCHVER and PMDEVARCH.ARCHPART are also defined as a single field, PMDEVARCH.ARCHID, so that PMDEVARCH.ARCHPART is PMDEVARCH.ARCHID[11:0]. This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Accessing the PMDEVARCH:

PMDEVARCH can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFBC</td>
<td>PMDEVARCH</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
### 5.3.21 PMDEVID, Performance Monitors Device ID register

The PMDEVID characteristics are:

**Purpose**

Provides information about features of the Performance Monitors implementation.

**Configurations**

- If `FEAT_DoPD` is implemented, this register is in the Core power domain.
- If `FEAT_DoPD` is not implemented, this register is in the Debug power domain.

This register is required from Armv8.2 and in any implementation that includes `FEAT_PCSRv8p2`. Otherwise, its location is RES0.

---

**Note**

Before Armv8.2, the PC Sample-based Profiling Extension can be implemented in the external debug register space, as indicated by the value of `EDDEVID.PCSample`.

---

**Attributes**

PMDEVID is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-4</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>4-0</td>
<td>PCSample</td>
</tr>
</tbody>
</table>

**Bits [31:4]**

Reserved, RES0.

**PCSample, bits [3:0]**

Indicates the level of PC Sample-based Profiling support using Performance Monitors registers.

- **0b0000**: PC Sample-based Profiling Extension is not implemented in the Performance Monitors register space.
- **0b0001**: PC Sample-based Profiling Extension is implemented in the Performance Monitors register space.

All other values are reserved.

`FEAT_PCSRv8p2` implements the functionality identified by the value `0b0001`.

**Accessing the PMDEVID:**

PMDEVID can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFC8</td>
<td>PMDEVID</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When `FEAT_DoPD` is not implemented or `IsCorePowered()` accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.22 PMDEVTYPE, Performance Monitors Device Type register

The PMDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PEs performance monitor interface.

Configurations

Implementation of this register is OPTIONAL.

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

Attributes

PMDEVTYPE is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Mask</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-8</td>
<td></td>
<td>Reserved, RES0</td>
<td>00000000</td>
</tr>
<tr>
<td>7-4</td>
<td></td>
<td>Subtype. Indicates this is a component within a PE.</td>
<td>0010</td>
</tr>
<tr>
<td>3-0</td>
<td></td>
<td>Major type. Indicates this is a performance monitor component.</td>
<td>0110</td>
</tr>
</tbody>
</table>

Accessing the PMDEVTYPE:

PMDEVTYPE can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFCC</td>
<td>PMDEVTYPE</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.23 PMEVCNTR<\(n\rangle\_EL0, Performance Monitors Event Count Registers, \(n = 0 - 30\)

The PMEVCNTR<\(n\rangle\_EL0 characteristics are:

**Purpose**

Holds event counter \(n\), which counts events, where \(n\) is 0 to 30.

**Configurations**

External register PMEVCNTR<\(n\rangle\_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMEVCNTR<\(n\rangle\_EL0[31:0].

External register PMEVCNTR<\(n\rangle\_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMEVCNTR<\(n\rangle\_EL0[31:0].

PMEVCNTR<\(n\rangle\_EL0 is in the Core power domain.

**Attributes**

PMEVCNTR<\(n\rangle\_EL0 is a:

- 64-bit register when FEAT_PMUv3p5 is implemented
- 32-bit register otherwise

**Field descriptions**

*When FEAT_PMUv3p5 is implemented:*

<table>
<thead>
<tr>
<th>Bit range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:0]</td>
<td>Event counter (n)</td>
</tr>
<tr>
<td>[31:0]</td>
<td>Event counter (n)</td>
</tr>
</tbody>
</table>

**Bits [63:0]**

Event counter \(n\). Value of event counter \(n\), where \(n\) is the number of this register and is a number from 0 to 30.

If the highest implemented Exception level is using AArch32, the optional external interface to the performance monitors is implemented, and the PMCR.LP and HDCR.HLP bits are RAZ/WI, then locations in the external interface to the performance monitors that map to PMEVCNTR<\(n\rangle\_EL0[63:32] return UNKNOWN values on reads.

If the implementation does not support AArch64, bits [63:32] of the event counters are not required to be implemented.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

<table>
<thead>
<tr>
<th>Bit range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>Event counter (n)</td>
</tr>
</tbody>
</table>

**Bits [31:0]**

Event counter \(n\). Value of event counter \(n\), where \(n\) is the number of this register and is a number from 0 to 30.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing the PMEVCNTR\<n>_EL0:**

External accesses to the performance monitors ignore the following controls:

  • PMUSERENR_EL0.
  • If implemented, MDCR_EL2:{TPM, TPMCR, HPMN}.
  • MDCR_EL3.TPM.

This means that all counters are accessible regardless of the current Exception level or privilege of the access.

If FEAT_PMUv3p5 is not implemented, when IsCorePowered(), DoubleLockStatus(), OSLockStatus() or !AllowExternalPMUAccess(), 32-bit accesses to 0x004+8\times n have a CONSTRAINED UNPREDICTABLE behavior.

**Note**

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMEVCNTR\<n>_EL0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0x000 + (8 * n)</td>
<td>PMEVCNTR&lt;n&gt;_EL0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

  • When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
  • When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
  • Otherwise accesses to this register generate an error response.
I5.3.24 PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n>_EL0 characteristics are:

**Purpose**

Configures event counter n, where n is 0 to 30.

**Configurations**

External register PMEVTYPER<n>_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMEVTYPER<n>_EL0[63:0].

External register PMEVTYPER<n>_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMEVTYPER<n>[31:0].

PMEVTYPER<n>_EL0 is in the Core power domain.

If event counter n is not implemented:

- When IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalPMUAccess(), accesses are RES0.
- Otherwise, it is CONSTRAINED UNPREDICTABLE whether accesses to this register are RES0 or generate an error response.

**Attributes**

PMEVTYPER<n>_EL0 is a 64-bit register.

**Field descriptions**

TC, bits [63:61]

*When FEAT_PMUv3_TH is implemented:*

Threshold Control. Defines the threshold function. In the description of this field, the value V is the value the event specified by PMEVTYPER<n>_EL0.evtCount would increment the counter by on a processor cycle if the threshold function is disabled. Comparisons treat V and this PMEVTYPER<n>_EL0.TH as unsigned integer values.

- **0b000** Not-equal. The counter increments by V on each processor cycle when V is not equal to PMEVTYPER<n>_EL0.TH. If PMEVTYPER<n>_EL0.TH is zero, the threshold function is disabled.
- **0b001** Not-equal, count. The counter increments by 1 on each processor cycle when V is not equal to PMEVTYPER<n>_EL0.TH.
- **0b010** Equals. The counter increments by V on each processor cycle when V is equal to PMEVTYPER<n>_EL0.TH.
- **0b011** Equals, count. The counter increments by 1 on each processor cycle when V is equal to PMEVTYPER<n>_EL0.TH.
- **0b100** Greater-than-or-equal. The counter increments by V on each processor cycle when V is PMEVTYPER<n>_EL0.TH or more.
- **0b101** Greater-than-or-equal, count. The counter increments by 1 on each processor cycle when V is PMEVTYPER<n>_EL0.TH or more.
0b110 Less-than. The counter increments by 1 on each processor cycle when V is less than PMEVTYPER<n>_EL0.TH.

0b111 Less-than, count. The counter increments by 1 on each processor cycle when V is less than PMEVTYPER<n>_EL0.TH.

The reset behavior of this field is:
- On a Warm reset:
  - When AArch32 is supported, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bits [60:44]
Reserved, RES0.

TH, bits [43:32]

When FEAT_PMUv3_TH is implemented:
Threshold value. Provides the unsigned value for the threshold function defined by PMEVTYPER<n>_EL0.TC.

If PMEVTYPER<n>_EL0.TC is 0b000 and PMEVTYPER<n>_EL0.TH is zero, then the threshold function is disabled.

If PMMIR.THWIDTH is less than 12, then bits PMEVTYPER<n>_EL0.TH[11:PMMIR.THWIDTH] are RES0. This accounts for the behavior when writing a value greater-than-or-equal-to 2^{PMMIR.THWIDTH}.

The reset behavior of this field is:
- On a Warm reset:
  - When AArch32 is supported, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

P, bit [31]
Privileged filtering bit. Controls counting in EL1.

If EL3 is implemented, then counting in Non-secure EL1 is further controlled by the PMEVTYPER<n>_EL0.NSK bit.

If FEAT_RME is implemented, then counting in Realm EL1 is further controlled by the PMEVTYPER<n>_EL0.RLK bit.

0b0 Count events in EL1.

0b1 Do not count events in EL1.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [30]
User filtering bit. Controls counting in EL0.

If EL3 is implemented, then counting in Non-secure EL0 is further controlled by the PMEVTYPER<n>_EL0.NSU bit.

If FEAT_RME is implemented, then counting in Realm EL0 is further controlled by the PMEVTYPER<n>_EL0.RLU bit.

0b0 Count events in EL0.

0b1 Do not count events in EL0.
The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

**NSK, bit [29]**

*When EL3 is implemented:*

Non-secure EL1 (kernel) modes filtering bit. Controls counting in Non-secure EL1.

If the value of this bit is equal to the value of the PMEVTYPER<\(n\)>_EL0.P bit, events in Non-secure EL1 are counted.

Otherwise, events in Non-secure EL1 are not counted.

The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

**NSU, bit [28]**

*When EL3 is implemented:*

Non-secure EL0 (Unprivileged) filtering bit. Controls counting in Non-secure EL0.

If the value of this bit is equal to the value of the PMEVTYPER<\(n\)>_EL0.U bit, events in Non-secure EL0 are counted.

Otherwise, events in Non-secure EL0 are not counted.

The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

**NSH, bit [27]**

*When EL2 is implemented:*

EL2 (Hypervisor) filtering bit. Controls counting in EL2.

If FEAT_SEL2 and EL3 are implemented, counting in Secure EL2 is further controlled by the PMEVTYPER<\(n\)>_EL0.SH bit.

If FEAT_RME is implemented, then counting in Realm EL2 is further controlled by the PMEVTYPER<\(n\)>_EL0.RLH bit.

0b0  Do not count events in EL2.
0b1  Count events in EL2.

The reset behavior of this field is:
  • On a Warm reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

**M, bit [26]**

*When EL3 is implemented:*

EL3 filtering bit.

If the value of this bit is equal to the value of the PMEVTYPER<\(n\)>_EL0.P bit, events in EL3 are counted.

Otherwise, events in EL3 are not counted.

Most applications can ignore this field and set its value to 0b0.
Note
This field is not visible in the AArch32 PMEVTYPER<n> System register.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

MT, bit [25]
When (FEAT_MTPMU is implemented and enabled) or an IMPLEMENTATION DEFINED multi-threaded PMU Extension is implemented:
Multithreading.
0b0 Count events only on controlling PE.
0b1 Count events from any PE with the same affinity at level 1 and above as this PE.

Note
- When the lowest level of affinity consists of logical PEs that are implemented using a multi-threading type approach, an implementation is described as multi-threaded. That is, the performance of PEs at the lowest affinity level is highly interdependent.
- Events from a different thread of a multithreaded implementation are not Attributable to the thread counting the event.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

SH, bit [24]
When FEAT_SEL2 is implemented and EL3 is implemented:
Secure EL2 filtering.
If the value of this bit is not equal to the value of the PMEVTYPER<n>_EL0.NSH bit, events in Secure EL2 are counted.
Otherwise, events in Secure EL2 are not counted.

Note
This field is not visible in the AArch32 PMEVTYPER<n> System register.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

T, bit [23]
When FEAT_TME is implemented:
Transactional state filtering bit. Controls counting in Transactional state.
0b0 This bit has no effect on the filtering of events.
0b1 Do not count events in Transactional state.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Otherwise:
    Reserved, RES0.

RLK, bit [22]

When FEAT_RME is implemented:
    Realm EL1 (kernel) filtering bit. Controls counting in Realm EL1.
    If the value of this bit is equal to the value of the PMEVTYPER<\text{n}>_EL0.P bit, events in Realm
    EL1 are counted.
    Otherwise, events in Realm EL1 are not counted.
    The reset behavior of this field is:
    • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
    Reserved, RES0.

RLU, bit [21]

When FEAT_RME is implemented:
    Realm EL0 (unprivileged) filtering bit. Controls counting in Realm EL0.
    If the value of this bit is equal to the value of the PMEVTYPER<\text{n}>_EL0.U bit, events in Realm
    EL0 are counted.
    Otherwise, events in Realm EL0 are not counted.
    The reset behavior of this field is:
    • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
    Reserved, RES0.

RLH, bit [20]

When FEAT_RME is implemented:
    Realm EL2 filtering bit. Controls counting in Realm EL2.
    If the value of this bit is not equal to the value of the PMEVTYPER<\text{n}>_EL0.NSH bit, events in Realm
    EL2 are counted.
    Otherwise, events in Realm EL2 are not counted.
    The reset behavior of this field is:
    • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
    Reserved, RES0.

Bits [19:16]

Reserved, RES0.

evtCount[15:10], bits [15:10]

When FEAT_PMUv3p1 is implemented:
    Extension to evtCount[9:0]. For more information, see evtCount[9:0].
    The reset behavior of this field is:
    • On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
    Reserved, RES0.
**evtCount[9:0], bits [9:0]**

Event to count. The event number of the event that is counted by event counter PMEVCNTR<\text{n}>_EL0.

Software must program this field with an event that is supported by the PE being programmed. The ranges of event numbers allocated to each type of event are shown in Table D11-7 on page D11-5275.

If FEAT_PMUv3p8 is implemented and PMEVTYPER<\text{n}>_EL0(evtCount is programmed to an event that is reserved or not supported by the PE, no events are counted and the value returned by a direct or external read of the PMEVTYPER<\text{n}>_EL0(evtCount field is the value written to the field.

--- Note ---

Arm recommends this behavior for all implementations of FEAT_PMUv3.

---

Otherwise, if PMEVTYPER<\text{n}>_EL0(evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the value written:

- For the range 0x0000 to 0x003F, no events are counted and the value returned by a direct or external read of the PMEVTYPER<\text{n}>_EL0(evtCount field is the value written to the field.
- If FEAT_PMUv3p1 is implemented, for the range 0x4000 to 0x403F, no events are counted and the value returned by a direct or external read of the PMEVTYPER<\text{n}>_EL0(evtCount field is the value written to the field.
- For other values, it is UNPREDICTABLE what event, if any, is counted, and the value returned by a direct or external read of the PMEVTYPER<\text{n}>_EL0(evtCount field is UNKNOWN.

--- Note ---

UNPREDICTABLE means the event must not expose privileged information.

---

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing the PMEVTYPER<\text{n}>_EL0:**

If FEAT_PMUv3_TH or FEAT_PMUv3p8 is implemented, bits [63:32] of this interface are accessible at offset 0xA00 + (4*n). Otherwise accesses at this offset are IMPLEMENTATION DEFINED.

--- Note ---

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

---

PMEVTYPER<\text{n}>_EL0[31:0] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0x400 + (4 * n)</td>
<td>PMEVTYPER&lt;\text{n}&gt;_EL0</td>
<td>31:0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to PMEVTYPER<\text{n}>_EL0[31:0] are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to PMEVTYPER<\text{n}>_EL0[31:0] are RW.
- Otherwise accesses to PMEVTYPER<\text{n}>_EL0[31:0] generate an error response.
PMEVTYPER<n>_EL0[63:32] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xA00 + (4 * n)</td>
<td>PMEVTYPER&lt;n&gt;_EL0</td>
<td>63:32</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to PMEVTYPER<n>_EL0[63:32] are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to PMEVTYPER<n>_EL0[63:32] are RW.
- Otherwise accesses to PMEVTYPER<n>_EL0[63:32] generate an error response.
PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear register

The PMINTENCLR_EL1 characteristics are:

**Purpose**
Disables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR_EL0, and the event counters PMEVCNTR<\(\leq n\)> EL0. Reading the register shows which overflow interrupt requests are enabled.

**Configurations**
External register PMINTENCLR_EL1 bits [31:0] are architecturally mapped to AArch64 System register PMINTENCLR_EL1[31:0].
External register PMINTENCLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register PMINTENCLR[31:0].
PMINTENCLR_EL1 is in the Core power domain.

**Attributes**
PMINTENCLR_EL1 is a 32-bit register.

**Field descriptions**

C, bit [31]
PMCCNTR_EL0 overflow interrupt request disable bit. Possible values are:

0b0 When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

0b1 When read, means the cycle counter overflow interrupt request is enabled. When written, disables the cycle count overflow interrupt request.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P<\(\leq n\)>, bit [n], for \(n = 30\) to \(0\)
Event counter overflow interrupt request disable bit for PMEVCNTR<\(\leq n\)> EL0.
If PMCFGR.N is less than 31, bits [30:PMCFGR.N] are RAZ/WI.

0b0 When read, means that the PMEVCNTR<\(\leq n\)> EL0 event counter interrupt request is disabled. When written, has no effect.

0b1 When read, means that the PMEVCNTR<\(\leq n\)> EL0 event counter interrupt request is enabled. When written, disables the PMEVCNTR<\(\leq n\)> EL0 interrupt request.

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.
Accessing the PMINTENCLR_EL1:

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMINTENCLR_EL1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xC60</td>
<td>PMINTENCLR_EL1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.
I5.3.26  PMI宁SENSET_EL1, Performance Monitors Interrupt Enable Set register

The PMI宁SENSET_EL1 characteristics are:

**Purpose**

Enables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNT_R_EL0, and the event counters PMEVCNT_R<n>_EL0. Reading the register shows which overflow interrupt requests are enabled.

**Configurations**

External register PMI宁SENSET_EL1 bits [31:0] are architecturally mapped to AArch64 System register PMI宁SENSET_EL1[31:0].

External register PMI宁SENSET_EL1 bits [31:0] are architecturally mapped to AArch32 System register PMI宁SENSET[31:0].

PMI宁SENSET_EL1 is in the Core power domain.

**Attributes**

PMI宁SENSET_EL1 is a 32-bit register.

**Field descriptions**

C, bit [31]
PMCCNT_R_EL0 overflow interrupt request enable bit. Possible values are:

**0b0**  When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

**0b1**  When read, means the cycle counter overflow interrupt request is enabled. When written, enables the cycle count overflow interrupt request.

The reset behavior of this field is:

•  On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

P<n>, bit [n], for n = 30 to 0

Event counter overflow interrupt request enable bit for PMEVCNT_R<n>_EL0.

If PMCFGR.N is less than 31, bits [30:PMCFGR.N] are RAZ/WI.

**0b0**  When read, means that the PMEVCNT_R<n>_EL0 event counter interrupt request is disabled. When written, has no effect.

**0b1**  When read, means that the PMEVCNT_R<n>_EL0 event counter interrupt request is enabled. When written, enables the PMEVCNT_R<n>_EL0 interrupt request.

The reset behavior of this field is:

•  On a Warm reset, this field resets to an architecturally **UNKNOWN** value.
**Accessing the PMINTENSET_EL1:**

--- **Note** ---

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMINTENSET_EL1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xC40</td>
<td>PMINTENSET_EL1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.
### I5.3.27 PMITCTRL, Performance Monitors Integration mode Control register

The PMITCTRL characteristics are:

**Purpose**

Enables the Performance Monitors to switch from default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

**Configurations**

It is IMPLEMENTATION DEFINED whether PMITCTRL is implemented in the Core power domain or in the Debug power domain.

Implementation of this register is OPTIONAL.

**Attributes**

PMITCTRL is a 32-bit register.

**Field descriptions**

![Field descriptions diagram]

**Bits [31:1]**

Reserved, RES0.

**IME, bit [0]**

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection. The integration mode behavior is IMPLEMENTATION DEFINED.

- **0b0** Normal operation.
- **0b1** Integration mode enabled.

The reset behavior of this field is:

- The following resets apply:
  - If the register is implemented in the Core power domain:
    - On a Cold reset, this field resets to 0.
    - On an External debug reset, the value of this field is unchanged.
    - On a Warm reset, the value of this field is unchanged.
  - If the register is implemented in the External debug power domain:
    - On a Cold reset, the value of this field is unchanged.
    - On an External debug reset, this field resets to 0.
    - On a Warm reset, the value of this field is unchanged.

**Accessing the PMITCTRL:**

PMITCTRL can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xF00</td>
<td>PMITCTRL</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register are IMPDEF.
### I5.3.28 PMLAR, Performance Monitors Lock Access Register

The PMLAR characteristics are:

**Purpose**

Allows or disallows access to the Performance Monitors registers through a memory-mapped interface.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Performance Monitors registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Performance Monitors registers. It does not, and cannot, prevent all accidental or malicious damage.

**Configurations**

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE in the Core power domain.

If FEAT_DoPD is not implemented, this register is in the Debug power domain.

Software uses PMLAR to set or clear the lock, and PMLSR to check the current status of the lock.

**Attributes**

PMLAR is a 32-bit register.

**Field descriptions**

*When Software Lock is implemented:*

```
+---+---+
|   |   |
| #1 | #0 |
+---+---+
  KEY

KEY, bits [31:0]

Lock Access control. Writing the key value 0xC5ACCE55 to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

**Otherwise:**

```
+---+---+
|   |   |
| #1 | #0 |
+---+---+
  RES0

Otherwise

Bits [31:0]

Reserved, RES0.

**Accessing the PMLAR:**

PMLAR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFB0</td>
<td>PMLAR</td>
</tr>
</tbody>
</table>
This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are WO.
- Otherwise accesses to this register generate an error response.
I5.3.29 PMLSR, Performance Monitors Lock Status Register

The PMLSR characteristics are:

**Purpose**

 Indicates the current status of the software lock for Performance Monitors registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Performance Monitors registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Performance Monitors registers. It does not, and cannot, prevent all accidental or malicious damage.

**Configurations**

- If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE in the Core power domain.
- If FEAT_DoPD is not implemented, this register is in the Debug power domain.

Software uses PMLAR to set or clear the lock, and PMLSR to check the current status of the lock.

**Attributes**

PMLSR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:3]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>nTT, bit [2]</td>
<td>Not thirty-two bit access required. Reads as 0b0. Access to this field is RO.</td>
</tr>
<tr>
<td>SLK, bit [1]</td>
<td><strong>When Software Lock is implemented and FEAT_DoPD is not implemented:</strong> Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when Software Lock is not implemented, this field is RES0. For memory-mapped accesses when Software Lock is implemented, possible values of this field are: 0b0 Lock clear. Writes are permitted to this component's registers. 0b1 Lock set. Writes to this component's registers are ignored, and reads have no side effects. The reset behavior of this field is: • On an External debug reset, this field resets to 1. <strong>Otherwise:</strong> Reserved, RAZ.</td>
</tr>
</tbody>
</table>
SLI, bit [0]

Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ. For memory-mapped accesses, the value of this field is IMPLEMENTATION DEFINED.

Permitted values are:

- 0b0 Software Lock not implemented or not memory-mapped access.
- 0b1 Software Lock implemented and memory-mapped access.

Accessing the PMLSR:

PMLSR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFB4</td>
<td>PMLSR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.30 **PMMIR, Performance Monitors Machine Identification Register**

The PMMIR characteristics are:

**Purpose**

Describes Performance Monitors parameters specific to the implementation.

**Configurations**

PMMIR is in the Core power domain.

This register is present only when FEAT_PMUv3p4 is implemented. Otherwise, direct accesses to PMMIR are RES0.

**Attributes**

PMMIR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31–24</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>23–20</td>
<td>THWIDTH</td>
<td></td>
</tr>
<tr>
<td>19–16</td>
<td>BUS_SLOTS</td>
<td></td>
</tr>
<tr>
<td>15–8</td>
<td>SLOTS</td>
<td></td>
</tr>
<tr>
<td>7–0</td>
<td>BUS_WIDTH</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:24]**

Reserved, RES0.

**THWIDTH, bits [23:20]**

Indicates implementation of the FEAT_PMUv3_TH feature, and, if implemented, the size of the PMEVTYPER<\text{n}>_EL0.TH field.

- 0b0000: FEAT_PMUv3_TH is not implemented.
- 0b0001: 1 bit. PMEVTYPER<\text{n}>_EL0.TH[11:1] are RES0.
- 0b0010: 2 bits. PMEVTYPER<\text{n}>_EL0.TH[11:2] are RES0.
- 0b0011: 3 bits. PMEVTYPER<\text{n}>_EL0.TH[11:3] are RES0.
- 0b0100: 4 bits. PMEVTYPER<\text{n}>_EL0.TH[11:4] are RES0.
- 0b0101: 5 bits. PMEVTYPER<\text{n}>_EL0.TH[11:5] are RES0.
- 0b0110: 6 bits. PMEVTYPER<\text{n}>_EL0.TH[11:6] are RES0.
- 0b0111: 7 bits. PMEVTYPER<\text{n}>_EL0.TH[11:7] are RES0.
- 0b1000: 8 bits. PMEVTYPER<\text{n}>_EL0.TH[11:8] are RES0.
- 0b1001: 9 bits. PMEVTYPER<\text{n}>_EL0.TH[11:9] are RES0.
- 0b1010: 10 bits. PMEVTYPER<\text{n}>_EL0.TH[11:10] are RES0.
- 0b1011: 11 bits. PMEVTYPER<\text{n}>_EL0.TH[11] is RES0.
- 0b1100: 12 bits.

All other values are reserved.

If FEAT_PMUv3_TH is not implemented, this field is zero.

Otherwise, the largest value that can be written to PMEVTYPER<\text{n}>_EL0.TH is $2^{(\text{PMMIR.THWIDTH})}$ minus one.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.
BUS_WIDTH, bits [19:16]

Bus width. Indicates the number of bytes each BUS_ACCESS event relates to. Encoded as Log2(number of bytes), plus one.

- 0b0000  The information is not available.
- 0b0011  Four bytes.
- 0b0100  8 bytes.
- 0b0101  16 bytes.
- 0b0110  32 bytes.
- 0b0111  64 bytes.
- 0b1000  128 bytes.
- 0b1001  256 bytes.
- 0b1010  512 bytes.
- 0b1011  1024 bytes.
- 0b1100  2048 bytes.

All other values are reserved.

Each transfer is up to this number of bytes. An access might be smaller than the bus width.

When this field is nonzero, each access counted by BUS_ACCESS is at most BUS_WIDTH bytes. An implementation might treat a wide bus as multiple narrower buses, such that a wide access on the bus increments the BUS_ACCESS counter by more than one.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

BUS_SLOTS, bits [15:8]

Bus count. The largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle.

When this field is nonzero, the largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle is BUS_SLOTS.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

SLOTS, bits [7:0]

Operation width. The largest value by which the STALL_SLOT event might increment by in a single cycle. If the STALL_SLOT event is not implemented, this field might read as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing the PMMIR:

If the Core power domain is off or in a low-power state, access on this interface returns an Error.

PMMIR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xE40</td>
<td>PMMIR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), or DoubleLockStatus(), or OSLockStatus() or !AllowExternalPMUAccess() accesses to this register generate an error response.
• Otherwise accesses to this register are RO.
I5.3.31 PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear register

The PMOVSCLR_EL0 characteristics are:

**Purpose**

Contains the state of the overflow bit for the Cycle Count Register, PMCCNTR_EL0, and each of the implemented event counters PMEVCNTR\(<n>_EL0. Writing to this register clears these bits.

**Configurations**

External register PMOVSCLR_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMOVSCLR_EL0[31:0].

External register PMOVSCLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMOVSR[31:0].

PMOVSCLR_EL0 is in the Core power domain.

**Attributes**

PMOVSCLR_EL0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**C, bit [31]**

Cycle counter overflow clear bit.

- **0b0** When read, means the cycle counter has not overflowed since this bit was last cleared. When written, has no effect.
- **0b1** When read, means the cycle counter has overflowed since this bit was last cleared. When written, clears the cycle counter overflow bit to 0.

**PMCR_EL0.LC** controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

**P<n>, bit [n], for n = 30 to 0**

Event counter overflow clear bit for PMEVCNTR\(<n>_EL0.

- **If PMCFGR.N is less than 31**, bits [30:PMCFGR.N] are RAZ/WI.
- **0b0** When read, means that PMEVCNTR\(<n>_EL0 has not overflowed since this bit was last cleared. When written, has no effect.
- **0b1** When read, means that PMEVCNTR\(<n>_EL0 has overflowed since this bit was last cleared. When written, clears the PMEVCNTR\(<n>_EL0 overflow bit to 0.**
If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<n>_EL0[31:0] or unsigned overflow of PMEVCNTR<n>_EL0[63:0].

The reset behavior of this field is:
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing the PMOVSCLR_EL0:**

--- Note ---
SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMOVSCLR_EL0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xC80</td>
<td>PMOVSCLR_EL0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.
I5.3.32 **PMOVSET_EL0, Performance Monitors Overflow Flag Status Set register**

The PMOVSET_EL0 characteristics are:

**Purpose**

Sets the state of the overflow bit for the Cycle Count Register, PMCCNTR_EL0, and each of the implemented event counters PMEVCNTR<\(n\)>_EL0.

**Configurations**

External register PMOVSET_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMOVSET_EL0[31:0].

External register PMOVSET_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMOVSET[31:0].

PMOVSET_EL0 is in the Core power domain.

**Attributes**

PMOVSET_EL0 is a 32-bit register.

**Field descriptions**

|-------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

**C, bit [31]**

Cycle counter overflow set bit.

- **0b0** When read, means the cycle counter has not overflowed since this bit was last cleared. When written, has no effect.

- **0b1** When read, means the cycle counter has overflowed since this bit was last cleared. When written, sets the cycle counter overflow bit to 1.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**P<\(n\)>, bit [n], for n = 30 to 0**

Event counter overflow set bit for PMEVCNTR<\(n\)>_EL0.

If PMCFGR.N is less than 31, bits [30:PMCFGR.N] are RAZ/WI.

- **0b0** When read, means that PMEVCNTR<\(n\)>_EL0 has not overflowed since this bit was last cleared. When written, has no effect.

- **0b1** When read, means that PMEVCNTR<\(n\)>_EL0 has overflowed since this bit was last cleared. When written, sets the PMEVCNTR<\(n\)>_EL0 overflow bit to 1.
If FEAT_PMUV3p5 is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<n>_EL0[31:0] or unsigned overflow of PMEVCNTR<n>_EL0[63:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Accessing the PMOVSSET_EL0:**

--- Note ---

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

PMOVSSET_EL0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xCC0</td>
<td>PMOVSSET_EL0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and SoftwareLockStatus() accesses to this register are RO.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are RW.
- Otherwise accesses to this register generate an error response.
### 5.3.33 PMPCSR, Program Counter Sample Register

The PMPCSR characteristics are:

**Purpose**

Holds a sampled instruction address value.

**Configurations**

PMPCSR is in the Core power domain.

This register is present only when FEAT_PCSRv8p2 is implemented. Otherwise, direct accesses to PMPCSR are RES0.

---

**Note**

Before Armv8.2, the PC Sample-based Profiling Extension can be implemented in the external debug register space, as indicated by the value of EDDEVID.PCSample.

Support for 64-bit atomic reads is IMPLEMENTATION DEFINED. If 64-bit atomic reads are implemented, a 64-bit read of PMPCSR has the same side-effect as a 32-bit read of PMCSR[31:0] followed by a 32-bit read of PMPCSR[63:32], returning the combined value. For example, if the PE is in Debug state then a 64-bit atomic read returns bits[31:0] == 0xFFFFFFFF and bits[63:32] UNKNOWN.

**Attributes**

PMPCSR is a 64-bit register.

**Field descriptions**

PMPCSR is a 64-bit register.

#### NS, bit [63]

*When FEAT_RME is implemented:*

Together with the NSE field, indicates the Security state that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

<table>
<thead>
<tr>
<th>NSE</th>
<th>NS</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>0b0</td>
<td>Secure.</td>
</tr>
<tr>
<td>0b0</td>
<td>0b1</td>
<td>Non-secure.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b0</td>
<td>Root.</td>
</tr>
<tr>
<td>0b1</td>
<td>0b1</td>
<td>Realm.</td>
</tr>
</tbody>
</table>

*Otherwise:*

Non-secure state sample. Indicates the Security state that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

If EL3 is not implemented, this bit indicates the Effective value of SCR.NS.

0b0 Sample is from Secure state.
Sample is from Non-secure state.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

EL, bits [62:61]
Exception level status sample. Indicates the Exception level that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

- 0b00  Sample is from EL0.
- 0b01  Sample is from EL1.
- 0b10  Sample is from EL2.
- 0b11  Sample is from EL3.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

T, bit [60]
When FEAT_TME is implemented:
Transactional state of the sample. Indicates the Transactional state that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

- 0b0  Sample is from Non-transactional state.
- 0b1  Sample is from Transactional state.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

NSE, bit [59]
When FEAT_RME is implemented:
Together with the NS field, indicates the Security state that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.
For a description of the values derived by evaluating NS and NSE together, see PMPCSR.NS.

Otherwise:
Reserved, RES0.

Bits [58:56]
Reserved, RES0.

PCSample[55:32], bits [55:32]
Bits[55:32] of the sampled instruction address value. The translation regime that PMPCSR samples can be determined from PMPCSR.{NS,EL}.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

PCSample[31:0], bits [31:0]
Bits[31:0] of the sampled instruction address value.
PMPCSR[31:0] reads as 0xFFFFFFFF when any of the following are true:
• The PE is in Debug state.
• PC Sample-based profiling is prohibited.
If a branch instruction has retired since the PE left reset state, then the first read of PMPCSR[31:0] is permitted but not required to return 0xFFFFFFFF.
PMPCSR[31:0] reads as an unknown value when any of the following are true:

- The PE is in reset state.
- No branch instruction has retired since the PE left reset state, Debug state, or a state where PC Sample-based Profiling is prohibited.
- No branch instruction has retired since the last read of PMPCSR[31:0].

For the cases where a read of PMPCSR[31:0] returns \(0xFFFFFFFF\) or an unknown value, the read has the side-effect of setting PMPCSR[63:32], PMCID1SR, PMCID2SR, and PMVIDSR to unknown values.

Otherwise, a read of PMPCSR[31:0] returns bits [31:0] of the sampled instruction address value and has the side-effect of indirectly writing to PMPCSR[63:32], PMCID1SR, PMCID2SR, and PMVIDSR. The translation regime that PMPCSR samples can be determined from PMPCSR.{NS,EL}.

For a read of PMPCSR[31:0] from the memory-mapped interface, if PMLSR.SLK == 1, meaning the optional Software Lock is locked, then the side-effect of the access does not occur and PMPCSR[63:32], PMCID1SR, PMCID2SR, and PMVIDSR are unchanged.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally unknown value.

### Accessing the PMPCSR:

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register unknown, see "Permitted behavior that might make the PC Sample-based profiling registers unknown" on page H7-10354.

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0x200</td>
<td>PMPCSR</td>
<td>31:0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to PMPCSR[31:0] are RO.
- Otherwise accesses to PMPCSR[31:0] generate an error response.

PMPCSR[31:0] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0x220</td>
<td>PMPCSR</td>
<td>31:0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to PMPCSR[31:0] are RO.
- Otherwise accesses to PMPCSR[31:0] generate an error response.
PMPCSR[63:32] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0x204</td>
<td>PMPCSR</td>
<td>63:32</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:
- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to PMPCSR[63:32] are RO.
- Otherwise accesses to PMPCSR[63:32] generate an error response.

PMPCSR[63:32] can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0x224</td>
<td>PMPCSR</td>
<td>63:32</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:
- When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to PMPCSR[63:32] are RO.
- Otherwise accesses to PMPCSR[63:32] generate an error response.
I5.3.34 PMPIDR0, Performance Monitors Peripheral Identification Register 0

The PMPIDR0 characteristics are:

**Purpose**

Provides information to identify a Performance Monitor component.

For more information, see *About the Peripheral identification scheme on page K6-11696.*

**Configurations**

Implementation of this register is OPTIONAL.

If `FEAT_DoPD` is implemented, this register is in the Core power domain. If `FEAT_DoPD` is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

**Attributes**

PMPIDR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit位</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-8</td>
<td>RES0</td>
</tr>
<tr>
<td>7-0</td>
<td>PART_0</td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**PART_0, bits [7:0]**

Part number, least significant byte.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Accessing the PMPIDR0:**

PMPIDR0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFE0</td>
<td>PMPIDR0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When `FEAT_DoPD` is not implemented or `IsCorePowered()` accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.35  PMPIDR1, Performance Monitors Peripheral Identification Register 1

The PMPIDR1 characteristics are:

**Purpose**

Provides information to identify a Performance Monitor component.
For more information, see *About the Peripheral identification scheme* on page K6-11696.

**Configurations**

Implementation of this register is *OPTIONAL*.
If *FEAT_DoPD* is implemented, this register is in the Core power domain. If *FEAT_DoPD* is not implemented, this register is in the Debug power domain.
This register is required for CoreSight compliance.

**Attributes**

PMPIDR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:8]</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>[7:4]</td>
<td>DES_0</td>
</tr>
<tr>
<td>[3:0]</td>
<td>PART_1</td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**DES_0, bits [7:4]**

Designer, least significant nibble of JEP106 ID code. For Arm Limited, this field is 0b1011.
This field has an **IMPLEMENTATION DEFINED** value.
Access to this field is RO.

**PART_1, bits [3:0]**

Part number, most significant nibble.
This field has an **IMPLEMENTATION DEFINED** value.
Access to this field is RO.

**Accessing the PMPIDR1:**

PMPIDR1 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFE4</td>
<td>PMPIDR1</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When *FEAT_DoPD* is not implemented or `IsCorePowered()` accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.36  **PMPIDR2, Performance Monitors Peripheral Identification Register 2**

The PMPIDR2 characteristics are:

**Purpose**

Provides information to identify a Performance Monitor component.

For more information, see *About the Peripheral identification scheme* on page K6-11696.

**Configurations**

Implementation of this register is OPTIONAL.

If `FEAT_DoPD` is implemented, this register is in the Core power domain. If `FEAT_DoPD` is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

**Attributes**

PMPIDR2 is a 32-bit register.

**Field descriptions**

```
+---------+---------+
| 31      | 8       |
| [31:8]  | RES0    |
| 7       | 4       |
| [7:4]   | REVISION|
| 3       | 2       |
| [3:2]   | DES_1   |
| 1       | JEDC    |
```

**Bits [31:8]**

Reserved, RES0.

**REVISION, bits [7:4]**

Part major revision. Parts can also use this field to extend Part number to 16-bits.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**JEDEC, bit [3]**

Indicates a JEP106 identity code is used.

Reads as 0b1.

Access to this field is RO.

**DES_1, bits [2:0]**

Designer, most significant bits of JEP106 ID code. For Arm Limited, this field is 0b11.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Accessing the PMPIDR2:**

PMPIDR2 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFE8</td>
<td>PMPIDR2</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When `FEAT_DoPD` is not implemented or `IsCorePowered()` accesses to this register are RO.
• Otherwise accesses to this register generate an error response.
### 15.3.37 PMPIDR3, Performance Monitors Peripheral Identification Register 3

The PMPIDR3 characteristics are:

**Purpose**

Provides information to identify a Performance Monitor component. For more information, see *About the Peripheral identification scheme on page K6-11696.*

**Configurations**

Implementation of this register is OPTIONAL.

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

**Attributes**

PMPIDR3 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits [31:8]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>REVAND, bits [7:4]</td>
<td>Part minor revision. Parts using PMPIDR2.REVISION as an extension to the Part number must use this field as a major revision number. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.</td>
</tr>
<tr>
<td>CMOD, bits [3:0]</td>
<td>Customer modified. Indicates someone other than the Designer has modified the component. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.</td>
</tr>
</tbody>
</table>

**Accessing the PMPIDR3:**

PMPIDR3 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFEC</td>
<td>PMPIDR3</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.38 PMPIDR4, Performance Monitors Peripheral Identification Register 4

The PMPIDR4 characteristics are:

**Purpose**
Provides information to identify a Performance Monitor component.
For more information, see *About the Peripheral identification scheme* on page K6-11696.

**Configurations**
Implementation of this register is OPTIONAL.
If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.
This register is required for CoreSight compliance.

**Attributes**
PMPIDR4 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-8</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>7-4</td>
<td>SIZE, bits [7:4]</td>
<td>0b0000</td>
</tr>
<tr>
<td>3-0</td>
<td>DES_2, bits [3:0]</td>
<td>0b0100</td>
</tr>
</tbody>
</table>

**Bits [31:8]**
Reserved, RES0.

**SIZE, bits [7:4]**
Size of the component. $\log_2$ of the number of 4KB pages from the start of the component to the end of the component ID registers.
Reads as 0b0000.
Access to this field is RO.

**DES_2, bits [3:0]**
Designer, JEP106 continuation code, least significant nibble. For Arm Limited, this field is 0b0100.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

**Accessing the PMPIDR4:**

PMPIDR4 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xFD0</td>
<td>PMPIDR4</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:
- When FEAT_DoPD is not implemented or IsCorePowered() accesses to this register are RO.
- Otherwise accesses to this register generate an error response.
I5.3.39 **PMSWINC_EL0, Performance Monitors Software Increment register**

The PMSWINC_EL0 characteristics are:

**Purpose**

Increments a counter that is configured to count the Software increment event, event 0x00. For more information, see SW_INCR.

**Configurations**

External register PMSWINC_EL0 bits [31:0] are architecturally mapped to AArch64 System register PMSWINC_EL0[31:0].

External register PMSWINC_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMSWINC[31:0].

PMSWINC_EL0 is in the Core power domain.

Implementation of this register is OPTIONAL.

If this register is implemented, use of it is deprecated.

If 1 is written to bit [n] from the external debug interface, it is CONSTRAINED UNPREDICTABLE whether or not a SW_INCR event is created for counter n. This is consistent with not implementing the register in the external debug interface.

**Attributes**

PMSWINC_EL0 is a 32-bit register.

**Field descriptions**

Bit [31]

Reserved, RES0.

**P<n>, bit [n], for n = 30 to 0**

Event counter software increment bit for PMEVCNTR<n>_EL0.

If PMCFGR.N is less than 31, bits [30:PMCFGR.N] are WI.

0b0 No action. The write to this bit is ignored.

0b1 It is CONSTRAINED UNPREDICTABLE whether a SW_INCR event is generated for event counter n.

**Accessing the PMSWINC_EL0:**

--- Note ---

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.
PMSWINC_EL0 can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0xCA0</td>
<td>PMSWINC_EL0</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are WI.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalPMUAccess() and !SoftwareLockStatus() accesses to this register are WO.
- Otherwise accesses to this register generate an error response.
I5.3.40 PMVIDSR, VMID Sample Register

The PMVIDSR characteristics are:

**Purpose**
Contains the sampled VMID value that is captured on reading PMPCSR[31:0].

**Configurations**
PMVIDSR is in the Core power domain.

This register is present only when FEAT_PCSRv8p2 is implemented and EL2 is implemented. Otherwise, direct accesses to PMVIDSR are RES0.

--- Note ---
Before Armv8.2, the PC Sample-based Profiling Extension can be implemented in the external debug register space, as indicated by the value of EDDEVID.PCSample.

---

**Attributes**
PMVIDSR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-16</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>15-8</td>
<td>VMID[15:8]</td>
</tr>
<tr>
<td>7-0</td>
<td>VMID</td>
</tr>
</tbody>
</table>

**Bits [31:16]**
Reserved, RES0.

**VMID[15:8], bits [15:8]**

*When FEAT_VMID16 is implemented:*

Extension to VMID[7:0]. For more information, see VMID[7:0].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

**VMID, bits [7:0]**

VMID sample. The VMID associated with the most recent PMPCSR sample. When the most recent PMPCSR sample was generated:

- This field is set to an UNKNOWN value if any of the following apply:
  - EL2 is disabled in the current Security state.
  - The PE is executing at EL2.
  - EL2 is enabled in the current Security state, the PE is executing at EL0, EL2 is using AArch64, HCR_EL2.E2H == 1, and HCR_EL2.TGE == 1.

- Otherwise:
  - If EL2 is using AArch64 and either FEAT_VMID16 is not implemented or VTCR_EL2.VS is 1, this field is set to VTTBR_EL2.VMID.
  - If EL2 is using AArch64, FEAT_VMID16 is implemented, and VTCR_EL2.VS is 0, PMVIDSR.VMID[7:0] is set to VTTBR_EL2.VMID[7:0] and PMVIDSR.VMID[15:8] is RES0.
  - If EL2 is using AArch32, this field is set to VTTBR.VMID.
Because the value written to PMVIDSR is an indirect read of the VMID value, it is CONSTRAINED
UNPREDICTABLE whether PMVIDSR is set to the original or new value if PMPCSR samples:
• An instruction that writes to the VMID value.
• The next Context synchronization event.
• Any instruction executed between these two instructions.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Accessing the PMVIDSR:**

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see
Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN on page H7-10354.

PMVIDSR can be accessed through the external debug interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMU</td>
<td>0x20C</td>
<td>PMVIDSR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

• When IsCorePowered(), !DoubleLockStatus() and !OSLockStatus() accesses to this register are RO.
• Otherwise accesses to this register generate an error response.
## 5.4 External Activity Monitors Extension registers summary

The memory-mapped interface to the Activity Monitors Extension registers provides read-only access to:
- Read-only copies of the Activity Monitors Extension System registers, with the exception of AMUSERENR.
- An implementation identification register, AMIIDR.
- If they are implemented, the OPTIONAL Activity Monitors CoreSight and ID registers.

The locations of the registers are defined as offsets from a base address. The base address of the memory-mapped view must be aligned to a 4KB boundary, but is otherwise IMPLEMENTATION DEFINED. Activity Monitors external register views defines this memory map.

### 5.4.1 Activity Monitors external register views

Table I5-3 on page I5-10836 shows the external view of the Activity Monitors registers. All implemented registers are RO. Offsets within the 4KB region not defined in this table are RAZ/WI.

Each entry in the Name column links to the register description in Activity Monitors external register descriptions on page I5-10838, and:
- If the System register? on page I5-10836 column of the table shows that the register is a System register, the memory-mapped interface provides a view of the System register described in:
  - Activity Monitors registers on page D17-6816, for the AArch64 System register.
  - Activity Monitors registers on page G8-10047, for the AArch32 System register.
- Otherwise, the register is accessible only using the external memory-mapped interface.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>System register?</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMEVCTR&lt;n&gt;</td>
<td>Activity Monitor Event Counter registers 0</td>
<td>Yes</td>
<td>0x000+8n</td>
</tr>
<tr>
<td>AMEVCTR&lt;n&gt;</td>
<td>Activity Monitor Event Counter registers 1</td>
<td>Yes</td>
<td>0x100+8n</td>
</tr>
<tr>
<td>AMEVTYPR&lt;n&gt;</td>
<td>Activity Monitor Event Type registers 0</td>
<td>Yes</td>
<td>0x400+4n</td>
</tr>
<tr>
<td>AMEVTYPR&lt;n&gt;</td>
<td>Activity Monitor Event Type registers 1</td>
<td>Yes</td>
<td>0x480+4n</td>
</tr>
<tr>
<td>AMCNTENSET0</td>
<td>Activity Monitors Counter Enable Set register 0</td>
<td>Yes</td>
<td>0xC00</td>
</tr>
<tr>
<td>AMCNTENSET1</td>
<td>Activity Monitors Counter Enable Set register 1</td>
<td>Yes</td>
<td>0xC04</td>
</tr>
<tr>
<td>AMCNTENCCLR0</td>
<td>Activity Monitors Counter Enable Clear register 0</td>
<td>Yes</td>
<td>0xC20</td>
</tr>
<tr>
<td>AMCNTENCCLR1</td>
<td>Activity Monitors Counter Enable Clear register 1</td>
<td>Yes</td>
<td>0xC24</td>
</tr>
<tr>
<td>AMCGCR</td>
<td>Activity Monitors Counter Group Configuration Register</td>
<td>Yes</td>
<td>0xCE0</td>
</tr>
<tr>
<td>AMCFGR</td>
<td>Activity Monitors Configuration Register</td>
<td>Yes</td>
<td>0xE00</td>
</tr>
<tr>
<td>AMCR</td>
<td>Activity Monitors Control Register</td>
<td>Yes</td>
<td>0xE04</td>
</tr>
<tr>
<td>AMIIDR</td>
<td>Activity Monitors Implementation Identification Register</td>
<td>No</td>
<td>0xE08</td>
</tr>
<tr>
<td>AMDEVAFF0*a</td>
<td>Device Affinity registers</td>
<td>No</td>
<td>0xF08</td>
</tr>
<tr>
<td>AMDEVAFF1*a</td>
<td></td>
<td>No</td>
<td>0xFAC</td>
</tr>
<tr>
<td>AMDEVARCH*a</td>
<td>Device Architecture register</td>
<td>No</td>
<td>0xFBC</td>
</tr>
<tr>
<td>AMDEVTYPE*a</td>
<td>Device Type register</td>
<td>No</td>
<td>0xFCC</td>
</tr>
</tbody>
</table>
### Table I5-3 Activity Monitors external register views (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>System register?</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMPIDR4a</td>
<td>Peripheral ID registers</td>
<td>No</td>
<td>0xFD0</td>
</tr>
<tr>
<td>AMPIDR0a</td>
<td></td>
<td>No</td>
<td>0xFE0</td>
</tr>
<tr>
<td>AMPIDR1a</td>
<td></td>
<td>No</td>
<td>0xFE4</td>
</tr>
<tr>
<td>AMPIDR2a</td>
<td></td>
<td>No</td>
<td>0xFE8</td>
</tr>
<tr>
<td>AMPIDR3a</td>
<td></td>
<td>No</td>
<td>0xFEC</td>
</tr>
<tr>
<td>AMCIDR0a</td>
<td>Component ID registers</td>
<td>No</td>
<td>0xFF0</td>
</tr>
<tr>
<td>AMCIDR1a</td>
<td></td>
<td>No</td>
<td>0xFF4</td>
</tr>
<tr>
<td>AMCIDR2a</td>
<td></td>
<td>No</td>
<td>0xFF8</td>
</tr>
<tr>
<td>AMCIDR3a</td>
<td></td>
<td>No</td>
<td>0xFFC</td>
</tr>
</tbody>
</table>

a. CoreSight interface registers, see *Management registers and CoreSight compliance on page K6-11688.*
I5.5 Activity Monitors external register descriptions

This section lists the external Activity Monitors registers.
I5.5.1 AMCFGR, Activity Monitors Configuration Register

The AMCFGR characteristics are:

**Purpose**

Global configuration register for the activity monitors.
Provides information on supported features, the number of counter groups implemented, the total number of activity monitor event counters implemented, and the size of the counters. AMCFGR is applicable to both the architected and the auxiliary counter groups.

**Configurations**

External register AMCFGR bits [31:0] are architecturally mapped to AArch64 System register AMCFGR_EL0[31:0].
External register AMCFGR bits [31:0] are architecturally mapped to AArch32 System register AMCFGR[31:0].
The power domain of AMCFGR is IMPLEMENTATION DEFINED.
This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCFGR are RES0.

**Attributes**

AMCFGR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCG</td>
<td>Defines the number of counter groups. The number of implemented counter groups is [AMCFGR.NCG + 1]. If the number of implemented auxiliary activity monitor event counters is zero, this field has a value of 0b0000. Otherwise, this field has a value of 0b0001. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.</td>
</tr>
<tr>
<td>Bits [27:25]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>HDBG, bit [24]</td>
<td>Halt-on-debug supported. This feature must be supported, and so this bit is 0b1. 0b0 AMCR.HDBG is RES0. 0b1 AMCR.HDBG is read/write. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.</td>
</tr>
<tr>
<td>Bits [23:14]</td>
<td>Reserved, RAZ.</td>
</tr>
</tbody>
</table>
SIZE, bits [13:8]

Defines the size of activity monitor event counters.

The size of the activity monitor event counters implemented by the Activity Monitors Extension is [AMCFGR.SIZE + 1].

The counters are 64-bit.

Note
Software also uses this field to determine the spacing of counters in the memory-map. The counters are at doubleword-aligned addresses.

Reads as 0b111111.
Access to this field is RO.

N, bits [7:0]

Defines the number of activity monitor event counters.

The total number of counters implemented in all groups by the Activity Monitors Extension is [AMCFGR.N + 1].

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Accessing the AMCFGR:

AMCFGR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xE0</td>
<td>AMCFGR</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
### I5.5.2 AMCGCR, Activity Monitors Counter Group Configuration Register

The AMCGCR characteristics are:

**Purpose**

Provides information on the number of activity monitor event counters implemented within each counter group.

**Configurations**

External register AMCGCR bits [31:0] are architecturally mapped to AArch64 System register `AMCGCR_EL0[31:0]`.

External register AMCGCR bits [31:0] are architecturally mapped to AArch32 System register `AMCGCR[31:0]`.

The power domain of AMCGCR is IMPLEMENTATION DEFINED.

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCGCR are RES0.

**Attributes**

AMCGCR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-16</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
<tr>
<td>15-8</td>
<td>CG1NC, Counter Group 1 Number of Counters</td>
<td>00000100</td>
</tr>
<tr>
<td>7-0</td>
<td>CG0NC, Counter Group 0 Number of Counters</td>
<td>00000010</td>
</tr>
</tbody>
</table>

**Bits [31:16]**

Reserved, RES0.

**CG1NC, bits [15:8]**

Counter Group 1 Number of Counters. The number of counters in the auxiliary counter group.

In an implementation that includes FEAT_AMUv1, the permitted range of values is 0 to 16.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**CG0NC, bits [7:0]**

Counter Group 0 Number of Counters. The number of counters in the architected counter group.

Reads as 0x04.

Access to this field is RO.

**Accessing the AMCGCR:**

AMCGCR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xCE0</td>
<td>AMCGCR</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.5.3 AMCIDR0, Activity Monitors Component Identification Register 0

The AMCIDR0 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Component Identification scheme on page K6-11699.

Configurations

The power domain of AMCIDR0 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when FEAT_AMUv1 is implemented.

Attributes

AMCIDR0 is a 32-bit register.

Field descriptions

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>PRMBL_0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Preamble.

Reads as 0x0D.

Access to this field is RO.

Accessing the AMCIDR0:

AMCIDR0 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xFF0</td>
<td>AMCIDR0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
5.5.4 AMCIDR1, Activity Monitors Component Identification Register 1

The AMCIDR1 characteristics are:

**Purpose**

Provides information to identify an activity monitors component.

For more information, see *About the Component Identification scheme* on page K6-11699.

**Configurations**

The power domain of AMCIDR1 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when FEAT_AMUv1 is implemented.

**Attributes**

AMCIDR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Component Offset Instance</th>
<th>0xFF4</th>
<th>AMCIDR1</th>
</tr>
</thead>
</table>

**Bits [31:8]**

Reserved, RES0.

**CLASS, bits [7:4]**

Component class.

0b1001  CoreSight component.

Other values are defined by the CoreSight Architecture.

This field reads as 0x9.

**PRMBL_1, bits [3:0]**

Preamble.

Reads as 0b0000.

Access to this field is RO.

**Accessing the AMCIDR1:**

AMCIDR1 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xFF4</td>
<td>AMCIDR1</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.5.5   AMCIDR2, Activity Monitors Component Identification Register 2

The AMCIDR2 characteristics are:

**Purpose**

Provides information to identify an activity monitors component.

For more information, see *About the Component Identification scheme on page K6-11699*.

**Configurations**

The power domain of AMCIDR2 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when FEAT_AMUv1 is implemented.

**Attributes**

AMCIDR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:8]</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>[7:0]</td>
<td>PRMBL_2, bits [7:0]</td>
</tr>
<tr>
<td></td>
<td>Preamble</td>
</tr>
<tr>
<td></td>
<td>Reads as 0x05</td>
</tr>
<tr>
<td></td>
<td>Access to this field is RO</td>
</tr>
</tbody>
</table>

**Accessing the AMCIDR2:**

AMCIDR2 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xFF8</td>
<td>AMCIDR2</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.5.6  AMCIDR3, Activity Monitors Component Identification Register 3

The AMCIDR3 characteristics are:

**Purpose**

Provides information to identify an activity monitors component.

For more information, see *About the Component Identification scheme on page* K6-11699.

**Configurations**

The power domain of AMCIDR3 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when FEAT_AMUv1 is implemented.

**Attributes**

AMCIDR3 is a 32-bit register.

**Field descriptions**

| Bits [31:8] | Reserved, RES0 |
| Bit 7 | PRMBL_3, bits [7:0] |

Preamble.

Reads as 0xB1.

Access to this field is RO.

**Accessing the AMCIDR3:**

AMCIDR3 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xFFC</td>
<td>AMCIDR3</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
### I5.5.7 AMCNTENCLR0, Activity Monitors Count Enable Clear Register 0

The AMCNTENCLR0 characteristics are:

**Purpose**

Disable control bits for the architected activity monitors event counters, AMEVCNTR0<n>.

**Configurations**

External register AMCNTENCLR0 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENCLR0_EL0[31:0].

External register AMCNTENCLR0 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENCLR0[31:0].

The power domain of AMCNTENCLR0 is IMPLEMENTATION DEFINED.

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCNTENCLR0 are RES0.

**Attributes**

AMCNTENCLR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-16</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>15-4</td>
<td>Reserved, RAZ/WI.</td>
</tr>
<tr>
<td>3-0</td>
<td>P&lt;n&gt;, bit [n], for n = 3 to 0</td>
</tr>
</tbody>
</table>

**P<n>, bit [n], for n = 3 to 0**

Activity monitor event counter disable bit for AMEVCNTR0<n>.

**Note**

AMCGCR.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

- **0b0** When read, means that AMEVCNTR0<n> is disabled.
- **0b1** When read, means that AMEVCNTR0<n> is enabled.

The reset behavior of this field is:

- On an AMU reset, this field resets to 0.
Accessing the AMCNTENCLR0:

AMCNTENCLR0 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xC20</td>
<td>AMCNTENCLR0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.5.8 AMCNTENCLR1, Activity Monitors Count Enable Clear Register 1

The AMCNTENCLR1 characteristics are:

**Purpose**

Disable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>.

**Configurations**

External register AMCNTENCLR1 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENCLR1_EL0[31:0].

External register AMCNTENCLR1 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENCLR1[31:0].

The power domain of AMCNTENCLR1 is **IMPLEMENTATION DEFINED**.

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCNTENCLR1 are RES0.

**Attributes**

AMCNTENCLR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-16</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>15-0</td>
<td>P&lt;n&gt;, bit [n], for n = 15 to 0</td>
</tr>
<tr>
<td></td>
<td>Activity monitor event counter disable bit for AMEVCNTR1&lt;n&gt;.</td>
</tr>
<tr>
<td></td>
<td>When N is less than 16, bits [15:N] are RAZ, where N is the value in AMCGCR.CG1NC.</td>
</tr>
<tr>
<td></td>
<td>Possible values of each bit are:</td>
</tr>
<tr>
<td></td>
<td>0b0 When read, means that AMEVCNTR1&lt;n&gt; is disabled.</td>
</tr>
<tr>
<td></td>
<td>0b1 When read, means that AMEVCNTR1&lt;n&gt; is enabled.</td>
</tr>
<tr>
<td></td>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>• On an AMU reset, this field resets to 0.</td>
</tr>
</tbody>
</table>

**Accessing the AMCNTENCLR1:**

If the number of auxiliary activity monitor event counters implemented is zero, reads of AMCNTENCLR1 are RAZ. Software must treat reserved accesses as RES0. See *Access requirements for reserved and unallocated registers* on page I1-10720.

--- **Note** ---

The number of auxiliary activity monitor event counters implemented is zero exactly when AMCFGR.NCG == 0b0000.
AMCNTENCLR1 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xC24</td>
<td>AMCNTENCLR1</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.5.9 AMCNTENSET0, Activity Monitors Count Enable Set Register 0

The AMCNTENSET0 characteristics are:

**Purpose**

Enable control bits for the architected activity monitors event counters, AMEVCNTR0<n>.

**Configurations**

External register AMCNTENSET0 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENSET0_EL0[31:0].

External register AMCNTENSET0 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENSET0[31:0].

The power domain of AMCNTENSET0 is IMPLEMENTATION DEFINED.

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCNTENSET0 are RES0.

**Attributes**

AMCNTENSET0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>RAZ/WI</th>
<th>P3</th>
<th>P2</th>
<th>P1</th>
<th>P0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:16]**

Reserved, RES0.

**Bits [15:4]**

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

**P<n>, bit [n], for n = 3 to 0**

Activity monitor event counter enable bit for AMEVCNTR0<n>.

**Note**

AMCGCR.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

| 0b0  | When read, means that AMEVCNTR0<n> is disabled. |
| 0b1  | When read, means that AMEVCNTR0<n> is enabled. |

The reset behavior of this field is:

- On an AMU reset, this field resets to 0.
Accessing the AMCNTENSET0:

AMCNTENSET0 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xC00</td>
<td>AMCNTENSET0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
### AMCNTENSET1, Activity Monitors Count Enable Set Register 1

The AMCNTENSET1 characteristics are:

**Purpose**

Enable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<\text{n}>.

**Configurations**

External register AMCNTENSET1 bits [31:0] are architecturally mapped to AArch64 System register AMCNTENSET1_EL0[31:0].

External register AMCNTENSET1 bits [31:0] are architecturally mapped to AArch32 System register AMCNTENSET1[31:0].

The power domain of AMCNTENSET1 is IMPLEMENTATION DEFINED.

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMCNTENSET1 are RES0.

**Attributes**

AMCNTENSET1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0 (Reserved)</td>
</tr>
<tr>
<td>15-16</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>8</td>
<td>P15</td>
</tr>
<tr>
<td>7</td>
<td>P14</td>
</tr>
<tr>
<td>6</td>
<td>P13</td>
</tr>
<tr>
<td>5</td>
<td>P12</td>
</tr>
<tr>
<td>4</td>
<td>P11</td>
</tr>
<tr>
<td>3</td>
<td>P10</td>
</tr>
<tr>
<td>2</td>
<td>P9</td>
</tr>
<tr>
<td>1</td>
<td>P8</td>
</tr>
<tr>
<td>0</td>
<td>P7</td>
</tr>
</tbody>
</table>

**Bits [31:16]**

Reserved, RES0.

**P<\text{n}>, bit [\text{n}], for \text{n} = 15 to 0**

Activity monitor event counter enable bit for AMEVCNTR1<\text{n}>.

When \( N \) is less than 16, bits [15:16] are RAZ, where \( N \) is the value in AMCGCR.GC1NC.

Possible values of each bit are:

0b0  When read, means that AMEVCNTR1<\text{n}> is disabled.

0b1  When read, means that AMEVCNTR1<\text{n}> is enabled.

The reset behavior of this field is:

- On an AMU reset, this field resets to 0.

**Accessing the AMCNTENSET1:**

If the number of auxiliary activity monitor event counters implemented is zero, reads of AMCNTENSET1 are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers on page I1-10720.

_____ **Note** _____

The number of auxiliary activity monitor counters implemented is zero exactly when AMCFGR.NCG == 0b0000.
AMCNTENSET1 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xC04</td>
<td>AMCNTENSET1</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
5.5.11 AMCR, Activity Monitors Control Register

The AMCR characteristics are:

**Purpose**
Global control register for the activity monitors implementation. AMCR is applicable to both the
architected and the auxiliary counter groups.

**Configurations**

External register AMCR bits [31:0] are architecturally mapped to AArch64 System register
AMCR_EL0[31:0].

External register AMCR bits [31:0] are architecturally mapped to AArch32 System register
AMCR[31:0].

The power domain of AMCR is IMPLEMENTATION DEFINED.

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to
AMCR are RES0.

**Attributes**

AMCR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-11</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>10</td>
<td>HDBG, bit [10]</td>
</tr>
<tr>
<td>9-0</td>
<td>Reserved, RES0</td>
</tr>
</tbody>
</table>

**Bits [31:11]**

Reserved, RES0.

**HDBG, bit [10]**

This bit controls whether activity monitor counting is halted when the PE is halted in Debug state.

- **0b0**: Activity monitors do not halt counting when the PE is halted in Debug state.
- **0b1**: Activity monitors halt counting when the PE is halted in Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

**Bits [9:0]**

Reserved, RES0.

**Accessing the AMCR:**

AMCR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xE04</td>
<td>AMCR</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.5.12 AMDEVAFF0, Activity Monitors Device Affinity Register 0

The AMDEVAFF0 characteristics are:

**Purpose**

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the AMU component relates to.

**Configurations**

The power domain of AMDEVAFF0 is IMPLEMENTATION DEFINED. Implementation of this register is OPTIONAL. This register is present only when FEAT_AMUv1 is implemented.

**Attributes**

AMDEVAFF0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIDR_EL1lo</td>
<td>Read-only copy of the low half of MPIDR_EL1, as seen from the highest implemented Exception level.</td>
</tr>
</tbody>
</table>

**Accessing the AMDEVAFF0:**

AMDEVAFF0 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xFA8</td>
<td>AMDEVAFF0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.5.13 AMDEVAFF1, Activity Monitors Device Affinity Register 1

The AMDEVAFF1 characteristics are:

**Purpose**

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the AMU component relates to.

**Configurations**

The power domain of AMDEVAFF1 is IMPLEMENTATION DEFINED. Implementation of this register is OPTIONAL. This register is present only when FEAT_AMUv1 is implemented.

**Attributes**

AMDEVAFF1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>MPIDR_EL1hi</td>
</tr>
</tbody>
</table>

**MPIDR_EL1hi, bits [31:0]**

MPIDR_EL1 high half. Read-only copy of the high half of MPIDR_EL1, as seen from the highest implemented Exception level.

**Accessing the AMDEVAFF1:**

AMDEVAFF1 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xFAC</td>
<td>AMDEVAFF1</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.5.14 AMDEVARCH, Activity Monitors Device Architecture Register

The AMDEVARCH characteristics are:

**Purpose**

Identifies the programmers’ model architecture of the AMU component.

**Configurations**

The power domain of AMDEVARCH is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when FEAT_AMUv1 is implemented.

**Attributes**

AMDEVARCH is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARCHITECT</td>
<td>[31:21]</td>
<td>Defines the architecture of the component. For AMU, this is Arm Limited.</td>
</tr>
<tr>
<td>PRESENT</td>
<td>[20]</td>
<td>Indicates that the DEVARCH is present. Reads as 0.</td>
</tr>
<tr>
<td>REVISION</td>
<td>[19:16]</td>
<td>Defines the architecture revision. For architectures defined by Arm this is the minor revision. 0x0000 Architecture revision is AMUv1. All other values are reserved. Access to this field is RO.</td>
</tr>
<tr>
<td>ARCHID</td>
<td>[15:0]</td>
<td>Defines this part to be an AMU component. For architectures defined by Arm this is further subdivided. For AMU:</td>
</tr>
</tbody>
</table>

- Bits [15:12] are the architecture version, 0x0.
- Bits [11:0] are the architecture part number, 0xA66.

This corresponds to AMU architecture version AMUv1. Reads as 0xA66. Access to this field is RO.
Accessing the AMDEVARCH:

AMDEVARCH can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xFBC</td>
<td>AMDEVARCH</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.5.15 AMDEVTYPE, Activity Monitors Device Type Register

The AMDEVTYPE characteristics are:

**Purpose**

Indicates to a debugger that this component is part of a PE’s performance monitor interface.

**Configurations**

The power domain of AMDEVTYPE is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when FEAT_AMUv1 is implemented.

**Attributes**

AMDEVTYPE is a 32-bit register.

**Field descriptions**

| Bits [31:8] | Reserved, RES0. |
| 0b0001 | Component within a PE. |
| Access to this field is RO. |

| 0b0110 | Performance monitor component |
| Access to this field is RO. |

**Accessing the AMDEVTYPE:**

AMDEVTYPE can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xFCC</td>
<td>AMDEVTYPE</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
### AMEVCNTR0\(<n>\), Activity Monitors Event Counter Registers 0, \(n = 0 - 3\)

The AMEVCNTR0\(<n>\) characteristics are:

**Purpose**

Provides access to the architected activity monitor event counters.

**Configurations**

External register AMEVCNTR0\(<n>\) bits [63:0] are architecturally mapped to AArch64 System register AMEVCNTR0\(<n>\)_EL0[63:0].

External register AMEVCNTR0\(<n>\) bits [63:0] are architecturally mapped to AArch32 System register AMEVCNTR0\(<n>\).[63:0].

The power domain of AMEVCNTR0\(<n>\) is IMPLEMENTATION DEFINED.

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMEVCNTR0\(<n>\) are RES0.

**Attributes**

AMEVCNTR0\(<n>\) is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field Description</th>
<th>Offset in AArch64</th>
<th>Offset in AArch32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACNT, bits [63:0]</td>
<td>0x000 + (8 * (n))</td>
<td>0x000 + (8 * (n))</td>
</tr>
</tbody>
</table>

**ACNT, bits [63:0]**

Architected activity monitor event counter \(n\).

Value of architected activity monitor event counter \(n\), where \(n\) is the number of this register and is a number from 0 to 3.

The reset behavior of this field is:

- On an AMU reset, this field resets to 0.

**Accessing the AMEVCNTR0\(<n>\):**

If \(<n>\) is greater than or equal to the number of architected activity monitor event counters, reads of AMEVCNTR0\(<n>\) are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers on page I1-10720.

**Note**

AMECGCR.CG0NC identifies the number of architected activity monitor event counters.

AMEVCNTR0\(<n>\)[31:0] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0x000 + (8 * (n))</td>
<td>AMEVCNTR0(&lt;n&gt;)</td>
<td>31:0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
AMEVCNTR0<\n>[63:32] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0x004 + (8 * n)</td>
<td>AMEVCNTR0&lt;\n&gt;</td>
<td>63:32</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
### I5.5.17 AMEVCNTR1<\(n\)>, Activity Monitors Event Counter Registers 1, \(n = 0 - 15\)

The AMEVCNTR1<\(n\)> characteristics are:

**Purpose**

Provides access to the auxiliary activity monitor event counters.

**Configurations**

External register AMEVCNTR1<\(n\)> bits [63:0] are architecturally mapped to AArch64 System register AMEVCNTR1<\(n\>EL0[63:0].

External register AMEVCNTR1<\(n\)> bits [63:0] are architecturally mapped to AArch32 System register AMEVCNTR1<\(n\>[63:0].

The power domain of AMEVCNTR1<\(n\)> is IMPLEMENTATION DEFINED.

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMEVCNTR1<\(n\)> are RES0.

**Attributes**

AMEVCNTR1<\(n\)> is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>ACNT</td>
</tr>
<tr>
<td>31-0</td>
<td>ACNT</td>
</tr>
</tbody>
</table>

**ACNT, bits [63:0]**

Auxiliary activity monitor event counter \(n\).

Value of auxiliary activity monitor event counter \(n\), where \(n\) is the number of this register and is a number from 0 to 15.

The reset behavior of this field is:
- On an AMU reset, this field resets to 0.

**Accessing the AMEVCNTR1<\(n\>):**

If \(<n\> is greater than or equal to the number of auxiliary activity monitor event counters, reads of AMEVCNTR1<\(n\)> are RAZ. Software must treat reserved accesses as RES0. See *Access requirements for reserved and unallocated registers* on page I1-10720.

---

**Note**

AMECGCR.CG1NC identifies the number of auxiliary activity monitor event counters.

AMEVCNTR1<\(n\>][31:0] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0x100 + (8 * (n))</td>
<td>AMEVCNTR1&lt;(n&gt;)</td>
<td>31:0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
AMEVCNTR1<n>[63:32] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0x104 + (8 * n)</td>
<td>AMEVCNTR1&lt;n&gt;</td>
<td>63:32</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
### 5.5.18  AMEVTYPER0<n>, Activity Monitors Event Type Registers 0, n = 0 - 3

The AMEVTYPER0<n> characteristics are:

**Purpose**

Provides information on the events that an architected activity monitor event counter AMEVCNTR0<n> counts.

**Configurations**

External register AMEVTYPER0<n> bits [31:0] are architecturally mapped to AArch64 System register AMEVTYPER0<n>_EL0[31:0].

External register AMEVTYPER0<n> bits [31:0] are architecturally mapped to AArch32 System register AMEVTYPER0<n>[31:0].

The power domain of AMEVTYPER0<n> is IMPLEMENTATION DEFINED.

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMEVTYPER0<n> are RES0.

**Attributes**

AMEVTYPER0<n> is a 32-bit register.

**Field descriptions**

![Field Descriptions Diagram](image)

**Bits [31:16]**

Reserved, RES0.

**evtCount, bits [15:0]**

Event to count. The event number of the event that is counted by the architected activity monitor event counter AMEVCNTR0<n>. The value of this field is architecturally mandated for each architected counter.

The following table shows the mapping between required event numbers and the corresponding counters:

<table>
<thead>
<tr>
<th>Event Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0011</td>
<td>Processor frequency cycles</td>
</tr>
<tr>
<td>0x4004</td>
<td>Constant frequency cycles</td>
</tr>
<tr>
<td>0x0008</td>
<td>Instructions retired</td>
</tr>
<tr>
<td>0x4005</td>
<td>Memory stall cycles</td>
</tr>
</tbody>
</table>

**Accessing the AMEVTYPER0<n>:**

If <n> is greater than or equal to the number of architected activity monitor event counters, reads of AMEVTYPER0<n> are RAZ. Software must treat reserved accesses as RES0. See *Access requirements for reserved and unallocated registers* on page I1-10720.

**Note**

AMCGCR.CG0NC identifies the number of architected activity monitor event counters.
AMEVTYPER0<n> can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0x400 + (4 * n)</td>
<td>AMEVTYPER0&lt;n&gt;</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.5.19 AMEVTYPER1\langle n \rangle, Activity Monitors Event Type Registers 1, n = 0 - 15

The AMEVTYPER1\langle n \rangle characteristics are:

**Purpose**

Provides information on the events that an auxiliary activity monitor event counter AMEVCNTR1\langle n \rangle counts.

**Configurations**

External register AMEVTYPER1\langle n \rangle bits [31:0] are architecturally mapped to AArch64 System register AMEVTYPER1\langle n \rangle\_EL0[31:0].

External register AMEVTYPER1\langle n \rangle bits [31:0] are architecturally mapped to AArch32 System register AMEVTYPER1\langle n \rangle[31:0].

The power domain of AMEVTYPER1\langle n \rangle is IMPLEMENTATION DEFINED.

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMEVTYPER1\langle n \rangle are RES0.

**Attributes**

AMEVTYPER1\langle n \rangle is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>evtCount</td>
</tr>
</tbody>
</table>

**Accessing the AMEVTYPER1\langle n \rangle:**

If \langle n \rangle is greater than or equal to the number of auxiliary activity monitor event counters, reads of AMEVTYPER1\langle n \rangle are RAZ. Software must treat reserved accesses as RES0. See *Access requirements for reserved and unallocated registers* on page I1-10720.

**Note**

AMCGCR.CG1NC identifies the number of auxiliary activity monitor event counters.

AMEVTYPER1\langle n \rangle can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0x480+(4*\langle n \rangle)</td>
<td>AMEVTYPER1\langle n \rangle</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
### I5.5.20 AMIIDR, Activity Monitors Implementation Identification Register

The AMIIDR characteristics are:

**Purpose**

Defines the implementer and revisions of the AMU.

**Configurations**

The power domain of AMIIDR is IMPLEMENTATION DEFINED.

This register is present only when FEAT_AMUv1 is implemented. Otherwise, direct accesses to AMIIDR are RES0.

**Attributes**

AMIIDR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductID</td>
<td>bits [31:20]</td>
</tr>
<tr>
<td>Variant</td>
<td>bits [19:16]</td>
</tr>
<tr>
<td>Revision</td>
<td>bits [15:12]</td>
</tr>
<tr>
<td>Implementer</td>
<td>bits [11:0]</td>
</tr>
</tbody>
</table>

**ProductID, bits [31:20]**

This field is an AMU part identifier.

- If AMPIDR0 is implemented, AMPIDR0.PART_0 matches bits [27:20] of this field.
- If AMPIDR1 is implemented, AMPIDR1.PART_1 matches bits [31:28] of this field.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Variant, bits [19:16]**

This field distinguishes product variants or major revisions of the product.

- If AMPIDR2 is implemented, AMPIDR2.REVISION matches AMIIDR.Variant.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Revision, bits [15:12]**

This field distinguishes minor revisions of the product.

- If AMPIDR3 is implemented, AMPIDR3.REVAND matches AMIIDR.Revision.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Implementer, bits [11:0]**

Contains the JEP106 code of the company that implemented the AMU.

For an Arm implementation, this field reads as 0x43B.

- Bit 7 is RES0
- Bits [6:0] contain the JEP106 identity code of the implementer.

- If AMPIDR4 is implemented, AMPIDR4.DES_2 matches bits [11:8] of this field.
- If AMPIDR2 is implemented, AMPIDR2.DES_1 matches bits [6:4] of this field.
- If AMPIDR1 is implemented, AMPIDR1.DES_0 matches bits [3:0] of this field.

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

**Accessing the AMIIDR:**

AMIIDR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xE08</td>
<td>AMIIDR</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.5.21 AMPIDR0, Activity Monitors Peripheral Identification Register 0

The AMPIDR0 characteristics are:

**Purpose**

Provides information to identify an activity monitors component.

For more information, see About the Peripheral identification scheme on page K6-11696.

**Configurations**

The power domain of AMPIDR0 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when FEAT_AMUv1 is implemented.

**Attributes**

AMPIDR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-8</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>7-0</td>
<td>PART_0, Part number, least significant byte. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.</td>
</tr>
</tbody>
</table>

**Accessing the AMPIDR0:**

AMPIDR0 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xFE0</td>
<td>AMPIDR0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.5.22 AMPIDR1, Activity Monitors Peripheral Identification Register 1

The AMPIDR1 characteristics are:

**Purpose**

Provides information to identify an activity monitors component.

For more information, see About the Peripheral identification scheme on page K6-11696.

**Configurations**

The power domain of AMPIDR1 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when FEAT_AMUv1 is implemented.

**Attributes**

AMPIDR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>DES_0</td>
<td>PART_1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**DES_0, bits [7:4]**

Designer, least significant nibble of JEP106 ID code.

For Arm Limited, this field is 0b1011.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**PART_1, bits [3:0]**

Part number, most significant nibble.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Accessing the AMPIDR1:**

AMPIDR1 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xFE4</td>
<td>AMPIDR1</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
### 5.5.23 AMPIDR2, Activity Monitors Peripheral Identification Register 2

The AMPIDR2 characteristics are:

**Purpose**

Provides information to identify an activity monitors component.

For more information, see *About the Peripheral identification scheme* on page K6-11696.

**Configurations**

The power domain of AMPIDR2 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when FEAT_AMUv1 is implemented.

**Attributes**

AMPIDR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:8]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[7:4]</td>
<td>REVISION, Part major revision. Parts can also use this field to extend Part number to 16-bits. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.</td>
</tr>
<tr>
<td>[3]</td>
<td>JEDEC, indicates a JEP106 identity code is used. Reads as 0b1. Access to this field is RO.</td>
</tr>
<tr>
<td>[2:0]</td>
<td>DES_1, Designer, most significant bits of JEP106 ID code. For Arm Limited, this field is 0b011. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.</td>
</tr>
</tbody>
</table>

**Accessing the AMPIDR2:**

AMPIDR2 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xFE8</td>
<td>AMPIDR2</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
### I5.5.24 AMPIDR3, Activity Monitors Peripheral Identification Register 3

The AMPIDR3 characteristics are:

**Purpose**

Provides information to identify an activity monitors component.

For more information, see *About the Peripheral identification scheme* on page K6-11696.

**Configurations**

The power domain of AMPIDR3 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when FEAT_AMUv1 is implemented.

**Attributes**

AMPIDR3 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field Description</th>
<th>Value</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-8</td>
<td>RES0</td>
<td>Reserved, RES0.</td>
<td>RO</td>
</tr>
<tr>
<td>7-4</td>
<td>REVAND</td>
<td>Part minor revision. Parts using AMPIDR2.REVISION as an extension to the Part number must use this field as a major revision number. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.</td>
<td>RO</td>
</tr>
<tr>
<td>3-0</td>
<td>CMOD</td>
<td>Customer modified. Indicates someone other than the Designer has modified the component. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.</td>
<td>RO</td>
</tr>
</tbody>
</table>

**Accessing the AMPIDR3:**

AMPIDR3 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xFE</td>
<td>AMPIDR3</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.5.25  **AMPIDR4, Activity Monitors Peripheral Identification Register 4**

The AMPIDR4 characteristics are:

**Purpose**

Provides information to identify an activity monitors component.

For more information, see *About the Peripheral identification scheme* on page K6-11696.

**Configurations**

The power domain of AMPIDR4 is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

This register is present only when FEAT_AMUv1 is implemented.

**Attributes**

AMPIDR4 is a 32-bit register.

**Field descriptions**

![Field Descriptions](image)

**Bits [31:8]**

Reserved, RES0.

**SIZE, bits [7:4]**

Size of the component. \( \log_2 \) of the number of 4KB pages from the start of the component to the end of the component ID registers.

Reads as 0b0000.

Access to this field is RO.

**DES_2, bits [3:0]**

Designer. JEP106 continuation code, least significant nibble.

For Arm Limited, this field is 0b0100.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Accessing the AMPIDR4:**

AMPIDR4 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMU</td>
<td>0xF00</td>
<td>AMPIDR4</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
5.6 **Generic Timer memory-mapped registers overview**

The Generic Timer memory-mapped registers are implemented as multiple register frames, with each register frame having its own base address, as follows:

- A single CNTCTLBase register frame, at base address CNTCTLBase.
- Between one and seven CNTBase\(N\) register frames, each with its own base address CNTBase\(N\).
- For each CNTBase\(N\) register frame, if required, a CNTEL0Base\(N\) register frame, at base address CNTEL0Base\(N\), that provides an EL0 view of the CNTBase\(N\) register frame.

For more information, see:

- **Memory-mapped timer components** on page I2-10728.
- **The CNTBase\(N\) and CNTEL0Base\(N\) frames** on page I2-10729. This section includes the memory map of the CNTBase\(N\) and CNTBase\(N\) register frames.
- **The CNTCTLBase frame** on page I2-10728. This section includes the memory map of the CNTCTLBase register frame.

--- **Note** ---

*Providing a complete set of features in a system level implementation* on page K7-11702 gives an implementation example for a system level implementation of the Generic Timer."
I5.7 Generic Timer memory-mapped register descriptions

This section describes the Generic Timer registers. *Generic Timer memory-mapped registers overview on page I5-10874* gives an overview of these registers, and includes links to their memory maps.
I5.7.1 CNTACR<n>, Counter-timer Access Control Registers, n = 0 - 7

The CNTACR<n> characteristics are:

**Purpose**

Provides top-level access controls for the elements of a timer frame. CNTACR<n> provides the controls for frame CNTBaseN.

In addition to the CNTACR<n> control:

- CNTNSAR controls whether CNTACR<n> is accessible by Non-secure accesses.
- If frame CNTEL0BaseN is implemented, the CNTEL0ACR in frame CNTBaseN provides additional control of accesses to frame CNTEL0BaseN.

**Configurations**

The power domain of CNTACR<n> is IMPLEMENTATION DEFINED.

For more information, see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-10723.

Implemented only if the value of CNTTIDR.Frame<n> is 1.

An implementation of the counters might not provide configurable access to some or all of the features. In this case, the associated field in the CNTACR<n> register is:

- RAZ/WI if access is always denied.
- RAO/WI if access is always permitted.

**Attributes**

CNTACR<n> is a 32-bit register.

**Field descriptions**

![Register Diagram]

**Bits [31:6]**

Reserved, RES0.

**RWPT, bit [5]**

Read/write access to the EL1 Physical Timer registers CNTP_CVAL, CNTP_TVAL, and CNTP_CTL, in frame <n>.

- 0b0 No access to the EL1 Physical Timer registers in frame <n>. The registers are RES0.
- 0b1 Read/write access to the EL1 Physical Timer registers in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

**RWVT, bit [4]**

Read/write access to the Virtual Timer register CNTV_CVAL, CNTV_TVAL, and CNTV_CTL, in frame <n>.

- 0b0 No access to the Virtual Timer registers in frame <n>. The registers are RES0.
- 0b1 Read/write access to the Virtual Timer registers in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.
RVOFF, bit [3]

Read-only access to CNTVOFF, in frame \(<n>\).

0b0 No access to CNTVOFF in frame \(<n>\). The register is RES0.

0b1 Read-only access to CNTVOFF in frame \(<n>\).

The reset behavior of this field is:

* On a Timer reset, this field resets to an architecturally UNKNOWN value.

RFRQ, bit [2]

Read-only access to CNTFRQ, in frame \(<n>\).

0b0 No access to CNTFRQ in frame \(<n>\). The register is RES0.

0b1 Read-only access to CNTFRQ in frame \(<n>\).

The reset behavior of this field is:

* On a Timer reset, this field resets to an architecturally UNKNOWN value.

RVCT, bit [1]

Read-only access to CNTVCT, in frame \(<n>\).

0b0 No access to CNTVCT in frame \(<n>\). The register is RES0.

0b1 Read-only access to CNTVCT in frame \(<n>\).

The reset behavior of this field is:

* On a Timer reset, this field resets to an architecturally UNKNOWN value.

RPCT, bit [0]

Read-only access to CNTPCT, in frame \(<n>\).

0b0 No access to CNTPCT in frame \(<n>\). The register is RES0.

0b1 Read-only access to CNTPCT in frame \(<n>\).

The reset behavior of this field is:

* On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing the CNTACR\(<n>\):

In a system that recognizes two Security states:

* CNTACR\(<n>\) is always accessible by Secure accesses.

* CNTNSAR.NS\(<n>\) determines whether CNTACR\(<n>\) is accessible by Non-secure accesses.

CNTACR\(<n>\) can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTCTLBase</td>
<td>0x040 + (4 * n)</td>
<td>CNTACR(&lt;n&gt;)</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.7.2 CNTCR, Counter Control Register

The CNTCR characteristics are:

**Purpose**

Enables the counter, controls the counter frequency setting, and controls counter behavior during debug.

**Configurations**

The power domain of CNTCR is IMPLEMENTATION DEFINED.

For more information, see Power and reset domains for the system level implementation of the Generic Timer on page I2-10723.

**Attributes**

CNTCR is a 32-bit register.

**Field descriptions**

![Field descriptions diagram]

**Bits [31:18]**

Reserved, RES0.

**FCREQ, bits [17:8]**

Frequency change request. Indicates the number of the entry in the Frequency modes table to select.

Selecting an unimplemented entry, or an entry that contains 0, has no effect on the counter.

The maximum number of entries in the Frequency modes table is IMPLEMENTATION DEFINED up to a maximum of 1004 entries, see The Frequency modes table on page I2-10725. An implementation is only required to implement an FCREQ field that can hold values from 0 to the highest supported Frequency modes table entry. Any unrequired most-significant bits of FCREQ can be implemented as RES0.

The reset behavior of this field is:
- On a Timer reset, this field resets to 0.

**Bits [7:3]**

Reserved, RES0.

**SCEN, bit [2]**

Reserved, RES0.

**When FEAT_CNTSC is implemented:**

Scale Enable.

- **0b0** Scaling is not enabled. The counter value is incremented by 0x1.0000000 for each counter tick.
- **0b1** Scaling is enabled. The counter is incremented by CNTSCR.ScaleVal for each counter tick.

The SCEN bit can only be changed when the counter is disabled, when CNTCR.EN == 0.

If the value of CNTCR.SCEN changes when CNTCR.EN == 1 then:
- The counter value becomes UNKNOWN.
- The counter value remains UNKNOWN on future ticks of the clock.
When the CNTCV register in the CNTControlBase frame of the memory mapped counter module is written to, the accumulated fraction information is reset to zero.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**HDBG, bit [1]**

Halt-on-debug. Controls whether a Halt-on-debug signal halts the system counter:

- 0b0  System counter ignores Halt-on-debug.
- 0b1  Asserted Halt-on-debug signal halts system counter update.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

**EN, bit [0]**

Enables the counter:

- 0b0  System counter disabled.
- 0b1  System counter enabled.

The reset behavior of this field is:

- On a Timer reset, this field resets to 0.

**Accessing the CNTCR:**

In a system that supports Secure and Non-secure memory maps the CNTControlBase frame, that includes this register, is implemented only in the Secure memory map.

CNTCR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTControlBase</td>
<td>0x000</td>
<td>CNTCR</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
### 5.7.3 CNTCV, Counter Count Value register

The CNTCV characteristics are:

**Purpose**

Indicates the current count value.

**Configurations**

The power domain of CNTCV is IMPLEMENTATION DEFINED.

For more information, see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-10723.

**Attributes**

CNTCV is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>CountValue</td>
<td>0x0000</td>
<td>63:0</td>
</tr>
</tbody>
</table>

**CountValue, bits [63:0]**

Indicates the counter value.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

**Accessing the CNTCV:**

<table>
<thead>
<tr>
<th>Frame</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNTControlBase</td>
<td>RW</td>
</tr>
<tr>
<td>CNTReadBase</td>
<td>RO</td>
</tr>
</tbody>
</table>

A write to CNTCV must be visible in the CNTPCT register of each running processor in a finite time.

For the instance of the register in the CNTControlBase frame:

- In a system that supports Secure and Non-secure memory maps, the CNTControlBase frame, and therefore this register instance, is implemented only in the Secure memory map.
- If the counter is enabled, the effect of writing to the register is UNKNOWN.

In an implementation that supports 64-bit atomic memory accesses, this register must be accessible using a 64-bit atomic access.

CNTCV[63:0] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTControlBase</td>
<td>0x008</td>
<td>CNTCV</td>
<td>63:0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
CNTCV[63:0] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTReadBase</td>
<td>0x000</td>
<td>CNTCV</td>
<td>63:0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
### I5.7.4 CNTEL0ACR, Counter-timer EL0 Access Control Register

The CNTEL0ACR characteristics are:

**Purpose**  
An implementation of CNTEL0ACR in the frame at CNTBaseN controls whether the CNTPCT, CNTVCT, CNTFRQ, EL1 Physical Timer, and Virtual Timer registers are visible in the frame at CNTEL0BaseN.

**Configurations**  
The power domain of CNTEL0ACR is IMPLEMENTATION DEFINED.  
Implementation of this register is OPTIONAL.  
For more information, see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-10723.

**Attributes**  
CNTEL0ACR is a 32-bit register.

#### Field descriptions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-10</td>
<td>Reserved, RES0</td>
<td></td>
</tr>
</tbody>
</table>
| 9    | EL0PTEN  | Second view read/write access control for the EL1 Physical Timer registers. This bit controls whether the CNTP_CVAL, CNTP_TVAL, and CNTP_CTL registers in the current CNTBaseN frame are also accessible in the corresponding CNTEL0BaseN frame.  
0b0 No access. Registers are RES0 in the second view.  
0b1 Access permitted. If the registers are accessible in the current frame then they are accessible in the second view.  
The reset behavior of this field is:  
• On a Timer reset, this field resets to an architecturally UNKNOWN value. |
| 8    | EL0VTEN  | Second view read/write access control for the Virtual Timer registers. This bit controls whether the CNTV_CVAL, CNTV_TVAL, and CNTV_CTL registers in the current CNTBaseN frame are also accessible in the corresponding CNTEL0BaseN frame.  
0b0 No access. Registers are RES0 in the second view.  
0b1 Access permitted. If the registers are accessible in the current frame then they are accessible in the second view.  
The definition of this bit means that, if the Virtual Timer registers are not implemented in the current CNTBaseN frame, then the Virtual Timer register addresses are RES0 in the corresponding CNTEL0BaseN frame, regardless of the value of this bit.  
The reset behavior of this field is:  
• On a Timer reset, this field resets to an architecturally UNKNOWN value. |
Bits [7:2]

Reserved, RES0.

EL0VCTEN, bit [1]

Second view read access control for CNTVCT and CNTFRQ.

0b0  CNTVCT is not visible in the second view.
    If EL0PCTEN is set to 0, CNTFRQ is not visible in the second view.

0b1  Access permitted. If CNTVCT and CNTFRQ are visible in the current frame then they are visible in the second view.

The reset behavior of this field is:

• On a Timer reset, this field resets to an architecturally UNKNOWN value.

EL0PCTEN, bit [0]

Second view read access control for CNTPCT and CNTFRQ.

0b0  CNTPCT is not visible in the second view.
    If EL0VCTEN is set to 0, CNTFRQ is not visible in the second view.

0b1  Access permitted. If CNTPCT and CNTFRQ are visible in the current frame then they are visible in the second view.

The reset behavior of this field is:

• On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing the CNTEL0ACR:

CNTEL0ACR can be implemented in any implemented CNTBaseN frame.

CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-10731 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

• Whether the CNTBaseN frame has virtual timer capability.
• Whether the corresponding CNTEL0BaseN frame is implemented.
• For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

If CNTEL0ACR is not implemented in an implemented CNTBaseN frame:

• The register location in that frame is RAZ/WI.
• If the corresponding CNTEL0BaseN frame is implemented, the registers CNTFRQ, CNTP_CTL, CNTP_CVAL, CNTP_TVAL, CNTPCT, CNTV_CTL, CNTV_CVAL, CNTV_TVAL, and CNTVCT are not visible in that frame.

CNTEL0ACR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTBaseN</td>
<td>0x014</td>
<td>CNTEL0ACR</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
### 5.7.5 CNTFID0, Counter Frequency ID

The CNTFID0 characteristics are:

**Purpose**

Indicates the base frequency of the system counter.

**Configurations**

The power domain of CNTFID0 is IMPLEMENTATION DEFINED.

For more information, see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-10723.

The possible frequencies for the system counter are stored in the Frequency modes table as 32-bit words starting with the base frequency, CNTFID0. For more information, see *The Frequency modes table* on page I2-10725.

The final entry in the Frequency modes table must be followed by a 32-bit word of zero value, to mark the end of the table.

Typically, the Frequency modes table will be in read-only memory. However, a system implementation might use read/write memory for the table, and initialize the table entries as part of its start-up sequence.

If the Frequency modes table is in read/write memory, Arm strongly recommends that the table is not updated once the system is running.

**Attributes**

CNTFID0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Frequency, bits [31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>The base frequency of the system counter, in Hz.</td>
</tr>
<tr>
<td>The reset behavior of this field is:</td>
</tr>
<tr>
<td>• On a Timer reset, this field resets to an architecturally <em>UNKNOWN</em> value.</td>
</tr>
</tbody>
</table>

**Accessing the CNTFID0:**

It is IMPLEMENTATION DEFINED whether this register is RO or RW

In a system that supports Secure and Non-secure memory maps the CNTControlBase frame, that includes this register, is implemented only in the Secure memory map.

CNTFID0 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTControlBase</td>
<td>0x020</td>
<td>CNTFID0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO or RW.
5.7.6 CNTFID\(<n>\), Counter Frequency IDs, \(n > 0, n = 1 - 1003\)

The CNTFID\(<n>\) characteristics are:

**Purpose**
Indicates alternative system counter update frequencies.

**Configurations**
The power domain of CNTFID\(<n>\) is IMPLEMENTATION DEFINED.

For more information, see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-10723.

The possible frequencies for the system counter are stored in the Frequency modes table as 32-bit words starting with the base frequency, CNTFID0, see *The Frequency modes table* on page I2-10725.

The number of CNTFID\(<n>\) registers is IMPLEMENTATION DEFINED, and the only required CNTFID\(<n>\) register is CNTFID0.

The final entry in the Frequency modes table must be followed by a 32-bit word of zero value, to mark the end of the table.

The architecture can support up to 1004 entries in the Frequency modes table, including the zero-word end marker, and the number of entries is IMPLEMENTATION DEFINED up to this limit. For an implementation that includes registers in the IMPLEMENTATION DEFINED register space 0x0C0-0x0FC, the maximum number of entries in the Frequency modes table is 40, including the zero-word end marker.

Typically, the Frequency modes table will be in read-only memory. However, a system implementation might use read/write memory for the table, and initialize the table entries as part of its start-up sequence.

If the Frequency modes table is in read/write memory, Arm strongly recommends that the table is not updated once the system is running.

**Attributes**
CNTFID\(<n>\) is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td></td>
</tr>
</tbody>
</table>

**Frequency, bits [31:0]**
A system counter update frequency, in Hz. Must be an exact divisor of the base frequency. Arm strongly recommends that all frequency values in the Frequency modes table are integer power-of-two divisors of the base frequency.

When the system timer is operating at a lower frequency than the base frequency, the increment applied at each counter update is given by:

\[ \text{increment} = \frac{\text{base frequency}}{\text{selected frequency}} \]

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

**Accessing the CNTFID\(<n>\):**
It is IMPLEMENTATION DEFINED whether this register is RO or RW.
In a system that supports Secure and Non-secure memory maps the CNTControlBase frame, that includes these registers, is implemented only in the Secure memory map.

CNTFID<n> can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNCTControlBase</td>
<td>0x020 + (4 * n)</td>
<td>CNTFID&lt;n&gt;</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO or RW.
I5.7.7 CNTFRQ, Counter-timer Frequency

The CNTFRQ characteristics are:

**Purpose**

This register is provided so that software can discover the frequency of the system counter. The instance of the register in the CNTCTLB frame must be programmed with this value as part of system initialization. The value of the register is not interpreted by hardware.

**Configurations**

The power domain of CNTFRQ is IMPLEMENTATION DEFINED.

For more information see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-10723.

**Attributes**

CNTFRQ is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock frequency</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:0]**

Clock frequency. Indicates the system counter clock frequency, in Hz.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

**Accessing the CNTFRQ:**

CNTFRQ must be implemented as an RW register in the CNTCTLB frame.

In a system that recognizes two Security states, the instance of the register in the CNTCTLB frame is only accessible by Secure accesses.

CNTFRQ can be implemented as a RO register in any implemented CNTB frame, and in the corresponding CNTEL0B frame.

*CNTCTLB status and control fields for the CNTB and CNTEL0B frames* on page I2-10731 describes the status fields that identify whether a CNTB frame is implemented, and for an implemented frame:

- Whether the CNTB frame has virtual timer capability.
- Whether the corresponding CNTEL0B frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTB frame, and any corresponding CNTEL0B frame, is accessible by Non-secure accesses.

For an implemented CNTB frame:

- CNTFRQ is accessible in that frame, as a RO register, if the value of CNTACR<n>.RFRQ is 1.
- Otherwise, the CNTFRQ address in that frame is RAZ/WI.

For an implemented CNTEL0B frame:

- CNTFRQ is accessible as a RO register in that frame if both:
  - CNTFRQ is accessible in the corresponding CNTB frame.
  - Either the value of CNTEL0ACR.EL0VCTEN is 1 or the value of CNTEL0ACR.EL0PCTEN is 1.
• Otherwise, the CNTFRQ address in that frame is RAZ/WI.

CNTFRQ can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTBaseN</td>
<td>0x010</td>
<td>CNTFRQ</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.

CNTFRQ can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTEL0BaseN</td>
<td>0x010</td>
<td>CNTFRQ</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.

CNTFRQ can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTCTLBase</td>
<td>0x000</td>
<td>CNTFRQ</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
## 15.7.8 CNTID, Counter Identification Register

The CNTID characteristics are:

**Purpose**

Indicates whether counter scaling is implemented.

**Configurations**

The power domain of CNTID is IMPLEMENTATION DEFINED.

This register is present only when FEAT_CNTSC is implemented. Otherwise, direct accesses to CNTID are RES0.

**Attributes**

CNTID is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-4</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>3-0</td>
<td>CNTSC</td>
</tr>
</tbody>
</table>

**CNTSC, bits [3:0]**

Indicates whether Counter Scaling is implemented

- 0b0000: Counter scaling is not implemented.
- 0b0001: Counter scaling is implemented.

All other values are reserved.

**Accessing the CNTID:**

In a system that supports Secure and Non-secure memory maps, the CNTControlBase frame, that includes this register, is implemented only in the Secure memory map.

CNTID can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTControlBase</td>
<td>0x1C</td>
<td>CNTID</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.7.9 CNTNSAR, Counter-timer Non-secure Access Register

The CNTNSAR characteristics are:

Purpose

Provides the highest-level control of whether frames CNTBaseN and CNTEL0BaseN are accessible by Non-secure accesses.

Configurations

The power domain of CNTNSAR is IMPLEMENTATION DEFINED.

For more information, see Power and reset domains for the system level implementation of the Generic Timer on page I2-10723.

Attributes

CNTNSAR is a 32-bit register.

Field descriptions

| Bits [31:8] | Reserved, RES0. |
| NS<n>, bit [n], for n = 7 to 0 | Non-secure access to frame n. |
| 0b0 | Secure access only. Behaves as RES0 to Non-secure accesses. |
| 0b1 | Secure and Non-secure accesses permitted. |

This bit also determines whether, in the CNTCTLBase frame, CNTACR<n> and CNTVOFF<n> are accessible to Non-secure accesses.

If frame CNTBase<n>:
- Is not implemented, then NS<n> is RES0.
- Is not Configurable access, and is accessible only by Secure accesses, then NS<n> is RES0.
- Is not Configurable access, and is accessible by both Secure and Non-secure accesses, then NS<n> is RES1.

The reset behavior of this field is:
- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing the CNTNSAR:

In a system that recognizes two Security states, this register is only accessible by Secure accesses.

CNTNSAR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTCTLBase</td>
<td>0x004</td>
<td>CNTNSAR</td>
</tr>
</tbody>
</table>
Accesses to this interface are RW.
I5.7.10    CNTP_CTL, Counter-timer Physical Timer Control

The CNTP_CTL characteristics are:

**Purpose**
Control register for the EL1 physical timer.

**Configurations**
The power domain of CNTP_CTL is IMPLEMENTATION DEFINED.
For more information, see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-10723.

**Attributes**
CNTP_CTL is a 32-bit register.

**Field descriptions**

![Diagram of CNTP_CTL register]

**Bits [31:3]**
Reserved, RES0.

**ISTATUS, bit [2]**
The status of the timer. This bit indicates whether the timer condition is met:
0b0 Timer condition is not met.
0b1 Timer condition is met.
When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met.
ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.
When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.
The reset behavior of this field is:
- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

**IMASK, bit [1]**
Timer interrupt mask bit. Permitted values are:
0b0 Timer interrupt is not masked by the IMASK bit.
0b1 Timer interrupt is masked by the IMASK bit.
For more information, see the description of the ISTATUS bit.
The reset behavior of this field is:
- On a Timer reset, this field resets to an architecturally UNKNOWN value.

**ENABLE, bit [0]**
Enables the timer. Permitted values are:
0b0 Timer disabled.
0b1 Timer enabled.
Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTP_TV AL continues to count down.
--- Note ---
Disabling the output signal might be a power-saving option.

The reset behavior of this field is:
• On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing the CNTP_CTL:
CNTP_CTL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

_CNTCTRLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames_ on page I2-10731 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

• Whether the CNTBaseN frame has virtual timer capability.
• Whether the corresponding CNTEL0BaseN frame is implemented.
• For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

For an implemented CNTBaseN frame:
• CNTP_CTL is accessible in that frame if the value of CNTACR<n>.RWPT is 1.
• Otherwise, the CNTP_CTL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:
• CNTP_CTL is accessible in that frame if both:
  — CNTP_CTL is accessible in the corresponding CNTBaseN frame:
  — The value of CNTEL0ACR.EL0PTEN is 1.
• Otherwise, the CNTP_CTL address in that frame is RAZ/WI.

CNTP_CTL can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTBaseN</td>
<td>0x02C</td>
<td>CNTP_CTL</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.

CNTP_CTL can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTEL0BaseN</td>
<td>0x02C</td>
<td>CNTP_CTL</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.7.11 CNTP_CVAL, Counter-timer Physical Timer CompareValue

The CNTP_CVAL characteristics are:

**Purpose**

Holds the 64-bit compare value for the EL1 physical timer.

**Configurations**

The power domain of CNTP_CVAL is IMPLEMENTATION DEFINED.

For more information, see Power and reset domains for the system level implementation of the Generic Timer on page I2-10723.

**Attributes**

CNTP_CVAL is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CompareValue, bits [63:0]**

Holds the EL1 physical timer CompareValue.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer.

When the timer condition is met:

- CNTP_CTL.ISTATUS is set to 1.
- An interrupt is generated if CNTP_CTL.IMASK is 0.

When CNTP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

**Accessing the CNTP_CVAL:**

CNTP_CVAL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-10731 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

For an implemented CNTBaseN frame:

- CNTP_CVAL is accessible in that frame if the value of CNTACR<n>.RWPT is 1.
- Otherwise, the CNTP_CVAL address in that frame is RAZ/WI.
For an implemented CNTEL0BaseN frame:

- CNTP_CVAL is accessible in that frame if both:
  - CNTP_CVAL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0PTEN is 1.
- Otherwise, the CNTP_CVAL address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTP_CVAL register must be accessible as an atomic 64-bit value.

CNTP_CVAL[31:0] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTBaseN</td>
<td>0x020</td>
<td>CNTP_CVAL</td>
<td>31:0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.

CNTP_CVAL[31:0] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTEL0BaseN</td>
<td>0x020</td>
<td>CNTP_CVAL</td>
<td>31:0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.

CNTP_CVAL[63:32] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTBaseN</td>
<td>0x024</td>
<td>CNTP_CVAL</td>
<td>63:32</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.

CNTP_CVAL[63:32] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTEL0BaseN</td>
<td>0x024</td>
<td>CNTP_CVAL</td>
<td>63:32</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
## I5.7.12 CNTP_TVAL, Counter-timer Physical Timer TimerValue

The CNTP_TVAL characteristics are:

### Purpose
Holds the timer value for the EL1 physical timer.

### Configurations
The power domain of CNTP_TVAL is IMPLEMENTATION DEFINED.
For more information, see Power and reset domains for the system level implementation of the Generic Timer on page I2-10723.

### Attributes
CNTP_TVAL is a 32-bit register.

### Field descriptions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>TimerValue</td>
</tr>
</tbody>
</table>

**TimerValue, bits [31:0]**

The TimerValue view of the EL1 physical timer.
On a read of this register:
- If CNTP_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTP_CTL.ENABLE is 1, the value returned is (CompareValue - CNTPCT).

On a write of this register, CompareValue is set to (CNTPCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.
When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:
- CNTP_CTL.ISTATUS is set to 1.
- If CNTP_CTL.IMASK is 0, an interrupt is generated.
When CNTP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count, so the TimerValue view appears to continue to count down.
The reset behavior of this field is:
- On a Timer reset, this field resets to an architecturally UNKNOWN value.

### Accessing the CNTP_TVAL:

CNTP_TVAL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

CNTCTBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-10731 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.
For an implemented CNTBaseN frame:

- CNTP_TV AL is accessible in that frame if the value of CNTACR<n>.RWPT is 1.
- Otherwise, the CNTP_TV AL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTP_TV AL is accessible in that frame if both:
  - CNTP_TV AL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0PTEN is 1.
- Otherwise, the CNTP_TV AL address in that frame is RAZ/WI.

CNTP_TV AL can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTBaseN</td>
<td>0x028</td>
<td>CNTP_TV AL</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.

CNTP_TV AL can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTEL0BaseN</td>
<td>0x028</td>
<td>CNTP_TV AL</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.7.13 CNTPCT, Counter-timer Physical Count

The CNTPCT characteristics are:

**Purpose**

Holds the 64-bit physical count value.

**Configurations**

The power domain of CNTPCT is IMPLEMENTATION DEFINED.

For more information, see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-10723.

**Attributes**

CNTPCT is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Physical count value</th>
</tr>
</thead>
</table>

**Accessing the CNTPCT:**

CNTPCT can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame, as a RO register.

*CNTCTRLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames* on page I2-10731 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

For an implemented CNTBaseN frame:

- CNTPCT is accessible in that frame, as a RO register, if the value of CNTACR<n>.RPCT is 1.
- Otherwise, the CNTPCT address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTPCT is accessible in that frame if both:
  - CNTPCT is accessible in the corresponding CNTBaseN frame.
  - The value of CNTEL0ACR.EL0PCTEN is 1.
- Otherwise, the CNTPCT address in that frame is RAZ/WI.
If the implementation supports 64-bit atomic accesses, then the CNTPCT register must be accessible as an atomic 64-bit value.

CNTPCT[31:0] can be accessed through its memory-mapped interface:

```
Component  Frame  Offset  Instance  Range
Timer       CNTBaseN  0x000  CNTPCT  31:0
```

Accesses to this interface are RO.

CNTPCT[31:0] can be accessed through its memory-mapped interface:

```
Component  Frame  Offset  Instance  Range
Timer       CNTEL0BaseN  0x000  CNTPCT  31:0
```

Accesses to this interface are RO.

CNTPCT[63:32] can be accessed through its memory-mapped interface:

```
Component  Frame  Offset  Instance  Range
Timer       CNTBaseN  0x004  CNTPCT  63:32
```

Accesses to this interface are RO.

CNTPCT[63:32] can be accessed through its memory-mapped interface:

```
Component  Frame  Offset  Instance  Range
Timer       CNTEL0BaseN  0x004  CNTPCT  63:32
```

Accesses to this interface are RO.
### CNTSCR, Counter Scale Register

The CNTSCR characteristics are:

**Purpose**

Enables the counter, controls the counter frequency setting, and controls counter behavior during debug.

**Configurations**

The power domain of CNTSCR is IMPLEMENTATION DEFINED.

This register is present only when FEAT_CNTSC is implemented. Otherwise, direct accesses to CNTSCR are RES0.

For more information, see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-10723.

**Attributes**

CNTSCR is a 32-bit register.

**Field descriptions**

ScaleVal, bits [31:0]

Scale Value

When counter scaling is enabled, ScaleVal is the average amount added to the counter value for one period of the frequency of the Generic counter as described in the CNTFRQ register.

The actual rate of update of the counter value is determined by the counter update frequency.

ScaleVal is expressed as an unsigned fixed point number with an 8-bit integer value and a 24-bit fractional value.

CNTSCR.ScaleVal can only be changed when CNTCR.EN == 0. If the value of this field is changed when CNTCR.EN == 1:

- The counter value becomes UNKNOWN.
- The counter value remains UNKNOWN on future ticks of the clock.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

**Accessing the CNTSCR:**

In a system that supports Secure and Non-secure memory maps theCNTControlBase frame, that includes this register, is implemented only in the Secure memory map.

CNTSCR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTControlBase</td>
<td>0x10</td>
<td>CNTSCR</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.7.15 CNTSR, Counter Status Register

The CNTSR characteristics are:

**Purpose**

Provides counter frequency status information.

**Configurations**

The power domain of CNTSR is IMPLEMENTATION DEFINED.

For more information, see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-10723.

**Attributes**

CNTSR is a 32-bit register.

**Field descriptions**

Bits [31:18]

Reserved, RES0.

FCACK, bits [17:8]

Frequency Change Acknowledge. Indicates the currently selected entry in the Frequency modes table, see *The Frequency modes table* on page I2-10725.

The reset behavior of this field is:

- On a Timer reset, this field resets to 0.

Bits [7:2]

Reserved, RES0.

DBGH, bit [1]

Indicates whether the counter is halted because the Halt-on-debug signal is asserted:

- $0b0$ Counter is not halted.
- $0b1$ Counter is halted.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

**Accessing the CNTSR:**

In a system that supports Secure and Non-secure memory maps the CNTControlBase frame, that includes this register, is implemented only in the Secure memory map.

CNTSR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTControlBase</td>
<td>0x004</td>
<td>CNTSR</td>
</tr>
</tbody>
</table>
Accesses to this interface are RO.
I5.7.16  CNTTIDR, Counter-timer Timer ID Register

The CNTTIDR characteristics are:

**Purpose**

Indicates the implemented timers in the memory map, and their features. For each value of N from 0 to 7 it indicates whether:

- Frame CNTBaseN is a view of an implemented timer.
- Frame CNTBaseN has a second view, CNTEL0BaseN.
- Frame CNTBaseN has a virtual timer capability.

**Configurations**

The power domain of CNTTIDR is IMPLEMENTATION DEFINED.

For more information, see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-10723.

**Attributes**

CNTTIDR is a 32-bit register.

**Field descriptions**

Frame<n>, bits [4n+3:4n], for n = 7 to 0

A 4-bit field indicating the features of frame CNTBase<n>.

Bit[3] of the field is RES0.

Bit[2], the FEL0 subfield, indicates whether frame CNTBase<n> has a second view, CNTEL0Base<n>. The possible values of this bit are:

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Frame&lt;n&gt; does not have a second view. The register in the first view of the frame is</td>
</tr>
<tr>
<td>0b1</td>
<td>Frame&lt;n&gt; has a second view, CNTEL0Base&lt;n&gt;.</td>
</tr>
</tbody>
</table>

If bit[0] is 0, bit[2] is RES0.

Bit[1], the FVI subfield, indicates whether both:

- Frame CNTBase<n> implements the virtual timer registers CNTV_CVAL, CNTV_TVAL, and CNTV_CTL.
- This CNTCTLBase frame implements the virtual timer offset register CNTVOFF<n>.

The possible values of bit[1] are:

<table>
<thead>
<tr>
<th>Bit[1]</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Frame&lt;n&gt; does not have virtual capability. The virtual time and offset registers are</td>
</tr>
<tr>
<td>0b1</td>
<td>Frame&lt;n&gt; has virtual capability. The virtual time and offset registers are implemented</td>
</tr>
</tbody>
</table>

If bit[0] is 0, bit[1] is RES0.
Bit[0], the FI subfield, indicates whether frame CNTBase<n> is implemented. The possible values of this bit are:

<table>
<thead>
<tr>
<th>Bit[0]</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>Frame&lt;n&gt; is not implemented. All registers associated with the frame are.</td>
</tr>
<tr>
<td>0b1</td>
<td>Frame&lt;n&gt; is implemented</td>
</tr>
</tbody>
</table>

**Accessing the CNTTIDR:**

In a system that recognizes two Security states this register is accessible by both Secure and Non-secure accesses. CNTTIDR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTCTLBase</td>
<td>0x08</td>
<td>CNTTIDR</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
### 5.7.17 CNTV_CTL, Counter-timer Virtual Timer Control

The CNTV_CTL characteristics are:

**Purpose**
Control register for the virtual timer.

**Configurations**
The power domain of CNTV_CTL is IMPLEMENTATION DEFINED.
For more information, see [Power and reset domains for the system level implementation of the Generic Timer](#) on page I2-10723.

**Attributes**
CNTV_CTL is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>3</td>
<td>ISTATUS</td>
<td>The status of the timer. This bit indicates whether the timer condition is met:</td>
</tr>
<tr>
<td></td>
<td>0b0</td>
<td>Timer condition is not met.</td>
</tr>
<tr>
<td></td>
<td>0b1</td>
<td>Timer condition is met.</td>
</tr>
<tr>
<td>2</td>
<td>IMASK</td>
<td>Timer interrupt mask bit. Permitted values are:</td>
</tr>
<tr>
<td></td>
<td>0b0</td>
<td>Timer interrupt is not masked by the IMASK bit.</td>
</tr>
<tr>
<td></td>
<td>0b1</td>
<td>Timer interrupt is masked by the IMASK bit.</td>
</tr>
<tr>
<td>1</td>
<td>ENABLE</td>
<td>Enables the timer. Permitted values are:</td>
</tr>
<tr>
<td></td>
<td>0b0</td>
<td>Timer disabled.</td>
</tr>
<tr>
<td></td>
<td>0b1</td>
<td>Timer enabled.</td>
</tr>
</tbody>
</table>

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.
When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.
The reset behavior of this field is:
- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

### IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:
- 0b0: Timer interrupt is not masked by the IMASK bit.
- 0b1: Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.
The reset behavior of this field is:
- On a Timer reset, this field resets to an architecturally UNKNOWN value.

### ENABLE, bit [0]

Enables the timer. Permitted values are:
- 0b0: Timer disabled.
- 0b1: Timer enabled.
Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTV_TV AL continues to count down.
Note
Disabling the output signal might be a power-saving option.

The reset behavior of this field is:
- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing the CNTV_CTL:

CNTV_CTL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

CNTV_CTL can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTBaseN</td>
<td>0x03C</td>
<td>CNTV_CTL</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.

CNTV_CTL can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTEL0BaseN</td>
<td>0x03C</td>
<td>CNTV_CTL</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.7.18 CNTV_CVAL, Counter-timer Virtual Timer CompareValue

The CNTV_CVAL characteristics are:

**Purpose**

Holds the 64-bit compare value for the virtual timer.

**Configurations**

The power domain of CNTV_CVAL is IMPLEMENTATION DEFINED.

For more information, see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-10723.

**Attributes**

CNTV_CVAL is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompareValue</td>
<td>[63:0]</td>
</tr>
</tbody>
</table>

**CompareValue, bits [63:0]**

Holds the virtual timer CompareValue.

When CNTV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- An interrupt is generated if CNTV_CTL.IMASK is 0.

When CNTV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

**Accessing the CNTV_CVAL:**

CNTV_CVAL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

*CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames* on page I2-10731 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTV_CVAL is accessible in that frame if the value of CNTACR<n>.RWVT is 1.
- Otherwise, the CNTV_CVAL address in that frame is RAZ/WI.
For an implemented CNTEL0BaseN frame:

- CNTV_CVAL is accessible in that frame if both:
  - CNTV_CVAL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0VTEN is 1.

- Otherwise, the CNTV_CVAL address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTV_CVAL register must be accessible as an atomic 64-bit value.

CNTV_CVAL[31:0] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTBaseN</td>
<td>0x030</td>
<td>CNTV_CVAL</td>
<td>31:0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.

CNTV_CVAL[31:0] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTEL0BaseN</td>
<td>0x030</td>
<td>CNTV_CVAL</td>
<td>31:0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.

CNTV_CVAL[63:32] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTBaseN</td>
<td>0x034</td>
<td>CNTV_CVAL</td>
<td>63:32</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.

CNTV_CVAL[63:32] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTEL0BaseN</td>
<td>0x034</td>
<td>CNTV_CVAL</td>
<td>63:32</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.7.19 CNTV_TVAL, Counter-timer Virtual Timer TimerValue

The CNTV_TVAL characteristics are:

Purpose

Holds the timer value for the virtual timer.

Configurations

The power domain of CNTV_TVAL is IMPLEMENTATION DEFINED.

For more information, see Power and reset domains for the system level implementation of the Generic Timer on page I2-10723.

Attributes

CNTV_TVAL is a 32-bit register.

Field descriptions

TimerValue, bits [31:0]

The TimerValue view of the virtual timer.

On a read of this register:

- If CNTV_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTV_CTL.ENABLE is 1, the value returned is (CompareValue - CNTVCT).

On a write of this register, CompareValue is set to (CNTVCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- If CNTV_CTL.IMASK is 0, an interrupt is generated.

When CNTV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing the CNTV_TVAL:

CNTV_TVAL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-10731 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.
For an implemented CNTBaseN frame that has virtual timer capability:

- CNTV_TVAL is accessible in that frame if the value of CNTACR<n>.RWVT is 1.
- Otherwise, the CNTV_TVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTV_TVAL is accessible in that frame if both:
  - CNTV_TVAL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0VTEN is 1.
- Otherwise, the CNTV_TVAL address in that frame is RAZ/WI.

CNTV_TVAL can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTBaseN</td>
<td>0x038</td>
<td>CNTV_TVAL</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.

CNTV_TVAL can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTEL0BaseN</td>
<td>0x038</td>
<td>CNTV_TVAL</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.7.20 CNTVCT, Counter-timer Virtual Count

The CNTVCT characteristics are:

**Purpose**

Holds the 64-bit virtual count value.

**Configurations**

The power domain of CNTVCT is IMPLEMENTATION DEFINED.

For more information, see [Power and reset domains for the system level implementation of the Generic Timer on page I2-10723](#).

**Attributes**

CNTVCT is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [63:0]</th>
<th>Virtual count value</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:0]**

Virtual count value.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

**Accessing the CNTVCT:**

CNTVCT can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame, as a RO register.

[CntctlBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-10731](#) describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

For an implemented CNTBaseN frame:

- CNTVCT is accessible in that frame, as a RO register, if the value of CNTACR<n>.RVCT is 1.
- Otherwise, the CNTVCT address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTVCT is accessible in that frame if both:
  - CNTVCT is accessible in the corresponding CNTBaseN frame.
  - The value of CNTEL0ACR.EL0VCTEN is 1.
- Otherwise, the CNTVCT address in that frame is RAZ/WI.
If the implementation supports 64-bit atomic accesses, then the CNTVCT register must be accessible as an atomic 64-bit value.

CNTVCT[31:0] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTBaseN</td>
<td>0x008</td>
<td>CNTVCT</td>
<td>31:0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.

CNTVCT[31:0] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTEL0BaseN</td>
<td>0x008</td>
<td>CNTVCT</td>
<td>31:0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.

CNTVCT[63:32] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTBaseN</td>
<td>0x00C</td>
<td>CNTVCT</td>
<td>63:32</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.

CNTVCT[63:32] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTEL0BaseN</td>
<td>0x00C</td>
<td>CNTVCT</td>
<td>63:32</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.7.21 CNTVOFF, Counter-timer Virtual Offset

The CNTVOFF characteristics are:

**Purpose**

Holds the 64-bit virtual offset for a CNTBaseN frame that has virtual timer capability. This is the offset between real time and virtual time.

**Configurations**

The power domain of CNTVOFF is IMPLEMENTATION DEFINED.

For more information, see Power and reset domains for the system level implementation of the Generic Timer on page I2-10723.

**Attributes**

CNTVOFF is a 64-bit register.

**Field descriptions**

```
63  32  Virtual offset
31  0  Virtual offset
```

**Bits [63:0]**

Virtual offset.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

**Accessing the CNTVOFF:**

CNTVOFF is implemented, as a RO register, in any implemented CNTBaseN frame that has virtual timer capability.

CNTCTRLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-10731 describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTVOFF is accessible in that frame, as a RO register, if the value of CNTACR<\n>.RVOFF is 1.
- Otherwise, the CNTVOFF address in that frame is RAZ/WI.

**Note**

CNTVOFF is never visible in any CNTEL0BaseN frame. This means that the CNTVOFF address in any implemented CNTEL0BaseN frame is RAZ/WI.

In an implementation that supports 64-bit atomic accesses, a CNTVOFF{<\n>} register must be accessible as an atomic 64-bit value.
CNTVOFF[31:0] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTBaseN</td>
<td>0x018</td>
<td>31:0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.

CNTVOFF[63:32] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTBaseN</td>
<td>0x01C</td>
<td>63:32</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
### 15.7.22 CNTVOFF<\(n\)>, Counter-timer Virtual Offsets, \(n = 0 - 7\)

The CNTVOFF<\(n\)> characteristics are:

**Purpose**

Holds the 64-bit virtual offset for frame CNTBase<\(n\)>.
This is the offset between real time and virtual time.

**Configurations**

The power domain of CNTVOFF<\(n\)> is IMPLEMENTATION DEFINED.

Implementation of this register is OPTIONAL.

For more information, see *Power and reset domains for the system level implementation of the Generic Timer* on page I2-10723.

**Attributes**

CNTVOFF<\(n\)> is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Virtual offset</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Virtual offset</td>
</tr>
</tbody>
</table>

**Bits [63:0]**

Virtual offset.

The reset behavior of this field is:

* On a Timer reset, this field resets to an architecturally UNKNOWN value.

**Accessing the CNTVOFF<\(n\)>:**

In the CNTCTLBase frame a CNTVOFF<\(n\)> register must be implemented, as a RW register, for each CNTBaseN frame that has virtual timer capability.

For more information, see *CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames* on page I2-10731.

--- **Note** ---

The value of <\(n\)> in an instance of CNTVOFF<\(n\)> specifies the value of N for the associated CNTBaseN frame.

In a system that recognizes two Security states, for any CNTVOFF<\(n\)> register in the CNTCTLBase frame:

* CNTVOFF<\(n\)> is always accessible by Secure accesses.
* CNTNSAR.NS<\(n\)> determines whether CNTVOFF<\(n\)> is accessible by Non-secure accesses.

The register location of any unimplemented CNTVOFF<\(n\)> register in the CNTCTLBase frame is RAZ/WI.

The CNTVOFF<\(n\)> register is accessible in the CNTBaseN frame using CNTVOFF.

In an implementation that supports 64-bit atomic accesses, then the CNTVOFF<\(n\)> registers must be accessible as atomic 64-bit values.
CNTVOFF<n>[31:0] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTCTLBase</td>
<td>0x0000 + (8 * n)</td>
<td>31:0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.

CNTVOFF<n>[63:32] can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTCTLBase</td>
<td>0x0004 + (8 * n)</td>
<td>63:32</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
5.7.23 CounterID<n>, Counter ID registers, n = 0 - 11

The CounterID<n> characteristics are:

**Purpose**

IMPLEMENTATION DEFINED identification registers 0 to 11 for the memory-mapped Generic Timer.

**Configurations**

The power domain of CounterID<n> is IMPLEMENTATION DEFINED.

For more information, see Power and reset domains for the system level implementation of the Generic Timer on page I2-10723.

These registers are implemented independently in each of the implemented Generic Timer memory-mapped frames.

If the implementation of the Counter ID registers requires an architecture version, the value for this version of the Arm Generic Timer is version 0.

The Counter ID registers can be implemented as a set of CoreSight ID registers, comprising Peripheral ID Registers and Component ID Registers. An implementation of these registers for the Generic Timer must use a Component class value of 0xF.

**Attributes**

CounterID<n> is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [31:0]**

IMPLEMENTATION DEFINED.

**Accessing the CounterID<n>:**

These registers must be implemented, as RO registers, in every implemented Generic Timer memory-mapped frame.

For the CNTCTLBase frame, in a system that recognizes two Security states these registers are accessible by both Secure and Non-secure accesses.

For the CNTControlBase frame, in a system that supports Secure and Non-secure memory maps the frame is implemented only in the Secure memory map, meaning these registers are implemented only in the Secure memory map.

For the CNTBaseN frames, CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames on page I2-10731 describes the status fields that identify whether a frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses.
CounterID<\textit{n}> can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTControlBase</td>
<td>0xFD0 + (4 * \textit{n})</td>
<td>CounterID&lt;\textit{n}&gt;</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.

CounterID<\textit{n}> can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTReadBase</td>
<td>0xFD0 + (4 * \textit{n})</td>
<td>CounterID&lt;\textit{n}&gt;</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.

CounterID<\textit{n}> can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTBaseN</td>
<td>0xFD0 + (4 * \textit{n})</td>
<td>CounterID&lt;\textit{n}&gt;</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.

CounterID<\textit{n}> can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTEL0BaseN</td>
<td>0xFD0 + (4 * \textit{n})</td>
<td>CounterID&lt;\textit{n}&gt;</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.

CounterID<\textit{n}> can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>CNTCTLBase</td>
<td>0xFD0 + (4 * \textit{n})</td>
<td>CounterID&lt;\textit{n}&gt;</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.8 RAS register descriptions

This section describes the RAS registers.
### I5.8.1 ERRCIDR0, Component Identification Register 0

The ERRCIDR0 characteristics are:

**Purpose**

Provides discovery information about the component.

For more information, see *About the Peripheral identification scheme* on page K6-11696.

**Configurations**

Implementation of this register is OPTIONAL.

ERRCIDR0 is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRCIDR0 is a 32-bit register.

**Field descriptions**

![Field Description Table]

- ** Bits [31:8] **
  - Reserved, RES0.

- **PRMBL_0, bits [7:0]**
  - Component identification preamble, segment 0.
  - Reads as 0x0D.
  - Access to this field is RO.

**Accessing the ERRCIDR0:**

ERRCIDR0 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xFF0</td>
<td>ERRCIDR0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
### 5.8.2 ERRCIDR1, Component Identification Register 1

The ERRCIDR1 characteristics are:

**Purpose**

Provides discovery information about the component.

For more information, see *About the Peripheral identification scheme* on page K6-11696.

**Configurations**

Implementation of this register is OPTIONAL.

ERRCIDR1 is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRCIDR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:8]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b1111</td>
<td>Generic peripheral with IMPLEMENTATION DEFINED register layout.</td>
</tr>
<tr>
<td>Other values are defined by the CoreSight Architecture.</td>
<td></td>
</tr>
<tr>
<td>This field reads as 0xF.</td>
<td></td>
</tr>
<tr>
<td>PRMBL_1, bits [3:0]</td>
<td>Component identification preamble, segment 1.</td>
</tr>
<tr>
<td>Reads as 0b0000.</td>
<td></td>
</tr>
<tr>
<td>Access to this field is RO.</td>
<td></td>
</tr>
</tbody>
</table>

**Accessing the ERRCIDR1:**

ERRCIDR1 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xFF4</td>
<td>ERRCIDR1</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.8.3 ERRCIDR2, Component Identification Register 2

The ERRCIDR2 characteristics are:

**Purpose**

Provides discovery information about the component.
For more information, see *About the Peripheral identification scheme on page K6-11696.*

**Configurations**

Implementation of this register is *OPTIONAL.*
ERRCIDR2 is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRCIDR2 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>RES0</td>
</tr>
<tr>
<td>8</td>
<td>PRMBL_2</td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**PRMBL_2, bits [7:0]**

Component identification preamble, segment 2.
Reads as 0x05.
Access to this field is RO.

**Accessing the ERRCIDR2:**

ERRCIDR2 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xFF8</td>
<td>ERRCIDR2</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.8.4 ERRCIDR3, Component Identification Register 3

The ERRCIDR3 characteristics are:

**Purpose**

Provides discovery information about the component.

For more information, see About the Peripheral identification scheme on page K6-11696.

**Configurations**

Implementation of this register is OPTIONAL.

ERRCIDR3 is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRCIDR3 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits [31:8]</th>
<th>Reserved, RES0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRMBL_3, bits [7:0]</td>
<td>Component identification preamble, segment 3. Reads as 0xB1. Access to this field is RO.</td>
</tr>
</tbody>
</table>

**Accessing the ERRCIDR3:**

ERRCIDR3 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xFFC</td>
<td>ERRCIDR3</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
### 5.8.5 ERRCRICR0, Critical Error Interrupt Configuration Register 0

The ERRCRICR0 characteristics are:

**Purpose**

Critical Error Interrupt configuration register.

**Configurations**

This register is present only when (the Critical Error Interrupt is implemented or the implementation does not use the recommended layout for the ERRIRQCR<n> registers) and interrupt configuration registers are implemented. Otherwise, direct accesses to ERRCRICR0 are RES0.

ERRCRICR0 is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRCRICR0 is a 64-bit register.

**Field descriptions**

*When the Critical Error Interrupt is implemented and the implementation uses the recommended layout for the ERRIRQCR<n> registers:*

- **Bits [63:56]**
  
  Reserved, RES0.

- **ADDR, bits [55:2]**
  
  Message Signaled Interrupt address. \((\text{ERRCRICR0.ADDR} \ll 2)\) is the address that the component writes to when signaling the Critical Error Interrupt. Bits [1:0] of the address are always zero.
  
  The physical address size supported by the component is IMPLEMENTATION DEFINED.
  
  Unimplemented high-order physical address bits are RES0.
  
  The reset behavior of this field is:
  
  - On an Error recovery reset, this field resets to an architecturally UNKNOWN value.

- **Bits [1:0]**
  
  Reserved, RES0.

*When the implementation does not use the recommended layout for the ERRIRQCR<n> registers:*

- **IMPLEMENTATION DEFINED, bits [63:0]**
  
  IMPLEMENTATION DEFINED.
Accessing the ERRCRICR0:

ERRCRICR0 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xEA0</td>
<td>ERRCRICR0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.8.6  ERRCRICR1, Critical Error Interrupt Configuration Register 1

The ERRCRICR1 characteristics are:

**Purpose**

Critical Error Interrupt configuration register.

**Configurations**

This register is present only when (the Critical Error Interrupt is implemented or the implementation does not use the recommended layout for the ERRIRQCR<n> registers) and interrupt configuration registers are implemented. Otherwise, direct accesses to ERRCRICR1 are RES0. ERRCRICR1 is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRCRICR1 is a 32-bit register.

**Field descriptions**

*When the Critical Error Interrupt is implemented and the implementation uses the recommended layout for the ERRIRQCR<n> registers:*

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>Payload for the message signaled interrupt. The reset behavior of this field is:</td>
</tr>
<tr>
<td></td>
<td>- On an Error recovery reset, this field resets to an architecturally UNKNOWN value.</td>
</tr>
</tbody>
</table>

*When the implementation does not use the recommended layout for the ERRIRQCR<n> registers:*

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td>IMPLEMENTATION DEFINED.</td>
</tr>
</tbody>
</table>

**Accessing the ERRCRICR1:**

ERRCRICR1 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xEA8</td>
<td>ERRCRICR1</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.8.7 ERRCRICR2, Critical Error Interrupt Configuration Register 2

The ERRCRICR2 characteristics are:

Purpose
Critical Error Interrupt control and configuration register.

Configurations
This register is present only when (the Critical Error Interrupt is implemented or the implementation
does not use the recommended layout for the ERRIRQCR<n> registers) and interrupt configuration
registers are implemented. Otherwise, direct accesses to ERRCRICR2 are RES0.
ERRCRICR2 is implemented only as part of a memory-mapped group of error records.

Attributes
ERRCRICR2 is a 32-bit register.

Field descriptions
When the Critical Error Interrupt is implemented and the implementation uses the recommended
layout for the ERRIRQCR<n> registers:

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>IRQEN</td>
<td>MemAttr</td>
<td>NSMSI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits [31:8]
Reserved, RES0.

IRQEN, bit [7]
When the component supports disabling message signaled interrupts:
Message signaled interrupt enable. Enables generation of message signaled interrupts.
0b0 Disabled.
0b1 Enabled.
The reset behavior of this field is:
• On an Error recovery reset, this field resets to 0.

Otherwise:
Reserved, RES0.
Message signaled interrupt enable.
Message signaled interrupts are always enabled.

NSMSI, bit [6]
When the component supports configuring the Security attribute for message signaled interrupts and
the component does not allow Non-secure writes to ERRCRICR2:
Security attribute. Defines the physical address space for message signaled interrupts.
0b0 Secure.
0b1 Non-secure.
The reset behavior of this field is:
• On a Error recovery reset, this field resets to an IMPLEMENTATION DEFINED value.

When the component allows Non-secure writes to ERRCRICR2:
Reserved, RES0.
Security attribute. Defines the physical address space for message signaled interrupts. The Security attribute used for message signaled interrupts is Non-secure.

Otherwise:
Reserved, RES0.

Security attribute. Defines the physical address space for message signaled interrupts. The Security attribute for message signaled interrupts is IMPLEMENTATION DEFINED.

\(SH, \text{ bits } [5:4]\)

\textbf{When the component supports configuring the Shareability domain for message signaled interrupts:}

Shareability. Defines the Shareability domain for message signaled interrupts.

- 0b00 Not shared.
- 0b10 Outer Shareable.
- 0b11 Inner Shareable.

All other values are reserved.

This field is ignored when ERRCRICR2.MemAttr specifies any of the following memory types:

- Any Device memory type.
- Normal memory, Inner Non-cacheable, Outer Non-cacheable.

All Device and Normal Inner Non-cacheable Outer Non-cacheable memory regions are always treated as Outer Shareable.

The reset behavior of this field is:

- On an Error recovery reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Shareability.

The Shareability domain for message signaled interrupts is IMPLEMENTATION DEFINED.

\(\text{MemAttr, bits } [3:0]\)

\textbf{When the component supports configuring the memory type for message signaled interrupts:}

Memory type. Defines the memory type and attributes for message signaled interrupts.

- 0b0000 Device-nGnRnE memory.
- 0b0001 Device-nGnRE memory.
- 0b0010 Device-nGRE memory.
- 0b0011 Device-GRE memory.
- 0b0101 Normal memory, Inner Non-cacheable, Outer Non-cacheable.
- 0b0110 Normal memory, Inner Write-Through, Outer Non-cacheable.
- 0b0111 Normal memory, Inner Write-Back, Outer Non-cacheable.
- 0b1001 Normal memory, Inner Non-cacheable, Outer Write-Through.
- 0b1010 Normal memory, Inner Write-Through, Outer Write-Through.
- 0b1011 Normal memory, Inner Write-Back, Outer Write-Through.
- 0b1101 Normal memory, Inner Non-cacheable, Outer Write-Back.
- 0b1110 Normal memory, Inner Write-Through, Outer Write-Back.
- 0b1111 Normal memory, Inner Write-Back, Outer Write-Back.

All other values are reserved.

\textbf{Note}

This is the same format as the VMSA\text{v8-64} stage 2 memory region attributes.
The reset behavior of this field is:
- On an Error recovery reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
- Reserved, RES0.
- Memory type.

The memory type used for message signaled interrupts is IMPLEMENTATION DEFINED.

**When the implementation does not use the recommended layout for the ERRIRQCR<n> registers:**

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
</tbody>
</table>

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

**Accessing the ERRCRICR2:**

ERRCRICR2 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xEAC</td>
<td>ERRCRICR2</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
### I5.8.8 ERRDEVAFF, Device Affinity Register

The ERRDEVAFF characteristics are:

**Purpose**

For a group of error records that has affinity with a single PE or a group of PEs, ERRDEVAFF is a copy of MPIDR_EL1 or part of MPIDR_EL1:

- If the group of error records has affinity with a single PE, the affinity level is 0, ERRDEVAFF reads the same value as MPIDR_EL1, and ERRDEVAFF.F0V reads-as-one to indicate affinity level 0.
- If the group of error records has affinity with a group of PEs, the affinity level is 1, 2, or 3, parts of ERRDEVAFF reads the same value as parts of MPIDR_EL1, and the rest of ERRDEVAFF indicates the level.

For example, if the group of PEs is a subset of the PEs at affinity level 1 then all of the following are true:

- All the PEs in the group have the same values in MPIDR_EL1.Aff3,Aff2, and these values are equal to ERRDEVAFF.Aff3,Aff2.
- ERRDEVAFF.Aff1 is nonzero and not 0x80, and ERRDEVAFF.F0V read-as-zero, to indicate at least affinity level 1. The subset of PEs at level 1 that the group of error records has affinity with is indicated by the least-significant set bit in ERRDEVAFF.Aff1. In this example, if ERRDEVAFF.Aff1[2:0] is 0b100, then the group of error records has affinity with the up-to 8 PEs that have MPIDR_EL1.Aff1[7:3] == ERRDEVAFF.Aff1[7:3].

If RAS System Architecture v1.1 is not implemented, ERRDEVAFF can only describe a group of error records that is affine with a single PE or all the PEs at an affinity level.

**Configurations**

This register is present only when the group of error records has affinity with a PE or cluster of PEs. Otherwise, direct accesses to ERRDEVAFF are RES0.

ERRDEVAFF is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRDEVAFF is a 64-bit register.

**Field descriptions**

- **Bits [63:40]**  
  Reserved, RES0.
- **Aff3, bits [39:32]**  
  PE affinity level 3. The MPIDR_EL1.Aff3 field, viewed from the highest Exception level of the associated PE or PEs.
- **F0V, bit [31]**  
  Indicates that the ERRDEVAFF.Aff0 field is valid.
  
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0</td>
<td>ERRDEVAFF.Aff0 is not valid, and the PE affinity is above level 0 or a subset of level 0.</td>
</tr>
<tr>
<td>0b1</td>
<td>ERRDEVAFF.Aff0 is valid, and the PE affinity is at level 0.</td>
</tr>
</tbody>
</table>
U, bit [30]

**When ERRDEVAFF.F0V == 1:**

Uniprocessor. The MPIDR_EL1.U field, viewed from the highest Exception level of the associated PE.

**Otherwise:**

Reserved, UNKNOWN.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

**When ERRDEVAFF.F0V == 1:**

Multithreaded. The MPIDR_EL1.MT field, viewed from the highest Exception level of the associated PE.

**Otherwise:**

Reserved, UNKNOWN.

Aff2, bits [23:16]

**When affine with a PE or PEs at affinity level 2 or below:**

PE affinity level 2. The MPIDR_EL1.Aff2 field, viewed from the highest Exception level of the associated PE or PEs.

**When affine with a sub-set of PEs at affinity level 2:**

PE affinity level 2. Defines part of the MPIDR_EL1.Aff2 field, viewed from the highest Exception level of the associated PEs.

0bxxxxxxx1 ERRDEVAFF.Aff2[7:1] is the value of MPIDR_EL1.Aff2[7:1], viewed from the highest Exception level of the associated PEs.

0bxxxxxx10 ERRDEVAFF.Aff2[7:2] is the value of MPIDR_EL1.Aff2[7:2], viewed from the highest Exception level of the associated PEs.

0bxxxxx100 ERRDEVAFF.Aff2[7:3] is the value of MPIDR_EL1.Aff2[7:3], viewed from the highest Exception level of the associated PEs.

0bxxxx1000 ERRDEVAFF.Aff2[7:4] is the value of MPIDR_EL1.Aff2[7:4], viewed from the highest Exception level of the associated PEs.

0bxxx10000 ERRDEVAFF.Aff2[7:5] is the value of MPIDR_EL1.Aff2[7:5], viewed from the highest Exception level of the associated PEs.

0bxx100000 ERRDEVAFF.Aff2[7:6] is the value of MPIDR_EL1.Aff2[7:6], viewed from the highest Exception level of the associated PEs.

0bx1000000 ERRDEVAFF.Aff2[7] is the value of MPIDR_EL1.Aff2[7], viewed from the highest Exception level of the associated PEs.

**Otherwise:**

PE affinity level 2. Indicates whether the PE affinity is at level 3.

0x80 PE affinity is at level 3.

All other values are reserved.

Aff1, bits [15:8]

**When affine with a PE or PEs at affinity level 1 or below:**

PE affinity level 1. The MPIDR_EL1.Aff1 field, viewed from the highest Exception level of the associated PE or PEs.
**When affine with a sub-set of PEs at affinity level 1:**

PE affinity level 1. Defines part of the MPIDR_EL1.Aff1 field, viewed from the highest Exception level of the associated PEs.

- 0bxxxxxxx1 ERRDEVAFF.Aff1[7:1] is the value of MPIDR_EL1.Aff1[7:1], viewed from the highest Exception level of the associated PEs.
- 0bxxxxxx10 ERRDEVAFF.Aff1[7:2] is the value of MPIDR_EL1.Aff1[7:2], viewed from the highest Exception level of the associated PEs.
- 0bxxxxx100 ERRDEVAFF.Aff1[7:3] is the value of MPIDR_EL1.Aff1[7:3], viewed from the highest Exception level of the associated PEs.
- 0bxxxx1000 ERRDEVAFF.Aff1[7:4] is the value of MPIDR_EL1.Aff1[7:4], viewed from the highest Exception level of the associated PEs.
- 0bxxx10000 ERRDEVAFF.Aff1[7:5] is the value of MPIDR_EL1.Aff1[7:5], viewed from the highest Exception level of the associated PEs.
- 0bxx100000 ERRDEVAFF.Aff1[7:6] is the value of MPIDR_EL1.Aff1[7:6], viewed from the highest Exception level of the associated PEs.
- 0bx1000000 ERRDEVAFF.Aff1[7] is the value of MPIDR_EL1.Aff1[7], viewed from the highest Exception level of the associated PEs.

**Otherwise:**

PE affinity level 1. Indicates whether the PE affinity is at level 2.

- 0x00 PE affinity is above level 2 or a subset of level 2.
- 0x80 PE affinity is at level 2.

**Aff0, bits [7:0]**

**When affine with a PE at affinity level 0:**

PE affinity level 0. The MPIDR_EL1.Aff0 field, viewed from the highest Exception level of the associated PE.

**When affine with a sub-set of PEs at affinity level 0:**

PE affinity level 0. Defines part of the MPIDR_EL1.Aff0 field, viewed from the highest Exception level of the associated PEs.

- 0bxxxxxxx1 ERRDEVAFF.Aff0[7:1] is the value of MPIDR_EL1.Aff0[7:1], viewed from the highest Exception level of the associated PEs.
- 0bxxxxxx10 ERRDEVAFF.Aff0[7:2] is the value of MPIDR_EL1.Aff0[7:2], viewed from the highest Exception level of the associated PEs.
- 0bxxxxx100 ERRDEVAFF.Aff0[7:3] is the value of MPIDR_EL1.Aff0[7:3], viewed from the highest Exception level of the associated PEs.
- 0bxxxx1000 ERRDEVAFF.Aff0[7:4] is the value of MPIDR_EL1.Aff0[7:4], viewed from the highest Exception level of the associated PEs.
- 0bxxx10000 ERRDEVAFF.Aff0[7:5] is the value of MPIDR_EL1.Aff0[7:5], viewed from the highest Exception level of the associated PEs.
- 0bxx100000 ERRDEVAFF.Aff0[7:6] is the value of MPIDR_EL1.Aff0[7:6], viewed from the highest Exception level of the associated PEs.
- 0bx1000000 ERRDEVAFF.Aff0[7] is the value of MPIDR_EL1.Aff0[7], viewed from the highest Exception level of the associated PEs.

**Otherwise:**

PE affinity level 0. Indicates whether the PE affinity is at level 1.

- 0x00 PE affinity is above level 1 or a subset of level 1.
- 0x80 PE affinity is at level 1.
Accessing the ERRDEVAFF:

ERRDEVAFF can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xFA8</td>
<td>ERRDEVAFF</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
### 5.8.9 ERRDEVARCH, Device Architecture Register

The ERRDEVARCH characteristics are:

**Purpose**

Provides discovery information for the component.

**Configurations**

ERRDEVARCH is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRDEVARCH is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>21</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARCHITECT</td>
<td>REVISION</td>
<td>ARCHVER</td>
<td>ARCHPART</td>
<td>PRESENT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ARCHITECT, bits [31:21]**

Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code.

0b01000111011 JEP106 continuation code 0x4, ID code 0x3B. Arm Limited.

Other values are defined by the JEDEC JEP106 standard.

This field reads as 0x23B.

**PRESENT, bit [20]**

DEV ARCH Present. Defines that the DEV ARCH register is present.

0 Device Architecture information not present.

1 Device Architecture information present.

This field reads as 1.

**REVISION, bits [19:16]**

Revision. Defines the architecture revision of the component.

0000 RAS System Architecture v1.0.

0001 RAS System Architecture v1.1. As 0000 and also:

- Simplifies ERR<n>-STATUS.
- Adds support for additional ERR<n>MISC<m> registers.
- Adds support for the optional RAS Timestamp Extension.
- Adds support for the optional Common Fault Injection Model Extension.

All other values are reserved.

**ARCHVER, bits [15:12]**

Architecture Version. Defines the architecture version of the component.

0000 RAS System Architecture v1.

All other values are reserved.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as 0b0000.
ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

0xA00 RAS System Architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].
This field reads as 0xA00.

**Accessing the ERRDEVARCH:**

ERRDEVARCH can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xFBCC</td>
<td>ERRDEVARCH</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
### I5.8.10 ERRDEVID, Device Configuration Register

The ERRDEVID characteristics are:

**Purpose**

Provides discovery information for the component.

**Configurations**

ERRDEVID is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRDEVID is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits [31:16]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>NUM, bits [15:0]</td>
<td>Highest numbered index of the error records in this group, plus one. Each implemented record is owned by a node. A node might own multiple records. This manual describes a group of error records accessed via a standard 4KB memory-mapped peripheral. For a 4KB peripheral, up to 24 error records can be accessed if the Common Fault Injection Model is implemented, and up to 56 otherwise. This field has an IMPLEMENTATION DEFINED value. Access to this field is RO.</td>
</tr>
</tbody>
</table>

**Accessing the ERRDEVID:**

ERRDEVID can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xFC8</td>
<td>ERRDEVID</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.8.11 ERRERICR0, Error Recovery Interrupt Configuration Register 0

The ERRERICR0 characteristics are:

Purpose

Error Recovery Interrupt configuration register.

Configurations

This register is present only when (the Error Recovery Interrupt is implemented or the implementation does not use the recommended layout for the ERRIRQCR<n> registers) and interrupt configuration registers are implemented. Otherwise, direct accesses to ERRERICR0 are RES0.

ERRERICR0 is implemented only as part of a memory-mapped group of error records.

Attributes

ERRERICR0 is a 64-bit register.

Field descriptions

When the Error Recovery Interrupt is implemented and the implementation uses the recommended layout for the ERRIRQCR<n> registers:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits [63:56]</td>
<td>Reserved, RES0.</td>
</tr>
</tbody>
</table>
| ADDR, bits [55:2] | Message Signaled Interrupt address. (ERRERICR0.ADDR << 2) is the address that the component writes to when signaling the Error Recovery Interrupt. Bits [1:0] of the address are always zero. The physical address size supported by the component is IMPLEMENTATION DEFINED. Unimplemented high-order physical address bits are RES0. The reset behavior of this field is:  
  - On an Error recovery reset, this field resets to an architecturally UNKNOWN value. |
| Bits [1:0] | Reserved, RES0. |

When the implementation does not use the recommended layout for the ERRIRQCR<n> registers:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED, bits [63:0]</td>
<td>IMPLEMENTATION DEFINED.</td>
</tr>
</tbody>
</table>
Accessing the ERRERICR0:

ERRERICR0 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xE90</td>
<td>ERRERICR0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.8.12 ERRERICR1, Error Recovery Interrupt Configuration Register 1

The ERRERICR1 characteristics are:

**Purpose**

Error Recovery Interrupt configuration register.

**Configurations**

This register is present only when (the Error Recovery Interrupt is implemented or the implementation does not use the recommended layout for the ERRIRQCR<n> registers) and interrupt configuration registers are implemented. Otherwise, direct accesses to ERRERICR1 are RES0.

ERRERICR1 is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRERICR1 is a 32-bit register.

**Field descriptions**

*When the Error Recovery Interrupt is implemented and the implementation uses the recommended layout for the ERRIRQCR<n> registers:*

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| DATA, bits [31:0] | Payload for the message signaled interrupt. The reset behavior of this field is:  
  • On an Error recovery reset, this field resets to an architecturally UNKNOWN value. |

*When the implementation does not use the recommended layout for the ERRIRQCR<n> registers:*

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED, bits [31:0]</td>
<td>IMPLEMENTATION DEFINED.</td>
</tr>
</tbody>
</table>

**Accessing the ERRERICR1:**

ERRERICR1 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xE8</td>
<td>ERRERICR1</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
### 5.8.13 ERRERICR2, Error Recovery Interrupt Configuration Register 2

The ERRERICR2 characteristics are:

**Purpose**

Error Recovery Interrupt control and configuration register.

**Configurations**

This register is present only when (the Error Recovery Interrupt is implemented or the implementation does not use the recommended layout for the ERRIRQCR<n> registers) and interrupt configuration registers are implemented. Otherwise, direct accesses to ERRERICR2 are RES0.

ERRERICR2 is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRERICR2 is a 32-bit register.

**Field descriptions**

*When the Error Recovery Interrupt is implemented and the implementation uses the recommended layout for the ERRIRQCR<n> registers:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:8</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>7</td>
<td><strong>IRQEN</strong>, bit [7]</td>
</tr>
<tr>
<td>6</td>
<td><strong>NSMSI</strong>, bit [6]</td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**IRQEN, bit [7]**

*When the component supports disabling message signaled interrupts:*

Message signaled interrupt enable. Enables generation of message signaled interrupts.

- `0b0`: Disabled.
- `0b1`: Enabled.

The reset behavior of this field is:

- On an Error recovery reset, this field resets to 0.

**Otherwise:**

Reserved, RES0.

Message signaled interrupt enable.

Message signaled interrupts are always enabled.

**NSMSI, bit [6]**

*When the component supports configuring the Security attribute for message signaled interrupts and the component does not allow Non-secure writes to ERRERICR2:*

Security attribute. Defines the physical address space for message signaled interrupts.

- `0b0`: Secure.
- `0b1`: Non-secure.

The reset behavior of this field is:

- On a Error recovery reset, this field resets to an IMPLEMENTATION DEFINED value.
When the component allows Non-secure writes to ERRERICR2:

Reserved, RES0.

Security attribute. Defines the physical address space for message signaled interrupts.
The Security attribute used for message signaled interrupts is Non-secure.

Otherwise:

Reserved, RES0.

Security attribute. Defines the physical address space for message signaled interrupts.
The Security attribute for message signaled interrupts is IMPLEMENTATION DEFINED.

SH, bits [5:4]

When the component supports configuring the Shareability domain for message signaled interrupts:

Shareability. Defines the Shareability domain for message signaled interrupts.

0b00 Not shared.
0b10 Outer Shareable.
0b11 Inner Shareable.

All other values are reserved.

This field is ignored when ERRERICR2.MemAttr specifies any of the following memory types:

• Any Device memory type.
• Normal memory, Inner Non-cacheable, Outer Non-cacheable.

All Device and Normal Inner Non-cacheable Outer Non-cacheable memory regions are always treated as Outer Shareable.

The reset behavior of this field is:

• On an Error recovery reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Shareability.

The Shareability domain for message signaled interrupts is IMPLEMENTATION DEFINED.

MemAttr, bits [3:0]

When the component supports configuring the memory type for message signaled interrupts:

Memory type. Defines the memory type and attributes for message signaled interrupts.

0b0000 Device-nGnRnE memory.
0b0001 Device-nGnRE memory.
0b0010 Device-nGRE memory.
0b0011 Device-GRE memory.
0b0101 Normal memory, Inner Non-cacheable, Outer Non-cacheable.
0b0110 Normal memory, Inner Write-Through, Outer Non-cacheable.
0b0111 Normal memory, Inner Write-Back, Outer Non-cacheable.
0b1001 Normal memory, Inner Non-cacheable, Outer Write-Through.
0b1010 Normal memory, Inner Write-Through, Outer Non-cacheable.
0b1011 Normal memory, Inner Write-Back, Outer Write-Through.
0b1101 Normal memory, Inner Non-cacheable, Outer Write-Back.
0b1110 Normal memory, Inner Write-Through, Outer Write-Back.
0b1111 Normal memory, Inner Write-Back, Outer Write-Back.

All other values are reserved.
--- Note ---
This is the same format as the VMSAv8-64 stage 2 memory region attributes.

The reset behavior of this field is:
- On an Error recovery reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.
Memory type.
The memory type used for message signaled interrupts is IMPLEMENTATION DEFINED.

*When the implementation does not use the recommended layout for the ERRIRQCR<n> registers:*

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [31:0]**

**IMPLEMENTATION DEFINED.**

**Accessing the ERRERICR2:**

ERRERICR2 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xE9C</td>
<td>ERRERICR2</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW,
I5.8.14 ERRFHICR0, Fault Handling Interrupt Configuration Register 0

The ERRFHICR0 characteristics are:

**Purpose**

Fault Handling Interrupt configuration register.

**Configurations**

This register is present only when (the Fault Handling Interrupt is implemented or the implementation does not use the recommended layout for the ERRIRQCR<n> registers) and interrupt configuration registers are implemented. Otherwise, direct accesses to ERRFHICR0 are RES0.

ERRFHICR0 is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRFHICR0 is a 64-bit register.

**Field descriptions**

*When the Fault Handling Interrupt is implemented and the implementation uses the recommended layout for the ERRIRQCR<n> registers:*

![Diagram of ERRFHICR0 with fields and bit allocations]

**Bits [63:56]**

Reserved, RES0.

**ADDR, bits [55:2]**

Message Signaled Interrupt address. (ERRFHICR0.ADDR << 2) is the address that the component writes to when signaling the Fault Handling Interrupt. Bits [1:0] of the address are always zero.

The physical address size supported by the component is IMPLEMENTATION DEFINED.

Unimplemented high-order physical address bits are RES0.

The reset behavior of this field is:

- On an Error recovery reset, this field resets to an architecturally UNKNOWN value.

**Bits [1:0]**

Reserved, RES0.

*When the implementation does not use the recommended layout for the ERRIRQCR<n> registers:*

![Diagram of IMPLEMENTATION DEFINED with fields and bit allocations]

**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.
**Accessing the ERRFHICR0:**

ERRFHICR0 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xE80</td>
<td>ERRFHICR0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.8.15 ERRFHICR1, Fault Handling Interrupt Configuration Register 1

The ERRFHICR1 characteristics are:

**Purpose**
Fault Handling Interrupt configuration register.

**Configurations**
This register is present only when (the Fault Handling Interrupt is implemented or the implementation does not use the recommended layout for the ERRIRQCR<n> registers) and interrupt configuration registers are implemented. Otherwise, direct accesses to ERRFHICR1 are RES0.

ERRFHICR1 is implemented only as part of a memory-mapped group of error records.

**Attributes**
ERRFHICR1 is a 32-bit register.

**Field descriptions**

*When the Fault Handling Interrupt is implemented and the implementation uses the recommended layout for the ERRIRQCR<n> registers:*

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td></td>
</tr>
</tbody>
</table>

DATA, bits [31:0]
Payload for the message signaled interrupt.
The reset behavior of this field is:
• On an Error recovery reset, this field resets to an architecturally UNKNOWN value.

*When the implementation does not use the recommended layout for the ERRIRQCR<n> registers:*

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
</tbody>
</table>

IMPLEMENTATION DEFINED, bits [31:0]
IMPLEMENTATION DEFINED.

**Accessing the ERRFHICR1:**
ERRFHICR1 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xE88</td>
<td>ERRFHICR1</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.8.16 ERRFHICR2, Fault Handling Interrupt Configuration Register 2

The ERRFHICR2 characteristics are:

**Purpose**
Fault Handling Interrupt control and configuration register.

**Configurations**
This register is present only when (the Fault Handling Interrupt is implemented or the implementation does not use the recommended layout for the ERRIRQCR<\(n\) registers) and interrupt configuration registers are implemented. Otherwise, direct accesses to ERRFHICR2 are RES0.

ERRFHICR2 is implemented only as part of a memory-mapped group of error records.

**Attributes**
ERRFHICR2 is a 32-bit register.

**Field descriptions**
*When the Fault Handling Interrupt is implemented and the implementation uses the recommended layout for the ERRIRQCR<\(n\) registers:*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:8]</td>
<td>Reserved, RES0.</td>
<td>0</td>
</tr>
<tr>
<td>[7]</td>
<td>IRQEN, bit</td>
<td>0 1</td>
</tr>
</tbody>
</table>

**IRQEN, bit [7]***
*When the component supports disabling message signaled interrupts:*
Message signaled interrupt enable. Enables generation of message signaled interrupts.

0b0 Disabled.

0b1 Enabled.

The reset behavior of this field is:
* On an Error recovery reset, this field resets to 0.

**Otherwise:**
Reserved, RES0.
Message signaled interrupt enable.
Message signaled interrupts are always enabled.

**NSMSI, bit [6]***
*When the component supports configuring the Security attribute for message signaled interrupts and the component does not allow Non-secure writes to ERRFHICR2:*
Security attribute. Defines the physical address space for message signaled interrupts.

0b0 Secure.

0b1 Non-secure.

The reset behavior of this field is:
* On a Error recovery reset, this field resets to an IMPLEMENTATION DEFINED value.
When the component allows Non-secure writes to ERRFHIR2:
Reserved, RES0.
Security attribute. Defines the physical address space for message signaled interrupts.
The Security attribute used for message signaled interrupts is Non-secure.

Otherwise:
Reserved, RES0.
Security attribute. Defines the physical address space for message signaled interrupts.
The Security attribute for message signaled interrupts is IMPLEMENTATION DEFINED.

SH, bits [5:4]

When the component supports configuring the Shareability domain for message signaled interrupts:
Shareability. Defines the Shareability domain for message signaled interrupts.
0b00 Not shared.
0b10 Outer Shareable.
0b11 Inner Shareable.
All other values are reserved.
This field is ignored when ERRFHIR2.MemAttr specifies any of the following memory types:
• Any Device memory type.
• Normal memory, Inner Non-cacheable, Outer Non-cacheable.
All Device and Normal Inner Non-cacheable Outer Non-cacheable memory regions are always treated as Outer Shareable.
The reset behavior of this field is:
• On an Error recovery reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.
Shareability.
The Shareability domain for message signaled interrupts is IMPLEMENTATION DEFINED.

MemAttr, bits [3:0]

When the component supports configuring the memory type for message signaled interrupts:
Memory type. Defines the memory type and attributes for message signaled interrupts.
0b0000 Device-nGnRnE memory.
0b0001 Device-nGnRE memory.
0b0010 Device-nGRE memory.
0b0011 Device-GRE memory.
0b0101 Normal memory, Inner Non-cacheable, Outer Non-cacheable.
0b0110 Normal memory, Inner Write-Through, Outer Non-cacheable.
0b0111 Normal memory, Inner Write-Back, Outer Non-cacheable.
0b1001 Normal memory, Inner Non-cacheable, Outer Write-Through.
0b1010 Normal memory, Inner Write-Through, Outer Write-Through.
0b1011 Normal memory, Inner Write-Back, Outer Write-Through.
0b1101 Normal memory, Inner Non-cacheable, Outer Write-Back.
0b1110 Normal memory, Inner Write-Through, Outer Write-Back.
0b1111 Normal memory, Inner Write-Back, Outer Write-Back.
All other values are reserved.
--- Note ---
This is the same format as the VMAv8-64 stage 2 memory region attributes.

The reset behavior of this field is:
• On an Error recovery reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**
Reserved, RES0.
Memory type.
The memory type used for message signaled interrupts is **IMPLEMENTATION DEFINED**.

*When the implementation does not use the recommended layout for the ERRIRQCR<n> registers:*

```
+----------------+-------------------+
| 31 | 0 | IMPLEMENTATION DEFINED |
+----------------+-------------------+
```

**IMPLEMENTATION DEFINED, bits [31:0]**

**IMPLEMENTATION DEFINED.**

**Accessing the ERRFHICR2:**
ERRFHICR2 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xE8C</td>
<td>ERRFHICR2</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.8.17 ERRGSR, Error Group Status Register

The ERRGSR characteristics are:

**Purpose**

Shows the status for the records in the group.

**Configurations**

ERRGSR is implemented only as part of a memory-mapped group of error records.

This manual describes a group of error records accessed via a standard 4KB memory-mapped peripheral. For a 4KB peripheral, up to 24 error records can be accessed if the Common Fault Injection Model is implemented, and up to 56 otherwise.

**Attributes**

ERRGSR is a 64-bit register.

**Field descriptions**

![ERRGSR Register Diagram]

- **Bits [63:56]**
  - Reserved, RES0.

- **S<\(m\)>**, bit [\(m\)], for \(m = 55\) to 0

  *When error record \(<m>\) is implemented and error record \(<m>\) supports this type of reporting:*

  - The status for error record \(<m>\). A read-only copy of ERR<\(n\)>STATUS.V.
  - 0b0 No error.
  - 0b1 One or more errors.

  If the Common Fault Injection Model is implemented, up-to 24 records can be implemented meaning bits [55:24] are RES0.

**Accessing the ERRGSR:**

ERRGSR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xE00</td>
<td>ERRGSR</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.8.18  ERRIIDR, Implementation Identification Register

The ERRIIDR characteristics are:

**Purpose**

Defines the implementer of the component.

**Configurations**

Implementation of this register is OPTIONAL.

This register is present only when RAS System Architecture v1.1 is implemented.

**Attributes**

ERRIIDR is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31-20 | ProductID      | Part number, bits [11:0]. The part number is selected by the designer of the component.  
            |                 | If ERRPIDR0 and ERRPIDR1 are implemented, ERRPIDR0.PART_0 matches bits [7:0] of ERRIIDR.ProductID and ERRPIDR1.PART_1 matches bits [11:8] of ERRIIDR.ProductID.  |
| 19-16 | Variant        | Component major revision. This field distinguishes product variants or major revisions of the product.  
            |                 | If ERRPIDR2 is implemented, ERRPIDR2.REVISION matches ERRIIDR.Variant.  |
| 15-12 | Revision       | Component minor revision. This field distinguishes minor revisions of the product.  
            |                 | If ERRPIDR3 is implemented, ERRPIDR3.REVAND matches ERRIIDR.Revision.  |
| 11-0  | Implementer    | Contains the JEP106 code of the company that implemented the RAS component. For an Arm implementation, this field has the value 0x43B.  
            |                 | Bits [11:8] contain the JEP106 continuation code of the implementer, and bits [6:0] contain the JEP106 identity code of the implementer. Bit 7 is RES0.  
            |                 | If ERRPIDR4 is implemented, ERRPIDR2 is implemented, and ERRPIDR1 is implemented, ERRPIDR4.DES_2 matches bits [11:8] of ERRIIDR.Implementer, ERRPIDR2.DES_1 matches bits [6:4] of ERRIIDR.Implementer, and ERRPIDR1.DES_0 matches bits [3:0] of ERRIIDR.Implementer.  |

**Accessing the ERRIIDR:**

ERRIIDR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xE10</td>
</tr>
</tbody>
</table>
Accesses to this interface are RO.
5.8.19 ERRIMPDEF<\(n\)>, IMPLEMENTATION DEFINED Register \(n\), \(n = 0 \text{ - } 191\)

The ERRIMPDEF<\(n\)> characteristics are:

**Purpose**

IMPLEMENTATION DEFINED RAS extensions.

**Configurations**

This register is present only when the Common Fault Injection Model Extension is not implemented, UInt(ERRDEVID.NUM) <= 32 and an implementation implements ERRIMPDEF<\(n\)>. Otherwise, direct accesses to ERRIMPDEF<\(n\)> are RES0.

**Attributes**

ERRIMPDEF<\(n\)> is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>31-0</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED.

**Accessing the ERRIMPDEF<\(n\)>:**

ERRIMPDEF<\(n\)> can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0x800 + (8 * (n))</td>
<td>ERRIMPDEF&lt;(n)&gt;</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
ERRIRQCR<\text{n}>, Generic Error Interrupt Configuration Register <\text{n}>, \text{n} = 0 - 15

The ERRIRQCR<\text{n}> characteristics are:

**Purpose**

The ERRIRQCR<\text{n}> registers are reserved for IMPLEMENTATION DEFINED interrupt configuration registers.

The architecture provides a recommended layout for the ERRIRQCR<\text{n}> registers. These registers are named:

- ERRFHIR0, ERRFHIR1, and ERRFHIR2 for the fault handling interrupt controls.
- ERRERI0, ERRERI1, and ERRERI2 for the error recovery interrupt controls.
- ERRIRICR0, ERRIRICR1, and ERRIRICR2 for the critical error interrupt controls.
- ERRIRQSR for the status register.

This section describes the generic, IMPLEMENTATION DEFINED, format.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

This register is present only when the interrupt configuration registers are implemented. Otherwise, direct accesses to ERRIRQCR<\text{n}> are RES0.

ERRIRQCR<\text{n}> is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRIRQCR<\text{n}> is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
<tr>
<td>31-0</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED controls. The content of these registers is IMPLEMENTATION DEFINED.

**Accessing the ERRIRQCR<\text{n}>**:

ERRIRQCR<\text{n}> can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xE80 + (8 * \text{n})</td>
<td>ERRIRQCR&lt;\text{n}&gt;</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
### 5.8.21 ERRIRQSR, Error Interrupt Status Register

The ERRIRQSR characteristics are:

**Purpose**
Interrupt status register.

**Configurations**
This register is present only when interrupt configuration registers are implemented. Otherwise, direct accesses to ERRIRQSR are RES0.

ERRIRQSR is implemented only as part of a memory-mapped group of error records.

**Attributes**
ERRIRQSR is a 64-bit register.

**Field descriptions**

*When the implementation uses the recommended layout for the ERRIRQCR<\textit{n}> registers:*

![Register Diagram]

**Bits [63:6]**
Reserved, RES0.

**CRIERR, bit [5]**

*When the Critical Error Interrupt is implemented:*
Critical Error Interrupt Error.

- **0b0** Critical Error Interrupt write has not returned an error since this field was last cleared to zero.
- **0b1** Critical Error Interrupt write has returned an error since this field was last cleared to zero.

The reset behavior of this field is:
- On an Error recovery reset, this field resets to an architecturally UNKNOWN value.
Access to this field is W1C.

*Otherwise:*
Reserved, RES0.

**CRI, bit [4]**

*When the Critical Error Interrupt is implemented:*
Critical Error Interrupt write in progress.

- **0b0** Critical Error Interrupt write not in progress.
- **0b1** Critical Error Interrupt write in progress.

Software must not disable an interrupt whilst the write is in progress.
Note

This field does not indicate whether an interrupt is active, but rather whether a write triggered by the interrupt is in progress.

To determine whether an interrupt is active, software must examine the individual ERR<n>STATUS registers.

Access to this field is RO.

Otherwise:
Reserved, RES0.

ERIERR, bit [3]

When the Error Recovery Interrupt is implemented:

Error Recovery Interrupt Error.

0b0  Error Recovery Interrupt write has not returned an error since this field was last cleared to zero.

0b1  Error Recovery Interrupt write has returned an error since this field was last cleared to zero.

The reset behavior of this field is:

•  On an Error recovery reset, this field resets to an architecturally UNKNOWN value.

Access to this field is W1C.

Otherwise:
Reserved, RES0.

ERI, bit [2]

When the Error Recovery Interrupt is implemented:

Error Recovery Interrupt write in progress.

0b0  Error Recovery Interrupt write not in progress.

0b1  Error Recovery Interrupt write in progress.

Software must not disable an interrupt whilst the write is in progress.

Note

This field does not indicate whether an interrupt is active, but rather whether a write triggered by the interrupt is in progress.

To determine whether an interrupt is active, software must examine the individual ERR<n>STATUS registers.

Access to this field is RO.

Otherwise:
Reserved, RES0.

FHIERR, bit [1]

When the Fault Handling Interrupt is implemented:

Fault Handling Interrupt Error.

0b0  Fault Handling Interrupt write has not returned an error since this field was last cleared to zero.

0b1  Fault Handling Interrupt write has returned an error since this field was last cleared to zero.

The reset behavior of this field is:

•  On an Error recovery reset, this field resets to an architecturally UNKNOWN value.
Access to this field is W1C.

Otherwise:
Reserved, RES0.

FHI, bit [0]

When the Fault Handling Interrupt is implemented:
Fault Handling Interrupt write in progress.
0b0 Fault Handling Interrupt write not in progress.
0b1 Fault Handling Interrupt write in progress.
Software must not disable an interrupt whilst the write is in progress.

--- Note ---
This field does not indicate whether an interrupt is active, but rather whether a write triggered by
the interrupt is in progress.
To determine whether an interrupt is active, software must examine the individual ERR<n>STATUS
registers.

Otherwise:
Reserved, RES0.

When the implementation does not use the recommended layout for the ERRIRQCR<n> registers:

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
</tbody>
</table>

IMPLEMENTATION DEFINED, bits [63:0]
IMPLEMENTATION DEFINED.

Accessing the ERRIRQSR:
ERRIRQSR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xEF8</td>
<td>ERRIRQSR</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.8.22 ERR<n>ADDR, Error Record <n> Address Register, n = 0 - 65534

The ERR<n>ADDR characteristics are:

**Purpose**

If an address is associated with a detected error, then it is written to ERR<n>ADDR when the error is recorded. It is IMPLEMENTATION DEFINED how the recorded address maps to the software-visible physical address. Software might have to reconstruct the actual physical addresses using the identity of the node and knowledge of the system.

**Configurations**

This register is present only when error record <n> is implemented and error record <n> includes an address associated with an error. Otherwise, direct accesses to ERR<n>ADDR are RES0.

ERR<n>FR describes the features implemented by the node that owns error record <n>. <q> is the index of the first error record owned by the same node as error record <n>. If the node owns a single record, then q = n.

**Attributes**

ERR<n>ADDR is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS</td>
<td>Non-secure attribute. With ERR&lt;n&gt;ADDR.NSE, indicates the physical address space of the recorded location.</td>
</tr>
<tr>
<td>SI</td>
<td>Secure address.</td>
</tr>
<tr>
<td>AI</td>
<td>Address in a Realm.</td>
</tr>
<tr>
<td>VA</td>
<td>Address in a VMA.</td>
</tr>
<tr>
<td>RES0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>PADDR</td>
<td>Physical address.</td>
</tr>
</tbody>
</table>

**NS, bit [63]**

When FEAT_RME is implemented:

- Non-secure attribute. With ERR<n>ADDR.NSE, indicates the physical address space of the recorded location.
- 0b0 When ERR<n>ADDR.NSE == 0: ERR<n>ADDR.PADDR is a Secure address.
- 0b1 When ERR<n>ADDR.NSE == 1: ERR<n>ADDR.PADDR is a Root address.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When FEAT_RME is not implemented:

- Non-secure attribute.
- 0b0 ERR<n>ADDR.PADDR is a Secure address.
- 0b1 ERR<n>ADDR.PADDR is a Non-secure address.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.
SI, bit [62]

When FEAT_RME is implemented:

Secure Incorrect. Indicates whether ERR<n>ADDR.{NS, NSE} are valid.

0b0 ERR<n>ADDR.{NS, NSE} are correct. That is, they match the programmers' view of the physical address space for the recorded location.

0b1 ERR<n>ADDR.{NS, NSE} might not be correct, and might not match the programmers' view of the physical address space for the recorded location.

It is IMPLEMENTATION DEFINED whether this field is read-only or read/write.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

When FEAT_RME is not implemented:

Secure Incorrect. Indicates whether ERR<n>ADDR.NS is valid.

0b0 ERR<n>ADDR.NS is correct. That is, it matches the programmers' view of the Non-secure attribute for the recorded location.

0b1 ERR<n>ADDR.NS might not be correct, and might not match the programmers' view of the Non-secure attribute for the recorded location.

It is IMPLEMENTATION DEFINED whether this field is read-only or read/write.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AI, bit [61]

Address Incorrect. Indicates whether ERR<n>ADDR.PADDR is a valid physical address that is known to match the programmers' view of the physical address for the recorded location.

0b0 ERR<n>ADDR.PADDR is a valid physical address. That is, it matches the programmers' view of the physical address for the recorded location.

0b1 ERR<n>ADDR.PADDR might not be a valid physical address, and might not match the programmers' view of the physical address for the recorded location.

It is IMPLEMENTATION DEFINED whether this field is read-only or read/write.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

VA, bit [60]

Virtual Address. Indicates whether ERR<n>ADDR.PADDR field is a virtual address.

0b0 ERR<n>ADDR.PADDR is not a virtual address.

0b1 ERR<n>ADDR.PADDR is a virtual address.

No context information is provided for the virtual address. When ERR<n>ADDR.VA is 1, ERR<n>ADDR.{NS, SI, AI} read as {0, 1, 1}.

Support for this field is optional. If this field is not implemented and ERR<n>ADDR.PADDR field is a virtual address, then ERR<n>ADDR.{NS, SI, AI} read as {0, 1, 1}.

It is IMPLEMENTATION DEFINED whether this field is read-only or read/write.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

NSE, bit [59]

When FEAT_RME is implemented:

Physical Address Space. Together with ERR<n>ADDR.NS, indicates the address space for ERR<n>ADDR.PADDR.
The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

Bits [58:56]
Reserved, RES0.

PADDR, bits [55:0]
Physical Address. Address of the recorded location. If the physical address size implemented by this component is smaller than the size of this field, then high-order bits are unimplemented and either RES0 or have a fixed read-only IMPLEMENTATION DEFINED value. Low-order address bits might also be unimplemented and RES0, for example, if the physical address is always aligned to the size of a protection granule.
The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing the ERR<n>ADDR:
ERR<n>ADDR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0x018 + (64 * n)</td>
<td>ERR&lt;n&gt;ADDR</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:
- When the Common Fault Injection Model Extension is implemented by the node that owns this error record, ERR<q>PFGF.AV == 0 and ERR<n>STATUS.AV == 1 accesses to this register are RO.
- When the Common Fault Injection Model Extension is not implemented by the node that owns this error record and ERR<n>STATUS.AV == 1 accesses to this register are RO.
- Otherwise accesses to this register are RW.
5.8.23 ERR<n>CTLR, Error Record <n> Control Register, n = 0 - 65534

The ERR<n>CTLR characteristics are:

**Purpose**

The error control register contains enable bits for the node that writes to this record:
- Enabling error detection and correction.
- Enabling the critical error, error recovery, and fault handling interrupts.
- Enabling in-band error response for uncorrected errors.

For each bit, if the node does not support the feature, then the bit is RES0. The definition of each record is IMPLEMENTATION DEFINED.

**Configurations**

This register is present only when error record <n> is implemented and error record <n> is the first error record owned by a node. Otherwise, direct accesses to ERR<n>CTLR are RES0.

ERR<n>FR describes the features implemented by the node.

**Attributes**

ERR<n>CTLR is a 64-bit register.

**Field descriptions**

**IMPLEMENTATION DEFINED, bits [63:32]**

Reserved for IMPLEMENTATION DEFINED controls. Must permit SBZP write policy for software.

**Bits [31:14]**

Reserved, RES0.

**CI, bit [13]**

*When ERR<n>FR.CI == 0b10:*

Critical error interrupt enable. When enabled, the critical error interrupt is generated for a critical error condition.

- 0b0 Critical error interrupt not generated for critical errors. Critical errors are treated as Uncontained errors.
- 0b1 Critical error interrupt generated for critical errors.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.
Bit [12]

Reserved, RES0.

WDUI, bit [11]

When $ERR<n>FR.DUI == 0b11$:

Error recovery interrupt for Deferred errors on writes enable.

When enabled, the error recovery interrupt is generated for errors recorded as Deferred error on writes.

$0b0$  Error recovery interrupt not generated for Deferred errors on writes.

$0b1$  Error recovery interrupt generated for Deferred errors on writes.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

DUI, bit [10]

When $ERR<n>FR.DUI == 0b10$:

$DUI$

Error recovery interrupt for Deferred errors enable.

When $ERR<n>FR.DUI == 0b10$, this control applies to errors arising from both reads and writes.

When enabled, the error recovery interrupt is generated for all errors recorded as Deferred error.

$0b0$  Error recovery interrupt not generated for Deferred errors.

$0b1$  Error recovery interrupt generated for Deferred errors.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When $ERR<n>FR.DUI == 0b11$:

$RDUI$

Error recovery interrupt for Deferred errors on reads enable.

When $ERR<n>FR.DUI == 0b11$, this field is named RDUI.

When enabled, the error recovery interrupt is generated for errors recorded as Deferred error on reads.

$0b0$  Error recovery interrupt not generated for Deferred errors on reads.

$0b1$  Error recovery interrupt generated for Deferred errors on reads.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

WCFI, bit [9]

When $ERR<n>FR.CFI == 0b11$:

Fault handling interrupt for Corrected errors on writes enable.
When enabled:
- If the node implements Corrected error counters for writes, then the fault handling interrupt is generated when a counter overflows and the overflow bit for the counter is set to 1. For more information, see ERR<n>MISC0.
- Otherwise, the fault handling interrupt is also generated for errors recorded as Corrected error on writes.

0b0  Fault handling interrupt not generated for Corrected errors on writes.
0b1  Fault handling interrupt generated for Corrected errors on writes.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

CFI, bit [8]

**When ERR<n>FR.CFI == 0b10:**

CFI
Fault handling interrupt for Corrected errors enable.

When ERR<n>FR.CFI == 0b10, this control applies to errors arising from both reads and writes.

When enabled:
- If the node implements Corrected error counters, then the fault handling interrupt is generated when a counter overflows and the overflow bit for the counter is set to 1. For more information, see ERR<n>MISC0.
- Otherwise, the fault handling interrupt is also generated for all errors recorded as Corrected error.

0b0  Fault handling interrupt not generated for Corrected errors.
0b1  Fault handling interrupt generated for Corrected errors.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**When ERR<n>FR.CFI == 0b11:**

RCFI
Fault handling interrupt for Corrected errors on reads enable.

When ERR<n>FR.CFI == 0b11, this field is named RCFI.

When enabled:
- If the node implements Corrected error counters for reads, then the fault handling interrupt is generated when a counter overflows and the overflow bit for the counter is set to 1. For more information, see ERR<n>MISC0.
- Otherwise, the fault handling interrupt is also generated for errors recorded as Corrected error on reads.

0b0  Fault handling interrupt not generated for Corrected errors on reads.
0b1  Fault handling interrupt generated for Corrected errors on reads.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.
Otherwise:
Reserved, RES0.

**WUE, bit [7]**

*When \( ERR_n<FR.UE >= 0b11 \):*

In-band error response on writes enable.

When enabled, responses to writes that detect an error that is not corrected and is not deferred are signaled with an in-band error response (External Abort).

It is IMPLEMENTATION DEFINED whether an uncorrected error that is deferred and recorded as Deferred error, but is not deferred to the Requester, will signal an in-band error response to the Requester.

0b0 In-band error response for uncorrected errors on writes disabled.

0b1 In-band error response for uncorrected errors on writes enabled.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally **UNKNOWN** value.

Otherwise:
Reserved, RES0.

**WFI, bit [6]**

*When \( ERR_n<FR.FI >= 0b11 \):*

Fault handling interrupt on writes enable.

When enabled:
- The fault handling interrupt is generated for errors recorded as either Deferred error or Uncorrected error on writes.
- If the corresponding fault handling interrupt for Corrected errors control is not implemented:
  - If the node implements Corrected error counters for writes, then the fault handling interrupt is also generated when a counter overflows and the overflow bit for the counter is set to 1.
  - Otherwise, the fault handling interrupt is also generated for errors recorded as Corrected error on writes.

0b0 Fault handling interrupt on writes disabled.

0b1 Fault handling interrupt on writes enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally **UNKNOWN** value.

Otherwise:
Reserved, RES0.

**WUI, bit [5]**

*When \( ERR_n<FR.UI >= 0b11 \):*

Uncorrected error recovery interrupt on writes enable.

When enabled, the error recovery interrupt is generated for errors recorded as Uncorrected error on writes.

0b0 Error recovery interrupt on writes disabled.

0b1 Error recovery interrupt on writes enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.
The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**UE, bit [4]**

*When ERR<\text{n}>FR.UE == 0b10:*  
**UE**  
In-band error response enable.  
When ERR<\text{n}>FR.UE == 0b10, this control applies to errors arising from both reads and writes. When enabled, responses to transactions that detect an error that is not corrected and is not deferred are signaled with an in-band error response (External Abort). It is IMPLEMENTATION DEFINED whether an uncorrected error that is deferred and recorded as Deferred error, but is not deferred to the Requester, will signal an in-band error response to the Requester.  
0b0  In-band error response for uncorrected errors disabled.  
0b1  In-band error response for uncorrected errors enabled.  

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

*When ERR<\text{n}>FR.UE == 0b11:*  
**RUE**  
In-band error response on reads enable.  
When ERR<\text{n}>FR.UE == 0b11, this field is named RUE. When enabled, responses to reads that detect an error that is not corrected and is not deferred are signaled with an in-band error response (External Abort). It is IMPLEMENTATION DEFINED whether an uncorrected error that is deferred and recorded as Deferred error, but is not deferred to the Requester, will signal an in-band error response to the Requester.  
0b0  In-band error response for uncorrected errors on reads disabled.  
0b1  In-band error response for uncorrected errors on reads enabled.  

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**FI, bit [3]**

*When ERR<\text{n}>FR.FI == 0b10:*  
**FI**  
Fault handling interrupt enable.  
When ERR<\text{n}>FR.FI == 0b10, this control applies to errors arising from both reads and writes. When enabled:

- The fault handling interrupt is generated for all errors recorded as either Deferred error or Uncorrected error.
- If the fault handling interrupt for Corrected errors control is not implemented:
  — If the node implements Corrected error counters, then the fault handling interrupt is also generated when a counter overflows and the overflow bit for the counter is set to 1.
— Otherwise, the fault handling interrupt is also generated for all errors recorded as Corrected error.

0b0  Fault handling interrupt disabled.
0b1  Fault handling interrupt enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

When \( ERR<n>FR.FI == 0b11 \):  

**RFI**  
Fault handling interrupt on reads enable.

When \( ERR<n>FR.FI == 0b11 \), this field is named RFI.

When enabled:
• The fault handling interrupt is generated for errors recorded as either Deferred error or Uncorrected error on reads.
• If the corresponding fault handling interrupt for Corrected errors control is not implemented:
  — If the node implements Corrected error counters for reads, then the fault handling interrupt is also generated when a counter overflows and the overflow bit for the counter is set to 1.
  — Otherwise, the fault handling interrupt is also generated for errors recorded as Corrected error on reads.

0b0  Fault handling interrupt on reads disabled.
0b1  Fault handling interrupt on reads enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**UI, bit [2]**

When \( ERR<n>FR.UI == 0b10 \):

**UI**  
Uncorrected error recovery interrupt enable.

When \( ERR<n>FR.UI == 0b10 \), this control applies to errors arising from both reads and writes.
When enabled, the error recovery interrupt is generated for all errors recorded as Uncorrected error.

0b0  Error recovery interrupt disabled.
0b1  Error recovery interrupt enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

When \( ERR<n>FR.UI == 0b11 \):

**RUI**  
Uncorrected error recovery interrupt on reads enable.

When \( ERR<n>FR.UI == 0b11 \), this field is named RUI.
When enabled, the error recovery interrupt is generated for errors recorded as Uncorrected error on reads.

- **0b0** Error recovery interrupt on reads disabled.
- **0b1** Error recovery interrupt on reads enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally **UNKNOWN** value.

**Otherwise:**
Reserved, RES0.

**IMPLEMENTATION DEFINED, bit [1]**
Reserved for **IMPLEMENTATION DEFINED** controls. Must permit SBZP write policy for software.

**ED, bit [0]**

*When ERR<n>FR.ED == 0b10:*  
Error reporting and logging enable. When disabled, the node behaves as if error detection and correction are disabled, and no errors are recorded or signaled by the node. Arm recommends that, when disabled, correct error detection and correction codes are written for writes, unless disabled by an **IMPLEMENTATION DEFINED** control for error injection.

- **0b0** Error reporting disabled.
- **0b1** Error reporting enabled.

It is **IMPLEMENTATION DEFINED** whether the node fully disables error detection and correction when reporting is disabled. That is, even with error reporting disabled, the node might continue to silently correct errors. Uncorrected errors might result in corrupt data being silently propagated by the node.

**Note**
If this node requires initialization after Cold reset to prevent signaling false errors, then Arm recommends this field is set to 0 on Cold reset, meaning errors are not reported from Cold reset. This allows boot software to initialize a node without signaling errors. Software can enable error reporting after the node is initialized. Otherwise, the Cold reset value is **IMPLEMENTATION DEFINED**.

If the Cold reset value is 1, the reset values of other controls in this register are also **IMPLEMENTATION DEFINED** and should not be **UNKNOWN**.

The reset behavior of this field is:
- On a Cold reset, this field resets to an **IMPLEMENTATION DEFINED** value.

**Otherwise:**
Reserved, RES0.

**Accessing the ERR<n>CTLR:**

ERR<n>CTLR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0x008 + (64 * n)</td>
<td>ERR&lt;n&gt;CTLR</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
### 5.8.24 ERR<n>FR, Error Record <n> Feature Register, n = 0 - 65534

The ERR<n>FR characteristics are:

**Purpose**

Defines whether <n> is the first record owned by a node:

- If <n> is the first error record owned by a node, then ERR<n>FR.ED is not 0b00.
- If <n> is not the first error record owned by a node, then ERR<n>FR.ED is 0b00.

If <n> is the first record owned by the node, defines which of the common architecturally-defined features are implemented by the node and, of the implemented features, which are software programmable.

**Configurations**

This register is present only when error record <n> is implemented. Otherwise, direct accesses to ERR<n>FR are RES0.

**Attributes**

ERR<n>FR is a 64-bit register.

**Field descriptions**

*When error record <n> is not the first error record owned by the node:*

![Diagram](image)

- **Bits [63:2]**
  
  Reserved, RES0.

- **ED, bits [1:0]**
  
  Error reporting and logging. Indicates error record <n> is not the first error record owned the node.
  
  - 0b00 Error record <n> is not the first error record owned by the node.
  
  This field reads as 0b00.

*When error record <n> is the first error record owned by the node:*

![Diagram](image)

- **IMPLEMENTATION DEFINED, bits [63:55]**
  
  When ERR<n>FR.FRX != 1:
  
  Reserved for identifying IMPLEMENTATION DEFINED controls.
Otherwise:
Reserved, RES0.

CE, bits [54:53]
When ERR<\text{n}>FR.FRX == 1:
Corrected Error recording. Describes the types of Corrected errors the node can record, if any.
- 0b00: Does not record Corrected errors.
- 0b01: Records only transient or persistent Corrected errors. That is, Corrected errors recorded
  by setting ERR<\text{n}>STATUS.CE to either 0b01 or 0b11.
- 0b10: Records only non-specific Corrected errors. That is, Corrected errors recorded by
  setting ERR<\text{n}>STATUS.CE to 0b10.
- 0b11: Records all types of Corrected error.
Otherwise:
Reserved for identifying IMPLEMENTATION DEFINED controls.

DE, bit [52]
When ERR<\text{n}>FR.FRX == 1:
Deferred Error recording. Describes whether the node supports recording Deferred errors.
- 0b0: Does not record Deferred errors.
- 0b1: Records Deferred errors.
Otherwise:
Reserved for identifying IMPLEMENTATION DEFINED controls.

UEO, bit [51]
When ERR<\text{n}>FR.FRX == 1:
Latent or Restartable Error recording. Describes whether the node supports recording Latent or
Restartable errors.
- 0b0: Does not record Latent or Restartable errors.
- 0b1: Records Latent or Restartable errors.
Otherwise:
Reserved for identifying IMPLEMENTATION DEFINED controls.

UER, bit [50]
When ERR<\text{n}>FR.FRX == 1:
Signaled or Recoverable Error recording. Describes whether the node supports recording Signaled
or Recoverable errors.
- 0b0: Does not record Signaled or Recoverable errors.
- 0b1: Records Signaled or Recoverable errors.
Otherwise:
Reserved for identifying IMPLEMENTATION DEFINED controls.

UEU, bit [49]
When ERR<\text{n}>FR.FRX == 1:
Unrecoverable Error recording. Describes whether the node supports recording Unrecoverable
errors.
- 0b0: Does not record Unrecoverable errors.
- 0b1: Records Unrecoverable errors.
Otherwise:
Reserved for identifying IMPLEMENTATION DEFINED controls.
UC, bit [48]

When ERR<\text{n}^>FR.FRX == 1:

Uncontainable Error recording. Describes whether the node supports recording Uncontainable errors.

0b0 Does not record Uncontainable errors.
0b1 Records Uncontainable errors.

Otherwise:

Reserved for identifying IMPLEMENTATION DEFINED controls.

IMPLEMENTATION DEFINED, bits [47:32]

Reserved for identifying IMPLEMENTATION DEFINED controls.

FRX, bit [31]

When RAS System Architecture v1.1 is implemented:

Feature Register extension. Defines whether ERR<\text{n}^>FR[63:48] are architecturally defined.

0b0 ERR<\text{n}^>FR[63:48] are IMPLEMENTATION DEFINED.
0b1 ERR<\text{n}^>FR[63:48] are defined by the architecture.

Otherwise:

Reserved, RES0.

Bits [30:26]

Reserved, RES0.

TS, bits [25:24]

Timestamp Extension. Indicates whether, for each error record <\text{m}^> owned by this node, ERR<\text{n}^>MISC3 is used as the timestamp register, and, if it is, the timebase used by the timestamp.

0b00 Does not support a timestamp register.
0b01 Implements a timestamp register in ERR<\text{n}^>MISC3 for each error record <\text{m}^> owned by the node. The timestamp uses the same timebase as the system Generic Timer.

Note

For an error record that has an affinity to a PE, this is the same timer that is visible through CNTPCT_EL0 at the highest Exception level on that PE.

0b10 Implements a timestamp register in ERR<\text{n}^>MISC3 for each error record <\text{m}^> owned by the node. The timestamp uses an IMPLEMENTATION DEFINED timebase.

All other values are reserved.

CI, bits [23:22]

Critical error interrupt. Indicates whether the critical error interrupt and associated controls are implemented by the node.

0b00 Does not support the critical error interrupt. ERR<\text{n}^>CTRL.CI is RES0.
0b01 Critical error interrupt is supported and always enabled. ERR<\text{n}^>CTRL.CI is RES0.
0b10 Critical error interrupt is supported and controllable using ERR<\text{n}^>CTRL.CI.

All other values are reserved.

INJ, bits [21:20]

Fault Injection Extension. Indicates whether the Common Fault Injection Model Extension is implemented by the node.

0b00 Does not support the Common Fault Injection Model Extension.
0b01 Supports the Common Fault Injection Model Extension. See ERR<\text{n}^>PFGF for more information.
All other values are reserved.

**CEO, bits [19:18]**

*When ERR<n>FR.CEC != 0b000:*

Corrected Error overwrite. Indicates the behavior of the node when a second or subsequent Corrected error is recorded and a first Corrected error has previously been recorded by an error record <m> owned by the node.

- **0b00** Keeps the previous error syndrome.
- **0b01** If ERR<n>STATUS.OF is 1 before the Corrected error is counted, then the error record keeps the previous syndrome. Otherwise the previous syndrome is overwritten.

All other values are reserved.

The second or subsequent Corrected error is counted by the Corrected error counter, regardless of the value of this field. If counting the error causes unsigned overflow of the counter, then ERR<n>STATUS.OF is set to 1.

This means, that if no other error is subsequently recorded that overwrites the syndrome:

- If ERR<n>FR.CEO is 0b00, the error record holds the syndrome for the first recorded Corrected error.
- If ERR<n>FR.CEO is 0b01, the error record holds the syndrome for the most recently recorded Corrected error before the counter overflows.

*Otherwise:*

Reserved, RES0.

**DUI, bits [17:16]**

*When ERR<n>FR.UI != 0b00:*

Error recovery interrupt for deferred errors control. Indicates whether the enabling and disabling of error recovery interrupts on deferred errors is supported by the node.

- **0b00** Does not support the enabling and disabling of error recovery interrupts on deferred errors. ERR<n>CTRL.DUI is RES0.
- **0b10** Enabling and disabling of error recovery interrupts on deferred errors is supported and controllable using ERR<n>CTRL.DUI.
- **0b11** Enabling and disabling of error recovery interrupts on deferred errors is supported, and controllable using ERR<n>CTRL.WDUI for writes and ERR<n>CTRL.RDUI for reads.

All other values are reserved.

*Otherwise:*

Reserved, RES0.

**RP, bit [15]**

*When ERR<n>FR.CEC != 0b000:*

Repeat counter. Indicates whether the node implements a second Corrected error counter in ERR<n>MISC0 for each error record <m> owned by the node that can record countable errors.

- **0b0** Implements a single Corrected error counter in ERR<n>MISC0 for each error record <m> owned by the node that can record countable errors.
- **0b1** Implements a first (repeat) counter and a second (other) counter in ERR<n>MISC0 for each error record <m> owned by the node that can record countable errors. The repeat counter is the same size as the primary error counter.

*Otherwise:*

Reserved, RES0.
CEC, bits [14:12]

Corrected Error Counter. Indicates whether the node implements the standard Corrected error counter mechanisms in ERR<n>MISC0 for each error record <m> owned by the node that can record countable errors.

- **0b000**: Does not implement the standard Corrected error counter model.
- **0b010**: Implements an 8-bit Corrected error counter in ERR<n>MISC0[39:32] for each error record <m> owned by the node that can record countable errors.
- **0b100**: Implements a 16-bit Corrected error counter in ERR<n>MISC0[47:32] for each error record <m> owned by the node that can record countable errors.

All other values are reserved.

--- Note ---

Implementations might include other error counter models, or might include the standard model and not indicate this in ERR<n>FR.

CFI, bits [11:10]

*When ERR<n>FR.FI != 0b00:*

Fault handling interrupt for corrected errors control. Indicates whether the enabling and disabling of fault handling interrupts on corrected errors is supported by the node.

- **0b0**: Does not support the enabling and disabling of fault handling interrupts on corrected errors. ERR<n>CTRL.CFI is RES0.
- **0b10**: Enabling and disabling of fault handling interrupts on corrected errors is supported and controllable using ERR<n>CTRL.CFI.
- **0b11**: Enabling and disabling of fault handling interrupts on corrected errors is supported, and controllable using ERR<n>CTRL.WCFI for writes and ERR<n>CTRL.RCFI for reads.

All other values are reserved.

*Otherwise:*

Reserved, RES0.

UE, bits [9:8]

In-band error response (External Abort). Indicates whether the in-band error response and associated controls are implemented by the node.

- **0b0**: Does not support the in-band error response. ERR<n>CTRL.UE is RES0.
- **0b1**: In-band error response is supported and always enabled. ERR<n>CTRL.UE is RES0.
- **0b10**: In-band error response is supported and controllable using ERR<n>CTRL.UE.
- **0b11**: In-band error response is supported, and controllable using ERR<n>CTRL.WUE for writes and ERR<n>CTRL.RUE for reads.

It is IMPLEMENTATION DEFINED whether an uncorrected error that is deferred and recorded as Deferred error, but is not deferred to the Requester, will signal an in-band error response to the Requester.

FI, bits [7:6]

Fault handling interrupt. Indicates whether the fault handling interrupt and associated controls are implemented by the node.

- **0b0**: Does not support the fault handling interrupt. ERR<n>CTRL.FI is RES0.
- **0b1**: Fault handling interrupt is supported and always enabled. ERR<n>CTRL.FI is RES0.
- **0b10**: Fault handling interrupt is supported and controllable using ERR<n>CTRL.FI.
- **0b11**: Fault handling interrupt is supported, and controllable using ERR<n>CTRL.WFI for writes and ERR<n>CTRL.RFI for reads.
UI, bits [5:4]
Error recovery interrupt for uncorrected errors. Indicates whether the error handling interrupt and associated controls are implemented by the node.
0b00  Does not support the error handling interrupt. ERR<n>CTRL.UI is RES0.
0b01  Error handling interrupt is supported and always enabled. ERR<n>CTRL.UI is RES0.
0b10  Error handling interrupt is supported and controllable using ERR<n>CTRL.UI.
0b11  Error handling interrupt is supported, and controllable using ERR<n>CTRL.WUI for writes and ERR<n>CTRL.RUI for reads.

IMPLEMENTATION DEFINED, bits [3:2]
IMPLEMENTATION DEFINED.

ED, bits [1:0]
Error reporting and logging. Indicates error record <n> is the first record owned the node, and whether the node implements the controls for enabling and disabling error reporting and logging.
0b01  Error reporting and logging always enabled. ERR<n>CTRL.ED is RES0.
0b10  Error reporting and logging is controllable using ERR<n>CTRL.ED.
All other values are reserved.

Accessing the ERR<n>FR:
ERR<n>FR can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0x000 + (64 * n)</td>
<td>ERR&lt;n&gt;-FR</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.8.25 ERR<n>MISC0, Error Record <n> Miscellaneous Register 0, n = 0 - 65534

The ERR<n>MISC0 characteristics are:

**Purpose**

IMPLEMENTATION DEFINED error syndrome register. The miscellaneous syndrome registers might contain:

- Information to locate where the error was detected.
- If the error was detected within a FRU, the identity of the FRU.
- A Corrected error counter or counters.
- Other state information not present in the corresponding status and address registers.

If the node that owns error record <n> implements architecturally-defined Corrected error counters (ERR<n>FR.CEC != 0b000), and error record <n> can record countable errors, then ERR<n>MISC0 implements the architecturally-defined Corrected error counter or counters.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

This register is present only when error record <n> is implemented. Otherwise, direct accesses to ERR<n>MISC0 are RES0.

ERR<n>FR describes the features implemented by the node that owns error record <n>. <q> is the index of the first error record owned by the same node as error record <n>. If the node owns a single record, then q = n.

For IMPLEMENTATION DEFINED fields in ERR<n>MISC0, writing zero returns the error record to an initial quiescent state.

In particular, if any IMPLEMENTATION DEFINED syndrome fields might generate a Fault Handling or Error Recovery Interrupt request, writing zero is sufficient to deactivate the Interrupt request.

Fields that are read-only, non-zero, and ignore writes are compliant with this requirement.

--- Note ---

Arm recommends that any IMPLEMENTATION DEFINED syndrome field that can generate a Fault Handling, Error Recovery, Critical, or IMPLEMENTATION DEFINED, interrupt request is disabled at Cold reset and is enabled by software writing an IMPLEMENTATION DEFINED nonzero value to an IMPLEMENTATION DEFINED field in ERR<n>CTLR.

**Attributes**

ERR<n>MISC0 is a 64-bit register.

**Field descriptions**

*When ERR<q>FR.CEC == 0b000:*

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED syndrome.
When $ERR<q>FR.CEC == 0b100$ and $ERR<q>FR.RP == 0$: 

<table>
<thead>
<tr>
<th>63</th>
<th>48</th>
<th>47</th>
<th>46</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td>OF</td>
<td>CEC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IMPLEMENTATION DEFINED, bits [63:48]
IMPLEMENTATION DEFINED syndrome.

OF, bit [47]
Sticky overflow bit. Set to 1 when $ERR<n>MISC0.CEC$ is incremented and wraps through zero.
0b0 Counter has not overflowed.
0b1 Counter has overflowed.
A direct write that modifies this field might indirectly set $ERR<n>STATUS.OF$ to an UNKNOWN value and a direct write to $ERR<n>STATUS.OF$ that clears it to zero might indirectly set this field to an UNKNOWN value.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

CEC, bits [46:32]
Corrected error count. Incremented for each Corrected error. It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether Deferred and Uncorrected errors are counted.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

When $ERR<q>FR.CEC == 0b010$ and $ERR<q>FR.RP == 0$: 

<table>
<thead>
<tr>
<th>63</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td>OF</td>
<td>CEC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IMPLEMENTATION DEFINED, bits [63:40]
IMPLEMENTATION DEFINED syndrome.

OF, bit [39]
Sticky overflow bit. Set to 1 when $ERR<n>MISC0.CEC$ is incremented and wraps through zero.
0b0 Counter has not overflowed.
0b1 Counter has overflowed.
A direct write that modifies this field might indirectly set $ERR<n>STATUS.OF$ to an UNKNOWN value and a direct write to $ERR<n>STATUS.OF$ that clears it to zero might indirectly set this field to an UNKNOWN value.
The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.
CEC, bits [38:32]
Corrected error count. Incremented for each Corrected error. It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether Deferred and Uncorrected errors are counted.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [31:0]
IMPLEMENTATION DEFINED syndrome.

When ERR<q>FR.CEC == 0b100 and ERR<q>FR.RP == 1:

OFO, bit [63]
Sticky overflow bit, other. Set to 1 when ERR<n>MISC0.CECO is incremented and wraps through zero.

0b0 Other counter has not overflowed.
0b1 Other counter has overflowed.

A direct write that modifies this field might indirectly set ERR<n>STATUS.OF to an UNKNOWN value and a direct write to ERR<n>STATUS.OF that clears it to zero might indirectly set this field to an UNKNOWN value.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

CECO, bits [62:48]
Corrected error count, other. Incremented for each countable error that is not accounted for by incrementing ERR<n>MISC0.CECR.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

OFR, bit [47]
Sticky overflow bit, repeat. Set to 1 when ERR<n>MISC0.CECR is incremented and wraps through zero.

0b0 Repeat counter has not overflowed.
0b1 Repeat counter has overflowed.

A direct write that modifies this field might indirectly set ERR<n>STATUS.OF to an UNKNOWN value and a direct write to ERR<n>STATUS.OF that clears it to zero might indirectly set this field to an UNKNOWN value.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

CECR, bits [46:32]
Corrected error count, repeat. Incremented for the first countable error, which also records other syndrome for the error, and subsequently for each countable error that matches the recorded other syndrome. Corrected errors are countable errors. It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether Deferred and Uncorrected errors are countable errors.
Note

For example, the other syndrome might include the set and way information for an error detected in a cache. This might be recorded in the IMPLEMENTATION DEFINED ERR<n>MISC<m> fields on a first Corrected error. ERR<n>MISC0.CECR is then incremented for each subsequent Corrected Error in the same set and way.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED syndrome.

When ERR<q>FR.CEC == 0b010 and ERR<q>FR.RP == 1:

<table>
<thead>
<tr>
<th>63</th>
<th>48</th>
<th>47</th>
<th>46</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OFO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OFR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IMPLEMENTATION DEFINED, bits [63:48]

IMPLEMENTATION DEFINED syndrome.

OFO, bit [47]

Sticky overflow bit, other. Set to 1 when ERR<n>MISC0.CECO is incremented and wraps through zero.

0b0 Other counter has not overflowed.
0b1 Other counter has overflowed.

A direct write that modifies this field might indirectly set ERR<n>STATUS.OF to an UNKNOWN value and a direct write to ERR<n>STATUS.OF that clears it to zero might indirectly set this field to an UNKNOWN value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

CECO, bits [46:40]

Corrected error count, other. Incremented for each countable error that is not accounted for by incrementing ERR<n>MISC0.CECR.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

OFR, bit [39]

Sticky overflow bit, repeat. Set to 1 when ERR<n>MISC0.CECR is incremented and wraps through zero.

0b0 Repeat counter has not overflowed.
0b1 Repeat counter has overflowed.

A direct write that modifies this field might indirectly set ERR<n>STATUS.OF to an UNKNOWN value and a direct write to ERR<n>STATUS.OF that clears it to zero might indirectly set this field to an UNKNOWN value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
CECR, bits [38:32]

Corrected error count, repeat. Incremented for the first countable error, which also records other syndrome for the error, and subsequently for each countable error that matches the recorded other syndrome. Corrected errors are countable errors. It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether Deferred and Uncorrected errors are countable errors.

Note

For example, the other syndrome might include the set and way information for an error detected in a cache. This might be recorded in the IMPLEMENTATION DEFINED ERR<n>MISC<m> fields on a first Corrected error. ERR<n>MISC0.CECR is then incremented for each subsequent Corrected Error in the same set and way.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED syndrome.

Accessing the ERR<n>MISC0:

Reads from ERR<n>MISC0 return an IMPLEMENTATION DEFINED value and writes have IMPLEMENTATION DEFINED behavior.

If the Common Fault Injection Mechanism is implemented by the node that owns this error record, and ERR<n>PFGF.MV is 1, then some parts of this register are read/write when ERR<n>STATUS.MV is 0. See ERR<n>PFGF.MV for more information.

For other parts of this register, or if the Common Fault Injection Mechanism is not implemented, then Arm recommends that:

- Miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write.
- When ERR<n>STATUS.MV is 1, the miscellaneous syndrome specific to the most recently recorded error ignores writes.

Note

These recommendations allow a counter to be reset in the presence of a persistent error, while preventing specific information, such as that identifying a FRU, from being lost if an error is detected while the previous error is being logged.

ERR<n>MISC0 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0x020 + (64 * n)</td>
<td>ERR&lt;n&gt;MISC0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.8.26 **ERR<n>MISC1, Error Record <n> Miscellaneous Register 1, n = 0 - 65534**

The ERR<n>MISC1 characteristics are:

**Purpose**

IMPLEMENTATION DEFINED error syndrome register. The miscellaneous syndrome registers might contain:

- Information to locate where the error was detected.
- If the error was detected within a FRU, the identity of the FRU.
- A Corrected error counter or counters.
- Other state information not present in the corresponding status and address registers.

**Configurations**

This register is present only when error record <n> is implemented. Otherwise, direct accesses to ERR<n>MISC1 are RES0.

ERR<n>FR describes the features implemented by the node that owns error record <n>. <q> is the index of the first error record owned by the same node as error record <n>. If the node owns a single record, then q = n.

For IMPLEMENTATION DEFINED fields in ERR<n>MISC1, writing zero returns the error record to an initial quiescent state.

In particular, if any IMPLEMENTATION DEFINED syndrome fields might generate a Fault Handling or Error Recovery Interrupt request, writing zero is sufficient to deactivate the Interrupt request.

**Note**

Arm recommends that any IMPLEMENTATION DEFINED syndrome field that can generate a Fault Handling, Error Recovery, Critical, or IMPLEMENTATION DEFINED, interrupt request is disabled at Cold reset and is enabled by software writing an IMPLEMENTATION DEFINED nonzero value to an IMPLEMENTATION DEFINED field in ERR<n>CTLR.

**Attributes**

ERR<n>MISC1 is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit 63</th>
<th>Bit 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td>IMPLEMENTATION DEFINED</td>
</tr>
</tbody>
</table>

**IMPLEMENTATION DEFINED, bits [63:0]**

IMPLEMENTATION DEFINED syndrome.

**Accessing the ERR<n>MISC1:**

Reads from ERR<n>MISC1 return an IMPLEMENTATION DEFINED value and writes have IMPLEMENTATION DEFINED behavior.

If the Common Fault Injection Mechanism is implemented by the node that owns this error record, and ERR<n>PFGF.MV is 1, then some parts of this register are read/write when ERR<n>STATUS.MV is 0. See ERR<n>PFGF.MV for more information.
For other parts of this register, or if the Common Fault Injection Mechanism is not implemented, then Arm recommends that:

- Miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write.
- When ERR<n>STATUS.MV is 1, the miscellaneous syndrome specific to the most recently recorded error ignores writes.

**Note**

These recommendations allow a counter to be reset in the presence of a persistent error, while preventing specific information, such as that identifying a FRU, from being lost if an error is detected while the previous error is being logged.

ERR<n>MISC1 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>$0x028 + (64 * n)$</td>
<td>ERR&lt;n&gt;MISC1</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.8.27 ERR<\text{n}>MISC2, Error Record <\text{n}> Miscellaneous Register 2, \text{n} = 0 - 65534

The ERR<\text{n}>MISC2 characteristics are:

**Purpose**

IMPLEMENTATION DEFINED error syndrome register. The miscellaneous syndrome registers might contain:

- Information to locate where the error was detected.
- If the error was detected within a FRU, the identity of the FRU.
- A Corrected error counter or counters.
- Other state information not present in the corresponding status and address registers.

**Configurations**

This register is present only when (an implementation implements ERR<\text{n}>MISC2 or RAS System Architecture v1.1 is implemented) and error record <\text{n}> is implemented. Otherwise, direct accesses to ERR<\text{n}>MISC2 are RES0.

ERR<\text{n}>FR describes the features implemented by the node that owns error record <\text{n}>. <\text{q}> is the index of the first error record owned by the same node as error record <\text{n}>. If the node owns a single record, then <\text{q} > = <\text{n}>.

For IMPLEMENTATION DEFINED fields in ERR<\text{n}>MISC2, writing zero returns the error record to an initial quiescent state.

In particular, if any IMPLEMENTATION DEFINED syndrome fields might generate a Fault Handling or Error Recovery Interrupt request, writing zero is sufficient to deactivate the Interrupt request.

Fields that are read-only, non-zero, and ignore writes are compliant with this requirement.

If RAS System Architecture v1.1 is not implemented, Arm recommends that ERR<\text{n}>MISC2 does not require zeroing to return the record to a quiescent state.

---

**Note**

Arm recommends that any IMPLEMENTATION DEFINED syndrome field that can generate a Fault Handling, Error Recovery, Critical, or IMPLEMENTATION DEFINED, interrupt request is disabled at Cold reset and is enabled by software writing an IMPLEMENTATION DEFINED nonzero value to an IMPLEMENTATION DEFINED field in ERR<\text{n}>CTLR.

---

**Attributes**

ERR<\text{n}>MISC2 is a 64-bit register.

**Field descriptions**

![ IMPLEMENTATION DEFINED, bits [63:0] ]

IMPLEMENTATION DEFINED syndrome.

**Accessing the ERR<\text{n}>MISC2:**

Reads from ERR<\text{n}>MISC2 return an IMPLEMENTATION DEFINED value and writes have IMPLEMENTATION DEFINED behavior.
If the Common Fault Injection Mechanism is implemented by the node that owns this error record, and \( \text{ERR}_n\text{-PFGF.MV} \) is 1, then some parts of this register are read/write when \( \text{ERR}_n\text{-STATUS.MV} \) is 0. See \( \text{ERR}_n\text{-PFGF.MV} \) for more information.

For other parts of this register, or if the Common Fault Injection Mechanism is not implemented, then Arm recommends that:

- Miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write.
- When \( \text{ERR}_n\text{-STATUS.MV} \) is 1, the miscellaneous syndrome specific to the most recently recorded error ignores writes.

--- Note ---

These recommendations allow a counter to be reset in the presence of a persistent error, while preventing specific information, such as that identifying a FRU, from being lost if an error is detected while the previous error is being logged.

ERR\( _n \)-MISC2 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0x030 + (64 * n)</td>
<td>( \text{ERR}_n\text{-MISC2} )</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.8.28 ERR<\(n\)>MISC3, Error Record \(<n\>\) Miscellaneous Register 3, \(n = 0 - 65534\)

The ERR<\(n\)>MISC3 characteristics are:

**Purpose**

IMPLEMENTATION DEFINED error syndrome register. The miscellaneous syndrome registers might contain:

- Information to locate where the error was detected.
- If the error was detected within a FRU, the identity of the FRU.
- A Corrected error counter or counters.
- Other state information not present in the corresponding status and address registers.

If the node that owns error record \(n\) supports the RAS Timestamp Extension (ERR<\(n\)>FR.TS \(!= 0b00\)), then ERR<\(n\)>MISC3 contains the timestamp value for error record \(n\) when the error was detected. Otherwise the contents of ERR<\(n\)>MISC3 are IMPLEMENTATION DEFINED.

For details of this, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

**Configurations**

This register is present only when (an implementation implements ERR<\(n\)>MISC3 or RAS System Architecture v1.1 is implemented) and error record \(<n\>\) is implemented. Otherwise, direct accesses to ERR<\(n\)>MISC3 are RES0.

ERR<\(n\)>FR describes the features implemented by the node that owns error record \(<n\>\). \(<q\>\) is the index of the first error record owned by the same node as error record \(<n\>\). If the node owns a single record, then \(q = n\).

For IMPLEMENTATION DEFINED fields in ERR<\(n\)>MISC3, writing zero returns the error record to an initial quiescent state.

In particular, if any IMPLEMENTATION DEFINED syndrome fields might generate a Fault Handling or Error Recovery Interrupt request, writing zero is sufficient to deactivate the Interrupt request.

Fields that are read-only, non-zero, and ignore writes are compliant with this requirement.

If RAS System Architecture v1.1 is not implemented, Arm recommends that ERR<\(n\)>MISC3 does not require zeroing to return the record to a quiescent state.

--- **Note** ---

Arm recommends that any IMPLEMENTATION DEFINED syndrome field that can generate a Fault Handling, Error Recovery, Critical, or IMPLEMENTATION DEFINED interrupt request is disabled at Cold reset and is enabled by software writing an IMPLEMENTATION DEFINED nonzero value to an IMPLEMENTATION DEFINED field in ERR<\(n\)>CTLR.

**Attributes**

ERR<\(n\)>MISC3 is a 64-bit register.

**Field descriptions**

*When ERR<\(q\)>FR.TS \(!= 0b00\):*

![TS field diagram](image-url)
TS, bits [63:0]

Timestamp. Timestamp value recorded when the error was detected. Valid only if ERR<\text{n}>STATUS.V == 1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO or RW.

When ERR<\text{q}>FR.TS == 0b00:

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>IMPLEMENTATION DEFINED</td>
<td></td>
</tr>
</tbody>
</table>

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED syndrome.

Accessing the ERR<\text{n}>MISC3:

Reads from ERR<\text{n}>MISC3 return an IMPLEMENTATION DEFINED value and writes have IMPLEMENTATION DEFINED behavior.

If the Common Fault Injection Mechanism is implemented by the node that owns this error record, and ERR<\text{n}>PFGF.MV is 1, then some parts of this register are read/write when ERR<\text{n}>STATUS.MV is 0. See ERR<\text{n}>PFGF.MV for more information.

For other parts of this register, or if the Common Fault Injection Mechanism is not implemented, then Arm recommends that:

- Miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write.
- When ERR<\text{n}>STATUS.MV is 1, the miscellaneous syndrome specific to the most recently recorded error ignores writes.

Note

These recommendations allow a counter to be reset in the presence of a persistent error, while preventing specific information, such as that identifying a FRU, from being lost if an error is detected while the previous error is being logged.

ERR<\text{n}>MISC3 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0x038 + (64 * \text{n})</td>
<td>ERR&lt;\text{n}&gt;MISC3</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
ERR<n>PFGCDN, Error Record <n> Pseudo-fault Generation Countdown Register, n = 0 - 65534

The ERR<n>PFGCDN characteristics are:

**Purpose**

Generates one of the errors enabled in the corresponding ERR<n>PFGCTL register.

For details of this, see the Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.

**Configurations**

This register is present only when error record <n> is implemented, the node implements the Common Fault Injection Model Extension (ERR<n>FR.INJ != 0b00) and error record <n> is the first error record owned by a node. Otherwise, direct accesses to ERR<n>PFGCDN are RES0. ERR<n>FR describes the features implemented by the node.

**Attributes**

ERR<n>PFGCDN is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:32]</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>[31:0]</td>
<td>CDN, bits</td>
</tr>
<tr>
<td> </td>
<td>    Software writes 1 to ERR&lt;n&gt;PFGCTL.CDNEN.</td>
</tr>
<tr>
<td> </td>
<td>    The Error Generation Counter decrements to zero and ERR&lt;n&gt;PFGCTL.R is 1.</td>
</tr>
<tr>
<td> </td>
<td>  While ERR&lt;n&gt;PFGCTL.CDNEN is 1 and the Error Generation Counter is nonzero, the counter decrements by 1 for each cycle at an IMPLEMENTATION DEFINED clock rate. When the counter reaches zero, one of the errors enabled in the ERR&lt;n&gt;PFGCTL register is generated.</td>
</tr>
</tbody>
</table>

---

**Accessing the ERR<n>PFGCDN:**

ERR<n>PFGCDN can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0x810 + (64 * n)</td>
<td>ERR&lt;n&gt;PFGCDN</td>
</tr>
</tbody>
</table>
Accesses to this interface are RW.
## 5.8.30 ERR<n>PFGCTL, Error Record <n> Pseudo-fault Generation Control Register, n = 0 - 65534

The ERR<n>PFGCTL characteristics are:

**Purpose**

Enables controlled fault generation.

**Configurations**

This register is present only when error record <n> is implemented, the node implements the Common Fault Injection Model Extension (ERR<n>FR.INJ != 0b00) and error record <n> is the first error record owned by a node. Otherwise, direct accesses to ERR<n>PFGCTL are RES0.

ERR<n>PFGF describes the Common Fault Injection features implemented by the node.

**Attributes**

ERR<n>PFGCTL is a 64-bit register.

**Field descriptions**

![Field Diagram]

**Bits [63:32]**

Reserved, RES0.

**CDNEN, bit [31]**

Countdown Enable. Controls transfers of the value held in ERR<n>PFGCDN to the Error Generation Counter and enables this counter.

- 0b0 The Error Generation Counter is disabled.
- 0b1 The Error Generation Counter is enabled. On a write of 1 to this field, the Error Generation Counter is set to ERR<n>PFGCDN.CDN.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.

**R, bit [30]**

*When the node supports this control:

Restart. Controls whether the Error Generation Counter restarts or stops counting on reaching zero.

- 0b0 On reaching zero, the Error Generation Counter will stop counting.
- 0b1 On reaching zero, the Error Generation Counter is set to ERR<n>PFGCDN.CDN.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

**Otherwise:**

Reserved, RES0.

**Bits [29:13]**

Reserved, RES0.
Bit [12]

When the node supports this control and the node always sets ERR<\(n\)>STATUS.MV to 0b1 when an injected error is recorded:

Reserved, RAO/WI.

When the node supports this control:

Miscellaneous syndrome. The value written to ERR<\(n\)>STATUS.MV when an injected error is recorded.

- 0b0: ERR<\(n\)>STATUS.MV is set to 0 when an injected error is recorded.
- 0b1: ERR<\(n\)>STATUS.MV is set to 1 when an injected error is recorded.

The reset behavior of this field is:

- On an Error recovery reset, this field resets to an architecturally UNKNOWN value.

When the node always sets ERR<\(n\)>STATUS.MV to 0b1 when an injected error is recorded and this field is RAO/WI:

Reserved, RAO/WI.

Otherwise:

Reserved, RES0.

Bit [11]

When the node supports this control, and the node always sets ERR<\(n\)>STATUS.AV to 0b1 when an injected error is recorded:

Reserved, RAO/WI.

When the node supports this control:

Address syndrome. The value written to ERR<\(n\)>STATUS.AV when an injected error is recorded.

- 0b0: ERR<\(n\)>STATUS.AV is set to 0 when an injected error is recorded.
- 0b1: ERR<\(n\)>STATUS.AV is set to 1 when an injected error is recorded.

The reset behavior of this field is:

- On an Error recovery reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PN, bit [10]

When the node supports this control:

Poison flag. The value written to ERR<\(n\)>STATUS.PN when an injected error is recorded.

- 0b0: ERR<\(n\)>STATUS.PN is set to 0 when an injected error is recorded.
- 0b1: ERR<\(n\)>STATUS.PN is set to 1 when an injected error is recorded.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ER, bit [9]

When the node supports this control:

Error Reported flag. The value written to ERR<\(n\)>STATUS.ER when an injected error is recorded.

- 0b0: ERR<\(n\)>STATUS.ER is set to 0 when an injected error is recorded.
- 0b1: ERR<\(n\)>STATUS.ER is set to 1 when an injected error is recorded.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.
Otherwise:
Reserved, RES0.

CI, bit [8]

When the node supports this control:
Critical Error flag. The value written to ERR<n>STATUS.CI when an injected error is recorded.

0b0    ERR<n>STATUS.CI is set to 0 when an injected error is recorded.
0b1    ERR<n>STATUS.CI is set to 1 when an injected error is recorded.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

CE, bits [7:6]

When the node supports this control:
Corrected Error generation enable. Controls the type of injected Corrected error generated by the fault injection feature of the node.

0b00    An injected Corrected error will not be generated by the fault injection feature of the node.
0b01    An injected non-specific Corrected error is generated in the fault injection state.
ERR<n>STATUS.CE is set to 0b10 when the injected error is recorded.
0b10    An injected transient Corrected error is generated in the fault injection state.
ERR<n>STATUS.CE is set to 0b01 when the injected error is recorded.
0b11    An injected persistent Corrected error is generated in the fault injection state.
ERR<n>STATUS.CE is set to 0b11 when the injected error is recorded.

The set of permitted values for this field is defined by ERR<n>PFGF.CE.

The node enters the fault injection state when the Error Generation Counter decrements to zero. It is IMPLEMENTATION DEFINED whether the injected error is generated when the error is generated on an access to the component in the fault injection state and the data is not consumed.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.

DE, bit [5]

When the node supports this control:
Deferred Error generation enable. Controls whether an injected Deferred error is generated by the fault injection feature of the node.

0b0    An injected Deferred error will not be generated by the fault generation feature of the node.
0b1    An injected Deferred error is generated in the fault injection state.

The node enters the fault injection state when the Error Generation Counter decrements to zero. It is IMPLEMENTATION DEFINED whether the injected error is generated when the error is generated on an access to the component in the fault injection state and the data is not consumed.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:
Reserved, RES0.
UEO, bit [4]

*When the node supports this control:*

Latent or Restartable Error generation enable. Controls whether an injected Latent or Restartable error is generated by the fault injection feature of the node.

0b0  An injected Latent or Restartable error will not be generated by the fault generation feature of the node.

0b1  An injected Latent or Restartable error is generated in the fault injection state.

The node enters the fault injection state when the Error Generation Counter decrements to zero. It is IMPLEMENTATION DEFINED whether the injected error is generated when the error is generated on an access to the component in the fault injection state and the data is not consumed.

The reset behavior of this field is:

* On a Cold reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

UER, bit [3]

*When the node supports this control:*

Signaled or Recoverable Error generation enable. Controls whether an injected Signaled or Recoverable error is generated by the fault injection feature of the node.

0b0  An injected Signaled or Recoverable error will not be generated by the fault generation feature of the node.

0b1  An injected Signaled or Recoverable error is generated in the fault injection state.

The node enters the fault injection state when the Error Generation Counter decrements to zero. It is IMPLEMENTATION DEFINED whether the injected error is generated when the error is generated on an access to the component in the fault injection state and the data is not consumed.

The reset behavior of this field is:

* On a Cold reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.

UEU, bit [2]

*When the node supports this control:*

Unrecoverable Error generation enable. Controls whether an injected Unrecoverable error is generated by the fault injection feature of the node.

0b0  An injected Unrecoverable error will not be generated by the fault generation feature of the node.

0b1  An injected Unrecoverable error is generated in the fault injection state.

The node enters the fault injection state when the Error Generation Counter decrements to zero. It is IMPLEMENTATION DEFINED whether the injected error is generated when the error is generated on an access to the component in the fault injection state and the data is not consumed.

The reset behavior of this field is:

* On a Cold reset, this field resets to an architecturally UNKNOWN value.

*Otherwise:*

Reserved, RES0.
UC, bit [1]

When the node supports this control:

Uncontainable Error generation enable. Controls whether an injected Uncontainable error is generated by the fault injection feature of the node.

0b0 An injected Uncontainable error will not be generated by the fault generation feature of the node.

0b1 An injected Uncontainable error is generated in the fault injection state.

The node enters the fault injection state when the Error Generation Counter decrements to zero. It is IMPLEMENTATION DEFINED whether the injected error is generated when the error is generated on an access to the component in the fault injection state and the data is not consumed.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

OF, bit [0]

When the node supports this control:

Overflow flag. The value written to ERR<n>STATUS.OF when an injected error is recorded.

0b0 ERR<n>STATUS.OF is set to 0 when an injected error is recorded.

0b1 ERR<n>STATUS.OF is set to 1 when an injected error is recorded.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing the ERR<n>PFGCTL:

ERR<n>PFGCTL can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0x808 + (64 * n)</td>
<td>ERR&lt;n&gt;PFGCTL</td>
</tr>
</tbody>
</table>

Accesses to this interface are RW.
I5.8.31 ERR<n>PFGF, Error Record <n> Pseudo-fault Generation Feature Register, n = 0 - 65534

The ERR<n>PFGF characteristics are:

**Purpose**
Defines which common architecturally-defined fault generation features are implemented.

**Configurations**
This register is present only when error record <n> is implemented, the node implements the Common Fault Injection Model Extension (ERR<n>FR.INJ != 0b00) and error record <n> is the first error record owned by a node. Otherwise, direct accesses to ERR<n>PFGF are RES0.

ERR<n>FR describes the features implemented by the node.

**Attributes**
ERR<n>PFGF is a 64-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[63:31]</td>
<td>Reserved, RES0.</td>
<td></td>
</tr>
<tr>
<td>0b0</td>
<td>The node does not support this feature. ERR&lt;n&gt;PFGCTL.R is RES0.</td>
<td></td>
</tr>
<tr>
<td>0b1</td>
<td>Error Generation Counter restart mode is implemented and is controlled by ERR&lt;n&gt;PFGCTL.R. ERR&lt;n&gt;PFGCTL.R is a read/write field.</td>
<td></td>
</tr>
<tr>
<td>0b0</td>
<td>When an injected error is recorded, the node sets ERR&lt;n&gt;STATUS.{IERR, SERR} to IMPLEMENTATION DEFINED values. ERR&lt;n&gt;STATUS.{IERR, SERR} are UNKNOWN when ERR&lt;n&gt;STATUS.V is 0.</td>
<td></td>
</tr>
<tr>
<td>0b1</td>
<td>When an injected error is recorded, the node does not update the ERR&lt;n&gt;STATUS.{IERR, SERR} fields. ERR&lt;n&gt;STATUS.{IERR, SERR} are writable when ERR&lt;n&gt;STATUS.V is 0.</td>
<td></td>
</tr>
<tr>
<td>[28]</td>
<td>NA, bit</td>
<td>No access required. Defines whether this component fakes detection of the error on an access to the component or spontaneously in the fault injection state.</td>
</tr>
<tr>
<td>0b0</td>
<td>The component fakes detection of the error on an access to the component.</td>
<td></td>
</tr>
</tbody>
</table>
The component fakes detection of the error spontaneously in the fault injection state.

**Bits [27:13]**

Reserved, RES0.

**MV, bit [12]**

Miscellaneous syndrome.

Defines whether software can control all or part of the syndrome recorded in the ERR<\text{n}>MISC<\text{m}> registers when an injected error is recorded.

It is IMPLEMENTATION DEFINED which ERR<\text{n}>MISC<\text{m}> syndrome fields, if any, are updated by the node when an injected error is recorded. Some syndrome fields might always be updated by the node when an error, including an injected error, is recorded. For example, a corrected error counter might always be updated when any countable error, including a injected countable error, is recorded.

0b0 When an injected error is recorded, the node might update the ERR<\text{n}>MISC<\text{m}> registers:

- If any syndrome is recorded by the node in the ERR<\text{n}>MISC<\text{m}> registers, then ERR<\text{n}>STATUS.MV is set to 1.
- Otherwise, ERR<\text{n}>STATUS.MV is unchanged.

If the node always sets ERR<\text{n}>STATUS.MV to 1 when recording an injected error then ERR<\text{n}>PFGCTL.MV might be RAO/WI. Otherwise ERR<\text{n}>PFGCTL.MV is RES0.

0b1 When an injected error is recorded, the node might update some, but not all ERR<\text{n}>MISC<\text{m}> syndrome fields:

- If any syndrome is recorded by the node in the ERR<\text{n}>MISC<\text{m}> registers, then ERR<\text{n}>STATUS.MV is set to 1.
- Otherwise, ERR<\text{n}>STATUS.MV is set to ERR<\text{n}>PFGCTL.MV.

ERR<\text{n}>MISC<\text{m}> syndrome fields that are not updated by the node are writable when ERR<\text{n}>STATUS.MV is 0.

If the node always sets ERR<\text{n}>STATUS.MV to 1 when recording an injected error then ERR<\text{n}>PFGCTL.MV is RAO/WI. Otherwise ERR<\text{n}>PFGCTL.MV is a read/write field.

If ERR<\text{n}>PFGF.MV is 1, software can write specific additional syndrome values into the ERR<\text{n}>MISC<\text{m}> registers when setting up a fault injection event. The permitted values that can be written to these registers are IMPLEMENTATION DEFINED.

**AV, bit [11]**

Address syndrome. Defines whether software can control the address recorded in ERR<\text{n}>ADDR when an injected error is recorded.

0b0 When an injected error is recorded, the node might record an address in ERR<\text{n}>ADDR. If an address is recorded in ERR<\text{n}>ADDR, then ERR<\text{n}>STATUS.AV is set to 1. Otherwise, ERR<\text{n}>ADDR and ERR<\text{n}>STATUS.AV are unchanged.

If the node always records an address and sets ERR<\text{n}>STATUS.AV to 1 when recording an injected error then ERR<\text{n}>PFGCTL.AV might be RAO/WI. Otherwise ERR<\text{n}>PFGCTL.AV is RES0.

0b1 When an injected error is recorded, the node does not update ERR<\text{n}>ADDR and does one of:

- Sets ERR<\text{n}>STATUS.AV to ERR<\text{n}>PFGCTL.AV. ERR<\text{n}>PFGCTL.AV is a read/write field.
- Sets ERR<\text{n}>STATUS.AV to 1. ERR<\text{n}>PFGCTL.AV is RAO/WI.

ERR<\text{n}>ADDR is writable when ERR<\text{n}>STATUS.AV is 0.

If ERR<\text{n}>PFGF.AV is 1, software can write a specific address value into ERR<\text{n}>ADDR when setting up a fault injection event.
PN, bit [10]

*When the node supports this flag:*

Poison flag. Describes how the fault generation feature of the node sets the \(\text{ERR}<\text{n}>\text{STATUS}.\text{PN}\) status flag.

- \(0b0\) When an injected error is recorded, it is IMPLEMENTATION DEFINED whether the node sets \(\text{ERR}<\text{n}>\text{STATUS}.\text{PN}\) to 1. \(\text{ERR}<\text{n}>\text{PFGCTL}.\text{PN}\) is \text{RES0}.

- \(0b1\) When an injected error is recorded, \(\text{ERR}<\text{n}>\text{STATUS}.\text{PN}\) is set to \(\text{ERR}<\text{n}>\text{PFGCTL}.\text{PN}\). \(\text{ERR}<\text{n}>\text{PFGCTL}.\text{PN}\) is a read/write field.

This behavior replaces the architecture-defined rules for setting the \(\text{ERR}<\text{n}>\text{STATUS}.\text{PN}\) bit.

*Otherwise:*

Reserved, RAZ.

ER, bit [9]

*When the node supports this flag:*

Error Reported flag. Describes how the fault generation feature of the node sets the \(\text{ERR}<\text{n}>\text{STATUS}.\text{ER}\) status flag.

- \(0b0\) When an injected error is recorded, the node sets \(\text{ERR}<\text{n}>\text{STATUS}.\text{ER}\) according to the architecture-defined rules for setting the ER field. \(\text{ERR}<\text{n}>\text{PFGCTL}.\text{ER}\) is \text{RES0}.

- \(0b1\) When an injected error is recorded, \(\text{ERR}<\text{n}>\text{STATUS}.\text{ER}\) is set to \(\text{ERR}<\text{n}>\text{PFGCTL}.\text{ER}\). This behavior replaces the architecture-defined rules for setting the ER bit. \(\text{ERR}<\text{n}>\text{PFGCTL}.\text{ER}\) is a read/write field.

*Otherwise:*

Reserved, RAZ.

CI, bit [8]

*When the node supports this flag:*

Critical Error flag. Describes how the fault generation feature of the node sets the \(\text{ERR}<\text{n}>\text{STATUS}.\text{CI}\) status flag.

- \(0b0\) When an injected error is recorded, it is IMPLEMENTATION DEFINED whether the node sets \(\text{ERR}<\text{n}>\text{STATUS}.\text{CI}\) to 1. \(\text{ERR}<\text{n}>\text{PFGCTL}.\text{CI}\) is \text{RES0}.

- \(0b1\) When an injected error is recorded, \(\text{ERR}<\text{n}>\text{STATUS}.\text{CI}\) is set to \(\text{ERR}<\text{n}>\text{PFGCTL}.\text{CI}\). \(\text{ERR}<\text{n}>\text{PFGCTL}.\text{CI}\) is a read/write field.

This behavior replaces the architecture-defined rules for setting the \(\text{ERR}<\text{n}>\text{STATUS}.\text{CI}\) bit.

*Otherwise:*

Reserved, RAZ.

CE, bits [7:6]

*When the node supports this type of error:*

Corrected Error generation. Describes the types of Corrected error that the fault generation feature of the node can generate.

- \(0b00\) The fault generation feature of the node does not generate Corrected errors. \(\text{ERR}<\text{n}>\text{PFGCTL}.\text{CE}\) is \text{RES0}.

- \(0b01\) The fault generation feature of the node allows generation of a non-specific Corrected error, that is, a Corrected error that is recorded by setting \(\text{ERR}<\text{n}>\text{STATUS}.\text{CE}\) to \(0b10\). \(\text{ERR}<\text{n}>\text{PFGCTL}.\text{CE}\) is a read/write field. The values \(0b10\) and \(0b11\) in \(\text{ERR}<\text{n}>\text{PFGCTL}.\text{CE}\) are reserved.

- \(0b11\) The fault generation feature of the node allows generation of transient or persistent Corrected errors, that is, Corrected errors that are recorded by setting \(\text{ERR}<\text{n}>\text{STATUS}.\text{CE}\) to \(0b01\) or \(0b11\) respectively. \(\text{ERR}<\text{n}>\text{PFGCTL}.\text{CE}\) is a read/write field. The value \(0b01\) in \(\text{ERR}<\text{n}>\text{PFGCTL}.\text{CE}\) is reserved.

All other values are reserved.
If $ERR_{i}^{n}$FR.FR.X is 1, then $ERR_{i}^{n}$FR.CE indicates whether the node supports this type of error.

**Otherwise:**

Reserved, RAZ.

**DE, bit [5]**

*When the node supports this type of error:*

Deferred Error generation. Describes whether the fault generation feature of the node can generate Deferred errors.

- $0b0$: The fault generation feature of the node does not generate Deferred errors. $ERR_{i}^{n}$PFGCTRL.DE is RES0.
- $0b1$: The fault generation feature of the node allows generation of Deferred errors. $ERR_{i}^{n}$PFGCTRL.DE is a read/write field.

*Otherwise:*

Reserved, RAZ.

**UEO, bit [4]**

*When the node supports this type of error:*

Latent or Restartable Error generation. Describes whether the fault generation feature of the node can generate Latent or Restartable errors.

- $0b0$: The fault generation feature of the node does not generate Latent or Restartable errors. $ERR_{i}^{n}$PFGCTRL.UEO is RES0.
- $0b1$: The fault generation feature of the node allows generation of Latent or Restartable errors. $ERR_{i}^{n}$PFGCTRL.UEO is a read/write field.

*Otherwise:*

Reserved, RAZ.

**UER, bit [3]**

*When the node supports this type of error:*

Signaled or Recoverable Error generation. Describes whether the fault generation feature of the node can generate Signaled or Recoverable errors.

- $0b0$: The fault generation feature of the node does not generate Signaled or Recoverable errors. $ERR_{i}^{n}$PFGCTRL.UER is RES0.
- $0b1$: The fault generation feature of the node allows generation of Signaled or Recoverable errors. $ERR_{i}^{n}$PFGCTRL.UER is a read/write field.

*Otherwise:*

Reserved, RAZ.

**UEU, bit [2]**

*When the node supports this type of error:*

Unrecoverable Error generation. Describes whether the fault generation feature of the node can generate Unrecoverable errors.

- $0b0$: The fault generation feature of the node does not generate Unrecoverable errors. $ERR_{i}^{n}$PFGCTRL.UEU is RES0.
- $0b1$: The fault generation feature of the node allows generation of Unrecoverable errors. $ERR_{i}^{n}$PFGCTRL.UEU is a read/write field.
If ERR<n>FR.FRX is 1, then ERR<n>FR.UEU indicates whether the node supports this type of error.

**Otherwise:**
Reserved, RAZ.

**UC, bit [1]**

*When the node supports this type of error:*
Uncontainable Error generation. Describes whether the fault generation feature of the node can generate Uncontainable errors.

0b0 The fault generation feature of the node does not generate Uncontainable errors. ERR<n>PFGCTL.UC is RES0.

0b1 The fault generation feature of the node allows generation of Uncontainable errors. ERR<n>PFGCTL.UC is a read/write field.

If ERR<n>FR.FRX is 1, then ERR<n>FR.UC indicates whether the node supports this type of error.

**Otherwise:**
Reserved, RAZ.

**OF, bit [0]**

*When the node supports this flag:*
Overflow flag. Describes how the fault generation feature of the node sets the ERR<n>STATUS.OF status flag.

0b0 When an injected error is recorded, the node sets ERR<n>STATUS.OF according to the architecture-defined rules for setting the OF field. ERR<n>PFGCTL.OF is RES0.

0b1 When an injected error is recorded, ERR<n>STATUS.OF is set to ERR<n>PFGCTL.OF. This behavior replaces the architecture-defined rules for setting the OF bit. ERR<n>PFGCTL.OF is a read/write field.

**Otherwise:**
Reserved, RAZ.

**Accessing the ERR<n>PFGF:**

ERR<n>PFGF can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0x800 + (64 * n)</td>
<td>ERR&lt;n&gt;PFGF</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
**Err<n>STATUS, Error Record <n> Primary Status Register, n = 0 - 65534**

The Err<n>STATUS characteristics are:

**Purpose**

Contains status information for error record <n>, including:
- Whether any error has been detected (valid).
- Whether any detected error was not corrected, and returned to a Requester.
- Whether any detected error was not corrected and deferred.
- Whether an error record has been discarded because additional errors have been detected before the first error was handled by software (overflow).
- Whether any error has been reported.
- Whether the other error record registers contain valid information.
- Whether the error was reported because poison data was detected or because a corrupt value was detected by an error detection code.
- A primary error code.
- An IMPLEMENTATION DEFINED extended error code.

Within this register:
- Err<n>STATUS. {AV, V, MV} are valid bits that define whether error record <n> registers are valid.
- Err<n>STATUS. {UE, OF, CE, DE, UET} encode the types of error or errors recorded.
- Err<n>STATUS. {CI, ER, PN, IERR, SERR} are syndrome fields.

**Configurations**

This register is present only when error record <n> is implemented. Otherwise, direct accesses to Err<n>STATUS are RES0.

Err<n>FR describes the features implemented by the node that owns error record <n>. <q> is the index of the first error record owned by the same node as error record <n>. If the node owns a single record, then q = n.

For IMPLEMENTATION DEFINED fields in Err<n>STATUS, writing zero returns the error record to an initial quiescent state.

In particular, if any IMPLEMENTATION DEFINED syndrome fields might generate a Fault Handling or Error Recovery Interrupt request, writing zero is sufficient to deactivate the Interrupt request.

Fields that are read-only, non-zero, and ignore writes are compliant with this requirement.

--- **Note** ---

Arm recommends that any IMPLEMENTATION DEFINED syndrome field that can generate a Fault Handling, Error Recovery, Critical, or IMPLEMENTATION DEFINED, interrupt request is disabled at Cold reset and is enabled by software writing an IMPLEMENTATION DEFINED nonzero value to an IMPLEMENTATION DEFINED field in Err<n>CTRL.

--- **Attributes** ---

Err<n>STATUS is a 64-bit register.
Field descriptions

When RAS System Architecture v1.1 is implemented:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>A V, bit</td>
</tr>
<tr>
<td>30</td>
<td>V, bit</td>
</tr>
<tr>
<td>29</td>
<td>UE, bit</td>
</tr>
</tbody>
</table>

Bits [63:32]

Reserved, RES0.

AV, bit [31]

When error record <n> includes an address associated with an error:

Address Valid.

0b0 ERR<n>ADDR not valid.

0b1 ERR<n>ADDR contains an address associated with the highest priority error recorded by this record.

The reset behavior of this field is:

• On a Cold reset, this field resets to 0.

Access to this field is W1C.

Otherwise:

Reserved, RES0.

V, bit [30]

Status Register Valid.

0b0 ERR<n>STATUS not valid.

0b1 ERR<n>STATUS valid. At least one error has been recorded.

The reset behavior of this field is:

• On a Cold reset, this field resets to 0.

Access to this field is W1C.

UE, bit [29]

Uncorrected Error.

0b0 No errors have been detected, or all detected errors have been either corrected or deferred.

0b1 At least one detected error was not corrected and not deferred.

When clearing ERR<n>STATUS.V to 0, if this field is nonzero, then Arm recommends that software write 1 to this field to clear this field to zero.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

• When ERR<n>STATUS.V == 0, access to this field is UNKNOWN/WI.

• Otherwise, access to this field is W1C
**External System Control Register Descriptions**

**15.8 RAS register descriptions**

---

### ER, bit [28]

**Error Reported.**

- **0b0**: No in-band error response (External Abort) signaled to the Requester making the access or other transaction.
- **0b1**: An in-band error response was signaled by the component to the Requester making the access or other transaction. This can be because any of the following are true:
  - The applicable one of the `ERR<n>CTLR.{WUE, RUE, UE}` fields is implemented and was 1 when an error was detected and not corrected.
  - The applicable one of the `ERR<n>CTLR.{WUE, RUE, UE}` fields is not implemented and the component always reports errors.

It is **IMPLEMENTATION DEFINED** whether an uncorrected error that is deferred and recorded as a Deferred error, but is not deferred to the Requester, will signal an in-band error response to the Requester, causing this field to be set to 1. If no in-band error response to the Requester, this field is set to 0.

---

**Note**

An in-band error response signaled by the component might be masked and not generate any exception.

---

When clearing `ERR<n>STATUS.V` to 0, if this field is nonzero, then Arm recommends that software write 1 to this field to clear this field to zero.

The reset behavior of this field is:

- **On a Cold reset, this field resets to an architecturally UNKNOWN value.**

Accessing this field has the following behavior:

- **UNKNOWN/WI** if all of the following are true:
  - `ERR<n>STATUS.UE == 0`
  - `ERR<n>STATUS.DE == 0`
  - this field can be set to 0b1 by a Deferred error
- **UNKNOWN/WI** if all of the following are true:
  - `ERR<n>STATUS.UE == 0`
  - this field is never set to 0b1 by a Deferred error
- **When `ERR<n>STATUS.V == 0`, access to this field is UNKNOWN/WI.**
- **Otherwise, access to this field is W1C**

---

### OF, bit [27]

**Overflow.**

Indicates that multiple errors have been detected. This field is set to 1 when one of the following occurs:

- A Corrected error counter is implemented, an error is counted, and the counter overflows.
- `ERR<n>STATUS.V` was previously 1, a Corrected error counter is not implemented, and a Corrected error is recorded.
- `ERR<n>STATUS.V` was previously 1, and a type of error other than a Corrected error is recorded.

Otherwise, this field is unchanged when an error is recorded.

If a Corrected error counter is implemented:

- A direct write that modifies the counter overflow flag indirectly might set this field to an UNKNOWN value.
- A direct write to this field that clears this field to zero might indirectly set the counter overflow flag to an UNKNOWN value.

- **0b0**: Since this field was last cleared to zero, no error syndrome has been discarded and, if a Corrected error counter is implemented, it has not overflowed.
0b1 Since this field was last cleared to zero, at least one error syndrome has been discarded or, if a Corrected error counter is implemented, it might have overflowed.

When clearing ERR<n>STATUS.V to 0, if this field is nonzero, then Arm recommends that software write 1 to this field to clear this field to zero.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:
- When ERR<n>STATUS.V == 0, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is W1C

MV, bit [26]

*When error record <n> includes an additional information for an error:*

Miscellaneous Registers Valid.

0b0 ERR<n>MISC<m> not valid.
0b1 The contents of the ERR<n>MISC<m> registers contain additional information for an error recorded by this record.

**Note**

If the ERR<n>MISC<m> registers can contain additional information for a previously recorded error, then the contents must be self-describing to software or a user. For example, certain fields might relate only to Corrected errors, and other fields only to the most recent error that was not discarded.

The reset behavior of this field is:
- On a Cold reset, this field resets to 0.

Access to this field is W1C.

*Otherwise:*

Reserved, RES0.

CE, bits [25:24]

Corrected Error.

0b00 No errors were corrected.
0b01 At least one transient error was corrected.
0b10 At least one error was corrected.
0b11 At least one persistent error was corrected.

The mechanism by which a component or node detects whether a Corrected error is transient or persistent is IMPLEMENTATION DEFINED. If no such mechanism is implemented, then the node sets this field to 0b10 when a corrected error is recorded.

When clearing ERR<n>STATUS.V to 0, if this field is nonzero, then Arm recommends that software write ones to this field to clear this field to zero.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:
- When ERR<n>STATUS.V == 0, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is W1C

DE, bit [23]

Deferred Error.

0b0 No errors were deferred.
0b1 At least one error was not corrected and deferred.
Support for deferring errors is IMPLEMENTATION DEFINED.

When clearing ERR<n>STATUS.V to 0, if this field is nonzero, then Arm recommends that software write 1 to this field to clear this field to zero.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When ERR<n>STATUS.V == 0, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is W1C

PN, bit [22]

Poison.

0b0  Uncorrected error or Deferred error recorded because a corrupt value was detected, for example, by an error detection code (EDC), or Corrected error recorded.

0b1  Uncorrected error or Deferred error recorded because a poison value was detected.

When clearing ERR<n>STATUS.V to 0, if this field is nonzero, then Arm recommends that software write 1 to this field to clear this field to zero.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- UNKNOWN/WI if any of the following are true:
  - ERR<n>STATUS.V == 0
  - ERR<n>STATUS.DE == 0 and ERR<n>STATUS.UE == 0
- Otherwise, access to this field is W1C

UET, bits [21:20]

Uncorrected Error Type. Describes the state of the component after detecting or consuming an Uncorrected error.

0b00  Uncorrected error, Uncontainable error (UC).

0b01  Uncorrected error, Unrecoverable error (UEU).

0b10  Uncorrected error, Latent or Restartable error (UEO).

0b11  Uncorrected error, Signaled or Recoverable error (UER).

--- Note ---

Software might use the information in the error record registers to determine what recovery is necessary.

---

When clearing ERR<n>STATUS.V to 0, if this field is nonzero, then Arm recommends that software write ones to this field to clear this field to zero.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- UNKNOWN/WI if any of the following are true:
  - ERR<n>STATUS.V == 0
  - ERR<n>STATUS.UE == 0
- Otherwise, access to this field is W1C

CI, bit [19]

Critical Error. Indicates whether a critical error condition has been recorded.

0b0  No critical error condition.

0b1  Critical error condition.
When clearing ERR<\textless{}n\textgreater{}STATUS.V to 0, if this field is nonzero, then Arm recommends that software write 1 to this field to clear this field to zero.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When ERR<\textless{}n\textgreater{}STATUS.V == 0, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is W1C

### Bits [18:16]

Reserved, RES0.

### IERR, bits [15:8]

IMPLEMENTATION DEFINED error code. Used with any primary error code ERR<\textless{}n\textgreater{}STATUS.SERR value. Further IMPLEMENTATION DEFINED information can be placed in the ERR<\textless{}n\textgreater{}MISC<m> registers.

The implemented set of valid values that this field can take is IMPLEMENTATION DEFINED. If any value not in this set is written to this register, then the value read back from this field is UNKNOWN.

______ Note ______

This means that one or more bits of this field might be implemented as fixed read-as-zero or read-as-one values.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- UNKNOWN/WI if all of the following are true:
  - the Common Fault Injection Model Extension is not implemented by the node that owns this error record
  - ERR<\textless{}n\textgreater{}STATUS.V == 0
- UNKNOWN/WI if all of the following are true:
  - ERR<q>PFGF.SYN == 0
  - ERR<\textless{}n\textgreater{}STATUS.V == 0
- Otherwise, access to this field is RW

### SERR, bits [7:0]

Architecturally-defined primary error code. The primary error code might be used by a fault handling agent to triage an error without requiring device-specific code. For example, to count and threshold corrected errors in software, or generate a short log entry.

0x00  No error.
0x01  IMPLEMENTATION DEFINED error.
0x02  Data value from (non-associative) internal memory. For example, ECC from on-chip SRAM or buffer.
0x03  IMPLEMENTATION DEFINED pin. For example, nSEI pin.
0x04  Assertion failure. For example, consistency failure.
0x05  Error detected on internal data path. For example, parity on ALU result.
0x06  Data value from associative memory. For example, ECC error on cache data.
0x07  Address/control value from associative memory. For example, ECC error on cache tag.
0x08  Data value from a TLB. For example, ECC error on TLB data.
0x09  Address/control value from a TLB. For example, ECC error on TLB tag.
0x0A  Data value from producer. For example, parity error on write data bus.
0x0B  Address/control value from producer. For example, parity error on address bus.
0x0C  Data value from (non-associative) external memory. For example, ECC error in SDRAM.

0x0D  Illegal address (software fault). For example, access to unpopulated memory.

0x0E  Illegal access (software fault). For example, byte write to word register.

0x0F  Illegal state (software fault). For example, device not ready.

0x10  Internal data register. For example, parity on a SIMD&FP register. For a PE, all general-purpose, stack pointer, SIMD&FP, and SVE registers are data registers.

0x11  Internal control register. For example, Parity on a System register. For a PE, all registers other than general-purpose, stack pointer, SIMD&FP, and SVE registers are control registers.

0x12  Error response from Completer of access. For example, error response from cache write-back.

0x13  External timeout. For example, timeout on interaction with another component.

0x14  Internal timeout. For example, timeout on interface within the component.

0x15  Deferred error from Completer not supported at Requester. For example, poisoned data received from the Completer of an access by a Requester that cannot defer the error further.

0x16  Deferred error from Requester not supported at Completer. For example, poisoned data received from the Requester of an access by a Completer that cannot defer the error further.

0x17  Deferred error from Completer passed through. For example, poisoned data received from the Completer of an access and returned to the Requester.

0x18  Deferred error from Requester passed through. For example, poisoned data received from the Requester of an access and deferred to the Completer.

0x19  Error recorded by PCIe error logs. Indicates that the component has recorded an error in a PCIe error log. This might be the PCIe device status register, AER, DVSEC, or other mechanisms defined by PCIe.

0x1A  Other internal error. For example, parity error on internal state of the component that is not covered by another primary error code.

All other values are reserved.

The implemented set of valid values that this field can take is IMPLEMENTATION DEFINED. If any value not in this set is written to this register, then the value read back from this field is UNKNOWN.

Note
This means that one or more bits of this field might be implemented as fixed read-as-zero or read-as-one values.

The reset behavior of this field is:
• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:
• UNKNOWN/WI if all of the following are true:
  — the Common Fault Injection Model Extension is not implemented by the node that owns this error record
  — ERR<n>STATUS.V == 0
• UNKNOWN/WI if all of the following are true:
  — ERR<q>PFGF.SYN == 0
  — ERR<n>STATUS.V == 0
• Otherwise, access to this field is RW
When RAS System Architecture v1.0 is implemented:

<table>
<thead>
<tr>
<th></th>
<th>RES0</th>
<th>8</th>
<th>7</th>
<th>IERR</th>
<th>SERR</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td></td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td>26</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [63:32]**

Reserved, RES0.

**AV, bit [31]**

*When error record <n> includes an address associated with an error:*

Address Valid.

0b0   ERR<n>ADDR not valid.
0b1   ERR<n>ADDR contains an address associated with the highest priority error recorded by this record.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.

Accessing this field has the following behavior:

- RO if all of the following are true:
  - ERR<n>STATUS.DE == 0
  - ERR<n>STATUS.UE == 0
  - ERR<n>STATUS.CE != 0b00
  - ERR<n>STATUS.CE is not being cleared to 0b00 in the same write

- RO if all of the following are true:
  - ERR<n>STATUS.DE != 0
  - ERR<n>STATUS.DE is not being cleared to 0b0 in the same write

- RO if all of the following are true:
  - ERR<n>STATUS.UE != 0
  - ERR<n>STATUS.UE is not being cleared to 0b0 in the same write

- Otherwise, access to this field is W1C

**Otherwise:**

Reserved, RES0.

**V, bit [30]**

Status Register Valid.

0b0   ERR<n>STATUS not valid.
0b1   ERR<n>STATUS valid. At least one error has been recorded.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0.

Accessing this field has the following behavior:

- RO if all of the following are true:
  - ERR<n>STATUS.CE != 0b00
  - ERR<n>STATUS.CE is not being cleared to 0b00 in the same write

- RO if all of the following are true:
  - ERR<n>STATUS.DE != 0
  - ERR<n>STATUS.DE is not being cleared to 0b0 in the same write

- Otherwise, access to this field is W1C
• RO if all of the following are true:
  — ERR<n>STATUS.UE != 0
  — ERR<n>STATUS.UE is not being cleared to 0b0 in the same write
• Otherwise, access to this field is W1C

**UE, bit [29]**

Uncorrected Error.

0b0 No errors have been detected, or all detected errors have been either corrected or deferred.

0b1 At least one detected error was not corrected and not deferred.

When clearing ERR<n>STATUS.V to 0, if this field is nonzero, then Arm recommends that software write 1 to this field to clear this field to zero.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

• When ERR<n>STATUS.V == 0, access to this field is UNKNOWN/WI.
• RO if all of the following are true:
  — ERR<n>STATUS.OF == 1
  — ERR<n>STATUS.OF is not being cleared to 0b0 in the same write
• Otherwise, access to this field is W1C

**ER, bit [28]**

Error Reported.

0b0 No in-band error response (External Abort) signaled to the Requester making the access or other transaction.

0b1 An in-band error response was signaled by the component to the Requester making the access or other transaction. This can be because any of the following are true:

• The applicable one of the ERR<n>CTLR.{WUE, RUE, UE} fields is implemented and was 1 when an error was detected and not corrected.
• The applicable one of the ERR<n>CTLR.{WUE, RUE, UE} fields is not implemented and the component always reports errors.

It is IMPLEMENTATION DEFINED whether an uncorrected error that is deferred and recorded as a Deferred error, but is not deferred to the Requester, will signal an in-band error response to the Requester, causing this field to be set to 1. If no in-band error response to the Requester, this field is set to 0.

_________ Note ___________

An in-band error response signaled by the component might be masked and not generate any exception.

_________ Note ___________

If this field is nonzero, then Arm recommends that software write 1 to this field to clear this field to zero, when any of:

• Clearing ERR<n>STATUS.V to 0.
• Clearing ERR<n>STATUS.UE to 0, if this field is never set to 1 by a Deferred error.
• Clearing ERR<n>STATUS.{UE,DE} to {0,0}, if this field can be set to 1 by a Deferred error.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

• UNKNOWN/WI if all of the following are true:
  — ERR<n>STATUS.UE == 0
  — ERR<n>STATUS.DE == 0
— this field can be set to 0b1 by a Deferred error

• **UNKNOWN/WI** if all of the following are true:
  — ERR<n>STATUS.UE == 0
  — this field is never set to 0b1 by a Deferred error

• When ERR<n>STATUS.V == 0, access to this field is UNKNOWN/WI.

• **RO** if all of the following are true:
  — ERR<n>STATUS.DE == 0
  — ERR<n>STATUS.UE == 0
  — ERR<n>STATUS.CE != 0b00
  — ERR<n>STATUS.CE is not being cleared to 0b00 in the same write

• RO if all of the following are true:
  — ERR<n>STATUS.DE == 0
  — ERR<n>STATUS.UE != 0
  — ERR<n>STATUS.DE is not being cleared to 0b0 in the same write

• RO if all of the following are true:
  — ERR<n>STATUS.UE == 0
  — ERR<n>STATUS.DE != 0
  — ERR<n>STATUS.DE is not being cleared to 0b0 in the same write

• Otherwise, access to this field is W1C

**OF, bit [27]**

Overflow.

Indicates that multiple errors have been detected. This field is set to 1 when one of the following occurs:

• An Uncorrected error is detected and ERR<n>STATUS.UE == 1.

• A Deferred error is detected, ERR<n>STATUS.DE == 0 and ERR<n>STATUS.DE == 1.

• A Corrected error is detected, no Corrected error counter is implemented, ERR<n>STATUS.UE == 0, ERR<n>STATUS.DE == 0, and ERR<n>STATUS.CE != 0b00. ERR<n>STATUS.CE might be updated for the new Corrected error.

• A Corrected error counter is implemented, ERR<n>STATUS.UE == 0, ERR<n>STATUS.DE == 0, and the counter overflows.

It is IMPLEMENTATION DEFINED whether this field is set to 1 when one of the following occurs:

• A Deferred error is detected and ERR<n>STATUS.UE == 1.

• A Corrected error is detected, no Corrected error counter is implemented, and ERR<n>STATUS.UE, DE] != [0, 0].

• A Corrected error counter is implemented, ERR<n>STATUS.UE, DE] != [0, 0], and the counter overflows.

It is IMPLEMENTATION DEFINED whether this field is cleared to 0 when one of the following occurs:

• An Uncorrected error is detected and ERR<n>STATUS.UE == 0.

• A Deferred error is detected, ERR<n>STATUS.DE == 0, and ERR<n>STATUS.DE == 0.

• A Corrected error is detected, ERR<n>STATUS.DE == 0, ERR<n>STATUS.DE == 0, and ERR<n>STATUS.CE == 0b0.

The IMPLEMENTATION DEFINED clearing of this field might also depend on the value of the other error status fields.

If a Corrected error counter is implemented:

• A direct write that modifies the counter overflow flag indirectly might set this field to an UNKNOWN value.

• A direct write to this field that clears this field to 0 might indirectly set the counter overflow flag to an UNKNOWN value.

0b0  

If ERR<n>STATUS.UE == 1, then no error syndrome for an Uncorrected error has been discarded.
If ERR<\textsubscript{n}>STATUS.UE == 0 and ERR<\textsubscript{n}>STATUS.DE == 1, then no error syndrome
for a Deferred error has been discarded.

If ERR<\textsubscript{n}>STATUS.UE == 0, ERR<\textsubscript{n}>STATUS.DE == 0, and a Corrected error
counter is implemented, then the counter has not overflowed.

If ERR<\textsubscript{n}>STATUS.UE == 0, ERR<\textsubscript{n}>STATUS.DE == 0, ERR<\textsubscript{n}>STATUS.CE != 0b00,
and no Corrected error counter is implemented, then no error syndrome for a
Corrected error has been discarded.

--- Note ---

This field might have been set to 1 when an error syndrome was discarded and later
cleared to 0 when a higher priority syndrome was recorded.

---

0b1

At least one error syndrome has been discarded or, if a Corrected error counter is
implemented, it might have overflowed.

When clearing ERR<\textsubscript{n}>STATUS.V to 0, if this field is nonzero, then Arm recommends that
software write 1 to this field to clear this field to zero.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

• When ERR<\textsubscript{n}>STATUS.V == 0, access to this field is UNKNOWN/WI.

• Otherwise, access to this field is WIC

\textit{MV, bit [26]}

\textit{When error record \textlt{n} includes an additional information for an error:}

Miscellaneous Registers Valid.

0b0   ERR<\textsubscript{n}>MISC<\textsubscript{m}> not valid.

0b1   The IMPLEMENTATION DEFINED contents of the ERR<\textsubscript{n}>MISC<\textsubscript{m}> registers contains
additional information for an error recorded by this record.

--- Note ---

If the ERR<\textsubscript{n}>MISC<\textsubscript{m}> registers can contain additional information for a previously recorded
error, then the contents must be self-describing to software or a user. For example, certain fields
might relate only to Corrected errors, and other fields only to the most recent error that was not
discarded.

---

The reset behavior of this field is:

• On a Cold reset, this field resets to 0.

Accessing this field has the following behavior:

• RO if all of the following are true:
  — ERR<\textsubscript{n}>STATUS.DE == 0
  — ERR<\textsubscript{n}>STATUS.UE == 0
  — ERR<\textsubscript{n}>STATUS.CE != 0b00
  — ERR<\textsubscript{n}>STATUS.CE is not being cleared to 0b00 in the same write

• RO if all of the following are true:
  — ERR<\textsubscript{n}>STATUS.UE == 0
  — ERR<\textsubscript{n}>STATUS.DE != 0
  — ERR<\textsubscript{n}>STATUS.DE is not being cleared to 0b0 in the same write

• RO if all of the following are true:
  — ERR<\textsubscript{n}>STATUS.UE != 0
  — ERR<\textsubscript{n}>STATUS.UE is not being cleared to 0b0 in the same write

• Otherwise, access to this field is WIC
Otherwise:
Reserved, RES0.

CE, bits [25:24]
Corrected Error.

- 0b00: No errors were corrected.
- 0b01: At least one transient error was corrected.
- 0b10: At least one error was corrected.
- 0b11: At least one persistent error was corrected.

The mechanism by which a component or node detects whether a Corrected error is transient or persistent is IMPLEMENTATION DEFINED. If no such mechanism is implemented, then the node sets this field to 0b10 when a corrected error is recorded.

When clearing ERR<STATUS.V to 0, if this field is nonzero, then Arm recommends that software write ones to this field to clear this field to zero.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:
- When ERR<STATUS.V == 0, access to this field is UNKNOWN/WI.
- RO if all of the following are true:
  - ERR<STATUS.OF == 1
  - ERR<STATUS.OF is not being cleared to 0b0 in the same write
- Otherwise, access to this field is W1C

DE, bit [23]
Deferred Error.

- 0b0: No errors were deferred.
- 0b1: At least one error was not corrected and deferred.

Support for deferring errors is IMPLEMENTATION DEFINED.

When clearing ERR<STATUS.V to 0, if this field is nonzero, then Arm recommends that software write 1 to this field to clear this field to zero.

The reset behavior of this field is:
- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:
- When ERR<STATUS.V == 0, access to this field is UNKNOWN/WI.
- RO if all of the following are true:
  - ERR<STATUS.OF == 1
  - ERR<STATUS.OF is not being cleared to 0b0 in the same write
- Otherwise, access to this field is W1C

PN, bit [22]
Poison.

- 0b0: Uncorrected error or Deferred error recorded because a corrupt value was detected, for example, by an error detection code (EDC), or Corrected error recorded.
- 0b1: Uncorrected error or Deferred error recorded because a poison value was detected.

If this field is nonzero, then Arm recommends that software write 1 to this field to clear this field to zero, when any of:
- Clearing ERR<STATUS.V to 0.
- Clearing both ERR<STATUS.{DE, UE} to 0.
The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally Unknown value.

Accessing this field has the following behavior:

- Unknown/WI if any of the following are true:
  - $\text{ERR}_n\text{STATUS}.V == 0$
  - $\text{ERR}_n\text{STATUS}.DE == 0$ and \( \text{ERR}_n\text{STATUS}.UE == 0 \)
- RO if all of the following are true:
  - $\text{ERR}_n\text{STATUS}.DE == 0$
  - $\text{ERR}_n\text{STATUS}.UE == 0$
  - $\text{ERR}_n\text{STATUS}.CE != 0b00$
    - $\text{ERR}_n\text{STATUS}.CE$ is not being cleared to 0b00 in the same write
- RO if all of the following are true:
  - $\text{ERR}_n\text{STATUS}.UE == 0$
  - $\text{ERR}_n\text{STATUS}.DE != 0$
  - $\text{ERR}_n\text{STATUS}.DE$ is not being cleared to 0b0 in the same write
- RO if all of the following are true:
  - $\text{ERR}_n\text{STATUS}.UE != 0$
  - $\text{ERR}_n\text{STATUS}.UE$ is not being cleared to 0b0 in the same write
- Otherwise, access to this field is W1C

UET, bits [21:20]

Uncorrected Error Type. Describes the state of the component after detecting or consuming an Uncorrected error.

- 0b00 Uncorrected error, Uncontainable error (UC).
- 0b01 Uncorrected error, Unrecoverable error (UEU).
- 0b10 Uncorrected error, Latent or Restartable error (UEO).
- 0b11 Uncorrected error, Signaled or Recoverable error (UER).

--- Note ---

Software might use the information in the error record registers to determine what recovery is necessary.

---

If this field is nonzero, then Arm recommends that software write ones to this field to clear this field to zero, when any of:

- Clearing $\text{ERR}_n\text{STATUS}.V$ to 0.
- Clearing $\text{ERR}_n\text{STATUS}.UE$ to 0.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally Unknown value.

Accessing this field has the following behavior:

- Unknown/WI if any of the following are true:
  - $\text{ERR}_n\text{STATUS}.V == 0$
  - $\text{ERR}_n\text{STATUS}.UE == 0$
- RO if all of the following are true:
  - $\text{ERR}_n\text{STATUS}.DE == 0$
  - $\text{ERR}_n\text{STATUS}.UE == 0$
  - $\text{ERR}_n\text{STATUS}.CE != 0b00$
    - $\text{ERR}_n\text{STATUS}.CE$ is not being cleared to 0b00 in the same write
- RO if all of the following are true:
  - $\text{ERR}_n\text{STATUS}.UE == 0$
  - $\text{ERR}_n\text{STATUS}.DE != 0$
  - $\text{ERR}_n\text{STATUS}.DE$ is not being cleared to 0b0 in the same write
- RO if all of the following are true:
  - $\text{ERR}_n\text{STATUS}.UE != 0$
  - $\text{ERR}_n\text{STATUS}.UE$ is not being cleared to 0b0 in the same write
- Otherwise, access to this field is W1C
ERR<n>STATUS.DE is not being cleared to 0b0 in the same write

• RO if all of the following are true:
  — ERR<n>STATUS.DE != 0
  — ERR<n>STATUS.UE != 0
  — ERR<n>STATUS.UE is not being cleared to 0b0 in the same write

• Otherwise, access to this field is W1C

### Bits [19:16]
Reserved, RES0.

### IERR, bits [15:8]
IMPLEMENTATION DEFINED error code. Used with any primary error code ERR<n>STATUS.SERR value. Further IMPLEMENTATION DEFINED information can be placed in the ERR<n>MISC<m> registers.

The implemented set of valid values that this field can take is IMPLEMENTATION DEFINED. If any value not in this set is written to this register, then the value read back from this field is UNKNOWN.

--- **Note** ---
This means that one or more bits of this field might be implemented as fixed read-as-zero or read-as-one values.

The reset behavior of this field is:

• On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

• UNKNOWN/WI if all of the following are true:
  — the Common Fault Injection Model Extension is not implemented by the node that owns this error record
  — ERR<n>STATUS.V == 0

• UNKNOWN/WI if all of the following are true:
  — ERR<n>PFGF.SYN == 0
  — ERR<n>STATUS.V == 0

• RO if all of the following are true:
  — ERR<n>STATUS.DE == 0
  — ERR<n>STATUS.UE == 0
  — ERR<n>STATUS.CE != 0b00
  — ERR<n>STATUS.CE is not being cleared to 0b00 in the same write

• RO if all of the following are true:
  — ERR<n>STATUS.UE == 0
  — ERR<n>STATUS.DE != 0
  — ERR<n>STATUS.DE is not being cleared to 0b0 in the same write

• RO if all of the following are true:
  — ERR<n>STATUS.UE != 0
  — ERR<n>STATUS.UE is not being cleared to 0b0 in the same write

• Otherwise, access to this field is RW

### SERR, bits [7:0]
Architecturally-defined primary error code. The primary error code might be used by a fault handling agent to triage an error without requiring device-specific code. For example, to count and threshold corrected errors in software, or generate a short log entry.

0x00 No error.
0x01 IMPLEMENTATION DEFINED error.
0x02 Data value from (non-associative) internal memory. For example, ECC from on-chip SRAM or buffer.
0x03  IMPLEMENTATION DEFINED pin. For example, nSEI pin.
0x04  Assertion failure. For example, consistency failure.
0x05  Error detected on internal data path. For example, parity on ALU result.
0x06  Data value from associative memory. For example, ECC error on cache data.
0x07  Address/control value from associative memory. For example, ECC error on cache tag.
0x08  Data value from a TLB. For example, ECC error on TLB data.
0x09  Address/control value from a TLB. For example, ECC error on TLB tag.
0x0A  Data value from producer. For example, parity error on write data bus.
0x0B  Address/control value from producer. For example, parity error on address bus.
0x0C  Data value from (non-associative) external memory. For example, ECC error in SDRAM.
0x0D  Illegal address (software fault). For example, access to unpopulated memory.
0x0E  Illegal access (software fault). For example, byte write to word register.
0x0F  Illegal state (software fault). For example, device not ready.
0x10  Internal data register. For example, parity on a SIMD&FP register. For a PE, all general-purpose, stack pointer, SIMD&FP, and SVE registers are data registers.
0x11  Internal control register. For example, Parity on a System register. For a PE, all registers other than general-purpose, stack pointer, SIMD&FP, and SVE registers are control registers.
0x12  Error response from Completer of access. For example, error response from cache write-back.
0x13  External timeout. For example, timeout on interaction with another component.
0x14  Internal timeout. For example, timeout on interface within the component.
0x15  Deferred error from Completer not supported at Requester. For example, poisoned data received from the Completer of an access by a Requester that cannot defer the error further.
0x16  Deferred error from Requester not supported at Completer. For example, poisoned data received from the Requester of an access by a Completer that cannot defer the error further.
0x17  Deferred error from Completer passed through. For example, poisoned data received from the Completer of an access and returned to the Requester.
0x18  Deferred error from Requester passed through. For example, poisoned data received from the Requester of an access and deferred to the Completer.
0x19  Error recorded by PCIe error logs. Indicates that the component has recorded an error in a PCIe error log. This might be the PCIe device status register, AER, DVSEC, or other mechanisms defined by PCIe.
0x1A  Other internal error. For example, parity error on internal state of the component that is not covered by another primary error code.

All other values are reserved.

The implemented set of valid values that this field can take is IMPLEMENTATION DEFINED. If any value not in this set is written to this register, then the value read back from this field is UNKNOWN.

--- Note ---

This means that one or more bits of this field might be implemented as fixed read-as-zero or read-as-one values.

---

The reset behavior of this field is:

* On a Cold reset, this field resets to an architecturally UNKNOWN value.
Accessing this field has the following behavior:

- **UNKNOWN/WI if all of the following are true:**
  - the Common Fault Injection Model Extension is not implemented by the node that owns this error record
  - ERR<\text{n}>STATUS.V == 0

- **UNKNOWN/WI if all of the following are true:**
  - ERR<\text{q}>PFGF.SYN == 0
  - ERR<\text{n}>STATUS.V == 0

- **RO if all of the following are true:**
  - ERR<\text{n}>STATUS.DE == 0
  - ERR<\text{n}>STATUS.UE == 0
  - ERR<\text{n}>STATUS.CE != 0b00
  - ERR<\text{n}>STATUS.CE is not being cleared to 0b00 in the same write

- **RO if all of the following are true:**
  - ERR<\text{n}>STATUS.UE == 0
  - ERR<\text{n}>STATUS.DE != 0
  - ERR<\text{n}>STATUS.DE is not being cleared to 0b0 in the same write

- **RO if all of the following are true:**
  - ERR<\text{n}>STATUS.UE != 0
  - ERR<\text{n}>STATUS.UE is not being cleared to 0b0 in the same write

- **Otherwise, access to this field is RW**

**Accessing the ERR<\text{n}>STATUS:**

ERR<\text{n}>STATUS.\{AV, V, UE, ER, OF, MV, CE, DE, PN, UET, CI\} are write-one-to-clear (W1C) fields, meaning writes of zero are ignored, and a write of one or all-ones to the field clears the field to zero.

ERR<\text{n}>STATUS.\{IERR, SERR\} are read/write (RW) fields, although the set of implemented valid values is IMPLEMENTATION DEFINED. See also ERR<\text{n}>PFGF.SYN.

After reading ERR<\text{n}>STATUS, software must clear the valid fields in the register to allow new errors to be recorded. However, between reading the register and clearing the valid fields, a new error might have overwritten the register. To prevent this error being lost by software, the register prevents updates to fields that might have been updated by a new error.

When RAS System Architecture v1.0 is implemented:

- Writes to ERR<\text{n}>STATUS.\{UE, DE, CE\} are ignored if ERR<\text{n}>STATUS.OF is 1 and is not being cleared to 0.

- Writes to ERR<\text{n}>STATUS.V are ignored if any of ERR<\text{n}>STATUS.\{UE, DE, CE\} are nonzero and are not being cleared to zero.

- Writes to ERR<\text{n}>STATUS.\{AV, MV\} and the ERR<\text{n}>STATUS.\{ER, PN, UET, IERR, SERR\} syndrome fields are ignored if the highest priority nonzero error status field is not being cleared to zero. The error status fields in priority order from highest to lowest, are ERR<\text{n}>STATUS.UE, ERR<\text{n}>STATUS.DE, and ERR<\text{n}>STATUS.CE.

When RAS System Architecture v1.1 is implemented, a write to the register is ignored if all of:

- Any of ERR<\text{n}>STATUS.\{V, UE, OF, CE, DE\} are nonzero before the write.

- The write does not clear the nonzero ERR<\text{n}>STATUS.\{V, UE, OF, CE, DE\} fields to zero by writing ones to the applicable field or fields.

Some of the fields in ERR<\text{n}>STATUS are also defined as UNKNOWN where certain combinations of ERR<\text{n}>STATUS.\{V, DE, UE\} are zero. The rules for writes to ERR<\text{n}>STATUS allow a node to implement such a field as a fixed read-only value.
For example, when RAS System Architecture v1.1 is implemented, a write to ERR<n>STATUS when ERR<n>STATUS.V is 1 results in either ERR<n>STATUS.V field being cleared to zero, or ERR<n>STATUS.V not changing. Since all fields in ERR<n>STATUS, other than ERR<n>STATUS.{AV, V, MV}, usually read as UNKNOWN values when ERR<n>STATUS.V is zero, this means those fields can be implemented as read-only if applicable.

To ensure correct and portable operation, when software is clearing the valid fields in the register to allow new errors to be recorded, Arm recommends that software:

- Read ERR<n>STATUS and determine which fields need to be cleared to zero.
- Write ones to all the W1C fields that are nonzero in the read value.
- Write zero to all the W1C fields that are zero in the read value.
- Write zero to all the RW fields.

Otherwise, these fields might not have the correct value when a new fault is recorded.

An exception is when the node supports writing to these fields as part of fault injection. See also ERR<n>-PFGF.SYN.

ERR<n>STATUS can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0x010 + (64 * n)</td>
<td>ERR&lt;n&gt;STATUS</td>
</tr>
</tbody>
</table>

This interface is accessible as follows:

- When ERR<n>STATUS.V != 0, ERR<n>STATUS.V is not being cleared to 0b0 in the same write and RAS System Architecture v1.1 is implemented accesses to this register are RO.
- When ERR<n>STATUS.UE != 0, ERR<n>STATUS.UE is not being cleared to 0b0 in the same write and RAS System Architecture v1.1 is implemented accesses to this register are RO.
- When ERR<n>STATUS.OF != 0, ERR<n>STATUS.OF is not being cleared to 0b0 in the same write and RAS System Architecture v1.1 is implemented accesses to this register are RO.
- When ERR<n>STATUS.CE != 0b00, ERR<n>STATUS.CE is not being cleared to 0b00 in the same write and RAS System Architecture v1.1 is implemented accesses to this register are RO.
- When ERR<n>STATUS.DE != 0, ERR<n>STATUS.DE is not being cleared to 0b0 in the same write and RAS System Architecture v1.1 is implemented accesses to this register are RO.
- Otherwise accesses to this register are RW.
I5.8.33 ERRPIDR0, Peripheral Identification Register 0

The ERRPIDR0 characteristics are:

**Purpose**

Provides discovery information about the component.

For more information, see *About the Peripheral identification scheme on page K6-11696*.

**Configurations**

Implementation of this register is OPTIONAL.

ERRPIDR0 is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRPIDR0 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit Positions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:8</td>
<td>Reserved, RES0.</td>
</tr>
<tr>
<td>7:0</td>
<td>PART_0, Part number, bits [7:0].</td>
</tr>
</tbody>
</table>

The part number is selected by the designer of the component. The designer chooses whether to use a 12-bit or a 16-bit part number:

- **If a 12-bit part number is used**, it is stored in ERRPIDR1.PART_1 and ERRPIDR0.PART_0. There are 8 bits, ERRPIDR2.REVISION and ERRPIDR3.REV AND, available to define the revision of the component.

- **If a 16-bit part number is used**, it is stored in ERRPIDR2.PART_2, ERRPIDR1.PART_1 and ERRPIDR0.PART_0. There are 4 bits, ERRPIDR3.REVISION, available to define the revision of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**Accessing the ERRPIDR0:**

ERRPIDR0 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xFE0</td>
<td>ERRPIDR0</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.8.34 ERRPIDR1, Peripheral Identification Register 1

The ERRPIDR1 characteristics are:

**Purpose**

Provides discovery information about the component.

For more information, see *About the Peripheral identification scheme* on page K6-11696.

**Configurations**

Implementation of this register is OPTIONAL.

ERRPIDR1 is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRPIDR1 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>17</th>
<th>10</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>DES0</td>
<td>PART1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**DES_0, bits [7:4]**

Designer, JEPI06 identification code, bits [3:0]. ERRPIDR1.DES_0 and ERRPIDR2.DES_1 together form the JEDEC-assigned JEPI06 identification code for the designer of the component. The parity bit in the JEPI06 identification code is not included. The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

--- **Note** ---

For a component designed by Arm Limited, the JEPI06 identification code is 0x3B.

---

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**PART_1, bits [3:0]**

Part number, bits [11:8].

The part number is selected by the designer of the component. The designer chooses whether to use a 12-bit or a 16-bit part number:

- If a 12-bit part number is used, it is stored in ERRPIDR1.PART_1 and ERRPIDR0.PART_0. There are 8 bits, ERRPIDR2.REVISION and ERRPIDR3.REVAND, available to define the revision of the component.
- If a 16-bit part number is used, it is stored in ERRPIDR2.PART_2, ERRPIDR1.PART_1 and ERRPIDR0.PART_0. There are 4 bits, ERRPIDR3.REVISION, available to define the revision of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.
Accessing the ERRPID1:

ERRPID1 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xFE4</td>
<td>ERRPID1</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.8.35 ERRPIDR2, Peripheral Identification Register 2

The ERRPIDR2 characteristics are:

Purpose

Provides discovery information about the component.

For more information, see About the Peripheral identification scheme on page K6-11696.

Configurations

Implementation of this register is OPTIONAL.

ERRPIDR2 is implemented only as part of a memory-mapped group of error records.

Attributes

ERRPIDR2 is a 32-bit register.

Field descriptions

When the component uses a 12-bit part number:

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>REVISION</td>
<td>1</td>
<td>DES_1</td>
<td></td>
<td>JEDEC</td>
</tr>
</tbody>
</table>

Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Component major revision. ERRPIDR2.REVISION and ERRPIDR3.REVAND together form the revision number of the component, with ERRPIDR2.REVISION being the most significant part and ERRPIDR3.REVAND the least significant part. When a component is changed, ERRPIDR2.REVISION or ERRPIDR3.REVAND are increased to ensure that software can differentiate the different revisions of the component. ERRPIDR3.REVAND should be set to 0b0000 when ERRPIDR2.REVISION is increased.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used.

Reads as 0b1.

Access to this field is RO.

DES_1, bits [2:0]

Designer, JEP106 identification code, bits [6:4]. ERRPIDR1.DES_0 and ERRPIDR2.DES_1 together form the JEDEC-assigned JEP106 identification code for the designer of the component. The parity bit in the JEP106 identification code is not included. The code identifies the designer of the component, which might not be not the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

Note

For a component designed by Arm Limited, the JEP106 identification code is 0x38.

This field has an IMPLEMENTATION DEFINED value.
When the component uses a 16-bit part number:

Bits [31:8]
Reserved, RES0.

PART_2, bits [7:4]
Part number, bits [15:12].
The part number is selected by the designer of the component. The designer chooses whether to use a 12-bit or a 16-bit part number:
• If a 12-bit part number is used, it is stored in ERRPIDR1.PART_1 and ERRPIDR0.PART_0. There are 8 bits, ERRPIDR2.REVISION and ERRPIDR3.REV AND, available to define the revision of the component.
• If a 16-bit part number is used, it is stored in ERRPIDR2.PART_2, ERRPIDR1.PART_1 and ERRPIDR0.PART_0. There are 4 bits, ERRPIDR3.REVISION, available to define the revision of the component.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

JEDEC, bit [3]
JEDEC-assigned JEP106 implementer code is used.
Reads as 0b1.
Access to this field is RO.

DES_1, bits [2:0]
Designer, JEP106 identification code, bits [6:4]. ERRPIDR1.DES_0 and ERRPIDR2.DES_1 together form the JEDEC-assigned JEP106 identification code for the designer of the component. The parity bit in the JEP106 identification code is not included. The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

Note
For a component designed by Arm Limited, the JEP106 identification code is 0x3B.

This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Accessing the ERRPIDR2:

ERRPIDR2 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xFE8</td>
<td>ERRPIDR2</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.8.36 ERRPIDR3, Peripheral Identification Register 3

The ERRPIDR3 characteristics are:

**Purpose**

Provides discovery information about the component.

For more information, see *About the Peripheral identification scheme on page K6-11696*.

**Configurations**

Implementation of this register is OPTIONAL.

ERRPIDR3 is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRPIDR3 is a 32-bit register.

**Field descriptions**

*When the component uses a 12-bit part number:*

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>RES0</td>
<td>REVAND</td>
<td>CMOD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**REVAND, bits [7:4]**

Component minor revision. ERRPIDR2.REVISION and ERRPIDR3.REVAND together form the revision number of the component, with ERRPIDR2.REVISION being the most significant part and ERRPIDR3.REVAND the least significant part. When a component is changed, ERRPIDR2.REVISION or ERRPIDR3.REVAND are increased to ensure that software can differentiate the different revisions of the component. ERRPIDR3.REVAND should be set to 0b0000 when ERRPIDR2.REVISION is increased.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**CMOD, bits [3:0]**

Customer Modified.

Indicates the component has been modified.

A value of 0b0000 means the component is not modified from the original design.

Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.

For any two components with the same Unique Component Identifier:

- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same non-zero value, it does not necessarily mean that they have the same modifications.
- If the value of the CMOD field of either of the two components is non-zero, they might not be identical, even though they have the same Unique Component Identifier.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.
When the component uses a 16-bit part number:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES0</td>
<td>REVISION</td>
<td>CMOD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits [31:8]
Reserved, RES0.

REVISION, bits [7:4]
Component revision. When a component is changed, ERRPIDR3.REVISION is increased to ensure that software can differentiate the different revisions of the component.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

CMOD, bits [3:0]
Customer Modified.
Indicates the component has been modified.
A value of 0b0000 means the component is not modified from the original design.
Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.
For any two components with the same Unique Component Identifier:
• If the value of the CMOD fields of both components equals zero, the components are identical.
• If the CMOD fields of both components have the same non-zero value, it does not necessarily mean that they have the same modifications.
• If the value of the CMOD field of either of the two components is non-zero, they might not be identical, even though they have the same Unique Component Identifier.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Accessing the ERRPIDR3:
ERRPIDR3 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xFEC</td>
<td>ERRPIDR3</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
I5.8.37  ERRPIDR4, Peripheral Identification Register 4

The ERRPIDR4 characteristics are:

**Purpose**

Provides discovery information about the component.

For more information, see *About the Peripheral identification scheme on page K6-11696.*

**Configurations**

Implementation of this register is OPTIONAL.

ERRPIDR4 is implemented only as part of a memory-mapped group of error records.

**Attributes**

ERRPIDR4 is a 32-bit register.

**Field descriptions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>27-8</td>
<td>SIZE</td>
</tr>
<tr>
<td>3-0</td>
<td>DES_2</td>
</tr>
</tbody>
</table>

**Bits [31:8]**

Reserved, RES0.

**SIZE, bits [7:4]**

Size of the component.

The distance from the start of the address space used by this component to the end of the component identification registers.

A value of 0b0000 means one of the following is true:

- The component uses a single 4KB block.
- The component uses an IMPLEMENTATION DEFINED number of 4KB blocks.

Any other value means the component occupies $2^{\text{ERRPIDR4.SIZE}}$ 4KB blocks.

Using this field to indicate the size of the component is deprecated. This field might not correctly indicate the size of the component. Arm recommends that software determine the size of the component from the Unique Component Identifier fields, and other IMPLEMENTATION DEFINED registers in the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

**DES_2, bits [3:0]**

Designer, JEP106 continuation code. This is the JEDEC-assigned JEP106 bank identifier for the designer of the component, minus 1. The code identifies the designer of the component, which might not be not the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

--- Note ---

For a component designed by Arm Limited, the JEP106 bank is 5, meaning this field has the value 0x4.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.
Accessing the ERRPIDR4:

ERRPIDR4 can be accessed through its memory-mapped interface:

<table>
<thead>
<tr>
<th>Component</th>
<th>Offset</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>0xFD0</td>
<td>ERRPIDR4</td>
</tr>
</tbody>
</table>

Accesses to this interface are RO.
Part J

Architectural Pseudocode
Chapter J1
Armv8 Pseudocode

This chapter contains pseudocode that describes many features of the A-profile architecture. It contains the following sections:

- *Pseudocode for AArch64 operation* on page J1-11026.
- *Pseudocode for AArch32 operation* on page J1-11257.
- *Shared pseudocode* on page J1-11354.
J1.1 Pseudocode for AArch64 operation

This section holds the pseudocode for execution in AArch64 state. Functions that are listed in this section are identified as AArch64.FunctionName. Some of these functions have an equivalent AArch32 function, AArch32.FunctionName. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example aarch64/debug/breakpoint.

The top-level sections of the AArch64 pseudocode hierarchy are:

- aarch64/debug
- aarch64/exceptions on page J1-11050.
- aarch64/functions on page J1-11077.
- aarch64/instrs on page J1-11174.
- aarch64/translation on page J1-11212.

J1.1.1 aarch64/debug

This section includes the following pseudocode functions:

- aarch64/debug/brbe/BRBCycleCountingEnabled on page J1-11027.
- aarch64/debug/brbe/BRBEBranch on page J1-11027.
- aarch64/debug/brbe/BRBEBranchOnISB on page J1-11028.
- aarch64/debug/brbe/BRBEDebugStateExit on page J1-11028.
- aarch64/debug/brbe/BRBEEException on page J1-11028.
- aarch64/debug/brbe/BRBEEExceptionReturn on page J1-11030.
- aarch64/debug/brbe/BRBEFreeze on page J1-11030.
- aarch64/debug/brbe/BRBEJSB on page J1-11030.
- aarch64/debug/brbe/BRBEMispredictAllowed on page J1-11031.
- aarch64/debug/brbe/BRBETimeStamp on page J1-11031.
- aarch64/debug/brbe/BRB_IALL on page J1-11031.
- aarch64/debug/brbe/BRB_INJ on page J1-11032.
- aarch64/debug/brbe/Branch on page J1-11032.
- aarch64/debug/brbe/BranchEncCycleCount on page J1-11032.
- aarch64/debug/brbe/BranchMispredict on page J1-11032.
- aarch64/debug/brbe/BranchRawCycleCount on page J1-11032.
- aarch64/debug/brbe/BranchRecordAllowed on page J1-11032.
- aarch64/debug/brbe/Contents on page J1-11033.
- aarch64/debug/brbe/FilterBranchRecord on page J1-11033.
- aarch64/debug/brbe/FirstBranchAfterProhibited on page J1-11033.
- aarch64/debug/brbe/GetBRBENumRecords on page J1-11034.
- aarch64/debug/brbe/Getter on page J1-11034.
- aarch64/debug/brbe/ShouldBRBEFreeze on page J1-11034.
- aarch64/debug/brbe/UpdateBranchRecordBuffer on page J1-11035.
- aarch64/debug/breakpoint/AArch64.BreakpointMatch on page J1-11035.
- aarch64/debug/breakpoint/AArch64.BreakpointValueMatch on page J1-11036.
- aarch64/debug/breakpoint/AArch64.StateMatch on page J1-11037.
- aarch64/debug/enables/AArch64.GenerateDebugExceptions on page J1-11039.
- aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom on page J1-11039.
- aarch64/debug/pmu/AArch64.CheckForPMUOverflow on page J1-11039.
- aarch64/debug/pmu/AArch64.ClearEventCounters on page J1-11040.
- aarch64/debug/pmu/AArch64.CountPMUEvents on page J1-11040.
- aarch64/debug/pmu/AArch64.GetNumEventCountersAccessible on page J1-11042.
- aarch64/debug/pmu/AArch64.IncrementEventCounter on page J1-11042.
• aarch64/debug/pmud/AArch64.PMUCycle on page J1-11043.
• aarch64/debug/pmud/AArch64.PMUSwIncrement on page J1-11044.
• aarch64/debug/statisticalprofiling/CollectContextIDR1 on page J1-11044.
• aarch64/debug/statisticalprofiling/CollectContextIDR2 on page J1-11044.
• aarch64/debug/statisticalprofiling/CollectPhysicalAddress on page J1-11044.
• aarch64/debug/statisticalprofiling/CollectTimeStamp on page J1-11044.
• aarch64/debug/statisticalprofiling/OpType on page J1-11045.
• aarch64/debug/statisticalprofiling/ProfilingBufferEnabled on page J1-11045.
• aarch64/debug/statisticalprofiling/ProfilingBufferOwner on page J1-11046.
• aarch64/debug/statisticalprofiling/ProfilingSynchronizationBarrier on page J1-11046.
• aarch64/debug/statisticalprofiling/StatisticalProfilingEnabled on page J1-11047.
• aarch64/debug/statisticalprofiling/TimeStamp on page J1-11048.
• aarch64/debug/takeexceptiondbg/AArch64.TakeExceptionInDebugState on page J1-11048.
• aarch64/debug/watchpoint/AArch64.WatchpointByteMatch on page J1-11049.
• aarch64/debug/watchpoint/AArch64.WatchpointMatch on page J1-11050.

aarch64/debug/brbe/BRBCycleCountingEnabled

// BRBCycleCountingEnabled()
// =========================
// Returns TRUE if the BRBINF<n>_EL1.{CCU, CC} fields are valid, FALSE otherwise.

    boolean BRBCycleCountingEnabled()
    {
        if EL2Enabled() && BRBCR_EL2.CC == '0' then return FALSE;
        if BRBCR_EL1.CC == '0' then return FALSE;
        return TRUE;
    }

aarch64/debug/brbe/BRBEBranch

// BRBEBranch()
// ============
// Called to write branch record for the following branches when BRB is active:
// // direct branches,
// // indirect branches,
// // direct branches with link,
// // indirect branches with link,
// // returns from subroutines.

    BRBEBranch(BranchType br_type, boolean cond, bits(64) target_address)
    {
        if BranchRecordAllowed(PSTATE.EL) && FilterBranchRecord(br_type, cond) then
            bits(6) branch_type;
            case br_type of
                when BranchType_DIR
                    branch_type = if cond then '001000' else '000000';
                when BranchType_INDIR
                    branch_type = '000001';
                when BranchType_DIRCALL
                    branch_type = '000010';
                when BranchType_INDCALL
                    branch_type = '000011';
                when BranchType_RET
                    branch_type = '000101';
                otherwise
                    Unreachable();
            endcase;
            bit ccu;
            bits(14) cc;
            (ccu, cc) = BranchEncCycleCount();
            bit lastfailed = if HaveTME() then BRBFSCR_EL1.LASTFAILED else '0';
            bit transactional = if HaveTME() && TSTATE.depth > 0 then '1' else '0';
            bits(2) el = PSTATE.EL;
            bit mispredict = if BRBEMispredictAllowed() && BranchMispredict() then '1' else '0';
            UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, branch_type, el, mispredict, 
                '11', PC[], target_address);
    }
BRBFCR_EL1.LASTFAILED = '0';
PMUEvent(PMU_EVENT_BRB_FILTRATE);
return;

aarch64/debug/brbe/BRBEBranchOnISB

// BRBEBranchOnISB()
// ================
// Returns TRUE if ISBs generate Branch records, and FALSE otherwise.
boolean BRBEBranchOnISB()
return boolean IMPLEMENTATION_DEFINED "ISB generates Branch records";

aarch64/debug/brbe/BRBEDebugStateExit

// BRBEDebugStateExit()
// ====================
// Called to write Debug state exit branch record when BRB is active.
BRBEDebugStateExit(bits(64) target_address)
if BranchRecordAllowed(PSTATE.EL) then
// Debug state is a prohibited region, therefore ccu=1, cc=0, source_address=0
bits(6) branch_type = '111001';
bit ccu = '1';
bits(14) cc = Zeros(14);
bit lastfailed = if HaveTME() then BRBFCR_EL1.LASTFAILED else '0';
bit transactional = '0';
bits(2) el = PSTATE.EL;
bit mispredict = '0';
UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, branch_type, el, mispredict, '01', Zeros(64), target_address);
BRBFCR_EL1.LASTFAILED = '0';
PMUEvent(PMU_EVENT_BRB_FILTRATE);
return;

aarch64/debug/brbe/BRBEException

// BRBEException()
// ===============
// Called to write exception branch record when BRB is active.
BRBEException(Exception exception, bits(64) preferred_exception_return,
bits(64) target_address_in, bits(2) target_el, boolean trappedsyscallinst)
bits(64) target_address = target_address_in;
case target_el of
when EL3 if !HaveBRBEv1p1() || (MDCR_EL3.E3BREC == MDCR_EL3.E3BREW) then return;
when EL2 if BRBCR_EL2.EXCEPTION == '0' then return;
when EL1 if BRBCR_EL1.EXCEPTION == '0' then return;
boolean source_valid = BranchRecordAllowed(PSTATE.EL);
boolean target_valid = BranchRecordAllowed(target_el);
if source_valid || target_valid then
bits(6) branch_type;
case exception of
when Exception_Uncategorized branch_type = '100011'; // Trap
when Exception_WFxTrap branch_type = '100011'; // Trap
when Exception_CP15RTTrap branch_type = '100011'; // Trap
when Exception_CP15RRTrap branch_type = '100011'; // Trap
when Exception_CP14RTTrap branch_type = '100011'; // Trap
when Exception_CP14DTTrap branch_type = '100011'; // Trap
when Exception_AdvSIMDFPAccessTrap branch_type = '100011'; // Trap
when Exception_FPIDTrap branch_type = '100011'; // Trap
when Exception_PACTrap branch_type = '100011'; // Trap
when Exception_CP14RRRTTrap branch_type = '100011'; // Trap
when Exception_BranchTarget branch_type = '101011'; // Inst Fault
when Exception_IllegalState branch_type = '100011'; // Trap
when Exception_SupervisorCall
  if !trappedsyscallinst then branch_type = '100010'; // Call
  else branch_type = '100011'; // Trap
when Exception_HypervisorCall branch_type = '100010'; // Call
when Exception_MonitorCall
  if !trappedsyscallinst then branch_type = '100010'; // Call
  else branch_type = '100011'; // Trap
when Exception_SystemRegisterTrap branch_type = '100011'; // Trap
when Exception_SVEAccessTrap branch_type = '100011'; // Trap
when Exception_SMEAccessTrap branch_type = '100011'; // Trap
when Exception_ERetTrap branch_type = '100011'; // Trap
when Exception_PACFail branch_type = '101100'; // Data Fault
when Exception_InstructionAbort branch_type = '101011'; // Inst Fault
when Exception_PCAlignment branch_type = '101010'; // Alignment
when Exception_DataAbort branch_type = '101100'; // Data Fault
when Exception_NV2DataAbort branch_type = '101100'; // Data Fault
when Exception_SPAlignment branch_type = '101010'; // Alignment
when Exception_FPTrappedException branch_type = '100011'; // Trap
when Exception_SError branch_type = '100100'; // System Error
when Exception_Breakpoint branch_type = '100110'; // Inst debug
when Exception_SoftwareStep branch_type = '100110'; // Inst debug
when Exception_PACpoint branch_type = '100111'; // Data debug
when Exception_NV2PACpoint branch_type = '100111'; // Data debug
when Exception_INT branch_type = '100110'; // Inst debug
when Exception_FIQ branch_type = '100111'; // FIQ
when Exception_MemCpyMemSet branch_type = '100011'; // Trap
otherwise Unreachable();

bit ccu;
bits(14) cc;
(ccu, cc) = BranchEncCycleCount();
bit lastfailed = if HaveTME() then BRBFCR_EL1.LASTFAILED else '0';
bit transactional = if source_valid && HaveTME() && TSTATE.depth > 0 then '1' else '0';
bits(2) el = if target_valid then target_el else '00';
bit mispredict = '0';
bit sv = if source_valid then '1' else '0';
bit tv = if target_valid then '1' else '0';
bits(64) source_address = if source_valid then preferred_exception_return else Zeros(64);

if !target_valid then
  target_address = Zeros(64);
else
  target_address = AArch64.BranchAddr(target_address, target_el);

UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, branch_type, el, mispredict, sv:tv, source_address, target_address);

BRBFCR_EL1.LASTFAILED = '0';
PMUEvent(PMU_EVENT_BRB_FILTRATE);
return;
aarch64/debug/brbe/BRBEExceptionReturn

// BRBEExceptionReturn()
// =====================
// Called to write exception return branch record when BRB is active.

BRBEExceptionReturn(bits(64) target_address_in, bits(2) source_el)
bits(64) target_address = target_address_in;
case source_el of
  when EL3 if !HaveBRBEv1p1() || (MDCR_EL3.E3BREC == MDCR_EL3.E3BREW) then return;
  when EL2 if BRBCR_EL2.ERTN == '0' then return;
  when EL1 if BRBCR_EL1.ERTN == '0' then return;

boolean source_valid = BranchRecordAllowed(source_el);
boolean target_valid = BranchRecordAllowed(PSTATE.EL);

if source_valid || target_valid then
  bits(6) branch_type = '000111';
  bit ccu;
  bits(14) cc;
  (ccu, cc) = BranchEncCycleCount();
  bit lastfailed = if HaveTME() then BRBFCR_EL1.LASTFAILED else '0';
  bit transactional = if source_valid && HaveTME() && TSTATE.depth > 0 then '1' else '0';
  bits(2) el = if target_valid then PSTATE.EL else '00';
  bit mispredict = if source_valid && BRBEMispredictAllowed() && BranchMispredict() then '1' else '0';
  bit sv = if source_valid then '1' else '0';
  bit tv = if target_valid then '1' else '0';
  bits(64) source_address = if source_valid then PC[] else Zeros(64);
  if !target_valid then
    target_address = Zeros(64);
  UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, branch_type, el, mispredict, sv:tv, source_address, target_address);

BRBFCR_EL1.LASTFAILED = '0';
PMUEvent(PMU_EVENT_BRB_FILTRATE);
return;

aarch64/debug/brbe/BRBEFreeze

// BRBEFreeze()
// ============
// Generates BRBE freeze event.

BRBEFreeze()

BRBFCR_EL1.PAUSED = '1';
BRBTS_EL1 = GetTimestamp(BRBETimestamp());

aarch64/debug/brbe/BRBEISB

// BRBEISB()
// =========
// Handles ISB instruction for BRBE.

BRBEISB()

boolean branch_conditional = FALSE;
BRBEBranche(BranchType_DIR, branch_conditional, PC[] + 4);
aarch64/debug/brbe/BRBEMispredictAllowed

// BRBEMispredictAllowed()
// =======================
// Returns TRUE if the recording of branch misprediction is allowed, FALSE otherwise.

boolean BRBEMispredictAllowed()
if EL2Enabled() && BRBCR_EL2.MPRED == '0' then return FALSE;
if BRBCR_EL1.MPRED == '0' then return FALSE;
return TRUE;

aarch64/debug/brbe/BRBETimeStamp

// BRBETimeStamp()
// ===============
// Returns captured timestamp.

TimeStamp BRBETimeStamp()
if HaveEL(EL2) then
    TS_el2 = BRBCR_EL2.TS;
    if !HaveECVExt() && TS_el2 == '10' then
        // Reserved value
        (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP, 2);
        case TS_el2 of
            when '00' // Falls out to check BRBCR_EL1.TS
                return TimeStamp_Virtual;
            when '01'
                return TimeStamp_OffsetPhysical;
            when '10'
                assert HaveECVExt(); // Otherwise ConstrainUnpredictableBits removes this case
                return TimeStamp_Physical;
            when '11'
                return TimeStamp_Physical;
            otherwise
                Unreachable();           // ConstrainUnpredictableBits removes this case
    TS_el1 = BRBCR_EL1.TS;
    if TS_el1 == '00' || (!HaveECVExt() && TS_el1 == '10') then
        // Reserved value
        (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable_EL1TIMESTAMP, 2);
        case TS_el1 of
            when '01'
                return TimeStamp_Virtual;
            when '10'
                return TimeStamp_OffsetPhysical;
            when '11'
                return TimeStamp_Physical;
            otherwise
                Unreachable();           // ConstrainUnpredictableBits removes this case

aarch64/debug/brbe/BRB_IALL

// BRB_IALL()
// ==========
// Called to perform invalidation of branch records

BRB_IALL()
for i = 0 to GetBRBENumRecords() - 1
    Records_SRC[i] = Zeros(64);
    Records_TGT[i] = Zeros(64);
    Records_INF[i] = Zeros(64);
aarch64/debug/brbe/BRB_INJ

// BRB_INJ()
// =========
// Called to perform manual injection of branch records.

BRB_INJ()
UpdateBranchRecordBuffer(BRBINFINJ_EL1.CCU, BRBINFINJ_EL1.CC, BRBINFINJ_EL1.LASTFAILED,
BRBINFINJ_EL1.T, BRBINFINJ_EL1.TYPE, BRBINFINJ_EL1.EL,
BRBINFINJ_EL1.MPRED, BRBINFINJ_EL1.VALID, BRBSRCINJ_EL1.ADDRESS,
BRBTGTINJ_EL1.ADDRESS);

BRBINFINJ_EL1 = bits(64) UNKNOWN;
BRBSRCINJ_EL1 = bits(64) UNKNOWN;
BRBTGTINJ_EL1 = bits(64) UNKNOWN;

if ConstrainUnpredictableBool(Unpredictable_BRBFILTRATE) then PMUEvent(PMU_EVENT_BRB_FILTRATE);

aarch64/debug/brbe/Branch

type BRBSRCType;
type BRBTGTType;
type BRBINFType;

aarch64/debug/brbe/BranchEncCycleCount

// The first return result is '1' if either of the following is true, and '0' otherwise:
// - This is the first Branch record after the PE exited a Prohibited Region.
// - This is the first Branch record after cycle counting has been enabled.
// If the first return return is '0', the second return result is the encoded cycle count
// since the last branch.
// The format of this field uses a mantissa and exponent to express the cycle count value.
// - bits[7:0] indicate the mantissa M.
// - bits[13:8] indicate the exponent E.
// The cycle count is expressed using the following function:
// cycle_count = (if IsZero(E) then UInt(M) else UInt('1':M:Zeros(UInt(E)-1)))
// A value of all ones in both the mantissa and exponent indicates the cycle count value
// exceeded the size of the cycle counter.
// If the cycle count is not known, the second return result is zero.
(bit, bits(14)) BranchEncCycleCount();

aarch64/debug/brbe/BranchMispredict

// Returns TRUE if the branch being executed was mispredicted, FALSE otherwise.
boolean BranchMispredict();

aarch64/debug/brbe/BranchRawCycleCount

// If the cycle count is known, the return result is the cycle count since the last branch.
integer BranchRawCycleCount();

aarch64/debug/brbe/BranchRecordAllowed

// BranchRecordAllowed()
// ================
// Returns TRUE if branch recording is allowed, FALSE otherwise.

boolean BranchRecordAllowed(bits(2) el)
if ELUsingArch32(el) then
    return FALSE;
if BRBFCR_EL1.PAUSED == '1' then
    return FALSE;

if el == EL3 && HaveBRBEv1p1() then
    return (MDCR_EL3.E3BREC != MDCR_EL3.E3BREW);

if HaveEL(EL3) && (MDCR_EL3.SBRBE == '00' ||
  (CurrentSecurityState() == SS_Secure && MDCR_EL3.SBRBE == '01')) then
    return FALSE;

case el of
  when EL3    return FALSE;                // FEAT_BRBEv1p1 not implemented
  when EL2    return BRBCR_EL2.E2BRE == '1';
  when EL1    return BRBCR_EL1.E1BRE == '1';
  when EL0
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      return BRBCR_EL2.E0HBRE == '1';
    else
      return BRBCR_EL1.E0BRE == '1';

aarch64/debug/brbe/Contents

array [0..63] of BRBSRCType Records_SRC;
array [0..63] of BRBTGTType Records_TGT;
array [0..63] of BRBINFType Records_INF;

aarch64/debug/brbe/FilterBranchRecord

// FilterBranchRecord()
// ====================
// Returns TRUE if the branch record is not filtered out, FALSE otherwise.

boolean FilterBranchRecord(BranchType br, boolean cond)
  case br of
    when BranchType_DIRCALL
      return BRBFCR_EL1.DIRCALL != BRBFCR_EL1.EnI;
    when BranchType_INDCELL
      return BRBFCR_EL1.INDCALL != BRBFCR_EL1.EnI;
    when BranchType_RET
      return BRBFCR_EL1.RTN != BRBFCR_EL1.EnI;
    when BranchType_DIR
      if cond then
        return BRBFCR_EL1.CONDDIR != BRBFCR_EL1.EnI;
      else
        return BRBFCR_EL1.DIRECT != BRBFCR_EL1.EnI;
    when BranchType_INDIR
      return BRBFCR_EL1.INDIRECT != BRBFCR_EL1.EnI;
    otherwise  Unreachable();
    return FALSE;

aarch64/debug/brbe/FirstBranchAfterProhibited

// Returns TRUE if branch recorded is the first branch after a prohibited region, // FALSE otherwise.
FirstBranchAfterProhibited();
aarch64/debug/brbe/GetBRBENumRecords

// GetBRBENumRecords()
// ===================
// Returns the number of branch records implemented.

integer GetBRBENumRecords()
    assert UInt(BRBIDR0_EL1.NUMREC) IN {0x08, 0x10, 0x20, 0x40};
    return integer IMPLEMENTATION_DEFINED "Number of BRB records";

aarch64/debug/brbe/Getter

// Getter functions for branch records
// ===================================
// Functions used by MRS instructions that access branch records

BRBSRCType BRBSRC_EL1[integer n]
    assert n IN {0..31};
    integer record = UInt(BRBFCR_EL1.BANK:n<4:0>);
    if record < GetBRBENumRecords() then
        return Records_SRC[record];
    else
        return Zeros(64);

BRBTGTType BRBTGT_EL1[integer n]
    assert n IN {0..31};
    integer record = UInt(BRBFCR_EL1.BANK:n<4:0>);
    if record < GetBRBENumRecords() then
        return Records_TGT[record];
    else
        return Zeros(64);

BRBINFType BRBINF_EL1[integer n]
    assert n IN {0..31};
    integer record = UInt(BRBFCR_EL1.BANK:n<4:0>);
    if record < GetBRBENumRecords() then
        return Records_INF[record];
    else
        return Zeros(64);

aarch64/debug/brbe/ShouldBRBEFreeze

// ShouldBRBEFreeze()
// =================
// Returns TRUE if the BRBE freeze event conditions have been met, and FALSE otherwise.

boolean ShouldBRBEFreeze()
    if !BranchRecordAllowed(PSTATE.EL) then return FALSE;
    integer cntrs = GetNumEventCounters();
    if HaveEL_EL2 then
        integer hpmn = UInt(MDCR_EL2.HPMN);
        if BRBFCR_EL1.FZP == '1' && hpmn != 0 && !IsZero(PMOVSLR_EL0<hpmn-1:0>) then
            return TRUE;
        if BRBFCR_EL1.FZP == '1' && cntrs != hpmn && !IsZero(PMOVSLR_EL0<cntrs-1:hpmn>) then
            return TRUE;
        if BRBFCR_EL1.FZP == '1' && !IsZero(PMOVSLR_EL0<cntrs-1:0>) then
            return TRUE;
    return FALSE;
aarch64/debug/brbe/UpdateBranchRecordBuffer

// UpdateBranchRecordBuffer()
// =========================
// Add a new Branch record to the buffer.

UpdateBranchRecordBuffer(bit ccu, bits(14) cc, bit lastfailed, bit transactional, bits(6) branch_type,
bits(2) el, bit mispredict, bits(2) valid, bits(64) source_address, bits(64) target_address)

// Shift the Branch Records in the buffer
for i = GetBRBENumRecords() - 1 downto 1
    Records_SRC[i] = Records_SRC[i - 1];
    Records_TGT[i] = Records_TGT[i - 1];
    Records_INF[i] = Records_INF[i - 1];

    Records_INF[0].CCU = ccu;
    Records_INF[0].CC = cc;
    Records_INF[0].EL = el;
    Records_INF[0].VALID = valid;
    Records_INF[0].T = transactional;
    Records_INF[0].LASTFAILED = lastfailed;
    Records_INF[0].MPRED = mispredict;
    Records_INF[0].TYPE = branch_type;

    Records_SRC[0] = source_address;
    Records_TGT[0] = target_address;

return;

aarch64/debug/breakpoint/AArch64.BreakpointMatch

// AArch64.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress, AccType acctype, integer size)
assert !ELUsingAArch32(S1TranslationRegime());
assert n < NumBreakpointsImplemented();
enabled = DBGBCR_EL1[n].E == '1';
ispriv = PSTATE.EL != EL0;
linked = DBGBCR_EL1[n].BT IN {'0x01'};
isbreakpnt = TRUE;
linked_to = FALSE;
ssce = if HaveRME() then DBGBCR_EL1[n].SSCE else '0';
state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, ssce, DBGBCR_EL1[n].HMC, DBGBCR_EL1[n].PMC,
linked, DBGBCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);
value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to);
if HaveAArch32() && size == 4 then                     // Check second halfword
    // If the breakpoint address and BAS of an Address breakpoint match the address of the
    // second halfword of an instruction, but not the address of the first halfword, it is
    // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
    // event.
match_i = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to);
    if !value_match && match_i then
        value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
    if vaddress[i] == '1' && DBGBCR_EL1[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR_EL1[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGVR_EL1[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
match = value_match && state_match && enabled;
return match;

aarch64/debug/breakpoint/AArch64.BreakpointValueMatch

// AArch64.BreakpointValueMatch()
// ==============================

boolean AArch64.BreakpointValueMatch(integer n_in, bits(64) vaddress, boolean linked_to)

// "n" is the identity of the breakpoint unit to match against.
// "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
// matching breakpoints.
// "linked_to" is TRUE if this is a call from StateMatch for linking.
integer n = n_in;

// If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
// no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
if n >= NumBreakpointsImplemented() then
  Constraint c;
  (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1,
Unpredictable_BPNOTIMPL);
  assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
  if c == Constraint_DISABLED then return FALSE;

// If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
// call from StateMatch for linking).
if DBGCCR_EL1[n].E == '0' then return FALSE;

context_aware = (n >= (NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented()));

// If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
dbgtype = DBGCCR_EL1[n].BT;
  Constraint c;
if ((dbgtype IN {'011x','11xx'} && !HaveVirtHostExt() && !HaveV82Debug()) ||    // Context matching
  dbgtype IN {'010x'} ||                                                      // Reserved
  (!dbgtype IN {'0x0x'}) && !context_aware) ||                                  // Context
  dbgtype IN {'1xxx'} && !HaveEL(EL2)) then                                 // EL2 extension
  (c, dbgtype) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE, 4);
  assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
  if c == Constraint_DISABLED then return FALSE;

// Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
match_addr = (dbgtype IN {'0x0x'});
match_vmvid = (dbgtype IN {'10xx'});
match_cid = (dbgtype IN {'001x'});
match_cid1 = (dbgtype IN {'10lx', 'xl1x'});
match_cid2 = (dbgtype IN {'1lx'});
linked = (dbgtype IN {'xxx1'});

// If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
// VMID and/or context ID match, of if not context-aware. The above assertions mean that the
// code can just test for match_addr == TRUE to confirm all these things.
if linked_to && (!linked || match_addr) then return FALSE;

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linked && !match_addr then return FALSE;

// Do the comparison.
boolean BVR_match;
if match_addr then
boolean byte_select_match;
byte = UInt(vaddress<1:0>);
if HaveAArch32() then
// T32 instructions can be executed at EL0 in an AArch64 translation regime.
assert byte IN {0,2}; // "vaddress" is halfword aligned
byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
else
assert byte == 0; // "vaddress" is word aligned
byte_select_match = TRUE; // DBGBCR_EL1[n].BAS<byte> is RES1
// If the DBGxVR<n>.RESS field bits are not a sign extension of the MSB
// of DBGxVR<n>.VA, it is UNPREDICTABLE whether they appear to be
// included in the match.
// If 'vaddress' is outside of the current virtual address space, then the access
// generates a Translation fault.
top = AArch64.VAMax();
if !IsOnes(DBGVR_EL1[n]<63:top>) && !IsZero(DBGVR_EL1[n]<63:top>) then
if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
top = 63;
BVR_match = (vaddress<top:2> == DBGBVR_EL1[n]<top:2>) && byte_select_match;
elsif match_cid then
if IsInHost() then
BVR_match = (CONTEXTIDR_EL2<31:0> == DBGBVR_EL1[n]<31:0>);
else
BVR_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1<31:0> == DBGBVR_EL1[n]<31:0>);
elsif match_cid1 then
BVR_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() && CONTEXTIDR_EL1<31:0> ==
DBGBVR_EL1[n]<31:0>);
boolean BXVR_match;
if match_vmid then
bits(16) vmid;
bits(16) bvr_vmid;
if !Have16bitVMID() || VTCR_EL2.VS == '0' then
vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
bvr_vmid = ZeroExtend(DBGVR_EL1[n]<39:32>, 16);
else
vmid = VTTBR_EL2.VMID;
bvr_vmid = DBGVR_EL1[n]<47:32>;
BXVR_match = (PSTATE.EL != EL3 && !IsInHost() &&
VMID == bvr_vmid);
elsif match_cid2 then
BXVR_match = (PSTATE.EL != EL3 && (HaveVirtHostExt() || HaveV82Debug())
&&
DBGBVR_EL1[n]<63:32> == CONTEXTIDR_EL2<31:0>);
bvr_match_valid = (match_addr || match_cid || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);
match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);
return match;

aarch64/debug/breakpoint/AArch64.StateMatch

// AArch64.StateMatch()
// ===============
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) SSC_in, bit SSCE_in, bit HMC_in,
bits(2) PxC_in, boolean linked_in, bits(4) LBN,
boolean isbreakpt, AccType acctype, boolean ispriv)
if !HaveRME() then assert SSCE_in == '0';
// "SSC_in", "SSCE_in", "HMC_in", "PxC_in" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
// "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
// "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
// "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.
bits(2) SSC = SSC_in;
bit SSCE = SSCE_in;
bit HMC = HMC_in;
bits(2) PxC = PxC_in;
boolean linked = linked_in;

// If parameters are set to a reserved type, behaves as either disabled or a defined type
Constraint c;
(c, SSC, SSCE, HMC, PxC) = CheckValidStateMatch(SSC, SSCE, HMC, PxC, isbreakpnt);
if c == Constraint_DISABLED then return FALSE;
// Otherwise the HMC,SSC,SSCE,PxC values are either valid or the values returned by
// CheckValidStateMatch are valid.
EL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
EL2_match = HaveEL(EL2) && ((HMC == '1' && (SSC:PxC != '1000')) || SSC == '11');
EL1_match = PxC<0> == '1';
EL0_match = PxC<1> == '1';

boolean priv_match;
if HaveNV2Ext() && accctype == AccType_NV2REGISTER && !isbreakpnt then
  priv_match = EL2_match;
elsif !ispriv && !isbreakpnt then
  priv_match = EL0_match;
else
  case PSTATE.EL of
    when EL3  priv_match = EL3_match;
    when EL2  priv_match = EL2_match;
    when EL1  priv_match = EL1_match;
    when EL0  priv_match = EL0_match;
  end_case
end_case

boolean security_state_match;
ss = CurrentSecurityState();
case SSCE:SSC of
  when '000' security_state_match = HMC == '1' || ss != SS_Root;
  when '001' security_state_match = ss == SS_NonSecure;
  when '010' security_state_match = (HMC == '1' && ss != SS_Root) || ss == SS_Secure;
  when '011' security_state_match = (HMC == '1' && ss != SS_Root) || ss == SS_Secure;
  when '101' security_state_match = ss == SS_Realm;
end_case

integer lb;
if linked then
  // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
  // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
  // UNKNOWN breakpoint that is context-aware.
  lb = UInt(LBN);
  first_ctx_cmp = NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented();
  last_ctx_cmp = NumBreakpointsImplemented() - 1;
  if (lb < first_ctx_cmp || lb > last_ctx_cmp) then
    (c, lb) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable_BPNOTCTXCMP);
  end_case
end_case
assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
case c of
  when Constraint_DISABLED  return FALSE; // Disabled
  when Constraint_NONE      linked = FALSE; // No linking
  // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint
end_case

boolean linked_match;
if linked then
  vaddress = bits(64) UNKNOWN;
  linked_to = TRUE;
  linked_match = AArch64.BreakpointValueMatch(lb, vaddress, linked_to);
end_case

return priv_match && security_state_match && (!linked || linked_match);

AArch64.GenerateDebugExceptions

// AArch64.GenerateDebugExceptions()
// -----------------------------------

boolean AArch64.GenerateDebugExceptions()
    ss = CurrentSecurityState();
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, ss, PSTATE.D);

AArch64.GenerateDebugExceptionsFrom

// AArch64.GenerateDebugExceptionsFrom()
// -------------------------------------

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from_el, SecurityState from_state, bit mask)
    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;
    route_to_el2 = (HaveEL(EL2) && (from_state != SS_Secure || IsSecureEL2Enabled()) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));
    target = (if route_to_el2 then EL2 else EL1);
    boolean enabled;
    if HaveEL(EL3) && from_state == SS_Secure then
        enabled = MDCR_EL3.SDD == '0';
    if from_el == EL0 && ELUsingAArch32(EL1) then
        enabled = enabled || SDER32_EL3.SUIDEN == '1';
    else
        enabled = TRUE;
    if from_el == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from_el);
    return enabled;

AArch64.CheckForPMUOverflow

// AArch64.CheckForPMUOverflow()
// -----------------------------

boolean pmuirq;
    bit E;
    if ShouldBRBEFreeze() then
        BRBEFreeze();
    pmuirq = PMCR_EL0.E == '1' && PMINTENSET_EL1.C == '1' && PMOVSSET_EL0.C == '1';
    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            E = if AArch64.PMUCounterIsHyp(idx) then MDCR_EL2.HPME else PMCR_EL0.E;
            if E == '1' && PMINTENSET_EL1<idx> == '1' && PMOVSSET_EL0<idx> == '1' then pmuirq = TRUE;
    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);
    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)

reference

armv8 Pseudocode
J1.1 Pseudocode for AArch64 operation

ARM DDI 0487I.a
Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential
ID081822
J1-11039
aarch64/debug/pmu/AArch64.ClearEventCounters

// AArch64.ClearEventCounters()
// ============================
// Zero all the event counters.

AArch64.ClearEventCounters()
    integer counters = AArch64.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMEVCNTR_EL0[idx] = Zeros(64);

aarch64/debug/pmu/AArch64.CountPMUEvents

// AArch64.CountPMUEvents()
// ========================
// Return TRUE if counter "idx" should count its event.
// For the cycle counter, idx == CYCLE_COUNTER_ID.

boolean AArch64.CountPMUEvents(integer idx)
    constant integer num_counters = GetNumEventCounters();
    assert idx == CYCLE_COUNTER_ID || idx < num_counters;
    boolean debug;
    boolean enabled;
    boolean prohibited;
    boolean filtered;
    boolean frozen;
    boolean resvd_for_el2;
    bit E;
    bit spme;
    bits(32) ovflws;
    // Event counting is disabled in Debug state
    debug = Halted();

    // Software can reserve some counters for EL2
    resvd_for_el2 = AArch64.PMUCounterIsHyp(idx);
    ss = CurrentSecurityState();

    // Main enable controls
    if idx == CYCLE_COUNTER_ID then
        enabled = PMCR_EL0.E == '1' && PMCNTENSET_EL0.C == '1';
    else
        E = if resvd_for_el2 then MDCR_EL2.HPME else PMCR_EL0.E;
        enabled = E == '1' && PMCNTENSET_EL0<idx> == '1';

    // Event counting is allowed unless it is prohibited by any rule below
    prohibited = FALSE;
    // Event counting in Secure state is prohibited if all of:
    // * EL3 is implemented
    // * MDCR_EL3.SPME == 0, and either:
    //   - FEAT_PMUv3p7 is not implemented
    //   - MDCR_EL3.MMPXM == 0
    if HaveEL(EL3) && ss == SS_Secure then
        if HavePMUv3p7() then
            prohibited = MDCR_EL3.SPME == '0';
        else
            prohibited = MDCR_EL3.SPME == '0';

    // Event counting at EL3 is prohibited if all of:
    // * FEAT_PMUv3p7 is implemented
    // * One of the following is true:
    //   - MDCR_EL3.SPME == 0
    //   - PMNx is not reserved for EL2
    // * MDCR_EL3.MMPXM == 1
    if !prohibited && PSTATE.EL == EL3 && HavePMUv3p7() then
prohibited = MDCR_EL3.MPMX == '1' && (MDCR_EL3.SPME == '0' || !resvd_for_el2);

// Event counting at EL2 is prohibited if all of:
// * The HPMD Extension is implemented
// * PMNx is not reserved for EL2
// * MDCR_EL2.HPMD == 1
if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
  prohibited = MDCR_EL2.HPMD == '1';

// The IMPLEMENTATION DEFINED authentication interface might override software
if prohibited && !HaveNoSecurePMUDisableOverride() then
  prohibited = !ExternalSecureNoninvasiveDebugEnabled();

// Event counting might be frozen
frozen = FALSE;

// If FEAT_PMUv3p7 is implemented, event counting can be frozen
if HavePMUv3p7() then
  hpmn = UInt(MDCR_EL2.HPMN);
  ovflws = ZeroExtend(PMOVSSET_EL0<num_counters-1:0>, 32);
  bit FZ;
  if resvd_for_el2 then
    FZ = MDCR_EL2.HPMFZO;
    ovflws<hpmn-1:0> = Zeros(hpmn);
  else
    FZ = PMCR_EL0.FZO;
    if HaveEL(EL2) && hpmn < num_counters then
      ovflws<num_counters-1:hpmn> = Zeros(num_counters - hpmn);
    frozen = (FZ == '1') && !IsZero(ovflws);

// PMCR_EL0.DP disables the cycle counter when event counting is prohibited
if (prohibited || frozen) && idx == CYCLE_COUNTER_ID then
  enabled = enabled && (PMCR_EL0.DP == '0');
// Otherwise whether event counting is prohibited does not affect the cycle counter
  prohibited = FALSE;
  frozen = FALSE;

// If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
// This is not overridden by PMCR_EL0.DP.
if HavePMUv3p5() && idx == CYCLE_COUNTER_ID then
  if (HaveEL(EL3) && ss == SS_Secure &&
      MDCR_EL3.SCCD == '1') then
    prohibited = TRUE;
  if PSTATE.EL == EL2 && MDCR_EL2.HCCD == '1' then
    prohibited = TRUE;

// If FEAT_PMUv3p7 is implemented, cycle counting an be prohibited at EL3.
// This is not overridden by PMCR_EL0.DP.
if HavePMUv3p7() && idx == CYCLE_COUNTER_ID then
  if PSTATE.EL == EL3 && MDCR_EL3.MCCD == '1' then
    prohibited = TRUE;

// Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH, RLK, RLU, RLH} bits
filter = if idx == CYCLE_COUNTER_ID then PMCCFILTR_EL0<31:0> else PMEVTYPER_EL0[idx]<31:0>;

P = filter<31>;
U = filter<30>;
NSK = if HaveEL(EL3) then filter<29> else '0';
NSU = if HaveEL(EL3) then filter<28> else '0';
NSH = if HaveEL(EL2) then filter<27> else '0';
M = if HaveEL(EL3) then filter<26> else '0';
SH = if HaveEL(EL3) && HaveSecureEL2Ext() then filter<24> else '0';
RLK = if HaveRME() then filter<22> else '0';
RLU = if HaveRME() then filter<21> else '0';
RLH = if HaveRME() then filter<20> else '0';
ss = CurrentSecurityState();
case PSTATE.EL of
when EL0
    case ss of
        when SS_NonSecure filtered = U != NSU;
        when SS_Secure filtered = U == '1';
        when SS_Realm filtered = U != RLU;
    when EL1
    case ss of
        when SS_NonSecure filtered = P != NSK;
        when SS_Secure filtered = P == '1';
        when SS_Realm filtered = P != RLK;
    when EL2
    case ss of
        when SS_NonSecure filtered = NSH == '0';
        when SS_Secure filtered = NSH == SH;
        when SS_Realm filtered = NSH == RLH;
    when EL3
    filtered = M != P;
    return !debug && enabled && !prohibited && !filtered && !frozen;

aarch64/debug/pmu/AArch64.GetNumEventCountersAccessible

// AArch64.GetNumEventCountersAccessible()
// ===============================
// Return the number of event counters that can be accessed at the current Exception level.

integer AArch64.GetNumEventCountersAccessible()
    integer n;
    integer total_counters = GetNumEventCounters();
    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
        n = UInt(MDCR_EL2.HPMN);
        if n > total_counters || (!HaveFeatHPMN0() && n == 0) then
            (-, n) = ConstrainUnpredictableInteger(0, total_counters,
                Unpredictable_PMUEVENTCOUNTER);
        else
            n = total_counters;
    return n;

aarch64/debug/pmu/AArch64.IncrementEventCounter

// AArch64.IncrementEventCounter()
// ===============================
// Increment the specified event counter by the specified amount.

AArch64.IncrementEventCounter(integer idx, integer increment)
    integer old_value;
    integer new_value;
    integer ovflw;
    bit lp;
    old_value = UInt(PMEVCNTR_EL0[idx]);
    new_value = old_value + PMUCountValue(idx, increment);
    if HavePMUv3p5() then
        PMEVCNTR_EL0[idx] = new_value<63:0>;
        lp = if AArch64.PMUCounterIsHyp(idx) then MDCR_EL2.HLP else PMCR_EL0.LP;
        ovflw = if lp == '1' then 64 else 32;
    else
        PMEVCNTR_EL0[idx] = ZeroExtend(new_value<31:0>, 64);
        ovflw = 32;
    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET_EL0<idx> = '1';
PMOVSLR_EL0<idx> = '1';
   // Check for the CHAIN event from an even counter
if idx<0> == '0' && idx + 1 < GetNumEventCounters() && (!HavePMUv3p5() || lp == '0') then
   PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);

aarch64/debug/pmu/AArch64.PMUCounterIsHyp

   // AArch64.PMUCounterIsHyp
   // =======================
   // Returns TRUE if a counter is reserved for use by EL2, FALSE otherwise.
boolean AArch64.PMUCounterIsHyp(integer n)
   boolean resvd_for_el2;
   // Software can reserve some event counters for EL2
if n != CYCLE_COUNTER_ID && HaveEL(EL2) then
   resvd_for_el2 = n >= UInt(MDCR_EL2.HPMN);
   if UInt(MDCR_EL2.HPMN) > GetNumEventCounters() || (!HaveFeatHPMN0() && IsZero(MDCR_EL2.HPMN))
then
   resvd_for_el2 = boolean UNKNOWN;
else
   resvd_for_el2 = FALSE;
return resvd_for_el2;

aarch64/debug/pmu/AArch64.PMUCycle

   // AArch64.PMUCycle()
   // ==================
   // Called at the end of each cycle to increment event counters and
   // check for PMU overflow. In pseudocode, a cycle ends after the
   // execution of the operational pseudocode.
AArch64.PMUCycle()
   if !HavePMUv3() then
   return;
   PMUEvent(PMU_EVENT_CPU_CYCLES);
   integer counters = GetNumEventCounters();
   if counters != 0 then
      for idx = 0 to counters - 1
         if AArch64.CountPMUEvents(idx) then
            accumulated = PMUEventAccumulator[idx];
            AArch64.IncrementEventCounter(idx, accumulated);
            PMUEventAccumulator[idx] = 0;
            integer old_value;
            integer new_value;
            integer ovflw;
            if (AArch64.CountPMUEvents(CYCLE_COUNTER_ID) &&
               (!HaveAArch32() || PMCR_EL0.LC == '1' || PMCR_EL0.D == '0' || HasElapsed64Cycles())) then
               old_value = UInt(PMCCNTR_EL0);
               new_value = old_value + 1;
               PMCCNTR_EL0 = new_value<63:0>;
               if HaveAArch32() then
                  ovflw = if PMCR_EL0.LC == '1' then 64 else 32;
               else
                  ovflw = 64;
               if old_value<64:ovflw> != new_value<64:ovflw> then
                  PMOVSSET_EL0.C = '1';
                  PMOVSCLR_EL0.C = '1';
AArch64.CheckForPMUOverflow();

aarch64/debug/pmu/AArch64.PMUSwIncrement

// AArch64.PMUSwIncrement()
// ========================
// Generate PMU Events on a write to PMUSWINC_EL0.
AArch64.PMUSwIncrement(bits(32) sw_incr)
    integer counters = AArch64.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            if sw_incr<idx> == '1' then
                PMUEvent(PMU_EVENT_SW_INCR, 1, idx);

aarch64/debug/statisticalprofiling/CollectionsContextIDR1

// CollectContextIDR1()
// ====================
boolean CollectContextIDR1()
    if !StatisticalProfilingEnabled() then return FALSE;
    if PSTATE.EL == EL2 then return FALSE;
    if EL2Enabled() && HCR_EL2.TGE == '1' then return FALSE;
    return PMSCR_EL1.CX == '1';

aarch64/debug/statisticalprofiling/CollectionsContextIDR2

// CollectContextIDR2()
// ====================
boolean CollectContextIDR2()
    if !StatisticalProfilingEnabled() then return FALSE;
    if !EL2Enabled() then return FALSE;
    return PMSCR_EL2.CX == '1';

aarch64/debug/statisticalprofiling/CollectionsPhysicalAddress

// CollectPhysicalAddress()
// ========================
boolean CollectPhysicalAddress()
    if !StatisticalProfilingEnabled() then return FALSE;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        return PMSCR_EL2.PA == '1' && (owning_el == EL2 || PMSCR_EL1.PA == '1');
    else
        return PMSCR_EL1.PA == '1';

aarch64/debug/statisticalprofiling/CollectionsTimeStamp

// CollectTimeStamp()
// ==================
TimeStamp CollectTimeStamp()
    if !StatisticalProfilingEnabled() then return TimeStamp_None;
    (-, owning_el) = ProfilingBufferOwner();
if owning_el == EL2 then
    if PMSCR_EL2.TS == '0' then return TimeStamp_None;
    else
        if PMSCR_EL1.TS == '0' then return TimeStamp_None;
        bits(2) PCT_el1;
        if !HaveECVExt() then
            PCT_el1 = '0':PMSCR_EL1.PCT<0>;       // PCT<1> is RES0
        else
            PCT_el1 = PMSCR_EL1.PCT;
        if PCT_el1 == '10' then
            // Reserved value
            (-, PCT_el1) = ConstrainUnpredictableBits(Unpredictable_PMSCR_PCT, 2);
        if EL2Enabled() then
            bits(2) PCT_el2;
            if !HaveECVExt() then
                PCT_el2 = '0':PMSCR_EL2.PCT<0>;   // PCT<1> is RES0
            else
                PCT_el2 = PMSCR_EL2.PCT;
            if PCT_el2 == '10' then
                // Reserved value
                (-, PCT_el2) = ConstrainUnpredictableBits(Unpredictable_PMSCR_PCT, 2);
            case PCT_el2 of
                when '00'
                    return if IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
                when '01'
                    if owning_el == EL2 then return TimeStamp_Physical;
                when '11'
                    assert HaveECVExt();          // FEAT_ECV must be implemented
                    if owning_el == EL1 && PCT_el1 == '00' then
                        return if IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
                    else
                        return TimeStamp_OffsetPhysical;
                    otherwise
                        Unreachable();
            case PCT_el1 of
                when '00' return if IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
                when '01' return TimeStamp_Physical;
                when '11'
                    assert HaveECVExt();              // FEAT_ECV must be implemented
                    return TimeStamp_OffsetPhysical;
                otherwise Unreachable();
        if owning_el == EL1 then
            if PCT_el1 == '00' then
                return if IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
            else
                return TimeStamp_OffsetPhysical;
            otherwise
                Unreachable();
        else

aarch64/debug/statisticalprofiling/OpType

    enumeration OpType {
        OpType_Load,           // Any memory-read operation other than atomics, compare-and-swap, and swap
        OpType_Store,          // Any memory-write operation, including atomics without return
        OpType_LoadAtomic,     // Atomics with return, compare-and-swap and swap
        OpType_Branch,         // Software write to the PC
        OpType_Other           // Any other class of operation
    };

aarch64/debug/statisticalprofiling/ProfilingBufferEnabled

    // ProfilingBufferEnabled()
    // ==============

    boolean ProfilingBufferEnabled()  
    if !HaveStatisticalProfiling() then return FALSE;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    bits(2) state_bits;
    if HaveRME() then
state_bits = SCR_EL3.<NSE,NS>;
else
state_bits = '0' : SCR_EL3.NS;

boolean state_match;
case owning_ss of
    when SS_Secure  state_match = state_bits == '00';
    when SS_NonSecure state_match = state_bits == '01';
    when SS_Realm    state_match = state_bits == '11';
return (!ELUsingAArch32(owning_el) && state_match &&
                        PMBLIMITR_EL1.E == '1' && PMBSR_EL1.S == '0');

aarch64/debug/statisticalprofiling/ProfilingBufferOwner

// ProfilingBufferOwner()
// ================

(SecurityState, bits(2)) ProfilingBufferOwner()

    SecurityState owning_ss;

    if HaveEL(EL3) then
        bits(3) state_bits;
        if HaveRME() then
            state_bits = MDCR_EL3.<NSPBE,NSPB>;
            if state_bits IN {'10x'} then
                // Reserved value
                (-, state_bits) = ConstrainUnpredictableBits(Unpredictable_RESERVEDNSxB, 3);
            else
                state_bits = '0' : MDCR_EL3.NSPB;
            case state_bits of
                when '00x' owning_ss = SS_Secure;
                when '01x' owning_ss = SS_NonSecure;
                when '11x' owning_ss = SS_Realm;
            else
                owning_ss = if SecureOnlyImplementation() then SS_Secure else SS_NonSecure;
            bits(2) owning_el;
            if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
                owning_el = if MDCR_EL2.E2PB == '00' then EL2 else EL1;
            else
                owning_el = EL1;
        return (owning_ss, owning_el);

aarch64/debug/statisticalprofiling/ProfilingSynchronizationBarrier

// Barrier to ensure that all existing profiling data has been formatted, and profiling buffer
// addresses have been translated such that writes to the profiling buffer have been initiated.
// A following DSB completes when writes to the profiling buffer have completed.
ProfilingSynchronizationBarrier();

aarch64/debug/statisticalprofiling/SPECollectRecord

// SPECollectRecord()
// ===========
// Returns TRUE if the sampled class of instructions or operations, as
// determined by PMSFCR_EL1, are recorded and FALSE otherwise.

boolean SPECollectRecord(bits(64) events, integer total_latency, OpType optype)
assert StatisticalProfilingEnabled();

    bits(64) mask = 0xAA<63:0>;                             // Bits [7,5,3,1]
if HaveSVE() then mask<18:17> = Ones();                 // Predicate flags
if HaveTME() then mask<16> = '1';
if HaveStatisticalProfilingv1p1() then mask<11> = '1';  // Alignment Flag
if HaveStatisticalProfilingv1p2() then mask<6> = '1';   // Not taken flag

mask<63:48> = bits(16) IMPLEMENTATION_DEFINED "SPE mask 63:48";
mask<31:24> = bits(8) IMPLEMENTATION_DEFINED "SPE mask 31:24";
mask<15:12> = bits(4) IMPLEMENTATION_DEFINED "SPE mask 15:12";

// Check for UNPREDICTABLE case
if (HaveStatisticalProfilingv1p2() && PMSCR_EL1.<FnE,FE> == '11' &&
!IsZero(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)) then
if ConstrainUnpredictableBool() then
return FALSE;
else
// Filtering by event
if PMSCR_EL1.FE == '1' && !IsZero(PMSEVFR_EL1) then
  e = events AND mask;
  m = PMSEVFR_EL1 AND mask;
  if !IsZero(NOT(e) AND m) then return FALSE;

// Filtering by inverse event
if (HaveStatisticalProfilingv1p2() && PMSCR_EL1.FnE == '1' &&
!IsZero(PMSNEVFR_EL1)) then
  e = events AND mask;
  m = PMSEVFR_EL1 AND mask;
  if !IsZero(e AND m) then return FALSE;

// Filtering by type
if PMSCR_EL1.FT == '1' && !IsZero(PMSFCR_EL1.<B,LD,ST>) then
  case optype of
    when OpType_Branch
      if PMSCR_EL1.B == '0' then return FALSE;
    when OpType_Load
      if PMSCR_EL1.LD == '0' then return FALSE;
    when OpType_Store
      if PMSCR_EL1.ST == '0' then return FALSE;
    when OpType_LoadAtomic
      if PMSCR_EL1.<LD,ST> == '00' then return FALSE;
    otherwise
      return FALSE;

// Filtering by latency
if PMSCR_EL1.FL == '1' && !IsZero(PMSLATFR_EL1.MINLAT) then
  if total_latency < UInt(PMSLATFR_EL1.MINLAT) then
    return FALSE;

// Check for UNPREDICTABLE cases
if ((PMSCR_EL1.FE == '1' && IsZero(PMSEVFR_EL1 AND mask)) ||
  (PMSCR_EL1.FT == '1' && IsZero(PMSFCR_EL1.<B,LD,ST>)) ||
  (PMSCR_EL1.FL == '1' && IsZero(PMSLATFR_EL1.MINLAT))) then
return ConstrainUnpredictableBool();

if (HaveStatisticalProfilingv1p2() &&
((PMSCR_EL1.FnE == '1' && IsZero(PMSNEVFR_EL1 AND mask)) ||
(PMSCR_EL1.<FnE,FE> == '11' &&
  !IsZero(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)))) then
return ConstrainUnpredictableBool();

return TRUE;
boolean StatisticalProfilingEnabled()
    if !HaveStatisticalProfiling() || UsingAArch32() || !ProfilingBufferEnabled() then
        return FALSE;

    tge_set = EL2Enabled() && HCR_EL2.TGE == '1';
    (owning_ss, owning_el) = ProfilingBufferOwner();
    if (UInt(owning_el) < UInt(PSTATE.EL) || (tge_set && owning_el == EL1) ||
        owning_ss != CurrentSecurityState()) then
        return FALSE;

    bit spe_bit;
    case PSTATE.EL of
        when EL3  Unreachable();
        when EL2  spe_bit = PMSCR_EL2.E2SPE;
        when EL1  spe_bit = PMSCR_EL1.E1SPE;
        when EL0  spe_bit = (if tge_set then PMSCR_EL2.E0HSPE else PMSCR_EL1.E0SPE);
        return spe_bit == '1';

aarch64/debug/statisticalprofiling/TimeStamp

enumeration TimeStamp {
    TimeStamp_None,              // No timestamp
    TimeStamp_CoreSight,         // CoreSight time (IMPLEMENTATION DEFINED)
    TimeStamp_Physical,          // Physical counter value with no offset
    TimeStamp_OffsetPhysical,    // Physical counter value minus CNTPOFF_EL2
    TimeStamp_Virtual };         // Physical counter value minus CNTOFF_EL2

aarch64/debug/takeexceptiondbg/AArch64.TakeExceptionInDebugState

// AArch64.TakeExceptionInDebugState()
// ===============
// Take an exception in Debug state to an Exception level using AArch64.
AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception_in)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL); ExceptionRecord exception = exception_in;
    boolean sync_errors;
    boolean iesb_req;
    if HaveIESB() then
        sync_errors = SCTLR[target_el].IESB == '1';
        if HaveDoubleFaultExt() then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' || & target_el == EL3);
        if !ConstrainUnpredictableBool(Unpredictable_IESEBinDebug) then
            sync_errors = FALSE;
        else
            sync_errors = FALSE;
    if HaveTME() && TSTATE.depth > 0 then
        TMFailure cause;
        case exception.exctype of
            when Exception_SoftwareBreakpoint cause = TMFailure_DBG;
            when Exception_Breakpoint         cause = TMFailure_DBG;
            when Exception_Watchpoint         cause = TMFailure_DBG;
            when Exception_SoftwareStep       cause = TMFailure_DBG;
            otherwise                         cause = TMFailure_ERR;
            FailTransaction(cause, FALSE);
            SynchronizeContext();
            // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
            from_32 = UsingAArch32();
            if from_32 then AArch64.MaybeZeroRegisterUppers();
            if from_32 && HaveSME() && PSTATE.SM == '1' then
ResetSVEState();
else
    MaybeZeroSVEUppers(target_el);

AArch64.ReportException(exception, target_el);

PSTATE.EL = target_el;
PSTATE.mRW = '0';
PSTATE.SP = '1';

SPSR[] = bits(64) UNKNOWN;
ELR[] = bits(64) UNKNOWN;

PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
PSTATE.IL = '0';
if from_32 then                             // Coming from AArch32
    PSTATE.IT = '00000000';
    PSTATE.T = '0';                         // PSTATE.J is RES0
if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
     SCCLR[].SPAN == '0') then
    PSTATE.PAN = '1';
if HaveUAOExt() then PSTATE.UAO = '0';
if HaveBTIExt() then PSTATE.BTYPE = '00';
if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
if HaveMTEExt() then PSTATE.TCO = '1';

DLR_EL0 = bits(64) UNKNOWN;
DSPSR_EL0 = bits(64) UNKNOWN;
EDSCR.ERR = '1';
UpdateEDSCRFields();                        // Update ESCR processor state flags.
if sync_errors then
    SynchronizeErrors();
EndOfInstruction();

// AArch64.WatchpointByteMatch()

integer top = AArch64.VAMax();
bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3;            // Word or doubleword
byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
mask = UInt(DBGWCR_EL1[n].MASK);

if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
    byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBASES);
else
    LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1));
    MSB = (DBGWCR_EL1[n].BAS + LSB);
    if IsZero(MSB AND (MSB - 1)) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
        bottom = 3;                          // For the whole doubleword

if mask > 0 && mask <= 2 then
    Constraint c;
    (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
    assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};

aarch64/debug/watchpoint/AArch64.WatchpointByteMatch

// AArch64.WatchpointByteMatch(integer n, AccType acctype, bits(64) vaddress)

case c of
  when Constraint.Disabled return FALSE; // Disabled
  when Constraint.None mask = 0; // No masking
  // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

boolean WVR_match;
if mask > bottom then
  // If the DBGxVR<->EL1.RES field bits are not a sign extension of the MSB
  // of DBGBVR<->EL1.VA, it is UNPREDICTABLE whether they appear to be
  // included in the match.
  if !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) then
    if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
      top = 63;
      WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
    // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
    if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then
      WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
  else
    WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;
  return WVR_match && byte_select_match;

aarch64/debug/watchpoint/AArch64.WatchpointMatch

// AArch64.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size, boolean ispriv,
  AccType acctype, boolean iswrite)
assert !ELUsingArch32(S1TranslationRegime());
assert n < NumWatchpointsImplemented();

// "ispriv" is:
//  * FALSE for all loads, stores, and atomic operations executed at EL0.
//  * TRUE if the access is unprivileged.
//  * TRUE for all other loads, stores, and atomic operations.

enabled = DBGWCR_EL1[n].E == '1';
linked = DBGWCR_EL1[n].WT == '1';
isbreakpnt = FALSE;

ssce = if HaveRME() then DBGWCR_EL1[n].SSCE else '0';
state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, ssce, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
  linked, DBGWCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);

ls_match = FALSE;
if IsAtomicRW(acctype) then
  ls_match = (DBGWCR_EL1[n].LSC != '00');
elseif iswrite then
  ls_match = (DBGWCR_EL1[n].LSC<1> != '0');
else
  ls_match = (DBGWCR_EL1[n].LSC<0> != '0');

value_match = FALSE;
for byte = 0 to size - 1
  value_match = value_match || AArch64.WatchpointByteMatch(n, acctype, vaddress + byte);
return value_match && state_match && ls_match && enabled;

J1.1.2  aarch64/exceptions

This section includes the following pseudocode functions:
  * aarch64/exceptions/aborts/AArch64.Abort on page J1-11052.
  * aarch64/exceptions/aborts/AArch64.AbortSyndrome on page J1-11052.
aarch64/exceptions/aborts/AArch64.CheckPCAlignment on page J1-11052.
aarch64/exceptions/aborts/AArch64.DataAbort on page J1-11053.
aarch64/exceptions/aborts/AArch64.EffectiveTCF on page J1-11053.
aarch64/exceptions/aborts/AArch64.InstructionAbort on page J1-11054.
aarch64/exceptions/aborts/AArch64.PCAlignmentFault on page J1-11054.
aarch64/exceptions/aborts/AArch64.RaiseTagCheckFault on page J1-11054.
aarch64/exceptions/aborts/AArch64.ReportTagCheckFault on page J1-11055.
aarch64/exceptions/aborts/AArch64.TagCheckFault on page J1-11056.
aarch64/exceptions/aborts/BranchTargetException on page J1-11056.
aarch64/exceptions/aborts/TakeGPECException on page J1-11056.
aarch64/exceptions/async/AArch64.TakePhysicalFIQException on page J1-11057.
aarch64/exceptions/async/AArch64.TakePhysicalIRQException on page J1-11058.
aarch64/exceptions/async/AArch64.TakePhysicalSErrorException on page J1-11058.
aarch64/exceptions/async/AAArch64.TakeVirtualFIQException on page J1-11059.
aarch64/exceptions/async/AArch64.TakeVirtualIRQException on page J1-11059.
aarch64/exceptions/async/AArch64.TakeVirtualSErrorException on page J1-11059.
aarch64/exceptions/debug/AAArch64.BreakpointException on page J1-11059.
aarch64/exceptions/debug/AArch64.SoftwareBreakpoint on page J1-11060.
aarch64/exceptions/debug/AArch64.SoftwareStepException on page J1-11060.
aarch64/exceptions/debug/AArch64.VectorCatchException on page J1-11061.
aarch64/exceptions/debug/AArch64.WatchpointException on page J1-11061.
aarch64/exceptions/exceptions/AArch64.ExceptionClass on page J1-11061.
aarch64/exceptions/exceptions/AArch64.ReportException on page J1-11062.
aarch64/exceptions/exceptions/AArch64.ResetControlRegisters on page J1-11063.
aarch64/exceptions/exceptions/AArch64.TakeReset on page J1-11063.
aarch64/exceptions/ieeefp/AArch64.FPTrappedException on page J1-11064.
aarch64/exceptions/syscalls/AArch64.CallHypervisor on page J1-11065.
aarch64/exceptions/syscalls/AArch64.CallSecureMonitor on page J1-11065.
aarch64/exceptions/syscalls/AArch64.CallSupervisor on page J1-11065.
aarch64/exceptions/takeexception/AArch64.TakeException on page J1-11066.
aarch64/exceptions/traps/AAArch64.AArch32SystemAccessTrap on page J1-11067.
aarch64/exceptions/traps/AAArch64.AArch32SystemAccessTrapSyndrome on page J1-11068.
aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrapSyndrome on page J1-11068.
aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap on page J1-11069.
aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps on page J1-11069.
aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled on page J1-11070.
aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled64 on page J1-11070.
aarch64/exceptions/traps/AArch64.CheckFPEnabled on page J1-11070.
aarch64/exceptions/traps/AArch64.CheckFPEnabled64 on page J1-11070.
aarch64/exceptions/traps/AArch64.CheckForERetTrap on page J1-11071.
aarch64/exceptions/traps/AArch64.CheckForSMCUndefOrTrap on page J1-11071.
aarch64/exceptions/traps/AArch64.CheckForSVCTrap on page J1-11072.
aarch64/exceptions/traps/AArch64.CheckIlleagalState on page J1-11072.
aarch64/exceptions/traps/AArch64.CheckMonitorModeTrap on page J1-11073.
aarch64/exceptions/traps/AArch64.SystemAccessTrap on page J1-11073.
aarch64/exceptions/traps/AArch64.SystemAccessTrapSyndrome on page J1-11073.
aarch64/exceptions/traps/AArch64.UndefiinedFault on page J1-11074.
aarch64/exceptions/traps/AArch64.WFxTrap on page J1-11074.
aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64 on page J1-11075.
aarch64/exceptions/traps/CheckFPEnabled64 on page J1-11075.
• `aarch64/exceptions/traps/CheckLDST64BEnabled` on page J1-11075.
• `aarch64/exceptions/traps/CheckST64BV0Enabled` on page J1-11076.
• `aarch64/exceptions/traps/CheckST64BVEnabled` on page J1-11076.
• `aarch64/exceptions/traps/LDST64BTrap` on page J1-11077.
• `aarch64/exceptions/traps/WFETrapDelay` on page J1-11077.
• `aarch64/exceptions/traps/WaitForEventUntilDelay` on page J1-11077.

### aarch64/exceptions/aborts/AArch64.Abort

```c
// AArch64.Abort()
// ===============
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(bits(64) vaddress, FaultRecord fault)
if IsDebugException(fault) then
  if fault.acctype == AccType_IFETCH then
    if UsingAAArch32() && fault.debugmoe == DebugException_VectorCatch then
      AArch64.VectorCatchException(fault);
    else
      AArch64.BreakpointException(fault);
  else
    AArch64.WatchpointException(vaddress, fault);
  elsif fault.gpcf.gpf != GPCF_None && ReportAsGPCException(fault) then
    TakeGPCException(vaddress, fault);
  elsif fault.acctype == AccType_IFETCH then
    AArch64.InstructionAbort(vaddress, fault);
  else
    AArch64.DataAbort(vaddress, fault);
```

### aarch64/exceptions/aborts/AArch64.AbortSyndrome

```c
// AArch64.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(64) vaddress)
exception = ExceptionSyndrome(exceptype);
d_side = exceptype IN {Exception_DataAbort, Exception_NV2DataAbort, Exception_Watchpoint, Exception_NV2Watchpoint};
(exception.syndrome, exception.syndrome2) = AArch64.FaultSyndrome(d_side, fault);
exception.vaddress = ZeroExtend(vaddress, 64);
if IPAValid(fault) then
  exception.ipavalid = TRUE;
  exception.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
  exception.ipaddress = fault.ipaddress.address;
else
  exception.ipavalid = FALSE;
return exception;
```

### aarch64/exceptions/aborts/AArch64.CheckPCAlignment

```c
// AArch64.CheckPCAlignment()
// ==========================

AArch64.CheckPCAlignment()

bits(64) pc = ThisInstrAddr(64);
```
if pc[1:0] != '00' then
    AArch64.PCAlignmentFault();

aarch64/exceptions/aborts/AArch64.DataAbort

// AArch64.DataAbort()
// ===================
AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)
/\route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
\route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
\                  (HCR_EL2.TGE == '1' ||
\                      (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
\                      (HaveRME() && fault.gpcf.gpf == GPCF_Fail && HCR_EL2.GPF == '1') ||
\                      (HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER) ||
\                      IsSecondStage(fault));

bits(64) preferred_exception_return = ThisInstrAddr(64);
integer vect_offset;
if (HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3) &&
    IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
    vect_offset = 0x180;
else
    vect_offset = 0x0;

ExceptionRecord exception;
if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
    exception = AArch64.AbortSyndrome(Exception_NV2DataAbort, fault, vaddress);
else
    exception = AArch64.AbortSyndrome(Exception_DataAbort, fault, vaddress);

bits(2) target_el = EL1;
if PSTATE.EL == EL3 || route_to_el3 then
    target_el = EL3;
elsif PSTATE.EL == EL2 || route_to_el2 then
    target_el = EL2;
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/aborts/AArch64.EffectiveTCF

// AArch64.EffectiveTCF()
// ======================
// Returns the TCF field applied to tag check faults in the given Exception level.

bits(2) AArch64.EffectiveTCF(AccType acctype)
bits(2) tcf, el;
el = S1TranslationRegime();
if el == EL3 then
tcf = SCTL_EL3.TCF;
elsif el == EL2 then
    if AArch64.AccessUsesEL(acctype) == EL0 then
        tcf = SCTL_EL2.TCF0;
    else
        tcf = SCTL_EL2.TCF;
elsif el == EL1 then
    if AArch64.AccessUsesEL(acctype) == EL0 then
        tcf = SCTL_EL1.TCF0;
    else
        tcf = SCTL_EL1.TCF;
if tcf == '11' then //reserved value
    if !HaveMTE3Ext() then
        (tcf, -) = ConstrantUnpredictableBits(Unpredictable_RESTCF, 2);
return tcf;

aarch64/exceptions/aborts/AArch64.InstructionAbort

// AArch64.InstructionAbort()
// =================================

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)
// External aborts on instruction fetch must be taken synchronously
if HaveDoubleFaultExt() then assert fault.statuscode != Fault_AsyncExternal;
route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
(HCR_EL2.TGE == '1' ||
(HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
(HaveRME() && fault.gpcf.gpf == GPCF_Fail && HCR_EL2.GPF == '1') ||
IsSecondStage(fault));

ExceptionRecord exception;
bits(64) preferred_exception_return = ThisInstrAddr(64);
integer vect_offset;
if (HaveDoubleFaultExt() && PSTATE.EL == EL3 || route_to_el3) &&
  IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
  vect_offset = 0x180;
else
  vect_offset = 0x0;
exception = AArch64.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);

bits(2) target_el = EL1;
if PSTATE.EL == EL3 || route_to_el3 then
  target_el = EL3;
elsif PSTATE.EL == EL2 || route_to_el2 then
  target_el = EL2;
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/aborts/AArch64.PCAlignmentFault

// AArch64.PCAlignmentFault()
// =================================
// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault(fault)
bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_PCAlignment);
exception.vaddress = ThisInstrAddr(64);
bits(2) target_el = EL1;
if UInt(PSTATE.EL) > UInt(EL1) then
  target_el = PSTATE.EL;
elsif EL2Enabled() && HCR_EL2.TGE == '1' then
  target_el = EL2;
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/aborts/AArch64.RaiseTagCheckFault

// AArch64.RaiseTagCheckFault()
// =================================
// Raise a tag check fault exception.

AArch64.RaiseTagCheckFault(bits(64) va, boolean write)

bits(64) preferred_exception_return = ThisInstrAddr(64);
integer vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_DataAbort);
exception.syndrome<5:0> = '010001';
if write then
    exception.syndrome<6> = '1';

exception.vaddress = bits(4) UNKNOWN : va<59:0>;

bits(2) target_el = EL1;
if UInt(PSTATE.EL) > UInt(EL1) then
    target_el = PSTATE.EL;
elif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
    target_el = EL2;
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/aborts/AArch64.ReportTagCheckFault

// AArch64.ReportTagCheckFault()
// =============================
// Records a tag check fault exception into the appropriate TFSR_ELx.
AArch64.ReportTagCheckFault(bits(2) el, bit ttbr)
    if el == EL3 then
        assert ttbr == '0';
        TFSR_EL3.TF0 = '1';
elif el == EL2 then
    if ttbr == '0' then
        TFSR_EL2.TF0 = '1';
else
    TFSR_EL2.TF1 = '1';
elif el == EL1 then
    if ttbr == '0' then
        TFSR_EL1.TF0 = '1';
else
    TFSR_EL1.TF1 = '1';
elif el == EL0 then
    if ttbr == '0' then
        TFSRE0_EL1.TF0 = '1';
else
    TFSRE0_EL1.TF1 = '1';

aarch64/exceptions/aborts/AArch64.SPAlignmentFault

// AArch64.SPAlignmentFault()
// ==========================
// Called on an unaligned stack pointer in AArch64 state.
AArch64.SPAlignmentFault()

bits(64) preferred_exception_return = ThisInstrAddr(64);
vector_offset = 0x0;

exception = ExceptionSyndrome(Exception_SPAlignment);

bits(2) target_el = EL1;
if UInt(PSTATE.EL) > UInt(EL1) then
    target_el = PSTATE.EL;
elif EL2Enabled() && HCR_EL2.TGE == '1' then
    target_el = EL2;
AArch64.TakeException(target_el, exception, preferred_exception_return, vector_offset);
aarch64/exceptions/aborts/AArch64.TagCheckFault

// AArch64.TagCheckFault()
// =======================
// Handle a tag check fault condition.

AArch64.TagCheckFault(bits(64) vaddress, AccType acctype, boolean iswrite)
  bits(2) tcf, el;
  el = AArch64.AccessUsesEL(acctype);
  tcf = AArch64.EffectiveTCF(acctype);
  case tcf of
    when '00'       // Tag Check Faults have no effect on the PE
      return;
    when '01'       // Tag Check Faults cause a synchronous exception
      AArch64.RaiseTagCheckFault(vaddress, iswrite);
    when '10'       // Tag Check Faults are asynchronously accumulated
      AArch64.ReportTagCheckFault(el, vaddress<55>);
    when '11'       // Tag Check Faults cause a synchronous exception on reads or on
        // a read/write access, and are asynchronously accumulated on writes
      // Check for access performing both a read and a write.
      if !iswrite || IsAtomicRW(acctype) then
        AArch64.RaiseTagCheckFault(vaddress, iswrite);
      else
        AArch64.ReportTagCheckFault(el, vaddress<55>);
  end;

aarch64/exceptions/aborts/BranchTargetException

// BranchTargetException()
// =======================
// Raise branch target exception.

AArch64.BranchTargetException(bits(52) vaddress)
  bits(64) preferred_exception_return = ThisInstrAddr(64);
  vect_offset = 0x0;
  exception = ExceptionSyndrome(Exception_BranchTarget);
  exception.syndrome<1:0>   = PSTATE.BTYPE;
  exception.syndrome<24:2>  = Zeros(23);         // RES0
  bits(2) target_el = EL1;
  if UInt(PSTATE.EL) > UInt(EL1) then
    target_el = PSTATE.EL;
  elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
    target_el = EL2;
  AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/aborts/TakeGPCException

// TakeGPCException()
// ===================
// Report Granule Protection Exception faults

TakeGPCException(bits(64) vaddress, FaultRecord fault)
  assert HaveRME();
  assert HavePrivATExt();
  assert HaveAtomicExt();
  assert HaveAccessFlagUpdateExt();
  assert HaveDirtyBitModifierExt();
  assert HaveDoubleFaultExt();
  ExceptionRecord exception;
  exception.exceptype = Exception_GPC;
exception.vaddress  = ZeroExtend(vaddress, 64);
exception.paddress  = fault.paddress;

if IPAValid(fault) then
  exception.ipavalid  = TRUE;
  exception.NS        = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
  exception.ipaddress = fault.ipaddress.address;
else
  exception.ipavalid = FALSE;

// Populate the fields grouped in ISS
exception.syndrome<24:22> = Zeros(3); // RES0
exception.syndrome<21>    = if fault.gpcfs2walk then '1' else '0';  // S2PTW
exception.syndrome<20>    = if fault.acctype == AccType_IFETCH then '1' else '0'; // InD
exception.syndrome<19:14> = EncodeGPCSC(fault.gpcf); // GPCSC
if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
  exception.syndrome<13> = '1'; // VNCR
else
  exception.syndrome<13> = '0'; // VNCR
exception.syndrome<12:11> = '00'; // RES0
exception.syndrome<10:9>  = '00'; // RES0

if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT, AccType_ATPAN} then
  exception.syndrome<8> = '1'; // CM
else
  exception.syndrome<8> = '0'; // CM

exception.syndrome<7> = if fault.s2fs1walk then '1' else '0'; // S1PTW

if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT, AccType_ATPAN} then
  exception.syndrome<6> = '1'; // WnR
elsif fault.statuscode IN {Fault_HWUpdateAccessFlag, Fault_Exclusive} then
  exception.syndrome<6> = bit UNKNOWN; // WnR
elsif IsAtomicRW(fault.acctype) && IsExternalAbort(fault) then
  exception.syndrome<6> = bit UNKNOWN; // WnR
else
  exception.syndrome<6> = if fault.write then '1' else '0'; // WnR

exception.syndrome<5:0> = EncodeLDFSC(fault.statuscode, fault.level); // xFSC

bits(64) preferred_exception_return = ThisInstrAddr(64);

integer vect_offset;
if IsExternalAbort(fault) && SCR_EL3.EASE == '1' then
  vect_offset = 0x180;
else
  vect_offset = 0x0;

AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/async/AArch64.TakePhysicalFIQException

// AArch64.TakePhysicalFIQException()
// -------------------------------------
AArch64.TakePhysicalFIQException()

route_to_el3 = HaveEL(EL3) && SCR_EL3.FIQ == '1';
route_to_el2 = (PSTATE.EL IN [EL0, EL1]) && EL2Enabled() &&
  (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x100;
exception = ExceptionSyndrome(Exception_FIQ);

if route_to_el3 then
  AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
elsif PSTATE.EL == EL2 || route_to_el2 then
    assert PSTATE.EL != EL3;
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    assert PSTATE.EL IN {EL0, EL1};
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/async/AArch64.TakePhysicalIRQException

// AArch64.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.
AArch64.TakePhysicalIRQException()

    route_to_el3 = HaveEL(EL3) & SCR_EL3.IRQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} & EL2Enabled() &
                   (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/async/AArch64.TakePhysicalSErrorException

// AArch64.TakePhysicalSErrorException()
// =====================================
AArch64.TakePhysicalSErrorException(bits(25) syndrome)

    route_to_el3 = HaveEL(EL3) & SCR_EL3.EA == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} & EL2Enabled() &
                   (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1')));
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x180;

    bits(2) target_el;
    if PSTATE.EL == EL3 || route_to_el3 then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_el2 then
        target_el = EL2;
    else
        target_el = EL1;

    if IsSErrorEdgeTriggered(target_el, syndrome) then
        ClearPendingPhysicalSError();
    exception = ExceptionSyndrome(Exception_SError);
    exception.syndrome = syndrome;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
`aarch64/exceptions/async/AArch64.TakeVirtualFIQException`

```pseudocode```
// AArch64.TakeVirtualFIQException()
// ==============================
AArch64.TakeVirtualFIQException()
assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';  // Virtual IRQ enabled if TGE==0 and FMO==1
bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x100;
exception = ExceptionSyndrome(Exception_FIQ);
AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

`aarch64/exceptions/async/AArch64.TakeVirtualIRQException`

```pseudocode```
// AArch64.TakeVirtualIRQException()
// ==============================
AArch64.TakeVirtualIRQException()
assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';  // Virtual IRQ enabled if TGE==0 and IMO==1
bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x80;
exception = ExceptionSyndrome(Exception_IRQ);
AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

`aarch64/exceptions/async/AArch64.TakeVirtualSErrorException`

```pseudocode```
// AArch64.TakeVirtualSErrorException()
// ====================================
AArch64.TakeVirtualSErrorException(bits(25) syndrome)
assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';  // Virtual SError enabled if TGE==0 and AMO==1
bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x180;
exception = ExceptionSyndrome(ExceptionSError);
if HaveRASExt() then
    exception.syndrome<24> = VSESR_EL2.IDS;
    exception.syndrome<23:0> = VSESR_EL2.ISS;
else
    impdef_syndrome = syndrome<24> == '1';
    if impdef_syndrome then exception.syndrome = syndrome;
ClearPendingVirtualSError();
AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

`aarch64/exceptions/debug/AArch64.BreakpointException`

```pseudocode```
// AArch64.BreakpointException()
// =============================
AArch64.BreakpointException(FaultRecord fault)
assert PSTATE.EL != EL3;
```

---
route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
       (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x0;

vaddress = bits(64) UNKNOWN;
exception = AArch64.AbtSyndrome(Exception_Breakpoint, fault, vaddress);
if PSTATE.EL == EL2 || route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/debug/AArch64.SoftwareBreakpoint

// AArch64.SoftwareBreakpoint()
// ============================
AArch64.SoftwareBreakpoint(bits(16) immediate)

route_to_el2 = (PSTATE.EL IN {EL0, EL1} &&
       EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_SoftwareBreakpoint);
exception.syndrome<15:0> = immediate;
if UInt(PSTATE.EL) > UInt(EL1) then
  AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elsif route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/debug/AArch64.SoftwareStepException

// AArch64.SoftwareStepException()
// ===============================
AArch64.SoftwareStepException()

assert PSTATE.EL != EL3;

route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
       (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_SoftwareStep);
if SoftwareStep_DidNotStep() then
  exception.syndrome<24> = '0';
else
  exception.syndrome<24> = '1';
  exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';
  exception.syndrome<5:0> = '100010';                // IFSC = Debug Exception
if PSTATE.EL == EL2 || route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/debug/AArch64.VectorCatchException

// AArch64.VectorCatchException()
// --------------------------------------------
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.
AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
    assert EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddress);
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/debug/AArch64.WatchpointException

// AArch64.WatchpointException()
// --------------------------------------------
AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                   (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    ExceptionRecord exception;
    if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
        exception = AArch64.AbortSyndrome(Exception_NV2Watchpoint, fault, vaddress);
    else
        exception = AArch64.AbortSyndrome(Exception_Watchpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/exceptions/AArch64.ExceptionClass

// AArch64.ExceptionClass()
// --------------------------
// Returns the Exception Class and Instruction Length fields to be reported in ESR
(integer, bit) AArch64.ExceptionClass(Exception exceptype, bits(2) target_el)

    il_is_valid = TRUE;
    from_32 = UsingAArch32();

    integer ec;
    case exceptype of
        when Exception_Uncategorized        ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap              ec = 0x01;
        when Exception_CP15RTTrap           ec = 0x03; assert from_32;
        when Exception_CP15RRTTrap          ec = 0x04; assert from_32;
when Exception_CP14RTTrap           ec = 0x05; assert from_32;
when Exception_CP14DTTrap           ec = 0x06; assert from_32;
when Exception_AdvSIMDAccessTrap   ec = 0x07;
when Exception_FPIDTrap            ec = 0x08;
when Exception_PACTrap             ec = 0x09;
when Exception_LDS64BTrap          ec = 0x0A;
when Exception_TSTARTAccessTrap    ec = 0x1B;
when Exception_GPC                 ec = 0x1E;
when Exception_CP14RRTTrap          ec = 0x0C; assert from_32;
when Exception_BranchTarget        ec = 0x0D;
when Exception_IllegalState        ec = 0x0E; il_is_valid = FALSE;
when Exception_SupervisorCall      ec = 0x11;
when Exception_HypervisorCall      ec = 0x12;
when Exception_MonitorCall         ec = 0x13;
when Exception_SystemRegisterTrap  ec = 0x18; assert !from_32;
when Exception_SVEAccessTrap       ec = 0x19; assert !from_32;
when Exception_ERetTrap            ec = 0x1A; assert !from_32;
when Exception_PACFail             ec = 0x1C; assert !from_32;
when Exception_InstructionAbort    ec = 0x2A; il_is_valid = FALSE;
when Exception_PCAlignment         ec = 0x22; il_is_valid = FALSE;
when Exception_DataAbort           ec = 0x24;
when Exception_NV2DataAbort        ec = 0x25;
when Exception_SPAlignment         ec = 0x26; il_is_valid = FALSE; assert !from_32;
when Exception_MemCpyMemSet        ec = 0x27;
when Exception_FPTrappedException  ec = 0x28;
when Exception_SMEAccessTrap       ec = 0x1D; assert !from_32;
when Exception_ERetTrap            ec = 0x2F; il_is_valid = FALSE;
when Exception_Breakpoint          ec = 0x30; il_is_valid = FALSE;
when Exception_SoftwareStep        ec = 0x32; il_is_valid = FALSE;
when Exception_Watchpoint          ec = 0x34; il_is_valid = FALSE;
when Exception_NV2Watchpoint       ec = 0x35; il_is_valid = FALSE;
when Exception_SoftwareBreakpoint  ec = 0x36;
when Exception_VectorCatch         ec = 0x3A; il_is_valid = FALSE; assert from_32;
otherwise                          Unreachable();

if ec IN {0x20,0x24,0x28,0x25,0x26,0x30,0x32,0x34} && target_el == PSTATE.EL then
  ec = ec + 1;

if ec IN {0x11,0x12,0x13,0x18,0x19} && !from_32 then
  ec = ec + 4;

bit il;
if il_is_valid then
  il = if ThisInstrLength() == 32 then '1' else '0';
else
  il = '1';
assert from_32 || il == '1'; // AArch64 instructions always 32-bit

return (ec,il);

aarch64/exceptions/exceptions/AArch64.ReportException

  // AArch64_ReportException()
  // ==================================
  // Report syndrome information for exception taken to AArch64 state.

AArch64_ReportException(ExceptionRecord exception, bits(2) target_el)

    Exception exception_type = exception.exceptype;
    (ec,il) = AArch64_ExceptionClass(exception_type, target_el);
    iss = exception.syndrome;
    iss2 = exception.syndrome2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
      il = '1';

    return (ec,il);
ESR[target_el] = (Zeros(27) : // <63:37> 
iss2 : // <36:32> 
ec<5:0> : // <31:26> 
il : // <25> 
iss); // <24:0>

if exceptype IN {
  Exception_InstructionAbort,
  Exception_PCAlignment,
  Exception_DataAbort,
  Exception_NV2DataAbort,
  Exception_NV2Watchpoint,
  Exception_GPC,
  Exception_Watchpoint
} then
  FAR[target_el] = exception.vaddress;
else
  FAR[target_el] = bits(64) UNKNOWN;

if exception.ipavalid then
  HPFAR_EL2<43:4> = exception.ipaddress<51:12>;
  if IsSecureEL2Enabled() & CurrentSecurityState() == SS_Secure then
    HPFAR_EL2.NS = exception.NS;
  else
    HPFAR_EL2.NS = '0';
elsif target_el == EL2 then
  HPFAR_EL2<43:4> = bits(40) UNKNOWN;

if exception.exceptype == Exception_GPC then
  MFAR_EL3.FPA = ZeroExtend(exception.paddress.address<AArch64.PAMax()-1:12>, 40);
  case exception.paddress.paspace of
    when PAS_Secure     MFAR_EL3.<NSE,NS> = '00';
    when PAS_NonSecure  MFAR_EL3.<NSE,NS> = '01';
    when PAS_Root       MFAR_EL3.<NSE,NS> = '10';
    when PAS_Realm      MFAR_EL3.<NSE,NS> = '11';
  return;

aarch64/exceptions/exceptions/AArch64.ResetControlRegisters

// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch64.ResetControlRegisters(boolean cold_reset);

aarch64/exceptions/exceptions/AArch64.TakeReset

// AArch64.TakeReset()
// ===============
// Reset into AArch64 state
AArch64.TakeReset(boolean cold_reset)
  assert HaveAArch64();

  // Enter the highest implemented Exception level in AArch64 state
  PSTATE.nRW = '0';
  if HaveEL(EL3) then
    PSTATE.EL = EL3;
  elsif HaveEL(EL2) then
    PSTATE.EL = EL2;
  else
    PSTATE.EL = EL1;

  // Reset System registers and other system components
  AArch64.ResetControlRegisters(cold_reset);
// Reset all other PSTATE fields
PSTATE.SP = '1';              // Select stack pointer
PSTATE.<D,A,I,F>  = '1111';   // All asynchronous exceptions masked
PSTATE.SS = '0';              // Clear software step bit
PSTATE.DIT = '0';             // PSTATE.DIT is reset to 0 when resetting into AArch64
PSTATE.IL = '0';              // Clear Illegal Execution state bit

TSTATE.depth = 0;             // Non-transactional state

// All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
// below are UNKNOWN bitstrings after reset. In particular, the return information registers
// ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
// is impossible to return from a reset in an architecturally defined way.
AArch64.ResetGeneralRegisters();
AArch64.ResetSIMDFPRegisters();
AArch64.ResetSpecialRegisters();
ResetExternalDebugRegisters(cold_reset);

bits(64) rv;                      // IMPLEMENTATION DEFINED reset vector
if HaveEL(EL3) then
  rv = RVBAR_EL3;
elif HaveEL(EL2) then
  rv = RVBAR_EL2;
else
  rv = RVBAR_EL1;

// The reset vector must be correctly aligned
assert IsZero(rv<63:AArch64.PAMax()> && IsZero(rv<1:0>);

boolean branch_conditional = FALSE;
BranchTo(rv, BranchType_RESET, branch_conditional);

aarch64/exceptions/ieeefp/AArch64.FPTrappedException

// AArch64.FPTrappedException()
// =============================================
AArch64.FPTrappedException(boolean is_ase, bits(8) accumulated_exceptions)
    exception = ExceptionSyndrome(Exception_FPTrappedException);
    if is_ase then
        if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
            exception.syndrome<23> = '1';                          // TFV
        else
            exception.syndrome<23> = '0';                          // TFV
        end if
    else
        exception.syndrome<23> = '1';                              // TFV
    end if
    exception.syndrome<10:8> = bits(3) UNKNOWN;                    // VECITR
    if exception.syndrome<23> == '1' then
        exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
    else
        exception.syndrome<7,4:0> = bits(6) UNKNOWN;
    end if

    route_to_el2 = EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
aarch64/exceptions/syscalls/AArch64.CallHypervisor

// AArch64.CallHypervisor()
// ========================
// Performs a HVC call

AArch64.CallHypervisor(bits(16) immediate)
assert HaveEL(EL2);

if UsingAArch32() then AArch32.ITAdvance();
SSAdvance();
bits(64) preferred_exception_return = NextInstrAddr(64);
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_HypervisorCall);
exception.syndrome<15:0> = immediate;

if PSTATE.EL == EL3 then
  AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/syscalls/AArch64.CallSecureMonitor

// AArch64.CallSecureMonitor()
// ===========================

AArch64.CallSecureMonitor(bits(16) immediate)
assert HaveEL(EL3) && !ELUsingAArch32(EL3);
if UsingAArch32() then AArch32.ITAdvance();
SSAdvance();
bits(64) preferred_exception_return = NextInstrAddr(64);
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_MonitorCall);
exception.syndrome<15:0> = immediate;

AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/syscalls/AArch64.CallSupervisor

// AArch64.CallSupervisor()
// ========================
// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate_in)
bits(16) immediate = immediate_in;
if UsingAArch32() then AArch32.ITAdvance();
SSAdvance();
route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

bits(64) preferred_exception_return = NextInstrAddr(64);
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_SupervisorCall);
exception.syndrome<15:0> = immediate;

if UInt(PSTATE.EL) > UInt(EL1) then
  AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elsif route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
AArch64.TakeException(bits(2) target_el, ExceptionRecord exception_in,
bits(64) preferred_exception_return, integer vect_offset_in)
assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
ExceptionRecord exception = exception_in;
boolean sync_errors;
boolean iesb_req;
if HaveIESB() then
    sync_errors = SCTLR[target_el].IESB == '1';
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
    if sync_errors && InsertIESBBeforeException(target_el) then
        SynchronizeErrors();
        iesb_req = FALSE;
        sync_errors = FALSE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
else
    sync_errors = FALSE;

if HaveTME() && TSTATE.depth > 0 then
    TMFailure cause;
case exception.exceptype of
    when Exception_SoftwareBreakpoint cause = TMFailure_DBG;
    when Exception_Breakpoint         cause = TMFailure_DBG;
    when Exception_Watchpoint         cause = TMFailure_DBG;
    when Exception_SoftwareStep       cause = TMFailure_DBG;
other                   cause = TMFailure_ERR;
Endcase ();
FailTransaction(cause, FALSE);
SynchronizeContext();

// If coming from AArch32 state, the top parts of the X[] registers might be set to zero
from_32 = UsingAArch32();
if from_32 then AArch64.MaybeZeroRegisterUppers();
if from_32 && HaveSME() && PSTATE.SM == '1' then
    ResetSVEState();
else
    MaybeZeroSVEUppers(target_el);

integer vect_offset = vect_offset_in;
if UInt(target_el) > UInt(PSTATE.EL) then
    boolean lower_32;
    if target_el == EL3 then
        if EL2Enabled() then
            lower_32 = ELUsingAArch32(EL2);
        else
            lower_32 = ELUsingAArch32(EL1);
    else
        lower_32 = ELUsingAArch32(target_el - 1);
    vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);
elsif PSTATE.SP == '1' then
    vect_offset = vect_offset + 0x200;

bits(64) spsr = GetPSRFromPSTATE(AArch64_NonDebugState, 64);

if PSTATE.EL == EL1 && target_el == EL1 && EL2Enabled() then
    if HaveNV2Ext() && (HCR_EL2.<NV,NV1,NV2> == '100' || HCR_EL2.<NV,NV1,NV2> == '111') then
        spsr<3:2> = '10';
    else
        if HaveNVExt() && HCR_EL2.<NV,NV1> == '10' then
            spsr<3:2> = '10';
spsr<3:2> = '10';
if HaveBTIExt() && !UsingAArch32() then
    boolean zero_btype;
    // SPSR[].BTYPE is only guaranteed valid for these exception types
    if exception.exceptype IN {Exception_SError, Exception_IRQ, Exception_FIQ,
        Exception_SoftwareStep, Exception_PCAlignment, Exception_InstructionAbort,
        Exception_Breakpoint, Exception_VectorCatch, Exception_SoftwareBreakpoint,
        Exception_IllegalState, Exception_BranchTarget} then
        zero_btype = FALSE;
    else
        zero_btype = ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE);
    if zero_btype then spsr<11:10> = '00';
if HaveNV2Ext() && exception.exceptype == Exception_NV2DataAbort && target_el == EL3 then
    // External aborts are configured to be taken to EL3
    exception.exceptype = Exception_DataAbort;
if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
    AArch64.ReportException(exception, target_el);
if HaveBRBExt() then
    BRBException(exception.exceptype, preferred_exception_return,
        VBAR[target_el]<63:11>:vect_offset<10:0>, target_el,
        exception.trappedsyscallinst);

PSTATE.EL = target_el;
PSTATE.nRW = '0';
PSTATE.SP = '1';
SPSR[] = spsr;
ELR[] = preferred_exception_return;

PSTATE.SS = '0';
if HaveFeatNMI() && !ELUsingAArch32(target_el) then PSTATE.ALLINT = NOT SCTLR[].SPINTMASK;
PSTATE.<DAIF> = '1111';
PSTATE.IE = '0';
if from_32 then                             // Coming from AArch32
    PSTATE.IT = '00000000';
PSTATE.J = '0';                         // PSTATE.J is RES0
if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
    SCTLR[].SPAN == '0') then
    PSTATE.PAN = '1';
if HaveUAOExt() then then PSTATE.UAO = '0';
if HaveBTIExt() then then PSTATE.BTYPE = '00';
if HaveSSBSExt() then PSTATE.SSBS = SCTLR[].DSSBS;
if HaveMTEExt() then then PSTATE.TCO = '1';

boolean branch_conditional = FALSE;
BranchTo(VBAR<63:11>:vect_offset<10:0>, BranchType_EXCEPTION, branch_conditional);

CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

if sync_errors then                             // Synchronizing
    SynchronizeErrors();
    iesb_req = TRUE;
    TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
EndOfInstruction();

aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrap

    // AArch64.AArch32SystemAccessTrap()
    // -----------------------------------
    // Trapped AARCH32 System register access.
AArch64.AArch32SystemAccessTrap(bits(2) target_el, integer ec)
assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);
bits(64) preferred_exception_return = ThisInstrAddr(64);
vec_offset = 0x0;

exception = AArch64.AArch32SystemAccessTrapSyndrome(ThisInstr(), ec);
AArch64.TakeException(target_el, exception, preferred_exception_return, vec_offset);

aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrapSyndrome

// AArch64.AArch32SystemAccessTrapSyndrome()
// =========================================
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR instructions,
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord exception;
    case ec of
        when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTrap);
        when 0x4    exception = ExceptionSyndrome(Exception_CP14RTTrap);
        when 0x5    exception = ExceptionSyndrome(Exception_CP14DTTrap);
        when 0x6    exception = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros(20);
    if exception.exceptype == Exception_Uncategorized then
        return exception;
    elsif exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap, Exception_CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        if exception.exceptype != Exception_FPIDTrap then    // When trap is not for VMRS
            iss<19:17> = instr<7:5>;           // opc2
            iss<16:14> = instr<23:21>;         // opc1
            iss<13:10> = instr<19:16>;         // CRn
            iss<4:1>   = instr<3:0>;           // CRm
        else
            iss<19:17> = '000';
            iss<16:14> = '111';
            iss<13:10> = instr<19:16>;     // reg
            iss<4:1>   = '0000';
        end;
        if instr<20> == '1' && instr<15:12> == '1111' then    // MRC, Rt==15
            iss<9:5> = '11111';
        elsif instr<20> == '0' && instr<15:12> == '1111' then // MCR, Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        end;
    elsif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss<19:16> = instr<7:4>;           // opc1
        if instr<19:16> == '1111' then    // Rt2==15
            iss<14:10> = bits(5) UNKNOWN;
        else
            iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>;
        end;
        if instr<15:12> == '1111' then    // Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
        end;
    end;
Armv8 Pseudocode

J1.1 Pseudocode for AArch64 operation

iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
iss<4:1> = instr<3:0>; // CRm
elsif exception.exceptype == Exception_CP14DTTrap then
  // Trapped LDC/STC
  iss<19:12> = instr<7:0>; // imm8
  iss<4> = instr<23>; // U
  iss<2:1> = instr<24,21>; // P,W
  if instr<19:16> == '1111' then // Rn==15, LDC(Literal addressing)/STC
    iss<9:5> = bits(5) UNKNOWN;
    iss<3> = '1';
    iss<0> = instr<20>; // Direction
  end
  exception.syndrome<24:20> = ConditionSyndrome();
  exception.syndrome<19:0> = iss;
return exception;

aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap

// AArch64.AdvSIMDFPAccessTrap()
// =============================
// Trapped access to Advanced SIMD or FP registers due to CPACR[].
AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
  bits(64) preferred_exception_return = ThisInstrAddr(64);
  ExceptionRecord exception;
  vect_offset = 0x0;
  route_to_el2 = (target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1');
if route_to_el2 then
  exception = ExceptionSyndrome(Exception_Uncategorized);
AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
  exception.syndrome<24:20> = ConditionSyndrome();
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
return;

aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps

// AArch64.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained AArch32 traps to System registers in the
// coproc=0b1111 encoding space by HSTR_EL2, HCR_EL2, and SCTLR_ELx.
AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
  trapped_encoding = (((CRn == 9  && CRm IN {0,1,2, 5,6,7,8 }) ||
    (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
    (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

  // Check for MRC and MCR disabled by SCTLR_EL1.TIDCP.
  if (HaveFeatTIDCP1() && PSTATE.EL == EL0 && !IsInHost() &&
    !ELUsingAArch32(EL1) && SCTLR_EL1.TIDCP == '1' && trapped_encoding) then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
      AArch64.AArch32SystemAccessTrap(EL2, 0x3);
    else
      AArch64.AArch32SystemAccessTrap(EL1, 0x3);
  end

  // Check for coarse-grained Hyp traps
  if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
    // Check for MRC and MCR disabled by SCTLR_EL2.TIDCP.
    if (HaveFeatTIDCP1() && PSTATE.EL == EL0 && !IsInHost() &&

 ARM DDI 04871.a Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. J1-11069
 ID081822 Non-Confidential
SCTLR_EL2.TIDCP == '1' && trapped_encoding) then
AArch64.AArch32SystemAccessTrap(EL2, 0x3);

major = if nreg == 1 then CRn else CRm;
// Check for MCR, MRC, MCRR, and MRRC disabled by HSTR_EL2<CRn/CRm>
// and MRC and MCR disabled by HCR_EL2.TIDCP.
if (((IsInHost() && !(major IN {4,14}) && HSTR_EL2<major> == '1') ||
    (HCR_EL2.TIDCP == '1' && nreg == 1 && trapped_encoding)) then
  if (PSTATE.EL == EL0 &&
    boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0") then
    UNDEFINED;
  AArch64.AArch32SystemAccessTrap(EL2, 0x3);

aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled

// AArch64.CheckFPAdvSIMDEnabled()
// -----------------------------------
AArch64.CheckFPAdvSIMDEnabled()
AArch64.CheckFPEnabled();
// Check for illegal use of Advanced
// SIMD in Streaming SVE Mode
if HaveSME() && PSTATE.SM == '1' && !IsFullA64Enabled() then
  SMEAccessTrap(SMEExceptionType_Streaming, PSTATE.EL);

aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap

// AArch64.CheckFPAdvSIMDTrap()
// -----------------------------
// Check against CPTR_EL2 and CPTR_EL3.
AArch64.CheckFPAdvSIMDTrap()
  if HaveEL(EL3) && CPTR_EL3.TFP == '1' && EL3DDTrapPriority() then
    UNDEFINED;
  if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
    // Check if access disabled in CPTR_EL2
    if HaveVirtHostExt() && HCR_EL2.EZH == '1' then
      boolean disabled;
      case CPTR_EL2.FPEN of
        when 'x0' disabled = TRUE;
        when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
        when '11' disabled = FALSE;
      if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
      else
        if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

if HaveEL(EL3) then
  // Check if access disabled in CPTR_EL3
  if CPTR_EL3.TFP == '1' then
    if Halted() && ESCR.SDD == '1' then
      UNDEFINED;
    else
      AArch64.AdvSIMDFPAccessTrap(EL3);

aarch64/exceptions/traps/AArch64.CheckFPEnabled

// AArch64.CheckFPEnabled()
// ------------------------
// Check against CPACR[]
AArch64.CheckFPEnabled()
  if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
// Check if access disabled in CPACR_EL1
boolean disabled;
case CPACR_EL1.FPEN of
  when 'x0' disabled = TRUE;
  when '01' disabled = PSTATE.EL == EL0;
  when '11' disabled = FALSE;
if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

AArch64.CheckFPAdvSIMDTrap();  // Also check against CPTR_EL2 and CPTR_EL3

aarch64/exceptions/traps/AArch64.CheckForERetTrap

// AArch64.CheckForERetTrap()
// Check for trap on ERET, ERETAA, ERETAB instruction
AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)

  route_to_el2 = FALSE;
  // Non-secure EL1 execution of ERET, ERETAA, ERETAB when either HCR_EL2.NV or HFGITR_EL2.ERET is set,
  // is trapped to EL2
  route_to_el2 = (PSTATE.EL == EL1 && EL2Enabled() &&
                 ((HaveNVExt() && HCR_EL2.NV == '1') ||
                  (HaveFGTExt() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
                   HFGITR_EL2.ERET == '1')));
if route_to_el2 then
  ExceptionRecord exception;
  bits(64) preferred_exception_return = ThisInstrAddr(64);
  vect_offset = 0x0;
  exception = ExceptionSyndrome(Exception_ERetTrap);
  if eret_with_pac then
    exception.syndrome<1> = '0';
    exception.syndrome<0> = '0';  // RES0
  else
    exception.syndrome<1> = '1';
    if pac_uses_key_a then
      exception.syndrome<0> = '0';
    else
      exception.syndrome<0> = '1';
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/traps/AArch64.CheckForSMCUndefOrTrap

// AArch64.CheckForSMCUndefOrTrap()
// Check for UNDEFINED or trap on SMC instruction
AArch64.CheckForSMCUndefOrTrap(bits(16) imm)
if PSTATE.EL == EL0 then UNDEFINED;
if (!(PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1') &&
    HaveEL(EL3) && SCR_EL3.SMD == '1') then
  route_to_el2 = FALSE;
if !HaveEL(EL3) then
  if PSTATE.EL == EL1 && EL2Enabled() then
    if HaveNVExt() && HCR_EL2.NV == '1' && HCR_EL2.TSC == '1' then
      route_to_el2 = TRUE;
    else
      UNDEFINED;
  else
    UNDEFINED;
  else
    route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
if route_to_el2 then
  bits(64) preferred_exception_return = ThisInstrAddr(64);
  vect_offset = 0x0;
  exception = Exception Syndrome(Exception_MonitorCall);
  exception.syndrome<15:0> = imm;
  exception.trappedsyscallinst = TRUE;
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/traps/AArch64.CheckForSVCTrap

// AArch64.CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction
AArch64.CheckForSVCTrap(bits(16) immediate)
if HaveFGTExt() then
  route_to_el2 = FALSE;
if PSTATE.EL == EL0 then
  route_to_el2 = (!UsingAArch32() && !ELUsingAArch32(EL1) &&
  EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
  (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));
elsif PSTATE.EL == EL1 then
  route_to_el2 = (EL2Enabled() && HFGITR_EL2.SVC_EL1 == '1' &&
  (!HaveEL(EL3) || SCR_EL3.FGTEn == '1'));
if route_to_el2 then
  exception = Exception Syndrome(Exception_SupervisorCall);
  exception.syndrome<15:0> = immediate;
  exception.trappedsyscallinst = TRUE;
  bits(64) preferred_exception_return = ThisInstrAddr(64);
  vect_offset = 0x0;
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/traps/AArch64.CheckForWFxTrap

// AArch64.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction
AArch64.CheckForWFxTrap(bits(2) target_el, WFxType wfxtype)
assert HaveEL(target_el);
boolean is_wfe = wfxtype IN {WFxType_WFE, WFxType_WFET};
boolean trap;
case target_el of
  when EL1
    trap = (if is_wfe then SCTLR[].nTWE else SCTLR[].nTWI) == '0';
  when EL2
    trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
  when EL3
    trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';
if trap then
  AArch64.WFxTrap(wfxtype, target_el);

aarch64/exceptions/traps/AArch64.CheckIllegalState

// AArch64.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.
AArch64.CheckIllegalState()
if PSTATE.IL == '1' then
  route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
  bits(64) preferred_exception_return = ThisInstrAddr(64);
  vect_offset = 0x0;
  exception = ExceptionSyndrome(Exception_IllegalState);
  if UInt(PSTATE.EL) > UInt(EL1) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
  elsif route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
  else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/traps/AArch64.MonitorModeTrap

// AArch64.MonitorModeTrap()
// =========================
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()
bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x0;
exception = ExceptionSyndrome(Exception_Uncategorized);
if IsSecureEL2Enabled() then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
  AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/traps/AArch64.SystemAccessTrap

// AArch64.SystemAccessTrap()
// ==========================
// Trapped access to AArch64 System register or system instruction.

AArch64.SystemAccessTrap(bits(2) target_el, integer ec)
assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);
bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x0;
exception = AArch64.SystemAccessTrapSyndrome(ThisInstr(), ec);
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/traps/AArch64.SystemAccessTrapSyndrome

// AArch64.SystemAccessTrapSyndrome()
// ==================================
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

ExceptionRecord AArch64.SystemAccessTrapSyndrome(bits(32) instr_in, integer ec)
  ExceptionRecord exception;
  bits(32) instr = instr_in;
  case ec of
  when 0x0                                                     // Trapped access due to unknown reason.
    exception = ExceptionSyndrome(Exception_Uncategorized);
  when 0x7                                                     // Trapped access to SVE, Advance SIMD&FP System register.
    exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
exception.syndrome<24:20> = ConditionSyndrome();  // Trapped access to System register or system instruction.
    exception = ExceptionSyndrome(Exception_SystemRegisterTrap);
    instr = ThisInstr();
    exception.syndrome<21:20> = instr<20:19>;  // Op0
    exception.syndrome<19:17> = instr<7:5>;  // Op2
    exception.syndrome<16:14> = instr<18:16>;  // Op1
    exception.syndrome<13:10> = instr<15:12>;  // CRn
    exception.syndrome<9:5> = instr<4:0>;  // Rt
    exception.syndrome<4:1> = instr<11:8>;  // CRm
    exception.syndrome<0> = instr<21>;  // Direction
when 0x19  // Trapped access to SVE System register
    exception = ExceptionSyndrome(Exception_SVEAccessTrap);
when 0x1D  // Trapped access to SME System register
    exception = ExceptionSyndrome(Exception_SMEAccessTrap);
otherwise
Unreachable();
return exception;

aarch64/exceptions/traps/AArch64.UndefinedFault

// AArch64.UndefinedFault()
// ==========================
AArch64.UndefinedFault()

route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x0;
exception = ExceptionSyndrome(Exception_Uncategorized);
if UInt(PSTATE.EL) > UInt(EL1) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elsif route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

aarch64/exceptions/traps/AArch64.WFxTrap

// AArch64.WFxTrap()
// ==================
AArch64.WFxTrap(WFxType wfxtype, bits(2) target_el)
assert UInt(target_el) > UInt(PSTATE.EL);
bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x0;
exception = ExceptionSyndrome(Exception_WFxTrap);
exception.syndrome<24:20> = ConditionSyndrome();
case wfxtype of
    when WFxType_WFI
        exception.syndrome<1:0> = '00';
    when WFxType_WFE
        exception.syndrome<1:0> = '01';
    when WFxType_WFIT
        exception.syndrome<1:0> = '10';
exception.syndrome\[2\] = '1';  // Register field is valid
exception.syndrome\[9\:5\] = \text{ThisInstr()\[4\:0\]};
when WFxTypeWFET
exception.syndrome\[1\:0\] = '11';
exception.syndrome\[2\] = '1';  // Register field is valid
exception.syndrome\[9\:5\] = \text{ThisInstr()\[4\:0\]};

if target\_el == EL1 \&\& EL2Enabled() \&\& HCR\_EL2.TGE == '1' then
AArch64.TakeException(EL2, exception, preferred\_exception\_return, vect\_offset);
else
AArch64.TakeException(target\_el, exception, preferred\_exception\_return, vect\_offset);

\text{aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64}

// CheckFPAdvSIMDEnabled64()
// =========================
// AArch64 instruction wrapper
CheckFPAdvSIMDEnabled64()
AArch64.CheckFPAdvSIMDEnabled();

\text{aarch64/exceptions/traps/CheckFPEnabled64}

// CheckFPEnabled64()
// ===============
// AArch64 instruction wrapper
CheckFPEnabled64()
AArch64.CheckFPEnabled();

\text{aarch64/exceptions/traps/CheckLDST64BEnabled}

// CheckLDST64BEnabled()  
// =====================  
// Checks for trap on ST64B and LD64B instructions
CheckLDST64BEnabled()
boolean trap = FALSE;
bits(25) iss = ZeroExtend('10', 25);  // 0x2
bits(2) target\_el;

if PSTATE.EL == EL0 then
    if !IsInHost() then
        trap = SCTLR\_EL1.EnALS == '0';
        target\_el = if EL2Enabled() \&\& HCR\_EL2.TGE == '1' then EL2 else EL1;
    else
        trap = SCTLR\_EL2.EnALS == '0';
        target\_el = EL2;
    else
        target\_el = EL1;

if trap \&\& EL2Enabled() \&\& HaveFeatHCX() \&\&
((PSTATE.EL == EL0 \&\& !IsInHost()) \|\| PSTATE.EL == EL1) then
    trap = !IsHCRXEL2Enabled() \|\| HCRX\_EL2.EnALS == '0';
    target\_el = EL2;

if trap then LDST64BTrap(target\_el, iss);
aarch64/exceptions/traps/CheckST64BV0Enabled

// CheckST64BV0Enabled()
// =====================
// Checks for trap on ST64BV0 instruction

CheckST64BV0Enabled()
    boolean trap = FALSE;
    bits(25) iss = ZeroExtend('1', 25);  // 0x1
    bits(2) target_el;
    if (PSTATE.EL != EL3 && HaveEL(EL3) &&
        SCR_EL3.EnAS0 == '0' && EL3SOOTrapPriority()) then
        UNDEFINED;
    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnAS0 == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnAS0 == '0';
            target_el = EL2;
        if !trap && EL2Enabled() && HaveFeatHCX() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1) then
            trap = !IsHCRXEL2Enabled() && HCRX_EL2.EnAS0 == '0';
            target_el = EL2;
        if !trap && PSTATE.EL != EL3 then
            trap = HaveEL(EL3) && SCR_EL3.EnAS0 == '0';
            target_el = EL3;
        if trap then
            if target_el == EL3 && Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                LDST64BTrap(target_el, iss);
    }

aarch64/exceptions/traps/CheckST64BVEnabled

// CheckST64BVEnabled()
// ====================
// Checks for trap on ST64BV instruction

CheckST64BVEnabled()
    boolean trap = FALSE;
    bits(25) iss = Zeros(25);
    bits(2) target_el;
    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnASR == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnASR == '0';
            target_el = EL2;
        if !trap && EL2Enabled() && HaveFeatHCX() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1) then
            trap = !IsHCRXEL2Enabled() && HCRX_EL2.EnASR == '0';
            target_el = EL2;
        if !trap && PSTATE.EL == EL3 then
            trap = HaveEL(EL3) && SCR_EL3.EnASR == '0';
            target_el = EL3;
        if trap then
            if target_el == EL3 && Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                LDST64BTrap(target_el, iss);


```c
LDST64BTrap(bits(2) target_el, bits(25) iss)
// preferred_exception_return = ThisInstrAddr(64);
veget_offset = 0x0;

exception = ExceptionSyndrome(Exception_LDST64BTrap);
exception.syndrome = iss;
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
return;
```

```c
WFETrapDelay(bits(2) target_el)
boolean delay_enabled;
integer delay;

case target_el of
when EL1
if !IsInHost() then
  delay_enabled = SCTLR_EL1.TWEDEn == '1';
  delay = 1 << (UInt(SCTLR_EL1.TWEDEL) + 8);
else
  delay_enabled = SCTLR_EL2.TWEDEn == '1';
  delay = 1 << (UInt(SCTLR_EL2.TWEDEL) + 8);
when EL2
  assert EL2Enabled();
  delay_enabled = HCR_EL2.TWEDEn == '1';
  delay = 1 << (UInt(HCR_EL2.TWEDEL) + 8);
when EL3
  delay_enabled = SCR_EL3.TWEDEn == '1';
  delay = 1 << (UInt(SCR_EL3.TWEDEL) + 8);

return (delay_enabled, delay);
```

```c
WaitForEventUntilDelay(boolean delay_enabled, integer delay);
```

## J1.1.3 aarch64/functions

This section includes the following pseudocode functions:

- [aarch64/functions/aborts/AArch64.FaultSyndrome](#)
- [aarch64/functions/aborts/LS64InstructionSyndrome](#)
- [aarch64/functions/cache/AArch64.DataMemZero](#)
- [aarch64/functions/cache/AArch64.TagMemZero](#)
- [aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass](#)
- [aarch64/functions/exclusive/AArch64.IsExclusiveVA](#)
- [aarch64/functions/exclusive/AArch64.MarkExclusiveVA](#)
• aarch64/functions/exclusive/AArch64.SetExclusiveMonitors on page J1-11085.
• aarch64/functions/fusedrstep/FPRSqrtStepFused on page J1-11086.
• aarch64/functions/fusedrstep/FPRRecipStepFused on page J1-11087.
• aarch64/functions/memory/AArch64.AccessIsTagChecked on page J1-11087.
• aarch64/functions/memory/AArch64.AddressWithAllocationTag on page J1-11088.
• aarch64/functions/memory/AArch64.AllocationTagFromAddress on page J1-11088.
• aarch64/functions/memory/AArch64.CheckAlignment on page J1-11088.
• aarch64/functions/memory/AArch64.CheckTag on page J1-11089.
• aarch64/functions/memory/AArch64.MemSingle on page J1-11089.
• aarch64/functions/memory/AArch64.MemTag on page J1-11091.
• aarch64/functions/memory/AArch64.PhysicalTag on page J1-11092.
• aarch64/functions/memory/AArch64.TranslateAddressForAtomicAccess on page J1-11092.
• aarch64/functions/memory/AddressSupportsLS64 on page J1-11093.
• aarch64/functions/memory/CheckAllInAlignedQuantity on page J1-11093.
• aarch64/functions/memory/CheckSPAlignment on page J1-11093.
• aarch64/functions/memory/CheckSingleAccessAttributes on page J1-11094.
• aarch64/functions/memory/IsTagCheckedInstruction on page J1-11095.
• aarch64/functions/memory/Mem on page J1-11095.
• aarch64/functions/memory/MemAtomic on page J1-11097.
• aarch64/functions/memory/MemAtomicCompareAndSwap on page J1-11098.
• aarch64/functions/memory/MemLoad64B on page J1-11098.
• aarch64/functions/memory/MemStore64B on page J1-11099.
• aarch64/functions/memory/MemStore64BWithRet on page J1-11099.
• aarch64/functions/memory/MemStore64BWithRetStatus on page J1-11100.
• aarch64/functions/memory/NVMem on page J1-11100.
• aarch64/functions/memory/PhysMemTagRead on page J1-11101.
• aarch64/functions/memory/PhysMemTagWrite on page J1-11101.
• aarch64/functions/memory/SetTagCheckedInstruction on page J1-11101.
• aarch64/functions/mops/COPYPostSizeChoice on page J1-11101.
• aarch64/functions/mops/COPYPreSizeChoice on page J1-11101.
• aarch64/functions/mops/COPYSizeChoice on page J1-11102.
• aarch64/functions/mops/CheckMOPSEnabled on page J1-11102.
• aarch64/functions/mops/MOPSSStage on page J1-11102.
• aarch64/functions/mops/MaxBlockSizeCopiedBytes on page J1-11102.
• aarch64/functions/mops/MemCpyAccessType on page J1-11102.
• aarch64/functions/mops/MemCpyDirectionChoice on page J1-11103.
• aarch64/functions/mops/MemCpyOptionA on page J1-11103.
• aarch64/functions/mops/MemCpyParametersIllformedE on page J1-11103.
• aarch64/functions/mops/MemCpyParametersIllformedM on page J1-11103.
• aarch64/functions/mops/MemCpyZeroSizeCheck on page J1-11103.
• aarch64/functions/mops/MemSetAccessType on page J1-11103.
• aarch64/functions/mops/MemSetParametersIllformedE on page J1-11104.
• aarch64/functions/mops/MemSetParametersIllformedM on page J1-11104.
• aarch64/functions/mops/MemSetZeroSizeCheck on page J1-11104.
• aarch64/functions/mops/MismatchedCpySetTargetEL on page J1-11104.
• aarch64/functions/mops/MismatchedCpyException on page J1-11104.
• aarch64/functions/mops/MismatchedMemCpyException on page J1-11105.
• aarch64/functions/mops/SETPostSizeChoice on page J1-11105.
• aarch64/functions/mops/SETPreSizeChoice on page J1-11106.
• aarch64/functions/mops/SETSizeChoice on page J1-11106.
• aarch64/functions/pac/addpac/AddPAC on page J1-11106.
• aarch64/functions/pac/addpacda/AddPACDA on page J1-11107.
• aarch64/functions/pac/addpacdb/AddPACDB on page J1-11108.
• aarch64/functions/pac/addpacga/AddPACGA on page J1-11109.
• aarch64/functions/pac/addpacga/AddPACIA on page J1-11109.
• aarch64/functions/pac/addpacib/AddPACIB on page J1-11110.
• aarch64/functions/pac/auth/AArch64.PACFailException on page J1-11111.
• aarch64/functions/pac/auth/Auth on page J1-11111.
• aarch64/functions/pac/authda/AuthDA on page J1-11112.
• aarch64/functions/pac/authdb/AuthDB on page J1-11113.
• aarch64/functions/pac/authia/AuthIA on page J1-11114.
• aarch64/functions/pac/authib/AuthIB on page J1-11115.
• aarch64/functions/pac/calcbottompacbit/CalculateBottomPACBit on page J1-11115.
• aarch64/functions/pac/computepac/ComputePAC on page J1-11116.
• aarch64/functions/pac/computepac/ComputePACIMPDEF on page J1-11116.
• aarch64/functions/pac/computepac/ComputePACQARMA on page J1-11116.
• aarch64/functions/pac/computepac/PACCellInvShuffle on page J1-11117.
• aarch64/functions/pac/computepac/PACCellShuffle on page J1-11118.
• aarch64/functions/pac/computepac/PACInvSub on page J1-11118.
• aarch64/functions/pac/computepac/PACMult on page J1-11119.
• aarch64/functions/pac/computepac/PACSub on page J1-11119.
• aarch64/functions/pac/computepac/PacSub1 on page J1-11120.
• aarch64/functions/pac/computepac/RC on page J1-11120.
• aarch64/functions/pac/computepac/RotCell on page J1-11120.
• aarch64/functions/pac/computepac/TweakCellInvRot on page J1-11120.
• aarch64/functions/pac/computepac/TweakCellRot on page J1-11121.
• aarch64/functions/pac/computepac/TweakInvShuffle on page J1-11121.
• aarch64/functions/pac/computepac/UsePACIMP on page J1-11122.
• aarch64/functions/pac/computepac/UsePACQARMA3 on page J1-11122.
• aarch64/functions/pac/computepac/UsePACQARMA5 on page J1-11122.
• aarch64/functions/pac/pac/ConstPACField on page J1-11122.
• aarch64/functions/pac/pac/HaveEnhancedPAC on page J1-11122.
• aarch64/functions/pac/pac/HaveEnhancedPAC2 on page J1-11122.
• aarch64/functions/pac/pac/HaveFPAC on page J1-11123.
• aarch64/functions/pac/pac/HaveFPACCombined on page J1-11123.
• aarch64/functions/pac/pac/HavePACExt on page J1-11123.
• aarch64/functions/pac/pac/HavePACIMPAuth on page J1-11123.
• aarch64/functions/pac/pac/HavePACIMPGeneric on page J1-11123.
• aarch64/functions/pac/pac/HavePACQARMA3Auth on page J1-11123.
• aarch64/functions/pac/pac/HavePACQARMA3Generic on page J1-11124.
• aarch64/functions/pac/pac/HavePACQARMA5Auth on page J1-11124.
• aarch64/functions/pac/pac/HavePACQARMA5Generic on page J1-11124.
• aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges on page J1-11124.
• aarch64/functions/pac/strip/Strip on page J1-11124.
• aarch64/functions/pac/trappacuse/TrapPACUse on page J1-11125.
• aarch64/functions/ras/AArch64.ESBOperation on page J1-11125.
• aarch64/functions/ras/AArch64.PhysicalSErrorSyndrome on page J1-11126.
• aarch64/functions/ras/AArch64.ReportDeferredSError on page J1-11126.
• aarch64/functions/ras/AArch64.vESBOperation on page J1-11126.
• aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers on page J1-11126.
• aarch64/functions/registers/AArch64.ResetGeneralRegisters on page J1-11127.
• aarch64/functions/registers/AArch64.ResetSIMDFPRegisters on page J1-11127.
• aarch64/functions/registers/AArch64.ResetSpecialRegisters on page J1-11127.
• aarch64/functions/registers/AArch64.ResetSystemRegisters on page J1-11128.
• aarch64/functions/registers/PC on page J1-11128.
• aarch64/functions/registers/SP on page J1-11128.
• aarch64/functions/registers/V on page J1-11128.
• aarch64/functions/registers/Vpart on page J1-11129.
• aarch64/functions/registers/X on page J1-11129.
• aarch64/functions/sme/HaveEBF16 on page J1-11130.
• aarch64/functions/sme/HaveSME on page J1-11130.
• aarch64/functions/sme/HaveSMEF64F64 on page J1-11130.
• aarch64/functions/sme/HaveSMEI16I64 on page J1-11130.
• aarch64/functions/sme/System on page J1-11130.
• aarch64/functions/sme/ZAslice on page J1-11131.
• aarch64/functions/sme/ZAtile on page J1-11131.
• aarch64/functions/sme/ZAvslice on page J1-11132.
• aarch64/functions/sme/ZAvslice on page J1-11132.
• aarch64/functions/sve/AArch32.IsFPEnabled on page J1-11133.
• aarch64/functions/sve/AArch64.IsFPEnabled on page J1-11133.
• aarch64/functions/sve/AnyActiveElement on page J1-11134.
• aarch64/functions/sve/BitDeposit on page J1-11134.
• aarch64/functions/sve/BitExtract on page J1-11134.
• aarch64/functions/sve/BitGroup on page J1-11134.
• aarch64/functions/sve/CeilPow2 on page J1-11135.
• aarch64/functions/sve/CheckNonStreamingSVEEnabled on page J1-11135.
• aarch64/functions/sve/CheckNormalSVEEnabled on page J1-11135.
• aarch64/functions/sve/CheckSMEAccess on page J1-11136.
• aarch64/functions/sve/CheckSMEAndZAEnabled on page J1-11137.
• aarch64/functions/sve/CheckSMEEnabled on page J1-11137.
• aarch64/functions/sve/CheckStreamingSVEAndZAEnabled on page J1-11138.
• aarch64/functions/sve/CheckStreamingSVEEnabled on page J1-11138.
• aarch64/functions/sve/CurrentVL on page J1-11138.
• aarch64/functions/sve/DecodePredCount on page J1-11138.
• aarch64/functions/sve/ElemFFR on page J1-11139.
• aarch64/functions/sve/ElemP on page J1-11139.
• aarch64/functions/sve/FFR on page J1-11139.
• aarch64/functions/sve/FPCompareNE on page J1-11140.
• aarch64/functions/sve/FPCompareUN on page J1-11140.
• aarch64/functions/sve/FPConvertSVE on page J1-11140.
• aarch64/functions/sve/FPExpA on page J1-11141.
• aarch64/functions/sve/FPExpCoefficient on page J1-11141.
• aarch64/functions/sve/FPLogB on page J1-11144.
• aarch64/functions/sve/FPMinNormal on page J1-11144.
• aarch64/functions/sve/FPOne on page J1-11144.
• aarch64/functions/sve/FPPointFive on page J1-11145.
• aarch64/functions/sve/FPProcess on page J1-11145.
- aarch64/functions/sve/FPScale on page J1-11145.
- aarch64/functions/sve/FPTrigMAdd on page J1-11146.
- aarch64/functions/sve/FPTrigMAddCoefficient on page J1-11146.
- aarch64/functions/sve/FPTrigSMul on page J1-11147.
- aarch64/functions/sve/FPTrigSSel on page J1-11147.
- aarch64/functions/sve/FirstActive on page J1-11147.
- aarch64/functions/sve/FloorPow2 on page J1-11148.
- aarch64/functions/sve/HaveSMEFullA64 on page J1-11148.
- aarch64/functions/sve/HaveSVE on page J1-11148.
- aarch64/functions/sve/HaveSVE2 on page J1-11148.
- aarch64/functions/sve/HaveSVE2AES on page J1-11148.
- aarch64/functions/sve/HaveSVE2BitPerm on page J1-11148.
- aarch64/functions/sve/HaveSVE2PMULL128 on page J1-11149.
- aarch64/functions/sve/HaveSVE2SHA3 on page J1-11149.
- aarch64/functions/sve/HaveSVE2SM4 on page J1-11149.
- aarch64/functions/sve/HaveSVEFP32MatMulExt on page J1-11149.
- aarch64/functions/sve/HaveSVEFP64MatMulExt on page J1-11149.
- aarch64/functions/sve/ImplementedSMEVectorLength on page J1-11149.
- aarch64/functions/sve/ImplementedSVEVectorLength on page J1-11150.
- aarch64/functions/sve/InStreamingMode on page J1-11150.
- aarch64/functions/sve/IsEven on page J1-11150.
- aarch64/functions/sve/IsFPEnabled on page J1-11150.
- aarch64/functions/sve/IsFullA64Enabled on page J1-11151.
- aarch64/functions/sve/IsOdd on page J1-11152.
- aarch64/functions/sve/IsPow2 on page J1-11152.
- aarch64/functions/sve/IsSVEEnabled on page J1-11152.
- aarch64/functions/sve/IsStreamingSVEEnabled on page J1-11152.
- aarch64/functions/sve/LastActive on page J1-11153.
- aarch64/functions/sve/LastActiveElement on page J1-11153.
- aarch64/functions/sve/MaxImplementedSVL on page J1-11153.
- aarch64/functions/sve/MaxImplementedVL on page J1-11153.
- aarch64/functions/sve/MaybeZeroSVEUppers on page J1-11153.
- aarch64/functions/sve/MemNF on page J1-11154.
- aarch64/functions/sve/MemSingleNF on page J1-11155.
- aarch64/functions/sve/NVL on page J1-11156.
- aarch64/functions/sve/NoneActive on page J1-11156.
- aarch64/functions/sve/P on page J1-11156.
- aarch64/functions/sve/PredTest on page J1-11157.
- aarch64/functions/sve/ReducePredicated on page J1-11157.
- aarch64/functions/sve/ResetSMEState on page J1-11157.
- aarch64/functions/sve/ResetSVEState on page J1-11157.
- aarch64/functions/sve/Reverse on page J1-11158.
- aarch64/functions/sve/SMEAccessTrap on page J1-11158.
- aarch64/functions/sve/SMEExceptionType on page J1-11158.
- aarch64/functions/sve/SVEAccessTrap on page J1-11158.
- aarch64/functions/sve/SVECmp on page J1-11159.
- aarch64/functions/sve/SYEMoveMaskPreferred on page J1-11159.
- aarch64/functions/sve/SYL on page J1-11160.
- aarch64/functions/sve/SetPSTATE_SM on page J1-11160.
• aarch64/functions/sve/SetPSTATE_SVCR on page J1-11160.
• aarch64/functions/sve/SetPSTATE_ZA on page J1-11160.
• aarch64/functions/sve/ShiftSat on page J1-11160.
• aarch64/functions/sve/SupportedNonPowerTwoVL on page J1-11161.
• aarch64/functions/sve/SupportedPowerTwoSVL on page J1-11161.
• aarch64/functions/sve/System on page J1-11161.
• aarch64/functions/sve/Z on page J1-11161.
• aarch64/functions/sysregisters/CNTKCTL on page J1-11161.
• aarch64/functions/sysregisters/CNTKCTLT on page J1-11162.
• aarch64/functions/sysregisters/CPACR on page J1-11162.
• aarch64/functions/sysregisters/CPACRT on page J1-11162.
• aarch64/functions/sysregisters/ELR on page J1-11162.
• aarch64/functions/sysregisters/ESR on page J1-11163.
• aarch64/functions/sysregisters/ESRT on page J1-11163.
• aarch64/functions/sysregisters/FAR on page J1-11163.
• aarch64/functions/sysregisters/MAIRT on page J1-11164.
• aarch64/functions/sysregisters/SCTRLR on page J1-11164.
• aarch64/functions/sysregisters/SCTRLRT on page J1-11165.
• aarch64/functions/sysregisters/VBAR on page J1-11165.
• aarch64/functions/system/AArch64.AllocationTagAccessIsEnabled on page J1-11165.
• aarch64/functions/system/AArch64.CheckSystemAccess on page J1-11165.
• aarch64/functions/system/AArch64.ChooseNonExcludedTag on page J1-11166.
• aarch64/functions/system/AArch64.ExecutingBROrBLROrRetInstr on page J1-11166.
• aarch64/functions/system/AArch64.ExecutingBTIIInstr on page J1-11166.
• aarch64/functions/system/AArch64.ExecutingERETInstr on page J1-11167.
• aarch64/functions/system/AArch64.ImpDefSysInstr on page J1-11167.
• aarch64/functions/system/AArch64.ImpDefSysInstrWithResult on page J1-11167.
• aarch64/functions/system/AArch64.ImpDefSysRegRead on page J1-11167.
• aarch64/functions/system/AArch64.ImpDefSysRegWrite on page J1-11167.
• aarch64/functions/system/AArch64.NextRandomTagBit on page J1-11167.
• aarch64/functions/system/AArch64.RandomTag on page J1-11167.
• aarch64/functions/system/AArch64.SysInstr on page J1-11168.
• aarch64/functions/system/AArch64.SysInstrWithResult on page J1-11168.
• aarch64/functions/system/AArch64.SysRegRead on page J1-11168.
• aarch64/functions/system/AArch64.SysRegWrite on page J1-11168.
• aarch64/functions/system/BTypeCompatible on page J1-11168.
• aarch64/functions/system/BTypeCompatible_BTI on page J1-11168.
• aarch64/functions/system/BTypeCompatible_PACIXSP on page J1-11168.
• aarch64/functions/system/BTypeNext on page J1-11169.
• aarch64/functions/system/ChooseRandomNonExcludedTag on page J1-11169.
• aarch64/functions/system/InGuardedPage on page J1-11169.
• aarch64/functions/system/IsHCRXEL2Enabled on page J1-11169.
• aarch64/functions/system/SetBTypeCompatible on page J1-11169.
• aarch64/functions/system/SetBTypeNext on page J1-11170.
• aarch64/functions/system/SetInGuardedPage on page J1-11170.
• aarch64/functions/tme/CheckTMEEnabled on page J1-11170.
• aarch64/functions/tme/CheckTransactionalSystemAccess on page J1-11170.
• aarch64/functions/tme/CommitTransactionalWrites on page J1-11171.
• aarch64/functions/tme/DiscardTransactionalWrites on page J1-11171.
aarch64/functions/tme/FailTransaction on page J1-11171.
aarch64/functions/tme/MemHasTransactionalAccess on page J1-11171.
aarch64/functions/tme/RestoreTransactionCheckpoint on page J1-11172.
aarch64/functions/tme/StartTrackingTransactionalReadsWrites on page J1-11172.
aarch64/functions/tme/TMFailure on page J1-11172.
aarch64/functions/tme/TMSState on page J1-11173.
aarch64/functions/tme/TSTATE on page J1-11173.
aarch64/functions/tme/TakeTransactionCheckpoint on page J1-11173.
aarch64/functions/tme/TransactionStartTrap on page J1-11174.

aarch64/functions/aborts/AArch64.FaultSyndrome

// AArch64.FaultSyndrome()
// =======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// an Exception level using AArch64.

<bits(25), bits(5)) AArch64.FaultSyndrome(boolean d_side, FaultRecord fault)
assert fault.statuscode != Fault_None;

bits(25) iss = Zeros(25);
bits(5) iss2 = Zeros(5);

if HaveRASExt() && fault.statuscode == Fault_SyncExternal then
  iss<12:11> = fault.errortype; // SET
if d_side then
  if HaveFeatLS64() && fault.acctype == AccType_ATOMICLS64 then
    (iss, iss2) = LS64InstructionSyndrome();
  elseif IsSecondStage(fault) && !fault.s2fs1walk &&
    (!HaveRASExt() && fault.acctype == AccType_TTW &&
    boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk")) then
    iss<24:14> = LSInstructionSyndrome();
  if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
    iss<13> = '1';  // Fault is generated by use of VNCR_EL2
  if HaveFeatLS64() && fault.statuscode IN (Fault_AccessFlag, Fault_Translation,
    Fault_Permission) then
    iss<12:11> = GetLoadStoreType();
  if fault.acctype IN (AccType_DC, AccType_IC, AccType_AT, AccType_ATPAN) then
    iss<8> = '1';  iss<6> = '1';
  else
    iss<6> = if fault.write then '1' else '0';
  if IsExternalAbort(fault) then iss<9> = fault.extflag;
  iss<7> = if fault.s2fs1walk then '1' else '0';
  iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);
  return (iss, iss2);

bits(6) EncodeGPCSC(GPCFRecord gpcf)
assert gpcf.level IN [0,1];

case gpcf.gpf of
  when GPCF_AddressSize return '000000':gpcf.level<0>;
  when GPCF_Walk return '000010':gpcf.level<0>;
  when GPCF_Fail return '000100':gpcf.level<0>;
  when GPCF_EABT return '001010':gpcf.level<0>;

aarch64/functions/aborts/LS64InstructionSyndrome

// Returns the syndrome information and LST for a Data Abort by a
// ST64B, ST64BV, ST64BV0, or LD64B instruction. The syndrome information
// includes the ISS2, extended syndrome field.
(bits(5), bits(11)) LS64InstructionSyndrome();

aarch64/functions/cache/AArch64.DataMemZero

// AArch64.DataMemZero()
// =====================
// Write Zero to data memory.
AArch64.DataMemZero(bits(64) regval, bits(64) vaddress, AddressDescriptor memaddrdesc_in, integer size)
    iswrite = TRUE;
    AddressDescriptor memaddrdesc = memaddrdesc_in;
    accdesc = CreateAccessDescriptor(AccType_DCZVA);
    if HaveTME() then
        accdesc.transactional = TSTATE.depth > 0;
        if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc.memattrs) then
            FailTransaction(TMFailure_IMP, FALSE);
    for i = 0 to size-1
        if HaveMTE2Ext() then
            if AArch64.AccessIsTagChecked(vaddress, AccType_DCZVA) then
                bits(4) ptag = AArch64.PhysicalTag(vaddress);
                if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
                    if boolean IMPLEMENTATION_DEFINED "DC_ZVA tag fault reported with lowest faulting address" then
                        AArch64.TagCheckFault(vaddress, AccType_DCZVA, iswrite);
                    else
                        AArch64.TagCheckFault(regval, AccType_DCZVA, iswrite);
            memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Zeros(8));
            if IsFault(memstatus) then
                HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    return;

aarch64/functions/cache/AArch64.TagMemZero

// AArch64.TagMemZero()
// ====================
// Write Zero to tag memory.
AArch64.TagMemZero(bits(64) vaddress_in, integer size)
    bits(64) vaddress = vaddress_in;
    integer count = size >> LOG2_TAG_GRANULE;
    bits(4) tag = AArch64.AllocationTagFromAddress(vaddress);
    for i = 0 to count-1
        AArch64.MemTag[vaddress, AccType_NORMAL] = tag;
        vaddress = vaddress + TAG_GRANULE;
    return;

aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass

// AArch64.ExclusiveMonitorsPass()
// ===============================
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.
boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)
// It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
// before or after the check on the local Exclusives monitor. As a result a failure
// of the local monitor can occur on some implementations even if the memory
// access would give a memory abort.

acctype = AccType_ATOMIC;
iswrite = TRUE;
aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
passed = AArch64.IsExclusiveVA(address, ProcessorID(), size);
if !passed then
    return FALSE;
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);
passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
ClearExclusiveLocal(ProcessorID());
if passed then
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
return passed;

aarch64/functions/exclusive/AArch64.IsExclusiveVA

// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.

// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
// It is always safe to return TRUE which will check the physical address only.
boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);

aarch64/functions/exclusive/AArch64.MarkExclusiveVA

// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);

aarch64/functions/exclusive/AArch64.SetExclusiveMonitors

// AArch64.SetExclusiveMonitors()
// ==============================
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.
AArch64.SetExclusiveMonitors(bits(64) address, integer size)
    acctype = AccType_ATOMIC;
iswrite = FALSE;
    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    return;

if memaddrdesc.memattrs.shareability != Shareability_NSH then
    MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    AArch64.MarkExclusiveVA(address, ProcessorID(), size);

aarch64/functions/fusedrstep/FPRSqrtStepFused

    // FPRSqrtStepFused()
    // ===============

    bits(N) FPRSqrtStepFused(bits(N) op1_in, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    bits(N) op1 = op1_in;
    boolean done;
    FPCRType fpcr = FPCR[];
    op1 = FPNeg(op1);
    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp;                          // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';             // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00';                // Use RNE rounding mode

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    FPRounding rounding = FPRoundingMode(fpcr);

    if !done then
        inf1 = (type1 == FPTYPE_Infinity);
        inf2 = (type2 == FPTYPE_Infinity);
        zero1 = (type1 == FPTYPE_Zero);
        zero2 = (type2 == FPTYPE_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0', N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        else
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);
        end if
    end if

    return result;

aarch64/functions/fusedrstep/FPRecipStepFused

    // FPRecipStepFused()
    // ================

    bits(N) FPRecipStepFused(bits(N) op1_in, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) op1 = op1_in;
    bits(N) result;
    boolean done;
FPCRTypene fpcr = FPCR[];
op1 = FPNeg(op1);

boolean altfp = HaveAltFP() && fpcr.AH == '1';
boolean fpexc = !altfp;  // Generate no floating-point exceptions
if altfp then fpcr.<FIZ,FZ> = '11';  // Flush denormal input and output to zero
if altfp then fpcr.RMode = '00';  // Use RNE rounding mode

(type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
(type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
FPRounding rounding = FPRoundingMode(fpcr);

if !done then
    inf1  = (type1 == FPType_Infinity);
    inf2  = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);
    zero2 = (type2 == FPType_Zero);
    if (inf1 && zero2) || (zero1 && inf2) then
        result = FPTwo('0', N);
    elsif inf1 || inf2 then
        result = FPInfinity(sign1 EOR sign2, N);
    else
        // Fully fused multiply-add
        result_value = 2.0 + (value1 * value2);
        if result_value == 0.0 then
            // Sign of exact zero result depends on rounding mode
            sign = if rounding == FPRounding_NEGINF then '1' else '0';
            result = FPZero(sign, N);
        else
            result = FPRound(result_value, fpcr, rounding, fpexc, N);
        return result;

aarch64/functions/memory/AArch64.AccessIsTagChecked

// AArch64.AccessIsTagChecked()
// ============================
// TRUE if a given access is tag-checked, FALSE otherwise.

boolean AArch64.AccessIsTagChecked(bits(64) vaddr, AccType acctype)
if PSTATE.M<4> == '1' then return FALSE;

boolean is_instr = FALSE;
if EffectiveTBI(vaddr, is_instr, PSTATE.EL) == '0' then
    return FALSE;

if EffectiveTCMA(vaddr, PSTATE.EL) == '1' && (vaddr<59:55> == '00000' || vaddr<59:55> == '11111')
    return FALSE;

if !AArch64.AllocationTagAccessIsEnabled(acctype) then
    return FALSE;

if acctype IN {AccType_IFETCH, AccType_TTW, AccType_DC, AccType_IC} then
    return FALSE;

if acctype == AccType_NV2REGISTER then
    return FALSE;

if PSTATE.TCO=='1' then
    return FALSE;

if (HaveSME() && PSTATE.SM == '1' &&
    acctype IN {AccType_VEC, AccType_VECSTREAM,
boolean IMPLEMENTATION_DEFINED "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
return FALSE;

if(HaveSME() &&
    acctype IN {AccType_SME, AccType_SMESTREAM} &&
    boolean IMPLEMENTATION_DEFINED "No tag checking of SME LDR & STR instructions") then
return FALSE;

if !IsTagCheckedInstruction() then
return FALSE;

return TRUE;

aarch64/functions/memory/AArch64.AddressWithAllocationTag

// AArch64.AddressWithAllocationTag()
// ==================================
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as '0000'.

bits(64) AArch64.AddressWithAllocationTag(bits(64) address, AccType acctype, bits(4) allocation_tag)
    bits(64) result = address;
    bits(4) tag;
    if AArch64.AllocationTagAccessIsEnabled(acctype) then
        tag = allocation_tag;
    else
        tag = '0000';
    result<59:56> = tag;
    return result;

aarch64/functions/memory/AArch64.AllocationTagFromAddress

// AArch64.AllocationTagFromAddress()
// ==================================
// Generate an Allocation Tag from a 64-bit value containing a Logical Address Tag.

bits(4) AArch64.AllocationTagFromAddress(bits(64) tagged_address)
    return tagged_address<59:56>;

aarch64/functions/memory/AArch64.CheckAlignment

// AArch64.CheckAlignment()
// ========================

boolean AArch64.CheckAlignment(bits(64) address, integer alignment, AccType acctype,
    boolean iswrite)

aligned = (address == Align(address, alignment));
atomic = acctype IN { AccType_ATOMIC, AccType_ATOMICCRW, AccType_ORDEREDATOMIC,
    AccType_ORDEREDATOMICRW, AccType_ATOMICLS64, AccType_A32LSMD};
ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED,
    AccType_ORDEREDDATOMIC, AccType_ORDEREDDATOMICRW };

boolean check;
if SCTLR[].A == '1' then check = TRUE;
else if HaveLSE2Ext() then
    check = (UInt(address<3:0>) + alignment > 16) && ((ordered && SCTLR[].nAA == '0') || atomic);
else check = atomic || ordered;
if check && !aligned then
  secondstage = FALSE;
  AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));
return aligned;

aarch64/functions/memory/AArch64.CheckTag

// AArch64.CheckTag()
// ===============
// Performs a Tag Check operation for a memory access and returns
// whether the check passed

boolean AArch64.CheckTag(AddressDescriptor memaddrdesc, AccessDescriptor accdesc, bits(4) ptag, boolean write)
  if memaddrdesc.memattrs.tagged then
    (memstatus, readtag) = PhysMemTagRead(memaddrdesc, accdesc);
    if IsFault(memstatus) then
      HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
    return ptag == readtag;
  else
    return TRUE;

aarch64/functions/memory/AArch64.MemSingle

// AArch64.MemSingle[] - non-assignment (read) form
// ==================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned]
  boolean ispair = FALSE;
  return AArch64.MemSingle[address, size, acctype, aligned, ispair];
else
    accdesc = CreateAccessDescriptor(acctype);

boolean istagchecked = istagaccess;
if istagaccess then
    bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
    if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
        AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);

.atomic, splitpair) = CheckSingleAccessAttributes(address, memaddrdesc.memattrs, size, acctype,
iswrite, aligned, ispair);
PhysMemRetStatus memstatus;
if atomic then
    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
elsif splitpair then
    assert ispair;
    bits(halfsize * 8) lowhalf, highhalf;
    (memstatus, lowhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);
    memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
    (memstatus, highhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);
    value = highhalf:lowhalf;
else
    for i = 0 to size-1
        (memstatus, value<8*i+7:8*i>) = PhysMemRead(memaddrdesc, 1, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    return value;

// AArch64.MemSingle[] - assignment (write) form
// ==============================================================
AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned] = bits(size*8) value
    boolean ispair = FALSE;
    AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
    return;

// AArch64.MemSingle[] - assignment (write) form
// ==============================================================
// Perform an atomic, little-endian write of 'size' bytes.
AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned, boolean ispair] = bits(size*8) value
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    if HaveLSE2Ext() then
        assert CheckAlignInAlignedQuantity(address, size, 16);
    else
        assert address == Align(address, size);

AddressDescriptor memaddrdesc;
iswrite = TRUE;

boolean istagaccess = HaveMTE2Ext() & AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype);
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

// Memory array access
AccessDescriptor accdesc;
if HaveTME() then
    accdesc = CreateAccessDescriptor(acctype);
    accdesc.transactional = TSTATE.depth > 0;
else
    accdesc = CreateAccessDescriptor(acctype);

boolean istagchecked = istagaccess;
if istagchecked then
    bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
    if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
        AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);

PhysMemRetStatus memstatus;
    (atomic, splitpair) = CheckSingleAccessAttributes(address, memaddrdesc.memattrs, size, acctype, iswrite, aligned, ispair);
    if atomic then
        memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    else if splitpair then
        assert ispair;
        bits(halfsize*8) lowhalf, highhalf;
        <highhalf, lowhalf> = value;
        memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, lowhalf);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
        memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, highhalf);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
    else
        for i = 0 to size-1
            memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, value<8*i:8*i>);
            if IsFault(memstatus) then
                HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
return;

aarch64/functions/memory/AArch64.MemTag

// AArch64.MemTag[] - non-assignment (read) form
// ==============================================================
// Load an Allocation Tag from memory.

bits(4) AArch64.MemTag[bits(64) address, AccType acctype]
    AddressDescriptor memaddrdesc;
    bits(4) value;
    iswrite = FALSE;
    aligned = TRUE;
    istagaccess = AArch64.AllocationTagAccessIsEnabled(acctype);
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, TAG_GRANULE);
    accdesc = CreateAccessDescriptor(acctype);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
// Return the granule tag if tagging is enabled...
if istagaccess && memaddrdesc.memattrs.tagged then
    (memstatus, tag) = PhysMemTagRead(memaddrdesc, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
    return tag;
else
    // ...otherwise read tag as zero.
    return '0000';

// AArch64.MemTag[] - assignment (write) form
// =========================================
// Store an Allocation Tag to memory.
AArch64.MemTag[bits(64) address, AccType acctype] = bits(4) value
AddressDescriptor memaddrdesc;
iswrite = TRUE;

// Stores of allocation tags must be aligned
if address != Align(address, TAG_GRANULE) then
    boolean secondstage = FALSE;
    AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));
aligned = TRUE;
istagaccess = AArch64.AllocationTagAccessIsEnabled(acctype);
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, TAG_GRANULE);

// It is CONSTRAINED UNPREDICTABLE if tags stored to memory locations marked as Device
// generate an Alignment Fault or store the data to locations.
if memaddrdesc.memattrs.memtype == MemType_Device then
    c = ConstrainUnpredictable(Unpredictable_DEVICETAGSTORE);
    assert c IN {Constraint_NONE, Constraint_FAULT};
    if c == Constraint_FAULT then
        boolean secondstage = FALSE;
        AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);
acci_desc = CreateAccessDescriptor(acctype);

// Memory array access
if istagaccess && memaddrdesc.memattrs.tagged then
    memstatus = PhysMemTagWrite(memaddrdesc, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);

aarch64/functions/memory/AArch64.PhysicalTag

// AArch64.PhysicalTag()
// =====================
// Generate a Physical Tag from a Logical Tag in an address

bits(4) AArch64.PhysicalTag(bits(64) vaddr)
    return vaddr<59:56>;

aarch64/functions/memory/AArch64.TranslateAddressForAtomicAccess

// AArch64.TranslateAddressForAtomicAccess()
// ==========================================
// Performs an alignment check for atomic memory operations.
// Also translates 64-bit Virtual Address into Physical Address.
AddressDescriptor AArch64.TranslateAddressForAtomicAccess(bits(64) address, integer sizeinbits)
boolean iswrite = FALSE;
size = sizeinbits DIV 8;

assert size IN {1, 2, 4, 8, 16};
aligned = AArch64.CheckAlignment(address, size, AccType_ATOMICRW, iswrite);

// MMU or MPU lookup
boolean istagaccess = HaveMTE2Ext() && AArch64.AccessIsTagChecked(address, AccType_ATOMICRW);
memaddrdesc = AArch64.TranslateAddress(address, AccType_ATOMICRW, iswrite, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  AArch64.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH then
  ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

boolean istagchecked = istagaccess;
if istagchecked then
  bits(4) ptag = AArch64.PhysicalTag(address);
  accdesc = CreateAccessDescriptor(AccType_ATOMICRW);
  if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
    AArch64.TagCheckFault(address, AccType_ATOMICRW, iswrite);
return memaddrdesc;

aarch64/functions/memory/AddressSupportsLS64

// Returns TRUE if the 64-byte block following the given address supports the
// LD64B and ST64B instructions, and FALSE otherwise.
boolean AddressSupportsLS64(bits(64) address);

aarch64/functions/memory/CheckAllInAlignedQuantity

// CheckAllInAlignedQuantity()
// ===========================
// Returns TRUE if all accessed bytes are within one aligned quantity, FALSE otherwise.
boolean CheckAllInAlignedQuantity(bits(64) address, integer size, integer alignment)
assert(size <= alignment);
return Align(address+size-1, alignment) == Align(address, alignment);

aarch64/functions/memory/CheckSPAlignment

// CheckSPAlignment()
// ==================
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
bits(64) sp = SP[];
boolean stack_align_check;
if PSTATE.EL == EL0 then
  stack_align_check = (SCTLR[].SA0 != '0');
else
  stack_align_check = (SCTLR[].SA != '0');
if stack_align_check && sp != Align(sp, 16) then
  AArch64.SPAlignmentFault();
return;

aarch64/functions/memory/CheckSingleAccessAttributes

// CheckSingleAccessAttributes()
// =============================
// When FEAT_LSE2 is implemented, a MemSingle[] access needs to be further assessed once the memory
// attributes are determined.
// If it was aligned to access size or targets Normal Inner Write-Back, Outer Write-Back Cacheable
// memory then it is single copy atomic and there is no alignment fault.
// If not, for exclusives, atomics and non atomic acquire release instructions - it is CONSTRAINED
// UNPREDICTABLE
// if they generate an alignment fault. If they do not generate an alignment fault - they are
// single copy atomic.
// Otherwise it is IMPLEMENTATION DEFINED - if they are single copy atomic.
// The function returns (atomic, splitpair), where
// atomic indicates if the access is single copy atomic.
// splitpair indicates that a load/store pair is split into 2 single copy atomic accesses.
// when atomic and splitpair are both FALSE - the access is not single copy atomic and may be treated
// as byte accesses.

(Boolean, Boolean) CheckSingleAccessAttributes(bits(64) address, MemoryAttributes memattrs, integer
size,
  AccType acctype, boolean iswrite, boolean aligned, boolean ispair)

  isnormalwb = (memattrs.memtype == MemType_Normal &&
                memattrs.inner.attrs == MemAttr_WB &&
                memattrs.outer.attrs == MemAttr_WB);

  atomic    = TRUE;
  splitpair = FALSE;
  if isnormalwb then return (atomic, splitpair);

  accatomic  = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC,
                           AccType_ORDEREDATOMICRW, AccType_ATOMICLS64, AccType_A32LSMD};

  ordered    = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED,
                           AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW,
                           AccType_ATOMICORDEREDATOMIC, AccType_ATOMICORDEREDATOMICRW };

  if !aligned && (accatomic || ordered) then
    atomic = ConstrainUnpredictableBool(Unpredictable_MISALIGNEDATOMIC);
    if nonatomic then
      secondstage = FALSE;
      AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));
    else
      return (atomic, splitpair);
  end

  if ispair && aligned then
    // load / store pair requests that are aligned to each register access are split into 2 single
    copy atomic accesses
    atomic    = FALSE;
    splitpair = TRUE;
    return (atomic, splitpair);
  end

  if aligned then
    return (atomic, splitpair);

  atomic = boolean IMPLEMENTATION_DEFINED "Misaligned accesses within 16 byte aligned memory but not
  Normal Cacheable Writeback are Atomic";

  return (atomic, splitpair);
aarch64/functions/memory/IsTagCheckedInstruction

// Returns True if the current instruction uses tag-checked memory access, // False otherwise.
boolean IsTagCheckedInstruction();

aarch64/functions/memory/Mem

// Mem[] - non-assignment (read) form // ==================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access. // Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size, AccType acctype]
  boolean ispair = FALSE;
  return Mem[address, size, acctype, ispair];

bits(size*8) Mem[bits(64) address, integer size, AccType acctype, boolean ispair]
  assert size IN {1, 2, 4, 8, 16};
  constant halfsize = size DIV 2;
  bits(size*8) value;
  bits(halfsize*8) lowhalf, highhalf;
  boolean iswrite = FALSE;
  boolean aligned;
  if ispair then
    // check alignment on size of element accessed, not overall access size
    aligned = AArch64.CheckAlignment(address, halfsize, acctype, iswrite);
  else
    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
  boolean atomic;
  if size != 16 || !acctype IN {AccType_VEC, AccType_VECSTREAM}) then
    if !HaveLSE2Ext() then
      atomic = aligned;
    else
      atomic = CheckAllInAlignedQuantity(address, size, 16);
  elsif acctype IN {AccType_VEC, AccType_VECSTREAM} then
    // 128-bit SIMD&FP loads are treated as a pair of 64-bit single-copy atomic accesses
    // 64-bit aligned.
    atomic = address == Align(address, 8);
  else
    // 16-byte integer access
    atomic = address == Align(address, 16);
  if !atomic && ispair && address == Align(address, halfsize) then
    single_is_pair = FALSE;
    single_is_aligned = TRUE;
  lowhalf = AArch64.MemSingle[address, halfsize, acctype,
    single_is_aligned, single_is_pair];
  highhalf = AArch64.MemSingle[address + halfsize, halfsize, acctype,
    single_is_aligned, single_is_pair];
  value = highhalf:lowhalf;
  elsif atomic && ispair then
    value = AArch64.MemSingle[address, size, acctype, aligned, ispair];
  elsif !atomic then
    assert size > 1;
    value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];
  // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
  // access will generate an Alignment Fault, as to get this far means the first byte did
  // not, so we must be changing to a new translation page.
  if !aligned then
    c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
assert c IN {Constraint_FAULT, Constraint_NONE};
if c == Constraint_NONE then aligned = TRUE;
for i = 1 to size-1
value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, acctype, aligned];
elsif size == 16 & acctype IN {AccType_VEC, AccType_VECSTREAM} then
lowhalf = AArch64.MemSingle[address, halfsize, acctype, aligned, ispair];
highhalf = AArch64.MemSingle[address + halfsize, halfsize, acctype, aligned, ispair];
value = highhalf:lowhalf;
else
value = AArch64.MemSingle[address, size, acctype, aligned, ispair];
if BigEndian(acctype) then
value = BigEndianReverse(value);
return value;

// Mem[] - assignment (write) form
// ==============================================================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.
Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value_in
boolean ispair = FALSE;
Mem[address, size, acctype, ispair] = value_in;

Mem[bits(64) address, integer size, AccType acctype, boolean ispair] = bits(size*8) value_in
boolean iswrite = TRUE;
constant halfsize = size DIV 2;
bits(size*8) value = value_in;
bits(halfsize*8) lowhalf, highhalf;
boolean atomic;
boolean aligned;
if BigEndian(acctype) then
value = BigEndianReverse(value);
if ispair then
// check alignment on size of element accessed, not overall access size
aligned = AArch64.CheckAlignment(address, halfsize, acctype, iswrite);
else
aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
if ispair then
atomic = CheckAllInAlignedQuantity(address, size, 16);
elsif size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
if !HaveLSE2Ext() then
atomic = aligned;
else
atomic = CheckAllInAlignedQuantity(address, size, 16);
elsif (acctype IN {AccType_VEC, AccType_VECSTREAM}) then
// 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
// 64-bit aligned.
atomic = address == Align(address, 8);
else
// 16-byte integer access
atomic = address == Align(address, 16);
if !atomic & ispair & address == Align(address, halfsize) then
single_is_aligned = TRUE;
<highhalf, lowhalf> = value;
AArch64.MemSingle[address, halfsize, acctype,
single_is_aligned, ispair] = lowhalf;
AArch64.MemSingle[address + halfsize, halfsize, acctype,
single_is_aligned, ispair] = highhalf;
elsif atomic & ispair then
AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
elsif !atomic then
assert size > 1;
AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;}
// For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory access will generate an Alignment Fault, as to get this far means the first byte did not, so we must be changing to a new translation page.
if !aligned then
c = ConstrumUnpredictable(Unpredictable_DEVPAGE2);
assert c IN {Constraint_FAULT, Constraint_NONE};
if c == Constraint_NONE then aligned = TRUE;

for i = 1 to size-1
AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
elsif size == 16 & acctype IN {AccType_VEC, AccType_VECSTREAM} then
<highhalf, lowhalf> = value;
AArch64.MemSingle[address, halfsize, acctype, aligned, ispair] = lowhalf;
AArch64.MemSingle[address + halfsize, halfsize, acctype, aligned, ispair] = highhalf;
else
AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
return;

aarch64/functions/memory/MemAtomic

// MemAtomic()
// ============
// Performs load and store memory operations for a given virtual address.

bits(size) MemAtomic(bits(64) address, MemAtomicOp op, bits(size) value, AccType ldacctype, AccType stacctype)
bits(size) newvalue;
memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
ldaccdesc = CreateAccessDescriptor(ldacctype);
staccdesc = CreateAccessDescriptor(stacctype);

// All observers in the shareability domain observe the following load and store atomically.
(memstatus, oldvalue) = PhysMemRead(memaddrdesc, size DIV 8, ldaccdesc);
if IsFault(memstatus) then
HandleExternalReadAbort(memstatus, memaddrdesc, size DIV 8, ldaccdesc);
if BigEndian(ldacctype) then
oldvalue = BigEndianReverse(oldvalue);

case op of
when MemAtomicOp_ADD   newvalue = oldvalue + value;
when MemAtomicOp_BIC   newvalue = oldvalue AND NOT(value);
when MemAtomicOp_EOR   newvalue = oldvalue EOR value;
when MemAtomicOp_ORR   newvalue = oldvalue OR value;
when MemAtomicOp_SMAX  newvalue = if SInt(oldvalue) > SInt(value) then oldvalue else value;
when MemAtomicOp_SMIN  newvalue = if SInt(oldvalue) > SInt(value) then value else oldvalue;
when MemAtomicOp_UMAX  newvalue = if UInt(oldvalue) > UInt(value) then oldvalue else value;
when MemAtomicOp_UMIN  newvalue = if UInt(oldvalue) > UInt(value) then value else oldvalue;
when MemAtomicOp_SWP   newvalue = value;
if BigEndian(stacctype) then
newvalue = BigEndianReverse(newvalue);
memstatus = PhysMemWrite(memaddrdesc, size DIV 8, staccdesc, newvalue);
if IsFault(memstatus) then
HandleExternalWriteAbort(memstatus, memaddrdesc, size DIV 8, staccdesc);
// Load operations return the old (pre-operation) value return oldvalue;
aarch64/functions/memory/MemAtomicCompareAndSwap

// MemAtomicCompareAndSwap()
// ========================================
// Compares the value stored at the passed-in memory address against the passed-in expected
// value. If the comparison is successful, the value at the passed-in memory address is swapped
// with the passed-in new_value.

bits(size) MemAtomicCompareAndSwap(bits(64) address, bits(size) expectedvalue,
  bits(size) newvalue_in, AccType ldacctype, AccType stacctype)
  bits(size) newvalue = newvalue_in;
  memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
  ldaccdesc = CreateAccessDescriptor(ldacctype);
  staccdesc = CreateAccessDescriptor(stacctype);
  // All observers in the shareability domain observe the
  // following load and store atomically.
  (memstatus, oldvalue) = PhysMemRead(memaddrdesc, size DIV 8, ldaccdesc);
  if IsFault(memstatus) then
    HandleExternalReadAbort(memstatus, memaddrdesc, size DIV 8, ldaccdesc);
  if BigEndian(ldacctype) then
    oldvalue = BigEndianReverse(oldvalue);
  if oldvalue == expectedvalue then
    if BigEndian(stacctype) then
      newvalue = BigEndianReverse(newvalue);
    memstatus = PhysMemWrite(memaddrdesc, size DIV 8, staccdesc, newvalue);
    if IsFault(memstatus) then
      HandleExternalWriteAbort(memstatus, memaddrdesc, size DIV 8, staccdesc);
  return oldvalue;

aarch64/functions/memory/MemLoad64B

// MemLoad64B()
// ============
// Performs an atomic 64-byte read from a given virtual address.

bits(512) MemLoad64B(bits(64) address, AccType acctype)
  bits(512) data;
  boolean iswrite = FALSE;
  constant integer size = 64;
  aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
  if !AddressSupportsLS64(address) then
    c = ConstraintUnpredictable(Unpredictable_LS64UNSUPPORTED);
    assert c IN [Constraint_LIMITED_ATOMICITY, Constraint_FAULT];
    if c == Constraint_FAULT then
      // Generate a stage 1 Data Abort reported using the DFS code of 110101.
      boolean secondstage = FALSE;
      boolean s2fs1walk = FALSE;
      FaultRecord fault = AArch64.ExclusiveFault(acctype, iswrite, secondstage, s2fs1walk);
      AArch64.Abort(address, fault);
    else
      // Accesses are not single-copy atomic above the byte level
      for i = 0 to 63
        data<7+8*i : 8*i> = AArch64.MemSingle[address+8*i, 1, acctype, aligned];
      return data;
  AddressDescriptor memaddrdesc;
  istagaccess = HaveMTE2Ext() & AArch64.AccessIsTagChecked(address, acctype);
  memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
  // Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
if HaveTME() then
    accdesc.transactional = TSTATE.depth > 0;

boolean istagchecked = istagaccess;
if istagchecked then
    bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
    if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
        AArch64.TagCheckFault(address, acctype, iswrite);

PhysMemRetStatus memstatus;
(memstatus, data) = PhysMemRead(memaddrdesc, size, accdesc);
if IsFault(memstatus) then
    HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
return data;

aarch64/functions/memory/MemStore64B

// MemStore64B()
// =============
// Performs an atomic 64-byte store to a given virtual address. Function does
// not return the status of the store.
MemStore64B(bits(64) address, bits(512) value, AccType acctype)
boolean iswrite = TRUE;
constant integer size = 64;
aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
if !AddressSupportsLS64(address) then
    c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
    assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};
    if c == Constraint_FAULT then
        // Generate a Data Abort reported using the DFSC code of 110101.
        boolean secondstage = FALSE;
        boolean s2fs1walk = FALSE;
        fault = AArch64.ExclusiveFault(acctype, iswrite, secondstage, s2fs1walk);
        AArch64.Abort(address, fault);
    else
        // Accesses are not single-copy atomic above the byte level.
        for i = 0 to 63
            AArch64.MemSingle[address+8*i, 1, acctype, aligned] = value<7+8*i : 8*i>;
    else
        -= MemStore64BWithRet(address, value, acctype); // Return status is ignored by ST64B
    return;

aarch64/functions/memory/MemStore64BWithRet

// MemStore64BWithRet()
// ==============
// Performs an atomic 64-byte store to a given virtual address returning
// the status value of the operation.
bits(64) MemStore64BWithRet(bits(64) address, bits(512) value, AccType acctype)
AddressDescriptor memaddrdesc;
boolean iswrite = TRUE;
constant integer size = 64;

aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
istagaccess = HaveMTE2Ext() & AArch64.AccessIsTagChecked(address, acctype);
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);
    return ZeroExtend('1', 64);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
if HaveTME() then
    accdesc.transactional = TSTATE.depth > 0;

boolean istagchecked = istagaccess;
if istagchecked then
    bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
    if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
        AArch64.TagCheckFault(address, acctype, iswrite);
        return ZeroExtend('1', 64);

memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
if IsFault(memstatus) then
    HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
return memstatus.store64bstatus;

aarch64/functions/memory/MemStore64BWithRetStatus

// Generates the return status of memory write with ST64BV or ST64BV0
// instructions. The status indicates if the operation succeeded, failed,
// or was not supported at this memory location.
bits(64) MemStore64BWithRetStatus();

aarch64/functions/memory/NVMem

// NVMem[] - non-assignment form
// -----------------------------
// This function is the load memory access for the transformed System register read access
// when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
// The address for the load memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System register that
// supports transformation of register access to memory access.
bits(64) NVMem[integer offset]
assert offset > 0;
bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
return Mem[address, 8, AccType_NV2REGISTER];

// NVMem[] - assignment form
// -----------------------------
// This function is the store memory access for the transformed System register write access
// when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
// The address for the store memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System register that
// supports transformation of register access to memory access.

NVMem[integer offset] = bits(64) value
assert offset > 0;
  bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
  Mem[address, 8, AccType_NV2REGISTER] = value;
  return;

aarch64/functions/memory/PhysMemTagRead

// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access from the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.
(PhysMemRetStatus, bits(4)) PhysMemTagRead(AddressDescriptor desc, AccessDescriptor accdesc);

aarch64/functions/memory/PhysMemTagWrite

// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access to the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.
PhysMemRetStatus PhysMemTagWrite(AddressDescriptor desc, AccessDescriptor accdesc, bits (4) value);

aarch64/functions/memory/SetTagCheckedInstruction

// Flag the current instruction as using/not using memory tag checking.
SetTagCheckedInstruction(boolean checked);

aarch64/functions/mops/CPYPostSizeChoice

// Returns the size of the copy that is performed by the CPYE* instructions for this
// implementation given the parameters of the destination, source and size of the copy.
// Postsize is encoded as -l*size for an option A implementation if cpysize is negative.
bits(64) CPYPostSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);

aarch64/functions/mops/CPYPreSizeChoice

// Returns the size of the copy that is performed by the CPYP* instructions for this
// implementation given the parameters of the destination, source and size of the copy.
// Presize is encoded as -l*size for an option A implementation if cpysize is negative.
bits(64) CPYPreSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);
aarch64/functions/mops/CPYSizeChoice

// Returns the size of the block this performed for an iteration of the copy given the
// parameters of the destination, source and size of the copy.
integer CPYSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);

aarch64/functions/mops/CheckMOPSEnabled

// CheckMOPSEnabled()
// ==================
// Check for EL0 and EL1 access to the CPY* and SET* instructions.
CheckMOPSEnabled()
if (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
(HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0') &&
(!IsHCRXEL2Enabled() || HCRX_EL2.MSCEn == '0')) then
  UNDEFINED;
if (PSTATE_EL == EL0 && SCTLR_EL1.MSCEn == '0' &&
  (EL2Enabled() || HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0')) then
  UNDEFINED;
if PSTATE_EL == EL0 && IsInHost() && SCTLR_EL2.MSCEn == '0' then
  UNDEFINED;

aarch64/functions/mops/MOPSStage

enumeration MOPSStage { MOPSStage_Prologue, MOPSStage_Main, MOPSStage_Epilogue };

aarch64/functions/mops/MaxBlockSizeCopiedBytes

// MaxBlockSizeCopiedBytes()
// =========================
// Returns the maximum number of bytes that can used in a single block of the copy.
integer MaxBlockSizeCopiedBytes()
  return integer IMPLEMENTATION_DEFINED "Maximum bytes used in a single block of a copy";

aarch64/functions/mops/MemCpyAccessTypes

// MemCpyAccessTypes()
// ====================
// Return the read and write access types for a CPY* instruction.

(AccType, AccType) MemCpyAccessTypes(bits(4) options)
unpriv_at_el1 = PSTATE.EL == EL1 && !EL2Enabled() &&
  HaveNVExt() && HCR_EL2.<NV,NV1> == '1';
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '1';
rwpriv_at_el1 = options<1> == '1' && unpriv_at_el1;
rwpriv_at_el2 = options<1> == '1' && unpriv_at_el2;
wunpriv_at_el1 = options<0> == '1' && unpriv_at_el1;
wunpriv_at_el2 = options<0> == '1' && unpriv_at_el2;
user_access_override = HaveUAOExt() && PSTATE.UAO == '1';

AccType racctype;
if !user_access_override && (rwpriv_at_el1 || rwpriv_at_el2) then
  racctype = if options<3> == '0' then AccType_UNPRIV else AccType_UNPRIVSTREAM;
else
  racctype = if options<3> == '0' then AccType_NORMAL else AccType_STREAM;
AccType wacctype;
if !user_access_override && (wunpriv_at_el1 || wunpriv_at_el2) then
    wacctype = if options<2> == '0' then AccType_UNPRIV else AccType_UNPRIVSTREAM;
else
    wacctype = if options<2> == '0' then AccType_NORMAL else AccType_STREAM;
return (racctype, wacctype);

aarch64/functions/mops/MemCpyDirectionChoice

// Returns true if in the non-overlapping case of a memcpy of size cpysize bytes
// from the source address fromaddress to destination address toaddress is done
// in the forward direction on this implementation.
boolean MemCpyDirectionChoice(bits(64) fromaddress, bits(64) toaddress, bits(64) cpysize);

aarch64/functions/mops/MemCpyOptionA

// MemCpyOptionA()
// ===============
// Returns TRUE if the implementation uses Option A for the
// CPY*/SET* instructions, and FALSE otherwise.

boolean MemCpyOptionA()
    return boolean IMPLEMENTATION_DEFINED "CPY*/SET* instructions use Option A";

aarch64/functions/mops/MemCpyParametersIllformedE

// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYE* instruction for this implementation given the parameters of the destination,
// source and size of the copy.
boolean MemCpyParametersIllformedE(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);

aarch64/functions/mops/MemCpyParametersIllformedM

// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYM* instruction for this implementation given the parameters of the destination,
// source and size of the copy.
boolean MemCpyParametersIllformedM(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);

aarch64/functions/mops/MemCpyZeroSizeCheck

// Returns TRUE if the implementation option is checked on a copy of size zero remaining.
boolean MemCpyZeroSizeCheck();

aarch64/functions/mops/MemSetAccessType

// MemSetAccessType()
// ================
// Return the access type for a SET* instruction.

AccType MemSetAccessType(bits(2) options)
    unpriv_at_el1 = options<0> == '1' && PSTATE.EL == EL1 && !(EL2Enabled() &&
        HaveNVEExt() && HCR_EL2.<NV,NV1> == '11');
    unpriv_at_el2 = (options<0> == '1' && PSTATE.EL == EL2 &&
HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '1');

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';

AccType acctype;
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = if options<1> == '0' then AccType_UNPRIV else AccType_UNPRIVSTREAM;
else
  acctype = if options<1> == '0' then AccType_NORMAL else AccType_STREAM;

return acctype;

aarch64/functions/mops/MemSetParametersIllformedE

// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETE* or SETGE* instruction for this implementation given the
// parameters of the destination and size of the set.
boolean MemSetParametersIllformedE(bits(64) toaddress, bits(64) setsize,
                                  boolean IsSETGE);

aarch64/functions/mops/MemSetParametersIllformedM

// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETM* or SETGM* instruction for this implementation given the
// parameters of the destination and size of the copy.
boolean MemSetParametersIllformedM(bits(64) toaddress, bits(64) setsize,
                                  boolean IsSETGM);

aarch64/functions/mops/MemSetZeroSizeCheck

// Returns TRUE if the implementation option is checked on a copy of size zero remaining.
boolean MemSetZeroSizeCheck();

aarch64/functions/mops/MismatchedCpySetTargetEL

// MismatchedCpySetTargetEL()
// ==========================
// Return the target exception level for an Exception_MemCpyMemSet.

bits(2) MismatchedCpySetTargetEL()
bits(2) target_el;
if UInt(PSTATE.EL) > UInt(EL1) then
  target_el = PSTATE.EL;
elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
  target_el = EL2;
elsif (PSTATE.EL == EL1 && EL2Enabled() &&
    IsHCRXEL2Enabled() && HCRX_EL2.MCE2 == '1') then
  target_el = EL2;
else
  target_el = EL1;

return target_el;

aarch64/functions/mops/MismatchedMemCpyException

// MismatchedMemCpyException()
// ===========================
// Generates an exception for a CPY* instruction if the version
// is inconsistent with the state of the call.

MismatchedMemCpyException(boolean option_a, integer destreg, integer srcreg, integer sizereg,
    boolean wrong_option, boolean from_epilogue, bits(4) options)
bits(64) preferred_exception_return = ThisInstrAddr(64);
integer vect_offset = 0x0;
bits(2) target_el = MismatchedCpySetTargetEL();

ExceptionRecord exception = ExceptionSyndrome(Exception_MemCpyMemSet);
exception.syndrome<24> = '0';
exception.syndrome<23> = '0';
exception.syndrome<22:19> = options;
exception.syndrome<18> = if from_epilogue then '1' else '0';
exception.syndrome<17> = if wrong_option then '1' else '0';
exception.syndrome<16> = if option_a then '1' else '0';
// exception.syndrome<15> is RES0
exception.syndrome<14:10> = destreg<4:0>;
exception.syndrome<9:5> = srcreg<4:0>;
exception.syndrome<4:0> = sizereg<4:0>;
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

aarch64/functions/mops/MismatchedMemSetException

// MismatchedMemSetException()
// --------------------------
// Generates an exception for a SET* instruction if the version
// is inconsistent with the state of the call.

MismatchedMemSetException(boolean option_a, integer destreg, integer datareg, integer sizereg,
    boolean wrong_option, boolean from_epilogue, bits(2) options,
    boolean is_SETG)
bits(64) preferred_exception_return = ThisInstrAddr(64);
integer vect_offset = 0x0;
bits(2) target_el = MismatchedCpySetTargetEL();

ExceptionRecord exception = ExceptionSyndrome(Exception_MemCpyMemSet);
exception.syndrome<24> = '1';
exception.syndrome<23> = if is_SETG then '1' else '0';
// exception.syndrome<22:21> is RES0
exception.syndrome<20:19> = options;
exception.syndrome<18> = if from_epilogue then '1' else '0';
exception.syndrome<17> = if wrong_option then '1' else '0';
exception.syndrome<16> = if option_a then '1' else '0';
// exception.syndrome<15> is RES0
exception.syndrome<14:10> = destreg<4:0>;
exception.syndrome<9:5> = datareg<4:0>;
exception.syndrome<4:0> = sizereg<4:0>;
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

aarch64/functions/mops/SETPostSizeChoice

// Returns the size of the set that is performed by the SETE* or SETGE* instructions
// for this implementation, given the parameters of the destination and size of the set.
// Postsize is encoded as -size for an option A implementation if setsize is negative.
bits(64) SETPostSizeChoice(bits(64) toaddress, bits(64) setsize, boolean IsSETGE);
### aarch64/functions/mops/SETPreSizeChoice

// Returns the size of the set that is performed by the SETP* or SETGP* instructions
// for this implementation, given the parameters of the destination and size of the set.
// Presize is encoded as -size for an option A implementation if setsize is negative.
bits(64) SETPreSizeChoice(bits(64) toaddress, bits(64) setsize, boolean IsSETGP);

### aarch64/functions/mops/SETSizeChoice

// Returns the size of the block that is performed for an iteration of the set given
// the parameters of the destination and size of the set. The size of the block
// is an integer multiple of AlignSize.
integer SETSizeChoice(bits(64) toaddress, bits(64) setsize, integer AlignSize);

### aarch64/functions/pac/addpac/AddPAC

// AddPAC()
// =========
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.
bits(64) AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
  bits(64) PAC;
  bits(64) result;
  bits(64) ext_ptr;
  bits(64) extfield;
  bit selbit;
  boolean auth = TRUE;
  boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
  integer top_bit = if tbi then 55 else 63;

  // If tagged pointers are in use for a regime with two TTBRs, use bit<55> of
  // the pointer to select between upper and lower ranges, and preserve this.
  // This handles the awkward case where there is apparently no correct choice between
  // the upper and lower address range - ie an addr of 1xxxxxxx0... with TBI0=0 and TBI1=1
  // and 0xxxxxxx1 with TBI1=0 and TBI0=1:
  if PtrHasUpperAndLowerAddRanges() then
    assert S1TranslationRegime() IN {EL1, EL2};
    if S1TranslationRegime() == EL1 then
      // EL1 translation regime registers
      if data then
        if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then
          selbit = ptr<55>;
        else
          selbit = ptr<63>;
        else
          if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
            (TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then
            selbit = ptr<55>;
          else
            selbit = ptr<63>;
      else
        // EL2 translation regime registers
        if data then
          if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then
            selbit = ptr<55>;
          else
            selbit = ptr<63>;
        else
          if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
            (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then
            selbit = ptr<55>;
          else
            selbit = ptr<63>;
  else
    // If not using tagged pointers
    selbit = ptr<63>;
else selbit = if tbi then ptr<55> else ptr<63>;
if HaveEnhancedPAC2() && ConstPACField() then selbit = ptr<55>;
integer bottom_PAC_bit = CalculateBottomPACBit(selbit);

extfield = Replicate(selbit, 64);

// Compute the pointer authentication code for a ptr with good extension bits
if tbi then
    ext_ptr = (ptr<63:56> : extfield<(56-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>);
else
    ext_ptr = extfield<(64-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>;

PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>, auth);

// Check if the ptr has good extension bits and corrupt the pointer authentication code if not
if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:bottom_PAC_bit>) then
    if HaveEnhancedPAC2() then
        PAC = 0x0000000000000000<63:0>;
    elsif !HaveEnhancedPAC2() then
        PAC<top_bit-1> = NOT(PAC<top_bit-1>);
    // Preserve the determination between upper and lower address at bit<55> and insert PAC into
    // bits that are not used for the address or the tag(s).
    if !HaveEnhancedPAC2() then
        if tbi then
            result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
        else
            result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
        else
            if tbi then
                result = (ptr<63:56> : selbit :
                    (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
                    ptr<bottom_PAC_bit-1:0>);
            else
                result = ((ptr<63:56> EOR PAC<63:56>) : selbit :
                    (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
                    ptr<bottom_PAC_bit-1:0>);
            return result;

aarch64/functions/pac/addpacda/AddPACDA

// AddPACDA()
// =========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APDAKey_EL1.

bits(64) AddPACDA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDAKey_EL1;
    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
            when EL1
                Enable = SCTLR_EL1.EnDA;
TrapEL2 = El2Enabled() && HCR_EL2.API == '0';
TrapEL3 = HaveEl(EL3) && SCR_EL3.API == '0';

when EL2
Enable = SCTLR_EL2.EnDA;
TrapEL2 = FALSE;
TrapEL3 = HaveEl(EL3) && SCR_EL3.API == '0';

when EL3
Enable = SCTLR_EL3.EnDA;
TrapEL2 = FALSE;
TrapEL3 = FALSE;

if Enable == '0' then
  return x;
elsif TrapEL3 && EL3SDDTrapPriority() then
  UNDEFINED;
elsif TrapEL2 then
  TrapPACUse(EL2);
elsif TrapEL3 then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    TrapPACUse(EL3);
  else
    return AddPAC(x, y, APDAKey_EL1, TRUE);
end if;

aarch64/functions/pac/addpacdb/AddPACDB

// AddPACDB()
// =========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APDBKey_EL1.

bits(64) AddPACDB(bits(64) x, bits(64) y)
  boolean TrapEL2;
  boolean TrapEL3;
  bits(1) Enable;
  bits(128) APDBKey_EL1;

  APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;

  case PSTATE.EL of
    when EL0
      boolean IsEL1Regime = S1TranslationRegime() == EL1;
      Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
      TrapEL2 = (El2Enabled() && HCR_EL2.API == '0' &&
                 (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
      TrapEL3 = HaveEl(EL3) && SCR_EL3.API == '0';
    when EL1
      Enable = SCTLR_EL1.EnDB;
      TrapEL2 = El2Enabled() && HCR_EL2.API == '0';
      TrapEL3 = HaveEl(EL3) && SCR_EL3.API == '0';
    when EL2
      Enable = SCTLR_EL2.EnDB;
      TrapEL2 = FALSE;
      TrapEL3 = HaveEl(EL3) && SCR_EL3.API == '0';
    when EL3
      Enable = SCTLR_EL3.EnDB;
      TrapEL2 = FALSE;
      TrapEL3 = FALSE;

    if Enable == '0' then
      return x;
    elsif TrapEL3 && EL3SDDTrapPriority() then
      UNDEFINED;
    elsif TrapEL2 then

    end if;
  end case;

TrapPACUse(EL2);
elsif TrapEL3 then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        TrapPACUse(EL3);
    else
        return AddPAC(x, y, APDBKey_EL1, TRUE);
end if;

aarch64/functions/pac/addpacga/AddPACGA

// AddPACGA()
// =========
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of x, y and the APGAKey_EL1.

bits(64) AddPACGA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(128) APGAKey_EL1;
    boolean auth = FALSE;
    APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
        when EL3
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
    end case;
    if TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
        end if;
    else
        return ComputePAC(x, y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0>, auth)<63:32>:Zeros(32);
    end if;

aarch64/functions/pac/addpacia/AddPACIA

// AddPACIA()
// =========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APIAKey_EL1.

bits(64) AddPACIA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
bits(128) APIAKey_EL1;

APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;

case PSTATE.EL of
  when EL0
    boolean IsEL1Regime = S1TranslationRegime() == EL1;
    Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
    TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
          (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL1
    Enable = SCTLR_EL1.EnIA;
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL2
    Enable = SCTLR_EL2.EnIA;
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL3
    Enable = SCTLR_EL3.EnIA;
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;
if Enable == '0' then
  return x;
elsif TrapEL3 && EL3SDDTrapPriority() then
  UNDEFINED;
elsif TrapEL2 then
  TrapPACUse(EL2);
elsif TrapEL3 then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    TrapPACUse(EL3);
  else
    return AddPAC(x, y, APIAKey_EL1, FALSE);

aarch64/functions/pac/addpacib/AddPACIB

// AddPACIB()
// =========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APIBKey_EL1.

bits(64) AddPACIB(bits(64) x, bits(64) y)
  boolean TrapEL2;
  boolean TrapEL3;
  bits(1) Enable;
  bits(128) APIBKey_EL1;

APIBKey_EL1 = APIBKeyHi_EL1<63:0>:APIBKeyLo_EL1<63:0>;

case PSTATE.EL of
  when EL0
    boolean IsEL1Regime = S1TranslationRegime() == EL1;
    Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
    TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
          (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL1
    Enable = SCTLR_EL1.EnIB;
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL2
    Enable = SCTLR_EL2.EnIB;
TrapEL2 = FALSE;
TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
when EL3
   Enable = SCTLR_EL3.EnIB;
TrapEL2 = FALSE;
TrapEL3 = FALSE;
if Enable == '0' then
    return x;
elsif TrapEL3 && EL3SDDTrapPriority() then
    UNDEFINED;
elsif TrapEL2 then
    TrapPACUse(EL2);
elsif TrapEL3 then
    if Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
    else
        TrapPACUse(EL3);
    else
        return AddPAC(x, y, APIBKey_EL1, FALSE);
aarch64/functions/pac/auth/AArch64.PACFailException

// AArch64.PACFailException()
//=----------------------------------
// Generates a PAC Fail Exception
AArch64.PACFailException(bits(2) syndrome)
route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x0;
exception = ExceptionSyndrome(Exception_PACFail);
exception.syndrome<1:0>   = syndrome;
exception.syndrome<24:2>  = Zeros(23);                // RES0
if UInt(PSTATE.EL) > UInt(EL0) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elsif route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
aarch64/functions/pac/auth/Auth

// Auth()
//=------
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.
bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit key_number,
    boolean is_combined)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;
    boolean auth = TRUE;

    // Reconstruct the extension field used of adding the PAC to the pointer
    boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
integer bottom_PAC_bit = CalculateBottomPACBit(ptr<55>);
extfield = Replicate(ptr<55>, 64);

if tbi then
    original_ptr = (ptr<63:56> : extfield<(56-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>);
else
    original_ptr = extfield<(64-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>;

PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>, auth);

// Check pointer authentication code
if tbi then
    if !HaveEnhancedPAC2() then
        if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
            result = original_ptr;
        else
            error_code = key_number:NOT(key_number);
            result = original_ptr<63:55>:error_code:original_ptr<52:0>;
    else
        result = ptr;
        result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
        if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
            if result<54:bottom_PAC_bit> != Replicate(result<55>, (55-bottom_PAC_bit)) then
                error_code = (if data then '1' else '0'):key_number;
                AArch64.PACFailException(error_code);
            else
                if !HaveEnhancedPAC2() then
                    if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> && PAC<63:56> == ptr<63:56> then
                        result = original_ptr;
                    else
                        error_code = key_number:NOT(key_number);
                        result = original_ptr<63>:error_code:original_ptr<60:0>;
                else
                    result = ptr;
                    result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
                    result<63:56> = result<63:56> EOR PAC<63:56>;
                    if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                        if result<63:bottom_PAC_bit> != Replicate(result<55>, (64-bottom_PAC_bit)) then
                            error_code = (if data then '1' else '0'):key_number;
                            AArch64.PACFailException(error_code);
                        else
                            return result;
    else
        if !HaveEnhancedPAC2() then
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63>:error_code:original_ptr<60:0>;
        else
            result = ptr;
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            result<63:56> = result<63:56> EOR PAC<63:56>;
            if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                if result<63:bottom_PAC_bit> != Replicate(result<55>, (64-bottom_PAC_bit)) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
                else
                    return result;

aarch64/functions/pac/authda/AuthDA

// AuthDA()
// =========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACDA().

bits(64) AuthDA(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDAKey_EL1;
    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
Enable = SCTLR_EL1.EnDA;
TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';

when EL2
Enable = SCTLR_EL2.EnDA;
TrapEL2 = FALSE;
TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';

when EL3
Enable = SCTLR_EL3.EnDA;
TrapEL2 = FALSE;
TrapEL3 = FALSE;

if Enable == '0' then
  return x;
elsif TrapEL3 && EL3SDDTrapPriority() then
  UNDEFINED;
elsif TrapEL2 then
  TrapPACUse(EL2);
elsif TrapEL3 then
  if Halted() && EDSCR.SDD == '1' then
    UNDEFINED;
  else
    TrapPACUse(EL3);
  else
    return Auth(x, y, APDBKey_EL1, TRUE, '0', is_combined);
end if;

aarch64/functions/pac/authdb/AuthDB

// AuthDB()
// =========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of x, using
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) x, bits(64) y, boolean is_combined)
  boolean TrapEL2;
  boolean TrapEL3;
  bits(1)  Enable;
  bits(128) APDBKey_EL1;

  APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
  case PSTATE.EL of
    when EL0
      boolean IsEL1Regime = S1TranslationRegime() == EL1;
      Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
      TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
      (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL1
      Enable = SCTLR_EL1.EnDB;
      TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL2
      Enable = SCTLR_EL2.EnDB;
      TrapEL2 = FALSE;
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL3
      Enable = SCTLR_EL3.EnDB;
      TrapEL2 = FALSE;
      TrapEL3 = FALSE;
    if Enable == '0' then
      return x;
    elsif TrapEL3 && EL3SDDTrapPriority() then
      UNDEFINED;
  end case;
elsif TrapEL2 then
    TrapPACUse(EL2);
elsif TrapEL3 then
    if Halted() \&\& EDSCR.SDD == '1' then
        UNDEFINED;
    else
        TrapPACUse(EL3);
    else
        return Auth(x, y, APDBKey_EL1, TRUE, '1', is_combined);
end if;

aarch64/functions/pac/authia/AuthIA

// AuthIA()
// =========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIA().

bits(64) AuthIA(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
            TrapEL2 = (EL2Enabled() \&\& HCR_EL2.API == '0' \&\&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) \&\& SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIA;
            TrapEL2 = EL2Enabled() \&\& HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) \&\& SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) \&\& SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
        if Enable == '0' then
            return x;
        elsif TrapEL3 \&\& EL3SDDTrapPriority() then
            UNDEFINED;
        elsif TrapEL2 then
            TrapPACUse(EL2);
        elsif TrapEL3 then
            if Halted() \&\& EDSCR.SDD == '1' then
                UNDEFINED;
            else
                TrapPACUse(EL3);
            end if;
        else
            return Auth(x, y, APIAKey_EL1, FALSE, '0', is_combined);
        end if;
    end case;
aarch64/functions/pac/authib/AuthIB

// AuthIB()
// =========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithms and key as AddPACIB().

bits(64) AuthIB(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
    end;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
        end;
    else
        return Auth(x, y, APIBKey_EL1, FALSE, '1', is_combined);
    end;

aarch64/functions/pac/calcbottompacbit/CalculateBottomPACBit

// CalculateBottomPACBit()
// =======================

integer CalculateBottomPACBit(bit top_bit)
    integer tsz_field;
    boolean using64k;
    Constraint c;

    if PterHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            tsz_field = if top_bit == '1' then UInt(TCR_EL1.T1SZ) else UInt(TCR_EL1.T0SZ);
            using64k = if top_bit == '1' then TCR_EL1.T1SZ == '1' else TCR_EL1.T0SZ == '0';
        else
            // EL2 translation regime registers
            tsz_field = if top_bit == '1' then UInt(TCR_EL2.T1SZ) else UInt(TCR_EL2.T0SZ);
            using64k = if top_bit == '1' then TCR_EL2.T1SZ == '1' else TCR_EL2.T0SZ == '0';
        end;
    else
        // EL3 translation regime registers
        tsz_field = if top_bit == '1' then UInt(TCR_EL3.T1SZ) else UInt(TCR_EL3.T0SZ);
        using64k = if top_bit == '1' then TCR_EL3.T1SZ == '1' else TCR_EL3.T0SZ == '0';
    end;


else
    // EL2 translation regime registers
    assert HaveEL(EL2);
    tsz_field = if top_bit == '1' then UInt(TCR_EL2.T1SZ) else UInt(TCR_EL2.T0SZ);
    using64k = if top_bit == '1' then TCR_EL2.TG1 == '1' else TCR_EL2.TG0 == '0';
else
    tsz_field = if PSTATE.EL == EL2 then UInt(TCR_EL3.T0SZ) else UInt(TCR_EL3.T0SZ);
    using64k = if PSTATE.EL == EL2 then TCR_EL2.TG0 == '0' else TCR_EL3.TG0 == '0';

max_limit_tsz_field = (if !HaveSmallTranslationTableExt() then 39 else if using64k then 47 else 48);
if tsz_field > max_limit_tsz_field then
    // TCR_ELx.TySZ is out of range
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_NONE};
    if c == Constraint_FORCE then tsz_field = max_limit_tsz_field;

if tsz_field < tszmin then
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_NONE};
    if c == Constraint_FORCE then tsz_field = tszmin;

return (64-tsz_field);

aarch64/functions/pac/computepac/ComputePAC

// ComputePAC()
// ============

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1, boolean auth)
if UsePACIMP(auth) then
    return ComputePACIMPDEF(data, modifier, key0, key1);
if UsePACQARMA3(auth) then
    boolean isqarma3 = TRUE;
    return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
if UsePACQARMA5(auth) then
    boolean isqarma3 = FALSE;
    return ComputePACQARMA(data, modifier, key0, key1, isqarma3);

aarch64/functions/pac/computepac/ComputePACIMPDEF

// Compute IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.
bits(64) ComputePACIMPDEF(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1);

aarch64/functions/pac/computepac/ComputePACQARMA

// ComputePACQARMA()
// ================

// Compute QARMA3 or QARMA5 cryptographic algorithm for PAC calculation
bits(64) ComputePACQARMA(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1, boolean isqarma3)
    bits(64) workingval;
    bits(64) runningmod;
    bits(64) roundkey;
    bits(64) modk0;
    constant bits(64) Alpha = 0x0AC29B7C97C50DD<63:0>;

    integer iterations;
    RC[0] = 0x0000000000000000<63:0>;
    RC[1] = 0x13198A2E03707344<63:0>;
    RC[2] = 0xA4093822299F310D<63:0>;
    if isqarma3 then
        iterations = 2;
else // QARMA5
iterations = 4;
RC[3] = 0x082EFA98EC4E6C89<63:0>;
RC[4] = 0x452821E638D01377<63:0>;

modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
runningmod = modifier;
workingval = data EOR key0;

for i = 0 to iterations
    roundkey = key1 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = workingval EOR RC[i];
    if i > 0 then
        workingval = PACCellShuffle(workingval);
        workingval = PACMult(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACSub(workingval);
    runningmod = TweakShuffle(runningmod<63:0>);
    roundkey = modk0 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACSub(workingval);
    roundkey = key1 EOR workingval;
    workingval = workingval EOR key1;
    workingval = PACCellInvShuffle(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACInvSub(workingval);
    if i < iterations then
        workingval = PACMult(workingval);
    runningmod = TweakInvShuffle(runningmod<63:0>);
    roundkey = key1 EOR runningmod;
    workingval = workingval EOR RC[iterations-i];
    workingval = workingval EOR roundkey;
    workingval = workingval EOR Alpha;
    workingval = workingval EOR modk0;

return workingval;

aarch64/functions/pac/computepac/PACCellInvShuffle

  // PACCellInvShuffle()
  // ===============

  bits(64) PACCellInvShuffle(bits(64) indata)
  bits(64) outdata;
  outdata<3:0> = indata<15:12>;
outdata<7:4> = indata<27:24>; 
outdata<11:8> = indata<51:48>; 
outdata<15:12> = indata<39:36>; 
outdata<19:16> = indata<59:56>; 
outdata<23:20> = indata<47:44>; 
outdata<27:24> = indata<7:4>; 
outdata<31:28> = indata<19:16>; 
outdata<35:32> = indata<35:32>; 
outdata<39:36> = indata<15:12>; 
outdata<43:40> = indata<23:20>; 
outdata<47:44> = indata<39:36>; 
outdata<51:48> = indata<19:16>; 
outdata<55:52> = indata<3:0>; 
outdata<59:56> = indata<43:40>; 
outdata<63:60> = indata<63:60>; 
return outdata;

aarch64/functions/pac/computepac/PACCellShuffle

// PACCellShuffle()
// ================
bits(64) PACCellShuffle(bits(64) indata)
bits(64) outdata;
outdata<3:0> = indata<55:52>;
outdata<7:4> = indata<27:24>;
outdata<11:8> = indata<47:44>;
outdata<15:12> = indata<39:36>;
outdata<19:16> = indata<59:56>;
outdata<23:20> = indata<47:44>;
outdata<27:24> = indata<7:4>;
outdata<31:28> = indata<43:40>;
outdata<35:32> = indata<35:32>;
outdata<39:36> = indata<15:12>;
outdata<43:40> = indata<59:56>;
outdata<47:44> = indata<23:20>;
outdata<51:48> = indata<11:8>;
outdata<55:52> = indata<39:36>;
outdata<59:56> = indata<19:16>;
outdata<63:60> = indata<63:60>;
return outdata;

aarch64/functions/pac/computepac/PACInvSub

// PACInvSub()
// ===========
bits(64) PACInvSub(bits(64) Tinput)
// This is a 4-bit substitution from the PRINCE-family cipher
bits(64) Toutput;
for i = 0 to 15
    case Tinput<4*i+3:4*i> of
        when '0000' Toutput<4*i+3:4*i> = '0101';
        when '0001' Toutput<4*i+3:4*i> = '1110';
        when '0010' Toutput<4*i+3:4*i> = '1101';
        when '0011' Toutput<4*i+3:4*i> = '1000';
        when '0100' Toutput<4*i+3:4*i> = '1010';
        when '0101' Toutput<4*i+3:4*i> = '1011';
        when '0110' Toutput<4*i+3:4*i> = '0001';
        when '0111' Toutput<4*i+3:4*i> = '1001';
        when '1000' Toutput<4*i+3:4*i> = '0010';
        when '1001' Toutput<4*i+3:4*i> = '0110';
        when '1010' Toutput<4*i+3:4*i> = '1111';
        when '1011' Toutput<4*i+3:4*i> = '0000';
When '1100' \( T_{\text{output}}<4*i+3:4*i> = '0100'; \)
when '1101' \( T_{\text{output}}<4*i+3:4*i> = '1100'; \)
when '1110' \( T_{\text{output}}<4*i+3:4*i> = '0111'; \)
when '1111' \( T_{\text{output}}<4*i+3:4*i> = '0011'; \)
return \( T_{\text{output}}; \)

```
// PACMult()
// =========

bits(64) PACMult(bits(64) Sinput)
    bits(4) t0;
    bits(4) t1;
    bits(4) t2;
    bits(4) t3;
    bits(64) Soutput;
    for i = 0 to 3
        t0<3:0> = RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 2);
        t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
        t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
        t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        Soutput<4*i+3:4*i> = t3<3:0>;
        Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
        Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
        Soutput<4*(i+12)+3:4*(i+12)> = t0<3:0>;
    return Soutput;
```

```
// PACSub()
// ========

bits(64) PACSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000' \( T_{\text{output}}<4*i+3:4*i> = '1011'; \)
            when '0001' \( T_{\text{output}}<4*i+3:4*i> = '0110'; \)
            when '0010' \( T_{\text{output}}<4*i+3:4*i> = '1000'; \)
            when '0011' \( T_{\text{output}}<4*i+3:4*i> = '1111'; \)
            when '0100' \( T_{\text{output}}<4*i+3:4*i> = '1100'; \)
            when '0101' \( T_{\text{output}}<4*i+3:4*i> = '0011'; \)
            when '0110' \( T_{\text{output}}<4*i+3:4*i> = '0101'; \)
            when '0111' \( T_{\text{output}}<4*i+3:4*i> = '1010'; \)
            when '1000' \( T_{\text{output}}<4*i+3:4*i> = '0001'; \)
            when '1001' \( T_{\text{output}}<4*i+3:4*i> = '0111'; \)
            when '1010' \( T_{\text{output}}<4*i+3:4*i> = '0100'; \)
            when '1011' \( T_{\text{output}}<4*i+3:4*i> = '0010'; \)
            when '1100' \( T_{\text{output}}<4*i+3:4*i> = '0110'; \)
            when '1101' \( T_{\text{output}}<4*i+3:4*i> = '0001'; \)
            when '1110' \( T_{\text{output}}<4*i+3:4*i> = '0011'; \)
            when '1111' \( T_{\text{output}}<4*i+3:4*i> = '1010'; \)
        return Toutput;
```
aarch64/functions/pac/computepac/PacSub1

// PacSub1()
// =========

bits(64) PacSub1(bits(64) Tinput)
// This is a 4-bit substitution from Qarma sigmal
bits(64) Toutput;
for i = 0 to 15
  case Tinput<4*i+3:4*i> of
    when '0000' Toutput<4*i+3:4*i> = '1010';
    when '0001' Toutput<4*i+3:4*i> = '1101';
    when '0010' Toutput<4*i+3:4*i> = '1110';
    when '0011' Toutput<4*i+3:4*i> = '0110';
    when '0100' Toutput<4*i+3:4*i> = '1111';
    when '0101' Toutput<4*i+3:4*i> = '0111';
    when '0110' Toutput<4*i+3:4*i> = '0011';
    when '0111' Toutput<4*i+3:4*i> = '0101';
    when '1000' Toutput<4*i+3:4*i> = '1000';
    when '1001' Toutput<4*i+3:4*i> = '1000';
    when '1010' Toutput<4*i+3:4*i> = '0000';
    when '1011' Toutput<4*i+3:4*i> = '1100';
    when '1100' Toutput<4*i+3:4*i> = '1011';
    when '1101' Toutput<4*i+3:4*i> = '0001';
    when '1110' Toutput<4*i+3:4*i> = '0010';
    when '1111' Toutput<4*i+3:4*i> = '0100';
  return Toutput;

aarch64/functions/pac/computepac/RC

array bits(64) RC[0..4];

aarch64/functions/pac/computepac/RotCell

// RotCell()
// =========

bits(4) RotCell(bits(4) incell, integer amount)
bits(8) tmp;
bits(4) outcell;
// assert amount>3 || amount<1;
tmp<7:0> = incell<3:0>:incell<3:0>;
outcell = tmp<7:-amount:4-amount>;
return outcell;

aarch64/functions/pac/computepac/TweakCellInvRot

// TweakCellInvRot()
// ===============

bits(4) TweakCellInvRot(bits(4) incell)
bits(4) outcell;
outcell<3> = incell<2>;
outcell<2> = incell<1>;
outcell<1> = incell<0>;
outcell<0> = incell<0> EOR incell<3>;
return outcell;
aarch64/functions/pac/computepac/TweakCellRot

// TweakCellRot()
// ==============

bits(4) TweakCellRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
    return outcell;

aarch64/functions/pac/computepac/TweakInvShuffle

// TweakInvShuffle()
// ================

bits(64) TweakInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = TweakCellInvRot(indata<51:48>);
    outdata<7:4> = indata<55:52>;
    outdata<11:8> = indata<23:20>;
    outdata<15:12> = indata<27:24>;
    outdata<19:16> = indata<3:0>;
    outdata<23:20> = indata<7:4>;
    outdata<27:24> = TweakCellInvRot(indata<11:8>);
    outdata<31:28> = indata<15:12>;
    outdata<35:32> = TweakCellInvRot(indata<31:28>);
    outdata<39:36> = TweakCellInvRot(indata<63:60>);
    outdata<43:40> = TweakCellInvRot(indata<59:56>);
    outdata<47:44> = TweakCellInvRot(indata<47:44>);
    outdata<51:48> = indata<35:32>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = TweakCellInvRot(indata<47:44>);
    return outdata;

aarch64/functions/pac/computepac/TweakShuffle

// TweakShuffle()
// ==============

bits(64) TweakShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<19:16>;
    outdata<7:4> = indata<23:20>;
    outdata<11:8> = TweakCellRot(indata<27:24>);
    outdata<15:12> = indata<31:28>;
    outdata<19:16> = TweakCellRot(indata<47:44>);
    outdata<23:20> = indata<11:8>;
    outdata<27:24> = indata<15:12>;
    outdata<31:28> = TweakCellRot(indata<35:32>);
    outdata<35:32> = indata<51:48>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<39:36>;
    outdata<47:44> = TweakCellRot(indata<63:60>);
    outdata<51:48> = TweakCellRot(indata<3:0>);
    outdata<55:52> = indata<7:4>;
    outdata<59:56> = TweakCellRot(indata<43:40>);
    outdata<63:60> = TweakCellRot(indata<39:36>);
    return outdata;
aarch64/functions/pac/computePac/UsePACIMP

// UsePACIMP()
// ===========
// Checks whether IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

boolean UsePACIMP(boolean auth)
    return if auth then HavePACIMPAuth() else HavePACIMPGeneric();

aarch64/functions/pac/computePac/UsePACQARMA3

// UsePACQARMA3()
// ==============
// Checks whether QARMA3 cryptographic algorithm to be used for PAC calculation.

boolean UsePACQARMA3(boolean auth)
    return if auth then HavePACQARMA3Auth() else HavePACQARMA3Generic();

aarch64/functions/pac/computePac/UsePACQARMA5

// UsePACQARMA5()
// ===============
// Checks whether QARMAS cryptographic algorithm to be used for PAC calculation.

boolean UsePACQARMA5(boolean auth)
    return if auth then HavePACQARMA5Auth() else HavePACQARMA5Generic();

aarch64/functions/pac/pac/ConstPACField

// ConstPACField()
// ===============
// Returns TRUE if bit<55> can be used to determine the size of the PAC field, FALSE otherwise.

boolean ConstPACField()
    return (HaveEnhancedPAC2() &&
            boolean IMPLEMENTATION_DEFINED "Bit 55 determines the size of the PAC field");

aarch64/functions/pac/pac/HaveEnhancedPAC

// HaveEnhancedPAC()
// ===============
// Returns TRUE if support for EnhancedPAC is implemented, FALSE otherwise.

boolean HaveEnhancedPAC()
    return (HavePACExt() &&
            boolean IMPLEMENTATION_DEFINED "Has enhanced PAC functionality");

aarch64/functions/pac/pac/HaveEnhancedPAC2

// HaveEnhancedPAC2()
// ===============
// Returns TRUE if support for EnhancedPAC2 is implemented, FALSE otherwise.

boolean HaveEnhancedPAC2()
    return HasArchVersion(ARMv8p6) || (HasArchVersion(ARMv8p3)
            && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC 2 functionality");
aarch64/functions/pac/pac/HaveFPAC

    // HaveFPAC()
    // =========
    // Returns TRUE if support for FPAC is implemented, FALSE otherwise.

    boolean HaveFPAC()
    
    return HaveEnhancedPAC2() && boolean IMPLEMENTATION_DEFINED "Has FPAC functionality";

aarch64/functions/pac/pac/HaveFPACCombined

    // HaveFPACCombined()
    // =================
    // Returns TRUE if support for FPACCombined is implemented, FALSE otherwise.

    boolean HaveFPACCombined()
    
    return HaveFPAC() && boolean IMPLEMENTATION_DEFINED "Has FPAC Combined functionality";

aarch64/functions/pac/pac/HavePACExt

    // HavePACExt()
    // ===========
    // Returns TRUE if support for the PAC extension is implemented, FALSE otherwise.

    boolean HavePACExt()
    
    return HasArchVersion(ARMv8p3);

aarch64/functions/pac/pac/HavePACIMPAuth

    // HavePACIMPAuth()
    // ================
    // Returns TRUE if support for PAC IMP Auth is implemented, FALSE otherwise.

    boolean HavePACIMPAuth()
    
    return HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has PAC IMPDEF Auth functionality";

aarch64/functions/pac/pac/HavePACIMPGeneric

    // HavePACIMPGeneric()
    // ===================
    // Returns TRUE if support for PAC IMP Generic is implemented, FALSE otherwise.

    boolean HavePACIMPGeneric()
    
    return HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has PAC IMPDEF Generic functionality";

aarch64/functions/pac/pac/HavePACQARMA3Auth

    // HavePACQARMA3Auth()
    // ===================
    // Returns TRUE if support for PAC QARMA3 Auth is implemented, FALSE otherwise.

    boolean HavePACQARMA3Auth()
    
    return HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has PAC QARMA3 Auth functionality";
aarch64/functions/pac/pac/HavePACQARMA3Generic

// HavePACQARMA3Generic()
// ===============
// Returns TRUE if support for PAC QARMA3 Generic is implemented, FALSE otherwise.

boolean HavePACQARMA3Generic()
    return HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has PAC QARMA3 Generic functionality";

aarch64/functions/pac/pac/HavePACQARMA5Auth

// HavePACQARMA5Auth()
// ===============
// Returns TRUE if support for PAC QARMA5 Auth is implemented, FALSE otherwise.

boolean HavePACQARMA5Auth()
    return HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has PAC QARMA5 Auth functionality"

aarch64/functions/pac/pac/HavePACQARMA5Generic

// HavePACQARMA5Generic()
// ===============
// Returns TRUE if support for PAC QARMA5 Generic is implemented, FALSE otherwise.

boolean HavePACQARMA5Generic()
    return HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has PAC QARMA5 Generic functionality"

aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges

// PtrHasUpperAndLowerAddRanges()
// ===============
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise.

boolean PtrHasUpperAndLowerAddRanges()
    regime = TranslationRegime(PSTATE.EL, AccType_NORMAL);
    return HasUnprivileged(regime);

aarch64/functions/pac/strip/Strip

// Strip()
// =======
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.

bits(64) Strip(bits(64) A, boolean data)
    bits(64) original_ptr;
    bits(64) extfield;
    boolean tbi = EffectiveTBI(A, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(A<55>);
    extfield = Replicate(A<55>, 64);
    if tbi then
        original_ptr = (A<63:56> : extfield<(56-bottom_PAC_bit)-1:0> : A<bottom_PAC_bit-1:0>);
    else
        original_ptr = extfield<(64-bottom_PAC_bit)-1:0> : A<bottom_PAC_bit-1:0>;
    return original_ptr;
aarch64/functions/pac/trappacuse/TrapPACUse

// TrapPACUse()
// ============
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

TrapPACUse(bits(2) target_el)
    assert HaveEL(target_el) & target_el != EL0 & UInt(target_el) >= UInt(PSTATE.EL);

bits(64) preferred_exception_return = ThisInstrAddr(64);
ExceptionRecord exception;
vect_offset = 0;
exception = ExceptionSyndrome(Exception_PACTrap);
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

aarch64/functions/ras/AArch64.ESBOperation

// AArch64.ESBOperation()
// ===============
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64

AArch64.ESBOperation()
    boolean mask_active;

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (EL2Enabled() &&
        (HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1'));

    target = if route_to_el3 then EL3 elsif route_to_el2 then EL2 else EL1;
    if target == EL1 then
        mask_active = PSTATE.EL IN {EL0, EL1};
    elsif HaveVirtHostExt() && target == EL2 && HCR_EL2.<E2H,TGE> == '11' then
        mask_active = PSTATE.EL IN {EL0, EL2};
    else
        mask_active = PSTATE.EL == target;

    mask_set = (PSTATE.A == '1' && (!HaveDoubleFaultExt() || SCR_EL3.EA == '0' ||
        PSTATE.EL != EL3 || SCR_EL3.NMEA == '0'));

    intdis = Halted() || ExternalDebugInterruptsDisabled(target);

    masked = (UInt(target) < UInt(PSTATE.EL)) || intdis || (mask_active & mask_set);

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        // This function might be called for an interworking case, and INTdis is masking
        // the SError interrupt.
        if ELUsingAArch32(S1TranslationRegime()) then
            syndrome32 = AArch32.PhysicalSErrorSyndrome();
            DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
        else
            implicit_esb = FALSE;
            syndrome64 = AArch64.PhysicalSErrorSyndrome(implicit_esb);
            DISR_EL1 = AArch64.ReportDeferredSError(syndrome64);
            ClearPendingPhysicalSError();
            // Set ISR_EL1.A to 0

        return;
aarch64/functions/ras/AArch64.PhysicalSErrorSyndrome

// Return the SError syndrome
bits(25) AArch64.PhysicalSErrorSyndrome(boolean implicit_esb);

aarch64/functions/ras/AArch64.ReportDeferredSError

// AArch64.ReportDeferredSError()
// ===============================================
// Generate deferred SError syndrome
bits(64) AArch64.ReportDeferredSError(bits(25) syndrome)
    bits(64) target;
    target<31> = '1'; // A
    target<24> = syndrome<24>; // IDS
    target<23:0> = syndrome<23:0>; // ISS
    return target;

aarch64/functions/ras/AArch64.vESBOperation

// AArch64.vESBOperation()
// ======================
// Perform the AArch64 ESB operation for virtual SError interrupts, either for ESB
// executed in AArch64 state, or for ESB in AArch32 state with EL2 using AArch64 state
AArch64.vESBOperation()
assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
// If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
// SError interrupt might be pending
vSEI_enabled = HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
vSEI_pending = vSEI_enabled && HCR_EL2.VSE == '1';
vintdis = Halted() || ExternalDebugInterruptsDisabled(EL1);
vmasked = vintdis || PSTATE.A == '1';
// Check for a masked virtual SError pending
if vSEI_pending && vmasked then
    // This function might be called for the interworking case, and INTdis is masking
    // the virtual SError interrupt.
    if ELUsingAArch32(EL1) then
        VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
    else
        VDISR_EL2 = AArch64.ReportDeferredSError(VSESR_EL2<24:0>);
        HCR_EL2.VSE = '0'; // Clear pending virtual SError
    return;

aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers

// AArch64.MaybeZeroRegisterUppers()
// =================================
// On taking an exception to AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.
AArch64.MaybeZeroRegisterUppers()
assert UsingAArch32(); // Always called from AArch32 state before entering AArch64 state
integer first;
integer last;
boolean include_R15;
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
    first = 0; last = 14; include_R15 = FALSE;
elif PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) then
    ...
first = 0;  last = 30;  include_R15 = FALSE;
else
  first = 0;  last = 30;  include_R15 = TRUE;
for n = first to last
  if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
    _R[n]<63:32> = Zeros(32);
  return;

aarch64/functions/registers/AArch64.ResetGeneralRegisters

// AArch64.ResetGeneralRegisters()
// ==================================
AArch64.ResetGeneralRegisters()
for i = 0 to 30
  X[i, 64] = bits(64) UNKNOWN;
return;

aarch64/functions/registers/AArch64.ResetSIMDFPRegisters

// AArch64.ResetSIMDFPRegisters()
// =================================
AArch64.ResetSIMDFPRegisters()
for i = 0 to 31
  V[i, 128] = bits(128) UNKNOWN;
return;

aarch64/functions/registers/AArch64.ResetSpecialRegisters

// AArch64.ResetSpecialRegisters()
// =================================
AArch64.ResetSpecialRegisters()
// AArch64 special registers
SP_EL0 = bits(64) UNKNOWN;
SP_EL1 = bits(64) UNKNOWN;
SPSR_EL1 = bits(64) UNKNOWN;
ELR_EL1 = bits(64) UNKNOWN;
if HaveEL(EL2) then
  SP_EL2 = bits(64) UNKNOWN;
  SPSR_EL2 = bits(64) UNKNOWN;
  ELR_EL2 = bits(64) UNKNOWN;
if HaveEL(EL3) then
  SP_EL3 = bits(64) UNKNOWN;
  SPSR_EL3 = bits(64) UNKNOWN;
  ELR_EL3 = bits(64) UNKNOWN;
// AArch32 special registers that are not architecturally mapped to AArch64 registers
if HaveAArch32EL(EL1) then
  SPSR_fiq<31:0> = bits(32) UNKNOWN;
  SPSR_irq<31:0> = bits(32) UNKNOWN;
  SPSR_abt<31:0> = bits(32) UNKNOWN;
  SPSR_und<31:0> = bits(32) UNKNOWN;
// External debug special registers

DLR_EL0 = bits(64) UNKNOWN;
DSPSR_EL0 = bits(64) UNKNOWN;
return;

aarch64/functions/registers/AArch64.ResetSystemRegisters

AArch64.ResetSystemRegisters(boolean cold_reset);

aarch64/functions/registers/PC

// PC - non-assignment form
// ========================
// Read program counter.

bits(64) PC[]
  return _PC;

aarch64/functions/registers/SP

// SP[] - assignment form
// ======================
// Write to stack pointer from a 64-bit value.

SP[] = bits(64) value
  if PSTATE.SP == '0' then
    SP_EL0 = value;
  else
    case PSTATE.EL of
      when EL0  SP_EL0 = value;
      when EL1  SP_EL1 = value;
      when EL2  SP_EL2 = value;
      when EL3  SP_EL3 = value;
    return;
  end case;

// SP[] - non-assignment form
// ==========================
// Read stack pointer with slice of 64 bits.

bits(64) SP[]
  if PSTATE.SP == '0' then
    return SP_EL0;
  else
    case PSTATE.EL of
      when EL0  return SP_EL0;
      when EL1  return SP_EL1;
      when EL2  return SP_EL2;
      when EL3  return SP_EL3;
    end case;

aarch64/functions/registers/V

// V[] - assignment form
// =====================
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.

V[integer n, integer width] = bits(width) value
  assert n >= 0 && n <= 31;
  assert width IN {8,16,32,64,128};
  integer vlen = if IsSVEEnabled(PSTATE.EL) then CurrentVL else 128;
  if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
_Z[n] = ZeroExtend(value, MAX_VL);
else
   _Z[n]<vlen-1:0> = ZeroExtend(value, vlen);

// V[] - non-assignment form
// =========================
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width) V(integer n, integer width)
   assert n >= 0 && n <= 31;
   assert width IN {8,16,32,64,128};
   return _Z[n]<width-1:0>;

aarch64/functions/registers/Vpart

// Vpart[] - non-assignment form
// =============================
// Reads a 128-bit SIMD&FP register in up to two parts:
// part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the register;
// part 1 returns the top half of the bottom 64 bits or the top half of the 128-bit
// value held in the register.

bits(width) Vpart(integer n, integer part, integer width)
   assert n >= 0 && n <= 31;
   assert part IN {0, 1};
   if part == 0 then
     assert width < 128;
     return V[n, width];
   else
     assert width IN {32,64};
     bits(128) vreg = V[n, 128];
     return vreg<(width * 2)-1:width>;

// Vpart[] - assignment form
// =========================
// Writes a 128-bit SIMD&FP register in up to two parts:
// part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
// part 1 inserts a 64-bit value into the top half of the register.

Vpart(integer n, integer part, integer width) = bits(width) value
   assert n >= 0 && n <= 31;
   assert part IN {0, 1};
   if part == 0 then
     assert width < 128;
     V[n, width] = value;
   else
     assert width == 64;
     bits(64) vreg = V[n, 64];
     V[n, 128] = value<63:0> : vreg;

aarch64/functions/registers/X

// X[] - assignment form
// =====================
// Write to general-purpose register from either a 32-bit or a 64-bit value,
// where the size of the value is passed as an argument.

X(integer n, integer width) = bits(width) value
   assert n >= 0 && n <= 31;
   assert width IN {32,64};
   if n != 31 then
     _R[n] = ZeroExtend(value, 64);
   return;
// X[] - non-assignment form
// =========================
// Read from general-purpose register with an explicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n, integer width]
assert n >= 0 && n <= 31;
assert width IN {8,16,32,64};
if n != 31 then
  return _R[n]<width-1:0>;
else
  return Zeros(width);

aarch64/functions/sme/HaveEBF16

// HaveEBF16()
// =========
// Returns TRUE if the EBF16 extension is implemented, FALSE otherwise.

boolean HaveEBF16()
  return boolean IMPLEMENTATION_DEFINED "Have EBF16 extension";

aarch64/functions/sme/HaveSME

// HaveSME()
// =========
// Returns TRUE if the SME extension is implemented, FALSE otherwise.

boolean HaveSME()
  return boolean IMPLEMENTATION_DEFINED "Have SME extension";

aarch64/functions/sme/HaveSMEF64F64

// HaveSMEF64F64()
// =============
// Returns TRUE if the SMEF64F64 extension is implemented, FALSE otherwise.

boolean HaveSMEF64F64()
  return HaveSME() && boolean IMPLEMENTATION_DEFINED "Have SMEF64F64 extension";

aarch64/functions/sme/HaveSMEI16I64

// HaveSMEI16I64()
// =============
// Returns TRUE if the SMEI16I64 extension is implemented, FALSE otherwise.

boolean HaveSMEI16I64()
  return HaveSME() && boolean IMPLEMENTATION_DEFINED "Have SMEI16I64 extension";

aarch64/functions/sme/System

array bits(MAX_VL) _ZA[0..255];
aarch64/functions/sme/ZAhslice

// ZAhslice[] - non-assignment form
// ================================================

bits(width) ZAhslice[integer tile, integer esize, integer slice, integer width]
assert esize IN {8, 16, 32, 64, 128};
integer tiles = esize DIV 8;
assert tile >= 0 && tile < tiles;
integer slices = svl DIV esize;
assert slice >= 0 && slice < slices;
return ZAvector[tile + slice * tiles, width];

// ZAhslice[] - assignment form
// ================================================

ZAhslice[integer tile, integer esize, integer slice, integer width] = bits(width) value
assert esize IN {8, 16, 32, 64, 128};
integer tiles = esize DIV 8;
assert tile >= 0 && tile < tiles;
integer slices = svl DIV esize;
assert slice >= 0 && slice < slices;
ZAvector[tile + slice * tiles, width] = value;

aarch64/functions/sme/ZAslice

// ZAslice[] - non-assignment form
// ================================================

bits(width) ZAslice[integer tile, integer esize, boolean vertical, integer slice, integer width]
bits(width) result;
if vertical then
  result = ZAvslice[tile, esize, slice, width];
else
  result = ZAhslice[tile, esize, slice, width];
return result;

// ZAslice[] - assignment form
// ================================================

ZAslice[integer tile, integer esize, boolean vertical, integer slice, integer width] = bits(width) value
if vertical then
  ZAvslice[tile, esize, slice, width] = value;
else
  ZAhslice[tile, esize, slice, width] = value;

aarch64/functions/sme/ZAtile

// ZAtile[] - non-assignment form
// ================================================

bits(width) ZAtile[integer tile, integer esize, integer width]
constant integer svl = SVL;
integer slices = svl DIV esize;
assert width == svl * slices;
bits(width) result;
for slice = 0 to slices-1
  Elem[result, slice, svl] = ZAhslice[tile, esize, slice, svl];
return result;

// ZAtile[] - assignment form
// =========================
ZAtile[integer tile, integer esize, integer width] = bits(width) value
constant integer svl = SVL;
integer slices = svl DIV esize;
assert width == svl * slices;
for slice = 0 to slices-1
    ZAhslice[tile, esize, slice, svl] = Elem[value, slice, svl];

aarch64/functions/sme/ZAvector

// ZAvector[] - non-assignment form
// ===============================
bits(width) ZAvector[integer index, integer width] = bits(width) value
assert width == SVL;
assert index >= 0 && index < (width DIV 8);
return _ZA[index]<width-1:0>;

// ZAvector[] - assignment form
// =============================
ZAvector[integer index, integer width] = bits(width) value
assert width == SVL;
assert index >= 0 && index < (width DIV 8);
if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
    _ZA[index] = ZeroExtend(value, MAX_VL);
else
    _ZA[index]<width-1:0> = value;

aarch64/functions/sme/ZAvslice

// ZAvslice[] - non-assignment form
// ===============================
bits(width) ZAvslice[integer tile, integer esize, integer slice, integer width] = bits(width) value
integer slices = SVL DIV esize;
bits(width) result;
for s = 0 to slices-1
    bits(width) hslice = ZAhslice[tile, esize, s, width];
    Elem[result, s, esize] = Elem[hslice, slice, esize];
return result;

// ZAvslice[] - assignment form
// =============================
ZAvslice[integer tile, integer esize, integer slice, integer width] = bits(width) value
integer slices = SVL DIV esize;
for s = 0 to slices-1
    bits(width) hslice = ZAhslice[tile, esize, s, width];
    Elem[hslice, slice, esize] = Elem[value, s, esize];
    ZAhslice[tile, esize, s, width] = hslice;
aarch64/functions/sve/AArch32.IsFPEnabled

// AArch32.IsFPEnabled()
// =====================
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch32 state and FALSE otherwise.

boolean AArch32.IsFPEnabled(bits(2) el)
if el == EL0 && !ELUsingAArch32(EL1) then
    return AArch64.IsFPEnabled(el);
if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState() == SS_NonSecure then
    // Check if access disabled in NSACR
    if NSACR.cp10 == '0' then return FALSE;
if el IN {EL0, EL1} then
    // Check if access disabled in CPACR
    boolean disabled;
    case CPACR.cp10 of
        when '00' disabled = TRUE;
        when '01' disabled = el == EL0;
        when '10' disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
        when '11' disabled = FALSE;
    if disabled then return FALSE;
if el IN {EL0, EL1, EL2} && EL2Enabled() then
    if !ELUsingAArch32(EL2) then
        return AArch64.IsFPEnabled(EL2);
    if HCPTR.TCP10 == '1' then return FALSE;
if HaveEL(EL3) && !ELUsingAArch32(EL3) then
    // Check if access disabled in CPTR_EL3
    if CPTR_EL3.TFP == '1' then return FALSE;
return TRUE;

aarch64/functions/sve/AArch64.IsFPEnabled

// AArch64.IsFPEnabled()
// =====================
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch64 state and FALSE otherwise.

boolean AArch64.IsFPEnabled(bits(2) el)
if el IN {EL0, EL1} && !IsInHost() then
    // Check FP&SIMD at EL0/EL1
    boolean disabled;
    case CPACR_EL1.FPEN of
        when 'x0' disabled = TRUE;
        when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
        when '11' disabled = FALSE;
    if disabled then return FALSE;
    else
        if CPTR_EL2.TFP == '1' then return FALSE;
// Check if access disabled in CPTR_EL3
if HaveEL(EL3) then
if CPTR_EL3.TFP == '1' then return FALSE;
return TRUE;

aarch64/functions/sve/AnyActiveElement

// AnyActiveElement()
// ===============
// Return TRUE if there is at least one active element in mask. Otherwise, return FALSE.

boolean AnyActiveElement(bits(N) mask, integer esize)
return LastActiveElement(mask, esize) >= 0;

aarch64/functions/sve/BitDeposit

// BitDeposit()
// ============
// Deposit the least significant bits from DATA into result positions selected by non-zero bits in MASK, setting other result bits to zero.

bits(N) BitDeposit (bits(N) data, bits(N) mask)
bits(N) res = Zeros(N);
integer db = 0;
for rb = 0 to N-1
if mask<rb> == '1' then
res<rb> = data<db>;
db = db + 1;
return res;

aarch64/functions/sve/BitExtract

// BitExtract()
// ============
// Extract and pack DATA bits selected by the non-zero bits in MASK into the least significant result bits, setting other result bits to zero.

bits(N) BitExtract (bits(N) data, bits(N) mask)
bits(N) res = Zeros(N);
integer rb = 0;
for db = 0 to N-1
if mask<db> == '1' then
res<rb> = data<db>;
rb = rb + 1;
return res;

aarch64/functions/sve/BitGroup

// BitGroup()
// ===========
// Extract and pack DATA bits selected by the non-zero bits in MASK into the least significant result bits, and pack unselected bits into the most significant result bits.

bits(N) BitGroup (bits(N) data, bits(N) mask)
bits(N) res;
integer rb = 0;
// compress masked bits to right
for db = 0 to N-1
  if mask<db> == '1' then
    res<rb> = data<db>;
    rb = rb + 1;
  // compress unmasked bits to left
for db = 0 to N-1
  if mask<db> == '0' then
    res<rb> = data<db>;
    rb = rb + 1;
return res;

aarch64/functions/sve/CeilPow2

// CeilPow2()
// =========
// For a positive integer X, return the smallest power of 2 \geq X
integer CeilPow2(integer x)
if x == 0 then return 0;
if x == 1 then return 2;
return FloorPow2(x - 1) * 2;

aarch64/functions/sve/CheckNonStreamingSVEEnabled

// CheckNonStreamingSVEEnabled()
// =============================
// Checks for traps on SVE instructions that are not legal in streaming mode.
CheckNonStreamingSVEEnabled()
  CheckSVEEnabled();
  if HaveSME() && PSTATE.SM == '1' && !IsFullA64Enabled() then
    SMEAccessTrap(SMEExceptionType_Streaming, PSTATE.EL);

aarch64/functions/sve/CheckNormalSVEEnabled

// CheckNormalSVEEnabled()
// =======================
// Checks for traps on normal SVE instructions and instructions that
// access SVE System registers.
CheckNormalSVEEnabled()
  boolean disabled;
  if (HaveEL(EL3) && (CPTR_EL3.EZ == '0' || CPTR_EL3.TFP == '1') &&
    EL3SGDTTrapPriority()) then
    UNDEFINED;
  // Check if access disabled in CPACR_EL1
  if PSTATE_EL IN {EL0, EL1} && !IsInHost() then
    // Check SVE at EL0/EL1
    case CPACR_EL1.ZEN of
      when 'x0' disabled = TRUE;
      when '01' disabled = PSTATE_EL == EL0;
      when '11' disabled = FALSE;
    if disabled then SVEAccessTrap(EL1);
    // Check SIMD&FP at EL0/EL1
    case CPACR_EL1.FPEN of
      when 'x0' disabled = TRUE;
      when '01' disabled = PSTATE_EL == EL0;
      when '11' disabled = FALSE;
    if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);
// Check if access disabled in CPTR_EL2
if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
  if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
    // Check SVE at EL2
    case CPTR_EL2.ZEN of
      when 'x0' disabled = TRUE;
      when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
      when '11' disabled = FALSE;
    if disabled then SVEAccessTrap(EL2);
  // Check SIMD&FP at EL2
  case CPTR_EL2.FPEN of
    when 'x0' disabled = TRUE;
    when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
    when '11' disabled = FALSE;
  if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
  else
    if CPTR_EL2.TZ == '1' then SVEAccessTrap(EL2);
    if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);
  // Check if access disabled in CPTR_EL3
  if HaveEL(EL3) then
    if CPTR_EL3.EZ == '0' then
      if Halted() && EDSCR.SDD == '1' then UNDEFINED;
      else SVEAccessTrap(EL3);
    if CPTR_EL3.TFP == '1' then
      if Halted() && EDSCR.SDD == '1' then UNDEFINED;
      else AArch64.AdvSIMDFPAccessTrap(EL3);
  // Check if access disabled in CPACR_EL1
  if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
    // Check SME at EL0/EL1
    case CPACR_EL1.SMEN of
      when 'x0' disabled = TRUE;
      when '01' disabled = PSTATE.EL == EL0;
      when '11' disabled = FALSE;
    if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);
  if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
    if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
      // Check SME at EL2
      case CPTR_EL2.SMEN of
        when 'x0' disabled = TRUE;
        when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
        when '11' disabled = FALSE;
      if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
      else
        if CPTR_EL2.TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
    // Check if access disabled in CPTR_EL3

if HaveEL(EL3) then
    if CPTR_EL3.ESM == '0' then SMEAccessTrap(SMEEExceptionType_AccessTrap, EL3);

aarch64/functions/sve/CheckSMEAndZAEnabled

// CheckSMEAndZAEnabled()
// =====================

CheckSMEAndZAEnabled()
    CheckSMEEnabled();

    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEEExceptionType_InactiveZA, PSTATE.EL);

aarch64/functions/sve/CheckSMEEnabled

// CheckSMEEnabled()
// ===============

CheckSMEEnabled()
    boolean disabled;
    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
            if disabled then SMEAccessTrap(SMEEExceptionType_AccessTrap, EL1);

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

        if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
            if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
                // Check SME at EL2
                case CPTR_EL2.SMEN of
                    when 'x0' disabled = TRUE;
                    when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                    when '11' disabled = FALSE;
                    if disabled then SMEAccessTrap(SMEEExceptionType_AccessTrap, EL2);

                // Check SIMD&FP at EL2
                case CPTR_EL2.FPEN of
                    when 'x0' disabled = TRUE;
                    when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                    when '11' disabled = FALSE;
                    if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
                else
                    if CPTR_EL2.TSM == '1' then SMEAccessTrap(SMEEExceptionType_AccessTrap, EL2);
                    if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

            // Check if access disabled in CPTR_EL3
            if HaveEL(EL3) then
                if CPTR_EL3.ESM == '0' then SMEAccessTrap(SMEEExceptionType_AccessTrap, EL3);
                if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
aarch64/functions/sve/CheckSVEEnabled

// CheckSVEEnabled()
// =================
// Checks for traps on SVE instructions and instructions that
// access SVE System registers.
CheckSVEEnabled()
if HaveSME() && (PSTATE.SM == '1' || !HaveSVE()) then
    CheckStreamingSVEEnabled();
else
    CheckNormalSVEEnabled();

aarch64/functions/sve/CheckStreamingSVEAndZAEnabled

// CheckStreamingSVEAndZAEnabled()
// ===============================
CheckStreamingSVEAndZAEnabled()
CheckStreamingSVEEnabled();
if PSTATE.ZA == '0' then
    SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);

aarch64/functions/sve/CheckStreamingSVEEnabled

// CheckStreamingSVEEnabled()
// ==========================
CheckStreamingSVEEnabled()
CheckSMEEnabled();
if PSTATE.SM == '0' then
    SMEAccessTrap(SMEExceptionType_NotStreaming, PSTATE.EL);

aarch64/functions/sve/CurrentVL

// CurrentVL - non-assignment form
// ===============================
integer CurrentVL
return if HaveSME() && PSTATE.SM == '1' then SVL else NVL;

aarch64/functions/sve/DecodePredCount

// DecodePredCount()
// =================
integer DecodePredCount(bits(5) pattern, integer esize)
    integer elements = CurrentVL DIV esize;
    integer numElem;
    case pattern of
        when '00000' numElem = FloorPow2(elements);
        when '00001' numElem = if elements >= 1 then 1 else 0;
        when '00010' numElem = if elements >= 2 then 2 else 0;
        when '00011' numElem = if elements >= 3 then 3 else 0;
        when '00100' numElem = if elements >= 4 then 4 else 0;
        when '00101' numElem = if elements >= 5 then 5 else 0;
        when '00110' numElem = if elements >= 6 then 6 else 0;
        when '00111' numElem = if elements >= 7 then 7 else 0;
        when '01000' numElem = if elements >= 8 then 8 else 0;
when '01001' numElem = if elements >= 16 then 16 else 0;
when '01010' numElem = if elements >= 32 then 32 else 0;
when '01011' numElem = if elements >= 64 then 64 else 0;
when '01100' numElem = if elements >= 128 then 128 else 0;
when '01101' numElem = if elements >= 256 then 256 else 0;
when '11101' numElem = elements - (elements MOD 4);
when '11110' numElem = elements - (elements MOD 3);
when '11111' numElem = elements;
otherwise numElem = 0;
return numElem;

// ElemFFR[] - non-assignment form
// =================================
bit ElemFFR[integer e, integer esize]
    return ElemP[_FFR, e, esize];

// ElemFFR[] - assignment form
// ===========================
ElemFFR[integer e, integer esize] = bit value
    integer psize = esize DIV 8;
    integer n = e * psize;
    assert n >= 0 && (n + psize) <= CurrentVL DIV 8;
    _FFR<n+psize-1:n> = ZeroExtend(value, psize);
    return;

// ElemP[] - non-assignment form
// =============================
bit ElemP[bits(N) pred, integer e, integer esize]
    assert esize IN {8, 16, 32, 64, 128};
    integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n>;

// ElemP[] - assignment form
// =========================
ElemP[bits(N) &pred, integer e, integer esize] = bit value
    assert esize IN {8, 16, 32, 64, 128};
    integer psize = esize DIV 8;
    integer n = e * psize;
    assert n >= 0 && (n + psize) <= N;
    pred<n+psize-1:n> = ZeroExtend(value, psize);
    return;

// FFR[] - non-assignment form
// ===========================
bits(width) FFR[integer width]
    assert width == CurrentVL DIV 8;
    return _FFR<width-1:0>;

// FFR[] - assignment form
// =======================
Armv8 Pseudocode

J1.1 Pseudocode for AArch64 operation

FFR[integer width] = bits(width) value
assert width == CurrentVL DIV 8;
if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
    _FFR = ZeroExtend(value, MAX_PL);
else
    _FFR<width-1:0> = value;

aarch64/functions/sve/FPCompareNE

// FPCompareNE()
// =============

boolean FPCompareNE(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
boolean result;
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};

if op1_nan || op2_nan then
    result = TRUE;
    if type1 == FPType_SNaN || type2 == FPType_SNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
    else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 != value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;

aarch64/functions/sve/FPCompareUN

// FPCompareUN()
// =============

boolean FPCompareUN(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

if type1 == FPType_SNaN || type2 == FPType_SNaN then
    FPProcessException(FPExc_InvalidOp, fpcr);
result = type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN};
if !result then
    FPProcessDenorms(type1, type2, N, fpcr);

return result;

aarch64/functions/sve/FPConvertSVE

// FPConvertSVE()
// ===============

bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr_in, FPRounding rounding, integer M)
    FPCRType fpcr = fpcr_in;
fpcr.AHP = '0';
return FPConvert(op, fpcr, rounding, M);

// FPConvertSVE()
// ===============

bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr_in, integer M)
FPCRType fpcr = fpcr_in;
fpcr.AHP = '0';
return FPConvert(op, fpcr, FPRoundingMode(fpcr), M);

aarch64/functions/sve/FPExpA

// FPExpA()
// =========

bits(N) FPExpA(bits(N) op)
assert N IN {16,32,64};
bits(N) result;
bits(N) coeff;
integer idx = if N == 16 then UInt(op<4:0>) else UInt(op<5:0>);
coeff = FPExpCoefficient[idx, N];
if N == 16 then
    result<15:0> = '0':op<9:5>:coeff<9:0>;
elsif N == 32 then
    result<31:0> = '0':op<13:6>:coeff<22:0>;
else // N == 64
    result<63:0> = '0':op<16:6>:coeff<51:0>;
return result;

aarch64/functions/sve/FPExpCoefficient

// FPExpCoefficient()
// ==================

bits(N) FPExpCoefficient[integer index, integer N]
assert N IN {16,32,64};
integer result;
if N == 16 then
    case index of
        when  0 result = 0x0000;
        when  1 result = 0x0016;
        when  2 result = 0x002d;
        when  3 result = 0x0045;
        when  4 result = 0x005d;
        when  5 result = 0x0075;
        when  6 result = 0x008e;
        when  7 result = 0x00a8;
        when  8 result = 0x00c2;
        when  9 result = 0x00dc;
        when 10 result = 0x00f8;
        when 11 result = 0x0114;
        when 12 result = 0x0130;
        when 13 result = 0x014d;
        when 14 result = 0x016b;
        when 15 result = 0x0189;
        when 16 result = 0x01a8;
        when 17 result = 0x01c8;
        when 18 result = 0x01e8;
        when 19 result = 0x0209;
        when 20 result = 0x022b;
        when 21 result = 0x024e;
        when 22 result = 0x0271;
        when 23 result = 0x0295;
        when 24 result = 0x02ba;
        when 25 result = 0x02e0;
        when 26 result = 0x0306;
        when 27 result = 0x032e;
        when 28 result = 0x0356;
when 29 result = 0x037f;
when 30 result = 0x81a9;
when 31 result = 0x03d4;

elsif N == 32 then
  case index of
    when 0 result = 0x000000;
    when 1 result = 0x0164d2;
    when 2 result = 0x02cd87;
    when 3 result = 0x043a29;
    when 4 result = 0x05aac3;
    when 5 result = 0x071f62;
    when 6 result = 0x08980f;
    when 7 result = 0x0a14d5;
    when 8 result = 0x0b95c2;
    when 9 result = 0x0d1adf;
    when 10 result = 0x000000;
    when 11 result = 0x1031dc;
    when 12 result = 0x11c3d3;
    when 13 result = 0x135a2b;
    when 14 result = 0x14f4f0;
    when 15 result = 0x16942d;
    when 16 result = 0x1817f0;
    when 17 result = 0x19e046;
    when 18 result = 0x1b8d3a;
    when 19 result = 0x1d3eda;
    when 20 result = 0x1ef532;
    when 21 result = 0x20b051;
    when 22 result = 0x227043;
    when 23 result = 0x243516;
    when 24 result = 0x25fed7;
    when 25 result = 0x27cd94;
    when 26 result = 0x29a15b;
    when 27 result = 0x2b7a3a;
    when 28 result = 0x2d583f;
    when 29 result = 0x2f3b79;
    when 30 result = 0x3123f6;
    when 31 result = 0x3311c4;
    when 32 result = 0x3504f3;
    when 33 result = 0x36fd92;
    when 34 result = 0x38fbaf;
    when 35 result = 0x3aff5b;
    when 36 result = 0x3d08a4;
    when 37 result = 0x3fe179a;
    when 38 result = 0x4124c4d;
    when 39 result = 0x4346cd;
    when 40 result = 0x45672a;
    when 41 result = 0x478d75;
    when 42 result = 0x490b9be;
    when 43 result = 0x4bec15;
    when 44 result = 0x4e248c;
    when 45 result = 0x506334;
    when 46 result = 0x52a81e;
    when 47 result = 0x54f35b;
    when 48 result = 0x5744fd;
    when 49 result = 0x599d16;
    when 50 result = 0x5bfbb8;
    when 51 result = 0x5e60f5;
    when 52 result = 0x60cddf;
    when 53 result = 0x633f89;
    when 54 result = 0x65907;
    when 55 result = 0x68396a;
    when 56 result = 0x6ac8c7;
    when 57 result = 0x6dd4f30;
    when 58 result = 0x6fe4ba;
    when 59 result = 0x728177;
    when 60 result = 0x75257d;
    when 61 result = 0x77d0df;
when 62 \text{ result } = 0x7a83b3;
when 63 \text{ result } = 0x7d3e0c;

else // N == 64
    case index of
        when 0 \text{ result } = 0x0000000000000000;
        when 1 \text{ result } = 0x02c9a3e7778061;
        when 2 \text{ result } = 0x059b0031585774;
        when 3 \text{ result } = 0x0874518759b8c8;
        when 4 \text{ result } = 0x0b5586c9f890f;
        when 5 \text{ result } = 0x0e3ec3203d1a2;
        when 6 \text{ result } = 0x11d01d0125b51;
        when 7 \text{ result } = 0x1429a92e92d0;
        when 8 \text{ result } = 0x172883c7d517b8;
        when 9 \text{ result } = 0x1a358e6fcb75;
        when 10 \text{ result } = 0x1d4873168b99a;
        when 11 \text{ result } = 0x206388628c6d;
        when 12 \text{ result } = 0x2387a6e756238;
        when 13 \text{ result } = 0x26b4563e27cdd;
        when 14 \text{ result } = 0x29e90f51fdeee1;
        when 15 \text{ result } = 0x2d285e6e4030d;
        when 16 \text{ result } = 0x306fe8a21b715;
        when 17 \text{ result } = 0x33c08b2e416ff;
        when 18 \text{ result } = 0x371a7373a90cb;
        when 19 \text{ result } = 0x3a7d834e59ff7;
        when 20 \text{ result } = 0x3de64c123422;
        when 21 \text{ result } = 0x4160a21f72e2a;
        when 22 \text{ result } = 0x440e86061892d;
        when 23 \text{ result } = 0x486a285c13cc0;
        when 24 \text{ result } = 0x4b5d05362a27;
        when 25 \text{ result } = 0x4f9b27e6d02ca7;
        when 26 \text{ result } = 0x5342b56d04fb82;
        when 27 \text{ result } = 0x56f473e8b527da;
        when 28 \text{ result } = 0x5a8070d48429;
        when 29 \text{ result } = 0x5e76f15a02148;
        when 30 \text{ result } = 0x6247e0b03a585;
        when 31 \text{ result } = 0x6623882552225;
        when 32 \text{ result } = 0x6a09e667f3bcd;
        when 33 \text{ result } = 0x6dfb236c518a2f;
        when 34 \text{ result } = 0x71f7e8e5c7f74;
        when 35 \text{ result } = 0x75fe8564267c9;
        when 36 \text{ result } = 0x7914a73e0b187;
        when 37 \text{ result } = 0x7e2f3363ef462;
        when 38 \text{ result } = 0x82589994cc13;
        when 39 \text{ result } = 0x86bd99b4492ed;
        when 40 \text{ result } = 0x8ace5422aa0b8;
        when 41 \text{ result } = 0x8f2a99157736;
        when 42 \text{ result } = 0x93737b0c655e5;
        when 43 \text{ result } = 0x970b29d4e450;
        when 44 \text{ result } = 0x9c49182a3f090;
        when 45 \text{ result } = 0xa0e667503e565;
        when 46 \text{ result } = 0xa550b23e255d;
        when 47 \text{ result } = 0xa9e685579f9bf;
        when 48 \text{ result } = 0xae89f995a03ad;
        when 49 \text{ result } = 0xb33a2884f15f8;
        when 50 \text{ result } = 0xb77f67f2fb5e47;
        when 51 \text{ result } = 0xbcc1e904bc1d2;
        when 52 \text{ result } = 0xc199b0d85529c;
        when 53 \text{ result } = 0xc67f12e570d14b;
        when 54 \text{ result } = 0xcb7200ceff9069;
        when 55 \text{ result } = 0xd072d4a87897c;
        when 56 \text{ result } = 0xdc81b0c84a87;
        when 57 \text{ result } = 0xdae9683d0b3285;
        when 58 \text{ result } = 0xdfc9733789b5f;
        when 59 \text{ result } = 0xe502ee78b3ff6;
        when 60 \text{ result } = 0xea4af2aa490da;
        when 61 \text{ result } = 0xe1a8ee615a27;
        when 62 \text{ result } = 0xf5076586e4540;
when 63 result = 0xFA7C1819E90D8;
return result<N-1:0>;

aarch64/functions/sve/FPLogB

// FPLogB()
// =========

bits(N) FPLogB(bits(N) op, FPCRType fpcr)
assert N IN {16,32,64};
integer result;
(fptype,sign,value) = FPUnpack(op, fpcr);

if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN || fptype == FPTYPE_Zero then
FPProcessException(FPExc_InvalidOp, fpcr);
result = -(2^(N-1));            // MinInt, 100..00
elsif fptype == FPTYPE_Infinity then
result = 2^(N-1) - 1;           // MaxInt, 011..11
else
// FPUnpack has already scaled a subnormal input
value = Abs(value);
result = 0;
while value < 1.0 do
value = value * 2.0;
result = result - 1;
while value >= 2.0 do
value = value / 2.0;
result = result + 1;
FPProcessDenorm(fptype, N, fpcr);
return result<N-1:0>;

aarch64/functions/sve/FPMinNormal

// FPMinNormal()
// =============

bits(N) FPMinNormal(bit sign, integer N)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = Zeros(E-1):'1';
frac = Zeros(F);
return sign : exp : frac;

aarch64/functions/sve/FPOne

// FPOne()
// ========

bits(N) FPOne(bit sign, integer N)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '0':Ones(E-1); 
frac = Zeros(F);
return sign : exp : frac;
aarch64/functions/sve/FPPointFive

// FPPointFive()
// =============

bits(N) FPPointFive(bit sign, integer N)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '0':Ones(E-2):'0';
frac = Zeros(F);
return sign : exp : frac;

aarch64/functions/sve/FPProcess

// FPProcess()
// ===========

bits(N) FPProcess(bits(N) input)
bits(N) result;
assert N IN {16,32,64};
FPCRType fpcr = FPCR[];
(ftype,sign,value) = FPUnpack(input, fpcr);
if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
  result = FPProcessNaN(fptype, input, fpcr);
elsif fptype == FPTYPE_Infinity then
  result = FPInfinity(sign, N);
elsif fptype == FPTYPE_Zero then
  result = FPZero(sign, N);
else
  result = FPRound(value, fpcr, N);
  FPProcessDenorm(fptype, N, fpcr);
return result;

aarch64/functions/sve/FPScale

// FPScale()
// =========

bits(N) FPScale(bits (N) op, integer scale, FPCRType fpcr)
assert N IN {16,32,64};
bits(N) result;
(ftype,sign,value) = FPUnpack(op, fpcr);
if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
  result = FPProcessNaN(fptype, op, fpcr);
elsif fptype == FPTYPE_Zero then
  result = FPZero(sign, N);
elsif fptype == FPTYPE_Infinity then
  result = FPInfinity(sign, N);
else
  result = FPRound(value * (2.0^scale), fpcr, N);
  FPProcessDenorm(fptype, N, fpcr);
return result;
aarch64/functions/sve/FPTrigMAdd

// FPTrigMAdd()
// ============

bits(N) FPTrigMAdd(integer x_in, bits(N) op1, bits(N) op2_in, FPCRType fpcr)
assert N IN {16,32,64};
bits(N) coeff;
bits(N) op2 = op2_in;
integer x = x_in;
assert x >= 0;
assert x < 8;
if op2<N-1> == '1' then
  x = x + 8;
coeff    = FPTrigMAddCoefficient[x, N];
op2      = FPAbs(op2);
result   = FPMulAdd(coeff, op1, op2, fpcr);
return result;

aarch64/functions/sve/FPTrigMAddCoefficient

// FPTrigMAddCoefficient()
// =======================

bits(N) FPTrigMAddCoefficient[integer index, integer N]
assert N IN {16,32,64};
integer result;
if N == 16 then
  case index of
    when 0 result = 0x3c00;
    when 1 result = 0xb155;
    when 2 result = 0x2030;
    when 3 result = 0x0000;
    when 4 result = 0x0000;
    when 5 result = 0x0000;
    when 6 result = 0x0000;
    when 7 result = 0x0000;
    when 8 result = 0x3c00;
    when 9 result = 0xb800;
    when 10 result = 0x293a;
    when 11 result = 0x0000;
    when 12 result = 0x0000;
    when 13 result = 0x0000;
    when 14 result = 0x0000;
    when 15 result = 0x0000;
  elsif N == 32 then
    case index of
      when 0 result = 0x3f800000;
      when 1 result = 0xbe2aaaab;
      when 2 result = 0x3c088886;
      when 3 result = 0xb95008b9;
      when 4 result = 0x36369d6d;
      when 5 result = 0x00000000;
      when 6 result = 0x00000000;
      when 7 result = 0x00000000;
      when 8 result = 0x3f800000;
      when 9 result = 0xbf000000;
      when 10 result = 0x3d2aaaa6;
      when 11 result = 0xbab60705;
      when 12 result = 0x37cd37cc;
      when 13 result = 0x00000000;
      when 14 result = 0x00000000;
      when 15 result = 0x00000000;
else // N == 64
    case index of
        when 0 result = 0x3ff0000000000000;
        when 1 result = 0xbfc5555555555543;
        when 2 result = 0x3f8111111110f30c;
        when 3 result = 0xbf2a01a019b92fc6;
        when 4 result = 0x3ec71de351f3d22b;
        when 5 result = 0xbe5a5e5e2b60f7b91;
        when 6 result = 0x3de5d8408868552f;
        when 7 result = 0x0000000000000000;
        when 8 result = 0x3f80000000000000;
        when 9 result = 0xbfe0000000000000;
        when 10 result = 0x3fa5555555555536;
        when 11 result = 0xbf56c16c16c130a0b;
        when 12 result = 0x3efa01a019b1e8d8;
        when 13 result = 0xbe927e4f7282f468;
        when 14 result = 0x3e21ee96d2641b13;
        when 15 result = 0xbda8f76380fbb401;
    return result<N-1:0>;

aarch64/functions/sve/FPTrigSMul

    // FPTrigSMul()
    // ============
    bits(N) FPTrigSMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    result = FPMul(op1, op1, fpcr);
    fpexc = FALSE;
    (fptype, sign, value) = FPUnpack(result, fpcr, fpexc);
    if !(fptype IN {FPType_QNaN, FPType_SNaN}) then
        result<N-1> = op2<0>;
    return result;

aarch64/functions/sve/FPTrigSSel

    // FPTrigSSel()
    // ============
    bits(N) FPTrigSSel(bits(N) op1, bits(N) op2)
    assert N IN {16,32,64};
    bits(N) result;
    if op2<0> == '1' then
        result = FPOne(op2<1>, N);
    elseif op2<1> == '1' then
        result = FPNeg(op1);
    else
        result = op1;
    return result;

aarch64/functions/sve/FirstActive

    // FirstActive()
    // =============
    bit FirstActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
if ElemP[mask, e, esize] == '1' then return ElemP[x, e, esize];
return '0';

aarch64/functions/sve/FloorPow2

// FloorPow2()
// ===========
// For a positive integer X, return the largest power of 2 <= X

integer FloorPow2(integer x)
assert x >= 0;
integer n = 1;
if x == 0 then return 0;
while x >= 2^n do
    n = n + 1;
return 2^(n - 1);

aarch64/functions/sve/HaveSMEFullA64

// HaveSMEFullA64()
// ================
// Returns TRUE if the SME FA64 extension is implemented, FALSE otherwise.

boolean HaveSMEFullA64()
return HaveSME() && boolean IMPLEMENTATION_DEFINED "Have SME FA64 extension";

aarch64/functions/sve/HaveSVE

// HaveSVE()
// =========

boolean HaveSVE()
return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Have SVE ISA";

aarch64/functions/sve/HaveSVE2

// HaveSVE2()
// ===========
// Returns TRUE if the SVE2 extension is implemented, FALSE otherwise.

boolean HaveSVE2()
return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE2 extension";

aarch64/functions/sve/HaveSVE2AES

// HaveSVE2AES()
// =============
// Returns TRUE if the SVE2 AES extension is implemented, FALSE otherwise.

boolean HaveSVE2AES()
return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 AES extension";

aarch64/functions/sve/HaveSVE2BitPerm

// HaveSVE2BitPerm()
// ===============
// Returns TRUE if the SVE2 Bit Permissions extension is implemented, FALSE otherwise.
boolean HaveSVE2BitPerm()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 BitPerm extension";

aarch64/functions/sve/HaveSVE2PMULL128

// HaveSVE2PMULL128()
// ==================
// Returns TRUE if the SVE2 128 bit PMULL extension is implemented, FALSE otherwise.

boolean HaveSVE2PMULL128()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 128 bit PMULL extension";

aarch64/functions/sve/HaveSVE2SHA3

// HaveSVE2SHA3()
// ===============
// Returns TRUE if the SVE2 SHA3 extension is implemented, FALSE otherwise.

boolean HaveSVE2SHA3()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 SHA3 extension";

aarch64/functions/sve/HaveSVE2SM4

// HaveSVE2SM4()
// ===============
// Returns TRUE if the SVE2 SM4 extension is implemented, FALSE otherwise.

boolean HaveSVE2SM4()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 SM4 extension";

aarch64/functions/sve/HaveSVEFP32MatMulExt

// HaveSVEFP32MatMulExt()
// ========================
// Returns TRUE if single-precision floating-point matrix multiply instruction support implemented and FALSE otherwise.

boolean HaveSVEFP32MatMulExt()
    return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE FP32 Matrix Multiply extension";

aarch64/functions/sve/HaveSVEFP64MatMulExt

// HaveSVEFP64MatMulExt()
// ========================
// Returns TRUE if double-precision floating-point matrix multiply instruction support implemented and FALSE otherwise.

boolean HaveSVEFP64MatMulExt()
    return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE FP64 Matrix Multiply extension";

aarch64/functions/sve/ImplementedSMEVectorLength

// ImplementedSMEVectorLength()
// ============================
// Reduce SVE/SME vector length to a supported value (power of two)
integer ImplementedSMEVectorLength(integer nbits_in)
integer maxbits = MaxImplementedSVEVL();
assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
integer nbits = Min(nbits_in, maxbits);
assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;

// Search for a supported power-of-two VL less than or equal to nbits
while nbits > 128 do
    if IsPow2(nbits) && SupportedPowerTwoSVEVL(nbits) then return nbits;
    nbits = nbits - 128;

// Return the smallest supported power-of-two VL
nbits = 128;
while nbits < maxbits do
    if SupportedPowerTwoSVEVL(nbits) then return nbits;
    nbits = nbits * 2;

// The only option is maxbits
return maxbits;

aarch64/functions/sve/ImplementedSVEVectorLength

// ImplementedSVEVectorLength()
// ============================
// Reduce SVE vector length to a supported value (e.g. power of two)
integer ImplementedSVEVectorLength(integer nbits_in)
integer nbits = Min(nbits_in, MaxImplementedVL());
assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;
while nbits > 128 do
    if IsPow2(nbits) || SupportedNonPowerTwoSVEVL(nbits) then return nbits;
    nbits = nbits - 128;
return nbits;

aarch64/functions/sve/InStreamingMode

// InStreamingMode()
// ================

boolean InStreamingMode()
    return HaveSME() && PSTATE.SM == '1';

aarch64/functions/sve/IsEven

// IsEven()
// ========

boolean IsEven(integer val)
    return val MOD 2 == 0;

aarch64/functions/sve/IsFPEnabled

// IsFPEnabled()
// =============
// Returns TRUE if accesses to the Advanced SIMD and floating-point
// registers are enabled at the target exception level in the current
// execution state and FALSE otherwise.

boolean IsFPEnabled(bits(2) el)
    if ELUsingAArch32(el) then
return AArch32.IsFPEnabled(el);
else
  return AArch64.IsFPEnabled(el);

aarch64/functions/sve/IsFullA64Enabled

// IsFullA64Enabled()
// ==================
// Returns TRUE is full A64 is enabled in Streaming mode and FALSE otherwise.

boolean IsFullA64Enabled()
  if !HaveSMEFullA64() then return FALSE;
  if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
    if SMCR_EL1.FA64 == '0' then return FALSE;
  if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
    if SMCR_EL2.FA64 == '0' then return FALSE;
  if HaveEL(EL3) then
    if SMCR_EL3.FA64 == '0' then return FALSE;
  return TRUE;

aarch64/functions/sve/IsNormalSVEEnabled

// IsNormalSVEEnabled()
// ====================
// Returns TRUE if access to normal SVE is enabled at the target
// exception level and FALSE otherwise.

boolean IsNormalSVEEnabled(bits(2) el)
  boolean disabled;
  if ELUsingAArch32(el) then
    return FALSE;
  if el IN {EL0, EL1} && !IsInHost() then
    case CPACR_EL1.ZEN of
      when 'x0' disabled = TRUE;
      when '01' disabled = el == EL0;
      when '11' disabled = FALSE;
    if disabled then return FALSE;
  if el IN {EL0, EL1, EL2} && EL2Enabled() then
    if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
      case CPTR_EL2.ZEN of
        when 'x0' disabled = TRUE;
        when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
        when '11' disabled = FALSE;
      if disabled then return FALSE;
    else
      if CPTR_EL2.TZ == '1' then return FALSE;
  if HaveEL(EL3) then
    if CPTR_EL3.EZ == '0' then return FALSE;
```
return TRUE;

aarch64/functions/sve/IsOdd

// IsOdd()
// ========
boolean IsOdd(integer val)
return val MOD 2 == 1;

aarch64/functions/sve/IsPow2

// IsPow2()
// ========
// Return TRUE if positive integer X is a power of 2. Otherwise, return FALSE.
boolean IsPow2(integer x)
if x <= 0 then return FALSE;
return FloorPow2(x) == CeilPow2(x);

aarch64/functions/sve/IsSVEEnabled

// IsSVEEnabled()
// ===============
// Returns TRUE if access to SVE instructions and System registers is enabled at the target exception level and FALSE otherwise.
boolean IsSVEEnabled(bits(2) el)
if HaveSME() && PSTATE.SM == '1' then
return IsStreamingSVEEnabled(el);
elseif HaveSVE() then
return IsNormalSVEEnabled(el);
else
return FALSE;

aarch64/functions/sve/IsStreamingSVEEnabled

// IsStreamingSVEEnabled()
// ========================
// Returns TRUE if access to streaming SVE is enabled at the target exception level and FALSE otherwise.
boolean IsStreamingSVEEnabled(bits(2) el)
boolean disabled;
if ELUsingAArch32(el) then
return FALSE;
// Check if access disabled in CPACR_EL1
if el IN {EL0, EL1} && !IsInHost() then
// Check SME at EL0/EL1
case CPACR_EL1.SMEN of
  when 'x0' disabled = TRUE;
  when '01' disabled = el == EL0;
  when '11' disabled = FALSE;
  if disabled then return FALSE;
// Check if access disabled in CPTR_EL2
if el IN {EL0, EL1, EL2} && EL2Enabled() then
  if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
```
case CPTR_EL2.SMEN of  
    when 'x0' disabled = TRUE;
    when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
    when '11' disabled = FALSE;
    if disabled then return FALSE;
    else
        if CPTR_EL2.TSM == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then return FALSE;
    return TRUE;

aarch64/functions/sve/LastActive

    // LastActive()
    // ============
    bit LastActive(bits(N) mask, bits(N) x, integer esize)
        integer elements = N DIV (esize DIV 8);
        for e = elements-1 downto 0
            if ElemP[mask, e, esize] == '1' then return ElemP[x, e, esize];
        return '0';

aarch64/functions/sve/LastActiveElement

    // LastActiveElement()
    // ===============
    integer LastActiveElement(bits(N) mask, integer esize)
        integer elements = N DIV (esize DIV 8);
        for e = elements-1 downto 0
            if ElemP[mask, e, esize] == '1' then return e;
        return -1;

aarch64/functions/sve/MaxImplementedSVL

    // MaxImplementedSVL()
    // ==============
    integer MaxImplementedSVL()
        return integer IMPLEMENTATION_DEFINED;

aarch64/functions/sve/MaxImplementedVL

    // MaxImplementedVL()
    // ================
    integer MaxImplementedVL()
        return integer IMPLEMENTATION_DEFINED;

aarch64/functions/sve/MaybeZeroSVEUppers

    // MaybeZeroSVEUppers()
    // ===============
    MaybeZeroSVEUppers(bits(2) target_el)
        boolean lower_enabled;
if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) then
    return;
if target_el == EL3 then
    if EL2Enabled() then
        lower_enabled = IsFPEnabled(EL2);
    else
        lower_enabled = IsFPEnabled(EL1);
    elsif target_el == EL2 then
        assert !ELUsingAArch32(EL2);
        if HCR_EL2.TGE == '0' then
            lower_enabled = IsFPEnabled(EL1);
        else
            lower_enabled = IsFPEnabled(EL0);
        end if
    else
        assert target_el == EL1 && !ELUsingAArch32(EL1);
        lower_enabled = IsFPEnabled(EL0);
    end if
if lower_enabled then
    integer vl = if IsSVEEnabled(PSTATE.EL) then CurrentVL else 128;
    integer pl = vl DIV 8;
    for n = 0 to 31
        if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
            _Z[n] = ZeroExtend(_Z[n]<vl-1:0>, MAX_VL);
        end if
    end for
    for n = 0 to 15
        if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
            _P[n] = ZeroExtend(_P[n]<pl-1:0>, MAX_PL);
        end if
    end for
    if HaveSME() && PSTATE.ZA == '1' then
        integer accessiblevecs = SVL DIV 8;
        integer allvecs = MaxImplementedSVL() DIV 8;
        for n = 0 to accessiblevecs - 1
            if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
                _ZA[n] = ZeroExtend(_ZA[n]<SVL-1:0>, MAX_VL);
            end if
        end for
        for n = accessiblevecs to allvecs - 1
            if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
                _ZA[n] = Zeros(MAX_VL);
            end if
        end for
    end if
end if

// MemNF[] - non-assignment form
// =============================
(bits(8*size), boolean) MemNF(bits(64) address, integer size, AccType acctype)
assert size IN {1, 2, 4, 8, 16};
bits(8*size) value;
boolean bad;
aligned = (address == Align(address, size));
A = SCTLR[].A;
if !aligned && (A == '1') then
    return (bits(8*size) UNKNOWN, TRUE);
atomic = aligned || size == 1;
if !atomic then
    value<7:0>, bad = MemSingleNF[address, 1, acctype, aligned];
    if bad then
        return (bits(8*size) UNKNOWN, TRUE);
    end if
else
    return (bits(8*size) UNKNOWN, TRUE);
end if
// For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
// access will generate an Alignment Fault, as to get this far means the first byte did
// not, so we must be changing to a new translation page.
if !aligned then
    c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
    assert c IN {Constraint_FAULT, Constraint_NONE};
    if c == Constraint_NONE then aligned = TRUE;
for i = 1 to size-1
    (value<8*i+7:8*i>, bad) = MemSingleNF[address+i, 1, acctype, aligned];
    if bad then
        return (bits(8*size) UNKNOWN, TRUE);
else
    (value, bad) = MemSingleNF[address, size, acctype, aligned];
    if bad then
        return (bits(8*size) UNKNOWN, TRUE);
if BigEndian(acctype) then
    value = BigEndianReverse(value);
return (value, FALSE);

aarch64/functions/sve/MemSingleNF

// MemSingleNF[] - non-assignment form
// ===============
(bits(8*size), boolean) MemSingleNF[bits(64) address, integer size, AccType acctype, boolean aligned]
assert acctype IN {AccType_CNOTFIRST, AccType_NONFAULT};
bits(8*size) value;
boolean iswrite = FALSE;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
FaultRecord fault;

// Implementation may suppress NF load for any reason
if ConstrainUnpredictableBool(Unpredictable_NONFAULT) then
    return (bits(8*size) UNKNOWN, TRUE);
boolean istagaccess = HaveMTE2Ext() && AArch64.AccessIsTagChecked(address, acctype);

// MMU or MPU
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

// Non-fault load from Device memory must not be performed externally
if memaddrdesc.memattrs.memtype == MemType_Device then
    return (bits(8*size) UNKNOWN, TRUE);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    return (bits(8*size) UNKNOWN, TRUE);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
if HaveTME() then
    accdesc.transactional = TSTATE.depth > 0;
boolean istagchecked = istagaccess;
if istagchecked then
    bits(4) ptag = AArch64.PhysicalTag(address);
    if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
        return (bits(8*size) UNKNOWN, TRUE);

(memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
if IsFault(memstatus) then
    if IsExternalAbortTakenSynchronously(memstatus, iswrite, memaddrdesc, size, accdesc) then
J1.1 Pseudocode for AArch64 operation

```plaintext
return (bits(8*size) UNKNOWN, TRUE);
fault = NoFault();
fault.errortype = memstatus.errortype;
fault.acctype = memstatus.acctype;
fault.extflag = memstatus.extflag;
fault.statuscode = memstatus.statuscode;
PendSErrorInterrupt(fault);

return (value, FALSE);
```

```plaintext
aarch64/functions/sve/NVL

// NVL - non-assignment form
// ================
// Normal VL

integer NVL
    integer vl;
    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 & IsInHost()) then
        vl = UInt(ZCR_EL1.LEN);
    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 & IsInHost()) then
        vl = UInt(ZCR_EL2.LEN);
    elsif PSTATE.EL IN {EL0, EL1} & EL2Enabled() then
        vl = Min(vl, UInt(ZCR_EL2.LEN));
    if PSTATE.EL == EL3 then
        vl = UInt(ZCR_EL3.LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(ZCR_EL3.LEN));
    vl = (vl + 1) * 128;
    vl = ImplementedSVEVectorLength(vl);
    return vl;
```

```plaintext
aarch64/functions/sve/NoneActive

// NoneActive()
// ============

bit NoneActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' && ElemP[x, e, esize] == '1' then return '0';
    return '1';
```

```plaintext
aarch64/functions/sve/P

// P[] - non-assignment form
// ================

bits(width) P[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width == CurrentVL DIV 8;
    return _P[n]<width-1:0>;

// P[] - assignment form
// ================

P[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
```
assert width == CurrentVL DIV 8;
if ConstrainUnpredictableBool(impredictable_SVEZEROUPPER) then
    _P[n] = ZeroExtend(value, MAX_PL);
else
    _P[n]<width-1:0> = value;

aarch64/functions/sve/PredTest

// PredTest()
// =========

bits(4) PredTest(bits(N) mask, bits(N) result, integer esize)
    bit n = FirstActive(mask, result, esize);
    bit z = NoneActive(mask, result, esize);
    bit c = NOT LastActive(mask, result, esize);
    bit v = '0';
    return n:z:c:v;

aarch64/functions/sve/ReducePredicated

// ReducePredicated()
// ==================

bits(esize) ReducePredicated(ReduceOp op, bits(N) input, bits(M) mask, bits(esize) identity)
assert(N == M * 8);
integer p2bits = CeilPow2(N);
bits(p2bits) operand;
integer elements = p2bits DIV esize;
for e = 0 to elements-1
    if e * esize < N && ElemP[mask, e, esize] == '1' then
        Elem[operand, e, esize] = Elem[input, e, esize];
    else
        Elem[operand, e, esize] = identity;
return Reduce(op, operand, esize);

aarch64/functions/sve/ResetSMEState

// ResetSMEState()
// ===============

ResetSMEState()
    integer vectors = MAX_VL DIV 8;
    for n = 0 to vectors - 1
        _ZA[n] = Zeros(MAX_VL);

aarch64/functions/sve/ResetSVEState

// ResetSVEState()
// ===============

ResetSVEState()
    for n = 0 to 31
        _Z[n] = Zeros(MAX_VL);
    for n = 0 to 15
        _P[n] = Zeros(MAX_PL);
    _FFR = Zeros(MAX_PL);
    FPSR = ZeroExtend(0x0800009f<31:0>, 64);
aarch64/functions/sve/Reverse

// Reverse()
// =========
// Reverse subwords of M bits in an N-bit word

bits(N) Reverse(bits(N) word, integer M)
  bits(N) result;
  integer sw = N DIV M;
  assert N == sw * M;
  for s = 0 to sw-1
    Elem[result, (sw - 1) - s, M] = Elem[word, s, M];
  return result;

aarch64/functions/sve/SMEAccessTrap

// SMEAccessTrap()
// ===============
// Trapped access to SME registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SMEAccessTrap(SMEExceptionType etype, bits(2) target_el_in)
  bits(2) target_el = target_el_in;
  assert UInt(target_el) >= UInt(PSTATE.EL);
  if target_el == EL0 then
    target_el = EL1;
  boolean route_to_el2 = PSTATE.EL == EL0 && target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';
  exception = ExceptionSyndrome(Exception_SMEAccessTrap);
  bits(64) preferred_exception_return = ThisInstrAddr(64);
  vect_offset = 0x0;
  case etype of
    when SMEExceptionType_AccessTrap
      exception.syndrome<1:0> = '00';
    when SMEExceptionType_Streaming
      exception.syndrome<1:0> = '01';
    when SMEExceptionType_NotStreaming
      exception.syndrome<1:0> = '10';
    when SMEExceptionType_InactiveZA
      exception.syndrome<1:0> = '11';
    if route_to_el2 then
      AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
      AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
    
  aarch64/functions/sve/SMEExceptionType

enumeration SMEExceptionType {
  SMEExceptionType_AccessTrap,        // SME functionality trapped or disabled
  SMEExceptionType_Streaming,         // Illegal instruction in Streaming SVE mode
  SMEExceptionType_NotStreaming,      // Illegal instruction not in Streaming SVE mode
  SMEExceptionType_InactiveZA,        // Illegal instruction when ZA is inactive
};

aarch64/functions/sve/SVEAccessTrap

// SVEAccessTrap()
// ===============
// Trapped access to SVE registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SVEAccessTrap(bits(2) target_el)
  assert UInt(target_el) >= UInt(PSTATE.EL) && target_el != EL0 && HaveEL(target_el);
route_to_el2 = target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';

exception = ExceptionSyndrome(Exception_SVEAccessTrap);
bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x0;

if route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

aarch64/functions/sve/SVECmp

enumeration SVECmp { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cmp_UN };

aarch64/functions/sve/SVEMoveMaskPreferred

// SVEMoveMaskPreferred()
// ======================
// Return FALSE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single DUP instruction.
// Used as a condition for the preferred MOV<-DUPM alias.

boolean SVEMoveMaskPreferred(bits(13) imm13)
bits(64) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, 64);

// Check for 8 bit immediates
if !IsZero(imm<7:0>) then
    // Check for 'ffffffffffxy' or '00000000000000xy'
    if IsZero(imm<63:7>) && IsZero(imm<31:7>) then
        return FALSE;

    // Check for 'ffffffxyyyyyyy' or '000000xyyyyyyy'
    if imm<63:32> == imm<31:0> && IsZero(imm<31:7>) then
        return FALSE;

    // Check for 'yyyyyyyyyyyyy' or '00000000000000y'
    if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> && IsZero(imm<15:7>) then
        return FALSE;

// Check for 16 bit immediates
else
    // Check for 'ffffffffffffffxy00' or '000000000000xy00'
    if IsZero(imm<63:15>) then
        return FALSE;

    // Check for 'ffffffxyyyyyy' or '000000000000y'
    if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> then
        return FALSE;

    // Check for 'xy00xy00xy00x0'
    if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> then
        return FALSE;

return TRUE;
aarch64/functions/sve/SVL

// SVL - non-assignment form
// =========================
// Streaming SVL

integer SVL
    integer vl;
    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UInt(SMCR_EL1.LEN);
    elseif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(SMCR_EL2.LEN));
    elseif PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UInt(SMCR_EL2.LEN);
    endif
    if PSTATE.EL == EL3 then
        vl = UInt(SMCR_EL3.LEN);
    elseif HaveEL(EL3) then
        vl = Min(vl, UInt(SMCR_EL3.LEN));
    endif
    vl = (vl + 1) * 128;
    vl = ImplementedSMEVectorLength(vl);
    return vl;

aarch64/functions/sve/SetPSTATE_SM

// SetPSTATE_SM()
// ==============

SetPSTATE_SM(bit value)
    if PSTATE.SM != value then
        ResetSVEState();
        PSTATE.SM = value;
    endif

aarch64/functions/sve/SetPSTATE_SVCR

// SetPSTATE_SVCR
// ==============

SetPSTATE_SVCR(bits(32) svcr)
    SetPSTATE_SM(svr<0>);
    SetPSTATE_ZA(svr<1>);

aarch64/functions/sve/SetPSTATE_ZA

// SetPSTATE_ZA()
// ==============

SetPSTATE_ZA(bit value)
    if PSTATE.ZA != value then
        ResetSMEState();
        PSTATE.ZA = value;
    endif

aarch64/functions/sve/ShiftSat

// ShiftSat()
// ===========
integer ShiftSat(integer shift, integer esize)
    if shift > esize+1 then return esize+1;
    elsif shift < -(esize+1) then return -(esize+1);
    return shift;

aarch64/functions/sve/SupportedNonPowerTwoVL

    // SupportedNonPowerTwoVL()
    // ================
    
    boolean SupportedNonPowerTwoVL(integer nbits)
    return boolean IMPLEMENTATION_DEFINED;

aarch64/functions/sve/SupportedPowerTwoSVL

    // SupportedPowerTwoSVL()
    // ===============
    
    boolean SupportedPowerTwoSVL(integer nbits)
    return boolean IMPLEMENTATION_DEFINED;

aarch64/functions/sve/System

    constant integer MAX_VL = 2048;
    constant integer MAX_PL = 256;
    array bits(MAX_VL) _Z[0..31];
    array bits(MAX_PL) _P[0..15];
    bits(MAX_PL) _FFR;

aarch64/functions/sve/Z

    // Z[] - non-assignment form
    // ================
    
    bits(width) Z[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width == CurrentVL;
    return _Z[n]<width-1:0>;

    // Z[] - assignment form
    // ================
    
    Z[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == CurrentVL;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value, MAX_VL);
    else
        _Z[n]<width-1:0> = value;

aarch64/functions/sysregisters/CNTKCTL

    // CNTKCTL[] - non-assignment form
    // ================
    
    CNTKCTLType CNTKCTL[]
    bits(64) r;
    if IsInHost() then
        r = CNTHCTL_EL2;
**armv8 Pseudocode**

**J1.1 Pseudocode for AArch64 operation**

```c
return r;
r = CNTKCTL_EL1;
return r;
```

**aarch64/functions/sysregisters/CNTKCTLType**

type CNTKCTLType;

**aarch64/functions/sysregisters/CPACR**

```c
// CPACR[] - non-assignment form
// =============================
CPACRType CPACR[]
bits(64) r;
if IsInHost() then
  r = CPTR_EL2;
  return r;
  r = CPACR_EL1;
  return r;
```

**aarch64/functions/sysregisters/CPACRTypen**

type CPACRType;

**aarch64/functions/sysregisters/ELR**

```c
// ELR[] - non-assignment form
// ===========================
bits(64) ELR[bits(2) el]
  case el of
    when EL1  r = ELR_EL1;
    when EL2  r = ELR_EL2;
    when EL3  r = ELR_EL3;
    otherwise Unreachable();
    return r;

// ELR[] - non-assignment form
// ===========================
bits(64) ELR[]
  assert PSTATE.EL != EL0;
  return ELR[PSTATE.EL];

// ELR[] - assignment form
// =======================
ELR[bits(2) el] = bits(64) value
  bits(64) r = value;
  case el of
    when EL1  ELR_EL1 = r;
    when EL2  ELR_EL2 = r;
    when EL3  ELR_EL3 = r;
    otherwise Unreachable();
    return;

// ELR[] - assignment form
// =======================
```
ELR[] = bits(64) value
assert PSTATE.EL != EL0;
ELR[PSTATE.EL] = value;
return;

aarch64/functions/sysregisters/ESR

// ESR[] - non-assignment form
// ===========================
ESRType ESR[bits(2) regime]
bits(64) r;
case regime of
  when EL1  r = ESR_EL1;
  when EL2  r = ESR_EL2;
  when EL3  r = ESR_EL3;
  otherwise Unreachable();
return r;

// ESR[] - assignment form
// =======================
ESR[bits(2) regime] = ESRType value
bits(64) r = value;
case regime of
  when EL1  ESR_EL1 = r;
  when EL2  ESR_EL2 = r;
  when EL3  ESR_EL3 = r;
  otherwise Unreachable();
return;

aarch64/functions/sysregisters/ESRType
type ESRType;

aarch64/functions/sysregisters/FAR

// FAR[] - non-assignment form
// ===========================
bits(64) FAR[bits(2) regime]
bits(64) r;
case regime of
  when EL1  r = FAR_EL1;
  when EL2  r = FAR_EL2;
  when EL3  r = FAR_EL3;
  otherwise Unreachable();
return r;

// FAR[] - non-assignment form
// ===========================
bits(64) FAR[]
return FAR[SITranslationRegime()];

// FAR[] - assignment form
// ==============

FAR[bits(2) regime] = bits(64) value
bits(64) r = value;
case regime of
  when EL1  FAR_EL1 = r;
  when EL2  FAR_EL2 = r;
  when EL3  FAR_EL3 = r;
  otherwise Unreachable();
return;

// FAR[] - assignment form
// ==============

FAR[] = bits(64) value
FAR[SITranslationRegime()] = value;
return;

aarch64/functions/sysregisters/MAIR

// MAIR[] - non-assignment form
// ============================

MAIRType MAIR[bits(2) regime]
bits(64) r;
case regime of
  when EL1  r = MAIR_EL1;
  when EL2  r = MAIR_EL2;
  when EL3  r = MAIR_EL3;
  otherwise Unreachable();
return r;

// MAIR[] - non-assignment form
// ============================

MAIRType MAIR[]
return MAIR[SITranslationRegime()];

aarch64/functions/sysregisters/MAIRType
type MAIRType;

aarch64/functions/sysregisters/SCTLR

// SCTLR[] - non-assignment form
// =============================

SCTLRType SCTLR[bits(2) regime]
bits(64) r;
case regime of
  when EL1  r = SCTLR_EL1;
  when EL2  r = SCTLR_EL2;
  when EL3  r = SCTLR_EL3;
  otherwise Unreachable();
return r;

// SCTLR[] - non-assignment form
// =============================
SCTLRType SCTLR[]
    return SCTLR[S1TranslationRegime()];

aarch64/functions/sysregisters/SCTLRType

type SCTLRType;

aarch64/functions/sysregisters/VBAR

// VBAR[] - non-assignment form
// ============================
bits(64) VBAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = VBAR_EL1;
        when EL2  r = VBAR_EL2;
        when EL3  r = VBAR_EL3;
        otherwise Unreachable();
        return r;
// VBAR[] - non-assignment form
// ============================
bits(64) VBAR[]
    return VBAR[S1TranslationRegime()];

aarch64/functions/system/AArch64.AllocationTagAccessIsEnabled

// AArch64.AllocationTagAccessIsEnabled()
// ======================================
// Check whether access to Allocation Tags is enabled.
boolean AArch64.AllocationTagAccessIsEnabled(AccType acctype)
    bits(2) el = AArch64.AccessUsesEL(acctype);
    if SCR_EL3.ATA == '0' && el IN {EL0, EL1, EL2} then
        return FALSE;
    elsif HCR_EL2.ATA == '0' && el IN {EL0, EL1} && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' then
        return FALSE;
    elsif SCTLR_EL3.ATA == '0' && el == EL3 then
        return FALSE;
    elsif SCTLR_EL2.ATA == '0' && el == EL2 then
        return FALSE;
    elsif SCTLR_EL1.ATA == '0' && el == EL1 then
        return FALSE;
    elsif SCTLR_EL2.ATA0 == '0' && el == EL0 && EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' then
        return FALSE;
    elsif SCTLR_EL1.ATA0 == '0' && el == EL0 && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') then
        return FALSE;
    else
        return TRUE;

aarch64/functions/system/AArch64.CheckSystemAccess

// AArch64.CheckSystemAccess()
// ===========================
AArch64.CheckSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn,
bits(4) crm, bits(3) op2, bits(5) rt, bit read)
if (TSTATE.depth > 0 &&
    !CheckTransactionalSystemAccess(op0, op1, crn, crm, op2, read)) then
    FailTransaction(TMFailure_ERR, FALSE);

return;

aarch64/functions/system/AArch64.ChooseNonExcludedTag

// AArch64.ChooseNonExcludedTag()
// ==============================
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.

bits(4) AArch64.ChooseNonExcludedTag(bits(4) tag_in, bits(4) offset_in, bits(16) exclude)
    bits(4) tag = tag_in;
    bits(4) offset = offset_in;
    if IsOnes(exclude) then
        return '0000';
    if offset == '0000' then
        while exclude < UInt(tag) > == '1' do
            tag = tag + '0001';
        end
        while offset != '0000' do
            offset = offset - '0001';
            tag = tag + '0001';
            while exclude < UInt(tag) > == '1' do
                tag = tag + '0001';
            end
        end
        return tag;

aarch64/functions/system/AArch64.ExecutingBROrBLROrRetInstr

// AArch64.ExecutingBROrBLROrRetInstr()
// ===============================
// Returns TRUE if current instruction is a BR, BLR, RET, B[L]RA[B][Z], or RETA[B].

boolean AArch64.ExecutingBROrBLROrRetInstr()
    if !HaveBTIExt() then return FALSE;
    instr = ThisInstr();
    if instr<31:25> == '1101011' && instr<20:16> == '11111' then
        opc = instr<24:21>;
        return opc != '0101';
    else
        return FALSE;

aarch64/functions/system/AArch64.ExecutingBTIInstr

// AArch64.ExecutingBTIInstr()
// ===========================
// Returns TRUE if current instruction is a BTI.

boolean AArch64.ExecutingBTIInstr()
    if !HaveBTIExt() then return FALSE;
    instr = ThisInstr();
    if instr<31:22> == '1101010101' && instr<21:12> == '011011' then
        CRm = instr<11:8>;
        op2 = instr<7:5>;
        return (CRm == '0100' && op2<0> == '0');
else
    return FALSE;

.LayoutInflater

// AArch64ExecutingERETInstr()
// ==================================================
// Returns TRUE if current instruction is ERET.

boolean AArch64ExecutingERETInstr()
    instr = ThisInstr();
    return instr<31:12> == '11010110100111110000';

LError

// Execute an implementation-defined system instruction with write (source operand).
AArch64ImpDefSysInstr(integer el, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2, integer t);

AArch64ImpDefSysInstrWithResult

// Execute an implementation-defined system instruction with read (result operand).
AArch64ImpDefSysInstrWithResult(integer el, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2);

AArch64ImpDefSysRegRead

// Read from an implementation-defined System register and write the contents of the register to X[t].
AArch64ImpDefSysRegRead(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2, integer t);

AArch64ImpDefSysRegWrite

// Write to an implementation-defined System register.
AArch64ImpDefSysRegWrite(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2, integer t);

AArch64NextRandomTagBit

// AArch64.NextRandomTagBit()
// =========================
// Generate a random bit suitable for generating a random Allocation Tag.

bit AArch64.NextRandomTagBit()
    bits(16) lfsr = RCSR_EL1.SEED;
    bit top = lfsr<5> EOR lfsr<3> EOR lfsr<2> EOR lfsr<0>;
    RCSR_EL1.SEED = top:lfsr<15:1>;
    return top;

LayoutInflater

// AArch64RandomTag()
// =================
// Generate a random Allocation Tag.

bits(4) AArch64RandomTag()
    bits(4) tag;
    for i = 0 to 3
tag<i> = AArch64.NextRandomTagBit();
return tag;

aarch64/functions/system/AArch64.SysInstr

// Execute a system instruction with write (source operand).
AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);

aarch64/functions/system/AArch64.SysInstrWithResult

// Execute a system instruction with read (result operand).
// Writes the result of the instruction to X[t].
AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);

aarch64/functions/system/AArch64.SysRegRead

// Read from a System register and write the contents of the register to X[t].
AArch64.SysRegRead(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);

aarch64/functions/system/AArch64.SysRegWrite

// Write to a System register.
AArch64.SysRegWrite(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);

aarch64/functions/system/BTypeCompatible

boolean BTypeCompatible;

aarch64/functions/system/BTypeCompatible_BTI

// BTypeCompatible_BTI
// ===============
// This function determines whether a given hint encoding is compatible with the current value of
// PSTATE.BTYPE. A value of TRUE here indicates a valid Branch Target Identification instruction.

boolean BTypeCompatible_BTI(bits(2) hintcode)
case hintcode of
  when '00'
    return FALSE;
  when '01'
    return PSTATE.BTYPE != '11';
  when '10'
    return PSTATE.BTYPE != '10';
  when '11'
    return TRUE;

aarch64/functions/system/BTypeCompatible_PACIXSP

// BTypeCompatible_PACIXSP()
// ========================
// Returns TRUE if PACIASP, PACIBSP instruction is implicit compatible with PSTATE.BTYPE,
// FALSE otherwise.

boolean BTypeCompatible_PACIXSP()
if PSTATE.BTYPE IN {'01', '10'} then
  return TRUE;
elsif PSTATE.BTYPE == '11' then
    index = if PSTATE.EL == EL0 then 35 else 36;
    return SCTLR[<index>] == '0';
else
    return FALSE;

aarch64/functions/system/BTypeNext

bits(2) BTypeNext;

aarch64/functions/system/ChooseRandomNonExcludedTag

// The ChooseRandomNonExcludedTag function is used when GCR_EL1.RRND == '1' to generate random
// Allocation Tags.
//
// The resulting Allocation Tag is selected from the set [0,15], excluding any Allocation Tag where
// exclude[tag_value] == 1. If 'exclude' is all Ones, the returned Allocation Tag is '0000'.
//
// This function is permitted to generate a non-deterministic selection from the set of non-excluded
// Allocation Tags. A reasonable implementation is described by the Pseudocode used when
// GCR_EL1.RRND is 0, but with a non-deterministic implementation of NextRandomTagBit(). Implementations
// may choose to behave the same as GCR_EL1.RRND=0.
//
// This function can read RCSR_EL1 and/or write RCSR_EL1 to an IMPLEMENTATION DEFINED value.
// If it is not capable of writing RCSR_EL1.SEED[15:0] to zero from a previous non-zero
// RCSR_EL1.SEED value, it is IMPLEMENTATION DEFINED whether the randomness is significantly
// impacted if RCSR_EL1.SEED[15:0] is set to zero.
bits(4) ChooseRandomNonExcludedTag(bits(16) exclude_in);

aarch64/functions/system/InGuardedPage

boolean InGuardedPage;

aarch64/functions/system/IsHCRXEL2Enabled

// IsHCRXEL2Enabled()
// ==================
// Returns TRUE if access to HCRX_EL2 register is enabled, and FALSE otherwise.
// Indirect read of HCRX_EL2 returns 0 when access is not enabled.
boolean IsHCRXEL2Enabled()
assert(HaveFeatHCX());
if HaveEL(EL3) & SCR_EL3.HXEn == '0' then
    return FALSE;
return EL2Enabled();

aarch64/functions/system/SetBTypeCompatible

// SetBTypeCompatible()
// ====================
// Sets the value of BTypeCompatible global variable used by BTI
SetBTypeCompatible(boolean x)
    BTypeCompatible = x;
aarch64/functions/system/SetBTypeNext

// SetBTypeNext()
// ==============
// Set the value of BTypeNext global variable used by BTI

SetBTypeNext(bits(2) x)
  BTypeNext = x;

aarch64/functions/system/SetInGuardedPage

// SetInGuardedPage()
// ===============
// Global state updated to denote if memory access is from a guarded page.

SetInGuardedPage(boolean guardedpage)
  InGuardedPage = guardedpage;

aarch64/functions/tme/CheckTMEEncabled

// CheckTMEEncabled()
// ================
// Returns TRUE if access to TME instruction is enabled, FALSE otherwise.

CheckTMEEncabled()
  if PSTATE.EL IN {EL0, EL1, EL2} && HaveEL(EL3) then
    if SCR_EL3.TME == '0' then UNDEFINED;
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
      if HCR_EL2.TME == '0' then UNDEFINED;
    return;

aarch64/functions/tme/CheckTransactionalSystemAccess

// CheckTransactionalSystemAccess()
// ================================
// Returns TRUE if an AArch64 MSR, MRS, or SYS instruction is permitted in
// Transactional state, based on the opcode's encoding, and FALSE otherwise.

boolean CheckTransactionalSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, bit read)
  case read:op0:op1:crn:crm:op2 of
    when '0 00 011 0100 xxxx 11x' return TRUE;      // MSR (imm): DAIFSet, DAIFClr
    when '0 01 011 0111 0100 001' return TRUE;      // DC ZVA
    when '0 11 011 0100 0010 00x' return TRUE;      // MSR: NZCV, DAIF
    when '0 11 011 0100 0100 00x' return TRUE;      // MSR: FPCR, FPSR
    when '0 11 000 0100 0110 000' return TRUE;      // MSR: ICC_PMR_EL1
    when '0 11 011 1000 0100 100' return TRUE;      // MRS: PMSWINC_EL0
    when '1 11 xxx 0xxx xxxx xxx' return TRUE;      // MRS: op1=3, CRn=0..7
    when '1 11 xxx 100x xxxx xxx' return TRUE;      // MRS: op1=3, CRn=8..9
    when '1 11 xxx 1010 xxxx xxx' return TRUE;      // MRS: op1=3, CRn=10
    when '1 11 000 1100 1x00 010' return TRUE;      // MRS: op1=3, CRn=12 - ICC_HPPIRx_EL1
    when '1 11 000 1100 1011 011' return TRUE;      // MRS: op1=3, CRn=12 - ICC_RPR_EL1
    when '1 11 xxx 1110 0000 000' return TRUE;      // MRS: op1=3, CRn=13
    when '1 11 xxx 1010 xxx xxx xxx' return TRUE;   // MRS: op1=3, CRn=14
    when '0 01 011 0100 0110 111' return TRUE;      // CPP RCTX
    when 'x 11 xxx 1xxx xxxx xxx xxx' return boolean IMPLEMENTATION_DEFINED; // MRS: op1=3, CRn=11,15
    otherwise return FALSE;                         // all other SYS, SYSL, MRS, MSR
aarch64/functions/tme/CommitTransactionalWrites

// Makes all transactional writes to memory observable by other PEs and reset
// the transactional read and write sets.
CommitTransactionalWrites();

aarch64/functions/tme/DiscardTransactionalWrites

// Discards all transactional writes to memory and reset the transactional
// read and write sets.
DiscardTransactionalWrites();

aarch64/functions/tme/FailTransaction

// FailTransaction()
// =============
FailTransaction(TMFailure cause, boolean retry)
  FailTransaction(cause, retry, FALSE, Zeros(15));
  return;

// FailTransaction()
// =============
// Exits Transactional state and discards transactional updates to registers
// and memory.
FailTransaction(TMFailure cause, boolean retry, boolean interrupt, bits(15) reason)
  assert !retry || !interrupt;
  if HaveBRBExt() && BranchRecordAllowed(PSTATE.EL) then BRBFCR_EL1.LASTFAILED = '1';
  DiscardTransactionalWrites();
  // For trivial implementation no transaction checkpoint was taken
  if cause != TMFailure_TRIVIAL then RestoreTransactionCheckpoint();
  ClearExclusiveLocal(ProcessorID());
  bits(64) result = Zeros(64);
  result<23> = if interrupt then '1' else '0';
  result<15> = if retry && !interrupt then '1' else '0';
  case cause of
    when TMFailure_TRIVIAL result<24> = '1';
    when TMFailure_DBG result<22> = '1';
    when TMFailure_NEST result<21> = '1';
    when TMFailure_SIZE result<20> = '1';
    when TMFailure_ERR result<19> = '1';
    when TMFailure_IMP result<18> = '1';
    when TMFailure_MEM result<17> = '1';
    when TMFailure_CNCL result<16> = '1'; result<14:0> = reason;
    TSTATE.depth = 0;
    X[TSTATE.Rt, 64] = result;
    boolean branch_conditional = FALSE;
    BranchTo(TSTATE.nPC, BranchType_TMFAIL, branch_conditional);
    EndOfInstruction();
    return;

aarch64/functions/tme/MemHasTransactionalAccess

// MemHasTransactionalAccess()
// ===========================
// Function checks if transactional accesses are not supported for an address
// range or memory type.

boolean MemHasTransactionalAccess(MemoryAttributes memattrs)
if ((memattrs.shareability == Shareability_ISH ||
    memattrs.shareability == Shareability_OSH) &&
    memattrs.memtype == MemType_Normal &&
    memattrs.inner.attrs == MemAttr_WB &&
    memattrs.inner.hints == MemHint_RWA &&
    memattrs.inner.transient == FALSE &&
    memattrs.outer.hints == MemHint_RWA &&
    memattrs.outer.attrs == MemAttr_WB &&
    memattrs.outer.transient == FALSE) then
    return TRUE;
else
    return boolean IMPLEMENTATION_DEFINED "Memory Region does not support Transactional access";

aarch64/functions/tme/RestoreTransactionCheckpoint

// RestoreTransactionCheckpoint()
// -----------------------------
// Restores part of the PE registers from the transaction checkpoint.

RestoreTransactionCheckpoint()
  SP[] = TSTATE.SP;
  ICC_PMR_EL1 = TSTATE.ICC_PMR_EL1;
  PSTATE.<N,Z,C,V> = TSTATE.nzcv;
  PSTATE.<D,A,I,F> = TSTATE.<D,A,I,F>;
  for n = 0 to 30
    X[n, 64] = TSTATE.X[n];

  constant integer VL = CurrentVL;
  constant integer PL = VL DIV 8;
  if IsFPEnabled(PSTATE.EL) then
    if IsSVEEnabled(PSTATE.EL) then
      for n = 0 to 31
        Z[n, VL] = TSTATE.Z[n]<VL-1:0>;
      for n = 0 to 15
        P[n, PL] = TSTATE.P[n]<PL-1:0>;
        FFR[PL] = TSTATE.FFR<PL-1:0>;
    else
      for n = 0 to 31
        V[n, 128] = TSTATE.Z[n]<127:0>;
    FPCR = TSTATE.FPCR;
    FPSR = TSTATE.FPSR;
  return;

aarch64/functions/tme/StartTrackingTransactionalReadsWrites

// Starts tracking transactional reads and writes to memory.
StartTrackingTransactionalReadsWrites();

aarch64/functions/tme/TMFailure

enumeration TMFailure {
  TMFailure_CNCL, // Executed a TCANCEL instruction
  TMFailure_DBG, // A debug event was generated
  TMFailure_ERR, // A non-permissible operation was attempted
  TMFailure_NEST, // The maximum transactional nesting level was exceeded
  TMFailure_SIZE, // The transactional read or write set limit was exceeded
  TMFailure_MEM, // A transactional conflict occurred
  TMFailure_TRIVIAL, // Only a TRIVIAL version of TM is available
TMFailure_IMP    // Any other failure cause
};

aarch64/functions/tme/TMState

type TMState is (    
    integer depth,   // Transaction nesting depth
    integer Rt,     // TSTART destination register
    bits(64) nPC,   // Fallback instruction address
    array[0..30] of bits(64) X, // General purpose registers
    array[0..31] of bits(MAX_VL) Z, // Vector registers
    array[0..15] of bits(MAX_PL) P, // Predicate registers
    bits(MAX_PL) FFR, // First Fault Register
    bits(64) SP, // Stack Pointer at current EL
    bits(64) FPCR, // Floating-point Control Register
    bits(64) FPSR, // Floating-point Status Register
    bits(64) ICC_PMR_EL1, // Interrupt Controller Interrupt Priority Mask Register
    bits(4) nzcv, // Condition flags
    bits(1) D, // Debug mask bit
    bits(1) A, // SError interrupt mask bit
    bits(1) I, // IRQ mask bit
    bits(1) F, // FIQ mask bit
)

aarch64/functions/tme/TSTATE

TMState TSTATE;

aarch64/functions/tme/TakeTransactionCheckpoint

// TakeTransactionCheckpoint()
// --------------------------
// Captures part of the PE registers into the transaction checkpoint.

TakeTransactionCheckpoint()
TSTATE.SP = SP[ ];
TSTATE.ICC_PMR_EL1 = ICC_PMR_EL1;
TSTATE.nzcv = PSTATE.<N,Z,C,V>;
TSTATE.<D,A,I,F> = PSTATE.<D,A,I,F>;
for n = 0 to 30
  TSTATE.X[n] = X[n, 64];

constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if IsFPEnabled(PSTATE.EL) then
  if IsSVEEnabled(PSTATE.EL) then
    for n = 0 to 31
      TSTATE.Z[n]<VL-1:0> = Z[n, VL];
    for n = 0 to 15
      TSTATE.P[n]<PL-1:0> = P[n, PL];
    TSTATE.FFR<PL-1:0> = FFR[PL];
  else
    for n = 0 to 31
      TSTATE.Z[n]<127:0> = V[n, 128];
    TSTATE.FPCR = FPCR;
    TSTATE.FPSR = FPSR;
  return;
aarch64/functions/tme/TransactionStartTrap

// TransactionStartTrap()
// ======================
// Traps the execution of TSTART instruction.

TransactionStartTrap(integer dreg)
  bits(2) targetEL;
  bits(64) preferred_exception_return = ThisInstrAddr(64);
  vect_offset = 0x0;

  exception = Exception Syndrome(Exception_TSTARTAccessTrap);
  exception.syndrome<9:5> = dreg<4:0>;

  if UInt(PSTATE.EL) > UInt(EL1) then
    targetEL = PSTATE.EL;
  elsif EL2Enabled() && HCR_EL2.TGE == '1' then
    targetEL = EL2;
  else
    targetEL = EL1;
  end;
  AArch64.TakeException(targetEL, exception, preferred_exception_return, vect_offset);

J1.1.4  aarch64/instrs

This section includes the following pseudocode functions:
- aarch64/instrs/branch/eret/AArch64.ExceptionReturn on page J1-11176.
- aarch64/instrs/countop/CountOp on page J1-11176.
- aarch64/instrs/extendreg/DecodeRegExtend on page J1-11177.
- aarch64/instrs/extendreg/ExtendReg on page J1-11177.
- aarch64/instrs/extendreg/ExtendType on page J1-11177.
- aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp on page J1-11177.
- aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUnaryOp on page J1-11178.
- aarch64/instrs/float/convert/fpconvop/FPConvOp on page J1-11178.
- aarch64/instrs/integer/bitfield/bfxpreferred/BFXPreferred on page J1-11178.
- aarch64/instrs/integer/bitmasks/AltDecodeBitMasks on page J1-11178.
- aarch64/instrs/integer/bitmasks/DecodeBitMasks on page J1-11180.
- aarch64/instrs/integer/logical/movwprefereed/MoveWidePreferred on page J1-11180.
- aarch64/instrs/integer/shiftreg/DecodeShift on page J1-11181.
- aarch64/instrs/integer/shiftreg/ShiftReg on page J1-11181.
- aarch64/instrs/integer/ShiftType on page J1-11181.
- aarch64/instrs/logicalop/LogicalOp on page J1-11181.
- aarch64/instrs/memory/memop/MemAtomicOp on page J1-11181.
- aarch64/instrs/memory/memop/MemOp on page J1-11182.
- aarch64/instrs/memory/prefetch/Prefetch on page J1-11182.
- aarch64/instrs/system/barriers/barrierop/MemBarrierOp on page J1-11182.
- aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField on page J1-11183.
- aarch64/instrs/system/sysops/at/AArch64.AT on page J1-11183.
- aarch64/instrs/system/sysops/at/AArch64.EncodePAR on page J1-11184.
- aarch64/instrs/system/sysops/at/AArch64.PARFauluStatus on page J1-11185.
- aarch64/instrs/system/sysops/dc/AArch64.DC on page J1-11185.
- aarch64/instrs/system/sysops/ic/AArch64.IC on page J1-11187.
- aarch64/instrs/system/sysops/predictionrestrict/RestrictPrediction on page J1-11188.
- aarch64/instrs/system/sysops/tlbi/AArch32.DTLBI_ALL on page J1-11192.
- aarch64/instrs/system/sysops/tlbi/AArch32.DTLBI_ASID on page J1-11193.
- aarch64/instrs/system/sysops/tlbi/AAArch32.DTLBI_VA on page J1-11193.
- aarch64/instrs/system/sysops/tlbi/AArch32.ITLBI_ALL on page J1-11194.
- aarch64/instrs/system/sysops/tlbi/AArch32.ITLBI_ASID on page J1-11194.
- aarch64/instrs/system/sysops/tlbi/AArch32.ITLBI_VA on page J1-11195.
- aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_ALL on page J1-11195.
- aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_ASID on page J1-11196.
- aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_IP AS2 on page J1-11196.
- aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VA on page J1-11197.
- aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VAA on page J1-11197.
- aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VMALL on page J1-11197.
- aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VMALLS12 on page J1-11198.
- aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_ALL on page J1-11199.
- aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_ASID on page J1-11199.
- aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_IPAS2 on page J1-11200.
- aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_PAALL on page J1-11200.
- aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_RIPAS2 on page J1-11201.
- aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_RPA on page J1-11201.
- aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_RVA on page J1-11203.
- aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_RVAA on page J1-11203.
- aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VA on page J1-11204.
- aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VAA on page J1-11204.
- aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VMALL on page J1-11205.
- aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VMALLS12 on page J1-11205.
- aarch64/instrs/system/sysops/tlbi/ASID_NONE on page J1-11206.
- aarch64/instrs/system/sysops/tlbi/Broadcast on page J1-11206.
- aarch64/instrs/system/sysops/tlbi/DecodeTLBITG on page J1-11206.
- aarch64/instrs/system/sysops/tlbi/GPTTLBIMatch on page J1-11206.
- aarch64/instrs/system/sysops/tlbi/HasLargeAddress on page J1-11207.
- aarch64/instrs/system/sysops/tlbi/TLB Level on page J1-11207.
- aarch64/instrs/system/sysops/tlbi/TLBIMatch on page J1-11207.
- aarch64/instrs/system/sysops/tlbi/TLBIMemAttr on page J1-11209.
- aarch64/instrs/system/sysops/tlbi/TLBIOp on page J1-11209.
- aarch64/instrs/system/sysops/tlbi/TLBIRange on page J1-11209.
- aarch64/instrs/system/sysops/tlbi/TLBIRecord on page J1-11210.
- aarch64/instrs/system/sysops/tlbi/VMID on page J1-11211.
- aarch64/instrs/system/sysops/tlbi/VMID_NONE on page J1-11211.
- aarch64/instrs/vector/logical/immediateop/ImmediateOp on page J1-11211.
- aarch64/instrs/vector/reduce/reduceop/Reduce on page J1-11211.
- aarch64/instrs/vector/reduce/reduceop/ReduceOp on page J1-11212.
aarch64/instrs/branch/eret/AArch64.ExceptionReturn

// AArch64.ExceptionReturn()
// =========================

AArch64.ExceptionReturn(bits(64) new_pc_in, bits(64) spsr)
bits(64) new_pc = new_pc_in;
if HaveTME() && TSTATE.depth > 0 then
  FailTransaction(TMFailure_ERR, FALSE);

if HaveIESB() then
  sync_errors = SCTLR[].IESB == '1';
if HaveDoubleFaultExt() then
  sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && PSTATE.EL == EL3);
if sync_errors then
  SynchronizeErrors();
  iesb_req = TRUE;
  TakeUnmaskedPhysicalErrorInterrupts(iesb_req);
  SynchronizeContext();

// Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
bits(2) source_el = PSTATE.EL;
boolean illegal_psr_state = IllegalExceptionReturn(spsr);
SetPSTATEFromPSR(spsr, illegal_psr_state);
ClearExclusiveLocal(ProcessorID());
SendEventLocal();

if illegal_psr_state && spsr<4> == '1' then
  // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
  new_pc<63:32> = bits(32) UNKNOWN;
  new_pc<1:0> = bits(2) UNKNOWN;
elsif UsingAArch32() then                // Return to AArch32
  // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the
  // target instruction set state
  if PSTATE.T == '1' then
    new_pc<0> = '0';                 // T32
    else
    new_pc<1:0> = '00';              // A32
    else                          // Return to AArch64
  // ELR_ELx[63:56] might include a tag
  new_pc = AArch64.BranchAddr(new_pc, PSTATE.EL);
if HaveBRBExt() then
  BRBEExceptionReturn(new_pc, source_el);

if UsingAArch32() then
  if HaveSME() && PSTATE.SM == '1' then ResetSVEState();
  // 32 most significant bits are ignored.
  boolean branch_conditional = FALSE;
  BranchTo(new_pc<31:0>, BranchType_ERET, branch_conditional);
else
  BranchToAddr(new_pc, BranchType_ERET);

CheckExceptionCatch(FALSE);              // Check for debug event on exception return

aarch64/instrs/countop/CountOp

// DecodeRegExtend()
// =============
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
  case op of
    when '000' return ExtendType_UXTB;
    when '001' return ExtendType_UXTH;
    when '010' return ExtendType_UXTW;
    when '011' return ExtendType_UXTX;
    when '100' return ExtendType_SXTB;
    when '101' return ExtendType_SXTH;
    when '110' return ExtendType_SXTW;
    when '111' return ExtendType_SXTX;

// ExtendReg()
// ===========
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType exttype, integer shift, integer N)
  assert shift >= 0 && shift <= 4;
  bits(N) val = X[reg, N];
  boolean unsigned;
  integer len;

  case exttype of
    when ExtendType_SXTB unsigned = FALSE; len = 8;
    when ExtendType_SXTH unsigned = FALSE; len = 16;
    when ExtendType_SXTW unsigned = FALSE; len = 32;
    when ExtendType_SXTX unsigned = FALSE; len = 64;
    when ExtendType_UXTB unsigned = TRUE;  len = 8;
    when ExtendType_UXTH unsigned = TRUE;  len = 16;
    when ExtendType_UXTW unsigned = TRUE;  len = 32;
    when ExtendType_UXTX unsigned = TRUE;  len = 64;

 :len = Min(len, N - shift);
  return Extend(val<len-1:0> : Zeros(shift), N, unsigned);
aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUnaryOp

 enumeration FPUnaryOp   
{FPUnaryOp_ABS, FPUnaryOp_MOV,  
FPUnaryOp_NEG, FPUnaryOp_SQRT};

aarch64/instrs/float/convert/fpconvop/FPConvOp

 enumeration FPConvOp    
{FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,  
FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF  
, FPConvOp_CVT_FtoI_JS};

aarch64/instrs/integer/bitfield/bfxpreferred/BFXPreferred

 boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr) 

 // must not match UBFIX/SBFIX alias  
 if UInt(imms) < UInt(immr) then  
 return FALSE;

 // must not match LSR/ASR/LSL alias (imms == 31 or 63)  
 if imms == sf:'11111' then  
 return FALSE;

 // must not match UXTx/SXTx alias  
 if immr == '000000' then  
 // must not match 32-bit UXT[BH] or SXT[BH]  
 if sf == '0' && imms IN {'000111', '001111'} then  
 return FALSE;

 // must not match 64-bit SXT[BHW]  
 if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then  
 return FALSE;

 // must be UBFIX/SBFIX alias  
 return TRUE;

aarch64/instrs/integer/bitmasks/AltDecodeBitMasks

 (bits(M), bits(M)) AltDecodeBitMasks(bit immN, bits(6) imms, bits(6) immr,  
 boolean immediate, integer M)

 bits(64) tmask, wmask;  
 bits(6) tmask_and, wmask_and;  
 bits(6) tmask_or, wmask_or;  
 bits(6) levels;

 // Compute log2 of element size  
 // 2^len must be in range [2, M]  
 len = HighestSetBit(immN:NOT(imms));  
 if len < 1 then UNDEFINED;  
 assert M >= (1 << len);
// Determine s, r and s - r parameters
levels = ZeroExtend(Ones(len), 6);

// For logical immediates an all-ones value of s is reserved
// since it would generate a useless all-ones result (many times)
if immediate && (imms AND levels) == levels then
    UNDEFINED;

s = UInt(imms AND levels);
r = UInt(immr AND levels);
diff = s - r;    // 6-bit subtract with borrow

// Compute "top mask"
tmask_and = diff<5:0> OR NOT(levels);
tmask_or = diff<5:0> AND levels;

tmask = Ones(64);
tmask = ((tmask
    AND Replicate(Replicate(tmask_and<0>, 1) : Ones(1), 32))
    OR Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));
// optimization of first step:
// tmask = Replicate(tmask_and<0> : '1', 32);
tmask = ((tmask
    AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
    OR Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
tmask = ((tmask
    AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
    OR Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
tmask = ((tmask
    AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
    OR Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
tmask = ((tmask
    AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
    OR Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));
tmask = ((tmask
    AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
    OR Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

// Compute "wraparound mask"
wmask_and = immr OR NOT(levels);
wmask_or = immr AND levels;

wmask = Zeros(64);
wmask = ((wmask
    AND Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
    OR Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));
// optimization of first step:
// wmask = Replicate(wmask_or<0> : '0', 32);
wmask = ((wmask
    AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
    OR Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));
wmask = ((wmask
    AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
    OR Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));
wmask = ((wmask
    AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
    OR Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));
wmask = ((wmask
    AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
    OR Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));
wmask = ((wmask
    AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
    OR Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1));

if diff<6> != '0' then // borrow from s - r
    wmask = wmask AND tmask;
else
    wmask = wmask OR tmask;
return (wmask<M-1:0>, tmask<M-1:0>);

aarch64/instrs/integer/bitmasks/DecodeBitMasks

// DecodeBitMasks()
// ================
// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure
(bits(M), bits(M)) DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr,
  boolean immediate, integer M)

  bits(M) tmask, wmask;
  bits(6) levels;

  // Compute log2 of element size
  // 2^len must be in range [2, M]
  len = HighestSetBit(immN:NOT(imms));
  if len < 1 then UNDEFINED;
  assert M >= (1 << len);

  // Determine s, r and s - r parameters
  levels = ZeroExtend(Ones(len), 6);

  // For logical immediates an all-ones value of s is reserved
  // since it would generate a useless all-ones result (many times)
  if immediate && (imms AND levels) == levels then
    UNDEFINED;
  s = UInt(imms AND levels);
  r = UInt(immr AND levels);
  diff = s - r;    // 6-bit subtract with borrow

  esize = 1 << len;
  d = UInt(diff<len-1:0>);
  welem = ZeroExtend(Ones(s + 1), esize);
  telem = ZeroExtend(Ones(d + 1), esize);
  wmask = Replicate(ROR(welem, r));
  tmask = Replicate(telem);

  return (wmask, tmask);

aarch64/instrs/integer/ins-ext/insert/movewide/movewideop/MoveWideOp


aarch64/instrs/integer/logical/movwpreferred/MoveWidePreferred

// MoveWidePreferred()
// ================
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)

  integer s = UInt(imms);
  integer r = UInt(immr);
  integer width = if sf == '1' then 64 else 32;

  // element size must equal total immediate size
  if sf == '1' & & (!((immN:imms) IN {'1xxxxxx'}) then
    return FALSE;
  if sf == '0' & & (!((immN:imms) IN {'00xxxxx'}) then
    return FALSE;
// for MOVZ must contain no more than 16 ones
if s < 16 then
  // ones must not span halfword boundary when rotated
  return (-r MOD 16) <= (15 - s);

// for MOVN must contain no more than 16 zeros
if s >= width - 15 then
  // zeros must not span halfword boundary when rotated
  return (r MOD 16) <= (s - (width - 15));

return FALSE;

aarch64/instrs/integer/shiftreg/DecodeShift

// DecodeShift()
// =============
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
  case op of
    when '00'  return ShiftType_LSL;
    when '01'  return ShiftType_LSR;
    when '10'  return ShiftType_ASR;
    when '11'  return ShiftType_ROR;
  end case;

aarch64/instrs/integer/shiftreg/ShiftReg

// ShiftReg()
// =========
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType shiftype, integer amount, integer N)
  bits(N) result = X[reg, N];
  case shiftype of
    when ShiftType_LSL result = LSL(result, amount);
    when ShiftType_LSR result = LSR(result, amount);
    when ShiftType_ASR result = ASR(result, amount);
    when ShiftType_ROR result = ROR(result, amount);
  end case;
  return result;

aarch64/instrs/integer/shiftreg/ShiftType

enumeration ShiftType   {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};

aarch64/instrs/logicalop/LogicalOp

enumeration LogicalOp   {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};

aarch64/instrs/memory/memop/MemAtomicOp

Aarch64/instrs/memory/memop/MemOp

enumeration MemOp { MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH; }

Aarch64/instrs/memory/prefetch/Prefetch

// Prefetch()
// =========

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)

  PrefetchHint hint;
  integer target;
  boolean stream;

  case prfop<4:3> of
    when '00' hint = Prefetch_READ;   // PLD: prefetch for load
    when '01' hint = Prefetch_EXEC;   // PLI: preload instructions
    when '10' hint = Prefetch_WRITE;  // PST: prepare for store
    when '11' return;                 // unallocated hint
  target = UInt(prfop<2:1>);         // target cache level
  stream = (prfop<0> != '0');        // streaming (non-temporal)
  Hint_Prefetch(address, hint, target, stream);
  return;

Aarch64/instrs/system/barriers/barrierop/MemBarrierOp

enumeration MemBarrierOp   {  MemBarrierOp_DSB         // Data Synchronization Barrier
  , MemBarrierOp_DMB         // Data Memory Barrier
  , MemBarrierOp_ISB         // Instruction Synchronization Barrier
  , MemBarrierOp_SSSBB       // Speculative Synchronization Barrier to VA
  , MemBarrierOp_PSSBB       // Speculative SynchronizationBarrier to PA
  , MemBarrierOp_SB          // Speculation Barrier
  };

Aarch64/instrs/system/hints/syshintop/SystemHintOp

enumeration SystemHintOp {
  SystemHintOp_NOP,
  SystemHintOp_YIELD,
  SystemHintOp_WFE,
  SystemHintOp_WFI,
  SystemHintOp_SEV,
  SystemHintOp_SEVL,
  SystemHintOp_DGH,
  SystemHintOp_ESEB,
  SystemHintOp_PS5B,
  SystemHintOp_TSB,
  SystemHintOp_BT1,
  SystemHintOp_WFE1,
  SystemHintOp_WF1,
  SystemHintOp_CSRB
  };

MemAtomicOp UMIN,
MemAtomicOp SWP;
aarch64/intrs/system/register/cpsr/pstatefield/PSTATEField

enumeration PSTATEField

- PSTATEField_DAIFSet
- PSTATEField_DAIFClr
- PSTATEField_PAN
- PSTATEField_UAO
- PSTATEField_DIT
- PSTATEField_SSBS
- PSTATEField_TCO
- PSTATEField_SVCRSM
- PSTATEField_SVCRZA
- PSTATEField_SVCRSMZA
- PSTATEField_ALLINT
- PSTATEField_SP

aarch64/intrs/system/sysops/at/AArch64.AT

// AArch64.AT()
// ============
// Perform address translation as per AT instructions.

AArch64.AT(bits(64) address, TranslationStage stage_in, bits(2) el_in, ATAccess ataccess)

TranslationStage stage = stage_in;
bits(2) el = el_in;
if HaveRME() && PSTATE.EL == EL3 && SCR_EL3.<NSE,NS> == '10' && el != EL3 then UNDEFINED;
// For stage 1 translation, when HCR_EL2.<E2H, TGE> is {1,1} and requested EL is EL1,
// the EL2&0 translation regime is used.
if HCR_EL2.<E2H, TGE> == '11' && el == EL1 && stage == TranslationStage_1 then
  el = EL2;
if HaveEL(EL3) && stage == TranslationStage_12 && !EL2Enabled() then
  stage = TranslationStage_1;
acctype = if ataccess IN {ATAccess_Read, ATAccess_Write} then AccType_AT else AccType_ATPAN;
isset = ataccess IN {ATAccess_WritePAN, ATAccess_Write};
aligned = TRUE;
ispriv  = el != EL0;

fault = NoFault();
fault.acctype = acctype;
fault.write = isset;

Regime regime;
if stage == TranslationStage_12 then
  regime = Regime_EL10;
else
  regime = TranslationRegime(el, acctype);

AddressDescriptor addrdesc;
ss = SecurityStateAtEL(el);
if (el == EL0 && ELUsingAArch32(EL1)) || (el != EL0 && ELUsingAArch32(EL)) then
  if regime == Regime_EL2 || TTBCR.EAE == '1' then
    (fault, addrdesc) = AArch32.S1TranslateLD(fault, regime, ss, address<31:0>, acctype, aligned, isset, ispriv);
  else
    (fault, addrdesc) = AArch32.S1TranslateSD(fault, regime, ss, address<31:0>, acctype, aligned, isset, ispriv);
else
  (fault, addrdesc) = AArch64.S1Translate(fault, regime, ss, address, acctype, aligned, isset, ispriv);

if stage == TranslationStage_12 && fault.statuscode == Fault_None then
  boolean s2fs1walk;
  boolean s1aarch64;
  if ELUsingAArch32(EL1) && regime == Regime_EL10 && EL2Enabled() then
    addrdesc.vaddress = ZeroExtend(address, 64);
    s2fs1walk = FALSE;
(fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, ss, s2fs1walk, acctype, aligned, iswrite, ispriv);

elsif regime == Regime_EL10 && EL2Enabled() then
    s1aarch64 = TRUE;
    s2fs1walk = FALSE;
    (fault, addrdesc) = AArch64.S2Translate(fault, addrdesc, s1aarch64, ss, s2fs1walk, acctype, aligned, iswrite, ispriv);

is_ATS1Ex = stage != TranslationStage_12;
if fault.statuscode != Fault_None then
    addrdesc = CreateFaultyAddressDescriptor(address, fault);
    // Take an exception on:
    // * A Synchronous External abort occurs on translation table walk
    // * A stage 2 fault occurs on a stage 1 walk
    // * A GPC Exception (FEAT_RME)
    // * A GPF from ATS1E{1,0} when executed from EL1 and HCR_EL2.GPF == '1' (FEAT_RME)
    if (IsExternalAbort(fault) ||
        (PSTATE.EL == EL1 && fault.s2fs1walk) ||
        (HaveRME() && fault.gpcf.gpf != GPCF_None && (ReportAsGPCException(fault) ||
        (HCR_EL2.GPF == '1' && PSTATE.EL == EL1 && el IN {EL1, EL0} && is_ATS1Ex))
    ) then
        PAR_EL1 = bits(64) UNKNOWN;
        AArch64.Abort(address, addrdesc.Fault);
    return;

aarch64/instrs/system/sysops/at/AArch64.EncodePAR

// AArch64.EncodePAR()
// ===============
// Encode PAR register with result of translation.

AArch64.EncodePAR(Regime regime, boolean is_ATS1Ex, AddressDescriptor addrdesc)
    PAR_EL1 = Zeros(64);
    paspace = addrdesc.paddress.paspace;
    if !IsFault(addrdesc) then
        PAR_EL1.F = '0';
    if HaveRME() then
        if regime == Regime_EL3 then
            case paspace of
                when PAS_Secure  PAR_EL1.<NSE,NS> = '00';
                when PAS_NonSecure PAR_EL1.<NSE,NS> = '01';
                when PAS_Root     PAR_EL1.<NSE,NS> = '10';
                when PAS_Realm    PAR_EL1.<NSE,NS> = '11';
            elseif SecurityStateForRegime(regime) == SS_Secure then
                PAR_EL1.<NSE,NS> = bit UNKNOWN;
            elseif SecurityStateForRegime(regime) == SS_Realm then
                if regime == Regime_EL10 && is_ATS1Ex then
                    PAR_EL1.<NSE,NS> = bit UNKNOWN;
                else
                    PAR_EL1.<NSE,NS> = bit UNKNOWN;
                end
            else
                PAR_EL1.<NSE,NS> = '1';
            end
        elseif SecurityStateForRegime(regime) == SS_REALM then
            if regime == Regime_EL10 && is_ATS1Ex then
                PAR_EL1.<NSE,NS> = bit UNKNOWN;
            else
                PAR_EL1.<NSE,NS> = bit UNKNOWN;
            end
        else
            PAR_EL1.<NSE,NS> = '1';
        end
        if SecurityStateForRegime(regime) == SS_Secure then
            PAR_EL1.<NHE,NS> = '0';
        end
    end

PAR_EL1.NS = if paspace == PAS_Secure then '0' else '1';
else
  PAR_EL1.NS = bit UNKNOWN;
PAR_EL1.SH   = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
PAR_EL1.PA   = addrdesc.paddress.address<52-1:12>;
PAR_EL1.ATTR = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
PAR_EL1<10> = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";
else
  PAR_EL1.F   = '1';
  PAR_EL1.FST = AArch64.PARFaultStatus(addrdesc.fault);
  PAR_EL1.PTW = if addrdesc.fault.s2fs1walk then '1' else '0';
  PAR_EL1.S   = if addrdesc.fault.secondstage then '1' else '0';
  PAR_EL1<11> = '1'; // RES1
  PAR_EL1<63:48> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";
return;

aarch64/intrs/system/sysops/at/AArch64.PARFaultStatus

// AArch64.PARFaultStatus()
// ========================
// Fault status field decoding of 64-bit PAR.

bits(6) AArch64.PARFaultStatus(FaultRecord fault)
bits(6) fst;
if fault.statuscode == Fault_Domain then
  // Report Domain fault
  assert fault.level IN {1,2};
  fst<1:0> = if fault.level == 1 then '01' else '10';
  fst<5:2> = '1111';
else
  fst = EncodeLDFSC(fault.statuscode, fault.level);
return fst;

aarch64/intrs/system/sysops/dc/AArch64.DC

// AArch64.DC()
// ============
// Perform Data Cache Operation.

AArch64.DC(bits(64) regval, CacheType cachetype, CacheOp cacheop, CacheOpScope opscope_in)
  CacheOpScope opscope = opscope_in;
  AccType acctype = AccType_DC;
  CacheRecord cache;
  cache.acctype = acctype;
  cache.cachetype = cachetype;
  cache.cacheop = cacheop;
  cache.opscope = opscope;
if opscope == CacheOpScope_SetWay then
  ss = SecurityStateAtEL(PSTATE.EL);
  cache.cpas = CPASAtSecurityState(ss);
  cache.shareability = Shareability_NSH;
  (cache.set, cache.way, cache.level) = DecodeSW(regval, cachetype);
  if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
    (HCR_EL2.SWIO == '1' || HCR_EL2.<DC,VM> != '00')) then
    cache.cacheop = CacheOp_CleanInvalidate;
  CACHE_OP(cache);
  return;
if EL2Enabled() && !IsInHost() then
  if PSTATE.EL IN {EL0, EL1} then
cache.is_vmid_valid = TRUE;
    cache.vmid          = VMID[];
else
    cache.is_vmid_valid = FALSE;
else
    cache.is_vmid_valid = FALSE;

if PSTATE.EL == EL0 then
    cache.is_asid_valid = TRUE;
    cache.asid          = ASID[];
else
    cache.is_asid_valid = FALSE;

if opscope == CacheOpScope_PoDP && boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoDP" then
    opscope = CacheOpScope_PoP;
if opscope == CacheOpScope_PoP && boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoP" then
    opscope = CacheOpScope_PoC;
need_translate = DCInstNeedsTranslation(opscope);
iswrite = cacheop == CacheOp_Invalidate;
vaddress = regval;

size = 0; // by default no watchpoint address
if iswrite then
    size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
    assert size >= 4*(2^UInt(CTR_EL0.DminLine)) & size <= 2048;
    assert UInt(size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
    vaddress = Align(regval, size);

cache.translated = need_translate;
cache.vaddress = vaddress;

if need_translate then
    wasaligned = TRUE;
    memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);
    if IsFault(memaddrdesc) then
        AArch64.Abort(regval, memaddrdesc.fault);

    memattrs = memaddrdesc.memattrs;
cache.paddress = memaddrdesc.paddress;
cache.cpas = CPASAtPAS(memaddrdesc.paddress.paspace);
    cache.shareability = memattrs.shareability;
else
    cache.shareability = Shareability_NSH;
else
    cache.shareability = Shareability UNKNOWN;
cache.paddress = FullAddress UNKNOWN;

if cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.<DC,VM> != '00' then
    cache.cacheop = CacheOp_CleanInvalidate;

CACHE_OP(cache);
return;

aarch64/instrs/system/sysops/dc/AArch64.MemZero

// AArch64.MemZero()
// ================

AArch64.MemZero(bits(64) regval, CacheType cachetype)

AccType acctype = AccType_DCZVA;
boolean iswrite = TRUE;
boolean wasaligned = TRUE;

integer size = 4*(2^(UInt(DCZID_EL0.BS)));

bits(64) vaddress = Align(regval, size);

memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite,
            wasaligned, size);

if IsFault(memaddrdesc) then
    if IsDebugException(memaddrdesc.fault) then
        AArch64.Abort(vaddress, memaddrdesc.fault);
    else
        AArch64.Abort(regval, memaddrdesc.fault);
    else
        if cachetype == CacheType_Data then
            AArch64.DataMemZero(regval, vaddress, memaddrdesc, size);
        elsif cachetype == CacheType_Tag then
            if HaveMTEExt() then AArch64.TagMemZero(vaddress, size);
        elsif cachetype == CacheType_Data_Tag then
            if HaveMTEExt() then AArch64.TagMemZero(vaddress, size);
            AArch64.DataMemZero(regval, vaddress, memaddrdesc, size);
        return;

aarch64/instrs/system/sysops/ic/AArch64.IC

// AArch64.IC()
// ============
// Perform Instruction Cache Operation.

AArch64.IC(CacheOpScope opscope)
    regval = bits(64) UNKNOWN;
    AArch64.IC(regval, opscope);

// AArch64.IC()
// ============
// Perform Instruction Cache Operation.

AArch64.IC(bits(64) regval, CacheOpScope opscope)
    CacheRecord cache;
    AccType acctype = AccType_IC;

    cache.acctype = acctype;
    cache.cachetype = CacheType_Instruction;
    cache.cacheop = CacheOp_Invalidate;
    cache.opscope = opscope;

    if opscope IN {CacheOpScope_ALLU, CacheOpScope_ALLUIS} then
        ss = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);
        if (opscope == CacheOpScope_ALLUIS || (opscope == CacheOpScope_ALLU && PSTATE.EL == EL1
            && EL2Enabled() && HCR_EL2.FB == '1')) then
            cache.shareability = Shareability_ISH;
        else
            cache.shareability = Shareability_NSH;
        regval = regval;
        CACHE_OP(cache);
    else
        assert opscope == CacheOpScope_PoU;

        if EL2Enabled() && !IsInHost() then
            if PSTATE.EL IN {EL0, EL1} then
                cache.is_vmid_valid = TRUE;
                cache.vmid = VMID[];
            else
                cache.is_vmid_valid = FALSE;
            else
                ...
cache.is_vmid_valid = FALSE;
if PSTATE.EL == EL0 then
    cache.is_asid_valid = TRUE;
    cache.asid = ASID[];
else
    cache.is_asid_valid = FALSE;

bits(64) vaddress = regval;
need_translate = ICInstNeedsTranslation(opscope);

cache.vaddress = regval;
cache.shareability = Shareability_NSH;
cache.translated = need_translate;

if !need_translate then
    cache.paddress = FullAddress UNKNOWN;
    CACHE_OP(cache);
    return;

iswrite = FALSE;
wasaligned = TRUE;
size = 0;
memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);

if IsFault(memaddrdesc) then
    AArch64.Abort(regval, memaddrdesc.fault);

    cache.cpas = CPASAtPAS(memaddrdesc.paddress.paspace);
    cache.paddress = memaddrdesc.paddress;
    CACHE_OP(cache);
    return;

aarch64/instrs/system/sysops/predictionrestrict/RestrictPrediction

// RestrictPrediction()
// ====================
// Clear all predictions in the context.

AArch64.RestrictPrediction(bits(64) val, RestrictType restriction)

ExecutionCntxt c;
target_el = val<25:24>;

// If the instruction is executed at an EL lower than the specified
// level, it is treated as a NOP.
if UInt(target_el) > UInt(PSTATE.EL) then return;

bit ns = val<26>;
bit nse = val<27>;
ss = TargetSecurityState(ns, nse);
if ss == SS_Root && target_el != EL3 then return;

c.security = ss;
c.target_el = target_el;

if EL2Enabled() && !IsInHost() then
    if PSTATE.EL IN {EL0, EL1} then
        c.is_vmid_valid = TRUE;
        c.all_vmid = FALSE;
        c.vmid = VMID[];
    elsif target_el IN {EL0, EL1} then
        c.is_vmid_valid = TRUE;
        c.all_vmid = val<48> == '1';
        c.vmid = val<47:32>; // Only valid if val<48> == '0';
else
  c.is_vmid_valid = FALSE;
else
  c.is_vmid_valid = FALSE;

if PSTATE.EL == EL0 then
  c.is_asid_valid = TRUE;
  c.all_asid = FALSE;
  c.asid = ASID[];
elsif target_el == EL0 then
  c.is_asid_valid = TRUE;
  c.all_asid = val<16> == '1';
  c.asid = val<15:0>; // Only valid if val<16> == '0';
else
  c.is_asid_valid = FALSE;
  c.restriction = restriction;
  RESTRICT_PREDICTIONS(c);

aarch64/instrs/system/sysops/sysop/SysOp

// SysOp() // ========
SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
  case op1:CRn:CRm:op2 of
    when '000 0111 1000 000' return Sys_AT; // S1E1R
    when '000 0111 1000 001' return Sys_AT; // S1E1W
    when '000 0111 1000 010' return Sys_AT; // S1E0R
    when '000 0111 1000 011' return Sys_AT; // S1E0W
    when '100 0111 1000 000' return Sys_AT; // S1E2R
    when '100 0111 1000 001' return Sys_AT; // S1E2W
    when '100 0111 1000 010' return Sys_AT; // S1E1R
    when '100 0111 1000 011' return Sys_AT; // S1E1W
    when '100 0111 1000 100' return Sys_AT; // S12E1R
    when '100 0111 1000 101' return Sys_AT; // S12E1W
    when '100 0111 1000 110' return Sys_AT; // S12E0R
    when '100 0111 1000 111' return Sys_AT; // S12E0W
    when '110 0111 1000 000' return Sys_AT; // S1E0R
    when '110 0111 1000 001' return Sys_AT; // S1E0W
    when '001 0111 0010 100' return Sys_BRB; // IALL
    when '001 0111 0010 101' return Sys_BRB; // INJ
    when '000 0111 0110 001' return Sys_DC; // IVAC
    when '000 0111 0110 010' return Sys_DC; // ISW
    when '000 0111 0110 011' return Sys_DC; // IGVAC
    when '000 0111 1010 000' return Sys_DC; // IGSW
    when '110 0111 0100 001' return Sys_DC; // CVAC
    when '110 0111 0100 011' return Sys_DC; // CVAP
    when '011 0111 1010 001' return Sys_DC; // CGSW
    when '011 0111 1010 010' return Sys_DC; // CGDSW
    when '011 0111 1110 001' return Sys_DC; // CSGW
    when '011 0111 1110 010' return Sys_DC; // CISW
    when '011 0111 1110 100' return Sys_DC; // CIGSW
    when '011 0111 1110 110' return Sys_DC; // CIGDSW
    when '011 0111 0100 001' return Sys_DC; // ZVA
    when '011 0111 0100 011' return Sys_DC; // GVA
    when '011 0111 0100 100' return Sys_DC; // GZVA
    when '011 0111 1010 001' return Sys_DC; // CVAC
    when '011 0111 1010 011' return Sys_DC; // CVAP
    when '011 0111 1010 101' return Sys_DC; // CGVAP
    when '011 0111 1010 111' return Sys_DC; // CGDVAP
when '011 0111 1101 001' return Sys_DC;    // CVADP
when '011 0111 1101 011' return Sys_DC;    // CVGADP
when '011 0111 1110 001' return Sys_DC;    // CIVAC
when '011 0111 1110 011' return Sys_DC;    // CIGVAC
when '110 0111 1110 101' return Sys_DC;    // CIGDVAC
when '110 0111 1110 111' return Sys_DC;    // CIPAPA
when '000 1000 0001 000' return Sys_IC;    // IALLUIS
when '000 1000 0001 001' return Sys_IC;    // IALLU
when '011 0111 0101 001' return Sys_IC;    // IVAU
when '000 1000 0001 000' return Sys_TLBI;  // VMALLE1OS
when '000 1000 0001 001' return Sys_TLBI;  // VAE1OS
when '000 1000 0001 010' return Sys_TLBI;  // ASIDE1OS
when '000 1000 0001 101' return Sys_TLBI;  // VALE1OS
when '000 1000 0001 111' return Sys_TLBI;  // VAALE1OS
when '000 1000 0010 001' return Sys_TLBI;  // RVAE1IS
when '000 1000 0010 011' return Sys_TLBI;  // RVAAE1IS
when '000 1000 0010 101' return Sys_TLBI;  // RVALE1IS
when '000 1000 0010 111' return Sys_TLBI;  // RVAALE1IS
when '000 1000 0011 000' return Sys_TLBI;  // VMALLE1IS
when '000 1000 0011 001' return Sys_TLBI;  // VAE1IS
when '000 1000 0011 010' return Sys_TLBI;  // ASIDE1IS
when '000 1000 0011 011' return Sys_TLBI;  // VAAE1IS
when '000 1000 0011 101' return Sys_TLBI;  // VALE1IS
when '000 1000 0011 111' return Sys_TLBI;  // VAALE1IS
when '000 1001 0001 001' return Sys_TLBI;  // RVAE1OS
when '000 1001 0001 011' return Sys_TLBI;  // RVAAE1OS
when '000 1001 0001 101' return Sys_TLBI;  // RVALE1OS
when '000 1001 0001 111' return Sys_TLBI;  // RVAALE1OS
when '000 1001 0010 001' return Sys_TLBI;  // RVAE1
when '000 1001 0010 011' return Sys_TLBI;  // RVAAE1
when '000 1001 0010 101' return Sys_TLBI;  // RVALE1
when '000 1001 0010 111' return Sys_TLBI;  // RVAALE1
when '000 1001 0011 000' return Sys_TLBI;  // VMALLE1
when '000 1001 0011 001' return Sys_TLBI;  // VAE1
when '000 1001 0011 010' return Sys_TLBI;  // ASIDE1
when '000 1001 0011 011' return Sys_TLBI;  // VAAE1
when '000 1001 0011 101' return Sys_TLBI;  // VALE1
when '000 1001 0011 111' return Sys_TLBI;  // VAALE1
when '000 1001 0101 001' return Sys_TLBI;  // RVAE1OSNXS
when '000 1001 0101 011' return Sys_TLBI;  // RVAAE1OSNXS
when '000 1001 0101 101' return Sys_TLBI;  // RVALE1OSNXS
when '000 1001 0101 111' return Sys_TLBI;  // RVAALE1OSNXS
when '000 1001 0110 001' return Sys_TLBI;  // VMALLE1OSNXS
when '000 1001 0110 011' return Sys_TLBI;  // VAE1OSNXS
when '000 1001 0110 101' return Sys_TLBI;  // VALE1OSNXS
when '000 1001 0110 111' return Sys_TLBI;  // VAALE1OSNXS
when '000 1001 0111 000' return Sys_TLBI;  // VMALLE1NXS
when '000 1001 0111 001' return Sys_TLBI;  // VAE1NXS
when '000 1001 0111 010' return Sys_TLBI;  // VALE1NXS
when '000 1001 0111 011' return Sys_TLBI;  // VAALE1NXS
when '000 1001 0111 101' return Sys_TLBI;  // VALE1NXS
when '000 1001 0111 111' return Sys_TLBI;  // VAALE1NXS
when '000 1001 1001 001' return Sys_TLBI;  // RVAAE1NXS
when '000 1001 1001 011' return Sys_TLBI;  // RVAAE1ISNXS
when '000 1001 1001 101' return Sys_TLBI;  // RVAAE1OSNXS
when '000 1001 1001 111' return Sys_TLBI;  // RVAAE1OSISNXS
when '000 1001 1010 001' return Sys_TLBI;  // VMALLE1OS
when '000 1001 1010 011' return Sys_TLBI;  // VAE1OS
when '000 1001 1010 101' return Sys_TLBI;  // VALE1OS
when '000 1001 1010 111' return Sys_TLBI;  // VAALE1OS
when '000 1001 1011 000' return Sys_TLBI;  // VMALLE1IS
when '000 1001 1011 001' return Sys_TLBI;  // VALE1IS
when '000 1001 1011 010' return Sys_TLBI;  // VAALE1IS
when '000 1001 1011 011' return Sys_TLBI;  // VALE1IS
when '000 1001 1011 100' return Sys_TLBI;  // VAALE1IS
when '000 1001 1011 101' return Sys_TLBI;  // VALE1IS
when '000 1001 1011 110' return Sys_TLBI;  // VAALE1IS
when '000 1001 0111 011' return Sys_TLBI;  // VAAE1NXS
when '000 1001 0111 101' return Sys_TLBI;  // VALE1NXS
when '000 1001 0111 111' return Sys_TLBI;  // VALE1NXS
when '100 1000 0000 001' return Sys_TLBI;  // IPAS2E1IS
when '100 1000 0000 010' return Sys_TLBI;  // RIPA2E1IS
when '100 1000 0000 101' return Sys_TLBI;  // RIPA2E1IS
when '100 1000 0001 000' return Sys_TLBI;  // ALLE2OS
when '100 1000 0001 001' return Sys_TLBI;  // VAE2OS
when '100 1000 0001 100' return Sys_TLBI;  // ALLE1OS
when '100 1000 0001 101' return Sys_TLBI;  // VALE2OS
when '100 1000 0001 110' return Sys_TLBI;  // VMALLS12E1OS
when '100 1000 0010 001' return Sys_TLBI;  // RVAE2IS
when '100 1000 0010 101' return Sys_TLBI;  // RVALE2IS
when '100 1000 0011 000' return Sys_TLBI;  // ALLE2IS
when '100 1000 0011 001' return Sys_TLBI;  // VAE2IS
when '100 1000 0011 100' return Sys_TLBI;  // VALE2IS
when '100 1000 0011 101' return Sys_TLBI;  // VALE2IS
when '100 1000 0011 110' return Sys_TLBI;  // VMALLS12E1IS
when '100 1000 0100 000' return Sys_TLBI;  // IPAS2E1OS
when '100 1000 0100 001' return Sys_TLBI;  // IPAS2E1
when '100 1000 0100 010' return Sys_TLBI;  // RIPAS2E1
when '100 1000 0100 011' return Sys_TLBI;  // RIPAS2E1OS
when '100 1000 0100 100' return Sys_TLBI;  // IPAS2LE1OS
when '100 1000 0100 101' return Sys_TLBI;  // IPAS2LE1
when '100 1000 0100 110' return Sys_TLBI;  // RIPAS2LE1
when '100 1000 0100 111' return Sys_TLBI;  // RIPAS2LE1OS
when '100 1000 0101 001' return Sys_TLBI;  // RVAE2OS
when '100 1000 0101 101' return Sys_TLBI;  // RVALE2OS
when '100 1000 0110 001' return Sys_TLBI;  // RVAE2
when '100 1000 0110 101' return Sys_TLBI;  // RVALE2
when '100 1000 0111 000' return Sys_TLBI;  // ALLE2
when '100 1000 0111 001' return Sys_TLBI;  // VAE2
when '100 1000 0111 100' return Sys_TLBI;  // VALE2
when '100 1000 0111 101' return Sys_TLBI;  // VALE2
when '100 1000 0111 110' return Sys_TLBI;  // VMALLS12E1
when '100 1001 0000 001' return Sys_TLBI;  // IPAS2E1ISNXS
when '100 1001 0000 010' return Sys_TLBI;  // RIPAS2E1ISNXS
when '100 1001 0000 101' return Sys_TLBI;  // RIPAS2E1ISNXS
when '100 1001 0001 000' return Sys_TLBI;  // ALLE2OSNXS
when '100 1001 0001 001' return Sys_TLBI;  // VAE2OSNXS
when '100 1001 0001 100' return Sys_TLBI;  // ALLE1OSNXS
when '100 1001 0001 101' return Sys_TLBI;  // VALE2OSNXS
when '100 1001 0001 110' return Sys_TLBI;  // VMALLS12E1OSNXS
when '100 1001 0010 001' return Sys_TLBI;  // RVAE2ISNXS
when '100 1001 0010 101' return Sys_TLBI;  // RVALE2ISNXS
when '100 1001 0011 000' return Sys_TLBI;  // ALLE2ISNXS
when '100 1001 0011 001' return Sys_TLBI;  // VAE2ISNXS
when '100 1001 0011 100' return Sys_TLBI;  // ALLE1ISNXS
when '100 1001 0011 101' return Sys_TLBI;  // VALE2ISNXS
when '100 1001 0011 110' return Sys_TLBI;  // VMALLS12E1ISNXS
when '100 1001 0100 001' return Sys_TLBI;  // IPAS2E1OSNXS
when '100 1001 0100 101' return Sys_TLBI;  // RVAE2OSNXS
when '100 1001 0101 001' return Sys_TLBI;  // RVAE2OSNXS
when '100 1001 0101 101' return Sys_TLBI;  // RVAE2OSNXS
when '100 1001 0110 001' return Sys_TLBI;  // RVAE2
when '100 1001 0110 101' return Sys_TLBI;  // RVAE2
when '100 1001 0111 000' return Sys_TLBI;  // ALLE2
when '100 1001 0111 001' return Sys_TLBI;  // VAE2
when '100 1001 0111 100' return Sys_TLBI;  // ALLE2
when '100 1001 0111 101' return Sys_TLBI;  // VALE2
when '100 1001 0111 110' return Sys_TLBI;  // VMALLS12E1NXS
when '110 1000 0001 000' return Sys_TLBI;  // ALLE30S
when '110 1000 0001 001' return Sys_TLBI;  // VAE30S
when '110 1000 0001 100' return Sys_TLBI;  // PAALL0S
when '110 1000 0010 001' return Sys_TLBI;  // RVAE3IS
when '110 1000 0011 000' return Sys_TLBI;  // ALLE3IS
when '110 1000 0011 001' return Sys_TLBI;  // VAE3IS
when '110 1000 0011 100' return Sys_TLBI;  // PAALL
when '110 1000 0011 101' return Sys_TLBI;  // VAE3
when '110 1000 0100 011' return Sys_TLBI;  // RPAO5
when '110 1000 0100 111' return Sys_TLBI;  // RPAO5
when '110 1000 0101 001' return Sys_TLBI;  // RVAE3OS
when '110 1000 0101 101' return Sys_TLBI;  // RVALE3OS
when '110 1000 0110 001' return Sys_TLBI;  // ALLE3OSNXS
when '110 1000 0110 001' return Sys_TLBI;  // ALLE3OSNXS
when '110 1000 0110 101' return Sys_TLBI;  // RVALE3OSNXS
when '110 1000 0111 000' return Sys_TLBI;  // ALLE3NXS
when '110 1000 0111 001' return Sys_TLBI;  // VAE3NXS
when '110 1000 0111 101' return Sys_TLBI;  // VALE3NXS
when '110 1001 0001 000' return Sys_TLBI;  // ALLE3OSNXS
when '110 1001 0001 001' return Sys_TLBI;  // VAE3OSNXS
when '110 1001 0001 101' return Sys_TLBI;  // VALE3OSNXS
when '110 1001 0010 001' return Sys_TLBI;  // RVAE3ISNXS
when '110 1001 0010 101' return Sys_TLBI;  // RVALE3ISNXS
when '110 1001 0011 000' return Sys_TLBI;  // ALLE3ISNXS
when '110 1001 0011 001' return Sys_TLBI;  // VAE3ISNXS
when '110 1001 0011 101' return Sys_TLBI;  // VALE3ISNXS
when '110 1001 0101 001' return Sys_TLBI;  // RVAE3OSNXS
when '110 1001 0101 101' return Sys_TLBI;  // RVALE3OSNXS
when '110 1001 0110 001' return Sys_TLBI;  // RVAE3NXS
when '110 1001 0110 101' return Sys_TLBI;  // RVALE3NXS
when '110 1001 0111 000' return Sys_TLBI;  // ALLE3NXS
when '110 1001 0111 001' return Sys_TLBI;  // VAE3NXS
when '110 1001 0111 101' return Sys_TLBI;  // VALE3NXS
otherwise                return Sys_SYS;

aarch64/intrs/system/sysops/sysop/SystemOp

enumeration SystemOp {Sys_AT, Sys_BRB, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};

aarch64/intrs/system/sysops/tlbi/AArch32.DTLBI_ALL

// AArch32.DTLBI_ALL()
// =============
// Invalidate all data TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
AArch32.DTLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIR e;
    r.op           = TLBIOp_DALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    TLBIR(e);

if shareability != Shareability_NSH then Broadcast(shareability, r); return;

aarch64/instrs/system/sysops/tlbi/AArch32.DTLBI_ASID

// AArch32.DTLBI_ASID()
// =============
// Invalidate all data TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.
AArch32.DTLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability, TLBIMemAttr attr, bits(32) Rt)
  assert PSTATE.EL IN {EL3, EL2, EL1};
  TLBIRecord r;
  r.op = TLBIOp_DASID;
  r.from_aarch64 = FALSE;
  r.security = security;
  r.regime = regime;
  r.vmid = vmid;
  r.level = TLBILevel_Any;
  r.attr = attr;
  r.asid = Zeros(8) : Rt<7:0>;
  TLBI(r);
  if shareability != Shareability_NSH then Broadcast(shareability, r); return;

aarch64/instrs/system/sysops/tlbi/AArch32.DTLBI_VA

// AArch32.DTLBI_VA()
// ===============
// Invalidate by VA all stage 1 data TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.
AArch32.DTLBI_VA(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
  assert PSTATE.EL IN {EL3, EL2, EL1};
  TLBIRecord r;
  r.op = TLBIOp_DVA;
  r.from_aarch64 = FALSE;
  r.security = security;
  r.regime = regime;
  r.vmid = vmid;
  r.level = level;
  r.attr = attr;
  r.asid = Zeros(8) : Rt<7:0>;}
r.address = Zeros(32) : Rt<31:12> : Zeros(12);

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

aarch64/instrs/system/sysops/tlbi/AArch32.ITLBI_ALL

// AArch32.ITLBI_ALL()
//===----------------------------------------------------------------------===
// Invalidate all instruction TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.

AArch32.ITLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op = TLBIOp_IALL;
r.from_aarch64 = FALSE;
r.security = security;
r.regime = regime;
r.level = TLBILevel_Any;
attr = attr;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

aarch64/instrs/system/sysops/tlbi/AArch32.ITLBI_ASID

// AArch32.ITLBI_ASID()
//===----------------------------------------------------------------------===
// Invalidate all instruction TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.ITLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability, TLBIMemAttr attr, bits(32) Rt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op = TLBIOp_IASID;
r.from_aarch64 = FALSE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = TLBILevel_Any;
attr = attr;
r.asid = Zeros(8) : Rt<7:0>;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;
aarch64/instrs/system/sysops/tlbi/AArch32.ITLBI_VA

// AArch32.ITLBI_VA()
// ================
// Invalidate by VA all stage 1 instruction TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// when the indicated level is
//     TLBILevel_Any  : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// when attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.
AArch32.ITLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level,  TLBIMemAttr attr, bits(32) Rt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
  r.op           = TLBIOp_IVA;
  r.from_aarch64 = FALSE;
  r.security     = security;
  r.regime       = regime;
  r.vmid         = vmid;
  r.level        = level;
  r.attr         = attr;
  r.asid         = Zeros(8) : Rt<7:0>;
  r.address      = Zeros(32) : Rt<31:12> : Zeros(12);

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_ALL

// AArch32.TLBI_ALL()
// ================
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// when attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.
AArch32.TLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2};

TLBIRecord r;
  r.op           = TLBIOp_ALL;
  r.from_aarch64 = FALSE;
  r.security     = security;
  r.regime       = regime;
  r.level        = TLBILevel_Any;
  r.attr         = attr;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;
arm64/instrs/system/sysops/tlbi/AArch32.TLBI_ASID

// AArch32.TLBI_ASID()
// ===================
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
                   TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp_ASID;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = TLBILevel_Any;
    r.attr = attr;
    r.asid = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;

arm64/instrs/system/sysops/tlbi/AArch32.TLBI_IPAS2

// AArch32.TLBI_IPAS2()
// ====================
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters are derived from Rt.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2};
    assert security == SS_NonSecure;

    TLBIRecord r;
    r.op = TLBIOp_IPAS2;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.address = Zeros(24) : Rt<27:0> : Zeros(12);
    r.ipaspace = PAS_NonSecure;

    TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r); return;

aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VA

// AArch32.TLBI_VA()
// --------------------
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// when the indicated level is
// TLBILevel_Any : this applies to TLB entries at all levels
// TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLB_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.
AArch32.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
assert PSTATE.EL IN {EL3, EL2, EL1};
TLBRecord r;
  r.op = TLBIOp_VA;
  r.from_aarch64 = FALSE;
  r.security = security;
  r.regime = regime;
  r.vmid = vmid;
  r.level = level;
  r.attr = attr;
  r.asid = Zeros(8) : Rt<7:0>;
  r.address = Zeros(32) : Rt<31:12> : Zeros(12);
TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VAA

// AArch32.TLBI_VAA()
// --------------------
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// when the indicated level is
// TLBILevel_Any : this applies to TLB entries at all levels
// TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLB_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.
AArch32.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
assert PSTATE.EL IN {EL3, EL2, EL1};
TLBRecord r;
  r.op = TLBIOp_VAA;
  r.from_aarch64 = FALSE;
  r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = level;
r.attr = attr;
r.address = Zeros(32) : Rt<31:12> : Zeros(12);

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VMALL

// AArch32.TLBI_VMALL()
// ===============
// Invalidate all stage 1 entries for the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.
AArch32.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBRecord r;
  r.op = TLBIOp_VMALL;
  r.from_aarch64 = FALSE;
  r.security = security;
  r.regime = regime;
  r.level = TLBILevel_Any;
  r.vmid = vmid;
  r.attr = attr;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VMALLS12

// AArch32.TLBI_VMALLS12()
// ===============
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.
AArch32.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2};

TLBRecord r;
  r.op = TLBIOp_VMALLS12;
  r.from_aarch64 = FALSE;
  r.security = security;
  r.regime = regime;
  r.level = TLBILevel_Any;
  r.vmid = vmid;
r.attr = attr;

TLBI(r);
if shareability != Shareability_NS then Broadcast(shareability, r);
return;

aarch64/intrs/system/sysops/tlbi/AArch64.TLBI_ALL

// AArch64.TLBI_ALL()
// ================
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.
AArch64.TLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2};

TLBIRecord r;
  r.op = TLBIOp_ALL;
  r.from_aarch64 = TRUE;
  r.security = security;
  r.regime = regime;
  r.level = TLBILevel_Any;
  r.attr = attr;

TLBI(r);
if shareability != Shareability_NS then Broadcast(shareability, r);
return;

aarch64/intrs/system/sysops/tlbi/AArch64.TLBI_ASID

// AArch64.TLBI_ASID()
// ===============
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Xt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.
AArch64.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
  TLBIMemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
  r.op = TLBIOp_ASID;
  r.from_aarch64 = TRUE;
  r.security = security;
  r.regime = regime;
  r.vmid = vmid;
  r.level = TLBILevel_Any;
  r.attr = attr;
  r.asid = Xt<63:48>;

TLBI(r);
if shareability != Shareability_NS then Broadcast(shareability, r);
return;
aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_IPAS2

// AArch64.TLBI_IPAS2()
// ====================
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2};

TLBIRecord r;
  r.op = TLBIOp_IPAS2;
  r.from_aarch64 = TRUE;
  r.security = security;
  r.regime = regime;
  r.vmid = vmid;
  r.level = level;
  r.attr = attr;
  r.address = ZeroExtend(Xt<39:0> : Zeros(12), 64);

case security of
  when SS_NonSecure
    r.ipaspace = PAS_NonSecure;
  when SS_Secure
    r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;
  when SS_Realm
    r.ipaspace = PAS_Realm;
  otherwise
    // Root security state does not have stage 2 translation
    Unreachable();

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_PAALL

// AArch64.TLBI_PAALL()
// ====================
// TLB Invalidate ALL GPT Information.
// Invalidates cached copies of GPT entries from TLBs in the indicated
// Shareability domain.
// The invalidation applies to all TLB entries containing GPT information.

AArch64.TLBI_PAALL(Shareability shareability)
assert HaveRME() && PSTATE.EL == EL3;

TLBIRecord r;

// r.security and r.regime do not apply for TLBI by PA operations
  r.op = TLBIOp_PAALL;
  r.level = TLBILevel_Any;
  r.attr = TLBI_AllAttr;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

aarch64/intrs/system/sysops/tlbi/AArch64.TLBI_RIPAS2

// AArch64.TLBI_RIPAS2()
// =====================
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// shareability domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.
// When the indicated level is
// TLBILevel_Any : this applies to TLB entries at all levels
// TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
    r.op = TLBIop_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;

    bits(2) tg = Xt<47:46>;
    integer scale = UInt(Xt<45:44>);
    integer num = UInt(Xt<43:39>);
    integer baseaddr = SInt(Xt<36:0>);

boolean valid;
(valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);
if !valid then return;

case security of
    when SS_NonSecure
        r.ipaspace = PAS_NonSecure;
    when SS_Secure
        r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;
    when SS_Realm
        r.ipaspace = PAS_Realm;
    otherwise
        // Root security state does not have stage 2 translation
        Unreachable();

        TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;

aarch64/intrs/system/sysops/tlbi/AArch64.TLBI_RPA

// AArch64.TLBI_RPA()
// ==================
// TLB Range Invalidate GPT Information by PA.


// Invalidates cached copies of GPT entries from TLBs in the indicated
// Shareability domain.
// The invalidation applies to TLB entries containing GPT information relating
// to the indicated physical address range.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries containing GPT information
//                     from all levels of the GPT walk
//     TLBILevel_Last : this applies to TLB entries containing GPT information
//                     from the last level of the GPT walk

AArch64.TLBI_RPA(TLBILevel level, bits(64) Xt, Shareability shareability)
assert HaveRME() && PSTATE.EL == EL3;

TLBIRecord r;
integer range_bits;
integer p;

// r.security and r.regime do not apply for TLBI by PA operations
r.op    = TLBIOp_RPA;
r.level = level;
r.attr  = TLBI_AllAttr;

// SIZE field
case Xt<47:44> of
  when '0000' range_bits = 12; // 4KB
  when '0001' range_bits = 14; // 16KB
  when '0010' range_bits = 16; // 64KB
  when '0011' range_bits = 21; // 2MB
  when '0100' range_bits = 25; // 32MB
  when '0101' range_bits = 29; // 512MB
  when '0110' range_bits = 30; // 1GB
  when '0111' range_bits = 34; // 16GB
  when '1000' range_bits = 36; // 64GB
  when '1001' range_bits = 39; // 512GB
  otherwise range_bits = 0; // Reserved encoding

/* If SIZE selects a range smaller than PGS, then PGS is used instead */
case DecodePGS(GPCCR_EL3.PGS) of
  when PGS_4KB  p = 12;
  when PGS_16KB p = 14;
  when PGS_64KB p = 16;

  if range_bits < p then
    range_bits = p;

bits(52) BaseADDR = Zeros(52);
case GPCCR_EL3.PGS of
  when '00' BaseADDR<51:12> = Xt<39:0>; // 4KB
  when '10' BaseADDR<51:14> = Xt<39:2>; // 16KB
  when '01' BaseADDR<51:16> = Xt<39:4>; // 64KB

// The calculation here automatically aligns BaseADDR to the size of
// the region specified in SIZE. However, the architecture does not
// require this alignment and if BaseADDR is not aligned to the region
// specified by SIZE then no entries are required to be invalidated.
bits(52) start_addr = BaseADDR AND NOT ZeroExtend(Ones(range_bits), 52);
bits(52) end_addr   = start_addr + ZeroExtend(Ones(range_bits), 52);

// PASpace is not considered in TLBI by PA operations
r.address     = ZeroExtend(start_addr, 64);
r.end_address = ZeroExtend(end_addr, 64);

TLBI(r);
if shareability != Shareability_NSIX then Broadcast(shareability, r);
**AArch64/TLBI/AArch64.TLBI_RVA**

```c
// AArch64.TLBI_RVA()
// ===================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// when the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// when attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_RVA(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)

assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
    r.op           = TLBIOp_RVA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt<63:48>;

boolean valid;
(valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);
if !valid then return;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;
```

**AArch64/TLBI/AArch64.TLBI_RVAA**

```c
// AArch64.TLBI_RVAA()
// ===================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// when the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// when attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_RVAA(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)

assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
    r.op           = TLBIOp_RVAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
```
r.regime   = regime;
r.vmid     = vmid;
r.level    = level;
r.attr     = attr;

bits(2) tg  = Xt<47:46>;
integer scale = UInt(Xt<45:44>);
integer num  = UInt(Xt<43:39>);
integer baseaddr = SInt(Xt<36:0>);

boolean valid;

(valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);
if !valid then return;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VA

// AArch64.TLBI_VA()
// ================
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// when the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// when attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.
AArch64.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRrecord r;
r.op           = TLBIOp_VA;
r.from_aarch64 = TRUE;
r.security     = security;
r.regime       = regime;
r.vmid         = vmid;
r.level        = level;
r.attr         = attr;
r.asid         = Xt<63:48>;
r.address      = ZeroExtend(Xt<43:0> : Zeros(12), 64);

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VAA

// AArch64.TLBI_VAA()
// ================
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// when the indicated level is
// TLBILevel_Any : this applies to TLB entries at all levels
// TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// when attr is TLBIExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
  r.op       = TLBIOp_VAA;
  r.from_aarch64 = TRUE;
  r.security     = security;
  r.regime       = regime;
  r.vmid         = vmid;
  r.level        = level;
  r.attr         = attr;
  r.address      = ZeroExtend(Xt<43:0> : Zeros(12), 64);

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VMALL

// AArch64.TLBI_VMALL()
//================
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// when attr is TLBIExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
  r.op       = TLBIOp_VMALL;
  r.from_aarch64 = TRUE;
  r.security     = security;
  r.regime       = regime;
  r.vmid         = vmid;
  r.level        = TLBILevel_Any;
  r.attr         = attr;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VMALLS12

// AArch64.TLBI_VMALLS12()
//========================
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2};

TLBIRecord r;
	 r.op           = TLBIOp_VMALLS12;
	 r.from_aarch64 = TRUE;
	 r.security     = security;
	 r.regime       = regime;
	 r.level        = TLBILevel_Any;
	 r.vmid         = vmid;
	 r.attr         = attr;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

aarch64/instrs/system/sysops/tlbi/ASID_NONE
constant bits(16) ASID_NONE = Zeros(16);

aarch64/instrs/system/sysops/tlbi/Broadcast
// Broadcast()
// ===========
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the indicated shareability
// domain.

Broadcast(Shareability shareability, TLBIRecord r)
IMPLEMENTATION_DEFINED;

aarch64/instrs/system/sysops/tlbi/DecodeTLBITG
// DecodeTLBITG()
// ==============
// Decode translation granule size in TLBI range instructions

TGx DecodeTLBITG(bits(2) tg)
	 case tg of
		 when '01' return TGx_4KB;
		 when '10' return TGx_16KB;
		 when '11' return TGx_64KB;

aarch64/instrs/system/sysops/tlbi/GPTTLBIMatch
// GPTTLBIMatch()
// =============
// Determine whether the GPT TLB entry lies within the scope of inavlidation

boolean GPTTLBIMatch(TLBIRecord tlbi, GPTEntry entry)
assert tlbi.op IN {TLBIOp_RPA, TLBIOp_PAALL};

boolean match;
	 case tlbi.op of
		 when TLBIOp_RPA

match = (UINT(tlbi.address<51:entry.size>) <= UINT(entry.pa<51:entry.size>) &&
       UINT(tlbi.end_address<51:entry.size>) > UINT(entry.pa<51:entry.size>));
when TLBIop_PAALL
    match = TRUE;
return match;

aarch64/instrs/system/sysops/tlbi/HasLargeAddress

// HasLargeAddress()
// =============
// Returns TRUE if the regime is configured for 52 bit addresses, FALSE otherwise.

boolean HasLargeAddress(Regime regime)
if !Have52BitIPAAndPASpaceExt() then
    return FALSE;
case regime of
when Regime_EL3
    return TCR_EL3<32> == '1';
when Regime_EL2
    return TCR_EL2<32> == '1';
when Regime_EL20
    return TCR_EL2<59> == '1';
when Regime_EL10
    return TCR_EL1<59> == '1';
otherwise
    Unreachable();

aarch64/instrs/system/sysops/tlbi/TLBI

// TLBI()
// ======
// Performs TLB maintenance of operation on TLB to invalidate the matching transition table entries.

TLBI(TLBIRecord r)
IMPLEMENTATION_DEFINED;

aarch64/instrs/system/sysops/tlbi/TLBILevel

enumeration TLBILevel {
    TLBILevel_Any,
    TLBILevel_Last
};

aarch64/instrs/system/sysops/tlbi/TLBIMatch

// TLBIMatch()
// ============
// Determine whether the TLB entry lies within the scope of invalidation

boolean TLBIMatch(TLBIRecord tlbi, TLBRecord entry)
boolean match;
case tlbi.op of
when TLBIop_DALL, TLBIop_IALL
    match = (tlbi.security == entry.context.ss &&
              tlbi.regime    == entry.context.regime);  
when TLBIop_DASID, TLBIop_IASID
    match = (entry.context.includes_s1 &&
              tlbi.security == entry.context.ss &&
              tlbi.regime    == entry.context.regime &&
              (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
              ...);
(UseASID(entry.context) && entry.context.nG == '1' &&
        tlb.asid == entry.context.asid));

when TLBIOp_DVA, TLBIOp_IVA
match = (entry.context.includes_s1 &&
        tlb.security == entry.context.ss &&
        tlb.regime == entry.context.regime &&
        (UseVMID(entry.context) || tlb.vmid == entry.context.vmid) &&
        (UseASID(entry.context) || tlb.asid == entry.context.asid ||
          entry.context.nG == '0') &&
        tlb.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
        (!entry.walkstate.istable));

when TLBIOp_ALL
relax_regime = (tlb.from_aarch64 &&
               tlb.regime IN {Regime_EL20, Regime_EL2} &&
               entry.context.regime IN {Regime_EL20, Regime_EL2});
match = (tlb.security == entry.context.ss &&
          (tlb.regime == entry.context.regime || relax_regime));

when TLBIOp_ASID
match = (entry.context.includes_s1 &&
        tlb.security == entry.context.ss &&
        tlb.regime == entry.context.regime &&
        (UseVMID(entry.context) || tlb.vmid == entry.context.vmid) &&
        (UseASID(entry.context) && entry.context.nG == '1' &&
          tlb.asid == entry.context.asid));

when TLBIOp_IPAS2
match = (!entry.context.includes_s1 && entry.context.includes_s2 &&
        tlb.security == entry.context.ss &&
        tlb.regime == entry.context.regime &&
        (UseVMID(entry.context) || tlb.vmid == entry.context.vmid) &&
        tlb.ipaspace == entry.context.ipaspace &&
        tlb.address<51:entry.blocksize> == entry.context.ia<51:entry.blocksize> &&
        (!entry.walkstate.istable));

when TLBIOp_VAA
match = (entry.context.includes_s1 &&
        tlb.security == entry.context.ss &&
        tlb.regime == entry.context.regime &&
        (UseVMID(entry.context) || tlb.vmid == entry.context.vmid) &&
        tlb.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
        (!entry.walkstate.istable));

when TLBIOp_VA
match = (entry.context.includes_s1 &&
        tlb.security == entry.context.ss &&
        tlb.regime == entry.context.regime &&
        (UseVMID(entry.context) || tlb.vmid == entry.context.vmid) &&
        (UseASID(entry.context) || entry.context.nG == '0') &&
        tlb.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
        (!entry.walkstate.istable));

when TLBIOp_VMALL
match = (entry.context.includes_s1 &&
        tlb.security == entry.context.ss &&
        tlb.regime == entry.context.regime &&
        (UseVMID(entry.context) || tlb.vmid == entry.context.vmid));

when TLBIOp_VMALLS12
match = (tlb.security == entry.context.ss &&
        tlb.regime == entry.context.regime &&
        (UseVMID(entry.context) || tlb.vmid == entry.context.vmid));

when TLBIOp_RIPAS2
match = (!entry.context.includes_s1 && entry.context.includes_s2 &&
        tlb.security == entry.context.ss &&
        tlb.regime == entry.context.regime &&
        (UseVMID(entry.context) || tlb.vmid == entry.context.vmid) &&
        tlb.ipaspace == entry.context.ipaspace &&
        (tlb.tg != '00' && DecodeTLBITG(tlb.tg) == entry.context.tg) &&
        UInt(tlb.address) <= UInt(entry.context.ia) &&
        UInt(tlb.end_address) > UInt(entry.context.ia));

when TLBIOp_RVAA
match = (entry.context.includes_s1 &&
tlbi.security == entry.context.ss &&
(tlbi.regime == entry.context.regime &&
 (UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
 (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
 Uint(tlbi.address) <= Uint(entry.context.ia) &&
 Uint(tlbi.end_address) > Uint(entry.context.ia));

when TLBIop_RVA
match = (entry.context.includes_s1 &&
 tlbi.security == entry.context.ss &&
 tlbi.regime == entry.context.regime &&
 (UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
 (!UseASID(entry.context) || tlbi.asid == entry.context.asid ||
 entry.context.nG == '0') &&
 (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
 Uint(tlbi.address) <= Uint(entry.context.ia) &&
 Uint(tlbi.end_address) > Uint(entry.context.ia));

when TLBIop_RPA
match = (entry.context.includes_gpt &&
 Uint(tlbi.address) <= Uint(entry.walkstate.baseaddress.address) &&
 Uint(tlbi.end_address) > Uint(entry.walkstate.baseaddress.address));

when TLBIop_PAALL
match = entry.context.includes_gpt;

if tlbi.attr == TLBI_ExcludeXS && entry.context.xs == '1' then
 match = FALSE;

return match;

aarch64/intrs/system/sysops/tlbi/TLBIMemAttr

e enumeration TLBIMemAttr {
    TLBI_AllAttr,
    TLBI_ExcludeXS
};

aarch64/intrs/system/sysops/tlbi/TLBIOp

e enumeration TLBIOp {
    TLBIOp_DALL,          // AArch32 Data TLBI operations - deprecated
    TLBIOp_DASID,
    TLBIOp_DVA,
    TLBIOp_IALL,          // AArch32 Instruction TLBI operations - deprecated
    TLBIOp_IASID,
    TLBIOp_IVA,
    TLBIOp_ALL,
    TLBIOp_ASID,
    TLBIOp_IPAS2,
    TLBIOp_VAA,
    TLBIOp_VMALL,
    TLBIOp_VMALLS12,
    TLBIOp_RIPAS2,
    TLBIOp_RVAA,
    TLBIOp_RVA,
    TLBIOp_RPA,
    TLBIOp_PAALL,
};

aarch64/intrs/system/sysops/tlbi/TLBIRange

// TLBIRange()
// ===========
// Extract the input address range information from encoded Xt.
(boolean, bits(2), bits(64), bits(64)) TLBIRange(Regime regime, bits(64) Xt)
boolean valid = TRUE;
b_bits(64) start = Zeros(64);
b_bits(64) end = Zeros(64);
b_bits(2) tg = Xt<47:46>;
integer scale = UInt(Xt<45:44>);
integer num = UInt(Xt<43:39>);
integer tg_bits;
if tg == '00' then
  return (FALSE, tg, start, end);
end case of
  when '01' // 4KB
tg_bits = 12;
  if HasLargeAddress(regime) then
    start<52:16> = Xt<36:0>;
    start<63:53> = Replicate(Xt<36>, 11);
  else
    start<48:12> = Xt<36:0>;
    start<63:49> = Replicate(Xt<36>, 15);
  end when '10' // 16KB
tg_bits = 14;
  if HasLargeAddress(regime) then
    start<52:16> = Xt<36:0>;
    start<63:53> = Replicate(Xt<36>, 11);
  else
    start<50:14> = Xt<36:0>;
    start<63:51> = Replicate(Xt<36>, 13);
  end when '11' // 64KB
tg_bits = 16;
  start<52:16> = Xt<36:0>;
  start<63:53> = Replicate(Xt<36>, 11);
  otherwise
    Unreachable();
end
integer range = (num+1) << (5*scale + 1 + tg_bits);
end = start + range<63:0>;
if end<52> != start<52> then
  // overflow, saturate it
  end = Replicate(start<52>, 64-52) : Ones(52);
end
return (valid, tg, start, end);

aarch64/instrs/system/sysops/tlbi/TLBIRecord

type TLBIRecord is (tlbiop, from_aarch64, // originated as an AArch64 operation
  op, security, regime,
  vmid, asid, level,
  attr, ipaspace, // For operations that take IPA as input address
  address, end_address, start_address, end_address, // for range operations, end address
  tg, // for range operations, translation granule
)
aarch64/instrs/system/sysops/tlbi/VMID

// VMID[]
// ======
// Effective VMID.

bits(16) VMID[]
if EL2Enabled() then
  if !ELUsingAArch32(EL2) then
    if Have16bitVMID() && VTCR_EL2.VS == '1' then
      return VTTBR_EL2.VMID;
    else
      return ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
  else
    return ZeroExtend(VTTBR.VKID<7:0>, 16);
  endif
  if HaveEL(EL2) && HaveSecureEL2Ext() then
    return Zeros(16);
  else
    return VMID_NONE;
endif

aarch64/instrs/system/sysops/tlbi/VMID_NONE
constant bits(16) VMID_NONE = Zeros(16);

aarch64/instrs/vector/arithmetic/binary/uniform/logical/bsl-eor/vbitop/VBitOp
enumeration VBitOp {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};

aarch64/instrs/vector/arithmetic/unary/cmp/compareop/CompareOp

aarch64/instrs/vector/logical/immediateop/ImmediateOp

aarch64/instrs/vector/reduce/reduceop/Reduce

// Reduce()
// ========
bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
  integer half;
  bits(esize) hi;
  bits(esize) lo;
  bits(esize) result;
if N == esize then
    return input<esize-1:0>;

half = N DIV 2;
thi = Reduce(op, input<N-1:half>, esize, altfp);
lo = Reduce(op, input<half-1:0>, esize, altfp);

case op of
    when ReduceOp_FMINNUM
        result = FPMinNum(lo, hi, FPCR[]);
    when ReduceOp_FMAXNUM
        result = FPMaxNum(lo, hi, FPCR[]);
    when ReduceOp_FMIN
        result = FPMin(lo, hi, FPCR[], altfp);
    when ReduceOp_FMAX
        result = FPMax(lo, hi, FPCR[], altfp);
    when ReduceOp_FADD
        result = FPAdd(lo, hi, FPCR[]);
    when ReduceOp_ADD
        result = lo + hi;

return result;

aarch64/instrs/vector/reduce/reduceop/ReduceOp

enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
    ReduceOp_FMIN, ReduceOp_FMAX,
    ReduceOp_FADD, ReduceOp_ADD};

J1.1.5 aarch64/translation

This section includes the following pseudocode functions:
- aarch64/translation/debug/AArch64.CheckBreakpoint on page J1-11214.
- aarch64/translation/debug/AArch64.CheckDebug on page J1-11214.
- aarch64/translation/debug/AArch64.CheckWatchpoint on page J1-11215.
- aarch64/translation/vmsa_addrcalc/AArch64.BlockBase on page J1-11215.
- aarch64/translation/vmsa_addrcalc/AArch64.IASize on page J1-11216.
- aarch64/translation/vmsa_addrcalc/AArch64.NextTableBase on page J1-11216.
- aarch64/translation/vmsa_addrcalc/AArch64.PageBase on page J1-11216.
- aarch64/translation/vmsa_addrcalc/AArch64.PhysicalAddressSize on page J1-11217.
- aarch64/translation/vmsa_addrcalc/AArch64.S1StartLevel on page J1-11217.
- aarch64/translation/vmsa_addrcalc/AArch64.S2StartLevel on page J1-11217.
- aarch64/translation/vmsa_addrcalc/AArch64.TTEntryAddress on page J1-11218.
- aarch64/translation/vmsa_addrcalc/AArch64.TTEntryAddress on page J1-11219.
- aarch64/translation/vmsa_faults/AArch64.AddrTop on page J1-11219.
- aarch64/translation/vmsa_faults/AArch64.ContiguousBitFaults on page J1-11219.
- aarch64/translation/vmsa_faults/AArch64.DebugFault on page J1-11220.
- aarch64/translation/vmsa_faults/AArch64.ExclusiveFault on page J1-11220.
- aarch64/translation/vmsa_faults/AArch64.IPAsOutOfRange on page J1-11220.
- aarch64/translation/vmsa_faults/AArch64.OAOutOfRange on page J1-11220.
- aarch64/translation/vmsa_faults/AArch64.S1StartLevel on page J1-11221.
- aarch64/translation/vmsa_faults/AArch64.S1HasAlignmentFault on page J1-11221.
- aarch64/translation/vmsa_faults/AArch64.S1HasPermissionsFault on page J1-11221.
- aarch64/translation/vmsa_faults/AArch64.S1InvalidTxSZ on page J1-11223.
- aarch64/translation/vmsa_faults/AArch64.S2HasAlignmentFault on page J1-11223.
- aarch64/translation/vmsa_faults/AArch64.S2HasPermissionsFault on page J1-11223.
• aarch64/translation/vmsa_faults/AArch64.S2InconsistentSL on page J1-11224.
• aarch64/translation/vmsa_faults/AArch64.S2InvalidSL on page J1-11224.
• aarch64/translation/vmsa_faults/AArch64.S2InvalidTxSZ on page J1-11225.
• aarch64/translation/vmsa_faults/AArch64.VAIsOutOfRange on page J1-11225.
• aarch64/translation/vmsa_memattr/AArch64.S2ApplyFWBMemAttrs on page J1-11225.
• aarch64/translation/vmsa_tlbcontext/AArch64.GetS1TLBContext on page J1-11226.
• aarch64/translation/vmsa_tlbcontext/AArch64.GetS2TLBContext on page J1-11227.
• aarch64/translation/vmsa_tlbcontext/AArch64.TLBContextEL10 on page J1-11228.
• aarch64/translation/vmsa_tlbcontext/AArch64.TLBContextEL2 on page J1-11228.
• aarch64/translation/vmsa_translation/AArch64.AccessUsesEL on page J1-11229.
• aarch64/translation/vmsa_translation/AArch64.FullTranslate on page J1-11229.
• aarch64/translation/vmsa_translation/AArch64.MemSwapTableDesc on page J1-11229.
• aarch64/translation/vmsa_translation/AArch64.S1DisabledOutput on page J1-11230.
• aarch64/translation/vmsa_translation/AArch64.S1Translate on page J1-11234.
• aarch64/translation/vmsa_translation/AArch64.SettingAccessFlagPermitted on page J1-11235.
• aarch64/translation/vmsa_translation/AArch64.SettingDirtyStatePermitted on page J1-11236.
• aarch64/translation/vmsa_translation/AArch64.TranslateAddress on page J1-11236.
• aarch64/translation/vmsa_tentry/AArch64.BlockDesc Supported on page J1-11236.
• aarch64/translation/vmsa_tentry/AArch64.BlocknTFaults on page J1-11237.
• aarch64/translation/vmsa_tentry/AArch64.ContiguousBit on page J1-11237.
• aarch64/translation/vmsa_tentry/AArch64.DecodeDescriptorType on page J1-11237.
• aarch64/translation/vmsa_tentry/AArch64.S1ApplyOutputPerms on page J1-11238.
• aarch64/translation/vmsa_tentry/AArch64.S1ApplyTablePerms on page J1-11238.
• aarch64/translation/vmsa_tentry/AArch64.S2ApplyOutputPerms on page J1-11239.
• aarch64/translation/vmsa_walk/AArch64.S1InitialTTWState on page J1-11239.
• aarch64/translation/vmsa_walk/AArch64.S1NextWalkStateLast on page J1-11240.
• aarch64/translation/vmsa_walk/AArch64.S1NextWalkStateTable on page J1-11241.
• aarch64/translation/vmsa_walk/AArch64.S1Walk on page J1-11241.
• aarch64/translation/vmsa_walk/AArch64.S2InitialTTWState on page J1-11243.
• aarch64/translation/vmsa_walk/AArch64.S2NextWalkStateLast on page J1-11243.
• aarch64/translation/vmsa_walk/AArch64.S2NextWalkStateTable on page J1-11244.
• aarch64/translation/vmsa_walk/AArch64.S2Walk on page J1-11244.
• aarch64/translation/vmsa_walk/AArch64.SS2InitialTTWState on page J1-11246.
• aarch64/translation/vmsa_walk/AArch64.SS2OutputP4Space on page J1-11246.
• aarch64/translation/vmsa_walkparams/AArch64.BBMSupportLevel on page J1-11247.
• aarch64/translation/vmsa_walkparams/AArch64.GetS1TTWParams on page J1-11247.
• aarch64/translation/vmsa_walkparams/AArch64.GetS2TTWParams on page J1-11247.
• aarch64/translation/vmsa_walkparams/AArch64.GetVARange on page J1-11248.
• aarch64/translation/vmsa_walkparams/AArch64.HaveS1TG on page J1-11248.
• aarch64/translation/vmsa_walkparams/AArch64.HaveS2TG on page J1-11248.
• aarch64/translation/vmsa_walkparams/AArch64.MaxTxSZ on page J1-11249.
• aarch64/translation/vmsa_walkparams/AArch64.NSS2TTWParams on page J1-11249.
• aarch64/translation/vmsa_walkparams/AArch64.PAMax on page J1-11249.
• aarch64/translation/vmsa_walkparams/AArch64.S1DCacheEnabled on page J1-11250.
• aarch64/translation/vmsa_walkparams/AArch64.SIDecodeTG0 on page J1-11250.
• aarch64/translation/vmsa_walkparams/AArch64.SIDecodeTG1 on page J1-11250.
aarch64/translation/vmsa_walkparams/AArch64.S1Enabled on page J1-11251.
aarch64/translation/vmsa_walkparams/AArch64.S1ICacheEnabled on page J1-11251.
aarch64/translation/vmsa_walkparams/AArch64.S1MinTxSZ on page J1-11251.
aarch64/translation/vmsa_walkparams/AArch64.S1TTBR on page J1-11252.
aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL10 on page J1-11252.
aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL2 on page J1-11253.
aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL20 on page J1-11253.
aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL3 on page J1-11254.
aarch64/translation/vmsa_walkparams/AArch64.S2DecodeTG0 on page J1-11255.
aarch64/translation/vmsa_walkparams/AArch64.S2MinTxSZ on page J1-11255.
aarch64/translation/vmsa_walkparams/AArch64.SS2TTWParams on page J1-11255.
aarch64/translation/vmsa_walkparams/AArch64.VAMax on page J1-11256.

aarch64/translation/debug/AArch64.CheckBreakpoint

// AArch64.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.
FaultRecord AArch64.CheckBreakpoint(bits(64) vaddress, AccType acctype_in, integer size)
assert !ELUsingAArch32(S1TranslationRegime());
assert (UsingAArch32() && size IN {2,4}) || size == 4;
AccType acctype = acctype_in;
match = FALSE;
for i = 0 to NumBreakpointsImplemented() - 1
match_i = AArch64.BreakpointMatch(i, vaddress, acctype, size);
match = match || match_i;
if match && HaltOnBreakpointOrWatchpoint() then
reason = DebugHalt_Breakpoint;
Halt(reason);
elsif match then
acctype = AccType_IFETCH;
iswrite = FALSE;
return AArch64.DebugFault(acctype, iswrite);
else
return NoFault();

aarch64/translation/debug/AArch64.CheckDebug

// AArch64.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.
FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccType acctype, boolean iswrite, integer size)
FaultRecord fault = NoFault();
boolean generate_exception;

boolean d_side = (IsDataAccess(acctype) || acctype == AccType_DC);
boolean i_side = (acctype == AccType_IFETCH);
if HaveNV2Ext() && acctype == AccType_NV2REGISTER then
mask = '0';
ss = CurrentSecurityState();
generate_exception = AArch64.GenerateDebugExceptionsFrom(EL2, ss, mask) && MDSCR_EL1.MDE == '1';
else
generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
halt = HaltOnBreakpointOrWatchpoint();

if generate_exception || halt then
    if d_side then
        fault = AArch64.CheckWatchpoint(vaddress, acctype, iswrite, size);
    elsif i_side then
        fault = AArch64.CheckBreakpoint(vaddress, acctype, size);
    return fault;

aarch64/translation/debug/AArch64.CheckWatchpoint

// AArch64.CheckWatchpoint()
// ==============
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AccType acctype, 
    boolean iswrite_in, integer size)
assert !ELUsingAArch32(S1TranslationRegime());
    boolean iswrite = iswrite_in;

if acctype == AccType_DC then
    if !iswrite then
        return NoFault();
    elsif !IsDataAccess(acctype) then
        return NoFault();
    match = FALSE;
    match_on_read = FALSE;
    ispriv = AArch64.AccessUsesEL(acctype) != EL0;
    for i = 0 to NumWatchpointsImplemented() - 1
        if AArch64.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite) then
            match = TRUE;
            if DBGWCR_EL1[i].LSC<0> == '1' then
                match_on_read = TRUE;
            if match && IsAtomicRW(acctype) then
                iswrite = !match_on_read;
            if match && HaltOnBreakpointOrWatchpoint() then
                if acctype != AccType_NONFAULT && acctype != AccType_CNOTFIRST then
                    reason = DebugHalt_Watchpoint;
                    EDWAR = vaddress;
                    Halt(reason);
                else
                    // Fault will be reported and cancelled
                    return AArch64.DebugFault(acctype, iswrite);
            elsif match then
                return AArch64.DebugFault(acctype, iswrite);
            else
                return NoFault();
        aarch64/translation/vmsa_addrcalc/AArch64.BlockBase

// AArch64.BlockBase()
// ================
// Extract the address embedded in a block descriptor pointing to the base of
// a memory block

bits(52) AArch64.BlockBase(bits(64) descriptor, bit ds, TGx tgx, integer level)
bits(52) blockbase = Zeros(52);
if tgx == TGx_4KB && level == 2 then
    blockbase<47:21> = descriptor<47:21>;
elsif tgx == TGx_4KB && level == 1 then
    blockbase<47:30> = descriptor<47:30>;
elsif tgx == TGx_4KB && level == 0 then
    blockbase<47:39> = descriptor<47:39>;
elsif tgx == TGx_16KB && level == 2 then
    blockbase<47:25> = descriptor<47:25>;
elsif tgx == TGx_16KB && level == 1 then
    blockbase<47:36> = descriptor<47:36>;
elsif tgx == TGx_64KB && level == 2 then
    blockbase<47:29> = descriptor<47:29>;
elsif tgx == TGx_64KB && level == 1 then
    blockbase<47:42> = descriptor<47:42>;
else
    Unreachable();

if Have52BitPAExt() && tgx == TGx_64KB then
    blockbase<51:48> = descriptor<15:12>;
elsif ds == '1' then
    blockbase<51:48> = descriptor<9:8>:descriptor<49:48>;
return blockbase;

aarch64/translation/vmsa_addrcalc/AArch64.IASize

    // AArch64.IASize()
    // ================
    // Retrieve the number of bits containing the input address

    integer AArch64.IASize(bits(6) txsz)
        return 64 - UInt(txsz);

aarch64/translation/vmsa_addrcalc/AArch64.NextTableBase

    // AArch64.NextTableBase()
    // =======================
    // Extract the address embedded in a table descriptor pointing to the base of
    // the next level table of descriptors

    bits(52) AArch64.NextTableBase(bits(64) descriptor, bit ds, TGx tgx)
        bits(52) tablebase = Zeros(52);
        case tgx of
            when TGx_4KB  tablebase<47:12> = descriptor<47:12>;
            when TGx_16KB tablebase<47:14> = descriptor<47:14>;
            when TGx_64KB tablebase<47:16> = descriptor<47:16>;
        if Have52BitPAExt() && tgx == TGx_64KB then
            tablebase<51:48> = descriptor<15:12>;
        elsif ds == '1' then
            tablebase<51:48> = descriptor<9:8>:descriptor<49:48>;
        return tablebase;

aarch64/translation/vmsa_addrcalc/AArch64.PageBase

    // AArch64.PageBase()
    // ==================
    // Extract the address embedded in a page descriptor pointing to the base of
    // a memory page
bits(52) AArch64.PageBase(bits(64) descriptor, bit ds, TGx tgx)
bites(52) pagebase = Zeros(52);

case tgx of
  when TGx_4KB  pagebase<47:12> = descriptor<47:12>;
  when TGx_16KB pagebase<47:14> = descriptor<47:14>;
  when TGx_64KB pagebase<47:16> = descriptor<47:16>;

if Have52BitPAExt() && tgx == TGx_64KB then
  pagebase<51:48> = descriptor<15:12>;
elsif ds == '1' then
  pagebase<51:48> = descriptor<9:8>:descriptor<49:48>;
return pagebase;

aarch64/translation/vmsa_addrcalc/AArch64.PhysicalAddressSize

// AArch64.PhysicalAddressSize()
// ===================================
// Retrieve the number of bits bounding the physical address

integer AArch64.PhysicalAddressSize(bits(3) encoded_ps, TGx tgx)
  integer ps;
  integer max_ps;

case encoded_ps of
  when '000'  ps = 32;
  when '001'  ps = 36;
  when '010'  ps = 40;
  when '011'  ps = 42;
  when '100'  ps = 44;
  when '101'  ps = 48;
  when '110'  ps = 52;
  otherwise
    ps = integer IMPLEMENTATION_DEFINED "Reserved Intermediate Physical Address size value";

if tgx != TGx_64KB && !Have52BitIPAAndPASpaceExt() then
  max_ps = Min(48, AArch64.PAMax());
else
  max_ps = AArch64.PAMax();
return Min(ps, max_ps);

aarch64/translation/vmsa_addrcalc/AArch64.S1StartLevel

// AArch64.S1StartLevel()
// =======================
// Compute the initial lookup level when performing a stage 1 translation
// table walk

integer AArch64.S1StartLevel(S1TTWParams walkparams)
  // Input Address size
  isize   = AArch64.IASize(walkparams.txsz);
  granulebits = TGxGranuleBits(walkparams.tgx);
  stride   = granulebits - 3;
return FINAL_LEVEL - (((isize-1) - granulebits) DIV stride);

aarch64/translation/vmsa_addrcalc/AArch64.S2SLTTEntryAddress

// AArch64.S2SLTTEntryAddress()
// ===========================
// Compute the first stage 2 translation table descriptor address within the
// table pointed to by the base at the start level

FullAddress AArch64.S2SLTTEntryAddress(S2TTWParams walkparams, bits(52) ipa, FullAddress tablebase)
startlevel = AArch64.S2StartLevel(walkparams);
iasize = AArch64.IASize(walkparams.txsz);
granulebits = TGxGranuleBits(walkparams.tgx);
stride = granulebits - 3;
levels = FINAL_LEVEL - startlevel;

bits(52) index;
lsb = levels*stride + granulebits;
msb = iasize - 1;
index = ZeroExtend(ipa<msb:lsb>:Zeros(3), 52);

FullAddress descaddress;
descaddress.address = tablebase.address OR index;
descaddress.paspace = tablebase.paspace;
return descaddress;

aaarch64/translation/vmsa_addrcalc/AArch64.S2StartLevel

// AArch64.S2StartLevel()
// =======================
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch64.S2StartLevel(S2TTWParams walkparams)
case walkparams.tgx of
  when TGx_4KB
    case walkparams.sl2:walkparams.sl0 of
      when '000' return 2;
      when '001' return 1;
      when '010' return 0;
      when '011' return 3;
      when '100' return -1;
  when TGx_16KB
    case walkparams.sl0 of
      when '00' return 3;
      when '01' return 2;
      when '10' return 1;
      when '11' return 0;
  when TGx_64KB
    case walkparams.sl0 of
      when '00' return 3;
      when '01' return 2;
      when '10' return 1;

aaarch64/translation/vmsa_addrcalc/AArch64.TTBaseAddress

// AArch64.TTBaseAddress()
// =======================
// Retrieve the PA/IPA pointing to the base of the initial translation table

bits(52) AArch64.TTBaseAddress(bits(64) ttb, bits(6) txsz, bits(3) ps, bit ds, TGx tgx, integer startlevel)
bits(52) tablebase = Zeros(52);

// Input Address size
iasize = AArch64.IASize(txsz);
granulebits = TGxGranuleBits(tgx);
stride = granulebits - 3;
levels = FINAL_LEVEL - startlevel;
// Base address is aligned to size of the initial translation table in bytes
tsize = (iasize - (levels*stride + granulebits)) + 3;

if (Have52BitPAExt() && tgx == TGx_64KB && ps == '110') || (ds == '1') then
tsize = Max(tsize, 6);
ttablebase<51:6> = ttb<5:2>:ttb<47:6>;
else
  ttablebase<47:1> = ttb<47:1>;

  ttablebase = Align(ttablebase, 1 << tsize);

return ttablebase;

aarch64/translation/vmsa_addrcalc/AArch64.TTEntryAddress

// AArch64.TTEntryAddress()
// = ========================
// Compute translation table descriptor address within the table pointed to by
// the table base

FullAddress AArch64.TTEntryAddress(integer level, TGx tgx, bits(6) txsz,
  bits(64) ia, FullAddress tablebase)

// Input Address size
iasize = AArch64.IASize(txsz);
granulebits = TGxGranuleBits(tgx);
stride = granulebits - 3;
levels = FINAL_LEVEL - level;

bits(52) index;
lsb = levels*stride + granulebits;
msb = Min(iasize - 1, (lsb + stride) - 1);
index = ZeroExtend(ia<msb:lsb>:Zeros(3), 52);

FullAddress descaddress;
descaddress.address = tablebase.address OR index;
descaddress.paspace = tablebase.paspace;

return descaddress;

aarch64/translation/vmsa_faults/AArch64.AddrTop

// AArch64.AddrTop()
// = = = = = = = = = = = =
// Get the top bit position of the virtual address.
// Bits above are not accounted as part of the translation process.

integer AArch64.AddrTop(bit tbid, AccType acctype, bit tbi)
  if tbid == '1' && acctype == AccType_IFETCH then
    return 63;
  if tbi == '1' then
    return 55;
  else
    return 63;

aarch64/translation/vmsa_faults/AArch64.ContiguousBitFaults

// AArch64.ContiguousBitFaults()
// = = = = = = = = = = = = = =
// If contiguous bit is set, returns whether the translation size exceeds the
// input address size and if the implementation generates a fault

boolean AArch64.ContiguousBitFaults(bits(6) txsz, TGx tgx, integer level)
// Input Address size
iasize = AArch64.IASize(txsz);
// Translation size
tsize  = TranslationSize(tgx, level) + ContiguousSize(tgx, level);

fault = boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit"
return tsize > iasize && fault;

aarch64/translation/vmsa_faults/AArch64.DebugFault

// AArch64.DebugFault()
// ====================
// Return a fault record indicating a hardware watchpoint/breakpoint

FaultRecord AArch64.DebugFault(AccType acctype, boolean iswrite)
FaultRecord fault;
fault.statuscode  = Fault_Debug;
fault.acctype     = acctype;
fault.write       = iswrite;
fault.secondstage = FALSE;
fault.s2fs1walk   = FALSE;

return fault;

aarch64/translation/vmsa_faults/AArch64.ExclusiveFault

// AArch64.ExclusiveFault()
// ========================

FaultRecord AArch64.ExclusiveFault(AccType acctype, boolean iswrite,
boolean secondstage, boolean s2fs1walk)
FaultRecord fault;
fault.statuscode  = Fault_Exclusive;
fault.acctype     = acctype;
fault.write       = iswrite;
fault.secondstage = secondstage;
fault.s2fs1walk   = s2fs1walk;

return fault;

aarch64/translation/vmsa_faults/AArch64.IPAIsOutOfRange

// AArch64.IPAIsOutOfRange()
// =========================
// Check bits not resolved by translation are ZERO

boolean AArch64.IPAIsOutOfRange(bits(52) ipa, S2TTWParams walkparams)
//Input Address size
iasize = AArch64.IASize(walkparams.txsz);
if iasize < 52 then
    return !IsZero(ipa<51:iasize>);
else
    return FALSE;
aarch64/translation/vmsa_faults/AArch64.OAOutOfRange

// AArch64.OAOutOfRange()
// ======================
// Returns whether output address is expressed in the configured size number of bits

boolean AArch64.OAOutOfRange(TTWState walkstate, bits(3) ps, TGx tgx, bits(64) ia)
// Output Address size
oasize = AArch64.PhysicalAddressSize(ps, tgx);
if oasize < 52 then
  if walkstate.istable then
    baseaddress = walkstate.baseaddress.address;
    return !IsZero(baseaddress<51:oasize>);
  else
    // Output address
    oa = StageOA(ia, tgx, walkstate);
    return !IsZero(oa.address<51:oasize>);
else
  return FALSE;

aarch64/translation/vmsa_faults/AArch64.S1HasAlignmentFault

// AArch64.S1HasAlignmentFault()
// =============================
// Returns whether stage 1 output fails alignment requirement on data accesses
// to Device memory

boolean AArch64.S1HasAlignmentFault(AccType acctype, boolean aligned,
  bit ntlsmd, MemoryAttributes memattrs)
if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
  return FALSE;
if acctype == AccType_A32LSMD && ntlsmd == '0' && memattrs.device != DeviceType_GRE then
  return TRUE;
return !aligned || acctype == AccType_DCZVA;

aarch64/translation/vmsa_faults/AArch64.S1HasPermissionsFault

// AArch64.S1HasPermissionsFault()
// ===============================
// Returns whether stage 1 access violates permissions of target memory

boolean AArch64.S1HasPermissionsFault(Regime regime, SecurityState ss, TTWState walkstate,
  S1TTWParams walkparams, boolean ispriv, AccType acctype,
  boolean iswrite)
bit r;
bit w;
bit x;
permissions = walkstate.permissions;
if HasUnprivileged(regime) then
  bit pr;
  bit pw;
  bit ur;
  bit uw;
  // Apply leaf permissions
  case permissions.ap<2:1> of
    when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // Privileged access
    when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // No effect
    when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // Read-only, privileged access
    when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // Read-only
    otherwise false end;
  walkstate.permissions.ap<2:1> = pr;
  if iswrite then
    walkstate.permissions.ap<0:1> = pw;
  else
    walkstate.permissions.ap<0:1> = ur;
  end;
  permissions.ap<2:1> = pr;
end;
// Apply hierarchical permissions
case permissions.ap_table of
  when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
  when '01' (pr,pw,ur,uw) = ( pr, pw,'0','0'); // Privileged access
  when '10' (pr,pw,ur,uw) = ( pr,'0', ur,'0'); // Read-only
  when '11' (pr,pw,ur,uw) = ( pr,'0','0','0'); // Read-only, privileged access

// Locations writable by unprivileged cannot be executed by privileged
px = NOT(permissions.pxn OR permissions.pxn_table OR uw);
ux = NOT(permissions.uxn OR permissions.uxn_table);

pan_access = !(acctype IN {AccType_DC, AccType_IFETCH, AccType_AT, AccType_NV2REGISTER});
if HavePANExt() && pan_access && !((regime == Regime_EL10 && walkparams.nv1 == '1')) then
  bit pan;
  if (boolean IMPLEMENTATION_DEFINED "SCR_EL3.SIF affects EPAN" &&
      CurrentSecurityState() == SS_Secure &&
      walkstate.baseaddress.paspace == PAS_NonSecure &&
      walkparams.sif == '1') then
    ux = '0';
  if (boolean IMPLEMENTATION_DEFINED "Realm EL2&0 regime affects EPAN" &&
      CurrentSecurityState() == SS_Realm && regime == Regime_EL20 &&
      walkstate.baseaddress.paspace != PAS_Realm) then
    ux = '0';
    pan = PSTATE.PAN AND (ur OR uw OR (walkparams.epan AND ux));
  pr = pr AND NOT(pan);
  pw = pw AND NOT(pan);
  (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);
else
  // Apply leaf permissions
  case permissions.ap<2> of
    when '0' (r,w) = ('1','1'); // No effect
    when '1' (r,w) = ('1','0'); // Read-only

  // Apply hierarchical permissions
  case permissions.ap_table<1> of
    when '0' (r,w) = ( r , w ); // No effect
    when '1' (r,w) = ( r ,'0'); // Read-only

  x = NOT(permissions.xn OR permissions.xn_table);
// Prevent execution from writable locations if WXN is set
  x = x AND NOT(walkparams.wxn AND w);
// Prevent execution from Non-secure space by PE in secure state if SIF is set
  if ss == SS_Secure && walkstate.baseaddress.paspace == PAS_NonSecure then
    x = x AND NOT(walkparams.sif);
// Prevent execution from non-Root space by Root
  if ss == SS_Root && walkstate.baseaddress.paspace != PAS_Root then
    x = '0';
// Prevent execution from non-Realm space by Realm EL2 and Realm EL2&0
  if (ss == SS_Realm && regime IN {Regime_EL2, Regime_EL20} &&
      walkstate.baseaddress.paspace != PAS_Realm) then
    x = '0';

  if acctype == AccType_IFETCH then
    if (ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT &&
        walkstate.memattrs.memtype == MemType_Device) then
      return TRUE;
    return x == '0';
  elsif acctype == AccType_DC then
    if iswrite then
      return w == '0';
  else
    // DC from privileged context which do no write cannot Permission fault
    return !ispriv && (r == '0' ||
      (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
elsif acctype == AccType_IC then
    // IC instructions do not write
    assert !iswrite;
    impdef_ic_fault = boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution";

    // IC from privileged context cannot Permission fault
    return !ispriv && ((r == '0' && impdef_ic_fault) || (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
else if iswrite then
    return w == '0';
else
    return r == '0';

aarch64/translation/vmsa_faults/AArch64.S1InvalidTxSZ

// AArch64.S1InvalidTxSZ()
// =======================
// Detect erroneous configuration of stage 1 TxSZ field if the implementation
// does not constrain the value of TxSZ

boolean AArch64.S1InvalidTxSZ(S1TTWParams walkparams)
    mintxsz = AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    return UInt(walkparams.txsz) < mintxsz || UInt(walkparams.txsz) > maxtxsz;

aarch64/translation/vmsa_faults/AArch64.S2HasAlignmentFault

// AArch64.S2HasAlignmentFault()
// =============================
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory

boolean AArch64.S2HasAlignmentFault(AccType acctype, boolean aligned, MemoryAttributes memattrs)
    if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
        return FALSE;
    return !aligned || acctype == AccType_DCZVA;

aarch64/translation/vmsa_faults/AArch64.S2HasPermissionsFault

// AArch64.S2HasPermissionsFault()
// ===============================
// Returns whether stage 2 access violates permissions of target memory

boolean AArch64.S2HasPermissionsFault(boolean s2fs1walk, TTWState walkstate, SecurityState ss, S2TTWParams walkparams, boolean ispriv, AccType acctype, boolean iswrite)
    permissions = walkstate.permissions;
    memtype = walkstate.memattrs.memtype;
    r = permissions.s2ap<0>;
    w = permissions.s2ap<1>;
    bit px; 
    bit ux;
    case (permissions.s2xn:permissions.s2xnx) of
        when '00' (px,ux) = ('1','1');
        when '01' (px,ux) = ('0','1');
        when '10' (px,ux) = ('0','0');
        when '11' (px,ux) = ('1','0');
        x = if ispriv then px else ux;
if s2fs1walk && walkparams.ptw == '1' && memtype == MemType_Device then
  return TRUE;
// Prevent translation table walks in Non-secure space by Realm state
elsif s2fs1walk && ss == SS_Realm && walkstate.baseaddress.paspace != PAS_Realm then
  return TRUE;
elsif acctype == AccType_IFETCH then
  constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
  if constraint == Constraint_FAULT && memtype == MemType_Device then
    return TRUE;
  // Prevent execution from Non-secure space by Realm state
  if ss == SS_Realm && walkstate.baseaddress.paspace != PAS_Realm then
    return TRUE;
  return x == '0';
elsif acctype == AccType_DC then
  // AArch32 DC maintenance instructions operating by VA cannot fault.
  if iswrite then
    return !ELUsingAArch32(EL1) && w == '0';
  else
    return (!ispriv && !ELUsingAArch32(EL1) && r == '0') ||
      (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
elsif acctype == AccType_IC then
  // IC instructions do not write
  assert !iswrite;
  impdef_ic_fault = boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution";
  return (!ispriv && !ELUsingAArch32(EL1) && r == '0' && impdef_ic_fault) ||
    (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
elsif iswrite then
  return w == '0';
else
  return r == '0';

aarch64/translation/vmsa_faults/AArch64.S2InconsistentSL

// AArch64.S2InconsistentSL()
// =========================
// Detect inconsistent configuration of stage 2 TxSZ and SL fields

boolean AArch64.S2InconsistentSL(S2TTWParams walkparams)

startlevel = AArch64.S2StartLevel(walkparams);
levels = FINAL_LEVEL - startlevel;
granulebits = TGxGranuleBits(walkparams.tgx);
stride = granulebits - 3;

// Input address size must at least be large enough to be resolved from the start level
s1_min_iasize = (levels * stride + granulebits + 1); // Bits resolved by table walk, except initial level
+ 1); // At least 1 more bit to be decoded by initial level

// Can accomodate 1 more stride in the level + concatenation of up to 2\x4 tables
s1_max_iasize = s1_min_iasize + (stride-1) + 4;
// Configured Input Address size
iasize = AArch64.IASize(walkparams.txsz);

return iasize < s1_min_iasize || iasize > s1_max_iasize;

aarch64/translation/vmsa_faults/AArch64.S2InvalidSL

// AArch64.S2InvalidSL()
// =====================
// Detect invalid configuration of SL field
boolean AArch64.S2InvalidSL(S2TTWParams walkparams)
case walkparams.tgx of
when TGx_4KB
  case walkparams.sl2:walkparams.sl0 of
    when '1x1' return TRUE;
    when '1lx' return TRUE;
    when '010' return AArch64.PAMax() < 44;
    when '011' return !HaveSmallTranslationTableExt();
    otherwise return FALSE;
when TGx_16KB
  case walkparams.sl0 of
    when '11' return walkparams.ds == '0';
    when '10' return AArch64.PAMax() < 42;
    otherwise return FALSE;
when TGx_64KB
  case walkparams.sl0 of
    when '11' return TRUE;
    when '10' return AArch64.PAMax() < 44;
    otherwise return FALSE;

aarch64/translation/vmsa_faults/AArch64.S2InvalidTxSZ

// AArch64.S2InvalidTxSZ()
// =======================
// Detect erroneous configuration of stage 2 TxSZ field if the implementation
// does not constrain the value of TxSZ
boolean AArch64.S2InvalidTxSZ(S2TTWParams walkparams, boolean s1aarch64)
mintxsz = AArch64.S2MinTxSZ(walkparams.ds, walkparams.tgx, s1aarch64);
maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
return UInt(walkparams.txsz) < mintxsz || UInt(walkparams.txsz) > maxtxsz;

aarch64/translation/vmsa_faults/AArch64.VAIsOutOfRange

// AArch64.VAIsOutOfRange()
// ========================
// Check bits not resolved by translation are identical and of accepted value
boolean AArch64.VAIsOutOfRange(bits(64) va, AccType acctype, Regime regime, S1TTWParams walkparams)
addrtop = AArch64.AddrTop(walkparams.tbid, acctype, walkparams.tbi);
// Input Address size
iasize  = AArch64.IASize(walkparams.txsz);
if HasUnprivileged(regime) then
  if AArch64.GetVARange(va) == VARange_LOWER then
    return !IsZero(va<addrtop:iasize>);
  else
    return !IsOnes(va<addrtop:iasize>);
else
  return !IsZero(va<addrtop:iasize>);

aarch64/translation/vmsa_memattr/AArch64.S2ApplyFWBMemAttrs

// AArch64.S2ApplyFWBMemAttrs()
// ============================
// Apply stage 2 forced Write-Back on stage 1 memory attributes.
MemoryAttributes AArch64.S2ApplyFWBMemAttrs(MemoryAttributes s1_memattrs,
  bits(4) s2_attr, bits(2) s2_sh)
MemoryAttributes memattrs;
if s2_attr<2> == '0' then // S2 Device, S1 any
s2_device = DecodeDevice(s2_attr<1:0>);
memattrs.memtype = MemType_Device;
if s1_memattrs.memtype == MemType_Device then
    memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_device);
else
    memattrs.device = s2_device;
elsif s2_attr<1:0> == '11' then    // S2 attr = S1 attr
    memattrs = s1_memattrs;
elsif s2_attr<1:0> == '10' then    // Force writeback
    memattrs.memtype = MemType_Normal;
    memattrs.inner.attrs = MemAttr_WB;
    memattrs.outer.attrs = MemAttr_WB;
if (s1_memattrs.memtype == MemType_Normal &&
    s1_memattrs.inner.attrs != MemAttr_NC) then
    memattrs.inner.hints = s1_memattrs.inner.hints;
    memattrs.inner.transient = s1_memattrs.inner.transient;
else
    memattrs.inner.hints = MemHint_RWA;
    memattrs.inner.transient = FALSE;
if (s1_memattrs.memtype == MemType_Normal &&
    s1_memattrs.outer.attrs != MemAttr_NC) then
    memattrs.outer.hints = s1_memattrs.outer.hints;
    memattrs.outer.transient = s1_memattrs.outer.transient;
else
    memattrs.outer.hints = MemHint_RWA;
    memattrs.outer.transient = FALSE;
else                               // Non-cacheable unless S1 is device
    if s1_memattrs.memtype == MemType_Device then
        memattrs = s1_memattrs;
    else
        MemAttrHints cacheability_attr;
        cacheability_attr.attrs = MemAttr_NC;
        memattrs.memtype = MemType_Normal;
        memattrs.inner = cacheability_attr;
        memattrs.outer = cacheability_attr;
    endif
s2_shareability = DecodeShareability(s2_sh);
memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability, s2_shareability);
memattrs.tagged = IsS2ResultTagged(memattrs, s1_memattrs.tagged);
memattrs.shareability = EffectiveShareability(memattrs);
return memattrs;

aarch64/translation/vmsa_tlbcontext/AArch64.GetS1TLBContext

// AArch64.GetS1TLBContext()
// =========================
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch64.GetS1TLBContext(Regime regime, SecurityState ss, bits(64) va, TGx tg)
TLBContext tlbcontext;
case regime of
    when Regime_EL3  tlbcontext = AArch64.TLBContextEL3(ss, va, tg);
    when Regime_EL2  tlbcontext = AArch64.TLBContextEL2(ss, va, tg);
    when Regime_EL20 tlbcontext = AArch64.TLBContextEL20(ss, va, tg);
    when Regime_EL10 tlbcontext = AArch64.TLBContextEL10(ss, va, tg);
endcase
if tlbcontext.includes_s1  = TRUE;
// The following may be amended for EL1&0 Regime if caching of stage 2 is successful
tlbcontext.includes_s2 = FALSE;
// The following may be amended if Granule Protection Check passes
return tlbcontext;

aarch64/translation/vmsa_tlbcontext/AArch64.GetS2TLBContext

// AArch64.GetS2TLBContext()
// =========================
// Gather translation context for accesses with IPA to match against TLB entries

TLBContext AArch64.GetS2TLBContext(SecurityState ss, FullAddress ipa, TGx tg)
assert El2Enabled();

TLBContext tlbcontext;

    tlbcontext.ss       = ss;
    tlbcontext.regime   = Regime_EL10;
    tlbcontext.ipaspace = ipa.paspace;
    tlbcontext.vmid     = VMID[];
    tlbcontext.tg       = tg;
    tlbcontext.ia       = ZeroExtend(ipa.address, 64);
    if HaveCommonNotPrivateTransExt() then
        tlbcontext.cnp = if ipa.paspace == PAS_Secure then VSTTBR_EL2.CnP else VTTBR_EL2.CnP;
    else
        tlbcontext.cnp = '0';
    tlbcontext.includes_s1  = FALSE;
    tlbcontext.includes_s2  = TRUE;
// This amy be amended if Granule Protection Check passes
return tlbcontext;

aarch64/translation/vmsa_tlbcontext/AArch64.TLBContextEL10

// AArch64.TLBContextEL10()
// ========================
// Gather translation context for accesses under EL10 regime to match against TLB entries

TLBContext AArch64.TLBContextEL10(SecurityState ss, bits(64) va, TGx tg)

TLBContext tlbcontext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL10;
    tlbcontext.vmid   = VMID[];
    tlbcontext.asid   = if TCR_EL1.A1 == '0' then TTBR0_EL1.ASID else TTBR1_EL1.ASID;
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;
    if HaveCommonNotPrivateTransExt() then
        if AArch64.GetVARange(va) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_EL1.CnP;
        else
            tlbcontext.cnp = TTBR1_EL1.CnP;
        else
            tlbcontext.cnp = '0';
    return tlbcontext;
AArch64/translation/vmsa_tlbcontext/AArch64.TLBContextEL2

// AArch64.TLBContextEL2()
// =======================
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch64.TLBContextEL2(SecurityState ss, bits(64) va, TGx tg)
TLBContext tlbcontext;

  tlbcontext.ss = ss;
  tlbcontext.regime = Regime_EL2;
  tlbcontext.tg = tg;
  tlbcontext.ia = va;
  tlbcontext.cnp = if HaveCommonNotPrivateTransExt() then TTBR0_EL2.CnP else '0';

return tlbcontext;

AArch64/translation/vmsa_tlbcontext/AArch64.TLBContextEL20

// AArch64.TLBContextEL20()
// ========================
// Gather translation context for accesses under EL20 regime to match against TLB entries

TLBContext AArch64.TLBContextEL20(SecurityState ss, bits(64) va, TGx tg)
TLBContext tlbcontext;

  tlbcontext.ss = ss;
  tlbcontext.regime = Regime_EL20;
  tlbcontext.asid = if TCR_EL2.A1 == '0' then TTBR0_EL2.ASID else TTBR1_EL2.ASID;
  tlbcontext.tg = tg;
  tlbcontext.ia = va;

if HaveCommonNotPrivateTransExt() then
  if AArch64.GetVARange(va) == VARange_LOWER then
    tlbcontext.cnp = TTBR0_EL2.CnP;
  else
    tlbcontext.cnp = TTBR1_EL2.CnP;
else
  tlbcontext.cnp = '0';

return tlbcontext;

AArch64/translation/vmsa_tlbcontext/AArch64.TLBContextEL3

// AArch64.TLBContextEL3()
// =======================
// Gather translation context for accesses under EL3 regime to match against TLB entries

TLBContext AArch64.TLBContextEL3(SecurityState ss, bits(64) va, TGx tg)
TLBContext tlbcontext;

  tlbcontext.ss = ss;
  tlbcontext.regime = Regime_EL3;
  tlbcontext.tg = tg;
  tlbcontext.ia = va;
  tlbcontext.cnp = if HaveCommonNotPrivateTransExt() then TTBR0_EL3.CnP else '0';

return tlbcontext;
**AArch64/translation/vmsa_translation/AArch64.AccessUsesEL**

```plaintext
// AArch64.AccessUsesEL()
// ======================
// Returns the Exception Level of the regime that will manage the translation for a given access type.

bits(2) AArch64.AccessUsesEL(AccType acctype)
if acctype IN {AccType_UNPRIV, AccType_UNPRIVSTREAM} then
    return EL0;
elsif acctype == AccType_NV2REGISTER then
    return EL2;
else
    return PSTATE.EL;
```

**AArch64/translation/vmsa_translation/AArch64.FullTranslate**

```plaintext
// AArch64.FullTranslate()
// =======================
// Address translation as specified by VMSA
// Alignment check NOT due to memory type is expected to be done before translation

AddressDescriptor AArch64.FullTranslate(bits(64) va, AccType acctype,
    boolean iswrite, boolean aligned)

fault = NoFault();
fault.acctype = acctype;
fault.write = iswrite;

ispri = PSTATE.EL != EL0 && !(acctype IN {AccType_UNPRIV, AccType_UNPRIVSTREAM});
regime = TranslationRegime(PSTATE.EL, acctype);

ss = SecurityStateAtEL(PSTATE.EL);

AddressDescriptor ipa;
(fault, ipa) = AArch64.S1Translate(fault, regime, ss, va, acctype, aligned, iswrite, ispri);

if fault.statuscode != Fault_None then
    return CreateFaultyAddressDescriptor(va, fault);
assert (ss == SS_Realm) IMPLIES EL2Enabled();
if regime == Regime_EL10 && EL2Enabled() then
    s1aarch64 = TRUE;
s2fs1walk = FALSE;
AddressDescriptor pa;
(fault, pa) = AArch64.S2Translate(fault, ipa, s1aarch64, ss, s2fs1walk, acctype, aligned, iswrite, ispri);

if fault.statuscode != Fault_None then
    return CreateFaultyAddressDescriptor(va, fault);
else
    return pa;
else
    return ipa;
```

**AArch64/translation/vmsa_translation/AArch64.MemSwapTableDesc**

```plaintext
// AArch64.MemSwapTableDesc()
// ==========================
// Perform HW update of table descriptor as an atomic operation

(FaultRecord, bits(64)) AArch64.MemSwapTableDesc(FaultRecord fault_in, bits(64) prev_desc, bits(64) new_desc, bit ee, AddressDescriptor descupdateaddress)
```
descupdateaccess = CreateAccessDescriptor(AccType_ATOMICRW);
FaultRecord fault = fault_in;
boolean iswrite;

if HaveRME() then
    fault.gpcf = GranuleProtectionCheck(descupdateaddress, descupdateaccess);
    if fault.gpcf.gpf != GPCF_None then
        fault.statuscode = Fault_GPCFOnWalk;
        fault.paddress = descupdateaddress.paddress;
        fault.gpcfs2walk = fault.secondstage;
        return (fault, bits(64) UNKNOWN);

    // All observers in the shareability domain observe the
    // following memory read and write accesses atomically.
    (memstatus, mem_desc) = PhysMemRead(descupdateaddress, 8, descupdateaccess);
    if ee == '1' then
        mem_desc = BigEndianReverse(mem_desc);
    if IsFault(memstatus) then
        iswrite = FALSE;
        fault = HandleExternalTTWAbort(memstatus, iswrite, descupdateaddress, descupdateaccess, 8, fault);
        if IsFault(fault.statuscode) then
            fault.acctype = AccType_ATOMICRW;
            return (fault, bits(64) UNKNOWN);

    if mem_desc == prev_desc then
        ordered_new_desc = if ee == '1' then BigEndianReverse(new_desc) else new_desc;
        memstatus = PhysMemWrite(descupdateaddress, 8, descupdateaccess, ordered_new_desc);
        if IsFault(memstatus) then
            iswrite = TRUE;
            fault = HandleExternalTTWAbort(memstatus, iswrite, descupdateaddress, descupdateaccess, 8, fault);
            if IsFault(fault.statuscode) then
                fault.acctype = AccType_ATOMICRW;
                return (fault, bits(64) UNKNOWN);

        // Reflect what is now in memory (in little endian format)
        mem_desc = new_desc;
    return (fault, mem_desc);

aarch64/translation/vmsa_translation/AArch64.S1DisabledOutput

// AArch64.S1DisabledOutput()
// =========================
// Map the VA to IPA/PA and assign default memory attributes

(FaultRecord, AddressDescriptor) AArch64.S1DisabledOutput(FaultRecord fault_in, Regime regime,
SecurityState ss, bits(64) va_in,
AccType acctype, boolean aligned)

bits(64) va = va_in;
walkparams = AArch64.GetS1TTWParams(regime, va);
FaultRecord fault = fault_in;

// No memory page is guarded when stage 1 address translation is disabled
SetInGuardedPage(FALSE);

// Output Address
FullAddress oa;
oa.address = va<51:0>;
case ss of
    when SS_Secure
        oa.paspace = PAS_Secure;

when SS_NonSecure oa.paspace = PAS_NonSecure;
when SS_ROOT oa.paspace = PAS_ROOT;
when SS_REALM oa.paspace = PAS_REALM;

MemoryAttributes memattrs;
if regime == Regime_EL10 && EL2Enabled() && walkparams.dc == '1' then
    MemAttrHints default_cacheability;
    default_cacheability.attrs = MemAttr_WB;
    default_cacheability.hints = MemHint_RWA;
    default_cacheability.transient = FALSE;

    memattrs.memtype = MemType_Normal;
    memattrs.outer = default_cacheability;
    memattrs.inner = default_cacheability;
    memattrs.shareability = Shareability_NSH;
    memattrs.tagged = walkparams.dct == '1';
    memattrs.xs = '0';
elsif acctype == AccType_IFETCH then
    MemAttrHints i_cache_attr;
    if AArch64.S1ICacheEnabled(regime) then
        i_cache_attr.attrs = MemAttr_WT;
        i_cache_attr.hints = MemHint_RA;
        i_cache_attr.transient = FALSE;
    else
        i_cache_attr.attrs = MemAttr_NC;
    end

    memattrs.memtype = MemType_Normal;
    memattrs.outer = i_cache_attr;
    memattrs.inner = i_cache_attr;
    memattrs.shareability = Shareability_OSH;
    memattrs.tagged = FALSE;
    memattrs.xs = '1';
else
    memattrs.memtype = MemType_Device;
    memattrs.device = DeviceType_nGnRnE;
    memattrs.shareability = Shareability_OSH;
    memattrs.xs = '1';

fault.level = 0;
addrtop = AArch64.AddrTop(walkparams.tbid, acctype, walkparams.tbi);
if !IsZero(va<addrtop:AArch64.PAMax>()) then
    fault.statuscode = Fault_AddressSize;
elsif AArch64.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmd, memattrs) then
    fault.statuscode = Fault_Alignment;

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor)
else
    ipa = CreateAddressDescriptor(va_in, oa, memattrs);
    return (fault, ipa);

aarch64/translation/vmsa_translation/AArch64.S1Translate

// AArch64.S1Translate()
// =====================
// Translate VA to IPA/PA depending on the regime

(FaultRecord, AddressDescriptor) AArch64.S1Translate(FaultRecord fault_in, Regime regime,
    SecurityState ss, bits(64) va,
    AccType acctype, boolean aligned_in,
    boolean iswrite_in,
    boolean ispriv)

FaultRecord fault = fault_in;
boolean aligned = aligned_in;
boolean iswrite = iswrite_in;
// Prepare fault fields in case a fault is detected
fault.secondstage = FALSE;
fault.s2fs1walk = FALSE;

if !AArch64.S1Enabled(regime, acctype) then
    return AArch64.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);

walkparams = AArch64.GetS1TTWParams(regime, va);

if (AArch64.S1InvalidTxSZ(walkparams) ||
    (!ispriv && walkparams.e0pd == '1') ||
    (!ispriv && walkparams.nfd == '1' && IsDataAccess(acctype) && TSTATE.depth > 0) ||
    (!ispriv && walkparams.nfd == '1' && acctype == AccType_NONFAULT) ||
    AArch64.VAIsOutOfRange(va, acctype, regime, walkparams)) then
    fault.statuscode = Fault_Translation;
    fault.level = 0;
    return (fault, AddressDescriptor UNKNOWN);

AddressDescriptor descaddress;
TTWState walkstate;
bits(64) descriptor;
bits(64) new_desc;
bits(64) mem_desc;
repeat
    (fault, descaddress, walkstate, descriptor) = AArch64.S1Walk(fault, walkparams, va, regime,
        ss, acctype, iswrite, ispriv);
    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    if acctype == AccType_IFETCH then
        // Flag the fetched instruction is from a guarded page
        SetInGuardedPage(walkstate.guardedpage == '1');

    if AArch64.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmd,
        walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elsif IsAtomicRW(acctype) then
        if AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams,
            ispriv, acctype, FALSE) then
            // The Permission fault was not caused by lack of write permissions
            fault.statuscode = Fault_Permission;
        elsif AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams,
            ispriv, acctype, TRUE) then
            // The Permission fault was caused by lack of write permissions
            fault.statuscode = Fault_Permission;
        elsif AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams,
            ispriv, acctype, iswrite) then
            fault.statuscode = Fault_Permission;

        new_desc = descriptor;
        if walkparams.ha == '1' && AArch64.SettingAccessFlagPermitted(fault) then
            // Set descriptor AF bit
            new_desc<10> = '1';

        // If HW update of dirty bit is enabled, the walk state permissions
        // will already reflect a configuration permitting writes.  
        // The update of the descriptor occurs only if the descriptor bits in
        // memory do not reflect that and the access instigates a write.
        if (AArch64.SettingDirtyStatePermitted(fault) &&
            walkparams.ha == '1' && walkparams.hd == '1' &&
            descriptor<51> == '1' && // Descriptor DBM bit
            IsAtomicRW(acctype) && iswrite) &&
            (!acctype IN {AccType_AT, AccType_ATPAN, AccType_IC, AccType_DC})) then
            // Clear descriptor AP[2] bit permitting stage 1 writes
            new_desc<7> = '0';
        \n    "}
AddressDescriptor descupdateaddress;
FaultRecord s2fault;
// Either the access flag was clear or AP<2> is set
if new_desc != descriptor then
  if regime == Regime_EL10 && EL2Enabled() then
    s1aarch64 = TRUE;
    s2fs1walk = TRUE;
    aligned   = TRUE;
    iswrite   = TRUE;
    (s2fault, descupdateaddress) = AArch64.S2Translate(fault, descaddress, s1aarch64,
                                                          ss, s2fs1walk, AccType_ATOMICRW,
                                                          aligned, iswrite, ispriv);

  if s2fault.statusCode != Fault_None then
    return (s2fault, AddressDescriptor UNKNOWN);
  else
    descupdateaddress = descaddress;

    (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                                                walkparams.ee, descupdateaddress);

  until new_desc == descriptor || mem_desc == new_desc;
if fault.statusCode != Fault_None then
  return (fault, AddressDescriptor UNKNOWN);

// Output Address
oa = StageOA(va, walkparams.tgx, walkstate);
MemoryAttributes memattrs;
if (acctype == AccType_IFETCH &&
    (walkstate.memattrs.memtype == MemType_Device || !AArch64.S1ICacheEnabled(regime))) then
  // Treat memory attributes as Normal Non-Cacheable
  memattrs = NormalNCMemAttr();
  memattrs.xs = walkstate.memattrs.xs;
elsif (acctype != AccType_IFETCH && !AArch64.S1DCacheEnabled(regime) &&
      walkstate.memattrs.memtype == MemType_Normal) then
  // Treat memory attributes as Normal Non-Cacheable
  memattrs = NormalNCMemAttr();
  memattrs.xs = walkstate.memattrs.xs;
// The effect of SCTLR_ELx.C when '0' is Constrained UNPREDICTABLE
// on the Tagged attribute
if HaveMTE2Ext() && walkstate.memattrs.tagged then
  memattrs.tagged = ConstrainUnpredictableBool(Unpredictable_S1CTAGGED);
else
  memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
  memattrs.shareability = walkstate.memattrs.shareability;
else
  memattrs.shareability = EffectiveShareability(memattrs);

if acctype == AccType_ATOMICLS64 && memattrs.memtype == MemType_Normal then
  if memattrs.inner.attrs != MemAttr_NC || memattrs.outer.attrs != MemAttr_NC then
    fault.statusCode = Fault_Exclusive;
    return (fault, AddressDescriptor UNKNOWN);

ipa = CreateAddressDescriptor(va, oa, memattrs);
return (fault, ipa);
AArch64/translation/vmsa_translation/AArch64.S2Translate

// AArch64.S2Translate()
// =====================
// Translate stage 1 IPA to PA and combine memory attributes

(FaultRecord, AddressDescriptor) AArch64.S2Translate(FaultRecord fault_in, AddressDescriptor ipa, 
    boolean s1aarch64, SecurityState ss, 
    boolean s2fs1walk, AccType accctype, 
    boolean aligned, boolean iswrite, 
    boolean ispriv);

walkparams = AArch64.GetS2TTWParams(ss, ipa.paddress.paspace, s1aarch64);
FaultRecord fault = fault_in;

// Prepare fault fields in case a fault is detected
fault.statuscode  = Fault_None; // Ignore any faults from stage 1
fault.secondstage = TRUE;
fault.s2fs1walk   = s2fs1walk;
fault.ipaddress   = ipa.paddress;

if walkparams.vm != '1' then
    // Stage 2 translation is disabled
    return (fault, ipa);
if (AArch64.S2InvalidTxSZ(walkparams, s1aarch64) || 
    AArch64.S2InvalidSL(walkparams) || 
    AArch64.S2InconsistentSL(walkparams) || 
    AArch64.IPAIsOutOfRange(ipa.paddress.address, walkparams)) then
    fault.statuscode = Fault_Translation;
fault.level      = 0;
return (fault, AddressDescriptor UNKNOWN);

AddressDescriptor descaddress;
TTWState walkstate;
bits(64) descriptor;
bits(64) new_desc;
bits(64) mem_desc;
repeat
    (fault, descaddress, walkstate, descriptor) = AArch64.S2Walk(fault, ipa, walkparams, ss, 
        accctype, iswrite, s1aarch64);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);
    if AArch64.S2HasAlignmentFault(acctype, aligned, walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elsif IsAtomicRW(acctype) then
        if AArch64.S2HasPermissionsFault(s2fs1walk, walkstate, ss, walkparams, 
            ispriv, acctype, FALSE) then
            // The Permission fault was not caused by lack of write permissions
            fault.statuscode = Fault_Permission;
            fault.write      = FALSE;
        elsif AArch64.S2HasPermissionsFault(s2fs1walk, walkstate, ss, walkparams, 
            ispriv, acctype, TRUE) then
            // The Permission fault was caused by lack of write permissions.
            // However, HW updates, which are atomic writes for stage 1
            // descriptors, permissions fault reflect the original access.
            fault.statuscode = Fault_Permission;
            if !fault.s2fs1walk then
                fault.write = TRUE;
            elseif AArch64.S2HasPermissionsFault(s2fs1walk, walkstate, ss, walkparams, 
                ispriv, acctype, iswrite) then
                fault.statuscode = Fault_Permission;
                new_desc = descriptor;
                if walkparams.ha == '1' & AArch64.SettingAccessFlagPermitted(fault) then
                    // Set descriptor AF bit
                    new_desc<10> = '1';
        else
            fault.statuscode = Fault_None;
            if fault.secondstage then
                fault.secondstage = FALSE;
            end
        end
    else
        fault.statuscode = Fault_None;
        if fault.secondstage then
            fault.secondstage = FALSE;
        end
    end
end
// If HW update of dirty bit is enabled, the walk state permissions // will already reflect a configuration permitting writes. // The update of the descriptor occurs only if the descriptor bits in // memory do not reflect that and the access instigates a write.
if (AArch64.SettingDirtyStatePermitted(fault) &&
    walkparams.ha == '1' &&
    walkparams.hd == '1' &&
    descriptor<51> == '1' && // Descriptor DBM bit
    (IsAtomicRW(acctype) || iswrite) &&
    !(acctype IN {AccType_AT, AccType_ATPAN, AccType_IC, AccType_DC})) then
    // Set descriptor S2AP[1] bit permitting stage 2 writes
    new_desc<7> = '1';

    // Either the access flag was clear or S2AP<1> is clear
    if new_desc != descriptor then
        (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
            walkparams.ee, descaddress);

    until new_desc == descriptor || mem_desc == new_desc;

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    ipa_64 = ZeroExtend(ipa.paddress.address, 64);
    // Output Address
    oa = StageOA(ipa_64, walkparams.tgx, walkstate);
    MemoryAttributes s2_memattrs;
    if ((s2fs1walk &&
        walkstate.memattrs.memtype == MemType_Device && walkparams.ptw == '0') ||
        (acctype == AccType_IFETCH &&
            walkstate.memattrs.memtype == MemType_Device || HCR_EL2.ID == '1') ||
        (acctype != AccType_IFETCH &&
            walkstate.memattrs.memtype == MemType_Normal && HCR_EL2.CD == '1')) then
        // Treat memory attributes as Normal Non-Cacheable
        s2_memattrs = NormalNCMemAttr();
        s2_memattrs.xs = walkstate.memattrs.xs;
    else
        s2_memattrs = walkstate.memattrs;

    if !s2fs1walk && acctype == AccType_ATOMICLS64 && s2_memattrs.memtype == MemType_Normal then
        if s2_memattrs.inner.attrs != MemAttr_NC || s2_memattrs.outer.attrs != MemAttr_NC then
            fault.statuscode = Fault_Exclusive;
            return (fault, AddressDescriptor UNKNOWN);

    MemoryAttributes memattrs;
    if walkparams.fwb == '0' then
        memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs);
    else
        memattrs = s2_memattrs;

    pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);
    return (fault, pa);

aarch64/translation/vmsa_translation/AArch64.SettingAccessFlagPermitted

// AArch64.SettingAccessFlagPermitted()
// ==============
// Determine whether the access flag could be set by HW given the fault status

boolean AArch64.SettingAccessFlagPermitted(FaultRecord fault)
    if fault.statuscode == Fault_None then
        return TRUE;
    elseif fault.statuscode IN {Fault_Alignment, Fault_Permission} then
        return ConstrainUnpredictableBool(Unpredictable_AFUPDATE);
else
    return FALSE;

aarch64/translation/vmsa_translation/AArch64.SettingDirtyStatePermitted

// AArch64.SettingDirtyStatePermitted()
// ====================================
// Determine whether the dirty state could be set by HW given the fault status

boolean AArch64.SettingDirtyStatePermitted(FaultRecord fault)
if fault.statuscode == Fault_None then
    return TRUE;
elsif fault.statuscode == Fault_Alignment then
    return ConstrainUnpredictableBool(Unpredictable_DBUPDATE);
else
    return FALSE;

aarch64/translation/vmsa_translation/AArch64.TranslateAddress

// AArch64.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) va, AccType acctype, boolean iswrite, boolean aligned, integer size)
result = AArch64.FullTranslate(va, acctype, iswrite, aligned);
if !IsFault(result) && acctype != AccType_IFETCH then
    result.fault = AArch64.CheckDebug(va, acctype, iswrite, size);
if HaveRME() && !IsFault(result) && (acctype != AccType_DC || boolean IMPLEMENTATION_DEFINED "GPC Fault on DC operations") then
    accdesc = CreateAccessDescriptor(acctype);
    result.fault.gpcf = GranuleProtectionCheck(result, accdesc);
    if result.fault.gpcf.gpf != GPCF_None then
        result.fault.statuscode = Fault_GPCFOnOutput;
        result.fault.paddress = result.paddress;
        result.fault.acctype = acctype;
        result.fault.write = iswrite;
    if !IsFault(result) && acctype == AccType_IFETCH then
        result.fault = AArch64.CheckDebug(va, acctype, iswrite, size);
// Update virtual address for abort functions
result.vaddress = ZeroExtend(va, 64);
return result;

aarch64/translation/vmsa_ttentry/AArch64.BlockDescSupported

// AArch64.BlockDescSupported()
// ============================
// Determine whether a block descriptor is valid for the given granule size
// and level

boolean AArch64.BlockDescSupported(bit ds, TGx tgx, integer level)
case tgx of
    when TGx_4KB return level == 2 || level == 1 || (level == 0 && ds == '1');
    when TGx_16KB return level == 2 || (level == 1 && ds == '1');
    when TGx_64KB return level == 2 || (level == 1 && AArch64.PAMax() == 52);
return FALSE;

aarch64/translation/vmsa_ttentry/AArch64.BlocknTFaults

// AArch64.BlocknTFaults()
// =======================
// Identify whether the nT bit in a block descriptor is effectively set
// causing a translation fault

boolean AArch64.BlocknTFaults(bits(64) descriptor)
    if !HaveBlockBBM() then
        return FALSE;
    bbm_level = AArch64.BlockBBMSupportLevel();
    nT_faults = boolean IMPLEMENTATION_DEFINED "BBM level 1 or 2 support nT bit causes Translation Fault";
    return bbm_level IN {1, 2} && descriptor<16> == '1' && nT_faults;

aarch64/translation/vmsa_ttentry/AArch64.ContiguousBit

// AArch64.ContiguousBit()
// =======================
// Get the value of the contiguous bit

bit AArch64.ContiguousBit(TGx tgx, integer level, bits(64) descriptor)
    if tgx == TGx_64KB && level == 1 then
        return '0'; // RES0
    if tgx == TGx_16KB && level == 1 then
        return '0'; // RES0
    if tgx == TGx_4KB  && level == 0 then
        return '0'; // RES0
    return descriptor<52>;

aarch64/translation/vmsa_ttentry/AArch64.DecodeDescriptorType

// AArch64.DecodeDescriptorType()
// ==============================
// Determine whether the descriptor is a page, block or table

DescriptorType AArch64.DecodeDescriptorType(bits(64) descriptor, bit ds, TGx tgx, integer level)
    if descriptor<1:0> == '11' && level == FINAL_LEVEL then
        return DescriptorType_Page;
    elsif descriptor<1:0> == '11' then
        return DescriptorType_Table;
    elsif descriptor<1:0> == '01' then
        if AArch64.BlockDescSupported(ds, tgx, level) then
            return DescriptorType_Block;
        else
            return DescriptorType_Invalid;
else
    return DescriptorType_Invalid;

aarch64/translation/vmsa_ttentry/AArch64.S1ApplyOutputPerms

// AArch64.S1ApplyOutputPerms()
// ============================
// Apply output permissions encoded in stage 1 page/block descriptors

Permissions AArch64.S1ApplyOutputPerms(Permissions permissions_in, bits(64) descriptor,
                                            Regime regime, S1TTWParams walkparams)

    Permissions permissions = permissions_in;
    if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
        permissions.ap<2:1> = descriptor<7>:3';
        permissions.pxn  = descriptor<54>;
    elsif HasUnprivileged(regime) then
        permissions.ap<2:1> = descriptor<7:6>;
        permissions.pxn  = descriptor<54>;
    else
        permissions.ap<2:1> = descriptor<7>:1';
        permissions.xn   = descriptor<54>;
    // Descriptors marked with DBM set have the effective value of AP[2] cleared.
    // This implies no Permission faults caused by lack of write permissions are
    // reported, and the Dirty bit can be set.
    if walkparams.ha == '1' && walkparams.hd == '1' & descriptor<51> == '1' then
        permissions.ap<2> = '0';
    return permissions;

aarch64/translation/vmsa_ttentry/AArch64.S1ApplyTablePerms

// AArch64.S1ApplyTablePerms()
// ===========================
// Apply hierarchical permissions encoded in stage 1 table descriptors

Permissions AArch64.S1ApplyTablePerms(Permissions permissions_in, bits(64) descriptor,
                                                Regime regime, S1TTWParams walkparams)

    Permissions permissions = permissions_in;
    bits(2) ap_table;
    bit pxn_table;
    bit uxn_table;
    bit xn_table;
    if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
        ap_table = descriptor<62>:3';
        pxn_table = descriptor<60>;
        permissions.ap_table  = permissions.ap_table  OR ap_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;
    elsif HasUnprivileged(regime) then
        ap_table = descriptor<62:61>;
        uxn_table = descriptor<60>;
        pxn_table = descriptor<59>;
        permissions.ap_table  = permissions.ap_table  OR ap_table;
        permissions.uxn_table = permissions.uxn_table OR uxn_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;
    else
        ap_table = descriptor<62>:3';
        xn_table = descriptor<60>;
        permissions.ap_table  = permissions.ap_table  OR ap_table;
        permissions.xn_table = permissions.xn_table OR xn_table;

return permissions;

aarch64/translation/vmsa_ttentry/AArch64.S2ApplyOutputPerms

// AArch64.S2ApplyOutputPerms()
// Apply output permissions encoded in stage 2 page/block descriptors
Permissions AArch64.S2ApplyOutputPerms(bits(64) descriptor, S2TTWParams walkparams)
Permissions permissions;
permissions.s2ap = descriptor<7:6>;
permissions.s2xn = descriptor<54>;
if HaveExtendedExecuteNeverExt() then
permissions.s2xnx = descriptor<53>;
else
permissions.s2xnx = '0';

// Descriptors marked with DBM set have the effective value of S2AP[1] set.
// This implies no Permission faults caused by lack of write permissions are reported, and the Dirty bit can be set.
if walkparams.ha == '1' && walkparams.hd == '1' && descriptor<51> == '1' then
permissions.s2ap<1> = '1';
return permissions;

aarch64/translation/vmsa_walk/AArch64.S1InitialTTWState

// AArch64.S1InitialTTWState()
// Set properties of first access to translation tables in stage 1
TTWState AArch64.S1InitialTTWState(S1TTWParams walkparams, bits(64) va, Regime regime, SecurityState ss)
TTWState    walkstate;
FullAddress tablebase;
Permissions permissions;
startlevel  = AArch64.S1StartLevel(walkparams);
ttbr        = AArch64.S1TTBR(regime, va);
case ss of
when SS_Secure    tablebase.paspace = PAS_Secure;
when SS_NonSecure tablebase.paspace = PAS_NonSecure;
when SS_Root      tablebase.paspace = PAS_Root;
when SS_Realm     tablebase.paspace = PAS_Realm;
tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz, walkparams.ps, walkparams.ds, walkparams.tgx, startlevel);
permissions.ap_table = '00';
if HasUnprivileged(regime) then
permissions.uxn_table = '0';
else
permissions.xn_table  = '0';
walkstate.baseaddress = tablebase;
walkstate.level       = startlevel;
walkstate.istable     = TRUE;
// In regimes that support global and non-global translations, translation table entries from lookup levels other than the final level of lookup are treated as being non-global
walkstate.nG = if HasUnprivileged(regime) then '1' else '0';
walkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
walkstatepermissions = permissions;
return walkstate;

aarch64/translation/vmsa_walk/AArch64.S1NextWalkStateLast

// AArch64.S1NextWalkStateLast()
// =======================================
// Decode stage 1 page or block descriptor as output to this stage of translation

TTWState AArch64.S1NextWalkStateLast(TTWState currentstate, Regime regime, SecurityState ss,
S1TTWParams walkparams, bits(64) descriptor)
TTWState nextstate;

FullAddress baseaddress;

if currentstate.level == FINAL_LEVEL then
    baseaddress.address = AArch64.PageBase(descriptor, walkparams.ds, walkparams.tgx);
else
    baseaddress.address = AArch64.BlockBase(descriptor, walkparams.ds, walkparams.tgx, currentstate.level);

if currentstate.baseaddress.paspace == PAS_Secure then
    // Determine PA space of the block from NS bit
    baseaddress.paspace = if descriptor<5> == '0' then PAS_Secure else PAS_NonSecure;
elsif currentstate.baseaddress.paspace == PAS_Root then
    // Determine PA space of the block from NSE and NS bits
    case descriptor<11,5> of
    when '00' baseaddress.paspace = PAS_Secure;
    when '01' baseaddress.paspace = PAS_NonSecure;
    when '10' baseaddress.paspace = PAS_Root;
    when '11' baseaddress.paspace = PAS_Realm;
elsif (currentstate.baseaddress.paspace == PAS_Realm &&
regime IN {Regime_EL2, Regime_EL20}) then
    // Realm EL2 and EL2&0 regimes have a stage 1 NS bit
    baseaddress.paspace = if descriptor<5> == '0' then PAS_Realm else PAS_NonSecure;
elsif currentstate.baseaddress.paspace == PAS_Realm then
    // Realm EL1&0 regime does not have a stage 1 NS bit
    baseaddress.paspace = PAS_Realm;
else
    baseaddress.paspace = PAS_NonSecure;
nextstate.istable = FALSE;
nextstate.level = currentstate.level;
nextstate.baseaddress = baseaddress;

attrindx = descriptor<4:2>;
sh = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
attr = MAIRAttr(UInt(attrindx), walkparams.mair);
s1aarch64 = TRUE;

nextstate.memattrs = S1DecodeMemAttrs(attr, sh, s1aarch64);
nextstate.permissions = AArch64.S1ApplyOutputPerms(currentstate.permissions, descriptor, regime, walkparams);
nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, currentstate.level, descriptor);

if !HasUnprivileged(regime) then
    nextstate.nG = '0';
elsif ss == SS_Secure &
currentstate.baseaddress.paspace == PAS_NonSecure then
    // In Secure state, a translation must be treated as non-global,
    // regardless of the value of the nG bit,
    // if NSTable is set to 1 at any level of the translation table walk
    nextstate.nG = '1';
else
    nextstate.nG = descriptor<11>;

nextstate.guardedpage = descriptor<50>;

return nextstate;

aarch64/translation/vmsa_walk/AArch64.S1NextWalkStateTable

// AArch64.S1NextWalkStateTable()
// ==============================
// Decode stage 1 table descriptor to transition to the next level

TTWState AArch64.S1NextWalkStateTable(TTWState currentstate, Regime regime, S1TTWParams walkparams, bits(64) descriptor)

TTWState    nextstate;
FullAddress tablebase;

Tablebase.address = AArch64.NextTableBase(descriptor, walkparams.ds, walkparams.tgx);
if currentstate.baseaddress.paspace == PAS_Secure then
    // Determine PA space of the next table from NSTable bit
    tablebase.paspace = if descriptor<63> == '0' then PAS_Secure else PAS_NonSecure;
else
    // Otherwise bit 63 is RES0 and there is no NSTable bit
    tablebase.paspace = currentstate.baseaddress.paspace;

nextstate.istable     = TRUE;
nextstate.nG          = currentstate.nG;
nextstate.level       = currentstate.level + 1;
nextstate.baseaddress = tablebase;
nextstate.memattrs    = currentstate.memattrs;

if walkparams.hpd == '0' then
    nextstate.permissions = AArch64.S1ApplyTablePerms(currentstate.permissions, descriptor, regime, walkparams);
else
    nextstate.permissions = currentstate.permissions;

return nextstate;

aarch64/translation/vmsa_walk/AArch64.S1Walk

// AArch64.S1Walk()
// ================
// Traverse stage 1 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

(FaultRecord, AddressDescriptor, TTWState, bits(64)) AArch64.S1Walk(FaultRecord fault_in, S1TTWParams walkparams, bits(64) va, Regime regime, SecurityState ss, AccType acctype, boolean iswrite_in, boolean ispriv)

FaultRecord fault = fault_in;
boolean iswrite = iswrite_in;
if HasUnprivileged(regime) && AArch64.S1EPD(regime, va) == '1' then
    fault.statuscode = Fault_Translation;
    fault.level = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

walkstate = AArch64.S1InitialTTWState(walkparams, va, regime, ss);

// Detect Address Size Fault by TTB
if AArch64.0A0OutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
    fault.statuscode = Fault_AddressSize;
    fault.level = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

bits(64) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = va;
if !AArch64.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
        !boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1") then
        walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
    else
        walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

DescriptorType desctype;
repeat
    fault.level = walkstate.level;
    FullAddress descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.tgx,
                walkparams.txsz, va,
                walkstate.baseaddress);

    walkaddress.paddress = descaddress;

    if regime == Regime_EL10 & EL2Enabled() then
        slaarch64 = TRUE;
        s2fs1walk = TRUE;
        aligned = TRUE;
        iswrite = FALSE;
        (s2fault, s2walkaddress) = AArch64.S2Translate(fault, walkaddress, slaarch64, ss,
                         s2fs1walk, AccType_TTW, aligned,
                         iswrite, ispriv);

        if s2fault.statuscode != Fault_None then
            return (s2fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

        (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, fault, 64);
    else
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault, 64);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds, walkparams.tgx,
                                walkstate.level);

case desctype of
    when DescriptorType_Table
        walkstate = AArch64.S1NextWalkStateTable(walkstate, regime, walkparams,
                                   descriptor);

        // Detect Address Size Fault by table descriptor
        if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
            fault.statuscode = Fault_AddressSize;
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    when DescriptorType_Page, DescriptorType_Block
        walkstate = AArch64.S1NextWalkStateLast(walkstate, regime, ss,
                                   walkparams, descriptor);

    when DescriptorType_Invalid
        fault.statuscode = Fault_Translation;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    otherwise
        Unreachable();
until desctype IN {DescriptorType_Page, DescriptorType_Block};

if (walkstate.contiguous == '1' &
    AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx, walkstate.level)) then
    fault.statuscode = Fault_Translation;
elsif desctype == DescriptorType_Block & AArch64.BlocknTFaults(descriptor) then
    fault.statuscode = Fault_Translation;
// Detect Address Size Fault by final output
elsif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
    fault.statuscode = Fault_AddressSize;
// Check descriptor AF bit
elsif (descriptor<10> == '0' & walkparams.ha == '0' &
    !(acctype IN {AccType_DC, AccType_IC} &
    !(acctype DEFINED "Generate access flag fault on IC/DC operations"))) then
    fault.statuscode = Fault_AccessFlag;
return (fault, walkaddress, walkstate, descriptor);

aarch64/translation/vmsa_walk/AArch64.S2InitialTTWState

// AArch64.S2InitialTTWState()
// =======================================
// Set properties of first access to translation tables in stage 2

TTWState AArch64.S2InitialTTWState(SecurityState ss, TT2TTWParams walkparams)

    TTWState    walkstate;
    FullAddress tablebase;

    ttbr        = VTTBR_EL2;
    startlevel  = AArch64.S2StartLevel(walkparams);
    case ss of
        when SS_NonSecure tablebase.paspace = PAS_NonSecure;
        when SS_Realm    tablebase.paspace = PAS_Realm;
    end;

    tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz, walkparams.ps, walkparams.ds,
                                             walkparams.tgx, startlevel);

    walkstate.baseaddress = tablebase;
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return walkstate;

aarch64/translation/vmsa_walk/AArch64.S2NextWalkStateLast

// AArch64.S2NextWalkStateLast()
// =======================================
// Decode stage 2 page or block descriptor as output to this stage of translation

TTWState AArch64.S2NextWalkStateLast(TTWState currentstate, SecurityState ss,
                                     TT2TTWParams walkparams, AddressDescriptor ipa,
                                     bits(64) descriptor)

    TTWState    nextstate;
    FullAddress baseaddress;

    if ss == SS_Secure then
        baseaddress.paspace = AArch64.SS2OutputPASpace(walkparams, ipa.paddress.paspace);
    elsif ss == SS_Realm then
        if descriptor<55> == '1' then
            baseaddress.paspace = PAS_NonSecure;
        else
            baseaddress.paspace = PAS_Realm;
        end;
    else
        baseaddress.paspace = PAS_Realm;
    end;

    return nextstate;
baseaddress.paspace = PAS_NonSecure;
if currentstate.level == FINAL_LEVEL then
  baseaddress.address = AArch64.PageBase(descriptor, walkparams.ds, walkparams.tgx);
else
  baseaddress.address = AArch64.BlockBase(descriptor, walkparams.ds, walkparams.tgx, currentstate.level);
nextstate.istable     = FALSE;
nextstate.level       = currentstate.level;
nextstate.baseaddress = baseaddress;
nextstate.permissions = AArch64.S2ApplyOutputPerms(descriptor, walkparams);
s2_attr = descriptor<5:2>;
s2_sh   = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
s2_fnxs = descriptor<11>;
if walkparams.fwb == '1' then
  nextstate.memattrs = AArch64.S2ApplyFwBMemAttrs(ipa.memattrs, s2_attr, s2_sh);
  if s2_attr<1:0> == '10' then // Force writeback
    nextstate.memattrs.xs = '0';
  else
    nextstate.memattrs.xs = if s2_fnxs == '1' then '0' else ipa.memattrs.xs;
else
  s2aarch64 = TRUE;
  nextstate.memattrs = S2DecodeMemAttrs(s2_attr, s2_sh, s2aarch64);
  nextstate.memattrs.xs = if s2_fnxs == '1' then '0' else ipa.memattrs.xs;
nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, currentstate.level, descriptor);
return nextstate;

aarch64/translation/vmsa_walk/AArch64.S2NextWalkStateTable

// AArch64.S2NextWalkStateTable()
// ==============================
// Decode stage 2 table descriptor to transition to the next level
TTWState AArch64.S2NextWalkStateTable(TTWState currentstate, S2TTWParams walkparams, bits(64) descriptor)
TTWState    nextstate;
FullAddress tablebase;
tablebase.address = AArch64.NextTableBase(descriptor, walkparams.ds, walkparams.tgx);
tablebase.paspace = currentstate.baseaddress.paspace;
nextstate.istable     = TRUE;
nextstate.level       = currentstate.level + 1;
nextstate.baseaddress = tablebase;
nextstate.memattrs    = currentstate.memattrs;
return nextstate;

aarch64/translation/vmsa_walk/AArch64.S2Walk

// AArch64.S2Walk()
// ================
// Traverse stage 2 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor
(FaultRecord, AddressDescriptor, TTWState, bits(64)) AArch64.S2Walk(
  FaultRecord fault_in, AddressDescriptor ipa, S2TTWParams walkparams, SecurityState ss,
  AccType acctype, boolean iswrite, boolean s1aarch64)

FaultRecord fault = fault_in;
ipa_64 = ZeroExtend(ipa.paddress.address, 64);
TlwState walkstate;
if ss == SS_Secure then
    walkstate = AArch64.SS2InitialTTWState(walkparams, ipa.paddress.paspace);
else
    walkstate = AArch64.S2InitialTTWState(ss, walkparams);

// Detect Address Size Fault by TTB
if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
    fault.statuscode = Fault_AddressSize;
    fault.level = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

bits(64) descriptor;
AddressDescriptor walkaddress;
walkaddress.vaddress = ipa.vaddress;
if HCR_EL2.CD == '1' then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;
walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

DescriptorType desctype;
repeat
    fault.level = walkstate.level;
    FullAddress descaddress;
    if walkstate.level == AArch64.S2StartLevel(walkparams) then
        // Initial lookup might index into concatenated tables
        descaddress = AArch64.S2SLTTEntryAddress(walkparams, ipa.paddress.address, walkstate.baseaddress);
    else
        ipa_64 = ZeroExtend(ipa.paddress.address, 64);
        descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.tgx, walkparams.txsz, ipa_64, walkstate.baseaddress);
    walkaddress.paddress = descaddress;
    (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault, 64);
    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds, walkparams.tgx, walkstate.level);
case desctype of
    when DescriptorType_Table
        walkstate = AArch64.S2NextWalkStateTable(walkstate, walkparams, descriptor);
        // Detect Address Size Fault by table descriptor
        if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
            fault.statuscode = Fault_AddressSize;
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    when DescriptorType_Page, DescriptorType_Block
        walkstate = AArch64.S2NextWalkStateLast(walkstate, ss, walkparams, ipa, descriptor);

    when DescriptorType_Invalid
        fault.statuscode = Fault_Translation;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    otherwise
        Unreachable();
until desctype IN {DescriptorType_Page, DescriptorType_Block};
if (walkstate.contiguous == '1' &&
    AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx, walkstate.level)) then
    fault.statuscode = Fault_Translation;
elsif desctype == DescriptorType_Block && AArch64.BlocknTFaults(descriptor) then
    fault.statuscode = Fault_Translation;
// Detect Address Size Fault by final output
elsif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
    fault.statuscode = Fault_AddressSize;
// Check descriptor AF bit
elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
    ![acctype IN {AccType_DC, AccType_IC} &&
    !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations") then
    fault.statuscode = Fault_AccessFlag;
return (fault, walkaddress, walkstate, descriptor);

aarch64/translation/vmsa_walk/AArch64.SS2InitialTTWState

// AArch64.SS2InitialTTWState()
// ==================================================
// Set properties of first access to translation tables in Secure stage 2

TTWState AArch64.SS2InitialTTWState(S2TTWParams walkparams, PASpace ipaspace)

    TTWState walkstate;
    FullAddress tablebase;

    bits(64) ttbr;
    if ipaspace == PAS_Secure then
        ttbr = VSTTBR_EL2;
    else
        ttbr = VTTBR_EL2;

    if ipaspace == PAS_Secure then
        if walkparams.sw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;
    else
        if walkparams.nsw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;

    startlevel = AArch64.S2StartLevel(walkparams);
    tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz, walkparams.ps, walkparams.ds,
                                              walkparams.tgx, startlevel);

    walkstate.baseaddress = tablebase;
    walkstate.level = startlevel;
    walkstate.istable = TRUE;
    walkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return walkstate;

aarch64/translation/vmsa_walk/AArch64.SS2OutputPASpace

// AArch64.SS2OutputPASpace()
// ===============================
// Assign PA Space to output of Secure stage 2 translation

PASpace AArch64.SS2OutputPASpace(S2TTWParams walkparams, PASpace ipaspace)

    if ipaspace == PAS_Secure then
if walkparams.<sw,sa> == '00' then
    return PAS_Secure;
else
    return PAS_NonSecure;
else
    if walkparams.<sw,sa,nsw,nsa> == '0000' then
        return PAS_Secure;
    else
        return PAS_NonSecure;

aarch64/translation/vmsa_walkparams/AArch64.BBMSupportLevel

// AArch64.BBMSupportLevel()
// =========================
// Returns the level of FEAT_BBM supported

integer AArch64.BlockBBMSupportLevel()
    if !HaveBlockBBM() then
        return integer UNKNOWN;
    else
        return integer IMPLEMENTATION_DEFINED "Block BBM support level";

aarch64/translation/vmsa_walkparams/AArch64.GetS1TTWParams

// AArch64.GetS1TTWParams()
// ========================
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.

S1TTWParams AArch64.GetS1TTWParams(Regime regime, bits(64) va)
    S1TTWParams walkparams;
    varange = AArch64.GetVARange(va);
    case regime of
      when Regime_EL3  walkparams = AArch64.S1TTWParamsEL3();
      when Regime_EL2  walkparams = AArch64.S1TTWParamsEL2();
      when Regime_EL20 walkparams = AArch64.S1TTWParamsEL20(varange);
      when Regime_EL10 walkparams = AArch64.S1TTWParamsEL10(varange);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    min-txtsz = AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx);
    if UInt(walkparams.txsz) > maxtxsz then
        if !((boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum") then
            walkparams.txsz = maxtxsz<5:0>;
        elsif !Have52BitVAExt() && UInt(walkparams.txsz) < min-txtsz then
            if !((boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum") then
                walkparams.txsz = min-txtsz<5:0>;
            return walkparams;

aarch64/translation/vmsa_walkparams/AArch64.GetS2TTWParams

// AArch64.GetS2TTWParams()
// ========================
// Gather walk parameters for stage 2 translation

S2TTWParams AArch64.GetS2TTWParams(SecurityState ss, PASSpace ipaspace, boolean slaarch64)
    S2TTWParams walkparams;
    if ss == SS_NonSecure then
        walkparams = AArch64.NS2TTWParams(slaarch64);
    elsif HaveSecureEL2Ext() & ss == SS_Secure then
walkparams = AArch64.SS2TTWParams(ipaspace, slaarch64);
elsif ss == SS_Realm then
    // Realm stage 2 walk parameters are the same as for Non-secure
    walkparams = AArch64.NSS2TTWParams(slaarch64);
else
    Unreachable();

maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
mintxsz = AArch64.S2MinTxSZ(walkparams.ds, walkparams.tgx, slaarch64);
if UInt(walkparams.txsz) > maxtxsz then
    if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum") then
        walkparams.txsz = maxtxsz<5:0>;
    elsif !Have52BitPAExt() && UInt(walkparams.txsz) < mintxsz then
        if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum") then
            walkparams.txsz = mintxsz<5:0>;
    end
    return walkparams;

aarch64/translation/vmsa_walkparams/AArch64.GetVARange

// AArch64.GetVARange()
// ==============
// Determines if the VA that is to be translated lies in LOWER or UPPER address range.
VARange AArch64.GetVARange(bits(64) va)
    if va<55> == '0' then
        return VARange_LOWER;
    else
        return VARange_UPPER;

aarch64/translation/vmsa_walkparams/AArch64.HaveS1TG

// AArch64.HaveS1TG()
// ================
// Determine whether the given translation granule is supported for stage 1
boolean AArch64.HaveS1TG(TGx tx)
    case tx of
        when TGx_4KB  return boolean IMPLEMENTATION_DEFINED "Has 4K Translation Granule";
        when TGx_16KB return boolean IMPLEMENTATION_DEFINED "Has 16K Translation Granule";
        when TGx_64KB return boolean IMPLEMENTATION_DEFINED "Has 64K Translation Granule";

aarch64/translation/vmsa_walkparams/AArch64.HaveS2TG

// AArch64.HaveS2TG()
// ================
// Determine whether the given translation granule is supported for stage 2
boolean AArch64.HaveS2TG(TGx tx)
    assert HaveEL(EL2);
    if HaveGTGExt() then
        case tx of
            when TGx_4KB  return boolean IMPLEMENTATION_DEFINED "Has Stage 2 4K Translation Granule";
            when TGx_16KB return boolean IMPLEMENTATION_DEFINED "Has Stage 2 16K Translation Granule";
            when TGx_64KB return boolean IMPLEMENTATION_DEFINED "Has Stage 2 64K Translation Granule";
        else
            return AArch64.HaveS1TG(tx);
        end
    end
AARMv8 Pseudocode

J1.1 Pseudocode for AArch64 operation

```c
// AArch64.MaxTxSZ()
// ============
// Retrieve the maximum value of TxSZ indicating minimum input address size for both
// stages of translation

integer AArch64.MaxTxSZ(TGx tgx)
if HaveSmallTranslationTableExt() then
  case tgx of
    when TGx_4KB   return 48;
    when TGx_16KB  return 48;
    when TGx_64KB  return 47;
  return 39;

// AArch64.NSS2TTWParams()
// ================
// Gather walk parameters specific for Non-secure stage 2 translation

S2TTWParams AArch64.NSS2TTWParams(boolean s1aarch64)
S2TTWParams walkparams;

  walkparams.vm   = HCR_EL2.VM OR HCR_EL2.DC;
  walkparams.tgx  = AArch64.S2DecodeTG0(VTCR_EL2.TG0);
  walkparams.txsz = VTCR_EL2.T0SZ;
  walkparams.sl0  = VTCR_EL2.SL0;
  walkparams.ps   = VTCR_EL2.PS;
  walkparams.irgn = VTCR_EL2.IRGN0;
  walkparams.orgn = VTCR_EL2.ORGN0;
  walkparams.sh   = VTCR_EL2.SH0;
  walkparams.ee   = SCTLR_EL2.EE;

  walkparams.ptw = if HCR_EL2.TGE == '0' then HCR_EL2.PTW else '0';
  walkparams.fwb = if HaveStage2MemAttrControl() then HCR_EL2.FWB else '0';
  walkparams.ha  = if HaveAccessFlagUpdateExt() then VTCR_EL2.HA else '0';
  walkparams.hd  = if HaveDirtyBitModifierExt() then VTCR_EL2.HD else '0';
if walkparams.tgx IN [TGx_4KB, TGx_16KB] && Have52BitIPAAndPASpaceExt() then
  walkparams.ds = VTCR_EL2.DS;
else
  walkparams.ds = '0';
if walkparams.tgx == TGx_4KB && Have52BitIPAAndPASpaceExt() then
  walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
else
  walkparams.sl2 = '0';
walkparams.cmow = if HaveFeatCMOW() && IsHCRXEL2Enabled() then HCRX_EL2.CMOW else '0';
return walkparams;

// AArch64.PAMax()
// ================
// Returns the IMPLEMENTATION DEFINED maximum number of bits capable of representing
// physical address for this processor

integer AArch64.PAMax()
return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
```
aarch64/translation/vmsa_walkparams/AArch64.S1DCacheEnabled

// AArch64.S1DCacheEnabled()
// =========================
// Determine cacheability of stage 1 data accesses

boolean AArch64.S1DCacheEnabled(Regime regime)
case regime of
  when Regime_EL3  return SCTLR_EL3.C == '1';
  when Regime_EL2  return SCTLR_EL2.C == '1';
  when Regime_EL20 return SCTLR_EL2.C == '1';
  when Regime_EL10 return SCTLR_EL1.C == '1';

aarch64/translation/vmsa_walkparams/AArch64.S1DecodeTG0

// AArch64.S1DecodeTG0()
// ===============
// Decode stage 1 granule size configuration bits TG0

TGx AArch64.S1DecodeTG0(bits(2) tg0_in)
bits(2) tg0 = tg0_in;
TGx tgx;
if tg0 == '11' then
tg0 = bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size";
  case tg0 of
    when '00'   tgx = TGx_4KB;
    when '01'   tgx = TGx_64KB;
    when '10'   tgx = TGx_16KB;
  end case
if !AArch64.HaveS1TG(tgx) then
case bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size" of
  when '00'   tgx = TGx_4KB;
  when '01'   tgx = TGx_64KB;
  when '10'   tgx = TGx_16KB;
end case
return tgx;

aarch64/translation/vmsa_walkparams/AArch64.S1DecodeTG1

// AArch64.S1DecodeTG1()
// ===============
// Decode stage 1 granule size configuration bits TG1

TGx AArch64.S1DecodeTG1(bits(2) tg1_in)
bits(2) tg1 = tg1_in;
TGx tgx;
if tg1 == '00' then
tg1 = bits(2) IMPLEMENTATION_DEFINED "TG1 encoded granule size";
  case tg1 of
    when '00'   tgx = TGx_4KB;
    when '01'   tgx = TGx_64KB;
    when '10'   tgx = TGx_16KB;
  end case
if !AArch64.HaveS1TG(tgx) then
case bits(2) IMPLEMENTATION_DEFINED "TG1 encoded granule size" of
  when '10'   tgx = TGx_4KB;
  when '11'   tgx = TGx_64KB;
  when '01'   tgx = TGx_16KB;
end case

return tgx;

aarch64/translation/vmsa_walkparams/AArch64.S1EPD

// AArch64.S1EPD()
// ===============
// Determine whether stage 1 translation table walk is allowed for the VA range
bit AArch64.S1EPD(Regime regime, bits(64) va)
    assert HasUnprivileged(regime);
    varange = AArch64.GetVARange(va);
    case regime of
        when Regime_EL20 return if varange == VARange_LOWER then TCR_EL2.EPD0 else TCR_EL2.EPD1;
        when Regime_EL10 return if varange == VARange_LOWER then TCR_EL1.EPD0 else TCR_EL1.EPD1;

aarch64/translation/vmsa_walkparams/AArch64.S1Enabled

// AArch64.S1Enabled()
// ================
// Determine if stage 1 is enabled for the access type for this translation regime
boolean AArch64.S1Enabled(Regime regime, AccType acctype)
    case regime of
        when Regime_EL3  return SCTLR_EL3.M == '1';
        when Regime_EL2  return SCTLR_EL2.M == '1';
        when Regime_EL20 return SCTLR_EL2.M == '1';
        when Regime_EL10 return (!EL2Enabled() || HCR_EL2.<DC,TGE> == '00') && SCTLR_EL1.M == '1';

aarch64/translation/vmsa_walkparams/AArch64.S1ICacheEnabled

// AArch64.S1ICacheEnabled()
// ===============
// Determine cacheability of stage 1 instruction fetches
boolean AArch64.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime_EL3  return SCTLR_EL3.I == '1';
        when Regime_EL2  return SCTLR_EL2.I == '1';
        when Regime_EL20 return SCTLR_EL2.I == '1';
        when Regime_EL10 return SCTLR_EL1.I == '1';

aarch64/translation/vmsa_walkparams/AArch64.S1MinTxSZ

// AArch64.S1MinTxSZ()
// ===============
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 1
integer AArch64.S1MinTxSZ(bit ds, TGx tgx)
    if (Have52BitVAExt() & tgx == TGx_64KB) || ds == '1' then
        return 12;
    return 16;


```
// AArch64.S1TTBR()
// ================
// Identify stage 1 table base register for the acting translation regime

def AArch64.S1TTBR(Regime regime, bits(64) va)
   varange = AArch64.GetVARange(va);
   case regime of
      when Regime_EL3  return TTBR0_EL3;
      when Regime_EL2  return TTBR0_EL2;
      when Regime_EL20 return if varange == VARange_LOWER then TTBR0_EL2 else TTBR1_EL2;
      when Regime_EL10 return if varange == VARange_LOWER then TTBR0_EL1 else TTBR1_EL1;

// AArch64.S1TTWParamsEL10()
// =========================
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled)

def AArch64.S1TTWParamsEL10(VARange varange)
   S1TTWParams walkparams;
   if varange == VARange_LOWER then
      walkparams.tgx  = AArch64.S1DecodeTG0(TCR_EL1.TG0);
      walkparams.txsz = TCR_EL1.T0SZ;
      walkparams.irgn = TCR_EL1.IRGN0;
      walkparams.orgn = TCR_EL1.ORGN0;
      walkparams.sh   = TCR_EL1.SH0;
      walkparams.tbi  = TCR_EL1.TBI0;
      walkparams.nfd  = if HaveSVE() || HaveTME() then TCR_EL1.NFD0 else '0';
      walkparams.tbid = if HavePACExt() then TCR_EL1.TBID0 else '0';
      walkparams.e0pd = if HaveE0PDExt() then TCR_EL1.E0PD0 else '0';
      walkparams.hpd  = if AArch64.HaveHPDExt() then TCR_EL1.HPD0 else '0';
   else
      walkparams.tgx  = AArch64.S1DecodeTG1(TCR_EL1.TG1);
      walkparams.txsz = TCR_EL1.T1SZ;
      walkparams.irgn = TCR_EL1.IRGN1;
      walkparams.orgn = TCR_EL1.ORGN1;
      walkparams.sh   = TCR_EL1.SH1;
      walkparams.tbi  = TCR_EL1.TBI1;
      walkparams.nfd  = if HaveSVE() || HaveTME() then TCR_EL1.NFD1 else '0';
      walkparams.tbid = if HavePACExt() then TCR_EL1.TBID1 else '0';
      walkparams.e0pd = if HaveE0PDExt() then TCR_EL1.E0PD1 else '0';
      walkparams.hpd  = if AArch64.HaveHPDExt() then TCR_EL1.HPD1 else '0';
   end;
   walkparams.mair = MAIR_EL1;
   walkparams.wxn  = SCTLR_EL1.WXN;
   walkparams.ps   = TCR_EL1.IPS;
   walkparams.ee   = SCTLR_EL1.EE;
   walkparams.sif  = SCR_EL3.SIF;
   if EL2Enabled() then
      walkparams.dc = HCR_EL2.DC;
      walkparams.dct = if HaveMTE2Ext() then HCR_EL2.DCT else '0';
   end;
   if HaveTrapLoadStoreMultipleDeviceExt() then
      walkparams.ntlsmd = SCTLR_EL1.nTLSMD;
   else
      walkparams.ntlsmd = '1';
   end;
   if EL2Enabled() then
```
if HCR_EL2.<NV,NV1> == '01' then
    case ConstraintUnpredictable(Unpredictable_NVNV1) of
        when Constraint_NVNV1_00 walkparams.nv1 = '0';
        when Constraint_NVNV1_01 walkparams.nv1 = '1';
        when Constraint_NVNV1_11 walkparams.nv1 = '1';
    else
        walkparams.nv1 = HCR_EL2.NV1;
    end
else
    walkparams.nv1 = '0';
end
walkparams.epan = if HavePAN3Ext() then SCTLR_EL1.EPAN else '0';
walkparams.cmow = if HaveFeatCMOW() then SCTLR_EL1.CMOW else '0';
walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL1.HA else '0';
walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL1.HD else '0';
if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
    walkparams.ds = TCR_EL1.DS;
else
    walkparams.ds = '0';
end
return walkparams;

aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL2

// AArch64.S1TTWParamsEL2()
// ========================
// Gather stage 1 translation table walk parameters for EL2 regime
S1TTWParams AArch64.S1TTWParamsEL2()
S1TTWParams walkparams;
walkparams.tgx = AArch64.S1DecodeTG0(TCR_EL2.TG0);
walkparams.txsz = TCR_EL2.T0SZ;
walkparams.ps = TCR_EL2.PS;
walkparams.irgn = TCR_EL2.IRGN0;
walkparams.orgn = TCR_EL2.ORGN0;
walkparams.sh = TCR_EL2.SH0;
walkparams.tbi = TCR_EL2.TBI;
walkparams.mair = MAIR_EL2;
walkparams.wxn = SCTLR_EL2.WXN;
waklparams.ee = SCTLR_EL2.EE;
waklparams.sif = SCR_EL3.SIF;
walkparams.tbid = if HavePACExt() then TCR_EL2.TBID else '0';
waklparams.hpd = if AArch64.HaveHPDExt() then TCR_EL2.HPD else '0';
waklparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL2.HA else '0';
waklparams.hd = if HaveDirtyBitModifierExt() then TCR_EL2.HD else '0';
if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
    walkparams.ds = TCR_EL2.DS;
else
    walkparams.ds = '0';
end
return walkparams;

aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL20

// AArch64.S1TTWParamsEL20()
// =========================
// Gather stage 1 translation table walk parameters for EL2&0 regime
S1TTWParams AArch64.S1TTWParamsEL20(VARange varange)
S1TTWParams walkparams;
if varange == VARange_LOWER then
    walkparams.tgx = AArch64.S1DecodeTG0(TCR_EL2.TG0);
walkparams.txsz = TCR_EL2.T0SZ;
walkparams.irgn = TCR_EL2.IRGN0;
walkparams.orgn = TCR_EL2.ORGN0;
walkparams.sh = TCR_EL2.SHO;
walkparams.tbi = TCR_EL2.TB10;

walkparams.nfd = if HaveSVE() || HaveTME() then TCR_EL2.NFD0 else '0';
walkparams.tbid = if HavePACExt() then TCR_EL2.TBID0 else '0';
walkparams.epd = if HaveE0PDExt() then TCR_EL2.E0PD0 else '0';
walkparams.hpd = if AArch64.HaveHPDExt() then TCR_EL2.HPD0 else '0';
else
walkparams.tgx = AArch64.S1DecodeTG1(TCR_EL2.TG1);
walkparams.txsz = TCR_EL2.T1SZ;
walkparams.irgn = TCR_EL2.IRGN1;
walkparams.orgn = TCR_EL2.ORGN1;
walkparams.sh = TCR_EL2.SHI;
walkparams.tbi = TCR_EL2.TB11;

walkparams.nfd = if HaveSVE() || HaveTME() then TCR_EL2.NFD1 else '0';
walkparams.tbid = if HavePACExt() then TCR_EL2.TBID1 else '0';
walkparams.epd = if HaveE0PDExt() then TCR_EL2.E0PD1 else '0';
walkparams.hpd = if AArch64.HaveHPDExt() then TCR_EL2.HPD1 else '0';
walkparams.mair = MAIR_EL2;
walkparams.wxn = SCTLR_EL2.WXN;
walkparams.ps = TCR_EL2.IPS;
walkparams.ee = SCTLR_EL2.EE;
walkparams.sif = SCR_EL3.SIF;

if HaveTrapLoadStoreMultipleDeviceExt() then
    walkparams.ntlsmd = SCTLR_EL2.nTLSMD;
else
    walkparams.ntlsmd = '1';

walkparams.epan = if HavePAN3Ext() then SCTLR_EL2.EPAN else '0';
walkparams.cmow = if HaveFeatCMOW() then SCTLR_EL2.CMOW else '0';
walkparams.ha = if HaveAccessFlagUpdatedExt() then TCR_EL2.HA else '0';
walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL2.HD else '0';
if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
    walkparams.ds = TCR_EL2.DS;
else
    walkparams.ds = '0';

return walkparams;

aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL3

// AArch64.S1TTWParamsEL3()
// =========================
// Gather stage 1 translation table walk parameters for EL3 regime

S1TTWParams AArch64.S1TTWParamsEL3()
S1TTWParams walkparams;

walkparams.tgx = AArch64.S1DecodeTG0(TCR_EL3.TG0);
walkparams.txsz = TCR_EL3.T0SZ;
walkparams.ps = TCR_EL3.PS;
walkparams.irgn = TCR_EL3.IRGN0;
walkparams.orgn = TCR_EL3.ORGN0;
walkparams.sh = TCR_EL3.SHO;
walkparams.tbi = TCR_EL3.TBI;
walkparams.mair = MAIR_EL3;
walkparams.wxn = SCTLR_EL3.WXN;
walkparams.ee = SCTLR_EL3.EE;
walkparams.sif = SCR_EL3.SIF;
armv8_pseudocode

J1.1 Pseudocode for AArch64 operation

```c
walkparams_tcbid = if HavePACExt() then TCR_EL3.TBID else '0';
walkparams_hpd  = if AArch64.HaveHPDExt() then TCR_EL3.HPD else '0';
walkparams_ha   = if HaveAccessFlagUpdateExt() then TCR_EL3.HA else '0';
walkparams_hd   = if HaveDirtyBitModifierExt() then TCR_EL3.HD else '0';
if walkparams_tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
    walkparams_ds = TCR_EL3.DS;
else
    walkparams_ds = '0';
return walkparams;
```

```c
// AArch64.S2DecodeTG0()
// ================
// Decode stage 2 granule size configuration bits TG0
TGx AArch64.S2DecodeTG0(bits(2) tg0_in)
bits(2) tg0 = tg0_in;
TGx tgx;
if tg0 == '11' then
    tg0 = bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size";
    case tg0 of
        when '00'   tgx = TGx_4KB;
        when '01'   tgx = TGx_64KB;
        when '10'   tgx = TGx_16KB;
if !AArch64.HaveS2TG(tgx) then
    case bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size" of
        when '00'   tgx = TGx_4KB;
        when '01'   tgx = TGx_64KB;
        when '10'   tgx = TGx_16KB;
return tgx;
```

```c
// AArch64.S2MinTxSZ()
// ==============
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 2
integer AArch64.S2MinTxSZ(bit ds, TGx tgx, boolean s1aarch64)
ips = AArch64.PAMax();
if Have52BitPAExt() && tgx != TGx_64KB && ds == '0' then
    ips = Min(48, AArch64.PAMax());
min_txsz = 64 - ips;
if !s1aarch64 then
    // EL1 is AArch32
    min_txsz = Min(min_txsz, 24);
return min_txsz;
```

```c
// AArch64.SS2TTWParams()
// ================
// Gather walk parameters specific for secure stage 2 translation
S2TTWParams AArch64.SS2TTWParams(PASpace ipaspace, boolean s1aarch64)
```
S2TTWParams walkparams;
if ipaspace == PAS_Secure then
  walkparams.tgx = AArch64.S2DecodeTG0(VSTCR_EL2.TG0);
  walkparams.txsz = VSTCR_EL2.T0SZ;
  walkparams.sl0 = VSTCR_EL2.SL0;
  if walkparams.tgx == TGx_4KB && Have52BitIPAAndPASpaceExt() then
    walkparams.sl2 = VSTCR_EL2.SL2 AND VTCR_EL2.DS;
  else
    walkparams.sl2 = '0';
elsif ipaspace == PAS_NonSecure then
  walkparams.tgx = AArch64.S2DecodeTG0(VTCR_EL2.TG0);
  walkparams.txsz = VTCR_EL2.T0SZ;
  walkparams.sl0 = VTCR_EL2.SL0;
  if walkparams.tgx == TGx_4KB && Have52BitIPAAndPASpaceExt() then
    walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
  else
    walkparams.sl2 = '0';
else
  Unreachable();
walkparams.sw = VSTCR_EL2.SW;
walkparams.nsw = VTCR_EL2.NSW;
walkparams.sa = VSTCR_EL2.SA;
walkparams.nsa = VTCR_EL2.NSA;
walkparams.vm = HCR_EL2.VM OR HCR_EL2.DC;
walkparams.ps = VTCR_EL2.PS;
waklparams.irgn = VTCR_EL2.IRGN0;
walkparams.orgn = VTCR_EL2.ORGN0;
waklparams.sh = VTCR_EL2.SH0;
waklparams.ee = SCTLR_EL2.EE;
walkparams.ptw = if HCR_EL2.TGE == '0' then HCR_EL2.PTW else '0';
walkparams.fwb = if HaveStage2MemAttrControl() then HCR_EL2.FWB else '0';
walkparams.ha = if HaveAccessFlagUpdateExt() then VTCR_EL2.HA else '0';
walkparams.hd = if HaveDirtyBitModifierExt() then VTCR_EL2.HD else '0';
if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
  walkparams.ds = VTCR_EL2.DS;
else
  walkparams.ds = '0';
walkparams.cmow = if HaveFeatCMOW() && IsHCRXEL2Enabled() then HCRX_EL2.CMOW else '0';
return walkparams;

aarch64/translation/vmsa_walkparams/AArch64.VAMax

// AArch64.VAMax()
// ===============
// Returns the IMPLEMENTATION DEFINED maximum number of bits capable of representing
// the virtual address for this processor
integer AArch64.VAMax()
  return integer IMPLEMENTATION_DEFINED "Maximum Virtual Address Size";
J1.2 Pseudocode for AArch32 operation

This section holds the pseudocode for execution in AArch32 state. Functions that are listed in this section are identified as `AArch32.FunctionName`. Some of these functions have an equivalent AArch64 function, `AArch64.FunctionName`. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example `aarch32/debug/breakpoint`.

——— Note ————

Many AArch32 pseudocode functions have not been updated to show the constraints on the Armv7 UNPREDICTABLE behaviors that are described in Appendix K1 Architectural Constraints on UNPREDICTABLE Behaviors. Where AArch32 pseudocode shows something to be UNPREDICTABLE, check Appendix K1 for possible constraints on the permitted behavior.

The top-level sections of the AArch32 pseudocode hierarchy are:

- `aarch32/debug` on page J1-11258.
- `aarch32/exceptions` on page J1-11270.
- `aarch32/functions` on page J1-11290.
- `aarch32/translation` on page J1-11320.

J1.2.1 aarch32/debug

This section includes the following pseudocode functions:

- `aarch32/debug/VCRMatch/AArch32.VCRMatch` on page J1-11258.
- `aarch32/debug/pmu/AArch32.CheckForPMUOverflow` on page J1-11262.
- `aarch32/debug/pmu/AArch32.PMUCounterIsHyp` on page J1-11266.
- `aarch32/debug/pmu/AArch32.PMUCycle` on page J1-11266.
- `aarch32/debug/pmu/AArch32.PMUSwIncrement` on page J1-11267.
- `aarch32/debug/taakeexceptiondbg/AArch32.EnterHypModeInDebugState` on page J1-11267.
- `aarch32/debug/taakeexceptiondbg/AArch32.EnterModeInDebugState` on page J1-11267.
- `aarch32/debug/taakeexceptiondbg/AArch32.EnterMonitorModeInDebugState` on page J1-11268.

```c
// AArch32.VCRMatch()
// ==============

bool AArch32.VCRMatch(bits(32) vaddress)
{
    bool match = false;
    if UsingAArch32() && ELUsingAArch32(EL1) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        match_word = Zeros(32);
    return match;
}
```
ss = CurrentSecurityState();
if vaddress<31:5> == ExcVectorBase()<31:5> then
    if HaveEL(EL3) && ss == SS_NonSecure then
        match_word<UInt(vaddress<4:2>) + 24> = '1';  // Non-secure vectors
    else
        match_word<UInt(vaddress<4:2>) + 0> = '1';   // Secure vectors (or no EL3)
else
    if (HaveEL(EL3) && ELUsingAArch32(EL3) && vaddress<31:5> == MVBAR<31:5> &&
        ss == SS_Secure) then
        match_word<UInt(vaddress<4:2>) + 8> = '1';  // Monitor vectors
    // Mask out bits not corresponding to vectors.
    bits(32) mask;
    if !HaveEL(EL3) then
        mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0
    elsif !ELUsingAArch32(EL3) then
        mask = '11011110':'00000000':'00000000':'11011110'; // DBGVCR[15:8] are RES0
    else
        mask = '11011110':'00000000':'11011100':'11011110';
    match_word = match_word AND DBGVCR AND mask;
    match = !IsZero(match_word);
    if !IsZero(vaddress<1:0>) && match then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHDAPA);
    else
        match = FALSE;
    return match;

aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// -------------------------------------------------------------

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
// the state of the (DBGEN AND SPIDEN) signal.
if !HaveEL(EL3) && NonSecureOnlyImplementation() then return FALSE;
return DBGEN == HIGH && SPIDEN == HIGH;

aarch32/debug/breakpoint/AArch32.BreakpointMatch

// AArch32.BreakpointMatch()
// ----------------------------------------
// Breakpoint matching in an AAarch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress,
integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumBreakpointsImplemented();
    enabled = DBGBCR[n].E == '1';
ispriv = PSTATE.EL != EL0;
linked = DBGBCR[n].BT IN {'0x01'};
isbreakpnt = TRUE;
linked_to = FALSE;
state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC, linked, DBGBCR[n].LBN, isbreakpnt, ispriv);
(value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

if size == 4 then // Check second halfword
    // If the breakpoint address and BAS of an Address breakpoint match the address of the second halfword of an instruction, but not the address of the first halfword, it is CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug event.
    (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);
    if !value_match && match_i then
        value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
    if value_mismatch && !mismatch_i then
        value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);

    if vaddress<1> == '1' && DBGBCR[n].BAS == '1111' then // The above notwithstanding, if DBGBCR[n].BAS == '1111', then it is CONSTRAINED UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction at the address DBGVR[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
        if !value_mismatch then value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);

    match = value_match && state_match && enabled;
    mismatch = value_mismatch && state_match && enabled;

return (match, mismatch);

aarch32/debug/breakpoint/AArch32.BreakpointValueMatch

// AArch32.BreakpointValueMatch()
// -----------------------------
// The first result is whether an Address Match or Context breakpoint is programmed on the instruction at "address". The second result is whether an Address Mismatch breakpoint is programmed on the instruction, that is, whether the instruction should be stepped.

(boolean,boolean) AArch32.BreakpointValueMatch(integer n_in, bits(32) vaddress, boolean linked_to)

    // "n" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.
    integer n = n_in;

    if n >= NumBreakpointsImplemented() then
        Constraint c;
        (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1, Unpredictable_BPNOTIMPL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (FALSE,FALSE);

    if DBGBCR[n].E == '0' then return (FALSE,FALSE);

    context_aware = (n >= (NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented()));

    if BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    dbgtype = DBGBCR[n].BT;
    Constraint c;

    if ((dbgtype IN {'011x', '11xx'}) && !HaveVirtHostExt() && !HaveV82Debug()) || // Context matching
        ((dbgtype IN {'010x'}) && HaltOnBreakpointOrWatchpoint()) || // Address mismatch

Armv8 Pseudocode

J1.2 Pseudocode for AArch32 operation

(!dbgtype IN {'0x0x'}) && !context_aware) || // Context matching
(dbgtype IN {'1xxx'}) && !HaveEL(EL2)) then // EL2 extension
(c, dbgtype) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE, 4);
assert c IN {ConstraintDISABLED, ConstraintUNKNOWN};
if c == ConstraintDISABLED then return (FALSE,FALSE);
// Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
match_addr = (dbgtype IN {'0x0x'});
mismatch = (dbgtype IN {'010x'});
mismatch_vmid = (dbgtype IN {'10xx'});
mismatch_cid1 = (dbgtype IN {'xx1x'});
mismatch_cid2 = (dbgtype IN {'11xx'});
linked = (dbgtype IN {'xxx1'});

// If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a // VMID and/or context ID match, of if not context-aware. The above assertions mean that the // code can just test for match_addr == TRUE to confirm all these things.
if linked_to && (!linked || match_addr) then return (FALSE,FALSE);
// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linked && !match_addr then return (FALSE,FALSE);

// Do the comparison.
boolean BVR_match;
if match_addr then
  boolean byte_select_match;
  byte = UInt(vaddress<1:0>);
  assert byte IN {0,2}; // "vaddress" is halfword aligned
  byte_select_match = (DBGBCR[n].BAS<byte> == '1');
  integer top = 31;
  BVR_match = (vaddress<top:2> == DBGBVR[n]<top:2>) && byte_select_match;
elsif match_cid1 then
  BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBVR[n]<31:0>);
endif

boolean BXVR_match;
if match_vmid then
  bits(16) vmid;
  bits(16) bvr_vmid;
  if ELUsingAArch32(EL2) then
    vmid = ZeroExtend(VTTBR.VMID, 16);
    bvr_vmid = ZeroExtend(DBGBXVR[n]<7:0>, 16);
  elsif !Have16bitVMID() || VTCR_EL2.VS == '0' then
    vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
    bvr_vmid = ZeroExtend(DBGBXVR[n]<7:0>, 16);
  else
    vmid = VTTBR_EL2.VMID;
    bvr_vmid = DBGBXVR[n]<15:0>;
  endif
  BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                vmid == bvr_vmid);
elsif match_cid2 then
  BXVR_match = (PSTATE.EL != EL3 && (HaveVirtHostExt() || HaveV82Debug()) &&
                EL2Enabled() &&
                !ELUsingAArch32(EL2) &&
                DBGBXVR[n]<31:0> == CONTEXTIDR_EL2<31:0>);
endif

bvr_match_valid = (match_addr || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);

match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);
return (match && !mismatch, !match && mismatch);
aarch32/debug/breakpoint/AArch32.StateMatch

// AArch32.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) SSC_in, bit HMC_in,
    bits(2) PxC_in, boolean linked_in, bits(4) LBN,
    boolean isbreakpnt, boolean ispriv)

// "SSC_in","HMC_in","PxC_in" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
// "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
// "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
// "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.
bits(2) SSC = SSC_in;
bit HMC = HMC_in;
bits(2) PxC = PxC_in;
boolean linked = linked_in;

// If parameters are set to a reserved type, behaves as either disabled or a defined type
constraint c;
// SSCE value discarded as there is no SSCE bit in AArch32.
(c, SSC, -, HMC, PxC) = CheckValidStateMatch(SSC, '0', HMC, PxC, isbreakpnt);
if c == Constraint_DISABLED then return FALSE;
// Otherwise the HMC,SSC,PxC values are either valid or the values returned by
// CheckValidStateMatch are valid.
PL2_match = HaveEL(EL2) && ((HMC == '1' && (SSC:PxC != '1000')) || SSC == '11');
PL1_match = PxC<0> == '1';
PL0_match = PxC<1> == '1';
SSU_match = isbreakpnt && HMC == '0' && PxC == '00' && SSC != '11';

boolean priv_match;
if !ispriv && !isbreakpnt then
    priv_match = PL0_match;
elsif SSU_match then
    priv_match = PSTATE.M IN {M32_User,M32_Svc,M32_System};
else
    case PSTATE.EL of
        when EL3 priv_match = PL1_match; // EL3 and EL1 are both PL1
        when EL2 priv_match = PL2_match;
        when EL1 priv_match = PL1_match;
        when EL0 priv_match = PL0_match;
    end case

boolean security_state_match;
ss = CurrentSecurityState();
case SSC of
        when '00' security_state_match = TRUE; // Both
        when '01' security_state_match = ss == SS_NonSecure; // Non-secure only
        when '10' security_state_match = ss == SS_Secure; // Secure only
        when '11' security_state_match = (HMC == '1' || ss == SS_Secure); // HMC=1 -> Both, 0 -> Secure
    end case

integer lbn;
if linked then
    // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
    // it is CONstrained UNPredictable whether this gives no match, or LBN is mapped to some
    // UNKNOWN breakpoint that is context-aware.
    lbn = UInt(LBN);
    first_ctx_cmp = NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented();
    last_ctx_cmp = NumBreakpointsImplemented() - 1;
    if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
        (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp,
            Unpredictable_BPNOTCTXCMP);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
    else
        lbn = UInt(LBN);
    end case
end if
// Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

boolean linked_match;
if linked then
  vaddress = bits(32) UNKNOWN;
  linked_to = TRUE;
  (linked_match,-) = AArch32.BreakpointValueMatch(lbn, vaddress, linked_to);
return priv_match && security_state_match && (!linked || linked_match);

aarch32/debug/enables/AArch32.GenerateDebugExceptions

// AArch32.GenerateDebugExceptions()
// =================================

boolean AArch32.GenerateDebugExceptions()
ss = CurrentSecurityState();
return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, ss);

aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom

// AArch32.GenerateDebugExceptionsFrom()
// =====================================

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from_el, SecurityState from_state)
if !ELUsingAArch32(DebugTargetFrom(from_state)) then
  mask = '0'; // No PSTATE.D in AArch32 state
  return AArch64.GenerateDebugExceptionsFrom(from_el, from_state, mask);
if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
  return FALSE;
boolean enabled;
if HaveEL(EL3) && from_state == SS_Secure then
  assert from_el != EL2; // Secure EL2 always uses AArch64
  if IsSecureEL2Enabled() then
    // Implies that EL3 and EL2 both using AArch64
    enabled = MDCR_EL3.SDD == '0';
  else
    spd = if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32;
    if spd<3> == '1' then
      enabled = spd<0> == '1';
    else
      // SPD == 0b01 is reserved, but behaves the same as 0b00.
      enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
    if from_el == EL0 then enabled = enabled || SDER.SUIDEN == '1';
  else
    enabled = from_el != EL2;
return enabled;

aarch32/debug/pmu/AArch32.CheckForPMUOverflow

// AArch32.CheckForPMUOverflow()
// =============================
// Signal Performance Monitors overflow IRQ and CTI overflow events

AArch32.CheckForPMUOverflow()
if !ELUsingAArch32(EL1) then
  AArch64.CheckForPMUOverflow();
return;
bit hpme;
if HaveEL(EL2) then
    hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
    boolean pmuirq;
    bit E;
    pmuirq = PMCR.E == '1' && PMINTENSET.C == '1' && PMOVSSET.C == '1';
    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            E = if AArch32.PMUCounterIsHyp(idx) then hpme else PMCR.E;
            if E == '1' && PMINTENSET<idx> == '1' && PMOVSSET<idx> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR.)

aarch32/debug/pmu/AArch32.ClearEventCounters

    // AArch32.ClearEventCounters()
    // ===================================================
    // Zero all the event counters.
    AArch32.ClearEventCounters()
    if HaveAArch64() then
        // Force the counter to be cleared as a 64-bit counter.
        AArch64.ClearEventCounters();
        return;

    integer counters = AArch32.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMEVCNTR[idx] = Zeros(32);

aarch32/debug/pmu/AArch32.CountPMUEvents

    // AArch32.CountPMUEvents()
    // ========================
    // Return TRUE if counter "idx" should count its event.
    // For the cycle counter, idx == CYCLE_COUNTER_ID.
    boolean AArch32.CountPMUEvents(integer idx)
    constant integer num_counters = GetNumEventCounters();
    assert idx == CYCLE_COUNTER_ID || idx < num_counters;
    if !ELUsingAArch32(EL1) then return AArch64.CountPMUEvents(idx);
    boolean debug;
    boolean enabled;
    boolean prohibited;
    boolean filtered;
    boolean frozen;
    boolean resvd_for_el2;
    bit E;
    bit spme;
    bits(32) ovflws;
    // Event counting is disabled in Debug state
    debug = Halted();

    // Software can reserve some counters for EL2
    resvd_for_el2 = AArch32.PMUCounterIsHyp(idx);
    ss = CurrentSecurityState();

    // Main enable controls
    if idx == CYCLE_COUNTER_ID then
enabled = PMCR.E == '1' && PMCNTENSET.C == '1';
else
    if resvd_for_el2 then
        E = if ELUsingAArch32(EL2) then HDCR.HPME else MDCR_EL2.HPME;
    else
        E = PMCR.E;
    enabled = E == '1' && PMCNTENSET<idx> == '1';

// Event counting is allowed unless it is prohibited by any rule below
prohibited = FALSE;

// Event counting in Secure state is prohibited if all of:
// * EL3 is implemented
// * One of the following is true:
//   - EL3 is using AArch64, MDCR_EL3.SPME == 0, and either:
//     - FEAT_PMUv3p7 is not implemented
//     - MDCR_EL3.MPMX == 0
//   - EL3 is using AArch32 and SDCR.SPME == 0
// * Not executing at EL0, or SDER.SUNIDEN == 0
if HaveEL(EL3) & ss == SS_Secure then
    spme = if ELUsingAArch32(EL3) then SDCR.SPME else MDCR_EL3.SPME;
    if !ELUsingAArch32(EL3) && HavePMUv3p7() then
        prohibited = spme == '0' && MDCR_EL3.MPMX == '0';
    else
        prohibited = spme == '0';
    if prohibited && PSTATE.EL == EL0 then
        prohibited = SDER.SUNIDEN == '0';

// Event counting at EL2 is prohibited if all of:
// * The HPMD Extension is implemented
// * PMNx is not reserved for EL2
// * HDCR.HPMD == 1
if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
    prohibited = HDCR.HPMD == '1';

// The IMPLEMENTATION DEFINED authentication interface might override software
if prohibited && !HaveNoSecurePMUDisableOverride() then
    prohibited = !ExternalSecureNoninvasiveDebugEnabled();

// Event counting might be frozen
frozen = FALSE;

// If FEAT_PMUv3p7 is implemented, event counting can be frozen
if HavePMUv3p7() then
    bits(5) hpmn_bits;
    if HaveEL(EL2) then
        hpmn_bits = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
        hpmn = UInt(hpmn_bits);
        ovflws = ZeroExtend(PMOVSET<num_counters-1:0>, 32);
        bit FZ;
        if resvd_for_el2 then
            FZ = if ELUsingAArch32(EL2) then HDCR.HPMFZO else MDCR_EL2.HPMFZO;
            ovflws<hpmn-1:0> = Zeros(hpmn);
        else
            FZ = PMCR.FZO;
            if HaveEL(EL2) & hpmn < num_counters then
                ovflws<num_counters-1:hpmn> = Zeros(num_counters - hpmn);
                frozen = (FZ == '1') && !IsZero(ovflws);

// PMCR.DP disables the cycle counter when event counting is prohibited
if (prohibited || frozen) && idx == CYCLE_COUNTER_ID then
    enabled = enabled && (PMCR.DP == '0');
// Otherwise whether event counting is prohibited does not affect the cycle counter
prohibited = FALSE;
    frozen = FALSE;

// If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
// This is not overridden by PMCR.DP.
if HavePMUv3p5() && idx == CYCLE_COUNTER_ID then
  if HaveEL(EL3) && ss == SS_Secure then
    sccd = if ELUsingAArch32(EL3) then SDCR_SCCD else MDCR_EL3_SCCD;
    if sccd == '1' then prohibited = TRUE;
    if PSTATE.EL == EL2 && HDCR_HCCD == '1' then
      prohibited = TRUE;
  // Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
  filter = if idx == CYCLE_COUNTER_ID then PMCCFILTR else PMEVTYPER[idx];
  P = filter<31>;
  U = filter<30>;
  NSK = if HaveEL(EL3) then filter<29> else '0';
  NSU = if HaveEL(EL3) then filter<28> else '0';
  NSH = if HaveEL(EL2) then filter<27> else '0';
  ss = CurrentSecurityState();
  case PSTATE.EL of
    when EL0 filtered = if ss == SS_Secure then U == '1' else U != NSU;
    when EL1 filtered = if ss == SS_Secure then P == '1' else P != NSK;
    when EL2 filtered = NSH == '0';
    when EL3 filtered = P == '1';
  return !debug && enabled && !prohibited && !filtered && !frozen;

aarch32/debug/pmu/AArch32.GetNumEventCountersAccessible

// AArch32.GetNumEventCountersAccessible()
// =======================================
// Return the number of event counters that can be accessed at the current Exception level.
integer AArch32.GetNumEventCountersAccessible()
  integer n;
  integer total_counters = GetNumEventCounters();
  // Software can reserve some counters for EL2
  if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
    n = UInt(if !ELUsingAArch32(EL2) then MDCR_EL2_HPMN else HDCR_HPMN);
    if n > total_counters || (!HaveFeatHPMN0() && n == 0) then
      (-, n) = ConstrainUnpredictableInteger(0, total_counters, Unpredictable_PMUEVENTCOUNTER);
    else
      n = total_counters;
  return n;

aarch32/debug/pmu/AArch32.IncrementEventCounter

// AArch32.IncrementEventCounter()
// ===============================
// Increment the specified event counter by the specified amount.
AArch32.IncrementEventCounter(integer idx, integer increment)
  if HaveAArch64() then
    // Force the counter to be incremented as a 64-bit counter.
    AArch64.IncrementEventCounter(idx, increment);
    return;
  // In this model, event counters in an AArch32-only implementation are 32 bits and
  // the LP bits are RES0 in this model, even if FEAT_PMUv3p5 is implemented.
  integer old_value;
  integer new_value;
  integer ovflw;
  bit lp;
  old_value = UInt(PMEVCNTR[idx]);
new_value = old_value + PMUCountValue(idx, increment);

PMEVERCNR[idx] = new_value<31:0>;

if old_value<64:ovflw> != new_value<64:ovflw> then
  PMOVSSET<idx> = '1';
  PMOVSR<idx> = '1';
  // Check for the CHAIN event from an even counter
  if idx<0> == '0' && idx + 1 < GetNumEventCounters() then
    PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);

aarch32/debug/pmu/AArch32.PMUCounterIsHyp

// AArch32.PMUCounterIsHyp
// ===============
// Returns TRUE if a counter is reserved for use by EL2, FALSE otherwise.

boolean AArch32.PMUCounterIsHyp(integer n)
boolean resvd_for_el2;

// Software can reserve some event counters for EL2
if n != CYCLE_COUNTER_ID && HaveEL(EL2) then
  bits(5) hpmn_bits = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
  resvd_for_el2 = n >= UInt(hpmn_bits);
  if UInt(hpmn_bits) > GetNumEventCounters() || (!HaveFeatHPMN0() && IsZero(hpmn_bits)) then
    resvd_for_el2 = boolean UNKNOWN;
  else
    resvd_for_el2 = FALSE;

return resvd_for_el2;

aarch32/debug/pmu/AArch32.PMUCycle

// AArch32.PMUCycle()
// ===============
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

AArch32.PMUCycle()
if !HavePMUv3() then
  return;

PMUEvent(PMU_EVENT_CPU_CYCLES);

integer counters = GetNumEventCounters();
if counters != 0 then
  for idx = 0 to counters - 1
    if AArch32.CountPMUEvents(idx) then
      accumulated = PMUEventAccumulator[idx];
      AArch32.IncrementEventCounter(idx, accumulated);
      PMUEventAccumulator[idx] = 0;

integer old_value;
integer new_value;
integer ovflw;
if (AArch32.CountPMUEvents(CYCLE_COUNTER_ID) &&
(PMCR.LC == '1' || PMCR.D == '0' || HasElapsed64Cycles()) then
  old_value = UInt(PMCCNTR);
  new_value = old_value + 1;
  PMCCNTR = new_value<63:0>;
  ovflw = if PMCR.LC == '1' then 64 else 32;
if old_value<64:ovflw> != new_value<64:ovflw> then
    PMOVSET.C = '1';
    PMOVSR.C = '1';
    AArch32.CheckForPMUOverflow();

aarch32/debug/pmu/AArch32.PMUSwIncrement

    // AArch32.PMUSwIncrement()
    // ========================
    // Generate PMU Events on a write to PMSWINC.

    AArch32.PMUSwIncrement(bits(32) sw_incr)
    integer counters = AArch32.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            if sw_incr<idx> == '1' then
                PMUEvent(PMU_EVENT_SW_INCR, 1, idx);

aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

    // AArch32.EnterHypModeInDebugState()
    // ==================================
    // Take an exception in Debug state to Hyp mode.

    AArch32.EnterHypModeInDebugState(ExceptionRecord exception)
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = bits(32) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
    PMUEvent(PMU_EVENT_SW_INCR, 1, idx);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                   // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = HSCTRL.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();
    EndOfInstruction();

aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState

    // AArch32.EnterModeInDebugState()
    // ===============================
    // Take an exception in Debug state to a mode other than Monitor and Hyp mode.

    AArch32.EnterModeInDebugState(bits(5) target_mode)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
// PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
PSTATE.T = '1';                          // PSTATE.J is RES0
PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
DLR = bits(32) UNKNOWN;
DSPSR = bits(32) UNKNOWN;
PSTATE.E = SCTLR.EE;
PSTATE.IL = '0';
PSTATE.IT = '00000000';
if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
EDSCR.ERR = '1';
UpdateEDSCRFields();                      // Update ESCR processor state flags.
EndOfInstruction();

aarch32/debug/takeexceptiondbg/AArch32.EnterMonitorModeInDebugState

// AArch32.EnterMonitorModeInDebugState()
// =====================================
// Take an exception in Debug state to Monitor mode.
AArch32.EnterMonitorModeInDebugState()
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityState() == SS_Secure;
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                          // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
        if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
        DLR = bits(32) UNKNOWN;
        DSPSR = bits(32) UNKNOWN;
        ESCR.ERR = '1';
        UpdateEDSCRFields();                      // Update ESCR processor state flags.
        EndOfInstruction();

aarch32/debug/watchpoint/AArch32.WatchpointByteMatch

// AArch32.WatchpointByteMatch()
// ===========================
boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)
    integer top = 31;
    bottom = if DBGWVR[n]<2> == '1' then 2 else 3;            // Word or doubleword
    byte_select_match = (DBGWCR[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR[n].MASK);
    // If DBGWCR[n].MASK is non-zero value and DBGWCR[n].BAS is not set to '11111111', or
    // DBGWCR[n].BAS specifies a non- contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
if mask > 0 && !IsOnes(DBGWCR[n].BAS) then
  byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMSKANDBAS);
else
  LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1));
  MSB = (DBGWCR[n].BAS + LSB);
  if IsZero(MSB AND (MSB - 1)) then // Not contiguous
    byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASECONTIGUOUS);
    bottom = 3; // For the whole doubleword
  else
    LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1));
    MSB = (DBGWCR[n].BAS + LSB);
    if !IsZero(MSB AND (MSB - 1)) then // Not contiguous
      byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASECONTIGUOUS);
      bottom = 3; // For the whole doubleword
  end if
end if

// If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
if mask > 0 && mask <= 2 then
  Constraint c;
  (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
  assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
  case c of
    when Constraint_DISABLED  return FALSE;            // Disabled
    when Constraint_NONE      mask = 0;                // No masking
    // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value
    when Constraint_UNKNOWN
  end case
  return FALSE;
end if

boolean WVR_match;
if mask > bottom then
  // If the DBGxVR<n>_EL1.RES field bits are not a sign extension of the MSB
  // of DBGBVR<n>_EL1.VA, it is UNPREDICTABLE whether they appear to be
  // included in the match.
  if !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) then
    if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
      top = 63;
      WVR_match = (vaddress<top:mask> == DBGWVR[n]<top:mask>);
      // If masked bits of DBGBVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
      if WVR_match && !IsZero(DBGWVR[n]<mask-1:bottom>) then
        WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMSKEDBITS);
      else
        WVR_match = vaddress<top:bottom> == DBGWVR[n]<top:bottom>;
      end if
    end if
  end if
end if

return WVR_match && byte_select_match;

aarch32/debug/watchpoint/AArch32.WatchpointMatch

// AArch32.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size, boolean ispriv,
                                      AccType acctype, boolean iswrite)
  assert ELUsingAArch32(S1TranslationRegime());
  assert n < NumWatchpointsImplemented();

  // "ispriv" is:
  //  * FALSE for all loads, stores, and atomic operations executed at EL0.
  //  * FALSE if the access is unprivileged.
  //  * TRUE for all other loads, stores, and atomic operations.

  enabled = DBGWCR[n].E == '1';
  linked = DBGWCR[n].WT == '1';
  isbreakpnt = FALSE;

  state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
                                     linked, DBGWCR[n].LBN, isbreakpnt, ispriv);
  ls_match = FALSE;
  if iswrite then
    ls_match = (DBGWCR[n].LSC<1> != '0');
  else
    ls_match = (DBGWCR[n].LSC<0> != '0');
  end if

  value_match = FALSE;
  for byte = 0 to size - 1
    value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);
  end for

  return value_match && state_match && ls_match;
return value_match && state_match && ls_match && enabled;

### J1.2.2 aarch32/exceptions

This section includes the following pseudocode functions:

- `aarch32/exceptions/async/AArch32.TakeVirtualFIQException` on page J1-11276.
- `aarch32/exceptions/debug/DebugException` on page J1-11277.
- `aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64` on page J1-11278.
- `aarch32/exceptions/exceptions/ExcVectorBase` on page J1-11280.
- `aarch32/exceptions/ieeefp/AArch32.FPTrappedException` on page J1-11280.
- `aarch32/exceptions/takeexception/AArch32.EnterMode` on page J1-11283.
- `aarch32/exceptions/traps/AArch32.CheckForSMCUndefOrTrap` on page J1-11285.
- `aarch32/exceptions/traps/AArch32.CheckForSVCTrap` on page J1-11285.
- `aarch32/exceptions/traps/AArch32.CheckForWFXTrap` on page J1-11286.
• aarch32/exceptions/traps/AArch32.UndefinedFault on page J1-11290.

aarch32/exceptions/aborts/AArch32.Abort

// AArch32.Abort()
// ===============
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
    route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
        (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
        (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
    route_to_aarch64 = SCR_EL3.EA == '1' && IsExternalAbort(fault);

if route_to_aarch64 then
    AArch64.Abort(ZeroExtend(vaddress, 64), fault);
elsif fault.acctype == AccType_IFETCH then
    AArch32.TakePrefetchAbortException(vaddress, fault);
else
    AArch32.TakeDataAbortException(vaddress, fault);

aarch32/exceptions/aborts/AArch32.AbortSyndrome

// AArch32.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort exceptions taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(32) vaddress)

exception = ExceptionSyndrome(exceptype);

d_side = exceptype == Exception_DataAbort;

exception.syndrome = AArch32.FaultSyndrome(d_side, fault);
exception.vaddress = ZeroExtend(vaddress, 64);
if IPAValid(fault) then
    exception.ipavalid = TRUE;
else
    exception.ipavalid = FALSE;

return exception;

aarch32/exceptions/aborts/AArch32.CheckPCAlignment

// AArch32.CheckPCAlignment()
// ==========================

AArch32.CheckPCAlignment()

bits(32) pc = ThisInstrAddr(32);
if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1' || pc<0> == '1') then
    if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();

    // Generate an Alignment fault Prefetch Abort exception
vaddress = pc;
acctype = AccType_IFETCH;
iswrite = FALSE;
secondstage = FALSE;
AArch32.Abort(vaddress, AlignmentFault(acctype, iswrite, secondstage));

**aarch32/exceptions/aborts/AArch32.ReportDataAbort**

```c
// AArch32.ReportDataAbort()
// =========================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
   long_format = FALSE;
   if route_to_monitor && CurrentSecurityState() != SS_Secure then
      long_format = ((TTBCR_S.EAE == '1') ||
                     (IsExternalSyncAbort(fault) && ((PSTATE.EL == EL2 || TTBCR.EAE == '1')) ||
                      (fault.secondstage && boolean IMPLEMENTATION_DEFINED "Stage 2 synchronous
                       External abort reports using long-descriptor format when TTBCR_S.EAE is 0b0"));
   else
      long_format = TTBCR.EAE == '1';
d_side = TRUE;
bits(32) syndrome;
if long_format then
   syndrome = AArch32.FaultStatusLD(d_side, fault);
else
   syndrome = AArch32.FaultStatusSD(d_side, fault);
if fault.acctype == AccType_IC then
   bits(32) i_syndrome;
   if (!long_format &&
       boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
      i_syndrome = syndrome;
      syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
   else
      i_syndrome = bits(32) UNKNOWN;
      if route_to_monitor then
         IFSR_S = i_syndrome;
      else
         IFSR = i_syndrome;
      if route_to_monitor then
         DFSR_S = syndrome;
         DFAR_S = vaddress;
      else
         DFSR = syndrome;
         DFAR = vaddress;
   return;
```

**aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort**

```c
// AArch32.ReportPrefetchAbort()
// =============================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
   // The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
   // Normally, the current translation table format determines the format. For an abort from
   // Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
   // following applies:
   // * The Secure TTBCR.EAE is set to 1.
   // * It is taken from Hyp mode.
   // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
```
```plaintext
long_format = FALSE;
if route_to_monitor && CurrentSecurityState() != SS_Secure then
    long_format = TTBCR.S.EAE == '1' || PSTATE.EL == EL2 || TTBCR.EAE == '1';
else
    long_format = TTBCR.EAE == '1';

d_side = FALSE;
bits(32) fsr;
if long_format then
    fsr = AArch32.FaultStatusLD(d_side, fault);
else
    fsr = AArch32.FaultStatusSD(d_side, fault);

if route_to_monitor then
    IFSR_S = fsr;
    IFAR_S = vaddress;
else
    IFSR = fsr;
    IFAR = vaddress;
return;
```

```plaintext
aarch32/exceptions/aborts/AArch32.TakeDataAbortException

// AArch32.TakeDataAbortException()
// ---------------------------------
AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)
route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);  
route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&  
(HCR.TGE == '1' ||  
(HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||  
(IsDebugException(fault) && HDCR.TDE == '1') ||  
IsSecondStage(fault));

bits(32) preferred_exception_return = ThisInstrAddr(32);
vect_offset = 0x10;
lr_offset = 8;
if IsDebugException(fault) then DBGDCR.MOE = fault.debugmoe;
if route_to_monitor then  
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
elsif PSTATE.EL == EL2 || route_to_hyp then  
    exception = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress);
    if PSTATE.EL == EL2 then  
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else  
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else  
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

```plaintext
aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

// AArch32.TakePrefetchAbortException()
// -----------------------------------
AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)
route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&  
(HCR.TGE == '1' ||  
(HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||  
(IsDebugException(fault) && HDCR.TDE == '1') ||  
IsSecondStage(fault));
```
IsSecondStage(fault));

ExceptionRecord exception;
bits(32) preferred_exception_return = ThisInstrAddr(32);
vect_offset = 0x0C;

lr_offset = 4;

if IsDebugException(fault) then DBGDSCRExt.MOE = fault.debugmoe;
if route_to_monitor then
    AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
elsif PSTATE.EL == EL2 || route_to_hyp then
    if fault.statuscode == Fault_Alignment then // PC Alignment fault
        exception = ExceptionSyndrome(Exception_PCAlignment);
    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
    else
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
        aarch32/exceptions/async/AArch32.TakePhysicalFIQException
        // AArch32.TakePhysicalFIQException()
        // ==================================
        AArch32.TakePhysicalFIQException()

        // Check if routed to AArch64 state
        route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
        if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
            route_to_aarch64 = SCR_EL3.FIQ == '1';

        if route_to_aarch64 then AArch64.TakePhysicalFIQException();
        route_to_monitor = HaveEL(EL3) && SCR.FIQ == '1';
        route_to_hyp = (PSTATE.EL IN {EL0, EL1}) && SCR.FIQ == '1';
        bits(32) preferred_exception_return = ThisInstrAddr(32);
        lr_offset = 4;
        if route_to_monitor then
            AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
        elsif PSTATE.EL == EL2 || route_to_hyp then
            exception = ExceptionSyndrome(Exception_FIQ);
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
        else
            AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
        endif
        aarch32/exceptions/async/AArch32.TakePhysicalIRQException
        // AArch32.TakePhysicalIRQException()
        // ==================================
        // Take an enabled physical IRQ exception.
        AArch32.TakePhysicalIRQException()
// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
    route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.IMO == '1' && !IsInHost());
if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
    route_to_aarch64 = SCR_EL3.IRQ == '1';
if route_to_aarch64 then AArch64.TakePhysicalIRQException();

route_to_monitor = HaveEL(EL3) && SCR.IRQ == '1';
route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
    (HCR.TGE == '1' || HCR.IMO == '1'));
bits(32) preferred_exception_return = ThisInstrAddr(32);
lr_offset = 4;
if route_to_monitor then
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
elsif PSTATE.EL == EL2 || route_to_hyp then
    exception = ExceptionSyndrome(Exception_IRQ);
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
else
    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);

aarch32/exceptions/async/AArch32.TakePhysicalSErrorException

// AArch32.TakePhysicalSErrorException()
// ---------------------------------------
AArch32.TakePhysicalSErrorException(boolean parity, bit extflag, bits(2) pe_error_state,
    bits(25) full_syndrome)
// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
    route_to_aarch64 = (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1'));
if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
    route_to_aarch64 = SCR_EL3.EA == '1';
if route_to_aarch64 then
    AArch64.TakePhysicalSErrorException(full_syndrome);

route_to_monitor = HaveEL(EL3) && SCR.EA == '1';
route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
    (HCR.TGE == '1' || HCR.AMO == '1'));
bits(32) preferred_exception_return = ThisInstrAddr(32);
vect_offset = 0x10;
lr_offset = 8;
bits(2) target_el;
if route_to_monitor then
    target_el = EL3;
elsif PSTATE.EL == EL2 || route_to_hyp then
    target_el = EL2;
else
    target_el = EL1;
if IsSErrorEdgeTriggered(target_el, full_syndrome) then
    ClearPendingPhysicalSError();
fault = AsyncExternalAbort(parity, pe_error_state, extflag);
vaddress = bits(32) UNKNOWN;
case target_el of
    when EL3
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
when EL2
    exception = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress);
    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
when EL1
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
otherwise
    Unreachable();

aarch32/exceptions/async/AArch32.TakeVirtualFIQException

// AArch32.TakeVirtualFIQException()
// =================================
AArch32.TakeVirtualFIQException()
assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
if ELUsingAArch32(EL2) then   // Virtual IRQ enabled if TGE==0 and FMO==1
    assert HCR.TGE == '0' && HCR.FMO == '1';
else
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();
bits(32) preferred_exception_return = ThisInstrAddr(32);
vect_offset = 0x1C;
lr_offset = 4;
AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);

aarch32/exceptions/async/AArch32.TakeVirtualIRQException

// AArch32.TakeVirtualIRQException()
// =================================
AArch32.TakeVirtualIRQException()
assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
if ELUsingAArch32(EL2) then   // Virtual IRQs enabled if TGE==0 and IMO==1
    assert HCR.TGE == '0' && HCR.IMO == '1';
else
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualIRQException();
bits(32) preferred_exception_return = ThisInstrAddr(32);
vect_offset = 0x18;
lr_offset = 4;
AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);

aarch32/exceptions/async/AArch32.TakeVirtualSErrorException

// AArch32.TakeVirtualSErrorException()
// =====================================
AArch32.TakeVirtualSErrorException(bit extflag, bits(2) pe_error_state, bits(25) full_syndrome)
assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
if ELUsingAArch32(EL2) then   // Virtual SError enabled if TGE==0 and AMO==1
    assert HCR.TGE == '0' && HCR.AMO == '1';
assert HCR.TGE == '0' && HCR.AMO == '1';
else
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';

// Check if routed to AArch64 state
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
    AArch64.TakeVirtualSErrorException(full_syndrome);

route_to_monitor = FALSE;

bits(32) preferred_exception_return = ThisInstrAddr(32);
 vect_offset = 0x10;
 lr_offset = 8;

vaddress = bits(32) UNKNOWN;
parity = FALSE;
FaultRecord fault;
if HaveRASExt() then
    if ELUsingAArch32(EL2) then
        fault = AsyncExternalAbort(FALSE, VDFSR.AET, VDFSR.ExT);
    else
        fault = AsyncExternalAbort(FALSE, VSESR_EL2.AET, VSESR_EL2.ExT);
    else
        fault = AsyncExternalAbort(parity, pe_error_state, extflag);

ClearPendingVirtualSError();
AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);

aarch32/exceptions/debug/AArch32.SoftwareBreakpoint

// AArch32.SoftwareBreakpoint()
// -------------------------------------------

AArch32.SoftwareBreakpoint(bits(16) immediate)

if (EL2Enabled() && !ELUsingAArch32(EL2) &&
    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAArch32(EL1) then
    AArch64.SoftwareBreakpoint(immediate);
vaddress = bits(32) UNKNOWN;
acctype = AccType_IFETCH;           // Take as a Prefetch Abort
iswrite = FALSE;
entry = DebugException_BKPT;
fault = AArch32.DebugFault(acctype, iswrite, entry);
AArch32.Abort(vaddress, fault);

aarch32/exceptions/debug/DebugException

constant bits(4) DebugException_Breakpoint  = '0001';
constant bits(4) DebugException_BKPT        = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint  = '1010';

aarch32/exceptions/exceptions/AArch32.CheckAdvSIMDOrFPRegisterTraps

// AArch32.CheckAdvSIMDOrFPRegisterTraps()
// ----------------------------------------

// Check if an instruction that accesses an Advanced SIMD and
// Floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.

AArch32.CheckAdvSIMDOrFPRegisterTraps(bits(4) reg)

if PSTATE.EL == EL1 && EL2Enabled() then
tid0 = if ELUsingAArch32(EL2) then HCR.TID0 else HCR_EL2.TID0;
tid3 = if ELUsingAArch32(EL2) then HCR.TID3 else HCR_EL2.TID3;

if (tid0 == '1' && reg == '0000')                             // FPSID
  || (tid3 == '1' && reg IN {'0101', '0110', '0111'}) then    // MVFRx
    if ELUsingAArch32(EL2) then
      AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap

aarch32/exceptions/exceptions/AArch32.ExceptionClass

// AArch32.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in HSR

(integer,bit) AArch32.ExceptionClass(Exception exceptype)

  il_is_valid = TRUE;
  integer ec;
  case exceptype of
    when Exception_Uncategorized        ec = 0x00; il_is_valid = FALSE;
    when Exception_WFxTrap              ec = 0x01;
    when Exception_CPI15RTTrap          ec = 0x03;
    when Exception_CPI15RRTTrap         ec = 0x04;
    when Exception_CPI14RTTrap          ec = 0x05;
    when Exception_CPI14DTR Trap        ec = 0x06;
    when Exception_A dvSIMDPAccessTrap  ec = 0x07;
    when Exception_FPIDTrap             ec = 0x08;
    when Exception_PACTrap              ec = 0x09;
    when Exception_TSTARTAccessTrap     ec = 0x1B;
    when Exception_GPC                  ec = 0x1E;
    when Exception_CPI14RRTTrap         ec = 0x0C;
    when Exception_BranchTarget         ec = 0x0D;
    when Exception_IllegalState         ec = 0x0E; il_is_valid = FALSE;
    when Exception_SupervisorCall       ec = 0x11;
    when Exception_HypervisorCall       ec = 0x12;
    when Exception_MonitorCall          ec = 0x13;
    when Exception_InstructionAbort     ec = 0x20; il_is_valid = FALSE;
    when Exception_PCAlignment          ec = 0x22; il_is_valid = FALSE;
    when Exception_DataAbort            ec = 0x24;
    when Exception_NV2DataAbort         ec = 0x25;
    when Exception_FPTrappedException   ec = 0x28;
    otherwise                           Unreachable();
  end;

  if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
    ec = ec + 1;
  bit il;
  if il_is_valid then
    il = if ThisInstrLength() == 32 then '1' else '0';
  else
    il = '1';
  return (ec,il);

aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64

// AArch32.GeneralExceptionsToAArch64()
// ===============
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64()
aarch32/exceptions/exceptions/AArch32.ReportHypEntry

// AArch32.ReportHypEntry()
// ========================
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord exception)

    Exception exceptype = exception.exceptype;
    (ec,il) = AArch32.ExceptionClass(exceptype);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';
    HSR = ec<5:0>:il:iss;
    if exceptype IN {Exception_InstructionAbort, Exception_PCAlignment} then
        HIFAR = exception.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elsif exceptype == Exception_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = exception.vaddress<31:0>;
    if exception.ipavalid then
        HPFAR<31:4> = exception.ipaddress<39:12>;
    else
        HPFAR<31:4> = bits(28) UNKNOWN;
    return;

aarch32/exceptions/exceptions/AArch32.ResetControlRegisters

// Resets System registers and memory-mapped control registers that have architecturally-defined reset values to those values.
AArch32.ResetControlRegisters(boolean cold_reset);

aarch32/exceptions/exceptions/AArch32.TakeReset

// AArch32.TakeReset()
// ===================
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)

    assert !HaveAArch64();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32.WriteMode(M32_Svc);
        SCR.NS = '0';                     // Secure state
    elsif HaveEL(EL2) then
        AArch32.WriteMode(M32_Hyp);
    else
        AArch32.WriteMode(M32_Svc);

    // Reset System registers in the coproc=0b111x encoding space and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';
// Reset all other PSTATE fields, including instruction set and endianness according to the
// SCTLR values produced by the above call to ResetControlRegisters()
PSTATE.<A,I,F> = '111'; // All asynchronous exceptions masked
PSTATE.IT = '00000000'; // IT block state reset
if HaveEL(EL2) & !HaveEL(EL3) then
  PSTATE.T = HSCTLR.TE; // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
PSTATE.E = HSCTLR.EE; // Endianness: EE=0: little-endian, EE=1: big-endian
else
  PSTATE.T = SCTLR.TE; // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
PSTATE.E = SCTLR.EE; // Endianness: EE=0: little-endian, EE=1: big-endian
PSTATE.IL = '0'; // Clear Illegal Execution state bit

// All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
// below are UNKNOWN bitstrings after reset. In particular, the return information registers
// R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
// is impossible to return from a reset in an architecturally defined way.
AArch32.ResetGeneralRegisters();
AArch32.ResetSIMDFPRegisters();
AArch32.ResetSpecialRegisters();
ResetExternalDebugRegisters(cold_reset);

bits(32) rv; // IMPLEMENTATION DEFINED reset vector
if HaveEL(EL3) then
  if MVBAR<0> == '1' then // Reset vector in MVBAR
    rv = MVBAR<31:1>::'0';
  else
    rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
else
  rv = RVBAR<31:1>::'0';

// The reset vector must be correctly aligned
assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');

boolean branch_conditional = FALSE;
BranchTo(rv, BranchType_RESET, branch_conditional);

aarch32/exceptions/exceptions/ExcVectorBase

// ExcVectorBase()
// ===============

bits(32) ExcVectorBase()
  if SCTLR.V == '1' then // Hivecs selected, base = 0xFFFF0000
    return Ones(16):Zeros(16);
  else
    return VBAR<31:5>:Zeros(5);

aarch32/exceptions/ieeefp/AArch32.FPTrappedException

// AArch32.FPTrappedException()
// ============================

AArch32.FPTrappedException(bits(8) accumulated_exceptions)
if AArch32.GeneralExceptionsToAArch64() then
  is_ase = FALSE;
  element = 0;
  AArch64.FPTrappedException(is_ase, accumulated_exceptions);
FPEXC.DEX = '1';
FPEXC.TFV = '1';
FPEXC<7:4:0> = accumulated_exceptions<7:4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
FPEXC<10:8> = '111'; // VECITR is RES1
AArch32.TakeUndefInstrException();

aarch32/exceptions/syscalls/AArch32.CallHypervisor

// AArch32.CallHypervisor()
// ========================
// Performs a HVC call
AArch32.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);
    if !ELUsingAArch32(EL2) then
        AArch64.CallHypervisor(immediate);
    else
        AArch32.TakeHVCException(immediate);

aarch32/exceptions/syscalls/AArch32.CallSupervisor

// AArch32.CallSupervisor()
// ========================
// Calls the Supervisor
AArch32.CallSupervisor(bits(16) immediate_in)
    bits(16) immediate = immediate_in;
    if AArch32.CurrentCond() != '1110' then
        immediate = bits(16) UNKNOWN;
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CallSupervisor(immediate);
    else
        AArch32.TakeSVCException(immediate);

aarch32/exceptions/syscalls/AArch32.TakeHVCException

// AArch32.TakeHVCException()
// =========================
AArch32.TakeHVCException(bits(16) immediate)
    assert HaveEL(EL2) && ELUsingAArch32(EL2);
    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;
    exception = ExceptionSyndrome(Exception_HypervisorCall);
    exception.syndrome<15:0> = immediate;
    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);

aarch32/exceptions/syscalls/AArch32.TakeSMCEException

// AArch32.TakeSMCException()
// ==========================
AArch32.TakeSMCException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    AArch32.ITAdvance();
Armv8 Pseudocode

J1.2 Pseudocode for AArch32 operation

SSAdvance();
bits(32) preferred_exception_return = NextInstrAddr(32);
vect_offset = 0x08;
lr_offset = 0;

AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);

aarch32/exceptions/syscalls/AArch32.TakeSVCException

// AArch32.TakeSVCException()
// ---------------------------------------
AArch32.TakeSVCException(bits(16) immediate)

AArch32.ITAdvance();
SSAdvance();
route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

bits(32) preferred_exception_return = NextInstrAddr(32);
vect_offset = 0x08;
lr_offset = 0;

if PSTATE.EL == EL2 || route_to_hyp then
  exception = ExceptionSyndrome(Exception_SupervisorCall);
  exception.syndrome<15:0> = immediate;
  if PSTATE.EL == EL2 then
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
  else
    AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
else
  AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);

aarch32/exceptions/takeexception/AArch32.EnterHypMode

// AArch32.EnterHypMode()
// ---------------------------------------
// Take an exception to Hyp mode.
AArch32.EnterHypMode(ExceptionRecord exception, bits(32) preferred_exception_return,
integer vect_offset)

SynchronizeContext();
assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
  AArch32.ReportHypEntry(exception);
AArch32.WriteMode(M32_Hyp);
SPSR[] = spsr;
ELR_hyp = preferred_exception_return;
PSTATE.T = HSCTRL.TE;                        // PSTATE.J is RES0
PSTATE.SS = '0';
if HaveEL(EL3) || SCR_GEN[].EA == '0' then PSTATE.A = '1';
if HaveEL(EL3) || SCR_GEN[].IRQ == '0' then PSTATE.I = '1';
if HaveEL(EL3) || SCR_GEN[].FIQ == '0' then PSTATE.F = '1';
PSTATE.E = HSCTRL.EE;
PSTATE.IL = '0';
PSTATE.IT = '00000000';
if HaveSSBSExt() then PSTATE.SSBS = HSCTRL.DSSBS;
boolean branch_condition = FALSE;
BranchTo(HVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_condition);

CheckExceptionCatch(TRUE);                     // Check for debug event on exception entry
aarch32/exceptions/takeexception/AArch32.EnterMode

// AArch32.EnterMode()
// ====================
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset, integer vect_offset)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32_FIQ then
        PSTATE.<A,I,F> = '111';
    elsif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
        PSTATE.E = SCTLR.EE;
        PSTATE.IL = '0';
        PSTATE.IT = '00000000';
        if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
        if HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
        boolean branch_conditional = FALSE;
        BranchTo(ExcVectorBase()<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);
        CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry
    EndOfInstruction();

aarch32/exceptions/takeexception/AArch32.EnterMonitorMode

// AArch32.EnterMonitorMode()
// ==========================
// Take an exception to Monitor mode.

AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset, integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityState() == SS_Secure;
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.<A,I,F> = '111';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && !from_secure then
        PSTATE.PAN = '0';
elsif SCTLR.SPAN == '0' then
    PSTATE.PAN = '1';
if HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
boolean branch_conditional = FALSE;
BranchTo(MVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE);                      // Check for debug event on exception entry

EndOfInstruction();

aarch32/exceptions/traps/AArch32.CheckAdvSIMDOrFPEnabled

// AArch32.CheckAdvSIMDOrFPEnabled()
// ==================================
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

AArch32.CheckAdvSIMDOrFPEnabled(boolean fpexc_check, boolean advsimd)
if PSTATE.EL == EL0 && (EL2Enabled() || (ELUsingAArch32(EL2) && HCR_EL2.TGE == '0')) &&
    !ELUsingAArch32(EL1) then
    // The PE behaves as if FPEXC.EN is 1
    AArch64.CheckFPEnabled();
    AArch64.CheckFPAdvSIMDEnabled();
else if PSTATE.EL == EL0 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' &&
    !ELUsingAArch32(EL1) then
    if fpexc_check && HCR_EL2.RW == '0' then
        fpexc_en = bits(1) IMPLEMENTATION_DEFINED “FPEXC.EN value when TGE==1 and RW==0”;
        if fpexc_en == '0' then UNDEFINED;
    else
        cpacr_asedis = CPACR.ASEDIS;
        cpacr_cp10 = CPACR.cp10;
        if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState() == SS_NonSecure then
            // Check if access disabled in NSACR
            if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
            if NSACR.cp10 == '0' then cpacr_cp10 = '0';
        if PSTATE.EL != EL2 then
            // Check if Advanced SIMD disabled in CPACR
            if advsimd && cpacr_asedis == '1' then UNDEFINED;
        boolean disabled;
        case cpacr_cp10 of
        when '00' disabled = TRUE;
        when '01' disabled = PSTATE.EL == EL0;
        when '10' disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
        when '11' disabled = FALSE;
    if enabled then UNDEFINED;
    if fpexc_check && FPEXC.EN == '0' then UNDEFINED;
    AArch32.CheckFPAdvSIMDTrap(advsimd); // Also check against HCPTR and CPTR_EL3

aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

// AArch32.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
if EL2Enabled() && !ELUsingAArch32(EL2) then
    AArch64.CheckFPAdvSIMDTrap();
else if HaveEL(EL3) && !ELUsingAArch32(EL3) &&
    CPTR_EL3.TFP == '1' && EL3SDDTTrapPriority()) then
    UNDEFINED;

ss = CurrentSecurityState();
if HaveEL(EL2) && ss != SS_Secure then
    hcptr_tase = HCPTR.TASE;
    hcptr_cp10 = HCPTR.TCP10;
if HaveEL(EL3) && ELUsingAArch32(EL3) then
    // Check if access disabled in NSACR
    if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
    if NSACR.cp10 == '0' then hcptr_cp10 = '1';

    // Check if access disabled in HCPTR
    if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
        exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        exception.syndrome<24:20> = ConditionSyndrome();
        if advsimd then
            exception.syndrome<5> = '1';
        else
            exception.syndrome<5> = '0';
            exception.syndrome<3:0> = '1010'; // coproc field, always 0xA
        if PSTATE.EL == EL2 then
            AArch32.TakeUndfInstrException(exception);
        else
            AArch32.TakeHypTrapException(exception);
if HaveEL(EL3) && !ELUsingAArch32(EL3) then
    // Check if access disabled in CPTR_EL3
    if CPTR_EL3.TFP == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AdvSIMDFPAccessTrap(EL3);

aarch32/exceptions/traps/AArch32.CheckForSMCUndefOrTrap

// AArch32.CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction
AArch32.CheckForSMCUndefOrTrap()
if !HaveEL(EL3) || PSTATE.EL == EL0 then
    UNDEFINED;
if EL2Enabled() && ELUsingAArch32(EL2) then
    AArch64.CheckForSMCUndefOrTrap(Zeros(16));
else
    route_to_hyp = EL2Enabled() && PSTATE.EL == EL1 && HCR.TSC == '1';
    if route_to_hyp then
        exception = ExceptionSyndrome(Exception_MonitorCall);
        AArch32.TakeHypTrapException(exception);

aarch32/exceptions/traps/AArch32.CheckForSVCTrap

// AArch32.CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction
AArch32.CheckForSVCTrap(bits(16) immediate)
if HaveFGTExt() then
  route_to_el2 = FALSE;
if PSTATE.EL == EL0 then
  route_to_el2 = ((ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
                  (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));
if route_to_el2 then
  exception = ExceptionSyndrome(Exception_SupervisorCall);
  exception.syndrome<15:0> = immediate;
  exception.trappedsyscallinst = TRUE;
  bits(64) preferred_exception_return = ThisInstrAddr(64);
  vect_offset = 0x0;
  AArch64.TakeExceptionEL2(exception, preferred_exception_return, vect_offset);

aarch32/exceptions/traps/AArch32.CheckForWFxTrap

// AArch32.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction
AArch32.CheckForWFxTrap(bits(2) target_el, WFxType wfxtype)
  assert HaveEL(target_el);
  
  // Check for routing to AArch64
  if !ELUsingAArch32(target_el) then
    AArch64.CheckForWFxTrap(target_el, wfxtype);
    return;
  
  boolean is_wfe = wfxtype == WFxType_WFE;
  boolean trap;
  case target_el of
    when EL1
      trap = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
    when EL2
      trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
    when EL3
      trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';
  if trap then
    if target_el == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.WFxTrap(wfxtype, target_el);
    if target_el == EL3 then
      AArch32.TakeMonitorTrapException();
    elsif target_el == EL2 then
      exception = ExceptionSyndrome(Exception_WFxTrap);
      exception.syndrome<24:20> = Condition Syndrome();
      case wfxtype of
        when WFxType_WFI
          exception.syndrome<0> = '0';
        when WFxType_WFE
          exception.syndrome<0> = '1';
      AArch32.TakeHypTrapException(exception);
      else
        AArch32.TakeUndefInstrException();
  
  aarch32/exceptions/traps/AArch32.CheckITEnabled

// AArch32.CheckITEnabled()
// ========================
// Check whether the T32 IT instruction is disabled.
AArch32.CheckITEnabled(bits(4) mask)
bit it_disabled;
if PSTATE.EL == EL2 then
    it_disabled = HSCTLR.ITD;
else
    it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR[].ITD);
if it_disabled == '1' then
    if mask != '1000' then UNDEFINED;
    // Otherwise whether the IT block is allowed depends on hw1 of the next instruction.
    next_instr = AArch32.MemSingle[NextInstrAddr(32), 2, AccType_IFETCH, TRUE];
    if next_instr IN {'11xxxxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxx', '01001xxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
        // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
        // taken on the IT instruction or the next instruction. This is not reflected in
        // the pseudocode, which always takes the exception on the IT instruction. This
        // also does not take into account cases where the next instruction is UNPREDICTABLE.
        UNDEFINED;
    return;

aarch32/exceptions/traps/AArch32.CheckIllegalState

// AArch32.CheckIllegalState()
// ================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.
AArch32.CheckIllegalState()
if AArch32.GeneralExceptionsToAArch64() then
    AArch64.CheckIllegalState();
elsif PSTATE.IL == '1' then
    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x04;
    if PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_IllegalState);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.TakeUndefInstrException();

aarch32/exceptions/traps/AArch32.CheckSETENDEnabled

// AArch32.CheckSETENDEnabled()
// ================
// Check whether the AArch32 SETEND instruction is disabled.
AArch32.CheckSETENDEnabled()
bit setend_disabled;
if PSTATE.EL == EL2 then
    setend_disabled = HSCTLR.SED;
else
    setend_disabled = (if ELUsingAArch32(EL1) then SCTLR.SED else SCTLR[].SED);
if setend_disabled == '1' then
    UNDEFINED;
return;
aarch32/exceptions/traps/AArch32.SystemAccessTrap

// AArch32.SystemAccessTrap()
// -----------------------------------
// Trapped System register access.

AArch32.SystemAccessTrap(bits(5) mode, integer ec)
(valid, target_el) = ELFromM32(mode);
assert valid && HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

if target_el == EL2 then
    exception = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch32.TakeHypTrapException(exception);
else
    AArch32.TakeUndefInstrException();

aarch32/exceptions/traps/AArch32.SystemAccessTrapSyndrome

// AArch32.SystemAccessTrapSyndrome()
// -----------------------------------
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR instructions,
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch32.SystemAccessTrapSyndrome(bits(32) instr, integer ec)

ExceptionRecord exception;

case ec of
    when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
    when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTrap);
    when 0x4    exception = ExceptionSyndrome(Exception_CP15RRTTrap);
    when 0x5    exception = ExceptionSyndrome(Exception_CP14RTTrap);
    when 0x6    exception = ExceptionSyndrome(Exception_CP14DTTrap);
    when 0x7    exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
    when 0x8    exception = ExceptionSyndrome(Exception_FPIDTrap);
    when 0xC    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
    otherwise   Unreachable();

bits(20) iss = Zeros(20);

if exception.exceptype == Exception_Uncategorized then
    return exception;
elsif exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap, Exception_CP15RTTrap} then
    // Trapped MRC/MCR, VMRS on FPSID
    iss<13:10> = instr<19:16>;        // CRn, Reg in case of VMRS
    iss<8:5>   = instr<15:12>;        // Rt
    iss<9>     = '0';                 // RES0
    if exception.exceptype != Exception_FPIDTrap then    // When trap is not for VMRS
        iss<19:17> = instr<7:5>;          // opc2
        iss<16:14> = instr<23:21>;        // opc1
        iss<4:1>   = instr<3:0>;          // CRm
    else //VMRS Access
        iss<19:17> = '000';               //opc2 - Hardcoded for VMRS
        iss<16:14> = '111';               //opc1 - Hardcoded for VMRS
        iss<4:1>   = '0000';              //CRm  - Hardcoded for VMRS
    elsif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss<19:16> = instr<7:4>;          // opc1
        iss<13:10> = instr<19:16>;        // Rt2
        iss<8:5>   = instr<15:12>;        // Rt
        iss<4:1>   = instr<3:0>;          // CRm
    elsif exception.exceptype == Exception_CP14DTTrap then
        // Trapped LDC/STC
        iss<19:12> = instr<7:0>;          // imm8

iss<4> = instr<23>;          // U
iss<2:1> = instr<24,21>;       // P,W
if instr<19:16> == '1111' then // Rn==15, LDC(Literal addressing)/STC
iss<8:5> = bits(4) UNKNOWN;
iss<3>   = '1';
iss<0> = instr<20>;                  // Direction

exception.syndrome<24:20> = ConditionSyndrome();
exception.syndrome<19:0>  = iss;

return exception;

aarch32/exceptions/traps/AArch32.TakeHypTrapException

// AArch32.TakeHypTrapException()
// ==================================
// Exceptions routed to Hyp mode as a Hyp Trap exception.
AArch32.TakeHypTrapException(integer ec)
   exception = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
AArch32.TakeHypTrapException(exception);

// AArch32.TakeHypTrapException()
// ==================================
// Exceptions routed to Hyp mode as a Hyp Trap exception.
AArch32.TakeHypTrapException(ExceptionRecord exception)
   assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);
   bits(32) preferred_exception_return = ThisInstrAddr(32);
   vect_offset = 0x14;
   AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);

aarch32/exceptions/traps/AArch32.TakeMonitorTrapException

// AArch32.TakeMonitorTrapException()
// ==================================
// Exceptions routed to Monitor mode as a Monitor Trap exception.
AArch32.TakeMonitorTrapException()
   assert HaveEL(EL3) && ELUsingAArch32(EL3);
   bits(32) preferred_exception_return = ThisInstrAddr(32);
   vect_offset = 0x04;
   lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;
   AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);

aarch32/exceptions/traps/AArch32.TakeUndefInstrException

// AArch32.TakeUndefInstrException()
// ==================================
AArch32.TakeUndefInstrException()
   exception = ExceptionSyndrome(Exception_Uncategorized);
AArch32.TakeUndefInstrException(exception);

// AArch32.TakeUndefInstrException()
// ==================================
AArch32.TakeUndefInstrException(ExceptionRecord exception)
route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';
bits(32) preferred_exception_return = ThisInstrAddr(32);
vect_offset = 0x04;
lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;
if PSTATE.EL == EL2 then
  AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
elsif route_to_hyp then
  AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
else
  AArch32.EnterMode(M32_Undef, preferred_exception_return, lr_offset, vect_offset);

aarch32/exceptions/traps/AArch32.UndefnedFault

// AArch32.UndefnedFault()
// ========================
AArch32.UndefnedFault()

  if AArch32.GeneralExceptionsToAArch64() then AArch64.UndefnedFault();
  AArch32.Take.UndefnedInstrException();

J1.2.3 aarch32/functions

This section includes the following pseudocode functions:
• aarch32/functions/aborts/AArch32.DomainValid on page J1-11292.
• aarch32/functions/aborts/AArch32.FaultStatusLD on page J1-11292.
• aarch32/functions/aborts/AArch32.FaultStatusSD on page J1-11293.
• aarch32/functions/aborts/AArch32.FaultSyndrome on page J1-11293.
• aarch32/functions/aborts/EncodeSDFSC on page J1-11294.
• aarch32/functions/common/A32ExpandImm on page J1-11294.
• aarch32/functions/common/A32ExpandImm_C on page J1-11295.
• aarch32/functions/common/DecodeImmShift on page J1-11295.
• aarch32/functions/common/DecodeRegShift on page J1-11295.
• aarch32/functions/common/RRX on page J1-11295.
• aarch32/functions/common/RRX_C on page J1-11296.
• aarch32/functions/common/SRType on page J1-11296.
• aarch32/functions/common/Shift on page J1-11296.
• aarch32/functions/common/Shift_C on page J1-11296.
• aarch32/functions/common/T32ExpandImm on page J1-11296.
• aarch32/functions/common/T32ExpandImm_C on page J1-11297.
• aarch32/functions/common/VBitOps on page J1-11297.
• aarch32/functions/common/VCGEType on page J1-11297.
• aarch32/functions/common/VCGTType on page J1-11297.
• aarch32/functions/common/VFPNegMul on page J1-11297.
• aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps on page J1-11297.
• aarch32/functions/exclusive/AArch32.IsExclusiveVA on page J1-11298.
• aarch32/functions/exclusive/AArch32.MarkExclusiveVA on page J1-11299.
• aarch32/functions/exclusive/AArch32.SetExclusiveMonitors on page J1-11299.
• aarch32/functions/float/CheckAdvSIMDEnabled on page J1-11299.
• aarch32/functions/float/CheckAdvSIMDOrVFPEnabled on page J1-11299.
• aarch32/functions/float/CheckCryptoEnabled32 on page J1-11300.
• aarch32/functions/float/CheckVFPEndabled on page J1-11300.
• aarch32/functions/float/FPHalvedSub on page J1-11300.
• aarch32/functions/float/FPSqrtStep on page J1-11301.
• aarch32/functions/float/FPRecipStep on page J1-11301.
• aarch32/functions/float/StandardFPSCRValue on page J1-11301.
• aarch32/functions/memory/AArch32.CheckAlignment on page J1-11301.
• aarch32/functions/memory/Hint_PreloadData on page J1-11303.
• aarch32/functions/memory/Hint_PreloadDataForWrite on page J1-11303.
• aarch32/functions/memory/Hint_PreloadInstr on page J1-11303.
• aarch32/functions/memory/MemA on page J1-11303.
• aarch32/functions/memory/MemO on page J1-11304.
• aarch32/functions/memory/MemS on page J1-11304.
• aarch32/functions/memory/MemU on page J1-11304.
• aarch32/functions/memory/MemU_unpriv on page J1-11304.
• aarch32/functions/memory/Mem_with_type on page J1-11305.
• aarch32/functions/ras/AArch32.ESBOperation on page J1-11306.
• aarch32/functions/ras/AArch32.SErrorSyndrome on page J1-11307.
• aarch32/functions/ras/AArch32.vESBOperation on page J1-11307.
• aarch32/functions/registers/AArch32.ResetGeneralRegisters on page J1-11308.
• aarch32/functions/registers/AArch32.ResetSIMDFPRegisters on page J1-11308.
• aarch32/functions/registers/AArch32.ResetSpecialRegisters on page J1-11308.
• aarch32/functions/registers/AArch32.ResetSystemRegisters on page J1-11309.
• aarch32/functions/registers/ALUExceptionReturn on page J1-11309.
• aarch32/functions/registers/ALUWritePC on page J1-11309.
• aarch32/functions/registers/BXWritePC on page J1-11309.
• aarch32/functions/registers/BranchWritePC on page J1-11310.
• aarch32/functions/registers/LoadWritePC on page J1-11310.
• aarch32/functions/registers/D on page J1-11310.
• aarch32/functions/registers/Din on page J1-11311.
• aarch32/functions/registers/LR on page J1-11311.
• aarch32/functions/registers/LoadWritePC on page J1-11311.
• aarch32/functions/registers/LookUpRIndex on page J1-11311.
• aarch32/functions/registers/Monitor_mode_registers on page J1-11311.
• aarch32/functions/registers/PC on page J1-11312.
• aarch32/functions/registers/PCStoreValue on page J1-11312.
• aarch32/functions/registers/Q on page J1-11312.
• aarch32/functions/registers/Qin on page J1-11312.
• aarch32/functions/registers/R on page J1-11312.
• aarch32/functions/registers/RBankSelect on page J1-11313.
• aarch32/functions/registers/Rmode on page J1-11313.
• aarch32/functions/registers/S on page J1-11314.
• aarch32/functions/registers/_Dclone on page J1-11314.
• aarch32/functions/system/AArch32.ExceptionReturn on page J1-11314.
• aarch32/functions/system/AArch32.ExecutingCP10or11Instr on page J1-11315.
• aarch32/functions/system/AArch32.ITAdvance on page J1-11315.
• aarch32/functions/system/AArch32.SysRegRead on page J1-11315.
• aarch32/functions/system/AArch32.SysRegRead64 on page J1-11315.


```
---

aarch32/functions/system/AArch32.SysRegWrite on page J1-11316.
aarch32/functions/system/AArch32.SysRegWrite64 on page J1-11316.
aarch32/functions/system/AArch32.SysRegWriteM on page J1-11316.
aarch32/functions/system/AArch32.WriteModeByInstr on page J1-11316.
aarch32/functions/system/BadMode on page J1-11317.
aarch32/functions/system/BankedRegisterAccessValid on page J1-11317.
aarch32/functions/system/CPSRWriteByInstr on page J1-11318.
aarch32/functions/system/ConditionPassed on page J1-11318.
aarch32/functions/system/CurrentCond on page J1-11318.
aarch32/functions/system/InITBlock on page J1-11318.
aarch32/functions/system/LastInITBlock on page J1-11319.
aarch32/functions/system/SPSRWriteByInstr on page J1-11319.
aarch32/functions/system/SPSRaccessValid on page J1-11319.
aarch32/functions/system/SelectInstrSet on page J1-11320.
aarch32/functions/v6simd/Sat on page J1-11320.
aarch32/functions/v6simd/SignedSat on page J1-11320.
aarch32/functions/v6simd/UnsignedSat on page J1-11320.
```

```
aarch32/functions/aborts/AArch32.DomainValid

// AArch32.DomainValid()
// =====================
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

boolean AArch32.DomainValid(Fault statuscode, integer level)

assert statuscode != Fault_None;

case statuscode of
  when Fault_Domain
    return TRUE;
  when Fault_Translation, Fault_AccessFlag, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk
    return level == 2;
  otherwise
    return FALSE;

aarch32/functions/aborts/AArch32.FaultStatusLD

// AArch32.FaultStatusLD()
// =======================
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Long-descriptor format.

bits(32) AArch32.FaultStatusLD(boolean d_side, FaultRecord fault)

assert fault.statuscode != Fault_None;

bits(32) fsr = Zeros(32);
if HaveBASExt() & IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
if d_side then
  if fault.acctype IN {AccType_DC, AccType_IC,
    AccType_AT, AccType_ATPAN} then
    fsr<13> = '1'; fsr<11> = '1';
  else
    fsr<11> = if fault.write then '1' else '0';
  if IsExternalAbort(fault) then fsr<12> = fault.extflag;
  fsr<9> = '1';
  fsr<5:0> = EncodeLDFSC(fault.statuscode, fault.level);
```
return fsr;

aarch32/functions/aborts/AArch32.FaultStatusSD

// AArch32.FaultStatusSD()
// =======================
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Short-descriptor format.

bits(32) AArch32.FaultStatusSD(boolean d_side, FaultRecord fault)
assert fault.statuscode != Fault_None;

bits(32) fsr = Zeros(32);
if HaveRASExt() && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
if d_side then
  if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT, AccType_ATPAN} then
    fsr<13> = '1'; fsr<11> = '1';
  else
    fsr<11> = if fault.write then '1' else '0';
if IsExternalAbort(fault) then fsr<12> = fault.extflag;
fsr<9> = '0';
fsr<8:3:0> = EncodeSDFSC(fault.statuscode, fault.level);
if d_side then
  fsr<7:4> = fault.domain; // Domain field (data fault only)
return fsr;

aarch32/functions/aborts/AArch32.FaultSyndrome

// AArch32.FaultSyndrome()
// =======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// AArch32 Hyp mode.

bits(25) AArch32.FaultSyndrome(boolean d_side, FaultRecord fault)
assert fault.statuscode != Fault_None;

bits(25) iss = Zeros(25);
if HaveRASExt() && IsAsyncAbort(fault) then
  iss<11:10> = fault.errortype; // AET
if d_side then
  if (IsSecondStage(fault) & !fault.s2fs1walk &
      (!HaveRASExt() & fault.acctype == AccType_TTW &
       boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk")) then
    iss<24:14> = LSInstructionSyndrome();
  if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT, AccType_ATPAN} then
    iss<8> = '1'; iss<6> = '1';
  else
    iss<6> = if fault.write then '1' else '0';
if IsExternalAbort(fault) then iss<9> = fault.extflag;
iss<7> = if fault.s2fs1walk then '1' else '0';
iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);
return iss;
aarch32/functions/aborts/EncodeSDFSC

// EncodeSDFSC()
// =============
// Function that gives the Short-descriptor FSR code for different types of Fault

bits(5) EncodeSDFSC(Fault statuscode, integer level)

bits(5) result;

case statuscode of
  when Fault_AccessFlag
    assert level IN {1,2};
    result = if level == 1 then '00011' else '00110';
  when Fault_Alignment
    result = '00001';
  when Fault_Permission
    assert level IN {1,2};
    result = if level == 1 then '01101' else '01111';
  when Fault_Domain
    assert level IN {1,2};
    result = if level == 1 then '01001' else '01011';
  when Fault_Translation
    assert level IN {1,2};
    result = if level == 1 then '00101' else '00111';
  when Fault_SyncExternal
    result = '01000';
  when Fault_SyncExternalOnWalk
    assert level IN {1,2};
    result = if level == 1 then '01100' else '01110';
  when Fault_SyncParity
    result = '11001';
  when Fault_SyncParityOnWalk
    assert level IN {1,2};
    result = if level == 1 then '11100' else '11110';
  when Fault_AsyncParity
    result = '11000';
  when Fault_AsyncExternal
    result = '10110';
  when Fault_Debug
    result = '00010';
  when Fault_TLBConflict
    result = '10000';
  when Fault_Lockdown
    result = '10100'; // IMPLEMENTATION DEFINED
  when Fault_Exclusive
    result = '10101'; // IMPLEMENTATION DEFINED
  when Fault_ICacheMaint
    result = '00100';
  otherwise
    Unreachable();

return result;

aarch32/functions/common/A32ExpandImm

// A32ExpandImm()
// ===============

bits(32) A32ExpandImm(bits(12) imm12)

// PSTATE.C argument to following function call does not affect the imm32 result.
(imm32, -) = A32ExpandImm_C(imm12, PSTATE.C);

return imm32;
aarch32/functions/common/A32ExpandImm_C

// A32ExpandImm_C()
// ================
(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

unrotated_value = ZeroExtend(imm12<7:0>, 32);
(imm32, carry_out) = Shift_C(unrotated_value, SRT_Type_ROR, 2*UInt(imm12<11:8>), carry_in);
return (imm32, carry_out);

aarch32/functions/common/DecodeImmShift

// DecodeImmShift()
// ================
(SRT_Type, integer) DecodeImmShift(bits(2) srtype, bits(5) imm5)

SRT_Type shift_t;
integer shift_n;
case srtype of
  when '00' shift_t = SRT_Type_LSL;  shift_n = UInt(imm5);
  when '01' shift_t = SRT_Type_LSR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
  when '10' shift_t = SRT_Type_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
  when '11' if imm5 == '00000' then
    shift_t = SRT_Type_RRX;  shift_n = 1;
  else
    shift_t = SRT_Type_ROR;  shift_n = UInt(imm5);

return (shift_t, shift_n);

aarch32/functions/common/DecodeRegShift

// DecodeRegShift()
// ================
SRT_Type DecodeRegShift(bits(2) srtype)

SRT_Type shift_t;
case srtype of
  when '00' shift_t = SRT_Type_LSL;
  when '01' shift_t = SRT_Type_LSR;
  when '10' shift_t = SRT_Type_ASR;
  when '11' shift_t = SRT_Type_ROR;
return shift_t;

aarch32/functions/common/RRX

// RRX()
// ======
bits(N) RRX(bits(N) x, bit carry_in)
(result, -) = RRX_C(x, carry_in);
return result;
aarch32/functions/common/RRX_C

// RRX_C()
//===

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
result = carry_in : x<N-1:1>;
carry_out = x<0>;
return (result, carry_out);

aarch32/functions/common/SRType

enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};

aarch32/functions/common/Shift

// Shift()
//===

bits(N) Shift(bits(N) value, SRType srtype, integer amount, bit carry_in)
(result, -) = Shift_C(value, srtype, amount, carry_in);
return result;

aarch32/functions/common/Shift_C

// Shift_C()
//===

(bits(N), bit) Shift_C(bits(N) value, SRType srtype, integer amount, bit carry_in)
assert !(srtype == SRType_RRX && amount != 1);

bits(N) result;
bit carry_out;
if amount == 0 then
  (result, carry_out) = (value, carry_in);
else
  case srtype of
    when SRType_LSL
      (result, carry_out) = LSL_C(value, amount);
    when SRType_LSR
      (result, carry_out) = LSR_C(value, amount);
    when SRType_ASR
      (result, carry_out) = ASR_C(value, amount);
    when SRType_ROR
      (result, carry_out) = ROR_C(value, amount);
    when SRType_RRX
      (result, carry_out) = RRX_C(value, carry_in);
  return (result, carry_out);

aarch32/functions/common/T32ExpandImm

// T32ExpandImm()
//===

bits(32) T32ExpandImm(bits(12) imm12)

// PSTATE.C argument to following function call does not affect the imm32 result.
(imm32, -) = T32ExpandImm_C(imm12, PSTATE.C);
return imm32;

aarch32/functions/common/T32ExpandImm_C

// T32ExpandImm_C()
// ================

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)
bits(32) imm32;
bit carry_out;
if imm12<11:10> == '00' then
    case imm12<9:8> of
    when '00'
        imm32 = ZeroExtend(imm12<7:0>, 32);
    when '01'
        imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
    when '10'
        imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
    when '11'
        imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
    carry_out = carry_in;
else
    unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
    (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));
return (imm32, carry_out);

aarch32/functions/common/VBitOps

enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

aarch32/functions/common/VCGEType

enumeration VCGEType {VCGEType_signed, VCGEType_unsigned, VCGEType_fp};

aarch32/functions/common/VCGTtype

enumeration VCGTtype {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};

aarch32/functions/common/VFPNegMul

enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};

aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps

// AArch32.CheckCP15InstrCoarseTraps()
// ===============================

// Check for coarse-grained traps to System registers in the
// coproc=0b1111 encoding space by HSTR and HCR.

AArch32.CheckCP15InstrCoarseTraps(integer Crn, integer nreg, integer Crm)
if PSTATE.EL == EL0 && (!ELUsingAArch32(EL1) ||
    (EL2Enabled() && !ELUsingAArch32(EL2))) then
    AArch64.CheckCP15InstrCoarseTraps(Crn, nreg, Crm);
    trapped_encoding = ((Crn == 9 && Crm IN {0,1,2, 5,6,7,8 }) ||
Armv8 Pseudocode
J1.2 Pseudocode for AArch32 operation

(Crn == 10 && Crm IN {0,1, 4, 8 }) ||
(Crn == 11 && Crm IN {0,1,2,3,4,5,6,7,8,15});

// Check for coarse-grained Hyp traps
if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
  major = if nreg == 1 then Crn else Crm;
  // Check for MCR, MRC, MCR, and MRRC disabled by HSTR<Crn/Crm>
  // and MRC and MCR disabled by HCR.TIDCP.
  if ((!(major IN {4,14}) && HSTR<major> == '1') ||
      (HCR.TIDCP == '1' && nreg == 1 && trapped_encoding)) then
    if (PSTATE.EL == EL0 &&
    boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0") then
      UNDEFINED;
    if ELUsingAArch32(EL2) then
      AArch32.SystemAccessTrap(M32_Hyp, 0x3);
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x3);

aarch32/functions/exclusive/AArch32.ExclusiveMonitorsPass

// AArch32.ExclusiveMonitorsPass()
// =====================================================
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)

  // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
  // before or after the check on the local Exclusives monitor. As a result a failure
  // of the local monitor can occur on some implementations even if the memory
  // access would give an memory abort.

  acctype = AccType_ATOMIC;
  iswrite = TRUE;

  aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

  passed = AArch32.IsExclusiveVA(address, ProcessorID(), size);
  if !passed then
    return FALSE;

  memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
  // Check for aborts or debug exceptions
  if IsFault(memaddrdesc) then
    AArch32.Abort(address, memaddrdesc.fault);

  passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
  ClearExclusiveLocal(ProcessorID());

  if passed then
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
      passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

  return passed;

aarch32/functions/exclusive/AArch32.IsExclusiveVA

// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
// It is always safe to return TRUE which will check the physical address only.
boolean AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size);

aarch32/functions/exclusive/AArch32.MarkExclusiveVA

// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size);

aarch32/functions/exclusive/AArch32.SetExclusiveMonitors

// AArch32.SetExclusiveMonitors()
// ==================================
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.
AArch32.SetExclusiveMonitors(bits(32) address, integer size)
  acctype = AccType_ATOMIC;
  iswrite = FALSE;
  aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
  memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  return;
if memaddrdesc.memattrs.shareability != Shareability_NSH then
  MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
  MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
AArch32.MarkExclusiveVA(address, ProcessorID(), size);

aarch32/functions/float/CheckAdvSIMDEnabled

// CheckAdvSIMDEnabled()
// =====================
CheckAdvSIMDEnabled()
  fpexc_check = TRUE;
  advsimd = TRUE;
  AArch32.CheckAdvSIMDOrFPEnabled(fpexc_check, advsimd);
  // Return from CheckAdvSIMDOrFPEnabled() occurs only if Advanced SIMD access is permitted

// Make temporary copy of D registers
// _Dclone[] is used as input data for instruction pseudocode
for i = 0 to 31
  _Dclone[i] = D[i];
return;

aarch32/functions/float/CheckAdvSIMDOrVFPEnabled

// CheckAdvSIMDOrVFPEnabled()
// ==========================

CheckAdvSIMDorVFPEnabled(boolean include_fpxc_check, boolean advsimd)
AArch32.CheckAdvSIMDorVFPEnabled(include_fpxc_check, advsimd);
// Return from CheckAdvSIMDorVFPEnabled() occurs only if VFP access is permitted
return;

aarch32/functions/float/CheckCryptoEnabled32

// CheckCryptoEnabled32()
// ======================
CheckCryptoEnabled32()
CheckAdvSIMDEnabled();
// Return from CheckAdvSIMDEnabled() occurs only if access is permitted
return;

aarch32/functions/float/CheckVFPEnabled

// CheckVFPEnabled()
// ===============
CheckVFPEnabled(boolean include_fpxc_check)
advsimd = FALSE;
AArch32.CheckAdvSIMDorVFPEnabled(include_fpxc_check, advsimd);
// Return from CheckAdvSIMDorVFPEnabled() occurs only if VFP access is permitted
return;

aarch32/functions/float/FPHalvedSub

// FPHalvedSub()
// =============
bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRTYPE fpcr)
assert N IN {16,32,64};
rounding = FPRoundingMode(fpcr);
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
  zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
  if inf1 && inf2 && sign1 == sign2 then
    result = FPDefaultNaN(fpcr, N);
    FPProcessException(FPExc_InvalidOp, fpcr);
  elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
    result = FPInfinity('0', N);
  elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
    result = FPInfinity('1', N);
  elsif zero1 && zero2 && sign1 != sign2 then
    result = FPZero(sign1, N);
  else
    result_value = (value1 - value2) / 2.0;
    if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
      result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
      result = FPZero(result_sign, N);
    else
      result = FPRound(result_value, fpcr, N);
  end if
end if
return result;
aarch32/functions/float/FPRSqrtStep

// FPRSqrtStep()
// =============

bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
assert N IN {16, 32};
FPCRType fpcr = StandardFPSCRValue();
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
  zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
  bits(N) product;
  if (inf1 && zero2) || (zero1 && inf2) then
    product = FPZero('0', N);
  else
    product = FPMul(op1, op2, fpcr);
  bits(N) three = FPThree('0', N);
  result = FPHalvedSub(three, product, fpcr);
return result;

aarch32/functions/float/FPRecipStep

// FPRecipStep()
// =============

bits(N) FPRecipStep(bits(N) op1, bits(N) op2)
assert N IN {16, 32};
FPCRType fpcr = StandardFPSCRValue();
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
  zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
  bits(N) product;
  if (inf1 && zero2) || (zero1 && inf2) then
    product = FPZero('0', N);
  else
    product = FPMul(op1, op2, fpcr);
  bits(N) two = FPTwo('0', N);
  result = FPSub(two, product, fpcr);
return result;

aarch32/functions/float/StandardFPSCRValue

// StandardFPSCRValue()
// ========================

FPCRType StandardFPSCRValue()
bits(32) value = '00000' : FPSCR.AHP : '11000' : FPSCR.FZ16 : '0000000000000000000';
return ZeroExtend(value, 64);

aarch32/functions/memory/AArch32.CheckAlignment

// AArch32.CheckAlignment()
// ========================

boolean AArch32.CheckAlignment(bits(32) address, integer alignment, AccType acctype,
  boolean iswrite)
bit A;
if PSTATE.EL == EL0 && !ELUsingAArch32(S1TranslationRegime()) then
    A = SCTLR[].A; //use AArch64 register, when higher Exception level is using AArch64
elsif PSTATE.EL == EL2 then
    A = HSCTLR.A;
else
    A = SCTLR.A;
end if
aligned = (address == Align(address, alignment));
atomic  = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC,
                      AccType_ORDEREDATOMICRW, AccType_ATOMICLS64, AccType_A32LSMD};
ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED,
                      AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW };
vector  = acctype == AccType_VEC;
// AccType_VEC is used for SIMD element alignment checks only
check = (atomic || ordered || vector || A == '1');
if check && !aligned then
    secondstage = FALSE;
    AArch32.Abort(address, AlignmentFault(acctype, iswrite, secondstage));
return aligned;

aarch32/functions/memory/AArch32.MemSingle

// AArch32.MemSingle[] - non-assignment (read) form
//=================================================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned]
    boolean ispair = FALSE;
    return AArch32.MemSingle[address, size, acctype, aligned, ispair];

// AArch32.MemSingle[] - non-assignment (read) form
//=================================================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned, boolean ispair]
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);
    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);
    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    PhysMemRetStatus memstatus;
    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
    return value;

// AArch32.MemSingle[] - assignment (write) form
//=================================================================================

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned] = bits(size*8) value
    boolean ispair = FALSE;
    AArch32.MemSingle[address, size, acctype, aligned, ispair] = value;
```
return;

// AArch32.MemSingle[] - assignment (write) form
// =============================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned, boolean ispair] =
bits(size*8) value
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

AddressDescriptor memaddrdesc;
iswrite = TRUE;
memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch32.Abort(address, memaddrdesc.fault);
// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);
// Memory array access
accdesc = CreateAccessDescriptor(acctype);
PhysMemRetStatus memstatus;
memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
if IsFault(memstatus) then
    HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    return;
```

```
aarch32/functions/memory/Hint_PreloadData
Hint_PreloadData(bits(32) address);
```

```
aarch32/functions/memory/Hint_PreloadDataForWrite
Hint_PreloadDataForWrite(bits(32) address);
```

```
aarch32/functions/memory/Hint_PreloadInstr
Hint_PreloadInstr(bits(32) address);
```

```
aarch32/functions/memory/MemA
// MemA[] - non-assignment form
// ----------------------------
bits(8*size) MemA[bits(32) address, integer size]
    acctype = AccType_ATOMIC;
    return Mem_with_type[address, size, acctype];
// MemA[] - assignment form
// -------------------------
MemA[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_ATOMIC;
    Mem_with_type[address, size, acctype] = value;
    return;
```
**aarch32/functions/memory/MemO**

// MemO[] - non-assignment form
// ---------------------------------------

```
bits(8*size) MemO[bits(32) address, integer size]
    acctype = AccType_ORDERED;
    return Mem_with_type[address, size, acctype];
```

// MemO[] - assignment form
// ---------------------------------------

```
MemO[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_ORDERED;
    Mem_with_type[address, size, acctype] = value;
    return;
```

**aarch32/functions/memory/MemS**

// MemS[] - non-assignment form
// ---------------------------------------

```
Memory accessor for streaming load multiple instructions

bits(8*size) MemS[bits(32) address, integer size]
    acctype = AccType_A32LSMD;
    return Mem_with_type[address, size, acctype];
```

// MemS[] - assignment form
// ---------------------------------------

```
MemS[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_A32LSMD;
    Mem_with_type[address, size, acctype] = value;
    return;
```

**aarch32/functions/memory/MemU**

// MemU[] - non-assignment form
// ---------------------------------------

```
bits(8*size) MemU[bits(32) address, integer size]
    acctype = AccType_NORMAL;
    return Mem_with_type[address, size, acctype];
```

// MemU[] - assignment form
// ---------------------------------------

```
MemU[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_NORMAL;
    Mem_with_type[address, size, acctype] = value;
    return;
```

**aarch32/functions/memory/MemU_unpriv**

// MemU_unpriv[] - non-assignment form
// ---------------------------------------

```
bits(8*size) MemU_unpriv[bits(32) address, integer size]
    acctype = AccType_UNPRIV;
    return Mem_with_type[address, size, acctype];
```

// MemU_unpriv[] - assignment form
// ===============================
MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
acctype = AccType_UNPRIV;
Mem_with_type[address, size, acctype] = value;
return;

aarch32/functions/memory/Mem_with_type
// Mem_with_type[] - non-assignment (read) form
// ===============================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype]
boolean ispair = FALSE;
return Mem_with_type[address, size, acctype, ispair];

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype, boolean ispair]
assert size IN {1, 2, 4, 8, 16};
constant halfsize = size DIV 2;
bits(size = 8) value;
boolean iswrite = FALSE;
boolean aligned;
if ispair then
  // check alignment on size of element accessed, not overall access size
  aligned = AArch32.CheckAlignment(address, halfsize, acctype, iswrite);
else
  aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
if !aligned then
  assert size > 1;
  value<7:0> = AArch32.MemSingle[address, 1, acctype, aligned];
  // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
  // access will generate an Alignment Fault, as to get this far means the first byte did
  // not, so we must be changing to a new translation page.
  c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
  assert c IN {Constraint_FAULT, Constraint_NONE};
  if c == Constraint_NONE then aligned = TRUE;
  for i = 1 to size-1
    value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, acctype, aligned];
  else
    value = AArch32.MemSingle[address, size, acctype, aligned, ispair];
  if BigEndian(acctype) then
    value = BigEndianReverse(value);
  return value;

// Mem_with_type[] - assignment (write) form
// ===============================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccType acctype] = bits(size*8) value_in
boolean ispair = FALSE;
Mem_with_type[address, size, acctype, ispair] = value_in;

Mem_with_type[bits(32) address, integer size, AccType acctype, boolean ispair] = bits(size*8) value_in
boolean iswrite = TRUE;
constant halfsize = size DIV 2;
bits(size=8) value = value_in;
boolean aligned;
if BigEndian(acctype) then
value = BigEndianReverse(value);

if ispair then
    // check alignment on size of element accessed, not overall access size
    aligned = AArch32.CheckAlignment(address, halfsize, acctype, iswrite);
else
    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

if !aligned then
    assert size > 1;
    AArch32.MemSingle[address, 1, acctype, aligned] = value<7:0>;

// For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
// access will generate an Alignment Fault, as to get this far means the first byte did
// not, so we must be changing to a new translation page.
c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
assert c IN {Constraint_FAULT, Constraint_NONE};
if c == Constraint_NONE then aligned = TRUE;

for i = 1 to size-1
    AArch32.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
else
    AArch32.MemSingle[address, size, acctype, aligned, ispair] = value;
return;

aarch32/functions/ras/AArch32.ESBOperation

// AArch32.ESBOperation()
// ======================
// Perform the AArch32 ESB operation for ESB executed in AArch32 state

AArch32.ESBOperation()

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
    route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1';
if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
    route_to_aarch64 = SCR_EL3.EA == '1';
if route_to_aarch64 then
    AArch64.ESBOperation();
    return;

route_to_monitor = HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.EA == '1';
route_to_hyp = PSTATE.EL IN {EL0, EL1} && EL2Enabled() && (HCR.TGE == '1' || HCR.AMO == '1');

bits(5) target;
if route_to_monitor then
    target = M32_Monitor;
elif route_to_hyp || PSTATE.M == M32_Hyp then
    target = M32_Hyp;
else
    target = M32_Abort;

boolean mask_active;
if CurrentSecurityState() == SS_Secure then
    mask_active = TRUE;
elif target == M32_Monitor then
    mask_active = SCR.AW == '1' && (HaveEL(EL2) || (HCR.TGE == '0' && HCR.AMO == '0'));
else
    mask_active = target == M32_Abort || PSTATE.M == M32_Hyp;

mask_set = PSTATE.A == '1';
(-, el) = ELFromM32(target);
intdis = Halted() || ExternalDebugInterruptsDisabled(el);
masked   = intdis || (mask_active && mask_set);

// Check for a masked Physical SError pending that can be synchronized
// by an Error synchronization event.
if masked && IsSynchronizablePhysicalSErrorPending() then
    syndrome32 = AArch32.PhysicalSErrorSyndrome();
    DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
    ClearPendingPhysicalSError();

return;

aarch32/functions/ras/AArch32.PhysicalSErrorSyndrome

// Return the SError syndrome
AArch32.SErrorSyndrome AArch32.PhysicalSErrorSyndrome();

aarch32/functions/ras/AArch32.ReportDeferredSError

// AArch32.ReportDeferredSError()
// ================
// Return deferred SError syndrome

bits(32) AArch32.ReportDeferredSError(bits(2) AET, bit ExT)

bits(32) target;
target<31> = '1';                     // A
syndrome = Zeros(16);
if PSTATE.EL == EL2 then
    syndrome<11:10> = AET;           // AET
    syndrome<9>     = ExT;           // EA
    syndrome<5:0>   = '010001';       // DFSC
else
    syndrome<15:14> = AET;           // AET
    syndrome<12>    = ExT;           // ExT
    syndrome<9>     = TTBCR.EAE;     // LPAE
    if TTBCR.EAE == '1' then         // Long-descriptor format
        syndrome<5:0>   = '010001';   // STATUS
    else                           // Short-descriptor format
        syndrome<10,3:0> = '10110';   // FS
if HaveAArch64() then
    target<24:0> = ZeroExtend(syndrome, 25);// Any RES0 fields must be set to zero
else
    target<15:0> = syndrome;
return target;

aarch32/functions/ras/AArch32.SErrorSyndrome

type AArch32.SErrorSyndrome is (                  
    bits(2) AET,
    bit ExT
)

aarch32/functions/ras/AArch32.vESBOperation

// AArch32.vESBOperation()
// ==============
// Perform the ESB operation for virtual SError interrupts executed in AArch32 state

AArch32.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // Check for EL2 using AArch64 state
if !ELUsingAArch32(EL2) then
  AArch64.vESBOperation();
return;

// If physical SError interrupts are routed to Hyp mode, and TGE is not set, then a
// virtual SError interrupt might be pending
vSEI_enabled = HCR.TGE == '0' && HCR.AMO == '1';
vSEI_pending = vSEI_enabled && HCR.VA == '1';
vintdis = Halted() || ExternalDebugInterruptsDisabled(EL1);
vmasked = vintdis || PSTATE.A == '1';

// Check for a masked virtual SError pending
if vSEI_pending && vmasked then
  VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
  HCR.VA = '0';                      // Clear pending virtual SError
return;

aarch32/functions/registers/AArch32.ResetGeneralRegisters

// AArch32.ResetGeneralRegisters()
// =================================
AArch32.ResetGeneralRegisters()

  for i = 0 to 7
    R[i] = bits(32) UNKNOWN;
  for i = 8 to 12
    Rmode[i, M32_User] = bits(32) UNKNOWN;
    Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
  if HaveEL(EL2) then Rmode[13, M32_Hyp] = bits(32) UNKNOWN;  // No R14_hyp
  for i = 13 to 14
    Rmode[i, M32_User] = bits(32) UNKNOWN;
    Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
    Rmode[i, M32_IRQ] = bits(32) UNKNOWN;
    Rmode[i, M32_Svc] = bits(32) UNKNOWN;
    Rmode[i, M32_Abort] = bits(32) UNKNOWN;
    Rmode[i, M32_Undef] = bits(32) UNKNOWN;
  if HaveEL(EL3) then Rmode[i, M32_Monitor] = bits(32) UNKNOWN;

  return;

aarch32/functions/registers/AArch32.ResetSIMDFPRegisters

// AArch32.ResetSIMDFPRegisters()
// ==============================
AArch32.ResetSIMDFPRegisters()

  for i = 0 to 15
    Q[i] = bits(128) UNKNOWN;

  return;

aarch32/functions/registers/AArch32.ResetSpecialRegisters

// AArch32.ResetSpecialRegisters()
// ===============================
AArch32.ResetSpecialRegisters()

    // AArch32 special registers
    SPSR_fiq<31:0> = bits(32) UNKNOWN;
SPSR_irq<31:0> = bits(32) UNKNOWN;
SPSR_svc<31:0> = bits(32) UNKNOWN;
SPSR_abt<31:0> = bits(32) UNKNOWN;
SPSR_und<31:0> = bits(32) UNKNOWN;
if HaveEL(EL2) then
    SPSR_hyp = bits(32) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
if HaveEL(EL3) then
    SPSR_mon = bits(32) UNKNOWN;

// External debug special registers
DLR = bits(32) UNKNOWN;
DSPSR = bits(32) UNKNOWN;

return;

aarch32/functions/registers/AArch32.ResetSystemRegisters

AArch32.ResetSystemRegisters(boolean cold_reset);

aarch32/functions/registers/ALUExceptionReturn

// ALUExceptionReturn()
// ==============
ALUExceptionReturn(bits(32) address)
// if PSTATE.EL == EL2 then
// UNDEFINED;
elsif PSTATE.M IN {M32_User,M32_System} then
    Constraint c = ConstrainUnpredictable(Unpredictable_ALUEXCEPTIONRETURN);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF
            UNDEFINED;
        when Constraint_NOP
            EndOfInstruction();
    else
        AArch32.ExceptionReturn(address, SPSR[]);

aarch32/functions/registers/ALUWritePC

// ALUWritePC()
// =============
ALUWritePC(bits(32) address)
// if CurrentInstrSet() == InstrSet_A32 then
// BXWritePC(address, BranchType_INDIR);
else
    BranchWritePC(address, BranchType_INDIR);

aarch32/functions/registers/BXWritePC

// BXWritePC()
// ============
BXWritePC(bits(32) address_in, BranchType branch_type)
bits(32) address = address_in;
if address<0> == '1' then
    SelectInstrSet(InstrSet_T32);
    address<0> = '0';
else
SelectInstrSet(InstrSet_A32);
// For branches to an unaligned PC counter in A32 state, the processor takes the branch
// and does one of:
// * Forces the address to be aligned
// * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
if address<1> == '1' & ConstrainUnpredictableBool(Unpredictable_A32FORCEALIGNPC) then
  address<1> = '0';
boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
BranchTo(address, branch_type, branch_conditional);

aarch32/functions/registers/BranchWritePC

// BranchWritePC()
// ===============
BranchWritePC(bits(32) address_in, BranchType branch_type)
  bits(32) address = address_in;
  if CurrentInstrSet() == InstrSet_A32 then
    address<1:0> = '00';
  else
    address<0> = '0';
  boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
  BranchTo(address, branch_type, branch_conditional);

aarch32/functions/registers/CBWritePC

// CBWritePC()
// ===========
// Takes a branch from a CBNZ/CBZ instruction.
CBWritePC(bits(32) address_in)
  bits(32) address = address_in;
  assert CurrentInstrSet() == InstrSet_T32;
  address<0> = '0';
  boolean branch_conditional = TRUE;
  BranchTo(address, BranchType_DIR, branch_conditional);

aarch32/functions/registers/D

// D[] - non-assignment form
// ===============
bits(64) D[integer n]
  assert n >= 0 && n <= 31;
  base = (n MOD 2) * 64;
  bits(128) vreg = V[n DIV 2, 128];
  return vreg<base+63:base>;

// D[] - assignment form
// ===============
D[integer n] = bits(64) value
  assert n >= 0 && n <= 31;
  base = (n MOD 2) * 64;
  bits(128) vreg = V[n DIV 2, 128];
  vreg<base+63:base> = value;
  V[n DIV 2, 128] = vreg;
  return;
aarch32/functions/registers/Din

// Din[] - non-assignment form
// ===========================

bits(64) Din[integer n]
    assert n >= 0 && n <= 31;
    return _Dclone[n];

aarch32/functions/registers/LR

// LR - assignment form
// =============

LR = bits(32) value
    R[14] = value;
    return;

// LR - non-assignment form
// ===========

bits(32) LR
    return R[14];

aarch32/functions/registers/LoadWritePC

// LoadWritePC()
// =============

LoadWritePC(bits(32) address)
    BXWritePC(address, BranchType_INDIR);

aarch32/functions/registers/LookUpRIndex

// LookUpRIndex()
// =============

integer LookUpRIndex(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;

    integer result;
    case n of // Select index by mode:  usr irq svc abt und hyp
        when 8  result = RBankSelect(mode,  8, 24,  8,  8,  8,  8,  8);
        when 9  result = RBankSelect(mode,  9, 25,  9,  9,  9,  9,  9);
        when 10 result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
        when 11 result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
        when 12 result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
        when 13 result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
        when 14 result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
        otherwise result = n;
    end;

    return result;

aarch32/functions/registers/Monitor_mode_registers

bits(32) SP_mon;
bits(32) LR_mon;
aarch32/functions/registers/PC

// PC - non-assignment form
// =============

bits(32) PC
return R[15]; // This includes the offset from AArch32 state

aarch32/functions/registers/PCStoreValue

// PCStoreValue()
// ===============

bits(32) PCStoreValue()
// This function returns the PC value. On architecture versions before Armv7, it
// is permitted to instead return PC+4, provided it does so consistently. It is
// used only to describe A32 instructions, so it returns the address of the current
// instruction plus 8 (normally) or 12 (when the alternative is permitted).
return PC;

aarch32/functions/registers/Q

// Q[] - non-assignment form
// ================

bits(128) Q[integer n]
assert n >= 0 && n <= 15;
return V[n, 128];

// Q[] - assignment form
// ================

Q[integer n] = bits(128) value
assert n >= 0 && n <= 15;
V[n, 128] = value;
return;

aarch32/functions/registers/Qin

// Qin[] - non-assignment form
// ================

bits(128) Qin[integer n]
assert n >= 0 && n <= 15;
return Din[2*n+1]:Din[2*n];

aarch32/functions/registers/R

// R[] - assignment form
// ================

R[integer n] = bits(32) value
Rmode[n, PSTATE.M] = value;
return;

// R[] - non-assignment form
// ================

bits(32) R[integer n]
if n == 15 then
    offset = (if CurrentInstrSet() == InstrSet_A32 then 8 else 4);
return _PC<31:0> + offset;
else
  return Rmode[n, PSTATE.M];

aarch32/functions/registers/RBankSelect

// RBankSelect()
// ===============

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
                     integer svc, integer abt, integer und, integer hyp)

integer result;

  case mode of
    when M32_User    result = usr;  // User mode
    when M32_FIQ     result = fiq;  // FIQ mode
    when M32_IRQHandler result = irq; // IRQ mode
    when M32_Svc     result = svc;  // Supervisor mode
    when M32_Abort   result = abt;  // Abort mode
    when M32_Hyp     result = hyp;  // Hyp mode
    when M32_Undef   result = und;  // Undefined mode
    when M32_System  result = usr;  // System mode uses User mode registers
    otherwise        Unreachable(); // Monitor mode
  end case;

return result;

aarch32/functions/registers/Rmode

// Rmode[] - non-assignment form
// ================

bits(32) Rmode[integer n, bits(5) mode]
assert n >= 0 && n <= 14;

// Check for attempted use of Monitor mode in Non-secure state.
if CurrentSecurityState() != SS_Secure then assert mode != M32_Monitor;
assert !BadMode(mode);

if mode == M32_Monitor then
  if n == 13 then return SP_mon;
  elsif n == 14 then return LR_mon;
  else return _R[n]<31:0>;
else
  return _R[LookUpRIndex(n, mode)]<31:0>;

// Rmode[] - assignment form
// ================

Rmode[integer n, bits(5) mode] = bits(32) value
assert n >= 0 && n <= 14;

// Check for attempted use of Monitor mode in Non-secure state.
if CurrentSecurityState() != SS_Secure then assert mode != M32_Monitor;
assert !BadMode(mode);

if mode == M32_Monitor then
  if n == 13 then SP_mon = value;
  elsif n == 14 then LR_mon = value;
  else _R[n]<31:0> = value;
else
  // It is constrained unpredictable whether the upper 32 bits of the X
  // register are unchanged or set to zero. This is also tested for on
  // exception entry, as this applies to all AArch32 registers.
  if HaveAArch64() && ConstrainUnpredictableBool(Unpredictable ZEROUPPER) then
Armv8 Pseudocode

J1.2 Pseudocode for AArch32 operation

_R[LookUpRIndex(n, mode)] = ZeroExtend(value, 64);
else
    _R[LookUpRIndex(n, mode)]<31:0> = value;

return;

aarch32/functions/registers/S

// S[] - non-assignment form
// ===============

bits(32) S[integer n]
assert n >= 0 && n <= 31;
base = (n MOD 4) * 32;
bits(128) vreg = V[n DIV 4, 128];
return vreg<base+31:base>;

// S[] - assignment form
// ===============

S[integer n] = bits(32) value
assert n >= 0 && n <= 31;
base = (n MOD 4) * 32;
bits(128) vreg = V[n DIV 4, 128];
vreg<base+31:base> = value;
V[n DIV 4, 128] = vreg;
return;

aarch32/functions/registers/_Dclone

array bits(64) _Dclone[0..31];

aarch32/functions/system/AArch32.ExceptionReturn

// AArch32.ExceptionReturn()
// ===============

AArch32.ExceptionReturn(bits(32) new_pc_in, bits(32) spsr)
bits(32) new_pc = new_pc_in;
SynchronizeContext();
// Attempts to change to an illegal mode or state will invoke the Illegal Execution state
// mechanism
SetPSTATEFromPSR(spsr);
ClearExclusiveLocal(ProcessorID());
SendEventLocal();

if PSTATE.IL == '1' then
    // If the exception return is illegal, PC[1:0] are UNKNOWN
    new_pc<1:0> = bits(2) UNKNOWN;
else
    // LR[1:0] or LR[0] are treated as being 0, depending on the target instruction set state
    if PSTATE.T == '1' then
        new_pc<0> = '0';                 // T32
    else
        new_pc<1:0> = '00';              // A32

boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
BranchTo(new_pc, BranchType_ERET, branch_conditional);
CheckExceptionCatch(FALSE);              // Check for debug event on exception return
**AArch32.ExecutingCP10or11Instr**

```java
boolean AArch32.ExecutingCP10or11Instr()

instr = ThisInstr();
instr_set = CurrentInstrSet();
assert instr_set IN {InstrSet_A32, InstrSet_T32};

if instr_set == InstrSet_A32 then
    return ((instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> IN {'101x'});
else // InstrSet_T32
    return (instr<31:28> IN {'111x'} && (instr<27:24> == '1110' || instr<27:25> == '110') &&
            instr<11:8> IN {'101x'});
```

**AArch32.ITAdvance**

```java
AArch32.ITAdvance()

if PSTATE.IT<2:0> == '000' then
    PSTATE.IT = '00000000';
else
    PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
return;
```

**AArch32.SysRegRead**

```java
// Read from a 32-bit AArch32 System register and write the register's contents to R[t].
AArch32.SysRegRead(integer cp_num, bits(32) instr, integer t);
```

**AArch32.SysRegRead64**

```java
// Read from a 64-bit AArch32 System register and write the register's contents to R[t] and R[t2].
AArch32.SysRegRead64(integer cp_num, bits(32) instr, integer t, integer t2);
```

**AArch32.SysRegReadCanWriteAPSR**

```java
// Determines whether the AArch32 System register read instruction can write to APSR flags.

boolean AArch32.SysRegReadCanWriteAPSR(integer cp_num, bits(32) instr)
assert UsingAArch32();
assert (cp_num IN {14,15});
assert cp_num == UInt(instr<11:8>);

opc1 = UInt(instr<23:21>);
opc2 = UInt(instr<7:5>);
CRn = UInt(instr<19:16>);
CRm = UInt(instr<3:0>);

if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
    return TRUE;
return FALSE;
```
**aaarch32/functions/system/AArch32.SysRegWrite**

// Read the contents of R[t] and write to a 32-bit AArch32 System register.
AArch32.SysRegWrite(integer cp_num, bits(32) instr, integer t);

**aaarch32/functions/system/AArch32.SysRegWrite64**

// Read the contents of R[t] and R[t2] and write to a 64-bit AArch32 System register.
AArch32.SysRegWrite64(integer cp_num, bits(32) instr, integer t, integer t2);

**aaarch32/functions/system/AArch32.SysRegWriteM**

// Read a value from a virtual address and write it to an AArch32 System register.
AArch32.SysRegWriteM(integer cp_num, bits(32) instr, bits(32) address);

**aaarch32/functions/system/AArch32.WriteMode**

// AArch32.WriteMode()
// ===================
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.
AArch32.WriteMode(bits(5) mode)
    (valid,el) = ELFromM32(mode);
    assert valid;
    PSTATE.M = mode;
    PSTATE.EL = el;
    PSTATE.nRW = '1';
    PSTATE.SP = (if mode IN {M32_User,M32_System} then '0' else '1');
    return;

**aaarch32/functions/system/AArch32.WriteModeByInstr**

// AArch32.WriteModeByInstr()
// ==========================
// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.IL.
AArch32.WriteModeByInstr(bits(5) mode)
    (valid,el) = ELFromM32(mode);
    // 'valid' is set to FALSE if 'mode' is invalid for this implementation or the current value
    // of SCR.NS/SCR_EL3.NS. Additionally, it is illegal for an instruction to write 'mode' to
    // PSTATE.EL if it would result in any of:
    // * A change to a mode that would cause entry to a higher Exception level.
    if UInt(el) > UInt(PSTATE.EL) then
        valid = FALSE;
    // * A change to or from Hyp mode.
    if (PSTATE.M == M32_Hyp || mode == M32_Hyp) && PSTATE.M != mode then
        valid = FALSE;
    // * When EL2 is implemented, the value of HCR.TGE is '1', a change to a Non-secure EL1 mode.
    if PSTATE.M == M32_Monitor && HaveEL(EL2) && el == EL1 && SCR.NS == '1' && HCR.TGE == '1' then
        valid = FALSE;
    if !valid then
        PSTATE.IL = '1';
        else
            AArch32.WriteMode(mode);
aarch32/functions/system/BadMode

// BadMode()
// =========

boolean BadMode(bits(5) mode)
// Return TRUE if 'mode' encodes a mode that is not valid for this implementation
boolean valid;
case mode of
  when M32_Monitor
    valid = HaveAArch32EL(EL3);
  when M32_Hyp
    valid = HaveAArch32EL(EL2);
  when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
    // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
    // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
    // AArch64, then these modes are EL1 modes.
    // Therefore it is sufficient to test this implementation supports EL1 using AArch32.
    valid = HaveAArch32EL(EL1);
  when M32_User
    valid = HaveAArch32EL(EL0);
  otherwise
    valid = FALSE;           // Passed an illegal mode value
return !valid;

aarch32/functions/system/BankedRegisterAccessValid

// BankedRegisterAccessValid()
// ===========================

// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

case SYSm of
  when '000xx', '00100'                          // R8_usr to R12_usr
    if mode != M32_FIQ then UNPREDICTABLE;
  when '00101'                                   // SP_usr
    if mode == M32_System then UNPREDICTABLE;
  when '00110'                                   // LR_usr
    if mode IN {M32_Hyp, M32_System} then UNPREDICTABLE;
  when '010xx', '0110x', '01110'                 // R8_fiq to R12_fiq, SP_fiq, LR_fiq
    if mode == M32_FIQ then UNPREDICTABLE;
  when '1000x'                                   // LR_irq, SP_irq
    if mode == M32_IRQ then UNPREDICTABLE;
  when '1001x'                                   // LR_svc, SP_svc
    if mode == M32_Svc then UNPREDICTABLE;
  when '1010x'                                   // LR_abt, SP_abt
    if mode == M32_Abort then UNPREDICTABLE;
  when '1011x'                                   // LR_und, SP_und
    if mode == M32_Undef then UNPREDICTABLE;
  when '1110x'                                   // LR_mon, SP_mon
    if (!(HaveEL(EL3) || CurrentSecurityState() != SS_Secure ||
          mode == M32_Monitor)) then UNPREDICTABLE;
  when '11110'                                   // ELR_hyp, only from Monitor or Hyp mode
    if HaveEL(EL2) || mode IN {M32_Monitor, M32_Hyp}) then UNPREDICTABLE;
  when '11111'                                   // SP_hyp, only from Monitor mode
    if HaveEL(EL2) || !mode == M32_Monitor then UNPREDICTABLE;
  otherwise
    UNPREDICTABLE;
return;
aarch32/functions/system/CPSRWriteByInstr

// CPSRWriteByInstr()
// ================

CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0;              // PSTATE.<A,I,F,M> are not writable at EL0
    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask<3> == '1' then
        PSTATE.<N,Z,C,V,Q> = value<31:27>;    // Bits <26:24> are ignored
    if bytemask<2> == '1' then
        if HaveSSBSExt() then
            PSTATE.SSBS = value<23>;
        if privileged then
            PSTATE.PAN = value<22>;
        if HaveDITExt() then
            PSTATE.DIT = value<21>;
            // Bit <20> is RES0
            PSTATE.GE = value<19:16>;
    if bytemask<1> == '1' then
        // Bits <15:10> are RES0
        PSTATE.E = value<9>;                    // PSTATE.E is writable at EL0
        if privileged then
            PSTATE.A = value<8>;
    if bytemask<0> == '1' then
        if privileged then
            PSTATE.<I,F> = value<7:6>;           // Bit <5> is RES0
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(value<4:0>);
            return;

aarch32/functions/system/ConditionPassed

// ConditionPassed()
// ===============

boolean ConditionPassed()
    return ConditionHolds(AArch32.CurrentCond());

aarch32/functions/system/CurrentCond

bits(4) AArch32.CurrentCond();

aarch32/functions/system/InITBlock

// InITBlock()
// =========

boolean InITBlock()
    if CurrentInstrSet() == InstrSet_T32 then
        return PSTATE.IT<3:0> != '0000';
    else
        return FALSE;
aarch32/functions/system/LastInITBlock

// LastInITBlock()
// ===============

boolean LastInITBlock()
    return (PSTATE.IT<3:0> == '1000');

aarch32/functions/system/SPSRWriteByInstr

// SPSRWriteByInstr()
// ===============

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    bits(32) new_spsr = SPSR[];
    if bytemask<3> == '1' then
        new_spsr<31:24> = value<31:24>;  // N,Z,C,V,Q flags, IT[1:0], J bits
    if bytemask<2> == '1' then
    if bytemask<1> == '1' then
        new_spsr<15:8> = value<15:8>;    // IT[7:2] bits, E bit, A interrupt mask
    if bytemask<0> == '1' then
        new_spsr<7:0> = value<7:0>;      // I,F interrupt masks, T bit, Mode bits
    SPSR[] = new_spsr;                   // UNPREDICTABLE if User or System mode
    return;

aarch32/functions/system/SPSRaccessValid

// SPSRaccessValid()
// ===============

// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

SPSRaccessValid(bits(5) SYSm, bits(5) mode)
    case SYSm of
        when '01110'  // SPSR_fiq
            if mode == M32_FIQ then UNPREDICTABLE;
        when '10000'  // SPSR_irq
            if mode == M32_IRQ then UNPREDICTABLE;
        when '10010'  // SPSR_svc
            if mode == M32_Svc then UNPREDICTABLE;
        when '10100'  // SPSR_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '10110'  // SPSR_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '11100'  // SPSR_mon
            if (!HaveEL(EL3) || mode == M32_Monitor ||
            CurrentSecurityState() != SS_Secure) then UNPREDICTABLE;
        when '11110'  // SPSR_hyp
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
            otherwise
                UNPREDICTABLE;
    return;
aarch32/functions/system/SelectInstrSet

// SelectInstrSet()
// ================

SelectInstrSet(InstrSet iset)
    assert CurrentInstrSet() IN {InstrSet_A32, InstrSet_T32};
    assert iset IN {InstrSet_A32, InstrSet_T32};
    PSTATE.T = if iset == InstrSet_A32 then '0' else '1';
    return;

aarch32/functions/v6simd/Sat

// Sat()
// ======

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;

aarch32/functions/v6simd/SignedSat

// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;

aarch32/functions/v6simd/UnsignedSat

// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;

J1.2.4 aarch32/translation

This section includes the following pseudocode functions:
- aarch32/translation/faults/AArch32.S1HasAlignmentFault on page J1-11326.
- aarch32/translation/faults/AArch32.S1LDHasPermissionsFault on page J1-11326.
- aarch32/translation/faults/AArch32.S1SDHasPermissionsFault on page J1-11326.
aarch32/translation/attrs/AArch32.DefaultTEXDecode

// AArch32.DefaultTEXDecode()
// ==========================
// Apply short-descriptor format memory region attributes, without TEX remap

MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX_in, bit C_in, bit B_in, bit s)
MemoryAttributes memattrs;
bits(3) TEX = TEX_in;
bit C = C_in;
bit B = B_in;

// Reserved values map to allocated values
if (TEX == '001' & C:B == '01') || (TEX == '010' & C:B != '00') || TEX == '011' then
bits(5) texcb;
(-, texcb) = ConstrainUnpredictableBits(Unpredictable_RESTEXCB, 5);
TEX = texcb<4:2>; C = texcb<1>; B = texcb<0>;

// Distinction between Inner Shareable and Outer Shareable is not supported in this format
// A memory region is either Non-shareable or Outer Shareable

case TEX:C:B of
  when '00000'
    // Device-nGnRnE
    memattrs.memtype = MemType_Device;
    memattrs.device = DeviceType_nGnRnE;
    memattrs.shareability = Shareability_OSH;
  when '00001', '01000'
    // Device-nGnRE
    memattrs.memtype = MemType_Device;
    memattrs.device = DeviceType_nGnRE;
    memattrs.shareability = Shareability_OSH;
  when '00010'
    // Write-through Read allocate
    memattrs.memtype = MemType_Normal;
    memattrs.inner.attrs = MemAttr_WT;
    memattrs.inner.hints = MemHint_RA;
    memattrs.outer.attrs = MemAttr_WT;
    memattrs.outer.hints = MemHint_RA;
    memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
  when '00011'
    // Write-back Read allocate
    memattrs.memtype = MemType_Normal;
    memattrs.inner.attrs = MemAttr_WB;
    memattrs.inner.hints = MemHint_RA;
    memattrs.outer.attrs = MemAttr_WB;
    memattrs.outer.hints = MemHint_RA;
    memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
  when '00100'
    // Non-cacheable
    memattrs.memtype = MemType_Normal;
    memattrs.inner.attrs = MemAttr_NC;
    memattrs.outer.attrs = MemAttr_NC;
    memattrs.shareability = Shareability_OSH;
  when '00110'
    memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
  when '00111'
    // Write-back Read and Write allocate
    memattrs.memtype = MemType_Normal;
    memattrs.inner.attrs = MemAttr_WB;
    memattrs.inner.hints = MemHint_RWA;
    memattrs.outer.attrs = MemAttr_WB;
    memattrs.outer.hints = MemHint_RWA;
    memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
  when '1xxxx'
    // Cacheable, TEX<1:0> = Outer attrs, {C,B} = Inner attrs
    memattrs.memtype = MemType_Normal;
    memattrs.inner = DecodeSDFAttr(C:B);
    memattrs.outer = DecodeSDFAttr(TEX<1:0>);
    if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
      memattrs.shareability = Shareability_OSH;
    else
      memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
    otherwise
      memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
      // Reserved, handled above
      Unreachable();
  endcase

// The Transient hint is not supported in this format
memattrs.inner.transient = FALSE;
memattrs.outer.transient = FALSE;
memattrs.tagged = FALSE;
if memattrs.inner.attrs == MemAttr_WB && memattrs.outer.attrs == MemAttr_WB then
  memattrs.xs = '0';
else
  memattrs.xs = '1';
return memattrs;

aarch32/translation/attrs/AArch32.RemappedTEXDecode

// AArch32.RemappedTEXDecode()
// ===================================
// Apply short-descriptor format memory region attributes, with TEX remap

MemoryAttributes AArch32.RemappedTEXDecode(Regime regime, bits(3) TEX, bit C, bit B, bit s)

MemoryAttributes memattrs;
PRRR_Type prrr;
NMRR_Type nmrr;

region = UInt(TEX<0>:C:B);       // TEX<2:1> are ignored in this mapping scheme
if region == 6 then
    return MemoryAttributes IMPLEMENTATION_DEFINED;
if regime == Regime_EL30 then
    prrr = PRRR_S;
    nmrr = NMRR_S;
elsif HaveAArch32EL(EL3) then
    prrr = PRRR_NS;
    nmrr = NMRR_NS;
else
    prrr = PRRR;
    nmrr = NMRR;

base = 2 * region;
attrfield = prrr<base+1:base>;
if attrfield == '11' then      // Reserved, maps to allocated value
    Unreachable();
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tagged = FALSE;
else
    case attrfield of
        when '00'                  // Device-nGnRnE
            memattrs.memtype = MemType_Device;
            memattrs.device = DeviceType_nGnRnE;
            memattrs.shareability = Shareability_OSH;
        when '01'                  // Device-nGnRE
            memattrs.memtype = MemType_Device;
            memattrs.device = DeviceType_nGnRE;
            memattrs.shareability = Shareability_OSH;
        when '10'
            NSn = if s == '0' then prrr.NS0 else prrr.NS1;
            NOSm = prrr<region+24> AND NSn;
            IRn = nmrr<base+1:base>;
            ORn = nmrr<base+17:base+16>;
            memattrs.memtype = MemType_Normal;
            memattrs.inner = DecodeSDFAttr(IRn);
            memattrs.outer = DecodeSDFAttr(ORn);
            if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
                memattrs.shareability = Shareability_OSH;
            else
                bits(2) sh = NSn:NOSm;
                memattrs.shareability = DecodeShareability(sh);
            end when
        when '11'
            Unreachable();

        // The Transient hint is not supported in this format
        memattrs.inner.transient = FALSE;
        memattrs.outer.transient = FALSE;
        memattrs.tagged = FALSE;
if memattrs.inner.attrs == MemAttr_WB &
memattrs.outer.attrs == MemAttr_WB then
    memattrs.xs = '0';
else
    memattrs.xs = '1';
return memattrs;

aarch32/translation/debug/AArch32.CheckBreakpoint

// AArch32.CheckBreakpoint()
// -------------------------------
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch32.CheckBreakpoint(bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert size IN {2,4};
    match = FALSE;
    mismatch = FALSE;
    for i = 0 to NumBreakpointsImplemented() - 1
        (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, size);
        match = match || match_i;
        mismatch = mismatch || mismatch_i;
    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elsif (match || mismatch) then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException_Breakpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return NoFault();

aarch32/translation/debug/AArch32.CheckDebug

// AArch32.CheckDebug()
// ----------------------
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)
    FaultRecord fault = NoFault();
    boolean d_side = (IsDataAccess(acctype) || acctype == AccType_DC);
    boolean i_side = (acctype == AccType_IFETCH);
    generate_exception = AArch32.GenerateDebugExceptions() &
DBGDSCRext.MDBGen == '1';
    halt = HaltOnBreakpointOrWatchpoint();
    // Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
    vector_catch_first = ConstrainUnpredictableBool(Unpredictable_BPVECTORCATCHPRI);
    if i_side && vector_catch_first && generate_exception then
        fault = AArch32.CheckVectorCatch(vaddress, size);
    if fault.statuscode == Fault_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32.CheckWatchpoint(vaddress, acctype, iswrite, size);
        elseif i_side then
            fault = AArch32.CheckBreakpoint(vaddress, size);
        else
            return NoFault();

if memattrs.inner.attrs == MemAttr_WB &
memattrs.outer.attrs == MemAttr_WB then
    memattrs.xs = '0';
else
    memattrs.xs = '1';
return memattrs;
if fault.statuscode == Fault_None && i_side && !vector_catch_first && generate_exception then
    return AArch32.CheckVectorCatch(vaddress, size);

return fault;

aarch32/translation/debug/AArch32.CheckVectorCatch

// AArch32.CheckVectorCatch()
// ==========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when debug exceptions are enabled.
FaultRecord AArch32.CheckVectorCatch(bits(32) vaddress, integer size)
assert ELUsingAArch32(S1TranslationRegime());
macth = AArch32.VCRMatch(vaddress);
if size == 4 && !match && AArch32.VCRMatch(vaddress + 2) then
match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);
if match then
    acctype = AccType_IFETCH;
iswrite = FALSE;
d debugmoe = DebugException_VectorCatch;
    return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
    return NoFault();

aarch32/translation/debug/AArch32.CheckWatchpoint

// AArch32.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.
FaultRecord AArch32.CheckWatchpoint(bits(32) vaddress, AccType acctype,
boolean iswrite, integer size)
assert ELUsingAArch32(S1TranslationRegime());
if acctype == AccType_DC then
    if !iswrite then
        return NoFault();
    elsif !(boolean IMPLEMENTATION_DEFINED "DCIMVAC generates watchpoint") then
        return NoFault();
    elsif !IsDataAccess(acctype) then
        return NoFault();
else
    ispriv = AArch32.AccessUsesEL(acctype) != EL0;
    for i = 0 to NumWatchpointsImplemented() - 1
        if AArch32.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite) then
            match = TRUE;
    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        EDWAR = ZeroExtend(vaddress, 64);
        Halt(reason);
    elsif match then
        debugmoe = DebugException_Watchpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return NoFault();
aarch32/translation/faults/AArch32.DebugFault

// AArch32.DebugFault()
// ====================
// Return a fault record indicating a hardware watchpoint/breakpoint

FaultRecord AArch32.DebugFault(AccType acctype, boolean iswrite, bits(4) debugmoe)
    FaultRecord fault;
    fault.getStatusCode = Fault_Debug;
    fault.acctype = acctype;
    fault.write = iswrite;
    fault.debugmoe = debugmoe;
    fault.secondstage = FALSE;
    fault.s2fs1walk = FALSE;
    return fault;

aarch32/translation/faults/AArch32.IPAIsOutOfRange

// AArch32.IPAIsOutOfRange()
// =========================
// Check intermediate physical address bits not resolved by translation are ZERO

boolean AArch32.IPAIsOutOfRange(S2TTWParams walkparams, bits(40) ipa)
    // Input Address size
    iasize = AArch32.S2IASize(walkparams.t0sz);
    return iasize < 40 && !IsZero(ipa<39:iasize>);

aarch32/translation/faults/AArch32.S1HasAlignmentFault

// AArch32.S1HasAlignmentFault()
// =============================
// Returns whether stage 1 output fails alignment requirement on data accesses
to Device memory

boolean AArch32.S1HasAlignmentFault(AccType acctype, boolean aligned, bit ntlsmd, MemoryAttributes memattrs)
    if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
        return FALSE;
    if acctype == AccType_A32LSMD && ntlsmd == '0' && memattrs.device != DeviceType_GRE then
        return TRUE;
    return !aligned || acctype == AccType_DCZVA;

aarch32/translation/faults/AArch32.S1LDHasPermissionsFault

// AArch32.S1LDHasPermissionsFault()
// =================================
// Returns whether an access using stage 1 long-descriptor translation
// violates permissions of target memory

boolean AArch32.S1LDHasPermissionsFault(Regime regime, SecurityState ss, S1TTWParams walkparams, Permissions perms, MemType memtype, PASpace paspace, boolean ispriv, AccType acctype, boolean iswrite)
    bit r;
    bit w;
    bit x;
    bit pr;
    bit pw;
    bit ur;
bit uw;
bit xn;
if HasUnprivileged(regime) then
    // Apply leaf permissions
    case perms.ap<2:1> of
        when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
        when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
        when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
        when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
    endcase; // Apply leaf permissions
    // Apply hierarchical permissions
    case perms.ap_table of
        when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
        when '01' (pr,pw,ur,uw) = ( pr, pw,'0','0'); // Privileged access
        when '10' (pr,pw,ur,uw) = ( pr,'0', ur,'0'); // Read-only
        when '11' (pr,pw,ur,uw) = ( pr,'0','0','0'); // Read-only, privileged access
    endcase; // Apply hierarchical permissions

    xn   = perms.xn OR perms.xn_table;
    pxn  = perms.pxn OR perms.pxn_table;

    ux = ur AND NOT(xn OR (uw AND walkparams.wxn));
    px = pr AND NOT(xn OR pxn OR (pw AND walkparams.wxn) OR (uw AND walkparams.uwxn));

    pan_access = !(acctype IN {AccType_DC, AccType_IFETCH, AccType_AT});
    if HavePANExt() && pan_access then
        pan = PSTATE.PAN AND (ur OR uw);
        pr  = pr AND NOT(pan);
        pw  = pw AND NOT(pan);

        (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);
    endcase; // Apply leaf permissions
    // Prevent execution from Non-secure space by PE in Secure state if SIF is set
    if ss == SS_Secure && paspace == PAS_NonSecure then
        x = x AND NOT(walkparams.sif);
    else
    // Apply leaf permissions
        case perms.ap<2> of
            when '0' (r,w) = ('1','1'); // No effect
            when '1' (r,w) = ('1','0'); // Read-only
        endcase; // Apply leaf permissions
    endcase; // Apply leaf permissions

    if acctype == AccType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memtype == MemType_Device then
            return TRUE;
        else
            return x == '0';
        endif;
    elseif acctype IN {AccType_IC, AccType_DC} then
        return FALSE;
    elseif iswrite then
        return w == '0';
    else
        return r == '0';
    endif;

aarch32/translation/faults/AArch32.S1SDHasPermissionsFault

    // AArch32.S1SDHasPermissionsFault()
    // =================================
    // Returns whether an access using stage 1 short-descriptor translation
// violates permissions of target memory

```c
boolean AArch32.SISDHasPermissionsFault(Regime regime, SecurityState ss, Permissions perms_in,
   MemType memtype, PASpace paspace, boolean ispriv,
   AccType acctype, boolean iswrite)

   Permissions perms = perms_in;
   bit pr;
   bit pw;
   bit ur;
   bit uw;
   SCTLR_Type sctlr;
   if regime == Regime_EL30 then
        sctlr = SCTLR_S;
    elsif HaveAArch32EL(EL3) then
        sctlr = SCTLR_NS;
    else
        sctlr = SCTLR;
    end if
    if sctlr.AFE == '0' then
        // Map Reserved encoding '100'
        if perms.ap == '100' then
            perms.ap = bits(3) IMPLEMENTATION_DEFINED "Reserved short descriptor AP encoding";
        end if
    end if
    case perms.ap of
        when '000' (pr,pw,ur,uw) = ('0','0','0','0'); // No access
        when '001' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
        when '010' (pr,pw,ur,uw) = ('1','1','1','0'); // R/W at PL1, RO at PL0
        when '011' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
        when '100' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
        when '101' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL (deprecated)
        when '110' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
        when '111' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
    end case
    else // Simplified access permissions model
    case perms.ap<2:1> of
        when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
        when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
        when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
        when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
    end case
    ux = ur AND NOT(perms.xn OR (uw AND sctlr.WXN));
    px = pr AND NOT(perms.xn OR perms.pxn OR (pw AND sctlr.WXN) OR (uw AND sctlr.UWXN));
    pan_access = !(acctype IN {AccType_DC, AccType_IFETCH, AccType_AT});
    if HavePANExt() && pan_access then
        pan = PSTATE.PAN AND (ur OR uw);
        pr  = pr AND NOT(pan);
        pw  = pw AND NOT(pan);
    end if
    (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);
    // Prevent execution from Non-secure space by PE in Secure state if SIF is set
    if ss == SS_Secure && paspace == PAS_NonSecure then
        x = x AND NOT(if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF);
    end if
    if acctype == AccType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT & & memtype == MemType_Device then
            return TRUE;
        end if
        return x == '0';
    elseif acctype IN {AccType_IC, AccType_DC} then
        return FALSE;
    elseif iswrite then
        return w == '0';
    else
        return r == '0';
    end if
```
aarch32/translation/faults/AArch32.S2HasAlignmentFault

// AArch32.S2HasAlignmentFault()
// ========================================
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory

boolean AArch32.S2HasAlignmentFault(AccType acctype, boolean aligned,
                                      MemoryAttributes memattrs)
    if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
        return FALSE;
    return !aligned || acctype == AccType_DCZVA;

aarch32/translation/faults/AArch32.S2HasPermissionsFault

// AArch32.S2HasPermissionsFault()
// ===============================
// Returns whether stage 2 access violates permissions of target memory

boolean AArch32.S2HasPermissionsFault(boolean s2fs1walk, S2TTWParams walkparams,
                                        Permissions perms, MemType memtype,
                                        boolean ispriv, AccType acctype,
                                        boolean iswrite)
    bit px;
    bit ux;
    r = perms.s2ap<0>;
    w = perms.s2ap<1>;
    bit x;
    if HaveExtendedExecuteNeverExt() then
        case perms.s2xn:perms.s2xnx of
            when '00' (px, ux) = (r, r);
            when '01' (px, ux) = ('0', r);
            when '10' (px, ux) = ('0', '0');
            when '11' (px, ux) = (r, '0');
        endcase;
        x = if ispriv then px else ux;
    else
        x = r AND NOT(perms.s2xn);
    endcase;
    if s2fs1walk && walkparams.ptw == '1' && memtype == MemType_Device then
        return TRUE;
    elsif acctype == AccType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memtype == MemType_Device then
            return TRUE;
        else
            return x == '0';
        endcase;
    elsif acctype IN {AccType_IC, AccType_DC} then
        return FALSE;
    elsif iswrite then
        return w == '0';
    else
        return r == '0';

aarch32/translation/faults/AArch32.S2InconsistentSL

// AArch32.S2InconsistentSL()
// ==========================
// Detect inconsistent configuration of stage 2 T0SZ and SL fields

boolean AArch32.S2InconsistentSL(S2TTWParams walkparams)
    startlevel = AArch32.S2StartLevel(walkparams.sl0);
    levels = FINAL_LEVEL - startlevel;
granulebits = TGxGranuleBits(walkparams.tgx);
stride = granulebits - 1;

// Input address size must at least be large enough to be resolved from the start level
sl_min_iasize = 
    levels * stride // Bits resolved by table walk, except initial level
+ granulebits // Bits directly mapped to output address
+ 1); // At least 1 more bit to be decoded by initial level

// Can accommodate 1 more stride in the level + concatenation of up to 2^4 tables
sl_max_iasize = sl_min_iasize + (stride-1) + 4;
// Configured Input Address size
iasize = AArch32.S2IASize(walkparams.t0sz);
return iasize < sl_min_iasize || iasize > sl_max_iasize;

aarch32/translation/faults/AArch32.VAIsOutOfRange

// AArch32.VAIsOutOfRange()
// ========================
// Check virtual address bits not resolved by translation are identical
// and of accepted value

boolean AArch32.VAIsOutOfRange(Regime regime, S1TTWParams walkparams, bits(32) va)
if regime == Regime_EL2 then
    // Input Address size
    iasize = AArch32.S1IASize(walkparams.t0sz);
    return walkparams.t0sz != '000' && !IsZero(va<31:iasize>);
elsif walkparams.t1sz != '000' && walkparams.t0sz != '000' then
    // Lower range Input Address size
    lo_iasize = AArch32.S1IASize(walkparams.t0sz);
    // Upper range Input Address size
    up_iasize = AArch32.S1IASize(walkparams.t1sz);
    return !IsZero(va<31:lo_iasize>) && !IsOnes(va<31:up_iasize>);
else
    return FALSE;

aarch32/translation/tlbcontext/AArch32.GetS1TLBContext

// AArch32.GetS1TLBContext()
// =========================
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch32.GetS1TLBContext(Regime regime, SecurityState ss, bits(32) va)
TLBContext tlbcontext;
case regime of
    when Regime_EL2  tlbcontext = AArch32.TLBContextEL2(va);
    when Regime_EL10 tlbcontext = AArch32.TLBContextEL10(ss, va);
    when Regime_EL30 tlbcontext = AArch32.TLBContextEL30(va);
    tlbcontext.includes_s1 = TRUE;
// The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2 = FALSE;
return tlbcontext;

aarch32/translation/tlbcontext/AArch32.GetS2TLBContext

// AArch32.GetS2TLBContext()
// =========================
// Gather translation context for accesses with IPA to match against TLB entries

TLBContext AArch32.GetS2TLBContext(FullAddress ipa)
assert ipa.paspace == PAS_NonSecure;

TLBContext tlbcontext;

tlbcontext.ss          = SS_NonSecure;
tlbcontext.regime      = Regime_EL10;
tlbcontext.ipaspace    = ipa.paspace;
tlbcontext.vmid        = ZeroExtend(VTTBR.VMID, 16);
tlbcontext.tg          = TGx_4KB;
tlbcontext.includes_s1 = FALSE;
tlbcontext.includes_s2 = TRUE;
tlbcontext.ia          = ZeroExtend(ipa.address, 64);
tlbcontext.cnp         = if HaveCommonNotPrivateTransExt() then VTTBR.CnP else '0';

return tlbcontext;

aarch32/translation/tlbcontext/AArch32.TLBContextEL10

// AArch32.TLBContextEL10()
// ========================
// Gather translation context for accesses under EL10 regime
// (PL10 when EL3 is A64) to match against TLB entries

TLBContext AArch32.TLBContextEL10(SecurityState ss, bits(32) va)

TLBContext tlbcontext;
TTBCR_Type ttbcr;
TTBR0_Type ttbr0;
TTBR1_Type ttbr1;
CONTEXTIDR_Type contextidr;

if HaveAArch32EL(EL3) then
  ttbcr      = TTBCR_NS;
  ttbr0      = TTBR0_NS;
  ttbr1      = TTBR1_NS;
  contextidr = CONTEXTIDR_NS;
else
  ttbcr      = TTBCR;
  ttbr0      = TTBR0;
  ttbr1      = TTBR1;
  contextidr = CONTEXTIDR;

tlbcontext.ss     = ss;
tlbcontext.regime = Regime_EL10;

if AArch32.EL2Enabled(ss) then
  tlbcontext.vmid = ZeroExtend(VTTBR.VMID, 16);

if ttbcr.EAE == '1' then
  tlbcontext.asid = ZeroExtend(if ttbcr.A1 == '0' then ttbr0.ASID else ttbr1.ASID, 16);
else
  tlbcontext.asid = ZeroExtend(contextidr.ASID, 16);

tlbcontext.tg = TGx_4KB;
tlbcontext.ia = ZeroExtend(va, 64);

if HaveCommonNotPrivateTransExt() && ttbcr.EAE == '1' then
  if AArch32.GetVARange(va, ttbcr.T0SZ, ttbcr.T1SZ) == VARange_LOWER then
    tlbcontext.cnp = ttbr0.CnP;
  else
    tlbcontext.cnp = ttbr1.CnP;
else
  tlbcontext.cnp = '0';

return tlbcontext;
aarch32/translation/tlbcontext/AArch32.TLBContextEL2

// AArch32.TLBContextEL2()
// =======================
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch32.TLBContextEL2(bits(32) va)
                      TLBContext tlbcontext;
               tlbcontext.ss     = SS_NonSecure;
               tlbcontext.regime = Regime_EL2;
               tlbcontext.ia     = ZeroExtend(va, 64);
               tlbcontext.tg     = TGx_4KB;
               tlbcontext.cnp    = if HaveCommonNotPrivateTransExt() then HTTBR.CnP else '0';

               return tlbcontext;

aarch32/translation/tlbcontext/AArch32.TLBContextEL30

// AArch32.TLBContextEL30()
// ========================
// Gather translation context for accesses under EL30 regime
// (PL10 in Secure state and EL3 is A32) to match against TLB entries

TLBContext AArch32.TLBContextEL30(bits(32) va)
                      TLBContext tlbcontext;
               tlbcontext.ss     = SS_Secure;
               tlbcontext.regime = Regime_EL30;

               if TTBCR_S.EAE == '1' then
               tlbcontext.asid = ZeroExtend(if TTBCR_S.A1 == '0' then TTBR0_S.ASID else TTBR1_S.ASID, 16);
               else
               tlbcontext.asid = ZeroExtend(CONTEXTIDR_S.ASID, 16);
               tlbcontext.tg = TGx_4KB;
               tlbcontext.ia = ZeroExtend(va, 64);

               if HaveCommonNotPrivateTransExt() && TTBCR_S.EAE == '1' then
               if AArch32.GetVARange(va, TTBCR_S.T0SZ, TTBCR_S.T1SZ) == VARange_LOWER then
               tlbcontext.cnp = TTBR0_S.CnP;
               else
               tlbcontext.cnp = TTBR1_S.CnP;
               else
               tlbcontext.cnp = '0';
               return tlbcontext;

aarch32/translation/translation/AArch32.AccessUsesEL

// AArch32.AccessUsesEL()
// ======================
// Determine the privilege associated with the access

bits(2) AArch32.AccessUsesEL(AccType acctype)
               if acctype == AccType_UNPRIV then
               return EL0;
               else
               return PSTATE.EL;
// AArch32.EL2Enabled()
// ====================
// Returns whether EL2 is enabled for the given Security State

boolean AArch32.EL2Enabled(SecurityState ss)
if ss == SS_Secure then
  if !(HaveEL(EL2) && HaveSecureEL2Ext()) then
    return FALSE;
  elsif HaveEL(EL3) then
    return SCR_EL3.EEL2 == '1';
  else
    return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
  else
    return HaveEL(EL2);

// AArch32.FullTranslate()
// =======================
// Perform address translation as specified by VMSA-A32

AddressDescriptor AArch32.FullTranslate(bits(32) va, AccType acctype, boolean iswrite, boolean aligned)

// Prepare fault fields in case a fault is detected
fault = NoFault();
fault.acctype = acctype;
fault.write = iswrite;

regime = TranslationRegime(PSTATE.EL, acctype);
ispriv = PSTATE.EL != EL0 && acctype != AccType_UNPRIV;
ss = SecurityStateForRegime(regime);

// First Stage Translation
AddressDescriptor ipa;
if regime == Regime_EL2 || TTBCR.EAE == '1' then
  (fault, ipa) = AArch32.S1TranslateLD(fault, regime, ss, va, acctype, aligned, iswrite, ispriv);
else
  (fault, ipa, -) = AArch32.S1TranslateSD(fault, regime, ss, va, acctype, aligned, iswrite, ispriv);

if fault.statuscode != Fault_None then
  return CreateFaultyAddressDescriptor(ZeroExtend(va, 64), fault);
if regime == Regime_EL10 && EL2Enabled() then
  ipa.vaddress = ZeroExtend(va, 64);
s2fs1walk = FALSE;
AddressDescriptor pa;
(fault, pa) = AArch32.S2Translate(fault, ipa, ss, s2fs1walk, acctype, aligned, iswrite, ispriv);

if fault.statuscode != Fault_None then
  return CreateFaultyAddressDescriptor(ZeroExtend(va, 64), fault);
else
  return pa;
else
  return ipa;
aarch32/translation/translation/AArch32.OutputDomain

// AArch32.OutputDomain()
// ======================
// Determine the domain the translated output address

bits(2) AArch32.OutputDomain(Regime regime, bits(4) domain)
bits(2) Dn;
index = 2 * UInt(domain);
if regime == Regime_EL30 then
  Dn = DACR_S<index+1:index>;
elsif HaveAArch32EL(EL3) then
  Dn = DACR_NS<index+1:index>;
else
  Dn = DACR<index+1:index>;
if Dn == '10' then
  // Reserved value maps to an allocated value
  (-, Dn) = ConstrainUnpredictableBits(Unpredictable_RESDACR, 2);
return Dn;

aarch32/translation/translation/AArch32.S1DisabledOutput

// AArch32.S1DisabledOutput()
// ==========================
// Flat map the VA to IPA/PA, depending on the regime, assigning default memory attributes

(FaultRecord, AddressDescriptor) AArch32.S1DisabledOutput(FaultRecord fault_in, Regime regime,
SecurityState ss, bits(32) va,
AccType acctype, boolean aligned)

FaultRecord fault = fault_in;
// No memory page is guarded when stage 1 address translation is disabled
SetInGuardedPage(FALSE);

MemoryAttributes memattrs;
bit default_cacheable;
if regime == Regime_EL10 && AArch32.EL2Enabled(ss) then
  if ELStateUsingAArch32(EL2, ss == SS_Secure) then
    default_cacheable = HCR.DC;
  else
    default_cacheable = HCR_EL2.DC;
else
  default_cacheable = '0';
if default_cacheable == '1' then
  // Use default cacheable settings
  memattrs.memtype      = MemType_Normal;
  memattrs.inner.attrs  = MemAttr_WB;
  memattrs.inner.hints  = MemHint_RWA;
  memattrs.outer.attrs  = MemAttr_WB;
  memattrs.outer.hints  = MemHint_RWA;
  memattrs.shareability = Shareability_NSH;
  if !ELStateUsingAArch32(EL2, ss == SS_Secure) && HaveMTE2Ext() then
    memattrs.tagged = HCR_EL2.DCT == '1';
  else
    memattrs.tagged = FALSE;
elsif acctype == AccType_IFETCH then
  memattrs.memtype      = MemType_Normal;
  memattrs.shareability = Shareability_OSH;
  memattrs.tagged = FALSE;
if AArch32.S1ICacheEnabled(regime) then
  memattrs.inner.attrs = MemAttr_WT;
  memattrs.inner.hints = MemHint_RA;
  memattrs.outer.attrs = MemAttr_WT;
memattrs.outer.hints = MemHint_RA;
else
  memattrs.inner.attrs = MemAttr_NC;
  memattrs.outer.attrs = MemAttr_NC;
else
  // Treat memory region as Device
  memattrs.memtype = MemType_Device;
  memattrs.device = DeviceType_nGnRnE;
  memattrs.shareability = Shareability_OSH;
  memattrs.tagged = FALSE;

bit ntlsmd;
if HaveTrapLoadStoreMultipleDeviceExt() then
case regime of
  when Regime_EL30 ntlsmd = SCTLR_S.nTLSMD;
  when Regime_EL2 ntlsmd = HSCTLR.nTLSMD;
  when Regime_EL10 ntlsmd = if HaveAArch32EL(EL3) then SCTLR_NS.nTLSMD else SCTLR.nTLSMD;
else
  ntlsmd = '1';
if AArch32.S1HasAlignmentFault(acctype, aligned, ntlsmd, memattrs) then
  fault.statuscode = Fault_Alignment;
  return (fault, AddressDescriptor UNKNOWN);

FullAddress oa;
oa.address = ZeroExtend(va, 52);
oa.paspace = if ss == SS_Secure then PAS_Secure else PAS_NonSecure;
ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);
return (fault, ipa);

aarch32/translation/translation/AArch32.S1Enabled

// AArch32.S1Enabled()
// ================
// Returns whether stage 1 translation is enabled for the active translation regime
boolean AArch32.S1Enabled(Regime regime, SecurityState ss)
if regime == Regime_EL2 then
  return HSCTLR.M == '1';
elsif regime == Regime_EL30 then
  return SCTLR_S.M == '1';
elsif !AArch32.EL2Enabled(ss) then
  return (if HaveAArch32EL(EL3) then SCTLR_NS.M else SCTLR.M) == '1';
elsif ELStateUsingAArch32(EL2, ss == SS_Secure) then
  return HCR.<TGE,DC> == '00' && (if HaveAArch32EL(EL3) then SCTLR_NS.M else SCTLR.M) == '1';
else
  return HCR_EL2.<TGE,DC> == '00' && SCTLR.M == '1';

aarch32/translation/translation/AArch32.S1TranslateLD

// AArch32.S1TranslateLD()
// =======================
// Perform a stage 1 translation using long-descriptor format mapping VA to IPA/PA
// depending on the regime
(FaultRecord, AddressDescriptor) AArch32.S1TranslateLD(FaultRecord fault_in, Regime regime,
SecurityState ss, bits(32) va,
AccType acctype, boolean aligned,
boolean iswrite, boolean ispriv)
if !AArch32.S1Enabled(regime, ss) then
    return AArch32.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);

walkparams = AArch32.GetS1TTWParams(regime, va);

if AArch32.VAIsOutOfRange(regime, walkparams, va) then
    fault.level = 1;
    fault.statuscode = Fault_Translation;
    return (fault, AddressDescriptor UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S1WalkLD(fault, regime, ss, walkparams, va, ispriv);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

SetInGuardedPage(FALSE);  // AArch32-VMSA does not guard any pages

if AArch32.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmd, walkstate.memattrs) then
    fault.statuscode = Fault_Alignment;
elsif IsAtomicRW(acctype) then
    if AArch32.S1LDHasPermissionsFault(regime, ss, walkparams, walkstate.permissions, walkstate.memattrs.memtype, walkstate.baseaddress.paspace, ispriv, acctype, FALSE) then
        // The Permission fault was not caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write = FALSE;
    elsif AArch32.S1LDHasPermissionsFault(regime, ss, walkparams, walkstate.permissions, walkstate.memattrs.memtype, walkstate.baseaddress.paspace, ispriv, acctype, TRUE) then
        // The Permission fault was caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write = TRUE;
    elsif AArch32.S1LDHasPermissionsFault(regime, ss, walkparams, walkstate.permissions, walkstate.memattrs.memtype, walkstate.baseaddress.paspace, ispriv, acctype, iswrite) then
        fault.statuscode = Fault_Permission;
    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

MemoryAttributes memattrs;
if ((acctype == AccType_IFETCH &&
    (walkstate.memattrs.memtype == MemType_Device || !AArch32.S1ICacheEnabled(regime))) ||
    (acctype != AccType_IFETCH &&
    walkstate.memattrs.memtype == MemType_Normal && !AArch32.S1DCacheEnabled(regime))) then
    // Treat memory attributes as Normal Non-Cacheable
    memattrs = NormalNCMemAttr();
    memattrs.xs = walkstate.memattrs.xs;
else
    memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32.EL2Enabled(ss)) &&
    (if ELStateUsingAArch32(EL2, ss == SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
    (if !Boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1") then
    memattrs.shareability = walkstate.memattrs.shareability;
else
    memattrs.shareability = EffectiveShareability(memattrs);

// Output Address
oa = StageOA(ZeroExtend(va, 64), walkparams.tgx, walkstate);
ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);

return (fault, ipa);

aarch32/translation/translation/AArch32.S1TranslateSD

// AArch32.S1TranslateSD()
// ===============
// Perform a stage 1 translation using short-descriptor format mapping VA to IPA/PA
// depending on the regime

(FaultRecord, AddressDescriptor, SDFType) AArch32.S1TranslateSD(FaultRecord fault_in, Regime regime,
SecurityState ss, bits(32) va,
AccType acctype, boolean aligned,
boolean iswrite, boolean ispriv)

FaultRecord fault = fault_in;
fault.secondstage = FALSE;
fault.s2fs1walk = FALSE;

if !AArch32.S1Enabled(regime, ss) then
    AddressDescriptor ipa;
    (fault, ipa) = AArch32.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);
    return (fault, ipa, SDFType UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S1WalkSD(fault, regime, ss, va, ispriv);
if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN, SDFType UNKNOWN);

domain = AArch32.OutputDomain(regime, walkstate.domain);
SetInGuardedPage(FALSE);  // AArch32-VMSA does not guard any pages

bit ntlsmd;
if HaveTrapLoadStoreMultipleDeviceExt() then
    case regime of
        when Regime_EL3 then ntlsmd = SCTLR_S.nTLSMD;
        when Regime_EL1 then ntlsmd = if HaveAArch32EL(EL3) then SCTLR_NS.nTLSMD else SCTLR.nTLSMD;
        else
            ntlsmd = '1';

if AArch32.S1HasAlignmentFault(acctype, aligned, ntlsmd, walkstate.memattrs) then
    fault.statuscode = Fault_Alignment;
elsif !acctype IN {AccType_IC, AccType_DC} && domain == Domain_NoAccess then
    fault.statuscode = Fault_Domain;
elsif domain == Domain_Client then
    if IsAtomicRW(acctype) then
        if AArch32.S1SDHasPermissionsFault(regime, ss, walkstate.permissions,
            walkstate.memattrs.memtype,
            walkstate.baseaddress.paspace,
            ispriv, acctype, FALSE) then
            // The Permission fault was not caused by lack of write permissions
            fault.statuscode = Fault_Permission;
            fault.write = FALSE;
        elseif AArch32.S1SDHasPermissionsFault(regime, ss, walkstate.permissions,
            walkstate.memattrs.memtype,
            walkstate.baseaddress.paspace,
            ispriv, acctype, TRUE) then
            // The Permission fault was caused by lack of write permissions
            fault.statuscode = Fault_Permission;
            fault.write = TRUE;
        elseif AArch32.S1SDHasPermissionsFault(regime, ss, walkstate.permissions,
            walkstate.memattrs.memtype,
            walkstate.baseaddress.paspace,
ispriv, acctype, iswrite) then
    fault.statuscode = Fault_Permission;

if fault.statuscode != Fault_None then
    fault.domain = walkstate.domain;
    return (fault, AddressDescriptor UNKNOWN, walkstate.sdftype);

MemoryAttributes memattrs;
if ((acctype == AccType_IFETCH &&
    (walkstate.memattrs.memtype == MemType_Device || !AArch32.S1ICacheEnabled(regime))) ||
    (acctype != AccType_IFETCH &&
    walkstate.memattrs.memtype == MemType_Normal && !AArch32.S1DCacheEnabled(regime))) then
    // Treat memory attributes as Normal Non-Cacheable
    memattrs = NormalNCMemAttr();
    memattrs.xs = walkstate.memattrs.xs;
else
    memattrs = walkstate.memattrs;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && AArch32.EL2Enabled(ss) &&
        (if ELStateUsingAArch32(EL2, ss == SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
        (!boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1") then
            memattrs.shareability = walkstate.memattrs.shareability;
        else
            memattrs.shareability = EffectiveShareability(memattrs);

    // Output Address
    oa = AArch32.SDStageOA(walkstate.baseaddress, va, walkstate.sdftype);
    ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);
    return (fault, ipa, walkstate.sdftype);

aarch32/translation/translation/AArch32.S2Translate

// AArch32.S2Translate()
// =====================
// Perform a stage 2 translation mapping an IPA to a PA

(FaultRecord, AddressDescriptor) AArch32.S2Translate(FaultRecord fault_in, AddressDescriptor ipa,
SecurityState ss, boolean s2fs1walk,
AccType acctype, boolean aligned,
boolean iswrite, boolean ispriv)

FaultRecord fault = fault_in;
assert IsZero(ipa.paddress.address<51:40>);

if !ELStateUsingAArch32(EL2, ss == SS_Secure) then
    s1aarch64 = FALSE;
    return AArch64.S2Translate(fault, ipa, s1aarch64, ss, s2fs1walk, acctype, aligned, iswrite, ispriv);

    // Prepare fault fields in case a fault is detected
    fault.statuscode = Fault_None;
fault.secondstage = TRUE;
fault.ipaddress = ipa.paddress;
walkparams = AArch32.GetS2TTWParams();

if walkparams.vm == '0' then
    // Stage 2 is disabled
    return (fault, ipa);

if AArch32.IPAIsOutOfRange(walkparams, ipa.paddress.address<39:0>) then
    fault.statuscode = Fault_Translation;
fault.level = 1;
return (fault, AddressDescriptor UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S2Walk(fault, walkparams, ipa);

if fault.statuscode != Fault_None then
  return (fault, AddressDescriptor UNKNOWN);
if AArch32.S2HasAlignmentFault(acctype, aligned, walkstate.memattrs) then
  fault.statuscode = Fault_Alignment;
elsif IsAtomicRW(acctype) then
  assert !s2fs1walk; // AArch32 does not support HW update of TT
if AArch32.S2HasPermissionsFault(s2fs1walk, walkparams,
  walkstate.permissions,
  walkstate.memattrs.memtype,
  ispriv, acctype, FALSE) then
  // The Permission fault was not caused by lack of write permissions
  fault.statuscode = Fault_Permission;
  fault.write = FALSE;
elsif AArch32.S2HasPermissionsFault(s2fs1walk, walkparams,
  walkstate.permissions,
  walkstate.memattrs.memtype,
  ispriv, acctype, TRUE) then
  // The Permission fault was caused by lack of write permissions
  fault.statuscode = Fault_Permission;
  fault.write = TRUE;
elsif AArch32.S2HasPermissionsFault(s2fs1walk, walkparams,
  walkstate.permissions,
  walkstate.memattrs.memtype,
  ispriv, acctype, iswrite) then
  fault.statuscode = Fault_Permission;

MemoryAttributes s2_memattrs;
if ((s2fs1walk &&
    walkstate.memattrs.memtype == MemType_Device) ||
  (acctype == AccType_IFETCH &&
    walkstate.memattrs.memtype == MemType_Device || HCR2.ID == '1')) ||
  (acctype != AccType_IFETCH &&
    walkstate.memattrs.memtype == MemType_Normal && HCR2.CD == '1')) then
  // Treat memory attributes as Normal Non-Cacheable
  s2_memattrs = NormalNCMemAttr();
  s2_memattrs.xs = walkstate.memattrs.xs;
else
  s2_memattrs = walkstate.memattrs;

memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs);
ipa_64 = ZeroExtend(ipa.paddress.address<39:0>, 64);
// Output Address
oa = StageOA(ipa_64, walkparams.tgx, walkstate);
pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);
return (fault, pa);

aarch32/translation/translation/AArch32.SDStageOA

// AArch32.SDStageOA()
// ================
// Given the final walk state of a short-descriptor translation walk,
// map the untranslated input address bits to the base output address
FullAddress AArch32.SDStageOA(FullAddress baseaddress, bits(32) va, SDType sdftype)
integer tsize;

case sdftype of
  when SDType_SmallPage   tsize = 12;
  when SDType_LargePage   tsize = 16;
  when SDType_Section     tsize = 20;
when SDType_Supersection tsize = 24;

// Output Address
FullAddress oa;
oa.address = baseaddress.address<51:tsize>:va<tsize-1:0>;
oa.paspace = baseaddress.paspace;
return oa;

aarch32/translation/translation/AArch32.TranslateAddress

// AArch32.TranslateAddress()
// ================
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) va, AccType acctype,
    boolean iswrite, boolean aligned,
    integer size)

    regime = TranslationRegime(PSTATE.EL, acctype);
    if !RegimeUsingAArch32(regime) then
        return AArch64.TranslateAddress(ZeroExtend(va, 64), acctype, iswrite,
            aligned, size);
    result = AArch32.FullTranslate(va, acctype, iswrite, aligned);
    if !IsFault(result) then
        result.fault = AArch32.CheckDebug(va, acctype, iswrite, size);
    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(va, 64);
     return result;

aarch32/translation/walk/AArch32.DecodeDescriptorTypeLD

// AArch32.DecodeDescriptorTypeLD()
// ============
// Determine whether the long-descriptor is a page, block or table

DescriptorType AArch32.DecodeDescriptorTypeLD(bits(64) descriptor, integer level)

    if descriptor<1:0> == '11' && level == FINAL_LEVEL then
        return DescriptorType_Page;
    elsif descriptor<1:0> == '11' then
        return DescriptorType_Table;
    elsif descriptor<18,1> == '01' && level != FINAL_LEVEL then
        return DescriptorType_Block;
    else
        return DescriptorType_Invalid;

aarch32/translation/walk/AArch32.DecodeDescriptorTypeSD

// AArch32.DecodeDescriptorTypeSD()
// ============
// Determine the type of the short-descriptor

SDFType AArch32.DecodeDescriptorTypeSD(bits(32) descriptor, integer level)

    if level == 1 && descriptor<1:0> == '01' then
        return SDType_Table;
    elsif level == 1 && descriptor<18,1> == '01' then
        return SDType_Section;
    elsif level == 1 && descriptor<18,1> == '11' then
        return SDType_Supersection;
    elsif level == 2 && descriptor<1:0> == '01' then
        return SDType_LargePage;
    elsif level == 2 && descriptor<1:0> IN {'1x'} then
        return SDType_Invalid;
return SDFType_SmallPage;
else
  return SDFType_Invalid;

aarch32/translation/walk/AArch32.S1IASize

// AArch32.S1IASize()
//=--------------------------------
// Retrieve the number of bits containing the input address for stage 1 translation

integer AArch32.S1IASize(bits(3) txsz)
  return 32 - UInt(txsz);

aarch32/translation/walk/AArch32.S1WalkLD

// AArch32.S1WalkLD()
//=-------------------
// Traverse stage 1 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S1WalkLD(FaultRecord fault_in, Regime regime, SecurityState ss, S1TTWParams walkparams, bits(32) va, boolean ispriv)

  FaultRecord fault = fault_in;
  bits(3) txsz;
  bits(64) ttbr;
  bit epd;
  if regime == Regime_EL2 then
    ttbr = HTTBR;
    txsz = walkparams.t0sz;
  else
    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    bits(64) ttbr0;
    bits(64) ttbr1;
    TTBCR_Type ttcbr;
    if regime == Regime_EL30 then
      ttcbr = TTBCR_S;
      ttbr0 = TTBR0_S;
      ttbr1 = TTBR1_S;
    elsif HaveAArch32EL(EL3) then
      ttcbr = TTBCR_NS;
      ttbr0 = TTBR0_NS;
      ttbr1 = TTBR1_NS;
    else
      ttcbr = TTBCR;
      ttbr0 = TTBR0;
      ttbr1 = TTBR1;

    assert ttcbr.EAE == '1';
    if varange == VARange_LOWER then
      txsz = walkparams.t0sz;
      ttbr = ttbr0;
      epd = ttcbr.EPD0;
    else
      txsz = walkparams.t1sz;
      ttbr = ttbr1;
      epd = ttcbr.EPD1;

    if regime != Regime_EL2 & epd == '1' then
      fault.level = 1;
      fault.statuscode = Fault_Translation;
      return (fault, TTWState UNKNOWN);

  // Input Address size
  iasize = AArch32.S1IASize(txsz);
  granulebits = TGxGranuleBits(walkparams.tgx);
stride = granulebits - 3;
startlevel = FINAL_LEVEL - (((iasize-1) - granulebits) DIV stride);
levels = FINAL_LEVEL - startlevel;

if !IsZero(ttbr<47:40>) then
  fault.statuscode = Fault_AddressSize;
  fault.level = 0;
  return (fault, TTWState UNKNOWN);

FullAddress baseaddress;
baseelsb = (iasize - (levels*stride + granulebits)) + 3;
baseaddress.paspace = if ss == SS_Secure then PAS_Secure else PAS_NonSecure;
baseaddress.address = ZeroExtend(ttbr<39:baseelsb>:Zeros(baseelsb), 52);

TTWState walkstate;
walkstate.baseaddress = baseaddress;
walkstate.level = startlevel;
walkstate.istable = TRUE;
// In regimes that support global and non-global translations, translation
// table entries from lookup levels other than the final level of lookup
// are treated as being non-global
walkstate.mG = if HasUnprivileged(regime) then '1' else '0';
waklstate.memattr = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
waklstate.permissions.ap_table = '00';
waklstate.permissions.xn_table = '0';
waklstate.permissions.pxn_table = '0';

indexmsb = iasize - 1;
bits(64) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ZeroExtend(va, 64);

if !AArch32.S1DCacheEnabled(regime) then
  walkaddress.memattr = NormalNCMemAttr();
  walkaddress.memattr.xs = walkstate.memattr.xs;
else
  walkaddress.memattr = walkstate.memattr;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32.EL2Enabled(ss) &&
  (if ELStateIsAArch32(EL2, ss == SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
  ![boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1"])
  then
  walkaddress.memattr.shareability = walkstate.memattr.shareability;
else
  walkaddress.memattr.shareability = EffectiveShareability(walkaddress.memattr);

integer indexlsb;
DescriptorType desctype;
repeat
  fault.level = walkstate.level;
  indexlsb = (FINAL_LEVEL - walkstate.level)*stride + granulebits;
  bits(40) index = ZeroExtend(va<indexmsb:indexlsb>:'000', 40);
  walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index, 52);
  walkaddress.paddress.paspace = walkstate.baseaddress.paspace;
  // If there are two stages of translation, then the first stage table walk addresses
  // are themselves subject to translation
  if regime == Regime_EL10 && AArch32.EL2Enabled(ss) then
    s2fs1walk = TRUE;
s2acctype = AccType_TTW;
s2aligned = TRUE;
s2write = FALSE;
  (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, ss, s2fs1walk,
    s2acctype, s2aligned, s2write, ispriv);
  // Check for a fault on the stage 2 walk
if s2fault.statuscode != Fault_None then
    return (s2fault, TTWState UNKNOWN);

(fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, fault, 64);
else
    (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault, 64);
if fault.statuscode != Fault_None then
    return (fault, TTWState UNKNOWN);

destype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);

case destype of
    when DescriptorType_Table
        if !IsZero(descriptor<47:40>) then
            fault.statuscode = Fault_AddressSize;
            return (fault, TTWState UNKNOWN);

        walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12), 52);
        if walkstate.baseaddress.paspace == PAS_Secure && descriptor<63> == '1' then
            walkstate.baseaddress.paspace = PAS_NonSecure;

        if walkparams.hpd == '0' then
            walkstate.permissions.xn_table  = (walkstate.permissions.xn_table  OR descriptor<60>);
            walkstate.permissions.ap_table  = (walkstate.permissions.ap_table  OR descriptor<62:61>);
            walkstate.permissions.pxn_table = (walkstate.permissions.pxn_table OR descriptor<59>);

        walkstate.level = walkstate.level + 1;
        indexmsb = indexlsb - 1;

        when DescriptorType_Invalid
            fault.statuscode = Fault_Translation;
            return (fault, TTWState UNKNOWN);

        when DescriptorType_Page, DescriptorType_Block
            walkstate.istable = FALSE;

        until destype IN {DescriptorType_Page, DescriptorType_Block};

// Check the output address is inside the supported range
if !IsZero(descriptor<47:40>) then
    fault.statuscode = Fault_AddressSize;
    return (fault, TTWState UNKNOWN);

// Check the access flag
if descriptor<10> == '0' then
    fault.statuscode = Fault_AccessFlag;
    return (fault, TTWState UNKNOWN);

walkstate.permissions.xn  = descriptor<54>;
walkstate.permissions.pxn = descriptor<53>;
walkstate.permissions.ap  = descriptor<7:6>:'1';
walkstate.contiguous      = descriptor<52>;
if regime == Regime_EL2 then
    // All EL2 regime accesses are treated as Global
    walkstate.nG = '0';
elseif ss == SS_Secure && walkstate.baseaddress.paspace == PAS_NonSecure then
    // When a PE is using the Long-descriptor translation table format,
    // and is in Secure state, a translation must be treated as non-global,
    // regardless of the value of the nG bit.
    // if NSTable is set to 1 at any level of the translation table walk.
    walkstate.nG = '1';
else
    walkstate.nG = descriptor<11>;

Armv8 Pseudocode

J1.2 Pseudocode for AArch32 operation

```plaintext
walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb), 52);
if walkstate.baseaddress.paspace == PAS_Secure & descriptor<5> == '1' then
    walkstate.baseaddress.paspace = PAS_NonSecure;

memattr = descriptor<4:2>;
sh     = descriptor<9:8>;
attr   = MAIRAttr(UInt(memattr), walkparams.mair);
s1aarch64 = FALSE;
walkstate.mattrs = S1DecodeMemAttrs(attr, sh, s1aarch64);

return (fault, walkstate);

aarch32/translation/walk/AArch32.S1WalkSD

// AArch32.S1WalkSD()
// ==================
// Traverse stage 1 translation tables in short format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S1WalkSD(FaultRecord fault_in, Regime regime, SecurityState ss, bits(32) va, boolean ispriv)

FaultRecord fault = fault_in;
SCTLR_Type sctlr;
TTBCR_Type ttbcr;
TTBR0_Type ttbr0;
TTBR1_Type ttbr1;
// Determine correct translation control registers to use.
if regime == Regime_EL30 then
    sctlr = SCTLR_S;
    ttbcr = TTBCR_S;
    ttbr0 = TTBR0_S;
    ttbr1 = TTBR1_S;
elsif HaveAArch32EL(EL3) then
    sctlr = SCTLR_NS;
    ttbcr = TTBCR_NS;
    ttbr0 = TTBR0_NS;
    ttbr1 = TTBR1_NS;
else
    sctlr = SCTLR;
    ttbcr = TTBCR;
    ttbr0 = TTBR0;
    ttbr1 = TTBR1;

assert ttbcr.EAE == '0';
ee  = sctlr.EE;
afe = sctlr.AFE;
tre = sctlr.TRE;
n  = UInt(ttbcr.N);
bits(32) ttb;
bits(1) pd;
bits(2) irgn;
bits(2) rgn;
bits(1) s;
bits(1) nos;
if n == 0 || IsZero(va<31:(32-n)>) then
    ttb  = ttbr0.TTB0:Zeros(7);
    pd   = ttbr0.PD0;
    irgn = ttbr0.IRGN;
    rgn  = ttbr0.RGN;
    s    = ttbr0.S;
    nos  = ttbr0.NOS;
else
    n    = 0; // TTBR1 translation always treats N as 0
    ttb  = ttbr1.TTB1:Zeros(7);
    pd   = ttbr1.PD1;
    irgn = ttbr1.IRGN;
    rgn  = ttbr1.RGN;
```

J1-11344

Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential
s = ttbr1.S;
nos = ttbr1.NOS;

// Check if Translation table walk disabled for translations with this Base register.
if pd == '1' then
  fault.level = 1;
  fault.statuscode = Fault_Translation;
  return (fault, TTWState UNKNOWN);

FullAddress baseaddress;
baseaddress.paspace = if ss == SS_Secure then PAS_Secure else PAS_NonSecure;
baseaddress.address = ZeroExtend(ttb<31:14-n>:Zeros(14-n), 52);

TTWState walkstate;
walkstate.baseaddress = baseaddress;

// In regimes that support global and non-global translations, translation
// table entries from lookup levels other than the final level of lookup
// are treated as being non-global. Translations in Short-Descriptor Format
// always support global & non-global translations.
walkstate.nG \= '1';
walkstate.memattrs = WalkMemAttrs(s:nos, irgn, rgn);
walkstate.level = 1;
walkstate.istable = TRUE;

bits(4) domain;
bits(32) descriptor;
AddressDescriptor walkaddress;
walkaddress.vaddress = ZeroExtend(va, 64);

if !AArch32.S1DCacheEnabled(regime) then
  walkaddress.memattrs = NormalNCMemAttr();
  walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
  walkaddress.memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32.EL2Enabled(ss) &&
    (if ELStateUsingAArch32(EL2, ss == SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
    (!boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
  walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
  walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

bit nG;
bit ns;
bit pxn;
bits(3) ap;
bits(3) tex;
bit c;
bit b;
bit xn;
repeat
  fault.level = walkstate.level;
  bits(32) index;
  if walkstate.level == 1 then
    index = ZeroExtend(va<31-n:20>:'00', 32);
  else
    index = ZeroExtend(va<19:12>:'00', 32);
  walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index, 52);
  walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

if regime == Regime_EL10 && AArch32.EL2Enabled(ss) then
  s2fs1walk = TRUE;
s2acctype = AccType_TTW;
s2aligned = TRUE;
s2write = FALSE;
(s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, ss, s2fs1walk, 
s2acctype, s2aligned, s2write, ispriv);

if s2fault.statuscode != Fault_None then 
  return (s2fault, TTWState UNKNOWN);

(fault, descriptor) = FetchDescriptor(ee, s2walkaddress, fault, 32);
else 
  (fault, descriptor) = FetchDescriptor(ee, walkaddress, fault, 32);

if fault.statuscode != Fault_None then 
  return (fault, TTWState UNKNOWN);

walkstate.sdftype = AArch32.DecodeDescriptorTypeSD(descriptor, walkstate.level);

case walkstate.sdftype of
  when SDFType_Invalid
    fault.domain = domain;
    fault.statuscode = Fault_Translation;
    return (fault, TTWState UNKNOWN);

  when SDFType_Table
    domain = descriptor<8:5>;
    ns     = descriptor<3>;
    pxn    = descriptor<2>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:10>:Zeros(10), 52);
    walkstate.level = 2;

  when SDFType_SmallPage
    nG  = descriptor<11>;
    s   = descriptor<10>;
    ap  = descriptor<9,5:4>;
    tex = descriptor<8:6>;
    c   = descriptor<3>;
    b   = descriptor<2>;
    xn  = descriptor<0>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:12>:Zeros(12), 52);
    walkstate.istable = FALSE;

  when SDFType_LargePage
    xn  = descriptor<15>;
    tex = descriptor<14:12>;
    nG  = descriptor<11>;
    s   = descriptor<10>;
    ap  = descriptor<9,5:4>;
    c   = descriptor<3>;
    b   = descriptor<2>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:16>:Zeros(16), 52);
    walkstate.istable = FALSE;

  when SDFType_Section
    ns     = descriptor<19>;
    nG     = descriptor<17>;
    s      = descriptor<16>;
    ap     = descriptor<15,11:10>;
    tex    = descriptor<14:12>;
    domain = descriptor<8:5>;
    xn     = descriptor<4>;
    c      = descriptor<3>;


b = descriptor<2>;
pxn = descriptor<0>;

walkstate.baseaddress.address = ZeroExtend(descriptor<31:20>:Zeros(20), S2);
walkstate.istable = FALSE;

when SDFType_Supersection
ns = descriptor<19>;
nG = descriptor<17>;
s = descriptor<16>;
ap = descriptor<15,11:10>;
tex = descriptor<14:12>;
xn = descriptor<4>;
c = descriptor<3>;
b = descriptor<2>;
pxn = descriptor<0>;
domain = '0000';

walkstate.baseaddress.address = ZeroExtend(descriptor<8:5,23:20,31:24>:Zeros(24), S2);
walkstate.istable = FALSE;

until walkstate.sdftype != SDFType_Table;

if afe == '1' && ap<0> == '0' then
fault.domain = domain;
fault.statuscode = Fault_AccessFlag;
return (fault, TTWState UNKNOWN);

// Decode the TEX, C, B and S bits to produce target memory attributes
if tre == '1' then
walkstate.mematts = AArch32.RemappedTEXDecode(regime, tex, c, b, s);
elsif RemapRegsHaveResetValues() then
walkstate.mematts = AArch32.DefaultTEXDecode(tex, c, b, s);
else
walkstate.mematts = MemoryAttributes IMPLEMENTATION_DEFINED;
walkstate.permissions.ap = ap;
walkstate.permissions.xn = xn;
walkstate.permissions.pxn = pxn;
walkstate.domain = domain;
walkstate.nG = nG;

if ss == SS_Secure && ns == '0' then
walkstate.baseaddress.paspace = PAS_Secure;
else
walkstate.baseaddress.paspace = PAS_NonSecure;

return (fault, walkstate);

aarch32/translation/walk/AArch32.S2IASize

// AArch32.S2IASize()
// ===============
// Retrieve the number of bits containing the input address for stage 2 translation

integer AArch32.S2IASize(bits(4) t0sz)
return 32 - SInt(t0sz);
aarch32/translation/walk/AArch32.S2StartLevel

// AArch32.S2StartLevel()
// ======================
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch32.S2StartLevel(bits(2) sl0)
    return 2 - UInt(sl0);

aarch32/translation/walk/AArch32.S2Walk

// AArch32.S2Walk()
// ================
// Traverse stage 2 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S2Walk(FaultRecord fault_in, S2TTWParams walkparams, AddressDescriptor ipa)
    FaultRecord fault = fault_in;
    if walkparams.sl0 IN {'1x'} || AArch32.S2InconsistentSL(walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level = 1;
        return (fault, TTWState UNKNOWN);

// Input Address size
iasize = AArch32.S2IASize(walkparams.t0sz);
startlevel = AArch32.S2StartLevel(walkparams.sl0);
levels = FINAL_LEVEL - startlevel;
granulebits = TGxGranuleBits(walkparams.tgx);
stride = granulebits - 3;
if !IsZero(VTTBR<47:40>) then
    fault.statuscode = Fault_AddressSize;
    fault.level = 0;
    return (fault, TTWState UNKNOWN);

FullAddress baseaddress;
baselsb = (iasize - (levels*stride + granulebits)) + 3;
baseaddress.paspace = PAS_NonSecure;
baseaddress.address = ZeroExtend(VTTBR<39:baselsb>:Zeros(baselsb), 52);

TTWState walkstate;
w alkstate.baseaddress = baseaddress;
walkstate.level = startlevel;
w alkstate.istable = TRUE;
w alkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn,
w alkparams.orgn);
indexmsb = iasize - 1;
bits(64) descriptor;
AddressDescriptor walkaddress;
w alkaddress.vaddress = ipa.vaddress;
if HCR2.CD == '1' then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;
walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

integer indexlsb;
DescriptorType desctype;
repeat
    fault.level = walkstate.level;
    

indexlsb = (FINAL_LEVEL - walkstate.level) * stride + granulebits;
bits(40) index = ZeroExtend(ipa.paddress.address<indexmsb:indexlsb>:'000', 40);

walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index, 52);
walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

(fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault, 64);
if fault.statuscode != Fault_None then
    return (fault, TTWState UNKNOWN);

desctype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);
case desctype of
    when DescriptorType_Table
        if !IsZero(descriptor<47:40>) then
            fault.statuscode = Fault_AddressSize;
            return (fault, TTWState UNKNOWN);
        walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12), 52);
        walkstate.level = walkstate.level + 1;
        indexmsb = indexlsb - 1;
    when DescriptorType_Invalid
        fault.statuscode = Fault_Translation;
        return (fault, TTWState UNKNOWN);
    when DescriptorType_Page, DescriptorType_Block
        walkstate.istable = FALSE;
    until desctype IN {DescriptorType_Page, DescriptorType_Block};

    // Check the output address is inside the supported range
    if !IsZero(descriptor<47:40>) then
        fault.statuscode = Fault_AddressSize;
        return (fault, TTWState UNKNOWN);

    // Check the access flag
    if descriptor<10> == '0' then
        fault.statuscode = Fault_AccessFlag;
        return (fault, TTWState UNKNOWN);

    // Unpack the descriptor into address and upper and lower block attributes
    walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb), 52);

    walkstate.permissions.s2ap = descriptor<7:6>;
    walkstate.permissions.s2xn = descriptor<54>;
    if HaveExtendedExecuteNeverExt() then
        walkstate.permissions.s2xnx = descriptor<53>;
    else
        walkstate.permissions.s2xnx = '0';

    memattr = descriptor<5:2>;
    sh = descriptor<9:8>;
    s2aarch64 = FALSE;
    walkstate.memattrs = S2DecodeMemAttrs(memattr, sh, s2aarch64);
    walkstate.contiguous = descriptor<52>;

    return (fault, walkstate);

arch32/translation/walk/AArch32.TranslationSizeSD
// AArch32.TranslationSizeSD()
// ===========================
// Determine the size of the translation
integer AArch32.TranslationSizeSD(SDFType sdftype)
integer tsize;
case sdftype of
  when SDFType_SmallPage      tsize = 12;
  when SDFType_LargePage      tsize = 16;
  when SDFType_Section        tsize = 20;
  when SDFType_Supersection   tsize = 24;
return tsize;

aarch32/translation/walk/RemapRegsHaveResetValues

boolean RemapRegsHaveResetValues();

aarch32/translation/walkparams/AArch32.GetS1TTWParams

// AArch32.GetS1TTWParams()
// ========================
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.
S1TTWParams AArch32.GetS1TTWParams(Regime regime, bits(32) va)
S1TTWParams walkparams;
case regime of
  when Regime_EL2  walkparams = AArch32.S1TTWParamsEL2();
  when Regime_EL10 walkparams = AArch32.S1TTWParamsEL10(va);
  when Regime_EL30 walkparams = AArch32.S1TTWParamsEL30(va);
return walkparams;

aarch32/translation/walkparams/AArch32.GetS2TTWParams

// AArch32.GetS2TTWParams()
// ========================
// Gather walk parameters for stage 2 translation
S2TTWParams AArch32.GetS2TTWParams()
S2TTWParams walkparams;
walkparams.tgx  = TGx_4KB;
walkparams.s    = VTCR.S;
walkparams.t0sz = VTCR.T0SZ;
walkparams.s10  = VTCR.SL0;
walkparams.irgn = VTCR.IRGNO;
walkparams.orgn = VTCR.ORGN0;
walkparams.sh   = VTCR.SH0;
walkparams.ee   = HSCTLR.EE;
walkparams.ptw  = HCR.PTW;
walkparams.vm   = HCR.VM OR HCR.DC;
// VTCR.S must match VTCR.T0SZ[3]
if walkparams.s != walkparams.t0sz<3> then
  (-, walkparams.t0sz) = ConstrainUnpredictableBits(Unpredictable_RESVTCRS, 4);
return walkparams;
aarch32/translation/walkparams/AArch32.GetVARange

// AArch32.GetVARange()
// ====================
// Select the translation base address for stage 1 long-descriptor walks

VARange AArch32.GetVARange(bits(32) va, bits(3) t0sz, bits(3) t1sz)
    // Lower range Input Address size
    lo_iasize = AArch32.S1IASize(t0sz);
    // Upper range Input Address size
    up_iasize = AArch32.S1IASize(t1sz);
    if t1sz == '000' && t0sz == '000' then
        return VARange_LOWER;
    elsif t1sz == '000' then
        return if IsZero(va<31:lo_iasize>) then VARange_LOWER else VARange_UPPER;
    elsif t0sz == '000' then
        return if IsOnes(va<31:up_iasize>) then VARange_UPPER else VARange_LOWER;
    elsif IsZero(va<31:lo_iasize>) then
        return VARange_LOWER;
    elsif IsOnes(va<31:up_iasize>) then
        return VARange_UPPER;
    else
        // Will be reported as a Translation Fault
        return VARange UNKNOWN;
    end if;

aarch32/translation/walkparams/AArch32.S1DCacheEnabled

// AArch32.S1DCacheEnabled()
// =========================
// Determine cacheability of stage 1 data accesses

boolean AArch32.S1DCacheEnabled(Regime regime)
    case regime of
        when Regime_EL30 return SCTLR_S.C == '1';
        when Regime_EL2  return HSCTLR.C == '1';
        when Regime_EL10 return (if HaveAArch32EL(EL3) then SCTLR_NS.C else SCTLR.C) == '1';
    end case;

aarch32/translation/walkparams/AArch32.S1ICacheEnabled

// AArch32.S1ICacheEnabled()
// =========================
// Determine cacheability of stage 1 instruction fetches

boolean AArch32.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime_EL30 return SCTLR_S.I == '1';
        when Regime_EL2  return HSCTLR.I == '1';
        when Regime_EL10 return (if HaveAArch32EL(EL3) then SCTLR_NS.I else SCTLR.I) == '1';
    end case;

aarch32/translation/walkparams/AArch32.S1TTWParamsEL10

// AArch32.S1TTWParamsEL10()
// =========================
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled).

S1TTWParams AArch32.S1TTWParamsEL10(bits(32) va)
    bits(64) mair;
    bit sif;
    TTBCR_Type ttbcr;
    TTBCR2_Type ttbcr2;
    SCTLR_Type sclr;
if HaveAArch32EL(EL3) then
  ttbcr  = TTBCR_NS;
ttbcr2 = TTBCR2_NS;
sctlr  = SCTLR_NS;
mair   = MAIR1_NS:MAIR0_NS;
sif    = SCR.SIF;
else
  ttbcr  = TTBCR;
ttbcr2 = TTBCR2;
sctlr  = SCTLR;
mair   = MAIR1:MAIR0;
sif    = SCR_EL3.SIF;
assert ttbcr.EAE == '1';
S1TTWParams walkparams;
walkparams.t0sz   = ttbcr.T0SZ;
walkparams.t1sz   = ttbcr.T1SZ;
walkparams.ee     = sctlr.EE;
walkparams.wxn    = sctlr.WXN;
walkparams.uwxn   = sctlr.UWXN;
walkparams.ntlsmd = if HaveTrapLoadStoreMultipleDeviceExt() then sctlr.nTLSMD else '1';
walkparams.mair   = mair;
walkparams.sif    = sif;
varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
if varange == VARange_LOWER then
  walkparams.sh   = ttbcr.SH0;
  walkparams.ign  = ttbcr.IRG0;
  walkparams.orgn = ttbcr.ORG0;
  walkparams.hpd  = if AArch32.HaveHPDExt() then ttbcr.T2E AND ttbcr2.HPD0 else '0';
else
  walkparams.sh   = ttbcr.SH1;
  walkparams.ign  = ttbcr.IRG1;
  walkparams.orgn = ttbcr.ORG1;
  walkparams.hpd  = if AArch32.HaveHPDExt() then ttbcr.T2E AND ttbcr2.HPD1 else '0';
return walkparams;

aarch32/translation/walkparams/AArch32.S1TTWParamsEL2

// AArch32.S1TTWParamsEL2()
// ========================
// Gather stage 1 translation table walk parameters for EL2 regime
S1TTWParams AArch32.S1TTWParamsEL2()
S1TTWParams walkparams;
walkparams.tgx  = TGx_4KB;
walkparams.t0sz = HTCR.T0SZ;
walkparams.ign  = HTCR.IRG0;
walkparams.orgn = HTCR.ORG0;
walkparams.sh   = HTCR.SH0;
walkparams.hpd  = if AArch32.HaveHPDExt() then HTCR.HPD else '0';
walkparams.ee   = HSCTLR.EE;
walkparams.wxn  = HSCTLR.WXN;
if HaveTrapLoadStoreMultipleDeviceExt() then
  walkparams.ntlsmd = HSCTLR.nTLSMD;
else
  walkparams.ntlsmd = '1';
walkparams.mair = HMAIR1:HMAIR0;
return walkparams;
aarch32/translation/walkparams/AArch32.S1TTWParamsEL30

// AArch32.S1TTWParamsEL30()
// =======================================
// Gather stage 1 translation table walk parameters for EL3&0 regime

S1TTWParams AArch32.S1TTWParamsEL30(bits(32) va)
assert TTBCR_S.EAE == '1';
S1TTWParams walkparams;

walkparams.t0sz   = TTBCR_S.T0SZ;
walkparams.t1sz   = TTBCR_S.T1SZ;
walkparams.ee     = SCTLR_S.EE;
walkparams.wxn    = SCTLR_S.WXN;
walkparams.uwxn   = SCTLR_S.UWXN;
walkparams.ntlsmd = if HaveTrapLoadStoreMultipleDeviceExt() then SCTLR_S.nTLSMD else '1';
walkparams.mair   = MAIR1_S:MAIR0_S;
walkparams.sif    = SCR_SIF;
varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
if varange == VARange_LOWER then
    walkparams.sh   = TTBCR_S.SH0;
    walkparams.irgn = TTBCR_S.IRGN0;
    walkparams.orgn = TTBCR_S.ORGN0;
    walkparams.hpd  = if AArch32.HaveHPDExt() then TTBCR_S.T2E AND TTBCR2_S.HPD0 else '0';
else
    walkparams.sh   = TTBCR_S.SH1;
    walkparams.irgn = TTBCR_S.IRGN1;
    walkparams.orgn = TTBCR_S.ORGN1;
    walkparams.hpd  = if AArch32.HaveHPDExt() then TTBCR_S.T2E AND TTBCR2_S.HPD1 else '0';
return walkparams;
J1.3 Shared pseudocode

This section holds the pseudocode that is common to execution in AArch64 state and in AArch32 state. Functions listed in this section are identified only by a FunctionName, without an AArch64 or AArch32 prefix. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example shared/debug/DebugTarget.

The top-level sections of the shared pseudocode hierarchy are:

- shared/debug
- shared/exceptions on page J1-11382.
- shared/functions on page J1-11384.
- shared/trace on page J1-11528.
- shared/translation on page J1-11533.

J1.3.1 shared/debug

This section includes the following pseudocode functions:

- shared/debug/DebugTarget/DebugTarget on page J1-11356.
- shared/debug/DebugTarget/DebugTargetFrom on page J1-11356.
- shared/debug/OSLockStatus/OSLockStatus on page J1-11357.
- shared/debug/SoftwareLockStatus/Component on page J1-11357.
- shared/debug/SoftwareLockStatus/SoftwareLockStatus on page J1-11357.
- shared/debug/authentication/AccessState on page J1-11357.
- shared/debug/authentication/AllowExternalDebugAccess on page J1-11357.
- shared/debug/authentication/AllowExternalPMUAccess on page J1-11358.
- shared/debug/authentication/AllowExternalTraceAccess on page J1-11359.
- shared/debug/authentication/Debug_authentication on page J1-11359.
- shared/debug/authentication/ExternalInvasiveDebugEnabled on page J1-11359.
- shared/debug/authentication/ExternalNoninvasiveDebugAllowed on page J1-11360.
- shared/debug/authentication/ExternalNoninvasiveDebugEnabled on page J1-11360.
- shared/debug/authentication/ExternalRealmNoninvasiveDebugEnabled on page J1-11360.
- shared/debug/authentication/ExternalRootNoninvasiveDebugEnabled on page J1-11361.
- shared/debug/authentication/IsCorePowered on page J1-11362.
- shared/debug/breakpoint/CheckValidStateMatch on page J1-11362.
- shared/debug/breakpoint/NumBreakpointsImplemented on page J1-11363.
- shared/debug/breakpoint/NumContextAwareBreakpointsImplemented on page J1-11363.
- shared/debug/breakpoint/NumWatchpointsImplemented on page J1-11363.
- shared/debug/cti/CTI_SetEventLevel on page J1-11363.
- shared/debug/cti/CTI_SignalEvent on page J1-11363.
- shared/debug/cti/CrossTrigger on page J1-11363.
- shared/debug/dccanditr/CheckForDCCInterrupts on page J1-11363.
- shared/debug/dccanditr(DBGDTRRX_EL0 on page J1-11364.
- shared/debug/dccanditr(DBGDTRTX_EL0 on page J1-11365.
- shared/debug/dccanditr(DBGDTR_EL0 on page J1-11366.
shared/debug/ClearStickyErrors/ClearStickyErrors

    // ClearStickyErrors()
    // ===================
    ClearStickyErrors()
EDSCR.TXU = '0'; // Clear TX underrun flag
EDSCR.RXO = '0'; // Clear RX overrun flag
if Halted() then // in Debug state
    EDSCR.ITO = '0'; // Clear ITR overrun flag

// If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
// The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
// in the pseudocode.
if Halted() && EDSCR.ITE == '0' && ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERO) then
    return;
EDSCR.ERR = '0'; // Clear cumulative error flag
return;

shared/debug/DebugTarget/DebugTarget

// DebugTarget()
// =============
// Returns the debug exception target Exception level
bits(2) DebugTarget()
    ss = CurrentSecurityState();
    return DebugTargetFrom(ss);

shared/debug/DebugTarget/DebugTargetFrom

// DebugTargetFrom()
// ===============
// bits(2) DebugTargetFrom(SecurityState from_state)
// boolean route_to_el2;
bool route_to_el2;
    if HaveEL(EL2) && (from_state != SS_Secure ||
        (HaveSecureEL2Ext() && (!HaveEL(EL3) || SCR_EL3.EEL2 == '1'))) then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

bits(2) target;
    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && !HaveAArch64() && from_state == SS_Secure then
        target = EL3;
    else
        target = EL1;
    return target;

shared/debug/DoubleLockStatus/DoubleLockStatus

// DoubleLockStatus()
// ================
// Returns the state of the OS Double Lock.
// FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
// TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.
bool DoubleLockStatus()
    if !HaveDoubleLock() then
        return FALSE;
    elsif ELUsingAArch32(EL1) then

return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
else
  return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();

shared/debug/OSLockStatus/OSLockStatus

// OSLockStatus()
// ==============
// Returns the state of the OS Lock.

boolean OSLockStatus()
  return (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK) == '1';

shared/debug/SoftwareLockStatus/Component
enumeration Component {
  Component_PMU,
  Component_Debug,
  Component_CTI
};

shared/debug/SoftwareLockStatus/GetAccessComponent

// Returns the accessed component.
Component GetAccessComponent();

shared/debug/SoftwareLockStatus/SoftwareLockStatus

// SoftwareLockStatus()
// ====================
// Returns the state of the Software Lock.

boolean SoftwareLockStatus()
  Component component = GetAccessComponent();
  if !HaveSoftwareLock(component) then
    return FALSE;
  case component of
    when Component_Debug
      return EDLSR.SLK == '1';
    when Component_PMU
      return PMLSR.SLK == '1';
    when Component_CTI
      return CTILSR.SLK == '1';
    otherwise
      Unreachable();
  endcase;

shared/debug/authentication/AccessState

// Returns the Security state of the access.
SecurityState AccessState();

shared/debug/authentication/AllowExternalDebugAccess

// AllowExternalDebugAccess()
// ==========================
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed, FALSE otherwise.
boolean AllowExternalDebugAccess()
   // The access may also be subject to OS Lock, power-down, etc.
   return AllowExternalDebugAccess(AccessState());

// AllowExternalDebugAccess()
// =========================
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed for the given state, FALSE otherwise.

boolean AllowExternalDebugAccess(SecurityState access_state)
   // The access may also be subject to OS Lock, power-down, etc.
   if HaveRME() then
      case MDCR_EL3.<EDADE,EDAD> of
         when '00' return TRUE;
         when '01' return access_state IN {SS_Root, SS_Secure};
         when '10' return access_state IN {SS_Root, SS_Realm};
         when '11' return access_state == SS_Root;
      end case;
      if HaveSecureExtDebugView() then
         if access_state == SS_Secure then return TRUE;
         else
            if !ExternalInvasiveDebugEnabled() then return FALSE;
            if ExternalSecureInvasiveDebugEnabled() then return TRUE;
         end if;
      end if;
   if HaveEL(EL3) then
      EDAD_bit = if ELUsingAArch32(EL3) then SDCR.EDAD else MDCR_EL3.EDAD;
      return EDAD_bit == '0';
   else
      return NonSecureOnlyImplementation();
   end if;

shared/debug/authentication/AllowExternalPMUAccess

// AllowExternalPMUAccess()
// ========================
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed, FALSE otherwise.

boolean AllowExternalPMUAccess()
   // The access may also be subject to OS Lock, power-down, etc.
   return AllowExternalPMUAccess(AccessState());

// AllowExternalPMUAccess()
// ========================
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed for the given Security state, FALSE otherwise.

boolean AllowExternalPMUAccess(SecurityState access_state)
   // The access may also be subject to OS Lock, power-down, etc.
   if HaveRME() then
      case MDCR_EL3.<EPMADE,EPMAD> of
         when '00' return TRUE;
         when '01' return access_state IN {SS_Root, SS_Secure};
         when '10' return access_state IN {SS_Root, SS_Realm};
         when '11' return access_state == SS_Root;
      end case;
      if HaveSecureExtDebugView() then
         if access_state == SS_Secure then return TRUE;
         else
            if !ExternalInvasiveDebugEnabled() then return FALSE;
            if ExternalSecureInvasiveDebugEnabled() then return TRUE;
         end if;
      end if;
   if HaveEL(EL3) then
      EPMAD_bit = if ELUsingAArch32(EL3) then SDCR.EPMAD else MDCR_EL3.EPMAD;
      return EPMAD_bit == '0';
   else
      return NonSecureOnlyImplementation();
   end if;
else
    return NonSecureOnlyImplementation();

shared/debug/authentication/AllowExternalTraceAccess

// AllowExternalTraceAccess()
// "=========================
// Returns TRUE if an external Trace access to the Trace registers is allowed, FALSE otherwise.

boolean AllowExternalTraceAccess()
if !HaveTraceBufferExtension() then
    return TRUE;
else
    return AllowExternalTraceAccess(AccessState());

// AllowExternalTraceAccess()
// "=========================
// Returns TRUE if an external Trace access to the Trace registers is allowed for the
// given Security state, FALSE otherwise.

boolean AllowExternalTraceAccess(SecurityState access_state)
if !HaveTraceBufferExtension() then return TRUE;
assert HaveSecureExtDebugView();
if HaveRME() then
    case MDCR_EL3.ETAD,ETAD of
        when '00' return TRUE;
        when '01' return access_state IN {SS_Root, SS_Secure};
        when '10' return access_state IN {SS_Root, SS_Realm};
        when '11' return access_state == SS_Root;
if access_state == SS_Secure then return TRUE;
if HaveEL(EL3) then
    // External Trace access is not supported for EL3 using AArch32
    assert !ELUsingAArch32(EL3);
    return MDCR_EL3.ETAD == '0';
else
    return NonSecureOnlyImplementation();

shared/debug/authentication/Debug_authentication

signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN;
signal RLPIDEN;
signal RTPIDEN;

shared/debug/authentication/ExternalInvasiveDebugEnabled

// ExternalInvasiveDebugEnabled()
// "=================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGEN signal.

boolean ExternalInvasiveDebugEnabled()
return DBGEN == HIGH;
shared/debug/authentication/ExternalNoninvasiveDebugAllowed

// ExternalNoninvasiveDebugAllowed()
// // Returns TRUE if Trace and PC Sample-based Profiling are allowed

boolean ExternalNoninvasiveDebugAllowed()
    return ExternalNoninvasiveDebugAllowed(PSTATE.EL);

// ExternalNoninvasiveDebugAllowed()
// // ==================================================

boolean ExternalNoninvasiveDebugAllowed(bits(2) el)
    if !ExternalNoninvasiveDebugEnabled() then return FALSE;
    ss = SecurityStateAtEL(el);
    if ((ELUsingAArch32(EL3) || ELUsingAArch32(EL1)) && el == EL0 &&
        ss == SS_Secure && SDER.SUNIDEN == '1') then
        return TRUE;
    case ss of
        when SS_NonSecure return TRUE;
        when SS_Secure return ExternalSecureNoninvasiveDebugEnabled();
        when SS_Role return ExternalRealmNoninvasiveDebugEnabled();
        when SS_Root return ExternalRootNoninvasiveDebugEnabled();

shared/debug/authentication/ExternalNoninvasiveDebugEnabled

// ExternalNoninvasiveDebugEnabled()
// // This function returns TRUE if the FEAT_Debugv8p4 is implemented.
// // Otherwise, this function is IMPLEMENTATION DEFINED, and, in the
// // recommended interface, ExternalNoninvasiveDebugEnabled returns
// // the state of the (DBGEN OR NIDEN) signal.

boolean ExternalNoninvasiveDebugEnabled()
    return !HaveNoninvasiveDebugAuth() || ExternalInvasiveDebugEnabled() || NIDEN == HIGH;

shared/debug/authentication/ExternalRealmInvasiveDebugEnabled

// ExternalRealmInvasiveDebugEnabled()
// // The definition of this function is IMPLEMENTATION DEFINED.
// // In the recommended interface, this function returns the state of the
// // (DBGEN AND RLPIDEN) signal.

boolean ExternalRealmInvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return ExternalInvasiveDebugEnabled() && RLPIDEN == HIGH;

shared/debug/authentication/ExternalRealmNoninvasiveDebugEnabled

// ExternalRealmNoninvasiveDebugEnabled()
// // The definition of this function is IMPLEMENTATION DEFINED.
// // In the recommended interface, this function returns the state of the
// // (DBGEN AND RLPIDEN) signal.

boolean ExternalRealmNoninvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return ExternalRealmInvasiveDebugEnabled();
shared/debug/authentication/ExternalRootInvasiveDebugEnabled

// ExternalRootInvasiveDebugEnabled()
// -----------------------------------
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN AND SPIDEN AND RTPIDEN) signal.

boolean ExternalRootInvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return (ExternalInvasiveDebugEnabled() &&
            ExternalSecureInvasiveDebugEnabled() &&
            ExternalRealmInvasiveDebugEnabled() &&
            RTPIDEN == HIGH);

shared/debug/authentication/ExternalRootNoninvasiveDebugEnabled

// ExternalRootNoninvasiveDebugEnabled()
// -------------------------------------
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN AND SPIDEN AND RTPIDEN) signal.

boolean ExternalRootNoninvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return ExternalRootInvasiveDebugEnabled();

shared/debug/authentication/ExternalSecureInvasiveDebugEnabled

// ExternalSecureInvasiveDebugEnabled()
// -----------------------------------
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND SPIDEN) signal.
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.

boolean ExternalSecureInvasiveDebugEnabled()
    if !HaveEL(EL3) && !SecureOnlyImplementation() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;

shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

// ExternalSecureNoninvasiveDebugEnabled()
// -------------------------------------
// This function returns the value of ExternalSecureInvasiveDebugEnabled() when FEAT_Debugv8p4
// is implemented. Otherwise, the definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.

boolean ExternalSecureNoninvasiveDebugEnabled()
    if !HaveEL(EL3) && !SecureOnlyImplementation() then return FALSE;
    if HaveNoninvasiveDebugAuth() then
        return ExternalNoninvasiveDebugEnabled() && SPIDEN == HIGH || SPNIDEN == HIGH;
    else
        return ExternalSecureInvasiveDebugEnabled();

shared/debug/authentication/IsAccessSecure

// Returns TRUE when an access is Secure
boolean IsAccessSecure();
shared/debug/authentication/IsCorePowered

// Returns TRUE if the Core power domain is powered on, FALSE otherwise.
boolean IsCorePowered();

shared/debug/breakpoint/CheckValidStateMatch

// CheckValidStateMatch()
// ======================
// Checks for an invalid state match that will generate Constrained
// Unpredictable behaviour, otherwise returns Constraint_NONE.
(Constraint, bits(2), bit, bit, bits(2)) CheckValidStateMatch(bits(2) SSC_in, bit SSCE_in, bit HMC_in,
bits(2) PxC_in, boolean isbreakpnt)
if !HaveRME() then assert SSCE_in == '0';
boolean reserved = FALSE;
bits(2) SSC = SSC_in;
bit SSCE = SSCE_in;
bit HMC = HMC_in;
bits(2) PxC = PxC_in;

// Values that are not allocated in any architecture version
case HMC:SSCE:SSC:PxC of
  when '0 0 11 10' reserved = TRUE;
  when '1 0 00 x0' reserved = TRUE;
  when '1 0 01 10' reserved = TRUE;
  when '1 0 1x xx' reserved = SSC != '01' || (HMC:PxC) IN {'000','110'};
else reserved = FALSE;

// Match 'Usr/Sys/Svc' valid only for AArch32 breakpoints
if (!isbreakpnt || !HaveAArch32EL(EL1)) && HMC:PxC == '000' && SSC != '11' then
  reserved = TRUE;

// Both EL3 and EL2 are not implemented
if !HaveEL(EL3) && !HaveEL(EL2) && (HMC != '0' || SSC != '00') then
  reserved = TRUE;

// EL3 is not implemented
if !HaveEL(EL3) && SSC IN {'01','10'} && HMC:SSC:PxC != '10100' then
  reserved = TRUE;

// EL3 using AArch64 only
if (!HaveEL(EL3) || !HaveAArch64()) && HMC:SSC:PxC == '11000' then
  reserved = TRUE;

// EL2 is not implemented
if !HaveEL(EL2) && HMC:SSC:PxC == '11100' then
  reserved = TRUE;

// Secure EL2 is not implemented
if !HaveSecureEL2Ext() && (HMC:SSC:PxC) IN {'01100','10100','x11x1'} then
  reserved = TRUE;

if reserved then
  // If parameters are set to a reserved type, behaves as either disabled or a defined type
  Constraint c;
  (c, <HMC,SSC,SSCE,PxC>) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL, 6);
  assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
  if c == Constraint_DISABLED then
    return (c, bits(2) UNKNOWN, bit UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);
  // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value
  return (Constraint_NONE, SSC, SSCE, HMC, PxC);
shared/debug/breakpoint/NumBreakpointsImplemented

// NumBreakpointsImplemented()
// ===========================
// Returns the number of breakpoints implemented. This is indicated to software by
// DBGIDR.BRPs in AArch32 state, and ID_AA64DFR0_EL1.BRPs in AArch64 state.

integer NumBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of breakpoints";

shared/debug/breakpoint/NumContextAwareBreakpointsImplemented

// NumContextAwareBreakpointsImplemented()
// =======================================
// Returns the number of context-aware breakpoints implemented. This is indicated to software by
// DBGIDR.CTX_CMPs in AArch32 state, and ID_AA64DFR0_EL1.CTX_CMPs in AArch64 state.

integer NumContextAwareBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of context-aware breakpoints";

shared/debug/breakpoint/NumWatchpointsImplemented

// NumWatchpointsImplemented()
// ===========================
// Returns the number of watchpoints implemented. This is indicated to software by
// DBGIDR.WRPs in AArch32 state, and ID_AA64DFR0_EL1.WRPs in AArch64 state.

integer NumWatchpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of watchpoints";

shared/debug/cti/CTI_SetEventLevel

// Set a Cross Trigger multi-cycle input event trigger to the specified level.
// CTI_SetEventLevel(CrossTriggerIn id, signal level);

shared/debug/cti/CTI_SignalEvent

// Signal a discrete event on a Cross Trigger input event trigger.
// CTI_SignalEvent(CrossTriggerIn id);

shared/debug/cti/CrossTrigger

enumeration CrossTriggerOut {CrossTriggerOut_DebugRequest, CrossTriggerOut_IRQ, CrossTriggerOut_TraceExtIn0, CrossTriggerOut_TraceExtIn1, CrossTriggerOut_TraceExtIn2, CrossTriggerOut_TraceExtIn3, CrossTriggerOut_RestartRequest, CrossTriggerOut_RST03, CrossTriggerOut_TraceExtIn1, CrossTriggerOut_TraceExtIn2};

enumeration CrossTriggerIn {CrossTriggerIn_CrossHalt, CrossTriggerIn_PMUOverflow, CrossTriggerIn_RST02, CrossTriggerIn_RST03, CrossTriggerIn_TraceExtOut0, CrossTriggerIn_TraceExtOut1, CrossTriggerIn_TraceExtOut2, CrossTriggerIn_TraceExtOut3};

shared/debug/dccanditr/CheckForDCCInterrupts

// CheckForDCCInterrupts()
// =======================
CheckForDCCInterrupts()
commrx = (EDSCR.RXfull == '1');
commtx = (EDSCR.TXfull == '0');

// COMMRX and COMMTX support is optional and not recommended for new designs.
// SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
// SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

// The value to be driven onto the common COMMIRQ signal.
boolean commirq;
if ELUsingAArch32(EL1) then
    commirq = ((commrx && DBGDCCINT.RX == '1') ||
               (commtx && DBGDCCINT.TX == '1'));
else
    commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
               (commtx && MDCCINT_EL1.TX == '1'));
SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then HIGH else LOW);
return;

shared/debug/dccanditr/DBGDTRRX_EL0

// DBGDTRRX_EL0[] (external write)
// -------------------------------
// Called on writes to debug register 0x08C.

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value
if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return;
if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write
if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write
if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
    EDSCR.RXO = '1';  EDSCR.ERR = '1';              // Overrun condition: ignore write
    return;
EDSCR.RXfull = '1';
DTRRX = value;
if Halted() && EDSCR.MA == '1' then
    EDSCR.ITE = '0';                                // See comments in EDITR[] (external write)
    if ELUsingAArch32(EL1) then
        ExecuteA64(0x0D330501<31:0>); // A64 "MRS X1,DBGDTRRX_EL0"
        ExecuteA64(0x8B004401<31:0>);  // A64 "STR W1,[X0],#4"
        X[1, 64] = bits(64) UNKNOWN;
    else
        ExecuteT32(0x8F40<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);  // T32 "STR R1,[R0],#4"
        R[1] = bits(32) UNKNOWN;
    // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
    if EDSCR.ERR == '1' then
        EDSCR.RXfull = bit UNKNOWN;
        DBGDTRRX_EL0 = bits(64) UNKNOWN;
    else
        "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
        assert EDSCR.RXfull == '0';
        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)
        return;
    // DBGDTRRX_EL0[] (external read)

// ==============================
bits(32) DBGDTRRX_EL0[boolean memory_mapped]
  return DTRRX;

shared/debug/dccanditr/DBGDTRTX_EL0

  // DBGDTRTX_EL0[] (external read)
  // ==============================
  // Called on reads of debug register 0x080.
  // The Software lock is OPTIONAL.
  bits(32) DBGDTRTX_EL0[boolean memory_mapped]
    if EDPRSR<6:5,0> != '001' then                     // Check DLK, OSLK and PU bits
      IMPLEMENTATION_DEFINED "generate error response";
      return bits(32) UNKNOWN;
    end if;
    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.IE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;
    if EDSCR.ERR == '1' then return value;              // Error flag set: no side-effects
    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then
      return value;
    end if;
    if underrun then
      EDSCR.TXU = '1';  EDSCR.ERR = '1';              // Underrun condition: block side-effects
      return value;                                   // Return UNKNOWN
    end if;
    EDSCR.TXfull = '0';
    if Halted() && EDSCR.MA == '1' then
      EDSCR.IE = '0';                                // See comments in EDITR[] (external write)
    end if;
    if !UsingAArch32() then ExecuteA64(0xB8404401<31:0>);               // A64 "LDR W1,[X0],#4"
      else ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);      // T32 "LDR R1,[R0],#4"
        // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
          EDSCR.TXfull = bit UNKNOWN;
          DBGDTRTX_EL0 = bits(64) UNKNOWN;
        else
          if !UsingAArch32() then ExecuteA64(0xD5130501<31:0>);           // A64 "MSR DBGDTRTX_EL0,X1"
            else ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MSR DBGDTRTXint,R1"
              // "MSR DBGDTRTX_EL0,X1" calls DBGDTR_EL0[] (write) which sets TXfull.
              assert EDSCR.TXfull == '1';
              if !UsingAArch32() then
                X[1, 64] = bits(64) UNKNOWN;
              else
                R[1] = bits(32) UNKNOWN;
          end if;
          EDSCR.IE = '1';                                // See comments in EDITR[] (external write)
          return value;
        end if;
    end if;
    // DBGDTRTX_EL0[] (external write)
    // ==============================
    DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;  // Software lock locked: ignore write
      DTRTX = value;
      return;
shared/debug/dccanditr/DBGDTR_EL0

// DBGDTR_EL0[] (write)
// ====================
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value_in

bits(N) value = value_in;
// For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
// For MSR DBGDTR_EL0,<Xt>   N=64, value=X[t]<63:0>
assert N IN {32,64};
if EDSCR.TXfull == '1' then
  value = bits(N) UNKNOWN;
// On a 64-bit write, implement a half-duplex channel
if N == 64 then DTRRX = value<63:32>;
DTRTX = value<31:0>;        // 32-bit or 64-bit write
EDSCR.TXfull = '1';
return;

// DBGDTR_EL0[] (read)
// ===================
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[

  // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
  // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
  assert N IN {32,64};
  bits(N) result;
  if EDSCR.RXfull == '0' then
    result = bits(N) UNKNOWN;
  else
    // On a 64-bit read, implement a half-duplex channel
    // NOTE: the word order is reversed on reads with regards to writes
    if N == 64 then result<63:32> = DTRTX;
    result<31:0> = DTRRX;
    EDSCR.RXfull = '0';
  return result;

shared/debug/dccanditr/DTR

bits(32) DTRRX;
bits(32) DTRTX;

shared/debug/dccanditr/EDITR

// EDITR[] (external write)
// ========================
// Called on writes to debug register 0x084.

EDITR[boolean memory_mapped] = bits(32) value

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
  IMPLEMENTATION_DEFINED "generate error response";
  return;
if EDSCR.ERR == '1' then return; // Error flag set: ignore write
if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write
if !Halted() then return; // Non-debug state: ignore write
if EDSCR.ITE == '0' || EDSCR.MA == '1' then
  EDSCR.ITO = '1';
  EDSCR.ERR = '1'; // Overrun condition: block write
  return;
// ITE indicates whether the processor is ready to accept another instruction; the processor
// may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
// is no indication that the pipeline is empty (all instructions have completed). In this
// pseudocode, the assumption is that only one instruction can be executed at a time,
// meaning ITE acts like "InstrCompl".
EDSCR.ITE = '0';
if !UsingAArch32() then
  ExecuteA64(value);
else
  ExecuteT32(value<15:0>/*hw1*/, value<31:16> /*hw2*/);
EDSCR.ITE = '1';
return;

shared/debug/halting/DCPSInstruction

// DCPSInstruction()
// =============
// Operation of the DCPS instruction in Debug state
DCPSInstruction(bits(2) target_el)
  SynchronizeContext();
  bits(2) handle_el;
  case target_el of
    when EL1
      if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then
        handle_el = PSTATE.EL;
      elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        UNDEFINED;
      else
        handle_el = EL1;
    when EL2
      if !HaveEL(EL2) then
        UNDEFINED;
      elsif PSTATE.EL == EL3 && !UsingAArch32() then
        handle_el = EL3;
      elseif !IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure then
        UNDEFINED;
      else
        handle_el = EL2;
    when EL3
      if EDSCR.SDD == '1' || !HaveEL(EL3) then
        UNDEFINED;
      else
        handle_el = EL3;
    otherwise
      Unreachable();
  from_secure = CurrentSecurityState() == SS_Secure;
  if ELUsingAArch32(handle_el) then
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    assert UsingAArch32(); // Cannot move from AArch64 to AArch32
  case handle_el of
    when EL1
      AArch32.WriteMode(M32_Svc);
      if HavePANExt() && SCTLR.SPAN == '0' then
        PSTATE.PAN = '1';
    when EL2
      AArch32.WriteMode(M32_Hyp);
    when EL3
      AArch32.WriteMode(M32_Monitor);
      if HavePANExt() then
if !from_secure then
  PSTATE.PAN = '0';
elsif SCTLR.SPAN == '0' then
  PSTATE.PAN = '1';
if handle_el == EL2 then
  ELR_hyp = bits(32) UNKNOWN;  HSR = bits(32) UNKNOWN;
else
  LR = bits(32) UNKNOWN;
  SPSR[] = bits(32) UNKNOWN;
  PSTATE.E = SCTLR[].EE;
  DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
else                                               // Targeting AArch64
  from_32 = UsingAArch32();
  if from_32 then AArch64.MaybeZeroRegisterUppers();
  if from_32 && HaveSME() && PSTATE.SM == '1' then
    ResetsVEState();
  else
    MaybeZeroSVEUppers(target_el);
    PSTATE.nRW = '0';  PSTATE.SP = '1';  PSTATE.EL = handle_el;
    if HavePANExt() && ((handle_el == EL1 && SCTLR_EL1.SPAN == '0') ||
      (handle_el == EL2 && HCR_EL2.E2H == '1' &&
       HCR_EL2.TGE == '1' && SCTLR_EL2.SPAN == '0')) then
      PSTATE.PAN = '1';
    ELR[] = bits(64) UNKNOWN;  SPSR[] = bits(64) UNKNOWN;  ESR[] = bits(64) UNKNOWN;
    else
      PSTATE.PAN = '1';
      DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(64) UNKNOWN;
      if HaveUAOExt() then PSTATE.UAO = '0';
      if HaveMTEExt() then PSTATE.TCO = '1';
  endif
  PSTATE.E = SCTLR[].EE;
  DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
endif
UpdateEDSCRFields();                           // Update EDSR PE state flags
sync_errors = haveIESB() && SCTLR[].IESB == '1';
if HaveDoubleFaultExt() && !UsingAArch32() then
  sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
// SCTLR[].IESB might be ignored in Debug state.
if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
  sync_errors = FALSE;
if sync_errors then
  SynchronizeErrors();
return;

shared/debug/halting/DRPSInstruction

// DRPSInstruction()
// ===============
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

  SynchronizeContext();
  sync_errors = haveIESB() && SCTLR[].IESB == '1';
  if haveDoubleFaultExt() && !UsingAArch32() then
    sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    // SCTLR[].IESB might be ignored in Debug state.
  if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
    sync_errors = FALSE;
  if sync_errors then
    SynchronizeErrors();
  DebugRestorePSR();
  return;
shared/debug/halting/DebugHalt

constant bits(6) DebugHalt_Breakpoint = '000111';
constant bits(6) DebugHalt_EDBGRQ = '010011';
constant bits(6) DebugHalt_Step_Normal = '011011';
constant bits(6) DebugHalt_Step_Exclusive = '011111';
constant bits(6) DebugHalt_OSUnlockCatch = '100011';
constant bits(6) DebugHalt_ResetCatch = '100111';
constant bits(6) DebugHalt_Watchpoint = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess = '110011';
constant bits(6) DebugHalt_ExceptionCatch = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';

shared/debug/halting/DebugRestorePSR

// DebugRestorePSR()
// ================

DebugRestorePSR()

// PSTATE.<N,Z,C,V,Q,GE,SS,D,A,I,F> are not observable and ignored in Debug state, so
// behave as if UNKNOWN.
if UsingAArch32() then
  bits(32) spsr = SPSR[];
  SetPSTATEFromPSR(spsr);
  PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
  // In AArch32, all instructions are T32 and unconditional.
  PSTATE.IT = '00000000';  PSTATE.T = '1';        // PSTATE.J is RES0
  DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
else
  bits(64) spsr = SPSR[];
  SetPSTATEFromPSR(spsr);
  PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
  DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(64) UNKNOWN;
  UpdateEDSCRFields();                                // Update EDSCR PE state flags

shared/debug/halting/DisableITRAndResumeInstructionPrefetch

DisableITRAndResumeInstructionPrefetch();

shared/debug/halting/ExecuteA64

// Execute an A64 instruction in Debug state.
ExecuteA64(bits(32) instr);

shared/debug/halting/ExecuteT32

// Execute a T32 instruction in Debug state.
ExecuteT32(bits(16) hw1, bits(16) hw2);

shared/debug/halting/ExitDebugState

// ExitDebugState()
// ===============

ExitDebugState()

  assert Halted();
  SynchronizeContext();
// Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
// detect that the PE has restarted.
EDSCR.STATUS = '000001';  // Signal restarting
// Clear any pending Halting debug events
if Havev8p8Debug() then
  EDESR<3:0> = '0000';
else
  EDESR<2:0> = '000';

bits(64) new_pc;
bits(64) spsr;

if UsingAArch32() then
  new_pc = ZeroExtend(DLR, 64);
  spsr = ZeroExtend(DSPSR, 64);
else
  new_pc = DLR_EL0;
  spsr = DSPSR_EL0;

boolean illegal_psr_state = IllegalExceptionReturn(spsr);
// If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
if UsingAArch32() then
  SetPSTATEFromPSR(spsr<31:0>);
else
  SetPSTATEFromPSR(spsr);

boolean branch_conditional = FALSE;
if UsingAArch32() then
  if ConstrainUnpredictableBool(Unpredictable_RESTARTALIGNPC) then new_pc<0> = '0';
  // AArch32 branch
  BranchTo(new_pc<31:0>, BranchType_DBGEXIT, branch_conditional);
else
  // If targeting AArch32 then PC[63:32,1:0] might be set to UNKNOWN.
  if illegal_psr_state && spsr<4> == '1' then
    new_pc<63:32> = bits(32) UNKNOWN;
    new_pc<1:0> = bits(2) UNKNOWN;
  if HaveBRBExt() then
    BRBEDebugStateExit(new_pc);
  // A type of branch that is never predicted
  BranchTo(new_pc, BranchType_DBGEXIT, branch_conditional);

(EDSCR.STATUS,EDPRSR.SDR) = ('000010','1');        // Atomically signal restarted
UpdateEDSCRFields();                               // Stop signalling PE state
DisableITRAndResumeInstructionPrefetch();
return;

shared/debug/halting/Halt

// Halt()
// ======
Halt(bits(6) reason)
  boolean is_async = FALSE;
  Halt(reason, is_async);

// Halt()
// ======
Halt(bits(6) reason, boolean is_async)

  CTI_SignalEvent(CrossTriggerIn_CrossHalt); // Trigger other cores to halt

    bits(64) preferred_restart_address = ThisInstrAddr(64);
    bits(32) spsr_32;
    bits(64) spsr_64;
    if UsingAArch32() then
spsr_32 = GetPSRFromPSTATE(DebugState, 32);
else
spsr_64 = GetPSRFromPSTATE(DebugState, 64);

if (HaveBTIExt() && !is_async && !(reason IN {DebugHalt_Step_Normal, DebugHalt_Step_Exclusive,
DebugHalt_Step_NoSyndrome, DebugHalt_Breakpoint, DebugHalt_HaltInstruction})
&&
ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE)) then
if UsingAArch32() then
spsr_32<11:10> = '00';
else
spsr_64<11:10> = '00';
endif

if UsingAArch32() then
DLR = preferred_restart_address<31:0>;
DSPSR = spsr_32;
else
DLR_EL0 = preferred_restart_address;
DSPSR_EL0 = spsr_64;
endif

EDSCR.ITE = '1';
EDSCR.ITO = '0';
if HaveRME() then
EDSCR.SDD = if ExternalRootInvasiveDebugEnabled() then '0' else '1';
elsif CurrentSecurityState() == SS_Secure then
EDSCR.SDD = '0';
// If entered in Secure state, allow debug
elsif HaveEL(EL3) then
EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
else
EDSCR.SDD = '1';
// Otherwise EDSCR.SDD is RESI
EDSCR.MA = '0';

// In Debug state:
// * PSTATE.{SS,SSBS,D,A,I,F} are not observable and ignored so behave-as-if UNKNOWN.
// * PSTATE.{N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT} are also not observable, but since these
// are not changed on exception entry, this function also leaves them unchanged.
// * PSTATE.{IT,T} are ignored.
// * PSTATE.IL is ignored and behave-as-if 0.
// * PSTATE.BTYPE is ignored and behave-as-if 0.
// * PSTATE.TCO is set 1.
// * PSTATE.{UAO,PAN} are observable and not changed on entry into Debug state.
if UsingAArch32() then
PSTATE.<IT,SS,SSBS,A,I,F>T = bits(14) UNKNOWN;
else
PSTATE.<SS,SSBS,D,A,I,F> = bits(6) UNKNOWN;
PSTATE.TCO = '1';
PSTATE.BTYPE = '00';
PSTATE.IL = '0';
StopInstructionPrefetchAndEnableITR();
EDSCR.STATUS = reason; // Signal entered Debug state
UpdateEDSCRFields(); // Update EDSCR PE state flags.
return;
endif

shared/debug/halting/HaltOnBreakpointOrWatchpoint

// HaltOnBreakpointOrWatchpoint()
// ==============================
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';
shared/debug/halting/Halted

// Halted()
// ========

boolean Halted()
return !(EDSCR.STATUS IN {'000001', '000010'}); // Halted

shared/debug/halting/HaltingAllowed

// HaltingAllowed()
// ================
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
if Halted() || DoubleLockStatus() then
return FALSE;
ss = CurrentSecurityState();
case ss of
  when SS_NonSecure return ExternalInvasiveDebugEnabled();
  when SS_Secure return ExternalSecureInvasiveDebugEnabled();
  when SS_Root return ExternalRootInvasiveDebugEnabled();
  when SS_Realm return ExternalRealmInvasiveDebugEnabled();

shared/debug/halting/Restarting

// Restarting()
// ===========

boolean Restarting()
return EDSCR.STATUS == '000001'; // Restarting

shared/debug/halting/StopInstructionPrefetchAndEnableITR

StopInstructionPrefetchAndEnableITR();

shared/debug/halting/UpdateEDSCRFields

// UpdateEDSCRFields()
// ===================
// Update EDSCR PE state fields

UpdateEDSCRFields()
if !Halted() then
EDSCR.EL = '00';
if HaveRME() then
  EDSCR.<NSE,NS> = bits(2) UNKNOWN;
else
  EDSCR.NS = bit UNKNOWN;
else
  EDSCR.EL = PSTATE.EL;
ss = CurrentSecurityState();
if HaveRME() then
case ss of
  when SS_Secure EDSCR.<NSE,NS> = '00';
  when SS_NonSecure EDSCR.<NSE,NS> = '01';
  when SS_Root EDSCR.<NSE,NS> = '10';
  when SS_Realm EDSCR.<NSE,NS> = '11';
else
    EDSR.NS = if ss == SS_Secure then '0' else '1';

bits(4) RW;
RW<1> = if ELUsingAArch32(EL1) then '0' else '1';
if PSTATE.EL != EL0 then
    RW<0> = RW<1>;
else
    RW<0> = if UsingAArch32() then '0' else '1';
if !HaveEL(EL2) || (HaveEL(EL3) && SCR_GEN[].NS == '0' && !IsSecureEL2Enabled()) then
    RW<2> = RW<1>;
else
    RW<2> = if ELUsingAArch32(EL2) then '0' else '1';
if !HaveEL(EL3) then
    RW<3> = RW<2>;
else
    RW<3> = if ELUsingAArch32(EL3) then '0' else '1';

// The least-significant bits of EDSR.RW are UNKNOWN if any higher EL is using AArch32.
if RW<3> == '0' then RW<2:0> = bits(3) UNKNOWN;
elsif RW<2> == '0' then RW<1:0> = bits(2) UNKNOWN;
elsif RW<1> == '0' then RW<0> = bit UNKNOWN;
EDSR.RW = RW;
return;

shared/debug/haltingevents/CheckExceptionCatch

// CheckExceptionCatch()
// ================
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch(boolean exception_entry)
// Called after an exception entry or exit, that is, such that the Security state
// and PSTATE.EL are correct for the exception target. When FEAT_Debugv8p2
// is not implemented, this function might also be called at any time.
ss = SecurityStateAtEL(PSTATE.EL);
integer base;

case ss of
    when SS_Secure    base = 0;
    when SS_NonSecure base = 4;
    when SS_Realm     base = 16;
    when SS_Root      base = 0;
if HaltingAllowed() then
    boolean halt;
    if HaveExtendedECDebugEvents() then
        exception_exit = !exception_entry;
        increment = if ss == SS_Realm then 4 else 8;
        ctrl = EDECCR<UInt(PSTATE.EL) + base + increment>:EDECCR<UInt(PSTATE.EL) + base>;
        case ctrl of
            when '00'  halt = FALSE;
            when '01'  halt = TRUE;
            when '10'  halt = (exception_exit == TRUE);
            when '11'  halt = (exception_entry == TRUE);
        else
            halt = (EDECCR<UInt(PSTATE.EL) + base> == '1');
        endif
    if halt then
        if Havev8p8Debug() && exception_entry then
            EDESR.EC = '1';
        else
            Halt(DebugHalt_ExceptionCatch);
    endif
endif

shared/debug/haltingevents/CheckHaltingStep

// CheckHaltingStep()
// ====================
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep(boolean is_async)
if HaltingAllowed() && EDESR.SS == '1' then
    if !HaltingStep_DidNotStep() then
        Halt(DebugHalt_Step_NoSyndrome, is_async);
    elsif HaltingStep_SteppedEX() then
        Halt(DebugHalt_Step_Exclusive, is_async);
    else
        Halt(DebugHalt_Step_Normal, is_async);

shared/debug/haltingevents/CheckOSUnlockCatch

// CheckOSUnlockCatch()
// ====================
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
if ((HaveDoPD() && CTIDEVCTL.OSUCE == '1') || (!HaveDoPD() && EDECR.OSUCE == '1')) then
    if !Halted() then EDESR.OSUC = '1';

shared/debug/haltingevents/CheckPendingExceptionCatch

// CheckPendingExceptionCatch()
// =============================
// Check whether EDESR.EC has been set by an Exception Catch debug event.

CheckPendingExceptionCatch(boolean is_async)
if Havev8p8Debug() && HaltingAllowed() && EDESR.EC == '1' then
    Halt(DebugHalt_ExceptionCatch, is_async);

shared/debug/haltingevents/CheckPendingOSUnlockCatch

// CheckPendingOSUnlockCatch()
// ===========================
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
if HaltingAllowed() && EDESR.OSUC == '1' then
    boolean is_async = TRUE;
    Halt(DebugHalt_OSUnlockCatch, is_async);

shared/debug/haltingevents/CheckPendingResetCatch

// CheckPendingResetCatch()
// ========================
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
if HaltingAllowed() && EDESR.RC == '1' then
    boolean is_async = TRUE;
    Halt(DebugHalt_ResetCatch, is_async);
shared/debug/haltingevents/CheckResetCatch

// CheckResetCatch()
// =================
// Called after reset

CheckResetCatch()
  if (HaveDoPD() && CTIDEVCTL.RCE == '1') || (!HaveDoPD() && EDECR.RCE == '1') then
    EDESR.RC = '1';
  // If halting is allowed then halt immediately
  if HaltingAllowed() then Halt(DebugHalt_ResetCatch);

shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

// CheckSoftwareAccessToDebugRegisters()
// =====================================
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
  os_lock = (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK);
  if HaltingAllowed() && EDSCR.TDA == '1' && os_lock == '0' then
    Halt(DebugHalt_SoftwareAccess);

shared/debug/haltingevents/ExternalDebugRequest

// ExternalDebugRequest()
// ======================

ExternalDebugRequest()
  if HaltingAllowed() then
    boolean is_async = TRUE;
    Halt(DebugHalt_EDBGRQ, is_async);
  // Otherwise the CTI continues to assert the debug request until it is taken.

shared/debug/haltingevents/HaltingStep_DidNotStep

// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.

boolean HaltingStep_DidNotStep();

shared/debug/haltingevents/HaltingStep_SteppedEX

// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.

boolean HaltingStep_SteppedEX();

shared/debug/haltingevents/RunHaltingStep

// RunHaltingStep()
// ================

RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall,
  boolean reset)
  // "exception_generated" is TRUE if the previous instruction generated a synchronous exception
  // or was cancelled by an asynchronous exception.

  // if "exception_generated" is TRUE then "exception_target" is the target of the exception, and
  // "syscall" is TRUE if the exception is a synchronous exception where the preferred return
  // address is the instruction following that which generated the exception.
// "reset" is TRUE if exiting reset state into the highest EL.

if reset then assert !Halted();  // Cannot come out of reset halted
active = EDECR.SS == '1' && !Halted();

if active && reset then  // Coming out of reset with EDECR.SS set
  EDESR.SS = '1';
elsif active && HaltingAllowed() then
  boolean advance;
  if exception_generated && exception_target == EL3 then
    advance = syscall || ExternalSecureInvasiveDebugEnabled();
  else
    advance = TRUE;
  if advance then EDESR.SS = '1';
return;

shared/debug/interrupts/ExternalDebugInterruptsDisabled

// ExternalDebugInterruptsDisabled()
// =================================
// Determine whether EDSCR disables interrupts routed to 'target'.

boolean ExternalDebugInterruptsDisabled(bits(2) target)
boolean int_dis;
SecurityState ss = SecurityStateAtEL(target);
if Havev8p4Debug() then
  if EDSCR.INTdis[0] == '1' then
    case ss of
      when SS_NonSecure int_dis = ExternalInvasiveDebugEnabled();
      when SS_Secure int_dis = ExternalSecureInvasiveDebugEnabled();
      when SS_Realm int_dis = ExternalRealmInvasiveDebugEnabled();
      when SS_Root int_dis = ExternalRootInvasiveDebugEnabled();
    else
      int_dis = FALSE;
    end
  else
    case target of
      when EL3
        int_dis = (EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled());
      when EL2
        int_dis = (EDSCR.INTdis IN {'1x'} && ExternalInvasiveDebugEnabled());
      when EL1
        if ss == SS_Secure then
          int_dis = (EDSCR.INTdis IN {'1x'} && ExternalSecureInvasiveDebugEnabled());
        else
          int_dis = (EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled());
        end
    end
  end
return int_dis;

shared/debug/pmu/GetNumEventCounters

// GetNumEventCounters()
// =====================
// Returns the number of event counters implemented. This is indicated to software at the
// highest Exception level by PMCR.N in AArch32 state, and PMCR_EL0.N in AArch64 state.

integer GetNumEventCounters()
  return integer IMPLEMENTATION_DEFINED "Number of event counters";
shared/debug/pmu/HasElapsed64Cycles
// Returns TRUE if 64 cycles have elapsed between the last count, and FALSE otherwise.
boolean HasElapsed64Cycles();

shared/debug/pmu/PMUCountValue
// PMUCountValue()
// ===================
// Implements the PMU threshold function, if implemented.
// Returns the value to increment event counter 'n' by, if the event it is
// configured to count yields the value 'v' on this cycle.

integer PMUCountValue(integer n, integer v)
if !HavePMUv3TH() then
    return v;

integer T = UInt(PMEVTYPER_EL0[n].TH);

case PMEVTYPER_EL0[n].TC of
    when '000' return (if v != T then v else 0);    // Disabled or not-equal
    when '001' return (if v != T then 1 else 0);    // Not-equal, count
    when '010' return (if v == T then v else 0);    // Equals
    when '011' return (if v == T then 1 else 0);    // Equals, count
    when '100' return (if v >= T then v else 0);    // Greater-than-or-equal
    when '101' return (if v >= T then 1 else 0);    // Greater-than-or-equal, count
    when '110' return (if v <  T then v else 0);    // Less-than
    when '111' return (if v <  T then 1 else 0);    // Less-than, count

shared/debug/pmu/PMUCounterMask
constant integer CYCLE_COUNTER_ID = 31;

// PMUCounterMask()
// ================
// Return bitmask of accessible PMU counters.

bits(32) PMUCounterMask()
integer n;
if UsingAArch32() then
    n = AArch32.GetNumEventCountersAccessible();
else
    n = AArch64.GetNumEventCountersAccessible();
return '1' : ZeroExtend(Ones(n), 31);

shared/debug/pmu/PMUEvent
// PMUEvent()
// ===========
// Generate a PMU event. By default, increment by 1.

PMUEvent(bits(16) event)
    PMUEvent(event, 1);

// PMUEvent()
// ===========
// Accumulate a PMU Event.

PMUEvent(bits(16) event, integer increment)
    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMUEvent(event, increment, idx);
// PMUEvent()
// =========
// Accumulate a PMU Event for a specific event counter.

PMUEvent(bits(16) event, integer increment, integer idx)
if !HavePMUv3() then
    return;
if UsingAArch32() then
    if PMEVTYPER[idx].evtCount == event then
        PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
    else
        if PMEVTYPER_EL0[idx].evtCount == event then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;

shared/debug/pmum/integer
array integer PMUEventAccumulator[0..30];  // Accumulates PMU events for a cycle

shared/debug/samplebasedprofiling/CreatePCSample

// CreatePCSample()
// ================

CreatePCSample()
// In a simple sequential execution of the program, CreatePCSample is executed each time the PE // executes an instruction that can be sampled. An implementation is not constrained such that // reads of EDPCSRlo return the current values of PC, etc.

pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
pc_sample.pc = ThisInstrAddr(64);
pc_sample.el = PSTATE.EL;
pc_sample.rw = if UsingAArch32() then '0' else '1';
pc_sample.ss = CurrentSecurityState();
pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else CONTEXTIDR_EL1<31:0>;
pc_sample.has_el2 = PSTATE.EL != EL3 && EL2Enabled();
if pc_sample.has_el2 then
    if ELUsingAArch32(EL2) then
        pc_sample.vmid = ZeroExtend(VTTBR.VMID, 16);
    elsif !Have16bitVMID() || VTCR_EL2.VS == '0' then
        pc_sample.vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
    else
        pc_sample.vmid = VTTBR_EL2.VMID;
    end
    if (HaveVirtHostExt() || HaveV82Debug()) && !ELUsingAArch32(EL2) then
        pc_sample.contextidr_el2 = CONTEXTIDR_EL2<31:0>;
    else
        pc_sample.contextidr_el2 = bits(32) UNKNOWN;
    pc_sample.el0h = PSTATE.EL == EL0 && IsInHost();
    return;

shared/debug/samplebasedprofiling/EDPCSRlo

// EDPCSRlo[] (read)
// ================

bits(32) EDPCSRlo[boolean memory_mapped]
if EDPRSR<6:5,0> != '001' then /* Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return bits(32) UNKNOWN;
// The Software lock is OPTIONAL.
update = !memory_mapped || EDLSR.SLK == '0';        // Software locked: no side-effects

bits(32) sample;
if pc_sample.valid then
    sample = pc_sample.pc<31:0>;
if update then
    if HaveVirtHostExt() && EDSCR.SC2 == '1' then
        EDPCSRhi.PC = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
        EDPCSRhi.EL = pc_sample.el;
        EDPCSRhi.NS = (if pc_sample.ss == SS_Secure then '0' else '1');
    else
        EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
    EDVICHR = pc_sample.contextidr;
    if (HaveVirtHostExt() || HaveV82Debug()) && EDSCR.SC2 == '1' then
        EDVIDSR = (if pc_sample.has_el2 then pc_sample.contextidr_el2
                   else bits(32) UNKNOWN);
    else
        EDVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0}
                          then pc_sample.vmid else Zeros(16));
        EDVIDSR.NS = (if pc_sample.ss == SS_Secure then '0' else '1');
        EDVIDSR.E2 = (if pc_sample.el == EL2 then '1' else '0');
        EDVIDSR.E3 = (if pc_sample.el == EL3 then '1' else '0') AND pc_sample.rw;
        // The conditions for setting HV are not specified if PCSRhi is zero.
        // An example implementation may be "pc_sample.rw".
        EDVIDSR.HV = (if !IsZero(EDPCSRhi) then '1' else bit IMPLEMENTATION_DEFINED "0 or 1");
    else
        sample = Ones(32);
        if update then
            EDPCSRhi = bits(32) UNKNOWN;
            EDVICHR = bits(32) UNKNOWN;
            EDVIDSR = bits(32) UNKNOWN;

        return sample;

shared/debug/samplebasedprofiling/PCSample

type PCSample is (
    boolean valid,
    bits(64) pc,  // bits(64) pc
    bits(2) el,   // bits(2) el
    bit rw,       // bit rw,
    SecurityState ss,  // SecurityState ss,
    boolean has_el2, // boolean has_el2,
    bits(32) contextidr, // bits(32) contextidr
    bits(32) contextidr_el2,  // bits(32) contextidr_el2
    boolean el0h,        // boolean el0h,
    bits(16) vmid        // bits(16) vmid
)

PCSample pc_sample;

shared/debug/samplebasedprofiling/PMPCSR

// PMPCSR[] (read)
// ----------------

bits(32) PMPCSR[boolean memory_mapped]
if EDRS<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return bits(32) UNKNOWN;

// The Software lock is OPTIONAL.
update = !memory_mapped || PMLSR.SLK == '0';        // Software locked: no side-effects
bits(32) sample;
if pc_sample.valid then
    sample = pc_sample.pc<31:0>;
if update then
    PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
    PMPCSR.EL = pc_sample.el;
if HaveRME() then
    case pc_sample.ss of
    when SS_Secure
        PMPCSR.NSE = '0'; PMPCSR.NS = '0';
    when SS_NonSecure
        PMPCSR.NSE = '0'; PMPCSR.NS = '1';
    when SS_Root
        PMPCSR.NSE = '1'; PMPCSR.NS = '0';
    when SS_Realm
        PMPCSR.NSE = '1'; PMPCSR.NS = '1';
    else
        PMPCSR.NS = (if pc_sample.ss == SS_Secure then '0' else '1');
    endcase
    PMCID1SR = pc_sample.contextidr;
    PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else bits(32) UNKNOWN;
    PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0} && !pc_sample.el0h then pc_sample.vmid else bits(16) UNKNOWN);
else
    sample = Ones(32);
if update then
    PMPCSR<55:32> = bits(24) UNKNOWN;
    PMPCSR.EL = bits(2) UNKNOWN;
    PMPCSR.NS = bit UNKNOWN;
    PMCID1SR = bits(32) UNKNOWN;
    PMCID2SR = bits(32) UNKNOWN;
    PMVIDSR.VMID = bits(16) UNKNOWN;
return sample;

shared/debug/softwarestep/CheckSoftwareStep

    // CheckSoftwareStep()
    // =============
    // Take a Software Step exception if in the active-pending state
    CheckSoftwareStep()

        // Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
        // AArch32 state. However, because Software Step is only active when the debug target Exception
        // level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
        step_enabled = !ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() && MDSCR_EL1.SS
        == '1';
        if step_enabled && PSTATE.SS == '0' then
            AArch64.SoftwareStepException();

shared/debug/softwarestep/DebugExceptionReturnSS

    // DebugExceptionReturnSS()
    // ================
    // Returns value to write to PSTATE.SS on an exception return or Debug state exit.
    bit DebugExceptionReturnSS(bits(N) spsr)
        if UsingAArch32() then
            assert N == 32;
else
    assert N == 64;

assert Halted() || Restarting() || PSTATE.EL != EL0;

boolean enabled_at_source;
if Restarting() then
    enabled_at_source = FALSE;
elsif UsingAArch32() then
    enabled_at_source = AArch32.GenerateDebugExceptions();
else
    enabled_at_source = AArch64.GenerateDebugExceptions();

boolean valid;
bits(2) dest_el;
if IllegalExceptionReturn(spsr) then
    dest_el = PSTATE.EL;
else
    (valid, dest_el) = ELFromSPSR(spsr); assert valid;

dest_ss = SecurityStateAtEL(dest_el);
bit mask;
boolean enabled_at_dest;
dest_using_32 = (if dest_el == EL0 then spsr<4> == '1' else ELUsingAArch32(dest_el));
if dest_using_32 then
    enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest_el, dest_ss);
else
    mask = spsr<9>;
    enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest_el, dest_ss, mask);

 Eld = DebugTargetFrom(dest_ss);
bit SS_bit;
if !ELUsingAArch32(Eld) && MDSCR_EL1.SS == '1' && enabled_at_source && enabled_at_dest then
    SS_bit = spsr<21>;
else
    SS_bit = '0';

return SS_bit;

// SSAdvance()
// ===========
// Advance the Software Step state machine.

SSAdvance()

// A simpler implementation of this function just clears PSTATE.SS to zero regardless of the // current Software Step state machine. However, this check is made to illustrate that the // processor only needs to consider advancing the state machine from the active-not-pending // state.
target = DebugTarget();
step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
active_not_pending = step_enabled && PSTATE.SS == '1';
if active_not_pending then PSTATE.SS = '0';
return;

// Returns TRUE if the previously executed instruction was executed in the // inactive state, that is, if it was not itself stepped. // Might return TRUE or FALSE if the previously executed instruction was an ISB
boolean SoftwareStep_DidNotStep();

shared/debug/softwarestep/SoftwareStep_SteppedEX

// Returns a value that describes the previously executed instruction. The
// result is valid only if SoftwareStep_DidNotStep() returns FALSE.
// Might return TRUE or FALSE if the instruction was an AArch32 LDREX or LDAEX
// that failed its condition code test. Otherwise returns TRUE if the
// instruction was a Load-Exclusive class instruction, and FALSE if the
// instruction was not a Load-Exclusive class instruction.
boolean SoftwareStep_SteppedEX();

J1.3.2 shared/exceptions

This section includes the following pseudocode functions:
• shared/exceptions/exceptions/ConditionSyndrome.
• shared/exceptions/exceptions/Exception on page J1-11383.
• shared/exceptions/exceptions/ExceptionRecord on page J1-11383.
• shared/exceptions/exceptions/ExceptionSyndrome on page J1-11384.

shared/exceptions/exceptions/ConditionSyndrome

// ConditionSyndrome()
// =============
// Return CV and COND fields of instruction syndrome
bits(5) ConditionSyndrome()

bits(5) syndrome;

if UsingAArch32() then
  cond = AArch32.CurrentCond();
  if PSTATE.T == '0' then             // A32
    syndrome<4> = '1';
    // A conditional A32 instruction that is known to pass its condition code check
    // can be presented either with COND set to 0xE, the value for unconditional, or
    // the COND value held in the instruction.
    if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable_ESRCONDPASS) then
      syndrome<3:0> = '1110';
    else
      syndrome<3:0> = cond;
  else                                // T32
    // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
    // * CV set to 0 and COND is set to an UNKNOWN value
    // * CV set to 1 and COND is set to the condition code for the condition that
    //   applied to the instruction.
    if boolean IMPLEMENTATION_DEFINED "Condition valid for trapped T32" then
      syndrome<4> = '1';
      syndrome<3:0> = cond;
    else
      syndrome<4> = '0';
      syndrome<3:0> = bits(4) UNKNOWN;
  else
    syndrome<4> = '1';
    syndrome<3:0> = '1110';
return syndrome;
shared/exceptions/exceptions/Exception

enumeration Exception {Exception_Uncategorized, // Uncategorized or unknown reason
    Exception_WFxTrap,                 // Trapped WFI or WFE instruction
    Exception_CP15RTRTrap,             // Trapped AArch32 MCR or MRC access,
coproc=0b1111
    Exception_CP15RTTrap,              // Trapped AArch32 MCR or MRC access,
coproc=0b1111
    Exception_CP14RTTrap,              // Trapped AArch32 MCR or MRC access,
coproc=0b1110
    Exception_CP14DTTrap,              // Trapped AArch32 LDC or STC access,
coproc=0b1110
    Exception_CP14RRTTrap,             // Trapped AArch32 MRRC access, coproc=0b1110
    Exception_AdvSIMDFPAccessTrap,     // HCPTR-trapped access to SIMD or FP
    Exception_FPIDTrap,                // Trapped access to SIMD or FP ID register
    Exception_LDST64BTrap,             // Trapped access to ST64BV, ST64BV0, ST64B and
LD64B
    // Trapped BXJ instruction not supported in Armv8
    Exception_PACPTrap,                // Trapped invalid PAC use
    Exception_IllegalState,            // Illegal Execution state
    Exception_SupervisorCall,          // Supervisor Call
    Exception_HypervisorCall,          // Monitor Call or Trapped SMC instruction
    Exception_SystemRegisterTrap,       // Trapped MRS or MSR System register access
    Exception_ERetTrap,                // Trapped invalid ERET use
    Exception_InstructionAbort,         // Instruction Abort or Prefetch Abort
    Exception_PCAAlignment,            // PC alignment fault
    Exception_DataAbort,               // Data Abort
    Exception_NV2DataAbort,            // Data abort at EL1 reported as being from EL2
    Exception_PACFail,                 // PAC Authentication failure
    Exception_SPAlignment,             // SP alignment fault
    Exception_FFPtrappedException,      // IEEE trapped FP exception
    Exception_SError,                  // SError interrupt
    Exception_Breakpoint,              // (Hardware) Breakpoint
    Exception_SoftwareStep,            // Software Step
    Exception_Watchpoint,              // Watchpoint
    Exception_NV2Watchpoint,           // Watchpoint at EL1 reported as being from EL2
    Exception_SoftwareBreakpoint,      // Software Breakpoint Instruction
    Exception_VectorCatch,             // AArch32 Vector Catch
    Exception_IRQ,                     // IRQ interrupt
    Exception_SVEAccessTrap,           // HCPTR trapped access to SVE
    Exception_SMEAccessTrap,           // HCPTR trapped access to SME
    Exception_TSTARTAccessTrap,        // Trapped TSTART access
    Exception_GPC,                     // Granule protection check
    Exception_BranchTarget,            // Branch Target Identification
    Exception_MemCpyMemSet,            // Exception from a CPY* or SET* instruction
    Exception_FIQ);                    // FIQ interrupt

shared/exceptions/exceptions/ExceptionRecord

type ExceptionRecord is (Exception exceptype, // Exception class
    bits(25) syndrome,                  // Syndrome record
    bits(5) syndrome2,                 // ST64BV(0) return value register specifier
    FullAddress paddress,              // Physical fault address
    bits(64) vaddress,                 // Virtual fault address
    boolean ipvvalid,                  // Validity of Intermediate Physical fault address
    bit NS,                            // Intermediate Physical fault address space
    bits(52) ipaddress,                // Intermediate Physical fault address
    boolean trappedsyscallinst)        // Trapped SVC or SMC instruction
shared/exceptions/exceptions/ExceptionSyndrome

// ExceptionSyndrome()
// ===================
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception exceptype)
{
    ExceptionRecord r;
    r.exceptype = exceptype;
    // Initialize all other fields
    r.syndrome  = Zeros(25);
    r.syndrome2 = Zeros(5);
    r.vaddress  = Zeros(64);
    r.ipavalid  = FALSE;
    r.NS        = '0';
    r.ipaddress = Zeros(52);
    r.paddress.paspace = PASpace UNKNOWN;
    r.paddress.address = bits(52) UNKNOWN;
    r.trappedsyscallinst = FALSE;
    return r;
}

J1.3.3 shared/functions

This section includes the following pseudocode functions:

- shared/functions/aborts/EncodeLDFSC on page J1-11394.
- shared/functions/aborts/IPAValid on page J1-11394.
- shared/functions/aborts/IsAsyncAbort on page J1-11395.
- shared/functions/aborts/IsDebugException on page J1-11395.
- shared/functions/aborts/IsSecondStage on page J1-11396.
- shared/functions/aborts/LSInstructionSyndrome on page J1-11397.
- shared/functions/cache/CACHE_OP on page J1-11397.
- shared/functions/cache/CPASAtPAS on page J1-11397.
- shared/functions/cache/CPASAtSecurityState on page J1-11398.
- shared/functions/cache/CachePASpace on page J1-11398.
- shared/functions/cache/CacheRecord on page J1-11398.
- shared/functions/cache/CacheType on page J1-11399.
- shared/functions/cache/DCInstNeedsTranslation on page J1-11399.
- shared/functions/cache/DecodeSW on page J1-11399.
- shared/functions/cache/GetCacheInfo on page J1-11399.
- shared/functions/cache/ICInstNeedsTranslation on page J1-11400.
- shared/functions/common/ASR on page J1-11400.
- shared/functions/common/ASR_C on page J1-11400.
- shared/functions/common/Abs on page J1-11400.
- shared/functions/common/Align on page J1-11400.
- shared/functions/common/BitCount on page J1-11401.
- shared/functions/common/CountLeadingSignBits on page J1-11401.
shared/functions/common/CountLeadingZeroBits on page J1-11401.
shared/functions/common/Elem on page J1-11401.
shared/functions/common/Extend on page J1-11401.
shared/functions/common/HighestSetBit on page J1-11402.
shared/functions/common/Int on page J1-11402.
shared/functions/common/IsOnes on page J1-11402.
shared/functions/common/IsZero on page J1-11402.
shared/functions/common/IsZeroBit on page J1-11402.
shared/functions/common/LSL on page J1-11402.
shared/functions/common/LSL_C on page J1-11403.
shared/functions/common/LSR on page J1-11403.
shared/functions/common/LSR_C on page J1-11403.
shared/functions/common/LowestSetBit on page J1-11403.
shared/functions/common/Max on page J1-11403.
shared/functions/common/Min on page J1-11404.
shared/functions/common/Ones on page J1-11404.
shared/functions/common/ROR on page J1-11404.
shared/functions/common/ROR_C on page J1-11404.
shared/functions/common/Replicate on page J1-11405.
shared/functions/common/RoundDown on page J1-11405.
shared/functions/common/RoundTowardsZero on page J1-11405.
shared/functions/common/RoundUp on page J1-11405.
shared/functions/common/SInt on page J1-11405.
shared/functions/common/SignExtend on page J1-11405.
shared/functions/common/Split64to32 on page J1-11405.
shared/functions/common/UInt on page J1-11405.
shared/functions/common/ZeroExtend on page J1-11406.
shared/functions/common/Zeros on page J1-11406.
shared/functions/counters/AArch32.CheckTimerConditions on page J1-11406.
shared/functions/counters/AArch64.CheckTimerConditions on page J1-11407.
shared/functions/counters/GenericCounterTick on page J1-11408.
shared/functions/counters/IsTimerConditionMet on page J1-11408.
shared/functions/counters/PhysicalCount on page J1-11408.
shared/functions/counters/SetEventRegister on page J1-11408.
shared/functions/counters/TestEventCNTP on page J1-11408.
shared/functions/counters/TestEventCNTV on page J1-11409.
shared/functions/crc/BitReverse on page J1-11409.
shared/functions/crc/HaveCRCExt on page J1-11409.
shared/functions/crc/Poly32Mod2 on page J1-11410.
shared/functions/crypto/AESInvMixColumns on page J1-11410.
shared/functions/crypto/AESInvShiftRows on page J1-11410.
shared/functions/crypto/AESInvSubBytes on page J1-11411.
shared/functions/crypto/AESMixColumns on page J1-11411.
shared/functions/crypto/AESShiftRows on page J1-11412.
shared/functions/crypto/AESSubBytes on page J1-11412.
shared/functions/crypto/FFmul02 on page J1-11412.
shared/functions/crypto/FFmul03 on page J1-11413.
shared/functions/crypto/FFmul09 on page J1-11413.
shared/functions/crypto/FFmul0B on page J1-11414.
shared/functions/crypto/FFmul0D on page J1-11414.
• shared/functions/crypto/FFmul0E on page J1-11415.
• shared/functions/crypto/Have4ESExt on page J1-11415.
• shared/functions/crypto/HaveBit128PULLExt on page J1-11415.
• shared/functions/crypto/HaveSHA1Ext on page J1-11415.
• shared/functions/crypto/HaveSHA256Ext on page J1-11416.
• shared/functions/crypto/HaveSHA3Ext on page J1-11416.
• shared/functions/crypto/HaveSHA512Ext on page J1-11416.
• shared/functions/crypto/HaveSM3Ext on page J1-11416.
• shared/functions/crypto/HaveSM4Ext on page J1-11416.
• shared/functions/crypto/ROL on page J1-11417.
• shared/functions/crypto/SHA256hash on page J1-11417.
• shared/functions/crypto/SHAchoose on page J1-11417.
• shared/functions/crypto/SHAhashSIGMA0 on page J1-11417.
• shared/functions/crypto/SHAhashSIGMA1 on page J1-11417.
• shared/functions/crypto/SHAmajority on page J1-11417.
• shared/functions/crypto/SHAparity on page J1-11418.
• shared/functions/crypto/Sbox on page J1-11418.
• shared/functions/exclusive/ClearExclusiveByAddress on page J1-11418.
• shared/functions/exclusive/ClearExclusiveLocal on page J1-11418.
• shared/functions/exclusive/ClearExclusiveMonitors on page J1-11418.
• shared/functions/exclusive/ExclusiveMonitorsStatus on page J1-11418.
• shared/functions/exclusive/IsExclusiveGlobal on page J1-11419.
• shared/functions/exclusive/IsExclusiveLocal on page J1-11419.
• shared/functions/exclusive/MarkExclusiveGlobal on page J1-11419.
• shared/functions/exclusive/MarkExclusiveLocal on page J1-11419.
• shared/functions/exclusive/ProcessorID on page J1-11419.
• shared/functions/extension/AArch32.HaveHPDExt on page J1-11419.
• shared/functions/extension/AArch64.HaveHPDExt on page J1-11419.
• shared/functions/extension/Have16bitVMID on page J1-11419.
• shared/functions/extension/Have52BitIP AAndP ASpaceExt on page J1-11420.
• shared/functions/extension/Have52BitPAExt on page J1-11420.
• shared/functions/extension/Have52BitVAExt on page J1-11420.
• shared/functions/extension/HaveAArch32BF16Ext on page J1-11420.
• shared/functions/extension/HaveAArch32Int8MatMulExt on page J1-11420.
• shared/functions/extension/HaveAccessFlagUpdateExt on page J1-11421.
• shared/functions/extension/HaveAltFP on page J1-11421.
• shared/functions/extension/HaveAtomicExt on page J1-11421.
• shared/functions/extension/HaveBF16Ext on page J1-11421.
• shared/functions/extension/HaveBRBEv1p1 on page J1-11421.
• shared/functions/extension/HaveBRBExt on page J1-11421.
• shared/functions/extension/HaveBTIExt on page J1-11422.
• shared/functions/extension/HaveBlockBBM on page J1-11422.
• shared/functions/extension/HaveCNTSCEExt on page J1-11422.
• shared/functions/extension/HaveCommonNotPrivateTransExt on page J1-11422.
• shared/functions/extension/HaveDGHExt on page J1-11422.
• shared/functions/extension/HaveDITEExt on page J1-11422.
• shared/functions/extension/HaveDOTPExt on page J1-11423.
• shared/functions/extension/HaveDirtyBitModifierExt on page J1-11423.
• shared/functions/extension/HaveDoPD on page J1-11423.
• shared/functions/extension/HaveDoubleFaultExt on page J1-11423.
• shared/functions/extension/HaveDoubleLock on page J1-11423.
• shared/functions/extension/HaveE0PDExt on page J1-11423.
• shared/functions/extension/HaveECVExt on page J1-11424.
• shared/functions/extension/HaveETExt on page J1-11424.
• shared/functions/extension/HaveExtendedCacheSets on page J1-11424.
• shared/functions/extension/HaveExtendedECDebugEvents on page J1-11424.
• shared/functions/extension/HaveExtendedExecuteNeverExt on page J1-11424.
• shared/functions/extension/HaveFCADDExt on page J1-11424.
• shared/functions/extension/HaveFGTExt on page J1-11425.
• shared/functions/extension/HaveFJCVTZSExt on page J1-11425.
• shared/functions/extension/HaveFP16MulNoRoundingToFP32Ext on page J1-11425.
• shared/functions/extension/HaveFeatCMOW on page J1-11425.
• shared/functions/extension/HaveFeatHBC on page J1-11425.
• shared/functions/extension/HaveFeatHCX on page J1-11425.
• shared/functions/extension/HaveFeatHPMN0 on page J1-11426.
• shared/functions/extension/HaveFeatLS64 on page J1-11426.
• shared/functions/extension/HaveFeatLS64_ACCDATA on page J1-11426.
• shared/functions/extension/HaveFeatLS64_V on page J1-11426.
• shared/functions/extension/HaveFeatMOPS on page J1-11426.
• shared/functions/extension/HaveFeatNMI on page J1-11426.
• shared/functions/extension/HaveFeatRPRES on page J1-11427.
• shared/functions/extension/HaveFeatTIDCP1 on page J1-11427.
• shared/functions/extension/HaveFeatWFxT on page J1-11427.
• shared/functions/extension/HaveFeatXS on page J1-11427.
• shared/functions/extension/HaveFlagFormatExt on page J1-11427.
• shared/functions/extension/HaveFlagManipulateExt on page J1-11428.
• shared/functions/extension/HaveFrintExt on page J1-11428.
• shared/functions/extension/HaveGTGExt on page J1-11428.
• shared/functions/extension/HaveHPMDExt on page J1-11428.
• shared/functions/extension/HaveIDSExt on page J1-11428.
• shared/functions/extension/HaveIESB on page J1-11428.
• shared/functions/extension/HaveInt8MatMulExt on page J1-11429.
• shared/functions/extension/HaveLSE2Ext on page J1-11429.
• shared/functions/extension/HaveMPAMExt on page J1-11429.
• shared/functions/extension/HaveMPAMv0p1Ext on page J1-11429.
• shared/functions/extension/HaveMPAMv1p1Ext on page J1-11429.
• shared/functions/extension/HaveMTE2Ext on page J1-11429.
• shared/functions/extension/HaveMTE3Ext on page J1-11430.
• shared/functions/extension/HaveMTEExt on page J1-11430.
• shared/functions/extension/HaveNV2Ext on page J1-11430.
• shared/functions/extension/HaveNVExt on page J1-11430.
• shared/functions/extension/HaveNoSecurePMUDisableOverride on page J1-11430.
• shared/functions/extension/HaveNoninvasiveDebugAuth on page J1-11431.
• shared/functions/extension/HavePAN3Ext on page J1-11431.
• shared/functions/extension/HavePANExt on page J1-11431.
• shared/functions/extension/HavePMUv3 on page J1-11431.
• shared/functions/extension/HavePMUv3TH on page J1-11431.
• shared/functions/extension/HavePMUv3p1 on page J1-11431.
• shared/functions/extension/HavePMUv3p4 on page J1-11432.
• shared/functions/extension/HavePMUv3p5 on page J1-11432.
shared/functions/extension/HavePMUv3p7 on page J1-11432.
shared/functions/extension/HavePageBasedHardwareAttributes on page J1-11432.
shared/functions/extension/HavePrivATExt on page J1-11432.
shared/functions/extension/HaveQRDMLAHExt on page J1-11432.
shared/functions/extension/HaveRASExt on page J1-11432.
shared/functions/extension/HaveRME on page J1-11433.
shared/functions/extension/HaveRNG on page J1-11433.
shared/functions/extension/HaveSBExt on page J1-11433.
shared/functions/extension/HaveSSBSExt on page J1-11433.
shared/functions/extension/HaveSecureEL2Ext on page J1-11433.
shared/functions/extension/HaveSecureExtDebugView on page J1-11433.
shared/functions/extension/HaveSelfHostedTrace on page J1-11434.
shared/functions/extension/HaveSmallTranslationTblExt on page J1-11434.
shared/functions/extension/HaveSoftwareLock on page J1-11434.
shared/functions/extension/HaveStage2MemAttrControl on page J1-11434.
shared/functions/extension/HaveStatisticalProfiling on page J1-11434.
shared/functions/extension/HaveStatisticalProfilingv1p1 on page J1-11435.
shared/functions/extension/HaveStatisticalProfilingv1p2 on page J1-11435.
shared/functions/extension/HaveTME on page J1-11435.
shared/functions/extension/HaveTWEDExt on page J1-11435.
shared/functions/extension/HaveTraceBufferExtension on page J1-11435.
shared/functions/extension/HaveTraceExt on page J1-11435.
shared/functions/extension/HaveTrapLoadStoreMultipleDeviceExt on page J1-11436.
shared/functions/extension/HaveUAOExt on page J1-11436.
shared/functions/extension/HaveV82Debug on page J1-11436.
shared/functions/extension/HaveVirtHostExt on page J1-11436.
shared/functions/extension/Havev8p4Debug on page J1-11436.
shared/functions/extension/Havev8p8Debug on page J1-11436.
shared/functions/extension/InsertIESBBeforeException on page J1-11437.
shared/functions/extension/IsG1ActivityMonitorImplemented on page J1-11437.
shared/functions/extension/IsG1ActivityMonitorOffsetImplemented on page J1-11437.
shared/functions/externalaborts/HandleExternalAbort on page J1-11437.
shared/functions/externalaborts/HandleExternalReadAbort on page J1-11438.
shared/functions/externalaborts/HandleExternalTTWAbort on page J1-11438.
shared/functions/externalaborts/HandleExternalWriteAbort on page J1-11438.
shared/functions/externalaborts/IsExternalAbortTakenSynchronously on page J1-11439.
shared/functions/externalaborts/PendSErrorInterrupt on page J1-11439.
shared/functions/float/bfloat/BFDotAdd on page J1-11439.
shared/functions/float/bfloat/BFMatMulAdd on page J1-11440.
shared/functions/float/bfloat/BFMulAddH on page J1-11440.
shared/functions/float/bfloat/BFMulH on page J1-11440.
shared/functions/float/bfloat/BFNeg on page J1-11441.
shared/functions/float/bfloat/BFRound on page J1-11441.
shared/functions/float/bfloat/BFUnpack on page J1-11442.
shared/functions/float/bfloat/FPAdd_BF16 on page J1-11443.
shared/functions/float/bfloat/FPConvertBF on page J1-11443.
shared/functions/float/bfloat/FPRoundBF on page J1-11444.
shared/functions/float/fpabs/FixedToFP on page J1-11444.
shared/functions/float/fpabs/FPAbs on page J1-1144.
• shared/functions/float/fpadd/FPAdd on page J1-11445.
• shared/functions/float/fpcompare/FPCompare on page J1-11445.
• shared/functions/float/fpcompareeq/FPCompareEQ on page J1-11446.
• shared/functions/float/fpcomparege/FPCompareGE on page J1-11446.
• shared/functions/float/fpcomparegt/FPCompareGT on page J1-11447.
• shared/functions/float/fpconvert/FPConvert on page J1-11447.
• shared/functions/float/fpconvertnan/FPConvertNaN on page J1-11448.
• shared/functions/float/fpdefaultnan/FPDefaultNaN on page J1-11449.
• shared/functions/float/fpdecode/FPDecode on page J1-11449.
• shared/functions/float/fpdecode/FPDecodeRounding on page J1-11449.
• shared/functions/float/fpdiv/FPDiv on page J1-11449.
• shared/functions/float/fpdot/FPDot on page J1-11450.
• shared/functions/float/fpdot/FPDotAdd on page J1-11451.
• shared/functions/float/fpdot/FPDotAdd_ZA on page J1-11451.
• shared/functions/float/fpexcp/FPExc on page J1-11452.
• shared/functions/float/fpinfinity/FPInfinity on page J1-11452.
• shared/functions/float/fpminmuld/FPMatMulAdd on page J1-11452.
• shared/functions/float/fpmax/FPMax on page J1-11452.
• shared/functions/float/fpmaxnormal/FPMaxNormal on page J1-11453.
• shared/functions/float/fpmaxnum/FPMaxNum on page J1-11454.
• shared/functions/float/fpmerge/IsMerging on page J1-11454.
• shared/functions/float/fpmin/FPMin on page J1-11454.
• shared/functions/float/fpminnum/FPMinNum on page J1-11455.
• shared/functions/float/fpmul/FPMul on page J1-11456.
• shared/functions/float/fpmuladd/FPMulAdd on page J1-11456.
• shared/functions/float/fpmuladd/ZA on page J1-11457.
• shared/functions/float/fpmuladdh/FPMulAddH on page J1-11457.
• shared/functions/float/fpmuladdh/FPProcessNaNs3H on page J1-11459.
• shared/functions/float/fpmuls/FPMulX on page J1-11459.
• shared/functions/float/fpmeg/FPNeg on page J1-11460.
• shared/functions/float/fponepointfive/FPOnePointFive on page J1-11460.
• shared/functions/float/fpprocessdenorms/FPProcessDenorm on page J1-11460.
• shared/functions/float/fpprocessdenorms/FPProcessDenorms on page J1-11461.
• shared/functions/float/fpprocessdenorms/FPProcessDenorms3 on page J1-11461.
• shared/functions/float/fpprocessdenorms/FPProcessDenorms4 on page J1-11461.
• shared/functions/float/fpprocessexception/FPProcessException on page J1-11461.
• shared/functions/float/fpprocessnan/FPProcessNaN on page J1-11462.
• shared/functions/float/fpprocessnans/FPProcessNaNs on page J1-11463.
• shared/functions/float/fpprocessnans3/FPProcessNaNs3 on page J1-11463.
• shared/functions/float/fpprocessnans4/FPProcessNaNs4 on page J1-11464.
• shared/functions/float/fprecipestimate/FRRecipEstimate on page J1-11465.
• shared/functions/float/fprecipestimate/RecipEstimate on page J1-11467.
• shared/functions/float/fprecpx/FPRecpX on page J1-11467.
• shared/functions/float/fpround/FPRound on page J1-11468.
• shared/functions/float/fpround/FPRoundBase on page J1-11468.
• shared/functions/float/fpround/FPRoundCV on page J1-11471.
• shared/functions/float/fproundingmode/FPRoundMode on page J1-11471.
• shared/functions/float/fproundint/FPRoundInt on page J1-11472.
• shared/functions/float/fproundintn/FPRoundIntN on page J1-11473.
• shared/functions/float/fprsqrtestimate/FPRSqrtEstimate on page J1-11474.
• shared/functions/float/fprsqrtestimate/RecipSqrtEstimate on page J1-11475.
• shared/functions/float/fpsqrt/FP.Sqrt on page J1-11476.
• shared/functions/float/fpsub/FPSub on page J1-11477.
• shared/functions/float/fpthree/FPThree on page J1-11477.
• shared/functions/float/fptofixed/FPToFixed on page J1-11477.
• shared/functions/float/fptofixedjs/FPToFixedJS on page J1-11478.
• shared/functions/float/fptwo/FPTwo on page J1-11479.
• shared/functions/float/fptype/FPType on page J1-11479.
• shared/functions/float/fpunpack/FPUnpack on page J1-11480.
• shared/functions/float/fpunpack/FPUnpackBase on page J1-11480.
• shared/functions/float/fpunpack/FPUnpackCV on page J1-11482.
• shared/functions/float/fpzeros/FPSqrt on page J1-11482.
• shared/functions/float/vfpeX86/VPFEExpandImm on page J1-11482.
• shared/functions/integer/AddCarry/ADD with Carry on page J1-11483.
• shared/functions/integer/InterruptID on page J1-11483.
• shared/functions/integer/SetInterruptRequestLevel on page J1-11483.
• shared/functions/memory/AArch64.BranchAddr on page J1-11483.
• shared/functions/memory/AccessDescriptor on page J1-11483.
• shared/functions/memory/AddrTop on page J1-11484.
• shared/functions/memory/Allocation on page J1-11484.
• shared/functions/memory/BigEndian on page J1-11485.
• shared/functions/memory/BigEndianReverse on page J1-11485.
• shared/functions/memory/Cacheability on page J1-11485.
• shared/functions/memory/CreateAccessDescriptor on page J1-11485.
• shared/functions/memory/DataMemoryBarrier on page J1-11485.
• shared/functions/memory/DataSynchronizationBarrier on page J1-11485.
• shared/functions/memory/DeviceType on page J1-11486.
• shared/functions/memory/EffectiveTBI on page J1-11486.
• shared/functions/memory/EffectiveTCMA on page J1-11486.
• shared/functions/memory/Fault on page J1-11487.
• shared/functions/memory/FaultRecord on page J1-11487.
• shared/functions/memory/FullAddress on page J1-11487.
• shared/functions/memory/GPCF on page J1-11487.
• shared/functions/memory/GPCFRecord on page J1-11488.
• shared/functions/memory/Hint_Prefetch on page J1-11488.
• shared/functions/memory/IsDataAccess on page J1-11488.
• shared/functions/memory/MBReqDomain on page J1-11488.
• shared/functions/memory/MBReqTypes on page J1-11488.
• shared/functions/memory/MPAM on page J1-11488.
• shared/functions/memory/MemAttrHints on page J1-11489.
• shared/functions/memory/MemType on page J1-11489.
• shared/functions/memory/MemoryAttributes on page J1-11489.
• shared/functions/memory/P4Space on page J1-11489.
• shared/functions/memory/Permissions on page J1-11489.
• shared/functions/memory/PhysMemRead on page J1-11489.
• shared/functions/memory/PhysMemRetStatus on page J1-11490.
• shared/functions/memory/PhysMemWrite on page J1-11490.
• shared/functions/memory/PrefetchHint on page J1-11490.
• shared/functions/memory/Shareability on page J1-11490.
• shared/functions/memory/SpeculativeStoreBypassBarrierToPA on page J1-11490.
• shared/functions/memory/SpeculativeStoreBypassBarrierToVA on page J1-11490.
• shared/functions/memory/Tag on page J1-11490.
• shared/functions/mpam/AltPARTIDspace on page J1-11490.
• shared/functions/mpam/AltPIdRealm on page J1-11491.
• shared/functions/mpam/AltPIdSecure on page J1-11491.
• shared/functions/mpam/DefaultMPAMinfo on page J1-11492.
• shared/functions/mpam/DefaultPARTID on page J1-11492.
• shared/functions/mpam/DefaultPMG on page J1-11492.
• shared/functions/mpam/GenMPAMcurEL on page J1-11492.
• shared/functions/mpam/MAP_vPARTID on page J1-11493.
• shared/functions/mpam/MPAMisMisEnabled on page J1-11494.
• shared/functions/mpam/MPAMisVirtual on page J1-11494.
• shared/functions/mpam/PARTIDspaceFromSS on page J1-11494.
• shared/functions/mpam/UsePrimarySpaceEL10 on page J1-11495.
• shared/functions/mpam/UsePrimarySpaceEL2 on page J1-11495.
• shared/functions/mpam/genMPAM on page J1-11495.
• shared/functions/mpam/genPARTID on page J1-11496.
• shared/functions/mpam/genPMG on page J1-11496.
• shared/functions/mpam/getMPAM_PARTID on page J1-11496.
• shared/functions/mpam/getMPAM_PMG on page J1-11497.
• shared/functions/mpam/mapvpmw on page J1-11497.
• shared/functions/predictionrestrict/ASID on page J1-11498.
• shared/functions/predictionrestrict/ExecutionCntxt on page J1-11498.
• shared/functions/predictionrestrict/RESTRICT_PREDICTIONS on page J1-11498.
• shared/functions/predictionrestrict/RestrictType on page J1-11499.
• shared/functions/predictionrestrict/TargetSecurityState on page J1-11499.
• shared/functions/registers/BranchTo on page J1-11499.
• shared/functions/registers/BranchToAddr on page J1-11499.
• shared/functions/registers/BranchType on page J1-11500.
• shared/functions/registers/Hint.Addr on page J1-11500.
• shared/functions/registers/NextInstrAddr on page J1-11500.
• shared/functions/registers/ThisInstrAddr on page J1-11500.
• shared/functions/registers/_PC on page J1-11500.
• shared/functions/registers/_R on page J1-11501.
• shared/functions/sysregisters/SPSR on page J1-11501.
• shared/functions/system/ArchVersion on page J1-11501.
• shared/functions/system/BranchTargetCheck on page J1-11502.
• shared/functions/system/CloseEventRegister on page J1-11502.
• shared/functions/system/ClosePendingPhysicalSError on page J1-11502.
• shared/functions/system/ClosePendingVirtualSError on page J1-11502.
• shared/functions/system/ConditionHolds on page J1-11502.
• shared/functions/system/ConsumptionOfSpeculativeDataBarrier on page J1-11503.
• shared/functions/system/CurrentInstrSet on page J1-11503.
• shared/functions/system/CurrentPL on page J1-11503.
• shared/functions/system/CurrentSecurityState on page J1-11503.
• shared/functions/system/DSBAlias on page J1-11503.
• shared/functions/system/EL0 on page J1-11504.
• shared/functions/system/EL2Enabled on page J1-11504.
• shared/functions/system/EL3SDDTrapPriority on page J1-11504.
• shared/functions/system/ELFromM32 on page J1-11504.
• shared/functions/system/ELFromSPSR on page J1-11505.
• shared/functions/system/ELIsInHost on page J1-11505.
• shared/functions/system/ELStateUsingAArch32 on page J1-11506.
• shared/functions/system/ELStateUsingAArch32K on page J1-11506.
• shared/functions/system/ELUsingAArch32 on page J1-11506.
• shared/functions/system/ELUsingAArch32K on page J1-11507.
• shared/functions/system/EffectiveSCR_EL3_RW on page J1-11507.
• shared/functions/system/EffectiveTGE on page J1-11507.
• shared/functions/system/EndOfInstruction on page J1-11507.
• shared/functions/system/EnterLowPowerState on page J1-11507.
• shared/functions/system/EventRegister on page J1-11507.
• shared/functions/system/ExceptionalOccurrenceTargetState on page J1-11507.
• shared/functions/system/FIQPending on page J1-11508.
• shared/functions/system/GetAccumulatedFPEXceptions on page J1-11508.
• shared/functions/system/GetLoadStoreType on page J1-11508.
• shared/functions/system/GetPSRFromPSTATE on page J1-11508.
• shared/functions/system/HasArchVersion on page J1-11509.
• shared/functions/system/HaveAArch32 on page J1-11509.
• shared/functions/system/HaveAArch32EL on page J1-11509.
• shared/functions/system/HaveAArch64 on page J1-11509.
• shared/functions/system/HaveEL on page J1-11509.
• shared/functions/system/HaveELUsingSecurityState on page J1-11510.
• shared/functions/system/HaveFP16Ext on page J1-11510.
• shared/functions/system/HighestEL on page J1-11510.
• shared/functions/system/Hint_DGH on page J1-11510.
• shared/functions/system/Hint_WFE on page J1-11511.
• shared/functions/system/Hint_WFI on page J1-11511.
• shared/functions/system/Hint_Yield on page J1-11512.
• shared/functions/system/IRQPending on page J1-11512.
• shared/functions/system/IllegalExceptionReturn on page J1-11512.
• shared/functions/system/InstrSet on page J1-11513.
• shared/functions/system/InstructionSynchronizationBarrier on page J1-11513.
• shared/functions/system/InterruptPending on page J1-11513.
• shared/functions/system/IsASEInstruction on page J1-11513.
• shared/functions/system/IsCMOWControlledInstruction on page J1-11513.
• shared/functions/system/IsCurrentSecurityState on page J1-11513.
• shared/functions/system/IsEventRegisterSet on page J1-11514.
• shared/functions/system/IsHighestEL on page J1-11514.
• shared/functions/system/IsInHost on page J1-11514.
• shared/functions/system/IsPhysicalSErrorPending on page J1-11514.
• shared/functions/system/IsSErrorEdgeTriggered on page J1-11514.
• shared/functions/system/IsSecure on page J1-11514.
• shared/functions/system/IsSecureBelowEL3 on page J1-11515.
• shared/functions/system/IsSecureEL2Enabled on page J1-11515.
• shared/functions/system/IsSynchronizablePhysicalSErrorPending on page J1-11515.
• shared/functions/system/IsVirtualSErrorPending on page J1-11515.
• shared/functions/system/LocalTimeoutEvent on page J1-11516.
• shared/functions/system/Mode_Bits on page J1-11516.
• shared/functions/system/NonSecureOnlyImplementation on page J1-11516.
• shared/functions/system/PLOfEL on page J1-11516.
• shared/functions/system/PSTATE on page J1-11516.
• shared/functions/system/PhysicalCountInt on page J1-11516.
• shared/functions/system/PrivilegeLevel on page J1-11516.
• shared/functions/system/ProcState on page J1-11517.
• shared/functions/system/RestoredITBits on page J1-11517.
• shared/functions/system/SCRType on page J1-11518.
• shared/functions/system/SCR_GEN on page J1-11518.
• shared/functions/system/SecurityState on page J1-11518.
• shared/functions/system/SecurityStateAtEL on page J1-11518.
• shared/functions/system/SendEvent on page J1-11519.
• shared/functions/system/SendEventLocal on page J1-11519.
• shared/functions/system/SetAccumulatedFPExceptions on page J1-11519.
• shared/functions/system/SetPSTATEFromPSR on page J1-11519.
• shared/functions/system/ShouldAdvanceIT on page J1-11520.
• shared/functions/system/ShouldAdvanceSS on page J1-11520.
• shared/functions/system/SpeculationBarrier on page J1-11520.
• shared/functions/system/SynchronizeContext on page J1-11520.
• shared/functions/system/SynchronizeErrors on page J1-11520.
• shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts on page J1-11520.
• shared/functions/system/TakeUnmaskedSErrorInterrupts on page J1-11521.
• shared/functions/system/ThisInstr on page J1-11521.
• shared/functions/system/ThisInstrLength on page J1-11521.
• shared/functions/system/Unreachable on page J1-11521.
• shared/functions/system/UsingAArch32 on page J1-11521.
• shared/functions/system/VirtualFIQPending on page J1-11521.
• shared/functions/system/VirtualIRQPending on page J1-11521.
• shared/functions/system/WFxType on page J1-11521.
• shared/functions/system/WaitForEvent on page J1-11521.
• shared/functions/system/WaitForInterrupt on page J1-11522.
• shared/functions/unpredictable/ConstrainUnpredictable on page J1-11522.
• shared/functions/unpredictable/ConstrainUnpredictableBits on page J1-11522.
• shared/functions/unpredictable/ConstrainUnpredictableBool on page J1-11522.
• shared/functions/unpredictable/ConstrainUnpredictableInteger on page J1-11522.
• shared/functions/unpredictable/ConstrainUnpredictableProcedure on page J1-11523.
• shared/functions/unpredictable/Constrain on page J1-11523.
• shared/functions/unpredictable/Unpredictable on page J1-11523.
• shared/functions/vector/AdvSIMDExpandImm on page J1-11526.
• shared/functions/vector/MatMulAdd on page J1-11526.
• shared/functions/vector/PolynomialMult on page J1-11527.
• shared/functions/vector/SatQ on page J1-11527.
• shared/functions/vector/UnsignedRSqrtEstimate on page J1-11527.
• shared/functions/vector/UnsignedRecipEstimate on page J1-11528.
• shared/functions/vector/UnsignedSatQ on page J1-11528.
shared/functions/aborts/EncodeLDFSC

// EncodeLDFSC()
// =============
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault statuscode, integer level)
bits(6) result;
if level == -1 then
  assert Have52BitIPAAndPASpaceExt();
  case statuscode of
    when Fault_AddressSize          result = '101001';
    when Fault_Translation          result = '101011';
    when Fault_SyncExternalOnWalk   result = '010011';
    when Fault_SyncParityOnWalk     result = '011011'; assert !HaveRASExt();
    when Fault_GPCFOnWalk           result = '100011';
    otherwise                       Unreachable();
  return result;
end case statuscode of

case statuscode of
  when Fault_AddressSize          result = '0000':level<1:0>; assert level IN {0,1,2,3};
  when Fault_AccessFlag           result = '0010':level<1:0>; assert level IN {0,1,2,3};
  when Fault_Permission           result = '0011':level<1:0>; assert level IN {0,1,2,3};
  when Fault_Translation          result = '0001':level<1:0>; assert level IN {0,1,2,3};
  when Fault_SyncExternal         result = '010000';
  when Fault_SyncExternalOnWalk   result = '0101':level<1:0>; assert level IN {0,1,2,3};
  when Fault_SyncParity           result = '011000';
  when Fault_SyncParityOnWalk     result = '0111':level<1:0>; assert level IN {0,1,2,3};
  when Fault_AsyncParity          result = '011001';
  when Fault_AsyncExternal        result = '010001';
  when Fault_Alignment            result = '100001';
  when Fault_Debug                result = '100010';
  when Fault_GPCFOnWalk           result = '1001':level<1:0>; assert level IN {0,1,2,3};
  when Fault_GPCFOnOutput         result = '101000';
  when Fault_TLBConflict          result = '110000';
  when Fault_HWUpdateAccessFlag   result = '110001'; // IMPLEMENTATION DEFINED
  when Fault_Lockdown             result = '110100'; // IMPLEMENTATION DEFINED
  when Fault_Exclusive            result = '110101'; // IMPLEMENTATION DEFINED
  otherwise                       Unreachable();
return result;

shared/functions/aborts/IPAValid

// IPAValid()
// =========
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
assert fault.statuscode != Fault_None;
if fault.gpcf.gpf != GPCF_None then
  return fault.secondstage;
elsif fault.s2fs1walk then
  return fault.statuscode IN {
    Fault_AccessFlag,
    Fault_Permission,
    Fault_Translation,
    Fault_AddressSize
  }; 
elsif fault.secondstage then
  return fault.statuscode IN {
    Fault_AccessFlag,
    Fault_Translation,
    Fault_AddressSize
  }
};
else
    return FALSE;

shared/functions/aborts/IsAsyncAbort
// IsAsyncAbort()
// ==============
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

boolean IsAsyncAbort(Fault statuscode)
    assert statuscode != Fault_None;
    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// ==============

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.statuscode);

shared/functions/aborts/IsDebugException
// IsDebugException()
// ==================

boolean IsDebugException(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.statuscode == Fault_Debug;

shared/functions/aborts/IsExternalAbort
// IsExternalAbort()
// =================
// Returns TRUE if the abort currently being processed is an External abort and FALSE otherwise.

boolean IsExternalAbort(Fault statuscode)
    assert statuscode != Fault_None;
    return (statuscode IN {
        Fault_SyncExternal,
        Fault_SyncParity,
        Fault_SyncExternalOnWalk,
        Fault_SyncParityOnWalk,
        Fault_AsyncExternal,
        Fault_AsyncParity
    });

// IsExternalAbort()
// =================

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.statuscode) || fault.gpcf.gpf == GPCF_EABT;

shared/functions/aborts/IsExternalSyncAbort
// IsExternalSyncAbort()
// =====================
// Returns TRUE if the abort currently being processed is an external
// synchronous abort and FALSE otherwise.
boolean IsExternalSyncAbort(Fault statuscode)
assert statuscode != Fault_None;
return (statuscode IN {
    Fault_SyncExternal,
    Fault_SyncParity,
    Fault_SyncExternalOnWalk,
    Fault_SyncParityOnWalk
});

// IsExternalSyncAbort()
// ================

boolean IsExternalSyncAbort(FaultRecord fault)
return IsExternalSyncAbort(fault.statuscode) || fault.gpcf.gpf == GPCF_EABT;

shared/functions/aborts/IsFault

// IsFault()
// =========
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
return addrdesc.fault.statuscode != Fault_None;

// IsFault()
// =========
// Return TRUE if a fault is associated with a memory access.

boolean IsFault(Fault fault)
return fault != Fault_None;

// IsFault()
// =========
// Return TRUE if a fault is associated with status returned by memory.

boolean IsFault(PhysMemRetStatus retstatus)
return retstatus.statuscode != Fault_None;

shared/functions/aborts/IsSErrorInterrupt

// IsSErrorInterrupt()
// ===================
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault statuscode)
assert statuscode != Fault_None;
return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// ===================

boolean IsSErrorInterrupt(FaultRecord fault)
return IsSErrorInterrupt(fault.statuscode);

shared/functions/aborts/IsSecondStage

// IsSecondStage()
// ===============
**shared/functions/aborts/LSInstruction Syndrome**

// Returns the extended syndrome information for a second stage fault.
// <10> - Syndrome valid bit. The syndrome is valid only for certain types of access instruction.
// <9:8> - Access size.
// <7> - Sign extended (for loads).
// <6:2> - Transfer register.
// <1> - Transfer register is 64-bit.
// <0> - Instruction has acquire/release semantics.

bits(11) LSInstruction Syndrome();

**shared/functions/aborts/ReportAsGPCException**

// Determine whether the given GPCF is reported as a Granule Protection Check Exception
// rather than a Data or Instruction Abort

boolean ReportAsGPCException(FaultRecord fault)
assert HaveRME();
assert fault.statuscode IN {Fault_GPCFOnWalk, Fault_GPCFOnOutput};
assert fault.gpcf.gpf != GPCF_None;

case fault.gpcf.gpf of
  when GPCF_Walk return TRUE;
  when GPCF_AddressSize return TRUE;
  when GPCF_EABT return TRUE;
  when GPCF_Fail return SCR_EL3.GPF == '1' && PSTATE.EL != EL3;

**shared/functions/cache/CACHE_OP**

// Performs Cache maintenance operations as per CacheRecord.

CACHE_OP(CacheRecord cache)
  IMPLEMENTATION_DEFINED;

**shared/functions/cache/CPASAtPAS**

// Get cache PA space for given PA space.

CachePASpace CPASAtPAS(PASpace pas)
  case pas of
    when PAS_NonSecure return CPAS_NonSecure;
    when PAS_Secure return CPAS_Secure;
    when PAS_Root return CPAS_Root;
    when PAS_Real  return CPAS_Real;
shared/functions/cache/CPASAtSecurityState

// CPASAtSecurityState()
// =====================
// Get cache PA space for given security state.

CachePASpace CPASAtSecurityState(SecurityState ss) {
  case ss of
    when SS_NonSecure
      return CPAS_NonSecure;
    when SS_Secure
      return CPAS_SecureNonSecure;
    when SS_Root
      return CPAS_Any;
    when SS_Realm
      return CPAS_RealmNonSecure;
  
  return CPAS_RealmNonSecure;
}

shared/functions/cache/CacheOp
class enumeration CacheOp {
  CacheOp_Clean,
  CacheOp_Invalidate,
  CacheOp_CleanInvalidate
}

shared/functions/cache/CacheOpScope
class enumeration CacheOpScope {
  CacheOpScope_SetWay,
  CacheOpScope_PoU,
  CacheOpScope_PoC,
  CacheOpScope_PoP,
  CacheOpScope_PoDP,
  CacheOpScope_ALLU,
  CacheOpScope_ALLUIS
}

shared/functions/cache/CachePASpace
class enumeration CachePASpace {
  CPAS_NonSecure,    // match entries from any PA Space - possible from Root only
  CPAS_Any,          // match entries from any PA Space - possible from Root only
  CPAS_RealmNonSecure,  // match entries from Realm or Non-Secure PAS
  CPAS_Realm,
  CPAS_Root,
  CPAS_SecureNonSecure,  // match entries from Secure or Non-Secure PAS
  CPAS_Secure
}

shared/functions/cache/CacheRecord
type CacheRecord is {
  AccType          acctype,           // Access type
  CacheOp          cacheop,           // Cache operation
  CacheOpScope     opscope,           // Cache operation type
  CacheType        cachetype,         // Cache type
  bits(64)         regval,            // For VA operations
  FullAddress      paddress,          // For VA operations
  bits(64)         vaddress,          // For VA operations
  integer          set,               // For SW operations
  integer          way,                // For SW operations
}

J1-11398 Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. Non-Confidential
integer         level;          // For SW operations
Shareability    shareability,
boolean         translated,
boolean         is_vmid_valid, // is vmid valid for current context
bits(16)        vmid,
boolean         is_asid_valid, // is asid valid for current context
bits(16)        asid,
SecurityState   security,
// For cache operations to full cache or by set/way
// For operations by address, PA space in paddress
CachePASpace    cpas
)

shared/functions/cache/CacheType

enumeration CacheType {
    CacheType_Data,
    CacheType_Tag,
    CacheType_Data_Tag,
    CacheType_Instruction
};

shared/functions/cache/DCInstNeedsTranslation

// DCInstNeedsTranslation()
// ===============
// Check whether Data Cache operation needs translation.
boolean DCInstNeedsTranslation(CacheOpScope opscope)
    if CLIDR_EL1.LoC == '000' then
        return !boolean IMPLEMENTATION_DEFINED "No fault generated for DC operations if PoC is before
        any level of cache";
    if CLIDR_EL1.LoUU == '000' && opscope == CacheOpScope_PoU then
        return !boolean IMPLEMENTATION_DEFINED "No fault generated for DC operations if PoU is before
        any level of cache";
    return TRUE;

shared/functions/cache/DecodeSW

// DecodeSW()
// =========
// Decode input value into set, way and level for SW instructions.
(integer, integer, integer) DecodeSW(bits(64) regval, CacheType cachetype)
    level = UInt(regval[3:1]);
    (set, way, linesize) = GetCacheInfo(level, cachetype);
    return (set, way, level);

shared/functions/cache/GetCacheInfo

// Returns numsets, associativity & linesize.
(integer, integer, integer) GetCacheInfo(integer level, CacheType cachetype);
shared/functions/cache/ICInstNeedsTranslation

// ICInstNeedsTranslation()
// -----------------------
// Check whether Instruction Cache operation needs translation.

boolean ICInstNeedsTranslation(CacheOpScope opscope)
    return boolean IMPLEMENTATION_DEFINED "Instruction Cache needs translation";

shared/functions/common/ASR

// ASR()
// ======

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;

shared/functions/common/ASR_C

// ASR_C()
// ========

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

shared/functions/common/Abs

// Abs()
// ======

integer Abs(integer x)
    return if x >= 0 then x else -x;

// Abs()
// ======

real Abs(real x)
    return if x >= 0.0 then x else -x;

shared/functions/common/Align

// Align()
// =======

integer Align(integer x, integer y)
    return y * (x DIV y);

// Align()
// =======
bits(N) Align(bits(N) x, integer y)
  return Align(UInt(x), y)<N-1:0>;

shared/functions/common/BitCount

// BitCount()
// =========

integer BitCount(bits(N) x)
  integer result = 0;
  for i = 0 to N-1
    if x<i> == '1' then
      result = result + 1;
  return result;

shared/functions/common/CountLeadingSignBits

// CountLeadingSignBits()
// ======================

integer CountLeadingSignBits(bits(N) x)
  return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);

shared/functions/common/CountLeadingZeroBits

// CountLeadingZeroBits()
// ======================

integer CountLeadingZeroBits(bits(N) x)
  return N - (HighestSetBit(x) + 1);

shared/functions/common/Elem

// Elem[] - non-assignment form
// =============================

bits(size) Elem<bits(N) vector, integer e, integer size]
  assert e >= 0 && (e+1)*size <= N;
  return vector<e*size+size-1 : e*size>;

// Elem[] - assignment form
// ========================

Elem<bits(N) &vector, integer e, integer size] = bits(size) value
  assert e >= 0 && (e+1)*size <= N;
  vector<(e+1)*size-1:e*size> = value;
  return;

shared/functions/common/Extend

// Extend()
// ========

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
  return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);
shared/functions/common/HighestSetBit
// HighestSetBit()
// ===============
integer HighestSetBit(bits(N) x)
for i = N-1 downto 0
  if x＜i＞ == '1' then return i;
return -1;

shared/functions/common/Int
// Int()
// =====
integer Int(bits(N) x, boolean unsigned)
  result = if unsigned then UInt(x) else SInt(x);
  return result;

shared/functions/common/IsOnes
// IsOnes()
// ========
boolean IsOnes(bits(N) x)
  return x == Ones(N);

shared/functions/common/IsZero
// IsZero()
// ========
boolean IsZero(bits(N) x)
  return x == Zeros(N);

shared/functions/common/IsZeroBit
// IsZeroBit()
// ===========
bit IsZeroBit(bits(N) x)
  return if IsZero(x) then '1' else '0';

shared/functions/common/LSL
// LSL()
// =====
bits(N) LSL(bits(N) x, integer shift)
  assert shift >= 0;
  bits(N) result;
  if shift == 0 then
    result = x;
  else
    (result, -) = LSL_C(x, shift);
  return result;
shared/functions/common/LSL_C

// LSL_C()
// =====

(bits(N), bit) LSL_C(bits(N) x, integer shift)
assert shift > 0 & shift < 256;
extended_x = x : Zeros(shift);
result = extended_x<N-1:0>;
carry_out = extended_x<N>;
return (result, carry_out);

shared/functions/common/LSR

// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
assert shift >= 0;
bits(N) result;
if shift == 0 then
  result = x;
else
  (result, -) = LSR_C(x, shift);
return result;

shared/functions/common/LSR_C

// LSR_C()
// =====

(bits(N), bit) LSR_C(bits(N) x, integer shift)
assert shift > 0 & shift < 256;
extended_x = ZeroExtend(x, shift+N);
result = extended_x<shift+N-1:shift>;
carry_out = extended_x<shift-1>;
return (result, carry_out);

shared/functions/common/LowestSetBit

// LowestSetBit()
// =============

integer LowestSetBit(bits(N) x)
for i = 0 to N-1
  if x<i> == '1' then return i;
return N;

shared/functions/common/Max

// Max()
// =====

integer Max(integer a, integer b)
return if a >= b then a else b;

// Max()
// =====
real Max(real a, real b)
    return if a >= b then a else b;

shared/functions/common/Min

    // Min()
    // =====
    integer Min(integer a, integer b)
        return if a <= b then a else b;
    // Min()
    // =====
    real Min(real a, real b)
        return if a <= b then a else b;

shared/functions/common/Ones

    // Ones()
    // ======
    bits(N) Ones(integer N)
        return Replicate('1',N);
    // Ones()
    // ======
    bits(N) Ones()
        return Ones(N);

shared/functions/common/ROR

    // ROR()
    // ======
    bits(N) ROR(bits(N) x, integer shift)
        assert shift >= 0;
        bits(N) result;
        if shift == 0 then
            result = x;
        else
            (result, -) = ROR_C(x, shift);
        return result;

shared/functions/common/ROR_C

    // ROR_C()
    // =====
    (bits(N), bit) ROR_C(bits(N) x, integer shift)
        assert shift != 0 && shift < 256;
        m = shift MOD N;
        result = LSR(x,m) OR LSL(x,N-m);
        carry_out = result<N-1>;
        return (result, carry_out);
shared/functions/common/Replicate

bits(M*N) Replicate(bits(M) x, integer N);

shared/functions/common/RoundDown

integer RoundDown(real x);

shared/functions/common/RoundTowardsZero

// RoundTowardsZero()
// ===============
integer RoundTowardsZero(real x)
return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);

shared/functions/common/RoundUp

integer RoundUp(real x);

shared/functions/common/SInt

// SInt()
// =====
integer SInt(bits(N) x)
result = 0;
for i = 0 to N-1
  if x<i> == '1' then result = result + 2^i;
  if x<N-1> == '1' then result = result - 2^N;
return result;

shared/functions/common/SignExtend

// SignExtend()
// ===========
bits(N) SignExtend(bits(M) x, integer N)
assert N >= M;
return Replicate(x<8:M-1>, N-M) : x;

shared/functions/common/Split64to32

// Split64to32()
// ===========
(bits(32), bits(32)) Split64to32(bits(64) value)
return (value<63:32>, value<31:0>);

shared/functions/common(UInt)

// UInt()
// ======
integer UInt(bits(N) x)
result = 0;
for i = 0 to N-1
  if x<i> == '1' then result = result + 2^i;
return result;

shared/functions/common/ZeroExtend

// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x, integer N)
  assert N >= M;
  return Zeros(N-M) : x;

shared/functions/common/Zeros

// Zeros()
// ========

bits(N) Zeros(integer N)
  return Replicate('0',N);

// Zeros()
// ========

bits(N) Zeros()
  return Zeros(N);

shared/functions/counters/AArch32.CheckTimerConditions

// AArch32.CheckTimerConditions()
// =============================
// Checking timer conditions for all A32 timer registers

AArch32.CheckTimerConditions()
  boolean status;
  bits(64) offset;
  offset = Zeros(64);
  assert !HaveAArch64();
  if HaveEL(EL3) then
    if CNTP_CTL.S.ENABLE == '1' then
      status = IsTimerConditionMet(offset, CNTP_CVAL_S, CNTP_CTL.S.IMASK, InterruptID_CNTPS);
      CNTP_CTL.S.ISTATUS = if status then '1' else '0';
    else
      if CNTP_CTL_NS.ENABLE == '1' then
        status = IsTimerConditionMet(offset, CNTP_CVAL_NS, CNTP_CTL_NS.IMASK, InterruptID_CNTP);
        CNTP_CTL_NS.ISTATUS = if status then '1' else '0';
      else
        if CNTP_CTL.ENABLE == '1' then
          status = IsTimerConditionMet(offset, CNTP_CVAL, CNTP_CTL.IMASK, InterruptID_CNTP);
          CNTP_CTL.ISTATUS = if status then '1' else '0';
        else
          if CNTHP_CTL.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTHP_CVAL, CNTHP_CTL.IMASK, InterruptID_CNTHP);
            CNTHP_CTL.ISTATUS = if status then '1' else '0';
          else
            if CNTV_CTL_EL0.ENABLE == '1' then
              status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0, CNTV_CTL_EL0.IMASK, InterruptID_CNTV);
CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';

return;

shared/functions/counters/AArch64.CheckTimerConditions

// AArch64.CheckTimerConditions()
// =====================================================================
// Checking timer conditions for all A64 timer registers

AArch64.CheckTimerConditions()

boolean status;
bits(64) offset;
bit imask;

SecurityState ss = CurrentSecurityState();

boolean ecv = FALSE;
if HaveECVExt() then
  ecv = CNTHCTL_EL2.ECV == '1' && SCR_EL3.ECVEn == '1' && EL2Enabled();
if ecv then
  offset = CNTPOFF_EL2;
else
  offset = Zeros(64);
if CNTP_CTL_EL0.ENABLE == '1' then
  imask = CNTP_CTL_EL0.IMASK;
  if HaveRME() && ss IN {SS_Root, SS_Realm} && CNTHCTL_EL2.CNTPMASK == '1' then
    imask = '1';
  status = IsTimerConditionMet(offset, CNTP_CVAL_EL0, imask, InterruptID_CNTP);
  CNTP_CTL_EL0.ISTATUS = if status then '1' else '0';
if ((HaveEL(EL3) || (HaveEL(EL2) && !HaveSecureEL2Ext())) &&
    CNTHP_CTL_EL2.ENABLE == '1') then
  status = IsTimerConditionMet(Zeros(64), CNTHP_CVAL_EL2, CNTHP_CTL_EL2.IMASK, InterruptID_CNTHP);
  CNTHP_CTL_EL2.ISTATUS = if status then '1' else '0';
if HaveEL(EL2) && HaveSecureEL2Ext() && CNTHPS_CTL_EL2.ENABLE == '1' then
  status = IsTimerConditionMet(Zeros(64), CNTHPS_CVAL_EL2, CNTHPS_CTL_EL2.IMASK, InterruptID_CNTHPS);
  CNTHPS_CTL_EL2.ISTATUS = if status then '1' else '0';
if CNTPS_CTL_EL1.ENABLE == '1' then
  status = IsTimerConditionMet(offset, CNTPS_CVAL_EL1, CNTPS_CTL_EL1.IMASK, InterruptID_CNTPS);
  CNTPS_CTL_EL1.ISTATUS = if status then '1' else '0';
if CNTPV_CTL_EL0.ENABLE == '1' then
  imask = CNTPV_CTL_EL0.IMASK;
  if HaveRME() && ss IN {SS_Root, SS_Realm} && CNTHCTL_EL2.CNTPMASK == '1' then
    imask = '1';
  status = IsTimerConditionMet(CNTVOFF_EL2, CNTPV_CVAL_EL0, imask, InterruptID_CNTPV);
  CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';
if ((HaveVirtHostExt() && (HaveEL(EL3) || !HaveSecureEL2Ext())) &&
    CNTHV_CTL_EL2.ENABLE == '1') then
  status = IsTimerConditionMet(Zeros(64), CNTHV_CVAL_EL2, CNTHV_CTL_EL2.IMASK, InterruptID_CNTHV);
  CNTHV_CTL_EL2.ISTATUS = if status then '1' else '0';
if (HaveSecureEL2Ext() && HaveVirtHostExt()) &&
    CNTHVS_CTL_EL2.ENABLE == '1') then
  status = IsTimerConditionMet(Zeros(64), CNTHVS_CVAL_EL2, CNTHVS_CTL_EL2.IMASK, InterruptID_CNTHVS);
  CNTHVS_CTL_EL2.ISTATUS = if status then '1' else '0';
return;
shared/functions/counters/GenericCounterTick

// GenericCounterTick()
// ====================
// Increments PhysicalCount value for every clock tick.

GenericCounterTick()
    bits(64) prev_physical_count;
    if CNTCR.EN == '0' then
        if !HaveAArch64() then
            AArch32.CheckTimerConditions();
        else
            AArch64.CheckTimerConditions();
        return;
    prev_physical_count = PhysicalCountInt();
    if HaveCNTSCExt() && CNTCR.SCEN == '1' then
        PhysicalCount = PhysicalCount + ZeroExtend(CNTSCR, 88);
    else
        PhysicalCount<87:24> = PhysicalCount<87:24> + 1;
    if !HaveAArch64() then
        AArch32.CheckTimerConditions();
    else
        AArch64.CheckTimerConditions();
    TestEventCNTP(prev_physical_count, PhysicalCountInt());
    TestEventCNTV(prev_physical_count, PhysicalCountInt());
    return;

shared/functions/counters/IsTimerConditionMet

// IsTimerConditionMet()
// =====================

boolean IsTimerConditionMet(bits(64) offset, bits(64) compare_value,
    bits(1) imask, InterruptID intid)
    boolean condition_met;
    signal level;
    condition_met = (UInt(PhysicalCountInt() - offset) -
        UInt(compare_value)) > 0;
    level = if condition_met && imask == '0' then HIGH else LOW;
    SetInterruptRequestLevel(intid, level);
    return condition_met;

shared/functions/counters/PhysicalCount

bits(88) PhysicalCount;

shared/functions/counters/SetEventRegister

// SetEventRegister()
// ===================
// Sets the Event Register of this PE

SetEventRegister()
    EventRegister = '1';
    return;

shared/functions/counters/TestEventCNTP

// TestEventCNTP()
// ===============
// Generate Event stream from the physical counter
TestEventCNTP(bits(64) prev_physical_count, bits(64) current_physical_count)
bits(64) offset;
bits(1) samplebit, previousbit;
if CNTHCTL_EL2.EVNTEN == '1' then
    n = UInt(CNTHCTL_EL2.EVNTI);
    if HaveECVExt() && CNTHCTL_EL2.EVNTIS == '1' then
        n = n + 8;
    boolean ecv = FALSE;
    if HaveECVExt() then
        ecv = (EL2Enabled() && CNTHCTL_EL2.ECV == '1' &&
        SCR_EL3.ECVEn == '1');
        offset = if ecv then CNTPOFF_EL2 else Zeros(64);
    samplebit = (current_physical_count - offset)<n>;
    previousbit = (prev_physical_count - offset)<n>;
    if CNTHCTL_EL2.EVNTDIR == '0' then
        if previousbit == '0' && samplebit == '1' then SetEventRegister();
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister();
    return;

shared/functions/counters/TestEventCNTV

// TestEventCNTV()
// ===============
// Generate Event stream from the virtual counter
TestEventCNTV(bits(64) prev_physical_count, bits(64) current_physical_count)
bits(64) offset;
bits(1) samplebit, previousbit;
if (!(HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11') &&
    CNTKCTL_EL1.EVNTEN == '1') then
    n = UInt(CNTKCTL_EL1.EVNTI);
    if HaveECVExt() && CNTKCTL_EL1.EVNTIS == '1' then
        n = n + 8;
    if HaveEL(EL2) && (!EL2Enabled() || HCR_EL2.<E2H,TGE> != '11') then
        offset = CNTPOFF_EL2;
    else
        offset = Zeros(64);
    samplebit = (current_physical_count - offset)<n>;
    previousbit = (prev_physical_count - offset)<n>;
    if CNTKCTL_EL1.EVNTDIR == '0' then
        if previousbit == '0' && samplebit == '1' then SetEventRegister();
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister();
    return;

shared/functions/crc/BitReverse

// BitReverse()
// =============
bits(N) BitReverse(bits(N) data)
bits(N) result;
for i = 0 to N-1
    result<(N-i)-1> = data<i>;
return result;

shared/functions/crc/HaveCRCExt

// HaveCRCExt()
// =============
boolean HaveCRCExt()
    return HasArchVersion(ARMv8p1) || boolean IMPLEMENTATION_DEFINED "Have CRC extension";

shared/functions/crc/Poly32Mod2

// Poly32Mod2()
// ============
// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data_in, bits(32) poly)
    assert N > 32;
    bits(N) data = data_in;
    for i = N-1 downto 32
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR (poly:Zeros(i-32));
    return data<31:0>;

shared/functions/crypto/AESInvMixColumns

// AESInvMixColumns()
// ================
// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

bits(128) AESInvMixColumns(bits (128) op)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> = FFmul0E(in0<c*8+:8>) EOR FFmul0B(in1<c*8+:8>) EOR FFmul0D(in2<c*8+:8>) EOR
                      FFmul09(in3<c*8+:8>);
        out1<c*8+:8> = FFmul09(in0<c*8+:8>) EOR FFmul0E(in1<c*8+:8>) EOR FFmul0B(in2<c*8+:8>) EOR
                      FFmul0D(in3<c*8+:8>);
        out2<c*8+:8> = FFmul0D(in0<c*8+:8>) EOR FFmul09(in1<c*8+:8>) EOR FFmul0E(in2<c*8+:8>) EOR
                      FFmul0B(in3<c*8+:8>);
        out3<c*8+:8> = FFmul0B(in0<c*8+:8>) EOR FFmul0D(in1<c*8+:8>) EOR FFmul09(in2<c*8+:8>) EOR
                      FFmul0E(in3<c*8+:8>);

    return (out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
            out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
            out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
            out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>);

shared/functions/crypto/AESInvShiftRows

// AESInvShiftRows()
// ================
// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

bits(128) AESInvShiftRows(bits(128) op)
    return (op< 31: 24> : op< 55: 48> : op< 79: 72> : op<103: 96> :
            op<127:120> : op< 23: 16> : op< 47: 40> : op< 71: 64> :
op< 63: 56> : op< 87: 80> : op<111:104> : op<  7:  0> 
);

shared/functions/crypto/AESInvSubBytes

// AESInvSubBytes()
// =================
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

bits(128) AESInvSubBytes(bits(128) op)
// Inverse S-box values
bits(16*16*8) GF2_inv = (  
  /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
  /*F*/ 0x7d0c2155631469e126d677ba7e042b17<127:0> :
  /*E*/ 0x61953383cbebc8b0f52aee4d3be0a8<127:0> :
  /*D*/ 0xeef9c99397ae52d64ab519a97f5160<127:0> :
  /*C*/ 0x5f3ec0827593102b131c707833a62df1f<127:0> :
  /*B*/ 0x8f45ac78f6c98d9a20792c643e56f<127:0> :
  /*A*/ 0x1bbe18ae0e2b76f95c529d711af47<127:0> :
  /*9*/ 0x6edf751ce37f9e28535ade72724ac96<127:0> :
  /*8*/ 0x73e6bf040cecf297aad674f4111913<127:0> :
  /*7*/ 0x68a130103dbac102f3fca8f1e2c9d<127:0> :
  /*6*/ 0x0645b3080558e4f70ad88c800ab890<127:0> :
  /*5*/ 0x489d7e48129659868864f687<127:0> :
  /*4*/ 0x3d1886642b70b224d286a12e08<127:0> :
  /*3*/ 0x4c3fa2b954cee3d2c2e32947b54<127:0> :
  /*2*/ 0x53b9280c9a344439e3a877f2f9b229e0<127:0> :
  /*1*/ 0x75550b340bf38a53630d560952<127:0>
);  
bits(128) out;
for i = 0 to 15
  out<i*8+:8> = GF2_inv<UInt(op<i*8+:8>)*8+:8>;
return out;

shared/functions/crypto/AESMixColumns

// AESMixColumns()
// =============
// Transformation in the Cipher that takes all of the columns of the
// State and mixes their data (independently of one another) to
// produce new columns.

bits(128) AESMixColumns(bits (128) op)
bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;
bits(4*8) out0;
bits(4*8) out1;
bits(4*8) out2;
bits(4*8) out3;
for c = 0 to 3
  out0<c*8+:8> = FFmul02(in0<c*8+:8>) EOR FFmul03(in1<c*8+:8>) EOR in2<c*8+:8> EOR in3<c*8+:8>;
  out1<c*8+:8> = in0<c*8+:8> EOR FFmul02(in1<c*8+:8>) EOR FFmul03(in2<c*8+:8>) EOR FFmul03(in3<c*8+:8>);
  out2<c*8+:8> = in0<c*8+:8> EOR in1<c*8+:8> EOR FFmul02(in2<c*8+:8>) EOR in3<c*8+:8>;
  out3<c*8+:8> = FFmul03(in0<c*8+:8>) EOR in1<c*8+:8> EOR in2<c*8+:8> EOR FFmul02(in3<c*8+:8>);
shared/functions/crypto/AESShiftRows

// AESShiftRows()
// ==============
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

bits(128) AESShiftRows(bits(128) op)
return {
};

shared/functions/crypto/AESSubBytes

// AESSubBytes()
// =============
// Transformation in the Cipher that processes the State using a nonlinear
// byte substitution table (S-box) that operates on each of the State bytes
// independently.

bits(128) AESSubBytes(bits(128) op)
// S-box values
bits(16*16*8) GF2 = (/* F E D C B A 9 8 7 6 5 4 3 2 1 0 */
  /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c<127:0>: /*E*/ 0xdf2855cee9871e9b48ed9691198f8e1<127:0>: 
  /*D*/ 0x9e1dc186b95735610ef6034866b53e70<127:0>: /*C*/ 0x8a8bbd41f74ddec6e46a1ce2578ba<127:0>: 
  /*B*/ 0x08ae7a65eaf4566ca94ed58d6d37c8e7<127:0>: /*A*/ 0x79e4959162ac3c25c2406490a3a32e0<127:0>: 
  /*9*/ 0xdb0b5ede14b8ee4688902a22dc4f8160<127:0>: /*8*/ 0x731956d437ea741f44975fec130cdd<127:0>: 
  /*7*/ 0x2d56fc112f1dbd6cfc539d928f40a351<127:0>: /*6*/ 0x8a89fc35076f2f94583334d3fbaafe0<127:0>: 
  /*5*/ 0xc5f84c4a39bce65bb1fc20ed0d153<127:0>: /*4*/ 0x842fe329b3d3b52a05a6e1b1a2c8309<127:0>: 
  /*3*/ 0x7b52b7ebe28012079a059618c323c704<127:0>: /*2*/ 0x1531d871fe5a534ccf73f62693f9b7<127:0>: 
  /*1*/ 0xc072a49cafa2d4ad0f4795afa79c982ca<127:0>: /*0*/ 0x76ab7fe2b676103c56f6bf27b777c63<127:0>);
bits(128) out;
for i = 0 to 15
  out<i*8+:8> = GF2<UInt(op<i*8+:8>)*8+:8>;
return out;

shared/functions/crypto/FFmul02

// FFmul02()
// =========

bits(8) FFmul02(bits(8) b)
bits(256*8) FFmul_02 = (/*  */

Armv8 Pseudocode
J1.3 Shared pseudocode

/*
/*F*/
/*E*/
/*D*/
/*C*/
/*B*/
/*A*/
/*9*/
/*8*/
/*7*/
/*6*/
/*5*/
/*4*/
/*3*/
/*2*/
/*1*/
/*0*/

F E D C B A 9 8 7 6 5 4 3 2 1 0
*/
0xE5E7E1E3EDEFE9EBF5F7F1F3FDFFF9FB<127:0> :
0xC5C7C1C3CDCFC9CBD5D7D1D3DDDFD9DB<127:0> :
0xA5A7A1A3ADAFA9ABB5B7B1B3BDBFB9BB<127:0> :
0x858781838D8F898B959791939D9F999B<127:0> :
0x656761636D6F696B757771737D7F797B<127:0> :
0x454741434D4F494B555751535D5F595B<127:0> :
0x252721232D2F292B353731333D3F393B<127:0> :
0x050701030D0F090B151711131D1F191B<127:0> :
0xFEFCFAF8F6F4F2F0EEECEAE8E6E4E2E0<127:0> :
0xDEDCDAD8D6D4D2D0CECCCAC8C6C4C2C0<127:0> :
0xBEBCBAB8B6B4B2B0AEACAAA8A6A4A2A0<127:0> :
0x9E9C9A98969492908E8C8A8886848280<127:0> :
0x7E7C7A78767472706E6C6A6866646260<127:0> :
0x5E5C5A58565452504E4C4A4846444240<127:0> :
0x3E3C3A38363432302E2C2A2826242220<127:0> :
0x1E1C1A18161412100E0C0A0806040200<127:0>

);
return FFmul_02<UInt(b)*8+:8>;

shared/functions/crypto/FFmul03
// FFmul03()
// =========
bits(8) FFmul03(bits(8) b)
bits(256*8) FFmul_03 = (
/*
F E D C B A 9 8 7 6 5 4 3 2 1 0
*/
/*F*/ 0x1A191C1F16151013020104070E0D080B<127:0> :
/*E*/ 0x2A292C2F26252023323134373E3D383B<127:0> :
/*D*/ 0x7A797C7F76757073626164676E6D686B<127:0> :
/*C*/ 0x4A494C4F46454043525154575E5D585B<127:0> :
/*B*/ 0xDAD9DCDFD6D5D0D3C2C1C4C7CECDC8CB<127:0> :
/*A*/ 0xEAE9ECEFE6E5E0E3F2F1F4F7FEFDF8FB<127:0> :
/*9*/ 0xBAB9BCBFB6B5B0B3A2A1A4A7AEADA8AB<127:0> :
/*8*/ 0x8A898C8F86858083929194979E9D989B<127:0> :
/*7*/ 0x818287848D8E8B88999A9F9C95969390<127:0> :
/*6*/ 0xB1B2B7B4BDBEBBB8A9AAAFACA5A6A3A0<127:0> :
/*5*/ 0xE1E2E7E4EDEEEBE8F9FAFFFCF5F6F3F0<127:0> :
/*4*/ 0xD1D2D7D4DDDEDBD8C9CACFCCC5C6C3C0<127:0> :
/*3*/ 0x414247444D4E4B48595A5F5C55565350<127:0> :
/*2*/ 0x717277747D7E7B78696A6F6C65666360<127:0> :
/*1*/ 0x212227242D2E2B28393A3F3C35363330<127:0> :
/*0*/ 0x111217141D1E1B18090A0F0C05060300<127:0>
);
return FFmul_03<UInt(b)*8+:8>;

shared/functions/crypto/FFmul09
// FFmul09()
// =========
bits(8) FFmul09(bits(8) b)
bits(256*8) FFmul_09 = (
/*
F E D C B A 9 8 7 6 5 4 3 2 1 0
*/
/*F*/ 0x464F545D626B70790E071C152A233831<127:0> :
/*E*/ 0xD6DFC4CDF2FBE0E99E978C85BAB3A8A1<127:0> :
/*D*/ 0x7D746F6659504B42353C272E1118030A<127:0> :
/*C*/ 0xEDE4FFF6C9C0DBD2A5ACB7BE8188939A<127:0> :
/*B*/ 0x3039222B141D060F78716A635C554E47<127:0> :
/*A*/ 0xA0A9B2BB848D969FE8E1FAF3CCC5DED7<127:0> :
/*9*/ 0x0B0219102F263D34434A5158676E757C<127:0> :
/*8*/ 0x9B928980BFB6ADA4D3DAC1C8F7FEE5EC<127:0> :
/*7*/ 0xAAA3B8B18E879C95E2EBF0F9C6CFD4DD<127:0> :

ARM DDI 0487I.a
ID081822

Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1-11413


/*6*/ 0x3A3328211E170C05727B60695F4440<127:0> :
/*5*/ 0x9198838AB5BCA7AED900CFC2DF4EEFE<127:0> :
/*4*/ 0x0108131A252C373E49405B526D647F<127:0> :
/*3*/ 0xDDC5EC7F8F11AE39490B68F0B09A2AB<127:0> :
/*2*/ 0x4C455E5768617A73040D161F2029323B<127:0> :
/*1*/ 0xE7EF5FCC3CAD08A68DB848829990<127:0> :
 /*0*/ 0x77E656C53A41483F362D241B20900<127:0> :
};
return FFmul_09<UInt(b)*8+:8>;

shared/functions/crypto/FFmul0B

// FFmul0B()
// =========

bits(8) FFmul0B(bits(8) b)
bits(256*8) FFmul_0B =(
/* F E D C B A 9 8 7 6 5 4 3 2 1 0 */
/* F*/ 0xA3A8B5BE8F849992FBF0EDED7DOC1CA<127:0> :
/* E*/ 0x1318050E3F3429224B405D5667C717A<127:0> :
/* D*/ 0x6863E75444F5259303B262DC170A01<127:0> :
/* C*/ 0x5554348726F640061B0121A373C3<127:0> :
/* B*/ 0x555E434879726F403B262D170A01<127:0> :
/* A*/ 0x2E2538330209141F767D0685A14C47<127:0> :
/* 9*/ 0x9E98883B289A4ACCGDB0DEAEC1CF7<127:0> :
/* 8*/ 0x754F2497836560C8731221A363D0<127:0> :
/* 7*/ 0x4E4F2F9C8C0D8BC7A4490B680D<127:0> :
/* 6*/ 0xF243932030815E777C646A5804D4<127:0> :
/* 5*/ 0x9F98982B388A5AECC7C01DAEBEF0DF6<127:0> :
/* 4*/ 0x2A9B8488E59893F3A1E7DEDCC0C6<127:0> :
/* 3*/ 0x9198838AB5BCA7AED900CFC2DF4EEFE<127:0> :
 /*2*/ 0x4C455E5768617A73040D161F2029323B<127:0> :
/*1*/ 0xE7EF5FCC3CAD08A68DB848829990<127:0> :
 /*0*/ 0x77E656C53A41483F362D241B20900<127:0> :
);
return FFmul_0B<UInt(b)*8+:8>;

shared/functions/crypto/FFmul0D

// FFmul0D()
// =========

bits(8) FFmul0D(bits(8) b)
bits(256*8) FFmul_0D =(
/* F E D C B A 9 8 7 6 5 4 3 2 1 0 */
/* F*/ 0x979A8D80A3AEB9B4FFF2E5EB6CD01<127:0> :
/* E*/ 0x474A5D50737E6942F2235381B16010<127:0> :
/* D*/ 0x21363B181B02044495E5370D6A67<127:0> :
/* C*/ 0xCFC16E8BC50D294998EE8A0ADAB87<127:0> :
/* B*/ 0x999A8B80859885A61B1021A373C3<127:0> :
/* A*/ 0x2A270313E3049024F8557686DCC0C6<127:0> :
/* 9*/ 0x9198838AB5BCA7AED900CFC2DF4EEFE<127:0> :
/* 8*/ 0x4C455E5768617A73040D161F2029323B<127:0> :
/* 7*/ 0x9D9B618CAFA2B58F3EE94C7AC0D0<127:0> :
/* 6*/ 0x4B46515CC7F26568232E3934171A000<127:0> :
*/0*/ 0x77E656C53A41483F362D241B20900<127:0> :
);}return FFmul_0D<UInt(b)*8+:8>;
shared/functions/crypto/FFmul0E

// FFmul0E()
// =========

bits(8) FFmul0E(bits(8) b)
bits(256x8) FFmul_0E = ( 
    /* F E D C B A 9 8 7 6 5 4 3 2 1 0 */
    /* F*/ 0x8D83919FB5BBA9A7FDF3E1EFC5CBD9D7<127:0>:
    /* E*/ 0x6D63717F555B49471D13010F252B3937<127:0>:
    /* D*/ 0x56584A46E06727C6283A431E00620C<127:0>:
    /* C*/ 0x0B9BA44A48E0929CC2CD4D4F90E2EC<127:0>:
    /* B*/ 0x202E3C321B640A0385E4C26666747A<127:0>:
    /* A*/ 0xC8C9C92D2864E4A008ECA28886949A<127:0>:
    /* 9*/ 0x0B5E5E9193C9DD1F8B59999B30AFA1<127:0>:
    /* 8*/ 0x0B150792320316657799535D4F41<127:0>:
    /* 7*/ 0x0C32000F4F4E8E8E8B2A0B84B9896<127:0>:
    /* 6*/ 0x2C22303E14A08065CS32406668C87B6<127:0>:
    /* 5*/ 0x0B190B052F12333D67697B735DF1434D<127:0>:
    /* 4*/ 0xF7F0EBE5CFC13D087899B5FB1A3AD<127:0>:
    /* 3*/ 0x16F70D75975F18B11DF8D62927353B<127:0>:
    /* 2*/ 0x818F9D9387A58ABF1FDE3C9C0D5B<127:0>:
    /* 1*/ 0x28A4646826C7E702A243638121C0E0<127:0>:
    /* 0*/ 0x5444648626C7E702A243638121C0E0<127:0>
  );
return FFmul_0E<<Int(b)*8>:8>;
shared/functions/crypto/HaveSHA256Ext

// HaveSHA256Ext()
// ===============
// TRUE if SHA256 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA256Ext()
return boolean IMPLEMENTATION_DEFINED "Has SHA256 Crypto instructions";

shared/functions/crypto/HaveSHA3Ext

// HaveSHA3Ext()
// =============
// TRUE if SHA3 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA3Ext()
if !HasArchVersion(ARMv8p2) || !HaveSHA1Ext() && HaveSHA256Ext() then
return FALSE;
return boolean IMPLEMENTATION_DEFINED "Has SHA3 Crypto instructions";

shared/functions/crypto/HaveSHA512Ext

// HaveSHA512Ext()
// ===============
// TRUE if SHA512 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA512Ext()
if !HasArchVersion(ARMv8p2) || !HaveSHA1Ext() && HaveSHA256Ext() then
return FALSE;
return boolean IMPLEMENTATION_DEFINED "Has SHA512 Crypto instructions";

shared/functions/crypto/HaveSM3Ext

// HaveSM3Ext()
// ============
// TRUE if SM3 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM3Ext()
if !HasArchVersion(ARMv8p2) then
return FALSE;
return boolean IMPLEMENTATION_DEFINED "Has SM3 Crypto instructions";

shared/functions/crypto/HaveSM4Ext

// HaveSM4Ext()
// ============
// TRUE if SM4 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM4Ext()
if !HasArchVersion(ARMv8p2) then
return FALSE;
return boolean IMPLEMENTATION_DEFINED "Has SM4 Crypto instructions";
shared/functions/crypto/ROL

// ROL()
// ======
bits(N) ROL(bits(N) x, integer shift)
assert shift >= 0 && shift <= N;
if (shift == 0) then
  return x;
else
  return ROR(x, N-shift);

shared/functions/crypto/SHA256hash

// SHA256hash()
// =============
bits(128) SHA256hash(bits(128) x_in, bits(128) y_in, bits(128) w, boolean part1)
  bits(32) chs, maj, t;
  bits(128) x = x_in;
  bits(128) y = y_in;
  for e = 0 to 3
    chs = SHAchoose(y<31:0>, y<63:32>, y<95:64>);
    maj = SHAmajority(x<31:0>, x<63:32>, x<95:64>);
    t = y<127:96> + SHAhashSIGMA1(y<31:0>) + chs + Elem[w, e, 32];
    x<127:96> = t + x<127:96>;
    y<127:96> = t + SHAhashSIGMA0(x<31:0>) + maj;
    <y, x> = ROL(y : x, 32);
  return (if part1 then x else y);

shared/functions/crypto/SHAchoose

// SHAchoose()
// ===========
bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
  return (((y EOR z) AND x) EOR z);

shared/functions/crypto/SHAhashSIGMA0

// SHAhashSIGMA0()
// ===============
bits(32) SHAhashSIGMA0(bits(32) x)
  return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);

shared/functions/crypto/SHAhashSIGMA1

// SHAhashSIGMA1()
// ===============
bits(32) SHAhashSIGMA1(bits(32) x)
  return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);

shared/functions/crypto/SHAmajority

// SHAmajority()
// =============
bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
   return ((x AND y) OR ((x OR y) AND z));

shared/functions/crypto/SHAparity

   // SHAparity()
   // ===============
   bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
   return (x EOR y EOR z);

shared/functions/crypto/Sbox

   // Sbox()
   // =====
   // Used in SM4 crypto instruction
   bits(8) Sbox(bits(8) sboxin)
   bits(8) sboxout;
   bits(2048) sboxstring =
   0xd690e9fecce13db716b614c228fb2c052b679a762abe04c3aa441326498606999c4250f491ef987a33540b43edcfa62e4b31ca
   9c908e89580df9fa7a7f7c37317ba83593c19e685f4a8686b81b27164ada88f8eb0f4b70569d351e2408e663581a2
   25227c3b01278874d0046579fd32752c3602e7a0c4c99eeabf84d240c73b5a3f7f2ce8d15a1ebe8e65e49b341a55ad9313230f
   58c1e31df0e622e266ca06c02923ab0d5346fd5db3745def8e2f03f6a72665c6b518d1ba792bbddbc7f11d95c411f105ad90a
   c131885c7b2b2d74db012b8e5b4b08069974a8c96777e65b9f109c56ec6848f07dec3ad4d2079ee5f3ed7cb3948c2047;
   sboxout = sboxstring<<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8+8;
   return sboxout;

shared/functions/exclusive/ClearExclusiveByAddress

   // Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
   // record any part of the physical address region of size bytes starting at paddress.
   // It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
   // is also cleared if it records any part of the address region.
   ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);

shared/functions/exclusive/ClearExclusiveLocal

   // Clear the local Exclusives monitor for the specified processorid.
   ClearExclusiveLocal(integer processorid);

shared/functions/exclusive/ClearExclusiveMonitors

   // ClearExclusiveMonitors()
   // ================
   // Clear the local Exclusives monitor for the executing PE.
   ClearExclusiveMonitors()
       ClearExclusiveLocal(ProcessorID());

shared/functions/exclusive/ExclusiveMonitorsStatus

   // Returns '0' to indicate success if the last memory write by this PE was to
   // the same physical address region endorsed by ExclusiveMonitorsPass().
   // Returns '1' to indicate failure if address translation resulted in a different
// physical address.
bit ExclusiveMonitorsStatus();

shared/functions/exclusive/IsExclusiveGlobal

// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);

shared/functions/exclusive/IsExclusiveLocal

// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);

shared/functions/exclusive/MarkExclusiveGlobal

// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);

shared/functions/exclusive/MarkExclusiveLocal

// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);

shared/functions/exclusive/ProcessorID

// Return the ID of the currently executing PE.
integer ProcessorID();

shared/functions/extension/AArch32.HaveHPDExt

// AArch32.HaveHPDExt()
// ===============
boolean AArch32.HaveHPDExt()
return (HasArchVersion(ARMv8p2) &&
  boolean IMPLEMENTATION_DEFINED "Has AArch32 hierarchical permission disables");

shared/functions/extension/AArch64.HaveHPDExt

// AArch64.HaveHPDExt()
// ===============
boolean AArch64.HaveHPDExt()
return HasArchVersion(ARMv8p1);

shared/functions/extension/Have16bitVMID

// Have16bitVMID()
// ===============
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.
boolean Have16bitVMID()
return (HasArchVersion(ARMv8p1) && HaveEL(EL2) &&
        boolean IMPLEMENTATION_DEFINED "Has 16-bit VMID");

shared/functions/extension/Have52BitIPAAndPASpaceExt

// Have52BitIPAAndPASpaceExt()
// ========================================
// Returns TRUE if 52-bit IPA and PA extension support
// is implemented, and FALSE otherwise.

boolean Have52BitIPAAndPASpaceExt()
return (HasArchVersion(ARMv8p7) &&
        boolean IMPLEMENTATION_DEFINED "Has 52-bit IPA and PA support" &&
        Have52BitVAExt() && Have52BitPAExt());

shared/functions/extension/Have52BitPAExt

// Have52BitPAExt()
// ================
// Returns TRUE if Large Physical Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitPAExt()
return (HasArchVersion(ARMv8p2) &&
        boolean IMPLEMENTATION_DEFINED "Has large 52-bit PA/IPA support");

shared/functions/extension/Have52BitVAExt

// Have52BitVAExt()
// ================
// Returns TRUE if Large Virtual Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitVAExt()
return (HasArchVersion(ARMv8p2) &&
        boolean IMPLEMENTATION_DEFINED "Has large 52-bit VA support");

shared/functions/extension/HaveAArch32BF16Ext

// HaveAArch32BF16Ext()
// ====================
// Returns TRUE if AArch32 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveAArch32BF16Ext()
return (HasArchVersion(ARMv8p2) &&
        boolean IMPLEMENTATION_DEFINED "Has AArch32 BFloat16 extension");

shared/functions/extension/HaveAArch32Int8MatMulExt

// HaveAArch32Int8MatMulExt()
// ============================
// Returns TRUE if AArch32 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveAArch32Int8MatMulExt()
return (HasArchVersion(ARMv8p2) &&
        boolean IMPLEMENTATION_DEFINED "Has AArch32 Int8 Mat Mul extension");

shared/functions/extension/HaveAccessFlagUpdateExt

// HaveAccessFlagUpdateExt()
// ------------------------

boolean HaveAccessFlagUpdateExt()
    return HasArchVersion(ARMv8p1);

shared/functions/extension/HaveAltFP

// HaveAltFP()
// ===========
// Returns TRUE if alternative Floating-point extension support
// is implemented, and FALSE otherwise.

boolean HaveAltFP()
    return HasArchVersion(ARMv8p7);

shared/functions/extension/HaveAtomicExt

// HaveAtomicExt()
// ===============

boolean HaveAtomicExt()
    return HasArchVersion(ARMv8p1);

shared/functions/extension/HaveBF16Ext

// HaveBF16Ext()
// =============
// Returns TRUE if AArch64 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveBF16Ext()
    return (HasArchVersion(ARMv8p6) ||
            (HasArchVersion(ARMv8p2) &&
             boolean IMPLEMENTATION_DEFINED "Has AArch64 BFloat16 extension");

shared/functions/extension/HaveBRBEv1p1

// HaveBRBEv1p1()
// ==============
// Returns TRUE if BRBEv1p1 extension is implemented, and FALSE otherwise.

boolean HaveBRBEv1p1()
    return (HasArchVersion(ARMv9p3) &&
            boolean IMPLEMENTATION_DEFINED "Has BRBEv1p1 extension");

shared/functions/extension/HaveBRBExt

// HaveBRBExt()
// =============
// Returns TRUE if Branch Record Buffer Extension is implemented, and FALSE otherwise.
boolean HaveBRBExt()
    return boolean IMPLEMENTATION_DEFINED "Has Branch Record Buffer Extension";

shared/functions/extension/HaveBTIExt

    // HaveBTIExt()
    // ============
    // Returns TRUE if support for Branch Target Indentification is implemented.
    boolean HaveBTIExt()
        return HasArchVersion(ARMv8p5);

shared/functions/extension/HaveBlockBBM

    // HaveBlockBBM()
    // ==============  
    // Returns TRUE if support for changing block size without requiring
    // break-before-make is implemented.
    boolean HaveBlockBBM()
        return HasArchVersion(ARMv8p4);

shared/functions/extension/HaveCNTSCExt

    // HaveCNTSCExt()
    // ==============  
    // Returns TRUE if the Generic Counter Scaling is implemented, and FALSE
    // otherwise.
    boolean HaveCNTSCExt()
        return (HasArchVersion(ARMv8p4) &&
            boolean IMPLEMENTATION_DEFINED "Has Generic Counter Scaling support");

shared/functions/extension/HaveCommonNotPrivateTransExt

    // HaveCommonNotPrivateTransExt()
    // ==============================  
    boolean HaveCommonNotPrivateTransExt()
        return HasArchVersion(ARMv8p2);

shared/functions/extension/HaveDGHExt

    // HaveDGHExt()
    // ============  
    // Returns TRUE if Data Gathering Hint instruction support is implemented, and
    // FALSE otherwise.
    boolean HaveDGHExt()
        return boolean IMPLEMENTATION_DEFINED "Has AArch64 DGH extension";

shared/functions/extension/HaveDITEExt

    // HaveDITEExt()
    // =============
boolean HaveDITExt()
    return HasArchVersion(ARMv8p4);

shared/functions/extension/HaveDOTPExt

    boolean HaveDOTPExt()
    // =============
    // Returns TRUE if Dot Product feature support is implemented, and FALSE otherwise.
    boolean HaveDOTPExt()
    return HasArchVersion(ARMv8p4) ||
           (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has Dot Product extension");

shared/functions/extension/HaveDirtyBitModifierExt

    boolean HaveDirtyBitModifierExt()
    // =========================
    boolean HaveDirtyBitModifierExt()
    return HasArchVersion(ARMv8p1);

shared/functions/extension/HaveDoPD

    boolean HaveDoPD()
    // ==========
    // Returns TRUE if Debug Over Power Down extension support is implemented and FALSE otherwise.
    boolean HaveDoPD()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has DoPD extension";

shared/functions/extension/HaveDoubleFaultExt

    boolean HaveDoubleFaultExt()
    // ===============
    boolean HaveDoubleFaultExt()
    return (HasArchVersion(ARMv8p4) && HaveEL(EL3) && !ELUsingAAArch32(EL3) && HaveIESB());

shared/functions/extension/HaveDoubleLock

    boolean HaveDoubleLock()
    // ===================
    // Returns TRUE if support for the OS Double Lock is implemented.
    boolean HaveDoubleLock()
    return (!HasArchVersion(ARMv8p4) || boolean IMPLEMENTATION_DEFINED "OS Double Lock is implemented");

shared/functions/extension/HaveE0PDExt

    boolean HaveE0PDExt()
    // =============
    // Returns TRUE if support for constant fault times for unprivileged accesses to the memory map is implemented.
boolean HaveE0PDExt()
    return HasArchVersion(ARMv8p5);

shared/functions/extension/HaveECVExt

    // HaveECVExt()
    // ============
    // Returns TRUE if Enhanced Counter Virtualization extension
    // support is implemented, and FALSE otherwise.
    boolean HaveECVExt()
        return HasArchVersion(ARMv8p6);

shared/functions/extension/HaveETExt

    // HaveETExt()
    // ===========
    // Returns TRUE if Embedded Trace Extension is implemented, and FALSE otherwise.
    boolean HaveETExt()
        return boolean IMPLEMENTATION_DEFINED "Has Embedded Trace Extension";

shared/functions/extension/HaveExtendedCacheSets

    // HaveExtendedCacheSets()
    // =======================
    boolean HaveExtendedCacheSets()
        return HasArchVersion(ARMv8p3);

shared/functions/extension/HaveExtendedECDebugEvents

    // HaveExtendedECDebugEvents()
    // ===========================
    boolean HaveExtendedECDebugEvents()
        return HasArchVersion(ARMv8p2);

shared/functions/extension/HaveExtendedExecuteNeverExt

    // HaveExtendedExecuteNeverExt()
    // =============================
    boolean HaveExtendedExecuteNeverExt()
        return HasArchVersion(ARMv8p2);

shared/functions/extension/HaveFCADDExt

    // HaveFCADDExt()
    // ==============
    boolean HaveFCADDExt()
        return HasArchVersion(ARMv8p3);
shared/functions/extension/HaveFGTExt

// HaveFGTExt()
// ============
// Returns TRUE if Fine Grained Trap is implemented, and FALSE otherwise.

boolean HaveFGTExt()
    return HasArchVersion(ARMv8p6);

shared/functions/extension/HaveFJCVTZSExt

// HaveFJCVTZSExt()
// ================

boolean HaveFJCVTZSExt()
    return HasArchVersion(ARMv8p3);

shared/functions/extension/HaveFP16MulNoRoundingToFP32Ext

// HaveFP16MulNoRoundingToFP32Ext()
// ================================
// Returns TRUE if has FP16 multiply with no intermediate rounding accumulate
// to FP32 instructions, and FALSE otherwise

boolean HaveFP16MulNoRoundingToFP32Ext()
    if !HaveFP16Ext() then return FALSE;
    if HasArchVersion(ARMv8p4) then return TRUE;
    return (HasArchVersion(ARMv8p2) &
        boolean IMPLEMENTATION_DEFINED "Has accumulate FP16 product into FP32 extension");

shared/functions/extension/HaveFeatCMOW

// HaveFeatCMOW()
// ==============
// Returns TRUE if the SCTLR_EL1.CMOW bit is implemented and the SCTLR_EL2.CMOW and
// HCRX_EL2.CMOW bits are implemented if EL2 is implemented.

boolean HaveFeatCMOW()
    return HasArchVersion(ARMv8p8);

shared/functions/extension/HaveFeatHBC

// HaveFeatHBC()
// =============
// Returns TRUE if the BC instruction is implemented, and FALSE otherwise.

boolean HaveFeatHBC()
    return HasArchVersion(ARMv8p8);

shared/functions/extension/HaveFeatHCX

// HaveFeatHCX()
// =============
// Returns TRUE if HCRX_EL2 Trap Control register is implemented,
// and FALSE otherwise.

boolean HaveFeatHCX()
    return HasArchVersion(ARMv8p7);
shared/functions/extension/HaveFeatHPMNO

// HaveFeatHPMNO()
// ===============
// Returns TRUE if HDCR.HPMN or MDCR_EL2.HPMN is permitted to be 0 without
// generating UNPREDICTABLE behavior, and FALSE otherwise.

boolean HaveFeatHPMNO()
    return HasArchVersion(ARMv8p6) && HavePMUv3() && HaveFGTExt() && HaveEL(EL2);

shared/functions/extension/HaveFeatLS64

// HaveFeatLS64()
// ==============
// Returns TRUE if the LD64B, ST64B instructions are
// supported, and FALSE otherwise.

boolean HaveFeatLS64()
    return (HasArchVersion(ARMv8p7) &&
            boolean IMPLEMENTATION_DEFINED "Has Load Store 64-Byte instruction support");

shared/functions/extension/HaveFeatLS64_ACCDATA

// HaveFeatLS64_ACCDATA()
// =======================
// Returns TRUE if the ST64BV0 instruction is
// supported, and FALSE otherwise.

boolean HaveFeatLS64_ACCDATA()
    return (HasArchVersion(ARMv8p7) && HaveFeatLS64_V() &&
            boolean IMPLEMENTATION_DEFINED "Has Store 64-Byte EL0 with return instruction support");

shared/functions/extension/HaveFeatLS64_V

// HaveFeatLS64_V()
// ================
// Returns TRUE if the ST64BV instruction is
// supported, and FALSE otherwise.

boolean HaveFeatLS64_V()
    return (HasArchVersion(ARMv8p7) && HaveFeatLS64() &&
            boolean IMPLEMENTATION_DEFINED "Has Store 64-Byte with return instruction support");

shared/functions/extension/HaveFeatMOPS

// HaveFeatMOPS()
// ==============
// Returns TRUE if the CPY* and SET* instructions are supported, and FALSE otherwise.

boolean HaveFeatMOPS()
    return HasArchVersion(ARMv8p8);

shared/functions/extension/HaveFeatNMI

// HaveFeatNMI()
// =============
// Returns TRUE if the Non-Maskable Interrupt extension is
// implemented, and FALSE otherwise.
boolean HaveFeatNMI()
    return HasArchVersion(ARMv8p8);

shared/functions/extension/HaveFeatRPRES
    // HaveFeatRPRES()
    // ================
    // Returns TRUE if reciprocal estimate implements 12-bit precision
    // when FPSCR.AH=1, and FALSE otherwise.
    boolean HaveFeatRPRES()
        return (HasArchVersion(ARMv8p7) &&
                (boolean IMPLEMENTATION_DEFINED "Has increased Reciprocal Estimate and Square Root Estimate precision support") &&
                HaveAltFP());

shared/functions/extension/HaveFeatTIDCP1
    // HaveFeatTIDCP1()
    // ================
    // Returns TRUE if the SCTLR_EL1.TIDCP bit is implemented and the SCTLR_EL2.TIDCP bit
    // is implemented if EL2 is implemented.
    boolean HaveFeatTIDCP1()
        return HasArchVersion(ARMv8p8);

shared/functions/extension/HaveFeatWFxT
    // HaveFeatWFxT()
    // ================
    // Returns TRUE if WFET and WFIT instruction support is implemented,
    // and FALSE otherwise.
    boolean HaveFeatWFxT()
        return HasArchVersion(ARMv8p7);

shared/functions/extension/HaveFeatXS
    // HaveFeatXS()
    // ============
    // Returns TRUE if XS attribute and the TLBI and DSB instructions with nXS qualifier
    // are supported, and FALSE otherwise.
    boolean HaveFeatXS()
        return HasArchVersion(ARMv8p7);

shared/functions/extension/HaveFlagFormatExt
    // HaveFlagFormatExt()
    // ===================
    // Returns TRUE if flag format conversion instructions implemented.
    boolean HaveFlagFormatExt()
        return HasArchVersion(ARMv8p5);
shared/functions/extension/HaveFlagManipulateExt

// HaveFlagManipulateExt()
// ===============
// Returns TRUE if flag manipulate instructions are implemented.

boolean HaveFlagManipulateExt()
    return HasArchVersion(ARMv8p4);

shared/functions/extension/HaveFrintExt

// HaveFrintExt()
// =============
// Returns TRUE if FRINT instructions are implemented.

boolean HaveFrintExt()
    return HasArchVersion(ARMv8p5);

shared/functions/extension/HaveGTGExt

// HaveGTGExt()
// ============
// Returns TRUE if support for guest translation granule size is implemented.

boolean HaveGTGExt()
    return HasArchVersion(ARMv8p5);

shared/functions/extension/HaveHPMDExt

// HaveHPMDExt()
// =============

boolean HaveHPMDExt()
    return HavePMUv3p1();

shared/functions/extension/HaveIDSExt

// HaveIDSExt()
// ===========
// Returns TRUE if ID register handling feature is implemented.

boolean HaveIDSExt()
    return HasArchVersion(ARMv8p4);

shared/functions/extension/HaveIESB

// HaveIESB()
// =========

boolean HaveIESB()
    return (HaveRASExt() &&
        boolean IMPLEMENTATION_DEFINED "Has Implicit Error Synchronization Barrier");
shared/functions/extension/HaveInt8MatMulExt

// HaveInt8MatMulExt()
// ===================
// Returns TRUE if AArch64 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveInt8MatMulExt()
return (HasArchVersion(ARMv8p6) ||
(HasArchVersion(ARMv8p2) &&
 boolean IMPLEMENTATION_DEFINED "Has AArch64 Int8 Mat Mul extension");

shared/functions/extension/HaveLSE2Ext

// HaveLSE2Ext()
// =============
// Returns TRUE if LSE2 is implemented, and FALSE otherwise.

boolean HaveLSE2Ext()
return HasArchVersion(ARMv8p4);

shared/functions/extension/HaveMPAMExt

// HaveMPAMExt()
// =============
// Returns TRUE if MPAM is implemented, and FALSE otherwise.

boolean HaveMPAMExt()
return (HasArchVersion(ARMv8p2) &&
 boolean IMPLEMENTATION_DEFINED "Has MPAM extension");

shared/functions/extension/HaveMPAMv0p1Ext

// HaveMPAMv0p1Ext()
// =================
// Returns TRUE if MPAMv0p1 is implemented, and FALSE otherwise.

boolean HaveMPAMv0p1Ext()
return (HasArchVersion(ARMv8p6) &&
 HaveMPAMExt() &&
 boolean IMPLEMENTATION_DEFINED "Has enhanced MPAMv0p1 extension");

shared/functions/extension/HaveMPAMv1p1Ext

// HaveMPAMv1p1Ext()
// =================
// Returns TRUE if MPAMv1p1 is implemented, and FALSE otherwise.

boolean HaveMPAMv1p1Ext()
return (HasArchVersion(ARMv8p6) &&
 HaveMPAMExt() &&
 boolean IMPLEMENTATION_DEFINED "Has enhanced MPAMv1p1 extension");

shared/functions/extension/HaveMTE2Ext

// HaveMTE2Ext()
// =============
// Returns TRUE if MTE support is beyond EL0, and FALSE otherwise.
boolean HaveMTE2Ext()
    if !HasArchVersion(ARMv8p5) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE2 extension";

shared/functions/extension/HaveMTE3Ext

    // HaveMTE3Ext()
    // =============
    // Returns TRUE if MTE Asymmetric Fault Handling support is
    // implemented, and FALSE otherwise.

    boolean HaveMTE3Ext()
        return ((HasArchVersion(ARMv8p7) && HaveMTE2Ext()) || (HasArchVersion(ARMv8p5) &&
        boolean IMPLEMENTATION_DEFINED "Has MTE3 extension");

shared/functions/extension/HaveMTEEExt

    // HaveMTEEExt()
    // =============
    // Returns TRUE if instruction-only MTE implemented, and FALSE otherwise.

    boolean HaveMTEEExt()
        if !HasArchVersion(ARMv8p5) then
            return FALSE;
        if HaveMTE2Ext() then
            return TRUE;
        return boolean IMPLEMENTATION_DEFINED "Has MTE extension";

shared/functions/extension/HaveNV2Ext

    // HaveNV2Ext()
    // ===========
    // Returns TRUE if Enhanced Nested Virtualization is implemented.

    boolean HaveNV2Ext()
        return (HasArchVersion(ARMv8p4) && HaveNVExt() &&
        boolean IMPLEMENTATION_DEFINED "Has support for Enhanced Nested Virtualization");

shared/functions/extension/HaveNVExt

    // HaveNVExt()
    // ===========
    // Returns TRUE if Nested Virtualization is implemented.

    boolean HaveNVExt()
        return (HasArchVersion(ARMv8p3) &&
        boolean IMPLEMENTATION_DEFINED "Has Nested Virtualization");

shared/functions/extension/HaveNoSecurePMUDisableOverride

    // HaveNoSecurePMUDisableOverride()
    // ================================

    boolean HaveNoSecurePMUDisableOverride()
        return HasArchVersion(ARMv8p2);
shared/functions/extension/HaveNoninvasiveDebugAuth

  // HaveNoninvasiveDebugAuth()
  // =================================
  // Returns TRUE if the Non-invasive debug controls are implemented.

  boolean HaveNoninvasiveDebugAuth()
  return !HasArchVersion(ARMv8p4);

shared/functions/extension/HavePAN3Ext

  // HavePAN3Ext()
  // =============
  // Returns TRUE if SCTLR_EL1.EPAN and SCTLR_EL2.EPAN support is implemented,
  // and FALSE otherwise.

  boolean HavePAN3Ext()
  return HasArchVersion(ARMv8p7) || (HasArchVersion(ARMv8p1) &&
    boolean IMPLEMENTATION_DEFINED “Has PAN3 extension”);

shared/functions/extension/HavePANExt

  // HavePANExt()
  // ============

  boolean HavePANExt()
  return HasArchVersion(ARMv8p1);

shared/functions/extension/HavePMUv3

  // HavePMUv3()
  // ===========
  // Returns TRUE if the Performance Monitors extension is implemented, and FALSE otherwise.

  boolean HavePMUv3()
  return boolean IMPLEMENTATION_DEFINED “Has Performance Monitors extension”;

shared/functions/extension/HavePMUv3TH

  // HavePMUv3TH()
  // =============
  // Returns TRUE if the PMUv3 threshold extension is implemented, and FALSE otherwise.

  boolean HavePMUv3TH()
  return (HasArchVersion(ARMv8p8) && HavePMUv3() &&
    boolean IMPLEMENTATION_DEFINED “Has PMUv3 threshold extension”);

shared/functions/extension/HavePMUv3p1

  // HavePMUv3p1()
  // =============
  // Returns TRUE if the Performance Monitors extension is implemented, and FALSE otherwise.

  boolean HavePMUv3p1()
  return HasArchVersion(ARMv8p1) && HavePMUv3();
shared/functions/extension/HavePMUv3p4

// HavePMUv3p4()
// =============
// Returns TRUE if the PMUv3.4 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p4()
return HasArchVersion(ARMv8p4) && HavePMUv3();

shared/functions/extension/HavePMUv3p5

// HavePMUv3p5()
// =============
// Returns TRUE if the PMUv3.5 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p5()
return HasArchVersion(ARMv8p5) && HavePMUv3();

shared/functions/extension/HavePMUv3p7

// HavePMUv3p7()
// =============
// Returns TRUE if the PMUv3.7 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p7()
return HasArchVersion(ARMv8p7) && HavePMUv3();

shared/functions/extension/HavePageBasedHardwareAttributes

// HavePageBasedHardwareAttributes()
// =================================

boolean HavePageBasedHardwareAttributes()
return HasArchVersion(ARMv8p2);

shared/functions/extension/HavePrivATExt

// HavePrivATExt()
// ===============

boolean HavePrivATExt()
return HasArchVersion(ARMv8p2);

shared/functions/extension/HaveQRDMLAHExt

// HaveQRDMLAHExt()
// ================

boolean HaveQRDMLAHExt()
return HasArchVersion(ARMv8p1);

shared/functions/extension/HaveRASExt

// HaveRASExt()
// ============

boolean HaveRASExt()
return (HasArchVersion(ARMv8p2) ||
    boolean IMPLEMENTATION_DEFINED "Has RAS extension");

shared/functions/extension/HaveRME

// HaveRME()
// =========
// Returns TRUE if the Realm Management Extension is implemented, and FALSE otherwise.

boolean HaveRME()
    return boolean IMPLEMENTATION_DEFINED "Has RME extension";

shared/functions/extension/HaveRNG

// HaveRNG()
// =========
// Returns TRUE if Random Number Generator extension support is implemented and FALSE otherwise.

boolean HaveRNG()
    return HasArchVersion(ARMv8p5) && boolean IMPLEMENTATION_DEFINED "Has RNG extension";

shared/functions/extension/HaveSBExt

// HaveSBExt()
// ===========
// Returns TRUE if support for SB is implemented, and FALSE otherwise.

boolean HaveSBExt()
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SB extension";

shared/functions/extension/HaveSSBSExt

// HaveSSBSExt()
// =============
// Returns TRUE if support for SSBS is implemented, and FALSE otherwise.

boolean HaveSSBSExt()
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SSBS extension";

shared/functions/extension/HaveSecureEL2Ext

// HaveSecureEL2Ext()
// ==================
// Returns TRUE if Secure EL2 is implemented.

boolean HaveSecureEL2Ext()
    return HasArchVersion(ARMv8p4);

shared/functions/extension/HaveSecureExtDebugView

// HaveSecureExtDebugView()
// =========================
// Returns TRUE if support for Secure and Non-secure views of debug peripherals is implemented.
boolean HaveSecureExtDebugView()
    return HasArchVersion(ARMv8p4);

shared/functions/extension/HaveSelfHostedTrace

    // HaveSelfHostedTrace()
    // =====================
    boolean HaveSelfHostedTrace()
    return HasArchVersion(ARMv8p4);

shared/functions/extension/HaveSmallTranslationTblExt

    // HaveSmallTranslationTblExt()
    // ============================
    // Returns TRUE if Small Translation Table Support is implemented.
    boolean HaveSmallTranslationTableExt()
    return (HasArchVersion(ARMv8p4) &&
        boolean IMPLEMENTATION_DEFINED "Has Small Translation Table extension");

shared/functions/extension/HaveSoftwareLock

    // HaveSoftwareLock()
    // =================
    // Returns TRUE if Software Lock is implemented.
    boolean HaveSoftwareLock(Component component)
    if Havev8p4Debug() then
        return FALSE;
    if HaveDoPD() && component != Component_CTI then
        return FALSE;
    case component of
        when Component_Debug
            return boolean IMPLEMENTATION_DEFINED "Debug has Software Lock";
        when Component_PMU
            return boolean IMPLEMENTATION_DEFINED "PMU has Software Lock";
        when Component_CTI
            return boolean IMPLEMENTATION_DEFINED "CTI has Software Lock";
        otherwise
            Unreachable();
    
shared/functions/extension/HaveStage2MemAttrControl

    // HaveStage2MemAttrControl()
    // =================---------
    // Returns TRUE if support for Stage2 control of memory types and cacheability
    // attributes is implemented.
    boolean HaveStage2MemAttrControl()
    return HasArchVersion(ARMv8p4);

shared/functions/extension/HaveStatisticalProfiling

    // HaveStatisticalProfiling()
    // ==========================
    // Returns TRUE if Statistical Profiling Extension is implemented,
    // and FALSE otherwise.
boolean HaveStatisticalProfiling()
    return HasArchVersion(ARMv8p2);

shared/functions/extension/HaveStatisticalProfilingv1p1

    // HaveStatisticalProfilingv1p1()
    // ------------------------------------------
    // Returns TRUE if the SPEv1p1 extension is implemented, and FALSE otherwise.

    boolean HaveStatisticalProfilingv1p1()
    return (HasArchVersion(ARMv8p3) &&
            boolean IMPLEMENTATION_DEFINED "Has SPEv1p1 extension");

shared/functions/extension/HaveStatisticalProfilingv1p2

    // HaveStatisticalProfilingv1p2()
    // ------------------------------------------
    // Returns TRUE if the SPEv1p2 extension is implemented, and FALSE otherwise.

    boolean HaveStatisticalProfilingv1p2()
    return (HasArchVersion(ARMv8p7) && HaveStatisticalProfilingv1p1() &&
            boolean IMPLEMENTATION_DEFINED "Has SPEv1p2 extension");

shared/functions/extension/HaveTME

    // HaveTME()
    // =========

    boolean HaveTME()
    return boolean IMPLEMENTATION_DEFINED "Has Transactional Memory extension";

shared/functions/extension/HaveTWEDExt

    // HaveTWEDExt()
    // =============
    // Returns TRUE if Delayed Trapping of WFE instruction support is implemented,
    // and FALSE otherwise.

    boolean HaveTWEDExt()
    return boolean IMPLEMENTATION_DEFINED "Has TWED extension";

shared/functions/extension/HaveTraceBufferExtension

    // HaveTraceBufferExtension()
    // ===========================
    // Returns TRUE if Trace Buffer Extension is implemented, and FALSE otherwise.

    boolean HaveTraceBufferExtension()
    return boolean IMPLEMENTATION_DEFINED "Trace Buffer Extension implemented";

shared/functions/extension/HaveTraceExt

    // HaveTraceExt()
    // ===============
    // Returns TRUE if Trace functionality as described by the Trace Architecture
    // is implemented.
boolean HaveTraceExt()
   return boolean IMPLEMENTATION_DEFINED "Has Trace Architecture functionality";

shared/functions/extension/HaveTrapLoadStoreMultipleDeviceExt
   // HaveTrapLoadStoreMultipleDeviceExt()
   // -----------------------------------
   boolean HaveTrapLoadStoreMultipleDeviceExt()
      return HasArchVersion(ARMv8p2);

shared/functions/extension/HaveUAOExt
   // HaveUAOExt()
   // ===========
   boolean HaveUAOExt()
      return HasArchVersion(ARMv8p2);

shared/functions/extension/HaveV82Debug
   // HaveV82Debug()
   // =============
   boolean HaveV82Debug()
      return HasArchVersion(ARMv8p2);

shared/functions/extension/HaveVirtHostExt
   // HaveVirtHostExt()
   // ================
   boolean HaveVirtHostExt()
      return HasArchVersion(ARMv8p1);

shared/functions/extension/Havev8p4Debug
   // Havev8p4Debug()
   // ===============
   // Returns TRUE if support for the Debugv8p4 feature is implemented and FALSE otherwise.
   boolean Havev8p4Debug()
      return HasArchVersion(ARMv8p4);

shared/functions/extension/Havev8p8Debug
   // Havev8p8Debug()
   // ===============
   // Returns TRUE if support for the Debugv8p8 feature is implemented and FALSE otherwise.
   boolean Havev8p8Debug()
      return HasArchVersion(ARMv8p8);
shared/functions/extension/InsertIESBBeforeException

// InsertIESBBeforeException()
// ===========================
// Returns an implementation defined choice whether to insert an implicit error synchronization
// barrier before exception.
// If SCTLR_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable
// SError interrupt must be taken before executing any instructions in the exception handler.
// However, this can be before the branch to the exception handler is made.

boolean InsertIESBBeforeException(bits(2) el)
return (HaveIESB() &&
boolean IMPLEMENTATION_DEFINED "Has Implicit Error Synchronization Barrier before
Exception");

shared/functions/extension/IsG1ActivityMonitorImplemented

// Returns TRUE if a G1 activity monitor is implemented for the counter
// and FALSE otherwise.
boolean IsG1ActivityMonitorImplemented(integer i);

shared/functions/extension/IsG1ActivityMonitorOffsetImplemented

// Returns TRUE if a G1 activity monitor offset is implemented for the counter,
// and FALSE otherwise.
boolean IsG1ActivityMonitorOffsetImplemented(integer i);

shared/functions/externalaborts/HandleExternalAbort

// HandleExternalAbort()
// =====================
// Takes a Synchronous/Asynchronous abort based on fault.
HandleExternalAbort(PhysMemRetStatus memretstatus, boolean iswrite,
AddressDescriptor memaddrdesc, integer size,
AccessDescriptor accdesc)
assert (memretstatus.statuscode IN {Fault_SyncExternal, Fault_AsyncExternal} ||
( !HaveRASExt() && memretstatus.statuscode IN {Fault_SyncParity,
Fault_AsyncParity}) ) ;

fault = NoFault();
fault.statuscode = memretstatus.statuscode;
fault.write = iswrite;
fault.extflag = memretstatus.extflag;
fault.acctype = memretstatus.acctype;
// It is implementation specific whether External aborts signaled
// in-band synchronously are taken synchronously or asynchronously
if ( IsExternalSyncAbort(fault) &&
!IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
size, accdesc)) then
if fault.statuscode == Fault_SyncParity then
fault.statuscode = Fault_AsyncParity;
else
fault.statuscode = Fault_AsyncExternal;
fi
if HaveRASExt() then
fault.errortype = PEErrorState(memretstatus);
else
fault.errortype = bits(2) UNKNOWN;
fi
if IsExternalSyncAbort(fault) then
if UsingAArch32() then
AArch32.Abort(memaddrdesc.vaddress<31:0>, fault);
else

else
   AArch64.Abort(memaddrdesc.vaddress, fault);
else
   PendSErrorInterrupt(fault);

shared/functions/externalaborts/HandleExternalReadAbort

// HandleExternalReadAbort()
// =========================
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory read.
HandleExternalReadAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc,
                        integer size, AccessDescriptor accdesc)
   iswrite = FALSE;
   HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);

shared/functions/externalaborts/HandleExternalTTWAbort

// HandleExternalTTWAbort()
// ========================
// Take Asynchronous abort or update FaultRecord for Translation Table Walk
// based on PhysMemRetStatus.
FaultRecord HandleExternalTTWAbort(PhysMemRetStatus memretstatus, boolean iswrite,
                                   AddressDescriptor memaddrdesc,
                                   AccessDescriptor accdesc, integer size,
                                   FaultRecord input_fault)
   output_fault = input_fault;
   output_fault.extflag = memretstatus.extflag;
   output_fault.statuscode = memretstatus.statuscode;
   if (IsExternalSyncAbort(output_fault) &&
       !IsExternalAbortTakenSynchronously(memretstatus, iswrite,
                                           memaddrdesc,
                                           size, accdesc)) then
      if output_fault.statuscode == Fault_SyncParity then
         output_fault.statuscode = Fault_AsyncParity;
      else
         output_fault.statuscode = Fault_AsyncExternal;
   else
      output_fault.statuscode = Fault_AsyncExternal;
   if IsExternalSyncAbort(output_fault) then
      if output_fault.statuscode == Fault_SyncParity then
         output_fault.statuscode = Fault_SyncExternalOnWalk;
      else
         output_fault.statuscode = Fault_SyncExternalOnWalk;
   if HaveRASExt() then
      output_fault.errortype = PErrorState(memretstatus);
   else
      output_fault.errortype = bits(2) UNKNOWN;
   if !IsExternalSyncAbort(output_fault) then
      PendSErrorInterrupt(output_fault);
   output_fault.statuscode = Fault_None;
   return output_fault;

shared/functions/externalaborts/HandleExternalWriteAbort

// HandleExternalWriteAbort()
// ==========================
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory write.
HandleExternalWriteAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc,
  integer size, AccessDescriptor accdesc)
  iswrite = TRUE;
  HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);

shared/functions/externalaborts/IsExternalAbortTakenSynchronously

// Return an implementation specific value:
// TRUE if the fault returned for the access can be taken synchronously,
// FALSE otherwise.
//
// This might vary between accesses, for example depending on the error type
// or memory type being accessed.
// External aborts on data accesses and translation table walks on data accesses
// can be either synchronous or asynchronous.
//
// when FEAT_DoubleFault is not implemented, External aborts on instruction
// fetches and translation table walks on instruction fetches can be either
// synchronous or asynchronous.
// when FEAT_DoubleFault is implemented, all External abort exceptions on
// instruction fetches and translation table walks on instruction fetches
// must be synchronous.
boolean IsExternalAbortTakenSynchronously(PhysMemRetStatus memstatus,
  boolean iswrite,
  AddressDescriptor desc,
  integer size,
  AccessDescriptor accdesc);

shared/functions/externalaborts/PEErrorState

constant bits(2) Sync_UC   = '10'; // Synchronous Uncontainable
constant bits(2) Sync_UER  = '00'; // Synchronous Recoverable
constant bits(2) Sync_UEO  = '11'; // Synchronous Restartable
constant bits(2) ASync_UC  = '00'; // ASynchronous Uncontainable
constant bits(2) ASync_UEU = '01'; // ASynchronous Unrecoverable
constant bits(2) ASync_UER = '11'; // ASynchronous Recoverable
constant bits(2) ASync_UEO = '10'; // ASynchronous Restartable

bits(2) PEErrorState(PhysMemRetStatus memstatus);

shared/functions/externalaborts/PendSErrorInterrupt

// Pend the SError.
PendSErrorInterrupt(FaultRecord fault);

shared/functions/float/bfloat/BFDotAdd

// BFDotAdd()
// =========
// BFloat16 2-way dot-product and add to single-precision
// result = addend + op1_a*op2_a + op1_b*op2_b

bits(32) BFDotAdd(bits(32) addend, bits(16) op1_a, bits(16) op1_b,
  bits(16) op2_a, bits(16) op2_b, FPCRType fpcr_in)
  FPCRType fpcr = fpcr_in;
  bits(32) prod;
  bits(32) result;
  if !HaveEBF16() || fpcr.EBF == '0' then  // Standard BFloat16 behaviors
prod = FPAdd_BF16(BFMulH(op1_a, op2_a), BFMulH(op1_b, op2_b));
result = FPAdd_BF16(addend, prod);
else // Extended BFloat16 behaviors
    boolean isbfloat16 = TRUE;
    boolean fpexc = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1'; // Generate default NaN values
    prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
    result = FPAdd(addend, prod, fpcr, fpexc);
return result;

shared/functions/float/bfloat/BFMatMulAdd

// BFMatMulAdd()
// =============
// BFloat16 matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])
bits(N) BFMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2)
assert N == 128;
bits(N) result;
bits(32) sum;
for i = 0 to 1
    for j = 0 to 1
        sum = Elem[addend, 2*i + j, 32];
        for k = 0 to 1
            bits(16) elt1_a = Elem[op1, 4*i + 2*k + 0, 16];
            bits(16) elt1_b = Elem[op1, 4*i + 2*k + 1, 16];
            bits(16) elt2_a = Elem[op2, 4*j + 2*k + 0, 16];
            bits(16) elt2_b = Elem[op2, 4*j + 2*k + 1, 16];
            sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
        Elem[result, 2*i + j, 32] = sum;
return result;

shared/functions/float/bfloat/BFMulAddH

// BFMulAddH()
// ===========
// Used by BFMLALB and BFMLALT instructions.
bits(N) BFMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr_in)
bits(N) value1 = op1 : Zeros(N DIV 2);
bits(N) value2 = op2 : Zeros(N DIV 2);
FPCRType fpcr = fpcr_in;
boolean altfp = HaveAltFP() && fpcr.AH == '1'; // When TRUE:
boolean fpexc = !altfp; // Do not generate floating point exceptions
if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
if altfp then fpcr.RMode = '00'; // Use RNE rounding mode
return FPMulAdd(addend, value1, value2, fpcr, fpexc);

shared/functions/float/bfloat/ BFUI

// BFUI()
// =========
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.
bits(32) BFUI(bits(16) op1, bits(16) op2)
bits(32) result;

FPCType fpcr = FPCR[];
(type1, sign1, value1) = BFUnpack(op1);
(type2, sign2, value2) = BFUnpack(op2);
if type1 == FPType_QNaN || type2 == FPType_QNaN then
  result = FPDefaultNaN(fpcr, 32);
else
  inf1 = (type1 == FPType_Infinity);
  inf2 = (type2 == FPType_Infinity);
  zero1 = (type1 == FPType_Zero);
  zero2 = (type2 == FPType_Zero);
  if (inf1 && zero2) || (zero1 && inf2) then
    result = FPDefaultNaN(fpcr, 32);
  elsif inf1 || inf2 then
    result = FPInfinity(sign1 XOR sign2, 32);
  elsif zero1 || zero2 then
    result = FPZero(sign1 XOR sign2, 32);
  else
    result = BFRound(value1*value2);

return result;

shared/functions/float/bfloat/BFNeg

// BFNeg()
// ========
bits(16) BFNeg(bits(16) op)
  return NOT(op<15>) : op<14:0>;

shared/functions/float/bfloat/BFRound

// BFRound()
// =========
// Converts a real number OP into a single-precision value using the
// Round to Odd rounding mode and following BFloat16 computation behaviors.

bits(32) BFRound(real op)

  assert op != 0.0;
  bits(32) result;
  // Format parameters - minimum exponent, numbers of exponent and fraction bits.
  minimum_exp = -126;  E = 8;  F = 23;
  // Split value into sign, unrounded mantissa and exponent.
  bit sign;
  real mantissa;
  if op < 0.0 then
    sign = '1';  mantissa = -op;
  else
    sign = '0';  mantissa = op;
  exponent = 0;
  while mantissa < 1.0 do
    mantissa = mantissa * 2.0;  exponent = exponent - 1;
  while mantissa >= 2.0 do
    mantissa = mantissa / 2.0;  exponent = exponent + 1;
  // Fixed Flush-to-zero.
  if exponent < minimum_exp then
    return FPZero(sign, 32);
  // Start creating the exponent value for the result. Start by biasing the actual exponent
// so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
biased_exp = Max((exponent - minimum_exp) + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
error = mantissa * 2.0^F - Real(int_mant);

// Round to Odd
if error != 0.0 then
  int_mant<0> = '1';

// Deal with overflow and generate result.
if biased_exp >= 2^E - 1 then
  result = FPInfinity(sign, 32);  // Overflows generate appropriately-signed Infinity
else
  result = sign : biased_exp<30-F:0> : int_mant<F-1:0>;

return result;

shared/functions/float/bfloat/BFUnpack

// BFUnpack()
// =========
// Unpacks a BFloat16 or single-precision value into its type,
// sign bit and real number that it represents.
// The real number result has the correct sign for numbers and infinities,
// is very large in magnitude for infinities, and is 0.0 for NaNs.
// (These values are chosen to simplify the description of
// comparisons and conversions.)

(FPType, bit, real) BFUnpack(bits(N) fpval)
assert N IN {16,32};

bit sign;
bits(8) exp;
bits(23) frac;
if N == 16 then
  sign   = fpval<15>;
  exp    = fpval<14:7>;
  frac   = fpval<6:0> : Zeros(16);
else  // N == 32
  sign   = fpval<31>;
  exp    = fpval<30:23>;
  frac   = fpval<22:0>;

FPType fptype;
real value;
if IsZero(exp) then
  fptype = FPType_Zero;  value = 0.0;  // Fixed Flush to Zero
elsif IsOnes(exp) then
  if IsZero(frac) then
    fptype = FPType_Infinity;  value = 2.0^1000000;
  else    // no SNaN for BF16 arithmetic
    fptype = FPType_QNaN;  value = 0.0;
  else
    fptype = FPType_Nonzero;
    value = 2.0^(UInt(exp)-127) * (1.0 + Real(UInt(frac)) * 2.0^-23);
else
  if sign == '1' then value = -value;
return (fptype, sign, value);
shared/functions/float/bfloat/FPAdd_BF16

// FPAdd_BF16()
// ============
// Single-precision add following BFloat16 computation behaviors.

bits(32) FPAdd_BF16(bits(32) op1, bits(32) op2)

bits(32) result;

FPCRTypetype fpcr = FPCR[];
(type1,sign1,value1) = BFUnpack(op1);
(type2,sign2,value2) = BFUnpack(op2);
if type1 == FPTypenQNaN || type2 == FPTypenQNaN then
    result = FPDefaultNaN(fpcr, 32);
else
    inf1 = (type1 == FPTypenInfinity);
    inf2 = (type2 == FPTypenInfinity);
    zero1 = (type1 == FPTypenZero);
    zero2 = (type2 == FPTypenZero);
    if inf1 && inf2 && sign1 == NOT(sign2) then
        result = FPDefaultNaN(fpcr, 32);
    elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
        result = FPInfinity('0', 32);
    elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
        result = FPInfinity('1', 32);
    elsif zero1 && zero2 && sign1 == sign2 then
        result = FPZero(sign1, 32);
    else
        result_value = value1 + value2;
        if result_value == 0.0 then
            result = FPZero('0', 32);    // Positive sign when Round to Odd
        else
            result = BFRound(result_value);
    fi
return result;

shared/functions/float/bfloat/FPConvertBF

// FPConvertBF()
// =============
// Converts a single-precision OP to BFloat16 value with using rounding mode of
// Round to Nearest Even when executed from AArch64 state and
// FPCR.AH == '1', otherwise rounding is controlled by FPCR/FPSCR.

bits(16) FPConvertBF(bits(32) op, FPCRTypetype fpcr_in, FPRounding rounding_in)

FPCRTypetype fpcr = fpcr_in;
FPRoundingrounding = rounding_in;
bits(32) result;

boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
boolean fpexc = !altfp;                         // Generate no floating-point exceptions
if altfp then fpcr.<FIZ,FZ> = '11';             // Flush denormal input and output to zero
if altfp then rounding = FPRounding_TIEEVEN;    // Use RNE rounding mode

// Unpack floating-point operand, with always flush-to-zero if fpcr.AH == '1'.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

if fptype == FPTypenSNaN || fptype == FPTypenQNaN then
    if fpcr.DN == '1' then
        result = FPDefaultNaN(fpcr, 32);
    else
        result = FPConvertNaN(op, 32);
    fi
else
    if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    fi
fi
else
    result = FPConvertNaN(op, 32);
fi
return result;
result = FPInfinity(sign, 32);
elsif fptype == FPType_Zero then
    result = FPZero(sign, 32);
else
    result = FPRoundBF(value, fpcr, rounding, fpexc);

// Returns correctly rounded BF16 value from top 16 bits
return result<31:16>;

// FPConvertBF()
// =============
// Converts a single-precision operand to BFloat16 value.

bits(16) FPConvertBF(bits(32) op, FPCRType fpcr)
return FPConvertBF(op, fpcr, FPRoundingMode(fpcr));

shared/functions/float/bfloat/FPRoundBF

// FPRoundBF()
// =============
// Converts a real number OP into a BFloat16 value using the supplied
// rounding mode RMODE. The 'fpexc' argument controls the generation of
// floating-point exceptions.

bits(32) FPRoundBF(real op, FPCRType fpcr, FPRounding rounding, boolean fpexc)
    boolean isbfloat16 = TRUE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, 32);

shared/functions/float/fixedtofp/FixedToFP

// FixedToFP()
// =============
// Convert M-bit fixed point 'op' with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNING.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding, integer N)

assert N IN {16,32,64};
assert M IN {16,32,64};
bits(N) result;
assert fbits >= 0;
assert rounding != FPRounding_ODD;
// Correct signed-ness
int_operand = Int(op, unsigned);
// Scale by fractional bits and generate a real value
real_operand = Real(int_operand) / 2.0^fbits;
if real_operand == 0.0 then
    result = FPZero('0', N);
else
    result = FPRound(real_operand, fpcr, rounding, N);
return result;

shared/functions/float/fpabs/FPAbs

// FPAbs()
// =============
bits(N) FPAbs(bits(N) op)

assert N IN {16,32,64};
if !UsingAArch32() && HaveAltFP() then
    FPCRType fpcr = FPCR[];
    if fpcr.AH == '1' then
        (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
        if fptype IN {FPType_SNaN, FPType_QNaN} then
            return op;        // When fpcr.AH=1, sign of NaN has no consequence
    return '0' : op<N-2:0>;

shared/functions/float/fpadd/FPAdd

// FPAdd()
// =========

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean fpexc = TRUE;       // Generate floating-point exceptions
    return FPAdd(op1, op2, fpcr, fpexc);

// FPAdd()
// =========

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)

assert N IN {16,32,64};
rounding = FPRoundingMode(fpcr);
(type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
(type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

(done, result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
if !done then
    inf1  = (type1 == FPType_Infinity);  inf2  = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);      zero2 = (type2 == FPType_Zero);
    if inf1 && inf2 && sign1 == NOT(sign2) then
        result = FPDefaultNaN(fpcr, N);
    if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
        result = FPInfinity('0', N);
    elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
        result = FPInfinity('1', N);
    elsif zero1 && zero2 && sign1 == sign2 then
        result = FPZero(sign1, N);
    else
        result_value = value1 + value2;
        if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
            result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
            result = FPZero(result_sign, N);
        else
            result = FPRound(result_value, fpcr, rounding, fpexc, N);
        if fpexc then FPProcessDenorms(type1, type2, N, fpcr);
        return result;

shared/functions/float/fpcompare/FPCompare

// FPCompare()
// ===========

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRType fpcr)
shared/functions/float/fpcompareeq/FPCompareEQ

// FPCompareEQ()
// ============

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRType fpchr)
assert N IN {16, 32, 64};
(type1,sign1,value1) = FPUnpack(op1, fpchr);
(type2,sign2,value2) = FPUnpack(op2, fpchr);

boolean result;
if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
    result = FALSE;
else
    // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    result = (value1 == value2);

    FPProcessDenorms(type1, type2, N, fpchr);
return result;

shared/functions/float/fpcomparege/FPCompareGE

// FPCompareGE()
// ============

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRType fpchr)
assert N IN {16, 32, 64};
(type1,sign1,value1) = FPUnpack(op1, fpchr);
(type2,sign2,value2) = FPUnpack(op2, fpchr);

boolean result;
if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
    result = FALSE;
else
    // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    result = (value1 >= value2);

    FPProcessDenorms(type1, type2, N, fpchr);
return result;
return result;

shared/functions/float/fpcomparegt/FPCompareGT

// FPCompareGT()
// =============

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRType fpcr)

assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

boolean result;
if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
    result = FALSE;
    FPProcessException(FPExc_InvalidOp, fpcr);
else
    // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    result = (value1 > value2);
    FPProcessDenorms(type1, type2, N, fpcr);

return result;

shared/functions/float/fpconvert/FPConvert

// FPConvert()
// ===========

// Convert floating point 'op' with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

bits(M) FPConvert(bits(N) op, FPCRType fpcr, FPRounding rounding, integer M)

assert M IN {16,32,64};
assert N IN {16,32,64};
bits(M) result;

// Unpack floating-point operand optionally with flush-to-zero.
(fptype,sign,value) = FPUnpackCV(op, fpcr);
alt_hp = (M == 16) && (fpcr.AHP == '1');
if fptype == FPType_SNaN || fptype == FPType_QNaN then
    if alt_hp then
        result = FPZero(sign, M);
    elsif fpcr.DN == '1' then
        result = FPDefaultNaN(fpcr, M);
    else
        result = FPConvertNaN(op, M);
    if fptype == FPType_SNaN || alt_hp then
        FPProcessException(FPExc_InvalidOp,fpcr);
    elsif fptype == FPType_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        result = FPInfinity(sign, M);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, M);
    else
result = FPRoundCV(value, fpcr, rounding, M);
FPProcessDenorm(fptype, N, fpcr);

return result;

// FPConvert()
// ===========

bits(M) FPConvert(bits(N) op, FPCRType fpcr, integer M)
return FPConvert(op, fpcr, FPRoundingMode(fpcr), M);

shared/functions/float/fpconvertnan/FPConvertNaN

// FPConvertNaN()
// ================
// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op, integer M)

assert N IN {16,32,64};
assert M IN {16,32,64};
bits(M) result;
bits(51) frac;

sign = op<N-1>;

// Unpack payload from input NaN
case N of
  when 64 frac = op<50:0>;
  when 32 frac = op<21:0>:Zeros(29);
  when 16 frac = op<8:0>:Zeros(42);
// Repack payload into output NaN, while
// converting an SNaN to a QNaN.
case M of
  when 64 result = sign:Ones(M-52):frac;
  when 32 result = sign:Ones(M-23):frac<50:29>;
  when 16 result = sign:Ones(M-10):frac<50:42>;

return result;

shared/functions/float/fpcrtype/FPCRType
type FPCRType;

shared/functions/float/fpdecoderm/FPDecodeRM

// FPDecodeRM()
// ==============
// Decode most common AArch32 floating-point rounding encoding.
FPRounding FPDecodeRM(bits(2) rm)
FPRounding result;

case rm of
  when '00' result = FPRounding_TIEAWAY; // A
  when '01' result = FPRounding_TIEEVEN; // N
  when '10' result = FPRounding_POSINF; // P
  when '11' result = FPRounding_NEGINF; // M

return result;
shared/functions/float/fpdecoderounding/FPDecodeRounding

// FPDecodeRounding()
// ================

// Decode floating-point rounding mode and common AArch64 encoding.
FPRounding FPDecodeRounding(bits(2) rmode)
case rmode of
  when '00' return FPRounding_TIEEVEN; // N
  when '01' return FPRounding_POSINF;  // P
  when '10' return FPRounding_NEGINF;  // M
  when '11' return FPRounding_ZERO;    // Z

shared/functions/float/fpdefaultnan/FPDefaultNaN

// FPDefaultNaN()
// ===============

bits(N) FPDefaultNaN(integer N)
  FPCRType fpcr = FPCR[];
  return FPDefaultNaN(fpcr, N);

bits(N) FPDefaultNaN(FPCRType fpcr, integer N)

assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
bit sign = if HaveAltFP() && !UsingAArch32() then fpcr.AH else '0';

bits(E) exp  = Ones(E);
bits(F) frac = '1':Zeros(F-1);
return sign : exp : frac;

shared/functions/float/fpdiv/FPDiv

// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRType fpcr)

assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  inf1  = type1 == FPType_Infinity;
  inf2  = type2 == FPType_Infinity;
  zero1 = type1 == FPType_Zero;
  zero2 = type2 == FPType_Zero;
  if (inf1 && inf2) || (zero1 && zero2) then
    result = FPDefaultNaN(fpcr, N);
    FPProcessException(FPExc_InvalidOp, fpcr);
  elsif inf1 || zero2 then
    result = FPInfinity(sign1 EOR sign2, N);
    if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr);
  elsif zero1 || inf2 then
    result = FPZero(sign1 EOR sign2, N);
  else
    result = FPRound(value1/value2, fpcr, N);
if !zero2 then
FPProcessDenorms(type1, type2, N, fpcr);

return result;

shared/functions/float/fpdot/FPDot

// FPDot()
// ========
// Calculates single-precision result of 2-way 16-bit floating-point dot-product
// with a single rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'isbfloat16'
// determines whether input operands are BFloat16 or half-precision type.
// and 'fpexc' controls the generation of floating-point exceptions.

bits(N) FPDot(bits(N DIV 2) op1_a, bits(N DIV 2) op1_b, bits(N DIV 2) op2_a,
bits(N DIV 2) op2_b, FPCRType fpcr, boolean isbfloat16)

// Generate floating-point exceptions
return FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);

bits(N) FPDot(bits(N DIV 2) op1_a, bits(N DIV 2) op1_b, bits(N DIV 2) op2_a,
bits(N DIV 2) op2_b, FPCRType fpcr_in, boolean isbfloat16, boolean fpexc)

FPCRType fpcr = fpcr_in;
assert N == 32;
bits(N) result;
boolean done;
fpcr.AHP = '0';           // Ignore alternative half-precision option
rounding = FPRoundingMode(fpcr);

(type1_a,sign1_a,value1_a) = FPUnpackBase(op1_a, fpcr, fpexc, isbfloat16);
(type1_b,sign1_b,value1_b) = FPUnpackBase(op1_b, fpcr, fpexc, isbfloat16);
(type2_a,sign2_a,value2_a) = FPUnpackBase(op2_a, fpcr, fpexc, isbfloat16);
(type2_b,sign2_b,value2_b) = FPUnpackBase(op2_b, fpcr, fpexc, isbfloat16);

inf1_a = (type1_a == FPType_Infinity); zero1_a = (type1_a == FPType_Zero);
inf1_b = (type1_b == FPType_Infinity); zero1_b = (type1_b == FPType_Zero);
in2_a = (type2_a == FPType_Infinity); zero2_a = (type2_a == FPType_Zero);
in2_b = (type2_b == FPType_Infinity); zero2_b = (type2_b == FPType_Zero);

(done,result) = FPProcessNaNs4(type1_a, type1_b, type2_a, type2_b,
op1_a, op1_b, op2_a, op2_b, fpcr, fpexc);

if (((inf1_a && zero2_a) || (zero1_a && inf2_a)) &&
((inf1_b && zero2_b) || (zero1_b && inf2_b))) then
result = FPDefaultNaN(fpcr, N);
if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
if !done then
// Determine sign and type products will have if it does not cause an Invalid
// Operation.
signPa = sign1_a EOR sign2_a;
signPb = sign1_b EOR sign2_b;
inPa = inf1_a || inf2_a;
inPb = inf1_b || inf2_b;
zeroPa = zero1_a || zero2_a;
zeroPb = zero1_b || zero2_b;

// Non NaN-generated Invalid Operation cases are multiples of zero
// by infinity and additions of opposite-signed infinities.
invalidop = ((inf1_a && zero2_a) || (zero1_a && inf2_a) ||
(inf1_b && zero2_b) || (zero1_b && inf2_b) || (infPa && infPb && signPa != signPb));

if invalidop then
result = FPDefaultNaN(fpcr, N);
if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
// Other cases involving infinities produce an infinity of the same sign.
elsif (infPa && signPa == '0') || (infPb && signPb == '0') then
    result = FPInfinity('0', N);
elsif (infPa && signPa == '1') || (infPb && signPb == '1') then
    result = FPInfinity('1', N);

// Cases where the result is exactly zero and its sign is not determined by the
// rounding mode are additions of same-signed zeros.
elsif zeroPa && zeroPb && signPa == signPb then
    result = FPZero(signPa, N);

// Otherwise calculate fused sum of products and round it.
else
    result_value = (value1_a * value2_a) + (value1_b * value2_b);
    if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
        result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
        result = FPZero(result_sign, N);
    else
        result = FPRound(result_value, fpcr, rounding, fpexc, N);
    return result;

shared/functions/float/fpdot/FPDotAdd

// FPDotAdd()
// ============
// Half-precision 2-way dot-product and add to single-precision.

bits(N) FPDotAdd(bits(N) addend, bits(N DIV 2) op1_a, bits(N DIV 2) op1_b,
    bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRType fpcr)
assert N == 32;

bits(N) prod;
boolean isbfloat16 = FALSE;
boolean fpexc = TRUE; // Generate floating-point exceptions
prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
result = FPAdd(addend, prod, fpcr, fpexc);
return result;

shared/functions/float/fpdot/FPDotAdd_ZA

// FPDotAdd_ZA()
// ==============
// Half-precision 2-way dot-product and add to single-precision
// for SME ZA-targeting instructions.

bits(N) FPDotAdd_ZA(bits(N) addend, bits(N DIV 2) op1_a, bits(N DIV 2) op1_b,
    bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRType fpcr)
assert N == 32;

bits(N) prod;
boolean isbfloat16 = FALSE;
boolean fpexc = FALSE; // Do not generate floating-point exceptions
fpcr.DN = '1'; // Generate default NaN values
prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
result = FPAdd(addend, prod, fpcr, fpexc);
return result;
shared/functions/float/fpexc/FPExc

definition FPExc
    {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
    FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};

shared/functions/float/fpinfinity/FPInfinity

// FPInfinity()
// ============

bits(N) FPInfinity(bit sign, integer N)

assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
bits(E) exp = Ones(E);
bits(F) frac = Zeros(F);
return sign : exp : frac;

shared/functions/float/fpmatmul/FPMatMulAdd

// FPMatMulAdd()
// =============

// Floating point matrix multiply and add to same precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 2] * op2[2, 2])

bits(N) FPMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, integer esize, FPCRType fpcr)

assert N == esize * 2 * 2;
bits(N) result;
bits(esize) prod0, prod1, sum;
for i = 0 to 1
    for j = 0 to 1
        sum   = Elem[addend, 2*i + j, esize];
        prod0 = FPMul(Elem[op1, 2*i + 0, esize],
            Elem[op2, 2*j + 0, esize], fpcr);
        prod1 = FPMul(Elem[op1, 2*i + 1, esize],
            Elem[op2, 2*j + 1, esize], fpcr);
        sum   = FPAdd(sum, FPAdd(prod0, prod1, fpcr), fpcr);
        Elem[result, 2*i + j, esize] = sum;
return result;

shared/functions/float/fpmax/FPMax

// FPMax()
// ========

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return FPMax(op1, op2, fpcr, altfp);

// FPMax()
// ========

// Compare two inputs and return the larger value after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behaviour.

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr_in, boolean altfp)
assert N IN {16,32,64};
boolean done;
bits(N) result;
FPCRType fpcr = fpcr_in;
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
    // Alternate handling of zeros with differing sign
    return FPZero(sign2, N);
if altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
    // Alternate handling of NaN inputs
    FPProcessException(FPExc_InvalidOp, fpcr);
    result = if type2 == FPType_Zero then FPZero(sign2, N) else op2;
    done = TRUE;
else
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);

if !done then
    FPType fptype;
    bit sign;
    real value;
    if value1 > value2 then
        (fptype,sign,value) = (type1,sign1,value1);
    else
        (fptype,sign,value) = (type2,sign2,value2);
    if fptype == FPType_Infinity then
        result = FPInfinity(sign, N);
    elsif fptype == FPType_Zero then
        sign = sign1 AND sign2;         // Use most positive sign
        result = FPZero(sign, N);
    else
        // The use of FPRound() covers the case where there is a trapped underflow exception
        // for a denormalized number even though the result is exact.
        rounding = FPRoundingMode(fpcr);
        if altfp then    // Denormal output is not flushed to zero
            fpcr.FZ = '0';
            fpcr.FZ16 = '0';
        result = FPRound(value, fpcr, rounding, TRUE, N);
    FPProcessDenorms(type1, type2, N, fpcr);
    return result;

shared/functions/float/fpmaxnormal/FPMaxNormal

// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign, integer N)

assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = Ones(E-1) :'0';
frac = Ones(F);
return sign : exp : frac;
**shared/functions/float/fpmaxnum/FPMaxNum**

// FPMaxNum()
// =========

bits(N) FPMaxNum(bits(N) op1_in, bits(N) op2_in, FPCRType fpcr)

assert N IN {16,32,64};
bits(N) op1 = op1_in;
bits(N) op2 = op2_in;
(type1,-,-) = FPUnpack(op1, fpcr);
(type2,-,-) = FPUnpack(op2, fpcr);

boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

if !(altfp && type1_nan && type2_nan) then
  // Treat a single quiet-NaN as -Infinity.
  if type1 == FPType_QNaN && type2 != FPType_QNaN then
    op1 = FPInfinity('1', N);
  elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
    op2 = FPInfinity('1', N);
  else
    altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMax(op1, op2, fpcr, altfmaxfmin);
  fi

return result;

**shared/functions/float/fpmerge/IsMerging**

// IsMerging()
// ===========

// Returns TRUE if the output elements other than the lowest are taken from
// the destination register.

boolean IsMerging(FPCRType fpcr)

bit nep = if HaveSME() && PSTATE.SM == '1' && !IsFullA64Enabled() then '0' else fpcr.NEP;
return HaveAltFP() && !UsingAArch32() && nep == '1';

**shared/functions/float/fpmin/FPMin**

// FPMin()
// =======

// Compare two operands and return the smaller operand after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative behaviour.

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr_in, boolean altfp)

assert N IN {16,32,64};
boolean done;
bits(N) result;
FPCRType fpcr = fpcr_in;
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
// Alternate handling of zeros with differing sign
return FPZero(sign2, N);

if altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
// Alternate handling of NaN inputs
FPProcessException(FPExc_InvalidOp, fpcr);
result = if type2 == FPType_Zero then FPZero(sign2, N) else op2;
done = TRUE;
else
  (done, result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
endif

if !done then
FPType fptype;
bit sign;
real value;
FPRounding rounding;
if value1 < value2 then
  (fptype, sign, value) = (type1, sign1, value1);
else
  (fptype, sign, value) = (type2, sign2, value2);
if fptype == FPType_Infinity then
  result = FPInfinity(sign, N);
elsif fptype == FPType_Zero then
  sign = sign1 OR sign2; // Use most negative sign
  result = FPZero(sign, N);
else
  // The use of FPRound() covers the case where there is a trapped underflow exception
  // for a denormalized number even though the result is exact.
  rounding = FPRoundingMode(fpcr);
  if altfp then // Denormal output is not flushed to zero
    fpcr.FZ = '0';
    fpcr.FZ16 = '0';
  endif
  result = FPRound(value, fpcr, rounding, TRUE, N);
  FPProcessDenorms(type1, type2, N, fpcr);
endif
return result;

shared/functions/float/fpminnum/FPMinNum

// FPMinNum()
// =========

bits(N) FPMinNum(bits(N) op1_in, bits(N) op2_in, FPCRType fpcr)

assert N IN {16,32,64};
bits(N) op1 = op1_in;
bits(N) op2 = op2_in;
(type1,-,-) = FPUnpack(op1, fpcr);
(type2,-,-) = FPUnpack(op2, fpcr);

boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

if !(altfp && type1_nan && type2_nan) then
  // Treat a single quiet-NaN as +Infinity.
  if type1 == FPType_QNaN && type2 != FPType_QNaN then
    op1 = FPInfinity('0', N);
  elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
    op2 = FPInfinity('0', N);
  else
    altfmaxfmin = FALSE; // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMin(op1, op2, fpcr, altfmaxfmin);
endif
return result;

shared/functions/float/fpmul/FPMul

// FPMul()
// ========
bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)

assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  inf1 = (type1 == FPType_Infinity);
  inf2 = (type2 == FPType_Infinity);
  zero1 = (type1 == FPType_Zero);
  zero2 = (type2 == FPType_Zero);
  if (inf1 && zero2) || (zero1 && inf2) then
    result = FPDefaultNaN(fpcr, N);
    FPProcessException(FPExc_InvalidOp, fpcr);
  elsif inf1 || inf2 then
    result = FPInfinity(sign1 EOR sign2, N);
  elsif zero1 || zero2 then
    result = FPZero(sign1 EOR sign2, N);
  else
    result = FPRound(value1*value2, fpcr, N);
    FPProcessDenorms(type1, type2, N, fpcr);
return result;

shared/functions/float/fpmuladd/FPMulAdd

// FPMulAdd()  
// ===========
bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
  boolean fpexc = TRUE;  // Generate floating-point exceptions
return FPMulAdd(addend, op1, op2, fpcr, fpexc);

// FPMulAdd()
// ===========
// // Calculates addend + op1*op2 with a single rounding. The 'fpcr' argument
// // supplies the FPCR control bits, and 'fpexc' controls the generation of
// // floating-point exceptions.
bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2,
  FPCRType fpcr, boolean fpexc)

assert N IN {16,32,64};
(typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
(type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
(type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
rounding = FPUnpackingMode(fpcr);
inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
(done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);
if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
  if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
    result = FPDefaultNaN(fpcr, N);
  if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
  if !done then
    infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);
    // Determine sign and type product will have if it does not cause an
    // Invalid Operation.
    signP = sign1 EOR sign2;
    infP  = inf1 || inf2;
    zeroP = zero1 || zero2;
    // Non SNaN-generated Invalid Operation cases are multiplies of zero
    // by infinity and additions of opposite-signed infinities.
    invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);
    if invalidop then
      result = FPDefaultNaN(fpcr, N);
    if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    // Other cases involving infinities produce an infinity of the same sign.
    elsif (infA && signA == '0') || (infP && signP == '0') then
      result = FPInfinity('0', N);
    elsif (infA && signA == '1') || (infP && signP == '1') then
      result = FPInfinity('1', N);
    // Cases where the result is exactly zero and its sign is not determined by the
    // rounding mode are additions of same-signed zeros.
    elsif zeroA && zeroP && signA == signP then
      result = FPZero(signA, N);
    // Otherwise calculate numerical result and round it.
    else
      result_value = valueA + (value1 * value2);
      if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
        result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
        result = FPZero(result_sign, N);
      else
        result = FPRound(result_value, fpcr, rounding, fpexc, N);
      if !invalidop && fpexc then
        FPProcessDenorms3(typeA, type1, type2, N, fpcr);
return result;

shared/functions/float/fpmuladd/FPMulAdd_ZA

// FPMulAdd_ZA()
// =============
// Calculates addend + op1*op2 with a single rounding for SME ZA-targeting
// instructions.
bits(N) FPMulAdd_ZA(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr_in)
  FPCRType fpcr = fpcr_in;
  boolean fpexc = FALSE; // Do not generate floating-point exceptions
  fpcr.DN = '1'; // Generate default NaN values
  return FPMulAdd(addend, op1, op2, fpcr, fpexc);

shared/functions/float/fpmuladdh/FPMulAddH

// FPMulAddH()
// ===========
// Calculates addend + op1*op2.
bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr)
boolean fpexc = TRUE;       // Generate floating-point exceptions
return FPMulAddH(addend, op1, op2, fpcr, fpexc);

// FPMulAddH()
// ===========
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr, boolean fpexc)

assert N == 32;
rounding = FPRoundingMode(fpcr);
(typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
(type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
(type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);

(done, result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);
if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
  if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
    result = FPDefaultNaN(fpcr, N);
  if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
if !done then
  infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);
  // Determine sign and type product will have if it does not cause an
  // Invalid Operation.
  signP = sign1 XOR sign2;
  infP = inf1 || inf2;
  zeroP = zero1 || zero2;

  // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
  // additions of opposite-signed infinities.
  invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);
  if invalidop then
    result = FPDefaultNaN(fpcr, N);
  if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
  elsif (infA && signA == '0') || (infP && signP == '0') then
    result = FPInfinity('0', N);
  elsif (infA && signA == '1') || (infP && signP == '1') then
    result = FPInfinity('1', N);
  // Cases where the result is exactly zero and its sign is not determined by the
  // rounding mode are additions of same-signed zeros.
  elseif zeroA && zeroP && signA == signP then
    result = FPZero(signA, N);
  // Otherwise calculate numerical result and round it.
  else
    result_value = valueA + (value1 * value2);
    if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
      result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
      result = FPZero(result_sign, N);
    else
      result = FPRound(result_value, fpcr, rounding, fpexc, N);
  if !invalidop && fpexc then
    FPProcessDenorm(typeA, N, fpcr);
shared/functions/float/fpmuladdh/FPProcessNaNs3H

// FPProcessNaNs3H()
// ================

(boolean, bits(N)) FPProcessNaNs3H(FPType type1, FPType type2, FPType type3,
bits(N) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
FPCRType fpocr, boolean fpexc)

assert N IN {32,64};
bits(N) result;
FPType type_nan;
// When TRUE, use alternative NaN propagation rules.
boolean altfp = HaveAltFP() && !UsingAArch32() && fpocr.AH == '1';
boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
boolean op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};
if altfp then
  if (type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN) then
    type_nan = FPType_SNaN;
  else
    type_nan = FPType_QNaN;
else
  type_nan = FPType_QNaN;

boolean done;
if altfp && op1_nan && op2_nan && op3_nan then // <> register NaN selected
  done = TRUE; result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpocr, fpexc), N);
elsif altfp && op2_nan && (op1_nan || op3_nan) then // <> register NaN selected
  done = TRUE; result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpocr, fpexc), N);
elsif altfp && op3_nan && op1_nan then // <> register NaN selected
  done = TRUE; result = FPConvertNaN(FPProcessNaN(type_nan, op3, fpocr, fpexc), N);
elsif type1 == FPType_SNaN then
  done = TRUE; result = FPProcessNaN(type1, op1, fpocr, fpexc);
elsif type2 == FPType_SNaN then
  done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpocr, fpexc), N);
elsif type3 == FPType_SNaN then
  done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpocr, fpexc), N);
elsif type1 == FPType_QNaN then
  done = TRUE; result = FPProcessNaN(type1, op1, fpocr, fpexc);
elsif type2 == FPType_QNaN then
  done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpocr, fpexc), N);
elsif type3 == FPType_QNaN then
  done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpocr, fpexc), N);
else
  done = FALSE; result = Zeros(N); // 'Don't care' result
return (done, result);

shared/functions/float/fpmulx/FPMulX

// FPMulX()
// =========

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRType fpocr)

assert N IN {16,32,64};
bits(N) result;
boolean done;
(type1,sign1,value1) = FPUnpack(op1, fpocr);
(type2,sign2,value2) = FPUnpack(op2, fpocr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpocr);
if !done then
  inf1 = (type1 == FPType_Infinity);
  inf2 = (type2 == FPType_Infinity);
  zero1 = (type1 == FPType_Zero);
  zero2 = (type2 == FPType_Zero);

  if (inf1 && zero2) || (zero1 && inf2) then
    result = FPTwo(sign1 EOR sign2, N);
  elsif inf1 || inf2 then
    result = FPInfinity(sign1 EOR sign2, N);
  elsif zero1 || zero2 then
    result = FPZero(sign1 EOR sign2, N);
  else
    result = FPRound(value1*value2, fpcr, N);
  
  FPProcessDenorms(type1, type2, N, fpcr);

return result;

shared/functions/float/fpneg/FPNeg

// FPNeg()
// =======

bits(N) FPNeg(bits(N) op)

assert N IN {16,32,64};
if !UsingAArch32() && HaveAltFP() then
  FPCRType fpcr = FPCR[];
  if fpcr.AH == '1' then
    (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
    if fptype IN {FPType_SNaN, FPType_QNaN} then
      return op; // When fpcr.AH=1, sign of NaN has no consequence
    end

    return NOT(op<N-1>) : op<N-2:0>;

shared/functions/float/fponepointfive/FPOnePointFive

// FPOnePointFive()
// ================

bits(N) FPOnePointFive(bit sign, integer N)

assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp  = '0':Ones(E-1);
frac = '1':Zeros(F-1);
result = sign : exp : frac;
return result;

shared/functions/float/fpprocessdenorms/FPProcessDenorm

// FPProcessDenorm()
// ================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorm(FPType fptype, integer N, FPCRType fpcr)
  boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
if altfp && N != 16 && fptype == FPTYPE_Denormal then
  FPProcessException(FPExc_InputDenorm, fpcr);

shared/functions/float/fpprocessdenorms/FPProcessDenoms

// FPProcessDenoms()
// ===============
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenoms(FPTYPE type1, FPTYPE type2, integer N, FPCRType fpcr)
  boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
  if altfp && N != 16 && (type1 == FPTYPE_Denormal || type2 == FPTYPE_Denormal) then
    FPProcessException(FPExc_InputDenorm, fpcr);

shared/functions/float/fpprocessdenorms/FPProcessDenoms3

// FPProcessDenoms3()
// ===============
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenoms3(FPTYPE type1, FPTYPE type2, FPTYPE type3, integer N, FPCRType fpcr)
  boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
  if altfp && N != 16 && (type1 == FPTYPE_Denormal || type2 == FPTYPE_Denormal ||
                        type3 == FPTYPE_Denormal) then
    FPProcessException(FPExc_InputDenorm, fpcr);

shared/functions/float/fpprocessdenorms/FPProcessDenoms4

// FPProcessDenoms4()
// ===============
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenoms4(FPTYPE type1, FPTYPE type2, FPTYPE type3, FPTYPE type4, integer N, FPCRType fpcr)
  boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
  if altfp && N != 16 && (type1 == FPTYPE_Denormal || type2 == FPTYPE_Denormal ||
                          type3 == FPTYPE_Denormal || type4 == FPTYPE_Denormal) then
    FPProcessException(FPExc_InputDenorm, fpcr);

shared/functions/float/fpprocessexception/FPProcessException

// FPProcessException()
// ============
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRType fpcr)
  integer cumul;
  // Determine the cumulative exception bit number
  case exception of
    when FPExc_InvalidOp     cumul = 0;
    when FPExc_DivideByZero  cumul = 1;
    when FPExc_Overflow      cumul = 2;
    when FPExc_Underflow     cumul = 3;
    when FPExc_Inexact       cumul = 4;
    when FPExc_InputDenorm   cumul = 7;
  enable = cumul + 8;
if fpcr<enable> == '1' && (!HaveSME() || PSTATE.SM == '0' || IsFullA64Enabled()) then
// Trapping of the exception enabled.
// It is IMPLEMENTATION DEFINED whether the enable bit may be set at all,
// and if so then how exceptions and in what order that they may be
// accumulated before calling FPTrappedException().
bits(8) accumulated_exceptions = GetAccumulatedFPExceptions();
accumulated_exceptions<cumul> = '1';
if boolean IMPLEMENTATION_DEFINED "Support trapping of floating-point exceptions" then
if UsingAArch32() then
  AArch32.FPTrappedException(accumulated_exceptions);
else
  is_ase = IsASEInstruction();
  AArch64.FPTrappedException(is_ase, accumulated_exceptions);
else
  // The exceptions generated by this instruction are accumulated by the PE and
  // FPTrappedException is called later during its execution, before the next
  // instruction is executed. This field is cleared at the start of each FP instruction.
  SetAccumulatedFPExceptions(accumulated_exceptions);
elseif UsingAArch32() then
  // Set the cumulative exception bit
  FPSCR<cumul> = '1';
else
  // Set the cumulative exception bit
  FPSR<cumul> = '1';
return;

shared/functions/float/fpprocessnan/FPProcessNaN

// FPProcessNaN()
// ===============

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRType fpcr)
  boolean fpexc = TRUE;   // Generate floating-point exceptions
  return FPProcessNaN(fptype, op, fpcr, fpexc);

// FPProcessNaN()
// ===============

// Handle NaN input operands, returning the operand or default NaN value
// if fpcr.DN is selected. The 'fpcr' argument supplies the FPCR control bits.
// The 'fpexc' argument controls the generation of exceptions, regardless of
// whether 'fptype' is a signalling NaN or a quiet NaN.

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRType fpcr, boolean fpexc)

assert N IN {16,32,64};
assert fptype IN {FPType_QNaN, FPType_SNaN};
integer topfrac;

case N of
  when 16 topfrac =  9;
  when 32 topfrac = 22;
  when 64 topfrac = 51;
result = op;
if fptype == FPType_SNaN then
  result<topfrac> = '1';
if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
if fpcr.DN == '1' then  // DefaultNaN requested
  result = FPDefaultNaN(fpcr, N);
return result;
shared/functions/float/fpprocessnans/FPProcessNaNs

// FPProcessNaNs()
// ===============

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1,
bits(N) op2, FPCRType fpcr)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);

// FPProcessNaNs()
// ===============

// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'altfmaxfmin' controls
// alternative floating-point behaviour for FMAX, FMIN and variants. 'fpexc'
// controls the generation of floating-point exceptions. Status information
// is updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1, bits(N) op2,
FPCRType fpcr, boolean fpexc)

assert N IN {16,32,64};

boolean done;
bits(N) result;

boolean altfp
    = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};

boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};

boolean any_snan = type1 == FPType_SNaN || type2 == FPType_SNaN;

FPType  type_nan = if any_snan then FPType_SNaN else FPType_QNaN;

if altfp && op1_nan && op2_nan then
    // <n> register NaN selected
    done = TRUE;  result = FPProcessNaN(type_nan, op1, fpcr, fpexc);
elsif type1 == FPType_SNaN then
    done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
elsif type2 == FPType_SNaN then
    done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
elsif type1 == FPType_QNaN then
    done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
elsif type2 == FPType_QNaN then
    done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
else
    done = FALSE;  result = Zeros(N);  // 'Don't care' result

return (done, result);

shared/functions/float/fpprocessnans3/FPProcessNaNs3

// FPProcessNaNs3()
// ===============

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
bits(N) op1, bits(N) op2, bits(N) op3, FPCRType fpcr)

boolean fpexc = TRUE; // Generate floating-point exceptions
return FPProcessNaNs3(type1, type2, type3, op1, op2, op3, fpcr, fpexc);

// FPProcessNaNs3()
// ===============

// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3, bits(N) op1, bits(N) op2, bits(N) op3, 
    FPCRType fpcr, boolean fpexc)

assert N IN {16,32,64};

bits(N) result;

boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};

boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};

boolean op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};

boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

FPType type_nan;

if altfp then
  if type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN then
    type_nan = FPType_SNaN;
  else
    type_nan = FPType_QNaN;
else
  type_nan = FPType_QNaN;

boolean done;
if altfp & op1_nan & op2_nan & op3_nan then
  // <n> register NaN selected
  done = TRUE; result = FPProcessNaN(type_nan, op2, fpcr, fpexc);
elseif altfp & op2_nan & (op1_nan || op3_nan) then
  // <n> register NaN selected
  done = TRUE; result = FPProcessNaN(type_nan, op2, fpcr, fpexc);
elseif altfp & op3_nan & op1_nan then
  // <m> register NaN selected
  done = TRUE; result = FPProcessNaN(type_nan, op3, fpcr, fpexc);
elseif type1 == FPType_SNaN then
  done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
elseif type2 == FPType_SNaN then
  done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
elseif type3 == FPType_SNaN then
  done = TRUE; result = FPProcessNaN(type3, op3, fpcr, fpexc);
elseif type1 == FPType_QNaN then
  done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
elseif type2 == FPType_QNaN then
  done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
elseif type3 == FPType_QNaN then
  done = TRUE; result = FPProcessNaN(type3, op3, fpcr, fpexc);
else
  done = FALSE; result = Zeros(N); // 'Don't care' result

return (done, result);

shared/functions/float/fpprocessnans4/FPProcessNaNs4

// FPProcessNaNs4()
// ================
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
// The 'fpcr' argument supplies FPCR control bits.
// Status information is updated directly in the FPSR where appropriate.
// The 'fpexc' controls the generation of floating-point exceptions.

(boolean, bits(N)) FPProcessNaNs4(FPType type1, FPType type2, FPType type3, FPType type4, 
    bits(N DIV 2) op1, bits(N DIV 2) op2, bits(N DIV 2) op3, bits(N DIV 2) op4, 
    FPCRType fpcr, boolean fpexc)
assert N == 32;

bits(N) result;
boolean done;
// The FPCR.AH control does not affect these checks
if type1 == FPType_SNaN then
done = TRUE;  result = FPConvertNaN(FPProcessNaN(type1, op1, fpcr, fpexc), N);
elsif type2 == FPType_SNaN then
done = TRUE;  result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), N);
elsif type3 == FPType_SNaN then
done = TRUE;  result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), N);
elsif type4 == FPType_SNaN then
done = TRUE;  result = FPConvertNaN(FPProcessNaN(type4, op4, fpcr, fpexc), N);
elsif type1 == FPType_QNaN then
done = TRUE;  result = FPConvertNaN(FPProcessNaN(type1, op1, fpcr, fpexc), N);
elsif type2 == FPType_QNaN then
done = TRUE;  result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), N);
elsif type3 == FPType_QNaN then
done = TRUE;  result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), N);
elsif type4 == FPType_QNaN then
done = TRUE;  result = FPConvertNaN(FPProcessNaN(type4, op4, fpcr, fpexc), N);
else
  done = FALSE;  result = Zeros(N);  // 'Don't care' result

return (done, result);

shared/functions/float/fprecipestimate/FPRrecipEstimate

  // FPRrecipEstimate()
  // =============

bits(N) FPRrecipEstimate(bits(N) operand, FPCRType fpcr_in)

assert N IN {16,32,64};
FPCRType fpcr = fpcr_in;
bits(N) result;
boolean overflow_to_inf;
// When using alternative floating-point behaviour, do not generate
// floating-point exceptions, flush denormal input and output to zero,
// and use RNE rounding mode.
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
boolean fpexc = !altfp;
if altfp then fpcr.<FIZ,FZ> = '11';
if altfp then fpcr.RMode = '00';

(fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);
FPRounding rounding = FPRoundingMode(fpcr);
if fptype == FPType_SNaN || fptype == FPType_QNaN then
  result = FPProcessNaN(fptype, operand, fpcr, fpexc);
elsif fptype == FPType_Infinity then
  result = FPZero(sign, N);
elsif fptype == FPType_Zero then
  result = FPInfinity(sign, N);
if fpexc then FPProcessException(FPExc_DivideByZero, fpcr);
elsif (N == 16 && Abs(value) < 2.0^-16) ||
  (N == 32 && Abs(value) < 2.0^-128) ||
  (N == 64 && Abs(value) < 2.0^-1024)
) then
  case rounding of
    when FPRounding_TIEEVEN
      overflow_to_inf = TRUE;
    when FPRounding_POSINF
      overflow_to_inf = (sign == '0');
    when FPRounding_NEGINF

ARM DDI 0487I.a Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. J1-11465
ID081822
Non-Confidential
overflow_to_inf = (sign == '1');
when FPRounding_ZERO
  overflow_to_inf = FALSE;
result = if overflow_to_inf then FPInfinity(sign, N) else FPMaxNormal(sign, N);
if fpexc then
  FPProcessException(FPExc_Overflow, fpcr);
  FPProcessException(FPExc_Inexact, fpcr);
elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
  && (
    (N == 16 && Abs(value) >= 2.0^14) ||
    (N == 32 && Abs(value) >= 2.0^126) ||
    (N == 64 && Abs(value) >= 2.0^1022)
  ) then
  // Result flushed to zero of correct sign
  result = FPZero(sign, N);

  // Flush-to-zero never generates a trapped exception.
  if UsingAArch32() then
    FPSCR.UFC = '1';
  else
    if fpexc then FPSR.UFC = '1';
  else
    // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
    // calculate result exponent. Scaled value has copied sign bit,
    // exponent = 1022 = double-precision biased version of -1,
    // fraction = original fraction
    bits(52) fraction;
    integer exp;
    case N of
      when 16
        fraction = operand<9:0> : Zeros(42);
        exp = UInt(operand<14:10>);
      when 32
        fraction = operand<22:0> : Zeros(29);
        exp = UInt(operand<30:23>);
      when 64
        fraction = operand<51:0>;
        exp = UInt(operand<62:52>);
    if exp == 0 then
      if fraction<51> == '0' then
        exp = -1;
        fraction = fraction<49:0>:'00';
      else
        fraction = fraction<50:0>:'0';
    integer scaled;
    boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;
    if !increasedprecision then
      scaled = UInt('1':fraction<51:44>);
    else
      scaled = UInt('1':fraction<51:41>);
    integer result_exp;
    case N of
      when 16 result_exp =  29 - exp; // In range 29-30 = -1 to 29+1 = 30
      when 32 result_exp =  253 - exp; // In range 253-254 = -1 to 253+1 = 254
      when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046
    // Scaled is in range 256 .. 511 or 2048 .. 4095 range representing a
    // fixed-point number in range [0.5 .. 1.0].
    estimate = RecipEstimate(scaled, increasedprecision);
    // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
    // fixed-point result in the range [1.0 .. 2.0].
    // Convert to scaled floating point result with copied sign bit,
    // high-order bits from estimate, and exponent calculated above.
if !increasedprecision then
  fraction = estimate<7:0> : Zeros(44);
else
  fraction = estimate<11:0> : Zeros(40);

if result_exp == 0 then
  fraction = '1' : fraction<51:1>;
elsif result_exp == -1 then
  fraction = '01' : fraction<51:2>;
  result_exp = 0;
else
  case N of
    when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
    when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
    when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;
  end;
return result;

shared/functions/float/fprecipestimate/RecipEstimate

// RecipEstimate()
// ===============
// Compute estimate of reciprocal of 9-bit fixed-point number.
//
// a is in range 256 .. 511 or 2048 .. 4096 representing a number in
// the range 0.5 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191 representing a
// number in the range 1.0 to 511/256 or 1.00 to 8191/4096.

integer RecipEstimate(integer a_in, boolean increasedprecision)
integer a = a_in;
integer r;
if !increasedprecision then
  assert 256 <= a && a < 512;
  r = (a+1) DIV 2;     // Round to nearest
else
  assert 2048 <= a && a < 4096;
  real real_val = Real(2^25)/Real(a);
  r = RoundDown(real_val);  // Round to nearest
  assert 4096 <= r && r < 8192;
return r;

shared/functions/float/fprecpx/FPRecpX

// FPRecpX()
// =========

bits(N) FPRecpX(bits(N) op, FPCRType fpcr_in)

assert N IN {16,32,64};
FPCRType fpcr = fpcr_in;
integer esize;
when 16 esize = 5;
when 32 esize = 8;
when 64 esize = 11;

bits(N)           result;
bits(esize)       exp;
bits(esize)       max_exp;
bits(N-(esize+1)) frac = Zeros(N-(esize+1));

boolean altfp = HaveAltFP() && fpcr.AH == '1';
boolean fpexc = !altfp;                 // Generate no floating-point exceptions
if altfp then fpcr.<FIZ,FZ> = '11';     // Flush denormal input and output to zero
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

case N of
  when 16 exp = op<10+esize-1:10>;
  when 32 exp = op<23+esize-1:23>;
  when 64 exp = op<52+esize-1:52>;
max_exp = Ones(esize) - 1;
if fptype == FPType_SNaN || fptype == FPType_QNaN then
  result = FPProcessNaN(fptype, op, fpcr, fpexc);
else
  if IsZero(exp) then                 // Zero and denormals
    result = sign:max_exp:frac;
  else                                // Infinities and normals
    result = sign:NOT(exp):frac;

return result;

shared/functions/float/fpround/FPRound

// FPRound()
// =========
// Generic conversion from precise, unbounded real data type to IEEE format.

bits(N) FPRound(real op, FPCRType fpcr, integer N)
  return FPRound(op, fpcr, FPRoundingMode(fpcr), N);

// FPRound()
// =========
// For directed FP conversion, includes an explicit 'rounding' argument.

bits(N) FPRound(real op, FPCRType fpcr_in, FPRounding rounding, integer N)
  boolean fpexc = TRUE;    // Generate floating-point exceptions
  return FPRound(op, fpcr_in, rounding, fpexc, N);

// FPRound()
// =========
// For AltFP, includes an explicit FPEXC argument to disable exception
// generation and switches off Arm alternate half-precision mode.

bits(N) FPRound(real op, FPCRType fpcr_in, FPRounding rounding, boolean fpexc, integer N)
  FPCRType fpcr = fpcr_in;
  fpcr.AHP = '0';
  boolean isbfloat16 = FALSE;
  return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);

shared/functions/float/fpround/FPRoundBase

// FPRoundBase()
// =============
// For BFloat16, includes an explicit 'isbfloat16' argument.
bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding, boolean isbfloat16, integer N)
   boolean fpexc = TRUE;    // Generate floating-point exceptions
   return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);

// FPRoundBase()
// =============
// Convert a real number 'op' into an N-bit floating-point value using the
// supplied rounding mode 'rounding'.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding,
   boolean isbfloat16, boolean fpexc, integer N)

assert N IN {16,32,64};
assert op != 0.0;
assert rounding != FPRounding_TIEAWAY;
bits(N) result;

// Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
integer minimum_exp;
integer F;
integer E;
if N == 16 then
   minimum_exp = -14;  E = 5;  F = 10;
elsif N == 32 && isbfloat16 then
   minimum_exp = -126;  E = 8;  F = 7;
elsif N == 32 then
   minimum_exp = -126;  E = 8;  F = 23;
else  // N == 64
   minimum_exp = -1022;  E = 11;  F = 52;

// Split value into sign, unrounded mantissa and exponent.
bit sign;
real mantissa;
if op < 0.0 then
   sign = '1';  mantissa = -op;
else
   sign = '0';  mantissa = op;
exponent = 0;
while mantissa < 1.0 do
   mantissa = mantissa * 2.0;  exponent = exponent - 1;
while mantissa >= 2.0 do
   mantissa = mantissa / 2.0;  exponent = exponent + 1;
// When TRUE, detection of underflow occurs after rounding and the test for a
// denormalized number for single and double precision values occurs after rounding.
altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
// Deal with flush-to-zero before rounding if FPCR.AH != '1'.
if (!altfp && ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) &&
exponent < minimum_exp) then
   // Flush-to-zero never generates a trapped exception.
   if UsingAArch32() then
      FPSCR.UFC = '1';
   else
      if fpexc then FPSR.UFC = '1';
      return FPZero(sign, N);
biased_exp_unconstrained = (exponent - minimum_exp) + 1;
int_mant_unconstrained = RoundDown(mantissa * 2.0^F);
error_unconstrained = mantissa * 2.0^F - Real(int_mant_unconstrained);
// Start creating the exponent value for the result. Start by biasing the actual exponent
// so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
based_exp = Max((exponent - minimum_exp) + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
error = mantissa * 2.0^F - Real(int_mant);

// Underflow occurs if exponent is too small before rounding, and result is inexact or
// the Underflow exception is trapped. This applies before rounding if FPCR.AH != '1'.
boolean trapped_UF = fpcr.UFE == '1' && (!InStreamingMode() || IsFullA64Enabled());
if !altfp && biased_exp == 0 && (error != 0.0 || trapped_UF) then
  if fpexc then FPProcessException(FPExc_Underflow, fpcr);

// Round result according to rounding mode.
boolean round_up_unconstrained;
boolean round_up;
boolean overflow_to_inf;
if altfp then
  case rounding of
  when FPRounding_TIEEVEN
    round_up_unconstrained = (error_unconstrained > 0.5 ||
      (error_unconstrained == 0.5 && int_mant_unconstrained<0> == '1'));
    round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
    overflow_to_inf = TRUE;
  when FPRounding_POSINF
    round_up_unconstrained = (error_unconstrained != 0.0 && sign == '0');
    round_up = (error != 0.0 && sign == '0');
    overflow_to_inf = (sign == '0');
  when FPRounding_NEGINF
    round_up_unconstrained = (error_unconstrained != 0.0 && sign == '1');
    round_up = (error != 0.0 && sign == '1');
    overflow_to_inf = (sign == '1');
  when FPRounding_ZERO, FPRounding_ODD
    round_up_unconstrained = FALSE;
    round_up = FALSE;
    overflow_to_inf = FALSE;
  if round_up_unconstrained then
    int_mant_unconstrained = int_mant_unconstrained + 1;
  if int_mant_unconstrained == 2^(F+1) then    // Rounded up to next exponent
    biased_exp_unconstrained = biased_exp_unconstrained + 1;
    int_mant_unconstrained = int_mant_unconstrained DIV 2;

  // Deal with flush-to-zero and underflow after rounding if FPCR.AH == '1'.
  if biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 then
    // the result of unconstrained rounding is less than the minimum normalized number
  if (fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16) then   // Flush-to-zero
    if fpexc then
      FPSR.UFC = '1';
      FPProcessException(FPExc_Inexact, fpcr);
      return FPZero(sign, N);
    elseif error != 0.0 || trapped_UF then
      if fpexc then FPProcessException(FPExc_Underflow, fpcr);

  else // altfp == FALSE
    case rounding of
    when FPRounding_TIEEVEN
      round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
      overflow_to_inf = TRUE;
    when FPRounding_POSINF
      round_up = (error != 0.0 && sign == '0');
      overflow_to_inf = (sign == '0');
    when FPRounding_NEGINF
      round_up = (error != 0.0 && sign == '1');
      overflow_to_inf = (sign == '1');
    when FPRounding_ZERO, FPRounding_ODD
      round_up = FALSE;
      overflow_to_inf = FALSE;
if round_up then
  int_mant = int_mant + 1;
  if int_mant == 2^F then  // Rounded up from denormalized to normalized
    biased_exp = 1;
  if int_mant == 2^(F+1) then  // Rounded up to next exponent
    biased_exp = biased_exp + 1;
  int_mant = int_mant DIV 2;

  // Handle rounding to odd
  if error != 0.0 && rounding == FPRounding_ODD then
    int_mant<0> = '1';

  // Deal with overflow and generate result.
  if N != 16 || fpcr.AHP == '0' then  // Single, double or IEEE half precision
    if biased_exp >= 2^E - 1 then
      result = if overflow_to_inf then FPInfinity(sign, N) else FPMaxNormal(sign, N);
      error = 1.0;  // Ensure that an Inexact exception occurs
    else
      result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));
    else
      result = sign : Ones(N-1);  // Alternative half precision
      if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
      error = 0.0;  // Ensure that an Inexact exception does not occur
    else
      result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));

  // Deal with Inexact exception.
  if error != 0.0 then
    if fpexc then FPProcessException(FPExc_Inexact, fpcr);

return result;

shared/functions/float/fpround/FPRoundCV

// FPRoundCV()
// ===========
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZI16 and observes AHP.

bits(N) FPRoundCV(real op, FPCRType fpcr_in, FPRounding rounding, integer N)
  FPCRType fpcr = fpcr_in;
  fpcr.FZI16 = '0';
  boolean fpexc = TRUE;  // Generate floating-point exceptions
  boolean isbfloat16 = FALSE;
  return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);

shared/functions/float/fprounding/FPRounding

enumeration FPRounding  {FPRounding_TIEEVEN, FPRounding_POSINF,
                        FPRounding_NEGINF,  FPRounding_ZERO,
                        FPRounding_TIEAWAY, FPRounding_ODD};

shared/functions/float/fproundingmode/FPRoundingMode

// FPRoundingMode()
// ================
// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCRType fpcr)
  return FPDecodeRounding(fpcr.RMode);
shared/functions/float/fproundint/FPRoundInt

// FPRoundInt()
// ============

// Round op to nearest integral floating point value using rounding mode in FPCR/FPSCR.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to op.

bits(N) FPRoundInt(bits(N) op, FPCRType fpcr, FPRounding rounding, boolean exact)

assert rounding != FPRounding_ODD;
assert N IN {16,32,64};

// When alternative floating-point support is TRUE, do not generate
// Input Denormal floating-point exceptions.
altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
fpexc = !altfp;

// Unpack using FPCR to determine if subnormals are flushed-to-zero.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

result = FPZero(sign, N);

// Extract integer component.
int_result = RoundDown(value);
error = value - Real(int_result);

// Determine whether supplied rounding mode requires an increment.
bool round_up;
case rounding of
  when FPRounding_TIEEVEN
    round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
  when FPRounding_POSINF
    round_up = (error != 0.0);
  when FPRounding_NEGINF
    round_up = FALSE;
  when FPRounding.ZERO
    round_up = (error != 0.0 && int_result < 0);
  when FPRounding.TIEAWAY
    round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

if round_up then int_result = int_result + 1;

// Convert integer value into an equivalent real value.
real_result = Real(int_result);

// Re-encode as a floating-point value, result is always exact.
if real_result == 0.0 then
  result = FPZero(sign, N);
else
  result = FPRound(real_result, fpcr, FPRounding.ZERO, N);

// Generate inexact exceptions.
if error != 0.0 && exact then
  FPProcessException(FPExc_Inexact, fpcr);

return result;
shared/functions/float/fproundintn/FPRoundIntN

// FPRoundIntN()
// =============

bits(N) FPRoundIntN(bits(N) op, FPCRType fpcr, FPRounding rounding, integer intsize)
assert rounding != FPRounding_ODD;
assert N IN {32,64};
assert intsize IN {32, 64};
integer exp;
bits(N) result;
boolean round_up;
constant integer E = (if N == 32 then 8 else 11);
constant integer F = N - (E + 1);

// When alternative floating-point support is TRUE, do not generate
// Input Denormal floating-point exceptions.
altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
fpexc = !altfp;

// Unpack using FPCR to determine if subnormals are flushed-to-zero.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc);
if fptype IN {FPType_SNaN, FPType_QNaN, FPType_Infinity} then
  if N == 32 then
    exp = 126 + intsize;
    result = '1':exp<(E-1):0>:Zeros(F);
  else
    exp = 1022+intsize;
    result = '1':exp<(E-1):0>:Zeros(F);
    FProcessException(FPExc_InvalidOp, fpcr);
  elsif fptype == FPType_Zero then
    result = FPZero(sign, N);
else
  // Extract integer component.
  int_result = RoundDown(value);
  error = value - Real(int_result);

  // Determine whether supplied rounding mode requires an increment.
  case rounding of
    when FPRounding_TIEEVEN
      round_up = error > 0.5 || (error == 0.5 && int_result<0> == '1');
    when FPRounding_POSINF
      round_up = error != 0.0;
    when FPRounding_NEGINF
      round_up = FALSE;
    when FPRounding_ZERO
      round_up = error != 0.0 && int_result < 0;
    when FPRounding_TIEAWAY
      round_up = error > 0.5 || (error == 0.5 && int_result >= 0);
  if round_up then int_result = int_result + 1;
  error = value - Real(int_result);
  overflow = int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1);
  if overflow then
    if N == 32 then
      exp = 126 + intsize;
      result = '1':exp<(E-1):0>:Zeros(F);
    else
      exp = 1022 + intsize;
      result = '1':exp<(E-1):0>:Zeros(F);
      FProcessException(FPExc_InvalidOp, fpcr);
    // This case shouldn't set Inexact.
    error = 0.0;
  else
    // Convert integer value into an equivalent real value.
    real_result = Real(int_result);

ARM DDI 0487I.a
Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved.
ID081822
Non-Confidential
// Re-encode as a floating-point value, result is always exact.
if real_result == 0.0 then
  result = FPZero(sign, N);
else
  result = FPRound(real_result, fpcr, FPRounding_ZERO, N);

// Generate inexact exceptions.
if error != 0.0 then
  FPProcessException(FPExc_Inexact, fpcr);
return result;

shared/functions/float/fprsqrtestimate/FPRSqrtEstimate

// FPRSqrtEstimate()
// ===============

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRType fpcr_in)

assert N IN {16,32,64};
FPCRType fpcr = fpcr_in;

// When using alternative floating-point behaviour, do not generate
// floating-point exceptions and flush denormal input to zero.
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
boolean fpexc = !altfp;
if altfp then fpcr.<FIZ,FZ> = '11';
(fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

bits(N) result;
if fptype == FPType_SNaN || fptype == FPType_QNaN then
  result = FPProcessNaN(fptype, operand, fpcr, fpexc);
elsif fptype == FPType_Zero then
  result = FPInfinity(sign, N);
elsif sign == '1' then
  result = FPDefaultNaN(fpcr, N);
elsif fptype == FPType_Infinity then
  result = FPZero('0', N);
else
  // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
  // evenness or oddness of the exponent unchanged, and calculate result exponent.
  // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
  // biased version of -1 or -2, fraction = original fraction extended with zeros.
  bits(52) fraction;
  integer exp;
  case N of
    when 16
      fraction = operand<9:0> : Zeros(42);
      exp = UInt(operand<14:10>);
    when 32
      fraction = operand<22:0> : Zeros(29);
      exp = UInt(operand<30:23>);
    when 64
      fraction = operand<51:0>;
      exp = UInt(operand<62:52>);
  endcase
  if exp == 0 then
    while fraction<51> == '0' do
      fraction = fraction<50:0> : '0';
      exp = exp - 1;
    endwhile
  endif
  result = FPZero('0', N);
endcase
}
integer scaled;
boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;

if !increasedprecision then
  if exp<0> == '0' then
    scaled = UInt('1':fraction<51:44>);
  else
    scaled = UInt('01':fraction<51:45>);
else
  if exp<0> == '0' then
    scaled = UInt('1':fraction<51:41>);
  else
    scaled = UInt('01':fraction<51:42>);

integer result_exp;

  case N of
    when 16 result_exp = (  44 - exp) DIV 2;
    when 32 result_exp = ( 380 - exp) DIV 2;
    when 64 result_exp = (3068 - exp) DIV 2;
  endcase

estimate = RecipSqrtEstimate(scaled, increasedprecision);

  // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
  // fixed-point result in the range [1.0 .. 2.0].
  // Convert to scaled floating point result with copied sign bit and high-order
  // fraction bits, and exponent calculated above.

  case N of
    when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>:Zeros(2);
    when 32
      if !increasedprecision then
        result = '0' : result_exp<N-25:0> : estimate<7:0>:Zeros(15);
      else
        result = '0' : result_exp<N-25:0> : estimate<11:0>:Zeros(11);
    when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Zeros(44);
  endcase

return result;

shared/functions/float/fprsqrtestimate/RecipSqrtEstimate

// RecipSqrtEstimate()
// ===================
// Compute estimate of reciprocal square root of 9-bit fixed-point number.
// a_in is in range 128 .. 511 or 1024 .. 4095, with increased precision,
// representing a number in the range 0.25 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191, with increased precision,
// representing a number in the range 1.0 to 511/256 or 8191/4096.

integer RecipSqrtEstimate(integer a_in, boolean increasedprecision)

  integer a = a_in;
  integer r;

  if !increasedprecision then
    assert 128 <= a && a < 512;
    if a < 256 then // 0.25 .. 0.5
      a = a+1;
      a = a << 1; // in units of 1/512 rounded to nearest
    else
      a = a >> 1; // Discard bottom bit
      a = a+1; // in units of 1/256 rounded to nearest
    integer b = 512;
    while a*(b+1)*(b+1) < 2^28 do
      b = b+1;
      // b = largest b such that b < 2^14 / sqrt(a)
    r = (b+1) DIV 2; // Round to nearest
assert 256 <= r && r < 512;
else
  assert 1024 <= a && a < 4096;
  real real_val;
  real error;
  integer int_val;

  if a < 2048 then // 0.25... 0.5
    a = a*2 + 1; // Take 10 bits of fraction and force a 1 at the bottom
    real_val = Real(a)/2.0;
  else
    // 0.5...1.0
    a = (a >> 1) << 1; // Discard bottom bit
    a = a+1;
    real_val = Real(a);

  real_val = Sqrt(real_val); // This number will lie in the range of 32 to 64
  // Round to nearest even for a DP float number
  real_val = real_val + Real(2^47); // The integer is the size of the whole DP mantissa
  int_val = RoundDown(real_val); // Calculate rounding value
  error = real_val - Real(int_val);
  round_up = error > 0.5; // Error cannot be exactly 0.5 so do not need tie case
  if round_up then int_val = int_val+1;

real_val = Real(2^65)/Real(int_val); // Lies in the range 4096 <= real_val < 8192
int_val = RoundDown(real_val); // Round that (to nearest even) to give integer
error = real_val - Real(int_val);
round_up = error > 0.5 || (error == 0.5 && int_val<0> == '1'));
if round_up then int_val = int_val+1;

r = int_val;
assert 4096 <= r && r < 8192;
return r;

shared/functions/float/fpsqrt/FPSqrt
// FPSqrt()
// =========

bits(N) FPSqrt(bits(N) op, FPCRType fpcr)
assert N IN {16,32,64};
(fptype,sign,value) = FPUnpack(op, fpcr);

bits(N) result;
if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
  result = FPProcessNaN(fptype, op, fpcr);
elsif fptype == FPTYPE_Zero then
  result = FPZero(sign, N);
elsif fptype == FPTYPE_Infinity && sign == '0' then
  result = FPInfinity(sign, N);
elsif sign == '1' then
  result = FPDefaultNaN(fpcr, N);
  FPProcessException(FPExc_InvalidOp, fpcr);
elself else
  result = FPRound(Sqrt(value), fpcr, N);
  FPProcessDenorm(fptype, N, fpcr);
else
  return result;
shared/functions/float/fpsub/FPSub

// FPSub()
// ========

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr)

assert N IN {16,32,64};
rounding = FRoundingMode(fpcr);

(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
inf1 = (type1 == FPType_Infinity);
inf2 = (type2 == FPType_Infinity);
zero1 = (type1 == FPType_Zero);
zero2 = (type2 == FPType_Zero);

if inf1 && inf2 && sign1 == sign2 then
result = FPDefaultNaN(fpcr, N);
FPProcessException(FPExc_InvalidOp, fpcr);
elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
result = FPInfinity('0', N);
elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
result = FPInfinity('1', N);
elsif zero1 && zero2 && sign1 == NOT(sign2) then
result = FPZero(sign1, N);
else
result_value = value1 - value2;
if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
result = FPZero(result_sign, N);
else
result = FPRound(result_value, fpcr, rounding, N);
FPProcessDenorms(type1, type2, N, fpcr);

return result;

shared/functions/float/fpthree/FPThree

// FPThree()
// =========

bits(N) FPThree(bit sign, integer N)

assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '1':Zeros(E-1);
frac = '1':Zeros(F-1);
result = sign : exp : frac;

return result;

shared/functions/float/fptofixed/FPToFixed

// FPToFixed()
// ===========

// Convert N-bit precision floating point 'op' to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and Rounding.
bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding, integer M)

assert N IN {16,32,64};
assert M IN {16,32,64};
assert fbits >= 0;
assert rounding != FPRounding_ODD;

// When alternative floating-point support is TRUE, do not generate
// Input Denormal floating-point exceptions.
altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
fpexc = !altfp;

// Unpack using fpcr to determine if subnormals are flushed-to-zero.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

// If NaN, set cumulative flag or take exception.
if fptype == FPType_SNaN || fptype == FPType_QNaN then
    FPProcessException(FPExc_InvalidOp, fpcr);

// Scale by fractional bits and produce integer rounded towards minus-infinity.
value = value * 2.0^fbits;
int_result = RoundDown(value);
error = value - Real(int_result);

// Determine whether supplied rounding mode requires an increment.
boolean round_up;
case rounding of
    when FPRounding_TIEEVEN
        round_up = (error > 0.5 || (error == 0.5 && int_result<0) == '1'));
    when FPRounding_POSINF
        round_up = (error != 0.0);
    when FPRounding_NEGINF
        round_up = FALSE;
    when FPRounding_ZERO
        round_up = (error != 0.0 && int_result < 0);
    when FPRounding_TIEAWAY
        round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

if round_up then int_result = int_result + 1;

// Generate saturated result and exceptions.
(result, overflow) = SatQ(int_result, M, unsigned);
if overflow then
    FPProcessException(FPExc_InvalidOp, fpcr);
elseif error != 0.0 then
    FPProcessException(FPExc_Inexact, fpcr);
return result;

shared/functions/float/fptofixedjs/FPToFixedJS

// FPToFixedJS()
// =============

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

(bits(N), bit) FPToFixedJS(bits(M) op, FPCRType fpcr, boolean Is64, integer N)

assert M == 64 && N == 32;

// If FALSE, never generate Input Denormal floating-point exceptions.
fpcr_idenorm = !((HaveAltFP() && !UsingAArch32() && fpcr.AH == '1'));
// Unpack using fpcr to determine if subnormals are flushed-to-zero.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc_idenorm);

z = '1';
// If NaN, set cumulative flag or take exception.
if fptype == FPType_SNaN || fptype == FPType_QNaN then
    FPProcessException(FPExc_InvalidOp, fpcr);
    z = '0';

int_result = RoundDown(value);
error = value - Real(int_result);
// Determine whether supplied rounding mode requires an increment.
round_it_up = (error != 0.0 && int_result < 0);
if round_it_up then int_result = int_result + 1;
integer result;
if int_result < 0 then
    result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
else
    result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

// Generate exceptions.
if int_result < -(2^31) || int_result > (2^31)-1 then
    FPProcessException(FPExc_InvalidOp, fpcr);
    z = '0';
elsf error != 0.0 then
    FPProcessException(FPExc_Inexact, fpcr);
    z = '0';
elsf sign == '1' && value == 0.0 then
    z = '0';
elsf sign == '0' && value == 0.0 && !IsZero(op<51:0>) then
    z = '0';
if fptype == FPType_Infinity then result = 0;
return (result<N-1:0>, z);

shared/functions/float/fptwo/FPTwo

// FPTwo()
// =========

bits(N) FPTwo(bit sign, integer N)

assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '1':Zeros(E-1);
frac = Zeros(F);
result = sign : exp : frac;
return result;

shared/functions/float/fptype/FPType

enumeration FPType {FPType_Zero,
FPType_Denormal,
FPType_Nonzero,
FPType_Infinity,
FPType_QNaN,
FPType_SNaN};
shared/functions/float/fpunpack/FPUnpack

// FPUnpack()
// =========

(FTType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr_in)
  FPCRType fpcr = fpcr_in;
  fpcr.AHP = '0';
  boolean fpexc = TRUE; // Generate floating-point exceptions
  (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
  return (fp_type, sign, value);

// FPUnpack()
// =========

// Used by data processing, int/fixed to FP and FP to int/fixed conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(shared/functions/float/fpunpack/FPUnpackBase)

// FPUnpackBase()
// ==============

(shared/functions/float/fpunpack/FPUnpackBase)

// FPUnpackBase()
// ==============

(shared/functions/float/fpunpack/FPUnpackBase)

// FPUnpackBase()
// ==============

(shared/functions/float/fpunpack/FPUnpackBase)
if IsZero(exp16) then 
  if IsZero(frac16) || fpcr.FZ16 == '1' then 
    fptype = FPType_Zero;  value = 0.0; 
  else 
    fptype = FPType_Denormal;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10); 
  elsif IsOnes(exp16) && fpcr.AHP == '0' then  //Infinity or NaN in IEEE format 
    if IsZero(frac16) then 
      fptype = FPType_Infinity;  value = 2.0^1000000; 
    else 
      fptype = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN; 
      value = 0.0; 
    else 
      fptype = FPType_Nonzero; 
      value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10); 
  elsif N == 32 || isbfloat16 then 
    bits(8) exp32; 
    bits(23) frac32; 
    if isbfloat16 then 
      sign   = fpval<15>; 
      exp32  = fpval<14:7>; 
      frac32 = fpval<6:0> : Zeros(16); 
    else 
      sign   = fpval<31>; 
      exp32  = fpval<30:23>; 
      frac32 = fpval<22:0>; 
    if IsZero(exp32) then 
      // Produce zero if value is zero. 
      fptype = FPType_Zero;  value = 0.0; 
    elsif fz || fiz then        // Flush-to-zero if FIZ==1 or AH,FZ==01 
      fptype = FPType_Denormal;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23); 
    elsif IsOnes(exp32) then 
      if IsZero(frac32) then 
        fptype = FPType_Infinity;  value = 2.0^1000000; 
      else 
        fptype = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN; 
        value = 0.0; 
      else 
        fptype = FPType_Nonzero; 
        value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23); 
    else // N == 64 
      sign   = fpval<63>; 
      exp64  = fpval<62:52>; 
      frac64 = fpval<51:0>; 
    if IsZero(exp64) then 
      if IsZero(frac64) then 
        // Produce zero if value is zero. 
        fptype = FPType_Zero;  value = 0.0; 
      elsif fz || fiz then        // Flush-to-zero if FIZ==1 or AH,FZ==01 
        fptype = FPType_Denormal;  value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52); 
      else 
        fptype = FPType_Nonzero; 
        value = 2.0^1000000; 
      else 
        fptype = FPType_Infinity;  value = 2.0^1000000; 
      else 
        fptype = FPType_QNaN; 
        value = 0.0; 
      else 
        fptype = FPType_SNaN; 
        value = 0.0; 
    else 
      fptype = FPType_Nonzero; 
      value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52); 
      }
elsif IsOnes(exp64) then
  if IsZero(frac64) then
    fptype = FPType_Infinity; value = 2.0^1000000;
  else
    fptype = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
    value = 0.0;
  else
    fptype = FPType_Nonzero;
    value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);
  end if
if sign == '1' then value = -value;
return (fptype, sign, value);

shared/functions/float/fpunpack/FPUnpackCV

  // FPUnpackCV()
  // ============
  //
  // Used for FP to FP conversion instructions.
  // For half-precision data ignores FZ16 and observes AHP.

  (FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCRType fpcr_in)
  FPCRType fpcr = fpcr_in;
  fpcr.FZ16 = '0';
  boolean fpexc = TRUE;  // Generate floating-point exceptions
  (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
  return (fp_type, sign, value);

shared/functions/float/fpzero/FPZero

  // FPZero()
  // =========

  bits(N) FPZero(bit sign, integer N)
  assert N IN {16,32,64};
  constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
  constant integer F = N - (E + 1);
  exp  = Zeros(E);
  frac = Zeros(F);
  result = sign : exp : frac;
  return result;

shared/functions/float/vfpexpandimm/VFPExpandImm

  // VFPExpandImm()
  // ===============

  bits(N) VFPExpandImm(bits(8) imm8, integer N)
  assert N IN {16,32,64};
  constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
  constant integer F = (N - E) - 1;
  sign = imm8<7>;
  exp  = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
  frac = imm8<3:0>:Zeros(F-4);
  result = sign : exp : frac;
  return result;
**shared/functions/integer/AddWithCarry**

// AddWithCarry()  
// ==============  
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)  
integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);  
integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);  
bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>  
bit n = result<N-1>;  
bit z = if IsZero(result) then '1' else '0';  
bit c = if UInt(result) == unsigned_sum then '0' else '1';  
bit v = if SInt(result) == signed_sum then '0' else '1';  
return (result, n:z:c:v);

**shared/functions/interrupts/InterruptID**

enumeration InterruptID {  
InterruptID_PMUIRQ,  
InterruptID_COMMIRQ,  
InterruptID_CTIIIRQ,  
InterruptID_COMMRX,  
InterruptID_COMMTX,  
InterruptID_CNTP,  
InterruptID_CNTHP,  
InterruptID_CNTHPS,  
InterruptID_CNTPS,  
InterruptID_CNTV,  
InterruptID_CNTHV,  
InterruptID_CNTHVS,  
};

**shared/functions/interrupts/SetInterruptRequestLevel**

// Set a level-sensitive interrupt to the specified level.  
SetInterruptRequestLevel(InterruptID id, signal level);

**shared/functions/memory/AArch64.BranchAddr**

// AArch64.BranchAddr()  
// ===============  
// Return the virtual address with tag bits removed.  
// This is typically used when the address will be stored to the program counter.

bits(64) AArch64.BranchAddr(bits(64) vaddress, bits(2) el)  
assert !UsingAArch32();  
msbit = AddrTop(vaddress, TRUE, el);  
if msbit == 63 then  
return vaddress;  
elsif (el IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then  
return SignExtend(vaddress<msbit:0>, 64);  
else  
return ZeroExtend(vaddress<msbit:0>, 64);

**shared/functions/memory/AccType**

enumeration AccType {AccType_NORMAL, // Normal loads and stores  
AccType_STREAM, // Streaming loads and stores  
AccType_VEC, // Vector loads and stores  
AccType_VECSTREAM, // Streaming vector loads and stores}
AccType_SVE, \hspace{1cm} \text{// Scalable vector loads and stores}
AccType_SVESTREAM, \hspace{1cm} \text{// Scalable vector streaming loads and stores}
AccType_SME, \hspace{1cm} \text{// Scalable matrix loads and stores}
AccType_SMESTREAM, \hspace{1cm} \text{// Scalable matrix streaming loads and stores}
AccType_UNPRIVSTREAM, \hspace{1cm} \text{// Streaming unprivileged loads and stores}
AccType_A32LSMD, \hspace{1cm} \text{// Load and store multiple}
AccType_ATOMIC, \hspace{1cm} \text{// Atomic loads and stores}
AccType_ATOMICRW,
AccType_ORDERED, \hspace{1cm} \text{// Load-Acquire and Store-Release}
AccType_ORDEREDRW,
AccType_ORDEREDATOMIC,
AccType_ORDEREDATOMICRW,
AccType_ATOMICLS64, \hspace{1cm} \text{// Atomic 64-byte loads and stores}
AccType_LIMIITEDORDERED, \hspace{1cm} \text{// Load-LOAcquire and Store-LORelease}
AccType_UNPRIV,
AccType_IFETCH, \hspace{1cm} \text{// Instruction fetch}
AccType_TTW, \hspace{1cm} \text{// Translation table walk}
AccType_NONFAULT,
AccType_CNOTFIRST,
AccType_NV2REGISTER, \hspace{1cm} \text{// MRS/MSR instruction used at EL1 and which is converted to a memory access that uses the EL2 translation regime}
AccType_TRBE,
// Other operations
AccType_DC, \hspace{1cm} \text{// Data cache maintenance}
AccType_IC, \hspace{1cm} \text{// Instruction cache maintenance}
AccType_DCZVA,
AccType_ATPAN, \hspace{1cm} \text{// Address translation with PAN permission checks}
AccType_AT); \hspace{1cm} \text{// Address translation}

shared/functions/memory/AccessDescriptor
type AccessDescriptor is (  
    boolean transactional,  
    MPAMinfo mpam,  
    AccType acctype)

shared/functions/memory/AddrTop

// AddrTop()  
// =========  
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".  
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

integer AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
assert HaveEL(el);
    regime = S1TranslationRegime(el);
if ELUsingAArch32(regime) then  
    // AArch32 translation regime.  
    return 31;
else  
    if EffectiveTBI(address, IsInstr, el) == '1' then  
        return 55;
    else  
        return 63;

shared/functions/memory/Allocation
countant bits(2) MemHint_No = '00'; \hspace{1cm} \text{// No Read-Allocate, No Write-Allocate}
countant bits(2) MemHint_WA = '01'; \hspace{1cm} \text{// No Read-Allocate, Write-Allocate}
countant bits(2) MemHint_RA = '10'; \hspace{1cm} \text{// Read-Allocate, No Write-Allocate}
countant bits(2) MemHint_RWA = '11'; \hspace{1cm} \text{// Read-Allocate, Write-Allocate}
shared/functions/memory/BigEndian

// BigEndian()
// ===========

boolean BigEndian(AccType acctype)
    boolean bigend;
    if HaveNV2Ext() && acctype == AccType_NV2REGISTER then
        return SCTLR_EL2.EE == '1';
    end;
    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR[].E0E != '0');
    else
        bigend = (SCTLR[].EE != '0');
    end
    return bigend;

shared/functions/memory/BigEndianReverse

// BigEndianReverse()
// ==================

bits(width) BigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    integer half = width DIV 2;
    if width == 8 then return value;
    return BigEndianReverse(value<half-1:0>) : BigEndianReverse(value<width-1:half>);

shared/functions/memory/Cacheability

constant bits(2) MemAttr_NC = '00';     // Non-cacheable
constant bits(2) MemAttr_WT = '10';     // Write-through
constant bits(2) MemAttr_WB = '11';     // Write-back

shared/functions/memory/CreateAccessDescriptor

// CreateAccessDescriptor()
// ========================

AccessDescriptor CreateAccessDescriptor(AccType acctype)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.transactional = FALSE;
    accdesc.mpam = GenMPAMcurEL(acctype);
    return accdesc;

shared/functions/memory/DataMemoryBarrier

DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);

shared/functions/memory/DataSynchronizationBarrier

DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types, boolean nXS);
shared/functions/memory/DeviceType

enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};

shared/functions/memory/EffectiveTBI

// EffectiveTBI()
// ==============
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bit EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
bit tbi;
bit tbid;
assert HaveEL(el);
regime = S1TranslationRegime(el);
assert(!ELUsingAArch32(regime));

case regime of
 when EL1
   tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
   if HavePACExt() then
     tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
 when EL2
   if HaveVirtHostExt() && ELIsInHost(el) then
     tbi = if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0;
     if HavePACExt() then
       tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
   else
     tbi = TCR_EL2.TBI;
     if HavePACExt() then tbid = TCR_EL2.TBID;
 when EL3
   tbi = TCR_EL3.TBI;
   if HavePACExt() then tbid = TCR_EL3.TBID;

return (if tbi == '1' && (!HavePACExt() || tbid == '0' || !IsInstr) then '1' else '0');

shared/functions/memory/EffectiveTCMA

// EffectiveTCMA()
// ==============
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

bit EffectiveTCMA(bits(64) address, bits(2) el)
bit tcma;
assert HaveEL(el);
regime = S1TranslationRegime(el);
assert(!ELUsingAArch32(regime));

case regime of
 when EL1
   tcma = if address<55> == '1' then TCR_EL1.TCMA1 else TCR_EL1.TCMA0;
 when EL2
   if HaveVirtHostExt() && ELIsInHost(el) then
     tcma = if address<55> == '1' then TCR_EL2.TCMA1 else TCR_EL2.TCMA0;
   else
     tcma = TCR_EL2.TCMA;
 when EL3
   tcma = TCR_EL3.TCMA;

return tcma;
shared/functions/memory/Fault

enumeration Fault {
    Fault_None,
    Fault_AccessFlag,
    Fault_Alignment,
    Fault_Background,
    Fault_Domain,
    Fault_Permission,
    Fault_Translation,
    Fault_AddressSize,
    Fault_SyncExternal,
    Fault_SyncExternalOnWalk,
    Fault_SyncParity,
    Fault_SyncParityOnWalk,
    Fault_GPCFOnWalk,
    Fault_GPCFOnOutput,
    Fault_AsyncParity,
    Fault_AsyncExternal,
    Fault_Debug,
    Fault_TLBConflict,
    Fault_BranchTarget,
    Fault_HWUpdateAccessFlag,
    Fault_Lockdown,
    Fault_Exclusive,
    Fault_ICacheMaint
};

shared/functions/memory/FaultRecord

type FaultRecord is (Fault    statuscode, // Fault Status
    AccType  acctype,      // Type of access that faulted
    FullAddress ipaddress, // Intermediate physical address
    GPCFRecord gpcf,       // Granule Protection Check Fault record
    FullAddress paddress,  // Physical address
    boolean gpcfs2walk,    // GPC for a stage 2 translation table walk
    boolean s2fs1walk,     // Is on a Stage 1 translation table walk
    boolean write,         // TRUE for a write, FALSE for a read
    integer level,         // For translation, access flag and Permission faults
    bit extflag,           // IMPLEMENTATION DEFINED syndrome for External aborts
    boolean secondstage,   // Is a Stage 2 abort
    bits(4) domain,        // Domain number, AArch32 only
    bits(2) errortype,     // [Armv8.2 RAS] AArch32 AET or AArch64 SET
    bits(4) debugmoe)      // Debug method of entry, from AArch32 only

shared/functions/memory/FullAddress

type FullAddress is (  
PASpace paspace,
    bits(52) address
);

shared/functions/memory/GPCF

enumeration GPCF {
    GPCF_None,        // No fault
    GPCF_AddressSize, // GPT address size fault
    GPCF_Walk,        // GPT walk fault
    GPCF_EABT,        // Synchronous External abort on GPT fetch
    GPCF_Fail         // Granule protection fault
};
shared/functions/memory/GPCFRecord

type GPCFRecord is (
    GPCF gpf,
    integer level
)

shared/functions/memory/Hint_Prefetch

// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.

Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);

shared/functions/memory/IsDataAccess

// IsDataAccess()
// ==============
// Return TRUE if access is to data memory.

boolean IsDataAccess(AccType acctype)
    return !(acctype IN {AccType_IFETCH, AccType_TTW, AccType_DC, AccType_IC, AccType_AT, AccType_ATPAN});

shared/functions/memory/MBReqDomain

enumeration MBReqDomain    {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable, MBReqDomain_OuterShareable, MBReqDomain_FullSystem};

shared/functions/memory/MBReqTypes

enumeration MBReqTypes     {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};

shared/functions/memory/MPAM

type PARTIDtype = bits(16);
type PMGtype = bits(8);

enumeration PARTIDspaceType {
    PIDSpace_Secure,
    PIDSpace_ROOT,
    PIDSpace_REALM,
    PIDSpace_NONSECURE
};

type MPAMinfo is (
    PARTIDspaceType mpam_sp,
    PARTIDtype partid,
shared/functions/memory/MemAttrHints

type MemAttrHints is (  
    bits(2) attrs,  // See MemAttr_*, Cacheability attributes  
    bits(2) hints,  // See MemHint_*, Allocation hints  
    boolean transient
)

shared/functions/memory/MemType

enumeration MemType {MemType_Normal, MemType_Device};

shared/functions/memory/MemoryAttributes

type MemoryAttributes is (  
    MemType memtype,  // For Device memory types  
    DeviceType device,  // Inner hints and attributes  
    MemAttrHints inner,  // Outer hints and attributes  
    MemAttrHints outer,  // Shareability attribute  
    boolean tagged,  // Tagged access  
    bit xs  // XS attribute
)

shared/functions/memory/PASpace

enumeration PASpace {  
    PAS_NonSecure,  
    PAS_Secure,  
    PAS_Root,  
    PAS_Realm
};

shared/functions/memory/Permissions

type Permissions is (  
    bits(2) ap_table,  // Stage 1 hierarchical access permissions  
    bit xn_table,  // Stage 1 hierarchical execute-never for single EL regimes  
    bit pxn_table,  // Stage 1 hierarchical privileged execute-never  
    bit uxn_table,  // Stage 1 hierarchical unprivileged execute-never  
    bits(3) ap,  // Stage 1 access permissions  
    bit xn,  // Stage 1 execute-never for single EL regimes  
    bit uxn,  // Stage 1 unprivileged execute-never  
    bit pxn,  // Stage 1 privileged execute-never  
    bits(2) s2ap,  // Stage 2 access permissions  
    bit s2xn,  // Stage 2 extended execute-never  
    bit s2xn  // Stage 2 execute-never
)

shared/functions/memory/PhysMemRead

// Returns the value read from memory, and a status.  // Returned value is UNKNOWN if an External abort occurred while reading the memory.
// Otherwise the PhysMemRetStatus statuscode is Fault_None.
(PhysMemRetStatus, bits(8*size)) PhysMemRead(AddressDescriptor desc, integer size,
AccessDescriptor accdesc);

shared/functions/memory/PhysMemRetStatus

type PhysMemRetStatus is (Fault statuscode, bit extflag, // Fault Status
  bit errortype, // syndrome for External aborts
  bits(2) errortype, // optional error state
  bits(64) store64bstatus, // status of 64B store
  AccType accctype) // Type of access that faulted

shared/functions/memory/PhysMemWrite

// Writes the value to memory, and returns the status of the write.
// If there is an External abort on the write, the PhysMemRetStatus indicates this.
// Otherwise the statuscode of PhysMemRetStatus is Fault_None.
PhysMemRetStatus PhysMemWrite(AddressDescriptor desc, integer size, AccessDescriptor accdesc,
  bits(8*size) value);

shared/functions/memory/PrefetchHint

e Numberation PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};

shared/functions/memory/Shareability

enumeration Shareability {
  Shareability_NSH,
  Shareability_ISH,
  Shareability_OSH
};

shared/functions/memory/SpeculativeStoreBypassBarrierToPA

SpeculativeStoreBypassBarrierToPA();

shared/functions/memory/SpeculativeStoreBypassBarrierToVA

SpeculativeStoreBypassBarrierToVA();

shared/functions/memory/Tag

c onstant integer LOG2_TAG_GRANULE = 4;

const ant integer TAG_GRANULE = 1 << LOG2_TAG_GRANULE;

shared/functions/mpam/AltPARTIDspace

// AltPARTIDspace()
// ================
// From the Security state, EL and ALTSP configuration, determine
\[
\text{whether to primary space or the alt space is selected and which}
\text{PARTID space is the alternative space. Return that alternative}
\text{PARTID space if selected or the primary space if not.}
\]

\[
\text{PARTIDspaceType AltPARTIDspace(bits(2) el, SecurityState security,}
\text{PARTIDspaceType primaryPIdSpace)}
\]

\[
\text{case security of}
\]

\[
\text{when SS\_NonSecure}
\]

\[
\text{assert el \!\!=\!\!= EL3;}
\]

\[
\text{return primaryPIdSpace; // there is no ALTSP for Non\_secure}
\]

\[
\text{when SS\_Secure}
\]

\[
\text{assert el \!\!=\!\!= EL3;}
\]

\[
\text{if primaryPIdSpace == PIdSpace\_NonSecure then}
\]

\[
\text{return primaryPIdSpace;}
\]

\[
\text{return AltPIdSecure(el, primaryPIdSpace)}
\]

\[
\text{when SS\_Root}
\]

\[
\text{assert el \!\!=\!\!= EL3;}
\]

\[
\text{if MPAM3\_EL3\_ALTSP\_EL3 == '1' then}
\]

\[
\text{if MPAM3\_EL3\_RT\_ALTSP\_NS == '1' then}
\]

\[
\text{return PIdSpace\_NonSecure;}
\]

\[
\text{else}
\]

\[
\text{return PIdSpace\_Secure;}
\]

\[
\text{else}
\]

\[
\text{return primaryPIdSpace;}
\]

\[
\text{when SS\_Realm}
\]

\[
\text{assert el \!\!=\!\!= EL3;}
\]

\[
\text{return AltPIdRealm(el, primaryPIdSpace)}
\]

\[
\text{otherwise}
\]

\[
\text{Unreachable();}
\]

\[
\text{shared/functions/mpam/AltPIdRealm}
\]

\[
\text{// AltPIdRealm()}
\]

\[
\text{// =============}
\]

\[
\text{// Compute PARTID space as either the primary PARTID space or}
\]

\[
\text{// alternative PARTID space in the Realm Security state.}
\]

\[
\text{// Helper for AltPARTIDspace.}
\]

\[
\text{PARTIDspaceType AltPIdRealm(bits(2) el, PARTIDspaceType primaryPIdSpace)}
\]

\[
\text{PARTIDspaceType PIdSpace = primaryPIdSpace;}
\]

\[
\text{case el of}
\]

\[
\text{when EL0}
\]

\[
\text{if ELIsInHost(EL0) then}
\]

\[
\text{if \!UsePrimarySpaceEL2() then}
\]

\[
\text{PIdSpace = PIdSpace\_NonSecure;}
\]

\[
\text{elsif \!UsePrimarySpaceEL10() then}
\]

\[
\text{PIdSpace = PIdSpace\_NonSecure;}
\]

\[
\text{when EL1}
\]

\[
\text{if \!UsePrimarySpaceEL10() then}
\]

\[
\text{PIdSpace = PIdSpace\_NonSecure;}
\]

\[
\text{when EL2}
\]

\[
\text{if \!UsePrimarySpaceEL2() then}
\]

\[
\text{PIdSpace = PIdSpace\_NonSecure;}
\]

\[
\text{otherwise}
\]

\[
\text{Unreachable();}
\]

\[
\text{return PIdSpace;}
\]

\[
\text{shared/functions/mpam/AltPIdSecure}
\]

\[
\text{// AltPIdSecure()}
\]

\[
\text{// =============}
\]

\[
\text{// Compute PARTID space as either the primary PARTID space or}
\]

\[
\text{// alternative PARTID space in the Secure Security state.}
\]

\[
\text{// Helper for AltPARTIDspace.}
\]
PARTIDspaceType AltPIdSecure(bits(2) el, PARTIDspaceType primaryPIdSpace)

PARTIDspaceType PIdSpace = primaryPIdSpace;

boolean el2en = EL2Enabled();

case el of

when EL0
  if el2en then
    if ELIsInHost(EL0) then
      if !UsePrimarySpaceEL2() then
        PIdSpace = PIdSpace_NonSecure;
      elsif !UsePrimarySpaceEL1() then
        PIdSpace = PIdSpace_NonSecure;
      elsif MPAM3_EL3.ALTSP_HEN == '0' && MPAM3_EL3.ALTSP_HFC == '1' then
        PIdSpace = PIdSpace_NonSecure;
    when EL1
      if el2en then
        if !UsePrimarySpaceEL1() then
          PIdSpace = PIdSpace_NonSecure;
        elsif MPAM3_EL3.ALTSP_HEN == '0' && MPAM3_EL3.ALTSP_HFC == '1' then
          PIdSpace = PIdSpace_NonSecure;
        when EL2
          if !UsePrimarySpaceEL2() then
            PIdSpace = PIdSpace_NonSecure;
        otherwise
          Unreachable();
      return PIdSpace;

shared/functions/mpam/DefaultMPAMinfo

// DefaultMPAMinfo()
// =============
// Returns default MPAM info. The partidspace argument sets
// the PARTID space of the default MPAM information returned.

MPAMinfo DefaultMPAMinfo(PARTIDspaceType partidspace)

  MPAMinfo DefaultInfo;
  DefaultInfo.mpam_sp = partidspace;
  DefaultInfo.partid  = DefaultPARTID;
  DefaultInfo.pmg     = DefaultPMG;
  return DefaultInfo;

shared/functions/mpam/DefaultPARTID

const PARTIDtype DefaultPARTID = 0<15:0>;

shared/functions/mpam/DefaultPMG

const PMGtype DefaultPMG = 0<7:0>;

shared/functions/mpam/GenMPAMcurEL

// GenMPAMcurEL()
// =============
// Returns MPAMinfo for the current EL and security state.
// May be called if MPAM is not implemented (but in a version that supports
// MPAM), MPAM is disabled, or in AArch32. In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation. If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMcurEL(AccType accType)
bits(2) mpamEL;
boolean validEL = FALSE;
SecurityState security = CurrentSecurityState();
boolean InD = FALSE;
boolean InSM = FALSE;
PARTIDspaceType pspace = PARTIDspaceFromSS(security);
if pspace == PIdSpace_NonSecure && !MPAMisEnabled() then
    return DefaultMPAMinfo(pspace);
if UsingAArch32() then
    (validEL, mpamEL) = ELFromM32(PSTATE.M);
else
    mpamEL = if acctype == AccType_NV2REGISTER then EL2 else PSTATE.EL;
    validEL = TRUE;

case acctype of
    when AccType_IFETCH, AccType_IC
        InD = TRUE;
    when AccType_SME, AccType_SMESTREAM
        InD = (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
            boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1 label precedence");
    when AccType_VEC, AccType_VECSTREAM
        InD = (HaveSME() && PSTATE.SM == '1' &&
            (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
            boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1 label precedence");
    when AccType_SVE, AccType_SVESTREAM, AccType_NONFAULT, AccType_CNOTFIRST
        InD = (HaveSME() && PSTATE.SM == '1' &&
            (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
            boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1 label precedence");
    otherwise
        // Other access types are DATA accesses
        InD = FALSE;

if !validEL then
    return DefaultMPAMinfo(pspace);
elsif HaveRME() && MPAMIDR_EL1.HAS_ALTSP == '1' then
    // Substitute alternative PARTID space if selected
    pspace = AltPARTIDspace(mpamEL, security, pspace);
if HaveMPAMv0p1Ext() && MPAMIDR_EL1.HAS_FORCE_NS == '1' then
    if MPAM3_EL3.FORCE_NS == '1' && security == SS_Secure then
        pspace = PIdSpace_NonSecure;
else
    if !MPAMisEnabled() then
        return DefaultMPAMinfo(pspace);
else
    return genMPAM(mpamEL, InD, InSM, pspace);

shared/functions/mpam/MAP_vPARTID

// MAP_vPARTID()
// ==============
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.

(PARTIDtype, boolean) MAP_vPARTID(PARTIDtype vpartid)
    // should not ever be called if EL2 is not implemented
    // or is implemented but not enabled in the current
    // security state.
    PARTIDtype ret;
    boolean err;
    integer virt = UInt(vpartid);
    integer vpmrmax = UInt(MPAMIDR_EL1.VPMR_MAX);

    // vpartid_max is largest vpartid supported
    integer vpartid_max = (vpmrmax << 2) + 3;
// One of many ways to reduce vpartid to value less than vpartid_max.
if UInt(vpartid) > vpartid_max then
    virt = virt MOD (vpartid_max+1);

// Check for valid mapping entry.
if MPAMVPMV_EL2<virt> == '1' then
    // vpartid has a valid mapping so access the map.
    ret = mapvpmw(virt);
    err = FALSE;

// Is the default virtual PARTID valid?
elsif MPAMVPMV_EL2<0> == '1' then
    // Yes, so use default mapping for vpartid == 0.
    ret = MPAMVPM0_EL2<0 +: 16>;
    err = FALSE;

// Neither is valid so use default physical PARTID.
else
    ret = DefaultPARTID;
    err = TRUE;

// Check that the physical PARTID is in-range.
// This physical PARTID came from a virtual mapping entry.
integer partid_max = UInt(MPAMIDR_EL1.PARTID_MAX);
if UInt(ret) > partid_max then
    // Out of range, so return default physical PARTID
    ret = DefaultPARTID;
    err = TRUE;
return (ret, err);


// MPAMisEnabled()
// ===============
// Returns TRUE if MPAMisEnabled.

boolean MPAMisEnabled()
el = HighestEL();
case el of
    when EL3 return MPAM3_EL3.MPAMEN == '1';
    when EL2 return MPAM2_EL2.MPAMEN == '1';
    when EL1 return MPAM1_EL1.MPAMEN == '1';

// MPAMisVirtual()
// ===============
// Returns TRUE if MPAM is configured to be virtual at EL.

boolean MPAMisVirtual(bits(2) el)
return (MPAMIDR_EL1.HAS_HCR == '1' && EL2Enabled() &&
        ((el == EL0 && MPAMHCR_EL2.EL0_VPMEN == '1' &&
        (HCR_EL2.E2H == '0' || HCR_EL2.TGE == '0')) ||
        (el == EL1 && MPAMHCR_EL2.EL1_VPMEN == '1')));

// PARTIDspaceFromSS()
// =============
// Returns the primary PARTID space from the Security State.

PARTIDspaceType PARTIDspaceFromSS(SecurityState security)
case security of
when SS_NonSecure
    return PIdSpace_NonSecure;
when SS_Root
    return PIdSpace_Root;
when SS_Rolem
    return PIdSpace_Rolem;
when SS_Secure
    return PIdSpace_Secure;
otherwise
    Unreachable();

shared/functions/mpam/UsePrimarySpaceEL10

// UsePrimarySpaceEL10()
//=-----------------------
// Checks whether Primary space is configured in the
// MPAM3_EL3 and MPAM2_EL2 ALTSP control bits that affect
// MPAM ALTSP use at EL1 and EL0.

boolean UsePrimarySpaceEL10()
if MPAM3_EL3.ALTSP_HEN == '0' then
    return MPAM3_EL3.ALTSP_HFC == '0';
return !MPAMisEnabled() || !EL2Enabled() || MPAM2_EL2.ALTSP_HFC == '0';

shared/functions/mpam/UsePrimarySpaceEL2

// UsePrimarySpaceEL2()
//=---------------------
// Checks whether Primary space is configured in the
// MPAM3_EL3 and MPAM2_EL2 ALTSP control bits that affect
// MPAM ALTSP use at EL2.

boolean UsePrimarySpaceEL2()
if MPAM3_EL3.ALTSP_HEN == '0' then
    return MPAM3_EL3.ALTSP_HFC == '0';
return !MPAMisEnabled() || MPAM2_EL2.ALTSP_EL2 == '0';

shared/functions/mpam/genMPAM

// genMPAM()
//=--------
// Returns MPAMinfo for exception level el.
// If InD is TRUE returns MPAM information using PARTID_I and PMG_I fields
// of MPAMel_ELx register and otherwise using PARTID_D and PMG_D fields.
// If InSM is TRUE returns MPAM information using PARTID_D and PMG_D fields
// of MPAMSM_EL1 register.
// Produces a PARTID in PARTID space pspace.

MPAMinfo genMPAM(bits(2) el, boolean InD, boolean InSM, PARTIDspaceType pspace)
MPAMinfo returninfo;
PARTIDtype partidel;
boolean perr;
// gstplk is guest OS application locked by the EL2 hypervisor to
// only use EL1 the virtual machine's PARTIDs.
boolean gstplk = (el == EL0 && EL2Enabled() &&
    MPAMHCR_EL2.GSTAPP_PLK == '1' &&
    NCR_EL2.TGE == '0');
bits(2) eff_el = if gstplk then EL1 else el;
(partidel, perr) = genPARTID(eff_el, InD, InSM);
PARTIDtype groupel = genPMG(eff_el, InD, InSM, perr);
returninfo.mpam_sp = pspace;
returninfo.partid = partidel;
returninfo.pmg = groupel;
return returninfo;

shared/functions/mpam/genPARTID

// genPARTID()
// ===========
// Returns physical PARTID and error boolean for exception level el.
// If InD is TRUE then PARTID is from MPAMel_ELx.PARTID_I and
// otherwise from MPAMel_ELx.PARTID_D.
// If InSM is TRUE then PARTID is from MPAMSMSM_EL1.PARTID_D.

(PARTIDtype, boolean) genPARTID(bits(2) el, boolean InD, boolean InSM)
PARTIDtype partidel = getMPAM_PARTID(el, InD, InSM);
PARTIDtype partid_max = MPAMIDR_EL1.PARTID_MAX;
if UInt(partidel) > UInt(partid_max) then
  return (DefaultPARTID, TRUE);
if MPAMisVirtual(el) then
  return MAP_vPARTID(partidel);
else
  return (partidel, FALSE);

shared/functions/mpam/genPMG

// genPMG()
// =========
// Returns PMG for exception level el and I- or D-side (InD).
// If PARTID generation (genPARTID) encountered an error, genPMG() should be
// called with partid_err as TRUE.

PMGtype genPMG(bits(2) el, boolean InD, boolean InSM, boolean partid_err)
integer pmg_max = UInt(MPAMIDR_EL1.PMG_MAX);
// It is CONSTRAINED UNPREDICTABLE whether partid_err forces PMG to
// use the default or if it uses the PMG from getMPAM_PMG.
if partid_err then
  return DefaultPMG;
PMGtype groupel = getMPAM_PMG(el, InD, InSM);
if UInt(groupel) <= pmg_max then
  return groupel;
return DefaultPMG;

shared/functions/mpam/getMPAM_PARTID

// getMPAM_PARTID()
// ================
// Returns a PARTID from one of the MPAMn_ELx or MPAMSMSM_EL1 registers.
// If InSM is TRUE, the MPAMSMSM_EL1 register is used. Otherwise,
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PARTID_I field of that
// register. Otherwise, selects the PARTID_D field.

PARTIDtype getMPAM_PARTID(bits(2) MPAMn, boolean InD, boolean InSM)
PARTIDtype partid;
boolean el2avail = EL2Enabled();

if InSM then
  partid = MPAMSMSM_EL1.PARTID_D;
  return partid;
if InD then
  case MPAMn of
    when '11' partid = MPAM3_EL3.PARTID_I;
    when '10' partid = if el2avail then MPAM2_EL2.PARTID_I else Zeros(16);
when '01' partid = MPAM1_EL1.PARTID_I;
when '00' partid = MPAM0_EL1.PARTID_I;
otherwise partid = PARTIDtype UNKNOWN;
else
  case MPAMn of
    when '11' partid = MPAM3_EL3.PARTID_D;
    when '10' partid = if el2avail then MPAM2_EL2.PARTID_D else Zeros(16);
    when '01' partid = MPAM1_EL1.PARTID_D;
    when '00' partid = MPAM0_EL1.PARTID_D;
  otherwise partid = PARTIDtype UNKNOWN;
return partid;

shared/functions/mpam/getMPAM_PMG

// getMPAM_PMG()
// =============
// Returns a PMG from one of the MPAMn_ELx or MPAMSM_EL1 registers.
// If InSM is TRUE, the MPAMSM_EL1 register is selected. Otherwise,
// MPAMn-selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PMG_I field of that
// register. Otherwise, selects the PMG_D field.

PMGtype getMPAM_PMG(bits(2) MPAMn, boolean InD, boolean InSM)

PMGtype pmg;
boolean el2avail = EL2Enabled();

if InSM then
  pmg = MPAMSM_EL1.PMG_D;
  return pmg;
else
  if InD then
    case MPAMn of
      when '11' pmg = MPAM3_EL3.PMG_I;
      when '10' pmg = if el2avail then MPAM2_EL2.PMG_I else Zeros(8);
      when '01' pmg = MPAM1_EL1.PMG_I;
      when '00' pmg = MPAM0_EL1.PMG_I;
    otherwise pmg = PMGtype UNKNOWN;
  else
    case MPAMn of
      when '11' pmg = MPAM3_EL3.PMG_D;
      when '10' pmg = if el2avail then MPAM2_EL2.PMG_D else Zeros(8);
      when '01' pmg = MPAM1_EL1.PMG_D;
      when '00' pmg = MPAM0_EL1.PMG_D;
    otherwise pmg = PMGtype UNKNOWN;
  return pmg;

shared/functions/mpam/mapvpmw

// mapvpmw()
// =========
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

PARTIDtype mapvpmw(integer vpartid)

bits(64) vpmw;
integer wd = vpartid DIV 4;
case wd of
  when 0 vpmw = MPAMVPM0_EL2;
  when 1 vpmw = MPAMVPM1_EL2;
  when 2 vpmw = MPAMVPM2_EL2;
  when 3 vpmw = MPAMVPM3_EL2;
  when 4 vpmw = MPAMVPM4_EL2;
when 5 vpmw = MPAMVP5_EL2;
when 6 vpmw = MPAMVP6_EL2;
when 7 vpmw = MPAMVP7_EL2;
otherwise vpmw = Zeros(64);
// vpme_lsb selects LSB of field within register
integer vpme_lsb = (vpartid MOD 4) * 16;
return vpmw<vpme_lsb +: 16>;

shared/functions/predictionrestrict/ASID

// ASID[]
// ======
// Effective ASID.
bits(16) ASID[]
if EL2Enabled() & & !ELUsingAArch32(EL2) & & HCR_EL2.<E2H, TGE> == '11' then
  if TCR_EL2.A1 == '1' then
    return TTBR1_EL2.ASID;
  else
    return TTBR0_EL2.ASID;
elsif !ELUsingAArch32(EL1) then
  if TCR_EL1.A1 == '1' then
    return TTBR1_EL1.ASID;
  else
    return TTBR0_EL1.ASID;
else
  if TTBCR.EAE == '0' then
    return ZeroExtend(CONTEXTIDR.ASID, 16);
  else
    if TTBCR.A1 == '1' then
      return ZeroExtend(TTBR1.ASID, 16);
    else
      return ZeroExtend(TTBR0.ASID, 16);

shared/functions/predictionrestrict/ExecutionCntxt

type ExecutionCntxt is (
  boolean is_vmid_valid, // is vmid valid for current context
  boolean all_vmid,     // should the operation be applied for all vmids
  bits(16) vmid,        // if all_vmid = FALSE, vmid to which operation is applied
  boolean is_asid_valid, // is asid valid for current context
  boolean all_asid,     // should the operation be applied for all asids
  bits(16) asid,        // if all_asid = FALSE, ASID to which operation is applied
  bits(2) target_el,    // target EL at which operation is performed
  SecurityState security,
  RestrictType restriction // type of restriction operation
)

shared/functions/predictionrestrict/RESTRICT_PREDICTIONS

// RESTRICT_PREDICTIONS()
// =======================
// Clear all speculated values.
RESTRICT_PREDICTIONS(ExecutionCntxt c)
IMPLEMENTATION_DEFINED;
shared/functions/predictionrestrict/RestrictType

enumeration RestrictType {
  RestrictType_DataValue,
  RestrictType_ControlFlow,
  RestrictType_CachePrefetch
};

shared/functions/predictionrestrict/TargetSecurityState

// TargetSecurityState()
// ==============
// Decode the target security state for the prediction context.

SecurityState TargetSecurityState(bit NS, bit NSE)
  curr_ss = SecurityStateAtEL(PSTATE.EL);
  if curr_ss == SS_NonSecure then
    return SS_NonSecure;
  elsif curr_ss == SS_Secure then
    case NS of
      when '0' return SS_Secure;
      when '1' return SS_NonSecure;
    end;
  elsif HaveRME() then
    if curr_ss == SS_Root then
      case NSE:NS of
        when '00' return SS_Secure;
        when '01' return SS_NonSecure;
        when '11' return SS_Realm;
        when '10' return SS_Root;
      end;
    elsif curr_ss == SS_Realm then
      return SS_Realm;
    end;
  end;

shared/functions/registers/BranchTo

// BranchTo()
// =========
// Set program counter to a new address, with a branch type.
// Parameter branch_conditional indicates whether the executed branch has a conditional encoding.
// In AArch64 state the address might include a tag in the top eight bits.

BranchTo(bits(N) target, BranchType branch_type, boolean branch_conditional)
  Hint_Branch(branch_type);
  if N == 32 then
    assert UsingAArch32();
    _PC = ZeroExtend(target, 64);
  else
    assert N == 64 && !UsingAArch32();
    bits(64) target_vaddress = AArch64.BranchAddr(target<63:0>, PSTATE.EL);
    if (HaveBRBExt() &&
      branch_type IN {BranchType_DIR, BranchType_INDIR,
                      BranchType_DIRCALL, BranchType_INDCALL,
                      BranchType_RET}) then
      BRBEBran
    
shared/functions/registers/BranchToAddr

// BranchToAddr()
// ===============
// Set program counter to a new address, with a branch type.
// In AArch64 state the address does not include a tag in the top eight bits.

BranchToAddr(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target, 64);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target<63:0>;
    return;

shared/functions/registers/BranchType

enumeration BranchType {
    BranchType_DIRCALL, // Direct Branch with link
    BranchType_INDCALL, // Indirect Branch with link
    BranchType_ERET, // Exception return (indirect)
    BranchType_DBGEXIT, // Exit from Debug state
    BranchType_RET, // Indirect branch with function return hint
    BranchType_DIR, // Direct branch
    BranchType_INDIR, // Indirect branch
    BranchType_EXCEPTION, // Exception entry
    BranchType_TMFAIL, // Transaction failure
    BranchType_RESET, // Reset
    BranchType_UNKNOWN}; // Other

shared/functions/registers/Hint_Branch

// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.
Hint_Branch(BranchType hint);

shared/functions/registers/NextInstrAddr

// Return address of the sequentially next instruction.
bits(N) NextInstrAddr(integer N);

shared/functions/registers/ResetExternalDebugRegisters

// Reset the External Debug registers in the Core power domain.
ResetExternalDebugRegisters(boolean cold_reset);

shared/functions/registers/ThisInstrAddr

// ThisInstrAddr()
// ===============
// Return address of the current instruction.

bits(N) ThisInstrAddr(integer N)
    assert N == 64 || (N == 32 && UsingAArch32());
    return _PC<N-1:0>;

shared/functions/registers/_PC

// Program Counter, _PC
bits(64) _PC;
shared/functions/registers/_R

// Registers, _R
array bits(64) _R[0..30];

shared/functions/sysregisters/SPSR

// SPSR[] - non-assignment form
// ============================

bits(N) SPSR[]
bits(N) result;
if UsingAArch32() then
  assert N == 32;
case PSTATE.M of
    when M32_FIQ      result = SPSR_fiq<N-1:0>;
    when M32_IRQ      result = SPSR_irq<N-1:0>;
    when M32_Svc      result = SPSR_svc<N-1:0>;
    when M32_Monitor  result = SPSR_mon<N-1:0>;
    when M32_Abort    result = SPSR_abt<N-1:0>;
    when M32_Hyp      result = SPSR_hyp<N-1:0>;
    when M32.Undef    result = SPSR_und<N-1:0>;
    otherwise        Unreachable();
else
  assert N == 64;
case PSTATE.EL of
    when EL1          result = SPSR_EL1<N-1:0>;
    when EL2          result = SPSR_EL2<N-1:0>;
    when EL3          result = SPSR_EL3<N-1:0>;
    otherwise        Unreachable();
return result;

// SPSR[] - assignment form
// ========================

SPSR[] = bits(N) value
if UsingAArch32() then
  assert N == 32;
case PSTATE.M of
    when M32_FIQ      SPSR_fiq<N-1:0> = value<N-1:0>;
    when M32_IRQ      SPSR_irq<N-1:0> = value<N-1:0>;
    when M32_Svc      SPSR_svc<N-1:0> = value<N-1:0>;
    when M32_Monitor  SPSR_mon<N-1:0> = value<N-1:0>;
    when M32_Abort    SPSR_abt<N-1:0> = value<N-1:0>;
    when M32_Hyp      SPSR_hyp<N-1:0> = value<N-1:0>;
    when M32.Undef    SPSR_und<N-1:0> = value<N-1:0>;
    otherwise        Unreachable();
else
  assert N == 64;
case PSTATE.EL of
    when EL1          SPSR_EL1<N-1:0> = value<N-1:0>;
    when EL2          SPSR_EL2<N-1:0> = value<N-1:0>;
    when EL3          SPSR_EL3<N-1:0> = value<N-1:0>;
    otherwise        Unreachable();
return;

shared/functions/system/ArchVersion

enumeration ArchVersion {
  ARMv8p0,
  ARMv8p1,
  ARMv8p2,
  ARMv8p3,
  ARMv8p4
shared/functions/system/BranchTargetCheck

// BranchTargetCheck()
// ===============
// This function is executed checks if the current instruction is a valid target for a branch
// taken into, or inside, a guarded page. It is executed on every cycle once the current
// instruction has been decoded and the values of InGuardedPage and BTypeCompatible have been
// determined for the current instruction.

BranchTargetCheck()
    assert HaveBTIExt() && !UsingAArch32();

    // The branch target check considers two state variables:
    // * InGuardedPage, which is evaluated during instruction fetch.
    // * BTypeCompatible, which is evaluated during instruction decode.
    if InGuardedPage && PSTATE.BTYPE != '00' && !BTypeCompatible && !Halted() then
        bits(64) pc = ThisInstrAddr(64);
        AArch64.BranchTargetException(pc<51:0>);

        boolean branch_instr = AArch64.ExecutingBROrBLROrRetInstr();
        boolean bti_instr = AArch64.ExecutingBTIInstr();

        // PSTATE.BTYPE defaults to 00 for instructions that do not explicitly set BTYPE.
        if !(branch_instr || bti_instr) then
            BTypeNext = '00';

shared/functions/system/ClearEventRegister

// ClearEventRegister()
// ===============
// Clear the Event Register of this PE.

ClearEventRegister()
    EventRegister = '0';
    return;

shared/functions/system/ClearPendingPhysicalSError

// Clear a pending physical SError interrupt.
ClearPendingPhysicalSError();

shared/functions/system/ClearPendingVirtualSError

// Clear a pending virtual SError interrupt.
ClearPendingVirtualSError();

shared/functions/system/ConditionHolds

// ConditionHolds()
// ===============
// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
// Evaluate base condition.
boolean result;
case cond<3:1> of
    when '000' result = (PSTATE.Z == '1'); // EQ or NE
    when '001' result = (PSTATE.C == '1'); // CS or CC
    when '010' result = (PSTATE.N == '1'); // MI or PL
    when '011' result = (PSTATE.V == '1'); // VS or VC
    when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0'); // HI or LS
    when '101' result = (PSTATE.N == PSTATE.V); // GE or LT
    when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0'); // GT or LE
    when '111' result = TRUE; // AL

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;
    return result;

shared/functions/system/ConsumptionOfSpeculativeDataBarrier
ConsumptionOfSpeculativeDataBarrier();

shared/functions/system/CurrentInstrSet
// CurrentInstrSet()
// ================
InstrSet CurrentInstrSet()
    InstrSet result;
    if UsingAArch32() then
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet_A64;
    return result;

shared/functions/system/CurrentPL
// CurrentPL()
// =========
PrivilegeLevel CurrentPL()
    return PLOFEL(PSTATE.EL);

shared/functions/system/CurrentSecurityState
// CurrentSecurityState()
// ================
// Returns the effective security state at the exception level based off current settings.
SecurityState CurrentSecurityState()
    return SecurityStateAtEL(PSTATE.EL);

shared/functions/system/DSBAlias
enumeration DSBAlias {DSBAlias_SSBB, DSBAlias_PSSBB, DSBAlias_DSB};
shared/functions/system/EL0

constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';

shared/functions/system/EL2Enabled

// EL2Enabled()
// ============
// Returns TRUE if EL2 is present and executing
// - with the PE in Non-secure state when Non-secure EL2 is implemented, or
// - with the PE in Realm state when Realm EL2 is implemented, or
// - with the PE in Secure state when Secure EL2 is implemented and enabled, or
// - when EL3 is not implemented.

boolean EL2Enabled()
return HaveEL(EL2) && (!HaveEL(EL3) || SCR_GEN[].NS == '1' || IsSecureEL2Enabled());

shared/functions/system/EL3SDDTrapPriority

// EL3SDDTrapPriority()
// ====================
// Returns TRUE if in Debug state, EDSCR.SDD is set, and an EL3 trap by an
// EL3 control register has the priority of the original trap exception.
// The IMPLEMENTATION DEFINED priority may be different for each case.

boolean EL3SDDTrapPriority()
return (Halted() && EDSCR.SDD == '1' &&
boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'";)

shared/functions/system/ELFromM32

// ELFromM32()
// ===========

(boolean,bits(2)) ELFromM32(bits(5) mode)
// Convert an AArch32 mode encoding to an Exception level.
// Returns (valid,EL):
// 'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid for this implementation
// and the current value of SCR.NS/SCR_EL3.NS.
// 'EL' is the Exception level decoded from 'mode'.
bits(2) el;
boolean valid = !BadMode(mode);  // Check for modes that are not valid for this implementation
case mode of
when M32_Monitor
  el = EL3;
when M32_Hyp
  el = EL2;
when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
  if HaveEL(EL3) && !HaveAArch64() && SCR.NS == '0' then
  el = EL3 else EL1;
when M32_User
  el = EL0;
otherwise
  valid = FALSE;           // Passed an illegal mode value
if valid && el == EL2 && HaveEL(EL3) && SCR_GEN[].NS == '0' then
  valid = FALSE;               // EL2 only valid in Non-secure state in AArch32
elsif valid && HaveRME() && SCR_EL3.<NSE,NS> == '10' then
valid = FALSE;               // No AArch32 modes in Root state
if !valid then el = bits(2) UNKNOWN;
return (valid, el);

shared/functions/system/ELFromSPSR

// ELFromSPSR()
// ============
// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
// 'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
// 'EL' is the Exception level decoded from 'spsr'.

(boolean, bits(2)) ELFromSPSR(bits(N) spsr)

  bits(2) el;
  boolean valid;
  if spsr<4> == '0' then                      // AArch64 state
    el = spsr<3:2>;
    if !HaveAArch64() then                  // No AArch64 support
      valid = FALSE;
    elsif !HaveEL(el) then                  // Exception level not implemented
      valid = FALSE;
    elsif spsr<1> == '1' then               // M[1] must be 0
      valid = FALSE;
    elsif el == EL0 && spsr<0> == '1' then  // for EL0, M[0] must be 0
      valid = FALSE;
    elsif HaveRME() && el == EL3 && SCR_EL3.<NSE,NS> == '10' then
      valid = FALSE;                      // Only EL3 valid in Root state
    elsif el == EL2 && !IsSecureEL2Enabled() && SCR_EL3.NS == '0' then
      valid = FALSE;                      // Unless Secure EL2 is enabled, EL2 valid only in
    else
      valid = TRUE;
    elsif HaveAArch32() then                // AArch32 state
      (valid, el) = ELFromM32(spsr<4:0>);
    else
      valid = FALSE;
  if !valid then el = bits(2) UNKNOWN;
  return (valid, el);

shared/functions/system/ELIsInHost

// ELIsInHost()
// ============

boolean ELIsInHost(bits(2) el)
if !HaveVirtHostExt() || ELUsingAArch32(EL2) then
  return FALSE;
case el of
    when EL3
      return FALSE;
    when EL2
      return EL2Enabled() && HCR_EL2.E2H == '1';
    when EL1
      return FALSE;
    when EL0
      return EL2Enabled() && HCR_EL2.<E2H,TGE> == '11';
    otherwise
      Unreachable();
shared/functions/system/ELStateUsingAArch32

// ELStateUsingAArch32()
// ===============

boolean ELStateUsingAArch32(bits(2) el, boolean secure)
// See ELStateUsingAArch32K() for description. Must only be called in circumstances where
// result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
(known, aarch32) = ELStateUsingAArch32K(el, secure);
assert known;
return aarch32;

shared/functions/system/ELStateUsingAArch32K

// ELStateUsingAArch32K()
// =============

(booleantoolean) ELStateUsingAArch32K(bits(2) el, boolean secure)
// Returns (known, aarch32):
//   'known'   is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
//             using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
//   'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.
if !HaveAArch32EL(el) then
  return (TRUE, FALSE);                      // Exception level is using AArch64
elsif secure && el == EL2 then
  return (TRUE, FALSE);                      // Secure EL2 is using AArch64
elsif !HaveAArch64() then
  return (TRUE, TRUE);                       // Highest Exception level, and therefore all levels
are using AArch32

  // Remainder of function deals with the interprocessing cases when highest Exception level is using
  // AArch64

  booleantoolean aarch32 = booleantoolean UNKNOWN;
  booleantoolean known = booleantoolean TRUE;

  aarch32_below_el3 = HaveEL(EL3) && SCR_EL3.RW == '0' && (!secure || !HaveSecureEL2Ext() ||
  SCR_EL3.EEL2 == '0');
  aarch32_at_el1 = (aarch32_below_el3 || (HaveEL(EL2) &&
  (HaveSecureEL2Ext() && SCR_EL3.EEL2 == '1') || !secure) &&
  HCR_EL2.RW == '0' ||
  !HaveVirtHostExt());
  if el == EL0 && aarch32_at_el1 then // Only know if EL0 using AArch32 from PSTATE
    if PSTATE.EL == EL0 then
      aarch32 = PSTATE.nRW == '1';       // EL0 controlled by PSTATE
    else
      known = FALSE;                     // EL0 state is UNKNOWN
    else
      aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL3,EL0});
  if !known then aarch32 = booleantoolean UNKNOWN;
  return (known, aarch32);

shared/functions/system/ELUsingAArch32

// ELUsingAArch32()
// ===========

boolean ELUsingAArch32(bits(2) el)
return ELStateUsingAArch32(el, IsSecureBelowEL3());
shared/functions/system/ELUsingAArch32K

// ELUsingAArch32K()
// =============

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());

shared/functions/system/EffectiveSCR_EL3_RW

// EffectiveSCR_EL3_RW()
// ===============
// Returns effective SCR_EL3.RW value

bit EffectiveSCR_EL3_RW()
    if !HaveAArch64() then
        return '0';
    if !HaveAArch32EL(EL2) && !HaveAArch32EL(EL1) then
        return '1';
    if HaveAArch32EL(EL1) then
        if !HaveAArch32EL(EL2) && SCR_EL3.NS == '1' then
            return '1';
        if HaveSecureEL2Ext() && SCR_EL3.IEL2 == '1' && SCR_EL3.NS == '0' then
            return '1';
        return SCR_EL3.RW;

shared/functions/system/EffectiveTGE

// EffectiveTGE()
// ==============
// Returns effective TGE value

bit EffectiveTGE()
    if EL2Enabled() then
        return if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE;
    else
        return '0'; // Effective value of TGE is zero

shared/functions/system/EndOfInstruction

// Terminate processing of the current instruction.
EndOfInstruction();

shared/functions/system/EnterLowPowerState

// PE enters a low-power state.
EnterLowPowerState();

shared/functions/system/EventRegister

bits(1) EventRegister;

shared/functions/system/ExceptionalOccurrenceTargetState

enumeration ExceptionalOccurrenceTargetState {
    AArch32_NonDebugState,
    AArch64_NonDebugState,
DebugState

};

**shared/functions/system/FIQPending**

// Returns a tuple indicating if there is any pending physical FIQ
// and if the pending FIQ has superpriority.
(boolean, boolean) FIQPending();

**shared/functions/system/GetAccumulatedFPExceptions**

// Returns FP exceptions accumulated by the PE.
bits(8) GetAccumulatedFPExceptions();

**shared/functions/system/GetLoadStoreType**

// Returns the Load/Store Type. Used when a Translation fault,
// Access Flag fault, or Permission fault generates a Data Abort.
bits(2) GetLoadStoreType();

**shared/functions/system/GetPSRFromPSTATE**

// GetPSRFromPSTATE()
// ==================
// Return a PSR value which represents the current PSTATE

bits(N) GetPSRFromPSTATE(ExceptionalOccurrenceTargetState targetELState, integer N)
if UsingAArch32() && (targetELState IN {AArch32_NonDebugState, DebugState}) then
  assert N == 32;
else
  assert N == 64;
bits(N) spsr = Zeros(N);
spsr<31:28> = PSTATE.<N,Z,C,V>;
if HavePANExt() then spsr<22> = PSTATE.PAN;
spsr<20> = PSTATE.II;
if PSTATE.nRW == '1' then // AArch32 state
  spsr<27> = PSTATE.Q;
spsr<26:25> = PSTATE.IT<1:0>;
if HaveSSBSExt() then spsr<23> = PSTATE.SSBS;
if HaveDITExt() then
  if targetELState == AArch32_NonDebugState then
    spsr<21> = PSTATE.DIT;
  else //AArch64_NonDebugState or DebugState
    spsr<24> = PSTATE.DIT;
if targetELState IN {AArch64_NonDebugState, DebugState} then
  spsr<21> = PSTATE.T;
spsr<19:16> = PSTATE.GE;
spsr<15:10> = PSTATE.IT<7:2>;
spsr<9> = PSTATE.E;
spsr<8:6> = PSTATE.<A,I,F>; // No PSTATE.D in AArch32 state
if HaveFeatNMI() then spsr<13> = PSTATE.ALLINT;
if HaveSSBSExt() then spsr<12> = PSTATE.SSBS;
if HaveBTIExt() then spsr<11:10> = PSTATE.BTYPE;
else
  if HaveMTEExt() then spsr<25> = PSTATE.TCO;
  if HaveDITExt() then spsr<24> = PSTATE.DIT;
if HaveUAOExt() then spsr<23> = PSTATE.UAO;
spsr<21> = PSTATE.SS;
if HaveFeatNMI() then spsr<13> = PSTATE.ALLINT;
if HaveSSBSExt() then spsr<12> = PSTATE.SSBS;
if HaveBTIExt() then spsr<11:10> = PSTATE.BTYPE;
spsr<9:6>   = PSTATE.<D,A,I,F>;
spsr<4>     = PSTATE.nRW;
spsr<3:2>   = PSTATE.EL;
spsr<0>     = PSTATE.SP;
return spsr;

shared/functions/system/HasArchVersion

// HasArchVersion()
// ================
// Returns TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
    return version == ARMv8p0 || boolean IMPLEMENTATION_DEFINED;

shared/functions/system/HaveAArch32

// HaveAArch32()
// =============
// Return TRUE if AArch32 state is supported at at least EL0.

boolean HaveAArch32()
    return boolean IMPLEMENTATION_DEFINED "AArch32 state is supported at at least EL0";

shared/functions/system/HaveAArch32EL

// HaveAArch32EL()
// ===============

boolean HaveAArch32EL(bits(2) el)
    // Return TRUE if Exception level 'el' supports AArch32 in this implementation
    if !HaveEL(el) then
        return FALSE;                    // The Exception level is not implemented
    elsif !HaveAArch32() then
        return FALSE;                    // No Exception level can use AArch32
    elsif !HaveAArch64() then
        return TRUE;                     // All Exception levels are using AArch32
    elsif el == HighestEL() then
        return FALSE;                    // The highest Exception level is using AArch32
    elsif el == EL0 then
        return TRUE;                     // EL0 must support using AArch32 if any AArch32
    return boolean IMPLEMENTATION_DEFINED;

shared/functions/system/HaveAArch64

// HaveAArch64()
// =============
// Return TRUE if the highest Exception level is using AArch64 state.

boolean HaveAArch64()
    return boolean IMPLEMENTATION_DEFINED "Highest EL using AArch64";

shared/functions/system/HaveEL

// HaveEL()
// =========
// Return TRUE if Exception level 'el' is supported
boolean HaveEL(bits(2) el)
  if el IN {EL1,EL0} then
    return TRUE;                             // EL1 and EL0 must exist
  return boolean IMPLEMENTATION_DEFINED;

shared/functions/system/HaveELUsingSecurityState

// HaveELUsingSecurityState()
// ==========================
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

boolean HaveELUsingSecurityState(bits(2) el, boolean secure)
  case el of
    when EL3
      assert secure;
      return HaveEL(EL3);
    when EL2
      if secure then
        return HaveEL(EL2) && HaveSecureEL2Ext();
      else
        return HaveEL(EL2);
      otherwise
        return (HaveEL(EL3) ||
          (secure == boolean IMPLEMENTATION_DEFINED "Secure-only implementation"));

shared/functions/system/HaveFP16Ext

// HaveFP16Ext()
// =============
// Return TRUE if FP16 extension is supported

boolean HaveFP16Ext()
  return boolean IMPLEMENTATION_DEFINED;

shared/functions/system/HighestEL

// HighestEL()
// ===========
// Returns the highest implemented Exception level.

bits(2) HighestEL()
  if HaveEL(EL3) then
    return EL3;
  elsif HaveEL(EL2) then
    return EL2;
  else
    return EL1;

shared/functions/system/Hint_DGH

// Provides a hint to close any gathering occurring within the micro-architecture.
Hint_DGH();
shared/functions/system/Hint_WFE

// Hint_WFE()
// =========
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs or, for WFET, a local
// timeout event is generated when the virtual timer value equals or exceeds the supplied threshold value.

Hint_WFE(integer localtimeout, WFxType wfxtype)
  if IsEventRegisterSet() then
    ClearEventRegister();
  elsif HaveFeatWFxT() && LocalTimeoutEvent(localtimeout) then
    // No further operation if the local timeout has expired.
    EndOfInstruction();
  else
    bits(2) target_el;
    trap = FALSE;
    if PSTATE.EL == EL0 then
      // Check for traps described by the OS which may be EL1 or EL2.
      if HaveTWEDext() then
        sctlr     = SCTLR[];
        trap      = sctlr.nTWE == '0';
        target_el = EL1;
      else
        AArch64.CheckForWFxTrap(EL1, wfxtype);
      if !trap && PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        // Check for traps described by the Hypervisor.
        if HaveTWEDext() then
          trap    = HCR_EL2.TWE == '1';
          target_el = EL2;
        else
          AArch64.CheckForWFxTrap(EL2, wfxtype);
        if !trap && PSTATE.EL != EL3 then
          // Check for traps described by the Secure Monitor.
          if HaveTWEDext() then
            trap    = SCR_EL3.TWE == '1';
            target_el = EL3;
          else
            AArch64.CheckForWFxTrap(EL3, wfxtype);
          if trap && PSTATE.EL != EL3 then
            (delay_enabled, delay) = WFETrapDelay(target_el);    // (If trap delay is enabled, Delay amount)
            if !WaitForEventUntilDelay(delay_enabled, delay) then
              // Event did not arrive before delay expired
              AArch64.WFxTrap(wfxtype, target_el);             // Trap WFE
            else
              WaitForEvent(localtimeout);

shared/functions/system/Hint_WFI

// Hint_WFI()
// =========
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs or, for WFIT, a local timeout event is generated when the virtual timer value equals or exceeds the supplied threshold value.

Hint_WFI(integer localtimeout, WFxType wfxtype)
  if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);
  if InterruptPending() || (HaveFeatWFxT() && LocalTimeoutEvent(localtimeout)) then
    // No further operation if an interrupt is pending or the local timeout has expired.
EndOfInstruction();
else
  if PSTATE.EL == EL0 then
    // Check for traps described by the OS.
    AArch64.CheckForWfxTrap(EL1, wfxtype);
  if PSTATE.EL IN {EL0, EL1} & EL2Enabled() & !IsInHost() then
    // Check for traps described by the Hypervisor.
    AArch64.CheckForWfxTrap(EL2, wfxtype);
  if HaveEL(EL3) & PSTATE.EL != EL3 then
    // Check for traps described by the Secure Monitor.
    AArch64.CheckForWfxTrap(EL3, wfxtype);
  WaitForInterrupt(localtimeout);

shared/functions/system/Hint_Yield

// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance.
Hint_Yield();

shared/functions/system/IRQPending

// Returns a tuple indicating if there is any pending physical IRQ
// and if the pending IRQ has superpriority.
(booleantype) IRQPending();

shared/functions/system/IllegalExceptionReturn

// IllegalExceptionReturn()
// ========================
bool IllegalExceptionReturn(bits(N) spsr)

// Check for illegal return:
//   * To an unimplemented Exception level.
//   * To EL2 in Secure state, when SecureEL2 is not enabled.
//   * To EL0 using AArch64 state, with SPSR.M[0]==1.
//   * To AArch64 state with SPSR.M[1]==1.
//   * To AArch32 state with an illegal value of SPSR.M.
//     (valid, target) = ELFromSPSR(spsr);
//   if !valid then return TRUE;

// Check for return to higher Exception level
if UInt(target) > UInt(PSTATE.EL) then return TRUE;

spsr_mode_is_aarch32 = (spsr<4> == '1');

// Check for illegal return:
//   * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
//     Execution state used in the Exception level being returned to, as determined by
//     the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
//   * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
//     SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
//   * To AArch64 state from AArch32 state (should be caught by above)
(known, target_el_is_aarch32) = ELUsingAArch32K(target);
assert known || (target == EL0 & !ELUsingAArch32K(EL1));
if known & spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

// Check for illegal return from AArch32 to AArch64
if UsingAArch32() & !spsr_mode_is_aarch32 then return TRUE;

// Check for illegal return to EL1 when HCR.TGE is set and when either of
//   * SecureEL2 is enabled.
//   * SecureEL2 is not enabled and EL1 is in Non-secure state.
if HaveEL(EL2) && target == EL1 && HCR_EL2.TGE == '1' then
    if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;
    return FALSE;

shared/functions/system/InstrSet

equation InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};

shared/functions/system/InstructionSynchronizationBarrier

InstructionSynchronizationBarrier();

shared/functions/system/InterruptPending

// InterruptPending()
// ===============
// Returns TRUE if there are any pending physical or virtual
// interrupts, and FALSE otherwise.

boolean InterruptPending()
    boolean pending_virtual_interrupt = FALSE;
    (irq_pending, -) = IRQPending();
    (fiq_pending, -) = FIQPending();
    boolean pending_physical_interrupt = (irq_pending || fiq_pending ||
        IsPhysicalSErrorPending());

    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR_EL2.TGE == '0' then
        boolean virq_pending = HCR_EL2.IMO == '1' && (VirtualIRQPending() || HCR_EL2.VI == '1');
        boolean vfiq_pending = HCR_EL2.FMO == '1' && (VirtualFIQPending() || HCR_EL2.VF == '1');
        boolean vsei_pending = HCR_EL2.AMO == '1' && (IsVirtualSErrorPending() || HCR_EL2.VSE == '1');
        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;

    return pending_physical_interrupt || pending_virtual_interrupt;

shared/functions/system/IsASEInstruction

// Returns TRUE if the current instruction is an ASIMD or SVE vector instruction.

boolean IsASEInstruction();

shared/functions/system/IsCMOWControlledInstruction

// When using AArch64, returns TRUE if the current instruction is one of IC IVAU,
// DC CIVAC, DC CICDVAC, or DC CICVAC.
// When using AArch32, returns TRUE if the current instruction is ICIMVAU or DCCIMVAC.

boolean IsCMOWControlledInstruction();

shared/functions/system/IsCurrentSecurityState

// IsCurrentSecurityState()
// ================
// Returns TRUE if the current Security state matches
// the given Security state, and FALSE otherwise.

boolean IsCurrentSecurityState(SecurityState ss)
    return CurrentSecurityState() == ss;
shared/functions/system/IsEventRegisterSet

// IsEventRegisterSet()
// ===============
// Return TRUE if the Event Register of this PE is set, and FALSE if it is clear.

boolean IsEventRegisterSet()
    return EventRegister == '1';

shared/functions/system/IsHighestEL

// IsHighestEL()
// ===============
// Returns TRUE if given exception level is the highest exception level implemented

boolean IsHighestEL(bits(2) el)
    return HighestEL() == el;

shared/functions/system/IsInHost

// IsInHost()
// ===============

boolean IsInHost()
    return ELIsInHost(PSTATE.EL);

shared/functions/system/IsPhysicalSErrorPending

// Returns TRUE if a physical SError interrupt is pending.

boolean IsPhysicalSErrorPending();

shared/functions/system/IsSErrorEdgeTriggered

// IsSErrorEdgeTriggered()
// ===============
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.

boolean IsSErrorEdgeTriggered(bits(2) target_el, bits(25) syndrome)
    if HaveRASExt() then
        if HaveDoubleFaultExt() then
            return TRUE;
        if ELUsingAArch32(target_el) then
            if syndrome<11:10> != '00' then
                // AArch32 and not Uncontainable.
                return TRUE;
            else
                if syndrome<24> == '0' & syndrome<5:0> != '000000' then
                    // AArch64 and neither IMPLEMENTATION DEFINED syndrome nor Uncategorized.
                    return TRUE;
                return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";
        else
            return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";
    return FALSE;

shared/functions/system/IsSecure

// IsSecure()
// ===============
// Returns TRUE if current Exception level is in Secure state.
boolean IsSecure()
   if HaveEL(EL3) && !UsingAArch32() && PSTATE.EL == EL3 then
      return TRUE;
   elsif HaveEL(EL3) && UsingAArch32() && PSTATE.M == M32_Monitor then
      return TRUE;
   return IsSecureBelowEL3();

shared/functions/system/IsSecureBelowEL3

// IsSecureBelowEL3()
// ===============
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// Differs from IsSecure in that it ignores the current EL or Mode
// in considering security state.
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

boolean IsSecureBelowEL3()
   if HaveEL(EL3) then
      return SCR_GEN[].NS == '0';
   elsif HaveEL(EL2) && (!HaveSecureEL2Ext() || !HaveAArch64()) then
      // If Secure EL2 is not an architecture option then we must be Non-secure.
      return FALSE;
   else
      // TRUE if processor is Secure or FALSE if Non-secure.
      return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";

shared/functions/system/IsSecureEL2Enabled

// IsSecureEL2Enabled()
// ===============
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

boolean IsSecureEL2Enabled()
   if HaveEL(EL2) && HaveSecureEL2Ext() then
      if HaveEL(EL3) then
         if !ELUsingAArch32(EL3) && SCR_EL3.EEL2 == '1' then
            return TRUE;
         else
            return FALSE;
      else
         return SecureOnlyImplementation();
      else
         return FALSE;
   else
      return FALSE;

shared/functions/system/IsSynchronizablePhysicalSErrorPending

// Returns TRUE if a synchronizable physical SError interrupt is pending.
boolean IsSynchronizablePhysicalSErrorPending();

shared/functions/system/IsVirtualSErrorPending

// Returns TRUE if a virtual SError interrupt is pending.
boolean IsVirtualSErrorPending();
shared/functions/system/LocalTimeoutEvent

    // Returns TRUE if CNTVCT_EL0 equals or exceeds the localtimeout value.
    boolean LocalTimeoutEvent(integer localtimeout);

shared/functions/system/Mode_Bits

    constant bits(5) M32_User    = '10000';
    constant bits(5) M32_FIQ     = '10001';
    constant bits(5) M32_IRQ     = '10010';
    constant bits(5) M32_Svc     = '10011';
    constant bits(5) M32_Monitor = '10110';
    constant bits(5) M32_Abt     = '10111';
    constant bits(5) M32_Hyp     = '11010';
    constant bits(5) M32_Undef   = '11011';
    constant bits(5) M32_System  = '11111';

shared/functions/system/NonSecureOnlyImplementation

    // NonSecureOnlyImplementation()
    // =====================================
    // Returns TRUE if the security state is always Non-secure for this implementation.

    boolean NonSecureOnlyImplementation()
        return boolean IMPLEMENTATION_DEFINED "Non-secure only implementation";

shared/functions/system/PLOfEL

    // PLOfEL()
    // ========

    PrivilegeLevel PLOfEL(bits(2) el)
        case el of
            when EL3  return if !HaveAArch64() then PL1 else PL3;
            when EL2  return PL2;
            when EL1  return PL1;
            when EL0  return PL0;

shared/functions/system/PSTATE

    ProcState PSTATE;

shared/functions/system/PhysicalCountInt

    // PhysicalCountInt()
    // --------------------
    // Returns the integral part of physical count value of the System counter.

    bits(64) PhysicalCountInt()
        return PhysicalCount<87:24>;

shared/functions/system/PrivilegeLevel

    enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
shared/functions/system/ProcState

type ProcState is (
  bits (1) N,        // Negative condition flag
  bits (1) Z,        // Zero condition flag
  bits (1) C,        // Carry condition flag
  bits (1) V,        // Overflow condition flag
  bits (1) D,        // Debug mask bit                     [AArch64 only]
  bits (1) A,        // SError interrupt mask bit
  bits (1) I,        // IRQ mask bit
  bits (1) F,        // FIQ mask bit
  bits (1) PAN,      // Privileged Access Never Bit   [v8.1]
  bits (1) UAO,      // User Access Override [v8.2]
  bits (1) DIT,      // Data Independent Timing [v8.4]
  bits (1) TCO,      // Tag Check Override [v8.5, AArch64 only]
  bits (2) BTYPE,    // Branch Type                      [v8.5]
  bits (1) ZA,       // Accumulation array enabled [SME]
  bits (1) SM,       // Streaming SVE mode enabled  [SME]
  bits (1) ALLINT,   // Interrupt mask bit
  bits (1) SS,       // Software step bit
  bits (1) IL,       // Illegal Execution state bit
  bits (2) EL,       // Exception level
  bits (1) nRW,      // not Register Width: 0=64, 1=32
  bits (1) SP,       // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
  bits (1) Q,        // Cumulative saturation flag [AArch32 only]
  bits (4) GE,       // Greater than or Equal flags [AArch32 only]
  bits (1) SSBS,     // Speculative Store Bypass Safe
  bits (8) IT,       // If-then bits, RES0 in CPSR [AArch32 only]
  bits (1) J,        // J bit, RES0 [AArch32 only, RES0 in SPSR and CPSR]
  bits (1) T,        // T32 bit, RES0 in CPSR [AArch32 only]
  bits (1) E,        // Endianness bit [AArch32 only]
  bits (5) M         // Mode field [AArch32 only]
)

shared/functions/system/RestoredITBits

// RestoredITBits()
// ================
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(N) spsr)
  it = spsr<15:10,26:25>;
  // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
  // to zero or copied from the SPSR.
  if PSTATE.IL == '1' then
    if ConstrainUnpredictableBool(Unpredictable_ILZEROIT) then return '00000000';
    else return it;
  // The IT bits are forced to zero when they are set to a reserved value.
  if !IsZero(it<7:4>) && !IsZero(it<3:0>) then
    return '00000000';
  // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
  // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
  ifd = if PSTATE.EL == EL2 then HSCTRL.ITD else SCTLR.ITD;
  if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
    return '00000000';
  else
    return it;
shared/functions/system/SCRType

type SCRType;

shared/functions/system/SCR_GEN

// SCR_GEN[
// =========

SCRType SCR_GEN[];  
// AArch32 secure & AArch64 EL3 registers are not architecturally mapped
assert HaveEL(EL3);
bits(64) r;
if !HaveAArch64() then
  r = ZeroExtend(SCR, 64);
else
  r = SCR_EL3;
return r;

shared/functions/system/SafeOnlyImplementation

// SecureOnlyImplementation()
// ==============
// Returns TRUE if the security state is always Secure for this implementation.
boolean SecureOnlyImplementation()
  return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";

shared/functions/system/SecurityState

enumeration SecurityState {
  SS_NonSecure,
  SS_Root,
  SS_Realm,
  SS_Secure
};

shared/functions/system/SecurityStateAtEL

// SecurityStateAtEL()
// ===============
// Returns the effective security state at the exception level based off current settings.
SecurityState SecurityStateAtEL(bits(2) EL)
if HaveRME() then
  if EL == EL3 then return SS_Root;
  case SCR_EL3.<NSE, NS> of
    when '00' return SS_Secure;
    when '01' return SS_NonSecure;
    when '11' return SS_Realm;
  if !HaveEL(EL3) then
    if SecureOnlyImplementation() then
      return SS_Secure;
    else
      return SS_NonSecure;
  elseif EL == EL3 then
    return SS_Secure;
  else
    // For EL2 call only when EL2 is enabled in current security state
    assert(EL != EL2 || EL2Enabled());
    if !ELUsingAArch32(EL3) then
shared/functions/system/SendEvent

// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// when a PE executes the SEV instruction, it causes this function to be executed.
SendEvent();

shared/functions/system/SendEventLocal

// SendEventLocal()
// ================
// Set the local Event Register of this PE.
// when a PE executes the SEVL instruction, it causes this function to be executed.
SendEventLocal()
    EventRegister = '1';
    return;

shared/functions/system/SetAccumulatedFPExceptions

// Stores FP Exceptions accumulated by the PE.
SetAccumulatedFPExceptions(bits(8) accumulated_exceptions);

shared/functions/system/SetPSTATEFromPSR

// SetPSTATEFromPSR()
// ================
SetPSTATEFromPSR(bits(N) spsr)
    boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);

    // SetPSTATEFromPSR()
    // ===============
    // Set PSTATE based on a PSR value
SetPSTATEFromPSR(bits(N) spsr_in, boolean illegal_psr_state)
    bits(N) spsr = spsr_in;
    boolean from_aarch64 = !UsingAArch32();
    assert N == (if from_aarch64 then 64 else 32);
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    ShouldAdvanceSS = FALSE;
    if illegal_psr_state then
        PSTATE.IL = '1';
        if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
        if HaveBTIExt() then PSTATE.BTYPE = bits(2) UNKNOWN;
        if HaveUAOExt() then PSTATE.UAO = bit UNKNOWN;
        if HaveDITExt() then PSTATE.DIT = bit UNKNOWN;
        if HaveMTEExt() then PSTATE.TCO = bit UNKNOWN;
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then // AArch32
            AArch32.WriteMode(spsr<4:0>); // Sets PSTATE.EL correctly
            if HaveSSBSExt() then PSTATE.SSBS = spsr<23>;
        else // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL = spsr<3:2>;

        return if SCR_EL3.NS == '1' then SS_NonSecure else SS_Secure;
    else
        return if SCR.NS == '1' then SS_NonSecure else SS_Secure;

PSTATE.SP = spsr<0>;  
if HaveBTIExt() then PSTATE.BTYPE = spsr<11:10>; 
if HaveSSBSExt() then PSTATE.SSBS = spsr<12>; 
if HaveUAOExt() then PSTATE.UAO = spsr<23>; 
if HaveDITExt() then PSTATE.DIT = spsr<24>; 
if HaveMTEExt() then PSTATE.TCO = spsr<25>; 

// If PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the T bit is set to zero or 
// copied from SPSR. 
if PSTATE.IL == '1' && PSTATE.nRW == '1' then 
    if ConstrainUnpredictableBool(Unpredictable_ILZEROT) then spsr<5> = '0';

// State that is reinstated regardless of illegal exception return 
PSTATE.<N,Z,C,V> = spsr<31:28>; 
if HavePANExt() then PSTATE.PAN = spsr<22>; 
if PSTATE.nRW == '1' then                     // AArch32 state 
    PSTATE.Q = spsr<27>;                     // AArch32 state 
    PSTATE.IT = RestoredITBits(spsr); 
    ShouldAdvanceIT = FALSE; 
    if HaveDITExt() then PSTATE.DIT = (if (Restarting() || from_aarch64) then spsr<24> else 
        spsr<21>); 
    PSTATE.GE = spsr<19:16>; 
    PSTATE.E = spsr<9>;                      // No PSTATE.D in AArch32 state 
    PSTATE.<A,I,F> = spsr<8:6>;              // PSTATE.D in AArch32 state 
    PSTATE.T = spsr<5>;                       // PSTATE.J is RES0 
else                                          // AArch64 state 
    if HaveFeatNMI() then PSTATE.ALLINT = spsr<13>; 
    PSTATE.<D,A,I,F> = spsr<9:6>;            // No PSTATE.<Q,IT,GE,E,T> in AArch64 state 
return;

shared/functions/system/ShouldAdvanceIT

boolean ShouldAdvanceIT;

shared/functions/system/ShouldAdvanceSS

boolean ShouldAdvanceSS;

shared/functions/system/SpeculationBarrier

SpeculationBarrier();

shared/functions/system/SynchronizeContext

SynchronizeContext();

shared/functions/system/SynchronizeErrors

// Implements the error synchronization event. 
SynchronizeErrors();

shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts

// Take any pending unmasked physical SError interrupt. 
TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);
shared/functions/system/TakeUnmaskedSErrorInterrupts

// Take any pending unmasked physical SError interrupt or unmasked virtual SError interrupt.
TakeUnmaskedSErrorInterrupts();

shared/functions/system/ThisInstr

bits(32) ThisInstr();

shared/functions/system/ThisInstrLength

integer ThisInstrLength();

shared/functions/system/Unreachable

Unreachable()
    assert FALSE;

shared/functions/system/UsingAArch32

// UsingAArch32()
// ==============
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.

boolean UsingAArch32()
    boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAArch32() then assert !aarch32;
    if !HaveAArch64() then assert aarch32;
    return aarch32;

shared/functions/system/VirtualFIQPending

// Returns TRUE if there is any pending virtual FIQ.
boolean VirtualFIQPending();

shared/functions/system/VirtualIRQPending

// Returns TRUE if there is any pending virtual IRQ.
boolean VirtualIRQPending();

shared/functions/system/WFxType

enumeration WFxType {WFxType_WFE, WFxType_WFI, WFxType_WFET, WFxType_WFIT};

shared/functions/system/WaitForEvent

// WaitForEvent()
// ==============
// PE optionally suspends execution until one of the following occurs:
// - A WFE wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Event with Timeout (WFET) is executing, and a local timeout event occurs


// It is IMPLEMENTATION DEFINED whether restarting execution after the period of
// suspension causes the Event Register to be cleared.

WaitForEvent(integer localtimeout)
if !(IsEventRegisterSet() || (HaveFeatWFxT() && LocalTimeoutEvent(localtimeout))) then
    EnterLowPowerState();
return;

shared/functions/system/WaitForInterrupt

// WaitForInterrupt()
// ==================
// PE optionally suspends execution until one of the following occurs:
// - A WFI wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Interrupt with Timeout (WFT) is executing, and a local timeout event occurs.

WaitForInterrupt(integer localtimeout)
if !(HaveFeatWFxT() && LocalTimeoutEvent(localtimeout)) then
    EnterLowPowerState();
return;

shared/functions/unpredictable/ConstrainUnpredictable

// Return the appropriate Constraint result to control the caller's behavior.
// The return value is IMPLEMENTATION DEFINED within a permitted list for each
// UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)
Constraint ConstrainUnpredictable(Unpredictable which);

shared/functions/unpredictable/ConstrainUnpredictableBits

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONstrained.
(Constraint,bits(width)) ConstrainUnpredictableBits(Unpredictable which, integer width);

shared/functions/unpredictable/ConstrainUnpredictableBool

// This is a variant of the ConstrainUnpredictable function where the result is either
// Constraint_TRUE or Constraint_FALSE.
boolean ConstrainUnpredictableBool(Unpredictable which);

shared/functions/unpredictable/ConstrainUnpredictableInteger

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns an UNKNOWN
// value in the range low to high, inclusive.
(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high,
    Unpredictable which);
shared/functions/unpredictable/ConstrainUnpredictableProcedure

// This is a variant of ConstrainUnpredictable that implements a Constrained Unpredictable behavior for a given Unpredictable situation. The behavior is within permitted behaviors for a given Unpredictable situation, these are documented in the textual part of the architecture specification. // This function is expected to be refined in an IMPLEMENTATION DEFINED manner. The details of possible outcomes may not be present in the code and must be interpreted for each use with respect to the CONSTRAINED UNPREDICTABLE specifications for the specific area.

ConstrainUnpredictableProcedure(Unpredictable which);

shared/functions/unpredictable/Constraint

enumeration Constraint { // General
Constraint_NONE, // Instruction executes with no change or side-effect to its described behavior
Constraint_UNKNOWN, // Destination register has UNKNOWN value
Constraint_UNDEF, // Instruction is UNDEFINED
Constraint_UNDEFEL0, // Instruction is UNDEFINED at EL0 only
Constraint_NOP, // Instruction executes as NOP
Constraint_TRUE,
Constraint_FALSE,
Constraint_DISABLED,
Constraint_UNCOND, // Instruction executes unconditionally
Constraint_COND, // Instruction executes conditionally
Constraint_ADDITIONAL_DECODE, // Instruction executes with additional decode
// Load-store
Constraint_WBSUPPRESS,
Constraint_FAULT,
Constraint_LIMTED_ATOMICITY, // Accesses are not single-copy atomic above the byte level
Constraint_NVNV1_00,
Constraint_NVNV1_01,
Constraint_NVNV1_11,
Constraint_EL1TIMESTAMP, // Constrain to Virtual Timestamp
Constraint_EL2TIMESTAMP, // Constrain to Virtual Timestamp
Constraint_OSH, // Constrain to Outer Shareable
Constraint_ISH, // Constrain to Inner Shareable
Constraint_NSH, // Constrain to Nonshareable
Constraint_NC, // Constrain to Noncacheable
Constraint_WT, // Constrain to Writethrough
Constraint_WB, // Constrain to Writeback
// IPA too large
Constraint_FORCE, Constraint_FORCENOSLCHECK,
// PMSCR_PCT reserved values select Virtual timestamp
Constraint_PMSCR_PCT_VIRT};

shared/functions/unpredictable/Unpredictable

enumeration Unpredictable {
// VMSR on MVFR
Unpredictable_VMSR,
// Writeback/transfer register overlap (load)
Unpredictable_WBOVERLAPLD,
// Writeback/transfer register overlap (store)
Unpredictable_WBOVERLAPST,
// Load Pair transfer register overlap
Unpredictable_LDPOVERLAP,
// Store-exclusive base/status register overlap
Unpredictable_BASEOVERLAP,
// Store-exclusive data/status register overlap
Unpredictable_DATAOVERLAP,
// Load-store alignment checks
Unpredictable_DEVPAGE2,
// Instruction fetch from Device memory
Unpredictable_INSTRDEVICE,
// Reserved CPACR value
Unpredictable_RESCPACR,
// Reserved MAIR value
Unpredictable_RESMAIR,
// Effect of SCTLR_ELx.C on Tagged attribute
Unpredictable_S1CTAGGED,
// Reserved Stage 2 MemAttr value
Unpredictable_SZRESMEMATTR,
// Reserved TEX:C:B value
Unpredictable_RESTEXCB,
// Reserved PRR value
Unpredictable_RESPRRR,
// Reserved DACR field
Unpredictable_RESDACR,
// Reserved VTCR.S value
Unpredictable_RESVTCRS,
// Reserved TCR.TnSZ value
Unpredictable_RESTnSZ,
// Reserved SCTLR_ELx.TCF value
Unpredictable_RESTCF,
// Tag stored to Device memory
Unpredictable_DEVICEETAGSTORE,
// Out-of-range TCR.TnSZ value
Unpredictable_OORTnSZ,
// IPA size exceeds PA size
Unpredictable_LARGEIPA,
// Syndrome for a known-passing conditional A32 instruction
Unpredictable_ESRCONDPASS,
// Illegal State exception: zero PSTATE.IT
Unpredictable_ILZEROIT,
// Illegal State exception: zero PSTATE.T
Unpredictable_ILZEROT,
// Debug: prioritization of Vector Catch
Unpredictable_BPVECTORCATCHPRI,
// Debug Vector Catch: match on 2nd halfword
Unpredictable_VCMATCHHALF,
// Debug Vector Catch: match on Data Abort
// or Prefetch abort
Unpredictable_VCOMATCHDAPA,
// Debug watchpoints: non-zero MASK and non-ones BAS
Unpredictable_WPMASKANDBAS,
// Debug watchpoints: non-contiguous BAS
Unpredictable_WPBASCNTIGUOUS,
// Debug watchpoints: reserved MASK
Unpredictable_RESNPMASK,
// Debug watchpoints: non-zero MASKed bits of address
Unpredictable_WPMASKEDBIT,
// Debug breakpoints and watchpoints: reserved control bits
Unpredictable_RESBPPWCTRL,
// Debug breakpoints: not implemented
Unpredictable_BPNOTIMPL,
// Debug breakpoints: reserved type
Unpredictable_RESBPTYPE,
// Debug breakpoints: not-context-aware breakpoint
Unpredictable_BPNOTCTXCMP,
// Debug breakpoints: match on 2nd halfword of instruction
Unpredictable_BPMATCHHALF,
// Debug breakpoints: mismatch on 2nd halfword of instruction
Unpredictable_BPMESMATCHHALF,
  // Debug: restart to a misaligned AArch32 PC value
Unpredictable_RESTARTALIGNPC,
  // Debug: restart to a not-zero-extended AArch32 PC value
Unpredictable_RESTARTZEROUPPERPC,
  // Zero top 32 bits of X registers in AArch32 state
Unpredictable_ZEROUPPER,
  // Zero top 32 bits of PC on illegal return to
  // AArch32 state
Unpredictable_ERETZEROUPPERPC,
  // Force address to be aligned when interworking
  // branch to A32 state
Unpredictable_A32FORCEALIGNPC,
  // SMC disabled
Unpredictable_SMD,
  // FF speculation
Unpredictable_NONFAULT,
  // Zero top bits of Z registers in EL change
Unpredictable_SVEZEROUPPER,
  // Load mem data in NF loads
Unpredictable_SVELDNFDATA,
  // Write zeros in NF loads
Unpredictable_SVELDNFZERO,
  // SP alignment fault when predicate is all zero
Unpredictable_CHECKSPNONEACTIVE,
  // Zero top bits of ZA registers in EL change
Unpredictable_SMEZEROUPPER,
  // HCR_EL2.<NV,NV1> == '01'
Unpredictable_NNV1,
  // Reserved shareability encoding
Unpredictable_Shareability,
  // Access Flag Update by HW
Unpredictable_AFUPDATE,
  // Dirty Bit State Update by HW
Unpredictable_DBGxVR_RESS,
  // Consider SCTLR[].IESB in Debug state
Unpredictable_IESBinDebug,
  // Bad settings for PMSCFR_EL1/PMSEVR_EL1/PMSELATFR_EL1
Unpredictable_BADPMSFCR,
  // Zero saved BType value in SPSR_ELx/DPSR_EL0
Unpredictable_ZEROBTYPE,
  // Timestamp constrained to virtual or physical
Unpredictable_EL2TIMESTAMP,
Unpredictable_EL1TIMESTAMP,
  // Reserved MDCR_EL3.<NSTBE,NSTB> or MDCR_EL3.<NSPBE,NSPB> value
Unpredictable_RESERVEDNSxB,
  // WFET or WFIT instruction in Debug state
Unpredictable_WFxTDEBUG,
  // Address does not support LS64 instructions
Unpredictable_LS64UNSUPPORTED,
  // Misaligned exclusives, atomic, acquire/release
  // to region that is not Normal Cacheable WB
Unpredictable_MISALIGNEDATOMIC,
  // Clearing DCC/ITR sticky flags when instruction is in flight
Unpredictable_CLEARERRITEZERO,
  // ALUexceptionreturn when in user/system mode in
  // A32 instructions
Unpredictable_ALU EXCEPTIONRETURN,
  // Trap to register in debug state are ignored
Unpredictable_IGNORETRAPINDEBUG,
  // Compare DBGxVR.RESS for BP/WP
Unpredictable_DBGxVR_RESS,
  // Inaccessible event counter
Unpredictable_PMUEVENTCOUNTER,
  // Reserved PMSCR.PCT behaviour.
Unpredictable_PMSCR_PCT,
  // Generate BRB_FILTRATE event on BRB injection
Unpredictable_BRBFILTRATE,
// Operands for CPY*/SET* instructions overlap or
// use 0b11111 as a register specifier
Unpredictable_MPSOVERLAP31

};

shared/functions/vector/AdvSIMDExpandImm

// AdvSIMDExpandImm()
// ==================

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
bits(64) imm64;
case cmode<3:1> of
  when '000'
    imm64 = Replicate(Zeros(24):imm8, 2);
  when '001'
    imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
  when '010'
    imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
  when '011'
    imm64 = Replicate(imm8:Zeros(24), 2);
  when '100'
    imm64 = Replicate(Zeros(8):imm8, 4);
  when '101'
    imm64 = Replicate(imm8:Zeros(8), 4);
  when '110'
    if cmode<0> == '0' then
      imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
    else
      imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
  when '111'
    if cmode<0> == '0' & op == '0' then
      imm64 = Replicate(imm8, 8);
    if cmode<0> == '0' & op == '1' then
      imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
      imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
      imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
      imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
    if cmode<0> == '1' & op == '0' then
      imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
    imm64 = Replicate(imm32, 2);
    if cmode<0> == '1' & op == '1' then
      if UsingAArch32() then ReservedEncoding();
      imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,8):imm8<5:0>:Zeros(48);
return imm64;

shared/functions/vector/MatMulAdd

// MatMulAdd()
// ===========
// // Signed or unsigned 8-bit integer matrix multiply and add to 32-bit integer matrix
// // result[2, 2] = addend[2, 2] + (op1[2, 8] * op2[8, 2])

bits(N) MatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, boolean op1_unsigned, boolean op2_unsigned)
assert N == 128;
bits(N) result;
bias(32) sum;
integer prod;
for i = 0 to 1
for j = 0 to 1
    sum = Elem[addend, 2*i + j, 32];
    for k = 0 to 7
        prod = Int(Elem[op1, 8*i + k, 8], op1_unsigned) * Int(Elem[op2, 8*j + k, 8], op2_unsigned);
        sum = sum + prod;
    Elem[result, 2*i + j, 32] = sum;
return result;

shared/functions/vector/PolynomialMult

// PolynomialMult()
// ================
bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;

shared/functions/vector/SatQ

// SatQ()
// ======
(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);

shared/functions/vector/SignedSatQ

// SignedSatQ()
// ============
(bits(N), boolean) SignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);

shared/functions/vector/UnsignedRSqrtEstimate

// UnsignedRSqrtEstimate()
// =======================
bits(N) UnsignedRSqrtEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    if operand<N-1:N-2> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)
        // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
increasedprecision = FALSE;
estimate = RecipSqrtEstimate(UInt(operand<31:23>), increasedprecision);
// result is in the range 0x80000000 .. 0xffffffff representing [1.0 .. 2.0]
result = estimate<8:0> : Zeros(N-9);
return result;

shared/functions/vector/UnsignedRecipEstimate

// UnsignedRecipEstimate()
// ..............................

bits(N) UnsignedRecipEstimate(bits(N) operand)
assert N == 32;
bits(N) result;
if operand<N-1> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
    result = Ones(N);
else  // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)
    increasedprecision = FALSE;
estimate = RecipEstimate(UInt(operand<31:23>), increasedprecision);
// result is in the range 0x80000000 .. 0xffffffff representing [1.0 .. 2.0]
result = estimate<8:0> : Zeros(N-9);
return result;

shared/functions/vector/UnsignedSatQ

// UnsignedSatQ()
// ..............................

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
integer result;
boolean saturated;
if i > 2^N - 1 then
    result = 2^N - 1;  saturated = TRUE;
elsif i < 0 then
    result = 0;  saturated = TRUE;
else
    result = i;  saturated = FALSE;
return (result<N-1:0>, saturated);

J1.3.4 shared/trace

This section includes the following pseudocode functions:
• shared/trace/Common/GetTimestamp on page J1-11529.
• shared/trace/Common/PhysicalOffsetIsValid on page J1-11529.
• shared/trace/TraceBuffer/TraceBufferEnabled on page J1-11529.
• shared/trace/TraceBuffer/TraceBufferOwner on page J1-11530.
• shared/trace/TraceBuffer/TraceBufferRunning on page J1-11530.
• shared/trace/selfhosted/EffectiveE0HTRE on page J1-11530.
• shared/trace/selfhosted/EffectiveE0TRE on page J1-11530.
• shared/trace/selfhosted/EffectiveE1TRE on page J1-11531.
• shared/trace/selfhosted/EffectiveE2TRE on page J1-11531.
• shared/trace/selfhosted/SelfHostedTraceEnabled on page J1-11531.
• shared/trace/selfhosted/TraceAllowed on page J1-11531.
shared/trace/Common/GetTimestamp

// GetTimestamp()
// ==============
// Returns the Timestamp depending on the type

bits(64) GetTimestamp(TimeStamp timeStampType)
case timeStampType of
  when TimeStamp_Physical
    return PhysicalCountInt();
  when TimeStamp_Virtual
    return PhysicalCountInt() - CNTVOFF_EL2;
  when TimeStamp_OffsetPhysical
    bits(64) physoff = if PhysicalOffsetIsValid() then CNTPOFF_EL2 else Zeros();
    return PhysicalCountInt() - physoff;
  when TimeStamp_None
    return Zeros(64);
  when TimeStamp_CoreSight
    return bits(64) IMPLEMENTATION_DEFINED "CoreSight timestamp";
  otherwise
    Unreachable();

shared/trace/Common/PhysicalOffsetIsValid

// PhysicalOffsetIsValid()
// ================
// Returns whether the Physical offset for the timestamp is valid

boolean PhysicalOffsetIsValid()
  if !HaveAArch64() then
    return FALSE;
  elsif !HaveEL(EL2) || !HaveECVExt() then
    return FALSE;
  elsif HaveEL(EL3) && SCR_EL3:NS == '1' && EffectiveSCR_EL3_RW() == '0' then
    return FALSE;
  elsif HaveEL(EL3) && SCR_EL3:ECVEn == '0' then
    return FALSE;
  elsif CNTHCTL_EL2:ECV == '0' then
    return FALSE;
  else
    return TRUE;

shared/trace/TraceBuffer/TraceBufferEnabled

// TraceBufferEnabled()
// ===============

boolean TraceBufferEnabled()
  if !HaveTraceBufferExtension() || TRBLIMITR_EL1:E == '0' then
    return FALSE;
  if !SelfHostedTraceEnabled() then
    return FALSE;
  (-, el) = TraceBufferOwner();
  return !ELUsingAArch32(el);
shared/trace/TraceBufferOwner

// TraceBufferOwner()
// ==================
// Return the owning Security state and Exception level. Must only be called
// when SelfHostedTraceEnabled() is TRUE.

(SecurityState, bits(2)) TraceBufferOwner()
assert HaveTraceBufferExtension() && SelfHostedTraceEnabled();

SecurityState owning_ss;
if HaveEL(EL3) then
  bits(3) state_bits;
  if HaveRME() then
    state_bits = MDCR_EL3.<NSTBE,NSTB>;
    if state_bits IN {'10x'} then
      // Reserved value
      state_bits = ConstrainUnpredictableBits(Unpredictable_RESERVEDNSxB, 3);
    else
      state_bits = '0' : MDCR_EL3.NSTB;
  case state_bits of
    when '00x' owning_ss = SS_Secure;
    when '01x' owning_ss = SS_NonSecure;
    when '11x' owning_ss = SS_Realm;
  else
    owning_ss = if SecureOnlyImplementation() then SS_Secure else SS_NonSecure;
    bits(2) owning_el;
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
      owning_el = if MDCR_EL2.E2TB == '00' then EL2 else EL1;
    else
      owning_el = EL1;
    return (owning_ss, owning_el);

shared/trace/TraceBufferRunning

// TraceBufferRunning()
// ====================

boolean TraceBufferRunning()
return TraceBufferEnabled() && TRBSR_EL1.S == '0';

shared/trace/selfhosted/EffectiveE0HTRE

// EffectiveE0HTRE()
// =================
// Returns effective E0HTRE value

bit EffectiveE0HTRE()
return if ELUsingAArch32(EL2) then HTRFCR.E0HTRE else TRFCR_EL2.E0HTRE;

shared/trace/selfhosted/EffectiveE0TRE

// EffectiveE0TRE()
// ================
// Returns effective E0TRE value

bit EffectiveE0TRE()
return if ELUsingAArch32(EL1) then TRFCR.E0TRE else TRFCR_EL1.E0TRE;
shared/trace/selfhosted/EffectiveE1TRE

// EffectiveE1TRE()
// ================
// Returns effective E1TRE value

bit EffectiveE1TRE()
return if UsingAArch32() then TRFCR.E1TRE else TRFCR_EL1.E1TRE;

shared/trace/selfhosted/EffectiveE2TRE

// EffectiveE2TRE()
// ================
// Returns effective E2TRE value

bit EffectiveE2TRE()
return if UsingAArch32() then HTRFCR.E2TRE else TRFCR_EL2.E2TRE;

shared/trace/selfhosted/SelfHostedTraceEnabled

// SelfHostedTraceEnabled()
// ========================
// Returns TRUE if Self-hosted Trace is enabled.

boolean SelfHostedTraceEnabled()
if !(HaveTraceExt() && HaveSelfHostedTrace()) then return FALSE;
if EDSCR.TFO == '0' then return TRUE;
if HaveEL(EL3) then
secure_trace_enable = if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE;
if secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebugEnabled() then
return TRUE;
if (HaveRME() && MDCR_EL3.RLTE == '1' && !ExternalRealmNoninvasiveDebugEnabled()) then
return TRUE;
else
if SecureOnlyImplementation() && !ExternalSecureNoninvasiveDebugEnabled() then
return TRUE;
return FALSE;

shared/trace/selfhosted/TraceAllowed

// TraceAllowed()
// ==============
// Returns TRUE if Self-hosted Trace is allowed in the given Exception level.

boolean TraceAllowed(bits(2) el)
if !HaveTraceExt() then return FALSE;
if SelfHostedTraceEnabled() then
boolean trace_allowed;
ss = SecurityStateAtEL(el);
// Detect scenarios where tracing in this Security state is never allowed.
case ss of
when SS_NonSecure
trace_allowed = TRUE;
when SS_Secure
bit trace_bit;
if HaveEL(EL3) then
trace_bit = if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE;
else
trace_bit = '1';
trace_allowed = trace_bit == '1';
when SS_Realm

trace_allowed = MDCR_EL3.RLTE == '1';
when SS_Root
trace_allowed = FALSE;

// Tracing is prohibited if the trace buffer owning security state is not the
// current Security state or the owning Exception level is a lower Exception level.
if HaveTraceBufferExtension() && TraceBufferEnabled() then
    (owning_ss, owning_el) = TraceBufferOwner();
if (ss != owning_ss || Uint(owning_el) < Uint(el) ||
    (EffectiveTGE() == '1' && owning_el == EL1)) then
    trace_allowed = FALSE;

bit TRE_bit;
case el of
  when EL3  TRE_bit = if !HaveAArch64() then TRFCR.E1TRE else '0';
  when EL2  TRE_bit = EffectiveE2TRE();
  when EL1  TRE_bit = EffectiveE1TRE();
  when EL0
    if EffectiveTGE() == '1' then
      TRE_bit = EffectiveE0HTRE();
    else
      TRE_bit = EffectiveE0TRE();
return trace_allowed && TRE_bit == '1';
else
return ExternalNoninvasiveDebugAllowed(el);

shared/trace/selfhosted/TraceContextIDR2
// TraceContextIDR2()
// ===============

boolean TraceContextIDR2()
if !TraceAllowed(PSTATE.EL)|| !HaveEL(EL2) then return FALSE;
return (!SelfHostedTraceEnabled() || TRFCR_EL2.CX == '1');

shared/trace/selfhosted/TraceSynchronizationBarrier
// Memory barrier instruction that preserves the relative order of memory accesses to System
// registers due to trace operations and other memory accesses to the same registers
TraceSynchronizationBarrier();

shared/trace/selfhosted/TraceTimeStamp
// TraceTimeStamp()
// ================

TimeStamp TraceTimeStamp()
if SelfHostedTraceEnabled() then
  if HaveEL(EL2) then
    TS_el2 = TRFCR_EL2.TS;
    if !HaveECVExt() && TS_el2 == '10' then
      // Reserved value
      (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP, 2);
    case TS_el2 of
      when '00'
        // Falls out to check TRFCR_EL1.TS
      when '01'
        return TimeStamp_Virtual;
      when '10'
        assert HaveECVExt(); // Otherwise ConstrainUnpredictableBits removes this case
        return TimeStamp_OffsetPhysical;
  else
    return time stamps of TRFCR_EL0.TS,
    if !HaveECVExt() && TRFCR_EL0.CX == '10' then
      // Reserved value
      (-, TRFCR_EL0.TS) = ConstrainUnpredictableBits(Unpredictable_EL0TIMESTAMP, 2);
    case TRFCR_EL0.TS of
      when '00'
        // Falls out to check TRFCR_EL1.TS
      when '01'
        return TimeStamp_Virtual;
      when '10'
        assert HaveECVExt(); // Otherwise ConstrainUnpredictableBits removes this case
        return TimeStamp_OffsetPhysical;
  end case;
else
  return time stamps of TRFCR_EL0.TS,
when '11'
    return TimeStamp_Physical;

TS_el1 = TRFCR_EL1.TS;
if TS_el1 == '00' || (!HaveECVExt() && TS_el1 == '10') then
    // Reserved value
    (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable_EL1TIMESTAMP, 2);

    case TS_el1 of
        when '01'
            return TimeStamp_Virtual;
        when '10'
            assert HaveECVExt();
            return TimeStamp_OffsetPhysical;
        when '11'
            return TimeStamp_Physical;
        otherwise
            Unreachable();         // ConstrainUnpredictableBits removes this case
    else
        return TimeStamp_CoreSight;

shared/trace/system/IsTraceCorePowered

    // Returns TRUE if the Trace Core Power Domain is powered up
    boolean IsTraceCorePowered();

J1.3.5  shared/translation

This section includes the following pseudocode functions:

• shared/translation/at/ATAccess on page J1-11534.
• shared/translation/at/EncodePARAttrs on page J1-11535.
• shared/translation/at/PAREncodeShareability on page J1-11535.
• shared/translation/at/TranslationStage on page J1-11536.
• shared/translation/attrs/DecodeDevice on page J1-11536.
• shared/translation/attrs/DecodeLDFAttr on page J1-11536.
• shared/translation/attrs/DecodeSDFAttr on page J1-11536.
• shared/translation/attrs/DecodeShareability on page J1-11536.
• shared/translation/attrs/EffectiveShareability on page J1-11537.
• shared/translation/attrs/IsS2ResultTagged on page J1-11537.
• shared/translation/attrs/MAIRAttr on page J1-11538.
• shared/translation/attrs/NormalNCMemAttr on page J1-11538.
• shared/translation/attrs/S1ConstrainUnpredictableRESMAIR on page J1-11538.
• shared/translation/attrs/S1DecodeMemAttrs on page J1-11538.
• shared/translation/attrs/S2CombineSIAttrHints on page J1-11539.
• shared/translation/attrs/S2CombineSIDevice on page J1-11540.
• shared/translation/attrs/S2CombineSIMemAttrs on page J1-11540.
• shared/translation/attrs/S2CombineSIShareability on page J1-11541.
• shared/translation/attrs/S2DecodeCacheability on page J1-11541.
• shared/translation/attrs/S2DecodeMemAttrs on page J1-11541.
• shared/translation/attrs/WalkMemAttrs on page J1-11542.
• shared/translation/faults/AlignmentFault on page J1-11542.
• shared/translation/faults/AsyncExternalAbort on page J1-11542.
• shared/translation/faults/NoFault on page J1-11543.
• shared/translation/gpc/AbovePPS on page J1-11543.
• shared/translation/gpc/DecodeGPTBlock on page J1-11543.
 Armv8 Pseudocode
J1.3 Shared pseudocode

• shared/translation/gpc/DecodeGPTContiguous on page J1-11544.
• shared/translation/gpc/DecodeGPTGranules on page J1-11544.
• shared/translation/gpc/DecodeGPTTable on page J1-11544.
• shared/translation/gpc/DecodePGS on page J1-11545.
• shared/translation/gpc/DecodePPS on page J1-11545.
• shared/translation/gpc/GPCFault on page J1-11546.
• shared/translation/gpc/GPCNoFault on page J1-11546.
• shared/translation/gpc/GPCRegistersConsistent on page J1-11546.
• shared/translation/gpc/GPICheck on page J1-11546.
• shared/translation/gpc/GPIIndex on page J1-11547.
• shared/translation/gpc/GPIValid on page J1-11547.
• shared/translation/gpc/GPTL0Size on page J1-11547.
• shared/translation/gpc/GPTLevel0Index on page J1-11547.
• shared/translation/gpc/GPTLevel1Index on page J1-11548.
• shared/translation/gpc/GPTWalk on page J1-11548.
• shared/translation/gpc/GranuleProtectionCheck on page J1-11549.
• shared/translation/gpc/PGS on page J1-11550.
• shared/translation/gpc/Table on page J1-11550.
• shared/translation/translation/S1TranslationRegime on page J1-11550.
• shared/translation/vmsa/AddressDescriptor on page J1-11551.
• shared/translation/vmsa/ContiguousSize on page J1-11551.
• shared/translation/vmsa/CreateAddressDescriptor on page J1-11551.
• shared/translation/vmsa/CreateFaultyAddressDescriptor on page J1-11552.
• shared/translation/vmsa/DescriptorType on page J1-11552.
• shared/translation/vmsa/Domains on page J1-11552.
• shared/translation/vmsa/FetchDescriptor on page J1-11552.
• shared/translation/vmsa/HasUnprivileged on page J1-11553.
• shared/translation/vmsa/IsAtomicRW on page J1-11553.
• shared/translation/vmsa/Regime on page J1-11553.
• shared/translation/vmsa/RegimeUsingAArch32 on page J1-11553.
• shared/translation/vmsa/S1TTWParams on page J1-11554.
• shared/translation/vmsa/S2TTWParams on page J1-11554.
• shared/translation/vmsa/SDFType on page J1-11555.
• shared/translation/vmsa/SecurityStateForRegime on page J1-11555.
• shared/translation/vmsa/StageOA on page J1-11555.
• shared/translation/vmsa/TA on page J1-11555.
• shared/translation/vmsa/TGx on page J1-11555.
• shared/translation/vmsa/TGxGranuleBits on page J1-11556.
• shared/translation/vmsa/TLBContext on page J1-11556.
• shared/translation/vmsa/TLBRecord on page J1-11556.
• shared/translation/vmsa/TTWState on page J1-11556.
• shared/translation/vmsa/TranslationRegime on page J1-11556.
• shared/translation/vmsa/TranslationSize on page J1-11557.
• shared/translation/vmsa/UseASID on page J1-11557.
• shared/translation/vmsa/UseVMID on page J1-11557.
• shared/translation/vmsa/VARange on page J1-11558.

shared/translation/at/ATAccess

 enumeration ATAccess { 
  ATAccess_Read, 
  ATAccess_Write, 
  ATAccess_ReadPAN, 
}
shared/translation/at/EncodePARAttrs

// EncodePARAttrs()
// ================
// Convert orthogonal attributes and hints to 64-bit PAR ATTR field.

bits(8) EncodePARAttrs(MemoryAttributes memattrs)
    bits(8) result;
    if HaveMTEExt() && memattrs.tagged then
        result<7:0> = '11110000';
        return result;
    if memattrs.memtype == MemType_Device then
        result<7:4> = '0000';
        if memattrs.device == DeviceType_nGnRnE then
            result<3:0> = '0000';
            else
                result<3:0> = '0100';
        if memattrs.device == DeviceType_nGRE then
            result<3:0> = '1000';
        elsif memattrs.device == DeviceType_GRE then
            result<3:0> = '1100';
            else // DeviceType_GRE
                result<3:0> = '1100';
                else
                    if memattrs.outer.attrs == MemAttr_WT then
                        result<7:6> = if memattrs.outer.transient then '00' else '10';
                        result<5:4> = memattrs.outer.hints;
                        elsif memattrs.outer.attrs == MemAttr_WB then
                            result<7:6> = if memattrs.outer.transient then '01' else '11';
                            result<5:4> = memattrs.outer.hints;
                        else // MemAttr_NC
                            result<7:4> = '0100';
                            if memattrs.inner.attrs == MemAttr_WT then
                                result<3:2> = if memattrs.inner.transient then '00' else '10';
                                result<1:0> = memattrs.inner.hints;
                                elsif memattrs.inner.attrs == MemAttr_WB then
                                    result<3:2> = if memattrs.inner.transient then '01' else '11';
                                    result<1:0> = memattrs.inner.hints;
                                else // MemAttr_NC
                                    result<3:0> = '0100';
                                    return result;
    shared/translation/at/PAREncodeShareability

// PAREncodeShareability()
// =======================
// Derive 64-bit PAR SH field.

bits(2) PAREncodeShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType_Device ||
        (memattrs.inner.attrs == MemAttr_NC &&
        memattrs.outer.attrs == MemAttr_NC)) then
        // Force Outer-Shareable on Device and Normal Non-Cacheable memory
        return '10';
    case memattrs.shareability of
        when Shareability_NSH return '00';
when Shareability_ISH return '11';
when Shareability_OSH return '10';

shared/translation/at/TranslationStage

enumeration TranslationStage {
    TranslationStage_1,
    TranslationStage_12
};

shared/translation/attrs/DecodeDevice

// DecodeDevice()
// ==============
// Decode output Device type

DeviceType DecodeDevice(bits(2) device)
case device of
    when '00'   return DeviceType_nGnRnE;
    when '01'   return DeviceType_nGnRE;
    when '10'   return DeviceType_nGRE;
    when '11'   return DeviceType_GRE;

shared/translation/attrs/DecodeLDFAttr

// DecodeLDFAttr()
// ===============
// Decode memory attributes using LDF (Long Descriptor Format) mapping

MemAttrHints DecodeLDFAttr(bits(4) attr)
    MemAttrHints ldfattr;
    if attr IN {'x0xx'} then ldfattr.attrs = MemAttr_WT; // Write-through
    elsif attr == '0100' then ldfattr.attrs = MemAttr_NC; // Non-cacheable
    elsif attr IN {'x1xx'} then ldfattr.attrs = MemAttr_WB; // Write-back
    else Unreachable();

    // Allocation hints are applicable only to cacheable memory.
    if ldfattr.attrs != MemAttr_NC then
        case attr<1:0> of
            when '00' ldfattr.hints = MemHint_No;  // No allocation hints
            when '01' ldfattr.hints = MemHint_WA;  // Write-allocate
            when '10' ldfattr.hints = MemHint_RA;  // Read-allocate
            when '11' ldfattr.hints = MemHint_RWA; // Read/Write allocate

        // The Transient hint applies only to cacheable memory with some allocation hints.
        if ldfattr.attrs != MemAttr_NC && ldfattr.hints != MemHint_No then
            ldfattr_transient = attr<3> == '0';

        return ldfattr;

shared/translation/attrs/DecodeSDFAttr

// DecodeSDFAttr()
// ===============
// Decode memory attributes using SDF (Short Descriptor Format) mapping

MemAttrHints DecodeSDFAttr(bits(2) rgn)
    MemAttrHints sdfattr;
    case rgn of
when '00' // Non-cacheable (no allocate)
sdfattr.attrs = MemAttr_NC;
when '01' // Write-back, Read and Write allocate
sdfattr.attrs = MemAttr_WB;
sdfattr.hints = MemHint_RWA;
when '10' // Write-through, Read allocate
sdfattr.attrs = MemAttr_WT;
sdfattr.hints = MemHint_RA;
when '11' // Write-back, Read allocate
sdfattr.attrs = MemAttr_WB;
sdfattr.hints = MemHint_RA;
sdfattr.transient = FALSE;
return sdfattr;

shared/translation/attrs/DecodeShareability

// DecodeShareability()
// ===============
// Decode shareability of target memory region

Shareability DecodeShareability(bits(2) sh)
case sh of
  when '10' return Shareability_OSH;
  when '11' return Shareability_ISH;
  when '00' return Shareability_NSH;
  otherwise
    case ConstrainUnpredictable(Unpredictable_Shareability) of
      when Constraint_OSH return Shareability_OSH;
      when Constraint_ISH return Shareability_ISH;
      when Constraint_NSH return Shareability_NSH;

shared/translation/attrs/EffectiveShareability

// EffectiveShareability()
// ================
// Force Outer Shareability on Device and Normal iNCoNC memory

Shareability EffectiveShareability(MemoryAttributes memattrs)
if (memattrs.memtype == MemType_Device ||
    (memattrs.inner.attrs == MemAttr_NC &&
     memattrs.outer.attrs == MemAttr_NC)) then
  return Shareability_OSH;
else
  return memattrs.shareability;

shared/translation/attrs/IsS2ResultTagged

// IsS2ResultTagged()
// =================
// Determine whether the combined output memory attributes of stage 1 and
// stage 2 indicate tagged memory

boolean IsS2ResultTagged(MemoryAttributes s2_memattrs, boolean s1_tagged)
if !HaveMTE2Ext() then
  return FALSE;
return {
  s1_tagged &&
  (s2_memattrs.memtype == MemType_Normal) &&
  (s2_memattrs.inner.attrs == MemAttr_WB) &&
(s2_memattrs.inner.hints == MemHint_RWA) &&
(!s2_memattrs.inner.transient) &&
(s2_memattrs.outer.attrs == MemAttr_WB) &&
(s2_memattrs.outer.hints == MemHint_RWA) &&
(!s2_memattrs.outer.transient));

shared/translation/attrs/MAIRAttr

// MAIRAttr()
// =========
// Retrieve the memory attribute encoding indexed in the given MAIR

bits(8) MAIRAttr(integer index, MAIRType mair)
    bit_index = 8 * index;
    return mair<bit_index+7:bit_index>;

shared/translation/attrs/NormalNCMemAttr

// NormalNCMemAttr()
// ==================
// Normal Non-cacheable memory attributes

MemoryAttributes NormalNCMemAttr()
    MemAttrHints non_cacheable;
    non_cacheable.attrs = MemAttr_NC;
    MemoryAttributes nc_memattrs;
    nc_memattrs.memtype = MemType_Normal;
    nc_memattrs.outer = non_cacheable;
    nc_memattrs.inner = non_cacheable;
    nc_memattrs.shareability = Shareability_OSH;
    nc_memattrs.tagged = FALSE;
    return nc_memattrs;

shared/translation/attrs/S1ConstrainUnpredictableRESMAIR

// S1ConstrainUnpredictableRESMAIR()
// =================================
// Determine whether a reserved value occupies MAIR_ELx.AttrN

boolean S1ConstrainUnpredictableRESMAIR(bits(8) attr, boolean s1aarch64)
    case attr of
        when '0000xx01' return !(s1aarch64 && HaveFeatXS());
        when '0000xxxx' return attr<1:0> != '00';
        when '01000000' return !(s1aarch64 && HaveFeatXS());
        when '10100000' return !(s1aarch64 && HaveFeatXS());
        when '11110000' return !(s1aarch64 && HaveMTE2Ext());
        when 'xxxx0000' return TRUE;
        otherwise return FALSE;

shared/translation/attrs/S1DecodeMemAttrs

// S1DecodeMemAttrs()
// ==================
// Decode MAIR-format memory attributes assigned in stage 1

MemoryAttributes S1DecodeMemAttrs(bits(8) attr_in, bits(2) sh, boolean s1aarch64)
    bits(8) attr = attr_in;
    if S1ConstrainUnpredictableRESMAIR(attr, s1aarch64) then
(-, attr) = ConstrainUnpredictableBits(Unpredictable_RESMAIR, 8);

MemoryAttributes memattrs;

case attr of
  when '0000xxxx' // Device memory
    memattrs.memtype = MemType_Device;
    memattrs.device = DecodeDevice(attr<3:2>);
    memattrs.tagged = FALSE;
    memattrs.xs = if s1aarch64 then NOT attr<0> else '1';
  when '01000000'
    assert s1aarch64 && HaveFeatXS();
    memattrs.memtype = MemType_Normal;
    memattrs.tagged = FALSE;
    memattrs.outer.attrs = MemAttr_NC;
    memattrs.inner.attrs = MemAttr_NC;
    memattrs.xs = '0';
  when '10100000'
    assert s1aarch64 && HaveFeatXS();
    memattrs.memtype = MemType_Normal;
    memattrs.tagged = FALSE;
    memattrs.outer.attrs = MemAttr_WT;
    memattrs.outer.hints = MemHint_RA;
    memattrs.outer.transient = FALSE;
    memattrs.inner.attrs = MemAttr_WT;
    memattrs.inner.hints = MemHint_RA;
    memattrs.inner.transient = FALSE;
    memattrs.xs = '0';
  when '11110000' // Tagged memory
    assert s1aarch64 && HaveMTE2Ext();
    memattrs.memtype = MemType_Normal;
    memattrs.tagged = TRUE;
    memattrs.outer.attrs = MemAttr_WB;
    memattrs.outer.hints = MemHint_RWA;
    memattrs.outer.transient = FALSE;
    memattrs.inner.attrs = MemAttr_WB;
    memattrs.inner.hints = MemHint_RWA;
    memattrs.inner.transient = FALSE;
    memattrs.xs = '0';
  otherwise
    memattrs.memtype = MemType_Normal;
    memattrs.outer = DecodeLDFAttr(attr<7:4>);
    memattrs.inner = DecodeLDFAttr(attr<3:0>);
    memattrs.tagged = FALSE;
    if (memattrs.inner.attrs == MemAttr_WB &&
        memattrs.outer.attrs == MemAttr_WB) then
      memattrs.xs = '0';
    else
      memattrs.xs = '1';
    memattrs.shareability = DecodeShareability(sh);

return memattrs;

shared/translation/attrs/S2CombineS1AttrHints

// S2CombineS1AttrHints()
// ================
// Determine resultant Normal memory cacheability and allocation hints from
// combining stage 1 Normal memory attributes and stage 2 cacheability attributes.

MemAttrHints S2CombineS1AttrHints(MemAttrHints s1_attrhints, MemAttrHints s2_attrhints)
MemAttrHints attrhints;

if s1_attrhints.attrs == MemAttr_NC || s2_attrhints.attrs == MemAttr_NC then

attrhints.attrs = MemAttr_NC;
elsif s1_attrhintsattrs == MemAttr_WT || s2_attrhintsattrs == MemAttr_WT then
    attrhints.attrs = MemAttr_WT;
else
    attrhints.attrs = MemAttr_WB;

// Stage 2 does not assign any allocation hints
// Instead, they are inherited from stage 1
if attrhints.attrs != MemAttr_NC then
    attrhints.hints = s1_attrhints.hints;
    attrhints.transient = s1_attrhints.transient;
return attrhints;

shared/translation/attrs/S2CombineS1Device

// S2CombineS1Device()
// ==============
// Determine resultant Device type from combining output memory attributes
// in stage 1 and Device attributes in stage 2

DeviceType S2CombineS1Device(DeviceType s1_device, DeviceType s2_device)

if s1_device == DeviceType_nGnRnE || s2_device == DeviceType_nGnRnE then
    return DeviceType_nGnRnE;
elsif s1_device == DeviceType_nGnRE || s2_device == DeviceType_nGnRE then
    return DeviceType_nGnRE;
elsif s1_device == DeviceType_nGRE || s2_device == DeviceType_nGRE then
    return DeviceType_nGRE;
else
    return DeviceType_GRE;

shared/translation/attrs/S2CombineS1MemAttrs

// S2CombineS1Mem_attrs()
// ====================
// Combine stage 2 with stage 1 memory attributes

MemoryAttributes S2CombineS1Mem_attrs(MemoryAttributes s1_memattrs,
                                          MemoryAttributes s2_memattrs)

MemoryAttributes memattrs;

if s1_memattrs.memtype == MemType_Device && s2_memattrs.memtype == MemType_Device then
    memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_memattrs.device);
elsif s1_memattrs.memtype == MemType_Device then // S2 Normal, S1 Device
    memattrs = s1_memattrs;
elsif s2_memattrs.memtype == MemType_Device then // S2 Device, S1 Normal
    memattrs = s2_memattrs;
else // S2 Normal, S1 Normal
    memattrs.deb:type = MemType_Normal;
    memattrs.inner = S2CombineS1AttrHints(s1_memattrs.inner, s2_memattrs.inner);
    memattrs.outer = S2CombineS1AttrHints(s1_memattrs.outer, s2_memattrs.outer);
    memattrs.tagged = IsS2ResultTagged(memattrs, s1_memattrs.tagged);
memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability,
                                                s2_memattrs.shareability);
memattrs.xs = s2_memattrs.xs;
memattrs.shareability = EffectiveShareability(memattrs);
return memattrs;
shared/translation/attrs/S2CombineS1Shareability

// S2CombineS1Shareability()
// =========================
// Combine stage 2 shareability with stage 1

Shareability S2CombineS1Shareability(Shareability s1_shareability, Shareability s2_shareability)

if (s1_shareability == Shareability_OSH || s2_shareability == Shareability_OSH) then
  return Shareability_OSH;
elsif (s1_shareability == Shareability_ISH || s2_shareability == Shareability_ISH) then
  return Shareability_ISH;
else
  return Shareability_NSH;

shared/translation/attrs/S2DecodeCacheability

// S2DecodeCacheability()
// ======================
// Determine the stage 2 cacheability for Normal memory

MemAttrHints S2DecodeCacheability(bits(2) attr)

MemAttrHints s2attr;

case attr of
  when '01' s2attr.attrs = MemAttr_NC;  // Non-cacheable
  when '10' s2attr.attrs = MemAttr_WT;  // Write-through
  when '11' s2attr.attrs = MemAttr_WB;  // Write-back
otherwise // Constrained unpredictable
  case ConstrainUnpredictable(Unpredictable_S2RESMEMATTR) of
    when Constraint_NC s2attr.attrs = MemAttr_NC;
    when Constraint_WT s2attr.attrs = MemAttr_WT;
    when Constraint_WB s2attr.attrs = MemAttr_WB;

// Stage 2 does not assign hints or the transient property
// They are inherited from stage 1 if the result of the combination allows it
s2attr.hints     = bits(2) UNKNOWN;
s2attr.transient = boolean UNKNOWN;

return s2attr;

shared/translation/attrs/S2DecodeMemAttrs

// S2DecodeMemAttrs()
// ==================
// Decode stage 2 memory attributes

MemoryAttributes S2DecodeMemAttrs(bits(4) attr, bits(2) sh, boolean s2aarch64)

MemoryAttributes memattrs;

case attr of
  when '00xx' // Device memory
    memattrs.memtype      = MemType_Device;
    memattrs.device       = DecodeDevice(attr<1:0>);
  otherwise // Normal memory
    memattrs.memtype      = MemType_Normal;
    memattrs.outer        = S2DecodeCacheability(attr<3:2>);
    memattrs.inner        = S2DecodeCacheability(attr<1:0>);

  memattrs.shareability = DecodeShareability(sh);
return memattrs;

**shared/translation/attrs/WalkMemAttrs**

```c
// WalkMemAttrs()
// ==============
// Retrieve memory attributes of translation table walk

MemoryAttributes WalkMemAttrs(bits(2) sh, bits(2) irgn, bits(2) orgn)
MemoryAttributes walkmemattrs;

walkmemattrs.memtype      = MemType_Normal;
walkmemattrs.shareability = DecodeShareability(sh);
walkmemattrs.inner        = DecodeSDFAttr(irgn);
walkmemattrs.outer        = DecodeSDFAttr(orgn);
walkmemattrs.tagged       = FALSE;
if (walkmemattrs.inner.attrs == MemAttr_WB &&
    walkmemattrs.outer.attrs == MemAttr_WB) then
    walkmemattrs.xs = '0';
else
    walkmemattrs.xs = '1';

return walkmemattrs;
```

**shared/translation/faults/AlignmentFault**

```c
// AlignmentFault()
// ================

FaultRecord AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)
FaultRecord fault;

fault.statuscode  = Fault_Alignment;
fault.acctype     = acctype;
fault.write       = iswrite;
fault.secondstage = secondstage;

return fault;
```

**shared/translation/faults/AsyncExternalAbort**

```c
// AsyncExternalAbort()
// ====================
// Return a fault record indicating an asynchronous External abort

FaultRecord AsyncExternalAbort(boolean parity, bits(2) errortype, bit extflag)
FaultRecord fault;

fault.statuscode = if parity then Fault_AsyncParity else Fault_AsyncExternal;
fault.extflag    = extflag;
fault.errortype  = errortype;
fault.acctype    = AccType_NORMAL;
fault.secondstage = FALSE;
fault.s2fs1walk  = FALSE;

return fault;
```
shared/translation/faults/NoFault

// NoFault()
// =========
// Return a clear fault record indicating no faults have occurred

FaultRecord NoFault()
{
    FaultRecord fault;
    fault.statuscode = Fault_None;
    fault.acctype = AccType_NORMAL;
    fault.secondstage = FALSE;
    fault.s2fs1walk = FALSE;
    fault.gpcfs2walk = FALSE;
    fault.gpcf = GPCNoFault();

    return fault;
}

shared/translation/gpc/AbovePPS

// AbovePPS()
// =========
// Returns TRUE if an address exceeds the range configured in GPCCR_EL3.PPS.

boolean AbovePPS(bits(52) address)
{
    pps = DecodePPS();
    if pps == 52 then
        return FALSE;

    return !IsZero(address<51:pps>);
}

shared/translation/gpc/DecodeGPTBlock

// DecodeGPTBlock()
// ================
// Validate and decode a GPT Block descriptor

(GPCF, GPTEntry) DecodeGPTBlock(PGSe pgs, bits(64) entry)
{
    assert entry<3:0> == GPT_Block;
    GPTEntry result;

    if !IsZero(entry<63:8>) then
        return (GPCF_Walk, GPTEntry UNKNOWN);
    if !GPIValid(entry<7:4>) then
        return (GPCF_Walk, GPTEntry UNKNOWN);

    result.gpi = entry<7:4>;
    result.level = 0;

    // GPT information from a level 0 GPT Block descriptor is permitted
    // to be cached in a TLB as though the Block is a contiguous region
    // of granules each of the size configured in GPCCR_EL3.PGS.
    case pgs of
        when PGS_4KB result.size = GPTRange_4KB;
        when PGS_16KB result.size = GPTRange_16KB;
        when PGS_64KB result.size = GPTRange_64KB;
        otherwise Unreachable();
    result.contig_size = GPTL0Size();

    return (GPCF_None, result);
}
shared/translation/gpc/DecodeGPTContiguous

// DecodeGPTContiguous()
// ================
// Validate and decode a GPT Contiguous descriptor

(GPCF, GPTEntry) DecodeGPTContiguous(PGSe pgs, bits(64) entry)
assert entry<3:0> == GPT_Contig;
GPTEntry result;
if !IsZero(entry<63:10>) then
  return (GPCF_Walk, result);
result.gpi = entry<7:4>;
if !GPIValid(result.gpi) then
  return (GPCF_Walk, result);

  if pgs of
    when PGS_4KB  result.size = GPTRange_4KB;
    when PGS_16KB result.size = GPTRange_16KB;
    when PGS_64KB result.size = GPTRange_64KB;
  otherwise Unreachable();

  case entry<9:8> of
    when '01' result.contig_size = GPTRange_2MB;
    when '10' result.contig_size = GPTRange_32MB;
    when '11' result.contig_size = GPTRange_512MB;
  otherwise return (GPCF_Walk, GPTEntry UNKNOWN);

result.level = 1;
return (GPCF_None, result);

shared/translation/gpc/DecodeGPTGranules

// DecodeGPTGranules()
// ================
// Validate and decode a GPT Granules descriptor

(GPCF, GPTEntry) DecodeGPTGranules(PGSe pgs, integer index, bits(64) entry)
GPTEntry result;
for i = 0 to 15
  if !GPIValid(entry<i*4 +:4>) then
    return (GPCF_Walk, result);
result.gpi = entry<index*4 +:4>;

  if pgs of
    when PGS_4KB  result.size = GPTRange_4KB;
    when PGS_16KB result.size = GPTRange_16KB;
    when PGS_64KB result.size = GPTRange_64KB;
  otherwise Unreachable();

    result.contig_size = result.size; // No contiguity
result.level = 1;
return (GPCF_None, result);

shared/translation/gpc/DecodeGPTTable

// DecodeGPTTable()
// ================
// Validate and decode a GPT Table descriptor
(GPCF, GPTTable) DecodeGPTTable(PGSe pgs, bits(64) entry)
    assert entry<3:0> == GPT_Table;
    GPTTable result;
    if !IsZero(entry<63:52,11:4>) then
        return (GPCF_Walk, GPTTable UNKNOWN);
    l0sz = GPTL0Size();
    integer p;
    case pgs of
        when PGS_4KB  p = 12;
        when PGS_16KB p = 14;
        when PGS_64KB p = 16;
        otherwise Unreachable();
    if !IsZero(entry<(l0sz-p)-2:12>) then
        return (GPCF_Walk, GPTTable UNKNOWN);
    case pgs of
        when PGS_4KB  result.address = entry<51:17>:Zeros(17);
        when PGS_16KB result.address = entry<51:15>:Zeros(15);
        when PGS_64KB result.address = entry<51:13>:Zeros(13);
        otherwise Unreachable();
    // The address must be within the range covered by the GPT
    if AbovePPS(result.address) then
        return (GPCF_AddressSize, GPTTable UNKNOWN);
    return (GPCF_None, result);

shared/translation/gpc/DecodePGS

// DecodePGS()
// =========
PGSe DecodePGS(bits(2) pgs)
    case pgs of
        when '00' return PGS_4KB;
        when '10' return PGS_16KB;
        when '01' return PGS_64KB;
        otherwise Unreachable();

shared/translation/gpc/DecodePPS

// DecodePPS()
// =========
// Size of region protected by the GPT, in bits.
integer DecodePPS()
    case GPCCR_EL3.PPS of
        when '000' return 32;
        when '001' return 36;
        when '010' return 40;
        when '011' return 42;
        when '100' return 44;
        when '101' return 48;
        when '110' return 52;
        otherwise Unreachable();
shared/translation/gpc/GPCFault

// GPCFault()
// =========
// Constructs and returns a GPCF

GPCFRecord GPCFault(GPCF gpf, integer level)

GPCFRecord fault;
    fault.gpf  = gpf;
    fault.level = level;
return fault;

shared/translation/gpc/GPCNoFault

// GPCNoFault()
// ============
// Returns the default properties of a GPCF that does not represent a fault

GPCFRecord GPCNoFault()

GPCFRecord result;
    result.gpf = GPCF_None;
return result;

shared/translation/gpc/GPCRegistersConsistent

// GPCRegistersConsistent()
// ========================
// Returns whether the GPT registers are configured correctly.
// This returns false if any fields select a Reserved value.

boolean GPCRegistersConsistent()

// Check for Reserved register values
if GPCCR_EL3.PPS == '111' || DecodePPS() > AArch64.PAMax() then
    return FALSE;
if GPCCR_EL3.PGS == '11' then
    return FALSE;
if GPCCR_EL3.SH == '01' then
    return FALSE;
// Inner and Outer Non-cacheable requires Outer Shareable
if GPCCR_EL3.<ORGN, IRGN> == '0000' && GPCCR_EL3.SH != '10' then
    return FALSE;

return TRUE;

shared/translation/gpc/GPICheck

// GPICheck()
// =========
// Returns whether an access to a given physical address space is permitted
// given the configured GPI value.
// paspace: Physical address space of the access
// gpi: Value read from GPT for the access

boolean GPICheck(PASpace paspace, bits(4) gpi)

case gpi of
    when GPT_NoAccess  return FALSE;
    when GPT_Secure    return paspace == PAS_Secure;
    when GPT_NonSecure return paspace == PAS_NonSecure;
    when GPT_Root      return paspace == PAS_Root;
    when GPT_Realm     return paspace == PAS_Realm;
when GPT.Any       return TRUE;
otherwise          Unreachable();

shared/translation/gpc/GPIIndex

// GPIIndex()
 rencont
integer GPIIndex(bits(52) pa)
case DecodePGS(GPCCR_EL3.PGS) of
    when PGS_4KB  return UInt(pa<15:12>);
    when PGS_16KB return UInt(pa<17:14>);
    when PGS_64KB return UInt(pa<19:16>);
otherwise Unreachable();

shared/translation/gpc/GPIValid

// GPIValid()
 rencont
boolean GPIValid(bits(4) gpi)
return gpi IN {GPT_NoAccess,
               GPT_Secure,
               GPT_NonSecure,
               GPT_Root,
               GPT_Realm,
               GPT_Any};

shared/translation/gpc/GPTL0Size

// GPTL0Size()
 rencont
integer GPTL0Size()
case GPCCR_EL3.L0GPTSZ of
    when '0000' return GPTRange_1GB;
    when '0100' return GPTRange_16GB;
    when '0110' return GPTRange_64GB;
    when '1001' return GPTRange_512GB;
otherwise Unreachable();
return 30;

shared/translation/gpc/GPTLevel0Index

// GPTLevel0Index()
 rencont
integer GPTLevel0Index(bits(52) pa)
// Input address and index bounds
pps = DecodePPS();
10sz = GPTL0Size();
if pps <= 10sz then
    return 0;
return UInt(pa<pps-1:10sz>);
shared/translation/gpc/GPTLevel1Index

// GPTLevel1Index()
// =================
// Compute the level 1 index based on input PA.

integer GPTLevel1Index(bits(52) pa)
// Input address and index bounds
l0sz = GPTL0Size();
case DecodePGS(GPCCR_EL3.PGS) of
  when PGS_4KB  return UInt(pa<l0sz-1:16>);
  when PGS_16KB return UInt(pa<l0sz-1:18>);
  when PGS_64KB return UInt(pa<l0sz-1:20>);
  otherwise Unreachable();

shared/translation/gpc/GPTWalk

// GPTWalk()
// =========
// Get the GPT entry for a given physical address, pa

(GPCFRecord, GPTEntry) GPTWalk(bits(52) pa, AccessDescriptor accdesc)
// GPT base address
bits(52) base;
pgs = DecodePGS(GPCCR_EL3.PGS);
// The level 0 GPT base address is aligned to the greater of:
// * the size of the level 0 GPT, determined by GPCCR_EL3.{PPS, L0GPTSZ}.
// * 4KB
base = GPTBR_EL3.BADDR:Zeros(12);
pps = DecodePPS();
l0sz = GPTL0Size();
integer alignment = Max((pps - l0sz) + 3, 12);
base = base AND NOT ZeroExtend(Ones(alignment), 52);

// Access attributes and address for GPT fetches
AddressDescriptor gptaddrdesc;
gptaddrdesc.fault = NoFault();
gptaddrdesc.memattrs = WalkMemAttrs(GPCCR_EL3.SH, GPCCR_EL3.ORGN, GPCCR_EL3.IRGN);

// Address of level 0 GPT entry
gptaddrdesc.paddress.paspace = PAS_Root;
gptaddrdesc.paddress.address = base + GPTLevel0Index(pa) * 8;

// Fetch L0GPT entry
bits(64) level_0_entry;
PhysMemRetStatus memstatus;
(memstatus, level_0_entry) = PhysMemRead(gptaddrdesc, 8, accdesc);
if IsFault(memstatus) then
  return (GPCFault(GPCF_EABT, 0), GPTEntry UNKNOWN);
GPTEntry result;
GPTTable table;
GPCF gpf;
case level_0_entry<3:0> of
  when GPT_Block
    // Decode the GPI value and return that
    (gpf, result) = DecodeGPTBlock(pgs, level_0_entry);
    result.pa = pa;
    return (GPCFault(gpf, 0), result);
  when GPT_Table
    // Decode the table entry and continue walking
    (gpf, table) = DecodeGPTTable(pgs, level_0_entry);
    if gpf != GPCF_None then
      return (GPCFault(gpf, 0), GPTEntry UNKNOWN);
  otherwise
// GPF - invalid encoding
return (GPCFault(GPCF_Walk, 0), GPTEntry UNKNOWN);

// Must be a GPT Table entry
assert level_0_entry<3:0> == GPT_Table;

// Address of level 1 GPT entry
offset = GPTLevel1Index(pa) * 8;
gptaddrdesc.paddress.address = table.address + offset;

// Fetch L1GPT entry
bits(64) level_1_entry;
(memstatus, level_1_entry) = PhysMemRead(gptaddrdesc, 8, accdesc);
if IsFault(memstatus) then
  return (GPCFault(GPCF_EABT, 1), GPTEntry UNKNOWN);

case level_1_entry<3:0> of
  when GPT_Contig
    (gpf, result) = DecodeGPTContiguous(pgs, level_1_entry);
  otherwise
    gpi_index = GPIIndex(pa);
    (gpf, result) = DecodeGPTGranules(pgs, gpi_index, level_1_entry);

if gpf != GPCF_None then
  return (GPCFault(gpf, 1), GPTEntry UNKNOWN);

result.pa = pa;
return (GPCNoFault(), result);

shared/translation/gpc/GranuleProtectionCheck

// GranuleProtectionCheck()
// ========================
// Returns whether a given access is permitted, according to the
// granule protection check.
// addrdesc and accdesc describe the access to be checked.

GPCFRecord GranuleProtectionCheck(AddressDescriptor addrdesc, AccessDescriptor accdesc)

assert HaveRME();
// The address to be checked
address = addrdesc.paddress;

// Bypass mode - all accesses pass
if GPCR_EL3.GPC == '0' then
  return GPCNoFault();

// Configuration consistency check
if !GPCRegistersConsistent() then
  return GPCFault(GPCF_Walk, 0);

// Input address size check
if AbovePPS(address.address) then
  if address.paspace == PAS_NonSecure then
    return GPCNoFault();
  else
    return GPCFault(GPCF_Fail, 0);

// GPT base address size check
bits(52) gpt_base = GPTBR_EL3.BADDR:Zeros(12);
if AbovePPS(gpt_base) then
  return GPCFault(GPCF_AddressSize, 0);

// GPT lookup
(gpfc, gpt_entry) = GPTWalk(address.address, accdesc);
if gpfc.gpf != GPCF_None then
return gpcf;
// Check input physical address space against GPI
permitted = GPICheck(address.paspace, gpt_entry.gpi);
if !permitted then
  gpcf = GPCFault(GPCF_Fail, gpt_entry.level);
  return gpcf;
// Check passed
return GPCNoFault();

shared/translation/gpc/PGS

definition type: PGSe =
  PGS_4KB,
  PGS_16KB,
  PGS_64KB
;

shared/translation/gpc/Table

class constant bits(4) GPT_NoAccess = '0000';
class constant bits(4) GPT_Table = '0011';
class constant bits(4) GPT_Block = '0001';
class constant bits(4) GPT_Contig = '0001';
class constant bits(4) GPT_Secure = '1000';
class constant bits(4) GPT_NonSecure = '1001';
class constant bits(4) GPT_Root = '1010';
class constant bits(4) GPT_Realm = '1011';
class constant bits(4) GPT_Any = '1111';
class constant integer GPTRange_4KB = 12;
class constant integer GPTRange_16KB = 14;
class constant integer GPTRange_64KB = 16;
class constant integer GPTRange_2MB = 21;
class constant integer GPTRange_32MB = 25;
class constant integer GPTRange_512MB = 29;
class constant integer GPTRange_1GB = 30;
class constant integer GPTRange_16GB = 34;
class constant integer GPTRange_64GB = 36;
class constant integer GPTRange_512GB = 39;

definition type: GPTTable =
  bits(52) address // Base address of next table
;

definition type: GPTEntry =
  bits(4) gpi, // GPI value for this region
  integer size, // Region size
  integer contig_size, // Contiguous region size
  integer level, // Level of GPT lookup
  bits(52) pa // PA uniquely identifying the GPT entry
;

shared/translation/translation/S1TranslationRegime

  // S1TranslationRegime()
  // =====================
  // Stage 1 translation regime for the given Exception level
bits(2) S1TranslationRegime(bits(2) el)
  if el != EL0 then
return el;
  elsif HaveEl(EL3) && ELUsingAArch32(EL3) && SCR.NS == '0' then
    return EL3;
  elsif HaveVirtHostExt() && ELIsInHost(el) then
    return EL2;
  else
    return EL1;
  end if;

// S1TranslationRegime()
// =====================
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the System register accessors (SCTLR[], etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
return S1TranslationRegime(PSTATE.EL);

shared/translation/vmsa/AddressDescriptor

type AddressDescriptor is (
  FaultRecord      fault,      // fault.statuscode indicates whether the address is valid
  MemoryAttributes memattrs,
  FullAddress      paddress,
  bits(64)         vaddress
)

constant integer FINAL_LEVEL = 3;

shared/translation/vmsa/ContiguousSize

// ContiguousSize()
// ================
// Return the number of entries log 2 marking a contiguous output range

integer ContiguousSize(TGx tgx, integer level)
  case tgx of
    when TGx_4KB
      assert level IN {1, 2, 3};
      return 4;
    when TGx_16KB
      assert level IN {2, 3};
      return if level == 2 then 5 else 7;
    when TGx_64KB
      assert level IN {2, 3};
      return 5;
  end if;

shared/translation/vmsa/CreateAddressDescriptor

// CreateAddressDescriptor()
// =========================
// Set internal members for address descriptor type to valid values

AddressDescriptor CreateAddressDescriptor(bits(64) va, FullAddress pa,
  MemoryAttributes memattrs)
  AddressDescriptor addrdesc;
  addrdesc.paddress = pa;
  addrdesc.vaddress = va;
  addrdesc.memattrs = memattrs;
  addrdesc.fault    = NoFault();
  return addrdesc;
shared/translation/vmsa/CreateFaultyAddressDescriptor

// CreateFaultyAddressDescriptor()
// ===================================
// Set internal members for address descriptor type with values indicating error

AddressDescriptor CreateFaultyAddressDescriptor(bits(64) va, FaultRecord fault)
    AddressDescriptor addrdesc;
    addrdesc.vaddress = va;
    addrdesc.fault = fault;
    return addrdesc;

shared/translation/vmsa/DescriptorType

enumeration DescriptorType {
    DescriptorType_Table,
    DescriptorType_Block,
    DescriptorType_Page,
    DescriptorType_Invalid
};

shared/translation/vmsa/Domains

constant bits(2) Domain_NoAccess = '00';
constant bits(2) Domain_Client = '01';
constant bits(2) Domain_Manager = '11';

shared/translation/vmsa/FetchDescriptor

// FetchDescriptor()
// =================
// Fetch a translation table descriptor

(FaultRecord, bits(N)) FetchDescriptor(bit ee, AddressDescriptor walkaddress, FaultRecord fault_in,
    integer N)
    // 32-bit descriptors for AArch32 Short-descriptor format
    // 64-bit descriptors for AArch64 or AArch32 Long-descriptor format
    assert N == 32 || N == 64;
    bits(N) descriptor;
    FaultRecord fault = fault_in;
    AccessDescriptor walkacc;
    walkacc.acctype = AccType_TTW;
    // MPAM PARTID for translation table walk is determined by the access invoking the translation
    walkacc.mpam = GenMPAMcurEL(fault.acctype);
    if HaveRME() then
        fault.gpcf = GranuleProtectionCheck(walkaddress, walkacc);
        if fault.gpcf.gpf != GPCF_None then
            fault.statuscode = Fault_GPCFOnWalk;
            fault.paddress = walkaddress.paddress;
            fault.gpcfs2walk = fault.secondstage;
            return (fault, bits(N) UNKNOWN);
    PhysMemRetStatus memstatus;
    (memstatus, descriptor) = PhysMemRead(walkaddress, N DIV 8, walkacc);
    if IsFault(memstatus) then
        fault = HandleExternalTTWAbort(memstatus, fault.write, walkaddress,
            walkacc, N DIV 8, fault);
        if IsFault(fault.statuscode) then
            return (fault, bits(N) UNKNOWN);
if ee == '1' then
    descriptor = BigEndianReverse(descriptor);

    return (fault, descriptor);

shared/translation/vmsa/HasUnprivileged

// HasUnprivileged()
// ================
// Returns whether a translation regime serves EL0 as well as a higher EL

boolean HasUnprivileged(Regime regime)
    return (regime IN {
        Regime_EL20,
        Regime_EL30,
        Regime_EL10
    });

shared/translation/vmsa/IsAtomicRW

// IsAtomicRW()
// =============
// Is the access an atomic operation?

boolean IsAtomicRW(AccType acctype)
    return acctype IN {
        AccType_ATOMICRW,
        AccType_ORDEREDRW,
        AccType_ORDEREDATOMICRW
    };

shared/translation/vmsa/Regime

enumeration Regime {
    Regime_EL3,  // EL3
    Regime_EL30, // EL3&0 (PL1&0 when EL3 is AArch32)
    Regime_EL2,  // EL2
    Regime_EL20, // EL2&0
    Regime_EL10  // EL1&0
};

shared/translation/vmsa/RegimeUsingAArch32

// RegimeUsingAArch32()
// ================
// Determine if the EL controlling the regime executes in AArch32 state

boolean RegimeUsingAArch32(Regime regime)
    case regime of
        when Regime_EL10 return ELUsingAArch32(EL1);
        when Regime_EL30 return TRUE;
        when Regime_EL20 return FALSE;
        when Regime_EL2  return ELUsingAArch32(EL2);
        when Regime_EL3  return FALSE;
shared/translation/vmsa/S1TTWParams

```pseudocode
type S1TTWParams is (  // A64-VMSA exclusive parameters
    bit ha, // TCR_ELx.HA
    bit hd, // TCR_ELx.HD
    bit tbi, // TCR_ELx.TBI{x}
    bit tbid, // TCR_ELx.TBID{x}
    bit nfd, // TCR_EL1.NFDx or TCR_EL2.NFDx when HCR_EL2.E2H == '1'
    bit e0pd, // TCR_EL1.E0PDx or TCR_EL2.E0PDx when HCR_EL2.E2H == '1'
    bit ds, // TCR_ELx.DS
    bits(3) ps, // TCR_ELx.{I}PS
    bits(6) txsz, // TCR_ELx.TxSZ
    bit epan, // SCTLR_EL1.EPAN or SCTLR_EL2.EPAN when HCR_EL2.E2H == '1'
    bit dct, // HCR_EL2.DCT
    bit nvl, // HCR_EL2.NVI
    bit cmow, // SCTLR_EL1.CMOW or SCTLR_EL2.CMOW when HCR_EL2.E2H == '1'
  )

  // A32-VMSA exclusive parameters
  bits(3) t0sz, // TTBCR.T0SZ
  bits(3) t1sz, // TTBCR.T1SZ
  bit uwxn, // SCTLR.UWXN

  // Parameters common to both A64-VMSA & A32-VMSA (A64/A32)
  TGx         tgx, // TCR_ELx.TGx / Always TGx_4KB
  bits(2) irgn, // TCR_ELx.IRGNx / TTBCR.IRGNx or HTCR.IRGN0
  bits(2) orgn, // TCR_ELx.ORGNx / TTBCR.ORGNx or HTCR.ORG0
  bits(2) sh, // TCR_ELx.SHx / TTBCR.SHx or HTCR.SH0
  bit hpd, // TCR_ELx.HPD{x} / TTBCR2.HPDx or HTCR.HPD
  bit ee, // SCTLR_ELx.EE / SCTLR.EE or HSCTLR.EE
  bit wxn, // SCTLR_ELx.WXN / SCTLR.WXN or HSCTLR.WXN
  bit ntlsmd, // SCTLR_ELx.nTLSMD / SCTLR.nTLSMD or HSCTLR.nTLSMD
  bit dc, // HCR_EL2.DC / HCR.DC
  bit sif, // SCR_EL3.SIF / SCR.SIF
  MAIRType    mair // MAIR1:MAIR0 or HMAIR1:HMAIR0
)

shared/translation/vmsa/S2TTWParams

type S2TTWParams is (  // A64-VMSA exclusive parameters
    bit ha, // VTCR_EL2.HA
    bit hd, // VTCR_EL2.HD
    bit s2, // V(S)TCR_EL2.SL2
    bit ds, // VTCR_EL2.DS
    bit sw, // VSTCR_EL2.SW
    bit nsw, // VTCR_EL2.NSW
    bit sa, // VSTCR_EL2.SA
    bit nsa, // VTCR_EL2.NSA
    bits(3) ps, // VTCR_EL2.PS
    bits(6) txsz, // V(S)TCR_EL2.T0SZ
    bit fwb, // HCR_EL2.PTW
    bit cmow, // HCRX_EL2.CMOW

  )

  // A32-VMSA exclusive parameters
  bit s, // VTCR.S
  bits(4) t0sz, // VTCR.T0SZ

  // Parameters common to both A64-VMSA & A32-VMSA if implemented (A64/A32)
  TGx         tgx, // V(S)TCR_EL2.TG0 / Always TGx_4KB
  bits(2) sl0, // V(S)TCR_EL2.SL0 // VTCR.SL0
  bits(2) irgn, // VTCR_EL2.IRGN0 / VTCR.IRGN0
  bits(2) orgn, // VTCR_EL2.ORG0 / VTCR.ORG0
  bits(2) sh, // VTCR_EL2.SH0 / VTCR.SH0
  bit ee, // SCTLR_EL2.EE / HSCTLR.EE
  bit ptw, // HCR_EL2.PTW / HCR.PTW
```
bit vm // HCR_EL2.VM / HCR.VM

shared/translation/vmsa/SDFType

enumeration SDFType {
    SDFType_Table,
    SDFType_Invalid,
    SDFType_Supersession,
    SDFType_Section,
    SDFType_LargePage,
    SDFType_SmallPage
};

shared/translation/vmsa/SecurityStateForRegime

// SecurityStateForRegime()
// ========================
// Return the Security State of the given translation regime

SecurityState SecurityStateForRegime(Regime regime)
    case regime of
        when Regime_EL3 return SecurityStateAtEL(EL3);
        when Regime_EL30 return SS_Secure; // A32 EL3 is always Secure
        when Regime_EL2 return SecurityStateAtEL(EL2);
        when Regime_EL20 return SecurityStateAtEL(EL2);
        when Regime_EL10 return SecurityStateAtEL(EL1);

shared/translation/vmsa/StageOA

// StageOA()
// =========
// Given the final walk state (a page or block descriptor), map the untranslated
// input address bits to the output address

FullAddress StageOA(bits(64) ia, TGx tgx, TTWState walkstate)
    // Output Address
    FullAddress oa;
    integer csize;

    tsize = TranslationSize(tgx, walkstate.level);
    if walkstate.contiguous == '1' then
        csize = ContiguousSize(tgx, walkstate.level);
    else
        csize = 0;

    ia_msb = tsize + csize;
    oa.paspace = walkstate.baseaddress.paspace;
    oa.address = walkstate.baseaddress.address<51:ia_msb>:ia<ia_msb-1:0>;

    return oa;

shared/translation/vmsa/TGx

enumeration TGx {
    TGx_4KB,
    TGx_16KB,
    TGx_64KB
};
shared/translation/vmsa/TGxGranuleBits

// TGxGranuleBits()
// ================
// Retrieve the address size, in bits, of a granule

integer TGxGranuleBits(TGx tgx)
    case tgx of
        when TGx_4KB  return 12;
        when TGx_16KB return 14;
        when TGx_64KB return 16;

shared/translation/vmsa/TLBContext

type TLBContext is (SecurityState ss,
    Regime        regime,
    bits(16)      vmid,
    bits(16)      asid,
    bit           nG,
    PASpace       ipaspace, // Used in stage 2 lookups & invalidations only
    boolean       includes_s1,
    boolean       includes_s2,
    boolean       includes_gpt,
    bits(64)      ia,       // Input Address
    TGx           tg,
    bit           cnp,
    bit           xs)       // XS attribute (FEAT_XS)

shared/translation/vmsa/TLBRecord

type TLBRecord is (TLBContext  context,
    TTWState    walkstate,
    integer     blocksize,    // Number of bits directly mapped from IA to OA
    integer     contigsize,   // Number of entries log 2 marking a contiguous output range
    bits(64)    s1descriptor, // Stage 1 leaf descriptor in memory (valid if the TLB caches stage 1)
    bits(64)    s2descriptor  // Stage 2 leaf descriptor in memory (valid if the TLB caches stage 2)

shared/translation/vmsa/TTWState

type TTWState is (boolean             istable,
    integer             level,
    FullAddress         baseaddress,
    bit                 contiguous,
    bit                 nG,
    bit                 guardedpage,
    SDFType             sdftype, // AArch32 Short-descriptor format walk only
    bits(4)             domain, // AArch32 Short-descriptor format walk only
    MemoryAttributes    memattrs,
    Permissions         permissions)

shared/translation/vmsa/TranslationRegime

// TranslationRegime()
// ================
// Select the translation regime given the target EL and PE state
Regime Translation

Regime Translation(bits(2) el_in, AccType acc_type)

    bits(2) el;
    case acc_type of
        when AccType_NV2REGISTER
            assert EL2Enabled() && el_in == EL1;
            el = EL2;
        otherwise
            el = el_in;
        if el == EL3 then
            return if ELUsingAArch32(EL3) then Regime_EL30 else Regime_EL3;
        elsif el == EL2 then
            return if ELIsInHost(EL2) then Regime_EL20 else Regime_EL2;
        elsif el == EL1 then
            return Regime_EL10;
        elsif el == EL0 then
            if CurrentSecurityState() == SS_Secure && ELUsingAArch32(EL3) then
                return Regime_EL30;
            elsif ELIsInHost(EL0) then
                return Regime_EL20;
            else
                return Regime_EL10;
        else
            Unreachable();

shared/translation/vmsa/TranslationSize

    // TranslationSize()
    // ===========
    // Compute the number of bits directly mapped from the input address
    // to the output address
    integer TranslationSize(TGx tgx, integer level)
        granulebits = TGxGranuleBits(tgx);
        blockbits = (FINAL_LEVEL - level) * (granulebits - 3);
        return granulebits + blockbits;

shared/translation/vmsa/UseASID

    // UseASID()
    // =======
    // Determine whether the translation context for the access requires ASID or is a global entry
    boolean UseASID(TLBContext access)
        return HasUnprivileged(access.regime);

shared/translation/vmsa/UseVMID

    // UseVMID()
    // =======
    // Determine whether the translation context for the access requires VMID to match a TLB entry
    boolean UseVMID(TLBContext access)
        return access.regime == Regime_EL10 && EL2Enabled();
shared/translation/vmsa/VARange

enumeration VARange {
    VARange_LOWER,
    VARange_UPPER
};
Part K
Appendixes
Appendix K1
Architectural Constraints on UNPREDICTABLE Behaviors

This chapter describes the architectural constraints on UNPREDICTABLE behaviors in the Armv8 and later architectures. It contains the following sections:

- AArch32 CONSTRAINED UNPREDICTABLE behaviors on page K1-11562.
- AArch64 CONSTRAINED UNPREDICTABLE behaviors on page K1-11584.
K1.1 AArch32 CONSTRAINED UNPREDICTABLE behaviors

From the introduction of Armv8, the architecture defines architecturally required constraints on many behaviors that are UNPREDICTABLE in Armv7. The following sections define those constraints:

- Overview of the constraints on Armv7 UNPREDICTABLE behaviors on page K1-11563.
- Using R13 by instruction on page K1-11563.
- Using R15 by instruction on page K1-11563.
- Branching into an IT block on page K1-11564.
- Branching to an unaligned PC on page K1-11564.
- Loads and Stores to unaligned locations on page K1-11564.
- CONSTRAINED UNPREDICTABLE behavior associated with IT instructions and PSTATE.IT on page K1-11564.
- Unallocated System register access instructions on page K1-11565.
- SBZ or SBO fields T32 and A32 in instructions on page K1-11566.
- UNPREDICTABLE cases in immediate constants in T32 data-processing instructions on page K1-11566.
- UNPREDICTABLE cases in immediate constants in Advanced SIMD instructions on page K1-11566.
- CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values on page K1-11567.
- CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization on page K1-11567.
- Translation Table Base Address alignment on page K1-11568.
- Handling of System register control fields for Advanced SIMD and floating-point operation on page K1-11568.
- Mapping of non-idempotent memory locations using the Normal memory type on page K1-11569.
- The Performance Monitors Extension on page K1-11569.
- The Activity Monitors Extension on page K1-11571.
- Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED on page K1-11572.
- Out of range VA on page K1-11572.
- Instruction fetches from Device memory on page K1-11572.
- Multi-access instructions that load the PC from Device memory on page K1-11572.
- Programming CSSEL.R Level for a cache level that is not implemented on page K1-11572.
- Crossing a page boundary with different memory types or Shareability attributes on page K1-11573.
- Crossing a 4KB boundary with a Device access on page K1-11573.
- UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs on page K1-11573.
- CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings on page K1-11574.
- Out of range values of the Set/Way/Index fields in cache maintenance instructions on page K1-11574.
- CONSTRAINED UNPREDICTABLE behavior for A32 and T32 System instructions in the base instruction set on page K1-11575.
- CONSTRAINED UNPREDICTABLE behavior, A32 and T32 Advanced SIMD and floating-point instructions on page K1-11577.
- CONSTRAINED UNPREDICTABLE behaviors associated with the VTCR on page K1-11581.
- CONSTRAINED UNPREDICTABLE behavior of EL2 features on page K1-11581.
- Reserved values in System and memory-mapped registers and translation table entries on page K1-11583.
- CONSTRAINED UNPREDICTABLE behavior in Debug state on page K1-11583.
K1.1.1 Overview of the constraints on Armv7 UNPREDICTABLE behaviors

The term UNPREDICTABLE describes a number of cases where the architecture has a feature that software must not use. For execution in AArch32 state, where previous versions of the architecture define behavior as UNPREDICTABLE, from the introduction of Armv8-A the architecture specifies a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

Note
Software designed to be compatible with the Armv8-A and later architectures must not rely on these CONSTRAINED UNPREDICTABLE cases.

K1.1.2 Using R13 by instruction

In prior versions of the architecture, the use of R13 by instruction as a named register specifier was described as UNPREDICTABLE in the pseudocode. From the introduction of the Armv8-A architecture, the use of R13 as a named register specifier is not UNPREDICTABLE, unless this is specifically stated, and R13 can be used in the regular form. Bits[1:0] of R13 are not treated as SBZP in the Armv7 architecture or RES0 in the Armv8 and later architectures, but can hold any values programmed into them.

K1.1.3 Using R15 by instruction

All uses of R15 by instruction as a named register specifier for a source register that are described as CONSTRAINED UNPREDICTABLE in the pseudocode or in other places in this Manual must do one of the following:

- Cause the instruction to be treated as UNDEFINED.
- Cause the instruction to execute as a NOP.
- Read the program counter with the standard offset that applies for the current instruction set.
- Read the program counter with the standard offset that applies for the current instruction set with alignment to a word boundary.
- Read 0. This is Arm preferred behavior.
- Read or return an UNKNOWN value for the source register specified as R15.

All uses of R15 as a named register specifier for a destination register that are described as CONSTRAINED UNPREDICTABLE in the pseudocode or in other places in this reference manual must do one of the following:

- Cause the instruction to be treated as UNDEFINED.
- Cause the instruction to execute as a NOP.
- Ignore the write.
- Branch to an UNKNOWN location in either A32 or T32 state.

Instructions that are CONSTRAINED UNPREDICTABLE when the base register is R15 and the instruction specifies a writeback of the base register, are treated as having R15 as both a source register and a destination register.

For instructions that have two destination registers, for example LDRD, MRRC, and many of the multiply instructions, if Rt, R12, RdLo, or RdHi is R15, then the other destination register of the pair is UNKNOWN if the CONSTRAINED UNPREDICTABLE behavior for the write to R15 is either to ignore the write or to branch to an UNKNOWN location.

For instructions that affect any or all of PSTATE.\{N, Z, C, V\}, PSTATE.Q, and PSTATE.GE when the register specifier is not R15, any flags affected by an instruction that is CONSTRAINED UNPREDICTABLE when the register specifier is R15 become UNKNOWN.

In addition, for MRRC instructions that use R15 as the destination register descriptor, and therefore target APSR.\_nzcv where these are described as being CONSTRAINED UNPREDICTABLE, PSTATE.\{N, Z, C, V\} becomes UNKNOWN.
K1.1.4 Branching into an IT block

Branching into an IT block leads to CONSTRAINED UNPREDICTABLE behavior. Execution starts from the address determined by the branch, but each instruction in the IT block is:

- Executed as if it were not in an IT block. This means that it is executed unconditionally.
- Executed as if it had passed its Condition code check within an IT block.
- Executed as a NOP. That is, it behaves as if it had failed the Condition code check.

K1.1.5 Branching to an unaligned PC

In A32 state, when branching to an address that is not word aligned and is defined to be CONSTRAINED UNPREDICTABLE, one of the following behaviors must occur:

- The unaligned location is forced to be aligned.
- The unaligned address generates a Prefetch Abort on the first instruction using the unaligned PC value.
  
  If that instruction is executed at EL0 and either of the following applies, the exception is taken to EL2:
  - EL2 is using AArch32 and the value of HCR.TGE is 1.
  - EL2 is using AArch64 and the value of HCR_EL2.TGE is 1.

  If the instruction is executed at EL0 when the applicable TGE bit is 0 the exception is taken to EL1.

  If the instruction is executed at an Exception level that is higher than EL0 the exception is taken to the Exception level at which the instruction was executed.

  In all cases, the exception is generated only if the first instruction using the unaligned PC value is architecturally executed.

If the exception that results from a branch to an unaligned PC value:

- Is taken to an Exception level that is using AArch64, it is reported as a PC alignment fault exception, see ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault on page D17-5673.

- Is taken to an Exception level that is using AArch32, it is reported as a Prefetch Abort exception, see Prefetch Abort exception reporting a PC alignment fault exception on page G1-8972.

  — Note ————

Because bit[0] is used for interworking, it is impossible to specify a branch to A32 state when the bottom bit of the target address is 1. Therefore the bottom bit of IFAR, HIFAR, or FAR_ELx is 0 for all these cases.

K1.1.6 Loads and Stores to unaligned locations

Some unaligned loads and stores in the Armv8-A and later architectures are described as CONSTRAINED UNPREDICTABLE to do one of the following:

- Take an alignment fault.
- Perform the specified load or store to the unaligned memory location.

K1.1.7 CONSTRAINED UNPREDICTABLE behavior associated with IT instructions and PSTATE.IT

A number of instructions in the architecture are described as being CONSTRAINED UNPREDICTABLE either:

- Anywhere within an IT block.
- As an instruction within an IT block, other than the last instruction within an IT block.

Unless otherwise stated in this manual, when these instructions are committed for execution, one of the following occurs:

- An UNDEFINED exception results.
- The instructions are executed as if they had passed the Condition code check.
- The instructions execute as NOPs. This means that they behave as if they had failed the Condition code check.

The behavior might in some implementations vary from instruction to instruction, or between different instances of the same instruction.
Many instructions that are CONSTRAINED UNPREDICTABLE in an IT block are branch instructions or other non-sequential instructions that change the PC. Where these instructions are not treated as UNDEFINED within an IT block, the remaining iterations of the PSTATE.IT state machine must be treated in one of the following ways:

- **PSTATE.IT** is cleared to 0.
- **PSTATE.IT** advances for either a sequential or a nonsequential change of the PC in the same way as it does for instructions that are not CONSTRAINED UNPREDICTABLE that cause a sequential change of the PC.

**Note**

This does not apply to an instruction that is the last instruction in an IT block.

The instructions addressed by the updated PC must do one of the following:

- Execute as if they had passed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as NOPs. That is, they behave as if they had failed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as if they were unconditional, or, if the instructions are part of another IT block, in accordance with the behavior described in *Branching into an IT block on page K1-11564.*

The behavior might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

For exception returns or Debug state exits that cause PSTATE.IT to be set to a reserved value in T32 state or that return to A32 state with a nonzero value in PSTATE.IT, the PSTATE.IT bits are forced to ‘00000000’. The reserved values are:

```
PSTATE.IT[7:4] != ‘0000’ && PSTATE.IT[3:0] == ‘0000’
PSTATE.IT[2:0] != ‘000’ when SCTLR/SCTLR_EL_1.ITD == ‘1’
```

Exception returns or Debug state exits that set PSTATE.IT to a non-reserved value in T32 state can occur when the flow of execution returns to a point:

- Outside an IT block, but with the PSTATE.IT bits set to a value other than ‘00000000’.
- Inside an IT block, but with a different value of the PSTATE.IT bits than if the IT block had been executed without an exception return or Debug state exit.

In this case, the instructions at the target of the exception return or Debug state exit must do one of the following:

- Execute as if they passed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as NOPs. That is, they behave as if they failed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as if they were unconditional, or as if the instruction were part of another IT block, in accordance with the behavior in *Branching into an IT block on page K1-11564.*

The remaining iterations of the PSTATE.IT state machine must behave in one of the following ways:

- The PSTATE.IT state machine advances as if it were in an IT block.
- The PSTATE.IT bits are ignored.
- The PSTATE.IT bits are forced to ‘00000000’.

### K1.1.8 Unallocated System register access instructions

From the introduction of the Armv8-A architecture, accesses to unallocated System register encodings are UNDEFINED.

This includes:

- Reads using encodings that are defined as WO.
- Writes using encodings that are defined as RO.
- MCR or MRC accesses to using a set of \{coproc, Crn, opc1, Crm, opc2\} values that the Armv7 architecture defined as UNPREDICTABLE.
• MCRR and MRRC instructions with unallocated values of opc1 or Crm that are described as UNPREDICTABLE are UNDEFINED in the Armv8-A and later architectures.

K1.1.9 SBZ or SBO fields T32 and A32 in instructions

Many of the A32 and T32 instructions have (0) or (1) in the instruction decode to indicate should-be-zero, SBZ, or should-be-one, SBO. If the instruction bit pattern of an instruction is executed with these fields not having the should be values, one of the following must occur:

• The instruction is UNDEFINED.
• The instruction executes as a NOP.
• The instruction operates as if the bit had the should-be value.
• Any destination registers of the instruction become UNKNOWN.

The exceptions to this rule are:
• LDM, LDMIA, LDMFD on page F5-7600.
• LDMDB, LDMEA on page F5-7610.
• LDR (literal) on page F5-7619.
• LDRB (literal) on page F5-7629.
• LDRD (immediate) on page F5-7637.
• LDRD (register) on page F5-7643.
• LDRD (literal) on page F5-7640.
• LDRH (literal) on page F5-7658.
• LDRSB (literal) on page F5-7669.
• LDRSH (literal) on page F5-7680.
• POP on page F5-7789.
• PUSH on page F5-7797.
• SDIV on page F5-7878.
• STM, STMIA, STMEA on page F5-7972.
• STMDB, STMFD on page F5-7980.
• UDIV on page F5-8096.

K1.1.10 UNPREDICTABLE cases in immediate constants in T32 data-processing instructions

The description of immediate constants in T32 data processing Modified immediate constants in T32 instructions on page F1-7242 include constant values that were UNPREDICTABLE in Armv7. Instruction encodings on page F1-7224 describes 32-bit T32 instructions as \{hw1, hw2\}, where hw1 is the left-hand halfword in the 32-bit encoding diagram for the instruction. The UNPREDICTABLE cases are those where both:

• hw2[7:0] == 0b0000000.
• hw1[10] == 0 and either:
  — hw2[14:12] == 0b001.
  — hw2[14:12] == 0b100.
  — hw2[14:12] == 0b111.

From the introduction of the Armv8 architecture, the CONSTRAINED UNPREDICTABLE behavior is that these encodings produce the value 0b0000000.

K1.1.11 UNPREDICTABLE cases in immediate constants in Advanced SIMD instructions

The description of immediate constants in Modified immediate constants in T32 and A32 Advanced SIMD instructions on page F1-7245 include constant values that were UNPREDICTABLE in Armv7. The UNPREDICTABLE cases are those where:

• The bits that the encoding diagram shows as abcd are all 0.
  In the A32 encoding these are bits[24, 18:6, 3:0]. In the T32 encoding they are bits \{hw1[12, 2:0], hw2[3:0]\}.
  The bits that the encoding diagram shows as cmode[3:1] are one of \{0b001, 0b10, 0b11, 0b10\}.
In the A32 encoding these are bits[11:9]. In the T32 encoding they are bits hw2[11:9].

From the introduction of the Armv8 architecture, the CONSTRAINED UNPREDICTABLE behavior is that these encodings produce an immediate constant value of zero.

**K1.1.12 CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values**

The Arm architecture allows copies of System register control or data values to be cached in a cache or TLB. This can lead to CONSTRAINED UNPREDICTABLE behavior if the cache or TLB has not been correctly invalidated following a change of the control or data values.

Unless explicitly stated otherwise, the behavior of the PE is consistent with one of:

- The old data or control value.
- The new data or control value.
- An amalgamation of the old and new data or control values.

In an implementation that includes FEAT_TTCNP, this CONSTRAINED UNPREDICTABLE case can arise from misprogramming when setting TTBR.CnP to 1, as identified in the descriptions of the TTBR.CnP field. In this case, for a particular TTBR, the behavior of the PE is consistent with one of:

- The value of the translation table entry pointed to by that TTBR on one of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.
- An amalgamation of the values of the translation table entries pointed to by that TTBR on two or more of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.

--- Note ---

If the Effective value of a control or data value that determines the behavior of the PE results from the amalgamation of two or more values, then that Effective value must not generate a privilege violation. So, for example:

- Where the CONSTRAINED UNPREDICTABLE behavior occurs because inadequate invalidation of the TLB causes multiple hits in the TLB, the failure to invalidate the TLB by software executing at a given Exception level and Security state must not make it possible to access regions of memory with permissions or attributes that could not be accessed at that Exception level and Security state.

- Where the CONSTRAINED UNPREDICTABLE behavior occurs because of a programming error, on one or more PEs in the Inner Shareable domain, when using a TTBR.CnP value of 1 to share translation table entries, the misprogramming must not make it possible to access regions of memory with permissions or attributes that could not be accessed at the Exception level of that TTBR and the Security state corresponding to the translation table entries being shared.

Alternatively to this CONSTRAINED UNPREDICTABLE behavior, an implementation detecting multiple hits within a TLB might generate an exception, reporting the exception using the TLB Conflict fault code, see TLB conflict aborts on page G5-9218.

The choice between the behaviors might, in some implementations, vary for each use of a control or data value.

**K1.1.13 CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization**

The Arm architecture requires that changes to System registers must be synchronized before they take effect. This can lead to CONSTRAINED UNPREDICTABLE behavior if the synchronization has not been performed.

In these cases, the behavior of the PE is consistent with the unsynchronized control value being either the old value or the new value.

Where multiple control values are updated but not yet synchronized, each control value might independently be the old value or the new value.
In addition, where the unsynchronized control value applies to different areas of functionality, or what an implementation has constructed as different areas of functionality, those areas might independently treat the control value as being either the old value or the new value.

The choice between these behaviors might, in some implementations, vary for each use of a control value.

K1.1.14 Unallocated values with register fields of CP15 registers and Translation Table entries

Unless stated elsewhere, all unallocated or reserved values of fields with allocated values within CP15 registers and Translation Table entries behave in one in one of the following ways:

- The encoding maps onto any of the allocated values but otherwise does not cause UNPREDICTABLE behavior.
- The encodings cause effects that could be achieved by a combination of more than one of the allocated encodings.
- The encodings cause the field to have no functional effect.

K1.1.15 Translation Table Base Address alignment

A misaligned Translation Table Base Address can occur if:

- The VMSAv8-32 Short-descriptor translation table format is enabled and TTBR0[13-N:7], which is defined to be RES0, contains a nonzero value.
- The VMSAv8-32 Long-descriptor translation table format is enabled, and TTBR0[x-1:3], TTBR1[x-1:3], HTTBR[x-1:3], or VTTBR[x-1:3], which are defined to be RES0, contains a nonzero value.

In the event of a misaligned Translation Table Base Address, one of the following behaviors must occur:

- The field that is defined to be RES0 is treated as if all bits were zero:
  - The value that is read back might be the value written or it might be zero.
- The calculation of an address for a translation table walk using that register might be corrupted in those bits that are nonzero.

K1.1.16 Handling of System register control fields for Advanced SIMD and floating-point operation

For historical reasons described in Background to the System register interface on page G1-8994, each of the CPACR, HCPTR, and NSACR has a pair of control fields that were defined to have identical functionality for controlling Advanced SIMD and floating-point operation. These fields are:

- CPACR.{cp10, cp11}.
- HCPTR.{TCP10, TCP11}.
- NSACR.{cp10, cp11}.

The architecture requires that both fields in one of these pairs are programmed to the same value. If this is not done, then the CONSTRAINED UNPREDICTABLE behavior is that behavior is the same as if the cp11 or TCP11 control field was equal to the cp10 or TCP10 field. This is in all respects except for the value read back by a direct read of the register. After a register write that writes different values to the two fields of a pair, a direct read of the register might return an UNKNOWN value for the cp11 or TCP11 field.

Note

This means that, when different values are written to the {cp10, cp11} fields in a single register, the architecture permits but does not require that a read of that register returns the value written to the cp11 field.

CONSTRANGED UNPREDICTABLE CPACR and NSACR settings

If CPACR.cp<n> contains the encoding ‘18’, then one of the following behaviors must occur:

- The encoding maps onto any of the allocated values, but otherwise does not cause UNPREDICTABLE behavior.
- The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.
Note

In Armv7, CPACR had a D32DIS bit, and NSACR had an NSD32DIS bit. There is no CPACR.D32DIS or NSACR.NSD32DIS in Armv8-A and later architectures, and the corresponding bits in the two registers are RES0.

K1.1.17  Mapping of non-idempotent memory locations using the Normal memory type

If non-idempotent memory locations are mapped using the Normal memory type, the state of the non-idempotent memory location may become corrupted in the following circumstances:

- Speculative read accesses may cause accesses to the non-idempotent memory locations that would not occur as part of a simple sequential execution.
- Writes to non-idempotent memory locations might be merged or split. In this case, the number and size of writes seen by the memory location might not be the number and size that occur as part of a simple sequential execution.

K1.1.18  The Performance Monitors Extension

The following subsections describe CONSTRAINED UNPREDICTABLE behaviors when accessing the Performance Monitors Extension in AArch32 state:

- CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER or PMXEVCNTR.
- CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n> and PMEVTYPER<n> on page K1-11570.
- CONSTRAINED UNPREDICTABLE behavior caused by HDCR.HPMN on page K1-11570.

CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER or PMXEVCNTR

If FEAT_FGT is implemented, and EL2 is implemented in the current Security state, and EL1 is using AArch64, permitted access to PMXEVCNTR and PMXEVTYPER are not CONSTRAINED UNPREDICTABLE.

Otherwise, if PMSELR.SEL is greater than the number of event counters accessible at this Exception level, accesses to PMXEVTYPER or PMXEVCNTR can cause CONSTRAINED UNPREDICTABLE behavior. This occurs when one of the following is true:

- If PMSELR.SEL is not equal to 31, and PMSELR.SEL is greater than or equal to PMCR.N, and the PE is executing in EL2 or EL3.
- If FEAT_SEL2 is disabled or is not implemented, PMSELR.SEL is not 31, and PMSELR.SEL is greater than or equal to PMCR.N, and the PE is executing in Secure EL1 or Secure EL0.
- If PMSELR.SEL is not 31, and PMSELR.SEL is greater than or equal to HDCR.HPMN, and the PE is executing in EL1 or EL0.

In these UNPREDICTABLE cases, one of the following behaviors must occur:

- Accesses to PMXEVTYPER or PMXEVCNTR from that mode are UNDEFINED.
- Accesses to PMXEVTYPER or PMXEVCNTR from that mode behave as RAZ/WI.
- Accesses to PMXEVTYPER or PMXEVCNTR from that mode execute as NOPs.
- Accesses to PMXEVTYPER or PMXEVCNTR from that mode behave as if PMSELR.SEL contains an UNKNOWN value that is less than the number of counters accessible at the current Exception level and Security state.
- Accesses to PMXEVTYPER or PMXEVCNTR behave as if PMSELR.SEL is 31.
- If EL2 is implemented and enabled in the current Security state, and PMSELR.SEL is less than the number of accessible event counters but greater than the number of accessible counters at this Exception level, access to PMXEVTYPER or PMXEVCNTR from EL1 or permitted access from EL0 is trapped to EL2.

If PMSELR.SEL is equal to 31, then one of the following behaviors must occur:

- Accesses to PMXEVCNTR are UNDEFINED.
- Accesses to PMXEVCNTR behave as RAZ/WI.
- Accesses to PMXEVCNTR execute as NOPs.
• Accesses to **PMXEVCNTR** behave as if **PMSELR.SEL** contains an **UNKNOWN** value that is less than the number of counters accessible at the current Exception level and Security state.

• If EL2 is implemented and enabled in the current Security state, for an access to **PMXEVCNTR** from EL1 or a permitted access from EL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2.

**Note**

If EL2 is implemented and enabled in the current Security state, **HDCR.HPMN**, or **MDCR_EL2.HPMN**, identifies the number of accessible counters at EL0 or EL1. Otherwise, the number of accessible counters is the number of accessible event counters.

Accesses from EL0 to **PMXEVCNTR** are permitted when:

• EL1 is using AArch32 and the values of **PMUSERENR.{ER, EN}** are both 1.

• EL1 is using AArch64 and the values of **PMUSERENR_EL0.{ER, EN}** are both 1.

Accesses from EL0 to **PMXEVTRYP** are permitted when:

• EL1 is using AArch32 and the value of **PMUSERENR.EN** is 1.

• EL1 is using AArch64 and the value of **PMUSERENR_EL0.EN** is 1.

**CONSTRANGED UNPREDICTABLE accesses to PMEVCNTR<n> and PMEVTYPER<n>**

If **FEAT_FGT** is implemented, and EL2 is implemented in the current Security state, and EL1 is using AArch64, permitted access to **PMEVCNTR<n>** and **PMEVTYPER<n>** are not **CONSTRANGED UNPREDICTABLE**.

Otherwise, if <n> is greater than the number of event counters available in the current Exception level and state, reads and writes of **PMEVCNTR<n>** and **PMEVTYPER<n>** are **CONSTRANGED UNPREDICTABLE**, and the following behaviors are permitted:

• Accesses to the register are **UNDEFINED**.

• Accesses to the register behave as **RAZ/WI**.

• Accesses to the register execute as a **NOP**.

• If EL2 is implemented and enabled in the current Security state, for an access to **PMEVCNTR<n>** or **PMEVTYPER<n>** from EL1 or a permitted access from EL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2.

Accesses from EL0 are permitted to **PMEVCNTR<n>** when:

— EL1 is using AArch32 and the values of **PMUSERENR.{ER, EN}** are both 1.

— EL1 is using AArch64 and the values of **PMUSERENR_EL0.{ER, EN}** are both 1.

Accesses from EL0 are permitted to **PMEVTYPER<n>** when:

— EL1 is using AArch32 and the value of **PMUSERENR.EN** is 1.

— EL1 is using AArch64 and the value of **PMUSERENR_EL0.EN** is 1.

**Note**

If EL2 is implemented and enabled in the current Security state, at EL0 and EL1, **HDCR.HPMN**, or **MDCR_EL2.HPMN**, identifies the number of accessible counters. Otherwise, the number of accessible counters is the number of accessible event counters.

**CONSTRANGED UNPREDICTABLE behavior caused by HDCR.HPMN**

If **PMCR.N** is nonzero, and one of the following is true:

• **FEAT_HPMN0** is not implemented and **HDCR.HPMN** is set to 0.

• **HDCR.HPMN** is set to a value greater than **PMCR.N**.

The **CONSTRANGED UNPREDICTABLE** behavior is:

• The value returned by a direct read of **HDCR.HPMN** is **UNKNOWN**.
• Either:
  — An **UNKNOWN** number of counters are reserved for EL2 use. That is, the PE behaves as if
    `HDCR.HPMN` is set to an **UNKNOWN** nonzero value less than `PMCR.N`.
  — All counters are reserved for EL2 and EL3 use, meaning no counters are accessible from EL1 and EL0.

**K1.1.19 The Activity Monitors Extension**

The following subsections describe **CONSTRAINED UNPREDICTABLE** behaviors when accessing the Activity Monitors
registers in AArch32 state:

- **CONSTRAINED UNPREDICTABLE accesses to AMEVCNTR0<n> and AMEVTYPER0<n>**.

- **CONSTRAINED UNPREDICTABLE accesses to AMEVCNTR1<n> and AMEVTYPER1<n>**.

- **CONSTRAINED UNPREDICTABLE accesses to AMCNTENCLR1 and AMCNTENSET1**.

**CONSTRAINED UNPREDICTABLE accesses to AMEVCNTR0<n> and AMEVTYPER0<n>**

If `<n>` is greater than the number of architectured activity monitor event counters, reads and writes of
`AMEVCNTR0<n>` and `AMEVTYPER0<n>` are **CONSTRAINED UNPREDICTABLE**, and the following behaviors are permitted:

- Accesses to the register are **UNDEFINED**.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.

______ **Note** ________

`AMCGCR.CG0NC` identifies the number of architectured activity monitor event counters.

**CONSTRAINED UNPREDICTABLE accesses to AMEVCNTR1<n> and AMEVTYPER1<n>**

If `<n>` is greater than the number of auxiliary activity monitor event counters, reads and writes of
`AMEVCNTR1<n>` and `AMEVTYPER1<n>` are **CONSTRAINED UNPREDICTABLE**, and the following behaviors are permitted:

- Accesses to the register are **UNDEFINED**.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.

______ **Note** ________

`AMCGCR.CG1NC` identifies the number of auxiliary activity monitor event counters.

**CONSTRAINED UNPREDICTABLE accesses to AMCNTENCLR1 and AMCNTENSET1**

If the number of auxiliary activity monitor event counters that are implemented is zero, reads and writes of
`AMCNTENCLR1` and `AMCNTENSET1` are **CONSTRAINED UNPREDICTABLE**, and the following behaviors are permitted:

- Accesses to the register are **UNDEFINED**.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.

______ **Note** ________

The number of auxiliary activity monitor event counters that are implemented is zero exactly when `AMCFGR.NCG`
is `0b0000`.  

______ **Note** ________
K1.1.20 Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED

When a CONSTRAINED UNPREDICTABLE instruction is treated as UNDEFINED, this generates an exception:

- If this exception is taken to an Exception level that is using AArch64, then ESR_ELx is UNKNOWN.
- If this exception is taken to EL2 and EL2 is using AArch32, then the HSR is unknown.

Note
The value written to ESR or HSR must be consistent with a value that could be created as the result of an exception from the same Exception level that generated the exception, but resulted from a situation that is not CONSTRAINED UNPREDICTABLE at that Exception level. This is to avoid a possible privilege violation.

K1.1.21 Out of range VA

If the PE executes an instruction for which the instruction address, size, and alignment mean it contains the bytes 0xFFFF FFFF and 0x0000 0000, then the bytes that wrap around and appear to be from 0x0000 0000 onwards come from an UNKNOWN address.

If the PE executes a load or store instruction for which the computed address, total access size, and alignment mean it accesses bytes 0xFFFF FFFF and 0x0000 0000, then the bytes that wrap around and appear to be from 0x0000 0000 onwards come from an UNKNOWN address.

K1.1.22 Instruction fetches from Device memory

Instruction fetches from Device memory are CONSTRAINED UNPREDICTABLE.

If a location in memory has the Device attribute and is not marked as execute-never, then an implementation might perform speculative instruction accesses to this memory location when address translation is enabled.

If a branch causes the program counter to point to a location in memory with the Device attribute that is not marked as execute-never for the current Exception level for instruction fetches, then an implementation must perform one of the following behaviors:

- It treats the instruction fetch as if it were to a memory location with the Normal, Non-cacheable attribute.
- It generates a Permission fault.

K1.1.23 Multi-access instructions that load the PC from Device memory

Multi-access instructions that load the PC from Device memory when address translation is enabled are UNPREDICTABLE in AArch32 state. From the introduction of the Armv8-A architecture in AArch32 state an implementation must perform one of the following behaviors:

- It loads the PC from the memory location as if the memory location had the Normal Non-cacheable attribute.
- It generates a Permission fault.

K1.1.24 Programming CSSEL.R.Level for a cache level that is not implemented

If CSSEL.R.Level is programmed to a cache level that is not implemented, then a read of CSSEL.R returns an UNKNOWN value in CSSEL.R.Level.

If CSSEL.R.Level is programmed to a cache level that is not implemented, then on a read of CCSIDR an implementation must perform one of the following behaviors:

- The CCSIDR read is treated as a NOP.
- The CCSIDR read is UNDEFINED.
- The CCSIDR read returns an UNKNOWN value.
When `FEAT_CCIDX` is implemented, `CCSIDR2` is implemented. If `CSSEL.R.Level` is programmed to a cache level that is not implemented, then on a read of `CCSIDR2` an implementation must perform one of the following behaviors:

- The `CCSIDR2` read is treated as a NOP.
- The `CCSIDR2` read is UNDEFINED.
- The `CCSIDR2` read returns an UNKNOWN value.

**K1.1.25 Crossing a page boundary with different memory types or Shareability attributes**

A memory access from a load or store instruction that crosses a page boundary to a memory location that has a different memory type or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- Each memory access generated by the instruction uses the memory type and Shareability attribute associated with its own address.
- The instruction generates an alignment fault caused by the memory type.

For the Non-secure PL1&0 translation regime:

- If the stage 1 translation causes the mismatch, the resulting exception is taken to PL1.
- If the stage 2 translation causes the mismatch, the resulting exception is taken to PL2.
- If both stages of translation cause the mismatch, the resulting exception can be taken to either PL1 or PL2.
- The instruction executes as a NOP.

**K1.1.26 Crossing a 4KB boundary with a Device access**

A memory access from a load or store instruction to Device memory that crosses a 4KB boundary results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- All memory accesses generated by the instruction are performed as if the presence of the boundary had no effect on the memory accesses.
- All memory accesses generated by the instruction are performed as if the presence of the boundary had no effect on the memory accesses, except that there is no guarantee of ordering between memory accesses.
- The instruction generates an Alignment fault caused by the memory type.

For the Non-secure PL1&0 translation regime:

- If the stage 1 translation causes the boundary to be crossed, the resulting exception is taken to PL1.
- If the stage 2 translation causes the boundary to be crossed, the resulting exception is taken to PL2.
- If both stages of translation cause the boundary to be crossed, the resulting exception can be taken to either PL1 or PL2.
- The instruction executes as a NOP.

**Note**

The boundary referred to is between two Device memory regions that are both of 4KB and aligned to 4KB.

**K1.1.27 UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs**

`Load-Exclusive and Store-Exclusive instruction usage restrictions on page E2-7217` defines a Load-Exclusive/Store-Exclusive pair, and identifies various CONSTRAINED UNPREDICTABLE behaviors associated with using Load-Exclusive/Store-Exclusive pairs. These cases were UNPREDICTABLE in Armv7. In summary, these cases are:

- The target virtual address of a `StoreExcl` instruction is different from the virtual address of the preceding `LoadExcl` instruction in the same thread of execution.
- The transaction size of a `StoreExcl` instruction is different from the transaction size of the preceding `LoadExcl` instruction in the same thread of execution.
The memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, either:

- Because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction.
- Because the `LoadExcl` instruction and the `StoreExcl` instruction use different virtual addresses, with different attributes, that point to the same physical address.

In addition, the effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE.

See the descriptions in *Load-Exclusive and Store-Exclusive instruction usage restrictions on page E2-7217* for the permitted behavior in each of these cases, and any constraints that might apply to whether the case is CONSTRAINED UNPREDICTABLE.

---

**Note**

Additional CONSTRAINED UNPREDICTABLE cases can apply to Load-Exclusive and Store-Exclusive instructions, see *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 System instructions in the base instruction set on page K1-11575*.

---

**K1.1.28**  CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings

The A32 and T32 instruction sets include encodings that result in CONSTRAINED UNPREDICTABLE behavior when they are decoded.

**CONSTRAINED UNPREDICTABLE behavior of CRC32 instruction encodings**

In the A32 and T32 instruction sets, there are encodings of the CRC32 and CRC32C instructions that result in CONSTRAINED UNPREDICTABLE behavior. These encodings are listed in the following places in the A32 and T32 instruction sets:

- *Cyclic Redundancy Check on page F4-7382* for the A32 instruction set, with \(sz = 11\).
- *Data-processing (two source registers) on page F3-7367* for the T32 instruction set, with \(\text{op1} = 10\) and \(\text{op2} = 11\).

The CONSTRAINED UNPREDICTABLE behavior for these encodings is described in *CRC32 on page F5-7540* and *CRC32C on page F5-7543*.

**CONSTRAINED UNPREDICTABLE behavior of other A32 instruction encodings**

In the A32 instruction set, there are encodings that result in CONSTRAINED UNPREDICTABLE behavior. These encodings are listed in:

- *Miscellaneous on page F4-7421*.
- *Memory hints and barriers on page F4-7431*.
- *Barriers on page F4-7432*.

The CONSTRAINED UNPREDICTABLE behavior is that an implementation must treat the encodings in one of the following ways:

- The instruction encoding is UNDEFINED.
- The instruction encoding executes as a NOP.

**K1.1.29**  Out of range values of the Set/Way/Index fields in cache maintenance instructions

In the cache maintenance by set/way instructions DCCISW, DCCSW, and DCISW, if any set/way/index argument is larger than the value supported by the implementation, then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
• The instruction performs cache maintenance on one of:
  — No cache lines.
  — A single arbitrary cache line.
  — Multiple arbitrary cache lines.

—— Note ————
This CONSTRAINED UNPREDICTABLE behavior applies, also, to the A64 cache maintenance by set/way instructions DC CISW, DC CSW, and DC ISW.

K1.1.30 CONSTRAINED UNPREDICTABLE behavior for A32 and T32 System instructions in the base instruction set

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different A32 and T32 System instructions.

—— Note ————
If an instruction can result in CONSTRAINED UNPREDICTABLE behavior that is not specific to that particular instruction, see the relevant section in this appendix for a description of the CONSTRAINED UNPREDICTABLE behavior.

SRS (T32)
For a description of this instruction and the encoding, see SRS, SRSDA, SRSDB, SRSIA, SRSIB on page F5-7936.

CONSTRAINED UNPREDICTABLE behavior
For all encodings:
• If the instruction specifies an illegal mode field, then one of the following behaviors must occur:
  — The instruction is UNDEFINED.
  — The instruction executes as a NOP.
  — R13 of the current mode is used.
  — The store occurs to an UNKNOWN address, and if the instruction specifies writeback, any general-purpose register that can be accessed without privilege violation from the current Exception level become UNKNOWN.

SRS (A32)
For a description of this instruction and the encoding, see SRS, SRSDA, SRSDB, SRSIA, SRSIB on page F5-7936.

CONSTRAINED UNPREDICTABLE behavior
For all encodings:
• If the instruction specifies an illegal mode field, then one of the following behaviors must occur:
  — The instruction is UNDEFINED.
  — The instruction executes as a NOP.
  — R13 of the current mode is used.
  — The store occurs to an UNKNOWN address, and if the instruction specifies writeback, any general-purpose register that can be accessed without privilege violation from the current Exception level become UNKNOWN.

SUBS PC, LR and related instructions (T32)
For a description of this instruction and the encoding, see the exception return form of SUB, SUBS (immediate) on page F5-8039.
**CONSTRAINED UNPREDICTABLE behavior**

For all encodings:

- If this instruction is executed in User mode or in System mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
- If the instruction transfers an illegal mode encoding to PSTATE.M, then this invokes the illegal exception return.

---

**Note**

An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.

---

For encoding T5:

- If hw1[3:0] are not 0b1110, and the instruction is executed when not in Hyp mode, System mode, or User mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction is treated as a NOP.
  - The instruction is treated as if hw1[3:0] are 0b1110.
  - The program counter is set using the value in the register specified by hw1[3:0].

**SUBS PC. LR and related instructions (A32)**

For a description of this instruction and the encoding, see the exception return forms of *MOV, MOVs (register)* on page F5-7719 and *SUB, SUBs (immediate)* on page F5-8039.

**CONSTRAINED UNPREDICTABLE behavior**

For all encodings:

- If this instruction is executed in User mode or in System mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
- If the instruction transfers an illegal mode encoding to PSTATE.M, then this invokes the illegal exception return.

---

**Note**

An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.
K1.1.31 CONSTRUCTED UNPREDICTABLE behavior, A32 and T32 Advanced SIMD and floating-point instructions

This section lists the CONSTRUCTED UNPREDICTABLE behavior for the different A32 and T32 Advanced SIMD and floating-point instructions listed in Alphabetical list of Advanced SIMD and floating-point instructions on page F6-8166.

---

**Note**

- The pseudocode used in this section to describe cases that can result in CONSTRUCTED UNPREDICTABLE behavior does not necessarily match the encoding specific pseudocode for a specific instruction.

- If an instruction can result in CONSTRUCTED UNPREDICTABLE behavior that is not specific to that particular instruction, see the relevant section in this appendix for a description of the CONSTRUCTED UNPREDICTABLE behavior.

---

**VCVT (between floating-point and fixed-point)**

For a description of this instruction and the encoding, see [VCVT (between floating-point and fixed-point, floating-point)] on page F6-8329.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRUCTED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRUCTED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRUCTED UNPREDICTABLE.

**VLD1 (multiple single elements)**

For a description of this instruction and the encoding, see [VLD1 (multiple single elements)] on page F6-8429.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRUCTED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRUCTED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRUCTED UNPREDICTABLE.

**VLD1 (single element to all lanes)**

For a description of this instruction and the encoding, see [VLD1 (single element to all lanes)] on page F6-8426.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRUCTED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRUCTED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRUCTED UNPREDICTABLE.

**VLD2 (multiple 2-element structures)**

For a description of this instruction and the encoding, see [VLD2 (multiple 2-element structures)] on page F6-8445.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRUCTED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRUCTED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRUCTED UNPREDICTABLE.

**VLD2 (single 2-element structure to one lane)**

For a description of this instruction and the encoding, see [VLD2 (single 2-element structure to one lane)] on page F6-8436.
If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VLD2 (single 2-element structure to all lanes)**

For a description of this instruction and the encoding, see *VLD2 (single 2-element structure to all lanes)* on page F6-8442.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VLD3 (multiple 3-element structures)**

For a description of this instruction and the encoding, see *VLD3 (multiple 3-element structures)* on page F6-8459.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VLD3 (single 3-element structure to one lane)**

For a description of this instruction and the encoding, see *VLD3 (single 3-element structure to one lane)* on page F6-8450.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VLD3 (single 3-element structure to all lanes)**

For a description of this instruction and the encoding, see *VLD3 (single 3-element structure to all lanes)* on page F6-8456.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VLD4 (multiple 4-element structures)**

For a description of this instruction and the encoding, see *VLD4 (multiple 4-element structures)* on page F6-8472.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VLD4 (single 4-element structure to one lane)**

For a description of this instruction and the encoding, see *VLD4 (single 4-element structure to one lane)* on page F6-8462.
If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VLD4 (single 4-element structure to all lanes)**

For a description of this instruction and the encoding, see VLD4 (single 4-element structure to all lanes) on page F6-8468.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VLDM**

For a description of this instruction and the encoding, see VLDM, VLDMDB, VLDMIA on page F6-8476.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VMOV (between two general-purpose registers and two single-precision registers)**

For a description of this instruction and the encoding, see VMOV (between two general-purpose registers and two single-precision registers) on page F6-8556.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VMOV (between two general-purpose registers and a doubleword floating-point register)**

For a description of this instruction and the encoding, see VMOV (between two general-purpose registers and a doubleword floating-point register) on page F6-8535.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VST1 (multiple single elements)**

For a description of this instruction and the encoding, see VST1 (multiple single elements) on page F6-8803.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VST2 (multiple 2-element structures)**

For a description of this instruction and the encoding, see VST2 (multiple 2-element structures) on page F6-8817.
If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VST2 (single 2-element structure from one lane)**

For a description of this instruction and the encoding, see *VST2 (single 2-element structure from one lane)* on page F6-8811.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VST3 (multiple 3-element structures)**

For a description of this instruction and the encoding, see *VST3 (multiple 3-element structures)* on page F6-8828.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VST3 (single 3-element structure from one lane)**

For a description of this instruction and the encoding, see *VST3 (single 3-element structure from one lane)* on page F6-8822.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VST4 (multiple 4-element structures)**

For a description of this instruction and the encoding, see *VST4 (multiple 4-element structures)* on page F6-8838.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VST4 (single 4-element structure from one lane)**

For a description of this instruction and the encoding, see *VST4 (single 4-element structure from one lane)* on page F6-8831.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

**VSTM**

For a description of this instruction and the encoding, see *VSTM, VSTMDB, VSTMIA* on page F6-8842.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.
K1.1.32 CONSTRAINED UNPREDICTABLE behaviors associated with the VTCR

The following subsections describe the CONSTRAINED UNPREDICTABLE behavior associated with programming the VTCR:

- **Misprogramming VTCR.S.**
- **Misprogramming VTCR.{SL0, T0SZ}**

### Misprogramming VTCR.S

VTCR.S must be programmed to the value of T0SZ[3], or the effect is CONSTRAINED UNPREDICTABLE. From the introduction of the Armv8-A architecture, if VTCR.S is not programmed correctly, then the VTCR.T0SZ value is treated as an UNKNOWN value.

- **Note**

The CONSTRAINED UNPREDICTABLE behavior described in **Misprogramming VTCR.{SL0, T0SZ)** means the UNKNOWN VTCR.T0SZ value might generate a Translation fault.

### Misprogramming VTCR.{SL0, T0SZ}

If the stage 2 input address size, as programmed in VTCR.T0SZ, is out of range with respect to the starting level, as programmed in the VTCR.SL0 field, or the VTCR.SL0 field is programmed to a reserved value, then at the time of a translation walk that uses the stage 2 translation, a stage 2 level 1 Translation Fault is generated.

K1.1.33 CONSTRAINED UNPREDICTABLE behavior of EL2 features

The following sections, and the instruction descriptions, describe CONSTRAINED UNPREDICTABLE behavior that can occur in an implementation that includes EL2 where EL2 can use AArch32:

- **ERET in User mode or System mode.**
- **Accessing Hyp mode from outside Hyp mode.**
- **Modifying PSTATE.M when in Hyp mode.**
- **Use of Hyp mode in Secure state on page K1-11582.**
- **Exception return to Hyp mode on page K1-11582.**
- **Stage 1 default memory type on page K1-11582.**
- **Trapping of general exceptions to Hyp mode on page K1-11582.**
- **MSR (banked register) and MRS (banked register) on page K1-11582.**

### ERET in User mode or System mode

If ERET is executed in User mode or System mode, it behaves as described in **SUBS PC, LR and related instructions (T32)** on page K1-11575.

### Accessing Hyp mode from outside Hyp mode

Attempting to change into Hyp mode or out of Hyp mode using the MSR or CPS instruction invokes the illegal exception return by not changing the mode, and setting PSTATE.IL to 1.

SRS using the Hyp mode SP from Non-secure modes other than Hyp mode, or from Secure state, is handled as described in **SRS (T32)** on page K1-11575 and **SRS (A32)** on page K1-11575.

### Modifying PSTATE.M when in Hyp mode

Attempting to change into Hyp mode or out of Hyp mode using the MSR or CPS instruction invokes the illegal exception return by not changing the mode, and setting PSTATE.IL to 1.

SRS using the Hyp mode SP from Non-secure modes other than Hyp mode, or from Secure state, is handled as described in **SRS (T32)** on page K1-11575 and **SRS (A32)** on page K1-11575.
Architectural Constraints on UNPREDICTABLE Behaviors

K1.1 AArch32 CONSTRAINED UNPREDICTABLE behaviors

Use of Hyp mode in Secure state

Attempting to change into Hyp mode or out of Hyp mode using the MSR or CPS instruction invokes the illegal exception return by not changing the mode, and setting PSTATE.IL to 1.

SRS using the Hyp mode SP from Non-secure modes other than Hyp mode, or from Secure state, is handled as described in SRS (T32) on page K1-11575 and SRS (A32) on page K1-11575.

Exception return to Hyp mode

Exception returns to Hyp mode when SCR.NS == 0 or from a Non-secure PL1 mode invokes the illegal exception return.

Stage 1 default memory type

If HCR.DC == 1, then the behavior of the PE when executing in a Non-secure mode other than Hyp mode is consistent with:

- SCTLR.M == 0, regardless of the actual value of SCTLR.M, other than for the value returned by an explicit read of SCTLR.M.
- HCR.VM == 1, regardless of the actual value of HCR.VM, other than for an explicit read of this bit.

Trapping of general exceptions to Hyp mode

Attempting to perform an exception return to a Non-secure PL1 mode when HCR.TGE == 1 invokes an illegal exception return.

Attempting to change from Monitor mode to a Non-secure PL1 mode when HCR.TGE == 1 by executing a CPS or MSR instruction generates an Illegal Execution state exception, by not changing the mode, and setting PSTATE.IL to 1.

When EL3 is using AArch32, attempting to change from a Secure PL1 mode to a Non-secure PL1 mode when HCR.TGE is set, by changing SCR.NS from 0 to 1, results in no change of SCR.NS.

Because taking an exception into Non-secure PL1 modes leads to a CONSTRAINED UNPREDICTABLE situation, the following additional properties apply when HCR.TGE == 1:

- All exceptions that would be routed to EL1 are routed to EL2.
- Non-secure SCTLR.M is treated as being 0, regardless of its actual value, other than for an explicit read of this bit.
- HCR.FMO, HCR.IMO, and HCR.AMO are treated as being 1, regardless of their actual value, other than for an explicit read of these bits.
- All virtual interrupts are disabled.
- Any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts are disabled.

MSR (banked register) and MRS (banked register)

If the target register specified by the {R, SYSm} fields of the instruction encoding is not accessible from the PE mode in which the instruction was executed (see Usage restrictions on the banked register transfer instructions on page F5-8161), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- For MSR (banked register) instructions, the destination general-purpose register becomes UNKNOWN.
- For MSR (banked register) instructions, if the register specified could be accessed from the current mode by other mechanisms, then this register is UNKNOWN. Otherwise, the instruction is a NOP.

If the instruction was executed specifying an unallocated {R, SYSm} field value or an unimplemented register (see Encoding the register argument in the banked register transfer instructions on page F5-8162), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
• The instruction executes as a NOP.
• An allocated MRS (banked register) or MSR (banked register) instruction is executed.

**K1.1.34 Reserved values in System and memory-mapped registers and translation table entries**

Unless otherwise stated, all unallocated or reserved values of fields with allocated values within the AArch32 System registers, memory-mapped registers, and translation table entries behave in one of the following ways:

• The encoding maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.
• The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.
• The encoding causes the field to have no functional effect.

**Note**

These constraints are identical to those for the equivalent AArch64 definitions, as given in Reserved values in System and memory-mapped registers and translation table entries on page K1-11600.

Unless otherwise stated, when a direct write of a System register or memory-mapped register writes an unallocated value to a field, if that value is unallocated in all contexts then a subsequent read of the field returns an UNKNOWN value.

**K1.1.35 CONSTRAINED UNPREDICTABLE behavior in Debug state**

*Behavior in Debug state on page H2-10242* of this manual describes the CONSTRAINED UNPREDICTABLE behaviors that are specifically associated with Debug state.
K1.2 AArch64 CONSTRAINED UNPREDICTABLE behaviors

It contains the following sections:

- Overview of the constraints on AArch64 UNPREDICTABLE behaviors.
- SBZ or SBO fields in A64 instructions.
- CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values on page K1-11585.
- CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization on page K1-11585.
- Translation table base address alignment on page K1-11586.
- The Performance Monitors Extension on page K1-11586.
- The Activity Monitors Extension on page K1-11588.
- Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED on page K1-11588.
- Out of range virtual address on page K1-11588.
- Mapping of non-idempotent memory locations using the Normal memory type on page K1-11589.
- Instruction fetches from Device memory on page K1-11589.
- Programming the CSSELRL_EL1.[Level, InD, TnD] for a cache level that is not implemented on page K1-11589.
- Crossing a page boundary with different memory types or Shareability attributes on page K1-11589.
- Crossing a peripheral boundary with a Device access on page K1-11590.
- CONSTRAINED UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs on page K1-11591.
- CONSTRAINED UNPREDICTABLE behavior for A64 instructions on page K1-11591.
- Out of range values of the Set/ Way/ Index fields in cache maintenance instructions on page K1-11600.
- Reserved values in System and memory-mapped registers and translation table entries on page K1-11600.
- CONSTRAINED UNPREDICTABLE behavior in Debug state on page K1-11601.

K1.2.1 Overview of the constraints on AArch64 UNPREDICTABLE behaviors

The term UNPREDICTABLE describes a number of cases where the architecture has a feature that software must not use. For execution in AArch64 state, the Armv8-A and later architectures specify a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

Note
Software designed to be compatible with the Armv8-A and later architectures must not rely on these CONSTRAINED UNPREDICTABLE cases being handled in any way other than those listed under the heading CONSTRAINED UNPREDICTABLE.

K1.2.2 SBZ or SBO fields in A64 instructions

Some A64 instructions have (0) or (1) in the instruction decode to indicate should-be-zero, SBZ, or should-be-one, SBO, as described in Fixed values in AArch64 instruction and System register descriptions on page C2-243. Except for specific cases identified in CONSTRAINED UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs on page K1-11591, if the instruction bit pattern of an instruction is executed with these fields not having the should be values, one of the following must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction operates as if the bit had the should-be value.
- Any destination registers of the instruction become UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.
### K1.2.3 CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values

The Arm architecture allows copies of control values or data values to be cached in a cache or TLB. This can lead to UNPREDICTABLE behavior if the cache or TLB has not been correctly invalidated following a change of the control or data values.

Unless explicitly stated otherwise, the behavior of the PE is consistent with one of:

- The old data or control value.
- The new data or control value.
- An amalgamation of the old and new data or control values.

In an implementation that includes FEAT_TTCNP, this CONSTRAINED UNPREDICTABLE case can arise from misprogramming when setting TTBR.CnP to 1, as identified in the descriptions of the TTBR.CnP field. In this case, for a particular TTBR, the behavior of the PE is consistent with one of:

- The value of the translation table entry pointed to by that TTBR on one of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.
- An amalgamation of the values of the translation table entries pointed to by that TTBR on two or more of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.

---

**Note**

If the *Effective value* of a control or data value that determines the behavior of the PE results from the amalgamation of two or more values, then that *Effective value* must not generate a privilege violation. So, for example:

- Where the CONSTRAINED UNPREDICTABLE behavior occurs because inadequate invalidation of the TLB causes multiple hits in the TLB, the failure to invalidate the TLB by software executing at a given Exception level and Security state must not make it possible to access regions of memory with permissions or attributes that could not be accessed at that Exception level and Security state.

- Where the CONSTRAINED UNPREDICTABLE behavior occurs because of a programming error, on one or more PEs in the Inner Shareable domain, when using a TTBR.CnP value of 1 to share translation table entries, the misprogramming must not make it possible to access regions of memory with permissions or attributes that could not be accessed at the Exception level of that TTBR and the Security state corresponding to the translation table entries being shared.

---

Alternatively to this CONSTRAINED UNPREDICTABLE behavior, an implementation detecting multiple hits in a TLB might generate an exception, reporting the exception using the TLB conflict fault code, see *TLB conflict abort* on page D8-5183.

The choice between the behaviors might, in some implementations, vary for each use of a control or data value.

### K1.2.4 CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization

The Arm architecture requires that changes to System registers must be synchronized before they take effect. This can lead to UNPREDICTABLE behavior if the synchronization has not been performed.

In these cases, the behavior of the PE is consistent with the unsynchronized control value being either the old value or the new value.

Where multiple control values are updated but not yet synchronized, each control value might independently be the old value or the new value.

In addition, where the unsynchronized control value applies to different areas of functionality, or what an implementation has constructed as different areas of functionality, those areas might independently treat the control value as being either the old value or the new value.

The choice between these behaviors might, in some implementations, vary for each use of a control value.
K1.2.5   Translation table base address alignment

In the translation table base registers TTBR0_EL1, TTBR1_EL1, TTBR0_EL2, VTTBR_EL2, and TTBR0_EL3, register bits[48:x] hold the translation table base address, where x depends on the translation table granule size and the size of the addressed translation table, as described in Translation granules on page D8-5081. Register bits[(x-1):0], unless redefined for another purpose, correspond to bits[(x-1):0] of the translation table base address and therefore are RES0.

Note

• When FEAT_LPA is implemented and the 64KB granule size is used, register bits[5:2] are redefined to hold bits[51:48] of the translation table base address.

• When FEAT_TTCNP is implemented register bit[0] is redefined as the CnP bit.

For these registers, if one or more RES0 bits in register bits [(x-1):0] does not have a value of 0, this can result in a misaligned translation table base address. In this case, one of the following behaviors must occur:

• The field that is defined to be RES0 is treated as if all the bits had a value of 0:
  — The value read back might be the value written or it might be zero.

• The calculation of an address for a translation table walk using those registers might be corrupted in those bits that are nonzero.

For more information, see the appropriate TTBR.BADDR field description.

K1.2.6   The Performance Monitors Extension

The following subsections describe CONSTRAINED UNPREDICTABLE behaviors when accessing the Performance Monitors Extension in AArch64 state:

• CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPE_EL0 or PMXEVTYPE_EL0.
• CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n>_EL0 and PMEVTYPE<n>_EL0 on page K1-11587.
• CONSTRAINED UNPREDICTABLE behavior caused by MDCR_EL2.HPMN on page K1-11587.

CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPE_EL0 or PMXEVTYPE_EL0

If FEAT_FGT is implemented, and EL2 is implemented in the current Security state, and EL1 is using AArch64, permitted access to PMXEVTYPE_EL0 and PMXEVCNTR_EL0 are not CONSTRAINED UNPREDICTABLE.

Otherwise, if PMSELR_EL0.SEL is greater than the number of event counters accessible at this Exception level, accesses to PMXEVTYPE_EL0 and PMXEVCNTR_EL0 can cause CONSTRAINED UNPREDICTABLE behavior. This occurs when one of the following is true:

• If PMSELR_EL0.SEL is not equal to 31, and PMSELR_EL0.SEL is greater than or equal to PMCR_EL0.N, and the PE is executing in EL2 or EL3.

• If FEAT_SEL2 is disabled or is not implemented, PMSELR_EL0.SEL is not 31, and PMSELR_EL0.SEL is greater than or equal to PMCR_EL0.N, and the PE is executing in Secure EL1 or Secure EL0.

• If PMSELR_EL0.SEL is not 31, and PMSELR_EL0.SEL is greater than or equal to MDCR_EL2.HPMN, and the PE is executing in EL0 or EL1.

In these cases, one of the following behaviors must occur:

• Accesses to PMXEVTYPE_EL0 or PMXEVCNTR_EL0 from that state are UNDEFINED.

• Accesses to PMXEVTYPE_EL0 or PMXEVCNTR_EL0 from that state behave as RAZ/WI.

• Accesses to PMXEVTYPE_EL0 or PMXEVCNTR_EL0 from that state execute as NOPs.

• Accesses to PMXEVTYPE_EL0 or PMXEVCNTR_EL0 from that state behave as if PMSELR_EL0.SEL contains an UNKNOWN value that is less than the number of counters accessible at the current Exception level and Security state.

• Accesses to PMXEVTYPE_EL0 or PMXEVCNTR_EL0 from that state behave as if PMSELR_EL0.SEL is 31.
• If EL2 is implemented and enabled in the current Security state, and PMSELR_EL0.SEL is less than the number of accessible event counters but greater than or equal to the number of accessible counters at the current Exception level, access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from EL1 or a permitted access from EL0 is trapped to EL2.

**Note**
If EL2 is implemented and enabled in the current Security state, MDCR_EL2.HPMN identifies the number of accessible event counters.

Accesses from EL0 to PMXEVCNTR_EL0 are permitted when:
- EL1 is using AArch32 and the values of PMUSERENR.{ER, EN} are both 1.
- EL1 is using AArch64 and the values of PMUSERENR_EL0.{ER, EN} are both 1.
Accesses from EL0 to PMXEVTYPER_EL0 are permitted when:
- EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
- EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

If PMSELR_EL0.SEL is equal to 31, then one of the following behaviors must occur:
• Accesses to PMXEVCNTR_EL0 are UNDEFINED.
• Accesses to PMXEVCNTR_EL0 behave as RAZ/WI.
• Accesses to PMXEVCNTR_EL0 execute as NOPs.
• Accesses to PMXEVCNTR_EL0 behave as if PMSELR_EL0.SEL contains an unknown value that is less than the number of counters accessible at the current Exception level and Security state.

**CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0**

If FEAT_FGT is implemented, and EL2 is implemented in the current Security state, and EL1 is using AArch64, permitted access to PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0 are not CONSTRAINED UNPREDICTABLE.

Otherwise, if <n> is greater than the number of event counters available in the current Exception level and state, reads and writes of PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0 are CONSTRAINED UNPREDICTABLE, the following behaviors are permitted:
• Accesses to the register are UNDEFINED.
• Accesses to the register behave as RAZ/WI.
• Accesses to the register execute as NOPs.
• If EL2 is implemented and enabled in the current Security state, for an access to PMEVCNTR<n>_EL0 or PMEVTYPER<n>_EL0 from EL1 or a permitted access from EL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2.
Accesses from EL0 to PMEVCNTR<n>_EL0 are permitted when:
- EL1 is using AArch32 and the value of PMUSERENR.{ER, EN} are both 1.
- EL1 is using AArch64 and the value of PMUSERENR_EL0.{ER, EN} are both 1.
Accesses from EL0 to PMEVTYPER<n>_EL0 are permitted when:
- EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
- EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

**CONSTRAINED UNPREDICTABLE behavior caused by MDCR_EL2.HPMN**

If PMCR_EL0.N is nonzero, and one of the following is true:
• FEAT_HPMN0 is not implemented and MDCR_EL2.HPMN is set to 0.
• MDCR_EL2.HPMN is set to a value greater than PMCR_EL0.N.

The following CONSTRAINED UNPREDICTABLE behavior applies:
• The value returned by a direct read of MDCR_EL2.HPMN is UNKNOWN.
• Either:
  — An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if
    MDCR_EL2.HPMN is set to an UNKNOWN nonzero value less than PMCR_EL0.N.
  — All counters are reserved for EL2 and EL3 use, meaning no counters are accessible from EL1 and EL0.

K1.2.7 The Activity Monitors Extension

If \(<n>\) is greater than the number of architected activity monitor event counters, reads and writes of
AMEVCNTR0<\(n>\)_EL0 and AMEVTYPE0<\(n>\)_EL0 are CONSTRAINED UNPREDICTABLE, and the following
behaviors are permitted:
• Accesses to the register are UNDEFINED.
• Accesses to the register behave as RAZ/WI.
• Accesses to the register execute as a NOP.

—— Note ————
AMCGCR_EL0.CG0NC identifies the number of architected activity monitor event counters.

——— Note ———
If \(<n>\) is greater than the number of auxiliary activity monitor event counters, reads and writes of
AMEVCNTR1<\(n>\)_EL0 and AMEVTYPE1<\(n>\)_EL0 are CONSTRAINED UNPREDICTABLE, and the following
behaviors are permitted:
• Accesses to the register are UNDEFINED.
• Accesses to the register behave as RAZ/WI.
• Accesses to the register execute as a NOP.

——— Note ———
AMCGCR_EL0.CG1NC identifies the number of auxiliary activity monitor event counters.

If the number of auxiliary activity monitor event counters that are implemented is zero, reads and writes of
AMCNTENCLR1_EL0 and AMCNTENSET0_EL0 are CONSTRAINED UNPREDICTABLE, and the following
behaviors are permitted:
• Accesses to the register are UNDEFINED.
• Accesses to the register behave as RAZ/WI.
• Accesses to the register execute as a NOP.

——— Note ———
The number of auxiliary activity monitor event counters that are implemented is zero exactly when
AMCFGR_EL0.NCG is 0b0000.

K1.2.8 Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as
UNDEFINED

When a CONSTRAINED UNPREDICTABLE instruction is treated as UNDEFINED, ESR_ELx is UNKNOWN.

—— Note ————
The value written to ESR_ELx must be consistent with a value that could be created as the result of an exception
from the same Exception level that generated the exception, but was the result of a situation that is not CONSTRAINED
UNPREDICTABLE at that Exception level. This is to avoid a possible privilege violation.

K1.2.9 Out of range virtual address

If the PE executes a load or store instruction with tagged addressing disabled in the current translation regime, and
where the computed virtual address, total access size, and alignment mean that it accesses the bytes at 0xFFFF FFFF
FFFF FFFF and 0x0000 0000 0000 0000, then the bytes that appear to be from 0x0000 0000 0000 0000 onwards are
accessed at an UNKNOWN address.
If the PE executes a load or store instruction with tagged addressing enabled in the current translation regime, and where the computed address, total access size, and alignment mean that it accesses the bytes at 0xFFFF FFFF FFFF FFFF and 0x0000 0000 0000 0000, then the bytes that appear to be from 0x0000 0000 0000 0000 onwards are accessed at an unknown address and the tags associated with address also become unknown.

Note

Because of program counter alignment constraints, it is impossible for a PE to fetch an A64 instruction that includes both the byte at virtual address 0xFFFF FFFF FFFF FFFF and the byte at virtual address 0x0000 0000 0000 0000.

### K1.2.10 Mapping of non-idempotent memory locations using the Normal memory type

If non-idempotent memory locations are mapped using the Normal memory type, the state of the non-idempotent memory location may become corrupted in following circumstances:

- Speculative read accesses may cause accesses to the non-idempotent memory locations that would not occur as part of a simple sequential execution.
- Writes to non-idempotent memory locations might be merged or split. In this case, the number and size of writes seen by the memory location might not be the number and size that occur as part of a simple sequential execution.

### K1.2.11 Instruction fetches from Device memory

Instruction fetches from Device memory are CONSTRAINED UNPREDICTABLE.

If a location in memory has the Device attribute and is not marked as execute-never, then an implementation might perform speculative instruction accesses to this memory location at times when address translation is enabled.

For an instruction fetch from a memory location with the Device attribute that is not marked as execute-never for the current Exception level, an implementation can either:

- Treat the instruction fetch as if it were to a memory location with the Normal, Non-cacheable attribute.
- Take a Permission fault.

### K1.2.12 Programming the CSSELR_EL1.{Level, InD, TnD} for a cache level that is not implemented

If the CSSELR_EL1.{Level, InD, TnD} is programmed to a cache level that is not implemented, then a read of CSSELR_EL1 returns an UNKNOWN value in CSSELR_EL1.{Level, InD, TnD}.

If CSSELR_EL1.{Level, InD, TnD} is programmed to a cache level that is not implemented, then on a read of CCSIDR_EL1 an implementation must perform one of the following behaviors:

- The CCSIDR_EL1 read is treated as a NOP.
- The CCSIDR_EL1 read is UNDEFINED.
- The CCSIDR_EL1 read returns an UNKNOWN value.

When FEAT_CCIDX is implemented, CCSIDR2_EL1 is implemented. If CSSELR_EL1.{Level, InD, TnD} is programmed to a cache level that is not implemented, then on a read of CCSIDR2_EL1 an implementation must perform one of the following behaviors:

- The CCSIDR2_EL1 read is treated as a NOP.
- The CCSIDR2_EL1 read is UNDEFINED.
- The CCSIDR2_EL1 read returns an UNKNOWN value.

### K1.2.13 Crossing a page boundary with different memory types or Shareability attributes

A memory access from a load or store instruction that crosses a page boundary to a memory location that has a different memory type or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- Each memory access generated by the instruction uses the memory type and Shareability attribute associated with its own address.
- The instruction generates an Alignment fault caused by the memory type.
For the EL1&0 translation regime, when EL2 is enabled in the current Security state:
— If the stage 1 translation generated the mismatch, the resulting exception is taken to EL1.
— If the stage 2 translation generated the mismatch, the resulting exception is taken to EL2.
— If both stages of translation generate the mismatch, the exception can be taken to either EL1 or EL2.
• The instruction executes as a NOP.

K1.2.14 Crossing a peripheral boundary with a Device access

Performing memory accesses from one load or store instruction to Device memory that crosses a boundary corresponding to the smallest translation granule size of the implementation causes CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation performs one of the following behaviors:
• All memory accesses generated by the instruction are performed as if the boundary has no effect on the memory accesses.
• All memory accesses generated by the instruction are performed as if the boundary has no effect on the memory accesses except that there is no guarantee of ordering between memory accesses.
• The instruction generates an alignment fault caused by the memory type.

For the EL1&0 translation regime, when EL2 is enabled in the current Security state:
— If the stage 1 translation causes the boundary to be crossed, the resulting exception is taken to EL1.
— If the stage 2 translation causes the boundary to be crossed, the resulting exception is taken to EL2.
— If both stages of translation cause the boundary to be crossed, the resulting exception can be taken to either EL1 or EL2.
• The instruction executes as a NOP.

Note
The boundary referred to is between two Device memory regions that are both:
• Of the size of the smallest implemented translation granule.
• Aligned to the size of the smallest implemented translation granule.

K1.2.15 CONSTRAINED UNPREDICTABLE behaviors with SVE memory accesses

The CONSTRAINED UNPREDICTABLE behaviors that are associated with memory accesses due to loads and stores also apply to SVE vector load and store instructions. The CONSTRAINED UNPREDICTABLE behaviors are described in Crossing a page boundary with different memory types or Shareability attributes on page K1-11589 and Crossing a peripheral boundary with a Device access.

The CONSTRAINED UNPREDICTABLE behaviors apply to the following SVE memory accesses:
• When an SVE unpredicated contiguous load or store instruction accesses an address range that crosses a boundary between memory types, Shareability attributes, or peripherals.
• When an SVE predicated contiguous load or store instruction performs memory accesses that are associated with Active elements on both sides of a boundary between different memory types, Shareability attributes, or peripherals.
• When an SVE predicated non-contiguous load or store instruction performs a memory access that is associated with an Active element that crosses a boundary between memory types, Shareability attributes, or peripherals.

Memory addresses that are associated with Inactive elements cannot trigger CONSTRAINED UNPREDICTABLE behaviors.

If SVE vector loads and stores trigger a CONSTRAINED UNPREDICTABLE behavior that then generates an alignment fault, the fault is handled in the same as any other synchronous memory fault caused by an SVE load or store instruction.
K1.2.16 CONSTRAINED UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs

Load-Exclusive and Store-Exclusive instruction usage restrictions on page B2-218 defines a Load-Exclusive/Store-Exclusive pair, and identifies various CONSTRAINED UNPREDICTABLE behaviors associated with using Load-Exclusive/Store-Exclusive pairs. In summary, these cases are:

- The target virtual address of a StoreExcl instruction is different from the virtual address of the preceding LoadExcl instruction in the same thread of execution.
- The transaction size of a StoreExcl instruction is different from the transaction size of the preceding LoadExcl instruction in the same thread of execution.
- The StoreExcl instruction accesses a different number of registers than the preceding LoadExcl instruction in the same thread of execution.
- The memory attributes for a StoreExcl instruction are different from the memory attributes for the preceding LoadExcl instruction in the same thread of execution, either:
  - Because the translation of the accessed address changes between the LoadExcl instruction and the StoreExcl instruction.
  - Because the LoadExcl instruction and the StoreExcl instruction use different virtual addresses, with different attributes, that point to the same physical address.

In addition, the effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE.

See the descriptions in Load-Exclusive and Store-Exclusive instruction usage restrictions on page B2-218 for the permitted behavior in each of these cases, and any constraints that might apply to whether the case is CONSTRAINED UNPREDICTABLE.

K1.2.17 CONSTRAINED UNPREDICTABLE behavior for A64 instructions

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different A64 instructions listed in Chapter C6 A64 Base Instruction Descriptions and Chapter C7 A64 Advanced SIMD and Floating-point Instruction Descriptions.

LDAXP

For a description of this instruction and the encoding, see LDAXP on page C6-1542.

CONSTRARED UNPREDICTABLE behavior

If \( t = t_2 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value \( 0x0 \). When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

LDNP

For a description of this instruction and the encoding, see LDNP on page C6-1568.

CONSTRARED UNPREDICTABLE behavior

If \( t = t_2 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
• For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

**LDNP (SIMD&FP)**

For a description of this instruction and the encoding, see *LDNP (SIMD&FP)* on page C7-2480.

**CONSTRAINED UNPREDICTABLE behavior**

If \( t = t_2 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

**LDP**

For a description of this instruction and the encoding, see *LDP* on page C6-1570.

**CONSTRAINED UNPREDICTABLE behavior**

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and \( (t = n || t_2 = n) \) \&\& \( n \neq 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If \( t = t_2 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs all of the loads using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

**LDP (SIMD&FP)**

For a description of this instruction and the encoding, see *LDP (SIMD&FP)* on page C7-2482.

**CONSTRAINED UNPREDICTABLE behavior**

If \( t = t_2 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
• The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.

• For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

LDPSW

For a description of this instruction and the encoding, see LDPSW on page C6-1574.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and (t == n || t2 == n) && n != 31, then one of the following behaviors must occur:

• The instruction is UNDEFINED.

• The instruction executes as a NOP.

• The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

• For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If t == t2, then one of the following behaviors must occur:

• The instruction is UNDEFINED.

• The instruction executes as a NOP.

• The instruction performs all of the loads using the specified addressing mode, and the register loaded is set to an UNKNOWN value.

• For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

LDR (immediate)

For a description of this instruction and the encoding, see LDR (immediate) on page C6-1577.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and n == t && n != 31, then one of the following behaviors must occur:

• The instruction is UNDEFINED.

• The instruction executes as a NOP.

• The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

• For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.
LDRB (immediate)

For a description of this instruction and the encoding, see LDRB (immediate) on page C6-1586.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and \( n = t \land n \neq 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value \( 0x0 \). When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

LDRH (immediate)

For a description of this instruction and the encoding, see LDRH (immediate) on page C6-1591.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and \( n = t \land n \neq 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value \( 0x0 \). When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

LDRSB (immediate)

For a description of this instruction and the encoding, see LDRSB (immediate) on page C6-1596.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and \( n = t \land n \neq 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value \( 0x0 \). When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.
Note

Pre-indexed addressing and post-indexed addressing imply writeback.

LDRSH (immediate)

For a description of this instruction and the encoding, see LDRSH (immediate) on page C6-1602.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and \( n = t \) \&\& \( n \neq 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

LDRSW (immediate)

For a description of this instruction and the encoding, see LDRSW (immediate) on page C6-1608.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and \( n = t \) \&\& \( n \neq 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

LDXP

For a description of this instruction and the encoding, see LDXP on page C6-1670.

CONSTRAINED UNPREDICTABLE behavior

If \( t = t_2 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

STP

For a description of this instruction and the encoding, see STP on page C6-1892.

CONSTRANGED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and \((t == n || t2 == n) \&\& n != 31\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

________ Note __________

Pre-indexed addressing and post-indexed addressing imply writeback.

STLXP

For a description of this instruction and the encoding, see STLXP on page C6-1880.

CONSTRANGED UNPREDICTABLE behavior

If \(s == t || (s == t2)\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If \(s == n \&\& n != 31\) then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

STLXR

For a description of this instruction and the encoding, see STLXR on page C6-1883.

CONSTRANGED UNPREDICTABLE behavior

If \(s == t\), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
• For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If \( s = n \& \& n \neq 31 \) then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as a NOP.
• The instruction performs the store to an UNKNOWN address.
• For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

STLXRB

For a description of this instruction and the encoding, see "STLXRB on page C6-1886."

CONSTRANDED UNPREDICTABLE behavior

If \( s = t \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as a NOP.
• The instruction performs the store to the specified address, but the value stored is UNKNOWN.
• For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If \( s = n \& \& n \neq 31 \) then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as a NOP.
• The instruction performs the store to an UNKNOWN address.
• For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

STLXRH

For a description of this instruction and the encoding, see "STLXRH on page C6-1888."

CONSTRANDED UNPREDICTABLE behavior

If \( s = t \), then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as a NOP.
• The instruction performs the store to the specified address, but the value stored is UNKNOWN.
• For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If \( s = n \& \& n \neq 31 \) then one of the following behaviors must occur:

• The instruction is UNDEFINED.
• The instruction executes as a NOP.
• The instruction performs the store to an UNKNOWN address.
• For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.
STR (immediate)

For a description of this instruction and the encoding, see STR (immediate) on page C6-1895.

**CONSTRANDED UNPREDICTABLE behavior**

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and \( n = t \) && \( n \neq 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

--- Note ---

Pre-indexed addressing and post-indexed addressing imply writeback.

---

STRB (immediate)

For a description of this instruction and the encoding, see STRB (immediate) on page C6-1900.

**CONSTRANDED UNPREDICTABLE behavior**

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and \( n = t \) && \( n \neq 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

--- Note ---

Pre-indexed addressing and post-indexed addressing imply writeback.

---

STRH (immediate)

For a description of this instruction and the encoding, see STRH (immediate) on page C6-1905.

**CONSTRANDED UNPREDICTABLE behavior**

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and \( n = t \) && \( n \neq 31 \), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

--- Note ---

Pre-indexed addressing and post-indexed addressing imply writeback.
STXP

For a description of this instruction and the encoding, see STXP on page C6-1950.

**CONSTRAINED UNPREDICTABLE behavior**

If \( s = t \) \( \mid (s = t_2) \), then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If \( s = n \) \&\& \( n \neq 31 \) then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

STXR

For a description of this instruction and the encoding, see STXR on page C6-1953.

**CONSTRAINED UNPREDICTABLE behavior**

If \( s = t \), then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If \( s = n \) \&\& \( n \neq 31 \) then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

STXRB

For a description of this instruction and the encoding, see STXRB on page C6-1955.

**CONSTRAINED UNPREDICTABLE behavior**

If \( s = t \), then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value 0x0. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.
If \( s = n \land n \neq 31 \) then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and \text{HCR\_EL2\.TIDCP} is 1, the instruction is trapped to EL2 with EC value \text{0x0}. When \text{FEAT\_TIDCP1} is implemented, for execution at EL0, this also applies when \text{SCTLR\_EL1\.TIDCP} or \text{SCTLR\_EL2\.TIDCP} is 1.

\text{STXRH}

For a description of this instruction and the encoding, see \textit{STXRH} on page C6-1957.

\textit{CONSTRAINED UNPREDICTABLE behavior}

If \( s = t \), then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and \text{HCR\_EL2\.TIDCP} is 1, the instruction is trapped to EL2 with EC value \text{0x0}. When \text{FEAT\_TIDCP1} is implemented, for execution at EL0, this also applies when \text{SCTLR\_EL1\.TIDCP} or \text{SCTLR\_EL2\.TIDCP} is 1.

If \( s = n \land n \neq 31 \) then one of the following behaviors must occur:
- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and \text{HCR\_EL2\.TIDCP} is 1, the instruction is trapped to EL2 with EC value \text{0x0}. When \text{FEAT\_TIDCP1} is implemented, for execution at EL0, this also applies when \text{SCTLR\_EL1\.TIDCP} or \text{SCTLR\_EL2\.TIDCP} is 1.

\textbf{K1.2.18 Out of range values of the Set/Way/Index fields in cache maintenance instructions}

In the cache maintenance by set/way instructions \text{DC CISW}, \text{DC CSW}, and \text{DC ISW}, if any set/way/index argument is larger than the value supported by the implementation, then the behavior is \textit{CONSTRAINED UNPREDICTABLE} and one of the following occurs:
- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

\textit{Note}

This \textit{CONSTRAINED UNPREDICTABLE} behavior applies, also, to the AArch32 cache maintenance by set/way instructions \text{DCCISW}, \text{DCCSW}, and \text{DCISW}.

\textbf{K1.2.19 Reserved values in System and memory-mapped registers and translation table entries}

Unless otherwise stated in this manual, all unallocated or reserved values of fields with allocated values within AArch64 System registers, memory-mapped registers, and translation table entries behave in one of the following ways:
- The unallocated value maps onto any of the allocated values, but otherwise does not cause \textit{CONSTRAINED UNPREDICTABLE} behavior.
- The unallocated value causes effects that could be achieved by a combination of more than one of the allocated values.
- The unallocated value causes the field to have no functional effect.
Note

These constraints are identical to those for the equivalent AArch32 definitions, as given in Reserved values in System and memory-mapped registers and translation table entries on page K1-11583.

Unless otherwise stated, when a direct write of a System register or memory-mapped register writes an unallocated value to a field, if that value is unallocated in all contexts then a subsequent read of the field returns an UNKNOWN value.

K1.2.20 CONSTRAINED UNPREDICTABLE behavior in Debug state

Behavior in Debug state on page H2-10242 of this manual describes the CONSTRAINED UNPREDICTABLE behaviors that are specifically associated with Debug state.
Architectural Constraints on UNPREDICTABLE Behaviors
K1.2 AArch64 CONSTRAINED UNPREDICTABLE behaviors
Appendix K2
Recommendations for Reporting Memory Attributes on an Interconnect

This appendix describes the Arm recommendations for reporting the memory attributes that are assigned by the PE. It contains the following section:

• Arm recommendations for reporting memory attributes on an interconnect on page K2-11604.
K2.1  Arm recommendations for reporting memory attributes on an interconnect

The Arm architecture defines the architectural interface between software and the PE hardware. This means the mechanisms by which different memory type and Cacheability attributes are presented on an interface to an interconnect fabric such as AMBA® AXI are, strictly, outside the scope of the architecture. This appendix describes an approach for the interface between a PE implementation and an interconnect fabric that Arm strongly recommends, but these recommendations do not form part of the Arm architecture.

K2.1.1  Effect of microarchitectural choices on memory attributes

Implementations of the Arm architecture permit considerable variability in the presentation of memory attributes on the interconnect fabric, particularly in cases where the PE implementation does not provide optimized support for a memory type. For example, an implementation might treat Write-Through locations as Non-cacheable at some level of cache, because functionally this is consistent with the definition of Write-Through, but for the particular implementation the performance trade-off does not merit the hardware directly providing Write-Through capability. However, in such implementations, the assigned memory attributes are not changed by the microarchitectural choices. The microarchitecture simply implements different ways of handling some memory attributes.

Therefore, Arm strongly recommended that where any or all of the following memory attributes are presented on the interface between a PE and an interconnect fabric, the attributes that are presented are completely consistent with the attributes defined by the translation system:

- The memory type, Normal or Device.
- The Early Write Acknowledgment attribute.
- The ordering requirements.
- The Shareability.
- The Cacheability, including where practicable, the allocation hints.

Effect when memory accesses are forced to be Non-cacheable

Arm also strongly recommends that the effects of forcing accesses to Normal memory to be Non-cacheable, as described in Enabling and disabling the caching of memory accesses on page D7-5047 for AArch64 and in Enabling and disabling the caching of memory accesses in AArch32 state on page G4-9117 for AArch32, are reflected on the interconnect by the memory type and attributes used for memory transactions generated while the cache is disabled.
Appendix K3
ETE Recommended Configurations

This appendix describes the ETE features that Arm recommends are implemented. It contains the following section:
•  Configurations on page K3-11606.
K3.1 Configurations

For optional features not described here, it is IMPLEMENTATION DEFINED whether the feature is implemented. For features which have an IMPLEMENTATION DEFINED size or number, and are not described here, the size or number of that feature is IMPLEMENTATION DEFINED.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATBTRIG</td>
<td>ATB Trigger Support</td>
<td>Yes, if ATB is implemented</td>
</tr>
<tr>
<td>NUMACPAIRS</td>
<td>Address Comparator pairs</td>
<td>4</td>
</tr>
<tr>
<td>NUMCIDC</td>
<td>Context Identifier Comparators</td>
<td>&gt;= 1</td>
</tr>
<tr>
<td>NUMVMIDC</td>
<td>Virtual Context Identifier Comparators</td>
<td>&gt;= 1, if EL2 is implemented</td>
</tr>
<tr>
<td>NUMCNTR</td>
<td>Number of Counters</td>
<td>2</td>
</tr>
<tr>
<td>NUMEVENT</td>
<td>Number of ETEEvents</td>
<td>4</td>
</tr>
<tr>
<td>NUMEXTINSEL</td>
<td>Number of External Input Selectors</td>
<td>4</td>
</tr>
<tr>
<td>NUMRSPAIR</td>
<td>Number of Resource selection pairs</td>
<td>&gt;= 8</td>
</tr>
<tr>
<td>NUMSEQSTATE</td>
<td>Number of Sequencer states</td>
<td>4</td>
</tr>
<tr>
<td>NUMSSCC</td>
<td>Number of Single-shot Comparator Controls</td>
<td>&gt;= 1</td>
</tr>
<tr>
<td>RETSTACK</td>
<td>Return stack</td>
<td>Yes</td>
</tr>
<tr>
<td>STALLCTL</td>
<td>PE stalling capability</td>
<td>Yes</td>
</tr>
<tr>
<td>TRACEIDSIZE</td>
<td>Trace ID size</td>
<td>7-bits, if ATB is implemented</td>
</tr>
<tr>
<td>CCITMIN</td>
<td>Cycle count minimum threshold</td>
<td>4</td>
</tr>
<tr>
<td>CCSIZE</td>
<td>Cycle counter size</td>
<td>&gt;= 12</td>
</tr>
<tr>
<td>WFXMODE</td>
<td>WFI, WFIT, WFE, and WFET instruction classification</td>
<td>WFI, WFIT, WFE, and WFET instructions are classified as P0 instructions</td>
</tr>
</tbody>
</table>
Appendix K4
ETE and TRBE Software Usage Examples

This appendix describes a simple trace decompressor. It contains the following sections:

- Trace analyzer on page K4-11608.
- ETE programming on page K4-11654.
- Trace examples on page K4-11660.
- Differences between ETM and ETE on page K4-11671.
- Context switching on page K4-11673.
- Controlling generation of trace buffer management events on page K4-11675.
K4.1 Trace analyzer

K4.1.1 Introduction

Using Trace Info elements to start trace analysis

After the trace analyzer has located an Alignment Synchronization packet and synchronized with the trace byte stream, it must search for the following elements to begin to analyze the trace byte stream:

1. A Trace Info element.
2. A Context element and a Target Address element.

The trace unit might not generate a Context element and Target Address element immediately after it generates a Trace Info element.

If a Cancel element cancels a Trace Info element then the trace analyzer can still use the information from the discarded Trace Info element, but if the Context element and Target Address element are also discarded, then the trace analyzer must wait for the trace unit to generate a new Context element and Target Address element.

Encountering Trace Info elements after trace analysis has started

The trace unit might generate Trace Info elements periodically, as a result of trace protocol synchronization requests. This is useful if trace is stored in a circular buffer, because it provides multiple points where trace analysis can start. After a trace analyzer observes the first Trace Info element, it can ignore subsequent Trace Info elements in the same trace session because the static trace programming cannot change and the speculation depth is updated by other element types during the trace session.

Decompression information

To decompress a trace byte stream, the trace analyzer requires a number of values which differ between implementations. All the information required by the decompressor to analyze the trace byte stream is provided in the TRCIDR0 to TRCIDR13 registers.

Table K4-1 on page K4-11608 lists the static variables required by the decompressing stages and the fields that provide this information.

<table>
<thead>
<tr>
<th>Variable</th>
<th>ID Field</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commit Mode</td>
<td>TRCIDR0.COMMOPT</td>
<td>1</td>
</tr>
<tr>
<td>Virtual context identifier size</td>
<td>TRCIDR2.VMIDSIZE</td>
<td>1</td>
</tr>
<tr>
<td>Maximum speculation depth</td>
<td>TRCIDR8.MAXSPEC</td>
<td>1</td>
</tr>
<tr>
<td>Transactional Memory support</td>
<td>TRCIDR0.COMMTRANS</td>
<td>1 &amp; 2</td>
</tr>
<tr>
<td>WFx Instructions</td>
<td>TRCIDR2.WFXMODE</td>
<td>3</td>
</tr>
</tbody>
</table>

K4.1.2 ETE common pseudocode

Element ASL

Atom enumeration

// Atom
// ====
// Atom enum. Atoms are either E (taken) or N (not taken).
enumeration Atom {
    Atom_E,
    Atom_N
};

AtomElement()

// AtomElement()
// ============
// Generates an Atom element based on the given atom.
Element AtomElement(Atom t)
    Element a;
    a.kind = ELEM_ATOM;
    a.payload.atom_type = t;
    return a;

QElement()

// QElement()
// =========
// Generates a Q element based on the number of elements
// to resolve.
Element QElement(integer count)
    Element a;
    a.kind = ELEM_Q;
    a.payload.count = count;
    return a;

CancelElement()

// CancelElement()
// ===============
// Generates a Cancel element based on a given number
// of elements to cancel.
Element CancelElement(integer count)
    Element a;
    a.kind = ELEM_CANCEL;
    a.payload.count = count;
    return a;

CommitElement()

// CommitElement()
// ===============
// Generates a commit element based on the
// number of elements to commit.
Element CommitElement(integer count)
    Element a;
    a.kind = ELEM_COMMIT;
    a.payload.count = count;
    return a;

ContextElement()

// ContextElement()
// ================
// Generates a context element based on context ID, VMID,
// Exception level, Security state and AArch32/64 state.
Element ContextElement(bits(32) context_id,
bits(32) vmid,
bits(2) exception_level,
SecurityLevel secure,
boolean sixty_four_bit)

Element a;
	a.kind = ELEM_CONTEXT;
	a.payload.context_id = context_id;
	a.payload.vmid = vmid;
	a.payload.exception_level = exception_level;
	a.payload.security = secure;
	a.payload.sixty_four_bit = sixty_four_bit;

return a;

// CycleCountElement() // CycleCountElement()
// ===================
// Generates a Cycle Count element based on a number of cycles.
Element CycleCountElement(integer count)

Element a;
	a.kind = ELEM_CYCLE_COUNT;
	a.payload.count = count;

return a;

// DiscardElement() // DiscardElement()
// ================
// Generates a Discard element.
Element DiscardElement()

Element a;
	a.kind = ELEM_DISCARD;

return a;

// ExceptionElement() // ExceptionElement()
// ==================
// Generates an Exception element based on the address to branch
to and the type of exception.
Element ExceptionElement(integer exception_type, bits(64) address)

Element a;
	a.kind = ELEM_EXCEPTION;
	a.payload.exception_type = exception_type;
	a.payload.address = address;

return a;

// EventElement() // EventElement()
// ==============
// Generates an Event element based on the number of the event that fired.
Element EventElement(integer idx)

Element a;
	a.kind = ELEM_EVENT;
	a.payload.event_id = idx;

return a;
**MispredictElement()**

```c
// MispredictElement()
// ===============
// Generates a Mispredict element.
Element MispredictElement()
    Element a;
    a.kind = ELEM_MISPREDICT;
    return a;
```

**OverflowElement()**

```c
// OverflowElement()
// ===============
// Generates an Overflow element.
Element OverflowElement()
    Element a;
    a.kind = ELEM_OVERFLOW;
    return a;
```

**TimestampElement()**

```c
// TimestampElement()
// ================
// Generates a Timestamp element based on a timestamp value
// and a cycle count value.
Element TimestampElement(integer timestamp, integer cycles)
    Element a;
    a.kind = ELEM_TIMESTAMP;
    a.payload.timestamp = timestamp;
    a.payload.cycle_count = cycles;
    return a;
```

**TraceInfoElement()**

```c
// TraceInfoElement()
// =================
// Generates a Trace Info element based on cycle counting parameters,
// speculation depth, and transaction status.
Element TraceInfoElement(boolean cc_enabled,
                         integer cc_threshold,
                         integer current_spec_depth,
                         boolean in_transaction)
    Element a;
    a.kind = ELEM_TRACE_INFO;
    a.payload.cc_enabled = cc_enabled;
    a.payload.cc_threshold = cc_threshold;
    a.payload.current_spec_depth = current_spec_depth;
    a.payload.in_transaction = in_transaction;
    return a;
```

**TraceOnElement()**

```c
// TraceOnElement()
// ===============
// Generates a Trace On element.
Element TraceOnElement()
    Element a;
```
a.kind = ELEM_TRACE_ON;
return a;

**TargetAddressElement()**

// TargetAddressElement()
// ================
// Generates a Target Address element based on a given
// address and sub_isa.
Element TargetAddressElement(AddressHistoryBufferEntry reg)
    Element a;
    a.kind = ELEM_TARGET_ADDRESS;
    a.payload.address = reg.address;
    a.payload.sub_isa = reg.sub_isa;
    return a;

**SourceAddressElement()**

// SourceAddressElement()
// ================
// Generates a Source Address element based on an instruction's address
// and sub_isa.
Element SourceAddressElement(AddressHistoryBufferEntry reg)
    Element a;
    a.kind = ELEM_SOURCE_ADDRESS;
    a.payload.address = reg.address;
    a.payload.sub_isa = reg.sub_isa;
    return a;

**TransactionStartElement()**

// TransactionStartElement()
// ================
Element TransactionStartElement()
    Element a;
    a.kind = ELEM_TRANS_START;
    return a;

**TransactionCommitElement()**

// TransactionCommitElement()
// ================
Element TransactionCommitElement()
    Element a;
    a.kind = ELEM_TRANS_COMMIT;
    return a;

**TransactionFailureElement()**

// TransactionFailureElement()
// ================
Element TransactionFailureElement()
    Element a;
    a.kind = ELEM_TRANS_FAILURE;
    return a;
Decompressor enumerations

SubISA enumeration

// SubISA
// ========
// Represents sub instruction set.
// IS0 = AArch32 or AArch64, IS1 = AArch32 Thumb
enumeration SubISA {
    IS0,
    IS1
};

SynchronisationState enumeration

// States to represent synchronisation of the reconstructor, state
// transitions as follows:
// ____________
// | Init State | Input         | Final State |
// |____________|---------------|-------------|
// | NOT_SYNC   | context element| CONTEXT     |
// | NOT_SYNC   | address element | ADDRESS     |
// | ADDRESS    | context element | FULL_SYNC   |
// | ADDRESS    | overflow element| NOT_SYNC    |
// | ADDRESS    | discard element | NOT_SYNC    |
// | ADDRESS    | trace on element| NOT_SYNC    |
// | ADDRESS    | exception element| NOT_SYNC   |
// | CONTEXT    | address element | FULL_SYNC   |
// | CONTEXT    | overflow element| NOT_SYNC    |
// | CONTEXT    | discard element | NOT_SYNC    |
// | CONTEXT    | trace on element| NOT_SYNC    |
// | FULL_SYNC  | indirect branch| CONTEXT     |
// | FULL_SYNC  | discard element | NOT_SYNC    |
// | FULL_SYNC  | overflow element| NOT_SYNC    |
// | FULL_SYNC  | trace on element| NOT_SYNC    |
// ____________
enumeration SynchronisationState {
    NOT_SYNC_STATE, // Not syncing, need sync
    CONTEXT_STATE, // Have context, need address
    ADDRESS_STATE, // Have address, need context
    FULL_SYNC_STATE // Fully synced
};

InstType enumeration

// InstType
// =======
// Instruction type. Cannot use BranchType as this does not cover other P0
// non-branching instructions (WFE/WFI, ISB).
// WFX counts as 'Other' if it is not a P0 element (see TRCIDR2.WFXMODE).
enumeration InstType {
    InstType_BRANCH_DIR, // Direct branch
    InstType_BRANCH_INDIR, // Indirect branch
    InstType_WFX, // WFI/WFE instruction
    InstType_ISB, // Instruction barrier
    InstType_OTHER // Non-P0 instructions
};

Decompressor functions

EndOfStream()

// EndOfStream()
// =============
// Returns TRUE iff all the data in the stream have been consumed.
boolean EndOfStream(bits(S) stream);

ReservedEncoding()

// ReservedEncoding()
// =============
// The trace byte stream is not compliant to the protocol. The trace analyzer
// has to stop.
ReservedEncoding();

ReadAndConsume()

// ReadAndConsume()
// =============
// Reads the next N bits from the trace byte stream and returns them, also
// updating the trace byte stream pointer.
bits(N) ReadAndConsume(integer N, bits(S) stream);

LogDecompressor()

// Instrumentation functions
// ================
LogDecompressor(Decomp_Level lvl, string details);
LogElem(Decomp_Level lvl, Element e, string details);
integer GetNextDebugId();
ERROR(string msg);
LogReturnStack();
PrintElement(Element e);
string ExcepTypeToStr(integer type_val);

SBZ()

// SBZ()
// =====
// Raise an error if the field B is not zero.
SBZ(bits(N) B);

ResolutionQueue

// ResolutionQueue.Initialize()
// ==============
// If decompression starts at a Trace Info element that has a non-zero
// speculation depth, the trace analyzer must wait until the speculation
// of these unseen P0 elements has been resolved.
//
// Set the number of unseen P0 elements that are outstanding that need to be
// resolved.
ResolutionQueue.Initialize(integer i);

// ResolutionQueue.Uninitialized()
// ===============
// Returns TRUE if the resolution queue is uninitialized.
boolean ResolutionQueue.Uninitialized();

// ResolutionQueue.Aligned()
// ================
// Returns TRUE if all the unseen P0 elements have been resolved.
boolean ResolutionQueue.Aligned();

// ResolutionQueue.Align()
// ==============
// Mark the oldest oldest unseen P0 element as resolved.
ResolutionQueue.Align();
// ResolutionQueue.Length()
// ========================
// Returns the number of elements in the queue.
integer ResolutionQueue.Length();

// ResolutionQueue.PopBack()
// =========================
// Discards the element at the back (youngest) of the queue.
ResolutionQueue.PopBack();

// ResolutionQueue.Back()
// ======================
// Returns the element at the back (youngest) of the queue.
Element ResolutionQueue.Back();

// ResolutionQueue.PopFront()
// ==========================  
// Removes the element at the front (oldest) from the queue.
ResolutionQueue.PopFront();

// ResolutionQueue.Front()
// =======================  
// Returns the element at the front (oldest) of the queue.
Element ResolutionQueue.Front();

// ResolutionQueue.CommitFront()
// ==============================  
// Commits the element at the front of the queue.
ResolutionQueue.CommitFront();

// ResolutionQueue.Push()
// ======================  
// Add element e to the back of the queue.
ResolutionQueue.Push(Element e);

TransactionQueue

// TransactionQueue.Length()
// =========================
// Return the number of entries in the transaction queue.
integer TransactionQueue.Length();

// TransactionQueue.FrontPop()
// ===========================
// Remove the first entry in the transaction queue.
TransactionQueue.FrontPop();

// TransactionQueue.Front()
// ========================
// Return the element at the front of the transaction queue.
Element TransactionQueue.Front();

// TransactionQueue.Push()
// =======================
// Add an element to the back of the transaction queue.
TransactionQueue.Push(Element e);

// TransactionQueue.InTransaction()
// ================================
// Are we currently in a transaction?
boolean TransactionQueue.InTransaction();

// TransactionQueue.StartTransaction()
// ===================================
// Enter a transaction.
TransactionQueue.StartTransaction();
// TransactionQueue.EndTransaction()
// =================================
// Leave a transaction.
TransactionQueue.EndTransaction();

**ReturnStack**

// ReturnStack.Reset()
// ==================
// Resets the return stack.
ReturnStack.Reset();

// ReturnStack.Push(bits(64) addr, SubISA sub_isa)
// ===============================================
// Pushes onto the return stack.
ReturnStack.Push(bits(64) addr, SubISA sub_isa);

// ReturnStack.Pop()
// ===============
// Pops the top of the return stack.
ReturnStackEntry ReturnStack.Pop();

// ReturnStack.IsEmpty()
// =====================
// Returns TRUE iff the return stack is empty.
boolean ReturnStack.IsEmpty();

**AddressHistoryBufferEntry()**

// AddressHistoryBufferEntry
// =========================
// An entry in the address history buffer.
type AddressHistoryBufferEntry is (bits(64) address, SubISA sub_isa)

AddressHistoryBufferEntry UNKNOWN_ADDRESS;

**AddressHistoryBuffer()**

// AddressHistoryBuffer.Reset()
// ===========================
// Resets the address history buffer.
AddressHistoryBuffer.Reset()
for i = 0 to 2
   DSTATE.IA.address_history_buffer[i].address = Zeros();
   DSTATE.IA.address_history_buffer[i].sub_isa = IS0;

return;

// AddressHistoryBuffer.Add()
// ==========================
// Adds an address to the address history buffer.
AddressHistoryBuffer.Add(AddressHistoryBufferEntry entry)
   DSTATE.IA.address_history_buffer[2] = DSTATE.IA.address_history_buffer[1];
   DSTATE.IA.address_history_buffer[1] = DSTATE.IA.address_history_buffer[0];
   DSTATE.IA.address_history_buffer[0] = entry;
return;

AddressHistoryBuffer.Add(bits(64) address, SubISA sub_isa)
   AddressHistoryBufferEntry entry;
   entry.address = address;
   entry.sub_isa = sub_isa;
   AddressHistoryBuffer.Add(entry);
return;
// AddressHistoryBuffer.Get()
// =========================
// Returns the given entry from the address history buffer.
AddressHistoryBufferEntry AddressHistoryBuffer.Get(integer n)
    assert n < 3;
    return DSTATE.IA.address_history_buffer[n];

ProgramImage

// ProgramImage.DecodeNextInst()
// ==================================
// Returns the decoded next instruction in the program image.
DecodedInst ProgramImage.DecodeNextInst();

// ProgramImageDecodeAvailable()
// ==================================
// Returns TRUE iff we are currently inside the program image.
boolean ProgramImage.DecodeAvailable();

ExceptionWithUnknownAddress()

// ExceptionWithUnknownAddress()
// ==================================
// Does this exception type have an unknown
// preferred exception return address.
boolean ExceptionWithUnknownAddress(Element e)
    case e.payload.exception_type<4:0> of
    when '00000', '11001'
        return TRUE;
    when '11000'
        ERROR("Transaction Failure Element");
    otherwise
        return FALSE;

Data encodings

POD()

// POD()
// ======
// Return data from stream in Plain Old Data Little Endian format.
bits(N) POD(integer N, bits(S) stream)
    return ReadAndConsume(N, stream);

ULEB128()

// ULEB128()
// =========
// Gets N bits of continuable data from the stream.
bits(N) ULEB128(bits(S) stream)
    return BitReplacement(stream, Zeros(N));

BitReplacement()

// BitReplacement()
// ================
// Gets N bits of continuable, bit replacement data from the stream.
bits(N) BitReplacement(bits(S) stream, bits(N) original)
    R = original;
    I = 0;
    bits(8) BYTE;
    repeat
        BYTE = ReadAndConsume(8, stream);
K4.1.3 Stage 1, parsing the byte stream

The first stage of analyzing the trace is to convert from the bits of the trace byte stream to the elements that are encoded in that trace byte stream.

The ETE architecture enables a trace unit to use techniques that can reduce the trace bandwidth and trace storage requirements. Some of these techniques require the trace analyzer to retain some information between packets so that it can successfully analyze future packets.

Retained state

- The trace analyzer maintains an independent copy of the address history buffer of the last three Target Address elements.
- The address history buffer in the trace analyzer is required to reconstruct the Target Address elements from the trace byte stream.
- The trace analyzer must maintain the current speculation depth of the parsed trace byte stream.
- The trace analyzer must have the maximum speculation depth supported by the trace unit.
- The trace analyzer maintains a copy of the last Timestamp element value decompressed.
- The last Timestamp element value in the trace analyzer is required to reconstruct the full timestamp value for a Timestamp Packet.
- The trace analyzer must maintain a copy of the context:
  - Context identifier.
  - Virtual context identifier.
  - AArch64 or AArch32 state
  - Exception level
  - Security state
- The trace analyzer must maintain a copy of the cycle count threshold.

```typescript
// InstructionParserState()
// // InstructionParserState
// // ===============
// // State of the instruction parser.

type InstructionParserState is (  
bits(64) timestamp,          // The most recently broadcast timestamp value.  
    // The Address History Buffer.  
    array [0..2] of AddressHistoryBufferEntry address_history_buffer,

    // Context parameters.  
    bits(32) context_id,        // Most recently broadcast Context ID.  
    bits(32) vmid,              // Most recently broadcast VMID.  
    bits(2) exception_level,    // Most recently broadcast Exception level.  
);  
```
SecurityLevel security, // Most recently broadcast Security state. 
boolean sixty_four_bit, // Most recently broadcast AArch state
  // (32 or 64).

// Speculation
integer current_spec_depth,  // The current speculation depth.
boolean T,                   // The current transactional state.

// Trace Session static
integer cc_threshold,        // Cycle count threshold value.

// Static state
integer max_spec_depth,      // The maximum speculation depth.
boolean commit_mode,         // Commit mode.
boolean comm_trans           // How transactions traced.
)

Parsing

The first stage of the decompressor is to convert from the trace byte stream to trace element stream. The trace byte stream can start at the an Alignment Sync packet boundary.

Parse_Trace()

// Parse_Trace()
// =============
// Parses a trace bytestream generated by an ETE trace unit.
Parse_Trace(bits(S) stream)
repeat
  header = ReadAndConsume(8, stream);
case header of
  when '00000000' Parse_ExtensionPacket(header, stream);
  when '00000001' Parse_TraceInfoPacket(header, stream);
  when '0000001x' Parse_TimestampPacket(header, stream);
  when '00000100' TraceOnPacket();
  when '00000110' Parse_ExceptionPacket(header, stream);
  when '00001010' TransactionStartPacket();
  when '00001011' TransactionCommitPacket();
  when '0000110x' Parse_CycleCountPackets(header, stream);
  when '0000111x' Parse_CycleCountPackets(header, stream);
  when '0001xxxx' Parse_CycleCountPackets(header, stream);
  when '00101101' Parse_CommitPacket(header, stream);
  when '0010111x' Parse_CancelPackets(header, stream);
  when '001100xx' Parse_MispredictPacket(header, stream);
  when '001101xx' Parse_CancelPackets(header, stream);
  when '00111xxx' Parse_CancelPackets(header, stream);
  when '01100000' // Ignore packet
  when '01100xxx' Parse_EventTracingPacket(header, stream);
  when '10000000' Parse_ContextPacket(header, stream);
  when '10000001' Parse_AddressWithContextPacket(header, stream);
  when '10000101' Parse_AddressWithContextPacket(header, stream);
  when '10000110' Parse_TargetAddressPacket(header, stream);
  when '10000111' Parse_TargetAddressPacket(header, stream);
  when '10001000' Parse_QPacket(header, stream);
  when '10001001' Parse_SourceAddressPacket(header, stream);
  when '1000101x' Parse_SourceAddressPacket(header, stream);
  when '10010000' Parse_CANCEL_Packets(header, stream);
  when '10010010' Parse_TargetAddressPacket(header, stream);
  when '100101xx' Parse_TargetAddressPacket(header, stream);
  when '1001100x' Parse_TargetAddressPacket(header, stream);
  when '1001101x' Parse_TargetAddressPacket(header, stream);
  when '10011101' Parse_TargetAddressPacket(header, stream);
  when '10011110' Parse_TargetAddressPacket(header, stream);
  when '11xxxxxx' Parse_AtomPackets(header, stream);
  otherwise ReservedEncoding();
Alignment Sync packet

The Alignment Sync packet is a unique sequence of bits that identifies the boundary of another packet. The unique sequence is a header byte, 0b00000000, followed by a minimum of ten payload bytes of 0b00000000 and one final payload byte of 0b10000000.

Parse_ExtensionPacket()

// Parse_ExtensionPacket()
// =======================
// Parses Alignment Synchronization, Discard and Overflow packets.

Parse_ExtensionPacket(bits(8) header, bits(S) stream)
extension = ReadAndConsume(8, stream);
case extension of
  when '00000000' // A-sync
    while extension == '00000000' do
      extension = ReadAndConsume(8, stream);
      if extension != '10000000' then
        ReservedEncoding();
    end
  when '00000011' // Discard
    DiscardPacket();
  when '00000101' // Overflow
    OverflowPacket();
  otherwise
    ReservedEncoding();
return;

Discard

The current speculation depth must be reset to 0.

DiscardPacket()

// DiscardPacket()
// ===============
// Processes a Discard packet.

DiscardPacket()
Emit(DiscardElement());
if DSTATE.IA.T then
  Emit(TransactionFailureElement());
DSTATE.IA.current_spec_depth = 0;
DSTATE.IA.T = FALSE;
return;

Overflow

OverflowPacket()

// OverflowPacket()
// ================
// Processes an Overflow packet.

OverflowPacket()
ETE and TRBE Software Usage Examples  
K4.1 Trace analyzer

 emit(DiscardElement());
 emit(OverflowElement());

 if DSTATE.IA.T then
   emit(TransactionFailureElement());

 DSTATE.IA.T = FALSE;
 DSTATE.IA.current_spec_depth = 0;
 return;

**Trace Info**

A Trace Info packet indicates where the compression algorithms used by the trace unit have been set to a known architectural state. As the architectural state of the compression algorithms is known a trace analyzer can start decompression of the trace byte stream at this point.

If the trace unit exposes some trace speculation to the trace analyzer then the trace info packet indicates the trace speculation depth at this point in the trace element stream.

If the trace analyzer starts analysis where the trace speculation depth is nonzero then the analyzer should ignore speculation depth of Commit elements.

A Trace Info Packet sets all entries of the address history buffer to have an address of 0x0 and to sub_isa of IS0.

The current_spec_depth is set to the speculation depth indicated in the trace info element.

**Parse_TraceInfoPacket()**

```c
// Parse_TraceInfoPacket()
// =======================
// Parses a Trace Info packet.

Parse_TraceInfoPacket(bits(8) header, bits(S) stream)
  bits(8) INFO = Zeros();
  bits(96) SPEC = Zeros();
  bits(96) CYCT = Zeros();

  bits(8) PLCTL = ReadAndConsume(8, stream);
  // Extract the INFO section if present
  if PLCTL<0> == '1' then
    INFO = ReadAndConsume(8, stream);
  // Extract the SPEC section if present
  if PLCTL<2> == '1' then
    SPEC = ULEB128(stream);
  // Extract the CYCT section if present
  if PLCTL<3> == '1' then
    CYCT = ULEB128(stream);
  TraceInfoPacket(PLCTL, INFO, SPEC, CYCT);
  return;

TraceInfoPacket()
// TraceInfoPacket()
// ===============
// Processes a Trace Info packet.

TraceInfoPacket(bits(8) PLCTL,
    bits(8) INFO,
    bits(SN) SPEC,
    bits(CN) CYCT)
DSTATE.IA.timestamp = Zeros();
```
DSTATE.IA.context_id = Zeros();
DSTATE.IA.vid = Zeros();
DSTATE.IA.exception_level = EL0;
DSTATE.IA.security = SecurityLevel_SECURE;
DSTATE.IA.sixty_four_bit = FALSE;
AddressHistoryBuffer.Reset();

cc_threshold = if INFO<0> == '1' then UInt(CYCT) else 0;
DSTATE.IA.current_spec_depth = UInt(SPEC);

Emit(TraceInfoElement(INFO<0> == '1',     // cc_enabled
    cc_threshold,
    DSTATE.IA.current_spec_depth,
    INFO<6> == '1'));

return;

**Trace On**

The Trace On packet indicates that there has been a discontinuity in the instruction trace element stream. It is output whenever a gap occurs, after the gap occurs. This means that a Trace On packet is output:

- When trace generation becomes operative, after the first A-Sync and Trace Info packets but before any packet types that indicate any P0 elements.
- After a trace unit buffer overflow. Again, the Trace On packet is output after the A-Sync and Trace Info packets but before any packet types that indicate any P0 elements.
- After gaps caused by filtering. For example, if filtering is applied to the generation of the trace element stream, so that the trace unit only generates trace for a particular program code sequence, the trace unit might spend much of its time in an inactive state, only generating trace periodically. In this case, a Trace On packet is output after each discontinuity in the trace element stream. The Trace On packet must be output before any packet types that indicate any P0 elements.

**TraceOnPacket()**

```cpp
// TraceOnPacket()
// ===============
// Processes a Trace On packet.
```

TraceOnPacket()
Emit(TraceOnElement());
return;

**Speculation**

The Commit element must modify the current speculation depth.

**Parse_CommitPacket()**

```cpp
// Parse_CommitPacket()
// ===============
// Parses a Commit packet.
```

Parse_CommitPacket(bits(8) header, bits(S) stream)
    bits(32) COMMIT = ULEB128(stream);
    CommitPacket(COMMIT);
    return;

**CommitPacket()**

```cpp
// CommitPacket()
// ===============
// Processes a Commit packet.
```

CommitPacket(bits(N) COMMIT)
The **Cancel element** must modify the current speculation depth.

### Parse_CancelPackets()

// Parse_CancelFormatPackets()  
// ================  
// Parses all the various Cancel packets.

Parse_CancelPackets(bits(8) header, bits(S) stream)  
case header of  
when '0010111x'  
   // Cancel Format 1  
   M = header<0>;  
   bits(32) CANCEL = ULEB128(stream);  
   CancelFormat1Packet(M, CANCEL);  
when '001101xx'  
   // Cancel Format 2  
   A2 = header<1:0>;  
   CancelFormat2Packet(A2);  
when '00111xxx'  
   // Cancel Format 3  
   A = header<0>;  
   CC = header<2:1>;  
   CancelFormat3Packet(CC, A);  
return;

### CancelFormat1Packet()

// CancelFormat1Packet()  
// ==============  
// Processes a Cancel packet, format 1.

CancelFormat1Packet(bit M, bits(N) CANCEL)  
count = UInt(CANCEL);  
Emit(CancelElement(count));  
UpdateSpecDepth(-count);  
if M == '1' then  
   Emit(MispredictElement());  
return;

### CancelFormat2Packet()

// CancelFormat2Packet()  
// ==============  
// Processes a Cancel packet, format 2.

CancelFormat2Packet(bits(2) A)  
case A of  
when '01'  
   HandleAtom(Atom_E);  
when '10'  
   HandleAtom(Atom_E);  
   HandleAtom(Atom_E);  
when '11'  
   HandleAtom(Atom_N);  
   count = 1;  
   Emit(CancelElement(count));  
   UpdateSpecDepth(-count);  
   Emit(MispredictElement());  
return;
CancelFormat3Packet()
// CancelFormat3Packet()
// =====================
// Processes a Cancel packet, format 3.
CancelFormat3Packet(bits(2) CC, bit A)
   if A == '1' then
      HandleAtom(Atom_E);
      count = UInt(CC) + 2;
      Emit(CancelElement(count));
      UpdateSpecDepth(-count);
      Emit(MispredictElement());
      return;
Mispredict
Parse_MispredictPacket()
// Parse_MispredictPacket()
// ========================
// Parses a Mispredict packet.
Parse_MispredictPacket(bits(8) header, bits(S) stream)
   A = header<1:0>;
   MispredictPacket(A);
   return;
MispredictPacket()
// MispredictPacket()
// ==================
// Processes a Mispredict packet.
MispredictPacket(bits(2) A)
   case A of
      when '01'
         HandleAtom(Atom_E);
      when '10'
         HandleAtom(Atom_E);
         HandleAtom(Atom_E);
      when '11'
         HandleAtom(Atom_N);
      otherwise
         Emit(MispredictElement());
         return;
Atom packets
Each Atom element decoded from an Atom packet must increment the current speculation depth of this stage of the decompressor.

Parse_AtomPacket()
// Parse_AtomPacket()
// ===================
// Parses all the various Atom packets.
Parse_AtomPacket(bits(8) header, bits(S) stream)
   case header of
      when '1111011x'                 // Atom Format 1
         bit A = header<0>;
         AtomFormat1Packet(A);
when '110110xx'  // Atom Format 2
    bits(2) A = header<1:0>;
    AtomFormat2Packet(A);

when '11111xxx'  // Atom Format 3
    bits(3) A = header<2:0>;
    AtomFormat3Packet(A);

when '110111xx'  // Atom Format 4
    bits(2) A = header<1:0>;
    AtomFormat4Packet(A);

when '11110101'  // Atom Format 5.1
    AtomFormat5_1Packet();

when '11010101', '11010110', '11010111' // Atom Format 5.2
    bits(2) A = header<1:0>;
    AtomFormat5_2Packet(A);

when '11xxxxxx'  // Atom Format 6
    bit A = header<5>;
    bits(5) COUNT = header<4:0>;
    AtomFormat6Packet(A, COUNT);
    return;

AtomFormat1Packet()

// AtomFormat1Packet()
// ===============
// Processes an Atom packet, format 1.

AtomFormat1Packet(bit A)
    if A == '1' then
        HandleAtom(Atom_E);
    else
        HandleAtom(Atom_N);
    return;

AtomFormat2Packet()

// AtomFormat2Packet()
// ===============
// Processes an Atom packet, format 2.

AtomFormat2Packet(bits(2) A)
    for I = 0 to 1
        if A<I> == '1' then
            HandleAtom(Atom_E);
        else
            HandleAtom(Atom_N);
    return;

AtomFormat3Packet()

// AtomFormat3Packet()
// ===============
// Processes an Atom packet, format 3.

AtomFormat3Packet(bits(3) A)
    for I = 0 to 2
        if A<I> == '1' then
            HandleAtom(Atom_E);
        else
            HandleAtom(Atom_N);
return;

**AtomFormat4Packet()**

//AtomFormat4Packet()
//==================
//Processes an Atom packet, format 4.

AtomFormat4Packet(bits(2) A)
case A of
  when '00'
    HandleAtom(Atom_N);
    HandleAtom(Atom_E);
    HandleAtom(Atom_E);
    HandleAtom(Atom_E);
  when '01'
    HandleAtom(Atom_N);
    HandleAtom(Atom_N);
    HandleAtom(Atom_N);
    HandleAtom(Atom_N);
  when '10'
    HandleAtom(Atom_N);
    HandleAtom(Atom_E);
    HandleAtom(Atom_N);
    HandleAtom(Atom_E);
  when '11'
    HandleAtom(Atom_E);
    HandleAtom(Atom_N);
    HandleAtom(Atom_E);
    HandleAtom(Atom_N);

return;

**AtomFormat5_1Packet()**

//AtomFormat5_1Packet()
//=====================
//Processes an Atom packet, format 5.1.

AtomFormat5_1Packet()
  HandleAtom(Atom_N);
  HandleAtom(Atom_E);
  HandleAtom(Atom_E);
  HandleAtom(Atom_E);
  return;

**AtomFormat5_2Packet()**

// AtomFormat5_2Packet()
//=====================
//Processes an Atom packet, format 5.2.

AtomFormat5_2Packet(bits(2) A)
case A of
  when '01'
    HandleAtom(Atom_N);
    HandleAtom(Atom_N);
    HandleAtom(Atom_N);
    HandleAtom(Atom_N);
    HandleAtom(Atom_N);
  when '10'
    HandleAtom(Atom_N);
    HandleAtom(Atom_E);
    HandleAtom(Atom_N);
    HandleAtom(Atom_E);
    HandleAtom(Atom_N);

return;
when '11'
    HandleAtom(Atom_E);
    HandleAtom(Atom_N);
    HandleAtom(Atom_E);
    HandleAtom(Atom_N);
    HandleAtom(Atom_E);
    return;

AtomFormat6Packet()

// AtomFormat6Packet()
// ===============
// Processes an Atom packet, format 6.

AtomFormat6Packet(bit A, bits(5) COUNT)
    for I = 0 to UInt(COUNT) + 2
        HandleAtom(Atom_E);
    if A == '1' then
        HandleAtom(Atom_N);
    else
        HandleAtom(Atom_E);
    return;

Q packets

The Q element must increment the current speculation depth at this stage of the decompressor.

Parse_QPacket()

// Parse_QPacket()
// ===============
// Parses a Q packet.

Parse_QPacket(bits(8) header, bits(5) stream)
    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);
    bits(32) count;
    TYPE = header<3:0>;
    case TYPE of
        when '0000', '0001', '0010'
            ExactMatchBytes(header, stream);
            count = ULEB128(stream);
            QPacket(TYPE, entry, count);
        when '0101'
            ShortAddressBytes(IS0, stream);
            count = ULEB128(stream);
            QPacket(TYPE, entry, count);
        when '0110'
            ShortAddressBytes(IS1, stream);
            count = ULEB128(stream);
            QPacket(TYPE, entry, count);
        when '1010', '1011'
            LongAddressBytes(header, stream);
            count = ULEB128(stream);
            QPacket(TYPE, entry, count);
        when '1100'
            count = ULEB128(stream);
            QPacket(TYPE, UNKNOWN_ADDRESS, count);
        when '1111'
            QPacket(TYPE, UNKNOWN_ADDRESS, UNKNOWN_COUNT);
ETE and TRBE Software Usage Examples
K4.1 Trace analyzer

QPacket()
// QPacket()
// =========
// Processes a Q packet.
QPacket(bits(4) TYPE, AddressHistoryBufferEntry A, bits(CN) COUNT)
Emit(QElement(UInt(COUNT)));
UpdateSpecDepth(1);

// The decoding of the Address field is done by the AddressPacket function,
// but this did not Emit the address element.
if (TYPE != '11xx' && TYPE != '00xx') then
  Emit(TargetAddressElement(A));
return;

Source address packets

Parse_SourceAddressPacket()
// Parse_SourceAddressPacket()
// ===========================
// Parses a Source Address packet.
Parse_SourceAddressPacket(bits(8) header, bits(S) stream)
  AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);
  case header of
    when '10111000'
      data1 = ReadAndConsume(8, stream);
data2 = ReadAndConsume(8, stream);
data3 = ReadAndConsume(48, stream);
      entry.sub_isa = IS0;
      entry.address<63:0> = data3:datal2<6:0>:data1<6:0>:'00';
      AddressHistoryBuffer.Add(entry);
      UpdateSpecDepth(1);
      Emit(SourceAddressElement(entry));
    when '10111001'
      data11 = ReadAndConsume(8, stream);
data21 = ReadAndConsume(56, stream);
      a = entry;
      entry.sub_isa = IS1;
      entry.address<63:0> = data21:datal2<6:0>:0';
      AddressHistoryBuffer.Add(entry);
      UpdateSpecDepth(1);
      Emit(SourceAddressElement(entry));
    when '10111010'
      data2 = ReadAndConsume(8, stream);
data22 = ReadAndConsume(8, stream);
data32 = ReadAndConsume(16, stream);
      a = entry;
      entry.sub_isa = IS0;
      entry.address<31:0> = data32:datal2<6:0>:data2<6:0>:00';
      AddressHistoryBuffer.Add(entry);
      UpdateSpecDepth(1);
      Emit(SourceAddressElement(entry));
when '10110111'  
    data13 = ReadAndConsume(8, stream);  
    data23 = ReadAndConsume(24, stream);  
    a = entry;  
    entry.sub_isa = IS1;  
    entry.address<31:0> = data23:数据<6:0>:'0';  
    AddressHistoryBuffer.Add(entry);  
    UpdateSpecDepth(1);  
    Emit(SourceAddressElement(entry));

when '10110000'  
    data14 = ReadAndConsume(8, stream);  
    a = entry;  
    entry.sub_isa = IS0;  
    if data14<7> == '1' then  
      data24 = ReadAndConsume(8, stream);  
      entry.address<16:0> = data24<7:0>:data14<6:0>:'00';  
    else  
      entry.address<8:0> = data14<6:0>:'00';  
    AddressHistoryBuffer.Add(entry);  
    UpdateSpecDepth(1);  
    Emit(SourceAddressElement(entry));

when '10011101'  
    data15 = ReadAndConsume(8, stream);  
    a = entry;  
    entry.sub_isa = IS0;  
    if data15<7> == '1' then  
      data25 = ReadAndConsume(8, stream);  
      entry.address<15:0> = data25<7:0>:data15<6:0>:'0';  
    else  
      entry.address<7:0> = data15<6:0>:'0';  
    AddressHistoryBuffer.Add(entry);  
    UpdateSpecDepth(1);  
    Emit(SourceAddressElement(entry));

when '101100xx'             // Exact match  
    q = UInt(header<1:0>);  
    entry = AddressHistoryBuffer.Get(q);  
    AddressHistoryBuffer.Add(entry);  
    UpdateSpecDepth(1);  
    Emit(SourceAddressElement(entry));

return;

Exceptions

The Exception element must increment the current speculation depth at this stage of the decompressor.

Parse_ExceptionPacket()

// Parse_ExceptionPacket()  
// ===============  
// Parses an exception packet.

Parse_ExceptionPacket(bits(8) header, bits(S) stream)  
    payload = ReadAndConsume(8, stream);  
    bits(2) E;  
    E<<0> = payload<0>;  
    E<<1> = payload<6>;  
    bits(S) TYPE = payload<5:1>;  
    bits(8) AH;  
    if E == '01' || E == '10' then  
      // Treat the ADDRESS bytes as an Address packet, including
// updating the address registers.
AH = ReadAndConsume(8, stream);

case AH of
  when '1001000x', '10010010' // Exact Match.
    ExactMatchBytes(AH, stream);
  when '10010101' // Short Address IS0.
    ShortAddressBytes(IS0, stream);
  when '10010110' // Short Address IS1.
    ShortAddressBytes(IS1, stream);
  when '1001101x', '10011101', '10011110' // Long Address.
    LongAddressBytes(AH, stream);
  when '1000001x', '10000101', '10000110' // Long Address with Context.
    LongAddressBytes(AH, stream);
  when '1000001x', '10000101', '10000110' // Unknown address
    UnknownAddressHistoryBuffer();
  when '01110000' // Unknown address
    UnknownAddressHistoryBuffer();
ExceptionPacket(E, TYPE, AH);
else
  ReservedEncoding();
return;

ExceptionPacket()
// ExceptionPacket()
// ================
// Processes an Exception packet.

ExceptionPacket(bits(2) E, bits(5) TYPE, bits(8) AH)
AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);
case E of
  when '01'
    if TYPE == '11000' then
      Emit(TransactionFailureElement());
    else
      Emit(ExceptionElement(UInt(TYPE), entry.address));
  when '10'
    // The new context and address must now be Emitted.
    if AH<7:4> == '1000' then // Long Address with Context
      Emit(ContextElement(DSTATE.IA.context_id,
                         DSTATE.IA.vmid,
                         DSTATE.IA.exception_level,
                         DSTATE.IA.security,
                         DSTATE.IA.sixty_four_bit));
      Emit(TargetAddressElement(entry));
    else
      if TYPE == '00000' && DSTATE.IA.T then
        Emit(TransactionFailureElement());
      else
        Emit(ExceptionElement(UInt(TYPE), entry.address));
    endif
  otherwise
    ReservedEncoding();
  endif
UpdateSpecDepth(1);
return;
Address and context

**Parse_TargetAddressPacket()**

// Parse_TargetAddressPacket()
// ================
// Parses a Target Address packet.

```c
Parse_TargetAddressPacket(bits(8) header, bits(S) stream)
```

```c
case header of
  when '1001101x', '10011101', '10011110' // Long Address
    LongAddressBytes(header, stream);
  when '10010101' // Short Address IS0
    ShortAddressBytes(IS0, stream);
  when '10010110' // Short Address IS1
    ShortAddressBytes(IS1, stream);
  when '1001000x', '10010010' // Exact match
    ExactMatchBytes(header, stream);

AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);
Emit(TargetAddressElement(entry));
return;
```

**Parse_LongAddressBytes()**

// LongAddressBytes()
// ================
// Reads and parses the long address form used in some packets.

```c
LongAddressBytes(bits(8) header, bits(S) stream)
```

```c
case header<2:0> of
  when '010' // 32-bit IS0
    data32_is0 = ReadAndConsume(16, stream);
    A8_2 = data32_is0<6:0>; SBZ(data32_is0<7>);
    A15_9 = data32_is0<14:8>; SBZ(data32_is0<15>);
    bits(16) A31_16;
    A31_16 = POD(16, stream);
    LongAddressPacket(header, A31_16:A15_9:A8_2);
  when '011' // 32-bit IS1
    data32_is1 = ReadAndConsume(8, stream);
    A7_1 = data32_is1<6:0>; SBZ(data32_is1<7>);
    A31_8 = POD(24, stream);
    LongAddressPacket(header, A31_8:A7_1);
  when '101' // 64-bit IS0
    data64_is0 = ReadAndConsume(16, stream);
    A8_2 = data64_is0<6:0>;
    SBZ(data64_is0<7>);
    A15_9 = data64_is0<14:8>;
    SBZ(data64_is0<15>);
    A63_16 = POD(48, stream);
    LongAddressPacket(header, A63_16:A15_9:A8_2);
  when '110' // 64-bit IS1
    data64_is1 = ReadAndConsume(8, stream);
    A7_1 = data64_is1<6:0>; SBZ(data64_is1<7>);
    A63_8 = POD(56, stream);
    LongAddressPacket(header, A63_8:A7_1);

return;
```
LongAddressPacket()

// LongAddressPacket()
// ===================
// Parses the long form address used in some packets.

LongAddressPacket(bits(8) header, bits(AN) A)

    a = AddressHistoryBuffer.Get(0);

    // Called from a variety of packet types, so only look at bits <2:0> of // the header.
    case header<2:0> of
        when '010'              // 32-bit address, IS0
            assert(AN == 30);
            a.sub_isa = IS0;
            // address<63:32> unchanged
            a.address<31:2> = A<29:0>;
            a.address<1:0> = '00';
        when '011'              // 32-bit address, IS1
            assert(AN == 31);
            a.sub_isa = IS1;
            // address<63:32> unchanged
            a.address<31:1> = A<30:0>;
            a.address<0> = '0';
        when '101'              // 64-bit address, IS0
            assert(AN == 62);
            a.sub_isa = IS0;
            a.address<63:2> = A<61:0>;
            a.address<1:0> = '00';
        when '110'              // 64-bit address, IS1
            assert(AN == 63);
            a.sub_isa = IS1;
            a.address<63:1> = A<62:0>;
            a.address<0> = '0';
    UpdateAddressHistoryBuffer(a.address, a.sub_isa);
    return;

Parse_ShortAddressBytes()

// ShortAddressBytes()
// ===================
// Reads and parses the short form address used in some packets.

ShortAddressBytes(SubISA sub_isa, bits(S) stream)
    bits(15) data;

    if sub_isa == IS0 then
        data = BitReplacement(stream,
                                AddressHistoryBuffer.Get(0).address<16:2>);
    else
        data = BitReplacement(stream,
                                AddressHistoryBuffer.Get(0).address<15:1>);

    ShortAddressPacket(sub_isa, data);
    return;

ShortAddressPacket()

// ShortAddressPacket()
// ====================
// Parses the short form address used in some packets.

ShortAddressPacket(SubISA sub_isa, bits(AN) A)
a = AddressHistoryBuffer.Get(0);
assert (AN == 7 || AN == 15);

case sub_isa of
  when IS0                                // IS0
    a.sub_isa = IS0;
    // address<63:AN+2> unchanged
    a.address<AN+1:0> = A:'00';
  when IS1                                // IS1
    a.sub_isa = IS1;
    // address<63:AN+1> unchanged
    a.address<AN:0> = A:'0';

  UpdateAddressHistoryBuffer(a.address, a.sub_isa);
  return;

Parse_ExactMatchBytes()

// ExactMatchBytes()
// ================
// Reads and parses an exact address match used in some packets.

ExactMatchBytes(bits(8) header, bits(S) stream)
  QE = header<1:0>;
  ExactMatchPacket(QE);
  return;

ExactMatchPacket()

// ExactMatchPacket()
// ================
// Parses an exact address match.

ExactMatchPacket(bits(2) QE)
  q = UInt(QE);
  AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(q);
  AddressHistoryBuffer.Add(entry);
  return;

R_{TKBWR}

The Context element value is created by combining the value encoded in a Context Packet and the last Context element value.

Parse_AddressWithContextPacket()

// Parse_AddressWithContextPacket()
// ===============================
// Parses a Target Address with Context packet.

Parse_AddressWithContextPacket(bits(8) header, bits(S) stream)
  LongAddressBytes(header, stream);
  ContextBytes(stream);
  AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);
  Emit(ContextElement(DSTATE.IA.context_id, DSTATE.IA.vmid, DSTATE.IA.exception_level, DSTATE.IA.security, DSTATE.IA.sixty_four_bit));
  Emit(TargetAddressElement(entry));
  return;

Parse_ContextBytes()

// ContextBytes()
// =============
/ Generates a Context packet from the stream.

ContextBytes(bits(S) stream)
    payload = ReadAndConsume(8, stream);
    EL = payload<1:0>;
    SBZ(payload<3:2>);
    SF = payload<4>;
    NS = payload<5>;
    V = payload<6>;
    C = payload<7>;
    case C:V of
    when '00'
        ContextPacket('1', C, V, NS, SF, EL,
                    DSTATE.IA.vmid,
                    DSTATE.IA.context_id);
    when '01'
        bits(32) VMID;
        VMID = POD(32, stream);
        ContextPacket('1', C, V, NS, SF, EL,
                    VMID,
                    DSTATE.IA.context_id);
    when '10'
        context_id = POD(32, stream);
        ContextPacket('1', C, V, NS, SF, EL,
                    DSTATE.IA.vmid,
                    context_id);
    when '11'
        bits(32) VMID;
        VMID = POD(32, stream);
        context_id = POD(32, stream);
        ContextPacket('1', C, V, NS, SF, EL,
                    VMID,
                    context_id);
    return;

// Parse_ContextPacket()
// =====================
// Parses a Context packet.

Parse_ContextPacket(bits(8) header, bits(S) stream)
    P = header<0>;
    if P == '0' then
        ContextPacket(P, '0', '0', '0', '0', '00',
                    DSTATE.IA.vmid,
                    DSTATE.IA.context_id);
    else
        ContextBytes(stream);
        Emit(ContextElement(DSTATE.IA.context_id,
                    DSTATE.IA.vmid,
                    DSTATE.IA.exception_level,
                    DSTATE.IA.security,
                    DSTATE.IA.sixty_four_bit));
    return;

ContextPacket()
// ContextPacket()
// ==============
// Processes a Context packet.

ContextPacket(bit P,
              bit C,
              bit V,
              bit NS,
              bit SF,
              bits(2) EL,
bits(32) VMID,
bits(32) context_id)
if P == '1' then
  if C == '1' then
    DSTATE.IA.context_id = context_id;
  if V == '1' then
    DSTATE.IA.vmid = VMID;
    DSTATE.IA.exception_level<1:0> = EL<1:0>;
    if NS == '1' then
      DSTATE.IA.security = SecurityLevel_NONSECURE;
    else
      DSTATE.IA.security = SecurityLevel_SECURE;
    DSTATE.IA.sixty_four_bit = (SF == '1');
    return;

Transactions

TransactionStartPacket()

// Transaction Start Packet()
// ============================
// Processes a Transaction Start packet.
TransactionStartPacket()
  Emit(TransactionStartElement());
  DSTATE.IA.T = TRUE;
  if DSTATE.IA.comm_trans then
    UpdateSpecDepth(1);
  return;

TransactionCommitPacket()

// TransactionCommitPacket()
// =========================
// Processes a Transaction Commit packet.
TransactionCommitPacket()
  Emit(TransactionCommitElement());
  DSTATE.IA.T = FALSE;
  return;

Timestamps

The Timestamp element value is created by combining the value encoded in a Timestamp Packet and the last Timestamp element value.

Parse_TimestampPacket()

// Parse_TimestampPacket()
// =======================
// Parses a Timestamp packet.
Parse_TimestampPacket(bits(8) header, bits(S) stream)
  bit N;
  bits(64) TS = BitReplacement(stream, DSTATE.IA.timestamp);
  bits(20) COUNT = Zeros();
  N = header<0>;
  if N == '1' then
    COUNT = ULEB128(stream);
    TimestampPacket(N, TS, COUNT);
  else
    TimestampPacket(N, TS, COUNT);
  return;
**Parse_CycleCountPackets()**

// Parse_CycleCountPackets()
// ========================
// Parses all the various Cycle Count packets.

Parse_CycleCountPackets(bits(8) header, bits(S) stream)

  case header of
    when '0000111x'                 // Cycle Count format 1
      bit U = header<0>;
      bits(32) COMMIT = Zeros();
      bits(20) COUNT = Zeros();
      if TRCIDR0.COMMOPT == '0' then
        COMMIT = ULEB128(stream);
      if U == '0' then
        COUNT = ULEB128(stream);
      CycleCountFormat1Packet(U, COMMIT, COUNT);
    when '0000110x'                 // Cycle Count format 2
      bits(8) payload = POD(8, stream);
      bits(4) BBBB = payload<3:0>;
      if TRCIDR0.COMMOPT == '0' then
        bit F = header<0>;
        bits(4) AAAA = payload<7:4>;
      CycleCountFormat2Packet(F, AAAA, BBBB);
      else
        CycleCountFormat2Packet('1', '1111', BBBB);
    when '0001xxxx'                 // Cycle Count format 3
      bits(2) BB = header<1:0>;
      if TRCIDR0.COMMOPT == '0' then
        bits(2) AA = header<3:2>;
      CycleCountFormat3Packet(AA, BB);
      else
        CycleCountFormat3Packet('00', BB);
  return;

**TimestampPacket()**

// TimestampPacket()
// ================
// Processes a Timestamp packet.

TimestampPacket(bit N, bits(64) TS, bits(CN) COUNT)

  DSTATE.IA.timestamp = TS;
  if N == '1' then
    Emit(TimestampElement(UInt(DSTATE.IA.timestamp), UInt(COUNT)));
  else
    Emit(TimestampElement(UInt(DSTATE.IA.timestamp), integer UNKNOWN));
  return;

**CycleCountFormat1Packet()**

// CycleCountFormat1Packet()
// =========================
// Processes a Cycle Count packet, format 1.

CycleCountFormat1Packet(bit U, bits(N) COMMIT, bits(20) COUNT)

  if UInt(COMMIT) > 0 then
    Emit(CommitElement(UInt(COMMIT)));
    UpdateSpecDepth(-UInt(COMMIT));
  if U == '1' then
    Emit(CycleCountElement(integer UNKNOWN));
  else
    Emit(CycleCountElement(UInt(COUNT) + DSTATE.IA.cc_threshold));

return;

**CycleCountFormat2Packet()**

// CycleCountFormat2Packet()
// =========================
// Processes a Cycle Count packet, format 2.

CycleCountFormat2Packet(bit F, bits(4) AAAA, bits(4) BBBB)
if F == '1' then
    commit_count = DSTATE.IA.max_spec_depth + UInt(AAAA) - 15;
else
    commit_count = UInt(AAAA) + 1;
if commit_count > 0 then
    Emit(CommitElement(commit_count));
    UpdateSpecDepth(-commit_count);
    Emit(CycleCountElement(DSTATE.IA.cc_threshold + UInt(BBBB)));
return;

**CycleCountFormat3Packet()**

// CycleCountFormat3Packet()
// =========================
// Processes a Cycle Count packet, format 3.

CycleCountFormat3Packet(bits(2) AA, bits(2) BB)
if !DSTATE.IA.commit_mode then
    Emit(CommitElement(UInt(AA) + 1));
    UpdateSpecDepth(-(UInt(AA) + 1));
    Emit(CycleCountElement(DSTATE.IA.cc_threshold + UInt(BB)));
return;

**Event tracing**

**Parse_EventTracingPacket()**

// Parse_EventTracingPacket()
// ==========================
// Parses an Event packet.

Parse_EventTracingPacket(bits(8) header, bits(S) stream)
    EventTracingPacket(header<3:0>);
return;

**EventTracingPacket()**

// EventTracingPacket()
// ====================
// Processes an Event packet.

EventTracingPacket(bits(4) EVENT)
    for I = 0 to 3
        if EVENT<I> == '1' then
            Emit(EventElement(I));
    return;

**Functions**

**ReadAndConsume()**

// ReadAndConsume()
// ================
// Reads the next N bits from the trace byte stream and returns them, also
// updating the trace byte stream pointer.

bits(N) ReadAndConsume(integer N, bits(S) stream)
if DSTATE.stream_ptr == S then
    print ("Reached end of trace\n");
    assert(FALSE);

    bits(N) data = stream[DSTATE.stream_ptr + N - 1: DSTATE.stream_ptr];
    DSTATE.stream_ptr = DSTATE.stream_ptr + N;
    return data;

HandleAtom()

// HandleAtom()
// =============
// Emits an atom, and updates the speculation depth.

HandleAtom(Atom t)
    Emit(AtomElement(t));
    UpdateSpecDepth(1);
    return;

UpdateSpecDepth()

// UpdateSpecDepth()
// ===============
// Update the speculation depth by a number of elements.

UpdateSpecDepth(integer count)
    DSTATE.IA.current_spec_depth = DSTATE.IA.current_spec_depth + count;
    if DSTATE.IA.current_spec_depth > DSTATE.IA.max_spec_depth then
        commit_number = DSTATE.IA.current_spec_depth - DSTATE.IA.max_spec_depth;
        Emit(CommitElement(commit_number));
        DSTATE.IA.current_spec_depth = DSTATE.IA.max_spec_depth;
    
    return;

UpdateAddressHistoryBuffer()

// UpdateAddressHistoryBuffer()
// ============================
// Adds the given address and sub_isa to the AHB.

UpdateAddressHistoryBuffer(bits(64) address, SubISA sub_isa)
    AddressHistoryBuffer.Add(address, sub_isa);
    return;

UnknownAddressHistoryBuffer()

// UnknownAddressHistoryBuffer()
// =============================
// Adds an unknown address and sub_isa to the AHB.

UnknownAddressHistoryBuffer()
    AddressHistoryBuffer.Add(Zeros(), IS0);
    return;

K4.1.4 Stage 2, speculation resolution

IKRXLC The resolution stage operates on the Elements in turn. The Elements are buffered until their resolution is determined.
**ETE and TRBE Software Usage Examples**

**K4.1 Trace analyzer**

---

## Emit

```c
// Emit()
// ======
Emit(Element e)
  e.committed = FALSE;
  case e.kind of
    when ELEM_TRACE_INFO   ProcessTraceInfo(e);
      // Speculation Support
      when ELEMCANCEL
        ProcessCancel(e);
      when ELEM_COMMIT
        ProcessCommit(e);
      when ELEM_DISCARD
        ProcessDiscard(e);
      // Transactional Support
      when ELEM_TRANS_START
        ProcessTransactionStart(e);
      when ELEM_TRANS_COMMIT
        ProcessTransactionCommit(e);
      when ELEM_TRANS_FAILURE
        ProcessTransactionFailure(e);
    when ELEM_TRANS_FAILURE
      otherwise Stack(e);
```

---

## Trace Info element

The *Trace Info element* can be used as a point to start decompression of the trace element stream. When the *Trace Info element* is generated there might still be some speculative *P0 elements*. The number of speculative *P0 elements* is indicated by the current speculation depth member of the *Trace Info element*.

If the analysis of the trace starts with a *Trace Info element* with a non-zero current speculation depth the decompressor must ignore the *Commit element* or *Cancel elements* for these *P0 elements* as they will not have been observed by the decompressor.

```c
ProcessTraceInfo()
// ProcessTraceInfo()
// ===============
// Processes a Trace Info element, resetting the analyzer to a known state.
ProcessTraceInfo(Element e)
  if ResolutionQueue.Uninitialized() then
    ResolutionQueue.Initialize(e.payload.current_spec_depth);
  if e.payload.in_transaction then
    TransactionQueue.StartTransaction();
    Stack(e);
  return;
```

## Commit element

The *Commit element* marks a number of *P0 elements* as resolved, and if at the head of the queue pass these elements onto the next stage of the decompressor.

```c
ProcessCommit()
// ProcessCommit()
// ===============
// Processes a Commit element, committing the given number of speculative
// elements.
ProcessCommit(Element e)
  integer I = 0;
```
repeat
      if !ResolutionQueue.Aligned() then
        I = I + 1;
        ResolutionQueue.Align();
      else
        case ResolutionQueue.Front().kind of
          when ELEM_EXCEPTION, ELEM_ATOM, ELEM_Q, ELEM_SOURCE_ADDRESS
            if !ResolutionQueue.Front().committed then
              I = I + 1;
              ProcessTransaction(ResolutionQueue.Front());
              AnalyzeElement(ResolutionQueue.Front());
              ResolutionQueue.PopFront();
          when ELEM_TRANS_START
            if (TRCIDR0.COMMTRANS == '0' &&
                !ResolutionQueue.Front().committed) then
              I = I + 1;
              ProcessTransaction(ResolutionQueue.Front());
              ResolutionQueue.PopFront();
            otherwise
              ProcessTransaction(ResolutionQueue.Front());
              ResolutionQueue.PopFront();
      until I == e.payload.count;

      return;

Cancel element

For example, if a Cancel element indicates that the three most recent P0 elements are canceled, then the trace analyzer must discard:

- The Cancel element.
- All elements back to, and including, the third most recent P0 element.
- Any Trace On elements encountered in that section of the element stream.

When discarding P0 elements that have been canceled, a trace analyzer must also discard many other element types that occur in the element stream between the Cancel element and the oldest P0 element that the Cancel element cancels. Table K4-2 on page K4-11640 shows which elements must be discarded.

<table>
<thead>
<tr>
<th>Element</th>
<th>Behavior on cancelation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom</td>
<td>Remove</td>
</tr>
<tr>
<td>Commit</td>
<td>Illegal</td>
</tr>
<tr>
<td>Context</td>
<td>Remove</td>
</tr>
<tr>
<td>Cycle Count</td>
<td>Process</td>
</tr>
<tr>
<td>Discard</td>
<td>Illegal</td>
</tr>
<tr>
<td>Exception</td>
<td>Remove</td>
</tr>
<tr>
<td>Event</td>
<td>Keep</td>
</tr>
<tr>
<td>Mispredict</td>
<td>Remove</td>
</tr>
<tr>
<td>Overflow</td>
<td>Illegal</td>
</tr>
</tbody>
</table>
When a Cancel element occurs, a trace analyzer must not discard Cycle Count elements.

When a Cancel element occurs, a trace analyzer must not discard ETEEvents.

When a Cancel element occurs, a trace analyzer must discard Mispredict elements.

**ProcessCancel()**

```c
// ProcessCancel()
// ===============
// Processes a Cancel element, canceling the given number of speculative elements.
ProcessCancel(Element e)
  integer I = 0;
  repeat
    if !ResolutionQueue.Aligned() then
      I = I + 1;
      ResolutionQueue.Align();
    else
      case ResolutionQueue.Back().kind of
        when ELEM_ATOM, ELEM_EXCEPTION, ELEM_Q, ELEM_SOURCE_ADDRESS
          if !ResolutionQueue.Back().committed then
            I = I + 1;
        when ELEM_TRANS_START
          if (TRCIDR0.COMMTRANS == '0' &&
              !ResolutionQueue.Back().committed) then
            I = I + 1;
            TransactionQueue.EndTransaction();
        when ELEM_CYCLE_COUNT, ELEM_EVENT, ELEM_TRACE_INFO
          AnalyzeElement(ResolutionQueue.Back());
        when ELEM_TIMESTAMP
          AnalyzeElement(ResolutionQueue.Back());
        end case;
      end if
    end if
  until I == e.payload.count;
  return;
```

**Discard element**

A Discard element indicates that tracing has become inactive while uncommitted P0 elements remain.

The trace analyzer must cancel all speculative P0 elements.
ProcessDiscard()
// ProcessDiscard()
//================
// Processes a Discard element, discarding all speculative elements.
ProcessDiscard(Element e)

while !ResolutionQueue.Aligned() do
    ResolutionQueue.Align();

while ResolutionQueue.Length() > 0 do
    case ResolutionQueue.Back().kind of
        when ELEM_EVENT, ELEM_TRACE_INFO, ELEM_TIMESTAMP
            AnalyzeElement(ResolutionQueue.Back());
            ResolutionQueue.PopBack();
        otherwise
            ResolutionQueue.PopBack();

    TransactionQueue.EndTransaction();
    return;

Stack

The Elements processed by this stage of the decompressor must be stored temporarily until the speculation has been resolved.

Stack()
// Stack()
//=======
// Pushes an element onto the resolution queue.
// TODO: Move to ResolutionQueue.asl and rename to Enqueue
Stack(Element e)
    ResolutionQueue.Push(e);
    return;

K4.1.5 Stage 2, transaction resolution

Transaction

ProcessTransaction()
// ProcessTransaction()
//====================
// Push the element into the transaction queue if we are in a transaction,
// else analyze it immediately.
ProcessTransaction(Element e)
    if TransactionQueue.InTransaction() then
        TransactionQueue.Push(e);
    else
        AnalyzeElement(e);

Transaction Start element

ProcessTransactionStart()
// ProcessTransactionStart()
//====================
// Processes a Transaction Start element, marking we are now in a transaction.
ProcessTransactionStart(Element e)
  TransactionQueue.StartTransaction();
  Stack(e);
  return;

Transaction Commit element

ProcessTransactionCommit()

// ProcessTransactionCommit()
// =========================

// Processes a Transaction Commit element, committing all elements in the
// transaction queue and ending the current transaction.

ProcessTransactionCommit(Element e)
  while TransactionQueue.Length() > 0 do
    AnalyzeElement(TransactionQueue.Front());
    TransactionQueue.FrontPop();
    TransactionQueue.EndTransaction();

Transaction Failure element

When a Transaction Failure element occurs, the trace analyzer must process the Cycle Count elements if it is
maintaining a cumulative cycle count. Otherwise it must discard the Cycle Count elements that are associated with
P0 elements within the transaction.

When a Transaction Failure element occurs, a trace analyzer must not discard the ETEEEvents.

ProcessTransactionFailure()

// ProcessTransactionFailure()
// ===========================

// Processes a Transaction Failure element, discarding all elements in
// the transaction queue and ending the transaction.

ProcessTransactionFailure(Element e)
  TransactionQueue.EndTransaction();
  return;

K4.1.6 Stage 3, analysis

Analyze element

AnalyzeElement()

// AnalyzeElement()
// ================

// Analyzes any element.

AnalyzeElement(Element e)
  case e.kind of
    when ELEM_TARGET_ADDRESS AnalyzeTargetAddress(e);
    when ELEM_CONTEXT AnalyzeContext(e);
    when ELEM_MISPREDICT AnalyzeMispredict(e);
    when ELEM_TRACE_ON AnalyzeTraceOn(e);
    when ELEM_ATOM AnalyzeAtom(e);
    when ELEM_EXCEPTION AnalyzeException(e);
    when ELEM_Q AnalyzeQ(e);
    when ELEM_CANCEL ERROR("cancel element reached analysis stage");
    when ELEM_COMMIT ERROR("commit element reached analysis stage");
    when ELEM_DISCARD AnalyzeDiscard(e);
    when ELEM_OVERFLOW AnalyzeOverflow(e);
    when ELEM_EVENT AnalyzeEvent(e);
when ELEM_TRACE_INFO     AnalyzeTraceInfo(e);
when ELEM_TIMESTAMP      AnalyzeTimestamp(e);
when ELEM_CYCLE_COUNT    AnalyzeCycleCount(e);
when ELEM_SOURCE_ADDRESS AnalyzeSourceAddress(e);
otherwise
    ERROR("Unrecognised element kind in analysis stage");
return;

**Retained state**

The trace analyzer must maintain a copy of the context:

- Context identifier.
- Virtual context identifier.
- AArch64 or AArch32 state
- Exception level
- Security state

**ReconstructState()**

```c
// ReconstructState
// ================
// Temporary storage of reconstructor state, can change after resolution.

type ReconstructState is (  
    bits(64) address,       // Current address
    bits(32) context_id,    // Current context ID
    bits(32) vmid,          // Current VMID
    bits(2) exception_level,// Current Exception level
    SecurityLevel security, // Current Security state
    boolean sixty_four_bit, // Are we in AArch64?
    SubISA sub_isa          // Current sub_isa
)
```

**Operation of the return stack**

The trace analyzer maintains an independent copy of the return stack which is used to determine when Target Address elements have been removed and then infer the target of indirect P0 instructions.

The trace analyzer return stack only operates after a certain point in the tracing flow, that is after the trace analyzer has decoded the trace packets and after all the elements that indicate speculative execution, except for Mispredict elements, have been removed from the trace element stream.

Whenever a Branch with Link instruction is initially traced with an E Atom element, the link return address and sub_isa are pushed onto the trace analyzer return stack. This means that the trace unit return stack grows with each new entry, until its maximum depth is reached and the oldest entries start being discarded.

Whenever an indirect P0 instruction is traced with a final E Atom element, and no Target Address element is traced before the next P0 element, the top entry of the trace analyzer return stack is removed and the value of that entry is the target of the indirect P0 instruction.

A trace analyzer is not required to be aware of the depth of the trace unit return stack, and implements a return stack depth of 15 entries.

A trace analyzer return stack push always occurs whenever a Branch with Link instruction is traced with an E Atom element, even if the status of the E Atom element later changes to be an N Atom element as a result of a subsequent Mispredict element.

For example, the following sequence might occur:

1. The PE speculatively executes a Branch with Link instruction that the trace unit traces with an E Atom element. The trace unit pushes the target address of the Branch with Link instruction onto the trace unit return stack.
The trace analyzer receives the E Atom element and pushes the target address of the Branch with Link instruction onto the trace analyzer return stack.

3. The PE then cancels the speculative execution. The trace unit generates a Mispredict element.

4. The trace analyzer receives the Mispredict element and changes the status of the E Atom element so that it becomes an N Atom element. The trace analyzer then knows which direction the program flow has taken, and also knows that the target address stored at the top of the trace analyzer return stack is mispredicted.

Note: Whenever the trace unit generates a Mispredict element to correct a Branch with Link instruction to an N Atom element, the mispredicted address remains in the return stack because there is no reason to remove it. There are no adverse consequences of leaving such a mispredicted address in the stack.

If more than one Mispredict element is output corresponding to a particular Atom element, the status of the Atom element alternates between E and N until it settles in its final E or N state. If the final state of the Atom element is E, then when the PE executes an indirect P0 instruction and the trace unit compares the target address with the top entry in its return stack, an address match might occur. An address match can only occur if the final status of the Atom element is E.

The trace analyzer never needs to discard the entries in its copy of the return stack. If the trace unit discards the entries in its return stack then the entries in the trace analyzer return stack remain. As more entries are pushed on to the return stack, the old entries are discarded when they are pushed off the end of the stack.

The trace analyzer does not need to prevent the return stack from being modified while in a branch broadcasting region. The fact that the trace unit discards the entries in its return stack when entering the branch broadcasting region ensures that the return stack in the trace unit and the return stack in the trace analyzer remain synchronized.

UpdateReturnStack()

// UpdateReturnStack()
// ===================
// Push the given instruction to the return stack if necessary.

UpdateReturnStack(DecodedInst inst)
if inst.is_link then
    nxt_state = DSTATE.current_analyzer_state;
    nxt_state.address = nxt_state.address + inst.size;
    if !nxt_state.sixty_four_bit then
        nxt_state.address<63:32> = Zeros();
    ReturnStack.Push(nxt_state.address, nxt_state.sub_isa);
return;

Atom element

An Atom element implies that one or more instructions have been traced, up to and including the next P0 instruction.

A trace analyzer must analyze each instruction in the program image from the current address until it observes a P0 instruction. This indicates that the PE has executed each instruction between the current address and the P0 instruction.

AnalyzeAtom()

// AnalyzeAtom()
// =============
// Analyzes an atom element.

AnalyzeAtom(Element e)
if e.payload.atom_type == Atom_E then
    DSTATE.most_recent_branch_was_taken = TRUE;
else
    DSTATE.most_recent_branch_was_taken = FALSE;
CheckForReturnStackMatch();
if DSTATE.sync_state == ADDRESS_STATE then
    DSTATE.sync_state = NOT_SYNC_STATE;
if DSTATE.sync_state != FULL_SYNC_STATE then
    // If we are unsure of context or address then we cannot meaningfully
    // analyze the atom.
    return;

boolean cur_inst_is_branch = FALSE;
// Continue logging instructions until we hit a P0 instruction.
while !cur_inst_is_branch do
    if !ProgramImage.DecodeAvailable() then
        DSTATE.sync_state = CONTEXT_STATE;
        if DSTATE.return_stack_clear_pending then
            ReturnStack.Reset();
            return;
        decoded_inst = ProgramImage.DecodeNextInst();

        case decoded_inst.branchtype of
            when InstType_BRANCH_DIR, InstType_BRANCH_INDIR
                ProcessBranchInstruction(decoded_inst,
                                   DSTATE.most_recent_branch_was_taken);
            cur_inst_is_branch = TRUE;
            UpdateReturnStack(decoded_inst);
            when InstType_WFX, InstType_ISB
                ProcessBranchInstruction(decoded_inst,
                                   DSTATE.most_recent_branch_was_taken);
            cur_inst_is_branch = TRUE;
            when InstType_OTHER
                ReconstructState nxt_state = DSTATE.current_analyzer_state;
                nxt_state.address = nxt_state.address + decoded_inst.size;
                if !nxt_state.sixty_four_bit then
                    // mask off the left-most bits
                    nxt_state.address<63:32> = Replicate('0', 32);
                DSTATE.next_analyzer_state = nxt_state;
        OutputInstruction(decoded_inst);
        DSTATE.current_analyzer_state = DSTATE.next_analyzer_state;
        if DSTATE.return_stack_clear_pending then
            ReturnStack.Reset();
            return;

Context element

AnalyzeContext()

// AnalyzeContext()
// ================
// Analyzes a context element.

AnalyzeContext(Element e)
    DSTATE.current_analyzer_state.context_id = e.payload.context_id;
    DSTATE.current_analyzer_state.vmid = e.payload.vmid;
    DSTATE.current_analyzer_state.security = e.payload.security;
    DSTATE.current_analyzer_state.exception_level = e.payload.exception_level;
    DSTATE.current_analyzer_state.sixty_four_bit = e.payload.sixty_four_bit;

    case DSTATE.sync_state of
        when NOT_SYNC_STATE
            DSTATE.sync_state = CONTEXT_STATE;
        when ADDRESS_STATE
            DSTATE.sync_state = FULL_SYNC_STATE;
        otherwise
return;

Exception element

For an Exception element, a trace analyzer must analyze each instruction from the current address, up to but not including the exception return address that the element provides. The PE has executed each instruction in that address range. The number of instructions that are executed can be zero.

Note: Trace analysis tools must be aware, that if PE execution is at the top of memory space, the address that the Exception element provides might be lower than the target address of the most recent P0 element.

AnalyzeException()

// AnalyzeException()
// ===============
// Analyzes an exception element.

AnalyzeException(Element e)
    CheckForReturnStackMatch();
    if DSTATE.sync_state == CONTEXT_STATE then
        DSTATE.sync_state = NOT_SYNC_STATE;
    if DSTATE.sync_state != FULL_SYNC_STATE then
        return;

    integer PER = UInt(e.payload.address);
    if (ExceptionWithUnknownAddress(e)) then
        continue_forward = FALSE;
    elsif UInt(DSTATE.current_analyzer_state.address) < PER then
        continue_forward = TRUE;
    else
        continue_forward = FALSE;

    // Continue logging instructions until we reach the specified address.
    while continue_forward do
        if !ProgramImage.DecodeAvailable() then
            DSTATE.sync_state = CONTEXT_STATE;
            if DSTATE.return_stack_clear_pending then
                ReturnStack.Reset();
            return;

        decoded_inst = ProgramImage.DecodeNextInst();
        if decoded_inst.branchtype == InstType_OTHER then
            ReconstructState nxt_state = DSTATE.current_analyzer_state;
            nxt_state.address = nxt_state.address + decoded_inst.size;
            DSTATE.next_analyzer_state = nxt_state;
            if !DSTATE.next_analyzer_state.sixty_four_bit then
                // mask off the left-most bits
                DSTATE.next_analyzer_state.address<63:32> = Replicate('0', 32);
            else
                ProcessBranchInstruction(decoded_inst, FALSE);
                UpdateReturnStack(decoded_inst);
                OutputInstruction(decoded_inst);

        bits(64) next_addr = DSTATE.current_analyzer_state.address + decoded_inst.size;
        DSTATE.current_analyzer_state.address = next_addr;
        if !DSTATE.current_analyzer_state.sixty_four_bit then
            // mask off the left-most bits
            DSTATE.current_analyzer_state.address<63:32> = Replicate('0', 32);
        if UInt(DSTATE.current_analyzer_state.address) >= PER then
            continue_forward = FALSE;

        if DSTATE.return_stack_clear_pending then
            ReturnStack.Reset();

    return;
return;

Source address element

A Source Address element indicates that one or more instructions have been traced, up to and including the instruction at the address associated with the element.

A Source Address element indicates that the instruction at the address associated with the element was taken.

A trace analyzer must analyze each instruction in the program image from the current address until it analyzes the instruction at the address associated with the Source Address element. This indicates that the PE has executed each instruction between the current address and that instruction, and each P0 instruction except the final instruction was not taken.

AnalyzeSourceAddress()

// AnalyzeSourceAddress()
// ======================
// Analyzes a source address element.
AnalyzeSourceAddress(Element e)
CheckForReturnStackMatch();

if DSTATE.sync_state == ADDRESS_STATE then
DSTATE.sync_state = NOT_SYNC_STATE;
if DSTATE.sync_state != FULL_SYNC_STATE then
// If we are unsure of context or address then we cannot meaningfully
// analyze the source address.
return;

DSTATE.most_recent_branch_was_taken = FALSE;
integer address = UInt(e.payload.address);

// Continue logging instructions until we hit the specified address.
while (UInt(DSTATE.current_analyzer_state.address) <= address) do
if !ProgramImage.DecodeAvailable() then
DSTATE.sync_state = CONTEXT_STATE;
if DSTATE.return_stack_clear_pending then
ReturnStack.Reset();
return;

decoded_inst = ProgramImage.DecodeNextInst();
if decoded_inst.branchtype == InstType_OTHER then
ReconstructState nxt_state = DSTATE.current_analyzer_state;
nxt_state.address = nxt_state.address + decoded_inst.size;
DSTATE.next_analyzer_state = nxt_state;
if DSTATE.next_analyzer_state.sixty_four_bit then
DSTATE.next_analyzer_state.address<63:32> = Replicate('0', 32);
else
if DSTATE.current_analyzer_state.address == e.payload.address then
DSTATE.most_recent_branch_was_taken = TRUE;
ProcessBranchInstruction(decoded_inst, DSTATE.most_recent_branch_was_taken);
cur_inst_is_branch = TRUE;
UpdateReturnStack(decoded_inst);
OutputInstruction(decoded_inst);
DSTATE.current_analyzer_state = DSTATE.next_analyzer_state;
if DSTATE.return_stack_clear_pending then
ReturnStack.Reset();
return;
Target Address element

AnalyzeTargetAddress()

// AnalyzeTargetAddress()
// ================
// Analyzes a target address element.

AnalyzeTargetAddress(Element e)
  DSTATE.current_analyzer_state.address = e.payload.address;
  DSTATE.current_analyzer_state.sub_isa = e.payload.sub_isa;
  case DSTATE.sync_state of
    when NOT_SYNC_STATE
      DSTATE.sync_state = ADDRESS_STATE;
    when CONTEXT_STATE
      DSTATE.sync_state = FULL_SYNC_STATE;
    otherwise
      return;

Trace Info element

AnalyzeTraceInfo()

// AnalyzeTraceInfo()
// ================
// Analyzes a trace info element.

AnalyzeTraceInfo(Element e)
  CheckForReturnStackMatch();
  return_stack_clear_pending = TRUE;
  return;

Trace On element

AnalyzeTraceOn()

// AnalyzeTraceOn()
// ================
// Analyzes a trace on element.

AnalyzeTraceOn(Element e)
  return_stack_clear_pending = TRUE;
  DSTATE.sync_state = NOT_SYNC_STATE;
  return;

Mispredict element

When a Mispredict element corresponds to an Atom element for a direct P0 instruction, before the trace analyzer can calculate the target of the direct P0 instruction, it must apply any applicable Mispredict elements so that it can determine whether it is an E Atom element or an N Atom element.

AnalyzeMispredict()

// AnalyzeMispredict()
// ================
// Analyzes a mispredict element.

AnalyzeMispredict(Element e)
  DSTATE.most_recent_branch_was_taken = !DSTATE.most_recent_branch_was_taken;
  ReconstructState nxt_state;
  nxt_state = UpdateBranchState(DSTATE.most_recent_branch_decoded_inst,
DSTATE.most_recent_branch_state,
DSTATE.most_recent_branch_was_taken);

DSTATE.current_analyzer_state = nxt_state;
return;

ETEvent element

AnalyzeEvent()
// AnalyzeEvent()
// ==============
// Analyzes an event element.
AnalyzeEvent(Element e);

Discard element

When a trace analyzer encounters a Discard element it must be aware that if the last committed P0 element is a conditional P0 instruction, the E or N status of that Atom element might not be correct. This is because the trace unit might be unable to generate any Mispredict elements that the conditional P0 instruction might require.

If the last P0 instruction is an indirect P0 instruction then the target address indicated in the trace stream might be incorrect. This is because the trace unit might be unable to generate any Target Address elements that the indirect P0 instruction might require.

AnalyzeDiscard()
// AnalyzeDiscard()
// ================
// Analyzes a discard element.
AnalyzeDiscard(Element e)

Overflow element

AnalyzeOverflow()
// AnalyzeOverflow()
// =================
// Analyzes an overflow element.
AnalyzeOverflow(Element e)

Q element

When a trace analyzer encounters a Q element which has a count of M executed instructions, it must proceed through the program image, analyzing each instruction until it has analyzed M instructions. If it encounters a conditional P0 instruction, the status of the condition code check for that instruction is UNKNOWN. The status of these P0 instructions is not explicitly given in the trace element stream but it might be possible to infer the status of a given P0 instruction that is based on other trace that is generated. After the trace analyzer has analyzed M instructions, the following Target Address element indicates where PE execution continues.

AnalyzeQ()
// AnalyzeQ()
// =========
// Analyzes a Q element.
AnalyzeQ(Element e)
    CheckForReturnStackMatch();

    q_with_count = e.payload.count > 0;
    if q_with_count then
        further_analysis_possible = TRUE;
    else
        further_analysis_possible = FALSE;
        DSTATE.sync_state = CONTEXT_STATE;
        // If we have no count then just wait to resync, it is not safe to guess
    
    i = 0;
    while further_analysis_possible do
        if ProgramImage.DecodeAvailable() then
            decoded_inst = ProgramImage.DecodeNextInst();
            addr = DSTATE.current_analyzer_state.address + decoded_inst.size;
            DSTATE.current_analyzer_state.address = addr;
        else
            DSTATE.sync_state = CONTEXT_STATE;
            return;
        
        i = i + 1;
        further_analysis_possible = (i < e.payload.count);
        OutputInstruction(decoded_inst);
        return;

Timestamp element

AnalyzeTimestamp()

    // AnalyzeTimestamp()
    // ================
    // Analyzes a timestamp element.
    
    AnalyzeTimestamp(Element e);

Cycle Count element

To produce a total cycle count, a trace analyzer can cumulatively add the values from all Cycle Count elements.

A trace analyzer must not use the cycle count values in Timestamp elements to produce a total cycle count. Such cycle count values are not a Cycle Count element.

AnalyzeCycleCount()

    // AnalyzeCycleCount()
    // ===============
    // Analyzes a cycle count element.
    
    AnalyzeCycleCount(Element e);

Functions

OutputInstruction()

    // OutputInstruction()
    // ================
    // Log that an instruction has been executed.
    
    OutputInstruction(DecodedInst inst)
        DSTATE.inst_out_count = DSTATE.inst_out_count + 1;
        return;
CheckReturnStackMatch()
// CheckReturnStackMatch()
// =======================
// Check if there is a return stack match, and log the result.
CheckReturnStackMatch()
if DSTATE.sync_state == CONTEXT_STATE then
  if !ReturnStack.IsEmpty() then
    ReturnStackEntry top = ReturnStack.Pop();
    DSTATE.current_analyzer_state.address = top.address;
    DSTATE.current_analyzer_state.sub_isa = top.sub_isa;
    DSTATE.sync_state = FULL_SYNC_STATE;
  return;

UpdateBranchState()
// UpdateBranchState()
// =======================
// Returns an updated state based on what was executed.
UpdateBranchState(DecodedInst inst, boolean branch_was_taken)
out_state = DSTATE.current_analyzer_state;
out_state.address = in_state.address;
out_state.sixty_four_bit = in_state.sixty_four_bit;
out_state.sub_isa = in_state.sub_isa;
if branch_was_taken then
  if inst.branchtype == InstType_BRANCH_INDIR then
    DSTATE.sync_state = CONTEXT_STATE;
  else
    if inst.branchtype == InstType_BRANCH_DIR then
      out_state.address = out_state.address + inst.addressoffset;
    else
      out_state.address = out_state.address + inst.size;
      if !in_state.sixty_four_bit then
        out_state.address<63:32> = Zeros();
      out_state.sub_isa = inst.next_sub_isa;
    else
      out_state.address = out_state.address + inst.size;
      if !out_state.sixty_four_bit then
        out_state.address<63:32> = Zeros();
  return out_state;

ProcessBranchInstruction()
// ProcessBranchInstruction()
// ===========================
// Store current state before a branch instruction, as it could change if there
// is a misprediction.
ProcessBranchInstruction(DecodedInst inst, boolean branch_was_taken)
DSTATE.most_recent_branch_state = DSTATE.current_analyzer_state;
DSTATE.most_recent_branch_decoded_inst = inst;
DSTATE.most_recent_branch_was_taken = branch_was_taken;
DSTATE.next_analyzer_state = UpdateBranchState(inst,
  DSTATE.current_analyzer_state, branch_was_taken);
return;
DecodedInst()

// DecodedInst
// ===========
// Data extracted from an instruction.

type DecodedInst is (  
    bits(32) instruction, // The instruction itself  
    InstType branchtype, // Type of P0 instruction  
    boolean is_link, // Is it a linking branch?  
    integer size, // Size (32 or 16)  
    SubISA next_sub_isa, // sub_isa of the following instruction to be  
        // executed  
    bits(64) addressoffset
)

**K4.2 ETE programming**

### K4.2.1 Example code sequences

**SPWXHJ** The enabling sequence should be from the trace sink, such as the trace buffer, to the trace unit. This is to ensure the trace sink is ready to capture trace before the trace unit generates any trace.

**SWPVBX** The disabling sequence should be from the trace unit to the trace sink. This is to ensure that any buffered trace reaches the trace sink while the trace sink is still enabled.

**Enabling the trace unit**

**SXCRYM** The following example describes the code sequence to enable the trace unit.

```
;; Program the trace unit registers, except TRCPRGCTRL
ISB              ;; Synchronize the System Register updates.
MOV x0, #0x1
MSR TRCPRGCTRL, x0 ;; Enable the ETE.
;; Wait for TRCSTATR.IDLE==0
poll_idle
ISB              ;; Synchronize the write to TRCPRGCTRL
MRS x1, TRCSTATR
TBNZ x1, #1, poll_idle
```

**Disabling the trace unit**

**SVRJNB** The following example describes the code sequence to disable the trace unit.

```
STP x0, x1, [sp, #-16]!
MRS x0, TRFCR_EL1 ;; Save the current programming of TRFCR_EL1.
MOV x1, #0x3
BIC x1, x0, x1
MSR TRFCR_EL1, x1 ;; Clear the values of TRFCR_EL1.ExTRE.
;; to put the PE in to a Trace Prohibited region
ISB                ;; Synchronize the entry to the Trace Prohibited region
TSB CSYNC          ;; Ensure that all trace has reached the trace buffer and address translations have taken place.
MDV x1, #0x0
MSR TRCPRGCTRL, x1 ;; Disable the trace unit
;; Wait for TRCSTATR.IDLE==1 and TRCSTATR.PMSTABLE==1
poll_idle
ISB
MRS x1, TRCSTATR
AND x1, x1, #3
CMP x1, #3
B.NE poll_idle
MSR TRFCR_EL1, x0 ;; Restore the programming of TRFCR_EL1.
LDP x0, x1, [sp], #16
```

**Example save restore routine**

**SXJRPO** The following example describes the code sequence for saving the trace unit state over a power down.

```
STP x0, x1, [sp, #-16]!
;; Enter a Trace Prohibited region
MRS x0, TRFCR_EL1 ;; Save the current programming of TRFCR_EL1.
MOV x1, #0x3
BIC x1, x0, x1
MSR TRFCR_EL1, x1 ;; Clear the values of TRFCR_EL1.ExTRE.
ISB                ;; Synchronizes the entry to the Trace Prohibited region
TSB CSYNC          ;; Ensure the trace unit is synchronized
```
MOV x1, #1
MSR OSLAR_EL1, x1    ;; Lock the OS lock
;; Wait for TRCSTATR.PMSTABLE==1
_poll_pmstable
    ISB
MRS x1, TRCSTATR
TST x1, #2
B.EQ poll_pmstable

MSR TRFCR_EL1, x0    ;; Restore the programming of TRFCR_EL1.

<save the trace unit registers, including TRCPRGCTL>

;; Wait for TRCSTATR.IDLE==1
_poll_idle
    ISB
MRS x1, TRCSTATR
TST x1, #1
B.EQ poll_idle
LDP x0, x1, [sp], #16

SDNGPX The following example describes the code sequence for restoring the trace unit state when power is restored.

<restore the trace unit registers, including TRCPRGCTL>

    STP x0, x1, [sp, #-16]!
MOV x0, #0
MSR OSLAR_EL1, x0    ;; Clear the OS lock
LDP x0, x1, [sp], #16
ISB

IDHQGM When programming the trace unit, it is important to be aware that the loops that poll TRCSTATR in Figure D4-2 on page D4-4769 might never complete. Arm recommends that such scenarios are avoided except in rare conditions. However, some system conditions might prevent a trace unit from either leaving the idle state or becoming idle. In particular, a trace unit might never become idle if the trace unit is unable to output all trace data to a system condition.

STTPKR If multiple reads of TRCSTATR are required, a Context synchronization event is required between each read of TRCSTATR to ensure any change to the trace unit state is observed.

K4.2.2 Minimal programming

Table K4-3 on page K4-11655 gives the values for programming the trace unit to enable trace for tracing of a single process at EL0.

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| TRCCONFIGR        | 0x000018C1 | Enable:
|                   |       | • The return stack.                                                         |
|                   |       | • Global timestamping.                                                      |
|                   |       | • Context identifier tracing.                                               |
|                   |       | • Virtual context identifier tracing.                                      |
| TRCEVENTCTLR0     | 0x00000000 | Disable all event tracing                                                   |
| TRCEVENTCTLR1     | 0x00000000 |                                                               |
| TRCSTALLCTLR      | 0x00000000 | Disable stalling, if implemented                                            |
| TRCSYNCPR         | 0x0000000C | Enable trace protocol synchronization every 4096 bytes of trace             |
| TRCTRACEIDR       | Nonzero | Set a value for the trace ID                                                |
SHBZVM Disabling tracing of Secure state might not be strictly necessary as secure tracing might be disabled by MDCR_EL3.STE, but Arm recommends not enabling trace for un-required Exception levels, to limit the amount of trace.

SQCWTJ Disabling tracing of EL1 and EL2 of Non-secure state might not be strictly necessary as non-secure tracing might be disabled by TRFCR_EL2.E2TRE and TRFCR_EL1.E1TRE, but Arm recommends not enabling trace for un-required Exception levels to limit the amount of trace.

K4.2.3 Filtering models

IVDBGM Different trace applications require different usage models of a trace unit. For example, one trace application might only require basic program flow trace, whereas another might require tracing of a specific program function.

The ETE architecture provides for each of these usage models. A trace unit can be implemented with a particular set of implementation options, so that a trade-off between functionality and cost can be achieved in meeting the requirements of a trace application. Discovery of the particular set of implementation is achieved by reading TRCIDR0 to TRCIDR13.

In a trace unit that includes all implementation options, the simplest way to use the trace unit is to turn on tracing of all aspects of PE operation and let the trace analyzer pick out the required information. However, full trace comes at a high cost in terms of port bandwidth and trace storage. These costs have an impact on the design of a system, so that a higher pin count and larger buffers might be required.

A trace unit provides on-chip filtering, that facilitates a reduction of the trace bandwidth and therefore provides for a lower system cost. By suspending and enabling trace during a trace that is run to suit the particular requirements of the trace run, the best use of both port bandwidth and trace storage can be made.

The ETE architecture provides the following basic filtering models:

Continuous tracing

This is where no filtering is applied. The following modes can be used:

- Continuous instruction tracing only, where only the instruction trace stream is output.

Instruction-based filtering

This is where instruction tracing, and data tracing if it is implemented and enabled, is active only for certain code sequences, such as for a particular process or function.

For all the possible filtering modes, the trace unit can be programmed before a trace run to enable various options, including:

- Context identifier tracing, if implemented, to indicate to a trace analyzer the Context identifier value.
- Virtual context identifier tracing, if implemented, to distinguish between different virtual machines.
- Cycle counting, to enable a trace analyzer to analyze program performance.

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRCTSCTRL</td>
<td>0x00000000</td>
<td>Disable the timestamp event</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The trace unit still generates timestamps due to other reasons such as trace protocol synchronization.</td>
</tr>
<tr>
<td>TRCVICTLR</td>
<td>0x006F0201</td>
<td>Enable ViewInst to trace everything, with the start/stop logic started</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disable:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- EL1 in Non-secure state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- EL2 in Non-secure state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- EL3-EL0 in Secure state.</td>
</tr>
<tr>
<td>TRCVIECTL</td>
<td>0x00000000</td>
<td>No address range filtering for logic started</td>
</tr>
<tr>
<td>TRCVISSCTRL</td>
<td>0x00000000</td>
<td>No start or stop points for ViewInst</td>
</tr>
</tbody>
</table>

Table K4-3 Minimal programming values (continued)
• Global timestamping, if implemented, to enable correlation of the two trace streams with other trace sources in the system.
• Branch broadcasting, if implemented, to force all taken *P0 instruction* targets to be traced with an explicit target address.

A trace unit is programmed for continuous instruction tracing when no filtering is applied to the instruction trace stream.

When a trace unit is programmed for continuous instruction tracing, ViewInst is always active during a trace run. See *Filtering trace generation* on page D4-4814.

### K4.2.4 Filtering used the exclude function

The following table describes the programming for excluding a single address range:

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRCCONFIGR</td>
<td>0x00018C1</td>
<td>Enable:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• the return stack,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• global timestamping,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Context identifier,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Virtual context identifier tracing.</td>
</tr>
<tr>
<td>TRCEVENTCTRL0R</td>
<td>0x00000000</td>
<td>Disable all event tracing.</td>
</tr>
<tr>
<td>TRCEVENTCTRL1R</td>
<td>0x00000000</td>
<td>Disable stalling, if implemented.</td>
</tr>
<tr>
<td>TRCSYNCPR</td>
<td>0x0000000C</td>
<td>Enable trace protocol synchronization every 4096 bytes of trace.</td>
</tr>
<tr>
<td>TRCTSTALLCTR</td>
<td>0x00000000</td>
<td>Disable stalling, if implemented.</td>
</tr>
<tr>
<td>TRCTRACEIDR</td>
<td>Nonzero</td>
<td>Set a value for the trace ID.</td>
</tr>
<tr>
<td>TRCTSLCTLR</td>
<td>0x00000000</td>
<td>Disable the timestamp event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The trace unit still generates timestamps due to other reasons such as trace protocol synchronization.</td>
</tr>
<tr>
<td>TRCVICTLR</td>
<td>0x006F2001</td>
<td>Enable ViewInst to trace everything, with the start/stop logic started.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disable:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• EL1 in Non-secure state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• EL2 in Non-secure state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• EL3-EL0 in Secure state.</td>
</tr>
<tr>
<td>TRCIIETL</td>
<td>0x00010000</td>
<td>Use ARC0 for the exclude logic.</td>
</tr>
<tr>
<td>TRCVISSLCTR</td>
<td>0x00000000</td>
<td>No start or stop points for ViewInst.</td>
</tr>
<tr>
<td>TRCACATR0</td>
<td>0x00000000</td>
<td>The comparator status to match on all instructions at this Virtual address</td>
</tr>
<tr>
<td>TRCACVR0</td>
<td><em>Start Address</em></td>
<td></td>
</tr>
<tr>
<td>TRCACATR1</td>
<td>0x00000000</td>
<td>The comparator status to match on all instructions at this Virtual address</td>
</tr>
<tr>
<td>TRCACVR1</td>
<td><em>End Address</em></td>
<td></td>
</tr>
</tbody>
</table>
K4.2.5 Filtering used the include function

The following table describes the programming for including a single address range:

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRCCONFIGR</td>
<td>0x00018C1</td>
<td>Enable:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• the return stack,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• global timestamping,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Context identifier,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Virtual context identifier tracing.</td>
</tr>
<tr>
<td>TRCEVENTCTL0R</td>
<td>0x00000000</td>
<td>Disable all event tracing.</td>
</tr>
<tr>
<td>TRCEVENTCTL1R</td>
<td>0x00000000</td>
<td>Disable event tracing.</td>
</tr>
<tr>
<td>TRCSTALLCTLR</td>
<td>0x00000000</td>
<td>Disable stalling, if implemented.</td>
</tr>
<tr>
<td>TRCSYNCPR</td>
<td>0x000000C</td>
<td>Enable trace protocol synchronization every 4096 bytes of trace.</td>
</tr>
<tr>
<td>TRCTRACEIDR</td>
<td>Nonzero</td>
<td>Set a value for the trace ID.</td>
</tr>
<tr>
<td>TRCTSCTLR</td>
<td>0x00000000</td>
<td>Disable the timestamp event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The trace unit still generates timestamps due to other reasons such as trace protocol synchronization.</td>
</tr>
<tr>
<td>TRCVICTLR</td>
<td>0x006F0201</td>
<td>Enable ViewInst to trace everything, with the start/stop logic started.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disable:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• EL1 in Non-secure state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• EL2 in Non-secure state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• EL3-EL0 in Secure state tracing.</td>
</tr>
<tr>
<td>TRCVIIECTLR</td>
<td>0x00000001</td>
<td>Use ARC0 for the include logic.</td>
</tr>
<tr>
<td>TRCVISSCTLR</td>
<td>0x00000000</td>
<td>No start or stop points for ViewInst.</td>
</tr>
<tr>
<td>TRCACATR0</td>
<td>0x00000000</td>
<td>The comparator status to match on all instructions at this Virtual address</td>
</tr>
<tr>
<td>TRCACVR0</td>
<td>Start Address</td>
<td></td>
</tr>
<tr>
<td>TRCACATR1</td>
<td>0x00000000</td>
<td>The comparator status to match on all instructions at this Virtual address</td>
</tr>
<tr>
<td>TRCACVR1</td>
<td>End Address</td>
<td></td>
</tr>
</tbody>
</table>

K4.2.6 OS save and restore routines

When the PE is context switching of trace unit the following registers need to save and restored. Not all these registers are necessarily implemented for all implementations. Please refer to the register description page for information on if the register is implemented.

- TRCPRGCTRLR
- TRCCONFIGR
- TRCAUXCTRLR
- TRCEVENTCTL0R
- TRCEVENTCTL1R
- TRCRSR
- TRCSTALLCTRLR
If the trace unit has not been programmed since the last context switch then there is no requirement to save and restore the registers.
**K4.3 Trace examples**

**K4.3.1 Basic Examples**

The following example shows basic program trace.

**Table K4-4 Example of program trace**

<table>
<thead>
<tr>
<th>Execution</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
</table>
| 0x1000 B -> 0x2000 | trace_info(...) | Tracing begins here, therefore the trace unit must generate both:  
• A Context element.  
• A Target Address element.  
The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element. |
| 0x2000 MOV | | |
| 0x2004 LDR | | |
| 0x2008 CMP | | |
| 0x200C BEQ -> 0x3000 | atom(N) | This branch is not taken, so the trace unit generates an N Atom element. The N Atom element implies the execution of the three previous instructions and the BEQ instruction. |
| 0x2010 STR | | |
| | IRQ exception (IRQ,0x2014) | An IRQ occurs. The Exception element indicates the STR instruction was executed. |

The following example shows basic program trace when filtering is applied. In this example, the trace unit is programmed to exclude all instructions in the address range 0x2000 to 0x200F inclusive, and the trace unit is programmed to start tracing when the instruction at 0x1000 is accessed.

**Table K4-5 Example of program trace with filtering**

<table>
<thead>
<tr>
<th>Execution</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
</table>
| 0x1000 B -> 0x2000 | trace_info(...) | Tracing begins here, therefore the trace unit must generate both:  
• A Context element.  
• An Target Address element.  
The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element. |
| 0x2000 MOV | N | |
| 0x2004 LDR | N | |
| 0x2008 CMP | N | |
| 0x200C BEQ -> 0x3000 | N | |
| 0x2010 STR | Y trace_on() address(0x2010) | An IRQ occurs. The Exception element indicates the STR instruction was executed. |
| | IRQ exception (IRQ,0x2014) | |
The example in Table K4-6 on page K4-11661 shows basic program trace and demonstrates the canceling of some speculative execution because of an exception. In this example the trace unit is programmed to start tracing when the instruction at 0x1000 is accessed.

### Table K4-6 Example of basic program trace when an exception occurs, example one

<table>
<thead>
<tr>
<th>Execution</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000 B -&gt; 0x2000</td>
<td>trace_info()</td>
<td>Tracing begins here, therefore the trace unit must generate both:</td>
</tr>
<tr>
<td></td>
<td>trace_on()</td>
<td>• A Context element.</td>
</tr>
<tr>
<td></td>
<td>context()</td>
<td>• A Target Address element.</td>
</tr>
<tr>
<td></td>
<td>address(0x1000) atom(E)</td>
<td>The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element.</td>
</tr>
<tr>
<td>0x2000 MOV</td>
<td></td>
<td>None of these instructions are traced as P0 elements, therefore no trace elements are generated.</td>
</tr>
<tr>
<td>0x2004 LDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x2008 CMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x200C BEQ -&gt; 0x3000 (not taken)</td>
<td>atom(N)</td>
<td>This branch is not taken, so the trace unit generates an N Atom element. The N Atom element implies the execution of the three previous instructions and the BEQ instruction.</td>
</tr>
<tr>
<td>0x2010 STR</td>
<td></td>
<td>This instruction is not traced as a P0 element, therefore no trace element is generated.</td>
</tr>
<tr>
<td></td>
<td>cancel(1)</td>
<td>This cancels the N Atom element that was generated for the branch at 0x200C. The trace analyzer must discard the N Atom element, plus the three instructions that it implied.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Although PE execution has also canceled execution of the STR instruction, the trace analyzer is unaware of this, because the STR instruction was never traced. This is because no P0 elements were generated that would indicate execution of the STR instruction.</td>
</tr>
<tr>
<td>IRQ</td>
<td>exception(IRQ,0x2000)</td>
<td>The trace unit generates an Exception element with the address 0x2000, which indicates no instructions have executed since the target of the branch at 0x1000.</td>
</tr>
<tr>
<td></td>
<td>commit(2)</td>
<td>This commits the E Atom element that was generated for the Branch instruction at 0x1000, plus the Exception element that was generated for the IRQ exception.</td>
</tr>
</tbody>
</table>
The example in Table K4-7 on page K4-11662 shows basic program trace, and shows the trace generated when a synchronous Data Abort occurs. In this example the trace unit is programmed to start tracing when the instruction at 0x1000 is accessed.

<table>
<thead>
<tr>
<th>Execution</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
</table>
| 0x1000    | trace_info()   | Tracing begins here, therefore the trace unit must generate both:
|           | trace_on()     | • A Context element. |
|           | context()      | • A Target Address element. |
|           | address(0x1000)| The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element. |
|           | atom(E)        | |
| 0x2000    | MOV            | None of these instructions are traced as P0 elements, therefore no trace elements are generated. |
| 0x2004    | LDR            | |
| 0x2008    | CMP            | |
| 0x200C    | BEQ -> 0x3000  | This branch is not taken, so the trace unit generates an N Atom element. The N Atom element implies the execution of the three previous instructions and the BEQ instruction. |
|           | (not taken)    | |
|           | atom(N)        | |
| 0x2010    | STR            | This instruction is not traced as a P0 element, therefore no trace element is generated. |
| LDR aborts Cancel back to and including 0x2004 | cancel(1) | This cancels the N Atom element that was generated for the branch at 0x200C. The trace analyzer must discard the N Atom element, plus the four instructions that it implied. |
|           | exception_(data fault, 0x2004) | |
| commit all execution | commit(2) | This commits the E Atom element that was generated for the Branch instruction at 0x1000, plus the Exception element that was generated for the IRQ exception. |
The example in Table K4-8 on page K4-11663 extends the example shown in Table K4-7 on page K4-11662, and shows how exceptions are traced when two exceptions occur without any execution between them. In this example the trace unit is programmed to start tracing when the instruction at 0x1000 is accessed.

Table K4-8 Example of basic program trace when two consecutive exceptions occur

<table>
<thead>
<tr>
<th>Execution</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
</table>
| 0x1000 B -> 0x2000 | trace_info() trace_on() context() address(0x1000) atom(E) | Tracing begins here, therefore the trace unit must generate both:  
  • A Context element.  
  • A Target Address element.  
The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element. |
| 0x2000 MOV | None of these instructions are traced as P0 elements, therefore no trace elements are generated. |
| 0x2004 LDR | |
| 0x2008 CMP | |
| 0x200C BEQ -> 0x3000 (not taken) | atom(N) | This branch is not taken, so the trace unit generates an N Atom element. The N Atom element implies the execution of the three previous instructions and the BEQ instruction. |
| 0x2010 STR | This instruction is not traced as a P0 element, therefore no trace element is generated. |
| LDR aborts Cancel back to and including 0x2004 | cancel(1) exception_(data fault, 0x2004) | This cancels the N Atom element that was generated for the branch at 0x200C. The trace analyzer must discard the N Atom element, plus the four instructions that it implied.  
Note  
Although PE execution has also canceled execution of the STR instruction, the trace analyzer is unaware of this, because the STR instruction was never traced. This is because no P0 elements were generated that would indicate execution of the STR instruction.  
The data fault exception occurred at 0x2004. The Exception element indicates the MOV instruction at 0x2000 was executed.  
In summary:  
1. The MOV instruction was first implied by the N Atom element at 0x200C. However, the trace analyzer canceled this because of the Cancel element.  
2. The MOV instruction is now implied by the Exception element. |
| IRQ | address(0x4000) exception(IRQ, 0x4000) | This Exception element contains the address of the exception vector of the DataFault exception. This implies that no instructions have executed since the DataFault exception. |
| commit all execution | commit(3) | This commits the E Atom element that was generated for the Branch instruction at 0x1000, plus the Exception element generated for the Data fault exception and the Exception element that was generated for the IRQ exception. |
## K4.3.2 Examples of changes in context

When the PE executes an instruction that changes the execution context, the exact time at which the new element is traced depends on the PE operation after the write. An example of an instruction that changes the execution context is an instruction that writes a value to the CONTEXTIDR_EL1. See [Context element on page D4-4786](#) for more information about the rules controlling the generation of Context elements. This section provides examples of PE trace that contain changes of execution context to illustrate these rules. This section is split into the following:

- Exception in software executed after context synchronization.
- Exception immediately after ISB.
- Exception immediately before ISB.

Table K4-9 on page K4-11664 shows a write to the CONTEXTIDR_EL1 register, followed by an ISB to synchronize that write, followed by an exception that changes the context again.

### Table K4-9 Program trace containing a context changing operation

<table>
<thead>
<tr>
<th>Execution</th>
<th>Context</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
</table>
| 0x1000 B -> 0x2000 | 0xAA | trace_info() trace_on() context(0xAA) address(0x1000) atom(E) | Tracing begins here, therefore the trace unit must generate both:  
  - A Context element.  
  - A Target Address element.  
The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element. |
| 0x2000 MSR CONTEXTIDR | 0xAA | None of these instructions are traced as P0 elements, therefore no trace elements are generated. The instructions might be executed from context 0xAA or 0xBB but they are always traced as occurring from context 0xAA. |
| 0x2004 ADD | 0xAA or 0xBB | atom(E) | The trace unit generates an E Atom element, because the ISB is a Context synchronization event. All execution is traced as executing in context 0xAA. |
| 0x2008 ISB | 0xAA or 0xBB | atom(E) | A Context element is traced to indicate the new context. |
| 0x200C SUB | 0xBB | context(0xBB) | An IRQ exception occurs. The trace unit generates an Exception element. |
| IRQ | 0xBB | exception(IRQ,0x2010) | |
| 0x3000 B -> 0x4000 | 0xCC | context(0xCC) address(0x3000) atom(E) | A Context element is traced to indicate the new context.  
A Target Address element is also traced, because an Exception element is always followed by a Target Address element to indicate the address that the exception has been taken to.  
Finally, the instruction executed is a taken branch, so the trace unit must generate an E Atom element. |
Table K4-10 Program trace containing a context changing operation (exception immediately after ISB)

<table>
<thead>
<tr>
<th>Execution</th>
<th>Context</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
</table>
| 0x1000 B→ 0x2000 0xAA | trace_info() trace_on() context(0xAA) address(0x1000) atom(E) | Tracing begins here, therefore the trace unit must generate both:  
• A Context element.  
• A Target Address element.  
The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element. |
| 0x2000 MSR CONTEXTIDR 0xAA | | None of these instructions are traced as P0 elements, therefore no trace elements are generated. The instructions might be executed from context 0xAA or 0xBB but they are always traced as occurring from context 0xAA. |
| 0x2004 ADD 0xAA or 0xBB | | The trace unit generates an E Atom element, because the ISB is a Context synchronization event. All execution is traced as executing in context 0xAA. |
| 0x2008 ISB 0xAA or 0xBB atom(E) | | A Context element is traced to indicate the new context. An IRQ exception occurs. The trace unit generates an Exception element. |
| IRQ 0xBB | context(0xBB) exception(IRQ,0x200C) | | |
| 0x3000 B→ 0x4000 0xCC | context(0xCC) address(0x3000) atom(E) | A Context element is traced to indicate the new context.  
A Target Address element is also traced, because an Exception element is always followed by a Target Address element to indicate the address that the exception has been taken to.  
Finally, the instruction executed is a taken branch, so the trace unit must generate an E Atom element. |

Table K4-11 Program trace containing a context changing operation (exception immediately before ISB)

<table>
<thead>
<tr>
<th>Execution</th>
<th>Context</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
</table>
| 0x1000 B→ 0x2000 0xAA | trace_info() trace_on() context(0xAA) address(0x1000) atom(E) | Tracing begins here, therefore the trace unit must generate both:  
• A Context element.  
• A Target Address element.  
The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element. |
| 0x2000 MSR CONTEXTIDR 0xAA | | None of these instructions are traced as P0 elements, therefore no trace elements are generated. The instructions might be executed from context 0xAA or 0xBB but they are always traced as occurring from context 0xAA. |
Table K4-11 Program trace containing a context changing operation (exception immediately before ISB) (continued)

<table>
<thead>
<tr>
<th>Execution</th>
<th>Context</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>ADD</td>
<td>0x1000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IRQ</td>
<td>exception(IRQ,0x2008)</td>
<td>An IRQ exception occurs. The trace unit generates an Exception element.</td>
</tr>
<tr>
<td>0x3000</td>
<td>B -&gt; 0x4000</td>
<td>context(0x3000) address(0x1000) atom(E)</td>
<td>A Context element is traced to indicate the new context. A Target Address element is also traced, because an Exception element is always followed by a Target Address element to indicate the address that the exception has been taken to. Finally, the instruction executed is a taken branch, so the trace unit must generate an E Atom element.</td>
</tr>
</tbody>
</table>

K4.3.3 An example of the use of the trace unit return stack

This section contains two examples of tracing the same piece of program code. However:
- In the first example the trace unit return stack is disabled.
- In the second example trace unit the return stack is enabled.

The examples demonstrate that use of the trace unit return stack can help to reduce the amount of trace generated. Table K4-12 on page K4-11666 shows the first example, and Table K4-13 on page K4-11667 shows the second example. In these examples, the trace unit is programmed for basic program flow, where only branch instructions are traced as P0 instructions, and is programmed to start tracing when the instruction at 0x1000 is accessed.

Table K4-12 Basic program trace when Branch with Link instructions are executed and the return stack is disabled

<table>
<thead>
<tr>
<th>Execution</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>trace_info()</td>
<td>Tracing begins here, therefore the trace unit must generate both:</td>
</tr>
<tr>
<td></td>
<td>trace_on()</td>
<td>• A Context element.</td>
</tr>
<tr>
<td></td>
<td>context()</td>
<td>• A Target Address element.</td>
</tr>
<tr>
<td></td>
<td>address(0x1000)</td>
<td>The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element.</td>
</tr>
<tr>
<td></td>
<td>atom(E)</td>
<td></td>
</tr>
<tr>
<td>0x2000</td>
<td>MOV</td>
<td>None of these instructions are traced as P0 elements, therefore no trace elements are generated.</td>
</tr>
<tr>
<td>0x2004</td>
<td>LDR</td>
<td></td>
</tr>
<tr>
<td>0x2008</td>
<td>CMP</td>
<td></td>
</tr>
<tr>
<td>0x200C</td>
<td>BLEQ -&gt; 0x3000 (not taken)</td>
<td>This branch is not taken, so the trace unit generates an N Atom element. The N Atom element implies the execution of the three previous instructions and the BLEQ instruction.</td>
</tr>
<tr>
<td>0x2010</td>
<td>STR</td>
<td></td>
</tr>
<tr>
<td>0x2014</td>
<td>BX LR</td>
<td>This branch is taken, so the trace unit generates an E Atom element. The E Atom element implies the execution of the STR and the BX instructions.</td>
</tr>
</tbody>
</table>
### Table K4-12 Basic program trace when Branch with Link instructions are executed and the return stack is disabled

<table>
<thead>
<tr>
<th>Execution</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>MOV</td>
<td>address(0x1000) This instruction is not traced as a P0 element, therefore no trace element is generated. However, the last instruction executed was a taken indirect branch instruction, so the trace unit generates an <strong>Target Address element</strong> to indicate the target of that branch.</td>
</tr>
<tr>
<td>0x1008</td>
<td>B \rightarrow 0x4000</td>
<td>atom(E) This branch is taken, so the trace unit generates an E <strong>Atom element</strong>. The E <strong>Atom element</strong> implies the execution of the MOV instruction at 0x1004 and the B instruction.</td>
</tr>
</tbody>
</table>
| commit all execution | commit(4) | This commits all four of the following:  
  • The E **Atom element** generated for the branch at 0x1000.  
  • The N **Atom element** generated for the branch at 0x200C.  
  • The E **Atom element** generated for the branch at 0x2014.  
  • The E **Atom element** generated for the branch at 0x1008. |

### Table K4-13 Basic program trace when Branch with Link instructions are executed and the return stack is enabled

<table>
<thead>
<tr>
<th>Execution</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
</table>
| 0x1000    | BL \rightarrow 0x2000 | trace_info()  
 trace_on()  
 context()  
 address(0x1000)  
 atom(E) Tracing begins here, therefore the trace unit must generate both:  
  • A **Context element**.  
  • A **Target Address element**.  
 The instruction executed is a taken branch, so in addition, the trace unit must generate an E **Atom element**. In addition, because the return stack is enabled, the Branch with Link instruction causes the address 0x1004 to be pushed onto the trace unit return stack. |
| 0x2000    | MOV            | None of these instructions are traced as P0 elements, therefore no trace elements are generated. |
| 0x2004    | LDR            |       |
| 0x2008    | CMP            |       |
| 0x200C    | BLEQ \rightarrow 0x3000 (not taken) | atom(E) This branch is not taken, so the trace unit generates an N **Atom element**. The N **Atom element** implies the execution of the three previous instructions and the BLEQ instruction. Nothing is pushed onto the trace unit return stack because the branch is not taken. |
| 0x2010    | STR            |       |
| 0x2014    | BX LR          | atom(E) This branch is taken, so the trace unit generates an E **Atom element**. The E **Atom element** implies the execution of the STR and the BX instructions. |
K4.3.4 Transactions

The following is an example of a successful transaction traced by a trace unit with no speculation in the trace element stream.

<table>
<thead>
<tr>
<th>Execution</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0x1000$ MOV</td>
<td>This instruction is not traced as a \textit{P0 element}, therefore no trace element is generated. The address of this instruction matches the top entry on the trace unit return stack. Therefore, the trace analyzer knows to restart program execution here and an \textit{Target Address element} is not required. The top entry on the return stack, address $0x1004$, is popped from the return stack.</td>
<td></td>
</tr>
<tr>
<td>$0x1008$ B -&gt; $0x4000$ atom(E)</td>
<td>This branch is taken, so the trace unit generates an \textit{E Atom element}. The \textit{E Atom element} implies the execution of the \textit{MOV} instruction at $0x1004$ and the \textit{B} instruction.</td>
<td></td>
</tr>
</tbody>
</table>
| commit all execution | commit(4) | This commits all four of the following:  
  • The \textit{E Atom element} generated for the branch at $0x1000$.  
  • The \textit{N Atom element} generated for the branch at $0x200C$.  
  • The \textit{E Atom element} generated for the branch at $0x2014$.  
  • The \textit{E Atom element} generated for the branch at $0x1008$. |

<table>
<thead>
<tr>
<th>Execution</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0x1000$ B -&gt; $0x2000$ Trace_info(...) Trace_on() Target_address($0x2000$)</td>
<td>Trace on.</td>
<td></td>
</tr>
<tr>
<td>$0x2000$ TSTART Atom(E) Transaction_start()</td>
<td>The transaction starts.</td>
<td></td>
</tr>
<tr>
<td>$0x2004$ B -&gt; $0x2400$ Atom(E)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0x2804$ B -&gt; $0x2808$ Atom(E)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0x2808$ TCOMMIT Transaction_commit()</td>
<td>Transaction finishes.</td>
<td></td>
</tr>
<tr>
<td>$0x2808$</td>
<td>$0x2808$</td>
<td>$0x2808$</td>
</tr>
</tbody>
</table>
The following is an example of a failed transaction traced by a trace unit with no speculation in the trace element stream.

Table K4-15 Example of trace with a transaction failure

<table>
<thead>
<tr>
<th>Execution</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2000</td>
<td>TSTART</td>
<td>Trace_on()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target_address(0x2000)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Atom(E)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transaction_start()</td>
</tr>
<tr>
<td>0x2004</td>
<td>TST</td>
<td></td>
</tr>
<tr>
<td>0x2008</td>
<td>BEQ</td>
<td>Atom(N)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x2804</td>
<td>B -&gt; 0x3000</td>
<td>Atom(E)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target_address(0x3000)</td>
</tr>
</tbody>
</table>

Transaction fails

Transaction_failure()

This address is where execution resumes after the transaction failure.

0x2004 TST          Target_address(0x2004)

0x2008 BEQ          Atom(E)

{…}

The following is an example of trace with speculation in the trace element stream, and a failed transaction occurs. The speculative elements inside the transaction are cancelled using Cancel elements, and the transaction as a whole is indicated to have failed using a Transaction Failure element.

Table K4-16 Example of trace with a failed transaction

<table>
<thead>
<tr>
<th>Execution</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2000</td>
<td>TSTART</td>
<td>Trace_on()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target_address(0x2000)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Atom(E)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transaction_start()</td>
</tr>
<tr>
<td>0x2004</td>
<td>TST</td>
<td></td>
</tr>
<tr>
<td>0x2008</td>
<td>BEQ</td>
<td>Atom(N)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x2804</td>
<td>B -&gt; 0x3000</td>
<td>Atom(E)</td>
</tr>
</tbody>
</table>

Transaction fails

Transaction_failure()

Cancel(2)

Transaction fails.

0x2004 TST          Target_address(0x2004)

0x2008 BEQ          Atom(E)

{…}

The following is an example of a transaction which is started speculatively and is traced using speculative trace elements, but that speculation is subsequently resolved as misspeculation. The Cancel elements are used to cancel execution back past the start of the transaction.
### Table K4-17 Example of trace with a transaction failure

<table>
<thead>
<tr>
<th>Execution</th>
<th>Trace elements</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>{...}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1000</td>
<td>B -&gt; 0x2000</td>
<td>Atom(E)</td>
</tr>
</tbody>
</table>
| 0x2000    | TSTART         | Atom(E) | The transaction starts.  
|           |                | Transaction_start() | |
| 0x2004    | TST            |        | |
| 0x2008    | BEQ            | Atom(N) | |
| {...}     |                |       |
| 0x2804    | B -> 0x3000    | Atom(E) | The transaction was only speculatively started. It is optional if a Transaction Failure element is traced, because the Cancel element cancels the Transaction Start element. The Mispredict element corrects the incorrectly speculated first branch. |
|           | Cancel(4)      | Mispredict | |

0x1004    | {...}         |       |
| {...}    |                |       |
ETE and TRBE Software Usage Examples

K4.4 Differences between ETM and ETE

This section describes the primary functional differences between ETMv4 and ETE.

- Removal of data trace documentation, since this is not permitted in A-profile.
- Removal of conditional non-branch documentation, since this is not permitted in A-profile.
- TRCDEVARCH.PRESENT == 1 is mandatory.
- TRCDEVARCH.ARCHVER and TRCDEVARCH.REVISION take new values.
- TRCIFR1.TRCARCHMAJ and TRCIFR1.TRCARCHMIN take new values.
- TRCIDR9 is fixed at zero.
- Context identifier tracing is mandatory, defined in TRCIDR2.CIDSIZE.
- Virtual context identifier tracing is mandatory when the PE implements EL2, defined in TRCIDR2.VMIDSIZE.
- The Virtual context identifier is always based on CONTEXTIDR_EL2, with support for tracing VTTBR_EL2.VMID removed.
- 64-bit timestamp is the only supported timestamp size.
- Timestamping is mandatory in ETE.
- TRCIDR2.IASIZE is only permitted to indicate a 64-bit instruction address size.
- External Inputs are unified with the PMU event space, with new TRCEXTINSEL<n> registers introduced.
- TRCIDR5.NUMEXTIN indicates the unified External Input model.
- Added TRCRSR.EXITIN for reading and setting the External Input Selectors state.
- Added TRCRSR.EVENT for reading and setting the ETEEvent state.
- Added TRCRSR.TA for reading and setting whether tracing was active.
- Changed requirements for the tracing of Exceptions to be dependent on the new TRCRSR.TA field.
- Removal of memory-mapped accesses. This was deprecated in ETMv4.4 for Armv8-A.
- Removal of trace unit sharing.
- Added a requirement that trace must be output within finite time.
- Added a requirement that the trace unit resources are paused when entering a Trace Prohibited region.
- Added a bit to TRCSSCSR<n> to indicate that the Single-shot Comparator Control fired while the resources are paused.
- Added requirements for dependencies on the TSB CSYNC instruction.
- Execution of TSB CSYNC instruction requests a timestamp element.
- The Unified Power Domain Model from ETMv4.5 for Armv8-A is mandatory in ETE.
- Changes to the enable and disable code sequences.
- Addition of the tracing of Transactional state.
- Tightened the requirements for obeying the order of start point and stop points for the ViewInst start/stop function, and tightened the rules for programming the start/stop function.
- Addition of Source Address elements.
- Added rules to require no trace to be generated in Trace Prohibited regions, under some circumstances.
- Added constraints for the effect of system instructions causing the trace unit to become enabled or disabled.
- Additional constraints for the forced tracing of exceptions around Trace Prohibited regions, to ensure trace is not generated in Trace Prohibited regions.
- Removed the flexibility around tracing of an Exceptional occurrence immediately after a Trace Prohibited region or when trace generation becomes operative. Such Exceptional occurrences are not traced.
- Added a requirement that the resource operations must complete before a TSB instruction completes.
• Defined the behavior of the visibility of reads and writes to trace unit registers from system instructions, external debugger and by the trace unit.
• Changed branch broadcasting to be required in all implementations, see TRCIDR0.TRCCB.
• TRCSYNCPR is read/write in all implementations, see TRCIDR3.SYNCPR.
• Forced tracing of System Error exceptions is required in all implementations, see TRCIDR3.TRCERR.
• Changed cycle counting to be required in all implementations, see TRCIDR0.TRCCCI.
• Removed the trace unit OS Lock mechanism, and changed to require the PE OS Lock to affect the trace unit.
• Removed the Exception Return element and Exception Return packet.
• Constrained TRCCLAIMCLR and TRCCLAIMSET to not require explicit synchronization.
• Added more constraints to the operation of the Single-shot Comparator Controls when the trace unit becomes disabled, or when entering a Trace Prohibited region.
• Added more constraints to the operation of the ViewInst start/stop function when the trace unit becomes disabled, or when entering a Trace Prohibited region.
• Constrained the behavior of cycle counting after a trace unit buffer overflow, to require the cycle count to be traced as UNKNOWN on the first Cycle Count element after an overflow.
• Export of PMU events to the trace unit is not affected by PMCR.X or PMCR_EL0.X.
• Event elements are permitted to be generated before the first Trace Info element.
K4.5  Context switching

When switching out a process being traced, to save the current trace context and ensure all trace operations are written to the correct context:

1. Prohibit program-flow trace using TRFCR_ELx. In many cases this is done before the process is traced. For example if all of the following are true:
   - TRFCR_EL1.E0TRE is 1 to allow tracing of a process executed at EL0.
   - TRFCR_EL1.E1TRE is 0 to prohibit tracing of the Operating System performing the context switch executed at EL1.

2. Execute a Context synchronization event to guarantee no new program-flow trace is generated. In the common case, this Context synchronization event is an exception taken to an Exception level where tracing is prohibited.
   - If the trace unit is an ETE and the ETE is enabled, this also pauses the ETE Resources.

3. Execute a TSB CSYNC instruction to ensure the program-flow trace is flushed.

4. If necessary, disable the trace unit.
   - For an ETE this is necessary if context is being switched. Software must set TRCPRGCTLR.EN to zero. This is necessary as:
     - The ETE must be disabled if saving the ETE state, as the ETE System registers can only be read when the ETE is disabled.
     - ETE trace compression logic is stateful, and disabling the ETE resets this compression state.

5. Disable the Trace Buffer Unit. Set TRBLIMITR_EL1.E to zero.
   - This must be done before changing the VMSA System registers to prevent the Trace Buffer Unit from speculatively accessing translation table entries.

6. Execute a DSB operation.
   - This is required if software will be reading the trace buffer contents, to ensure the writes to memory are Complete.

7. Execute a further Context synchronization event.
   - This is required to synchronize the effects of any System register writes since the previous Context synchronization event.
   - This is also required if software will be reading the Trace Buffer Unit or trace unit System registers as part of the context switch, to capture indirect writes to those registers by trace operations synchronized by the TSB CSYNC.
   - For a subsequent direct read to capture the indirect write to TRBSR_EL1 resulting from an External abort on a completed write, this Context synchronization event must follow the DSB above.

8. Save and/or change the context. For example, save the MDCR_EL3, MDCR_EL2 (if applicable), Trace Buffer Unit, trace unit, and TRFCR_ELx System registers, and update the VMSA System registers for the new process.

In other uses cases, tracing is not prohibited when software wants to save the trace context. For this case, if using an ETE, the sequence is slightly different:

1. Disable the ETE. Set TRCPRGCTLR.EN to zero.
2. Execute a Context synchronization event to guarantee no new program-flow trace is generated.
3. Execute a TSB CSYNC instruction to ensure the program-flow trace is flushed.
4. Execute a DSB and/or Context synchronization event as required by the previous example.
5. Save and/or change the context.

For an ETE this sequence does not guarantee that all instructions before disabling the ETE are traced. The ETE might discard trace for preceding instructions when it is disabled.
To restore the state of the Trace Buffer Unit and trace unit for switching in a process being traced, while tracing is prohibited:

1. Restore the context. For example:
   - Restore MDCR_EL3, MDCR_EL2 (if applicable), and the Trace Buffer Unit System registers, other than TRBLIMITR_EL1.
   - Restore the trace unit System registers, other than enabling the trace unit.
   - Ensure the TRFCR_ELx System registers are correct for the process being traced.
   - Update VMSA System registers for the process being returned to.

2. Execute a Context synchronization event to guarantee the trace unit and Trace Buffer Unit will observe the new values of the System registers written by the previous step.

3. Enable the Trace Buffer Unit by setting TRBLIMITR_EL1.E to 1. This must be done after setting up the correct VMSA System registers for the trace buffer, as the Trace Buffer Unit might now speculatively prefetch and cache address translations. See BSMLW and YKCND.

4. If necessary, enable the trace unit. If using an ETE, software must set TRCPRGCTRLR.EN to one.

5. Execute a Context synchronization event to guarantee tracing is allowed. In the common case, this is an ERET instruction that returns to a different Exception level where tracing is allowed.

This must be done after saving the state from the previous process, if applicable.

Because the Trace Buffer Unit can prefetch and cache address translations when the Trace Buffer Unit is enabled:

- Software must not enable the Trace Buffer Unit before programming the System registers for the owning translation regime. In particular, during a context switch operation:
  - If switching from a context using the Trace Buffer Unit, the Trace Buffer Unit must be disabled before modifying the System registers for the owning translation regime being switched from.
  - If switching to a context using the Trace Buffer Unit, the Trace Buffer Unit must not be enabled until after modifying the System registers for the owning translation regime being switched to.

- The Trace Buffer Unit must not be enabled when the PE is not executing in the owning Security state or when executing at EL3 and SCR_EL3.NS does not indicate the owning Security state.

- In normal conditions, enabling the Trace Buffer Unit early before returning to the context being traced might be advantageous if the implementation does prefetch address translations.
K4.6 Controlling generation of trace buffer management events

The TRBE does not include a direct capability to program the Trace Buffer Unit to generate a maintenance interrupt when the trace buffer reaches a programmed level below the Limit pointer, and continue collecting trace until either the interrupt is serviced or (possibly) the trace buffer fills (whichever comes first). This allows an almost lossless collection of trace.

However, the Trace Buffer Unit can be programmed to give similar behavior in one of the following ways:

1. Using Wrap mode. At the start of a trace session, configure the Base pointer and Limit pointer for the trace buffer as normal, but set the trace buffer mode to Wrap mode the current write pointer to point part way through the trace buffer, such that the remaining space in the trace buffer is the watermark level. When the amount of trace collected reaches the watermark level, the current write pointer is wrapped and a trace buffer management event is generated, but trace continues to be collected. This approach has the following advantages and disadvantages:
   • The trace buffer management event is generated and the trace unit receives the TRB_WRAP event at the watermark level.
   • The oldest trace in the trace buffer will be lost if more trace is generated than fits in the trace buffer, because it is overwritten by newer trace. Note that some loss of trace is inevitable if more trace is generated than fits in the trace buffer.
   • The trace history does not start at the start of the trace buffer, and must be aligned by software.

2. Use a Trigger Event. At the start of a trace session, configure the Base pointer and Limit pointer for the trace buffer as normal, and set the trace buffer mode to Fill mode and the current write pointer to the start of the trace buffer. Set the trigger mode to IRQ on trigger, the Trigger Counter to the watermark level, and TRBSR_EL1.TRG to 1. When the amount of trace collected reaches the watermark level, a Trigger Event occurs and a trace buffer management event is generated, but trace continues to be collected. This approach has the following advantages and disadvantages:
   • The trace buffer management event is generated and the trace unit receives the TRB_TRIG event at the watermark level.
   • The newest trace in the trace buffer will be discarded if more trace is generated than fits in the trace buffer. To overwrite the oldest trace instead, set the trace buffer mode to Circular Buffer mode.
   • This method cannot be used if also searching for a Detected Trigger event from the trace unit.
   • The current write pointer does not have to be set to the start of the trace buffer. If the trace buffer already contains data that software does not want to be overwritten, the current write pointer can be set to point to after this data. In this case using Circular Buffer mode or Stop on trigger can also be used to control when collection is stopped and what data is overwritten.
K4.6 Controlling generation of trace buffer management events
Appendix K5
Stages of execution

This appendix shows the relationship between the stages of execution and the Transactional Memory Extension. It contains the following sections:

• *Stages of execution without the Transactional Memory Extension (TME)* on page K5-11678.
• *Stages of execution with the TME* on page K5-11679.
K5.1  Stages of execution without the Transactional Memory Extension (TME)

Figure K5-1 shows the stages of execution in a PE that does not implement FEAT_TME.
K5.2 Stages of execution with the TME

Figure K5-2 shows the stages of execution in a PE that does implement FEAT_TME.
Stages of execution
K5.2 Stages of execution with the TME
Appendix K6
Recommended External Debug Interface

This appendix describes the recommended external debug interface. It contains the following sections:

• About the recommended external debug interface on page K6-11682.
• PMUEVENT bus on page K6-11686.
• Recommended authentication interface on page K6-11687.
• Management registers and CoreSight compliance on page K6-11688.

Note

This recommended external debug interface specification is not part of the Arm architecture specification. Implementers and users of the architecture must not consider this appendix as a requirement of the architecture. It is included as an appendix to this manual only:

• As reference material for users of Arm products that implement this interface.
• As an example of how an external debug interface might be implemented.

The inclusion of this appendix is no indication of whether any Arm products might, or might not, implement this external debug interface. For details of the implemented external debug interface, you must always see the appropriate product documentation.
K6.1 About the recommended external debug interface

See the Note on the first page of this appendix for information about the architectural status of this recommended debug interface.

This specification provides a recommended external debug interface for Armv8 and later architectures to define a standard set of connections for validation environments. In general, the connection between components, such as between the PE and Trace extension, is not described here, although the table does include the signals for the CTI connection. Table K6-1 on page K6-11682 shows the signals in the recommended interface.

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBGEN</td>
<td>In</td>
<td>External debug enable</td>
<td></td>
</tr>
<tr>
<td>SPIDEN</td>
<td>In</td>
<td>Secure privileged external debug enable</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Secure privileged self-hosted debug enable</td>
<td>Only in Secure AArch32 modes when enabled by MDCR_EL3.SPDM32</td>
</tr>
<tr>
<td>NIDEN</td>
<td>In</td>
<td>External profiling and trace enable</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>If FEAT_Debugv8p4 is implemented, this signal is not implemented.</td>
<td></td>
</tr>
<tr>
<td>SPNIDEN</td>
<td>In</td>
<td>Secure external profiling and trace enable</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>If FEAT_Debugv8p4 is implemented, this signal is not implemented.</td>
<td></td>
</tr>
<tr>
<td>EDBGRQ</td>
<td>In</td>
<td>External halt request</td>
<td>IMPLEMENTATION DEFINED mechanism to halt the PE. See $EDBGRQ and DBGACK$ on page K6-11685.</td>
</tr>
<tr>
<td>DBGACK</td>
<td>Out</td>
<td>Debug Acknowledge</td>
<td>Indicate to the system that a PE is in Debug state. See $EDBGRQ and DBGACK$ on page K6-11685.</td>
</tr>
<tr>
<td>COMMIRQ</td>
<td>Out</td>
<td>DCC interrupt</td>
<td>Interface to an interrupt controller. See $Interrupt-driven use of the DCC on page H4-10314$ and the pseudocode for function $CheckForDCCInterrupts()$.</td>
</tr>
<tr>
<td>PMUIRQ</td>
<td>Out</td>
<td>Performance Monitor overflow</td>
<td>Interface to an interrupt controller. See $Behavior on overflow on page D11-5250$.</td>
</tr>
<tr>
<td>COMMRX</td>
<td>Out</td>
<td>DTRRX is full</td>
<td>Provided for legacy connection to an interrupt controller only. See $Interrupt-driven use of the DCC on page H4-10314$ and the pseudocode for function $CheckForDCCInterrupts()$.</td>
</tr>
<tr>
<td>COMMTX</td>
<td>Out</td>
<td>DTRTX is empty</td>
<td></td>
</tr>
<tr>
<td>DBGNOPWRDWN</td>
<td>Out</td>
<td>Emulate low-power state request</td>
<td>Interface to a power controller. See $Emulating low-power states on page H6-10340$.</td>
</tr>
<tr>
<td>DBGPWRUPREQ</td>
<td>Out</td>
<td>Core powerup request</td>
<td>Interface to a power controller. See $Powerup request mechanism on page H6-10338$.</td>
</tr>
<tr>
<td>DBGRSTREQ</td>
<td>Out</td>
<td>Warm reset request</td>
<td>Interface to a power controller. See $EDPRCR.CWRR$.</td>
</tr>
</tbody>
</table>
### Table K6-1 Recommended debug interface signals (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBGBUSCANCELEX</td>
<td>Out</td>
<td>Allow asynchronous entry to Debug state</td>
<td>Extension to the bus interface. See EDRCR.CBRRQ.</td>
</tr>
<tr>
<td>DBGPWRRDUP</td>
<td>In</td>
<td>Core powerup status</td>
<td>Interface to a power controller. See EDPRSR.PU.</td>
</tr>
<tr>
<td>DBGROMADDR[n:12]</td>
<td>In</td>
<td>MDRAR_EL1.ROMADDR</td>
<td>$n$ depends on the size of the physical address space. Arm recommends these signals are tied LOW.</td>
</tr>
<tr>
<td>DBGROMADDRV</td>
<td>In</td>
<td>MDRAR_EL1.Valid</td>
<td>Arm recommends these signals are tied LOW.</td>
</tr>
<tr>
<td>PRESETEDBG</td>
<td>In</td>
<td>External debug reset</td>
<td>-</td>
</tr>
<tr>
<td>CPUPORESET</td>
<td>In</td>
<td>Cold reset</td>
<td>-</td>
</tr>
<tr>
<td>CORERESET</td>
<td>In</td>
<td>Warm reset</td>
<td>-</td>
</tr>
<tr>
<td>PSELDDBG</td>
<td>In</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PENABDBG</td>
<td>In</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PWRIEDDBG</td>
<td>In</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRDATADBG[31:0]</td>
<td>Out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PWDATADBG[31:0]</td>
<td>In</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PADDRDBG[n:2]b</td>
<td>In</td>
<td>Debug APB interface</td>
<td></td>
</tr>
<tr>
<td>PREADYDBG</td>
<td>Out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSLVERRDBG</td>
<td>Out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCLKDBG</td>
<td>In</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCLKENDBG</td>
<td>In</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PPRTOTDBG[1]</td>
<td>In</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTICHIN</td>
<td>In</td>
<td>CoreSight channel interface</td>
<td>For details, see Arm® CoreSight™ Architecture Specification. The ACK signals are not required if the channel interface is synchronous.</td>
</tr>
<tr>
<td>CTICHOUTACK</td>
<td>In</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTICHOUT</td>
<td>Out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTICHINACK</td>
<td>Out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTIRQ</td>
<td>Out</td>
<td>CTI interrupt, see Description and allocation of CTI triggers on page H5-10325</td>
<td>Implements a handshake for an edge-sensitive interrupt.</td>
</tr>
<tr>
<td>CTIRQACK</td>
<td>In</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATDATA[nx8-1:0]</td>
<td>Out</td>
<td>AMBA 4 ATB interface</td>
<td>For details, see the AMBA 4 ATB Protocol Specification, ATBv1.0 and ATBv1.1. Only available if the OPTIONAL Trace extension is implemented.</td>
</tr>
<tr>
<td>ATBYTES[n-1:0]</td>
<td>Out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATID[6:0]</td>
<td>Out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATREADY</td>
<td>In</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table K6-1 Recommended debug interface signals (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATVALID</td>
<td>Out</td>
<td>AMBA 4 ATB interface(^c)</td>
<td>For details, see the AMBA 4 ATB Protocol Specification, ATBv1.0 and ATBv1.1. Only available if the OPTIONAL Trace extension is implemented.</td>
</tr>
<tr>
<td>AFREADY</td>
<td>Out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AFVALID</td>
<td>Out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYNCREQ</td>
<td>In</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATCLK</td>
<td>In</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATCLKEN</td>
<td>In</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATRESET</td>
<td>In</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. This is the port where the PE completes debug APB transactions. Arm recommends a single port for all integrated debug components.
b. The value of \( n \) depends on the size of the address space occupied by the Debug port.
c. This is the port where the PE outputs trace.

Figure K6-1 shows the recommended debug interface.

![Recommended External Debug Interface Diagram](image-url)

Figure K6-1 Recommended external debug interface, including the APB4 Completer port
K6.1.1 EDBGRQ and DBGACK

EDBGRQ is an IMPLEMENTATION DEFINED means of generating the External Debug Request debug event described in External Debug Request debug event on page H3-10291.

The PE asserts DBGACK when the PE is in Debug state. The PE might also include variants of this signal:

DBGTRIGGER
Asserted by the PE when it commits to entering Debug state.

DBGCPUDONE
Asserted by the PE when it has completed all Non-debug state memory accesses and Debug state entry is complete. DBGCPUDONE indicates that memory accesses issued by the PE result from operations originating from debugger commands.

In previous architecture versions, these signals provide an interface between the PE and cross-trigger logic. In Armv8, the architectural Cross-Trigger Interface provides this functionality for external debuggers.

K6.1.2 Secure and Non-secure views of the debug registers

If FEAT_Debugv8p4 is implemented, the external debug interface has views of Secure and Non-secure debug registers. The DAP must ensure that accesses are made only when permitted. The Arm debug interface describes a standard APB-AP programmers model for APB4 which signals Secure and Non-secure accesses on the PPRTDBG[1] signal, and is recommended for new designs.

If FEAT_Debugv8p4 is implemented, and an APB-AP implements an APB3 Requester port, which does not support Secure and Non-secure views, Arm recommends that the following is implemented:

• If SPIDEN is HIGH and DBGEN is HIGH, all external debug accesses are treated as Secure.
• If either SPIDEN is LOW or DBGEN is LOW, all external debug accesses are treated as Non-secure.

If the PE APB Completer port is APB4, this might be implemented by, for example, fixing PPRTDBG[1] to the inverse of (SPIDEN & DBGEN) when bridging from APB3 to APB4.
# K6.2 PMUEVENT bus

The **PMUEVENT** bus exports Performance Monitor events from the PE to an on-chip agent. Arm recommends that it has the following characteristics:

- The bus is synchronous.
- The width of the bus is IMPLEMENTATION DEFINED.
- It is IMPLEMENTATION DEFINED which events are exported on the bus.
- Each exported event occupies a contiguous sub-field of the bus. Arm recommends that the sub-fields of the bus are occupied in the same order as the event numbers.
- If the event can only occur once per cycle, it occupies a single bit. If the event can occur more than once per cycle, it is IMPLEMENTATION DEFINED how the event is encoded. The encoding depends on constraints such as the designated use of the event bus and the number of pins available. For example, the event can be encoded:
  - As a count, using a plain binary number. This is the most useful encoding when exporting to an external counter. It is not a useful encoding for exporting to a Trace extension external input.
  - As a count, using thermometer encoding. This is the most useful encoding when exporting to a Trace extension.
  - Using a single bit encoding to indicate whether the event count is zero or nonzero. This is useful for exporting to an activity monitor where the number of pins is constrained.

If an ETMv4 Trace extension is implemented, the **PMUEVENT** bus is normally connected to the Trace extension using the external inputs. TRCEXTINSELR multiplexes a wide **PMUEVENT** bus to a narrow set of inputs. An external **PMUEVENT** bus might also be provided. For more information, contact Arm.

When **FEAT_ETE** is implemented, PMU events are connected to the trace unit, however the PMUEVENT bus is not used to connect the PMU events to the trace unit. See *External inputs on page D4-4870.*
Recommended authentication interface

An implementation of the Armv8 and later architectures must support debug authentication described in Required debug authentication on page H1-10230.

The details of the debug authentication interface are IMPLEMENTATION DEFINED, but Arm recommends the use of the CoreSight interface, which includes the following signals for external debug authentication:

- **DBGEN**.
- **SPIDEN**.

If FEAT_Debugv8p4 is not implemented, Arm also recommends using the following signals:

- **NIDEN**.
- **SPNIDEN**.

Arm recommends an interface in which **DBGEN** and **SPIDEN** are also used for self-hosted Secure debug authentication if either:

- EL3 is using AArch32 and SDCR.SPD == 0b00.
- Secure EL1 is using AArch32 and MDCR_EL3.SPD32 == 0b00.

If EL3 is not implemented and the PE is in Non-secure state, **SPIDEN** and **SPNIDEN** are not implemented, and the PE behaves as if these signals were tied LOW.

If EL3 is not implemented and the PE is in Secure state, **SPIDEN** is usually connected to **DBGEN** and **SPNIDEN** is connected to **NIDEN**, but this is not required. The recommended interface is defined as if all four signals are implemented.

How the authentication signals are driven is IMPLEMENTATION DEFINED. For example, the signals might be hard-wired, connected to fuses, or to an authentication module. The architecture permits PEs within a cluster to have independent authentication interfaces, but this is not required. Arm recommends that any Trace extension has the same authentication interface as the PE it is connected to.

If FEAT_Debugv8p4 and CoreSight ETR are both implemented, the ETR has an independent **DBGEN** signal that must be tied HIGH to enable self-hosted use of trace.

Table K6-2 on page K6-11687 shows the debug authentication pseudocode functions and the recommended implementations.

Table K6-2 Recommended implementation of debug enable pseudocode functions

<table>
<thead>
<tr>
<th>Pseudocode function</th>
<th>Description</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()</td>
<td>Secure invasive self-hosted debug enabled in AArch32 state (legacy)</td>
<td>(DBGEN AND SPIDEN)</td>
</tr>
<tr>
<td>ExternalSecureNoninvasiveDebugEnabled()a</td>
<td>Secure non-invasive debug enabled</td>
<td>(DBGEN OR NIDEN)(^b) AND (SPIDEN OR SPNIDEN)(^c)</td>
</tr>
<tr>
<td>ExternalSecureInvasiveDebugEnabled()</td>
<td>Secure invasive debug enabled</td>
<td>(DBGEN AND SPIDEN)</td>
</tr>
<tr>
<td>ExternalNoninvasiveDebugEnabled()d</td>
<td>Non-secure non-invasive debug enabled</td>
<td>(DBGEN OR NIDEN)(^b)</td>
</tr>
<tr>
<td>ExternalInvasiveDebugEnabled()</td>
<td>Non-secure invasive debug enabled</td>
<td>DBGEN</td>
</tr>
</tbody>
</table>

a. If FEAT_Debugv8p4 is implemented, ExternalSecureNoninvasiveDebugEnabled() == ExternalSecureInvasiveDebugEnabled().

b. If FEAT_Debugv8p4 is implemented, the NIDEN signal is not implemented.

c. If FEAT_Debugv8p4 is implemented, the SPNIDEN signal is not implemented.

d. If FEAT_Debugv8p4 is implemented, ExternalNoninvasiveDebugEnabled() == TRUE.

The Debug_authentication() pseudocode function on shared/debug on page J1-11354 defines the authentication signals DBGEN, SPIDEN, NIDEN, and SPNIDEN.
K6.4 Management registers and CoreSight compliance

The CoreSight architecture requires the implementation of a set of management registers that occupy the memory map from 0xF00 upwards in each of the debug components.

CoreSight compliance and complete implementation of the management registers is OPTIONAL, but Arm recommends that the registers are implemented.

The CoreSight architecture specification recommends that any integration test registers are implemented starting from 0xEFC downwards. Each of the debug components has an IMPLEMENTATION DEFINED region from 0xE80 to 0xEFC for this purpose.

K6.4.1 CoreSight interface register map

Table K6-3 on page K6-11688 shows the external management register maps for the following registers:

- **ED**: These are the external debug register.
- **CTI**: These are the Cross-trigger interface registers.
- **PMU**: These are the Performance Monitors registers.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Mnemonic</th>
<th>ED</th>
<th>CTI</th>
<th>PMU</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF00</td>
<td>EDITCTRL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Integration Model Control registers</td>
</tr>
<tr>
<td>0xF04-0xF0C</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Reserved, RES0</td>
</tr>
<tr>
<td>0xFA0</td>
<td>DBGCLAIMSET_EL1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLAIM Tag Set registers</td>
</tr>
<tr>
<td>0xFA4</td>
<td>DBGCLAIMCLR_EL1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>CLAIM Tag Clear registers</td>
</tr>
<tr>
<td>0xFA8</td>
<td>EDDEVAFF0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Device Affinity registers</td>
</tr>
<tr>
<td>0x0AC</td>
<td>EDDEVAFF1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Device Affinity registers</td>
</tr>
<tr>
<td>0xFB0</td>
<td>EDLAR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Lock Access register</td>
</tr>
<tr>
<td>0xFB4</td>
<td>EDLSR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Lock Status register</td>
</tr>
<tr>
<td>0xFB8</td>
<td>DBGAUTHSTATUS_EL1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Authentication Status register</td>
</tr>
<tr>
<td>0xFC0</td>
<td>EDDEVID2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Device Architecture register</td>
</tr>
<tr>
<td>0xFC4</td>
<td>EDDEVID1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Device ID register</td>
</tr>
<tr>
<td>0xFC8</td>
<td>EDDEVID</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Device Type register</td>
</tr>
<tr>
<td>0xFC0</td>
<td>EDPIDR4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Peripheral ID registers</td>
</tr>
<tr>
<td>0xFD0</td>
<td>EDPIDR0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Peripheral ID registers</td>
</tr>
<tr>
<td>0xFE4</td>
<td>EDPIDR1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Peripheral ID registers</td>
</tr>
<tr>
<td>0xFE8</td>
<td>EDPIDR2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Peripheral ID registers</td>
</tr>
<tr>
<td>0xFEC</td>
<td>EDPIDR3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Peripheral ID registers</td>
</tr>
</tbody>
</table>
K6.4.2 Management register access permissions

Access to the OPTIONAL Integration Control register (ITCTRL) is IMPLEMENTATION DEFINED.

Table K6-4 on page K6-11690, Table K6-5 on page K6-11691, Table K6-6 on page K6-11692, Table K6-7 on page K6-11693, and Table K6-8 on page K6-11694 show the response to accesses by the external debug interface to the CoreSight management registers.

Note

Access to the CoreSight management registers is not affected by the values of EDAD and EPMAD.

Table K6-4 on page K6-11690, Table K6-5 on page K6-11691, Table K6-6 on page K6-11692, Table K6-7 on page K6-11693, and Table K6-8 on page K6-11694 include reserved management registers, because the CoreSight architecture requires that these registers are always RES0. The descriptions in Reserved and unallocated registers on page H8-10366 do not apply to reserved management registers if the implementation is CoreSight compliant.

If OPTIONAL memory-mapped access to the external debug interface is supported, there are additional constraints on memory-mapped accesses. See Register access permissions for memory-mapped accesses on page H8-10362.

When HaveSecureExtDebugView() == TRUE, each debug component has a Secure and Non-secure view. The Secure view of a debug component is mapped into Secure physical memory and the Non-secure view of a debug component is mapped into Non-secure memory. Apart from access conditions, the Non-secure and Secure views of the debug components are identical.

The terms in Table K6-4 on page K6-11690, Table K6-5 on page K6-11691, Table K6-6 on page K6-11692, Table K6-7 on page K6-11693, and Table K6-8 on page K6-11694 are defined as follows:

**Domain**
This describes the power domain in which the register is logically implemented. Registers described as implemented in the Core power domain might be implemented in the Debug power domain, as long as they exhibit the required behavior.

If FEAT_DoPD is implemented, most External debug interface registers are in the Core power domain, as shown in Table K6-4 on page K6-11690 and Table K6-7 on page K6-11693.

If FEAT_DoPD is not implemented, most of the registers are in the Debug Power Domain, as shown in Table K6-5 on page K6-11691 and Table K6-8 on page K6-11694.

**Conditions**
This lists the conditions under which the access is attempted.

To determine the access permissions for a register, read these columns from left to right, and stop at first column that lists the condition as being true.
The conditions are:

**Off**  
\[ \text{EDPRSR.PU} = 0. \] The Core power domain is completely off, or in low-power state. In these cases, the Core power domain registers cannot be accessed.

--- **Note** ---

When the Debug power domain is off, all accesses to the registers in the external Debug power domain return an error.

**DLK**  
If the OS Double Lock is implemented and `DoubleLockStatus() == TRUE`. The OS Double Lock is locked.

**OSLK**  
\[ \text{OSLSR.OSLK} = 1. \] The OS Lock is locked.

**Default**  
This provides the default access permissions, if there are no conditions that prevent access to the register.

**SLK**  
This provides the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the OPTIONAL Software Lock is locked. See *Register access permissions for memory-mapped accesses* on page H8-10362. If FEAT_DoPD is implemented, the Software Lock is not locked, or not implemented, this column is ignored.

The access permissions are:

- **-**  
  This means that the default access permission applies. See the Default column, or the SLK column, if applicable.

- **RO**  
  This means that the register or field is read-only.

- **RW**  
  This means that the register or field is read/write. Individual fields within the register might be RO. See the relevant register description for details.

- **RC**  
  This means that the bit clears to 0 after a read.

- **(SE)**  
  This means that accesses to this register have indirect write side effects. A side effect occurs when a direct read or a direct write of a register creates an indirect write to the same register or to another register.

- **WO**  
  This means that the register or field is write-only.

- **WI**  
  This means that the register or field ignores writes.

- **IMP DEF**  
  This means that the access permissions are IMPLEMENTATION DEFINED.

---

**Table K6-4 External debug interface access permissions, CoreSight registers (debug) if FEAT_DoPD is implemented**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Conditions (priority left to right)</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF00</td>
<td>EDITCTRL</td>
<td>IMP DEF</td>
<td>IMPLEMENTATION DEFINED</td>
<td>IMP DEF</td>
</tr>
<tr>
<td>0xF04-0xF8C</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
<td>RES0</td>
</tr>
<tr>
<td>0xFA0</td>
<td>DBGCLAIMSET_EL1</td>
<td>Core</td>
<td>Error</td>
<td>Error</td>
</tr>
<tr>
<td>0xFA4</td>
<td>DBGCLAIMCLR_EL1</td>
<td>Core</td>
<td>Error</td>
<td>-</td>
</tr>
<tr>
<td>0xFA8</td>
<td>EDDEVAFF0</td>
<td>Core</td>
<td>Error</td>
<td>-</td>
</tr>
<tr>
<td>0xFC0</td>
<td>EDDEVAFF1</td>
<td>Core</td>
<td>Error</td>
<td>-</td>
</tr>
<tr>
<td>0xFB0</td>
<td>EDLAR</td>
<td>Core</td>
<td>Error</td>
<td>-</td>
</tr>
</tbody>
</table>
## Table K6-4 External debug interface access permissions, CoreSight registers (debug) if FEAT_DoPD is implemented (continued)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Conditions (priority left to right)</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFB4</td>
<td>EDLSR</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFB8</td>
<td>DBGAUTHSTATUS_EL1</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFBc</td>
<td>EDDEVARCH</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFC0</td>
<td>EDDEVID2</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFC4</td>
<td>EDDEVID1</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFC8</td>
<td>EDDEVID</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFCC</td>
<td>EDDEVTYPE</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFD0</td>
<td>EDPIDR4</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFD4-0xFDc</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
<td>RES0</td>
</tr>
<tr>
<td>0xFE0-0xFEc</td>
<td>EDPIDR0</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFE4</td>
<td>EDPIDR1</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFE8</td>
<td>EDPIDR2</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFEc</td>
<td>EDPIDR3</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFF0</td>
<td>EDCIDR0</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFF4</td>
<td>EDCIDR1</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFF8</td>
<td>EDCIDR2</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFFc</td>
<td>EDCIDR3</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
</tbody>
</table>

## Table K6-5 External debug interface access permissions, CoreSight registers (debug) if FEAT_DoPD is not implemented

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Conditions (priority left to right)</th>
<th>Default</th>
<th>SLK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF00</td>
<td>EDITCTRL</td>
<td>IMP</td>
<td>DEFINED</td>
<td>RO/WI</td>
<td></td>
</tr>
<tr>
<td>0xFB4-0xF8c</td>
<td>Reserved</td>
<td>Debug</td>
<td>-</td>
<td>RES0</td>
<td>-</td>
</tr>
<tr>
<td>0xFA0</td>
<td>DBGCLAIMSET_EL1</td>
<td>Core</td>
<td>Error Error Error</td>
<td>RW (SE)</td>
<td>RO</td>
</tr>
<tr>
<td>0xFA4</td>
<td>DBGCLAIMCLR_EL1</td>
<td>Core</td>
<td>Error Error Error</td>
<td>RW (SE)</td>
<td>RO</td>
</tr>
<tr>
<td>0xFA8</td>
<td>EDDEVAFF0</td>
<td>Debug</td>
<td>-</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFAC</td>
<td>EDDEVAFF1</td>
<td>Debug</td>
<td>-</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFB0</td>
<td>EDLAR</td>
<td>Debug</td>
<td>-</td>
<td>WO (SE)</td>
<td>-</td>
</tr>
</tbody>
</table>
### Table K6-5 External debug interface access permissions, CoreSight registers (debug) if FEAT_DoPD is not implemented (continued)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Conditions (priority left to right)</th>
<th>Default</th>
<th>SLK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFB4</td>
<td>EDLSR</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFB8</td>
<td>DBGAUTHSTATUS_EL1</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFBC</td>
<td>EDDEVARCH</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFC0</td>
<td>EDDEVVID2</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFC4</td>
<td>EDDEVVID1</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFC8</td>
<td>EDDEVID</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFFC</td>
<td>EDDEVTYPE</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFD0</td>
<td>EDPIDR4</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFD4-0xFD8</td>
<td>Reserved</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RES0</td>
<td>-</td>
</tr>
<tr>
<td>0xFE0-0xFE3</td>
<td>EDPIDR0</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFE4</td>
<td>EDPIDR1</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFE8</td>
<td>EDPIDR2</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFE8</td>
<td>EDPIDR3</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFF0</td>
<td>EDCIDR0</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFF4</td>
<td>EDCIDR1</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFF8</td>
<td>EDCIDR2</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFFF</td>
<td>EDCIDR3</td>
<td>Debug</td>
<td>-   -   -</td>
<td>RO</td>
<td>-</td>
</tr>
</tbody>
</table>

### Table K6-6 External debug interface access permissions, CoreSight registers (CTI)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Conditions (priority left to right)</th>
<th>Default</th>
<th>SLK</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFB0</td>
<td>CTITITCTRL</td>
<td>IMP</td>
<td>IMPLEMENTATION DEFINED</td>
<td>IMP DEF</td>
<td>RO/WI</td>
<td></td>
</tr>
<tr>
<td>0xFB4-0xF83</td>
<td>Reserved</td>
<td>Debug</td>
<td>-</td>
<td>RES0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFA0</td>
<td>CTICLAIMSET</td>
<td>Debug</td>
<td>-</td>
<td>RW (SE)</td>
<td>RO</td>
<td></td>
</tr>
<tr>
<td>0xFA4</td>
<td>CTICLAIMCLR</td>
<td>Debug</td>
<td>-</td>
<td>RW (SE)</td>
<td>RO</td>
<td></td>
</tr>
<tr>
<td>0xFA8</td>
<td>CTIDEVAFF0</td>
<td>Debug</td>
<td>-</td>
<td>RO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFC0</td>
<td>CTIDEVAFF1</td>
<td>Debug</td>
<td>-</td>
<td>RO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFB0</td>
<td>CTILAR</td>
<td>Debug</td>
<td>-</td>
<td>WO (SE)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table K6-6 External debug interface access permissions, CoreSight registers (CTI) (continued)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Conditions (priority left to right)</th>
<th>Default</th>
<th>SLK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFB4</td>
<td>CTILSR</td>
<td>Debug</td>
<td>- - -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFB8</td>
<td>CTIAUTHSTATUS</td>
<td>Debug</td>
<td>- - -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFC6</td>
<td>CTIDEVARCH</td>
<td>Debug</td>
<td>- - -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFC0</td>
<td>CTIDEVID2</td>
<td>Debug</td>
<td>- - -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFC4</td>
<td>CTIDEVID1</td>
<td>Debug</td>
<td>- - -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFC8</td>
<td>CTIDEVID</td>
<td>Debug</td>
<td>- - -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFD0</td>
<td>CTIDEVTYPE</td>
<td>Debug</td>
<td>- - -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFD0-0xFD4</td>
<td>Reserved</td>
<td>Debug</td>
<td>- - -</td>
<td>RES0</td>
<td>-</td>
</tr>
<tr>
<td>0xFE0</td>
<td>CTIPIDR0</td>
<td>Debug</td>
<td>- - -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFE4</td>
<td>CTIPIDR1</td>
<td>Debug</td>
<td>- - -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFE8</td>
<td>CTIPIDR2</td>
<td>Debug</td>
<td>- - -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFE8</td>
<td>CTIPIDR3</td>
<td>Debug</td>
<td>- - -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFFF0</td>
<td>CTICIDR0</td>
<td>Debug</td>
<td>- - -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFFF4</td>
<td>CTICIDR1</td>
<td>Debug</td>
<td>- - -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFFF8</td>
<td>CTICIDR2</td>
<td>Debug</td>
<td>- - -</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFFC</td>
<td>CTICIDR3</td>
<td>Debug</td>
<td>- - -</td>
<td>RO</td>
<td>-</td>
</tr>
</tbody>
</table>

### Table K6-7 External debug interface access permissions, CoreSight registers (PMU) if FEAT_DoPD is implemented

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Conditions (priority left to right)</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF00</td>
<td>PMITCTRL</td>
<td>IMP DEF</td>
<td>IMPLEMENTATION DEFINED</td>
<td>IMP DEF</td>
</tr>
<tr>
<td>0xF04-0xFA4</td>
<td>Reserved</td>
<td>- - -</td>
<td>RES0</td>
<td></td>
</tr>
<tr>
<td>0xFA8</td>
<td>PMDEVAFF0</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFC0</td>
<td>PMDEVAFF1</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFB0</td>
<td>PMLAR</td>
<td>Core</td>
<td>Error</td>
<td>WO (SE)</td>
</tr>
<tr>
<td>0xFB4</td>
<td>PMLSR</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFB8</td>
<td>PMAUTHSTATUS</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
</tbody>
</table>
Table K6-7 External debug interface access permissions, CoreSight registers (PMU) if FEAT_DoPD is implemented (continued)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Conditions (priority left to right)</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFBC</td>
<td>PMDEVARCH</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFC0-0xFC4</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
<td>RES0</td>
</tr>
<tr>
<td>0xFC8</td>
<td>PMDEVID+a</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFCC</td>
<td>PMDEVTYPE</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFD0</td>
<td>PMPIDR4</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFD4-0xFDc</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
<td>RES0</td>
</tr>
<tr>
<td>0xFE0</td>
<td>PMPIDR0</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFE4</td>
<td>PMPIDR1</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFE8</td>
<td>PMPIDR2</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFEC</td>
<td>PMPIDR3</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFF0</td>
<td>PMCIDR0</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFF4</td>
<td>PMCIDR1</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFF8</td>
<td>PMCIDR2</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
<tr>
<td>0xFFC</td>
<td>PMCIDR3</td>
<td>Core</td>
<td>Error</td>
<td>RO</td>
</tr>
</tbody>
</table>

a. Implemented from Armv8.2, or if FEAT_PCSRv8p2 is implemented. Otherwise this location is RES0.

Table K6-8 External debug interface access permissions, CoreSight registers (PMU) if FEAT_DoPD is not implemented

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Conditions (priority left to right)</th>
<th>Default</th>
<th>SLK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF00</td>
<td>PMITCTRL</td>
<td>IMP DEF</td>
<td>IMPLEMENTATION DEFINED</td>
<td>IMP DEF</td>
<td>RO/WI</td>
</tr>
<tr>
<td>0xF04-0xFA4</td>
<td>Reserved</td>
<td>Debug</td>
<td>-</td>
<td>RES0</td>
<td>-</td>
</tr>
<tr>
<td>0xFA8</td>
<td>PMDEVAFF0</td>
<td>Debug</td>
<td>-</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFAC</td>
<td>PMDEVAFF1</td>
<td>Debug</td>
<td>-</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFB0</td>
<td>PMLAR</td>
<td>Debug</td>
<td>-</td>
<td>WO (SE)</td>
<td>-</td>
</tr>
<tr>
<td>0xFB4</td>
<td>PMLSR</td>
<td>Debug</td>
<td>-</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFB8</td>
<td>PMAUTHSTATUS</td>
<td>Debug</td>
<td>-</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFBC</td>
<td>PMDEVARCH</td>
<td>Debug</td>
<td>-</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFC0-0xFC4</td>
<td>Reserved</td>
<td>Debug</td>
<td>-</td>
<td>RES0</td>
<td>-</td>
</tr>
</tbody>
</table>
Table K6-8 External debug interface access permissions, CoreSight registers (PMU) if FEAT_DoPD is not implemented (continued)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Domain</th>
<th>Conditions (priority left to right)</th>
<th>Default</th>
<th>SLK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFC8</td>
<td>PMDEVIDa</td>
<td>Debug</td>
<td>Off</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFCC</td>
<td>PMDEVTYPE</td>
<td>Debug</td>
<td>Off</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFD0</td>
<td>PMPIDR4</td>
<td>Debug</td>
<td>Off</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFD4-0xFD0</td>
<td>Reserved</td>
<td>Debug</td>
<td>Off</td>
<td>RES0</td>
<td>-</td>
</tr>
<tr>
<td>0xFE0</td>
<td>PMPIDR0</td>
<td>Debug</td>
<td>Off</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFE4</td>
<td>PMPIDR1</td>
<td>Debug</td>
<td>Off</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFE8</td>
<td>PMPIDR2</td>
<td>Debug</td>
<td>Off</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFE0</td>
<td>PMPIDR3</td>
<td>Debug</td>
<td>Off</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFF0</td>
<td>PMCIDR0</td>
<td>Debug</td>
<td>Off</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFF4</td>
<td>PMCIDR1</td>
<td>Debug</td>
<td>Off</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFF8</td>
<td>PMCIDR2</td>
<td>Debug</td>
<td>Off</td>
<td>RO</td>
<td>-</td>
</tr>
<tr>
<td>0xFFC</td>
<td>PMCIDR3</td>
<td>Debug</td>
<td>Off</td>
<td>RO</td>
<td>-</td>
</tr>
</tbody>
</table>

a. Implemented from Armv8.2, or if FEAT_PCSRv8p2 is implemented. Otherwise this location is RES0.

K6.4.3 Management register resets

Table K6-9 on page K6-11695 shows the management register resets. This table does not include:
- Read-only identification registers that have a fixed value from reset. These registers include those with the DEVAFFn, DEVARCH, DEVID[n], DEVTYPE, PIDRn, and CIDRn mnemonics.
- Registers that have the AUTHSTATUS mnemonic. This is a read-only status register that reflects the status outside of the reset domain of the register.
- Registers that have the LAR mnemonic. These are write-only registers that only have an effect on writes.

All other fields in the management registers are reset to an IMPLEMENTATION DEFINED value which can be UNKNOWN. The registers are in the reset domain specified in the table.

Table K6-9 on page K6-11695 shows a summary of the management register resets.

<table>
<thead>
<tr>
<th>Register</th>
<th>Reset domain</th>
<th>Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTITCTRL</td>
<td>IMPLEMENTATION DEFINED</td>
<td>IME</td>
<td>0</td>
<td>Integration mode enable</td>
</tr>
<tr>
<td>EDITCTRL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMITCTRL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
K6.4.4 About the Peripheral identification scheme

The Peripheral Identification scheme provides the standard information required by all components that conform to the Arm® Debug Interface Architecture Specification, ADIv5.0 to ADIv5.2, that implements the CoreSight identification scheme. They identify a peripheral in a particular namespace. For more information, see the Arm® CoreSight™ Architecture Specification.

Table K6-10 on page K6-11696 lists the Peripheral ID Registers that make up the Peripheral Identification scheme for each component.

| Table K6-10 Peripheral Identification Registers |

| Reference |

<table>
<thead>
<tr>
<th>Register offset</th>
<th>Description</th>
<th>External Debug</th>
<th>CTI</th>
<th>Performance Monitors</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFD0</td>
<td>Peripheral ID4</td>
<td>EDPIDR4</td>
<td>CTIPIDR4</td>
<td>PMPIDR4</td>
</tr>
<tr>
<td>0xFD4</td>
<td>Reserved for Peripheral ID5</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0xFD8</td>
<td>Reserved for Peripheral ID6</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0xFD0</td>
<td>Peripheral ID7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0xFE0</td>
<td>Peripheral ID0</td>
<td>EDPIDR0</td>
<td>CTIPIDR0</td>
<td>PMPIDR0</td>
</tr>
<tr>
<td>0xFE4</td>
<td>Peripheral ID1</td>
<td>EDPIDR1</td>
<td>CTIPIDR1</td>
<td>PMPIDR1</td>
</tr>
<tr>
<td>0xFE8</td>
<td>Peripheral ID2</td>
<td>EDPIDR2</td>
<td>CTIPIDR2</td>
<td>PMPIDR2</td>
</tr>
<tr>
<td>0xFEC</td>
<td>Peripheral ID3</td>
<td>EDPIDR3</td>
<td>CTIPIDR3</td>
<td>PMPIDR3</td>
</tr>
</tbody>
</table>

Figure K6-2 shows the register field allocation scheme for the Peripheral ID Registers.

---

### Figure K6-2 Peripheral ID register format

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | RES0 | Periopheral ID data |
```
Software can consider the eight Peripheral ID Registers as defining a single 64-bit Peripheral ID, as shown in Figure K6-3.

Figure K6-3  Mapping between Peripheral ID Registers and a 64-bit Peripheral ID Value

Figure K6-3 shows the fields in the 64-bit Peripheral ID value, and includes the field values for fields that:
• Have fixed values, including the bits that are reserved.
• Have fixed values in an implementation that is designed by Arm.

For more information about the fields and their values, see Table K6-11 on page K6-11697.

Table K6-11 Fields in the Peripheral Identification Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Description</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB count</td>
<td>4 bits</td>
<td>Log2 of the number of 4KB blocks occupied by the implementation.</td>
<td>EDPIDR4, CTIPIDR4, PMPIDR4</td>
</tr>
<tr>
<td>JEP106 code</td>
<td>4+7 bits</td>
<td>Identifies the designer of the implementation. This value consists of:</td>
<td>EDPIDR1, EDPIDR2, EDPIDR4, CTIPIDR1, CTIPIDR2, CTIPIDR4, PMPIDR1, PMPIDR2, PMPIDR4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A 4-bit continuation code, also described as the bank number.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A 7-bit identification code.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For implementations designed by Arm, the continuation code is 0x4, indicating</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bank 5, and the identity code is 0x3B.</td>
<td></td>
</tr>
<tr>
<td>RevAnd</td>
<td>4 bits</td>
<td>Manufacturing revision number. Indicates a late modification to the</td>
<td>EDPIDR3, CTIPIDR3, PMPIDR3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>implementation, usually as a result of an <em>Engineering Change Order</em> (ECO).</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field starts at 0x0 and is incremented by the integrated circuit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>manufacturer on metal fixes.</td>
<td></td>
</tr>
<tr>
<td>Customer modified</td>
<td>4 bits</td>
<td>Indicates an endorsed modification to the implementation.</td>
<td>EDPIDR3, CTIPIDR3, PMPIDR3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the system designer cannot modify the implementation supplied by the</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>implementation designer, then this field is RES0.</td>
<td></td>
</tr>
</tbody>
</table>

For implementations designed by Arm, the continuation code is 0x4, indicating bank 5, and the identity code is 0x3B.
A component is identified uniquely by the combination of the following fields:

- JEP106 continuation code.
- JEP106 identity code.
- Part number.
- Revision.
- Customer Modified.
- RevAnd.

For components with a Component class of 0x9, Debug component, indicated by the Component Identification Registers, multiple components can have the same Part number, provided each component has a different CoreSight Device type. However, Arm strongly recommends that each device has a unique Part number. For more information:

- About the Component Identification Registers, see About the Component Identification scheme on page K6-11699.
- About the CoreSight Device type, see EDDEVTYPE, CTIDEVTYPE, or PMDEVTYPE.
- About CoreSight components and their identification, see the Arm® Debug Interface Architecture Specification.

### Allocating revisions and part numbers

Within the Peripheral Identification registers, the allocation of major and minor revisions, part numbers, and customer-modified fields is IMPLEMENTATION DEFINED, with the following set of restrictions so that:

- The REVISION field must increase monotonically with revisions.
  
  **Note**
  
  Arm recommends that the REVISION field is updated for each update to the RTL, regardless of whether this is a major or minor update.

- The REVAND field should increase monotonically with revisions.
  
  **Note**
  
  Arm recommends that the REVAND field is used only for post-release changes. For example, those due to engineering change order (ECO) fixes related to the debug component of the processor.

- The PART field must have a degree of uniqueness:
  - Two component designs can have the same part number so long as they are sub-components of the same part and the programmers’ model for the part has the means to disambiguate sub-components.
  - Otherwise, two component designs must have unique part numbers.

The DEVARCH (if implemented) or DEVTYPE (otherwise) register provides the means to disambiguate sub-components of the Debug Architecture.
A ROM table has no DEVTYPE or DEVARCH register. However, if it is the only CLASS 0x1 component in a processor cluster, it can still be disambiguated.

Multiple instances of the same component design have the same part number.

**K6.4.5 About the Component Identification scheme**

The Component Identification Registers identify the processor as an Arm Debug Interface v5 component. For more information, see the *Arm® Debug Interface Architecture Specification* and the *Arm® CoreSight™ Architecture Specification*.

The Component Identification Registers occupy the last four words of the 4KB block of debug registers.

<table>
<thead>
<tr>
<th>Table K6-12 Component Identification Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register offset</td>
</tr>
<tr>
<td>0xFF0</td>
</tr>
<tr>
<td>0xFF0</td>
</tr>
<tr>
<td>0xFF0</td>
</tr>
<tr>
<td>0xFF0</td>
</tr>
</tbody>
</table>

Figure K6-5 shows the register field allocation scheme for the Component ID Registers.

Software can consider the eight Component ID Registers as defining a single 32-bit Component ID, as shown in Figure K6-6.

**Figure K6-5 Component ID Register format**

**Figure K6-6 Mapping between Component ID Registers and a 32-bit Component ID Value**
Recommended External Debug Interface
K6.4 Management registers and CoreSight compliance
Appendix K7

Additional Information for Implementations of the Generic Timer

This appendix gives additional information about implementations of the Generic Timer. It contains the following sections:

• Providing a complete set of features in a system level implementation on page K7-11702.
• Gray-count scheme for timer distribution scheme on page K7-11704.
Providing a complete set of features in a system level implementation

As an example system design, using memory-mapped Generic Timer components as described in Chapter 12 System Level Implementation of the Generic Timer, the feature set of a System registers counter and timer, in an implementation that includes Non-secure EL2 and EL3, can be implemented using the following set of timer frames:

- A CNTCTLBase control frame.
- The following CNTBaseNN timer frames:
  - **Frame 0** Accessible by Non-secure accesses, with second view and virtual capability. This provides the Non-secure EL1&0 timers.
  - **Frame 1** Accessible by Non-secure accesses, with no second view and no virtual capability. This provides the Non-secure EL2 timers.
  - **Frame 2** Accessible only by Secure accesses, with a second view but no virtual capability. This provides the Secure PL1&0 timers, meaning:
    - Compared to a PE where EL3 is using AArch32, it provides the only Secure state timer.
    - Compared to a PE where EL3 is using AArch64, it provides the Secure EL1&0 timer.
  - **Frame 3** Accessible only by Secure accesses, with no second view and no virtual capability. This provides the Secure EL3 timers.

**Note**
This frame is not required for a memory-mapped timer that provides only the feature set of a PE for which EL3 is using AArch32.

In this implementation, the full set of implemented frames, and accessibility as memory pages in the different translation regimes, is as follows:

**CNTCTLBase**
The control frame. This frame is accessible in both the Secure and Non-secure memory maps, and:

- In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is not accessible.

**CNTBase0**
The first view of the Non-secure EL1&0 timers. This frame is accessible only in the Non-secure memory map, and:

- In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is accessible only at EL1.

**CNPTE0Base0**
The second view of CNTBase0, meaning it is the EL0 view of the Non-secure EL1&0 timers. This frame is accessible only in the Non-secure memory map, and:

- In the Secure EL1&0 translation regime, the architecture permits this frame to be accessible at EL1, or at EL1 and EL0, but does not require either of these options.
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is accessible at EL1 and EL0.

**CNTBase1**
The first and only view of the Non-secure EL2 timers. This frame is accessible only in Non-secure memory map, and:

- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
  - In the Secure EL3 translation regime, this frame is accessible.
- When EL3 is using AArch32, in the Secure PL1&0 translation regime, this frame is accessible only at PL1 (EL3).
- In the Non-secure EL2 translation regime, this frame is accessible.
• In the Non-secure EL1&0 translation regime, this frame is not accessible.

CNTBase2

The first view of the Secure EL1&0, or PL1&0 timers.

--- Note ---

In AArch64 state, these timers are always called the Secure EL1&0 timers. In AArch32 state they are usually called the Secure PL1&0 timers because, in AArch32 Secure state, whether some of the PE modes map to EL1 or to EL3 depends on whether EL3 is using AArch64 or is using AArch32, see Security state, Exception levels, and AArch32 execution privilege on page G1-8908.

This frame is accessible only in the Secure memory map, and:

• When EL3 is using AArch64:
  — In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
  — In the Secure EL3 translation regime, this frame is accessible.

• When EL3 is using AArch32, in the Secure PL1&0 translation regime, this frame is accessible only at PL1 (EL3).

• Because the frame is in Secure memory, it is not accessible in any Non-secure translation regime.

CNTEL0Base2

The second view of CNTBase2, meaning it is the EL0 view of the Secure EL1&0, or PL1&0, timers.

--- Note ---

See the Note in the description of the CNTBase2 frame for more information about the naming of these timers.

This frame is accessible only in the Secure memory map, and:

• When EL3 is using AArch64:
  — In the Secure EL1&0 translation regime, this frame is accessible at EL1 and EL0.
  — In the Secure EL3 translation regime, this frame is accessible.

• When EL3 is using AArch32, in the Secure PL1&0 translation regime, this frame is accessible at PL1 (EL3) and EL0.

• Because the frame is in Secure memory, it is not accessible in any Non-secure translation regime.

CNTBase3

The first and only view of the EL3 timers. This frame is accessible only in the Secure memory map, and:

• When EL3 is using AArch64:
  — In the Secure EL1&0 translation regime, this frame is not accessible.
  — In the Secure EL3 translation regime, this frame is accessible.

• When EL3 is using AArch32, this frame is not accessible.

• Because the frame is in Secure memory, it is not accessible in any Non-secure translation regime.
K7.2 Gray-count scheme for timer distribution scheme

The distribution of the Counter value using a Gray-code provides a relatively simple mechanism to avoid any danger of the count being sampled with an intermediate value even if the clocking is asynchronous. It has a further advantage that the distribution is relatively low power, since only one bit changes on the main distribution wires for each clock tick.

A suitable Gray-coding scheme can be achieved with the following logic:

\[
\begin{align*}
\text{Gray}[N] &= \text{Count}[N] \\
\text{Gray}[i] &= (\text{XOR(Gray}[N:i+1])) \text{ XOR Count}[i] \text{ for } N-1 \geq i \geq 0 \\
\text{Count}[i] &= \text{XOR(Gray}[N:i]) \text{ for } N \geq i \geq 0
\end{align*}
\]

This is for an \(N+1\) bit counter, where Count is a conventional binary count value, and Gray is the corresponding Gray count value.

--- Note ---

This scheme has the advantage of being relatively simple to switch, in either direction, between operating with low-frequency and low-precision, and operating with high-frequency and high-precision. To achieve this, the ratio of the frequencies must be \(2^n\), where \(n\) is an integer. A switch-over can occur only on the \(2n+1\) boundary to avoid losing the Gray-coding property on a switch-over.
Appendix K8

Legacy Instruction Syntax for AArch32 Instruction Sets

This appendix describes the legacy instruction syntax in the Arm instruction sets, and their Unified Assembler Language (UAL) equivalents. It contains the following section:

• Legacy Instruction Syntax on page K8-11706.
K8.1 Legacy Instruction Syntax

Early versions of the Arm Architecture defined an assembly language for A32 (ARM) instructions, and a separate assembly language for T32 (Thumb) instructions. UAL is based on the A32 assembly language, with some changes to the instruction syntax. The appendix describes those changes. The pre-UAL mnemonics are compatible with UAL, and might be supported by an assembler.

The original T32 assembly language is not compatible with UAL, and is not described in the manual.

K8.1.1 Pre-UAL instruction syntax for the A32 base instructions

Table K8-1 on page K8-11706 lists the syntax for the A32 base instructions that have changed after UAL was introduced.

<table>
<thead>
<tr>
<th>Pre-UAL syntax</th>
<th>UAL equivalent</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC&lt;&lt;S</td>
<td>ADCS&lt;&lt;</td>
<td>ADC, ADCS (immediate) on page F5-7443, ADC, ADCS (register) on page F5-7446, ADC, ADCS (register-shifted register) on page F5-7450</td>
</tr>
<tr>
<td>ADD&lt;&lt;S</td>
<td>ADDS&lt;&lt;</td>
<td>ADD, ADDS (immediate) on page F5-7452, ADD, ADDS (register) on page F5-7456, ADD, ADDS (register-shifted register) on page F5-7460, ADD, ADDS (SP plus immediate) on page F5-7462, ADD, ADDS (SP plus register) on page F5-7465</td>
</tr>
<tr>
<td>AND&lt;&lt;S</td>
<td>ANDS&lt;&lt;</td>
<td>AND, ANDS (immediate) on page F5-7474, AND, ANDS (register) on page F5-7477, AND, ANDS (register-shifted register) on page F5-7481</td>
</tr>
<tr>
<td>BIC&lt;&lt;S</td>
<td>BICS&lt;&lt;</td>
<td>BIC, BICS (immediate) on page F5-7498, BIC, BICS (register) on page F5-7501, BIC, BICS (register-shifted register) on page F5-7505</td>
</tr>
<tr>
<td>EOR&lt;&lt;S</td>
<td>EORS&lt;&lt;</td>
<td>EOR, EORS (immediate) on page F5-7561, EOR, EORS (register) on page F5-7564, EOR, EORS (register-shifted register) on page F5-7568</td>
</tr>
<tr>
<td>LDC&lt;&lt;L</td>
<td>LDCL&lt;&lt;</td>
<td>LDC (immediate) on page F5-7596, LDC (literal) on page F5-7598</td>
</tr>
<tr>
<td>LDM&lt;&lt;IA, LDM&lt;&lt;FD</td>
<td>LDM&lt;&lt;</td>
<td>LDM, LDMIA, LDMFD on page F5-7600</td>
</tr>
<tr>
<td>LDM&lt;&lt;DA, LDM&lt;&lt;FA</td>
<td>LDMDA&lt;&lt;</td>
<td>LDMDA, LDMEA on page F5-7608</td>
</tr>
<tr>
<td>LDM&lt;&lt;DB, LDM&lt;&lt;EA</td>
<td>LDMDB&lt;&lt;</td>
<td>LDMDB, LDMEA on page F5-7610</td>
</tr>
<tr>
<td>LDM&lt;&lt;IB, LDM&lt;&lt;ED</td>
<td>LDMIB&lt;&lt;</td>
<td>LDMIB, LDME on page F5-7613</td>
</tr>
<tr>
<td>LDR&lt;&lt;B</td>
<td>LDRB&lt;&lt;</td>
<td>LDRB (immediate) on page F5-7625, LDRB (literal) on page F5-7629, LDRB (register) on page F5-7631</td>
</tr>
<tr>
<td>LDR&lt;&lt;BT</td>
<td>LDRBT&lt;&lt;</td>
<td>LDRBT on page F5-7634</td>
</tr>
<tr>
<td>LDR&lt;&lt;D</td>
<td>LDRD&lt;&lt;</td>
<td>LDRD (immediate) on page F5-7637, LDRD (literal) on page F5-7640, LDRD (register) on page F5-7643</td>
</tr>
</tbody>
</table>
Table K8-1 Pre-UAL instruction syntax for the A32 base instructions (continued)

<table>
<thead>
<tr>
<th>Pre-UAL syntax</th>
<th>UAL equivalent</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR&lt;&lt;H</td>
<td>LDRH&lt;&lt;</td>
<td>LDRH (immediate) on page F5-7654, LDRH (literal) on page F5-7658, LDRH (register) on page F5-7660</td>
</tr>
<tr>
<td>LDR&lt;&lt;SB</td>
<td>LDRSB&lt;&lt;</td>
<td>LDRSB (immediate) on page F5-7666, LDRSB (literal) on page F5-7669, LDRSB (register) on page F5-7671</td>
</tr>
<tr>
<td>LDR&lt;&lt;SH</td>
<td>LDRSH&lt;&lt;</td>
<td>LDRSH (immediate) on page F5-7677, LDRSH (literal) on page F5-7680, LDRSH (register) on page F5-7682</td>
</tr>
<tr>
<td>LDR&lt;&lt;T</td>
<td>LDRT&lt;&lt;</td>
<td>LDRT on page F5-7688</td>
</tr>
<tr>
<td>MLA&lt;&lt;S</td>
<td>MLAS&lt;&lt;</td>
<td>MLA, MLAS on page F5-7711</td>
</tr>
<tr>
<td>LSLS &lt;Rd&gt;, &lt;Rn&gt;, #0</td>
<td>MOV&lt;rd&gt;, &lt;rm&gt;</td>
<td>MOV, MOVS (immediate) on page F5-7715, MOV, MOVS (register) on page F5-7719</td>
</tr>
<tr>
<td>MOV&lt;&lt;S</td>
<td>MOVS&lt;&lt;</td>
<td>MOV, MOVS (immediate) on page F5-7715, MOV, MOVS (register) on page F5-7719</td>
</tr>
<tr>
<td>MUL&lt;&lt;S</td>
<td>MULS&lt;&lt;</td>
<td>MUL, MULS on page F5-7749</td>
</tr>
<tr>
<td>MVN&lt;&lt;S</td>
<td>MVNS&lt;&lt;</td>
<td>MVN, MVNS (immediate) on page F5-7751, MVN, MVNS (register) on page F5-7753, MVN, MVNS (register-shifted register) on page F5-7756</td>
</tr>
<tr>
<td>ORR&lt;&lt;S</td>
<td>ORRS&lt;&lt;</td>
<td>ORR, ORRS (immediate) on page F5-7764, ORR, ORRS (register) on page F5-7767, ORR, ORRS (register-shifted register) on page F5-7771</td>
</tr>
<tr>
<td>QADDSUBX</td>
<td>QASX</td>
<td>QASX on page F5-7808</td>
</tr>
<tr>
<td>QSUBADDX</td>
<td>QSAX</td>
<td>QSAX on page F5-7814</td>
</tr>
<tr>
<td>RSB&lt;&lt;S</td>
<td>RSBS&lt;&lt;</td>
<td>RSB, RSBS (immediate) on page F5-7845, RSB, RSBS (register) on page F5-7848, RSB, RSBS (register-shifted register) on page F5-7851</td>
</tr>
<tr>
<td>RSC&lt;&lt;S</td>
<td>RSCS&lt;&lt;</td>
<td>RSC, RSCS (immediate) on page F5-7853, RSC, RSCS (register) on page F5-7855, RSC, RSCS (register-shifted register) on page F5-7857</td>
</tr>
<tr>
<td>SADDSUBX</td>
<td>SASX</td>
<td>SASX on page F5-7863</td>
</tr>
<tr>
<td>SBC&lt;&lt;S</td>
<td>SBSCS&lt;&lt;</td>
<td>SBC, SBSCS (immediate) on page F5-7867, SBC, SBSCS (register), SBC, SBSCS (register-shifted register) on page F5-7874</td>
</tr>
<tr>
<td>SHADDSUBX</td>
<td>SHASX</td>
<td>SHASX on page F5-7892</td>
</tr>
<tr>
<td>SHSUBADDX</td>
<td>SHSAX</td>
<td>SHSAX on page F5-7894</td>
</tr>
<tr>
<td>SM1&lt;&lt;</td>
<td>SMC&lt;&lt;</td>
<td>SMC on page F5-7900</td>
</tr>
<tr>
<td>SMLAL&lt;&lt;S</td>
<td>SMLALS&lt;&lt;</td>
<td>SMLAL, SMLALS on page F5-7906</td>
</tr>
<tr>
<td>SMULL&lt;&lt;S</td>
<td>SMULLS&lt;&lt;</td>
<td>SMULL, SMULLS on page F5-7930</td>
</tr>
</tbody>
</table>
Table K8-1 Pre-UAL instruction syntax for the A32 base instructions (continued)

<table>
<thead>
<tr>
<th>Pre-UAL syntax</th>
<th>UAL equivalent</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSUBADDX&lt;&lt;</td>
<td>SSAX&lt;&lt;</td>
<td>SSAX on page F5-7944</td>
</tr>
<tr>
<td>STC&lt;&lt;L</td>
<td>STCL&lt;&lt;</td>
<td>STC on page F5-7952</td>
</tr>
<tr>
<td>STM&lt;&lt;EA, STM&lt;&lt;IA</td>
<td>STM&lt;&lt;</td>
<td>STM, STMIA, STMIA on page F5-7972</td>
</tr>
<tr>
<td>STM&lt;&lt;DA, STM&lt;&lt;ED</td>
<td>STMDA&lt;&lt;</td>
<td>STMDA, STMED on page F5-7978</td>
</tr>
<tr>
<td>STM&lt;&lt;DB, STM&lt;&lt;FD</td>
<td>STMDB&lt;&lt;</td>
<td>STMDB, STMFD on page F5-7980</td>
</tr>
<tr>
<td>STM&lt;&lt;IB, STM&lt;&lt;FA</td>
<td>STMB&lt;&lt;</td>
<td>STMB, STMFA on page F5-7983</td>
</tr>
<tr>
<td>STR&lt;&lt;B</td>
<td>STRB&lt;&lt;</td>
<td>STRB (immediate) on page F5-7993, STRB (register) on page F5-7997</td>
</tr>
<tr>
<td>STR&lt;&lt;BT</td>
<td>STRBT&lt;&lt;</td>
<td>STRBT on page F5-8000</td>
</tr>
<tr>
<td>STR&lt;&lt;D</td>
<td>STRD&lt;&lt;</td>
<td>STRD (immediate) on page F5-8004, STRD (register) on page F5-8008</td>
</tr>
<tr>
<td>STR&lt;&lt;H</td>
<td>STRH&lt;&lt;</td>
<td>STRH (immediate) on page F5-8022, STRH (register) on page F5-8026</td>
</tr>
<tr>
<td>STR&lt;&lt;T</td>
<td>STRT&lt;&lt;</td>
<td>STRT on page F5-8033</td>
</tr>
<tr>
<td>SUB&lt;&lt;S</td>
<td>SUBS&lt;&lt;</td>
<td>SUB, SUBS (immediate) on page F5-8039, SUB, SUBS (register) on page F5-8043, SUB, SUBS (register-shifted register) on page F5-8047, SUB, SUBS (SP minus immediate) on page F5-8049, SUB, SUBS (SP minus register) on page F5-8052</td>
</tr>
<tr>
<td>SWI</td>
<td>SVC</td>
<td>SVC on page F5-8055</td>
</tr>
<tr>
<td>UADDSUBX</td>
<td>UASX</td>
<td>UASX on page F5-8090</td>
</tr>
<tr>
<td>UHADDSUBX</td>
<td>UHASX</td>
<td>UHASX on page F5-8102</td>
</tr>
<tr>
<td>UHSUBADDX</td>
<td>UHSAX</td>
<td>UHSAX on page F5-8104</td>
</tr>
<tr>
<td>UMLAL&lt;&lt;S</td>
<td>UMLALS&lt;&lt;</td>
<td>UMLAL, UMLALS on page F5-8112</td>
</tr>
<tr>
<td>UMULL&lt;&lt;S</td>
<td>UMULLS&lt;&lt;</td>
<td>UMULL, UMULLS on page F5-8114</td>
</tr>
<tr>
<td>UQADDSUBX</td>
<td>UQASX</td>
<td>UQASX on page F5-8120</td>
</tr>
<tr>
<td>UQSUBADDX</td>
<td>UQASX</td>
<td>UQASX on page F5-8122</td>
</tr>
<tr>
<td>USUBADDX</td>
<td>USAX</td>
<td>USAX on page F5-8136</td>
</tr>
<tr>
<td>UEXT8</td>
<td>UXTB</td>
<td>UXTB on page F5-8148</td>
</tr>
<tr>
<td>UEXT16</td>
<td>UXTH</td>
<td>UXTH on page F5-8152</td>
</tr>
</tbody>
</table>
### K8.1.2  Pre-UAL instruction syntax for the A32 floating-point instructions

Table K8-2 on page K8-11709 lists the syntax for A32 floating-point instructions that have changed after UAL was introduced.

**Table K8-2 Pre-UAL instruction syntax for A32 floating-point instructions**

<table>
<thead>
<tr>
<th>Pre-UAL syntax</th>
<th>UAL equivalent</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FABSD</td>
<td>VABS.F64</td>
<td><em>VABS on page F6-8213</em></td>
</tr>
<tr>
<td>FABSS</td>
<td>VABS.F32</td>
<td></td>
</tr>
<tr>
<td>FADD</td>
<td>VADD.F64</td>
<td><em>VADD (floating-point) on page F6-8227</em></td>
</tr>
<tr>
<td>FADDS</td>
<td>VADD.F32</td>
<td></td>
</tr>
<tr>
<td>FCMPEZD</td>
<td>VCMPE.F64</td>
<td><em>VCMPE on page F6-8304</em></td>
</tr>
<tr>
<td>FCMPEZS</td>
<td>VCMPE.F32</td>
<td></td>
</tr>
<tr>
<td>FCMPS</td>
<td>VCMPS</td>
<td></td>
</tr>
<tr>
<td>FCMPEZD</td>
<td>VCMPS</td>
<td></td>
</tr>
<tr>
<td>FCMPS</td>
<td>VCMPS</td>
<td></td>
</tr>
<tr>
<td>FCONSTD &lt;Dd&gt;, #&lt;imm8&gt;</td>
<td>VMOV.F64 &lt;Dd&gt;, #&lt;fpimm&gt;</td>
<td><em>VMOV (immediate) on page F6-8539</em></td>
</tr>
<tr>
<td>FCONSTS &lt;Sd&gt;, #&lt;imm8&gt;</td>
<td>VMOV.F32 &lt;Sd&gt;, #&lt;fpimm&gt;</td>
<td>For more information, see <em>FCONST</em> on page K8-11712.</td>
</tr>
<tr>
<td>FCPYD</td>
<td>VMOV.F64</td>
<td><em>VMOV (register) on page F6-8546</em></td>
</tr>
<tr>
<td>FCPYS</td>
<td>VMOV.F32</td>
<td></td>
</tr>
<tr>
<td>FCVTDS</td>
<td>VCVT.F64.F32</td>
<td>*VCVT (between double-precision and single-precision) on page F6-8312</td>
</tr>
<tr>
<td>FCVTSD</td>
<td>VCVT.F32.F64</td>
<td></td>
</tr>
<tr>
<td>FDIVD</td>
<td>VDIV.F64</td>
<td><em>VDIV on page F6-8363</em></td>
</tr>
<tr>
<td>FDIVS</td>
<td>VDIV.F32</td>
<td></td>
</tr>
<tr>
<td>FLDSD, FLDMIAD</td>
<td>VLDM.F64</td>
<td><em>VLDM, VLDMD, VLDMIA on page F6-8476</em></td>
</tr>
<tr>
<td>FLDMS</td>
<td>VLDM.F32</td>
<td></td>
</tr>
<tr>
<td>FLDS</td>
<td>VLDR.F32</td>
<td><em>VLDR (immediate) on page F6-8481</em></td>
</tr>
<tr>
<td>FMACD</td>
<td>VMLA.F64</td>
<td><em>VMLA (floating-point) on page F6-8507</em></td>
</tr>
<tr>
<td>FMACS</td>
<td>VMLA.F32</td>
<td></td>
</tr>
<tr>
<td>FMDCR &lt;Dd&gt;, &lt;Rt&gt;</td>
<td>VMOV &lt;Dd&gt;[1], &lt;Rt&gt;</td>
<td><em>VMOV (general-purpose register to scalar) on page F6-8550</em></td>
</tr>
<tr>
<td>FMDDR</td>
<td>VMOV &lt;Dd&gt;[0], &lt;Rt&gt;</td>
<td></td>
</tr>
<tr>
<td>FMDCRR</td>
<td>VMOV</td>
<td><em>VMOV (between two general-purpose registers and a doubleword floating-point register) on page F6-8535</em></td>
</tr>
<tr>
<td>FMROH &lt;Rt&gt;, &lt;Dd&gt;</td>
<td>VMOV &lt;Rt&gt;, &lt;Dd&gt;[1]</td>
<td><em>VMOV (scalar to general-purpose register) on page F6-8554</em></td>
</tr>
<tr>
<td>FMRDL</td>
<td>VMOV &lt;Rt&gt;, &lt;Dd&gt;[0]</td>
<td></td>
</tr>
<tr>
<td>Pre-UAL syntax</td>
<td>UAL equivalent</td>
<td>See</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------</td>
<td>-----</td>
</tr>
<tr>
<td>FMRRD</td>
<td>VMOV</td>
<td>VMOV (between two general-purpose registers and a doubleword floating-point register) on page F6-8535</td>
</tr>
<tr>
<td>FMRRS</td>
<td>VMOV</td>
<td>VMOV (between two general-purpose registers and two single-precision registers) on page F6-8556</td>
</tr>
<tr>
<td>FMRRS</td>
<td>VMOV</td>
<td>VMOV (between general-purpose register and single-precision) on page F6-8552</td>
</tr>
<tr>
<td>FMRRS</td>
<td>VMRS</td>
<td>VMRS on page F6-8565</td>
</tr>
<tr>
<td>FMSCD</td>
<td>VNNLS.F64</td>
<td>VNMLS on page F6-8599</td>
</tr>
<tr>
<td>FMSCS</td>
<td>VNNLS.F32</td>
<td></td>
</tr>
<tr>
<td>FMSR</td>
<td>VMOV</td>
<td>VMOV (between general-purpose register and single-precision) on page F6-8552</td>
</tr>
<tr>
<td>FMSRR</td>
<td>VMOV</td>
<td>VMOV (between two general-purpose registers and two single-precision registers) on page F6-8556</td>
</tr>
<tr>
<td>FSMSTAT</td>
<td>VMRS APSR_nzcv, FPSCR</td>
<td>VMRS on page F6-8565</td>
</tr>
<tr>
<td>FMULD</td>
<td>VMUL.F64</td>
<td>VMUL (floating-point) on page F6-8571</td>
</tr>
<tr>
<td>FMULS</td>
<td>VMUL.F32</td>
<td></td>
</tr>
<tr>
<td>FMRX</td>
<td>VMRS</td>
<td>VMRS on page F6-8565</td>
</tr>
<tr>
<td>FNEGD</td>
<td>VNEG.F64</td>
<td>VNEG on page F6-8592</td>
</tr>
<tr>
<td>FNEGS</td>
<td>VNEG.F32</td>
<td></td>
</tr>
<tr>
<td>FNMACD</td>
<td>VMLS.F64</td>
<td>VNMLS on page F6-8599</td>
</tr>
<tr>
<td>FNMACS</td>
<td>VMLS.F32</td>
<td></td>
</tr>
<tr>
<td>FMSCD</td>
<td>VNNLA.F64</td>
<td>VNMLA on page F6-8596</td>
</tr>
<tr>
<td>FMSCS</td>
<td>VNNLA.F32</td>
<td></td>
</tr>
<tr>
<td>FMULD</td>
<td>VNMUL.F64</td>
<td>VNMUL on page F6-8602</td>
</tr>
<tr>
<td>FMULS</td>
<td>VNMUL.F32</td>
<td></td>
</tr>
<tr>
<td>FSHTOD</td>
<td>VCVT.F64.S16</td>
<td>VCVT (between floating-point and fixed-point, floating-point) on page F6-8329</td>
</tr>
<tr>
<td>FSHTOS</td>
<td>VCVT.F32.S16</td>
<td></td>
</tr>
<tr>
<td>FSITOD</td>
<td>VCVT.F64.S32</td>
<td>VCVT (between floating-point and integer, Advanced SIMD) on page F6-8316, VCVTR on page F6-8354</td>
</tr>
<tr>
<td>FSITOS</td>
<td>VCVT.F32.S32</td>
<td>VCVT (between floating-point and fixed-point, floating-point) on page F6-8329</td>
</tr>
<tr>
<td>FSQRTD</td>
<td>VSQRT.F64</td>
<td>VSQRT on page F6-8790</td>
</tr>
<tr>
<td>FSQRTS</td>
<td>VSQRT.F32</td>
<td></td>
</tr>
<tr>
<td>FSTD</td>
<td>VSTR</td>
<td>VSTR on page F6-8847</td>
</tr>
</tbody>
</table>
Table K8-2 Pre-UAL instruction syntax for A32 floating-point instructions (continued)

<table>
<thead>
<tr>
<th>Pre-UAL syntax</th>
<th>UAL equivalent</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSTMD, FSTMIA</td>
<td>VSTM.F64</td>
<td>VSTM, VSTMDB, VSTMIA on page F6-8842</td>
</tr>
<tr>
<td>FSTMS</td>
<td>VSTM.F32</td>
<td></td>
</tr>
<tr>
<td>FSTS</td>
<td>VSTR</td>
<td>VSTR on page F6-8847</td>
</tr>
<tr>
<td>FSUBD</td>
<td>VSUB.F64</td>
<td>VSUB (floating-point) on page F6-8850</td>
</tr>
<tr>
<td>FSUBS</td>
<td>VSUB.F32</td>
<td></td>
</tr>
<tr>
<td>FTOSHD</td>
<td>VCVT.S16.F64</td>
<td></td>
</tr>
<tr>
<td>FTOSHS</td>
<td>VCVT.S16.F23</td>
<td></td>
</tr>
<tr>
<td>FTOSID</td>
<td>VCVT.S32.F64</td>
<td></td>
</tr>
<tr>
<td>FTOSIS</td>
<td>VCVT.S32.F32</td>
<td></td>
</tr>
<tr>
<td>FTOSIZD</td>
<td>VCVTR.S32.F64</td>
<td>VCVTR on page F6-8354</td>
</tr>
<tr>
<td>FTOSIZS</td>
<td>VCVTR.S32.F32</td>
<td></td>
</tr>
<tr>
<td>FTOSLD</td>
<td>VCVT.S32.F64</td>
<td>VCVT (between floating-point and fixed-point, floating-point) on page F6-8329</td>
</tr>
<tr>
<td>FTOSLS</td>
<td>VCVT.S32.F32</td>
<td></td>
</tr>
<tr>
<td>FTOUHD</td>
<td>VCVT.U16.F64</td>
<td></td>
</tr>
<tr>
<td>FTOUHS</td>
<td>VCVT.U16.F32</td>
<td></td>
</tr>
<tr>
<td>FTOUID</td>
<td>VCVT.U32.F64</td>
<td>VCVT (between floating-point and integer, Advanced SIMD) on page F6-8316</td>
</tr>
<tr>
<td>FTOUIS</td>
<td>VCVT.U32.F32</td>
<td></td>
</tr>
<tr>
<td>FTOUIZD</td>
<td>VCVTR.U32.F64</td>
<td>VCVTR on page F6-8354</td>
</tr>
<tr>
<td>FTOUIZS</td>
<td>VCVTR.U32.F32</td>
<td></td>
</tr>
<tr>
<td>FTOULD</td>
<td>VCVT.U32.F64</td>
<td>VCVT (between floating-point and fixed-point, floating-point) on page F6-8329</td>
</tr>
<tr>
<td>FTOULS</td>
<td>VCVT.U32.F32</td>
<td></td>
</tr>
<tr>
<td>FUITOD,</td>
<td>VCVT.F64.U16</td>
<td></td>
</tr>
<tr>
<td>FUITOS</td>
<td>VCVT.F64.U16</td>
<td></td>
</tr>
<tr>
<td>FUITOD</td>
<td>VCVT.F64.U32</td>
<td>VCVT (between floating-point and integer, Advanced SIMD) on page F6-8316</td>
</tr>
<tr>
<td>FUITOS</td>
<td>VCVT.F32.U32</td>
<td></td>
</tr>
<tr>
<td>FULTOD</td>
<td>VCVT.F64.U32</td>
<td>VCVT (between floating-point and fixed-point, floating-point) on page F6-8329</td>
</tr>
<tr>
<td>FULTOS</td>
<td>VCVT.F32.U32</td>
<td></td>
</tr>
</tbody>
</table>
K8.1.3 FCONST

The syntax of FCONST is

FCONST<dest>{<c>} <Fd>, #<imm8>

where:

<dest> Specifies the destination data type. It must be one of:
S Single-precision floating-point.
D Double-precision floating-point.

<c> This is an optional field. It specifies the condition under which the instruction is executed. See Conditional execution on page F1-7229 for the range of available conditions and their encoding. If <c> is omitted, it defaults to always (AL).

<Fd> Specifies the destination register. It must be one of:
<6d> 64-bit name of the SIMD&FP destination register.
<5d> 32-bit name of the SMID&FP destination register.

<imm8> Specifies the immediate value used to generate the floating-point constant.

FCONSTD{<c>} <6d>, #<imm8> maps to VMOV.F64 <6d>, #<fpimm>
FCONSTS{<c>} <5d>, #<imm8> maps to VMOV.F32 <5d>, #<fpimm>
Appendix K9
Address Translation Examples

This appendix gives examples of address translations using the translation regimes described in Chapter D8 The AArch64 Virtual Memory System Architecture and Chapter G5 The AArch32 Virtual Memory System Architecture. It contains the following sections:

- AArch64 Address translation examples on page K9-11714.
- AArch32 Address translation examples on page K9-11726.

--- Note ---

This chapter gives examples of translation table lookups for the Armv8 address translation stages. It does not define any part of the address translation mechanism. If any information in this appendix appears to contradict the information in Chapter D8 The AArch64 Virtual Memory System Architecture or Chapter G5 The AArch32 Virtual Memory System Architecture then the information in Chapter D8 or Chapter G5 must be taken as the definition of the required behavior.
K9.1 AArch64 Address translation examples

The VMSAv8 address translation stages that are controlled by an Exception level that is using AArch64 are described in Relationhip between translation regimes and implemented Exception levels on page D8-5084. Address translation on page D8-5080 describes the VMSAv8-64 address translation scheme. This section gives examples of the use of that scheme, for common translation requirements.

System registers relevant to MMU operation on page D8-5086 specifies the relevant registers, including the TCR_ELx and TTBR_ELx, or TTBR_ELxs, for each stage of address translation.

For any stage of translation, a TCR_ELx.TnSZ field indicates the supported input address size. For a stage of address translation controlled from an Exception level using AArch64, the supported input address size is $2^{(64-TnSZ)}$.

This section describes:

• Performing the initial lookup, for an address for which the initial lookup is either:
  — At the highest lookup level used for the appropriate translation granule size.
  — Because of the concatenation of translation tables at the initial lookup level, one level down from the highest level used for the translation granule size.

These descriptions take account of the following cases:

— The IA size is smaller than the largest size for the translation level, see Input address size configuration on page D8-5090.
— For a stage 2 translation, translation tables are concatenated, to move the initial lookup level down by one level, see Concatenated translation tables on page D8-5094.

For examples of performing the initial lookup, see Examples of performing the initial lookup.

• The full translation flow for resolving a page of memory. These examples describe resolving the largest IA size supported by the initial lookup level. For these examples, see Full translation flows for VMSAv8-64 address translation on page K9-11720.

K9.1.1 Examples of performing the initial lookup

The address ranges used for the initial translation table lookup depend on the translation granule, as described in:

• Performing the initial lookup using the 4KB translation granule.
• Performing the initial lookup using the 16KB granule on page K9-11716.
• Performing the initial lookup using the 64KB translation granule on page K9-11718.

Performing the initial lookup using the 4KB translation granule

This subsection describes examples of the initial lookup when using the 4KB translation granule that Stage 2 address translation using the 4KB translation granule on page D8-5103 describes as starting at level 0 or at level 1. It includes those stage 2 translations where concatenation of translation tables is required for the lookup to start at level 1. This means that it gives specific examples of the mechanisms described in Translation process on page D8-5093.

Note

For stage 2 translations, the same principles apply to an initial lookup that Stage 2 address translation using the 4KB translation granule on page D8-5103 describes as starting at level 1. In this case, for some IA sizes concatenation of translation tables means the lookup can, instead, start at level 2.

The following subsections describe these examples of the initial lookup:

• Initial lookup at level 0, 4KB translation granule on page K9-11715.
• Initial lookup at level 1, 4KB translation granule on page K9-11715.

In all cases, for a stage 2 translation, the VTCR_EL2.SL0 field must indicate the required initial lookup level, and this level must be consistent with the value of the VTCR_EL2.T0SZ field, see Stage 2 address translation using the 4KB translation granule on page D8-5103.
Initial lookup at level 0, 4KB translation granule

This subsection describes initial lookups with an input address width of \((n+1)\) bits, meaning the input address is \(IA[n:0]\). As described in Translation using the 4KB granule on page D8-5100, a stage 1 or stage 2 initial lookup at level 0 is required when \(39 \leq n \leq 47\). For these lookups:

- \(TTBR\_ELx[47:(n-35)]\) specify the translation table base address.
- \(Bits[n:39]\) of the input address are \(bits[(n-36):3]\) of the descriptor offset in the translation table.

---

**Note**

This means that, when the input address width is less than 48 bits:

- The size of the translation table is reduced.
- More low-order bits of the \(TTBR\_ELx\) are required to specify the translation table base address.
- Fewer input address bit are used to specify the descriptor offset in the translation table.

For example, if the input address width is 46 bits:

- The translation table size is 1KB.
- \(TTBR\_ELx\) bits\(47:10\) specify the translation table base address.
- Input address bits\(45:39\) specify bits\(9:3\) of the descriptor offset.

Figure K9-1 shows this lookup.

---

Figure K9-1 Initial lookup for VMSAv8-64 using the 4KB granule, starting at level 0

Initial lookup at level 1, 4KB translation granule

This subsection describes initial lookups with an input address width of \((n+1)\) bits, meaning the input address is \(IA[n:0]\).

For a stage 1 or stage 2 initial lookup at level 1, without use of concatenated translation tables

As described in Translation using the 4KB granule on page D8-5100, this applies to \(IA[n:0]\), where \(30 \leq n \leq 38\). For these lookups:

- There is a single translation table at this level.
- \(TTBR\_ELx[47:(n-26)]\) specify the translation table base address.
- \(Bits[n:30]\) of the input address are \(bits[(n-27):3]\) of the descriptor offset in the translation table.
Address Translation Examples
K9.1 AArch64 Address translation examples

Figure K9-2 shows this lookup.

```
+------------+-----------------------------+-----------------------------+
| 47         | y+f                        | 0                           |
| RES0       |                            | 0                           |
+------------+-----------------------------+-----------------------------+
```

Input address

```
+---------------+-----------------------------+-----------------------------+-----------------------------+-----------------------------+-----------------------------+
| 48|47 |                             | 30 29                       | x|1 |                             | 0                           |
| RES0*       |                            | 0                           | 0                           | 0                           | 0                           |
| TTBR        |                            | 0                           | 0                           | 0                           | 0                           |
+---------------+-----------------------------+-----------------------------+-----------------------------+-----------------------------+-----------------------------+
```

Register-defined

Translation table base address [47:x]

Supported input address range is IA[y:0], 4 ≤ x ≤ 12, y = x + 26.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K9-2 Initial lookup for VMSAv8-64 using the 4KB granule, starting at level 1, without concatenation

For a stage 2 initial lookup at level 1, with concatenated translation tables

As described in Stage 2 address translation using the 4KB translation granule on page D8-5103, this applies to IA[n:0], where 39 ≤ n ≤ 42. For these lookups:

- There are $2^{(n-38)}$ concatenated translation tables at this level.
- These concatenated translation tables must be aligned to $2^{(n-38)} \times 4KB$. This means TTBR_ELx[47:(n-27):12] must be zero.
- TTBR_ELx[47:(n-26)] specify the base address of the block of concatenated translation tables.
- Bits[n:30] of the input address are bits[(n-27):3] of the descriptor offset from the base address of the block of concatenated translation tables.

Figure K9-3 shows this lookup.

```
+------------+-----------------------------+-----------------------------+
| 47         | y+1                        | 0                           |
| RES0       |                            | 0                           |
+------------+-----------------------------+-----------------------------+
```

Input address

```
+---------------+-----------------------------+-----------------------------+-----------------------------+-----------------------------+-----------------------------+-----------------------------+
| 48|47 |                             | 30 29                       | x|1 |                             | 12 11                      | 0                           |
| RES0*       |                            | 0                           | 0                           | 0                           | 0                           | 0                           |
| TTBR        |                            | 0                           | 0                           | 0                           | 0                           | 0                           |
+---------------+-----------------------------+-----------------------------+-----------------------------+-----------------------------+-----------------------------+-----------------------------+
```

Register-defined

Translation table base address [47:x]

Supported input address range is IPA[y:0], 4 ≤ x ≤ 16, y = x + 26. The field marked ‡ must be zero.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K9-3 Initial lookup for VMSAv8-64 using the 4KB granule, starting at level 1, with concatenation

Performing the initial lookup using the 16KB granule

This subsection describes examples of the initial lookup when using the 16KB translation granule that Stage 2 address translation using the 16KB translation granule on page D8-5109 describes as starting at level 0 or at level 1. It includes those stage 2 translations where concatenation of translation tables is required for the lookup to start at level 1. This means that it gives specific examples of the mechanisms described in Translation process on page D8-5093.
Note

For stage 2 translations, the same principles apply to an initial lookup that Stage 2 address translation using the 16KB translation granule on page D8-5109 describes as starting at level 1. In this case, for some IA sizes concatenation of translation tables means the lookup can, instead, start at level 2.

The following subsections describe these examples of the initial lookup:

- Initial lookup at level 0, 16KB translation granule.
- Initial lookup at level 1, 16KB translation granule.

In all cases, for a stage 2 translation, the VTCR_EL2.SL0 field must indicate the required initial lookup level, and this level must be consistent with the value of the VTCR_EL2.T0SZ field, see Stage 2 address translation using the 16KB translation granule on page D8-5109.

Initial lookup at level 0, 16KB translation granule

This subsection describes initial lookups with an input address width of \((n+1)\) bits, meaning the input address is IA\([n:0]\). As described in Translation using the 16KB granule on page D8-5106, the only case where an address translation using the 16KB granule starts at level 0 is a stage 1 translation of a 48-bit input address, IA\([47:0]\). For this lookup:

- The required translation table has only two entries, meaning its size is 16 bytes, and it must be aligned to 16 bytes.
- TTBR_ELx[47:4] specify the translation table base address.

Figure K9-4 shows this lookup.

![Figure K9-4 Initial lookup for VMSAv8-64 using the 16KB granule, starting at level 0](image)

Supported input address range is IA\([47:0]\). The field marked ‡ is RES0*. For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor. * Field has additional properties to the default RES0 definition, see the register description for more information.

Initial lookup at level 1, 16KB translation granule

This subsection describes initial lookups with an input address width of \((n+1)\) bits, meaning the input address is IA\([n:0]\).

For a stage 1 or stage 2 initial lookup at level 1, without use of concatenated translation tables

As described in Translation using the 16KB granule on page D8-5106, this applies to IA\([n:0]\), where \(36 \leq n \leq 46\). For these lookups:

- There is a single translation table at this level.
- TTBR_ELx\([47:(n-32)]\) specify the translation table base address.
- Bit[n:36] of the input address are bits\([(n-33):3]\) of the descriptor offset in the translation table.
**Figure K9-5 shows this lookup.**

For a stage 2 initial lookup at level 1, with concatenated translation tables

As described in *Stage 2 address translation using the 16KB translation granule* on page D8-5109, the only case where an address translation using the 16KB granule starts at level 1 because of concatenation of translation tables is a stage 2 translation of a 48-bit input address, IA[47:0]. For this lookup:

- There are two concatenated translation tables at this level.
- These concatenated translation tables must be aligned to $2 \times 16\text{KB}$. This means TTBR_ELx[14] must be zero.
- TTBR_ELx[47:15] specify the base address of the block of two concatenated translation tables.
- Bits[47:36] of the input address are bits[14:3] of the descriptor offset from the base address of the block of concatenated translation tables.

**Figure K9-6 shows this lookup.**

For a stage 2 initial lookup at level 1, without concatenation of translation tables

As described in *Stage 2 address translation using the 16KB translation granule* on page D8-5109, the only case where an address translation using the 16KB granule starts at level 1 because of concatenation of translation tables is a stage 2 translation of a 48-bit input address, IA[47:0]. For this lookup:

- There are two concatenated translation tables at this level.
- These concatenated translation tables must be aligned to $2 \times 16\text{KB}$. This means TTBR_ELx[14] must be zero.
- TTBR_ELx[47:15] specify the base address of the block of two concatenated translation tables.
- Bits[47:36] of the input address are bits[14:3] of the descriptor offset from the base address of the block of concatenated translation tables.

**Figure K9-6 Initial lookup for VMSAv8-64 using the 16KB granule, starting at level 1, without concatenation**

**Performing the initial lookup using the 64KB translation granule**

This subsection describes examples of the initial lookup when using the 64KB translation granule that *Stage 2 address translation using the 64KB translation granule* on page D8-5115 describes as starting at level 1 or at level 2. It includes those stage 2 translations where concatenation of translation tables is required for the lookup to start at level 2. This means that it gives specific examples of the mechanisms described in *Translation process on page D8-5093*. 

**Figure K9-6 Initial lookup for VMSAv8-64 using the 16KB granule, starting at level 1, with concatenation**

Supported input address range is IA[y:0], $4 \leq x \leq 14$, $y = x + 32$.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

**Figure K9-6 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 1, with concatenation**

Supported input address range is IPA[47:0]. The bit marked ‡ must be zero.

* Field has additional properties to the default RES0 definition, see the register description for more information.
Note

For stage 2 translations, the same principles apply to an initial lookup that *Stage 2 address translation using the 64KB translation granule* on page D8-5115 describes as starting at level 2. In this case, for some IA sizes concatenation of translation tables means the lookup can, instead, start at level 3.

The following subsections describe these examples of the initial lookup:

- *Initial lookup at level 1, 64KB translation granule.*
- *Initial lookup at level 2, 64KB translation granule.*

In all cases, for a stage 2 translation, the VTCR_EL2.SL0 field must indicate the required initial lookup level, and this level must be consistent with the value of the VTCR_EL2.T0SZ field, see *Stage 2 address translation using the 64KB translation granule* on page D8-5115.

**Initial lookup at level 1, 64KB translation granule**

This subsection describes initial lookups with an input address width of \((n+1)\) bits, meaning the input address is \(IA[n:0]\). As described in *Translation using the 64KB granule* on page D8-5112, a stage 1 or stage 2 initial lookup at level 1 is required when \(42 \leq n \leq 47\). For these lookups:

- The size of the translation table is \(2^{(n-39)}\) bytes. This means the size of the translation table, at this level, is always less than the granule size. The address of this translation table must align to the size of the table.
- Bits\([n:42]\) of the input address are bits\([n-39:3]\) of the descriptor offset in the translation table.
- Bits\([47:(n-38)]\) of the TTBR_ELx specify the translation table base address.

Figure K9-7 shows this lookup.

![Figure K9-7 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 1](image)

Supported input address range is \(IA[y:x] = x + 38\). When \(y = 47\) the field marked ‡ is absent.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

**Initial lookup at level 2, 64KB translation granule**

This subsection describes initial lookups with an input address width of \((n+1)\) bits, meaning the input address is \(IA[n:0]\).

For a stage 1 or stage 2 initial lookup at level 2, without the use of concatenated translation tables

As described in *on page D8-5112*, this applies to \(IA[n:0]\), where \(29 \leq n \leq 41\). For these lookups:

- There is a single translation table at this level.
- TTBR_ELx\([47:(n-25)]\) of the specify the translation table base address.
- Bits\([n:29]\) of the input address are bits\([n-26:3]\) of the descriptor offset in the translation table.
Figure K9-8 shows this lookup.

![Figure K9-8 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 2, without concatenation](image)

For a stage 2 initial lookup at level 2, with concatenated translation tables

As described in *Stage 2 address translation using the 64KB translation granule* on page D8-5115, this applies to IA[n:0], where 42 ≤ n ≤ 45. For these lookups:

- There are 2^(m-41) concatenated translation tables at this level.
- These concatenated translation tables must be aligned to 2^(m-41)×64KB. This means TTBR_ELx[(n-26):16] must be zero.
- TTBR_ELx[47:(n-25)] specify the base address of the block of translation tables.
- Bits[n:42] of the input address are bits[(n-26):16] of the descriptor offset from the base address of the block of translation tables.

Figure K9-9 shows this lookup.

![Figure K9-9 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 2, with concatenation](image)

Supported input address range is IPA[y:0], 4 ≤ x ≤ 20, y = x + 25.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K9-9 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 2, with concatenation

**K9.1.2 Full translation flows for VMSAv8-64 address translation**

In a translation table walk, only the first lookup uses the translation table base address from the appropriate TTBR_ELx. Subsequent lookups use a combination of address information from:

- The table descriptor read in the previous lookup.
- The input address.
This section describes example full translation flows, from the initial lookup to the address of a memory page. The example flows:

- Resolve the maximum-sized IA range supported by the initial lookup level.
- Do not have any concatenation of translation tables.
- Cover only the 4KB and the 64KB translation granules.

**Examples of performing the initial lookup** on page K9-11714 described how either reducing the IA range or concatenating translation tables affects the initial lookup.

--- **Note** ---

Reducing the IA range or concatenating translation tables affects only the initial lookup.

---

The following sections describe full VMSAv8-64 translation flows, down to an entry for a memory page:

- **The address and properties fields shown in the translation flows.**
- **Full translation flow using the 4KB granule and starting at level 0** on page K9-11722.
- **Full translation flow using the 4KB granule and starting at level 1** on page K9-11723.
- **Full translation flow using the 64KB granule and starting at level 1** on page K9-11724.
- **Full translation flow using the 64KB granule and starting at level 2** on page K9-11725.

**The address and properties fields shown in the translation flows**

For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state:

- Any descriptor address is the IPA of the required descriptor.
- The final output address is the IPA of the block or page.

In these cases, an EL1&0 stage 2 translation is performed to translate the IPA to the required PA.

For all other translations, the final output address is the PA of the block or page, and any descriptor address is the PA of the descriptor.

*Properties* indicates register or translation table fields that return information, other than address information, about the translation or the targeted memory region. For more information, see **Translation table descriptor formats** on page D8-5123.
Full translation flow using the 4KB granule and starting at level 0

Figure K9-10 shows the complete translation flow for a stage 1 translation table walk for a 48-bit input address. This lookup must start with a level 0 lookup. For more information about the fields shown in the figure, see The address and properties fields shown in the translation flows on page K9-11721.

For details of Properties fields, see the register or descriptor description.

* Field has additional properties to the default RES0 definition, see the register description for more information.

**Figure K9-10 Complete stage 1 translation of a 48-bit address using the 4KB translation granule**
If the level 1 lookup or level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K9-10 on page K9-11722 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.

Full translation flow using the 4KB granule and starting at level 1

Figure K9-11 shows the complete translation flow for a stage 1 translation table walk for a 39-bit input address. This lookup must start with a level 1 lookup. For more information about the fields shown in the figure, see The address and properties fields shown in the translation flows on page K9-11721.

For details of Properties fields, see the register or descriptor description.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K9-11 Complete stage 1 translation of a 39-bit address using the 4KB translation granule

If the level 1 lookup or the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K9-11 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.
Comparing this translation with the translation for a 48-bit address, shown in Figure K9-10 on page K9-11722, shows how the translation for the 42-bit address start the same lookup process one stage later.

**Full translation flow using the 64KB granule and starting at level 1**

Figure K9-12 shows the complete translation flow for a stage 1 translation table walk for a 48-bit input address. This lookup must start with a level 1 lookup. For more information about the fields shown in the figure, see *The address and properties fields shown in the translation flows on page K9-11721*.

![Full translation flow diagram](diagram.png)

**Figure K9-12 Complete stage 1 translation of a 48-bit address using the 64KB translation granule**

If the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K9-12 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.
The level 1 lookup resolves only 6 bits of the input address. As described in *Performing the initial lookup using the 64KB translation granule on page K9-11718*, this means:

- The translation table size for this level is only 512 bytes.
- The required translation table alignment for this level is 512 bytes.
- The Base address field in the TTBR_ELx is extended, at the low-order end, to be bits[47:9].

**Full translation flow using the 64KB granule and starting at level 2**

Figure K9-13 shows the complete translation flow for a stage 1 translation table walk for a 42-bit input address. This lookup must start with a level 2 lookup. For more information about the fields shown in the figure, see *The address and properties fields shown in the translation flows on page K9-11721*.

![Translation Flow Diagram](image)

**Figure K9-13 Complete stage 1 translation of a 42-bit address using the 64KB translation granule**

If the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K9-13 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.

Comparing this translation with the translation for a 48-bit address, shown in Figure K9-12 on page K9-11724, shows:

- The translation for the 42-bit address starts the same lookup process one stage later.
- Because the initial lookup resolves 13 bits of address:
  - The translation table size for this level is 64KB.
  - The required translation table alignment for this level is 64KB.
  - The Base address field in the TTBR_ELx is bits[47:16].
K9.2 AArch32 Address translation examples

The following sections give address translation examples for the VMSAv8-32 address translation formats:

- Address translation examples using the VMSAv8-32 Short descriptor translation table format.
- Address translation examples using the VMSAv8-32 Long descriptor translation table format on page K9-11731.

K9.2.1 Address translation examples using the VMSAv8-32 Short descriptor translation table format

VMSAv8-32 Short-descriptor Translation Table format descriptors on page G5-9164 describes the memory section and page option for a single VMSAv8-32 address translation. The following sections show the full translation flow for each of these options:

- Translation flow for a Supersection.
- Translation flow for a Section on page K9-11728.
- Translation flow for a Large page on page K9-11729.
- Translation flow for a Small page on page K9-11730.

The address and Properties fields shown in the translation flows on page K9-11730 summarizes the information returned by the lookup.

Translation flow for a Supersection

Figure K9-14 on page K9-11727 shows the complete translation flow for a Supersection. For more information about the fields shown in this figure, see The address and Properties fields shown in the translation flows on page K9-11730.
Address Translation Examples

K9.2 AArch32 Address translation examples

Figure K9-14 VMSAv8-32 Short-descriptor Supersection address translation

For a translation based on TTBR0, N is the value of TTBCR.N.
For a translation based on TTBR1, N is 0.
For details of Properties fields, see the register or descriptor description.

‡ This field is absent if N is 0.
BA = Base address.

Figure K9-14 shows how, when the input address, the VA, addresses a Supersection, the top four bits of the Supersection index bits of the address overlap the bottom four bits of the Table index bits. For more information, see Additional requirements for Short-descriptor format translation tables on page G5-9167.
Translation flow for a Section

Figure K9-15 shows the complete translation flow for a Section. For more information about the fields shown in this figure, see The address and Properties fields shown in the translation flows on page K9-11730.

For a translation based on TTBR0, N is the value of TTBCR.N.
For a translation based on TTBR1, N is 0.
For details of Properties fields, see the register or descriptor description.

Figure K9-15 VMSAv8-32 Short-descriptor Section address translation
Translation flow for a Large page

Figure K9-16 shows the complete translation flow for a Large page. For more information about the fields shown in this figure, see The address and Properties fields shown in the translation flows on page K9-11730.

- **Input address**
- **TTBR**
- **Level 1 descriptor address**
- **Level 1 descriptor**
- **Level 2 descriptor address**
- **Level 2 descriptor**
- **Output address, A[39:0]

† This field is absent if N is 0.
L1 = Level 1, L2 = Level 2.
For a translation based on TTBR0, N is the value of TTBCR.N.
For a translation based on TTBR1, N is 0.
For details of Properties fields, see the register or descriptor description.

---

**Note**

Figure K9-16 shows how, when the input address, the VA, addresses a Large page, the top four bits of the page index bits of the address overlap the bottom four bits of the level 1 table index bits. For more information, see Additional requirements for Short-descriptor format translation tables on page G5-9167.
Translation flow for a Small page

Figure K9-17 shows the complete translation flow for a Small page. For more information about the fields shown in this figure, see The address and Properties fields shown in the translation flows.

<table>
<thead>
<tr>
<th>31</th>
<th>32-N1-N</th>
<th>20</th>
<th>19</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>➕</td>
<td>L1 table index</td>
<td>L2 table index</td>
<td>Page index</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input address

<table>
<thead>
<tr>
<th>31</th>
<th>14-N1-N</th>
<th>13-N</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>➕</td>
<td>Translation base</td>
<td>RESD</td>
<td>Properties</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TTBR

<table>
<thead>
<tr>
<th>39</th>
<th>32</th>
<th>31</th>
<th>14-N1-N</th>
<th>13-N</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0</td>
<td>Translation base</td>
<td>L1 table index</td>
<td>0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Level 1 lookup

<table>
<thead>
<tr>
<th>31</th>
<th>10</th>
<th>9</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>➕</td>
<td>Translation table base address</td>
<td>Properties</td>
<td>0 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Level 1 descriptor

<table>
<thead>
<tr>
<th>39</th>
<th>32</th>
<th>31</th>
<th>10</th>
<th>9</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0</td>
<td>Translation table base address</td>
<td>L2 table index</td>
<td>0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Level 2 lookup

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>➕</td>
<td>Small page base address</td>
<td>Properties</td>
<td>1 x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Level 2 descriptor

<table>
<thead>
<tr>
<th>39</th>
<th>32</th>
<th>31</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0</td>
<td>Small page base address</td>
<td>Page index</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output address, A[39:0]

كات bleibt the address and Properties fields shown in the translation flows.

For the Non-secure PL1&0 stage 1 translation tables:
- Any descriptor address is the IPA of the required descriptor.
- The final output address is the IPA of the Section, Supersection, Large page, or Small page.

In these cases, a PL1&0 stage 2 translation is performed to translate the IPA to the required PA.
Otherwise, the address is the PA of the descriptor, Section, Supersection, Large page, or Small page.

*Properties* indicates register or translation table fields that return information, other than address information, about the translation or the targeted memory region. For more information, see *Information returned by a translation table lookup* on page G5-9159, and the description of the register or translation table descriptor.

For translations using the Short-descriptor translation table format, *VMSAv8-32 Short-descriptor Translation Table format descriptors* on page G5-9164 describes the descriptors formats.

### K9.2.2 Address translation examples using the VMSAv8-32 Long descriptor translation table format

As described in *Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format* on page G5-9187, in a translation table walk, only the first lookup uses the translation table base address from the appropriate TTBR. Subsequent lookups use a combination of address information from:

- The table descriptor read in the previous lookup.
- The input address.

The following sections give examples of full VMSAv8-32 Long-descriptor format address translation flows, down to an entry for a 4KB page:

- *Full translation flow, starting at level 1 lookup.*
- *Full translation flow, starting at level 2 lookup* on page K9-11733.

*The address and Properties fields shown in the translation flows* on page K9-11730 summarizes the information returned by the lookup.

#### Full translation flow, starting at level 1 lookup

Figure K9-18 on page K9-11732 shows the complete translation flow for a VMSAv8-32 Long-descriptor stage 1 translation table walk that starts with a level 1 lookup. For more information about the fields shown in the figure, see *The address and Properties fields shown in the translation flows* on page K9-11730.
Figure K9-18 Complete VMSAv8-32 Long-descriptor format stage 1 translation, starting at level 1

If the level 1 lookup or the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

If bits[47:40] of the TTBR or the descriptor are not zero then the lookup will generate an Address size fault, see Address size fault on page G5-9240.

A stage 2 translation that starts at a level 1 lookup differs from the translation shown in Figure K9-18 only as follows:

- The possible values of $n$ are 4-13, to support an input address of between 31 and 40 bits.
- A descriptor and output addresses are always PAs.
**Full translation flow, starting at level 2 lookup**

Figure K9-19 shows the complete translation flow for a stage 1 VMSA v8-32 Long-descriptor translation table walk that starts at a level 2 lookup. For more information about the fields shown in the figure, see The address and Properties fields shown in the translation flows on page K9-11730.

For details of Properties fields, see the register or descriptor description.

‡ See the lookup description for more information about bits[40:47] of the TTBR and descriptors

**Figure K9-19 Complete VMSA v8-32 Long-descriptor format stage 1 translation, starting at level 2**

If the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

If bits[47:40] of the TTBR or the descriptor are not zero then the lookup will generate an Address size fault, see Address size fault on page G5-9240.

A stage 2 translation that starts at a level 2 lookup differs from the translation shown in Figure K9-19 only as follows:

- The possible values of \( n \) are 7-16, to support an input address of up to 34 bits.
- The descriptor and output addresses are always PAs.

The address and Properties fields shown in the translation flows

For the Non-secure PL1&0 stage 1 translation:

- Any descriptor address is the IPA of the required descriptor.
- The final output address is the IPA of the block or page.

In these cases, a PL1&0 stage 2 translation is performed to translate the IPA to the required PA.
For all other translations, the final output address is the PA of the block or page, and any descriptor address is the PA of the descriptor.

*Properties* indicates register or translation table fields that return information, other than address information, about the translation or the targeted memory region. For more information, see *Information returned by a translation table lookup* on page G5-9159, and the description of the register or translation table descriptor.

For translations using the Long-descriptor translation table format, *VMSA v8-32 Long-descriptor Translation Table format descriptors* on page G5-9173 describes the descriptors formats.
Appendix K10
Example OS Save and Restore Sequences

This appendix provides possible OS Save and Restore sequences for a Debug implementation using the V8A or later architectures. It contains the following sections:
- Save Debug registers on page K10-11736.
- Restore Debug registers on page K10-11738.
K10.1 Save Debug registers

This section shows how to save the registers that are used by an external debugger.

; On entry, X0 points to a block to save the debug registers in.
; Returns the pointer beyond the block and corrupts X1-X3

SaveDebugRegisters

; (1) Set OS Lock.
MOV     X2,#1                       ; Set the OS Lock. In AArch64 state, the OS Lock
MSR     OSLAR_EL1,X2                ; is writable via OSLAR.
ISB                                 ; Context synchronization event

; (2) Walk through the registers, saving them
MRS     X1,OSDTRRX_EL1              ; Read DTRRX
MRS     X2,OSDTRTX_EL1              ; Read DTRTX
STP     W1,W2,[X0],#8               ; Save { DTRRX, DTRTX }
MRS     X1,OSECCR_EL1               ; Read ECCR
MRS     X2,MDSCR_EL1                ; Read DSCR
STP     W1,W2,[X0],#8               ; Save { ECCR, DSCR }
[ AARCH32_SUPPORTED
MRS     X1,DBGVCR32_EL2             ; Read DBGVCR
MRS     X2,DBGCLAIMCLR_EL1          ; Read CLAIM - note, have to read via CLAIMCLR
STP     W1,W2,[X0],#8               ; Save { VCR, CLAIM }
]

;; Macros for saving off a "register pair"
;; $WB     is W for watchpoint, B for breakpoint
;; $num    is the pair's number
;; X0 contains a pointer for the value words
;; X1 contains a pointer for the control words
;; W2 contains the max index
MACRO
SaveRP  $WB,$num, $exit
MRS     X3,DBG$WB.VR$num._EL1       ; Read DBGxVRn
STR     X3,[X0],#8                  ; Save { xVRn }
MRS     X3,DBG$WB.CR$num._EL1       ; Read DBGxCRn
STR     W3,[X0],#4                  ; Save { xCRn }.
[ $num > 1 :LAND: $num < 15
CMP     W1,#$num
BEQ     $exit
]
MEND

; (3) Breakpoints
MRS     X1,ID_AA64DFR0_EL1
UBFX    W1,W1,#12,#4                ; Extract BRPs field
MACRO
SaveBRP $num ; Save a Breakpoint Register Pair
SaveRP  B,$num,SaveDebugRegisters_Watchpoints
MEND
SaveBRP 0
SaveBRP 1
SaveBRP 2
;; and so on to ...
SaveBRP 15

SaveDebugRegisters_Watchpoints

; (4) Watchpoints
MRS     X1,ID_AA64DFR0_EL1
UBFX    W1,W1,#20,#4                ; Extract WRP field
MACRO
SaveWRP $num ; Save a Watchpoint Register Pair
SaveRP  W,$num,SaveDebugRegisters_Exit
MEND
SaveWRP 0
SaveWRP 1
SaveWRP 2
;; and so on to ...
SaveWRP 15

SaveDebugRegisters.Exit

; (5) Return the pointer to first word not read. This pointer is already in X0, so
; all that is needed is to return from this function. The OS double-lock (OSDLR_EL1.DLK) is
; locked later, just before the final entry to WFI state.
RET
K10.2 Restore Debug registers

This section shows how to restore the registers that are used by an external debugger.

; On entry, X0 points to a block of saved debug registers.
; Returns the pointer beyond the block and corrupts R1-R3,R12.

RestoreDebugRegisters
; (1) Lock OS Lock. The lock will already be set, but this write is included to ensure it
; is locked.
MOV     X2,#1                       ; Lock the OS Lock. In AArch64 state, the OS Lock
MSR     OSLAR_EL1,X2                ; is writable via OSLAR.
ISB                                 ; Context synchronization event
MSR     MDSCR_EL1, XZR               ; Initialize MDSCR_EL1

; (2) Walk through the registers, restoring them
LDP     W1,W2,[X0],#8               ; Read { DTRRX, DTRTX }
MSR     OSDTRRX_EL1,X1              ; Restore DTRRX
MSR     OSDTRTX_EL1,X2              ; Restore DTRTX
LDP     W1,W3,[X0],#8               ; Read { DSCR, ECCR }
MSR     OSECCR_EL1,X2               ; Restore ECCR
[ AARCH32_SUPPORTED
LDP     W1,W2,[X0],#8               ; Read { VCR, CLAIM }
MSR     DBGVCR32_EL2,X1             ; Restore DBGVCR
MSR     DBGCLAIMSET_EL1,X2          ; Restore CLAIM - note, writes CLAIMSET
]

;; Macro for restoring a "register pair"
MACRO
RestoreRP $WB,$num,$exit
LDR     X3,[X0],#8                  ; Read { xVRn }
MSR     DBG$WB.VR$num._EL1,X3       ; Restore DBGxVRn
LDR     W3,[X0],#4                  ; Read { xCRn }
MSR     DBG$WB.CR$num._EL1,X3       ; Restore DBGxCRn
[ $num >= 1 :LAND: $num < 15
CMP     W1,#$num
BEQ     $exit
]
MEND

; (3) Breakpoints
MRS     X1,ID_AA64DFR0_EL1
UBFX    W1,W1,#12,#4                ; Extract BRPs field
MACRO
RestoreBRP $num ; Restore a Breakpoint Register Pair
RestoreRP B,$num,RestoreDebugRegisters_Watchpoints
MEND
RestoreBRP 0
RestoreBRP 1
RestoreBRP 2
;; and so on until ...
RestoreBRP 15

RestoreDebugRegisters_Watchpoints
; (4) Watchpoints
MRS     X1,ID_AA64DFR0_EL1          ; Read DBGDIDR
UBFX    W1,W1,#20,#4                ; Extract WRPs field
MACRO
RestoreWRP $num ; Restore a Watchpoint Register Pair
RestoreRP W,$num,RestoreDebugRegisters.Exit
MEND
RestoreWRP 0
RestoreWRP 1
RestoreWRP 2
;; and so on until ...
RestoreWRP 15
RestoreDebugRegisters.Exit
MSR MDSCR_EL1, X3 ; Restore DSCR

; (5) Clear the OS Lock.
ISB
MOV    X2, #0 ; Clear the OS Lock. In AArch64 state, the OS Lock
MSR    OSLAR_EL1, X2 ; is writable via OSLAR.

; (6) A final ISB guarantees the restored register values are visible to subsequent
; instructions.
ISB

; (7) Return the pointer to first word not read. This pointer is already in X0, so
; all that is needed is to return from this function.
RET
Appendix K11
Recommended Upload and Download Processes for External Debug

This appendix contains the following section:
• Using memory access mode in AArch64 state on page K11-11742.

Note
This description is not part of the Arm architecture specification. It is included here as supplementary information, for the convenience of developers and users who might find this information useful.
K11.1 Using memory access mode in AArch64 state

Figure K11-1 and Figure K11-2 on page K11-11743 show the processes for using memory access mode to implement a download (external host to target) and an upload (target to external host).

To transfer \( n \) words of data:
- The download sequence needs \( n+6 \) accesses by the external debug interface.
- The upload sequence needs \( n+8 \) accesses by the external debug interface.

In both cases, in the innermost loop the debugger can make an external access to a DTR without polling EDSCR after each write as underrun and overrun detection prevent failure. Normally external accesses from the debugger are outpaced by the memory accesses of the PE, making underruns and overruns unlikely. If this is not the case, the EDSCR.ERR flag is set to 1. This is checked once at the end of the sequence, although a debugger can check it more often, for example once for each page. If the EDSCR.ERR flag is set to 1 because of overrun or underrun, the debugger can restart. The address to restart from is frozen in X0. EDSCR.ERR might also be set because of a Data abort.

If underruns and overruns are common, the debugger can pace itself accordingly.

--- Note ---
- The base address must be a multiple of 4.
- The order of the writes that set up the address does not matter in Debug state.

Figure K11-1 Fast code download in AArch64 state (external host to target)
In Figure K11-1 on page K11-11742, the sequence for the fast code download is as follows:

1. Setup. From the external debug interface:
   a. Write address [31:0] to DBGDTRRX_EL0.
   b. Write address [63:32] to DBGDTRTX_EL0.
   c. Write `MRS X0, DBGDTR_EL0` to `EDITR`. The PE executes this instruction.
   d. Set `EDSCR.MA` to 1.

2. Loop \( n \) times. From the external debug interface:
   a. Write to `DBGDTRRX_EL0`. The PE reads the word from DTRRX and stores it to memory. It increments X0 by 4.

3. Epilogue. From the external debug interface:
   a. Clear `EDSCR.MA` to 0.
   b. Read `EDSCR` to check for overruns or Data Aborts during download.

---

In Figure K11-2, the sequence for the fast data upload in AArch64 state (target to external host) is as follows:

1. Setup. From the external debug interface:
   a. Write address [31:0] to DBGDTRRX_EL0.

---

**Figure K11-2 Fast data upload in AArch64 state (target to external host)**
b. Write address [63:32] to DBGDTRTX_EL0.
c. Write MSR X0, DBGDTR_EL0 to EDITR.
d. Write MSR DBGDTR_EL0, X0 to EDITR. This dummy operation ensures EDSCR.TXfull == 1.
e. Set EDSCR.MA to 1.
f. Read DBGDTRTX_EL0 and discard the value. The PE returns the previous DTR value, loads the first word, and writes it to DTR. It increments X0 by 4.

2. Loop \( n-1 \) times. From the external debug interface:
   a. Read DBGDTRTX_EL0. The PE returns the previous DTRTX value, loads a new word, and writes it to DTRTX. It increments X0 by 4.

3. Epilogue. From the external debug interface:
   a. Clear EDSCR.MA to 0.
   b. Read DBGDTRTX_EL0 for the \( n \)th value.
   c. Read EDSCR to check for underruns, overruns or Data Aborts during upload.
Appendix K12
Software Usage Examples

This appendix gives software usage examples, for cases where these are likely to contribute significantly to an understanding of the Arm architecture.

It contains the following sections:

• Use of the Advanced SIMD complex number instructions on page K12-11746.
• Use of the Armv8.2 extensions to the Cryptographic Extension on page K12-11748.
K12.1 Use of the Advanced SIMD complex number instructions

FEAT_FCMA provides instructions to aid floating-point computations of complex numbers. This section illustrates the use of these instructions for complex arithmetic. It is not part of the Arm architecture definition.

This section uses the AArch64 instructions FCA00 and FOILA - usage of the AArch32 instructions VCA00 and VOMLA is similar.

When using the instructions implemented by FEAT_FCMA, a complex number is represented in a SIMD&FP register as a pair of adjacent elements, each holding a floating-point number, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number.

K12.1.1 Complex addition

Simple complex addition on a vector of complex numbers is already provided by the vector form of the FADD instruction.

The functionality that FCA00 adds is to rotate each complex number in the second vector by 90 degrees or 270 degrees counterclockwise (considering the complex numbers on an Argand diagram) before performing the addition. Mathematically, this is equivalent to multiplying the second complex number by i or -i before addition.

This means, given a complex number z stored in a pair of elements in one vector, and a complex number w stored in the corresponding element pair in another vector:

- FADD calculates z + w.
- FCA00 calculates z ± iw.

K12.1.2 Complex multiplication

The FOILA instruction does not provide functionality for complex multiplication directly. However, a pair of FOILA instructions can provide this function.

The FOILA instruction operates on corresponding pairs of complex numbers stored in SIMD&FP vector registers, and adds the result to the corresponding complex number in the destination SIMD&FP vector register. This computation is as follows:

1. The second complex number is rotated by 0, 90, 180 or 270 degrees counterclockwise.
2. That complex number is multiplied by either the real or imaginary part of the first complex number:
   - When the rotation is 0 or 180 degrees, the real part is used.
   - When the rotation is 90 or 270 degrees, the imaginary part is used.
3. The resulting complex number is added to the corresponding complex number in the destination register.

Mathematically, considering the complex numbers on an Argand diagram:

- Rotation by 180 degrees is equivalent to negation.
- Rotation by 90 degrees is equivalent to multiplying by i.
- Rotation by 270 degrees is equivalent to multiplying by -i.

This means that, for a first complex number z, where z = a+bi, and a second complex number w, if initially the corresponding complex number in the destination register is zero:

- When the rotation is 0 degrees the result of the multiply-add is aw.
- When the rotation is 180 degrees, the result is -aw.
- When the rotation is 90 degrees, the result is biw.
- When the rotation is 270 degrees, the result is -biw.

This means that, if the destination register is zeroed and an FOILA instruction is executed with a rotation parameter of 0, and then the same instruction is executed with a rotation parameter of 90:

- The first execution returns aw in the destination register.
- The second execution accumulates biw to this, meaning the result is aw+biw.
- This result is the product of (a+bi)w, which is the product zw.
So, this pair of instructions can be used to implement complex multiplication.

After zeroing V0, the syntax of a pair of instructions to perform this complex number multiplication might be:

\[ \text{FCMLA V0.4S, V1.4S, V2.4S, #0} \]
\[ \text{FCMLA V0.4S, V1.4S, V2.4S, #90} \]

Other simple pairs of \text{FCMLA} instructions perform useful computations. For example, considering a first complex number \( z \) and second complex number \( w \), defined as before, and a destination register that has been zeroed before the first \text{FCMLA} instruction is executed:

1. The following pair of instructions calculates the complex conjugate of \( z \) multiplied by \( w \).
   \[ \text{FCMLA V0.4S, V1.4S, V2.4S, #0} \]
   \[ \text{FCMLA V0.4S, V1.4S, V2.4S, #270} \]

2. The following pair of instructions calculates the negation of \( z \) multiplied by \( w \).
   \[ \text{FCMLA V0.4S, V1.4S, V2.4S, #180} \]
   \[ \text{FCMLA V0.4S, V1.4S, V2.4S, #270} \]

3. The following pair of instructions calculates the negation of the complex conjugate of \( z \) multiplied by \( w \).
   \[ \text{FCMLA V0.4S, V1.4S, V2.4S, #180} \]
   \[ \text{FCMLA V0.4S, V1.4S, V2.4S, #90} \]

\textbf{Note}

For these examples, the following caveats must be considered:

- \text{FCMLA} performs a fused multiply-add, meaning there is no intermediate rounding. This lack of intermediate rounding can give unexpected results in some cases. Arm expects that these instructions are only used in situations where the effect of the rounding of these results is not material to the calculation.

- When using the \text{FCMLA} instructions, the behavior of \((\infty + \infty i)\) multiplied by \((0+i)\) is \((\text{NaN}+\text{NaNi})\), rather than the result expected by ISO C, which is complex \( \infty \).
K12.2 Use of the Armv8.2 extensions to the Cryptographic Extension

K12.2.1 Use of the SHA512 instructions

These instructions are implemented when FEAT_SHA512 is implemented.

The following code sequence shows the use of the SHA512 instructions to calculate a SHA512 hash iteration of 80 rounds. This code is not fully optimized.

```assembly
// X0 contains the pointer to the bottom of the (padded) 16*64 bytes of message to be
// hashed, with space above that message to hold a further 64 + 64 bytes of working
// data
// X1 contains the pointer to the 0th element of 80 64-bit constants (in ascending addresses) defined in
// the SHA2 specification
// X2 contains a loop variable
// V4,V5,V6, V7 hold VS0 to VS3 respectively
// V8 holds running hash V1
// V9 holds running hash V0
MOV  X2, #0
loop1:
    LD1  {V0.2D}, [X0]              // Data
    LD1  {V1.2D}, [X1]              // K values
    ADD  X1, X1, #16
    ADD  X0, X0, #16
    ADD  X2, X2, #16
    ADD  V2.2D, V0.2D, V1.2D
    EXT  V2.16B, V2.16B, V2.16B, #8
    EXT  V8.16B, V6.16B, V7.16B, #8
    EXT  V9.16B, V5.16B, V6.16B, #8
    ADD  V7.2D, V7.2D, V2.2D
    SHA512H Q7, Q8, V9.2D
    ADD  V10.2D, V5.2D, V7.2D
    SHA512H2 Q7, Q5, V4.2D
    MOV  V5.16B, V4.16B
    MOV  V4.16B, V7.16B
    MOV  V7.16B, V6.16B
    MOV  V6.16B, V10.16B
    CMP  X2, #128
    BLT   loop1

// work out pointers to previous words in the data
SUB  X3, X0, #128
SUB  X4, X0, #112
SUB  X5, X0, #16
SUB  X6, X0, #56
loop2:
    LD1  {V11.2D}, [X3]
    LD1  {V12.2D}, [X4]
    LD1  {V13.2D}, [X5]
    LD1  {V14.2D}, [X6]
    SHA512SU0 V11.2D, V12.2D
    SHA512SU1 V11.2D, V13.2D, V14.2D
    ST1  {V11.2D}, [X0]
    LD1  {V1.2D}, [X1]              // K values
    ADD  X0, X0, #16
    ADD  X1, X1, #16
    ADD  X3, X3, #16
    ADD  X4, X4, #16
    ADD  X5, X5, #16
    ADD  X6, X6, #16
    ADD  X2, X2, #16
    ADD  V2.2D, V11.2D, V1.2D
    EXT  V2.16B, V2.16B, V2.16B, #8
    EXT  V8.16B, V6.16B, V7.16B, #8
    EXT  V9.16B, V5.16B, V6.16B, #8
    ADD  V7.2D, V7.2D, V2.2D
    SHA512H Q7, Q8, V9.2D
```

// X0 contains the pointer to the bottom of the (padded) 16*64 bytes of message to be
// hashed, with space above that message to hold a further 64 + 64 bytes of working
// data
// X1 contains the pointer to the 0th element of 80 64-bit constants (in ascending addresses) defined in
// the SHA2 specification
// X2 contains a loop variable
// V4,V5,V6, V7 hold VS0 to VS3 respectively
// V8 holds running hash V1
// V9 holds running hash V0
K12.2.2 Use of the SHA3 instructions

These instructions are implemented when FEAT_SHA3 is implemented.

The following code sequence shows the use of the SHA3 instructions to obtain the combined theta, phi, rho and chi operations of a SHA3 iteration. Arm expects the iota operation to be performed using a lookup table.

This code is not fully optimized for multiple iterations.

```
// Input State:
//   x=0  x=1  x=2  x=3  x=4
// y=0  v12  v13  v14  v10  v11
// y=1  v7   v8   v9   v5   v6
// y=2  v2   v3   v4   v0   v1
// y=3  v22  v23  v24  v20  v21
// y=4  v17  v18  v19  v15  v16

//-- Theta Calculations --/
eor3 v25.16B, v12.16B, v7.16B, v2.16B
eor3 v25.16B, v25.16B, v22.16B, v17.16B
eor3 v27.16B, v14.16B, v9.16B, v4.16B
eor3 v27.16B, v27.16B, v24.16B, v19.16B
eor3 v28.16B, v10.16B, v5.16B, v0.16B
er3 v29.16B, v11.16B, v6.16B, v1.16B
rax1 v30.2D, v29.2D, v26.2D
rax1 v31.2D, v27.2D, v29.2D
rax1 v29.2D, v25.2D, v27.2D
rax1 v27.2D, v28.2D, v25.2D
rax1 v25.2D, v26.2D, v28.2D

//-- Phi\rho Stage --/
er v12.8B, v12.8B, v30.8B
xar v26.2D, v21.2D, v27.2D, #56
xar v21.2D, v15.2D, v31.2D, #8
xar v15.2D, v22.2D, v30.2D, #23
xar v22.2D, v11.2D, v27.2D, #37
xar v11.2D, v16.2D, v27.2D, #50
xar v16.2D, v18.2D, v29.2D, #62
xar v18.2D, v5.2D, v31.2D, #9
xar v5.2D, v23.2D, v29.2D, #19
xar v23.2D, v7.2D, v30.2D, #28
xar v7.2D, v10.2D, v31.2D, #36
xar v10.2D, v20.2D, v31.2D, #43
xar v20.2D, v24.2D, v25.2D, #49
xar v24.2D, v3.2D, v29.2D, #54
xar v3.2D, v9.2D, v25.2D, #58
xar v9.2D, v2.2D, v30.2D, #61
xar v2.2D, v13.2D, v29.2D, #63
xar v13.2D, v8.2D, v29.2D, #20
xar v8.2D, v6.2D, v27.2D, #44
xar v6.2D, v19.2D, v25.2D, #3
xar v19.2D, v1.2D, v27.2D, #25
xar v1.2D, v17.2D, v30.2D, #46
```
xar v17.2D, v14.2D, v25.2D, #2  
xar v14.2D, v4.2D, v25.2D, #21  
xar v4.2D, v0.2D, v31.2D, #39

// XAR Output:  
//  
// v12  v2  v17  v7  v22  
// v23  v13  v3  v18  v8  
// v9  v24  v14  v4  v19  
// v15  v5  v20  v10  v26  
// v1  v16  v6  v21  v11  
//  
// temp: v0, v25, v27, v28, v29, v30, v31

// Phi Output:  
//  
// v12  v13  v14  v10  v11  
// v7  v8  v9  v5  v6  
// v2  v3  v4  v26  v1  
// v22  v23  v24  v20  v21  
// v17  v18  v19  v15  v16

//-- Chi transformations --/
bcax v31.16B, v26.16B, v2.16B, v1.16B  
bcaa v27.16B, v1.16B, v3.16B, v2.16B  
bcaax v28.16B, v2.16B, v4.16B, v3.16B  
bcaax v0.16B, v5.16B, v7.16B, v6.16B  
bcaax v1.16B, v6.16B, v8.16B, v7.16B  
bcaax v2.16B, v7.16B, v9.16B, v8.16B  
bcaax v3.16B, v8.16B, v5.16B, v9.16B  
bcaax v4.16B, v9.16B, v6.16B, v5.16B  
bcaax v5.16B, v10.16B, v12.16B, v11.16B  
bcaax v10.16B, v13.16B, v17.16B, v16.16B  
bcaax v11.16B, v16.16B, v18.16B, v17.16B  
bcaax v12.16B, v17.16B, v19.16B, v18.16B  
bcaax v15.16B, v20.16B, v22.16B, v21.16B  
bcaax v17.16B, v22.16B, v24.16B, v23.16B  
bcaax v18.16B, v23.16B, v20.16B, v24.16B  

//-- Output State from Chi: --/
//  
// x=0  x=1  x=2  x=3  x=4  
// y=0  v7  v8  v9  v5  v6  
// y=1  v2  v3  v4  v0  v1  
// y=2  v28  v29  v30  v31  v27  
// y=3  v17  v18  v19  v15  v16  
// y=4  v12  v13  v14  v10  v11

K12.2.3 Use of the SM3 instructions

These instructions are implemented when FEAT_SM3 is implemented.

The following code sequence shows the use of the SM3 instructions to generate a SM3 hash.

.macro MessageExpand VA, VB, VC, VD, VOUT  
EXT \VOUT().16B, \VB().16B, \VC().16B, #12  
SM3PARTW1 \VOUT().4S, \VA().4S, \VD().4S

K12-11750 Copyright © 2013-2022 Arm Limited or its affiliates. All rights reserved. Non-Confidential
EXT V17.16B, \VA().16B, \VB().16B, #12
EXT V18.16B, \VC().16B, \VD().16B, #8
SM3PARTW2 \VOUT().4S, V18.4S, V17.4S
.ends

.macro HashPt1 VA, VB, Number SM3SS1 V23.4S, V20.4S, V22.4S, V19.4S
EOR V21.16B, \VA().16B, \VB().16B
SM3TT1a V20.4S, V23.4S, V21.S\[Number\]
SM3TT2a V19.4S, V23.4S, \VA().S\[Number\]
SHL V24.4S, V22.4S, #1
SRI V24.4S, V23.4S, #31
MOV V22.16B, V24.16B
.ends

.macro HashPt2 VA, VB, Number SM3SS1 V23.4S, V20.4S, V25.4S, V19.4S
EOR V21.16B, \VA().16B, \VB().16B
SM3TT1b V20.4S, V23.4S, V21.S\[Number\]
SM3TT2b V19.4S, V23.4S, \VA().S\[Number\]
SHL V26.4S, V25.4S, #1
SRI V26.4S, V25.4S, #31
MOV V25.16B, V26.16B
.ends

// V0-V3 holds the initial message
// V19 holds EFGH which is the lower half of the input hash
// V20 holds ABCD which is the upper half of the input hash
// V21 = current VPrime
// V22 holds T in bits[127:96] = 0x79cc4519
// V25 holds second value of T in bits[127:96] = 0x9d8a7a87<31:0>

MessageExpand V0, V1, V2, V3, V4
MessageExpand V1, V2, V3, V4, V5
MessageExpand V2, V3, V4, V5, V6
MessageExpand V3, V4, V5, V6, V7
MessageExpand V4, V5, V6, V7, V8
MessageExpand V5, V6, V7, V8, V9
MessageExpand V6, V7, V8, V9, V10
MessageExpand V7, V8, V9, V10, V11
MessageExpand V8, V9, V10, V11, V12
MessageExpand V9, V10, V11, V12, V13
MessageExpand V10, V11, V12, V13, V14
MessageExpand V11, V12, V13, V14, V15
MessageExpand V12, V13, V14, V15, V16

MOV V29.16B, V19.16B
MOV V30.16B, V20.16B

HashPt1 V0,V1, 0
HashPt1 V0,V1, 1
HashPt1 V0,V1, 2
HashPt1 V0,V1, 3
HashPt1 V1,V2, 0
HashPt1 V1,V2, 1
HashPt1 V1,V2, 2
HashPt1 V1,V2, 3
HashPt1 V2,V3, 0
HashPt1 V2,V3, 1
HashPt1 V2,V3, 2
HashPt1 V2,V3, 3
HashPt1 V3,V4, 0
HashPt1 V3,V4, 1
HashPt1 V3,V4, 2
HashPt1 V3,V4, 3
HashPt2 V4,V5, 0
HashPt2 V4,V5, 1
K12.2.4 Use of the SM4 instructions

These instructions are implemented when FEAT_SM4 is implemented.

The following code sequences show the use of the SM4 instructions to perform SM4 encryption and decryption:

Encryption

```plaintext
// Encryption
// V0 contains 0xb27022dc677d919756aa3350a3b1bac6<127:0>
// V8 contains the Key
// V2 contains the data to be encrypted
// V16 contains: 0x545b6269383f464d1c232a3100070e15;
// V17 contains: 0xc4cbd2d9a8af6b6d8c9399a170777e85;
```

EOR V19.16B, V29.16B, V19.16B
EOR V20.16B, V30.16B, V20.16B

// V19 holds EFGH which is the lower half of the output hash
// V20 holds ABCD which is the upper half of the output hash
K12.2 Use of the Armv8.2 extensions to the Cryptographic Extension

Software Usage Examples

ID081822
Non-Confidential

// V18 contains: 0x343b4249181f262dfc030a11e0e7eef5;
// V19 contains: 0xa4abb2b9888f696d6c737a8150575e65;
// V20 contains: 0xe4bb229f8ff000ddec3aef3c0c7ced5;
// V21 contains: 0x848b9296686f767d4c535a6130733e45;
// V22 contains: 0xf4fb0209d8dfe6edbcc3cad1a8a7ae65;
// V23 contains: 0x646b7279484f565d2c333a4110171e25;

EOR V8.16b, V8.16b, V0.16b;
SM4EKEY V8.4S, V8.4S, V16.4S
SM4EKEY V9.4S, V8.4S, V17.4S
SM4EKEY V10.4S, V9.4S, V18.4S
SM4EKEY V11.4S, V10.4S, V19.4S
SM4EKEY V12.4S, V11.4S, V20.4S
SM4EKEY V13.4S, V12.4S, V21.4S
SM4EKEY V14.4S, V13.4S, V22.4S
SM4EKEY V15.4S, V14.4S, V23.4S

SM4E V2.4S, V8.4S
SM4E V2.4S, V9.4S
SM4E V2.4S, V10.4S
SM4E V2.4S, V11.4S
SM4E V2.4S, V12.4S
SM4E V2.4S, V13.4S
SM4E V2.4S, V14.4S
SM4E V2.4S, V15.4S

// need to reverse the order of the keys to do a decryption:
REV64 V8.4S, V8.4S
EXT V8.16b, V8.16b, V8.16b, #8
REV64 V9.4S, V9.4S
EXT V9.16b, V9.16b, V9.16b, #8
REV64 V10.4S, V10.4S
EXT V10.16b, V10.16b, V10.16b, #8
REV64 V11.4S, V11.4S
EXT V11.16b, V11.16b, V11.16b, #8
REV64 V12.4S, V12.4S
EXT V12.16b, V12.16b, V12.16b, #8
REV64 V13.4S, V13.4S

Decryption

// Decryption
// V0 contains: 0xb27022dc677d919756aa3350a3b1bac6c127:0;
// V8 contains: 0xa4abb2b9888f696d6c737a8150575e65;
// V2 contains: 0x343b4249181f262dfc030a11e0e7eef5;
// V17 contains: 0x545b6269383f464d1c232a3100070e15;
// V18 contains: 0x343b4249181f262dfc030a11e0e7eef5;
// V19 contains: 0xa4abb2b9888f696d6c737a8150575e65;
// V20 contains: 0xe4bb229f8ff000ddec3aef3c0c7ced5;
// V21 contains: 0x848b9296686f767d4c535a6130733e45;
// V22 contains: 0xf4fb0209d8dfe6edbcc3cad1a8a7ae65;
// V23 contains: 0x646b7279484f565d2c333a4110171e25;

// need to reverse the order of the words at the end of the operation
REV64 V2.4S, V2.4S
EXT V2.16b, V2.16b, V2.16b, #8

REV64 V8.4S, V8.4S
EXT V8.16b, V8.16b, V8.16b, #8
REV64 V9.4S, V9.4S
EXT V9.16b, V9.16b, V9.16b, #8
REV64 V10.4S, V10.4S
EXT V10.16b, V10.16b, V10.16b, #8
REV64 V11.4S, V11.4S
EXT V11.16b, V11.16b, V11.16b, #8
REV64 V12.4S, V12.4S
EXT V12.16b, V12.16b, V12.16b, #8
REV64 V13.4S, V13.4S
Software Usage Examples
K12.2 Use of the Armv8.2 extensions to the Cryptographic Extension

EXT V13.16B, V13.16B, V13.16B, #8
REV64 V14.4S, V14.4S
EXT V14.16B, V14.16B, V14.16B, #8
REV64 V15.4S, V15.4S
EXT V15.16B, V15.16B, V15.16B, #8

SM4E V2.4S, V15.4S
SM4E V2.4S, V14.4S
SM4E V2.4S, V13.4S
SM4E V2.4S, V12.4S
SM4E V2.4S, V11.4S
SM4E V2.4S, V10.4S
SM4E V2.4S, V9.4S
SM4E V2.4S, V8.4S

// final reversal of the order of the words in the result:
REV64 V2.4S, V2.4S
EXT V2.16B, V2.16B, V2.16B, #8
Appendix K13
Barrier Litmus Tests

This appendix gives examples of the use of the barrier instructions provided by the Arm architecture. It contains the following sections:

• Introduction on page K13-11756.
• Load-Acquire, Store-Release and barriers on page K13-11759.
• Load-Acquire Exclusive, Store-Release Exclusive and barriers on page K13-11763.
• Using a mailbox to send an interrupt on page K13-11768.
• Cache and TLB maintenance instructions and barriers on page K13-11769.
• Armv7 compatible approaches for ordering, using DMB and DSB barriers on page K13-11781.

Note
This information is not part of the Arm architecture specification. It is included here as supplementary information, for the convenience of developers and users who might require this information.
K13.1 Introduction

The exact rules for the insertion of barriers into code sequences is a very complicated subject, and this appendix describes many of the corner cases and behaviors that are possible in an implementation of the Arm architecture.

This appendix is to help programmers, hardware design engineers, and validation engineers understand the need for the different kinds of barriers.

K13.1.1 Overview of memory consistency

Early generations of microprocessors were relatively simple processing engines that executed each instruction in program order. In such processors, the effective behavior was that each instruction was executed in its entirety before a subsequent instruction started to be executed. This behavior is sometimes referred to as the Sequential Execution Model (SEM), and in this Manual it is described as Simple sequential execution of the program.

In later processor generations, the needs to increase processor performance, both in terms of the frequency of operation and the number of instructions executed each cycle, mean that such a simple form of execution is abandoned. Many techniques, such as pipelining, write buffering, caching, speculation, and out-of-order execution, are introduced to provide improved performance.

For general purpose PEs, such as Arm, these microarchitectural innovations are largely hidden from the programmer by a number of microarchitectural techniques. These techniques ensure that, within an individual PE, the behavior of the PE largely remains the same as the SEM. There are some exceptions to this where explicit synchronization is required. In the Arm architecture, these are limited to cases such as:

- Synchronization of changes to the instruction stream.
- Synchronization of changes to System registers.

In both these cases, the ISB instruction provides the necessary synchronization.

While the effect of ordering is largely hidden from the programmer within a single PE, the microarchitectural innovations have a profound impact on the ordering of memory accesses. Write buffering, speculation, and cache coherency protocols, in particular, can all mean that the order in which memory accesses occur, as seen by an external observer, differs significantly from the order of accesses that would appear in the SEM. This is usually invisible in a uniprocessor environment, but the effect becomes much more significant when multiple PEs are trying to communicate with memory. In reality, these effects are often only significant at particular synchronization boundaries between the different threads of execution.

The problems that arise from memory ordering considerations are sometimes described as the problem of memory consistency. Processor architectures have adopted one or more memory consistency models, or memory models, that describe the permitted limits of the memory re-ordering that can be performed by an implementation of the architecture. The comparison and categorization of these has generated significant research and comment in academic circles, and Arm recommends the Memory Consistency Models for Shared Memory-Multiprocessors paper as an excellent detailed treatment of this subject.

This appendix does not reproduce such a work, but instead concentrates on some cases that demonstrate the features of the weakly-ordered memory model of the Arm architecture from Armv6. In particular, the examples show how the use of the DMB and DSB memory barrier instructions can provide the necessary safeguards to limit memory ordering effects at the required synchronization points.

K13.1.2 Barrier operation definitions

The following reference, or provide, definitions of terms used in this appendix:

- **DMB**
  See Data Memory Barrier (DMB) on page B2-175.

- **DSB**
  See Data Synchronization Barrier (DSB) on page B2-178.

- **ISB**
  See Instruction Synchronization Barrier (ISB) on page B2-175.

- **Observer, Completion**
  See Definition of the Arm memory model on page B2-159.
  See Completion and endpoint ordering on page B2-168.
Program order

The order of instructions as they appear in an assembly language program. This appendix does not attempt to describe or define the legal transformations from a program written in a higher level programming language, such as C or C++, into the machine language that can then be disassembled to give an equivalent assembly language program. Such transformations are a function of the semantics of the higher level language and the capabilities and options on the compiler.

K13.1.3 Conventions

Many of the examples are written in a stylized extension to Arm assembler, to avoid confusing the examples with unnecessary code sequences.

AArch32

The construct `WAIT([Rx]==1)` describes the following sequence:

```
loop
  LDR R12, [Rx]
  CMP R12, #1
  BNE loop
```

Also, the construct `WAIT_ACQ([Rx]==1)` describes the following sequence:

```
loop
  LDA R12, [Rx]       ; load acquire ensures it is ordered before subsequent loads/stores
  CMP R12, #1
  BNE loop
```

R12 is chosen as an arbitrary temporary register that is not in use. It is named to permit the generation of a false dependency to ensure ordering.

AArch64

The construct `WAIT([Xx]==1)` describes the following sequence:

```
loop
  LDR W12, [Xx]
  CMP W12, #1
  B.NE loop
```

Also, the construct `WAIT_ACQ([Xx]==1)` describes the following sequence:

```
loop
  LDAR W12, [Xx]      ; load acquire ensures it is ordered before subsequent loads/stores
  CMP W12, #1
  B.NE loop
```

For each example, a code sequence is preceded by an identifier of the observer running it:

- P0, P1…Px refer to caching coherent PEs that implement the Armv9 or Armv8 architecture and are in the same shareability domain.
- E0, E1…Ex refer to non-caching observers that do not participate in the coherency protocol, but execute Armv9 or Armv8 instructions and have a weakly ordered memory model. This does not preclude these observers being different objects, such as DMA engines or other system Requesters.

These observers are unsynchronized other than as required by the documented code sequence.

Note

Throughout this appendix, `Armv9 or Armv8 instruction` and `instruction` refer to instructions from the A64, A32, or T32 instruction set, provided by Armv9 or Armv8 implementations.
Results are expressed in terms of `<agent>:<register>`, such as `P0:R5`. The results can be described as:

**Permissible**
This does not imply that the results expressed are required or are the only possible results. In most cases they are results that would not be possible under a sequentially consistent running of the code sequences on the agents involved. In general terms, this means that these results might be unexpected to anyone unfamiliar with memory consistency issues.

**Not permissible**
Results that the architecture expressly forbids.

**Required**
Results that the architecture expressly requires.

The examples omit the required shareability domain arguments of `DMB` and `DSB` instructions. The arguments are assumed to be selected appropriately for the shareability domains of the observers.

In AArch32 state, where the barrier function in the litmus test can be achieved by a `DMB ST`, that is a barrier to stores only, this is shown by the use of `DMB [ST]`. This indicates that the `ST` qualifier can be omitted without affecting the result of the test. In some implementations `DMB ST` is faster than `DMB`.

For AArch64 code, the shareability domain of the `DMB` or `DSB` must be included. This is shown in this manual using the notation `DMB <domain>` and `DSB <domain>` respectively.

Except where otherwise stated, other conventions are:

- All memory initializes to 0.
- `R0` and `W0` contain the value 1.
- `R1` - `R4` and `W1` - `W4` contain arbitrary independent addresses that initialize to the same value on all PEs. The addresses held in these registers are shareable and:
  - The addresses held in `R1` and `R2` are in Write-Back Cacheable Normal memory.
  - The address held in `R3` is in Write-Through Cacheable Normal memory.
  - The address held in `R4` is in Non-cacheable Normal memory.
- `R5` - `R8` and `W5` - `W8` contain:
  - When used with an `STR` instruction, `0x55`, `0x66`, `0x77`, and `0x88` respectively.
  - When used with an `LDR` instruction, the value 0.
- `R11` and `W11` contain a new instruction or new translation table entry, as appropriate, and `R10` contains the virtual address and the ASID, for use in this change of translation table entry.
- Memory locations are Normal memory locations unless otherwise stated.

The examples use mnemonics for the cache maintenance and TLB maintenance instructions. The following tables describe the mnemonics:

- Cache maintenance system instructions on page K17-11905.
- TLB maintenance system instructions on page K17-11907.
K13.2 Load-Acquire, Store-Release and barriers


The following sections show that most of the examples in sections Simple ordering and barrier cases on page K13-11781 and Load-Exclusive, Store-Exclusive and barriers on page K13-11785 can be achieved using the Load-Acquire and Store-Release instructions without the need for additional barriers.

K13.2.1 Message passing

The following sections describe:

- Resolving weakly-ordered message passing by using Acquire and Release.
- Resolving message passing by the use of Store-Release and address dependency on page K13-11760.

Resolving weakly-ordered message passing by using Acquire and Release

The message passing problem described in Weakly-ordered message passing problem on page K13-11781 can be solved by the use of Load-Acquire and Store-Release instructions when accessing the communications flag:

**AArch32**

P1

```assembly
STR R5, [R1]          ; sets new data
STL R0, [R2]          ; sends flag indicating data ready, which is ordered after the STR
```

P2

```assembly
WAIT_ACQ([R2]==1)     ; waits on flag
LDR R5, [R1]
```

**AArch64**

P1

```assembly
STR W5, [X1]          ; sets new data
STLR W0, [X2]         ; sends flag indicating data ready, which is ordered after the STR
```

P2

```assembly
WAIT_ACQ([X2]==1)     ; waits on flag
LDR W5, [X1]
```

This ensures the observed order of both the reads and the writes allows transfer of data such that the result P2::R5==0x55 is guaranteed.

This approach also works with multiple observers, in a way that further observers use the same sequence as P2 uses:

**AArch32**

P3

```assembly
WAIT_ACQ([R2]==1)     ; waits on flag
LDR R5, [R1]
```

**AArch64**

P3

```assembly
WAIT_ACQ([X2]==1)     ; waits on flag
LDR W5, [X1]
```
Resolving message passing by the use of Store-Release and address dependency

The lack of ordering of stores discussed in Message passing with multiple observers on page K13-11782 can be resolved by the use of Store-Release for the store of the valid flag by P1, even when the observers are using an address dependency:

**AArch32**

P1

```assembly
STR R5, [R1] ; sets new data
STL R0, [R2] ; sends flag indicating data ready using a Store-Release
```

P2

```assembly
WAIT([R2]==1)
AND R12, R12, #0 ; R12 is the destination of LDR in the WAIT macro
LDR R5, [R1, R12] ; the load has an address dependency on R12
               ; and so is ordered after the flag has been seen
```

**AArch64**

P1

```assembly
STR W5, [X1] ; sets new data
STLR W0, [X2] ; sends flag indicating data ready using a Store-Release
```

P2

```assembly
WAIT([X2]==1)
AND W12, W12, WZR ; W12 is the destination of LDR in the WAIT macro
LDR W5, [X1, W12] ; the load has an address dependency on W12
               ; and so is ordered after the flag has been seen
```

This ensures the observed order of the writes allows transfer of data such that P2:R5 and P3:R5 contain the same value of 0x55.

This approach also works with multiple observers, in a way that further observers use the same sequence as P2 uses:

**AArch32**

P3

```assembly
WAIT([R2]==1)
AND R12, R12, #0 ; R12 is the destination of LDR in the WAIT macro
LDR R5, [R1, R12] ; the load has an address dependency on R12
               ; and so is ordered after the flag has been seen
```

**AArch64**

P3

```assembly
WAIT([X2]==1)
AND W12, W12, WZR ; R12 is the destination of LDR in the WAIT macro
LDR W5, [X1, W12] ; the load has an address dependency on W12
               ; and so is ordered after the flag has been seen
```
K13.2.2 Address dependency with object construction

When accessing an object-oriented data structure, the address dependency rule means that barriers are not required, even when initializing the object. A Store-Release can be used to ensure the order of the update of the base address:

AArch32

P1

```
STR R5, [R1, #offset] ; sets new data in a field
STL R1, [R2] ; updates base address
```

P2

```
LDR R1, [R2] ; reads base address
CMP R1, #0 ; checks if it is valid
BEQ null_trap
LDR R5, [R1, #offset] ; uses base address to read field
```

AArch64

P1

```
STR W5, [X1, #offset] ; sets new data in a field
STLR X1, [X2] ; updates base address
```

P2

```
LDR X1, [X2] ; reads base address
CMP X1, #0 ; check if it is valid
B.EQ null_trap
LDR W5, [X1, #offset] ; uses base address to read field
```

It is required that P2:R5==0x55 if the null_trap is not taken. This avoids P2 observing a partially constructed object from P1. Significantly, P2 does not need a barrier to ensure this behavior.

The read of the base address in P2 could be a Load-Acquire, but it is not necessary in this case.
K13.2.3  **WFE and WFI and barriers**

The Wait For Event and Wait For Interrupt instructions permit the PE to suspend execution and enter a low-power state. An explicit `DSB` barrier instruction is required if it is necessary to ensure memory accesses made before the `WFI` or `WFE` are visible to other observers, unless some other mechanism has ensured this visibility. Examples of other mechanisms that would guarantee the required visibility are the `DMB` described in *Posting a store before polling for acknowledgement on page K13-11784*, or a dependency on a load.

The following example requires the `DSB` to ensure that the store is visible:

**AArch32**

```
P1
  STR R0, [R2]
  DSB
Loop
  WFI
  B Loop
```

**AArch64**

```
P1
  STR W0, [X2]
  DSB <domain>
Loop
  WFI
  B Loop
```

This requirement is unchanged in Armv8 and later architectures by the presence of Load-Acquire or Store-Release.
K13.3 Load-Acquire Exclusive, Store-Release Exclusive and barriers

The Armv8 architecture adds the acquire and release semantics to Load-Exclusive and Store-Exclusive instructions, which allows them to gain ordering acquire and/or release semantics.

The Load-Exclusive instruction can be specified to have acquire semantics, and the Store-Exclusive instruction can be specified to have release semantics. These can be arbitrarily combined to allow the atomic update created by a successful Load-Exclusive and Store-Exclusive pair to have any of:

- No Ordering semantics (using LDREX and STREX).
- Acquire only semantics (using LDAEX and STREX).
- Release only semantics (using LDREX and STLEX).
- Sequentially consistent semantics (using LDAEX and STLEX).

In addition, the Armv8 and later specifications require that the clearing of a global monitor will generate an event for the PE associated with the global monitor, which can simplify the use of WFE, by removing the need for a DSB barrier and SEV instruction.

K13.3.1 Acquiring a lock

A common use of Load-Exclusive and Store-Exclusive instructions is to claim a lock to permit entry into a critical region. This is typically performed by testing a lock variable that indicates 0 for a free lock and some other value, commonly 1 or an identifier of the process holding the lock, for a taken lock.

Note

The inclusion of AArch32 PLDw instructions or AArch64 PRFM PST+ instructions in these examples is not a functional requirement, but will improve performance on many implementations. The performance benefit of adding these instructions will vary between different implementations of the architecture.

For a critical region, the requirement on taking a lock is usually for acquire semantics, while the clearing of a lock requires release semantics:

AArch32

Px

```
PLDw[R1]            ; preload into cache in unique state
Loop
  LDAEX R5, [R1]    ; read lock with acquire
  CMP R5, #0        ; check if 0
  STREXEQ R5, R0, [R1] ; attempt to store new value
  CMPEQ R5, #0      ; test if store succeeded
  BNE Loop          ; retry if not

; loads and stores in the critical region can now be performed
```

AArch64

Px

```
PRFM PSTLKKEEP, [X1] ; preload into cache in unique state
Loop
  LDAXR W5, [X1]     ; read lock with acquire
  CBNZ W5, Loop     ; check if 0
  STXR W5, W0, [X1] ; attempt to store new value
  CBNZ W5, Loop     ; test if store succeeded and retry if not

; loads and stores in the critical region can now be performed
```

The acquire associated with the load is sufficient to ensure the required ordering in a lock situation. The Store-Exclusive will fail (and so be retried) if there is a store to the location being monitored between the Load-Exclusive and the Store-Exclusive.
**K13.3.2 Releasing a lock**

The converse operation of releasing a lock does not require the use of Load-Exclusive and Store-Exclusive instructions, because only a single observer is able to write to the lock. However, often it is necessary for any observer to observe any memory updates, or any values that are loaded into memory, before they observe the release of the lock. Therefore, the lock release needs release semantics:

**AArch32**

```
Px

; loads and stores in the critical region
MOV R0, #0
STL R0, [R1] ; clear the lock with release semantics
```

**AArch64**

```
Px

; loads and stores in the critical region
STLR WZR, [X1] ; clear the lock with release semantics
```

**K13.3.3 Ticket locks**

When a lock is free, in order to avoid a rush to get the lock by many PEs, the use of ticket locks is common in more advanced systems. When the use is requested, the ticket locks determine the order of the users of the critical sections, in order to avoid starvation that can occur with a simple contention based spin lock.

A ticket lock allocates each thread a ticket number when it first requests the lock, and then compares that number with the current number for the lock. If they are the same, then the critical section can be entered. Otherwise the thread waits until the current number is equal to the ticket number for that thread.

The reading of the current number of the lock needs acquire semantics for the lock to be acquired.

--- Note ---

- The code in this section is little-endian code, as it views the combined current and next values as a single combined quantity. The addresses of the current and next ticket values need to be adjusted for a big-endian system.

- The inclusion of AArch32 `PLDW` instructions or AArch64 `PFRM PST` instructions in these examples is not a functional requirement, but will improve performance on many implementations. The performance benefit of adding these instructions will vary between different implementations of the architecture.

This is shown in the implementation below:

**AArch32**

```
Px

; R1 holds two 16 bit quantities
; the lower halfword holds the current ticket number
; the higher halfword holds the next ticket number
PLDW[R1] ; preload into cache in unique state
Loop1
LDAEX R5, [R1] ; read current and next
ADD R3, R5, #0x10000 ; increment the next number
STREX R6, R3, [R1] ; and update the value
CMP R6, #0 ; did the exclusive pass
BNE Loop1 ; retry if not
CMP R5, R5, ROR #16 ; is the current ticket ours
MOV R6, R5
BEQ block_start
Loop2
LDAH R6, [R1] ; read current value
```
CMP R6, R5, LSR #16 ; compare it with our allocated ticket
BNE Loop2 ; retry (spin) if it is not the same
block_start

AArch64

Px

; X1 holds 2 16 bit quantities
; the lower halfword holds the current ticket number
; the higher halfword holds the next ticket number

PRFM PSTLIKEEP, [X1] ; preload into cache in unique state
Loop1
LDAW R5, [X1] ; read current and next
ADD W5, W5, #0x10000 ; increment the next number
STXR W6, W5, [X1] ; and update the value
CBNZ W6, Loop1 ; did the exclusive pass - retry if not

AND W6, W5, #0xFFFF
CMP W6, W5, LSR #16 ; is the current ticket ours
B.EQ block_start

Loop2
LDAH R6, [X1] ; read current value
CMP W6, W5, LSR #16 ; compare it with the our allocated ticket
B.NE Loop2 ; retry (spin) if it isn't the same
block_start

Releasing the ticket lock simply involves incrementing the current ticket number, which is assumed in this example to be in R6, and doing a Store-Release:

AArch32

ADD R6, R6, #1
STLH R6, [R1]

AArch64

ADD W6, W6, #1
STLH W6, [X1]

K13.3.4 Use of Wait For Event (WFE) and Send Event (SEV) with locks

The Armv8 and later architectures can use the Wait For Event mechanism to minimize the energy cost of polling variables by putting the PE into a low power state, suspending execution, until an asynchronous exception or an explicit event is seen by that PE. In Armv8 and later architectures, the event can be generated as a result of clearing the global monitor, so removing the need for a DSB barrier or an explicit send event message.

This can be used with simple locks or with ticket locks.

--- Note ---

The inclusion of AArch32 PLDW instructions or AArch64 PRFM PSTL instructions in these examples is not a functional requirement, but will improve performance on many implementations. The performance benefit of adding these instructions will vary between different implementations of the architecture.

---

Simple lock

The following is an example of lock acquire code using WFE:

AArch32

Px

PLDW[R1] ; preload into cache in unique state
Barrier Litmus Tests

K13.3 Load-Acquire Exclusive, Store-Release Exclusive and barriers

Loop
LDAEX R5, [R1]        ; read lock with acquire
CMP R5, #0            ; check if 0
WFENE                 ; sleep if the lock is held
STREXEQ R5, R0, [R1]  ; attempt to store new value
CMPEQ R5, #0          ; test if store succeeded
BNE Loop              ; retry if not

AArch64

Px

SEVL                  ; invalidates the WFE on the first loop iteration
PRFM PSTL1KEEP, [X1]  ; allocate into cache in unique state
Loop
WFE
LDAXR W5, [X1]        ; read lock with acquire
CBNZ W5, Loop         ; check if 0
STXR W5, W0, [X1]     ; attempt to store new value
CBNZ W5, Loop         ; test if store succeeded and retry if not

; loads and stores in the critical region can now be performed

And the following is an example of lock release code:

AArch32

Px

; loads and stores in the critical region
MOV R0, #0
STL R0, [R1]          ; clear the lock

AArch64

Px

; loads and stores in the critical region
STLR WZR, [X1]        ; clear the lock

Ticket lock

In the Ticket lock case, the Load-Exclusive instruction can be used to move the monitor into the exclusive state for
the express purpose of creating an event when the monitor changes state:

AArch32

Px

; R1 holds 2 16 bit quantities
; the lower halfword holds the current ticket number
; the higher halfword holds the next ticket number
PLDW[R1]              ; preload into cache in unique state
Loop1
LDAEX R5, [R1]        ; read current and next
ADD R3, R5, #0x10000  ; increment the next number
STREX R6, R3, [R1]    ; and update the value
CMP R6, #0            ; did the exclusive pass
BNE Loop              ; retry if not
CMP R5, R5, ROR #16   ; is the current ticket ours
BEQ block_start
SEVL
Loop2
WFE                   ; wait if there has not been a change to the count since last
LDAEXH R6, [R1]       ; check the current count
CMP R6, R5, LSR #16   ; check if it is equal
BNE Loop2

block_start

AArch64

Px

; X1 holds 2 16 bit quantities
; the lower halfword holds the current ticket number
; the higher halfword holds the next ticket number

PRFM PSTL1KEEP, [X1]  ; preload into cache in unique state
Loop1
LDAXR W5, [X1]        ; read current and next
ADD W3, W5, #0x10000  ; increment the next number
STXR W6, W3, [X1]     ; and update the value
CBNZ W6, Loop1        ; did the exclusive pass – retry if not

AND W6, W5, 0xFFFF
CMP W6, W5, LSR #16   ; is the current ticket ours
B.EQ block_start

SEVL

Loop2
WFE
LDAXHR W6, [X1]       ; read current value
CMP W6, W5, LSR #16   ; compare it with our allocated ticket
B.NE Loop2            ; retry (spin) if it is not the same

block_start
K13.4 Using a mailbox to send an interrupt

In some message passing systems, it is common for one observer to update memory and then notify a second observer of the update by sending an interrupt, using a mailbox.

Although a memory access might be made to initiate the sending of the mailbox interrupt, a DSB instruction is required to ensure the completion of previous memory accesses.

Therefore, the following sequence is required to ensure that P2 observes the updated value:

**AArch32**

P1

```assembly
STR R5, [R1]          ; message stored to shared memory location
DSB ST
STR R0, [R4]          ; R4 contains the address of a mailbox
```

P2

```assembly
; interrupt service routine
LDR R5, [R1]
```

**AArch64**

P1

```assembly
STR W5, [X1]          ; message stored to shared memory location
DSB ST
STR W0, [X4]          ; R4 contains the address of a mailbox
```

P2

```assembly
; interrupt service routine
LDR W5, [X1]
```
K13.5 Cache and TLB maintenance instructions and barriers

The following sections describe the use of barriers with cache and TLB maintenance instructions:

- Data cache maintenance instructions.
- Instruction cache maintenance instructions on page K13-11773.
- TLB maintenance instructions and barriers on page K13-11776.

K13.5.1 Data cache maintenance instructions

The following sections describe the use of barriers with data cache maintenance instructions:

- Message passing to non-caching observers.
- Multiprocessing message passing to non-caching observers.
- Invalidating DMA buffers, non-functional example on page K13-11770.
- Invalidating DMA buffers, functional example with single PE on page K13-11771.
- Invalidating DMA buffers, functional example with multiple coherent PEs on page K13-11772.

Message passing to non-caching observers

The Armv8 and later architectures require the use of DMB instructions to ensure the ordering of data cache maintenance instructions and their effects. The Load-Acquire and Store-Release instructions have no effect on cache maintenance instruction. This means the following message passing approaches can be used when communicating between caching observers and non-caching observers:

**AArch32**

P1

```assembly
STR R5, [R1]          ; updates data (assumed to be in P1 cache)
DCCMVAC R1            ; cleans cache to point of coherency
DMB                   ; ensures effects of the clean will be observed before the flag is set
STR R0, [R4]          ; sends flag to external agent (Non-cacheable location)

E1

WAIT_ACQ ([R4] == 1)  ; waits for the flag (with order)
LDR R5, [R1]          ; reads the data
```

**AArch64**

P1

```assembly
STR W5, [X1]          ; updates data (assumed to be in P1 cache)
DC.CVAC, X1           ; cleans cache to point of coherency
DMB ISH               ; ensures effects of the clean will be observed before the flag is set
STR W0, [X4]          ; sends flag to external agent (Non-cacheable location)

E1

WAIT_ACQ ([X4] == 1)  ; waits for the flag (with order)
LDR W5, [X1]          ; reads the data
```

In this example, it is required that E1:R5 == 0x55.

Multiprocessing message passing to non-caching observers

The broadcast nature of the cache maintenance instructions combined with properties of barriers, means that the message passing principle for non-caching observers is:

**AArch32**

P1

```assembly
```
Barrier Litmus Tests
K13.5 Cache and TLB maintenance instructions and barriers

```assembly
STR R5, [R1]          ; updates data (assumed to be in P1 cache)
STL R0, [R2]          ; sends a flag for P2 (ordered by the store release)

P2
WAIT ([R2] == 1)      ; waits for P1 flag
DMB                   ; ensures cache clean is observed after P1 flag is observed
DCMVAC R1             ; cleans cache to point of coherency – will clean P1 cache
DMB                   ; ensures effects of the clean will be observed before the
                        ; flag to E1 is set
STR R0, [R4]          ; sends flag to E1

E1
WAIT_ACQ ([R4] == 1)  ; waits for P2 flag (ordered)
LDR R5, [R1]          ; reads data

AArch64

P1
STR W5, [X1]          ; updates data (assumed to be in P1 cache)
STLR W0, [X2]         ; sends a flag for P2 (ordered)

P2
WAIT ([X2] == 1)      ; waits for P1 flag
DMB SY                ; ensures cache clean is observed after P1 flag is observed
DCMVAC, X1            ; cleans cache to point of coherency, will clean P1 cache
DMB SY                ; ensures effects of the clean will be observed before the
                        ; flag to E1 is set
STR W0, [X4]          ; sends flag to E1

E1
WAIT_ACQ ([X4] == 1)  ; waits for P2 flag
LDR W5, [X1]          ; reads data

In this example, it is required that E1:R5==0x55. The clean operation executed by P2 affects the data location in the
P1 cache. The cast-out from the P1 cache is guaranteed to be observed before P2 updates [R4].

Note
The cache maintenance instructions are not ordered by the Load-Acquire and Store-Release instructions.

Invalidating DMA buffers, non-functional example

The basic scheme for communicating with an external observer that is a process that passes data in to a Cacheable
memory region must take account of the architectural requirement that regions with a Normal Cacheable attribute
can be allocated into a cache at any time, for example as a result of speculation. The following example shows this
possibility:

AArch32

P1

DCIMVAC R1            ; ensures cache is not dirty. A clean operation could be used
                        ; but as the DMA will subsequently overwrite this region an
                        ; invalidate operation is sufficient and usually more efficient
DMB                   ; ensures cache invalidation is observed before the next store
                        ; is observed
STR R0, [R3]          ; sends flag to external agent
WAIT_ACQ ([R4]==1)    ; waits for a different flag from an external agent
LDR R5, [R1]          ; reads data
```
WAIT ([R3] == 1) ; waits for flag
STR R5, [R1] ; stores new data
STL R0, [R4] ; sends a flag

AArch64

P1
DC IVAC, X1 ; ensure cache is not dirty. A clean operation could be used
; but as the DMA will subsequently overwrite this region an
DMB SY ; invalidate operation is sufficient and usually more efficient
; is observed
STR W0, [X3] ; sends flag to external agent
WAIT_ACQ ([X4]==1) ; waits for a different flag from an external agent
LDR W5, [X1]

E1
WAIT ([X3] == 1) ; waits for flag
STR W5, [X1] ; stores new data
STLR W0, [X4] ; sends a flag

If a speculative access occurs, there is no guarantee that the cache line containing [R1] is not brought back into
the cache after the cache invalidation, but before [R1] is written by E1. Therefore, the result P1:R5=0 is permissible.

Invalidating DMA buffers, functional example with single PE

AArch32

P1
DCIMVAC R1 ; ensures cache is not dirty. A clean operation could be used
; but as the DMA will subsequently overwrite this region an
DMB ; invalidate operation is sufficient and usually more efficient
; is observed
STR R0, [R3] ; sends flag to external agent
WAIT ([R4]==1) ; waits for a different flag from an external agent
DMB ; ensures that cache invalidate is observed after the flag
; from external agent is observed
DCIMVAC R1 ; ensures cache discards stale copies before use
LDR R5, [R1]

E1
WAIT ([R3] == 1) ; waits for flag
STR R5, [R1] ; stores new data
STL R0, [R4] ; sends a flag

AArch64

P1
DC IVAC, X1 ; ensures cache is not dirty. A clean operation could be used
; but as the DMA will subsequently overwrite this region an
DMB SY ; invalidate operation is sufficient and usually more efficient
; is observed
STR W0, [X3] ; sends flag to external agent
WAIT ([X4]==1) ; waits for a different flag from an external agent
DMB SY ; ensures that cache invalidate is observed after the flag
; from external agent is observed
DC IVAC, X1 ; ensures cache discards stale copies before use
LDR W5, [X1]

E1
WAIT ([X3] == 1) ; waits for flag
STR W5, [X1] ; stores new data
STLR W0, [X4] ; sends a flag

In this example, the result P1:R5 = 0x55 is required. Including a cache invalidation after the store by E1 to [R1] is observed ensures that the line is fetched from external memory after it has been updated.

Invalidating DMA buffers, functional example with multiple coherent PEs

The broadcasting of cache maintenance instructions, and the use of DMB instructions to ensure their observability, means that the previous example extends naturally to a multiprocessor system. Typically this requires a transfer of ownership of the region that the external observer is updating.

**AArch32**

**P0**

(Use data from [R1], potentially using [R1] as scratch space)

STL R0, [R2] ; signals release of [R1]
WAIT_ACQ ([R2] == 0) ; waits for new value from DMA
LDR R5, [R1]

**P1**

WAIT ([R2] == 1) ; waits for release of [R1] by P0
DCIMVAC R1 ; ensures caches are not dirty, an invalidate is sufficient
DMB
STR R0, [R3] ; requests new data for [R1]
WAIT ([R4] == 1) ; waits for new data
DMB
DCIMVAC R1 ; ensures caches discard stale copies before use
DMB
MOV R0, #0
STR R0, [R2] ; signals availability of new [R1]

**E1**

WAIT ([R3] == 1) ; waits for new data request
STR R5, [R1] ; sends new [R1]
DMB [ST]
STR R0, [R2] ; indicates that new data is available to P1

**AArch64**

**P0**

(Use data from [X1], potentially using [X1] as scratch space)

STLR W0, [X2] ; signals release of [X1]
WAIT_ACQ ([X2] == 0) ; waits for new value from DMA
LDR W5, [X1]

**P1**

WAIT ([X2] == 1) ; waits for release of [R1] by P0
DC IVAC, X1 ; ensures caches are not dirty, an invalidate is sufficient
DMB SY
STR W0, [X3] ; requests new data for [R1]
WAIT ([X4] == 1) ; waits for new data
DMB SY
DCIMVAC X1 ; ensures caches discard stale copies before use
DMB SY
STR WZR, [X2] ; signals availability of new [R1]

**E1**

WAIT ([X3] == 1) ; waits for new data request
STR W5, [X1] ; sends new [R1]
STR W0, [X4] ; indicates new data is available to P1
In this example, the result $P0:R5 = 0x55$ is required. The $DMB$ issued by $P1$ after the first data cache invalidation ensures that effect of the cache invalidation on $P0$ is seen by $E1$ before the store by $E1$ to $[R1]$. The $DMB$ issued by $P1$ after the second data cache invalidation ensures that its effects are seen before the store of 0 to the semaphore location in $[R2]$.

K13.5.2 Instruction cache maintenance instructions

The following sections describe the use of barriers with instruction cache maintenance instructions:

- Ensuring the visibility of updates to instructions for a uniprocessor.
- Ensuring the visibility of updates to instructions for a multiprocessor.

Ensuring the visibility of updates to instructions for a uniprocessor

On a single PE, the agent that causes instruction fetches, or instruction cache linefills, is a separate memory system observer from the agent that causes data accesses. Therefore, any operations to invalidate the instruction cache can rely only on seeing updates to memory that are complete. This must be ensured by the use of a $DSB$ instruction.

Also, instruction cache maintenance instructions are only guaranteed to complete after the execution of a $DSB$, and an $ISB$ is required to discard any instructions that might have been prefetched before the instruction cache invalidation completed. Therefore, on a uniprocessor, to ensure the visibility of an update to code and to branch to it, the following sequence is required:

**AArch32**

```
P1
  STR R11, [R1]         ; R11 contains a new instruction to be stored in program memory
  DCOMAU R1            ; clean to PoU makes the new instruction visible to the instruction cache
  DSB                  ; clean to PoU makes new instruction cache state visible
  ICIMVAU R1           ; ensures instruction cache/branch predictor discards stale data
  BPIMVA R1            ; ensures completion of the invalidation
  DSB                  ; ensures completion of the invalidation
  ISB                  ; ensures instruction fetch path sees new instruction cache state
  BX R1                ; branch to new instruction
```

In AArch64 state, the branch predictor maintenance is not required.

**AArch64**

```
P1
  STR W11, [X1]         ; W11 contains a new instruction to be stored in program memory
  DC CVAU, X1           ; clean to PoU makes the new instruction visible to instruction cache
  DSB ISH              ; clean to PoU makes new instruction cache state visible
  IC IVAU, X1           ; ensures instruction cache/branch predictor discards stale data
  DSB ISH              ; ensures completion of the invalidation
  ISB                  ; ensures instruction fetch path sees new instruction cache state
  BR X1                ; branch to new instruction
```

**Note**
Where the changes to the instructions span multiple cache lines, then the data cache and instruction cache maintenance instructions can be duplicated to cover each of the lines to be cleaned and to be invalidated.

Ensuring the visibility of updates to instructions for a multiprocessor

The Armv8 and later architectures require a PE that executes an instruction cache maintenance instruction to execute a $DSB$ instruction to ensure completion of the maintenance operation. This ensures that the cache maintenance instruction is complete on all PEs in the Inner Shareable shareability domain.

An $ISB$ is not broadcast, and so does not affect other PEs. This means that any other PE must perform its own $ISB$ synchronization after it knows that the update is visible, if it is necessary to ensure its synchronization with the update. The following example shows how this might be done:
**AArch32**

P1

```
STR R11, [R1]         ; R11 contains a new instruction to be stored in program memory
DCCMVAU R1            ; clean to PoU makes the new instruction visible to the instruction cache
DSB                   ; ensures completion of the clean on all PEs
ICIMVAU R1            ; ensures instruction cache discards stale data
BPIMVA R               ; ensures branch predictor discards stale data
DSB                   ; ensures completion of the instruction cache and branch predictor
          ; invalidation on all PEs
STR R0, [R2]          ; sets flag to signal completion
ISB                   ; synchronizes context on this PE
BX R1                 ; branches to new code
```

P2-Px

```
WAIT ([R2] == 1)      ; waits for flag signalling completion
ISB                   ; synchronizes context on this PE
BX R1                 ; branches to new code
```

**AArch64**

P1

```
STR X11, [X1]         ; X11 contains a new instruction to be stored in program memory
DC CVAU, X1           ; clean to PoU makes the new instruction visible to the instruction cache
DSB ISH               ; ensures completion of the clean on all PEs
IC IVAU, X1           ; ensures instruction cache/branch predictor discards stale data
DSB ISH               ; ensures completion of the instruction cache/branch predictor
          ; invalidation on all PEs
STR W0, [X2]          ; sets flag to signal completion
ISB                   ; synchronizes context on this PE
BR X1                 ; branches to new code
```

P2-Px

```
WAIT ([X2] == 1)      ; waits for flag signalling completion
ISB                   ; synchronizes context on this PE
BR X1                 ; branches to new code
```

**Nonfunctional approach**

The following sequence does not have the same effect, because a DSB is not required to complete the instruction cache maintenance instructions that other PEs issue:

**AArch32**

P1

```
STR R11, [R1]         ; R11 contains a new instruction to be stored in program memory
DCCMVAU R1            ; clean to PoU makes the new instruction visible to the instruction cache
DSB                   ; ensures completion of the clean on all PEs
ICIMVAU R1            ; ensures instruction cache discards stale data
BPIMVA R               ; ensures branch predictor discards stale data
DMB                   ; ensures ordering of the store after the invalidation
          ; DOES NOT guarantee completion of instruction cache/branch
          ; predictor on other PEs
STR R0, [R2]          ; sets flag to signal completion
DSB                   ; ensures completion of the invalidation on all PEs
ISB                   ; synchronizes context on this PE
BX R1                 ; branches to new code
```

P2-Px

```
WAIT ([R2] == 1)      ; waits for flag signalling completion
DSB                   ; this DSB does not guarantee completion of P1
           ; ICIMVAU/BPIMVA
ISB                   ; synchronizes context on this PE
BX R1                 ; branches to new code
```
AArch64

P1

```
STR W11, [X1]         ; W11 contains a new instruction to be stored in program memory
DC CVAU, X1           ; clean to PoU makes the new instruction visible to instruction cache
DSB ISH               ; ensures completion of the clean on all PEs
IC IVAU, X1           ; ensures instruction cache/branch predictor discards stale data
DMB ISH               ; ensures ordering of the store after the invalidation
; DOES NOT guarantee completion of instruction cache/branch
; predictor on other PEs
STR W0, [X2]          ; sets flag to signal completion
DSB ISH               ; ensures completion of the invalidation on all PEs
ISB                    ; synchronizes context on this PE
BR X1                  ; branches to new code
```

P2-Px

```
WAIT ([X2] == 1)      ; waits for flag signalling completion
DSB ISH               ; this DSB does not guarantee completion of P1
; ICIMVAU/BPIMVA
ISB                    
BR X1                  
```

In this example, P2…Px might not see the updated region of code at R1.
K13.5.3 TLB maintenance instructions and barriers

The following sections describe the use of barriers with TLB maintenance instructions:

- Ensuring the visibility of updates to translation tables for a uniprocessor.
- Ensuring the visibility of updates to translation tables for a multiprocessor.
- Paging memory in and out on page K13-11777.

Ensuring the visibility of updates to translation tables for a uniprocessor

On a single PE, the agent that causes translation table walks is a separate memory system observer from the agent that causes data accesses. Therefore, any operations to invalidate the TLB can only rely on seeing updates to memory that are complete. This must be ensured by the use of a DSB instruction.

The Armv8 and later architectures require that translation table walks look in the data or unified caches at L1, so such systems do not require data cache cleaning.

After the translation tables update, any old copies of entries that might be held in the TLBs must be invalidated. This operation is only guaranteed to affect all instructions, including instruction fetches and data accesses, after the execution of a DSB and an ISB. Therefore, the code for updating a translation table entry is:

AArch32

P1

```
STR R11, [R1]         ; updates the translation table entry
DSB                   ; ensures visibility of the update to translation table walks
TLBIMVA R10
BPIALL
DSB                   ; ensures completion of the BP and TLB invalidation
ISB                   ; synchronises context on this PE
; new translation table entry can be relied upon at this point and all accesses
; generated by this observer using
; the old mapping have been completed
```

AArch64

P1

```
STR X11, [X1]         ; updates the translation table entry
DSB ISH               ; ensures visibility of the update to translation table walks
TLBI VAE1, X10        ; assumes we are in the EL1
DSB ISH               ; ensures completion of the TLB invalidation
ISB                   ; synchronises context on this PE
; new translation table entry can be relied upon at this point and all accesses
; generated by this observer using
; the old mapping have been completed
```

Importantly, by the end of this sequence, all accesses that used the old translation table mappings have been observed by all observers.

An example of this is where a translation table entry is marked as invalid. Such a system must provide a mechanism to ensure that any access to a region of memory being marked as invalid has completed before any action is taken as a result of the region being marked as invalid.

Ensuring the visibility of updates to translation tables for a multiprocessor

The same code sequence can be used in a multiprocessing system. The Armv8 and later architectures require a PE that executes a TLB maintenance instruction to execute a DSB instruction to ensure completion of the maintenance operation. This ensures that the TLB maintenance instruction is complete on all PEs in the Inner Shareable shareability domain.

The completion of a DSB that completes a TLB maintenance instruction ensures that all accesses that used the old mapping have completed.
AArch32

P1

STR R11, [R1] ; updates the translation table entry
DSB ; ensures visibility of the update to translation table walks
TLBIMVAIS R10
BPIALLIS
DSB ; ensures completion of the BP and TLB invalidation
ISB ; Note ISB is not broadcast and must be executed locally
; on other PEs
; new translation table entry can be relied upon at this point and all accesses
; generated by any observers affected by the broadcast TLBIMVAIS operation using
; the old mapping have been completed

AArch64

P1

STR X11, [X1] ; updates the translation table entry
DSB ISH ; ensures visibility of the update to translation table walks
TLBI VAE1IS, X10
DSB ISH ; ensures completion of the TLB invalidation
ISB ; Note ISB is not broadcast and must be executed locally
; on other PEs
; new translation table entry can be relied upon at this point and all accesses
; generated by any observers affected by the broadcast TLBIMVAIS operation using
; the old mapping have been completed

The completion of the TLB maintenance instruction is guaranteed only by the execution of a `DSB` by the observer that performed the TLB maintenance instruction. The execution of a `DSB` by a different observer does not have this effect, even if the `DSB` is known to be executed after the TLB maintenance instruction is observed by that different observer.

Paging memory in and out

In a multiprocessor system there is a requirement to ensure the visibility of translation table updates when paging regions of memory into RAM from a backing store. This might, or might not, also involve paging existing locations in memory from RAM to a backing store. In such situations, the operating system selects one or more pages of memory that might be in use but are suitable to discard, with or without copying to a backing store, depending on whether or not the region of memory is writable. Disabling the translation table mappings for a page, and ensuring the visibility of that update to the translation tables, prevents agents accessing the page.

For this reason, it is important that the `DSB` that is performed after the TLB invalidation ensures that no other updates to memory using those mappings are possible.

An example sequence for the paging out of an updated region of memory, and the subsequent paging in of memory, is as follows:

AArch32

P1

STR R11, [R1] ; updates the translation table for the region being paged out
DSB ; ensures visibility of the update to translation table walks
TLBIMVAIS R10 ; invalidates the old entry
DSB ; ensures completion of the invalidation on all PEs
ISB ; ensures visibility of the invalidation
BL SaveMemoryPageToBackingStore
BL LoadMemoryFromBackingStore
DSB ; ensures completion of the memory transfer (this could be part of
; LoadMemoryFromBackingStore)
ICIALLUIS ; also invalidates the branch predictor
DSB ; ensures completion of the instruction cache
; and branch predictor invalidation
STR R9, [R1] ; creates a new translation table entry with a new mapping
DSB ; ensures visibility of the new translation table mapping
Barrier Litmus Tests

K13.5 Cache and TLB maintenance instructions and barriers

ISB ; ensures synchronisation of this instruction stream

AArch64

P1

STR X11, [X1] ; updates the translation table for the region being paged out
DSB ISH ; ensures visibility of the update to translation table walks
TLBI VAE1IS, X10 ; invalidates the old entry
DSB ISH ; ensures completion of the invalidation on all PEs
ISB ; ensures visibility of the invalidation
BL SaveMemoryPageToBackingStore
BL LoadMemoryFromBackingStore
DSB ISH ; ensures completion of the memory transfer (this could be part of
 ; LoadMemoryFromBackingStore)
IC IALLUIS ; also invalidates the branch predictor
DSB ISH ; ensures completion of the instruction cache
 ; and branch predictor invalidation
STR X9, [X1] ; creates a new translation table entry with a new mapping
DSB ISH ; ensures visibility of the new translation table mapping
ISB ; ensures synchronisation of this instruction stream

This example assumes the memory copies are performed by an observer that is coherent with the caches of PE P1. This observer might be P1 itself, using a specific paging mapping. For clarity, the example omits the functional descriptions of SaveMemoryPageToBackingStore and LoadMemoryFromBackingStore. LoadMemoryFromBackingStore is required to ensure that the memory updates that it makes are visible to instruction fetches.

In this example, the use of ICIALLUIS in AArch32 state and IC IALLUIS in AArch64 state to invalidate the entire instruction cache is a simplification that might not be optimal for performance. An alternative approach involves invalidating all of the lines in the caches using ICIMVAU in AArch32 state and IC IVAU operations in AArch64 state. This invalidation must be done when the mapping used for the ICIMVAU and IC IVAU operations is valid but not executable.

Using break-before-make when updating translation table entries

The Arm Architecture requires that reads to the same location are observed in order, and since application level software relies on this behavior, the operating system needs to maintain this illusion when it is changing a virtual to physical address mapping for a location, as is the case with copy on write or other memory management techniques. This illusion can be maintained provided that the software uses a break-before-make sequence when updating translation table entries whenever multiple threads of execution can use the same translation tables and the change to the translation entries involves any of:

- Changing the memory type.
- Changing the cacheability attributes
- Changing the output address (OA), if the OA of at least one of the old translation table entry and the new translation table entry is writable.

The architecture requires use of a break-before make sequence in these situations, see Using break-before-make when updating translation table entries on page D8-5198 for more information. However, if software did not use a break-before-make approach, an implementation might give a result that would occur if the two reads to the same virtual address did not occur in program order. An example of such an occurrence would be an implementation of copy-on-write, where one PE is performing two reads to the same virtual address at the same time as a second PE, running code associated with the operating system, is copying the data from one physical location that is mapped to by that virtual address, where the page was mapped as read-only, to a different physical location which will be mapped as read/write.

If the operating system changed the address mapping without going through an invalid entry, then it would be possible for a third PE to perform a write to the location that would be seen by the first load by the first PE, and not seen by the second load by the same PE.

The required break-before-make code sequence in this case is:

AArch32
**K13.5 Cache and TLB maintenance instructions and barriers**

**P1**

; R1, R2 contain an invalid translation table entry (that is, one with bit[0] == 0)
; R3 contains the address of the translation table entry
; R4 contains the Virtual Address and ASID of the VA being remapped
; R5, R6 contain the new valid translation table entry

STR R1, R2, [R3]          ; stores invalid entry
DSB ISH                    ; ensures visibility of the update to translation table walks
TLBIMVAIS R4               ; invalidates the old entry
DSB ISH                    ; ensures completion of the invalidation on all PEs
ICIALLUIS                  ; also invalidates the branch predictor
STR R5, R6, [R3]           ; store new mapping
DSB ISH                    ; ensures visibility of the update to translation table walks
ISB                        ; ensures synchronisation of this instruction stream

--- Note ---

This example shows an update to an entry in a translation table that is using the long-descriptor format.

**AArch64**

**P1**

; X1 contains an invalid translation table entry (that is, one with bit[0] == 0)
; X2 contains the address of the translation table entry
; X3 contains the Virtual Address and ASID of the VA being remapped
; X4 contains the new valid translation table entry

STR X1, [X2]              ; stores invalid entry
DSB ISH                   ; ensures visibility of the update to translation table walks
TLB IVAEIS, X3            ; invalidates the old entry
DSB ISH                   ; ensures completion of the invalidation on all PEs
IC IALLUIS                ; also invalidates the branch predictor
STR X4, [X2]              ; store new mapping
DSB ISH                   ; ensures visibility of the update to translation table walks
ISB                       ; ensures synchronisation of this instruction stream

If this sequence is correctly followed, then the architecture guarantees that the loads to a virtual address being remapped will be seen in the correct order.

The instruction cache maintenance is only required if the mapping from input address to output address has been changed as part of the change of the translation table entries, and the memory being moved is executable. In this example, the use of ICIALLUIS in AArch32 state and IC IALLUIS in AArch64 state to invalidate the entire instruction cache is a simplification that might not be optimal for performance. An alternative approach involves invalidating all of the lines in the caches using ICIMVAU in AArch32 state, and IC IVAU in AArch64 state. This invalidation must be done when the mapping used for the ICIMVAU and IC IVAU operations is valid but not executable.

**K13.5.4 Ordering of Memory-mapped device control with payloads**

With a Memory-mapped peripheral, such as a DMA, which can also access memory for its own use, it is common to have control or status registers which are Memory-mapped. These registers need to be accessed in an ordered manner with respect to the data that the Memory-mapped peripheral is handling.

Two simple examples of this are:

- When a processing element is writing a buffer of data, and then writing to a control register in the DMA peripheral to start that peripheral to access the buffer of data.

- When a DMA peripheral has written to a buffer of data in memory, and the processing element is reading a status register to determine that the DMA transfer has completed, and then is reading the data.

For the case of the processing element writing a buffer of data, before starting the DMA peripheral, the ordering requirements between the stores to the data buffer and the stores to the Memory-mapped a to the DMA peripheral can be met by the insertion of a DSB <domain> instruction between these sets of accesses as this ensures the global observation of the stores before the DMA is started. This is shown by the following code:

**AArch32**
P1

```
STR R5, [R2]          ; data written to the data buffer
DSB
STR R0, [R4]          ; R4 contains the address of the DMA control register
```

**AArch64**

P1

```
STR W5, [X2]          ; data written to the data buffer
DSB <domain>
STR W0, [X4]          ; X4 contains the address of the DMA control register
```

For the case of DMA peripheral writing the data buffer and then setting a status register when those stores are complete (and so globally observed) and then having this status register polled by the processing element before the processing element reads the data buffer, the processing element must insert a DSB <domain> between the load that reads the status register, and the read of the buffer. A DMB, or load-acquire, is not sufficient as this problem is not solely concerned with observation order, since the polling read is actually a read of a status register at a Completer, not the polling a data value that has been written by an observer.

For this case, the code is therefore:

**AArch32**

P1

```
WAIT ([R4] == 1)      ; R4 contains the address of the status register,
; and the value '1' indicates completion of the DMA transfer
DSB
LDR R5, [R2]          ; reads data from the data buffer
```

**AArch64**

P1

```
WAIT ([X4] == 1)      ; X4 contains the address of the status register,
; and the value '1' indicates completion of the DMA transfer
DSB <domain>
LDR W5, [X2]          ; reads data from the data buffer
```
K13.6 **Armv7 compatible approaches for ordering, using DMB and DSB barriers**

The following sections describe the Armv7 compatible approaches for ordering, using DMB and DSB barriers:

- Simple ordering and barrier cases.
- Load-Exclusive, Store-Exclusive and barriers on page K13-11785.
- Using a mailbox to send an interrupt on page K13-11787.
- Cache and TLB maintenance instructions and barriers on page K13-11787.

**K13.6.1 Simple ordering and barrier cases**

Arm implements a weakly consistent memory model for Normal memory. In general terms, this means that the order of memory accesses observed by other observers might not be the order that appears in the program, for either loads or stores.

This section includes examples of this.

**Simple weakly consistent ordering example**

P1

```
STR R5, [R1]
LDR R6, [R2]
```

P2

```
STR R6, [R2]
LDR R5, [R1]
```

In the absence of barriers, the result of P1: R6=0, P2: R5=0 is permissible.

**Message passing**

The following sections describe:

- Weakly-ordered message passing problem.
- Message passing with multiple observers on page K13-11782.

**Weakly-ordered message passing problem**

P1

```
STR R5, [R1] ; sets new data
STR R0, [R2] ; sends flag indicating data ready
```

P2

```
WAIT([R2]==1) ; waits on flag
LDR R5, [R1] ; reads new data
```

In the absence of barriers, an end result of P2: R5=0 is permissible.

**Resolving by the addition of barriers**

The addition of barriers, to ensure the observed order of the reads and the writes, ensures that data is transferred so that the result P2: R5==0x55 is guaranteed, as follows:

P1

```
STR R5, [R1] ; sets new data
DMB [ST] ; ensures all observers observe data before the flag
STR R0, [R2] ; sends flag indicating data ready
```

P2

```
WAIT([R2]==1) ; waits on flag
```
DMB                      ; ensures that the load of data is after the flag has been observed
LDR R5, [R1]

**Resolving by the use of barriers and address dependency**

There is a rule within the Arm architecture that:

- Where the value returned by a read is used for computation of the virtual address of a subsequent read or write, then these two memory accesses are observed in program order.

Where the value returned by a read is used for computation of the virtual address of a subsequent read or write, this is called an **address dependency**. An address dependency exists even if the value returned by the first read has no effect on the virtual address. This might occur if the value returned is masked off before it is used, or if it confirms a predicted address value that it might have changed.

This restriction applies only when the data value returned by a read is used as a data value to calculate the address of a subsequent read or write. It does not apply if the data value returned by a read determines the condition flags values, and the values of the flags are used for condition code evaluation to determine the address of a subsequent read, either through conditional execution or the evaluation of a branch. This is called a **control dependency**.

Where both a control and address dependency exist, the ordering behavior is consistent with the address dependency.

Table K13-1 on page K13-11782 shows examples of address dependencies, control dependencies, and an address and control dependency.

**Table K13-1 Dependency examples**

<table>
<thead>
<tr>
<th>Address dependency</th>
<th>Control dependency</th>
<th>Address and control dependency^a</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR r1, [r0]</td>
<td>LDR r1, [r0]</td>
<td>LDR r1, [r0]</td>
</tr>
<tr>
<td>LDR r2, [r1]</td>
<td>AND r1, r1, #0</td>
<td>CMP r1, #55</td>
</tr>
<tr>
<td>LDR r2, [r3, r1]</td>
<td>LDRNE r2, [r3]</td>
<td>MOVNE r4, #22</td>
</tr>
<tr>
<td></td>
<td>LDR r2, [r3, r4]</td>
<td></td>
</tr>
</tbody>
</table>

[a. The address dependency takes priority.]

This means that the data transfer example of *Weakly-ordered message passing problem* on page K13-11781 can also be satisfied as shown in the following example:

**P1**

```
STR R5, [R1]          ; sets new data
DMB [ST]              ; ensures all observers observe data before the flag
STR R0, [R2]          ; sends flag indicating data ready
```

**P2**

```
WAIT([R2]==1)
AND R12, R12, #0      ; R12 is destination of LDR in WAIT macro
LDR R5, [R1, R12]     ; the load has an address dependency on R12
                      ; and so is ordered after the flag has been seen
```

The load of R5 by P2 is ordered with respect to the load from [R2] because there is an address dependency using R12. P1 uses a DMB to ensure that P2 does not observe the write of [R2] before the write of [R1].

**Message passing with multiple observers**

Where the ordering of Normal memory accesses is not resolved by the use of barriers or dependencies, then different observers might observe the accesses in a different order, as shown in the following example:
Barrier Litmus Tests

K13.6 Armv7 compatible approaches for ordering, using DMB and DSB barriers

P1

STR R5, [R1] ; sets new data
STR R0, [R2] ; sends flag indicating data ready

P2

WAIT([R2]==1)
AND R12, R12, #0 ; R12 is destination of LDR in WAIT macro
LDR R5, [R1, R12] ; the load has an address dependency on R12
; and so is ordered after the flag has been seen

P3

WAIT([R2]==1)
AND R12, R12, #0 ; R12 is destination of LDR in WAIT macro
LDR R5, [R1, R12] ; the load has an address dependency on R12
; and so is ordered after the flag has been seen

In this case, it is permissible for P2:R5 and P3:R5 to contain different values, because there is no order guaranteed between the two stores performed by P1.

Resolving by the addition of barriers

The addition of a barrier by P1, as shown in the following example, ensures the observed order of the writes, transferring data so that P2:R5 and P3:R5 both contain the value 0x55:

P1

STR R5, [R1] ; sets new data
DMB [ST] ; ensures all observers observe data before the flag
STR R0, [R2] ; sends flag indicating data ready

P2

WAIT([R2]==1)
AND R12, R12, #0 ; R12 is destination of LDR in WAIT macro
LDR R5, [R1, R12] ; the load has an address dependency on R12
; and so is ordered after the flag has been seen

P3

WAIT([R2]==1)
AND R12, R12, #0 ; R12 is destination of LDR in WAIT macro
LDR R5, [R1, R12] ; the load has an address dependency on R12
; and so is ordered after the flag has been seen

Address dependency with object construction

When accessing an object-oriented data structure, the address dependency rule means that barriers are not required, even when initializing the object:

P1

STR R5, [R1, #offset] ; sets new data in a field
DMB [ST] ; ensures all observers observe data before base address is updated
STR R1, [R2] ; updates base address

P2

LDR R1, [R2] ; reads for base address
CMP R1, #0 ; checks if it is valid
BEQ null_trap
LDR R5, [R1, #offset] ; uses base address to read field

If the null_trap is not taken, it is required that P2:R5==0x55. This avoids P2 observing a partially constructed object from P1. Significantly, P2 does not require a barrier to ensure this behavior.
P1 requires a barrier to ensure the observed order of the writes by P1. In general, the impact of requiring a barrier during the construction phase is much less than the impact of requiring a barrier for every read access.

**Posting a store before polling for acknowledgement**

In the case where an observer stores to a location, and then polls for an acknowledge from a different observer, the weak ordering of the memory model can lead to a deadlock, as the following example shows:

```plaintext
P1
  STR R0, [R2]
  WAIT ([R3]==1)

P2
  WAIT ([R2]==1)
  STR R0, [R3]
```

In Armv7 implementations that do not include the Multiprocessing Extensions, then this can deadlock because P2 might not observe the store by P1 in finite time. For Armv7 implementations with the Multiprocessing Extensions, for Armv8, and for later architectures, this is not an issue as all stores must be observed by all observers within their shareability domain in finite time.

The addition of a `DMB` instruction prevents this deadlock in Armv7 implementations that do not include the Multiprocessing Extensions:

```plaintext
P1
  STR R0, [R2]
  DMB
  WAIT ([R3]==1)

P2
  WAIT ([R2]==1)
  STR R0, [R3]
```

The `DMB` executed by P1 ensures that P2 observes the store by P1 before it observes the load by P1. This ensures a timely completion.

The following example is a variant of the previous example, where the two observers poll the same memory location:

```plaintext
P1
  STR R0, [R2]
  WAIT ([R2]==2)

P2
  WAIT ([R2]==1)
  LDR R0, [R2]
  ADD R0, R0, #1
  STR R0, [R2]
```

In this example, the same deadlock can occur in Armv7 implementations that do not include the Multiprocessing Extensions, because the architecture permits P1 to read the result of its own store to [R2] early, and continue doing so for an indefinite amount of time. The addition of a `DMB` instruction prevents this deadlock:

```plaintext
P1
  STR R0, [R2]
  DMB
  WAIT ([R2]==2)

P2
  WAIT ([R2]==1)
  LDR R0, [R2]
```
ADD R0, R0, #1
STR R0, [R2]

**WFE and WFI and barriers**

The Wait For Event and Wait For Interrupt instructions permit the PE to suspend execution and enter a low-power state. A DSB barrier instruction is required if it is necessary to ensure that memory accesses made before the WFI or WFE are visible to other observers, unless some other mechanism has ensured this visibility. Examples of other mechanism that would guarantee the required visibility are the DMB described in *Posting a store before polling for acknowledgement* on page K13-11784, or a dependency on a load.

The following example requires the DSB to ensure that the store is visible:

P1

```
STR R0, [R2]
DSB
Loop
  WFI
  B Loop
```

However, if the example in *Posting a store before polling for acknowledgement* on page K13-11784 is extended to include a WFE, there is no risk of a deadlock. The extended example is:

P1

```
STR R0, [R2]
DMB
Loop
  LDR R12, [R3]
  CMP R12, #1
  WFENE
  BNE Loop
```

P2

```
WAIT ([R2]==1)
STR R0, [R3]
DSB
SEV
```

In this example:

- The DMB by P1 ensures that P2 observes the store by P1 before it observes the load by P1.
- The dependency of the WFE on the result of the load by P1 means that this load must complete before P1 executes the WFE.

For more information about SEV, see *Use of Wait For Event (WFE) and Send Event (SEV) with locks* on page K13-11786.

**K13.6.2 Load-Exclusive, Store-Exclusive and barriers**

The Load-Exclusive and Store-Exclusive instructions, described in *Synchronization and semaphores* on page B2-211, are predictable only with Normal memory. These instructions do not have any implicit barrier functionality. Therefore, any use of these instructions to implement locks of any type requires the addition of explicit barriers.

**Acquiring a lock**

A common use of Load-Exclusive and Store-Exclusive instructions is to claim a lock to permit entry into a critical region. This is typically performed by testing a lock variable that indicates 0 for a free lock and some other value, commonly 1 or an identifier of the process holding the lock, for a taken lock.
Barrier Litmus Tests
K13.6 Armv7 compatible approaches for ordering, using DMB and DSB barriers

The lack of implicit barriers in the Load-Exclusive and Store-Exclusive instructions means that the mechanism requires a DMB instruction between acquiring a lock and making the first access to the critical region, to ensure that all observers observe the successful claim of the lock before they observe any subsequent loads or stores to the region. This example shows Px acquiring a lock:

Px

Loop
LDREX R5, [R1] ; reads lock
CMP R5, #0 ; checks if 0
STREXEQ R5, R0, [R1] ; attempts to store new value
CMPEQ R5, #0 ; tests if store succeeded
BNE Loop ; retries if not
DMB ; ensures that all subsequent accesses are observed after the gaining of the lock is observed
; loads and stores in the critical region can now be performed

Releasing a lock

The converse operation of releasing a lock does not require the use of Load-Exclusive and Store-Exclusive instructions, because only a single observer is able to write to the lock. However, often it is necessary for any observer to observe any memory updates, or any values that are loaded into memory, before they observe the release of the lock. Therefore, a DMB usually precedes the lock release, as the following example shows.

Px

; loads and stores in the critical region
MOV R0, #0
DMB ; ensures all previous accesses are observed before the lock is cleared
STR R0, [R1] ; clears the lock

Use of Wait For Event (WFE) and Send Event (SEV) with locks

The Armv8 and later architectures include Wait For Event and Send Event instructions, that can be executed to reduce the required number of iterations of a lock-acquire loop, or spinlock, to reduce power. The basic mechanism involves an observer that is in a spinlock executing a WFE instruction that suspends execution on that observer until an asynchronous exception or an explicit event, sent by some other observer using the SEV instruction, is seen by the suspended observer. An observer that holds the lock executes an SEV instruction to send an event after it has released the lock.

The Event signal is a non-memory communication, and therefore the memory update that releases the lock must be observable by all observers before the SEV instruction is executed and the event is sent. This requires the use of DSB instruction, rather than DMB.

Therefore, the following is an example of lock acquire code using WFE:

Px

Loop
LDREX R5, [R1] ; reads lock
CMP R5, #0 ; checks if 0
WFE ; sleeps if the lock is held
STREXEQ R5, R0, [R1] ; attempts to store new value
CMPEQ R5, #0 ; tests if store succeeded
BNE Loop ; retries if not
DMB ; ensures that all subsequent accesses are observed after the gaining of the lock is observed
; loads and stores in the critical region can now be performed

And the following is an example of lock release code using SEV:

Px

; loads and stores in the critical region
MOV R0, #0
DMB ; ensures all previous accesses are observed before the lock is cleared
K13.6.3 Using a mailbox to send an interrupt

In some message passing systems, it is common for one observer to update memory and then notify a second observer of the update by sending an interrupt, using a mailbox.

Although a memory access might be made to initiate the sending of the mailbox interrupt, a `DSB` instruction is required to ensure the completion of previous memory accesses.

Therefore, the following sequence is required to ensure that P2 observes the updated value:

```
P1
  STR R5, [R1]            ; message stored to shared memory location
  DSB [ST]                ; R4 contains the address of a mailbox
  STR R1, [R4]            ; ensures completion of the store that cleared the lock before
                         ; sending the event
P2
  ; interrupt service routine
  LDR R5, [R1]
```

**Note**
The `DSB` executed by P1 ensures global observation of the store to [R1]. The interrupt timing ensures that the code executed by P2 is executed after the global observation of the update to [R1], and therefore must see this update. In some implementations, this might be implemented by requiring that interrupts flush non-coherent buffers that hold speculatively loaded data.

K13.6.4 Cache and TLB maintenance instructions and barriers

The following sections describe the use of barriers with cache and TLB maintenance instructions:

- **Data cache maintenance instructions** on page K13-11790.
- **Instruction cache maintenance instructions** on page K13-11791.
- **TLB maintenance instructions and barriers** on page K13-11791.

**Data cache maintenance instructions**

The following sections describe the use of barriers with data cache maintenance instructions:

- **Message passing to non-caching observers** on page K13-11788.
- **Multiprocessing message passing to non-caching observers** on page K13-11788.
- **Invalidating DMA buffers, non-functional example** on page K13-11788.
- **Invalidating DMA buffers, functional example with single PE** on page K13-11789.
- **Invalidating DMA buffers, functional example with multiple coherent PEs** on page K13-11789.

**Message passing to non-caching observers**

The Armv8 and later architectures require the use of `DMB` instructions to ensure the ordering of data cache maintenance instructions and their effects. This means the following message passing approaches can be used when communicating between caching observers and non-caching observers:

```
P1
  STR R5, [R1]               ; updates data (assumed to be in P1's cache)
  DCOMVAC R1                 ; cleans cache to point of coherency
  DMB                        ; ensures effects of the clean will be observed before the flag is set
  STR R0, [R4]               ; sends flag to external agent (Non-cacheable location)
```
### Multiprocessing message passing to non-caching observers

The broadcast nature of the cache maintenance instructions in Armv8 and later architectures, and in Armv7 implementations that include the Multiprocessing Extensions, combined with properties of barriers, means that the message passing principle for non-caching observers is:

**P1**

- `STR R5, [R1]` ; updates data (assumed to be in P1's cache)
- `DMB [ST]` ; ensures new data is observed before the flag to P2 is set
- `STR R0, [R2]` ; sends flag to P2

**P2**

- `WAIT ([R2] == 1)` ; waits for flag from P1
- `DMB` ; ensures cache clean is observed after P1 flag is observed
- `DCMVAC R1` ; cleans cache to point of coherency - this cleans the cache of P1
- `DMB` ; ensures effects of the clean are observed before the flag to E1 is set
- `STR R0, [R4]` ; sends flag to E1

**E1**

- `WAIT ([R4] == 1)` ; waits for flag from P2
- `DMB` ; ensures that flag has been observed before reading the data
- `LDR R5, [R1]` ; reads the data

In this example, it is required that E1:R5==0x55. The clean operation executed by P2 affects the data location in the P1 cache. The cast-out from the P1 cache is guaranteed to be observed before P2 updates [R4].

### Invalidating DMA buffers, non-functional example

The basic scheme for communicating with an external observer that is a process that passes data in to a Cacheable memory region must take account of the architectural requirement that regions with a Normal Cacheable attribute can be allocated into a cache at any time, for example as a result of speculation. The following example shows this possibility:

**P1**

- `DCMVAC R1` ; ensures caches are not dirty. A clean operation could be used but the DMA overwrites this region so an invalidate operation is sufficient and usually more efficient
- `DMB` ; ensures cache invalidation is observed before the next store is observed
- `STR R0, [R3]` ; sends flag to external agent
- `WAIT ([R4]==1)` ; waits for a different flag from an external agent
- `DMB` ; observes flag from external agent before reading new data. However [R1] could have been brought into cache earlier
- `LDR R5, [R1]`

**E1**

- `WAIT ([R3] == 1)` ; waits for flag
- `STR R5, [R1]` ; stores new data
- `DMB` ; sends a flag

If a speculative access occurs, there is no guarantee that the cache line containing [R1] is not brought back into the cache after the cache invalidation, but before [R1] is written by E1. Therefore, the result P1:R5=0 is permissible.
Invalidate DMA buffers, functional example with single PE

P1

DCIMVAC R1  ; ensures cache is not dirty. A clean operation could be
             ; used but the DMA overwrites this region so an invalidate operation
             ; is sufficient and usually more efficient
DMB         ; ensures cache invalidation is observed before the next store is observed
STR R0, [R3] ; sends flag to external agent
WAIT ([R4]==1) ; waits for a different flag from an external agent
DMB         ; ensures that cache invalidate is observed after the flag
            ; from external agent is observed
DCIMVAC R1  ; ensures cache discards stale copies before use
LDR R5, [R1]

E1

WAIT ([R3] == 1) ; waits for flag
STR R5, [R1]  ; stores new data
DMB [ST]      ; sends a flag

In this example, the result P1:R5 == 0x55 is required. Including a cache invalidation after the store by E1 to [R1] is observed ensures that the line is fetched from external memory after it has been updated.

Invalidate DMA buffers, functional example with multiple coherent PEs

The broadcasting of cache maintenance instructions, and the use of DMB instructions to ensure their observability, means that the previous example extends naturally to a multiprocessor system. Typically this requires a transfer of ownership of the region that the external observer is updating.

P0

(Use data from [R1], potentially using [R1] as scratch space)
DMB
STR R0, [R2]  ; signals release of [R1]
WAIT ([R2] == 0) ; waits for new value from DMA
DMB
LDR R5, [R1]

P1

WAIT ([R2] == 1) ; waits for release of [R1] by P0
DCIMVAC R1  ; ensures caches are not dirty, invalidate is sufficient
DMB
STR R0, [R3] ; requests new data for [R1]
WAIT ([R4] == 1) ; waits for new data
DMB
DCIMVAC R1  ; ensures caches discard stale copies before use
DMB
MOV R0, #0
STR R0, [R2]  ; signals availability of new [R1]

E1

WAIT ([R3] == 1) ; waits for new data request
STR R5, [R1]  ; sends new [R1]
DMB [ST]      ; indicates new data available to P1
STR R0, [R4]  ; sends flag

In this example, the result P0:R5 == 0x55 is required. The DMB issued by P1 after the first data cache invalidation ensures that effect of the cache invalidation on P0 is seen by E1 before the store by E1 to [R1]. The DMB issued by P1 after the second data cache invalidation ensures that its effects are seen before the store of 0 to the semaphore location in [R2].
Instruction cache maintenance instructions

The following sections describe the use of barriers with instruction cache maintenance instructions:

- **Ensuring the visibility of updates to instructions for a uniprocessor.**
- **Ensuring the visibility of updates to instructions for a multiprocessor.**

### Ensuring the visibility of updates to instructions for a uniprocessor

On a single PE, the agent that causes instruction fetches, or instruction cache linefills, is a separate memory system observer from the agent that causes data accesses. Therefore, any operations to invalidate the instruction cache can rely only on seeing updates to memory that are complete. This must be ensured by the use of a `DSB` instruction.

Also, instruction cache maintenance instructions are only guaranteed to complete after the execution of a `DSB`, and an `ISB` is required to discard any instructions that might have been prefetched before the instruction cache invalidation completed. Therefore, on a uniprocessor, to ensure the visibility of an update to code and to branch to it, the following sequence is required:

```
P1
  STR R11, [R1]            ; R11 contains a new instruction to store in program memory
  DCCMVAU R1               ; clean to PoU makes new instructions visible to instruction cache
  DSB                      ; ensures completion of the invalidation
  ICIMVAU R1               ; ensures instruction cache and branch predictor discard stale data
  BPIMVA R1
  DSB                      ; ensures completion of the instruction cache/branch predictor discard stale data
  ISB                      ; ensures instruction fetch path observes new instruction cache state
  BX R1
```

### Ensuring the visibility of updates to instructions for a multiprocessor

Armv8 and later architectures, and an Armv7 implementation that includes the Multiprocessing Extensions, require a PE that executes an instruction cache maintenance instruction to execute a `DSB` instruction to ensure completion of the maintenance operation. This ensures that the cache maintenance instruction is complete on all PEs in the Inner Shareable shareability domain.

An `ISB` is not broadcast, and so does not affect other PEs. This means that any other PE must perform its own `ISB` synchronization after it knows that the update is visible, if it is necessary to ensure its synchronization with the update. The following example shows how this might be done:

```
P1
  STR R11, [R1]            ; R11 contains a new instruction to store in program memory
  DCCMVAU R1               ; clean to PoU makes new instructions visible to instruction cache
  DSB                      ; ensures completion of the clean on all processors
  ICIMVAU R1               ; ensures instruction cache/branch predictor discards stale data
  BPIMVA R1
  DSB                      ; ensures completion of the instruction cache and branch predictor invalidation on all PEs
  STR R0, [R2]             ; sets flag to signal completion
  ISB                      ; synchronizes context on this PE
  BX R1
```

```
P2-Px
  WAIT ([R2] == 1)         ; waits for flag signaling completion
  ISB                      ; synchronizes context on this processor
  BX R1
```

### Nonfunctional approach

The following sequence does not have the same effect, because a `DSB` is not required to complete the instruction cache maintenance instructions that other PEs issue:

```
P1
  STR R11, [R1]            ; R11 contains a new instruction to store in program memory
  DCCMVAU R1               ; clean to PoU makes new instructions visible to instruction cache
```
Barrier Litmus Tests

K13.6 Armv7 compatible approaches for ordering, using DMB and DSB barriers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSB</td>
<td>; ensure completion of the clean on all PEs</td>
</tr>
<tr>
<td>ICIMVAU R1</td>
<td>; ensure instruction cache/branch predictor discards stale data</td>
</tr>
<tr>
<td>BPIMVA R1</td>
<td>; ensure ordering of the store after the invalidation</td>
</tr>
<tr>
<td>DMB</td>
<td>; DOES NOT guarantee completion of instruction cache/branch</td>
</tr>
<tr>
<td>STR R0, [R2]</td>
<td>; sets flag to signal completion</td>
</tr>
<tr>
<td>DSB</td>
<td>; ensures completion of the invalidation on all PEs</td>
</tr>
<tr>
<td>ISB</td>
<td>; synchronizes context on this PE</td>
</tr>
<tr>
<td>BX R1</td>
<td>; branches to new code</td>
</tr>
</tbody>
</table>

P2-Px

WAIT ([R2] == 1) ; waits for flag signaling completion
DSB ; this DSB does not guarantee completion of P1's ICIMVAU/BPIMVA
ISB
BX R1

In this example, P2...Px might not see the updated region of code at R1.

TLB maintenance instructions and barriers

The following sections describe the use of barriers with TLB maintenance instructions:

- **Ensuring the visibility of updates to translation tables for a uniprocessor**.
- **Ensuring the visibility of updates to translation tables for a multiprocessor** on page K13-11792.
- **Paging memory in and out** on page K13-11792.

### Ensuring the visibility of updates to translation tables for a uniprocessor

On a single PE, the agent that causes translation table walks is a separate memory system observer from the agent that causes data accesses. Therefore, any operations to invalidate the TLB can only rely on seeing updates to memory that are complete. This must be ensured by the use of a DSB instruction.

In the Armv8 and later architectures, and in an Armv7 implementation that includes the Multiprocessing Extensions, translation table walks must look in the data or unified caches at L1, so such systems do not require data cache cleaning.

After the translation tables update, any old copies of entries that might be held in the TLBs must be invalidated. This operation is only guaranteed to affect all instructions, including instruction fetches and data accesses, after the execution of a DSB and an ISB. Therefore, the code for updating a translation table entry is:

P1

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR R11, [R1]</td>
<td>; updates the translation table entry</td>
</tr>
<tr>
<td>DSB</td>
<td>; ensures visibility of the update to translation table walks</td>
</tr>
<tr>
<td>TLBIMVA R10</td>
<td></td>
</tr>
<tr>
<td>BPIMVA</td>
<td></td>
</tr>
<tr>
<td>DSB</td>
<td>; ensures completion of the BP and TLB invalidation</td>
</tr>
<tr>
<td>ISB</td>
<td>; synchronizes context on this PE</td>
</tr>
<tr>
<td>BX R1</td>
<td>; new translation table entry can be relied upon at this point and all accesses</td>
</tr>
<tr>
<td></td>
<td>; generated by this observer using the old mapping have been completed</td>
</tr>
</tbody>
</table>

Importantly, by the end of this sequence, all accesses that used the old translation table mappings have been observed by all observers.

An example of this is where a translation table entry is marked as invalid. Such a system must provide a mechanism to ensure that any access to a region of memory being marked as invalid has completed before any action is taken as a result of the region being marked as invalid.
Ensuring the visibility of updates to translation tables for a multiprocessor

The same code sequence can be used in a multiprocessing system. In the Armv8 and later architectures, and in an Armv7 implementation that includes the Multiprocessing Extensions, a PE that executes a TLB maintenance instruction must execute a DSB instruction to ensure completion of the maintenance operation. This ensures that the TLB maintenance instruction is complete on all PEs in the Inner Shareable shareability domain.

The completion of a DSB that completes a TLB maintenance instruction ensures that all accesses that used the old mapping have completed.

P1

```
STR R11, [R1] ; updates the translation table entry
DSB ; ensures visibility of the update to translation table walks
TLBIMVAIS R10
BPIALLIS
DSB ; ensures completion of the BP and TLB invalidation
ISB ; Note ISB is not broadcast and must be executed locally on other PEs
; new translation table entry can be relied upon at this point and all accesses generated by any
; observers affected by the broadcast TLBIMVAIS operation using the old mapping have completed
```

The completion of the TLB maintenance instruction is guaranteed only by the execution of a DSB by the observer that performed the TLB maintenance instruction. The execution of a DSB by a different observer does not have this effect, even if the DSB is known to be executed after the TLB maintenance instruction is observed by that different observer.

Paging memory in and out

In a multiprocessor system there is a requirement to ensure the visibility of translation table updates when paging regions of memory into RAM from a backing store. This might, or might not, also involve paging existing locations in memory from RAM to a backing store. In such situations, the operating system selects one or more pages of memory that might be in use but are suitable to discard, with or without copying to a backing store, depending on whether or not the region of memory is writable. Disabling the translation table mappings for a page, and ensuring the visibility of that update to the translation tables, prevents agents accessing the page.

For this reason, it is important that the DSB that is performed after the TLB invalidation ensures that no other updates to memory using those mappings are possible.

An example sequence for the paging out of an updated region of memory, and the subsequent paging in of memory, is as follows:

P1

```
STR R11, [R1] ; updates the translation table for the region being paged out
DSB ; ensures visibility of the update to translation table walks
TLBIMVAIS R10
; invalidates the old entry
DSB ; ensures completion of the invalidation on all processors
ISB ; ensures visibility of the invalidation
BL SaveMemoryPageToBackingStore
BL LoadMemoryFromBackingStore
DSB ; ensures completion of the memory transfer (this could be part of
; LoadMemoryFromBackingStore
ICIALLUIS ; also invalidates the branch predictor
DSB ; ensures completion of the instruction cache
; and branch predictor invalidation
STR R9, [R1] ; creates a new translation table entry with a new mapping
DSB ; ensures visibility of the new translation table mapping
ISB ; ensures synchronization of this instruction stream
```

This example assumes the memory copies are performed by an observer that is coherent with the caches of PE P1. This observer might be P1 itself, using a specific paging mapping. For clarity, the example omits the functional descriptions of SaveMemoryPageToBackingStore and LoadMemoryFromBackingStore. LoadMemoryFromBackingStore is required to ensure that the memory updates that it makes are visible to instruction fetches.
In this example, the use of ICIALLUIS to invalidate the entire instruction cache is a simplification that might not be optimal for performance. An alternative approach involves invalidating all of the lines in the caches using ICIMVAU operations. This invalidation must be done when the mapping used for the ICIMVAU operations is valid but not executable.
Appendix K14
Random Number Generation

This appendix provides further information on the generation of random numbers using FEAT_RNG. It contains the following sections:

- Properties of the generated random number on page K14-11796.
Properties of the generated random number

When FEAT_RNG is implemented, reads of the RNDR and RNDRRS registers return a 64-bit random number. The random numbers must meet the properties and conform to the standards that are detailed in this section.

The output random number is from a Deterministic Random Bit Generator (DRBG), which is seeded from a True Random Number Generator (TRNG).

The TRNG provides entropy in the form of random numbers, from the sampled output of an unpredictable physical process.

The TRNG should conform to:
- The NIST SP800-90B standard.
- The NIST SP800-22 standard.
- The FIPS 140-2 standard.
- The BSI AIS-31 standard.

The DRBG produces random numbers from a cryptographically secure algorithm.

The DRBG is seeded from the TRNG.

The DRBG algorithm should conform to the NIST SP800-90A Rev 1 standard.

The DRBG is reseeded after an IMPLEMENTATION DEFINED number of random numbers has been generated and read using the RNDR register.

The DRBG is reseeded immediately before the random number is generated and read using the RNDRRS register.

The entire random number generation should conform to the NIST SP800-90C standard.

Note

- Since a TRNG can only generate random bits at a limited rate, the random number bits are commonly collected in an “entropy pool” until needed. An implementation should ensure that lower privileged software cannot impact the performance of higher privileged software by entirely draining this “entropy pool”. The refill time cost of the “entropy pool” should be paid for by the persistent caller.

- When FEAT_RNG_TRAP is implemented, reads of the RNDR and RNDRRS registers may be trapped to EL3. For more information about this trapping behavior, see control fields ID_AA64PFR1_EL1.RNDR_trap and SCR_EL3.TRNDR.
Appendix K15
Legacy Feature Naming Convention

This appendix maps the legacy feature names for the Armv8.x extensions. It contains the following sections:

- *The Armv8.0 architecture* on page K15-11798.
- *The Armv8.1 architecture extension* on page K15-11799.
- *The Armv8.2 architecture extension* on page K15-11800.
- *The Armv8.3 architecture extension* on page K15-11802.
- *The Armv8.4 architecture extension* on page K15-11803.
- *The Armv8.5 architecture extension* on page K15-11804.
- *The Armv8.6 architecture extension* on page K15-11805.
### K15.1 The Armv8.0 architecture

Table K15-1 on page K15-11798 provides details of the mapping of the legacy names of Armv8.0 features to their current names.

<table>
<thead>
<tr>
<th>Feature name</th>
<th>Legacy feature name</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEAT_AES</td>
<td>ARMv8.0-AES</td>
<td>Advanced SIMD AES instructions</td>
</tr>
<tr>
<td>FEAT_PMULL</td>
<td>ARMv8.0-CP15SDISABLE2</td>
<td>CP15SDISABLE2</td>
</tr>
<tr>
<td>FEAT_CSV2</td>
<td>ARMv8.0-CSV2</td>
<td>Cache Speculation Variant 2</td>
</tr>
<tr>
<td>FEAT_CSV3</td>
<td>ARMv8.0-CSV3</td>
<td>Cache Speculation Variant 3</td>
</tr>
<tr>
<td>FEAT_DGH</td>
<td>ARMv8.0-DGH</td>
<td>Data Gathering Hint</td>
</tr>
<tr>
<td>FEAT_DoubleLock</td>
<td>ARMv8.0-DoubleLock</td>
<td>Double Lock</td>
</tr>
<tr>
<td>FEAT_ETS</td>
<td>ARMv8.0-ETS</td>
<td>Enhanced Translation Synchronization</td>
</tr>
<tr>
<td>FEAT_PCSRv8</td>
<td>ARMv8.0-PCSample</td>
<td>PC Sample-based Profiling Extension</td>
</tr>
<tr>
<td>FEAT_PMUv3</td>
<td>PMUv3</td>
<td>PMU Extensions</td>
</tr>
<tr>
<td>FEAT_RAS</td>
<td>RAS</td>
<td>The Reliability, Availability, and Serviceability Extension</td>
</tr>
<tr>
<td>FEAT_SB</td>
<td>ARMv8.0-SB</td>
<td>Speculation Barrier</td>
</tr>
<tr>
<td>FEAT_SHA1</td>
<td>ARMv8.0-SHA</td>
<td>Advanced SIMD SHA1 instructions</td>
</tr>
<tr>
<td>FEAT_SPECRES</td>
<td>ARMv8.0-PredInv</td>
<td>Speculation restriction instructions</td>
</tr>
<tr>
<td>FEAT_SSBS</td>
<td>ARMv8.0-SSBS</td>
<td>Speculative Store Bypass Safe</td>
</tr>
</tbody>
</table>
K15.2  The Armv8.1 architecture extension

Table K15-2 on page K15-11799 provides details of the mapping of the legacy names of Armv8.1 features to their current names.

<table>
<thead>
<tr>
<th>Feature name</th>
<th>Legacy feature name</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEAT_HAFDBS</td>
<td>ARMv8.1-TTHM</td>
<td>Hardware management of the Access flag and dirty state</td>
</tr>
<tr>
<td>FEAT_HPDS</td>
<td>ARMv8.1-HPD</td>
<td>Hierarchical permission disables</td>
</tr>
<tr>
<td>FEAT_LOR</td>
<td>ARMv8.1-LOR</td>
<td>Limited ordering regions</td>
</tr>
<tr>
<td>FEAT_LSE</td>
<td>ARMv8.1-LSE</td>
<td>Large System Extensions</td>
</tr>
<tr>
<td>FEAT_PAN</td>
<td>ARMv8.1-PAN</td>
<td>Privileged access never</td>
</tr>
<tr>
<td>FEAT_PMUv3p1</td>
<td>ARMv8.1-PMU</td>
<td>PMU Extensions v3.1</td>
</tr>
<tr>
<td>FEAT_RDM</td>
<td>ARMv8.1-RDMA</td>
<td>Advanced SIMD rounding double multiply accumulate instructions</td>
</tr>
<tr>
<td>FEAT_VHE</td>
<td>ARMv8.1-VHE</td>
<td>Virtualization Host Extensions</td>
</tr>
<tr>
<td>FEAT_VMID16</td>
<td>ARMv8.1-VMID16</td>
<td>16-bit VMID</td>
</tr>
</tbody>
</table>
K15.3 **The Armv8.2 architecture extension**

Table K15-3 on page K15-11800 provides details of the mapping of the legacy names of Armv8.2 features to their current names.

<table>
<thead>
<tr>
<th>Feature name</th>
<th>Legacy feature name</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEAT_AA32BF16</td>
<td>ARMv8.2-AA32BF16</td>
<td>AArch32 BFloat16 instructions</td>
</tr>
<tr>
<td>FEAT_AA32HPD</td>
<td>ARMv8.2-AA32HPD</td>
<td>AArch32 hierarchical permission disables</td>
</tr>
<tr>
<td>FEAT_AA32I8MM</td>
<td>ARMv8.2-AA32I8MM</td>
<td>AArch32 Int8 matrix multiplication instructions</td>
</tr>
<tr>
<td>FEAT_ASMv8p2</td>
<td>ARMv8.2-A64ISA</td>
<td>Armv8.2 changes to the A64 ISA</td>
</tr>
<tr>
<td>FEAT_BF16</td>
<td>ARMv8.2-BF16</td>
<td>AArch64 BFloat16 instructions</td>
</tr>
<tr>
<td>FEAT_Debugv8p2</td>
<td>ARMv8.2-Debug</td>
<td>Debug v8.2</td>
</tr>
<tr>
<td>FEAT_DotProd</td>
<td>ARMv8.2-DotProd</td>
<td>Advanced SIMD dot product instructions</td>
</tr>
<tr>
<td>FEAT_DPBP</td>
<td>ARMv8.2-DCCVADP</td>
<td>DC CVAP instruction</td>
</tr>
<tr>
<td>FEAT_DPBP2</td>
<td>ARMv8.2-DCCVADP</td>
<td>DC CVADP instruction</td>
</tr>
<tr>
<td>FEAT_EVT</td>
<td>ARMv8.2-EVT</td>
<td>Enhanced Virtualization Traps</td>
</tr>
<tr>
<td>FEAT_FHM</td>
<td>ARMv8.2-FHM</td>
<td>Floating-point half-precision multiplication instructions</td>
</tr>
<tr>
<td>FEAT_FP16</td>
<td>ARMv8.2-FP16</td>
<td>Half-precision floating-point data processing</td>
</tr>
<tr>
<td>FEAT_HPDS2</td>
<td>ARMv8.2-TTPBHA</td>
<td>Translation table page-based hardware attributes</td>
</tr>
<tr>
<td>FEAT_IESB</td>
<td>ARMv8.2-IESB</td>
<td>Implicit Error Synchronization event</td>
</tr>
<tr>
<td>FEAT_LPA</td>
<td>ARMv8.2-LPA</td>
<td>Large PA and IPA support</td>
</tr>
<tr>
<td>FEAT_LSMAOC</td>
<td>ARMv8.2-LSMAOC</td>
<td>AArch32 Load/Store Multiple instruction atomicity and ordering controls</td>
</tr>
<tr>
<td>FEAT_LVA</td>
<td>ARMv8.2-LVA</td>
<td>Large VA support</td>
</tr>
<tr>
<td>FEAT_PAN2</td>
<td>ARMv8.2-ATS1E1</td>
<td>AT S1E1R and AT S1E1W instruction variants affected by PSTATE.PAN</td>
</tr>
<tr>
<td>FEAT_PCSRv8p2</td>
<td>ARMv8.2-PCSSample</td>
<td>PC Sample-based profiling</td>
</tr>
<tr>
<td>FEAT_SHA256</td>
<td>ARMv8.0-SHA</td>
<td>Advanced SIMD SHA256 instructions</td>
</tr>
<tr>
<td>FEAT_SHA3</td>
<td>ARMv8.2-SHA</td>
<td>Advanced SIMD SHA3 instructions</td>
</tr>
<tr>
<td>FEAT_SHA512</td>
<td>ARMv8.2-SHA</td>
<td>Advanced SIMD SHA512 instructions</td>
</tr>
<tr>
<td>FEAT_SM3</td>
<td>ARMv8.2-SM</td>
<td>Advanced SIMD SM3 instructions</td>
</tr>
<tr>
<td>FEAT_SM4</td>
<td>ARMv8.2-SM</td>
<td>Advanced SIMD SM4 instructions</td>
</tr>
<tr>
<td>FEAT_SPE</td>
<td>SPE</td>
<td>The Statistical Profiling Extension (SPE)</td>
</tr>
<tr>
<td>FEAT_SVE</td>
<td>SVE</td>
<td>The Scalable Vector Extension (SVE)</td>
</tr>
<tr>
<td>FEAT_TTCNP</td>
<td>ARMv8.2-TTCNP</td>
<td>Translation table Common not private translations</td>
</tr>
</tbody>
</table>
Table K15-3 Mapping of legacy names of Armv8.2 features to current names (continued)

<table>
<thead>
<tr>
<th>Feature name</th>
<th>Legacy feature name</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEAT_UAO</td>
<td>ARMv8.2-UAO</td>
<td>Unprivileged Access Override control</td>
</tr>
<tr>
<td>FEAT_VPIPT</td>
<td>ARMv8.2-VPIPT</td>
<td>VMID-aware PIPT instruction cache</td>
</tr>
<tr>
<td>FEAT_XNX</td>
<td>ARMv8.2-TTS2UXN</td>
<td>Translation table stage 2 Unprivileged Execute-never</td>
</tr>
</tbody>
</table>
### K15.4 The Armv8.3 architecture extension

Table K15-4 on page K15-11802 provides details of the mapping of the legacy names of Armv8.3 features to their current names.

<table>
<thead>
<tr>
<th>Feature name</th>
<th>Legacy feature name</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEAT_CCIDX</td>
<td>ARMv8.3-CCIDX</td>
<td>Extended cache index</td>
</tr>
<tr>
<td>FEAT_DoPD</td>
<td>ARMv8.3-DoPD</td>
<td>Debug over Powerdown</td>
</tr>
<tr>
<td>FEAT_FCMA</td>
<td>ARMv8.3-CompNum</td>
<td>Floating-point complex number instructions</td>
</tr>
<tr>
<td>FEAT_JSCVT</td>
<td>ARMv8.3-JSconv</td>
<td>JavaScript conversion instructions</td>
</tr>
<tr>
<td>FEAT_LRCPC</td>
<td>ARMv8.3-RCpc</td>
<td>Load-acquire RCpc instructions</td>
</tr>
<tr>
<td>FEAT_NV</td>
<td>ARMv8.3-NV</td>
<td>Nested virtualization support</td>
</tr>
<tr>
<td>FEAT_PAuth</td>
<td>ARMv8.3-PAuth</td>
<td>Pointer authentication</td>
</tr>
<tr>
<td>FEAT_SPEv1p1</td>
<td>ARMv8.3-SPE</td>
<td>Armv8.3 Statistical Profiling Extensions</td>
</tr>
</tbody>
</table>
# K15.5 The Armv8.4 architecture extension

Table K15-5 on page K15-11803 provides details of the mapping of the legacy names of Armv8.4 features to their current names.

<table>
<thead>
<tr>
<th>Feature name</th>
<th>Legacy feature name</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEAT_AMUv1</td>
<td>AMUv1</td>
<td>Activity Monitors Extensions v1</td>
</tr>
<tr>
<td>FEAT_BBM</td>
<td>ARMv8.4-TTRem</td>
<td>Translation table break-before-make levels</td>
</tr>
<tr>
<td>FEAT_CNTSC</td>
<td>ARMv8.4-CNTSC</td>
<td>Generic Counter Scaling</td>
</tr>
<tr>
<td>FEAT_Debugv8p4</td>
<td>ARMv8.4-Debug</td>
<td>Debug v8.4</td>
</tr>
<tr>
<td>FEAT_DIT</td>
<td>ARMv8.4-DIT</td>
<td>Data Independent Timing instructions</td>
</tr>
<tr>
<td>FEAT_DoubleFault</td>
<td>ARMv8.4-DFE</td>
<td>Double Fault Extension</td>
</tr>
<tr>
<td>FEAT_FlagM</td>
<td>ARMv8.4-CondM</td>
<td>Flag manipulation instructions v2</td>
</tr>
<tr>
<td>FEAT_IDST</td>
<td>ARMv8.4-IDST</td>
<td>ID space trap handling</td>
</tr>
<tr>
<td>FEAT_LRCPC2</td>
<td>ARMv8.4-RCpc</td>
<td>Load-acquire RCpc instructions v2</td>
</tr>
<tr>
<td>FEAT_LSE2</td>
<td>ARMv8.4-LSE</td>
<td>Large System Extensions v2</td>
</tr>
<tr>
<td>FEAT_MPAM</td>
<td>MPAM</td>
<td>The Memory Partitioning and Monitoring (MPAM) Extension</td>
</tr>
<tr>
<td>FEAT_NV2</td>
<td>ARMv8.4-NV</td>
<td>Enhanced nested virtualization support</td>
</tr>
<tr>
<td>FEAT_PMUv3p4</td>
<td>ARMv8.4-PMU</td>
<td>PMU Extensions v3.4</td>
</tr>
<tr>
<td>FEAT_RASv1p1</td>
<td>ARMv8.4-RAS</td>
<td>RAS Extension v1.1</td>
</tr>
<tr>
<td>FEAT_S2FWB</td>
<td>ARMv8.4-S2FWB</td>
<td>Stage 2 forced Write-Back</td>
</tr>
<tr>
<td>FEAT_SEL2</td>
<td>ARMv8.4-SecEL2</td>
<td>Secure EL2</td>
</tr>
<tr>
<td>FEAT_TLBIOs</td>
<td>ARMv8.4-TLBI</td>
<td>TLB invalidate instructions in Outer Shareable domain</td>
</tr>
<tr>
<td>FEAT_TLBIRANGE</td>
<td>ARMv8.4-TLBI</td>
<td>TLB invalidate range instructions</td>
</tr>
<tr>
<td>FEAT_TRF</td>
<td>ARMv8.4-Trace</td>
<td>Self-hosted Trace Extensions</td>
</tr>
<tr>
<td>FEAT_TTL</td>
<td>ARMv8.4-TTL</td>
<td>Translation Table Level</td>
</tr>
<tr>
<td>FEAT_TTST</td>
<td>ARMv8.4-TTST</td>
<td>Small translation tables</td>
</tr>
</tbody>
</table>
K15.6 The Armv8.5 architecture extension

Table K15-6 on page K15-11804 provides details of the mapping of the legacy names of Armv8.5 features to their current names.

<table>
<thead>
<tr>
<th>Feature name</th>
<th>Legacy feature name</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEAT_BTI</td>
<td>ARMv8.5-BTI</td>
<td>Branch Target Identification</td>
</tr>
<tr>
<td>FEAT_E0PD</td>
<td>ARMv8.5-E0PD</td>
<td>Preventing EL0 access to halves of address maps</td>
</tr>
<tr>
<td>FEAT_ExS</td>
<td>ARMv8.5-CSEH</td>
<td>Context synchronization and exception handling</td>
</tr>
<tr>
<td>FEAT_FlagM2</td>
<td>ARMv8.5-CondM</td>
<td>Enhancements to flag manipulation instructions</td>
</tr>
<tr>
<td>FEAT_FRINTTS</td>
<td>ARMv8.5-FRINT</td>
<td>Floating-point to integer instructions</td>
</tr>
<tr>
<td>FEAT_GTG</td>
<td>ARMv8.5-GTG</td>
<td>Guest translation granule size</td>
</tr>
<tr>
<td>FEAT_MTE</td>
<td>ARMv8.5-MemTag</td>
<td>Memory Tagging Extension</td>
</tr>
<tr>
<td>FEAT_MTE2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FEAT_PMUv3p5</td>
<td>ARMv8.5-PMU</td>
<td>PMU Extensions v3.5</td>
</tr>
<tr>
<td>FEAT_RNG</td>
<td>ARMv8.5-RNG</td>
<td>Random number generator</td>
</tr>
</tbody>
</table>
## K15.7 The Armv8.6 architecture extension

Table K15-7 on page K15-11805 provides details of the mapping of the legacy names of Armv8.6 features to their current names.

### Table K15-7 Mapping of legacy names of Armv8.6 features to current names

<table>
<thead>
<tr>
<th>Feature name</th>
<th>Legacy feature name</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEAT_AMUv1p1</td>
<td>ARMv8.6-AMU</td>
<td>AMU Extensions v1.1</td>
</tr>
<tr>
<td>FEAT_ECV</td>
<td>ARMv8.6-ECV</td>
<td>Enhanced Counter Virtualization</td>
</tr>
<tr>
<td>FEAT_FGT</td>
<td>ARMv8.6-FGT</td>
<td>Fine Grain Traps</td>
</tr>
<tr>
<td>FEAT_FPAC</td>
<td>ARMv8.3-FPAC</td>
<td>Faulting on AUT* instructions</td>
</tr>
<tr>
<td>FEAT_MPAMv0p1</td>
<td>ARMv8.6-MPAM</td>
<td>Memory Partitioning and Monitoring Extension v0.1</td>
</tr>
<tr>
<td>FEAT_MPAMv1p1</td>
<td>ARMv8.6-MPAM</td>
<td>Memory Partitioning and Monitoring Extension v1.1</td>
</tr>
<tr>
<td>FEAT_MTPMU</td>
<td>ARMv8.6-MTPMU</td>
<td>Multi-threaded PMU Extensions</td>
</tr>
<tr>
<td>FEAT_PAuth2</td>
<td>ARMv8.3-PAuth2</td>
<td>Enhancements to pointer authentication</td>
</tr>
<tr>
<td>FEAT_TWED</td>
<td>ARMv8.6-TWED</td>
<td>Delayed Trapping of WFE</td>
</tr>
</tbody>
</table>
Appendix K16
Arm Pseudocode Definition

This appendix provides a definition of the pseudocode that is used in this manual, and defines some helper procedures and functions that are used by pseudocode. It contains the following sections:

• About the Arm pseudocode on page K16-11808.
• Pseudocode for instruction descriptions on page K16-11809.
• Data types on page K16-11811.
• Operators on page K16-11816.
• Statements and control structures on page K16-11822.
• Built-in functions on page K16-11827.
• Miscellaneous helper procedures and functions on page K16-11830.
• Arm pseudocode definition index on page K16-11832.

--- Note ---
This appendix is not a formal language definition for the pseudocode. It is a guide to help understand the use of Arm pseudocode. This appendix is not complete. Changes are planned for future releases.
K16.1 About the Arm pseudocode

The Arm pseudocode provides precise descriptions of some areas of the Arm architecture. This includes description of the decoding and operation of all valid instructions. Pseudocode for instruction descriptions on page K16-11809 gives general information about this instruction pseudocode, including its limitations.

The following sections describe the Arm pseudocode in detail:

• Data types on page K16-11811.
• Operators on page K16-11816.
• Statements and control structures on page K16-11822.

Built-in functions on page K16-11827 and Miscellaneous helper procedures and functions on page K16-11830 describe some built-in functions and pseudocode helper functions that are used by the pseudocode functions that are described elsewhere in this manual. Arm pseudocode definition index on page K16-11832 contains the indexes to the pseudocode.

K16.1.1 General limitations of Arm pseudocode

The pseudocode statements IMPLEMENTATION_DEFINED, SEE, UNDEFINED, and UNPREDICTABLE indicate behavior that differs from that indicated by the pseudocode being executed. If one of them is encountered:

• Earlier behavior indicated by the pseudocode is only specified as occurring to the extent required to determine that the statement is executed.
• No subsequent behavior indicated by the pseudocode occurs.

For more information, see Special statements on page K16-11826.
K16.2 Pseudocode for instruction descriptions

Each instruction description includes pseudocode that provides a precise description of what the instruction does, subject to the limitations described in General limitations of Arm pseudocode on page K16-11808 and Limitations of the instruction pseudocode on page K16-11810.

In the instruction pseudocode, instruction fields are referred to by the names shown in the encoding diagram for the instruction. Instruction encoding diagrams and instruction pseudocode gives more information about the pseudocode provided for each instruction.

K16.2.1 Instruction encoding diagrams and instruction pseudocode

Instruction descriptions in this manual contain:

- An Encoding section, containing one or more encoding diagrams, each followed by some encoding-specific pseudocode that translates the fields of the encoding into inputs for the common pseudocode of the instruction, and picks out any encoding-specific special cases.

- An Operation section, containing common pseudocode that applies to all of the encodings being described. The Operation section pseudocode contains a call to the EncodingSpecificOperations() function, either at its start or only after a condition code check performed by if ConditionPassed() then.

An encoding diagram specifies each bit of the instruction as one of the following:

- An obligatory 0 or 1, represented in the diagram as 0 or 1. If this bit does not have this value, the encoding corresponds to a different instruction.

- A should be 0 or 1, represented in the diagram as (0) or (1). If this bit does not have this value, the instruction is CONSTRAINED UNPREDICTABLE. For more information, see SBZ or SBO fields T32 and A32 in instructions on page K1-11566.

- A named single bit or a bit in a named multi-bit field. The cond field in bits[31:28] of many A32/T32 instructions has some special rules associated with it.

An encoding diagram matches an instruction if all obligatory bits are identical in the encoding diagram and the instruction, and one of the following is true:

- The encoding diagram is not for an A32/T32 instruction.

- The encoding diagram is for an A32/T32 instruction that does not have a cond field in bits[31:28].

- The encoding diagram is for an A32/T32 instruction that has a cond field in bits[31:28], and bits[31:28] of the instruction are not 0b1111.

In the context of the instruction pseudocode, the execution model for an instruction is:

1. Find all encoding diagrams that match the instruction. It is possible that no encoding diagram matches. In that case, abandon this execution model and consult the relevant instruction set chapter instead to find out how the instruction is to be treated. The bit pattern of such an instruction is usually reserved and UNDEFINED, though there are some other possibilities. For example, unallocated hint instructions are documented as being reserved and executed as NOPs.

2. If the operation pseudocode for the matching encoding diagrams starts with a condition code check, perform that check. If the condition code check fails, abandon this execution model and treat the instruction as a NOP. If there are multiple matching encoding diagrams, either all or none of their corresponding pieces of common pseudocode start with a condition code check.

3. Perform the encoding-specific pseudocode for each of the matching encoding diagrams independently and in parallel. Each such piece of encoding-specific pseudocode starts with a bitstring variable for each named bit or multi-bit field in its corresponding encoding diagram, named the same as the bit or multi-bit field and initialized with the values of the corresponding bit or bits from the bit pattern of the instruction.

In a few cases, the encoding diagram contains more than one bit or field with same name. In these cases, the values of the different instances of those bits or fields must be identical. The encoding-specific pseudocode contains a special case using the Consistent() function to specify what happens if they are not identical. Consistent() returns TRUE if all instruction bits or fields with the same name as its argument have the same value, and FALSE otherwise.

If there are multiple matching encoding diagrams, all but one of the corresponding pieces of pseudocode must contain a special case that indicates that it does not apply. Discard the results of all such pieces of pseudocode and their corresponding encoding diagrams.
There is now one remaining piece of pseudocode and its corresponding encoding diagram left to consider. This pseudocode might also contain a special case, most commonly one indicating that it is CONSTRAINED UNPREDICTABLE. If so, abandon this execution model and treat the instruction according to the special case.

4. Check the *should be* bits of the encoding diagram against the corresponding bits of the bit pattern of the instruction. If any of them do not match, abandon this execution model and treat the instruction as CONSTRAINED UNPREDICTABLE, see SBZ or SBO fields T32 and A32 in instructions on page K1-11566.

5. Perform the rest of the operation pseudocode for the instruction description that contains the encoding diagram. That pseudocode starts with all variables set to the values they were left with by the encoding-specific pseudocode.

The ConditionPassed() call in the common pseudocode, if present, performs step 2, and the EncodingSpecificOperations() call performs steps 3 and 4.

### K16.2.2 Limitations of the instruction pseudocode

The pseudocode descriptions of instruction functionality have a number of limitations. These are mainly due to the fact that, for clarity and brevity, the pseudocode is a sequential and mostly deterministic language.

These limitations include:

- Pseudocode does not describe the ordering requirements when an instruction generates multiple memory accesses. For a description of the ordering requirements on memory accesses, see Ordering constraints on page E2-7174.
- Pseudocode does not describe the exact rules when an instruction that generates any of the following fails its condition code check:
  - UNDEFINED instruction.
  - Hyp trap.
  - Monitor trap.
  - Trap to AArch64 exception.

In such cases, the UNDEFINED pseudocode statement or call to the applicable trap function lies inside the if ConditionPassed() then ... structure, either directly or in the EncodingSpecificOperations() function call, and so the pseudocode indicates that the instruction executes as a NOP. For the exact rules, see:

- Conditional execution of undefined instructions on page G1-8966.
- EL2 configurable controls on page G1-9010.
- EL3 configurable controls on page G1-9030.
- Configurable instruction controls on page D1-4665.

- Pseudocode does not describe the exact ordering requirements when a single floating-point instruction generates more than one floating-point exception and one or more of those floating-point exceptions is trapped. Combinations of floating-point exceptions on page E1-7151 describes the exact rules.

--- **Note**

There is no limitation in the case where all the floating-point exceptions are untrapped, because the pseudocode specifies the same behavior as the cross-referenced section.

---

- An exception can be taken during execution of the pseudocode for an instruction, either explicitly as a result of the execution of a pseudocode function such as Abort(), or implicitly, for example if an interrupt is taken during execution of an LDM instruction. If this happens, the pseudocode does not describe the extent to which the normal behavior of the instruction occurs. To determine that, see the descriptions of the exceptions in Handling exceptions that are taken to an Exception level using AArch32 on page G1-8929.
K16.3 Data types

This section describes:

- General data type rules.
- Bitstrings.
- Integers on page K16-11812.
- Reals on page K16-11812.
- Booleans on page K16-11812.
- Enumerations on page K16-11813.
- Structures on page K16-11813.
- Tuples on page K16-11814.
- Arrays on page K16-11815.

K16.3.1 General data type rules

Arm architecture pseudocode is a strongly typed language. Every literal and variable is of one of the following types:

- Bitstring.
- Integer.
- Boolean.
- Real.
- Enumeration.
- Tuple.
- Struct.
- Array.

The type of a literal is determined by its syntax. A variable can be assigned to without an explicit declaration. The variable implicitly has the type of the assigned value. For example, the following assignments implicitly declare the variables \( x \), \( y \) and \( z \) to have types integer, bitstring of length 1, and Boolean, respectively.

\[
x = 1;
y = '1';
z = TRUE;
\]

Variables can also have their types declared explicitly by preceding the variable name with the name of the type. The following example declares explicitly that a variable named \( \text{count} \) is an integer.

\[
\text{integer} \ \text{count};
\]

This is most often done in function definitions for the arguments and the result of the function.

The remaining subsections describe each data type in more detail.

K16.3.2 Bitstrings

This section describes the bitstring data type.

Syntax

\[
\text{bits(N)} \quad \text{The type name of a bitstring of length N.}
\]

\[
\text{bit} \quad \text{A synonym of bits(1).}
\]

Description

A bitstring is a finite-length string of 0s and 1s. Each length of bitstring is a different type. The minimum permitted length of a bitstring is 0.
Bitstring constants literals are written as a single quotation mark, followed by the string of 0s and 1s, followed by another single quotation mark. For example, the two constants literals of type bit are '0' and '1'. Spaces can be included in bitstrings for clarity.

The bits in a bitstring are numbered from left to right N-1 to 0. This numbering is used when accessing the bitstring using bitslices. In conversions to and from integers, bit N-1 is the MSByte and bit 0 is the LSByte. This order matches the order in which bitstrings derived from encoding diagrams are printed.

Every bitstring value has a left-to-right order, with the bits being numbered in standard little-endian order. That is, the leftmost bit of a bitstring of length N is bit (N-1) and its right-most bit is bit 0. This order is used as the most-significant-to-least-significant bit order in conversions to and from integers. For bitstring constants and bitstrings that are derived from encoding diagrams, this order matches the way that they are printed.

Bitstrings are the only concrete data type in pseudocode, corresponding directly to the contents values that are manipulated in registers, memory locations, and instructions. All other data types are abstract.

K16.3.3  Integers

This section describes the data type for integer numbers.

**Syntax**

```plaintext
integer
```

The type name for the integer data type.

**Description**

Pseudocode integers are unbounded in size and can be either positive or negative. That is, they are mathematical integers rather than what computer languages and architectures commonly call integers. Computer integers are represented in pseudocode as bitstrings of the appropriate length, and the pseudocode provides functions to interpret those bitstrings as integers.

Integer literals are normally written in decimal form, such as 0, 15, -1234. They can also be written in C-style hexadecimal form, such as 0x55 or 0x80000000. Hexadecimal integer literals are treated as positive unless they have a preceding minus sign. For example, 0x80000000 is the integer +2^31. If -2^31 needs to be written in hexadecimal, it must be written as -0x80000000.

K16.3.4  Reals

This section describes the data type for real numbers.

**Syntax**

```plaintext
real
```

The type name for the real data type.

**Description**

Pseudocode reals are unbounded in size and precision. That is, they are mathematical real numbers, not computer floating-point numbers. Computer floating-point numbers are represented in pseudocode as bitstrings of the appropriate length, and the pseudocode provides functions to interpret those bitstrings as reals.

Real constant literals are written in decimal form with a decimal point. This means 0 is an integer constant literal, but 0.0 is a real constant literal.

K16.3.5  Booleans

This section describes the Boolean data type.

**Syntax**

```plaintext
boolean
```

The type name for the Boolean data type.
TRUE  The two values a Boolean variable can take.

**Description**

A Boolean is a logical TRUE or FALSE value.

--- **Note** ---

This is not the same type as bit, which is a bitstring of length 1. A Boolean can only take on one of two values: TRUE or FALSE.

---

**K16.3.6 Enumerations**

This section describes the enumeration data type.

**Syntax and examples**

```
enumeration      Keyword to defined a new enumeration type.

enumeration Example {Example_One, Example_Two, Example_Three};
```

A definition of a new enumeration called Example, which can take on the values Example_One, Example_Two, Example_Three.

**Description**

An enumeration is a defined set of named values.

An enumeration must contain at least one named value. A named value must not be shared between enumerations.

Enumerations must be defined explicitly, although a variable of an enumeration type can be declared implicitly by assigning one of the named values to it. By convention, each named value starts with the name of the enumeration followed by an underscore. The name of the enumeration is its *type name*, or *type*, and the named values are its possible *values*.

---

**K16.3.7 Structures**

This section describes the structure data type.

**Syntax and examples**

```
type              The keyword used to declare the structure data type.

type ShiftSpec is (bits(2) shift, integer amount)
```

An example definition for a new structure called ShiftSpec that contains a bitstring member called shift and an integer member named amount. Structure definitions must not be terminated with a semicolon.

```
ShiftSpec abc;
```

A declaration of a variable named abc of type ShiftSpec.

```
abc.shift
```

Syntax to refer to the individual members within the structure variable.

**Description**

A structure is a compound data type composed of one or more data items. The data items can be of different data types. This can include compound data types. The data items of a structure are called its members and are named.
In the syntax section, the example defines a structure called ShiftSpec with two members. The first is a bitstring of length 2 named shift and the second is an integer named amount. After declaring a variable of that type named abc, the members of this structure are referred to as abc.shift and abc.amount.

Every definition of a structure creates a different type, even if the number and type of their members are identical. For example:

type ShiftSpec1 is (bits(2) shift, integer amount)
type ShiftSpec2 is (bits(2) shift, integer amount)

ShiftSpec1 and ShiftSpec2 are two different types despite having identical definitions. This means that the value in a variable of type ShiftSpec1 cannot be assigned to variable of type ShiftSpec2.

K16.3.8 Tuples

This section describes the tuple data type.

Examples

(bits(32) shifter_result, bit shifter_carry_out)
   An example of the tuple syntax.

(shift_t, shift_n) = ('00', 0);
   An example of assigning values to a tuple.

Description

A tuple is an ordered set of data items, separated by commas and enclosed in parentheses. The items can be of different types and a tuple must contain at least one data item.

Tuples are often used as the return type for functions that return multiple results. For example, in the syntax section, the example tuple is the return type of the function Shift_C(), which performs a standard A32/T32 shift or rotation. Its return type is a tuple containing two data items, with the first of type bits(32) and the second of type bit.

Each tuple is a separate compound data type. The compound data type is represented as a comma-separated list of ordered data types between parentheses. This means that the example tuple at the start of this section is of type (bits(32), bit). The general principle that types can be implied by an assignment extends to implying the type of the elements in the tuple. For example, in the syntax section, the example assignment implicitly declares:

• shift_t to be of type bits(2).
• shift_n to be of type integer.
• (shift_t, shift_n) to be a tuple of type (bits(2), integer).
K16.3.9 Arrays

This section describes the array data type.

Syntax

array

The type name for the array data type.

array data_type array_name[A..B];

Declaration of an array of type data_type, which might be compound data type. It is named array_name and is indexed with an integer range from A to B.

Description

An array is an ordered set of fixed size containing items of a single data type. This can include compound data types. Pseudocode arrays are indexed by either enumerations or integer ranges. An integer range is represented by the lower inclusive end of the range, then ..., then the upper inclusive end of the range.

For example:

The following example declares an array of 31 bitstrings of length 64, indexed from 0 to 30.

array bits(64) _R[0..30];

Arrays are always explicitly declared, and there is no notation for a constant literal array. Arrays always contain at least one element data item, because:

• Enumerations always contain at least one symbolic constant named value.
• Integer ranges always contain at least one integer.

An array declared with an enumeration type as the index must be accessed using enumeration values of that enumeration type. An array declared with an integer range type as the index must be accessed using integer values from that inclusive range. Accessing such an array with an integer value outside of the range is a coding error.

Arrays do not usually appear directly in pseudocode. The items that syntactically look like arrays in pseudocode are usually array-like functions such as R[i], MemU[address, size] or Elem[vector, i, size]. These functions package up and abstract additional operations normally performed on accesses to the underlying arrays, such as register banking, memory protection, endian-dependent byte ordering, exclusive-access housekeeping and Advanced SIMD element processing. See Function and procedure calls on page K16-11822.
K16.4 Operators

This section describes:

- Relational operators.
- Boolean operators.
- Bitstring operators on page K16-11817.
- Arithmetic operators on page K16-11817.
- The assignment operator on page K16-11818.
- Precedence rules on page K16-11820.
- Conditional expressions on page K16-11820.
- Operator polymorphism on page K16-11820.

K16.4.1 Relational operators

The following operations yield results of type boolean.

**Equality and non-equality**

If two variables \( x \) and \( y \) are of the same type, their values can be tested for equality by using the expression \( x == y \) and for non-equality by using the expression \( x != y \). In both cases, the result is of type boolean.

Both \( x \) and \( y \) must be of type \text{bits}(N), \text{real}, \text{enumeration}, \text{boolean}, \text{or integer}. Named values from an enumeration can only be compared if they are both from the same enumeration. An exception is that a bitstring can be tested for equality with an integer to allow a \( d==15 \) test.

A special form of comparison is defined with a bitstring literal that can contain bit values '0', '1', and 'x'. Any bit with value 'x' is ignored in determining the result of the comparison. For example, if \( \text{opcode} \) is a 4-bit bitstring, the expression \( \text{opcode} == '1x0x' \) matches the values '1000', '1100', '1001', and '1101'. This is known as a bitmask.

---

**Note**

This special form is permitted in the implied equality comparisons in the when parts of case .. of .. structures.

**Comparisons**

If \( x \) and \( y \) are integers or reals, then \( x < y, x <= y, x > y, \) and \( x >= y \) are less than, less than or equal, greater than, and greater than or equal comparisons between them, producing Boolean results.

**Set membership with IN**

\(<\text{expression}> \ IN \ \{<\text{set}>\} \) produces TRUE if \( \text{expression} \) is a member of \( \text{set} \). Otherwise, it is FALSE. \( \text{set} \) must be a list of expressions separated by commas.

K16.4.2 Boolean operators

If \( x \) is a Boolean expression, then \( !x \) is its logical inverse.

If \( x \) and \( y \) are Boolean expressions, then \( x \ & \ y \) is the result of ANDing them together. As in the C language, if \( x \) is FALSE, the result is determined to be FALSE without evaluating \( y \).

---

**Note**

This is known as short circuit evaluation.

---

If \( x \) and \( y \) are booleans, then \( x \ || \ y \) is the result of ORing them together. As in the C language, if \( x \) is TRUE, the result is determined to be TRUE without evaluating \( y \).
Note

If \( x \) and \( y \) are booleans or Boolean expressions, then the result of \( x \neq y \) is the same as the result of exclusive-ORing \( x \) and \( y \) together. The operator EOR only accepts bitstring arguments.

K16.4.3 Bitstring operators

The following operations can be applied only to bitstrings.

Logical operations on bitstrings

If \( x \) is a bitstring, \( \text{NOT}(x) \) is the bitstring of the same length obtained by logically inverting every bit of \( x \).

If \( x \) and \( y \) are bitstrings of the same length, \( x \text{ AND } y \), \( x \text{ OR } y \), and \( x \text{ EOR } y \) are the bitstrings of that same length obtained by logically ANDing, logically ORing, and exclusive-ORing corresponding bits of \( x \) and \( y \) together.

Bitstring concatenation and slicing

If \( x \) and \( y \) are bitstrings of lengths \( N \) and \( M \) respectively, then \( x:y \) is the bitstring of length \( N+M \) constructed by concatenating \( x \) and \( y \) in left-to-right order.

The bitstring slicing operator addresses specific bits in a bitstring. This can be used to create a new bitstring from extracted bits or to set the value of specific bits. Its syntax is \( x<i, j, k, \ldots, n> \), where \( x \) is the integer or bitstring being sliced, and \( <integer_list> \) is a comma-separated list of integers enclosed in angle brackets. The length of the resulting bitstring is equal to the number of integers in \( <integer_list> \). In \( x<i> \), each of the integers in \( <integer_list> \) must be:

- \( \geq 0 \).
- \( < \text{Len}(x) \) if \( x \) is a bitstring.

The definition of \( x<i> \) depends on whether \( integer_list \) contains more than one integer:

- If \( integer_list \) contains just one integer \( i \), \( x<i> \) is defined to be:
  - If \( x \) is a bitstring, \( '0' \) if bit \( i \) of \( x \) is a zero and \( '1' \) if bit \( i \) of \( x \) is a one.
  - If \( x \) is an integer, and \( y \) is the unique integer in the range 0 to \( 2^{(i+1)}-1 \) that is congruent to \( x \) modulo \( 2^{i+1} \). Then \( x<i> \) is \( '0' \) if \( y < 2^i \) and \( '1' \) if \( y \geq 2^i \).

Loosely, this definition treats an integer as equivalent to a sufficiently long two’s complement representation of it as a bitstring.

The notation for a range expression is \( i:j \) with \( i \geq j \) is shorthand for the integers in order from \( i \) down to \( j \), with both end values included. For example, \( \text{instr}<31:28> \) represents \( \text{instr}<31, 30, 29, 28> \).

\( x<i> \) is assignable provided \( x \) is an assignable bitstring and no integer appears more than once in \( <integer_list> \). In particular, \( x<i> \) is assignable if \( x \) is an assignable bitstring and \( 0 \leq i < \text{Len}(x) \).

Encoding diagrams for registers frequently show named bits or multi-bit fields. For example, the encoding diagram for the APSR shows its bit<31> as \( N \). In such cases, the syntax APSR.<N> is used as a more readable synonym for APSR<31> as named bits can be referred to with the same syntax as referring to members of a struct. A comma-separated list of named bits enclosed in angle brackets following the register name allows multiple bits to be addressed simultaneously. For example, APSR.<N, C, Q> is synonymous with APSR <31, 29, 27>.

K16.4.4 Arithmetic operators

Most pseudocode arithmetic is performed on integer or real values, with operands obtained by conversions from bitstrings and results converted back to bitstrings. As these data types are the unbounded mathematical types, no issues arise about overflow or similar errors.
Unary plus and minus

If \( x \) is an integer or real, then \(+x\) is \( x \) unchanged, \(-x\) is \( x \) with its sign reversed. Both are of the same type as \( x \).

Addition and subtraction

If \( x \) and \( y \) are integers or reals, \( x+y \) and \( x-y \) are their sum and difference. Both are of type \textit{integer} if \( x \) and \( y \) are both of type \textit{integer}, and \textit{real} otherwise.

There are two cases where the types of \( x \) and \( y \) can be different. A bitstring and an integer can be added together to allow the operation \( PC + 4 \). An integer can be subtracted from a bitstring to allow the operation \( PC - 2 \).

If \( x \) and \( y \) are bitstrings of the same length \( N \), so that \( N = \text{Len}(x) = \text{Len}(y) \), then \( x+y \) and \( x-y \) are the least significant \( N \) bits of the results of converting \( x \) and \( y \) to integers and adding or subtracting them. Signed and unsigned conversions produce the same result:

\[
\begin{align*}
x+y &= (\text{SInt}(x) + \text{SInt}(y))_{N-1:0} \\
     &= (\text{UInt}(x) + \text{UInt}(y))_{N-1:0} \\
x-y &= (\text{SInt}(x) - \text{SInt}(y))_{N-1:0} \\
     &= (\text{UInt}(x) - \text{UInt}(y))_{N-1:0}
\end{align*}
\]

If \( x \) is a bitstring of length \( N \) and \( y \) is an integer, \( x+y \) and \( x-y \) are the bitstrings of length \( N \) defined by \( x+y = x + y_{N-1:0} \) and \( x-y = x - y_{N-1:0} \). Similarly, if \( x \) is an integer and \( y \) is a bitstring of length \( M \), \( x+y \) and \( x-y \) are the bitstrings of length \( M \) defined by \( x+y = x_{M-1:0} + y \) and \( x-y = x_{M-1:0} - y \).

Multiplication

If \( x \) and \( y \) are integers or reals, then \( x \times y \) is the product of \( x \) and \( y \). It is of type \textit{integer} if \( x \) and \( y \) are both of type \textit{integer}, and \textit{real} otherwise.

Division and modulo

If \( x \) and \( y \) are reals, then \( x/y \) is the result of dividing \( x \) by \( y \), and is always of type \textit{real}.

If \( x \) and \( y \) are integers, then \( x \div y \) and \( x \mod y \) are defined by:

\[
\begin{align*}
x \div y &= \text{RoundDown}(x/y) \\
x \mod y &= x - y \times (x \div y)
\end{align*}
\]

It is a pseudocode error to use any of \( x/y \), \( x \mod y \), or \( x \div y \) in any context where \( y \) can be zero.

Scaling

If \( x \) and \( n \) are of type \textit{integer}, then:

- \( x \ll n = \text{RoundDown}(x \times 2^n) \).
- \( x \gg n = \text{RoundDown}(x \times 2^{\neg n}) \).

Raising to a power

If \( x \) is an integer or a real and \( n \) is an integer, then \( x^n \) is the result of raising \( x \) to the power of \( n \), and:

- If \( x \) is of type \textit{integer}, then \( x^n \) is of type \textit{integer}.
- If \( x \) is of type \textit{real}, then \( x^n \) is of type \textit{real}.

K16.4.5 The assignment operator

The assignment operator is the \( = \) character, which assigns the value of the right-hand side to the left-hand side. An assignment statement takes the form:

\[
<\text{assignable_expression}> = <\text{expression}>
\]

This following subsection defines valid expression syntax.
General expression syntax

An expression is one of the following:

- A literal.
- A variable, optionally preceded by a data type name to declare its type.
- The word `UNKNOWN` preceded by a data type name to declare its type.
- The result of applying a language-defined operator to other expressions.
- The result of applying a function to other expressions.

Variable names normally consist of alphanumeric and underscore characters, starting with an alphabetic or underscore character.

Each register defined in an Arm architecture specification defines a correspondingly named pseudocode bitstring variable, and that variable has the stated behavior of the register. For example, if a bit of a register is defined as RAZ/WI, then the corresponding bit of its variable reads as '0' and ignore writes.

An expression like `bits(32) UNKNOWN` indicates that the result of the expression is a value of the given type, but the architecture does not specify what value it is and software must not rely on such values. The value produced must not:

- Return information that cannot be accessed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE or CONSTRAINED UNPREDICTABLE and do not return UNKNOWN values,
- Be promoted as providing any useful information to software.

--- Note ---

UNKNOWN values are similar to the definition of UNPREDICTABLE, but do not indicate that the entire architectural state becomes unspecified.

Only the following expressions are assignable. This means that these are the only expressions that can be placed on the left-hand side of an assignment.

- Variables.
- The results of applying some operators to other expressions.
  
  The description of each language-defined operator that can generate an assignable expression specifies the circumstances under which it does so. For example, those circumstances might require that one or more of the expressions the operator operates on is an assignable expression.
- The results of applying array-like functions to other expressions. The description of an array-like function specifies the circumstances under which it can generate an assignable expression.

--- Note ---

If the right-hand side in an assignment is a function returning a tuple, an item in the assignment destination can be written as `-` to indicate that the corresponding item of the assigned tuple value is discarded. For example:

```
(shifted, -) = LSL_C(operand, amount);
```

The expression on the right-hand side itself can be a tuple. For example:

```
(x, y) = (function_1(), function_2());
```

Every expression has a data type.

- For a literal, this data type is determined by the syntax of the literal.
- For a variable, there are the following possible sources for the data type
  - An optional preceding data type name.
  - A data type the variable was given earlier in the pseudocode by recursive application of this rule.
  - A data type the variable is being given by assignment, either by direct assignment to the variable, or by assignment to a list of which the variable is a member.

It is a pseudocode error if none of these data type sources exists for a variable, or if more than one of them exists and they do not agree about the type.

- For a language-defined operator, the definition of the operator determines the data type.
• For a function, the definition of the function determines the data type.

K16.4.6 Precedence rules

The precedence rules for expressions are:

1. Literals, variables and function invocations are evaluated with higher priority than any operators using their results, but see Boolean operators on page K16-11816.
2. Operators on integers follow the normal operator precedence rules of exponentiation before multiply/divide before add/subtract, with sequences of multiply/divides or add/subtracts evaluated left-to-right.
3. Other expressions must be parenthesized to indicate operator precedence if ambiguity is possible, but need not be if all permitted precedence orders under the type rules necessarily lead to the same result. For example, if i, j and k are integer variables, i > 0 && j > 0 && k > 0 is acceptable, but i > 0 && j > 0 || k > 0 is not.

K16.4.7 Conditional expressions

If x and y are two values of the same type and t is a value of type boolean, then if t then x else y is an expression of the same type as x and y that produces x if t is TRUE and y if t is FALSE.

K16.4.8 Operator polymorphism

Operators in pseudocode can be polymorphic, with different functionality when applied to different data types. Each resulting form of an operator has a different prototype definition. For example, the operator + has forms that act on various combinations of integers, reals and bitstrings.

Table K16-1 on page K16-11820 summarizes the operand types valid for each unary operator and the result type. Table K16-2 on page K16-11820 summarizes the operand types valid for each binary operator and the result type.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operand Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>NOT</td>
<td>bits(N)</td>
<td>bits(N)</td>
</tr>
<tr>
<td>!</td>
<td>boolean</td>
<td>boolean</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>First operand type</th>
<th>Second operand type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>bits(N)</td>
<td>integer</td>
<td>boolean</td>
</tr>
<tr>
<td></td>
<td>integer</td>
<td>integer</td>
<td>boolean</td>
</tr>
<tr>
<td></td>
<td>real</td>
<td>real</td>
<td>boolean</td>
</tr>
<tr>
<td></td>
<td>enumeration</td>
<td>enumeration</td>
<td>boolean</td>
</tr>
<tr>
<td></td>
<td>boolean</td>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>!=</td>
<td>bits(N)</td>
<td>bits(N)</td>
<td>boolean</td>
</tr>
<tr>
<td></td>
<td>integer</td>
<td>integer</td>
<td>boolean</td>
</tr>
<tr>
<td></td>
<td>real</td>
<td>real</td>
<td>boolean</td>
</tr>
</tbody>
</table>
## Table K16-2 Result and operand types permitted for binary operators (continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>First operand type</th>
<th>Second operand type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code>, <code>&gt;</code></td>
<td>integer</td>
<td>integer</td>
<td>boolean</td>
</tr>
<tr>
<td><code>&lt;=</code>, <code>&gt;=</code></td>
<td>real</td>
<td>real</td>
<td></td>
</tr>
<tr>
<td><code>+</code>, <code>-</code></td>
<td>integer</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>real</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>bits(N)</td>
<td>bits(N)</td>
<td>bits(N)</td>
</tr>
<tr>
<td></td>
<td>integer</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td><code>&lt;&lt;</code>, <code>&gt;&gt;</code></td>
<td>integer</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td><code>*</code></td>
<td>real</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>bits(N)</td>
<td>bits(N)</td>
<td>bits(N)</td>
</tr>
<tr>
<td><code>/</code></td>
<td>real</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td><code>DIV</code></td>
<td>integer</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td><code>MOD</code></td>
<td>integer</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>bits(N)</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td><code>&amp;&amp;</code>, `</td>
<td></td>
<td>`</td>
<td>boolean</td>
</tr>
<tr>
<td><code>AND</code>, <code>OR</code>, <code>EOR</code></td>
<td>bits(N)</td>
<td>bits(N)</td>
<td>bits(N)</td>
</tr>
<tr>
<td><code>^</code></td>
<td>integer</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>real</td>
<td>integer</td>
<td>real</td>
</tr>
</tbody>
</table>
K16.5  Statements and control structures

This section describes the statements and program structures available in the pseudocode:

- **Statements and Indentation.**
- **Function and procedure calls.**
- **Conditional control structures** on page K16-11824.
- **Loop control structures** on page K16-11825.
- **Special statements** on page K16-11826.
- **Comments** on page K16-11826.

K16.5.1  Statements and Indentation

A simple statement is either an assignment, a function call, or a procedure call. Each statement must be terminated with a semicolon.

Indentation normally indicates the structure in compound statements. The statements contained in structures such as `if ... then ... else ...` or procedure and function definitions are indented more deeply than the statement structure itself. The end of a compound statement structure and their end is indicated by returning to the original indentation level or less.

Indentation is normally done by four spaces for each level. Standard indentation uses four spaces for each level of indent.

K16.5.2  Function and procedure calls

This section describes how functions and procedures are defined and called in the pseudocode.
Procedure and function definitions

A procedure definition has the form:

<procedure name>(<argument prototypes>)

<statement 1>
<statement 2>
...
<statement n>

where <argument prototypes> consists of zero or more argument definitions, separated by commas. Each argument definition consists of a type name followed by the name of the argument.

Note

This first definition line is not terminated by a semicolon. This distinguishes it from a procedure call.

A function definition is similar, but also declares the return type of the function:

<return type> <function name>(<argument prototypes>)

<statement 1>
<statement 2>
...
<statement n>

Note

A function or procedure name can include a ".". This is a convention used for functions that have similar but different behaviors in AArch32 and AArch64 states.

Array-like functions are similar, but are written with square brackets and have two forms. These two forms exist because reading from and writing to an array element require different functions. They are frequently used in memory operations. An array-like function definition with a return type is equivalent to reading from an array. For example:

<return type> <function name>[<argument prototypes>]

<statement 1>
<statement 2>
...
<statement n>

Its related function definition with no return type is equivalent to writing to an array. For example:

<function name>[<argument prototypes>] = <value prototype>

<statement 1>
<statement 2>
...
<statement n>

The value prototype determines what data type can be written to the array. The two related functions must share the same name, but the value prototype and return type can be different.

Procedure calls

A procedure call has the form:

<procedure_name>(<arguments>);

Return statements

A procedure return has the form:

return;

A function return has the form:
return <expression>;

where <expression> is of the type declared in the function prototype line.

K16.5.3 Conditional control structures

This section describes how conditional control structures are used in the pseudocode.

if ... then ... else ...

In addition to being a ternary operator, a multi-line if ... then ... else ... structure can act as a control structure and has the form:

if <boolean_expression> then
  <statement 1>;
  <statement 2>;
  ...
  <statement n>;
elsif <boolean_expression> then
  <statement a>;
  <statement b>;
  ...
  <statement z>;
else
  <statement A>;
  <statement B>;
  ...
  <statement Z>;

The block of lines consisting of elsif and its indented statements is optional, and multiple elsif blocks can be used.

The block of lines consisting of else and its indented statements is optional.

Abbreviated one-line forms can be used when the then part, and in the else part if it is present, contain only simple statements such as:

if <boolean_expression> then <statement 1>;
if <boolean_expression> then <statement 1>; else <statement A>;
if <boolean_expression> then <statement 1>; <statement 2>; else <statement A>;

Note

In these forms, <statement 1>, <statement 2>, and <statement A> must be terminated by semicolons. This, and the fact that the else part is optional, distinguish its use as a control structure from its use as a ternary operator.

case ... of ...

A case ... of ... structure has the form:

case <expression> of
  when <literal values1>
    <statement 1>;
    <statement 2>;
    ...
    <statement m>;
  when <literal values2>
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;
  ... more "when" groups if required ...
  otherwise
<statement A>;
<statement B>;
...
<statement Z>;

In this structure, <literal values1> and <literal values2> consist of literal values of the same type as <expression>, separated by commas. There can be additional when groups in the structure. Abbreviated one line forms of when and otherwise parts can be used when they contain only simple statements.

If <expression> has a bitstring type, the literal values can also include bitstring literals containing 'x' bits, known as bitmasks. For details, see Equality and non-equality on page K16-11816.

K16.5.4 Loop control structures

This section describes the three loop control structures used in the pseudocode.

repeat ... until ...

A repeat ... until ... structure has the form:

repeat
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;
until <boolean_expression>;

It executes the statement block at least once, and the loop repeats until <boolean expression> evaluates to TRUE. Variables explicitly declared inside the loop body have scope local to that loop and might not be accessed outside the loop body.

while ... do

A while ... do structure has the form:

while <boolean_expression> do
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;

It begins executing the statement block only if the Boolean expression is true. The loop then runs until the expression is false.

for ...

A for ... structure has the form:

for <assignable_expression> = <integer_expr1> to <integer_expr2>
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;

The <assignable_expression> is initialized to <integer_expr1> and compared to <integer_expr2>. If <integer_expr1> is less than <integer_expr2>, the loop body is executed and the <assignable_expression> incremented by one. This repeats until <assignable expression> is more than or equal to <integer_expr2>.

There is an alternate form:

for <assignable_expression> = <integer_expr1> downto <integer_expr2>

where <integer_expr1> is decremented after the loop body executes and continues until <assignable expression> is less than or equal to <integer_expr2>. 
K16.5.5 Special statements

This section describes statements with particular architecturally defined behaviors.

UNDEFINED

This subsection describes the statement:

UNDEFINED;

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is that the Undefined Instruction exception is taken.

UNPREDICTABLE

This subsection describes the statement:

UNPREDICTABLE;

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is UNPREDICTABLE.

SEE...

This subsection describes the statement:

SEE <reference>;

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is that nothing occurs as a result of the current pseudocode because some other piece of pseudocode defines the required behavior. The <reference> indicates where that other pseudocode can be found.

It usually refers to another instruction, but can also refer to another encoding or note of the same instruction.

IMPLEMENTATION_DEFINED

This subsection describes the statement:

IMPLEMENTATION_DEFINED {"<text>"};

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is IMPLEMENTATION DEFINED. An optional <text> field can give more information.

K16.5.6 Comments

The pseudocode supports two styles of comments:

• // starts a comment that is terminated by the end of the line.
• /* starts a comment that is terminated by */.

/**/ statements might not be nested, and the first */ ends the comment.

Note

Comment lines do not require a terminating semicolon.
K16.6 Built-in functions

This section describes:

- Bitstring manipulation functions.
- Arithmetic functions on page K16-11828.

K16.6.1 Bitstring manipulation functions

The following bitstring manipulation functions are defined:

**Bitstring length and most significant bit**

If \( x \) is a bitstring:

- The bitstring length function \( \text{Len}(x) \) returns the length of \( x \) as an integer.

**Bitstring concatenation and replication**

If \( x \) is a bitstring and \( n \) is an integer with \( n \geq 0 \):

- \( \text{Replicate}(x, n) \) is the bitstring of length \( n \times \text{Len}(x) \) consisting of \( n \) copies of \( x \) concatenated together.
- \( \text{Zeros}(n) = \text{Replicate}(\text{'0'}, n) \).
- \( \text{Ones}(n) = \text{Replicate}(\text{'1'}, n) \).

**Bitstring count**

If \( x \) is a bitstring, \( \text{BitCount}(x) \) is an integer result equal to the number of bits of \( x \) that are ones.
Testing a bitstring for being all zero or all ones

If x is a bitstring:
- IsZero(x) produces TRUE if all of the bits of x are zeros and FALSE if any of them are ones.
- IsZeroBit(x) produces '1' if all of the bits of x are zeros and '0' if any of them are ones.

IsOnes(x) and IsOnesBit(x) work in the corresponding ways. This means:

\[
\begin{align*}
\text{IsZero}(x) & = (\text{BitCount}(x) == 0) \\
\text{IsOnes}(x) & = (\text{BitCount}(x) == \text{Len}(x)) \\
\text{IsZeroBit}(x) & = \text{if IsZero}(x) \text{ then '1' else '0'} \\
\text{IsOnesBit}(x) & = \text{if IsOnes}(x) \text{ then '1' else '0'}
\end{align*}
\]

Lowest and highest set bits of a bitstring

If x is a bitstring, and \( N = \text{Len}(x) \):
- LowestSetBit(x) is the minimum bit number of any of the bits of x that are ones. If all of its bits are zeros, \( \text{LowestSetBit}(x) = N \).
- HighestSetBit(x) is the maximum bit number of any of the bits of x that are ones. If all of its bits are zeros, \( \text{HighestSetBit}(x) = -1 \).
- CountLeadingZeroBits(x) is the number of zero bits at the left end of x, in the range 0 to \( N \). This means:
  \[
  \text{CountLeadingZeroBits}(x) = N - 1 - \text{HighestSetBit}(x).
  \]
- CountLeadingSignBits(x) is the number of copies of the sign bit of x at the left end of x, excluding the sign bit itself, and is in the range 0 to \( N-1 \). This means:
  \[
  \text{CountLeadingSignBits}(x) = \text{CountLeadingZeroBits}(<N-1:1> \text{ EOR } x<N-2:0>).
  \]

Zero-extension and sign-extension of bitstrings

If x is a bitstring and i is an integer, then ZeroExtend(x, i) is x extended to a length of i bits, by adding sufficient zero bits to its left. That is, if i == \( \text{Len}(x) \), then ZeroExtend(x, i) = x, and if i > \( \text{Len}(x) \), then:

\[
\text{ZeroExtend}(x, i) = \text{Replicate('0', i-\text{Len}(x)}) : x
\]

If x is a bitstring and i is an integer, then SignExtend(x, i) is x extended to a length of i bits, by adding sufficient copies of its leftmost bit to its left. That is, if i == \( \text{Len}(x) \), then SignExtend(x, i) = x, and if i > \( \text{Len}(x) \), then:

\[
\text{SignExtend}(x, i) = \text{Replicate(\text{TopBit}(x), i-\text{Len}(x)}) : x
\]

It is a pseudocode error to use either ZeroExtend(x, i) or SignExtend(x, i) in a context where it is possible that i < \( \text{Len}(x) \).

Converting bitstrings to integers

If x is a bitstring, SInt() is the integer whose two’s complement representation is x.

UInt() is the integer whose unsigned representation is x.

\text{Int}(x, \text{unsigned}) returns either SInt(x) or UInt(x) depending on the value of its second argument.

K16.6.2 Arithmetic functions

This section defines built-in arithmetic functions.

Absolute value

If x is either of type real or integer, Abs(x) returns the absolute value of x. The result is the same type as x.
Rounding and aligning

If \( x \) is a real:
- \( \text{RoundDown}(x) \) produces the largest integer \( n \) such that \( n \leq x \).
- \( \text{RoundUp}(x) \) produces the smallest integer \( n \) such that \( n \geq x \).
- \( \text{RoundTowardsZero}(x) \) produces:
  - \( \text{RoundDown}(x) \) if \( x > 0.0 \).
  - 0 if \( x = 0.0 \).
  - \( \text{RoundUp}(x) \) if \( x < 0.0 \).

If \( x \) and \( y \) are both of type integer, \( \text{Align}(x, y) = y \times (x \text{ DIV } y) \), and is of type integer.

If \( x \) is of type bitstring and \( y \) is of type integer, \( \text{Align}(x, y) = (\text{Align}(\text{UInt}(x), y))<\text{Len}(x)-1:0> \), and is a bitstring of the same length as \( x \).

It is a pseudocode error to use either form of \( \text{Align}(x, y) \) in any context where \( y \) can be 0. In practice, \( \text{Align}(x, y) \) is only used with \( y \) a constant power of two, and the bitstring form used with \( y = 2^n \) has the effect of producing its argument with its \( n \) low-order bits forced to zero.

Maximum and minimum

If \( x \) and \( y \) are integers or reals, then \( \text{Max}(x, y) \) and \( \text{Min}(x, y) \) are their maximum and minimum respectively. \( x \) and \( y \) must both be of type integer or of type real. The function returns a value of the same type as its operands.
K16.7 Miscellaneous helper procedures and functions

This section lists the prototypes of miscellaneous helper procedures and functions used by the pseudocode, together with a brief description of the effect of the procedure or function. The pseudocode does not define the operation of these helper procedures and functions.

Note

Chapter J1 Armv8 Pseudocode also has an entry for each of these functions, but currently these entries do not say anything about the effect of the function. When this information is added in Chapter J1, this section will be removed from the manual.

K16.7.1 EndOfInstruction()

This procedure terminates processing of the current instruction.

EndOfInstruction();

K16.7.2 Hint_Debug()

This procedure supplies a hint to the debug system.

Hint_Debug(bits(4) option);

K16.7.3 Hint_PreloadData()

This procedure performs a preload data hint.

Hint_PreloadData(bits(32) address);

K16.7.4 Hint_PreloadDataForWrite()

This procedure performs a preload data hint with a probability that the use will be for a write.

Hint_PreloadDataForWrite(bits(32) address);

K16.7.5 Hint_PreloadInstr()

This procedure performs a preload instructions hint.

Hint_PreloadInstr(bits(32) address);

K16.7.6 Hint_Yield()

This procedure performs a Yield hint.

Hint_Yield();

K16.7.7 IsExternalAbort()

This function returns TRUE if the abort currently being processed is an External abort and FALSE otherwise. It is used only in exception entry pseudocode.

boolean IsExternalAbort(Fault type)
    assert type != Fault_None;

boolean IsExternalAbort(FaultRecord fault);
K16.7.8 IsAsyncAbort()

This function returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE otherwise. It is used only in exception entry pseudocode.

```c
boolean IsAsyncAbort(Fault type) {
    assert type != Fault_None;
    boolean IsAsyncAbort(FaultRecord fault);
}
```

K16.7.9 LSInstructionSyndrome()

This function returns the extended syndrome information for a fault reported in the HSR.

```c
bits(11) LSInstructionSyndrome();
```

K16.7.10 ProcessorID()

This function returns an integer that uniquely identifies the executing PE in the system.

```c
integer ProcessorID();
```

K16.7.11 RemapRegsHaveResetValues()

This function returns TRUE if the remap registers PRRR and NMRR have their IMPLEMENTATION DEFINED reset values, and FALSE otherwise.

```c
boolean RemapRegsHaveResetValues();
```

K16.7.12 ResetControlRegisters()

This function resets the System registers and memory-mapped control registers that have architecturally defined reset values to those values. For more information about the affected registers, see:

- Reset behavior on page D1-4674.
- PE state on reset into AArch32 state on page G1-8986.

```c
AArch64.ResetControlRegisters(boolean ResetIsCold)
AArch32.ResetControlRegisters(boolean ResetIsCold)
```

K16.7.13 ThisInstr()

This function returns the bitstring encoding of the currently executing instruction.

```c
bits(32) ThisInstr();
```

Note: Currently, this function is used only on 32-bit instruction encodings.

K16.7.14 ThisInstrLength()

This function returns the length, in bits, of the current instruction. This means it returns 32 or 16:

```c
integer ThisInstrLength();
```
K16.8 Arm pseudocode definition index

This section contains the following tables:

- Table K16-3 on page K16-11832 which contains the pseudocode data types.
- Table K16-4 on page K16-11832 which contains the pseudocode operators.
- Table K16-5 on page K16-11833 which contains the pseudocode keywords and control structures.
- Table K16-6 on page K16-11834 which contains the statements with special behaviors.

Table K16-3 Index of pseudocode data types

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>Type name for the array type</td>
</tr>
<tr>
<td>bit</td>
<td>Keyword equivalent to <code>bits(1)</code></td>
</tr>
<tr>
<td>bits(N)</td>
<td>Type name for the bitstring of length N data type</td>
</tr>
<tr>
<td>boolean</td>
<td>Type name for the Boolean data type</td>
</tr>
<tr>
<td>enumeration</td>
<td>Keyword to define a new enumeration type</td>
</tr>
<tr>
<td>integer</td>
<td>Type name for the integer data type</td>
</tr>
<tr>
<td>real</td>
<td>Type name for the real data type</td>
</tr>
<tr>
<td>type</td>
<td>Keyword to define a new structure</td>
</tr>
</tbody>
</table>

Table K16-4 Index of pseudocode operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Unary minus on integers or reals</td>
</tr>
<tr>
<td></td>
<td>Subtraction of integers, reals, and bitstrings</td>
</tr>
<tr>
<td></td>
<td>Used in the left-hand side of an assignment or a tuple to discard the result</td>
</tr>
<tr>
<td>+</td>
<td>Unary plus on integers or reals</td>
</tr>
<tr>
<td></td>
<td>Addition of integers, reals, and bitstrings</td>
</tr>
<tr>
<td>.</td>
<td>Extract named member from a list</td>
</tr>
<tr>
<td></td>
<td>Extract named bit or field from a register</td>
</tr>
<tr>
<td>:</td>
<td>Bitstring concatenation</td>
</tr>
<tr>
<td></td>
<td>Integer range in bitstring extraction operator</td>
</tr>
<tr>
<td>!</td>
<td>Boolean NOT</td>
</tr>
<tr>
<td>!=</td>
<td>Comparison for inequality</td>
</tr>
<tr>
<td>(...)</td>
<td>Around arguments of procedure or function</td>
</tr>
<tr>
<td>[...]</td>
<td>Around array index</td>
</tr>
<tr>
<td></td>
<td>Around arguments of array-like function</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication of integers, reals, and bitstrings</td>
</tr>
<tr>
<td>/</td>
<td>Division of reals</td>
</tr>
</tbody>
</table>
### Table K16-4 Index of pseudocode operators (continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>Boolean AND</td>
</tr>
<tr>
<td>&lt;</td>
<td><em>Less than</em> comparison of integers and reals</td>
</tr>
<tr>
<td>&lt;..&gt;</td>
<td>Slicing of specified bits of bitstring or integer</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Multiply integer by power of 2</td>
</tr>
<tr>
<td>&lt;=</td>
<td><em>Less than or equal</em> comparison of integers and reals</td>
</tr>
<tr>
<td>=</td>
<td>Assignment operator</td>
</tr>
<tr>
<td>==</td>
<td>Comparison for equality</td>
</tr>
<tr>
<td>&gt;</td>
<td><em>Greater than</em> comparison of integers and reals</td>
</tr>
<tr>
<td>&gt;=</td>
<td><em>Greater than or equal</em> comparison of integers and reals</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Divide integer by power of 2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Exponential operator</td>
</tr>
<tr>
<td>AND</td>
<td>Bitwise AND of bitstrings</td>
</tr>
<tr>
<td>DIV</td>
<td>Quotient from integer division</td>
</tr>
<tr>
<td>EOR</td>
<td>Bitwise EOR of bitstrings</td>
</tr>
<tr>
<td>IN</td>
<td>Tests membership of a certain expression in a set of values</td>
</tr>
<tr>
<td>MOD</td>
<td>Remainder from integer division</td>
</tr>
<tr>
<td>NOT</td>
<td>Bitwise inversion of bitstrings</td>
</tr>
<tr>
<td>OR</td>
<td>Bitwise OR of bitstrings</td>
</tr>
<tr>
<td>case ... of ...</td>
<td>Control structure for the</td>
</tr>
<tr>
<td>if ... then ... else ...</td>
<td>Condition expression selecting between two values</td>
</tr>
</tbody>
</table>

### Table K16-5 Index of pseudocode keywords and control structures

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/<em>...</em>/</td>
<td>Comment delimiters</td>
</tr>
<tr>
<td>//</td>
<td>Introduces comment terminated by end of line</td>
</tr>
<tr>
<td>FALSE</td>
<td>One of two values a Boolean can take (other than TRUE)</td>
</tr>
<tr>
<td>for ... = ...to ...</td>
<td>Loop control structure, counting up from the initial value to the upper limit</td>
</tr>
<tr>
<td>for ... = ... downto ...</td>
<td>Loop control structure, counting down from the initial value to the lower limit</td>
</tr>
<tr>
<td>if ... then ... else ...</td>
<td>Conditional control structure</td>
</tr>
<tr>
<td>otherwise</td>
<td>Introduces default case in case ... of ... control structure</td>
</tr>
</tbody>
</table>
repeat ... until ...
Loop control structure that runs at least once until the termination condition is satisfied

return
Procedure or function return

TRUE
One of two values a Boolean can take (other than FALSE)

when
Introduces specific case in case ... of ... control structure

while ... do ...
Loop control structure that runs until the termination condition is satisfied

Table K16-6 Index of special statements

Keyword | Meaning
---|---
IMPLEMENTATION_DEFINED | Describes IMPLEMENTATION DEFINED behavior
SEE | Points to other pseudocode to use instead
UNDEFINED | Cause Undefined Instruction exception
UNKNOWN | Unspecified value
UNPREDICTABLE | Unspecified behavior
Appendix K17
Registers Index

This appendix provides indexes to the register descriptions in this manual. It contains the following sections:

- *Introduction and register disambiguation* on page K17-11836.
- *Alphabetical index of AArch64 registers and System instructions* on page K17-11848.
- *Functional index of AArch64 registers and System instructions* on page K17-11868.
- *Alphabetical index of AArch32 registers and System instructions* on page K17-11889.
- *Functional index of AArch32 registers and System instructions* on page K17-11899.
- *Alphabetical index of memory-mapped registers* on page K17-11911.
- *Functional index of memory-mapped registers* on page K17-11923.
K17.1 Introduction and register disambiguation

In some sections of this manual, registers are referred to by a general name, where the description applies to more than one context. Generally, this is one of the following:

- The description applies to both AArch32 state and AArch64 state, and therefore the register names could apply to either AArch32 System registers or AArch64 System registers.
- The description applies to multiple Exception levels, and therefore at a particular Exception level the register names need to take the appropriate Exception level suffix, _EL0, _EL1, _EL2, or _EL3.

The following sections disambiguate the general register names:

- **Register name disambiguation by Execution state.**
- **Register name disambiguation by Exception level on page K17-11846.**

K17.1.1 Register name disambiguation by Execution state

Table K17-1 on page K17-11836 disambiguates the general names of the registers by Execution state.

<table>
<thead>
<tr>
<th>General name</th>
<th>Short description</th>
<th>AArch64 register</th>
<th>AArch32 register</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTEXTIDR</td>
<td>Context ID</td>
<td>CONTEXTIDR_EL1</td>
<td>CONTEXTIDR</td>
</tr>
<tr>
<td>DBGAUTHSTATUS</td>
<td>Debug Authentication Status</td>
<td>DBGAUTHSTATUS_EL1</td>
<td>DBGAUTHSTATUS</td>
</tr>
<tr>
<td>DBGBCR</td>
<td>Debug Breakpoint Control Registers</td>
<td>DBGBCR_&lt;n&gt;_EL1</td>
<td>DBGBCR_&lt;n&gt;</td>
</tr>
<tr>
<td>DBGBVR</td>
<td>Debug Breakpoint Value Registers</td>
<td>DBGBVR_&lt;n&gt;_EL1</td>
<td>DBGBVR_&lt;n&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DBGBXVR_&lt;n&gt;</td>
</tr>
<tr>
<td>DBGCLAIMCLR</td>
<td>Debug CLAIM Tag Clear register</td>
<td>DBGCLAIMCLR_EL1</td>
<td>DBGCLAIMCLR</td>
</tr>
<tr>
<td>DBGCLAIMSET</td>
<td>Debug CLAIM Tag Set register</td>
<td>DBGCLAIMSET_EL1</td>
<td>DBGCLAIMSET</td>
</tr>
<tr>
<td>DBGDTRRX</td>
<td>Debug Data Transfer Register, Receive</td>
<td>DBGDTRRX_EL0</td>
<td>DBGDTRRXint</td>
</tr>
<tr>
<td>DBGDTRTX</td>
<td>Debug Data Transfer Register, Transmit</td>
<td>DBGDTRTX_EL0</td>
<td>DBGDTRTXint</td>
</tr>
<tr>
<td>DBGPRCR</td>
<td>Debug Power Control Register</td>
<td>DBGPRCR_EL1</td>
<td>DBGPRCR</td>
</tr>
<tr>
<td>DBGVCR</td>
<td>Debug Vector Catch Register</td>
<td>DBGVCR32_EL2</td>
<td>DBGVCR</td>
</tr>
<tr>
<td>DBGWCR</td>
<td>Debug Watchpoint Control Registers</td>
<td>DBGWCR_&lt;n&gt;_EL1</td>
<td>DBGWCR_&lt;n&gt;</td>
</tr>
<tr>
<td>DBGWVR</td>
<td>Debug Watchpoint Value Registers</td>
<td>DBGWVR_&lt;n&gt;_EL1</td>
<td>DBGWVR_&lt;n&gt;</td>
</tr>
<tr>
<td>DCCINT</td>
<td>Debug Comms Channel Interrupt Enable</td>
<td>MDCCINT_EL1</td>
<td>DBGDCCINT</td>
</tr>
<tr>
<td>DCCSR</td>
<td>Debug Comms Channel Status Register</td>
<td>MDCCSR_EL0</td>
<td>DBGDSCRint</td>
</tr>
<tr>
<td>DLR</td>
<td>Debug Link Register</td>
<td>DLR_EL0[31:0]</td>
<td>DLR</td>
</tr>
<tr>
<td>DSCR</td>
<td>Debug System Control Register</td>
<td>MDSCR_EL1</td>
<td>DBGDSCRext</td>
</tr>
<tr>
<td>DSPSR</td>
<td>Debug Saved PE State Register</td>
<td>DSPSR_EL1</td>
<td>DSPSR</td>
</tr>
<tr>
<td>FAR</td>
<td>Fault Address Register</td>
<td>FAR_EL1</td>
<td>DFAR, IFAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FAR_EL2</td>
<td>HDFAR, HIFAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FAR_EL3</td>
<td>FAR_EL3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HPFAR_EL2</td>
<td>HPFAR</td>
</tr>
<tr>
<td>General name</td>
<td>Short description</td>
<td>AArch64 register</td>
<td>AArch32 register</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------</td>
<td>-----------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>HCR</td>
<td>Hypervisor Configuration Register</td>
<td>HCR_EL2</td>
<td>HCR</td>
</tr>
<tr>
<td>HDCR</td>
<td>Hyp or EL2 Debug Control Register</td>
<td>MDCR_EL2</td>
<td>HDCR</td>
</tr>
<tr>
<td>HSCTLR</td>
<td>Hypervisor System Control Register</td>
<td>SCTLR_EL2</td>
<td>HSCTLR</td>
</tr>
<tr>
<td>HTTBR</td>
<td>EL2 Translation Table Base Register</td>
<td>TTBR0_EL2</td>
<td>HTTBR</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Status Register</td>
<td>ISR_EL1</td>
<td>ISR</td>
</tr>
<tr>
<td>MPIDR</td>
<td>Multiprocessor Affinity Register</td>
<td>MPIDR_EL1</td>
<td>MPIDR</td>
</tr>
<tr>
<td>OSDLR</td>
<td>OS Double-Lock Register</td>
<td>OSDLR_EL1</td>
<td>DBGOSDLR</td>
</tr>
<tr>
<td>OSDTRRX</td>
<td>OS Lock Data Transfer Register, Receive</td>
<td>OSDTRRX_EL1</td>
<td>DBGDTRRXext</td>
</tr>
<tr>
<td>OSDTRTX</td>
<td>OS Lock Data Transfer Register, Transmit</td>
<td>OSDTRTX_EL1</td>
<td>DBGDTRTXext</td>
</tr>
<tr>
<td>OSECCR</td>
<td>OS Lock Exception Catch Control Register</td>
<td>OSECCR_EL1</td>
<td>DBGOSECCR</td>
</tr>
<tr>
<td>OSLAR</td>
<td>OS Lock Access Register</td>
<td>OSLAR_EL1</td>
<td>DBGOSLAR</td>
</tr>
<tr>
<td>OSLSR</td>
<td>OS Lock Status Register</td>
<td>OSLSR_EL1</td>
<td>DBGOSLSR</td>
</tr>
<tr>
<td>PMMIR</td>
<td>Performance Monitors Machine Identification Register</td>
<td>PMMIR_EL1</td>
<td>PMMIR</td>
</tr>
<tr>
<td>SCR</td>
<td>Secure Configuration Register</td>
<td>SCR_EL3</td>
<td>SCR</td>
</tr>
<tr>
<td>SCTLR</td>
<td>System Control Register</td>
<td>SCTLR_EL1</td>
<td>SCTLR (NS)</td>
</tr>
<tr>
<td>SDCR</td>
<td>Secure or EL3 Debug Configuration Register</td>
<td>MDCR_EL3</td>
<td>SDCR</td>
</tr>
<tr>
<td>SDER</td>
<td>Secure Debug Enable Register</td>
<td>SDER32_EL3</td>
<td>SDER</td>
</tr>
<tr>
<td>SPSR</td>
<td>Saved Program Status Register</td>
<td>SPSR_EL1</td>
<td>SPSR (general description)</td>
</tr>
<tr>
<td>TCR</td>
<td>Translation Control Register</td>
<td>TCR_EL1</td>
<td>TTBCR(NS)</td>
</tr>
<tr>
<td>TTBR</td>
<td>Translation Table Base Register</td>
<td>TTBR0_EL1</td>
<td>TTBR0</td>
</tr>
</tbody>
</table>
Table K17-1 Disambiguation of general names of registers by Execution state (continued)

<table>
<thead>
<tr>
<th>General name</th>
<th>Short description</th>
<th>AArch64 register</th>
<th>AArch32 register</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBAR</td>
<td>Vector Base Address Register</td>
<td>VBAR_EL1</td>
<td>VBAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VBAR_EL2</td>
<td>VBAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VBAR_EL3</td>
<td>VBAR</td>
</tr>
<tr>
<td>VCR</td>
<td>PL1&amp;0 stage 2 Translation Control Register</td>
<td>VTCR_EL2</td>
<td>VTCR</td>
</tr>
<tr>
<td>VTTBR</td>
<td>PL1&amp;0 stage 2 Translation Table Base Register</td>
<td>VTTBR_EL2</td>
<td>VTTBR</td>
</tr>
</tbody>
</table>

Table K17-2 on page K17-11838 disambiguates the general names of the System registers that provide access to the Performance Monitors by Execution state.

Table K17-2 Disambiguation of general names of the Performance Monitors System registers by Execution state

<table>
<thead>
<tr>
<th>General name</th>
<th>Short description</th>
<th>AArch64 register</th>
<th>AArch32 register</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMCCFILTR</td>
<td>Cycle Count Filter Register</td>
<td>PMCCFILTR_EL0</td>
<td>PMCCFILTR</td>
</tr>
<tr>
<td>PMCCNTR</td>
<td>Cycle Count Register</td>
<td>PMCCNTR_EL0</td>
<td>PMCCNTR</td>
</tr>
<tr>
<td>PMCEID0</td>
<td>Performance Monitors Common Event Identification Register 0</td>
<td>PMCEID0_EL0</td>
<td>PMCEID0</td>
</tr>
<tr>
<td>PMCEID1</td>
<td>Performance Monitors Common Event Identification Register 1</td>
<td>PMCEID1_EL0</td>
<td>PMCEID1</td>
</tr>
<tr>
<td>PMCEID2</td>
<td>Performance Monitors Common Event Identification Register 2</td>
<td>PMCEID0_EL0</td>
<td>PMCEID2</td>
</tr>
<tr>
<td>PMCEID3</td>
<td>Performance Monitors Common Event Identification Register 3</td>
<td>PMCEID1_EL0</td>
<td>PMCEID3</td>
</tr>
<tr>
<td>PMCNTENCLR</td>
<td>Performance Monitors Count Enable Clear register</td>
<td>PMCNTENCLR_EL0</td>
<td>PMCNTENCLR</td>
</tr>
<tr>
<td>PMCNTENSET</td>
<td>Performance Monitors Count Enable Set register</td>
<td>PMCNTENSET_EL0</td>
<td>PMCNTENSET</td>
</tr>
<tr>
<td>PMCR</td>
<td>Performance Monitors Control Register</td>
<td>PMCR_EL0</td>
<td>PMCR</td>
</tr>
<tr>
<td>PMEVCNTR&lt;n&gt;</td>
<td>Performance Monitors Event Count Registers, n = 0-30</td>
<td>PMEVCNTR&lt;n&gt;_EL0</td>
<td>PMEVCNTR&lt;n&gt;</td>
</tr>
<tr>
<td>PMEVTYPER&lt;n&gt;</td>
<td>Performance Monitors Event Type Registers, n = 0-30</td>
<td>PMEVTYPER&lt;n&gt;_EL0</td>
<td>PMEVTYPER&lt;n&gt;</td>
</tr>
<tr>
<td>PMINTENCLR</td>
<td>Performance Monitors Interrupt Enable Clear register</td>
<td>PMINTENCLR_EL1</td>
<td>PMINTENCLR</td>
</tr>
<tr>
<td>PMINTENSET</td>
<td>Performance Monitors Interrupt Enable Set register</td>
<td>PMINTENSET_EL1</td>
<td>PMINTENSET</td>
</tr>
<tr>
<td>PMMIR</td>
<td>Performance Monitors Machine Identification Register</td>
<td>PMMIR_EL1</td>
<td>PMMIR</td>
</tr>
<tr>
<td>PMOVSCLR</td>
<td>Performance Monitors Overflow Flag Status Register</td>
<td>PMOVSCLR_EL0</td>
<td>PMOVSR</td>
</tr>
<tr>
<td>PMOVSSET</td>
<td>Performance Monitors Overflow Flag Status Set register</td>
<td>PMOVSSET_EL0</td>
<td>PMOVSSET</td>
</tr>
<tr>
<td>PMSELR</td>
<td>Performance Monitors Event Counter Selection Register</td>
<td>PMSELR_EL0</td>
<td>PMSELR</td>
</tr>
<tr>
<td>PMSWINC</td>
<td>Performance Monitors Software Increment register</td>
<td>PMSWINC_EL0</td>
<td>PMSWINC</td>
</tr>
<tr>
<td>PMUSERENR</td>
<td>Performance Monitors User Enable Register</td>
<td>PMUSERENR_EL0</td>
<td>PMUSERENR</td>
</tr>
<tr>
<td>PMXEVCNTR</td>
<td>Performance Monitors Selected Event Count Register</td>
<td>PMXEVCNTR_EL0</td>
<td>PMXEVCNTR</td>
</tr>
<tr>
<td>PMXEVTYPEPER</td>
<td>Performance Monitors Selected Event Type Register</td>
<td>PMXEVTYPEPER_EL0</td>
<td>PMXEVTYPEPER</td>
</tr>
</tbody>
</table>
Table K17-3 on page K17-11839 disambiguates the general names of the System registers that provide access to the Activity Monitors by Execution state.

<table>
<thead>
<tr>
<th>General name</th>
<th>Short description</th>
<th>AArch64 register</th>
<th>AArch32 register</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMCFGR</td>
<td>Activity Monitors Configuration Register</td>
<td>AMCFGR_EL0</td>
<td>AMCFGR</td>
</tr>
<tr>
<td>AMCGCR</td>
<td>Activity Monitors Counter Group Configuration Register</td>
<td>AMCGCR_EL0</td>
<td>AMCGCR</td>
</tr>
<tr>
<td>AMCNTENCLR0</td>
<td>Activity Monitors Count Enable Clear Register 0</td>
<td>AMCNTENCLR0_EL0</td>
<td>AMCNTENCLR0</td>
</tr>
<tr>
<td>AMCNTENCLR1</td>
<td>Activity Monitors Count Enable Clear Register 1</td>
<td>AMCNTENCLR1_EL0</td>
<td>AMCNTENCLR1</td>
</tr>
<tr>
<td>AMCNTENSET0</td>
<td>Activity Monitors Count Enable Set Register 0</td>
<td>AMCNTENSET0_EL0</td>
<td>AMCNTENSET0</td>
</tr>
<tr>
<td>AMCNTENSET1</td>
<td>Activity Monitors Count Enable Set Register 1</td>
<td>AMCNTENSET1_EL0</td>
<td>AMCNTENSET1</td>
</tr>
<tr>
<td>AMCR</td>
<td>Activity Monitors Control Register</td>
<td>AMCR_EL0</td>
<td>AMCR</td>
</tr>
<tr>
<td>AMEVCNTR0&lt;n&gt;</td>
<td>Activity Monitors Event Counter Registers 0, n = 0-15</td>
<td>AMEVCNTR0&lt;n&gt;_EL0</td>
<td>AMEVCNTR0&lt;n&gt;</td>
</tr>
<tr>
<td>AMEVCNTR1&lt;n&gt;</td>
<td>Activity Monitors Event Counter Registers 1, n = 0-15</td>
<td>AMEVCNTR1&lt;n&gt;_EL0</td>
<td>AMEVCNTR1&lt;n&gt;</td>
</tr>
<tr>
<td>AMEVTYPE0&lt;n&gt;</td>
<td>Activity Monitors Event Type Registers 0, n = 0-15</td>
<td>AMEVTYPE0&lt;n&gt;_EL0</td>
<td>AMEVTYPE0&lt;n&gt;</td>
</tr>
<tr>
<td>AMEVTYPE1&lt;n&gt;</td>
<td>Activity Monitors Event Type Registers 1, n = 0-15</td>
<td>AMEVTYPE1&lt;n&gt;_EL0</td>
<td>AMEVTYPE1&lt;n&gt;</td>
</tr>
<tr>
<td>AMUSERENR</td>
<td>Activity Monitors User Enable Register</td>
<td>AMUSERENR_EL0</td>
<td>AMUSERENR</td>
</tr>
</tbody>
</table>

Table K17-4 on page K17-11839 disambiguates the general names of the System registers that provide access to the Generic Timer System by Execution state.

<table>
<thead>
<tr>
<th>General name</th>
<th>Short description</th>
<th>AArch64 register</th>
<th>AArch32 register</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNTFRQ</td>
<td>Counter-timer Frequency register</td>
<td>CNTFRQ_EL0</td>
<td>CNTFRQ</td>
</tr>
<tr>
<td>CNTHCTL</td>
<td>Counter-timer Hypervisor Control register</td>
<td>CNTHCTL_EL2</td>
<td>CNTHCTL</td>
</tr>
<tr>
<td>CNTHP_CTL</td>
<td>Counter-timer Hypervisor Physical Timer Control register</td>
<td>CNTHP_CTL_EL2</td>
<td>CNTHP_CTL</td>
</tr>
<tr>
<td>CNTHP_CVAL</td>
<td>Counter-timer Hypervisor Physical Timer CompareValue register</td>
<td>CNTHP_CVAL_EL2</td>
<td>CNTHP_CVAL</td>
</tr>
<tr>
<td>CNTHP_TVAL</td>
<td>Counter-timer Hypervisor Physical Timer Value register</td>
<td>CNTHP_TVAL_EL2</td>
<td>CNTHP_TVAL</td>
</tr>
<tr>
<td>CNTKCTL</td>
<td>Counter-timer Kernel Control register</td>
<td>CNTKCTL_EL1</td>
<td>CNTKCTL</td>
</tr>
<tr>
<td>CNTP_CTL</td>
<td>Counter-timer Physical Timer Control register</td>
<td>CNTP_CTL_EL0</td>
<td>CNTP_CTL</td>
</tr>
<tr>
<td>CNTP_CVAL</td>
<td>Counter-timer Physical Timer CompareValue register</td>
<td>CNTP_CVAL_EL0</td>
<td>CNTP_CVAL</td>
</tr>
<tr>
<td>CNTP_TVAL</td>
<td>Counter-timer Physical Timer Value register</td>
<td>CNTP_TVAL_EL0</td>
<td>CNTP_TVAL</td>
</tr>
<tr>
<td>CNTPCT</td>
<td>Counter-timer Physical Count register</td>
<td>CNTPCT_EL0</td>
<td>CNTPCT</td>
</tr>
<tr>
<td>CNTPS_CTL</td>
<td>Counter-timer Physical Secure Timer Control register</td>
<td>CNTPS_CTL_EL1</td>
<td>-</td>
</tr>
<tr>
<td>CNTPS_CVAL</td>
<td>Counter-timer Physical Secure Timer CompareValue register</td>
<td>CNTPS_CVAL_EL1</td>
<td>-</td>
</tr>
<tr>
<td>CNTPS_TVAL</td>
<td>Counter-timer Physical Secure Timer Value register</td>
<td>CNTPS_TVAL_EL1</td>
<td>-</td>
</tr>
</tbody>
</table>
Table K17-4 Disambiguation of general names of the Generic Timer System registers by Execution state (continued)

<table>
<thead>
<tr>
<th>General name</th>
<th>Short description</th>
<th>AArch64 register</th>
<th>AArch32 register</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTLR_EL1[31:0]</td>
<td>Counter-timer Virtual Timer Control register</td>
<td>ACTLR_EL1[31:0]</td>
<td>ACTLR</td>
</tr>
<tr>
<td>AFSR0_EL1[31:0]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>AFSR0_EL1[31:0]</td>
<td>AFSR</td>
</tr>
<tr>
<td>AFSR1_EL1[31:0]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>AFSR1_EL1[31:0]</td>
<td>AFSR</td>
</tr>
<tr>
<td>AMAIR_EL1[31:0]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>AMAIR_EL1[31:0]</td>
<td>AMAIR0</td>
</tr>
<tr>
<td>AMAIR_EL1[63:32]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>AMAIR_EL1[63:32]</td>
<td>AMAIR1</td>
</tr>
<tr>
<td>CONTEXTIDR_EL1[31:0]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>CONTEXTIDR_EL1[31:0]</td>
<td>CONTEXTIDR</td>
</tr>
<tr>
<td>CPACR_EL1[31:0]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>CPACR_EL1[31:0]</td>
<td>CPACR</td>
</tr>
<tr>
<td>CSSEL_EL1[31:0]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>CSSEL_EL1[31:0]</td>
<td>CSSEL</td>
</tr>
<tr>
<td>DACR32_EL2[31:0]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>DACR32_EL2[31:0]</td>
<td>DACR</td>
</tr>
<tr>
<td>FAR_EL1[31:0]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>FAR_EL1[31:0]</td>
<td>DFAR</td>
</tr>
<tr>
<td>ESR_EL1[31:0]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>ESR_EL1[31:0]</td>
<td>DFSR</td>
</tr>
<tr>
<td>HACR_EL2[31:0]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>HACR_EL2[31:0]</td>
<td>HACR</td>
</tr>
<tr>
<td>ACTLR_EL2[31:0]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>ACTLR_EL2[31:0]</td>
<td>HACTLR</td>
</tr>
<tr>
<td>AFSR0_EL1[31:0]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>AFSR0_EL1[31:0]</td>
<td>HAFSR</td>
</tr>
<tr>
<td>AFSR1_EL1[31:0]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>AFSR1_EL1[31:0]</td>
<td>HAIFSR</td>
</tr>
<tr>
<td>AMAIR_EL2[31:0]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>AMAIR_EL2[31:0]</td>
<td>HAMAIR0</td>
</tr>
<tr>
<td>CPTR_EL2[31:0]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>CPTR_EL2[31:0]</td>
<td>HCPR</td>
</tr>
<tr>
<td>HCR_EL2[31:0]</td>
<td>Counter-timer Virtual Timer CompareValue register</td>
<td>HCR_EL2[31:0]</td>
<td>HCR</td>
</tr>
</tbody>
</table>

Table K17-5 on page K17-11840 shows the mappings between the writable AArch64 System registers and the AArch32 System registers.

Table K17-5 Mapping of writable AArch64 System registers to the AArch32 System registers

<table>
<thead>
<tr>
<th>AArch64 register</th>
<th>AArch32 register</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTLR_EL1[31:0]</td>
<td>ACTLR</td>
</tr>
<tr>
<td>ACTLR_EL1[63:32]</td>
<td>ACTLR2 if implemented</td>
</tr>
<tr>
<td>AFSR0_EL1[31:0]</td>
<td>AFSR</td>
</tr>
<tr>
<td>AFSR1_EL1[31:0]</td>
<td>AFSR</td>
</tr>
<tr>
<td>AMAIR_EL1[31:0]</td>
<td>AMAIR0</td>
</tr>
<tr>
<td>AMAIR_EL1[63:32]</td>
<td>AMAIR1</td>
</tr>
<tr>
<td>CONTEXTIDR_EL1[31:0]</td>
<td>CONTEXTIDR</td>
</tr>
<tr>
<td>CPACR_EL1[31:0]</td>
<td>CPACR</td>
</tr>
<tr>
<td>CSSEL_EL1[31:0]</td>
<td>CSSEL</td>
</tr>
<tr>
<td>DACR32_EL2[31:0]</td>
<td>DACR</td>
</tr>
<tr>
<td>FAR_EL1[31:0]</td>
<td>DFAR</td>
</tr>
<tr>
<td>ESR_EL1[31:0]</td>
<td>DFSR</td>
</tr>
<tr>
<td>HACR_EL2[31:0]</td>
<td>HACR</td>
</tr>
<tr>
<td>ACTLR_EL2[31:0]</td>
<td>HACTLR</td>
</tr>
<tr>
<td>ACTLR_EL2[63:32]</td>
<td>HACTLR2 if implemented</td>
</tr>
<tr>
<td>AFSR0_EL1[31:0]</td>
<td>HAFSR</td>
</tr>
<tr>
<td>AFSR1_EL1[31:0]</td>
<td>HAIFSR</td>
</tr>
<tr>
<td>AMAIR_EL2[31:0]</td>
<td>HAMAIR0</td>
</tr>
<tr>
<td>AMAIR_EL2[63:32]</td>
<td>HAMAIR1</td>
</tr>
<tr>
<td>CPTR_EL2[31:0]</td>
<td>HCPR</td>
</tr>
<tr>
<td>HCR_EL2[31:0]</td>
<td>HCR</td>
</tr>
<tr>
<td>HCR_EL2[63:32]</td>
<td>HCR2</td>
</tr>
<tr>
<td>AArch64 register</td>
<td>AArch32 register</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>MDCR_EL2[31:0]</td>
<td>HDCR</td>
</tr>
<tr>
<td>FAR_EL2[31:0]</td>
<td>HDFAR</td>
</tr>
<tr>
<td>FAR_EL2[63:32]</td>
<td>HIFAR</td>
</tr>
<tr>
<td>MAIR_EL2[31:0]</td>
<td>HMAIR0</td>
</tr>
<tr>
<td>MAIR_EL2[63:32]</td>
<td>HMAIR1</td>
</tr>
<tr>
<td>HPFAR_EL2[31:0]</td>
<td>HPFAR</td>
</tr>
<tr>
<td>SCTLR_EL2[31:0]</td>
<td>HSCTLR</td>
</tr>
<tr>
<td>ESR_EL2[31:0]</td>
<td>HSR</td>
</tr>
<tr>
<td>HSTR_EL2[31:0]</td>
<td>HSTR</td>
</tr>
<tr>
<td>TCR_EL3[31:0]</td>
<td>HTCR</td>
</tr>
<tr>
<td>TPIDR_EL2[31:0]</td>
<td>HTPIDR</td>
</tr>
<tr>
<td>TTBR0_EL2[47:1]</td>
<td>HTTBR</td>
</tr>
<tr>
<td>VBAR_EL2[31:0]</td>
<td>HVBAR</td>
</tr>
<tr>
<td>FAR_EL1[63:32]</td>
<td>IFAR</td>
</tr>
<tr>
<td>IFSR32_EL2[31:0]</td>
<td>IFSR</td>
</tr>
<tr>
<td>MAIR_EL1[63:32]</td>
<td>NMRR or MAIR1</td>
</tr>
<tr>
<td>PAR_EL1[63:0]</td>
<td>PAR</td>
</tr>
<tr>
<td>MAIR_EL1[31:0]</td>
<td>PRRR or MAIR0</td>
</tr>
<tr>
<td>RMR_EL1[31:0]</td>
<td>RMR (at EL1)</td>
</tr>
<tr>
<td>RMR_EL2[31:0]</td>
<td>HRMR</td>
</tr>
<tr>
<td>RMR_EL3[31:0]</td>
<td>RMR (at EL3)</td>
</tr>
<tr>
<td>SCTLR_EL1[31:0]</td>
<td>SCTLR</td>
</tr>
<tr>
<td>SDER32_EL3[31:0]</td>
<td>SDER</td>
</tr>
<tr>
<td>TPIDR_EL1[31:0]</td>
<td>TPIDPRW</td>
</tr>
<tr>
<td>TPIDRRO_EL0[31:0]</td>
<td>TPIDRURO</td>
</tr>
<tr>
<td>TPIDR_EL0[31:0]</td>
<td>TPIDRURW</td>
</tr>
<tr>
<td>TCR_EL1[31:0]</td>
<td>TTBCR</td>
</tr>
<tr>
<td>TCR_EL1[63:32]</td>
<td>TTBCR2 if implemented</td>
</tr>
<tr>
<td>TTBR0_EL1[63:0]</td>
<td>TTBR0</td>
</tr>
<tr>
<td>TTBR1_EL1[63:0]</td>
<td>TTBR1</td>
</tr>
<tr>
<td>VBAR_EL1[31:0]</td>
<td>VBAR</td>
</tr>
<tr>
<td>VMPIDR_EL2[31:0]</td>
<td>VMPIDR</td>
</tr>
</tbody>
</table>
### Table K17-5 Mapping of writable AArch64 System registers to the AArch32 System registers

<table>
<thead>
<tr>
<th>AArch64 register</th>
<th>AArch32 register</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPIDR_EL2[31:0]</td>
<td>VPIDR</td>
</tr>
<tr>
<td>VTCR_EL2[31:0]</td>
<td>VTCR</td>
</tr>
<tr>
<td>VTTBR_EL2[63:0]</td>
<td>VTTBR</td>
</tr>
<tr>
<td><strong>Timer registers</strong></td>
<td></td>
</tr>
<tr>
<td>CNTFRQ_EL0</td>
<td>CNTFRQ</td>
</tr>
<tr>
<td>CNTHCTL_EL2</td>
<td>CNTHCTL</td>
</tr>
<tr>
<td>CNTHP_CTL_EL2</td>
<td>CNTP_CTL</td>
</tr>
<tr>
<td>CNTHP_CVAL_EL2</td>
<td>CNTHP_CVAL</td>
</tr>
<tr>
<td>CNTHP_TVAL_EL2</td>
<td>CNTHVS_TVAL</td>
</tr>
<tr>
<td>CNTHPS_CTL_EL2</td>
<td>CNTHPS_CTL</td>
</tr>
<tr>
<td>CNTHPS_CVAL_EL2</td>
<td>CNTHPS_CVAL</td>
</tr>
<tr>
<td>CNTHPS_TVAL_EL2</td>
<td>CNTHPS_TVAL</td>
</tr>
<tr>
<td>CNTKCTL_EL1</td>
<td>CNTKCTL</td>
</tr>
<tr>
<td>CNTP_CTL_EL0</td>
<td>CNTP_CTL^</td>
</tr>
<tr>
<td>CNTP_CVAL_EL0</td>
<td>CNTP_CVAL^</td>
</tr>
<tr>
<td>CNTP_TVAL_EL0</td>
<td>CNTP_TVAL^</td>
</tr>
<tr>
<td>CNTPCT_EL0[63:0]</td>
<td>CNTPCT</td>
</tr>
<tr>
<td>CNTV_CTL_EL0[31:0]</td>
<td>CNTV_CTL</td>
</tr>
<tr>
<td>CNTV_CVAL_EL0[63:0]</td>
<td>CNTV_CVAL</td>
</tr>
<tr>
<td>CNTV_TVAL_EL0[31:0]</td>
<td>CNTV_TVAL</td>
</tr>
<tr>
<td>CNTHV_CTL_EL2[63:0]</td>
<td>CNTHV_CTL</td>
</tr>
<tr>
<td>CNTHV_CVAL_EL2[63:0]</td>
<td>CNTHV_CVAL</td>
</tr>
<tr>
<td>CNTHV_TVAL_EL2[63:0]</td>
<td>CNTHV_TVAL</td>
</tr>
<tr>
<td>CNTHVS_CTL_EL2[31:0]</td>
<td>CNTHVS_CTL</td>
</tr>
<tr>
<td>CNTHVS_CVAL_EL2[63:0]</td>
<td>CNTHVS_CVAL</td>
</tr>
<tr>
<td>CNTHVS_TVAL_EL2[63:0]</td>
<td>CNTHVS_TVAL</td>
</tr>
<tr>
<td>CNTVCT_EL0[63:0]</td>
<td>CNTVCT</td>
</tr>
<tr>
<td>CNTVOFF_EL2[63:0]</td>
<td>CNTVOFF</td>
</tr>
<tr>
<td><strong>Debug System registers</strong></td>
<td></td>
</tr>
<tr>
<td>DBGAUTHSTATUS_EL1[31:0]</td>
<td>DBGAUTHSTATUS</td>
</tr>
<tr>
<td>DBGBCR&lt;n&gt;_EL1[31:0]</td>
<td>DBGBCR&lt;n&gt;</td>
</tr>
<tr>
<td>DBGBVAR&lt;n&gt;_EL1[31:0]</td>
<td>DBGBVAR&lt;n&gt;</td>
</tr>
</tbody>
</table>
# Registers Index

## K17.1 Introduction and register disambiguation

<table>
<thead>
<tr>
<th>AArch64 register</th>
<th>AArch32 register</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBGBVR&lt;n&gt;_EL1[63:32]</td>
<td>DBGBXVR&lt;n&gt;</td>
</tr>
<tr>
<td>DBGCLAIMCLR_EL1[31:0]</td>
<td>DBGCLAIMCLR</td>
</tr>
<tr>
<td>DBGCLAIMSET_EL1[31:0]</td>
<td>DBGCLAIMSET</td>
</tr>
<tr>
<td>DBGDTR_EL0[63:32]</td>
<td>DBGDTRRXint</td>
</tr>
<tr>
<td>DBGDTR_EL0[31:0]</td>
<td>DBGDTRTXint</td>
</tr>
<tr>
<td>DBGDTRRX_EL0[31:0]</td>
<td>DBGDTRRXint</td>
</tr>
<tr>
<td>DBGDTRTX_EL0[31:0]</td>
<td>DBGDTRTXint</td>
</tr>
<tr>
<td>DBGPRC_EL1[31:0]</td>
<td>DBGPRC</td>
</tr>
<tr>
<td>DBGVCR32_EL2[31:0]</td>
<td>DBGVCR</td>
</tr>
<tr>
<td>DBGWCR&lt;n&gt;_EL1[31:0]</td>
<td>DBGWCR&lt;n&gt;</td>
</tr>
<tr>
<td>DBGWVR&lt;n&gt;_EL1[31:0]</td>
<td>DBGWVR&lt;n&gt;</td>
</tr>
<tr>
<td>ID_DFR0_EL1[31:0]</td>
<td>ID_DFR0</td>
</tr>
<tr>
<td>MDCCSR_EL0b[30:29]</td>
<td>DBGDSCRintb</td>
</tr>
<tr>
<td>MDCR_EL2[31:0]</td>
<td>HDCR</td>
</tr>
<tr>
<td>MDRAR_EL1[63:0]</td>
<td>DBGDRAR</td>
</tr>
<tr>
<td>MDSRCR_EL1b[31:0]</td>
<td>DBGDSCRextb</td>
</tr>
<tr>
<td>OSDLR_EL1[31:0]</td>
<td>DBGOSDLR</td>
</tr>
<tr>
<td>OSDTRRX_EL1b[31:0]</td>
<td>DBGDTRRXextb</td>
</tr>
<tr>
<td>OSDTRTX_EL1b[31:0]</td>
<td>DBGDTRTXextb</td>
</tr>
<tr>
<td>OSECCR_EL1[31:0]</td>
<td>DBGSECCR</td>
</tr>
<tr>
<td>OSLAR_EL1[31:0]</td>
<td>DBGOSLR</td>
</tr>
<tr>
<td>OLSR_EL1[31:0]</td>
<td>DBGOSLSR</td>
</tr>
<tr>
<td>SDER32_EL3[31:0]</td>
<td>SDER</td>
</tr>
</tbody>
</table>

### Performance Monitors System registers

- PMCCNTR_EL0[31:0] | PMCCNTR (MRC/MCR)
- PMCEID0_EL0[31:0] | PMCEID0
- PMCEID0_EL0[63:32] | PMCEID2
- PMCEID1_EL0[31:0] | PMCEID1
- PMCEID1_EL0[63:32] | PMCEID3
- PMCNTRCLR_EL0[31:0] | PMCNTRCLR
- PMCNTRSET_EL0[31:0] | PMCNTRSET
- PMCR_EL0[31:0] | PMCR
### Table K17-5 Mapping of writable AArch64 System registers to the AArch32 System registers

<table>
<thead>
<tr>
<th>AArch64 register</th>
<th>AArch32 register</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMEVCNTR&lt;n&gt;_EL0</td>
<td>PMEVCNTR&lt;n&gt;</td>
</tr>
<tr>
<td>PMEVTYPE&lt;n&gt;_EL0</td>
<td>PMEVTYPE&lt;n&gt;</td>
</tr>
<tr>
<td>PMINTENCLR_EL1</td>
<td>PMINTENCLR</td>
</tr>
<tr>
<td>PMINTENSET_EL1</td>
<td>PMINTENSET</td>
</tr>
<tr>
<td>PMSELR_EL0</td>
<td>PMSELR</td>
</tr>
<tr>
<td>PMSWINC_EL0</td>
<td>PMSWINC</td>
</tr>
<tr>
<td>PMUSERENR_EL0</td>
<td>PMUSERENR</td>
</tr>
<tr>
<td>PMXEVNCTR_EL0</td>
<td>PMXEVNCTR</td>
</tr>
<tr>
<td>PMXEVTYPER_EL0</td>
<td>PMXEVTYPER</td>
</tr>
</tbody>
</table>

**Activity Monitors System registers**

<table>
<thead>
<tr>
<th>AArch64 register</th>
<th>AArch32 register</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMCNTENCLR0_EL0</td>
<td>AMCNTENCLR0</td>
</tr>
<tr>
<td>AMCNTENCLR1_EL0</td>
<td>AMEVCTR1&lt;n&gt;</td>
</tr>
<tr>
<td>AMCNTENSET0_EL0</td>
<td>AMCNTENSET0</td>
</tr>
<tr>
<td>AMCNTENSET1_EL0</td>
<td>AMCNTENSET1</td>
</tr>
<tr>
<td>AMCR_EL0</td>
<td>AMCR</td>
</tr>
<tr>
<td>AMEVCTR0&lt;n&gt;_EL0</td>
<td>AMEVCTR0&lt;n&gt;</td>
</tr>
<tr>
<td>AMEVCTR1&lt;n&gt;_EL0</td>
<td>AMEVCTR1&lt;n&gt;</td>
</tr>
<tr>
<td>AMEVCTR1&lt;n&gt;_EL0</td>
<td>AMEVCTR1&lt;n&gt;</td>
</tr>
</tbody>
</table>

**RAS System registers**

<table>
<thead>
<tr>
<th>AArch64 register</th>
<th>AArch32 register</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISR_EL1</td>
<td>DISR</td>
</tr>
<tr>
<td>ERRIDR_EL1</td>
<td>ERRIDR</td>
</tr>
<tr>
<td>ERRSELR_EL1</td>
<td>ERRSELR</td>
</tr>
<tr>
<td>ERXADDR_EL1</td>
<td>ERXADDR</td>
</tr>
<tr>
<td>ERXADDR_EL1[63:32]</td>
<td>ERXADDR2</td>
</tr>
<tr>
<td>ERXCTRLR_EL1</td>
<td>ERXCTRLR</td>
</tr>
<tr>
<td>ERXCTRLR_EL1[63:32]</td>
<td>ERXCTRLR2</td>
</tr>
<tr>
<td>ERXFR_EL1[63:32]</td>
<td>ERXFR2</td>
</tr>
<tr>
<td>ERXFR_EL1[63:32]</td>
<td>ERXFR2</td>
</tr>
<tr>
<td>ERXMISC0_EL1</td>
<td>ERXMISC0</td>
</tr>
<tr>
<td>ERXMISC0_EL1[63:32]</td>
<td>ERXMISC1</td>
</tr>
<tr>
<td>ERXMISC1_EL1</td>
<td>ERXMISC2</td>
</tr>
<tr>
<td>ERXMISC1_EL1[63:32]</td>
<td>ERXMISC3</td>
</tr>
</tbody>
</table>
There are a small number of AArch32 System registers that are not mapped to any AArch64 System registers. The AArch64 registers listed in Table K17-6 can be used to access these from a higher Exception level that is using AArch64. The registers shown in the table are UNDEFINED if EL1 cannot use AArch32.

Table K17-6 AArch64 registers for accessing registers that are only used in AArch32 state

<table>
<thead>
<tr>
<th>AArch32 register</th>
<th>Register for access from AArch64 state</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DACR</td>
<td>DACR32_EL2</td>
<td>Domain Access Control Register</td>
</tr>
<tr>
<td>DBGVCR</td>
<td>DBGVCR32_EL2</td>
<td>Debug Vector Catch Register</td>
</tr>
<tr>
<td>FPEXC</td>
<td>FPEXC32_EL2</td>
<td>Floating-Point Exception Control Register</td>
</tr>
<tr>
<td>IFSR</td>
<td>IFSR32_EL2</td>
<td>Instruction Fault Status Register</td>
</tr>
<tr>
<td>SDER</td>
<td>SDER32_EL3</td>
<td>AArch32 Secure Debug Enable Register</td>
</tr>
</tbody>
</table>

Table K17-7 shows the AArch64 System registers that allow access from AArch64 state to the AArch32 ID registers. These AArch64 registers are UNKNOWN if no Exception level can use AArch32.

Table K17-7 AArch64 registers that access the AArch32 ID registers

<table>
<thead>
<tr>
<th>AArch32 register</th>
<th>Register for access from AArch64 state</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID_AFR0</td>
<td>ID_AFR0_EL1</td>
<td>AArch32 Auxiliary Feature Register 0</td>
</tr>
<tr>
<td>ID_DFR0</td>
<td>ID_DFR0_EL1</td>
<td>AArch32 Debug Feature Register 0</td>
</tr>
<tr>
<td>ID_ISAR0</td>
<td>ID_ISAR0_EL1</td>
<td>EL1, AArch32 Instruction Set Attribute Register 0</td>
</tr>
<tr>
<td>ID_ISAR1</td>
<td>ID_ISAR1_EL1</td>
<td>EL1, AArch32 Instruction Set Attribute Register 1</td>
</tr>
<tr>
<td>ID_ISAR2</td>
<td>ID_ISAR2_EL1</td>
<td>EL1, AArch32 Instruction Set Attribute Register 2</td>
</tr>
<tr>
<td>ID_MMFR3</td>
<td>ID_MMFR3_EL1</td>
<td>EL1, AArch32 Instruction Set Attribute Register 3</td>
</tr>
<tr>
<td>ID_ISAR4</td>
<td>ID_ISAR4_EL1</td>
<td>EL1, AArch32 Instruction Set Attribute Register 4</td>
</tr>
<tr>
<td>ID_ISAR5</td>
<td>ID_ISAR5_EL1</td>
<td>EL1, AArch32 Instruction Set Attribute Register 5</td>
</tr>
</tbody>
</table>
### K17.1.2 Register name disambiguation by Exception level

Table K17-8 on page K17-11846 disambiguates the general names of the AArch64 System registers by Exception level.

<table>
<thead>
<tr>
<th>General form</th>
<th>EL0</th>
<th>EL1</th>
<th>EL2</th>
<th>EL3</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFSR0_ELx</td>
<td>-</td>
<td>AFSR0_EL1</td>
<td>AFSR0_EL2</td>
<td>AFSR0_EL3</td>
</tr>
<tr>
<td>AFSR1_ELx</td>
<td>-</td>
<td>AFSR1_EL1</td>
<td>AFSR1_EL2</td>
<td>AFSR1_EL3</td>
</tr>
<tr>
<td>CONTEXTIDR_ELx</td>
<td>-</td>
<td>CONTEXTIDR_EL1</td>
<td>CONTEXTIDR_EL2</td>
<td>-</td>
</tr>
<tr>
<td>CPTR_ELx</td>
<td>-</td>
<td>-</td>
<td>CPTR_EL2</td>
<td>CPTR_EL3</td>
</tr>
<tr>
<td>ELR_ELx</td>
<td>-</td>
<td>ELR_EL1</td>
<td>ELR_EL2</td>
<td>ELR_EL3</td>
</tr>
<tr>
<td>ESR_ELx</td>
<td>-</td>
<td>ESR_EL1</td>
<td>ESR_EL2</td>
<td>ESR_EL3</td>
</tr>
<tr>
<td>FAR_ELx</td>
<td>-</td>
<td>FAR_EL1</td>
<td>FAR_EL2</td>
<td>FAR_EL3</td>
</tr>
<tr>
<td>MAIR_ELx</td>
<td>-</td>
<td>MAIR_EL1</td>
<td>MAIR_EL2</td>
<td>MAIR_EL3</td>
</tr>
<tr>
<td>RMR_ELx</td>
<td>-</td>
<td>RMR_EL1</td>
<td>RMR_EL2</td>
<td>RMR_EL3</td>
</tr>
<tr>
<td>RVBAR_ELx</td>
<td>-</td>
<td>RVBAR_EL1</td>
<td>RVBAR_EL2</td>
<td>RVBAR_EL3</td>
</tr>
<tr>
<td>SCTLR_ELx</td>
<td>-</td>
<td>SCTLR_EL1</td>
<td>SCTLR_EL2</td>
<td>SCTLR_EL3</td>
</tr>
<tr>
<td>SCXTNUM_ELx</td>
<td>SCXTNUM_EL0</td>
<td>SCXTNUM_EL1</td>
<td>SCXTNUM_EL2</td>
<td>SCXTNUM_EL3</td>
</tr>
<tr>
<td>SP_ELx</td>
<td>SP_EL0</td>
<td>SP_EL1</td>
<td>SP_EL2</td>
<td>SP_EL3</td>
</tr>
<tr>
<td>SPSR_ELx</td>
<td>-</td>
<td>SPSR_EL1</td>
<td>SPSR_EL2</td>
<td>SPSR_EL3</td>
</tr>
<tr>
<td>TCR_ELx</td>
<td>-</td>
<td>TCR_EL1</td>
<td>TCR_EL2</td>
<td>TCR_EL3</td>
</tr>
<tr>
<td>TFSR_ELx</td>
<td>TFSRE0_EL1</td>
<td>TFSR_EL1</td>
<td>TFSR_EL2</td>
<td>TFSR_EL3</td>
</tr>
<tr>
<td>TTBR0_ELx</td>
<td>-</td>
<td>TTBR0_EL1</td>
<td>TTBR0_EL2</td>
<td>TTBR0_EL3</td>
</tr>
</tbody>
</table>
#### Table K17-8 Disambiguation of AArch64 System registers by Exception level (continued)

<table>
<thead>
<tr>
<th>General form</th>
<th>EL0</th>
<th>EL1</th>
<th>EL2</th>
<th>EL3</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTBR1_ELx</td>
<td>-</td>
<td>TTBR1_EL1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VBAR_ELx</td>
<td>-</td>
<td>VBAR_EL1</td>
<td>VBAR_EL2</td>
<td>VBAR_EL3</td>
</tr>
<tr>
<td>ZCR_ELx</td>
<td>-</td>
<td>ZCR_EL1</td>
<td>ZCR_EL2</td>
<td>ZCR_EL3</td>
</tr>
</tbody>
</table>
K17.2  **Alphabetical index of AArch64 registers and System instructions**

This section is an index of AArch64 registers and System instructions in alphabetical order.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCDATA_EL1</td>
<td>ACCDATA_EL1, Accelerator Data on page D17-5556</td>
</tr>
<tr>
<td>ACTLR_EL1</td>
<td>ACTLR_EL1, Auxiliary Control Register (EL1) on page D17-5558</td>
</tr>
<tr>
<td>ACTLR_EL2</td>
<td>ACTLR_EL2, Auxiliary Control Register (EL2) on page D17-5560</td>
</tr>
<tr>
<td>ACTLR_EL3</td>
<td>ACTLR_EL3, Auxiliary Control Register (EL3) on page D17-5562</td>
</tr>
<tr>
<td>AFSR0_EL1</td>
<td>AFSR0_EL1, Auxiliary Fault Status Register 0 (EL1) on page D17-5564</td>
</tr>
<tr>
<td>AFSR0_EL2</td>
<td>AFSR0_EL2, Auxiliary Fault Status Register 0 (EL2) on page D17-5567</td>
</tr>
<tr>
<td>AFSR0_EL3</td>
<td>AFSR0_EL3, Auxiliary Fault Status Register 0 (EL3) on page D17-5570</td>
</tr>
<tr>
<td>AFSR1_EL1</td>
<td>AFSR1_EL1, Auxiliary Fault Status Register 1 (EL1) on page D17-5572</td>
</tr>
<tr>
<td>AFSR1_EL2</td>
<td>AFSR1_EL2, Auxiliary Fault Status Register 1 (EL2) on page D17-5575</td>
</tr>
<tr>
<td>AFSR1_EL3</td>
<td>AFSR1_EL3, Auxiliary Fault Status Register 1 (EL3) on page D17-5578</td>
</tr>
<tr>
<td>AIDR_EL1</td>
<td>AIDR_EL1, Auxiliary ID Register on page D17-5580</td>
</tr>
<tr>
<td>ALLINT</td>
<td>ALLINT, All Interrupt Mask Bit on page C5-672</td>
</tr>
<tr>
<td>AMAIR_EL1</td>
<td>AMAIR_EL1, Auxiliary Memory Attribute Indirection Register (EL1) on page D17-5581</td>
</tr>
<tr>
<td>AMAIR_EL2</td>
<td>AMAIR_EL2, Auxiliary Memory Attribute Indirection Register (EL2) on page D17-5584</td>
</tr>
<tr>
<td>AMAIR_EL3</td>
<td>AMAIR_EL3, Auxiliary Memory Attribute Indirection Register (EL3) on page D17-5587</td>
</tr>
<tr>
<td>AMCFGR_EL0</td>
<td>AMCFGR_EL0, Activity Monitors Configuration Register on page D17-6817</td>
</tr>
<tr>
<td>AMCG1IDR_EL0</td>
<td>AMCG1IDR_EL0, Activity Monitors Counter Group 1 Identification Register on page D17-6820</td>
</tr>
<tr>
<td>AMCGCR_EL0</td>
<td>AMCGCR_EL0, Activity Monitors Counter Group Configuration Register on page D17-6822</td>
</tr>
<tr>
<td>AMCNTENCLR0_EL0</td>
<td>AMCNTENCLR0_EL0, Activity Monitors Count Enable Clear Register 0 on page D17-6824</td>
</tr>
<tr>
<td>AMCNTENCLR1_EL0</td>
<td>AMCNTENCLR1_EL0, Activity Monitors Count Enable Clear Register 1 on page D17-6827</td>
</tr>
<tr>
<td>AMCNTENSET0_EL0</td>
<td>AMCNTENSET0_EL0, Activity Monitors Count Enable Set Register 0 on page D17-6830</td>
</tr>
<tr>
<td>AMCNTENSET1_EL0</td>
<td>AMCNTENSET1_EL0, Activity Monitors Count Enable Set Register 1 on page D17-6833</td>
</tr>
<tr>
<td>AMCR_EL0</td>
<td>AMCR_EL0, Activity Monitors Control Register on page D17-6836</td>
</tr>
<tr>
<td>AMEVCTR0&lt;\n&gt;EL0</td>
<td>AMEVCTR0&lt;\n&gt;EL0, Activity Monitors Event Counter Registers 0, n = 0 - 3 on page D17-6839</td>
</tr>
<tr>
<td>AMEVCTR1&lt;\n&gt;EL0</td>
<td>AMEVCTR1&lt;\n&gt;EL0, Activity Monitors Event Counter Registers 1, n = 0 - 15 on page D17-6842</td>
</tr>
<tr>
<td>AMEVCTVOFF0&lt;\n&gt;EL2</td>
<td>AMEVCTVOFF0&lt;\n&gt;EL2, Activity Monitors Event Counter Virtual Offset Registers 0, n = 0 - 15 on page D17-6845</td>
</tr>
<tr>
<td>Register Description, see page</td>
<td>Page Numbers</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>AMEVCNTVOFF1&lt;n&gt;_EL2</td>
<td>D17-6847</td>
</tr>
<tr>
<td>AMEVTYPER0&lt;n&gt;_EL0</td>
<td>D17-6850</td>
</tr>
<tr>
<td>AMEVTYPER1&lt;n&gt;_EL0</td>
<td>D17-6852</td>
</tr>
<tr>
<td>AMUSERENR_EL0</td>
<td>D17-6855</td>
</tr>
<tr>
<td>APDAKeyHi_EL1</td>
<td>D17-5589</td>
</tr>
<tr>
<td>APDAKeyLo_EL1</td>
<td>D17-5591</td>
</tr>
<tr>
<td>APDBKeyHi_EL1</td>
<td>D17-5593</td>
</tr>
<tr>
<td>APDBKeyLo_EL1</td>
<td>D17-5595</td>
</tr>
<tr>
<td>APGAKeyHi_EL1</td>
<td>D17-5597</td>
</tr>
<tr>
<td>APGAKeyLo_EL1</td>
<td>D17-5599</td>
</tr>
<tr>
<td>APIAKeyHi_EL1</td>
<td>D17-5601</td>
</tr>
<tr>
<td>APIAKeyLo_EL1</td>
<td>D17-5603</td>
</tr>
<tr>
<td>APIBKeyHi_EL1</td>
<td>D17-5605</td>
</tr>
<tr>
<td>APIBKeyLo_EL1</td>
<td>D17-5607</td>
</tr>
<tr>
<td>AT S12E0R</td>
<td>C5-842</td>
</tr>
<tr>
<td>AT S12E0W</td>
<td>C5-844</td>
</tr>
<tr>
<td>AT S12E1R</td>
<td>C5-846</td>
</tr>
<tr>
<td>AT S12E1W</td>
<td>C5-848</td>
</tr>
<tr>
<td>AT S1E0R</td>
<td>C5-850</td>
</tr>
<tr>
<td>AT S1E0W</td>
<td>C5-852</td>
</tr>
<tr>
<td>AT S1E1R</td>
<td>C5-854</td>
</tr>
<tr>
<td>AT S1E1RP</td>
<td>C5-856</td>
</tr>
<tr>
<td>AT S1E1W</td>
<td>C5-858</td>
</tr>
<tr>
<td>AT S1E1WP</td>
<td>C5-860</td>
</tr>
<tr>
<td>AT S1E2R</td>
<td>C5-862</td>
</tr>
<tr>
<td>AT S1E2W</td>
<td>C5-863</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>AT S1E3R</td>
<td>AT S1E3R, Address Translate Stage 1 EL3 Read on page C5-864</td>
</tr>
<tr>
<td>AT S1E3W</td>
<td>AT S1E3W, Address Translate Stage 1 EL3 Write on page C5-865</td>
</tr>
<tr>
<td>BRB IALL</td>
<td>BRB IALL, Invalidate the Branch Record Buffer on page C5-1160</td>
</tr>
<tr>
<td>BRB INJ</td>
<td>BRB INJ, Branch Record Injection into the Branch Record Buffer on page C5-1162</td>
</tr>
<tr>
<td>BRBCR_EL1</td>
<td>BRBCR_EL1, Branch Record Buffer Control Register (EL1) on page D17-6913</td>
</tr>
<tr>
<td>BRBCR_EL2</td>
<td>BRBCR_EL2, Branch Record Buffer Control Register (EL2) on page D17-6919</td>
</tr>
<tr>
<td>BRBFCR_EL1</td>
<td>BRBFCR_EL1, Branch Record Buffer Function Control Register on page D17-6925</td>
</tr>
<tr>
<td>BRBIDR0_EL1</td>
<td>BRBIDR0_EL1, Branch Record Buffer ID0 Register on page D17-6930</td>
</tr>
<tr>
<td>BRBINFINJ_EL1</td>
<td>BRBINFINJ_EL1, Branch Record Buffer Information Injection Register on page D17-6937</td>
</tr>
<tr>
<td>BRBINF&lt;n&gt;_EL1</td>
<td>BRBINF&lt;n&gt;_EL1, Branch Record Buffer Information Register &lt;n&gt;, n = 0 - 31 on page D17-6932</td>
</tr>
<tr>
<td>BRBSRCINJ_EL1</td>
<td>BRBSRCINJ_EL1, Branch Record Buffer Source Address Injection Register on page D17-6945</td>
</tr>
<tr>
<td>BRBSRC&lt;n&gt;_EL1</td>
<td>BRBSRC&lt;n&gt;_EL1, Branch Record Buffer Source Address Register &lt;n&gt;, n = 0 - 31 on page D17-6943</td>
</tr>
<tr>
<td>BRBTGTINJ_EL1</td>
<td>BRBTGTINJ_EL1, Branch Record Buffer Target Address Injection Register on page D17-6950</td>
</tr>
<tr>
<td>BRBTGT&lt;n&gt;_EL1</td>
<td>BRBTGT&lt;n&gt;_EL1, Branch Record Buffer Target Address Register &lt;n&gt;, n = 0 - 31 on page D17-6948</td>
</tr>
<tr>
<td>BRBTS_EL1</td>
<td>BRBTS_EL1, Branch Record Buffer Timestamp Register on page D17-6953</td>
</tr>
<tr>
<td>CCSIDR_EL1</td>
<td>CCSIDR_EL1, Current Cache Size ID Register on page D17-5611</td>
</tr>
<tr>
<td>CCSIDR2_EL1</td>
<td>CCSIDR2_EL1, Current Cache Size ID Register 2 on page D17-5609</td>
</tr>
<tr>
<td>CFP RCTX</td>
<td>CFP RCTX, Control Flow Prediction Restriction by Context on page C5-1147</td>
</tr>
<tr>
<td>CLIDR_EL1</td>
<td>CLIDR_EL1, Cache Level ID Register on page D17-5614</td>
</tr>
<tr>
<td>CNTFRQ_EL0</td>
<td>CNTFRQ_EL0, Counter-timer Frequency register on page D17-7015</td>
</tr>
<tr>
<td>CNTHTCTL_EL2</td>
<td>CNTHTCTL_EL2, Counter-timer Hypervisor Control register on page D17-7017</td>
</tr>
<tr>
<td>CNTHP_CTL_EL2</td>
<td>CNTHP_CTL_EL2, Counter-timer Hypervisor Physical Timer Control register on page D17-7029</td>
</tr>
<tr>
<td>CNTHP_CVAL_EL2</td>
<td>CNTHP_CVAL_EL2, Counter-timer Physical Timer CompareValue register (EL2) on page D17-7033</td>
</tr>
<tr>
<td>CNTHP_TVAL_EL2</td>
<td>CNTHP_TVAL_EL2, Counter-timer Physical Timer TimerValue register (EL2) on page D17-7037</td>
</tr>
<tr>
<td>CNTHPS_CTL_EL2</td>
<td>CNTHPS_CTL_EL2, Counter-timer Secure Physical Timer Control register (EL2) on page D17-7041</td>
</tr>
<tr>
<td>CNTHPS_CVAL_EL2</td>
<td>CNTHPS_CVAL_EL2, Counter-timer Secure Physical Timer CompareValue register (EL2) on page D17-7045</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CNTHPS_TV_AL_EL2</td>
<td>CNTHPS_TV_AL_EL2, Counter-timer Secure Physical Timer Value register (EL2) on page D17-7049</td>
</tr>
<tr>
<td>CNTHV_CTL_EL2</td>
<td>CNTHV_CTL_EL2, Counter-timer Virtual Timer Control register (EL2) on page D17-7053</td>
</tr>
<tr>
<td>CNTHV_CV_AL_EL2</td>
<td>CNTHV_CV_AL_EL2, Counter-timer Virtual Timer CompareValue register (EL2) on page D17-7057</td>
</tr>
<tr>
<td>CNTHV_TV_AL_EL2</td>
<td>CNTHV_TV_AL_EL2, Counter-timer Virtual Timer TimerValue Register (EL2) on page D17-7060</td>
</tr>
<tr>
<td>CNTHVS_CTL_EL2</td>
<td>CNTHVS_CTL_EL2, Counter-timer Secure Virtual Timer Control register (EL2) on page D17-7064</td>
</tr>
<tr>
<td>CNTHVS_CV_AL_EL2</td>
<td>CNTHVS_CV_AL_EL2, Counter-timer Secure Virtual Timer CompareValue register (EL2) on page D17-7068</td>
</tr>
<tr>
<td>CNTHVS_TV_AL_EL2</td>
<td>CNTHVS_TV_AL_EL2, Counter-timer Secure Virtual Timer TimerValue register (EL2) on page D17-7071</td>
</tr>
<tr>
<td>CNTKCTL_EL1</td>
<td>CNTKCTL_EL1, Counter-timer Kernel Control register on page D17-7076</td>
</tr>
<tr>
<td>CNTP_CTL_EL0</td>
<td>CNTP_CTL_EL0, Counter-timer Physical Timer Control register on page D17-7081</td>
</tr>
<tr>
<td>CNTP_CV_AL_EL0</td>
<td>CNTP_CV_AL_EL0, Counter-timer Physical Timer CompareValue register on page D17-7085</td>
</tr>
<tr>
<td>CNTP_TV_AL_EL0</td>
<td>CNTP_TV_AL_EL0, Counter-timer Physical Timer TimerValue register on page D17-7089</td>
</tr>
<tr>
<td>CNTPCT_EL0</td>
<td>CNTPCT_EL0, Counter-timer Physical Count register on page D17-7095</td>
</tr>
<tr>
<td>CNTPCTSS_EL0</td>
<td>CNTPCTSS_EL0, Counter-timer Self-Synchronized Physical Count register on page D17-7093</td>
</tr>
<tr>
<td>CNTPOFF_EL2</td>
<td>CNTPOFF_EL2, Counter-timer Physical Offset register on page D17-7100</td>
</tr>
<tr>
<td>CNTPS_CTL_EL1</td>
<td>CNTPS_CTL_EL1, Counter-timer Physical Secure Timer Control register on page D17-7097</td>
</tr>
<tr>
<td>CNTPS_CV_AL_EL1</td>
<td>CNTPS_CV_AL_EL1, Counter-timer Physical Secure Timer CompareValue register on page D17-7102</td>
</tr>
<tr>
<td>CNTPS_TV_AL_EL1</td>
<td>CNTPS_TV_AL_EL1, Counter-timer Physical Secure Timer TimerValue register on page D17-7104</td>
</tr>
<tr>
<td>CNTV_CTL_EL0</td>
<td>CNTV_CTL_EL0, Counter-timer Virtual Timer Control register on page D17-7106</td>
</tr>
<tr>
<td>CNTV_CV_AL_EL0</td>
<td>CNTV_CV_AL_EL0, Counter-timer Virtual Timer CompareValue register on page D17-7110</td>
</tr>
<tr>
<td>CNTV_TV_AL_EL0</td>
<td>CNTV_TV_AL_EL0, Counter-timer Virtual Timer TimerValue register on page D17-7114</td>
</tr>
<tr>
<td>CNTVCT_EL0</td>
<td>CNTVCT_EL0, Counter-timer Virtual Count register on page D17-7120</td>
</tr>
<tr>
<td>CNTVCTSS_EL0</td>
<td>CNTVCTSS_EL0, Counter-timer Self-Synchronized Virtual Count register on page D17-7118</td>
</tr>
<tr>
<td>CNTVOFF_EL2</td>
<td>CNTVOFF_EL2, Counter-timer Virtual Offset register on page D17-7122</td>
</tr>
<tr>
<td>CONTEXTIDR_EL1</td>
<td>CONTEXTIDR_EL1, Context ID Register (EL1) on page D17-5617</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CONTEXTIDR_EL2</td>
<td>CONTEXTIDR_EL2, Context ID Register (EL2) on page D17-5620</td>
</tr>
<tr>
<td>CPACR_EL1</td>
<td>CPACR_EL1, Architectural Feature Access Control Register on page D17-5623</td>
</tr>
<tr>
<td>CPP RCTX</td>
<td>CPP RCTX, Cache Prefetch Prediction Restriction by Context on page C5-1151</td>
</tr>
<tr>
<td>CPTTR_EL2</td>
<td>CPTTR_EL2, Architectural Feature Trap Register (EL2) on page D17-5629</td>
</tr>
<tr>
<td>CPTTR_EL3</td>
<td>CPTTR_EL3, Architectural Feature Trap Register (EL3) on page D17-5639</td>
</tr>
<tr>
<td>CSSELR_EL1</td>
<td>CSSELR_EL1, Cache Size Selection Register on page D17-5644</td>
</tr>
<tr>
<td>CTR_EL0</td>
<td>CTR_EL0, Cache Type Register on page D17-5647</td>
</tr>
<tr>
<td>CurrentEL</td>
<td>CurrentEL, Current Exception Level on page C5-674</td>
</tr>
<tr>
<td>DACK32_EL2</td>
<td>DACK32_EL2, Domain Access Control Register on page D17-5650</td>
</tr>
<tr>
<td>DAIIF</td>
<td>DAIIF, Interrupt Mask Bits on page C5-676</td>
</tr>
<tr>
<td>DBGAUTHSTATUS_EL1</td>
<td>DBGAUTHSTATUS_EL1, Debug Authentication Status register on page D17-6394</td>
</tr>
<tr>
<td>DBGBCR&lt;n&gt;_EL1</td>
<td>DBGBCR&lt;n&gt;_EL1, Debug Breakpoint Control Registers, n = 0 - 15 on page D17-6397</td>
</tr>
<tr>
<td>DBGBVRO&lt;n&gt;_EL1</td>
<td>DBGBVRO&lt;n&gt;_EL1, Debug Breakpoint Value Registers, n = 0 - 15 on page D17-6402</td>
</tr>
<tr>
<td>DBGCLAIMCLR_EL1</td>
<td>DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register on page D17-6408</td>
</tr>
<tr>
<td>DBGCLAIMSET_EL1</td>
<td>DBGCLAIMSET_EL1, Debug CLAIM Tag Set register on page D17-6411</td>
</tr>
<tr>
<td>DBGDTR_EL0</td>
<td>DBGDTR_EL0, Debug Data Transfer Register, half-duplex on page D17-6414</td>
</tr>
<tr>
<td>DBGDTRRX_EL0</td>
<td>DBGDTRRX_EL0, Debug Data Transfer Register, Receive on page D17-6417</td>
</tr>
<tr>
<td>DBGDTRTX_EL0</td>
<td>DBGDTRTX_EL0, Debug Data Transfer Register, Transmit on page D17-6419</td>
</tr>
<tr>
<td>DBGPRCR_EL1</td>
<td>DBGPRCR_EL1, Debug Power Control Register on page D17-6421</td>
</tr>
<tr>
<td>DBGVCR32_EL2</td>
<td>DBGVCR32_EL2, Debug Vector Catch Register on page D17-6424</td>
</tr>
<tr>
<td>DBGWCR&lt;n&gt;_EL1</td>
<td>DBGWCR&lt;n&gt;_EL1, Debug Watchpoint Control Registers, n = 0 - 15 on page D17-6429</td>
</tr>
<tr>
<td>DBGWVRO&lt;n&gt;_EL1</td>
<td>DBGWVRO&lt;n&gt;_EL1, Debug Watchpoint Value Registers, n = 0 - 15 on page D17-6434</td>
</tr>
<tr>
<td>DC CGDSW</td>
<td>DC CGDSW, Clean of Data and Allocation Tags by Set/Way on page C5-777</td>
</tr>
<tr>
<td>DC CGDVAC</td>
<td>DC CGDVAC, Clean of Data and Allocation Tags by VA to PoC on page C5-779</td>
</tr>
<tr>
<td>DC CGDVADP</td>
<td>DC CGDVADP, Clean of Data and Allocation Tags by VA to PoDP on page C5-781</td>
</tr>
<tr>
<td>DC CGDVAP</td>
<td>DC CGDVAP, Clean of Data and Allocation Tags by VA to PoP on page C5-783</td>
</tr>
<tr>
<td>DC CGSW</td>
<td>DC CGSW, Clean of Allocation Tags by Set/Way on page C5-785</td>
</tr>
<tr>
<td>DC CGVAC</td>
<td>DC CGVAC, Clean of Allocation Tags by VA to PoC on page C5-787</td>
</tr>
<tr>
<td>DC CGVADP</td>
<td>DC CGVADP, Clean of Allocation Tags by VA to PoDP on page C5-789</td>
</tr>
<tr>
<td>DC CGVAP</td>
<td>DC CGVAP, Clean of Allocation Tags by VA to PoP on page C5-791</td>
</tr>
<tr>
<td>DC CIGDPAPA</td>
<td>DC CIGDPAPA, Clean and Invalidate of Data and Allocation Tags by PA to PoPA on page C5-793</td>
</tr>
</tbody>
</table>
### Table K17-9 Alphabetical index of AArch64 Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC CIGDSW</td>
<td><strong>DC CIGDSW</strong>, Clean and Invalidate of Data and Allocation Tags by Set/Way on page C5-795</td>
</tr>
<tr>
<td>DC CIGDVAC</td>
<td><strong>DC CIGDVAC</strong>, Clean and Invalidate of Data and Allocation Tags by VA to PoC on page C5-797</td>
</tr>
<tr>
<td>DC CIGSW</td>
<td><strong>DC CIGSW</strong>, Clean and Invalidate of Allocation Tags by Set/Way on page C5-799</td>
</tr>
<tr>
<td>DC CIGVAC</td>
<td><strong>DC CIGVAC</strong>, Clean and Invalidate of Allocation Tags by VA to PoC on page C5-801</td>
</tr>
<tr>
<td>DC CIPAPA</td>
<td><strong>DC CIPAPA</strong>, Data or unified Cache line Clean and Invalidate by PA to PoPA on page C5-803</td>
</tr>
<tr>
<td>DC CISW</td>
<td><strong>DC CISW</strong>, Data or unified Cache line Clean and Invalidate by Set/Way on page C5-805</td>
</tr>
<tr>
<td>DC CIVAC</td>
<td><strong>DC CIVAC</strong>, Data or unified Cache line Clean and Invalidate by VA to PoC on page C5-807</td>
</tr>
<tr>
<td>DC CSW</td>
<td><strong>DC CSW</strong>, Data or unified Cache line Clean by Set/Way on page C5-809</td>
</tr>
<tr>
<td>DC CVAC</td>
<td><strong>DC CVAC</strong>, Data or unified Cache line Clean by VA to PoC on page C5-811</td>
</tr>
<tr>
<td>DC CVADP</td>
<td><strong>DC CVADP</strong>, Data or unified Cache line Clean by VA to PoDP on page C5-813</td>
</tr>
<tr>
<td>DC CVAP</td>
<td><strong>DC CVAP</strong>, Data or unified Cache line Clean by VA to PoP on page C5-815</td>
</tr>
<tr>
<td>DC CVAU</td>
<td><strong>DC CVAU</strong>, Data or unified Cache line Clean by VA to PoU on page C5-817</td>
</tr>
<tr>
<td>DC GVA</td>
<td><strong>DC GVA</strong>, Data Cache set Allocation Tag by VA on page C5-819</td>
</tr>
<tr>
<td>DC GZVA</td>
<td><strong>DC GZVA</strong>, Data Cache set Allocation Tags and Zero by VA on page C5-821</td>
</tr>
<tr>
<td>DC IGDSW</td>
<td><strong>DC IGDSW</strong>, Invalidate of Data and Allocation Tags by Set/Way on page C5-823</td>
</tr>
<tr>
<td>DC IGDVAC</td>
<td><strong>DC IGDVAC</strong>, Invalidate of Data and Allocation Tags by VA to PoC on page C5-825</td>
</tr>
<tr>
<td>DC IGSW</td>
<td><strong>DC IGSW</strong>, Invalidate of Allocation Tags by Set/Way on page C5-827</td>
</tr>
<tr>
<td>DC IGVAC</td>
<td><strong>DC IGVAC</strong>, Invalidate of Allocation Tags by VA to PoC on page C5-829</td>
</tr>
<tr>
<td>DC ISW</td>
<td><strong>DC ISW</strong>, Data or unified Cache line Invalidate by Set/Way on page C5-831</td>
</tr>
<tr>
<td>DC IVAC</td>
<td><strong>DC IVAC</strong>, Data or unified Cache line Invalidate by VA to PoC on page C5-833</td>
</tr>
<tr>
<td>DC ZVA</td>
<td><strong>DC ZVA</strong>, Data Cache Zero by VA on page C5-835</td>
</tr>
<tr>
<td>DCZID_EL0</td>
<td><strong>DCZID_EL0</strong>, Data Cache Zero ID register on page D17-5652</td>
</tr>
<tr>
<td>DISR_EL1</td>
<td><strong>DISR_EL1</strong>, Deferred Interrupt Status Register on page D17-6957</td>
</tr>
<tr>
<td>DIT</td>
<td><strong>DIT</strong>, Data Independent Timing on page C5-679</td>
</tr>
<tr>
<td>DLR_EL0</td>
<td><strong>DLR_EL0</strong>, Debug Link Register on page D17-6437</td>
</tr>
<tr>
<td>DSPSR_EL0</td>
<td><strong>DSPSR_EL0</strong>, Debug Saved Program Status Register on page D17-6438</td>
</tr>
<tr>
<td>DVP RCTX</td>
<td><strong>DVP RCTX</strong>, Data Value Prediction Restriction by Context on page C5-1155</td>
</tr>
<tr>
<td>ELR_EL1</td>
<td><strong>ELR_EL1</strong>, Exception Link Register (EL1) on page C5-682</td>
</tr>
<tr>
<td>ELR_EL2</td>
<td><strong>ELR_EL2</strong>, Exception Link Register (EL2) on page C5-686</td>
</tr>
<tr>
<td>ELR_EL3</td>
<td><strong>ELR_EL3</strong>, Exception Link Register (EL3) on page C5-689</td>
</tr>
</tbody>
</table>
### Table K17-9 Alphabetical index of AArch64 Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERRIDR_EL1</td>
<td>ERRIDR_EL1, Error Record ID Register on page D17-6960</td>
</tr>
<tr>
<td>ERRSELR_EL1</td>
<td>ERRSELR_EL1, Error Record Select Register on page D17-6962</td>
</tr>
<tr>
<td>ERXADDR_EL1</td>
<td>ERXADDR_EL1, Selected Error Record Address Register on page D17-6965</td>
</tr>
<tr>
<td>ERXCTRLR_EL1</td>
<td>ERXCTRLR_EL1, Selected Error Record Control Register on page D17-6968</td>
</tr>
<tr>
<td>ERXF_EL1</td>
<td>ERXF_EL1, Selected Error Record Feature Register on page D17-6971</td>
</tr>
<tr>
<td>ERXMISC0_EL1</td>
<td>ERXMISC0_EL1, Selected Error Record Miscellaneous Register 0 on page D17-6973</td>
</tr>
<tr>
<td>ERXMISC1_EL1</td>
<td>ERXMISC1_EL1, Selected Error Record Miscellaneous Register 1 on page D17-6976</td>
</tr>
<tr>
<td>ERXMISC2_EL1</td>
<td>ERXMISC2_EL1, Selected Error Record Miscellaneous Register 2 on page D17-6979</td>
</tr>
<tr>
<td>ERXMISC3_EL1</td>
<td>ERXMISC3_EL1, Selected Error Record Miscellaneous Register 3 on page D17-6982</td>
</tr>
<tr>
<td>ERXPFGCDN_EL1</td>
<td>ERXPFGCDN_EL1, Selected Pseudo-fault Generation Countdown register on page D17-6985</td>
</tr>
<tr>
<td>ERXPFGCTL_EL1</td>
<td>ERXPFGCTL_EL1, Selected Pseudo-fault Generation Control register on page D17-6988</td>
</tr>
<tr>
<td>ERXPFGF_EL1</td>
<td>ERXPFGF_EL1, Selected Pseudo-fault Generation Feature register on page D17-6991</td>
</tr>
<tr>
<td>ERXSTATUS_EL1</td>
<td>ERXSTATUS_EL1, Selected Error Record Primary Status Register on page D17-6993</td>
</tr>
<tr>
<td>ESR_EL1</td>
<td>ESR_EL1, Exception Syndrome Register (EL1) on page D17-5654</td>
</tr>
<tr>
<td>ESR_EL2</td>
<td>ESR_EL2, Exception Syndrome Register (EL2) on page D17-5705</td>
</tr>
<tr>
<td>ESR_EL3</td>
<td>ESR_EL3, Exception Syndrome Register (EL3) on page D17-5759</td>
</tr>
<tr>
<td>FAR_EL1</td>
<td>FAR_EL1, Fault Address Register (EL1) on page D17-5807</td>
</tr>
<tr>
<td>FAR_EL2</td>
<td>FAR_EL2, Fault Address Register (EL2) on page D17-5812</td>
</tr>
<tr>
<td>FAR_EL3</td>
<td>FAR_EL3, Fault Address Register (EL3) on page D17-5816</td>
</tr>
<tr>
<td>FPCR</td>
<td>FPCR, Floating-point Control Register on page C5-691</td>
</tr>
<tr>
<td>FPEXC32_EL2</td>
<td>FPEXC32_EL2, Floating-Point Exception Control register on page D17-5819</td>
</tr>
<tr>
<td>FPSR</td>
<td>FPSR, Floating-point Status Register on page C5-700</td>
</tr>
<tr>
<td>GCR_EL1</td>
<td>GCR_EL1, Tag Control Register on page D17-5825</td>
</tr>
<tr>
<td>GMID_EL1</td>
<td>GMID_EL1, Multiple tag transfer ID register on page D17-5827</td>
</tr>
<tr>
<td>GPCCR_EL3</td>
<td>GPCCR_EL3, Granule Protection Check Control Register (EL3) on page D17-7005</td>
</tr>
<tr>
<td>GPTBR_EL3</td>
<td>GPTBR_EL3, Granule Protection Table Base Register on page D17-7009</td>
</tr>
<tr>
<td>HACR_EL2</td>
<td>HACR_EL2, Hypervisor Auxiliary Control Register on page D17-5828</td>
</tr>
<tr>
<td>HAFGRTR_EL2</td>
<td>HAFGRTR_EL2, Hypervisor Activity Monitors Fine-Grained Read Trap Register on page D17-5830</td>
</tr>
<tr>
<td>HCR_EL2</td>
<td>HCR_EL2, Hypervisor Configuration Register on page D17-5834</td>
</tr>
<tr>
<td>HCRX_EL2</td>
<td>HCRX_EL2, Extended Hypervisor Configuration Register on page D17-5866</td>
</tr>
<tr>
<td>HDFGRTR_EL2</td>
<td>HDFGRTR_EL2, Hypervisor Debug Fine-Grained Read Trap Register on page D17-5872</td>
</tr>
</tbody>
</table>
### Table K17-9 Alphabetical index of AArch64 Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFGWTR_EL2</td>
<td>HDFGWTR_EL2, Hypervisor Debug Fine-Grained Write Trap Register on page D17-5894</td>
</tr>
<tr>
<td>HFGITR_EL2</td>
<td>HFGITR_EL2, Hypervisor Fine-Grained Instruction Trap Register on page D17-5914</td>
</tr>
<tr>
<td>HFGRTR_EL2</td>
<td>HFGRTR_EL2, Hypervisor Fine-Grained Read Trap Register on page D17-5933</td>
</tr>
<tr>
<td>HFGWTR_EL2</td>
<td>HFGWTR_EL2, Hypervisor Fine-Grained Write Trap Register on page D17-5948</td>
</tr>
<tr>
<td>HPFAR_EL2</td>
<td>HPFAR_EL2, Hypervisor IPA Fault Address Register on page D17-5961</td>
</tr>
<tr>
<td>HSTR_EL2</td>
<td>HSTR_EL2, Hypervisor System Trap Register on page D17-5964</td>
</tr>
<tr>
<td>IC IALLU</td>
<td>IC IALLU, Instruction Cache Invalidate All to PoU on page C5-837</td>
</tr>
<tr>
<td>IC IALLUIS</td>
<td>IC IALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable on page C5-838</td>
</tr>
<tr>
<td>IC IVAU</td>
<td>IC IVAU, Instruction Cache line Invalidate by VA to PoU on page C5-839</td>
</tr>
<tr>
<td>ID_AA64AFR0_EL1</td>
<td>ID_AA64AFR0_EL1, AArch64 Auxiliary Feature Register 0 on page D17-5967</td>
</tr>
<tr>
<td>ID_AA64AFR1_EL1</td>
<td>ID_AA64AFR1_EL1, AArch64 Auxiliary Feature Register 1 on page D17-5969</td>
</tr>
<tr>
<td>ID_AA64DFR0_EL1</td>
<td>ID_AA64DFR0_EL1, AArch64 Debug Feature Register 0 on page D17-5970</td>
</tr>
<tr>
<td>ID_AA64DFR1_EL1</td>
<td>ID_AA64DFR1_EL1, AArch64 Debug Feature Register 1 on page D17-5975</td>
</tr>
<tr>
<td>ID_AA64ISAR0_EL1</td>
<td>ID_AA64ISAR0_EL1, AArch64 Instruction Set Attribute Register 0 on page D17-5976</td>
</tr>
<tr>
<td>ID_AA64ISAR1_EL1</td>
<td>ID_AA64ISAR1_EL1, AArch64 Instruction Set Attribute Register 1 on page D17-5981</td>
</tr>
<tr>
<td>ID_AA64ISAR2_EL1</td>
<td>ID_AA64ISAR2_EL1, AArch64 Instruction Set Attribute Register 2 on page D17-5987</td>
</tr>
<tr>
<td>ID_AA64MMFR0_EL1</td>
<td>ID_AA64MMFR0_EL1, AArch64 Memory Model Feature Register 0 on page D17-5990</td>
</tr>
<tr>
<td>ID_AA64MMFR1_EL1</td>
<td>ID_AA64MMFR1_EL1, AArch64 Memory Model Feature Register 1 on page D17-5994</td>
</tr>
<tr>
<td>ID_AA64MMFR2_EL1</td>
<td>ID_AA64MMFR2_EL1, AArch64 Memory Model Feature Register 2 on page D17-5999</td>
</tr>
<tr>
<td>ID_AA64PFR0_EL1</td>
<td>ID_AA64PFR0_EL1, AArch64 Processor Feature Register 0 on page D17-6004</td>
</tr>
<tr>
<td>ID_AA64PRF1_EL1</td>
<td>ID_AA64PRF1_EL1, AArch64 Processor Feature Register 1 on page D17-6009</td>
</tr>
<tr>
<td>ID_AA64SFR0_EL1</td>
<td>ID_AA64SFR0_EL1, SVE Feature ID register 0 on page D17-6013</td>
</tr>
<tr>
<td>ID_AA64ZFR0_EL1</td>
<td>ID_AA64ZFR0_EL1, SVE Feature ID register 0 on page D17-6016</td>
</tr>
<tr>
<td>ID_AFR0_EL1</td>
<td>ID_AFR0_EL1, AArch32 Auxiliary Feature Register 0 on page D17-6020</td>
</tr>
<tr>
<td>ID_DFR0_EL1</td>
<td>ID_DFR0_EL1, AArch32 Debug Feature Register 0 on page D17-6022</td>
</tr>
<tr>
<td>ID_DFR1_EL1</td>
<td>ID_DFR1_EL1, Debug Feature Register 1 on page D17-6026</td>
</tr>
<tr>
<td>ID_ISAR0_EL1</td>
<td>ID_ISAR0_EL1, AArch32 Instruction Set Attribute Register 0 on page D17-6028</td>
</tr>
<tr>
<td>ID_ISAR1_EL1</td>
<td>ID_ISAR1_EL1, AArch32 Instruction Set Attribute Register 1 on page D17-6031</td>
</tr>
<tr>
<td>ID_ISAR2_EL1</td>
<td>ID_ISAR2_EL1, AArch32 Instruction Set Attribute Register 2 on page D17-6034</td>
</tr>
<tr>
<td>ID_ISAR3_EL1</td>
<td>ID_ISAR3_EL1, AArch32 Instruction Set Attribute Register 3 on page D17-6037</td>
</tr>
<tr>
<td>ID_ISAR4_EL1</td>
<td>ID_ISAR4_EL1, AArch32 Instruction Set Attribute Register 4 on page D17-6040</td>
</tr>
<tr>
<td>ID_ISAR5_EL1</td>
<td>ID_ISAR5_EL1, AArch32 Instruction Set Attribute Register 5 on page D17-6043</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ID_ISAR6_EL1</td>
<td>ID_ISAR6_EL1, AArch32 Instruction Set Attribute Register 6 on page D17-6046</td>
</tr>
<tr>
<td>ID_MMFR0_EL1</td>
<td>ID_MMFR0_EL1, AArch32 Memory Model Feature Register 0 on page D17-6049</td>
</tr>
<tr>
<td>ID_MMFR1_EL1</td>
<td>ID_MMFR1_EL1, AArch32 Memory Model Feature Register 1 on page D17-6052</td>
</tr>
<tr>
<td>ID_MMFR2_EL1</td>
<td>ID_MMFR2_EL1, AArch32 Memory Model Feature Register 2 on page D17-6056</td>
</tr>
<tr>
<td>ID_MMFR3_EL1</td>
<td>ID_MMFR3_EL1, AArch32 Memory Model Feature Register 3 on page D17-6060</td>
</tr>
<tr>
<td>ID_MMFR4_EL1</td>
<td>ID_MMFR4_EL1, AArch32 Memory Model Feature Register 4 on page D17-6064</td>
</tr>
<tr>
<td>ID_MMFR5_EL1</td>
<td>ID_MMFR5_EL1, AArch32 Memory Model Feature Register 5 on page D17-6068</td>
</tr>
<tr>
<td>ID_PFR0_EL1</td>
<td>ID_PFR0_EL1, AArch32 Processor Feature Register 0 on page D17-6070</td>
</tr>
<tr>
<td>ID_PFR1_EL1</td>
<td>ID_PFR1_EL1, AArch32 Processor Feature Register 1 on page D17-6074</td>
</tr>
<tr>
<td>ID_PFR2_EL1</td>
<td>ID_PFR2_EL1, AArch32 Processor Feature Register 2 on page D17-6078</td>
</tr>
<tr>
<td>IFSR32_EL2</td>
<td>IFSR32_EL2, Instruction Fault Status Register (EL2) on page D17-6081</td>
</tr>
<tr>
<td>ISR_EL1</td>
<td>ISR_EL1, Interrupt Status Register on page D17-6086</td>
</tr>
<tr>
<td>LORC_EL1</td>
<td>LORC_EL1, LORegion Control (EL1) on page D17-6088</td>
</tr>
<tr>
<td>LOREA_EL1</td>
<td>LOREA_EL1, LORegion End Address (EL1) on page D17-6091</td>
</tr>
<tr>
<td>LORID_EL1</td>
<td>LORID_EL1, LORegionID (EL1) on page D17-6094</td>
</tr>
<tr>
<td>LORN_EL1</td>
<td>LORN_EL1, LORegion Number (EL1) on page D17-6096</td>
</tr>
<tr>
<td>LORSA_EL1</td>
<td>LORSA_EL1, LORegion Start Address (EL1) on page D17-6099</td>
</tr>
<tr>
<td>MAIR_EL1</td>
<td>MAIR_EL1, Memory Attribute Indirection Register (EL1) on page D17-6103</td>
</tr>
<tr>
<td>MAIR_EL2</td>
<td>MAIR_EL2, Memory Attribute Indirection Register (EL2) on page D17-6108</td>
</tr>
<tr>
<td>MAIR_EL3</td>
<td>MAIR_EL3, Memory Attribute Indirection Register (EL3) on page D17-6112</td>
</tr>
<tr>
<td>MDCCINT_EL1</td>
<td>MDCCINT_EL1, Monitor DCC Interrupt Enable Register on page D17-6446</td>
</tr>
<tr>
<td>MDCCSR_EL0</td>
<td>MDCCSR_EL0, Monitor DCC Status Register on page D17-6449</td>
</tr>
<tr>
<td>MDCR_EL2</td>
<td>MDCR_EL2, Monitor Debug Configuration Register (EL2) on page D17-6452</td>
</tr>
<tr>
<td>MDCR_EL3</td>
<td>MDCR_EL3, Monitor Debug Configuration Register (EL3) on page D17-6464</td>
</tr>
<tr>
<td>MDRAR_EL1</td>
<td>MDRAR_EL1, Monitor Debug ROM Address Register on page D17-6481</td>
</tr>
<tr>
<td>MDSR_EL1</td>
<td>MDSR_EL1, Monitor Debug System Control Register on page D17-6484</td>
</tr>
<tr>
<td>MFAR_EL3</td>
<td>MFAR_EL3, PA Fault Address Register on page D17-7012</td>
</tr>
<tr>
<td>MIDR_EL1</td>
<td>MIDR_EL1, Main ID Register on page D17-6115</td>
</tr>
<tr>
<td>MPAM0_EL1</td>
<td>MPAM0_EL1</td>
</tr>
<tr>
<td>MPAM1_EL1</td>
<td>MPAM1_EL1</td>
</tr>
<tr>
<td>MPAM2_EL2</td>
<td>MPAM2_EL2</td>
</tr>
<tr>
<td>MPAM3_EL3</td>
<td>MPAM3_EL3</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MPAMHCR_EL2</td>
<td>MPAMHCR_EL2</td>
</tr>
<tr>
<td>MPAMIDR_EL1</td>
<td>MPAMIDR_EL1</td>
</tr>
<tr>
<td>MPAMSM_EL1</td>
<td>MPAMSM_EL1</td>
</tr>
<tr>
<td>MPAMVPM0_EL2</td>
<td>MPAMVPM0_EL2</td>
</tr>
<tr>
<td>MPAMVPM1_EL2</td>
<td>MPAMVPM1_EL2</td>
</tr>
<tr>
<td>MPAMVPM2_EL2</td>
<td>MPAMVPM2_EL2</td>
</tr>
<tr>
<td>MPAMVPM3_EL2</td>
<td>MPAMVPM3_EL2</td>
</tr>
<tr>
<td>MPAMVPM4_EL2</td>
<td>MPAMVPM4_EL2</td>
</tr>
<tr>
<td>MPAMVPM5_EL2</td>
<td>MPAMVPM5_EL2</td>
</tr>
<tr>
<td>MPAMVPM6_EL2</td>
<td>MPAMVPM6_EL2</td>
</tr>
<tr>
<td>MPAMVPM7_EL2</td>
<td>MPAMVPM7_EL2</td>
</tr>
<tr>
<td>MPAMVPMV_EL2</td>
<td>MPAMVPMV_EL2</td>
</tr>
<tr>
<td>MIPDR_EL1</td>
<td>MIPDR_EL1, Multiprocessor Affinity Register on page D17-6118</td>
</tr>
<tr>
<td>MVFR0_EL1</td>
<td>MVFR0_EL1, AArch32 Media and VFP Feature Register 0 on page D17-6120</td>
</tr>
<tr>
<td>MVFR1_EL1</td>
<td>MVFR1_EL1, AArch32 Media and VFP Feature Register 1 on page D17-6124</td>
</tr>
<tr>
<td>MVFR2_EL1</td>
<td>MVFR2_EL1, AArch32 Media and VFP Feature Register 2 on page D17-6128</td>
</tr>
<tr>
<td>NZCV</td>
<td>NZCV, Condition Flags on page C5-706</td>
</tr>
<tr>
<td>OSDLR_EL1</td>
<td>OSDLR_EL1, OS Double Lock Register on page D17-6490</td>
</tr>
<tr>
<td>OSDTRRX_EL1</td>
<td>OSDTRRX_EL1, OS Lock Data Transfer Register, Receive on page D17-6493</td>
</tr>
<tr>
<td>OSDTRTX_EL1</td>
<td>OSDTRTX_EL1, OS Lock Data Transfer Register, Transmit on page D17-6496</td>
</tr>
<tr>
<td>OSECCR_EL1</td>
<td>OSECCR_EL1, OS Lock Exception Catch Control Register on page D17-6499</td>
</tr>
<tr>
<td>OSLAR_EL1</td>
<td>OSLAR_EL1, OS Lock Access Register on page D17-6502</td>
</tr>
<tr>
<td>OSLSR_EL1</td>
<td>OSLSR_EL1, OS Lock Status Register on page D17-6504</td>
</tr>
<tr>
<td>PAN</td>
<td>PAN, Privileged Access Never on page C5-708</td>
</tr>
<tr>
<td>PAR_EL1</td>
<td>PAR_EL1, Physical Address Register on page D17-6130</td>
</tr>
<tr>
<td>PMBIDR_EL1</td>
<td>PMBIDR_EL1, Profiling Buffer ID Register on page D17-6859</td>
</tr>
<tr>
<td>PMBLIMITR_EL1</td>
<td>PMBLIMITR_EL1, Profiling Buffer Limit Address Register on page D17-6862</td>
</tr>
<tr>
<td>PMBPTR_EL1</td>
<td>PMBPTR_EL1, Profiling Buffer Write Pointer Register on page D17-6865</td>
</tr>
<tr>
<td>PMBSR_EL1</td>
<td>PMBSR_EL1, Profiling Buffer Status/syndrome Register on page D17-6868</td>
</tr>
<tr>
<td>PMCCFILTR_EL0</td>
<td>PMCCFILTR_EL0, Performance Monitors Cycle Count Filter Register on page D17-6741</td>
</tr>
<tr>
<td>PMCCNTR_EL0</td>
<td>PMCCNTR_EL0, Performance Monitors Cycle Count Register on page D17-6747</td>
</tr>
<tr>
<td>PMCEID0_EL0</td>
<td>PMCEID0_EL0, Performance Monitors Common Event Identification register 0 on page D17-6750</td>
</tr>
</tbody>
</table>
### Table K17-9 Alphabetical index of AArch64 Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMCEID1_EL0</td>
<td>PMCEID1_EL0, Performance Monitors Common Event Identification register 1 on page D17-6753</td>
</tr>
<tr>
<td>PMCNTENCLR_EL0</td>
<td>PMCNTENCLR_EL0, Performance Monitors Count Enable Clear register on page D17-6756</td>
</tr>
<tr>
<td>PMCNTENSET_EL0</td>
<td>PMCNTENSET_EL0, Performance Monitors Count Enable Set register on page D17-6759</td>
</tr>
<tr>
<td>PMCR_EL0</td>
<td>PMCR_EL0, Performance Monitors Control Register on page D17-6762</td>
</tr>
<tr>
<td>PMEVCNTR&lt;n&gt;_EL0</td>
<td>PMEVCNTR&lt;n&gt;_EL0, Performance Monitors Event Count Registers, n = 0 - 30 on page D17-6769</td>
</tr>
<tr>
<td>PMEVTPYPER&lt;n&gt;_EL0</td>
<td>PMEVTPYPER&lt;n&gt;_EL0, Performance Monitors Event Type Registers, n = 0 - 30 on page D17-6773</td>
</tr>
<tr>
<td>PMINTENCLR_EL1</td>
<td>PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear register on page D17-6782</td>
</tr>
<tr>
<td>PMINTENSET_EL1</td>
<td>PMINTENSET_EL1, Performance Monitors Interrupt Enable Set register on page D17-6785</td>
</tr>
<tr>
<td>PMMIR_EL1</td>
<td>PMMIR_EL1, Performance Monitors Machine Identification Register on page D17-6788</td>
</tr>
<tr>
<td>PMOVSCLR_EL0</td>
<td>PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear Register on page D17-6791</td>
</tr>
<tr>
<td>PMOVSET_EL0</td>
<td>PMOVSET_EL0, Performance Monitors Overflow Flag Status Register on page D17-6795</td>
</tr>
<tr>
<td>PMSCR_EL1</td>
<td>PMSCR_EL1, Statistical Profiling Control Register (EL1) on page D17-6874</td>
</tr>
<tr>
<td>PMSCR_EL2</td>
<td>PMSCR_EL2, Statistical Profiling Control Register (EL2) on page D17-6879</td>
</tr>
<tr>
<td>PMSELR_EL0</td>
<td>PMSELR_EL0, Performance Monitors Event Counter Selection Register on page D17-6799</td>
</tr>
<tr>
<td>PMSEVFR_EL1</td>
<td>PMSEVFR_EL1, Sampling Event Filter Register on page D17-6884</td>
</tr>
<tr>
<td>PMSFCR_EL1</td>
<td>PMSFCR_EL1, Sampling Filter Control Register on page D17-6889</td>
</tr>
<tr>
<td>PMSICR_EL1</td>
<td>PMSICR_EL1, Sampling Interval Counter Register on page D17-6893</td>
</tr>
<tr>
<td>PMSIDR_EL1</td>
<td>PMSIDR_EL1, Sampling Profiling ID Register on page D17-6896</td>
</tr>
<tr>
<td>PMSIRR_EL1</td>
<td>PMSIRR_EL1, Sampling Interval Reload Register on page D17-6900</td>
</tr>
<tr>
<td>PMSLATFR_EL1</td>
<td>PMSLATFR_EL1, Sampling Latency Filter Register on page D17-6903</td>
</tr>
<tr>
<td>PMSNEVFR_EL1</td>
<td>PMSNEVFR_EL1, Sampling Inverted Event Filter Register on page D17-6906</td>
</tr>
<tr>
<td>PMSWINC_EL0</td>
<td>PMSWINC_EL0, Performance Monitors Software Increment register on page D17-6802</td>
</tr>
<tr>
<td>PMUSERENR_EL0</td>
<td>PMUSERENR_EL0, Performance Monitors User Enable Register on page D17-6804</td>
</tr>
<tr>
<td>PMXEVCNTR_EL0</td>
<td>PMXEVCNTR_EL0, Performance Monitors Selected Event Count Register on page D17-6808</td>
</tr>
<tr>
<td>PMXEVTYPER_EL0</td>
<td>PMXEVTYPER_EL0, Performance Monitors Selected Event Type Register on page D17-6812</td>
</tr>
<tr>
<td>REVIDR_EL1</td>
<td>REVIDR_EL1, Revision ID Register on page D17-6137</td>
</tr>
</tbody>
</table>
### Table K17-9 Alphabetical index of AArch64 Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGSR_EL1</td>
<td>RGSR_EL1, Random Allocation Tag Seed Register. on page D17-6138</td>
</tr>
<tr>
<td>RMR_EL1</td>
<td>RMR_EL1, Reset Management Register (EL1) on page D17-6140</td>
</tr>
<tr>
<td>RMR_EL2</td>
<td>RMR_EL2, Reset Management Register (EL2) on page D17-6142</td>
</tr>
<tr>
<td>RMR_EL3</td>
<td>RMR_EL3, Reset Management Register (EL3) on page D17-6144</td>
</tr>
<tr>
<td>RNDR</td>
<td>RNDR, Random Number on page D17-6146</td>
</tr>
<tr>
<td>RNDRRS</td>
<td>RNDRRS, Reseeded Random Number on page D17-6148</td>
</tr>
<tr>
<td>RVBAR_EL1</td>
<td>RVBAR_EL1, Reset Vector Base Address Register (if EL2 and EL3 not implemented)</td>
</tr>
<tr>
<td></td>
<td>on page D17-6150</td>
</tr>
<tr>
<td>RVBAR_EL2</td>
<td>RVBAR_EL2, Reset Vector Base Address Register (if EL3 not implemented)</td>
</tr>
<tr>
<td></td>
<td>on page D17-6151</td>
</tr>
<tr>
<td>RVBAR_EL3</td>
<td>RVBAR_EL3, Reset Vector Base Address Register (if EL3 implemented)</td>
</tr>
<tr>
<td></td>
<td>on page D17-6152</td>
</tr>
<tr>
<td>S3_&lt;op1&gt;_&lt;<em>Cn&gt;</em>&lt;<em>Cm&gt;</em>&lt;_op2&gt;</td>
<td>S3_&lt;op1&gt;_&lt;<em>Cn&gt;</em>&lt;<em>Cm&gt;</em>&lt;_op2&gt;, IMPLEMENTATION DEFINED registers on page D17-6153</td>
</tr>
<tr>
<td>SCR_EL3</td>
<td>SCR_EL3, Secure Configuration Register on page D17-6155</td>
</tr>
<tr>
<td>SCTLR_EL1</td>
<td>SCTLR_EL1, System Control Register (EL1) on page D17-6169</td>
</tr>
<tr>
<td>SCTLR_EL2</td>
<td>SCTLR_EL2, System Control Register (EL2) on page D17-6191</td>
</tr>
<tr>
<td>SCTLR_EL3</td>
<td>SCTLR_EL3, System Control Register (EL3) on page D17-6214</td>
</tr>
<tr>
<td>SCXTNUM_EL0</td>
<td>SCXTNUM_EL0, EL0 Read/Write Software Context Number on page D17-6224</td>
</tr>
<tr>
<td>SCXTNUM_EL1</td>
<td>SCXTNUM_EL1, EL1 Read/Write Software Context Number on page D17-6227</td>
</tr>
<tr>
<td>SCXTNUM_EL2</td>
<td>SCXTNUM_EL2, EL2 Read/Write Software Context Number on page D17-6231</td>
</tr>
<tr>
<td>SCXTNUM_EL3</td>
<td>SCXTNUM_EL3, EL3 Read/Write Software Context Number on page D17-6234</td>
</tr>
<tr>
<td>SDER32_EL2</td>
<td>SDER32_EL2, AArch32 Secure Debug Enable Register on page D17-6506</td>
</tr>
<tr>
<td>SDER32_EL3</td>
<td>SDER32_EL3, AArch32 Secure Debug Enable Register on page D17-6509</td>
</tr>
<tr>
<td>SMCR_EL1</td>
<td>SMCR_EL1, SME Control Register (EL1) on page D17-6236</td>
</tr>
<tr>
<td>SMCR_EL2</td>
<td>SMCR_EL2, SME Control Register (EL2) on page D17-6241</td>
</tr>
<tr>
<td>SMCR_EL3</td>
<td>SMCR_EL3, SME Control Register (EL3) on page D17-6246</td>
</tr>
<tr>
<td>SMIDR_EL1</td>
<td>SMIDR_EL1, Streaming Mode Identification Register on page D17-6248</td>
</tr>
<tr>
<td>SMPRI_EL1</td>
<td>SMPRI_EL1, Streaming Mode Priority Register on page D17-6254</td>
</tr>
<tr>
<td>SMPRIMAP_EL2</td>
<td>SMPRIMAP_EL2, Streaming Mode Priority Mapping Register on page D17-6250</td>
</tr>
<tr>
<td>SP_EL0</td>
<td>SP_EL0, Stack Pointer (EL0) on page C5-710</td>
</tr>
<tr>
<td>SP_EL1</td>
<td>SP_EL1, Stack Pointer (EL1) on page C5-712</td>
</tr>
<tr>
<td>SP_EL2</td>
<td>SP_EL2, Stack Pointer (EL2) on page C5-714</td>
</tr>
<tr>
<td>SP_EL3</td>
<td>SP_EL3, Stack Pointer (EL3) on page C5-716</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>SPSel</td>
<td>SPSel, Stack Pointer Select on page C5-717</td>
</tr>
<tr>
<td>SPSR_abt</td>
<td>SPSR_abt, Saved Program Status Register (Abort mode) on page C5-719</td>
</tr>
<tr>
<td>SPSR_EL1</td>
<td>SPSR_EL1, Saved Program Status Register (EL1) on page C5-724</td>
</tr>
<tr>
<td>SPSR_EL2</td>
<td>SPSR_EL2, Saved Program Status Register (EL2) on page C5-734</td>
</tr>
<tr>
<td>SPSR_EL3</td>
<td>SPSR_EL3, Saved Program Status Register (EL3) on page C5-743</td>
</tr>
<tr>
<td>SPSR_fiq</td>
<td>SPSR_fiq, Saved Program Status Register (FIQ mode) on page C5-751</td>
</tr>
<tr>
<td>SPSR_irq</td>
<td>SPSR_irq, Saved Program Status Register (IRQ mode) on page C5-756</td>
</tr>
<tr>
<td>SPSR_und</td>
<td>SPSR_und, Saved Program Status Register (Undefined mode) on page C5-761</td>
</tr>
<tr>
<td>SSBS</td>
<td>SSBS, Speculative Store Bypass Safe on page C5-766</td>
</tr>
<tr>
<td>SVCR</td>
<td>SVCR, Streaming Vector Control Register on page C5-768</td>
</tr>
<tr>
<td>TCO</td>
<td>TCO, Tag Check Override on page C5-772</td>
</tr>
<tr>
<td>TCR_EL1</td>
<td>TCR_EL1, Translation Control Register (EL1) on page D17-6257</td>
</tr>
<tr>
<td>TCR_EL2</td>
<td>TCR_EL2, Translation Control Register (EL2) on page D17-6272</td>
</tr>
<tr>
<td>TCR_EL3</td>
<td>TCR_EL3, Translation Control Register (EL3) on page D17-6292</td>
</tr>
<tr>
<td>TFSR_EL1</td>
<td>TFSR_EL1, Tag Fault Status Register (EL1) on page D17-6301</td>
</tr>
<tr>
<td>TFSR_EL2</td>
<td>TFSR_EL2, Tag Fault Status Register (EL2) on page D17-6306</td>
</tr>
<tr>
<td>TFSR_EL3</td>
<td>TFSR_EL3, Tag Fault Status Register (EL3) on page D17-6310</td>
</tr>
<tr>
<td>TFSRE0_EL1</td>
<td>TFSRE0_EL1, Tag Fault Status Register (EL0) on page D17-6299</td>
</tr>
<tr>
<td>TLBI ALLE1, TLBI ALLE1NXS</td>
<td>TLBI ALLE1, TLBI ALLE1NXS, TLB Invalidate All, EL1 on page C5-867</td>
</tr>
<tr>
<td>TLBI ALLE1IS, TLBI ALLE1ISNXS</td>
<td>TLBI ALLE1IS, TLBI ALLE1ISNXS, TLB Invalidate All, EL1, Inner Shareable on page C5-869</td>
</tr>
<tr>
<td>TLBI ALLE1OS, TLBI ALLE1OSNXS</td>
<td>TLBI ALLE1OS, TLBI ALLE1OSNXS, TLB Invalidate All, EL1, Outer Shareable on page C5-871</td>
</tr>
<tr>
<td>TLBI ALLE2, TLBI ALLE2NXS</td>
<td>TLBI ALLE2, TLBI ALLE2NXS, TLB Invalidate All, EL2 on page C5-873</td>
</tr>
<tr>
<td>TLBI ALLE2IS, TLBI ALLE2ISNXS</td>
<td>TLBI ALLE2IS, TLBI ALLE2ISNXS, TLB Invalidate All, EL2, Inner Shareable on page C5-875</td>
</tr>
<tr>
<td>TLBI ALLE2OS, TLBI ALLE2OSNXS</td>
<td>TLBI ALLE2OS, TLBI ALLE2OSNXS, TLB Invalidate All, EL2, Outer Shareable on page C5-877</td>
</tr>
<tr>
<td>TLBI ALLE3, TLBI ALLE3NXS</td>
<td>TLBI ALLE3, TLBI ALLE3NXS, TLB Invalidate All, EL3 on page C5-879</td>
</tr>
<tr>
<td>TLBI ALLE3IS, TLBI ALLE3ISNXS</td>
<td>TLBI ALLE3IS, TLBI ALLE3ISNXS, TLB Invalidate All, EL3, Inner Shareable on page C5-881</td>
</tr>
<tr>
<td>TLBI ALLE3OS, TLBI ALLE3OSNXS</td>
<td>TLBI ALLE3OS, TLBI ALLE3OSNXS, TLB Invalidate All, EL3, Outer Shareable on page C5-883</td>
</tr>
<tr>
<td>TLBI ASIDE1, TLBI ASIDE1NXS</td>
<td>TLBI ASIDE1, TLBI ASIDE1NXS, TLB Invalidate by ASID, EL1 on page C5-885</td>
</tr>
</tbody>
</table>
### Table K17-9 Alphabetical index of AArch64 Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB ASIDE1IS, TLBI ASIDE1ISNXS</td>
<td>TLBI ASIDE1IS, TLBI ASIDE1ISNXS, TLB Invalidate by ASID, EL1, Inner Shareable on page C5-888</td>
</tr>
<tr>
<td>TLBI ASIDE1OS, TLBI ASIDE1OSNXS</td>
<td>TLBI ASIDE1OS, TLBI ASIDE1OSNXS, TLB Invalidate by ASID, EL1, Outer Shareable on page C5-891</td>
</tr>
<tr>
<td>TLBI IPAS2E1, TLBI IPAS2E1INXS</td>
<td>TLBI IPAS2E1, TLBI IPAS2E1INXS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1 on page C5-894</td>
</tr>
<tr>
<td>TLBI IPAS2E1IS, TLBI IPAS2E1INXS</td>
<td>TLBI IPAS2E1IS, TLBI IPAS2E1INXS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable on page C5-898</td>
</tr>
<tr>
<td>TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS</td>
<td>TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable on page C5-902</td>
</tr>
<tr>
<td>TLBI IPAS2E1LE1, TLBI IPAS2E1LE1INXS</td>
<td>TLBI IPAS2E1LE1, TLBI IPAS2E1LE1INXS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1 on page C5-906</td>
</tr>
<tr>
<td>TLBI IPAS2E1LE1IS, TLBI IPAS2E1LE1INXS</td>
<td>TLBI IPAS2E1LE1IS, TLBI IPAS2E1LE1INXS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable on page C5-910</td>
</tr>
<tr>
<td>TLBI IPAS2E1LE1OS, TLBI IPAS2E1LE1OSNXS</td>
<td>TLBI IPAS2E1LE1OS, TLBI IPAS2E1LE1OSNXS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable on page C5-914</td>
</tr>
<tr>
<td>TLBI PAALL</td>
<td>TLBI PAALL, TLB Invalidate GPT Information by PA, All Entries, Local on page C5-918</td>
</tr>
<tr>
<td>TLBI PAALLOS</td>
<td>TLBI PAALLOS, TLB Invalidate GPT Information by PA, All Entries, Outer Shareable on page C5-919</td>
</tr>
<tr>
<td>TLBI RIPAS2E1, TLBI RIPAS2E1INXS</td>
<td>TLBI RIPAS2E1, TLBI RIPAS2E1INXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1 on page C5-920</td>
</tr>
<tr>
<td>TLBI RIPAS2E1IS, TLBI RIPAS2E1INXS</td>
<td>TLBI RIPAS2E1IS, TLBI RIPAS2E1INXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable on page C5-924</td>
</tr>
<tr>
<td>TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS</td>
<td>TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable on page C5-928</td>
</tr>
<tr>
<td>TLBI RIPAS2E1LE1, TLBI RIPAS2E1LE1INXS</td>
<td>TLBI RIPAS2E1LE1, TLBI RIPAS2E1LE1INXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1 on page C5-932</td>
</tr>
<tr>
<td>TLBI RIPAS2E1LE1IS, TLBI RIPAS2E1LE1INXS</td>
<td>TLBI RIPAS2E1LE1IS, TLBI RIPAS2E1LE1INXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable on page C5-936</td>
</tr>
<tr>
<td>TLBI RIPAS2E1LE1OS, TLBI RIPAS2E1LE1OSNXS</td>
<td>TLBI RIPAS2E1LE1OS, TLBI RIPAS2E1LE1OSNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable on page C5-940</td>
</tr>
<tr>
<td>TLBI RPALOS</td>
<td>TLBI RPALOS, TLB Range Invalidate GPT Information by PA, Last level, Outer Shareable on page C5-944</td>
</tr>
<tr>
<td>TLBI RPAOS</td>
<td>TLBI RPAOS, TLB Range Invalidate GPT Information by PA, Outer Shareable on page C5-946</td>
</tr>
<tr>
<td>TLBI RVAAE1, TLBI RVAAE1INXS</td>
<td>TLBI RVAAE1, TLBI RVAAE1INXS, TLB Range Invalidate by VA, All ASID, EL1 on page C5-948</td>
</tr>
<tr>
<td>TLBI RVAAE1IS, TLBI RVAAE1INXS</td>
<td>TLBI RVAAE1IS, TLBI RVAAE1INXS, TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable on page C5-952</td>
</tr>
<tr>
<td>TLBI RVAAE1OS, TLBI RVAAE1OSNXS</td>
<td>TLBI RVAAE1OS, TLBI RVAAE1OSNXS, TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable on page C5-956</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>----------</td>
<td>------------------</td>
</tr>
<tr>
<td>TLBI RVAALE1, TLBI RVAALE1NXS</td>
<td>TLBI RVAALE1, TLBI RVAALE1NXS, TLB Range Invalidate by VA, All ASID, Last level, EL1 on page C5-960</td>
</tr>
<tr>
<td>TLBI RVAALE1IS, TLBI RVAALE1ISNXS</td>
<td>TLBI RVAALE1IS, TLBI RVAALE1ISNXS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable on page C5-964</td>
</tr>
<tr>
<td>TLBI RVAALE1OS, TLBI RVAALE1OSNXS</td>
<td>TLBI RVAALE1OS, TLBI RVAALE1OSNXS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable on page C5-968</td>
</tr>
<tr>
<td>TLBI RVAE1, TLBI RVAE1NXS</td>
<td>TLBI RVAE1, TLBI RVAE1NXS, TLB Range Invalidate by VA, EL1 on page C5-972</td>
</tr>
<tr>
<td>TLBI RVAE1IS, TLBI RVAE1ISNXS</td>
<td>TLBI RVAE1IS, TLBI RVAE1ISNXS, TLB Range Invalidate by VA, EL1, Inner Shareable on page C5-976</td>
</tr>
<tr>
<td>TLBI RVAE1OS, TLBI RVAE1OSNXS</td>
<td>TLBI RVAE1OS, TLBI RVAE1OSNXS, TLB Range Invalidate by VA, EL1, Outer Shareable on page C5-980</td>
</tr>
<tr>
<td>TLBI RVAE2, TLBI RVAE2NXS</td>
<td>TLBI RVAE2, TLBI RVAE2NXS, TLB Range Invalidate by VA, EL2 on page C5-984</td>
</tr>
<tr>
<td>TLBI RVAE2IS, TLBI RVAE2ISNXS</td>
<td>TLBI RVAE2IS, TLBI RVAE2ISNXS, TLB Range Invalidate by VA, EL2, Inner Shareable on page C5-988</td>
</tr>
<tr>
<td>TLBI RVAE2OS, TLBI RVAE2OSNXS</td>
<td>TLBI RVAE2OS, TLBI RVAE2OSNXS, TLB Range Invalidate by VA, EL2, Outer Shareable on page C5-992</td>
</tr>
<tr>
<td>TLBI RVAE3, TLBI RVAE3NXS</td>
<td>TLBI RVAE3, TLBI RVAE3NXS, TLB Range Invalidate by VA, EL3 on page C5-996</td>
</tr>
<tr>
<td>TLBI RVAE3IS, TLBI RVAE3ISNXS</td>
<td>TLBI RVAE3IS, TLBI RVAE3ISNXS, TLB Range Invalidate by VA, EL3, Inner Shareable on page C5-999</td>
</tr>
<tr>
<td>TLBI RVAE3OS, TLBI RVAE3OSNXS</td>
<td>TLBI RVAE3OS, TLBI RVAE3OSNXS, TLB Range Invalidate by VA, EL3, Outer Shareable on page C5-1002</td>
</tr>
<tr>
<td>TLBI RVALE1, TLBI RVALE1NXS</td>
<td>TLBI RVALE1, TLBI RVALE1NXS, TLB Range Invalidate by VA, Last level, EL1 on page C5-1005</td>
</tr>
<tr>
<td>TLBI RVALE1IS, TLBI RVALE1ISNXS</td>
<td>TLBI RVALE1IS, TLBI RVALE1ISNXS, TLB Range Invalidate by VA, Last level, EL1, Inner Shareable on page C5-1009</td>
</tr>
<tr>
<td>TLBI RVALE1OS, TLBI RVALE1OSNXS</td>
<td>TLBI RVALE1OS, TLBI RVALE1OSNXS, TLB Range Invalidate by VA, Last level, EL1, Outer Shareable on page C5-1013</td>
</tr>
<tr>
<td>TLBI RVALE2, TLBI RVALE2NXS</td>
<td>TLBI RVALE2, TLBI RVALE2NXS, TLB Range Invalidate by VA, Last level, EL2 on page C5-1017</td>
</tr>
<tr>
<td>TLBI RVALE2IS, TLBI RVALE2ISNXS</td>
<td>TLBI RVALE2IS, TLBI RVALE2ISNXS, TLB Range Invalidate by VA, Last level, EL2, Inner Shareable on page C5-1021</td>
</tr>
<tr>
<td>TLBI RVALE2OS, TLBI RVALE2OSNXS</td>
<td>TLBI RVALE2OS, TLBI RVALE2OSNXS, TLB Range Invalidate by VA, Last level, EL2, Outer Shareable on page C5-1025</td>
</tr>
<tr>
<td>TLBI RVALE3, TLBI RVALE3NXS</td>
<td>TLBI RVALE3, TLBI RVALE3NXS, TLB Range Invalidate by VA, Last level, EL3 on page C5-1029</td>
</tr>
<tr>
<td>TLBI RVALE3IS, TLBI RVALE3ISNXS</td>
<td>TLBI RVALE3IS, TLBI RVALE3ISNXS, TLB Range Invalidate by VA, Last level, EL3, Inner Shareable on page C5-1032</td>
</tr>
<tr>
<td>TLBI RVALE3OS, TLBI RVALE3OSNXS</td>
<td>TLBI RVALE3OS, TLBI RVALE3OSNXS, TLB Range Invalidate by VA, Last level, EL3, Outer Shareable on page C5-1035</td>
</tr>
<tr>
<td>TLBI VAAE1, TLBI VAAE1NXS</td>
<td>TLBI VAAE1, TLBI VAAE1NXS, TLB Invalidate by VA, All ASID, EL1 on page C5-1038</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TLBI VAAE1IS, TLBI VAAE1ISNXS</td>
<td>TLBI VAAE1IS, TLBI VAAE1ISNXS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable on page C5-1042</td>
</tr>
<tr>
<td>TLBI VAAE1OS, TLBI VAAE1OSNXS</td>
<td>TLBI VAAE1OS, TLBI VAAE1OSNXS, TLB Invalidate by VA, All ASID, EL1, Outer Shareable on page C5-1046</td>
</tr>
<tr>
<td>TLBI VAALE1, TLBI VAALE1NXS</td>
<td>TLBI VAALE1, TLBI VAALE1NXS, TLB Invalidate by VA, All ASID, Last level, EL1 on page C5-1050</td>
</tr>
<tr>
<td>TLBI VAALE1IS, TLBI VAALE1ISNXS</td>
<td>TLBI VAALE1IS, TLBI VAALE1ISNXS, TLB Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable on page C5-1054</td>
</tr>
<tr>
<td>TLBI VAALE1OS, TLBI VAALE1OSNXS</td>
<td>TLBI VAALE1OS, TLBI VAALE1OSNXS, TLB Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable on page C5-1058</td>
</tr>
<tr>
<td>TLBI VAE1, TLBI VAE1NXS</td>
<td>TLBI VAE1, TLBI VAE1NXS, TLB Invalidate by VA, EL1 on page C5-1062</td>
</tr>
<tr>
<td>TLBI VAE1IS, TLBI VAE1ISNXS</td>
<td>TLBI VAE1IS, TLBI VAE1ISNXS, TLB Invalidate by VA, EL1, Inner Shareable on page C5-1066</td>
</tr>
<tr>
<td>TLBI VAE1OS, TLBI VAE1OSNXS</td>
<td>TLBI VAE1OS, TLBI VAE1OSNXS, TLB Invalidate by VA, EL1, Outer Shareable on page C5-1070</td>
</tr>
<tr>
<td>TLBI VAE2, TLBI VAE2NXS</td>
<td>TLBI VAE2, TLBI VAE2NXS, TLB Invalidate by VA, EL2 on page C5-1074</td>
</tr>
<tr>
<td>TLBI VAE2IS, TLBI VAE2ISNXS</td>
<td>TLBI VAE2IS, TLBI VAE2ISNXS, TLB Invalidate by VA, EL2, Inner Shareable on page C5-1078</td>
</tr>
<tr>
<td>TLBI VAE2OS, TLBI VAE2OSNXS</td>
<td>TLBI VAE2OS, TLBI VAE2OSNXS, TLB Invalidate by VA, EL2, Outer Shareable on page C5-1082</td>
</tr>
<tr>
<td>TLBI VAE3, TLBI VAE3NXS</td>
<td>TLBI VAE3, TLBI VAE3NXS, TLB Invalidate by VA, EL3 on page C5-1086</td>
</tr>
<tr>
<td>TLBI VAE3IS, TLBI VAE3ISNXS</td>
<td>TLBI VAE3IS, TLBI VAE3ISNXS, TLB Invalidate by VA, EL3, Inner Shareable on page C5-1089</td>
</tr>
<tr>
<td>TLBI VAE3OS, TLBI VAE3OSNXS</td>
<td>TLBI VAE3OS, TLBI VAE3OSNXS, TLB Invalidate by VA, EL3, Outer Shareable on page C5-1092</td>
</tr>
<tr>
<td>TLBI VALE1, TLBI VALE1NXS</td>
<td>TLBI VALE1, TLBI VALE1NXS, TLB Invalidate by VA, Last level, EL1 on page C5-1095</td>
</tr>
<tr>
<td>TLBI VALE1IS, TLBI VALE1ISNXS</td>
<td>TLBI VALE1IS, TLBI VALE1ISNXS, TLB Invalidate by VA, Last level, EL1, Inner Shareable on page C5-1099</td>
</tr>
<tr>
<td>TLBI VALE1OS, TLBI VALE1OSNXS</td>
<td>TLBI VALE1OS, TLBI VALE1OSNXS, TLB Invalidate by VA, Last level, EL1, Outer Shareable on page C5-1103</td>
</tr>
<tr>
<td>TLBI VALE2, TLBI VALE2NXS</td>
<td>TLBI VALE2, TLBI VALE2NXS, TLB Invalidate by VA, Last level, EL2 on page C5-1107</td>
</tr>
<tr>
<td>TLBI VALE2IS, TLBI VALE2ISNXS</td>
<td>TLBI VALE2IS, TLBI VALE2ISNXS, TLB Invalidate by VA, Last level, EL2, Inner Shareable on page C5-1111</td>
</tr>
<tr>
<td>TLBI VALE2OS, TLBI VALE2OSNXS</td>
<td>TLBI VALE2OS, TLBI VALE2OSNXS, TLB Invalidate by VA, Last level, EL2, Outer Shareable on page C5-1115</td>
</tr>
<tr>
<td>TLBI VALE3, TLBI VALE3NXS</td>
<td>TLBI VALE3, TLBI VALE3NXS, TLB Invalidate by VA, Last level, EL3 on page C5-1119</td>
</tr>
<tr>
<td>TLBI VALE3IS, TLBI VALE3ISNXS</td>
<td>TLBI VALE3IS, TLBI VALE3ISNXS, TLB Invalidate by VA, Last level, EL3, Inner Shareable on page C5-1122</td>
</tr>
</tbody>
</table>
### Table K17-9 Alphabetical index of AArch64 Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLBI VALE3OS, TLBI VALE3OSNXS</td>
<td>TLBI VALE3OS, TLBI VALE3OSNXS, TLB Invalidate by VA, Last level, EL3, Outer Shareable on page C5-1125</td>
</tr>
<tr>
<td>TLBI VMALLE1, TLBI VMALLE1NXS</td>
<td>TLBI VMALLE1, TLBI VMALLE1NXS, TLB Invalidate by VMID, All at stage 1, EL1 on page C5-1128</td>
</tr>
<tr>
<td>TLBI VMALLE1IS, TLBI VMALLE1ISNXS</td>
<td>TLBI VMALLE1IS, TLBI VMALLE1ISNXS, TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable on page C5-1131</td>
</tr>
<tr>
<td>TLBI VMALLE1OS, TLBI VMALLE1OSNXS</td>
<td>TLBI VMALLE1OS, TLBI VMALLE1OSNXS, TLB Invalidate by VMID, All at stage 1, EL1, Outer Shareable on page C5-1134</td>
</tr>
<tr>
<td>TLBI VMALLS12E1, TLBI VMALLS12E1NXS</td>
<td>TLBI VMALLS12E1, TLBI VMALLS12E1NXS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1 on page C5-1137</td>
</tr>
<tr>
<td>TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS</td>
<td>TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable on page C5-1140</td>
</tr>
<tr>
<td>TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS</td>
<td>TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Outer Shareable on page C5-1143</td>
</tr>
<tr>
<td>TPIDR2_EL0</td>
<td>TPIDR2_EL0, EL0 Read/Write Software Thread ID Register 2 on page D17-6312</td>
</tr>
<tr>
<td>TPIDR_EL0</td>
<td>TPIDR_EL0, EL0 Read/Write Software Thread ID Register on page D17-6315</td>
</tr>
<tr>
<td>TPIDR_EL1</td>
<td>TPIDR_EL1, EL1 Software Thread ID Register on page D17-6317</td>
</tr>
<tr>
<td>TPIDR_EL2</td>
<td>TPIDR_EL2, EL2 Software Thread ID Register on page D17-6319</td>
</tr>
<tr>
<td>TPIDR_EL3</td>
<td>TPIDR_EL3, EL3 Software Thread ID Register on page D17-6321</td>
</tr>
<tr>
<td>TPIDRRO_EL0</td>
<td>TPIDRRO_EL0, EL0 Read-Only Software Thread ID Register on page D17-6323</td>
</tr>
<tr>
<td>TRBASER_EL1</td>
<td>TRBASER_EL1, Trace Buffer Base Address Register on page D17-6520</td>
</tr>
<tr>
<td>TRBIDR_EL1</td>
<td>TRBIDR_EL1, Trace Buffer ID Register on page D17-6523</td>
</tr>
<tr>
<td>TRBLIMITR_EL1</td>
<td>TRBLIMITR_EL1, Trace Buffer Limit Address Register on page D17-6526</td>
</tr>
<tr>
<td>TRBMAR_EL1</td>
<td>TRBMAR_EL1, Trace Buffer Memory Attribute Register on page D17-6529</td>
</tr>
<tr>
<td>TRBPTR_EL1</td>
<td>TRBPTR_EL1, Trace Buffer Write Pointer Register on page D17-6534</td>
</tr>
<tr>
<td>TRBSR_EL1</td>
<td>TRBSR_EL1, Trace Buffer Status/syndrome Register on page D17-6537</td>
</tr>
<tr>
<td>TRBTRG_EL1</td>
<td>TRBTRG_EL1, Trace Buffer Trigger Counter Register on page D17-6543</td>
</tr>
<tr>
<td>TRCACATR&lt;n&gt;</td>
<td>TRCACATR&lt;n&gt;, Address Comparator Access Type Register &lt;n&gt;, n = 0 - 15 on page D17-6546</td>
</tr>
<tr>
<td>TRCACVR&lt;n&gt;</td>
<td>TRCACVR&lt;n&gt;, Address Comparator Value Register &lt;n&gt;, n = 0 - 15 on page D17-6553</td>
</tr>
<tr>
<td>TRCAUTHSTATUS</td>
<td>TRCAUTHSTATUS, Authentication Status Register on page D17-6556</td>
</tr>
<tr>
<td>TRCAUXCTLR</td>
<td>TRCAUXCTLR, Auxiliary Control Register on page D17-6560</td>
</tr>
<tr>
<td>TRCBBCTLR</td>
<td>TRCBBCTLR, Branch Broadcast Control Register on page D17-6563</td>
</tr>
<tr>
<td>TRCCTCLR</td>
<td>TRCCTCLR, Cycle Count Control Register on page D17-6566</td>
</tr>
<tr>
<td>TRCCIDCCTLR0</td>
<td>TRCCIDCCTLR0, Context Identifier Comparator Control Register 0 on page D17-6569</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TRCCIDCCTLR1</td>
<td>TRCCIDCCTLR1, Context Identifier Comparator Control Register 1 on page D17-6573</td>
</tr>
<tr>
<td>TRCCIDCVR&lt;n&gt;</td>
<td>TRCCIDCVR&lt;n&gt;, Context Identifier Comparator Value Registers &lt;n&gt;, n = 0 - 7 on page D17-6577</td>
</tr>
<tr>
<td>TRCCLAIMCLR</td>
<td>TRCCLAIMCLR, Claim Tag Clear Register on page D17-6580</td>
</tr>
<tr>
<td>TRCCLAIMSET</td>
<td>TRCCLAIMSET, Claim Tag Set Register on page D17-6583</td>
</tr>
<tr>
<td>TRCCNTCTLR&lt;n&gt;</td>
<td>TRCCNTCTLR&lt;n&gt;, Counter Control Register &lt;n&gt;, n = 0 - 3 on page D17-6586</td>
</tr>
<tr>
<td>TRCCNTRLDVR&lt;n&gt;</td>
<td>TRCCNTRLDVR&lt;n&gt;, Counter Reload Value Register &lt;n&gt;, n = 0 - 3 on page D17-6590</td>
</tr>
<tr>
<td>TRCCNTVR&lt;n&gt;</td>
<td>TRCCNTVR&lt;n&gt;, Counter Value Register &lt;n&gt;, n = 0 - 3 on page D17-6593</td>
</tr>
<tr>
<td>TRCCONFIGR</td>
<td>TRCCONFIGR, Trace Configuration Register on page D17-6596</td>
</tr>
<tr>
<td>TRCDEVARCH</td>
<td>TRCDEVARCH, Device Architecture Register on page D17-6601</td>
</tr>
<tr>
<td>TRCDEVID</td>
<td>TRCDEVID, Device Configuration Register on page D17-6603</td>
</tr>
<tr>
<td>TRCEVENTCTL0R</td>
<td>TRCEVENTCTL0R, Event Control 0 Register on page D17-6605</td>
</tr>
<tr>
<td>TRCEVENTCTL1R</td>
<td>TRCEVENTCTL1R, Event Control 1 Register on page D17-6610</td>
</tr>
<tr>
<td>TRCEXTINSELR&lt;n&gt;</td>
<td>TRCEXTINSELR&lt;n&gt;, External Input Select Register &lt;n&gt;, n = 0 - 3 on page D17-6613</td>
</tr>
<tr>
<td>TRCIDR0</td>
<td>TRCIDR0, ID Register 0 on page D17-6616</td>
</tr>
<tr>
<td>TRCIDR1</td>
<td>TRCIDR1, ID Register 1 on page D17-6621</td>
</tr>
<tr>
<td>TRCIDR10</td>
<td>TRCIDR10, ID Register 10 on page D17-6623</td>
</tr>
<tr>
<td>TRCIDR11</td>
<td>TRCIDR11, ID Register 11 on page D17-6625</td>
</tr>
<tr>
<td>TRCIDR12</td>
<td>TRCIDR12, ID Register 12 on page D17-6627</td>
</tr>
<tr>
<td>TRCIDR13</td>
<td>TRCIDR13, ID Register 13 on page D17-6629</td>
</tr>
<tr>
<td>TRCIDR2</td>
<td>TRCIDR2, ID Register 2 on page D17-6631</td>
</tr>
<tr>
<td>TRCIDR3</td>
<td>TRCIDR3, ID Register 3 on page D17-6634</td>
</tr>
<tr>
<td>TRCIDR4</td>
<td>TRCIDR4, ID Register 4 on page D17-6638</td>
</tr>
<tr>
<td>TRCIDR5</td>
<td>TRCIDR5, ID Register 5 on page D17-6642</td>
</tr>
<tr>
<td>TRCIDR6</td>
<td>TRCIDR6, ID Register 6 on page D17-6645</td>
</tr>
<tr>
<td>TRCIDR7</td>
<td>TRCIDR7, ID Register 7 on page D17-6647</td>
</tr>
<tr>
<td>TRCIDR8</td>
<td>TRCIDR8, ID Register 8 on page D17-6649</td>
</tr>
<tr>
<td>TRCIDR9</td>
<td>TRCIDR9, ID Register 9 on page D17-6651</td>
</tr>
<tr>
<td>TRCIMSPEC&lt;n&gt;</td>
<td>TRCIMSPEC&lt;n&gt;, IMP DEF Register &lt;n&gt;, n = 0 - 7 on page D17-6653</td>
</tr>
<tr>
<td>TRCIMSPEC0</td>
<td>TRCIMSPEC0, IMP DEF Register 0 on page D17-6653</td>
</tr>
<tr>
<td>TRCOSLSR</td>
<td>TRCOSLSR, Trace OS Lock Status Register on page D17-6660</td>
</tr>
<tr>
<td>TRCPRGCTRLR</td>
<td>TRCPRGCTRLR, Programming Control Register on page D17-6662</td>
</tr>
<tr>
<td>TRCQCTRLR</td>
<td>TRCQCTRLR, Q Element Control Register on page D17-6665</td>
</tr>
</tbody>
</table>
### Table K17-9 Alphabetical index of AArch64 Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRCRSCTRLR&lt;n&gt;</td>
<td>TRCRSCTRLR&lt;n&gt;, Resource Selection Control Register &lt;n&gt;, n = 2 - 31 on page D17-6668</td>
</tr>
<tr>
<td>TRCRSR</td>
<td>TRCRSR, Resources Status Register on page D17-6676</td>
</tr>
<tr>
<td>TRCSEQEV&lt;n&gt;</td>
<td>TRCSEQEV&lt;n&gt;, Sequencer State Transition Control Register &lt;n&gt;, n = 0 - 2 on page D17-6679</td>
</tr>
<tr>
<td>TRCSEQRSCTRL</td>
<td>TRCSEQRSCTRL, Sequencer Reset Control Register on page D17-6683</td>
</tr>
<tr>
<td>TRCSEQST</td>
<td>TRCSEQST, Sequencer State Register on page D17-6686</td>
</tr>
<tr>
<td>TRCSSCC&lt;n&gt;</td>
<td>TRCSSCC&lt;n&gt;, Single-shot Comparator Control Register &lt;n&gt;, n = 0 - 7 on page D17-6689</td>
</tr>
<tr>
<td>TRCSSCSR&lt;n&gt;</td>
<td>TRCSSCSR&lt;n&gt;, Single-shot Comparator Control Status Register &lt;n&gt;, n = 0 - 7 on page D17-6692</td>
</tr>
<tr>
<td>TRCSSPCIC&lt;n&gt;</td>
<td>TRCSSPCIC&lt;n&gt;, Single-shot Processing Element Comparator Input Control Register &lt;n&gt;, n = 0 - 7 on page D17-6696</td>
</tr>
<tr>
<td>TRCSTALLCTL</td>
<td>TRCSTALLCTL, Stall Control Register on page D17-6699</td>
</tr>
<tr>
<td>TRCSSTAT</td>
<td>TRCSSTAT, Trace Status Register on page D17-6702</td>
</tr>
<tr>
<td>TRCSYNCP</td>
<td>TRCSYNCP, Synchronization Period Register on page D17-6704</td>
</tr>
<tr>
<td>TRCTRACEIDR</td>
<td>TRCTRACEIDR, Trace ID Register on page D17-6707</td>
</tr>
<tr>
<td>TRCTSCCTRL</td>
<td>TRCTSCCTRL, Timestamp Control Register on page D17-6710</td>
</tr>
<tr>
<td>TRCVICTL</td>
<td>TRCVICTL, ViewInst Main Control Register on page D17-6713</td>
</tr>
<tr>
<td>TRCVIIECTL</td>
<td>TRCVIIECTL, ViewInst Include/Exclude Control Register on page D17-6720</td>
</tr>
<tr>
<td>TRCVIPCSSCTL</td>
<td>TRCVIPCSSCTL, ViewInst Start/Stop PE Comparator Control Register on page D17-6723</td>
</tr>
<tr>
<td>TRCVISSCTL</td>
<td>TRCVISSCTL, ViewInst Start/Stop Control Register on page D17-6726</td>
</tr>
<tr>
<td>TRCVMIDCCTL0</td>
<td>TRCVMIDCCTL0, Virtual Context Identifier Comparator Control Register 0 on page D17-6729</td>
</tr>
<tr>
<td>TRCVMIDCCTL1</td>
<td>TRCVMIDCCTL1, Virtual Context Identifier Comparator Control Register 1 on page D17-6733</td>
</tr>
<tr>
<td>TRCVMIDCV&lt;n&gt;</td>
<td>TRCVMIDCV&lt;n&gt;, Virtual Context Identifier Comparator Value Register &lt;n&gt;, n = 0 - 7 on page D17-6737</td>
</tr>
<tr>
<td>TRFCR_EL1</td>
<td>TRFCR_EL1, Trace Filter Control Register (EL1) on page D17-6511</td>
</tr>
<tr>
<td>TRFCR_EL2</td>
<td>TRFCR_EL2, Trace Filter Control Register (EL2) on page D17-6515</td>
</tr>
<tr>
<td>TTBR0_EL1</td>
<td>TTBR0_EL1, Translation Table Base Register 0 (EL1) on page D17-6325</td>
</tr>
<tr>
<td>TTBR0_EL2</td>
<td>TTBR0_EL2, Translation Table Base Register 0 (EL2) on page D17-6330</td>
</tr>
<tr>
<td>TTBR0_EL3</td>
<td>TTBR0_EL3, Translation Table Base Register 0 (EL3) on page D17-6334</td>
</tr>
<tr>
<td>TTBR1_EL1</td>
<td>TTBR1_EL1, Translation Table Base Register 1 (EL1) on page D17-6337</td>
</tr>
<tr>
<td>TTBR1_EL2</td>
<td>TTBR1_EL2, Translation Table Base Register 1 (EL2) on page D17-6342</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>UAO</td>
<td>UAO, User Access Override on page C5-774</td>
</tr>
<tr>
<td>VBAR_EL1</td>
<td>VBAR_EL1, Vector Base Address Register (EL1) on page D17-6346</td>
</tr>
<tr>
<td>VBAR_EL2</td>
<td>VBAR_EL2, Vector Base Address Register (EL2) on page D17-6349</td>
</tr>
<tr>
<td>VBAR_EL3</td>
<td>VBAR_EL3, Vector Base Address Register (EL3) on page D17-6352</td>
</tr>
<tr>
<td>VDISR_EL2</td>
<td>VDISR_EL2, Virtual Deferred Interrupt Status Register on page D17-6996</td>
</tr>
<tr>
<td>VMPIDR_EL2</td>
<td>VMPIDR_EL2, Virtualization Multiprocessor ID Register on page D17-6354</td>
</tr>
<tr>
<td>VNCR_EL2</td>
<td>VNCR_EL2, Virtual Nested Control Register on page D17-6357</td>
</tr>
<tr>
<td>VPIDR_EL2</td>
<td>VPIDR_EL2, Virtualization Processor ID Register on page D17-6359</td>
</tr>
<tr>
<td>VSESRR_EL2</td>
<td>VSESRR_EL2, Virtual SError Exception Syndrome Register on page D17-7001</td>
</tr>
<tr>
<td>VSCTCR_EL2</td>
<td>VSCTCR_EL2, Virtualization Secure Translation Control Register on page D17-6362</td>
</tr>
<tr>
<td>VSTTBR_EL2</td>
<td>VSTTBR_EL2, Virtualization Secure Translation Table Base Register on page D17-6367</td>
</tr>
<tr>
<td>VTCR_EL2</td>
<td>VTCR_EL2, Virtualization Translation Control Register on page D17-6370</td>
</tr>
<tr>
<td>VTTTBR_EL2</td>
<td>VTTTBR_EL2, Virtualization Translation Table Base Register on page D17-6378</td>
</tr>
<tr>
<td>ZCR_EL1</td>
<td>ZCR_EL1, SVE Control Register (EL1) on page D17-6382</td>
</tr>
<tr>
<td>ZCR_EL2</td>
<td>ZCR_EL2, SVE Control Register (EL2) on page D17-6387</td>
</tr>
<tr>
<td>ZCR_EL3</td>
<td>ZCR_EL3, SVE Control Register (EL3) on page D17-6391</td>
</tr>
</tbody>
</table>
K17.3 Functional index of AArch64 registers and System instructions

This section is an index of the AArch64 registers and System instructions, divided by functional group. Each of the following sections lists the registers for a functional group:

- Special-purpose registers.
- VMSA-specific registers on page K17-11869.
- ID registers on page K17-11870.
- Performance monitors registers on page K17-11871.
- Activity monitors registers on page K17-11872.
- Debug registers on page K17-11873.
- Trace registers on page K17-11874.
- Branch Record Buffer registers on page K17-11876.
- RAS registers on page K17-11876.
- Root Security state registers on page K17-11877.
- Generic timer registers on page K17-11877.
- Cache maintenance system instructions on page K17-11878.
- Address translation system instructions on page K17-11879.
- TLB maintenance system instructions on page K17-11880.
- Prediction restriction System instructions on page K17-11883.
- Base system registers on page K17-11883.

K17.3.1 Special-purpose registers

This section is an index to the registers in the Special-purpose registers functional group.

Table K17-10 Special-purpose registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELR_EL1</td>
<td>ELR_EL1</td>
</tr>
<tr>
<td>ELR_EL2</td>
<td>ELR_EL2</td>
</tr>
<tr>
<td>ELR_EL3</td>
<td>ELR_EL3</td>
</tr>
<tr>
<td>SP_EL0</td>
<td>SP_EL0</td>
</tr>
<tr>
<td>SP_EL1</td>
<td>SP_EL1</td>
</tr>
<tr>
<td>SP_EL2</td>
<td>SP_EL2</td>
</tr>
<tr>
<td>SP_EL3</td>
<td>SP_EL3</td>
</tr>
<tr>
<td>SPSR_abt</td>
<td>SPSR_abt</td>
</tr>
<tr>
<td>SPSR_EL1</td>
<td>SPSR_EL1</td>
</tr>
<tr>
<td>SPSR_EL2</td>
<td>SPSR_EL2</td>
</tr>
<tr>
<td>SPSR_EL3</td>
<td>SPSR_EL3</td>
</tr>
<tr>
<td>SPSR_fiq</td>
<td>SPSR_fiq</td>
</tr>
<tr>
<td>SPSR_irq</td>
<td>SPSR_irq</td>
</tr>
<tr>
<td>SPSR_und</td>
<td>SPSR_und</td>
</tr>
</tbody>
</table>
### K17.3.2 VMSA-specific registers

This section is an index to the registers in the Virtual memory control registers functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMAIR_EL1</td>
<td>AMAIR_EL1</td>
</tr>
<tr>
<td>AMAIR_EL2</td>
<td>AMAIR_EL2</td>
</tr>
<tr>
<td>AMAIR_EL3</td>
<td>AMAIR_EL3</td>
</tr>
<tr>
<td>CONTEXTIDR_EL1</td>
<td>CONTEXTIDR_EL1</td>
</tr>
<tr>
<td>CONTEXTIDR_EL2</td>
<td>CONTEXTIDR_EL2</td>
</tr>
<tr>
<td>DACR32_EL2</td>
<td>DACR32_EL2</td>
</tr>
<tr>
<td>GPCCR_EL3</td>
<td>GPCCR_EL3</td>
</tr>
<tr>
<td>GPTBR_EL3</td>
<td>GPTBR_EL3</td>
</tr>
<tr>
<td>LORC_EL1</td>
<td>LOREA_EL1</td>
</tr>
<tr>
<td>LOREA_EL1</td>
<td>LOREA_EL1</td>
</tr>
<tr>
<td>LORID_EL1</td>
<td>LORID_EL1</td>
</tr>
<tr>
<td>LORN_EL1</td>
<td>LORN_EL1</td>
</tr>
<tr>
<td>LORSA_EL1</td>
<td>LORSA_EL1</td>
</tr>
<tr>
<td>MAIR_EL1</td>
<td>MAIR_EL1</td>
</tr>
<tr>
<td>MAIR_EL2</td>
<td>MAIR_EL2</td>
</tr>
<tr>
<td>MAIR_EL3</td>
<td>MAIR_EL3</td>
</tr>
<tr>
<td>TCR_EL1</td>
<td>TCR_EL1</td>
</tr>
<tr>
<td>TCR_EL2</td>
<td>TCR_EL2</td>
</tr>
<tr>
<td>TCR_EL3</td>
<td>TCR_EL3</td>
</tr>
<tr>
<td>TTBR0_EL1</td>
<td>TTBR0_EL1</td>
</tr>
<tr>
<td>TTBR0_EL2</td>
<td>TTBR0_EL2</td>
</tr>
<tr>
<td>TTBR0_EL3</td>
<td>TTBR0_EL3</td>
</tr>
<tr>
<td>TTBR1_EL1</td>
<td>TTBR1_EL1</td>
</tr>
<tr>
<td>TTBR1_EL2</td>
<td>TTBR1_EL2</td>
</tr>
<tr>
<td>VTCR_EL2</td>
<td>VTCR_EL2</td>
</tr>
<tr>
<td>VTTBR_EL2</td>
<td>VTTBR_EL2</td>
</tr>
</tbody>
</table>
### K17.3.3 ID registers

This section is an index to the registers in the Identification registers functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCSIDR2_EL1</td>
<td>CCSIDR2_EL1</td>
</tr>
<tr>
<td>CCSIDR_EL1</td>
<td>CCSIDR_EL1</td>
</tr>
<tr>
<td>CLIDR_EL1</td>
<td>CLIDR_EL1</td>
</tr>
<tr>
<td>CSSELR_EL1</td>
<td>CSSELR_EL1</td>
</tr>
<tr>
<td>CTR_EL0</td>
<td>CTR_EL0</td>
</tr>
<tr>
<td>DCZID_EL0</td>
<td>DCZID_EL0</td>
</tr>
<tr>
<td>GMID_EL1</td>
<td>GMID_EL1</td>
</tr>
<tr>
<td>ID_AA64AFR0_EL1</td>
<td>ID_AA64AFR0_EL1</td>
</tr>
<tr>
<td>ID_AA64AFR1_EL1</td>
<td>ID_AA64AFR1_EL1</td>
</tr>
<tr>
<td>ID_AA64DFR0_EL1</td>
<td>ID_AA64DFR0_EL1</td>
</tr>
<tr>
<td>ID_AA64DFR1_EL1</td>
<td>ID_AA64DFR1_EL1</td>
</tr>
<tr>
<td>ID_AA64ISAR0_EL1</td>
<td>ID_AA64ISAR0_EL1</td>
</tr>
<tr>
<td>ID_AA64ISAR1_EL1</td>
<td>ID_AA64ISAR1_EL1</td>
</tr>
<tr>
<td>ID_AA64ISAR2_EL1</td>
<td>ID_AA64ISAR2_EL1</td>
</tr>
<tr>
<td>ID_AA64MMFR0_EL1</td>
<td>ID_AA64MMFR0_EL1</td>
</tr>
<tr>
<td>ID_AA64MMFR1_EL1</td>
<td>ID_AA64MMFR1_EL1</td>
</tr>
<tr>
<td>ID_AA64MMFR2_EL1</td>
<td>ID_AA64MMFR2_EL1</td>
</tr>
<tr>
<td>ID_AA64PFR0_EL1</td>
<td>ID_AA64PFR0_EL1</td>
</tr>
<tr>
<td>ID_AA64PFR1_EL1</td>
<td>ID_AA64PFR1_EL1</td>
</tr>
<tr>
<td>ID_AA64SMFR0_EL1</td>
<td>ID_AA64SMFR0_EL1</td>
</tr>
<tr>
<td>ID_AA64ZFR0_EL1</td>
<td>ID_AA64ZFR0_EL1</td>
</tr>
<tr>
<td>ID_AFR0_EL1</td>
<td>ID_AFR0_EL1</td>
</tr>
<tr>
<td>ID_DFR0_EL1</td>
<td>ID_DFR0_EL1</td>
</tr>
<tr>
<td>ID_DFR1_EL1</td>
<td>ID_DFR1_EL1</td>
</tr>
<tr>
<td>ID_ISAR0_EL1</td>
<td>ID_ISAR0_EL1</td>
</tr>
<tr>
<td>ID_ISAR1_EL1</td>
<td>ID_ISAR1_EL1</td>
</tr>
<tr>
<td>ID_ISAR2_EL1</td>
<td>ID_ISAR2_EL1</td>
</tr>
<tr>
<td>ID_ISAR3_EL1</td>
<td>ID_ISAR3_EL1</td>
</tr>
<tr>
<td>ID_ISAR4_EL1</td>
<td>ID_ISAR4_EL1</td>
</tr>
<tr>
<td>ID_ISAR5_EL1</td>
<td>ID_ISAR5_EL1</td>
</tr>
</tbody>
</table>
This section is an index to the registers in the Performance Monitors registers functional group.

**Table K17-13 Performance monitors registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMCCFILTR_EL0</td>
<td>PMCCFILTR_EL0</td>
</tr>
<tr>
<td>PMCCNTR_EL0</td>
<td>PMCCNTR_EL0</td>
</tr>
<tr>
<td>PMCEID0_EL0</td>
<td>PMCEID0_EL0</td>
</tr>
<tr>
<td>PMCEID1_EL0</td>
<td>PMCEID1_EL0</td>
</tr>
<tr>
<td>PMCNTENCLR_EL0</td>
<td>PMCNTENCLR_EL0</td>
</tr>
<tr>
<td>PMCNTENSET_EL0</td>
<td>PMCNTENSET_EL0</td>
</tr>
<tr>
<td>PMCR_EL0</td>
<td>PMCR_EL0</td>
</tr>
<tr>
<td>PMEVCNTR&lt;(n)&gt;_EL0</td>
<td>PMEVCNTR&lt;(n)&gt;_EL0</td>
</tr>
<tr>
<td>PMEVTYPER&lt;(n)&gt;_EL0</td>
<td>PMEVTYPER&lt;(n)&gt;_EL0</td>
</tr>
<tr>
<td>PMINTENCLR_EL1</td>
<td>PMINTENCLR_EL1</td>
</tr>
<tr>
<td>PMINTENSET_EL1</td>
<td>PMINTENSET_EL1</td>
</tr>
<tr>
<td>PMMIR_EL1</td>
<td>PMMIR_EL1</td>
</tr>
</tbody>
</table>
### K17.3.5 Activity monitors registers

This section is an index to the registers in the Activity Monitors registers functional group.

#### Table K17-14 Activity monitors registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMCFGR_EL0</td>
<td>AMCFGR_EL0</td>
</tr>
<tr>
<td>AMCG1HDR_EL0</td>
<td>AMCG1HDR_EL0</td>
</tr>
<tr>
<td>AMCGCR_EL0</td>
<td>AMCGCR_EL0</td>
</tr>
<tr>
<td>AMCNTENCLR0_EL0</td>
<td>AMCNTENCLR0_EL0</td>
</tr>
<tr>
<td>AMCNTENCLR1_EL0</td>
<td>AMCNTENCLR1_EL0</td>
</tr>
<tr>
<td>AMCNTENSET0_EL0</td>
<td>AMCNTENSET0_EL0</td>
</tr>
<tr>
<td>AMCNTENSET1_EL0</td>
<td>AMCNTENSET1_EL0</td>
</tr>
<tr>
<td>AMCR_EL0</td>
<td>AMCR_EL0</td>
</tr>
<tr>
<td>AMEVCTR0&lt;n&gt;_EL0</td>
<td>AMEVCTR0&lt;n&gt;_EL0</td>
</tr>
<tr>
<td>AMEVCTR1&lt;n&gt;_EL0</td>
<td>AMEVCTR1&lt;n&gt;_EL0</td>
</tr>
<tr>
<td>AMEVCTVOFF0&lt;n&gt;_EL2</td>
<td>AMEVCTVOFF0&lt;n&gt;_EL2</td>
</tr>
<tr>
<td>AMEVCTVOFF1&lt;n&gt;_EL2</td>
<td>AMEVCTVOFF1&lt;n&gt;_EL2</td>
</tr>
<tr>
<td>AMEVTPER&lt;n&gt;_EL0</td>
<td>AMEVTPER&lt;n&gt;_EL0</td>
</tr>
<tr>
<td>AMUSERENR_EL0</td>
<td>AMUSERENR_EL0</td>
</tr>
</tbody>
</table>
## K17.3.6 Debug registers

This section is an index to the registers in the Debug registers functional group.

### Table K17-15 Debug registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBGAUTHSTATUS_EL1</td>
<td>DBGAUTHSTATUS_EL1</td>
</tr>
<tr>
<td>DBGBCR&lt;n&gt;_EL1</td>
<td>DBGBCR&lt;n&gt;_EL1</td>
</tr>
<tr>
<td>DBGVR&lt;n&gt;_EL1</td>
<td>DBGVR&lt;n&gt;_EL1</td>
</tr>
<tr>
<td>DBGCLAIMCLR_EL1</td>
<td>DBGCLAIMCLR_EL1</td>
</tr>
<tr>
<td>DBGCLAIMSET_EL1</td>
<td>DBGCLAIMSET_EL1</td>
</tr>
<tr>
<td>DBGDTR_EL0</td>
<td>DBGDTR_EL0</td>
</tr>
<tr>
<td>DBGDTRRX_EL0</td>
<td>DBGDTRRX_EL0</td>
</tr>
<tr>
<td>DBGDTRTX_EL0</td>
<td>DBGDTRTX_EL0</td>
</tr>
<tr>
<td>DBGPRCR_EL1</td>
<td>DBGPRCR_EL1</td>
</tr>
<tr>
<td>DBGVCR32_EL2</td>
<td>DBGVCR32_EL2</td>
</tr>
<tr>
<td>DBGWCR&lt;n&gt;_EL1</td>
<td>DBGWCR&lt;n&gt;_EL1</td>
</tr>
<tr>
<td>DBGWVR&lt;n&gt;_EL1</td>
<td>DBGWVR&lt;n&gt;_EL1</td>
</tr>
<tr>
<td>DLR_EL0</td>
<td>DLR_EL0</td>
</tr>
<tr>
<td>DSPSR_EL0</td>
<td>DSPSR_EL0</td>
</tr>
<tr>
<td>MDCCINT_EL1</td>
<td>MDCCINT_EL1</td>
</tr>
<tr>
<td>MDCCSR_EL0</td>
<td>MDCCSR_EL0</td>
</tr>
<tr>
<td>MDRAR_EL1</td>
<td>MDRAR_EL1</td>
</tr>
<tr>
<td>MDSCR_EL1</td>
<td>MDSCR_EL1</td>
</tr>
<tr>
<td>OSDLR_EL1</td>
<td>OSDLR_EL1</td>
</tr>
<tr>
<td>OSDTRRX_EL1</td>
<td>OSDTRRX_EL1</td>
</tr>
<tr>
<td>OSDTRTX_EL1</td>
<td>OSDTRTX_EL1</td>
</tr>
<tr>
<td>OSECCR_EL1</td>
<td>OSECCR_EL1</td>
</tr>
<tr>
<td>OSLAR_EL1</td>
<td>OSLAR_EL1</td>
</tr>
<tr>
<td>OLSR_EL1</td>
<td>OLSR_EL1</td>
</tr>
<tr>
<td>TRFCR_EL1</td>
<td>TRFCR_EL1</td>
</tr>
<tr>
<td>TRFCR_EL2</td>
<td>TRFCR_EL2</td>
</tr>
</tbody>
</table>
### K17.3.7 Trace registers

This section is an index to the registers in the Trace unit registers functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRBBASER_EL1</td>
<td>TRBBASER_EL1</td>
</tr>
<tr>
<td>TRBIDR_EL1</td>
<td>TRBIDR_EL1</td>
</tr>
<tr>
<td>TRBLIMITR_EL1</td>
<td>TRBLIMITR_EL1</td>
</tr>
<tr>
<td>TRBMAR_EL1</td>
<td>TRBMAR_EL1</td>
</tr>
<tr>
<td>TRBPTR_EL1</td>
<td>TRBPTR_EL1</td>
</tr>
<tr>
<td>TRBSR_EL1</td>
<td>TRBSR_EL1</td>
</tr>
<tr>
<td>TRBTRG_EL1</td>
<td>TRBTRG_EL1</td>
</tr>
<tr>
<td>TRCACATR&lt;n&gt;</td>
<td>TRCACATR&lt;n&gt;</td>
</tr>
<tr>
<td>TRCACVR&lt;n&gt;</td>
<td>TRCACVR&lt;n&gt;</td>
</tr>
<tr>
<td>TRCAUXCTLR</td>
<td>TRCAUXCTLR</td>
</tr>
<tr>
<td>TRCBBCTRLR</td>
<td>TRCBBCTRLR</td>
</tr>
<tr>
<td>TRCCCCTLR</td>
<td>TRCCCCTLR</td>
</tr>
<tr>
<td>TRCCIDCCTRLR0</td>
<td>TRCCIDCCTRLR0</td>
</tr>
<tr>
<td>TRCCIDCCTRLR1</td>
<td>TRCCIDCCTRLR1</td>
</tr>
<tr>
<td>TRCCIDCVR&lt;n&gt;</td>
<td>TRCCIDCVR&lt;n&gt;</td>
</tr>
<tr>
<td>TRCCCLAIMCLR</td>
<td>TRCCCLAIMCLR</td>
</tr>
<tr>
<td>TRCCCLAIMSET</td>
<td>TRCCCLAIMSET</td>
</tr>
<tr>
<td>TRCCNTCTRLR&lt;n&gt;</td>
<td>TRCCNTCTRLR&lt;n&gt;</td>
</tr>
<tr>
<td>TRCCNTRLDVR&lt;n&gt;</td>
<td>TRCCNTRLDVR&lt;n&gt;</td>
</tr>
<tr>
<td>TRCCNTVR&lt;n&gt;</td>
<td>TRCCNTVR&lt;n&gt;</td>
</tr>
<tr>
<td>TRCCONFIGR</td>
<td>TRCCONFIGR</td>
</tr>
<tr>
<td>TRACEVENTCTL0R</td>
<td>TRACEVENTCTL0R</td>
</tr>
<tr>
<td>TRACEVENTCTL1R</td>
<td>TRACEVENTCTL1R</td>
</tr>
<tr>
<td>TRCEXTINSELR&lt;n&gt;</td>
<td>TRCEXTINSELR&lt;n&gt;</td>
</tr>
<tr>
<td>TRCIDR0</td>
<td>TRCIDR0</td>
</tr>
<tr>
<td>TRCIDR1</td>
<td>TRCIDR1</td>
</tr>
<tr>
<td>TRCIDR10</td>
<td>TRCIDR10</td>
</tr>
<tr>
<td>TRCIDR11</td>
<td>TRCIDR11</td>
</tr>
<tr>
<td>TRCIDR12</td>
<td>TRCIDR12</td>
</tr>
<tr>
<td>TRCIDR13</td>
<td>TRCIDR13</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>TRCIDR2</td>
<td>TRCIDR2</td>
</tr>
<tr>
<td>TRCIDR3</td>
<td>TRCIDR3</td>
</tr>
<tr>
<td>TRCIDR4</td>
<td>TRCIDR4</td>
</tr>
<tr>
<td>TRCIDR5</td>
<td>TRCIDR5</td>
</tr>
<tr>
<td>TRCIDR6</td>
<td>TRCIDR6</td>
</tr>
<tr>
<td>TRCIDR7</td>
<td>TRCIDR7</td>
</tr>
<tr>
<td>TRCIDR8</td>
<td>TRCIDR8</td>
</tr>
<tr>
<td>TRCIDR9</td>
<td>TRCIDR9</td>
</tr>
<tr>
<td>TRCIMSPEC0</td>
<td>TRCIMSPEC0</td>
</tr>
<tr>
<td>TRCIMSPEC&lt;n&gt;</td>
<td>TRCIMSPEC&lt;n&gt;</td>
</tr>
<tr>
<td>TRCPRGCTRLR</td>
<td>TRCPRGCTRLR</td>
</tr>
<tr>
<td>TRCQCTRLR</td>
<td>TRCQCTRLR</td>
</tr>
<tr>
<td>TRCRSCTRLR&lt;n&gt;</td>
<td>TRCRSCTRLR&lt;n&gt;</td>
</tr>
<tr>
<td>TRCRSR</td>
<td>TRCRSR</td>
</tr>
<tr>
<td>TRCSEQEVR&lt;n&gt;</td>
<td>TRCSEQEVR&lt;n&gt;</td>
</tr>
<tr>
<td>TRCSEQRSEVTR</td>
<td>TRCSEQRSEVTR</td>
</tr>
<tr>
<td>TRCSEQSTR</td>
<td>TRCSEQSTR</td>
</tr>
<tr>
<td>TRCSSCCR&lt;n&gt;</td>
<td>TRCSSCCR&lt;n&gt;</td>
</tr>
<tr>
<td>TRCSSCSR&lt;n&gt;</td>
<td>TRCSSCSR&lt;n&gt;</td>
</tr>
<tr>
<td>TRCSSPCICR&lt;n&gt;</td>
<td>TRCSSPCICR&lt;n&gt;</td>
</tr>
<tr>
<td>TRCSTALLCTRLR</td>
<td>TRCSTALLCTRLR</td>
</tr>
<tr>
<td>TRCSTATR</td>
<td>TRCSTATR</td>
</tr>
<tr>
<td>TRCSYNCPR</td>
<td>TRCSYNCPR</td>
</tr>
<tr>
<td>TRCTRACEIDR</td>
<td>TRCTRACEIDR</td>
</tr>
<tr>
<td>TRCTSCCTRLR</td>
<td>TRCTSCCTRLR</td>
</tr>
<tr>
<td>TRCVICTLR</td>
<td>TRCVICTLR</td>
</tr>
<tr>
<td>TRCVIIECTRLR</td>
<td>TRCVIIECTRLR</td>
</tr>
<tr>
<td>TRCVIPCSCTRLR</td>
<td>TRCVIPCSCTRLR</td>
</tr>
<tr>
<td>TRCVISSCTRLR</td>
<td>TRCVISSCTRLR</td>
</tr>
<tr>
<td>TRCVMIDCCTRLR0</td>
<td>TRCVMIDCCTRLR0</td>
</tr>
<tr>
<td>TRCVMIDCCTRLR1</td>
<td>TRCVMIDCCTRLR1</td>
</tr>
<tr>
<td>TRCVMIDCVR&lt;n&gt;</td>
<td>TRCVMIDCVR&lt;n&gt;</td>
</tr>
</tbody>
</table>
### K17.3.8 Branch Record Buffer registers

This section is an index to the registers in the Branch Record Buffer Extension registers functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRBCR_EL1</td>
<td>BRBCR_EL1</td>
</tr>
<tr>
<td>BRBCR_EL2</td>
<td>BRBCR_EL2</td>
</tr>
<tr>
<td>BRBFCR_EL1</td>
<td>BRBFCR_EL1</td>
</tr>
<tr>
<td>BRBIDR0_EL1</td>
<td>BRBIDR0_EL1</td>
</tr>
<tr>
<td>BRBINFINJ_EL1</td>
<td>BRBINFINJ_EL1</td>
</tr>
<tr>
<td>BRBINF&lt;n&gt;_EL1</td>
<td>BRBINF&lt;n&gt;_EL1</td>
</tr>
<tr>
<td>BRBSCINJ_EL1</td>
<td>BRBSCINJ_EL1</td>
</tr>
<tr>
<td>BRBSRC&lt;n&gt;_EL1</td>
<td>BRBSRC&lt;n&gt;_EL1</td>
</tr>
<tr>
<td>BRBTGTINJ_EL1</td>
<td>BRBTGTINJ_EL1</td>
</tr>
<tr>
<td>BRBTGT&lt;n&gt;_EL1</td>
<td>BRBTGT&lt;n&gt;_EL1</td>
</tr>
<tr>
<td>BRBTS_EL1</td>
<td>BRBTS_EL1</td>
</tr>
</tbody>
</table>

### K17.3.9 RAS registers

This section is an index to the registers in the RAS registers functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISR_EL1</td>
<td>DISR_EL1</td>
</tr>
<tr>
<td>ERRIDR_EL1</td>
<td>ERRIDR_EL1</td>
</tr>
<tr>
<td>ERRSELR_EL1</td>
<td>ERRSELR_EL1</td>
</tr>
<tr>
<td>ERXADDR_EL1</td>
<td>ERXADDR_EL1</td>
</tr>
<tr>
<td>ERXCTR_EL1</td>
<td>ERXCTR_EL1</td>
</tr>
<tr>
<td>ERXFR_EL1</td>
<td>ERXFR_EL1</td>
</tr>
<tr>
<td>ERXMISC0_EL1</td>
<td>ERXMISC0_EL1</td>
</tr>
<tr>
<td>ERXMISC1_EL1</td>
<td>ERXMISC1_EL1</td>
</tr>
<tr>
<td>ERXMISC2_EL1</td>
<td>ERXMISC2_EL1</td>
</tr>
<tr>
<td>ERXMISC3_EL1</td>
<td>ERXMISC3_EL1</td>
</tr>
<tr>
<td>ERXPFGCDN_EL1</td>
<td>ERXPFGCDN_EL1</td>
</tr>
<tr>
<td>ERXPFGCTL_EL1</td>
<td>ERXPFGCTL_EL1</td>
</tr>
<tr>
<td>ERXPFGF_EL1</td>
<td>ERXPFGF_EL1</td>
</tr>
</tbody>
</table>
K17.3.10 Root Security state registers

This section is an index to the registers in the Root Security state registers functional group.

Table K17-19 Root Security state registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERXSTATUS_EL1</td>
<td>ERXSTATUS_EL1</td>
</tr>
<tr>
<td>VDISR_EL2</td>
<td>VDISR_EL2</td>
</tr>
<tr>
<td>VSESril_EL2</td>
<td>VSESril_EL2</td>
</tr>
</tbody>
</table>

K17.3.11 Generic timer registers

This section is an index to the registers in the Generic Timer registers functional group.

Table K17-20 Generic timer registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNTFRQ_EL0</td>
<td>CNTFRQ_EL0</td>
</tr>
<tr>
<td>CNTHV_CTL_EL2</td>
<td>CNTHV_CTL_EL2</td>
</tr>
<tr>
<td>CNTHV_CVAL_EL2</td>
<td>CNTHV_CVAL_EL2</td>
</tr>
<tr>
<td>CNTHV_TVAL_EL2</td>
<td>CNTHV_TVAL_EL2</td>
</tr>
<tr>
<td>CNTHVS_CTL_EL2</td>
<td>CNTHVS_CTL_EL2</td>
</tr>
<tr>
<td>CNTHVS_CVAL_EL2</td>
<td>CNTHVS_CVAL_EL2</td>
</tr>
<tr>
<td>CNTHVS_TVAL_EL2</td>
<td>CNTHVS_TVAL_EL2</td>
</tr>
<tr>
<td>CNTKCTL_EL1</td>
<td>CNTKCTL_EL1</td>
</tr>
<tr>
<td>CNTP_CTL_EL0</td>
<td>CNTP_CTL_EL0</td>
</tr>
<tr>
<td>CNTP_CVAL_EL0</td>
<td>CNTP_CVAL_EL0</td>
</tr>
<tr>
<td>CNTP_TVAL_EL0</td>
<td>CNTP_TVAL_EL0</td>
</tr>
<tr>
<td>CNTPCT_EL0</td>
<td>CNTPCT_EL0</td>
</tr>
<tr>
<td>CNTPCTSS_EL0</td>
<td>CNTPCTSS_EL0</td>
</tr>
<tr>
<td>CNTPOFF_EL2</td>
<td>CNTPOFF_EL2</td>
</tr>
<tr>
<td>CNTPS_CTL_EL1</td>
<td>CNTPS_CTL_EL1</td>
</tr>
<tr>
<td>CNTPS_CVAL_EL1</td>
<td>CNTPS_CVAL_EL1</td>
</tr>
</tbody>
</table>
K17.3.12 Cache maintenance system instructions

This section is an index to the registers in the Cache maintenance instructions functional group.

Table K17-21 Cache maintenance system instructions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC CGDSW</td>
<td>DC CGDSW</td>
</tr>
<tr>
<td>DC CGDVAC</td>
<td>DC CGDVAC</td>
</tr>
<tr>
<td>DC CGDVADP</td>
<td>DC CGDVADP</td>
</tr>
<tr>
<td>DC CGDVAP</td>
<td>DC CGDVAP</td>
</tr>
<tr>
<td>DC CGSW</td>
<td>DC CGSW</td>
</tr>
<tr>
<td>DC CGVAC</td>
<td>DC CGVAC</td>
</tr>
<tr>
<td>DC CGVADP</td>
<td>DC CGVADP</td>
</tr>
<tr>
<td>DC CGVAP</td>
<td>DC CGVAP</td>
</tr>
<tr>
<td>DC CIGDPAPA</td>
<td>DC CIGDPAPA</td>
</tr>
<tr>
<td>DC CIGDSW</td>
<td>DC CIGDSW</td>
</tr>
<tr>
<td>DC CIGDVAC</td>
<td>DC CIGDVAC</td>
</tr>
<tr>
<td>DC CIGSW</td>
<td>DC CIGSW</td>
</tr>
<tr>
<td>DC CIGVAC</td>
<td>DC CIGVAC</td>
</tr>
<tr>
<td>DC CIPAPA</td>
<td>DC CIPAPA</td>
</tr>
<tr>
<td>DC CISW</td>
<td>DC CISW</td>
</tr>
<tr>
<td>DC CIVAC</td>
<td>DC CIVAC</td>
</tr>
<tr>
<td>DC CSW</td>
<td>DC CSW</td>
</tr>
<tr>
<td>DC CVAC</td>
<td>DC CVAC</td>
</tr>
<tr>
<td>DC CVADP</td>
<td>DC CVADP</td>
</tr>
<tr>
<td>DC CVAP</td>
<td>DC CVAP</td>
</tr>
<tr>
<td>DC CVAU</td>
<td>DC CVAU</td>
</tr>
</tbody>
</table>
K17.3.13 Address translation system instructions

This section is an index to the registers in the Address translation instructions functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT S12E0R</td>
<td>AT S12E0R</td>
</tr>
<tr>
<td>AT S12E0W</td>
<td>AT S12E0W</td>
</tr>
<tr>
<td>AT S12E1R</td>
<td>AT S12E1R</td>
</tr>
<tr>
<td>AT S12E1W</td>
<td>AT S12E1W</td>
</tr>
<tr>
<td>AT S1E0R</td>
<td>AT S1E0R</td>
</tr>
<tr>
<td>AT S1E0W</td>
<td>AT S1E0W</td>
</tr>
<tr>
<td>AT S1E1R</td>
<td>AT S1E1R</td>
</tr>
<tr>
<td>AT S1E1RP</td>
<td>AT S1E1RP</td>
</tr>
<tr>
<td>AT S1E1W</td>
<td>AT S1E1W</td>
</tr>
<tr>
<td>AT S1E1WP</td>
<td>AT S1E1WP</td>
</tr>
<tr>
<td>AT S1E2R</td>
<td>AT S1E2R</td>
</tr>
<tr>
<td>AT S1E2W</td>
<td>AT S1E2W</td>
</tr>
<tr>
<td>AT S1E3R</td>
<td>AT S1E3R</td>
</tr>
<tr>
<td>AT S1E3W</td>
<td>AT S1E3W</td>
</tr>
</tbody>
</table>
### K17.3.14 TLB maintenance system instructions

This section is an index to the registers in the TLB maintenance instructions functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLBI ALLE1, TLBI ALLE1NXS</td>
<td>TLBI ALLE1, TLBI ALLE1NXS</td>
</tr>
<tr>
<td>TLBI ALLE1IS, TLBI ALLE1ISNXS</td>
<td>TLBI ALLE1IS, TLBI ALLE1ISNXS</td>
</tr>
<tr>
<td>TLBI ALLE1OS, TLBI ALLE1OSNXS</td>
<td>TLBI ALLE1OS, TLBI ALLE1OSNXS</td>
</tr>
<tr>
<td>TLBI ALLE2, TLBI ALLE2NXS</td>
<td>TLBI ALLE2, TLBI ALLE2NXS</td>
</tr>
<tr>
<td>TLBI ALLE2IS, TLBI ALLE2ISNXS</td>
<td>TLBI ALLE2IS, TLBI ALLE2ISNXS</td>
</tr>
<tr>
<td>TLBI ALLE2OS, TLBI ALLE2OSNXS</td>
<td>TLBI ALLE2OS, TLBI ALLE2OSNXS</td>
</tr>
<tr>
<td>TLBI ALLE3, TLBI ALLE3NXS</td>
<td>TLBI ALLE3, TLBI ALLE3NXS</td>
</tr>
<tr>
<td>TLBI ALLE3IS, TLBI ALLE3ISNXS</td>
<td>TLBI ALLE3IS, TLBI ALLE3ISNXS</td>
</tr>
<tr>
<td>TLBI ALLE3OS, TLBI ALLE3OSNXS</td>
<td>TLBI ALLE3OS, TLBI ALLE3OSNXS</td>
</tr>
<tr>
<td>TLBI ASIDE1, TLBI ASIDE1NXS</td>
<td>TLBI ASIDE1, TLBI ASIDE1NXS</td>
</tr>
<tr>
<td>TLBI ASIDE1IS, TLBI ASIDE1ISNXS</td>
<td>TLBI ASIDE1IS, TLBI ASIDE1ISNXS</td>
</tr>
<tr>
<td>TLBI ASIDE1OS, TLBI ASIDE1OSNXS</td>
<td>TLBI ASIDE1OS, TLBI ASIDE1OSNXS</td>
</tr>
<tr>
<td>TLBI IPAS2E1, TLBI IPAS2E1NXS</td>
<td>TLBI IPAS2E1, TLBI IPAS2E1NXS</td>
</tr>
<tr>
<td>TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS</td>
<td>TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS</td>
</tr>
<tr>
<td>TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS</td>
<td>TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS</td>
</tr>
<tr>
<td>TLBI IPAS2E1LE1, TLBI IPAS2E1LE1NXS</td>
<td>TLBI IPAS2E1LE1, TLBI IPAS2E1LE1NXS</td>
</tr>
<tr>
<td>TLBI IPAS2E1LE1IS, TLBI IPAS2E1LE1ISNXS</td>
<td>TLBI IPAS2E1LE1IS, TLBI IPAS2E1LE1ISNXS</td>
</tr>
<tr>
<td>TLBI IPAS2E1LE1OS, TLBI IPAS2E1LE1OSNXS</td>
<td>TLBI IPAS2E1LE1OS, TLBI IPAS2E1LE1OSNXS</td>
</tr>
<tr>
<td>TLBI PAALL</td>
<td>TLBI PAALL</td>
</tr>
<tr>
<td>TLBI PAALLOS</td>
<td>TLBI PAALLOS</td>
</tr>
<tr>
<td>TLBI RIPAS2E1, TLBI RIPAS2E1NXS</td>
<td>TLBI RIPAS2E1, TLBI RIPAS2E1NXS</td>
</tr>
<tr>
<td>TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS</td>
<td>TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS</td>
</tr>
<tr>
<td>TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS</td>
<td>TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS</td>
</tr>
<tr>
<td>TLBI RIPAS2E1LE1, TLBI RIPAS2E1LE1NXS</td>
<td>TLBI RIPAS2E1LE1, TLBI RIPAS2E1LE1NXS</td>
</tr>
<tr>
<td>TLBI RIPAS2E1LE1IS, TLBI RIPAS2E1LE1ISNXS</td>
<td>TLBI RIPAS2E1LE1IS, TLBI RIPAS2E1LE1ISNXS</td>
</tr>
</tbody>
</table>
## Table K17-23 TLB maintenance system instructions (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLBI RIPA2LE1OS, TLBI RIPA2LE1OSNXS</td>
<td>TLBI RIPA2LE1OS, TLBI RIPA2LE1OSNXS</td>
</tr>
<tr>
<td>TLBI RPAOS</td>
<td>TLBI RPAOS</td>
</tr>
<tr>
<td>TLBI RVAE1, TLBI RVAE1INXS</td>
<td>TLBI RVAE1, TLBI RVAE1INXS</td>
</tr>
<tr>
<td>TLBI RVAE1IS, TLBI RVAE1ISNXS</td>
<td>TLBI RVAE1IS, TLBI RVAE1ISNXS</td>
</tr>
<tr>
<td>TLBI RVAE1OS, TLBI RVAE1OSNXS</td>
<td>TLBI RVAE1OS, TLBI RVAE1OSNXS</td>
</tr>
<tr>
<td>TLBI RVALE1, TLBI RVALE1INXS</td>
<td>TLBI RVALE1, TLBI RVALE1INXS</td>
</tr>
<tr>
<td>TLBI RVALE1IS, TLBI RVALE1ISNXS</td>
<td>TLBI RVALE1IS, TLBI RVALE1ISNXS</td>
</tr>
<tr>
<td>TLBI RVALE1OS, TLBI RVALE1OSNXS</td>
<td>TLBI RVALE1OS, TLBI RVALE1OSNXS</td>
</tr>
<tr>
<td>TLBI RVALE2, TLBI RVALE2INXS</td>
<td>TLBI RVALE2, TLBI RVALE2INXS</td>
</tr>
<tr>
<td>TLBI RVALE2IS, TLBI RVALE2ISNXS</td>
<td>TLBI RVALE2IS, TLBI RVALE2ISNXS</td>
</tr>
<tr>
<td>TLBI RVALE2OS, TLBI RVALE2OSNXS</td>
<td>TLBI RVALE2OS, TLBI RVALE2OSNXS</td>
</tr>
<tr>
<td>TLBI RVAE3, TLBI RVAE3INXS</td>
<td>TLBI RVAE3, TLBI RVAE3INXS</td>
</tr>
<tr>
<td>TLBI RVAE3IS, TLBI RVAE3ISNXS</td>
<td>TLBI RVAE3IS, TLBI RVAE3ISNXS</td>
</tr>
<tr>
<td>TLBI RVAE3OS, TLBI RVAE3OSNXS</td>
<td>TLBI RVAE3OS, TLBI RVAE3OSNXS</td>
</tr>
<tr>
<td>TLBI RVALE1, TLBI RVALE1INXS</td>
<td>TLBI RVALE1, TLBI RVALE1INXS</td>
</tr>
<tr>
<td>TLBI RVALE1IS, TLBI RVALE1ISNXS</td>
<td>TLBI RVALE1IS, TLBI RVALE1ISNXS</td>
</tr>
<tr>
<td>TLBI RVALE1OS, TLBI RVALE1OSNXS</td>
<td>TLBI RVALE1OS, TLBI RVALE1OSNXS</td>
</tr>
<tr>
<td>TLBI RVALE2, TLBI RVALE2INXS</td>
<td>TLBI RVALE2, TLBI RVALE2INXS</td>
</tr>
<tr>
<td>TLBI RVALE2IS, TLBI RVALE2ISNXS</td>
<td>TLBI RVALE2IS, TLBI RVALE2ISNXS</td>
</tr>
<tr>
<td>TLBI RVALE2OS, TLBI RVALE2OSNXS</td>
<td>TLBI RVALE2OS, TLBI RVALE2OSNXS</td>
</tr>
<tr>
<td>TLBI RVALE3, TLBI RVALE3INXS</td>
<td>TLBI RVALE3, TLBI RVALE3INXS</td>
</tr>
<tr>
<td>TLBI RVALE3IS, TLBI RVALE3ISNXS</td>
<td>TLBI RVALE3IS, TLBI RVALE3ISNXS</td>
</tr>
<tr>
<td>TLBI RVALE3OS, TLBI RVALE3OSNXS</td>
<td>TLBI RVALE3OS, TLBI RVALE3OSNXS</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>TLBI VAAE1, TLBI VAAE1NXS</td>
<td>TLBI VAAE1, TLBI VAAE1NXS</td>
</tr>
<tr>
<td>TLBI VAAE1IS, TLBI VAAE1ISNXS</td>
<td>TLBI VAAE1IS, TLBI VAAE1ISNXS</td>
</tr>
<tr>
<td>TLBI VAAE1OS, TLBI VAAE1OSNXS</td>
<td>TLBI VAAE1OS, TLBI VAAE1OSNXS</td>
</tr>
<tr>
<td>TLBI VAALE1, TLBI VAALE1NXS</td>
<td>TLBI VAALE1, TLBI VAALE1NXS</td>
</tr>
<tr>
<td>TLBI VAALE1IS, TLBI VAALE1ISNXS</td>
<td>TLBI VAALE1IS, TLBI VAALE1ISNXS</td>
</tr>
<tr>
<td>TLBI VAALE1OS, TLBI VAALE1OSNXS</td>
<td>TLBI VAALE1OS, TLBI VAALE1OSNXS</td>
</tr>
<tr>
<td>TLBI VAE1, TLBI VAE1NXS</td>
<td>TLBI VAE1, TLBI VAE1NXS</td>
</tr>
<tr>
<td>TLBI VAE1IS, TLBI VAE1ISNXS</td>
<td>TLBI VAE1IS, TLBI VAE1ISNXS</td>
</tr>
<tr>
<td>TLBI VAE1OS, TLBI VAE1OSNXS</td>
<td>TLBI VAE1OS, TLBI VAE1OSNXS</td>
</tr>
<tr>
<td>TLBI VAE2, TLBI VAE2NXS</td>
<td>TLBI VAE2, TLBI VAE2NXS</td>
</tr>
<tr>
<td>TLBI VAE2IS, TLBI VAE2ISNXS</td>
<td>TLBI VAE2IS, TLBI VAE2ISNXS</td>
</tr>
<tr>
<td>TLBI VAE2OS, TLBI VAE2OSNXS</td>
<td>TLBI VAE2OS, TLBI VAE2OSNXS</td>
</tr>
<tr>
<td>TLBI VAE3, TLBI VAE3NXS</td>
<td>TLBI VAE3, TLBI VAE3NXS</td>
</tr>
<tr>
<td>TLBI VAE3IS, TLBI VAE3ISNXS</td>
<td>TLBI VAE3IS, TLBI VAE3ISNXS</td>
</tr>
<tr>
<td>TLBI VAE3OS, TLBI VAE3OSNXS</td>
<td>TLBI VAE3OS, TLBI VAE3OSNXS</td>
</tr>
<tr>
<td>TLBI VALE1, TLBI VALE1NXS</td>
<td>TLBI VALE1, TLBI VALE1NXS</td>
</tr>
<tr>
<td>TLBI VALE1IS, TLBI VALE1ISNXS</td>
<td>TLBI VALE1IS, TLBI VALE1ISNXS</td>
</tr>
<tr>
<td>TLBI VALE1OS, TLBI VALE1OSNXS</td>
<td>TLBI VALE1OS, TLBI VALE1OSNXS</td>
</tr>
<tr>
<td>TLBI VALE2, TLBI VALE2NXS</td>
<td>TLBI VALE2, TLBI VALE2NXS</td>
</tr>
<tr>
<td>TLBI VALE2IS, TLBI VALE2ISNXS</td>
<td>TLBI VALE2IS, TLBI VALE2ISNXS</td>
</tr>
<tr>
<td>TLBI VALE2OS, TLBI VALE2OSNXS</td>
<td>TLBI VALE2OS, TLBI VALE2OSNXS</td>
</tr>
<tr>
<td>TLBI VALE3, TLBI VALE3NXS</td>
<td>TLBI VALE3, TLBI VALE3NXS</td>
</tr>
<tr>
<td>TLBI VALE3IS, TLBI VALE3ISNXS</td>
<td>TLBI VALE3IS, TLBI VALE3ISNXS</td>
</tr>
<tr>
<td>TLBI VALE3OS, TLBI VALE3OSNXS</td>
<td>TLBI VALE3OS, TLBI VALE3OSNXS</td>
</tr>
<tr>
<td>TLBI VMALLE1, TLBI VMALLE1NXS</td>
<td>TLBI VMALLE1, TLBI VMALLE1NXS</td>
</tr>
<tr>
<td>TLBI VMALLE1IS, TLBI VMALLE1ISNXS</td>
<td>TLBI VMALLE1IS, TLBI VMALLE1ISNXS</td>
</tr>
<tr>
<td>TLBI VMALLE1OS, TLBI VMALLE1OSNXS</td>
<td>TLBI VMALLE1OS, TLBI VMALLE1OSNXS</td>
</tr>
</tbody>
</table>
### K17.3.15 Prediction restriction System instructions

This section is an index to the registers in the prediction restriction instructions functional group.

<table>
<thead>
<tr>
<th>System instruction</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFP RCTX</td>
<td>CFP RCTX</td>
</tr>
<tr>
<td>CPP RCTX</td>
<td>CPP RCTX</td>
</tr>
<tr>
<td>DVP RCTX</td>
<td>DVP RCTX</td>
</tr>
</tbody>
</table>

### K17.3.16 Base system registers

This section is an index to the registers in the functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCDATA_EL1</td>
<td>ACCDATA_EL1</td>
</tr>
<tr>
<td>ACTLR_EL1</td>
<td>ACTLR_EL1</td>
</tr>
<tr>
<td>ACTLR_EL2</td>
<td>ACTLR_EL2</td>
</tr>
<tr>
<td>ACTLR_EL3</td>
<td>ACTLR_EL3</td>
</tr>
<tr>
<td>AFSR0_EL1</td>
<td>AFSR0_EL1</td>
</tr>
<tr>
<td>AFSR0_EL2</td>
<td>AFSR0_EL2</td>
</tr>
<tr>
<td>AFSR0_EL3</td>
<td>AFSR0_EL3</td>
</tr>
<tr>
<td>AFSR1_EL1</td>
<td>AFSR1_EL1</td>
</tr>
<tr>
<td>AFSR1_EL2</td>
<td>AFSR1_EL2</td>
</tr>
<tr>
<td>AFSR1_EL3</td>
<td>AFSR1_EL3</td>
</tr>
<tr>
<td>AIDR_EL1</td>
<td>AIDR_EL1</td>
</tr>
<tr>
<td>ALLINT</td>
<td>ALLINT</td>
</tr>
<tr>
<td>APDAKeyHi_EL1</td>
<td>APDAKeyHi_EL1</td>
</tr>
<tr>
<td>APDAKeyLo_EL1</td>
<td>APDAKeyLo_EL1</td>
</tr>
</tbody>
</table>
## Table K17-25 Base system registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>APDBKeyHi_EL1</td>
<td>APDBKeyHi_EL1</td>
</tr>
<tr>
<td>APDBKeyLo_EL1</td>
<td>APDBKeyLo_EL1</td>
</tr>
<tr>
<td>APGAKeyHi_EL1</td>
<td>APGAKeyHi_EL1</td>
</tr>
<tr>
<td>APGAKeyLo_EL1</td>
<td>APGAKeyLo_EL1</td>
</tr>
<tr>
<td>APIAKeyHi_EL1</td>
<td>APIAKeyHi_EL1</td>
</tr>
<tr>
<td>APIAKeyLo_EL1</td>
<td>APIAKeyLo_EL1</td>
</tr>
<tr>
<td>APIBKeyHi_EL1</td>
<td>APIBKeyHi_EL1</td>
</tr>
<tr>
<td>APIBKeyLo_EL1</td>
<td>APIBKeyLo_EL1</td>
</tr>
<tr>
<td>CNTHCTL_EL2</td>
<td>CNTHCTL_EL2</td>
</tr>
<tr>
<td>CNTHP_CTL_EL2</td>
<td>CNTHP_CTL_EL2</td>
</tr>
<tr>
<td>CNTHP_CVAL_EL2</td>
<td>CNTHP_CVAL_EL2</td>
</tr>
<tr>
<td>CNTHP_TVAL_EL2</td>
<td>CNTHP_TVAL_EL2</td>
</tr>
<tr>
<td>CNTHPS_CTL_EL2</td>
<td>CNTHPS_CTL_EL2</td>
</tr>
<tr>
<td>CNTHPS_CVAL_EL2</td>
<td>CNTHPS_CVAL_EL2</td>
</tr>
<tr>
<td>CNTHPS_TVAL_EL2</td>
<td>CNTHPS_TVAL_EL2</td>
</tr>
<tr>
<td>CNTVOFF_EL2</td>
<td>CNTVOFF_EL2</td>
</tr>
<tr>
<td>CPACR_EL1</td>
<td>CPACR_EL1</td>
</tr>
<tr>
<td>CPTR_EL2</td>
<td>CPTR_EL2</td>
</tr>
<tr>
<td>CPTR_EL3</td>
<td>CPTR_EL3</td>
</tr>
<tr>
<td>CurrentEL</td>
<td>CurrentEL</td>
</tr>
<tr>
<td>DAIF</td>
<td>DAIF</td>
</tr>
<tr>
<td>DIT</td>
<td>DIT</td>
</tr>
<tr>
<td>ESR_EL1</td>
<td>ESR_EL1</td>
</tr>
<tr>
<td>ESR_EL2</td>
<td>ESR_EL2</td>
</tr>
<tr>
<td>ESR_EL3</td>
<td>ESR_EL3</td>
</tr>
<tr>
<td>FAR_EL1</td>
<td>FAR_EL1</td>
</tr>
<tr>
<td>FAR_EL2</td>
<td>FAR_EL2</td>
</tr>
<tr>
<td>FAR_EL3</td>
<td>FAR_EL3</td>
</tr>
<tr>
<td>FPCR</td>
<td>FPCR</td>
</tr>
<tr>
<td>FPEXC32_EL2</td>
<td>FPEXC32_EL2</td>
</tr>
<tr>
<td>FPSR</td>
<td>FPSR</td>
</tr>
<tr>
<td>GCR_EL1</td>
<td>GCR_EL1</td>
</tr>
</tbody>
</table>
## Table K17-25 Base system registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>HACR_EL2</td>
<td>HACR_EL2</td>
</tr>
<tr>
<td>HAFGRTR_EL2</td>
<td>HAFGRTR_EL2</td>
</tr>
<tr>
<td>HCR_EL2</td>
<td>HCR_EL2</td>
</tr>
<tr>
<td>HCRX_EL2</td>
<td>HCRX_EL2</td>
</tr>
<tr>
<td>HDFGRTR_EL2</td>
<td>HDFGRTR_EL2</td>
</tr>
<tr>
<td>HDFGWTR_EL2</td>
<td>HDFGWTR_EL2</td>
</tr>
<tr>
<td>HFGITR_EL2</td>
<td>HFGITR_EL2</td>
</tr>
<tr>
<td>HFGRTR_EL2</td>
<td>HFGRTR_EL2</td>
</tr>
<tr>
<td>HFGWTR_EL2</td>
<td>HFGWTR_EL2</td>
</tr>
<tr>
<td>HPFAR_EL2</td>
<td>HPFAR_EL2</td>
</tr>
<tr>
<td>HSTR_EL2</td>
<td>HSTR_EL2</td>
</tr>
<tr>
<td>IFSR32_EL2</td>
<td>IFSR32_EL2</td>
</tr>
<tr>
<td>ISR_EL1</td>
<td>ISR_EL1</td>
</tr>
<tr>
<td>MDCR_EL2</td>
<td>MDCR_EL2</td>
</tr>
<tr>
<td>MDCR_EL3</td>
<td>MDCR_EL3</td>
</tr>
<tr>
<td>MPAM0_EL1</td>
<td>MPAM0_EL1</td>
</tr>
<tr>
<td>MPAM1_EL1</td>
<td>MPAM1_EL1</td>
</tr>
<tr>
<td>MPAM2_EL2</td>
<td>MPAM2_EL2</td>
</tr>
<tr>
<td>MPAM3_EL3</td>
<td>MPAM3_EL3</td>
</tr>
<tr>
<td>MPAMHCR_EL2</td>
<td>MPAMHCR_EL2</td>
</tr>
<tr>
<td>MPAMSM_EL1</td>
<td>MPAMSM_EL1</td>
</tr>
<tr>
<td>MPAMVPM0_EL2</td>
<td>MPAMVPM0_EL2</td>
</tr>
<tr>
<td>MPAMVPM1_EL2</td>
<td>MPAMVPM1_EL2</td>
</tr>
<tr>
<td>MPAMVPM2_EL2</td>
<td>MPAMVPM2_EL2</td>
</tr>
<tr>
<td>MPAMVPM3_EL2</td>
<td>MPAMVPM3_EL2</td>
</tr>
<tr>
<td>MPAMVPM4_EL2</td>
<td>MPAMVPM4_EL2</td>
</tr>
<tr>
<td>MPAMVPM5_EL2</td>
<td>MPAMVPM5_EL2</td>
</tr>
<tr>
<td>MPAMVPM6_EL2</td>
<td>MPAMVPM6_EL2</td>
</tr>
<tr>
<td>MPAMVPM7_EL2</td>
<td>MPAMVPM7_EL2</td>
</tr>
<tr>
<td>MPAMVPMV_EL2</td>
<td>MPAMVPMV_EL2</td>
</tr>
<tr>
<td>MVFR0_EL1</td>
<td>MVFR0_EL1</td>
</tr>
<tr>
<td>MVFR1_EL1</td>
<td>MVFR1_EL1</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>MVFR2_EL1</td>
<td>MVFR2_EL1</td>
</tr>
<tr>
<td>NZCV</td>
<td>NZCV</td>
</tr>
<tr>
<td>PAN</td>
<td>PAN</td>
</tr>
<tr>
<td>PAR_EL1</td>
<td>PAR_EL1</td>
</tr>
<tr>
<td>PMBIDR_EL1</td>
<td>PMBIDR_EL1</td>
</tr>
<tr>
<td>PMLIMITR_EL1</td>
<td>PMLIMITR_EL1</td>
</tr>
<tr>
<td>PMBPTR_EL1</td>
<td>PMBPTR_EL1</td>
</tr>
<tr>
<td>PMBSR_EL1</td>
<td>PMBSR_EL1</td>
</tr>
<tr>
<td>PMSCR_EL1</td>
<td>PMSCR_EL1</td>
</tr>
<tr>
<td>PMSCR_EL2</td>
<td>PMSCR_EL2</td>
</tr>
<tr>
<td>PMSEVFR_EL1</td>
<td>PMSEVFR_EL1</td>
</tr>
<tr>
<td>PMSFCR_EL1</td>
<td>PMSFCR_EL1</td>
</tr>
<tr>
<td>PMSICR_EL1</td>
<td>PMSICR_EL1</td>
</tr>
<tr>
<td>PMSIDR_EL1</td>
<td>PMSIDR_EL1</td>
</tr>
<tr>
<td>PMSIIRR_EL1</td>
<td>PMSIIRR_EL1</td>
</tr>
<tr>
<td>PMSLATFR_EL1</td>
<td>PMSLATFR_EL1</td>
</tr>
<tr>
<td>PMSNEVFR_EL1</td>
<td>PMSNEVFR_EL1</td>
</tr>
<tr>
<td>RGSR_EL1</td>
<td>RGSR_EL1</td>
</tr>
<tr>
<td>RMR_EL1</td>
<td>RMR_EL1</td>
</tr>
<tr>
<td>RMR_EL2</td>
<td>RMR_EL2</td>
</tr>
<tr>
<td>RMR_EL3</td>
<td>RMR_EL3</td>
</tr>
<tr>
<td>RNDR</td>
<td>RNDR</td>
</tr>
<tr>
<td>RNDRRS</td>
<td>RNDRRS</td>
</tr>
<tr>
<td>RVBAR_EL1</td>
<td>RVBAR_EL1</td>
</tr>
<tr>
<td>RVBAR_EL2</td>
<td>RVBAR_EL2</td>
</tr>
<tr>
<td>RVBAR_EL3</td>
<td>RVBAR_EL3</td>
</tr>
<tr>
<td>S3_&lt;op1&gt;<em>.</em>&lt;Cn&gt;.<em>&lt;Cm&gt;.</em>&lt;op2&gt;</td>
<td>S3_&lt;op1&gt;<em>.</em>&lt;Cn&gt;.<em>&lt;Cm&gt;.</em>&lt;op2&gt;</td>
</tr>
<tr>
<td>SCR_EL3</td>
<td>SCR_EL3</td>
</tr>
<tr>
<td>SCTLR_EL1</td>
<td>SCTLR_EL1</td>
</tr>
<tr>
<td>SCTLR_EL2</td>
<td>SCTLR_EL2</td>
</tr>
<tr>
<td>SCTLR_EL3</td>
<td>SCTLR_EL3</td>
</tr>
<tr>
<td>SCXTNUM_EL0</td>
<td>SCXTNUM_EL0</td>
</tr>
</tbody>
</table>
### Table K17-25 Base system registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCXTNUM_EL1</td>
<td>SCXTNUM_EL1</td>
</tr>
<tr>
<td>SCXTNUM_EL2</td>
<td>SCXTNUM_EL2</td>
</tr>
<tr>
<td>SCXTNUM_EL3</td>
<td>SCXTNUM_EL3</td>
</tr>
<tr>
<td>SDER32_EL2</td>
<td>SDER32_EL2</td>
</tr>
<tr>
<td>SDER32_EL3</td>
<td>SDER32_EL3</td>
</tr>
<tr>
<td>SMCR_EL1</td>
<td>SMCR_EL1</td>
</tr>
<tr>
<td>SMCR_EL2</td>
<td>SMCR_EL2</td>
</tr>
<tr>
<td>SMCR_EL3</td>
<td>SMCR_EL3</td>
</tr>
<tr>
<td>SMPRI_EL1</td>
<td>SMPRI_EL1</td>
</tr>
<tr>
<td>SMPRIMAP_EL2</td>
<td>SMPRIMAP_EL2</td>
</tr>
<tr>
<td>SPSel</td>
<td>SPSel</td>
</tr>
<tr>
<td>SSBS</td>
<td>SSBS</td>
</tr>
<tr>
<td>SVCR</td>
<td>SVCR</td>
</tr>
<tr>
<td>TCO</td>
<td>TCO</td>
</tr>
<tr>
<td>TFSR_EL1</td>
<td>TFSR_EL1</td>
</tr>
<tr>
<td>TFSR_EL2</td>
<td>TFSR_EL2</td>
</tr>
<tr>
<td>TFSR_EL3</td>
<td>TFSR_EL3</td>
</tr>
<tr>
<td>TFSRE0_EL1</td>
<td>TFSRE0_EL1</td>
</tr>
<tr>
<td>TPIDR2_EL0</td>
<td>TPIDR2_EL0</td>
</tr>
<tr>
<td>TPIDR_EL0</td>
<td>TPIDR_EL0</td>
</tr>
<tr>
<td>TPIDR_EL1</td>
<td>TPIDR_EL1</td>
</tr>
<tr>
<td>TPIDR_EL2</td>
<td>TPIDR_EL2</td>
</tr>
<tr>
<td>TPIDR_EL3</td>
<td>TPIDR_EL3</td>
</tr>
<tr>
<td>TPIDRRO_EL0</td>
<td>TPIDRRO_EL0</td>
</tr>
<tr>
<td>TRCAUTHSTATUS</td>
<td>TRCAUTHSTATUS</td>
</tr>
<tr>
<td>TRCDEVARCH</td>
<td>TRCDEVARCH</td>
</tr>
<tr>
<td>TRCDEVID</td>
<td>TRCDEVID</td>
</tr>
<tr>
<td>TRCOSLSR</td>
<td>TRCOSLSR</td>
</tr>
<tr>
<td>UAO</td>
<td>UAO</td>
</tr>
<tr>
<td>VBAR_EL1</td>
<td>VBAR_EL1</td>
</tr>
<tr>
<td>VBAR_EL2</td>
<td>VBAR_EL2</td>
</tr>
<tr>
<td>VBAR_EL3</td>
<td>VBAR_EL3</td>
</tr>
</tbody>
</table>
### Table K17-25 Base system registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMPIDR_EL2</td>
<td>VMPIDR_EL2</td>
</tr>
<tr>
<td>VNCR_EL2</td>
<td>VNCR_EL2</td>
</tr>
<tr>
<td>VPIDR_EL2</td>
<td>VPIDR_EL2</td>
</tr>
<tr>
<td>VSTCR_EL2</td>
<td>VSTCR_EL2</td>
</tr>
<tr>
<td>VSTTBR_EL2</td>
<td>VSTTBR_EL2</td>
</tr>
<tr>
<td>ZCR_EL1</td>
<td>ZCR_EL1</td>
</tr>
<tr>
<td>ZCR_EL2</td>
<td>ZCR_EL2</td>
</tr>
<tr>
<td>ZCR_EL3</td>
<td>ZCR_EL3</td>
</tr>
</tbody>
</table>
## K17.4 Alphabetical index of AArch32 registers and System instructions

This section is an index of AArch32 registers and System instructions in alphabetical order.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTLR</td>
<td>ACTLR, Auxiliary Control Register on page G8-9338</td>
</tr>
<tr>
<td>ACTLR2</td>
<td>ACTLR2, Auxiliary Control Register 2 on page G8-9340</td>
</tr>
<tr>
<td>ADFSR</td>
<td>ADFSR, Auxiliary Data Fault Status Register on page G8-9342</td>
</tr>
<tr>
<td>AIDR</td>
<td>AIDR, Auxiliary ID Register on page G8-9344</td>
</tr>
<tr>
<td>AIFSR</td>
<td>AIFSR, Auxiliary Instruction Fault Status Register on page G8-9345</td>
</tr>
<tr>
<td>AMAIR0</td>
<td>AMAIR0, Auxiliary Memory Attribute Indirection Register 0 on page G8-9347</td>
</tr>
<tr>
<td>AMAIR1</td>
<td>AMAIR1, Auxiliary Memory Attribute Indirection Register 1 on page G8-9350</td>
</tr>
<tr>
<td>AMCFGR</td>
<td>AMCFGR, Activity Monitors Configuration Register on page G8-10048</td>
</tr>
<tr>
<td>AMCGCR</td>
<td>AMCGCR, Activity Monitors Counter Group Configuration Register on page G8-10051</td>
</tr>
<tr>
<td>AMCNTENCLR0</td>
<td>AMCNTENCLR0, Activity Monitors Count Enable Clear Register 0 on page G8-10053</td>
</tr>
<tr>
<td>AMCNTENCLR1</td>
<td>AMCNTENCLR1, Activity Monitors Count Enable Clear Register 1 on page G8-10056</td>
</tr>
<tr>
<td>AMCNTENSET0</td>
<td>AMCNTENSET0, Activity Monitors Count Enable Set Register 0 on page G8-10059</td>
</tr>
<tr>
<td>AMCNTENSET1</td>
<td>AMCNTENSET1, Activity Monitors Count Enable Set Register 1 on page G8-10062</td>
</tr>
<tr>
<td>AMCR</td>
<td>AMCR, Activity Monitors Control Register on page G8-10065</td>
</tr>
<tr>
<td>AMEVCTR0&lt;n&gt;</td>
<td>AMEVCTR0&lt;n&gt;, Activity Monitors Event Counter Registers 0, n = 0 - 3 on page G8-10068</td>
</tr>
<tr>
<td>AMEVCTR1&lt;n&gt;</td>
<td>AMEVCTR1&lt;n&gt;, Activity Monitors Event Counter Registers 1, n = 0 - 15 on page G8-10071</td>
</tr>
<tr>
<td>AMEVTYPER0&lt;n&gt;</td>
<td>AMEVTYPER0&lt;n&gt;, Activity Monitors Event Type Registers 0, n = 0 - 3 on page G8-10074</td>
</tr>
<tr>
<td>AMEVTYPER1&lt;n&gt;</td>
<td>AMEVTYPER1&lt;n&gt;, Activity Monitors Event Type Registers 1, n = 0 - 15 on page G8-10077</td>
</tr>
<tr>
<td>AMUSERENR</td>
<td>AMUSERENR, Activity Monitors User Enable Register on page G8-10080</td>
</tr>
<tr>
<td>APSR</td>
<td>APSR, Application Program Status Register on page G8-9353</td>
</tr>
<tr>
<td>ATS12NSOPR</td>
<td>ATS12NSOPR, Address Translate Stages 1 and 2 Non-secure Only PL1 Read on page G8-9355</td>
</tr>
<tr>
<td>ATS12NSOPW</td>
<td>ATS12NSOPW, Address Translate Stages 1 and 2 Non-secure Only PL1 Write on page G8-9356</td>
</tr>
<tr>
<td>ATS12NSOUR</td>
<td>ATS12NSOUR, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Read on page G8-9357</td>
</tr>
<tr>
<td>ATS12NSOUW</td>
<td>ATS12NSOUW, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Write on page G8-9358</td>
</tr>
<tr>
<td>ATS1CPR</td>
<td>ATS1CPR, Address Translate Stage 1 Current state PL1 Read on page G8-9359</td>
</tr>
<tr>
<td>ATS1CPRP</td>
<td>ATS1CPRP, Address Translate Stage 1 Current state PL1 Read PAN on page G8-9360</td>
</tr>
</tbody>
</table>
### Table K17-26 Alphabetical index of AArch32 Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATS1CPW</td>
<td>ATS1CPW, Address Translate Stage 1 Current state PL1 Write on page G8-9361</td>
</tr>
<tr>
<td>ATS1CPWP</td>
<td>ATS1CPWP, Address Translate Stage 1 Current state PL1 Write PAN on page G8-9362</td>
</tr>
<tr>
<td>ATS1CUR</td>
<td>ATS1CUR, Address Translate Stage 1 Current state Unprivileged Read on page G8-9363</td>
</tr>
<tr>
<td>ATS1CUW</td>
<td>ATS1CUW, Address Translate Stage 1 Current state Unprivileged Write on page G8-9364</td>
</tr>
<tr>
<td>ATS1HR</td>
<td>ATS1HR, Address Translate Stage 1 Hyp mode Read on page G8-9365</td>
</tr>
<tr>
<td>ATS1HW</td>
<td>ATS1HW, Address Translate Stage 1 Hyp mode Write on page G8-9367</td>
</tr>
<tr>
<td>BPIALL</td>
<td>BPIALL, Branch Predictor Invalidate All on page G8-9369</td>
</tr>
<tr>
<td>BPIALLIS</td>
<td>BPIALLIS, Branch Predictor Invalidate All, Inner Shareable on page G8-9370</td>
</tr>
<tr>
<td>BPIIMVA</td>
<td>BPIIMVA, Branch Predictor Invalidate by V4 on page G8-9371</td>
</tr>
<tr>
<td>CCSIDR</td>
<td>CCSIDR, Current Cache Size ID Register on page G8-9372</td>
</tr>
<tr>
<td>CCSIDR2</td>
<td>CCSIDR2, Current Cache Size ID Register 2 on page G8-9375</td>
</tr>
<tr>
<td>CFPRCTX</td>
<td>CFPRCTX, Control Flow Prediction Restriction by Context on page G8-9377</td>
</tr>
<tr>
<td>CLIDR</td>
<td>CLIDR, Cache Level ID Register on page G8-9380</td>
</tr>
<tr>
<td>CNTFRQ</td>
<td>CNTFRQ, Counter-timer Frequency register on page G8-10145</td>
</tr>
<tr>
<td>CNTCTL</td>
<td>CNTCTL, Counter-timer Hyp Control register on page G8-10147</td>
</tr>
<tr>
<td>CNTHP_CTL</td>
<td>CNTHP_CTL, Counter-timer Hyp Physical Timer Control register on page G8-10150</td>
</tr>
<tr>
<td>CNTHP_CVAL</td>
<td>CNTHP_CVAL, Counter-timer Hyp Physical CompareValue register on page G8-10154</td>
</tr>
<tr>
<td>CNTHP_TV</td>
<td>CNTHP_TV, Counter-timer Hyp Physical TimerValue register on page G8-10158</td>
</tr>
<tr>
<td>CNTHPS_CTL</td>
<td>CNTHPS_CTL, Counter-timer Secure Physical Timer Control Register (EL2) on page G8-10163</td>
</tr>
<tr>
<td>CNTHPS_CVAL</td>
<td>CNTHPS_CVAL, Counter-timer Secure Physical Timer CompareValue Register (EL2) on page G8-10167</td>
</tr>
<tr>
<td>CNTHPS_TV</td>
<td>CNTHPS_TV, Counter-timer Secure Physical TimerValue Register (EL2) on page G8-10170</td>
</tr>
<tr>
<td>CNTHV_CTL</td>
<td>CNTHV_CTL, Counter-timer Virtual Timer Control register (EL2) on page G8-10174</td>
</tr>
<tr>
<td>CNTHV_CVAL</td>
<td>CNTHV_CVAL, Counter-timer Virtual Timer CompareValue register (EL2) on page G8-10177</td>
</tr>
<tr>
<td>CNTHV_TV</td>
<td>CNTHV_TV, Counter-timer Virtual TimerValue register (EL2) on page G8-10180</td>
</tr>
<tr>
<td>CNTHVS_CTL</td>
<td>CNTHVS_CTL, Counter-timer Secure Virtual Timer Control Register (EL2) on page G8-10183</td>
</tr>
<tr>
<td>CNTHVS_CVAL</td>
<td>CNTHVS_CVAL, Counter-timer Secure Virtual Timer CompareValue Register (EL2) on page G8-10186</td>
</tr>
<tr>
<td>CNTHVS_TV</td>
<td>CNTHVS_TV, Counter-timer Secure Virtual TimerValue Register (EL2) on page G8-10189</td>
</tr>
<tr>
<td>CNTKCTL</td>
<td>CNTKCTL, Counter-timer Kernel Control register on page G8-10192</td>
</tr>
</tbody>
</table>
Table K17-26 Alphabetical index of AArch32 Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNTP_CTL</td>
<td>CNTP_CTL, Counter-timer Physical Timer Control register on page G8-10195</td>
</tr>
<tr>
<td>CNTP_CVAL</td>
<td>CNTP_CVAL, Counter-timer Physical Timer CompareValue register on page G8-10199</td>
</tr>
<tr>
<td>CNTP_TVAL</td>
<td>CNTP_TVAL, Counter-timer Physical TimerTimerValue register on page G8-10202</td>
</tr>
<tr>
<td>CNTPCT</td>
<td>CNTPCT, Counter-timer Physical Count register on page G8-10206</td>
</tr>
<tr>
<td>CNTPCTSS</td>
<td>CNTPCTSS, Counter-timer Self-Synchronized Physical Count register on page G8-10208</td>
</tr>
<tr>
<td>CNTV_CTL</td>
<td>CNTV_CTL, Counter-timer Virtual Timer Control register on page G8-10210</td>
</tr>
<tr>
<td>CNTV_CVAL</td>
<td>CNTV_CVAL, Counter-timer Virtual Timer CompareValue register on page G8-10213</td>
</tr>
<tr>
<td>CNTV_TVAL</td>
<td>CNTV_TVAL, Counter-timer Virtual TimerTimerValue register on page G8-10216</td>
</tr>
<tr>
<td>CNTVCT</td>
<td>CNTVCT, Counter-timer Virtual Count register on page G8-10219</td>
</tr>
<tr>
<td>CNTVCTSS</td>
<td>CNTVCTSS, Counter-timer Self-Synchronized Virtual Counter register on page G8-10221</td>
</tr>
<tr>
<td>CNTVOFF</td>
<td>CNTVOFF, Counter-timer Virtual Offset register on page G8-10223</td>
</tr>
<tr>
<td>CONTEXTIDR</td>
<td>CONTEXTIDR, Context ID Register on page G8-9382</td>
</tr>
<tr>
<td>CP15DMB</td>
<td>CP15DMB, Data Memory Barrier System instruction on page G8-9385</td>
</tr>
<tr>
<td>CP15DSB</td>
<td>CP15DSB, Data Synchronization Barrier System instruction on page G8-9387</td>
</tr>
<tr>
<td>CP15ISB</td>
<td>CP15ISB, Instruction Synchronization Barrier System instruction on page G8-9389</td>
</tr>
<tr>
<td>CPACR</td>
<td>CPACR, Architectural Feature Access Control Register on page G8-9391</td>
</tr>
<tr>
<td>CPPRCTX</td>
<td>CPPRCTX, Cache Prefetch Prediction Restriction by Context on page G8-9401</td>
</tr>
<tr>
<td>CPSR</td>
<td>CPSR, Current Program Status Register on page G8-9396</td>
</tr>
<tr>
<td>CSSELR</td>
<td>CSSELR, Cache Size Selection Register on page G8-9404</td>
</tr>
<tr>
<td>CTR</td>
<td>CTR, Cache Type Register on page G8-9407</td>
</tr>
<tr>
<td>DACR</td>
<td>DACR, Domain Access Control Register on page G8-9410</td>
</tr>
<tr>
<td>DBGAUTHSTATUS</td>
<td>DBGAUTHSTATUS, Debug Authentication Status register on page G8-9832</td>
</tr>
<tr>
<td>DBGBCR&lt;n&gt;</td>
<td>DBGBCR&lt;n&gt;, Debug Breakpoint Control Registers, n = 0 - 15 on page G8-9835</td>
</tr>
<tr>
<td>DBGBVRR&lt;n&gt;</td>
<td>DBGBVRR&lt;n&gt;, Debug Breakpoint Value Registers, n = 0 - 15 on page G8-9840</td>
</tr>
<tr>
<td>DBGBXVR&lt;n&gt;</td>
<td>DBGBXVR&lt;n&gt;, Debug Breakpoint Extended Value Registers, n = 0 - 15 on page G8-9844</td>
</tr>
<tr>
<td>DBGCLAIMCLR</td>
<td>DBGCLAIMCLR, Debug CLAIM Tag Clear register on page G8-9848</td>
</tr>
<tr>
<td>DBGCLAIMSET</td>
<td>DBGCLAIMSET, Debug CLAIM Tag Set register on page G8-9851</td>
</tr>
<tr>
<td>DBGDCCINT</td>
<td>DBGDCCINT, DCC Interrupt Enable Register on page G8-9854</td>
</tr>
<tr>
<td>DBGDEVID</td>
<td>DBGDEVID, Debug Device ID register 0 on page G8-9858</td>
</tr>
<tr>
<td>DBGDEVID1</td>
<td>DBGDEVID1, Debug Device ID register 1 on page G8-9861</td>
</tr>
<tr>
<td>DBGDEVID2</td>
<td>DBGDEVID2, Debug Device ID register 2 on page G8-9863</td>
</tr>
<tr>
<td>DBGDIDR</td>
<td>DBGDIDR, Debug ID Register on page G8-9865</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DBGDRAR</td>
<td>DBGDRAR, Debug ROM Address Register on page G8-9868</td>
</tr>
<tr>
<td>DBGDSAR</td>
<td>DBGDSAR, Debug Self Address Register on page G8-9872</td>
</tr>
<tr>
<td>DBGDSCRxext</td>
<td>DBGDSCRxext, Debug Status and Control Register, External View on page G8-9875</td>
</tr>
<tr>
<td>DBGDSCRint</td>
<td>DBGDSCRint, Debug Status and Control Register, Internal View on page G8-9882</td>
</tr>
<tr>
<td>DBGDTRRXext</td>
<td>DBGDTRRXext, Debug OS Lock Data Transfer Register, Receive, External View on page G8-9887</td>
</tr>
<tr>
<td>DBGDTRRXint</td>
<td>DBGDTRRXint, Debug Data Transfer Register, Receive on page G8-9891</td>
</tr>
<tr>
<td>DBGDTRTXext</td>
<td>DBGDTRTXext, Debug OS Lock Data Transfer Register, Transmit on page G8-9893</td>
</tr>
<tr>
<td>DBGDTRXint</td>
<td>DBGDTRXint, Debug Data Transfer Register, Transmit on page G8-9897</td>
</tr>
<tr>
<td>DBGOSDLR</td>
<td>DBGOSDLR, Debug OS Double Lock Register on page G8-9899</td>
</tr>
<tr>
<td>DBGOSECCR</td>
<td>DBGOSECCR, Debug OS Lock Exception Catch Control Register on page G8-9902</td>
</tr>
<tr>
<td>DBGOSLAR</td>
<td>DBGOSLAR, Debug OS Lock Access Register on page G8-9905</td>
</tr>
<tr>
<td>DBGOSLSR</td>
<td>DBGOSLSR, Debug OS Lock Status Register on page G8-9907</td>
</tr>
<tr>
<td>DBGPRCR</td>
<td>DBGPRCR, Debug Power Control Register on page G8-9909</td>
</tr>
<tr>
<td>DBGVCr</td>
<td>DBGVCr, Debug Vector Catch Register on page G8-9912</td>
</tr>
<tr>
<td>DBGWCr&lt;n&gt;</td>
<td>DBGWCr&lt;n&gt;, Debug Watchpoint Control Registers, n = 0 - 15 on page G8-9920</td>
</tr>
<tr>
<td>DBGWFAR</td>
<td>DBGWFAR, Debug Watchpoint Fault Address Register on page G8-9925</td>
</tr>
<tr>
<td>DBGWVR&lt;n&gt;</td>
<td>DBGWVR&lt;n&gt;, Debug Watchpoint Value Registers, n = 0 - 15 on page G8-9927</td>
</tr>
<tr>
<td>DCCIMVAC</td>
<td>DCCIMVAC, Data Cache line Clean and Invalidate by VA to PoC on page G8-9412</td>
</tr>
<tr>
<td>DCCISW</td>
<td>DCCISW, Data Cache line Clean and Invalidate by Set/Way on page G8-9414</td>
</tr>
<tr>
<td>DCCMVC</td>
<td>DCCMVC, Data Cache line Clean by VA to PoC on page G8-9416</td>
</tr>
<tr>
<td>DCCMVAU</td>
<td>DCCMVAU, Data Cache line Clean by VA to PoU on page G8-9417</td>
</tr>
<tr>
<td>DCCSW</td>
<td>DCCSW, Data Cache line Clean by Set/Way on page G8-9419</td>
</tr>
<tr>
<td>DCIMVAC</td>
<td>DCIMVAC, Data Cache line Invalidate by VA to PoC on page G8-9421</td>
</tr>
<tr>
<td>DCISW</td>
<td>DCISW, Data Cache line Invalidate by Set/Way on page G8-9423</td>
</tr>
<tr>
<td>DFAR</td>
<td>DFAR, Data Fault Address Register on page G8-9425</td>
</tr>
<tr>
<td>DFSR</td>
<td>DFSR, Data Fault Status Register on page G8-9427</td>
</tr>
<tr>
<td>DISR</td>
<td>DISR, Deferred Interrupt Status Register on page G8-10084</td>
</tr>
<tr>
<td>DLR</td>
<td>DLR, Debug Link Register on page G8-9930</td>
</tr>
<tr>
<td>DSPSR</td>
<td>DSPSR, Debug Saved Program Status Register on page G8-9931</td>
</tr>
<tr>
<td>DTLBIALL</td>
<td>DTLBIALL, Data TLB Invalidate All on page G8-9434</td>
</tr>
<tr>
<td>DTLBIASID</td>
<td>DTLBIASID, Data TLB Invalidate by ASID match on page G8-9436</td>
</tr>
<tr>
<td>DTLBIMVA</td>
<td>DTLBIMVA, Data TLB Invalidate by VA on page G8-9438</td>
</tr>
</tbody>
</table>
## Table K17-26 Alphabetical index of AArch32 Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>DVPRCTX</td>
<td>DVPRCTX, Data Value Prediction Restriction by Context on page G8-9440</td>
</tr>
<tr>
<td>ELR_hyp</td>
<td>ELR_hyp, Exception Link Register (Hyp mode) on page G8-9443</td>
</tr>
<tr>
<td>ERRIDR</td>
<td>ERRIDR, Error Record ID Register on page G8-10089</td>
</tr>
<tr>
<td>ERRSELR</td>
<td>ERRSELR, Error Record Select Register on page G8-10091</td>
</tr>
<tr>
<td>ERXADDR</td>
<td>ERXADDR, Selected Error Record Address Register on page G8-10094</td>
</tr>
<tr>
<td>ERXADDR2</td>
<td>ERXADDR2, Selected Error Record Address Register 2 on page G8-10097</td>
</tr>
<tr>
<td>ERXCTLR</td>
<td>ERXCTLR, Selected Error Record Control Register on page G8-10100</td>
</tr>
<tr>
<td>ERXCTLR2</td>
<td>ERXCTLR2, Selected Error Record Control Register 2 on page G8-10103</td>
</tr>
<tr>
<td>ERXFR</td>
<td>ERXFR, Selected Error Record Feature Register on page G8-10106</td>
</tr>
<tr>
<td>ERXFR2</td>
<td>ERXFR2, Selected Error Record Feature Register 2 on page G8-10108</td>
</tr>
<tr>
<td>ERXMISC0</td>
<td>ERXMISC0, Selected Error Record Miscellaneous Register 0 on page G8-10110</td>
</tr>
<tr>
<td>ERXMISC1</td>
<td>ERXMISC1, Selected Error Record Miscellaneous Register 1 on page G8-10113</td>
</tr>
<tr>
<td>ERXMISC2</td>
<td>ERXMISC2, Selected Error Record Miscellaneous Register 2 on page G8-10116</td>
</tr>
<tr>
<td>ERXMISC3</td>
<td>ERXMISC3, Selected Error Record Miscellaneous Register 3 on page G8-10119</td>
</tr>
<tr>
<td>ERXMISC4</td>
<td>ERXMISC4, Selected Error Record Miscellaneous Register 4 on page G8-10122</td>
</tr>
<tr>
<td>ERXMISC5</td>
<td>ERXMISC5, Selected Error Record Miscellaneous Register 5 on page G8-10125</td>
</tr>
<tr>
<td>ERXMISC6</td>
<td>ERXMISC6, Selected Error Record Miscellaneous Register 6 on page G8-10128</td>
</tr>
<tr>
<td>ERXMISC7</td>
<td>ERXMISC7, Selected Error Record Miscellaneous Register 7 on page G8-10131</td>
</tr>
<tr>
<td>ERXSTATUS</td>
<td>ERXSTATUS, Selected Error Record Primary Status Register on page G8-10134</td>
</tr>
<tr>
<td>FCSEIDR</td>
<td>FCSEIDR, FCSE Process ID register on page G8-9444</td>
</tr>
<tr>
<td>FPEXC</td>
<td>FPEXC, Floating-Point Exception Control register on page G8-9446</td>
</tr>
<tr>
<td>FPSCR</td>
<td>FPSCR, Floating-Point Status and Control Register on page G8-9452</td>
</tr>
<tr>
<td>FPSID</td>
<td>FPSID, Floating-Point System ID register on page G8-9461</td>
</tr>
<tr>
<td>HACR</td>
<td>HACR, Hyp Auxiliary Configuration Register on page G8-9465</td>
</tr>
<tr>
<td>HACTLR</td>
<td>HACTLR, Hyp Auxiliary Control Register on page G8-9467</td>
</tr>
<tr>
<td>HACTLR2</td>
<td>HACTLR2, Hyp Auxiliary Control Register 2 on page G8-9469</td>
</tr>
<tr>
<td>HADFSR</td>
<td>HADFSR, Hyp Auxiliary Data Fault Status Register on page G8-9471</td>
</tr>
<tr>
<td>HAIFSR</td>
<td>HAIFSR, Hyp Auxiliary Instruction Fault Status Register on page G8-9473</td>
</tr>
<tr>
<td>HAMAIR0</td>
<td>HAMAIR0, Hyp Auxiliary Memory Attribute Indirection Register 0 on page G8-9475</td>
</tr>
<tr>
<td>HAMAIR1</td>
<td>HAMAIR1, Hyp Auxiliary Memory Attribute Indirection Register 1 on page G8-9477</td>
</tr>
<tr>
<td>HCPTTR</td>
<td>HCPTTR, Hyp Architectural Feature Trap Register on page G8-9479</td>
</tr>
<tr>
<td>HCR</td>
<td>HCR, Hyp Configuration Register on page G8-9485</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>HCR2</td>
<td>HCR2, Hyp Configuration Register 2 on page G8-9497</td>
</tr>
<tr>
<td>HDCR</td>
<td>HDCR, Hyp Debug Control Register on page G8-9935</td>
</tr>
<tr>
<td>HDFAR</td>
<td>HDFAR, Hyp Data Fault Address Register on page G8-9503</td>
</tr>
<tr>
<td>HIFAR</td>
<td>HIFAR, Hyp Instruction Fault Address Register on page G8-9505</td>
</tr>
<tr>
<td>HMAIR0</td>
<td>HMAIR0, Hyp Memory Attribute Indirection Register 0 on page G8-9507</td>
</tr>
<tr>
<td>HMAIR1</td>
<td>HMAIR1, Hyp Memory Attribute Indirection Register 1 on page G8-9510</td>
</tr>
<tr>
<td>HPFAR</td>
<td>HPFAR, Hyp IPA Fault Address Register on page G8-9513</td>
</tr>
<tr>
<td>HRMR</td>
<td>HRMR, Hyp Reset Management Register on page G8-9515</td>
</tr>
<tr>
<td>HSCTLR</td>
<td>HSCTLR, Hyp System Control Register on page G8-9517</td>
</tr>
<tr>
<td>HSR</td>
<td>HSR, Hyp Syndrome Register on page G8-9523</td>
</tr>
<tr>
<td>HSTR</td>
<td>HSTR, Hyp System Trap Register on page G8-9545</td>
</tr>
<tr>
<td>HTCR</td>
<td>HTCR, Hyp Translation Control Register on page G8-9547</td>
</tr>
<tr>
<td>HTPIDR</td>
<td>HTPIDR, Hyp Software Thread ID Register on page G8-9551</td>
</tr>
<tr>
<td>HTRFCR</td>
<td>HTRFCR, Hyp Trace Filter Control Register on page G8-9944</td>
</tr>
<tr>
<td>HTTBR</td>
<td>HTTBR, Hyp Translation Table Base Register on page G8-9553</td>
</tr>
<tr>
<td>HVBAR</td>
<td>HVBAR, Hyp Vector Base Address Register on page G8-9556</td>
</tr>
<tr>
<td>ICIALLU</td>
<td>ICIALLU, Instruction Cache Invalidate All to PoU on page G8-9558</td>
</tr>
<tr>
<td>ICIALLUIS</td>
<td>ICIALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable on page G8-9560</td>
</tr>
<tr>
<td>ICIMVAU</td>
<td>ICIMVAU, Instruction Cache line Invalidate by VA to PoU on page G8-9561</td>
</tr>
<tr>
<td>ID_AFR0</td>
<td>ID_AFR0, Auxiliary Feature Register 0 on page G8-9563</td>
</tr>
<tr>
<td>ID_DFR0</td>
<td>ID_DFR0, Debug Feature Register 0 on page G8-9565</td>
</tr>
<tr>
<td>ID_DFR1</td>
<td>ID_DFR1, Debug Feature Register 1 on page G8-9569</td>
</tr>
<tr>
<td>ID_ISAR0</td>
<td>ID_ISAR0, Instruction Set Attribute Register 0 on page G8-9571</td>
</tr>
<tr>
<td>ID_ISAR1</td>
<td>ID_ISAR1, Instruction Set Attribute Register 1 on page G8-9574</td>
</tr>
<tr>
<td>ID_ISAR2</td>
<td>ID_ISAR2, Instruction Set Attribute Register 2 on page G8-9577</td>
</tr>
<tr>
<td>ID_ISAR3</td>
<td>ID_ISAR3, Instruction Set Attribute Register 3 on page G8-9580</td>
</tr>
<tr>
<td>ID_ISAR4</td>
<td>ID_ISAR4, Instruction Set Attribute Register 4 on page G8-9583</td>
</tr>
<tr>
<td>ID_ISAR5</td>
<td>ID_ISAR5, Instruction Set Attribute Register 5 on page G8-9586</td>
</tr>
<tr>
<td>ID_ISAR6</td>
<td>ID_ISAR6, Instruction Set Attribute Register 6 on page G8-9589</td>
</tr>
<tr>
<td>ID_MMFR0</td>
<td>ID_MMFR0, Memory Model Feature Register 0 on page G8-9592</td>
</tr>
<tr>
<td>ID_MMFR1</td>
<td>ID_MMFR1, Memory Model Feature Register 1 on page G8-9595</td>
</tr>
<tr>
<td>ID_MMFR2</td>
<td>ID_MMFR2, Memory Model Feature Register 2 on page G8-9599</td>
</tr>
</tbody>
</table>
## Table K17-26 Alphabetical index of AArch32 Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID_MMFR3</td>
<td>ID_MMFR3, Memory Model Feature Register 3 on page G8-9602</td>
</tr>
<tr>
<td>ID_MMFR4</td>
<td>ID_MMFR4, Memory Model Feature Register 4 on page G8-9605</td>
</tr>
<tr>
<td>ID_MMFR5</td>
<td>ID_MMFR5, Memory Model Feature Register 5 on page G8-9608</td>
</tr>
<tr>
<td>ID_PFR0</td>
<td>ID_PFR0, Processor Feature Register 0 on page G8-9610</td>
</tr>
<tr>
<td>ID_PFR1</td>
<td>ID_PFR1, Processor Feature Register 1 on page G8-9613</td>
</tr>
<tr>
<td>ID_PFR2</td>
<td>ID_PFR2, Processor Feature Register 2 on page G8-9617</td>
</tr>
<tr>
<td>IFAR</td>
<td>IFAR, Instruction Fault Address Register on page G8-9619</td>
</tr>
<tr>
<td>IFSR</td>
<td>IFSR, Instruction Fault Status Register on page G8-9621</td>
</tr>
<tr>
<td>ISR</td>
<td>ISR, Interrupt Status Register on page G8-9626</td>
</tr>
<tr>
<td>ITLBIALL</td>
<td>ITLBIALL, Instruction TLB Invalidate All on page G8-9628</td>
</tr>
<tr>
<td>ITLBIASID</td>
<td>ITLBIASID, Instruction TLB Invalidate by ASID match on page G8-9630</td>
</tr>
<tr>
<td>ITLBIMVA</td>
<td>ITLBIMVA, Instruction TLB Invalidate by VA on page G8-9632</td>
</tr>
<tr>
<td>JIDR</td>
<td>JIDR, Jazelle ID Register on page G8-9634</td>
</tr>
<tr>
<td>JMCR</td>
<td>JMCR, Jazelle Main Configuration Register on page G8-9635</td>
</tr>
<tr>
<td>JOSCR</td>
<td>JOSCR, Jazelle OS Control Register on page G8-9637</td>
</tr>
<tr>
<td>MAIR0</td>
<td>MAIR0, Memory Attribute Indirection Register 0 on page G8-9639</td>
</tr>
<tr>
<td>MAIR1</td>
<td>MAIR1, Memory Attribute Indirection Register 1 on page G8-9643</td>
</tr>
<tr>
<td>MIDR</td>
<td>MIDR, Main ID Register on page G8-9647</td>
</tr>
<tr>
<td>MPIDR</td>
<td>MPIDR, Multiprocessor Affinity Register on page G8-9650</td>
</tr>
<tr>
<td>MVBAR</td>
<td>MVBAR, Monitor Vector Base Address Register on page G8-9652</td>
</tr>
<tr>
<td>MVFR0</td>
<td>MVFR0, Media and VFP Feature Register 0 on page G8-9654</td>
</tr>
<tr>
<td>MVFR1</td>
<td>MVFR1, Media and VFP Feature Register 1 on page G8-9658</td>
</tr>
<tr>
<td>MVFR2</td>
<td>MVFR2, Media and VFP Feature Register 2 on page G8-9662</td>
</tr>
<tr>
<td>NMRR</td>
<td>NMRR, Normal Memory Remap Register on page G8-9664</td>
</tr>
<tr>
<td>NSACR</td>
<td>NSACR, Non-Secure Access Control Register on page G8-9668</td>
</tr>
<tr>
<td>PAR</td>
<td>PAR, Physical Address Register on page G8-9672</td>
</tr>
<tr>
<td>PMCCFILTR</td>
<td>PMCCFILTR, Performance Monitors Cycle Count Filter Register on page G8-9964</td>
</tr>
<tr>
<td>PMCCNTR</td>
<td>PMCCNTR, Performance Monitors Cycle Count Register on page G8-9968</td>
</tr>
<tr>
<td>PMCEID0</td>
<td>PMCEID0, Performance Monitors Common Event Identification register 0 on page G8-9974</td>
</tr>
<tr>
<td>PMCEID1</td>
<td>PMCEID1, Performance Monitors Common Event Identification register 1 on page G8-9977</td>
</tr>
</tbody>
</table>
### Table K17-26 Alphabetical index of AArch32 Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMCEID2</td>
<td>PMCEID2, Performance Monitors Common Event Identification register 2 on page G8-9980</td>
</tr>
<tr>
<td>PMCEID3</td>
<td>PMCEID3, Performance Monitors Common Event Identification register 3 on page G8-9983</td>
</tr>
<tr>
<td>PMCNTENCLR</td>
<td>PMCNTENCLR, Performance Monitors Count Enable Clear register on page G8-9986</td>
</tr>
<tr>
<td>PMCNTENSET</td>
<td>PMCNTENSET, Performance Monitors Count Enable Set register on page G8-9990</td>
</tr>
<tr>
<td>PMCR</td>
<td>PMCR, Performance Monitors Control Register on page G8-9994</td>
</tr>
<tr>
<td>PMEVCNTR&lt;n&gt;</td>
<td>PMEVCNTR&lt;n&gt;, Performance Monitors Event Count Registers, n = 0 - 30 on page G8-10002</td>
</tr>
<tr>
<td>PMEVTYPER&lt;n&gt;</td>
<td>PMEVTYPER&lt;n&gt;, Performance Monitors Event Type Registers, n = 0 - 30 on page G8-10006</td>
</tr>
<tr>
<td>PMINTENCLR</td>
<td>PMINTENCLR, Performance Monitors Interrupt Enable Clear register on page G8-10013</td>
</tr>
<tr>
<td>PMINTENSET</td>
<td>PMINTENSET, Performance Monitors Interrupt Enable Set register on page G8-10016</td>
</tr>
<tr>
<td>PMMIR</td>
<td>PMMIR, Performance Monitors Machine Identification Register on page G8-9947</td>
</tr>
<tr>
<td>PMOVSR</td>
<td>PMOVSR, Performance Monitors Overflow Flag Status Register on page G8-10019</td>
</tr>
<tr>
<td>PMOVSSSET</td>
<td>PMOVSSSET, Performance Monitors Overflow Flag Status Set register on page G8-10023</td>
</tr>
<tr>
<td>PMSELR</td>
<td>PMSELR, Performance Monitors Event Counter Selection Register on page G8-10027</td>
</tr>
<tr>
<td>PMSWINC</td>
<td>PMSWINC, Performance Monitors Software Increment register on page G8-10031</td>
</tr>
<tr>
<td>PMUSERENR</td>
<td>PMUSERENR, Performance Monitors User Enable Register on page G8-10034</td>
</tr>
<tr>
<td>PMXEVCNTR</td>
<td>PMXEVCNTR, Performance Monitors Selected Event Count Register on page G8-10037</td>
</tr>
</tbody>
</table>
| PMXEVTYPE<
> | PMXEVTYPE<n>, Performance Monitors Selected Event Type Register on page G8-10042 |
| PRRR     | PRRR, Primary Region Remap Register on page G8-9683 |
| REVIDR   | REVIDR, Revision ID Register on page G8-9687 |
| RMR      | RMR, Reset Management Register on page G8-9688 |
| RVBAR    | RVBAR, Reset Vector Base Address Register on page G8-9690 |
| SCR      | SCR, Secure Configuration Register on page G8-9692 |
| SCTLR    | SCTLR, System Control Register on page G8-9697 |
| SDCR     | SDCR, Secure Debug Control Register on page G8-9950 |
| SDER     | SDER, Secure Debug Enable Register on page G8-9956 |
| SPSR     | SPSR, Saved Program Status Register on page G8-9706 |
| SPSR_abt | SPSR_abt, Saved Program Status Register (Abort mode) on page G8-9710 |
| SPSR_fiq | SPSR_fiq, Saved Program Status Register (FIQ mode) on page G8-9714 |
| SPSR_hyp | SPSR_hyp, Saved Program Status Register (Hyp mode) on page G8-9718 |
| SPSR_irq | SPSR_irq, Saved Program Status Register (IRQ mode) on page G8-9722 |
### Table K17-26 Alphabetical index of AArch32 Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPSR_mon</td>
<td><em>SPSR_mon, Saved Program Status Register (Monitor mode)</em> on page G8-9726</td>
</tr>
<tr>
<td>SPSR_svc</td>
<td><em>SPSR_svc, Saved Program Status Register (Supervisor mode)</em> on page G8-9730</td>
</tr>
<tr>
<td>SPSR_und</td>
<td><em>SPSR_und, Saved Program Status Register (Undefined mode)</em> on page G8-9734</td>
</tr>
<tr>
<td>TCMTR</td>
<td><em>TCMTR, TCM Type Register</em> on page G8-9738</td>
</tr>
<tr>
<td>TLBIALL</td>
<td><em>TLBIALL, TLB Invalidate All</em> on page G8-9739</td>
</tr>
<tr>
<td>TLBIALLH</td>
<td><em>TLBIALLH, TLB Invalidate All, Hyp mode</em> on page G8-9741</td>
</tr>
<tr>
<td>TLBIALLHIS</td>
<td><em>TLBIALLHIS, TLB Invalidate All, Hyp mode, Inner Shareable</em> on page G8-9742</td>
</tr>
<tr>
<td>TLBIALLIS</td>
<td><em>TLBIALLIS, TLB Invalidate All, Inner Shareable</em> on page G8-9743</td>
</tr>
<tr>
<td>TLBIALLNSNH</td>
<td><em>TLBIALLNSNH, TLB Invalidate All, Non-Secure Non-Hyp</em> on page G8-9745</td>
</tr>
<tr>
<td>TLBIALLNSNHIS</td>
<td><em>TLBIALLNSNHIS, TLB Invalidate All, Non-Secure Non-Hyp, Inner Shareable</em> on page G8-9746</td>
</tr>
<tr>
<td>TLBIASID</td>
<td><em>TLBIASID, TLB Invalidate by ASID match</em> on page G8-9747</td>
</tr>
<tr>
<td>TLBIASIDIS</td>
<td><em>TLBIASIDIS, TLB Invalidate by ASID match, Inner Shareable</em> on page G8-9749</td>
</tr>
<tr>
<td>TLBIIPAS2</td>
<td><em>TLBIIPAS2, TLB Invalidate by Intermediate Physical Address, Stage 2</em> on page G8-9751</td>
</tr>
<tr>
<td>TLBIIPAS2IS</td>
<td><em>TLBIIPAS2IS, TLB Invalidate by Intermediate Physical Address, Stage 2, Inner Shareable</em> on page G8-9753</td>
</tr>
<tr>
<td>TLBIIPAS2L</td>
<td><em>TLBIIPAS2L, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level</em> on page G8-9755</td>
</tr>
<tr>
<td>TLBIIPAS2LIS</td>
<td><em>TLBIIPAS2LIS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, Inner Shareable</em> on page G8-9757</td>
</tr>
<tr>
<td>TLBIMVA</td>
<td><em>TLBIMVA, TLB Invalidate by VA</em> on page G8-9759</td>
</tr>
<tr>
<td>TLBIMVAA</td>
<td><em>TLBIMVAA, TLB Invalidate by VA, All ASID</em> on page G8-9761</td>
</tr>
<tr>
<td>TLBIMVAAIS</td>
<td><em>TLBIMVAAIS, TLB Invalidate by VA, All ASID, Inner Shareable</em> on page G8-9763</td>
</tr>
<tr>
<td>TLBIMVAAL</td>
<td><em>TLBIMVAAL, TLB Invalidate by VA, All ASID, Last level</em> on page G8-9765</td>
</tr>
<tr>
<td>TLBIMVAALIS</td>
<td><em>TLBIMVAALIS, TLB Invalidate by VA, All ASID, Last level, Inner Shareable</em> on page G8-9767</td>
</tr>
<tr>
<td>TLBIMVAH</td>
<td><em>TLBIMVAH, TLB Invalidate by VA, Hyp mode</em> on page G8-9769</td>
</tr>
<tr>
<td>TLBIMVAHIS</td>
<td><em>TLBIMVAHIS, TLB Invalidate by VA, Hyp mode, Inner Shareable</em> on page G8-9771</td>
</tr>
<tr>
<td>TLBIMVAIS</td>
<td><em>TLBIMVAIS, TLB Invalidate by VA, Inner Shareable</em> on page G8-9773</td>
</tr>
<tr>
<td>TLBIMVAL</td>
<td><em>TLBIMVAL, TLB Invalidate by VA, Last level</em> on page G8-9775</td>
</tr>
<tr>
<td>TLBIMVALH</td>
<td><em>TLBIMVALH, TLB Invalidate by VA, Last level, Hyp mode</em> on page G8-9777</td>
</tr>
<tr>
<td>TLBIMVALHIS</td>
<td><em>TLBIMVALHIS, TLB Invalidate by VA, Last level, Hyp mode, Inner Shareable</em> on page G8-9779</td>
</tr>
<tr>
<td>TLBIMVALIS</td>
<td><em>TLBIMVALIS, TLB Invalidate by VA, Last level, Inner Shareable</em> on page G8-9781</td>
</tr>
<tr>
<td>TLBTR</td>
<td><em>TLBTR, TLB Type Register</em> on page G8-9783</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TPIDRPRW</td>
<td>TPIDRPRW, PL1 Software Thread ID Register on page G8-9785</td>
</tr>
<tr>
<td>TPIDRUDO</td>
<td>TPIDRUDO, PL0 Read-Only Software Thread ID Register on page G8-9787</td>
</tr>
<tr>
<td>TPIDRURW</td>
<td>TPIDRURW, PL0 Read/Write Software Thread ID Register on page G8-9789</td>
</tr>
<tr>
<td>TRFCR</td>
<td>TRFCR, Trace Filter Control Register on page G8-9959</td>
</tr>
<tr>
<td>TTBCR</td>
<td>TTBCR, Translation Table Base Control Register on page G8-9791</td>
</tr>
<tr>
<td>TTBCR2</td>
<td>TTBCR2, Translation Table Base Control Register 2 on page G8-9797</td>
</tr>
<tr>
<td>TTBR0</td>
<td>TTBR0, Translation Table Base Register 0 on page G8-9803</td>
</tr>
<tr>
<td>TTBR1</td>
<td>TTBR1, Translation Table Base Register 1 on page G8-9809</td>
</tr>
<tr>
<td>VBAR</td>
<td>VBAR, Vector Base Address Register on page G8-9815</td>
</tr>
<tr>
<td>VDFSRR</td>
<td>VDFSRR, Virtual SError Exception Syndrome Register on page G8-10137</td>
</tr>
<tr>
<td>VDISR</td>
<td>VDISR, Virtual Deferred Interrupt Status Register on page G8-10139</td>
</tr>
<tr>
<td>VMPIDR</td>
<td>VMPIDR, Virtualization Multiprocessor ID Register on page G8-9817</td>
</tr>
<tr>
<td>VPIDR</td>
<td>VPIDR, Virtualization Processor ID Register on page G8-9820</td>
</tr>
<tr>
<td>VTCR</td>
<td>VTCR, Virtualization Translation Control Register on page G8-9824</td>
</tr>
<tr>
<td>VTTBR</td>
<td>VTTBR, Virtualization Translation Table Base Register on page G8-9828</td>
</tr>
</tbody>
</table>
K17.5  Functional index of AArch32 registers and System instructions

This section is an index of the AArch32 registers and System instructions, divided by functional group. Each of the following sections lists the registers for a functional group:

- Special-purpose registers.
- VMSA-specific registers on page K17-11900.
- ID registers on page K17-11900.
- Performance monitors registers on page K17-11901.
- Activity Monitors registers on page K17-11902.
- Debug registers on page K17-11903.
- RAS registers on page K17-11904.
- Generic timer registers on page K17-11905.
- Cache maintenance system instructions on page K17-11905.
- Address translation system instructions on page K17-11906.
- TLB maintenance system instructions on page K17-11907.
- Legacy feature registers and system instructions on page K17-11908.
- Base system registers on page K17-11908.

K17.5.1  Special-purpose registers

This section is an index to the registers in the Processor state registers functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLR</td>
<td>DLR</td>
</tr>
<tr>
<td>DSPSR</td>
<td>DSPSR</td>
</tr>
<tr>
<td>ELR_hyp</td>
<td>ELR_hyp</td>
</tr>
<tr>
<td>SPSR</td>
<td>SPSR</td>
</tr>
<tr>
<td>SPSR_abt</td>
<td>SPSR_abt</td>
</tr>
<tr>
<td>SPSR_fiq</td>
<td>SPSR_fiq</td>
</tr>
<tr>
<td>SPSR_hyp</td>
<td>SPSR_hyp</td>
</tr>
<tr>
<td>SPSR_irq</td>
<td>SPSR_irq</td>
</tr>
<tr>
<td>SPSR_mon</td>
<td>SPSR_mon</td>
</tr>
<tr>
<td>SPSR_svc</td>
<td>SPSR_svc</td>
</tr>
<tr>
<td>SPSR_und</td>
<td>SPSR_und</td>
</tr>
</tbody>
</table>
K17.5.2 VMSA-specific registers

This section is an index to the registers in the Virtual memory control registers functional group.

Table K17-28 VMSA-specific registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMAIR0</td>
<td>AMAIR0</td>
</tr>
<tr>
<td>AMAIR1</td>
<td>AMAIR1</td>
</tr>
<tr>
<td>CONTEXTIDR</td>
<td>CONTEXTIDR</td>
</tr>
<tr>
<td>DACR</td>
<td>DACR</td>
</tr>
<tr>
<td>HAMAIR0</td>
<td>HAMAIR0</td>
</tr>
<tr>
<td>HAMAIR1</td>
<td>HAMAIR1</td>
</tr>
<tr>
<td>HMAIR0</td>
<td>HMAIR0</td>
</tr>
<tr>
<td>HMAIR1</td>
<td>HMAIR1</td>
</tr>
<tr>
<td>HTCR</td>
<td>HTCR</td>
</tr>
<tr>
<td>HTTBR</td>
<td>HTTBR</td>
</tr>
<tr>
<td>MAIR0</td>
<td>MAIR0</td>
</tr>
<tr>
<td>MAIR1</td>
<td>MAIR1</td>
</tr>
<tr>
<td>NMRR</td>
<td>NMRR</td>
</tr>
<tr>
<td>PRRR</td>
<td>PRRR</td>
</tr>
<tr>
<td>TTBCR</td>
<td>TTBCR</td>
</tr>
<tr>
<td>TTBCR2</td>
<td>TTBCR2</td>
</tr>
<tr>
<td>TTBR0</td>
<td>TTBR0</td>
</tr>
<tr>
<td>TTBR1</td>
<td>TTBR1</td>
</tr>
<tr>
<td>VTCR</td>
<td>VTCR</td>
</tr>
<tr>
<td>VTTBR</td>
<td>VTTBR</td>
</tr>
</tbody>
</table>

K17.5.3 ID registers

This section is an index to the registers in the Identification registers functional group.

Table K17-29 ID registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCSIDR</td>
<td>CCSIDR</td>
</tr>
<tr>
<td>CCSIDR2</td>
<td>CCSIDR2</td>
</tr>
<tr>
<td>CLIDR</td>
<td>CLIDR</td>
</tr>
<tr>
<td>CSSELR</td>
<td>CSSELR</td>
</tr>
<tr>
<td>CTR</td>
<td>CTR</td>
</tr>
</tbody>
</table>
### K17.5.4 Performance monitors registers

This section is an index to the registers in the Performance Monitors registers functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMCCFILTR</td>
<td>PMCCFILTR</td>
</tr>
<tr>
<td>PMCCNTR</td>
<td>PMCCNTR</td>
</tr>
<tr>
<td>PMCEID0</td>
<td>PMCEID0</td>
</tr>
</tbody>
</table>

---

**Table K17-30 Performance monitors registers**
K17.5 Functional index of AArch32 registers and System instructions

Table K17-30 Performance monitors registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMCEID1</td>
<td>PMCEID1</td>
</tr>
<tr>
<td>PMCEID2</td>
<td>PMCEID2</td>
</tr>
<tr>
<td>PMCEID3</td>
<td>PMCEID3</td>
</tr>
<tr>
<td>PMCNTENCLR</td>
<td>PMCNTENCLR</td>
</tr>
<tr>
<td>PMCNTENSET</td>
<td>PMCNTENSET</td>
</tr>
<tr>
<td>PMCR</td>
<td>PMCR</td>
</tr>
<tr>
<td>PMEVCNTR&lt;α&gt;</td>
<td>PMEVCNTR&lt;α&gt;</td>
</tr>
<tr>
<td>PMEVTYPER&lt;α&gt;</td>
<td>PMEVTYPER&lt;α&gt;</td>
</tr>
<tr>
<td>PMINTENCLR</td>
<td>PMINTENCLR</td>
</tr>
<tr>
<td>PMINTENSET</td>
<td>PMINTENSET</td>
</tr>
<tr>
<td>PMMIR</td>
<td>PMMIR</td>
</tr>
<tr>
<td>PMOVSR</td>
<td>PMOVSR</td>
</tr>
<tr>
<td>PMOVSET</td>
<td>PMOVSET</td>
</tr>
<tr>
<td>PMSELR</td>
<td>PMSELR</td>
</tr>
<tr>
<td>PMSWINC</td>
<td>PMSWINC</td>
</tr>
<tr>
<td>PMUSERENR</td>
<td>PMUSERENR</td>
</tr>
<tr>
<td>PMXEVCNTR</td>
<td>PMXEVCNTR</td>
</tr>
<tr>
<td>PMXEVTYPEPER</td>
<td>PMXEVTYPEPER</td>
</tr>
</tbody>
</table>

K17.5.5 Activity Monitors registers

This section is an index to the registers in the Activity Monitors registers functional group.

Table K17-31 Activity monitors registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMCFGR</td>
<td>AMCFGR</td>
</tr>
<tr>
<td>AMCGCR</td>
<td>AMCGCR</td>
</tr>
<tr>
<td>AMCNTENCLR0</td>
<td>AMCNTENCLR0</td>
</tr>
<tr>
<td>AMCNTENCLR1</td>
<td>AMCNTENCLR1</td>
</tr>
<tr>
<td>AMCNTENSET0</td>
<td>AMCNTENSET0</td>
</tr>
<tr>
<td>AMCNTENSET1</td>
<td>AMCNTENSET1</td>
</tr>
<tr>
<td>AMCR</td>
<td>AMCR</td>
</tr>
<tr>
<td>AMEVCNTR0&lt;α&gt;</td>
<td>AMEVCNTR0&lt;α&gt;</td>
</tr>
<tr>
<td>AMEVCNTR1&lt;α&gt;</td>
<td>AMEVCNTR1&lt;α&gt;</td>
</tr>
</tbody>
</table>
### K17.5.6 Debug registers

This section is an index to the registers in the Debug registers functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBGAUTHSTATUS</td>
<td>DBGAUTHSTATUS</td>
</tr>
<tr>
<td>DBGBCR&lt;n&gt;</td>
<td>DBGBCR&lt;n&gt;</td>
</tr>
<tr>
<td>DBGBV&lt;n&gt;</td>
<td>DBGBV&lt;n&gt;</td>
</tr>
<tr>
<td>DBGXVR&lt;n&gt;</td>
<td>DBGXVR&lt;n&gt;</td>
</tr>
<tr>
<td>DBGCLAIMCLR</td>
<td>DBGCLAIMCLR</td>
</tr>
<tr>
<td>DBGCLAIMSET</td>
<td>DBGCLAIMSET</td>
</tr>
<tr>
<td>DBGDCINT</td>
<td>DBGDCINT</td>
</tr>
<tr>
<td>DBGDEVID</td>
<td>DBGDEVID</td>
</tr>
<tr>
<td>DBGDEV1D</td>
<td>DBGDEV1D</td>
</tr>
<tr>
<td>DBGDEV2D</td>
<td>DBGDEV2D</td>
</tr>
<tr>
<td>DBGDIDR</td>
<td>DBGDIDR</td>
</tr>
<tr>
<td>DBGDRAR</td>
<td>DBGDRAR</td>
</tr>
<tr>
<td>DBGDSAR</td>
<td>DBGDSAR</td>
</tr>
<tr>
<td>DBGDSRext</td>
<td>DBGDSRext</td>
</tr>
<tr>
<td>DBGDSRint</td>
<td>DBGDSRint</td>
</tr>
<tr>
<td>DBGDRRXext</td>
<td>DBGDRRXext</td>
</tr>
<tr>
<td>DBGDRRXint</td>
<td>DBGDRRXint</td>
</tr>
<tr>
<td>DBGDRTXext</td>
<td>DBGDRTXext</td>
</tr>
<tr>
<td>DBGDRTXint</td>
<td>DBGDRTXint</td>
</tr>
<tr>
<td>DBGOSDLR</td>
<td>DBGOSDLR</td>
</tr>
<tr>
<td>DBGOSECCR</td>
<td>DBGOSECCR</td>
</tr>
<tr>
<td>DBGOSLAR</td>
<td>DBGOSLAR</td>
</tr>
<tr>
<td>DBGOSLSR</td>
<td>DBGOSLSR</td>
</tr>
<tr>
<td>DBGPRCR</td>
<td>DBGPRCR</td>
</tr>
</tbody>
</table>
K17.5.7 RAS registers

This section is an index to the registers in the RAS registers functional group.

Table K17-33 RAS registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISR</td>
<td>DISR</td>
</tr>
<tr>
<td>ERRIDR</td>
<td>ERRIDR</td>
</tr>
<tr>
<td>ERRSELR</td>
<td>ERRSELR</td>
</tr>
<tr>
<td>ERXADDR</td>
<td>ERXADDR</td>
</tr>
<tr>
<td>ERXADDR2</td>
<td>ERXADDR2</td>
</tr>
<tr>
<td>ERXCTLR</td>
<td>ERXCTLR</td>
</tr>
<tr>
<td>ERXCTLR2</td>
<td>ERXCTLR2</td>
</tr>
<tr>
<td>ERXFR</td>
<td>ERXFR</td>
</tr>
<tr>
<td>ERXFR2</td>
<td>ERXFR2</td>
</tr>
<tr>
<td>ERXMISC0</td>
<td>ERXMISC0</td>
</tr>
<tr>
<td>ERXMISC1</td>
<td>ERXMISC1</td>
</tr>
<tr>
<td>ERXMISC2</td>
<td>ERXMISC2</td>
</tr>
<tr>
<td>ERXMISC3</td>
<td>ERXMISC3</td>
</tr>
<tr>
<td>ERXMISC4</td>
<td>ERXMISC4</td>
</tr>
<tr>
<td>ERXMISC5</td>
<td>ERXMISC5</td>
</tr>
<tr>
<td>ERXMISC6</td>
<td>ERXMISC6</td>
</tr>
<tr>
<td>ERXMISC7</td>
<td>ERXMISC7</td>
</tr>
<tr>
<td>ERXSTATUS</td>
<td>ERXSTATUS</td>
</tr>
<tr>
<td>VDFSR</td>
<td>VDFSR</td>
</tr>
<tr>
<td>VDISR</td>
<td>VDISR</td>
</tr>
</tbody>
</table>
K17.5.8  Generic timer registers

This section is an index to the registers in the Generic Timer registers functional group.

Table K17-34 Generic timer registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNTFRQ</td>
<td>CNTFRQ</td>
</tr>
<tr>
<td>CNTHP_CTL</td>
<td>CNTHP_CTL</td>
</tr>
<tr>
<td>CNTHPS_CTL</td>
<td>CNTHPS_CTL</td>
</tr>
<tr>
<td>CNTHPS_CVAL</td>
<td>CNTHPS_CVAL</td>
</tr>
<tr>
<td>CNTHPS_TVAL</td>
<td>CNTHPS_TVAL</td>
</tr>
<tr>
<td>CNTHV_CTL</td>
<td>CNTHV_CTL</td>
</tr>
<tr>
<td>CNTHV_CVAL</td>
<td>CNTHV_CVAL</td>
</tr>
<tr>
<td>CNTHV_TVAL</td>
<td>CNTHV_TVAL</td>
</tr>
<tr>
<td>CNTHVS_CTL</td>
<td>CNTHVS_CTL</td>
</tr>
<tr>
<td>CNTHVS_CVAL</td>
<td>CNTHVS_CVAL</td>
</tr>
<tr>
<td>CNTHVS_TVAL</td>
<td>CNTHVS_TVAL</td>
</tr>
<tr>
<td>CNTKCTL</td>
<td>CNTKCTL</td>
</tr>
<tr>
<td>CNTP_CTL</td>
<td>CNTP_CTL</td>
</tr>
<tr>
<td>CNTP_CVAL</td>
<td>CNTP_CVAL</td>
</tr>
<tr>
<td>CNTP_TVAL</td>
<td>CNTP_TVAL</td>
</tr>
<tr>
<td>CNTPCT</td>
<td>CNTPCT</td>
</tr>
<tr>
<td>CNTPCTSS</td>
<td>CNTPCTSS</td>
</tr>
<tr>
<td>CNTV_CTL</td>
<td>CNTV_CTL</td>
</tr>
<tr>
<td>CNTV_CVAL</td>
<td>CNTV_CVAL</td>
</tr>
<tr>
<td>CNTV_TVAL</td>
<td>CNTV_TVAL</td>
</tr>
<tr>
<td>CNTVCT</td>
<td>CNTVCT</td>
</tr>
<tr>
<td>CNTVCTSS</td>
<td>CNTVCTSS</td>
</tr>
</tbody>
</table>

K17.5.9  Cache maintenance system instructions

This section is an index to the registers in the Cache maintenance instructions functional group.

Table K17-35 Cache maintenance system instructions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPIALL</td>
<td>BPIALL</td>
</tr>
<tr>
<td>BPIALLIS</td>
<td>BPIALLIS</td>
</tr>
<tr>
<td>BPIMVA</td>
<td>BPIMVA</td>
</tr>
</tbody>
</table>
### K17.5.10 Address translation system instructions

This section is an index to the registers in the Address translation instructions functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCCIMVAC</td>
<td>DCCIMVAC</td>
</tr>
<tr>
<td>DCCISW</td>
<td>DCCISW</td>
</tr>
<tr>
<td>DCCMVAC</td>
<td>DCCMVAC</td>
</tr>
<tr>
<td>DCCMVAU</td>
<td>DCCMVAU</td>
</tr>
<tr>
<td>DCCSW</td>
<td>DCCSW</td>
</tr>
<tr>
<td>DCIMVAC</td>
<td>DCIMVAC</td>
</tr>
<tr>
<td>DCISW</td>
<td>DCISW</td>
</tr>
<tr>
<td>ICIALLU</td>
<td>ICIALLU</td>
</tr>
<tr>
<td>ICIALLUIS</td>
<td>ICIALLUIS</td>
</tr>
<tr>
<td>ICIMVAU</td>
<td>ICIMVAU</td>
</tr>
</tbody>
</table>

Table K17-35 Cache maintenance system instructions (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATS12NSOPR</td>
<td>ATS12NSOPR</td>
</tr>
<tr>
<td>ATS12NSOPW</td>
<td>ATS12NSOPW</td>
</tr>
<tr>
<td>ATS12NSOUR</td>
<td>ATS12NSOUR</td>
</tr>
<tr>
<td>ATS12NSOUW</td>
<td>ATS12NSOUW</td>
</tr>
<tr>
<td>ATS1CPR</td>
<td>ATS1CPR</td>
</tr>
<tr>
<td>ATS1CPRP</td>
<td>ATS1CPRP</td>
</tr>
<tr>
<td>ATS1CPW</td>
<td>ATS1CPW</td>
</tr>
<tr>
<td>ATS1CPWP</td>
<td>ATS1CPWP</td>
</tr>
<tr>
<td>ATS1CUR</td>
<td>ATS1CUR</td>
</tr>
<tr>
<td>ATS1CUW</td>
<td>ATS1CUW</td>
</tr>
<tr>
<td>ATS1HR</td>
<td>ATS1HR</td>
</tr>
<tr>
<td>ATS1HW</td>
<td>ATS1HW</td>
</tr>
</tbody>
</table>
### K17.5.11 TLB maintenance system instructions

This section is an index to the registers in the TLB maintenance instructions functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFPRCTX</td>
<td>CFPRCTX</td>
</tr>
<tr>
<td>CPPRCTX</td>
<td>CPPRCTX</td>
</tr>
<tr>
<td>DTLBIALL</td>
<td>DTLBIALL</td>
</tr>
<tr>
<td>DTLBIASID</td>
<td>DTLBIASID</td>
</tr>
<tr>
<td>DTLBIMVA</td>
<td>DTLBIMVA</td>
</tr>
<tr>
<td>DVPRCTX</td>
<td>DVPRCTX</td>
</tr>
<tr>
<td>ITLBIALL</td>
<td>ITLBIALL</td>
</tr>
<tr>
<td>ITLBIASID</td>
<td>ITLBIASID</td>
</tr>
<tr>
<td>ITLBIMVA</td>
<td>ITLBIMVA</td>
</tr>
<tr>
<td>TLBIALL</td>
<td>TLBIALL</td>
</tr>
<tr>
<td>TLBIALLH</td>
<td>TLBIALLH</td>
</tr>
<tr>
<td>TLBIALLHIS</td>
<td>TLBIALLHIS</td>
</tr>
<tr>
<td>TLBIALLIS</td>
<td>TLBIALLIS</td>
</tr>
<tr>
<td>TLBIALLNSNH</td>
<td>TLBIALLNSNH</td>
</tr>
<tr>
<td>TLBIALLNSNHIS</td>
<td>TLBIALLNSNHIS</td>
</tr>
<tr>
<td>TLBIASID</td>
<td>TLBIASID</td>
</tr>
<tr>
<td>TLBIASIDIS</td>
<td>TLBIASIDIS</td>
</tr>
<tr>
<td>TLBIIPAS2</td>
<td>TLBIIPAS2</td>
</tr>
<tr>
<td>TLBIIPAS2IS</td>
<td>TLBIIPAS2IS</td>
</tr>
<tr>
<td>TLBIIPAS2L</td>
<td>TLBIIPAS2L</td>
</tr>
<tr>
<td>TLBIIPAS2LIS</td>
<td>TLBIIPAS2LIS</td>
</tr>
<tr>
<td>TLBIMVA</td>
<td>TLBIMVA</td>
</tr>
<tr>
<td>TLBIMVAA</td>
<td>TLBIMVAA</td>
</tr>
<tr>
<td>TLBIMVAAIS</td>
<td>TLBIMVAAIS</td>
</tr>
<tr>
<td>TLBIMVAAL</td>
<td>TLBIMVAAL</td>
</tr>
<tr>
<td>TLBIMVAALIS</td>
<td>TLBIMVAALIS</td>
</tr>
<tr>
<td>TLBIMVAH</td>
<td>TLBIMVAH</td>
</tr>
<tr>
<td>TLBIMVAHIS</td>
<td>TLBIMVAHIS</td>
</tr>
<tr>
<td>TLBIMVAIS</td>
<td>TLBIMVAIS</td>
</tr>
<tr>
<td>TLBIMVAL</td>
<td>TLBIMVAL</td>
</tr>
</tbody>
</table>
K17.5.12 Prediction restriction instructions

This section is an index to the registers in the Prediction restriction instructions functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLBIMVALH</td>
<td>TLBIMVALH</td>
</tr>
<tr>
<td>TLBIMVALHIS</td>
<td>TLBIMVALHIS</td>
</tr>
<tr>
<td>TLBIMVALIS</td>
<td>TLBIMVALIS</td>
</tr>
</tbody>
</table>

K17.5.13 Legacy feature registers and system instructions

This section is an index to the registers in the Legacy feature registers functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP15DMB</td>
<td>CP15DMB</td>
</tr>
<tr>
<td>CP15DSB</td>
<td>CP15DSB</td>
</tr>
<tr>
<td>CP15ISB</td>
<td>CP15ISB</td>
</tr>
<tr>
<td>FCSEIDR</td>
<td>FCSEIDR</td>
</tr>
<tr>
<td>JIDR</td>
<td>JIDR</td>
</tr>
<tr>
<td>JMCR</td>
<td>JMCR</td>
</tr>
<tr>
<td>JOSCR</td>
<td>JOSCR</td>
</tr>
</tbody>
</table>

K17.5.14 Base system registers

This section is an index to the registers in the Base system registers functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTLR</td>
<td>ACTLR</td>
</tr>
<tr>
<td>ACTLR2</td>
<td>ACTLR2</td>
</tr>
<tr>
<td>ADFSR</td>
<td>ADFSR</td>
</tr>
<tr>
<td>AIDR</td>
<td>AIDR</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>AIFSR</td>
<td>AIFSR</td>
</tr>
<tr>
<td>APSR</td>
<td>APSR</td>
</tr>
<tr>
<td>CNTHCTL</td>
<td>CNTHCTL</td>
</tr>
<tr>
<td>CNTHP_CVAL</td>
<td>CNTHP_CVAL</td>
</tr>
<tr>
<td>CNTHP_TVAL</td>
<td>CNTHP_TVAL</td>
</tr>
<tr>
<td>CNTVOFF</td>
<td>CNTVOFF</td>
</tr>
<tr>
<td>CPACR</td>
<td>CPACR</td>
</tr>
<tr>
<td>CPSR</td>
<td>CPSR</td>
</tr>
<tr>
<td>DFAR</td>
<td>DFAR</td>
</tr>
<tr>
<td>DFSR</td>
<td>DFSR</td>
</tr>
<tr>
<td>FPEXC</td>
<td>FPEXC</td>
</tr>
<tr>
<td>FPSCR</td>
<td>FPSCR</td>
</tr>
<tr>
<td>FPSID</td>
<td>FPSID</td>
</tr>
<tr>
<td>HACR</td>
<td>HACR</td>
</tr>
<tr>
<td>HACTLR</td>
<td>HACTLR</td>
</tr>
<tr>
<td>HACTLR2</td>
<td>HACTLR2</td>
</tr>
<tr>
<td>HADFSR</td>
<td>HADFSR</td>
</tr>
<tr>
<td>HAIFSR</td>
<td>HAIFSR</td>
</tr>
<tr>
<td>HCPTR</td>
<td>HCPTR</td>
</tr>
<tr>
<td>HCR</td>
<td>HCR</td>
</tr>
<tr>
<td>HCR2</td>
<td>HCR2</td>
</tr>
<tr>
<td>HDCR</td>
<td>HDCR</td>
</tr>
<tr>
<td>HDFAR</td>
<td>HDFAR</td>
</tr>
<tr>
<td>HIFAR</td>
<td>HIFAR</td>
</tr>
<tr>
<td>HPFAR</td>
<td>HPFAR</td>
</tr>
<tr>
<td>HRMR</td>
<td>HRMR</td>
</tr>
<tr>
<td>HSCTLR</td>
<td>HSCTLR</td>
</tr>
<tr>
<td>HSR</td>
<td>HSR</td>
</tr>
<tr>
<td>HSTR</td>
<td>HSTR</td>
</tr>
<tr>
<td>HTPIDR</td>
<td>HTPIDR</td>
</tr>
<tr>
<td>HTRFCR</td>
<td>HTRFCR</td>
</tr>
<tr>
<td>HVBAR</td>
<td>HVBAR</td>
</tr>
</tbody>
</table>
Table K17-40 Base system registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFAR</td>
<td>IFAR</td>
</tr>
<tr>
<td>IFSR</td>
<td>IFSR</td>
</tr>
<tr>
<td>ISR</td>
<td>ISR</td>
</tr>
<tr>
<td>MVBAR</td>
<td>MVBAR</td>
</tr>
<tr>
<td>MVFR0</td>
<td>MVFR0</td>
</tr>
<tr>
<td>MVFR1</td>
<td>MVFR1</td>
</tr>
<tr>
<td>MVFR2</td>
<td>MVFR2</td>
</tr>
<tr>
<td>NSACR</td>
<td>NSACR</td>
</tr>
<tr>
<td>PAR</td>
<td>PAR</td>
</tr>
<tr>
<td>RMR</td>
<td>RMR</td>
</tr>
<tr>
<td>RVBAR</td>
<td>RVBAR</td>
</tr>
<tr>
<td>SCR</td>
<td>SCR</td>
</tr>
<tr>
<td>SCTLR</td>
<td>SCTLR</td>
</tr>
<tr>
<td>SDCR</td>
<td>SDCR</td>
</tr>
<tr>
<td>SDER</td>
<td>SDER</td>
</tr>
<tr>
<td>TPIDRPRW</td>
<td>TPIDRPRW</td>
</tr>
<tr>
<td>TPIDRUC</td>
<td>TPIDRUC</td>
</tr>
<tr>
<td>TPIDURW</td>
<td>TPIDURW</td>
</tr>
<tr>
<td>VBAR</td>
<td>VBAR</td>
</tr>
<tr>
<td>VMPIDR</td>
<td>VMPIDR</td>
</tr>
<tr>
<td>VPIDR</td>
<td>VPIDR</td>
</tr>
</tbody>
</table>
# K17.6 Alphabetical index of memory-mapped registers

This section is an index of memory-mapped registers in alphabetical order.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMCFGR</td>
<td>AMCFGR, Activity Monitors Configuration Register on page I5-10839</td>
</tr>
<tr>
<td>AMCGCR</td>
<td>AMCGCR, Activity Monitors Counter Group Configuration Register on page I5-10841</td>
</tr>
<tr>
<td>AMCIDR0</td>
<td>AMCIDR0, Activity Monitors Component Identification Register 0 on page I5-10842</td>
</tr>
<tr>
<td>AMCIDR1</td>
<td>AMCIDR1, Activity Monitors Component Identification Register 1 on page I5-10843</td>
</tr>
<tr>
<td>AMCIDR2</td>
<td>AMCIDR2, Activity Monitors Component Identification Register 2 on page I5-10844</td>
</tr>
<tr>
<td>AMCIDR3</td>
<td>AMCIDR3, Activity Monitors Component Identification Register 3 on page I5-10845</td>
</tr>
<tr>
<td>AMCNTENCLR0</td>
<td>AMCNTENCLR0, Activity Monitors Count Enable Clear Register 0 on page I5-10846</td>
</tr>
<tr>
<td>AMCNTENCLR1</td>
<td>AMCNTENCLR1, Activity Monitors Count Enable Clear Register 1 on page I5-10848</td>
</tr>
<tr>
<td>AMCNTENSET0</td>
<td>AMCNTENSET0, Activity Monitors Count Enable Set Register 0 on page I5-10850</td>
</tr>
<tr>
<td>AMCNTENSET1</td>
<td>AMCNTENSET1, Activity Monitors Count Enable Set Register 1 on page I5-10852</td>
</tr>
<tr>
<td>AMCR</td>
<td>AMCR, Activity Monitors Control Register on page I5-10854</td>
</tr>
<tr>
<td>AMDEVAFF0</td>
<td>AMDEVAFF0, Activity Monitors Device Affinity Register 0 on page I5-10855</td>
</tr>
<tr>
<td>AMDEVAFF1</td>
<td>AMDEVAFF1, Activity Monitors Device Affinity Register 1 on page I5-10856</td>
</tr>
<tr>
<td>AMDEVARCH</td>
<td>AMDEVARCH, Activity Monitors Device Architecture Register on page I5-10857</td>
</tr>
<tr>
<td>AMDEVTYPE</td>
<td>AMDEVTYPE, Activity Monitors Device Type Register on page I5-10859</td>
</tr>
<tr>
<td>AMEVCNTR0&lt;n&gt;</td>
<td>AMEVCNTR0&lt;n&gt;, Activity Monitors Event Counter Registers 0, n = 0 - 3 on page I5-10860</td>
</tr>
<tr>
<td>AMEVCNTR1&lt;n&gt;</td>
<td>AMEVCNTR1&lt;n&gt;, Activity Monitors Event Counter Registers 1, n = 0 - 15 on page I5-10862</td>
</tr>
<tr>
<td>AMEVTYPER0&lt;n&gt;</td>
<td>AMEVTYPER0&lt;n&gt;, Activity Monitors Event Type Registers 0, n = 0 - 3 on page I5-10864</td>
</tr>
<tr>
<td>AMEVTYPER1&lt;n&gt;</td>
<td>AMEVTYPER1&lt;n&gt;, Activity Monitors Event Type Registers 1, n = 0 - 15 on page I5-10866</td>
</tr>
<tr>
<td>AMIIDR</td>
<td>AMIIDR, Activity Monitors Implementation Identification Register on page I5-10867</td>
</tr>
<tr>
<td>AMPIDR0</td>
<td>AMPIDR0, Activity Monitors Peripheral Identification Register 0 on page I5-10869</td>
</tr>
<tr>
<td>AMPIDR1</td>
<td>AMPIDR1, Activity Monitors Peripheral Identification Register 1 on page I5-10870</td>
</tr>
<tr>
<td>AMPIDR2</td>
<td>AMPIDR2, Activity Monitors Peripheral Identification Register 2 on page I5-10871</td>
</tr>
<tr>
<td>AMPIDR3</td>
<td>AMPIDR3, Activity Monitors Peripheral Identification Register 3 on page I5-10872</td>
</tr>
<tr>
<td>AMPIDR4</td>
<td>AMPIDR4, Activity Monitors Peripheral Identification Register 4 on page I5-10873</td>
</tr>
<tr>
<td>ASICCTL</td>
<td>ASICCTL, CTI External Multiplexer Control register on page H9-10661</td>
</tr>
<tr>
<td>CNTACR&lt;n&gt;</td>
<td>CNTACR&lt;n&gt;, Counter-timer Access Control Registers, n = 0 - 7 on page I5-10876</td>
</tr>
<tr>
<td>CNTCR</td>
<td>CNTCR, Counter Control Register on page I5-10878</td>
</tr>
<tr>
<td>CNTCV</td>
<td>CNTCV, Counter Count Value register on page I5-10880</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CNTEL0ACR</td>
<td>CNTEL0ACR, Counter-timer EL0 Access Control Register on page 15-10882</td>
</tr>
<tr>
<td>CNTFID0</td>
<td>CNTFID0, Counter Frequency ID on page 15-10884</td>
</tr>
<tr>
<td>CNTFID&lt;n&gt;</td>
<td>CNTFID&lt;n&gt;, Counter Frequency IDs, n &gt; 0, n = 1 - 1003 on page 15-10885</td>
</tr>
<tr>
<td>CNTFRQ</td>
<td>CNTFRQ, Counter-timer Frequency on page 15-10887</td>
</tr>
<tr>
<td>CNTID</td>
<td>CNTID, Counter Identification Register on page 15-10889</td>
</tr>
<tr>
<td>CNTNSAR</td>
<td>CNTNSAR, Counter-timer Non-secure Access Register on page 15-10890</td>
</tr>
<tr>
<td>CNTP_CTL</td>
<td>CNTP_CTL, Counter-timer Physical Timer Control on page 15-10892</td>
</tr>
<tr>
<td>CNTP_CVAL</td>
<td>CNTP_CVAL, Counter-timer Physical Timer CompareValue on page 15-10894</td>
</tr>
<tr>
<td>CNTP_TVAL</td>
<td>CNTP_TVAL, Counter-timer Physical Timer TimeValue on page 15-10896</td>
</tr>
<tr>
<td>CNTPCT</td>
<td>CNTPCT, Counter-timer Physical Count on page 15-10898</td>
</tr>
<tr>
<td>CNTSCR</td>
<td>CNTSCR, Counter Scale Register on page 15-10900</td>
</tr>
<tr>
<td>CNTSR</td>
<td>CNTSR, Counter Status Register on page 15-10901</td>
</tr>
<tr>
<td>CNTTIDR</td>
<td>CNTTIDR, Counter-timer Timer ID Register on page 15-10903</td>
</tr>
<tr>
<td>CNTV_CTL</td>
<td>CNTV_CTL, Counter-timer Virtual Timer Control on page 15-10905</td>
</tr>
<tr>
<td>CNTV_CVAL</td>
<td>CNTV_CVAL, Counter-timer Virtual Timer CompareValue on page 15-10907</td>
</tr>
<tr>
<td>CNTV_TVAL</td>
<td>CNTV_TVAL, Counter-timer Virtual Timer TimeValue on page 15-10909</td>
</tr>
<tr>
<td>CNTVCT</td>
<td>CNTVCT, Counter-timer Virtual Count on page 15-10911</td>
</tr>
<tr>
<td>CNTVOFF</td>
<td>CNTVOFF, Counter-timer Virtual Offset on page 15-10913</td>
</tr>
<tr>
<td>CNTVOFF&lt;n&gt;</td>
<td>CNTVOFF&lt;n&gt;, Counter-timer Virtual Offsets, n = 0 - 7 on page 15-10915</td>
</tr>
<tr>
<td>CounterID&lt;n&gt;</td>
<td>CounterID&lt;n&gt;, Counter ID registers, n = 0 - 11 on page 15-10917</td>
</tr>
<tr>
<td>CTIAPPCLEAR</td>
<td>CTIAPPCLEAR, CTI Application Trigger Clear register on page H9-10662</td>
</tr>
<tr>
<td>CTIAPPULSE</td>
<td>CTIAPPULSE, CTI Application Pulse register on page H9-10664</td>
</tr>
<tr>
<td>CTIAPPSET</td>
<td>CTIAPPSET, CTI Application Trigger Set register on page H9-10666</td>
</tr>
<tr>
<td>CTIAUTHSTATUS</td>
<td>CTIAUTHSTATUS, CTI Authentication Status register on page H9-10668</td>
</tr>
<tr>
<td>CTICHINSTATUS</td>
<td>CTICHINSTATUS, CTI Channel In Status register on page H9-10670</td>
</tr>
<tr>
<td>CTICHOUTSTATUS</td>
<td>CTICHOUTSTATUS, CTI Channel Out Status register on page H9-10671</td>
</tr>
<tr>
<td>CTICIDR0</td>
<td>CTICIDR0, CTI Component Identification Register 0 on page H9-10673</td>
</tr>
<tr>
<td>CTICIDR1</td>
<td>CTICIDR1, CTI Component Identification Register 1 on page H9-10674</td>
</tr>
<tr>
<td>CTICIDR2</td>
<td>CTICIDR2, CTI Component Identification Register 2 on page H9-10675</td>
</tr>
<tr>
<td>CTICIDR3</td>
<td>CTICIDR3, CTI Component Identification Register 3 on page H9-10676</td>
</tr>
<tr>
<td>CTICLAIMCLR</td>
<td>CTICLAIMCLR, CTI CLAIM Tag Clear register on page H9-10677</td>
</tr>
<tr>
<td>CTICLAIMSET</td>
<td>CTICLAIMSET, CTI CLAIM Tag Set register on page H9-10679</td>
</tr>
</tbody>
</table>
Table K17-41 Alphabetical index of Memory-Mapped Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTICONTROL</td>
<td>CTICONTROL, CTI Control register on page H9-10681</td>
</tr>
<tr>
<td>CTIDEVAFF0</td>
<td>CTIDEVAFF0, CTI Device Affinity register 0 on page H9-10682</td>
</tr>
<tr>
<td>CTIDEVAFF1</td>
<td>CTIDEVAFF1, CTI Device Affinity register 1 on page H9-10683</td>
</tr>
<tr>
<td>CTIDEVARCCH</td>
<td>CTIDEVARCCH, CTI Device Architecture register on page H9-10684</td>
</tr>
<tr>
<td>CTIDEVCTL</td>
<td>CTIDEVCTL, CTI Device Control register on page H9-10686</td>
</tr>
<tr>
<td>CTIDEVID</td>
<td>CTIDEVID, CTI Device ID register 0 on page H9-10687</td>
</tr>
<tr>
<td>CTIDEVID1</td>
<td>CTIDEVID1, CTI Device ID register 1 on page H9-10689</td>
</tr>
<tr>
<td>CTIDEVID2</td>
<td>CTIDEVID2, CTI Device ID register 2 on page H9-10690</td>
</tr>
<tr>
<td>CTIDEVTYP</td>
<td>CTIDEVTYP, CTI Device Type register on page H9-10691</td>
</tr>
<tr>
<td>CTIGATE</td>
<td>CTIGATE, CTI Channel Gate Enable register on page H9-10692</td>
</tr>
<tr>
<td>CTIINEN&lt;n&gt;</td>
<td>CTIINEN&lt;n&gt;, CTI Input Trigger to Output Channel Enable registers, n = 0 - 31 on page H9-10694</td>
</tr>
<tr>
<td>CTINTACK</td>
<td>CTINTACK, CTI Output Trigger Acknowledge register on page H9-10696</td>
</tr>
<tr>
<td>CTITCTRL</td>
<td>CTITCTRL, CTI Integration mode Control register on page H9-10698</td>
</tr>
<tr>
<td>CTILAR</td>
<td>CTILAR, CTI Lock Access Register on page H9-10700</td>
</tr>
<tr>
<td>CTILSR</td>
<td>CTILSR, CTI Lock Status Register on page H9-10701</td>
</tr>
<tr>
<td>CTIOUTEN&lt;n&gt;</td>
<td>CTIOUTEN&lt;n&gt;, CTI Input Channel to Output Trigger Enable registers, n = 0 - 31 on page H9-10703</td>
</tr>
<tr>
<td>CTPIDR0</td>
<td>CTPIDR0, CTI Peripheral Identification Register 0 on page H9-10705</td>
</tr>
<tr>
<td>CTPIDR1</td>
<td>CTPIDR1, CTI Peripheral Identification Register 1 on page H9-10706</td>
</tr>
<tr>
<td>CTPIDR2</td>
<td>CTPIDR2, CTI Peripheral Identification Register 2 on page H9-10707</td>
</tr>
<tr>
<td>CTPIDR3</td>
<td>CTPIDR3, CTI Peripheral Identification Register 3 on page H9-10708</td>
</tr>
<tr>
<td>CTPIDR4</td>
<td>CTPIDR4, CTI Peripheral Identification Register 4 on page H9-10709</td>
</tr>
<tr>
<td>CTITRIGINSTATUS</td>
<td>CTITRIGINSTATUS, CTI Trigger In Status register on page H9-10710</td>
</tr>
<tr>
<td>CTITRIGOUTSTATUS</td>
<td>CTITRIGOUTSTATUS, CTI Trigger Out Status register on page H9-10711</td>
</tr>
<tr>
<td>DBGAUTHSTATUS_EL1</td>
<td>DBGAUTHSTATUS_EL1, Debug Authentication Status register on page H9-10384</td>
</tr>
<tr>
<td>DBGBCR&lt;n&gt;_&lt;EL1&gt;</td>
<td>DBGBCR&lt;n&gt;_&lt;EL1&gt;, Debug Breakpoint Control Registers, n = 0 - 15 on page H9-10387</td>
</tr>
<tr>
<td>DBGBVRI&lt;n&gt;_&lt;EL1&gt;</td>
<td>DBGBVRI&lt;n&gt;_&lt;EL1&gt;, Debug Breakpoint Value Registers, n = 0 - 15 on page H9-10391</td>
</tr>
<tr>
<td>DBGCLAIMCLR_EL1</td>
<td>DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register on page H9-10396</td>
</tr>
<tr>
<td>DBGCLAIMSET_EL1</td>
<td>DBGCLAIMSET_EL1, Debug CLAIM Tag Set register on page H9-10398</td>
</tr>
<tr>
<td>DBGDTTRRX_EL0</td>
<td>DBGDTTRRX_EL0, Debug Data Transfer Register, Receive on page H9-10400</td>
</tr>
<tr>
<td>DBGDTRTX_EL0</td>
<td>DBGDTRTX_EL0, Debug Data Transfer Register, Transmit on page H9-10402</td>
</tr>
<tr>
<td>DBGWCR&lt;n&gt;_&lt;EL1&gt;</td>
<td>DBGWCR&lt;n&gt;_&lt;EL1&gt;, Debug Watchpoint Control Registers, n = 0 - 15 on page H9-10404</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DBGWVR&lt;(n)&gt; _EL1</td>
<td>DBGWVR&lt;(n)&gt; _EL1, Debug Watchpoint Value Registers, (n = 0 - 15) on page H9-10408</td>
</tr>
<tr>
<td>EDAA32PFR</td>
<td>EDAA32PFR, External Debug Auxiliary Processor Feature Register on page H9-10410</td>
</tr>
<tr>
<td>EDACR</td>
<td>EDACR, External Debug Auxiliary Control Register on page H9-10413</td>
</tr>
<tr>
<td>EDCIDR0</td>
<td>EDCIDR0, External Debug Component Identification Register 0 on page H9-10415</td>
</tr>
<tr>
<td>EDCIDR1</td>
<td>EDCIDR1, External Debug Component Identification Register 1 on page H9-10416</td>
</tr>
<tr>
<td>EDCIDR2</td>
<td>EDCIDR2, External Debug Component Identification Register 2 on page H9-10417</td>
</tr>
<tr>
<td>EDCIDR3</td>
<td>EDCIDR3, External Debug Component Identification Register 3 on page H9-10418</td>
</tr>
<tr>
<td>EDCIDSR</td>
<td>EDCIDSR, External Debug Context ID Sample Register on page H9-10419</td>
</tr>
<tr>
<td>EDDEVAFF0</td>
<td>EDDEVAFF0, External Debug Device Affinity register 0 on page H9-10421</td>
</tr>
<tr>
<td>EDDEVAFF1</td>
<td>EDDEVAFF1, External Debug Device Affinity register 1 on page H9-10422</td>
</tr>
<tr>
<td>EDDEVARCH</td>
<td>EDDEVARCH, External Debug Device Architecture register on page H9-10423</td>
</tr>
<tr>
<td>EDDEVID</td>
<td>EDDEVID, External Debug Device ID register 0 on page H9-10425</td>
</tr>
<tr>
<td>EDDEVID1</td>
<td>EDDEVID1, External Debug Device ID register 1 on page H9-10427</td>
</tr>
<tr>
<td>EDDEVID2</td>
<td>EDDEVID2, External Debug Device ID register 2 on page H9-10429</td>
</tr>
<tr>
<td>EDDEVTYPE</td>
<td>EDDEVTYPE, External Debug Device Type register on page H9-10430</td>
</tr>
<tr>
<td>EDDFR</td>
<td>EDDFR, External Debug Feature Register on page H9-10431</td>
</tr>
<tr>
<td>EDECCCR</td>
<td>EDECCCR, External Debug Exception Catch Control Register on page H9-10434</td>
</tr>
<tr>
<td>EDECRR</td>
<td>EDECRR, External Debug Execution Control Register on page H9-10442</td>
</tr>
<tr>
<td>EDESR</td>
<td>EDESR, External Debug Event Status Register on page H9-10444</td>
</tr>
<tr>
<td>EDHSR</td>
<td>EDHSR, External Debug Halt Status Register on page H9-10446</td>
</tr>
<tr>
<td>EDITCTRL</td>
<td>EDITCTRL, External Debug Integration mode Control register on page H9-10449</td>
</tr>
<tr>
<td>EDITR</td>
<td>EDITR, External Debug Instruction Transfer Register on page H9-10451</td>
</tr>
<tr>
<td>EDLAR</td>
<td>EDLAR, External Debug Lock Access Register on page H9-10453</td>
</tr>
<tr>
<td>EDLSR</td>
<td>EDLSR, External Debug Lock Status Register on page H9-10455</td>
</tr>
<tr>
<td>EDPCSR</td>
<td>EDPCSR, External Debug Program Counter Sample Register on page H9-10457</td>
</tr>
<tr>
<td>EDPFR</td>
<td>EDPFR, External Debug Processor Feature Register on page H9-10460</td>
</tr>
<tr>
<td>EDPIDR0</td>
<td>EDPIDR0, External Debug Peripheral Identification Register 0 on page H9-10464</td>
</tr>
<tr>
<td>EDPIDR1</td>
<td>EDPIDR1, External Debug Peripheral Identification Register 1 on page H9-10465</td>
</tr>
<tr>
<td>EDPIDR2</td>
<td>EDPIDR2, External Debug Peripheral Identification Register 2 on page H9-10466</td>
</tr>
<tr>
<td>EDPIDR3</td>
<td>EDPIDR3, External Debug Peripheral Identification Register 3 on page H9-10468</td>
</tr>
<tr>
<td>EDPIDR4</td>
<td>EDPIDR4, External Debug Peripheral Identification Register 4 on page H9-10469</td>
</tr>
<tr>
<td>EDPRCR</td>
<td>EDPRCR, External Debug Power/Reset Control Register on page H9-10470</td>
</tr>
</tbody>
</table>
## Table K17-41 Alphabetical index of Memory-Mapped Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDPRSR</td>
<td>EDPRSR, External Debug Processor Status Register on page H9-10475</td>
</tr>
<tr>
<td>EDRCR</td>
<td>EDRCR, External Debug Reserve Control Register on page H9-10487</td>
</tr>
<tr>
<td>EDSCR</td>
<td>EDSCR, External Debug Status and Control Register on page H9-10489</td>
</tr>
<tr>
<td>EDVIDSR</td>
<td>EDVIDSR, External Debug Virtual Context Sample Register on page H9-10497</td>
</tr>
<tr>
<td>EDWAR</td>
<td>EDWAR, External Debug Watchpoint Address Register on page H9-10500</td>
</tr>
<tr>
<td>ERRCIDR0</td>
<td>ERRCIDR0, Component Identification Register 0 on page I5-10920</td>
</tr>
<tr>
<td>ERRCIDR1</td>
<td>ERRCIDR1, Component Identification Register 1 on page I5-10921</td>
</tr>
<tr>
<td>ERRCIDR2</td>
<td>ERRCIDR2, Component Identification Register 2 on page I5-10922</td>
</tr>
<tr>
<td>ERRCIDR3</td>
<td>ERRCIDR3, Component Identification Register 3 on page I5-10923</td>
</tr>
<tr>
<td>ERRCRICR0</td>
<td>ERRCRICR0, Critical Error Interrupt Configuration Register 0 on page I5-10924</td>
</tr>
<tr>
<td>ERRCRICR1</td>
<td>ERRCRICR1, Critical Error Interrupt Configuration Register 1 on page I5-10926</td>
</tr>
<tr>
<td>ERRCRICR2</td>
<td>ERRCRICR2, Critical Error Interrupt Configuration Register 2 on page I5-10927</td>
</tr>
<tr>
<td>ERRDEVAFF</td>
<td>ERRDEVAFF, Device Affinity Register on page I5-10930</td>
</tr>
<tr>
<td>ERRDEVARC</td>
<td>ERRDEVARC, Device Architecture Register on page I5-10934</td>
</tr>
<tr>
<td>ERRDEVID</td>
<td>ERRDEVID, Device Configuration Register on page I5-10936</td>
</tr>
<tr>
<td>ERRERICR0</td>
<td>ERRERICR0, Error Recovery Interrupt Configuration Register 0 on page I5-10937</td>
</tr>
<tr>
<td>ERRERICR1</td>
<td>ERRERICR1, Error Recovery Interrupt Configuration Register 1 on page I5-10939</td>
</tr>
<tr>
<td>ERRERICR2</td>
<td>ERRERICR2, Error Recovery Interrupt Configuration Register 2 on page I5-10940</td>
</tr>
<tr>
<td>ERRFHICR0</td>
<td>ERRFHICR0, Fault Handling Interrupt Configuration Register 0 on page I5-10943</td>
</tr>
<tr>
<td>ERRFHICR1</td>
<td>ERRFHICR1, Fault Handling Interrupt Configuration Register 1 on page I5-10945</td>
</tr>
<tr>
<td>ERRFHICR2</td>
<td>ERRFHICR2, Fault Handling Interrupt Configuration Register 2 on page I5-10946</td>
</tr>
<tr>
<td>ERRGSR</td>
<td>ERRGSR, Error Group Status Register on page I5-10949</td>
</tr>
<tr>
<td>ERRIIDR</td>
<td>ERRIIDR, Implementation Identification Register on page I5-10950</td>
</tr>
<tr>
<td>ERRIMPD&lt;(n)&gt;</td>
<td>ERRIMPD&lt;(n)&gt;, IMPLEMENTATION DEFINED Register &lt;(n)&gt;, (n = 0 - 191) on page I5-10952</td>
</tr>
<tr>
<td>ERRIRQ&lt;(n)&gt;</td>
<td>ERRIRQ&lt;(n)&gt;, Generic Error Interrupt Configuration Register &lt;(n)&gt;, (n = 0 - 15) on page I5-10953</td>
</tr>
<tr>
<td>ERRIRQSR</td>
<td>ERRIRQSR, Error Interrupt Status Register on page I5-10954</td>
</tr>
<tr>
<td>ERR&lt;(n)&gt;ADDR</td>
<td>ERR&lt;(n)&gt;ADDR, Error Record &lt;(n)&gt; Address Register, (n = 0 - 65534) on page I5-10957</td>
</tr>
<tr>
<td>ERR&lt;(n)&gt;CTRLR</td>
<td>ERR&lt;(n)&gt;CTRLR, Error Record &lt;(n)&gt; Control Register, (n = 0 - 65534) on page I5-10960</td>
</tr>
<tr>
<td>ERR&lt;(n)&gt;FR</td>
<td>ERR&lt;(n)&gt;FR, Error Record &lt;(n)&gt; Feature Register, (n = 0 - 65534) on page I5-10967</td>
</tr>
<tr>
<td>ERR&lt;(n)&gt;MISC0</td>
<td>ERR&lt;(n)&gt;MISC0, Error Record &lt;(n)&gt; Miscellaneous Register 0, (n = 0 - 65534) on page I5-10973</td>
</tr>
</tbody>
</table>
### Table K17-41 Alphabetical index of Memory-Mapped Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR&lt;n&gt;MISC1</td>
<td>ERR&lt;n&gt;MISC1, Error Record &lt;n&gt; Miscellaneous Register 1, n = 0 - 65534 on page 15-10978</td>
</tr>
<tr>
<td>ERR&lt;n&gt;MISC2</td>
<td>ERR&lt;n&gt;MISC2, Error Record &lt;n&gt; Miscellaneous Register 2, n = 0 - 65534 on page 15-10980</td>
</tr>
<tr>
<td>ERR&lt;n&gt;MISC3</td>
<td>ERR&lt;n&gt;MISC3, Error Record &lt;n&gt; Miscellaneous Register 3, n = 0 - 65534 on page 15-10982</td>
</tr>
<tr>
<td>ERR&lt;n&gt;PFGCDN</td>
<td>ERR&lt;n&gt;PFGCDN, Error Record &lt;n&gt; Pseudo-fault Generation Countdown Register, n = 0 - 65534 on page 15-10984</td>
</tr>
<tr>
<td>ERR&lt;n&gt;PFGCTL</td>
<td>ERR&lt;n&gt;PFGCTL, Error Record &lt;n&gt; Pseudo-fault Generation Control Register, n = 0 - 65534 on page 15-10986</td>
</tr>
<tr>
<td>ERR&lt;n&gt;PFGF</td>
<td>ERR&lt;n&gt;PFGF, Error Record &lt;n&gt; Pseudo-fault Generation Feature Register, n = 0 - 65534 on page 15-10991</td>
</tr>
<tr>
<td>ERR&lt;n&gt;STATUS</td>
<td>ERR&lt;n&gt;STATUS, Error Record &lt;n&gt; Primary Status Register, n = 0 - 65534 on page 15-10996</td>
</tr>
<tr>
<td>ERRPIDR0</td>
<td>ERRPIDR0, Peripheral Identification Register 0 on page 15-11013</td>
</tr>
<tr>
<td>ERRPIDR1</td>
<td>ERRPIDR1, Peripheral Identification Register 1 on page 15-11014</td>
</tr>
<tr>
<td>ERRPIDR2</td>
<td>ERRPIDR2, Peripheral Identification Register 2 on page 15-11016</td>
</tr>
<tr>
<td>ERRPIDR3</td>
<td>ERRPIDR3, Peripheral Identification Register 3 on page 15-11018</td>
</tr>
<tr>
<td>ERRPIDR4</td>
<td>ERRPIDR4, Peripheral Identification Register 4 on page 15-11020</td>
</tr>
<tr>
<td>MIDR_EL1</td>
<td>MIDR_EL1, Main ID Register on page 19-10502</td>
</tr>
<tr>
<td>MPAMCFG_CASSOC</td>
<td>MPAMCFG_CASSOC</td>
</tr>
<tr>
<td>MPAMCFG_CMAX</td>
<td>MPAMCFG_CMAX</td>
</tr>
<tr>
<td>MPAMCFG_CMIN</td>
<td>MPAMCFG_CMIN</td>
</tr>
<tr>
<td>MPAMCFG_CPBM&lt;n&gt;</td>
<td>MPAMCFG_CPBM&lt;n&gt;</td>
</tr>
<tr>
<td>MPAMCFG_DIS</td>
<td>MPAMCFG_DIS</td>
</tr>
<tr>
<td>MPAMCFG_EN</td>
<td>MPAMCFG_EN</td>
</tr>
<tr>
<td>MPAMCFG_EN_FLAGS</td>
<td>MPAMCFG_EN_FLAGS</td>
</tr>
<tr>
<td>MPAMCFG_INTPARTID</td>
<td>MPAMCFG_INTPARTID</td>
</tr>
<tr>
<td>MPAMCFG_MBW_MAX</td>
<td>MPAMCFG_MBW_MAX</td>
</tr>
<tr>
<td>MPAMCFG_MBW_MIN</td>
<td>MPAMCFG_MBW_MIN</td>
</tr>
<tr>
<td>MPAMCFG_MBW_PBM&lt;n&gt;</td>
<td>MPAMCFG_MBW_PBM&lt;n&gt;</td>
</tr>
<tr>
<td>MPAMCFG_MBW_PROP</td>
<td>MPAMCFG_MBW_PROP</td>
</tr>
<tr>
<td>MPAMCFG_MBW_WINWD</td>
<td>MPAMCFG_MBW_WINWD</td>
</tr>
<tr>
<td>MPAMCFG_PART_SEL</td>
<td>MPAMCFG_PART_SEL</td>
</tr>
<tr>
<td>MPAMCFG_PRI</td>
<td>MPAMCFG_PRI</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>MPAMF_AIDR</td>
<td>MPAMF_AIDR</td>
</tr>
<tr>
<td>MPAMF_CCAP_IDR</td>
<td>MPAMF_CCAP_IDR</td>
</tr>
<tr>
<td>MPAMF_CPOR_IDR</td>
<td>MPAMF_CPOR_IDR</td>
</tr>
<tr>
<td>MPAMF_CSUMON_IDR</td>
<td>MPAMF_CSUMON_IDR</td>
</tr>
<tr>
<td>MPAMF_ECR</td>
<td>MPAMF_ECR</td>
</tr>
<tr>
<td>MPAMF_ERR_MSI_ADDR_H</td>
<td>MPAMF_ERR_MSI_ADDR_H</td>
</tr>
<tr>
<td>MPAMF_ERR_MSI_ADDR_L</td>
<td>MPAMF_ERR_MSI_ADDR_L</td>
</tr>
<tr>
<td>MPAMF_ERR_MSI_ATTR</td>
<td>MPAMF_ERR_MSI_ATTR</td>
</tr>
<tr>
<td>MPAMF_ERR_MSI_DATA</td>
<td>MPAMF_ERR_MSI_DATA</td>
</tr>
<tr>
<td>MPAMF_ERR_MSI_MPAM</td>
<td>MPAMF_ERR_MSI_MPAM</td>
</tr>
<tr>
<td>MPAMF_ESR</td>
<td>MPAMF_ESR</td>
</tr>
<tr>
<td>MPAMF_IDR</td>
<td>MPAMF_IDR</td>
</tr>
<tr>
<td>MPAMF_IMPL_IDR</td>
<td>MPAMF_IMPL_IDR</td>
</tr>
<tr>
<td>MPAMF_MBW_IDR</td>
<td>MPAMF_MBW_IDR</td>
</tr>
<tr>
<td>MPAMF_MBWUMON_IDR</td>
<td>MPAMF_MBWUMON_IDR</td>
</tr>
<tr>
<td>MPAMF_MSMON_IDR</td>
<td>MPAMF_MSMON_IDR</td>
</tr>
<tr>
<td>MPAMF_PARTID_NRW_IDR</td>
<td>MPAMF_PARTID_NRW_IDR</td>
</tr>
<tr>
<td>MPAMF_PRI_IDR</td>
<td>MPAMF_PRI_IDR</td>
</tr>
<tr>
<td>MPAMF_SIDR</td>
<td>MPAMF_SIDR</td>
</tr>
<tr>
<td>MSMON_CAPT_EVNT</td>
<td>MSMON_CAPT_EVNT</td>
</tr>
<tr>
<td>MSMON_CFG_CSU_CTL</td>
<td>MSMON_CFG_CSU_CTL</td>
</tr>
<tr>
<td>MSMON_CFG_CSU_FLT</td>
<td>MSMON_CFG_CSU_FLT</td>
</tr>
<tr>
<td>MSMON_CFG_MBWU_CTL</td>
<td>MSMON_CFG_MBWU_CTL</td>
</tr>
<tr>
<td>MSMON_CFG_MBWU_FLT</td>
<td>MSMON_CFG_MBWU_FLT</td>
</tr>
<tr>
<td>MSMON_CFG_MON_SEL</td>
<td>MSMON_CFG_MON_SEL</td>
</tr>
<tr>
<td>MSMON_CSU</td>
<td>MSMON_CSU</td>
</tr>
<tr>
<td>MSMON_CSU_CAPTURE</td>
<td>MSMON_CSU_CAPTURE</td>
</tr>
<tr>
<td>MSMON_CSU_OFSR</td>
<td>MSMON_CSU_OFSR</td>
</tr>
<tr>
<td>MSMON_MBWU</td>
<td>MSMON_MBWU</td>
</tr>
<tr>
<td>MSMON_MBWU_CAPTURE</td>
<td>MSMON_MBWU_CAPTURE</td>
</tr>
<tr>
<td>MSMON_MBWU_L</td>
<td>MSMON_MBWU_L</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>MSMON_MBWU_L_CAPTURE</td>
<td>MSMON_MBWU_L_CAPTURE</td>
</tr>
<tr>
<td>MSMON_MBWU_OFSR</td>
<td>MSMON_MBWU_OFSR</td>
</tr>
<tr>
<td>MSMON_OFLOW_MSI_ADDR_H</td>
<td>MSMON_OFLOW_MSI_ADDR_H</td>
</tr>
<tr>
<td>MSMON_OFLOW_MSI_ADDR_L</td>
<td>MSMON_OFLOW_MSI_ADDR_L</td>
</tr>
<tr>
<td>MSMON_OFLOW_MSI_ATTR</td>
<td>MSMON_OFLOW_MSI_ATTR</td>
</tr>
<tr>
<td>MSMON_OFLOW_MSI_DATA</td>
<td>MSMON_OFLOW_MSI_DATA</td>
</tr>
<tr>
<td>MSMON_OFLOW_MSI_MPAM</td>
<td>MSMON_OFLOW_MSI_MPAM</td>
</tr>
<tr>
<td>MSMON_OFLOW_SR</td>
<td>MSMON_OFLOW_SR</td>
</tr>
<tr>
<td>OSLAR_EL1</td>
<td>OSLAR_EL1, OS Lock Access Register on page H9-10504</td>
</tr>
<tr>
<td>PMAUTHSTATUS</td>
<td>PMAUTHSTATUS, Performance Monitors Authentication Status register on page 15-10754</td>
</tr>
<tr>
<td>PMCCFILTR_EL0</td>
<td>PMCCFILTR_EL0, Performance Monitors Cycle Counter Filter Register on page 15-10756</td>
</tr>
<tr>
<td>PMCCNTR_EL0</td>
<td>PMCCNTR_EL0, Performance Monitors Cycle Counter on page 15-10760</td>
</tr>
<tr>
<td>PMCEID0</td>
<td>PMCEID0, Performance Monitors Common Event Identification register 0 on page 15-10762</td>
</tr>
<tr>
<td>PMCEID1</td>
<td>PMCEID1, Performance Monitors Common Event Identification register 1 on page 15-10764</td>
</tr>
<tr>
<td>PMCEID2</td>
<td>PMCEID2, Performance Monitors Common Event Identification register 2 on page 15-10766</td>
</tr>
<tr>
<td>PMCEID3</td>
<td>PMCEID3, Performance Monitors Common Event Identification register 3 on page 15-10768</td>
</tr>
<tr>
<td>PMCFGR</td>
<td>PMCFGR, Performance Monitors Configuration Register on page 15-10770</td>
</tr>
<tr>
<td>PMCID1SR</td>
<td>PMCID1SR, CONTEXTIDR_EL1 Sample Register on page 15-10777</td>
</tr>
<tr>
<td>PMCID2SR</td>
<td>PMCID2SR, CONTEXTIDR_EL2 Sample Register on page 15-10779</td>
</tr>
<tr>
<td>PMCIDR0</td>
<td>PMCIDR0, Performance Monitors Component Identification Register 0 on page 15-10773</td>
</tr>
<tr>
<td>PMCIDR1</td>
<td>PMCIDR1, Performance Monitors Component Identification Register 1 on page 15-10774</td>
</tr>
<tr>
<td>PMCIDR2</td>
<td>PMCIDR2, Performance Monitors Component Identification Register 2 on page 15-10775</td>
</tr>
<tr>
<td>PMCIDR3</td>
<td>PMCIDR3, Performance Monitors Component Identification Register 3 on page 15-10776</td>
</tr>
<tr>
<td>PMCNTENCLR_EL0</td>
<td>PMCNTENCLR_EL0, Performance Monitors Count Enable Clear register on page 15-10781</td>
</tr>
<tr>
<td>PMCNTENSET_EL0</td>
<td>PMCNTENSET_EL0, Performance Monitors Count Enable Set register on page 15-10783</td>
</tr>
<tr>
<td>PMCR_EL0</td>
<td>PMCR_EL0, Performance Monitors Control Register on page 15-10785</td>
</tr>
<tr>
<td>PMDEVAFF0</td>
<td>PMDEVAFF0, Performance Monitors Device Affinity register 0 on page 15-10790</td>
</tr>
<tr>
<td>PMDEVAFF1</td>
<td>PMDEVAFF1, Performance Monitors Device Affinity register 1 on page 15-10791</td>
</tr>
</tbody>
</table>
### Table K17-41 Alphabetical index of Memory-Mapped Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMDEVARCH</td>
<td>PMDEVARCH, Performance Monitors Device Architecture register on page 15-10792</td>
</tr>
<tr>
<td>PMDEVID</td>
<td>PMDEVID, Performance Monitors Device ID register on page 15-10794</td>
</tr>
<tr>
<td>PMDEVTYPE</td>
<td>PMDEVTYPE, Performance Monitors Device Type register on page 15-10795</td>
</tr>
<tr>
<td>PMEVCNTR&lt;n&gt;_EL0</td>
<td>PMEVCNTR&lt;n&gt;_EL0, Performance Monitors Event Count Registers, n = 0 - 30 on page 15-10796</td>
</tr>
<tr>
<td>PMEVTYPER&lt;n&gt;_EL0</td>
<td>PMEVTYPER&lt;n&gt;_EL0, Performance Monitors Event Type Registers, n = 0 - 30 on page 15-10798</td>
</tr>
<tr>
<td>PMINTENCLR_EL1</td>
<td>PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear register on page 15-10805</td>
</tr>
<tr>
<td>PMINTENSET_EL1</td>
<td>PMINTENSET_EL1, Performance Monitors Interrupt Enable Set register on page 15-10807</td>
</tr>
<tr>
<td>PMITCTRL</td>
<td>PMITCTRL, Performance Monitors Integration mode Control register on page 15-10809</td>
</tr>
<tr>
<td>PMLAR</td>
<td>PMLAR, Performance Monitors Lock Access Register on page 15-10811</td>
</tr>
<tr>
<td>PMLSR</td>
<td>PMLSR, Performance Monitors Lock Status Register on page 15-10813</td>
</tr>
<tr>
<td>PMMIR</td>
<td>PMMIR, Performance Monitors Machine Identification Register on page 15-10815</td>
</tr>
<tr>
<td>PMOVSCLR_EL0</td>
<td>PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear register on page 15-10818</td>
</tr>
<tr>
<td>PMOVSSET_EL0</td>
<td>PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set register on page 15-10820</td>
</tr>
<tr>
<td>PMPCSR</td>
<td>PMPCSR, Program Counter Sample Register on page 15-10822</td>
</tr>
<tr>
<td>PMPIDR0</td>
<td>PMPIDR0, Performance Monitors Peripheral Identification Register 0 on page 15-10826</td>
</tr>
<tr>
<td>PMPIDR1</td>
<td>PMPIDR1, Performance Monitors Peripheral Identification Register 1 on page 15-10827</td>
</tr>
<tr>
<td>PMPIDR2</td>
<td>PMPIDR2, Performance Monitors Peripheral Identification Register 2 on page 15-10828</td>
</tr>
<tr>
<td>PMPIDR3</td>
<td>PMPIDR3, Performance Monitors Peripheral Identification Register 3 on page 15-10830</td>
</tr>
<tr>
<td>PMPIDR4</td>
<td>PMPIDR4, Performance Monitors Peripheral Identification Register 4 on page 15-10831</td>
</tr>
<tr>
<td>PMSWINC_EL0</td>
<td>PMSWINC_EL0, Performance Monitors Software Increment register on page 15-10832</td>
</tr>
<tr>
<td>PMVIDSR</td>
<td>PMVIDSR, VMID Sample Register on page 15-10834</td>
</tr>
<tr>
<td>TRCACATR&lt;n&gt;</td>
<td>TRCACATR&lt;n&gt;, Address Comparator Access Type Register &lt;n&gt;, n = 0 - 15 on page H9-10507</td>
</tr>
<tr>
<td>TRCACVR&lt;n&gt;</td>
<td>TRCACVR&lt;n&gt;, Address Comparator Value Register &lt;n&gt;, n = 0 - 15 on page H9-10513</td>
</tr>
<tr>
<td>TRCAUTHSTATUS</td>
<td>TRCAUTHSTATUS, Authentication Status Register on page H9-10515</td>
</tr>
<tr>
<td>TRCAUXCTLR</td>
<td>TRCAUXCTLR, Auxiliary Control Register on page H9-10518</td>
</tr>
<tr>
<td>TRCBBCTLR</td>
<td>TRCBBCTLR, Branch Broadcast Control Register on page H9-10519</td>
</tr>
<tr>
<td>TRCCCCTLR</td>
<td>TRCCCCTLR, Cycle Count Control Register on page H9-10521</td>
</tr>
<tr>
<td>TRCCIDCCTLR0</td>
<td>TRCCIDCCTLR0, Context Identifier Comparator Control Register 0 on page H9-10522</td>
</tr>
</tbody>
</table>
## Table K17-41 Alphabetical index of Memory-Mapped Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRCCIDCCTLR1</td>
<td>TRCCIDCCTLR1, Context Identifier Comparator Control Register 1 on page H9-10524</td>
</tr>
<tr>
<td>TRCCIDCVR&lt;n&gt;</td>
<td>TRCCIDCVR&lt;n&gt;, Context Identifier Comparator Value Registers &lt;n&gt;, n = 0 - 7 on page H9-10526</td>
</tr>
<tr>
<td>TRCCIDR0</td>
<td>TRCCIDR0, Component Identification Register 0 on page H9-10527</td>
</tr>
<tr>
<td>TRCCIDR1</td>
<td>TRCCIDR1, Component Identification Register 1 on page H9-10528</td>
</tr>
<tr>
<td>TRCCIDR2</td>
<td>TRCCIDR2, Component Identification Register 2 on page H9-10529</td>
</tr>
<tr>
<td>TRCCIDR3</td>
<td>TRCCIDR3, Component Identification Register 3 on page H9-10530</td>
</tr>
<tr>
<td>TRCCLAIMCLR</td>
<td>TRCCLAIMCLR, Claim Tag Clear Register on page H9-10531</td>
</tr>
<tr>
<td>TRCCLAIMSET</td>
<td>TRCCLAIMSET, Claim Tag Set Register on page H9-10533</td>
</tr>
<tr>
<td>TRCCNTCTLR&lt;n&gt;</td>
<td>TRCCNTCTLR&lt;n&gt;, Counter Control Register &lt;n&gt;, n = 0 - 3 on page H9-10535</td>
</tr>
<tr>
<td>TRCCNTRLDVR&lt;n&gt;</td>
<td>TRCCNTRLDVR&lt;n&gt;, Counter Reload Value Register &lt;n&gt;, n = 0 - 3 on page H9-10538</td>
</tr>
<tr>
<td>TRCCNTVR&lt;n&gt;</td>
<td>TRCCNTVR&lt;n&gt;, Counter Value Register &lt;n&gt;, n = 0 - 3 on page H9-10539</td>
</tr>
<tr>
<td>TRCCONFIGR</td>
<td>TRCCONFIGR, Trace Configuration Register on page H9-10540</td>
</tr>
<tr>
<td>TRCDEVAFF</td>
<td>TRCDEVAFF, Device Affinity Register on page H9-10544</td>
</tr>
<tr>
<td>TRCDEVARCH</td>
<td>TRCDEVARCH, Device Architecture Register on page H9-10545</td>
</tr>
<tr>
<td>TRCDEVID</td>
<td>TRCDEVID, Device Configuration Register on page H9-10547</td>
</tr>
<tr>
<td>TRCDEVID1</td>
<td>TRCDEVID1, Device Configuration Register 1 on page H9-10548</td>
</tr>
<tr>
<td>TRCDEVID2</td>
<td>TRCDEVID2, Device Configuration Register 2 on page H9-10549</td>
</tr>
<tr>
<td>TRCDEVTYPE</td>
<td>TRCDEVTYPE, Device Type Register on page H9-10550</td>
</tr>
<tr>
<td>TRCEVENTCTL0R</td>
<td>TRCEVENTCTL0R, Event Control 0 Register on page H9-10551</td>
</tr>
<tr>
<td>TRCEVENTCTL1R</td>
<td>TRCEVENTCTL1R, Event Control 1 Register on page H9-10555</td>
</tr>
<tr>
<td>TRCEXTINSELR&lt;n&gt;</td>
<td>TRCEXTINSELR&lt;n&gt;, External Input Select Register &lt;n&gt;, n = 0 - 3 on page H9-10557</td>
</tr>
<tr>
<td>TRCIDR0</td>
<td>TRCIDR0, ID Register 0 on page H9-10559</td>
</tr>
<tr>
<td>TRCIDR1</td>
<td>TRCIDR1, ID Register 1 on page H9-10563</td>
</tr>
<tr>
<td>TRCIDR10</td>
<td>TRCIDR10, ID Register 10 on page H9-10565</td>
</tr>
<tr>
<td>TRCIDR11</td>
<td>TRCIDR11, ID Register 11 on page H9-10566</td>
</tr>
<tr>
<td>TRCIDR12</td>
<td>TRCIDR12, ID Register 12 on page H9-10567</td>
</tr>
<tr>
<td>TRCIDR13</td>
<td>TRCIDR13, ID Register 13 on page H9-10568</td>
</tr>
<tr>
<td>TRCIDR2</td>
<td>TRCIDR2, ID Register 2 on page H9-10569</td>
</tr>
<tr>
<td>TRCIDR3</td>
<td>TRCIDR3, ID Register 3 on page H9-10572</td>
</tr>
<tr>
<td>TRCIDR4</td>
<td>TRCIDR4, ID Register 4 on page H9-10575</td>
</tr>
<tr>
<td>TRCIDR5</td>
<td>TRCIDR5, ID Register 5 on page H9-10579</td>
</tr>
<tr>
<td>TRCIDR6</td>
<td>TRCIDR6, ID Register 6 on page H9-10581</td>
</tr>
</tbody>
</table>
### Table K17-41 Alphabetical index of Memory-Mapped Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRCIDR7</td>
<td>TRCIDR7, ID Register 7 on page H9-10583</td>
</tr>
<tr>
<td>TRCIDR8</td>
<td>TRCIDR8, ID Register 8 on page H9-10584</td>
</tr>
<tr>
<td>TRCIDR9</td>
<td>TRCIDR9, ID Register 9 on page H9-10585</td>
</tr>
<tr>
<td>TRCIMSPEC0</td>
<td>TRCIMSPEC0, IMP DEF Register 0 on page H9-10586</td>
</tr>
<tr>
<td>TRCIMSPEC&lt;n&gt;</td>
<td>TRCIMSPEC&lt;n&gt;, IMP DEF Register &lt;n&gt;, n = 1 - 7 on page H9-10588</td>
</tr>
<tr>
<td>TRCITCTRL</td>
<td>TRCITCTRL, Integration Mode Control Register on page H9-10589</td>
</tr>
<tr>
<td>TRCLAR</td>
<td>TRCLAR, Lock Access Register on page H9-10590</td>
</tr>
<tr>
<td>TRCLSR</td>
<td>TRCLSR, Lock Status Register on page H9-10591</td>
</tr>
<tr>
<td>TRCOSLSR</td>
<td>TRCOSLSR, Trace OS Lock Status Register on page H9-10593</td>
</tr>
<tr>
<td>TRCPDCR</td>
<td>TRCPDCR, PowerDown Control Register on page H9-10595</td>
</tr>
<tr>
<td>TRCPDSR</td>
<td>TRCPDSR, PowerDown Status Register on page H9-10596</td>
</tr>
<tr>
<td>TRCPIDR0</td>
<td>TRCPIDR0, Peripheral Identification Register 0 on page H9-10598</td>
</tr>
<tr>
<td>TRCPIDR1</td>
<td>TRCPIDR1, Peripheral Identification Register 1 on page H9-10599</td>
</tr>
<tr>
<td>TRCPIDR2</td>
<td>TRCPIDR2, Peripheral Identification Register 2 on page H9-10601</td>
</tr>
<tr>
<td>TRCPIDR3</td>
<td>TRCPIDR3, Peripheral Identification Register 3 on page H9-10603</td>
</tr>
<tr>
<td>TRCPIDR4</td>
<td>TRCPIDR4, Peripheral Identification Register 4 on page H9-10605</td>
</tr>
<tr>
<td>TRCPIDR5</td>
<td>TRCPIDR5, Peripheral Identification Register 5 on page H9-10607</td>
</tr>
<tr>
<td>TRCPIDR6</td>
<td>TRCPIDR6, Peripheral Identification Register 6 on page H9-10608</td>
</tr>
<tr>
<td>TRCPIDR7</td>
<td>TRCPIDR7, Peripheral Identification Register 7 on page H9-10609</td>
</tr>
<tr>
<td>TRCPRGCTLR</td>
<td>TRCPRGCTLR, Programming Control Register on page H9-10610</td>
</tr>
<tr>
<td>TRCQCTRL</td>
<td>TRCQCTRL, Q Element Control Register on page H9-10611</td>
</tr>
<tr>
<td>TRCRSCTRL&lt;n&gt;</td>
<td>TRCRSCTRL&lt;n&gt;, Resource Selection Control Register &lt;n&gt;, n = 2 - 31 on page H9-10613</td>
</tr>
<tr>
<td>TRCRSR</td>
<td>TRCRSR, Resources Status Register on page H9-10620</td>
</tr>
<tr>
<td>TRCSEQEVR&lt;n&gt;</td>
<td>TRCSEQEVR&lt;n&gt;, Sequencer State Transition Control Register &lt;n&gt;, n = 0 - 2 on page H9-10622</td>
</tr>
<tr>
<td>TRCSEQRSTEVR</td>
<td>TRCSEQRSTEVR, Sequencer Reset Control Register on page H9-10625</td>
</tr>
<tr>
<td>TRCSEQSTR</td>
<td>TRCSEQSTR, Sequencer State Register on page H9-10627</td>
</tr>
<tr>
<td>TRCSSCCR&lt;n&gt;</td>
<td>TRCSSCCR&lt;n&gt;, Single-shot Comparator Control Register &lt;n&gt;, n = 0 - 7 on page H9-10628</td>
</tr>
<tr>
<td>TRCSSCSR&lt;n&gt;</td>
<td>TRCSSCSR&lt;n&gt;, Single-shot Comparator Control Status Register &lt;n&gt;, n = 0 - 7 on page H9-10630</td>
</tr>
<tr>
<td>TRCSSPCICR&lt;n&gt;</td>
<td>TRCSSPCICR&lt;n&gt;, Single-shot Processing Element Comparator Input Control Register &lt;n&gt;, n = 0 - 7 on page H9-10633</td>
</tr>
<tr>
<td>TRCSTALLCTLR</td>
<td>TRCSTALLCTLR, Stall Control Register on page H9-10635</td>
</tr>
</tbody>
</table>
# Table K17-41 Alphabetical index of Memory-Mapped Registers (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRCSTATR</td>
<td>TRCSTATR, Trace Status Register on page H9-10637</td>
</tr>
<tr>
<td>TRCSYNCR</td>
<td>TRCSYNCR, Synchronization Period Register on page H9-10638</td>
</tr>
<tr>
<td>TRCTRACEIDR</td>
<td>TRCTRACEIDR, Trace ID Register on page H9-10640</td>
</tr>
<tr>
<td>TRCTSCTRL</td>
<td>TRCTSCTRL, Timestamp Control Register on page H9-10641</td>
</tr>
<tr>
<td>TRCVICTLR</td>
<td>TRCVICTLR, ViewInst Main Control Register on page H9-10643</td>
</tr>
<tr>
<td>TRCVIIECTLR</td>
<td>TRCVIIECTLR, ViewInst Include/Exclude Control Register on page H9-10649</td>
</tr>
<tr>
<td>TRCVIPCSSCTRL</td>
<td>TRCVIPCSSCTRL, ViewInst Start/Stop PE Comparator Control Register on page H9-10651</td>
</tr>
<tr>
<td>TRCVISSCTRL</td>
<td>TRCVISSCTRL, ViewInst Start/Stop Control Register on page H9-10653</td>
</tr>
<tr>
<td>TRCVMICCTR0</td>
<td>TRCVMICCTR0, Virtual Context Identifier Comparator Control Register 0 on page H9-10655</td>
</tr>
<tr>
<td>TRCVMICCTR1</td>
<td>TRCVMICCTR1, Virtual Context Identifier Comparator Control Register 1 on page H9-10657</td>
</tr>
<tr>
<td>TRCVMICCR&lt;n&gt;</td>
<td>TRCVMICCR&lt;n&gt;, Virtual Context Identifier Comparator Value Register &lt;n&gt;, n = 0 - 7 on page H9-10659</td>
</tr>
</tbody>
</table>
K17.7 Functional index of memory-mapped registers

This section is an index of the memory-mapped registers, divided by functional group. Each of the following sections lists the registers for a functional group:

- ID registers.
- Performance monitors registers.
- Debug registers on page K17-11925.
- RAS registers on page K17-11926.
- Cross-trigger interface registers on page K17-11927.

K17.7.1 ID registers

This section is an index to the registers in the Identification registers functional group.

### Table K17-42 ID registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDAA32PFR</td>
<td>EDAA32PFR</td>
</tr>
<tr>
<td>EDDFR</td>
<td>EDDFR</td>
</tr>
<tr>
<td>EDPFR</td>
<td>EDPFR</td>
</tr>
<tr>
<td>MIDR_EL1</td>
<td>MIDR_EL1</td>
</tr>
</tbody>
</table>

K17.7.2 Performance monitors registers

This section is an index to the registers in the Performance Monitors registers functional group.

### Table K17-43 Performance monitors registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMAUTHSTATUS</td>
<td>PMAUTHSTATUS</td>
</tr>
<tr>
<td>PMCCFILTR_EL0</td>
<td>PMCCFILTR_EL0</td>
</tr>
<tr>
<td>PMCCNTR_EL0</td>
<td>PMCCNTR_EL0</td>
</tr>
<tr>
<td>PMCEID0</td>
<td>PMCEID0</td>
</tr>
<tr>
<td>PMCEID1</td>
<td>PMCEID1</td>
</tr>
<tr>
<td>PMCEID2</td>
<td>PMCEID2</td>
</tr>
<tr>
<td>PMCEID3</td>
<td>PMCEID3</td>
</tr>
<tr>
<td>PMCFGR</td>
<td>PMCFGR</td>
</tr>
<tr>
<td>PMCID1SR</td>
<td>PMCID1SR</td>
</tr>
<tr>
<td>PMCID2SR</td>
<td>PMCID2SR</td>
</tr>
<tr>
<td>PMCIDR0</td>
<td>PMCIDR0</td>
</tr>
<tr>
<td>PMCIDR1</td>
<td>PMCIDR1</td>
</tr>
<tr>
<td>PMCIDR2</td>
<td>PMCIDR2</td>
</tr>
<tr>
<td>PMCIDR3</td>
<td>PMCIDR3</td>
</tr>
<tr>
<td>PMCNTENCLR_EL0</td>
<td>PMCNTENCLR_EL0</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>PMCNTENSET_EL0</td>
<td>PMCNTENSET_EL0</td>
</tr>
<tr>
<td>PMCR_EL0</td>
<td>PMCR_EL0</td>
</tr>
<tr>
<td>PMDEVAFF0</td>
<td>PMDEVAFF0</td>
</tr>
<tr>
<td>PMDEVAFF1</td>
<td>PMDEVAFF1</td>
</tr>
<tr>
<td>PMDEVARCH</td>
<td>PMDEVARCH</td>
</tr>
<tr>
<td>PMDEVID</td>
<td>PMDEVID</td>
</tr>
<tr>
<td>PMDEVTYPE</td>
<td>PMDEVTYPE</td>
</tr>
<tr>
<td>PMEVCNTR&lt;n&gt;_EL0</td>
<td>PMEVCNTR&lt;n&gt;_EL0</td>
</tr>
<tr>
<td>PMEVTYPER&lt;n&gt;_EL0</td>
<td>PMEVTYPER&lt;n&gt;_EL0</td>
</tr>
<tr>
<td>PMINTENCLR_EL1</td>
<td>PMINTENCLR_EL1</td>
</tr>
<tr>
<td>PMINTENSET_EL1</td>
<td>PMINTENSET_EL1</td>
</tr>
<tr>
<td>PMITCTRL</td>
<td>PMITCTRL</td>
</tr>
<tr>
<td>PMLAR</td>
<td>PMLAR</td>
</tr>
<tr>
<td>PMLSR</td>
<td>PMLSR</td>
</tr>
<tr>
<td>PMMIR</td>
<td>PMMIR</td>
</tr>
<tr>
<td>PMOVSCLR_EL0</td>
<td>PMOVSCLR_EL0</td>
</tr>
<tr>
<td>PMOVSSSET_EL0</td>
<td>PMOVSSSET_EL0</td>
</tr>
<tr>
<td>PMPCSР</td>
<td>PMPCSР</td>
</tr>
<tr>
<td>PMPIDR0</td>
<td>PMPIDR0</td>
</tr>
<tr>
<td>PMPIDR1</td>
<td>PMPIDR1</td>
</tr>
<tr>
<td>PMPIDR2</td>
<td>PMPIDR2</td>
</tr>
<tr>
<td>PMPIDR3</td>
<td>PMPIDR3</td>
</tr>
<tr>
<td>PMPIDR4</td>
<td>PMPIDR4</td>
</tr>
<tr>
<td>PMSWINC_EL0</td>
<td>PMSWINC_EL0</td>
</tr>
<tr>
<td>PMVIDSR</td>
<td>PMVIDSR</td>
</tr>
</tbody>
</table>
### K17.7.3 Debug registers

This section is an index to the registers in the Debug registers functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBGAUTHSTATUS_EL1</td>
<td>DBGAUTHSTATUS_EL1</td>
</tr>
<tr>
<td>DBGBCR&lt;n&gt;_EL1</td>
<td>DBGBCR&lt;n&gt;_EL1</td>
</tr>
<tr>
<td>DBGBVR&lt;n&gt;_EL1</td>
<td>DBGBVR&lt;n&gt;_EL1</td>
</tr>
<tr>
<td>DBGCLAIMCLR_EL1</td>
<td>DBGCLAIMCLR_EL1</td>
</tr>
<tr>
<td>DBGCLAIMSET_EL1</td>
<td>DBGCLAIMSET_EL1</td>
</tr>
<tr>
<td>DBGDTRRX_EL0</td>
<td>DBGDTRRX_EL0</td>
</tr>
<tr>
<td>DBGDTRTX_EL0</td>
<td>DBGDTRTX_EL0</td>
</tr>
<tr>
<td>DBGWCR&lt;n&gt;_EL1</td>
<td>DBGWCR&lt;n&gt;_EL1</td>
</tr>
<tr>
<td>DBGWVR&lt;n&gt;_EL1</td>
<td>DBGWVR&lt;n&gt;_EL1</td>
</tr>
<tr>
<td>EDACR</td>
<td>EDACR</td>
</tr>
<tr>
<td>EDCIDR0</td>
<td>EDCIDR0</td>
</tr>
<tr>
<td>EDCIDR1</td>
<td>EDCIDR1</td>
</tr>
<tr>
<td>EDCIDR2</td>
<td>EDCIDR2</td>
</tr>
<tr>
<td>EDCIDR3</td>
<td>EDCIDR3</td>
</tr>
<tr>
<td>EDCIDSR</td>
<td>EDCIDSR</td>
</tr>
<tr>
<td>EDEVAFF0</td>
<td>EDEVAFF0</td>
</tr>
<tr>
<td>EDEVAFF1</td>
<td>EDEVAFF1</td>
</tr>
<tr>
<td>EDDEVARCH</td>
<td>EDDEVARCH</td>
</tr>
<tr>
<td>EDDEVID</td>
<td>EDDEVID</td>
</tr>
<tr>
<td>EDDEVID1</td>
<td>EDDEVID1</td>
</tr>
<tr>
<td>EDDEVID2</td>
<td>EDDEVID2</td>
</tr>
<tr>
<td>EDDEVTYPE</td>
<td>EDDEVTYPE</td>
</tr>
<tr>
<td>EDECCR</td>
<td>EDECCR</td>
</tr>
<tr>
<td>EDECR</td>
<td>EDECR</td>
</tr>
<tr>
<td>EDESR</td>
<td>EDESR</td>
</tr>
<tr>
<td>EDHSR</td>
<td>EDHSR</td>
</tr>
<tr>
<td>EDITCTRL</td>
<td>EDITCTRL</td>
</tr>
<tr>
<td>EDITR</td>
<td>EDITR</td>
</tr>
<tr>
<td>EDLAR</td>
<td>EDLAR</td>
</tr>
<tr>
<td>EDLSR</td>
<td>EDLSR</td>
</tr>
</tbody>
</table>
### K17.7.4 RAS registers

This section is an index to the registers in the RAS registers functional group.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERRCIDR0</td>
<td>ERRCIDR0</td>
</tr>
<tr>
<td>ERRCIDR1</td>
<td>ERRCIDR1</td>
</tr>
<tr>
<td>ERRCIDR2</td>
<td>ERRCIDR2</td>
</tr>
<tr>
<td>ERRCIDR3</td>
<td>ERRCIDR3</td>
</tr>
<tr>
<td>ERRCRICR0</td>
<td>ERRCRICR0</td>
</tr>
<tr>
<td>ERRCRICR1</td>
<td>ERRCRICR1</td>
</tr>
<tr>
<td>ERRCRICR2</td>
<td>ERRCRICR2</td>
</tr>
<tr>
<td>ERRDEVAFF</td>
<td>ERRDEVAFF</td>
</tr>
<tr>
<td>ERRDEVARCH</td>
<td>ERRDEVARCH</td>
</tr>
<tr>
<td>ERRDEVID</td>
<td>ERRDEVID</td>
</tr>
<tr>
<td>ERRERICR0</td>
<td>ERRERICR0</td>
</tr>
<tr>
<td>ERRERICR1</td>
<td>ERRERICR1</td>
</tr>
<tr>
<td>ERRERICR2</td>
<td>ERRERICR2</td>
</tr>
<tr>
<td>ERRFHICR0</td>
<td>ERRFHICR0</td>
</tr>
</tbody>
</table>

**Table K17-44 Debug registers (continued)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDPCSR</td>
<td>EDPCSR</td>
</tr>
<tr>
<td>EDPIDR0</td>
<td>EDPIDR0</td>
</tr>
<tr>
<td>EDPIDR1</td>
<td>EDPIDR1</td>
</tr>
<tr>
<td>EDPIDR2</td>
<td>EDPIDR2</td>
</tr>
<tr>
<td>EDPIDR3</td>
<td>EDPIDR3</td>
</tr>
<tr>
<td>EDPIDR4</td>
<td>EDPIDR4</td>
</tr>
<tr>
<td>EDPRCR</td>
<td>EDPRCR</td>
</tr>
<tr>
<td>EDPRSR</td>
<td>EDPRSR</td>
</tr>
<tr>
<td>EDRCR</td>
<td>EDRCR</td>
</tr>
<tr>
<td>EDSER</td>
<td>EDSER</td>
</tr>
<tr>
<td>EDVIDSR</td>
<td>EDVIDSR</td>
</tr>
<tr>
<td>EDWAR</td>
<td>EDWAR</td>
</tr>
<tr>
<td>OSLAR_EL1</td>
<td>OSLAR_EL1</td>
</tr>
</tbody>
</table>
### Cross-trigger interface registers

This section is an index to the registers in the Cross-Trigger Interface registers functional group.

#### Table K17-46 Cross-trigger interface registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description, see</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASICCTL</td>
<td>ASICCTL</td>
</tr>
<tr>
<td>CTIAPPCLEAR</td>
<td>CTIAPPCLEAR</td>
</tr>
<tr>
<td>CTIAPPULSE</td>
<td>CTIAPPULSE</td>
</tr>
<tr>
<td>CTIAPPSET</td>
<td>CTIAPPSET</td>
</tr>
<tr>
<td>Register</td>
<td>Description, see</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>CTIAUTHSTATUS</td>
<td>CTIAUTHSTATUS</td>
</tr>
<tr>
<td>CTICHINSTATUS</td>
<td>CTICHINSTATUS</td>
</tr>
<tr>
<td>CTICHOUTSTATUS</td>
<td>CTICHOUTSTATUS</td>
</tr>
<tr>
<td>CTICIDR0</td>
<td>CTICIDR0</td>
</tr>
<tr>
<td>CTICIDR1</td>
<td>CTICIDR1</td>
</tr>
<tr>
<td>CTICIDR2</td>
<td>CTICIDR2</td>
</tr>
<tr>
<td>CTICIDR3</td>
<td>CTICIDR3</td>
</tr>
<tr>
<td>CTICCLAIMCLR</td>
<td>CTICCLAIMCLR</td>
</tr>
<tr>
<td>CTICCLAIMSET</td>
<td>CTICCLAIMSET</td>
</tr>
<tr>
<td>CTICONTROL</td>
<td>CTICONTROL</td>
</tr>
<tr>
<td>CTIDEVAFF0</td>
<td>CTIDEVAFF0</td>
</tr>
<tr>
<td>CTIDEVAFF1</td>
<td>CTIDEVAFF1</td>
</tr>
<tr>
<td>CTIDEVARCH</td>
<td>CTIDEVARCH</td>
</tr>
<tr>
<td>CTIDEVCTL</td>
<td>CTIDEVCTL</td>
</tr>
<tr>
<td>CTIDEVID</td>
<td>CTIDEVID</td>
</tr>
<tr>
<td>CTIDEVID1</td>
<td>CTIDEVID1</td>
</tr>
<tr>
<td>CTIDEVID2</td>
<td>CTIDEVID2</td>
</tr>
<tr>
<td>CTIDEVTYPE</td>
<td>CTIDEVTYPE</td>
</tr>
<tr>
<td>CTIGATE</td>
<td>CTIGATE</td>
</tr>
<tr>
<td>CTIINEN&lt;\n&gt;</td>
<td>CTIINEN&lt;\n&gt;</td>
</tr>
<tr>
<td>CTIINTACK</td>
<td>CTIINTACK</td>
</tr>
<tr>
<td>CTIITCTRL</td>
<td>CTIITCTRL</td>
</tr>
<tr>
<td>CTILAR</td>
<td>CTILAR</td>
</tr>
<tr>
<td>CTILSR</td>
<td>CTILSR</td>
</tr>
<tr>
<td>CTIOUTEN&lt;\n&gt;</td>
<td>CTIOUTEN&lt;\n&gt;</td>
</tr>
<tr>
<td>CTIPIDR0</td>
<td>CTIPIDR0</td>
</tr>
<tr>
<td>CTIPIDR1</td>
<td>CTIPIDR1</td>
</tr>
<tr>
<td>CTIPIDR2</td>
<td>CTIPIDR2</td>
</tr>
<tr>
<td>CTIPIDR3</td>
<td>CTIPIDR3</td>
</tr>
<tr>
<td>CTIPIDR4</td>
<td>CTIPIDR4</td>
</tr>
<tr>
<td>CTITRIGINSTATUS</td>
<td>CTITRIGINSTATUS</td>
</tr>
<tr>
<td>CTITRIGOUTSTATUS</td>
<td>CTITRIGOUTSTATUS</td>
</tr>
</tbody>
</table>
**Glossary**

**A32 instruction**
A word that specifies an operation to be performed by a PE that is executing in an Exception level that is using AArch32 and is in A32 state. A32 instructions must be word-aligned.

A32 instructions were previously called ARM instructions.

*See also* A32 state, A64 instruction, T32 instruction.

**A32 state**
The AArch32 Instruction set state in which the PE executes A32 instructions.

A32 state was previously called ARM state.

*See also* T32 instruction, T32 state.

**A64 instruction**
A word that specifies an operation to be performed by a PE that is executing in an Exception level that is using AArch64. A64 instructions must be word-aligned.

*See also* A32 instruction, T32 instruction.

**AArch32**
The 32-bit Execution state. In AArch32 state, addresses are held in 32-bit registers, and instructions in the base instruction sets use 32-bit registers for their processing. AArch32 state supports the T32 and A32 instruction sets.

*See also* AArch64, A32 instruction, T32 instruction.

**AArch64**
The 64-bit Execution state. In AArch64 state, addresses are held in 64-bit registers, and instructions in the base instruction set can use 64-bit registers for their processing. AArch64 state supports the A64 instruction set.

*See also* AArch32, A64 instruction.

**Abort**
An exception caused by an illegal memory access. Aborts can be caused by the external memory system or the MMU.

**Active element**
An Active element is an SVE vector element or predicate element that is a source register element or destination register element used by an SVE instruction. When the corresponding element in the instruction's Governing predicate is TRUE, the element is Active. If an instruction is unpredicated, all of the vector elements or predicate elements are implicitly Active.
Addressing mode

Means a method for generating the memory address used by a load/store instruction.

Advanced SIMD

A feature of the Arm architecture that provides SIMD operations on a register file of SIMD and floating-point registers. Where an implementation supports both Advanced SIMD and floating-point instructions, these instructions operate on the same register file.

Aligned

A data item stored at an address that is exactly divisible by the highest power of 2 that divides exactly into its size in bytes. Aligned halfwords, words and doublewords therefore have addresses that are divisible by 2, 4 and 8 respectively.

An aligned access is one where the address of the access is aligned to the size of each element of the access.

Analysis of the trace element stream

Refers to the process of:

- Tracing elements that carry information that a trace analyzer requires to enable it to analyze the trace successfully.
- Tracing elements that either directly indicate program execution, or carry information about program execution.

A trace element stream might also contain trace elements that contain timing information.

This term is distinct from analysis of program execution.

Architecturally executed

An instruction is architecturally executed only if it would be executed in a simple sequential execution of the program. When such an instruction has been executed and retired it has been architecturally executed. Any instruction that, in a simple sequential execution of a program, is treated as a NOP because it fails its condition code check, is an architecturally executed instruction.

In a PE that performs speculative execution, an instruction is not architecturally executed if the PE discards the results of a speculative execution.

See also Condition code check, Simple sequential execution.

Architecturally mapped

Where this manual describes a register as being architecturally mapped to another register, this indicates that, in an implementation that supports both of the registers, the two registers access the same state.

Architecturally UNKNOWN

An architecturally UNKNOWN value is a value that is not defined by the architecture but must meet the requirements of the definition of UNKNOWN. Implementations can define the value of the field, but are not required to do so.

See also IMPLEMENTATION DEFINED.

ARM core registers

Some older documentation uses ARM core registers to refer to the following set of registers for execution in AArch32 state:

- The 13 general-purpose registers, R0-R12, that software can use for processing.
- SP, the stack pointer, that can also be referred to as R13.
- LR, the link register, that can also be referred to as R14.
- PC, the program counter, that can also be referred to as R15.

See also General-purpose registers.

ARM instruction

See A32 instruction.

Associativity

See Cache associativity.

Asynchronous accumulation

Faults that are accumulated in a status register, where the update to the register is asynchronous to the instruction that causes the fault.
Atomicity
Describes either single-copy atomicity or multi-copy atomicity. Atomicity in the Arm architecture on page B2-154 defines these forms of atomicity for the Arm architecture.
See also Multi-copy atomicity, Single-copy atomicity.

Banked register
A register that has multiple instances, with the instance that is in use depending on the PE mode, Security state, or other PE state.

Base register
A register specified by a load/store instruction that is used as the base value for the address calculation for the instruction. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the virtual address that is sent to memory.

Base register writeback
Describes writing back a modified value to the base register used in an address calculation.

Behaves as if
Where this manual indicates that a PE behaves as if a certain condition applies, all descriptions of the operation of the PE must be re-evaluated taking account of that condition, together with any other conditions that affect operation.

Big-endian memory
Means that, for example:
- A byte or halfword at a word-aligned address is the most significant byte or halfword in the word at that address.
- A byte at a halfword-aligned address is the most significant byte in the halfword at that address.
See also Endianness, Little-endian memory.

Blocking
Describes an operation that does not permit following instructions to be executed before the operation completes.
A non-blocking operation can permit following instructions to be executed before the operation completes, and in the event of encountering an exception does not signal an exception to the PE. This enables implementations to retire following instructions while the non-blocking operation is executing, without the need to retain precise PE state.

Branch prediction
Is where a PE selects a future execution path to fetch along. For example, after a branch instruction, the PE can choose to speculatively fetch either the instruction following the branch or the instruction at the branch target.
See also Prefetching.

Breakpoint
A debug event triggered by the execution of a particular instruction, specified by one or both of the address of the instruction and the state of the PE when the instruction is executed.

Byte
An 8-bit data item.

Cache associativity
The number of locations in a cache set to which an address can be assigned. Each location is identified by its way value.

Cache level
The position of a cache in the cache hierarchy. In the Arm architecture, the lower numbered levels are those closest to the PE. For more information, see Terms used in describing the cache maintenance instructions on page D7-5050.

Cache line
The basic unit of storage in a cache. Its size in words is always a power of two, usually 4 or 8 words. A cache line must be aligned to a suitable memory boundary. A memory cache line is a block of memory locations with the same size and alignment as a cache line. Memory cache lines are sometimes loosely called cache lines.

Cache lockdown
Enables critical software and data to be loaded into the cache so that the cache lines containing them are not subsequently reallocated. It alleviates the delays caused by accessing a cache in a worst-case situation. This ensures that all subsequent accesses to the software and data concerned are cache hits and so complete quickly.

Cache miss
A memory access that cannot be processed at high speed because the data it addresses is not in the cache.

Cache sets
Areas of a cache, divided up to simplify and speed up the process of determining whether a cache hit occurs. The number of cache sets is always a power of two.
Cache way
A cache way consists of one cache line from each cache set. The cache ways are indexed from 0 to \((\text{Associativity}-1)\). Each cache line in a cache way is chosen to have the same index as the cache way. For example, cache way \(n\) consists of the cache line with index \(n\) from each cache set.

Coherence order
See Coherent.

Coherent
Data accesses from a set of observers to a byte in memory are coherent if accesses to that byte in memory by the members of that set of observers are consistent with there being a single total order of all writes to that byte in memory by all members of the set of observers. This single total order of all to writes to that memory location is the coherence order for that byte in memory.

Commit window
The Commit window defines the range of \(P^0\) elements that are committed by a Commit element. The oldest \(P^0\) element in the Commit window is the first \(P^0\) element committed when a Commit element occurs. By default, the Commit window starts on the oldest uncommitted \(P^0\) element and moves forward to the next uncommitted \(P^0\) element, with each \(P^0\) element committed by a Commit element. The Commit Window Move element moves the start of the Commit window by a number of \(P^0\) elements, to allow a Commit element to commit \(P^0\) elements that are younger than the oldest uncommitted \(P^0\) elements, leaving these older \(P^0\) elements uncommitted.

Completer
An agent in a computing system that responds to and completes a memory transaction that was initiated by a Requester.

See also Requester.

Constructive instruction encoding
A constructive instruction encoding is an instruction encoding where the destination register is encoded independently of the source registers.

Condition code check
The process of determining whether a conditional instruction executes normally or is treated as a \(\text{NOP}\). For an instruction that includes a condition code field, that field is compared with the condition flags to determine whether the instruction is executed normally. For a T32 instruction in an IT block, the value of \(\text{PSTATE}.\text{IT}\) determines whether the instruction is executed normally.

See also Condition code field, Condition flags, Conditional execution.

Condition code field
A 4-bit field in an instruction that specifies the condition under which the instruction executes.

See also Condition code check.

Condition flags
The N, Z, C, and V bits of \(\text{PSTATE}\), an SPSR, or FPSCR. See the register descriptions for more information.

See also Condition code check, \(\text{PSTATE}\).

Conditional execution
When a conditional instruction starts executing, if the condition code check returns TRUE, the instruction executes normally. Otherwise, it is treated as a \(\text{NOP}\).

See also Condition code check.

CONSTRANDED UNPREDICTABLE
Where an instruction can result in UNPREDICTABLE behavior, the Armv8 and later architectures specify a narrow range of permitted behaviors. This range is the range of CONSTRANDED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRANDED UNPREDICTABLE behavior.

Execution at Non-secure EL1 or EL0 of an instruction that is CONSTRANDED UNPREDICTABLE can be implemented as generating a trap exception that is taken to EL2, provided that at least one instruction that is not UNPREDICTABLE and is not CONSTRANDED UNPREDICTABLE causes a trap exception that is taken to EL2.

In body text, the term CONSTRANDED UNPREDICTABLE is shown in SMALL CAPITALS.

See also UNPREDICTABLE.
**Context switch**  The saving and restoring of computational state when switching between different threads or processes. In this manual, the term context switch describes any situation where the context is switched by an operating system and might or might not include changes to the address space.

**Context synchronization event**

One of:

- Performing an ISB operation. An ISB operation is performed when an ISB instruction is executed and does not fail its condition code check.
- Exception entry, if FEAT_ExS is not implemented or the exception is taken to AArch32 or if FEAT_ExS is implemented and the appropriate SCTLR_ELx.EIS bit is set.
- Return from an exception, if FEAT_ExS is not implemented, or the exception is returning from AArch32 or if FEAT_ExS is implemented and the appropriate SCTLR_ELx.EOS bit is set.
- Exit from Debug state.
- Executing a DCPs instruction.
- Executing a DRPS instruction.

The effects of a Context synchronization event are:

- All unmasked interrupts that are pending at the time of the Context synchronization event are taken before the first instruction after the Context synchronization event.
- If halting is allowed, all Halting debug events that are pending at the time of the Context synchronization event are taken before the first instruction after the Context synchronization event.
- No instructions appearing in program order after an instruction that causes a Context synchronization event will have performed any part of their functionality until the Context synchronization event has occurred.
- All direct and indirect writes to System registers that are made before the Context synchronization event affect any instruction, including a direct read, that appears in program order after the instruction causing the Context synchronization event.
- All completed changes to the translation tables for entries that, before the change, were not permitted to be cached in a TLB, affect all instruction fetches that appear in program order after the instruction causing the Context synchronization event.
- All invalidations of TLBs, instruction caches, and, in AArch32 state, branch predictors, that are completed before the Context synchronization event affect all instructions that appear in program order after an instruction causing a Context synchronization event.
- In AArch32 state, all Non-cacheable writes that are completed before the Context synchronization event affect all instructions that appear in program order after an instruction causing a Context synchronization event.
- Changes to the Debug external authentication interfaces that are made before the Context synchronization event affect any instruction that appears in program order after the instruction causing the Context synchronization event.

---

**Note**

- The architecture requires that instructions that generate Context synchronization events do not appear to be executed speculatively, except that the performance monitor counters are permitted to reveal such speculation.
- *Context synchronization events* were previously described as *context synchronization operations*.

**Conventional memory**

Memory locations from which generic OSs and application run-times will expect to create allocations for general software use.

**CoreSight Address Translation Unit**

A form of System MMU for trace streams.
Data independent timing (DIT)
The time that it takes to execute a piece of code where the time is not a function of the data being operated on. For more information, see About PSTATE.DIT on page B1-147 and About the DIT bit on page E1-7139.

Debugger
In most of this manual, debugger refers to any agent that is performing debug. However, some chapters or parts of this manual require a more rigorous definition, and define debugger locally. See:
- Definition of a debugger in the context of self-hosted debug on page D2-4692.
- Definition of a debugger in the context of self-hosted debug on page G2-9038.
- Definition and constraints of a debugger in the context of external debug on page H1-10228.

Deprecated
Something that is present in the Arm architecture for backwards compatibility. Whenever possible software must avoid using deprecated features. Features that are deprecated but are not optional are present in current implementations of the Arm architecture, but might not be present, or might be deprecated and OPTIONAL, in future versions of the Arm architecture.

See also OPTIONAL.

Destructive instruction encoding
A destructive instruction encoding is an instruction encoding where one of the source registers is also used as the destination register.

Digital signal processing (DSP)
Algorithms for processing signals that have been sampled and converted to digital form. DSP algorithms often use saturated arithmetic.

Direct Memory Access (DMA)
An operation that accesses main memory directly, without the PE performing any accesses to the data concerned.

Direct read
A direct read of a System register is a read performed by a System register access instruction.

For more information, see Direct read on page D17-5549.

See also Direct write, Indirect read, Indirect write.

Direct write
A direct write of a System register is a write performed by a System register access instruction.

For more information, see Direct write on page D17-5549.

See also Direct read, Indirect read, Indirect write.

DMA
See Direct Memory Access (DMA).

DNM
See Do-Not-Modify (DNM).

Domain
In the Arm architecture, domain is used in the following contexts.

Shareability domain Defines a set of observers for which the shareability attributes make the data or unified caches transparent for data accesses.

Power domain Defines a block of logic with a single, common, power supply.

Memory regions domain
When using the Short-descriptor translation table format, defines a collection of Sections, Large pages and Small pages of memory, that can have their access permissions switched rapidly by writing to the Domain Access Control Register (DACR). Arm deprecates any use of memory regions domains.

Do-Not-Modify (DNM)
Means the value must not be altered by software. DNM fields read as UNKNOWN values, and must only be written with the value read from the same field on the same PE.

Double-precision value
Consists of two consecutive 32-bit words that are interpreted as a basic double-precision floating-point number according to the IEEE Standard for Floating-point Arithmetic.

Doubleword
A 64-bit data item. Doublewords are normally at least word-aligned in Arm systems.
**Doubleword-aligned**
Means that the address is divisible by 8.

**DSP**
See Digital signal processing (DSP).

**Effective value**
A register control field, meaning a field in a register that controls some aspect of the behavior, can be described as having an **Effective value**:

- In some cases, the description of a control $a$ specifies that when control $a$ is active it causes a register control field $b$ to be treated as having a fixed value for all purposes other than direct reads, or direct reads and direct writes, of the register containing control field $b$. When control $a$ is active that fixed value is described as the **Effective value** of register control field $b$. For example, when the value of HCR.DC is 1, the **Effective value** of HCR.VM is 1, regardless of its actual value.

  - **Note**
    Where a register control field is introduced in a particular version of the architecture, and is not implemented in an earlier version of the architecture, typically it will have an **Effective value** in that earlier version of the architecture.

  - Otherwise, the **Effective value** of a register control field is the value of that field.

**Effective SVE vector length**
The current accessible SVE vector length as constrained by ZCR_ELx and the Maximum implemented SVE vector length.

**Element number**
For a given element size of $N$ bits, elements within a vector or predicate register are numbered with element[0] always representing bits[(N-1):0], element[1] always representing bits[(2N-1):N], and so on.

**Endianness**
An aspect of the system memory mapping.
See also Big-endian memory and Little-endian memory.

**ETEEvent**
A feature of the trace unit that is used to generate **Event elements** and drive External Outputs. Each ETEEvent can be programmed to be sensitive to resource events.

**Event trace**
The trace uses **Event elements** that indicate certain events have occurred in the program that the PE is executing. The program events to be indicated are selected before a trace session.

**Exceptional occurrence**
Events indicated by an **Exception element** by the ETE architecture, including the following:

- PE architectural exceptions.
- ETE defined exceptions.
- IMPLEMENTATION DEFINED exceptions.

**Exception vector**
Handles an event. For example, an exception could handle an external interrupt or an undefined instruction.

**Execution stream**
The stream of instructions that would have been executed by sequential execution of the program.

**Explicit memory effect**
A read from memory, or a write to memory, generated by a load or store instruction executed by the PE. Reads and writes generated by hardware translation table accesses, as well as instruction fetches and SPE writes to the profiling buffer, are not explicit memory effects.

**External abort**
An abort that is generated by the external memory system.
Fast Context Switch Extension (FCSE)
Modifies the behavior of an Arm memory system to enable multiple programs running on the Arm PE to use identical address ranges, while ensuring that the addresses they present to the rest of the memory system differ. The FCSE is:
• Deprecated from Armv6 and not supported in Armv8.
• Obsolete from the introduction of the Multiprocessing Extensions.

See Fast Context Switch Extension (FCSE).

First active element
The First active element of an SVE vector or predicate register is the lowest numbered element that is an Active element.

First-fault load
SVE provides a First-fault option for some SVE vector load instructions. This option causes memory access faults to be suppressed if they do not occur as a result of the First active element of the vector. Instead, the FFR is updated to indicate which of the active vector elements were not successfully loaded.

Flat address mapping
Is where the physical address for every access is equal to its virtual address.

Flush-to-zero mode
A processing mode that optimizes the performance of some floating-point algorithms by replacing the denormalized operands and Intermediate results with zeros, without significantly affecting the accuracy of their final results.

Gather-load
Gather-load is a mechanism that allows the elements of a vector to be read from non-contiguous memory locations using a vector of addresses, where the addresses are constructed according to the addressing mode.

General-purpose registers
The registers that the base instructions use for processing:

• In AArch32 state the general-purpose registers are R0-R14, that can also be described as R0-R12, SP, LR.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Older documentation defines the AArch32 general-purpose registers as R0-R12, and the Arm core registers as R0-R12, SP, LR, and PC.</td>
</tr>
</tbody>
</table>

• In AArch64 state the general-purpose registers are:
  --- W0-W30 when accessed as 32-bit registers.
  --- X0-X30 when accessed as 64-bit registers.

See also High registers, Low registers.

Generated by
The memory model is written in terms of reads from memory and writes to memory. These reads and writes:

• Are generated by instructions such as loads, stores, and atomic memory accesses.
• Correspond to the memory accesses, other than translation table walks, that are defined in the instruction pseudocode.

Some instructions generate more than one read or write.

Governing predicate
The predicate register that is used to determine the Active elements of a predicated instruction is known as the Governing predicate for that instruction.

Halfword
A 16-bit data item. Halfwords are normally halfword-aligned in Arm systems.

Halfword-aligned
Means that the address is divisible by 2.
High registers
In AArch32 state, the general-purpose registers R8-R14. Most 16-bit T32 instructions cannot access the high registers.

Note
In some contexts, high registers refers to R8-R15, meaning R8-R14 and the PC.

See also General-purpose registers, Low registers.

High vectors
An alternative location for the exception vectors. The high vector address range is near the top of the address space, rather than at the bottom.

IGNORED
Indicates that the architecture guarantees that the bit or field is not interpreted or modified by hardware.

In body text, the term IGNORED is shown in SMALL CAPITALS.

Immediate and offset fields
Are unsigned unless otherwise stated.

Immediate value
A value that is encoded directly in the instruction and used as numeric data when the instruction is executed. Many A64, A32, and T32 instructions can be used with an immediate argument.

IMP
An abbreviation used in diagrams to indicate that one or more bits have IMPLEMENTATION DEFINED behavior.

IMPLEMENTATION DEFINED
Means that the behavior is not architecturally defined, but must be defined and documented by individual implementations.

In body text, the term IMPLEMENTATION DEFINED is shown in SMALL CAPITALS.

Inactive element
An Inactive element is an SVE vector element or predicate element that is an unused source register element or destination register element for the associated instruction. When the corresponding element of an instruction's Governing predicate is FALSE, the element is inactive.

Index register
A register specified in some load and store instructions. The value of this register is used as an offset to be added to or subtracted from the base register value to form the virtual address that is sent to memory. Some instruction forms permit the index register value to be shifted before the addition or subtraction.

Indirect read
When an instruction uses a System register value to establish operating conditions, that use of the System register is an indirect read of the System register.

For more information, including additional examples of indirect reads, see Indirect read on page D17-5549.

See also Direct read, Direct write, Indirect write.

Indirect write
An indirect write of a System register occurs when the contents of a register are updated by some mechanism other than a Direct write to that register. For example, an indirect write to a register might occur as a side-effect of executing an instruction that does not perform a direct write to the register, or because of some operation performed by an external agent.

For more information, see Indirect write on page D17-5549.

See also Direct read, Direct write, Indirect read.

Inline literals
These are constant addresses and other data items held in the same area as the software itself. They are automatically generated by compilers, and can also appear in assembler code.

Intermediate physical address (IPA)
An implementation of virtualization, the address to which a Guest OS maps a VA. A hypervisor might then map the IPA to a PA. Typically, the Guest OS is unaware of the translation from IPA to PA.

See also Physical address (PA), Virtual address (VA).

Interworking
A method of working that permits branches between software using the A32 and T32 instruction sets.
IPA  
See Intermediate physical address (IPA).

Last active element  
The Last active element of an SVE vector or predicate register is the highest numbered element that is an Active element.

Level  
See Cache level.

Level of Coherence (LoC)  
The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of coherency. For more information, see Terms used in describing the cache maintenance instructions on page D7-5050.

See also Cache level, Point of coherency (PoC).

Level of Unification, Inner Shareable (LoUIS)  
The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for the Inner Shareable shareability domain. For more information, see Terms used in describing the cache maintenance instructions on page D7-5050.

See also Cache level, Point of unification (PoU).

Level of Unification, uniprocessor (LoUU)  
For a PE, the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for that PE. For more information, see Terms used in describing the cache maintenance instructions on page D7-5050.

See also Cache level, Point of unification (PoU).

Line  
See Cache line.

Little-endian memory  
Means that, for example:

• A byte or halfword at a word-aligned address is the least significant byte or halfword in the word at that address.

• A byte at a halfword-aligned address is the least significant byte in the halfword at that address.

See also Big-endian memory, Endianness.

Load/Store architecture  
An architecture where data-processing operations only operate on register contents, not directly on memory contents.

LoC  
See Level of Coherence (LoC).

LoUIS  
See Level of Unification, Inner Shareable (LoUIS).

LoUU  
See Level of Unification, uniprocessor (LoUU).

Lockdown  
See Cache lock down.

Low registers  
In AArch32 state, general-purpose registers R0-R7. Unlike the high registers, all T32 instructions can access the Low registers.

See also General-purpose registers, High registers.

Maximum implemented SVE vector length  
The maximum SVE vector length supported by an implementation.

Memory barrier  
See Memory barriers on page B2-174.

Memory coherency  
The problem of ensuring that when a memory location is read, either by a data read or an instruction fetch, the value actually obtained is always the value that was most recently written to the location. This can be difficult when there are multiple possible physical locations, such as main memory and at least one of a write buffer and one or more levels of cache.
Memory element
An item of data in memory that is transferred to or from a vector element by an SVE load or store instruction. Each memory element has an access size and a type. The memory element access size is specified by each load and store instruction independently of the vector element size.

Memory Management Unit (MMU)
Provides detailed control of the part of a memory system that provides a single stage of address translation. Most of the control is provided using translation tables that are held in memory, and define the attributes of different regions of the physical memory map.

Memory Protection Unit (MPU)
A hardware unit whose registers provide simple control of a limited number of protection regions in memory.

Merging predication
When a predicated SVE instruction specifies merging predication, the Inactive elements of the destination register remain unchanged.

Microarchitecturally-finished
An operation that has finished all of its operational pseudocode, although the results of any memory accesses, including translation table walks and updates, are not yet coherent with other observers.

Microarchitecturally-unfinished
An operation that has not completed all of its operational pseudocode.

Miss
See Cache miss.

MMU
See Memory Management Unit (MMU).

MPU
See Memory Protection Unit (MPU).

Multi-copy atomicity
The form of atomicity described in Requirements for multi-copy atomicity on page B2-156.
See also Atomicity, Single-copy atomicity.

NaN
Not a Number. A floating-point value that can be used when neither a numeric value nor an infinity is appropriate. A NaN can be a quiet NaN, that propagate through most floating-point operations, or a signaling NaN, that causes an Invalid Operation floating-point exception when used. For more information, see the IEEE Standard for Floating-point Arithmetic.
See also Quiet NaN, Signaling NaN.

Natural eviction
A natural eviction is an eviction that occurs in the course of the normal operation of the memory system, rather than because of an operation that explicitly causes an eviction from the cache, such as the execution of a cache maintenance instruction. Typically, a natural eviction occurs when the caching algorithm requires data to be cached but the cache does not have room for that data.

Non-fault load
SVE provides a Non-fault option for some SVE vector load instructions. This option causes all memory access faults to be suppressed. Instead, the FFR is updated to indicate which of the active vector elements were not successfully loaded.

Observer
A PE or mechanism in the system, such as a peripheral device, that can generate reads from or writes to memory.

Obsolete
Obsolete indicates something that is no longer supported by Arm. When an architectural feature is described as obsolete, this indicates that the architecture has no support for that feature, although an earlier version of the architecture did support it.

Offset addressing
Means that the memory address is formed by adding or subtracting an offset to or from the base register value.

OPTIONAL
When applied to a feature of the architecture, OPTIONAL indicates a feature that is not required in an implementation of the Arm architecture:

• If a feature is OPTIONAL and deprecated, this indicates that the feature is being phased out of the architecture. Arm expects such a features to be included in a new implementation only if there is a known backwards-compatibility reason for the inclusion of the feature.
A feature that is **OPTIONAL** and deprecated might not be present in future versions of the architecture.

- A feature that is **OPTIONAL** but not deprecated is, typically, a feature added to a version of the Arm architecture after the initial release of that version of the architecture. Arm recommends that such features are included in all new implementations of the architecture.

In body text, these meanings of the term **OPTIONAL** are shown in **SMALL CAPITALS**.

--- **Note** ---

Do not confuse these Arm-specific uses of **OPTIONAL** with other uses of **optional**, where it has its usual meaning. These include:

- Optional arguments in the syntax of many instructions.
- Behavior determined by an implementation choice, for example the optional byte order reversal in an Armv7-R implementation, where the SCTLR.IE bit indicates the implemented option.

---

*See also* Deprecated.

**PA**

*See* Physical address (PA).

**Packed access**

A memory access that is performed as a result of an SVE load or store instruction for which the vector element size and the memory element size are the same.

**PE**

*See* Processing element (PE).

**Physical address (PA)**

An address that identifies a location in the physical memory map. *See also* Intermediate physical address (IPA), Virtual address (VA).

**PoC**

*See* Point of coherency (PoC).

**PoP**

*See* Point of persistence (PoP).

**PoU**

*See* Point of unification (PoU).

**Point of coherency (PoC)**

For a particular VA, the point at which all agents that can access memory are guaranteed to see the same copy of a memory location. For more information, see *Terms used in describing the cache maintenance instructions* on page D7-5050.

**Point of persistence (PoP)**

The point in a memory system where there is a system guarantee that there is sufficient energy within the system to ensure that a write to memory will be persistent if system power is removed. For more information, see *Terms used in describing the cache maintenance instructions* on page D7-5050.

**Point of unification (PoU)**

For a particular PE, the point by which the instruction and data caches and the translation table walks of that PE are guaranteed to see the same copy of a memory location. For more information, see *Terms used in describing the cache maintenance instructions* on page D7-5050.

**Post-indexed addressing**

Means that the memory address is the base register value, but an offset is added to or subtracted from the base register value and the result is written back to the base register.

**Predicate**

A one-dimensional array of predicate elements of the same size.

**Predicate element**

Individual subdivisions of a predicate register that can be 1, 2, 4, or 8 bits in size. The predicate element size is specified independently by each instruction and is always one-eighth the size of the corresponding vector element. The lowest-numbered bit of each predicate element holds the Boolean value of that element, where 1 represents TRUE and 0 represents FALSE.
**Predicated instruction**
An SVE instruction that has a Governing predicate operand, which determines the Active and Inactive elements for that instruction.

**Predicate register**
An SVE predicate register, P0-P15, having a length that is a multiple of 16 bits, in the range 16 to 256, inclusive.

**Prefetching**
Prefetching refers to speculatively fetching instructions or data from the memory system. In particular, instruction prefetching is the process of fetching instructions from memory before the instructions that precede them, in simple sequential execution of the program, have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

In this manual, references to instruction or data fetching apply also to prefetching, unless the context explicitly indicates otherwise.

--- **Note** ---
The Prefetch Abort exception can be generated on any instruction fetch, and is not limited to speculative instruction fetches.

---

**Prefixed instruction**
The instruction that immediately follows a MOVPRFX instruction in program order.

**Pre-indexed addressing**
Means that the memory address is formed in the same way as for offset addressing, but the memory address is also written back to the base register.

**Processing element (PE)**

**Protection region**
A memory region whose position, size, and other properties are defined by Memory Protection Unit registers.

**Protection Unit**
See Memory Protection Unit (MPU).

**Pseudo-instruction**
UAL assembler syntax that assembles to an instruction encoding that is expected to disassemble to a different assembler syntax, and is described in this manual under that other syntax. For example, MOV <Rd>, <Rm>, LSL #<n> is a pseudo-instruction that is expected to disassemble as LSL <Rd>, <Rm>, #<n>.

**PSTATE**
An abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

See also Condition flags.

**Quadword**
A 128-bit data item. Quadwords are normally at least word-aligned in Arm systems.

**Quadword-aligned**
Means that the address is divisible by 16.

**Quiet NaN**
A NaN that propagates unchanged through most floating-point operations.

See also NaN, Signaling NaN.

**RAO**
See Read-As-One (RAO).

**RAZ**
See Read-As-Zero (RAZ).

**RAO/SBOP**
In versions of the Arm architecture before Armv8, Read-As-One, Should-Be-One-or-Preserved on writes.

From the introduction of the Armv8 architecture, RES1 replaces this description.

See also UNK/SBOP, Read-As-One (RAO), RES1, Should-Be-One-or-Preserved (SBOP).
RAO/WI    
Read-As-One, Writes Ignored.  
Hardware must implement the field as Read-as-One, and must ignore writes to the field.  
Software can rely on the field reading as all 1s, and on writes being ignored.  
This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.  
See also Read-As-One (RAO).

RAZ/SBZP 
In versions of the Arm architecture before Armv8, Read-As-Zero, Should-Be-Zero-or-Preserved on writes.  
From the introduction of the Armv8 architecture, RES0 replaces this description.  
See also UNK/SBZP, Read-As-Zero (RAZ), RES0, Should-Be-Zero-or-Preserved (SBZP).

RAZ/WI    
Read-As-Zero, Writes Ignored.  
Hardware must implement the field as Read-as-Zero, and must ignore writes to the field.  
Software can rely on the field reading as all 0s, and on writes being ignored.  
This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.  
See also Read-As-Zero (RAZ).

Read-allocate cache  
A cache in which a cache miss on reading data causes a cache line to be allocated into the cache.

Read-As-One (RAO)  
Hardware must implement the field as reading as all 1s.  
Software:  
• Can rely on the field reading as all 1s.  
• Must use a SBOP policy to write to the field.  
This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.  
See also RAO/SBOP, RAO/WI, RES1.

Read-As-Zero (RAZ)  
Hardware must implement the field as reading as all 0s.  
Software:  
• Can rely on the field reading as all 0s  
• Must use a SBZP policy to write to the field.  
This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.  
See also RAZ/SBZP, RAZ/WI, RES0.

Read, modify, write  
In a read, modify, write instruction sequence, a value is read to a general-purpose register, the relevant fields updated in that register, and the new value written back.

Requester  
An agent in a computing system that is capable of initiating memory transactions.  
See also Completer.

RES0    
A reserved bit. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.  
Within the architecture, there are some cases where a register bit or field:  
• Is RES0 in some defined architectural context.  
• Has different defined behavior in a different architectural context.  

Note  
RES0 is not used in descriptions of instruction encodings.
• Where an AArch32 System register is *Architecturally mapped* to an AArch64 System register, and a bit or field in that register is RES0 in one Execution state and has defined behavior in the other Execution state, this is an example of a bit or field with behavior that depends on the architectural context.

This means the definition of RES0 for fields in read/write registers is:

**If a bit is RES0 in all contexts**

For a bit in a read/write register, it is *IMPLEMENTATION DEFINED* whether:

1. The bit is hardwired to 0. In this case:
   • Reads of the bit always return 0.
   • Writes to the bit are ignored.
2. The bit can be written. In this case:
   • An indirect write to the register sets the bit to 0.
   • A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.
   If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an *UNKNOWN* value.
   • A direct write to the bit must update a storage location associated with the bit.
   • The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES0 bits or fields follow behavior 1 or behavior 2 is *IMPLEMENTATION DEFINED* on a bit-by-bit basis.

**If a bit is RES0 only in some contexts**

For a bit in a read/write register, when the bit is described as RES0:

• An indirect write to the register sets the bit to 0.
• A read of the bit must return the value last successfully written to the bit, by either a direct or an indirect write, regardless of the use of the register when the bit was written.
  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an *UNKNOWN* value.
• A direct write to the bit must update a storage location associated with the bit.
• While the use of the register is such that the bit is described as RES0, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is *IMPLEMENTATION DEFINED* whether:

• Writes to the bit are ignored, and reads of the bit return an *UNKNOWN* value.
• The value of the bit can be written, and a read returns the last value written to the bit.

The RES0 description can apply to bits or fields that are read-only, or are write-only:

• For a read-only bit, RES0 indicates that the bit reads as 0, but software must treat the bit as *UNKNOWN*.
• For a write-only bit, RES0 indicates that software must treat the bit as *SBZ*.

A bit that is RES0 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

• Must not rely on the bit reading as 0.
• Must use an *SBZP* policy to write to the bit.

This RES0 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES0.
In body text, the term RES0 is shown in SMALL CAPITALS.

See also Read-As-Zero (RAZ), RES0H, RES1, Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

**RES0H**

A reserved bit or field with SBZP. This behavior uses the hardwired to 0 subset of the RES0 definition.

See also Read-As-Zero (RAZ), RES0, RES1, Should-Be-Zero-or-Preserved (SBZP).

**RES1**

A reserved bit. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Within the architecture, there are some cases where a register bit or field:

- Is RES1 in some defined architectural context.
- Has different defined behavior in a different architectural context.

--- **Note**

- RES1 is not used in descriptions of instruction encodings.
- Where an AArch32 System register is *Architecturally mapped* to an AArch64 System register, and a bit or field in that register is RES1 in one Execution state and has defined behavior in the other Execution state, this is an example of a bit or field with behavior that depends on the architectural context.

This means the definition of RES1 for fields in read/write registers is:

**If a bit is RES1 in all contexts**

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 1. In this case:
   - Reads of the bit always return 1.
   - Writes to the bit are ignored.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to 1.
   - A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.
   - If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
   - A direct write to the bit must update a storage location associated with the bit.
   - The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES1 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a bit-by-bit basis.

**If a bit is RES1 only in some contexts**

For a bit in a read/write register, when the bit is described as RES1:

- An indirect write to the register sets the bit to 1.
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

--- **Note**

As indicated in this list, this value might be written by an indirect write to the register.

---

If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.

- A direct write to the bit must update a storage location associated with the bit.
• While the use of the register is such that the bit is described as RES1, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:
  • Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
  • The value of the bit can be written, and a read returns the last value written to the bit.

The RES1 description can apply to bits or fields that are read-only, or are write-only:
  • For a read-only bit, RES1 indicates that the bit reads as 1, but software must treat the bit as UNKNOWN.
  • For a write-only bit, RES1 indicates that software must treat the bit as SBO.

A bit that is RES1 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:
  • Must not rely on the bit reading as 1.
  • Must use an SBOP policy to write to the bit.

This RES1 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES1.

In body text, the term RES1 is shown in SMALL CAPITALS.

See also Read-As-One (RAO), RES0, RES0H, Should-Be-One-or-Preserved (SBOP), UNKNOWN.

Reserved

Unless otherwise stated:
  • Instructions that are reserved or that access reserved registers have UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.
  • Bit positions described as reserved are:
    — In an RW or WO register, RES0.
    — In an RO register, UNK.

See also CONSTRAINED UNPREDICTABLE, RES0, RES0H, RES1, UNDEFINED, UNK, UNPREDICTABLE.

RESS

Reserved, Sign extended. A register value is extended by copying the sign bit into all of the reserved bits to the left of the most significant bit of the field. The values of these bits are identical to the most significant bit of the value being extended.

Within the architecture, a register bit or field can be treated:
  • As RESS in few defined architectural contexts.
  • In a different defined behavior in other architectural contexts.

Rewind point

A rewind point is a point in the program flow to which execution can return if all subsequent execution is found to have been incorrectly speculatively executed.

RISC

Reduced Instruction Set Computer.

Rounding error

The value of the rounded result of an arithmetic operation minus the exact result of the operation.

Rounding mode

Specifies how the exact result of a floating-point operation is rounded to a value that is representable in the destination format. The rounding modes are defined by the IEEE Standard for Floating-point Arithmetic, see Floating-point standards, and terminology on page A1-57.

Saturated arithmetic

Integer arithmetic in which a result that would be greater than the largest representable number is set to the largest representable number, and a result that would be less than the smallest representable number is set to the smallest representable number. Signed saturated arithmetic is often used in DSP algorithms. It contrasts with the normal signed integer arithmetic used in Arm processors, in which overflowing results wrap around from +2^{31}–1 to –2^{31} or vice versa.

SBO

See Should-Be-One (SBO).

SBOP

See Should-Be-One-or-Preserved (SBOP).
**Glossary**

**SBZ**  
See Should-Be-Zero (SBZ).

**SBZP**  
See Should-Be-Zero-or-Preserved (SBZP).

**Scalable vector register**  
An SVE vector register, Z0-Z31, having a length that is a multiple of 128 bits, in the range 128 bits to 2048 bits, inclusive.

**Scalar base register**  
A scalar base register refers to an AArch64 general-purpose register, X0-X31, or the current stack pointer, SP.

**Scalar index register**  
A scalar index register refers to an AArch64 general-purpose register, X0-X31, or for certain instructions, XZR.

**Scatter-store**  
Scatter-store is a mechanism that allows the elements of a vector to be written to non-contiguous memory locations using a vector of addresses, where the addresses are constructed according to the addressing mode.

**Security hole**  
A mechanism by which execution at the current level of privilege can achieve an outcome that cannot be achieved at the current or a lower level of privilege using instructions that are not UNPREDICTABLE and are not CONSTRAINED UNPREDICTABLE. The Arm architecture forbids security holes.  
See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

**Self-modifying code**  
Code that writes one or more instructions to memory and then executes them. When using self-modifying code, you must use cache maintenance and barrier instructions to ensure synchronization. For more information, see Caches and memory hierarchy on page B2-184.

**Set**  
See Cache sets.

**Should-Be-One (SBO)**  
Hardware must ignore writes to the field.  
Arm strongly recommends that software writes the field as all 1s. If software writes a value that is not all 1s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.  
This description can apply to a single bit that should be written as 1, or to a field that should be written as all 1s.  
See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

**Should-Be-One-or-Preserved (SBOP)**  
From the introduction of the Armv8 architecture, the description Should-Be-One-or-Preserved (SBOP) is superseded by RES1.  
Hardware must ignore writes to the field.  
When writing this field, software must either write all 1s to this field or, if the register is being restored from a previously read state, write the previously read value to this field. If this is not done, then the result is unpredictable.  
This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.  
See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

**Should-Be-Zero (SBZ)**  
Hardware must ignore writes to the field.  
Arm strongly recommends that software writes the field as all 0s. If software writes a value that is not all 0s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.  
This description can apply to a single bit that should be written as 0, or to a field that should be written as all 0s.  
See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

**Should-Be-Zero-or-Preserved (SBZP)**  
From the introduction of the Armv8 architecture, the description Should-Be-Zero-or-Preserved (SBZP) is superseded by RES0.
Hardware must ignore writes to the field.

When writing this field, software must either write all 0s to this field or, if the register is being restored from a previously read state, write the previously read value to this field. If this is not done, then the result is unpredictable.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

**Signaling NaN**

An Invalid Operation floating-point exception occurs whenever any floating-point operation receives a signaling NaN as an operand. Signaling NaNs can be used in debugging, to track down some uses of uninitialized variables.

See also NaN, Quiet NaN.

**Signed immediate and offset fields**

Are encoded in two’s complement notation unless otherwise stated.

**SIMD**

Single-Instruction, Multiple-Data.

The SIMD instructions in AArch32 state are:

- The instructions summarized in *Parallel addition and subtraction instructions on page F2-7266*.
- The Advanced SIMD instructions summarized in *Advanced SIMD and floating-point instructions on page E1-7140*, when operating on vectors.

**Simple sequential execution**

The behavior of an implementation that fetches, decodes and completely executes each instruction before proceeding to the next instruction. Such an implementation performs no speculative accesses to memory, including to instruction memory. The implementation does not pipeline any phase of execution. In practice, this is the theoretical execution model that the architecture is based on, and Arm does not expect this model to correspond to a realistic implementation of the architecture.

**Single-copy atomicity**

The form of atomicity described in *Properties of single-copy atomic accesses on page B2-156*.

See also Atomicity, Multi-copy atomicity.

**Single-precision value**

A 32-bit word that is interpreted as a basic single-precision floating-point number according to the *IEEE Standard for Floating-point Arithmetic*.

**Spatial locality**

The observed effect that after a program has accessed a memory location, it is likely to also access nearby memory locations in the near future. Caches with multi-word cache lines exploit this effect to improve performance.

**Special-purpose register**

One of a specified set of registers for which all direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization:

- *Special-purpose registers on page C5-671* specifies the AArch64 Special-purpose registers.
- *AArch32 Special-purpose registers on page G1-8918* lists the AArch32 Special-purpose registers.

**Speculative**

Speculative operations are:

- Operations that are generated by instructions that appear in the Execution stream after a branch that is not architecturally resolved.
- Operations that are generated by instructions that appear in the Execution stream after an instruction where a synchronous exception condition has not been architecturally resolved.
- Operations that are generated by conditional instructions for which the conditions for the instruction have not been architecturally resolved.
- Operations that are generated by instructions that appear in the Execution stream after the point at which a precise asynchronous exception will be taken.
• Reads or writes generated by load or store instructions for which the data being written or the address being accessed comes from a register that has not been architecturally resolved.

• Operations generated by the hardware that are not directly generated by any instructions appearing in the Execution stream.

• Memory effects (M2) generated by load or store instructions (LS2) appearing in program order after load or store instructions (LS1) that generate memory effects (M1) where all of the following apply:
  — M1 is locally-ordered-before M2.
  — LS1 has not been executed before LS2.

See also Execution stream.

Speculative Microarchitecturally-finished
An operation that has finished all of its operational pseudocode, on a predicted execution path.

Speculative Microarchitecturally-unfinished
An operation that is in progress on a predicted execution path.

Superpriority
An attribute that is used to denote virtual and physical IRQ and FIQ interrupts as non-maskable. IRQ and FIQ interrupts with Superpriority can be taken under certain conditions where usually they would be masked by the PSTATE.{I, F} bits.

T32 instruction
One or two halfwords that specify an operation to be performed by a PE that is executing in an Exception level that is using AArch32 and is in T32 state. T32 instructions must be halfword-aligned.

T32 state
The AArch32 Instruction set state in which the PE executes T32 instructions.

T32 state was previously called Thumb state.

See also A32 instruction, A64 instruction, T32 state.

Taken locally
Taken locally is a qualifier that determines which instances of an exception are counted by particular PMU events. See Counting exceptions taken locally or not taken locally on page D11-5273.

In this context, an exception that is Taken locally means an exception that is one of:
• Taken to the current Exception level.
  
  Note
  This is not possible when the current Exception level is EL0.

• Taken from EL0 to EL1.
• Taken from EL0 to EL2 because the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

  Note
  An exception taken from EL0 to EL2 because the Effective value of HCR_EL2.{E2H, TGE} is {0, 1} is not Taken locally. This includes exceptions taken to EL2 using AArch32 when HCR.TGE is 1.

Temporal locality
The observed effect that after a program has accesses a memory location, it is likely to access the same memory location again in the near future. Caches exploit this effect to improve performance.

Thumb instruction
See T32 instruction.

TLB
See Translation Lookaside Buffer (TLB).

TLB lockdown
A way to prevent specific translation table walk results being accessed. This ensures that accesses to the associated memory areas never cause a translation table walk.
Translation Lookaside Buffer (TLB)
A memory structure containing the results of translation table walks. They help to reduce the average cost of a memory access. Usually, there is a TLB for each memory interface of the Arm implementation.

Translation table
A table held in memory that defines the properties of memory areas of various sizes from 1KB to 1MB.

Translation table walk
The process of doing a full translation table lookup. It is performed automatically by hardware.

Trap enable bits
In VFPv2, VFPv3U, and VFPv4U, determine whether trapped or untrapped exception handling is selected. If trapped exception handling is selected, the way it is carried out is IMPLEMENTATION DEFINED.

Unaligned
An unaligned access is an access where the address of the access is not aligned to the size of an element of the access.

Unaligned memory accesses
Are memory accesses that are not, or might not be, appropriately halfword-aligned, word-aligned, or doubleword-aligned.

Unallocated
Except where otherwise stated in this manual, an instruction encoding is unallocated if the architecture does not assign a specific function to the entire bit pattern of the instruction, but instead describes it as CONSTRAINED UNPREDICTABLE, UNDEFINED, UNPREDICTABLE, or as an unallocated hint instruction.

A bit in a register is unallocated if the architecture does not assign a function to that bit.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED, UNPREDICTABLE.

UNDEFINED
Indicates cases where an attempt to execute a particular encoding bit pattern generates an exception, that is taken to the current Exception level, or to the default Exception level for taking exceptions if the UNDEFINED encoding was executed at EL0. This applies to:

- Any encoding that is not allocated to any instruction.
- Any encoding that is defined as never accessible at the current Exception level.
- Some cases where an enable, disable, or trap control means an encoding is not accessible at the current Exception level.

If the generated exception is taken to an Exception level that is using AArch32 then it is taken as an Undefined Instruction exception.

Note
On reset, the default Exception level for taking exceptions from EL0 is EL1. However, an implementation might include controls that can change this, effectively making EL1 inactive. See the description of the Exception model for more information

In body text, the term UNDEFINED is shown in SMALL CAPITALS.

See also Undefined Instruction exception on page G1-8964.

Unified cache
Is a cache used for both processing instruction fetches and processing data loads and stores.

Unindexed addressing
Means addressing in which the base register value is used directly as the virtual address to send to memory, without adding or subtracting an offset. In most types of load/store instruction, unindexed addressing is performed by using offset addressing with an immediate offset of 0.

UNK
An abbreviation indicating that software must treat a field as containing an UNKNOWN value.

Hardware must implement the bit as read as 0, or all 0s for a multi-bit field. Software must not rely on the field reading as zero.

See also UNKNOWN.

UNK/SBOP
Hardware must implement the field as Read-As-One, and must ignore writes to the field.
Software must not rely on the field reading as all 1s, and except for writing back to the register it must treat the value as if it is UNKNOWN. Software must use an SBOP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also Read-As-One (RAO), Should-Be-One-or-Preserved (SBOP), UNKNOWN.

UNK/SBZP
Hardware must implement the bit as Read-As-Zero, and must ignore writes to the field.
Software must not rely on the field reading as all 0s, and except for writing back to the register must treat the value as if it is UNKNOWN. Software must use an SBZP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also Read-As-Zero (RAZ), Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

UNKNOWN
An UNKNOWN value does not contain valid data, and can vary from implementation to implementation. An UNKNOWN value must not return information that cannot be accessed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE, are not CONSTRAINED UNPREDICTABLE, and do not return UNKNOWN values.

An UNKNOWN value can vary from moment to moment, and instruction to instruction, unless it has previously been assigned, other than at reset, to one of the following registers:
• Any of the general-purpose registers.
• Any of the Advanced SIMD and floating-point registers.
• Any of the Scalable Vector Extension registers.
• Any of the PSTATE N, Z, C, or V flags.

An UNKNOWN value must not be documented or promoted as having a defined value or effect.

In body text, the term UNKNOWN is shown in SMALL CAPITALS.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED, UNK, UNPREDICTABLE.

Unpacked access
A memory access that is performed as a result of a load or store instruction for which the vector element size is larger than the memory element size.

Unpredicated instruction
An SVE instruction that does not have a Governing predicate operand and implicitly treats all other vector and predicate elements as Active.

UNPREDICTABLE
Means the behavior cannot be relied upon. UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

An instruction that is UNPREDICTABLE can be implemented as UNDEFINED.

Execution at Non-secure EL1 or EL0 of an instruction that is UNPREDICTABLE can be implemented as generating a trap exception that is taken to EL2, provided that at least one instruction that is not UNPREDICTABLE and is not CONSTRAINED UNPREDICTABLE causes a trap exception that is taken to EL2.

In body text, the term UNPREDICTABLE is shown in SMALL CAPITALS.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED.

VA
See Virtual address (VA).

VFP
In Armv7, an extension to the Arm architecture, that provides single-precision and double-precision floating-point arithmetic.

Vector
A one-dimensional array of vector elements of the same size and data type.
Vector element
Individual subdivisions of a vector register that can be 8, 16, 32, 64 or 128 bits in size. The vector element size and data type is specified independently by each instruction.

Virtual address (VA)
An address generated by an Arm PE. This means it is an address that might be held in the program counter of the PE. For a PMSA implementation, the virtual address is identical to the physical address.

See also Intermediate physical address (IPA), Physical address (PA).

Watchpoint
A debug event triggered by an access to memory, specified in terms of the address of the location in memory being accessed.

Way
See Cache way.

Wl
Writes Ignored. In a register that software can write to, a WI attribute applied to a bit or field indicates that the bit or field ignores the value written by software and retains the value it had before that write.

See also RAO/WI, RAZ/WI, RES0, RES1.

Word
A 32-bit data item. Words are normally word-aligned in Arm systems.

Word-aligned
Means that the address is divisible by 4.

Write-allocate cache
A cache in which a cache miss on storing data causes a cache line to be allocated into the cache.

Write-back cache
A cache in which when a cache hit occurs on a store access, the data is only written to the cache. Data in the cache can therefore be more up-to-date than data in main memory. Any such data is written back to main memory when the cache line is cleaned or reallocated. Another common term for a write-back cache is a copy-back cache.

Write-through cache
A cache in which when a cache hit occurs on a store access, the data is written both to the cache and to main memory. This is normally done via a write buffer, to avoid slowing down the PE.

Write buffer
A block of high-speed memory that optimizes stores to main memory.

Zeroing predication
When a predicated instruction specifies zeroing predication, the Inactive elements of the destination register are set to zero.